Principia Softwarica: The Plan 9 Windowing System `rio`

version 1.0

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike

June 30, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1 Introduction **10**

1.1 Motivations . . . 10
1.2 The Plan 9 windowing system: `rio` . . . 11
1.3 Other windowing systems . . . 12
1.4 Getting started . . . 15
1.5 Requirements . . . 16
1.6 About this document . . . 17
1.7 Copyright . . . 17
1.8 Acknowledgments . . . 17

2 Overview **18**

2.1 Windowing system principles . . . 18
2.1.1 The window . . . 20
2.1.2 Desktop system . . . 20
2.1.3 Display server . . . 21
2.1.4 Stacking, tiling, and compositing . . . 21
2.1.5 Window compositor . . . 22
2.1.6 Window manager . . . 22
2.1.7 Input focus . . . 22
2.1.8 Terminal emulator . . . 23
2.1.9 Windowing system API . . . 24
2.1.10 Window applications versus graphical applications . . . 26
2.2 `rio` interfaces . . . 27
2.2.1 Command-line interface . . . 28
2.2.2 Graphical user interface . . . 28
2.2.3 Filesystem interface . . . 30
2.3 `hellorio.c` . . . 35
2.3.1 Skeleton and output code . . . 35
2.3.2 Input code . . . 37
2.3.3 The event loop . . . 39
2.4 Code organization . . . 40
2.5 Software architecture . . . 40
2.5.1 Library and header dependencies . . . 41
2.5.2 Processes, procs, and threads relationships . . . 42
2.6 Book structure . . . 45

3 Core Data Structures **47**

3.1 Device handles . . . 47
3.1.1 Output device: `display` and `view` . . . 47
3.1.2 Input devices: `mousectl` and `keyboardctl` . . . 47

3.2 The `desktop` base layer . . . 48
3.3 Windows . . . 48
3.3.1 The `Window` structure . . . 48
3.3.2 All `windows` . . . 51
3.3.3 The current window: `input` . . . 51
3.3.4 Graphical windows . . . 51
3.3.5 Textual windows . . . 51
3.4 Filesystem server . . . 52
3.4.1 Server state: `Filsys` and `filsys` . . . 53
3.4.2 File state: `Fid` . . . 53
3.4.3 Worker request on a `Fid`: `Xfid` . . . 55
3.4.4 9P callbacks: `fcall` . . . 56
3.5 Putting it together: `rio`'s data structures . . . 56

4 `main()` 58
4.1 `threadmain()` skeleton . . . 58
4.2 Graphics initialization . . . 59
4.3 Mouse initialization . . . 60
4.4 Keyboard initialization . . . 60
4.5 Channels creation . . . 60
4.6 Threads creation . . . 61
4.7 Filesystem server initialization . . . 61
4.7.1 `filsysinit()` . . . 61
4.7.2 Worker allocator: `xfidinit()` . . . 62

5 Procs and Threads 64
5.1 Keyboard thread . . . 64
5.2 Mouse thread . . . 65
5.2.1 Application mouse events . . . 67
5.2.2 Windowing system mouse events . . . 68
5.3 Window threads: `winctl()` . . . 71
5.3.1 Keyboard events listening . . . 72
5.3.2 Mouse events listening . . . 73
5.3.3 Control events listening . . . 74
5.4 Filesystem server proc . . . 75
5.4.1 Reading and dispatching: `filsysproc()` . . . 75
5.4.2 Protocol handshake: `filsysversion()` . . . 76
5.5 `Xfid` worker pool . . . 77
5.5.1 Allocator thread: `xfidallocthread()` . . . 77
5.5.2 Worker threads: `xfidctl()` . . . 79

6 Cursors 81
6.1 Cursor graphics . . . 81
6.1.1 Classic cursors . . . 81
6.1.2 Border and corner cursors . . . 83
6.2 Setting the cursor . . . 85
6.2.1 Low-level setter: `riosetcursor()` . . . 85
6.2.2 Cursor dispatch: `cornercursor()` . . . 85
6.2.3 Window cursor: `wsetcursor()` . . . 86

7 Window Manager **88**
7.1 Overview . . . 88
7.2 Right-click system menu . . . 88
7.3 Window borders click . . . 90
7.4 Window control message: `Wctlmesg` . . . 91
7.5 Window creation . . . 92
7.5.1 Window thread creation: `new()` . . . 92
7.5.2 `Window` allocation: `wmk()` . . . 94
7.5.3 Window process creation: `winshell()` . . . 97
7.5.4 Namespace adjustments: `filsysmount()` . . . 99
7.5.5 Public layer: `wsetname()` . . . 100
7.5.6 Mouse action `sweep()` . . . 101
7.5.7 Trace of a new window creation . . . 103
7.6 Window focus . . . 104
7.7 Window deletion . . . 105
7.7.1 Menu handler: `delete()` . . . 106
7.7.2 Timeout for slow clients: `deletetimeoutproc()` and `deletethread()` . . . 107
7.7.3 Two-phase teardown: `Deleted` and `Exited` . . . 108
7.7.4 Mouse action `pointto()` . . . 110
7.8 Window move . . . 111
7.8.1 Menu handler: `move()` . . . 111
7.8.2 Mouse action `drag()` . . . 111
7.9 Window resize . . . 113
7.9.1 Menu handler: `resize()` . . . 113
7.9.2 Mouse action `bandsize()` . . . 115
7.10 Window visibility . . . 118
7.10.1 Menu handler: `hide()` . . . 118
7.10.2 `hidden` windows . . . 118
7.10.3 Menu handler: `unhide()` . . . 119

8 Filesystem Server **121**
8.1 Clients, server, and the kernel in between . . . 121
8.2 Additional data structures . . . 121
8.2.1 Qid file identification: `Qxxx` . . . 121
8.2.2 Directory table: `Dirtab` and `dirtab` . . . 122
8.3 Replying to the client: `filsysrespond()` . . . 123
8.4 `attach` . . . 125
8.4.1 `filsysattach()` . . . 126
8.4.2 `xfidattach()` . . . 127
8.5 `walk` . . . 128
8.5.1 `filsyswalk()` . . . 128
8.5.2 Cloning `fid` . . . 130
8.5.3 `'..'` . . . 130
8.5.4 Error management . . . 131
8.6 `open` . . . 131
8.6.1 `filsysopen()` . . . 132
8.6.2 `xfidopen()` . . . 133
8.7 `clunk` (and close) . . . 133
8.7.1 `filsysclunk()` . . . 133
8.7.2 `xfidclose()` . . . 134

8.8 read . . . 134
8.8.1 filsysread() . . . 135
8.8.2 Reading a directory . . . 135
8.8.3 xfidread() . . . 136
8.9 write . . . 136
8.9.1 filsyswrite() . . . 137
8.9.2 xfidwrite() . . . 137
8.10 stat . . . 137
8.10.1 filsysstat() . . . 138
8.11 Forbidden operations . . . 139

9 Virtual Devices **141**
9.1 Device virtualization across windowing systems . . . 141
9.2 /mnt/wsys/mouse . . . 141
9.2.1 Reading /mnt/wsys/mouse: file server side . . . 142
9.2.2 Writing /mnt/wsys/mouse . . . 144
9.2.3 Trace of a mouse click . . . 144
9.3 /mnt/wsys/cons . . . 146
9.3.1 Reading /mnt/wsys/cons: file server side . . . 147
9.3.2 Writing /mnt/wsys/cons: file server side . . . 149
9.3.3 Bytes versus runes and partial runes . . . 150
9.3.4 Trace of a key press . . . 151
9.4 /mnt/wsys/consctl . . . 152
9.5 /mnt/wsys/cursor . . . 153
9.6 /dev/draw/ and /mnt/wsys/winname . . . 154
9.6.1 Trace of a drawing operation . . . 154
9.6.2 /mnt/wsys/winname . . . 158

10 Graphical Windows **159**
10.1 Graphical window setup . . . 159
10.1.1 initdraw() . . . 159
10.1.2 initmouse(), mouse-open mode . . . 159
10.1.3 initkeyboard(), raw-access mode . . . 160
10.2 Mouse events . . . 161
10.2.1 Mouse state queue . . . 161
10.2.2 Reading /mnt/wsys/mouse: window thread side . . . 162
10.3 Keyboard events . . . 164
10.3.1 Raw keys queue . . . 164
10.3.2 Reading /mnt/wsys/cons: raw mode . . . 164
10.4 Resize events . . . 167
10.5 Graphical window teardown . . . 168
10.6 /mnt/wsys/window . . . 169

11 Textual Windows **172**
11.1 Overview . . . 172
11.2 Textual window creation . . . 173
11.2.1 Scrollbar . . . 174
11.2.2 Frame . . . 174
11.2.3 Frame colors . . . 175
11.3 Content modification . . . 176

11.3.1 Inserting runes: `winsert()` . . . 176
11.3.2 Growing array . . . 177
11.4 Content rendering . . . 178
11.4.1 Updating the visible text: `frinsert()` . . . 179
11.4.2 Keeping the cursor visible: `wshow()` . . . 179
11.4.3 Drawing the scrollbar: `wscrdraw()` . . . 180
11.4.4 Moving the frame origin . . . 182
11.4.5 Selecting . . . 185
11.4.6 Repainting . . . 186
11.5 Keyboard events . . . 186
11.5.1 Text input queue . . . 186
11.5.2 Reading `/mnt/wsys/cons`: cooked mode . . . 187
11.5.3 Navigation keys . . . 188
11.5.4 Special keys . . . 191
11.6 Application output events . . . 195
11.6.1 Writing `/mnt/wsys/cons`: window thread side . . . 195
11.7 Mouse events . . . 197
11.7.1 Middle click menu . . . 198
11.7.2 Other clicks . . . 199
11.8 Automatic scrolling mode . . . 200
11.9 Scroll bar interaction . . . 200
11.10 Resize . . . 205
11.11 `/mnt/wsys/text` . . . 205

12 Windowing System Files **207**
12.1 Overview . . . 207
12.2 `/mnt/wsys/winid` . . . 208
12.3 `/mnt/wsys/label` . . . 208
12.4 `/mnt/wsys/screen` . . . 209
12.5 `/mnt/wsys/wsys/<windid>/` . . . 209
12.6 `/mnt/wsys/wctl` . . . 211
12.6.1 Reading `/mnt/wsys/wctl` . . . 212
12.6.2 Writing `/mnt/wsys/wctl` (controlling windows) . . . 215

13 Advanced Topics **219**
13.1 Notes (signals) . . . 219
13.2 Recursive `rio` . . . 220
13.3 Timer . . . 222
13.4 Command-line options . . . 225
13.4.1 Automatic scrolling: `rio -s` . . . 225
13.4.2 Initial command: `rio -i` . . . 226
13.4.3 Fake keyboard input: `rio -k` . . . 226
13.4.4 Font selection: `rio -f` . . . 229
13.5 Advanced terminal editing . . . 229
13.5.1 Snarf . . . 229
13.5.2 Plumb . . . 234
13.5.3 Auto complete . . . 246
13.5.4 Word selection . . . 252
13.6 External access . . . 256
13.6.1 External mount: `/srv/rio.<user>.<pid>` . . . 256

13.6.2 Command-line control: `/srv/riowctl.<user>.<pid>` . . . 258
13.7 Advanced fileserver features . . . 266
13.7.1 Flushing . . . 266
13.7.2 Authentification . . . 269
13.8 Additional control messages . . . 270
13.8.1 Holding mode . . . 270
13.8.2 Waking the window thread: `Wakeup` . . . 272
13.9 Security . . . 272

14 Conclusion **274**
14.1 Patterns and techniques . . . 274
14.2 Connections to other books . . . 276
14.3 Beyond the Plan 9 windowing system . . . 276

A Debugging **278**

B Error Management **280**
B.1 Error codes . . . 280
B.2 `error()` . . . 282

C Utilities **283**
C.1 Memory management . . . 283
C.2 Arithmetic . . . 284
C.3 Unicode . . . 284

D The Frame Library **287**
D.1 The `Frame` widget . . . 287
D.1.1 `Frame` . . . 287
D.1.2 `frinit()` . . . 288
D.1.3 Frame colors . . . 289
D.1.4 Frame tick . . . 289
D.2 Boxes, strings, and coordinates . . . 291
D.2.1 Frame boxes . . . 291
D.2.2 Box string storage . . . 294
D.2.3 Frame rune position, point, and box number . . . 294
D.3 Selection and repainting . . . 299
D.3.1 Frame selection . . . 299
D.3.2 Repainting . . . 302
D.4 Incremental rendering . . . 303
D.4.1 Incremental delete: `frdelete()` . . . 314
D.4.2 Drawing the tick: `frtick()` . . . 316
D.4.3 Drawing the text and the selection: `frdrawsel()` . . . 317

E Windowing System Applications **320**
E.1 `lens` . . . 320
E.2 `statusbar` . . . 324
E.3 `winwatch` . . . 327
E.4 `window` . . . 329
E.5 `wloc` . . . 331
E.6 `label` . . . 332

F Extra Code 333
F.1 rio/ . . . 333
F.1.1 rio/dat.h . . . 333
F.1.2 rio/fns.h . . . 335
F.1.3 rio/globals.c . . . 338
F.1.4 rio/rio.c . . . 338
F.1.5 rio/thread_mouse.c . . . 339
F.1.6 rio/thread_keyboard.c . . . 339
F.1.7 rio/threads_window.c . . . 339
F.1.8 rio/threads_worker.c . . . 340
F.1.9 rio/threads_misc.c . . . 340
F.1.10 rio/wm.c . . . 340
F.1.11 rio/data.c . . . 341
F.1.12 rio/cursor.c . . . 342
F.1.13 rio/wind.c . . . 342
F.1.14 rio/processes_winshell.c . . . 343
F.1.15 rio/terminal.c . . . 343
F.1.16 rio/snarf.c . . . 345
F.1.17 rio/graphical_window.c . . . 345
F.1.18 rio/9p.c . . . 346
F.1.19 rio/proc_fileserver.c . . . 346
F.1.20 rio/fsys.c . . . 346
F.1.21 rio/xfid.c . . . 347
F.1.22 rio/scrl.c . . . 348
F.1.23 rio/time.c . . . 349
F.1.24 rio/wctl.c . . . 349
F.1.25 rio/error.c . . . 350
F.1.26 rio/util.c . . . 350
F.2 apps/ . . . 350
F.2.1 apps/lens.c . . . 350
F.2.2 apps/statusbar.c . . . 352
F.2.3 apps/winwatch.c . . . 356
F.3 include/ . . . 361
F.3.1 include/complete.h . . . 361
F.3.2 include/frame.h . . . 361
F.3.3 include/plumb.h . . . 362
F.4 libcomplete/ . . . 363
F.4.1 libcomplete/complete.c . . . 363
F.5 libframe/ . . . 363
F.5.1 libframe/frbox.c . . . 363
F.5.2 libframe/frdelete.c . . . 364
F.5.3 libframe/frdraw.c . . . 364
F.5.4 libframe/frinit.c . . . 364
F.5.5 libframe/frinsert.c . . . 365
F.5.6 libframe/frptofchar.c . . . 365
F.5.7 libframe/frselect.c . . . 365
F.5.8 libframe/frstr.c . . . 366
F.5.9 libframe/frutil.c . . . 366
F.6 libplumb/ . . . 366

F.6.1 `libplumb/event.c` . . . 366
F.6.2 `libplumb/mesg.c` . . . 367
F.6.3 `libplumb/plumbsendtext.c` . . . 367

Glossary **368**

Index **369**

References **383**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a windowing system.

1.1 Motivations

Why a windowing system? Because I think you are a better programmer if you fully understand how things work under the hood, and one of the first thing you should see on your screen is a set of windows. The *windowing system* is the program allowing you to create and manipulate those windows.

Windowing systems are usually coupled with a *graphics system* to form a graphical user interface (GUI). GUIs, introduced in the 1970's with the Xerox Alto [TML+79], were a vast improvement over text-based user interfaces, to the point where every mainstream operating systems now come with a GUI (e.g., Microsoft Windows, macOS, or Linux with X Window and Wayland).

A windowing system relies on a graphics system to render the graphics of a window on a specific rectangular surface of the screen. However, a window is not just a surface; it is also a process. Thus, a windowing system manages not only a set of surfaces, but also a set of processes. This is similar to what a kernel does. Moreover, just like the kernel manages the CPU and memory and virtualizes those resources shared among multiple processes, a windowing system manages the screen and input devices (e.g., the mouse, the keyboard) and virtualizes those resources shared among multiple windows. The windowing system is a natural extension of the kernel. In fact, the need for multiple processes and a multi-tasking kernel is less obvious without a windowing system. Linux offers virtual consoles where the user can launch independent commands, but those consoles are a poor's man windowing system.

Surprisingly, there is almost no book explaining how a windowing system works, even though there is a myriad of books on kernels. I can cite *The NeWS book: An Introduction to the Network/Extensible Window System* [GRA89], or one chapter of *Computer Graphics, Principles and Practice* [FDFH90] dedicated to user interface software. Books on operating systems usually do not even include a chapter on windowing systems. This is a pity because the windowing system is as important as the kernel for the user.

Here are a few questions I hope this book will answer:

- What is the software architecture of a windowing system? Is the windowing system a regular program? How does it have access to the mouse and the screen? Does it need special privileges from the kernel? How does it cooperate with the kernel?
- How does the windowing system manage multiple windows/processes? How does it communicate with those processes?
- How does the windowing system control access to the screen, a resource used by multiple windows at the same time? How does it cooperate with the graphics system?

- How does the windowing system intercept the drawing operations done by windows to make sure they can not draw in other windows? How is the screen virtualized?
- How does the windowing system handle the mouse device? When are mouse events dispatched to the windows? How does the windowing system decide which window should receive the mouse event?
- How does the introduction of the mouse, graphics, and windows changes the programming model of an application? How can an application react to a mouse event? How can the windowing system itself react to a mouse event?
- How does the windowing system handle the keyboard device? How does it decide which window should receive a keyboard event? How does it deliver a keyboard event to this window?
- How are overlapping windows managed? Where are stored the pixels of a window overlapped by another window? How are those pixels restored on the screen when an overlapped window is exposed back?
- What are the differences among a windowing system, a window manager, a window compositor, a window server, and a desktop system?
- How does a terminal emulator (e.g., `xterm`) work? What are the standard input and output of traditional command-line applications when running under an emulator? How does the emulator offer a backward-compatible environment for those applications?
- What happens when you type `ls` in a terminal emulator? What are the set of programs involved in such a command? What is the trace of such a command through the different layers of the software stack, from the keyboard interrupt to the display of text glyphs on the screen in the appropriate window?
- What happens when the user type `^C` in a terminal emulator to interrupt a process? Which process gets the interrupt? How is it delivered to the process?

1.2 The Plan 9 windowing system: `rio`

I will explain in this book the code of the Plan 9 windowing system `rio` [Pik00b][1], which contains about 8 800 lines of code (LOC). `rio` is written entirely in C.

In most operating systems (e.g., macOS, Microsoft Windows), the windowing system is *strongly* coupled with the graphics system. In Plan 9, the windowing system `rio`, and the graphics system, called `draw`, are clearly separated; `rio` is a user-space program that relies on `draw`, which is implemented as a device in the kernel (for more information on `draw`, read the Graphics book [Pad16c]). In Plan 9, you can run graphical applications with or without `rio`. In fact, `rio` itself is just a graphical application.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. `rio` comes from a series of windowing systems designed by Rob Pike: $8\frac{1}{2}$ [Pik91], the direct ancestor of `rio`, running also under Plan 9; the "Concurrent Window System" [Pik89], programmed in the Newsqueak language; and `mpx`, the windowing system of the Blit [Pik83a] machine. `mpx` was the first windowing system for Unix, and actually one of the first windowing system back in 1982[2]. It was created even before the Macintosh, X Window, and Microsoft Windows.

There is another reason `rio` rewards study: it is a small masterpiece of *concurrent* programming. The lineage above is no accident—the "Concurrent Window System" was written in Newsqueak precisely to explore communicating processes—and `rio` inherits that DNA. Built from Plan 9's `libthread` *procs*, *threads*, and

[1] See http://plan9.bell-labs.com/magic/man2html/4/rio for its manual page.

[2] See https://www.youtube.com/watch?v=emh22gT5e9k for an historical demo of the Blit and `mpx`, which has a user interface almost identical to `rio`.

channels, it is a parade of concurrency design patterns, from active objects and worker pools to channels that themselves carry channels. I point them out as they appear through the book, and collect them in Section 14.1.

Like many other services in Plan 9, some of the `rio` services are accessible through files. Indeed, `rio` is a graphical application *and* a filesystem. To understand why a part of `rio` is implemented as a filesystem, you need (1) to have a general idea on how to implement a windowing system, and (2) be familiar with some of the advanced features of the Plan 9 kernel.

Regarding the first point, at a high level, a windowing system is a program that uses the mouse (via `/dev/mouse` under Plan 9), the keyboard (via `/dev/cons`), and the screen (via `/dev/draw`). Now, an application running in a window is not different; such an application also wants to use the mouse, the keyboard, and the screen. Thus, a windowing system can be seen simply as a *multiplexer*; the windowing system can use the *physical devices* (managed by the kernel) and serve *virtual devices* to the multiple windows running under it[3].

Regarding the second point, the Plan 9 kernel has a few original features that makes it easy to implement virtual devices served by programs in user space. Those features are the *per-process namespace*, the *union-mount*, and the *file-server protocol 9P* (see the KERNEL book [Pad14] for more information on those features, or the two Plan 9 articles [PPD+95, PPT+93], which contain both good introductions to those features). With `rio`, the virtual devices are accessible under `/mnt/wsys/` (e.g., `/mnt/wsys/mouse`, `/mnt/wsys/cons`), but also under `/dev/` (e.g., `/dev/mouse`, `/dev/cons`), thanks to the union-mount feature. Thanks to the per-process namespace, the applications running under `rio` see a different `/dev/mouse`, `/dev/cons`, and could see a different `/dev/draw`[4]. Finally, thanks to 9P, all those virtual device files can be served by a single user-space program: `rio`.

The consequence of this design is worth spelling out. Because `rio` serves its virtual devices through the same filesystem protocol the kernel uses for its real devices, any program written to use `/dev/mouse`, `/dev/cons`, and `/dev/draw` will run unchanged whether it is launched from the bare console or from inside a `rio` window. Other windowing systems need a separate library (Xlib, the Wayland client, the Win32 API, Cocoa/AppKit) with its own connection setup, its own event loop, and its own mental model. `rio` needs none of that because the "API" is just file I/O on the same path names. This is the payoff of Plan 9's everything-is-a-file philosophy taken to its logical conclusion: a windowing system with no special client library.

Another nice side effect of the multiplexer approach used by `rio` is that `rio` can run under itself (see Section 13.2). This is useful for development and debugging purposes. Moreover, because you can export filesystems through the network in Plan 9, `rio` is also a networked windowing system, similar to X Window (even though the code of `rio` does not include a single line of code related to networking). Thus, in Plan 9, programs running on one machine can have their window displayed on another machine.

1.3 Other windowing systems

Here are a few windowing systems that I considered for this book, but which I ultimately discarded:

- Xorg[5], which I mentioned already in the GRAPHICS book [Pad16c], is the most popular open-source implementation of the X Window System [SG86], a windowing system (and a graphics system) designed in the 1980's at MIT. However, its codebase is enormous. In fact, the whole system is divided in hundreds of repositories[6] to better handle its complexity. One of this repository, `xserver`[7], which contains the code of the display server, has already more than 500 000 LOC. This does not even include the code of the window manager (e.g., `twm` with 17 000 LOC), the terminal emulator (e.g., `xterm` with 80 000 LOC), or

[3]For more information, see Section 2.1.9 and especially Figure 2.3.

[4]For `/dev/draw`, the `draw` device can already multiplex the screen among multiple clients (in `/dev/draw/1/`, `/dev/draw/2`, etc). There is no need for a virtual `/dev/draw`. However, the ancestor of `rio`, $8\frac{1}{2}$ [Pik91], was serving a virtual `/dev/draw` device file, which was more elegant but also more inefficient. For more information, see Section 2.2.3.

[5]http://xorg.freedesktop.org

[6]https://cgit.freedesktop.org/xorg

[7]https://cgit.freedesktop.org/xorg/xserver/

the libraries required by clients to communicate with the display server (e.g., Xlib with 150 000 LOC). Xorg, in total, contains more than two orders of magnitude more code than `rio`.

Part of the reason for the enormous size of Xorg is that Xorg supports many graphic cards, many monitors, many input devices, and many extensions (e.g., 3D operations). Another reason is that X Window is an old program; programmers extended X Window for more than 40 years now. Programmers added many extensions while still being forced to remain backward compatible with applications designed in the 1980's.

X Window defines a communication protocol, X11, for a networked clients/server architecture. Client applications must use *sockets* to connect to the display server. Unfortunately, the set of mechanisms used by clients to interact with the screen, mouse, or keyboard is quite different from the one offered by the kernel, for instance, the simple opening of files in `/dev/` such as `/dev/mouse`. In some sense, X Window *masks* the features of the underlying kernel. On the opposite, `rio` is *transparent* [Pik88] and instead generalizes the services offered by the kernel, for instance, with the virtual device file `/dev/mouse`. Of course, the use of sockets in X Window allows client applications to display their result on another machine on the network. However, this is also possible with `rio`, for free, thanks to the 9P protocol (see the NETWORK book [Pad16d]).

Finally, the graphics and windowing systems of Xorg are strongly coupled; this coupling makes the whole system harder to understand than `rio` and `draw`, which we can study separately.

- Wayland[8] is a protocol, similar to X11, specifying the communication between a display server, called a Wayland *compositor*, and a set of local clients. Weston[9] is a reference implementation of a Wayland compositor. Wayland and Weston grew out of the frustation of some developers of Xorg with the complexity of X Window, as well as the difficulty for X Window to support the modern needs of a windowing system: transluscent windows, drop shadows on the window's border as in macOS Aqua, fancy window-switcher such as macOS Expose, etc.

  Fortunately, during the last twenty years, lots of the code of Xorg got gradually moved out of the display server and put either in the Linux kernel (e.g., the resolution setting of the screen, called KMS for kernel mode setting, or the ability to interact directly with the graphics hardware, called DRM for direct rendering manager), or in external libraries (e.g., Cairo for an advanced drawing API, or `libinput` for a generic interface to the input devices). What remains in Xorg is an old drawing API, the ability to have remote applications, and a display server that is backward compatible with old applications. The developers of Wayland used this opportunity to redesign from scratch a modern windowing system, while reusing lots of the code that was now outside Xorg.

  There are many differences between Wayland and X11. For instance, Wayland does not specify any drawing API. Instead, it assumes the clients do their own graphics rendering by using libraries such as Cairo on locally-shared image buffers. Weston then just uses those shared buffers and composes them together (hence the use of the word "compositor"), while possibly applying effects during the image composition such as transluscence. The use of locally-shared buffers means that Wayland does not support remote applications. Fortunately, most users now run and display their applications on the same machine.

  The code of Wayland and Weston is far smaller than Xorg: 120 000 LOC (not including the tests). However, this is still one order of magnitude more code than `rio`. Moreover, Weston relies on many libraries (e.g., Cairo, `libinput`), as well as lots of code and subsystems of the Linux graphics stack (e.g., KMS, DRM, GEM, fbdev, evdev); this would add lots of code to explain.

- Nano-X[10] (previously known as MicroWindows) is a windowing system and graphics system designed originally for small devices such as personal digital assistants (PDAs, the ancestors of mobile phones). It

[8] https://wayland.freedesktop.org/

[9] https://cgit.freedesktop.org/wayland/weston/

[10] http://www.microwindows.org/

started as a fork of Mini-X, a graphics system for MINIX. Both Mini-X and Nano-X are modeled after X Window, and offer an API similar to Xlib. Nano-X added a clients/server architecture to Mini-X, as well as a window manager, to become a full windowing system.

  Nano-X is highly portable, with support for many machines (e.g., x86 desktops, MIPS machines, ARM embedded devices). Moreover, Nano-X does not require any external graphics library; it just requires an access to the framebuffer from the Linux kernel. It is far smaller than Xorg: 80 000 LOC (not including the tests, application demos, the Win32 API, and the fonts). However, this is still bigger than the code of `draw` and `rio` combined.

- Oberon[11] is an operating system designed by Niklaus Wirth and Jürg Gutknecht around 1990 and described in full, hardware included, in their book *Project Oberon* [WG92]. The *entire* system—a compiler, the operating system, and the windowing environment, with the hardware design on the side—is famously documented end to end in that single book. The windowing part proper, the `Display`, `Viewers`, `MenuViewers`, and `TextFrames` modules, is well under 2 000 lines of Oberon, a tiny fraction of `rio`.

  That compactness, however, comes from giving up what makes a windowing system interesting for this book: Oberon is not a clients/server design at all. It runs in a single address space, in a single language, and its "windows" are tiled *viewers* cooperatively scheduled inside one process, rather than independent programs multiplexed over a protocol. Oberon is a beautiful lesson in minimalism, but it sidesteps exactly the multiplexing and isolation problems that this book is about.

- Haiku[12] is an open-source recreation of BeOS, the operating system developed in the late 1990's by Be Inc. for its multimedia workstations. Its windowing system, the `app_server`, is however much larger than `rio`: around 80 000 LOC for the server alone (the client-side "Interface Kit" is separate). It is also deeply entangled with the rest of Haiku—it relies on the BeAPI that every client links against, and on Haiku's own kernel and messaging primitives.

- Orbital[13] is the windowing system of Redox OS, an operating system written from scratch in Rust and organized, like Plan 9, around the principle that everything is a file. Orbital is a small clients/server display server: a client creates a window and receives its input events by opening an `orbital:` *scheme*, Redox's analogue of a Plan 9 file server, much as `rio` clients go through files in `/dev/`. Unlike `rio`, though, Orbital is local-only: Redox schemes are a local IPC mechanism, with no network-transparent transport like 9P, so a client cannot display on a remote machine for free.

  At around 4 000 LOC for the server, plus another 4 800 LOC for `orbclient`, the client-side library that applications link against, Orbital is in the same size class as `rio`, and it is arguably the modern system closest in spirit to this book's subject. I set it aside mainly because studying it would require first explaining the Rust language and its compiler, a huge undertaking that would extend Principia Softwarica by a lot.

Figure 1.1 presents a timeline of major windowing systems. I think `rio` represents the best compromise for this book: it implements the essential features of a windowing system while still having a small and understandable codebase (8 800 LOC).

This timeline differs from the lineage diagrams in the other Principia Softwarica books. There, most of the important systems—compilers, shells, editors, kernels—are open-source, and many descend directly from research or UNIX code one can read. Windowing systems are the opposite: the field was driven by commercial, closed-source products (the Macintosh, Windows, NeXTSTEP, BeOS, the SGI and Sun desktops), and only a handful of open-source windowing systems (X Window, and much later Wayland) can be freely studied. This scarcity of readable code is one more reason `rio` makes a good subject for this book.

[11] https://projectoberon.net/
[12] https://www.haiku-os.org/
[13] https://gitlab.redox-os.org/redox-os/orbital

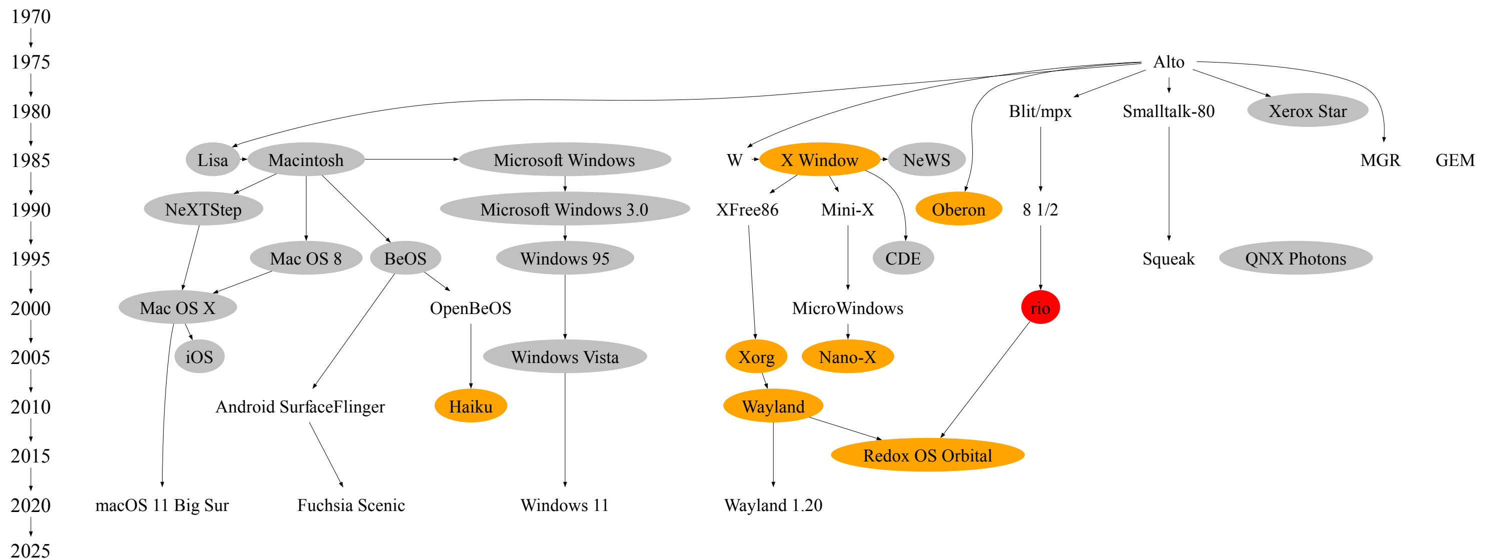


Figure 1.1: Windowing systems timeline

Figure 1.2: The screen, just after `rio` started and the user right-clicked.

1.4 Getting started

To play with `rio`, you will first need to install the Plan 9 fork used in Principia Softwarica (see `https://www.principia-softwarica.org/getting-started.html`). Once installed, you can test `rio` under Plan 9 by executing the following commands:

```
1   $ bind -a '#v' /dev
2   $ vga -l 640x480x8
3   # screen should change layout
4   $ bind -a '#i' /dev
5   $ rio
```

Then, if you right-click with the mouse somewhere on the screen, you should see graphics similar to the one in Figure 1.2.

Line 1 through 4, above, install the graphics system of Plan 9 (`draw`) and configure it to run at the `640x480x8` resolution (See the Graphics book [Pad16c] for more information on `draw`). Line 5 then executes `rio`, which should take over the screen to create the graphics shown in Figure 1.2.

Note that plan9port[14] includes a program called `rio`, but it is not a port of the real Plan 9 `rio`—it is an X11 window manager (based on `9wm`) that merely imitates `rio`'s user interface. The real `rio`, described in this book, is deeply tied to Plan 9's kernel (`/dev/cons`, `/dev/draw`, 9P, per-process namespaces) and can only run under a full Plan 9 system.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. Moreover, because `rio` relies on many advanced features of the Plan 9 kernel, and builds upon the Plan 9 graphics system `draw`, I strongly suggest you to read the Kernel book [Pad14] and Graphics book [Pad16c] before reading this book. Note that `rio` is implemented as a filesystem in user space, and uses the protocol 9P to communicate with the kernel. Thus, it can also be useful to read the Network book [Pad16d], which describes 9P. In the same way, I also suggest you to read the Libcore book [Pad16a], which explains the thread library, which is heavily used by `rio`.

Book	Concepts	Device Files	Codes	Headers
Graphics book	display server, drawing API, shared image, overlapping layers	`/dev/draw` `/dev/winname` `/dev/vgactl`	`#i #v`	`draw.h window.h mouse.h cursor.h keyboard.h`
Kernel book	filesystem, device, pipe, console, per-process namespace, union-mount, shared memory	`/dev/cons` `/dev/consctl` `/dev/mouse` `/dev/pipes/`	`#c #m #|`	`syscall.h`
Libcore book	channel, proc, thread, message, message queue			`libc.h thread.h`
Network book	remote procedure call (RPC), filesystem in user space, 9P protocol	`/srv`	`#s`	`fcall.h`

Table 1.1: Principia Softwarica books related to the Windows book.

Table 1.1 presents the list of related Principia Softwarica books, as well as the concepts, devices, and header files used by `rio` and introduced by those books. The most important book in Table 1.1 is the Graphics book. Regarding the three other books, you can probably understand most of the code in the following chapters without reading those books if you read at least *Plan 9 from Bell Labs* [PPD+95], as well as *The Use of Name Spaces in Plan 9* [PPT+93]; those two articles introduce many of the concepts listed in Table 1.1.

Read the other way around, Table 1.1 shows that `rio` sits at the crossroads of almost every part of Plan 9: it builds at once on the graphics system, on the filesystem and namespace machinery of the kernel, on the thread library, and on the 9P protocol. In a sense `rio` is a culmination of the system—a single user-space program that ties together the abstractions introduced by all the other books.

As I said in Section 1.1, there are very few books explaining the concepts, theories, and algorithms used in windowing systems. I can cite *The NeWS book* [GRA89], *Methodology of Window Management* [HDF+86], and one chapter of *Computer Graphics, Principles and Practice* [FDFH90]. Those books are useful, but they are not mandatory to understand this book.

If, while reading this book, you have specific questions on the interfaces of `rio`, or on the API used by `rio`, you can find answers in certain manual pages. Those pages are located under `docs/man/` in my Plan 9 repository. Here is a list of pages relevant to `rio` and a short description of their content:

[14] https://9fans.github.io/plan9port/

- `1/rio`: the command-line and graphical user interface of `rio`
- `4/rio`: the filesystem interface of `rio`
- `2/draw`: the `draw.h` API
- `2/window`: the `window.h` API
- `2/keyboard`: the `keyboard.h` API
- `2/mouse`: the `mouse.h` API
- `2/thread`: the thread and channel library
- `5/0intro`: the 9P protocol

Finally, the `windows/docs/` directory in my Plan 9 repository contains documents describing either `rio` [Pik00b] or ancestors of `rio` [Pik91, Pik89, Pik88, Pik83a]. All those documents are useful to understand some of the design decisions presented in this book.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is made of source code from Plan 9, so those parts are copyright by Plan 9 Foundation. The prose is mine and is licensed under the All Rights Reserved License.

1.8 Acknowledgments

I would like to acknowledge of course the author of `rio`, Rob Pike, who wrote in some sense most of this book.

Chapter 2

Overview

Before showing the source code of `rio` in the following chapters, I first give in this chapter an overview of the general principles of a windowing system. I also describe quickly the graphical user interface of `rio`, as well as its filesystem interface in `/mnt/wsys/`. I also show the code of a toy application running under `rio`. Finally, I define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

2.1 Windowing system principles

A *windowing system* is a program (or a set of programs) with a graphical user interface (GUI) allowing the user to create and manipulate windows. A *window* is a usually rectangular and resizeable surface of the screen containing the GUI of another program. Just like the kernel is a *meta-program*, that is a program allowing the user to run other programs, the windowing system is a *meta-GUI*, that is a graphical user interface allowing the user to run other graphical user interfaces.

In addition to windows, a windowing system traditionally uses *icons*, *menus*, and a *pointer*, or *WIMP* for short. Note that windowing systems are not the only kind of meta-GUIs. For example, the user interface of phones running under iOS or Android do not have any windows. Moreover, the user can use multiple pointers at the same time through his multiple fingers. Those interfaces are called *post-WIMP* interfaces.

A windowing system has many components, which sometimes can even be separate programs. I mentioned them already in Section 1.3: the display server, the window compositor, the window manager, and the terminal emulator. Figure 2.1 presents the relationships between those components. Figure 2.1 shows also four important layers: the hardware at the very bottom, the kernel and windowing system in the middle, and the applications at the top. Moreover, in this book, I divide applications in three different categories:

- *Textual applications*: those are command-line programs, which just read and output text (e.g., `grep`).
- *Graphical applications*: those are programs drawing on the whole screen and using the mouse (e.g., the Doom video game)
- *Window applications*: those are programs drawing and using the mouse inside a window (e.g., the Microsoft minesweeper video game)

A windowing system is mostly concerned with the last category, but it usually offers a way to run in a special environment the other kinds of applications. For instance, the terminal emulator in Figure 2.1 allows to run textual applications.

The following sections will explain the relations between the different components in Figure 2.1 as well as the role of each of those components. I will also quickly describe how those components are implemented in X Window and `rio`. But before, I need to better characterize what is a window.

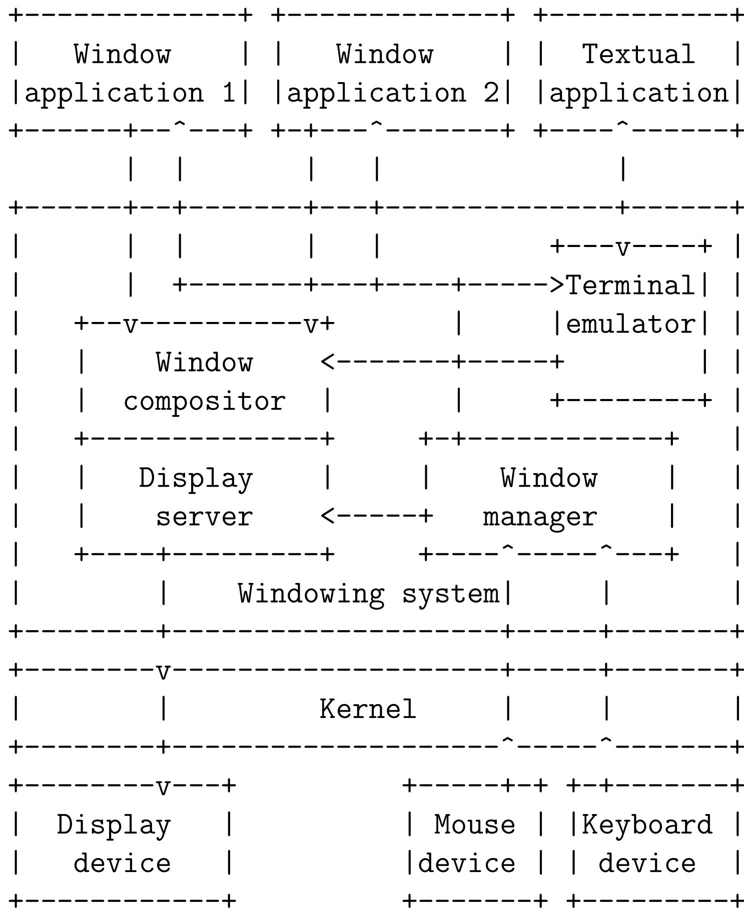


Figure 2.1: Components of a windowing system.

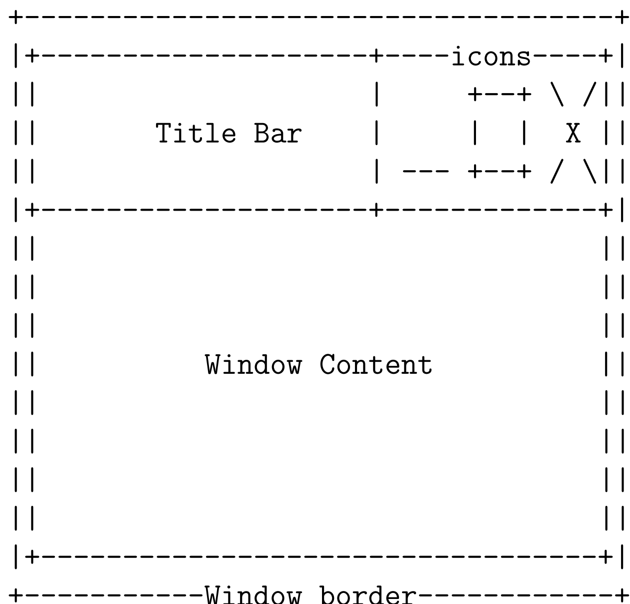


Figure 2.2: Window elements.

2.1.1 The window

A window is multiple things at the same time. First, it is a *delimited surface* of the screen containing the visual *output* of a running program[1]. It is also an *interactive region* of the screen responding to *input* from the mouse; when the mouse hovers inside a window, the underlying program can react. Finally, it is a *container* that can be manipulated from the outside; a window can be moved, resized, closed, etc.

A window surface is made of multiple elements, as shown in Figure 2.2. Here is the list of those elements as well as their functions:

- *Window content*: contains the GUI of the underlying running program
- *Window border*: allows the user to move or resize the window
- *Title bar*: contains usually the name of the program
- *Windowing system icons*: allows the user to close, expand, or hide the window

Only the first element is mandatory; the other elements, forming the *window decoration*, are all optionals. For instance, in `rio`, windows have a border but they have neither a title bar nor icons (see Figure 2.6).

2.1.2 Desktop system

A *desktop system* is a kind of windowing system promoting the following metaphor: a windowing system is like a physical desk in an office. A desktop system usually includes applications and icons mimicking the real-life objects of an office: a garbage can, folders, a clock, a rolodex, an alarm, a calendar, etc. The window is then like a paper on a desk; it can be moved around, or stacked on top of other papers. Each window represents a separate activity.

The Xerox Star [SIKH82] was the first desktop system. Desktop systems are the most popular windowing systems (e.g., Microsoft Windows, macOS), because they offer a familiar interface to the user. On Linux, the

[1]The term "window" is actually a misnomer [Pik83b]. In graphics terminology, such an output is called instead the *viewport*.

desktop systems KDE[2] and GNOME[3] are implemented as a set of applications on top of X Window. Plan 9 does not have a desktop system; `rio` does not use any icon and does not promote an office metaphor. `rio` is not a WIMP, just a WMP.

2.1.3 Display server

The first component of a windowing system is the display server. A *display server* is a graphics system that accepts drawing commands from multiple *clients* via a *communication protocol*, and then translates those commands into instructions to the graphics card. The display server is responsible for all the visual output of all the applications, as well as the visual output of the windowing system itself (e.g., the window decorations, the background image). This is why in Figure 2.1, the applications, window manager, and terminal emulator are all connected to the display server (through the window compositor, which I will explain in Section 2.1.5). A display server uses a clients/server architecture because it needs to serve many clients: the multiple processes corresponding to the multiple windows on the screen.

In X Window (as well as in most windowing systems), the display server is an integral part of the windowing system. Moreover, the communication protocol of X Window, X11, is used not only to carry drawing commands from the clients, but also to relay input events from the devices to the clients. In that case, the display server acts also as a *window server* as it manages all the communications with the windows.

One of the first display server was Rob Pike's Blit terminal (Bell Labs, 1982). X Window (MIT, 1984, Bob Scheifler and Jim Gettys) was the first *networked* display server—its key innovation was separating the server from the clients so they could run on different machines, which is why X can display a remote application's window on a local screen.

In Plan 9, the display server `draw` is outside `rio`, in the kernel, and serves only the drawing commands from the clients. The communication protocol of `draw` is described in the GRAPHICS book [Pad16c] (and involves the `/dev/draw/<n>/data` files).

2.1.4 Stacking, tiling, and compositing

In `rio`, as well as in many other windowing systems, windows can overlap each other. Thus, windows can be *stacked* on top of each other, hiding the pixels of the windows below. In other windowing systems, windows instead are *tiled* automatically next to each other (or hidden completely). Finally, in recent windowing systems, windows can be *composited* with each other; the windowing system composes the images representing the different windows together and can apply special effects, for instance, transluscence or drop shadows as in macOS[4].

In a stacking system, windows live on an abstract Z-axis and can overlap. The one with the highest Z value sits on top and its pixels hide whatever is underneath. X Window, old Microsoft Windows, old macOS, `rio`, and every desktop system most users know work this way. The benefit is that the user can put twenty windows on screen even if only ten fit; hidden windows sit patiently under the visible ones and re-emerge on demand. The cost is that every motion of a window forces the newly *exposed* region to be repainted by whichever program owned it—the origin of X11's `Expose` events.

A tiling system forbids overlap. Every window gets a non-overlapping rectangular region, and the manager redivides the screen whenever a window is created, destroyed, or resized. Plan 9's own $8\frac{1}{2}$—the predecessor to `rio`—could be used as a tiler. Today the idea lives on in keyboard-driven managers like `dwm` or `xmonad`. The benefit is that every window is always fully visible, so the user never forgets where things are; the cost is that a window cannot be bigger than its slice, so workflows that rely on swapping many windows in and out quickly feel cramped.

[2] https://www.kde.org/

[3] https://www.gnome.org/

[4] Another nice effect is to zoom out and tile automatically all the images of the windows, as in macOS Expose, to get a bird's eye view of all the windows.

In a compositing system, every window draws into its own off-screen buffer, and a dedicated compositor combines those buffers into the final framebuffer with effects like alpha-blending, drop shadows, scaling, or 3D transforms. macOS Quartz (2000), Windows DWM (Vista, 2007), and every Wayland compositor (Weston, Sway, Mutter, KWin) take this route. Compositing subsumes stacking and also eliminates `Expose`-style repaint storms: a hidden window's pixels stay in its buffer, so moving another window in front of it costs only a composite pass, not a repaint by the owning program. The cost is (GPU) memory for all those buffers which is why compositing only became practical once GPUs became universal.

2.1.5 Window compositor

A *window compositor* is an optional component of a windowing system (found in most modern windowing systems) allowing windows to be *composited* with each other. As said in the previous section, each window application draws first in an *off-screen image*. Then, the compositor composes all those images together while applying possibly advanced effects such as transluscence or drop shadows. Finally, the composition is sent to the display server, which outputs the result on the screen. This is why in Figure 2.1, the window applications are all connected first to the compositor. The compositor and display server are usually tightly coupled, hence the direct contact between the two respective boxes in Figure 2.1.

In `rio`, which favors a minimalist approach, there is no compositor; each window application is connected directly to the display server (`draw`). That means `rio` does not offer special effects such as transluscence, but it would not be difficult to extend `rio` to include a compositor to support those effects.

2.1.6 Window manager

The *window manager* is the component responsible for all the user inputs to the windowing system: inputs from the mouse, the keyboard, or other devices. Those inputs can lead to the moving, resizing, opening, closing, or hiding of windows, hence the term “window manager”. Indeed, the action of the mouse over the window decorations can trigger the changes to the windows listed above. In fact, the window manager is also responsible for the display of those window decorations. This is why in Figure 2.1, the window manager is connected to the display server,

When the mouse is over the window content, the role of the window manager is then to *relay* the input events to the application. This is why in Figure 2.1, the mouse and keyboard devices are connected to the window manager, which is connected itself to all the window applications (and the terminal emulator) in order to *dispatch* the input to the appropriate window.

The concept of a window manager as a separate program was one of X Window's innovations (1984). Before that—on the Alto, the Blit, the Macintosh—the display server and window manager were always integrated into one program. X's separation let users choose or write their own manager. There are more than 50 different window managers available for X Window, each with different window decorations, different input policies, different menus, etc. The window manager communicates also with the display server, like other clients. However, the window manager is assigned a special role by the display server.

In Weston and Wayland, the display server, compositor, and window manager are all parts of the same program.

With `rio`, there is only one window manager, which is also an integral part of the windowing system. There is only one style of window decoration, kept to a minimum: no title bar, no icon, just a thin blue window border (see Figure 2.6).

2.1.7 Input focus

The previous section introduced the window manager's role as the dispatcher of mouse and keyboard input. Mouse input has an obvious target: the window under the pointer. Keyboard input does not, because the keyboard has no pointer. The windowing system must therefore maintain an explicit notion of *input focus*: a

designated window that receives keystrokes until something changes it. How that "something" is chosen is one of the oldest and most opinionated design questions in the field.

Windowing systems have tried roughly three focus policies:

- *Click-to-focus.* A window only gains focus when you click it, and keeps it until you click another window. This is the default on Microsoft Windows, macOS, and most modern X Window and Wayland desktops. The argument in favour is that focus changes are deliberate: you never lose a keystroke to the wrong window just because the mouse drifted over it.

- *Focus-follows-mouse (sloppy focus).* The window under the pointer receives keyboard input, and focus changes silently every time the pointer crosses a border. This was common on classic X Window and is still used by many tiling window managers (`dwm`, `xmonad`). The argument is speed: moving to a terminal and typing needs no click. The cost is that you can type into the wrong window if the pointer happens to sit somewhere you forgot, and focusing a window without moving the pointer (e.g. via a keyboard shortcut) becomes awkward.

- *Focus-on-pointer (strict pointer focus).* Like sloppy focus but with no memory: the instant the pointer leaves the window, focus is gone, even if you have not entered another. Typing into empty desktop does nothing. This was the original X Window behaviour and is rarely used today because it is hard to keep the pointer inside the window you wanted.

`rio` sits on the click-to-focus side. A left-click on an unfocused window both raises it to the top of the stack and updates the global `input`[51e] pointer—the window that receives keystrokes from the keyboard dispatcher. The focused window keeps the focus until another click changes it or the window is deleted.

2.1.8 Terminal emulator

The last component of a windowing system is the terminal emulator. A *terminal emulator* provides a backward-compatible environment for command-line applications to run inside a window, without having to rewrite and recompile those applications. The emulator has also usually some basic line-editing capabilities such as handling the backspace key or copy-pasting.

The terminal emulator is an optional component of the windowing system that most users never use, but that most programmers can not live without. Indeed, thanks to the emulator, the programmer can run all his classic tools (e.g., shells, compilers, linkers, `grep`) under the windowing system, and even run multiple tools at the same time in different windows. An alternative is to use an integrated development environment (IDE), but IDEs rarely integrate all the tools used by a programmer.

The environment needed by a command-line application under UNIX or Plan 9 is minimal: three opened files, in the first three file descriptors of the process, corresponding to the *standard input*, *standard output*, and *standard error*. Those file descriptors correspond usually to the teletype device (`/dev/tty`) under UNIX, or one of the virtual console under Linux (`/dev/tty1`, `/dev/tty2`, etc.), or finally the console device under Plan 9 (`/dev/cons`). Those file descriptors can also correspond to regular files when the user uses redirections in the shell (e.g., `ls > list.txt`, see the SHELL book [Pad18]).

Under a windowing system, those input/output descriptors must be connected to the emulator. This is why in Figure 2.1, the terminal emulator is connected in both directions to the textual application; the terminal emulator relays from the window manager the keyboard input to the application, and relays from the application the output text to the display server (by drawing this text with a special font in the window).

In X Window, terminal emulators (e.g., `xterm`, `rxvt`) are separate programs. The standard input and output of command-line applications running under those terminals are connected to a *pseudo-tty* (PTY), which is a pair of *pseudo-devices* (the master and the slave) managed by the kernel. Those pseudo-devices and connected themselves to the terminal program in user space, as well as to the command-line application. I must admit I do not fully understand how it works. The code of `xterm` is very complex with more than 80 000 LOC (ten times

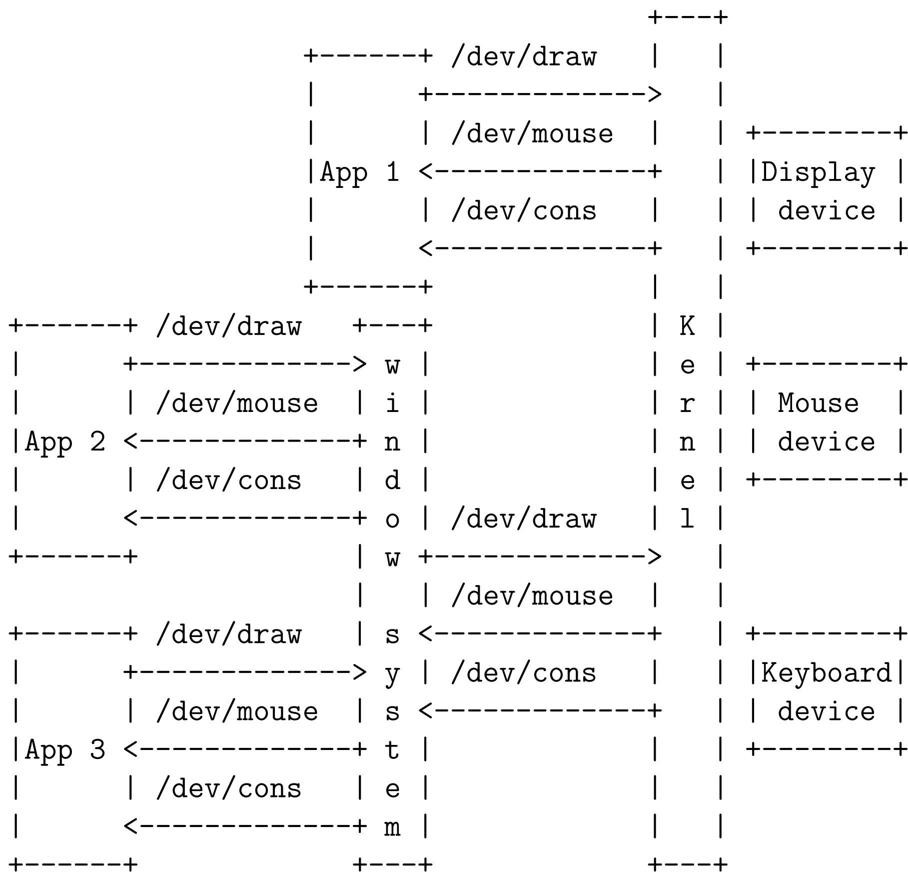


Figure 2.3: Windowing system as a multiplexer.

more code than the code of `rio`, which is the whole windowing system). `rxvt` is smaller, but still has 37 000 LOC.

With `rio`, the terminal emulator is an integral part of the windowing system and accounts for only 2600 LOC (including the code to support scrollbars, filename completion, copy-pasting, etc). This is mainly because the pseudo-tty is replaced by a more general approach: *virtual devices* and *filesystems in user space*. Indeed, under `rio`, a command-line application still opens the console device by opening in read and write modes `/dev/cons`. However, this file, thanks to the per-process namespace feature of Plan 9, is not the device file managed by the kernel but a virtual device managed by `rio`; every file request on `/dev/cons` is redirected to a 9P request to `rio` (which itself uses the "real" `/dev/cons` managed by the kernel).

2.1.9 Windowing system API

I just described the internal structure of a windowing system, as well as the environment needed by command-line applications when running under a terminal emulator. But, what about the external interface of a windowing system, as well as the needs of a window application? What API should a windowing system offer to its clients?

Obviously, a window application wants to access the screen, the mouse, and optionaly the keyboard. Note that there is already an API to access those devices under Plan 9. Indeed, a Plan 9 graphical application (e.g., Doom) can simply read or write in the `/dev/draw`, `/dev/mouse`, and `/dev/cons` devices files, which are managed by the Plan 9 kernel (see the Graphics book [Pad16c] and Kernel book [Pad14]). The top of Figure 2.3 illustrates one such graphical application called `App 1`. This program can also use functions from the `draw.h`, `mouse.h`, and `keyboard.h` header files instead of using directly the device files, but those functions are just thin wrappers that ultimately read and write in the corresponding device files.

Thus, under Plan 9, one way to define the external interface of a windowing system is to just mimic the

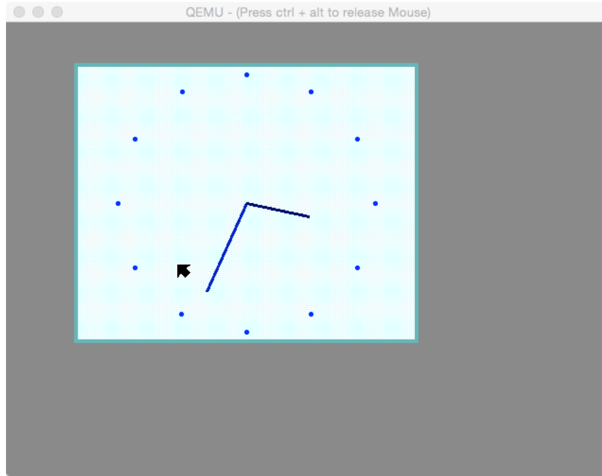


Figure 2.4: `clock` running inside `rio`.

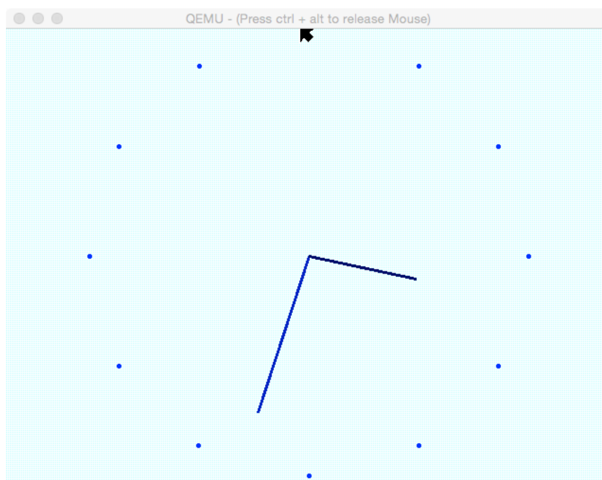


Figure 2.5: `clock` running without `rio`.

interface defined by the kernel, as shown at the bottom of Figure 2.3. As mentioned in Section 1.2, `rio` is implemented as a *multiplexer*; it accesses the physical devices (managed by the kernel) at the bottom right of Figure 2.3, and provides virtual devices to its clients (`App 2` and `App 3`) with similar interfaces at the bottom left of Figure 2.3.

This multiplexer approach makes `rio` a *transparent* [Pik88] windowing system. Indeed, `rio` does not need to introduce a new API. In fact, this transparence enables the same graphical application to run with or without `rio`, as shown respectively in Figure 2.4 and Figure 2.5 for the `clock` application. Under Plan 9, there is almost no difference between a graphical application and a window application. This transparence enables also `rio` to run recursively under itself (see Section 13.2).

X Window, on the opposite, is not a transparent windowing system. It introduces a special protocol and special APIs to access the screen, the mouse, and the keyboard. You can not run a window application without X Window, and you can not run a graphical application (e.g., Doom) inside a window in X Window (unless you rewrite Doom to use the X Window API). Concretely, this special API is called Xlib. An X client never touches the screen, mouse, or keyboard directly; it calls Xlib functions that marshal the corresponding X11 protocol requests and write them on the socket to the X Window display server (see Section 1.3 and the sockets mentioned there). So even the API is, at bottom, just a wrapper serializing requests onto a connection.

In Plan 9, by contrast, the “API” is nothing more than `read` and `write` on the `/dev/` files the kernel already

exposes, which is why a window application and a bare graphical application end up looking almost the same.

2.1.10 Window applications versus graphical applications

Window applications and graphical applications have a lot in common: they both draw on the screen, use the mouse, and possibly use the keyboard. As I said above, in Plan 9, the difference between a graphical application and a window application is fuzzy since you can also run graphical applications inside a window. However, this is possible only because of some careful design decisions in `rio` and `draw`. There are a few differences between running inside or outside a windowing system, as explained in the following sections.

A virtual screen

One of the role of a windowing system is to offer abstractions hiding complexity, just like the kernel does. Indeed, thanks to *virtual memory*, a process thinks it is alone and has access to the whole memory, starting at address 0. This simplifies greatly programming; the programmer does not have to care about the physical memory layout and which memory is used by other processes. In the same way, thanks to preemptive scheduling, a process thinks it is alone and has exclusive access to the CPU. Again, the programmer does not have to care about the other processes and how they use the processor (or processors). Each process uses a *virtual CPU*.

When running without `rio`, a graphical application has access to the whole screen. The origin point (`Pt(0,0)`) corresponds to the top left corner of the screen (see the GRAPHICS book [Pad16c]). This should be similar when the graphical application is running under `rio`. This is why `rio` offers a *virtual screen* to each window, where the origin point corresponds to the top left corner of the window content.

Thanks to the virtual screen, a graphical application running in a window does not even know it is running inside a windowing system. Moreover, the programmer does not have to care about the other windows, or where is located the window on the screen; whatever the location, drawing a line from `Pt(0,0)` will always start from the top left corner of the window, even though the window itself is located at the very right of the screen. The programmer can use *virtual coordinates* (a.k.a. *logical coordinates*); those coordinates are then translated in *physical coordinates* on the screen by the windowing system.

A virtual mouse

What is true for the screen should also be true for the mouse. This is why `rio` offers also a *virtual mouse* to each window; the location of the mouse read by the window application is relative to the window, not the whole screen. The programmer can use virtual coordinates again.

With `rio`, the programmer does not have to check if the mouse is on top of its window, or if the mouse is used concurrently by another program; all of this is handled automatically by the windowing system, which hides complexity by offering a virtual mouse. When a window is at the top, and the mouse cursor over this window, `rio` then *dispatches* the mouse event to the corresponding process.

Overlapping windows

In addition to a reduced drawing surface, an important difference for a graphical application running inside a window is that windows can overlap each other, and so hide each other. Where are stored the hidden pixels? How are those hidden pixels restored when the window is exposed back?

In X Window, because the early graphics machines did not have much memory, the hidden pixels are not saved anywhere. Instead, the display server *notifies* the client by an *expose event* when a part of its window is exposed back. It is the responsibility of the client, and so of the programmer, to draw back what was hidden.

With `rio`, the programmer does not have to care about overlapping windows. The windowing system hides complexity by storing the hidden pixels in *off-screen images*. When a window is exposed back, the windowing system then copies the pixels from those off-screen images back to the screen. This is consistent with the idea of the virtual screen: the programmer does not have to care about the other windows.

To manage overlapping windows, `rio` relies on a special data structure of `draw`: the image layer (see the GRAPHICS book [Pad16c] and [Pik83b]). A *layer* is an image that overlaps a rectangular sub-area of another image called the *base layer*. With `draw`, the programmer can use multiple layers stacked on top of each other and on top of the base layer. When a program draws in a layer, the pixels overlapped by another layer are automatically saved in an off-screen image. All the drawing functions of `draw.h` have special code to handle the case where the image passed as a parameter is a layer. The programmer can also use additional functions from `window.h` that are valid only for layers, for instance, moving a layer at the top with `topwindow()`. This function possibly copies the hidden pixels saved in an off-screen image (if the layer was overlapped) back to the base layer (e.g., the screen).

A layer is similar to a window, but the layer does not have any associated process; it is just a graphic construct. It is one of the building block of `rio`. Indeed, as you will see later in Section ??, with `rio` *each window is associated to a layer*; each window will draw in its layer. Moreover, when the user clicks on a window, `rio` internally calls `topwindow()` with the appropriate layer as a parameter.

Creating windows

A windowing system should offer an API to create windows, not just to draw things in a window. In Plan 9, the toplevel windows, whose dimensions are specified by the user (see Section 2.2.2), are created by `rio`, but each graphical application can also create *sub-windows* inside its window. Just like `rio` internally uses layers to represent toplevel windows, a graphical application can also use layers to represent sub-windows. Thanks again to `window.h`, an application such as the editor `acme` can use multiple layers to represent different files in multiple columns. In the same way, a dialog box, a menu, or any *widget* can be represented internally as a layer (see the WIDGETS book [Pad26] for more information). By using a layer, the programmer of the widget does not have to care about the pixels overlapped by the widget (or the pixels of the widget overlapped by another widget).

Note that layers do not have a border. To implement the window decorations, `rio` draws a blue border rectangle inside the layer. Note also that sub-windows have to be inside the parent window. It is not possible to create a layer that extends above the boundaries of the window. However, `rio` offers another API to create toplevel windows. This API requires advanced features of `rio` and so is explained later in Chapter 13.

Resizing windows

The last difference between a window application and a graphical application is that windows can be resized. Unfortunately, as opposed to overlapping windows, this complexity can not be hidden to the programmer.

In most windowing systems, the window application is notified of a *resize event* when its window is resized. It is then the responsibility of the programmer to provide a *callback* for such an event that redraws everything.

In X Window, this event is communicated to the client application through the general *event mechanism* of X11. At startup time, the application communicates to the display server an *event mask* specifying the set of events the application is interested in. If the resize event matches the event mask, then X Window will notify the client of a resize event.

With `rio`, the event is communicated to the application through the `/dev/mouse` virtual device file. A graphical application usually reads this file to keep track of the changes to the mouse location and the states of its buttons. During a resize event, `/dev/mouse` contains the character `'r'` (for resize), instead of the character `'m'` (for mouse).

2.2 `rio` interfaces

I just described the general principles of a windowing system, and illustrated those principles with examples from X Window and `rio`. I will now focus exclusively on `rio` and give more details about its interfaces.

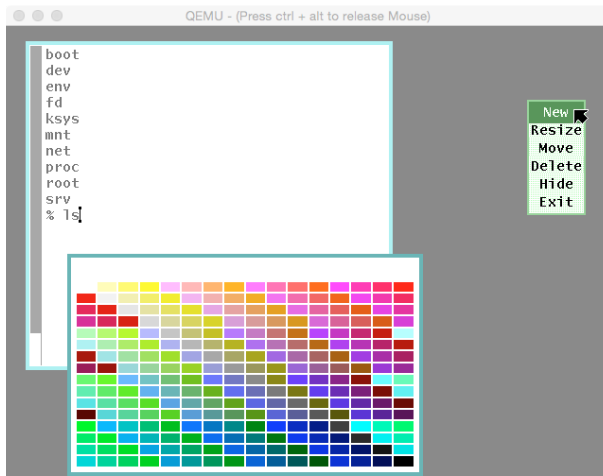


Figure 2.6: Graphical user interface of `rio` after a right-click.

`rio` is first a program you run through the command line, with optional command-line arguments. Then, it takes over the screen and becomes a graphical application. Finally, it internally spawns a new process acting as a filesystem. Thus, `rio` has three different interfaces, which I will describe in the following sections.

2.2.1 Command-line interface

The command-line interface of `rio` is pretty simple:

```
$ rio -help
usage: rio [-f font] [-i initcmd] [-k kbdcmd] [-s]
```

Most of the time you will run `rio` without any argument, as shown in Section 1.4. The arguments are all optionals and correspond to advanced features of `rio` I will explain in Chapter 13.

2.2.2 Graphical user interface

The most important interface of `rio` is of course its graphical user interface. Once launched from the command-line, `rio` takes over the whole screen[5] and displays a grey background image, as shown in Figure 1.2. Then, using the mouse, you can create new windows. Figure 2.6 illustrates the main elements of `rio`'s GUI.

As mentioned in Section 2.1, `rio` has a minimalist interface: no icon, no title bar, just a thin blue border around windows. You can left-click on a window to put it at the top if it was overlapped. Moreover, by doing so, the window gets the *focus*; this means every keyboard input will be sent to this window. When a window gets the focus, its border changes from a light blue to a darker blue, as shown at the bottom of Figure 2.6. Moreover, for window terminals, if the window loses the focus, the text inside the window changes from black to a light grey, as shown in the left of Figure 2.6.

By right-clicking outside any window, on the grey background image of `rio`, you trigger a *system menu* allowing you to create, resize, move, delete, or hide a window, as shown in the right of Figure 2.6. Note that `rio` requires a mouse with three buttons (left, middle, and right). To create a new window, activate the system menu, then hold the right-click, choose `New`, and finally release the right-click. The cursor will then change from a big arrow to a plus sign, to indicate a different *mode* of operation. You must now specify a rectangle by right-clicking again and *hold* while drawing a rectangle on the screen, as shown in Figure 2.7. Once you release the right-click, a new window terminal will appear, as shown in Figure 2.8, with a shell prompt at the top.

[5] Unless it is run recursively inside one of its window, as shown in Section 13.2.

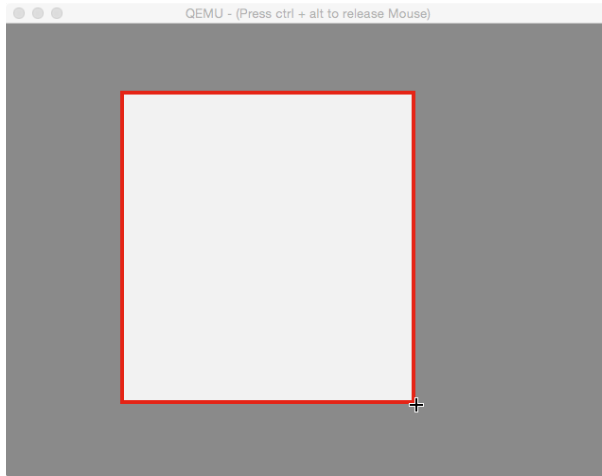


Figure 2.7: Creating a new window.

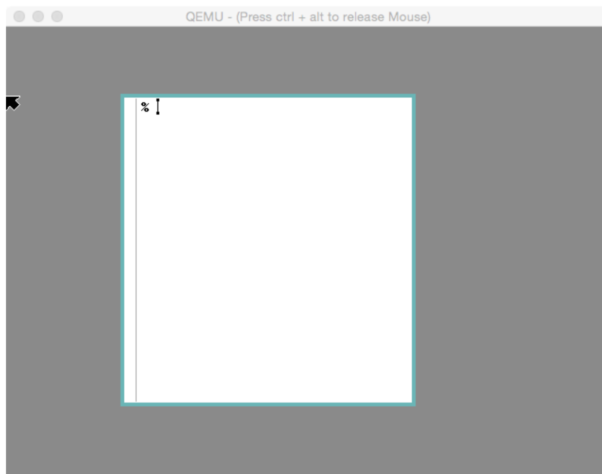


Figure 2.8: Built-in terminal running under `rio`.

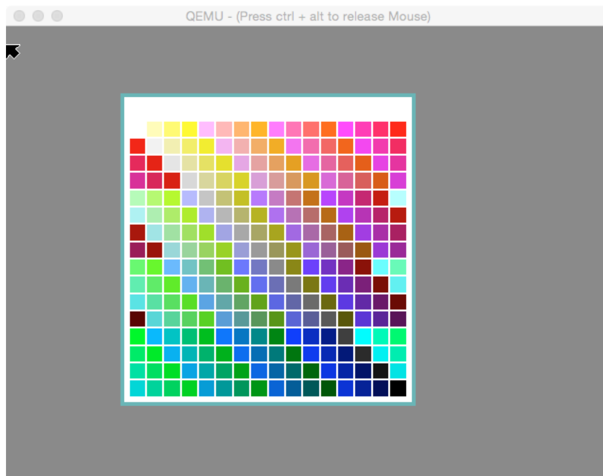


Figure 2.9: `colors` graphical application running under `rio`.

Because there is no icon in `rio`, to launch a graphical application you need first to create a window terminal. Once created, you can type in the terminal window the name of the graphical application you want to launch, for instance, `colors`. This application should then take over the virtual screen of the window terminal[6], as shown in Figure 2.9. Once you quit this graphical application, the terminal will be back.

You can also use the border of the window to resize or move the window. When the mouse is over the border, the cursor changes again to indicate a possible action. By left-clicking or middle-clicking on the border, you can respectively resize or move the window.

For a full description of the GUI of `rio`, I refer you to its manual page in `docs/man/1/rio` in my Plan 9 repository. You can also watch the historical demo of the Blit and `mpx` at `https://www.youtube.com/watch?v=emh22gT5e9k`; the user interface of `mpx` is almost identical to the one in `rio`.

2.2.3 Filesystem interface

As opposed to the two previous interfaces, the last interface of `rio` is almost invisible to the user. It is the set of files served by `rio` to all its window processes: the filesystem interface. It is mostly invisible because it is mainly an interface for programs, not users. Moreover, as explained in Section 2.1.9, `rio` is a transparent windowing system; the files served by `rio` mimic and replace existing device files managed by the kernel (e.g., `/dev/mouse`), making the interface harder to notice.

In the following sections, I will describe the files served by `rio`. Those files act as interfaces to access the screen, the mouse, the keyboard, but also the windowing system itself. I will prefix those files with `/mnt/wsys/` because that is where the `rio` filesystem is originally mounted by the window processes. However, as mentioned in Section 1.2, those files are also available in `/dev`, thanks to the union-mount feature of the Plan 9 kernel. It is by being accessible in `/dev/` that `rio` can become a multiplexer, as explained in Section 2.1.9. However, by using the `/mnt/wsys` prefix, it is clearer that the file involved is the file served by `rio` (e.g., `/mnt/wsys/mouse`) and not the real device served by the kernel (e.g., `/dev/mouse`).

In the following sections, I will also repeat partly explanations found in the Graphics book [Pad16c] and Kernel book [Pad14], as `rio` emulates interfaces provided by the graphics system and the kernel. However, the interfaces have a few differences because the context in which the graphical application runs is different.

[6] Just like it takes over the whole screen when launched outside `rio`.

The screen, `/dev/draw` and `/mnt/wsys/winname`

The first interface `rio` needs to mimic is the interface to the screen, which under Plan 9 involves the `/dev/draw/` directory and the `draw.h` header file.

Figure 2.3 showed a windowing system multiplexing `/dev/draw`, `/dev/mouse`, and `/dev/cons`. This was how $8\frac{1}{2}$ [Pik91], the ancestor of `rio`, was implemented. Under $8\frac{1}{2}$, each window was seeing a `/mnt/wsys/draw` virtual device file in which the process could write drawing commands. $8\frac{1}{2}$ would then convert the logical coordinates in the drawing command to physical coordinates, and send the resulting command to the real `/dev/draw` device file managed by the kernel. This was elegant. However, each drawing operation involved two communications: one between the application and the windowing system, and the other between the windowing system and the kernel. Those two round-trips were too inefficient for certain applications.

To remove one round-trip, `rio` does not multiplex `/dev/draw`. Instead, each window application connects directly to the `draw` device, which is more efficient. However, this forced the `draw` device to serve multiple processes in addition to `rio` (in `/dev/draw/2/`, `/dev/draw/3/`, etc), and to become a display server. Moreover, this required a new mechanism for `rio` to communicate to `draw` which part of the screen is allocated to each client of `draw`. This mechanism is complex and involves multiple features of `draw` and `rio`:

1. Instead of `/mnt/wsys/draw`, `rio` serves a file named `/mnt/wsys/winname`, which contains a different string for each window (e.g., `"window.3"`).

2. This string corresponds to the name of a layer allocated by `rio` for the window (see Section 2.1.10 for an introduction to layers). This layer has the dimension specified by the user during the creation of the window (see Figure 2.7).

3. This layer, which is also an image, is made *public* by `rio`. Thanks to an inter-process communication (IPC) feature of `draw`, this public image can be accessed by multiple processes (see the GRAPHICS book [Pad16c]).

4. On the client side, remember that each graphical application must first call `initdraw()` (see the GRAPHICS book [Pad16c]). `initdraw()` calls itself `gengetwindow()`, which reads `/dev/winname` and grabs a reference to the corresponding public image. This reference is then stored in the global `view`. By using `view` instead of `display->image` (see the GRAPHICS book [Pad16c]) for the arguments of the drawing functions of `draw.h`, the graphical application can become also a window application, for free.

Note that `initdraw()` calls `gengetwindow()` even when the graphical application runs outside `rio`. To *bootstrap*, the `draw` device serves also a `/dev/winname` file, which contains the string `noborder.screen.1`. This string corresponds also to an image: the whole screen. In that case, the global `view` and `display->image` references both the screen.

For more information, Section 9.6.1 explains the full trace of a drawing operation through the windowing system, the graphics system, and the kernel.

The mouse, `/mnt/wsys/mouse` and `/mnt/wsys/cursor`

The second interface `rio` needs to mimic is the interface to the mouse, which under Plan 9 involves the `/dev/mouse` and `/dev/cursor` files, as well as the `mouse.h` header file.

The `/dev/mouse` interface defined by the mouse device driver in the kernel is simple. As mentioned in Section 2.1.10, a graphical application can keep track of changes to the mouse location or the states of its buttons by reading (with `read()`) `/dev/mouse`. When the user does an action with the mouse, the `read()` system call returns, and the buffer parameter of `read()` is modified to contain the character `'m'` (for mouse), followed by integers encoding the location and the button states of the mouse. The graphical application can then inspect those integers and modify the GUI, or do nothing.

As opposed to `/dev/draw`, `rio` does multiplex `/dev/mouse`[7] as shown in Figure 2.3. The main job of `rio` regarding the mouse interface is to relay a `read()` by the window application on `/mnt/wsys/mouse` to a `read()` on the real `/dev/mouse`, and to convert the physical coordinates of the mouse to logical coordinates. Moreover, as mentioned in Section 2.1.10, `rio` abuses `/mnt/wsys/mouse` to also transmit the resize events to the graphical application. The buffer contains then the character `'r'` (for resize).

After a window application reads the `'r'` character in `/mnt/wsys/mouse` (resize), it is the responsibility of the programmer to call `getwindow()`, which internally calls `gengetwindow()`, the function I mentioned before. `getwindow()` is a thin wrapper around `gengetwindow()` that supplies the default arguments: it constructs the `/dev/winname` path from the display and passes the global `view` and `screen` pointers. `gengetwindow()` is the lower-level function that does the actual work—reading `/dev/winname`, fetching the named image, allocating a screen and window on it—but takes all these as explicit parameters[8]. `getwindow()` reads `/dev/winname`, which should now contain a new string corresponding to a new public image with the new dimension of the window. Then, `getwindow()` grabs the reference to this new public image in which the graphical application should now draw in. `getwindow()` also updates the global `view`.

Another job of `rio` is to multiplex `/dev/cursor` and to offer a *virtual cursor* to each application. Thanks to `/mnt/wsys/cursor`, each application can use a different cursor and can change the cursor. When the user hovers a window, the cursor changes according to what the window application wrote in his `/mnt/wsys/cursor` file; if nothing was written, a default cursor is used.

The fact that a program running in a window opened or not its `/mnt/wsys/mouse` file is an important information for `rio`. Depending on this opened status, `rio` will behave differently. For instance, if an application opens `/mnt/wsys/mouse`, many features of the terminal emulator are automatically disabled. Indeed, using the mouse is a strong hint for `rio` that the application is a graphical application, not a command-line application. Another hint is the opening of `/dev/draw/new` by the application, but this file is not managed by `rio`, and so out of reach of `rio`. In the rest of the book, I will use the term *graphical window* for a window in which the underlying program opened `/mnt/wsys/mouse`, and *textual window* otherwise.

The last important information regarding the mouse interface concerns the `mouse.h` header file. A programming alternative to using directly `/dev/mouse` is to use the functions from `mouse.h` such as `initmouse()` (see the GRAPHICS book [Pad16c]). `initmouse()` internally uses `/dev/mouse`, but wraps the device file in a data structure (`Mousectl`) allowing the use of *threads* and *channels* (see the LIBCORE book [Pad16a]). Using channels gives more flexibility to the programmer, as you will see first in the `hellorio` example (Section 2.3) and then in the rest of the book. For instance, the program can *react* simultaneously to changes in `/dev/mouse` and `/dev/cons`, and so can handle mouse events as well as keyboard events. Without `mouse.h`, the programmer can either read synchronously `/dev/mouse`, or read synchronously `/dev/cons`, but not both at the same time[9]. Note that `rio` itself calls `initmouse()` at startup time, and uses heavily channels and threads.

The keyboard, `/mnt/wsys/cons` and `/mnt/wsys/consctl`

The last interface `rio` needs to mimic is the interface to the keyboard, which under Plan 9 involves the `/dev/cons` and `/dev/consctl` files, as well as the `keyboard.h` header file.

[7] Why the difference? Why is there not a mouse server, just like there is a display server? Because there is no need to optimise the access to `/dev/mouse`. Indeed, the two round trips underlying the access to `/mnt/wsys/mouse` are not as critical as the access to an hypothetical `/mnt/wsys/draw`. In the context of a video game, a game application may have to generate 30 to 60 images per second; this may require hundreds or more calls to drawing functions of `draw.h`, and many accesses to `/dev/draw`. However, user actions are slower and the application does not need to react to a mouse event so frequently. Moreover, the size of the data involved with `/dev/mouse` is small: just a few bytes to represent a mouse location and button states.

[8] Why not just call `initdraw()` again after a resize? Because `initdraw()` does far more than fetching the window image: it reopens the display connection (`/dev/draw`), reinitializes fonts, and rewrites the window label. All of that is unnecessary after a resize—the display is still connected, the fonts are still loaded. Only the window image changed (because `rio` allocated a new layer with the new dimensions). `getwindow()` is the surgical version that refreshes just the one thing that changed.

[9] There is no `select()` system call in Plan 9. Threads, channels, and the Plan 9 function `alt()` are the Plan 9's way to do things done usually with `select()` under UNIX.

/dev/cons stands for *console device*; it is the device representing the terminal. The /dev/cons interface defined by the kernel is simple: a program reading from /dev/cons will read characters from the keyboard; a program writing to /dev/cons will output text on the screen. At boot time, the first Plan 9 process opens /dev/cons two times: one in read-mode, and the other in write-mode. Those two first file descriptors correspond to the *standard input* and *standard output* of the program (see the Kernel book [Pad14]). Those file descriptors and then inherited through fork() and exec() by the shell, as well as by the command-line applications launched from the shell, unless the user used redirections (see the Shell book [Pad18]). Remember that the printf() function from the C library internally does some write(1, ...) and the scanf() function internally does some read(0, ...).

The kernel via its terminal device driver offers also a few convenient features by default regarding the input of characters. For instance, when a program reads /dev/cons, the user can interactively edit the characters to sent to the program by using the *backspace* or *delete* keys to correct typing mistakes. He can also use the *cursor* keys to move in the line. Moreover, the characters typed are automatically *echoed* on the screen, making it easy to see the typing mistakes. Finally, the characters are sent only when the user types the *newline* character. By putting those features in the terminal device, all command-line applications do not have to care about typing mistakes.

Note that those line-editing features can also be disabled by the application. Indeed, in certain contexts, the application may not want the input to be buffered. For instance, a video game wants to respond as soon as possible to the key typed by the user; it does not make sense to force each time the user to also type the newline character. This is why the kernel provides also the /dev/consctl (for console control) device file. By writing rawon (for *raw access* on) in /dev/consctl, the application indicates to the kernel that the default line-editing features of the console device should be disabled; the application wants raw access to the keyboard (note that rio itself writes rawon in /dev/consctl at startup time, so it can handle the keyboard itself). A call to read() on /dev/cons will then return after each key is typed. Moreover, no character will be echoed by default on the screen.

These two behaviors have conventional names you will meet throughout this book and the wider UNIX world: the default, line-edited and newline-buffered behavior is called *cooked mode*, while the character-at-a-time behavior enabled by rawon is called *raw mode*. The kernel code implementing cooked mode—buffering a line and letting the user edit it before the program sees it—is traditionally called the *line discipline*.

The job of rio regarding the keyboard interface depends also on whether the application in the window wants or not raw access to the keyboard. This is why rio multiplex not only /dev/cons, but also /dev/consctl. The behavior of /mnt/wsys/cons depends on the opening and content of /mnt/wsys/consctl. This is usually correlated with whether or not the window is a graphical window (raw access on), or textual window (line-editing on).

For graphical windows, writing on /mnt/wsys/cons should be considered an error. Indeed, the graphical application should instead use the string() function from draw.h to output text on the screen at a specific location via /dev/draw (see the Graphics book [Pad16c]). Only reads on /mnt/wsys/cons should be supported by rio. Moreover, each key typed should be sent directly to the graphical application if its window has currently the focus.

For textual windows, the job of rio and its terminal emulator is to imitate what the console device does by default, including the line-editing features. In fact, under rio, the user can also use *copy-pasting* with the mouse to edit a line before it is sent to the program. Both reads and writes on /mnt/wsys/cons should be supported by rio for textual windows. Reads to /mnt/wsys/cons by a command-line application should return only when the user typed the newline character in the terminal emulator. Writes to /mnt/wsys/cons should be converted to calls to string() by the terminal emulator, in order to output text at the right place in the window (possibly applying line wrapping).

Similar to the mouse, a programming alternative to using directly /dev/cons is to use the functions from keyboard.h such as initkeyboard() (see the Graphics book [Pad16c]). initkeyboard() internally enables raw access to the keyboard by writing rawon in /dev/consctl. initkeyboard() internally uses /dev/cons, but

wraps the device file in a data structure (`Keyboardctl`) allowing also the use of threads and channels. Again, like any window application, `rio` calls `initkeyboard()` at startup time. You will first meet this `initkeyboard()` pattern in the small standalone `hellorio` example program (Section 2.3), and later see `rio` itself rely on it in Chapter 4.

Other `/mnt/wsys/` files

I just presented the main files served by `rio` to its window processes. Those files are virtual versions of device files managed by the kernel. Thanks to those virtual devices, `rio` is a transparent windowing system enabling graphical applications to run inside windows.

`rio` serves also files that are interfaces to advanced features of the windowing system itself. Here is the list of those files and a short description of their content (Chapter 12 and Chapter 13 will give more details about those files):

- `/mnt/wsys/winid`: This read-only file contains a number unique to each window, the *window identifier*. This identifier is useful in conjunction with `/mnt/wsys/wsys/` presented below. Remember that each window process sees its own `/mnt/wsys/` files thanks to the per-process namespace—just like each window sees a different `/mnt/wsys/cons` (bound to `/dev/cons`), each window sees a different `/mnt/wsys/winid` returning its own identifier.

- `/mnt/wsys/label`: `rio` does not use title bars, but each window can write a string called a *window label* in its `/mnt/wsys/label` file, to describe the window. This label is then used by the system menu; the menu lists all the hidden windows by their labels (see Section 7.10)

- `/mnt/wsys/screen`: This read-only file contains an image (in the Plan 9 image format) representing the content of the whole screen at the moment `/mnt/wsys/screen` was read. Unlike the other `/mnt/wsys/` files, this one is not per-window: every window sees the same whole-screen image. Thanks to this file, it is very easy to take screenshots in Plan 9.

- `/mnt/wsys/window`: This read-only file contains also an image, but representing only the content of the window. This one *is* per-window, unlike `/mnt/wsys/screen`.

- `/mnt/wsys/text`: This read-only file is useful only for textual windows. It contains a full dump of the text displayed in the terminal.

- `/mnt/wsys/snarf`: This file is used for copy-pasting (see Section 13.5.1).

- `/mnt/wsys/wdir`: This file is used for filename completion (see Section 13.5.3).

- `/mnt/wsys/wsys/`: This directory allows to explore the set of windows. `/mnt/wsys/wsys/` is to windows what `/proc/` is to processes (see the Kernel book [Pad14]). The *key* used for `/mnt/wsys/wsys/` is not the process identifier, as in `/proc`, but the window identifier presented above. Just like `/mnt/wsys/` contains files representing information about the window of the program, `/mnt/wsys/2/` contains the same files but representing the information about the window with the window identifier 2.

- `/mnt/wsys/wctl`: This file is used to control programmatically a window. Just like a user can use the mouse to act on a window, for instance, by clicking on the border of a window to resize it, a program can use `/mnt/wsys/wctl` to do similar things. By writing *control commands* in `/mnt/wsys/wctl`, a program can control its own window. In fact, a program can control also another window, for instance, by writing in `/mnt/wsys/wsys/2/wctl` (see Section 12.6 for more information).

For more information on the files served by `rio` and their format, I refer you to the documentation of `rio` in `docs/man/4/rio` in my Plan 9 repository.

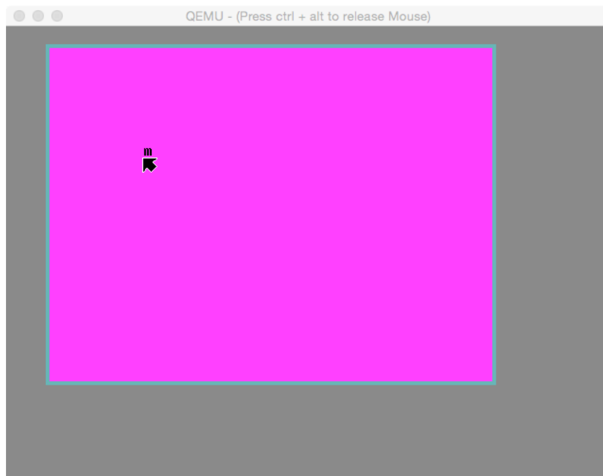


Figure 2.10: hellorio running in a window, after the user typed 'm'.

2.3 hellorio.c

Now that you know the principles and the interfaces of rio, I can present the code of a toy window application: hellorio. hellorio is a simple application written in C displaying initially Hello Rio at the location of the mouse. Then, when the user types a key, the Hello Rio message is replaced by the typed character, as illustrated in Figure 2.10. Finally, you can quit the application by typing the letter 'q' (for quit).

The goal of this section is to illustate some of the concepts I introduced before with concrete code. Moreover, the code of hellorio in hellorio.c, although simplistic, is a good introduction to the code of rio itself. Indeed, rio also needs to use the mouse, the keyboard, or draw things on the screen. hellorio.c will introduce also the use of channels and threads, which are heavily used by rio.

I already presented in the GRAPHICS book [Pad16c] the code of a toy graphical application: hellodraw. As I said in Section 2.1.9, a graphical application can also be a window application under Plan 9. However, this requires a few changes in the code of the graphical application, for instance, the use of the global view as mentioned in Section 2.2.3. hellodraw was not using this global, and so could not run inside a window. Moreover, hellodraw was just using the graphics facilities of draw, with functions from draw.h. hellorio is an *interactive* graphical application that also uses the mouse and the keyboard, with functions from mouse.h and keyboard.h. Finally, by handling the resize event, hellorio can also become a window application.

2.3.1 Skeleton and output code

Here is the skeleton of hellorio.c:

⟨rio/tests/hellorio.c 35⟩≡
```
#include <u.h>
#include <libc.h>

#include <draw.h>
#include <mouse.h>
#include <keyboard.h>

#include <thread.h>

Image *bgcolor;
Point mouseloc;
Rune str[20];
char buf[50];
```

```
⟨type EventType (hellorio.c) 38a⟩

void redraw(void);

void threadmain(int argc, char* argv[]) {
  int result;
  Keyboardctl* keyboardctl;
  Mousectl*    mousectl;
  ⟨threadmain() other locals (hellorio.c) 38b⟩

  result = initdraw(nil, nil, "Hello Rio");
  ⟨threadmain() sanity check result (hellorio.c) 37a⟩
  mousectl = initmouse(nil, view);
  ⟨threadmain() sanity check mousectl (hellorio.c) 37b⟩
  keyboardctl = initkeyboard(nil);
  ⟨threadmain() sanity check keyboardctl (hellorio.c) 37c⟩

  bgcolor = allocimage(display, Rect(0,0,1,1), RGBA32, true, DMagenta);
  runestrcpy(str, L"Hello Rio");
  mouseloc = Pt(200, 200);

  ⟨threadmain() alts setup (hellorio.c) 38c⟩
  redraw();
  ⟨threadmain() event loop (hellorio.c) 39d⟩
}

void redraw(void)
{

  draw(view, view->r, bgcolor, nil, ZP);
  runestring(view, mouseloc, display->black, ZP, font, str);

  sprint(buf, "%d, %d", mouseloc.x, mouseloc.y);
  string(view, addpt(mouseloc, Pt(20, 20)), display->black, ZP, font, buf);

  flushimage(display, true);
}
```

Here are a few important things to note about the skeleton of `hellorio.c`, from top to bottom, as well as how the code compares to the code of `hellodraw.c` in the GRAPHICS book [Pad16c]:

- In addition to `draw.h`, `hellorio.c` also includes `mouse.h` and `keyboard.h`, as it uses the mouse and the keyboard.

- `hellorio.c` also includes `thread.h`, a header file containing functions related to threads, the `Channel` structure, and the declarations of primitives operating on channels (e.g., `send()`, `recv()`). A *channel* is essentially a queue of *messages*. Threads communicate and *synchronize* with each other by exchanging messages through channels (see the LIBCORE book [Pad16a]). Even though the code in `hellorio.c` does not create threads explicitely, some of the functions called from `hellorio.c` (e.g., `initmouse()`, `initkeyboard()`) do create threads internally (see the GRAPHICS book [Pad16c]).

  The Plan 9 thread library defines two separate constructs to carry computations: threads and procs. A *proc* is a Plan 9 process containing cooperatively-scheduled *threads*. Remember that in Plan 9, processes can share memory with each other (via the `RFMEM` flag to `rfork()`, see the KERNEL book [Pad14]). Thus, just like multiple threads in the same proc share memory, multiple procs can also share memory, and can communicate with each other through channels declared in this shared address space. In other operating systems, a Plan 9 proc is similar to a *system thread*, and a Plan 9 thread is similar to a *light-weight thread* (or a *coroutine*).

- As I said in the GRAPHICS book [Pad16c], with `draw` *everything is an image*, including colors, as shown by the type of the global `bgcolor`.

- The key typed by the user is stored in an array of `Rune`s. A *rune* is the name given to a *unicode character* in Plan 9 (see the LIBCORE book [Pad16a]). The type of a key entered in the keyboard is not a `char` but a `Rune` in Plan 9, which is convenient. Indeed, the code of `hellorio.c` will work also if the user enters chinese characters, or letters with european accents. Even special combinations such as the `Control` key and the `d` key will return a single rune, whose representation `^d` will be correctly displayed on the screen (thanks to `runestring()` called in `redraw()`).

- Most of the graphics-related code is not in `main()`, as in `hellodraw.c`, but in a separate function: `redraw()`. This is because this code will be called multiple times, after each input event, as you will see soon.

- The entry point of `hellorio.c` is not `main()` but `threadmain()`. Indeed, the thread library provides already a `main()` that sets up a proc with a single thread executing `threadmain()`.

- Similar to `hellodraw.c`, the first call of `hellorio.c` is to `initdraw()`, to connect to the display server. `hellorio.c` also calls `initmouse()` and `initkeyboard()` to connect respectively to the mouse and the keyboard device. Those two functions internally create a proc reading synchronously on respectively `/dev/mouse` and `/dev/cons` (see the GRAPHICS book [Pad16c]). Those two functions uses a proc, and not a thread, because a thread should not block. Indeed, if for instance a thread is blocked on a read on `/dev/mouse`, the other threads in the same proc would also be blocked. Threads are cooperatively-scheduled; they need to cooperate with each other.

- Regarding the graphics-related code, `hellorio.c` is very similar to `hellodraw.c`, with the use of functions of `draw.h` such as `allocimage()`, `draw()`, `runestring()`, or `flushimage()`. The main difference is the use of the global `view`, set by `initdraw()` (see Section 2.2.3), instead of `display->image`, for the arguments of the drawing functions.

The skeleton of `hellorio.c`, above, omits the error management code shown below:

⟨`threadmain()` *sanity check result (hellorio.c)* 37a⟩≡ (35)
```
  if (result < 0)
    exits("Error in initdraw");
```

⟨`threadmain()` *sanity check mousectl (hellorio.c)* 37b⟩≡ (35)
```
  if(mousectl == nil)
    exits("can't find mouse");
```

⟨`threadmain()` *sanity check keyboardctl (hellorio.c)* 37c⟩≡ (35)
```
  if(keyboardctl == nil)
    exits("can't find keyboard");
```

In the rest of this book, I will usually not comment the error-management code. Such code is often trivial (but necessary).

2.3.2 Input code

I have shown the code responsible for the visual output of `hellorio`. I can now present the code dealing with the inputs to `hellorio`, making `hellorio` an interactive program.

`hellorio` must handle three different kinds of inputs: inputs from the mouse, from the keyboard, and from the windowing system itself with its resize event. The type below defines those different *event types* for `hellorio`:

⟨*type EventType (hellorio.c)* 38a⟩≡ (35)
```
enum EventType {
  EMouse,
  EKey,
  EResize,

  NALT
};
```

`hellorio` needs to deal *simultaneously* with the mouse, keyboard, and windowing system. Indeed, the program can not just synchronously read `/dev/mouse`; maybe the next event will be a keyboard event, not a mouse event. To do so, the `thread.h` header file defines the `Alt` (for alternative) structure. As mentioned in Section 2.2.3, there is no `select()` system call in Plan 9. Instead, Plan 9 provides the `alt()` function (see the LIBCORE book [Pad16a]) that takes as an argument a map of event types to `Alt`. This map is stored first in a local variable:

⟨`threadmain()` *other locals (hellorio.c)* 38b⟩≡ (35) 39a ▷
```
// map<EventType, Alt>
Alt alts[NALT+1];
```

Each value of this map (each "alternative") contains essentially a channel. A call to `alt()` will then return if anything is exchanged (received or sent) on *one of the channels* of the map. In fact, `alt()` will also return the event type associated with the channel in which a message was exchanged. With `alt()`, it is possible to listen to multiple channels at the same time (or to emit to multiple channels at the same time).

For each event type, the local variable `alts` is initialized with the channel to receive from (or send to) in the `Alt.c` field:

⟨`threadmain()` *alts setup (hellorio.c)* 38c⟩≡ (35) 39c ▷
```
alts[EMouse].c = mousectl->c;
alts[EMouse].v = &mouse;
alts[EMouse].op = CHANRCV;

alts[EKey].c = keyboardctl->c;
alts[EKey].v = keys;
alts[EKey].op = CHANRCV;

alts[EResize].c = mousectl->resizec;
alts[EResize].v = nil;
alts[EResize].op = CHANRCV;
```

The `Alt.op` field contains the type of *channel operation*, Here, `CHANRCV` because the program is listening (receiving) on all channels. It is also possible to emit simultaneously on multiple channels by using instead `CHANSND`.

`mousectl` and `keyboardctl`, above, are two local variables containing the return value of respectively `initmouse()` and `initkeyboard()` (see the skeleton of `hellorio.c`). The types of those variables are respectively the structure `Mousectl` and the structure `Keyboardctl`, defined respectively in `mouse.h` and `keyboard.h`. Both structures are explained in the GRAPHICS book [Pad16c]. The most important fields in those structures are `Mousectl.c` and `Keyboardctl.c` which are the channels used to communicate with the procs created by `initmouse()` and `initkeyboard()`. For instance, the proc created by `initmouse()` reads synchronously on `/dev/mouse`, and when `read()` returns data, this data is sent on `Mousectl.c`. This data can then be read by any thread receiving on `Mousectl.c`. Thanks to `alts`, `hellorio` can listen simultaneously on changes in `/dev/mouse` or `/dev/cons`.

The field `Alt.v` must contain a pointer to an area where to store the data received on a channel (or the data to send, if the channel operation was `CHANSND`). The size of this area depends on the *channel type* (see the

LIBCORE book [Pad16a])[10]. For the keyboard, `Keyboardctl.c` is a channel containing up to 20 buffered keys (so even if the user types really fast, no data is lost).

```
⟨threadmain() other locals (hellorio.c) 39a⟩+≡                    (35) ◁38b 39b▷
  Rune keys[20];
```

For the mouse, `Mousectl.c` is a channel containing just one `Mouse`:

```
⟨threadmain() other locals (hellorio.c) 39b⟩+≡                    (35) ◁39a ??▷
  Mouse mouse;
```

Each array of `Alts` must finish with a special *end marker*: `CHANEND`.

```
⟨threadmain() alts setup (hellorio.c) 39c⟩+≡                           (35) ◁38c
  alts[NALT].op = CHANEND;
```

2.3.3 The event loop

I can finally show the last piece of code of `hellorio.c`: the *event loop* and the call to `alt()`:

```
⟨threadmain() event loop (hellorio.c) 39d⟩≡                                  (35)
  for(;;) {
    switch(alt(alts)){
    ⟨threadmain() event loop cases (hellorio.c) 39e⟩
    }
    redraw();
  }
```

When one of the channel in `alts` receives a message, `alt()` returns the corresponding event type. The program can then switch on the event type, inspect the message referenced from `Alt.v`, and modify globals.

For the mouse, the program modifies the global `mouseloc` with the last coordinate of the mouse:

```
⟨threadmain() event loop cases (hellorio.c) 39e⟩≡                      (39d) 39f▷
  case EMouse:
    mouseloc = mouse.xy;
    break;
```

This global is then used in `redraw()` as a parameter to `draw()` to display some text at the right location.

`redraw()` also prints `mouseloc` as text next to the string, so you can read off the pointer's position. Those numbers are relative to the window itself—(0,0) is the window's top-left corner, not the screen's—so the same spot inside the window always reports the same coordinates, wherever the window happens to sit. We will see later how `rio` gives each window these logical coordinates.

For the keyboard, the program modifies the local `keys`. As mentioned above, the channel type of `keyboardctl->c` is an array of 20 runes. A channel is a bufferred or unbuffered queue of messages. The call to `alt()` stores only one message in the area pointed by `Alt.v`. It is the responsibility of the programmer to call `nbrecv()` to transfer the remaining messages in the queue.

```
⟨threadmain() event loop cases (hellorio.c) 39f⟩+≡               (39d) ◁39e 40▷
  case EKey:
      for(i=1; i<nelem(keys)-1; i++)
          if(nbrecv(keyboardctl->c, keys+i) <= 0)
              break;
      keys[i] = L'\0';
      runestrcpy(str, keys);
      if(keys[0] == L'q')
        exits("done");
      break;
```

[10] Why not store the data in the channel itself? Because Plan 9 channels are untyped (`void*`), so every channel operation—`send()`, `recv()`, and `alt()`—requires the caller to provide the buffer, just like `read(fd, buf, n)` takes a buffer parameter rather than returning the data. A typed language like Go or OCaml can return the value directly from a receive.

The L prefix is C's *wide-character* literal syntax: `L'q'` produces a wide character rather than a `char`. In Plan 9 that wide character is a `Rune`, the integer type holding one Unicode code point (in standard UNIX C it would be a `wchar_t`). The keyboard channel delivers `Rune`s, so `keys` is a `Rune` array (see the LIBCORE book [Pad16a]).

For the resize event, the program just needs to call `getwindow()`, which will update the global `view`, as explained in Section 2.2.3:

⟨`threadmain()` *event loop cases (hellorio.c)* 40⟩+≡ (39d) ◁39f

```
  case EResize:
    if(getwindow(display, Refnone) < 0)
      exits("failed to re-attach window");
    break;
```

This concludes the code of `hellorio.c`.

With command-line programs, the program is in control; the program ask questions to the user. With graphical applications (and window applications), the user is in control; the application *reacts* to external events. This is a new programming model. The application is more than an interactive program, it is a *reactive* program.

2.4 Code organization

The code of `rio` in my Plan 9 repository is split in multiple directories, but the most important one is `windows/rio/`. Table 2.1 presents short descriptions of the source files in `windows/rio/`, as well as the main entities (e.g., structures, functions, globals) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed. The other directories of `rio` correspond to libraries used by code in `windows/rio`:

- `windows/libframe/`: This library contains the generic code of a *text widget*. This widget consists of a scrollbar on the left and an editable text area on the right. This library is used by the terminal emulator of `rio`, and is discussed in Appendix D. This library is also used by the Plan 9 editor `acme`.
- `windows/libplumb/`: This library is used to interact with the Plan 9 *plumb* mechanism [Pik00a], which is triggered in `rio` by the middle-click menu of the terminal emulator (see Section 13.5.2). Plumbing is a generalization of the cut-and-paste, drag-and-drop, and Multipurpose Internet Mail Extensions (MIME) mechanisms found in maintream operating systems. It allows applications to cooperate with each other without having to know each other.
- `windows/libcomplete/`: This library is used to provide filename completion in the terminal emulator (see Section 13.5.3).

The code of `rio` depends also on code in `lib_graphics/` and `lib_core/`, but the code in those directories is explained respectively in the GRAPHICS book [Pad16c] and LIBCORE book [Pad16a].

2.5 Software architecture

This section looks at `rio`'s architecture from two complementary angles. First its static architecture: the libraries and header files `rio` is built upon, and the kernel devices underneath them. Then its dynamic architecture: how `rio` is split at runtime into cooperating procs and threads. The two following subsections cover these in turn.

Function	Ch.	File	Entities	LOC
data structures	3	`dat.h`	`Window`[49] `Filsys`[53a] `Fid`[53e] `Xfid`[55b]	335
globals	3	`globals.c`	`mousectl`[48a] `windows`[51a] `input`[51e] `filsys`[53b]	42
entry point	4	`rio.c`	`threadmain()`[58]	174
prototypes	4	`fns.h`		108
keyboard thread	5	`thread_keyboard.c`	`keyboardthread()`[64]	36
mouse thread	5	`thread_mouse.c`	`mousethread()`[66a]	226
window threads	5	`threads_window.c`	`winctl()`[71b]	378
fileserver proc	5	`proc_fileserver.c`	`filsysproc()`[75] `filsysmount()`[99c]	172
master and worker threads	5	`threads_worker.c`	`xfidctl()`[79c]	128
cursor graphics	6	`data.c`	`crosscursor`[81] `corners`[83a]	165
cursor operations	6	`cursor.c`	`riosetcursor()`[85b]	21
window manager	7	`wm.c`	`button3menu()`[88b] `new()`[92c] `drag()`[111c]	559
window process creation	7	`processes_winshell.c`	`winshell()`[97d]	62
window methods	7	`wind.c`	`wtop()`[104b] `wclose()`[109b]	387
closing threads	7	`threads_misc.c`	`deletethread()`[108b] `winclosethread()`[109a]	42
fileserver utilities	8	`9p.c`	`filsysrespond()`[124] `filsyscancel()`[267d]	46
fileserver methods	8	`fsys.c`	`filsysattach()`[126] `filsysopen()`[132a]	461
virtual device methods	9	`xfid.c`	`xfidopen()`[133a] `xfidread()`[136b]	691
graphical window	10	`graphical_window.c`	`waddraw()`[164c]	19
terminal emulator	11	`terminal.c`	`winsert()`[177a] `wshow()`[179b] `wkeyctl()`[73b]	976
terminal scrolling	11	`scrl.c`	`wscrdraw()`[180b] `wscroll()`[202]	174
external window control	13	`wctl.c`	`wctlthread()`[259b] `parsewctl()`[263c]	452
copy/paste clipboard	13	`snarf.c`	`putsnarf()`[233k] `getsnarf()`[234a]	60
timer	13	`time.c`	`timerstart()`[223b]	115
error management	B	`error.c`	`error()`[282c] `derror()`[282d]	20
utilities	C	`util.c`	`min()`[284a] `emalloc()`[283b] `strrune()`[285c]	124
Total				5973

Table 2.1: Chapters and source files of `windows/rio/`.

2.5.1 Library and header dependencies

Table 1.1 in the Introduction already mapped, for each related Principia Softwarica book, the devices and header files `rio` relies on. Here I revisit those same dependencies from `rio`'s own side: which libraries it links against, and why.

`rio` is first a Plan 9 graphical application. It is not a special program; it relies, like all graphical applications, on devices managed by the Plan 9 kernel and its graphics stack:

- `/dev/draw/` to access the display device
- `/dev/mouse` and `/dev/cursor` to access the mouse device
- `/dev/cons` and `/dev/consctl` to access the keyboard device

`rio` communicates with the kernel through system calls (e.g., `open()`, `read()`) on those devices (see the Kernel book [Pad14]).

In fact, rio does not use directly those devices. Instead, it uses functions from draw.h, mouse.h, and keyboard.h. Those header files offer data structures and functions that are convenient wrappers around those device files. Thus, rio depends mainly on lib_graphics/libdraw/, the library behind draw.h, mouse.h, and keyboard.h (see the GRAPHICS book [Pad16c]).

rio uses many functions from draw.h, but also many functions from window.h. Indeed, rio makes heavy use of draw's layers (see Section 2.1.10 for an introduction to layers). Many of the window management tasks are actually done by draw, not rio. For instance, it is the libdraw function topwindow() which does most of the heavy lifting when you click on a window to put it at the top; it is topwindow() that restores the pixels overlapped previously by other windows.

rio is also a Plan 9 filesystem that communicates with the kernel using the 9P protocol (see the NETWORK book [Pad16d]). rio is a file server that answers to file requests on virtual devices accessed by its windows. Thus, rio must be two different things at once: a GUI and a server. Moreover, rio must coordinate many *independent* activities: user actions with the keyboard, user actions with the mouse, computations in the window processes, and actions from those window processes such as file requests on virtual devices. What is a good software architecture to deal with all those activities? A natural architecture is to use *threads*. Indeed, each independent activity can be represented by an independent thread. Then, *channels* can be used to communicate messages between threads. Both channel and thread functions are declared in thread.h. Thus, in addition to lib_graphics/libdraw/, rio depends also on lib_core/libthread/, the library behind thread.h (see the LIBCORE book [Pad16a]).

⟨*rio includes* 42⟩≡ (350b 349 348 347 346 345 343 342 341 340 339 338)

```
#include <draw.h>
#include <cursor.h>
#include <mouse.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

#include "dat.h"
#include "fns.h"
```

2.5.2 Processes, procs, and threads relationships

Figure 2.11 presents the threads and processes resulting from the execution of rio and two other programs in two different windows, as displayed in Figure 2.12. As mentioned in Section 2.3, the Plan 9 thread library provides two separate constructs to represent computation: procs and threads. A *proc* is a Plan 9 process containing one or more cooperatively-scheduled *threads*. Procs and processes are represented by enclosing rectangles in Figure 2.11, while threads are represented by enclosed rectangles. Moreover, multiple procs sharing the same address space are enclosed by a dashed rectangle.

Layers and components

Figure 2.11 can be decomposed in four different layers:

- The hardware layer, at the very bottom
- The kernel layer, providing abstractions of the hardware
- The filesystem layer (and its many device files), one of the main abstractions offered by the kernel
- The user processes layer

The user processes can themselves be divided in three different groups:

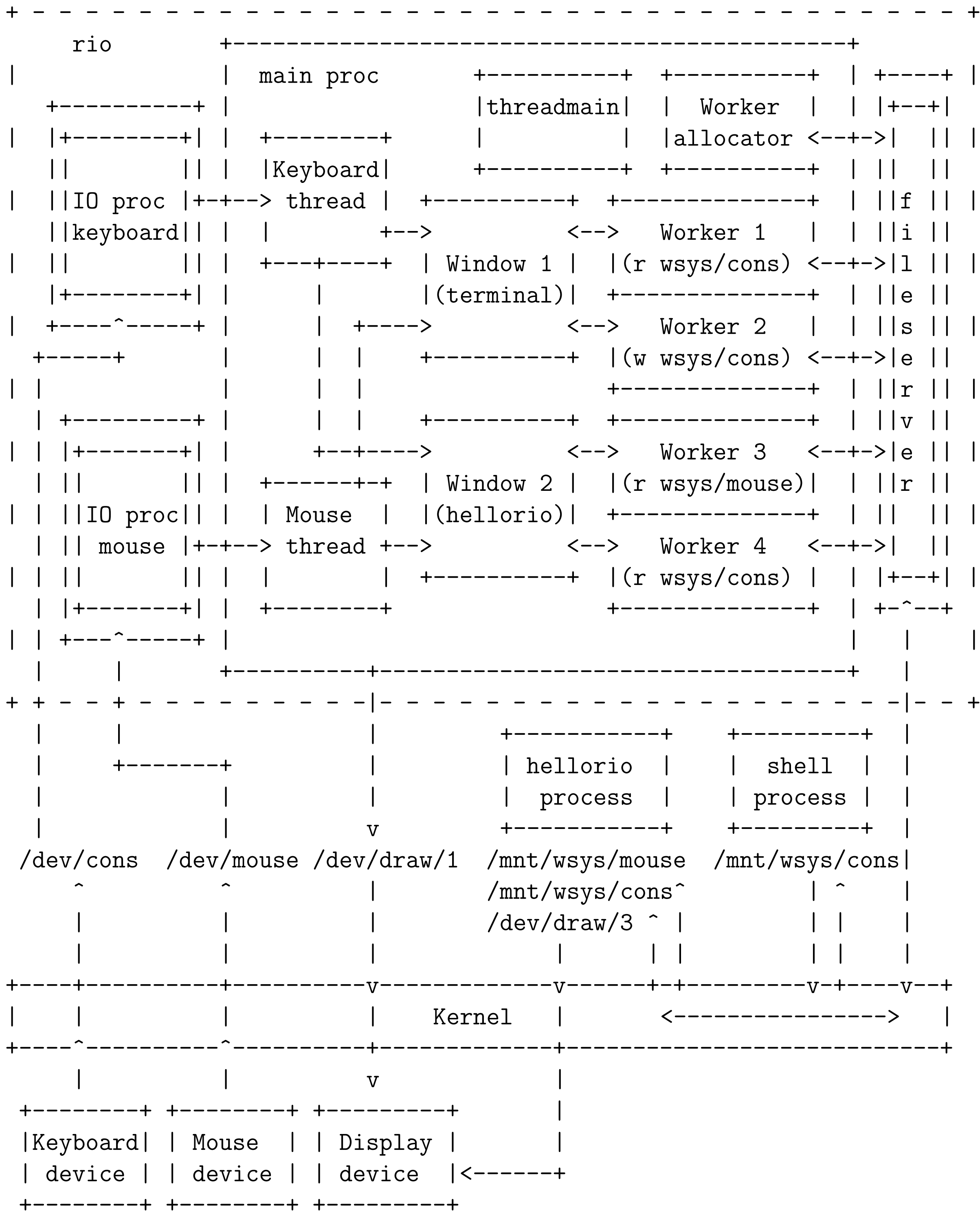


Figure 2.11: Processes, procs, and threads in `rio`.

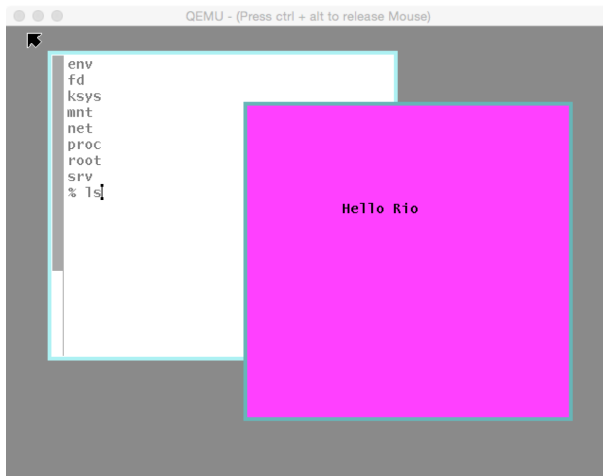


Figure 2.12: `hellorio` and a shell running in two different windows.

- A set of procs created by `rio` and sharing the same address space, at the top of Figure 2.11
- The `hellorio` process (see Section 2.3), running inside a window on the right in Figure 2.12
- A shell process, running in a terminal emulator on the left in Figure 2.12.

Of course, the most important group is the first group, which contains the procs of `rio`. `rio` is made of four different procs:

- `IO-proc-keyboard`: A proc containing a single thread reading `/dev/cons`, at the top left in Figure 2.11
- `IO-proc-mouse`: A proc containing a single thread reading `/dev/mouse`, at the middle left in Figure 2.11
- `fileserver`: A proc containing a single thread communicating with the kernel using the 9P protocol, at the top right in Figure 2.11
- `main-proc`: A proc containing many threads writing in `/dev/draw/1/`, in the middle of Figure 2.11

The I/O procs

I mentioned already the first two procs in Section 2.3 in the context of the `hellorio` program[11]. They are the results of calls to respectively `initkeyboard()` and `initmouse()` by `rio` during startup. Both procs are usually blocked in a `read()` on a device file. Once the user performs an action, with the keyboard or the mouse, one of the `read()` returns and the corresponding proc (actually the single thread in the proc) sends a message on the `keyboardctl->c` or `mousectl->c` channels. Two threads of `main-proc`, the keyboard thread and the mouse thread, are listening on those channels.

On procs and threads for concurrency

The use of multiple procs and threads to read one device file is due to the design of threads in Plan 9. As mentioned in Section 2.3, in Plan 9, threads are scheduled cooperatively. They should avoid to block on a system call doing IO. Indeed, if one of the thread in the proc is blocked, all the threads in the proc are blocked (see the LIBCORE book [Pad16a]). However, one huge advantage of this design is that threads do *not* need to use *locks*, an error-prone concurrency construct, for *mutual exclusion* with each other. By construction, only

[11] In fact, `hellorio` in Figure 2.11 creates also two IO procs. They are not shown in the figure to simplify things. Those procs would read `/mnt/wsys/cons` and `/mnt/wsys/mouse`.

one thread of a proc is running at one time (but multiple procs can run in parallel). The reason this removes the need for locks is subtle. Because scheduling is cooperative, a thread keeps the processor until it explicitly *yields*—and it yields only at well-defined points, typically a `send()` or `recv()` on a channel. A thread therefore never gives up control in the middle of updating a shared structure; by the time another thread of the same proc runs, the data is always in a consistent state. That is exactly the guarantee a lock would buy you, obtained here for free. Threads in *different* procs can genuinely run in parallel and do not share this guarantee though, but they can coordinate mostly by passing messages on channels to synchronize, rather than by locking shared memory.

One might worry that running a single thread at a time is a serious limitation, but in practice it is not. These threads never perform heavy computation that would monopolize the processor; they only react to events—a key press, a mouse movement, a file request from a client—and hand control back as soon as the event is handled. Such events are driven almost entirely by the user, and humans are slow, so the threads spend most of their time idle and cooperative scheduling never becomes a bottleneck. Crucially, the high-bandwidth work—drawing pixels—does not even reach `rio`: clients talk to the display server directly through `/dev/draw` (e.g., `hellorio` on `/dev/draw/3` in Figure 2.11), so `rio`'s threads only field the comparatively rare mouse and keyboard traffic from its virtual device files, which is human-paced.

The `fileserver` proc and its pipe

The third proc, `fileserver`, creates a *pipe* and reads in a loop one side of the pipe. On the other side of the pipe are clients of `rio`'s filesystem (`hellorio` and the shell process in Figure 2.11). Section 7.5.7 will give more details about this pipe and how `rio` and its "clients" communicate. Moreover, I make this client/server vocabulary precise at the start of Chapter 8.

`main-proc`

The last proc, `main-proc`, is the most important proc of `rio`. It contains many threads. I mentioned already the keyboard and mouse threads above. Both are created by `threadmain()`[58], the first thread of `main-proc`. Moreover, *each window is represented by a thread*. Each window, in addition to being associated with an external process (e.g., the `hellorio` process in Figure 2.11), is also associated with a thread (e.g., `Window-2` in Figure 2.11). Finally, *each opened file of `rio`'s filesystem is represented as a (worker) thread*.

Summary

Figure 2.11 is the single most important diagram in this book. If you understand it, you understand `rio`'s shape: a shared-memory `main-proc` with many cooperative threads (one per window, plus keyboard/mouse/worker-allocator), two dedicated I/O procs that "absorb" blocking reads on `/dev/cons` and `/dev/mouse`, a filesystem proc that runs the 9P loop, and a pool of short-lived worker threads that handle one 9P request each. Everything inside the dashed rectangle shares one address space; everything below it is a separate Plan 9 process communicating via the 9P pipe. Spend a moment matching the windows you see in Figure 2.12 to the window threads in Figure 2.11, and then to the shell and `hellorio` processes to the boxes outside `rio`'s address space.

2.6 Book structure

You now have enough background to understand the source code of `rio`. The rest of the book is organized as follows. I will start by describing the core data structures of `rio` in Chapter 3. Then, I will use a top-down approach, starting with Chapter 4 with the description of `main()` and the core initializations in `rio`. I will continue in Chapter 5 by presenting the main functions called by the threads and procs of `rio`. I will describe also the messages and channels used by those threads. Then, I will describe the code to manage cursors in Chapter 6, before presenting the code of the window manager in Chapter 7. Starting from a mouse click, I will

describe the main functions to create, focus, delete, move, resize, and hide windows. In Chapter 8 I will switch to a bottom-up approach, and switch the focus from `main-proc` to `fileserver`, by presenting the filesystem methods of the `fileserver` proc. Those methods are entry points that offload the work to worker threads and specific functions to handle the different virtual devices of `rio` under `/mnt/wsys/` (e.g., `/mnt/wsys/cons`, `/mnt/wsys/mouse`). I will present the code to serve those virtual devices in Chapter 9. Some virtual devices behave differently depending if the window is a graphical or textual window. This is why I will describe the specifics of graphical windows in Chapter 10, and of textual windows in Chapter 11, including the code of the terminal emulator. Chapter 12 presents the code to serve the other files under `/mnt/wsys` (e.g., `/mnt/wsys/ctl`). Chapter 13 presents advanced functionalities of `rio` that I did not present before to simplify the explanations. Finally, Chapter 14 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `rio` in Appendix A, and code to manage errors in Appendix B. Appendix C contains the code of utility functions used by `rio`, but which are not specific to `rio`. Finally, Appendix E contains examples of graphical applications that extends the window-manager component of `rio`, for instance, a window-switching bar.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter, I will present the core data structures of `rio`, which are essentially: the device handles for the display, the mouse, and the keyboard; the `Window`[49] structure, the central data structure of a windowing system, grouping graphics, input, process, and text information for each window; and the `Filsys`[53a] and `Fid`[53e] structures that support `rio`'s role as a filesystem server, maintaining per-client file state for the 9P protocol.

Note that the original source code of `rio` has been slightly modified to be arguably easier to read. Moreover, in addition to my explanations, I have also sometimes added a few comments in the code itself. The original comments from Rob Pike are using the standard C syntax `/**/` while my comments are using the `//` syntax, so they are easy to differentiate.

3.1 Device handles

`rio` is first of all an ordinary graphical application, so we begin with the device handles it shares with any Plan 9 GUI program, before turning to the structures specific to a windowing system.

3.1.1 Output device: `display` and `view`

`rio` uses the global `display` (set by `initdraw()`, see the Graphics book [Pad16c]) to communicate with the display server, and the global `view` which represents the portion of the screen that `rio` owns (the full screen, unless running recursively inside another `rio`). In turn, `rio` will provide each of its client windows with their own `view`—a sub-portion of `rio`'s screen.

The `display` and `view` globals are not defined here because they are parts of the `libdraw` library and are declared in `draw.h`, but `viewr`[47] below is defined in `rio` and is essentially an alias for `view->r`.

⟨*global* `viewr` 47⟩≡ (339a)
```
Rectangle viewr;
```

3.1.2 Input devices: `mousectl` and `keyboardctl`

Just as `rio` needs a display for output, it needs handles for its input devices. `Mousectl` and `Keyboardctl` (defined in `mouse.h` and `keyboard.h`) are wrappers that internally create a separate proc to read synchronously from `/dev/mouse` and `/dev/cons`, then forward events through a channel. This lets `rio` receive input without blocking

its other threads (see Section 2.3). Both handles are kept in the globals `mousectl`[48a] and `keyboardctl`[48b] shown below.

⟨*global* `mousectl` 48a⟩≡ (338a)
```
Mousectl *mousectl;
```

⟨*global* `keyboardctl` 48b⟩≡ (338a)
```
Keyboardctl *keyboardctl;
```

⟨*global* `mouse` 48c⟩≡ (338a)
```
// alias for &mousectl->m
Mouse *mouse;
```

3.2 The `desktop` base layer

A `Screen`[1] (from `window.h`) is the graphics library's support for overlapping images—it manages the z-order and repainting of layers that overlap each other. `rio` allocates a `Screen` on top of `view` to serve as the "desktop": every window will be a layer on this desktop.

⟨*global* `desktop` 48d⟩≡ (338a)
```
Screen *desktop;
```

When two windows overlap, libdraw's `Screen` is what works out which parts of each window are actually visible and keeps them painted correctly—so `rio`'s own code never has to track the visible regions itself. This is one of the reasons `rio`'s code is so short compared to other windowing systems: the hard part of the job sits in the GRAPHICS book [Pad16c], not here.

The `background`[48e] and `red`[48f] globals below are color images (with `draw` everything is an `Image`, including colors). The red color is used to highlight window borders during move/resize operations.

⟨*global* `background` 48e⟩≡ (338a)
```
Image *background;
```

⟨*global* `red` 48f⟩≡ (338a)
```
Image *red;
```

3.3 Windows

The `Window`[49] structure is where the complexity of `rio` lives. This section presents `Window` and its associated globals: the `windows`[51a] array holding all managed windows, and the `input`[51e] pointer tracking which window has the focus.

3.3.1 The `Window` structure

The `Window`[49] structure is the central data structure of `rio` (much like `Proc` in the KERNEL book [Pad14]). It is the server-side representation of a client window—the client application itself has no notion of a "window"; it simply sees a `display` and a `view` image (through `/dev/winname`).

The structure below groups together nine categories of fields: an ID and label (visible through `/mnt/wsys/winid` and `/mnt/wsys/label`), graphics (the window's image and screen coordinates), mouse and keyboard channels (for receiving dispatched input events), control (for window management messages like resize or delete), a link to

[1]The name `Screen` is inherited from `libdraw`, where it denotes the surface that images are allocated *on*; despite the name, it has nothing to do with the physical screen.

the external process running in the window, configuration flags, and finally the fields specific to textual windows (the text buffer, cursors, and frame rendering) or graphical windows (where the application draws directly).

⟨*struct* `Window` 49⟩≡ (333b)
```
struct Window
{
    //--------------------------------------------------------------------------
    // ID
    //--------------------------------------------------------------------------
    ⟨Window id fields 50a⟩

    //--------------------------------------------------------------------------
    // Graphics
    //--------------------------------------------------------------------------
    ⟨Window graphics fields 50c⟩

    //--------------------------------------------------------------------------
    // Mouse
    //--------------------------------------------------------------------------
    ⟨Window mouse fields 74a⟩

    //--------------------------------------------------------------------------
    // Keyboard
    //--------------------------------------------------------------------------
    ⟨Window keyboard fields 72b⟩

    //--------------------------------------------------------------------------
    // Control
    //--------------------------------------------------------------------------
    ⟨Window control fields 74d⟩

    //--------------------------------------------------------------------------
    // Process
    //--------------------------------------------------------------------------
    ⟨Window process fields 98a⟩

    //--------------------------------------------------------------------------
    // Config
    //--------------------------------------------------------------------------
    ⟨Window config fields 52f⟩

    //--------------------------------------------------------------------------
    // Textual Window
    //--------------------------------------------------------------------------
    ⟨Window textual window fields 51g⟩

    //--------------------------------------------------------------------------
    // Graphical Window
    //--------------------------------------------------------------------------
    ⟨Window graphical window fields 51f⟩

    //--------------------------------------------------------------------------
    // Misc
    //--------------------------------------------------------------------------
    ⟨Window other fields 51d⟩

    //--------------------------------------------------------------------------
    // Extra
    //--------------------------------------------------------------------------
    ⟨Window extra fields 50f⟩
```

```
};
```

I will cover the different field categories gradually throughout this book. For now, the most important fields are id (a unique integer identifying the window, visible through /mnt/wsys/winid) and name (used by the graphics layer to find the window's image via namedimage(), visible through /mnt/wsys/winname).

⟨Window *id fields* 50a⟩≡ (49) 50e ▷

```
int     id;       // visible through /mnt/wsys/winid
char    name[32]; // visible through /mnt/wsys/winname
```

Each window gets a unique id from a global counter id[50b], incremented in wmk()[94c]. This ID appears in the filesystem path /mnt/wsys/<id>/ and identifies the window across the 9P protocol.

⟨*global* id 50b⟩≡ (342b)

```
static int id;
```

The i field below holds the window's public image—the rectangle of pixels visible on screen. This is the image returned by allocwindow() in the Graphics book [Pad16c] library, and it is named after the window (e.g., "window.2") so that the client application can look it up via namedimage(). Most of the time i is a layer (a composited image managed by the draw device), but when a window is hidden it becomes a plain off-screen image, since hidden windows are removed from the layer stack entirely.

⟨Window *graphics fields* 50c⟩≡ (49) 50d ▷

```
// ref_own<Image>, public image for the window (name in /dev/winname)
Image *i;
```

The screenr field below records the window's physical position on screen. The application never sees this value—it works entirely in logical coordinates. rio gives each window logical coordinates by calling originwindow() when it creates or resizes the window (see wmk()): the image's i->r is shifted so that the application's drawable starts at (0,0), regardless of where the window sits on screen. So hellorio sees its content based at (0,0) in i->r, while the window actually sits at, say, screenr = (100,50, 500,350) on screen. The application draws into i using these logical coordinates, and the draw device maps them to the physical screen. Likewise, rio converts mouse coordinates from physical to logical (as we will see in mousethread()[66a]) before forwarding them, so the application receives mouse positions consistent with its own (0,0)-based i->r.

⟨Window *graphics fields* 50d⟩+≡ (49) ◁ 50c

```
// claude: i->r is logical (0,0-based, set via originwindow in wmk); screenr
// claude: is the physical screen position, so the two now differ.
Rectangle screenr; /* screen coordinates of window */
```

⟨Window *id fields* 50e⟩+≡ (49) ◁ 50a 101b ▷

```
char    *label;   // writable through /mnt/wsys/label
```

Several execution contexts touch a Window concurrently. Most of them—the mouse thread, the keyboard thread, each window's own winctl()[71b] thread, and the filesystem worker threads—run inside rio's main proc (see Figure 2.11) and are cooperatively scheduled, so they never preempt one another; in particular only one worker ever runs at a time (the workers exist to keep several requests in flight, not to gain parallelism). The real exception is filsysproc()[75], which reads 9P requests on the server pipe in a separate proc: it runs in parallel with the main proc and itself reaches into Windows while handling some of those requests. Window therefore embeds a Ref for reference counting and a QLock for mutual exclusion.

⟨Window *extra fields* 50f⟩≡ (49) 50g ▷

```
Ref;
```

⟨Window *extra fields* 50g⟩+≡ (49) ◁ 50f

```
QLock;
```

The QLock guards against that genuine cross-proc parallelism: filsysproc() and a main-proc thread could otherwise update the same Window at the same instant. The Ref keeps a Window alive while a pointer to it is still held across a blocking point—a worker may be parked on a channel, or filsysproc() partway through a request, when the mouse thread runs a delete; without the reference count the Window could be freed under their feet (see Libcore book [Pad16a] for the details of the Ref and QLock synchronization primitives).

3.3.2 All windows

The global `windows`[51a] array is the essence of the windowing system: it holds all the managed windows. The `topped`[51c] counter is a logical timestamp—each time a window gets the focus, it receives the current `topped` value (then incremented). This allows `wpointto()`[69] to find the topmost window at a given point by comparing `Window.topped` values across all overlapping windows.

⟨*global* `windows` 51a⟩≡ (338a)
```
// growing_array<ref_own<Window>> (allocated = nwindow+1)
Window **windows;
```

⟨*global* `nwindow` 51b⟩≡ (338a)
```
int nwindow;
```

⟨*global* `topped` 51c⟩≡ (342b)
```
static int  topped;
```

⟨`Window` *other fields* 51d⟩≡ (49) 96g ▷
```
int   topped;
```

3.3.3 The current window: input

The `input`[51e] global plays a role similar to `up` in the KERNEL book [Pad14]: just as `up` always points to the currently running process, `input` always points to the window that has the keyboard focus—the one that receives typed characters. It is `nil` when no window has focus (e.g., when the mouse is on the desktop background).

⟨*global* `input` 51e⟩≡ (338a)
```
//option<ref<Window>>, the window with the focus! the window to send input to
Window *input;
```

3.3.4 Graphical windows

As mentioned earlier, `rio` distinguishes graphical from textual windows at runtime: if a client opens `/dev/mouse`, `Window.mouseopen`[51f] is set to `true` and `rio` treats the window as graphical (forwarding raw mouse events). Otherwise, the window is textual and `rio` provides the built-in terminal emulator with text selection, scrolling, and editing.

⟨`Window` *graphical window fields* 51f⟩≡ (49) 164a ▷
```
bool mouseopen;
```

3.3.5 Textual windows

A textual window (the default, used as a terminal) maintains a rune buffer (`r`/`nr`) holding the full text content, two cursor positions (`q0`/`q1`) for the selection, an origin (`org`) indicating which rune starts the visible portion, and an "output point" (`qh`) that separates text already sent by the application from text typed by the user.

⟨`Window` *textual window fields* 51g⟩≡ (49)
```
⟨Window textual window fields, text data 51h⟩
⟨Window textual window fields, text cursors 52a⟩
⟨Window textual window fields, visible text 52b⟩
⟨Window textual window fields, graphics 52d⟩
```

⟨`Window` *textual window fields, text data* 51h⟩≡ (51g)
```
// growing_array<Rune> (allocated = Window.maxr, used = nr)
Rune  *r;
uint  nr; /* number of runes in window */
uint  maxr; /* number of runes allocated in r */
```

⟨Window *textual window fields, text cursors* 52a⟩≡ (51g) 52c ▷

```
// index in Window.r
uint  q0; // cursor, where entered text go (and selection start)
// index in Window.r
uint  q1; // selection end or same value than q0 when no selection
```

⟨Window *textual window fields, visible text* 52b⟩≡ (51g)

```
// index in Window.r
uint  org;
```

⟨Window *textual window fields, text cursors* 52c⟩+≡ (51g) ◁ 52a

```
// index in Window.r
uint  qh; // output point
```

These fields are easier to picture on a concrete buffer. Suppose the terminal has already scrolled through some history, a command has just printed the date, the shell has printed a fresh prompt, and the user has typed (but not yet "entered") the start of the next command:

```
r[] holds, left to right:

  [ ...earlier output... ][ Mon Jun 22\n%  ][ echo hi ]
    r[0 .. org)             r[org .. qh)      r[qh .. nr)
    scrolled off the top    visible program   typed by the user,
                            output + prompt    not yet sent
                                                        ^
                                                        q0 == q1

all markers are indices into r[] (nr runes total):
  org    -> first rune drawn at the top of the window;
            scrolling only moves org.
  qh     -> output point: programs wrote r[0..qh), the user typed
            r[qh..nr).
  q0,q1  -> the selection r[q0..q1); here q0==q1, so it is just the
            text cursor, sitting at the end of what was typed.
```

The `Frame` structure (from the `libframe` library, see Appendix D) handles the rendering of visible text—wrapping, selection highlighting, and scrollbar interaction.

⟨Window *textual window fields, graphics* 52d⟩≡ (51g) 52e ▷

```
Frame frm;
```

`scrollr` is the rectangle reserved for the window's scrollbar, to the left of the text `Frame` (see Figure 2.12). The `scrolling` flag below is a plain `bool` because it is just an on/off mode, `false` by default: when set, the window scrolls automatically to follow new output, so a program writing to `/dev/cons` never blocks waiting for the user to scroll down (see the auto-scroll extension in Section 11.8).

⟨Window *textual window fields, graphics* 52e⟩+≡ (51g) ◁ 52d

```
Rectangle scrollr;
```

⟨Window *config fields* 52f⟩≡ (49) 107a ▷

```
bool scrolling;
```

3.4 Filesystem server

`rio` is not just a graphical application—it is also a filesystem server. In Plan 9, any process can act as a filesystem server by listening on a pipe for 9P messages (see the NETWORK book [Pad16d]). `rio` serves the `/mnt/wsys/` namespace: when a client application opens `/mnt/wsys/cons` or `/mnt/wsys/mouse`, those accesses are translated into 9P messages sent through a pipe to `rio`, which dispatches them to the appropriate window.

3.4.1 Server state: `Filsys` and `filsys`

The `Filsys`[53a] structure holds the two ends of the pipe (`cfd` for the client side, `sfd` for the server side), the user name (for security checks), a hash table of `Fid`[53e] entries tracking per-client file state, and more.

⟨*struct* `Filsys` 53a⟩≡ (333b)

```
struct Filsys
{
    // client
    fdt  cfd;
    // server
    fdt  sfd;

    // ref_own<string>
    char *user;

    // map<fid, Fid> (next in bucket = Fid.next)
    Fid  *fids[Nhash];

    ⟨Filsys other fields 56a⟩
};
```

Uses `Nhash` 53c.

The global `filsys` holds `rio`'s file-server state: the pipe it listens on and the table of files currently open. It is kept global mainly for `winshell()`[97d], the code that launches a shell in a new window—it calls `filsysmount()`[99c] to attach the new window to `/mnt/wsys/`, and needs the global to find where the server pipe is.

⟨*global* `filsys` 53b⟩≡ (338a)

```
Filsys *filsys;
```

`rio` tracks its open files with the classic C hash-table idiom found throughout Principia Softwarica: a fixed array of (`Nhash`) buckets above, where each bucket is the head of a singly-linked list of `Fid` structures chained through their `next` field below. To find a file, `rio` hashes its fid number down to a bucket index and walks that short list.

⟨*constant* `Nhash` 53c⟩≡ (333b)

```
#define Nhash 16
```

⟨`Fid` *extra fields* 53d⟩≡ (53e) 54a ▷

```
// list<Fid> (head = Filsys.fids[i])
Fid  *next;
```

3.4.2 File state: `Fid`

A fid (file ID) is a 9P concept: a 32-bit handle the client uses to refer to a file on the server, similar to a file descriptor but at the protocol level. A `Qid` is the server-side file identifier, analogous to an inode number in Unix—it uniquely identifies a file on the server. While fids are chosen by the client and may be reused, Qids are assigned by the server and remain stable for a given file. The server must maintain state for each fid (whether it is open, in what mode, which `Qid` identifies the underlying file).

⟨*struct* `Fid` 53e⟩≡ (333b)

```
struct Fid
{
    // the key
    int  fid;

    // the value
    Qid  qid;
    // the state
```

```
    bool open;
    int  mode;

    ⟨Fid other fields 55a⟩

    // Extra
    ⟨Fid extra fields 53d⟩
};
```

Note that the 9P "client" I am refering to in this section is not exactly the window application itself—it is the kernel acting on behalf of the application. When `hellorio` calls `read()` on `/dev/cons`, the kernel translates that into a 9P message (`Tread`) sent through the pipe to `rio`'s `filsysproc()`[75].

The `busy` flag says whether a `Fid` slot is currently in use. `rio` recycles `Fid` structures rather than freeing them, so `newfid()`[54b] looks for a slot with `busy` cleared to reuse; it is set when a client first references a fid and cleared again when the client is done with it.

⟨`Fid` *extra fields* 54a⟩+≡ (53e) ◁53d

```
bool busy;
```

Before a client can touch a file it must obtain a `Fid` for it. `newfid()` is the lookup-or-allocate routine that bridges a 9P fid number and `rio`'s in-memory `Fid`: it hashes the number into the right bucket and returns the matching `Fid` if one already exists, otherwise reusing a free slot or allocating a fresh one.

⟨*function* `newfid` 54b⟩≡ (346c)

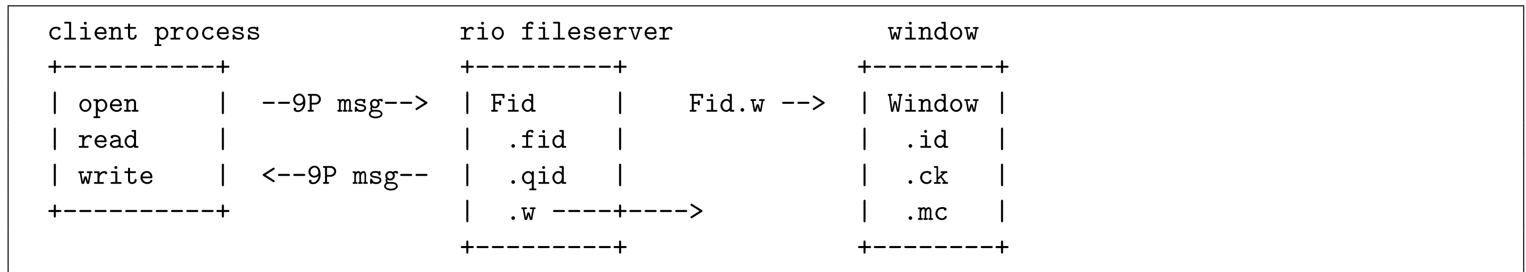
```
/// filsysproc | filsyswalk -> <>
Fid*
newfid(Filsys *fs, int fid)
{
    Fid *f, *ff, **fh;

    ff = nil; // free fid

    // lookup_hash(fid, fs->fids)
    fh = &fs->fids[fid&(Nhash-1)];
    for(f=*fh; f; f=f->next) {
        if(f->fid == fid)
            // found!
            return f;
        else if(ff==nil && !f->busy)
            ff = f;
    }
    // else
    if(ff){
        ff->fid = fid;
        return ff;
    }
    // else

    f = emalloc(sizeof(Fid));
    f->fid = fid;

    // insert_hash(f, fs->fids)
    f->next = *fh;
    *fh = f;

    return f;
}
```

Uses `Nhash` 53c.

The `Fid.w` field below closes the circle. A `Qid`'s `path` is not opaque here: `rio` packs the window id in its high bits and the kind of device file (`cons`, `mouse`, `wctl`, ...) in its low byte, so the `qid` alone says both which window and which of its files a fid refers to. During attach or walk (shown in Chapter 8), `rio` reads that window id back out, resolves it once with `wlookid()`[128a], and caches the resulting `Window` pointer in `w` (taking a reference). Later operations follow `f->w` directly rather than rescanning the `windows` array, so `w` is not independent state—it is just the `qid`'s window id, already resolved.

⟨`Fid` *other fields* 55a⟩≡ (53e) 123c ▷
```
  Window *w;
```

The `Fid.w` field is the bridge between the filesystem abstraction and the windowing system. When a "client" process accesses a file under `/mnt/wsys/`, the 9P protocol resolves the path to a `Fid`, and `Fid.w` points to the `Window` that owns that file:



3.4.3 Worker request on a `Fid`: `Xfid`

The filesystem server proc (`filsysproc()`[75]) reads 9P requests from the pipe, but it cannot process them all synchronously—multiple windows and client processes may be making concurrent requests. So `filsysproc()` acts as a *master* that dispatches each request to a *worker* thread. The `Xfid` ("extended fid") structure combines the incoming 9P parsed request (`Fcall`) with a channel to a worker thread.

⟨*struct* `Xfid` 55b⟩≡ (333b)
```
  struct Xfid
  {
          // incoming parsed request
          Fcall req;
          // answer buffer
          byte *buf;

          // handler to worker thread
          // chan<void(*)(Xfid*)> (listener = xfidctl, senders = filsysxxx)
          Channel *c; /* chan(void(*)(Xfid*)) */

          // ref<Fid>
          Fid *f;

          Filsys *fs;

          ⟨Xfid flushing fields 266a⟩
          ⟨Xfid other fields 266c⟩

          // Extra
          Ref;
          ⟨Xfid extra fields 78c⟩
  };
```

`rio` only ever has one `Filsys`[53a] (the global `filsys`[53b]), so `Xfid`'s `fs` field always points to it and could indeed be replaced by the global. It is carried in each `Xfid` anyway so that a worker has everything it needs to answer a request—including where to write the reply—without reaching for a global.

The master sends a function pointer through `Xfid.c`, telling the worker what operation to execute (e.g., `xfidopen()`[133a], `xfidread`[136b]). Workers are pooled via the `cxfidalloc`[62b]/`cxfidfree`[62c] channels to avoid creating a new thread for every request.

⟨`Filsys` *other fields* 56a⟩≡ (53a)

```
// chan<ref<Xfid>> (listener = filsysproc, sender = xfidallocthread)
Channel *cxfidalloc; /* chan(Xfid*) */
```

3.4.4 9P callbacks: `fcall`

The `fcall`[56b] dispatch table below maps 9P message types (`Tattach`, `Twalk`, `Topen`, `Tread`, `Twrite`, etc.) to handler functions. When a client process executes a file operation (open, read, write) on a file under `/mnt/wsys/`, the kernel translates that into a 9P message (prefixed with `T` for "transmit"), which `filsysproc()`[75] dispatches through this table.

⟨*global* `fcall` 56b⟩≡ (346c)

```
Xfid*  (*fcall[Tmax])(Filsys*, Xfid*, Fid*) =
{
    [Tattach]  = filsysattach,

    [Twalk]    = filsyswalk,

    [Topen]    = filsysopen,
    [Tclunk]   = filsysclunk,
    [Tread]    = filsysread,
    [Twrite]   = filsyswrite,
    [Tstat]    = filsysstat,

    ⟨fcall other methods 76f⟩
};
```

Uses `filsysattach()` 126, `filsysclunk()` 133c, `filsysopen()` 132a, `filsysread()` 135a, `filsysstat()` 138a, `filsyswalk()` 129, and `filsyswrite()` 137a.

The handler may return `nil` if it delegates work to a worker thread, or the `Xfid` back if the worker can be reused immediately.

Each `filsysxxx` handler implements one 9P message type (`filsyswalk()`[129], `filsysread()`[135a], `filsysopen()`[132 and so on); we meet them gradually in Chapter 8.

3.5 Putting it together: `rio`'s data structures

Before `main()` (actually `threadmain()`[58]) starts wiring all those structures together, it is worth seeing how they relate on a concrete example. Rather than listing fields in the abstract, Figure 3.1 shows the instances `rio` holds while two windows are open: a focused terminal and a graphical client overlapping it. A `*` marks a pointer field that an arrow follows.

The three groups mirror `rio`'s three roles. The globals are the handles it holds as a graphical application (the `display`, its `view`, the `desktop`[48d] layers, and the mouse and keyboard readers); the `windows`[51a] array is the heart of the windowing system, one `Window`[49] per managed window; and the `Filsys`[53a]/`Fid`[53e] table is the per-client state `rio` keeps as a 9P file server. Notice where the two sides meet: a `Fid`'s `qid` encodes the window id, and its `w` field caches the `Window` that id resolves to (as we saw above), so a request from a client is routed straight to the right window. The lone `Xfid` at the bottom is a worker caught mid-request, serving the blocking `/dev/mouse` read on `fid` 7 through its `f` back-pointer.

```
GLOBALS    display *--> /dev/draw connection      (set by initdraw())
           view    *--> Image of rio's whole screen
           desktop *--> Screen (manager of overlapping layers, on view)
           mousectl    *--> proc reading /dev/mouse -.
           keyboardctl *--> proc reading /dev/cons  -+--> event channels
           filsys  *--> Filsys (the 9P server end of the pipe)
           input   *--> windows[0]                    (focused window)

WINDOWS    windows *--> [ windows[0], windows[1] ]    nwindow = 2
                          *           *
                          |           |
                          v           v
          +-----------------------+ +-----------------------+
          | Window (terminal)     | | Window (graphical)    |
          | id=1 name="window.1"  | | id=2 name="window.2"  |
          | topped=7 (focused)    | | topped=5              |
          | mouseopen=false       | | mouseopen=true        |
          | i *--> layer on       | | i *--> layer (overlaps|
          |        desktop        | |        windows[0])    |
          | screenr=(0,0,600,400) | | screenr=(300,200,...) |
          | r,nr *--> rune buffer | +-----------------------+
          | org=0 qh=13 q0=q1=19  |
          +-----------------------+

FILESYSTEM filsys *--> Filsys
           fids[h] *--> Fid{fid=4} --> Fid{fid=7} --> nil   (a hash chain)
                          *               *
                          v               v
          +--------------------------+ +--------------------------+
          | Fid fid=4                | | Fid fid=7                |
          | qid.path=QID(1,Qcons)    | | qid.path=QID(2,Qmouse)   |
          |   -> /dev/cons of win 1  | |   -> /dev/mouse of win 2 |
          | w *--> windows[0]        | | w *--> windows[1]        |
          +--------------------------+ +--------------------------+
                                                  ^
                                                  | x->f
                                       +--------------------------+
                                       | Xfid (in-flight worker)  |
                                       | req = Tread (mouse,tag=5)|
                                       | f *--> Fid{fid=7}        |
                                       +--------------------------+
```

Figure 3.1: `rio`'s live data structures, with two windows open.

Chapter 4

main()

I now switch from the bottom-up approach of Chapter 3 to a top-down approach: I will describe in the following chapters the main functions of rio, starting in this chapter with threadmain()[58], the entry point of rio.

4.1 threadmain() skeleton

The entry point of rio is threadmain()[58], not main(): the thread library provides its own main() that sets up the cooperative scheduler and a first proc, then calls threadmain() as its initial thread (see the LIBCORE book [Pad16a]). The initialization below follows three phases matching rio's three roles: first as a graphical application (connecting to the display, mouse, and keyboard), then as a concurrent application (creating channels and threads), and finally as a filesystem server (setting up the 9P pipe and workers). Once everything is ready, threadmain() blocks on exitchan[60e], waiting for the user to select "Exit" from the right-click menu.

```
⟨function threadmain 58⟩≡                                        (338b)
  /// main -> threadcreate(<>)
  void threadmain(int argc, char *argv[])
  {
      ⟨main() locals 96i⟩

      ARGBEGIN{
      ⟨main() command line processing 225d⟩
      }ARGEND

      ⟨main() set some globals 96j⟩

      // Rio, a graphical application

      ⟨main() graphics initializations 59b⟩

      // Rio, a concurrent application

      ⟨main() communication channels creation 60f⟩
      ⟨main() threads creation 61b⟩

      // Rio, a filesystem server

      filsys = filsysinit(xfidinit());
      ⟨main() if filsys is nil 59a⟩
      else{
          ⟨main() error management after everything setup 219b⟩

          // blocks until get exit message on exitchan
          recv(exitchan, nil);
```

```
    }
    killprocs();
    threadexitsall(nil);
}
```

In the rest of this book, I will usually not comment the error-management code. Such code is often trivial (but necessary). Appendix B explains the (small) error-management infrastructure used in `rio`.

⟨`main()` *if filsys is nil* 59a⟩≡ (58)

```
if(filsys == nil)
    fprint(STDERR, "rio: can't create file system server: %r\n");
```

4.2 Graphics initialization

The graphics initialization is almost identical to `hellorio.c` (Section 2.3): `rio` calls `geninitdraw()` to connect to the display server, then allocates a `Screen` on top of `view` to serve as the desktop for overlapping windows. The call to `iconinit()`[59d] allocates the grey background and the red color used to highlight borders during window operations.

⟨`main()` *graphics initializations* 59b⟩≡ (58)

```
if(geninitdraw(nil, derror, nil, "rio", nil, Refnone) < 0){
    fprint(STDERR, "rio: can't open display: %r\n");
    exits("display open");
}
viewr = view->r;

iconinit(); // allocate background and red images

⟨main() mouse initialisation 60a⟩
⟨main() keyboard initialisation 60c⟩

desktop = allocscreen(view, background, false);
⟨main() sanity check desktop 59c⟩

draw(view, viewr, background, nil, ZP);
flushimage(display, true);
```

⟨`main()` *sanity check* `desktop` 59c⟩≡ (59b)

```
if(desktop == nil)
    error("can't allocate desktop");
```

⟨*function* `iconinit` 59d⟩≡ (341)

```
/// threadmain -> <>
void
iconinit(void)
{
    background = allocimage(display, Rect(0,0,1,1), RGB24, true, 0x777777FF);
    red        = allocimage(display, Rect(0,0,1,1), RGB24, true, 0xDD0000FF);
}
```

Uses `background` 48e and `red` 48f.

The reason for using `geninitdraw()` above rather than `initdraw()` like in `hellorio.c` is that `rio` needs to supply its own error handler (`derror()`[282d]): `initdraw()` installs a default handler that exits on errors, but a windowing system must recover gracefully.

4.3 Mouse initialization

This is the same setup any Plan 9 graphical program does—`hellorio` (Section 2.3) opened its mouse the same way. `rio` is a graphical application first, so before it can manage other windows it acquires its own mouse (and, next, keyboard).

```
⟨main() mouse initialisation 60a⟩≡                                    (59b)
  mousectl = initmouse(nil, view);
  ⟨main() sanity check mousectl 60b⟩
  mouse = &mousectl->m;
```

`initmouse()` takes `view` as an argument to store it internally in the `Mousectl` structure, so that later when code calls `readmouse()` with this control, it can reach for the display and flush any buffered draw commands before blocking on the mouse channel. This matters because drawing in Plan 9 is buffered (see the GRAPHICS book [Pad16c]): before reading mouse input, it is better to ensure the display is up to date with what the program has drawn so far.

```
⟨main() sanity check mousectl 60b⟩≡                                   (60a)
  if(mousectl == nil)
      error("can't find mouse");
```

4.4 Keyboard initialization

Keyboard initialization mirrors mouse initialization: `initkeyboard()` opens `/dev/cons`, spawns an I/O proc that reads runes and forwards them on a channel, and returns a `Keyboardctl` handle.

```
⟨main() keyboard initialisation 60c⟩≡                                 (59b)
  keyboardctl = initkeyboard(nil);
  ⟨main() sanity check keyboardctl 60d⟩
```

The split between a reader proc and a channel exists for the same reason as on the mouse side: reading from `/dev/cons` is a blocking syscall, and running it in a thread would stall every other thread in the same proc. Putting the blocking read in its own proc isolates the stall and lets the thread scheduler keep serving the rest of `rio`.

```
⟨main() sanity check keyboardctl 60d⟩≡                                (60c)
  if(keyboardctl == nil)
      error("can't find keyboard");
```

4.5 Channels creation

`rio`'s threads communicate almost exclusively through channels rather than shared variables protected by locks. The `exitchan`[60e] is the simplest example: `threadmain()`[58] blocks on a `recv()` on this channel, and the mouse thread sends a message when the user selects "Exit".

```
⟨global exitchan 60e⟩≡                                                (338a)
  // chan<unit> (listener = threadmain, sender = mousethread(Exit) | ?)
  Channel *exitchan; /* chan(int) */
```

```
⟨main() communication channels creation 60f⟩≡                    (58) 107e▷
  exitchan     = chancreate(sizeof(int), 0);
```

The `0` in `chancreate(sizeof(int), 0)` is the channel's *buffer size*: zero means an *unbuffered* (synchronous) channel, so a send blocks until a receiver is ready and vice versa. That is exactly right for a one-shot "Exit" rendezvous, and contrasts with the buffered `keyboardctl->c` (capacity 20) we saw earlier, where keystrokes had to be able to queue up.

Many more channels will be created later—each new window gets its own set of channels for keyboard, mouse, and control messages.

4.6 Threads creation

`main()` creates only two threads at startup: one for the keyboard and one for the mouse.

⟨*constant* `STACK` 61a⟩≡ (333b)
```
#define STACK 8192
```

⟨`main()` *threads creation* 61b⟩≡ (58) 108a ▷
```
threadcreate(keyboardthread, nil, STACK);
threadcreate(mousethread, nil, STACK);
```

These are dispatcher threads that listen for input events and route them to the appropriate window thread. These threads are distinct from the I/O procs that `initmouse()` and `initkeyboard()` create internally (see Figure 2.11): the I/O procs block on `read()` from device files and forward raw events onto channels, while these dispatcher threads receive from those channels and route events to the correct window.

Additional threads are created dynamically: each new window gets its own `winctl()`[71b] thread, the worker allocator gets a thread, and each filesystem worker gets a thread.

4.7 Filesystem server initialization

The filesystem server is initialized by first creating the worker allocator (`xfidinit()`[63]), then setting up the server itself (`filsysinit()`[61c]).

4.7.1 `filsysinit()`

`filsysinit()`[61c] creates the two-way pipe that will carry 9P messages between clients and the server, then spawns `filsysproc()`[75] as a separate proc (not a thread, because it blocks on `read()` of the pipe).

⟨*function* `filsysinit` 61c⟩≡ (346b)
```
/// threadmain -> <>
Filsys*
filsysinit(Channel *cxfidalloc)
{
    int pid;
    Filsys *fs;
    ⟨filsysinit() other locals 258d⟩

    ⟨filsysinit() install dumper 278b⟩
    fs = emalloc(sizeof(Filsys));
    if(cexecpipe(&fs->cfd, &fs->sfd) < 0)
        goto Rescue;

    ⟨filsysinit() set clockfd 138d⟩
    ⟨filsysinit() set fs user 269f⟩
    pid = getpid();

    fs->cxfidalloc = cxfidalloc;

    ⟨filsysinit() wctl pipe, process, and thread creation 258c⟩
    proccreate(filsysproc, fs, 10000);

    ⟨filsysinit() srv pipe 256b⟩

    return fs;

Rescue:
    free(fs);
```

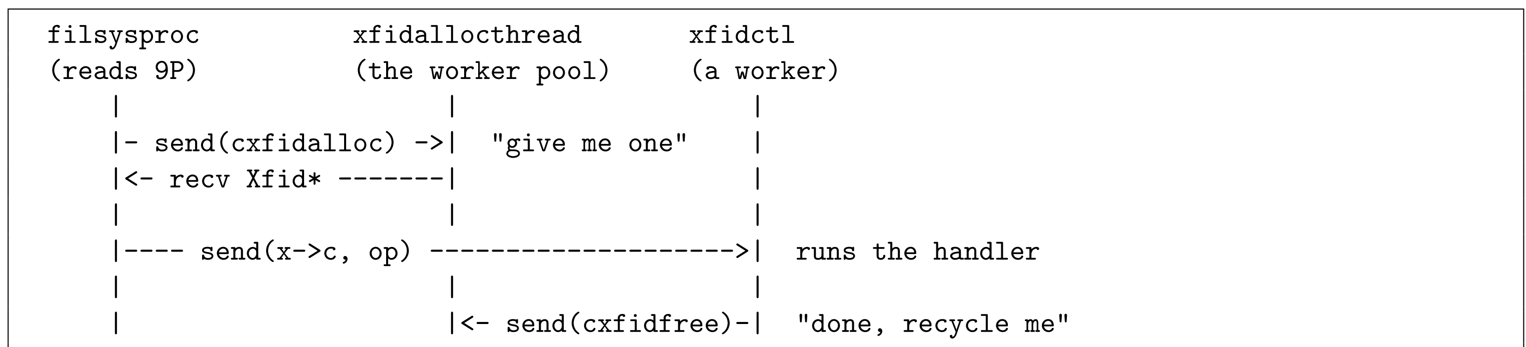
```
        return nil;
    }
```
Uses `cexecpipe()` 62a and `filsysproc()` 75.

The pipe connecting clients to `rio`'s filesystem server has two ends: the client side (`cfd`, shared with child processes for 9P communication) and the server side (`sfd`, read by `filsysproc()`). The child must close the server end before exec'ing, otherwise the child would hold an extra reference to `sfd`. If `filsysproc()` were to exit, the pipe would not fully hang up—because the child's copy of `sfd` keeps it alive—and the child's 9P requests through `cfd` would block forever waiting for responses from a dead server. The server end is explicitly closed in `filsysmount()`[99c], but `OCEXEC` used below provides an additional safety net. The standard `pipe()` syscall cannot set per-end flags like `OCEXEC`, so `cexecpipe()`[62a] binds the kernel pipe device `#|` directly and opens the two endpoints (`data` and `data1`) independently with different flags.

⟨*function* `cexecpipe` 62a⟩≡ (346b)
```
  /*
   * Build pipe with OCEXEC set on second fd.
   * Can't put it on both because we want to post one in /srv.
   */
  errorneg1
  cexecpipe(fdt *p0, fdt *p1)
  {
      /* pipe the hard way to get close on exec */
      if(bind("#|", "/mnt/temp", MREPL) < 0)
          return ERROR_NEG1;
      *p0 = open("/mnt/temp/data", ORDWR);
      // p1 = sfd = server end of the pipe closed automatically before exec for child
      *p1 = open("/mnt/temp/data1", ORDWR|OCEXEC);
      unmount(nil, "/mnt/temp");
      if(*p0<0 || *p1<0)
          return ERROR_NEG1;
      return OK_0;
  }
```

4.7.2 Worker allocator: `xfidinit()`

Rather than creating a new thread for every 9P request, `rio` *pools* worker threads via two channels: `cxfidalloc`[62b] to request a worker, and `cxfidfree`[62c] to return one. The allocator thread manages the pool, creating new workers only when needed:



⟨*global* `cxfidalloc` 62b⟩≡ (340a)
```
  // chan<ref<Xfid>> (listener = filsysproc, sender = xfidallocthread)
  static Channel *cxfidalloc; /* chan(Xfid*) */
```

⟨*global* `cxfidfree` 62c⟩≡ (340a)
```
  // chan<ref<Xfid>> (listener = xfidallocthread, sender = xfidctl worker)
  static Channel *cxfidfree; /* chan(Xfid*) */
```

⟨*function* `xfidinit` 63⟩≡ (340a)

```
/// threadmain -> <>
Channel*
xfidinit(void)
{
    cxfidalloc = chancreate(sizeof(Xfid*), 0);
    cxfidfree = chancreate(sizeof(Xfid*), 0);
    threadcreate(xfidallocthread, nil, STACK);
    return cxfidalloc;
}
```

Uses `STACK` 61a, `cxfidalloc-53` 62b, `cxfidfree-54` 62c, and `xfidallocthread()` 78d.

Chapter 5

Procs and Threads

A windowing system is fundamentally a *concurrent reactive* program: it must respond to mouse movements, keyboard presses, window management requests, and filesystem operations, all happening concurrently. `rio` handles this with a *producer/consumer* architecture built on threads and channels as shown in Figure 2.11. Two dispatcher threads (keyboard and mouse) listen for hardware events and route them to the appropriate window thread. Each window has its own thread (`winctl()`[71b]) that processes events in its own event loop. The filesystem server runs as a separate proc (because it blocks on I/O) and dispatches 9P requests to worker threads. This is similar to how the kernel handles interrupts and processes: each thread blocks waiting on a channel, and the scheduler yields to whichever thread has work to do.

5.1 Keyboard thread

The keyboard thread is a simple forwarder: it receives runes from the I/O proc (via `keyboardctl->c`) and passes them to the `input`[51e] window's `ck` channel—always the currently focused window. There is no per-rune routing decision to make.

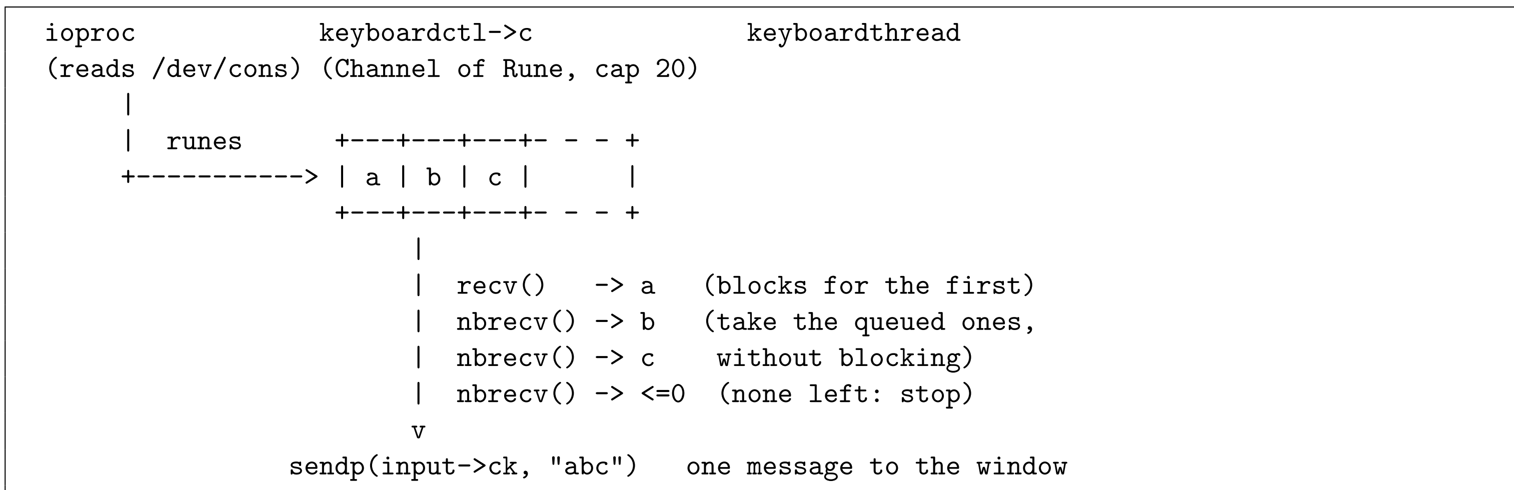
⟨*function* `keyboardthread` 64⟩≡ (339b)

```
/// threadmain -> threadcreate(<>, nil)
void
keyboardthread(void*)
{
    Rune buf[2][20];
    // points to buf[0] or buf[1]
    Rune *rp;
    int n, i;

    threadsetname("keyboardthread");

    n = 0;
    for(;;){
        rp = buf[n];
        n = 1-n;

        // Listen
        recv(keyboardctl->c, rp);

        for(i=1; i<nelem(buf[0])-1; i++)
            if(nbrecv(keyboardctl->c, rp+i) <= 0)
                break;
        rp[i] = L'\0';

        if(input != nil)
            // Dispatch, to "current" window thread!
```

```
            sendp(input->ck, rp);
    }
}
```

Uses `input` 51e and `keyboardctl` 48b.

Note that `keyboardctl->c` is a *buffered* channel: `initkeyboard()` creates it with `chancreate(sizeof(Rune), 20)` (see the GRAPHICS book [Pad16c]), so up to 20 runes can sit queued in it with no receiver waiting. `nbrecv()` is a *non-blocking* receive: it takes a value if one is already there, otherwise returns immediately (`<= 0`) rather than blocking. The thread combines the two to *batch* keystrokes: it does one blocking `recv()` for the first rune, then drains whatever else is already queued with `nbrecv()`, packing the runes into a single buffer forwarded to the window in one `sendp()`:



Without `nbrecv()` the code would still be correct—it could loop back and `recv()` each rune one at a time—but it would wake the focused window's thread once per keystroke. Batching coalesces a burst of typing (or a paste) into a single hand-off, which is the optimization worth making here.

The double-buffering trick (alternating between `buf[0]` and `buf[1]`) avoids overwriting the sent buffer before the receiver has consumed it, since `sendp()` transmits a pointer.

5.2 Mouse thread

The mouse thread is far more complex than the keyboard thread because mouse events have a dual nature: some belong to the windown application (e.g., left-click for text selection) and some belong to the window manager (e.g., right-click for the system menu, border clicks for move/resize). The mouse thread must decide for each event whether to dispatch it to the window thread (`sending = true`) or handle it locally as a windowing system operation.

The enum below indexes into an `Alt` array, just like in `hellorio.c` (Section 2.3.2). For now, `MMouse` is the only real entry, which makes `alt()` seem like overkill for a simple `recv()`—but the `Alt` array will be extended later with additional channels (e.g., for resize events).

⟨*enum* `Mxxx` 65a⟩≡ (339a)

```
enum {
    MMouse,
    ⟨Mxxx cases 220b⟩
    NALT
};
```

⟨`mousethread()` *locals* 65b⟩≡ (66a) 66c ▷

```
// map<enum<Mxxx>, Alt>
static Alt alts[NALT+1];
```

Uses `NALT-7` 65a.

We can now see the skeleton of `mousethread()`[66a]: like `hellorio`, it is an `alt()` loop (the thread-library analogue of UNIX `select()`) waiting on channels. For now there is a single case, `MMouse`, whose handling is filled in below; we will see more cases added gradually in later sections.

```
⟨function mousethread 66a⟩≡                                        (339a)
  /// threadmain -> threadcreate(<>, nil)
  void
  mousethread(void*)
  {
      ⟨mousethread() locals 65b⟩

      threadsetname("mousethread");

      ⟨mousethread() alts setup 66b⟩
      alts[NALT].op = CHANEND;

      for(;;)
          // message loop
          switch(alt(alts)){
          ⟨mousethread() event loop cases 66e⟩
          }
  }
```

Uses `NALT-7` 65a.

```
⟨mousethread() alts setup 66b⟩≡                                (66a)  220c ▷
  // listen
  alts[MMouse].c = mousectl->c;
  alts[MMouse].v = &mousectl->m;
  alts[MMouse].op = CHANRCV;
```

Uses `MMouse-5` 65a and `mousectl` 48a.

```
⟨mousethread() locals 66c⟩+≡                              (66a)  ◁ 65b  66d ▷
  Window *winput;
  Point xy; // logical coord
```

```
⟨mousethread() locals 66d⟩+≡                              (66a)  ◁ 66c  68b ▷
  bool sending = false;
```

The `MMouse` case below is the most complex event handler in `rio`. Two key variables drive the logic: `winput` captures which window the mouse is currently over (which may differ from `input`[51e], the keyboard-focused window), and `xy` holds the mouse coordinates converted from screen-relative (physical) to window-relative (logical). The rest is a cascade of conditions deciding whether to scroll, move, resize, or forward the event to the window—I will defer the details to later sections since each case involves its own protocol.

```
⟨mousethread() event loop cases 66e⟩≡                          (66a)  220d ▷
  case MMouse:
      ⟨mousethread() if wkeyboard and button 6 227c⟩
  Again:
      winput = input;
      ⟨mousethread() if wkeyboard and ptinrect 227d⟩

      if(winput != nil && winput->i != nil){
          /* convert to logical coordinates */
          xy.x = mouse->xy.x + (winput->i->r.min.x - winput->screenr.min.x);
          xy.y = mouse->xy.y + (winput->i->r.min.y - winput->screenr.min.y);

          ⟨mousethread() goto Sending if scroll buttons 201d⟩

          inside = ptinrect(mouse->xy, insetrect(winput->screenr, Selborder));
```

```
        ⟨mousethread() set scrolling 201c⟩
        ⟨mousethread() set moving to true for some conditions 90c⟩
        else
            ⟨mousethread() set sending to true for some conditions 68a⟩
    }else
        sending = false;

    // Application mouse event
    ⟨mousethread() if sending 68c⟩
    // Windowing system mouse event
    ⟨mousethread() if not sending 68f⟩

  ⟨mousethread() Drain label 70d⟩
```
Uses MMouse-5 65a, Selborder 67, input 51e, and mouse 48c.

The conversion code above translates mouse coordinates from physical screen space to the window's logical space, so the application receives clicks in the same coordinate system it draws in. `rio` sets up that system with `originwindow()` when it creates the window (see `wmk()`[94c]): `screenr` holds the physical position while `i->r` is based at (0,0):

```
screen (view)
+------------------------------------------+
|                                          |
|    screenr = (100,50, 500,350)           |
|    i->r    = (0,0, 400,300)              |
|    +--------------------------------+    |
|    |                                |    |
|    |        X  <-- mouse click      |    |
|    |                                |    |
|    +--------------------------------+    |
|                                          |
+------------------------------------------+

mouse->xy = (250, 200)    (physical, from /dev/mouse)

xy.x = 250 + (0 - 100) = 150
xy.y = 200 + (0 - 50)  = 150

xy = (150, 150)           (logical, sent to hellorio)
```

`Selborder` is the width, in pixels, of the border drawn around the selected window. It is used just above to compute `inside`: the pointer counts as "inside" the window only if it falls within `screenr` shrunk by `Selborder` (`insetrect()`), so clicks landing on the border itself are reserved for window-management actions (like resizing) rather than forwarded to the application.

```
⟨constant Selborder 67⟩≡                                          (333a)
  Selborder  = 4, /* border of selected window */
```

The mouse thread now splits into two paths, treated in the next two subsections: events forwarded to the application (when the pointer is inside a window that wants them), and events kept by `rio` itself for window management (move, resize, and the menus).

5.2.1 Application mouse events

A mouse event is forwarded to the window when: the cursor is inside the window and either it is a left-click (used for text selection in textual windows), or `Window.mouseopen`[51f] is set (meaning the application opened

/mnt/wsys/mouse and wants raw mouse events), or automatic scrolling is active.

⟨mousethread() *set sending to true for some conditions* 68a⟩≡ (66e)
```
  if(inside &&
     ((mouse->buttons&1) || winput->mouseopen || scrolling))
       sending = true;
```
Uses mouse 48c.

Here is the application case: when **sending** is set, the mouse thread hands the event off to the window's own thread.

⟨mousethread() *locals* 68b⟩+≡ (66a) ◁66d 68e▷
```
  Mouse tmp;
```

⟨mousethread() *if sending* 68c⟩≡ (66e)
```
  if(sending){
  Sending:
      ⟨mousethread() when sending mouse message to window, set the cursor 68d⟩

      tmp = mousectl->m;
      tmp.xy = xy; // logical coordinates

      // Dispatch, to "current" window thread!
      send(winput->mc.c, &tmp);
      continue;
  }
```
Uses mousectl 48a.

Before handing the event over, the thread picks the right cursor shape: a corner cursor when the pointer rests on a border (so the user knows a resize is possible), the plain cursor otherwise. The cursor machinery itself (cornercursor()[85c], wsetcursor()[87]) is covered in Chapter 6.

⟨mousethread() *when sending mouse message to window, set the cursor* 68d⟩≡ (68c)
```
  if(mouse->buttons == 0){
      // cornercursor will call wsetcursor if cursor not on the border
      cornercursor(winput, mouse->xy, false);
      sending = false;
  }else
      wsetcursor(winput, false);
```
Uses cornercursor() 85c, mouse 48c, and wsetcursor() 87.

5.2.2 Windowing system mouse events

When a mouse event is *not* forwarded to the application, the mouse thread handles it as a windowing system operation. It first determines which window is under the cursor with wpointto()[69], then dispatches based on the button pressed: left-click on an unfocused window brings it to the front, middle-click opens the edit menu (cut/paste), and right-click opens the system menu (New/Resize/Move/Delete/Hide/Exit). Clicking on a window's border initiates a move or resize operation.

⟨mousethread() *locals* 68e⟩+≡ (66a) ◁68b 90b▷
```
  Window *w;
```

⟨mousethread() *if not sending* 68f⟩≡ (66e)
```
  w = wpointto(mouse->xy);

  ⟨mousethread() when not sending, set cursor part1 70a⟩
  ⟨mousethread() if moving and buttons 90e⟩
  ⟨mousethread() when not sending, set cursor part2 70b⟩
```

```
⟨mousethread() when not sending, if buttons 70c⟩
moving = false;
break;
```
Uses `mouse` 48c and `wpointto()` 69.

⟨*function* `wpointto` 69⟩≡ (342b)
```
Window*
wpointto(Point pt)
{
    int i;
    Window *v, *w;

    w = nil;
    for(i=0; i<nwindow; i++){
        v = windows[i];
        if(ptinrect(pt, v->screenr))
         if(!v->deleted)
          if(w==nil || v->topped > w->topped)
            w = v;
    }
    return w;
}
```
Uses `nwindow` 51b and `windows` 51a.

The loop above iterates over all `windows`[51a], keeps only those whose `screenr` contains `pt`, and picks the one with the largest `topped`[51c] counter. Because `topped` is bumped every time a window is raised (`w->topped = ++topped`), the window most recently brought to the front wins. I find it helpful to trace a concrete case with three overlapping windows:

```
windows[] = { A, B, C }          topped counters
                                 A.topped = 7
view (desktop)                   B.topped = 12
+-------------------------------+  C.topped = 15   (most recent)
|   A         .                 |
|   +---------+..               |
|   |         | B               |
|   |   +-----+-------+         |
|   +---|     |       |         |
|       |     |   C   |         |
|       |  +--+-------+----+    |
|       |  |  |       |    |    |
|       +--|  +-------+    |    |   click at pt
|          |      *        |    |   (* = pt)
|          +---------------+    |
+-------------------------------+

wpointto(pt):
  i=0 A: ptinrect? no          -> w = nil
  i=1 B: ptinrect? no          -> w = nil
  i=2 C: ptinrect? yes         -> w = C   (returned)
```

If the click had landed in the small sliver where `B` and `C` overlap but not `A`, both would pass `ptinrect` and the `v->topped > w->topped` tie-break would still pick `C` because `15 > 12`. The cost is linear in `nwindow`, which is fine for a human-scale desktop but would hurt if `rio` managed thousands of layers—in that case, I would keep a sorted z-list instead and stop at the first hit.

On the windowing-system path the thread also updates the cursor shape: if the pointer is over some window it may switch to a resize-corner cursor (`cornercursor()`[85c]), otherwise it restores the default arrow (`riosetcursor()`[85b]).

⟨`mousethread()` *when not sending, set cursor part1* 70a⟩≡ (68f)
```
/* change cursor if over anyone's border */
if(w != nil)
    cornercursor(w, mouse->xy, false);
else
    riosetcursor(nil, false);
```
Uses `cornercursor()` 85c, `mouse` 48c, and `riosetcursor()` 85b.

⟨`mousethread()` *when not sending, set cursor part2* 70b⟩≡ (68f)
```
if(w != nil)
    cornercursor(w, mouse->xy, false);
```
Uses `cornercursor()` 85c and `mouse` 48c.

If a button is pressed while `rio` is keeping the event, the thread dispatches on which button. On the focused window (or empty space), left does nothing here (text selection is handled in the window thread), middle opens the edit menu and right the window menu. A click on an unfocused window instead tops it—giving it the focus—and may then pass the event on.

⟨`mousethread()` *when not sending, if buttons* 70c⟩≡ (68f)
```
/* we're not sending the event, but if button is down maybe we should */
if(mouse->buttons){
    /* w->topped will be zero or less if window has been bottomed */
    if(w==nil || (w==winput && w->topped > 0)){
        if(mouse->buttons & 1){       // LEFT-click case
            ;
        }else if(mouse->buttons & 2){ // MIDDLE-click case
            if(winput && !winput->mouseopen)
                ⟨mousethread() middle click under certain conditions 198a⟩
        }else if(mouse->buttons & 4)  // RIGHT-click case
                ⟨mousethread() right click under certain conditions 88a⟩
    }else{
        /* if button 1 event in the window, top the window and wait for button up. */
        /* otherwise, top the window and pass the event on */
        ⟨mousethread() click on unfocused window, set w 104a⟩
        if(w && (mouse->buttons!=1 || winborder(w, mouse->xy)))
            // input changed
            goto Again;

        goto Drain;
    }
}
```
Uses `mouse` 48c and `winborder()` 86a.

The bodies of the left-, middle-, and right-click cases are filled in gradually over the following sections, as each opens its own little protocol (text selection, the edit menu, the window menu).

After a move or resize operation completes, there may be leftover mouse events (button-held messages) still queued from the I/O proc. If the thread returned to the main `alt()` loop immediately, those stale events would be misinterpreted as new actions. So `Drain` spins through `readmouse()` until all buttons are released, discarding the residual events, then jumps back to `Again` to re-evaluate the mouse position with a clean state.

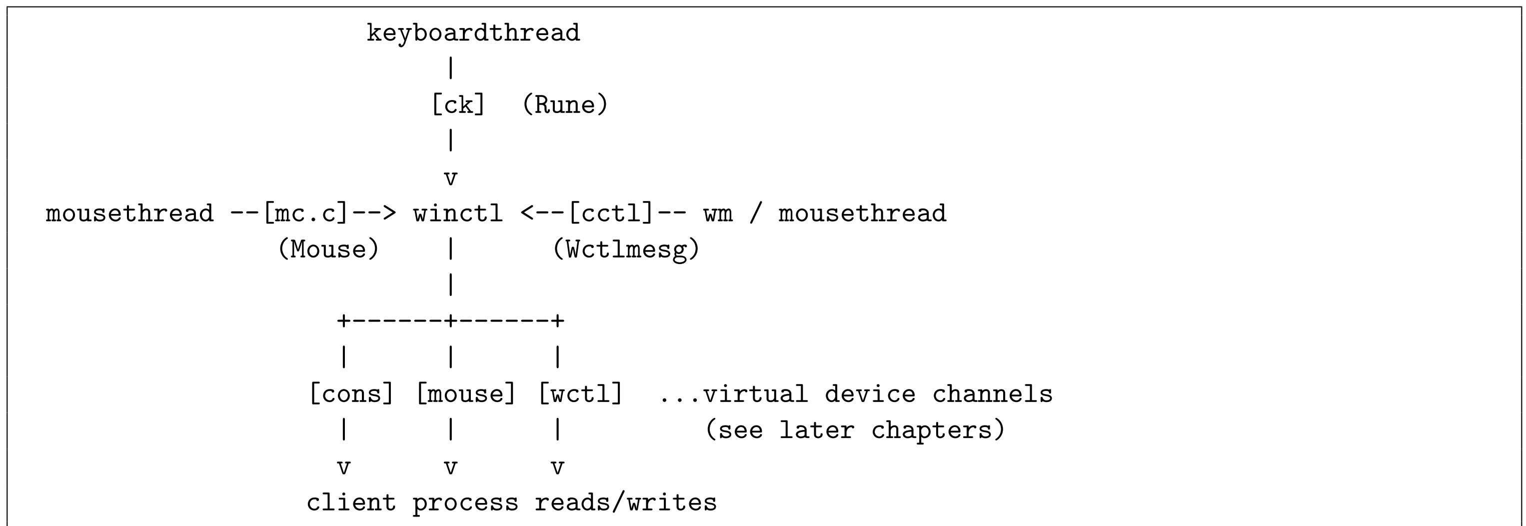
⟨`mousethread()` *Drain label* 70d⟩≡ (66e)
```
Drain:
    do {
        readmouse(mousectl);
    } while(mousectl->m.buttons);
```

```
    moving = false;
    goto Again; /* recalculate mouse position, cursor */
```
Uses `mousectl` 48a.

5.3 Window threads: `winctl()`

We now reach the thread that ties the others together, and the most interesting one in `rio`: the per-window `winctl()`[71b] thread.

Each window has its own `winctl()` thread that runs an `alt()` event loop listening on multiple channels simultaneously: keyboard events (from the keyboard thread), mouse events (from the mouse thread), control messages (from the window manager), and additional channels for the virtual device files (`/mnt/wsys/cons`, `/mnt/wsys/mouse`, etc.) that will be presented in later chapters. This is the "active object" pattern: each window is an independent concurrent entity processing its own event stream.



For now, I will focus on the three core connections—keyboard, mouse, and control—that make each window a responsive entity. The additional channels for virtual device files will be presented in later chapters.

⟨*enum* `Wxxx` 71a⟩≡ (339c)
```
  enum {
      WKey,
      WMouse,
      WCtl,
      ⟨Wxxx cases 163a⟩

      NWALT
  };
```

⟨*function* `winctl` 71b⟩≡ (339c)
```
  /// ... -> new() -> threadcreate(<>, w)
  void
  winctl(void *arg)
  {
      Window *w = arg;
      // map<enum<Wxxx>, Alt>
      Alt alts[NWALT+1];
      // enum<Wxxx>
      int event;
      char buf[128];
      ⟨winctl() other locals 72c⟩
```

```
    snprint(buf, sizeof buf, "winctl-id%d", w->id);
    threadsetname(buf);

    ⟨winctl() channels creation 163c⟩

    ⟨winctl() alts setup 72d⟩
    alts[NWALT].op = CHANEND;

    for(;;){
        ⟨winctl() alts adjustments 163f⟩
        // event loop
        event = alt(alts);
        ⟨winctl() trace w->id and event 278e⟩
        ⟨winctl() sanity check event 72a⟩
        switch(event){
        ⟨winctl() event loop cases 73a⟩
        }

        if(!w->deleted)
            flushimage(display, true);
    }
}
```

Uses `NWALT-32` 71a.

Note that after each event, `winctl()` calls `flushimage()` unless the window has been deleted (the `deleted` field will be explained in Chapter 7). Most events result in some visual change—text appearing, a cursor moving, a border redrawn—and `flushimage()` batches all pending draw operations into a single write to the draw device.

⟨`winctl()` *sanity check* `event` 72a⟩≡ (71b)

```
if(event == -1) {
  fprint(STDERR, "winctl: interrupted %r");
  exits("interrupted");
}
```

The event loop has three core cases, matching the `Wxxx` enum: `WKey`, `WMouse`, and `WCtl`. The next subsections take them in turn—keyboard and mouse forwarding are short, while `WCtl`, the window-control handler, is where most of the window-management logic lives.

5.3.1 Keyboard events listening

The first connection to wire up is the keyboard. Each window has a `Window.ck`[72b] channel that carries batches of `Rune`s from `keyboardthread()`[64].

⟨`Window` *keyboard fields* 72b⟩≡ (49)

```
// chan<Rune, 20> (listener = winctl, sender = keyboardthread)
Channel *ck;  /* chan(Rune[10]) */
```

⟨`winctl()` *other locals* 72c⟩≡ (71b) 74e ▷

```
Rune *kbdr;
```

⟨`winctl()` *alts setup* 72d⟩≡ (71b) 74b ▷

```
alts[WKey].c = w->ck;
alts[WKey].v = &kbdr;
alts[WKey].op = CHANRCV;
```

Uses `WKey-25` 71a.

An `Alt` entry just says where a receive (or send) should happen: `.c` is the channel, `.op` the direction, and `.v` the *address* of the variable to fill—the same `&v` you would pass to `recv(c, &v)`. Each message on `Window.ck` is a `Rune*` (a pointer to the rune batch), so the destination `kbdr` is a `Rune*` and `.v = &kbdr` is a `Rune**`.

```
w->ck:  [ ... | Rune* | ... ]      one pointer per message
                  |
      alt() stores it into *.v   ==>   kbdr --> ['a' 'b' 'c' '\0']
```

`sendp()` and `recvp()` are just conveniences for channels whose element is a pointer: they pass (or return) the pointer directly instead of its address. Concretely, the keyboard thread's `sendp(ck, rp)` is the same as `send(ck, &rp)`, and receiving with `recvp(ck)` into `kbdr` is the same as `recv(ck, &kbdr)`—which is exactly what the `alt()` entry above, with `.v = &kbdr`, does.

When a window has focus, the keyboard thread sends typed runes on `ck`, and `winctl()`[71b] dispatches each rune to `wkeyctl()`[73b] for processing.

⟨`winctl()` *event loop cases* 73a⟩≡ (71b) 74c ▷
```
case WKey:
    for(i=0; kbdr[i] != L'\0'; i++)
        wkeyctl(w, kbdr[i]);
    break;
```
Uses `WKey-25` 71a and `wkeyctl()` 73b.

⟨*function* `wkeyctl` 73b⟩≡ (343b)
```
void
wkeyctl(Window *w, Rune r)
{
    ⟨wkeyctl() locals 187b⟩

    ⟨wkeyctl() sanity check rune 73c⟩
    ⟨wkeyctl() return if window was deleted 73d⟩

    /* navigation keys work only when mouse is not open */
    ⟨wkeyctl() when mouse not opened and navigation keys 188d⟩

    ⟨wkeyctl() if rawing 164b⟩
    ⟨wkeyctl() if holding 272a⟩

    ⟨wkeyctl() when not rawing 187a⟩
}
```

I will explain the different cases of `wkeyctl()` gradually in later chapters. The logic is not a simple textual-vs-graphical split: `rio` allows unusual combinations, such as a graphical application (mouse opened) that still uses buffered console input, or a textual application in raw mode. In the latter case, `rio` still wants navigation keys (like arrow keys) to work for scrolling, which is why the `mouseopen` check comes first.

⟨`wkeyctl()` *sanity check rune* 73c⟩≡ (73b)
```
if(r == 0)
    return;
```

⟨`wkeyctl()` *return if window was deleted* 73d⟩≡ (73b)
```
if(w->deleted)
    return;
```

5.3.2 Mouse events listening

The second connection is the mouse. Each window embeds a full `Mousectl` structure whose `c` channel receives mouse events from `mousethread()`[66a]. This is a *second* `Mousectl`, distinct from the global `mousectl`[48a] that reads

the physical `/dev/mouse`: that one feeds `mousethread()`, while this per-window copy is how `mousethread()` forwards events on to a particular window. A window reuses the whole `Mousectl` type because it needs the same fields (a `Mouse`, a channel `c`, and a `resizec`) to deliver events to `winctl()`[71b]. Not every mouse event reaches the window—only those that the mouse thread has determined belong to this window (as opposed to window-management operations like move or resize).

⟨`Window` *mouse fields* 74a⟩≡ (49) 86d ▷

```
// mc.c = chan<Mouse> (listener = winctl, sender = mousethread)
Mousectl mc;
```

⟨`winctl()` *alts setup* 74b⟩+≡ (71b) ◁ 72d 74f ▷

```
alts[WMouse].c = w->mc.c;
alts[WMouse].v = &w->mc.m;
alts[WMouse].op = CHANRCV;
```

Uses `WMouse-26` 71a.

⟨`winctl()` *event loop cases* 74c⟩+≡ (71b) ◁ 73a 74g ▷

```
case WMouse:
    ⟨winctl() WMouse case if mouseopen 162d⟩
    else
    ⟨winctl() WMouse case if not mouseopen 199d⟩
    break;
```

Uses `WMouse-26` 71a.

As explained earlier, the behavior depends on whether the application has opened `/mnt/wsys/mouse` (`Window.mous` Graphical applications receive raw mouse events; textual applications get `rio`'s built-in text selection and scrolling. The details of each case will be covered in later chapters.

5.3.3 Control events listening

The third and final core connection is the control channel `Window.cctl`[74d], which carries `Wctlmesg`[91b] messages—resize, delete, hide, unhide, and other window-management commands. Unlike keyboard and mouse events that originate from hardware, control messages come from the window manager itself (e.g., when the user selects "Delete" from the right-click menu).

⟨`Window` *control fields* 74d⟩≡ (49)

```
// chan<Wctlmesg, 20> (listener = winctl, sender = mousethread | ...)
Channel *cctl;  /* chan(Wctlmesg)[20] */
```

⟨`winctl()` *other locals* 74e⟩+≡ (71b) ◁ 72c 162c ▷

```
Wctlmesg wcm;
```

⟨`winctl()` *alts setup* 74f⟩+≡ (71b) ◁ 74b 163e ▷

```
alts[WCtl].c = w->cctl;
alts[WCtl].v = &wcm;
alts[WCtl].op = CHANRCV;
```

Uses `WCtl-27` 71a.

When `wctlmesg()`[92a] returns `Exited`, the window is being destroyed and the thread exits.

⟨`winctl()` *event loop cases* 74g⟩+≡ (71b) ◁ 74c 163h ▷

```
case WCtl:
    if(wctlmesg(w, wcm.type, wcm.r, wcm.image) == Exited){
        ⟨winctl() Wctl case, free channels if wctlmesg is Excited 163d⟩
        threadexits(nil);
    }
    continue;
```

Uses `Exited` 109e, `WCtl-27` 71a, and `wctlmesg()` 92a.

The full lifecycle of a window—creation, hiding, unhiding, and deletion—involves subtle interactions between the window thread, the filesystem workers, and the mouse thread. I will explain `wctlmesg()` and its cases in Chapter 7.

5.4 Filesystem server proc

The filesystem server runs as a separate proc rather than a thread because it blocks on `read()` of the pipe—and a blocking thread would freeze all other threads in the same proc.

We start with `filsysproc()`[75], the loop that reads requests; it relies on a pool of worker threads that we build in the following sections.

5.4.1 Reading and dispatching: `filsysproc()`

`filsysproc()`[75] is the server's main loop. It owns the kernel-facing end of the pipe and spends its life blocked in `read9pmsg()`, waking only when a client triggers a new 9P request.

Each incoming 9P request is parsed, the fid is looked up, and the request is dispatched through the `fcall`[56b] table to the appropriate handler.

⟨*function* `filsysproc` 75⟩≡ (346b)

```
/// threadmain -> filsysinit -> proccreate(<>, fs, ...)
static
void
filsysproc(void *arg)
{
    Filsys *fs = arg;
    int n;
    byte *buf;
    Xfid *x = nil;
    Fid *f;
    ⟨filsysproc() other locals 76d⟩

    threadsetname("FILSYSPROC");

    for(;;){
        buf = emalloc(messagesize+UTFmax); /* UTFmax for appending partial rune in xfidwrite */

        n = read9pmsg(fs->sfd, buf, messagesize);
        ⟨filsysproc() sanity check n 76b⟩
        if(x == nil){
            send(fs->cxfidalloc, nil);
            recv(fs->cxfidalloc, &x);
            x->fs = fs;
        }
        x->buf = buf;

        if(convM2S(buf, n, &x->req) != n)
            error("convert error in convM2S");
        ⟨filsysproc() dump Fcall if debug 278c⟩

        ⟨filsysproc() sanity check x type 76e⟩
        else{
            ⟨filsysproc() if x type is Tversion or Tauth 76c⟩
            else
                f = newfid(fs, x->req.fid);
            x->f = f;
```

```
            // Dispatch
            x  = (*fcall[x->req.type])(fs, x, f);
        }
        ⟨filsysproc() end of loop 77d⟩
    }
 }
```
Uses error() 282c, fcall 56b, messagesize 76a, and newfid() 54b.

⟨*global* messagesize 76a⟩≡ (346a)
```
 int messagesize = 8192+IOHDRSZ; /* good start */
```
Uses messagesize 76a.

A few points about this loop:

- The buffer is dynamically allocated each iteration rather than reused because it is handed off to the `Xfid` worker thread (via `x->buf = buf`). The worker may still be processing the previous request when `filsysproc` reads the next one, so each request needs its own buffer.
- The extra `UTFmax` bytes are for `xfidwrite()`[137b], which may need to append a partial rune left over from a previous write.
- `convM2S()` ("convert Message to Struct") deserializes the raw 9P bytes into an `Fcall` structure with typed fields (`type`, `fid`, `count`, etc.).

The protocol between `filsysproc()` and the worker allocator is a two-step handshake: `filsysproc()` sends `nil` on `cxfidalloc`[62b] to request a worker, then receives the `Xfid`[55b] pointer back on the same channel. The returned `Xfid` may be a recycled one or a freshly allocated one with a new worker thread. After dispatching through `fcall`, if the handler returns non-nil, the worker can be reused immediately; if it returns nil, the handler is still running asynchronously in the worker thread.

⟨filsysproc() *sanity check n* 76b⟩≡ (75)
```
 if(n <= 0){
     yield(); /* if threadexitsall'ing, will not return */
     fprint(STDERR, "rio: %d: read9pmsg: %d %r\n", getpid(), n);
     errorshouldabort = false;
     error("eof or i/o error on server channel");
 }
```
Uses error() 282c and errorshouldabort 282a.

⟨filsysproc() *if x type is Tversion or Tauth* 76c⟩≡ (75)
```
 if(x->req.type==Tversion || x->req.type==Tauth)
     f = nil;
```

⟨filsysproc() *other locals* 76d⟩≡ (75)
```
 Fcall fc;
```

⟨filsysproc() *sanity check x type* 76e⟩≡ (75)
```
 if(fcall[x->req.type] == nil)
     x = filsysrespond(fs, x, &fc, Ebadfcall);
```
Uses Ebadfcall 280d, fcall 56b, and filsysrespond() 124.

5.4.2 Protocol handshake: `filsysversion()`

The first 9P message a client must send is always `Tversion`, which negotiates the protocol version and maximum message size. This is handled directly by `filsysproc()`[75] without dispatching to a worker, since it is a one-time handshake.

⟨fcall *other methods* 76f⟩≡ (56b) 139b▷
```
 [Tversion] = filsysversion,
```
Uses filsysversion() 77a.

⟨*function* filsysversion 77a⟩≡ (346c)
```
static
Xfid*
filsysversion(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    ⟨filsysversion() sanity checks 77c⟩
    // else
    messagesize = x->req.msize;
    fc.msize = messagesize;
    fc.version = "9P2000";
    return filsysrespond(fs, x, &fc, nil);
}
```
Uses filsysrespond() 124 and messagesize 76a.

⟨*global* firstmessage 77b⟩≡ (346c)
```
bool firstmessage = true;
```
Uses firstmessage 77b.

⟨filsysversion() *sanity checks* 77c⟩≡ (77a) 77e ▷
```
if(!firstmessage)
    return filsysrespond(x->fs, x, &fc, "version request not first message");
```
Uses filsysrespond() 124 and firstmessage 77b.

⟨filsysproc() *end of loop* 77d⟩≡ (75)
```
firstmessage = false;
```
Uses firstmessage 77b.

⟨filsysversion() *sanity checks* 77e⟩+≡ (77a) ◁ 77c
```
if(x->req.msize < 256)
    return filsysrespond(x->fs, x, &fc, "version: message size too small");
if(strncmp(x->req.version, "9P2000", 6) != 0)
    return filsysrespond(x->fs, x, &fc, "unrecognized 9P version");
```
Uses filsysrespond() 124.

5.5 Xfid worker pool

We saw `filsysproc()`[75] hand each request off to a worker; here is the pool those workers come from.

The worker pool is made of two kinds of thread: a single *allocator* thread that hands out and recycles `Xfid` worker handles, and the *worker* threads themselves, each of which executes one dispatched request before returning to the pool. Both appear as distinct nodes in Figure 2.11.

5.5.1 Allocator thread: xfidallocthread()

The allocator thread manages a pool (arena) of `Xfid` worker handles. When `filsysproc()`[75] requests a worker (`Alloc`), the allocator either recycles one from the free list or creates a new one with its own `xfidctl()`[79c] thread. When a worker finishes (`Free`), it is returned to the free list.

⟨*enum* Xxxx 77f⟩≡ (340a)
```
enum {
    Alloc,
    Free,

    N
};
```

Do not confuse these two *lists* with the two *channels* `cxfidalloc`[62b]/`cxfidfree`[62c] seen earlier. The lists are the allocator's private bookkeeping—`xfid`[78a] holds every worker ever created, `xfidfree`[78b] holds the currently idle ones—while the channels are its *interface* to the rest of `rio`: a request on `cxfidalloc` makes the allocator take a worker from `xfidfree` (creating one, recorded in `xfid`, if none is free), and a worker handed back on `cxfidfree` goes onto `xfidfree`.

⟨*global* `xfid` 78a⟩≡ (340a)
```
// list<ref_own<Xfid>> (next = Xfid.next)
static Xfid *xfid;
```

⟨*global* `xfidfree` 78b⟩≡ (340a)
```
// list<ref_own<Xfid>> (next = Xfid.free)
static Xfid *xfidfree;
```

This is the same intrusive-linked-list idiom as `Fid.next`, but an `Xfid` belongs to *two* lists at once—the all-workers list and the idle list—so it carries one link pointer per list: `next` threads it onto `xfid`, `free` onto `xfidfree`.

⟨`Xfid` *extra fields* 78c⟩≡ (55b)
```
Xfid *next;
Xfid *free;
```

We can now read `xfidallocthread()`[78d] itself: an `alt()` loop with the two cases `Alloc` and `Free` above, popping a worker off `xfidfree` or pushing one back onto it.

⟨*function* `xfidallocthread` 78d⟩≡ (340a)
```
void
xfidallocthread(void*)
{
    Xfid *x;
    static Alt alts[N+1];

    alts[Alloc].c = cxfidalloc;
    alts[Alloc].v = nil;
    alts[Alloc].op = CHANRCV;
    alts[Free].c = cxfidfree;
    alts[Free].v = &x;
    alts[Free].op = CHANRCV;
    alts[N].op = CHANEND;

    for(;;){
        // event loop
        switch(alt(alts)){
        case Alloc:
            x = xfidfree;
            if(x)
                xfidfree = x->free;
            else{
                x = emalloc(sizeof(Xfid));
                x->c = chancreate(sizeof(void(*)(Xfid*)), 0);
                ⟨xfidallocthread() create flushc channel 266b⟩

                // insert_list(x, xfid)
                x->next = xfid;
                xfid = x;

                // new Xfid threads!
                threadcreate(xfidctl, x, 16384);
            }
            ⟨xfidallocthread() sanity check x when Alloc 79a⟩
            incref(x);
```

```
            sendp(cxfidalloc, x);
            break;

        case Free:
            ⟨xfidallocthread() sanity check x when Free 79b⟩
            // insert_list(x, xfidfree)
            x->free = xfidfree;
            xfidfree = x;
            break;
        }
    }
}
```

Uses Alloc-55 77f, Free-56 77f, N-57 77f, cxfidalloc-53 62b, cxfidfree-54 62c, xfid-52 78a, xfidctl() 79c, and xfidfree-51 78b.

The key line in that loop is `threadcreate(xfidctl, x, ...)`: every new worker runs `xfidctl()`, the thread that actually executes the `xfidXXX` handlers doing the 9P work. We turn to it next.

Why route allocation through a dedicated thread and channels instead of just calling an allocation function directly? Because the free list (`xfidfree`) is shared state whose users live in different procs: `filsysproc()` runs in its own proc while the workers run in the main proc. A plain function would therefore need a lock around the pool. Giving a single thread sole *ownership* of the list, and talking to it over channels, sidesteps that: only `xfidallocthread()` ever touches `xfidfree`, and channels already carry requests safely across the proc boundary. This is the CSP model [Hoa85] on which Plan 9's threads are built, later popularized by the Go proverb: do not communicate by sharing memory; instead, share memory by communicating.

⟨`xfidallocthread()` *sanity check x when Alloc* 79a⟩≡ (78d)

```
if(x->ref != 0){
    fprint(STDERR, "%p incref %ld\n", x, x->ref);
    error("incref");
}
if(x->flushtag != -1)
    error("flushtag in allocate");
```

Uses error() 282c.

⟨`xfidallocthread()` *sanity check x when Free* 79b⟩≡ (78d)

```
if(x->ref != 0){
    fprint(STDERR, "%p decref %ld\n", x, x->ref);
    error("decref");
}
if(x->flushtag != -1)
    error("flushtag in free");
```

Uses error() 282c.

5.5.2 Worker threads: `xfidctl()`

We reach the last of `rio`'s key threads, and after `winctl()` [71b] the most important: the worker thread `xfidctl()` [79c], where the actual file-server work happens.

Each worker thread is an elegant loop: it blocks on `recvp()` waiting for a function pointer, then calls it. This is essentially receiving "code to execute" through a channel—the Plan 9 equivalent of a closure. After execution, the worker decrements its reference count and returns itself to the free pool if nobody else holds a reference.

⟨*function* `xfidctl` 79c⟩≡ (340a)

```
void
xfidctl(void *arg)
{
    Xfid *x = arg;
```

```
    void (*f)(Xfid*);
    char buf[64];

    snprint(buf, sizeof buf, "xfid.%p", x);

    threadsetname(buf);

    for(;;){
        f = recvp(x->c);

        // Executing a xfidxxx()
        (*f)(x);

        if(decref(x) == 0)
            sendp(cxfidfree, x);
    }
}
```

Uses `cxfidfree-54` 62c.

The functions sent through the channel are the 9P operation handlers: `xfidattach()` [127c], `xfidopen()` [133a], `xfidread()` [136b], `xfidwrite()` [137b], `xfidclose()` [134], and `xfidflush()` [267c]. Each one implements one step of the file protocol—for instance, when a client reads `/mnt/wsys/cons`, the master sends `xfidread()` to a worker, which then retrieves console data from the appropriate window. These handlers are covered in later chapters.

Chapter 6

Cursors

Before diving into the heart of window management, I will cover the simpler topic of cursor handling. `rio` uses different cursor shapes to provide visual feedback about what operation is possible: a plus sign for creating windows, a box for moving, a gunsight for deleting, and directional arrows when hovering over window borders and corners.

6.1 Cursor graphics

6.1.1 Classic cursors

The `crosscursor`[81] below (a plus sign) appears during window creation (`sweep()`[101d], see Figure 2.7).

⟨*global crosscursor (rio/data.c)* 81⟩≡ (341)

```
Cursor crosscursor = {
    {-7, -7},
    {0x03, 0xC0, 0x03, 0xC0, 0x03, 0xC0, 0x03, 0xC0,
     0x03, 0xC0, 0x03, 0xC0, 0xFF, 0xFF, 0xFF, 0xFF,
     0xFF, 0xFF, 0xFF, 0xFF, 0x03, 0xC0, 0x03, 0xC0,
     0x03, 0xC0, 0x03, 0xC0, 0x03, 0xC0, 0x03, 0xC0, },
    {0x00, 0x00, 0x01, 0x80, 0x01, 0x80, 0x01, 0x80,
     0x01, 0x80, 0x01, 0x80, 0x01, 0x80, 0x7F, 0xFE,
     0x7F, 0xFE, 0x01, 0x80, 0x01, 0x80, 0x01, 0x80,
     0x01, 0x80, 0x01, 0x80, 0x01, 0x80, 0x00, 0x00, }
};
```

Each cursor is a `Cursor` structure (defined in `cursor.h`) containing a *hotspot offset* and two 16x16 bitmaps. The hotspot offset tells the system how much to shift the bitmap so that the *active point* (the pixel that counts as the click location) aligns with the mouse position. For example, `crosscursor` has offset `-7, -7`: the bitmap is shifted 7 pixels up and 7 pixels left, placing the center of the plus sign exactly on the mouse pointer:

```
       +----------------+
       |       ||       |     The bitmap's top-left corner is at
       |       ||       |     mouse position + offset, i.e.,
       |       ||       |     (mouse.x-7, mouse.y-7).
       |=======XX=======|     So the center pixel (7,7) within
       |       ||       |     the bitmap lands exactly on the
       |       ||       |     mouse position.
       |       ||       |
       +----------------+
                ^
                mouse pointer
```

The two bitmaps work together for transparency: the first is the mask, the second is the image. Where a mask bit is set, the cursor pixel is visible (white or black depending on the image bit). Where a mask bit is clear, the background shows through—this is how cursors get their non-rectangular, transparent outlines. Each bitmap is one bit per pixel, so a 16x16 cursor is 256 bits. A hexadecimal constant from `0x00` to `0xFF` holds one byte (8 bits), so the bitmap is the 32 bytes shown above (256/8 = 32); each row of the cursor takes 2 of them (16 pixels = 16 bits = 2 bytes).

`boxcursor`[82a] appears during window moves (`drag()`[111c]), `sightcursor`[82b] when "Delete" is selected from the system menu, `whitearrow`[82c] in holding mode, and `query`[82d] for unknown states. The default arrow cursor lives in the draw library, not in `rio`.

⟨*global boxcursor (rio/data.c)* 82a⟩≡ (341)

```
Cursor boxcursor = {
    {-7, -7},
    {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
     0xFF, 0xFF, 0xF8, 0x1F, 0xF8, 0x1F, 0xF8, 0x1F,
     0xF8, 0x1F, 0xF8, 0x1F, 0xF8, 0x1F, 0xFF, 0xFF,
     0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, },
    {0x00, 0x00, 0x7F, 0xFE, 0x7F, 0xFE, 0x7F, 0xFE,
     0x70, 0x0E, 0x70, 0x0E, 0x70, 0x0E, 0x70, 0x0E,
     0x70, 0x0E, 0x70, 0x0E, 0x70, 0x0E, 0x70, 0x0E,
     0x7F, 0xFE, 0x7F, 0xFE, 0x7F, 0xFE, 0x00, 0x00, }
};
```

⟨*global sightcursor (rio/data.c)* 82b⟩≡ (341)

```
Cursor sightcursor = {
    {-7, -7},
    {0x1F, 0xF8, 0x3F, 0xFC, 0x7F, 0xFE, 0xFB, 0xDF,
     0xF3, 0xCF, 0xE3, 0xC7, 0xFF, 0xFF, 0xFF, 0xFF,
     0xFF, 0xFF, 0xFF, 0xFF, 0xE3, 0xC7, 0xF3, 0xCF,
     0x7B, 0xDF, 0x7F, 0xFE, 0x3F, 0xFC, 0x1F, 0xF8, },
    {0x00, 0x00, 0x0F, 0xF0, 0x31, 0x8C, 0x21, 0x84,
     0x41, 0x82, 0x41, 0x82, 0x41, 0x82, 0x7F, 0xFE,
     0x7F, 0xFE, 0x41, 0x82, 0x41, 0x82, 0x41, 0x82,
     0x21, 0x84, 0x31, 0x8C, 0x0F, 0xF0, 0x00, 0x00, }
};
```

⟨*global whitearrow (rio/data.c)* 82c⟩≡ (341)

```
Cursor whitearrow = {
    {0, 0},
    {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFC,
     0xFF, 0xF0, 0xFF, 0xF0, 0xFF, 0xF8, 0xFF, 0xFC,
     0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFC,
     0xF3, 0xF8, 0xF1, 0xF0, 0xE0, 0xE0, 0xC0, 0x40, },
    {0xFF, 0xFF, 0xFF, 0xFF, 0xC0, 0x06, 0xC0, 0x1C,
     0xC0, 0x30, 0xC0, 0x30, 0xC0, 0x38, 0xC0, 0x1C,
     0xC0, 0x0E, 0xC0, 0x07, 0xCE, 0x0E, 0xDF, 0x1C,
     0xD3, 0xB8, 0xF1, 0xF0, 0xE0, 0xE0, 0xC0, 0x40, }
};
```

⟨*global query (rio/data.c)* 82d⟩≡ (341)

```
Cursor query = {
    {-7,-7},
    {0x0f, 0xf0, 0x1f, 0xf8, 0x3f, 0xfc, 0x7f, 0xfe,
     0x7c, 0x7e, 0x78, 0x7e, 0x00, 0xfc, 0x01, 0xf8,
     0x03, 0xf0, 0x07, 0xe0, 0x07, 0xc0, 0x07, 0xc0,
     0x07, 0xc0, 0x07, 0xc0, 0x07, 0xc0, 0x07, 0xc0, },
    {0x00, 0x00, 0x0f, 0xf0, 0x1f, 0xf8, 0x3c, 0x3c,
     0x38, 0x1c, 0x00, 0x3c, 0x00, 0x78, 0x00, 0xf0,
     0x01, 0xe0, 0x03, 0xc0, 0x03, 0x80, 0x03, 0x80,
```

```
    0x00, 0x00, 0x03, 0x80, 0x03, 0x80, 0x00, 0x00, }
};
```

6.1.2 Border and corner cursors

When the mouse hovers over a window border, `rio` shows a directional cursor indicating which edge or corner is under the mouse. The `corners`[83a] array maps the 3x3 grid of possible positions (top-left, top, top-right, left, center, right, bottom-left, bottom, bottom-right) to the corresponding cursor. The center entry is `nil` because the center is inside the window, not on the border.

⟨*global corners (rio/data.c)* 83a⟩≡ (341)
```
Cursor *corners[9] = {
    &tl,    &t,   &tr,
    &l,    nil,   &r,
    &bl,    &b,   &br,
};
```
Uses `b` 84c, `bl` 84d, `br` 84b, `l` 84e, `r` 84a, `t-15` 83c, `tl` 83b, and `tr` 83d.

⟨*global* `tl` 83b⟩≡ (341)
```
Cursor tl = {
    {-4, -4},
    {0xfe, 0x00, 0x82, 0x00, 0x8c, 0x00, 0x87, 0xff,
     0xa0, 0x01, 0xb0, 0x01, 0xd0, 0x01, 0x11, 0xff,
     0x11, 0x00, 0x11, 0x00, 0x11, 0x00, 0x11, 0x00,
     0x11, 0x00, 0x11, 0x00, 0x11, 0x00, 0x1f, 0x00, },
    {0x00, 0x00, 0x7c, 0x00, 0x70, 0x00, 0x78, 0x00,
     0x5f, 0xfe, 0x4f, 0xfe, 0x0f, 0xfe, 0x0e, 0x00,
     0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00,
     0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00, 0x00, 0x00, }
};
```

⟨*global* `t` 83c⟩≡ (341)
```
static Cursor t = {
    {-7, -8},
    {0x00, 0x00, 0x00, 0x00, 0x03, 0x80, 0x06, 0xc0,
     0x1c, 0x70, 0x10, 0x10, 0x0c, 0x60, 0xfc, 0x7f,
     0x80, 0x01, 0x80, 0x01, 0x80, 0x01, 0xff, 0xff,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, },
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00,
     0x03, 0x80, 0x0f, 0xe0, 0x03, 0x80, 0x03, 0x80,
     0x7f, 0xfe, 0x7f, 0xfe, 0x7f, 0xfe, 0x00, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, }
};
```

⟨*global* `tr` 83d⟩≡ (341)
```
Cursor tr = {
    {-11, -4},
    {0x00, 0x7f, 0x00, 0x41, 0x00, 0x31, 0xff, 0xe1,
     0x80, 0x05, 0x80, 0x0d, 0x80, 0x0b, 0xff, 0x88,
     0x00, 0x88, 0x0, 0x88, 0x00, 0x88, 0x00, 0x88,
     0x00, 0x88, 0x00, 0x88, 0x00, 0x88, 0x00, 0xf8, },
    {0x00, 0x00, 0x00, 0x3e, 0x00, 0x0e, 0x00, 0x1e,
     0x7f, 0xfa, 0x7f, 0xf2, 0x7f, 0xf0, 0x00, 0x70,
     0x00, 0x70, 0x00, 0x70, 0x00, 0x70, 0x00, 0x70,
     0x00, 0x70, 0x00, 0x70, 0x00, 0x70, 0x00, 0x00, }
};
```

⟨*global* r 84a⟩≡ (341)
```
Cursor r = {
    {-8, -7},
    {0x07, 0xc0, 0x04, 0x40, 0x04, 0x40, 0x04, 0x58,
     0x04, 0x68, 0x04, 0x6c, 0x04, 0x06, 0x04, 0x02,
     0x04, 0x06, 0x04, 0x6c, 0x04, 0x68, 0x04, 0x58,
     0x04, 0x40, 0x04, 0x40, 0x04, 0x40, 0x07, 0xc0, },
    {0x00, 0x00, 0x03, 0x80, 0x03, 0x80, 0x03, 0x80,
     0x03, 0x90, 0x03, 0x90, 0x03, 0xf8, 0x03, 0xfc,
     0x03, 0xf8, 0x03, 0x90, 0x03, 0x90, 0x03, 0x80,
     0x03, 0x80, 0x03, 0x80, 0x03, 0x80, 0x00, 0x00, }
};
```

⟨*global* br 84b⟩≡ (341)
```
Cursor br = {
    {-11, -11},
    {0x00, 0xf8, 0x00, 0x88, 0x00, 0x88, 0x00, 0x88,
     0x00, 0x88, 0x00, 0x88, 0x00, 0x88, 0x00, 0x88,
     0xff, 0x88, 0x80, 0x0b, 0x80, 0x0d, 0x80, 0x05,
     0xff, 0xe1, 0x00, 0x31, 0x00, 0x41, 0x00, 0x7f, },
    {0x00, 0x00, 0x00, 0x70, 0x00, 0x70, 0x00, 0x70,
     0x0, 0x70, 0x00, 0x70, 0x00, 0x70, 0x00, 0x70,
     0x00, 0x70, 0x7f, 0xf0, 0x7f, 0xf2, 0x7f, 0xfa,
     0x00, 0x1e, 0x00, 0x0e, 0x00, 0x3e, 0x00, 0x00, }
};
```

⟨*global* b 84c⟩≡ (341)
```
Cursor b = {
    {-7, -7},
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
     0xff, 0xff, 0x80, 0x01, 0x80, 0x01, 0x80, 0x01,
     0xfc, 0x7f, 0x0c, 0x60, 0x10, 0x10, 0x1c, 0x70,
     0x06, 0xc0, 0x03, 0x80, 0x00, 0x00, 0x00, 0x00, },
    {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
     0x00, 0x00, 0x7f, 0xfe, 0x7f, 0xfe, 0x7f, 0xfe,
     0x03, 0x80, 0x03, 0x80, 0x0f, 0xe0, 0x03, 0x80,
     0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, }
};
```

⟨*global* bl 84d⟩≡ (341)
```
Cursor bl = {
    {-4, -11},
    {0x1f, 0x00, 0x11, 0x00, 0x11, 0x00, 0x11, 0x00,
     0x11, 0x00, 0x11, 0x00, 0x11, 0x00, 0x11, 0x00,
     0x11, 0xff, 0xd0, 0x01, 0xb0, 0x01, 0xa0, 0x01,
     0x87, 0xff, 0x8c, 0x00, 0x82, 0x00, 0xfe, 0x00, },
    {0x00, 0x00, 0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00,
     0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00, 0x0e, 0x00,
     0x0e, 0x00, 0x0f, 0xfe, 0x4f, 0xfe, 0x5f, 0xfe,
     0x78, 0x00, 0x70, 0x00, 0x7c, 0x00, 0x00, 0x0, }
};
```

⟨*global* l 84e⟩≡ (341)
```
Cursor l = {
    {-7, -7},
    {0x03, 0xe0, 0x02, 0x20, 0x02, 0x20, 0x1a, 0x20,
     0x16, 0x20, 0x36, 0x20, 0x60, 0x20, 0x40, 0x20,
     0x60, 0x20, 0x36, 0x20, 0x16, 0x20, 0x1a, 0x20,
     0x02, 0x20, 0x02, 0x20, 0x02, 0x20, 0x03, 0xe0, },
    {0x00, 0x00, 0x01, 0xc0, 0x01, 0xc0, 0x01, 0xc0,
```

```
    0x09, 0xc0, 0x09, 0xc0, 0x1f, 0xc0, 0x3f, 0xc0,
    0x1f, 0xc0, 0x09, 0xc0, 0x09, 0xc0, 0x01, 0xc0,
    0x01, 0xc0, 0x01, 0xc0, 0x01, 0xc0, 0x00, 0x00, }
};
```

6.2 Setting the cursor

Cursor management involves a tension between `rio` (which wants to show border cursors and operation cursors) and client applications (which can set their own cursor via `/dev/cursor`). `cornercursor()`[85c] resolves this: if the mouse is on a window border, it shows the appropriate directional cursor; otherwise it defers to the window's own cursor via `wsetcursor()`[87].

6.2.1 Low-level setter: `riosetcursor()`

Since changing the cursor requires a write to `/dev/cursor`—a system call—`riosetcursor()`[85b] caches the last cursor set and skips the write if it has not changed. The `force` parameter bypasses this optimization for cases where the cursor may have been changed externally (e.g., after a window gains focus and needs to reassert its cursor).

⟨*global* `lastcursor` 85a⟩≡ (342a)

```
Cursor *lastcursor;
```

⟨*function* `riosetcursor` 85b⟩≡ (342a)

```
/// cornercursor | ... -> <>
void
riosetcursor(Cursor *p, bool force)
{
    if(!force && p==lastcursor)
        return;
    // else
    setcursor(mousectl, p);
    lastcursor = p;
}
```

Uses `lastcursor` 85a and `mousectl` 48a.

`setcursor()` (from `mouse.h`, in the Graphics book [Pad16c]) is what actually changes the on-screen pointer: it writes the `Cursor`'s hotspot and bitmaps to `/dev/cursor`, the kernel's cursor device, which the hardware then displays; passing `nil` resets it to the default arrow. `riosetcursor()` is a thin wrapper that skips the write when the cursor has not changed since last time (the `lastcursor` optimization).

6.2.2 Cursor dispatch: `cornercursor()`

This is the main cursor dispatch function, called from `mousethread()`[66a] on every mouse move. It checks whether the mouse is on the window border via `winborder()`[86a] (point inside `screenr` but outside the inset rectangle): if so, it picks the appropriate directional cursor from the `corners`[83a] array; otherwise it falls through to `wsetcursor()`[87], which shows the application's cursor or the default arrow.

⟨*function* `cornercursor` 85c⟩≡ (340c)

```
/// mousethread -> <>
void
cornercursor(Window *w, Point p, bool force)
{
    if(w != nil && winborder(w, p))
        riosetcursor(corners[whichcorner(w, p)], force);
    else
```

```
        wsetcursor(w, force);
}
```
Uses `corners` 83a, `riosetcursor()` 85b, `whichcorner()` 86b, `winborder()` 86a, and `wsetcursor()` 87.

⟨*function* `winborder` 86a⟩≡ (342b)
```
bool
winborder(Window *w, Point xy)
{
    return ptinrect(xy, w->screenr) &&
           !ptinrect(xy, insetrect(w->screenr, Selborder));
}
```
Uses `Selborder` 67.

The `whichcorner()`[86b] function maps a point on the window border to an index in the 3x3 `corners` array.

⟨*function* `whichcorner` 86b⟩≡ (340c)
```
int
whichcorner(Window *w, Point p)
{
    int i, j;

    i = portion(p.x, w->screenr.min.x, w->screenr.max.x);
    j = portion(p.y, w->screenr.min.y, w->screenr.max.y);
    return 3*j + i;
}
```
Uses `portion()` 86c.

The `portion()`[86c] helper divides each axis into three zones using a hardcoded 20-pixel threshold: near the start (0), in the middle (1), or near the end (2). The result `3*j + i` gives the flat array index corresponding to the grid position (top-left = 0, top = 1, ..., bottom-right = 8).

⟨*function* `portion` 86c⟩≡ (340c)
```
int
portion(int x, int lo, int hi)
{
    x -= lo;
    hi -= lo;

    if(x < 20)
        return 0; // top
    if(x > hi-20)
        return 2; // below
    return 1; // middle
}
```

6.2.3 Window cursor: `wsetcursor()`

`cornercursor()`[85c] above handles the resize-corner shapes; away from a corner it defers to the window's own cursor. That per-window cursor is what `wsetcursor()`[87] sets. Each window stores its application-requested cursor in the `cursor` field, with `cursorp` acting as an option: when non-nil, it points to `cursor` (meaning the app set a custom cursor); when nil, the default arrow is used.

⟨`Window` *mouse fields* 86d⟩+≡ (49) ◁74a 161a▷
```
Cursor  cursor;
// option<ref<Cursor>> (to Window.cursor when not None)
Cursor  *cursorp;
```

The `wsetcursor()` function checks whether the mouse is actually over this window via `wpointto()`[69] and whether a system menu operation is in progress via `menuing`[102a], ensuring that `rio`'s own cursor shapes take priority during window management.

⟨*function* wsetcursor 87⟩≡ (342b)
```
/// mousethread | wcurrent | ... -> <>
void
wsetcursor(Window *w, bool force)
{
    Cursor *p;

    if(w==nil || w->i==nil || Dx(w->screenr)<=0)
        p = nil;
    else if(wpointto(mouse->xy) == w){
        p = w->cursorp;
        ⟨wsetcursor() if holding 270e⟩
    }else
        p = nil;

    if(!menuing)
        riosetcursor(p, force);
}
```
Uses `menuing` 102a, `mouse` 48c, `riosetcursor()` 85b, and `wpointto()` 69.

The `menuing` flag prevents cursor changes during system menu operations, when `rio` must keep control of the cursor.

Chapter 7

Window Manager

Now that I have covered the threads and procs that form `rio`'s concurrent architecture—and the cursor management a window manager also leans on—I can present the window manager proper: the code that creates, moves, resizes, hides, and deletes windows.

7.1 Overview

In Plan 9, the window manager and the window server are the same program, unlike X Window where they are separate. Window management in `rio` can be triggered from several sources, but they all converge to the same mechanism: sending a `Wctlmesg`[91b] to the target `Window.cctl`[74d] channel, where its `winctl()`[71b] thread processes it:

```
Right-click menu ------+
Border click (move/    |
  resize) -------------+---> wsendctlmesg(w->cctl) ---> winctl()
/mnt/wsys/wctl write --+                                   |
/srv/riowctl write ----+                               wctlmesg()
External mount --------+                              processes it
```

This chapter covers each trigger in turn: the right-click system menu, border clicks, the `Wctlmesg` dispatch, window creation (with its many substeps), focus, deletion, move, resize, and visibility.

7.2 Right-click system menu

The right-click menu ("system menu") is the primary user interface for window management in `rio`. It offers: New (create a window), Resize, Move, Delete, Hide, and Exit menu entries. Hidden windows also appear as extra menu entries, allowing the user to unhide them. Most of this code runs in the mouse thread's context, but the actual modifications are performed by sending `Wctlmesg`[91b] messages to the target window's thread.

⟨`mousethread()` *right click under certain conditions* 88a⟩≡ (70c)

```
  button3menu();
```

Uses `button3menu()` 88b.

⟨*function* `button3menu` 88b⟩≡ (340c)

```
  /// mousethread -> event loop -> <>
  void
  button3menu(void)
  {
      int i;
```

```
    ⟨button3menu() menu3str adjustments with hidden windows 119c⟩

    sweeping = true;
    i = menuhit(3, mousectl, &menu3, desktop);
    switch(i){
    ⟨button3menu() cases 90a⟩
    case -1:
        break;
    }
    sweeping = false;
}
```

Uses `desktop` 48d, `menu3` 89a, `mousectl` 48a, and `sweeping` 89d.

`menuhit()` is a libdraw routine (see the GRAPHICS book [Pad16c]): it pops up a menu, lets the user pick, and returns the chosen item's index (or `-1` if none). The `3` says it tracks mouse button 3, `mousectl` is the mouse to follow, `&menu3` the menu to show, and `desktop` the `Screen` on which to allocate the menu's image—menuhit draws the menu as its own overlapping layer on top of the windows, so it must share `rio`'s desktop.

⟨*global* `menu3` 89a⟩≡ (340c)

```
Menu menu3 = { .item = menu3str };
```

Uses `menu3str` 89b.

⟨*global* `menu3str` 89b⟩≡ (340c)

```
char*  menu3str[100] = {
 [New]  "New",
 [Reshape] "Resize",
 [Move]  "Move",
 [Delete] "Delete",
 [Hide]  "Hide",
 [Exit]  "Exit",
 nil
};
```

⟨*enum* `_anon_ (rio/rio.c)` 89c⟩≡ (340c)

```
enum RightMenuCommand
{
    New,

    Reshape,
    Move,
    Delete,
    Hide,

    Exit,

    Hidden,
};
```

The `sweeping`[89d] flag is set while a system menu action is in progress. It prevents client applications from interfering—for example, an application writing to `/dev/mouse` to warp the cursor would be disruptive in the middle of a menu interaction.

⟨*global* `sweeping` 89d⟩≡ (338a)

```
bool sweeping;
```

`sweeping` and `menuing`[102a] are easy to confuse—both are raised while the user drives a window-manager gesture—but they guard different things. `menuing` is about `rio`'s *own* cursor: it is set while `rio` has swapped in a special pointer (the cross or sight cursor) to prompt the user to point at or sweep out a window, and it stops the normal cursor logic from reverting that pointer. `sweeping` instead keeps *clients* out of the way: while a menu

is up or a border is being dragged, it blocks an application from, say, warping the mouse through /dev/mouse mid-gesture.

We can now see the menu's action cases one at a time, starting with the simplest, Exit. The code below closes the loop with threadmain()[58], which blocks on exitchan[60e] with a recv(exitchan) call. When the user selects "Exit," the send() below unblocks threadmain() and rio shuts down.

⟨button3menu() *cases* 90a⟩≡ (88b) 92b ▷
```
  case Exit:
      send(exitchan, nil);
      break;
```
Uses Exit-13 89c and exitchan 60e.

The other menu actions—creating, reshaping, moving, hiding, and deleting windows—are filled in gradually over the following sections.

7.3 Window borders click

The right-click menu is not the only way to trigger a window-management action: clicking on a window's border is a second source, handled directly by the mouse thread.

When the mouse is on a window's border and a button is pressed, the mouse thread initiates a move or resize operation directly, without going through the right-click menu. Left or middle click on the border triggers bandsize()[115] (resize), while right-click triggers drag()[111c] (move). The result is sent as a Reshaped[91a] or Moved[91a] message to the window's winctl()[71b] thread.

⟨mousethread() *locals* 90b⟩+≡ (66a) ◁ 68e 90d ▷
```
  bool moving = false;
```

⟨mousethread() *set moving to true for some conditions* 90c⟩≡ (66e)
```
  /* topped will be zero or less if window has been bottomed */
  if(!sending && !scrolling
     && winborder(winput, mouse->xy) && winput->topped > 0){
      moving = true;
  }
```
Uses mouse 48c and winborder() 86a.

⟨mousethread() *locals* 90d⟩+≡ (66a) ◁ 90b 201b ▷
```
  bool inside, band;
  Window *oin;
  Image *i;
  Rectangle r;
```

⟨mousethread() *if moving and buttons* 90e⟩≡ (68f)
```
  if(moving && (mouse->buttons&7)){
      oin = winput;
      band = mouse->buttons & 3; // left or middle click

      sweeping = true;
      if(band)
          i = bandsize(winput);
      else
          i = drag(winput, &r);
      sweeping = false;

      if(i != nil){
          if(winput == oin){
              if(band)
                  wsendctlmesg(winput, Reshaped, i->r, i);
```

```
            else
                wsendctlmesg(winput, Moved, r, i);
            cornercursor(winput, mouse->xy, true);
        }else
            freeimage(i);
    }
}
```

Uses `Moved` 91a, `Reshaped` 91a, `bandsize()` 115, `cornercursor()` 85c, `drag()` 111c, `mouse` 48c, `sweeping` 89d, and `wsendctlmesg()` 91c.

Beware the near-identical names: the `Reshaped` sent here is a *control message*, not the `Reshape` *menu command* of the previous section.

7.4 Window control message: `Wctlmesg`

The `Wctlmesg`[91b] structure is the message type sent from the mouse thread (or other sources) to a window's `winctl()`[71b] thread through its `Window.cctl`[74d] channel. The message carries the operation type (`Reshaped`[91a], `Moved`[91a], etc.), a rectangle (for operations that change geometry), and an optional image (the new window layer after a resize or move). Note the past tense in the enum names (`Reshaped`, not Reshape)—the image has already been allocated by the time the message arrives; the window thread just needs to adopt it.

⟨*enum* `wctlmesgkind` 91a⟩≡ (333b)

```
enum ControlMessage /* control messages */
{
    Reshaped, // Resized, Hide/Expose
    Moved,
    ⟨Wctlmesgkind cases 106b⟩
};
```

⟨*struct* `Wctlmesg` 91b⟩≡ (333b)

```
struct Wctlmesg
{
    // enum<Wctlmesgkind>
    int  type;

    Rectangle r;
    // option<ref<Image>>
    Image *image;
};
```

`wsendctlmesg()`[91c] is the *sender* side: it packs the type, rectangle, and image into a `Wctlmesg` and sends it on the window's `cctl` channel.

⟨*function* `wsendctlmesg` 91c⟩≡ (342b)

```
/// mousethread | new | delete | wclose | move | resize | whide | ... -> <>
void
wsendctlmesg(Window *w, int type, Rectangle r, Image *image)
{
    Wctlmesg wcm;

    wcm.type = type;
    wcm.r = r;
    wcm.image = image;

    ⟨wsendctlmesg() trace w->id and type 278f⟩
    send(w->cctl, &wcm);
}
```

wctlmesg()[92a] is the *receiver* side, running in the window's winctl() thread: it switches on the message type and carries out the operation.

⟨*function* wctlmesg 92a⟩≡ (339c)

```
/// winctl -> <>
int
wctlmesg(Window *w, int m, Rectangle r, Image *i)
{
    char buf[64];

    ⟨wctlmesg() trace w->id and m 278g⟩
    switch(m){
    ⟨wctlmesg() cases 106c⟩
    default:
        error("unknown control message");
        break;
    }
    return m;
}
```

Uses error() 282c.

As with the menu, the wctlmesg() cases are presented gradually in the sections that follow.

7.5 Window creation

Window creation is the most important operation in rio—it ties together all the subsystems: graphics (allocating a layer on the desktop), concurrency (creating a new winctl()[71b] thread), the filesystem (mounting /mnt/wsys for the new window), and processes (forking a shell). When the user selects "New" from the system menu, sweep()[101d] lets them draw a rectangle, then new()[92c] creates everything needed for the new window (see Section 7.5.7 for the high-level overview).

⟨button3menu() *cases* 92b⟩+≡ (88b) ◁90a 105d▷

```
case New:
    new(sweep(), false, scrolling, 0, nil, "/bin/rc", nil);
    break;
```

Uses New-8 89c, new() 92c, scrolling 225c, and sweep() 101d.

7.5.1 Window thread creation: new()

new() orchestrates the creation of a window in several steps: allocate channels, build the Window[49] structure with wmk()[94c], add it to the global windows[51a] array, create a winctl()[71b] thread, fork a shell process (if pid == 0), and register the window's name in the graphics layer system.

When pid is non-zero, the calling process already exists (e.g., an external program that mounted /srv/rio.<user> and just needs a window created for it—see Section 13.6 for this external mount mechanism. For this section you can read the code assuming pid == 0 (the common case).

⟨*function* new 92c⟩≡ (340c)

```
/// ((right click -> button3menu) | wctlnew) -> <>
Window*
new(Image *i, bool hideit, bool scrollit, int pid, char *dir, char *cmd, char **argv)
{
    Channel *cm, *ck, *cctl;
    Channel *cpid;
    Mousectl *mc;
    Window *w;
    ⟨new() other locals 97b⟩
```

```
    ⟨new() sanity check i 94a⟩

    ⟨new() channels creation 93a⟩
    cpid = chancreate(sizeof(int), 0);
    ⟨new() sanity check channels 94b⟩

    ⟨new() mc allocation 93b⟩

    // create Window data structure
    w = wmk(i, mc, ck, cctl, scrollit);
    free(mc); /* wmk copies *mc */

    // growing array
    windows = erealloc(windows, ++nwindow * sizeof(Window*));
    windows[nwindow-1] = w;
    ⟨new() if hideit 119a⟩
    ⟨new() trace w->id 279a⟩

    // create a new thread! for this new window!
    threadcreate(winctl, w, STACK);

    if(!hideit)
        wcurrent(w);

    flushimage(display, true);

    // create a new process
    ⟨new() if pid == 0, create winshell process and set pid 97c⟩
    ⟨new() sanity check pid received from winshell 99a⟩
    wsetpid(w, pid, true);

    // create a new layer
    wsetname(w);

    if(dir)
        w->dir = estrdup(dir);

    chanfree(cpid);
    return w;
}
```

Uses `STACK` 61a, `nwindow` 51b, `wcurrent()` 105a, `winctl()` 71b, `windows` 51a, `wmk()` 94c, `wsetname()` 101a, and `wsetpid()` 98c.

`new()` starts by creating the three channels the window's `winctl()` thread will listen on: `Window.mc`[74a] for mouse events, `Window.ck`[72b] for keyboard runes, and `Window.cctl`[74d] for control messages (buffered, so senders need not block).

⟨`new()` *channels creation* 93a⟩≡ (92c)

```
cm = chancreate(sizeof(Mouse), 0);
ck = chancreate(sizeof(Rune*), 0);
cctl = chancreate(sizeof(Wctlmesg), 4);
```

⟨`new()` *mc allocation* 93b⟩≡ (92c)

```
mc = emalloc(sizeof(Mousectl));
*mc = *mousectl;
mc->image = i;
mc->c = cm;
```

Uses `mousectl` 48a.

⟨new() *sanity check i* 94a⟩≡ (92c)
```
if(i == nil)
    return nil;
```

⟨new() *sanity check channels* 94b⟩≡ (92c)
```
if(cm==nil || ck==nil || cctl==nil)
    error("new: channel alloc failed");
```
Uses error() 282c.

7.5.2 Window allocation: wmk()

wmk() is the constructor for the Window[49] structure. It assigns a unique id (monotonically increasing) and a topped[51c] counter that tracks stacking order—higher values mean more recently focused.

⟨*function* wmk 94c⟩≡ (342b)
```
/// new -> <>
Window*
wmk(Image *i, Mousectl *mc, Channel *ck, Channel *cctl, bool scrolling)
{
    Window *w;
    Rectangle r;

    ⟨wmk() colors initialisation 175a⟩

    w = emalloc(sizeof(Window));
    w->i = i;
    w->screenr = i->r;
    // set logical (0,0) origin for the client (-Borderwidth undoes the
    // client-side border inset in getgenwindow()); screenr keeps the physical position.
    originwindow(i, Pt(-Borderwidth, -Borderwidth), i->r.min);
    w->cursorp = nil;

    w->id = ++id;
    w->topped = ++topped;

    w->label = estrdup("<unnamed>");

    ⟨wmk() channels settings 95a⟩
    ⟨wmk() channels creation 142b⟩
    ⟨wmk() textual window settings 173⟩
    ⟨wmk() process settings 96f⟩

    ⟨wmk() drawing border 95b⟩
    ⟨wmk() drawing scrollbar 180a⟩

    incref(w); /* ref will be removed after mounting; avoids delete before ready to be deleted */
    return w;
}
```
Uses id-17 50b and topped-16 51c.

The initial incref() at the end is subtle: it prevents the window from being freed before the child process has finished mounting rio's filesystem. The matching decref() happens in winshell()[97d] after filsysmount()[99c] completes (via wclose()[109b]).

originwindow() used above is a libdraw routine (see the Graphics book [Pad16c]); it repositions a window's coordinate system. Recall that a rio window image is a layer—an image allocated on the desktop[48d] Screen and composited over the others by the draw device. A layer has an internal rectangle r (its logical coordinates) that is independent of where the layer is actually shown on screen. originwindow(w, log, scr) asks

the draw device to make the layer's logical origin `log` while compositing it at screen position `scr`, and updates `w->r` (and `w->clipr`) locally to match. Afterwards, code drawing into the image works in the logical coordinates of `w->r`, and the draw device's layer machinery translates them to physical pixels each time it composites the layer onto the screen—real conversion work that lives inside the `draw` device (see the GRAPHICS book [Pad16c]).

Just after setting `w->i` and capturing the physical position in `w->screenr`, `wmk()` gives the window its logical coordinates with `originwindow()`. The shift is to `Pt(-Borderwidth, -Borderwidth)`, not `(0,0)`: a client opens this image and insets it by `Borderwidth` (in `libdraw`'s `gengetwindow()`) to skip the border, so pre-shifting the image by `-Borderwidth` lands the application's drawable exactly at `(0,0)`. `screenr` still holds the physical position, and since `i->r.min` is now `-Borderwidth` the `mousethread()`[66a] conversion delivers mouse positions in that same `(0,0)`-based frame.

A concrete example makes the chain clearer. Suppose `Borderwidth` is 4 and `rio` placed the window at `(100,50)-(500,350)`. Just before the call `i->r` and `screenr` are both that physical rectangle—which is why `i->r.min` is passed as the on-screen position, and why `screenr` is captured first, before the shift moves `i->r` away:

```
originwindow(i, (-4,-4), i->r.min):   shift logical origin to (-4,-4),
                                      keep showing at i->r.min = (100,50)
after it:
   i->r    = (-4,-4)-(396,296)   logical coords (whole image, border incl.)
   screenr = (100,50)-(500,350)  physical position (unchanged)
   client's drawable = insetrect(i->r, 4) = (0,0)-(392,292)

     screen (framebuffer)
   +-------------------------------------------+
   | screenr.min (100,50)                      |
   |   +---- border 4px ----------------+      |
   |   |  (0,0) content   = screen (104,54)    |
   |   |     * (10,10)    = screen (114,64)    |
   |   +--------------------------------+      |
   +-------------------------------------------+

draw   logical->screen (draw device):  P - i->r.min + screenr.min
       (10,10) - (-4,-4) + (100,50) = (114,64)
mouse  screen->logical (mousethread):  M + (i->r.min - screenr.min)
       (114,64) + ((-4,-4) - (100,50)) = (10,10)
```

The two directions are exact inverses: a client drawing at logical `(10,10)` hits screen pixel `(114,64)`, and a click on that pixel is reported back to the client as `(10,10)`.

`wmk()` stores the channels created in `new()`[92c] into the `Window`.

⟨`wmk()` *channels settings* 95a⟩≡ (94c)
```
w->mc = *mc;
w->ck = ck;
w->cctl = cctl;
```

`wmk()` then draws the window's initial border.

⟨`wmk()` *drawing border* 95b⟩≡ (94c)
```
wborder(w, Selborder);
```
Uses `Selborder` 67 and `wborder()` 96a.

⟨*function* wborder 96a⟩≡ (342b)
```
void
wborder(Window *w, int type)
{
    Image *col;

    ⟨wborder() sanity check w 96e⟩
    ⟨wborder() if holding 270f⟩
    else{
        if(type == Selborder)
            col = titlecol;
        else
            col = lighttitlecol;
    }

    border(w->i, w->i->r, Selborder, col, ZP);
}
```
Uses Selborder 67, lighttitlecol-21 96c, and titlecol-20 96b.

The border uses `titlecol`[96b] for the selected window and the paler `lighttitlecol`[96c] for unselected ones.

⟨*global* titlecol 96b⟩≡ (342b)
```
static Image *titlecol;
```

⟨*global* lighttitlecol 96c⟩≡ (342b)
```
static Image *lighttitlecol;
```

⟨wmk() *extra colors initialisation* 96d⟩≡ (175a) 176b▷
```
titlecol      = allocimage(display, Rect(0,0,1,1), CMAP8, true, DGreygreen);
lighttitlecol= allocimage(display, Rect(0,0,1,1), CMAP8, true, DPalegreygreen);
```
Uses lighttitlecol-21 96c and titlecol-20 96b.

⟨wborder() *sanity check w* 96e⟩≡ (96a)
```
if(w->i == nil)
    return;
```

`wmk()`'s last step initializes the window's process-related fields: `Window.notefd`[98b] (the note channel to the window's process, not open yet) and `Window.dir`[96g] (its working directory, copied from `startdir`).

⟨wmk() *process settings* 96f⟩≡ (94c)
```
w->notefd = -1;
w->dir = estrdup(startdir);
```
Uses startdir 96h.

`dir` is the window's working directory, exported to clients as `/dev/wdir`.

⟨Window *other fields* 96g⟩+≡ (49) ◁51d 114b▷
```
char  *dir; // /dev/wdir
```

⟨*global* startdir 96h⟩≡ (338a)
```
char  *startdir;
```

⟨main() *locals* 96i⟩≡ (58) 195d▷
```
char buf[256];
```

⟨main() *set some globals* 96j⟩≡ (58) 195e▷
```
if(getwd(buf, sizeof buf) == nil)
    startdir = estrdup(".");
else
    startdir = estrdup(buf);
```

7.5.3 Window process creation: `winshell()`

For each new window, `rio` forks a new process (`winshell()`[97d]) that will `exec` a shell (usually `rc -i`).

⟨*global* `rcargv` 97a⟩≡ (340c)
```
char *rcargv[] = { "rc", "-i", nil };
```

⟨`new()` *other locals* 97b⟩≡ (92c)
```
void **arg;
```

⟨`new()` *if pid == 0, create* `winshell` *process and set* `pid` 97c⟩≡ (92c)
```
if(pid == 0){
    arg = emalloc(5 * sizeof(void*));
    arg[0] = w;
    arg[1] = cpid;
    arg[2] = cmd;
    if(argv == nil)
        arg[3] = rcargv;
    else
        arg[3] = argv;
    arg[4] = dir;

    ⟨new() trace before winshell() 279b⟩
    proccreate(winshell, arg, 8192);
    pid = recvul(cpid);
    ⟨new() trace after winshell() 279c⟩

    free(arg);
}
```
Uses `rcargv` 97a and `winshell()` 97d.

Why a separate `winshell()` proc rather than just forking here? Because in a `libthread` program new processes are created with `proccreate`, not a raw `fork` or `rfork`[1]. In that proc `winshell()` sets up a private namespace (with `rfork`), mounts `rio`'s filesystem, and `exec`s the shell as you will see soon. The new process id is handed back over `cpid` (`recvul`) so `rio` can remember it—to kill the process when the window is deleted—and `arg` is just the marshalling needed to pass several parameters through `proccreate`'s single `void*`.

⟨*function* `winshell` 97d⟩≡ (343a)
```
/// new -> proccreate(<>, ...)
void
winshell(void *args)
{
    Window *w;
    Channel *pidc;
    void **arg;
    char *cmd, *dir;
    char **argv;
    errorneg1 err;

    arg = args;

    w    = arg[0];
    pidc = arg[1];
    cmd  = arg[2];
    argv = arg[3];
```

[1] All of a `libthread` program's procs must stay in one *rendezvous group*—that is what makes channels work across procs—so a new proc is made with `proccreate`, an `rfork` of `RFPROC|RFMEM` (never `RFREND`) that preserves the group, gives the proc its own stack, and registers it with the thread scheduler. A bare `fork`/`rfork` to *create* a proc would fall outside that machinery. (`rfork` *is* the right tool for adjusting a single process's namespace and fds, as `winshell()` does next.) See `thread(2)` and the LIBCORE book [Pad16a].

```
    dir  = arg[4];

    ⟨winshell() trace cmd 279d⟩
    // copy namespace/file-descriptors/environment-variables (do not share)
    rfork(RFNAMEG|RFFDG|RFENVG);

    ⟨winshell() adjust namespace 99b⟩
    ⟨winshell() reassign STDIN/STDOUT after namespace adjustment 100a⟩

    if(wclose(w) == false){ /* remove extra ref hanging from creation */

        ⟨winshell() trace before procexec() 279e⟩
        notify(nil);
        dup(STDOUT, STDERR); // STDERR = STDOUT
        if(dir)
            chdir(dir);

        // Exec!!
        procexec(pidc, cmd, argv);
        _exits("exec failed"); // should never be reached
    }
}
```

Uses wclose() 109b.

Before exec'ing, `winshell()` must set up the correct namespace: it calls `rfork(RFNAMEG|RFFDG|RFENVG)` to get a private copy of the namespace (`RFNAMEG`), file descriptors (`RFFDG`), and environment variables (`RFENVG`), then mounts `rio`'s filesystem so that the child's `/dev/cons`, `/dev/mouse`, and other files are served by `rio`'s window. This is the key mechanism: the shell thinks it is talking to normal device files, but those files are actually virtual files served by `rio`.

Why private copies? The child needs its own namespace so that mounting `rio`'s filesystem on `/dev` and `/mnt/wsys` does not affect the parent or other windows. It also needs private file descriptors so that closing and reopening `/dev/cons` points to this window's virtual console, not the parent's. Note though that the child still inherits the pipe file descriptor to `rio`'s filesystem server (via the fork/exec shared-fd model)—`RFFDG` copies the fd table, so the child keeps a copy of `Filsys.cfd`[53a].

Each window tracks the `pid` of its child process and keeps an open file descriptor to `/proc/<pid>/notepg`, which allows `rio` to send notes (signals) to the process group—for instance, to interrupt a running command when the window is deleted (see Section 13.1).

⟨Window *process fields* 98a⟩≡ (49) 98b ▷

```
  int  pid;
```

⟨Window *process fields* 98b⟩+≡ (49) ◁ 98a

```
  // /proc/<pid>/notepg
  fdt   notefd;
```

⟨*function* wsetpid 98c⟩≡ (342b)

```
  /// new -> <>
  void
  wsetpid(Window *w, int pid, bool dolabel)
  {
      char buf[128];
      fdt fd;

      w->pid = pid;
      ⟨wsetpid() trace w->id and w->pid 279f⟩

      if(dolabel){
          sprint(buf, "rc %d", pid);
```

```
            free(w->label);
            w->label = estrdup(buf);
        }

        sprint(buf, "/proc/%d/notepg", pid);
        fd = open(buf, OWRITE|OCEXEC);
        if(w->notefd > 0)
            close(w->notefd);
        w->notefd = fd;
    }
```

⟨new() *sanity check pid received from winshell* 99a⟩≡ (92c)

```
    if(pid == 0){
        /* window creation failed */
        wsendctlmesg(w, Deleted, ZR, nil);
        chanfree(cpid);
        return nil;
    }
```

Uses `Deleted` 106b and `wsendctlmesg()` 91c.

7.5.4 Namespace adjustments: `filsysmount()`

Remember that the namespace setup below runs in the new child proc, `winshell()`[97d], which has just rforked a private namespace—so the bindings it makes affect only the window's own process, not `rio` or the other windows. `filsysmount()`[99c] mounts `rio`'s client pipe onto `/mnt/wsys`, passing the window ID as the *mount spec* so that `rio` knows which window the client belongs to. It then binds `/mnt/wsys` before `/dev`, so that `/dev/cons`, `/dev/mouse`, etc. resolve to `rio`'s virtual files rather than the real device files. This bind-before-`/dev` is the key step that makes `rio`'s virtualization work.

⟨winshell() *adjust namespace* 99b⟩≡ (97d)

```
    err = filsysmount(filsys, w->id);
    ⟨winshell() sanity check err filsysmount 100d⟩
```

Uses `filsys` 53b and `filsysmount()` 99c.

⟨*function* filsysmount 99c⟩≡ (346b)

```
    /// winshell -> <>
    /*
     * Called only from a different FD group
     */
    errorneg1
    filsysmount(Filsys *fs, int id)
    {
        char buf[32];
        errorneg1 err;

        ⟨filsysmount() trace start 279g⟩

        close(fs->sfd); /* close server end so mount won't hang if exiting */
        sprint(buf, "%d", id); // mount spec string
        err = mount(fs->cfd, -1, "/mnt/wsys", MREPL, buf);
        ⟨filsysmount() sanity check err mount 100e⟩
        err = bind("/mnt/wsys", "/dev", MBEFORE);
        ⟨filsysmount() sanity check err bind 100f⟩
        ⟨filsysmount() trace end 279h⟩
        return OK_0;
    }
```

The child process can now simply opens /dev/cons for stdin and stdout, but these are now served by rio.

⟨winshell() *reassign STDIN/STDOUT after namespace adjustment* 100a⟩≡ (97d)
```
// reassign stdin/stdout to virtualized /dev/cons from filsysmount
close(STDIN);
err = open("/dev/cons", OREAD);
⟨winshell() sanity check err open cons stdin 100b⟩
close(STDOUT);
err = open("/dev/cons", OWRITE);
⟨winshell() sanity check err open cons stdout 100c⟩
```

The rest of winshell() is error handling: on any failure it reports a zero pid back to the parent (through pidc) and bails out with threadexits.

⟨winshell() *sanity check err open cons stdin* 100b⟩≡ (100a)
```
if(err < 0){
    fprint(STDERR, "can't open /dev/cons: %r\n");
    sendul(pidc, 0);
    threadexits("/dev/cons");
}
```

⟨winshell() *sanity check err open cons stdout* 100c⟩≡ (100a)
```
if(err < 0){
    fprint(STDERR, "can't open /dev/cons: %r\n");
    sendul(pidc, 0);
    threadexits("open"); /* BUG? was terminate() */
}
```

⟨winshell() *sanity check err filsysmount* 100d⟩≡ (99b)
```
if(err < 0){
    fprint(STDERR, "mount failed: %r\n");
    sendul(pidc, 0);
    threadexits("mount failed");
}
```

filsysmount() itself checks both of its steps—the mount() and the bind()—returning -1 if either fails.

⟨filsysmount() *sanity check err mount* 100e⟩≡ (99c)
```
if(err < 0){
    fprint(STDERR, "mount failed: %r\n");
    return ERROR_NEG1;
}
```

⟨filsysmount() *sanity check err bind* 100f⟩≡ (99c)
```
if(err < 0){
    fprint(STDERR, "bind failed: %r\n");
    return ERROR_NEG1;
}
```

7.5.5 Public layer: wsetname()

The namespace adjustments above handle /dev/cons and /dev/mouse but not /dev/draw—as I mentioned before, virtualizing the drawing protocol would be too slow. Instead, the client directly talks to the display server but needs to find its window's image. wsetname()[101a] publishes the window's image under a unique name

(e.g., `window.2.0`), and the client discovers it through `/dev/winname` via `geninitdraw()` (see the Graphics book [Pad16c]).

⟨*function* `wsetname` 101a⟩≡ (342b)
```
/// mousethread -> new -> <>
void
wsetname(Window *w)
{
    int i, n;
    char err[ERRMAX];

    n = sprint(w->name, "window.%d.%d", w->id, w->namecount++);

    if(nameimage(w->i, w->name, true) > 0)
        return;
    // else
    ⟨wsetname() if image name already in use, try another name 101c⟩
}
```

`Window.namecount`[101b] is bumped on every `wsetname()`, giving each published image a fresh name (`window.2.0`, `window.2.1`, ...). This matters on resize: the window gets a brand-new image, so it needs a new name for the client to re-discover it through `/dev/winname`.

⟨`Window` *id fields* 101b⟩+≡ (49) ◁50e
```
uint  namecount;
```

The name can still be taken if the previous image under it has not been freed yet—for instance a client slow to release its old window image after a resize. `wsetname()` then appends a letter (`A`..`Z`) until it finds a free name.

⟨`wsetname()` *if image name already in use, try another name* 101c⟩≡ (101a)
```
for(i='A'; i<='Z'; i++){
    // ok try again
    if(nameimage(w->i, w->name, true) > 0)
        return;
    // else, retry

    errstr(err, sizeof err);
    if(strcmp(err, "image name in use") != 0)
        break;
    w->name[n] = i;
    w->name[n+1] = '\0';
}
// else
w->name[0] = '\0';
fprint(STDERR, "rio: setname failed: %s\n", err);
```

7.5.6 Mouse action `sweep()`

Earlier, walking through `new()`, we passed over how the new window's rectangle is chosen. That is `sweep()`[101d]'s job. `sweep()` lets the user interactively draw a rectangle for the new window. It shows a cross cursor, waits for the right button to be pressed, then tracks the mouse while the button is held, allocating and freeing temporary window images to give visual feedback. Once the button is released, the temporary image is replaced by a final one with `Refbackup` (so the graphics library saves what is underneath for proper overlapping).

⟨*function* `sweep` 101d⟩≡ (340c)
```
Image*
sweep(void)
{
    Point p0, p;
    Rectangle r;
```

```
    Image *i, *oi;

    i = nil;

    menuing = true;
    riosetcursor(&crosscursor, true);
    while(mouse->buttons == 0)
        readmouse(mousectl);

    p0 = onscreen(mouse->xy);
    p = p0;
    r = Rpt(p0, p);
    oi = nil;

    while(mouse->buttons == 4){ // right click
        readmouse(mousectl);

        if(mouse->buttons != 4 && mouse->buttons != 0)
            break;
        if(!eqpt(mouse->xy, p)){
            p = onscreen(mouse->xy);
            r = canonrect(Rpt(p0, p));

            if(Dx(r)>5 && Dy(r)>5){
                i = allocwindow(desktop, r, Refnone, 0xEEEEEEFF); /* grey */
                freeimage(oi);
                ⟨sweep() sanity check i 103a⟩
                oi = i;
                border(i, r, Selborder, red, ZP);
                flushimage(display, true);
            }
        }
    }
    ⟨sweep() sanity check mouse buttons, i, and rectangle size 103b⟩
    oi = i;
    i = allocwindow(desktop, oi->r, Refbackup, DWhite);
    freeimage(oi);
    ⟨sweep() sanity check i 103a⟩
    border(i, r, Selborder, red, ZP);
    cornercursor(input, mouse->xy, true);
    goto Return;
⟨sweep() Rescue handler 103c⟩

 Return:
    moveto(mousectl, mouse->xy); /* force cursor update; ugly */
    menuing = false;
    return i;
}
```

Uses `Selborder` 67, `cornercursor()` 85c, `crosscursor` 81, `desktop` 48d, `input` 51e, `menuing` 102a, `mouse` 48c, `mousectl` 48a, `onscreen()` 102b, `red` 48f, and `riosetcursor()` 85b.

We finally see the `menuing`[102a] flag mentioned earlier in `wsetcursor()`[87]: it is set while `sweep()` waits for the user to draw a window rectangle. During this time, `rio` must not honor per-window cursor changes, since the cross cursor must stay active.

⟨*global* `menuing` 102a⟩≡ (338a)

```
bool menuing;/* menu action is pending; waiting for window to be indicated */
```

⟨*function* `onscreen` 102b⟩≡ (340c)

```
Point
```

```
onscreen(Point p)
{
    p.x = max(view->clipr.min.x, p.x);
    p.x = min(view->clipr.max.x, p.x);
    p.y = max(view->clipr.min.y, p.y);
    p.y = min(view->clipr.max.y, p.y);
    return p;
}
```
Uses max() 284b and min() 284a.

⟨sweep() *sanity check i* 103a⟩≡ (101)
```
if(i == nil)
    goto Rescue;
```

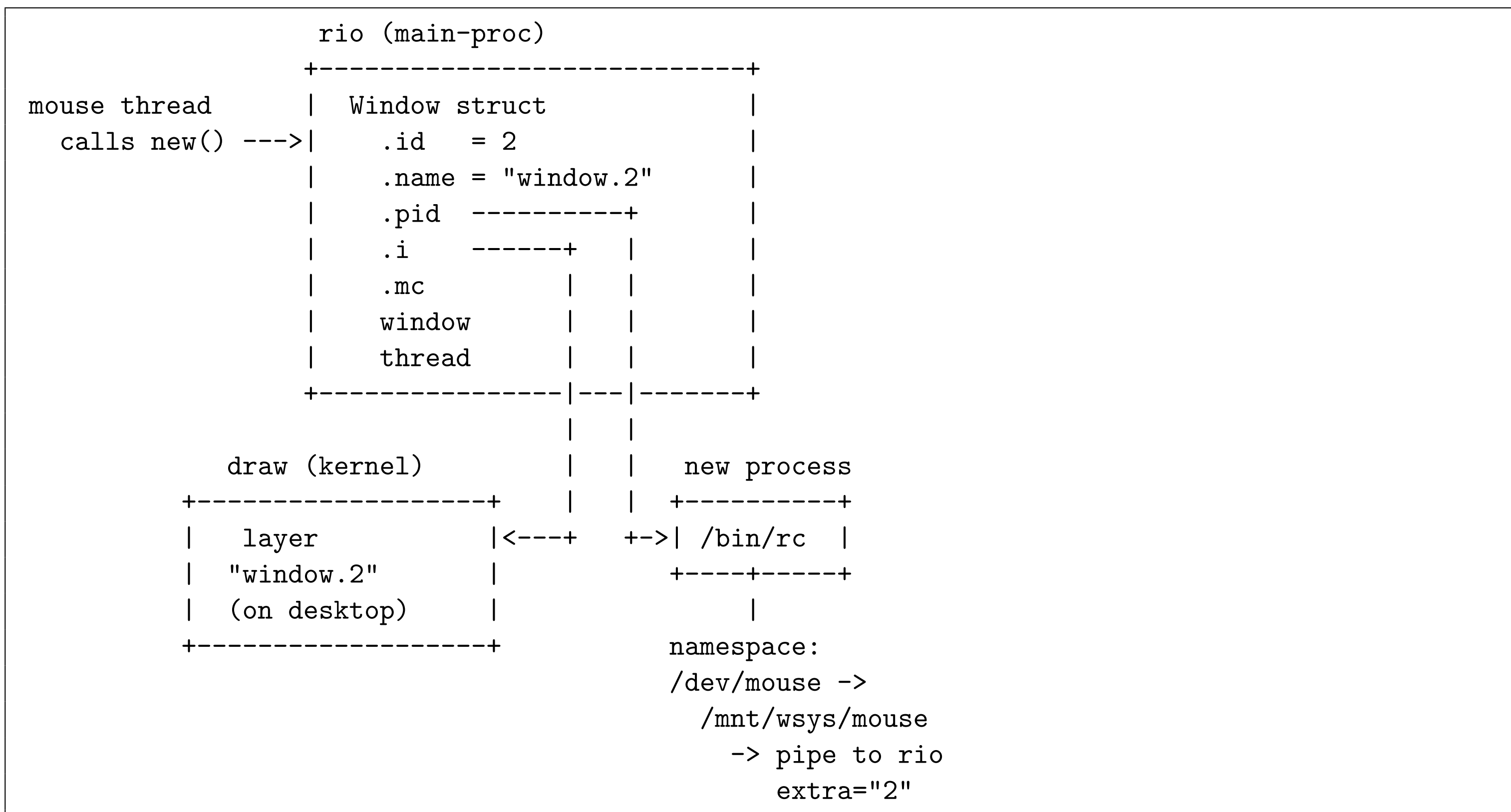
⟨sweep() *sanity check mouse buttons, i, and rectangle size* 103b⟩≡ (101d)
```
if(mouse->buttons != 0)
    goto Rescue;
if(i==nil || Dx(i->r) < 100 || Dy(i->r) < 3*font->height)
    goto Rescue;
```
Uses mouse 48c.

⟨sweep() *Rescue handler* 103c⟩≡ (101d)
```
Rescue:
    freeimage(i);
    i = nil;
    cornercursor(input, mouse->xy, true);
    while(mouse->buttons)
        readmouse(mousectl);
```
Uses cornercursor() 85c, input 51e, mouse 48c, and mousectl 48a.

7.5.7 Trace of a new window creation

We have now seen all the code behind window creation; this section retraces it end to end as a summary. Creating a window from the menu puts the mouse thread in control (the mouse trace itself comes in Section 9.2.3), and from there new()[92c] builds five things: a new process (forked and exec'd in winshell()[97d]); its private namespace (the mount() and bind() of filsysmount()[99c], detailed in the KERNEL book [Pad14]); a new Window[49] structure, linked into windows[51a] by wmk()[94c]; a new window thread, winctl()[71b], with its channels; and a new image layer, published under a unique name by wsetname()[101a]. The diagram below shows the resulting entities and how they link; how the per-message traffic is then multiplexed over the shared pipe is left to Chapter 8.

Putting it together, the new Window is the hub: it ties the shell process (through its pid), the image layer on the desktop, and the window thread that the mouse and keyboard threads feed through its channels.



7.6 Window focus

After all the machinery of window creation, changing focus takes far less code. Clicking on an unfocused window brings it to the front via `wtop()`[104b], which calls `topwindow()` (see the GRAPHICS book [Pad16c]) to raise the image layer, then `wcurrent()`[105a] to update the global `input`[51e] pointer and repaint the border colors—the focused window gets a different border color than unfocused ones.

⟨`mousethread()` *click on unfocused window, set w* 104a⟩≡ (70c)

```
w = wtop(mouse->xy);
```

Uses `mouse` 48c and `wtop()` 104b.

⟨*function* `wtop` 104b⟩≡ (342b)

```
Window*
wtop(Point pt)
{
    Window *w;

    w = wpointto(pt);
    if(w){
        if(w->topped == topped)
            return nil;
        // else
        topwindow(w->i); // window.h (was in draw.h)
        wcurrent(w);
        flushimage(display, true);
        w->topped = ++topped;
    }
    return w;
}
```

Uses `topped-16` 51c, `wcurrent()` 105a, and `wpointto()` 69.

`wcurrent()` updates the global `input` pointer (the focused window) and repaints both the old and new focus windows via `wrepaint()`[105b]. The repaint is necessary because `rio` draws different border colors for focused versus unfocused windows—the focused window gets `titlecol`[96b] (dark grey-green) and unfocused ones get `lighttitlecol`[96c] (pale grey-green). If the window is in text mode, `wrepaint()` also redraws the text content because the foreground/background colors may differ for the focused window.

⟨*function* `wcurrent` 105a⟩≡ (342b)
```
void
wcurrent(Window *w)
{
    Window *oi;

    ⟨wcurrent() if wkeyboard 227e⟩
    oi = input;
    // updated input!
    input = w;

    if(oi && oi != w)
        wrepaint(oi);
    if(w){
        wrepaint(w);
        wsetcursor(w, false);
    }
    ⟨wcurrent() wakeup w and oi 212d⟩
}
```
Uses `input` 51e, `wrepaint()` 105b, and `wsetcursor()` 87.

⟨*function* `wrepaint` 105b⟩≡ (342b)
```
void
wrepaint(Window *w)
{

    ⟨wrepaint() update cols 175b⟩
    ⟨wrepaint() after updated cols, redraw content if mouse not opened 186⟩

    if(w == input){
        wborder(w, Selborder);
        wsetcursor(w, false);
    }else
        wborder(w, Unselborder);
}
```
Uses `Selborder` 67, `Unselborder` 105c, `input` 51e, `wborder()` 96a, and `wsetcursor()` 87.

⟨*constant* `Unselborder` 105c⟩≡ (333a)
```
Unselborder  = 1, /* border of unselected window */
```

7.7 Window deletion

Deleting a window is a two-phase process. First, `rio` sends a "hangup" note to the window's process group (via `Window.notefd`[98b]) and starts a timeout proc. If the process exits cleanly, the window transitions from `Deleted`[106b] to `Exited`[109e] and the `winctl()`[71b] thread terminates. If the process does not exit within the timeout, `deletethread()`[108b] forces cleanup. This careful approach avoids blocking the window manager thread while waiting for a potentially unresponsive process.

⟨`button3menu()` *cases* 105d⟩+≡ (88b) ◁92b 111a▷
```
case Delete:
    delete();
```

```
    break;
```
Uses Delete-11 89c and delete() 106a.

7.7.1 Menu handler: delete()

delete()[106a] is the menu handler: it calls pointto()[110a] with wait=true (wait for the user to complete the click), then sends a Deleted[106b] control message to the selected window's winctl()[71b] thread.

⟨*function* delete 106a⟩≡ (340c)
```
void
delete(void)
{
    Window *w;

    w = pointto(true);
    if(w)
        wsendctlmesg(w, Deleted, ZR, nil);
}
```
Uses Deleted 106b, pointto() 110a, and wsendctlmesg() 91c.

⟨Wctlmesgkind *cases* 106b⟩≡ (91a) 109e ▷
```
Deleted,
```

⟨wctlmesg() *cases* 106c⟩≡ (92a) 109f ▷
```
case Deleted:
    ⟨wctlmesg() break if window was deleted 107b⟩
    write(w->notefd, "hangup", 6);
    proccreate(deletetimeoutproc, estrdup(w->name), 4096);
    wclosewin(w);
    break;
```
Uses Deleted 106b, deletetimeoutproc() 107c, and wclosewin() 106d.

The actual cleanup is asynchronous—the winctl() thread handles the message by sending a "hangup" note to the process group and starting a timeout.

wclosewin() handles the immediate visual cleanup: it marks the window as deleted, clears the focus if this was the active window, removes it from the hidden[118c] array (in case it was hidden), removes it from the global windows[51a] array with memmove(), and frees the layer image so it disappears from the screen. Note that the Window[49] struct itself is not freed here—that happens later in the Exited handler, after the child process has terminated.

⟨*function* wclosewin 106d⟩≡ (342b)
```
void
wclosewin(Window *w)
{
    int i;

    w->deleted = true;

    if(w == input){
        input = nil;
        wsetcursor(w, false);
    }
    ⟨wclosewin() if wkeyboard 227f⟩
    ⟨wclosewin() remove w from hidden 119b⟩

    for(i=0; i<nwindow; i++)
        if(windows[i] == w){
            --nwindow;
```

```
            memmove(windows+i, windows+i+1, (nwindow-i)*sizeof(Window*));

            freeimage(w->i);
            w->i = nil;
            return;
        }
    error("unknown window in closewin");
}
```
Uses error() 282c, input 51e, nwindow 51b, windows 51a, and wsetcursor() 87.

We have already seen Window.deleted[107a] used as a guard in winctl() (to skip flushimage()) and in wkeyctl()[73b] (to discard input). Now that I present the deletion logic, its role becomes clearer: deleted is set to true as soon as a Deleted control message is processed, even though the window's thread has not exited yet. During this interval, the window is being torn down—its image has been freed, a hangup note has been sent to the client—but pending events may still arrive on its channels. The deleted flag lets every handler bail out early rather than operate on a half-destroyed window.

⟨Window *config fields* 107a⟩+≡ (49) ◁52f 160a▷
```
bool deleted;
```

Here is the first of those bail-outs in the flesh: at the top of wctlmesg()[92a], before handling any window-control message, rio checks w->deleted and returns at once. A Wctlmesg[91b] can already be in flight from several sources—a menu action, a /mnt/wsys/wctl write, a pending request—when the window is torn down, so each handler must re-check rather than assume no message arrives after deletion.

⟨wctlmesg() *break if window was deleted* 107b⟩≡ (272b 160e 106c)
```
if(w->deleted)
    break;
```

7.7.2 Timeout for slow clients: deletetimeoutproc() and deletethread()

The timeout mechanism handles clients that hold a reference to the window image (via nameimage()) and refuse to let go. deletetimeoutproc()[107c] is a separate process (not a thread) because sleep() is blocking. After 750ms it sends the window name on deletechan[107d].

⟨*function* deletetimeoutproc 107c⟩≡ (339c)
```
/// winctl -> wctlmesg(Deleted) -> proccreate(<>)
void
deletetimeoutproc(void *v)
{
    char *s = v;

    sleep(750); /* remove window from screen after 3/4 of a second */
    sendp(deletechan, s);
}
```
Uses deletechan 107d.

⟨*global* deletechan 107d⟩≡ (338a)
```
// chan<string> (listener = deletethread, sender = deletetimeoutproc)
Channel* deletechan;
```

⟨main() *communication channels creation* 107e⟩+≡ (58) ◁60f 108d▷
```
deletechan   = chancreate(sizeof(char*), 0);
```

The `deletethread()`[108b] on the other end looks up the image by name and, if the client still holds it, moves it off-screen with `originwindow()` so at least it is invisible. The `freeimage()` that follows drops the reference obtained by `namedimage()`, but the client may still hold its own reference—the image only truly disappears when all references are gone. Note this is a different use of `originwindow()` than in `wmk()`[94c]: there it established a logical origin for the window, whereas here it just shoves the window's screen position off the edge of the display to hide it.

⟨`main()` *threads creation* 108a⟩+≡ (58) ◁61b 108e▷
```
threadcreate(deletethread, nil, STACK);
```

⟨*function* `deletethread` 108b⟩≡ (340b)
```
/* thread to make Deleted windows that the client still holds disappear offscreen after an interval */
void
deletethread(void*)
{
    char *s;
    Image *i;

    threadsetname("deletethread");

    for(;;){
        s = recvp(deletechan);

        i = namedimage(display, s);
        if(i != nil){
            /* move it off-screen to hide it, since client is slow in letting it go */
            originwindow(i, i->r.min, view->r.max);
        }
        freeimage(i);
        free(s);
    }
}
```
Uses `deletechan` 107d.

7.7.3 Two-phase teardown: `Deleted` and `Exited`

The second phase of deletion uses reference counting. Each mount of `rio`'s filesystem increments the window's reference count (in `xfidattach()`[127c]), and closing such a mount decrements it—the 9P `clunk` operation, detailed in Chapter 8, signalled here via `winclosechan`[108c]. When the last reference drops to zero, `wclose()`[109b] sends the `Exited`[109e] message, which triggers the final cleanup: freeing the frame, closing the note file descriptor, freeing all channels, and freeing the `Window`[49] struct itself.

The `winclosechan` channel bridges the filesystem server process and the main thread group. The filesystem process cannot directly manipulate window state (different thread group), so it sends the window pointer on this channel, and `winclosethread()`[109a] calls `wclose()` in the correct context.

⟨*global* `winclosechan` 108c⟩≡ (338a)
```
// chan<ref<Window>> (listener = winclosethread, sender = filsyswalk | filsysclunk )
Channel *winclosechan; /* chan(Window*); */
```

⟨`main()` *communication channels creation* 108d⟩+≡ (58) ◁107e
```
winclosechan = chancreate(sizeof(Window*), 0);
```

⟨`main()` *threads creation* 108e⟩+≡ (58) ◁108a 222d▷
```
threadcreate(winclosethread, nil, STACK);
```

⟨*function* winclosethread 109a⟩≡ (340b)
```
/* thread to allow fsysproc to synchronize window closing with main proc */
void
winclosethread(void*)
{
    Window *w;

    threadsetname("winclosethread");

    for(;;){
        w = recvp(winclosechan);
        wclose(w);
    }
}
```
Uses `wclose()` 109b and `winclosechan` 108c.

⟨*function* wclose 109b⟩≡ (342b)
```
bool
wclose(Window *w)
{
    int i;

    i = decref(w);
    if(i > 0)
        return false;
    ⟨wclose() sanity check i 109c⟩
    ⟨wclose() sanity check w 109d⟩

    wsendctlmesg(w, Exited, ZR, nil);
    return true;
}
```
Uses `Exited` 109e and `wsendctlmesg()` 91c.

⟨`wclose()` *sanity check i* 109c⟩≡ (109b)
```
if(i < 0)
    error("negative ref count");
```
Uses `error()` 282c.

⟨`wclose()` *sanity check w* 109d⟩≡ (109b)
```
if(!w->deleted)
    wclosewin(w);
```
Uses `wclosewin()` 106d.

We can now see the second message, `Exited`, and the actual teardown it triggers.

⟨`Wctlmesgkind` *cases* 109e⟩+≡ (91a) ◁106b 144c▷
```
Exited,
```

⟨`wctlmesg()` *cases* 109f⟩+≡ (92a) ◁106c 114a▷
```
case Exited:
    frclear(&w->frm, true);
    close(w->notefd);
    chanfree(w->mc.c);
    chanfree(w->ck);
    chanfree(w->cctl);
    chanfree(w->conswrite);
    chanfree(w->consread);
    chanfree(w->mouseread);
    chanfree(w->wctlread);
    free(w->raw);
```

```
    free(w->r);
    free(w->dir);
    free(w->label);
    free(w);
    break;
```

Uses `Exited` 109e and `frclear()` 289d.

7.7.4 Mouse action `pointto()`

`pointto()`[110a] is the shared "pick a window" gesture used by `Delete`, `Move`, and `Reshape`. It changes the cursor to a gunsight, waits for the user to click, and returns the window under the cursor (or `nil` if the user clicked on the background or with the wrong button). The `wait` parameter controls whether the function waits for the button to be released before returning. `Delete` passes `true` (wait for a full click), while `Move` passes `false` because the same button press will continue into the `drag()`[111c] phase.

⟨*function* `pointto` 110a⟩≡ (340c)

```
Window*
pointto(bool wait)
{
    Window *w;

    menuing = true;
    riosetcursor(&sightcursor, true);

    while(mouse->buttons == 0)
        readmouse(mousectl);

    if(mouse->buttons == 4)
        w = wpointto(mouse->xy);
    else
        w = nil;

    if(wait){
        while(mouse->buttons){
            ⟨pointto() cancel pointto if clicked another button 110b⟩
            readmouse(mousectl);
        }
        if(w != nil && wpointto(mouse->xy) != w)
            w = nil;
    }

    cornercursor(input, mouse->xy, false);
    moveto(mousectl, mouse->xy); /* force cursor update; ugly */
    menuing = false;
    return w;
}
```

Uses `cornercursor()` 85c, `input` 51e, `menuing` 102a, `mouse` 48c, `mousectl` 48a, `riosetcursor()` 85b, `sightcursor` 82b, and `wpointto()` 69.

⟨`pointto()` *cancel pointto if clicked another button* 110b⟩≡ (110a)

```
if(mouse->buttons!=4 && w != nil){ /* cancel */
    cornercursor(input, mouse->xy, false);
    w = nil;
}
```

Uses `cornercursor()` 85c, `input` 51e, and `mouse` 48c.

7.8 Window move

Moving a window is a two-step mouse action: first `pointto()`[110a] lets the user select which window to move (by showing a gunsight cursor), then `drag()`[111c] tracks the mouse while redrawing a rubber-band border to show the new position. When the button is released, the old image content is copied into a new window at the destination.

⟨`button3menu()` *cases* 111a⟩+≡ (88b) ◁105d 113b▷

```
  case Move:
      move();
      break;
```

Uses `Move-10` 89c and `move()` 111b.

7.8.1 Menu handler: `move()`

`move()`[111b] is the system-menu "Move" handler: it picks a window, lets the user `drag()`[111c] it to a new position, and sends a `Moved` message.

⟨*function* `move` 111b⟩≡ (340c)

```
  void
  move(void)
  {
      Window *w;
      Image *i;
      Rectangle r;

      w = pointto(false);
      if(w == nil)
          return;
      i = drag(w, &r);
      if(i)
          wsendctlmesg(w, Moved, r, i);
      cornercursor(input, mouse->xy, true);
  }
```

Uses `Moved` 91a, `cornercursor()` 85c, `drag()` 111c, `input` 51e, `mouse` 48c, `pointto()` 110a, and `wsendctlmesg()` 91c.

The `Moved`[91a] case in `wctlmesg()`[92a] shares its handler with `Reshaped`[91a]—a move is just a reshape where the dimensions stay the same and the old content is copied over. We will see this code soon when we cover a window resize.

7.8.2 Mouse action `drag()`

`drag()`[111c] implements the visual feedback loop for moving a window. It computes `dm`, the offset from the mouse to the window's top-left corner, so the window follows the cursor at the same relative position where the user grabbed it. While the button is held, `drawborder()`[112b] draws a red rubber-band outline at the current position. On release, a new window image is allocated at the final position and the old content is copied into it with `draw`. If the user releases the wrong button (anything other than a clean release), the move is cancelled: the cursor is moved back to its original position (`om`) and `nil` is returned.

⟨*function* `drag` 111c⟩≡ (340c)

```
  Image*
  drag(Window *w, Rectangle *rp)
  {
      Image *i;
      Image *ni = nil;
      Point p, op, d, dm, om;
      Rectangle r;
```

```
    i = w->i;

    menuing = true;
    om = mouse->xy;
    riosetcursor(&boxcursor, true);

    dm = subpt(mouse->xy, w->screenr.min);
    d = subpt(i->r.max, i->r.min);
    op = subpt(mouse->xy, dm);

    drawborder(Rect(op.x, op.y, op.x+d.x, op.y+d.y), true);
    flushimage(display, true);

    while(mouse->buttons == 4){
        p = subpt(mouse->xy, dm);
        if(!eqpt(p, op)){
            // will move previously drawn rectangle thx to originwindow
            drawborder(Rect(p.x, p.y, p.x+d.x, p.y+d.y), true);
            flushimage(display, true);
            op = p;
        }
        readmouse(mousectl);
    }

    r = Rect(op.x, op.y, op.x+d.x, op.y+d.y);
    drawborder(r, false);

    cornercursor(w, mouse->xy, true);
    moveto(mousectl, mouse->xy); /* force cursor update; ugly */
    menuing = false;

    flushimage(display, true);
    if(mouse->buttons == 0)
        ni = allocwindow(desktop, r, Refbackup, DWhite);
    ⟨drag() sanity check mouse buttons and ni 112a⟩
    draw(ni, ni->r, i, nil, i->r.min);
    *rp = r;
    return ni;
}
```

Uses `boxcursor` 82a, `cornercursor()` 85c, `desktop` 48d, `drawborder()` 112b, `menuing` 102a, `mouse` 48c, `mousectl` 48a, and `riosetcursor()` 85b.

⟨`drag()` *sanity check mouse buttons and ni* 112a⟩≡ (111c)

```
if(mouse->buttons!=0 || ni==nil){
    moveto(mousectl, om);
    while(mouse->buttons)
        readmouse(mousectl);
    *rp = Rect(0, 0, 0, 0);
    return nil;
}
```

Uses `mouse` 48c and `mousectl` 48a.

`drawborder()` draws a rubber-band rectangle using four separate edge images (not a single rectangle), because a single layer would obscure the window content underneath. Using four thin strips keeps the interior transparent. `drawedge()`[113a] optimizes updates: if the edge dimensions have not changed (as in a move), it simply repositions the existing image with `originwindow` rather than freeing and reallocating.

⟨*function* `drawborder` 112b⟩≡ (340c)

```
void
```

```
drawborder(Rectangle r, bool show)
{
    static Image *b[4];
    int i;
    if(!show){
        for(i = 0; i < 4; i++){
            freeimage(b[i]);
            b[i] = nil;
        }
    }else{
        r = canonrect(r);
        drawedge(&b[0], Rect(r.min.x, r.min.y, r.min.x+Borderwidth, r.max.y));
        drawedge(&b[1], Rect(r.min.x+Borderwidth, r.min.y, r.max.x-Borderwidth, r.min.y+Borderwidth));
        drawedge(&b[2], Rect(r.max.x-Borderwidth, r.min.y, r.max.x, r.max.y));
        drawedge(&b[3], Rect(r.min.x+Borderwidth, r.max.y-Borderwidth, r.max.x-Borderwidth, r.max.y));
    }
}
```

Uses `drawedge()` 113a.

⟨*function* `drawedge` 113a⟩≡ (340c)

```
void
drawedge(Image **bp, Rectangle r)
{
    Image *b = *bp;
    if(b != nil && Dx(b->r) == Dx(r) && Dy(b->r) == Dy(r))
        originwindow(b, r.min, r.min);
    else{
        freeimage(b);
        *bp = allocwindow(desktop, r, Refbackup, DRed);
    }
}
```

Uses `desktop` 48d.

7.9 Window resize

Resizing a window can be triggered from the menu or by dragging a border; either way it ends in a `Reshaped`[91a] message.

⟨`button3menu()` *cases* 113b⟩+≡ (88b) ◁111a 118a▷

```
case Reshape:
    resize();
    break;
```

Uses `Reshape-9` 89c and `resize()` 113c.

7.9.1 Menu handler: `resize()`

`resize()`[113c] is almost identical to `move()`[111b]: select a window with `pointto()`[110a], draw a new rectangle with `sweep()`[101d], and send a `Reshaped`[91a] message. The key difference is that `pointto()` uses `wait=true` here (the user completes one click to select, then does a separate sweep gesture), and `sweep()` lets the user define an entirely new rectangle rather than dragging the existing one.

⟨*function* `resize` 113c⟩≡ (340c)

```
void
resize(void)
{
    Window *w;
```

```
    Image *i;

    w = pointto(true);
    if(w == nil)
        return;
    i = sweep();
    if(i)
        wsendctlmesg(w, Reshaped, i->r, i);
}
```

Uses `Reshaped` 91a, `pointto()` 110a, `sweep()` 101d, and `wsendctlmesg()` 91c.

Here is the shared handler. `Moved` and `Reshaped` take the same path (`Moved` is just a resize that keeps the same size): `wresize()`[114c] adopts the new image, re-publishes the window name (so the client reconnects via `getwindow()`), and reflows the text content for a textual window.

⟨`wctlmesg()` *cases* 114a⟩+≡ (92a) ◁109f 144d▷

```
case Moved:
case Reshaped:
    if(w->deleted){
        freeimage(i);
        break;
    }
    w->screenr = r;
    strcpy(buf, w->name);
    wresize(w, i, m==Moved);
    w->wctlready = true;

    proccreate(deletetimeoutproc, estrdup(buf), 4096);

    if(Dx(r) > 0){
        if(w != input)
            wcurrent(w);
    }else if(w == input)
        wcurrent(nil);
    flushimage(display, true);
    break;
```

Uses `Moved` 91a, `Reshaped` 91a, `deletetimeoutproc()` 107c, `input` 51e, `wcurrent()` 105a, and `wresize()` 114c.

⟨`Window` *other fields* 114b⟩+≡ (49) ◁96g 142a▷

```
bool   resized;
```

`wresize()` swaps the window's image. If this is a move (or the dimensions are unchanged), the old content is copied into the new image with `draw()`. The old image is freed, and `wsetname()`[101a] publishes the new image under an incremented name so the client can reconnect via `getwindow()`. The `Window.resized` flag and `mouse.counter` increment ensure the client gets a resize event on its next `/dev/mouse` read, which prompts it to call `getwindow()` to pick up the new image.

⟨*function* `wresize` 114c⟩≡ (342b)

```
void
wresize(Window *w, Image *i, bool move)
{
    Rectangle r, or;

    or = w->i->r;
    if(move || (Dx(or)==Dx(i->r) && Dy(or)==Dy(i->r)))
        draw(i, i->r, w->i, nil, w->i->r.min);
    freeimage(w->i);
    w->i = i;
    // claude: re-establish the logical (0,0) origin on the new image (see wmk);
    // claude: the content copy above ran first, in physical coordinates.
```

```
    originwindow(i, Pt(-Borderwidth, -Borderwidth), i->r.min);
    wsetname(w); // publish new window name by incrementing namecount
    w->mc.image = i;

    ⟨wresize() textual window updates 205a⟩

    wborder(w, Selborder);
    w->topped = ++topped;

    w->resized = true;
    w->mouse.counter++;
}
```

Uses `Selborder` 67, `topped-16` 51c, `wborder()` 96a, and `wsetname()` 101a.

7.9.2 Mouse action `bandsize()`

`bandsize()`[115] is the alternative resize gesture triggered by clicking on a window's border (rather than using the menu). It determines which corner or edge the user grabbed via `whichcorner()`[86b], snaps the cursor to that corner with `cornerpt()`[117] and `wmovemouse()`[116c], then enters a tracking loop where `whichrect()`[116d] computes the new rectangle by anchoring the opposite corner/edge and letting the grabbed one follow the mouse. This feels like "pulling" a corner to resize.

⟨*function* `bandsize` 115⟩≡ (340c)

```
Image*
bandsize(Window *w)
{
    Image *i;
    Rectangle r, or;
    Point p, startp;
    int which, but;

    p = mouse->xy;
    but = mouse->buttons;
    which = whichcorner(w, p);
    p = cornerpt(w->screenr, p, which);
    wmovemouse(w, p);

    readmouse(mousectl);
    r = whichrect(w->screenr, p, which);
    drawborder(r, true);

    or = r;
    startp = p;

    while(mouse->buttons == but){
        p = onscreen(mouse->xy);
        r = whichrect(w->screenr, p, which);
        if(!eqrect(r, or) && goodrect(r)){
            drawborder(r, true);
            flushimage(display, true);
            or = r;
        }
        readmouse(mousectl);
    }

    p = mouse->xy;
    drawborder(or, false);
    flushimage(display, true);
```

```
    wsetcursor(w, true);
    ⟨bandsize() sanity check mouse buttons, rectanglr or, point p 116a⟩
    i = allocwindow(desktop, or, Refbackup, DWhite);
    ⟨bandsize() sanity check i 116b⟩
    border(i, r, Selborder, red, ZP);
    return i;
}
```

Uses `Selborder` 67, `cornerpt()` 117, `desktop` 48d, `drawborder()` 112b, `goodrect()` 261b, `mouse` 48c, `mousectl` 48a, `onscreen()` 102b, `red` 48f, `whichcorner()` 86b, `whichrect()` 116d, `wmovemouse()` 116c, and `wsetcursor()` 87.

⟨`bandsize()` *sanity check mouse buttons, rectanglr* `or`, *point* `p` 116a⟩≡ (115)

```
if(mouse->buttons!=0 || Dx(or)<100 || Dy(or)<3*font->height){
    while(mouse->buttons)
        readmouse(mousectl);
    return nil;
}
if(abs(p.x - startp.x) + abs(p.y - startp.y) <= 1)
    return nil;
```

Uses `mouse` 48c and `mousectl` 48a.

⟨`bandsize()` *sanity check i* 116b⟩≡ (115)

```
if(i == nil)
    return nil;
```

The minimum size check (`Dx < 100` or `Dy < 3*font->height`) prevents the user from shrinking a window below a usable size. The "movement threshold" check (`abs(p.x-startp.x) + abs(p.y-startp.y) <= 1`) prevents an accidental click on the border from triggering a resize.

`wmovemouse()` does the reverse conversion: given a point in the window's logical coordinates it adds `screenr.min - i->r.min` to get physical screen coordinates, then warps the pointer there with `moveto()`.

⟨*function* `wmovemouse` 116c⟩≡ (342b)

```
/*
 * Convert back to physical coordinates
 */
void
wmovemouse(Window *w, Point p)
{
    p.x += w->screenr.min.x - w->i->r.min.x;
    p.y += w->screenr.min.y - w->i->r.min.y;
    moveto(mousectl, p);
}
```

Uses `mousectl` 48a.

`whichrect()` and `cornerpt()` use a 3x3 grid encoding (from `whichcorner()`): 0=top-left, 1=top-edge, 2=top-right, 3=left-edge, 5=right-edge, 6=bottom-left, 7=bottom-edge, 8=bottom-right (4 would be the center, which is never used). `cornerpt()` snaps the cursor to the actual corner or edge position, and `whichrect()` computes a new rectangle by moving only the grabbed edge/corner while the opposite side stays fixed.

⟨*function* `whichrect` 116d⟩≡ (340c)

```
Rectangle
whichrect(Rectangle r, Point p, int which)
{
    switch(which){
    case 0: /* top left */
        r = Rect(p.x, p.y, r.max.x, r.max.y);
        break;
    case 2: /* top right */
        r = Rect(r.min.x, p.y, p.x, r.max.y);
```

```
        break;
    case 6: /* bottom left */
        r = Rect(p.x, r.min.y, r.max.x, p.y);
        break;
    case 8: /* bottom right */
        r = Rect(r.min.x, r.min.y, p.x, p.y);
        break;
    case 1: /* top edge */
        r = Rect(r.min.x, p.y, r.max.x, r.max.y);
        break;
    case 5: /* right edge */
        r = Rect(r.min.x, r.min.y, p.x, r.max.y);
        break;
    case 7: /* bottom edge */
        r = Rect(r.min.x, r.min.y, r.max.x, p.y);
        break;
    case 3:  /* left edge */
        r = Rect(p.x, r.min.y, r.max.x, r.max.y);
        break;
    }
    return canonrect(r);
}
```

⟨*function* cornerpt 117⟩≡ (340c)

```
Point
cornerpt(Rectangle r, Point p, int which)
{
    switch(which){
    case 0: /* top left */
        p = Pt(r.min.x, r.min.y);
        break;
    case 2: /* top right */
        p = Pt(r.max.x,r.min.y);
        break;
    case 6: /* bottom left */
        p = Pt(r.min.x, r.max.y);
        break;
    case 8: /* bottom right */
        p = Pt(r.max.x, r.max.y);
        break;
    case 1: /* top edge */
        p = Pt(p.x,r.min.y);
        break;
    case 5: /* right edge */
        p = Pt(r.max.x, p.y);
        break;
    case 7: /* bottom edge */
        p = Pt(p.x, r.max.y);
        break;
    case 3:  /* left edge */
        p = Pt(r.min.x, p.y);
        break;
    }
    return p;
}
```

7.10 Window visibility

The last window-management feature is visibility: hiding a window and later bringing it back.

⟨button3menu() *cases* 118a⟩+≡ (88b) ◁113b 119d▷
```
case Hide:
    hide();
    break;
```
Uses Hide-12 89c and hide() 118b.

7.10.1 Menu handler: hide()

⟨*function* hide 118b⟩≡ (340c)
```
void
hide(void)
{
    Window *w;

    w = pointto(true);
    if(w == nil)
        return;
    whide(w);
}
```
Uses pointto() 110a and whide() 118e.

7.10.2 hidden windows

The hidden[118c] array is a simple fixed-size list of currently hidden windows. Its contents are appended to the right-click system menu (after the standard New/Reshape/Move/Delete/Hide entries), giving the user a way to bring hidden windows back. The menu entries show each Window.label[50e] (typically rc <pid>).

⟨*global* hidden 118c⟩≡ (338a)
```
// array<ref<Window>> (size valid = nhidden)
Window *hidden[100];
```

⟨*global* nhidden 118d⟩≡ (338a)
```
int  nhidden;
```

The hiding trick is in the choice of allocimage() versus allocwindow(): whide()[118e] allocates a plain in-memory image (not a layer on the desktop) and sends a Reshaped[91a] message with it. The Reshaped handler in wctlmesg()[92a] calls wresize()[114c], which swaps the window's image—so the window now draws to an off-screen buffer. The old layer image is freed, making the window disappear. The window's thread continues to run normally, just drawing to an invisible image.

⟨*function* whide 118e⟩≡ (340c)
```
int
whide(Window *w)
{
    Image *i;
    int j;

    for(j=0; j<nhidden; j++)
        if(hidden[j] == w) /* already hidden */
            return -1;

    i = allocimage(display, w->screenr, w->i->chan, false, DWhite);
    if(i){
        hidden[nhidden++] = w;
```

```
        wsendctlmesg(w, Reshaped, ZR, i);
        return 1;
    }
    return 0;
}
```

Uses Reshaped 91a, hidden 118c, nhidden 118d, and wsendctlmesg() 91c.

⟨new() *if hideit* 119a⟩≡ (92c)

```
if(hideit){
    hidden[nhidden++] = w;
    w->screenr = ZR;
}
```

Uses hidden 118c and nhidden 118d.

⟨wclosewin() *remove w from hidden* 119b⟩≡ (106d)

```
// delete_list(w, hidden)
for(i=0; i<nhidden; i++)
    if(hidden[i] == w){
        --nhidden;
        memmove(hidden+i, hidden+i+1, (nhidden-i)*sizeof(hidden[0]));
        hidden[nhidden] = nil;
        break;
    }
```

Uses hidden 118c and nhidden 118d.

The hidden entries start at index `Hidden` in `menu3str`, just past the fixed menu items, and the list is `nil`-terminated.

⟨button3menu() *menu3str adjustments with hidden windows* 119c⟩≡ (88b)

```
for(i=0; i<nhidden; i++)
    menu3str[i+Hidden] = hidden[i]->label;
menu3str[i+Hidden] = nil;
```

Uses Hidden-14 89c, hidden 118c, menu3str 89b, and nhidden 118d.

7.10.3 Menu handler: `unhide()`

Unhiding reverses the trick: `wunhide()`[120b] allocates a real layer (via `allocwindow()`) and sends a `Reshaped`[91a] message. The `Reshaped` handler swaps the off-screen image for the new on-screen layer, publishes it via `wsetname()`[101a], and the window reappears. The menu integration is handled by the `default` case in the `button3menu()`[88b] switch—any menu index past the standard entries corresponds to a hidden window.

⟨button3menu() *cases* 119d⟩+≡ (88b) ◁118a

```
default:
    unhide(i);
    break;
```

Uses unhide() 119e.

⟨*function* unhide 119e⟩≡ (340c)

```
void
unhide(int h)
{
    Window *w;

    h -= Hidden;
    w = hidden[h];
    ⟨unhide() sanity check w 120a⟩
    wunhide(h);
}
```

Uses Hidden-14 89c, hidden 118c, and wunhide() 120b.

⟨`unhide()` *sanity check w* 120a⟩≡ (119e)

```
if(w == nil)
    return;
```

⟨*function* `wunhide` 120b⟩≡ (340c)

```
int
wunhide(int h)
{
    Image *i;
    Window *w;

    w = hidden[h];
    i = allocwindow(desktop, w->i->r, Refbackup, DWhite);
    if(i){
        --nhidden;
        memmove(hidden+h, hidden+h+1, (nhidden-h)*sizeof(Window*));
        wsendctlmesg(w, Reshaped, w->i->r, i);
        return 1;
    }
    return 0;
}
```

Uses `Reshaped` 91a, `desktop` 48d, `hidden` 118c, `nhidden` 118d, and `wsendctlmesg()` 91c.

Chapter 8

Filesystem Server

Until now, I have presented the code of an application that can create rectangles on the screen, move and resize them. But for those rectangles to become real windows, they need a filesystem server that virtualizes the device files (`/dev/cons`, `/dev/mouse`, etc.) for each client process.

This chapter presents the 9P operations that `rio` implements: `attach` (where the kernel, on behalf of the application, connects to `filsysproc()`[75]), `walk` (navigating to the desired file), `open`, `read`, `write`, `stat`, and `clunk` (similar to closing). Each operation has two parts: a `filsysxxx()` function that runs in the `filsysproc()` context and handles the fast path, and an `xfidxxx()` function that runs in a worker thread for operations that may block (like reading `/dev/cons` when no data is available) in the `main-proc` context (see Figure 2.11).

8.1 Clients, server, and the kernel in between

It is worth pinning down the words "client" and "server", which recur throughout this chapter and can feel abstract for a windowing system. `rio` is a server—more precisely a file server, since the resource it shares is a namespace of files (`/mnt/wsys/cons`, `/mnt/wsys/mouse`, ...) rather than storage or pixels. Its clients are the window applications: a shell, an editor, or `hellorio`, each running inside a window. The word "client" hides a subtlety, though. A window application never speaks 9P itself; it just calls `open()`, `read()`, and `write()` on ordinary-looking files (e.g., `/dev/cons`). It is the kernel—the mount driver set up when the window's namespace was built—that turns those system calls into 9P request messages and sends them down the pipe to `rio`. So when I say a request comes "from the client," the immediate sender is really the kernel, acting on behalf of the application. The full chain is `application` → kernel mount driver → pipe → `rio`.

8.2 Additional data structures

The operations in the following sections share two data structures, gathered here first: how a `Qid` names a file, and the directory table `dirtab`[123b] the server walks and lists.

8.2.1 Qid file identification: `Qxxx`

Each virtual file under `/mnt/wsys/` is identified by a `Qid` whose "path" encodes both the window ID and the file type (encoded by the `Qxxx`[122d] enum). The `QID()`[122a], `WIN()`[122b], and `FILE()`[122c] macros below pack and unpack these two pieces of information into a single integer. This lets the server quickly determine which window and which device file a request targets.

The integer they pack is the `Qid`'s `path` field. Despite the name it is not a pathname: in Plan 9, as in the 9P protocol generally, a `Qid`'s `path` is an opaque server-assigned integer that uniquely identifies a file—the role

a UNIX inode number plays. The server is free to compute it however it likes, so `rio` simply builds it from the window ID and file type instead of from any on-disk inode.

⟨*function* `QID` 122a⟩≡ (333b)
```
#define QID(winid,qxxx) ((winid<<8)|(qxxx))
```

⟨*function* `WIN` 122b⟩≡ (333b)
```
#define WIN(q) ((((ulong)(q).path)>>8) & 0xFFFFFF)
```

⟨*function* `FILE` 122c⟩≡ (333b)
```
#define FILE(q) (((ulong)(q).path) & 0xFF)
```

⟨*enum* `qid` 122d⟩≡ (333b)
```
enum Qxxx
{
    Qdir,   /* /dev for this window */
    ⟨Qxxx other cases 141a⟩

    QMAX,
};
```

I will introduce each `Qxxx` case gradually throughout the book, but to give an idea: the enum includes `Qcons` (the text console), `Qmouse` (mouse events), `Qconsctl` (console control such as raw mode), `Qcursor` (custom cursors), `Qwinname` (the layer name), `Qwindow` (the window image), `Qwctl` (window control commands), `Qsnarf` (the clipboard), and a few others. Each corresponds to a virtual device file that appears under `/mnt/wsys/`. For example, if window 2 opens `/mnt/wsys/cons`, the `Qid.path` is built by `QID(2, Qcons)`. Since `Qcons` is 1 in the enum and the window ID is shifted left by 8 bits, the resulting path is `0x201`:

```
qid.path = QID(winid=2, qxxx=Qcons=1)
         = (2 << 8) | 1
         = 0x0201

31           8 7         0
+------ ... --+-----------+
|  window ID  |   Qxxx    |
|     0x02    |   0x01    |
+------ ... --+-----------+

WIN(q)  = (path >> 8) & 0xFFFFFF  ->  2      (window 2)
FILE(q) = path & 0xFF             ->  1      (Qcons)
```

That `Qid` does not travel alone: it is stored in a `Fid`[53e], the fileserver's record for one open-file session. The `fid` number is assigned by the kernel on behalf of the client process, and `rio` maintains a `Fid` struct keyed by that number whose `Fid.qid`[53e] field holds the `Qid` `walk` resolved for it. The reason `Fid` and `Qid` are kept separate is that multiple processes—or the same process twice—can open the same file independently, each with its own mode and state. The `Qid` is the file's identity; the `Fid` tracks one particular session on that file.

8.2.2 Directory table: `Dirtab` and `dirtab`

The `Dirtab`[123a] structure is a static directory template: it maps a file name (e.g., `"cons"`, `"mouse"`) to its `Qxxx`[122d] enum value and permissions. The global `dirtab`[123b] array lists every virtual file that appears under

each window's directory. When the fileserver handles a `walk` or `stat` request, it looks up the target name in `dirtab` to find the corresponding `Qxxx` and permissions.

⟨*struct* Dirtab 123a⟩≡ (333b)
```
struct Dirtab
{
    char *name;
    // bitset<enum<QTxxx>>
    byte type;
    // enum<Qxxx>
    uint qid;
    uint perm;
};
```

⟨*global* dirtab 123b⟩≡ (346c)
```
Dirtab dirtab[]=
{
    { ".",  QTDIR, Qdir,  0500|DMDIR },
    ⟨dirtab array elements 141b⟩
    { nil, }
};
```
Uses `Qdir` 122d.

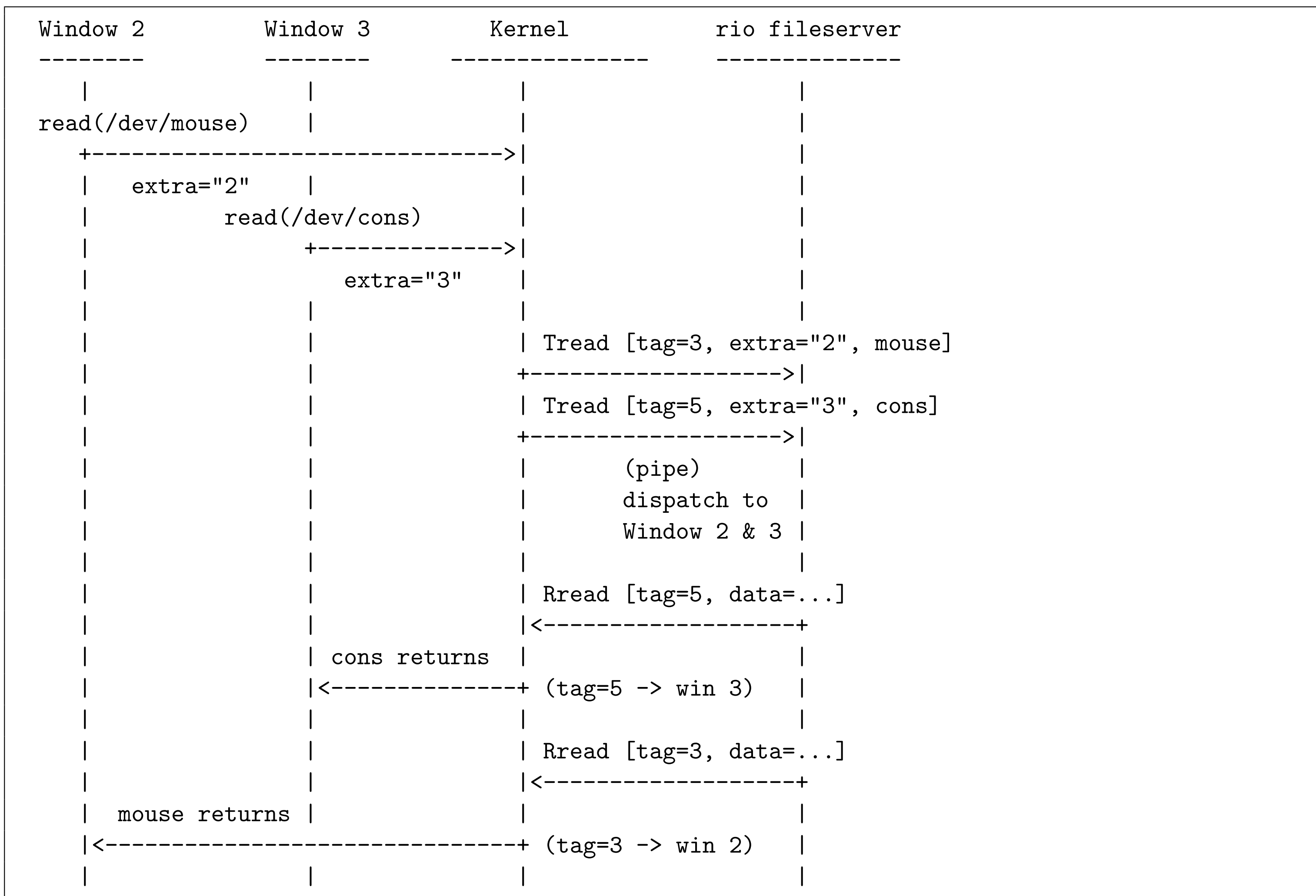
Again, I will introduce each `dirtab` entry—`mouse`, `cons`, `consctl`, `cursor`, `winname`, and so on—in the chapter or section devoted to that virtual device.

The `Fid` struct points back to a `Dirtab` entry via its `Fid.dir`[123c] field, so once a file is walked to, its metadata is readily available.

⟨`Fid` *other fields* 123c⟩+≡ (53e) ◁55a 151a▷
```
// ref<Dirtab>
Dirtab *dir;
```

8.3 Replying to the client: `filsysrespond()`

Before examining the individual 9P operations (`attach`, `walk`, etc.), here is the utility function used to send replies. In 9P, the reply message type is always the request type plus one (`Tattach` → `Rattach`, etc.), unless there is an error in which case it is `Rerror`. The response is sent through `Filsys.sfd`[53a], the server side of the pipe. Many functions call `filsysrespond()`[124]: every `filsysxxx()` handler and every `xfidxxx()` worker. These callers live in different procs—a `filsysxxx()` such as `filsyswalk()`[129] runs in the `fileserver` proc while an `xfidxxx()` such as `xfidopen()`[133a] runs in a worker thread of the `main-proc`—so they genuinely run in parallel (threads within one proc are scheduled cooperatively and never do). This is safe because pipe writes are atomic: the kernel guarantees that a complete message written to a pipe lands in one piece, so concurrent replies never interleave, and the tag/fid fields we will see soon let the kernel match replies to their requests. This is what lets a single pipe serve many windows at once. The diagram below follows two windows reading different files concurrently: the kernel writes each request atomically, stamped with the window id (`extra`, from the mount spec) and a unique *tag*; `rio` may reply in any order, and the kernel uses the tag to wake the right client.



With the atomic-pipe picture in mind, here at last is the code of `filsysrespond()` itself.

⟨*function* `filsysrespond` 124⟩≡ (346a)

```
/// filsysxxx | xfidxxx -> <>
Xfid*
filsysrespond(Filsys *fs, Xfid *x, Fcall *fc, char *err)
{
    int n;

    ⟨filsysrespond() if err 125c⟩
    else
        fc->type = x->req.type+1; // Reply type just after Transmit type

    fc->fid = x->req.fid;
    fc->tag = x->req.tag;

    ⟨filsysrespond() sanitize buf 125a⟩
    n = convS2M(fc, x->buf, messagesize);
    ⟨filsysrespond() sanity check n 125b⟩

    if(write(fs->sfd, x->buf, n) != n)
        error("write error in respond");
    ⟨filsysrespond() dump Fcall t if debug 278d⟩
    free(x->buf);
    x->buf = nil;
    return x;
```

```
}
```

Uses `error()` 282c and `messagesize` 76a.

⟨`filsysrespond()` *sanitize buf* 125a⟩≡ (124)

```
if(x->buf == nil)
    x->buf = malloc(messagesize);
```

Uses `messagesize` 76a.

⟨`filsysrespond()` *sanity check n* 125b⟩≡ (124)

```
if(n <= 0)
    error("convert error in convS2M");
```

Uses `error()` 282c.

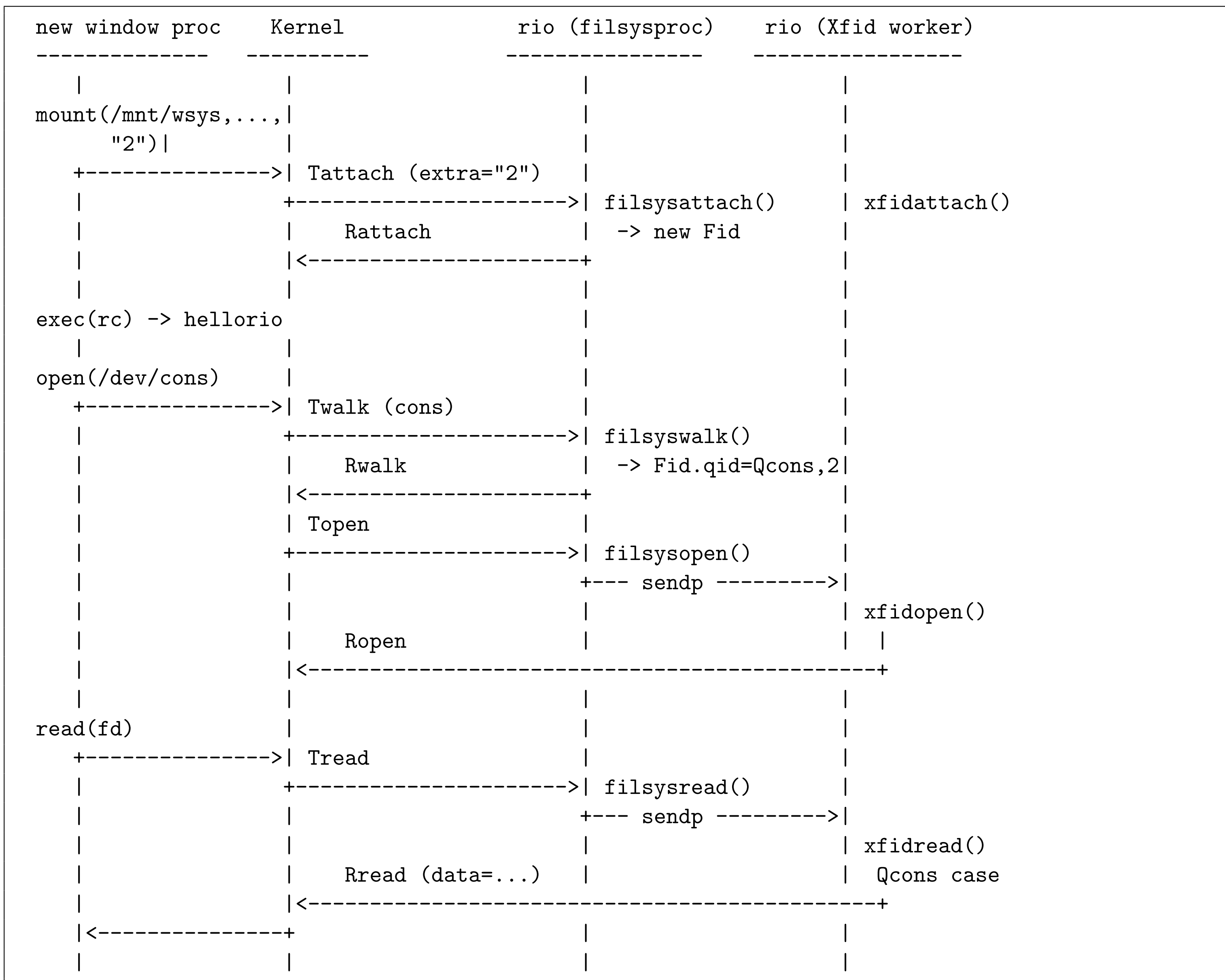
⟨`filsysrespond()` *if err* 125c⟩≡ (124)

```
if(err){
    fc->type = Rerror;
    fc->ename = err;
}
```

8.4 `attach`

`attach` is the first 9P operation a client performs after mounting. The mount spec string (e.g., "2", see the code of `filsysmount()`[99c]) carries the window ID, which `xfidattach()`[127c] uses to look up the corresponding `Window`[49]. This associates the client's fid with a specific window in `Fid.w`[55a], so that subsequent `walk`/`read`/`write` operations on that fid know which window they target.

Here is the typical sequence of 9P operations for a window. The `mount()` is done by the spawning process (before it `execs` to `rc`, in `filsysmount()`), which triggers `Tattach`. Later, when a client like `hellorio` opens `/dev/cons`, the kernel translates each system call into 9P messages on the pipe:



Notice that `walk` is handled entirely by `filsysproc()`[75], while the other operations (`attach`, `open`, `read`, `write`) are dispatched to `Xfid`[55b] worker threads via `sendp()`—`attach` included, as we are about to see it hand off to `xfidattach()`.

8.4.1 `filsysattach()`

The first operation is `attach`; here is its `filsysproc()`[75] half, `filsysattach()`[126]. `filsysproc()` reaches it by indexing the global `fcall`[56b] dispatch table with the request type: `fcall[Tattach]`.

⟨*function* `filsysattach` 126⟩≡ (346c)

```
static
Xfid*
filsysattach(Filsys *, Xfid *x, Fid *f)
{
    ⟨filsysattach() locals 127a⟩

    ⟨filsysattach() sanity check same user 127b⟩
    f->busy = true;
    f->open = false;
```

```
    f->qid.path = Qdir; // no window id, Qdir valid for all
    f->qid.type = QTDIR;
    f->qid.vers = 0;

    f->dir = &dirtab[0]; // entry for "."

    f->nrpart = 0;

    sendp(x->c, xfidattach);
    return nil;
}
```

Uses `Qdir` 122d, `dirtab` 123b, and `xfidattach()` 127c.

```
⟨filsysattach() locals 127a⟩≡                                    (126)
  Fcall fc;
```

```
⟨filsysattach() sanity check same user 127b⟩≡                    (126)
  if(strcmp(x->req.uname, x->fs->user) != 0)
      return filsysrespond(x->fs, x, &fc, Eperm);
```

Uses `Eperm` 280a and `filsysrespond()` 124.

8.4.2 `xfidattach()`

`xfidattach()`[127c] is the worker half: it resolves the mount's window id to a `Window`[49] and links it to the fid, under the `all`[128b] lock (further below).

```
⟨function xfidattach 127c⟩≡                                      (347)
  /// filsysattach !-> <>
  void
  xfidattach(Xfid *x)
  {
      Fcall fc;
      int id;
      Window *w = nil;
      bool newlymade = false;
      ⟨xfidattach() other locals 128c⟩

      fc.qid = x->f->qid;
      qlock(&all);
      ⟨xfidattach() if mount "new ..." 257c⟩
      else{
          // mount(fs->cfd, ..., "/mnt/wsys", ..., "2"), winid
          id = atoi(x->req.aname);
          w = wlookid(id);
      }
      x->f->w = w;
      ⟨xfidattach() sanity check w 128d⟩
      if(!newlymade) /* counteract dec() in winshell() */
          incref(w);
      qunlock(&all);

      filsysrespond(x->fs, x, &fc, nil);
  }
```

Uses `all` 128b, `filsysrespond()` 124, and `wlookid()` 128a.

Why does `attach` need a worker at all, when looking a window up in the shared `windows`[51a] table is something the `fileserver` proc does inline elsewhere (in `filsyswalk()`[129])? Because an `attach` can do more than look up: with a "new" spec string it actually creates a window (Section 13.6), and window creation must run in the `main-proc` context. Routing every `attach` through an `Xfid` worker, which runs in `main-proc`, keeps one code path valid for both the create and the lookup case. So `filsysattach()`[126] merely initializes the fid and `sendp()`s to `xfidattach()`. `walk` needs no worker because it never creates anything—it only resolves an existing window—so the `fileserver` proc handles it itself, taking the `all` lock just long enough to look the window up.

⟨*function* `wlookid` 128a⟩≡ (342b)

```
Window*
wlookid(int id)
{
    int i;

    for(i=0; i<nwindow; i++)
        if(windows[i]->id == id)
            return windows[i];
    return nil;
}
```

Uses `nwindow` 51b and `windows` 51a.

`all` below guards the global window table—the `windows` array and its `nwindow`[51b] count. Two of `rio`'s procs reach into it concurrently: the `fileserver` proc reads it (in `filsyswalk()` and `filsysread()`[135a], to resolve and to list windows) while `main-proc` threads grow and shrink it as windows are created and destroyed. Because those two procs run in parallel, the shared table needs a lock to keep a reader from seeing it mid-update.

⟨*global* `all` 128b⟩≡ (338a)

```
QLock all;   /* BUG */
```

⟨`xfidattach()` *other locals* 128c⟩≡ (127c) 257b ▷

```
char *err = Eunkid;
```

Uses `Eunkid` 281h.

⟨`xfidattach()` *sanity check w* 128d⟩≡ (127c)

```
if(w == nil){
    qunlock(&all);
    x->f->busy = false;
    filsysrespond(x->fs, x, &fc, err);
    return;
}
```

Uses `all` 128b and `filsysrespond()` 124.

8.5 walk

`walk` translates a file path (e.g., "cons" or "mouse") into a fid associated with the correct `Qid`. Unlike Unix's one-component-at-a-time `namei()`, 9P `walk` sends all path elements in a single message for network efficiency. The server must redo part of the kernel's path resolution work, looking up each name in `dirtab`[123b].

8.5.1 filsyswalk()

`walk` is the most intricate 9P operation because it must handle several cases in one routine: simple name lookup (scanning `dirtab`[123b] for "cons" or "mouse"), fid cloning (when `fid` and `newfid` differ), and ".." navigation.

The key design choice is that `qid` and `dir` are kept in local variables during the walk and only committed to `f` if every path component succeeds—a partial walk must not corrupt the fid state.

⟨*function* `filsyswalk` 129⟩≡ (346c)

```
static
Xfid*
filsyswalk(Filsys *fs, Xfid *x, Fid *f)
{
    Fcall fc;
    Qid qid;
    Dirtab *dir;
    int i;
    int id;
    uchar type;
    ulong path;
    Dirtab *d;
    Window *w;
    char *err;
    ⟨filsyswalk() other locals 130b⟩

    ⟨filsyswalk() sanity check if opened 131a⟩
    ⟨filsyswalk() if clone walk message 130c⟩

    fc.nwqid = 0;
    err = nil;

    /* update f->qid, f->dir only if walk completes */
    qid = f->qid;
    dir = f->dir;

    if(x->req.nwname > 0){
        for(i=0; i < x->req.nwname; i++){
            ⟨filsyswalk() sanity check current qid is a directory 131d⟩
            ⟨filsyswalk() if dotdot 130e⟩
            ⟨filsyswalk() if Qwsys, then goto Accept 210b⟩
            ⟨filsyswalk() if snarf 232c⟩
            id = WIN(f->qid);
            d = dirtab;
            d++; /* skip '.' */
            for(; d->name; d++)
                if(strcmp(x->req.wname[i], d->name) == 0){
                    path = d->qid;
                    type = d->type;
                    dir = d;
                    goto Accept;
                }
            break; /* file not found */
        }
        ⟨filsyswalk() sanity check i and err 131b⟩
    }
    ⟨filsyswalk() sanity check err and nwqid 131c⟩
    else if(fc.nwqid == x->req.nwname){
        f->dir = dir;
        f->qid = qid;
    }

    return filsysrespond(fs, x, &fc, err);
}
```

Uses `WIN` 122b, `dirtab` 123b, and `filsysrespond()` 124.

⟨filsyswalk() *accept label and code* 130a⟩≡ (130e)

```
Accept:
⟨filsyswalk() sanity check path elements 131e⟩
qid.type = type;
qid.path = QID(id, path);
qid.vers = 0;
fc.wqid[fc.nwqid++] = qid;
continue;
```

Uses `QID` 122a.

8.5.2 Cloning `fid`

When a `walk` message carries a `newfid` different from `fid`, the server must clone the fid before walking it. This is how the kernel implements operations like `open()` without losing the directory fid: it first clones the fid via a walk, then opens the clone (with a separate `open` message). The clone copies the window reference and bumps `incref()` so the window stays alive. After cloning, `f` is reassigned to `nf` so the rest of the walk operates on the new fid.

⟨filsyswalk() *other locals* 130b⟩≡ (129)

```
Fid *nf = nil;
```

⟨filsyswalk() *if clone walk message* 130c⟩≡ (129)

```
if(x->req.fid  != x->req.newfid){
    /* BUG: check exists */
    nf = newfid(fs, x->req.newfid);
    ⟨filsyswalk() when clone walk message, sanity check nf 130d⟩
    nf->busy = true;
    nf->open = false;
    nf->dir = f->dir;
    nf->qid = f->qid;
    nf->w = f->w;
    incref(f->w);
    nf->nrpart = 0; /* not open, so must be zero */
    f = nf; /* walk f */
}
```

Uses `newfid()` 54b.

⟨filsyswalk() *when clone walk message, sanity check nf* 130d⟩≡ (130c)

```
if(nf->busy)
    return filsysrespond(fs, x, &fc, "clone to busy fid");
```

Uses `filsysrespond()` 124.

8.5.3 '..'

Walking ".." always returns to the window's root directory (`Qdir`), since the `rio` namespace is only one level deep—there are no subdirectories under `/mnt/wsys/`.

⟨filsyswalk() *if dotdot* 130e⟩≡ (129)

```
if(strcmp(x->req.wname[i], "..") == 0){
    type = QTDIR;
    path = Qdir;
    dir = &dirtab[0];
    ⟨filsyswalk() when in dotdot, if Qwsysdir adjust path 210d⟩
    id = 0;
    ⟨filsyswalk() accept label and code 130a⟩
}
```

Uses `Qdir` 122d and `dirtab` 123b.

Why does walking `..` reach `rio` at all, rather than being resolved by the kernel? Because the directory tree under `/mnt/wsys` is entirely synthetic—`rio`, not the kernel, invents the per-window directories and their contents. The kernel's mount driver knows only the mount *point*; it cannot know that the parent of `/mnt/wsys/2/cons` is the window's directory, so it forwards every `walk` element, `..` included, as a 9P `Twalk` for the server to resolve. (The kernel does shortcut a `..` that would climb back out of the mount, but within the served tree the server owns the answer.)

8.5.4 Error management

`filsyswalk()`[129] guards against several error cases, gathered here: walking an already-open fid, a missing or non-directory path element, and an over-long name.

⟨`filsyswalk()` *sanity check if opened* 131a⟩≡ (129)
```
  if(f->open)
      return filsysrespond(fs, x, &fc, "walk of open file");
```
Uses `filsysrespond()` 124.

⟨`filsyswalk()` *sanity check i and err* 131b⟩≡ (129)
```
  if(i==0 && err==nil)
      err = Eexist;
```
Uses `Eexist` 280b.

⟨`filsyswalk()` *sanity check err and nwqid* 131c⟩≡ (129)
```
  if(err!=nil || fc.nwqid < x->req.nwname){
      if(nf){
          if(nf->w)
              sendp(winclosechan, nf->w);
          nf->open = false;
          nf->busy = false;
      }
  }
```
Uses `winclosechan` 108c.

⟨`filsyswalk()` *sanity check current qid is a directory* 131d⟩≡ (129)
```
  if(!(qid.type & QTDIR)){
      err = Enotdir;
      break;
  }
```
Uses `Enotdir` 280c.

⟨`filsyswalk()` *sanity check path elements* 131e⟩≡ (130a)
```
  if(i == MAXWELEM){
      err = "name too long";
      break;
  }
```

8.6 `open`

`open` illustrates the two-layer pattern clearly. The `filsysopen()`[132a] fast path only validates permissions—it checks the requested mode against the `dirtab`[123b] entry's permission bits, rejecting `OEXEC` and `ORCLOSE` since `rio` does not support executing or remove-on-close for its virtual files. The actual per-device open logic (e.g., setting `Window.mouseopen`[51f] or `Window.ctlopen`[152c]) runs in the `xfidopen()`[133a] worker thread, which has access to the window state.

8.6.1 filsysopen()

⟨*function* filsysopen 132a⟩≡ (346c)
```
static
Xfid*
filsysopen(Filsys *fs, Xfid *x, Fid *f)
{
    ⟨filsysopen() locals 132b⟩

    ⟨filsysopen() sanity check mode 132c⟩
    sendp(x->c, xfidopen);
    return nil;
⟨filsysopen() deny label 132f⟩
}
```
Uses xfidopen() 133a.

⟨filsysopen() *locals* 132b⟩≡ (132a) 132e ▷
```
int m;
```

⟨filsysopen() *sanity check mode* 132c⟩≡ (132a)
```
⟨filsysopen() sanitize mode 132d⟩
/* can't execute or remove anything */
if(x->req.mode==OEXEC || (x->req.mode & ORCLOSE))
    goto Deny;

switch(x->req.mode){
case OREAD:
    m = 0400;
    break;
case OWRITE:
    m = 0200;
    break;
case ORDWR:
    m = 0600;
    break;
default:
    goto Deny;
}
if(((f->dir->perm & ~(DMDIR|DMAPPEND)) & m) != m)
    goto Deny;
```

⟨filsysopen() *sanitize mode* 132d⟩≡ (132c)
```
/* can't truncate anything, so just disregard */
x->req.mode &= ~(OTRUNC|OCEXEC);
```

⟨filsysopen() *locals* 132e⟩+≡ (132a) ◁ 132b
```
Fcall fc;
```

⟨filsysopen() *deny label* 132f⟩≡ (132a)
```
Deny:
    return filsysrespond(fs, x, &fc, Eperm);
```
Uses Eperm 280a and filsysrespond() 124.

8.6.2 xfidopen()

The per-file open logic runs in the worker, `xfidopen()`[133a].

⟨*function* `xfidopen` 133a⟩≡ (347)
```
void
xfidopen(Xfid *x)
{
    Fcall fc;
    Window *w = x->f->w;

    ⟨xfidxxx() respond error if window was deleted 133b⟩
    switch(FILE(x->f->qid)){
    ⟨xfidopen() cases 152d⟩
    }

    x->f->open = true;
    x->f->mode = x->req.mode;

    fc.qid = x->f->qid;
    fc.iounit = messagesize-IOHDRSZ;
    filsysrespond(x->fs, x, &fc, nil);
}
```
Uses `FILE` 122c, `filsysrespond()` 124, and `messagesize` 76a.

I will present the `xfidopen()` switch cases one by one in the chapters devoted to each virtual device. Most files need no special open logic, but some do—for instance, opening `Qmouse` tells `rio` that this window is a graphical application (and set `Window.mouseopen`[51f]).

Every worker first checks whether the window was deleted out from under it (the `Window.deleted`[107a] flag and the two-phase teardown are covered in the window manager, Section 7.7); if so it replies `Edeleted`[281b] and stops.

⟨`xfidxxx()` *respond error if window was deleted* 133b⟩≡ (137b 136b 133a)
```
if(w->deleted){
    filsysrespond(x->fs, x, &fc, Edeleted);
    return;
}
```
Uses `Edeleted` 281b and `filsysrespond()` 124.

8.7 clunk (and close)

In 9P, releasing a file handle is called clunk—an English word meaning a dull thud, as if dropping something. The mapping between `close()` and `Tclunk` is not quite 1-to-1. For example, if a process calls `dup()` to duplicate a file descriptor, the kernel now has two fds pointing to the same underlying channel. Closing the first fd does *not* generate a `Tclunk`—the kernel only sends `Tclunk` to the server when the last reference is dropped. So several `close()` calls may produce zero or one `Tclunk`. The Plan 9 designers used a distinct name to keep the protocol layer separate from the system call layer.

8.7.1 filsysclunk()

`filsysclunk()`[133c] is the `filsysproc()`[75] half of clunk: it frees the fid, delegating to a worker only when the file was actually open.

⟨*function* `filsysclunk` 133c⟩≡ (346c)
```
static
Xfid*
```

```
filsysclunk(Filsys *fs, Xfid *x, Fid *f)
{
    Fcall fc;

    if(f->open){
        f->busy = false;
        f->open = false;
        sendp(x->c, xfidclose);
        return nil;
    }
    // else
    if(f->w)
        sendp(winclosechan, f->w);
    f->busy = false;
    f->open = false;
    return filsysrespond(fs, x, &fc, nil);
}
```

Uses `filsysrespond()` 124, `winclosechan` 108c, and `xfidclose()` 134.

8.7.2 `xfidclose()`

`xfidclose()`[134] is the worker half: it runs any per-file close logic, then drops the window reference with `wclose()`[109b].

⟨*function* `xfidclose` 134⟩≡ (347)

```
void
xfidclose(Xfid *x)
{
    Fcall fc;
    Window *w = x->f->w;
    ⟨xfidclose() other locals 232e⟩

    switch(FILE(x->f->qid)){
    ⟨xfidclose() cases 153a⟩
    }
    wclose(w);
    filsysrespond(x->fs, x, &fc, nil);
}
```

Uses `FILE` 122c, `filsysrespond()` 124, and `wclose()` 109b.

As with `xfidopen()`[133a], the per-file `xfidclose()` cases are presented gradually in the device chapters.

8.8 `read`

Read is where the two-layer split pays off the most. Directory reads (listing the files under `/mnt/wsys/`) can be handled entirely in the `filsysproc()`[75] context by iterating over `dirtab`[123b]—no window state is needed. But reading a device file like `/dev/mouse` or `/dev/cons` may have to wait until data is available, so it is dispatched to an `xfidread()`[136b] worker thread. This deserves a pause, because Plan 9 threads are cooperatively scheduled: a thread that makes a blocking syscall freezes its whole proc (exactly why the `fileserver` proc is kept separate because it reads the pipe). The worker gets away with waiting because it does not wait on a syscall—it waits on a channel with `recv()`. The genuinely blocking `read()` lives elsewhere, up in the `IO-proc-keyboard` reading `/dev/cons` (or `IO-proc-mouse` reading `/dev/mouse`), each isolated in its own proc for just this reason; the input it collects travels by channel through `main-proc`'s keyboard or mouse thread and the target window before reaching the worker. A channel `recv()` is a cooperative yield point, so the libthread scheduler just runs another thread of `main-proc` meanwhile; the proc keeps going.

8.8.1 filsysread()

`filsysread()`[135a] is that fast path: a directory read is served right here, anything else is handed to a worker.

⟨*function* `filsysread` 135a⟩≡ (346c)
```
static
Xfid*
filsysread(Filsys *fs, Xfid *x, Fid *f)
{
    int o, e;
    uint clock;
    byte *b;
    int n;
    Fcall fc;
    <filsysread() other locals 135c>

    if(!(f->qid.type & QTDIR)){
        sendp(x->c, xfidread);
        return nil;
    }
    // else, a directory, fast path

    o = x->req.offset;
    e = x->req.offset + x->req.count;
    clock = getclock();
    b = malloc(messagesize-IOHDRSZ); /* avoid memset of emalloc */
    <filsysread() sanity check b 135b>

    n = 0;
    switch(FILE(f->qid)){
    <filsysread() cases 136a>
    }

    fc.data = (char*)b;
    fc.count = n;
    filsysrespond(fs, x, &fc, nil);
    free(b);
    return x;
}
```
Uses `FILE` 122c, `filsysrespond()` 124, `getclock()` 139a, `messagesize` 76a, and `xfidread()` 136b.

⟨`filsysread()` *sanity check b* 135b⟩≡ (135a)
```
if(b == nil)
    return filsysrespond(fs, x, &fc, "out of memory");
```
Uses `filsysrespond()` 124.

The directory case is handled by `filsysread()` itself, shown in the next subsection; the device files (`/dev/cons`, `/dev/mouse`, . . .) are read in the worker `xfidread()`[136b], further below.

8.8.2 Reading a directory

To read a directory, the server iterates over `dirtab`[123b] and calls `dostat()`[138b] on each entry, which serializes a `Dir` structure into the 9P wire format via `convD2M()` (see the LIBCORE book [Pad16a]). The `o` and `e` variables implement offset-based pagination: the server formats all entries but only copies those whose cumulative offset falls within the requested range. This is needed because 9P directory reads must be deterministic and restartable from any byte offset.

⟨`filsysread()` *other locals* 135c⟩≡ (135a) 210f ▷
```
Dirtab *d;
```

⟨filsysread() *cases* 136a⟩≡ (135a) 211a ▷

```
case Qdir:
case Qwsysdir:
    d = dirtab;
    d++; /* first entry is '.' */
    for(i=0; d->name != nil && i<e; i+=len){
        len = dostat(fs, WIN(x->f->qid), d, b+n, x->req.count - n, clock);
        if(len <= BIT16SZ)
            break;
        if(i >= o)
            n += len;
        d++;
    }
    break;
```

Uses Qdir 122d, Qwsysdir 210c, WIN 122b, dirtab 123b, and dostat() 138b.

8.8.3 xfidread()

xfidread()[136b] is the worker that serves reads of the device files, waiting until data is available. As just noted, that wait will be a a recv() on a channel, not a blocking syscall, so it yields to the rest of main-proc rather than freezing it.

⟨*function* xfidread 136b⟩≡ (347)

```
void
xfidread(Xfid *x)
{
    uint qid;
    int off, cnt;
    Fcall fc;
    Fcall *req = &x->req;
    Window *w = x->f->w;
    ⟨xfidread() other locals 143b⟩

    ⟨xfidxxx() respond error if window was deleted 133b⟩
    qid = FILE(x->f->qid);
    off = req->offset;
    cnt = req->count;

    switch(qid){
    ⟨xfidread() cases 143c⟩
    default:
        fprint(STDERR, "unknown qid %d in read\n", qid);
        sprint(buf, "unknown qid in read");
        filsysrespond(x->fs, x, &fc, buf);
        break;
        }
}
```

Uses FILE 122c and filsysrespond() 124.

As with xfidopen()[133a], I will present the xfidread() cases one by one in the chapters devoted to each virtual device file. Each case has very different behavior: Qcons returns buffered text, Qmouse blocks until a mouse event is available, Qwinname returns a short string, and so on.

8.9 write

write is the simplest 9P operation: since there is no way to write to a directory (the kernel rejects that before it reaches the server), filsyswrite()[137a] can unconditionally dispatch to the xfidwrite()[137b] worker. All the

interesting logic—converting bytes to runes for `Qcons`, parsing mouse warp coordinates for `Qmouse`, interpreting control messages for `Qconsctl`—lives in the per-device cases of `xfidwrite()`.

8.9.1 filsyswrite()

⟨*function* `filsyswrite` 137a⟩≡ (346c)

```
static
Xfid*
filsyswrite(Filsys*, Xfid *x, Fid*)
{
    sendp(x->c, xfidwrite);
    return nil;
}
```

Uses `xfidwrite()` 137b.

8.9.2 xfidwrite()

⟨*function* `xfidwrite` 137b⟩≡ (347)

```
void
xfidwrite(Xfid *x)
{
    Fcall fc;
    Fcall *req = &x->req;
    Window *w = x->f->w;
    uint qid;
    int off, cnt;
    char buf[256];
    ⟨xfidwrite() other locals 144a⟩

    ⟨xfidxxx() respond error if window was deleted 133b⟩
    qid = FILE(x->f->qid);
    cnt = req->count;
    off = req->offset;
    req->data[cnt] = '\0';

    switch(qid){
    ⟨xfidwrite() cases 144b⟩
    default:
        fprint(STDERR, buf, "unknown qid %d in write\n", qid);
        sprint(buf, "unknown qid in write");
        filsysrespond(x->fs, x, &fc, buf);
        return;
    }

    fc.count = cnt;
    filsysrespond(x->fs, x, &fc, nil);
}
```

Uses `FILE` 122c and `filsysrespond()` 124.

8.10 stat

The `stat` operation returns metadata about a file—name, permissions, size, timestamps—similar to Unix's `stat()` system call. In `rio`, the metadata is synthesized on the fly from the `dirtab`[123b] entry and the window identifier, since none of these virtual files exist on disk.

8.10.1 filsysstat()

⟨*function* filsysstat 138a⟩≡ (346c)

```
static
Xfid*
filsysstat(Filsys *fs, Xfid *x, Fid *f)
{
	Fcall fc;

	fc.stat = emalloc(messagesize-IOHDRSZ);
	fc.nstat = dostat(fs, WIN(x->f->qid), f->dir, fc.stat, messagesize-IOHDRSZ, getclock());
	x = filsysrespond(fs, x, &fc, nil);
	free(fc.stat);
	return x;
}
```

Uses WIN 122b, dostat() 138b, filsysrespond() 124, getclock() 139a, and messagesize 76a.

We have already called dostat()[138b] twice—in the Qdir case of filsysread()[135a] to format each directory entry, and just now in filsysstat()[138a]. Here at last is the definition they share.

⟨*function* dostat 138b⟩≡ (346c)

```
static
int
dostat(Filsys *fs, int id, Dirtab *dir, uchar *buf, int nbuf, uint clock)
{
	Dir d;

	d.qid.path = QID(id, dir->qid);
	⟨dostat() adjust vers for snarf 233c⟩
	else
		d.qid.vers = 0;
	d.qid.type = dir->type;
	d.mode = dir->perm;
	d.length = 0; /* would be nice to do better */
	d.name = dir->name;
	d.uid = fs->user;
	d.gid = fs->user;
	d.muid = fs->user;
	d.atime = clock;
	d.mtime = clock;

	return convD2M(&d, buf, nbuf);
}
```

Uses QID 122a.

Note there is no xfidstat() worker: stat never blocks (it only reads dirtab[123b] metadata), so filsysstat() answers directly in the filsysproc()[75] context.

The directory listings above stamp each entry with a modification time. That time comes from /dev/time, opened once into clockfd[138c] and read by getclock()[139a].

⟨*global* clockfd 138c⟩≡ (346c)

```
fdt clockfd;
```

⟨filsysinit() *set clockfd* 138d⟩≡ (61c)

```
clockfd = open("/dev/time", OREAD|OCEXEC);
```

Uses clockfd 138c.

⟨*function* getclock 139a⟩≡ (346c)

```
/// filsysstat | filsysread -> <>
static
uint
getclock(void)
{
    char buf[32];

    seek(clockfd, 0, 0);
    read(clockfd, buf, sizeof buf);
    return atoi(buf);
}
```

Uses clockfd 138c.

8.11 Forbidden operations

Some 9P operations make no sense for `rio`'s virtual filesystem. A client cannot `create` new files—the set of device files (`cons`, `mouse`, etc.) is fixed by `rio`, not extensible by applications. Likewise, `remove` is forbidden because these virtual files are not real filesystem entries that can be deleted. And `wstat` (change metadata) is rejected because the permissions and ownership are determined by `rio` itself. All three simply respond with `Eperm`.

⟨fcall *other methods* 139b⟩+≡ (56b) ◁76f 267a▷

```
[Tcreate]  = filsyscreate,
[Tremove]  = filsysremove,
[Twstat]   = filsyswstat,
```

Uses filsyscreate() 139c, filsysremove() 139d, and filsyswstat() 139e.

⟨*function* filsyscreate 139c⟩≡ (346c)

```
static
Xfid*
filsyscreate(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, Eperm);
}
```

Uses Eperm 280a and filsysrespond() 124.

⟨*function* filsysremove 139d⟩≡ (346c)

```
static
Xfid*
filsysremove(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, Eperm);
}
```

Uses Eperm 280a and filsysrespond() 124.

⟨*function* filsyswstat 139e⟩≡ (346c)

```
static
Xfid*
filsyswstat(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;
```

```
      return filsysrespond(fs, x, &fc, Eperm);
  }
```

Uses Eperm 280a and filsysrespond() 124.

Chapter 9

Virtual Devices

The previous chapters built the file-server machinery—`Filsys`[53a], the `Fid`[53e] table, the `Xfid`[55b] workers, and the 9P operations that drive them (e.g., `filsysread()`[135a], `xfidread()`[136b]). What we never pinned down is which files those operations actually serve.

This chapter presents the virtual device files that `rio` serves under `/mnt/wsys/`. Because `/mnt/wsys` is bound before `/dev`, accesses to `/dev/mouse`, `/dev/cons`, etc. from client applications are intercepted by `rio`. For each device, the implementation involves two sides: the *producer* side where the `winctl()`[71b] thread offers data when it has events to report—events it receives in turn from the IO procs reading the real kernel `/dev/mouse` and `/dev/cons` (Figure 2.11)—and the *consumer* side where the `xfidread()` worker thread waits until data is available, then formats it as a textual response for the client.

9.1 Device virtualization across windowing systems

Every major windowing system has to solve the same problem—"get the mouse and keyboard to the right application"—and, `rio` aside, they nearly all solve it the *same* way: the application links a dedicated client library that speaks a custom binary protocol to a central server. The differences are mostly cosmetic. Under X Window the library is `Xlib` or `xcb` and events arrive as messages like `KeyPress` and `ButtonPress`; Wayland uses `libwayland-client` with a leaner protocol; macOS programs link `AppKit` and talk to a `WindowServer`; Win32 programs use `user32.dll` and receive `WM_MOUSEMOVE`, `WM_KEYDOWN`, and the like. Same shape, different spellings.

`rio` does it with no protocol and no client library at all. The application just opens `/dev/mouse` or `/dev/cons` and reads textual data; the file interface is indistinguishable from reading the real kernel devices of the same name. The same program that reads `/dev/cons` inside a window can run outside `rio` and read the real kernel `/dev/cons` with no code change, which is the concrete meaning of `rio`'s "transparent windowing system" [Pik88].

9.2 /mnt/wsys/mouse

We can now see the `Qxxx`[122d] and `dirtab`[123b] entries for each virtual device file, along with their `xfidread()`[136b] and `xfidwrite()`[137b] implementations. I start with the mouse because its protocol is simpler than the console's: the mouse only needs to pass a single `Mouse` value at a time, whereas the console must deal with byte/rune conversions and partial characters as we will see soon.

⟨`Qxxx` *other cases* 141a⟩≡ (122d) 146 ▷

```
  Qmouse,
```

⟨*dirtab array elements* 141b⟩≡ (123b) 147a ▷

```
  { "mouse",  QTFILE, Qmouse,  0600 },
```

Uses `Qmouse` 141a.

9.2.1 Reading `/mnt/wsys/mouse`: file server side

When a client reads `/mnt/wsys/mouse`, the `xfidread()`[136b] worker never touches the mouse itself: it waits for the window's `winctl()`[71b] thread to hand it an event. The two *rendezvous* through a *channel of channels*:the `Window.mouseread`[142a] channel set up below carries a `Mousereadmesg`[142c], which in turn holds the channel on which the `Mouse` value finally arrives.

⟨`Window` *other fields* 142a⟩+≡ (49) ◁114b 147b▷

```
// chan<chan<Mouse> > (listener = xfidread(Qmouse), sender = winctl)
Channel  *mouseread; /* chan(Mousereadmesg) */
```

⟨`wmk()` *channels creation* 142b⟩≡ (94c) 147c▷

```
w->mouseread =  chancreate(sizeof(Mousereadmesg), 0);
```

⟨*struct* `Mousereadmesg` 142c⟩≡ (333b)

```
struct Mousereadmesg
{
    // chan<Mouse> (listener = xfidread(Qmouse), sender = winctl)
    Channel *cm;  /* chan(Mouse) */
};
```

Why this channel of channel pattern? Consider a concrete scenario: the user moves the mouse rapidly, generating positions (100,50), (120,55), (140,60), (160,65). The `winctl()` thread sees all four events, relayed to it by `mousethread()`[66a], which in turn received them from the `IO-proc-mouse` reading the real `/dev/mouse` (Figure 2.11); but the client application only calls `read("/dev/mouse")` now. With a naive direct channel, `winctl()` would have queued all four positions; the client would get the stale (100,50) from the head. With the channel-of-channels pattern, no `Mouse` value is produced until the client asks for one:

```
xfidread (worker)                    winctl (window thread)
+-----------------+                  +-------------------+
| client's read   |                  | has seen 4 events,|
| is pending, so  |  Mousereadmesg   | pos now (160,65); |
| the worker is   |  {cm: replyCh}   | offers to send    |
| RECEIVING on    |<-----------------+ only now that it  |
|  w->mouseread   |                  | has data          |
|                 |                  |                   |
| recv(cm, &ms)   |  Mouse{160,65}   | send(mrm.cm,      |
|                 |<-----------------+  &currentMouse)   |
+-----------------+                  +-------------------+
```

The rendezvous runs the opposite way from what you might guess: the *worker* blocks *receiving* on `Window.mousere` while `winctl()` offers to *send* on it only when it actually has a fresh event (otherwise that arm of its `alt()` is disabled). Because the channel is unbuffered, the exchange fires only when both hold—a reader is waiting *and* `winctl()` has data—and `winctl()` snapshots the freshest mouse state at that instant, handing over the *reply channel* `mrm.cm` and then the `Mouse` itself. The client gets (160,65)—the position that matters—never a stale queued value. Freshness is the main payoff, but the rendezvous earns its keep twice more: `winctl()` never snapshots or queues a mouse state nobody is waiting for, and because the worker reaches the data by *waiting* on a channel, that wait is a single `alt()` that can also watch a flush channel and cancel the read cleanly (Section 13.7.1).

We can now see the code. The `Qmouse` case of `xfidread()` waits with an `alt()` on two channels at once, and the enum below names its slots: `MRdata` for an arriving `Mousereadmesg` on `Window.mouseread`, and `MRflush` for a flush (flush handling is deferred to Section 13.7.1). On the `MRdata` branch it performs the second half of the channel-of-channels handshake—`recv(mrm.cm, &ms)`—and formats the `Mouse` as the text the client reads.

NMR is not a real case: it is simply the *count* of slots, used to place the CHANEND terminator that marks the end of the alts array passed to alt().

⟨*enum* _anon_ (rio/xfid.c)4 143a⟩≡ (347)
```
  enum { MRdata, MRflush, NMR };
```

⟨xfidread() *other locals* 143b⟩≡ (136b) 148c ▷
```
  Alt alts[NCR+1];
  Mousereadmesg mrm;
  Mouse ms;
  int n, c;
```
Uses NCR-95 148b.

⟨xfidread() *cases* 143c⟩≡ (136b) 148d ▷
```
  case Qmouse:
      ⟨xfidxxx() set flushtag 267e⟩

      alts[MRdata].c = w->mouseread;
      alts[MRdata].v = &mrm;
      alts[MRdata].op = CHANRCV;
      ⟨xfidread() when Qmouse, set alts for flush 268a⟩
      alts[NMR].op = CHANEND;

      switch(alt(alts)){
      case MRdata:
          break;
      ⟨xfidread() when Qmouse, switch alt flush case 268b⟩
      }
      /* received data */
      ⟨xfidxxx() unset flushtag 267f⟩
      ⟨xfidread() when Qmouse, if flushing 268c⟩

      qlock(&x->active);
      recv(mrm.cm, &ms);
      c = 'm';
      ⟨xfidread() when Qmouse, adjust c for resize message if resized 168a⟩
      n = sprint(buf, "%c%11d %11d %11d %11d ", c, ms.xy.x, ms.xy.y, ms.buttons, ms.msec);
      w->resized = false;

      fc.data = buf;
      fc.count = min(n, cnt);
      filsysrespond(x->fs, x, &fc, nil);
      qunlock(&x->active);
      break;
```
Uses MRdata-96 143a, NMR-98 143a, Qmouse 141a, filsysrespond() 124, and min() 284a.

Two subtleties are worth flagging. The qlock(&x->active) around the recv()/filsysrespond()[124] pair serializes this delivery against a concurrent *flush* of the same request, so a reply is never written for a read the client has already abandoned. And the very same channel doubles as a resize notification: c starts as 'm' but is switched to announce a resize when Window.resized[114b] is set—which is why a mouse read can also wake a client that was really waiting to learn its window changed size. A *flush* here is the 9P Tflush request: the client telling rio to abandon a still-pending read, typically because the program was interrupted while blocked in read("/dev/mouse"). Such a read can block indefinitely, so xfidread() must be ready to be torn down this way—hence the alt() on a flush channel and the x->active lock; the full mechanism is in Section 13.7.1.

9.2.2 Writing /mnt/wsys/mouse

Writing to `/dev/mouse` lets a program warp[1] the cursor programmatically, but only under strict conditions: the window must have focus (`w==input`) and be visible on screen. The write data must start with "m" followed by the x and y coordinates. The actual cursor movement is routed through `wsendctlmesg()`[91c] with `Movemouse`[144c], so it goes through the `winctl()`[71b] thread and respects the `sweeping`[89d] guard—you cannot warp the mouse while the user is in the middle of a window management operation.

⟨`xfidwrite()` *other locals* 144a⟩≡ (137b) 150a▷

```
  char *p;
  Point pt;
```

⟨`xfidwrite()` *cases* 144b⟩≡ (137b) 150c▷

```
  case Qmouse:
      if(w!=input || Dx(w->screenr)<=0)
          break;
      if(req->data[0] != 'm'){
          filsysrespond(x->fs, x, &fc, Ebadmouse);
          return;
      }
      p = nil;
      pt.x = strtoul(req->data+1, &p, 0);
      if(p == nil){
          filsysrespond(x->fs, x, &fc, Eshort);
          return;
      }
      pt.y = strtoul(p, nil, 0);
      if(w==input && wpointto(mouse->xy)==w)
          wsendctlmesg(w, Movemouse, Rpt(pt, pt), nil);
      break;
```

Uses `Ebadmouse` 281l, `Eshort` 281f, `Movemouse` 144c, `Qmouse` 141a, `filsysrespond()` 124, `input` 51e, `mouse` 48c, `wpointto()` 69, and `wsendctlmesg()` 91c.

⟨`Wctlmesgkind` *cases* 144c⟩+≡ (91a) ◁109e 160d▷

```
  Movemouse,
```

⟨`wctlmesg()` *cases* 144d⟩+≡ (92a) ◁114a 160e▷

```
  case Movemouse:
      if(sweeping || !ptinrect(r.min, w->i->r))
          break;
      wmovemouse(w, r.min);
      break;
```

Uses `Movemouse` 144c, `sweeping` 89d, and `wmovemouse()` 116c.

9.2.3 Trace of a mouse click

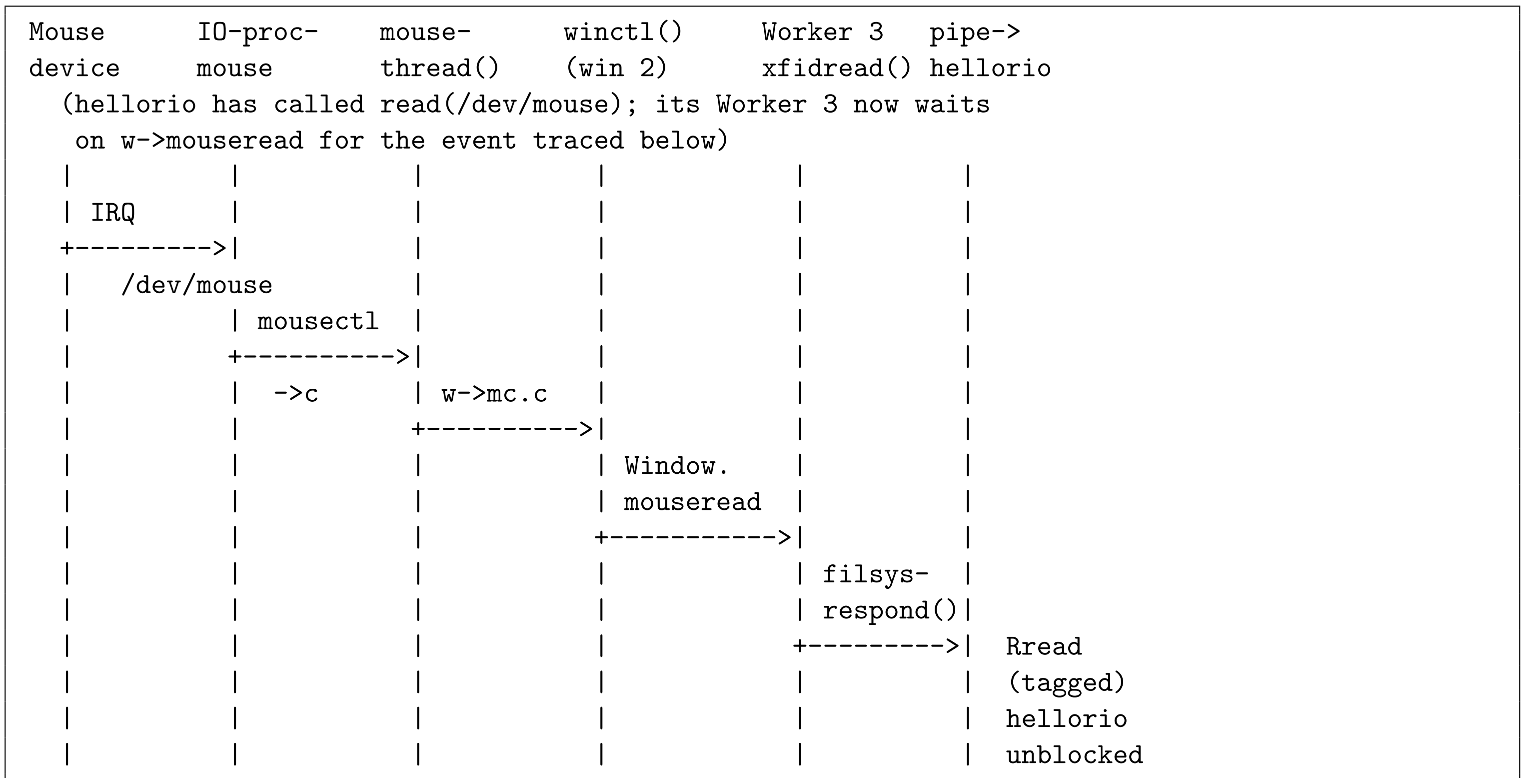
We have seen the mouse device's code in isolation; now let us watch a single click travel through it, end to end.

When you click or move the mouse, the information is first transmitted from the mouse device to the kernel, at the bottom left in Figure 2.11. This information must then find a way to `rio` and possibly to the window with the focus, for instance, `hellorio` at the bottom right in Figure 2.11. The goal of this section is to describe the full trace of a mouse click in the window of `hellorio`. End to end the path is short: the kernel wakes `IO-proc-mouse`, which reads the event and passes it to `mousethread()`[66a]; that thread routes it to the focused window's `winctl()`[71b] thread, which hands it to the window's worker thread (`Worker 3`) waiting in `xfidread()`[136b]; the worker then writes the 9P reply straight into the pipe to unblock `hellorio`. The trace below walks each of these hops on Figure 2.11, so it is worth spotting two pieces of that figure first: the worker

[1] To warp the cursor is to move the pointer to a new screen position from software, rather than by physically moving the mouse.

threads of `main-proc` (one per open file, the bridge between a window thread and a client's blocked `read`), and the arrows that carry the event left-to-right across `main-proc`.

Here is the data flow of a mouse click from the hardware to `hellorio`, through the different threads of `rio`:



```
Mouse      IO-proc-    mouse-      winctl()     Worker 3   pipe->
device     mouse       thread()    (win 2)      xfidread() hellorio
  (hellorio has called read(/dev/mouse); its Worker 3 now waits
   on w->mouseread for the event traced below)
  |          |            |            |             |          |
  | IRQ      |            |            |             |          |
  +--------->|            |            |             |          |
  |   /dev/mouse          |            |             |          |
  |          | mousectl   |            |             |          |
  |          +----------->|            |             |          |
  |          |  ->c       | w->mc.c    |             |          |
  |          |            +----------->|             |          |
  |          |            |            | Window.     |          |
  |          |            |            | mouseread   |          |
  |          |            |            +------------>|          |
  |          |            |            |             | filsys-  |
  |          |            |            |             | respond()|
  |          |            |            |             +--------->|  Rread
  |          |            |            |             |          |  (tagged)
  |          |            |            |             |          |  hellorio
  |          |            |            |             |          |  unblocked
```

Mouse initialization

Before you click on the mouse, the window application `hellorio` must first become ready to receive such a mouse event by calling `initmouse()`. The trace resulting from such a call is explained below:

1. When `hellorio` starts, it opens `/dev/mouse` (via `initmouse()`) and reads `/dev/mouse` (via its own IO proc, see Section 2.3).

2. The kernel resolves the access to `/dev/mouse` to the file `/mnt/wsys/mouse`, because of a `bind()` call in the parent process of `hellorio` (see `filsysmount()`[99c]).

3. The kernel relays a read on `/mnt/wsys/mouse` to a 9P request written in the pipe to `fileserver` proc, because of a call to `mount()` in the parent process of `hellorio` (see `filsysmount()` again). At this point, `hellorio` is blocked (actually its mouse IO proc) waiting for his `read()` system call to return.

4. The kernel scheduler eventually schedules the `fileserver` proc, which will read from the pipe, decode the 9P request, and start to process the request.

Worker initialization

The question now is what should the `fileserver` proc do with the read request on `/mnt/wsys/mouse` from `hellorio`? It can not block until you move the mouse. Indeed, another process may also need to communicate with the `fileserver` proc in the mean time, for instance, a process in a terminal window may want to display some text by writing in its `/mnt/wsys/cons` file. To manage independent file requests, the `fileserver` proc offloads work to independent worker threads. As we saw before, *Each worker thread represents an opened file of* `rio`*'s filesystem.*

In the case of `hellorio` reading `/mnt/wsys/mouse`, the worker thread is called `Worker 3` in Figure 2.11. This thread will then wait for mouse events by listening on a special channel.Once `Worker 3` receives a mouse event, it writes the 9P answer into the pipe *itself*, by calling `filsysrespond()`[124]—it does not hand the event back to the `fileserver` proc. The reply is a single atomic message stamped with the request's tag, and it can be written even while `filsysproc()`[75] is busy reading the next request. At this point, the kernel will unblock the `hellorio` process (actually its mouse IO proc), which can modify the GUI or do nothing depending on the mouse event returned from `/mnt/wsys/mouse`.

Mouse event

The remaining question is how does the mouse event arrive to the `Worker 3` thread? Now that `hellorio` is ready to receive mouse events, and `Worker 3` has been created, I can finally explain the trace resulting from a mouse click in the window of `hellorio`, as shown in the diagram above:

1. A click on the mouse device, at the bottom left in Figure 2.11, generates an hardware interrupt, which is managed by the kernel (see the KERNEL book [Pad14]).
2. The kernel looks for a process waiting on `/dev/mouse` and so unblocks the `IO-proc-mouse` proc of `rio`, at the left in Figure 2.11.
3. This proc contains a single thread that was reading on `/dev/mouse`. This thread parses the mouse event contained in the data read from `/dev/mouse`, and sends it to `mousectl->c`.
4. The message is received by the `mousethread()`[66a], which was listening on `mousectl->c`. The mouse thread then looks for the window with the focus and relays the message on `input->mc.c`.
5. The `Window 2` thread, at the middle of Figure 2.11, which was listening on `w->mc.c` (as well as other channels via `alt()`) receives the message on `w->mc.c`. It needs then to transmit this message to `Worker 3`. When the `fileserver` proc created the `Worker 3` thread, it also created a channel to communicate with this thread. This channel is stored in the `Window.mouseread`[142a] field of the `Window`[49] structured allocated for `hellorio`. Thus, `Window 2` can communicate with `Worker 3` by writing the mouse event through this channel.

At this point, the `Worker 3` thread, which was listening on `Window.mouseread`, receives the mouse event and replies to the client itself: it calls `filsysrespond()`[124], which writes an `Rread` message— stamped with the request's tag so the kernel routes it to the right client—directly into the pipe. It never goes back through the `fileserver` proc. This concludes the trace of mouse click.

Note that if you click with the mouse outside any window, the mouse event does not go to any window thread. Instead, the event stops at the `mousethread()`, which contains code to process the event to possibly move, resize, hide, or create a new window. Indeed, as mentioned in Section 7.5.7, it is `mousethread()` that creates windows and all their associated entities (processes, namespaces, layers, threads, and `Windows`).

9.3 /mnt/wsys/cons

The console device is the most complex virtual device because it bridges two mismatched interfaces: the client reads and writes a stream of *bytes*, while the `winctl()`[71b] thread works in *runes* and may have nothing to give when the client asks. Reconciling the byte/rune boundary and that waiting is what makes the code further below intricate.

```
⟨Qxxx other cases 146⟩+≡                                    (122d) ◁141a 152a▷
  Qcons,
```

⟨*dirtab array elements* 147a⟩+≡ (123b) ◁141b 152b▷
```
  { "cons",  QTFILE, Qcons,  0600 },
```
Uses `Qcons` 146.

9.3.1 Reading `/mnt/wsys/cons`: file server side

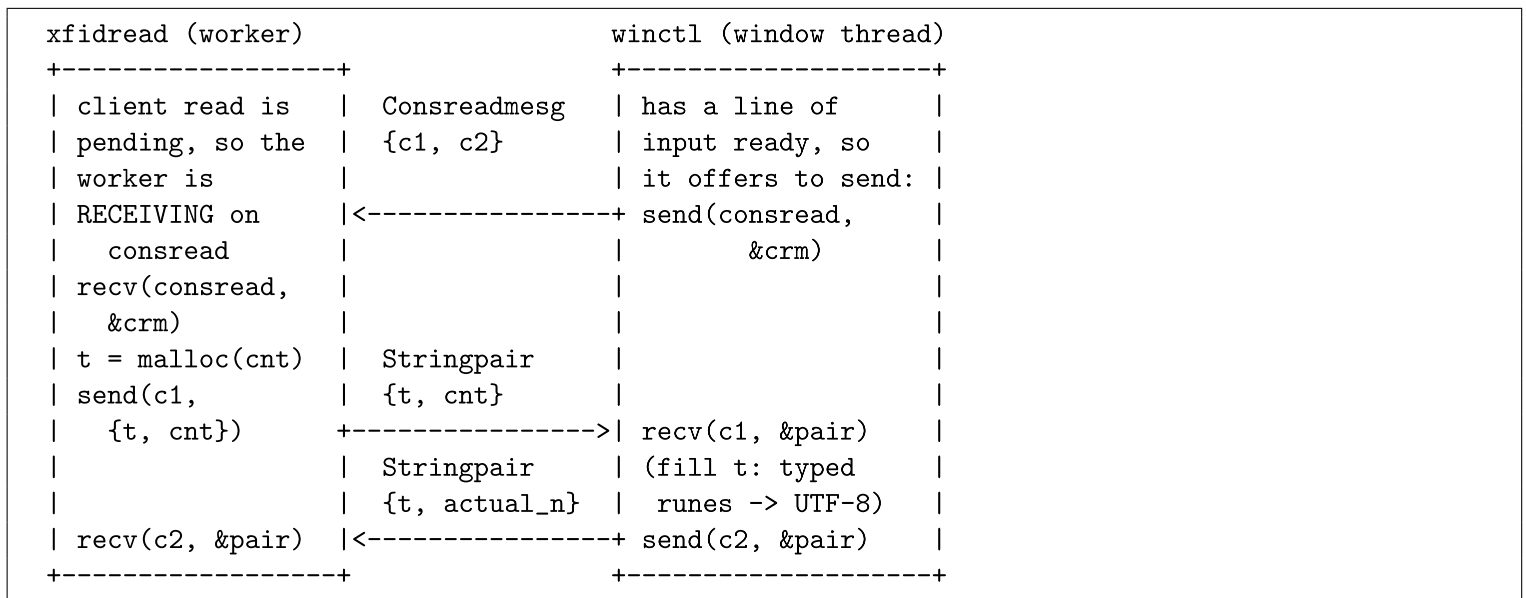
Reading `/mnt/wsys/cons` has the same shape as the mouse: the `xfidread()`[136b] worker rendezvous with `winctl()`[71b] instead of reading anything itself. The console handshake is heavier, though—it needs two inner channels rather than one. And the reason for those channels of channel is not the same reason than in the mouse's case: there the extra hop let us drop stale positions and deliver only the latest, whereas typed characters must never be dropped—they queue in the window's input buffer. Here the two channels instead negotiate a variable-length byte transfer (the reader announces how many bytes it can take; `winctl()` returns at most that many). You might ask why not put `c1` and `c2` straight into the `Window`[49] as two fields and skip the wrapping message. They could be: like `mrm.cm`, they are created once per window (by `winctl()`) and reused. What the channel-of-channels really buys is the *rendezvous*—the unbuffered `Window.consread`[147b] is what synchronises the worker and `winctl()` (the worker waits to receive on it; `winctl()` sends only when a line is ready), and it is the single channel the worker's `alt()` watches alongside a flush. Handing the reply channels over *in* that rendezvous keeps them bound to it rather than loose in the `Window`; it is not that a fresh pair is needed per read.

⟨`Window` *other fields* 147b⟩+≡ (49) ◁142a 149a▷
```
  // chan<Consreadmesg> (listener = xfidread(Qcons), sender = winctl)
  Channel  *consread; /* chan(Consreadmesg) */
```

⟨`wmk()` *channels creation* 147c⟩+≡ (94c) ◁142b 149b▷
```
  w->consread =  chancreate(sizeof(Consreadmesg), 0);
```

⟨*struct* `Consreadmesg` 147d⟩≡ (333b)
```
  struct Consreadmesg
  {
      // chan<ref<array<Rune>> (listener = winctl, sender = xfidread(Qcons))
      Channel *c1;  /* chan(tuple(char*, int) == Stringpair) */
      // chan<ref<array<Rune>> (listener = xdidread(Qcons), sender = winctl)
      Channel *c2;  /* chan(tuple(char*, int) == Stringpair) */
  };
```

As I said before, unlike the mouse, the console uses two channels inside the `Consreadmesg`: `c1` to pass the read buffer from the worker to `winctl()`, and `c2` to pass the filled data back. The two-channel protocol is needed because the worker must first tell `winctl()` how many bytes it can accept (via `c1`), and then `winctl()` fills that buffer and returns it (via `c2`). The exchange is easiest to follow as a picture:



The mouse needs only one reply channel because it always returns a fixed-size `Mouse` struct. The console, however, must negotiate a variable-length byte transfer, and the byte/rune boundary does not always align—a multibyte UTF-8 rune may span two reads, requiring *partial rune* bookkeeping as we will see soon.

⟨*struct* Stringpair 148a⟩≡ (333b)
```
struct Stringpair /* rune and nrune or byte and nbyte */
{
    void *s;
    int  ns;
};
```

As with the mouse, we can now see the code. The enum below names the `alt()` slots for the `Qcons` read: `CRdata` for a `Consreadmesg` arriving on `Window.consread`, and `CRflush` for a flush. On the data branch the worker runs the two-channel handshake just diagrammed—it allocates the byte buffer, sends it on `c1`, and receives the filled bytes back on `c2`—before replying to the client. (As with the mouse, the flush case is deferred to Section 13.7.1.)

⟨*enum* _anon_ (rio/xfid.c) *3* 148b⟩≡ (347)
```
enum { CRdata, CRflush, NCR };
```

⟨xfidread() *other locals* 148c⟩+≡ (136b) ◁143b 170a▷
```
Consreadmesg crm;
Channel *c1, *c2; /* chan (tuple(char*, int)) */
char *t;
Stringpair pair;
```

⟨xfidread() *cases* 148d⟩+≡ (136b) ◁143c 153f▷
```
case Qcons:
    ⟨xfidxxx() set flushtag 267e⟩

    alts[CRdata].c = w->consread;
    alts[CRdata].v = &crm;
    alts[CRdata].op = CHANRCV;
    ⟨xfidread() when Qcons, set alts for flush 268d⟩
    alts[NCR].op = CHANEND;

    switch(alt(alts)){
    case CRdata:
        break;
    ⟨xfidread() when Qcons, switch alt flush case 268e⟩
    }
    /* received data */
    ⟨xfidxxx() unset flushtag 267f⟩

    c1 = crm.c1;
    c2 = crm.c2;
    t = malloc(cnt+UTFmax+1); /* room to unpack partial rune plus */
    pair.s = t;
    pair.ns = cnt;
    send(c1, &pair);

    ⟨xfidread() when Qcons, if flushing 268f⟩

    qlock(&x->active);
    recv(c2, &pair);
    fc.data = pair.s;
    fc.count = pair.ns;
    filsysrespond(x->fs, x, &fc, nil);
    free(t);
    qunlock(&x->active);
```

```
    break;
```
Uses `CRdata-93` 148b, `NCR-95` 148b, `Qcons` 146, and `filsysrespond()` 124.

A couple of subtleties. The buffer is `malloc`'d with `UTFmax` extra bytes so `winctl()` has room to prepend a *partial rune* left over from a previous read (the leftover is remembered in `Fid.nrpart`[151a]); a multibyte UTF-8 character must never be split across the byte count the client asked for. And as in the mouse case, `qlock(&x->active)` brackets the `recv`/`filsysrespond()` so the reply cannot race a flush of the same request.

9.3.2 Writing `/mnt/wsys/cons`: file server side

Console writing follows the same channel handshake as reading, but the data flows in the opposite direction: the client's bytes go from `xfidwrite()`[137b] to `winctl()`[71b]. A critical subtlety is that the data must be converted from UTF-8 bytes to complete runes *before* being sent, because `winctl()` works in rune space—it cannot display half a multibyte character:

```
xfidwrite (worker)                    winctl (window thread)
+------------------+                  +--------------------+
| client writes    |                  |                    |
| "héllo" (bytes)  |                  |                    |
|                  |                  |                    |
| cvttorunes()     |                  |                    |
| bytes -> runes   |                  |                    |
| (store partial   |                  |                    |
|  in f->rpart)    |                  |                    |
|                  |  Conswritemesg   |                    |
| recv(conswrite,  |  {cw}            |                    |
|   &cwm)          |<-----------------+ send(conswrite,    |
|                  |  w->conswrite    |        &cwm)       |
|                  |                  |                    |
|                  |  Stringpair      |                    |
| send(cwm.cw,     |  {runes, nr}     |                    |
|  {r, nr})        +----------------->| recv(cw, &pair)    |
|                  |                  | (display runes)    |
+------------------+                  +--------------------+
```

Note the reversed initiative compared to reading: for writing, it is `winctl()` that sends the `Conswritemesg`[149c] first (signaling it is ready to accept output), and then the worker sends the rune data through the `cw` channel.

⟨`Window` *other fields* 149a⟩+≡ (49) ◁147b 152c▷
```
  // chan<Conswritemesg> (listener = xfidwrite(Qcons), sender = winctl)
  Channel  *conswrite; /* chan(Conswritemesg) */
```

⟨`wmk()` *channels creation* 149b⟩+≡ (94c) ◁147c 213b▷
```
  w->conswrite = chancreate(sizeof(Conswritemesg), 0);
```

⟨*struct* `Conswritemesg` 149c⟩≡ (333b)
```
  struct Conswritemesg
  {
      // chan<ref<array<Rune>> (listener = winctl, sender = xfidwrite(Qcons))
      Channel *cw;  /* chan(Stringpair) */
  };
```

⟨*enum* `_anon_ (rio/xfid.c)`*2* 149d⟩≡ (347)
```
  enum { CWdata, CWflush, NCW };
```

```
⟨xfidwrite() other locals 150a⟩+≡                                   (137b) ◁144a 150b▷
  Rune *r;
  int nr; // nb runes
  int nb; // nb bytes
```

```
⟨xfidwrite() other locals 150b⟩+≡                                   (137b) ◁150a 151d▷
  Alt alts[NCW+1];
  Conswritemesg cwm;
  Stringpair pair;
```

Uses NCW-92 149d.

```
⟨xfidwrite() cases 150c⟩+≡                                          (137b) ◁144b 153b▷
  case Qcons:

      ⟨xfidwrite() when Qcons, look for previous partial rune bytes 151c⟩
      r = runemalloc(cnt);
      cvttorunes(req->data, cnt-UTFmax, r, &nb, &nr, nil);
      ⟨xfidwrite() when Qcons, look if more full runes 151e⟩
      ⟨xfidwrite() when Qcons, store remaining partial rune bytes 151b⟩

      ⟨xfidxxx() set flushtag 267e⟩

      alts[CWdata].c = w->conswrite;
      alts[CWdata].v = &cwm;
      alts[CWdata].op = CHANRCV;
      ⟨xfidwrite() when Qcons, set alts for flush 268g⟩
      alts[NCW].op = CHANEND;

      switch(alt(alts)){
      case CWdata:
          break;
      ⟨xfidwrite() when Qcons, switch alt flush case 268h⟩
      }

      /* received data */
      ⟨xfidxxx() unset flushtag 267f⟩
      ⟨xfidwrite() when Qcons, if flushing 269a⟩

      qlock(&x->active);
      pair.s = r;
      pair.ns = nr;
      send(cwm.cw, &pair);
      fc.count = req->count;
      filsysrespond(x->fs, x, &fc, nil);
      qunlock(&x->active);
      return;
```

Uses CWdata-90 149d, NCW-92 149d, Qcons 146, cvttorunes() 285b, filsysrespond() 124, and runemalloc 284d.

9.3.3 Bytes versus runes and partial runes

The file interface deals in bytes, but the display works in runes. A UTF-8 character can be up to `UTFmax` (4) bytes, and a single `write()` system call may split a multibyte character across two calls. To handle this, each `Fid`[53e] keeps a small buffer `Fid.rpart`[151a] of leftover bytes from the previous write. On the next write, these bytes are prepended to the new data before rune conversion. The conversion pipeline works in two passes: `cvttorunes()` converts the bulk of the data (stopping `UTFmax` bytes before the end), then a `fullrune()` loop

picks up any remaining complete runes at the tail. Whatever bytes are left over—an incomplete rune at the end—are stashed back into `rpart` for next time.

⟨`Fid` *other fields* 151a⟩+≡ (53e) ◁123c
```
uchar rpart[UTFmax];
int  nrpart;
```

⟨`xfidwrite()` *when Qcons, store remaining partial rune bytes* 151b⟩≡ (150c)
```
// assert(cnt-nb < UTFMAX);
if(nb < cnt){
    memmove(x->f->rpart, req->data + nb, cnt-nb);
    x->f->nrpart = cnt-nb;
}
```

⟨`xfidwrite()` *when Qcons, look for previous partial rune bytes* 151c⟩≡ (150c)
```
nr = x->f->nrpart;
if(nr > 0){
    memmove(req->data + nr, req->data, cnt); /* there's room: see malloc in filsysproc */
    memmove(req->data, x->f->rpart, nr);
    cnt += nr;
    x->f->nrpart = 0;
}
```

⟨`xfidwrite()` *other locals* 151d⟩+≡ (137b) ◁150b
```
int c;
```

⟨`xfidwrite()` *when Qcons, look if more full runes* 151e⟩≡ (150c)
```
/* approach end of buffer */
while(fullrune(req->data + nb, cnt-nb)){
    c = nb;
    nb += chartorune(&r[nr], req->data + c);
    if(r[nr])
        nr++;
}
```

9.3.4 Trace of a key press

The trace of a key press in the window of `hellorio` is similar to the trace of a mouse click. The trace starts also at the bottom left in Figure 2.11, but this time with the keyboard device. The key event is transmitted from the kernel to the `IO-proc-keyboard` through `/dev/cons`, then relayed to the keyboard thread through `keyboardctl->c`, then to the window thread `Window 2` through a `Window.ck`[72b] channel, then to the worker thread `Worker 4` through a `Window.consread`[147b] channel, then to the `fileserver` proc, and finally back to the kernel and `hellorio` through a pipe. `hellorio` finally reads the key event through his `/mnt/wsys/cons` file that was opened (in raw access mode) at startup via a call to `initkeyboard()`.

For textual windows, such as the shell process in Figure 2.11 (represented by the `Window 1` thread), the trace of a key press is more complicated. Indeed, `rio` provides advanced line-editing features that complicates the flow of data to `/mnt/wsys/cons`. Moreover, this file is opened both in read and write modes, resulting from two worker threads in Figure 2.11: `Worker 1` and `Worker 2`. This is why the trace of a key press in a textual window will be explained later in Chapter 11.

The trace of a key press follows a similar path, but through the keyboard side:

```
 Keyboard   IO-proc      Keyboard   Window 2     Worker 4   Fileserver
 device     keyboard     thread     thread       thread     proc
   |          |            |          |            |          |
   | IRQ      |            |          |            |          |
   +--------->|            |          |            |          |
   |   /dev/cons           |          |            |          |
   |          | kbdctl     |          |            |          |
   |          +----------->|          |            |          |
   |          |  ->c       | Window.ck |           |          |
   |          |            +---------->|           |          |
   |          |            |          | Window.    |          |
   |          |            |          | consread   |          |
   |          |            |          +----------->|          |
   |          |            |          |            | channel  |
   |          |            |          |            +--------->|
   |          |            |          |            |          |
   |          |            |          |            |   9P Rread
   |          |            |          |            |   (pipe)
   |          |            |          |            |          |
   |          |            |          |            |    hellorio
   |          |            |          |            |    unblocked
```

For a textual window (e.g., the shell), the path again is more complex because the window thread also handles line editing, echoing, and scrolling before the data reaches the worker—this will be explained again in Chapter 11.

9.4 /mnt/wsys/consctl

The `consctl` device is much simpler than `/mnt/wsys/mouse` or `/mnt/wsys/cons`: it has no channel-of-channels protocol, no reading side, and no data flowing back to the client. A client writes control strings like `"rawon"` or `"holdon"` to change how the console processes input—we will see those cases later in the textual windows chapter.

⟨`Qxxx` *other cases* 152a⟩+≡ (122d) ◁146 153c▷
```
  Qconsctl,
```

⟨*dirtab array elements* 152b⟩+≡ (123b) ◁147a 153d▷
```
  { "consctl", QTFILE, Qconsctl, 0200 },
```
Uses `Qconsctl` 152a.

Like `Window.mouseopen`[51f], `Window.ctlopen`[152c] enforces exclusive access—only one client at a time may open `/dev/consctl`. But unlike `mouseopen`, which triggers the switch from textual to graphical mode, `ctlopen` has no such side effect; it merely gates access to the control commands.

⟨`Window` *other fields* 152c⟩+≡ (49) ◁149a 181c▷
```
  bool ctlopen;
```

⟨`xfidopen()` *cases* 152d⟩≡ (133a) 159▷
```
  case Qconsctl:
      if(w->ctlopen){
          filsysrespond(x->fs, x, &fc, Einuse);
          return;
      }
      w->ctlopen = true;
      break;
```
Uses `Einuse` 281a, `Qconsctl` 152a, and `filsysrespond()` 124.

⟨xfidclose() *cases* 153a⟩≡ (134) 153e▷
```
  case Qconsctl:
      ⟨xfidclose() Qconsctl case, if rawing 160c⟩
      ⟨xfidclose() Qconsctl case, if holding 271h⟩
      w->ctlopen = false;
      break;
```
Uses Qconsctl 152a.

The body of the Qconsctl write case—the raw-mode and hold-mode control messages a client can send—is filled in later, with the rest of the console-control handling.

⟨xfidwrite() *cases* 153b⟩+≡ (137b) ◁150c 153g▷
```
  case Qconsctl:
      ⟨xfidwrite() Qconsctl case 160b⟩
      // else
      filsysrespond(x->fs, x, &fc, "unknown control message");
      return;
```
Uses Qconsctl 152a and filsysrespond() 124.

9.5 /mnt/wsys/cursor

The cursor device lets a client application set a custom cursor shape. Writing the binary cursor data (hotspot offset plus two 2x16 bitmaps, see for example boxcursor[82a]) stores it in Window.cursor[86d] and points Window.cursorp[86] at it; writing less than the required 2*4+2*2*16 bytes clears the custom cursor, reverting to the default arrow. Closing the file also clears it.

⟨Qxxx *other cases* 153c⟩+≡ (122d) ◁152a 158a▷
```
  Qcursor,
```

⟨*dirtab array elements* 153d⟩+≡ (123b) ◁152b 158b▷
```
  { "cursor",  QTFILE, Qcursor, 0600 },
```
Uses Qcursor 153c.

⟨xfidclose() *cases* 153e⟩+≡ (134) ◁153a 168b▷
```
  case Qcursor:
      w->cursorp = nil;
      wsetcursor(w, false);
      break;
```
Uses Qcursor 153c and wsetcursor() 87.

⟨xfidread() *cases* 153f⟩+≡ (136b) ◁148d 158c▷
```
  case Qcursor:
      filsysrespond(x->fs, x, &fc, "cursor read not implemented");
      break;
```
Uses Qcursor 153c and filsysrespond() 124.

⟨xfidwrite() *cases* 153g⟩+≡ (137b) ◁153b 209a▷
```
  case Qcursor:
      if(cnt < 2*4+2*2*16)
          w->cursorp = nil;
      else{
          w->cursor.offset.x = BGLONG(req->data+0*4);
          w->cursor.offset.y = BGLONG(req->data+1*4);
          memmove(w->cursor.clr, req->data+2*4, 2*2*16);
          w->cursorp = &w->cursor;
      }
      wsetcursor(w, !sweeping);
      break;
```
Uses Qcursor 153c, sweeping 89d, and wsetcursor() 87.

BGLONG() here unpacks a big-endian 32-bit integer from the raw byte stream; it is one of the byte-order extraction macros provided by LIBCORE book [Pad16a].

9.6 /dev/draw/ and /mnt/wsys/winname

Drawing is the one device rio does not demultiplex. Where /dev/cons and /dev/mouse are virtual files served by the worker threads we just saw, each window process opens the kernel's own /dev/draw directly (as /dev/draw/<n>), so pixels never pass through rio's file server. That makes drawing more efficient but moves the complexity into per-window graphics setup. The one small file rio does still serve here is /mnt/wsys/winname: it holds the *name* of the window's image, which a window process reads to attach to its own drawing surface on the shared screen.

9.6.1 Trace of a drawing operation

The final trace illustrating Figure 2.11 is the trace of a drawing operation by hellorio. As I said in Section 2.2.3, the ancestor of rio, 8-1/2 [Pik91], was offering a virtual /mnt/wsys/draw device file; the trace of a drawing operation was then similar to the trace of a mouse click or a key press in Section 9.2.3 and Section 9.3.4. However, for efficiency reasons, with rio the window applications are connected directly to the draw device, as shown in Figure 2.11 with the hellorio process connected to draw via /dev/draw/3/. This simplifies the trace of a drawing operation, but complicates the initialization of graphics for the window process. Indeed, as I explained partly already in different sections, rio and draw needs to cooperate to enable window applications to draw on the screen:

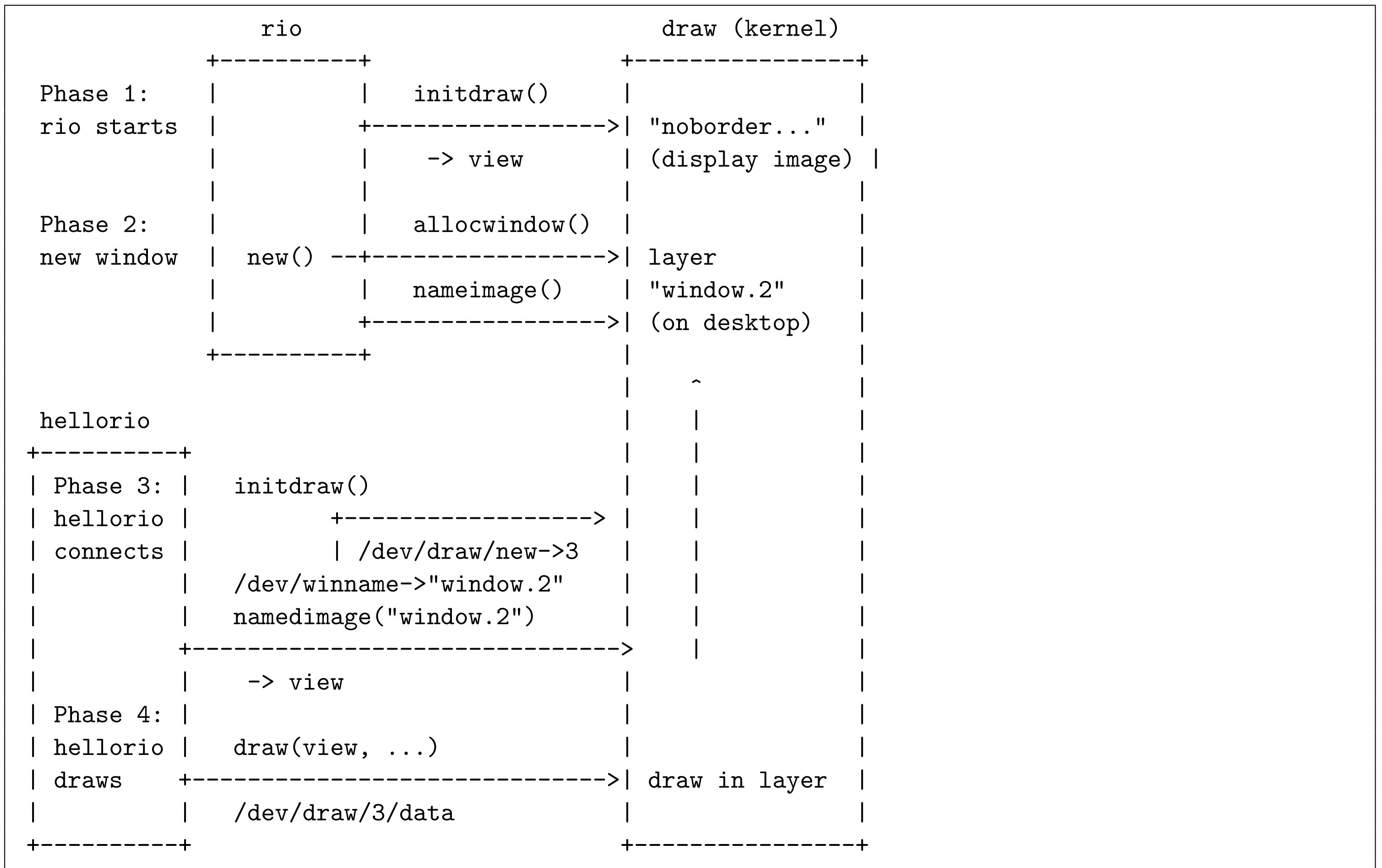
- rio is using draw's layers to implement overlapping windows (see Section 2.1.10)
- draw is using a clients/server architecture to support the multiple clients of rio (see Section 2.2.3)
- draw supports the naming and sharing of layers across multiple clients (see Section 2.2.3)
- hellorio uses in his code the global view, which is a reference to a layer, to draw things in its window (see Section 2.3)
- rio is using /mnt/wsys/winname to communicate the name of a layer to the window (see Section 7.5.5)

The goal of this section is to put together all those pieces of information to better understand how a window application draws on the screen.

The trace unfolds chronologically in four phases:

1. **rio starts up**: rio calls initdraw() and obtains a reference to the display image (stored in view).
2. **A new window is created**: rio allocates a layer on view and gives it a public name (e.g., "window.2").
3. **hellorio connects**: hellorio calls initdraw(), reads /mnt/wsys/winname to discover the layer name, and grabs a reference to it (also stored in its own view).
4. **hellorio draws**: hellorio calls draw(view, ...), which goes directly to the draw device—bypassing rio's fileserver entirely.

The traces below are simplified; certain explanations about the draw protocol are contained in the GRAPHICS book [Pad16c]. The following diagram illustrates those four phases. Note that the draw client number (/dev/draw/3/) is unrelated to the window identifier in the layer name (window.2)—it simply depends on the order in which processes connect to draw:



Phase 1: `rio` starts up

The goal of this phase is for `rio` to obtain a reference to the display image, stored in the global `view`. This is the image on which `rio` will later create layers for each window. Note that `rio` itself is a graphical application; it calls `initdraw()` just like `hellorio` does. The difference is that `rio` connects to the real `/dev/` devices, not virtual devices under `/mnt/wsys/`[2].

The trace of `initdraw()` in `rio` is as follows:

1. When `rio` starts, it calls `initdraw()`, which opens `/dev/draw/new` to open a new connexion to the display server (see the Graphics book [Pad16c]).

2. The kernel resolves the access to `/dev/draw/new` to a method of the `draw` device in the kernel, because of a call to `bind("#i", "/dev/")` in the parent process of `rio` (see Section 1.4). Since `rio` is the first application to open `/dev/draw/new`, the kernel returns a file handler to `/dev/draw/1/ctl`. This is why in Figure 2.11, the `rio` process is connected to the `/dev/draw/1` directory.

3. `initdraw()` reads the information about the screen contained in `/dev/draw/1/ctl` and sets the global `display`.

4. `initdraw()` calls `gengetwindow()`, which opens and reads `/dev/winname`.

5. The kernel resolves the access to `/dev/winname` to a method of the `draw` device. The content returned for this file is `"noborder.screen.1"`, the name given by `draw` to the framebuffer image.

[2] Unless `rio` is run recursively inside one of its own windows, in which case it also sees virtual devices from the parent `rio` (see Section 13.2).

6. `initdraw()` then calls `namedimage()` to grab a reference to the image named `"noborder.screen.1"`. Remember that API calls to `draw` such as `namedimage()` are translated by `libdraw` in written commands in `/dev/draw/1/data` (see the GRAPHICS book [Pad16c]).

7. The kernel resolves the access to `/dev/draw/1/data` to a method of the `draw` device. This method parses the `namedimage()` command, creates a new image identifier, find the image corresponding to `"noborder.screen.1"` (the framebuffer), internally links the new image identifier to this image, and finally returns the new image identifier.

8. `initdraw()` reads information about this new image identifier in `/dev/draw/1/ctl` and stores the information in the global `view`.

Once `rio` called `initdraw()`, it can draw things on the screen by passing the global `view` as an argument to drawing functions. Moreover, after `initmouse()`, `rio` enables also the user to use the mouse.

Phase 2: a new window is created

The goal of this phase is for `rio` to create a layer for the new window and make it accessible by giving it a public name.

The trace leading to the creation of a new layer is as follows:

1. To create a new window, you must right-click on the mouse while dragging the mouse to specify a rectangle on the screen (see Section 2.2.2). All those actions are handled by code in the mouse thread of `rio`. This code also uses the global `view` as an argument to functions in `draw.h`, for instance, to draw on the screen the white rectangle specified with the mouse, as well as its red border (see Figure 2.7).

2. Once you release the right-click, `rio` creates internally many new entities: a process, a namespace, a `Window`[49] (stored in the global array `windows`[51a]), and a thread (see Section 7.5.7). `rio` also calls `allocwindow()` from `window.h` to create a new layer. It passes `view` as the base-layer argument to `allocwindow()`, and the rectangle specified with the mouse for the dimension of the layer. The call to `allocwindow()` is translated by `libdraw` in a command written in `/dev/draw/1/data`. The `view` argument is reduced to its image identifier.

3. The kernel parses the `allocwindow()` command written in `/dev/draw/1/data` and allocates a new layer with a new image identifier. It uses the framebuffer as the base layer for this layer, since the image identifier written in `/dev/draw/1/data` was previously linked by the kernel to the framebuffer. The kernel also records in a list the set of layers associated with the framebuffer, and put this new layer at the top of the list. If a program draws in this layer, it will draw directly in the framebuffer since the layer is at the top. Finally, the kernel returns the new image identifier to `rio`.

4. `rio` stores the new image identifier in the `Window.i`[50c] field of the `Window` structure allocated for the new window. `rio` creates also a new name for this layer, e.g. `"window.2"`. and stores it in the `Window.name`[50a] field. `rio` then calls `nameimage()` with `Window.i` and `Window.name` as arguments, to name and share the layer. This call is translated again by `libdraw` in a command written in `/dev/draw/1/data`.

5. The kernel parses the `nameimage()` command and adds in a global hash table in the kernel an association between `"window.2"` and the layer created previously.

Remember that when you create a new window, this window is always first a textual window managed by the terminal emulator. This window is initially connected to a shell process, as shown at the bottom right in Figure 2.11. Because the terminal emulator is an integral part of `rio`, it has access to the globals of `rio` such as `view` or `windows`. Thus, the terminal emulator can draw in the layer stored previously in `Window.i`. It

can translate text output from the shell process through `/mnt/wsys/cons` to calls to drawing functions taking `Window.i` as an argument (e.g., `string()`).

At some point, you can launch `hellorio` from this terminal window by typing the name of the command in the shell.

Phase 3: `hellorio` connects to the layer

The goal of this phase is for `hellorio` to discover and grab a reference to the layer that `rio` created in Phase 2. The key difference with Phase 1 is that `/dev/winname` now resolves through `rio`'s fileserver (because of the namespace set up in Phase 2), not directly through the `draw` device.

Here is the trace of `initdraw()` in `hellorio`:

1. When `hellorio` starts, it also calls `initdraw()` (like `rio` did), which opens `/dev/draw/new`. The kernel returns a file handler to a new `/dev/draw/` directory (e.g., `/dev/draw/3/ctl`). This is why in Figure 2.11, the `hellorio` process is connected to `/dev/draw/3/`.

2. `initdraw()` calls `gengetwindow()`, which opens and reads `/dev/winname`.

3. The kernel resolves this time the access to `/dev/winname` to `/mnt/wsys/winname`, and communicates the read request on this file to the `fileserver` proc. This is different from the trace of `initdraw()` in `rio`. Indeed, the namespace set in the parent process of `hellorio` by `rio` is different from the namespace of `rio` itself.

4. The `fileserver` proc allocates a new worker thread to serve `/mnt/wsys/winname` for `hellorio`. Note that this thread is not shown in Figure 2.11. Indeed, this worker thread exists only during the call to `initdraw()`. Before returning, `initdraw()` closes `/dev/winname`, and so `fileserver` releases the worker thread.

5. The worker thread looks for the name of the layer for `hellorio` in the `Window.name` field of the `Window`[49] associated with `hellorio`. This `Window` is stored in the global array `windows`[51a]. To find the appropriate `Window`, the worker thread compares the window identifier stored in `Window.id` and the window identifier in the 9P request (which was originally passed to `mount()` in the parent process of `hellorio`, see Section ??). Finally, the worker thread returns `"window.2"`, the content of `Window.name` for the window from where `hellorio` was launched.

6. `initdraw()` then calls `namedimage()` to grab a reference to the image named `"window.2"`. The kernel creates a new image identifier, find the image corresponding to `"window.2"` by looking in a global hash table (this image is the layer created by `rio` above), internally links the new image identifier to this image, and finally returns the new image identifier. `initdraw()` then stores the new image identifier in the global `view` of the `hellorio` process.

This concludes the graphics initialization for `hellorio`. At this point, `hellorio`'s `view` and `rio`'s `Window.i` both refer to the same layer in the kernel—exactly as shown in the diagram at the beginning of this section.

Phase 4: `hellorio` draws

This is the simplest phase: `hellorio` draws directly to `draw` in the kernel, with no involvement from `rio`'s fileserver.

Once `hellorio` called `initdraw()`, it can draw things in its window by passing the global `view` as an argument to drawing functions. This time, `view` corresponds to an image layer in the kernel, not the framebuffer. However, this layer is linked to the framebuffer, as well as the other layers created for the other windows. Here is the trace of the call to `draw()` in the `redraw()` function in `hellorio.c` (see Section 2.3.1):

1. When `hellorio` calls `draw()`, the call is translated by `libdraw` in a command written in `/dev/draw/3/data`. The `view` argument is reduced to its image identifier.

2. The kernel resolves the access to `/dev/draw/3/data` to a method of the `draw` device. This method parses the `draw()` command and fetches the image associated to the image identifier written in `/dev/draw/3/data`. This image is actually a layer, which is associated to the framebuffer.

3. The kernel then calls `memdraw()`, a function from `libmemlayer` that contains special code to handle image layers. `memdraw()` then goes through the list of layers associated to the framebuffer to check if the layer of `hellorio` is the top layer. If it is, `memdraw()` calls `memimagedraw()` from `libmemdraw` to draw pixels directly in the framebuffer. If it is not the top layer, `memdraw()` allocates a new off-screen image and calls `memimagedraw()` to draw in this image instead for all the rectangles overlapped by other layers.

Later, if you click on the window of `hellorio` to make it the top window, `rio` will call the `draw` function `topwindow()`. This command will be parsed by the kernel. The kernel will readjust the list of layers associated to the framebuffer, and copy back the pixels from the off-screen image associated to the layer of `hellorio` to the framebuffer.

9.6.2 `/mnt/wsys/winname`

Despite the complex collaboration between `rio` and `draw` described in Section 9.6.1—involving layers, `namedimage()`, and the four-phase initialization—the virtual `/dev/winname` device itself is trivially simple: it just returns the string in `Window.name`[50a]. All the complexity lives in how that name is established and used, not in serving it.

⟨`Qxxx` *other cases* 158a⟩+≡ (122d) ◁153c 169b▷
```
  Qwinname,
```

⟨*dirtab array elements* 158b⟩+≡ (123b) ◁153d 169c▷
```
  { "winname", QTFILE, Qwinname, 0400 },
```
Uses `Qwinname` 158a.

⟨`xfidread()` *cases* 158c⟩+≡ (136b) ◁153f 170b▷
```
  case Qwinname:
      n = strlen(w->name);
      ⟨xfidread() when Qwinname case, sanity check n 158d⟩
      t = estrdup(w->name);
      goto Text;
```
Uses `Qwinname` 158a.

⟨`xfidread()` *when Qwinname case, sanity check n* 158d⟩≡ (158c)
```
  if(n == 0){
      filsysrespond(x->fs, x, &fc, "window has no name");
      break;
  }
```
Uses `filsysrespond()` 124.

This concludes the generic virtual device code. The behavior of `/mnt/wsys/mouse` and `/mnt/wsys/cons` varies depending on whether the window is graphical or textual—the next two chapters cover those differences.

Chapter 10

Graphical Windows

A graphical window is one where the application draws directly using the `draw` API rather than using `/dev/cons` for text I/O. The key difference is `Window.mouseopen`[51f]: when the application opens `/dev/mouse`, `rio` stops intercepting middle-click and right-click for its own menus, forwards all mouse events to the application, and stops repainting the window's text content. Graphical applications may also request "raw" keyboard mode via `/dev/consctl`, receiving keystrokes immediately rather than after a newline.

10.1 Graphical window setup

This section covers the setup that transforms a window from textual to graphical mode. A graphical application typically calls three library functions at startup: `initdraw()` to connect to the `draw` device, `initmouse()` to open `/dev/mouse`, and optionally `initkeyboard()` to write `"rawon"` on `/dev/consctl`. Each of these triggers specific behavior in `rio`'s filesystem server. Throughout I name these files as the application sees them—`/dev/mouse`, `/dev/consctl`, and so on, the names that appear in the library code. Under `rio` those names resolve though, through the window's namespace, to `rio`'s own virtual `/mnt/wsys/mouse`, `/mnt/wsys/consctl`, etc.

10.1.1 `initdraw()`

The `initdraw()` call is handled entirely by the kernel's `draw` device, not by `rio`'s filesystem server—so there is no `rio` code to show here. However, `initdraw()` is a crucial step: it establishes the connection to `/dev/draw` and reads `/dev/winname` to discover the layer name (see Section 9.6.1).

10.1.2 `initmouse()`, mouse-open mode

Unlike `initdraw()`, the `initmouse()` call does trigger `rio` code: it opens `/mnt/wsys/mouse`, which arrives as an `xfidopen()`[133a] request with `Qmouse`. This is where `rio` switches the window into graphical mode by setting `mouseopen` to true.

⟨`xfidopen()` *cases* 159⟩+≡ (133a) ◁152d 212a▷

```
  case Qmouse:
      if(w->mouseopen){
          filsysrespond(x->fs, x, &fc, Einuse);
          return;
      }
      // else
      w->mouseopen = true;
      /*
       * Reshaped: there's a race if the appl. opens the
       * window, is resized, and then opens the mouse,
       * but that's rare.  The alternative is to generate
```

```
     * a resized event every time a new program starts
     * up in a window that has been resized since the
     * dawn of time.  We choose the lesser evil.
     */
    w->resized = false;
    break;
```
Uses `Einuse` 281a, `Qmouse` 141a, and `filsysrespond()` 124.

10.1.3 initkeyboard(), raw-access mode

The `Window.rawing`[160a] field tracks whether the window is in "raw" keyboard mode. When an application writes `"rawon"` to `/dev/consctl`, keystrokes are delivered immediately without waiting for a newline. Despite being declared `bool`, `rawing` is actually used as a reference count: multiple `"rawon"` writes increment it, and only when the matching `"rawoff"` brings it back to zero does `rio` switch modes. This is a dubious design—it is hard to imagine a legitimate use case for nested raw mode—but it means a stray extra `"rawon"` will not accidentally cancel raw mode on the first `"rawoff"`.

⟨`Window` *config fields* 160a⟩+≡ (49) ◁107a 270c▷
```
  bool rawing;
```

⟨`xfidwrite()` *Qconsctl case* 160b⟩≡ (153b) 271g▷
```
  if(strncmp(req->data, "rawon", 5)==0){
      ⟨xfidwrite() Qconsctl case, if rawon message and holding mode 271i⟩
      if(w->rawing++ == 0)
          wsendctlmesg(w, Rawon, ZR, nil);
      break;
  }
  if(strncmp(req->data, "rawoff", 6)==0 && w->rawing){
      if(--w->rawing == 0)
          wsendctlmesg(w, Rawoff, ZR, nil);
      break;
  }
```
Uses `Rawoff` 160d, `Rawon` 160d, and `wsendctlmesg()` 91c.

⟨`xfidclose()` *Qconsctl case, if rawing* 160c⟩≡ (153a)
```
  if(w->rawing){
      w->rawing = false;
      wsendctlmesg(w, Rawoff, ZR, nil);
  }
```
Uses `Rawoff` 160d and `wsendctlmesg()` 91c.

As with the mouse, a `"rawon"`/`"rawoff"` write becomes a `Wctlmesg`[91b] (`Rawon`[160d] or `Rawoff`[160d]) so the mode change is applied in the `winctl()`[71b] thread. `Rawon` has nothing left to do—`w->rawing` was already bumped in `xfidwrite()`[137b]—while `Rawoff` still has to reprocess any keys that were buffered raw, now that cooked mode resumes.

⟨`Wctlmesgkind` *cases* 160d⟩+≡ (91a) ◁144c 168c▷
```
  Rawon,
  Rawoff,
```

⟨`wctlmesg()` *cases* 160e⟩+≡ (92a) ◁144d 168d▷
```
  case Rawon:
      // already setup w->rawing in xfidwrite, nothing else todo
      break;
  case Rawoff:
      ⟨wctlmesg() break if window was deleted 107b⟩
      ⟨wctlmesg() When Rawoff, process raw keys in non rawing mode 165a⟩
      break;
```
Uses `Rawoff` 160d and `Rawon` 160d.

10.2 Mouse events

This section shows the *window-thread side* of mouse event delivery—the producer/consumer machinery that sits between `mousethread()`[66a] and the client's `read("/dev/mouse")`. The earlier *file-server side* showed how the `xfidread()`[136b] worker waits for an event; here we see how `winctl()`[71b] queues click events, decides when to offer data, and responds on the reply channel.

10.2.1 Mouse state queue

Each window keeps its own `Mouseinfo`[161b] struct to buffer mouse events destined for the client application. This is the producer's staging area: `winctl()`[71b] writes into it when it receives events from `mousethread()`[66a], and reads from it when the client is ready via the channel-of-channels protocol described in the previous chapter.

⟨`Window` *mouse fields* 161a⟩+≡ (49) ◁86d
```
  Mouseinfo mouse;
```

The mouse event delivery system uses a circular queue per window to buffer click/release events, but only the latest position for move events. This is because missing a click matters (the user expected an action), but missing intermediate moves does not (only the final position matters). The `Mouseinfo.counter`[161c]/`Mouseinfo.lastcounter` pair tracks whether there is a new event: if they differ, the client gets the latest state on the next `read()`.

⟨*struct* `Mouseinfo` 161b⟩≡ (333b)
```
  struct Mouseinfo
  {
      // queue of mouse clicks and releases
      Mousestate queue[16];

      // consumer
      int ri; /* read index into queue */
      // producer
      int wi; /* write index */

      bool qfull;/* filled the queue; no more recording until client comes back */
      ⟨Mouseinfo other fields 161c⟩
  };
```

⟨`Mouseinfo` *other fields* 161c⟩≡ (161b) 162a▷
```
  ulong counter; /* serial no. of last mouse event we received */
  ulong lastcounter;/* serial no. of last mouse event sent to client */
```

The shape of the buffer is easier to see when drawn. Here is a window whose client has already consumed four mouse events and is about to read a fifth, with two more enqueued behind it:

```
             ri=4                          wi=7
              v                             v
  queue[16]: +---+---+---+---+---+---+---+---+---+---+ ...
             | . | . | . | . |Mv |Lk |Mv | _ | _ | _ |
             +---+---+---+---+---+---+---+---+---+---+
              0   1   2   3   4   5   6   7
               (already read)  ^^^^^^^^^^^
                                to deliver
                                (Lk = left click, Mv = move)

   counter     = 423   <- bumped by winctl on every enqueue
   lastcounter = 419   <- snapshot client saw on previous read
   -> counter != lastcounter: there is something new
```

The 16-slot ring is tiny on purpose: if the client falls that far behind, `Mouseinfo.qfull` is set and further events are silently dropped until `ri == wi` again. Move events in particular are coalesced at the producer side—`winctl()` overwrites the latest position instead of pushing it—so the 16 slots effectively hold up to 16 distinct click/release transitions, which is plenty even for a user double-clicking frantically. This is the exact opposite of keyboard events, where dropping a rune would swallow user input; there `Window.nraw`[164a] is strictly append-and-consume (see the raw-keys queue later).

⟨`Mouseinfo` *other fields* 162a⟩+≡ (161b) ◁161c
```
int lastb; /* last button state we received */
```

⟨*struct* `Mousestate` 162b⟩≡ (333b)
```
struct Mousestate
{
    Mouse;
    ulong counter; /* serial no. of mouse event */
};
```

10.2.2 Reading `/mnt/wsys/mouse`: window thread side

This is the *window-thread side* of the mouse reading code. The *file-server side* in the previous chapter showed the `xfidread()`[136b] worker waiting to receive a `Mousereadmesg`[142c] and then the reply. Here we see the `winctl()`[71b] side: the producer that enqueues click events into `w->mouse.queue`, and the consumer that hands one over (or snapshots the current position) when a reader is waiting.

Producer

The producer is the `WMouse` arm of `winctl()`[71b]: when an event arrives from `mousethread()`[66a] and the client has the mouse open, `winctl()` records *button transitions* (clicks and releases) into the ring buffer, skipping pure moves—those are picked up later as the current position.

⟨`winctl()` *other locals* 162c⟩+≡ (71b) ◁74e 163b▷
```
Mousestate *mp;
int lastb = -1;
```

⟨`winctl()` *WMouse case if mouseopen* 162d⟩≡ (74c)
```
if(w->mouseopen) {
    w->mouse.counter++;

    /* queue click events */
    if(!w->mouse.qfull && lastb != w->mc.m.buttons) { /* add to ring */

        //insert_queue(w->mc, w->mouse.queue)
        mp = &w->mouse.queue[w->mouse.wi];
        if(++w->mouse.wi == nelem(w->mouse.queue))
            w->mouse.wi = 0;
        if(w->mouse.wi == w->mouse.ri)
            w->mouse.qfull = true;
        mp->Mouse = w->mc.m;
        mp->counter = w->mouse.counter;

        lastb = w->mc.m.buttons;
    }
}
```

Consumer

The consumer is the `WMouseread` arm. `winctl()`[71b] enables it—offers to send (`CHANSND`)—only when the mouse is open and a new event is pending (`counter != lastcounter`); the worker's matching receive then completes the rendezvous.

⟨`Wxxx` *cases* 163a⟩≡ (71a) 165b▷
```
  WMouseread,
```

⟨`winctl()` *other locals* 163b⟩+≡ (71b) ◁162c 163g▷
```
  Mousereadmesg mrm;
```

⟨`winctl()` *channels creation* 163c⟩≡ (71b) 165d▷
```
  mrm.cm = chancreate(sizeof(Mouse), 0);
```

⟨`winctl()` *Wctl case, free channels if wctlmesg is Excited* 163d⟩≡ (74g) 165e▷
```
  chanfree(mrm.cm);
```

⟨`winctl()` *alts setup* 163e⟩+≡ (71b) ◁74f 165f▷
```
  alts[WMouseread].c = w->mouseread;
  alts[WMouseread].v = &mrm;
  alts[WMouseread].op = CHANSND;
```
Uses `WMouseread-28` 163a.

⟨`winctl()` *alts adjustments* 163f⟩≡ (71b) 165g▷
```
  if(w->mouseopen && w->mouse.counter != w->mouse.lastcounter)
      alts[WMouseread].op = CHANSND;
  else
      alts[WMouseread].op = CHANNOP;
```
Uses `WMouseread-28` 163a.

And here at last is the heart of it: when the rendezvous fires, `winctl()` dequeues the oldest click—or, if the queue is empty, snapshots the current position—and sends it on the reply channel `mrm.cm`.

⟨`winctl()` *other locals* 163g⟩+≡ (71b) ◁163b 165c▷
```
  Mousestate m;
```

⟨`winctl()` *event loop cases* 163h⟩+≡ (71b) ◁74g 166a▷
```
  case WMouseread:
      /* send a queued event or, if the queue is empty, the current state */
      /* if the queue has filled, we discard all the events it contained. */
      /* the intent is to discard frantic clicking by the user during long latencies. */
      w->mouse.qfull = false;
      // if produced more than read
      if(w->mouse.wi != w->mouse.ri) {
          m = w->mouse.queue[w->mouse.ri];
          if(++w->mouse.ri == nelem(w->mouse.queue))
              w->mouse.ri = 0;
      } else
          m = (Mousestate){w->mc.m, w->mouse.counter};

      w->mouse.lastcounter = m.counter;
      // consumed and relay
      send(mrm.cm, &m.Mouse);
      continue;
```
Uses `WMouseread-28` 163a.

10.3 Keyboard events

Keyboard event delivery for graphical windows is simpler than mouse events: there is no circular queue with click/move distinction, no `counter/lastcounter` bookkeeping. Instead, raw keystrokes are simply appended to a growing array `Window.raw`[164a] and consumed one rune at a time. The complexity here is different—it lies in the interaction between raw mode and cooked (buffered) mode, and in the rune/byte conversion when the data reaches the client.

10.3.1 Raw keys queue

These fields are only relevant when `Window.rawing`[160a] is true, i.e., when a client has written `"rawon"` on `/mnt/wsys/consctl`. In that mode, `wkeyctl()`[73b] appends each keystroke to `Window.raw`[164a] instead of inserting it into the text buffer. The runes accumulate here until the client reads `/dev/cons`, at which point the `WCread` consumer in `winctl()`[71b] converts them to UTF-8 bytes and sends them out.

⟨`Window` *graphical window fields* 164a⟩+≡ (49) ◁51f

```
// growing_array<Rune> (size = Window.nraw)
Rune  *raw;
uint  nraw;
```

10.3.2 Reading `/mnt/wsys/cons`: raw mode

Reading `/mnt/wsys/cons` in raw mode has the same producer/consumer shape as the mouse, but over `Window.raw`[164a] the producer is `wkeyctl()`[73b] piling up keystrokes, and the consumer is `winctl()`[71b]'s `WCread` arm handing them to the `xfidread()`[136b] worker.

Producer

When raw mode is active, keystrokes bypass the normal insertion path and go directly to the `Window.raw`[164a] queue via `waddraw()`[164c]. The condition `w->mouseopen || w->q0 == w->nr` below distinguishes two cases: in a graphical window (`mouseopen`), all raw keys are queued unconditionally—the application is fully in charge of keyboard handling. In a textual window, only keys typed at the very end of the buffer (where `q0 == nr`) take the raw path, because the user might have moved the cursor backward to edit earlier text, which should go through the normal insertion/line-discipline path. This subtlety means a textual window with `rawing == true` can still support partial line editing: the user can click on earlier text, edit it with backspace, and only characters typed at the very end go through raw mode. In practice this rarely matters because most applications that request raw mode also open the mouse (becoming graphical windows), but the code handles both cases correctly.

⟨`wkeyctl()` *if rawing* 164b⟩≡ (73b)

```
if(w->rawing && (w->mouseopen || w->q0 == w->nr)){
    waddraw(w, &r, 1);
    return;
}
```

Uses `waddraw()` 164c.

⟨*function* `waddraw` 164c⟩≡ (345b)

```
void
waddraw(Window *w, Rune *r, int nr)
{
    w->raw = runerealloc(w->raw, w->nraw+nr);
    runemove(w->raw + w->nraw, r, nr);
    w->nraw += nr;
}
```

Uses `runemove` 285a and `runerealloc` 284e.

When the application sends `Rawoff`, any runes that were accumulated in `Window.raw` during raw mode are replayed through `wkeyctl()`[73b] in normal (line-buffered) mode. Since `Window.rawing`[160a] is now false, `wkeyctl()` will insert them into the text buffer and process special characters normally. The one-at-a-time `runemove()` shift is inefficient but the raw queue is typically short.

```
⟨wctlmesg() When Rawoff, process raw keys in non rawing mode 165a⟩≡                    (160e)
  while(w->nraw > 0){
      wkeyctl(w, w->raw[0]);
      --w->nraw;
      runemove(w->raw, w->raw+1, w->nraw);
  }
```

Uses `runemove` 285a and `wkeyctl()` 73b.

Consumer

The consumer mirrors the mouse's: `winctl()`[71b]'s `WCread` arm, offered only when there are raw runes waiting, hands them to the `xfidread()`[136b] worker through the two-channel `Consreadmesg`[147d] handshake.

```
⟨Wxxx cases 165b⟩+≡                                          (71a) ◁163a 195f▷
  WCread,
```

```
⟨winctl() other locals 165c⟩+≡                               (71b) ◁163g 165h▷
  Consreadmesg crm;
```

```
⟨winctl() channels creation 165d⟩+≡                          (71b) ◁163c 196a▷
  crm.c1 = chancreate(sizeof(Stringpair), 0);
  crm.c2 = chancreate(sizeof(Stringpair), 0);
```

```
⟨winctl() Wctl case, free channels if wctlmesg is Excited 165e⟩+≡     (74g) ◁163d 196b▷
  chanfree(crm.c1);
  chanfree(crm.c2);
```

```
⟨winctl() alts setup 165f⟩+≡                                 (71b) ◁163e 196c▷
  alts[WCread].c = w->consread;
  alts[WCread].v = &crm;
  alts[WCread].op = CHANSND;
```

Uses `WCread-29` 165b.

```
⟨winctl() alts adjustments 165g⟩+≡                           (71b) ◁163f 196d▷
  ⟨winctl() alts adjustments, if holding 270d⟩
  else if((w->rawing && w->nraw>0) || npart)
      alts[WCread].op = CHANSND;
  else{
      alts[WCread].op = CHANNOP;
      ⟨winctl() alts adjustments, revert to CHANSND if newline in queue 188a⟩
  }
```

Uses `WCread-29` 165b.

```
⟨winctl() other locals 165h⟩+≡                               (71b) ◁165c 166b▷
  Stringpair pair;
  char *t;
  int nb;
  int i, c, wid;
```

The WCread consumer handles both raw and buffered modes in a single loop, distinguished by whether `w->qh == w->nr`:

- **Raw mode** (`w->qh == w->nr`): runes come from `w->raw`, one at a time. Each rune is converted to UTF-8 bytes via `runetochar` and the raw queue is shifted left.
- **Buffered mode** (`w->qh < w->nr`): runes come from the text buffer starting at `qh`. The `qh` pointer advances as runes are consumed, and the loop breaks on newline or EOT (`'\004'`).

In both modes, the rune-to-byte conversion may produce a rune whose UTF-8 encoding straddles the byte count boundary requested by the client. The partial-rune bookkeeping with `part`/`npart` saves the overflow bytes for the next read—the same pattern used on the `xfidwrite()`[137b] side for the opposite conversion.

```
⟨winctl() event loop cases 166a⟩+≡                    (71b) ◁163h 196f▷
  case WCread:
      recv(crm.c1, &pair);
      t = pair.s;
      nb = pair.ns;

      i = 0;
      ⟨winctl() when WCRead, copy in t previous partial rune bytes 167b⟩

      while(i<nb && (w->nraw > 0 || w->qh < w->nr)){

          // raw mode
          if(w->qh == w->nr){
              wid = runetochar(t+i, &w->raw[0]);
              w->nraw--;
              runemove(w->raw, w->raw+1, w->nraw);
          // buffered mode
          }else
              wid = runetochar(t+i, &w->r[w->qh++]);

          i += wid;
          ⟨winctl() when WCRead, break if newline and handle EOF character 188b⟩
          }
      }
      ⟨winctl() when WCRead, handle EOF character after while loop 188c⟩
      ⟨winctl() when WCRead, store overflow bytes of partial rune 167a⟩

      pair.s = t;
      pair.ns = i;
      send(crm.c2, &pair);
      continue;
```

Uses `WCread-29` 165b and `runemove` 285a.

Runes vs bytes, partial runes

One detail deserves its own look. Because the client reads *bytes* while `winctl()`[71b] holds *runes*, a multibyte rune can overflow the byte count the client asked for; these fields stash the leftover bytes until the next read—the mirror image of the partial-rune handling on the `xfidwrite()`[137b] side.

```
⟨winctl() other locals 166b⟩+≡                    (71b) ◁165h 195g▷
  char part[3]; // UTFMAX-1
  int npart = 0;
```

⟨`winctl()` *when WCRead, store overflow bytes of partial rune* 167a⟩≡ (166a)
```
if(i > nb){
    npart = i-nb;
    memmove(part, t+nb, npart);
    i = nb;
}
```

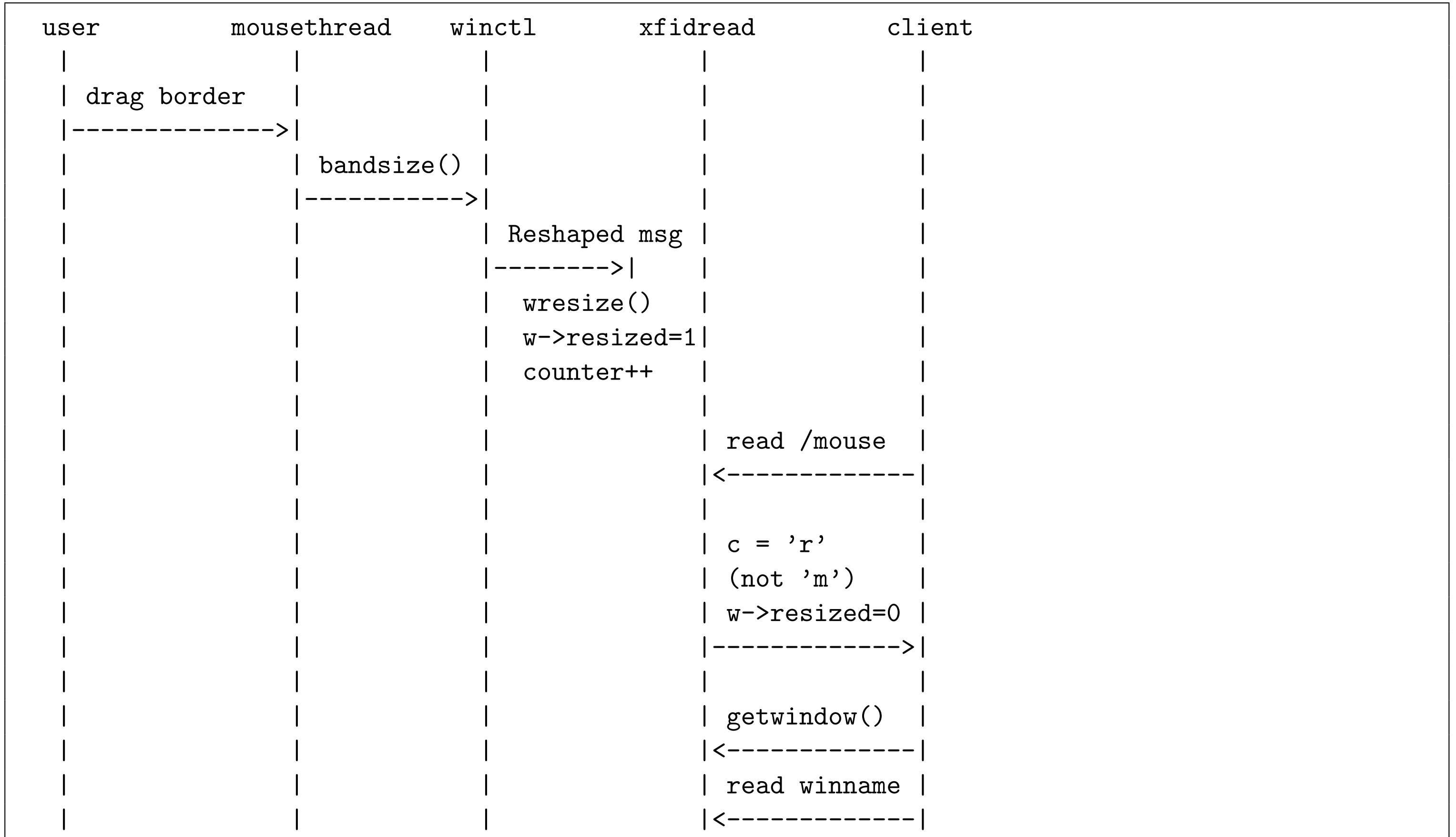
⟨`winctl()` *when WCRead, copy in t previous partial rune bytes* 167b⟩≡ (166a)
```
i = npart;
npart = 0;
if(i)
    memmove(t, part, i);
```

10.4 Resize events

Mouse events and raw keys are the two device streams a graphical window lives on; the last important one is the *resize* notification.

When a window is resized, the client must be notified so it can redraw. The notification is piggybacked on the mouse event stream: `wresize()`[114c] sets `w->resized = true` and increments `w->mouse.counter`. When the application reads `/mnt/wsys/mouse`, the response has a "r" prefix instead of "m", signaling a resize. The application calls `getwindow()` to remap the new window image.

The full handshake ties together four actors: the user dragging a border, the mouse thread inside `rio`, the target window's `winctl()`[71b] thread, and the client application blocked on `/mnt/wsys/mouse`. It is worth walking through end-to-end because the "r"-prefix trick is the only hint the client gets that anything structural changed:



The reason rio piggybacks on /mnt/wsys/mouse instead of using a dedicated /dev/resize file is backward compatibility: every graphical client already blocks on the mouse stream in its main event loop, so no extra select()/alt() is needed to notice a resize. The "r" byte is simply a one-character tag at the start of the same text packet that normally carries m x y buttons msec. libdraw's getwindow() wrapper (in Graphics book [Pad16c]) hides this from applications—they just see a resize event surface through their normal mouse channel. The cost is that a resize is invisible to any client that is not reading the mouse, and that the Window.resized[114b] flag must be cleared in xfidread() so the client does not get the same "r" over and over.

⟨xfidread() *when Qmouse, adjust c for resize message if resized* 168a⟩≡ (143c)

```
  if(w->resized)
      c = 'r';
```

10.5 Graphical window teardown

This section mirrors *Graphical window setup* at the start of the chapter. Setup turned a textual window into a graphical one—opening /dev/mouse, switching on mouseopen, and possibly entering raw keyboard mode. Teardown undoes that when the application exits or closes those files: the window stops diverting events to the client, drops back to handling them itself, and repaints its terminal frame, becoming an ordinary textual window once more. That last step makes this a natural bridge toward the textual-window machinery.

Closing /mnt/wsys/mouse runs the Qmouse case of xfidclose()[134], which clears w->mouseopen: the window stops forwarding mouse events to a client and goes back to handling them itself.

⟨xfidclose() *cases* 168b⟩+≡ (134) ◁153e 212b▷

```
  case Qmouse:
      w->mouseopen = false;
      w->resized = false;
      if(w->i != nil)
          wsendctlmesg(w, Refresh, w->i->r, nil);
      break;
```

Uses Qmouse 141a, Refresh 168c, and wsendctlmesg() 91c.

The line worth a word is the Refresh[168c] that rio sends to itself. While the graphical client was running it had been drawing its own content straight into the window image (through /dev/draw), painting over rio's terminal text. Having just cleared mouseopen, rio now needs to restore its own view of the window, and it asks the winctl()[71b] thread to do so through a Refresh control message:

⟨Wctlmesgkind *cases* 168c⟩+≡ (91a) ◁160d 271f▷

```
  Refresh,
```

⟨wctlmesg() *cases* 168d⟩+≡ (92a) ◁160e 272b▷

```
  case Refresh:
      if(w->deleted || Dx(w->screenr)<=0 || !rectclip(&r, w->i->r))
          break;
      if(!w->mouseopen)
          wrefresh(w, r);
      flushimage(display, true);
      break;
```

Uses Refresh 168c and wrefresh() 169a.

The handler's actual repaint is wrefresh()[169a]. It always repaints the window border, and—when the mouse is not open—redraws the text frame and selection underneath. The mouseopen test (here and in the handler

above) is just a guard, so `rio` never paints over a client that still owns the pixels; on this teardown path it is already false, so the frame is redrawn.

⟨*function* `wrefresh` 169a⟩≡ (342b)
```
void
wrefresh(Window *w, Rectangle)
{
    Frame *frm = &w->frm;

    /* BUG: rectangle is ignored */
    if(w == input)
        wborder(w, Selborder);
    else
        wborder(w, Unselborder);
    if(w->mouseopen)
        return;
    // else

    draw(w->i, insetrect(w->i->r, Borderwidth), frm->cols[BACK], nil,
         w->i->r.min);
    frm->ticked = false;
    if(frm->p0 > 0)
        frdrawsel(frm, frptofchar(frm, 0), 0, frm->p0, 0);
    if(frm->p1 < frm->nchars)
        frdrawsel(frm, frptofchar(frm, frm->p1), frm->p1, frm->nchars, 0);
    frdrawsel(frm, frptofchar(frm, frm->p0), frm->p0, frm->p1, 1);
    w->lastsr = ZR;
    wscrdraw(w);
}
```
Uses `BACK` 289f, `Selborder` 67, `Unselborder` 105c, `frdrawsel()` 317b, `frptofchar()` 295a, `input` 51e, `wborder()` 96a, and `wscrdraw()` 180b.

The keyboard has a symmetric teardown, though its code lives with the raw-mode machinery in Section 10.1.3. Just as setup wrote `"rawon"` to enter raw mode, closing `/dev/consctl` sends a `Rawoff`[160d]: the `Qconsctl` case of `xfidclose()` drops `rawing` back to zero and restores cooked, line-edited input. So between this section and that one, a window unwinds both halves of its graphical setup—the mouse and the keyboard—and is a plain textual window again.

10.6 /mnt/wsys/window

With `/dev/cons` and `/dev/mouse` behind us—the devices every window needs—we turn to a file that exists only for *graphical* clients: `/mnt/wsys/window`.

Reading `/mnt/wsys/window` returns the window's image ID and rectangle (in the same "12 12 12 12 id" text format used by `/dev/draw`). This is how `getwindow()` in the `libdraw` library reconnects to a window image after a resize: it reads this file to learn the new window dimensions, then uses the image ID to obtain a reference from the `draw` device. The read has two parts, selected by the file offset. At offset 0 through `5*12-1` (60 bytes), it returns a text header with the pixel format and the window rectangle. Past that offset, `readwindow()` returns the raw pixel data of the window image via `unloadimage()`—this lets a program capture a screenshot of its own window. The `caseImage` label is shared with another virtual device (`Qscreen`), avoiding code duplication.

⟨`Qxxx` *other cases* 169b⟩+≡ (122d) ◁158a 205b▷
```
Qwindow,
```

⟨*dirtab array elements* 169c⟩+≡ (123b) ◁158b 205c▷
```
{ "window",  QTFILE, Qwindow, 0400 },
```
Uses `Qwindow` 169b.

⟨xfidread() *other locals* 170a⟩+≡ (136b) ◁148c 214c▷

```
  Image *i;
  Rectangle r;
  char buf[128];
  char cbuf[30];
```

⟨xfidread() *cases* 170b⟩+≡ (136b) ◁158c 206a▷

```
  case Qwindow:
      i = w->i;
      if(i == nil || Dx(w->screenr)<=0){
          filsysrespond(x->fs, x, &fc, Enowindow);
          return;
      }
      r = w->screenr;
      /* fall through */

  caseImage:
      if(off < 5*12){
          n = sprint(buf, "%11s %11d %11d %11d %11d ",
              chantostr(cbuf, view->chan),
              i->r.min.x, i->r.min.y, i->r.max.x, i->r.max.y);
          t = estrdup(buf);
          goto Text;
      }
      t = malloc(cnt);
      fc.data = t;
      n = readwindow(i, t, r, off, cnt); /* careful; fc.count is unsigned */
      if(n < 0){
          buf[0] = '\0';
          errstr(buf, sizeof buf);
          filsysrespond(x->fs, x, &fc, buf);
      }else{
          fc.count = n;
          filsysrespond(x->fs, x, &fc, nil);
      }
      free(t);
      return;
```

Uses `Enowindow` 281k, `Qwindow` 169b, `filsysrespond()` 124, and `readwindow()` 170c.

⟨*function* readwindow 170c⟩≡ (347)

```
  int
  readwindow(Image *i, char *t, Rectangle r, int offset, int n)
  {
      int ww, y;

      offset -= 5*12;
      ww = bytesperline(r, view->depth);
      r.min.y += offset/ww;
      if(r.min.y >= r.max.y)
          return 0;
      y = r.min.y + n/ww;
      if(y < r.max.y)
          r.max.y = y;
      if(r.max.y <= r.min.y)
          return 0;
      return unloadimage(i, r, (uchar*)t, n);
  }
```

This concludes the graphical window chapter. Graphical windows are the simpler case: the application draws directly via /dev/draw, and rio's role is limited to dispatching mouse and keyboard events. The next chapter

covers textual windows—`rio`'s terminal emulator—which is considerably more complex because `rio` must handle line editing, text rendering, scrolling, and all the subtleties of emulating `/dev/cons`.

Chapter 11

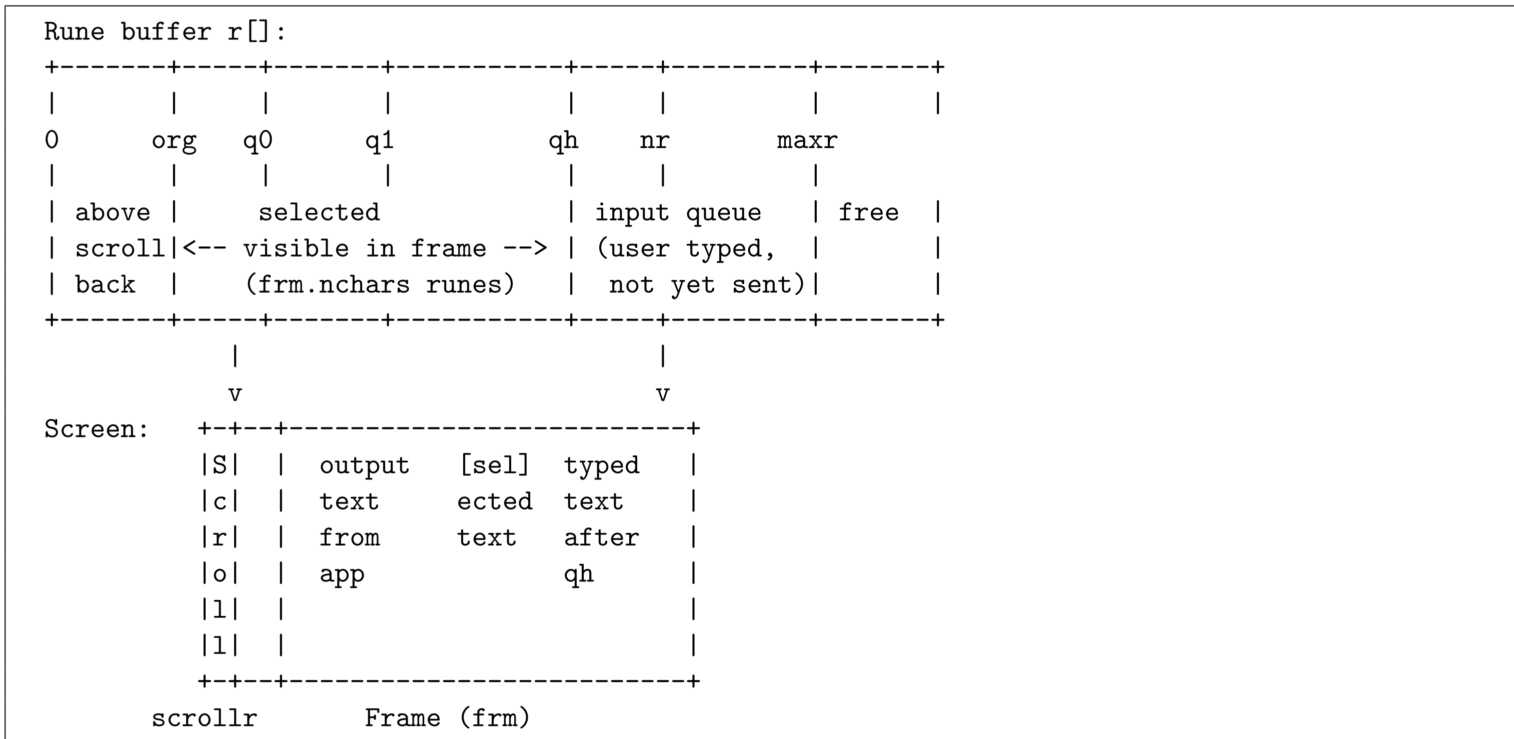
Textual Windows

A textual window is `rio`'s terminal emulator. The challenge is to let command-line programs (like `rc`) work unmodified—they just read and write `/dev/cons`—while providing richer services like mouse text selection, copy/paste, and scrolling. This separation is possible because `/dev/cons` is *virtualized*: `rio` is not echoing to a hardware terminal but keeping its own editable text buffer. Owning that buffer is what lets it track the application's write position (the *output point* `Window.qh`[52c]) as just one index into the buffer, independent of where the user is reading or selecting with the mouse.

A textual window is essentially a small editor: a data model (the rune buffer `Window.r`[51h]), cursors into it (the output point `Window.qh` and the selection `Window.q0`[52a]/`Window.q1`[52a]), a viewport (`Window.org`[52b]), and a rendering engine (the `Frame`[361b] library that lays out and paints the text). This is one of the largest chapters in the book: the textual window has many moving parts—how text is modified, how it is rendered, and the three event handlers that drive it (the keyboard, with its line discipline, navigation, and erase keys; the application output, with backspace processing and flow control; and the mouse, for selection, menu, and scrollbar). I present them bottom-up: content modification first, then rendering, then the handlers. The `Frame` widget the window builds on is documented separately, in Appendix D.

11.1 Overview

The following diagram shows the relationship between the data model (the rune buffer) and the view (what the Frame widget displays on screen). The rune buffer `Window.r`[51h] holds all text ever written by the application plus what the user has typed. `Window.org`[52b] is the index of the first visible rune; `Frame.nchars` is how many runes fit in the frame. The selection (`Window.q0`[52a]/`Window.q1`[52a]) and the output point (`Window.qh`[52c]) are indices into this same buffer:

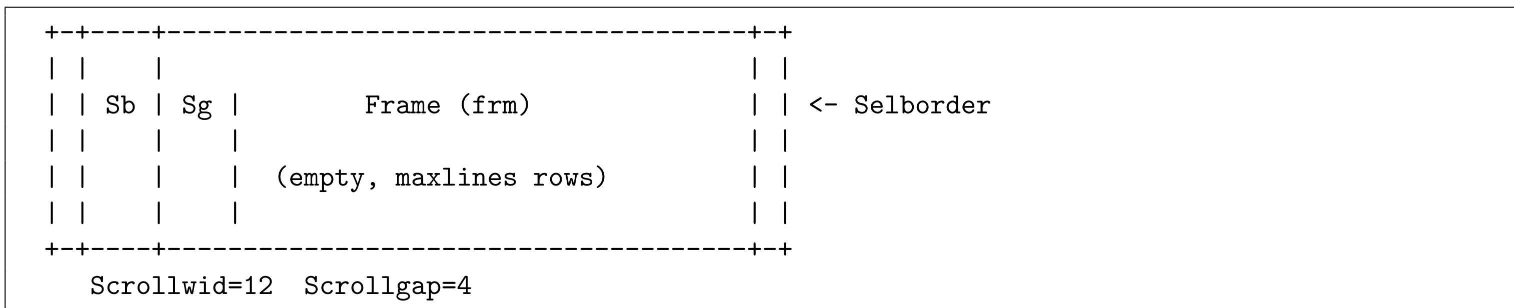


Key invariant: `org <= qh <= nr`. Text before `qh` was produced by the application (e.g., a shell prompt); text from `qh` to `nr` was typed by the user and will be sent to the application when the user presses Enter. The selection (`q0`/`q1`) can be anywhere—the user might click on old output text or on input they have not yet sent.

11.2 Textual window creation

This section fills in the textual-window-specific setup that `wmk()`[94c] performs beyond the common window creation seen earlier. The three steps are: compute the scrollbar rectangle (left strip), initialize the `Frame` widget (remaining area to the right), and paint the initial background. At this point, the rune buffer is empty (`nr == 0`, `q0 == q1 == qh == org == 0`) because `emalloc()` zeros the `Window`[49] struct. The first content will appear when the child process (e.g., `rc`) writes its prompt to `/dev/cons`, triggering a `WCwrite` event.

The following ASCII diagram shows the initial layout of a textual window, with dimensions annotated. `Sb` is the scrollbar column (`Scrollwid` = 12 pixels wide), and `Sg` is the gap between the scrollbar and the text frame (`Scrollgap` = 4 pixels wide):



⟨`wmk()` *textual window settings* 173⟩≡ (94c) 174c ▷

```
  ⟨wmk() textual window settings, set scrollbar 174b⟩
  ⟨wmk() textual window settings, set frame 174e⟩

  r = insetrect(w->i->r, Selborder);
  draw(w->i, r, cols[BACK], nil, w->frm.entire.min);
```

Uses `BACK` 289f, `Selborder` 67, and `cols-18` 289g.

The `Frame.cols` array holds the colours the frame draws with—background, text, and the highlight for selected text.

11.2.1 Scrollbar

The scrollbar occupies a `Scrollwid`[174a] (12 pixel) wide strip on the left side of the window, just inside the selection border.

⟨*constants Scrollxxx* 174a⟩≡ (333a) 174d ▷
```
Scrollwid   = 12, /* width of scroll bar */
```

Its rectangle is stored in `Window.scrollr`[52e] and used in two places: by `wscrdraw()`[180b] for drawing the scrollbar thumb, and by `wmousectl()`[199e] to detect clicks in the scrollbar area (which are dispatched to `wscroll()`[202] rather than to the text selection code). The `Window.scrolling`[52f] flag controls auto-scroll behavior and is initialized from the global `scrolling` default.

⟨`wmk()` *textual window settings, set scrollbar* 174b⟩≡ (173)
```
r = insetrect(w->i->r, Selborder+1);
w->scrollr = r;
w->scrollr.max.x = r.min.x+Scrollwid;
```
Uses `Scrollwid` 174a and `Selborder` 67.

⟨`wmk()` *textual window settings* 174c⟩+≡ (94c) ◁ 173
```
w->scrolling = scrolling; // autoscroll
```

11.2.2 Frame

The `Window.frm`[52d] field is a Frame, the Plan 9 text widget from the `libframe` library (see Appendix D). It is not rio-specific—`acme` and `sam` use it too—so I treat it here as a black box with a small public interface and defer its internals to the appendix. `rio` only ever calls a handful of its functions: `frinit()`[288d] to set up the widget over an image, `frinsert()`[304]/`frdelete()`[314] to add or remove runes (the widget then redraws only the affected lines), `frselect()`[300] and `frdrawsel()`[317b] for mouse selection, and `frcharofpt()`[297c]/`frptofchar()`[295a] to convert between a pixel point and a rune index. The fields `rio` reads directly are `Frame.nchars` (runes currently displayed), `Frame.nlines`/`Frame.maxlines` (visible versus maximum lines), and `Frame.p0`/`Frame.p1` (the selection range).

The window's drawable area (inset by `Selborder`) is divided into a scrollbar on the left (`Scrollwid` = 12 pixels wide) and the text frame on the right, separated by a `Scrollgap` (4 pixel) gap. The frame rectangle is passed to `frinit()`, which clips its bottom edge to a whole number of font-height lines and computes `maxlines`. For a typical 24-line terminal with a 16-pixel font, `maxlines` would be 24 and the clipped rectangle would be $24 \times 16 = 384$ pixels tall, with any leftover pixels below the frame unused.

⟨*constants Scrollxxx* 174d⟩+≡ (333a) ◁ 174a
```
Scrollgap   = 4, /* gap right of scroll bar */
```

⟨`wmk()` *textual window settings, set frame* 174e⟩≡ (173)
```
r = insetrect(w->i->r, Selborder+1);
r.min.x += Scrollwid+Scrollgap;
frinit(&w->frm, r, font, i, cols);
⟨wmk() textual window settings, extra frame settings 181d⟩
```
Uses `Scrollgap` 174d, `Scrollwid` 174a, `Selborder` 67, `cols-18` 289g, and `frinit()` 288d.

11.2.3 Frame colors

The frame uses five colors indexed by `BACK`[289f], `HIGH`[289f], `BORD`[289f], `TEXT`[289f], and `HTEXT`[289f]:

- `BACK` (white): normal text background.
- `HIGH` (light grey, `0xCCCCCC`): background of selected text.
- `BORD` (medium grey, `0x999999`): scrollbar and border fill.
- `TEXT` (black): normal text foreground.
- `HTEXT` (black): selected text foreground (same as TEXT in `rio`).

The key visual trick is that when a window loses focus, `wsetcols()`[175c] swaps the text color from black to dark grey (`0x666666`), giving the “inactive window” appearance without changing any content—only the colors in `frm.cols` are modified, followed by a `frredraw()`[302b]. This is a lightweight way to indicate which window has keyboard focus.

The colors are allocated once (guarded by `cols[0] == nil`) and shared by all windows. White background, light grey highlight for selected text, medium grey for the border/scrollbar, and black text. These are 1x1 pixel images used as tile patterns in `draw()` calls—a common Plan 9 idiom for solid color fills where the 1x1 image is tiled across the destination rectangle by the graphics system.

⟨`wmk()` *colors initialisation* 175a⟩≡ (94c)

```
if(cols[0] == nil){
    cols[BACK] = display->white;
    cols[HIGH] = allocimage(display, Rect(0,0,1,1), CMAP8, true, 0xCCCCCCFF);
    cols[BORD] = allocimage(display, Rect(0,0,1,1), CMAP8, true, 0x999999FF);
    cols[TEXT] = display->black;
    cols[HTEXT] = display->black;
    ⟨wmk() extra colors initialisation 96d⟩
);
}
```

Uses `BACK` 289f, `BORD` 289f, `HIGH` 289f, `HTEXT` 289f, `TEXT` 289f, and `cols-18` 289g.

⟨`wrepaint()` *update cols* 175b⟩≡ (105b)

```
wsetcols(w);
```

Uses `wsetcols()` 175c.

The `wsetcols()` function implements the visual focus indicator: the focused window (`w == input`) gets black text, while unfocused windows get dark grey text (`darkgrey`, `0x666666`). This is the only rendering difference between focused and unfocused windows—the background, selection highlight, and border colors remain the same. The simplicity of this approach (one color swap plus a repaint) is characteristic of `rio`’s minimalist design: there are no drop shadows, title bar color changes, or border thickness variations that other window systems use to indicate focus.

⟨*function* `wsetcols` 175c⟩≡ (342b)

```
void
wsetcols(Window *w)
{
    ⟨wsetcols() if holding 271a⟩
    else
        if(w == input)
            w->frm.cols[TEXT] = w->frm.cols[HTEXT] = display->black;
        else
            w->frm.cols[TEXT] = w->frm.cols[HTEXT] = darkgrey;
}
```

Uses `HTEXT` 289f, `TEXT` 289f, `darkgrey-19` 176a, and `input` 51e.

⟨*global* darkgrey 176a⟩≡ (342b)

```
static Image *darkgrey;
```

⟨wmk() *extra colors initialisation* 176b⟩+≡ (175a) ◁96d 271e▷

```
/* greys are multiples of 0x11111100+0xFF, 14* being palest */
darkgrey = allocimage(display, Rect(0,0,1,1), CMAP8, true, 0x666666FF);
```

Uses darkgrey-19 176a.

11.3 Content modification

When text is inserted (either from keyboard input or application output), `winsert()`[177a] moves the runes after the insertion point to make space, copies the new runes in, then adjusts all the cursor positions (`q0`, `q1`, `qh`, `org`) that fall after the insertion point. This is simpler than a *gap buffer*, the classic editor data structure that keeps a movable empty *gap* at the cursor so inserting there is free and only moving the cursor shifts memory. `rio` does not bother: `winsert()` just `runemove()`s the runes after the insertion point. That would be wasteful in a general editor, but a terminal almost always inserts at the very end, where there is nothing after it to shift, so the plain array wins.

Here is a walkthrough of what happens when the user types a single character 'x' at the cursor. The call chain is:

1. `wkeyctl()`[73b] receives the rune from `winctl()`[71b].
2. `winsert(w, &r, 1, w->q0)` shifts `r[q0..]` right by 1, inserts 'x', increments `nr`, and bumps `q0`/`q1` forward.
3. Since `q0 >= org`, `frinsert(&w->frm, &r, &r+1, q0-org)` updates the on-screen box array: it scans 'x' into a new one-character box (`bxscan`[310]), shifts existing boxes right on screen, and draws the new character.
4. `wshow(w, q0+1)` checks that the cursor is still visible; if it scrolled off the bottom, it adjusts `org` via `wsetorigin`[183].

⟨*simplified code when entered a single rune* r *in a terminal* 176c⟩≡

```
winsert(w, &r, 1, w->q0);
wshow(w, w->q0);
```

The real `wkeyctl()` does more around this core—it first dispatches special keys (navigation, erase, completion) and handles raw mode—but for an ordinary character it comes down to exactly these two calls: insert, then make sure the cursor is visible.

11.3.1 Inserting runes: `winsert()`

`winsert()`[177a] is the core insertion function. The algorithm is straightforward: shift the runes after `q0` to the right by `n` positions (with `runemove()`), copy the new runes into the gap, then adjust all four cursor positions (`q0`, `q1`, `qh`, `org`). The cursor adjustment is subtle: the `<=` vs `<` distinction matters. `q0` and `q1` use `<=` so that inserting *at* the cursor pushes it forward; `qh` uses `<` so that inserting at the output point leaves `qh` in place (the newly typed text goes *after* `qh`, which is what you want when typing at the prompt). Similarly, `org` uses `<` so that inserting at the viewport start does not shift the view. Example: the shell printed "$ " (org=0, qh=2, q0=2, q1=2). The user types 'a':

```
Before:  |$| |       q0=2, qh=2, org=0, nr=2
Insert at q0=2:
After:   |$| |a|     q0=3 (<=2, bumped), qh=2 (<2, stays), nr=3
```

The cursor moved past 'a', but the output point stayed—exactly right: 'a' is user input waiting to be sent.

⟨*function* winsert 177a⟩≡ (343b)
```
uint
winsert(Window *w, Rune *r, int n, uint q0)
{
    uint m;

    ⟨winsert() sanity check n 177c⟩
    ⟨winsert() if size of rune array is getting really big 178b⟩
    ⟨winsert() grow rune array if reach maxr 178a⟩

    // move to the right the runes after the cursor q0 to make some space
    runemove(w->r + q0 + n, w->r + q0, w->nr - q0);
    // fill the space
    runemove(w->r + q0, r, n);
    w->nr += n;

    ⟨winsert() adjust cursors 177b⟩

    return q0;
}
```
Uses runemove 285a.

⟨winsert() *adjust cursors* 177b⟩≡ (177a)
```
/* if output touches, advance selection, not qh; works best for keyboard and output */
if(q0 <= w->q0)
    w->q0 += n; // move the q0 cursor
if(q0 <= w->q1)
    w->q1 += n;
if(q0 < w->qh)
    w->qh += n;

if(q0 < w->org)
    w->org += n;
else
   ⟨winsert() when q0 >= w->org, possibly update visible text 179a⟩
```

⟨winsert() *sanity check n* 177c⟩≡ (177a)
```
if(n == 0)
    return q0;
```

11.3.2 Growing array

The rune buffer uses a two-threshold strategy to limit memory. When `Window.nr`[51h] exceeds `HiWater` (640K runes, roughly 1.2 MB), the oldest `HiWater - LoWater` = 240K runes are discarded at once by shifting the array left with `runemove()` and adjusting all cursors. The bulk pruning amortizes the cost of the shift—doing it one rune at a time would be prohibitively expensive. `MinWater` (20K) provides headroom so that a burst of output after pruning does not immediately trigger another reallocation.

⟨*enum* _anon_ (rio/wind.c) 177d⟩≡ (343b)
```
enum
{
    HiWater = 640000, /* max size of history */
    LoWater = 400000, /* min size of history after max'ed */
    MinWater = 20000, /* room to leave available when reallocating */
};
```

The allocation strategy for the rune buffer uses geometric doubling (up to `HiWater`) with a `MinWater` headroom to minimize `realloc` calls. For a typical terminal session, the buffer grows quickly at first and then stabilizes around `HiWater` once pruning kicks in.

⟨`winsert()` *grow rune array if reach maxr* 178a⟩≡ (177a)

```
if(w->nr+n > w->maxr){
    /*
     * Minimize realloc breakage:
     * Allocate at least MinWater
     *  Double allocation size each time
     * But don't go much above HiWater
     */
    m = max(min(2*(w->nr+n), HiWater), w->nr+n)+MinWater;
    if(m > HiWater)
        m = max(HiWater+MinWater, w->nr+n);
    if(m > w->maxr){
        w->r = runerealloc(w->r, m);
        w->maxr = m;
    }
}
```

Uses `HiWater-33` 177d, `MinWater-35` 177d, `max()` 284b, `min()` 284a, and `runerealloc` 284e.

When the buffer exceeds `HiWater`, the oldest text is discarded. The guard `q0 >= w->org && q0 >= w->qh` ensures pruning only happens when the insertion is at or after both the viewport and the output point—otherwise removing old text could invalidate what the user is looking at or what the application is about to read. The amount removed is `min(HiWater - LoWater, min(org, qh))`, so it never removes more text than exists before the earlier of the two pointers.

⟨`winsert()` *if size of rune array is getting really big* 178b⟩≡ (177a)

```
if(w->nr + n > HiWater && q0 >= w->org && q0 >= w->qh){
    m = min(HiWater-LoWater, min(w->org, w->qh));
    w->org -= m;
    w->qh -= m;
    if(w->q0 > m)
        w->q0 -= m;
    else
        w->q0 = 0;
    if(w->q1 > m)
        w->q1 -= m;
    else
        w->q1 = 0;
    w->nr -= m;
    runemove(w->r, w->r+m, w->nr);
    q0 -= m;
}
```

Uses `HiWater-33` 177d, `LoWater-34` 177d, `min()` 284a, and `runemove` 285a.

11.4 Content rendering

Rendering is where `rio` hands off to the Frame library (Appendix D). `rio`'s own job here is small: translate a rune-buffer edit into the right `frinsert()`[304]/`frdelete()`[314] call, then keep the cursor on screen. The Frame library does the hard part—turning that edit into an *incremental* screen update that redraws only the lines that actually changed.

Why incremental rendering? Because a textual window can easily have 50–80 lines of visible text, each containing dozens of characters. Redrawing all of them on every keystroke would be visibly slow on the hardware Plan 9 targets. Instead, inserting one character typically only repaints one line (or less), making typing feel

instant. The two entry points are `frinsert()`, which inserts runes and redraws only the affected lines, and `frdelete()`, which removes a range and collapses the display. Both are detailed in Appendix D; here I show only where and how `rio` calls them. The pipeline below spans both layers: the `w...` functions are rio's (this chapter), while `frinsert()/frdelete()` and their helpers belong to the Frame library (Appendix D):

```
winsert()/wdelete()     -- modify the rune buffer and cursors
     |
     v
frinsert()/frdelete()   -- update the box array and screen
     |
     +--- bxscan()       -- convert new runes to boxes
     +--- _frdrawtext() -- draw the new boxes
     +--- frdrawsel()   -- repaint selection highlighting
     +--- _frclean()    -- merge adjacent boxes
     |
     v
wshow() -> wsetorigin() -> wfill()  -- scroll viewport if needed
```

11.4.1 Updating the visible text: `frinsert()`

`frinsert()`[304] is the `Frame`[361b] library's incremental update: rather than repaint the window, it splices the new runes into the on-screen box array and redraws only what changed.

⟨`winsert()` *when* `q0 >= w->org`, *possibly update visible text* 179a⟩≡ (177b)

```
if(q0 <= w->org + w->frm.nchars)
    frinsert(&w->frm, r, r+n, q0 - w->org); // echo back
```

Uses `frinsert()` 304.

11.4.2 Keeping the cursor visible: `wshow()`

The `wshow()`[179b] function ensures that position `q0` is visible on screen. If `q0` is already within the viewport (`org` to `org + nchars`), it just redraws the scrollbar (which may have changed if text was inserted or deleted). Otherwise, it recomputes `org` to place `q0` near the bottom of the viewport (leaving 4/5 of the screen as context above) and calls `wsetorigin()`[183] to reflow the display from the new origin. This function is called after every text modification (`winsert()`[177a], navigation key, application output with auto-scroll). It is the glue that keeps the cursor visible without the caller needing to worry about viewport management.

⟨*function* `wshow` 179b⟩≡ (343b)

```
void
wshow(Window *w, uint q0)
{
    int qe;
    int nl;
    uint q;

    qe = w->org + w->frm.nchars;
    if(w->org <= q0 && (q0 < qe || (q0 == qe && qe == w->nr)))
        wscrdraw(w);
    ⟨wshow() else, when q0 is out of scope 182c⟩
}
```

Uses `wscrdraw()` 180b.

11.4.3 Drawing the scrollbar: `wscrdraw()`

⟨`wmk()` *drawing scrollbar* 180a⟩≡ (94c)
```
wscrdraw(w);
```
Uses `wscrdraw()` 180b.

The scrollbar is drawn into a scratch image (`scrtmp`) and then blitted to the window in a single operation, avoiding flicker from intermediate drawing states. The "thumb" (the white portion indicating the viewport position) is computed by `scrpos`[180c], which maps the character range `org..org+nchars` onto the scrollbar height proportionally to `nr` (total characters). The border color fills the non-thumb area, creating a visual distinction: the dark portion above the thumb represents text before the viewport, the dark portion below represents text after it. When all text fits on screen, the entire scrollbar is white.

⟨*function* `wscrdraw` 180b⟩≡ (348)
```
void
wscrdraw(Window *w)
{
    Rectangle r, r1, r2;
    Image *b;

    scrtemps();
    ⟨wscrdraw() sanity check the window image 181f⟩
    r = w->scrollr;
    b = scrtmp;
    // r1 is translation of r to (0,...)
    r1 = r;
    r1.min.x = 0;
    r1.max.x = Dx(r);
    r2 = scrpos(r1, w->org, w->org + w->frm.nchars, w->nr);
    if(!eqrect(r2, w->lastsr)){
        w->lastsr = r2;
        /* move r1, r2 to (0,0) to avoid clipping */
        r2 = rectsubpt(r2, r1.min);
        r1 = rectsubpt(r1, r1.min);
        draw(b, r1, w->frm.cols[BORD], nil, ZP);
        draw(b, r2, w->frm.cols[BACK], nil, ZP);
        // little separation line
        r2.min.x = r2.max.x-1;
        draw(b, r2, w->frm.cols[BORD], nil, ZP);

        // transfer back to main image
        draw(w->i, r, b, nil, Pt(0, r1.min.y));
    }
}
```
Uses `BACK` 289f, `BORD` 289f, `scrpos()` 180c, `scrtemps()` 182a, and `scrtmp-50` 181g.

The `scrpos()`[180c] function maps a character range `p0..p1` (the visible portion) onto a rectangle within the scrollbar `r`, proportional to the total buffer size `tot`. The calculation is straightforward: `q.min.y` = `r.min.y` + `h * p0 / tot` and `q.max.y` = `r.max.y` − `h * (tot - p1) / tot`. The thumb (white portion) represents what the user can see; the surrounding border-colored area represents text above and below the viewport. A minimum height of 2 pixels ensures the thumb is always visible even for very large buffers. For buffers over 1M characters, the values are shifted right by 10 bits before the multiplication to prevent integer overflow in the 32-bit arithmetic.

⟨*function* `scrpos` 180c⟩≡ (348)
```
static
Rectangle
scrpos(Rectangle r, uint p0, uint p1, uint tot)
{
```

```
    Rectangle q;
    int h;

    q = r;
    h = q.max.y - q.min.y; // Dy(r)
    if(tot == 0)
        return q;
    ⟨scrpos() adjust integers if total is big 181b⟩
    if(p0 > 0)
        q.min.y += h*p0 / tot;
    if(p1 < tot)
        q.max.y -= h*(tot-p1) / tot;

    ⟨scrpos() last adjustments 181a⟩
    return q;
}
```

⟨scrpos() *last adjustments* 181a⟩≡ (180c)

```
if(q.max.y < q.min.y+2){
    if(q.min.y+2 <= r.max.y)
        q.max.y = q.min.y+2;
    else
        q.min.y = q.max.y-2;
}
```

⟨scrpos() *adjust integers if total is big* 181b⟩≡ (180c)

```
if(tot > 1024*1024){
    tot>>=10;
    p0>>=10;
    p1>>=10;
}
```

An optimization avoids redundant redraws: `Window.lastsr`[181c] caches the previous thumb rectangle, and the scrollbar is only repainted if the new rectangle differs. This matters because `wscrdraw()`[180b] is called on every text change, and most of the time the scrollbar position does not change visibly.

⟨Window *other fields* 181c⟩+≡ (49) ◁152c 212c▷

```
Rectangle lastsr;
```

⟨wmk() *textual window settings, extra frame settings* 181d⟩≡ (174e) 195a▷

```
w->lastsr = ZR;
```

⟨wresize() *textual window updates, reset lastsr* 181e⟩≡ (205a)

```
w->lastsr = ZR;
```

⟨wscrdraw() *sanity check the window image* 181f⟩≡ (180b)

```
if(w->i == nil)
    error("scrdraw");
```

Uses error() 282c.

⟨*global* scrtmp 181g⟩≡ (348)

```
static Image *scrtmp;
```

⟨*constant* BIG 181h⟩≡ (333a)

```
BIG   = 3, /* factor by which window dimension can exceed screen */
```

⟨*function* scrtemps 182a⟩≡ (348)

```
static
void
scrtemps(void)
{
    int h;

    if(scrtmp)
        return;
    h = BIG * Dy(view->r);
    scrtmp = allocimage(display, Rect(0, 0, 32, h), view->chan, false, DWhite);
    ⟨scrtemps() sanity check scrtmp 182b⟩
}
```

Uses BIG 181h and scrtmp-50 181g.

⟨scrtemps() *sanity check scrtmp* 182b⟩≡ (182a)

```
if(scrtmp == nil)
    error("scrtemps");
```

Uses error() 282c and scrtmp-50 181g.

11.4.4 Moving the frame origin

When the viewport must change (because the target position is off-screen), `wsetorigin()`[183] updates `Window.org`[52b] and incrementally adjusts the frame. If the new origin is close to the old one, it uses `frdelete()`[314] or `frinsert()`[304] at position 0 to shift the content rather than redrawing everything. Only when the shift is too large does it clear the frame and rebuild from scratch. The `exact` flag controls whether `org` must be a line boundary: when false (e.g., during incremental scrolling), the function searches forward for the nearest newline to avoid starting mid-line.

When `q0` is off-screen, `wshow()`[179b] recenters the viewport. It backs up by 4/5 of the visible lines from `q0` to find a new origin that places `q0` near the bottom of the screen—this keeps some context above the cursor. The guard against going backward prevents a pathological case with very long lines. The `while` loop at the end handles the rare case where the initial repositioning still does not make `q0` visible (again, very long lines).

⟨wshow() *else, when q0 is out of scope* 182c⟩≡ (179b)

```
else{
    nl = 4 * w->frm.maxlines / 5;
    q = wbacknl(w, q0, nl);
    /* avoid going backwards if trying to go forwards - long lines! */
    if(!(q0 > w->org && q < w->org))
        wsetorigin(w, q, true);

    while(q0 > w->org + w->frm.nchars)
        wsetorigin(w, w->org+1, false);
}
```

Uses wbacknl() 182d and wsetorigin() 183.

The `wbacknl()`[182d] function finds the start of the `n`th line before position `p` by scanning backward for newlines. To handle very long lines (e.g., a binary file dumped to the terminal), it caps the backward scan at 128 characters per line—if no newline is found within 128 characters, it treats that as a line boundary. This prevents pathological performance on single extremely long lines.

⟨*function* wbacknl 182d⟩≡ (343b)

```
uint
wbacknl(Window *w, uint p, uint n)
{
    int i, j;
```

```
    /* look for start of this line if n==0 */
    if(n==0 && p>0 && w->r[p-1]!='\n')
        n = 1;

    i = n;
    while(i-->0 && p>0){
        --p; /* it's at a newline now; back over it */
        if(p == 0)
            break;
        /* at 128 chars, call it a line anyway */
        for(j=128; --j>0 && p>0; p--)
            if(w->r[p-1]=='\n')
                break;
    }
    return p;
}
```

The `wsetorigin()` function changes the viewport origin and updates the display. It uses three strategies depending on how far the new origin is from the old one:

- **Small forward shift** (`a >= 0` and `a < nchars`): call `frdelete(0, a)` to remove the top `a` characters, which shifts everything up. Then `wfill()`[184c] fills in new text at the bottom.
- **Small backward shift** (`a < 0` and `-a < nchars`): call `frinsert()` at position 0 to push existing text down and make room for the newly revealed text at the top.
- **Large shift**: clear the entire frame and rebuild from scratch.

When `exact` is false (e.g., during incremental scrolling), the function searches forward up to 256 characters for a newline, so the viewport always starts at a line boundary rather than in the middle of a line.

⟨*function* `wsetorigin` 183⟩≡ (343b)

```
void
wsetorigin(Window *w, uint org, bool exact)
{
    int i, a, fixup;
    Rune *r;
    uint n;
    Frame *frm = &w->frm;

    if(org>0 && !exact){
        /* org is an estimate of the char posn; find a newline */
        /* don't try harder than 256 chars */
        for(i=0; i<256 && org < w->nr; i++){
            if(w->r[org] == '\n'){
                org++;
                break;
            }
            org++;
        }
    }
    a = org - w->org;
    fixup = 0;
    if(a>=0 && a < frm->nchars){
        frdelete(frm, 0, a);
        fixup = 1; /* frdelete can leave end of last line in wrong selection mode; it doesn't know what follows
    }else if(a<0 && -a < frm->nchars){
        n = w->org - org;
        r = runemalloc(n);
        runemove(r, w->r+org, n);
```

```
        frinsert(frm, r, r+n, 0);
        free(r);
    }else
        frdelete(frm, 0, frm->nchars);
    w->org = org;
    wfill(w);
    wscrdraw(w);
    wsetselect(w, w->q0, w->q1);
    if(fixup && frm->p1 > frm->p0)
        frdrawsel(frm, frptofchar(frm, frm->p1-1), frm->p1-1, frm->p1, 1);
}
```

Uses `frdelete()` 314, `frdrawsel()` 317b, `frinsert()` 304, `frptofchar()` 295a, `runemalloc` 284d, `runemove` 285a, `wfill()` 184c, `wscrdraw()` 180b, and `wsetselect()` 185.

⟨`Frame` *text fields* 184a⟩≡ (288a) 194d ▷

```
ushort lastlinefull; /* last line fills frame */
```

⟨`frinit()` *initialize other fields* 184b⟩≡ (288d) 194e ▷

```
f->lastlinefull = 0;
```

After a viewport change or deletion, the frame may have empty lines at the bottom. `wfill()` fills them by grabbing runes from `w->r` starting at `org + nchars` (the first rune after what is currently displayed) and feeding them to `frinsert()` in batches of up to 2000 runes. The function counts newlines in each batch and stops after inserting enough to fill `maxlines - nlines` remaining lines. This avoids scanning the entire remaining buffer when only a few lines need filling. The outer `do/while` loop repeats until `lastlinefull` is set by `_frclean`[313a] inside `frinsert`[304], indicating that the frame is completely filled. If the rune buffer is exhausted (`n == 0`), the loop also terminates—the frame simply has fewer lines than `maxlines`.

⟨*function* `wfill` 184c⟩≡ (343b)

```
void
wfill(Window *w)
{
    Rune *rp;
    int i, n, m, nl;
    Frame *frm = &w->frm;

    if(frm->lastlinefull)
        return;
    rp = malloc(messagesize);
    do{
        n = w->nr - (w->org + frm->nchars);
        if(n == 0)
            break;
        if(n > 2000) /* educated guess at reasonable amount */
            n = 2000;
        runemove(rp, w->r + (w->org + frm->nchars), n);
        /*
         * it's expensive to frinsert more than we need, so
         * count newlines.
         */
        nl = frm->maxlines - frm->nlines;
        m = 0;
        for(i=0; i<n; ){
            if(rp[i++] == '\n'){
                m++;
                if(m >= nl)
                    break;
            }
        }
```

```
        frinsert(frm, rp, rp+i, frm->nchars);
    } while(frm->lastlinefull == false);
    free(rp);
}
```

Uses frinsert() 304, messagesize 76a, and runemove 285a.

11.4.5 Selecting

wsetselect()[185] updates the selection (Window.q0[52a], Window.q1[52a]) and repaints only the changed portions. It converts the buffer-wide positions (q0, q1) to frame-local positions (Frame.p0, Frame.p1) by subtracting Window.org[52b], then compares against the old selection to minimize redrawing. If the new and old selections overlap, it only paints the delta (the parts that changed from selected to unselected or vice versa). frselect()[300] is the interactive selection loop: it tracks the mouse while button 1 is held down, extending or shrinking the selection as the cursor moves. If the cursor goes above or below the frame, it triggers scrolling via the scroll callback.

The wsetselect() function translates buffer-wide coordinates q0/q1 to frame-local p0/p1 (by subtracting org and clamping to nchars), then incrementally repaints only the portions that changed. The four-way comparison is the key optimization: when the user drags a selection, each mouse event extends or shrinks one end by a few characters. Rather than repainting the entire selection (which could be hundreds of characters), wsetselect() identifies which characters changed state (selected ↔ unselected) and repaints only those. This makes interactive selection smooth even for large selections.

⟨*function* wsetselect 185⟩≡ (343b)

```
void
wsetselect(Window *w, uint q0, uint q1)
{
    int p0, p1;
    Frame *frm = &w->frm;

    /* w->p0 and w->p1 are always right; w->q0 and w->q1 may be off */
    w->q0 = q0;
    w->q1 = q1;
    /* compute desired p0,p1 from q0,q1 */
    p0 = q0-w->org;
    p1 = q1-w->org;
    if(p0 < 0)
        p0 = 0;
    if(p1 < 0)
        p1 = 0;
    if(p0 > frm->nchars)
        p0 = frm->nchars;
    if(p1 > frm->nchars)
        p1 = frm->nchars;
    if(p0 == frm->p0 && p1 == frm->p1)
        return;

    /* screen disagrees with desired selection */
    if(frm->p1 <= p0 || p1 <= frm->p0 || p0==p1 || frm->p1 == frm->p0){
        /* no overlap or too easy to bother trying */
        frdrawsel(frm, frptofchar(frm, frm->p0), frm->p0, frm->p1, 0);
        frdrawsel(frm, frptofchar(frm, p0), p0, p1, 1);
        goto Return;
    }
    /* overlap; avoid unnecessary painting */
    if(p0 < frm->p0){
        /* extend selection backwards */
```

```
        frdrawsel(frm, frptofchar(frm, p0), p0, frm->p0, 1);
    }else if(p0 > frm->p0){
        /* trim first part of selection */
        frdrawsel(frm, frptofchar(frm, frm->p0), frm->p0, p0, 0);
    }
    if(p1 > frm->p1){
        /* extend selection forwards */
        frdrawsel(frm, frptofchar(frm, frm->p1), frm->p1, p1, 1);
    }else if(p1 < frm->p1){
        /* trim last part of selection */
        frdrawsel(frm, frptofchar(frm, p1), p1, frm->p1, 0);
    }

    Return:
    frm->p0 = p0;
    frm->p1 = p1;
}
```

Uses `frdrawsel()` 317b and `frptofchar()` 295a.

11.4.6 Repainting

A full repaint is needed when the frame colors change (window gains or loses focus). `frredraw()`[302b] redraws all text in three passes: unselected text before `p0`, selected text between `p0` and `p1`, and unselected text after `p1`. The tick must be removed before repainting and restored afterward, because the underlying pixels change.

⟨`wrepaint()` *after updated cols, redraw content if mouse not opened* 186⟩≡ (105b)

```
if(!w->mouseopen)
    frredraw(&w->frm);
```

Uses `frredraw()` 302b.

11.5 Keyboard events

In a textual window, keyboard input goes through two modes. In the normal buffered mode (also called "cooked" mode), characters are inserted at `Window.q0`[52a] and accumulated in the rune buffer. The user can edit freely—backspace, kill-word, even mouse-select and paste—until pressing Enter, at which point the text between the output point (`Window.qh`[52c]) and the newline is sent to the application via `/dev/cons`. In raw mode (activated by writing `"rawon"` to `/dev/consctl`), characters are sent immediately without waiting for a newline. Applications like text editors or interactive programs use this mode. Special keys (Delete, arrows, Home/End, Tab, `^A`, `^E`) are intercepted and handled directly by `rio` rather than being passed through to the application. This means that even in raw mode, arrow keys still work for scrolling the terminal viewport. The keyboard event flow is:

1. `keyboardthread()`[64] receives a rune from the keyboard device.
2. `winctl()`[71b] dispatches it to `wkeyctl()`[73b].
3. `wkeyctl()` checks for raw mode, navigation keys, special keys, then falls through to ordinary character insertion.

11.5.1 Text input queue

There is no separate queue data structure for keyboard input. The region `r[qh..nr]` *is* the input queue: it contains runes that the user typed (or pasted) but that the application has not yet consumed via `read("/dev/cons")`. When the user presses Enter, the line discipline (seen in the consumer below) scans this region for a newline

and sends everything up to and including it. This design has an elegant consequence: the user's input is visible on screen and fully editable before being committed. If you type "ls -la" and realize you meant "ls -lh", you can backspace and fix it. The application never sees "ls -la" followed by corrections—it only sees the final "ls -lh\n". This resembles UNIX's *cooked* (canonical) terminal mode, but with one real difference: *where* the editing lives. In UNIX, cooked-mode line editing is done by the kernel's tty line discipline—the terminal emulator (xterm and friends) merely relays bytes to and from the pseudo-tty—and only in *raw* mode does the application edit the line itself with a library like `readline`. Plan 9 instead does the editing in the terminal, in user space, where it can even use the mouse.

11.5.2 Reading /mnt/wsys/cons: cooked mode

We have already seen two pieces of the console read: the *file-server side* in the Virtual Devices chapter (the `xfidread()`[136b] worker that waits for a `Consreadmesg`[147d] from `winctl()`[71b]), and the raw-mode *window-thread side* in the Graphical Windows chapter. This is the third: the textual-window (cooked) path—how `wkeyctl()`[73b] inserts ordinary characters into the rune buffer (the producer), and how `winctl()` line-buffers them, scanning for a newline before enabling the `WCread` consumer. The key difference from raw mode is that the rune buffer `r[qh..nr]` serves as an editable line buffer. The user can type, backspace, `^W` to erase a word, even use mouse selection and paste—all of which modify `r` and move `q0`. None of these edits are visible to the application. Only when a newline appears in `r[qh..nr]` does the consumer activate and send the committed line.

Producer

```
⟨wkeyctl() when not rawing 187a⟩≡                                    (73b)
  // here when no navigation key, no rawing, no 0x1B holding

  ⟨wkeyctl() snarf and cut if not interrupt key 230a⟩
  switch(r){
  ⟨wkeyctl() special key cases and no special mode 192a⟩
  }
  // else

  /* otherwise ordinary character; just insert */
  ⟨wkeyctl() ordinary character 187c⟩
```

```
⟨wkeyctl() locals 187b⟩≡                                       (73b)  188e ▷
  uint q0;
```

```
⟨wkeyctl() ordinary character 187c⟩≡                                 (187a)
  q0 = w->q0;
  q0 = winsert(w, &r, 1, q0);
  wshow(w, q0+1);
```

Uses `winsert()` 177a and `wshow()` 179b.

Consumer

In cooked (non-raw) mode, the application's `read()` of `/dev/cons` should block until the user types a complete line. The event loop implements this by scanning from `qh` to `nr` looking for a newline or EOT (`'\004'`). Only when one is found does it enable the `WCread` alternative (setting it to `CHANSND`). This is the classic line discipline: characters are buffered and editable until Enter commits the whole line. This design means the user can freely edit text they have typed (using backspace, `^W`, `^U`, or even mouse selection and paste) *before* pressing Enter. The

application never sees the intermediate edits—only the final committed line. This is a significant improvement over a raw terminal where every keystroke is irrevocable.

⟨`winctl()` *alts adjustments, revert to CHANSND if newline in queue* 188a⟩≡ (165g)
```
/* this code depends on NL and EOT fitting in a single byte */
/* kind of expensive for each loop; worth precomputing? */
for(i = w->qh; i < w->nr; i++){
     c = w->r[i];
     // buffering, until get a newline in which case we are ready to send
     if(c=='\n' || c=='\004'){
         alts[WCread].op = CHANSND;
         break;
     }
 }
```
Uses `WCread-29` 165b.

⟨`winctl()` *when WCRead, break if newline and handle EOF character* 188b⟩≡ (166a)
```
c = t[i-wid]; /* knows break characters fit in a byte */
if(!w->rawing && (c == '\n' || c=='\004')){
    if(c == '\004')
        i--;
    break;
```

⟨`winctl()` *when WCRead, handle EOF character after while loop* 188c⟩≡ (166a)
```
if(i==nb && w->qh < w->nr && w->r[w->qh]=='\004')
    w->qh++;
```

11.5.3 Navigation keys

The navigation keys move the viewport or the cursor within the rune buffer. They are split into two groups:

- **Scrolling keys** (Down, Up, Page Down, Page Up, scroll wheel): these change `Window.org`[52b] without moving the cursor. The user can scroll through output history while their cursor stays at the typing position.
- **Arrow keys** (Left, Right): these move the selection point `q0`/`q1` character by character and call `wshow`[179b] to ensure the cursor stays visible. If the cursor moves off-screen, the viewport scrolls to follow.

These keys are only active when `mouseopen` is false, that is, when `rio` is acting as a terminal rather than forwarding raw input to a graphical application. In graphical mode, the application handles its own keyboard navigation.

⟨`wkeyctl()` *when mouse not opened and navigation keys* 188d⟩≡ (73b)
```
if(!w->mouseopen)
switch(r){
⟨wkeyctl() when mouse not opened, switch key cases 189a⟩
default:
    ; // no return! fallthrough
}
```

⟨`wkeyctl()` *locals* 188e⟩+≡ (73b) ◁187b 194a▷
```
uint q1;
int n, nb;
```

Text boundaries (Home, end)

⟨wkeyctl() *when mouse not opened, switch key cases* 189a⟩≡ (188d) 189b ▷
```
  case Khome:
      wshow(w, 0);
      return;
```
Uses wshow() 179b.

⟨wkeyctl() *when mouse not opened, switch key cases* 189b⟩+≡ (188d) ◁ 189a 189c ▷
```
  case Kend:
      wshow(w, w->nr);
      return;
```
Uses wshow() 179b.

Down

All three downward-scrolling keys—Down arrow, mouse scroll wheel, and Page Down—share the `case_Down` target via `goto`. They differ only in how many lines to scroll: Down moves by a third of the visible area, Page Down by two thirds, and mouse scroll by a small fixed amount (`mousescrollsize()`, which is typically 1–3 lines). The trick is converting a line count `n` into a character offset: `frcharofpt()`[297c] takes a pixel coordinate (the Y position `n` lines below the top of the frame) and returns the character index at that point, which becomes the new `org` passed to `wsetorigin()`[183]. The upward scrolling keys (Up, Page Up, scroll wheel up) work the same way but use `wbacknl()`[182d] to count lines backward instead.

⟨wkeyctl() *when mouse not opened, switch key cases* 189c⟩+≡ (188d) ◁ 189b 189d ▷
```
  case Kdown:
      n = w->frm.maxlines / 3;
      goto case_Down;
```

⟨wkeyctl() *when mouse not opened, switch key cases* 189d⟩+≡ (188d) ◁ 189c 189e ▷
```
  case Kscrollonedown:
      n = mousescrollsize(w->frm.maxlines);
      if(n <= 0)
          n = 1;
      goto case_Down;
```
Uses Kscrollonedown 203a.

⟨wkeyctl() *when mouse not opened, switch key cases* 189e⟩+≡ (188d) ◁ 189d 189f ▷
```
  case Kpgdown:
      n = 2 * w->frm.maxlines / 3;
      // Fallthrough
  case_Down:
      q0 = w->org +
          frcharofpt(&w->frm, Pt(w->frm.r.min.x,
                                 w->frm.r.min.y + n * w->frm.font->height));
      wsetorigin(w, q0, true);
      return;
```
Uses frcharofpt() 297c and wsetorigin() 183.

Up

⟨wkeyctl() *when mouse not opened, switch key cases* 189f⟩+≡ (188d) ◁ 189e 190a ▷
```
  case Kup:
      n = w->frm.maxlines/3;
      goto case_Up;
```

⟨wkeyctl() *when mouse not opened, switch key cases* 190a⟩+≡ (188d) ◁189f 190b▷

```
  case Kscrolloneup:
      n = mousescrollsize(w->frm.maxlines);
      if(n <= 0)
          n = 1;
      goto case_Up;
```

Uses Kscrolloneup 203a.

⟨wkeyctl() *when mouse not opened, switch key cases* 190b⟩+≡ (188d) ◁190a 190c▷

```
  case Kpgup:
      n = 2*w->frm.maxlines/3;
      // Fallthrough
  case_Up:
      q0 = wbacknl(w, w->org, n);
      wsetorigin(w, q0, true);
      return;
```

Uses wbacknl() 182d and wsetorigin() 183.

Left

⟨wkeyctl() *when mouse not opened, switch key cases* 190c⟩+≡ (188d) ◁190b 190d▷

```
  case Kleft:
      if(w->q0 > 0){
          q0 = w->q0 - 1;
          wsetselect(w, q0, q0);
          wshow(w, q0);
      }
      return;
```

Uses wsetselect() 185 and wshow() 179b.

Right

⟨wkeyctl() *when mouse not opened, switch key cases* 190d⟩+≡ (188d) ◁190c 190e▷

```
  case Kright:
      if(w->q1 < w->nr){
          q1 = w->q1+1;
          wsetselect(w, q1, q1);
          wshow(w, q1);
      }
      return;
```

Uses wsetselect() 185 and wshow() 179b.

Line boundaries

^E and ^A move the cursor to the end or beginning of the current line, following the emacs convention familiar to Unix users. The beginning-of-line case cleverly reuses wbswidth()[191b] with the ^U mode (erase to line start) to compute the backward distance, but instead of deleting characters, it just repositions q0. This avoids duplicating the backward-scan logic. Note that "beginning of line" stops at the output point qh—the user cannot move before the prompt.

⟨wkeyctl() *when mouse not opened, switch key cases* 190e⟩+≡ (188d) ◁190d 191a▷

```
  case 0x05: /* ^E: end of line */
      q0 = w->q0;
      while(q0 < w->nr && w->r[q0] != '\n')
          q0++;
      wsetselect(w, q0, q0);
```

```
    wshow(w, w->q0);
    return;
```

Uses wsetselect() 185 and wshow() 179b.

⟨wkeyctl() *when mouse not opened, switch key cases* 191a⟩+≡ (188d) ◁190e

```
case 0x01: /* ^A: beginning of line */
    if(w->q0==0 || w->q0 == w->qh || w->r[w->q0 - 1]=='\n')
        return;
    nb = wbswidth(w, 0x15 /* ^U */);
    wsetselect(w, w->q0 - nb, w->q0 - nb);
    wshow(w, w->q0);
    return;
```

Uses wbswidth() 191b, wsetselect() 185, and wshow() 179b.

The function `wbswidth()` ("backspace width") computes how many characters to erase backward from `q0`. Despite its name, the parameter `c` selects the mode: `0x08` erases exactly one character, `0x15` erases back to the beginning of the line (or to `qh`), and `0x17` erases back to the previous word boundary. The word-erase (`^W`) logic uses a two-phase approach: first skip non-alphanumeric characters (the "skipping" phase), then eat alphanumeric characters until hitting a non-alphanumeric one. For example, erasing backward from the cursor in "`foo   bar|`" (where | is the cursor) first skips the spaces, then eats "bar", stopping at the space after "foo". All three modes stop at `qh` (the output point), which prevents the user from erasing text that the application wrote—only user input can be deleted.

⟨*function* wbswidth 191b⟩≡ (343b)

```
int
wbswidth(Window *w, Rune c)
{
    uint q, stop;
    Rune r;
    ⟨wbswidth() other locals 192c⟩

    ⟨wbswidth() return if erase character 192b⟩

    q = w->q0;
    stop = 0;
    if(q > w->qh)
        stop = w->qh;

    while(q > stop){
        r = w->r[q-1];
        if(r == '\n'){  /* eat at most one more character */
            if(q == w->q0) /* eat the newline */
                --q;
            break;
        }
        ⟨wbswidth() if c == 0x17 192d⟩
        --q;
    }
    return w->q0-q;
}
```

11.5.4 Special keys

The three erase keys—`^H` (backspace), `^U` (kill line), and `^W` (kill word)—all flow through the same handler. They call `wbswidth()`[191b] to compute how many characters to erase, then `wdelete()`[193a] to remove them from the rune buffer. The key distinction is that none of these can erase past the output point `qh`: the guard `w->q0==w->qh` prevents the user from deleting text that the application wrote. The flow for backspace is: `wbswidth`[191b] returns

1, q0 is set to w->q0 - 1, then wdelete(w, q0, q0+1) removes that one rune, which in turn calls frdelete()[314] to update the screen. The ^U and ^W cases differ only in how many characters wbswidth() returns.

Delete

⟨wkeyctl() *special key cases and no special mode* 192a⟩≡ (187a) 194b ▷
```
  case 0x08: /* ^H: erase character */
  case 0x15: /* ^U: erase line */
  case 0x17: /* ^W: erase word */
      if(w->q0==0 || w->q0==w->qh)
          return;
      nb = wbswidth(w, r);
      q1 = w->q0;
      q0 = q1-nb;
      ⟨wkeyctl() when erase keys, adjust q0 and nb if before org 192e⟩
      if(nb > 0){
          wdelete(w, q0, q0+nb);
          wsetselect(w, q0, q0);
      }
      return;
```
Uses wbswidth() 191b, wdelete() 193a, and wsetselect() 185.

⟨wbswidth() *return if erase character* 192b⟩≡ (191b)
```
  /* there is known to be at least one character to erase */
  if(c == 0x08) /* ^H: erase character */
      return 1;
```

⟨wbswidth() *other locals* 192c⟩≡ (191b)
```
  bool skipping = true;
  uint eq;
```

⟨wbswidth() *if* c == 0x17 192d⟩≡ (191b)
```
  if(c == 0x17){
      eq = isalnum(r);
      if(eq && skipping) /* found one; stop skipping */
          skipping = false;
      else if(!eq && !skipping)
          break;
  }
```
Uses isalnum() 284c.

⟨wkeyctl() *when erase keys, adjust q0 and nb if before org* 192e⟩≡ (192a)
```
  if(q0 < w->org){
      q0 = w->org;
      nb = q1-q0;
  }
```

The wdelete()[193a] function mirrors winsert()[177a]: it removes runes from the buffer by shifting the tail left with runemove90, then carefully adjusts all four cursor positions (q0, q1, qh, org). The adjustment logic must handle every case—the deleted range might be entirely before, entirely after, or overlapping each cursor. If the deletion affects visible text (between org and org+nchars), it calls frdelete()[314] and wfill()[184c] to update the display incrementally.

The wdelete() function is the dual of winsert(): it removes runes q0 through q1 from the buffer by shifting the tail left with runemove(), then adjusts all four cursor positions. The cursor adjustment is more complex than for insertion because the deleted range can partially overlap a cursor. For w->q0 and w->q1, if the cursor is inside the deleted range it collapses to q0; if after, it shifts left by n. For w->qh, overlap pulls it back to q0

since the application cannot reference deleted text. If the deletion is visible on screen, **frdelete()** handles the display update and **wfill()** fills any empty space at the bottom.

⟨*function* `wdelete` 193a⟩≡ (343b)
```
void
wdelete(Window *w, uint q0, uint q1)
{
    uint n, p0, p1;
    Frame *frm = &w->frm;

    n = q1-q0;
    ⟨wdelete() sanity check n 193b⟩
    runemove(w->r+q0, w->r+q1, w->nr-q1);
    w->nr -= n;

    ⟨wdelete() adjust cursors 193c⟩
}
```
Uses `runemove` 285a.

⟨`wdelete()` *sanity check n* 193b⟩≡ (193a)
```
if(n == 0)
    return;
```

⟨`wdelete()` *adjust cursors* 193c⟩≡ (193a)
```
if(q0 < w->q0)
    w->q0 -= min(n, w->q0-q0);
if(q0 < w->q1)
    w->q1 -= min(n, w->q1-q0);

if(q1 < w->qh)
    w->qh -= n;
else if(q0 < w->qh)
    w->qh = q0;

if(q1 <= w->org)
    w->org -= n;
else
   ⟨wdelete() when q1 > w->org, possibly update visible text 193d⟩
```
Uses `min()` 284a.

⟨`wdelete()` *when* `q1 > w->org`*, possibly update visible text* 193d⟩≡ (193c)
```
if(q0 < w->org + frm->nchars){
    p1 = q1 - w->org;
    if(p1 > frm->nchars)
        p1 = frm->nchars;
    if(q0 < w->org){
        w->org = q0;
        p0 = 0;
    }else
        p0 = q0 - w->org;

    frdelete(frm, p0, p1);
    wfill(w);
}
```
Uses `frdelete()` 314 and `wfill()` 184c.

Interrupt

When the user types the Delete key (rune `0x7F`), `rio` sends an interrupt note to the child process by writing `"interrupt"` to `Window.notefd`[98b]. Before sending the interrupt, `qh` is advanced to the end of the buffer (`w->qh = w->nr`). This is important for a clean user experience: after the interrupted command exits, the shell will print a new prompt. By moving `qh` to the end, the prompt will appear after all existing text rather than overwriting the middle of the buffer. The `wshow` then scrolls to make `qh` visible.

⟨`wkeyctl()` *locals* 194a⟩+≡ (73b) ◁188e

```
int nr;
Rune *rp;
int *notefd;
```

⟨`wkeyctl()` *special key cases and no special mode* 194b⟩+≡ (187a) ◁192a 246b▷

```
case 0x7F:  /* send interrupt */
    w->qh = w->nr;
    wshow(w, w->qh);
    notefd = emalloc(sizeof(int));
    *notefd = w->notefd;
    proccreate(interruptproc, notefd, 4096);
    return;
```

Uses `interruptproc()` 194c and `wshow()` 179b.

The interrupt must be sent from a separate `proc` (not just a thread) because the Plan 9 kernel holds `p->debug` during process death. If the window thread sent the interrupt directly and the child was simultaneously dying, the write to `notefd` would deadlock on `p->debug`. Using `proccreate` ensures the note is sent from an independent process context.

⟨*function* `interruptproc` 194c⟩≡ (343b)

```
/*
 * Need to do this in a separate proc because if process we're interrupting
 * is dying and trying to print tombstone, kernel is blocked holding p->debug lock.
 */
void
interruptproc(void *v)
{
    int *notefd;

    notefd = v;
    write(*notefd, "interrupt", 9);
    free(notefd);
}
```

Tab

Tab stops are configurable via the `tabstop` environment variable (defaulting to 4 if not set). The value is stored in pixels as `frm.maxtab` = `tabstop` × the width of a "0" character. This means tab alignment adapts automatically to the font—a proportional font with narrow characters gets narrower tab stops. Tab alignment is computed at drawing time by `_frnewwid0`[312b]: it rounds up the current x position to the next multiple of `maxtab` (relative to the left margin), ensuring a minimum width of one space (`minwid`) so that tabs always produce visible whitespace.

⟨`Frame` *text fields* 194d⟩+≡ (288a) ◁184a 288b▷

```
ushort maxtab;  /* max size of tab, in pixels */
```

⟨`frinit()` *initialize other fields* 194e⟩+≡ (288d) ◁184b 289h▷

```
f->maxtab = 8 * stringwidth(ft, "0");
```

⟨wmk() *textual window settings, extra frame settings* 195a⟩+≡ (174e) ◁181d

```
w->frm.maxtab = maxtab * stringwidth(font, "0");
```

Uses maxtab 195c.

⟨wresize() *textual window updates, extra frame settings* 195b⟩≡ (205a)

```
w->frm.maxtab = maxtab * stringwidth(w->frm.font, "0");
```

Uses maxtab 195c.

⟨*global* maxtab 195c⟩≡ (338a)

```
int  maxtab = 0;
```

Uses maxtab 195c.

⟨main() *locals* 195d⟩+≡ (58) ◁96i 226a▷

```
char *s;
```

⟨main() *set some globals* 195e⟩+≡ (58) ◁96j 229c▷

```
s = getenv("tabstop");
if(s != nil)
    maxtab = strtol(s, nil, 0);
if(maxtab == 0)
    maxtab = 4;
free(s);
```

11.6 Application output events

When the application writes to /dev/cons, the data arrives at the winctl()[71b] thread through the Window.conswrite channel. The thread inserts the new runes at the output point (qh), advances qh, and updates the display. If automatic scrolling is enabled, wshow()[179b] scrolls the viewport to keep qh visible—the user sees the output appear in real time, like a traditional terminal. Otherwise, new output accumulates off-screen; the scrollbar thumb shrinks to indicate there is unread text below, and the WCwrite alternative is temporarily disabled to avoid unbounded buffering of invisible text (the application blocks on its write until the user scrolls to catch up). This flow-control mechanism is unique to rio—most terminal emulators let applications write as fast as they want, consuming arbitrary amounts of memory. In rio, the back-pressure ensures memory usage stays bounded by the HiWater limit even under heavy output.

11.6.1 Writing /mnt/wsys/cons: window thread side

The *file-server side* (Virtual Devices chapter) showed how xfidwrite()[137b] converts the client's bytes to runes and sends them through the channel-of-channels protocol. This is the *window-thread side*: winctl()[71b] as consumer—it receives the runes, inserts them at the output point qh via winsert()[177a], advances qh by the number of runes inserted, and updates the display. Note that application output always inserts at qh, which may be before q0 (the user's cursor). This means the user can be scrolling through old output or editing their command line while the application simultaneously writes new output at a different position in the buffer. The cursor positions are adjusted correctly by winsert()'s cursor adjustment logic.

Producer

Consumer

⟨Wxxx *cases* 195f⟩+≡ (71a) ◁165b 213f▷

```
WCwrite,
```

⟨winctl() *other locals* 195g⟩+≡ (71b) ◁166b 196e▷

```
Conswritemesg cwm;
```

⟨winctl() *channels creation* 196a⟩+≡ (71b) ◁165d 213d▷
```
cwm.cw = chancreate(sizeof(Stringpair), 0);
```

⟨winctl() *Wctl case, free channels if wctlmesg is Excited* 196b⟩+≡ (74g) ◁165e 213e▷
```
chanfree(cwm.cw);
```

⟨winctl() *alts setup* 196c⟩+≡ (71b) ◁165f 213g▷
```
alts[WCwrite].c = w->conswrite;
alts[WCwrite].v = &cwm;
alts[WCwrite].op = CHANSND;
```
Uses `WCwrite-30` 195f.

This is the flow-control gate for application output. When auto-scroll is off and the output point `qh` has moved past the visible area (`qh > org + nchars`), the `WCwrite` alternative is disabled (`CHANNOP`). This causes the application's `write("/dev/cons")` to block—it cannot produce more output until the user scrolls to catch up. This prevents unbounded memory growth from a program that produces output faster than the user reads it. With auto-scroll on, or in graphical mode, the gate is always open.

⟨winctl() *alts adjustments* 196d⟩+≡ (71b) ◁165g 213h▷
```
if(!w->scrolling && !w->mouseopen && w->qh >  w->org + w->frm.nchars)
    alts[WCwrite].op = CHANNOP;
else
    // scrolling || mouseopen || w->qh <= w->org + w->frm.nchars
    alts[WCwrite].op = CHANSND;
```
Uses `WCwrite-30` 195f.

⟨winctl() *other locals* 196e⟩+≡ (71b) ◁195g 197a▷
```
Rune *rp;
int nr;
```

The `WCwrite` handler is the consumer side: it receives runes from the application (e.g., the shell printing its prompt), processes any backspace characters, then inserts at `qh` and advances `qh`. The key flow-control mechanism is in the `alts adjustments`: when auto-scrolling is off and `qh` has moved past the visible area, `WCwrite` is disabled (`CHANNOP`) so the application blocks on its write—this prevents unbounded buffering of invisible output.

⟨winctl() *event loop cases* 196f⟩+≡ (71b) ◁166a 214a▷
```
case WCwrite:
    recv(cwm.cw, &pair);
    rp = pair.s;
    nr = pair.ns;

    ⟨winctl() when WCwrite, if runes contains backspace 197b⟩

    w->qh = winsert(w, rp, nr, w->qh) + nr;
    ⟨winctl() when WCwrite, if scrolling or mouseopen 196g⟩
    wsetselect(w, w->q0, w->q1);
    wscrdraw(w);

    free(rp);
    break;
```
Uses `WCwrite-30` 195f, `winsert()` 177a, `wscrdraw()` 180b, and `wsetselect()` 185.

⟨winctl() *when WCwrite, if scrolling or mouseopen* 196g⟩≡ (196f)
```
if(w->scrolling || w->mouseopen)
    wshow(w, w->qh);
```
Uses `wshow()` 179b.

⟨winctl() *other locals* 197a⟩+≡ (71b) ◁196e 213c▷
```
  Rune *bp, *tp, *up;
  int initial;
  uint qh;
```

When application output contains backspace characters (`\b`), they must be interpreted before insertion—the output should overwrite previous characters, not display literal backspaces. This is how programs like `cat -v` or progress bars that use `\r` work. The code scans for the first `\b`, then builds a filtered copy in `tp` where each backspace cancels the previous character (by decrementing `up`). If backspaces at the start of the output reach past the beginning of the batch (counted by `initial`), they must erase characters already in the buffer. These are deleted from the window via `wdelete()`[193a] before the filtered text is inserted. For example, if the application writes "abc`\b\b`XY", the buffer receives "aXY" (the two backspaces erase "c" and "b", then "XY" is inserted).

⟨winctl() *when WCwrite, if runes contains backspace* 197b⟩≡ (196f)
```
  bp = rp;
  for(i=0; i<nr; i++) {
      if(*bp++ == '\b'){
          --bp;
          initial = 0;
          tp = runemalloc(nr);
          runemove(tp, rp, i);
          up = tp+i;
          for(; i<nr; i++){
              *up = *bp++;
              if(*up == '\b')
                  if(up == tp)
                      initial++;
                  else
                      --up;
              else
                  up++;
          }
          if(initial){
              if(initial > w->qh)
                  initial = w->qh;
              qh = w->qh-initial;
              wdelete(w, qh, qh+initial);
              w->qh = qh;
          }
          free(rp);
          rp = tp;
          nr = up-tp;
          rp[nr] = 0;
          break;
      }
  }
```
Uses `runemalloc` 284d, `runemove` 285a, and `wdelete()` 193a.

11.7 Mouse events

In a textual window (when `Window.mouseopen`[51f] is false), mouse clicks are handled locally by `rio` rather than forwarded to the application. The three buttons have distinct roles:

- **Left click (button 1):** Text selection. Click to place the cursor, drag to select a range, double-click for word selection. Clicking in the scrollbar scrolls up.

- **Middle click (button 2):** Edit menu with Cut, Paste, Snarf, Plumb, Send, and Scroll. Clicking in the scrollbar jumps to an absolute position.
- **Right click (button 3):** Window management menu (handled by `mousethread`[66a], not by the window). Clicking in the scrollbar scrolls down.

This three-button protocol gives the terminal rich editing capabilities—cut/paste, word selection, scrollback—without requiring any support from the application running inside it. The application only sees text on `/dev/cons`; it has no idea the user just selected and pasted something with the mouse.

11.7.1 Middle click menu

⟨`mousethread()` *middle click under certain conditions* 198a⟩≡ (70c)

```
button2menu(winput);
```

Uses `button2menu()` 198b.

The `button2menu` function displays the edit menu using `menuhit` and dispatches the selected action. The menu offers six operations:

- **Cut:** Copy the selection to the snarf buffer and delete it.
- **Paste:** Insert the snarf buffer contents at `q0`.
- **Snarf:** Copy the selection to the snarf buffer without deleting (equivalent to "copy" in other systems).
- **Plumb:** Send the selected text to the plumber service, which can open files, URLs, or navigate to line numbers based on pattern rules.
- **Send:** Inject the selected text as if the user had typed it—useful for resending a previous command by selecting it and clicking Send.
- **Scroll:** Toggle automatic scrolling mode (see below).

The window is reference-counted (`incref`/`wclose`) to prevent it from being freed while the menu is displayed. After the menu action, a `Wakeup` message is sent to the window's control thread to ensure it reprocesses any pending events.

⟨*function* `button2menu` 198b⟩≡ (343b)

```
void
button2menu(Window *w)
{
    ⟨button2menu() return if window was deleted 198c⟩

    incref(w);
    ⟨button2menu() menu2str adjustments for scrolling 200b⟩
    switch(menuhit(2, mousectl, &menu2, desktop)){
    ⟨button2menu() cases 200c⟩
    }
    wclose(w); // decref

    wsendctlmesg(w, Wakeup, ZR, nil);
    flushimage(display, true);
}
```

Uses `Wakeup` 272c, `desktop` 48d, `menu2` 199a, `mousectl` 48a, `wclose()` 109b, and `wsendctlmesg()` 91c.

⟨`button2menu()` *return if window was deleted* 198c⟩≡ (198b)

```
if(w->deleted)
    return;
```

⟨*global* menu2 199a⟩≡ (343b)
```
Menu menu2 = { .item = menu2str };
```
Uses menu2str 199b.

⟨*global* menu2str 199b⟩≡ (343b)
```
char*  menu2str[] = {
 [Cut]  "cut",
 [Paste] "paste",
 [Snarf] "snarf",
 [Plumb] "plumb",
 [Send]  "send",
 [Scroll] "scroll",
 nil
};
```

⟨*enum* _anon_ (rio/rio.c)*2* 199c⟩≡ (343b)
```
enum
{
    Cut,
    Paste,
    Snarf,
    Plumb,
    Send,
    Scroll,
};
```

11.7.2 Other clicks

When the mouse is not inside the scrollbar, `wmousectl()`[199e] dispatches based on which button was pressed. Left-click (button 1) triggers text selection via `wselect()`[253]; middle and right clicks were already intercepted by `mousethread()`[66a] for the menu and window management, so they do not normally reach here. Mouse buttons 4 and 5 (the scroll wheel) are translated into synthetic `Kscrolloneup` and `Kscrollonedown` key events by calling `wkeyctl()`[73b]—an elegant reuse that makes the scroll wheel behave identically to the keyboard scroll keys, without any special-case code in the scrolling logic.

⟨`winctl()` *WMouse case if not mouseopen* 199d⟩≡ (74c)
```
wmousectl(w);
```
Uses wmousectl() 199e.

⟨*function* wmousectl 199e⟩≡ (343b)
```
void
wmousectl(Window *w)
{
    int but;

    if(w->mc.m.buttons == 1)
        but = 1;
    else if(w->mc.m.buttons == 2)
        but = 2;
    else if(w->mc.m.buttons == 4)
        but = 3;
    else{
        if(w->mc.m.buttons == 8)
            wkeyctl(w, Kscrolloneup);
        if(w->mc.m.buttons == 16)
            wkeyctl(w, Kscrollonedown);
        return;
    }
```

```
    incref(w);  /* hold up window while we track */
    ⟨wmousectl() goto Return if window was deleted 200a⟩

    ⟨wmousectl() if pt in scrollbar 201e⟩
    if(but == 1)
        wselect(w);

    /* else all is handled by main process */
   Return:
    wclose(w);
}
```

Uses Kscrollonedown 203a, Kscrolloneup 203a, wclose() 109b, wkeyctl() 73b, and wselect() 253.

⟨wmousectl() *goto Return if window was deleted* 200a⟩≡ (199e)

```
if(w->deleted)
    goto Return;
```

11.8 Automatic scrolling mode

The automatic scrolling mode controls a fundamental usability trade-off: should the terminal follow the output (like a traditional terminal), or should the user be able to read at their own pace (like a pager)? When `Window.scrolling`[52f] is true, the window automatically keeps the output point visible—each time the application writes, `wshow(w, w->qh)` scrolls the viewport to follow. This is useful when watching a build log or a long-running command. When false (the default), the user can scroll freely and new output accumulates off-screen without disturbing the view. The scrollbar thumb shrinks as more text piles up below, giving a visual cue that there is unread output. This mode is essential for reading a man page or examining earlier output while a command is still running. The middle-click menu toggles this flag with `w->scrolling ^= 1` and, if scrolling was just enabled, immediately jumps to the end of the buffer. The menu text dynamically switches between "scroll" and "noscroll" to reflect the current state.

⟨button2menu() *menu2str adjustments for scrolling* 200b⟩≡ (198b)

```
if(w->scrolling)
    menu2str[Scroll] = "noscroll";
else
    menu2str[Scroll] = "scroll";
```

Uses Scroll-41 199c and menu2str 199b.

⟨button2menu() *cases* 200c⟩≡ (198b) 230b ▷

```
case Scroll:
    if(w->scrolling ^= 1)
        wshow(w, w->nr);
    break;
```

Uses Scroll-41 199c and wshow() 179b.

11.9 Scroll bar interaction

The scrollbar is not just a position indicator—the user can click on it to scroll. The three-button protocol follows the Plan 9 convention:

- **Button 1 (left):** Scroll up. The distance depends on where in the scrollbar the user clicks—clicking near the top scrolls a little, clicking near the bottom scrolls a lot. More precisely, the Y offset within the scrollbar determines how many lines to go back.

- **Button 2 (middle):** Jump to absolute position. The Y coordinate maps proportionally to a character offset in the buffer, so clicking at 50% of the scrollbar height jumps to 50% of the text.
- **Button 3 (right):** Scroll down. The line at the click position becomes the new top of the viewport.

The scrollbar interaction is separate from text selection: the `Frame.scroll` callback is set to `wscroll()`[202] during window creation. When `frselect()`[300] detects the mouse dragged outside the frame (above or below), it calls this callback to scroll the text while maintaining the selection—enabling "drag to select past the viewport edge".

⟨`Frame` *scroll* 201a⟩≡ (288a)
```
  void (*scroll)(Frame*, int); /* scroll function provided by application */
```

⟨`mousethread()` *locals* 201b⟩+≡ (66a) ◁90d
```
  bool scrolling = false;
```

The `scrolling` local in `mousethread`[66a] (not to be confused with `Window.scrolling` which controls auto-scroll) is a sticky flag: once the user clicks inside the scrollbar, subsequent mouse events continue to be treated as scroll operations even if the mouse drifts outside the scrollbar rectangle. It resets only when all buttons are released.

⟨`mousethread()` *set scrolling* 201c⟩≡ (66e)
```
  if(winput->mouseopen)
      scrolling = false;
  else
    if(scrolling)
      scrolling = mouse->buttons;
    else
      scrolling = mouse->buttons && ptinrect(xy, winput->scrollr);
```
Uses `mouse` 48c.

⟨`mousethread()` *goto Sending if scroll buttons* 201d⟩≡ (66e)
```
  /* the up and down scroll buttons are not subject to the usual rules */
  if((mouse->buttons&(8|16)) && !winput->mouseopen)
      goto Sending;
```
Uses `mouse` 48c.

⟨`wmousectl()` *if pt in scrollbar* 201e⟩≡ (199e)
```
  if(ptinrect(w->mc.m.xy, w->scrollr)){
      if(but)
          wscroll(w, but);
      goto Return;
  }
```
Uses `wscroll()` 202.

The `wscroll()` function implements the classic Plan 9 three-button scrollbar protocol in a continuous tracking loop. While the button is held, the mouse is constrained to the scrollbar's X center (using `wmovemouse()`[116c]) so it does not drift into the text area. The Y position determines the scroll amount differently for each button: For **button 1** (scroll up), the Y offset within the scrollbar is converted to a line count: clicking near the top of the scrollbar scrolls one line, clicking near the bottom scrolls many lines. This count is passed to `wbacknl()`[182d] to find the new `org`. For **button 2** (absolute jump), the Y coordinate maps proportionally to a character offset: `p0 = nr * (y - min) / h`. The special-case arithmetic for `w->nr > 1024*1024` (shifting by 10 bits) avoids integer overflow in the `nr * y` multiplication for very large buffers. For **button 3** (scroll down), `frcharofpt()`[297c] is used to find the character at the click's Y position in the visible frame, and that character becomes the new top of the viewport. All three modes use a debounce mechanism: a 200ms `sleep()` on the first click prevents

accidental scrolling (e.g., when the user just wanted a single scroll step), followed by repeated scrolling at 100ms intervals via `wscrsleep()`[203c] for as long as the button is held.

⟨*function* `wscroll` 202⟩≡ (348)

```
void
wscroll(Window *w, int but)
{
    uint p0, oldp0;
    Rectangle s;
    int x, y, my, h, first;

    s = insetrect(w->scrollr, 1);
    h = s.max.y-s.min.y;
    x = (s.min.x+s.max.x)/2;
    oldp0 = ~0;
    first = true;
    do{
        flushimage(display, 1);
        if(w->mc.m.xy.x<s.min.x || s.max.x<=w->mc.m.xy.x){
            readmouse(&w->mc);
        }else{
            my = w->mc.m.xy.y;
            if(my < s.min.y)
                my = s.min.y;
            if(my >= s.max.y)
                my = s.max.y;
            if(!eqpt(w->mc.m.xy, Pt(x, my))){
                wmovemouse(w, Pt(x, my));
                readmouse(&w->mc);  /* absorb event generated by moveto() */
            }
            if(but == 2){
                y = my;
                if(y > s.max.y-2)
                    y = s.max.y-2;
                if(w->nr > 1024*1024)
                    p0 = ((w->nr>>10)*(y-s.min.y)/h)<<10; // >>
                else
                    p0 = w->nr*(y-s.min.y)/h;
                if(oldp0 != p0)
                    wsetorigin(w, p0, false);
                oldp0 = p0;
                readmouse(&w->mc);
                continue;
            }
            if(but == 1)
                p0 = wbacknl(w, w->org, (my-s.min.y)/w->frm.font->height);
            else
                p0 = w->org + frcharofpt(&w->frm, Pt(s.max.x, my));
            if(oldp0 != p0)
                wsetorigin(w, p0, true);
            oldp0 = p0;
            /* debounce */
            if(first){
                flushimage(display, 1);
                sleep(200);
                nbrecv(w->mc.c, &w->mc.m);
                first = false;
            }
            wscrsleep(w, 100);
        }
```

```
    }while(w->mc.m.buttons & (1<<(but-1))); // >>
    while(w->mc.m.buttons)
        readmouse(&w->mc);
}
```

Uses frcharofpt() 297c, wbacknl() 182d, wmovemouse() 116c, wscrsleep() 203c, and wsetorigin() 183.

⟨*enum* _anon_ (rio/dat.h) *2* 203a⟩≡ (333b)

```
enum
{
    Kscrolloneup = KF|0x20,
    Kscrollonedown = KF|0x21,
};
```

⟨*function* freescrtemps 203b⟩≡ (348)

```
void
freescrtemps(void)
{
    freeimage(scrtmp);
    scrtmp = nil;
}
```

Uses scrtmp-50 181g.

The wscrsleep() helper implements the auto-repeat behavior for scrollbar dragging. It sets up a two-channel alt: a timer channel and the mouse channel. If the timer fires first (the full dt milliseconds elapsed without mouse activity), the function returns and wscroll() scrolls another increment. If a mouse event arrives first, it checks whether the mouse moved significantly (more than 2 pixels vertically) or changed buttons; if so, it returns early so wscroll() can react to the new position. Small jitter is ignored. This gives the classic "press and hold to scroll" behavior: an initial 200ms debounce delay (in wscroll) prevents accidental scrolling, then repeated 100ms intervals provide smooth continuous scrolling as long as the button is held.

⟨*function* wscrsleep 203c⟩≡ (348)

```
void
wscrsleep(Window *w, uint dt)
{
    Timer *timer;
    int y, b;
    static Alt alts[3];

    timer = timerstart(dt);
    y = w->mc.m.xy.y;
    b = w->mc.m.buttons;
    alts[0].c = timer->c;
    alts[0].v = nil;
    alts[0].op = CHANRCV;
    alts[1].c = w->mc.c;
    alts[1].v = &w->mc.m;
    alts[1].op = CHANRCV;
    alts[2].op = CHANEND;
    for(;;)
        switch(alt(alts)){
        case 0:
            timerstop(timer);
            return;
        case 1:
            if(abs(w->mc.m.xy.y-y)>2 || w->mc.m.buttons!=b){
                timercancel(timer);
                return;
            }
            break;
```

```
        }
}
```

Uses timercancel() 223d, timerstart() 223b, and timerstop() 223c.

The `framescroll` callback is invoked by the frame library's `frselect()` when the user drags a selection past the top or bottom of the frame. It bridges the generic frame library (which knows nothing about windows or rune buffers) to the window-specific `wframescroll()`[204b]. The `wframescroll()` function shifts `org` by `dl` lines and extends the selection to track the drag. A negative `dl` scrolls up, positive scrolls down. The selection is always anchored at `selectq` (the initial click point, saved by `wselect()`[253]) and stretched to the new viewport edge. This means the user can drag above the frame to select text that was previously off-screen—the viewport scrolls to reveal it and the selection grows to include it. When `dl == 0`, the function just sleeps briefly via `wscrsleep`, providing the auto-repeat delay when the mouse hovers just above or below the frame boundary.

⟨*function* framescroll 204a⟩≡ (343b)

```
/*
 * called from frame library
 */
void
framescroll(Frame *f, int dl)
{
    if(f != &selectwin->frm)
        error("frameselect not right frame");
    wframescroll(selectwin, dl);
}
```

Uses error() 282c, selectwin-44 252d, and wframescroll() 204b.

⟨*function* wframescroll 204b⟩≡ (343b)

```
void
wframescroll(Window *w, int dl)
{
    uint q0;
    Frame *frm = &w->frm;

    if(dl == 0){
        wscrsleep(w, 100);
        return;
    }
    if(dl < 0){
        q0 = wbacknl(w, w->org, -dl);
        if(selectq > w->org + frm->p0)
            wsetselect(w, w->org + frm->p0, selectq);
        else
            wsetselect(w, selectq, w->org + frm->p0);
    }else{
        if(w->org + frm->nchars == w->nr)
            return;
        q0 = w->org + frcharofpt(frm, Pt(frm->r.min.x, frm->r.min.y + dl * frm->font->height));
        if(selectq >= w->org + frm->p1)
            wsetselect(w, w->org + frm->p1, selectq);
        else
            wsetselect(w, selectq, w->org + frm->p1);
    }
    wsetorigin(w, q0, true);
}
```

Uses frcharofpt() 297c, selectq-45 252e, wbacknl() 182d, wscrsleep() 203c, wsetorigin() 183, and wsetselect() 185.

11.10 Resize

When a window is resized, the frame must be re-initialized with the new rectangle. Two cases are handled:

- **Pure move** (size unchanged): the fast path. `frsetrects()`[289a] just updates the frame's coordinates without re-laying out the text. The box array and all cached widths remain valid.
- **True resize**: the full path. The frame is cleared (`frclear()`[289d]), re-initialized with the new rectangle (`frinit()`[288d]), the scrollbar rectangle is recomputed, the background is repainted, visible text is refilled with `wfill()`[184c], and the selection and scrollbar are redrawn.

This code is essentially a re-execution of the `wmk()`[94c] textual window setup, which is why the two look so similar. The duplication is unfortunate but hard to factor because the resize path must handle the existing text content while the creation path starts from an empty buffer.

⟨`wresize()` *textual window updates* 205a⟩≡ (114c)

```
⟨wresize() textual window updates, reset lastsr 181e⟩
r = insetrect(i->r, Selborder+1);
w->scrollr = r;
w->scrollr.max.x = r.min.x+Scrollwid;

r.min.x += Scrollwid+Scrollgap;

if(move)
    frsetrects(&w->frm, r, w->i);
else{
    frclear(&w->frm, false);
    frinit(&w->frm, r, w->frm.font, w->i, cols);
    wsetcols(w);
    ⟨wresize() textual window updates, extra frame settings 195b⟩
    r = insetrect(w->i->r, Selborder);
    draw(w->i, r, cols[BACK], nil, w->frm.entire.min);

    wfill(w);
    wsetselect(w, w->q0, w->q1);
    wscrdraw(w);
}
```

Uses `BACK` 289f, `Scrollgap` 174d, `Scrollwid` 174a, `Selborder` 67, `cols-18` 289g, `frclear()` 289d, `frinit()` 288d, `frsetrects()` 289a, `wfill()` 184c, `wscrdraw()` 180b, `wsetcols()` 175c, and `wsetselect()` 185.

11.11 /mnt/wsys/text

The `/mnt/wsys/text` file exposes the entire contents of a window's rune buffer as a read-only byte stream. Reading it returns a UTF-8 encoding of `w->r[0..w->nr]`. This is useful for programs that want to inspect what a terminal window currently contains—for example, a script could read its own window's text to implement command history or auto-completion. The implementation is trivially simple: `wcontents()`[206b] calls `runetobyte()` to convert the entire rune array to UTF-8, and the generic `Text:` label (shared with `Qwinname`) handles offset and count arithmetic before responding to the 9P read request. Note that the conversion allocates a fresh copy each time—there is no caching.

⟨`Qxxx` *other cases* 205b⟩+≡ (122d) ◁169b 208a▷

```
Qtext,
```

⟨*dirtab array elements* 205c⟩+≡ (123b) ◁169c 208b▷

```
{ "text",  QTFILE, Qtext,  0400 },
```

Uses `Qtext` 205b.

⟨xfidread() *cases* 206a⟩+≡ (136b) ◁170b 208c▷

```
case Qtext:
    t = wcontents(w, &n);
    goto Text;

Text:
    if(off > n){
        off = n;
        cnt = 0;
    }
    if(off+cnt > n)
        cnt = n-off;

    fc.data = t + off;
    fc.count = cnt;
    filsysrespond(x->fs, x, &fc, nil);
    free(t);
    break;
```

Uses `Qtext` 205b, `filsysrespond()` 124, and `wcontents()` 206b.

⟨*function* wcontents 206b⟩≡ (343b)

```
char*
wcontents(Window *w, int *ip)
{
    return runetobyte(w->r, w->nr, ip);
}
```

Uses `runetobyte()` 285d.

Chapter 12

Windowing System Files

The previous chapters covered the virtual device files that emulate hardware—/mnt/wsys/cons for the keyboard, /mnt/wsys/mouse for the mouse, and /mnt/wsys/winname for graphics. Those files make rio transparent: programs running in a window do not even know they are windowed. This chapter covers the remaining files under /mnt/wsys/, which serve a different purpose. They are the "reflective" part of rio, analogous to how /proc lets programs inspect and control kernel processes (see the KERNEL book [Pad14] and DEBUGGER book [Pad16b]). Through these files, programs can discover their own window identity (/mnt/wsys/winid, /mnt/wsys/label), query the screen (/mnt/wsys/screen), access other windows (/mnt/wsys/wsys/), and control windows programmatically (/mnt/wsys/wctl).

12.1 Overview

The full shape of the per-window filesystem that rio serves is the union of all the dirtab[123b] entries scattered across the book, which makes it hard to picture. Here is a consolidated view of what a client sees when it mounts /mnt/wsys/ (files introduced in earlier chapters are marked [done], files introduced in this chapter are marked [here]):

```
/mnt/wsys/                          (Qdir,     per-window root)
|
+-- cons           [done]  (Qcons)    text console read/write
+-- consctl        [done]  (Qconsctl) raw/cooked mode toggle
+-- mouse          [done]  (Qmouse)   mouse events (+ resize piggyback)
+-- cursor         [done]  (Qcursor)  custom cursor image
+-- winname        [done]  (Qwinname) current window-image name
+-- window         [done]  (Qwindow)  raw pixels of this window
|
+-- winid          [here]  (Qwinid)   numeric id (text)
+-- label          [here]  (Qlabel)   human label (rw)
+-- screen         [here]  (Qscreen)  full desktop image (shared)
+-- wctl           [here]  (Qwctl)    window-control commands
+-- text           [done]  (Qtext)    textual-window buffer
+-- snarf          [adv ]  (Qsnarf)   clipboard
|
+-- wsys/                   (Qwsys,   directory of window dirs)
    +-- 1/         -+
    +-- 2/          |  same tree recursively,
    +-- 3/          |  one subdir per existing window
    +-- ...        -+
```

The whole tree lives inside one process (rio), and the contents of a given file depend on which window the fid is attached to: opening /mnt/wsys/cons in window 2 is not the same file as opening it in window 3, even though the path string is identical. The Qid.path trick from Section 8.2 (QID(winid, qxxx)) is what lets the server tell them apart. The /mnt/wsys/wsys/<N>/ subtree is the one escape hatch: by walking through wsys/ to a numeric name, a client can reach any other window's files and drive it remotely. This is the mechanism behind window-management utilities like the statusbar and winwatch programs (see Appendix E).

Let us walk through these windowing-system files one at a time.

12.2 /mnt/wsys/winid

Reading /mnt/wsys/winid returns the numeric window ID as a string. A program uses this to discover its own window identity, typically to construct paths like /mnt/wsys/wsys/<id>/wctl when it needs to control a specific window (including its own). The implementation is straightforward—a single sprint():

⟨Qxxx *other cases* 208a⟩+≡ (122d) ◁205b 208d▷
```
  Qwinid,
```

⟨*dirtab array elements* 208b⟩+≡ (123b) ◁205c 208e▷
```
  { "winid",  QTFILE, Qwinid,  0400 },
```
Uses Qwinid 208a.

⟨xfidread() *cases* 208c⟩+≡ (136b) ◁206a 208f▷
```
  case Qwinid:
      n = sprint(buf, "%11d ", w->id);
      t = estrdup(buf);
      goto Text;
```
Uses Qwinid 208a.

12.3 /mnt/wsys/label

The label is a readable and writable string that identifies a window. Since rio windows have no title bar, the label is not displayed on screen—it appears only in the system menu's list of hidden windows (see Section 7.10), and it can be read by tools that query window state via wctl.

By default, initdraw() sets the label to the string passed as its third argument (e.g., "Hello Rio" in hellorio.c). A program can change its label at any time by writing to /mnt/wsys/label. The read path returns the label with proper offset handling; the write path replaces the entire label (writes at non-zero offsets are rejected to keep the implementation simple):

⟨Qxxx *other cases* 208d⟩+≡ (122d) ◁208a 209b▷
```
  Qlabel,
```

⟨*dirtab array elements* 208e⟩+≡ (123b) ◁208b 209c▷
```
  { "label",  QTFILE, Qlabel,  0600 },
```
Uses Qlabel 208d.

⟨xfidread() *cases* 208f⟩+≡ (136b) ◁208c 209d▷
```
  case Qlabel:
      n = strlen(w->label);
      if(off > n)
          off = n;
      if(off+cnt > n)
          cnt = n - off;

      fc.data = w->label + off;
```

```
    fc.count = cnt;
    filsysrespond(x->fs, x, &fc, nil);
    break;
```
Uses `Qlabel` 208d and `filsysrespond()` 124.

⟨`xfidwrite()` *cases* 209a⟩+≡ (137b) ◁153g 215a▷
```
  case Qlabel:
    if(off != 0){
        filsysrespond(x->fs, x, &fc, "non-zero offset writing label");
        return;
    }
    free(w->label);
    w->label = emalloc(cnt+1);
    memmove(w->label, req->data, cnt);
    w->label[cnt] = '\0';
    break;
```
Uses `Qlabel` 208d and `filsysrespond()` 124.

12.4 /mnt/wsys/screen

Unlike the other `/mnt/wsys/` files, `/mnt/wsys/screen` is not per-window: every window sees the same global screen. Reading it returns the image ID and rectangle of the entire display in the same format (see the GRAPHICS book [Pad16c]) as `/mnt/wsys/window`. This lets programs discover the full desktop geometry, which is useful for positioning new windows or for utilities like `lens` (Section E.1) that operate on the whole screen.

⟨`Qxxx` *other cases* 209b⟩+≡ (122d) ◁208d 209e▷
```
  Qscreen,
```

⟨*dirtab array elements* 209c⟩+≡ (123b) ◁208e 210a▷
```
  { "screen",  QTFILE, Qscreen, 0400 },
```
Uses `Qscreen` 209b.

⟨`xfidread()` *cases* 209d⟩+≡ (136b) ◁208f 214d▷
```
  case Qscreen:
    i = display->image;
    if(i == nil){
        filsysrespond(x->fs, x, &fc, "no top-level screen");
        break;
    }
    r = i->r;
    goto caseImage;
```
Uses `Qscreen` 209b and `filsysrespond()` 124.

The implementation delegates to the same `caseImage` code path used by `/mnt/wsys/window`, passing `display->image` instead of the window's own image:

12.5 /mnt/wsys/wsys/<windid>/

`/mnt/wsys/wsys/` is to windows what `/proc/` is to processes: a directory whose entries are numbered subdirectories, one per window. Listing `/mnt/wsys/wsys/` returns the IDs of all existing windows (sorted numerically). Walking into `/mnt/wsys/wsys/<winid>/` gives access to the same virtual device files (`cons`, `mouse`, `wctl`, etc.) as `/mnt/wsys/`, but for window `winid` instead of the caller's own window. This is how a script can create, resize, or delete another window: by writing commands to its `wctl` file (Section 12.6). For example, the `statusbar` program in the appendix (Section E.2) enumerates all windows, reads their labels, and displays a window list.

⟨`Qxxx` *other cases* 209e⟩+≡ (122d) ◁209b 210c▷
```
  Qwsys,  /* directory of window directories */
```

⟨*dirtab array elements* 210a⟩+≡ (123b) ◁209c 211d▷
```
{ "wsys",  QTDIR, Qwsys,  0500|DMDIR },
```
Uses `Qwsys` 209e.

The implementation is interesting because it uses `filsyswalk()`[129] to detect numeric path components and switch the fid's window context. When a walk reaches `Qwsys` and the next component is a number, the code looks up the window by ID, updates `f->w` to point to the target window, and continues the walk from `Qwsysdir`—at that point, the fid behaves as though the client had mounted that window's own `/mnt/wsys/`:

⟨`filsyswalk()` *if Qwsys, then goto Accept* 210b⟩≡ (129)
```
if(qid.path == Qwsys){
    /* is it a numeric name? */
    if(!numeric(x->req.wname[i]))
        break;
    /* yes: it's a directory */
    id = atoi(x->req.wname[i]);
    qlock(&all);
    w = wlookid(id);
    if(w == nil){
        qunlock(&all);
        break;
    }
    path = Qwsysdir;
    type = QTDIR;
    qunlock(&all);
    incref(w);
    sendp(winclosechan, f->w);
    f->w = w;
    dir = dirtab;
    goto Accept;
}
```
Uses `Qwsys` 209e, `Qwsysdir` 210c, `all` 128b, `dirtab` 123b, `numeric()` 210e, `winclosechan` 108c, and `wlookid()` 128a.

⟨`Qxxx` *other cases* 210c⟩+≡ (122d) ◁209e 211c▷
```
Qwsysdir,  /* window directory, child of wsys */
```

⟨`filsyswalk()` *when in dotdot, if Qwsysdir adjust path* 210d⟩≡ (130e)
```
if(FILE(qid) == Qwsysdir)
    path = Qwsys;
```
Uses `FILE` 122c, `Qwsys` 209e, and `Qwsysdir` 210c.

⟨*function* `numeric` 210e⟩≡ (346c)
```
static
int
numeric(char *s)
{
    for(; *s!='\0'; s++)
        if(*s<'0' || '9'<*s)
            return 0;
    return 1;
}
```

Listing `/mnt/wsys/wsys/` itself means emitting one directory entry per window. The `Qwsys` read does exactly that: under the `all`[128b] lock it snapshots the current window ids, sorts them, and `dostat()`[138b]s each as a subdirectory.

⟨`filsysread()` *other locals* 210f⟩+≡ (135a) ◁135c
```
int i, j, k;
int len;
int *ids;
Dirtab dt;
char buf[16];
```

⟨filsysread() *cases* 211a⟩+≡ (135a) ◁136a

```
  case Qwsys:

      qlock(&all);
      ids = emalloc(nwindow * sizeof(int));
      for(j=0; j<nwindow; j++)
          ids[j] = windows[j]->id;
      qunlock(&all);

      qsort(ids, nwindow, sizeof ids[0], idcmp);
      dt.name = buf;
      for(i=0, j=0; j<nwindow && i<e; i+=len){
          k = ids[j];
          sprint(dt.name, "%d", k);
          dt.qid = QID(k, Qdir);
          dt.type = QTDIR;
          dt.perm = DMDIR|0700;
          len = dostat(fs, k, &dt, b+n, x->req.count - n, clock);
          if(len == 0)
              break;
          if(i >= o)
              n += len;
          j++;
      }
      free(ids);
      break;
```

Uses `QID` 122a, `Qdir` 122d, `Qwsys` 209e, `all` 128b, `dostat()` 138b, `idcmp()` 211b, `nwindow` 51b, and `windows` 51a.

⟨*function* idcmp 211b⟩≡ (346c)

```
  static
  int
  idcmp(void *a, void *b)
  {
      return *(int*)a - *(int*)b;
  }
```

12.6 /mnt/wsys/wctl

The `/mnt/wsys/wctl` file is the programmatic interface to window management, the most powerful file in `/mnt/wsys/`. Reading `wctl` returns the window's current geometry, focus status, and visibility (hidden or visible) as a single line. Writing `wctl` accepts commands such as "new", "resize", "move", "delete", "hide", "unhide", "top", "bottom", "scroll", and "noscroll". This file is how tools like the `wctl(1)` command interact with `rio` from the shell—for example, `echo resize -r 0 0 640 480 > /mnt/wsys/wctl` resizes the current window. Combined with `/mnt/wsys/wsys/<winid>/wctl`, a script can control any window, enabling automation and scripting of window layouts in a way that is natural under Plan 9: everything is a file.

⟨Qxxx *other cases* 211c⟩+≡ (122d) ◁210c 227g▷

```
  Qwctl,
```

⟨*dirtab array elements* 211d⟩+≡ (123b) ◁210a 227h▷

```
  { "wctl",  QTFILE, Qwctl,  0600 },
```

Uses `Qwctl` 211c.

Opening `wctl` for reading is exclusive: only one reader is allowed at a time. As the comment in the code explains, ideally multiple readers could all see geometry changes (fan-out), but `rio`'s channel-based architecture

is designed for the `/dev/cons` model where each reader gets different data. Enforcing exclusivity avoids subtle races:

⟨xfidopen() *cases* 212a⟩+≡ (133a) ◁159 228a▷
```
  case Qwctl:
      if(x->req.mode==OREAD || x->req.mode==ORDWR){
          /*
           * It would be much nicer to implement fan-out for wctl reads,
           * so multiple people can see the resizings, but rio just isn't
           * structured for that.  It's structured for /dev/cons, which gives
           * alternate data to alternate readers.  So to keep things sane for
           * wctl, we compromise and give an error if two people try to
           * open it.  Apologies.
           */
          if(w->wctlopen){
              filsysrespond(x->fs, x, &fc, Einuse);
              return;
          }
          w->wctlopen = true;
          w->wctlready = true;
          wsendctlmesg(w, Wakeup, ZR, nil);
      }
      break;
```
Uses `Einuse` 281a, `Qwctl` 211c, `Wakeup` 272c, `filsysrespond()` 124, and `wsendctlmesg()` 91c.

⟨xfidclose() *cases* 212b⟩+≡ (134) ◁168b 232f▷
```
  case Qwctl:
      if(x->f->mode==OREAD || x->f->mode==ORDWR)
          w->wctlopen = false;
  break;
```
Uses `Qwctl` 211c.

⟨Window *other fields* 212c⟩+≡ (49) ◁181c 213a▷
```
  bool wctlopen;
  bool  wctlready;
```

The `Wakeup`[272c] message carries no data and does no work of its own; it merely nudges the `winctl()`[71b] thread to re-evaluate its `alt()` arms, so it notices that `wctlready` is now set and offers the fresh geometry. We meet `Wakeup` again, with the other control messages, in Section 13.8.2.

⟨wcurrent() *wakeup w and oi* 212d⟩≡ (105a)
```
  if(w != oi){
      if(oi){
          oi->wctlready = true;
          wsendctlmesg(oi, Wakeup, ZR, nil);
      }
      if(w){
          w->wctlready = true;
          wsendctlmesg(w, Wakeup, ZR, nil);
      }
  }
```
Uses `Wakeup` 272c and `wsendctlmesg()` 91c.

12.6.1 Reading `/mnt/wsys/wctl`

Reading `wctl` uses the same producer/consumer pattern as `/mnt/wsys/cons`: the worker thread (consumer) blocks on `Window.wctlread`[213a] until the window thread (producer) sends the current geometry. The read blocks until the geometry *changes* (a resize, move, or focus change sets `Window.wctlready`[212c] to true and wakes

the window thread). This makes `wctl` useful for programs that want to react to window changes—they simply read in a loop. It even reuses the console's message type: `wctlread` is a `chan(Consreadmesg)`, so the geometry line rides the very same two-channel buffer protocol as `/dev/cons` reads, rather than getting a message type of its own.

⟨`Window` *other fields* 213a⟩+≡ (49) ◁212c
```
// chan<Consreadmesg> (listener = , sender = )
Channel  *wctlread; /* chan(Consreadmesg) */
```

⟨`wmk()` *channels creation* 213b⟩+≡ (94c) ◁149b
```
w->wctlread =  chancreate(sizeof(Consreadmesg), 0);
```

Producer

The producer is `winctl()`[71b]'s `WWread` arm. It offers to send the geometry (`CHANSND`) only when `wctlready` is set—after a resize, move, or focus change—and disables that arm (`CHANNOP`) otherwise, which is exactly what makes a `wctl` read block until something changes. When it fires it formats the rectangle and the current/hidden flags into the line the consumer returns.

⟨`winctl()` *other locals* 213c⟩+≡ (71b) ◁197a 213i▷
```
Consreadmesg cwrm;
```

⟨`winctl()` *channels creation* 213d⟩+≡ (71b) ◁196a
```
cwrm.c1 = chancreate(sizeof(Stringpair), 0);
cwrm.c2 = chancreate(sizeof(Stringpair), 0);
```

⟨`winctl()` *Wctl case, free channels if wctlmesg is Excited* 213e⟩+≡ (74g) ◁196b
```
chanfree(cwrm.c1);
chanfree(cwrm.c2);
```

⟨`Wxxx` *cases* 213f⟩+≡ (71a) ◁195f
```
WWread,
```

⟨`winctl()` *alts setup* 213g⟩+≡ (71b) ◁196c
```
alts[WWread].c = w->wctlread;
alts[WWread].v = &cwrm;
alts[WWread].op = CHANSND;
```
Uses `WWread-31` 213f.

⟨`winctl()` *alts adjustments* 213h⟩+≡ (71b) ◁196d
```
if(w->deleted || !w->wctlready)
    alts[WWread].op = CHANNOP;
else
    alts[WWread].op = CHANSND;
```
Uses `WWread-31` 213f.

⟨`winctl()` *other locals* 213i⟩+≡ (71b) ◁213c
```
char *s;
```

⟨winctl() *event loop cases* 214a⟩+≡ (71b) ◁196f
```
  case WWread:
      w->wctlready = false;
      recv(cwrm.c1, &pair);
      if(w->deleted || w->i==nil)
          pair.ns = sprint(pair.s, "");
      else{
          s = "visible";
          for(i=0; i<nhidden; i++)
              if(hidden[i] == w){
                  s = "hidden";
                  break;
              }
          t = "notcurrent";
          if(w == input)
              t = "current";
          pair.ns = snprint(pair.s, pair.ns, "%11d %11d %11d %11d %s %s ",
              w->i->r.min.x, w->i->r.min.y, w->i->r.max.x, w->i->r.max.y, t, s);
      }
      send(cwrm.c2, &pair);
      continue;
```
Uses `WWread-31` 213f, `hidden` 118c, `input` 51e, and `nhidden` 118d.

Consumer

The consumer is the `xfidread()`[136b] `Qwctl` case. Like the console read, it `alt()`s on `wctlread` (with a flush), then runs the two-channel handshake to copy the geometry line into the client's buffer.

⟨*enum* `_anon_` `(rio/xfid.c)` *5* 214b⟩≡ (347)
```
  enum { WCRdata, WCRflush, NWCR };
```

⟨xfidread() *other locals* 214c⟩+≡ (136b) ◁170a
```
  Consreadmesg cwrm;
```

⟨xfidread() *cases* 214d⟩+≡ (136b) ◁209d 232g▷
```
  case Qwctl: /* read returns rectangle, hangs if not resized */
      if(cnt < 4*12){
          filsysrespond(x->fs, x, &fc, Etooshort);
          break;
      }
      ⟨xfidxxx() set flushtag 267e⟩

      alts[WCRdata].c = w->wctlread;
      alts[WCRdata].v = &cwrm;
      alts[WCRdata].op = CHANRCV;
      ⟨xfidread() when Qwctl, set alts for flush 269b⟩
      alts[NMR].op = CHANEND;

      switch(alt(alts)){
      case WCRdata:
          break;
      ⟨xfidread() when Qwctl, switch alt flush case 269c⟩
      }

      /* received data */
      ⟨xfidxxx() unset flushtag 267f⟩
      c1 = cwrm.c1;
      c2 = cwrm.c2;
      t = malloc(cnt+1); /* be sure to have room for NUL */
```

```
    pair.s = t;
    pair.ns = cnt+1;
    send(c1, &pair);
    ⟨xfidread() when Qwctl, if flushing 269d⟩

    qlock(&x->active);
    recv(c2, &pair);
    fc.data = pair.s;
    if(pair.ns > cnt)
        pair.ns = cnt;
    fc.count = pair.ns;
    filsysrespond(x->fs, x, &fc, nil);
    free(t);
    qunlock(&x->active);
    break;
```
Uses `Etooshort` 281d, `NMR-98` 143a, `Qwctl` 211c, `WCRdata-99` 214b, and `filsysrespond()` 124.

12.6.2 Writing `/mnt/wsys/wctl` (controlling windows)

Writing to `wctl` accepts commands like "new", "resize", "move", "delete", "hide", "unhide", "top", "bottom", "scroll", "noscroll", and "set". The `parsewctl()`[263c] function tokenizes the command string and extracts the rectangle, window ID, and other parameters. Commands that affect window geometry (`Move`, `Resize`) use the same `wsendctlmesg()`[91c] path as mouse operations—the window thread handles all geometry changes uniformly, regardless of whether they came from the mouse or the filesystem.

⟨`xfidwrite()` *cases* 215a⟩+≡ (137b) ◁209a 228b▷
```
  case Qwctl:
    if(writewctl(x, buf) < 0){
        filsysrespond(x->fs, x, &fc, buf);
        return;
    }
    flushimage(display, true);
    break;
```
Uses `Qwctl` 211c, `filsysrespond()` 124, and `writewctl()` 215b.

⟨*function* `writewctl` 215b⟩≡ (349b)
```
  int
  writewctl(Xfid *x, char *err)
  {
    int cnt, cmd, j, id, hideit, scrollit, pid;
    Image *i;
    char *arg, *dir;
    Rectangle rect;
    Window *w;

    w = x->f->w;
    cnt = x->req.count;
    x->req.data[cnt] = '\0';
    id = 0;

    rect = rectsubpt(w->screenr, view->r.min);
    cmd = parsewctl(&arg, rect, &rect, &pid, &id, &hideit, &scrollit, &dir, x->req.data, err);
    if(cmd < 0)
        return -1;

    if(mouse->buttons!=0 && cmd>=Top){
        strcpy(err, "action disallowed when mouse active");
        return -1;
```

```
    }

    if(id != 0){
        for(j=0; j<nwindow; j++)
            if(windows[j]->id == id)
                break;
        if(j == nwindow){
            strcpy(err, "no such window id");
            return -1;
        }
        w = windows[j];
        if(w->deleted || w->i==nil){
            strcpy(err, "window deleted");
            return -1;
        }
    }

    switch(cmd){
    ⟨writewctl() switch cmd cases 216a⟩
    }
    strcpy(err, "invalid wctl message");
    return -1;
}
```

Uses Top-66 260a, mouse 48c, nwindow 51b, and windows 51a.

The command string has already been turned into a `cmd` code plus arguments by `parsewctl()`, a parser shared with the external-access paths; its code is shown in Section 13.6.

The cases themselves are short. Each does the same work as the matching right-click menu entry, by calling the same core functions—`whide()`[118e], `wcurrent()`[105a], and so on. Driving a window through `wctl` and through the menu converge on the same code.

⟨writewctl() *switch* cmd *cases* 216a⟩≡ (215b) 216b▷

```
case Move:
    rect = Rect(rect.min.x, rect.min.y, rect.min.x+Dx(w->screenr), rect.min.y+Dy(w->screenr));
    rect = rectonscreen(rect);
    /* fall through */
case Resize:
    if(!goodrect(rect)){
        strcpy(err, Ebadwr);
        return -1;
    }
    if(eqrect(rect, w->screenr))
        return 1;
    i = allocwindow(desktop, rect, Refbackup, DWhite);
    if(i == nil){
        strcpy(err, Ewalloc);
        return -1;
    }
    border(i, rect, Selborder, red, ZP);
    wsendctlmesg(w, Reshaped, i->r, i);
    return 1;
```

Uses Ebadwr 280f, Ewalloc 280g, Move-62 260a, Reshaped 91a, Resize-61 260a, Selborder 67, desktop 48d, goodrect() 261b, rectonscreen() 263a, red 48f, and wsendctlmesg() 91c.

⟨writewctl() *switch* cmd *cases* 216b⟩+≡ (215b) ◁216a 217a▷

```
case Scroll:
    w->scrolling = 1;
    wshow(w, w->nr);
    wsendctlmesg(w, Wakeup, ZR, nil);
    return 1;
```

Uses Scroll-63 260a, Wakeup 272c, wsendctlmesg() 91c, and wshow() 179b.

⟨writewctl() *switch* cmd *cases* 217a⟩+≡ (215b) ◁216b 217b▷
```
  case Noscroll:
      w->scrolling = 0;
      wsendctlmesg(w, Wakeup, ZR, nil);
      return 1;
```
Uses Noscroll-64 260a, Wakeup 272c, and wsendctlmesg() 91c.

⟨writewctl() *switch* cmd *cases* 217b⟩+≡ (215b) ◁217a 217c▷
```
  case Top:
      wtopme(w);
      return 1;
```
Uses Top-66 260a and wtopme() 217d.

⟨writewctl() *switch* cmd *cases* 217c⟩+≡ (215b) ◁217b 217f▷
```
  case Bottom:
      wbottomme(w);
      return 1;
```
Uses Bottom-67 260a and wbottomme() 217e.

⟨*function* wtopme 217d⟩≡ (342b)
```
  void
  wtopme(Window *w)
  {
      if(w!=nil && w->i!=nil && !w->deleted && w->topped!=topped){
          topwindow(w->i);
          flushimage(display, 1);
          w->topped = ++topped;
      }
  }
```
Uses topped-16 51c.

⟨*function* wbottomme 217e⟩≡ (342b)
```
  void
  wbottomme(Window *w)
  {
      if(w!=nil && w->i!=nil && !w->deleted){
          bottomwindow(w->i);
          flushimage(display, 1);
          w->topped = - ++topped;
      }
  }
```
Uses topped-16 51c.

⟨writewctl() *switch* cmd *cases* 217f⟩+≡ (215b) ◁217c 217g▷
```
  case Current:
      wcurrent(w);
      return 1;
```
Uses Current-68 260a and wcurrent() 105a.

⟨writewctl() *switch* cmd *cases* 217g⟩+≡ (215b) ◁217f 218a▷
```
  case Hide:
      switch(whide(w)){
      case -1:
          strcpy(err, "window already hidden");
          return -1;
      case 0:
          strcpy(err, "hide failed");
```

```
        return -1;
    default:
        break;
    }
    return 1;
```

Uses Hide-69 260a and whide() 118e.

⟨writewctl() *switch* cmd *cases* 218a⟩+≡ (215b) ◁217g 218b▷

```
  case Unhide:
    for(j=0; j<nhidden; j++)
        if(hidden[j] == w)
            break;
    if(j == nhidden){
        strcpy(err, "window not hidden");
        return -1;
    }
    if(wunhide(j) == 0){
        strcpy(err, "hide failed");
        return -1;
    }
    return 1;
```

Uses Unhide-70 260a, hidden 118c, nhidden 118d, and wunhide() 120b.

⟨writewctl() *switch* cmd *cases* 218b⟩+≡ (215b) ◁218a 218c▷

```
  case Delete:
    wsendctlmesg(w, Deleted, ZR, nil);
    return 1;
```

Uses Delete-71 260a, Deleted 106b, and wsendctlmesg() 91c.

New is the one heavyweight case: it actually creates a window, through wctlnew()[265]. We return to it with the external-access mechanism in Section 13.6.

⟨writewctl() *switch* cmd *cases* 218c⟩+≡ (215b) ◁218b 218d▷

```
  case New:
    return wctlnew(rect, arg, pid, hideit, scrollit, dir, err);
```

Uses New-60 260a and wctlnew() 265.

Set attaches a process id to the window, via wsetpid()[98c], so rio knows which process the window belongs to—whom to send notes to when, for instance, the window is deleted. A program that adopts a window rio did not start a shell in uses this to register itself.

⟨writewctl() *switch* cmd *cases* 218d⟩+≡ (215b) ◁218c

```
  case Set:
    if(pid > 0)
        wsetpid(w, pid, 0);
    return 1;
```

Uses Set-65 260a and wsetpid() 98c.

Chapter 13

Advanced Topics

This chapter covers features and topics that go beyond `rio`'s basic operation: signal handling, `rio`'s command-line options (`rio -s`, `rio -i`, `rio -k`, `rio -f`), additional control messages, the timer subsystem, recursive `rio`, or external access to `rio`'s filesystem (thanks to the `/srv/rio.<user>.<pid>` and `/srv/riowctl.<user>.<pid>` files). It also covers advanced terminal editing features like snarf/paste, plumbing, auto-completion, and word selection.

13.1 Notes (signals)

A *note* is Plan 9's equivalent of a UNIX signal: a short string delivered asynchronously to a process. We have already seen `rio` on the *sending* side—posting a "hangup" note to a window's process group to kill it when the window is deleted. This section is the other side: what happens when a note arrives at `rio` *itself*, so it can tear down its windows and exit cleanly.

When `rio` itself receives a note, `shutdown()`[219c] first sends "hangup" to all window processes via `killprocs()`[220a], then checks if the note is one of the expected shutdown reasons ("delete", "hangup", "kill", "exit"). For expected notes, it calls `threadexitsall()` with a lock to prevent multiple threads from racing. For unexpected notes, it prints a diagnostic and aborts—a deliberate crash to aid debugging.

⟨*global* `oknotes` 219a⟩≡ (338b)

```
char *oknotes[] =
{
    "delete",
    "hangup",
    "kill",
    "exit",
    nil
};
```

⟨`main()` *error management after everything setup* 219b⟩≡ (58) 282b ▷

```
threadnotify(shutdown, true);
```

⟨*function* `shutdown` 219c⟩≡ (338b)

```
/// threadmain -> threadnotify(<>, true)
int
shutdown(void *, char *msg)
{
    int i;
    static Lock shutdownlk;

    killprocs();
    for(i=0; oknotes[i]; i++)
        if(strncmp(oknotes[i], msg, strlen(oknotes[i])) == 0){
```

```
            lock(&shutdownlk); /* only one can threadexitsall */
            threadexitsall(msg);
        }
    fprint(STDERR, "rio %d: abort: %s\n", getpid(), msg);
    abort();
    exits(msg);
    return 0;
}
```

Uses `killprocs()` 220a and `oknotes` 219a.

⟨*function* `killprocs` 220a⟩≡ (338b)

```
void
killprocs(void)
{
    int i;

    for(i=0; i<nwindow; i++)
        postnote(PNGROUP, windows[i]->pid, "hangup");
}
```

Uses `nwindow` 51b, `postnote()` 355b, and `windows` 51a.

Notes also tie back to the `Tflush` handling of Section 13.7.1: when a user interrupts a program blocked in a `read`, the note that unwinds the program is what makes the kernel send `rio` a `Tflush`, cancelling the pending `xfidread()`[136b] so no stale reply is delivered.

13.2 Recursive `rio`

Because `rio` is itself a regular graphical application that uses `/dev/draw`, `/dev/mouse`, and `/dev/cons`, it can run inside another instance of `rio`—each nested `rio` sees virtualized devices from its parent. This works almost for free, with no special case code needed for the basic functionality. The only additional code is handling the resize event from the parent `rio`, so that the inner `rio` can proportionally reposition all its windows within the new boundaries.

In Plan 9, as we saw before resize events are delivered through `/dev/mouse` as a special message type (the "r" prefix instead of "m"). `rio` just needs to listen on its existing `mousectl->resizec` channel.

⟨`Mxxx` *cases* 220b⟩≡ (65a)

```
MReshape,
```

⟨`mousethread()` *alts setup* 220c⟩+≡ (66a) ◁66b

```
alts[MReshape].c = mousectl->resizec;
alts[MReshape].v = nil;
alts[MReshape].op = CHANRCV;
```

Uses `MReshape-6` 220b and `mousectl` 48a.

⟨`mousethread()` *event loop cases* 220d⟩+≡ (66a) ◁66e

```
case MReshape:
    resized();
    break;
```

Uses `MReshape-6` 220b and `resized()` 221.

`resized()`[221] is a standard handler that any graphical application should implement. For `rio`, the logic is more involved: it must rebuild the entire `desktop`[48d], then proportionally scale every window's rectangle to fit

the new dimensions. Each window receives a `Reshaped`[91a] control message to redraw itself.

⟨*function* `resized` 221⟩≡ (339a)

```
void
resized(void)
{
    Image *im;
    int i, j;
    bool ishidden;
    Rectangle r;
    Point o, n;
    Window *w;

    // updates view (and screen)
    if(getwindow(display, Refnone) < 0)
        error("failed to re-attach window");

    freescrtemps();
    freescreen(desktop);

    desktop = allocscreen(view, background, false);
    ⟨resized() sanity check desktop 222a⟩
    draw(view, view->r, background, nil, ZP);

    // old view rectangle
    o = subpt(viewr.max, viewr.min);
    n = subpt(view->clipr.max, view->clipr.min);

    for(i=0; i<nwindow; i++){
        w = windows[i];
        ⟨resized() continue if window was deleted 222b⟩
        r = rectsubpt(w->i->r, viewr.min);
        r.min.x = (r.min.x*n.x)/o.x;
        r.min.y = (r.min.y*n.y)/o.y;
        r.max.x = (r.max.x*n.x)/o.x;
        r.max.y = (r.max.y*n.y)/o.y;
        r = rectaddpt(r, view->clipr.min);

        ishidden = false;
        for(j=0; j<nhidden; j++)
            if(w == hidden[j]){
                ishidden = true;
                break;
            }
        if(ishidden){
            im = allocimage(display, r, view->chan, false, DWhite);
            r = ZR;
        }else
            im = allocwindow(desktop, r, Refbackup, DWhite);

        if(im)
            wsendctlmesg(w, Reshaped, r, im);
    }
    viewr = view->r;
    flushimage(display, true);
}
```

Uses `Reshaped` 91a, `background` 48e, `desktop` 48d, `error()` 282c, `freescrtemps()` 203b, `hidden` 118c, `nhidden` 118d, `nwindow` 51b, `viewr` 47, `windows` 51a, and `wsendctlmesg()` 91c.

Note the ratio-based scaling: each window's coordinates are expressed as a fraction of the old `view` size and

then mapped onto the new size. This preserves the relative layout even when the aspect ratio changes. Hidden windows get a plain `allocimage()` instead of a `desktop` layer, since they are not visible.

⟨`resized()` *sanity check desktop* 222a⟩≡ (221)
```
if(desktop == nil)
    error("can't re-allocate desktop");
```
Uses `desktop` 48d and `error()` 282c.

⟨`resized()` *continue if window was deleted* 222b⟩≡ (221)
```
if(w->deleted)
    continue;
```

And that is essentially the whole of it: running `rio` inside one of its own windows takes almost no special code. A nested `rio` is just another graphical client of the virtual devices its window already provides, so it works for free—the only real plumbing is the few dozen lines above that resize the nested screen when its window changes, on top of `rio`'s 8 800 lines in all. Other systems pay far more. To nest a display X11 ships a *whole separate X server*: the old `Xnest` or the modern `Xephyr` Either way it is thousands of lines of nesting-specific code that is inert without the rest of the X server behind it, itself hundreds of thousands of lines. Nesting a Wayland compositor likewise means running a second full compositor as a client, and macOS's window server is a single system service you do not nest at all. Under Plan 9 you simply just run `rio`, and you can watch and debug it like any ordinary program.

13.3 Timer

The timer subsystem provides cancellable one-shot timers, used primarily by `wscrsleep()`[203c] for scroll bar repeat delays. The design uses a dedicated `timerproc()`[224] that polls at 1ms intervals, decrementing each `Timer.dt`[223a] countdown. When a timer fires, it sends on the `ctimer`[222c] channel; when cancelled, it is recycled into a freelist (`timer`[222f]).

The "scroll delays" pace a window's output. Without them, a program that spews output would scroll the screen as fast as the lines arrive—an unreadable blur. So when a window auto-scrolls, `wscrsleep()` waits a brief, cancellable moment before scrolling, letting a burst accumulate and roll by at a rate you can read. The freelist keeps that frequent little wait cheap.

⟨*global* `ctimer` 222c⟩≡ (349a)
```
// chan<?> (listener = ?, sender = ?)
static Channel* ctimer; /* chan(Timer*)[100] */
```

⟨`main()` *threads creation* 222d⟩+≡ (58) ◁108e
```
timerinit();
```

⟨*function* `timerinit` 222e⟩≡ (349a)
```
void
timerinit(void)
{
    ctimer = chancreate(sizeof(Timer*), 100);
    proccreate(timerproc, nil, STACK);
}
```
Uses `STACK` 61a, `ctimer-58` 222c, and `timerproc()` 224.

⟨*global* `timer` 222f⟩≡ (349a)
```
// freelist
static Timer *timer;
```

⟨*struct* `Timer` 223a⟩≡ (333b)
```
struct Timer
{
    int  dt;
    int  cancel;
    Channel *c; /* chan(int) */

    // ref_own<Timer> (head = timer)
    Timer *next;
};
```

Timer allocation uses a freelist (`timer`) to avoid repeated `malloc`/`free` for the common case of scroll delays. `timerstart()`[223b] either reuses a freed timer or allocates a new one, sets the countdown and sends it to `timerproc()` via the `ctimer` global channel.

⟨*function* `timerstart` 223b⟩≡ (349a)
```
/*
 * timeralloc() and timerfree() don't lock, so can only be
 * called from the main proc.
 */
Timer*
timerstart(int dt)
{
    Timer *t;

    t = timer;
    if(t)
        timer = timer->next;
    else{
        t = emalloc(sizeof(Timer));
        t->c = chancreate(sizeof(int), 0);
    }
    t->next = nil;
    t->dt = dt;
    t->cancel = false;
    sendp(ctimer, t);
    return t;
}
```
Uses `ctimer-58` 222c and `timer-59` 222f.

⟨*function* `timerstop` 223c⟩≡ (349a)
```
void
timerstop(Timer *t)
{
    t->next = timer;
    timer = t;
}
```
Uses `timer-59` 222f.

⟨*function* `timercancel` 223d⟩≡ (349a)
```
void
timercancel(Timer *t)
{
    t->cancel = true;
}
```

`timerproc()` is interesting for its blocking/polling hybrid: when no timers are active, it blocks on `recvp(ctimer)` waiting for work. When timers are active, it polls with `sleep(1)` and checks for new timer requests with non-

blocking `nbrecv()`. The `goto gotit` pattern avoids duplicating the timer-insertion code.

⟨*function* timerproc 224⟩≡ (349a)

```
static
void
timerproc(void*)
{
    int i, nt, na, dt, del;
    Timer **t, *x;
    uint old, new;

    rfork(RFFDG);
    threadsetname("TIMERPROC");

    t = nil;
    na = 0;
    nt = 0;
    old = msec();

    for(;;){
        sleep(1); /* will sleep minimum incr */
        new = msec();
        dt = new-old;
        old = new;
        if(dt < 0) /* timer wrapped; go around, losing a tick */
            continue;
        for(i=0; i<nt; i++){
            x = t[i];
            x->dt -= dt;
            del = 0;
            if(x->cancel){
                timerstop(x);
                del = 1;
            }else if(x->dt <= 0){
                /*
                 * avoid possible deadlock if client is
                 * now sending on ctimer
                 */
                if(nbsendul(x->c, 0) > 0)
                    del = 1;
            }
            if(del){
                memmove(&t[i], &t[i+1], (nt-i-1)*sizeof t[0]);
                --nt;
                --i;
            }
        }
        if(nt == 0){
            x = recvp(ctimer);
    gotit:
            if(nt == na){
                na += 10;
                t = realloc(t, na*sizeof(Timer*));
                if(t == nil)
                    abort();
            }
            t[nt++] = x;
            old = msec();
        }
        if(nbrecv(ctimer, &x) > 0)
```

```
            goto gotit;
        }
    }
```
Uses ctimer-58 222c, msec() 225a, and timerstop() 223c.

⟨*function* msec 225a⟩≡ (349a)
```
  static
  uint
  msec(void)
  {
      return nsec()/1000000;
  }
```

13.4 Command-line options

rio's command-line options are few: a font (-f), an initial command to run in the first window (-i), a keyboard command (-k), and the -s flag for scolling. The usage()[225b] message below lists them.

⟨*function* usage 225b⟩≡ (338b)
```
  void
  usage(void)
  {
      fprint(STDERR, "usage: rio [-f font] [-i initcmd] [-k kbdcmd] [-s]\n");
      exits("usage");
  }
```

13.4.1 Automatic scrolling: rio -s

By default, rio windows only auto-scroll when the user is not reading back through the output (i.e., when q0 is at the end). The -s flag makes all new windows scroll unconditionally, which is useful for monitoring logs or long-running commands.

Plan 9's default is the unusual one. Most terminals (xterm and the rest) always jump to the bottom on new output; Plan 9 instead holds your view in place while you are reading back, and only follows the tail once you return to the end. It is a small thing that proves surprisingly pleasant—you can read a streaming command's output without having to re-run it through less or more to keep it from scrolling away. Emacs's eshell imitates the same behaviour.

⟨*global* scrolling 225c⟩≡ (338a)
```
  bool  scrolling;
```

⟨main() *command line processing* 225d⟩≡ (58) 226b ▷
```
  case 's':
      scrolling = true;
      break;
```

There is no new code to show for the flag itself: the scrolling global is only the *default*. Each window copies it into its own Window.scrolling[52f] at creation (in wmk()[94c]), and the field is consulted in the terminal's output loop—the qh < q0 || scrolling test seen earlier—to decide whether to keep pulling output from the process and scroll.

13.4.2 Initial command: `rio -i`

The `-i` flag runs a command at startup, useful for automated setups that create multiple windows with specific programs. The command runs in a fresh process with its own environment, namespace, file descriptors, and note group.

⟨`main()` *locals* 226a⟩+≡ (58) ◁195d 226e▷
```
char *initstr = nil;
```

⟨`main()` *command line processing* 226b⟩+≡ (58) ◁225d 226f▷
```
case 'i':
    initstr = ARGF();
    if(initstr == nil)
        usage();
    break;
```

⟨`main()` *if initstr or kdbin* 226c⟩≡ (282b) 227b▷
```
if(initstr)
    proccreate(initcmd, initstr, STACK);
```

⟨*function* `initcmd` 226d⟩≡ (338b)
```
void
initcmd(void *arg)
{
    char *cmd;

    cmd = arg;
    rfork(RFENVG|RFFDG|RFNOTEG|RFNAMEG);
    procexecl(nil, "/bin/rc", "rc", "-c", cmd, nil);
    fprint(STDERR, "rio: exec failed: %r\n");
    exits("exec");
}
```

The `-c` flag tells `rc` to run the command string that follows instead of reading from standard input—the same `-c` every shell has, covered in the SHELL book [Pad18].

13.4.3 Fake keyboard input: `rio -k`

The `-k` flag creates a special "keyboard" window that feeds simulated keyboard input to `rio` through `/dev/kbdin`. This is used for software keyboard implementations (e.g., on touchscreen devices) or testing. The keyboard window gets special treatment: it always receives mouse input when the pointer is over it, it cannot become the "current" window (to avoid stealing focus from the target), and button 6 toggles its visibility.

⟨`main()` *locals* 226e⟩+≡ (58) ◁226a 227a▷
```
char *kbdin = nil;
```

⟨`main()` *command line processing* 226f⟩+≡ (58) ◁226b 229b▷
```
case 'k':
    if(kbdin != nil)
        usage();
    kbdin = ARGF();
    if(kbdin == nil)
        usage();
    break;
```

⟨*global* `wkeyboard` 226g⟩≡ (338a)
```
Window *wkeyboard; /* window of simulated keyboard */
```

⟨*global* `kbdargv` 226h⟩≡ (338b)
```
char *kbdargv[] = { "rc", "-c", nil, nil };
```

⟨main() *locals* 227a⟩+≡ (58) ◁226e
```
  Image *i;
  Rectangle r;
```

⟨main() *if initstr or kdbin* 227b⟩+≡ (282b) ◁226c
```
  if(kbdin){
      kbdargv[2] = kbdin;
      r = view->r;
      r.max.x = r.min.x+300;
      r.max.y = r.min.y+80;
      i = allocwindow(desktop, r, Refbackup, DWhite);
      wkeyboard = new(i, false, scrolling, 0, nil, "/bin/rc", kbdargv);
      if(wkeyboard == nil)
          error("can't create keyboard window");
  }
```

wkeyboard

⟨mousethread() *if wkeyboard and button 6* 227c⟩≡ (66e)
```
  if(wkeyboard!=nil && (mouse->buttons & (1<<5))){
      keyboardhide();
      break;
  }
```
Uses `keyboardhide()` 228d, `mouse` 48c, and `wkeyboard` 226g.

⟨mousethread() *if wkeyboard and ptinrect* 227d⟩≡ (66e)
```
  /* override everything for the keyboard window */
  if(wkeyboard!=nil && ptinrect(mouse->xy, wkeyboard->screenr)){
      /* make sure it's on top; this call is free if it is */
      wtopme(wkeyboard);
      winput = wkeyboard;
  }
```
Uses `mouse` 48c, `wkeyboard` 226g, and `wtopme()` 217d.

⟨wcurrent() *if wkeyboard* 227e⟩≡ (105a)
```
  if(wkeyboard!=nil && w==wkeyboard)
      return;
```
Uses `wkeyboard` 226g.

⟨wclosewin() *if wkeyboard* 227f⟩≡ (106d)
```
  if(w == wkeyboard)
      wkeyboard = nil;
```
Uses `wkeyboard` 226g.

/mnt/wsys/kdbin

The `/mnt/wsys/kdbin` file is write-only and restricted to the keyboard window. Writing to it converts the bytes to runes and injects them into the keyboard channel, making them appear as if the user had typed them on a real keyboard.

⟨Qxxx *other cases* 227g⟩+≡ (122d) ◁211c 232a▷
```
  Qkbdin,
```

⟨*dirtab array elements* 227h⟩+≡ (123b) ◁211d 232b▷
```
  { "kbdin",  QTFILE, Qkbdin,  0200 },
```
Uses `Qkbdin` 227g.

⟨xfidopen() *cases* 228a⟩+≡ (133a) ◁212a 232d▷

```
case Qkbdin:
    if(w !=  wkeyboard){
        filsysrespond(x->fs, x, &fc, Eperm);
        return;
    }
    break;
```

Uses Eperm 280a, Qkbdin 227g, filsysrespond() 124, and wkeyboard 226g.

⟨xfidwrite() *cases* 228b⟩+≡ (137b) ◁215a 233b▷

```
case Qkbdin:
    keyboardsend(req->data, cnt);
    break;
```

Uses Qkbdin 227g and keyboardsend() 228c.

⟨*function* keyboardsend 228c⟩≡ (347)

```
/*
 * Used by /dev/kbdin
 */
void
keyboardsend(char *s, int cnt)
{
    Rune *r;
    int i, nb, nr;

    r = runemalloc(cnt);
    /* BUGlet: partial runes will be converted to error runes */
    cvttorunes(s, cnt, r, &nb, &nr, nil);
    for(i=0; i<nr; i++)
        send(keyboardctl->c, &r[i]);
    free(r);
}
```

Uses cvttorunes() 285b, keyboardctl 48b, and runemalloc 284d.

Keyboard hide

Button 6 toggles the keyboard window's visibility. The handler forwards both the button-down and button-up mouse events to the keyboard window's mouse channel, letting the keyboard program handle the toggle logic itself. `rio` reports mouse buttons as a bitmask, and the numbering runs past the usual three: 1–3 are the physical buttons, 4 and 5 the two scroll-wheel directions, and 6 a further "button" that an ordinary three-button mouse never generates. It is meant for devices that show an *on-screen* keyboard (the Plan 9 touch ports), where something has to toggle that keyboard on and off.

⟨*function* keyboardhide 228d⟩≡ (339a)

```
/*
 * Button 6 - keyboard toggle - has been pressed.
 * Send event to keyboard, wait for button up, send that.
 * Note: there is no coordinate translation done here; this
 * is just about getting button 6 to the keyboard simulator.
 */
void
keyboardhide(void)
{
    send(wkeyboard->mc.c, mouse);
    do
        readmouse(mousectl);
    while(mouse->buttons & (1<<5));
    send(wkeyboard->mc.c, mouse);
```

```
  }
```

Uses `mouse` 48c, `mousectl` 48a, and `wkeyboard` 226g.

13.4.4 Font selection: `rio -f`

The `-f` flag overrides the default font. If not specified, `rio` falls back to the `font` environment variable, then to the system default Lucida. The font is validated before `rio` takes over the screen, and published in the environment for child processes.

⟨*global* `fontname` 229a⟩≡ (338b)

```
  char  *fontname;
```

⟨`main()` *command line processing* 229b⟩+≡ (58) ◁226f

```
  case 'f':
      fontname = ARGF();
      if(fontname == nil)
          usage();
      break;
```

⟨`main()` *set some globals* 229c⟩+≡ (58) ◁195e 233g▷

```
  if(fontname == nil)
      fontname = getenv("font");
  if(fontname == nil)
      fontname = "/lib/font/bit/lucm/unicode.9.font";

  /* check font before barging ahead */
  if(access(fontname, 0) < 0){
      fprint(STDERR, "rio: can't access %s: %r\n", fontname);
      exits("font open");
  }

  putenv("font", fontname);
```

Setting the `font` environment variable is how the choice propagates: Plan 9's `libdraw` reads `$font` in `initdraw()` to pick a program's default font, so every graphical client started under `rio`—and `rio`'s own terminals—inherits it unless it overrides the value. The mechanism lives in the GRAPHICS book [Pad16c].

13.5 Advanced terminal editing

`rio`'s terminal goes beyond a simple dumb terminal by providing editor-like features: cut/copy/paste ("snarf" in Plan 9 terminology), plumbing (inter-application message passing), auto-completion, and intelligent word selection. These features are accessed through the middle-click menu and special key bindings.

It is worth noting where these features live. In UNIX, line editing, history, and completion are usually the job of the shell, or of each program linking a library like `readline`. Plan 9 makes the opposite choice: `rc` does almost no line editing, and the work moves up into the terminal—that is, into `rio` itself. Every program that reads `/dev/cons` then inherits editing, snarf, and plumbing for free, and the terminal can draw on the full power of the mouse and the screen rather than being confined to the escape-sequence world of a `readline`-style *text user interface* (TUI)—the keyboard-driven, character-cell cousin of the GUI.

13.5.1 Snarf

"Snarf" is Plan 9's term for copy. The cut/copy/paste operations work with a process-local snarf buffer (the `snarf`[233i] rune array) and synchronize with the system-wide clipboard via `/dev/snarf`. When the user types

Ctrl-U (or uses the menu), the selected text is snarfed and then cut—but only if the key is not the interrupt character (0x7F), which should just cut without updating the snarf buffer.

Before processing an ordinary character, `wkeyctl`[73b] first handles a special interaction with the snarf buffer: if the user has placed the cursor before the output point (`q0 < qh`) and types something, the text between `q0` and `q1` is cut (snarfed and deleted) to prevent the user from accidentally typing into output that has already been sent. This "auto-cut" behavior is a consequence of a deeper invariant: the user's typed input must always be at or after `qh`. If the user clicks on old output text and starts typing, it would corrupt the output if inserted there. Instead, `rio` saves the selected text to the snarf buffer (so nothing is lost) and moves `q0` forward. The result is that the typed character appears at the end of the buffer where the application expects it.

⟨`wkeyctl()` *snarf and cut if not interrupt key* 230a⟩≡ (187a)
```
  if(r != 0x7F){ // 0x7F = Interrupt key
      wsnarf(w);
      wcut(w);
  }
```
Uses `wcut()` 231b and `wsnarf()` 231a.

Snarf menu

The middle-click menu provides the full set of clipboard operations and has been described before with the code of `button2menu()`[198b] used for textual windows. Here are the editing related entries.

⟨`button2menu()` *cases* 230b⟩+≡ (198b) ◁200c 230c▷
```
  case Cut:
      wsnarf(w);
      wcut(w);
      wscrdraw(w);
      break;
```
Uses `Cut-36` 199c, `wcut()` 231b, `wscrdraw()` 180b, and `wsnarf()` 231a.

⟨`button2menu()` *cases* 230c⟩+≡ (198b) ◁230b 230d▷
```
  case Snarf:
      wsnarf(w);
      break;
```
Uses `Snarf-38` 199c and `wsnarf()` 231a.

⟨`button2menu()` *cases* 230d⟩+≡ (198b) ◁230c 230e▷
```
  case Paste:
      getsnarf();
      wpaste(w);
      wscrdraw(w);
      break;
```
Uses `Paste-37` 199c, `getsnarf()` 234a, `wpaste()` 231c, and `wscrdraw()` 180b.

`Send` below is the interesting one: it pastes the snarf buffer at the end of the window's text (position `Window.nr`[51h]) and appends a newline if needed, effectively submitting the text to the shell as if the user had typed it and pressed Enter. In raw mode, the text goes through `waddraw()`[164c] instead.

⟨`button2menu()` *cases* 230e⟩+≡ (198b) ◁230d 234b▷
```
  case Send:
      getsnarf();
      wsnarf(w);
      if(nsnarf == 0)
          break;
      if(w->rawing){
          waddraw(w, snarf, nsnarf);
          if(snarf[nsnarf-1]!='\n' && snarf[nsnarf-1]!='\004')
                     waddraw(w, L"\n", 1);
```

```
    }else{
        winsert(w, snarf, nsnarf, w->nr);
        if(snarf[nsnarf-1]!='\n' && snarf[nsnarf-1]!='\004')
            winsert(w, L"\n", 1, w->nr);
    }
    wsetselect(w, w->nr, w->nr);
    wshow(w, w->nr);
    break;
```

Uses `Send-40` 199c, `getsnarf()` 234a, `nsnarf` 233h, `snarf` 233i, `waddraw()` 164c, `winsert()` 177a, `wsetselect()` 185, `wshow()` 179b, and `wsnarf()` 231a.

`wsnarf()`[231a] copies the selection into the global `snarf`[233i] buffer, bumps the version counter, and publishes to `/dev/snarf` so other programs can paste it. `wcut()`[231b] simply deletes the selected range. `wpaste()`[231c] first cuts any existing selection, then inserts the snarf buffer at the cursor—in raw mode, insertion at the end goes through `waddraw()` to bypass the normal output machinery.

⟨*function* `wsnarf` 231a⟩≡ (343b)

```
void
wsnarf(Window *w)
{
    if(w->q1 == w->q0)
        return;
    nsnarf = w->q1 - w->q0;
    snarf = runerealloc(snarf, nsnarf);
    snarfversion++; /* maybe modified by parent */
    runemove(snarf, w->r+w->q0, nsnarf);
    putsnarf();
}
```

Uses `nsnarf` 233h, `putsnarf()` 233k, `runemove` 285a, `runerealloc` 284e, `snarf` 233i, and `snarfversion` 233j.

⟨*function* `wcut` 231b⟩≡ (343b)

```
void
wcut(Window *w)
{
    if(w->q1 == w->q0)
        return;
    wdelete(w, w->q0, w->q1);
    wsetselect(w, w->q0, w->q0);
}
```

Uses `wdelete()` 193a and `wsetselect()` 185.

⟨*function* `wpaste` 231c⟩≡ (343b)

```
void
wpaste(Window *w)
{
    uint q0;

    if(nsnarf == 0)
        return;
    wcut(w);
    q0 = w->q0;
    if(w->rawing && q0==w->nr){
        waddraw(w, snarf, nsnarf);
        wsetselect(w, q0, q0);
    }else{
        q0 = winsert(w, snarf, nsnarf, w->q0);
        wsetselect(w, q0, q0+nsnarf);
    }
}
```

Uses `nsnarf` 233h, `snarf` 233i, `waddraw()` 164c, `wcut()` 231b, `winsert()` 177a, and `wsetselect()` 185.

/mnt/wsys/snarf

The snarf buffer is also exposed as the **/mnt/wsys/snarf** file, providing a filesystem interface to the clipboard. If the host already provides **/dev/snarf** (e.g., when running under another **rio**), that file takes priority and **rio** does not serve its own. The write protocol has an oddity: writes accumulate into a temporary buffer (**tsnarf**), and the actual snarf buffer is only updated when the file is closed. This avoids partial updates when a program writes the clipboard in multiple chunks.

⟨Qxxx *other cases* 232a⟩+≡ (122d) ◁227g 251c▷
```
  Qsnarf,
```

⟨*dirtab array elements* 232b⟩+≡ (123b) ◁227h 251d▷
```
  { "snarf",  QTFILE, Qsnarf,  0600 },
```
Uses Qsnarf 232a.

⟨filsyswalk() *if snarf* 232c⟩≡ (129)
```
  if(snarffd>=0 && strcmp(x->req.wname[i], "snarf")==0)
      break; /* don't serve /dev/snarf if it's provided in the environment */
```
Uses snarffd 233f.

⟨xfidopen() *cases* 232d⟩+≡ (133a) ◁228a
```
  case Qsnarf:
      if(x->req.mode==ORDWR || x->req.mode==OWRITE){
          if(tsnarf)
              free(tsnarf); /* collision, but OK */
          ntsnarf = 0;
          tsnarf = malloc(1);
      }
      break;
```
Uses Qsnarf 232a, ntsnarf-89 233e, and tsnarf-88 233d.

⟨xfidclose() *other locals* 232e⟩≡ (134)
```
  int nb, nulls;
```

⟨xfidclose() *cases* 232f⟩+≡ (134) ◁212b
```
  /* odd behavior but really ok: replace snarf buffer when /dev/snarf is closed */
  case Qsnarf:
      if(x->f->mode==ORDWR || x->f->mode==OWRITE){
          snarf = runerealloc(snarf, ntsnarf+1);
          cvttorunes(tsnarf, ntsnarf, snarf, &nb, &nsnarf, &nulls);
          free(tsnarf);
          tsnarf = nil;
          ntsnarf = 0;
      }
      break;
```
Uses Qsnarf 232a, cvttorunes() 285b, nsnarf 233h, ntsnarf-89 233e, runerealloc 284e, snarf 233i, and tsnarf-88 233d.

⟨xfidread() *cases* 232g⟩+≡ (136b) ◁214d 251e▷
```
  /* The algorithm for snarf and text is expensive but easy and rarely used */
  case Qsnarf:
      getsnarf();
      if(nsnarf)
          t = runetobyte(snarf, nsnarf, &n);
      else {
          t = nil;
          n = 0;
      }
      goto Text;
```
Uses Qsnarf 232a, getsnarf() 234a, nsnarf 233h, runetobyte() 285d, and snarf 233i.

⟨*constant* MAXSNARF 233a⟩≡ (347)
```
#define MAXSNARF 100*1024
```

⟨xfidwrite() *cases* 233b⟩+≡ (137b) ◁228b 252a▷
```
case Qsnarf:
    /* always append only */
    if(ntsnarf > MAXSNARF){ /* avoid thrashing when people cut huge text */
        filsysrespond(x->fs, x, &fc, Elong);
        return;
    }
    tsnarf = erealloc(tsnarf, ntsnarf+cnt+1); /* room for NUL */
    memmove(tsnarf+ntsnarf, req->data, cnt);
    ntsnarf += cnt;
    snarfversion++;
    break;
```
Uses Elong 281g, MAXSNARF-87 233a, Qsnarf 232a, filsysrespond() 124, ntsnarf-89 233e, snarfversion 233j, and tsnarf-88 233d.

⟨dostat() *adjust vers for snarf* 233c⟩≡ (138b)
```
if(dir->qid == Qsnarf)
    d.qid.vers = snarfversion;
```
Uses Qsnarf 232a and snarfversion 233j.

⟨*global* tsnarf 233d⟩≡ (347)
```
static char *tsnarf;
```

⟨*global* ntsnarf 233e⟩≡ (347)
```
static int ntsnarf;
```

⟨*global* snarffd 233f⟩≡ (345a)
```
fdt  snarffd;
```

⟨main() *set some globals* 233g⟩+≡ (58) ◁229c
```
snarffd = open("/dev/snarf", OREAD|OCEXEC);
```

⟨*global* nsnarf 233h⟩≡ (345a)
```
int  nsnarf;
```

⟨*global* snarf 233i⟩≡ (345a)
```
Rune* snarf;
```

⟨*global* snarfversion 233j⟩≡ (338a)
```
int  snarfversion; /* updated each time it is written */
```

`putsnarf()`[233k] writes the rune buffer to `/dev/snarf` in 256-rune blocks to avoid hitting `fprint()`'s buffer limit. It opens a fresh file descriptor each time because `/dev/snarf` commits on close—writing to the already-open `snarffd` would not trigger the update. `getsnarf()` does the reverse: reads `/dev/snarf` into a byte buffer, then converts to runes.

⟨*function* putsnarf 233k⟩≡ (345a)
```
/*
 * /dev/snarf updates when the file is closed, so we must open our own
 * fd here rather than use snarffd
 */
void
putsnarf(void)
{
    int fd, i, n;

    if(snarffd<0 || nsnarf==0)
```

```
        return;
    fd = open("/dev/snarf", OWRITE);
    if(fd < 0)
        return;
    /* snarf buffer could be huge, so fprint will truncate; do it in blocks */
    for(i=0; i<nsnarf; i+=n){
        n = nsnarf-i;
        if(n >= 256)
            n = 256;
        if(fprint(fd, "%.*S", n, snarf+i) < 0)
            break;
    }
    close(fd);
}
```

Uses `nsnarf` 233h, `snarf` 233i, and `snarffd` 233f.

⟨*function* `getsnarf` 234a⟩≡ (345a)

```
void
getsnarf(void)
{
    int i, n, nb, nulls;
    char *sn, buf[1024];

    if(snarffd < 0)
        return;
    sn = nil;
    i = 0;
    seek(snarffd, 0, 0);
    while((n = read(snarffd, buf, sizeof buf)) > 0){
        sn = erealloc(sn, i+n+1);
        memmove(sn+i, buf, n);
        i += n;
        sn[i] = 0;
    }
    if(i > 0){
        snarf = runerealloc(snarf, i+1);
        cvttorunes(sn, i, snarf, &nb, &nsnarf, &nulls);
        free(sn);
    }
}
```

Uses `cvttorunes()` 285b, `nsnarf` 233h, `runerealloc` 284e, `snarf` 233i, and `snarffd` 233f.

13.5.2 Plumb

Plumbing is Plan 9's inter-application messaging system. When the user selects text and chooses "Plumb" from the middle-click menu, `rio` sends the text to the plumber daemon, which routes it to the appropriate application based on pattern rules (e.g., a filename goes to the editor, a URL to the browser).

Other systems have pieces of this. macOS has Services and "data detectors" (click a detected address or date and it opens in the right app); Windows and the Linux desktops route by file type and URL scheme (`xdg-open`, MIME associations, registered handlers). Plan 9's plumber is more general and more uniform: rather than file-type and URL-scheme tables scattered across the system, a single user-editable rules file matches *arbitrary* text and decides where it goes, and any program can send or receive on the plumb channels. One small, inspectable mechanism instead of a dozen special cases.

⟨`button2menu()` *cases* 234b⟩+≡ (198b) ◁230e

```
case Plumb:
    wplumb(w);
```

```
    break;
```
Uses Plumb-39 199c and wplumb() 235a.

wplumb()[235a] constructs a Plumbmsg[362] with the selected text (or, if nothing is selected, expands the selection to the surrounding whitespace-delimited word). The `click` attribute tells the receiving application where the cursor was within the expanded word, so it can interpret sub-parts (like line numbers in "file.c:42"). If `plumbsend()` fails, the cursor briefly flashes to the query cursor to indicate the error.

⟨*function* wplumb 235a⟩≡ (343b)
```
void
wplumb(Window *w)
{
    Plumbmsg *m;
    static int fd = -2;
    char buf[32];
    uint p0, p1;
    Cursor *c;

    if(fd == -2)
        fd = plumbopen("send", OWRITE|OCEXEC);
    if(fd < 0)
        return;
    m = emalloc(sizeof(Plumbmsg));
    m->src = estrdup("rio");
    m->dst = nil;
    m->wdir = estrdup(w->dir);
    m->type = estrdup("text");
    p0 = w->q0;
    p1 = w->q1;
    if(w->q1 > w->q0)
        m->attr = nil;
    else{
        while(p0>0 && w->r[p0-1]!=' ' && w->r[p0-1]!='\t' && w->r[p0-1]!='\n')
            p0--;
        while(p1<w->nr && w->r[p1]!=' ' && w->r[p1]!='\t' && w->r[p1]!='\n')
            p1++;
        sprint(buf, "click=%d", w->q0-p0);
        m->attr = plumbunpackattr(buf);
    }
    if(p1-p0 > messagesize-1024){
        plumbfree(m);
        return; /* too large for 9P */
    }
    m->data = runetobyte(w->r+p0, p1-p0, &m->ndata);
    if(plumbsend(fd, m) < 0){
        c = lastcursor;
        riosetcursor(&query, 1);
        sleep(300);
        riosetcursor(c, 1);
    }
    plumbfree(m);
}
```
Uses lastcursor 85a, messagesize 76a, plumbfree() 240b, plumbopen() 236c, plumbsend() 239b, plumbunpackattr() 240c, query 82d, riosetcursor() 85b, and runetobyte() 285d.

⟨*struct* Plumbmsg 235b⟩≡ (362)
```
struct Plumbmsg
{
    char  *src;
    char  *dst;
```

```
    char  *wdir;
    char  *type;
    Plumbattr  *attr;
    int   ndata;
    char  *data;
};
```

⟨*struct* `Plumbattr` 236a⟩≡ (362)

```
struct Plumbattr
{
    char  *name;
    char  *value;
    Plumbattr  *next;
};
```

⟨*function* `plumbsendtext` 236b⟩≡ (367b)

```
int
plumbsendtext(int fd, char *src, char *dst, char *wdir, char *data)
{
    Plumbmsg m;

    m.src = src;
    m.dst = dst;
    m.wdir = wdir;
    m.type = "text";
    m.attr = nil;
    m.ndata = strlen(data);
    m.data = data;
    return plumbsend(fd, &m);
}
```

Uses `plumbsend()` 239b.

Plumb messages

The plumbing library (`libplumb`) implements the message protocol. A plumb message is a simple text format: six newline-terminated header lines (source, destination, working directory, type, attributes, data length) followed by the raw data bytes. `plumbopen()`[236c] searches for the plumber in multiple locations: first `/mnt/plumb`, then `/mnt/term/mnt/plumb` (for remote terminals using `cpu`), and finally tries to mount `plumbsrv` from the environment as a last resort.

⟨*function* `plumbopen` 236c⟩≡ (367a)

```
int
plumbopen(char *name, int omode)
{
    int fd, f;
    char *s, *plumber;
    char buf[128], err[ERRMAX];

    if(name[0] == '/')
        return open(name, omode);

    /* find elusive plumber */
    if(access("/mnt/plumb/send", AWRITE) >= 0)
        plumber = "/mnt/plumb";
    else if(access("/mnt/term/mnt/plumb/send", AWRITE) >= 0)
        plumber = "/mnt/term/mnt/plumb";
    else{
        /* last resort: try mounting service */
        plumber = "/mnt/plumb";
```

```
        s = getenv("plumbsrv");
        if(s == nil)
            return -1;
        f = open(s, ORDWR);
        free(s);
        if(f < 0)
            return -1;
        if(mount(f, -1, "/mnt/plumb", MREPL, "") < 0){
            close(f);
            return -1;
        }
        if(access("/mnt/plumb/send", AWRITE) < 0)
            return -1;
    }

    snprint(buf, sizeof buf, "%s/%s", plumber, name);
    fd = open(buf, omode);
    if(fd >= 0)
        return fd;

    /* try creating port; used by non-standard plumb implementations */
    rerrstr(err, sizeof err);
    fd = create(buf, omode, 0600);
    if(fd >= 0)
        return fd;
    errstr(err, sizeof err);

    return -1;
}
```

⟨*function* Strlen 237a⟩≡ (367a)

```
static int
Strlen(char *s)
{
    if(s == nil)
        return 0;
    return strlen(s);
}
```

⟨*function* Strcpy 237b⟩≡ (367a)

```
static char*
Strcpy(char *s, char *t)
{
    if(t == nil)
        return s;
    return strcpy(s, t) + strlen(t);
}
```

⟨*function* quote 237c⟩≡ (367a)

```
/* quote attribute value, if necessary */
static char*
quote(char *s, char *buf, char *bufe)
{
    char *t;
    int c;

    if(s == nil){
        buf[0] = '\0';
        return buf;
    }
```

```
    if(strpbrk(s, " '=\t") == nil)
        return s;
    t = buf;
    *t++ = '\'';
    while(t < bufe-2){
        c = *s++;
        if(c == '\0')
            break;
        *t++ = c;
        if(c == '\'')
            *t++ = c;
    }
    *t++ = '\'';
    *t = '\0';
    return buf;
}
```

⟨*function* plumbpackattr 238a⟩≡ (367a)

```
char*
plumbpackattr(Plumbattr *attr)
{
    int n;
    Plumbattr *a;
    char *s, *t, *buf, *bufe;

    if(attr == nil)
        return nil;
    if((buf = malloc(4096)) == nil)
        return nil;
    bufe = buf + 4096;
    n = 0;
    for(a=attr; a!=nil; a=a->next)
        n += Strlen(a->name) + 1 + Strlen(quote(a->value, buf, bufe)) + 1;
    s = malloc(n);
    if(s == nil) {
        free(buf);
        return nil;
    }
    t = s;
    *t = '\0';
    for(a=attr; a!=nil; a=a->next){
        if(t != s)
            *t++ = ' ';
        strcpy(t, a->name);
        strcat(t, "=");
        strcat(t, quote(a->value, buf, bufe));
        t += strlen(t);
    }
    if(t > s+n)
        abort();
    free(buf);
    return s;
}
```

Uses `Strlen()` 237a and `quote()` 237c.

⟨*function* plumblookup 238b⟩≡ (367a)

```
char*
plumblookup(Plumbattr *attr, char *name)
{
    while(attr){
```

```
        if(strcmp(attr->name, name) == 0)
            return attr->value;
        attr = attr->next;
    }
    return nil;
}
```

⟨*function* `plumbpack` 239a⟩≡ (367a)

```
char*
plumbpack(Plumbmsg *m, int *np)
{
    int n, ndata;
    char *buf, *p, *attr;

    ndata = m->ndata;
    if(ndata < 0)
        ndata = Strlen(m->data);
    attr = plumbpackattr(m->attr);
    n = Strlen(m->src)+1 + Strlen(m->dst)+1 + Strlen(m->wdir)+1 +
        Strlen(m->type)+1 + Strlen(attr)+1 + 16 + ndata;
    buf = malloc(n+1); /* +1 for '\0' */
    if(buf == nil){
        free(attr);
        return nil;
    }
    p = Strcpy(buf, m->src);
    *p++ = '\n';
    p = Strcpy(p, m->dst);
    *p++ = '\n';
    p = Strcpy(p, m->wdir);
    *p++ = '\n';
    p = Strcpy(p, m->type);
    *p++ = '\n';
    p = Strcpy(p, attr);
    *p++ = '\n';
    p += sprint(p, "%d\n", ndata);
    memmove(p, m->data, ndata);
    *np = (p-buf)+ndata;
    buf[*np] = '\0'; /* null terminate just in case */
    if(*np >= n+1)
        abort();
    free(attr);
    return buf;
}
```

Uses `Strcpy()` 237b, `Strlen()` 237a, and `plumbpackattr()` 238a.

⟨*function* `plumbsend` 239b⟩≡ (367a)

```
int
plumbsend(int fd, Plumbmsg *m)
{
    char *buf;
    int n;

    buf = plumbpack(m, &n);
    if(buf == nil)
        return -1;
    n = write(fd, buf, n);
    free(buf);
    return n;
}
```

Uses plumbpack() 239a.

⟨*function* plumbline 240a⟩≡ (367a)

```
static int
plumbline(char **linep, char *buf, int i, int n, int *bad)
{
    int starti;
    char *p;

    starti = i;
    while(i<n && buf[i]!='\n')
        i++;
    if(i == n)
        *bad = 1;
    else{
        p = malloc((i-starti) + 1);
        if(p == nil)
            *bad = 1;
        else{
            memmove(p, buf+starti, i-starti);
            p[i-starti] = '\0';
        }
        *linep = p;
        i++;
    }
    return i;
}
```

⟨*function* plumbfree 240b⟩≡ (367a)

```
void
plumbfree(Plumbmsg *m)
{
    Plumbattr *a, *next;

    free(m->src);
    free(m->dst);
    free(m->wdir);
    free(m->type);
    for(a=m->attr; a!=nil; a=next){
        next = a->next;
        free(a->name);
        free(a->value);
        free(a);
    }
    free(m->data);
    free(m);
}
```

⟨*function* plumbunpackattr 240c⟩≡ (367a)

```
Plumbattr*
plumbunpackattr(char *p)
{
    Plumbattr *attr, *prev, *a;
    char *q, *v, *buf, *bufe;
    int c, quoting;

    buf = malloc(4096);
    if(buf == nil)
        return nil;
    bufe = buf + 4096;
```

```
attr = prev = nil;
while(*p!='\0' && *p!='\n'){
    while(*p==' ' || *p=='\t')
        p++;
    if(*p == '\0')
        break;
    for(q=p; *q!='\0' && *q!='\n' && *q!=' ' && *q!='\t'; q++)
        if(*q == '=')
            break;
    if(*q != '=')
        break; /* malformed attribute */
    a = malloc(sizeof(Plumbattr));
    if(a == nil)
        break;
    a->name = malloc(q-p+1);
    if(a->name == nil){
        free(a);
        break;
    }
    memmove(a->name, p, q-p);
    a->name[q-p] = '\0';
    /* process quotes in value */
    q++; /* skip '=' */
    v = buf;
    quoting = 0;
    while(*q!='\0' && *q!='\n'){
        if(v >= bufe)
            break;
        c = *q++;
        if(quoting){
            if(c == '\''){
                if(*q == '\'')
                    q++;
                else{
                    quoting = 0;
                    continue;
                }
            }
        }else{
            if(c==' ' || c=='\t')
                break;
            if(c == '\''){
                quoting = 1;
                continue;
            }
        }
        *v++ = c;
    }
    a->value = malloc(v-buf+1);
    if(a->value == nil){
        free(a->name);
        free(a);
        break;
    }
    memmove(a->value, buf, v-buf);
    a->value[v-buf] = '\0';
    a->next = nil;
    if(prev == nil)
        attr = a;
    else
```

```
            prev->next = a;
        prev = a;
        p = q;
    }
    free(buf);
    return attr;
}
```

⟨*function* plumbaddattr 242a⟩≡ (367a)

```
Plumbattr*
plumbaddattr(Plumbattr *attr, Plumbattr *new)
{
    Plumbattr *l;

    l = attr;
    if(l == nil)
        return new;
    while(l->next != nil)
        l = l->next;
    l->next = new;
    return attr;
}
```

⟨*function* plumbdelattr 242b⟩≡ (367a)

```
Plumbattr*
plumbdelattr(Plumbattr *attr, char *name)
{
    Plumbattr *l, *prev;

    prev = nil;
    for(l=attr; l!=nil; l=l->next){
        if(strcmp(name, l->name) == 0)
            break;
        prev = l;
    }
    if(l == nil)
        return nil;
    if(prev)
        prev->next = l->next;
    else
        attr = l->next;
    free(l->name);
    free(l->value);
    free(l);
    return attr;
}
```

⟨*function* plumbunpackpartial 242c⟩≡ (367a)

```
Plumbmsg*
plumbunpackpartial(char *buf, int n, int *morep)
{
    Plumbmsg *m;
    int i, bad;
    char *ntext, *attr;

    m = malloc(sizeof(Plumbmsg));
    if(m == nil)
        return nil;
    memset(m, 0, sizeof(Plumbmsg));
    if(morep != nil)
```

```
            *morep = 0;
        bad = 0;
        i = plumbline(&m->src, buf, 0, n, &bad);
        i = plumbline(&m->dst, buf, i, n, &bad);
        i = plumbline(&m->wdir, buf, i, n, &bad);
        i = plumbline(&m->type, buf, i, n, &bad);
        i = plumbline(&attr, buf, i, n, &bad);
        i = plumbline(&ntext, buf, i, n, &bad);
        if(bad){
            plumbfree(m);
            return nil;
        }
        m->attr = plumbunpackattr(attr);
        free(attr);
        m->ndata = atoi(ntext);
        if(m->ndata != n-i){
            bad = 1;
            if(morep!=nil && m->ndata>n-i)
                *morep = m->ndata - (n-i);
        }
        free(ntext);
        if(!bad){
            m->data = malloc(n-i+1); /* +1 for '\0' */
            if(m->data == nil)
                bad = 1;
            else{
                memmove(m->data, buf+i, m->ndata);
                m->ndata = n-i;
                /* null-terminate in case it's text */
                m->data[m->ndata] = '\0';
            }
        }
        if(bad){
            plumbfree(m);
            m = nil;
        }
        return m;
    }
```

Uses `plumbfree()` 240b, `plumbline()` 240a, and `plumbunpackattr()` 240c.

⟨*function* `plumbunpack` 243a⟩≡ (367a)

```
    Plumbmsg*
    plumbunpack(char *buf, int n)
    {
        return plumbunpackpartial(buf, n, nil);
    }
```

Uses `plumbunpackpartial()` 242c.

⟨*function* `plumbrecv` 243b⟩≡ (367a)

```
    Plumbmsg*
    plumbrecv(int fd)
    {
        char *buf;
        Plumbmsg *m;
        int n, more;

        buf = malloc(8192);
        if(buf == nil)
            return nil;
        n = read(fd, buf, 8192);
```

```
    m = nil;
    if(n > 0){
        m = plumbunpackpartial(buf, n, &more);
        if(m==nil && more>0){
            /* we now know how many more bytes to read for complete message */
            buf = realloc(buf, n+more);
            if(buf == nil)
                return nil;
            if(readn(fd, buf+n, more) == more)
                m = plumbunpackpartial(buf, n+more, nil);
        }
    }
    free(buf);
    return m;
}
```

Uses `plumbunpackpartial()` 242c.

Event-based plumbing

The event-based plumbing interface (`eplumb()`) integrates plumb messages into the graphics event loop. Because a plumb message can be larger than a single 9P read, the `EQueue`[244a] structure buffers partial messages until all bytes arrive. `plumbevent()`[245b] first checks for an existing partial message for this event ID, and if not found, tries to unpack a complete message—stashing the buffer in the queue if the message is incomplete.

⟨*struct* `EQueue` 244a⟩≡ (366c)

```
struct EQueue
{
    int  id;
    char  *buf;
    int  nbuf;
    EQueue *next;
};
```

⟨*global* `equeue` 244b⟩≡ (366c)

```
static EQueue *equeue;
```

⟨*global* `eqlock` 244c⟩≡ (366c)

```
static Lock  eqlock;
```

⟨*function* `partial` 244d⟩≡ (366c)

```
static
int
partial(int id, Event *e, uchar *b, int n)
{
    EQueue *eq, *p;
    int nmore;

    lock(&eqlock);
    for(eq = equeue; eq != nil; eq = eq->next)
        if(eq->id == id)
            break;
    unlock(&eqlock);
    if(eq == nil)
        return 0;
    /* partial message exists for this id */
    eq->buf = realloc(eq->buf, eq->nbuf+n);
    if(eq->buf == nil)
        drawerror(display, "eplumb: cannot allocate buffer");
    memmove(eq->buf+eq->nbuf, b, n);
```

```
    eq->nbuf += n;
    e->v = plumbunpackpartial((char*)eq->buf, eq->nbuf, &nmore);
    if(nmore == 0){ /* no more to read in this message */
        lock(&eqlock);
        if(eq == equeue)
            equeue = eq->next;
        else{
            for(p = equeue; p!=nil && p->next!=eq; p = p->next)
                ;
            if(p == nil)
                drawerror(display, "eplumb: bad event queue");
            p->next = eq->next;
        }
        unlock(&eqlock);
        free(eq->buf);
        free(eq);
    }
    return 1;
}
```

Uses `eqlock-154` 244c, `equeue-153` 244b, and `plumbunpackpartial()` 242c.

⟨*function* `addpartial` 245a⟩≡ (366c)

```
static
void
addpartial(int id, char *b, int n)
{
    EQueue *eq;

    eq = malloc(sizeof(EQueue));
    if(eq == nil)
        return;
    eq->id = id;
    eq->nbuf = n;
    eq->buf = malloc(n);
    if(eq->buf == nil){
        free(eq);
        return;
    }
    memmove(eq->buf, b, n);
    lock(&eqlock);
    eq->next = equeue;
    equeue = eq;
    unlock(&eqlock);
}
```

Uses `eqlock-154` 244c and `equeue-153` 244b.

⟨*function* `plumbevent` 245b⟩≡ (366c)

```
static
int
plumbevent(int id, Event *e, uchar *b, int n)
{
    int nmore;

    if(partial(id, e, b, n) == 0){
        /* no partial message already waiting for this id */
        e->v = plumbunpackpartial((char*)b, n, &nmore);
        if(nmore > 0) /* incomplete message */
            addpartial(id, (char*)b, n);
    }
    if(e->v == nil)
```

```
        return 0;
    return id;
}
```

Uses `addpartial()` 245a, `partial()` 244d, and `plumbunpackpartial()` 242c.

⟨*function* `eplumb` 246a⟩≡ (366c)

```
int
eplumb(int key, char *port)
{
    int fd;

    fd = plumbopen(port, OREAD|OCEXEC);
    if(fd < 0)
        return -1;
    return estartfn(key, fd, 8192, plumbevent);
}
```

Uses `plumbevent()` 245b and `plumbopen()` 236c.

13.5.3 Auto complete

Pressing Ctrl-F or Insert triggers filename completion, similar to Tab completion in UNIX shells. `namecomplete()`[247a] works backward from the cursor to extract the current word and its path prefix, resolves relative paths against the window's working directory (the per-window `w->dir`, which `rio` also exposes as the `/mnt/wsys/wdir` file), and calls `complete()` from `libcomplete` to find matches. If there is a unique common prefix to advance, the text is inserted. If multiple matches exist and no progress can be made, the candidates are displayed inline before the current command line.

⟨`wkeyctl()` *special key cases and no special mode* 246b⟩+≡ (187a) ◁194b

```
case 0x06: /* ^F: file name completion */
case Kins:  /* Insert: file name completion */
    rp = namecomplete(w);
    if(rp == nil)
        return;
    nr = runestrlen(rp);
    q0 = w->q0;
    q0 = winsert(w, rp, nr, q0);
    wshow(w, q0+nr);
    free(rp);
    return;
```

Uses `namecomplete()` 247a, `winsert()` 177a, and `wshow()` 179b.

The `Completion`[361a] struct holds the result of one completion attempt: `advance` says whether the typed prefix can be extended, `complete` whether it now names a real file or directory, `string` the text to insert, and `filename` the candidate list to display when there is no unique advance.

⟨*struct* `Completion` 246c⟩≡ (361a)

```
struct Completion{
    uchar advance; /* whether forward progress has been made */
    uchar complete; /* whether the completion now represents a file or directory */
    char *string; /* the string to advance, suffixed " " or "/" for file or directory */
    int nmatch;  /* number of files that matched */
    int nfile;  /* number of files returned */
    char **filename; /* their names */
};
```

namecomplete() does the work: it walks back from the cursor to find the word being typed, splits it into a directory path and a partial name, and hands them to complete() to fill in the Completion above.

⟨*function* namecomplete 247a⟩≡ (343b)

```
Rune*
namecomplete(Window *w)
{
    int nstr, npath;
    Rune *rp, *path, *str;
    Completion *c;
    char *s, *dir, *root;

    /* control-f: filename completion; works back to white space or / */
    if(w->q0<w->nr && w->r[w->q0]>' ') /* must be at end of word */
        return nil;
    nstr = windfilewidth(w, w->q0, true);
    str = runemalloc(nstr);
    runemove(str, w->r+(w->q0-nstr), nstr);
    npath = windfilewidth(w, w->q0-nstr, false);
    path = runemalloc(npath);
    runemove(path, w->r+(w->q0-nstr-npath), npath);
    rp = nil;

    /* is path rooted? if not, we need to make it relative to window path */
    if(npath>0 && path[0]=='/'){
        dir = malloc(UTFmax*npath+1);
        sprint(dir, "%.*S", npath, path);
    }else{
        if(strcmp(w->dir, "") == 0)
            root = ".";
        else
            root = w->dir;
        dir = malloc(strlen(root)+1+UTFmax*npath+1);
        sprint(dir, "%s/%.*S", root, npath, path);
    }
    dir = cleanname(dir);

    s = smprint("%.*S", nstr, str);
    c = complete(dir, s);
    free(s);
    if(c == nil)
        goto Return;

    if(!c->advance)
        showcandidates(w, c);

    if(c->advance)
        rp = runesmprint("%s", c->string);

  Return:
    freecompletion(c);
    free(dir);
    free(path);
    free(str);
    return rp;
}
```

Uses complete() 249, freecompletion() 251a, runemalloc 284d, runemove 285a, showcandidates() 247b, and windfilewidth() 248.

⟨*function* showcandidates 247b⟩≡ (343b)

```
void
showcandidates(Window *w, Completion *c)
{
    int i;
    Fmt f;
    Rune *rp;
    uint nr, qline, q0;
    char *s;

    runefmtstrinit(&f);
    if (c->nmatch == 0)
        s = "[no matches in ";
    else
        s = "[";
    if(c->nfile > 32)
        fmtprint(&f, "%s%d files]\n", s, c->nfile);
    else{
        fmtprint(&f, "%s", s);
        for(i=0; i<c->nfile; i++){
            if(i > 0)
                fmtprint(&f, " ");
            fmtprint(&f, "%s", c->filename[i]);
        }
        fmtprint(&f, "]\n");
    }
    /* place text at beginning of line before host point */
    qline = w->qh;
    while(qline>0 && w->r[qline-1] != '\n')
        qline--;

    rp = runefmtstrflush(&f);
    nr = runestrlen(rp);

    q0 = w->q0;
    q0 += winsert(w, rp, runestrlen(rp), qline) - qline;
    free(rp);
    wsetselect(w, q0+nr, q0+nr);
}
```

Uses `winsert()` 177a and `wsetselect()` 185.

⟨*function* `windfilewidth` 248⟩≡ (343b)

```
int
windfilewidth(Window *w, uint q0, int oneelement)
{
    uint q;
    Rune r;

    q = q0;
    while(q > 0){
        r = w->r[q-1];
        if(r<=' ')
            break;
        if(oneelement && r=='/')
            break;
        --q;
    }
    return q0-q;
}
```

complete()

complete()[249] is a general-purpose filename completion function from libcomplete. It reads the directory, finds entries matching the prefix, computes the longest common prefix among all matches, and returns a Completion[361a] struct. The advance flag indicates whether forward progress was made; complete means the match is unique. A completed filename gets a trailing "/" for directories or " " for files. When no matches exist, it returns all directory entries so the caller can show them as suggestions.

⟨*function* complete 249⟩≡ (363a)
```
Completion*
complete(char *dir, char *s)
{
    long i, l, n, nfile, len, nbytes;
    int fd, minlen;
    Dir *dirp;
    char **name, *p;
    ulong* mode;
    Completion *c;

    if(strchr(s, '/') != nil){
        werrstr("slash character in name argument to complete()");
        return nil;
    }

    fd = open(dir, OREAD);
    if(fd < 0)
        return nil;

    n = dirreadall(fd, &dirp);
    if(n <= 0){
        close(fd);
        return nil;
    }

    /* find longest string, for allocation */
    len = 0;
    for(i=0; i<n; i++){
        l = strlen(dirp[i].name) + 1 + 1; /* +1 for /   +1 for \0 */
        if(l > len)
            len = l;
    }

    name = malloc(n*sizeof(char*));
    mode = malloc(n*sizeof(ulong));
    c = malloc(sizeof(Completion) + len);
    if(name == nil || mode == nil || c == nil)
        goto Return;
    memset(c, 0, sizeof(Completion));

    /* find the matches */
    len = strlen(s);
    nfile = 0;
    minlen = 1000000;
    for(i=0; i<n; i++)
        if(strncmp(s, dirp[i].name, len) == 0){
            name[nfile] = dirp[i].name;
            mode[nfile] = dirp[i].mode;
            if(minlen > strlen(dirp[i].name))
                minlen = strlen(dirp[i].name);
            nfile++;
```

```
        }

    if(nfile > 0) {
        /* report interesting results */
        /* trim length back to longest common initial string */
        for(i=1; i<nfile; i++)
            minlen = longestprefixlength(name[0], name[i], minlen);

        /* build the answer */
        c->complete = (nfile == 1);
        c->advance = c->complete || (minlen > len);
        c->string = (char*)(c+1);
        memmove(c->string, name[0]+len, minlen-len);
        if(c->complete)
            c->string[minlen++ - len] = (mode[0]&DMDIR)? '/' : ' ';
        c->string[minlen - len] = '\0';
        c->nmatch = nfile;
    } else {
        /* no match, so return all possible strings */
        for(i=0; i<n; i++){
            name[i] = dirp[i].name;
            mode[i] = dirp[i].mode;
        }
        nfile = n;
        c->nmatch = 0;
    }

    /* attach list of names */
    nbytes = nfile * sizeof(char*);
    for(i=0; i<nfile; i++)
        nbytes += strlen(name[i]) + 1 + 1;
    c->filename = malloc(nbytes);
    if(c->filename == nil)
        goto Return;
    p = (char*)(c->filename + nfile);
    for(i=0; i<nfile; i++){
        c->filename[i] = p;
        strcpy(p, name[i]);
        p += strlen(p);
        if(mode[i] & DMDIR)
            *p++ = '/';
        *p++ = '\0';
    }
    c->nfile = nfile;
    qsort(c->filename, c->nfile, sizeof(c->filename[0]), strpcmp);

  Return:
    free(name);
    free(mode);
    free(dirp);
    close(fd);
    return c;
}
```

Uses `longestprefixlength()` 250 and `strpcmp()` 251b.

⟨*function* `longestprefixlength` 250⟩≡ (363a)

```
static int
longestprefixlength(char *a, char *b, int n)
{
    int i, w;
```

```
    Rune ra, rb;

    for(i=0; i<n; i+=w){
        w = chartorune(&ra, a);
        chartorune(&rb, b);
        if(ra != rb)
            break;
        a += w;
        b += w;
    }
    return i;
}
```

⟨*function* `freecompletion` 251a⟩≡ (363a)

```
void
freecompletion(Completion *c)
{
    if(c){
        free(c->filename);
        free(c);
    }
}
```

⟨*function* `strpcmp` 251b⟩≡ (363a)

```
static int
strpcmp(const void *va, const void *vb)
{
    char *a, *b;

    a = *(char**)va;
    b = *(char**)vb;
    return strcmp(a, b);
}
```

/mnt/wsys/wdir

Each window tracks a working directory in `Window.dir`[96g], exposed as `/mnt/wsys/wdir`. This is used by both filename completion (to resolve relative paths) and plumbing (as the `wdir` field in plumb messages). Programs can write to this file to update the window's notion of the current directory—supporting relative or absolute paths, with `cleanname()` normalizing the result.

⟨`Qxxx` *other cases* 251c⟩+≡ (122d) ◁232a

```
Qwdir,
```

⟨*dirtab array elements* 251d⟩+≡ (123b) ◁232b

```
{ "wdir",  QTFILE, Qwdir,  0600 },
```

Uses `Qwdir` 251c.

⟨`xfidread()` *cases* 251e⟩+≡ (136b) ◁232g

```
case Qwdir:
    t = estrdup(w->dir);
    n = strlen(t);
    goto Text;
```

Uses `Qwdir` 251c.

⟨xfidwrite() *cases* 252a⟩+≡ (137b) ◁233b

```
case Qwdir:
    if(cnt == 0)
        break;
    if(req->data[cnt-1] == '\n'){
        if(cnt == 1)
            break;
        req->data[cnt-1] = '\0';
    }
    /* assume data comes in a single write */
    /*
     * Problem: programs like dossrv, ftp produce illegal UTF;
     * we must cope by converting it first.
     */
    snprint(buf, sizeof buf, "%.*s", cnt, req->data);
    if(buf[0] == '/'){
        free(w->dir);
        w->dir = estrdup(buf);
    }else{
        p = emalloc(strlen(w->dir) + 1 + strlen(buf) + 1);
        sprint(p, "%s/%s", w->dir, buf);
        free(w->dir);
        w->dir = cleanname(p);
    }
    break;
```

Uses `Qwdir` 251c.

13.5.4 Word selection

The last of the terminal's editing features is mouse selection.

`wselect()`[253] handles mouse-based text selection, including double-click to select words and bracket matching. A double-click is detected when two clicks happen within 500ms on the same window at the same position. While the button is held after selection, chording with button 2 cuts and button 3 pastes—a classic Plan 9 interaction pattern inherited from `sam` and `acme`.

Before `wselect()` itself, a couple of file-scope globals carry the state double-click detection needs across calls: which window was last clicked, and when.

⟨*global* clickwin 252b⟩≡ (343b)

```
static Window *clickwin;
```

⟨*global* clickmsec 252c⟩≡ (343b)

```
static uint clickmsec;
```

⟨*global* selectwin 252d⟩≡ (343b)

```
static Window *selectwin;
```

⟨*global* selectq 252e⟩≡ (343b)

```
static uint selectq;
```

The `wselect`[253] function handles button 1 click in a textual window. It first calls `frselect` to do the interactive mouse-tracking selection loop, then translates the frame-local selection (`frm.p0`/`frm.p1`) back to buffer-wide coordinates by adding `org`. If the selection is empty (a single click with no drag), the output point `qh` may be advanced to `q0`. This is the "click past the prompt" behavior: if the user clicks after the output point, `qh` advances so that subsequent application reads will include the text between the old and new positions. This lets the user "adopt" text by clicking past it—a distinctive `rio` interaction. If the click is a double-click (detected by comparing with the previous click time and position), the selection is expanded to a whole word

using `wordclick`. The word boundaries are determined by the character class tables in the Advanced Topics chapter.

⟨*function* `wselect` 253⟩≡ (343b)
```
void
wselect(Window *w)
{
    uint q0, q1;
    int b, x, y, first;
    Frame *frm = &w->frm;

    first = 1;
    selectwin = w;
    /*
     * Double-click immediately if it might make sense.
     */
    b = w->mc.m.buttons;
    q0 = w->q0;
    q1 = w->q1;
    selectq = w->org + frcharofpt(frm, w->mc.m.xy);
    if(clickwin==w && w->mc.m.msec-clickmsec<500)
    if(q0==q1 && selectq==w->q0){
        wdoubleclick(w, &q0, &q1);
        wsetselect(w, q0, q1);
        flushimage(display, 1);
        x = w->mc.m.xy.x;
        y = w->mc.m.xy.y;
        /* stay here until something interesting happens */
        do
            readmouse(&w->mc);
        while(w->mc.m.buttons==b && abs(w->mc.m.xy.x-x)<3 && abs(w->mc.m.xy.y-y)<3);
        w->mc.m.xy.x = x; /* in case we're calling frselect */
        w->mc.m.xy.y = y;
        q0 = w->q0; /* may have changed */
        q1 = w->q1;
        selectq = q0;
    }
    if(w->mc.m.buttons == b){
        frm->scroll = framescroll;
        frselect(frm, &w->mc);
        /* horrible botch: while asleep, may have lost selection altogether */
        if(selectq > w->nr)
            selectq = w->org + frm->p0;
        frm->scroll = nil;
        if(selectq < w->org)
            q0 = selectq;
        else
            q0 = w->org + frm->p0;
        if(selectq > w->org + frm->nchars)
            q1 = selectq;
        else
            q1 = w->org + frm->p1;
    }
    if(q0 == q1){
        if(q0==w->q0 && clickwin==w && w->mc.m.msec-clickmsec<500){
            wdoubleclick(w, &q0, &q1);
            clickwin = nil;
        }else{
            clickwin = w;
            clickmsec = w->mc.m.msec;
```

```
        }
    }else
        clickwin = nil;
    wsetselect(w, q0, q1);
    flushimage(display, 1);
    while(w->mc.m.buttons){
        w->mc.m.msec = 0;
        b = w->mc.m.buttons;
        if(b & 6){
            if(b & 2){
                wsnarf(w);
                wcut(w);
            }else{
                if(first){
                    first = 0;
                    getsnarf();
                }
                wpaste(w);
            }
        }
        wscrdraw(w);
        flushimage(display, 1);
        while(w->mc.m.buttons == b)
            readmouse(&w->mc);
        clickwin = nil;
    }
}
```

Uses `clickmsec-43` 252c, `clickwin-42` 252b, `framescroll()` 204a, `frcharofpt()` 297c, `frselect()` 300, `getsnarf()` 234a, `selectq-45` 252e, `selectwin-44` 252d, `wcut()` 231b, `wdoubleclick()` 255a, `wpaste()` 231c, `wscrdraw()` 180b, `wsetselect()` 185, and `wsnarf()` 231a.

⟨*global* `left1` 254a⟩≡ (343b)

```
static Rune left1[] =  { L'{', L'[', L'(', L'<', L'«', 0 };
```

⟨*global* `right1` 254b⟩≡ (343b)

```
static Rune right1[] = { L'}', L']', L')', L'>', L'»', 0 };
```

⟨*global* `left2` 254c⟩≡ (343b)

```
static Rune left2[] =  { L'\n', 0 };
```

⟨*global* `left3` 254d⟩≡ (343b)

```
static Rune left3[] =  { L'\'', L'"', L'`', 0 };
```

⟨*global* `left` 254e⟩≡ (343b)

```
Rune *left[] = {
    left1,
    left2,
    left3,
    nil
};
```

Uses `left1-46` 254a, `left2-48` 254c, and `left3-49` 254d.

⟨*global* `right` 254f⟩≡ (343b)

```
Rune *right[] = {
    right1,
    left2,
    left3,
    nil
};
```

Uses `left2-48` 254c, `left3-49` 254d, and `right1-47` 254b.

`wdoubleclick()`[255a] implements smart selection: it first tries bracket matching (braces, brackets, parentheses, angle brackets, guillemets), then line matching on newlines, then quote matching (single, double, backtick). If none of these match, it falls back to extending the selection to cover the full alphanumeric word. The `left/right` arrays define the matching pairs—note that `left2/left3` are shared between left and right because newlines and quotes are their own closing delimiters.

⟨*function* `wdoubleclick` 255a⟩≡ (343b)

```
void
wdoubleclick(Window *w, uint *q0, uint *q1)
{
    int c, i;
    Rune *r, *l, *p;
    uint q;

    for(i=0; left[i]!=nil; i++){
        q = *q0;
        l = left[i];
        r = right[i];
        /* try matching character to left, looking right */
        if(q == 0)
            c = '\n';
        else
            c = w->r[q-1];
        p = strrune(l, c);
        if(p != nil){
            if(wclickmatch(w, c, r[p-l], 1, &q))
                *q1 = q-(c!='\n');
            return;
        }
        /* try matching character to right, looking left */
        if(q == w->nr)
            c = '\n';
        else
            c = w->r[q];
        p = strrune(r, c);
        if(p != nil){
            if(wclickmatch(w, c, l[p-r], -1, &q)){
                *q1 = *q0+(*q0<w->nr && c=='\n');
                *q0 = q;
                if(c!='\n' || q!=0 || w->r[0]=='\n')
                    (*q0)++;
            }
            return;
        }
    }
    /* try filling out word to right */
    while(*q1<w->nr && isalnum(w->r[*q1]))
        (*q1)++;
    /* try filling out word to left */
    while(*q0>0 && isalnum(w->r[*q0-1]))
        (*q0)--;
}
```

Uses `isalnum()` 284c, `left` 254e, `right` 254f, `strrune()` 285c, and `wclickmatch()` 255b.

`wclickmatch()` walks the text in the given direction, counting nested delimiters. It handles nesting correctly: matching "{" while inside "{...{...}...}" finds the outer brace. For newlines, a nest count of 1 at the boundary is accepted, which allows double-clicking at the start of a line to select the whole line.

⟨*function* `wclickmatch` 255b⟩≡ (343b)

```
int
```

```
wclickmatch(Window *w, int cl, int cr, int dir, uint *q)
{
    Rune c;
    int nest;

    nest = 1;
    for(;;){
        if(dir > 0){
            if(*q == w->nr)
                break;
            c = w->r[*q];
            (*q)++;
        }else{
            if(*q == 0)
                break;
            (*q)--;
            c = w->r[*q];
        }
        if(c == cr){
            if(--nest==0)
                return 1;
        }else if(c == cl)
            nest++;
    }
    return cl=='\n' && nest==1;
}
```

13.6 External access

So far, every program that talked to `rio`'s filesystem was a child process that inherited the pipe's file descriptor. This section covers the two ways a process can reach a *running* `rio` from the outside, both built on Plan 9's `/srv`: the `/srv/rio.<user>.<pid>` mount, through which any program can open a window, and the `/srv/riowctl.<user>.<pid>` channel, used to drive `rio` from the command line.

13.6.1 External mount: `/srv/rio.<user>.<pid>`

Until now, we have seen `filsysmount()`[99c] called from within `rio` itself, in a child process that inherits the pipe's file descriptor. But what about external processes that were not spawned by `rio`? They need a bootstrap mechanism to get a handle on `rio`'s filesystem. This is where Plan 9's `/srv` comes in. `filsysinit()`[61c] publishes the client end of the pipe in `/srv/rio.<user>.<pid>`—Plan 9's equivalent of a named pipe. Any process can open this file, mount it, and gain access to `rio`'s filesystem, even if it is not a child of `rio`. By mounting with the spec "new ...", an external program can create a new window for itself—this is what `newwindow()` from the Graphics book [Pad16c] uses.

The `srvpipe`[256a] global holds the path string `/srv/rio.<user>.<pid>`, personalized with the username and process ID so multiple `rio` instances don't collide.

⟨*global srvpipe (rio/fsys.c)* 256a⟩≡ (346b)

```
char srvpipe[64];
```

⟨`filsysinit()` *srv pipe* 256b⟩≡ (61c)

```
/*
 * Post srv pipe
 */
sprint(srvpipe, "/srv/rio.%s.%d", fs->user, pid);
post(srvpipe, "wsys", fs->cfd);
```

Uses `post()` 257a and `srvpipe` 256a.

The `post()` function works with Plan 9's `/srv` device (`#s`): it creates a file in `/srv`, writes a file descriptor number into it as an ASCII string, and sets `ORCLOSE` so the entry is removed when `rio` exits. The `#s` driver is special—when another process opens this file, the kernel gives it a new file descriptor connected to the same pipe endpoint. The `putenv()` call also publishes the path in the environment so child processes can find it by name.

⟨*function* `post` 257a⟩≡ (346b)
```
void
post(char *name, char *envname, fdt srvfd)
{
    fdt fd;
    char buf[32];

    fd = create(name, OWRITE|ORCLOSE|OCEXEC, 0600);
    if(fd < 0)
        error(name);
    sprint(buf, "%d", srvfd);
    if(write(fd, buf, strlen(buf)) != strlen(buf))
        error("srv write");

    putenv(envname, name);
}
```
Uses `error()` 282c.

When an external client attaches with a spec string starting with "new", `xfidattach()` parses it with the same `parsewctl()`[263c] function used for `/dev/wctl` commands. This allows the client to specify the window rectangle, whether it should be hidden, and whether to scroll—the same parameters as the "new" wctl command.

⟨`xfidattach()` *other locals* 257b⟩+≡ (127c) ◁128c
```
Rectangle r;
int pid;
bool hideit = false;
bool scrollit;
char *dir, errbuf[ERRMAX];
Image *i;
```

⟨`xfidattach()` *if mount "new ..."* 257c⟩≡ (127c)
```
if(strncmp(x->req.aname, "new", 3) == 0){ /* new -dx -dy - new syntax, as in wctl */
    pid = 0;
    if(parsewctl(nil, ZR, &r, &pid, nil, &hideit, &scrollit, &dir, x->req.aname, errbuf) < 0)
        err = errbuf;
    else {
        if(!goodrect(r))
            err = Ebadrect;
        else{
            if(hideit)
                i = allocimage(display, r, view->chan, false, DWhite);
            else
                i = allocwindow(desktop, r, Refbackup, DWhite);
            if(i){
                border(i, r, Selborder, display->black, ZP);
                if(pid == 0)
                    pid = -1; /* make sure we don't pop a shell! - UGH */
                w = new(i, hideit, scrolling, pid, nil, nil, nil);
                flushimage(display, 1);
                newlymade = true;
            }else
                err = Ewindow;
        }
   }
```

```
}
```

Uses Ebadrect 281i, Ewindow 281j, Selborder 67, desktop 48d, goodrect() 261b, new() 92c, and scrolling 225c.

The pid is set to -1 to prevent rio from spawning a shell in the new window, since the external program will use it directly.

13.6.2 Command-line control: /srv/riowctl.<user>.<pid>

While /srv/rio.<user>.<pid> provides the full 9P filesystem interface, /srv/riowctl.<user>.<pid> is a simpler text-based control channel. An external process can write wctl commands (like "new") as plain text strings to create and manipulate windows from the command line, without needing to speak the 9P protocol. The architecture uses the same proc/thread split seen elsewhere in rio: wctlproc()[259a] runs in its own process doing blocking reads on the pipe, and sends the text over a channel to wctlthread()[259b] in main-proc where it is safe to manipulate window state.

⟨*global* wctlfd 258a⟩≡ (338a)

```
fdt wctlfd;
```

⟨*global srvwctl (rio/fsys.c)* 258b⟩≡ (346b)

```
char srvwctl[64];
```

The initialization creates the pipe, publishes it in /srv, then starts both the reading process and the dispatching thread.

⟨filsysinit() *wctl pipe, process, and thread creation* 258c⟩≡ (61c)

```
/*
 * Create and post wctl pipe
 */
⟨filsysinit() create wctl pipe 258e⟩

/*
 * Start server processes
 */
⟨filsysinit() create wctl process and thread 258f⟩
```

⟨filsysinit() *other locals* 258d⟩≡ (61c) 269e ▷

```
fdt p0;
// chan<??> (listener = ??, sender = ??)
Channel *c;
```

⟨filsysinit() *create wctl pipe* 258e⟩≡ (258c)

```
if(cexecpipe(&p0, &wctlfd) < 0)
    goto Rescue;
sprint(srvwctl, "/srv/riowctl.%s.%d", fs->user, pid);
post(srvwctl, "wctl", p0);
close(p0);
```

Uses cexecpipe() 62a, post() 257a, srvwctl 258b, and wctlfd 258a.

Why not just use /dev/wctl? Because /dev/wctl is only available to processes that have already mounted rio's filesystem. The /srv/riowctl pipe serves as a bootstrap for processes that have not (or cannot) mount rio.

⟨filsysinit() *create wctl process and thread* 258f⟩≡ (258c)

```
c = chancreate(sizeof(char*), 0);
if(c == nil)
    error("wctl channel");

proccreate(wctlproc, c, 4096);
threadcreate(wctlthread, c, 4096);
```

Uses error() 282c, wctlproc() 259a, and wctlthread() 259b.

wctlproc()

`wctlproc()`[259a] is the reading half: it sits in its own process doing blocking `read()` calls on the pipe, and forwards each complete command string over the channel. The `eofs` counter tolerates a burst of zero-length reads (which can happen with pipe semantics) before giving up.

⟨*function* `wctlproc` 259a⟩≡ (349b)
```
void
wctlproc(void *v)
{
    Channel *c = v;
    char *buf;
    int n, eofs;

    threadsetname("WCTLPROC");

    eofs = 0;
    for(;;){
        buf = emalloc(messagesize);
        // blocking call
        n = read(wctlfd, buf, messagesize-1); /* room for \0 */
        if(n < 0)
            break;
        if(n == 0){
            if(++eofs > 20)
                break;
            continue;
        }
        eofs = 0;

        buf[n] = '\0';
        sendp(c, buf);
    }
}
```
Uses `messagesize` 76a and `wctlfd` 258a.

wctlthread()

`wctlthread()`[259b] is the dispatching half, running in `main-proc` (see Figure 2.11). It receives command strings, parses them with `parsewctl()`[263c], and currently only handles the `New` command—the other wctl commands (resize, move, top, etc.) are handled through `/dev/wctl` instead.

⟨*function* `wctlthread` 259b⟩≡ (349b)
```
void
wctlthread(void *v)
{
    Channel *c = v;
    char *buf, *arg, *dir;
    int cmd, id, pid, hideit, scrollit;
    Rectangle rect;
    char err[ERRMAX];

    threadsetname("WCTLTHREAD");

    for(;;){
        buf = recvp(c);

        cmd = parsewctl(&arg, ZR, &rect, &pid, &id, &hideit, &scrollit, &dir, buf, err);
```

```
        switch(cmd){
        case New:
            wctlnew(rect, arg, pid, hideit, scrollit, dir, err);
        }
        free(buf);
    }
}
```
Uses New-60 260a and wctlnew() 265.

parsewctl()

The wctl command vocabulary covers window lifecycle (`New`, `Delete`), geometry (`Resize`, `Move`), z-order (`Top`, `Bottom`), visibility (`Hide`, `Unhide`), focus (`Current`), and scrolling mode (`Scroll`, `Noscroll`). Commands at or above `Top` are disallowed while a mouse button is pressed, to avoid interfering with drag operations.

⟨*enum* _anon_ (rio/wctl.c) 260a⟩≡ (349b)
```
/* >= Top are disallowed if mouse button is pressed */
enum
{
    New,
    Resize,
    Move,
    Scroll,
    Noscroll,
    Set,
    Top,
    Bottom,
    Current,
    Hide,
    Unhide,
    Delete,
};
```

⟨*global* cmds 260b⟩≡ (349b)
```
static char *cmds[] = {
    [New] = "new",
    [Resize] = "resize",
    [Move] = "move",
    [Scroll] = "scroll",
    [Noscroll] = "noscroll",
    [Set] = "set",
    [Top] = "top",
    [Bottom] = "bottom",
    [Current] = "current",
    [Hide] = "hide",
    [Unhide] = "unhide",
    [Delete] = "delete",
    nil
};
```

⟨*enum* _anon_ (rio/wctl.c)*2* 260c⟩≡ (349b)
```
enum
{
    Cd,
    Deltax,
    Deltay,
    Hidden,
    Id,
```

```
    Maxx,
    Maxy,
    Minx,
    Miny,

    PID,
    R,

    Scrolling,
    Noscrolling,
};
```

⟨*global* params 261a⟩≡ (349b)

```
static char *params[] = {
    [Cd]    = "-cd",
    [Deltax]   = "-dx",
    [Deltay]   = "-dy",
    [Hidden]   = "-hide",
    [Id]    = "-id",
    [Maxx]   = "-maxx",
    [Maxy]   = "-maxy",
    [Minx]   = "-minx",
    [Miny]   = "-miny",
    [PID]   = "-pid",
    [R]    = "-r",
    [Scrolling]   = "-scroll",
    [Noscrolling]  = "-noscroll",
    nil
};
```

Later `parsewctl()`[263c] is a mini command-line parser shared between `/dev/wctl`, `/srv/riowctl`, and the "new" attach spec. It first extracts the command verb, then loops over the dash-prefixed parameters. The `set()` helper supports relative adjustments: a bare number is absolute, `+n` adds, `-n` subtracts. Finally `rectonscreen()` clamps the result to the display bounds.

⟨*function* goodrect 261b⟩≡ (340c)

```
/*
 * Check that newly created window will be of manageable size
 */
int
goodrect(Rectangle r)
{
    if(!eqrect(canonrect(r), r))
        return 0;
    if(Dx(r)<100 || Dy(r)<3*font->height)
        return 0;
    /* must have some screen and border visible so we can move it out of the way */
    if(Dx(r) >= Dx(view->r) && Dy(r) >= Dy(view->r))
        return 0;
    /* reasonable sizes only please */
    if(Dx(r) > BIG*Dx(view->r))
        return 0;
    if(Dy(r) > BIG*Dx(view->r))
        return 0;
    return 1;
}
```

Uses BIG 181h.

⟨*function* word 261c⟩≡ (349b)

```
static
```

```
int
word(char **sp, char *tab[])
{
    char *s, *t;
    int i;

    s = *sp;
    while(isspace(*s))
        s++;
    t = s;
    while(*s!='\0' && !isspace(*s))
        s++;
    for(i=0; tab[i]!=nil; i++)
        if(strncmp(tab[i], t, strlen(tab[i])) == 0){
            *sp = s;
            return i;
    }
    return -1;
}
```

⟨*function* `set` 262a⟩≡ (349b)

```
int
set(int sign, int neg, int abs, int pos)
{
    if(sign < 0)
        return neg;
    if(sign > 0)
        return pos;
    return abs;
}
```

`newrect()` generates default positions for new windows using a simple cascading scheme: each new window is offset by 16 pixels from the previous one, cycling through 10 positions. This avoids stacking windows exactly on top of each other.

⟨*function* `newrect` 262b⟩≡ (349b)

```
Rectangle
newrect(void)
{
    static int i = 0;
    int minx, miny, dx, dy;

    dx = min(600, Dx(view->r) - 2*Borderwidth);
    dy = min(400, Dy(view->r) - 2*Borderwidth);
    minx = 32 + 16*i;
    miny = 32 + 16*i;
    i++;
    i %= 10;

    return Rect(minx, miny, minx+dx, miny+dy);
}
```

Uses `min()` 284a.

⟨*function* `shift` 262c⟩≡ (349b)

```
void
shift(int *minp, int *maxp, int min, int max)
{
    if(*minp < min){
        *maxp += min-*minp;
        *minp = min;
```

```
    }
    if(*maxp > max){
        *minp += max-*maxp;
        *maxp = max;
    }
}
```

⟨*function* `rectonscreen` 263a⟩≡ (349b)

```
Rectangle
rectonscreen(Rectangle r)
{
    shift(&r.min.x, &r.max.x, view->r.min.x, view->r.max.x);
    shift(&r.min.y, &r.max.y, view->r.min.y, view->r.max.y);
    return r;
}
```

Uses `shift()` 262c.

⟨*function* `riostrtol` 263b⟩≡ (349b)

```
/* permit square brackets, in the manner of %R */
int
riostrtol(char *s, char **t)
{
    int n;

    while(*s!='\0' && (*s==' ' || *s=='\t' || *s=='['))
        s++;
    if(*s == '[')
        s++;
    n = strtol(s, t, 10);
    if(*t != s)
        while((*t)[0] == ']')
            (*t)++;
    return n;
}
```

Here, at last, is `parsewctl()`, the little parser we kept deferring to. It turns a `wctl` command line—“`resize -r 0 0 640 480`” and the like—into a `cmd` code plus the parsed rectangle, pid, and flags, and is shared by the `/mnt/wsys/wctl` writer and the external-access paths.

⟨*function* `parsewctl` 263c⟩≡ (349b)

```
int
parsewctl(char **argp, Rectangle r, Rectangle *rp, int *pidp, int *idp, int *hiddenp, int *scrollingp, char **c
{
    int cmd, param, xy, sign;
    char *t;

    *pidp = 0;
    *hiddenp = 0;
    *scrollingp = scrolling;
    *cdp = nil;
    cmd = word(&s, cmds);
    if(cmd < 0){
        strcpy(err, "unrecognized wctl command");
        return -1;
    }
    if(cmd == New)
        r = newrect();

    strcpy(err, "missing or bad wctl parameter");
```

```
while((param = word(&s, params)) >= 0){
    switch(param){ /* special cases */
    case Hidden:
        *hiddenp = 1;
        continue;
    case Scrolling:
        *scrollingp = 1;
        continue;
    case Noscrolling:
        *scrollingp = 0;
        continue;
    case R:
        r.min.x = riostrtol(s, &t);
        if(t == s)
            return -1;
        s = t;
        r.min.y = riostrtol(s, &t);
        if(t == s)
            return -1;
        s = t;
        r.max.x = riostrtol(s, &t);
        if(t == s)
            return -1;
        s = t;
        r.max.y = riostrtol(s, &t);
        if(t == s)
            return -1;
        s = t;
        continue;
    }
    while(isspace(*s))
        s++;
    if(param == Cd){
        *cdp = s;
        while(*s && !isspace(*s))
            s++;
        if(*s != '\0')
            *s++ = '\0';
        continue;
    }
    sign = 0;
    if(*s == '-'){
        sign = -1;
        s++;
    }else if(*s == '+'){
        sign = +1;
        s++;
    }
    if(!isdigit(*s))
        return -1;
    xy = riostrtol(s, &s);

    switch(param){
    case Minx:
        r.min.x = set(sign, r.min.x-xy, xy, r.min.x+xy);
        break;
    case Miny:
        r.min.y = set(sign, r.min.y-xy, xy, r.min.y+xy);
        break;
    case Maxx:
```

```
            r.max.x = set(sign, r.max.x-xy, xy, r.max.x+xy);
            break;
        case Maxy:
            r.max.y = set(sign, r.max.y-xy, xy, r.max.y+xy);
            break;
        case Deltax:
            r.max.x = set(sign, r.max.x-xy, r.min.x+xy, r.max.x+xy);
            break;
        case Deltay:
            r.max.y = set(sign, r.max.y-xy, r.min.y+xy, r.max.y+xy);
            break;
        case Id:
            if(idp != nil)
                *idp = xy;
            break;
        case PID:
            if(pidp != nil)
                *pidp = xy;
            break;
        case -1:
            strcpy(err, "unrecognized wctl parameter");
            return -1;
        }
    }

    *rp = rectonscreen(rectaddpt(r, view->r.min));

    while(isspace(*s))
        s++;
    if(cmd!=New && *s!='\0'){
        strcpy(err, "extraneous text in wctl message");
        return -1;
    }

    if(argp)
        *argp = s;

    return cmd;
}
```

wctlnew()

wctlnew() ties the pieces together: it validates the rectangle, builds an **argv** for **rc**, allocates the window image (either a visible layer on **desktop** or a hidden off-screen image), and calls **new()** to create the window thread and shell process.

⟨*function* wctlnew 265⟩≡ (349b)

```
int
wctlnew(Rectangle rect, char *arg, int pid, int hideit, int scrollit, char *dir, char *err)
{
    char **argv;
    Image *i;

    if(!goodrect(rect)){
        strcpy(err, Ebadwr);
        return -1;
    }
    argv = emalloc(4*sizeof(char*));
    argv[0] = "rc";
```

```
    argv[1] = "-c";
    while(isspace(*arg))
        arg++;
    if(*arg == '\0'){
        argv[1] = "-i";
        argv[2] = nil;
    }else{
        argv[2] = arg;
        argv[3] = nil;
    }
    if(hideit)
        i = allocimage(display, rect, view->chan, false, DWhite);
    else
        i = allocwindow(desktop, rect, Refbackup, DWhite);
    if(i == nil){
        strcpy(err, Ewalloc);
        return -1;
    }
    border(i, rect, Selborder, red, ZP);

    new(i, hideit, scrollit, pid, dir, "/bin/rc", argv);

    free(argv); /* when new() returns, argv and args have been copied */
    return 1;
}
```

Uses `Ebadwr` 280f, `Ewalloc` 280g, `Selborder` 67, `desktop` 48d, `goodrect()` 261b, `new()` 92c, and `red` 48f.

13.7 Advanced fileserver features

This section collects the file-server corners deferred until now: how a client abandons a request it no longer wants (9P `Tflush`), and how `rio` handles 9P authentication.

13.7.1 Flushing

When a client process is interrupted while blocked on a read from `rio` (e.g., reading `/dev/mouse` or `/dev/cons`), the kernel sends a `Tflush` 9P message to cancel the pending request. This is tricky to implement because the `Xfid`[55b] thread handling the original request may be blocked waiting on a channel. The approach uses a combination of reference counting and a `Xfid.flushc`[266a] notification channel. `xfidflush()`[267c] finds the `Xfid` with the matching tag. If the target is not actively running (its `active` lock can be acquired), it sends directly on `flushc`. If the target is running, it waits for it to finish by queueing on the lock. The `flushing` flag tells the target to wake up and abort its operation. Each blocking read/write operation checks `x->flushing` after waking up and, if set, cancels the request.

⟨`Xfid` *flushing fields* 266a⟩≡ (55b)

```
int flushtag; /* our tag, so flush can find us */
// chan<int> (listener = ?, sender = ?)
Channel *flushc;/* channel(int) to notify us we're being flushed */
bool flushing; /* another Xfid is trying to flush us */
```

⟨`xfidallocthread()` *create flushc channel* 266b⟩≡ (78d)

```
x->flushc = chancreate(sizeof(int), 0); /* notification only; nodata */
x->flushtag = -1;
```

⟨`Xfid` *other fields* 266c⟩≡ (55b)

```
QLock active;
```

⟨fcall *other methods* 267a⟩+≡ (56b) ◁139b 270a▷
```
[Tflush]    = filsysflush,
```
Uses filsysflush() 267b.

⟨*function* filsysflush 267b⟩≡ (346c)
```
static
Xfid*
filsysflush(Filsys*, Xfid *x, Fid*)
{
    sendp(x->c, xfidflush);
    return nil;
}
```
Uses xfidflush() 267c.

⟨*function* xfidflush 267c⟩≡ (340a)
```
void
xfidflush(Xfid *x)
{
    Fcall fc;
    Xfid *xf;

    for(xf=xfid; xf; xf=xf->next)
        if(xf->flushtag == x->req.oldtag){
            xf->flushtag = -1;
            xf->flushing = true;
            incref(xf); /* to hold data structures up at tail of synchronization */
            if(xf->ref == 1)
                error("ref 1 in flush");
            if(canqlock(&xf->active)){
                qunlock(&xf->active);
                sendul(xf->flushc, 0);
            }else{
                qlock(&xf->active); /* wait for him to finish */
                qunlock(&xf->active);
            }
            xf->flushing = false;

            if(decref(xf) == 0)
                sendp(cxfidfree, xf);
            break;
        }
    filsysrespond(x->fs, x, &fc, nil);
}
```
Uses cxfidfree-54 62c, error() 282c, filsysrespond() 124, and xfid-52 78a.

⟨*function* filsyscancel 267d⟩≡ (346a)
```
void
filsyscancel(Xfid *x)
{
    if(x->buf){
        free(x->buf);
        x->buf = nil;
    }
}
```

⟨xfidxxx() *set flushtag* 267e⟩≡ (214d 150c 148d 143c)
```
x->flushtag = req->tag;
```

⟨xfidxxx() *unset flushtag* 267f⟩≡ (214d 150c 148d 143c)
```
x->flushtag = -1;
```

⟨xfidread() *when Qmouse, set alts for flush* 268a⟩≡ (143c)

```
alts[MRflush].c = x->flushc;
alts[MRflush].v = nil;
alts[MRflush].op = CHANRCV;
```

Uses MRflush-97 143a.

⟨xfidread() *when Qmouse, switch alt flush case* 268b⟩≡ (143c)

```
case MRflush:
    filsyscancel(x);
    return;
```

Uses MRflush-97 143a and filsyscancel() 267d.

⟨xfidread() *when Qmouse, if flushing* 268c⟩≡ (143c)

```
if(x->flushing){
    recv(x->flushc, nil);  /* wake up flushing xfid */
    recv(mrm.cm, nil);    /* wake up window and toss data */
    filsyscancel(x);
    return;
}
```

Uses filsyscancel() 267d.

⟨xfidread() *when Qcons, set alts for flush* 268d⟩≡ (148d)

```
alts[CRflush].c = x->flushc;
alts[CRflush].v = nil;
alts[CRflush].op = CHANRCV;
```

Uses CRflush-94 148b.

⟨xfidread() *when Qcons, switch alt flush case* 268e⟩≡ (148d)

```
case CRflush:
    filsyscancel(x);
    return;
```

Uses CRflush-94 148b and filsyscancel() 267d.

⟨xfidread() *when Qcons, if flushing* 268f⟩≡ (148d)

```
if(x->flushing){
    recv(x->flushc, nil); /* wake up flushing xfid */
    recv(c2, nil);    /* wake up window and toss data */
    free(t);
    filsyscancel(x);
    return;
}
```

Uses filsyscancel() 267d.

⟨xfidwrite() *when Qcons, set alts for flush* 268g⟩≡ (150c)

```
alts[CWflush].c = x->flushc;
alts[CWflush].v = nil;
alts[CWflush].op = CHANRCV;
```

Uses CWflush-91 149d.

⟨xfidwrite() *when Qcons, switch alt flush case* 268h⟩≡ (150c)

```
case CWflush:
    filsyscancel(x);
    return;
```

Uses CWflush-91 149d and filsyscancel() 267d.

⟨xfidwrite() *when Qcons, if flushing* 269a⟩≡ (150c)

```
  if(x->flushing){
      recv(x->flushc, nil); /* wake up flushing xfid */
      pair.s = runemalloc(1);
      pair.ns = 0;
      send(cwm.cw, &pair);  /* wake up window with empty data */
      filsyscancel(x);
      return;
  }
```

Uses filsyscancel() 267d and runemalloc 284d.

⟨xfidread() *when Qwctl, set alts for flush* 269b⟩≡ (214d)

```
  alts[WCRflush].c = x->flushc;
  alts[WCRflush].v = nil;
  alts[WCRflush].op = CHANRCV;
```

Uses WCRflush-100 214b.

⟨xfidread() *when Qwctl, switch alt flush case* 269c⟩≡ (214d)

```
  case WCRflush:
      filsyscancel(x);
      return;
```

Uses WCRflush-100 214b and filsyscancel() 267d.

⟨xfidread() *when Qwctl, if flushing* 269d⟩≡ (214d)

```
  if(x->flushing){
      recv(x->flushc, nil); /* wake up flushing xfid */
      recv(c2, nil);    /* wake up window and toss data */
      free(t);
      filsyscancel(x);
      return;
  }
```

Uses filsyscancel() 267d.

13.7.2 Authentification

`rio`'s security model is minimal. It reads the username from `/dev/user` at startup and stores it in `Filsys.user`[53a], which the filesystem uses to verify that 9P attach requests come from the same user. Beyond this, `rio` provides no authentication—the `Tauth` handler simply returns "authentication not required". Any process that can mount `rio`'s `/srv` pipe gets full access to all windows, similar to X11's permissive model. Even macOS doesn't really improve on this: its security ultimately relies on app review rather than technical isolation.

⟨filsysinit() *other locals* 269e⟩+≡ (61c) ◁258d

```
  fdt fd;
  char buf[128];
  int n;
```

⟨filsysinit() *set fs user* 269f⟩≡ (61c)

```
  fd = open("/dev/user", OREAD);
  strcpy(buf, "Jean-Paul_Belmondo"); // lol
  if(fd >= 0){
      n = read(fd, buf, sizeof buf-1);
      if(n > 0)
          buf[n] = '\0';
      close(fd);
  }
  fs->user = estrdup(buf);
```

⟨fcall *other methods* 270a⟩+≡ (56b) ◁267a
```
  [Tauth]     = filsysauth,
```
Uses filsysauth() 270b.

⟨*function* filsysauth 270b⟩≡ (346c)
```
  static
  Xfid*
  filsysauth(Filsys *fs, Xfid *x, Fid*)
  {
      Fcall fc;

      return filsysrespond(fs, x, &fc, "rio: authentication not required");
  }
```
Uses filsysrespond() 124.

13.8 Additional control messages

A few Wctlmesg[91b] kinds remain that did not fit the earlier flows. The first is *holding mode*.

13.8.1 Holding mode

Holding mode freezes a window's output: when Window.holding[270c] is true, the WCread alt is disabled so the window thread stops reading from the application's stdout. This lets the user examine output without it scrolling away. Holding is indicated visually through blue-tinted borders and text colors, and the cursor changes to a white arrow. It can be toggled by Escape or by writing "holdon"/"holdoff" to /dev/consctl. The holding counter supports nested holds (multiple programs can request it), and entering raw mode automatically disables holding.

The use case is a pause button. When a program floods the window faster than you can read—a build log, a chatty server—you press Escape to *hold* it: rio stops reading the program's stdout, so the text stops scrolling and, because that stdout pipe soon fills, the program itself blocks. You read at leisure, then unhold to let it run on. It is stronger than scrollback, which only lets you look back while new output keeps piling up.

⟨Window *config fields* 270c⟩+≡ (49) ◁160a
```
  bool holding;
```

⟨winctl() *alts adjustments, if holding* 270d⟩≡ (165g)
```
  if(w->holding)
      alts[WCread].op = CHANNOP;
```
Uses WCread-29 165b.

⟨wsetcursor() *if holding* 270e⟩≡ (87)
```
  if(p==nil && w->holding)
      p = &whitearrow;
```
Uses whitearrow 82c.

⟨wborder() *if holding* 270f⟩≡ (96a)
```
  if(w->holding){
      if(type == Selborder)
          col = holdcol;
      else
          col = paleholdcol;
  }
```
Uses Selborder 67, holdcol-22 271b, and paleholdcol-24 271c.

⟨wsetcols() *if holding* 271a⟩≡ (175c)

```
  if(w->holding)
      if(w == input)
          w->frm.cols[TEXT] = w->frm.cols[HTEXT] = holdcol;
      else
          w->frm.cols[TEXT] = w->frm.cols[HTEXT] = lightholdcol;
```

Uses HTEXT 289f, TEXT 289f, holdcol-22 271b, input 51e, and lightholdcol-23 271d.

⟨*global* holdcol 271b⟩≡ (342b)

```
  static Image *holdcol;
```

⟨*global* paleholdcol 271c⟩≡ (342b)

```
  static Image *paleholdcol;
```

⟨*global* lightholdcol 271d⟩≡ (342b)

```
  static Image *lightholdcol;
```

⟨wmk() *extra colors initialisation* 271e⟩+≡ (175a) ◁176b

```
  holdcol      = allocimage(display, Rect(0,0,1,1), CMAP8, true, DMedblue);
  lightholdcol = allocimage(display, Rect(0,0,1,1), CMAP8, true, DGreyblue);
  paleholdcol  = allocimage(display, Rect(0,0,1,1), CMAP8, true, DPalegreyblue
```

Uses holdcol-22 271b, lightholdcol-23 271d, and paleholdcol-24 271c.

⟨Wctlmesgkind *cases* 271f⟩+≡ (91a) ◁168c 272c▷

```
  Holdon,
  Holdoff,
```

⟨xfidwrite() *Qconsctl case* 271g⟩+≡ (153b) ◁160b

```
  if(strncmp(req->data, "holdon", 6)==0){
      if(w->holding++ == 0)
          wsendctlmesg(w, Holdon, ZR, nil);
      break;
  }
  if(strncmp(req->data, "holdoff", 7)==0 && w->holding){
      if(--w->holding == false)
          wsendctlmesg(w, Holdoff, ZR, nil);
      break;
  }
```

Uses Holdoff 271f, Holdon 271f, and wsendctlmesg() 91c.

⟨xfidclose() *Qconsctl case, if holding* 271h⟩≡ (153a)

```
  if(w->holding){
      w->holding = false;
      wsendctlmesg(w, Holdoff, ZR, nil);
  }
```

Uses Holdoff 271f and wsendctlmesg() 91c.

⟨xfidwrite() *Qconsctl case, if rawon message and holding mode* 271i⟩≡ (160b)

```
  if(w->holding){
      w->holding = false;
      wsendctlmesg(w, Holdoff, ZR, nil);
  }
```

Uses Holdoff 271f and wsendctlmesg() 91c.

⟨wkeyctl() *if holding* 272a⟩≡ (73b)

```
  if(r==0x1B || (w->holding && r==0x7F)){ /* toggle hold */
      if(w->holding)
          --w->holding;
      else
          w->holding++;
      wrepaint(w);
      if(r == 0x1B)
          return;
  }
```

Uses `wrepaint()` 105b.

⟨wctlmesg() *cases* 272b⟩+≡ (92a) ◁168d 272d▷

```
  case Holdon:
  case Holdoff:
      ⟨wctlmesg() break if window was deleted 107b⟩
      wrepaint(w);
      flushimage(display, true);
      break;
```

Uses `Holdoff` 271f, `Holdon` 271f, and `wrepaint()` 105b.

13.8.2 Waking the window thread: `Wakeup`

⟨Wctlmesgkind *cases* 272c⟩+≡ (91a) ◁271f

```
  Wakeup,
```

⟨wctlmesg() *cases* 272d⟩+≡ (92a) ◁272b

```
  case Wakeup:
      break;
```

Uses `Wakeup` 272c.

`Wakeup` is sent by `wcurrent()`, `button2menu()`, and `xfidopen(Qwctl)` to poke a window thread out of its `alt()` sleep. The message handler itself is a no-op—the purpose is just to make the window thread re-evaluate its alt conditions (e.g., to notice that it is now the current window or that a file has been opened).

13.9 Security

The authentication of Section 13.7.2 guards `rio`'s *network* boundary. Step back now to the wider question it raises: what protects one window from another?

`rio` has, frankly, almost no security model *between* the windows of one user—and that is deliberate: Plan 9 assumes the windows on a terminal all belong to the same person, who trusts them. Isolation comes from the *namespace*, not from access checks. Each client is handed a private namespace in which `/dev/cons` and `/dev/mouse` resolve, through the mount spec, to its own window's virtual files; a process cannot read another window's keystrokes for the simple reason that the other window's `/dev/cons` is not in its namespace at all.

The drawing path is the leaky part. `rio` connects each client straight to the kernel's `/dev/draw`, and that device exposes the whole screen image (`display->image`); a process that goes looking can read or scribble on pixels well outside its own window. Plan 9 tolerates this because, once more, it is all one user—there is no untrusted code to defend against.

Other systems span the spectrum. X Window is, if anything, *more* permissive than `rio`: any client on a display can read other windows' pixels, grab the keyboard, and synthesise input events, which is why keyloggers and screenshot tools need no special privilege under bare X. Wayland was designed largely to close exactly these holes—clients are isolated, cannot see each other's buffers or input, and must go through the compositor (and desktop "portals") even to take a screenshot. macOS and Windows sit nearer Wayland, gating screen capture

and global input behind explicit, user-granted permissions. Plan 9 never took this road; its answer to "what stops one window from spying on another?" is simply "they are both you."

The real boundary in Plan 9 is drawn elsewhere: at the *file server* and the *network*. A remote `rio` or file tree is reached over an authenticated 9P connection (see the Security book [**?**]), and a process is confined by the namespace it is given. Within a single trusted session, though, the windows are deliberately porous.

Chapter 14

Conclusion

You now know how the Plan 9 windowing system `rio` works—from the hardware interrupt when you click the mouse, through the chain of IO procs, threads, and channels, all the way to the 9P response that unblocks the window application. More generally, you now understand how a windowing system manages multiple windows, dispatches input, and provides an environment for applications.

`rio` is simultaneously three things: a graphical application (it calls `initdraw()` and draws on the screen), a window manager (it creates, moves, resizes, and deletes windows), and a file server (it serves `/mnt/wsys/` files through 9P). To each window `rio` gives its own virtual `/dev/cons`, `/dev/mouse`, and `/dev/winname`; a program running inside a window therefore sees exactly the interface it would on a bare console, and need not even know it is in a window. This multiplexer design has remarkable consequences. Because a window is indistinguishable from a bare console, `rio` can run inside one of its own windows—recursive `rio`; any program that reads and writes `/dev/` files works in a windowed environment unchanged; and `rio` gains network transparency for free through 9P.

14.1 Patterns and techniques

Many of the techniques below are concurrency design patterns, and unusually powerful ones—you might recognize several from Go. That is no accident: Rob Pike had a hand in `libthread`, in `rio`, and later in Go, and Plan 9's concurrent programs like `rio` were in effect a proving ground for the channel-based style Go would popularize. These techniques apply far beyond windowing systems:

- *Active objects*: one thread per window, communicating by message passing. In `rio` this is the `winctl()`[71b] thread—each window's state (its image, its input buffer, its mode) is owned by that one thread and touched only through the messages other threads send it, so there are no locks and no races. This is how Erlang processes and Go services are structured: eliminate locks by giving each object exclusive ownership of its state. The pattern is the actor model, traceable to Carl Hewitt in the 1970s and made mainstream by Erlang; Plan 9 arrives at it not through a language feature but through plain `libthread` threads selecting over channels with `alt()`.

- *Select over channels*: every long-lived thread in `rio`—`winctl()`, `mousethread()`[66a], and the `xfidread()`[136b] workers—sits in an `alt()` that waits on several channels at once and acts on whichever fires, instead of polling or dedicating a thread per source. This is exactly Go's `select`, and the role `select`/`poll`/`epoll` play in event-driven servers: one thread cleanly multiplexing many event streams.

- *Master/worker dispatch*: the `fileserver` proc reads each 9P request and hands the ones that might block to a pool of reusable `Xfid` worker threads—recycled rather than created per request—so its read loop never stalls waiting on a single slow client. The same architecture is used by web servers (like Nginx) and database connection pools: decouple request arrival rate from processing capacity.

- *Delegated thread creation*: a worker must run in `main-proc` to share its globals, but the request to make one arrives in the `fileserver` proc. `rio` bridges this with a worker allocator: the `fileserver` sends on `cxfidalloc`, and `xfidallocthread()`[78d] (a thread of `main-proc`) does the actual `threadcreate` and hands the new `Xfid` back over the same channel. Spawning is delegated across procs through a channel—a tidy answer to "create this in another address space for me."

- *Tag multiplexing*: a single pipe serves many clients at once because each request carries a unique *tag* and the kernel writes whole messages atomically, so the server may reply in any order and the kernel still routes each reply back to the right client. The same trick lets HTTP/2 interleave many streams over one TCP connection.

- *Channel-of-channels*: a worker sends its own reply channel along with a request; the responder uses that channel to deliver data. This is how futures and promises work, and how RPC systems deliver replies. It decouples requester from provider without shared state. The roots are in Tony Hoare's Communicating Sequential Processes, but first-class channels—the ability to send channels over channels—came into their own in Newsqueak, designed by Rob Pike, and passed from there through Alef and Limbo into Go. Pike singled this out as what makes channels far more than message queues: once a channel is itself a value you can communicate, the communication graph is no longer fixed, and processes can build entirely new communication structures at runtime.

- *Synchronous rendezvous*: `rio`'s channels are unbuffered (`chancreate(..., 0)`), so a `send` blocks until a matching `recv`; the communication is the synchronization. The mouse handshake relies on this—the worker's first message is not data but an "I am ready" signal, and only a rendezvous, never a queue, can guarantee the value handed back is the freshest one. These are the original CSP semantics, the same as an unbuffered Go channel.

- *Circular buffer*: the `Mouseinfo` queue buffers events with wrap-around indexes. This bounded ring buffer backs Unix pipes, kernel log buffers, audio drivers, and lock-free queues—constant-time enqueue and dequeue with bounded memory.

- *Async IO offloading*: the blocking `read`s on the real `/dev/cons` and `/dev/mouse` live in dedicated `IO-procs` that forward each event through a channel, so the cooperatively-scheduled `main-proc` never blocks on a syscall and the GUI stays live. The same idea drives Node.js (libuv's thread pool) and Go's runtime scheduler: keep the event loop responsive by moving blocking work elsewhere.

- *Fork/adjust/exec*: create a process, customize its namespace, then exec. This three-phase pattern is how Docker containers start (create namespace, set up mounts, exec the entrypoint) and how sandboxes are built (fork, drop privileges, exec).

These patterns speak to a long-running debate: threads versus events. The event camp (John Ousterhout's 1996 "Why Threads Are a Bad Idea") builds a server as a single thread spinning a `select`/`poll` loop over callbacks and explicit state machines—fast and race-free, but with the control flow shredded across callbacks. The thread camp (von Behren et al., "Why Events Are a Bad Idea", 2003) keeps code straight-line and blocking, at the cost of stacks and, with preemptive threads, locks and races.

`rio` refuses to pick a side, and that is the point. Its workers are written as threads—`xfidread()` just `recv`s and blocks, in plain sequential code—yet the `alt()` at the heart of every long-lived thread is precisely the event loop's `select`. Plan 9's `libthread` threads are cooperative, so there is no preemption and none of the lock-and-race tax the thread camp pays; the only truly blocking syscalls are exiled to separate IO procs (the offloading pattern above) so they never freeze the scheduler. The code reads like threads but behaves like an event loop. This CSP-flavoured synthesis—cheap cooperative threads plus channels and a `select`—is the same bet Go later made mainstream with goroutines.

14.2 Connections to other books

- GRAPHICS book [Pad16c]: `rio` is built entirely on top of `libdraw` and its layering extensions. Every window is a `Screen` layer, and all drawing uses the functions described in the GRAPHICS book [Pad16c].
- KERNEL book [Pad14]: `rio` depends on the kernel's `devcons`, `devmouse`, `devdraw`, and the `/srv` mechanism for publishing its file server. The kernel's process and namespace model makes it possible for each window to have its own view of `/dev`.
- WIDGETS book [Pad26]: the `libpanel` widget toolkit is most often used by programs running inside `rio` windows, through the virtual devices `rio` provides. It is not tied to `rio`, however—it draws on any `libdraw` display, so a program like `mothra` (the web browser) can use it outside `rio` just as well. The WIDGETS book [Pad26] covers these higher-level UI components (buttons, scrollbars, menus).
- SHELL book [Pad18]: `rc` is the default program that runs inside each `rio` window. The interaction between `rc`'s line reading and `rio`'s textual window (which provides editing and scrollback) is what gives the Plan 9 terminal its character.
- LIBCORE book [Pad16a]: `rio` uses `libthread` for its multi-threaded architecture, with procs, threads, and channels as the concurrency primitives.

14.3 Beyond the Plan 9 windowing system

`rio` is deliberately minimal: it provides windows, a menu for creating and arranging them, and terminal emulation. Modern windowing systems and desktop environments do considerably more:

- *Compositing*: modern compositors (macOS Quartz, Wayland compositors, Windows DWM) render each window into an off-screen buffer and composite them on the GPU, enabling transparency, shadows, and smooth animations. `rio` does only the barest version of this: it "composites" in software, stacking overlapping window layers through the draw device's layer mechanism—no off-screen buffers, no GPU.
- *Window decoration and theming*: most window managers draw title bars, close/minimize buttons, and support customizable themes. `rio` has no window decorations at all—windows are plain rectangles with a colored border indicating focus.
- *Desktop metaphors*: modern desktops provide taskbars, application launchers, virtual desktops, notification systems, drag-and-drop, and clipboard managers. `rio`'s interface is a right-click menu and a mouse-based window creation gesture—everything else is left to the programs running inside windows.
- *Accessibility*: modern windowing systems provide accessibility APIs (screen readers, magnification, keyboard navigation) as a core feature. `rio` builds in none of this—though here too its open device interface lets such tools live outside it: `lens` (Section E.1), a screen magnifier, is just an external program reading the shared screen image.
- *Multi-monitor and high-DPI*: modern systems handle multiple displays with different resolutions and scale factors. `rio` assumes a single screen at a fixed resolution.
- *Security and isolation*: modern systems keep windows from spying on one another—a process cannot read another window's pixels, snoop its keystrokes, or capture the screen without explicit permission (Wayland, macOS, Windows). `rio`, like bare `X11`, does not: it trusts every window of the one user and isolates only their *namespaces*, not their pixels (Section 13.9).

`rio`'s design reflects a fundamental insight: a windowing system is really a multiplexer for devices. By implementing this multiplexer as a file server, Plan 9 gets composability (nested `rio`), network transparency (run a program on a remote machine and have it draw into your local windows), and simplicity (programs are unaware of windowing) from a single mechanism. Modern systems achieve richer visual results, but from architectures that are far more complex—and run to orders of magnitude more code.

Appendix A

Debugging

Debugging `rio` has a particular circular problem: `rio` is the program that owns the screen and `/dev/cons`, so any `fprint(STDERR, ...)` it tries to emit would normally land in its own display. First, you can launch `rio` from a parent shell and redirect standard error outside the graphical session: `rio >[2] /tmp/errors` sends all diagnostics to a file you can tail from another window or read from the serial console. Second, since `rio` can run recursively (Section 13.2), you can debug a freshly-built `rio` from inside a known-working `rio`, keeping a rescue copy of the old binary at a path like `/save/rio_old` in case the new one refuses to start.

The only built-in debugging facility is the `DEBUG` compile-time flag. When enabled, `filsysproc()`[75] dumps each incoming 9P message on `STDERR` and `filsysrespond()`[124] dumps each reply, using `fmtinstall('F', fcallfmt)` to install a `%F` format specifier for `Fcall` structures. Combined with the stderr redirect above, this gives a readable trace of the `mount`/`attach`/`walk`/`open`/`read`/`write` sequence for any client running inside `rio`.

⟨*constant* DEBUG 278a⟩≡ (333b)
```
#define DEBUG false
```

⟨filsysinit() *install dumper* 278b⟩≡ (61c)
```
fmtinstall('F', fcallfmt);
```

`fcallfmt` is the `print` verb that renders an `Fcall` in readable form; it belongs to the 9P library (declared in `fcall.h`), not to `rio`, and is covered in Libcore book [Pad16a].

⟨filsysproc() *dump Fcall if debug* 278c⟩≡ (75)
```
if(DEBUG)
    fprint(STDERR, "rio:<-%F\n", &x->req);
```
Uses DEBUG 278a.

⟨filsysrespond() *dump Fcall t if debug* 278d⟩≡ (124)
```
if(DEBUG)
    fprint(STDERR, "rio:->%F\n", fc);
```
Uses DEBUG 278a.

The same `DEBUG` flag also gates a scattering of lighter trace prints throughout `rio`—logging each control message and window-thread event—handy when following the flow of messages between threads.

⟨winctl() *trace* w->id *and* event 278e⟩≡ (71b)
```
if(DEBUG) fprint(STDERR, "winctl: win=%d, event=%d\n", w->id, event);
```
Uses DEBUG 278a.

⟨wsendctlmesg() *trace* w->id *and* type 278f⟩≡ (91c)
```
if(DEBUG) fprint(STDERR, "wsendctlmesg: win=%d, type=%d\n", w->id, type);
```
Uses DEBUG 278a.

⟨wctlmesg() *trace* w->id *and* m 278g⟩≡ (92a)
```
if(DEBUG) fprint(STDERR, "wctlmesg: win=%d, type=%d\n", w->id, m);
```
Uses DEBUG 278a.

⟨new() *trace* w->id 279a⟩≡ (92c)
```
if(DEBUG) fprint(STDERR, "new: creating winctl thread for win=%d\n", w->id);
```
Uses DEBUG 278a.

⟨new() *trace before* winshell() 279b⟩≡ (97c)
```
if(DEBUG) fprint(STDERR, "new: creating new winshell\n");
```
Uses DEBUG 278a.

⟨new() *trace after* winshell() 279c⟩≡ (97c)
```
if(DEBUG) fprint(STDERR, "new: created winshell=%d\n", pid);
```
Uses DEBUG 278a.

⟨winshell() *trace* cmd 279d⟩≡ (97d)
```
if(DEBUG) fprint(STDERR, "winshell: cmd = %s\n", cmd);
```
Uses DEBUG 278a.

⟨winshell() *trace before* procexec() 279e⟩≡ (97d)
```
if(DEBUG) fprint(STDERR, "winshell: before procexec %s\n", cmd);
```
Uses DEBUG 278a.

⟨wsetpid() *trace* w->id *and* w->pid 279f⟩≡ (98c)
```
if(DEBUG) fprint(STDERR, "wsetpid: win=%d pid =%d\n", w->id, w->pid);
```
Uses DEBUG 278a.

⟨filsysmount() *trace start* 279g⟩≡ (99c)
```
if(DEBUG) fprint(STDERR, "filsysmount1\n");
```
Uses DEBUG 278a.

⟨filsysmount() *trace end* 279h⟩≡ (99c)
```
if(DEBUG) fprint(STDERR, "filsysmount2\n");
```
Uses DEBUG 278a.

Appendix B

Error Management

This short chapter covers how `rio` names its errors and reports them back to clients. Because `rio` is a 9P file server, an error is not just a local condition—it has to travel back to the client as a message, so the error strings themselves are part of the interface.

B.1 Error codes

`rio` reports errors as strings rather than numeric codes because that is what 9P carries: when `filsysrespond()`[124] is called with an error, the string is copied into an `Rerror` 9P message and sent back through the pipe to the client's kernel, which in turn reports it via the client's `errstr()` or `%r` format (see the KERNEL book [Pad14] and LIBCORE book [Pad16a]). The `Exxx` error strings below are kept as globals simply because they are shared constants reused by many handlers—the standard Plan 9 idiom for error strings. The wording matches the conventions of other Plan 9 fileservers (lowercase and terse, e.g., "permission denied", "file does not exist").

⟨*global* `Eperm` 280a⟩≡ (350a)
```
char Eperm[] = "permission denied";
```
Uses `Eperm` 280a.

⟨*global* `Eexist` 280b⟩≡ (346c)
```
char Eexist[] = "file does not exist";
```
Uses `Eexist` 280b.

⟨*global* `Enotdir` 280c⟩≡ (346c)
```
char Enotdir[] = "not a directory";
```
Uses `Enotdir` 280c.

⟨*global* `Ebadfcall` 280d⟩≡ (346c)
```
char Ebadfcall[] = "bad fcall type";
```
Uses `Ebadfcall` 280d.

⟨*global* `Eoffset` 280e⟩≡ (346c)
```
char Eoffset[] = "illegal offset";
```
Uses `Eoffset` 280e.

⟨*global* `Ebadwr` 280f⟩≡ (349b)
```
char Ebadwr[]  = "bad rectangle in wctl request";
```
Uses `Ebadwr` 280f.

⟨*global* `Ewalloc` 280g⟩≡ (349b)
```
char Ewalloc[]  = "window allocation failed in wctl request";
```
Uses `Ewalloc` 280g.

⟨*global* `Einuse` 281a⟩≡ (347)

```
char Einuse[] =  "file in use";
```

Uses `Einuse` 281a.

⟨*global* `Edeleted` 281b⟩≡ (347)

```
char Edeleted[] = "window deleted";
```

Uses `Edeleted` 281b.

⟨*global* `Ebadreq` 281c⟩≡ (347)

```
char Ebadreq[] = "bad graphics request";
```

Uses `Ebadreq` 281c.

⟨*global* `Etooshort` 281d⟩≡ (347)

```
char Etooshort[] = "buffer too small";
```

Uses `Etooshort` 281d.

⟨*global* `Ebadtile` 281e⟩≡ (347)

```
char Ebadtile[] = "unknown tile";
```

Uses `Ebadtile` 281e.

⟨*global* `Eshort` 281f⟩≡ (347)

```
char Eshort[] =  "short i/o request";
```

Uses `Eshort` 281f.

⟨*global* `Elong` 281g⟩≡ (347)

```
char Elong[] =   "snarf buffer too long";
```

Uses `Elong` 281g.

⟨*global* `Eunkid` 281h⟩≡ (347)

```
char Eunkid[] =  "unknown id in attach";
```

Uses `Eunkid` 281h.

⟨*global* `Ebadrect` 281i⟩≡ (347)

```
char Ebadrect[] =  "bad rectangle in attach";
```

Uses `Ebadrect` 281i.

⟨*global* `Ewindow` 281j⟩≡ (347)

```
char Ewindow[] =  "cannot make window";
```

Uses `Ewindow` 281j.

⟨*global* `Enowindow` 281k⟩≡ (347)

```
char Enowindow[] =  "window has no image";
```

Uses `Enowindow` 281k.

⟨*global* `Ebadmouse` 281l⟩≡ (347)

```
char Ebadmouse[] =  "bad format on /dev/mouse";
```

Uses `Ebadmouse` 281l.

⟨*global* `Ebadwrect` 281m⟩≡ (347)

```
char Ebadwrect[] =  "rectangle outside screen";
```

Uses `Ebadwrect` 281m.

⟨*global* `Ebadoffset` 281n⟩≡ (347)

```
char Ebadoffset[] =  "window read not on scan line boundary";
```

Uses `Ebadoffset` 281n.

B.2 error()

error()[282c] has a two-phase behavior controlled by errorshouldabort[282a]. Before graphics is fully initialized, errorshouldabort is false and an error just calls threadexitsall() to tear down gracefully—a failure here means rio never reached a usable state, so a clean exit is fine. Once everything is set up, errorshouldabort flips to true and error() calls abort() instead of exiting. The reason for abort() rather than exits() is that abort() leaves a core file (via SIGABRT-like behavior, see the LIBCORE book [Pad16a]) and puts the process in Plan 9's "broken" state so it can be inspected later with acid (see the DEBUGGER book [Pad16b]): you lose the running rio, but you gain a post-mortem.

⟨*global* errorshouldabort 282a⟩≡ (350a)
```
bool errorshouldabort = false;
```
Uses errorshouldabort 282a.

⟨main() *error management after everything setup* 282b⟩+≡ (58) ◁219b
```
errorshouldabort = true;/* suicide if there's trouble after this */
⟨main() if initstr or kdbin 226c⟩
```

⟨*function* error 282c⟩≡ (350a)
```
void
error(char *s)
{
    fprint(STDERR, "rio: %s: %r\n", s);
    if(errorshouldabort)
        abort();
    // else
    threadexitsall("error");
}
```
Uses errorshouldabort 282a.

derror()[282d] exists only to adapt error() to the signature libdraw expects from a user-supplied error callback. It is registered via geninitdraw() so that when the draw library encounters a protocol error on /dev/draw, the failure is routed through rio's own abort-with-core path instead of libdraw's default.

⟨*function* derror 282d⟩≡ (350a)
```
void
derror(Display*, char *errorstr)
{
    error(errorstr);
}
```
Uses error() 282c.

Appendix C

Utilities

The utility functions in this appendix fall into three groups. The first is abort-on-failure wrappers around `malloc()`, `realloc()`, and `strdup()`: `emalloc()`[283b], `erealloc()`[283a], and `estrdup()`[283c] call `error()`[282c] if allocation fails, so the rest of `rio` can assume a non-`nil` return and avoid cluttering every call site with sanity checks. The pattern is common in Plan 9 programs and matches the e-prefix convention in `libc` (see the LIBCORE book [Pad16a]).

The second group is rune-aware variants of standard string routines (`runemalloc()`[284d], `runemove()`[285a], `strrune()`[285c], `cvttorunes()`[285b]): `rio` stores text as `Rune` arrays internally but has to convert to and from UTF-8 at I/O boundaries, so these wrappers centralize the conversion and rune-sized arithmetic.

The third is `min`[284a]/`max`[284b]/`isalnum`[284c] — small helpers that exist because the standard headers either do not provide them (no generic `min` in C) or provide ASCII-only versions that fail on non-ASCII runes.

C.1 Memory management

⟨*function* `erealloc` 283a⟩≡ (350b)

```
void*
erealloc(void *p, uint n)
{
    p = realloc(p, n);
    if(p == nil)
        error("realloc failed");
    return p;
}
```

Uses `error()` 282c.

⟨*function* `emalloc` 283b⟩≡ (350b)

```
void*
emalloc(uint n)
{
    void *p;

    p = malloc(n);
    if(p == nil)
        error("malloc failed");
    memset(p, 0, n);
    return p;
}
```

Uses `error()` 282c.

⟨*function* `estrdup` 283c⟩≡ (350b)

```
char*
estrdup(char *s)
```

```
{
    char *p;

    p = malloc(strlen(s)+1);
    if(p == nil)
        error("strdup failed");
    strcpy(p, s);
    return p;
}
```

Uses `error()` 282c.

C.2 Arithmetic

⟨*function* `min` 284a⟩≡ (350b)

```
int
min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
```

⟨*function max (rio/util.c)* 284b⟩≡ (350b)

```
int
max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
```

⟨*function* `isalnum` 284c⟩≡ (350b)

```
//@Scheck: not dead, but conflict with the one in ctype.h
int isalnum(Rune c)
{
    /*
     * Hard to get absolutely right.  Use what we know about ASCII
     * and assume anything above the Latin control characters is
     * potentially an alphanumeric.
     */
    if(c <= ' ')
        return false;
    if(0x7F<=c && c<=0xA0)
        return false;
    if(utfrune("!\"#$%&'()*+,-./:;<=>?@[\\]^`{|}~", c))
        return false;
    return true;
}
```

C.3 Unicode

⟨*function* `runemalloc` 284d⟩≡ (335)

```
#define runemalloc(n)  malloc((n)*sizeof(Rune))
```

⟨*function* `runerealloc` 284e⟩≡ (335)

```
#define runerealloc(a, n) realloc(a, (n)*sizeof(Rune))
```

⟨*function* runemove 285a⟩≡ (335)

```
#define runemove(a, b, n) memmove(a, b, (n)*sizeof(Rune))
```

⟨*function* cvttorunes 285b⟩≡ (350b)

```
void
cvttorunes(char *p, int n, Rune *r, int *nb, int *nr, int *nulls)
{
    uchar *q;
    Rune *s;
    int j, w;

    /*
     * Always guaranteed that n bytes may be interpreted
     * without worrying about partial runes.  This may mean
     * reading up to UTFmax-1 more bytes than n; the caller
     * knows this.  If n is a firm limit, the caller should
     * set p[n] = 0.
     */
    q = (uchar*)p;
    s = r;
    for(j=0; j<n; j+=w){
        if(*q < Runeself){
            w = 1;
            *s = *q++;
        }else{
            w = chartorune(s, (char*)q);
            q += w;
        }
        if(*s)
            s++;
        else if(nulls)
                *nulls = true;
    }
    *nb = (char*)q-p;
    *nr = s-r;
}
```

⟨*function* strrune 285c⟩≡ (350b)

```
Rune*
strrune(Rune *s, Rune c)
{
    Rune c1;

    if(c == 0) {
        while(*s++)
            ;
        return s-1;
    }

    while(c1 = *s++)
        if(c1 == c)
            return s-1;
    return nil;
}
```

⟨*function* runetobyte 285d⟩≡ (350b)

```
char*
runetobyte(Rune *r, int n, int *ip)
{
    char *s;
```

```
	int m;

	s = emalloc(n*UTFmax+1);
	m = snprint(s, n*UTFmax+1, "%.*S", n, r);
	*ip = m;
	return s;
}
```

Appendix D

The Frame Library

The Frame library (used by both `rio` and the editor `sam`) is a text rendering engine that handles line wrapping, tab expansion, cursor display, text selection highlighting, and efficient incremental updates when characters are inserted or deleted. Despite the name "frame" (which might suggest a window border), it is really a text widget.

The fundamental design decision is to avoid redrawing the entire text whenever a character is inserted or deleted. Instead, the library works incrementally: the screen content is represented as a sequence of "boxes" (`Frbox`[361b]), where each box is a run of characters that share the same pixel width. When text is inserted, only the boxes after the insertion point need to shift, and only the gap between old and new positions needs repainting. This makes single-character insertion (the common case when typing) very fast, even with thousands of visible characters. The box abstraction also solves the variable-width Unicode problem: measuring a character's pixel width requires a font lookup, which is expensive. By grouping characters into runs and caching the total width in `Frbox.wid`, the library only measures widths when boxes are created or split, not on every rendering pass. Here is a concrete example of what the box array looks like for a terminal showing two lines:

```
Display:   % ls -la /tmp
           drwxrwxrwt  12 root root 4096 ...

Box array:
[0] nrune=11 ptr="% ls -la /t"  wid=88
[1] nrune=2  ptr="mp"           wid=16   (split at line wrap)
[2] nrune=-1 bc='\n'            wid=5000 (newline break box)
[3] nrune=14 ptr="drwxrwxrwt  1" wid=112
...
```

Text boxes are split at line boundaries (when a box would overflow the right margin, `_frcanfit()`[297a] determines how many characters fit and `_frsplitbox`[308b] divides it). Tabs and newlines get their own special boxes with `nrune == -1`.

D.1 The `Frame` widget

D.1.1 `Frame`

The `Frame` struct is the state of a text widget instance. It does *not* own the text data (which lives in `Window.r`)—it only knows how to display a slice of it, from `org` to `org + nchars`. The key fields fall into five groups:

- **Drawing surface:** `b` (the image to draw on), `r` (the text rectangle, clipped to whole lines), `entire` (the full rectangle including partial lines).

- **Text metrics:** `nchars` (how many runes are visible), `nlines` (how many lines have text), `maxlines` (how many lines fit).
- **Selection:** `p0`, `p1` (frame-local selection range, in rune indices relative to `org`).
- **Cursor:** `tick`, `tickback`, `ticked` (the text cursor image and its save buffer).
- **Box array:** `box`, `nbox`, `nalloc` (the cached layout of visible text).

⟨*struct* `Frame` 288a⟩≡ (361b)
```
struct Frame
{
    Image *b;      /* on which frame appears */
    Rectangle r;  /* in which text appears */

    Font *font;  /* of chars in the frame */
    Display *display; /* on which frame appears */

    ⟨Frame colors 289e⟩
    ⟨Frame text fields 184a⟩
    ⟨Frame tick fields 290a⟩
    ⟨Frame box fields 291c⟩
    ⟨Frame scroll 201a⟩

    ⟨Frame other fields 289c⟩
};
```

⟨`Frame` *text fields* 288b⟩+≡ (288a) ◁194d 288c▷
```
ushort nchars;  /* # runes in frame */
```

⟨`Frame` *text fields* 288c⟩+≡ (288a) ◁288b 289b▷
```
ushort nlines;  /* # lines with text */
```

D.1.2 `frinit()`

Initialization sets the drawing surface, computes how many lines of text fit in the rectangle (clipping the bottom to a whole number of font-height rows via `frsetrects`[289a]), zeros the text counters, sets up the color palette, and creates the tick cursor image. A subtle ordering dependency: the font must be set before `frsetrects` because `frsetrects` uses `f->font->height` to compute `maxlines`. The author notes that reordering these lines caused a segfault during development—an easy mistake because the dependency is not obvious from the function signatures.

⟨*function* `frinit` 288d⟩≡ (364c)
```
void
frinit(Frame *f, Rectangle r, Font *ft, Image *b, Image *cols[NCOL])
{
    f->font = ft;
    f->display = b->display;
    frsetrects(f, r, b);

    f->nchars = 0;
    f->nlines = 0;
    ⟨frinit() initialize other fields 184b⟩
}
```
Uses `NCOL` 289f and `frsetrects()` 289a.

⟨*function* frsetrects 289a⟩≡ (364c)
```
void
frsetrects(Frame *f, Rectangle r, Image *b)
{
    f->b = b;
    f->entire = r;
    f->r = r;
    f->r.max.y -= (r.max.y-r.min.y) % f->font->height;
    f->maxlines = (r.max.y-r.min.y) / f->font->height;
}
```

⟨Frame *text fields* 289b⟩+≡ (288a) ◁288c 299c▷
```
ushort maxlines; /* total # lines in frame */
```

⟨Frame *other fields* 289c⟩≡ (288a)
```
Rectangle entire;  /* of full frame */
```

⟨*function* frclear 289d⟩≡ (364c)
```
void
frclear(Frame *f, bool freeall)
{
    ⟨frclear() free boxes 293b⟩
    ⟨frclear() free ticks 291b⟩
}
```

D.1.3 Frame colors

⟨Frame *colors* 289e⟩≡ (288a)
```
Image *cols[NCOL]; /* text and background colors */
```

⟨*enum* _anon_ (include/frame.h) 289f⟩≡ (361b)
```
enum FrameColors {
    BACK, // Background
    HIGH, // Background highlighted text
    BORD, // Border
    TEXT, // Text
    HTEXT, // Highlited text

    NCOL
};
```

⟨*global* cols 289g⟩≡ (342b)
```
// map<Property, Color>
static Image *cols[NCOL];
```
Uses NCOL 289f.

⟨frinit() *initialize other fields* 289h⟩+≡ (288d) ◁194e 290c▷
```
if(cols != nil)
    memmove(f->cols, cols, sizeof f->cols);
```

D.1.4 Frame tick

The “tick” is the text cursor—a narrow 3-pixel-wide image drawn at the insertion point (`q0`). Rather than repainting the text underneath each time the tick moves, `rio` saves the pixels under the tick into `tickback` and restores them when the tick is removed. This save/restore approach is the same technique used for the mouse cursor in the kernel’s graphics layer. The `ticked` flag tracks whether the tick is currently on screen. When the user has an active selection range (`p0 != p1`), the tick is hidden—there is no blinking cursor inside a highlighted

selection. The tick reappears when the selection collapses to a point (a single click). Note that `rio`'s tick does not blink, unlike most modern terminals.

⟨`Frame` *tick fields* 290a⟩≡ (288a) 290b ▷
```
Image *tick; /* typing tick */
Image *tickback; /* saved image under tick */
```

⟨`Frame` *tick fields* 290b⟩+≡ (288a) ◁ 290a
```
bool ticked; /* flag: is tick onscreen? */
```

⟨`frinit()` *initialize other fields* 290c⟩+≡ (288d) ◁ 289h 291d ▷
```
if(f->tick == nil && f->cols[BACK] != nil)
    frinittick(f);
```
Uses `BACK` 289f and `frinittick()` 290e.

⟨*constant* `FRTICKW` 290d⟩≡ (361b)
```
#define FRTICKW 3
```

The tick image is built procedurally: a `FRTICKW`×height image filled with the background color, a 1-pixel vertical line in the text color down the center, and small squares at the top and bottom to make the ends visible. This creates the classic I-beam cursor shape. The `tickback` image has the same dimensions and stores whatever pixels were under the tick before it was drawn.

⟨*function* `frinittick` 290e⟩≡ (364c)
```
void
frinittick(Frame *f)
{
    Image *b = f->display->screenimage;
    Font *ft = f->font;

    ⟨frinittick() free old tick 290f⟩
    f->tick = allocimage(f->display, Rect(0, 0, FRTICKW, ft->height), b->chan, 0, DWhite);
    ⟨frinittick() sanity check tick 290g⟩
    ⟨frinittick() free old tickback 290h⟩
    f->tickback = allocimage(f->display, f->tick->r, b->chan, false, DWhite);
    ⟨frinittick() sanity check tickback 291a⟩
    /* background color */
    draw(f->tick, f->tick->r, f->cols[BACK], nil, ZP);
    /* vertical line */
    draw(f->tick, Rect(FRTICKW/2, 0, FRTICKW/2+1, ft->height), f->cols[TEXT], nil, ZP);
    /* box on each end */
    draw(f->tick, Rect(0, 0, FRTICKW, FRTICKW), f->cols[TEXT], nil, ZP);
    draw(f->tick, Rect(0, ft->height-FRTICKW, FRTICKW, ft->height), f->cols[TEXT], nil, ZP);
}
```
Uses `BACK` 289f, `FRTICKW` 290d, and `TEXT` 289f.

⟨`frinittick()` *free old tick* 290f⟩≡ (290e)
```
if(f->tick)
    freeimage(f->tick);
```

⟨`frinittick()` *sanity check tick* 290g⟩≡ (290e)
```
if(f->tick == nil)
    return;
```

⟨`frinittick()` *free old tickback* 290h⟩≡ (290e)
```
if(f->tickback)
    freeimage(f->tickback);
```

⟨`frinittick()` *sanity check tickback* 291a⟩≡ (290e)
```
if(f->tickback == nil){
    freeimage(f->tick);
    f->tick = nil;
    return;
}
```

⟨`frclear()` *free ticks* 291b⟩≡ (289d)
```
if(freeall){
    freeimage(f->tick);
    freeimage(f->tickback);
    f->tick = nil;
    f->tickback = nil;
}
f->ticked = false;
```

D.2 Boxes, strings, and coordinates

D.2.1 Frame boxes

The frame's content is stored as an array of boxes (`Frbox`). A box is either a run of regular characters (with a `ptr` to the UTF-8 bytes and a `wid` giving the total pixel width) or a special break character like a newline or tab (indicated by `nrune < 0`). The box abstraction serves two purposes:

1. **Cached widths:** Measuring character widths requires calling into the font subsystem, which involves looking up glyph metrics. By grouping characters into runs and storing the total width in `Frbox.wid`, the frame avoids remeasuring characters on every redraw or position lookup.

2. **Efficient updates:** When a character is inserted, only the boxes at the insertion point need to be split and adjusted. The rest of the array—potentially hundreds of boxes for a full screen of text—remains untouched. Similarly, deletion only affects the boxes in the deleted range.

The box array is a growing array (like a `vector` in C++) with `nbox` used entries and `nalloc` total capacity. When boxes are added, the array grows by `SLOP` (25) entries at a time.

⟨`Frame` *box fields* 291c⟩≡ (288a)
```
// growing_array<Frbox> (size = nalloc, unused after nbox)
Frbox *box;
ushort nbox;
ushort nalloc;
```

⟨`frinit()` *initialize other fields* 291d⟩+≡ (288d) ◁290c 299d▷
```
f->box = nil;
f->nbox = 0;
f->nalloc = 0;
```

`Frbox`

A box is either a run of printable characters (when `nrune >= 0`, with `ptr` pointing to the UTF-8 bytes and `wid` caching the total pixel width) or a special break character (when `nrune < 0`, with `bc` holding the character—tab or newline—and `minwid` giving the minimum width for tab stops). The `wid` field of a tab box is initially set to

a large sentinel (5000) and recomputed contextually by `_frnewwid` based on the current x position and tab stop alignment.

⟨*struct* Frbox 292a⟩≡ (361b)
```
struct Frbox
{
    long  wid;  /* in pixels */

    long  nrune;  /* <0 ==> negate and treat as break char */
    union{
        // array<byte> UTF8?
        uchar *ptr;
        struct{
            short bc; /* break char */
            short minwid;
        };
    };
};
```

⟨*function* NRUNE 292b⟩≡ (361b)
```
#define NRUNE(b) ((b)->nrune < 0 ? 1 : (b)->nrune)
```

Adding boxes

The box array is a dynamic array with slack: `_fraddbox` inserts `n` empty slots at position `bn` by shifting everything after it up, growing the allocation by `SLOP` (25) extra slots to amortize future insertions. `_frdelbox` is the inverse: it frees the string data in the target boxes (`_frfreebox`) then closes the gap by shifting the tail down (`_frclosebox`).

⟨*constant* SLOP 292c⟩≡ (363c)
```
#define SLOP 25
```

⟨*function* _fraddbox 292d⟩≡ (363c)
```
void
_fraddbox(Frame *f, int bn, int n) /* add n boxes after bn, shift the rest up,
                 * box[bn+n]==box[bn] */
{
    int i;

    ⟨_fraddbox() sanity check bn 292f⟩
    if(f->nbox+n > f->nalloc)
        _frgrowbox(f, n+SLOP);
    for(i=f->nbox; --i>=bn; )
        f->box[i+n] = f->box[i];
    f->nbox+=n;
}
```
Uses SLOP-147 292c and _frgrowbox() 292e.

⟨*function* _frgrowbox 292e⟩≡ (363c)
```
void
_frgrowbox(Frame *f, int delta)
{
    f->nalloc += delta;
    f->box = realloc(f->box, f->nalloc*sizeof(Frbox));
    ⟨_frgrowbox() sanity check box 293a⟩
}
```

⟨_fraddbox() *sanity check bn* 292f⟩≡ (292d)
```
if(bn > f->nbox)
    drawerror(f->display, "_fraddbox");
```

⟨_frgrowbox() *sanity check box* 293a⟩≡ (292e)
```
if(f->box == nil)
    drawerror(f->display, "_frgrowbox");
```

Free boxes

⟨frclear() *free boxes* 293b⟩≡ (289d)
```
if(f->nbox)
    _frdelbox(f, 0, f->nbox-1);
if(f->box)
    free(f->box);
f->box = nil;
```
Uses _frdelbox() 293c.

⟨*function* _frdelbox 293c⟩≡ (363c)
```
void
_frdelbox(Frame *f, int n0, int n1) /* inclusive */
{
    ⟨_frdelbox() sanity check n0 and n1 293f⟩
    _frfreebox(f, n0, n1);
    _frclosebox(f, n0, n1);
}
```
Uses _frclosebox() 293e and _frfreebox() 293d.

⟨*function* _frfreebox 293d⟩≡ (363c)
```
void
_frfreebox(Frame *f, int n0, int n1) /* inclusive */
{
    int i;

    ⟨_frfreebox() sanity check n0 and n1 293g⟩
    n1++;
    for(i=n0; i<n1; i++)
        if(f->box[i].nrune >= 0)
            free(f->box[i].ptr);
}
```

⟨*function* _frclosebox 293e⟩≡ (363c)
```
void
_frclosebox(Frame *f, int n0, int n1) /* inclusive */
{
    int i;

    ⟨_frclosebox() sanity check n0 and n1 294a⟩
    n1++;
    for(i=n1; i<f->nbox; i++)
        f->box[i-(n1-n0)] = f->box[i];
    f->nbox -= n1-n0;
}
```

⟨_frdelbox() *sanity check n0 and n1* 293f⟩≡ (293c)
```
if(n0>=f->nbox || n1>=f->nbox || n1<n0)
    drawerror(f->display, "_frdelbox");
```

⟨_frfreebox() *sanity check n0 and n1* 293g⟩≡ (293d)
```
if(n1<n0)
    return;
if(n0>=f->nbox || n1>=f->nbox)
    drawerror(f->display, "_frfreebox");
```

⟨_frclosebox() *sanity check n0 and n1* 294a⟩≡ (293e)
```
if(n0>=f->nbox || n1>=f->nbox || n1<n0)
    drawerror(f->display, "_frclosebox");
```

D.2.2 Box string storage

Each text box owns a heap-allocated UTF-8 string (`ptr`). These helpers manage the allocations: `_frallocstr` rounds up to 16-byte chunks via the ROUNDUP macro (which uses bit masking for efficiency), and `_frinsure` grows a box's string buffer when a merge or insertion would overflow it. The 16-byte chunk alignment means small strings waste at most 15 bytes, while frequent small operations (splitting and merging boxes) avoid calling `malloc` every time. This is important because a single character insertion can trigger several box splits and merges as the frame relays out text around the insertion point.

⟨*constant CHUNK (libframe/frstr.c)* 294b⟩≡ (366a)
```
#define CHUNK 16
```

⟨*function* ROUNDUP 294c⟩≡ (366a)
```
#define ROUNDUP(n) ((n+CHUNK)&~(CHUNK-1))
```

⟨*function* _frallocstr 294d⟩≡ (366a)
```
uchar *
_frallocstr(Frame *f, unsigned n)
{
    uchar *p;

    p = malloc(ROUNDUP(n));
    if(p == nil)
        drawerror(f->display, "out of memory");
    return p;
}
```
Uses ROUNDUP-152 294c.

⟨*function* _frinsure 294e⟩≡ (366a)
```
void
_frinsure(Frame *f, int bn, unsigned n)
{
    Frbox *b;
    uchar *p;

    b = &f->box[bn];
    if(b->nrune < 0)
        drawerror(f->display, "_frinsure");
    if(ROUNDUP(b->nrune) > n) /* > guarantees room for terminal NUL */
        return;
    p = _frallocstr(f, n);
    b = &f->box[bn];
    memmove(p, b->ptr, NBYTE(b)+1);
    free(b->ptr);
    b->ptr = p;
}
```
Uses NBYTE 308d, ROUNDUP-152 294c, and _frallocstr() 294d.

D.2.3 Frame rune position, point, and box number

The frame library works with three coordinate systems that all refer to the same character in the visible text:

- A rune index (`p`, `ulong`): offset from the start of the frame (not from `w->org`—that translation is done by the caller). For example, `frm.p0 == 5` means the selection starts at the 5th visible rune.
- A box number (`bn`, `int`): which `Frbox` in the `f->box` array contains that rune. A box holds a run of characters or a single special character (tab, newline).
- A pixel point (`pt`, `Point`): the (x,y) coordinate on screen where that character is drawn.

The conversion functions are:

- `frptofchar(p)` converts a rune index to a Point (by walking boxes and measuring widths). Used to position the tick cursor and to compute where selection painting should start.
- `frcharofpt(pt)` converts a Point to a rune index. Used to translate mouse click coordinates into text positions.
- `_frfindbox(p)` finds the box number for a rune index, splitting a box if `p` does not fall on a box boundary. Used by `frinsert`[304] and `frdelete`[314] to locate their insertion/deletion point in the box array.

All three functions walk the box array linearly from the beginning. For a frame with `maxlines` of 50 and an average of 3–5 boxes per line, the walks visit 150–250 boxes—fast enough for interactive use.

`frptofchar()`

Given a rune index `p`, return the pixel Point of its upper-left corner. The implementation (`_frptofcharptb`[295b]) walks the box array from box `bn`, consuming runes from `p` as it goes. For each box, it first checks for line wrap via `_frcklinewrap`: if the box would overflow the right margin, the point wraps to the start of the next line. If `p` falls before this box (i.e., `p >= NRUNE(b)`), the box's rune count is subtracted from `p` and `_fradvance` moves the point past the box. If `p` falls inside this box, the function measures individual characters with `stringnwidth` to find the exact x offset within the box. For ASCII characters (`r < Runeself`), the UTF-8 decoding is a single-byte fast path; multibyte runes go through `chartorune`.

⟨*function* `frptofchar` 295a⟩≡ (365b)

```
Point
frptofchar(Frame *f, ulong p)
{
    return _frptofcharptb(f, p, f->r.min, 0);
}
```

Uses `_frptofcharptb()` 295b.

⟨*function* `_frptofcharptb` 295b⟩≡ (365b)

```
Point
_frptofcharptb(Frame *f, ulong p, Point pt, int bn)
{
    Frbox *b;
    uchar *s;
    int w, l;
    Rune r;

    for(b = &f->box[bn]; bn<f->nbox; bn++,b++){
        _frcklinewrap(f, &pt, b);
        l=NRUNE(b);
        // p is in this box
        if(p < l){
            if(b->nrune > 0)
                for(s=b->ptr; p>0; s+=w, p--){
                    r = *s;
```

```
                    if(r < Runeself)
                        w = 1;
                    else
                        w = chartorune(&r, (char*)s);
                    pt.x += stringnwidth(f->font, (char*)s, 1);
                    ⟨_frptofcharptb() sanity check r and pt 296b⟩
                }
            break; // found it
        }
        // else
        p -= 1;
        _fradvance(f, &pt, b);
    }
    return pt;
}
```

Uses NRUNE 292b, _fradvance() 296a, and _frcklinewrap() 296c.

⟨*function* _fradvance 296a⟩≡ (366b)

```
void
_fradvance(Frame *f, Point *p, Frbox *b)
{
    if(b->nrune<0 && b->bc=='\n'){
        p->x = f->r.min.x;
        p->y += f->font->height;
    }else
        p->x += b->wid;
}
```

⟨_frptofcharptb() *sanity check r and pt* 296b⟩≡ (295b)

```
if(r==0 || pt.x > f->r.max.x)
    drawerror(f->display, "frptofchar");
```

⟨*function* _frcklinewrap 296c⟩≡ (366b)

```
void
_frcklinewrap(Frame *f, Point *p, Frbox *b)
{
    if((b->nrune<0? b->minwid : b->wid) > f->r.max.x - p->x){
        p->x = f->r.min.x;
        p->y += f->font->height;
    }
}
```

⟨*function* _frcklinewrap0 296d⟩≡ (366b)

```
void
_frcklinewrap0(Frame *f, Point *p, Frbox *b)
{
    if(_frcanfit(f, *p, b) == 0){
        p->x = f->r.min.x;
        p->y += f->font->height;
    }
}
```

Uses _frcanfit() 297a.

Given a box and a starting position, `_frcanfit()` returns how many runes from that box fit before the right margin. For break boxes, it is a yes/no check (does `minwid` fit?). For text boxes, if the whole box fits it returns `nrune`; otherwise it measures characters one by one from the left until the margin is exceeded, returning the

count that fits. A return of 0 means the box must wrap to the next line.

⟨*function* _frcanfit 297a⟩≡ (366b)
```
int
_frcanfit(Frame *f, Point pt, Frbox *b)
{
    int left, w, nr;
    uchar *p;
    Rune r;

    left = f->r.max.x - pt.x;
    if(b->nrune < 0)
        return b->minwid <= left;
    if(left >= b->wid)
        return b->nrune;
    for(nr=0,p=b->ptr; *p; p+=w,nr++){
        r = *p;
        if(r < Runeself)
            w = 1;
        else
            w = chartorune(&r, (char*)p);
        left -= stringnwidth(f->font, (char*)p, 1);
        if(left < 0)
            return nr;
    }
    drawerror(f->display, "_frcanfit can't");
    return 0;
}
```

⟨*function* _frptofcharnb 297b⟩≡ (365b)
```
Point
_frptofcharnb(Frame *f, ulong p, int nb) /* doesn't do final _fradvance to next line */
{
    Point pt;
    int nbox;

    // save
    nbox = f->nbox;
    f->nbox = nb;
    pt = _frptofcharptb(f, p, f->r.min, 0);
    // restore
    f->nbox = nbox;
    return pt;
}
```
Uses _frptofcharptb() 295b.

frcharofpt()

The inverse of `frptofchar`[295a]: given a pixel Point (typically from a mouse click), return the rune index at that position. The function works in two passes: first it walks boxes to find the right line (comparing `qt.y` against `pt.y`), then it walks boxes on that line to find the right character (comparing `qt.x` against `pt.x`). The `_frgrid` helper snaps the y coordinate to a line boundary so that clicks in the inter-line space (between the baseline of one line and the top of the next) map to the correct row. It also clamps `p.x` to `r.max.x`, so clicks to the right of the text area map to the end of the line.

⟨*function* frcharofpt 297c⟩≡ (365b)
```
ulong
frcharofpt(Frame *f, Point pt)
{
```

```
    Point qt;
    int w;
    uchar *s;
    Frbox *b;
    int bn;
    ulong p;
    Rune r;

    pt = _frgrid(f, pt);

    qt = f->r.min;
    // find the line
    for(b=f->box, bn=0, p=0; bn < f->nbox && qt.y < pt.y; bn++,b++){
        _frcklinewrap(f, &qt, b);
        if(qt.y >= pt.y)
            break;
        _fradvance(f, &qt, b);
        p += NRUNE(b);
    }
    // find the box in the line
    for(; bn<f->nbox && qt.x<=pt.x; bn++,b++){
        _frcklinewrap(f, &qt, b);
        if(qt.y > pt.y)
            break;
        if(qt.x + b->wid > pt.x){
            if(b->nrune < 0)
                _fradvance(f, &qt, b);
            else{
                s = b->ptr;
                for(;;){
                    if((r = *s) < Runeself)
                        w = 1;
                    else
                        w = chartorune(&r, (char*)s);
                    ⟨frcharofpt() sanity check r 299a⟩
                    qt.x += stringnwidth(f->font, (char*)s, 1);
                    s += w;
                    if(qt.x > pt.x)
                        break;
                    p++;
                }
            }
        }else{
            p += NRUNE(b);
            _fradvance(f, &qt, b);
        }
    }
    return p;
}
```

Uses NRUNE 292b, _fradvance() 296a, _frcklinewrap() 296c, and _frgrid() 298.

⟨*function* _frgrid 298⟩≡ (365b)

```
static
Point
_frgrid(Frame *f, Point p)
{
    p.y -= f->r.min.y;
    p.y -= p.y%f->font->height;
    p.y += f->r.min.y;
    if(p.x > f->r.max.x)
```

```
        p.x = f->r.max.x;
    return p;
}
```

⟨frcharofpt() *sanity check r* 299a⟩≡ (297c)

```
if(r == 0)
    drawerror(f->display, "end of string in frcharofpt");
```

_frdrawtext()

The simplest drawing routine: it walks every box in the frame, checking for line wraps via `_frcklinewrap`[296c], and draws text boxes with `stringbg` (which renders the string and fills the background in a single draw operation, avoiding the flicker that would result from clearing the background first and then drawing text on top). Break boxes (tabs and newlines) are skipped—their space is filled by the caller or by `frselectpaint`[301]. Tab boxes have variable width that depends on their position, so they are handled by the alignment logic elsewhere rather than by this generic draw routine. This function is used by `frinsert`[304] in Phase 3 to draw the newly inserted text into the gap created by shifting existing content down. It draws into the temporary `frame` structure's box array, not the main frame's—the boxes are spliced into the main array afterward.

⟨*function* _frdrawtext 299b⟩≡ (364b)

```
void
_frdrawtext(Frame *f, Point pt, Image *text, Image *back)
{
    Frbox *b;
    int nb;
    static int x;

    for(nb=0,b=f->box; nb<f->nbox; nb++, b++){
        _frcklinewrap(f, &pt, b);
        if(b->nrune >= 0){
            stringbg(f->b, pt, text, ZP, f->font, (char*)b->ptr, back, ZP);
        }
        pt.x += b->wid;
    }
}
```

Uses `_frcklinewrap()` 296c.

D.3 Selection and repainting

D.3.1 Frame selection

The frame tracks a selection range (`p0`, `p1`) in character coordinates relative to the frame origin (not the buffer origin `w->org`—the translation is done by `wsetselect`[185]). When `p0 == p1`, there is no selection and the tick cursor is shown at that position instead. The selection is rendered by painting the background of the selected range with `cols[HIGH]` (light grey) and the text with `cols[HTEXT]`. The `frdrawsel`[317b] functions handle the common case of extending or shrinking a selection incrementally (e.g., as the user drags the mouse), only repainting the characters that changed between the old and new selection boundaries. This avoids repainting the entire selection on every mouse movement.

⟨Frame *text fields* 299c⟩+≡ (288a) ◁289b 307b▷

```
ulong p0, p1;  /* selection */
```

⟨frinit() *initialize other fields* 299d⟩+≡ (288d) ◁291d

```
f->p0 = 0;
f->p1 = 0;
```

The `frselect` function is the interactive selection loop, called when button 1 is pressed. It tracks the mouse until the button is released, incrementally painting and unpainting the selection as the cursor moves. The `reg` (region) variable is the key state: it tracks whether the current endpoint `p1` is above (−1), at (0), or below (+1) the anchor point `p0`. This determines whether extending the selection means painting forward or backward. When the user drags past the anchor (the region sign flips), the entire old selection is cleared and rebuilt in the opposite direction—this prevents visual artifacts. If the mouse moves above or below the frame rectangle, the `scroll` callback (set to `framescroll`[204a]) fires, scrolling the viewport and extending the selection to follow. The `scrled` flag prevents a double mouse read when scrolling was triggered, since the scroll callback may itself consume mouse events. The `do/while` loop continues as long as the same button is held. On each iteration, it compares the new mouse position `q` with the previous `p1` and only repaints the delta—the range between the old and new endpoints. This incremental approach means that even a large selection being extended by one character only repaints that one character.

⟨*function* `frselect` 300⟩≡ (365c)

```
void
frselect(Frame *f, Mousectl *mc) /* when called, button 1 is down */
{
    ulong p0, p1, q;
    Point mp, pt0, pt1, qt;
    int reg, b, scrled;

    mp = mc->m.xy;
    b = mc->m.buttons;

    f->modified = 0;
    frdrawsel(f, frptofchar(f, f->p0), f->p0, f->p1, 0);
    p0 = p1 = frcharofpt(f, mp);
    f->p0 = p0;
    f->p1 = p1;
    pt0 = frptofchar(f, p0);
    pt1 = frptofchar(f, p1);
    frdrawsel(f, pt0, p0, p1, 1);
    reg = 0;
    do{
        scrled = 0;
        if(f->scroll){
            if(mp.y < f->r.min.y){
                (*f->scroll)(f, -(f->r.min.y-mp.y)/(int)f->font->height-1);
                p0 = f->p1;
                p1 = f->p0;
                scrled = 1;
            }else if(mp.y > f->r.max.y){
                (*f->scroll)(f, (mp.y-f->r.max.y)/(int)f->font->height+1);
                p0 = f->p0;
                p1 = f->p1;
                scrled = 1;
            }
            if(scrled){
                if(reg != region(p1, p0))
                    q = p0, p0 = p1, p1 = q; /* undo the swap that will happen below */
                pt0 = frptofchar(f, p0);
                pt1 = frptofchar(f, p1);
                reg = region(p1, p0);
            }
        }
        q = frcharofpt(f, mp);
        if(p1 != q){
            if(reg != region(q, p0)){ /* crossed starting point; reset */
```

```
                if(reg > 0)
                    frdrawsel(f, pt0, p0, p1, 0);
                else if(reg < 0)
                    frdrawsel(f, pt1, p1, p0, 0);
                p1 = p0;
                pt1 = pt0;
                reg = region(q, p0);
                if(reg == 0)
                    frdrawsel(f, pt0, p0, p1, 1);
            }
            qt = frptofchar(f, q);
            if(reg > 0){
                if(q > p1)
                    frdrawsel(f, pt1, p1, q, 1);
                else if(q < p1)
                    frdrawsel(f, qt, q, p1, 0);
            }else if(reg < 0){
                if(q > p1)
                    frdrawsel(f, pt1, p1, q, 0);
                else
                    frdrawsel(f, qt, q, p1, 1);
            }
            p1 = q;
            pt1 = qt;
        }
        f->modified = 0;
        if(p0 < p1) {
            f->p0 = p0;
            f->p1 = p1;
        }
        else {
            f->p0 = p1;
            f->p1 = p0;
        }
        if(scrled)
            (*f->scroll)(f, 0);
        flushimage(f->display, 1);
        if(!scrled)
            readmouse(mc);
        mp = mc->m.xy;
    }while(mc->m.buttons == b);
}
```

Uses `frcharofpt()` 297c, `frdrawsel()` 317b, `frptofchar()` 295a, and `region()` 302a.

The `frselectpaint` helper fills a rectangular region between two Points with a solid color. It handles three cases: single-line (one rectangle from `p0` to `p1`), multi-line (fill the end of the first line, fill full middle lines, fill the start of the last line), and the degenerate case where `p0` is at the frame bottom. This is used both to highlight the selection background and to clear areas after deletions.

⟨*function* `frselectpaint` 301⟩≡ (365c)

```
void
frselectpaint(Frame *f, Point p0, Point p1, Image *col)
{
    int n;
    Point q0, q1;

    q0 = p0;
    q1 = p1;
    q0.y += f->font->height;
    q1.y += f->font->height;
```

```
    n = (p1.y-p0.y)/f->font->height;
    if(f->b == nil)
        drawerror(f->display, "frselectpaint b==0");
    if(p0.y == f->r.max.y)
        return;
    if(n == 0)
        draw(f->b, Rpt(p0, q1), col, nil, ZP);
    else{
        if(p0.x >= f->r.max.x)
            p0.x = f->r.max.x-1;
        draw(f->b, Rect(p0.x, p0.y, f->r.max.x, q0.y), col, nil, ZP);
        if(n > 1)
            draw(f->b, Rect(f->r.min.x, q0.y, f->r.max.x, p1.y),
                col, nil, ZP);
        draw(f->b, Rect(f->r.min.x, p1.y, q1.x, q1.y),
            col, nil, ZP);
    }
}
```

⟨*function* `region` 302a⟩≡ (365c)

```
static
int
region(int a, int b)
{
    if(a < b)
        return -1;
    if(a == b)
        return 0;
    return 1;
}
```

D.3.2 Repainting

A full repaint draws the entire frame in three bands using `frdrawsel0`[318] with the appropriate colors: unselected text before `p0` (black on white), selected text between `p0` and `p1` (black on light grey), and unselected text after `p1` (black on white again). The tick must be hidden before repainting and restored afterward, because the pixels under it change. Full repaints are expensive compared to incremental updates, so they are only triggered when the colors themselves change (window gaining/losing focus) or when the frame is rebuilt from scratch (after a resize). Normal typing and selection use the incremental `frinsert`[304]/`frdelete`[314]/`frdrawsel`[317b] paths.

⟨*function* `frredraw` 302b⟩≡ (364b)

```
void
frredraw(Frame *f)
{
    bool ticked;
    Point pt;

    if(f->p0 == f->p1){
        ticked = f->ticked;
        if(ticked)
            frtick(f, frptofchar(f, f->p0), false);
        // redraw the text
        frdrawsel0(f, frptofchar(f, 0), 0, f->nchars, f->cols[BACK], f->cols[TEXT]);
        if(ticked)
            frtick(f, frptofchar(f, f->p0), true);
        return;
    }
    // else, redraw the selection and the text
```

```
    pt = frptofchar(f, 0);
    pt = frdrawsel0(f, pt, 0,     f->p0,     f->cols[BACK], f->cols[TEXT]);
    pt = frdrawsel0(f, pt, f->p0, f->p1,     f->cols[HIGH], f->cols[HTEXT]);
    pt = frdrawsel0(f, pt, f->p1, f->nchars, f->cols[BACK], f->cols[TEXT]);
}
```

Uses `BACK` 289f, `HIGH` 289f, `HTEXT` 289f, `TEXT` 289f, `frdrawsel0()` 318, `frptofchar()` 295a, and `frtick()` 317a.

D.4 Incremental rendering

⟨*global* `frame` 303a⟩≡ (365a)

```
static Frame  frame;
```

⟨*struct* `points_frinsert` 303b⟩≡ (365a)

```
struct points_frinsert {
    Point pt0, pt1;
};
```

⟨`frinsert()` *locals* 303c⟩≡ (304)

```
Point pt0, pt1, opt0, ppt0, ppt1, pt;
Frbox *b;
int n, n0, nn0, y;
ulong cn0;
Image *col;
Rectangle r;
static struct points_frinsert *pts;
static int nalloc=0;
int npts;
```

Uses `points_frinsert` 303b.

⟨*constant* `DELTA` 303d⟩≡ (365a)

```
#define DELTA 25
```

The `frinsert` function is the heart of the frame library and its most complex routine. It inserts new runes into the display without redrawing everything, working in three phases: **Phase 1: bxscan.** The new runes (`sp` to `ep`) are converted into boxes in a temporary `frame` structure. `bxscan` walks the rune sequence, grouping consecutive printable characters into text boxes and creating special boxes for tabs and newlines. It also computes `ppt0`/`ppt1`—the pixel coordinates where the insertion will appear on screen. **Phase 2: x-alignment.** After inserting text, every existing box after the insertion point shifts right. The loop starting at "Find point where old and new x's line up" walks forward through existing boxes, tracking two points: `pt0` (where each box *is* now) and `pt1` (where it *will be* after insertion). It records both positions in the `pts` array. The loop terminates when `pt1.x==pt0.x`—meaning the old and new layouts have realigned (because a line wrap puts both back to the left margin)—or when `pt1` falls off the bottom of the frame. **Phase 3: move down.** The final loop walks backward through `pts`, using `draw` to blit each box from its old position (`pts[i].pt0`) to its new position (`pts[i].pt1`). Going backward avoids overwriting source pixels before they are copied. After the moves, the newly inserted text is drawn into the gap left behind, and the box array is spliced to include the new boxes from `bxscan`. Here is a visual example of inserting "XY" at position 5 in the text "Hello World", where the line is wide enough to hold everything:

```
Before:   |H|e|l|l|o| |W|o|r|l|d|
          0 1 2 3 4 5 6 7 8 9 ...    (rune indices)
                    ^
                 insertion point (p0=5)

Phase 1 (bxscan):   scan "XY" into temp boxes
                     ppt0 = pixel of pos 5
                     ppt1 = pixel of pos 7 (after XY)

Phase 2 (x-alignment): walk from pos 5 in old layout
          pt0 = where " World" is now
          pt1 = where " World" will be (shifted right by width of "XY")
          Record pairs: pts[0] = {pt0=" ", pt1=" shifted"}
                        pts[1] = {pt0="W", pt1="W shifted"}
                        ... until pt1.x == pt0.x (line aligned)

Phase 3 (move down):  blit boxes backward from old to new positions
          Then draw "XY" in the gap

After:    |H|e|l|l|o|X|Y| |W|o|r|l|d|
```

Here is the full `frinsert` code. It is long and dense, but the structure follows the three phases described above. The critical invariant is that `pt0` always tracks where a box *currently* is on screen, while `pt1` tracks where it *will be* after the insertion. The difference between `pt1` and `pt0` is the pixel displacement caused by the inserted text. The `pts` array stores (old, new) position pairs for the backward copy loop in Phase 3. The code handles several edge cases:

- Boxes that must be split because they would straddle a line boundary after being shifted right (detected by `_frcanfit`).
- Text that falls off the bottom of the frame (`pt1.y == r.max.y`), which is truncated and removed from the box array.
- The selection (`p0`/`p1`) that may need extending when the insertion point is within it.
- Multi-line shifts, where the bulk text below the alignment point is moved with two `draw` calls instead of box-by-box blitting.

⟨*function* `frinsert` 304⟩≡ (365a)

```
void
frinsert(Frame *f, Rune *sp, Rune *ep, ulong p0)
{
    ⟨frinsert() locals 303c⟩

    if(p0>f->nchars || sp==ep || f->b==nil)
        return;

    n0 = _frfindbox(f, 0, 0, p0);
    cn0 = p0;
    nn0 = n0;
    pt0 = _frptofcharnb(f, p0, n0);
    ppt0 = pt0;
    opt0 = pt0;
    pt1 = bxscan(f, sp, ep, &ppt0);
```

```
ppt1 = pt1;

if(n0 < f->nbox){
    b = &f->box[n0];
    _frcklinewrap(f, &pt0, b); /* for frdrawsel() */
    _frcklinewrap0(f, &ppt1, b);
}
f->modified = true;
/*
 * ppt0 and ppt1 are start and end of insertion as they will appear when
 * insertion is complete. pt0 is current location of insertion position
 * (p0); pt1 is terminal point (without line wrap) of insertion.
 */

⟨frinsert() remove tick 316a⟩

/*
 * Find point where old and new x's line up
 * Invariants:
 * pt0 is where the next box (b, n0) is now
 * pt1 is where it will be after the insertion
 * If pt1 goes off the rectangle, we can toss everything from there on
 */
for(b = &f->box[n0],npts=0;
     pt1.x!=pt0.x && pt1.y!=f->r.max.y && n0<f->nbox; b++,n0++,npts++){
    _frcklinewrap(f, &pt0, b);
    _frcklinewrap0(f, &pt1, b);
    if(b->nrune > 0){
        n = _frcanfit(f, pt1, b);
        ⟨frinsert() sanity check n canfit 307a⟩
        if(n != b->nrune){
            _frsplitbox(f, n0, n);
            b = &f->box[n0];
        }
    }
    if(npts == nalloc){
        pts = realloc(pts, (npts+DELTA)*sizeof(pts[0]));
        nalloc += DELTA;
        b = &f->box[n0];
    }
    pts[npts].pt0 = pt0;
    pts[npts].pt1 = pt1;
    /* has a text box overflowed off the frame? */
    if(pt1.y == f->r.max.y)
        break;
    _fradvance(f, &pt0, b);
    pt1.x += _frnewwid(f, pt1, b);
    cn0 += NRUNE(b);
}
if(pt1.y > f->r.max.y)
    drawerror(f->display, "frinsert pt1 too far");
if(pt1.y==f->r.max.y && n0<f->nbox){
    f->nchars -= _frstrlen(f, n0);
    _frdelbox(f, n0, f->nbox-1);
}
if(n0 == f->nbox)
    f->nlines = (pt1.y-f->r.min.y)/f->font->height+(pt1.x>f->r.min.x);
else if(pt1.y!=pt0.y){
    int q0, q1;
```

```
    y = f->r.max.y;
    q0 = pt0.y+f->font->height;
    q1 = pt1.y+f->font->height;
    f->nlines += (q1-q0)/f->font->height;
    if(f->nlines > f->maxlines)
        chopframe(f, ppt1, p0, nn0);
    if(pt1.y < y){
        r = f->r;
        r.min.y = q1;
        r.max.y = y;
        if(q1 < y)
            draw(f->b, r, f->b, nil, Pt(f->r.min.x, q0));
        r.min = pt1;
        r.max.x = pt1.x+(f->r.max.x-pt0.x);
        r.max.y = q1;
        draw(f->b, r, f->b, nil, pt0);
    }
}
/*
 * Move the old stuff down to make room.  The loop will move the stuff
 * between the insertion and the point where the x's lined up.
 * The draw()s above moved everything down after the point they lined up.
 */
for((y=pt1.y==f->r.max.y?pt1.y:0),b = &f->box[n0-1]; --npts>=0; --b){
    pt = pts[npts].pt1;
    if(b->nrune > 0){
        r.min = pt;
        r.max = r.min;
        r.max.x += b->wid;
        r.max.y += f->font->height;
        draw(f->b, r, f->b, nil, pts[npts].pt0);
        /* clear bit hanging off right */
        if(npts==0 && pt.y>pt0.y){
            /*
             * first new char is bigger than first char we're
             * displacing, causing line wrap. ugly special case.
             */
            r.min = opt0;
            r.max = opt0;
            r.max.x = f->r.max.x;
            r.max.y += f->font->height;
            if(f->p0<=cn0 && cn0<f->p1) /* b+1 is inside selection */
                col = f->cols[HIGH];
            else
                col = f->cols[BACK];
            draw(f->b, r, col, nil, r.min);
        }else if(pt.y < y){
            r.min = pt;
            r.max = pt;
            r.min.x += b->wid;
            r.max.x = f->r.max.x;
            r.max.y += f->font->height;
            if(f->p0<=cn0 && cn0<f->p1) /* b+1 is inside selection */
                col = f->cols[HIGH];
            else
                col = f->cols[BACK];
            draw(f->b, r, col, nil, r.min);
        }
        y = pt.y;
        cn0 -= b->nrune;
```

```
        }else{
            r.min = pt;
            r.max = pt;
            r.max.x += b->wid;
            r.max.y += f->font->height;
            if(r.max.x >= f->r.max.x)
                r.max.x = f->r.max.x;
            cn0--;
            if(f->p0<=cn0 && cn0<f->p1) /* b is inside selection */
                col = f->cols[HIGH];
            else
                col = f->cols[BACK];
            draw(f->b, r, col, nil, r.min);
            y = 0;
            if(pt.x == f->r.min.x)
                y = pt.y;
        }
    }
    /* insertion can extend the selection, so the condition here is different */
    if(f->p0<p0 && p0<=f->p1)
        col = f->cols[HIGH];
    else
        col = f->cols[BACK];

    frselectpaint(f, ppt0, ppt1, col);

    _frdrawtext(&frame, ppt0, f->cols[TEXT], col);

    _fraddbox(f, nn0, frame.nbox);
    for(n=0; n<frame.nbox; n++)
        f->box[nn0+n] = frame.box[n];
    if(nn0>0 && f->box[nn0-1].nrune>=0 && ppt0.x-f->box[nn0-1].wid>=f->r.min.x){
        --nn0;
        ppt0.x -= f->box[nn0].wid;
    }
    n0 += frame.nbox;
    _frclean(f, ppt0, nn0, n0<f->nbox-1? n0+1 : n0);

    f->nchars += frame.nchars;
    if(f->p0 >= p0)
        f->p0 += frame.nchars;
    if(f->p0 > f->nchars)
        f->p0 = f->nchars;
    if(f->p1 >= p0)
        f->p1 += frame.nchars;
    if(f->p1 > f->nchars)
        f->p1 = f->nchars;

    ⟨frinsert() draw tick 316b⟩
}
```

Uses BACK 289f, DELTA-148 303d, HIGH 289f, NRUNE 292b, TEXT 289f, _fraddbox() 292d, _fradvance() 296a, _frcanfit() 297a, _frcklinewrap() 296c, _frcklinewrap0() 296d, _frclean() 313a, _frdelbox() 293c, _frdrawtext() 299b, _frfindbox() 308a, _frnewwid() 312a, _frptofcharnb() 297b, _frsplitbox() 308b, _frstrlen() 312c, bxscan() 310, chopframe() 312d, frame-150 303a, and frselectpaint() 301.

⟨frinsert() *sanity check n canfit* 307a⟩≡ (304)

```
  if(n == 0)
      drawerror(f->display, "_frcanfit==0");
```

⟨Frame *text fields* 307b⟩+≡ (288a) ◁299c

```
bool modified; /* changed since frselect() */
```

Finding a box boundary: `_frfindbox()`

Given a rune index `q`, find the box that contains it. If `q` falls in the middle of a box (not on a boundary), `_frsplitbox` splits that box so that `q` becomes a boundary. This ensures that `frinsert`[304] and `frdelete`[314] can splice the box array cleanly at the insertion/deletion point. The split works by duplicating the box (`dupbox`) and then trimming each copy from opposite ends: `truncatebox` shortens the first copy by dropping its last `n` characters (keeping the left portion), and `chopbox` shortens the second copy by dropping its first `n` characters (keeping the right portion). Both functions recompute `wid` by calling `stringwidth` on the remaining text. For example, splitting the box "Hello" at position 3 produces two boxes: "Hel" and "lo", each with its own width and string allocation.

⟨*function* `_frfindbox` 308a⟩≡ (363c)

```
/* find box containing q and put q on a box boundary */
int
_frfindbox(Frame *f, int bn, ulong p, ulong q)
{
    Frbox *b;

    for(b = &f->box[bn]; bn < f->nbox && p+NRUNE(b) <= q; bn++, b++)
        p += NRUNE(b);
    if(p != q)
        _frsplitbox(f, bn++, (int)(q-p));
    return bn;
}
```

Uses `NRUNE` 292b and `_frsplitbox()` 308b.

⟨*function* `_frsplitbox` 308b⟩≡ (363c)

```
void
_frsplitbox(Frame *f, int bn, int n)
{
    dupbox(f, bn);
    truncatebox(f, &f->box[bn], f->box[bn].nrune-n);
    chopbox(f, &f->box[bn+1], n);
}
```

Uses `chopbox()` 309b, `dupbox()` 308c, and `truncatebox()` 309a.

⟨*function* `dupbox` 308c⟩≡ (363c)

```
static
void
dupbox(Frame *f, int bn)
{
    uchar *p;

    if(f->box[bn].nrune < 0)
        drawerror(f->display, "dupbox");
    _fraddbox(f, bn, 1);
    if(f->box[bn].nrune >= 0){
        p = _frallocstr(f, NBYTE(&f->box[bn])+1);
        strcpy((char*)p, (char*)f->box[bn].ptr);
        f->box[bn+1].ptr = p;
    }
}
```

Uses `NBYTE` 308d, `_fraddbox()` 292d, and `_frallocstr()` 294d.

⟨*function* `NBYTE` 308d⟩≡ (361b)

```
#define NBYTE(b) strlen((char*)(b)->ptr)
```

⟨*function* truncatebox 309a⟩≡ (363c)
```
static
void
truncatebox(Frame *f, Frbox *b, int n) /* drop last n chars; no allocation done */
{
    if(b->nrune<0 || b->nrune<n)
        drawerror(f->display, "truncatebox");
    b->nrune -= n;
    runeindex(b->ptr, b->nrune)[0] = 0;
    b->wid = stringwidth(f->font, (char *)b->ptr);
}
```
Uses `runeindex()` 309c.

⟨*function* chopbox 309b⟩≡ (363c)
```
static
void
chopbox(Frame *f, Frbox *b, int n) /* drop first n chars; no allocation done */
{
    char *p;

    if(b->nrune<0 || b->nrune<n)
        drawerror(f->display, "chopbox");
    p = (char*)runeindex(b->ptr, n);
    memmove((char*)b->ptr, p, strlen(p)+1);
    b->nrune -= n;
    b->wid = stringwidth(f->font, (char *)b->ptr);
}
```
Uses `runeindex()` 309c.

⟨*function* runeindex 309c⟩≡ (363c)
```
static
uchar*
runeindex(uchar *p, int n)
{
    int i, w;
    Rune rune;

    for(i=0; i<n; i++,p+=w)
        if(*p < Runeself)
            w = 1;
        else{
            w = chartorune(&rune, (char*)p);
            USED(rune);
        }
    return p;
}
```

Runes to boxes: `bxscan()`

The `bxscan` function converts a sequence of runes into the frame's internal box representation, populating the static `frame` structure. The outer loop creates one box per iteration: if the current rune is a tab or newline, it gets a special box (`nrune == -1`) with `bc` set to the character; otherwise, consecutive printable characters are accumulated into a temporary UTF-8 buffer (`tmp`) until a tab, newline, or buffer overflow is reached, then stored as a text box. The function stops after filling `maxlines` worth of newlines—there is no point scanning text that would fall off the bottom of the frame. Finally, `_frdraw` lays out the boxes starting at `*ppt` and returns the endpoint, which becomes `pt1` in `frinsert`.

⟨*constant* TMPSIZE 309d⟩≡ (365a)
```
#define TMPSIZE 256
```

The `bxscan` function is the front end of `frinsert`[304]'s three-phase pipeline. It iterates over the new runes, building boxes in the static `frame` structure. For each rune, if it is a tab or newline, a break box is created with `nrune == -1`; otherwise, consecutive printable characters are accumulated into a temporary UTF-8 buffer `tmp` (up to `TMPSIZE` bytes), then stored as a text box with a heap-allocated copy. The function counts newlines and stops after filling `maxlines` worth of lines—there is no point scanning text that would not fit in the frame anyway. Finally, `_frdraw` lays out the boxes starting at `*ppt`, splitting any box that does not fit on the current line, and returns the endpoint. This endpoint becomes `pt1` in `frinsert`—the coordinate where the new text ends.

⟨*function* `bxscan` 310⟩≡ (365a)

```
static
Point
bxscan(Frame *f, Rune *sp, Rune *ep, Point *ppt)
{
    int w, c, nb, delta, nl, nr, rw;
    Frbox *b;
    char *s, tmp[TMPSIZE+3]; /* +3 for rune overflow */
    uchar *p;

    frame.r = f->r;
    frame.b = f->b;
    frame.font = f->font;
    frame.maxtab = f->maxtab;
    frame.nbox = 0;
    frame.nchars = 0;
    memmove(frame.cols, f->cols, sizeof frame.cols);
    delta = DELTA;
    nl = 0;
    for(nb=0; sp<ep && nl<=f->maxlines; nb++,frame.nbox++){
        if(nb == frame.nalloc){
            _frgrowbox(&frame, delta);
            if(delta < 10000)
                delta *= 2;
        }
        b = &frame.box[nb];
        c = *sp;
        if(c=='\t' || c=='\n'){
            b->bc = c;
            b->wid = 5000;
            b->minwid = (c=='\n')? 0 : stringwidth(frame.font, " ");
            b->nrune = -1;
            if(c=='\n')
                nl++;
            frame.nchars++;
            sp++;
        }else{
            s = tmp;
            nr = 0;
            w = 0;
            while(sp < ep){
                c = *sp;
                if(c=='\t' || c=='\n')
                    break;
                rw = runetochar(s, sp);
                if(s+rw >= tmp+TMPSIZE)
                    break;
                w += runestringnwidth(frame.font, sp, 1);
                sp++;
                s += rw;
```

```
                nr++;
            }
            *s++ = 0;
            p = _frallocstr(f, s-tmp);
            b = &frame.box[nb];
            b->ptr = p;
            memmove(p, tmp, s-tmp);
            b->wid = w;
            b->nrune = nr;
            frame.nchars += nr;
        }
    }
    _frcklinewrap0(f, ppt, &frame.box[0]);
    return _frdraw(&frame, *ppt);
}
```

Uses `DELTA-148` 303d, `TMPSIZE-149` 309d, `_frallocstr()` 294d, `_frcklinewrap0()` 296d, `_frdraw()` 311, `_frgrowbox()` 292e, and `frame-150` 303a.

The `_frdraw` function does a dry-run layout of the boxes in the temporary `frame`, computing where each box would appear on screen. It handles line wrapping (splitting boxes that overflow the right margin via `_frsplitbox`) and tab-stop computation (via `_frnewwid`). It also truncates boxes that fall below the bottom of the frame. The return value is the endpoint—the pixel Point after the last box.

⟨*function* `_frdraw` 311⟩≡ (364b)

```
Point
_frdraw(Frame *f, Point pt)
{
    Frbox *b;
    int nb, n;

    for(b=f->box,nb=0; nb<f->nbox; nb++, b++){
        _frcklinewrap0(f, &pt, b);
        if(pt.y == f->r.max.y){
            f->nchars -= _frstrlen(f, nb);
            _frdelbox(f, nb, f->nbox-1);
            break;
        }
        if(b->nrune > 0){
            n = _frcanfit(f, pt, b);
            if(n == 0)
                drawerror(f->display, "_frcanfit==0");
            if(n != b->nrune){
                _frsplitbox(f, nb, n);
                b = &f->box[nb];
            }
            pt.x += b->wid;
        }else{
            if(b->bc == '\n'){
                pt.x = f->r.min.x;
                pt.y+=f->font->height;
            }else
                pt.x += _frnewwid(f, pt, b);
        }
    }
    return pt;
}
```

Uses `_frcanfit()` 297a, `_frcklinewrap0()` 296d, `_frdelbox()` 293c, `_frnewwid()` 312a, `_frsplitbox()` 308b, and `_frstrlen()` 312c.

Tab width depends on context—a tab at x=100 is a different width than one at x=200, because it must align

to the next maxtab boundary. _frnewwid0 computes the tab width for a box at position pt: it rounds up to the next tab stop (relative to the left margin), ensuring a minimum width of one space character (minwid) and wrapping to the next line if the tab would overflow the right edge. _frnewwid is a thin wrapper that also stores the result in b->wid so subsequent uses of this box get the cached value.

⟨*function* _frnewwid 312a⟩≡ (366b)

```
int
_frnewwid(Frame *f, Point pt, Frbox *b)
{
    b->wid = _frnewwid0(f, pt, b);
    return b->wid;
}
```

Uses _frnewwid0() 312b.

⟨*function* _frnewwid0 312b⟩≡ (366b)

```
int
_frnewwid0(Frame *f, Point pt, Frbox *b)
{
    int c, x;

    c = f->r.max.x;
    x = pt.x;
    if(b->nrune>=0 || b->bc!='\t')
        return b->wid;
    if(x+b->minwid > c)
        x = pt.x = f->r.min.x;
    x += f->maxtab;
    x -= (x-f->r.min.x)%f->maxtab;
    if(x-pt.x<b->minwid || x>c)
        x = pt.x+b->minwid;
    return x-pt.x;
}
```

⟨*function* _frstrlen 312c⟩≡ (364b)

```
int
_frstrlen(Frame *f, int nb)
{
    int n;

    for(n=0; nb<f->nbox; nb++)
        n += NRUNE(&f->box[nb]);
    return n;
}
```

Uses NRUNE 292b.

Truncating overflow: chopframe()

When an insertion causes nlines to exceed maxlines, the frame has overflowed. chopframe walks forward from the insertion point until the pixel position reaches f->r.max.y (the bottom of the frame), counts how many characters fit, then deletes all boxes after that point. This truncates the frame to exactly maxlines lines.

⟨*function* chopframe 312d⟩≡ (365a)

```
static
void
chopframe(Frame *f, Point pt, ulong p, int bn)
{
    Frbox *b;

    for(b = &f->box[bn]; ; b++){
```

```
        if(b >= &f->box[f->nbox])
            drawerror(f->display, "endofframe");
        _frcklinewrap(f, &pt, b);
        if(pt.y >= f->r.max.y)
            break;
        p += NRUNE(b);
        _fradvance(f, &pt, b);
    }
    f->nchars = p;
    f->nlines = f->maxlines;
    if(b<&f->box[f->nbox])    /* BUG */
        _frdelbox(f, (int)(b-f->box), f->nbox-1);
}
```

Uses `NRUNE` 292b, `_fradvance()` 296a, `_frcklinewrap()` 296c, and `_frdelbox()` 293c.

Merging boxes: `_frclean()`

After an insertion or deletion, the box array may contain adjacent text boxes that could be merged—for instance, if a box was split for an insertion and the insertion was later deleted. `_frclean` scans boxes `n0` through `n1` and merges any pair of adjacent text boxes that fit on the same line (their combined width does not exceed the right margin). Merging reduces the box count and speeds up future walks. The function also walks the remaining boxes to update `lastlinefull`, which indicates whether the last line of text reaches the bottom of the frame. This flag is checked by `wfill`[184c] to decide whether more text needs to be loaded from the rune buffer.

⟨*function* `_frclean` 313a⟩≡ (366b)

```
void
_frclean(Frame *f, Point pt, int n0, int n1) /* look for mergeable boxes */
{
    Frbox *b;
    int nb, c;

    c = f->r.max.x;
    for(nb=n0; nb<n1-1; nb++){
        b = &f->box[nb];
        _frcklinewrap(f, &pt, b);
        while(b[0].nrune>=0 && nb<n1-1 && b[1].nrune>=0 && pt.x+b[0].wid+b[1].wid<c){
            _frmergebox(f, nb);
            n1--;
            b = &f->box[nb];
        }
        _fradvance(f, &pt, &f->box[nb]);
    }
    for(; nb<f->nbox; nb++){
        b = &f->box[nb];
        _frcklinewrap(f, &pt, b);
        _fradvance(f, &pt, &f->box[nb]);
    }
    f->lastlinefull = 0;
    if(pt.y >= f->r.max.y)
        f->lastlinefull = 1;
}
```

Uses `_fradvance()` 296a, `_frcklinewrap()` 296c, and `_frmergebox()` 313b.

⟨*function* `_frmergebox` 313b⟩≡ (363c)

```
void
_frmergebox(Frame *f, int bn)  /* merge bn and bn+1 */
{
    Frbox *b;
```

```
    b = &f->box[bn];
    _frinsure(f, bn, NBYTE(&b[0])+NBYTE(&b[1])+1);
    strcpy((char*)runeindex(b[0].ptr, b[0].nrune), (char*)b[1].ptr);
    b[0].wid += b[1].wid;
    b[0].nrune += b[1].nrune;
    _frdelbox(f, bn+1, bn+1);
}
```

Uses `NBYTE` 308d, `_frdelbox()` 293c, `_frinsure()` 294e, and `runeindex()` 309c.

D.4.1 Incremental delete: `frdelete()`

The `frdelete` function is the inverse of `frinsert`[304]: it removes characters `p0` through `p1` from the display and collapses the remaining text upward. Like `frinsert`, it works incrementally by tracking two positions:

- `pt0`: where each surviving box *will be* after deletion (shifted left).
- `pt1`: where each box *currently is*.

The loop copies boxes from `pt1` to `pt0` using `draw`, closing the gap left by the deleted text. When the deletion spans multiple lines, two bulk `draw` calls shift entire line bands upward. The vacated area at the bottom (where text used to be but no longer is) is filled with the background color. The return value is the number of lines freed (old `nlines` minus new `nlines`). The caller uses this to decide whether to call `wfill`[184c] to load more text from the rune buffer into the newly available space at the bottom of the frame.

The `frdelete` function removes characters `p0` through `p1`. Like `frinsert`, it works by tracking two positions: `pt0` (where each box will end up after deletion) and `pt1` (where it currently is). The loop copies boxes from their old positions to their new ones, collapsing the gap. When the deletion spans multiple lines, two `draw` calls shift the bulk text up: one for the partial first line and one for the remaining full lines. The return value is the number of lines freed, which the caller (`wsetorigin`[183] or `wdelete`[193a]) uses to decide whether to call `wfill` to fill in text from below the viewport.

⟨*function* `frdelete` 314⟩≡ (364a)

```
int
frdelete(Frame *f, ulong p0, ulong p1)
{
    Point pt0, pt1, ppt0;
    Frbox *b;
    int n0, n1, n;
    ulong cn1;
    Rectangle r;
    int nn0;
    Image *col;

    if(p0>=f->nchars || p0==p1 || f->b==nil)
        return 0;
    if(p1 > f->nchars)
        p1 = f->nchars;
    n0 = _frfindbox(f, 0, 0, p0);
    if(n0 == f->nbox)
        drawerror(f->display, "off end in frdelete");
    n1 = _frfindbox(f, n0, p0, p1);
    pt0 = _frptofcharnb(f, p0, n0);
    pt1 = frptofchar(f, p1);
    if(f->p0 == f->p1)
        frtick(f, frptofchar(f, f->p0), 0);
    nn0 = n0;
    ppt0 = pt0;
```

```
_frfreebox(f, n0, n1-1);
f->modified = 1;

/*
 * Invariants:
 *  - pt0 points to beginning, pt1 points to end
 *  - n0 is box containing beginning of stuff being deleted
 *  - n1, b are box containing beginning of stuff to be kept after deletion
 *  - cn1 is char position of n1
 *  - f->p0 and f->p1 are not adjusted until after all deletion is done
 */
b = &f->box[n1];
cn1 = p1;
while(pt1.x!=pt0.x && n1<f->nbox){
    _frcklinewrap0(f, &pt0, b);
    _frcklinewrap(f, &pt1, b);
    n = _frcanfit(f, pt0, b);
    if(n==0)
        drawerror(f->display, "_frcanfit==0");
    r.min = pt0;
    r.max = pt0;
    r.max.y += f->font->height;
    if(b->nrune > 0){
        if(n != b->nrune){
            _frsplitbox(f, n1, n);
            b = &f->box[n1];
        }
        r.max.x += b->wid;
        draw(f->b, r, f->b, nil, pt1);
        cn1 += b->nrune;
    }else{
        r.max.x += _frnewwid0(f, pt0, b);
        if(r.max.x > f->r.max.x)
            r.max.x = f->r.max.x;
        col = f->cols[BACK];
        if(f->p0<=cn1 && cn1<f->p1)
            col = f->cols[HIGH];
        draw(f->b, r, col, nil, pt0);
        cn1++;
    }
    _fradvance(f, &pt1, b);
    pt0.x += _frnewwid(f, pt0, b);
    f->box[n0++] = f->box[n1++];
    b++;
}
if(n1==f->nbox && pt0.x!=pt1.x) /* deleting last thing in window; must clean up */
    frselectpaint(f, pt0, pt1, f->cols[BACK]);
if(pt1.y != pt0.y){
    Point pt2;

    pt2 = _frptofcharptb(f, 32767, pt1, n1);
    if(pt2.y > f->r.max.y)
        drawerror(f->display, "frptofchar in frdelete");
    if(n1 < f->nbox){
        int q0, q1, q2;

        q0 = pt0.y+f->font->height;
        q1 = pt1.y+f->font->height;
        q2 = pt2.y+f->font->height;
        if(q2 > f->r.max.y)
```

```
            q2 = f->r.max.y;
        draw(f->b, Rect(pt0.x, pt0.y, pt0.x+(f->r.max.x-pt1.x), q0),
            f->b, nil, pt1);
        draw(f->b, Rect(f->r.min.x, q0, f->r.max.x, q0+(q2-q1)),
            f->b, nil, Pt(f->r.min.x, q1));
        frselectpaint(f, Pt(pt2.x, pt2.y-(pt1.y-pt0.y)), pt2, f->cols[BACK]);
    }else
        frselectpaint(f, pt0, pt2, f->cols[BACK]);
    }
    _frclosebox(f, n0, n1-1);
    if(nn0>0 && f->box[nn0-1].nrune>=0 && ppt0.x-f->box[nn0-1].wid>=(int)f->r.min.x){
        --nn0;
        ppt0.x -= f->box[nn0].wid;
    }
    _frclean(f, ppt0, nn0, n0<f->nbox-1? n0+1 : n0);
    if(f->p1 > p1)
        f->p1 -= p1-p0;
    else if(f->p1 > p0)
        f->p1 = p0;
    if(f->p0 > p1)
        f->p0 -= p1-p0;
    else if(f->p0 > p0)
        f->p0 = p0;
    f->nchars -= p1-p0;
    if(f->p0 == f->p1)
        frtick(f, frptofchar(f, f->p0), 1);
    pt0 = frptofchar(f, f->nchars);
    n = f->nlines;
    f->nlines = (pt0.y-f->r.min.y)/f->font->height+(pt0.x>f->r.min.x);
    return n - f->nlines;
}
```

Uses `BACK` 289f, `HIGH` 289f, `_fradvance()` 296a, `_frcanfit()` 297a, `_frcklinewrap()` 296c, `_frcklinewrap0()` 296d, `_frclean()` 313a, `_frclosebox()` 293e, `_frfindbox()` 308a, `_frfreebox()` 293d, `_frnewwid()` 312a, `_frnewwid0()` 312b, `_frptofcharnb()` 297b, `_frptofcharptb()` 295b, `_frsplitbox()` 308b, `frptofchar()` 295a, `frselectpaint()` 301, and `frtick()` 317a.

D.4.2 Drawing the tick: `frtick()`

The tick (text cursor) is drawn by saving the pixels underneath into `tickback`, then drawing the tick image on top. To remove the tick, the saved pixels are restored. This save/restore approach is faster than redrawing the text, but it means the tick must be removed before any text operation that changes the display under it, and redrawn afterward.

⟨`frinsert()` *remove tick* 316a⟩≡ (304)

```
if(f->p0 == f->p1)
    frtick(f, frptofchar(f, f->p0), false);
```

Uses `frptofchar()` 295a and `frtick()` 317a.

⟨`frinsert()` *draw tick* 316b⟩≡ (304)

```
if(f->p0 == f->p1)
    frtick(f, frptofchar(f, f->p0), true);
```

Uses `frptofchar()` 295a and `frtick()` 317a.

The tick drawing is a two-phase operation. When `ticked` is true: save the pixels at `pt` into `tickback`, then draw the `tick` image on top. When `ticked` is false: restore the saved pixels from `tickback`. The `pt.x--` offset

makes the tick visually sit just left of the character it precedes. If `r.max.x` would exceed the frame boundary, it is clamped to prevent drawing outside the frame.

⟨*function* `frtick` 317a⟩≡ (364b)

```
void
frtick(Frame *f, Point pt, bool ticked)
{
    Rectangle r;

    if(f->ticked==ticked || f->tick==nil || !ptinrect(pt, f->r))
        return;
    pt.x--; /* looks best just left of where requested */
    r = Rect(pt.x, pt.y, pt.x + FRTICKW, pt.y + f->font->height);
    /* can go into left border but not right */
    if(r.max.x > f->r.max.x)
        r.max.x = f->r.max.x;
    if(ticked){
        draw(f->tickback, f->tickback->r, f->b, nil, pt);
        draw(f->b, r, f->tick, nil, ZP);
    }else
        draw(f->b, r, f->tickback, nil, ZP);
    f->ticked = ticked;
}
```

Uses `FRTICKW` 290d.

D.4.3 Drawing the text and the selection: `frdrawsel()`

Despite the name, `frdrawsel` is used for *both* selected and unselected regions—the `issel` parameter just controls which colors are used. When `p0 == p1` (no selection range), it draws or erases the tick cursor instead. The heavy lifting is in `frdrawsel0`, which walks the box array from `p0` to `p1`, painting each box's background rectangle and then rendering the text on top with `stringnbg`. It handles partial boxes (when the region starts or ends mid-box) by advancing the pointer and adjusting the rune count. End-of-line fill is done by detecting when `_frcklinewrap` wraps to a new line and filling the remainder of the previous line with the background color.

⟨*function* `frdrawsel` 317b⟩≡ (364b)

```
void
frdrawsel(Frame *f, Point pt, ulong p0, ulong p1, bool issel)
{
    Image *back, *text;

    if(f->ticked)
        frtick(f, frptofchar(f, f->p0), false);

    if(p0 == p1){
        frtick(f, pt, issel);
        return;
    }
    // else

    if(issel){
        back = f->cols[HIGH];
        text = f->cols[HTEXT];
    }else{
        back = f->cols[BACK];
        text = f->cols[TEXT];
    }

    frdrawsel0(f, pt, p0, p1, back, text);
```

```
}
```

Uses BACK 289f, HIGH 289f, HTEXT 289f, TEXT 289f, frdrawsel0() 318, frptofchar() 295a, and frtick() 317a.

The `frdrawsel0` workhorse draws a range of text with given foreground and background colors. It walks the box array, painting each box's background rectangle and rendering text on top. It handles three tricky cases: (1) the region starts mid-box (advance the byte pointer and reduce `nr`), (2) the region ends mid-box (reduce `nr` without advancing), and (3) line-wrap fill (when wrapping to a new line, fill the remainder of the previous line with the background color so there is no gap).

⟨*function* `frdrawsel0` 318⟩≡ (364b)

```
Point
frdrawsel0(Frame *f, Point pt, ulong p0, ulong p1, Image *back, Image *text)
{
    Frbox *b;
    int nb, nr, w, x, trim;
    Point qt;
    uint p;
    char *ptr;

    p = 0;
    b = f->box;
    trim = 0;
    for(nb=0; nb<f->nbox && p<p1; nb++){
        //todo: nr = NRUNE(b);
        nr = b->nrune;
        if(nr < 0)
            nr = 1;

        if(p+nr <= p0)
            goto Continue;
        if(p >= p0){
            qt = pt;
            _frcklinewrap(f, &pt, b);
            /* fill in the end of a wrapped line */
            if(pt.y > qt.y)
                draw(f->b, Rect(qt.x, qt.y, f->r.max.x, pt.y), back, nil, qt);
        }
        ptr = (char*)b->ptr;
        if(p < p0){ /* beginning of region: advance into box */
            ptr += nbytes(ptr, p0-p);
            nr -= (p0-p);
            p = p0;
        }
        trim = 0;
        if(p+nr > p1){ /* end of region: trim box */
            nr -= (p+nr)-p1;
            trim = 1;
        }
        if(b->nrune<0 || nr==b->nrune)
            w = b->wid;
        else
            w = stringnwidth(f->font, ptr, nr);
        x = pt.x+w;
        if(x > f->r.max.x)
            x = f->r.max.x;
        draw(f->b, Rect(pt.x, pt.y, x, pt.y+f->font->height), back, nil, pt);
        if(b->nrune >= 0)
            stringnbg(f->b, pt, text, ZP, f->font, ptr, nr, back, ZP);
        pt.x += w;
        Continue:
```

```
            b++;
            p += nr;
        }
        /* if this is end of last plain text box on wrapped line, fill to end of line */
        if(p1>p0 &&  b>f->box && b<f->box+f->nbox && b[-1].nrune>0 && !trim){
            qt = pt;
            _frcklinewrap(f, &pt, b);
            if(pt.y > qt.y)
                draw(f->b, Rect(qt.x, qt.y, f->r.max.x, pt.y), back, nil, qt);
        }
        return pt;
    }
```

Uses _frcklinewrap() 296c and nbytes() 319.

⟨*function* nbytes 319⟩≡ (364b)

```
    static int
    nbytes(char *s0, int nr)
    {
        char *s;
        Rune r;

        s = s0;
        while(--nr >= 0)
            s += chartorune(&r, s);
        return s-s0;
    }
```

Appendix E

Windowing System Applications

Earlier you met `hellorio` (Section 2.3), a minimal *window application*: a client that opens a single window and draws into it, oblivious to the rest of the screen. The programs in this chapter are *windowing-system applications* instead—each reaches past its own window into the windowing system as a whole. `lens` reads the entire screen through `/dev/screen` and magnifies a region of it; `statusbar` mounts `/mnt/wsys` itself to carve out a managed strip; and `winwatch` enumerates every open window through `/dev/wsys`, reading each one's `label`. Yet none of them needs any special privilege: they are ordinary (filesystem) clients of the very interfaces built in the previous chapters, which is exactly what makes such tools so easy to write under Plan 9.

E.1 lens

⟨*global* `menustr` 320a⟩≡ (352a)

```
static char *menustr[] = {
    "zoom",
    "unzoom",
    "grid",
    "redraw",
    "exit",
    nil
};
```

⟨*global menu (apps/lens.c)* 320b⟩≡ (352a)

```
static Menu menu = {
    menustr,
    nil,
    -1
};
```

Uses `menustr-109` 320a.

⟨*enum* `_anon_ (apps/lens.c)`*2* 320c⟩≡ (352a)

```
enum {
    Mzoom,
    Munzoom,
    Mgrid,
    Mredraw,
    Mexit
};
```

⟨*function main (apps/lens.c)* 320d⟩≡ (352a)

```
void
main(int argc, char *argv[])
{
    Event e;
```

```
char buf[5*12];
ulong chan;
int d;

USED(argc, argv);

if(initdraw(nil, nil, "lens") < 0){
    fprint(2, "lens: initdraw failed: %r\n");
    exits("initdraw");
}
einit(Emouse|Ekeyboard);

red = allocimage(display, Rect(0, 0, 1, 1), CMAP8, 1, DRed);
chequer = allocimage(display, Rect(0, 0, 2, 2), GREY1, 1, DBlack);

draw(chequer, Rect(0, 0, 1, 1), display->white, nil, ZP);
draw(chequer, Rect(1, 1, 2, 2), display->white, nil, ZP);
lastp = divpt(addpt(view->r.min, view->r.max), 2);

screenfd = open("/dev/screen", OREAD);
if(screenfd < 0){
    fprint(2, "lens: can't open /dev/screen: %r\n");
    exits("screen");
}
if(read(screenfd, buf, sizeof buf) != sizeof buf){
    fprint(2, "lens: can't read /dev/screen: %r\n");
    exits("screen");
}
chan = strtochan(buf);
d = chantodepth(chan);
if(d < 8){
    fprint(2, "lens: can't handle screen format %11.11s\n", buf);
    exits("screen");
}
bypp = d/8;
screenr.min.x = atoi(buf+1*12);
screenr.min.y = atoi(buf+2*12);
screenr.max.x = atoi(buf+3*12);
screenr.max.y = atoi(buf+4*12);
screenbuf = malloc(bypp*Dx(screenr)*Dy(screenr));
if(screenbuf == nil){
    fprint(2, "lens: buffer malloc failed: %r\n");
    exits("malloc");
}
eresized(0);

for(;;)
    switch(event(&e)){
    case Ekeyboard:
        switch(e.kbdc){
        case 'q':
        case 0x7f:
        case '\04':
        caseexit:
            exits(nil);
        case '=':
        case '+':
        casezoom:
            if(mag < Maxmag){
                mag++;
```

```
                makegrid();
                drawit();
            }
            break;
        case 'g':
        casegrid:
            showgrid = !showgrid;
            makegrid();
            drawit();
            break;
        case '-':
        case '_':
        caseunzoom:
            if(mag > 1){
                mag--;
                makegrid();
                drawit();
            }
            break;
        case '.':
        case ' ':
        caseredraw:
            drawit();
            break;
        case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9': case'0':
            mag = e.kbdc-'0';
            if(mag == 0)
                mag = 10;
            makegrid();
            drawit();
            break;
        }
        break;
    case Emouse:
        if(e.mouse.buttons & 1){
            lastp = e.mouse.xy;
            drawit();
        }
        if(e.mouse.buttons & 4)
            switch(emenuhit(3, &e.mouse, &menu)){
            case Mzoom:
                goto casezoom;
            case Munzoom:
                goto caseunzoom;
            case Mgrid:
                goto casegrid;
            case Mredraw:
                goto caseredraw;
            case Mexit:
                goto caseexit;
            }
        break;
    }
}
```

Uses `Maxmag-103` 350c, `Mexit-108` 320c, `Mgrid-106` 320c, `Mredraw-107` 320c, `Munzoom-105` 320c, `Mzoom-104` 320c, `bypp-121` 351k, `chequer-115` 351e, `drawit()` 323a, `lastp-111` 351a, `mag-117` 351g, `makegrid()` 351l, `menu-110` 320b, `red-112` 351b, `screenbuf-120` 351j, `screenfd-116` 351f, `screenr-119` 351i, and `showgrid-118` 351h.

⟨*function eresized (apps/lens.c)* 322⟩≡ (352a)

```
void
```

```
eresized(int new)
{
    if(new && getwindow(display, Refnone) < 0){
        fprint(2, "lens: can't reattach to window: %r\n");
        exits("attach");
    }
    freeimage(tmp);
    tmp = allocimage(display, Rect(0, 0, Dx(view->r)-Edge, Dy(view->r)-Edge+Maxmag), view->chan, 0, DNofill);
    if(tmp == nil){
        fprint(2, "lens: allocimage failed: %r\n");
        exits("allocimage");
    }
    drawit();
}
```

Uses `Edge-102` 350c, `Maxmag-103` 350c, `drawit()` 323a, and `tmp-113` 351c.

⟨*function* `drawit` 323a⟩≡ (352a)

```
void
drawit(void)
{
    Rectangle r;
    border(view, view->r, Edge, red, ZP);
    magnify();
    r = insetrect(view->r, Edge);
    draw(view, r, tmp, nil, tmp->r.min);
    flushimage(display, true);
}
```

Uses `Edge-102` 350c, `magnify()` 323b, `red-112` 351b, and `tmp-113` 351c.

⟨*function* `magnify` 323b⟩≡ (352a)

```
void
magnify(void)
{
    int x, y, xx, yy, dd, i;
    int dx, dy;
    int xoff, yoff;
    uchar out[8192];
    uchar sp[4];

    dx = (Dx(tmp->r)+mag-1)/mag;
    dy = (Dy(tmp->r)+mag-1)/mag;
    xoff = lastp.x-Dx(tmp->r)/(mag*2);
    yoff  = lastp.y-Dy(tmp->r)/(mag*2);

    yy = yoff;
    dd = dy;
    if(yy < 0){
        dd += dy;
        yy = 0;
    }
    if(yy+dd > Dy(screenr))
        dd = Dy(screenr)-yy;
    seek(screenfd, 5*12+bypp*yy*Dx(screenr), 0);
    if(readn(screenfd, screenbuf+bypp*yy*Dx(screenr), bypp*Dx(screenr)*dd) != bypp*Dx(screenr)*dd){
        fprint(2, "lens: can't read screen: %r\n");
        return;
    }

    for(y=0; y<dy; y++){
        yy = yoff+y;
```

```
        if(yy>=0 && yy<Dy(screenr))
            for(x=0; x<dx; x++){
                xx = xoff+x;
                if(xx>=0 && xx<Dx(screenr)) /* snarf pixel at xx, yy */
                    for(i=0; i<bypp; i++)
                        sp[i] = screenbuf[bypp*(yy*Dx(screenr)+xx)+i];
                else
                    sp[0] = sp[1] = sp[2] = sp[3] = 0;

                for(xx=0; xx<mag; xx++)
                    if(x*mag+xx < tmp->r.max.x)
                        for(i=0; i<bypp; i++)
                            out[(x*mag+xx)*bypp+i] = sp[i];
            }
        else
            memset(out, 0, bypp*Dx(tmp->r));
        for(yy=0; yy<mag && y*mag+yy<Dy(tmp->r); yy++){
            werrstr("no error");
            if(loadimage(tmp, Rect(0, y*mag+yy, Dx(tmp->r), y*mag+yy+1), out, bypp*Dx(tmp->r)) != bypp*Dx(tmp->
                exits("load");
            }
        }
    }
    if (showgrid && mag && grid)
        draw(tmp, tmp->r, grid, nil, mulpt(Pt(xoff, yoff), mag));
}
```

Uses bypp-121 351k, grid-114 351d, lastp-111 351a, mag-117 351g, screenbuf-120 351j, screenfd-116 351f, screenr-119 351i, showgrid-118 351h, and tmp-113 351c.

E.2 statusbar

⟨*function* drawbar 324⟩≡ (356a)

```
void
drawbar(void)
{
    int i, j;
    int p;
    char buf[400], bar[200];
    static char lastbar[200];

    if(n > d || n < 0 || d <= 0)
        return;

    i = (Dx(rbar)*n)/d;
    p = (n*100LL)/d;

    if(textmode){
        if(Dx(rbar) > 150){
            rbar.min.x = 0;
            rbar.max.x = 150;
            return;
        }
        bar[0] = '|';
        for(j=0; j<i; j++)
            bar[j+1] = '#';
        for(; j<Dx(rbar); j++)
            bar[j+1] = '-';
        bar[j++] = '|';
```

```
		bar[j++] = ' ';
		sprint(bar+j, "%3d%% ", p);
		for(i=0; bar[i]==lastbar[i] && bar[i]; i++)
			;
		memset(buf, '\b', strlen(lastbar)-i);
		strcpy(buf+strlen(lastbar)-i, bar+i);
		if(buf[0])
			write(1, buf, strlen(buf));
		strcpy(lastbar, bar);
		return;
	}

	if(lastp == p && last == i)
		return;

	if(lastp != p){
		sprint(buf, "%d%%", p);

		stringbg(view, addpt(view->r.min, Pt(Dx(rbar)-30, 4)), text, ZP, display->defaultfont, buf, light, ZP);
		lastp = p;
	}

	if(last != i){
		if(i > last)
			draw(view, Rect(rbar.min.x+last, rbar.min.y, rbar.min.x+i, rbar.max.y),
				dark, nil, ZP);
		else
			draw(view, Rect(rbar.min.x+i, rbar.min.y, rbar.min.x+last, rbar.max.y),
				light, nil, ZP);
		last = i;
	}
	flushimage(display, 1);
}
```

Uses `d-133` 356a, `dark-128` 353a, `last-134` 353f, `lastp-135` 353g, `light-127` 352f, `n-132` 356a, `rbar-130` 353d, `text-129` 353b, and `textmode-125` 352d.

⟨*function eresized (apps/statusbar.c)* 325⟩≡ (356a)

```
void
eresized(int new)
{
	Point p, q;
	Rectangle r;

	if(new && getwindow(display, Refnone) < 0)
		fprint(2,"can't reattach to window");

	r = view->r;
	draw(view, r, light, nil, ZP);
	p = string(view, addpt(r.min, Pt(4,4)), text, ZP,
		display->defaultfont, title);

	p.x = r.min.x+4;
	p.y += display->defaultfont->height+4;

	q = subpt(r.max, Pt(4,4));
	rbar = Rpt(p, q);

	ptext = Pt(r.max.x-4-stringwidth(display->defaultfont, "100%"), r.min.x+4);
	border(view, rbar, -2, dark, ZP);
	last = 0;
```

```
    lastp = -1;

    drawbar();
}
```
Uses `dark-128` 353a, `drawbar()` 324, `last-134` 353f, `lastp-135` 353g, `light-127` 352f, `ptext-131` 353e, `rbar-130` 353d, `text-129` 353b, and `title-126` 352e.

⟨*function* `bar` 326a⟩≡ (356a)
```
void
bar(Biobuf *b)
{
    char *p, *f[2];
    Event e;
    int k, die, parent, child;

    parent = getpid();

    die = 0;
    if(textmode)
        child = -1;
    else
    switch(child = rfork(RFMEM|RFPROC)) {
    case 0:
        sleep(1000);
        while(!die && (k = eread(Ekeyboard|Emouse, &e))) {
            if(nokill==0 && k == Ekeyboard && (e.kbdc == 0x7F || e.kbdc == 0x03)) { /* del, ctl-c */
                die = 1;
                postnote(PNPROC, parent, "interrupt");
                _exits("interrupt");
            }
        }
        _exits(0);
    }

    while(!die && (p = Brdline(b, '\n'))) {
        p[Blinelen(b)-1] = '\0';
        if(tokenize(p, f, 2) != 2)
            continue;
        n = strtoll(f[0], 0, 0);
        d = strtoll(f[1], 0, 0);
        drawbar();
    }
    postnote(PNCTL, child, "kill");
}
```
Uses `PNCTL-123` 352b, `d-133` 356a, `drawbar()` 324, `n-132` 356a, `nokill-124` 352c, `postnote()` 355b, and `textmode-125` 352d.

⟨*function usage (apps/statusbar.c)* 326b⟩≡ (356a)
```
static void
usage(void)
{
    fprint(2, "usage: aux/statusbar [-kt] [-w minx,miny,maxx,maxy] 'title'\n");
    exits("usage");
}
```

⟨*function main (apps/statusbar.c)* 326c⟩≡ (356a)
```
void
main(int argc, char **argv)
{
    Biobuf b;
    char *p, *q;
```

```
    int lfd;

    p = "0,0,200,60";

    ARGBEGIN{
    case 'w':
        p = ARGF();
        break;
    case 't':
        textmode = 1;
        break;
    case 'k':
        nokill = 1;
        break;
    default:
        usage();
    }ARGEND;

    if(argc != 1)
        usage();

    title = argv[0];

    lfd = dup(0, -1);

    while(q = strchr(p, ','))
        *q = ' ';
    Binit(&b, lfd, OREAD);
    if(textmode || newwin(p) < 0){
        textmode = 1;
        rbar = Rect(0, 0, 60, 1);
    }else{
        if(initdraw(0, 0, "bar") < 0)
            exits("initdraw");
        initcolor();
        einit(Emouse|Ekeyboard);
        eresized(0);
    }
    bar(&b);

    exits(0);
}
```

E.3 winwatch

⟨*function* drawnowin 327⟩≡ (360)

```
static void
drawnowin(int i)
{
    Rectangle r;

    r = Rect(0,0,(Dx(view->r)-2*MARGIN+PAD)/cols-PAD, font->height);
    r = rectaddpt(rectaddpt(r, Pt(MARGIN+(PAD+Dx(r))*(i/rows),
                MARGIN+(PAD+Dy(r))*(i%rows))), view->r.min);
    draw(view, insetrect(r, -1), lightblue, nil, ZP);
}
```

Uses MARGIN-146 357g, PAD-145 357g, cols-143 360, lightblue-144 357f, and rows-142 360.

⟨*function* drawwin 328a⟩≡ (360)

```
static void
drawwin(int i)
{
    draw(view, win[i].r, lightblue, nil, ZP);
    _string(view, addpt(win[i].r.min, Pt(2,0)), display->black, ZP,
        font, win[i].label, nil, strlen(win[i].label),
        win[i].r, nil, ZP, SoverD);
    border(view, win[i].r, 1, display->black, ZP);
    win[i].dirty = 0;
}
```

Uses `lightblue-144` 357f and `win-138` 357b.

⟨*function redraw (apps/winwatch.c)* 328b⟩≡ (360)

```
static void
redraw(Image *view, int all)
{
    int i;

    all |= geometry();
    if(all)
        draw(view, view->r, lightblue, nil, ZP);
    for(i=0; i<nwin; i++)
        if(all || win[i].dirty)
            drawwin(i);
    if(!all)
        for(; i<onwin; i++)
            drawnowin(i);

    onwin = nwin;
}
```

Uses `drawnowin()` 327, `drawwin()` 328a, `geometry()` 359a, `lightblue-144` 357f, `nwin-139` 357c, `onwin-141` 357e, and `win-138` 357b.

⟨*function eresized (apps/winwatch.c)* 328c⟩≡ (360)

```
void
eresized(int new)
{
    if(new && getwindow(display, Refmesg) < 0)
        fprint(2,"can't reattach to window");
    geometry();
    redraw(view, 1);
}
```

Uses `geometry()` 359a.

⟨*function usage (apps/winwatch.c)* 328d⟩≡ (360)

```
static void
usage(void)
{
    fprint(2, "usage: winwatch [-e exclude] [-f font]\n");
    exits("usage");
}
```

⟨*function main (apps/winwatch.c)* 328e⟩≡ (360)

```
void
main(int argc, char **argv)
{
    char *fontname;
    int Etimer;
```

```
    Event e;

    fontname = "/lib/font/bit/lucidasans/unicode.8.font";
    ARGBEGIN{
    case 'f':
        fontname = EARGF(usage());
        break;
    case 'e':
        exclude = regcomp(EARGF(usage()));
        if(exclude == nil)
            sysfatal("Bad regexp");
        break;
    default:
        usage();
    }ARGEND

    if(argc)
        usage();

    initdraw(0, 0, "winwatch");
    lightblue = allocimagemix(display, DPalebluegreen, DWhite);
    if(lightblue == nil)
        sysfatal("allocimagemix: %r");
    if((font = openfont(display, fontname)) == nil)
        sysfatal("font '%s' not found", fontname);

    refreshwin();
    redraw(view, 1);
    einit(Emouse|Ekeyboard);
    Etimer = etimer(0, 2500);

    for(;;){
        switch(eread(Emouse|Ekeyboard|Etimer, &e)){
        case Ekeyboard:
            if(e.kbdc==0x7F || e.kbdc=='q')
                exits(0);
            break;
        case Emouse:
            if(e.mouse.buttons)
                click(e.mouse);
            /* fall through  */
        default: /* Etimer */
            refreshwin();
            redraw(view, 0);
            break;
        }
    }
}
```

E.4 `window`

The `window` script creates a new window and runs a command in it from the shell: it writes a "new" command—with the working directory and the command line—to the window-control file, so `window acme` opens a fresh window running `acme`.

⟨*scripts/window* 329⟩≡

```
#!/bin/rc
# window [many options] cmd [arg...] - create new window and run cmd in it
```

```
rfork e
fn checkwsys{
	if(~ $wsys ''){
		echo  'window: $wsys not defined'
		exit bad
	}
}

# original version used mount to do the work
fn oldway{
	switch($#*){
	case 0 1
		echo usage: window '''minx miny maxx maxy''' cmd args ...
		exit usage
	}

	checkwsys

	dir = /mnt/wsys
	srv = $wsys

	rfork ne
	{
		if(x=`{cat /dev/ppid}; mount $srv $dir N`{{echo $x $1 }| sed 's/^ //g;s/ +/,/g'}){
			shift
			bind -b $dir /dev
			echo -n `{basename $1} > /dev/label >[2] /dev/null
			exec $* < /dev/cons > /dev/cons >[2] /dev/cons
		}
	}&
}

# if argument is of form '100 100 200 200' or '100,100,200,200' use old way
if(~ $1 *[0-9][' ,'][0-9]*){
	oldway $*
	exit
}

# geometry parameters are:
#	-r 0 0 100 100
#	-dx n
#	-dy n
#	-minx n
#	-miny n
#	-maxx n
#	-maxy n
# where n can be a number, to set the value, or +number or -number to change it

# find wctl file
fn getwctl{
	if(~ $wctl ''){
		if(test -f /dev/wctl) echo /dev/wctl
		if not if(test -f /mnt/term/dev/wctl) echo /mnt/term/dev/wctl
		if not if(~ $service cpu) echo /mnt/term/srv/riowctl.*.*
		if not {
			echo window: '$wctl' not defined >[1=2]
			exit usage
		}
	}
	if not echo $wctl
```

```
}

# use mount to make local window
if(~ $1 -m){
	shift

	checkwsys

	dir = /mnt/wsys
	srv = $wsys
	rfork ne
	{
		unmount /mnt/acme /dev >[2]/dev/null
		if(mount $srv $dir 'new -pid '^`{cat /dev/ppid}^' '$"*){
			bind -b $dir /dev
			# toss geometry parameters to find command
			while(~ $1 -*)
				switch($1){
				case -dx -dy -minx -miny -maxx -maxy
					shift 2
				case -r
					shift 5
				case -scroll
					shift
				case -noscroll
					shift
				case -hide
					shift
				}
			if(~ $#* 0) cmd = rc
			if not cmd = $*
			echo -n `{basename $cmd(1)} > /dev/label >[2] /dev/null
			exec $cmd < /dev/cons > /dev/cons >[2] /dev/cons
		}
	}&
}

if not echo new -cd `{pwd} $* >> `{getwctl}
```

E.5 wloc

The `wloc` script dumps the current window layout: for every window it reads the rectangle from that window's `window` file and its `label`, and prints a `window -r ...` command. Saving its output and replaying it later recreates the same windows in the same places.

⟨*scripts/wloc* 331⟩≡

```
#!/bin/rc

rfork e
ifs='
'

for(i in `{ls /dev/wsys}) {
	echo window -r `{syscall -o read 0 buf 59 < $i/window >[2] /dev/null |
	sed 's/...........//; s/^ *//; s/  */ /g'} `{cat $i/label}
}
```

E.6 label

The `label` script sets the current window's title by writing its arguments to `/dev/label` (falling back to `/mnt/term/dev/label` when running on a remote terminal through `cpu`).

⟨*scripts/label* 332⟩≡

```
#!/bin/rc
# label word ... - write words into our label, if any
if (test -w /dev/label)
        echo -n $* > /dev/label
if not if (test -w   /mnt/term/dev/label)
        echo -n $* > /mnt/term/dev/label
```

Appendix F

Extra Code

This appendix collects the code chunks the preceding chapters referred to but did not show inline: header boilerplate, forward declarations, and the small uninteresting helpers whose definitions would only have interrupted the narrative. Nothing here is new—it is the remainder of `rio`'s source, included so that the literate program remains, as it must, the whole program and not just its interesting parts.

F.1 rio/

F.1.1 rio/dat.h

```
⟨enum _anon_ (rio/dat.h)3 333a⟩≡                                    (333b)
  enum
  {
      ⟨constant Selborder 67⟩
      ⟨constant Unselborder 105c⟩
      ⟨constants Scrollxxx 174a⟩
      ⟨constant BIG 181h⟩
  };
```

```
⟨rio/dat.h 333b⟩≡

  // forward decls
  typedef struct Window Window;
  typedef struct Wctlmesg Wctlmesg;
  typedef struct Filsys Filsys;
  typedef struct Fid Fid;
  typedef struct Xfid Xfid;
  typedef struct Consreadmesg Consreadmesg;
  typedef struct Conswritemesg Conswritemesg;
  typedef struct Stringpair Stringpair;
  typedef struct Dirtab Dirtab;
  typedef struct Mouseinfo Mouseinfo;
  typedef struct Mousereadmesg Mousereadmesg;
  typedef struct Mousestate Mousestate;
  typedef struct Ref Ref;
  typedef struct Timer Timer;

  //----------------------------------------------------------------------------
  // Data structures and constants
  //----------------------------------------------------------------------------

  ⟨enum qid 122d⟩
```

```
⟨function QID 122a⟩
⟨function WIN 122b⟩
⟨function FILE 122c⟩

⟨enum _anon_ (rio/dat.h)2 203a⟩

⟨constant STACK 61a⟩

⟨enum _anon_ (rio/dat.h)3 333a⟩

⟨constant DEBUG 278a⟩

⟨enum wctlmesgkind 91a⟩
⟨struct Wctlmesg 91b⟩

⟨struct Conswritemesg 149c⟩
⟨struct Consreadmesg 147d⟩

⟨struct Mousereadmesg 142c⟩

⟨struct Stringpair 148a⟩

⟨struct Mousestate 162b⟩
⟨struct Mouseinfo 161b⟩

⟨struct Window 49⟩

⟨struct Dirtab 123a⟩

⟨struct Fid 53e⟩
⟨struct Xfid 55b⟩

⟨constant Nhash 53c⟩

⟨struct Filsys 53a⟩

⟨struct Timer 223a⟩

//----------------------------------------------------------------------------
// Globals
//----------------------------------------------------------------------------

// see also draw.h globals: display, font, view

// globals.c

extern Screen *desktop;
extern Image *background;
extern Image *red;

extern Window **windows;
extern int  nwindow;
extern Window *input;

extern Window *hidden[100];
extern int  nhidden;

extern Filsys *filsys;

extern Keyboardctl *keyboardctl;
```

```
extern Mousectl *mousectl;
extern Mouse *mouse;

extern Channel *exitchan; // was static in rio.c
extern Channel* winclosechan;
extern Channel* deletechan;

extern int  snarfversion; /* updated each time it is written */

extern int  sweeping;
extern bool  menuing;
extern int  scrolling;

extern char  *startdir;

// misc
extern Window *wkeyboard; /* window of simulated keyboard */
extern QLock all;   /* BUG */
extern fdt  wctlfd;
extern int  maxtab;


// 9p.c
extern int  messagesize; /* negotiated in 9P version setup */


// thread_mouse.c
extern Rectangle viewr; // was static in rio.c

// data.c
extern Cursor boxcursor;
extern Cursor crosscursor;
extern Cursor sightcursor;
extern Cursor whitearrow;
extern Cursor query;
extern Cursor *corners[9];

// cursor.c
extern Cursor *lastcursor; // was static in wind.c

// snarf.c
extern fdt  snarffd;
extern Rune* snarf;
extern int  nsnarf;

// error.c
extern bool  errorshouldabort;
extern char Eperm[];
```

Uses `Consreadmesg` 147d, `Conswritemesg` 149c, `Dirtab` 123a, `Fid` 53e, `Filsys` 53a, `Mouseinfo` 161b, `Mousereadmesg` 142c, `Mousestate` 162b, `Stringpair` 148a, `Timer` 223a, `Wctlmesg` 91b, `Window` 49, and `Xfid` 55b.

F.1.2 rio/fns.h

⟨rio/fns.h 335⟩≡

```
// thread_keyboard.c (for rio.c)
void keyboardthread(void*);
// thread_mouse.c    (for rio.c)
void mousethread(void*);
```

```
// threads_misc.c    (for rio.c)
void  winclosethread(void*);
void  deletethread(void*);
// threads_worker.c (for rio.c)
Channel* xfidinit(void);
void xfidflush(Xfid*);
// proc_fileserver (for rio.c and process_winshell.c)
Filsys* filsysinit(Channel*);
int  filsysmount(Filsys*, int);
// threads_window.c
void    winctl(void*);
// processes_winshell
void winshell(void*);

// data.c (for rio.c)
void iconinit(void);
// timer.c (for rio.c and scrl.c)
void timerinit(void);
void timerstop(Timer*);
void timercancel(Timer*);
Timer* timerstart(int);

// fsys.c (for rio.c and xfid.c and process_winshell.c)

// cursor.c
void riosetcursor(Cursor*, int);

// wm.c (for thread_mouse.c)
void    cornercursor(Window *w, Point p, bool force);
Image *bandsize(Window*);
Image* drag(Window*, Rectangle*);
void    button3menu(void);
Window *new(Image*, int, int, int, char*, char*, char**); // for wkeyboard
int whide(Window*); // for wctl
int wunhide(int);   // for wctl

// wind.c
void    wsendctlmesg(Window*, int, Rectangle, Image*);
Window* wmk(Image*, Mousectl*, Channel*, Channel*, int);
int  wclose(Window*);
void wclosewin(Window*);
Window* wpointto(Point);
void wcurrent(Window*);
Window* wlookid(int);
void wresize(Window*, Image*, int);
void wrefresh(Window*, Rectangle);
void wrepaint(Window*);
int  winborder(Window*, Point);
Window* wtop(Point);
void wtopme(Window*);
void wbottomme(Window*);
void wsetcursor(Window*, bool);
void wmovemouse(Window*, Point);
void wfill(Window*);
void wsetname(Window*);
void wsetpid(Window*, int, int);


// TODO
int     goodrect(Rectangle);
```

```
// graphical_window.c
void waddraw(Window*, Rune*, int);

// terminal.c
void    button2menu(Window *w);
void wkeyctl(Window*, Rune);
void wmousectl(Window*);
void wdelete(Window*, uint, uint);
uint winsert(Window*, Rune*, int, uint);
void wshow(Window*, uint);
void wsetselect(Window*, uint, uint);
void wsetorigin(Window*, uint, int);
uint wbacknl(Window*, uint, uint);
char* wcontents(Window*, int*);

// scroll.c
void freescrtemps(void);
void wscrdraw(Window*);
void  wscrsleep(Window*, uint);
void wscroll(Window*, int);

// snarf.c
void putsnarf(void);
void getsnarf(void);

// 9p.c (for fsys.c and xfid.c)
Xfid* filsysrespond(Filsys*, Xfid*, Fcall*, char*);
void filsyscancel(Xfid*);

// xfid.c  (for fsys.c)
void xfidattach(Xfid*);
void xfidopen(Xfid*);
void xfidclose(Xfid*);
void xfidread(Xfid*);
void xfidwrite(Xfid*);

// wctl.c (for fsys.c and xfid.c)
void wctlproc(void*);
void wctlthread(void*);
int     parsewctl(char**, Rectangle, Rectangle*, int*, int*, int*, int*, char**, char*, char*);
int     writewctl(Xfid*, char*);

// util.c
int     min(int, int);
int     max(int, int);
Rune* strrune(Rune*, Rune);
int     isalnum(Rune);
void cvttorunes(char*, int, Rune*, int*, int*, int*);
char*  runetobyte(Rune*, int, int*);/* was (byte*,int) runetobyte(Rune*,int);*/
void* erealloc(void*, uint);
void*  emalloc(uint);
char*  estrdup(char*);

⟨function runemalloc 284d⟩
⟨function runerealloc 284e⟩
⟨function runemove 285a⟩

// error.c
void    derror(Display*, char *); // for main.c
```

```
void error(char*);
```

F.1.3 rio/globals.c

```
⟨rio/globals.c 338a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global mousectl 48a⟩
  ⟨global mouse 48c⟩
  ⟨global keyboardctl 48b⟩

  ⟨global desktop 48d⟩
  ⟨global background 48e⟩
  ⟨global red 48f⟩

  ⟨global windows 51a⟩
  ⟨global wkeyboard 226g⟩
  ⟨global nwindow 51b⟩

  ⟨global exitchan 60e⟩

  ⟨global winclosechan 108c⟩
  ⟨global deletechan 107d⟩

  ⟨global input 51e⟩
  ⟨global all 128b⟩
  ⟨global filsys 53b⟩
  ⟨global hidden 118c⟩
  ⟨global nhidden 118d⟩
  ⟨global scrolling 225c⟩

  ⟨global startdir 96h⟩
  ⟨global sweeping 89d⟩
  ⟨global wctlfd 258a⟩
  ⟨global menuing 102a⟩
  ⟨global snarfversion 233j⟩

  ⟨global maxtab 195c⟩
```

F.1.4 rio/rio.c

```
⟨rio/rio.c 338b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>
  #include <plumb.h>

  /*
   *  WASHINGTON (AP) - The Food and Drug Administration warned
   * consumers Wednesday not to use ‘‘Rio’’ hair relaxer products
   * because they may cause severe hair loss or turn hair green....
   *    The FDA urged consumers who have experienced problems with Rio
   * to notify their local FDA office, local health department or the
   * company at 1-800-543-3002.
   */
```

```
⟨global fontname 229a⟩

⟨global kbdargv 226h⟩

⟨function usage 225b⟩

⟨function initcmd 226d⟩

⟨global oknotes 219a⟩

⟨function killprocs 220a⟩
⟨function shutdown 219c⟩

⟨function threadmain 58⟩
```

F.1.5 rio/thread_mouse.c

```
⟨rio/thread_mouse.c 339a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>

  ⟨enum Mxxx 65a⟩

  ⟨function keyboardhide 228d⟩

  ⟨global viewr 47⟩

  ⟨function resized 221⟩

  ⟨function mousethread 66a⟩
```

F.1.6 rio/thread_keyboard.c

```
⟨rio/thread_keyboard.c 339b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨function keyboardthread 64⟩
```

F.1.7 rio/threads_window.c

```
⟨rio/threads_window.c 339c⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>

  ⟨enum Wxxx 71a⟩

  ⟨function deletetimeoutproc 107c⟩

  ⟨function wctlmesg 92a⟩

  ⟨function winctl 71b⟩
```

F.1.8 rio/threads_worker.c

```
⟨rio/threads_worker.c 340a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global xfidfree 78b⟩
  ⟨global xfid 78a⟩
  ⟨global cxfidalloc 62b⟩
  ⟨global cxfidfree 62c⟩

  ⟨enum Xxxx 77f⟩

  ⟨function xfidctl 79c⟩

  ⟨function xfidflush 267c⟩

  ⟨function xfidallocthread 78d⟩

  ⟨function xfidinit 63⟩
```

F.1.9 rio/threads_misc.c

```
⟨rio/threads_misc.c 340b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>

  ⟨function winclosethread 109a⟩
  ⟨function deletethread 108b⟩
```

F.1.10 rio/wm.c

```
⟨rio/wm.c 340c⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>

  // Most of the functions in this file are executed from
  // a threadmouse() context (via button3menu()). They send
  // a Wctlmesg to the window thread to get the actual modifications
  // done on the global windows state.

  //----------------------------------------------------------------------------
  // Menu
  //----------------------------------------------------------------------------

  ⟨enum _anon_ (rio/rio.c) 89c⟩

  ⟨global menu3str 89b⟩
  ⟨global menu3 89a⟩

  //----------------------------------------------------------------------------
  // Helpers
  //----------------------------------------------------------------------------
```

```
⟨function goodrect 261b⟩
⟨function portion 86c⟩
⟨function whichcorner 86b⟩
⟨function cornercursor 85c⟩

//----------------------------------------------------------------------------
// Mouse actions
//----------------------------------------------------------------------------

⟨function pointto 110a⟩

⟨function onscreen 102b⟩

⟨function sweep 101d⟩


⟨function drawedge 113a⟩
⟨function drawborder 112b⟩

⟨function drag 111c⟩

⟨function cornerpt 117⟩
⟨function whichrect 116d⟩

⟨function bandsize 115⟩

//----------------------------------------------------------------------------
// Window management
//----------------------------------------------------------------------------

⟨function delete 106a⟩

⟨function resize 113c⟩

⟨function move 111b⟩

⟨function whide 118e⟩
⟨function wunhide 120b⟩

⟨function hide 118b⟩
⟨function unhide 119e⟩

⟨global rcargv 97a⟩

⟨function new 92c⟩

//----------------------------------------------------------------------------
// Entry point
//----------------------------------------------------------------------------
⟨function button3menu 88b⟩
```

F.1.11 rio/data.c

⟨rio/data.c 341⟩≡

```
#include <u.h>
#include <libc.h>
⟨rio includes 42⟩
```

```
⟨global crosscursor (rio/data.c) 81⟩
⟨global boxcursor (rio/data.c) 82a⟩
⟨global sightcursor (rio/data.c) 82b⟩
⟨global whitearrow (rio/data.c) 82c⟩
⟨global query (rio/data.c) 82d⟩

⟨global tl 83b⟩
⟨global t 83c⟩
⟨global tr 83d⟩
⟨global r 84a⟩
⟨global br 84b⟩
⟨global b 84c⟩
⟨global bl 84d⟩
⟨global l 84e⟩

⟨global corners (rio/data.c) 83a⟩

⟨function iconinit 59d⟩
```

F.1.12 rio/cursor.c

```
⟨rio/cursor.c 342a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global lastcursor 85a⟩

  ⟨function riosetcursor 85b⟩
```

F.1.13 rio/wind.c

```
⟨rio/wind.c 342b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <window.h>

  ⟨global topped 51c⟩
  ⟨global id 50b⟩

  ⟨global cols 289g⟩
  ⟨global darkgrey 176a⟩

  ⟨global titlecol 96b⟩
  ⟨global lighttitlecol 96c⟩
  ⟨global holdcol 271b⟩
  ⟨global lightholdcol 271d⟩
  ⟨global paleholdcol 271c⟩

  ⟨function wsendctlmesg 91c⟩

  ⟨function wborder 96a⟩

  ⟨function wsetcols 175c⟩

  ⟨function wmk 94c⟩
```

```
⟨function wsetname 101a⟩

⟨function wresize 114c⟩

⟨function wrefresh 169a⟩

⟨function wclose 109b⟩

⟨function wrepaint 105b⟩

⟨function winborder 86a⟩

⟨function wmovemouse 116c⟩

⟨function wpointto 69⟩

⟨function wcurrent 105a⟩

⟨function wsetcursor 87⟩

⟨function wtop 104b⟩

⟨function wtopme 217d⟩
⟨function wbottomme 217e⟩

⟨function wlookid 128a⟩

⟨function wclosewin 106d⟩

⟨function wsetpid 98c⟩
```

F.1.14 rio/processes_winshell.c

```
⟨rio/processes_winshell.c 343a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨function winshell 97d⟩
```

F.1.15 rio/terminal.c

```
⟨rio/terminal.c 343b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <plumb.h>
  #include <complete.h>

  ⟨enum _anon_ (rio/wind.c) 177d⟩

  ⟨enum _anon_ (rio/rio.c)2 199c⟩

  ⟨global menu2str 199b⟩
  ⟨global menu2 199a⟩

  ⟨global clickwin 252b⟩
  ⟨global clickmsec 252c⟩
```

```
⟨global selectwin 252d⟩
⟨global selectq 252e⟩

//----------------------------------------------------------------------------
// Completion
//----------------------------------------------------------------------------

⟨function windfilewidth 248⟩

⟨function showcandidates 247b⟩

⟨function namecomplete 247a⟩

//----------------------------------------------------------------------------
// Editor
//----------------------------------------------------------------------------

⟨function wbswidth 191b⟩

⟨function wfill 184c⟩


⟨function wdelete 193a⟩

⟨function wbacknl 182d⟩

⟨function wsetorigin 183⟩

⟨function wshow 179b⟩

⟨function wsetselect 185⟩

⟨function winsert 177a⟩

⟨function wcontents 206b⟩

//----------------------------------------------------------------------------
// Cut/copy/paste
//----------------------------------------------------------------------------
⟨function wsnarf 231a⟩
⟨function wcut 231b⟩
⟨function wpaste 231c⟩

//----------------------------------------------------------------------------
// Scrolling
//----------------------------------------------------------------------------

⟨function wframescroll 204b⟩
⟨function framescroll 204a⟩

//----------------------------------------------------------------------------
// Selection
//----------------------------------------------------------------------------

⟨function wclickmatch 255b⟩

⟨global left1 254a⟩
⟨global right1 254b⟩
⟨global left2 254c⟩
⟨global left3 254d⟩
```

```
⟨global left 254e⟩
⟨global right 254f⟩

⟨function wdoubleclick 255a⟩

⟨function wselect 253⟩

//----------------------------------------------------------------------------
// Clicking
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------
// Plumb
//----------------------------------------------------------------------------
⟨function wplumb 235a⟩

//----------------------------------------------------------------------------
// Middle click
//----------------------------------------------------------------------------
⟨function button2menu 198b⟩

//----------------------------------------------------------------------------
// Mouse dispatch
//----------------------------------------------------------------------------
⟨function wmousectl 199e⟩

//----------------------------------------------------------------------------
// Key dispatch
//----------------------------------------------------------------------------
⟨function interruptproc 194c⟩

⟨function wkeyctl 73b⟩
```

F.1.16 rio/snarf.c

```
⟨rio/snarf.c 345a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global snarffd 233f⟩

  ⟨global snarf 233i⟩
  ⟨global nsnarf 233h⟩

  ⟨function putsnarf 233k⟩
  ⟨function getsnarf 234a⟩
```

F.1.17 rio/graphical_window.c

```
⟨rio/graphical_window.c 345b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨function waddraw 164c⟩
```

F.1.18 rio/9p.c

```
⟨rio/9p.c 346a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global messagesize 76a⟩

  ⟨function filsysrespond 124⟩

  ⟨function filsyscancel 267d⟩
```

F.1.19 rio/proc_fileserver.c

```
⟨rio/proc_fileserver.c 346b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  extern Xfid*  (*fcall[])(Filsys*, Xfid*, Fid*);
  extern char Ebadfcall[];
  extern bool firstmessage;
  extern Fid*  newfid(Filsys*, int);
  extern fdt clockfd;

  ⟨global srvpipe (rio/fsys.c) 256a⟩
  ⟨global srvwctl (rio/fsys.c) 258b⟩

  ⟨function filsysproc 75⟩

  ⟨function cexecpipe 62a⟩

  ⟨function post 257a⟩

  ⟨function filsysinit 61c⟩

  ⟨function filsysmount 99c⟩
```

F.1.20 rio/fsys.c

```
⟨rio/fsys.c 346c⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global Eexist 280b⟩
  ⟨global Enotdir 280c⟩
  ⟨global Ebadfcall 280d⟩
  ⟨global Eoffset 280e⟩

  ⟨global dirtab 123b⟩

  static uint  getclock(void);
  Fid*  newfid(Filsys*, int);
  static int  dostat(Filsys*, int, Dirtab*, uchar*, int, uint);

  ⟨global clockfd 138c⟩
```

```
⟨global firstmessage 77b⟩

static Xfid* filsysflush(Filsys*, Xfid*, Fid*);
static Xfid* filsysversion(Filsys*, Xfid*, Fid*);
static Xfid* filsysauth(Filsys*, Xfid*, Fid*);
static Xfid* filsysnop(Filsys*, Xfid*, Fid*);
static Xfid* filsysattach(Filsys*, Xfid*, Fid*);
static Xfid* filsyswalk(Filsys*, Xfid*, Fid*);
static Xfid* filsysopen(Filsys*, Xfid*, Fid*);
static Xfid* filsyscreate(Filsys*, Xfid*, Fid*);
static Xfid* filsysread(Filsys*, Xfid*, Fid*);
static Xfid* filsyswrite(Filsys*, Xfid*, Fid*);
static Xfid* filsysclunk(Filsys*, Xfid*, Fid*);
static Xfid* filsysremove(Filsys*, Xfid*, Fid*);
static Xfid* filsysstat(Filsys*, Xfid*, Fid*);
static Xfid* filsyswstat(Filsys*, Xfid*, Fid*);

⟨global fcall 56b⟩

⟨function filsysversion 77a⟩

⟨function filsysauth 270b⟩

⟨function filsysflush 267b⟩

⟨function filsysattach 126⟩

⟨function numeric 210e⟩

⟨function filsyswalk 129⟩

⟨function filsysopen 132a⟩

⟨function filsyscreate 139c⟩

⟨function idcmp 211b⟩

⟨function filsysread 135a⟩

⟨function filsyswrite 137a⟩

⟨function filsysclunk 133c⟩

⟨function filsysremove 139d⟩

⟨function filsysstat 138a⟩

⟨function filsyswstat 139e⟩

⟨function newfid 54b⟩

⟨function getclock 139a⟩

⟨function dostat 138b⟩
```

F.1.21 rio/xfid.c

```
⟨rio/xfid.c 347⟩≡
  #include <u.h>
```

```
#include <libc.h>
⟨rio includes 42⟩
#include <marshal.h>
#include <window.h>
#include <plumb.h>

⟨constant MAXSNARF 233a⟩

⟨global Einuse 281a⟩
⟨global Edeleted 281b⟩
⟨global Ebadreq 281c⟩
⟨global Etooshort 281d⟩
⟨global Ebadtile 281e⟩
⟨global Eshort 281f⟩
⟨global Elong 281g⟩
⟨global Eunkid 281h⟩
⟨global Ebadrect 281i⟩
⟨global Ewindow 281j⟩
⟨global Enowindow 281k⟩
⟨global Ebadmouse 281l⟩
⟨global Ebadwrect 281m⟩
⟨global Ebadoffset 281n⟩

⟨global tsnarf 233d⟩
⟨global ntsnarf 233e⟩

⟨function xfidattach 127c⟩

⟨function xfidopen 133a⟩
⟨function xfidclose 134⟩

⟨function keyboardsend 228c⟩

⟨enum _anon_ (rio/xfid.c)2 149d⟩

⟨function xfidwrite 137b⟩

⟨function readwindow 170c⟩

⟨enum _anon_ (rio/xfid.c)3 148b⟩
⟨enum _anon_ (rio/xfid.c)4 143a⟩
⟨enum _anon_ (rio/xfid.c)5 214b⟩

⟨function xfidread 136b⟩
```

F.1.22 rio/scrl.c

```
⟨rio/scrl.c 348⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global scrtmp 181g⟩

  ⟨function scrtemps 182a⟩

  ⟨function freescrtemps 203b⟩

  ⟨function scrpos 180c⟩
```

```
⟨function wscrdraw 180b⟩

⟨function wscrsleep 203c⟩

⟨function wscroll 202⟩
```

F.1.23 rio/time.c

```
⟨rio/time.c 349a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨global ctimer 222c⟩
  ⟨global timer 222f⟩

  ⟨function msec 225a⟩

  ⟨function timerstop 223c⟩
  ⟨function timercancel 223d⟩

  ⟨function timerproc 224⟩

  ⟨function timerinit 222e⟩
  ⟨function timerstart 223b⟩
```

F.1.24 rio/wctl.c

```
⟨rio/wctl.c 349b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩
  #include <plumb.h>
  #include <window.h>

  #include <ctype.h>

  ⟨global Ebadwr 280f⟩
  ⟨global Ewalloc 280g⟩

  ⟨enum _anon_ (rio/wctl.c) 260a⟩

  ⟨global cmds 260b⟩

  ⟨enum _anon_ (rio/wctl.c)2 260c⟩

  ⟨global params 261a⟩

  ⟨function word 261c⟩

  ⟨function set 262a⟩

  ⟨function newrect 262b⟩

  ⟨function shift 262c⟩

  ⟨function rectonscreen 263a⟩
```

```
⟨function riostrtol 263b⟩


⟨function parsewctl 263c⟩

⟨function wctlnew 265⟩

⟨function writewctl 215b⟩

⟨function wctlthread 259b⟩

⟨function wctlproc 259a⟩
```

F.1.25 rio/error.c

```
⟨rio/error.c 350a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>

  ⟨global errorshouldabort 282a⟩

  // could be in 9p.c too
  ⟨global Eperm 280a⟩

  ⟨function error 282c⟩
  ⟨function derror 282d⟩
```

F.1.26 rio/util.c

```
⟨rio/util.c 350b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨rio includes 42⟩

  ⟨function cvttorunes 285b⟩

  ⟨function erealloc 283a⟩
  ⟨function emalloc 283b⟩
  ⟨function estrdup 283c⟩

  ⟨function isalnum 284c⟩

  ⟨function strrune 285c⟩

  ⟨function min 284a⟩
  ⟨function max (rio/util.c) 284b⟩

  ⟨function runetobyte 285d⟩
```

F.2 apps/

F.2.1 apps/lens.c

```
⟨enum _anon_ (apps/lens.c) 350c⟩≡                                      (352a)
```

```
enum {
    Edge = 5,
    Maxmag = 16
};
```

⟨*global* `lastp` 351a⟩≡ (352a)

```
static Point lastp;
```

⟨*global red (apps/lens.c)* 351b⟩≡ (352a)

```
static Image *red;
```

⟨*global tmp (apps/lens.c)* 351c⟩≡ (352a)

```
static Image *tmp;
```

⟨*global* `grid` 351d⟩≡ (352a)

```
static Image *grid;
```

⟨*global* `chequer` 351e⟩≡ (352a)

```
static Image *chequer;
```

⟨*global* `screenfd` 351f⟩≡ (352a)

```
static int screenfd;
```

⟨*global* `mag` 351g⟩≡ (352a)

```
static int mag = 4;
```

Uses `mag-117` 351g.

⟨*global* `showgrid` 351h⟩≡ (352a)

```
static int showgrid = 0;
```

Uses `showgrid-118` 351h.

⟨*global* `screenr` 351i⟩≡ (352a)

```
static Rectangle screenr;
```

⟨*global* `screenbuf` 351j⟩≡ (352a)

```
static uchar *screenbuf;
```

⟨*global* `bypp` 351k⟩≡ (352a)

```
static int bypp;
```

⟨*function* `makegrid` 351l⟩≡ (352a)

```
void
makegrid(void)
{
    int m;
    if (grid != nil) {
        freeimage(grid);
        grid = nil;
    }
    if (showgrid) {
        m = mag;
        if (m < 5)
            m *= 10;
        grid = allocimage(display, Rect(0, 0, m, m),
            CHAN2(CGrey, 8, CAlpha, 8), 1, DTransparent);
        if (grid != nil){
            draw(grid, Rect(0, 0, m, 1), chequer, nil, ZP);
            draw(grid, Rect(0, 1, 1, m), chequer, nil, ZP);
        }
    }
}
```

Uses `chequer-115` 351e, `grid-114` 351d, `mag-117` 351g, and `showgrid-118` 351h.

```
⟨apps/lens.c 352a⟩≡
  #include <u.h>
  #include <libc.h>

  #include <draw.h>
  #include <window.h>
  #include <event.h>

  ⟨enum _anon_ (apps/lens.c) 350c⟩

  ⟨enum _anon_ (apps/lens.c)2 320c⟩

  ⟨global menustr 320a⟩

  ⟨global menu (apps/lens.c) 320b⟩

  ⟨global lastp 351a⟩
  ⟨global red (apps/lens.c) 351b⟩
  ⟨global tmp (apps/lens.c) 351c⟩
  ⟨global grid 351d⟩
  ⟨global chequer 351e⟩
  ⟨global screenfd 351f⟩
  ⟨global mag 351g⟩
  ⟨global showgrid 351h⟩
  ⟨global screenr 351i⟩
  ⟨global screenbuf 351j⟩

  void magnify(void);
  void makegrid(void);

  ⟨function drawit 323a⟩

  ⟨global bypp 351k⟩

  ⟨function main (apps/lens.c) 320d⟩

  ⟨function makegrid 351l⟩

  ⟨function eresized (apps/lens.c) 322⟩

  ⟨function magnify 323b⟩
```

F.2.2 apps/statusbar.c

```
⟨enum _anon_ (apps/statusbar.c) 352b⟩≡                                    (356a)
  enum {PNCTL=3};

⟨global nokill 352c⟩≡                                                     (356a)
  static int nokill;

⟨global textmode 352d⟩≡                                                   (356a)
  static int textmode;

⟨global title 352e⟩≡                                                      (356a)
  static char *title;

⟨global light 352f⟩≡                                                      (356a)
  static Image *light;
```

⟨*global* dark 353a⟩≡ (356a)
```
static Image *dark;
```

⟨*global text (apps/statusbar.c)* 353b⟩≡ (356a)
```
static Image *text;
```

⟨*function* initcolor 353c⟩≡ (356a)
```
static void
initcolor(void)
{
    text = display->black;
    light = allocimagemix(display, DPalegreen, DWhite);
    dark = allocimage(display, Rect(0,0,1,1), CMAP8, 1, DDarkgreen);
}
```
Uses dark-128 353a, light-127 352f, and text-129 353b.

⟨*global* rbar 353d⟩≡ (356a)
```
static Rectangle rbar;
```

⟨*global* ptext 353e⟩≡ (356a)
```
static Point ptext;
```

⟨*global* last 353f⟩≡ (356a)
```
static int last;
```

⟨*global lastp (apps/statusbar.c)* 353g⟩≡ (356a)
```
static int lastp = -1;
```
Uses lastp-135 353g.

⟨*global* backup 353h⟩≡ (356a)
```
static char backup[80];
```

⟨*function* rdenv 353i⟩≡ (356a)
```
/* all code below this line should be in the library, but is stolen from colors instead */
static char*
rdenv(char *name)
{
    char *v;
    int fd, size;

    fd = open(name, OREAD);
    if(fd < 0)
        return 0;
    size = seek(fd, 0, 2);
    v = malloc(size+1);
    if(v == 0){
        fprint(2, "%s: can’t malloc: %r\n", argv0);
        exits("no mem");
    }
    seek(fd, 0, 0);
    read(fd, v, size);
    v[size] = 0;
    close(fd);
    return v;
}
```

⟨*function* newwin 354⟩≡ (356a)

```
int
newwin(char *win)
{
    char *srv, *mntsrv;
    char spec[100];
    int srvfd, cons, pid;

    switch(rfork(RFFDG|RFPROC|RFNAMEG|RFENVG|RFNOTEG|RFNOWAIT)){
    case -1:
        fprint(2, "statusbar: can't fork: %r\n");
        return -1;
    case 0:
        break;
    default:
        exits(0);
    }

    srv = rdenv("/env/wsys");
    if(srv == 0){
        mntsrv = rdenv("/mnt/term/env/wsys");
        if(mntsrv == 0){
            fprint(2, "statusbar: can't find $wsys\n"); //$
            return -1;
        }
        srv = malloc(strlen(mntsrv)+10);
        sprint(srv, "/mnt/term%s", mntsrv);
        free(mntsrv);
        pid  = 0;   /* can't send notes to remote processes! */
    }else
        pid = getpid();
    USED(pid);
    srvfd = open(srv, ORDWR);
    free(srv);
    if(srvfd == -1){
        fprint(2, "statusbar: can't open %s: %r\n", srv);
        return -1;
    }
    sprint(spec, "new -r %s", win);
    if(mount(srvfd, -1, "/mnt/wsys", 0, spec) == -1){
        fprint(2, "statusbar: can't mount /mnt/wsys: %r (spec=%s)\n", spec);
        return -1;
    }
    close(srvfd);
    unmount("/mnt/acme", "/dev");
    bind("/mnt/wsys", "/dev", MBEFORE);
    cons = open("/dev/cons", OREAD);
    if(cons==-1){
    NoCons:
        fprint(2, "statusbar: can't open /dev/cons: %r");
        return -1;
    }
    dup(cons, 0);
    close(cons);
    cons = open("/dev/cons", OWRITE);
    if(cons==-1)
        goto NoCons;
    dup(cons, 1);
    dup(cons, 2);
    close(cons);
```

```
// wctlfd = open("/dev/wctl", OWRITE);
    return 0;
}
```

Uses rdenv() 353i.

⟨*function* screenrect 355a⟩≡ (356a)

```
Rectangle
screenrect(void)
{
    int fd;
    char buf[12*5];

    fd = open("/dev/screen", OREAD);
    if(fd == -1)
        fd=open("/mnt/term/dev/screen", OREAD);
    if(fd == -1){
        fprint(2, "%s: can't open /dev/screen: %r\n", argv0);
        exits("window read");
    }
    if(read(fd, buf, sizeof buf) != sizeof buf){
        fprint(2, "%s: can't read /dev/screen: %r\n", argv0);
        exits("screen read");
    }
    close(fd);
    return Rect(atoi(buf+12), atoi(buf+24), atoi(buf+36), atoi(buf+48));
}
```

⟨*function* postnote 355b⟩≡ (356a)

```
int
postnote(int group, int pid, char *note)
{
    char file[128];
    int f, r;

    switch(group) {
    case PNPROC:
        sprint(file, "/proc/%d/note", pid);
        break;
    case PNGROUP:
        sprint(file, "/proc/%d/notepg", pid);
        break;
    case PNCTL:
        sprint(file, "/proc/%d/ctl", pid);
        break;
    default:
        return -1;
    }

    f = open(file, OWRITE);
    if(f < 0)
        return -1;

    r = strlen(note);
    if(write(f, note, r) != r) {
        close(f);
        return -1;
    }
    close(f);
    return 0;
}
```

Uses PNCTL-123 352b.

```
⟨apps/statusbar.c 356a⟩≡
  #include <u.h>
  #include <libc.h>

  #include <draw.h>
  #include <window.h>

  #include <bio.h>
  #include <event.h>

  ⟨enum _anon_ (apps/statusbar.c) 352b⟩

  static char* rdenv(char*);
  int newwin(char*);
  Rectangle screenrect(void);

  ⟨global nokill 352c⟩
  ⟨global textmode 352d⟩
  ⟨global title 352e⟩

  ⟨global light 352f⟩
  ⟨global dark 353a⟩
  ⟨global text (apps/statusbar.c) 353b⟩

  ⟨function initcolor 353c⟩

  ⟨global rbar 353d⟩
  ⟨global ptext 353e⟩
  static vlong n, d;
  ⟨global last 353f⟩
  ⟨global lastp (apps/statusbar.c) 353g⟩

  ⟨global backup 353h⟩

  ⟨function drawbar 324⟩

  ⟨function eresized (apps/statusbar.c) 325⟩

  ⟨function bar 326a⟩

  ⟨function usage (apps/statusbar.c) 326b⟩
  ⟨function main (apps/statusbar.c) 326c⟩


  ⟨function rdenv 353i⟩

  ⟨function newwin 354⟩

  ⟨function screenrect 355a⟩

  ⟨function postnote 355b⟩
```

F.2.3 apps/winwatch.c

```
⟨struct Win 356b⟩≡                                                    (360)
  struct Win {
      int n;
```

```
    int dirty;
    char *label;
    Rectangle r;
};
```

⟨*global* exclude 357a⟩≡ (360)

```
static Reprog  *exclude  = nil;
```

Uses exclude-137 357a.

⟨*global* win 357b⟩≡ (360)

```
static Win *win;
```

⟨*global* nwin 357c⟩≡ (360)

```
static int nwin;
```

⟨*global* mwin 357d⟩≡ (360)

```
static int mwin;
```

⟨*global* onwin 357e⟩≡ (360)

```
static int onwin;
```

⟨*global* lightblue 357f⟩≡ (360)

```
static Image *lightblue;
```

⟨*enum* _anon_ (apps/winwatch.c) 357g⟩≡ (360)

```
enum {
    PAD = 3,
    MARGIN = 5
};
```

⟨*function erealloc (apps/winwatch.c)* 357h⟩≡ (360)

```
static void*
erealloc(void *v, ulong n)
{
    v = realloc(v, n);
    if(v == nil)
        sysfatal("out of memory reallocating %lud", n);
    return v;
}
```

⟨*function emalloc (apps/winwatch.c)* 357i⟩≡ (360)

```
static void*
emalloc(ulong n)
{
    void *v;

    v = malloc(n);
    if(v == nil)
        sysfatal("out of memory allocating %lud", n);
    memset(v, 0, n);
    return v;
}
```

⟨*function estrdup (apps/winwatch.c)* 358a⟩≡ (360)

```
static char*
estrdup(char *s)
{
    int l;
    char *t;

    if (s == nil)
        return nil;
    l = strlen(s)+1;
    t = emalloc(l);
    memcpy(t, s, l);

    return t;
}
```

⟨*function* `refreshwin` 358b⟩≡ (360)

```
static void
refreshwin(void)
{
    char label[128];
    int i, fd, lfd, n, nr, nw, m;
    Dir *pd;

    if((fd = open("/dev/wsys", OREAD)) < 0)
        return;

    nw = 0;
/* i'd rather read one at a time but rio won't let me */
    while((nr=dirread(fd, &pd)) > 0){
        for(i=0; i<nr; i++){
            n = atoi(pd[i].name);
            sprint(label, "/dev/wsys/%d/label", n);
            if((lfd = open(label, OREAD)) < 0)
                continue;
            m = read(lfd, label, sizeof(label)-1);
            close(lfd);
            if(m < 0)
                continue;
            label[m] = '\0';
            if(exclude != nil && regexec(exclude,label,nil,0))
                continue;

            if(nw < nwin && win[nw].n == n && strcmp(win[nw].label, label)==0){
                nw++;
                continue;
            }

            if(nw < nwin){
                free(win[nw].label);
                win[nw].label = nil;
            }

            if(nw >= mwin){
                mwin += 8;
                win = erealloc(win, mwin*sizeof(win[0]));
            }
            win[nw].n = n;
            win[nw].label = estrdup(label);
            win[nw].dirty = 1;
```

```
            win[nw].r = Rect(0,0,0,0);
            nw++;
        }
        free(pd);
    }
    while(nwin > nw)
        free(win[--nwin].label);
    nwin = nw;
    close(fd);
}
```

Uses `exclude-137` 357a, `mwin-140` 357d, `nwin-139` 357c, and `win-138` 357b.

⟨*function* `geometry` 359a⟩≡ (360)

```
static int
geometry(void)
{
    int i, ncols, z;
    Rectangle r;

    z = 0;
    rows = (Dy(view->r)-2*MARGIN+PAD)/(font->height+PAD);
    if(rows*cols < nwin || rows*cols >= nwin*2){
        ncols = nwin <= 0 ? 1 : (nwin+rows-1)/rows;
        if(ncols != cols){
            cols = ncols;
            z = 1;
        }
    }

    r = Rect(0,0,(Dx(view->r)-2*MARGIN+PAD)/cols-PAD, font->height);
    for(i=0; i<nwin; i++)
        win[i].r = rectaddpt(rectaddpt(r, Pt(MARGIN+(PAD+Dx(r))*(i/rows),
                    MARGIN+(PAD+Dy(r))*(i%rows))), view->r.min);

    return z;
}
```

Uses `MARGIN-146` 357g, `PAD-145` 357g, `cols-143` 360, `nwin-139` 357c, `rows-142` 360, and `win-138` 357b.

⟨*function* `click` 359b⟩≡ (360)

```
static void
click(Mouse m)
{
    int fd, i, j;
    char buf[128];

    if(m.buttons == 0 || (m.buttons & ~4))
        return;

    for(i=0; i<nwin; i++)
        if(ptinrect(m.xy, win[i].r))
            break;
    if(i == nwin)
        return;

    do
        m = emouse();
    while(m.buttons == 4);

    if(m.buttons != 0){
        do
```

```
            m = emouse();
        while(m.buttons);
        return;
    }

    for(j=0; j<nwin; j++)
        if(ptinrect(m.xy, win[j].r))
            break;
    if(j != i)
        return;

    sprint(buf, "/dev/wsys/%d/wctl", win[i].n);
    if((fd = open(buf, OWRITE)) < 0)
        return;
    write(fd, "unhide\n", 7);
    write(fd, "top\n", 4);
    write(fd, "current\n", 8);
    close(fd);
}
```

Uses `nwin-139` 357c and `win-138` 357b.

⟨`apps/winwatch.c` 360⟩≡

```
#include <u.h>
#include <libc.h>
#include <draw.h>
//#include <draw_private.h> // for _string, but should not use it
#include <window.h>
#include <event.h>
#include <regexp.h>

extern Point _string(Image *dst, Point pt, Image *src, Point sp, Font *f, char *s, Rune *r, int len, Rectangle

typedef struct Win Win;
⟨struct Win 356b⟩

⟨global exclude 357a⟩
⟨global win 357b⟩
⟨global nwin 357c⟩
⟨global mwin 357d⟩
⟨global onwin 357e⟩
static int rows, cols;
⟨global lightblue 357f⟩

extern Font *font;

⟨enum _anon_ (apps/winwatch.c) 357g⟩

⟨function erealloc (apps/winwatch.c) 357h⟩
⟨function emalloc (apps/winwatch.c) 357i⟩
⟨function estrdup (apps/winwatch.c) 358a⟩


⟨function refreshwin 358b⟩

⟨function drawnowin 327⟩
⟨function drawwin 328a⟩

⟨function geometry 359a⟩

⟨function redraw (apps/winwatch.c) 328b⟩
```

⟨*function eresized (apps/winwatch.c)* 328c⟩

⟨*function* `click` 359b⟩

⟨*function usage (apps/winwatch.c)* 328d⟩
⟨*function main (apps/winwatch.c)* 328e⟩

Uses `Win` 356b.

F.3 include/

F.3.1 include/complete.h

⟨`include/complete.h` 361a⟩≡

```
#pragma lib "libcomplete.a"
#pragma src "/sys/src/libcomplete"

typedef struct Completion Completion;

⟨struct Completion 246c⟩

Completion*  complete(char *dir, char *s);
void   freecompletion(Completion*);
```

Uses `Completion` 361a.

F.3.2 include/frame.h

⟨`include/frame.h` 361b⟩≡

```
#pragma lib "libframe.a"
#pragma src "/sys/src/libframe"

typedef struct Frbox Frbox;
typedef struct Frame Frame;

⟨enum _anon_ (include/frame.h) 289f⟩

⟨constant FRTICKW 290d⟩

⟨struct Frbox 292a⟩

⟨struct Frame 288a⟩

ulong frcharofpt(Frame*, Point);
Point frptofchar(Frame*, ulong);

int frdelete(Frame*, ulong, ulong);
void frinsert(Frame*, Rune*, Rune*, ulong);

void frselect(Frame*, Mousectl*);
void frselectpaint(Frame*, Point, Point, Image*);

void frdrawsel(Frame*, Point, ulong, ulong, int);
Point  frdrawsel0(Frame*, Point, ulong, ulong, Image*, Image*);

void frinit(Frame*, Rectangle, Font*, Image*, Image**);
void frsetrects(Frame*, Rectangle, Image*);
void frclear(Frame*, int);
```

```
// private??? frame_private.h?
uchar *_frallocstr(Frame*, unsigned);
void _frinsure(Frame*, int, unsigned);
Point _frdraw(Frame*, Point);
void _frgrowbox(Frame*, int);
void _frfreebox(Frame*, int, int);
void _frmergebox(Frame*, int);
void _frdelbox(Frame*, int, int);
void _frsplitbox(Frame*, int, int);
int _frfindbox(Frame*, int, ulong, ulong);
void _frclosebox(Frame*, int, int);
int _frcanfit(Frame*, Point, Frbox*);
void _frcklinewrap(Frame*, Point*, Frbox*);
void _frcklinewrap0(Frame*, Point*, Frbox*);
void _fradvance(Frame*, Point*, Frbox*);
int _frnewwid(Frame*, Point, Frbox*);
int _frnewwid0(Frame*, Point, Frbox*);
void _frclean(Frame*, Point, int, int);
void _frdrawtext(Frame*, Point, Image*, Image*);
void _fraddbox(Frame*, int, int);
Point _frptofcharptb(Frame*, ulong, Point, int);
Point _frptofcharnb(Frame*, ulong, int);
int _frstrlen(Frame*, int);

void frtick(Frame*, Point, int);
void frinittick(Frame*);

void frredraw(Frame*);

⟨function NRUNE 292b⟩
⟨function NBYTE 308d⟩
```

Uses `Frame` 361b and `Frbox` 361b.

F.3.3 include/plumb.h

⟨include/plumb.h 362⟩≡

```
#pragma lib "libplumb.a"
#pragma src "/sys/src/libplumb"

/*
 * Message format:
 * source application\n
 * destination port\n
 * working directory\n
 * type\n
 * attributes\n
 * nbytes\n
 * n bytes of data
 */

typedef struct Plumbattr Plumbattr;
typedef struct Plumbmsg Plumbmsg;

⟨struct Plumbmsg 235b⟩

⟨struct Plumbattr 236a⟩

int      plumbsend(int, Plumbmsg*);
```

```
int      plumbsendtext(int, char*, char*, char*, char*);
Plumbmsg* plumbrecv(int);
char*  plumbpack(Plumbmsg*, int*);
Plumbmsg* plumbunpack(char*, int);
Plumbmsg* plumbunpackpartial(char*, int, int*);
char*  plumbpackattr(Plumbattr*);
Plumbattr* plumbunpackattr(char*);
Plumbattr* plumbaddattr(Plumbattr*, Plumbattr*);
Plumbattr* plumbdelattr(Plumbattr*, char*);
void  plumbfree(Plumbmsg*);
char*  plumblookup(Plumbattr*, char*);
int      plumbopen(char*, int);

int  eplumb(int, char*);
```

Uses `Plumbattr` 362 and `Plumbmsg` 362.

F.4 `libcomplete/`

F.4.1 `libcomplete/complete.c`

⟨`libcomplete/complete.c` 363a⟩≡

```
#include <u.h>
#include <libc.h>
#include "complete.h"

⟨function longestprefixlength 250⟩

⟨function freecompletion 251a⟩

⟨function strpcmp 251b⟩

⟨function complete 249⟩
```

F.5 `libframe/`

⟨*libframe includes* 363b⟩≡ (366 365 364 363c)

```
#include <draw.h>
#include <thread.h>
#include <mouse.h>
#include <frame.h>
```

F.5.1 `libframe/frbox.c`

⟨`libframe/frbox.c` 363c⟩≡

```
#include <u.h>
#include <libc.h>
⟨libframe includes 363b⟩

⟨constant SLOP 292c⟩

⟨function _fraddbox 292d⟩

⟨function _frclosebox 293e⟩

⟨function _frdelbox 293c⟩
```

⟨*function* `_frfreebox` 293d⟩

⟨*function* `_frgrowbox` 292e⟩

⟨*function* `dupbox` 308c⟩

⟨*function* `runeindex` 309c⟩

⟨*function* `truncatebox` 309a⟩

⟨*function* `chopbox` 309b⟩

⟨*function* `_frsplitbox` 308b⟩

⟨*function* `_frmergebox` 313b⟩

⟨*function* `_frfindbox` 308a⟩

F.5.2 `libframe/frdelete.c`

```
⟨libframe/frdelete.c 364a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨function frdelete 314⟩
```

F.5.3 `libframe/frdraw.c`

```
⟨libframe/frdraw.c 364b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨function _frdrawtext 299b⟩

  ⟨function nbytes 319⟩

  ⟨function frdrawsel 317b⟩

  ⟨function frdrawsel0 318⟩

  ⟨function frredraw 302b⟩

  ⟨function frtick 317a⟩

  ⟨function _frdraw 311⟩

  ⟨function _frstrlen 312c⟩
```

F.5.4 `libframe/frinit.c`

```
⟨libframe/frinit.c 364c⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩
```

```
⟨function frinit 288d⟩

⟨function frinittick 290e⟩

⟨function frsetrects 289a⟩

⟨function frclear 289d⟩
```

F.5.5 libframe/frinsert.c

```
⟨libframe/frinsert.c 365a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨constant DELTA 303d⟩
  ⟨constant TMPSIZE 309d⟩
  ⟨global frame 303a⟩

  ⟨function bxscan 310⟩

  ⟨function chopframe 312d⟩

  ⟨struct points_frinsert 303b⟩

  ⟨function frinsert 304⟩
```

F.5.6 libframe/frptofchar.c

```
⟨libframe/frptofchar.c 365b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨function _frptofcharptb 295b⟩

  ⟨function frptofchar 295a⟩

  ⟨function _frptofcharnb 297b⟩

  ⟨function _frgrid 298⟩

  ⟨function frcharofpt 297c⟩
```

F.5.7 libframe/frselect.c

```
⟨libframe/frselect.c 365c⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨function region 302a⟩

  ⟨function frselect 300⟩

  ⟨function frselectpaint 301⟩
```

F.5.8 libframe/frstr.c

```
⟨libframe/frstr.c 366a⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨constant CHUNK (libframe/frstr.c) 294b⟩
  ⟨function ROUNDUP 294c⟩

  ⟨function _frallocstr 294d⟩

  ⟨function _frinsure 294e⟩
```

F.5.9 libframe/frutil.c

```
⟨libframe/frutil.c 366b⟩≡
  #include <u.h>
  #include <libc.h>
  ⟨libframe includes 363b⟩

  ⟨function _frcanfit 297a⟩

  ⟨function _frcklinewrap 296c⟩

  ⟨function _frcklinewrap0 296d⟩

  ⟨function _fradvance 296a⟩

  ⟨function _frnewwid 312a⟩

  ⟨function _frnewwid0 312b⟩

  ⟨function _frclean 313a⟩
```

F.6 libplumb/

F.6.1 libplumb/event.c

```
⟨libplumb/event.c 366c⟩≡
  #include <u.h>
  #include <libc.h>

  #include <draw.h>
  #include <event.h>

  #include "plumb.h"

  typedef struct EQueue EQueue;

  ⟨struct EQueue 244a⟩

  ⟨global equeue 244b⟩
  ⟨global eqlock 244c⟩

  ⟨function partial 244d⟩
```

```
    ⟨function addpartial 245a⟩

    ⟨function plumbevent 245b⟩

    ⟨function eplumb 246a⟩
```
Uses `EQueue` 244a.

F.6.2 libplumb/mesg.c

```
⟨libplumb/mesg.c 367a⟩≡
    #include <u.h>
    #include <libc.h>
    #include "plumb.h"

    ⟨function plumbopen 236c⟩

    ⟨function Strlen 237a⟩
    ⟨function Strcpy 237b⟩

    ⟨function quote 237c⟩

    ⟨function plumbpackattr 238a⟩

    ⟨function plumblookup 238b⟩

    ⟨function plumbpack 239a⟩

    ⟨function plumbsend 239b⟩

    ⟨function plumbline 240a⟩

    ⟨function plumbfree 240b⟩

    ⟨function plumbunpackattr 240c⟩

    ⟨function plumbaddattr 242a⟩

    ⟨function plumbdelattr 242b⟩

    ⟨function plumbunpackpartial 242c⟩

    ⟨function plumbunpack 243a⟩

    ⟨function plumbrecv 243b⟩
```

F.6.3 libplumb/plumbsendtext.c

```
⟨libplumb/plumbsendtext.c 367b⟩≡
    #include <u.h>
    #include <libc.h>
    #include "plumb.h"

    ⟨function plumbsendtext 236b⟩
```

Glossary

API = Application Programming Interface
GUI = Graphical User Interface
IDE = Integrated Development Environment
IPC = Inter Process Communication
RPC = Remote Procedure Call
LOC = Lines Of Code
PTY = Pseudo TTY
TTY = Teletype
WIMP = Window Icon Menu Pointer
MIME = Multipurpose Internet Mail Extensions
PDA = Personal Digital Assistant

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

addpartial(): 245a, 245b
all: 127c, 128b, 128d, 210b, 211a
Alloc-55: 77f, 78d
b: 83a, 84c
BACK: 169a, 173, 175a, 180b, 205a, 289f, 290c, 290e, 302b, 304, 314, 317b
background: 48e, 59d, 221
backup-136: 353h
bandsize(): 90e, 115
bar(): 326a
BIG: 181h, 182a, 261b
bl: 83a, 84d
BORD: 175a, 180b, 289f
Bottom-67: 217c, 260a
boxcursor: 82a, 111c
br: 83a, 84b
button2menu(): 198a, 198b
button3menu(): 88a, 88b
bxscan(): 304, 310
bypp-121: 320d, 323b, 351k
Cd-73: 260c
cexecpipe(): 61c, 62a, 258e
chequer-115: 320d, 351e, 351l
chopbox(): 308b, 309b
chopframe(): 304, 312d
CHUNK-151: 294b
click(): 359b
clickmsec-43: 252c, 253
clickwin-42: 252b, 253
clockfd: 138c, 138d, 139a
cmds-72: 260b
cols-143: 327, 359a, 360
cols-18: 173, 174e, 175a, 205a, 289g
complete(): 247a, 249
Completion: 361a, 361a
Completion (typedef): 361a
Consreadmesg: 147d, 333b
Consreadmesg.c1: 147d
Consreadmesg.c2: 147d

Consreadmesg (typedef): 333b
Conswritemesg: 149c, 333b
Conswritemesg.cw: 149c
Conswritemesg (typedef): 333b
ControlMessage: 91a
cornercursor(): 68d, 70a, 70b, 85c, 90e, 101d, 103c, 110a, 110b, 111b, 111c
cornerpt(): 115, 117
corners: 83a, 85c
CRdata-93: 148b, 148d
CRflush-94: 148b, 268d, 268e
crosscursor: 81, 101d
ctimer-58: 222c, 222e, 223b, 224
Current-68: 217f, 260a
Cut-36: 199c, 230b
cvttorunes(): 150c, 228c, 232f, 234a, 285b
CWdata-90: 149d, 150c
CWflush-91: 149d, 268g, 268h
cxfidalloc-53: 62b, 63, 78d
cxfidfree-54: 62c, 63, 78d, 79c, 267c
d-133: 324, 326a, 356a
dark-128: 324, 325, 353a, 353c
darkgrey-19: 175c, 176a, 176b
DEBUG: 278a, 278c, 278d, 278e, 278f, 278g, 279a, 279b, 279c, 279d, 279e, 279f, 279g, 279h
delete(): 105d, 106a
Delete-11: 89c, 105d
Delete-71: 218b, 260a
deletechan: 107c, 107d, 108b
Deleted: 99a, 106a, 106b, 106c, 218b
deletethread(): 108b
deletetimeoutproc(): 106c, 107c, 114a
DELTA-148: 303d, 304, 310
Deltax-74: 260c
Deltay-75: 260c
derror(): 282d
desktop: 48d, 88b, 101d, 111c, 113a, 115, 120b, 198b, 216a, 221, 222a, 257c, 265
Dirtab: 123a, 333b
dirtab: 123b, 126, 129, 130e, 136a, 210b
Dirtab.name: 123a
Dirtab.perm: 123a
Dirtab.qid: 123a
Dirtab.type: 123a
Dirtab (typedef): 333b
dostat(): 136a, 138a, 138b, 211a
drag(): 90e, 111b, 111c
drawbar(): 324, 325, 326a
drawborder(): 111c, 112b, 115
drawedge(): 112b, 113a
drawit(): 320d, 322, 323a
drawnowin(): 327, 328b

drawwin(): 328a, 328b
dupbox(): 308b, 308c
Ebadfcall: 76e, 280d, 280d
Ebadmouse: 144b, 281l, 281l
Ebadoffset: 281n, 281n
Ebadrect: 257c, 281i, 281i
Ebadreq: 281c, 281c
Ebadtile: 281e, 281e
Ebadwr: 216a, 265, 280f, 280f
Ebadwrect: 281m, 281m
Edeleted: 133b, 281b, 281b
Edge-102: 322, 323a, 350c
Eexist: 131b, 280b, 280b
Einuse: 152d, 159, 212a, 281a, 281a
Elong: 233b, 281g, 281g
emalloc(): 283b
Enotdir: 131d, 280c, 280c
Enowindow: 170b, 281k, 281k
Eoffset: 280e, 280e
Eperm: 127b, 132f, 139c, 139d, 139e, 228a, 280a, 280a
eplumb(): 246a
eqlock-154: 244c, 244d, 245a
EQueue: 244a, 366c
equeue-153: 244b, 244d, 245a
EQueue.buf: 244a
EQueue.id: 244a
EQueue.nbuf: 244a
EQueue.next: 244a
EQueue (typedef): 366c
erealloc(): 283a
eresized(): 322
error(): 75, 76b, 79a, 79b, 92a, 94b, 106d, 109c, 124, 125b, 181f, 182b, 204a, 221, 222a, 257a, 258f, 267c, 282c, 282d, 283a, 283b, 283c
errorshouldabort: 76b, 282a, 282a, 282c
Eshort: 144b, 281f, 281f
estrdup(): 283c
Etooshort: 214d, 281d, 281d
Eunkid: 128c, 281h, 281h
Ewalloc: 216a, 265, 280g, 280g
Ewindow: 257c, 281j, 281j
exclude-137: 357a, 357a, 358b
Exit-13: 89c, 90a
exitchan: 60e, 90a
Exited: 74g, 109b, 109e, 109f
fcall: 56b, 75, 76e
Fid: 53e, 333b
Fid.busy: 54a
Fid.dir: 123c
Fid.fid: 53e

Fid.mode: 53e
Fid.next: 53d
Fid.nrpart: 151a
Fid.open: 53e
Fid.qid: 53e
Fid.rpart: 151a
Fid.w: 55a
Fid (typedef): 333b
FILE: 122c, 133a, 134, 135a, 136b, 137b, 210d
Filsys: 53a, 333b
filsys: 53b, 99b
Filsys.cfd: 53a
Filsys.cxfidalloc: 56a
Filsys.fids: 53a
Filsys.sfd: 53a
Filsys.user: 53a
Filsys (typedef): 333b
filsysattach(): 56b, 126
filsysauth(): 270a, 270b
filsyscancel(): 267d, 268b, 268c, 268e, 268f, 268h, 269a, 269c, 269d
filsysclunk(): 56b, 133c
filsyscreate(): 139b, 139c
filsysflush(): 267a, 267b
filsysinit(): 61c
filsysmount(): 99b, 99c
filsysopen(): 56b, 132a
filsysproc(): 61c, 75
filsysread(): 56b, 135a
filsysremove(): 139b, 139d
filsysrespond(): 76e, 77a, 77c, 77e, 124, 127b, 127c, 128d, 129, 130d, 131a, 132f, 133a, 133b, 133c, 134, 135a, 135b, 136b, 137b, 138a, 139c, 139d, 139e, 143c, 144b, 148d, 150c, 152d, 153b, 153f, 158d, 159, 170b, 206a, 208f, 209a, 209d, 212a, 214d, 215a, 228a, 233b, 267c, 270b
filsysstat(): 56b, 138a
filsysversion(): 76f, 77a
filsyswalk(): 56b, 129
filsyswrite(): 56b, 137a
filsyswstat(): 139b, 139e
firstmessage: 77b, 77b, 77c, 77d
fontname: 229a
Frame: 361b, 361b
frame-150: 303a, 304, 310
Frame (typedef): 361b
FrameColors: 289f
framescroll(): 204a, 253
Frbox: 361b, 361b
Frbox (typedef): 361b
frcharofpt(): 189e, 202, 204b, 253, 297c, 300
frclear(): 109f, 205a, 289d
frdelete(): 183, 193d, 314

frdrawsel(): 169a, 183, 185, 300, 317b
frdrawsel0(): 302b, 317b, 318
Free-56: 77f, 78d
freecompletion(): 247a, 251a
freescrtemps(): 203b, 221
frinit(): 174e, 205a, 288d
frinittick(): 290c, 290e
frinsert(): 179a, 183, 184c, 304
frptofchar(): 169a, 183, 185, 295a, 300, 302b, 314, 316a, 316b, 317b
frredraw(): 186, 302b
frselect(): 253, 300
frselectpaint(): 301, 304, 314
frsetrects(): 205a, 288d, 289a
frtick(): 302b, 314, 316a, 316b, 317a, 317b
FRTICKW: 290d, 290e, 317a
geometry(): 328b, 328c, 359a
getclock(): 135a, 138a, 139a
getsnarf(): 230d, 230e, 232g, 234a, 253
goodrect(): 115, 216a, 257c, 261b, 265
grid-114: 323b, 351d, 351l
hidden: 118c, 118e, 119a, 119b, 119c, 119e, 120b, 214a, 218a, 221
Hidden-14: 89c, 119c, 119e
Hidden-76: 260c
hide(): 118a, 118b
Hide-12: 89c, 118a
Hide-69: 217g, 260a
HIGH: 175a, 289f, 302b, 304, 314, 317b
HiWater-33: 177d, 178a, 178b
holdcol-22: 270f, 271a, 271b, 271e
Holdoff: 271f, 271g, 271h, 271i, 272b
Holdon: 271f, 271g, 272b
HTEXT: 175a, 175c, 271a, 289f, 302b, 317b
iconinit(): 59d
id-17: 50b, 94c
Id-77: 260c
idcmp(): 211a, 211b
initcmd(): 226d
initcolor(): 353c
input: 51e, 64, 66e, 101d, 103c, 105a, 105b, 106d, 110a, 110b, 111b, 114a, 144b, 169a, 175c, 214a, 271a
interruptproc(): 194b, 194c
isalnum(): 192d, 255a, 284c
kbdargv: 226h
keyboardctl: 48b, 64, 228c
keyboardhide(): 227c, 228d
keyboardsend(): 228b, 228c
keyboardthread(): 64
killprocs(): 219c, 220a
Kscrollonedown: 189d, 199e, 203a
Kscrolloneup: 190a, 199e, 203a

l: 83a, 84e
last-134: 324, 325, 353f
lastcursor: 85a, 85b, 235a
lastp-111: 320d, 323b, 351a
lastp-135: 324, 325, 353g, 353g
left: 254e, 255a
left1-46: 254a, 254e
left2-48: 254c, 254e, 254f
left3-49: 254d, 254e, 254f
light-127: 324, 325, 352f, 353c
lightblue-144: 327, 328a, 328b, 357f
lightholdcol-23: 271a, 271d, 271e
lighttitlecol-21: 96a, 96c, 96d
longestprefixlength(): 249, 250
LoWater-34: 177d, 178b
mag-117: 320d, 323b, 351g, 351g, 351l
magnify(): 323a, 323b
main-122(): 320d
makegrid(): 320d, 351l
MARGIN-146: 327, 357g, 359a
max(): 102b, 178a, 284b
Maxmag-103: 320d, 322, 350c
MAXSNARF-87: 233a, 233b
maxtab: 195a, 195b, 195c, 195c
Maxx-78: 260c
Maxy-79: 260c
menu-110: 320b, 320d
menu2: 198b, 199a
menu2str: 199a, 199b, 200b
menu3: 88b, 89a
menu3str: 89a, 89b, 119c
menuing: 87, 101d, 102a, 110a, 111c
menustr-109: 320a, 320b
messagesize: 75, 76a, 76a, 77a, 124, 125a, 133a, 135a, 138a, 184c, 235a, 259a
Mexit-108: 320c, 320d
Mgrid-106: 320c, 320d
min(): 102b, 143c, 178a, 178b, 193c, 262b, 284a
MinWater-35: 177d, 178a
Minx-80: 260c
Miny-81: 260c
MMouse-5: 65a, 66b, 66e
mouse: 48c, 66e, 68a, 68d, 68f, 70a, 70b, 70c, 87, 90c, 90e, 101d, 103b, 103c, 104a, 110a, 110b, 111b, 111c, 112a, 115, 116a, 144b, 201c, 201d, 215b, 227c, 227d, 228d
mousectl: 48a, 66b, 68c, 70d, 85b, 88b, 93b, 101d, 103c, 110a, 111c, 112a, 115, 116a, 116c, 198b, 220c, 228d
Mouseinfo: 161b, 333b
Mouseinfo.counter: 161c
Mouseinfo.lastb: 162a
Mouseinfo.lastcounter: 161c
Mouseinfo.qfull: 161b

Mouseinfo.queue: 161b
Mouseinfo.ri: 161b
Mouseinfo.wi: 161b
Mouseinfo (typedef): 333b
Mousereadmesg: 142c, 333b
Mousereadmesg.cm: 142c
Mousereadmesg (typedef): 333b
Mousestate: 162b, 333b
Mousestate.counter: 162b
Mousestate (typedef): 333b
mousethread(): 66a
move(): 111a, 111b
Move-10: 89c, 111a
Move-62: 216a, 260a
Moved: 90e, 91a, 111b, 114a
Movemouse: 144b, 144c, 144d
MRdata-96: 143a, 143c
Mredraw-107: 320c, 320d
MReshape-6: 220b, 220c, 220d
MRflush-97: 143a, 268a, 268b
msec(): 224, 225a
Munzoom-105: 320c, 320d
mwin-140: 357d, 358b
Mzoom-104: 320c, 320d
n-132: 324, 326a, 356a
N-57: 77f, 78d
NALT-7: 65a, 65b, 66a
namecomplete(): 246b, 247a
NBYTE: 294e, 308c, 308d, 313b
nbytes(): 318, 319
NCOL: 288d, 289f, 289g
NCR-95: 143b, 148b, 148d
NCW-92: 149d, 150b, 150c
new(): 92b, 92c, 257c, 265
New-60: 218c, 259b, 260a
New-8: 89c, 92b
newfid(): 54b, 75, 130c
newrect(): 262b
newwin(): 354
Nhash: 53a, 53c, 54b
nhidden: 118d, 118e, 119a, 119b, 119c, 120b, 214a, 218a, 221
NMR-98: 143a, 143c, 214d
nokill-124: 326a, 352c
Noscroll-64: 217a, 260a
Noscrolling-85: 260c
NRUNE: 292b, 295b, 297c, 304, 308a, 312c, 312d
nsnarf: 230e, 231a, 231c, 232f, 232g, 233h, 233k, 234a
ntsnarf-89: 232d, 232f, 233b, 233e
numeric(): 210b, 210e

NWALT-32: 71a, 71b
NWCR-101: 214b
nwin-139: 328b, 357c, 358b, 359a, 359b
nwindow: 51b, 69, 92c, 106d, 128a, 211a, 215b, 220a, 221
oknotes: 219a, 219c
onscreen(): 101d, 102b, 115
onwin-141: 328b, 357e
PAD-145: 327, 357g, 359a
paleholdcol-24: 270f, 271c, 271e
params-86: 261a
partial(): 244d, 245b
Paste-37: 199c, 230d
PID-82: 260c
Plumb-39: 199c, 234b
plumbaddattr(): 242a
Plumbattr: 362, 362
Plumbattr (typedef): 362
plumbdelattr(): 242b
plumbevent(): 245b, 246a
plumbfree(): 235a, 240b, 242c
plumbline(): 240a, 242c
plumblookup(): 238b
Plumbmsg: 362, 362
Plumbmsg (typedef): 362
plumbopen(): 235a, 236c, 246a
plumbpack(): 239a, 239b
plumbpackattr(): 238a, 239a
plumbrecv(): 243b
plumbsend(): 235a, 236b, 239b
plumbsendtext(): 236b
plumbunpack(): 243a
plumbunpackattr(): 235a, 240c, 242c
plumbunpackpartial(): 242c, 243a, 243b, 244d, 245b
PNCTL-123: 326a, 352b, 355b
points_frinsert.pt0: 303b
points_frinsert.pt1: 303b
points_frinsert: 303b, 303c
pointto(): 106a, 110a, 111b, 113c, 118b
portion(): 86b, 86c
post(): 256b, 257a, 258e
postnote(): 220a, 326a, 355b
ptext-131: 325, 353e
putsnarf(): 231a, 233k
Qcons: 146, 147a, 148d, 150c
Qconsctl: 152a, 152b, 152d, 153a, 153b
Qcursor: 153c, 153d, 153e, 153f, 153g
Qdir: 122d, 123b, 126, 130e, 136a, 211a
QID: 122a, 130a, 138b, 211a
Qkbdin: 227g, 227h, 228a, 228b

Qlabel: 208d, 208e, 208f, 209a
QMAX: 122d
Qmouse: 141a, 141b, 143c, 144b, 159, 168b
Qscreen: 209b, 209c, 209d
Qsnarf: 232a, 232b, 232d, 232f, 232g, 233b, 233c
Qtext: 205b, 205c, 206a
query: 82d, 235a
quote(): 237c, 238a
Qwctl: 211c, 211d, 212a, 212b, 214d, 215a
Qwdir: 251c, 251d, 251e, 252a
Qwindow: 169b, 169c, 170b
Qwinid: 208a, 208b, 208c
Qwinname: 158a, 158b, 158c
Qwsys: 209e, 210a, 210b, 210d, 211a
Qwsysdir: 136a, 210b, 210c, 210d
Qxxx: 122d
r: 83a, 84a
R-83: 260c
Rawoff: 160b, 160c, 160d, 160e
Rawon: 160b, 160d, 160e
rbar-130: 324, 325, 353d
rcargv: 97a, 97c
rdenv(): 353i, 354
readwindow(): 170b, 170c
rectonscreen(): 216a, 263a
red: 48f, 59d, 101d, 115, 216a, 265
red-112: 320d, 323a, 351b
Ref (typedef): 333b
Refresh: 168b, 168c, 168d
refreshwin(): 358b
region(): 300, 302a
Reshape-9: 89c, 113b
Reshaped: 90e, 91a, 113c, 114a, 118e, 120b, 216a, 221
resize(): 113b, 113c
Resize-61: 216a, 260a
resized(): 220d, 221
right: 254f, 255a
right1-47: 254b, 254f
RightMenuCommand: 89c
riosetcursor(): 70a, 85b, 85c, 87, 101d, 110a, 111c, 235a
riostrtol(): 263b
ROUNDUP-152: 294c, 294d, 294e
rows-142: 327, 359a, 360
runeindex(): 309a, 309b, 309c, 313b
runemalloc: 150c, 183, 197b, 228c, 247a, 269a, 284d
runemove: 164c, 165a, 166a, 177a, 178b, 183, 184c, 193a, 197b, 231a, 247a, 285a
runerealloc: 164c, 178a, 231a, 232f, 234a, 284e
runetobyte(): 206b, 232g, 235a, 285d
screenbuf-120: 320d, 323b, 351j

screenfd-116: 320d, 323b, 351f
screenr-119: 320d, 323b, 351i
screenrect(): 355a
Scroll-41: 199c, 200b, 200c
Scroll-63: 216b, 260a
Scrollgap: 174d, 174e, 205a
scrolling: 92b, 225c, 257c
Scrolling-84: 260c
Scrollwid: 174a, 174b, 174e, 205a
scrpos(): 180b, 180c
scrtemps(): 180b, 182a
scrtmp-50: 180b, 181g, 182a, 182b, 203b
Selborder: 66e, 67, 86a, 95b, 96a, 101d, 105b, 114c, 115, 169a, 173, 174b, 174e, 205a, 216a, 257c, 265, 270f
selectq-45: 204b, 252e, 253
selectwin-44: 204a, 252d, 253
Send-40: 199c, 230e
set(): 262a
Set-65: 218d, 260a
shift(): 262c, 263a
showcandidates(): 247a, 247b
showgrid-118: 320d, 323b, 351h, 351h, 351l
shutdown(): 219c
sightcursor: 82b, 110a
SLOP-147: 292c, 292d
snarf: 230e, 231a, 231c, 232f, 232g, 233i, 233k, 234a
Snarf-38: 199c, 230c
snarffd: 232c, 233f, 233k, 234a
snarfversion: 231a, 233b, 233c, 233j
srvpipe: 256a, 256b
srvwctl: 258b, 258e
STACK: 61a, 63, 92c, 222e
startdir: 96f, 96h
Strcpy(): 237b, 239a
Stringpair: 148a, 333b
Stringpair.ns: 148a
Stringpair.s: 148a
Stringpair (typedef): 333b
Strlen(): 237a, 238a, 239a
strpcmp(): 249, 251b
strrune(): 255a, 285c
sweep(): 92b, 101d, 113c
sweeping: 88b, 89d, 90e, 144d, 153g
t-15: 83a, 83c
TEXT: 175a, 175c, 271a, 289f, 290e, 302b, 304, 317b
text-129: 324, 325, 353b, 353c
textmode-125: 324, 326a, 352d
Timer: 223a, 333b
timer-59: 222f, 223b, 223c
Timer.c: 223a

Timer.cancel: 223a
Timer.dt: 223a
Timer.next: 223a
Timer (typedef): 333b
timercancel(): 203c, 223d
timerinit(): 222e
timerproc(): 222e, 224
timerstart(): 203c, 223b
timerstop(): 203c, 223c, 224
title-126: 325, 352e
titlecol-20: 96a, 96b, 96d
tl: 83a, 83b
tmp-113: 322, 323a, 323b, 351c
TMPSIZE-149: 309d, 310
Top-66: 215b, 217b, 260a
topped-16: 51c, 94c, 104b, 114c, 217d, 217e
tr: 83a, 83d
truncatebox(): 308b, 309a
tsnarf-88: 232d, 232f, 233b, 233d
unhide(): 119d, 119e
Unhide-70: 218a, 260a
Unselborder: 105b, 105c, 169a
usage(): 225b
viewr: 47, 221
waddraw(): 164b, 164c, 230e, 231c
Wakeup: 198b, 212a, 212d, 216b, 217a, 272c, 272d
wbacknl(): 182c, 182d, 190b, 202, 204b
wborder(): 95b, 96a, 105b, 114c, 169a
wbottomme(): 217c, 217e
wbswidth(): 191a, 191b, 192a
wclickmatch(): 255a, 255b
wclose(): 97d, 109a, 109b, 134, 198b, 199e
wclosewin(): 106c, 106d, 109d
wcontents(): 206a, 206b
WCRdata-99: 214b, 214d
WCread-29: 165b, 165f, 165g, 166a, 188a, 270d
WCRflush-100: 214b, 269b, 269c
WCtl-27: 71a, 74f, 74g
wctlfd: 258a, 258e, 259a
Wctlmesg: 91b, 333b
wctlmesg(): 74g, 92a
Wctlmesg.image: 91b
Wctlmesg.r: 91b
Wctlmesg.type: 91b
Wctlmesg (typedef): 333b
wctlnew(): 218c, 259b, 265
wctlproc(): 258f, 259a
wctlthread(): 258f, 259b
wcurrent(): 92c, 104b, 105a, 114a, 217f

wcut(): 230a, 230b, 231b, 231c, 253
WCwrite-30: 195f, 196c, 196d, 196f
wdelete(): 192a, 193a, 197b, 231b
wdoubleclick(): 253, 255a
wfill(): 183, 184c, 193d, 205a
wframescroll(): 204a, 204b
whichcorner(): 85c, 86b, 115
whichrect(): 115, 116d
whide(): 118b, 118e, 217g
whitearrow: 82c, 270e
WIN: 122b, 129, 136a, 138a
Win: 356b, 360
win-138: 328a, 328b, 357b, 358b, 359a, 359b
Win.dirty: 356b
Win.label: 356b
Win.n: 356b
Win.r: 356b
Win (typedef): 360
winborder(): 70c, 85c, 86a, 90c
winclosechan: 108c, 109a, 131c, 133c, 210b
winclosethread(): 109a
winctl(): 71b, 92c
windfilewidth(): 247a, 248
Window: 49, 333b
Window.cctl: 74d
Window.ck: 72b
Window.consread: 147b
Window.conswrite: 149a
Window.ctlopen: 152c
Window.cursor: 86d
Window.cursorp: 86d
Window.deleted: 107a
Window.dir: 96g
Window.frm: 52d
Window.holding: 270c
Window.i: 50c
Window.id: 50a
Window.label: 50e
Window.lastsr: 181c
Window.maxr: 51h
Window.mc: 74a
Window.mouse: 161a
Window.mouseopen: 51f
Window.mouseread: 142a
Window.name: 50a
Window.namecount: 101b
Window.notefd: 98b
Window.nr: 51h
Window.nraw: 164a

Window.org: 52b
Window.pid: 98a
Window.q0: 52a
Window.q1: 52a
Window.qh: 52c
Window.r: 51h
Window.raw: 164a
Window.rawing: 160a
Window.resized: 114b
Window.screenr: 50d
Window.scrolling: 52f
Window.scrollr: 52e
Window.topped: 51d
Window.wctlopen: 212c
Window.wctlread: 213a
Window.wctlready: 212c
Window (typedef): 333b
windows: 51a, 69, 92c, 106d, 128a, 211a, 215b, 220a, 221
winsert(): 177a, 187c, 196f, 230e, 231c, 246b, 247b
winshell(): 97c, 97d
WKey-25: 71a, 72d, 73a
wkeyboard: 226g, 227c, 227d, 227e, 227f, 228a, 228d
wkeyctl(): 73a, 73b, 165a, 199e
wlookid(): 127c, 128a, 210b
wmk(): 92c, 94c
WMouse-26: 71a, 74b, 74c
wmousectl(): 199d, 199e
WMouseread-28: 163a, 163e, 163f, 163h
wmovemouse(): 115, 116c, 144d, 202
word(): 261c
wpaste(): 230d, 231c, 253
wplumb(): 234b, 235a
wpointto(): 68f, 69, 87, 104b, 110a, 144b
wrefresh(): 168d, 169a
wrepaint(): 105a, 105b, 272a, 272b
wresize(): 114a, 114c
writewctl(): 215a, 215b
wscrdraw(): 169a, 179b, 180a, 180b, 183, 196f, 205a, 230b, 230d, 253
wscroll(): 201e, 202
wscrsleep(): 202, 203c, 204b
wselect(): 199e, 253
wsendctlmesg(): 90e, 91c, 99a, 106a, 109b, 111b, 113c, 118e, 120b, 144b, 160b, 160c, 168b, 198b, 212a, 212d, 216a, 216b, 217a, 218b, 221, 271g, 271h, 271i
wsetcols(): 175b, 175c, 205a
wsetcursor(): 68d, 85c, 87, 105a, 105b, 106d, 115, 153e, 153g
wsetname(): 92c, 101a, 114c
wsetorigin(): 182c, 183, 189e, 190b, 202, 204b
wsetpid(): 92c, 98c, 218d
wsetselect(): 183, 185, 190c, 190d, 190e, 191a, 192a, 196f, 204b, 205a, 230e, 231b, 231c, 247b, 253

wshow(): 179b, 187c, 189a, 189b, 190c, 190d, 190e, 191a, 194b, 196g, 200c, 216b, 230e, 246b
wsnarf(): 230a, 230b, 230c, 230e, 231a, 253
wtop(): 104a, 104b
wtopme(): 217b, 217d, 227d
wunhide(): 119e, 120b, 218a
WWread-31: 213f, 213g, 213h, 214a
Xfid: 55b, 333b
xfid-52: 78a, 78d, 267c
Xfid.active: 266c
Xfid.buf: 55b
Xfid.c: 55b
Xfid.f: 55b
Xfid.flushc: 266a
Xfid.flushing: 266a
Xfid.flushtag: 266a
Xfid.free: 78c
Xfid.fs: 55b
Xfid.next: 78c
Xfid.req: 55b
Xfid (typedef): 333b
xfidallocthread(): 63, 78d
xfidattach(): 126, 127c
xfidclose(): 133c, 134
xfidctl(): 78d, 79c
xfidflush(): 267b, 267c
xfidfree-51: 78b, 78d
xfidinit(): 63
xfidopen(): 132a, 133a
xfidread(): 135a, 136b
xfidwrite(): 137a, 137b
_fraddbox(): 292d, 304, 308c
_fradvance(): 295b, 296a, 297c, 304, 312d, 313a, 314
_frallocstr(): 294d, 294e, 308c, 310
_frcanfit(): 296d, 297a, 304, 311, 314
_frcklinewrap(): 295b, 296c, 297c, 299b, 304, 312d, 313a, 314, 318
_frcklinewrap0(): 296d, 304, 310, 311, 314
_frclean(): 304, 313a, 314
_frclosebox(): 293c, 293e, 314
_frdelbox(): 293b, 293c, 304, 311, 312d, 313b
_frdraw(): 310, 311
_frdrawtext(): 299b, 304
_frfindbox(): 304, 308a, 314
_frfreebox(): 293c, 293d, 314
_frgrid(): 297c, 298
_frgrowbox(): 292d, 292e, 310
_frinsure(): 294e, 313b
_frmergebox(): 313a, 313b
_frnewwid(): 304, 311, 312a, 314
_frnewwid0(): 312a, 312b, 314

_frptofcharnb(): 297b, 304, 314
_frptofcharptb(): 295a, 295b, 297b, 314
_frsplitbox(): 304, 308a, 308b, 311, 314
_frstrlen(): 304, 311, 312c
__anon_enum_10: 149d
__anon_enum_11: 148b
__anon_enum_12: 143a
__anon_enum_13: 214b
__anon_enum_14: 350c
__anon_enum_15: 320c
__anon_enum_16: 352b
__anon_enum_17: 357g
__anon_enum_1: 203a
__anon_enum_2: 333a
__anon_enum_3: 65a
__anon_enum_4: 71a
__anon_enum_5: 177d
__anon_enum_6: 199c
__anon_enum_7: 77f
__anon_enum_8: 260a
__anon_enum_9: 260c
__anon_struct_1.bc: 292a
__anon_struct_1.minwid: 292a
__anon_struct_1: 292a
__anon_struct_2.ptr: 292a
__anon_struct_2: 292a

Bibliography

[FDFH90] James Foley, Andries Van Dam, Steven Feiner, and John Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, 1990. cited page(s) 10, 16

[GRA89] James Gosling, David S.H. Rosenthal, and Michelle J. Arden. *The NeWS Book, an Introduction to the Network/Extensible Window System*. Springer-Verlag, 1989. cited page(s) 10, 16

[HDF+86] F.R.A. Hopgood, D.A. Duce, E.V.C Fielding, K. Robinson, and A.S. Williams. *Methodology of Window Management*. Springer-Verlag, 1986. cited page(s) 16

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. Available at http://www.usingcsp.com/cspbook.pdf. cited page(s) 79

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 17

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 16

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 17

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 12, 16, 24, 30, 33, 34, 36, 41, 48, 51, 103, 146, 207, 276, 280

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 16, 32, 36, 37, 38, 39, 40, 42, 44, 50, 58, 97, 135, 154, 276, 278, 280, 282, 283

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 207, 282

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 11, 12, 15, 16, 21, 24, 26, 27, 30, 31, 32, 33, 35, 36, 37, 38, 40, 42, 47, 48, 50, 60, 65, 85, 89, 94, 95, 101, 104, 154, 155, 156, 168, 209, 229, 256, 276

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Network Stack /net*. 2016. cited page(s) 13, 16, 42, 52

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 23, 33, 226, 276

[Pad26] Yoann Padioleau. *Principia Softwarica: The Plan 9 Widgets libpanel*. 2026. cited page(s) 27, 276

[Pik83a] Rob Pike. The blit: A multiplexed graphics terminal. Technical report, Bell Labs, 1983. Also available at lib_graphics/docs/blit-1983.pdf. cited page(s) 11, 17

[Pik83b] Rob Pike. Graphics in overlapping bitmap layers. In *SIGGRAPH*, pages 331–356, 1983. Also available at lib_graphics/docs/pike-bitmap-1983.ps. cited page(s) 20, 27

[Pik88] Rob Pike. Window systems should be transparent. In *Computing Systems*, pages 279–296, 1988. Also available at windows/docs/transparent_wsys.pdf. cited page(s) 13, 17, 25, 141

[Pik89] Rob Pike. A concurrent window system. In *Computing Systems*, pages 133–154, 1989. Also available at windows/docs/concurrent_window_system.pdf. cited page(s) 11, 17

[Pik91] Rob Pike. 8 1/2, the plan 9 window system. In *USENIX Summer conference*, pages 257–265, 1991. Also available at windows/docs/81/2.pdf. cited page(s) 11, 12, 17, 31, 154

[Pik00a] Rob Pike. Plumbing and other utilities. Technical report, Bell Labs, 2000. Also available at windows/docs/plumb.ps. cited page(s) 40

[Pik00b] Rob Pike. Rio: Design of a concurrent window system. Technical report, Bell Labs, 2000. Also available at windows/docs/rio_slides.pdf. cited page(s) 11, 17

[PPD+95] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. In *Computing Systems*, pages 221–254, 1995. Also available at docs/articles/9.ps. cited page(s) 12, 16

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Operating Systems Review*, pages 72–76, 1993. Also available at docs/articles/names.ps. cited page(s) 12, 16

[SG86] Robert W. Scheifler and Jim Gettys. The X Window system. In *ACM Transaction of Graphics*, pages 79–109, 1986. cited page(s) 12

[SIKH82] David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem. The star user interface: an overview. In *AFIPS*, pages 515–528, 1982. cited page(s) 20

[TML+79] C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A personal computer. Technical report, Xerox PARC, 1979. CSL-79-11. cited page(s) 10

[WG92] Niklaus Wirth and Jurg Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992. cited page(s) 14