Principia Softwarica: The Plan 9 Windowing System `rio`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike

April 28, 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 **9**

1.1 Motivations . . . 9

1.2 The Plan 9 windowing system: `rio` . . . 10

1.3 Other windowing systems . . . 11

1.4 Getting started . . . 13

1.5 Requirements . . . 14

1.6 About this document . . . 15

1.7 Copyright . . . 15

1.8 Acknowledgments . . . 16

2 Overview **17**

2.1 Windowing system principles . . . 17

2.1.1 The window . . . 19

2.1.2 Display server . . . 20

2.1.3 Window compositor . . . 20

2.1.4 Stacking, tiling, and compositing . . . 20

2.1.5 Window manager . . . 21

2.1.6 Input focus . . . 22

2.1.7 Terminal emulator . . . 22

2.1.8 Windowing system API . . . 23

2.1.9 Window applications versus graphical applications . . . 25

2.2 `rio` interfaces . . . 27

2.2.1 Command-line interface . . . 27

2.2.2 Graphical user interface . . . 27

2.2.3 Filesystem interface . . . 30

2.3 `hellorio.c` . . . 34

2.3.1 Skeleton and output code . . . 35

2.3.2 Input code . . . 37

2.3.3 The event loop . . . 38

2.4 Code organization . . . 39

2.5 Software architecture . . . 40

2.5.1 Processes, procs, and threads relationships . . . 41

2.5.2 Trace of a new window creation . . . 44

2.5.3 Trace of a mouse click . . . 48

2.5.4 Trace of a key press . . . 50

2.5.5 Trace of a drawing operation . . . 51

2.6 Book structure . . . 55

3 Core Data Structures **57**
3.1 Device handles . . . 57
3.1.1 Output device: `display` and `view` . . . 57
3.1.2 Input devices: `mousectl` and `keyboardctl` . . . 57
3.2 Desktop, `desktop` . . . 58
3.3 Windows . . . 58
3.3.1 `Window` . . . 58
3.3.2 `windows` . . . 61
3.3.3 `current` . . . 61
3.3.4 Graphical windows . . . 61
3.3.5 Textual windows . . . 61
3.4 Filesystem server . . . 62
3.4.1 `FilSys` and `filsys` . . . 62
3.4.2 File state: `Fid` . . . 63
3.4.3 Workers and jobs: `Xfid` . . . 64
3.4.4 9P callbacks: `fcall` . . . 65

4 `main()` **66**
4.1 Graphics initialization . . . 67
4.2 Mouse initialization . . . 67
4.3 Keyboard initialization . . . 68
4.4 Channels creation . . . 68
4.5 Threads creation . . . 68
4.6 Filesystem server initialization . . . 69
4.6.1 `filsysinit()` . . . 69
4.6.2 Worker allocator: `xfidinit()` . . . 70

5 Procs and Threads **71**
5.1 Keyboard thread . . . 71
5.2 Mouse thread . . . 72
5.2.1 Application mouse events . . . 74
5.2.2 Windowing system mouse events . . . 75
5.3 Window threads . . . 77
5.3.1 Keyboard events listening . . . 79
5.3.2 Mouse events listening . . . 80
5.3.3 Control events listening . . . 80
5.4 Filesystem server proc . . . 81
5.4.1 `filsysproc()` . . . 81
5.4.2 `filsysversion()` . . . 83
5.5 `Xfid` allocator thread . . . 83
5.6 `Xfid` threads . . . 85

6 Cursors **86**
6.1 Cursor graphics . . . 86
6.1.1 Classic cursors . . . 86
6.1.2 Border and corner cursors . . . 88
6.2 Setting the cursor . . . 90
6.2.1 `riosetcursor()` . . . 90
6.2.2 `cornercursor()` . . . 90
6.2.3 `wsetcursor()` . . . 91

7 Window Manager **93**
7.1 Overview . . . 93
7.2 Right-click system menu . . . 93
7.3 Window borders click . . . 95
7.4 `Wctlmesg` . . . 95
7.5 Window creation . . . 96
7.5.1 Window thread creation: `new()` . . . 97
7.5.2 `Window` allocation: `wmk()` . . . 98
7.5.3 Window process creation: `winshell()` . . . 100
7.5.4 Namespace adjustments: `filsysmount()` . . . 102
7.5.5 Public layer: `wsetname()` . . . 103
7.5.6 Mouse action `sweep()` . . . 104
7.6 Window focus . . . 106
7.7 Window deletion . . . 107
7.7.1 `delete()` . . . 107
7.7.2 `deletetimeoutproc()` and `deletethread()` . . . 109
7.7.3 `Deleted` and `Exited` . . . 110
7.7.4 Mouse action `pointto()` . . . 111
7.8 Window move . . . 112
7.8.1 `move()` . . . 113
7.8.2 Mouse action `drag()` . . . 113
7.9 Window resize . . . 115
7.9.1 `resize()` . . . 115
7.9.2 Mouse action `bandsize()` . . . 116
7.10 Window visibility . . . 119
7.10.1 `hide()` . . . 119
7.10.2 `hidden` . . . 120
7.10.3 `unhide()` . . . 121

8 Filesystem Server **122**
8.1 Additional data structures . . . 122
8.1.1 `Qxxx` . . . 122
8.1.2 `dirtab` . . . 123
8.2 `filsysrespond()` . . . 124
8.3 Attach . . . 124
8.3.1 `filsysattach()` . . . 125
8.3.2 `xfidattach()` . . . 126
8.4 Walk . . . 127
8.4.1 `filsyswalk()` . . . 127
8.4.2 Cloning `fid` . . . 128
8.4.3 `..` . . . 129
8.4.4 Error management . . . 129
8.5 Open . . . 130
8.5.1 `filsysopen()` . . . 130
8.5.2 `xfidopen()` . . . 131
8.6 Clunk/Close . . . 131
8.6.1 `filsysclunk()` . . . 132
8.6.2 `xfidclose()` . . . 132
8.7 Read . . . 132
8.7.1 `filsysread()` . . . 133

8.7.2 Reading a directory . . . 133
8.7.3 `xfidread()` . . . 134
8.8 Write . . . 134
8.8.1 `filsyswrite()` . . . 134
8.8.2 `xfidwrite()` . . . 135
8.9 Stats . . . 135
8.9.1 `filsysstat()` . . . 135
8.10 Forbidden operations . . . 136

9 Virtual Devices **138**
9.1 Device virtualization across windowing systems . . . 138
9.2 `/mnt/wsys/mouse` . . . 138
9.2.1 Reading part1 . . . 139
9.2.2 Writing . . . 140
9.3 `/mnt/wsys/cons` . . . 141
9.3.1 Reading part1 . . . 141
9.3.2 Writing part1 . . . 143
9.3.3 Bytes versus runes, partial runes . . . 144
9.4 `/mnt/wsys/consctl` . . . 145
9.5 `/mnt/wsys/cursor` . . . 146
9.6 `/dev/draw/` and `/mnt/wsys/winname` . . . 147

10 Graphical Windows **148**
10.1 Graphical window setup . . . 148
10.1.1 `initdraw()` . . . 148
10.1.2 `initmouse()`, mouse-open mode . . . 148
10.1.3 `initkeyboard()`, raw-access mode . . . 149
10.2 Mouse events . . . 150
10.2.1 Mouse state queue . . . 150
10.2.2 `/mnt/wsys/mouse` reading part2 . . . 151
10.3 Keyboard events . . . 153
10.3.1 Raw keys queue . . . 153
10.3.2 `/mnt/wsys/cons` reading part2 . . . 153
10.4 Resize events . . . 156
10.5 `/mnt/wsys/window` . . . 156

11 Textual Windows **159**
11.1 Overview . . . 159
11.2 Textual window creation . . . 160
11.2.1 Scrollbar . . . 160
11.2.2 Frame . . . 161
11.3 Frame widget . . . 161
11.3.1 `Frame` . . . 162
11.3.2 `frinit()` . . . 162
11.3.3 Frame colors . . . 163
11.3.4 Frame tick . . . 165
11.3.5 Frame boxes . . . 166
11.3.6 Frame strings . . . 169
11.3.7 Frame rune position, point, and box number . . . 169
11.3.8 Frame selection . . . 174

11.4 Content modification . . . 174
11.4.1 winsert() . . . 175
11.4.2 Growing array . . . 176
11.5 Content rendering . . . 177
11.5.1 frinsert() . . . 177
11.5.2 frdelete() . . . 188
11.5.3 wshow() . . . 190
11.5.4 Drawing the scrollbar: wscrdraw() . . . 191
11.5.5 Drawing the tick: frtick() . . . 193
11.5.6 Drawing the text and the selection: frdrawsel() . . . 194
11.5.7 Moving the frame origin . . . 196
11.5.8 Selecting . . . 198
11.5.9 Repainting . . . 202
11.6 Keyboard events . . . 203
11.6.1 Text input queue . . . 203
11.6.2 /mnt/wsys/cons reading part3 . . . 203
11.6.3 Navigation keys . . . 205
11.6.4 Special keys . . . 208
11.7 Application output events . . . 211
11.7.1 /mnt/wsys/cons writing part2 . . . 212
11.8 Mouse events . . . 214
11.8.1 Middle click menu . . . 214
11.8.2 Other clicks . . . 215
11.9 Automatic scrolling mode . . . 216
11.10 Scroll bar interaction . . . 217
11.11 Resize . . . 221
11.12 /mnt/wsys/text . . . 222

12 Windowing System Files **223**
12.1 /mnt/wsys/winid . . . 224
12.2 /mnt/wsys/label . . . 224
12.3 /mnt/wsys/screen . . . 225
12.4 /mnt/wsys/wsys/ . . . 225
12.5 /mnt/wsys/wctl . . . 227
12.5.1 Reading . . . 228
12.5.2 Writing (controlling windows) . . . 230

13 Advanced Topics **234**
13.1 External mount: /srv/rio.user.pid . . . 234
13.2 Command-line control: /srv/riowctl.user.pid . . . 235
13.3 Recursive rio . . . 244
13.4 Advanced terminal editing . . . 246
13.4.1 Snarf . . . 246
13.4.2 Plumb . . . 251
13.4.3 Auto complete . . . 262
13.4.4 Word selection . . . 268
13.5 Automatic scrolling: rio -s . . . 272
13.6 Initial command: rio -i . . . 272
13.7 Fake keyboard input: rio -k . . . 273
13.7.1 rio -k . . . 273

13.7.2 wkeyboard . . . 274
13.7.3 /mnt/wsys/kdbin . . . 274
13.7.4 Keyboard hide . . . 275
13.8 Font selection: rio -f . . . 276
13.9 Holding mode . . . 276
13.10 Signals, notes . . . 278
13.11 Timer . . . 279
13.12 Flushing . . . 282
13.13 Security . . . 285
13.14 TODO Wakeup . . . 286
13.15 TODO Refresh . . . 286

14 Conclusion 287
14.1 Patterns and techniques . . . 287
14.2 Connections to other books . . . 288
14.3 Beyond the Plan 9 windowing system . . . 288

A Debugging 289

B Error Management 290
B.1 Error codes . . . 290
B.2 error(), derror() . . . 292

C Utilities 293

D Examples of Windowing System Applications TODO 296

E Extra Code 297
E.1 include/ . . . 297
E.1.1 include/complete.h . . . 297
E.1.2 include/frame.h . . . 297
E.1.3 include/plumb.h . . . 298
E.2 windows/rio/ . . . 299
E.2.1 windows/rio/dat.h . . . 299
E.2.2 windows/rio/fns.h . . . 301
E.2.3 windows/rio/globals.c . . . 304
E.2.4 windows/rio/rio.c . . . 305
E.2.5 windows/rio/thread_mouse.c . . . 305
E.2.6 windows/rio/thread_keyboard.c . . . 306
E.2.7 windows/rio/threads_window.c . . . 306
E.2.8 windows/rio/threads_worker.c . . . 307
E.2.9 windows/rio/threads_misc.c . . . 307
E.2.10 windows/rio/wm.c . . . 308
E.2.11 windows/rio/data.c . . . 309
E.2.12 windows/rio/cursor.c . . . 310
E.2.13 windows/rio/wind.c . . . 311
E.2.14 windows/rio/processes_winshell.c . . . 312
E.2.15 windows/rio/terminal.c . . . 312
E.2.16 windows/rio/snarf.c . . . 315
E.2.17 windows/rio/graphical_window.c . . . 315
E.2.18 windows/rio/9p.c . . . 316

E.2.19 windows/rio/proc_fileserver.c . . . 316
E.2.20 windows/rio/fsys.c . . . 317
E.2.21 windows/rio/xfid.c . . . 318
E.2.22 windows/rio/scrl.c . . . 319
E.2.23 windows/rio/time.c . . . 320
E.2.24 windows/rio/wctl.c . . . 320
E.2.25 windows/rio/error.c . . . 321
E.2.26 windows/rio/util.c . . . 321
E.3 windows/apps/ . . . 322
E.3.1 windows/apps/lens.c . . . 322
E.3.2 windows/apps/statusbar.c . . . 328
E.3.3 windows/apps/winwatch.c . . . 335
E.4 windows/libcomplete/ . . . 341
E.4.1 windows/libcomplete/complete.c . . . 341
E.5 windows/libframe/ . . . 341
E.5.1 windows/libframe/frbox.c . . . 341
E.5.2 windows/libframe/frdelete.c . . . 342
E.5.3 windows/libframe/frdraw.c . . . 342
E.5.4 windows/libframe/frinit.c . . . 343
E.5.5 windows/libframe/frinsert.c . . . 343
E.5.6 windows/libframe/frptofchar.c . . . 343
E.5.7 windows/libframe/frselect.c . . . 344
E.5.8 windows/libframe/frstr.c . . . 344
E.5.9 windows/libframe/frutil.c . . . 344
E.6 windows/libplumb/ . . . 345
E.6.1 windows/libplumb/event.c . . . 345
E.6.2 windows/libplumb/mesg.c . . . 345
E.6.3 windows/libplumb/plumbsendtext.c . . . 346

Glossary **347**

Index **348**

References **359**

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).

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.

The kernel-and-windowing-system parallel is worth taking seriously. Both multiplex scarce shared resources—the kernel multiplexes CPU and memory across processes, the windowing system multiplexes the screen, mouse, and keyboard across windows. Both enforce isolation—one process cannot stomp on another's memory, one window cannot draw into another's rectangle. Both dispatch asynchronous events—the kernel routes interrupts to device drivers, the windowing system routes mouse clicks to the focused window. The key difference is that the kernel runs in privileged mode while a windowing system like `rio` runs entirely in user space as a regular program, which makes it an unusually good object of study: everything interesting happens through ordinary system calls and ordinary data structures.

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.

Most questions in the list below come in pairs: a what question (what does the windowing system do when X happens) and a how question (how does it implement that). The goal of this book is to connect those two sides. If you only read the answers to the what questions, you end up with a vocabulary without a mental

model—terms like "window manager," "compositor," "display server," and "client" that feel interchangeable. Following the how through `rio`'s 8 800 lines of C grounds each term in a concrete data structure or thread, and makes clear why those distinctions exist in the first place.

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?

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.

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

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: `8-1/2` [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 one of the first windowing system back in 1982[2]. It was created even before the Macintosh, X Window, and Microsoft Windows.

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. 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—it never learns which. A UNIX windowing system needs a separate library (Xlib, the Wayland client, the Win32 API) 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 client library.

A nice side effect of the multiplexer approach used by `rio` is that `rio` can run under itself (see Section 13.3). 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

[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`.

[3] For more information, see Section 2.1.8 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`, `8-1/2` [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`

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 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 30 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 client/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 generic 9P protocol.

Finally, the graphics and windowing system parts 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 ten 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.

[6] https://cgit.freedesktop.org/xorg
[7] https://cgit.freedesktop.org/xorg/xserver/
[8] https://wayland.freedesktop.org/
[9] https://cgit.freedesktop.org/wayland/weston/

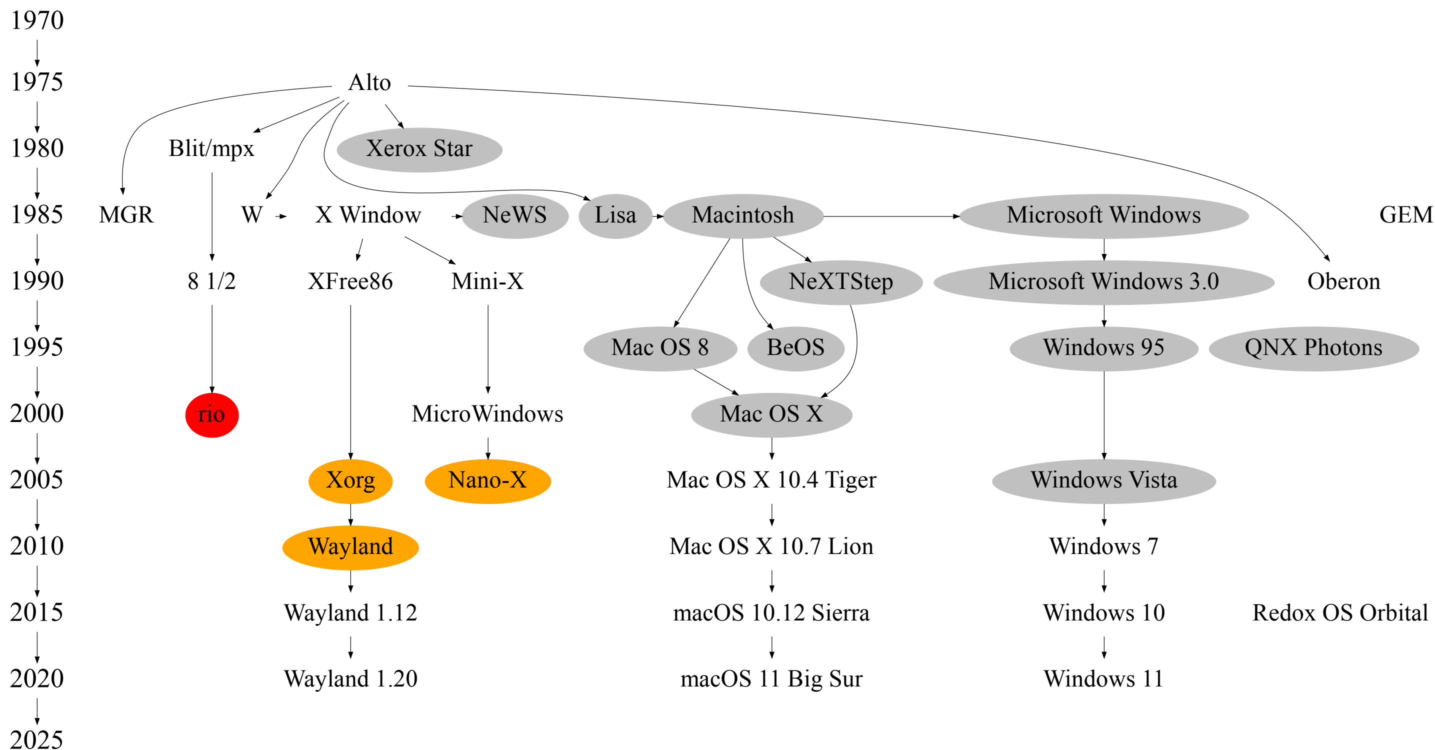


Figure 1.1: Windowing systems timeline

- Nano-X[10] (previously known as MicroWindows) is a windowing system and graphics system designed originally for small devices such as PDAs. It 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 client/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.

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).

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
```

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

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

```
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[11] 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`.

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.

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.

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

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 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.

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:

- `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 *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 desktop systems KDE[1] and GNOME[2] 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.

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`).

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

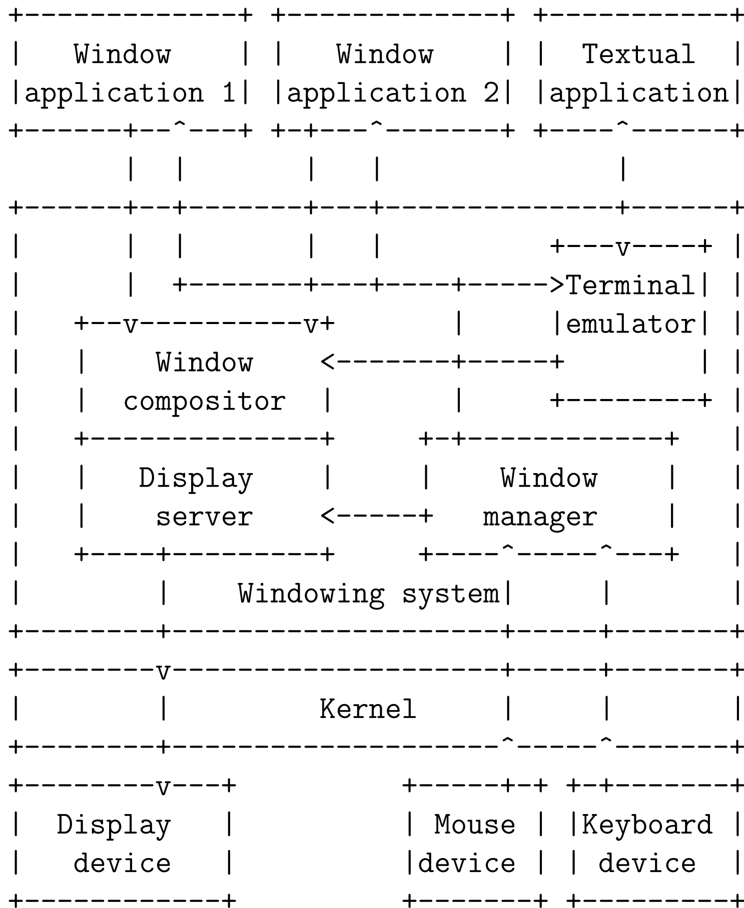


Figure 2.1: Components of a windowing system.

```
+-----------------------------------+
|+-------------------+----icons----+|
||                   |     +--+ \ /||
||       Title Bar   |     |  |  X ||
||                   | --- +--+ / \||
|+-------------------+-------------+|
||                                 ||
||                                 ||
||                                 ||
||          Window Content         ||
||                                 ||
||                                 ||
||                                 ||
||                                 ||
|+---------------------------------+|
+-----------Window border------------+
```

Figure 2.2: Window elements.

- *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.

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[3]. 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

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

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).

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].

2.1.2 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 2.1.3). A display server uses a client/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.

The first display server was Rob Pike's Blit terminal (Bell Labs, 1982), which multiplexed a graphical display over a serial line. 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/x/data` files).

2.1.3 Window compositor

A *window compositor* is an optional component of a windowing system (found in modern windowing systems) allowing windows to be *composited* with each other. 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.

2.1.4 Stacking, tiling, and compositing

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. Classic X11 without a compositor, old Microsoft Windows,

[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.

`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 `8½`—the predecessor to `rio`—was a tiler. Today the idea lives on in keyboard-driven managers like `i3`, `dwm`, `xmonad`, `awesome`, and `Sway`. 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. Pike moved Plan 9 from `8½` to `rio` precisely because users wanted more windows than fit on screen at once, which pure tiling cannot give without scrolling or workspaces.

A compositing system is the modern answer: 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), `compton`/`picom`, 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 plus a rendering pipeline running at vsync, which is why compositing only became practical once GPUs became universal.

`rio` is a pure stacker. It has no compositor and no tiling mode; windows overlap freely, and the only data structure the manager needs is a `topped` counter per window—the most recently raised window wins at any pixel where multiple overlap. Section 2.1.3 already noted that nothing in `rio`'s architecture would prevent a compositor being added later; for now, every window draws directly onto the display server, and `wpointto()` picks the owner of a given pixel by scanning the `windows` array for the highest `topped` value.

2.1.5 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 X Window's innovation (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, which is why Linux has more than fifty of them and why the "which window manager?" question is a perennial topic in the UNIX community.

In X Window, the window manager is a separate program. 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 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).

2.1.6 Input focus

The previous subsection 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 four 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 X11 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 X11 and is still used by many tiling window managers (i3, 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 X11 behaviour and is rarely used today because it is hard to keep the pointer inside the window you wanted.

- Compositor-controlled focus with activation tokens. Wayland adds a layer on top of click-to-focus: a client cannot "raise and focus itself" the way an X11 client could; it has to present an activation token the compositor issued when the user performed some deliberate action (clicking a link, opening a file). This is why on Wayland a background program calling "show my window" from a script does not steal focus the way its X11 equivalent did. The goal is focus-stealing prevention: the user, not the program, decides where keystrokes go.

`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` 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. There is no persistent focus policy menu, no sloppy-focus option, no activation tokens, no focus-stealing prevention—just one global pointer and one rule for updating it. Chapter 7's "Window focus" section shows the actual code path: `wpointto()` finds the window under the click, `wtop()` raises it via the graphics library's `topwindow()`, and `wcurrent()` assigns the new `input` and repaints the border colors so the user can see which window is focused.

2.1.7 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`).

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 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).

2.1.8 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 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 13.3).

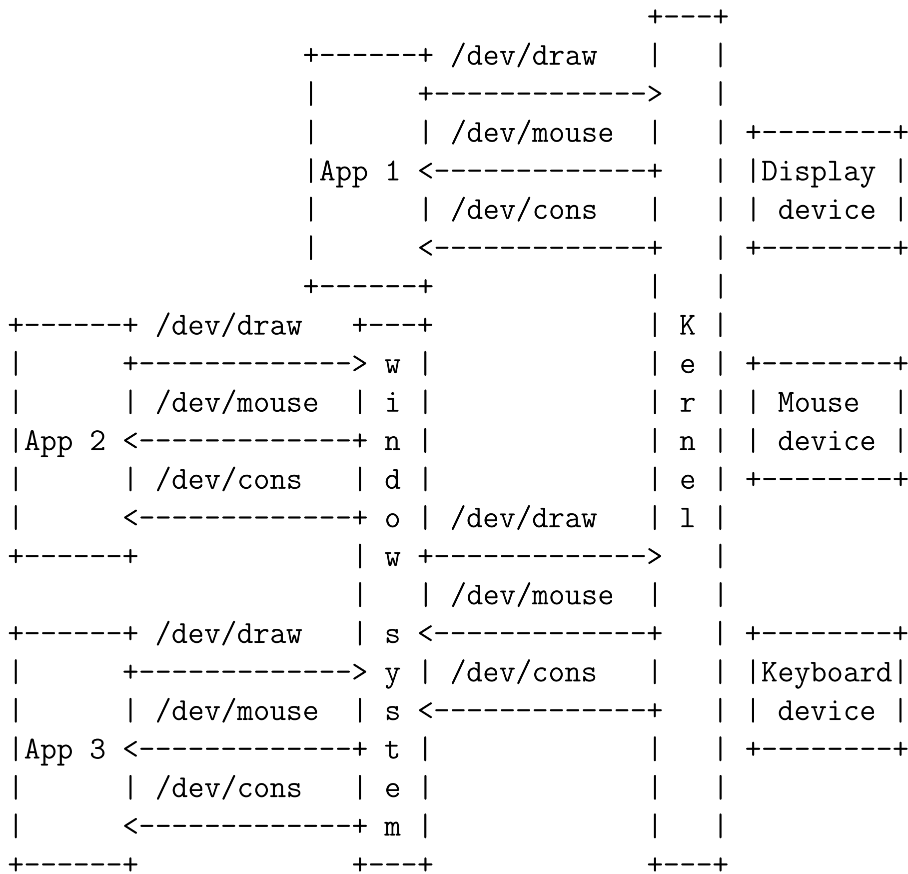


Figure 2.3: Windowing system as a multiplexer.

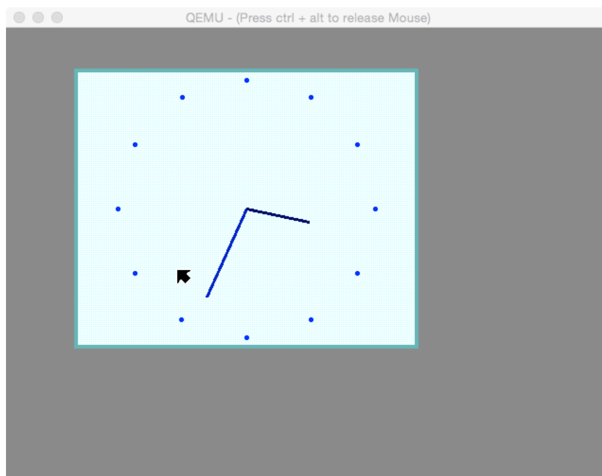


Figure 2.4: `clock` running inside `rio`.

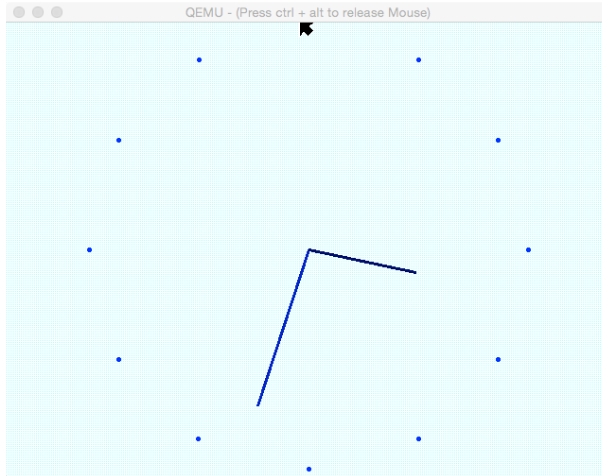


Figure 2.5: `clock` running without `rio`.

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).

2.1.9 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 2.5.2, 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.

`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:

⟨*function* `usage` 27⟩≡ (305a)

```
void
usage(void)
{
    fprint(STDERR, "usage: rio [-f font] [-i initcmd] [-k kbdcmd] [-s]\n");
    exits("usage");
}
```

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

[5]Unless it is run recursively inside one of its window, as shown in Section 13.3.

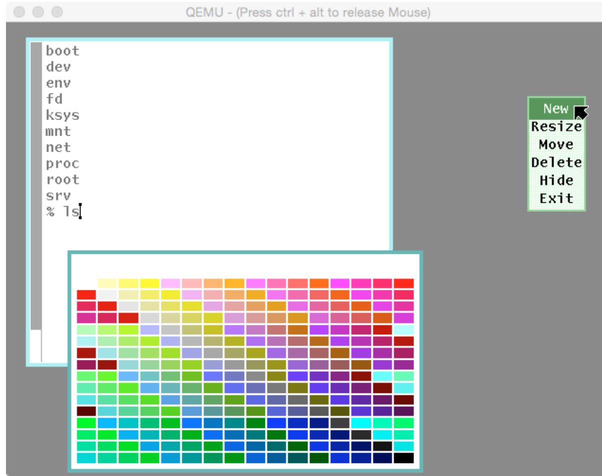


Figure 2.6: Graphical user interface of `rio` after a right-click.

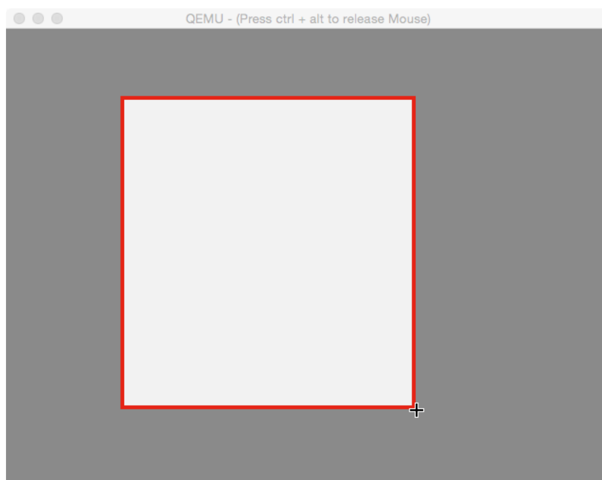


Figure 2.7: Creating a new window.

`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.

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`.

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

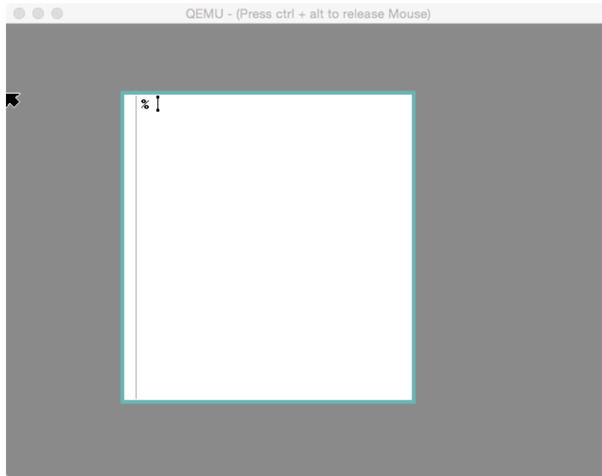


Figure 2.8: Built-in terminal running under `rio`.

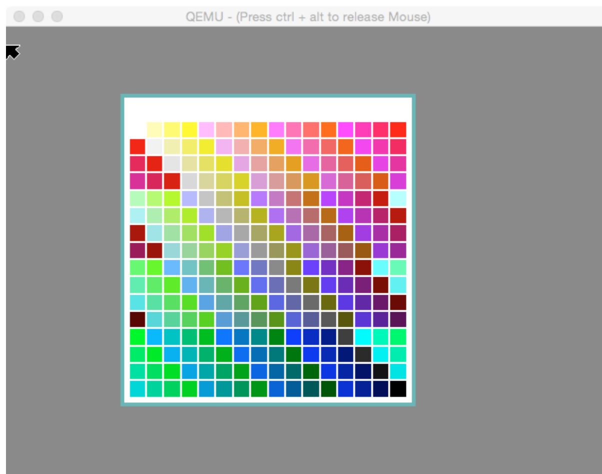


Figure 2.9: `colors` graphical application running under `rio`.

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 2.1.8, `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 2.1.8. 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.

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 `8-1/2` [Pik91], the ancestor of `rio`, was implemented. Under `8-1/2`, each window was seeing a `/mnt/wsys/draw` virtual device file in which the process could write drawing commands. `8-1/2` 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 2.1.9 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()`. `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 2.5.5 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 2.1.9, 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 2.1.9, `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`, 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. 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. `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.

[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.

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. 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[8]. Note that `rio` itself calls `initmouse()` at startup time, and uses heavily channels and threads, as you will see in the rest of the book.

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.

`/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 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 kernel, 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 kernel 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.

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

[8] 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.

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 kernel 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, as you will see in Chapter 4, `rio` itself calls `initkeyboard()` at startup time.

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 13.4.1).

- `/mnt/wsys/wdir`: This file is used for filename completion (see Section 13.4.3).

- `/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

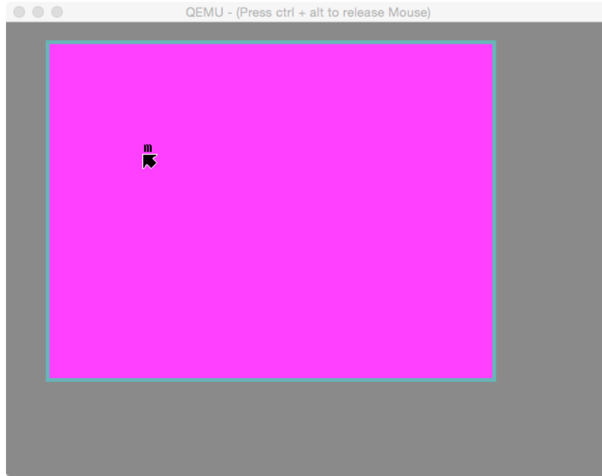


Figure 2.10: `hellorio` running in a window, after the user typed `'m'`.

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 12.5 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.

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 2.1.8, 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`:

```
⟨windows/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];

  ⟨type EventType (hellorio.c) 37a⟩

  void redraw(void);

  void threadmain(int argc, char* argv[]) {
    int result;
    Keyboardctl* keyboardctl;
    Mousectl*    mousectl;
    ⟨threadmain() other locals (hellorio.c) 37b⟩

    result = initdraw(nil, nil, "Hello Rio");
    ⟨threadmain() sanity check result (hellorio.c) 36a⟩
    mousectl = initmouse(nil, view);
    ⟨threadmain() sanity check mousectl (hellorio.c) 36b⟩
    keyboardctl = initkeyboard(nil);
    ⟨threadmain() sanity check keyboardctl (hellorio.c) 36c⟩

    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) 37c⟩
    redraw();
    ⟨threadmain() event loop (hellorio.c) 38d⟩
  }

  void redraw(void)
  {
    draw(view, view->r, bgcolor, nil, ZP);
    runestring(view, mouseloc, display->black, ZP, font, str);
    flushimage(display, true);
  }
```

Uses `bgcolor` 35, `mouseloc` 35, `redraw()` 35, and `str` 35.

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)* 36a⟩≡ (35)
```
  if (result < 0)
    exits("Error in initdraw");
```

⟨`threadmain()` *sanity check mousectl (hellorio.c)* 36b⟩≡ (35)
```
  if(mousectl == nil)
    exits("can't find mouse");
```

⟨`threadmain()` *sanity check keyboardctl (hellorio.c)* 36c⟩≡ (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)* 37a⟩≡ (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)* 37b⟩≡ (35) 38a ▷

```
// map<EventType, Alt>
Alt alts[NALT+1];
```

Uses `NALT`-63 37a.

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)* 37c⟩≡ (35) 38c ▷

```
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;
```

Uses `EKey`-61 37a, `EMouse`-60 37a, and `EResize`-62 37a.

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])[9]. 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)* 38a⟩+≡ (35) ◁37b 38b▷

```
Rune keys[20];
```

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

⟨`threadmain()` *other locals (hellorio.c)* 38b⟩+≡ (35) ◁38a ??▷

```
Mouse mouse;
```

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

⟨`threadmain()` *alts setup (hellorio.c)* 38c⟩+≡ (35) ◁37c

```
alts[NALT].op = CHANEND;
```

Uses `NALT-63` 37a.

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)* 38d⟩≡ (35)

```
for(;;) {
  switch(alt(alts)){
  ⟨threadmain() event loop cases (hellorio.c) 38e⟩
  }
  redraw();
}
```

Uses `redraw()` 35.

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)* 38e⟩≡ (38d) 39a▷

```
case EMouse:
  mouseloc = mouse.xy;
  break;
```

Uses `EMouse-60` 37a and `mouseloc` 35.

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

For the keyboard, the program modifies the global `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

[9] 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 can return the value directly from a receive.

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)* 39a⟩+≡ (38d) ◁38e 39b▷
```
  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;
```
Uses `EKey-61` 37a and `str` 35.

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)* 39b⟩+≡ (38d) ◁39a
```
  case EResize:
    if(getwindow(display, Refnone) < 0)
      exits("failed to re-attach window");
    break;
```
Uses `EResize-62` 37a.

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 Section 11.3. 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 13.4.2). 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 13.4.3).

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].

Function	Ch.	File	Entities	LOC
data structures	3	`dat.h`	`Window`[59] `Filsys`[62g] `Fid`[63c] `Xfid`[64c]	558
globals	3	`globals.c`	`mousectl`[57b] `windows`[61a] `input`[61e] `filsys`[62h]	108
entry point	4	`rio.c`	`threadmain()`[35]	264
prototypes	4	`fns.h`		135
keyboard thread	5	`thread_keyboard.c`	`keyboardthread()`[71]	47
mouse thread	5	`thread_mouse.c`	`mousethread()`[72c]	324
window threads	5	`threads_window.c`	`winctl()`[78]	508
fileserver proc	5	`proc_fileserver.c`	`filsysproc()`[81c] `filsysmount()`[102e]	253
master and worker threads	5	`threads_worker.c`	`xfidctl()`[85c]	176
cursor graphics	6	`data.c`	`crosscursor`[86] `corners`[88a]	213
cursor operations	6	`cursor.c`	`riosetcursor()`[90b]	31
window manager	7	`wm.c`	`button3menu()`[93b] `new()`[97b] `drag()`[113b]	756
window process creation	7	`processes_winshell.c`	`winshell()`[101b]	87
window methods	7	`wind.c`	`wtop()`[106d] `wclose()`[111a]	587
closing threads	7	`threads_misc.c`	`deletethread()`[109e] `winclosethread()`[110d]	58
fileserver utilities	8	`9p.c`	`filsysrespond()`[124a] `filsyscancel()`[283b]	71
fileserver methods	8	`fsys.c`	`filsysattach()`[125] `filsysopen()`[130c]	651
virtual device methods	9	`xfid.c`	`xfidopen()`[131d] `xfidread()`[134a]	926
graphical window	10	`graphical_window.c`	`waddraw()`[153c]	26
terminal emulator	11	`terminal.c`	`winsert()`[175b] `wshow()`[190] `wkeyctl()`[79e]	1236
terminal scrolling	11	`scrl.c`	`wscrdraw()`[191b] `wscroll()`[218c]	213
external window control	13	`wctl.c`	`wctlthread()`[237a] `parsewctl()`[241b]	538
copy/paste clipboard	13	`snarf.c`	`putsnarf()`[250g] `getsnarf()`[250h]	81
timer	13	`time.c`	`timerstart()`[281f]	144
error management	B	`error.c`	`error()`[292c] `derror()`[292d]	34
utilities	C	`util.c`	`min()`[293a] `emalloc()`[293d] `strrune()`[295a]	161
Total				8186

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

2.5 Software architecture

`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 2.1.9 for an introduction to layers). Many of the window management tasks are actually done by `draw`, not `rio`. For instance, it is the `draw` 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]).

I will present in the next section the thread (and process) architecture of `rio`. A good way to understand how those threads work together is to *trace* a user action through those different threads. This is why in the following sections, I will also explain the trace resulting from the creation of a new window by the user, the trace of a mouse click and key press in this window, and the trace of a call to a drawing function by the program running in this window.

2.5.1 Processes, procs, and threads relationships

The figure below 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 above 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 the diagram, then the shell and `hellorio` processes to the boxes outside `rio`'s address space.

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.

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:

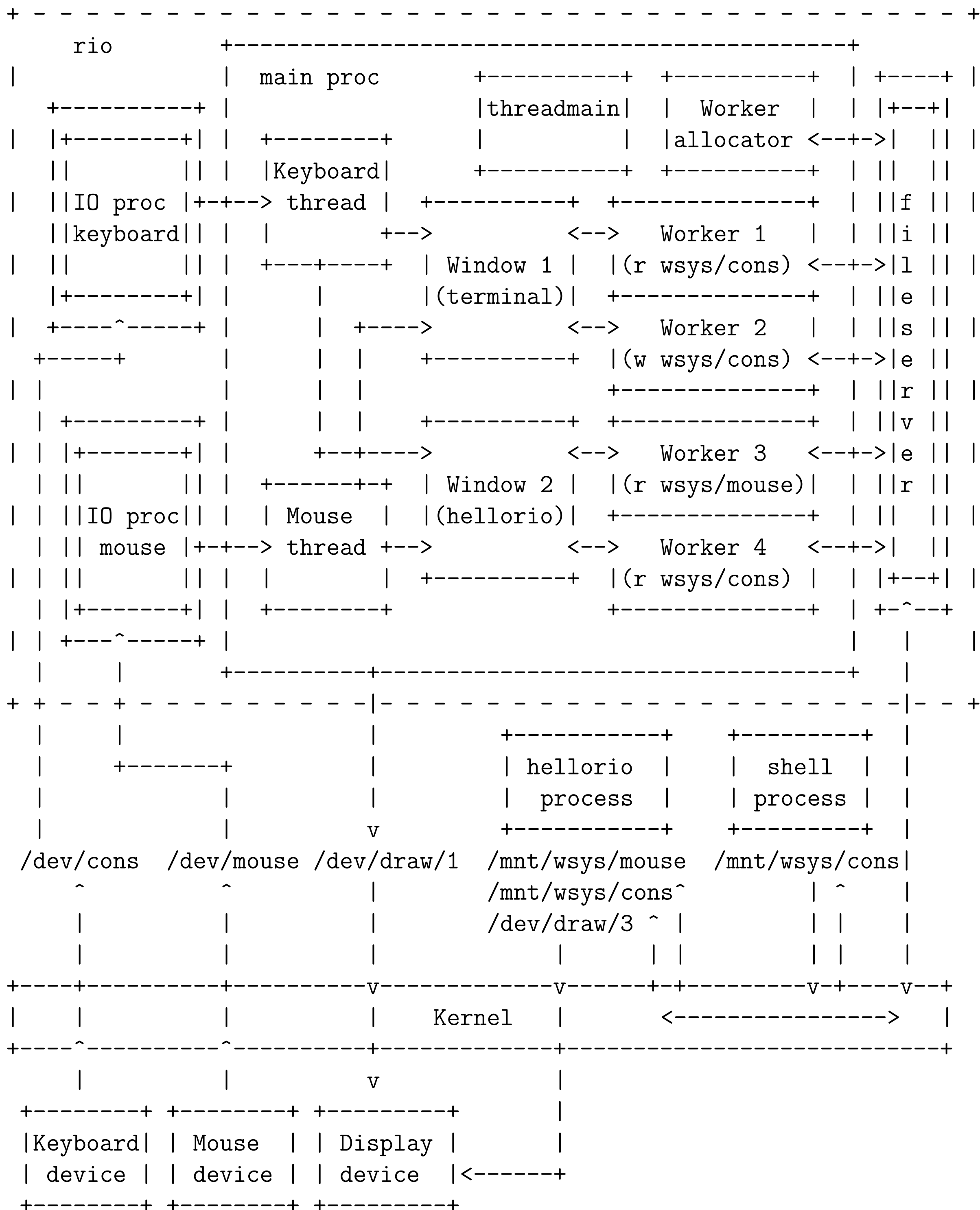


Figure 2.11: Processes, procs, and threads in `rio`.

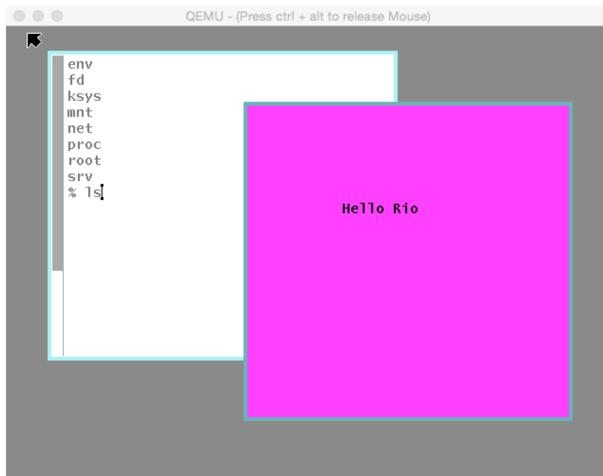


Figure 2.12: `hellorio` and a shell running in two different windows.

- 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

I mentioned already the first two procs in Section 2.3 in the context of the `hellorio` program[10]. 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.

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 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 one thread of a proc is running at one time (but multiple procs can run in parallel).

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 2.5.2 will give more details about this pipe and how `rio` and its clients communicate.

[10] 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`.

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, 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.* The next sections will clarify how those threads work together.

2.5.2 Trace of a new window creation

In Figure 2.11, threadmain, the first thread of main-proc, creates the two IO procs on the left, the fileserver proc on the right, and the keyboard and mouse threads in the middle. In this section, I will explain the trace leading to the creation of the shell and hellorio processes in Figure 2.11, as well as the creation of the associated window threads. In the next section, I will explain the creation of the worker threads.

As I said in Section 2.2.2, you create a new window by first activating the system menu with a right-click, then choosing New in this menu, and finally by drawing a rectangle with the mouse specifying the dimension of a window. The trace for those mouse actions will be explained soon in Section 2.5.3, but for now, the important thing is that eventually the mouse thread of main-proc will be in control, with the information about the new window dimension stored in a local variable. It is the mouse thread that creates windows. Each window creation involves the creation of a new process, a new namespace, a new Window structure, a new window thread, and a new image layer, as explained in the following sections.

A new process

When you create a new window, rio internally (1) forks a new process (with proccreate()), (2) adjusts the environment in the child process, and finally (3) executes (with procexec()) a command (the Plan 9 shell /bin/rc) from this child process. This is why the creation of the two windows in Figure 2.12 leads to the creation of two processes in Figure 2.11: hellorio and the shell process.

The sequence of steps above is a classic idiom in UNIX programming. Indeed, this is for instance the way the shell implements redirections, as in ls > list.txt. In that example, the shell adjusts the environment by changing the standard output file descriptor in the child process to point to a text file instead of the /dev/cons console device file (see the SHELL book [Pad18]). For rio, the adjust-the-environment part consists in changing the namespace in the child process by using the mount() and bind() system calls.

A new namespace

A *namespace* is the Plan 9 equivalent of a mount-table in UNIX (and its derivatives such as Linux or macOS). A *mount-table* allows to plug together multiple filesystems in a single uniform tree. However, in Plan 9, each process has its own mount-table, its own namespace; there is not a global mount-table as in UNIX. After the call to mount() mentioned above, the child process of rio will see special files in /mnt/wsys/, thanks to the per-process namespace feature of the Plan 9 kernel. Moreover, because of the bind() call, the same files will also be accessible under /dev/, and will take precedence over any existing files there, thanks to the union-mount feature of the Plan 9 kernel.

Before showing the actual calls to bind() and mount() in rio, here are the slightly simplified signatures of bind() and mount() from syscall.h:

⟨*simplified signatures of namespace functions* syscall.h 44⟩≡
```
error bind(char *str,   char *path, int flag);
error mount(fdt fd, ..., char *path, int flag, char *extra);
```

bind() takes as a parameter a string (str) and a path (path). If the string starts with a #, as in bind("#m", "/dev/", ...), the string must correspond to the code of a device in the Plan 9 kernel, for instance, #m is the code of the mouse device. In that case, every access from the process to files under path will be performed with

the methods of the corresponding device, for instance, `mousewalk()` if the process was listing files in `/dev/`. Remember that in Plan 9, devices are filesystems (see the KERNEL book [Pad14]). The mouse device, for instance, manages more than one file (e.g., `/dev/mouse`, `/dev/cursor`). Some devices such as `draw` manage even multiple directories (e.g., `/dev/draw/1/`, `/dev/draw/2/`). If the string did not start with a `#`, every access to files under path are redirected to the path denoted by the string. In effect, it is like `path` becomes an *alias* for the path denoted by `str`.

With `mount()`, the kernel delegates the management of a directory neither to a device in the kernel, nor to another directory, but to a userspace program. Indeed, `mount()` takes as a parameter a file descriptor (`fd`) and a path (`path`), in order to tell the kernel that every access to files under path should be handled by the process behind the file descriptor (see the KERNEL book [Pad14] and the NETWORK book [Pad16d]). This allows to implement *filesystems in user space*, a powerful and convenient feature. Note that the file descriptor argument can be anything; it can correspond to a pipe to a local process, but also to an opened connection on the network. Thus, filesystems can also be imported (and exported) through the network in Plan 9.

Both `bind()` and `mount()` operate on an existing path and alter the filesystem tree at this path. Moreover, in Plan 9, you can not only plug a filesystem on top of another filesystem at a specific directory, you can also join filesystems in a directory. Indeed, a directory can be the union of two or more filesystems. Such a directory is called a *union-mount*. Both `bind()` and `mount()` take a flag parameter specifying how to handle the union of the old content of the directory with the new content of the `bind()`'ed device (or directory) or the new `mount()`'ed user-space filesystem. For instance, the flag argument `MREPL` (for replace) indicates that the old content should be ignored and replaced by the new content. The flag argument `MAFTER` indicates that the old content should be kept and should take precedence over the new content in case of name conflicts. Indeed, two filesystems can contain files with identical names. Thanks to the union-mount and a series of calls to `bind()`, the `/dev` directory can contain many files and directories managed by multiple devices.

Here are the slightly simplified calls to `mount()` and `bind()` in the function `filsysmount()`[102e] called in the child process of `rio` when a new window is created:

⟨`filsysmount()` *simplified code* 45⟩≡
```
  err = mount(fs->cfd, ..., "/mnt/wsys", MREPL, windowid);
  ...
  err = bind("/mnt/wsys", "/dev", MBEFORE);
```

With the call to `bind()`, `/dev/` becomes the union of the old `/dev/` directory and the `/mnt/wsys` directory. The flag argument `MBEFORE` indicates that files under `/mnt/wsys` takes precedence over the old files under `/dev/`. This means than an access to `/dev/mouse` by the process who executed the code above (or the children of this process) will be redirected to an access to `/mnt/wsys/mouse`, and not to the old `/dev/mouse` file that was managed by the kernel (and which was there because of a previous call to `bind("#m", "/dev", ...)`). This is why in Figure 2.11, the `hellorio` and shell processes, at the bottom right, are connected to files starting with `/mnt/wsys/`, while `rio` itself is connected to files starting with `/dev/`, at the bottom left.

Regarding `mount()`, the file descriptor argument corresponds to one side of a pipe created by the `fileserver` proc (see Section 2.5.1) and stored in the `cfd` (for client file descriptor) field of the global `fsX` (for filesystem). The other side of the pipe is stored in the `sfd` (for server file descriptor) field of `fs`. Note that file descriptors and namespaces are inherited by default through `rfork()` (or `proccreate()`), and maintained through `exec()` (or `procexec()`). Thus, after the child process of `rio` has called `filsysmount()` and `exec()`'ed the shell `/bin/rc`, this shell process will still have in its file descriptor table in the kernel a link to the pipe created by the `fileserver` proc, and in its mount-table the information about `/mnt/wsys/`. This means that an access from this shell process to `/mnt/wsys/mouse` will be converted by the kernel to a 9P request written in the pipe (see the NETWORK book [Pad16d]). On the other side of the pipe, the `fileserver` proc will read the pipe, decode the 9P request, and return a result to the kernel through the pipe (e.g., an error code).

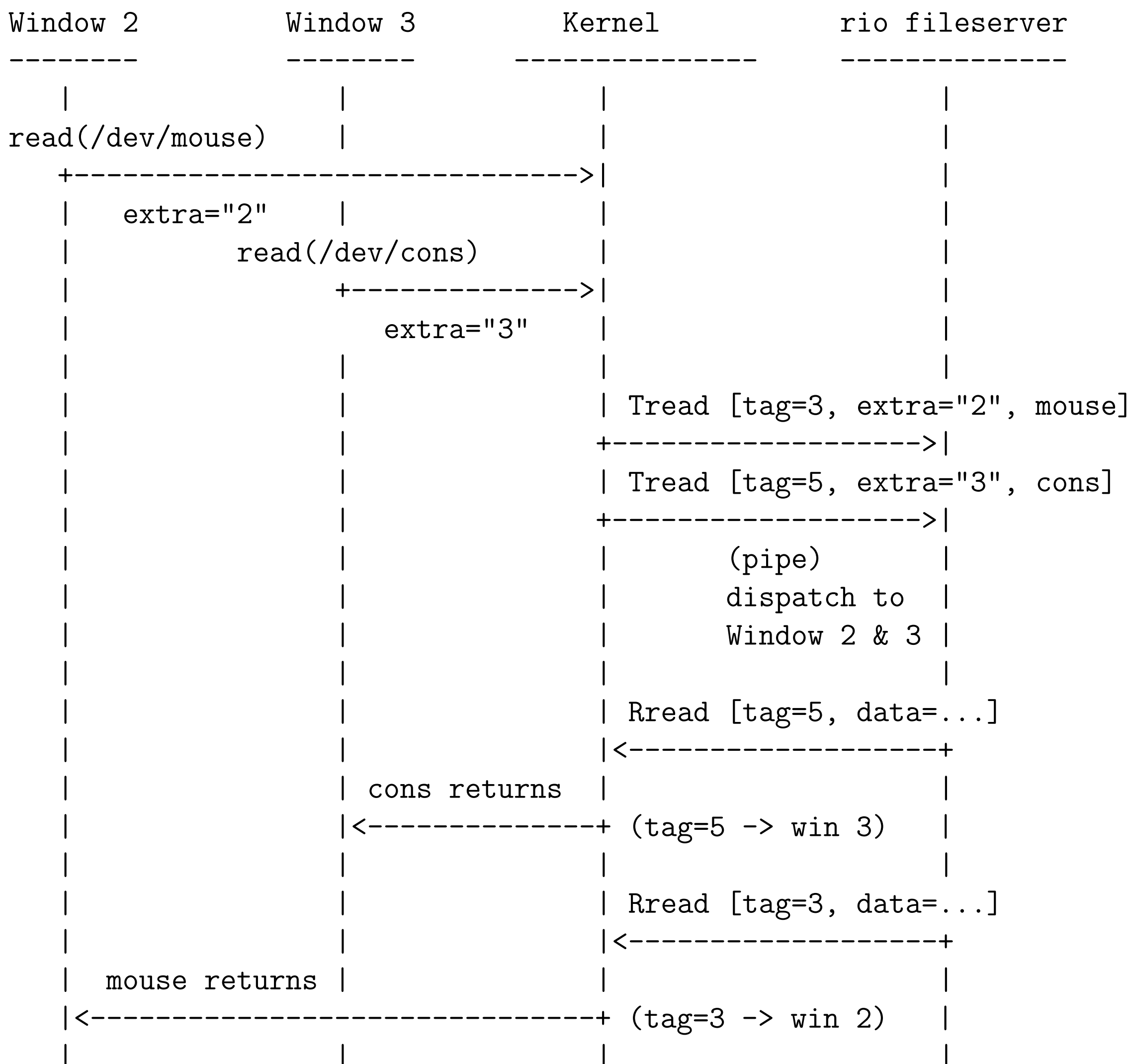
How can a single pipe serve multiple clients without message corruption? Two things make this work. First, Plan 9 pipes guarantee atomic writes up to 64 KB (`Maxatomic` in the kernel's queue implementation), and 9P messages are well within this limit, so a complete 9P message from one client is never interleaved with bytes from another. Second, the kernel itself manages the multiplexing: when a client process does a `read()` on

`/mnt/wsys/mouse`, the kernel translates it into a 9P `Tread` message and writes it atomically into the pipe. Each 9P message carries a unique *tag*, so when the `fileserver` writes back a response, the kernel matches the tag to know which client process to wake up.

Thus, it is through a pipe that `rio` and its clients are communicating. This communication is indirect. Indeed, the clients are not writing directly into the pipe; they are just performing system calls on (virtual device) files, as indicated for example by the arrows between the `hellorio` process and the kernel in Figure 2.11. However, the kernel redirects those system calls to 9P requests on the pipe created initially by the `fileserver` proc and inherited by those clients. The arrow connecting the `fileserver` proc to the kernel on the right of Figure 2.11 is the pipe.

Note that `mount()` can take an extra argument (called `extra` in the signature of `mount()` above). For `rio`, this argument contains the window identifier (in the `windowid` local variable in the code above) of the newly created window. This extra argument is stored in the process mount-table and passed by the kernel through the 9P request. Thus, the `fileserver` proc can know from which window (and so from which process) a file request comes from. This is important because the `fileserver` proc reads from a pipe that is shared by many processes.

The following diagram summarizes the full indirect communication path for two different windows. Window 2 reads `/dev/mouse` and window 3 reads `/dev/cons`, both going through the same pipe. The `extra` field (the window identifier from `mount()`) tells `rio` which window each request belongs to, while the `tag` lets the kernel match responses to the right syscall:



Each 9P message carries two identifiers that serve different purposes. The *extra* field is a Plan 9-specific extension to `mount()`: it is a string (here, the window identifier) stored in the process mount-table and attached

by the kernel to the 9P attach message. It tells the `fileserver` which `Window` the request belongs to. The *tag* is a 9P protocol concept managed by the kernel: it is a small integer that uniquely identifies each outstanding request, so the kernel can match a response back to the specific `read()` system call that triggered it—even when multiple requests from different windows are in flight on the same pipe. In short, `extra` tells `rio` *which window*, while the tag tells the kernel *which syscall* to complete. The client processes are unaware of both—they just call `read()` on what they think is `/dev/mouse` or `/dev/cons`. The kernel translates each system call into a 9P message and writes it atomically into the pipe (Plan 9 pipes guarantee atomic writes up to 64 KB, well above any 9P message size).

A new `Window`

Now that the process for the new window has been created, and its namespace adjusted, `rio` needs to keep track of the new window (and the new process). To do so, `rio` allocates a new `Window`[59] structure, which gathers information about a window. `rio` also adds this structure in the global `windows`[61a], which contains the list of all `Windows` (see Section 3.3 for more information). The `Window` structure has many fields. For instance, `Window.id` contains the window identifier, while `Window.pid` contains the process identifier of the process associated with the window. The other fields will be presented gradually in this document.

A new thread

Each window must deal with inputs from the mouse or the keyboard. Those inputs, which correspond to two independent activities of the user, are managed in `rio` by two independent threads in the left of Figure 2.11: the mouse and keyboard threads (and associated IO procs). Those inputs must then find a way to the right of Figure 2.11 until the `fileserver` proc, which is connected to the window processes (via a pipe). To separate concerns, the mouse and keyboard threads do not communicate directly with the `fileserver` proc. Instead, they communicate with *window threads* (e.g., `Window 1` and `Window 2` in the middle of Figure 2.11). Those threads handle the independent input activities to the independent windows. *Each window is represented by a thread.*

Thus, when you create a new window, `rio` creates internally a new window thread. Moreover, `rio` creates multiple channels to communicate with this new thread. For instance, `Window.mch` contains a channel used to exchange messages between the mouse thread and the window thread.

When you move the mouse, the event is transmitted at some point to the mouse thread of `rio`, which gets the information by listening on `mousectl->c` (this is similar to what `hellorio` did in Section 2.3.2). The mouse thread then looks for the window with the focus, which is stored in the global `currentX` (see Section 3.3.3), a pointer to a `Window`[59] in `windows`[61a]. It then writes information about the mouse event in `current->mch`. On the other side of the channel, a single window thread is listening on this channel and is ready to receive the mouse input for further processing (for instance, by relaying the event to a worker thread, as explained soon in Section 2.5.3).

Because the window with the focus can change, the mouse and keyboard threads can communicate with different window threads. This is why in Figure 2.11 the mouse and keyboard threads are connected to all the window threads.

A new layer

The last thing `rio` internally creates when you create a new window is a new image layer with the dimension of the window. This image is stored in the `Window.img` field of the window. *Each window is associated to one layer.* As I said in Section 2.1.9, an image layer is a convenient graphics construct that `rio` uses to implement overlapping windows. What we need now is a way to communicate this layer to the window application so that this application can draw in this new layer.

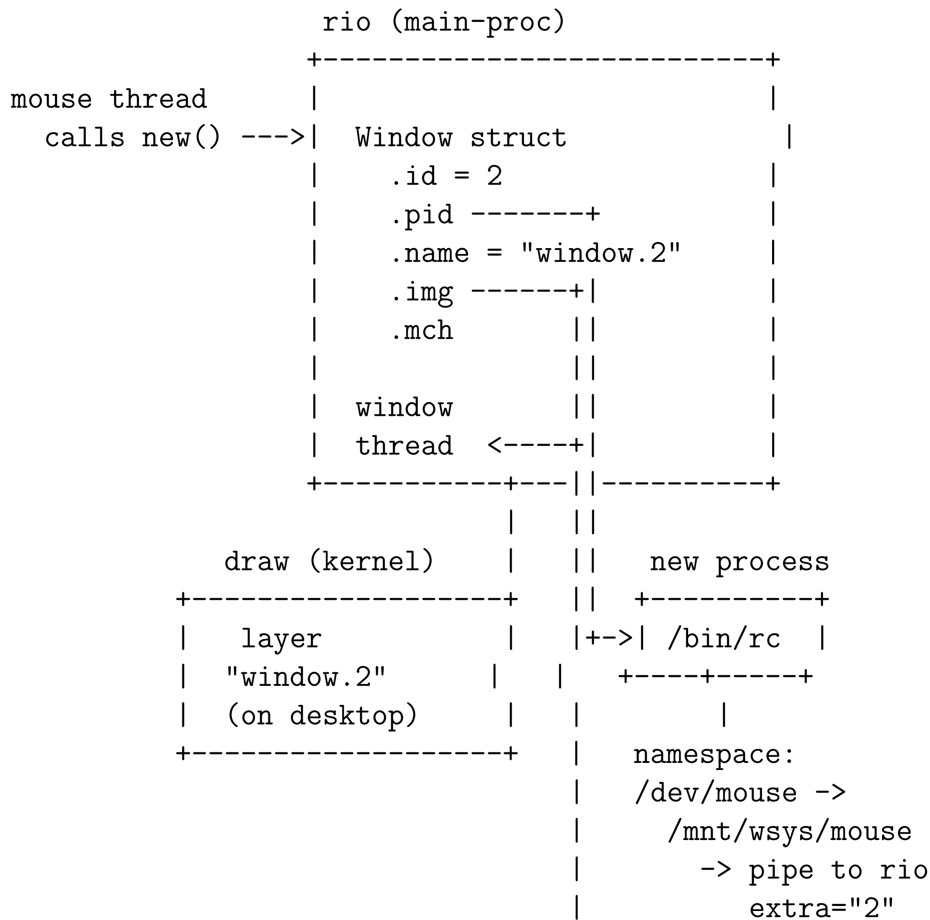
As mentioned in Section 2.2.3, each window application is connected directly to the `draw` device, as shown in Figure 2.11 with the `hellorio process` connected to `draw` via `/dev/draw/3/`. The drawing operations by the

window application do not go through a virtual `/mnt/wsys/draw` file that would be managed by `rio` (and its `fileserver` proc). However, it is `rio` that creates the layer and so knows where the window application should draw. Fortunately, `draw` has an IPC feature allowing multiple processes to share an image or an image layer. By giving a public name to the new image layer (with `nameimage()`), `rio` allows another process to retrieve a handle to the image (with `namedimage()`) if this process knows the name of the image. This unique name is stored in the `Window.name` field of the window. However, we still need a way to communicate this name to the window application. To do so, `rio` uses the same mechanism it uses to transmit any kind of information to its window: a file under `/mnt/wsys/`, in that case `/mnt/wsys/winname`, which contains the string in `Window.name`.

Note that each window application will read a different string from his `/mnt/wsys/winname` file. Indeed, each window process has a different namespace since a unique window identifier is passed to the `mount()` command (see Section 2.5.2). When a window application reads `/mnt/wsys/winname`, the request goes to the `fileserver` proc through a pipe, and `fileserver` knows which `Window.name` field to read since the window identifier is passed also in the request.

This concludes the trace of a new window creation. Now that `rio` has created all the necessary entities to support the new window, I can explain the trace of a mouse click in this new window in the next section.

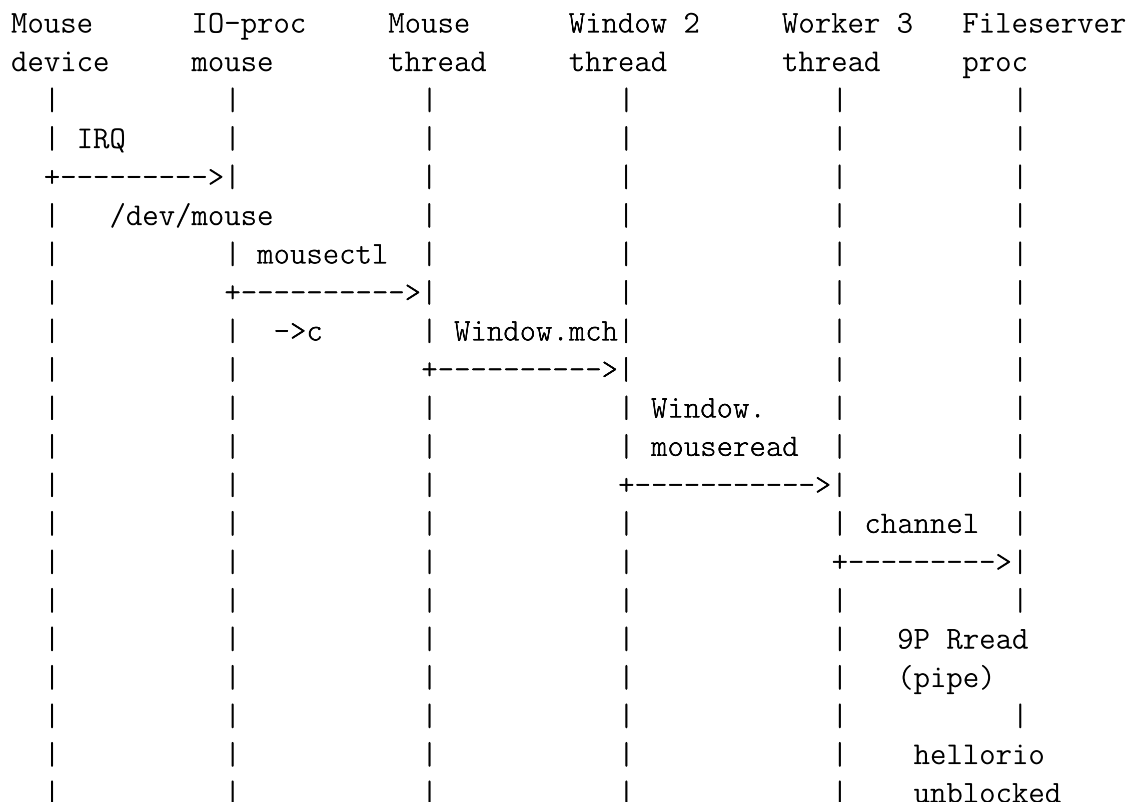
To summarize, creating a new window produces the following entities, all linked together:



2.5.3 Trace of a mouse click

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`. This trace will go through many elements of Figure 2.11. Moreover, in this section, I will also explain the creation of the last remaining threads of `main-proc`: the worker threads.

Here is the data flow of a mouse click from the hardware to `hellorio`, through the different threads of `rio`:



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 Section 2.5.2).

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 Section 2.5.2). 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. *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, as explained soon. Once `Worker 3` receives a mouse event, it can communicate back (by using another channel) the event to the `fileserver`

proc, which can write back the appropriate 9P answers into the pipe. 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`:

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 mouse thread, which was listening on `mousectl->c` (see Section 2.5.2). The mouse thread then looks for the window with the focus and relays the message on `wcurrent->mch`.

5. The `Window 2` thread, at the middle of Figure 2.11, which was listening on `Window.mch` (as well as other channels via `alt()`) receives the message on `Window.mch`. 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` field of the `Window` 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 relays it to the `fileserver` proc as explained above. This concludes the trace of mouse click.

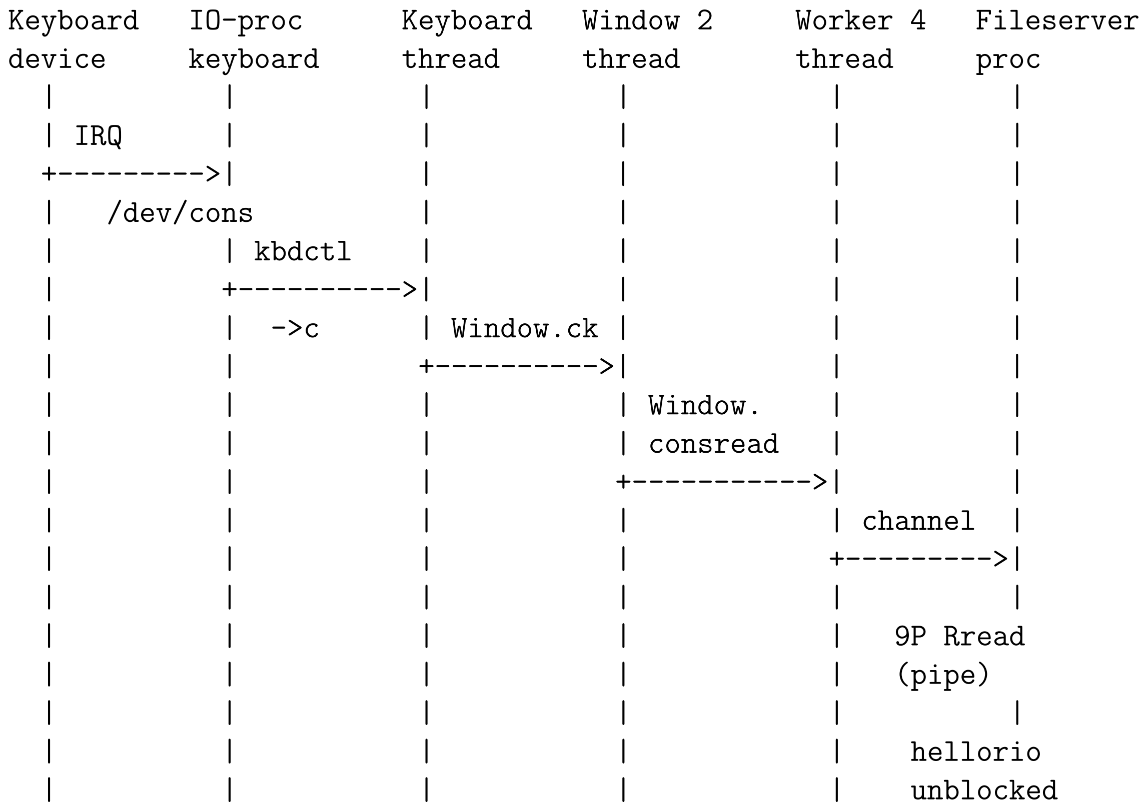
Note that if you click with the mouse outside any window, the mouse event does not go to any window thread. Instead, the event stops at the mouse thread, which contains code to process the event to possibly move, resize, hide, or create a new window. Indeed, as mentioned in Section 2.5.2, it is the mouse thread that creates windows and all their associated entities (processes, namespaces, layers, threads, and `Windows`).

2.5.4 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` channel, then to the worker thread `Worker 4` through a `Window.consread` 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:



For a textual window (e.g., the shell), the path is more complex because the window thread also handles line editing, echoing, and scrolling before the data reaches the worker—this will be explained in Chapter 11.

2.5.5 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 2.5.3 and Section 2.5.4. 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 2.1.9)
- `draw` is using a client/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 2.5.2)

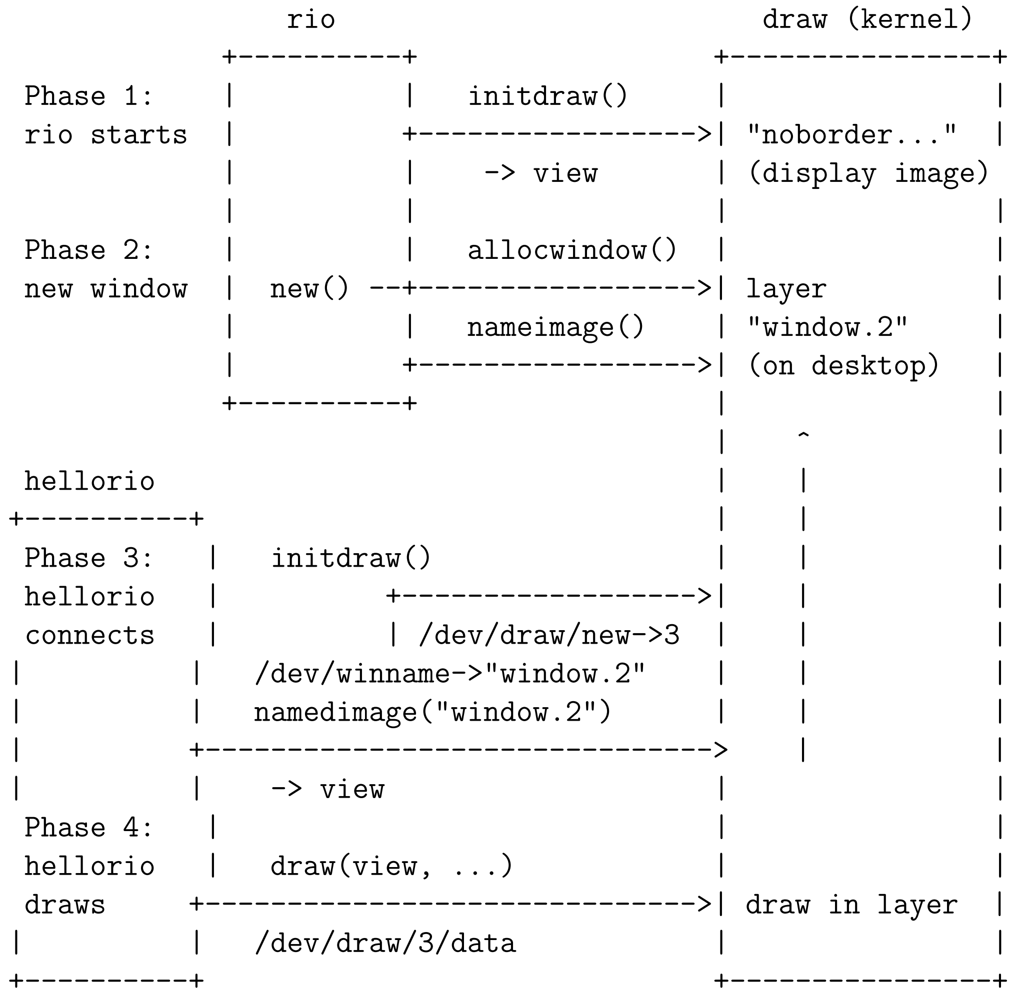
The goal of this section is to put together all those pieces of information to better understand how a window application draws on the screen.

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/`[11].

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]).

[11] 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 13.3).

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.

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`[59] (stored in the global array `windows`[61a]), and a thread (see Section 2.5.2). `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.img 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 field. rio then calls nameimage() with Window.img 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.img. It can translate text output from the shell process through /mnt/wsys/cons to calls to drawing functions taking Window.img 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[59] associated with hellorio. This Window is stored in the global array windows[61a]. 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 2.5.2). 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.img` 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.

This concludes the trace of a drawing operation, as well as the presentation of the software architecture of `rio`. The traces above are simplified, but the essential elements are there. For a more precise trace of a drawing operation, see the Graphics book [Pad16c]. For more details about the other traces, read the following chapters.

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 D 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 (like any graphical application); the `Window` structure, the central data structure of a windowing system, grouping graphics, input, process, and text information for each window; and the `Filsys` and `Fid` structures that support `rio`'s role as a filesystem server, maintaining per-client file state for the 9P protocol.

3.1 Device handles

3.1.1 Output device: `display` and `view`

Like any graphical application under Plan 9, `rio` uses the global `display` (set by `initdraw()`) 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 `draw` library and are declared in `draw.h`, but `viewr` below is defined in `rio` and is essentially an alias for `view->r`.

⟨*global* `viewr` 57a⟩≡ (305b)

```
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.

⟨*global* `mousectl` 57b⟩≡ (304)

```
Mousectl *mousectl;
```

⟨*global* `keyboardctl` 57c⟩≡ (304)

```
Keyboardctl *keyboardctl;
```

⟨*global* `mouse` 58a⟩≡ (304)
```
// alias for &mousectl->Mouse
Mouse *mouse;
```

3.2 Desktop, `desktop`

A `Screen` (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. The whole graphics stack that `rio` assembles at startup is a small three-level tree, which I find easier to picture before the window fields arrive:

```
view                        (/dev/winname, the physical framebuffer)
 |
 +-- background              (solid color, fills view)
 |
 +-- desktop : Screen        (overlapping-layers manager on top of view)
      |
      +-- windows[0]->i      (w0 Image, client draws here)
      +-- windows[1]->i      (w1 Image, possibly overlapping w0)
      +-- windows[2]->i      (w2 Image, topmost if w2->topped is max)
      +-- ...
```

`view` is the only thing that actually corresponds to framebuffer pixels; everything below it is a window-managed region whose repaints are serialized through `desktop`. When two windows overlap, libdraw's `Screen` takes care of clipping one against the other—so `rio`'s own code never needs an explicit clipping-region pass. This is the single biggest reason `rio`'s C is so short compared to an X11 window manager: the hard part of the job sits in GRAPHICS book [Pad16c], not here.

⟨*global* `desktop` 58b⟩≡ (304)
```
Screen *desktop;
```

The `background` and `red` globals are color images (in Plan 9, everything is an `Image`, including colors). The red color is used to highlight window borders during move/resize operations.

⟨*global* `background` 58c⟩≡ (304)
```
Image *background;
```

⟨*global* `red` 58d⟩≡ (304)
```
Image *red;
```

3.3 Windows

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

3.3.1 `Window`

The `Window` 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 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 59⟩≡ (299b)
```
struct Window
{
    //--------------------------------------------------------------------------
    // ID
    //--------------------------------------------------------------------------
    ⟨Window id fields 60a⟩

    //--------------------------------------------------------------------------
    // Graphics
    //--------------------------------------------------------------------------
    ⟨Window graphics fields 60c⟩

    //--------------------------------------------------------------------------
    // Mouse
    //--------------------------------------------------------------------------
    ⟨Window mouse fields 80a⟩

    //--------------------------------------------------------------------------
    // Keyboard
    //--------------------------------------------------------------------------
    ⟨Window keyboard fields 79a⟩

    //--------------------------------------------------------------------------
    // Control
    //--------------------------------------------------------------------------
    ⟨Window control fields 80e⟩

    //--------------------------------------------------------------------------
    // Process
    //--------------------------------------------------------------------------
    ⟨Window process fields 102a⟩

    //--------------------------------------------------------------------------
    // Config
    //--------------------------------------------------------------------------
    ⟨Window config fields 62f⟩

    //--------------------------------------------------------------------------
    // Textual Window
    //--------------------------------------------------------------------------
    ⟨Window textual window fields 61g⟩

    //--------------------------------------------------------------------------
    // Graphical Window
    //--------------------------------------------------------------------------
    ⟨Window graphical window fields 61f⟩

    //--------------------------------------------------------------------------
    // Misc
    //--------------------------------------------------------------------------
    ⟨Window other fields 61d⟩

    //--------------------------------------------------------------------------
    // Extra
```

```
    //--------------------------------------------------------------------
    ⟨Window extra fields 60f⟩
};
```

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* 60a⟩≡ (59) 60e ▷

```
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, incremented in `wmk()`[98e]. This ID appears in the filesystem path `/mnt/wsys/<id>/` and identifies the window across the 9P protocol.

⟨*global* `id` 60b⟩≡ (311)

```
static int id;
```

The `i` field 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* 60c⟩≡ (59) 60d ▷

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

The `screenr` field records the window's physical position on screen. The application never sees this value—it works entirely in logical coordinates. When `rio` used `originwindow()`, a window's image coordinates (`i->r`) could start at (0,0), so `hellorio` might see `i->r = (0,0, 400,300)` while the window actually sits at `screenr = (100,100, 500,400)` on screen. The application draws into `i` using logical coordinates, and the draw device maps them to the physical screen. Likewise, `rio` converts mouse coordinates from physical to logical (as we saw in `mousethread()`[72c]) before forwarding them to the window, so the application receives coordinates consistent with its own `i->r`. Today `originwindow()` is no longer called, so `i->r` and `screenr` are always equal and the conversion is a no-op, but the code still maintains both fields.

⟨Window *graphics fields* 60d⟩+≡ (59) ◁ 60c

```
/*
 * Rio once used originwindow, so screenr could be different from i->r.
 * Now they're always the same but the code doesn't assume so.
*/
Rectangle screenr; /* screen coordinates of window */
```

⟨Window *id fields* 60e⟩+≡ (59) ◁ 60a 103g ▷

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

Because multiple threads and procs access `Windows` concurrently—the mouse thread, the keyboard thread, the window's own `winctl` thread, and filesystem worker threads—`Window` embeds a `Ref` for reference counting and a `QLock` for mutual exclusion. The `Ref` prevents a window from being freed while another thread still holds a pointer to it—for instance, the mouse thread may be partway through a resize or delete operation on a window while a filesystem worker thread is simultaneously servicing a read on that same window's `/mnt/wsys/cons`. Without reference counting, the window could be deallocated under the worker's feet. The `QLock` serializes access to fields that can be modified from different threads (see LIBCORE book [Pad16a] for the details of these synchronization primitives).

⟨Window *extra fields* 60f⟩≡ (59) 60g ▷

```
Ref;
```

⟨Window *extra fields* 60g⟩+≡ (59) ◁ 60f

```
QLock;
```

3.3.2 `windows`

The global `windows` array is the essence of the windowing system: it holds all the managed windows. The `topped` counter is a logical timestamp—each time a window gets the focus, it receives the current `topped` value (then incremented). This allows `wpointto()`[76a] to find the topmost window at a given point by comparing `topped` values across all overlapping windows.

⟨*global* `window` 61a⟩≡ (304)
```
// growing_array<ref_own<Window>> (size = nwindow+1)
Window **windows;
```

⟨*global* `nwindow` 61b⟩≡ (304)
```
int nwindow;
```

⟨*global* `topped` 61c⟩≡ (311)
```
static int  topped;
```

⟨`Window` *other fields* 61d⟩≡ (59) 100b ▷
```
int   topped;
```

3.3.3 `current`

The `input` 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` 61e⟩≡ (304)
```
//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`, `mouseopen` 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* 61f⟩≡ (59) 153a ▷
```
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. The `Frame` structure (from the frame library) handles the rendering of visible text—wrapping, selection highlighting, and scrollbar interaction.

⟨`Window` *textual window fields* 61g⟩≡ (59)
```
⟨Window textual window fields, text data 61h⟩
⟨Window textual window fields, text cursors 62a⟩
⟨Window textual window fields, visible text 62b⟩
⟨Window textual window fields, graphics 62d⟩
```

⟨`Window` *textual window fields, text data* 61h⟩≡ (61g)
```
// growing_array<Rune> (size = Window.maxr)
Rune  *r;
uint  nr; /* number of runes in window */
uint  maxr; /* number of runes allocated in r */
```

⟨Window *textual window fields, text cursors* 62a⟩≡ (61g) 62c▷

```
// 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* 62b⟩≡ (61g)

```
uint  org;
```

⟨Window *textual window fields, text cursors* 62c⟩+≡ (61g) ◁62a

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

⟨Window *textual window fields, graphics* 62d⟩≡ (61g) 62e▷

```
Frame frm;
```

⟨Window *textual window fields, graphics* 62e⟩+≡ (61g) ◁62d

```
Rectangle scrollr;
```

⟨Window *config fields* 62f⟩≡ (59) 108c▷

```
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 FilSys and filsys

The `Filsys` structure holds the two ends of a pipe (`cfd` for the client side, `sfd` for the server side), the user name (for security checks), a hash table of `Fid` entries tracking per-client file state, and a channel to the worker allocator. The client side of the pipe is also published in `/srv/rio.user.pid` for external processes that want to connect to `rio`. Child processes do not need this `/srv` entry—they inherit the pipe's file descriptor directly through the fork/exec model of shared file descriptors.

⟨*struct* Filsys 62g⟩≡ (299b)

```
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 64b⟩
};
```

Uses Nhash 63a.

⟨*global* filsys 62h⟩≡ (304)

```
Filsys *filsys;
```

⟨*constant* Nhash 63a⟩≡ (299b)
```
#define Nhash 16
```

⟨Fid *extra fields* 63b⟩≡ (63c) 63e ▷
```
// 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). The `Fid.w` field closes the circle: once a fid is associated with a `Qid` (during attach or walk), `rio` can resolve it back to the corresponding `Window`. Note that the 9P "client" here is not the 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 `Tread` message sent through the pipe to `rio`'s `filsysproc`.

⟨*struct* Fid 63c⟩≡ (299b)
```
struct Fid
{
    // the key
    int  fid;

    // the state
    bool open;
    int  mode;

    ⟨Fid other fields 63d⟩

    // Extra
    ⟨Fid extra fields 63b⟩
};
```

⟨Fid *other fields* 63d⟩≡ (63c) 64a ▷
```
Qid  qid;
```

⟨Fid *extra fields* 63e⟩+≡ (63c) ◁ 63b
```
bool busy;
```

⟨*function* newfid 63f⟩≡ (317)
```
Fid*
newfid(Filsys *fs, int fid)
{
    Fid *f, *ff, **fh;

    ff = nil; // free fid
    fh = &fs->fids[fid&(Nhash-1)];

    // lookup_hash(fid, fs->fids)
    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 63a and emalloc() 293d.

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:

```
 client process              rio fileserver              window
 +-----------+              +----------+               +---------+
 | open      |  --9P msg--> | Fid      |     Fid.w --> | Window |
 | read      |              |  .fid    |               |  .id    |
 | write     |  <--9P msg-- |  .qid    |               |  .ck    |
 +-----------+              |  .w ----+---->            |  .mc    |
                             +----------+               +---------+
```

⟨`Fid` *other fields* 64a⟩+≡ (63c) ◁63d 123c▷

```
Window *w;
```

3.4.3 Workers and jobs: `Xfid`

The filesystem server proc (`filsysproc()`[81c]) 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 parsed request (`Fcall`) with a channel to a worker thread. The master sends a function pointer through `Xfid.c`, telling the worker what operation to execute. Workers are pooled via the `cxfidalloc`/`cxfidfree` channels to avoid creating a new thread for every request.

⟨`Filsys` *other fields* 64b⟩≡ (62g)

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

⟨*struct* `Xfid` 64c⟩≡ (299b)

```
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*)) */
```

```
        Fid *f;

        Filsys *fs;

        ⟨Xfid flushing fields 282a⟩
        ⟨Xfid other fields 282c⟩

        // Extra
        Ref;
        ⟨Xfid extra fields 84c⟩
};
```

3.4.4 9P callbacks: `fcall`

The `fcall` dispatch table 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()`[81c] dispatches through this table. The handler may return `nil` if it delegates work to a worker thread, or the `Xfid` back if the worker can be reused immediately.

⟨*global* `fcall` 65⟩≡ (317)

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

    [Twalk]    = filsyswalk,

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

    ⟨fcall other methods 83a⟩
};
```

Uses `filsysattach()` 125, `filsysclunk()` 132a, `filsysopen()` 130c, `filsysread()` 133a, `filsysstat()` 135b, `filsyswalk()` 127d, and `filsyswrite()` 134b.

Chapter 4

`main()`

The entry point of `rio` is `threadmain()` (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. The initialization 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`, waiting for the user to select "Exit" from the right-click menu.

⟨*function* `threadmain` 66a⟩≡ (305a)

```
void threadmain(int argc, char *argv[])
{
    ⟨main() locals 100d⟩

    ARGBEGIN{
    ⟨main() command line processing 272b⟩
    }ARGEND

    ⟨main() set some globals 100e⟩

    // Rio, a graphical application

    ⟨main() graphics initializations 67a⟩

    // Rio, a concurrent application

    ⟨main() communication channels creation 68c⟩
    ⟨main() threads creation 68e⟩

    // Rio, a filesystem server

    filsys = filsysinit(xfidinit());
    ⟨main() if filsys is nil 66b⟩
    else{
        ⟨main() error management after everything setup 292b⟩

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

Uses `exitchan` 68b, `filsys` 62h, `filsysinit()` 69a, `killprocs()` 279a, and `xfidinit()` 70c.

⟨`main()` *if filsys is nil* 66b⟩≡ (66a)

```
if(filsys == nil)
```

```
    fprint(STDERR, "rio: can't create file system server: %r\n");
```
Uses filsys 62h.

4.1 Graphics initialization

The graphics initialization is almost identical to `hellorio.c`: `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()` allocates the grey background and the red color used to highlight borders during window operations. The reason for using `geninitdraw()` rather than `initdraw()` is that `rio` needs to supply its own error handler ([¿derror()¿]): `initdraw()` installs a default handler that exits on errors, but a windowing system must recover gracefully.

⟨main() *graphics initializations* 67a⟩≡ (66a)
```
  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 67d⟩
  ⟨main() keyboard initialisation 68a⟩

  desktop = allocscreen(view, background, false);
  ⟨main() sanity check desktop 67b⟩

  draw(view, viewr, background, nil, ZP);
  flushimage(display, true);
```
Uses background 58c, derror() 292d, desktop 58b, iconinit() 67c, and viewr 57a.

⟨main() *sanity check desktop* 67b⟩≡ (67a)
```
  if(desktop == nil)
      error("can't allocate desktop");
```
Uses desktop 58b and error() 292c.

⟨*function* iconinit 67c⟩≡ (309)
```
  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 58c and red 58d.

4.2 Mouse initialization

⟨main() *mouse initialisation* 67d⟩≡ (67a)
```
  mousectl = initmouse(nil, view);
  if(mousectl == nil)
      error("can't find mouse");
  mouse = mousectl;
```
Uses error() 292c, mouse 58a, and mousectl 57b.

initmouse() takes view 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: before reading mouse input, it is better to ensure the display is up to date with what the program has drawn so far.

4.3 Keyboard initialization

Keyboard initialization mirrors mouse initialization: initkeyboard()X opens /dev/cons (the raw keyboard file), spawns an I/O proc that reads runes and forwards them on a channel, and returns a KeyboardctlX handle. 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() *keyboard initialisation* 68a⟩≡ (67a)
```
keyboardctl = initkeyboard(nil);
if(keyboardctl == nil)
    error("can't find keyboard");
```
Uses error() 292c and keyboardctl 57c.

4.4 Channels creation

rio's threads communicate exclusively through channels rather than shared variables protected by locks. The exitchan is the simplest example: threadmain() blocks on a recv() on this channel, and the mouse thread sends a message when the user selects "Exit". Many more channels will be created later—each new window gets its own set of channels for keyboard, mouse, and control messages.

⟨*global* exitchan 68b⟩≡ (304)
```
// chan<unit> (listener = threadmain, sender = mousethread(Exit) | ?)
Channel *exitchan; /* chan(int) */
```

⟨main() *communication channels creation* 68c⟩≡ (66a) 109c ▷
```
exitchan     = chancreate(sizeof(int), 0);
```
Uses exitchan 68b.

4.5 Threads creation

main() creates only two threads at startup: one for the keyboard and one for the mouse. 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 thread, the worker allocator gets a thread, and each filesystem worker gets a thread.

⟨*constant* STACK 68d⟩≡ (299b)
```
#define STACK 8192
```

⟨main() *threads creation* 68e⟩≡ (66a) 109d ▷
```
threadcreate(keyboardthread, nil, STACK);
threadcreate(mousethread, nil, STACK);
```
Uses STACK 68d, keyboardthread() 71, and mousethread() 72c.

4.6 Filesystem server initialization

The filesystem server is initialized by first creating the worker allocator (`xfidinit()`[70c]), then setting up the server itself (`filsysinit()`). `filsysinit()` creates the two-way pipe that will carry 9P messages between clients and the server, then spawns `filsysproc()`[81c] as a separate proc (not a thread, because it blocks on `read()` of the pipe).

4.6.1 filsysinit()

⟨*function* `filsysinit` 69a⟩≡ (316b)

```
Filsys*
filsysinit(Channel *cxfidalloc)
{
    int pid;
    Filsys *fs;
    ⟨filsysinit() other locals 236d⟩

    ⟨filsysinit() install dumper 289b⟩

    fs = emalloc(sizeof(Filsys));

    if(cexecpipe(&fs->cfd, &fs->sfd) < 0)
        goto Rescue;

    ⟨filsysinit() set clockfd 279c⟩
    ⟨filsysinit() set fs user 285d⟩
    pid = getpid();

    fs->cxfidalloc = cxfidalloc;

    ⟨filsysinit() wctl pipe, process, and thread creation 236c⟩

    proccreate(filsysproc, fs, 10000);

    ⟨filsysinit() srv pipe 234b⟩

    return fs;

Rescue:
    free(fs);
    return nil;
}
```

Uses `cexecpipe()` 69b, `emalloc()` 293d, and `filsysproc()` 81c.

The hardcoded 10000 for `proccreate()` is the stack size for the filesystem server proc. It could probably reuse the `STACK` constant (8192)—the difference is minor and likely just an oversight.

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()`[102e], and `OCEXEC` provides an additional safety net. The standard `pipe()` syscall cannot set per-end flags like `OCEXEC`, so `cexecpipe()` binds the kernel pipe device `#|` directly and opens the two endpoints (`data` and `data1`) independently with different flags.

⟨*function* `cexecpipe` 69b⟩≡ (316b)

```
/*
```

```
 * 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 = open("/mnt/temp/data1", ORDWR|OCEXEC);
    unmount(nil, "/mnt/temp");
    if(*p0<0 || *p1<0)
        return ERROR_NEG1;
    return OK_0;
}
```

4.6.2 Worker allocator: `xfidinit()`

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

```
  filsysproc (master)          xfidallocthread (allocator)
        |                               |
        |--- recv(cxfidalloc) -------->|  "give me a worker"
        |<-- Xfid* -------------------|
        |                               |
        |    ... dispatch 9P work ...   |
        |                               |
        |--- send(cxfidfree, x) ----->|  "worker done, take it back"
        |                               |
```

⟨*global* `cxfidalloc` 70a⟩≡ (307a)

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

⟨*global* `cxfidfree` 70b⟩≡ (307a)

```
// chan<ref<Xfid>> (listner = ??, sender = ??)
static Channel *cxfidfree; /* chan(Xfid*) */
```

⟨*function* `xfidinit` 70c⟩≡ (307a)

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

Uses `STACK` 68d, `cxfidalloc-77` 70a, `cxfidfree-78` 70b, and `xfidallocthread()` 84d.

Chapter 5

Procs and Threads

A windowing system is fundamentally a 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. 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`) 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 demultiplexer: it receives runes from the I/O proc (via `keyboardctl->c`) and forwards them to the current window's `ck` channel. 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. The `nbrecv()` loop batches multiple pending keystrokes into a single message for efficiency.

⟨*function* `keyboardthread` 71⟩≡ (306a)

```
// main -> 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` 61e and `keyboardctl` 57c.

5.2 Mouse thread

The mouse thread is more complex than the keyboard thread because mouse events have a dual nature: some belong to the 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 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 can be extended later with additional channels (e.g., for resize events).

⟨*enum* `Mxxx` 72a⟩≡ (305b)

```
enum {
    MMouse,
    ⟨Mxxx cases 244c⟩
    NALT
};
```

⟨`mousethread()` *locals* 72b⟩≡ (72c) 73a ▷

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

Uses `NALT-94` 72a.

⟨*function* `mousethread` 72c⟩≡ (305b)

```
// main -> threadcreate(<>, nil) -> <>
void
mousethread(void*)
{
    ⟨mousethread() locals 72b⟩

    threadsetname("mousethread");

    ⟨mousethread() alts setup 72d⟩
    alts[NALT].op = CHANEND;

    for(;;)
        // message loop
        switch(alt(alts)){
        ⟨mousethread() event loop cases 73c⟩
        }
}
```

Uses `NALT-94` 72a.

⟨`mousethread()` *alts setup* 72d⟩≡ (72c) 244d ▷

```
// listen
alts[MMouse].c = mousectl->c;
alts[MMouse].v = &mousectl->Mouse;
alts[MMouse].op = CHANRCV;
```

Uses `MMouse-92` 72a and `mousectl` 57b.

⟨mousethread() *locals* 73a⟩+≡ (72c) ◁72b 73b▷
```
  Window *winput;
  Point xy; // logical coord
```

⟨mousethread() *locals* 73b⟩+≡ (72c) ◁73a 74b▷
```
  bool sending = false;
```

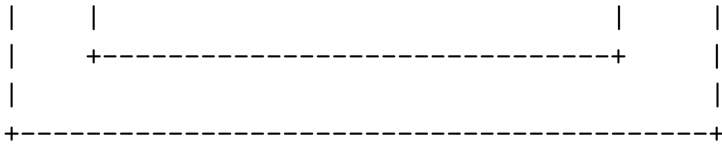
The MMouse case below is the most complex event handler in rio. Two key variables drive the logic: winput captures which window the mouse is currently over (which may differ from input, the keyboard-focused window), and xy holds the mouse coordinates converted from screen-relative (physical) to window-relative (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* 73c⟩≡ (72c) 244e▷
```
  case MMouse:
      ⟨mousethread() if wkeyboard and button 6 274b⟩
  Again:
      winput = input;
      ⟨mousethread() if wkeyboard and ptinrect 274c⟩

      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 218a⟩

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

          ⟨mousethread() set scrolling 217e⟩
          ⟨mousethread() set moving to true for some conditions 95b⟩
          else
             ⟨mousethread() set sending to true for some conditions 74a⟩
      }else
          sending = false;

      ⟨mousethread() if sending 74c⟩
      ⟨mousethread() if not sending 75b⟩

  ⟨mousethread() Drain label 77a⟩
```
Uses MMouse-92 72a, Selborder 73d, input 61e, and mouse 58a.

⟨*constant* Selborder 73d⟩≡ (299a)
```
  Selborder  = 4, /* border of selected window */
```

The conversion code above translates mouse coordinates from physical screen space to the window's image space. Originally, rio used originwindow() to set each window's image origin to (0,0), so the application could draw without knowing its screen position. In that design, screenr held the physical position and i->r started 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)
```

In the current code, `originwindow()` is no longer called during normal window creation, so `i->r` and `screenr` are always equal and the conversion is a no-op—the application receives raw screen coordinates and draws in those same coordinates. It is unclear why `originwindow()` was dropped; the conversion code was kept, suggesting the change may not have been intentional.

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 `mouseopen` is set (meaning the application opened `/mnt/wsys/mouse` and wants raw mouse events), or automatic scrolling is active. When `mouseopen` is not set, middle-click and right-click are intercepted by the mouse thread for window management actions (the edit menu and the system menu).

⟨`mousethread()` *set sending to true for some conditions* 74a⟩≡ (73c)
```
  if(inside &&
     ((mouse->buttons&1) || winput->mouseopen || scrolling))
       sending = true;
```
Uses `mouse` 58a.

⟨`mousethread()` *locals* 74b⟩+≡ (72c) ◁73b 75a▷
```
  Mouse tmp;
```

⟨`mousethread()` *if sending* 74c⟩≡ (73c)
```
  if(sending){
  Sending:
      ⟨mousethread() when sending mouse message to window, set the cursor 74d⟩

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

      // Dispatch, to current window thread!
      send(winput->mc.c, &tmp);
      continue;
  }
```
Uses `mousectl` 57b.

⟨`mousethread()` *when sending mouse message to window, set the cursor* 74d⟩≡ (74c)
```
  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()` 90c, `mouse` 58a, and `wsetcursor()` 91d.

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()`[76a], 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 75a⟩+≡                                        (72c) ◁74b 95a▷
  Window *w;
```

```
⟨mousethread() if not sending 75b⟩≡                                         (73c)
  w = wpointto(mouse->xy);

  ⟨mousethread() when not sending, set cursor part1 76b⟩
  ⟨mousethread() if moving and buttons 95d⟩
  ⟨mousethread() when not sending, set cursor part2 76c⟩

  ⟨mousethread() when not sending, if buttons 76d⟩
  moving = false;
  break;
```

Uses `mouse` 58a and `wpointto()` 76a.

The loop iterates over all windows, keeps only those whose `screenr` contains `pt`, and picks the one with the largest `topped` 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.

⟨*function* `wpointto` 76a⟩≡ (311)
```
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` 61b and `windows` 61a.

⟨`mousethread()` *when not sending, set cursor part1* 76b⟩≡ (75b)
```
/* change cursor if over anyone's border */
if(w != nil)
    cornercursor(w, mouse->xy, false);
else
    riosetcursor(nil, false);
```
Uses `cornercursor()` 90c, `mouse` 58a, and `riosetcursor()` 90b.

⟨`mousethread()` *when not sending, set cursor part2* 76c⟩≡ (75b)
```
if(w != nil)
    cornercursor(w, mouse->xy, false);
```
Uses `cornercursor()` 90c and `mouse` 58a.

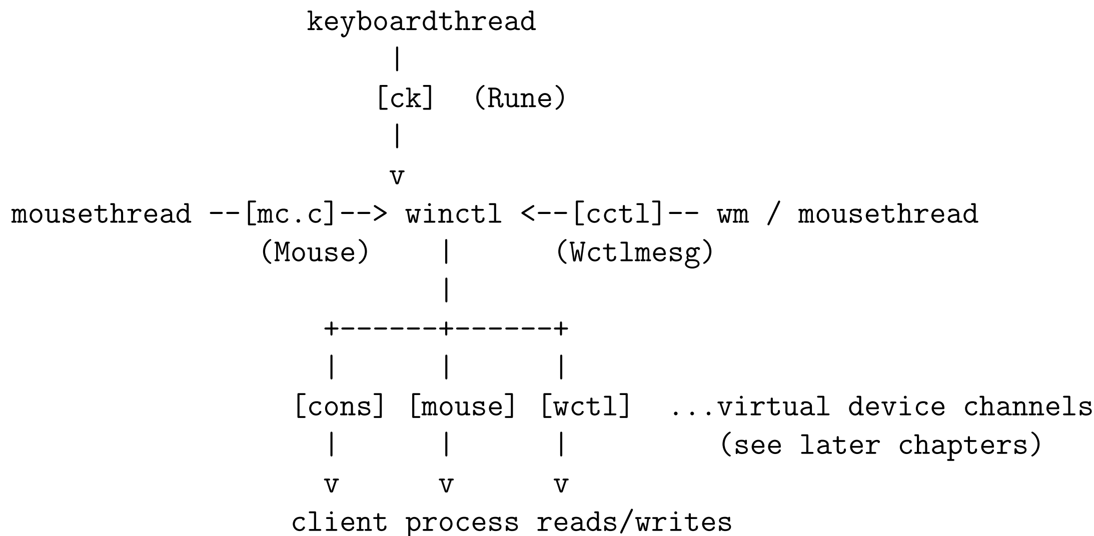
⟨`mousethread()` *when not sending, if buttons* 76d⟩≡ (75b)
```
/* we're not sending the event, but if button is down maybe we should */
if(mouse->buttons){
    /* w->topped will be zero or less if window has been bottomed */
    if(w==nil || (w==winput && w->topped > 0)){
        if(mouse->buttons & 1){
            ;
        }else if(mouse->buttons & 2){
            if(winput && !winput->mouseopen)
                ⟨mousethread() middle click under certain conditions 214⟩
        }else if(mouse->buttons & 4)
                ⟨mousethread() right click under certain conditions 93a⟩
    }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 106c⟩
        if(w && (mouse->buttons!=1 || winborder(w, mouse->xy)))
            // input changed
            goto Again;

        goto Drain;
    }
}
```
Uses `mouse` 58a and `winborder()` 90d.

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* 77a⟩≡ (73c)
```
Drain:
    do {
        readmouse(mousectl);
    } while(mousectl->buttons);
    moving = false;
    goto Again; /* recalculate mouse position, cursor */
```
Uses `mousectl` 57b.

5.3 Window threads

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` 77b⟩≡ (306b)
```
enum {
    WKey,
    WMouse,
    WCtl,
    ⟨Wxxx cases 152a⟩

    NWALT
};
```

`winctl()`[78] is the heart of each window. It is the thread where keyboard input gets processed, mouse events get interpreted, and control messages (resize, delete, hide) take effect. Every window gets its own instance, making each window an autonomous concurrent entity with its own event loop—the "active object" pattern.

⟨*function* `winctl` 78⟩≡ (306b)
```
// ... -> new() -> threadcreate(<>, w)
void
winctl(void *arg)
{
    Window *w = arg;
    // map<enum<Wxxx>, Alt>
    Alt alts[NWALT+1];
    int event;
    ⟨winctl() other locals 79b⟩

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

    ⟨winctl() channels creation 152c⟩

    ⟨winctl() alts setup 79c⟩
    alts[NWALT].op = CHANEND;

    for(;;){
        ⟨winctl() alts adjustments 152f⟩

        // to isolate messaging bug
        //alts[WKey].op = CHANNOP;
        //alts[WMouse].op = CHANNOP;
        ////alts[WCtl].op = CHANNOP;
        //alts[WMouseread].op = CHANNOP;
        //alts[WCread].op = CHANNOP;
        //alts[WCwrite].op = CHANNOP;
        //alts[WWread].op = CHANNOP;

        // event loop
        event = alt(alts);
        if(DEBUG) fprint(STDERR, "winctl: win=%d, event=%d\n", w->id, event);
        if(event == -1) {
          fprint(STDERR, "winctl: interrupted %r");
          exits("interrupted");
        }
        switch(event){
        ⟨winctl() event loop cases 79d⟩
        }

        if(!w->deleted)
            flushimage(display, true);
    }
}
```
Uses `DEBUG` 289a and `NWALT-91` 77b.

The `deleted` field will be explained in Chapter ??. Note that after each event, `winctl()` calls `flushimage()` unless the window has been deleted. 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.

5.3.1 Keyboard events listening

The first connection to wire up is the keyboard. Each window has a `ck` channel that carries batches of `Rune`s from `keyboardthread()`[71]. When a window has focus, the keyboard thread sends typed runes on `ck`, and `winctl()`[78] dispatches each rune to `wkeyctl()`[79e] for processing.

```
⟨Window keyboard fields 79a⟩≡                                    (59)
  // chan<Rune, 20> (listener = winctl, sender = keyboardthread)
  Channel *ck;  /* chan(Rune[10]) */
```

```
⟨winctl() other locals 79b⟩≡                                 (78) 80c ▷
  Rune *kbdr;
```

```
⟨winctl() alts setup 79c⟩≡                                   (78) 80b ▷
  alts[WKey].c = w->ck;
  alts[WKey].v = &kbdr;
  alts[WKey].op = CHANRCV;
```

Uses `WKey-84` 77b.

```
⟨winctl() event loop cases 79d⟩≡                             (78) 80d ▷
  case WKey:
      for(i=0; kbdr[i] != L'\0'; i++)
          wkeyctl(w, kbdr[i]);
      break;
```

Uses `WKey-84` 77b and `wkeyctl()` 79e.

```
⟨function wkeyctl 79e⟩≡                                          (312b)
  void
  wkeyctl(Window *w, Rune r)
  {
      ⟨wkeyctl() locals 204b⟩

      ⟨wkeyctl() sanity check rune 79f⟩
      ⟨wkeyctl() return if window was deleted 79g⟩

      /* navigation keys work only when mouse is not open */
      ⟨wkeyctl() when mouse not opened and navigation keys 205a⟩

      ⟨wkeyctl() if rawing 153b⟩
      ⟨wkeyctl() if holding 278b⟩

      ⟨wkeyctl() when not rawing 204a⟩
  }
```

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 79f⟩≡                               (79e)
  if(r == 0)
      return;
```

```
⟨wkeyctl() return if window was deleted 79g⟩≡                    (79e)
  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()`[72c]. This is not the same `Mousectl` as the one used by `rio`'s own IO proc for reading `/dev/mouse`—that one belongs to `rio` itself. Here, each `Window` has its own `Mousectl` because it needs the same fields (a `Mouse`, a channel `c`, and a `resizec`) to forward events to `winctl()`[78]. 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* 80a⟩≡ (59) 91c▷
```
// mc.c = chan<Mouse> (listener = winctl, sender = mousethread)
Mousectl mc;
```

⟨`winctl()` *alts setup* 80b⟩+≡ (78) ◁79c 80g▷
```
alts[WMouse].c = w->mc.c;
alts[WMouse].v = &w->mc.Mouse;
alts[WMouse].op = CHANRCV;
```
Uses `WMouse-85` 77b.

⟨`winctl()` *other locals* 80c⟩+≡ (78) ◁79b 80f▷
```
char buf[128]; // /dev/mouse interface
```

⟨`winctl()` *event loop cases* 80d⟩+≡ (78) ◁79d 81a▷
```
case WMouse:
    ⟨winctl() WMouse case if mouseopen 151f⟩
    else
    ⟨winctl() WMouse case if not mouseopen 216a⟩
    break;
```
Uses `WMouse-85` 77b.

As explained earlier, the behavior depends on whether the application has opened `/mnt/wsys/mouse` (`mouseopen`). 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 `cctl`, which carries `Wctlmesg`[96b] 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). When `wctlmesg()`[96d] returns `Exited`, the window is being destroyed and the thread exits.

⟨`Window` *control fields* 80e⟩≡ (59)
```
// chan<Wctlmesg, 20> (listener = winctl, sender = mousethread | ...)
Channel *cctl;  /* chan(Wctlmesg)[20] */
```

⟨`winctl()` *other locals* 80f⟩+≡ (78) ◁80c 151e▷
```
Wctlmesg wcm;
```

⟨`winctl()` *alts setup* 80g⟩+≡ (78) ◁80b 152e▷
```
alts[WCtl].c = w->cctl;
alts[WCtl].v = &wcm;
alts[WCtl].op = CHANRCV;
```
Uses `WCtl-86` 77b.

⟨winctl() *event loop cases* 81a⟩+≡ (78) ◁80d 152h▷
```
  case WCtl:
      if(wctlmesg(w, wcm.type, wcm.r, wcm.image) == Exited){
          ⟨winctl() Wctl case, free channels if wctlmesg is Excited 152d⟩
          threadexits(nil);
      }
      continue;
```
Uses Exited 111d, WCtl-86 77b, and wctlmesg() 96d.

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 ??.

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

⟨*global* messagesize 81b⟩≡ (316a)
```
  int messagesize = 8192+IOHDRSZ; /* good start */
```
Uses messagesize 81b.

5.4.1 filsysproc()

⟨*function* filsysproc 81c⟩≡ (316b)
```
  // main -> filsysinit -> proccreate(<>, fs, ...) -> <>
  static
  void
  filsysproc(void *arg)
  {
      Filsys *fs = arg;
      int n;
      byte *buf;
      Xfid *x = nil;
      Fid *f;
      ⟨filsysproc() other locals 82c⟩

      threadsetname("FILSYSPROC");

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

          n = read9pmsg(fs->sfd, buf, messagesize);
          ⟨filsysproc() sanity check n 82a⟩
          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 289c⟩

          ⟨filsysproc() sanity check x type 82d⟩
```

```
        else{
            ⟨filsysproc() if x type is Tversion or Tauth 82b⟩
            else
                f = newfid(fs, x->req.fid);
            x->f = f;

            // Dispatch
            x  = (*fcall[x->req.type])(fs, x, f);
        }
        ⟨filsysproc() end of loop 83e⟩
    }
}
```

Uses emalloc() 293d, error() 292c, fcall 65, messagesize 81b, and newfid() 63f.

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()[135a], 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()X to request a worker, then receives the Xfid[64c] 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* 82a⟩≡ (81c)

```
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() 292c and errorshouldabort 292a.

⟨filsysproc() *if x type is Tversion or Tauth* 82b⟩≡ (81c)

```
if(x->req.type==Tversion || x->req.type==Tauth)
    f = nil;
```

⟨filsysproc() *other locals* 82c⟩≡ (81c)

```
Fcall fc;
```

⟨filsysproc() *sanity check x type* 82d⟩≡ (81c)

```
if(fcall[x->req.type] == nil)
    x = filsysrespond(fs, x, &fc, Ebadfcall);
```

Uses Ebadfcall 290d, fcall 65, and filsysrespond() 124a.

5.4.2 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` without dispatching to a worker, since it is a one-time handshake.

⟨`fcall` *other methods* 83a⟩≡ (65) 136b ▷
```
  [Tversion] = filsysversion,
```
Uses `filsysversion()` 83b.

⟨*function* `filsysversion` 83b⟩≡ (317)
```
  static
  Xfid*
  filsysversion(Filsys *fs, Xfid *x, Fid*)
  {
      Fcall fc;

      ⟨filsysversion() sanity checks 83d⟩
      // else
      messagesize = x->req.msize;
      fc.msize = messagesize;
      fc.version = "9P2000";
      return filsysrespond(fs, x, &fc, nil);
  }
```
Uses `filsysrespond()` 124a and `messagesize` 81b.

⟨*global* `firstmessage` 83c⟩≡ (317)
```
  bool firstmessage = true;
```
Uses `firstmessage` 83c.

⟨`filsysversion()` *sanity checks* 83d⟩≡ (83b) 83f ▷
```
  if(!firstmessage)
      return filsysrespond(x->fs, x, &fc, "version request not first message");
```
Uses `filsysrespond()` 124a and `firstmessage` 83c.

⟨`filsysproc()` *end of loop* 83e⟩≡ (81c)
```
  firstmessage = false;
```
Uses `firstmessage` 83c.

⟨`filsysversion()` *sanity checks* 83f⟩+≡ (83b) ◁ 83d
```
  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()` 124a.

5.5 Xfid allocator thread

The allocator thread manages a pool (arena) of `Xfid` worker handles. When `filsysproc()`[81c] requests a worker (Alloc), the allocator either recycles one from the free list or creates a new one with its own `xfidctl` thread. When a worker finishes (Free), it is returned to the free list. The two free lists serve different purposes: `xfid` tracks all ever-allocated workers, while `xfidfree` tracks currently idle ones.

⟨*enum* `Xxxx` 83g⟩≡ (307a)
```
  enum {
      Alloc,
      Free,

      N
  };
```

⟨*global* xfid 84a⟩≡ (307a)
```
// list<ref_own<Xfid>> (next = Xfid.next)
static Xfid *xfid;
```

⟨*global* xfidfree 84b⟩≡ (307a)
```
// list<ref_own<Xfid>> (next = Xfid.free)
static Xfid *xfidfree;
```

⟨Xfid *extra fields* 84c⟩≡ (64c)
```
Xfid *next;
Xfid *free;
```

⟨*function* xfidallocthread 84d⟩≡ (307a)
```
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 282b⟩

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

                // new Xfid threads!
                threadcreate(xfidctl, x, 16384);
            }
            ⟨xfidallocthread() sanity check x when Alloc 85a⟩
            incref(x);

            sendp(cxfidalloc, x);
            break;

        case Free:
            ⟨xfidallocthread() sanity check x when Free 85b⟩
            // insert_list(x, xfidfree)
            x->free = xfidfree;
            xfidfree = x;
            break;
        }
    }
```

```
}
```

Uses Alloc-79 83g, Free-80 83g, N-81 83g, cxfidalloc-77 70a, cxfidfree-78 70b, emalloc() 293d, xfid-76 84a, xfidctl() 85c, and xfidfree-75 84b.

⟨xfidallocthread() *sanity check x when Alloc* 85a⟩≡ (84d)

```
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() 292c.

⟨xfidallocthread() *sanity check x when Free* 85b⟩≡ (84d)

```
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() 292c.

5.6 Xfid threads

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 85c⟩≡ (307a)

```
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-78 70b.

The functions sent through the channel are the 9P operation handlers: xfidattach()[126d], xfidopen()[131d], xfidread()[134a], xfidwrite()[135a], xfidclose()[132b], and xfidflush()[283a]. 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`[86] below (a plus sign) appears during window creation (`sweep()`[104d]), `boxcursor`[87a] during window moves (`drag()`[113b]), `sightcursor`[87b] when "Delete" is selected from the system menu, `whitearrow`[87c] in holding mode, and `query`[87d] for unknown states. The default arrow cursor lives in the draw library, not in `rio`.

Each cursor is a `Cursor` structure (defined in `cursor.h`) containing a *hotspot offset* and two 16x16 bitmaps.

⟨*global crosscursor (windows/rio/data.c)* 86⟩≡ (309)

```
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, }
};
```

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.

⟨*global boxcursor (windows/rio/data.c)* 87a⟩≡ (309)

```
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 (windows/rio/data.c)* 87b⟩≡ (309)

```
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 (windows/rio/data.c)* 87c⟩≡ (309)

```
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 (windows/rio/data.c)* 87d⟩≡ (309)

```
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` 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 (windows/rio/data.c)* 88a⟩≡ (309)
```
Cursor *corners[9] = {
    &tl,    &t,   &tr,
    &l,    nil,   &r,
    &bl,    &b,   &br,
};
```
Uses `b` 89b, `bl` 89c, `br` 89a, `l` 89d, `r` 88e, `t-64` 88c, `tl` 88b, and `tr` 88d.

⟨*global* `tl` 88b⟩≡ (309)
```
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` 88c⟩≡ (309)
```
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` 88d⟩≡ (309)
```
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` 88e⟩≡ (309)
```
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 89a⟩≡ (309)

```
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 89b⟩≡ (309)

```
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 89c⟩≡ (309)

```
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 89d⟩≡ (309)

```
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()`[90c] 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()`[91d]. The `menuing`[104c] flag prevents cursor changes during system menu operations, when `rio` must keep control of the cursor.

6.2.1 `riosetcursor()`

Since changing the cursor requires a write to `/dev/cursor`—a system call—`riosetcursor` 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` 90a⟩≡ (310)
```
Cursor *lastcursor;
```

⟨*function* `riosetcursor` 90b⟩≡ (310)
```
void
riosetcursor(Cursor *p, bool force)
{
    if(!force && p==lastcursor)
        return;
    setcursor(mousectl, p);
    lastcursor = p;
}
```
Uses `lastcursor` 90a and `mousectl` 57b.

6.2.2 `cornercursor()`

This is the main cursor dispatch function, called from `mousethread()`[72c] on every mouse move. It checks whether the mouse is on the window border via `winborder()` (point inside `screenr` but outside the inset rectangle): if so, it picks the appropriate directional cursor from the `corners`[88a] array; otherwise it falls through to `wsetcursor()`[91d], which shows the application's cursor or the default arrow.

⟨*function* `cornercursor` 90c⟩≡ (308)
```
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` 88a, `riosetcursor()` 90b, `whichcorner()` 91a, `winborder()` 90d, and `wsetcursor()` 91d.

⟨*function* `winborder` 90d⟩≡ (311)
```
bool
winborder(Window *w, Point xy)
{
    return ptinrect(xy, w->screenr) &&
           !ptinrect(xy, insetrect(w->screenr, Selborder));
}
```
Uses `Selborder` 73d.

The `whichcorner()` function maps a point on the window border to an index in the 3x3 `corners` array. The `portion` 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* `whichcorner` 91a⟩≡ (308)
```
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()` 91b.

⟨*function* `portion` 91b⟩≡ (308)
```
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 `wsetcursor()`

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. The `wsetcursor()` function checks whether the mouse is actually over this window via `wpointto()`[76a] and whether a system menu operation is in progress via `menuing`[104c], ensuring that `rio`'s own cursor shapes take priority during window management.

⟨`Window` *mouse fields* 91c⟩+≡ (59) ◁80a 150b▷
```
Cursor  cursor;
// option<ref<Cursor>> (to Window.cursor when not None)
Cursor  *cursorp;
```

⟨*function* `wsetcursor` 91d⟩≡ (311)
```
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 276f⟩
    }else
        p = nil;
```

```
      if(!menuing)
          riosetcursor(p, force);
  }
```

Uses menuing 104c, mouse 58a, riosetcursor() 90b, and wpointto() 76a.

Chapter 7

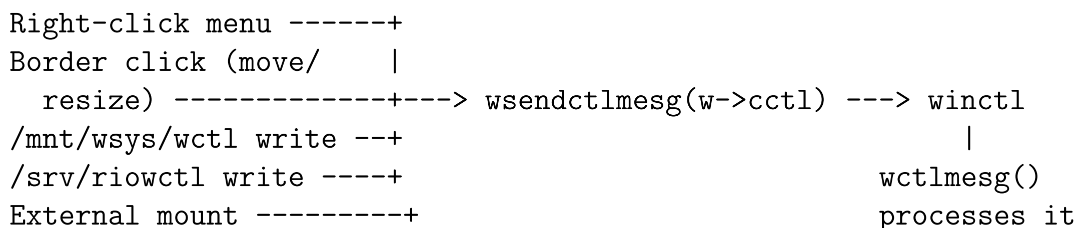
Window Manager

Now that I have covered the threads and procs that form `rio`'s concurrent architecture, I can present the window manager—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 X11 where they are separate. All window management paths converge to the same mechanism: sending a `Wctlmesg`[96b] to the target window's `cctl` channel, where its `winctl()`[78] thread will process it.

Window management operations can be triggered from several sources, but they all converge to the same mechanism:



```
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. 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` messages to the target window's thread.

⟨`mousethread()` *right click under certain conditions* 93a⟩≡ (76d)

```
button3menu();
```

Uses `button3menu()` 93b.

⟨*function* `button3menu` 93b⟩≡ (308)

```
void
button3menu(void)
{
    int i;

    ⟨button3menu() menu3str adjustments with hidden windows 121a⟩
```

```
    sweeping = true;
    switch(i = menuhit(3, mousectl, &menu3, desktop)){
    ⟨button3menu() cases 94e⟩
    case -1:
        break;
    }
    sweeping = false;
}
```

Uses desktop 58b, menu3 94b, mousectl 57b, and sweeping 94a.

The `sweeping` 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 94a⟩≡ (304)

```
bool sweeping;
```

⟨*global* menu3 94b⟩≡ (308)

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

Uses menu3str 94c.

⟨*global* menu3str 94c⟩≡ (308)

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

⟨*enum* _anon_ (windows/rio/rio.c) 94d⟩≡ (308)

```
enum RightMenuCommand
{
    New,

    Reshape,
    Move,
    Delete,
    Hide,

    Exit,

    Hidden,
};
```

The code below closes the loop with `threadmain()`, which blocks on `recv(exitchan)`. When the user selects "Exit," this `send()` unblocks `threadmain()` and `rio` shuts down.

⟨button3menu() *cases* 94e⟩≡ (93b) 97a ▷

```
case Exit:
    send(exitchan, nil);
    break;
```

Uses Exit-100 94d and exitchan 68b.

7.3 Window borders click

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()`[117a] (resize), while right-click triggers `drag()`[113b] (move). The result is sent as a `Reshaped` or `Moved` message to the window's `winctl` thread.

⟨`mousethread()` *locals* 95a⟩+≡ (72c) ◁75a 95c▷
```
  bool moving = false;
```

⟨`mousethread()` *set moving to true for some conditions* 95b⟩≡ (73c)
```
  /* 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` 58a and `winborder()` 90d.

⟨`mousethread()` *locals* 95c⟩+≡ (72c) ◁95a 217d▷
```
  bool inside, band;
  Window *oin;
  Image *i;
  Rectangle r;
```

⟨`mousethread()` *if moving and buttons* 95d⟩≡ (75b)
```
  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` 96a, `Reshaped` 96a, `bandsize()` 117a, `cornercursor()` 90c, `drag()` 113b, `mouse` 58a, `sweeping` 94a, and `wsendctlmesg()` 96c.

7.4 `Wctlmesg`

The `Wctlmesg` structure is the message type sent from the mouse thread (or other sources) to a window's `winctl` thread through its `cctl` channel. The message carries the operation type (Reshaped, Moved, 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 96a⟩≡ (299b)
```
enum ControlMessage /* control messages */
{
    Reshaped, // Resized, Hide/Expose
    Moved,
    ⟨Wctlmesgkind cases 108b⟩
};
```

⟨*struct* Wctlmesg 96b⟩≡ (299b)
```
struct Wctlmesg
{
    // enum<Wctlmesgkind>
    int  type;

    Rectangle r;
    Image *image;
};
```

⟨*function* wsendctlmesg 96c⟩≡ (311)
```
void
wsendctlmesg(Window *w, int type, Rectangle r, Image *image)
{
    Wctlmesg wcm;

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

    if(DEBUG) fprint(STDERR, "wsendctlmesg: win=%d, type=%d\n", w->id, type);
    send(w->cctl, &wcm);
}
```
Uses DEBUG 289a.

⟨*function* wctlmesg 96d⟩≡ (306b)
```
int
wctlmesg(Window *w, int m, Rectangle r, Image *i)
{
    char buf[64];

    if(DEBUG) fprint(STDERR, "wctlmesg: win=%d, type=%d\n", w->id, m);

    switch(m){
    ⟨wctlmesg() cases 108d⟩
    default:
        error("unknown control message");
        break;
    }
    return m;
}
```
Uses DEBUG 289a and error() 292c.

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 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()`[104d] lets them draw a rectangle, then `new()`[97b] creates everything needed for the new window (see Section 2.5.2 for the high-level overview).

⟨`button3menu()` *cases* 97a⟩+≡ (93b) ◁94e 107c▷
```
case New:
    new(sweep(), false, scrolling, 0, nil, "/bin/rc", nil);
    break;
```
Uses `New-95` 94d, `new()` 97b, `scrolling` 272a, and `sweep()` 104d.

7.5.1 Window thread creation: `new()`

`new()` orchestrates the creation of a window in several steps: allocate channels, build the `Window`[59] structure with `wmk()`[98e], add it to the global `windows`[61a] array, create a `winctl()`[78] 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.p` and just needs a window created for it—see the Advanced Topics chapter for this external mount mechanism.

⟨*function* `new` 97b⟩≡ (308)
```
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 100g⟩

    ⟨new() sanity check i 98c⟩

    ⟨new() channels creation 98a⟩
    cpid = chancreate(sizeof(int), 0);
    ⟨new() sanity check channels 98d⟩

    ⟨new() mc allocation 98b⟩

    // 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 120d⟩

    if(DEBUG) fprint(STDERR, "new: creating winctl thread for win=%d\n", w->id);

    // create a new thread! for this new window!
    threadcreate(winctl, w, 8192);

    if(!hideit)
        wcurrent(w);

    flushimage(display, true);

    // create a new process
    ⟨new() if pid == 0, create winshell process and set pid 100h⟩
    ⟨new() sanity check pid received from winshell 101a⟩
```

```
    wsetpid(w, pid, true);

    // create a new layer
    wsetname(w);

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

    chanfree(cpid);
    return w;
}
```
Uses DEBUG 289a, erealloc() 293c, estrdup() 294a, nwindow 61b, wcurrent() 106e, winctl() 78, windows 61a, wmk() 98e, wsetname() 104a, and wsetpid() 102c.

⟨new() *channels creation* 98a⟩≡ (97b)
```
cm = chancreate(sizeof(Mouse), 0);
ck = chancreate(sizeof(Rune*), 0);
cctl = chancreate(sizeof(Wctlmesg), 4);
```

⟨new() *mc allocation* 98b⟩≡ (97b)
```
mc = emalloc(sizeof(Mousectl));
*mc = *mousectl;
mc->image = i;
mc->c = cm;
```
Uses emalloc() 293d and mousectl 57b.

⟨new() *sanity check i* 98c⟩≡ (97b)
```
if(i == nil)
    return nil;
```

⟨new() *sanity check channels* 98d⟩≡ (97b)
```
if(cm==nil || ck==nil || cctl==nil)
    error("new: channel alloc failed");
```
Uses error() 292c.

7.5.2 Window allocation: wmk()

wmk() is the constructor for the Window[59] structure. It assigns a unique id (monotonically increasing) and a topped counter that tracks stacking order—higher values mean more recently focused.

⟨*function* wmk 98e⟩≡ (311)
```
Window*
wmk(Image *i, Mousectl *mc, Channel *ck, Channel *cctl, bool scrolling)
{
    Window *w;
    Rectangle r;

    ⟨wmk() colors initialisation 164c⟩

    w = emalloc(sizeof(Window));

    w->i = i;
    w->screenr = i->r;
    w->cursorp = nil;

    w->id = ++id;
    w->topped = ++topped;
```

```
    w->label = estrdup("<unnamed>");

    ⟨wmk() channels settings 99a⟩
    ⟨wmk() channels creation 139b⟩
    ⟨wmk() textual window settings 160a⟩
    ⟨wmk() process settings 100a⟩

    ⟨wmk() drawing border 99b⟩
    ⟨wmk() drawing scrollbar 191a⟩

    incref(w); /* ref will be removed after mounting; avoids delete before ready to be deleted */
    return w;
}
```

Uses `emalloc()` 293d, `estrdup()` 294a, `id-66` 60b, and `topped-65` 61c.

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()`[101b] after `filsysmount()`[102e] completes (via `wclose()`[111a]).

⟨`wmk()` *channels settings* 99a⟩≡ (98e)

```
w->mc = *mc;
w->ck = ck;
w->cctl = cctl;
```

⟨`wmk()` *drawing border* 99b⟩≡ (98e)

```
wborder(w, Selborder);
```

Uses `Selborder` 73d and `wborder()` 99c.

⟨*function* `wborder` 99c⟩≡ (311)

```
void
wborder(Window *w, int type)
{
    Image *col;

    ⟨wborder() sanity check w 99g⟩
    ⟨wborder() if holding 277a⟩
    else{
        if(type == Selborder)
            col = titlecol;
        else
            col = lighttitlecol;
    }

    border(w->i, w->i->r, Selborder, col, ZP);
}
```

Uses `Selborder` 73d, `lighttitlecol-70` 99e, and `titlecol-69` 99d.

⟨*global* `titlecol` 99d⟩≡ (311)

```
static Image *titlecol;
```

⟨*global* `lighttitlecol` 99e⟩≡ (311)

```
static Image *lighttitlecol;
```

⟨`wmk()` *extra colors initialisation* 99f⟩≡ (164c) 164g ▷

```
titlecol     = allocimage(display, Rect(0,0,1,1), CMAP8, true, DGreygreen);
lighttitlecol= allocimage(display, Rect(0,0,1,1), CMAP8, true, DPalegreygreen);
```

Uses `lighttitlecol-70` 99e and `titlecol-69` 99d.

⟨`wborder()` *sanity check w* 99g⟩≡ (99c)

```
if(w->i == nil)
    return;
```

⟨wmk() *process settings* 100a⟩≡ (98e)
```
  w->notefd = -1;
  w->dir = estrdup(startdir);
```
Uses estrdup() 294a and startdir 100c.

⟨Window *other fields* 100b⟩+≡ (59) ◁61d 116a▷
```
  char  *dir; // /dev/wdir
```

⟨*global* startdir 100c⟩≡ (304)
```
  char  *startdir;
```

⟨main() *locals* 100d⟩≡ (66a) 211f▷
```
  char buf[256];
```

⟨main() *set some globals* 100e⟩≡ (66a) 211g▷
```
  if(getwd(buf, sizeof buf) == nil)
      startdir = estrdup(".");
  else
      startdir = estrdup(buf);
```
Uses estrdup() 294a and startdir 100c.

7.5.3 Window process creation: winshell()

For each new window, rio forks a new process (winshell()[101b]) that will exec a shell (usually rc -i). 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. But 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 cfd.

⟨*global* rcargv 100f⟩≡ (308)
```
  char *rcargv[] = { "rc", "-i", nil };
```

⟨new() *other locals* 100g⟩≡ (97b)
```
  void **arg;
```

⟨new() *if pid == 0, create winshell process and set pid* 100h⟩≡ (97b)
```
  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;

      if(DEBUG) fprint(STDERR, "new: creating new winshell\n");
      proccreate(winshell, arg, 8192);

      pid = recvul(cpid);
```

```
    if(DEBUG) fprint(STDERR, "new: created winshell=%d\n", pid);

    free(arg);
}
```

Uses DEBUG 289a, emalloc() 293d, rcargv 100f, and winshell() 101b.

⟨new() *sanity check pid received from winshell* 101a⟩≡ (97b)

```
if(pid == 0){
    /* window creation failed */
    wsendctlmesg(w, Deleted, ZR, nil);
    chanfree(cpid);
    return nil;
}
```

Uses Deleted 108b and wsendctlmesg() 96c.

⟨*function* winshell 101b⟩≡ (312a)

```
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];
    dir  = arg[4];

    if(DEBUG) fprint(STDERR, "winshell: cmd = %s\n", cmd);

    // copy namespace/file-descriptors/environment-variables (do not share)
    rfork(RFNAMEG|RFFDG|RFENVG);

    ⟨winshell() adjust namespace 102d⟩
    ⟨winshell() reassign STDIN/STDOUT after namespace adjustment 103a⟩

    if(wclose(w) == false){ /* remove extra ref hanging from creation */

        if(DEBUG) fprint(STDERR, "winshell: before procexec %s\n", cmd);
        notify(nil);
        dup(STDOUT, STDERR); // STDERR = STDOUT
        if(dir)
            chdir(dir);

        // Exec!!
        procexec(pidc, cmd, argv);
        _exits("exec failed"); // should never be reached
    }
}
```

Uses DEBUG 289a and wclose() 111a.

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.

⟨Window *process fields* 102a⟩≡ (59) 102b ▷
```
int  pid;
```

⟨Window *process fields* 102b⟩+≡ (59) ◁ 102a
```
// /proc/<pid>/notepg
fdt   notefd;
```

⟨*function* wsetpid 102c⟩≡ (311)
```
void
wsetpid(Window *w, int pid, bool dolabel)
{
    char buf[128];
    fdt fd;

    w->pid = pid;
    if(DEBUG) fprint(STDERR, "wsetpid: win=%d pid =%d\n", w->id, w->pid);

    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;
}
```
Uses DEBUG 289a and estrdup() 294a.

7.5.4 Namespace adjustments: filsysmount()

This is the key step that makes rio's virtualization work. filsysmount() 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. Then it binds /mnt/wsys before /dev, so that /dev/cons, /dev/mouse, etc. resolve to rio's virtual files rather than the real device files. After that, the child process simply opens /dev/cons for stdin and stdout—but these are now served by rio.

⟨winshell() *adjust namespace* 102d⟩≡ (101b)
```
err = filsysmount(filsys, w->id);
⟨winshell() sanity check err filsysmount 103d⟩
```
Uses filsys 62h and filsysmount() 102e.

⟨*function* filsysmount 102e⟩≡ (316b)
```
/*
 * Called only from a different FD group
 */
errorneg1
filsysmount(Filsys *fs, int id)
{
    char buf[32];
    errorneg1 err;

    if(DEBUG) fprint(STDERR, "filsysmount1\n");

    close(fs->sfd); /* close server end so mount won't hang if exiting */
```

```
    sprint(buf, "%d", id);
    err = mount(fs->cfd, -1, "/mnt/wsys", MREPL, buf);
    ⟨filsysmount() sanity check err mount 103e⟩
    err = bind("/mnt/wsys", "/dev", MBEFORE);
    ⟨filsysmount() sanity check err bind 103f⟩

    if(DEBUG) fprint(STDERR, "filsysmount2\n");
    return OK_0;
}
```

Uses DEBUG 289a.

⟨winshell() *reassign STDIN/STDOUT after namespace adjustment* 103a⟩≡ (101b)

```
// reassign stdin/stdout to virtualized /dev/cons from filsysmount
close(STDIN);
err = open("/dev/cons", OREAD);
⟨winshell() sanity check err open cons stdin 103b⟩
close(STDOUT);
err = open("/dev/cons", OWRITE);
⟨winshell() sanity check err open cons stdout 103c⟩
```

⟨winshell() *sanity check err open cons stdin* 103b⟩≡ (103a)

```
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* 103c⟩≡ (103a)

```
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* 103d⟩≡ (102d)

```
if(err < 0){
    fprint(STDERR, "mount failed: %r\n");
    sendul(pidc, 0);
    threadexits("mount failed");
}
```

⟨filsysmount() *sanity check err mount* 103e⟩≡ (102e)

```
if(err < 0){
    fprint(STDERR, "mount failed: %r\n");
    return ERROR_NEG1;
}
```

⟨filsysmount() *sanity check err bind* 103f⟩≡ (102e)

```
if(err < 0){
    fprint(STDERR, "bind failed: %r\n");
    return ERROR_NEG1;
}
```

7.5.5 Public layer: `wsetname()`

The namespace adjustments handle `/dev/cons` and `/dev/mouse` but not `/dev/draw`—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` 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]).

⟨Window *id fields* 103g⟩+≡ (59) ◁60e

```
uint  namecount;
```

⟨*function* wsetname 104a⟩≡ (311)

```
// 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 104b⟩
}
```

⟨wsetname() *if image name already in use, try another name* 104b⟩≡ (104a)

```
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()

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).

We finally see the menuing flag mentioned earlier in wsetcursor()[91d]: 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 104c⟩≡ (304)

```
bool menuing;/* menu action is pending; waiting for window to be indicated */
```

⟨*function* sweep 104d⟩≡ (308)

```
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 105b⟩
                oi = i;
                border(i, r, Selborder, red, ZP);
                flushimage(display, true);
            }
        }
    }
    ⟨sweep() sanity check mouse buttons, i, and rectangle size 106a⟩
    oi = i;
    i = allocwindow(desktop, oi->r, Refbackup, DWhite);
    freeimage(oi);
    ⟨sweep() sanity check i 105b⟩
    border(i, r, Selborder, red, ZP);
    cornercursor(input, mouse->xy, true);
    goto Return;
⟨sweep() Rescue handler 106b⟩

 Return:
    moveto(mousectl, mouse->xy); /* force cursor update; ugly */
    menuing = false;
    return i;
}
```

Uses `Selborder` 73d, `cornercursor()` 90c, `crosscursor` 86, `desktop` 58b, `input` 61e, `menuing` 104c, `mouse` 58a, `mousectl` 57b, `onscreen()` 105a, `red` 58d, and `riosetcursor()` 90b.

⟨*function* `onscreen` 105a⟩≡ (308)

```
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()` 293b and `min()` 293a.

⟨`sweep()` *sanity check i* 105b⟩≡ (104)

```
if(i == nil)
    goto Rescue;
```

⟨`sweep()` *sanity check mouse buttons, i, and rectangle size* 106a⟩≡ (104d)

```
if(mouse->buttons != 0)
    goto Rescue;
if(i==nil || Dx(i->r) < 100 || Dy(i->r) < 3*font->height)
    goto Rescue;
```

Uses `mouse` 58a.

⟨`sweep()` *Rescue handler* 106b⟩≡ (104d)

```
Rescue:
    freeimage(i);
    i = nil;
    cornercursor(input, mouse->xy, true);
    while(mouse->buttons)
        readmouse(mousectl);
```

Uses `cornercursor()` 90c, `input` 61e, `mouse` 58a, and `mousectl` 57b.

7.6 Window focus

Clicking on an unfocused window brings it to the front via `wtop()`, which calls `topwindow()` (from the graphics library) to raise the image layer, then `wcurrent()`[106e] to update the global `input`[61e] pointer and repaint the border colors—the focused window gets a different border color than unfocused ones.

⟨`mousethread()` *click on unfocused window, set w* 106c⟩≡ (76d)

```
w = wtop(mouse->xy);
```

Uses `mouse` 58a and `wtop()` 106d.

⟨*function* `wtop` 106d⟩≡ (311)

```
Window*
wtop(Point pt)
{
    Window *w;

    w = wpointto(pt);
    if(w){
        if(w->topped == topped)
            return nil;
        topwindow(w->i); // window.h (was in draw.h)
        wcurrent(w);
        flushimage(display, true);
        w->topped = ++topped;
    }
    return w;
}
```

Uses `topped-65` 61c, `wcurrent()` 106e, and `wpointto()` 76a.

`wcurrent()` updates the global `input` pointer (the focused window) and repaints both the old and new focus windows via `wrepaint()`[107a]. The repaint is necessary because `rio` draws different border colors for focused versus unfocused windows—the focused window gets `titlecol`[99d] (dark grey-green) and unfocused ones get `lighttitlecol`[99e] (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` 106e⟩≡ (311)

```
void
wcurrent(Window *w)
{
    Window *oi;
```

```
    ⟨wcurrent() if wkeyboard 274d⟩
    oi = input;
    // updated input!
    input = w;

    if(oi && oi != w)
        wrepaint(oi);
    if(w){
        wrepaint(w);
        wsetcursor(w, false);
    }
    ⟨wcurrent() wakeup w and oi 228c⟩
}
```

Uses input 61e, wrepaint() 107a, and wsetcursor() 91d.

⟨*function* wrepaint 107a⟩≡ (311)

```
void
wrepaint(Window *w)
{

    ⟨wrepaint() update cols 164d⟩
    ⟨wrepaint() after updated cols, redraw content if mouse not opened 202b⟩

    if(w == input){
        wborder(w, Selborder);
        wsetcursor(w, false);
    }else
        wborder(w, Unselborder);
}
```

Uses Selborder 73d, Unselborder 107b, input 61e, wborder() 99c, and wsetcursor() 91d.

⟨*constant* Unselborder 107b⟩≡ (299a)

```
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 `notefd`) and starts a timeout proc. If the process exits cleanly, the window transitions from Deleted to Exited and the `winctl` thread terminates. If the process does not exit within the timeout, `deletethread()`[109e] forces cleanup. This careful approach avoids blocking the window manager thread while waiting for a potentially unresponsive process.

⟨button3menu() *cases* 107c⟩+≡ (93b) ◁97a 112c▷

```
case Delete:
    delete();
    break;
```

Uses Delete-98 94d and delete() 108a.

7.7.1 delete()

`delete()` is the menu handler: it calls `pointto()`[112a] with `wait=true` (wait for the user to complete the click), then sends a `Deleted` control message to the selected window's `winctl` thread. The actual cleanup is

asynchronous—the `winctl` thread handles the message by sending a hangup note to the process group and starting a timeout.

⟨*function* `delete` 108a⟩≡ (308)
```
void
delete(void)
{
    Window *w;

    w = pointto(true);
    if(w)
        wsendctlmesg(w, Deleted, ZR, nil);
}
```
Uses `Deleted` 108b, `pointto()` 112a, and `wsendctlmesg()` 96c.

⟨`Wctlmesgkind` *cases* 108b⟩≡ (96a) 111d ▷
```
Deleted,
```

We have already seen `deleted` used as a guard in `winctl()`[78] (to skip `flushimage`) and in `wkeyctl()`[79e] (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* 108c⟩+≡ (59) ◁ 62f 149b ▷
```
bool deleted;
```

The `Deleted` handler writes "hangup" to `notefd` (`/proc/<pid>/notepg`), which sends a note (Plan 9's equivalent of a signal) to the entire process group. Then it kicks off `deletetimeoutproc` as a safety net and calls `wclosewin()`[108f] to remove the window from the screen immediately. The window's thread does not exit yet—it waits for the `Exited` message that comes later when all filesystem references are closed.

⟨`wctlmesg()` *cases* 108d⟩≡ (96d) 111e ▷
```
case Deleted:
    ⟨wctlmesg() break if window was deleted 108e⟩
    write(w->notefd, "hangup", 6);
    proccreate(deletetimeoutproc, estrdup(w->name), 4096);
    wclosewin(w);
    break;
```
Uses `Deleted` 108b, `deletetimeoutproc()` 109a, `estrdup()` 294a, and `wclosewin()` 108f.

⟨`wctlmesg()` *break if window was deleted* 108e⟩≡ (278c 150a 108d)
```
if(w->deleted)
    break;
```

`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` array (in case it was hidden), removes it from the global `windows`[61a] array with `memmove`, and frees the layer image so it disappears from the screen. Note that the `Window` struct itself is not freed here—that happens later in the `Exited` handler, after the child process has terminated.

⟨*function* `wclosewin` 108f⟩≡ (311)
```
void
wclosewin(Window *w)
{
    int i;

    w->deleted = true;

    if(w == input){
```

```
        input = nil;
        wsetcursor(w, false);
    }
    ⟨wclosewin() if wkeyboard 274e⟩
    ⟨wclosewin() remove w from hidden 120e⟩

    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()` 292c, `input` 61e, `nwindow` 61b, `windows` 61a, and `wsetcursor()` 91d.

7.7.2 `deletetimeoutproc()` and `deletethread()`

The timeout mechanism handles clients that hold a reference to the window image (via `nameimage()`) and refuse to let go. `deletetimeoutproc` is a separate process (not a thread) because `sleep` is blocking. After 750ms it sends the window name on `deletechan`. The `deletethread` 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.

⟨*function* `deletetimeoutproc` 109a⟩≡ (306b)

```
void
deletetimeoutproc(void *v)
{
    char *s = v;

    sleep(750); /* remove window from screen after 3/4 of a second */
    sendp(deletechan, s);
}
```

Uses `deletechan` 109b.

⟨*global* `deletechan` 109b⟩≡ (304)

```
// chan<string> (listener = deletethread, sender = deletetimeoutproc)
Channel* deletechan;
```

⟨`main()` *communication channels creation* 109c⟩+≡ (66a) ◁68c 110b▷

```
deletechan   = chancreate(sizeof(char*), 0);
```

Uses `deletechan` 109b.

⟨`main()` *threads creation* 109d⟩+≡ (66a) ◁68e 110c▷

```
threadcreate(deletethread, nil, STACK);
```

Uses `STACK` 68d and `deletethread()` 109e.

⟨*function* `deletethread` 109e⟩≡ (307b)

```
/* 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` 109b.

7.7.3 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()`[126d]), and each clunk decrements it (via `winclosechan`[110a]). When the last reference drops to zero, `wclose()`[111a] sends the `Exited` message, which triggers the final cleanup: freeing the frame, closing the note file descriptor, freeing all channels, and freeing the `Window` 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` calls `wclose()` in the correct context.

⟨*global* `winclosechan` 110a⟩≡ (304)

```
// chan<ref<Window>> (listener = winclosethread, sender = filsyswalk | filsysclunk )
Channel *winclosechan; /* chan(Window*); */
```

⟨`main()` *communication channels creation* 110b⟩+≡ (66a) ◁109c

```
winclosechan = chancreate(sizeof(Window*), 0);
```

Uses `winclosechan` 110a.

⟨`main()` *threads creation* 110c⟩+≡ (66a) ◁109d 279f▷

```
threadcreate(winclosethread, nil, STACK);
```

Uses `STACK` 68d and `winclosethread()` 110d.

⟨*function* `winclosethread` 110d⟩≡ (307b)

```
/* 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()` 111a and `winclosechan` 110a.

⟨*function* wclose 111a⟩≡ (311)
```
bool
wclose(Window *w)
{
    int i;

    i = decref(w);
    if(i > 0)
        return false;
    ⟨wclose() sanity check i 111b⟩
    ⟨wclose() sanity check w 111c⟩

    wsendctlmesg(w, Exited, ZR, nil);
    return true;
}
```
Uses Exited 111d and wsendctlmesg() 96c.

⟨wclose() *sanity check i* 111b⟩≡ (111a)
```
if(i < 0)
    error("negative ref count");
```
Uses error() 292c.

⟨wclose() *sanity check w* 111c⟩≡ (111a)
```
if(!w->deleted)
    wclosewin(w);
```
Uses wclosewin() 108f.

⟨Wctlmesgkind *cases* 111d⟩+≡ (96a) ◁108b 141a▷
```
Exited,
```

⟨wctlmesg() *cases* 111e⟩+≡ (96d) ◁108d 115d▷
```
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 111d.

7.7.4 Mouse action pointto()

pointto() is the shared "pick a window" gesture used by Delete, Move, and Resize. 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()`[113b] phase.

⟨*function* `pointto` 112a⟩≡ (308)
```
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 112b⟩
            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()` 90c, `input` 61e, `menuing` 104c, `mouse` 58a, `mousectl` 57b, `riosetcursor()` 90b, `sightcursor` 87b, and `wpointto()` 76a.

⟨`pointto()` *cancel pointto if clicked another button* 112b⟩≡ (112a)
```
if(mouse->buttons!=4 && w != nil){ /* cancel */
    cornercursor(input, mouse->xy, false);
    w = nil;
}
```
Uses `cornercursor()` 90c, `input` 61e, and `mouse` 58a.

7.8 Window move

Moving a window is a two-step mouse action: first `pointto()`[112a] lets the user select which window to move (by showing a gunsight cursor), then `drag()`[113b] 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* 112c⟩+≡ (93b) ◁107c 115b▷
```
case Move:
    move();
    break;
```
Uses `Move-97` 94d and `move()` 113a.

7.8.1 move()

The Moved case in wctlmesg shares its handler with Reshaped—a move is just a reshape where the dimensions stay the same and the old content is copied over.

⟨*function* move 113a⟩≡ (308)
```
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 96a, cornercursor() 90c, drag() 113b, input 61e, mouse 58a, pointto() 112a, and wsendctlmesg() 96c.

7.8.2 Mouse action drag()

drag() 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()[114b] 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 113b⟩≡ (308)
```
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 114a⟩
    draw(ni, ni->r, i, nil, i->r.min);
    *rp = r;
    return ni;
}
```

Uses `boxcursor` 87a, `cornercursor()` 90c, `desktop` 58b, `drawborder()` 114b, `menuing` 104c, `mouse` 58a, `mousectl` 57b, and `riosetcursor()` 90b.

⟨`drag()` *sanity check mouse buttons and ni* 114a⟩≡ (113b)

```
if(mouse->buttons!=0 || ni==nil){
    moveto(mousectl, om);
    while(mouse->buttons)
        readmouse(mousectl);
    *rp = Rect(0, 0, 0, 0);
    return nil;
}
```

Uses `mouse` 58a and `mousectl` 57b.

`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()`[115a] 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` 114b⟩≡ (308)

```
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()` 115a.

⟨*function* drawedge 115a⟩≡ (308)

```
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` 58b.

7.9 Window resize

Resizing reuses `sweep()`[104d] to let the user draw a new rectangle, then sends a `Reshaped` message. In the `wctlmesg` handler, Moved and Reshaped share the same code path: `wresize()`[116b] replaces the window's image, re-publishes the name (so the client can reconnect via `getwindow()`), and triggers a reflow of the text content for textual windows.

⟨`button3menu()` *cases* 115b⟩+≡ (93b) ◁112c 119a▷

```
case Reshape:
    resize();
    break;
```

Uses `Reshape-96` 94d and `resize()` 115c.

7.9.1 resize()

`resize()` is almost identical to `move()`[113a]: select a window with `pointto()`[112a], draw a new rectangle with `sweep()`[104d], and send a `Reshaped` 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 115c⟩≡ (308)

```
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` 96a, `pointto()` 112a, `sweep()` 104d, and `wsendctlmesg()` 96c.

⟨`wctlmesg()` *cases* 115d⟩+≡ (96d) ◁111e 141b▷

```
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` 96a, `Reshaped` 96a, `deletetimeoutproc()` 109a, `estrdup()` 294a, `input` 61e, `wcurrent()` 106e, and `wresize()` 116b.

⟨`Window` *other fields* 116a⟩+≡ (59) ◁100b 139a▷
```
  bool   resized;
```

`wresize()`[116b] 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()`[104a] publishes the new image under an incremented name so the client can reconnect via `getwindow()`. The `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` 116b⟩≡ (311)
```
  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;
      wsetname(w); // publish new window name by incrementing namecount
      w->mc.image = i;

      ⟨wresize() textual window updates 221⟩

      wborder(w, Selborder);
      w->topped = ++topped;

      w->resized = true;
      w->mouse.counter++;
  }
```
Uses `Selborder` 73d, `topped-65` 61c, `wborder()` 99c, and `wsetname()` 104a.

7.9.2 Mouse action `bandsize()`

`bandsize()` 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()`[91a], snaps the cursor to that corner with `cornerpt()`[118d] and `wmovemouse()`[118b], then enters a tracking loop where `whichrect()`[118c] 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. 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.

⟨*function* `bandsize` 117a⟩≡ (308)

```
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 117b⟩
    i = allocwindow(desktop, or, Refbackup, DWhite);
    ⟨bandsize() sanity check i 118a⟩
    border(i, r, Selborder, red, ZP);
    return i;
}
```

Uses `Selborder` 73d, `cornerpt()` 118d, `desktop` 58b, `drawborder()` 114b, `goodrect()` 239a, `mouse` 58a, `mousectl` 57b, `onscreen()` 105a, `red` 58d, `whichcorner()` 91a, `whichrect()` 118c, `wmovemouse()` 118b, and `wsetcursor()` 91d.

⟨`bandsize()` *sanity check mouse buttons, rectanglr* `or`, *point* `p` 117b⟩≡ (117a)

```
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` 58a and `mousectl` 57b.

⟨`bandsize()` *sanity check i* 118a⟩≡ (117a)

```
  if(i == nil)
      return nil;
```

⟨*function* `wmovemouse` 118b⟩≡ (311)

```
  /*
   * 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` 57b.

`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` 118c⟩≡ (308)

```
  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` 118d⟩≡ (308)

```
  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

Hiding a window is an elegant trick: the window's on-screen layer image is replaced with a plain in-memory image (via `allocimage`, not `allocwindow`). Since it is no longer a layer, it does not appear on screen, but the window's content is preserved. The hidden window is added to the `hidden` array and appears as an extra entry in the right-click system menu so the user can unhide it later.

⟨`button3menu()` *cases* 119a⟩+≡ (93b) ◁115b 121b▷

```
case Hide:
    hide();
    break;
```

Uses `Hide-99` 94d and `hide()` 119b.

7.10.1 hide()

⟨*function* `hide` 119b⟩≡ (308)

```
void
hide(void)
{
    Window *w;

    w = pointto(true);
    if(w == nil)
        return;
    whide(w);
}
```

Uses `pointto()` 112a and `whide()` 120c.

7.10.2 hidden

The **hidden** 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's `label` (typically `rc <pid>`).

⟨*global* `hidden` 120a⟩≡ (304)
```
// array<ref<Window>> (size valid = nhidden)
Window *hidden[100];
```

⟨*global* `nhidden` 120b⟩≡ (304)
```
int  nhidden;
```

The hiding trick is in the choice of `allocimage` versus `allocwindow`: `whide()` allocates a plain in-memory image (not a layer on the desktop) and sends a `Reshaped` message with it. The `Reshaped` handler in `wctlmesg` calls `wresize()`[116b], 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` 120c⟩≡ (308)
```
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` 96a, `hidden` 120a, `nhidden` 120b, and `wsendctlmesg()` 96c.

⟨`new()` *if hideit* 120d⟩≡ (97b)
```
if(hideit){
    hidden[nhidden++] = w;
    w->screenr = ZR;
}
```
Uses `hidden` 120a and `nhidden` 120b.

⟨`wclosewin()` *remove w from hidden* 120e⟩≡ (108f)
```
// 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` 120a and `nhidden` 120b.

⟨button3menu() *menu3str adjustments with hidden windows* 121a⟩≡ (93b)
```
for(i=0; i<nhidden; i++)
    menu3str[i+Hidden] = hidden[i]->label;
menu3str[i+Hidden] = nil;
```
Uses `Hidden-101` 94d, `hidden` 120a, `menu3str` 94c, and `nhidden` 120b.

7.10.3 unhide()

Unhiding reverses the trick: `wunhide()`[121e] allocates a real layer (via `allocwindow`) and sends a `Reshaped` message. The `Reshaped` handler swaps the off-screen image for the new on-screen layer, publishes it via `wsetname()`[104a], and the window reappears. The menu integration is handled by the `default` case in the `button3menu` switch—any menu index past the standard entries corresponds to a hidden window.

⟨button3menu() *cases* 121b⟩+≡ (93b) ◁119a
```
default:
    unhide(i);
    break;
```
Uses `unhide()` 121c.

⟨*function* unhide 121c⟩≡ (308)
```
void
unhide(int h)
{
    Window *w;

    h -= Hidden;
    w = hidden[h];
    ⟨unhide() sanity check w 121d⟩
    wunhide(h);
}
```
Uses `Hidden-101` 94d, `hidden` 120a, and `wunhide()` 121e.

⟨unhide() *sanity check w* 121d⟩≡ (121c)
```
if(w == nil)
    return;
```

⟨*function* wunhide 121e⟩≡ (308)
```
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` 96a, `desktop` 58b, `hidden` 120a, `nhidden` 120b, and `wsendctlmesg()` 96c.

Chapter 8

Filesystem Server

Until now I have presented 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` which links the fid to the appropriate `Window`), `walk` (navigating to a file), `open`, `read`, `write`, `stat`, and `clunk`. Each operation has two layers: 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).

8.1 Additional data structures

8.1.1 Qxxx

Each virtual file under `/mnt/wsys/` is identified by a `Qid` whose path encodes both the window ID and the file type (`Qxxx`). The `QID`, `WIN`, and `FILE` 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.

⟨*function* `QID` 122a⟩≡ (299b)

```
#define QID(winid,qxxx) ((winid<<8)|(qxxx))
```

⟨*function* `WIN` 122b⟩≡ (299b)

```
#define WIN(q) ((((ulong)(q).path)>>8) & 0xFFFFFF)
```

⟨*function* `FILE` 122c⟩≡ (299b)

```
#define FILE(q) (((ulong)(q).path) & 0xFF)
```

⟨*enum* `qid` 122d⟩≡ (299b)

```
enum Qxxx
{
    Qdir,   /* /dev for this window */
    ⟨qid cases 138a⟩

    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)
```

Recall that a `Fid`[63c] is 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. Each `Fid` carries a `qid` field that identifies the actual file: a path like `/mnt/wsys/cons` is resolved by `walk` into a `Fid` whose `qid.path` encodes the window identifier and the device type (`Qcons`). 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.

8.1.2 dirtab

The `Dirtab` structure is a static directory template: it maps a file name (e.g., `"cons"`, `"mouse"`) to its `Qxxx` enum value and permissions. The global `dirtab` 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⟩≡ (299b)

```
struct Dirtab
{
    char *name;
    // bitset<enum<QTxxx>>
    byte type;
    // enum<Qxxx>
    uint qid;
    uint perm;
};
```

⟨*global* dirtab 123b⟩≡ (317)

```
Dirtab dirtab[]=
{
    { ".",  QTDIR, Qdir,  0500|DMDIR },
    ⟨dirtab array elements 138b⟩
    { nil, }
};
```

Uses `Qdir` 122d.

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 `dir` field, so once a file is walked to, its metadata is readily available.

⟨`Fid` *other fields* 123c⟩+≡ (63c) ◁64a 144c▷

```
// ref<Dirtab>
Dirtab *dir;
```

8.2 filsysrespond()

Before examining the individual 9P operations, 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 `fs->sfd`, the server side of the pipe. Even though multiple threads may call `filsysrespond` concurrently, the kernel guarantees that writes of complete messages to a pipe are atomic, and the tag/fid fields let the kernel match replies to their requests.

⟨*function* `filsysrespond` 124a⟩≡ (316a)
```
Xfid*
filsysrespond(Filsys *fs, Xfid *x, Fcall *fc, char *err)
{
    int n;

    ⟨filsysrespond() if err 124d⟩
    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 124b⟩
    n = convS2M(fc, x->buf, messagesize);
    ⟨filsysrespond() sanity check n 124c⟩

    if(write(fs->sfd, x->buf, n) != n)
        error("write error in respond");
    ⟨filsysrespond() dump Fcall t if debug 289d⟩
    free(x->buf);
    x->buf = nil;
    return x;
}
```
Uses `error()` 292c and `messagesize` 81b.

⟨`filsysrespond()` *sanitize buf* 124b⟩≡ (124a)
```
if(x->buf == nil)
    x->buf = malloc(messagesize);
```
Uses `messagesize` 81b.

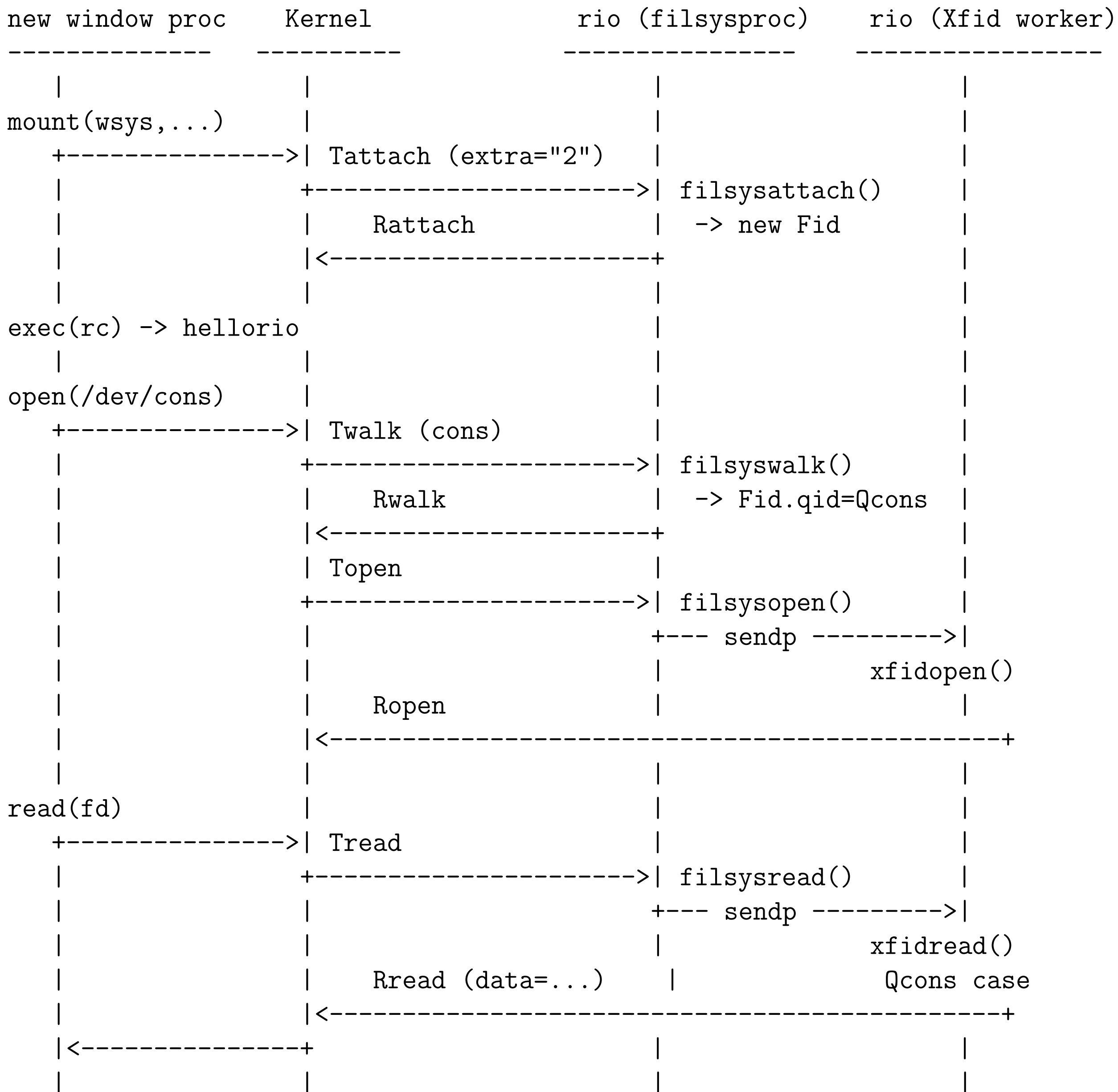
⟨`filsysrespond()` *sanity check n* 124c⟩≡ (124a)
```
if(n <= 0)
    error("convert error in convS2M");
```
Uses `error()` 292c.

⟨`filsysrespond()` *if err* 124d⟩≡ (124a)
```
if(err){
    fc->type = Rerror;
    fc->ename = err;
}
```

8.3 Attach

Attach is the first 9P operation a client performs after mounting. The mount spec string (e.g., "2") carries the window ID, which `xfidattach()`[126d] uses to look up the corresponding `Window`[59]. This associates the client's fid with a specific window, 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`), 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 the lightweight operations (`attach`, `walk`) are handled directly by `filsysproc`, while the heavier ones (`open`, `read`, `write`) are dispatched to `Xfid` worker threads via `sendp`.

8.3.1 filsysattach()

⟨*function* `filsysattach` 125⟩≡ (317)

```
static
Xfid*
filsysattach(Filsys *, Xfid *x, Fid *f)
{
    ⟨filsysattach() locals 126a⟩

    ⟨filsysattach() sanity check same user 126b⟩

    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; // entry for "."

    f->nrpart = 0;

    sendp(x->c, xfidattach);
    return nil;
}
```

Uses `Qdir` 122d, `dirtab` 123b, and `xfidattach()` 126d.

```
⟨filsysattach() locals 126a⟩≡                                         (125)
Fcall fc;
```

```
⟨filsysattach() sanity check same user 126b⟩≡                         (125)
if(strcmp(x->req.uname, x->fs->user) != 0)
    return filsysrespond(x->fs, x, &fc, Eperm);
```

Uses `Eperm` 290a and `filsysrespond()` 124a.

8.3.2 xfidattach()

```
⟨global all 126c⟩≡                                                    (304)
QLock all;   /* BUG */
```

```
⟨function xfidattach 126d⟩≡                                           (318)
void
xfidattach(Xfid *x)
{
    Fcall fc;
    int id;
    Window *w = nil;
    bool newlymade = false;
    ⟨xfidattach() other locals 127b⟩

    fc.qid = x->f->qid;
    qlock(&all);

    ⟨xfidattach() if mount "new ..." 235b⟩
    else{
        // mount(fs->cfd, ..., "/mnt/wsys", ..., "2"), winid
        id = atoi(x->req.aname);
        w = wlookid(id);
    }

    x->f->w = w;
    ⟨xfidattach() sanity check w 127c⟩
    if(!newlymade) /* counteract dec() in winshell() */
        incref(w);
    qunlock(&all);

    filsysrespond(x->fs, x, &fc, nil);
}
```

Uses `all` 126c, `filsysrespond()` 124a, and `wlookid()` 127a.

⟨*function* `wlookid` 127a⟩≡ (311)
```
Window*
wlookid(int id)
{
    int i;

    for(i=0; i<nwindow; i++)
        if(windows[i]->id == id)
            return windows[i];
    return nil;
}
```
Uses `nwindow` 61b and `windows` 61a.

⟨`xfidattach()` *other locals* 127b⟩≡ (126d) 235a ▷
```
char *err = Eunkid;
```
Uses `Eunkid` 291g.

⟨`xfidattach()` *sanity check w* 127c⟩≡ (126d)
```
if(w == nil){
    qunlock(&all);
    x->f->busy = false;
    filsysrespond(x->fs, x, &fc, err);
    return;
}
```
Uses `all` 126c and `filsysrespond()` 124a.

8.4 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`.

8.4.1 `filsyswalk()`

The walk function is the most intricate 9P operation because it must handle several cases in one routine: simple name lookup (scanning `dirtab` 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` 127d⟩≡ (317)
```
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 128b⟩
```

```
    ⟨filsyswalk() sanity check if opened 129d⟩
    ⟨filsyswalk() if clone walk message 129a⟩

    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 130a⟩
            ⟨filsyswalk() if dotdot 129c⟩
            ⟨filsyswalk() if Qwsys, then goto Accept 226d⟩
            ⟨filsyswalk() if snarf 248e⟩
            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 129e⟩
    }
    ⟨filsyswalk() sanity check err and nwqid 129f⟩
    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()` 124a.

⟨`filsyswalk()` *accept label and code* 128a⟩≡ (129c)

```
Accept:
⟨filsyswalk() sanity check path elements 130b⟩
qid.type = type;
qid.path = QID(id, path);
qid.vers = 0;
fc.wqid[fc.nwqid++] = qid;
continue;
```

Uses `QID` 122a.

8.4.2 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. 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* 128b⟩≡ (127d)

```
Fid *nf = nil;
```

⟨filsyswalk() *if clone walk message* 129a⟩≡ (127d)
```
if(x->req.fid  != x->req.newfid){
    /* BUG: check exists */
    nf = newfid(fs, x->req.newfid);
    ⟨filsyswalk() when clone walk message, sanity check nf 129b⟩
    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()` 63f.

⟨filsyswalk() *when clone walk message, sanity check nf* 129b⟩≡ (129a)
```
if(nf->busy)
    return filsysrespond(fs, x, &fc, "clone to busy fid");
```
Uses `filsysrespond()` 124a.

8.4.3 ..

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* 129c⟩≡ (127d)
```
if(strcmp(x->req.wname[i], "..") == 0){
    type = QTDIR;
    path = Qdir;
    dir = dirtab;
    ⟨filsyswalk() when in dotdot, if Qwsysdir adjust path 227b⟩
    id = 0;
    ⟨filsyswalk() accept label and code 128a⟩
}
```
Uses `Qdir` 122d and `dirtab` 123b.

8.4.4 Error management

⟨filsyswalk() *sanity check if opened* 129d⟩≡ (127d)
```
if(f->open)
    return filsysrespond(fs, x, &fc, "walk of open file");
```
Uses `filsysrespond()` 124a.

⟨filsyswalk() *sanity check i and err* 129e⟩≡ (127d)
```
if(i==0 && err==nil)
    err = Eexist;
```
Uses `Eexist` 290b.

⟨filsyswalk() *sanity check err and nwqid* 129f⟩≡ (127d)
```
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` 110a.

⟨filsyswalk() *sanity check current qid is a directory* 130a⟩≡ (127d)
```
if(!(qid.type & QTDIR)){
    err = Enotdir;
    break;
}
```
Uses Enotdir 290c.

⟨filsyswalk() *sanity check path elements* 130b⟩≡ (128a)
```
if(i == MAXWELEM){
    err = "name too long";
    break;
}
```

8.5 Open

Open illustrates the two-layer pattern clearly. The `filsysopen()` fast path only validates permissions—it checks the requested mode against the `dirtab` 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 `mouseopen` or `ctlopen`) runs in the `xfidopen()` worker thread, which has access to the window state.

8.5.1 filsysopen()

⟨*function* filsysopen 130c⟩≡ (317)
```
static
Xfid*
filsysopen(Filsys *fs, Xfid *x, Fid *f)
{
    ⟨filsysopen() locals 130d⟩

    ⟨filsysopen() sanity check mode 130e⟩
    sendp(x->c, xfidopen);
    return nil;
⟨filsysopen() deny label 131c⟩
}
```
Uses xfidopen() 131d.

⟨filsysopen() *locals* 130d⟩≡ (130c) 131b ▷
```
int m;
```

⟨filsysopen() *sanity check mode* 130e⟩≡ (130c)
```
⟨filsysopen() sanitize mode 131a⟩
/* 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* 131a⟩≡ (130e)

```
    /* can't truncate anything, so just disregard */
    x->req.mode &= ~(OTRUNC|OCEXEC);
```

⟨filsysopen() *locals* 131b⟩+≡ (130c) ◁130d

```
    Fcall fc;
```

⟨filsysopen() *deny label* 131c⟩≡ (130c)

```
    Deny:
        return filsysrespond(fs, x, &fc, Eperm);
```

Uses Eperm 290a and filsysrespond() 124a.

8.5.2 xfidopen()

⟨*function* xfidopen 131d⟩≡ (318)

```
    void
    xfidopen(Xfid *x)
    {
        Fcall fc;
        Window *w = x->f->w;

        ⟨xfidxxx() respond error if window was deleted 131e⟩
        switch(FILE(x->f->qid)){
        ⟨xfidopen() cases 145h⟩
        }

        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() 124a, and messagesize 81b.

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.

⟨xfidxxx() *respond error if window was deleted* 131e⟩≡ (135a 134a 131d)

```
    if(w->deleted){
        filsysrespond(x->fs, x, &fc, Edeleted);
        return;
    }
```

Uses Edeleted 291a and filsysrespond() 124a.

8.6 Clunk/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.

8.6.1 filsysclunk()

⟨*function* `filsysclunk` 132a⟩≡ (317)

```
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()` 124a, `winclosechan` 110a, and `xfidclose()` 132b.

8.6.2 xfidclose()

⟨*function* `xfidclose` 132b⟩≡ (318)

```
void
xfidclose(Xfid *x)
{
    Fcall fc;
    Window *w = x->f->w;
    ⟨xfidclose() other locals 249a⟩

    switch(FILE(x->f->qid)){
    ⟨xfidclose() cases 146a⟩
    }
    wclose(w);
    filsysrespond(x->fs, x, &fc, nil);
}
```

Uses `FILE` 122c, `filsysrespond()` 124a, and `wclose()` 111a.

8.7 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` context by iterating over `dirtab`—no window state is needed. But reading a device file like `/dev/mouse` or `/dev/cons` requires blocking until data is available, so it must be dispatched to an `xfidread()` worker thread.

8.7.1 filsysread()

⟨*function* filsysread 133a⟩≡ (317)
```
static
Xfid*
filsysread(Filsys *fs, Xfid *x, Fid *f)
{
    int o, e;
    uint clock;
    byte *b;
    int n;
    Fcall fc;
    ⟨filsysread() other locals 133c⟩

    if(!(f->qid.type & QTDIR)){
        sendp(x->c, xfidread);
        return nil;
    }
    // else, a directory

    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 133b⟩

    n = 0;
    switch(FILE(f->qid)){
    ⟨filsysread() cases 133d⟩
    }

    fc.data = (char*)b;
    fc.count = n;
    filsysrespond(fs, x, &fc, nil);
    free(b);
    return x;
}
```
Uses FILE 122c, filsysrespond() 124a, getclock() 279d, messagesize 81b, and xfidread() 134a.

⟨filsysread() *sanity check b* 133b⟩≡ (133a)
```
if(b == nil)
    return filsysrespond(fs, x, &fc, "out of memory");
```
Uses filsysrespond() 124a.

8.7.2 Reading a directory

To read a directory, the server iterates over `dirtab` and calls `dostat()` on each entry, which serializes a `Dir` structure into the 9P wire format via `convD2M`. 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* 133c⟩≡ (133a) 226c ▷
```
Dirtab *d;
```

⟨filsysread() *cases* 133d⟩≡ (133a) 226f ▷
```
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 227a, WIN 122b, dirtab 123b, and dostat() 136a.

8.7.3 xfidread()

⟨*function* xfidread 134a⟩≡ (318)

```
void
xfidread(Xfid *x)
{
    uint qid;
    int off, cnt;
    Fcall fc;
    Fcall *req = &x->req;
    Window *w = x->f->w;
    ⟨xfidread() other locals 139e⟩

    ⟨xfidxxx() respond error if window was deleted 131e⟩
    qid = FILE(x->f->qid);
    off = req->offset;
    cnt = req->count;

    switch(qid){
    ⟨xfidread() cases 139f⟩
    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() 124a.

As with `xfidopen()`, 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.

8.8 Write

Write is the simplest 9P operation at the `filsys` layer: since there is no way to write to a directory (the kernel rejects that before it reaches the server), `filsyswrite()` can unconditionally dispatch to the `xfidwrite()` 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()`.

8.8.1 filsyswrite()

⟨*function* filsyswrite 134b⟩≡ (317)

```
static
Xfid*
```

```
filsyswrite(Filsys*, Xfid *x, Fid*)
{
    sendp(x->c, xfidwrite);
    return nil;
}
```
Uses `xfidwrite()` 135a.

8.8.2 xfidwrite()

⟨*function* `xfidwrite` 135a⟩≡ (318)
```
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 140a>

    <xfidxxx() respond error if window was deleted 131e>
    qid = FILE(x->f->qid);
    cnt = req->count;
    off = req->offset;
    req->data[cnt] = '\0';

    switch(qid){
    <xfidwrite() cases 140b>
    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()` 124a.

8.9 Stats

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` entry and the window identifier, since none of these virtual files exist on disk.

8.9.1 filsysstat()

⟨*function* `filsysstat` 135b⟩≡ (317)
```
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() 136a, emalloc() 293d, filsysrespond() 124a, getclock() 279d, and messagesize 81b.

⟨*function* dostat 136a⟩≡ (317)

```
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 249f⟩
    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.

8.10 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* 136b⟩+≡ (65) ◁83a 282d▷

```
[Tcreate]  = filsyscreate,
[Tremove]  = filsysremove,
[Twstat]   = filsyswstat,
```

Uses filsyscreate() 136c, filsysremove() 137a, and filsyswstat() 137b.

⟨*function* filsyscreate 136c⟩≡ (317)

```
static
Xfid*
filsyscreate(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, Eperm);
}
```

Uses Eperm 290a and filsysrespond() 124a.

⟨*function* `filsysremove` 137a⟩≡ (317)

```
static
Xfid*
filsysremove(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, Eperm);
}
```

Uses `Eperm` 290a and `filsysrespond()` 124a.

⟨*function* `filsyswstat` 137b⟩≡ (317)

```
static
Xfid*
filsyswstat(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, Eperm);
}
```

Uses `Eperm` 290a and `filsysrespond()` 124a.

Chapter 9

Virtual Devices

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` thread offers data when it has events to report) and the consumer side (where the `xfidread` worker thread blocks 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 they take very different routes to it. `X11` clients connect to the display server over a Unix socket (`/tmp/.X11-unix/X0`) or TCP and speak the X protocol; input events arrive as binary protocol messages (`KeyPress`, `ButtonPress`, `MotionNotify`) read from the socket. `Wayland` is similar but with a different protocol (`wl_seat`, `wl_pointer`, `wl_keyboard` interfaces) and a different socket (`$XDG_RUNTIME_DIR/wayland-0`). macOS's Quartz talks to `WindowServer` via Mach ports and delivers events as `CGEventRef` structs through run-loop callbacks. Win32 applications receive events as `WM_MOUSEMOVE`, `WM_KEYDOWN`, etc. on a `WndProc` message queue. Every one of these requires a dedicated client library (`Xlib/xcb`, `libwayland-client`, `AppKit`, `user32.dll`) that knows the protocol.

`rio` does it with no protocol and no client library at all. The application just `open`s `/dev/mouse` or `/dev/cons` and `read`s textual records; 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" property from Section 2.1.8.

9.2 `/mnt/wsys/mouse`

We can now see the `QidX` and `dirtab`[123b] entries for each virtual device file, along with their `xfidread()`[134a] and `xfidwrite()`[135a] 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.

⟨`qid` *cases* 138a⟩≡ (122d) 141c ▷

```
  Qmouse,
```

⟨*dirtab array elements* 138b⟩≡ (123b) 141d ▷

```
  { "mouse",  QTFILE, Qmouse,  0600 },
```

Uses `Qmouse` 138a.

9.2.1 Reading part1

The mouse virtual device uses a "channel of channels" pattern. Consider a concrete scenario: the user moves the mouse rapidly, generating positions (100,50), (120,55), (140,60), (160,65). The `winctl`[78] thread sees all four events via `mousethread`[72c], 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)
 +-----------------+                 +-------------------+
 |                 |  Mousereadmesg  |  (has seen 4      |
 | client calls    |  {cm: replyCh}  |   mouse events,   |
 | read("/dev/     |                 |   current pos is  |
 |      mouse")    +---------------->|   now 160,65)     |
 |                 |  w->mouseread   |                   |
 |                 |                 |                   |
 |                 | Mouse{160,65}   | send(mrm.cm,      |
 | recv(replyCh,   |<----------------+   &currentMouse)  |
 |      &ms)       |  replyCh        |                   |
 +-----------------+                 +-------------------+
```

The first `send` on `w->mouseread` is really a synchronization message saying "I am ready to receive." Only then does `winctl` snapshot the freshest mouse state and send it back on the reply channel `mrm.cm`. The client gets (160,65)—the position that matters—instead of a stale queued value.

⟨Window *other fields* 139a⟩+≡ (59) ◁116a 141e▷
```
// chan<chan<Mouse> > (listener = xfidread(Qmouse), sender = winctl)
Channel  *mouseread; /* chan(Mousereadmesg) */
```

⟨wmk() *channels creation* 139b⟩≡ (98e) 142a▷
```
w->mouseread =  chancreate(sizeof(Mousereadmesg), 0);
```

⟨*struct* Mousereadmesg 139c⟩≡ (299b)
```
struct Mousereadmesg
{
    // chan<Mouse> (listener = xfidread(Qmouse), sender = winctl)
    Channel *cm;  /* chan(Mouse) */
};
```

⟨*enum* _anon_ (windows/rio/xfid.c)4 139d⟩≡ (318)
```
enum { MRdata, MRflush, NMR };
```

⟨xfidread() *other locals* 139e⟩≡ (134a) 142e▷
```
Alt alts[NCR+1];
Mousereadmesg mrm;
Mouse ms;
int n, c;
```
Uses NCR-9 142d.

⟨xfidread() *cases* 139f⟩≡ (134a) 142f▷
```
case Qmouse:
    ⟨xfidxxx() set flushtag 283c⟩

    alts[MRdata].c = w->mouseread;
    alts[MRdata].v = &mrm;
    alts[MRdata].op = CHANRCV;
    ⟨xfidread() when Qmouse, set alts for flush 283e⟩
```

```
    alts[NMR].op = CHANEND;

    switch(alt(alts)){
    case MRdata:
        break;
    ⟨xfidread() when Qmouse, switch alt flush case 283f⟩
    }
    /* received data */
    ⟨xfidxxx() unset flushtag 283d⟩
    ⟨xfidread() when Qmouse, if flushing 284a⟩

    qlock(&x->active);
    recv(mrm.cm, &ms);
    c = 'm';
    ⟨xfidread() when Qmouse, adjust c for resize message if resized 156⟩
    n = sprint(buf, "%c%11d %11d %11d %11ld ", 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-10` 139d, `NMR-12` 139d, `Qmouse` 138a, `filsysrespond()` 124a, and `min()` 293a.

9.2.2 Writing

Writing to `/dev/mouse` lets a program warp 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` with `Movemouse`, so it goes through the `winctl` thread and respects the `sweeping`[94a] guard—you cannot warp the mouse while the user is in the middle of a window management operation.

⟨`xfidwrite()` *other locals* 140a⟩≡ (135a) 143e ▷

```
  char *p;
  Point pt;
```

⟨`xfidwrite()` *cases* 140b⟩≡ (135a) 144b ▷

```
  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` 291k, `Eshort` 291e, `Movemouse` 141a, `Qmouse` 138a, `filsysrespond()` 124a, `input` 61e, `mouse` 58a, `wpointto()` 76a, and `wsendctlmesg()` 96c.

⟨Wctlmesgkind *cases* 141a⟩+≡ (96a) ◁111d 149e▷

```
  Movemouse,
```

⟨wctlmesg() *cases* 141b⟩+≡ (96d) ◁115d 150a▷

```
  case Movemouse:
      if(sweeping || !ptinrect(r.min, w->i->r))
          break;
      wmovemouse(w, r.min);
      break;
```

Uses Movemouse 141a, sweeping 94a, and wmovemouse() 118b.

9.3 /mnt/wsys/cons

The console device is the most complex virtual device because it mediates between two asynchronous worlds: the client process reading/writing bytes, and the winctl thread managing runes.

⟨qid *cases* 141c⟩+≡ (122d) ◁138a 145e▷

```
  Qcons,
```

⟨*dirtab array elements* 141d⟩+≡ (123b) ◁138b 145f▷

```
  { "cons",  QTFILE, Qcons,  0600 },
```

Uses Qcons 141c.

9.3.1 Reading part1

Unlike the mouse, the console uses two channels inside the Consreadmesg[142b]: c1 to pass the read buffer from the worker to winctl[78], 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):

```
  xfidread (worker)                  winctl (window thread)
  +------------------+               +--------------------+
  |                  |  Consreadmesg  |                    |
  | send(consread,   |  {c1, c2}     |                    |
  |   &crm)          +--------------->| recv(consread,     |
  |                  |  w->consread   |        &crm)       |
  |                  |               |                    |
  | t = malloc(cnt)  |  Stringpair    |                    |
  | send(c1,         |  {t, cnt}      |                    |
  |  {t, cnt})       +--------------->| recv(c1, &pair)    |
  |                  |               | (fill t with       |
  |                  |  Stringpair    |  typed runes,      |
  |                  |  {t, actual_n} |  converted to      |
  | recv(c2, &pair)  |<---------------+  UTF-8 bytes)      |
  |                  |               | send(c2, &pair)    |
  +------------------+               +--------------------+
```

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.

⟨Window *other fields* 141e⟩+≡ (59) ◁139a 143a▷

```
  // chan<Consreadmesg> (listener = xfidread(Qcons), sender = winctl)
  Channel  *consread; /* chan(Consreadmesg) */
```

⟨wmk() *channels creation* 142a⟩+≡ (98e) ◁139b 143b▷

```
w->consread =  chancreate(sizeof(Consreadmesg), 0);
```

⟨*struct* Consreadmesg 142b⟩≡ (299b)

```
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) */
};
```

⟨*struct* Stringpair 142c⟩≡ (299b)

```
struct Stringpair /* rune and nrune or byte and nbyte */
{
    void *s;
    int  ns;
};
```

⟨*enum* _anon_ (windows/rio/xfid.c)*3* 142d⟩≡ (318)

```
enum { CRdata, CRflush, NCR };
```

⟨xfidread() *other locals* 142e⟩+≡ (134a) ◁139e 157c▷

```
Consreadmesg crm;
Channel *c1, *c2; /* chan (tuple(char*, int)) */
char *t;
Stringpair pair;
```

⟨xfidread() *cases* 142f⟩+≡ (134a) ◁139f 146f▷

```
case Qcons:
    ⟨xfidxxx() set flushtag 283c⟩

    alts[CRdata].c = w->consread;
    alts[CRdata].v = &crm;
    alts[CRdata].op = CHANRCV;
    ⟨xfidread() when Qcons, set alts for flush 284b⟩
    alts[NCR].op = CHANEND;

    switch(alt(alts)){
    case CRdata:
        break;
    ⟨xfidread() when Qcons, switch alt flush case 284c⟩
    }
    /* received data */
    ⟨xfidxxx() unset flushtag 283d⟩

    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 284d⟩

    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-7` 142d, `NCR-9` 142d, `Qcons` 141c, and `filsysrespond()` 124a.

9.3.2 Writing part1

Console writing follows the same channel handshake as reading, but the data flows in the opposite direction: the client's bytes go from `xfidwrite()`[135a] to `winctl`[78]. 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`[143c] first (signaling it is ready to accept output), and then the worker sends the rune data through the `cw` channel.

⟨`Window` *other fields* 143a⟩+≡ (59) ◁141e 145g▷
```
  // chan<Conswritemesg> (listener = xfidwrite(Qcons), sender = winctl)
  Channel  *conswrite; /* chan(Conswritemesg) */
```

⟨`wmk()` *channels creation* 143b⟩+≡ (98e) ◁142a 229e▷
```
  w->conswrite = chancreate(sizeof(Conswritemesg), 0);
```

⟨*struct* `Conswritemesg` 143c⟩≡ (299b)
```
  struct Conswritemesg
  {
      // chan<ref<array<Rune>> (listener = winctl, sender = xfidwrite(Qcons))
      Channel *cw;  /* chan(Stringpair) */
  };
```

⟨*enum* `_anon_ (windows/rio/xfid.c)`*2* 143d⟩≡ (318)
```
  enum { CWdata, CWflush, NCW };
```

⟨`xfidwrite()` *other locals* 143e⟩+≡ (135a) ◁140a 144a▷
```
  Rune *r;
  int nr; // nb runes
  int nb; // nb bytes
```

```
⟨xfidwrite() other locals 144a⟩+≡                    (135a) ◁143e 145c▷
  Alt alts[NCW+1];
  Consreitemesg cwm;
  Stringpair pair;
```
Uses NCW-6 143d.

```
⟨xfidwrite() cases 144b⟩+≡                           (135a) ◁140b 146b▷
  case Qcons:

        ⟨xfidwrite() when Qcons, look for previous partial rune bytes 145b⟩
        r = runemalloc(cnt);
        cvttorunes(req->data, cnt-UTFmax, r, &nb, &nr, nil);
        ⟨xfidwrite() when Qcons, look if more full runes 145d⟩
        ⟨xfidwrite() when Qcons, store remaining partial rune bytes 145a⟩

        ⟨xfidxxx() set flushtag 283c⟩

        alts[CWdata].c = w->conswrite;
        alts[CWdata].v = &cwm;
        alts[CWdata].op = CHANRCV;
        ⟨xfidwrite() when Qcons, set alts for flush 284e⟩
        alts[NCW].op = CHANEND;

        switch(alt(alts)){
        case CWdata:
            break;
        ⟨xfidwrite() when Qcons, switch alt flush case 284f⟩
        }

        /* received data */
        ⟨xfidxxx() unset flushtag 283d⟩
        ⟨xfidwrite() when Qcons, if flushing 284g⟩

        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-4 143d, NCW-6 143d, Qcons 141c, cvttorunes() 294f, filsysrespond() 124a, and runemalloc 294c.

9.3.3 Bytes versus runes, 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` keeps a small buffer `rpart` 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 144c⟩+≡                              (63c) ◁123c
  uchar rpart[UTFmax];
  int  nrpart;
```

⟨xfidwrite() *when Qcons, store remaining partial rune bytes* 145a⟩≡ (144b)

```
// 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* 145b⟩≡ (144b)

```
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* 145c⟩+≡ (135a) ◁144a

```
int c;
```

⟨xfidwrite() *when Qcons, look if more full runes* 145d⟩≡ (144b)

```
/* approach end of buffer */
while(fullrune(req->data + nb, cnt-nb)){
    c = nb;
    nb += chartorune(&r[nr], req->data + c);
    if(r[nr])
        nr++;
}
```

9.4 /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.

⟨qid *cases* 145e⟩+≡ (122d) ◁141c 146c▷

```
Qconsctl,
```

⟨*dirtab array elements* 145f⟩+≡ (123b) ◁141d 146d▷

```
{ "consctl", QTFILE, Qconsctl, 0200 },
```

Uses `Qconsctl` 145e.

Like `mouseopenX`, `ctlopen` 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* 145g⟩+≡ (59) ◁143a 192d▷

```
bool ctlopen;
```

⟨xfidopen() *cases* 145h⟩≡ (131d) 148▷

```
case Qconsctl:
    if(w->ctlopen){
        filsysrespond(x->fs, x, &fc, Einuse);
        return;
    }
    w->ctlopen = true;
    break;
```

Uses `Einuse` 290h, `Qconsctl` 145e, and `filsysrespond()` 124a.

```
⟨xfidclose() cases 146a⟩≡                                  (132b)  146e▷
  case Qconsctl:
      ⟨xfidclose() Qconsctl case, if rawing 149d⟩
      ⟨xfidclose() Qconsctl case, if holding 277i⟩
      w->ctlopen = false;
      break;
```

Uses Qconsctl 145e.

```
⟨xfidwrite() cases 146b⟩+≡                          (135a)  ◁144b  146g▷
  case Qconsctl:
      ⟨xfidwrite() Qconsctl case 149c⟩
      // else
      filsysrespond(x->fs, x, &fc, "unknown control message");
      return;
```

Uses Qconsctl 145e and filsysrespond() 124a.

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) stores it in `w->cursor` and points `w->cursorp` 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. This is how programs like `acme` show a crosshair cursor over the tag line but revert to the arrow elsewhere.

```
⟨qid cases 146c⟩+≡                                  (122d)  ◁145e  147a▷
  Qcursor,
```

```
⟨dirtab array elements 146d⟩+≡                      (123b)  ◁145f  147b▷
  { "cursor",  QTFILE, Qcursor, 0600 },
```

Uses Qcursor 146c.

```
⟨xfidclose() cases 146e⟩+≡                          (132b)  ◁146a  149a▷
  case Qcursor:
      w->cursorp = nil;
      wsetcursor(w, false);
      break;
```

Uses Qcursor 146c and wsetcursor() 91d.

```
⟨xfidread() cases 146f⟩+≡                           (134a)  ◁142f  147c▷
  case Qcursor:
      filsysrespond(x->fs, x, &fc, "cursor read not implemented");
      break;
```

Uses Qcursor 146c and filsysrespond() 124a.

```
⟨xfidwrite() cases 146g⟩+≡                          (135a)  ◁146b  225a▷
  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 146c, sweeping 94a, and wsetcursor() 91d.

9.6 /dev/draw/ and /mnt/wsys/winname

Despite the complex collaboration between `rio` and `draw` described in Section 2.5.5—involving layers, `namedimage()`, and the four-phase initialization—the virtual `/dev/winname` device itself is trivially simple: it just returns the string in `w->name`. All the complexity lives in how that name is established and used, not in serving it.

⟨qid *cases* 147a⟩+≡ (122d) ◁146c 157a▷
```
  Qwinname,
```

⟨*dirtab array elements* 147b⟩+≡ (123b) ◁146d 157b▷
```
  { "winname", QTFILE, Qwinname, 0400 },
```
Uses `Qwinname` 147a.

⟨xfidread() *cases* 147c⟩+≡ (134a) ◁146f 157d▷
```
  case Qwinname:
      n = strlen(w->name);
      ⟨xfidread() when Qwinname case, sanity check n 147d⟩
      t = estrdup(w->name);
      goto Text;
```
Uses `Qwinname` 147a and `estrdup()` 294a.

⟨xfidread() *when Qwinname case, sanity check n* 147d⟩≡ (147c)
```
  if(n == 0){
      filsysrespond(x->fs, x, &fc, "window has no name");
      break;
  }
```
Uses `filsysrespond()` 124a.

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 `mouseopen`: when the application opens `/mnt/wsys/mouse`, `rio` stops intercepting middle-click and right-click for its own menus and forwards all mouse events to the application. 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 `/mnt/wsys/mouse`, and optionally `initkeyboard()` to write `"rawon"` on `/mnt/wsys/consctl`. Each of these triggers specific behavior in `rio`'s filesystem server.

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 `/mnt/wsys/winname` to discover the layer name (see Section 2.5.5).

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()`[131d] request with `Qmouse`. This is where `rio` switches the window into graphical mode by setting `mouseopen` to true.

```
⟨xfidopen() cases 148⟩+≡                                   (131d) ◁145h 228a▷
  case Qmouse:
      if(w->mouseopen){
          filsysrespond(x->fs, x, &fc, Einuse);
          return;
      }
      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` 290h, `Qmouse` 138a, and `filsysrespond()` 124a.

⟨`xfidclose()` *cases* 149a⟩+≡ (132b) ◁146e 228b▷

```
  case Qmouse:
    w->mouseopen = false;

    w->resized = false;
    if(w->i != nil)
        wsendctlmesg(w, Refresh, w->i->r, nil);
    break;
```

Uses `Qmouse` 138a, `Refresh` 286c, and `wsendctlmesg()` 96c.

10.1.3 `initkeyboard()`, raw-access mode

The `rawing` 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* 149b⟩+≡ (59) ◁108c 276d▷

```
  bool rawing;
```

⟨`xfidwrite()` *Qconsctl case* 149c⟩≡ (146b) 277h▷

```
  if(strncmp(req->data, "rawon", 5)==0){
    ⟨xfidwrite() Qconsctl case, if rawon message and holding mode 278a⟩
    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` 149e, `Rawon` 149e, and `wsendctlmesg()` 96c.

⟨`xfidclose()` *Qconsctl case, if rawing* 149d⟩≡ (146a)

```
  if(w->rawing){
    w->rawing = false;
    wsendctlmesg(w, Rawoff, ZR, nil);
  }
```

Uses `Rawoff` 149e and `wsendctlmesg()` 96c.

⟨`Wctlmesgkind` *cases* 149e⟩+≡ (96a) ◁141a 277g▷

```
  Rawon,
  Rawoff,
```

⟨wctlmesg() *cases* 150a⟩+≡ (96d) ◁141b 278c▷

```
  case Rawon:
      // already setup w->rawing in xfidwrite, nothing else todo
      break;
  case Rawoff:
      ⟨wctlmesg() break if window was deleted 108e⟩
      ⟨wctlmesg() When Rawoff, process raw keys in non rawing mode 154a⟩
      break;
```

Uses Rawoff 149e and Rawon 149e.

10.2 Mouse events

This section shows the `winctl`[78] side of mouse event delivery—the producer/consumer machinery that sits between `mousethread`[72c] and the client's `read("/dev/mouse")`. The previous chapter showed the `xfidread()`[134a] side (part 1); here we see part 2: how `winctl` queues click events, decides when to offer data, and responds on the reply channel.

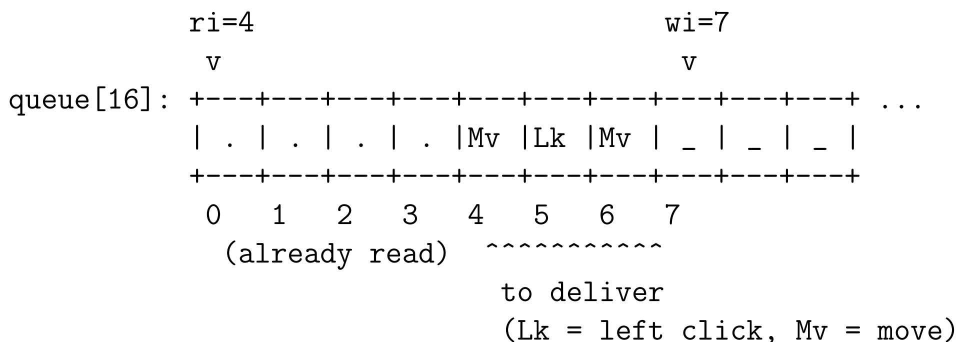
10.2.1 Mouse state queue

Each window keeps its own `Mouseinfo`[151a] struct to buffer mouse events destined for the client application. This is the producer's staging area: `winctl`[78] writes into it when it receives events from `mousethread`[72c], and reads from it when the client is ready via the channel-of-channels protocol described in the previous chapter.

⟨Window *mouse fields* 150b⟩+≡ (59) ◁91c

```
  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 `counter`/`lastcounter` pair tracks whether there is a new event: if they differ, the client gets the latest state on the next `read`. 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:




```
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, `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 `nraw` is strictly append-and-consume (see the raw-keys queue later).

⟨*struct* `Mouseinfo` 151a⟩≡ (299b)
```
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 151b⟩
};
```

⟨`Mouseinfo` *other fields* 151b⟩≡ (151a) 151c▷
```
ulong counter; /* serial no. of last mouse event we received */
ulong lastcounter;/* serial no. of last mouse event sent to client */
```

⟨`Mouseinfo` *other fields* 151c⟩+≡ (151a) ◁151b
```
int lastb; /* last button state we received */
```

⟨*struct* `Mousestate` 151d⟩≡ (299b)
```
struct Mousestate
{
    Mouse;
    ulong counter; /* serial no. of mouse event */
};
```

10.2.2 /mnt/wsys/mouse reading part2

This is part 2 of the mouse reading code. Part 1 in the previous chapter showed the `xfidread()`[134a] side—how the worker sends a `Mousereadmesg`[139c] and waits for a reply. Here we see the `winctl`[78] side: the producer that enqueues click events into `w->mouse.queue`, and the consumer that dequeues them (or snapshots the current position) when the client is ready.

Producer

⟨`winctl()` *other locals* 151e⟩+≡ (78) ◁80f 152b▷
```
Mousestate *mp;
int lastb = -1;
```

⟨`winctl()` *WMouse case if mouseopen* 151f⟩≡ (80d)
```
if(w->mouseopen) {
    w->mouse.counter++;

    /* queue click events */
    if(!w->mouse.qfull && lastb != w->mc.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;
        mp->counter = w->mouse.counter;

        lastb = w->mc.buttons;
    }
}
```

Consumer

⟨Wxxx *cases* 152a⟩≡ (77b) 154b ▷
```
WMouseread,
```

⟨winctl() *other locals* 152b⟩+≡ (78) ◁ 151e 152g ▷
```
Mousereadmesg mrm;
```

⟨winctl() *channels creation* 152c⟩≡ (78) 154d ▷
```
mrm.cm = chancreate(sizeof(Mouse), 0);
```

⟨winctl() *Wctl case, free channels if wctlmesg is Excited* 152d⟩≡ (81a) 154e ▷
```
chanfree(mrm.cm);
```

⟨winctl() *alts setup* 152e⟩+≡ (78) ◁ 80g 154f ▷
```
alts[WMouseread].c = w->mouseread;
alts[WMouseread].v = &mrm;
alts[WMouseread].op = CHANSND;
```
Uses WMouseread-87 152a.

⟨winctl() *alts adjustments* 152f⟩≡ (78) 154g ▷
```
if(w->mouseopen && w->mouse.counter != w->mouse.lastcounter)
    alts[WMouseread].op = CHANSND;
else
    alts[WMouseread].op = CHANNOP;
```
Uses WMouseread-87 152a.

⟨winctl() *other locals* 152g⟩+≡ (78) ◁ 152b 154c ▷
```
Mousestate m;
```

⟨winctl() *event loop cases* 152h⟩+≡ (78) ◁ 81a 155a ▷
```
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.Mouse, w->mouse.counter};

    w->mouse.lastcounter = m.counter;
    // consumed and relay
    send(mrm.cm, &m.Mouse);
    continue;
```
Uses WMouseread-87 152a.

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 `w->raw` and consumed one rune at a time. The complexity here is different—it lies in the interaction between raw mode and 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 `w->rawing` is true, i.e., when a client has written `"rawon"` on `/mnt/wsys/consctl`. In that mode, `wkeyctl`[79e] appends each keystroke to `raw` 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`[78] converts them to UTF-8 bytes and sends them out.

⟨`Window` *graphical window fields* 153a⟩+≡ (59) ◁61f

```
// growing_array<Rune> (size = Window.nraw)
Rune  *raw;
uint  nraw;
```

10.3.2 `/mnt/wsys/cons` reading part2

Producer

When raw mode is active, keystrokes bypass the normal insertion path and go directly to the `raw` queue via `waddraw`. The condition `w->mouseopen || w->q0 == w->nr` 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* 153b⟩≡ (79e)

```
if(w->rawing && (w->mouseopen || w->q0 == w->nr)){
    waddraw(w, &r, 1);
    return;
}
```

Uses `waddraw()` 153c.

⟨*function* `waddraw` 153c⟩≡ (315b)

```
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` 294e and `runerealloc` 294d.

When the application sends `Rawoff`, any runes that were accumulated in `w->raw` during raw mode are replayed through `wkeyctl`[79e] in normal (line-buffered) mode. Since `rawing` 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* 154a⟩≡ (150a)
```
while(w->nraw > 0){
     wkeyctl(w, w->raw[0]);
     --w->nraw;
     runemove(w->raw, w->raw+1, w->nraw);
}
```
Uses `runemove` 294e and `wkeyctl()` 79e.

Consumer

⟨`Wxxx` *cases* 154b⟩+≡ (77b) ◁152a 212a▷
```
WCread,
```

⟨`winctl()` *other locals* 154c⟩+≡ (78) ◁152g 154h▷
```
Consreadmesg crm;
```

⟨`winctl()` *channels creation* 154d⟩+≡ (78) ◁152c 212c▷
```
crm.c1 = chancreate(sizeof(Stringpair), 0);
crm.c2 = chancreate(sizeof(Stringpair), 0);
```

⟨`winctl()` *Wctl case, free channels if wctlmesg is Excited* 154e⟩+≡ (81a) ◁152d 212d▷
```
chanfree(crm.c1);
chanfree(crm.c2);
```

⟨`winctl()` *alts setup* 154f⟩+≡ (78) ◁152e 212e▷
```
alts[WCread].c = w->consread;
alts[WCread].v = &crm;
alts[WCread].op = CHANSND;
```
Uses `WCread-88` 154b.

⟨`winctl()` *alts adjustments* 154g⟩+≡ (78) ◁152f 212f▷
```
⟨winctl() alts adjustments, if holding 276e⟩
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 204d⟩
}
```
Uses `WCread-88` 154b.

⟨`winctl()` *other locals* 154h⟩+≡ (78) ◁154c 155b▷
```
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()`[135a] side for the opposite conversion.

⟨`winctl()` *event loop cases* 155a⟩+≡ (78) ◁152h 213a▷

```
case WCread:
    recv(crm.c1, &pair);
    t = pair.s;
    nb = pair.ns;

    i = 0;
    ⟨winctl() when WCRead, copy in t previous partial rune bytes 155d⟩

    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 204e⟩
        }
    }
    ⟨winctl() when WCRead, handle EOF character after while loop 204f⟩
    ⟨winctl() when WCRead, store overflow bytes of partial rune 155c⟩

    pair.s = t;
    pair.ns = i;
    send(crm.c2, &pair);
    continue;
```

Uses `WCread-88` 154b and `runemove` 294e.

Runes vs bytes, partial runes

⟨`winctl()` *other locals* 155b⟩+≡ (78) ◁154h 212b▷

```
char part[3]; // UTFMAX-1
int npart = 0;
```

⟨`winctl()` *when WCRead, store overflow bytes of partial rune* 155c⟩≡ (155a)

```
if(i > nb){
    npart = i-nb;
    memmove(part, t+nb, npart);
    i = nb;
}
```

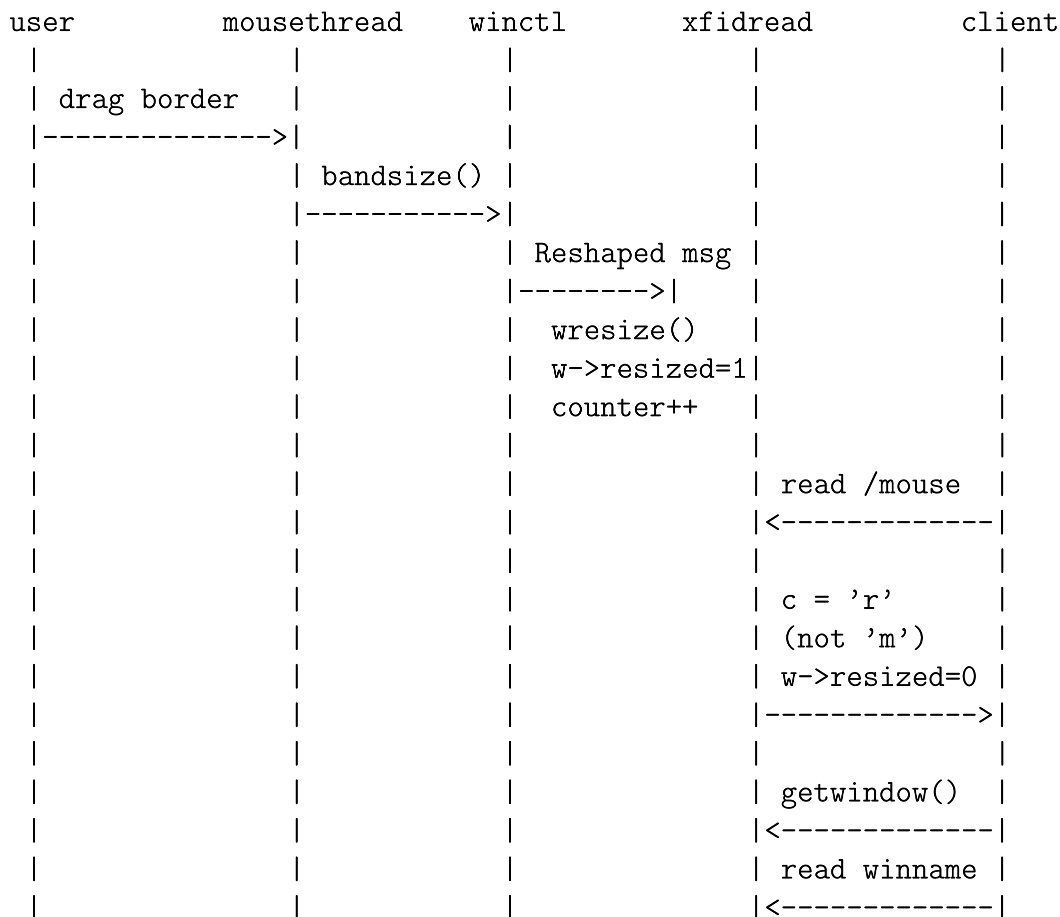
⟨`winctl()` *when WCRead, copy in t previous partial rune bytes* 155d⟩≡ (155a)

```
i = npart;
npart = 0;
if(i)
    memmove(t, part, i);
```

10.4 Resize events

When a window is resized, the client must be notified so it can redraw. The notification is piggybacked on the mouse event stream: `wresize` 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` 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 `resized` 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* 156⟩≡ (139f)

```
if(w->resized)
    c = 'r';
```

10.5 /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 draw 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.

⟨qid *cases* 157a⟩+≡ (122d) ◁147a 222a▷

```
  Qwindow,
```

⟨*dirtab array elements* 157b⟩+≡ (123b) ◁147b 222b▷

```
  { "window",  QTFILE, Qwindow, 0400 },
```

Uses `Qwindow` 157a.

⟨xfidread() *other locals* 157c⟩+≡ (134a) ◁142e 229b▷

```
  Image *i;
  Rectangle r;
  char buf[128];
  char cbuf[30];
```

⟨xfidread() *cases* 157d⟩+≡ (134a) ◁147c 222c▷

```
  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` 291j, `Qwindow` 157a, `estrdup()` 294a, `filsysrespond()` 124a, and `readwindow()` 157e.

⟨*function* `readwindow` 157e⟩≡ (318)

```
  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

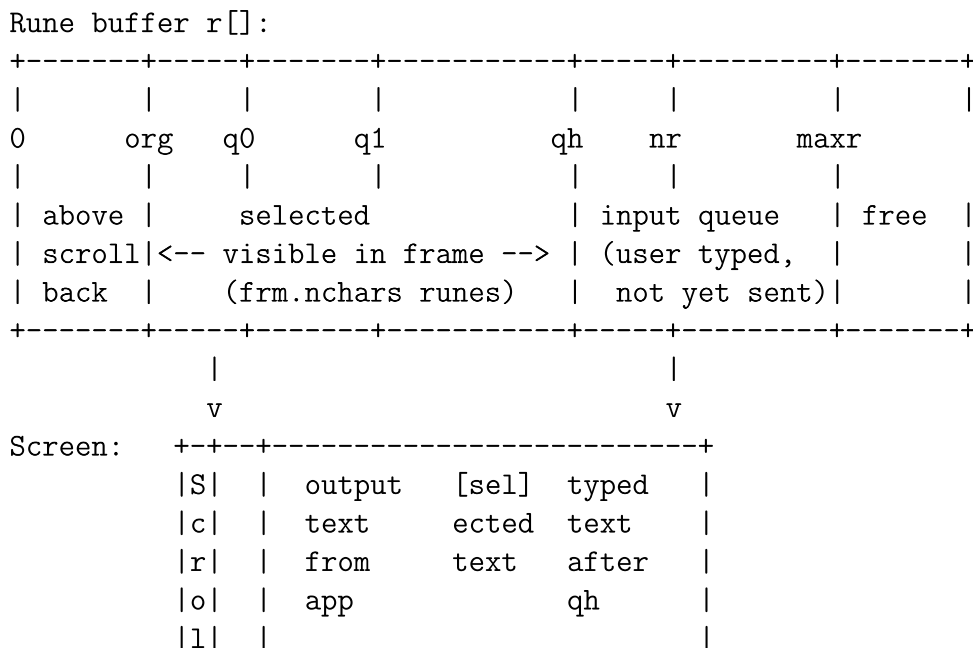
Textual Windows

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, cut/copy/paste, and scrolling. This is possible because `/dev/cons` is virtualized: writes from the application insert text at the output point (`qh`), while the user can independently navigate and select text with the mouse.

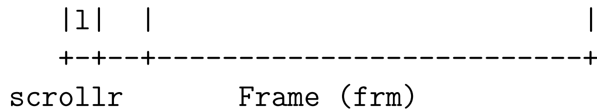
A textual window is essentially a simple editor, with a data model (the rune buffer `r`/`nr`), cursors (`q0`, `q1`, `qh`), a viewport (`org`), and a rendering engine (the `Frame` library that handles text layout, wrapping, and selection highlighting). This chapter is the largest in the book because the textual window touches every layer: the data model (rune buffer, cursors), the rendering engine (the Frame library with its box array, incremental insert/delete, and selection painting), the keyboard handler (line discipline, navigation, erase keys), the mouse handler (selection, menu, scrollbar), and the application output handler (backspace processing, flow control). The chapter is organized bottom-up: first the Frame widget internals, then content modification, then rendering, and finally the event handlers (keyboard, output, mouse) that drive everything.

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 `r` holds all text ever written by the application plus what the user has typed. `org` is the index of the first visible rune; `frm.nchars` is how many runes fit in the frame. The selection (`q0`/`q1`) and the output point (`qh`) are indices into this same buffer:



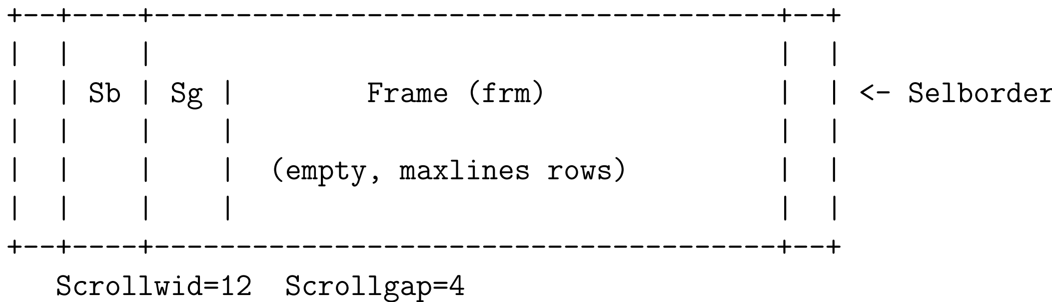
```
Rune buffer r[]:
+-------+-----+-------+----------+-----+--------+-------+
|       |     |       |          |     |        |       |
0      org   q0      q1         qh    nr       maxr
|       |     |       |          |     |        |
| above |     selected           | input queue  | free  |
| scroll|<-- visible in frame --> | (user typed, |       |
| back  |    (frm.nchars runes)  |  not yet sent)|       |
+-------+-----+-------+----------+-----+--------+-------+
              |                        |
              v                        v
Screen:   +-+--+-------------------------+
          |S|  |  output   [sel]  typed   |
          |c|  |  text     ected  text    |
          |r|  |  from     text   after   |
          |o|  |  app             qh      |
          |l|  |                          |
```



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()`[98e] 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` 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* 160a⟩≡ (98e) 161a ▷

```
  ⟨wmk() textual window settings, set scrollbar 160b⟩
  ⟨wmk() textual window settings, set frame 161b⟩

  r = insetrect(w->i->r, Selborder);
  draw(w->i, r, cols[BACK], nil, w->frm.entire.min);
```

Uses `Selborder` 73d and `cols-67` 164a.

11.2.1 Scrollbar

The scrollbar occupies a `Scrollwid` (12 pixel) wide strip on the left side of the window, just inside the selection border. Its rectangle is stored in `w->scrollr` and used in two places: by `wscrdraw()`[191b] for drawing the scrollbar thumb, and by `wmousectl()`[216b] to detect clicks in the scrollbar area (which are dispatched to `wscroll()`[218c] rather than to the text selection code). The `scrolling` flag controls auto-scroll behavior and is initialized from the global `scrolling` default.

⟨`wmk()` *textual window settings, set scrollbar* 160b⟩≡ (160a)

```
  r = insetrect(w->i->r, Selborder+1);
  w->scrollr = r;
  w->scrollr.max.x = r.min.x+Scrollwid;
```

Uses `Scrollwid` 160c and `Selborder` 73d.

⟨*constants Scrollxxx* 160c⟩≡ (299a) 161c ▷

```
  Scrollwid   = 12, /* width of scroll bar */
```

⟨wmk() *textual window settings* 161a⟩+≡ (98e) ◁160a
```
w->scrolling = scrolling; // autoscroll
```

11.2.2 Frame

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.

⟨wmk() *textual window settings, set frame* 161b⟩≡ (160a)
```
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 192e⟩
```
Uses `Scrollgap` 161c, `Scrollwid` 160c, `Selborder` 73d, and `cols-67` 164a.

⟨*constants Scrollxxx* 161c⟩+≡ (299a) ◁160c
```
Scrollgap   = 4, /* gap right of scroll bar */
```

11.3 Frame widget

The Frame library (used by both `rio` and `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`), 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, `_frcanfitX` determines how many characters fit and `_frsplitboxX` divides it). Tabs and newlines get their own special boxes with `nrune == -1`.

11.3.1 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` 162a⟩≡ (297b)

```
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 163e⟩
    ⟨Frame text fields 162b⟩
    ⟨Frame tick fields 165a⟩
    ⟨Frame box fields 166d⟩
    ⟨Frame scroll 217c⟩

    ⟨Frame other fields 163c⟩
};
```

⟨`Frame` *text fields* 162b⟩≡ (162a) 162c ▷

```
ushort nchars;  /* # runes in frame */
```

⟨`Frame` *text fields* 162c⟩+≡ (162a) ◁ 162b 163b ▷

```
ushort nlines;  /* # lines with text */
```

11.3.2 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 `frsetrectsX`), 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` 162d⟩≡ (343a)

```
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 164b⟩
}
```

⟨*function* frsetrects 163a⟩≡ (343a)

```
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* 163b⟩+≡ (162a) ◁162c 174b▷

```
ushort maxlines; /* total # lines in frame */
```

⟨Frame *other fields* 163c⟩≡ (162a)

```
Rectangle entire;  /* of full frame */
```

⟨*function* frclear 163d⟩≡ (343a)

```
void
frclear(Frame *f, bool freeall)
{
    ⟨frclear() free boxes 168b⟩
    ⟨frclear() free ticks 166c⟩
}
```

11.3.3 Frame colors

The frame uses five colors indexed by `BACK`, `HIGH`, `BORD`, `TEXT`, and `HTEXT`:

- `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`[164e] 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()`[202c]. This is a lightweight way to indicate which window has keyboard focus.

⟨Frame *colors* 163e⟩≡ (162a)

```
Image *cols[NCOL]; /* text and background colors */
```

⟨*enum* _anon_ (include/frame.h) 163f⟩≡ (297b)

```
enum FrameColors {
    BACK, // Background
    HIGH, // Background highlighted text
    BORD, // Border
    TEXT, // Text
    HTEXT, // Highlited text

    NCOL
};
```

⟨*global* cols 164a⟩≡ (311)
```
// map<Property, Color>
static Image *cols[NCOL];
```

⟨frinit() *initialize other fields* 164b⟩≡ (162d) 165c ▷
```
if(cols != nil)
    memmove(f->cols, cols, sizeof f->cols);
```

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* 164c⟩≡ (98e)
```
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 99f⟩
);
}
```
Uses `cols-67` 164a.

⟨wrepaint() *update cols* 164d⟩≡ (107a)
```
wsetcols(w);
```
Uses `wsetcols()` 164e.

The `wsetcols()`[164e] 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 164e⟩≡ (311)
```
void
wsetcols(Window *w)
{
    ⟨wsetcols() if holding 277b⟩
    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 `darkgrey-68` 164f and `input` 61e.

⟨*global* darkgrey 164f⟩≡ (311)
```
static Image *darkgrey;
```

⟨wmk() *extra colors initialisation* 164g⟩+≡ (164c) ◁ 99f 277f ▷
```
/* greys are multiples of 0x11111100+0xFF, 14* being palest */
darkgrey = allocimage(display, Rect(0,0,1,1), CMAP8, true, 0x666666FF);
```
Uses `darkgrey-68` 164f.

11.3.4 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* 165a⟩≡ (162a) 165b ▷
```
  Image *tick; /* typing tick */
  Image *tickback; /* saved image under tick */
```

⟨Frame *tick fields* 165b⟩+≡ (162a) ◁ 165a
```
  bool ticked; /* flag: is tick onscreen? */
```

⟨frinit() *initialize other fields* 165c⟩+≡ (162d) ◁ 164b 166e ▷
```
  if(f->tick == nil && f->cols[BACK] != nil)
      frinittick(f);
```

⟨*constant* FRTICKW 165d⟩≡ (297b)
```
  #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 165e⟩≡ (343a)
```
  void
  frinittick(Frame *f)
  {
      Image *b = f->display->screenimage;
      Font *ft = f->font;

      ⟨frinittick() free old tick 165f⟩
      f->tick = allocimage(f->display, Rect(0, 0, FRTICKW, ft->height), b->chan, 0, DWhite);
      ⟨frinittick() sanity check tick 165g⟩
      ⟨frinittick() free old tickback 166a⟩
      f->tickback = allocimage(f->display, f->tick->r, b->chan, false, DWhite);
      ⟨frinittick() sanity check tickback 166b⟩
      /* 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);
  }
```

⟨frinittick() *free old tick* 165f⟩≡ (165e)
```
  if(f->tick)
      freeimage(f->tick);
```

⟨frinittick() *sanity check tick* 165g⟩≡ (165e)
```
  if(f->tick == nil)
      return;
```

⟨frinittick() *free old tickback* 166a⟩≡ (165e)

```
if(f->tickback)
    freeimage(f->tickback);
```

⟨frinittick() *sanity check tickback* 166b⟩≡ (165e)

```
if(f->tickback == nil){
    freeimage(f->tick);
    f->tick = nil;
    return;
}
```

⟨frclear() *free ticks* 166c⟩≡ (163d)

```
if(freeall){
    freeimage(f->tick);
    freeimage(f->tickback);
    f->tick = nil;
    f->tickback = nil;
}
f->ticked = false;
```

11.3.5 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* 166d⟩≡ (162a)

```
// growing_array<Frbox> (size = nalloc, unused after nbox)
Frbox *box;
ushort nbox;
ushort nalloc;
```

⟨frinit() *initialize other fields* 166e⟩+≡ (162d) ◁165c 174c▷

```
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` 167a⟩≡ (297b)
```
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` 167b⟩≡ (297b)
```
#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` 167c⟩≡ (341b)
```
#define SLOP 25
```

⟨*function* `_fraddbox` 167d⟩≡ (341b)
```
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 167f⟩
    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;
}
```

⟨*function* `_frgrowbox` 167e⟩≡ (341b)
```
void
_frgrowbox(Frame *f, int delta)
{
    f->nalloc += delta;
    f->box = realloc(f->box, f->nalloc*sizeof(Frbox));
    ⟨_frgrowbox() sanity check box 168a⟩
}
```

⟨`_fraddbox()` *sanity check bn* 167f⟩≡ (167d)
```
if(bn > f->nbox)
    drawerror(f->display, "_fraddbox");
```

⟨`_frgrowbox()` *sanity check box* 168a⟩≡ (167e)
```
if(f->box == nil)
    drawerror(f->display, "_frgrowbox");
```

Free boxes

⟨`frclear()` *free boxes* 168b⟩≡ (163d)
```
if(f->nbox)
    _frdelbox(f, 0, f->nbox-1);
if(f->box)
    free(f->box);
f->box = nil;
```

⟨*function* `_frdelbox` 168c⟩≡ (341b)
```
void
_frdelbox(Frame *f, int n0, int n1) /* inclusive */
{
    ⟨_frdelbox() sanity check n0 and n1 168f⟩
    _frfreebox(f, n0, n1);
    _frclosebox(f, n0, n1);
}
```

⟨*function* `_frfreebox` 168d⟩≡ (341b)
```
void
_frfreebox(Frame *f, int n0, int n1) /* inclusive */
{
    int i;

    ⟨_frfreebox() sanity check n0 and n1 168g⟩
    n1++;
    for(i=n0; i<n1; i++)
        if(f->box[i].nrune >= 0)
            free(f->box[i].ptr);
}
```

⟨*function* `_frclosebox` 168e⟩≡ (341b)
```
void
_frclosebox(Frame *f, int n0, int n1) /* inclusive */
{
    int i;

    ⟨_frclosebox() sanity check n0 and n1 168h⟩
    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* 168f⟩≡ (168c)
```
if(n0>=f->nbox || n1>=f->nbox || n1<n0)
    drawerror(f->display, "_frdelbox");
```

⟨`_frfreebox()` *sanity check n0 and n1* 168g⟩≡ (168d)
```
if(n1<n0)
    return;
if(n0>=f->nbox || n1>=f->nbox)
    drawerror(f->display, "_frfreebox");
```

⟨`_frclosebox()` *sanity check n0 and n1* 168h⟩≡ (168e)
```
if(n0>=f->nbox || n1>=f->nbox || n1<n0)
    drawerror(f->display, "_frclosebox");
```

11.3.6 Frame strings

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 (windows/libframe/frstr.c)* 169a⟩≡ (344b)

```
#define CHUNK 16
```

⟨*function* `ROUNDUP` 169b⟩≡ (344b)

```
#define ROUNDUP(n) ((n+CHUNK)&~(CHUNK-1))
```

⟨*function* `_frallocstr` 169c⟩≡ (344b)

```
uchar *
_frallocstr(Frame *f, unsigned n)
{
    uchar *p;

    p = malloc(ROUNDUP(n));
    if(p == nil)
        drawerror(f->display, "out of memory");
    return p;
}
```

⟨*function* `_frinsure` 169d⟩≡ (344b)

```
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;
}
```

11.3.7 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 `frinsertX` and `frdeleteX` 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 (`_frptofcharptbX`) 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` 170a⟩≡ (343c)

```
Point
frptofchar(Frame *f, ulong p)
{
    return _frptofcharptb(f, p, f->r.min, 0);
}
```

⟨*function* `_frptofcharptb` 170b⟩≡ (343c)

```
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 171b⟩
                }
            break; // found it
        }
        // else
```

```
        p -= 1;
        _fradvance(f, &pt, b);
    }
    return pt;
}
```

⟨*function* `_fradvance` 171a⟩≡ (344c)

```
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* 171b⟩≡ (170b)

```
if(r==0 || pt.x > f->r.max.x)
    drawerror(f->display, "frptofchar");
```

⟨*function* `_frcklinewrap` 171c⟩≡ (344c)

```
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` 171d⟩≡ (344c)

```
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;
    }
}
```

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` 171e⟩≡ (344c)

```
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` 172a⟩≡ (343c)

```
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;
}
```

`frcharofpt()`

The inverse of `frptofcharX`: 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` 172b⟩≡ (343c)

```
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 173b⟩
                    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;
}
```

⟨*function* `_frgrid` 173a⟩≡ (343c)

```
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* 173b⟩≡ (172b)

```
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 `_frcklinewrapX`, 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 `frselectpaintX`. 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 `frinsertX` 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` 174a⟩≡ (342b)
```
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;
    }
}
```

11.3.8 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`[199]). 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 `frdrawselX` 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* 174b⟩+≡ (162a) ◁163b 182b▷
```
ulong p0, p1;  /* selection */
```

⟨`frinit()` *initialize other fields* 174c⟩+≡ (162d) ◁166e 198a▷
```
f->p0 = 0;
f->p1 = 0;
```

11.4 Content modification

When text is inserted (either from keyboard input or application output), `winsert`[175b] 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 (as in `efuns`) because a terminal usually inserts at the end, not in the middle, so the `runemove` cost is acceptable. Here is a walkthrough of what happens when the user types a single character 'x' at the cursor. The call chain is:

1. `wkeyctl`[79e] receives the rune from `winctl`[78].

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 (`bxscanX`), 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`[197a].

⟨*simplified code when entered a single rune* `r` *in a terminal* 175a⟩≡
```
winsert(w, &r, 1, w->q0);
wshow(w, w->q0);
```

11.4.1 winsert()

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` 175b⟩≡ (312b)
```
uint
winsert(Window *w, Rune *r, int n, uint q0)
{
    uint m;

    ⟨winsert() sanity check n 176a⟩
    ⟨winsert() if size of rune array is getting really big 176d⟩
    ⟨winsert() grow rune array if reach maxr 176c⟩

    // 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 175c⟩

    return q0;
}
```
Uses `runemove` 294e.

⟨`winsert()` *adjust cursors* 175c⟩≡ (175b)
```
/* 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 177a⟩
```

⟨winsert() *sanity check n* 176a⟩≡ (175b)
```
  if(n == 0)
      return q0;
```

11.4.2 Growing array

The rune buffer uses a two-threshold strategy to limit memory. When `nr` 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_ (windows/rio/wind.c)` 176b⟩≡ (312b)
```
  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* 176c⟩≡ (175b)
```
  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-16` 176b, `MinWater-18` 176b, `max()` 293b, `min()` 293a, and `runerealloc` 294d.

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* 176d⟩≡ (175b)
```
  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-16` 176b, `LoWater-17` 176b, `min()` 293a, and `runemove` 294e.

11.5 Content rendering

The rendering pipeline is the most complex part of the Frame library. Its job is to convert rune insertions and deletions into *incremental* screen updates—only redrawing the portion of the screen 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 key function is **frinsertX**, which is the most complex function in the frame library: it must scan the new text into boxes, figure out where existing text will shift, and only redraw the affected portion of the screen rather than repainting everything. The complementary **frdeleteX** removes characters and collapses the display. The rendering pipeline is driven by six main functions, each building on the lower-level ones:

```
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
```

11.5.1 `frinsert()`

⟨`winsert()` *when* `q0 >= w->org`, *possibly update visible text* 177a⟩≡ (175c)
```
  if(q0 <= w->org + w->frm.nchars)
      frinsert(&w->frm, r, r+n, q0 - w->org); // echo back
```

⟨*global* `frame` 177b⟩≡ (343b)
```
  static Frame  frame;
```

⟨*struct* `points_frinsert` 177c⟩≡ (343b)
```
  struct points_frinsert {
      Point pt0, pt1;
  };
```

⟨`frinsert()` *locals* 177d⟩≡ (179)
```
  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;
```

⟨*constant* DELTA 178⟩≡ (343b)
```
#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` 179⟩≡ (343b)

```
void
frinsert(Frame *f, Rune *sp, Rune *ep, ulong p0)
{
    ⟨frinsert() locals 177d⟩

    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 193d⟩

    /*
     * 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 182a⟩
            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 193e⟩
}
```

⟨frinsert() *sanity check n canfit* 182a⟩≡ (179)

```
if(n == 0)
    drawerror(f->display, "_frcanfit==0");
```

⟨Frame *text fields* 182b⟩+≡ (162a) ◁174b 197b▷

```
bool modified; /* changed since frselect() */
```

_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 `frinsertX` and `frdeleteX` 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 182c⟩≡ (341b)

```
/* 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;
}
```

⟨*function* _frsplitbox 182d⟩≡ (341b)

```
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);
}
```

⟨*function* dupbox 183a⟩≡ (341b)
```
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;
    }
}
```

⟨*function* NBYTE 183b⟩≡ (297b)
```
#define NBYTE(b) strlen((char*)(b)->ptr)
```

⟨*function* truncatebox 183c⟩≡ (341b)
```
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);
}
```

⟨*function* chopbox 183d⟩≡ (341b)
```
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);
}
```

⟨*function* runeindex 183e⟩≡ (341b)
```
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;
}
```

`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` 184a⟩≡ (343b)

```
#define TMPSIZE 256
```

The `bxscan` function is the front end of `frinsertX`'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 `frinsertX`—the coordinate where the new text ends.

⟨*function* `bxscan` 184b⟩≡ (343b)

```
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);
}
```

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` 185⟩≡ (342b)

```
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;
}
```

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` 186a⟩≡ (344c)

```
int
_frnewwid(Frame *f, Point pt, Frbox *b)
{
    b->wid = _frnewwid0(f, pt, b);
    return b->wid;
}
```

⟨*function* `_frnewwid0` 186b⟩≡ (344c)

```
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` 186c⟩≡ (342b)

```
int
_frstrlen(Frame *f, int nb)
{
    int n;

    for(n=0; nb<f->nbox; nb++)
        n += NRUNE(&f->box[nb]);
    return n;
}
```

_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` 187a⟩≡ (343b)

```
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);
}
```

_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`[198b] to decide whether more text needs to be loaded from the rune buffer.

⟨*function* `_frclean` 187b⟩≡ (344c)

```
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;
}
```

⟨*function* _frmergebox 188a⟩≡ (341b)

```
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);
}
```

11.5.2 frdelete()

The `frdelete` function is the inverse of `frinsertX`: it removes characters `p0` through `p1` from the display and collapses the remaining text upward. Like `frinsertX`, 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`[198b] 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 `frinsertX`, 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`[197a] or `wdelete`[209c]) uses to decide whether to call `wfill` to fill in text from below the viewport.

⟨*function* frdelete 188b⟩≡ (342a)

```
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;
}
```

11.5.3 wshow()

The `wshow` 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`[197a] to reflow the display from the new origin. This function is called after every text modification (`winsert`, 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` 190⟩≡ (312b)

```
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 196a⟩
}
```

Uses `wscrdraw()` 191b.

11.5.4 Drawing the scrollbar: `wscrdraw()`

⟨`wmk()` *drawing scrollbar* 191a⟩≡ (98e)

```
wscrdraw(w);
```

Uses `wscrdraw()` 191b.

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`[192a], 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. An optimization avoids redundant redraws: `w->lastsr` caches the previous thumb rectangle, and the scrollbar is only repainted if the new rectangle differs. This matters because `wscrdraw`[191b] is called on every text change, and most of the time the scrollbar position does not change visibly.

⟨*function* `wscrdraw` 191b⟩≡ (319)

```
void
wscrdraw(Window *w)
{
    Rectangle r, r1, r2;
    Image *b;

    scrtemps();
    ⟨wscrdraw() sanity check the window image 192g⟩
    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 `scrpos()` 192a, `scrtemps()` 193b, and `scrtmp-74` 192h.

The scrpos 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` 192a⟩≡ (319)
```
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 192c⟩
    if(p0 > 0)
        q.min.y += h*p0 / tot;
    if(p1 < tot)
        q.max.y -= h*(tot-p1) / tot;

    ⟨scrpos() last adjustments 192b⟩
    return q;
}
```

⟨`scrpos()` *last adjustments* 192b⟩≡ (192a)
```
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* 192c⟩≡ (192a)
```
if(tot > 1024*1024){
    tot>>=10;
    p0>>=10;
    p1>>=10;
}
```

⟨`Window` *other fields* 192d⟩+≡ (59) ◁145g 227c▷
```
Rectangle lastsr;
```

⟨`wmk()` *textual window settings, extra frame settings* 192e⟩≡ (161b) 211c▷
```
w->lastsr = ZR;
```

⟨`wresize()` *textual window updates, reset lastsr* 192f⟩≡ (221)
```
w->lastsr = ZR;
```

⟨`wscrdraw()` *sanity check the window image* 192g⟩≡ (191b)
```
if(w->i == nil)
    error("scrdraw");
```
Uses `error()` 292c.

⟨*global* `scrtmp` 192h⟩≡ (319)
```
static Image *scrtmp;
```

⟨*constant* BIG 193a⟩≡ (299a)
```
BIG   = 3, /* factor by which window dimension can exceed screen */
```

⟨*function* scrtemps 193b⟩≡ (319)
```
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 193c⟩
}
```
Uses BIG 193a and scrtmp-74 192h.

⟨scrtemps() *sanity check scrtmp* 193c⟩≡ (193b)
```
if(scrtmp == nil)
    error("scrtemps");
```
Uses error() 292c and scrtmp-74 192h.

11.5.5 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* 193d⟩≡ (179)
```
if(f->p0 == f->p1)
    frtick(f, frptofchar(f, f->p0), false);
```

⟨frinsert() *draw tick* 193e⟩≡ (179)
```
if(f->p0 == f->p1)
    frtick(f, frptofchar(f, f->p0), true);
```

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 193f⟩≡ (342b)
```
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;
}
```

11.5.6 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` 194a⟩≡ (342b)

```
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);
}
```

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` 194b⟩≡ (342b)

```
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;
}
```

⟨*function* nbytes 195⟩≡ (342b)

```
static int
nbytes(char *s0, int nr)
{
    char *s;
    Rune r;
```

```
    s = s0;
    while(--nr >= 0)
        s += chartorune(&r, s);
    return s-s0;
}
```

11.5.7 Moving the frame origin

When the viewport must change (because the target position is off-screen), \hlinkwsetorigin updates `org` and incrementally adjusts the frame. If the new origin is close to the old one, it uses `frdelete` or `frinsert` 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`[190] 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* 196a⟩≡ (190)

```
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()` 196b and `wsetorigin()` 197a.

The `wbacknl` 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` 196b⟩≡ (312b)

```
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` 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` 197a⟩≡ (312b)

```
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 `runemalloc` 294c, `runemove` 294e, `wfill()` 198b, `wscrdraw()` 191b, and `wsetselect()` 199.

⟨`Frame` *text fields* 197b⟩+≡ (162a) ◁182b 211a▷

```
ushort lastlinefull; /* last line fills frame */
```

⟨frinit() *initialize other fields* 198a⟩+≡ (162d) ◁174c 211b▷
```
  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 `frinsertX` 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 `_frcleanX` inside `frinsertX`, 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 198b⟩≡ (312b)
```
  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 `messagesize` 81b and `runemove` 294e.

11.5.8 Selecting

`\hlinkwsetselect` updates the selection (`q0`, `q1`) and repaints only the changed portions. It converts the buffer-wide positions (`q0`, `q1`) to frame-local positions (`p0`, `p1`) by subtracting `org`, 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). `\hlinkfrselect` 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`[199] identifies which characters changed state (selected ↔ unselected) and repaints only those. This makes interactive selection smooth even for large selections.

⟨*function* `wsetselect` 199⟩≡ (312b)

```
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;
}
```

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`[220a]) 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` 200⟩≡ (344a)

```
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->xy;
    b = mc->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->xy;
  }while(mc->buttons == b);
}
```

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` 201⟩≡ (344a)

```
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` 202a⟩≡ (344a)

```
static
int
region(int a, int b)
{
    if(a < b)
        return -1;
    if(a == b)
        return 0;
    return 1;
}
```

11.5.9 Repainting

A full repaint is needed when the frame colors change (window gains or loses focus). **`frredraw`** 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* 202b⟩≡ (107a)

```
if(!w->mouseopen)
    frredraw(&w->frm);
```

A full repaint draws the entire frame in three bands using **`frdrawsel0X`** 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 **`frinsertX`**/**`frdeleteX`**/**`frdrawselX`** paths.

⟨*function* `frredraw` 202c⟩≡ (342b)

```
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]);
}
```

11.6 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 `q0` 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 (`qh`) 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`[71] receives a rune from the keyboard device.
2. `winctl`[78] dispatches it to `wkeyctl`[79e].
3. `wkeyctl` checks for raw mode, navigation keys, special keys, then falls through to ordinary character insertion.

11.6.1 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 is fundamentally different from raw terminals (and from xterm-style terminals that relay every keystroke to the application and rely on the application's line-editing library).

11.6.2 `/mnt/wsys/cons` reading part3

Part 1 (Virtual Devices chapter) showed the `xfidread()`[134a] side that sends a `Consreadmesg`[142b] to `winctl`[78]. Part 2 (Graphical Windows chapter) showed how `winctl` handles raw keystrokes. This part 3 covers the textual-window path: how `wkeyctl`[79e] inserts ordinary characters into the rune buffer (producer), and how `winctl` implements line buffering by scanning for newlines before enabling the `WCread` alternative (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* 204a⟩≡ (79e)
```
  // here when no navigation key, no rawing, no 0x1B holding

  ⟨wkeyctl() snarf and cut if not interrupt key 246c⟩
  switch(r){
  ⟨wkeyctl() special key cases and no special mode 208a⟩
  }
  // else

  /* otherwise ordinary character; just insert */
  ⟨wkeyctl() ordinary character 204c⟩
```

⟨wkeyctl() *locals* 204b⟩≡ (79e) 205b ▷
```
  uint q0;
```

⟨wkeyctl() *ordinary character* 204c⟩≡ (204a)
```
  q0 = w->q0;
  q0 = winsert(w, &r, 1, q0);
  wshow(w, q0+1);
```
Uses `winsert()` 175b and `wshow()` 190.

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* 204d⟩≡ (154g)
```
  /* 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-88` 154b.

⟨winctl() *when WCRead, break if newline and handle EOF character* 204e⟩≡ (155a)
```
  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* 204f⟩≡ (155a)
```
  if(i==nb && w->qh < w->nr && w->r[w->qh]=='\004')
      w->qh++;
```

11.6.3 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 `org` 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`[190] 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* 205a⟩≡ (79e)
```
  if(!w->mouseopen)
  switch(r){
  ⟨wkeyctl() when mouse not opened, switch key cases 205c⟩
  default:
      ; // no return! fallthrough
  }
```

⟨`wkeyctl()` *locals* 205b⟩+≡ (79e) ◁204b 210b▷
```
  uint q1;
  int n, nb;
```

Text boundaries (Home, end)

⟨`wkeyctl()` *when mouse not opened, switch key cases* 205c⟩≡ (205a) 205d▷
```
  case Khome:
      wshow(w, 0);
      return;
```
Uses `wshow()` 190.

⟨`wkeyctl()` *when mouse not opened, switch key cases* 205d⟩+≡ (205a) ◁205c 205e▷
```
  case Kend:
      wshow(w, w->nr);
      return;
```
Uses `wshow()` 190.

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: `frcharofptX` 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`[197a]. The upward scrolling keys (Up, Page Up, scroll wheel up) work the same way but use `wbacknl`[196b] to count lines backward instead.

⟨`wkeyctl()` *when mouse not opened, switch key cases* 205e⟩+≡ (205a) ◁205d 206a▷
```
  case Kdown:
      n = w->frm.maxlines / 3;
      goto case_Down;
```

⟨wkeyctl() *when mouse not opened, switch key cases* 206a⟩+≡ (205a) ◁205e 206b▷
```
  case Kscrollonedown:
      n = mousescrollsize(w->frm.maxlines);
      if(n <= 0)
          n = 1;
      goto case_Down;
```
Uses `Kscrollonedown` 219a.

⟨wkeyctl() *when mouse not opened, switch key cases* 206b⟩+≡ (205a) ◁206a 206c▷
```
  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 `wsetorigin()` 197a.

Up

⟨wkeyctl() *when mouse not opened, switch key cases* 206c⟩+≡ (205a) ◁206b 206d▷
```
  case Kup:
      n = w->frm.maxlines/3;
      goto case_Up;
```

⟨wkeyctl() *when mouse not opened, switch key cases* 206d⟩+≡ (205a) ◁206c 206e▷
```
  case Kscrolloneup:
      n = mousescrollsize(w->frm.maxlines);
      if(n <= 0)
          n = 1;
      goto case_Up;
```
Uses `Kscrolloneup` 219a.

⟨wkeyctl() *when mouse not opened, switch key cases* 206e⟩+≡ (205a) ◁206d 206f▷
```
  case Kpgup:
      n = 2*w->frm.maxlines/3;
      // Fallthrough
  case_Up:
      q0 = wbacknl(w, w->org, n);
      wsetorigin(w, q0, true);
      return;
```
Uses `wbacknl()` 196b and `wsetorigin()` 197a.

Left

⟨wkeyctl() *when mouse not opened, switch key cases* 206f⟩+≡ (205a) ◁206e 207a▷
```
  case Kleft:
      if(w->q0 > 0){
          q0 = w->q0 - 1;
          wsetselect(w, q0, q0);
          wshow(w, q0);
      }
      return;
```
Uses `wsetselect()` 199 and `wshow()` 190.

Right

⟨wkeyctl() *when mouse not opened, switch key cases* 207a⟩+≡ (205a) ◁206f 207b▷
```
  case Kright:
      if(w->q1 < w->nr){
          q1 = w->q1+1;
          wsetselect(w, q1, q1);
          wshow(w, q1);
      }
      return;
```
Uses wsetselect() 199 and wshow() 190.

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 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* 207b⟩+≡ (205a) ◁207a 207c▷
```
  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() 199 and wshow() 190.

⟨wkeyctl() *when mouse not opened, switch key cases* 207c⟩+≡ (205a) ◁207b
```
  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() 207d, wsetselect() 199, and wshow() 190.

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 207d⟩≡ (312b)
```
  int
  wbswidth(Window *w, Rune c)
  {
      uint q, stop;
      Rune r;
      ⟨wbswidth() other locals 208c⟩

      ⟨wbswidth() return if erase character 208b⟩
```

```
    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 209a⟩
        --q;
    }
    return w->q0-q;
}
```

11.6.4 Special keys

The three erase keys—`^H` (backspace), `^U` (kill line), and `^W` (kill word)—all flow through the same handler. They call `wbswidth`[207d] to compute how many characters to erase, then `wdelete`[209c] 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` returns 1, `q0` is set to `w->q0 - 1`, then `wdelete(w, q0, q0+1)` removes that one rune, which in turn calls `frdeleteX` 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* 208a⟩≡ (204a) 210c ▷

```
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 209b⟩
    if(nb > 0){
        wdelete(w, q0, q0+nb);
        wsetselect(w, q0, q0);
    }
    return;
```

Uses `wbswidth()` 207d, `wdelete()` 209c, and `wsetselect()` 199.

⟨`wbswidth()` *return if erase character* 208b⟩≡ (207d)

```
/* there is known to be at least one character to erase */
if(c == 0x08) /* ^H: erase character */
    return 1;
```

⟨`wbswidth()` *other locals* 208c⟩≡ (207d)

```
bool skipping = true;
uint eq;
```

⟨wbswidth() *if* c == 0x17 209a⟩≡ (207d)
```
if(c == 0x17){
    eq = isalnum(r);
    if(eq && skipping) /* found one; stop skipping */
        skipping = false;
    else if(!eq && !skipping)
        break;
}
```
Uses isalnum() 294b.

⟨wkeyctl() *when erase keys, adjust q0 and nb if before org* 209b⟩≡ (208a)
```
if(q0 < w->org){
    q0 = w->org;
    nb = q1-q0;
}
```

The wdelete function mirrors winsert: it removes runes from the buffer by shifting the tail left with runemove, 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 and wfill to update the display incrementally.

The wdelete function is the dual of winsert[175b]: 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, frdeleteX handles the display update and wfill[198b] fills any empty space at the bottom.

⟨*function* wdelete 209c⟩≡ (312b)
```
void
wdelete(Window *w, uint q0, uint q1)
{
    uint n, p0, p1;
    Frame *frm = &w->frm;

    n = q1-q0;
    ⟨wdelete() sanity check n 209d⟩
    runemove(w->r+q0, w->r+q1, w->nr-q1);
    w->nr -= n;

    ⟨wdelete() adjust cursors 209e⟩
}
```
Uses runemove 294e.

⟨wdelete() *sanity check n* 209d⟩≡ (209c)
```
if(n == 0)
    return;
```

⟨wdelete() *adjust cursors* 209e⟩≡ (209c)
```
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 210a⟩
```

Uses min() 293a.

⟨wdelete() *when* q1 > w->org, *possibly update visible text* 210a⟩≡ (209e)

```
  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 wfill() 198b.

Interrupt

When the user types the Delete key (rune `0x7F`), `rio` sends an interrupt note to the child process by writing `"interrupt"` to `w->notefd`. 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* 210b⟩+≡ (79e) ◁205b

```
  int nr;
  Rune *rp;
  int *notefd;
```

⟨wkeyctl() *special key cases and no special mode* 210c⟩+≡ (204a) ◁208a 262c▷

```
  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 emalloc() 293d, interruptproc() 210d, and wshow() 190.

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 210d⟩≡ (312b)

```
  /*
   * 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 `_frnewwid0X`: 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* 211a⟩+≡ (162a) ◁197b

```
ushort maxtab;  /* max size of tab, in pixels */
```

⟨`frinit()` *initialize other fields* 211b⟩+≡ (162d) ◁198a

```
f->maxtab = 8 * stringwidth(ft, "0");
```

⟨`wmk()` *textual window settings, extra frame settings* 211c⟩+≡ (161b) ◁192e

```
w->frm.maxtab = maxtab * stringwidth(font, "0");
```

Uses `maxtab` 211e.

⟨`wresize()` *textual window updates, extra frame settings* 211d⟩≡ (221)

```
w->frm.maxtab = maxtab * stringwidth(w->frm.font, "0");
```

Uses `maxtab` 211e.

⟨*global* `maxtab` 211e⟩≡ (304)

```
int  maxtab = 0;
```

Uses `maxtab` 211e.

⟨`main()` *locals* 211f⟩+≡ (66a) ◁100d 272c▷

```
char *s;
```

⟨`main()` *set some globals* 211g⟩+≡ (66a) ◁100e 250c▷

```
s = getenv("tabstop");
if(s != nil)
    maxtab = strtol(s, nil, 0);
if(maxtab == 0)
    maxtab = 4;
free(s);
```

Uses `maxtab` 211e.

11.7 Application output events

When the application writes to `/dev/cons`, the data arrives at the `winctl`[78] thread through the `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`[190] 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.

11.7.1 /mnt/wsys/cons writing part2

Part 1 (Virtual Devices chapter) showed how `xfidwrite()`[135a] converts the client's bytes to runes and sends them through the channel-of-channels protocol. This part 2 is the `winctl`[78] consumer: it receives the runes, inserts them at the output point `qh` via `winsert`[175b], 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* 212a⟩+≡ (77b) ◁154b 230c▷
```
  WCwrite,
```

⟨winctl() *other locals* 212b⟩+≡ (78) ◁155b 212g▷
```
  Conswritemesg cwm;
```

⟨winctl() *channels creation* 212c⟩+≡ (78) ◁154d 230a▷
```
  cwm.cw = chancreate(sizeof(Stringpair), 0);
```

⟨winctl() *Wctl case, free channels if wctlmesg is Excited* 212d⟩+≡ (81a) ◁154e 230b▷
```
  chanfree(cwm.cw);
```

⟨winctl() *alts setup* 212e⟩+≡ (78) ◁154f 230d▷
```
  alts[WCwrite].c = w->conswrite;
  alts[WCwrite].v = &cwm;
  alts[WCwrite].op = CHANSND;
```
Uses `WCwrite-89` 212a.

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* 212f⟩+≡ (78) ◁154g 230e▷
```
  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-89` 212a.

⟨winctl() *other locals* 212g⟩+≡ (78) ◁212b 213c▷
```
  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 213a⟩+≡                    (78) ◁155a 230g▷
  case WCwrite:
      recv(cwm.cw, &pair);
      rp = pair.s;
      nr = pair.ns;

      ⟨winctl() when WCwrite, if runes contains backspace 213d⟩

      w->qh = winsert(w, rp, nr, w->qh) + nr;
      ⟨winctl() when WCwrite, if scrolling or mouseopen 213b⟩
      wsetselect(w, w->q0, w->q1);
      wscrdraw(w);

      free(rp);
      break;
```

Uses WCwrite-89 212a, winsert() 175b, wscrdraw() 191b, and wsetselect() 199.

```
⟨winctl() when WCwrite, if scrolling or mouseopen 213b⟩≡          (213a)
  if(w->scrolling || w->mouseopen)
      wshow(w, w->qh);
```

Uses wshow() 190.

```
⟨winctl() other locals 213c⟩+≡                        (78) ◁212g 229f▷
  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[209c] before the filtered text is inserted. For example, if the application writes “abc\b\bXY”, the buffer receives “aXY” (the two backspaces erase “c” and “b”, then “XY” is inserted).

```
⟨winctl() when WCwrite, if runes contains backspace 213d⟩≡        (213a)
  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` 294c, `runemove` 294e, and `wdelete()` 209c.

11.8 Mouse events

In a textual window (when `mouseopen` 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`[72c], 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.

11.8.1 Middle click menu

⟨`mousethread()` *middle click under certain conditions* 214⟩≡ (76d)

```
  button2menu(winput);
```

Uses `button2menu()` 215a.

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` 215a⟩≡ (312b)
```
void
button2menu(Window *w)
{
    ⟨button2menu() return if window was deleted 215b⟩

    incref(w);
    ⟨button2menu() menu2str adjustments for scrolling 217a⟩
    switch(menuhit(2, mousectl, &menu2, desktop)){
    ⟨button2menu() cases 217b⟩
    }
    wclose(w); // decref

    wsendctlmesg(w, Wakeup, ZR, nil);
    flushimage(display, true);
}
```
Uses `Wakeup` 286a, `desktop` 58b, `menu2` 215c, `mousectl` 57b, `wclose()` 111a, and `wsendctlmesg()` 96c.

⟨`button2menu()` *return if window was deleted* 215b⟩≡ (215a)
```
if(w->deleted)
    return;
```

⟨*global* `menu2` 215c⟩≡ (312b)
```
Menu menu2 = { .item = menu2str };
```
Uses `menu2str` 215d.

⟨*global* `menu2str` 215d⟩≡ (312b)
```
char*  menu2str[] = {
 [Cut]   "cut",
 [Paste] "paste",
 [Snarf] "snarf",
 [Plumb] "plumb",
 [Send]  "send",
 [Scroll] "scroll",
 nil
};
```

⟨*enum* `_anon_ (windows/rio/rio.c)`*2* 215e⟩≡ (312b)
```
enum
{
    Cut,
    Paste,
    Snarf,
    Plumb,
    Send,
    Scroll,
};
```

11.8.2 Other clicks

When the mouse is not inside the scrollbar, `wmousectl` dispatches based on which button was pressed. Left-click (button 1) triggers text selection via `wselect`[269]; middle and right clicks were already intercepted by `mousethread`[72c] 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[79e]—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* 216a⟩≡ (80d)
```
  wmousectl(w);
```
Uses wmousectl() 216b.

⟨*function* wmousectl 216b⟩≡ (312b)
```
  void
  wmousectl(Window *w)
  {
      int but;

      if(w->mc.buttons == 1)
          but = 1;
      else if(w->mc.buttons == 2)
          but = 2;
      else if(w->mc.buttons == 4)
          but = 3;
      else{
          if(w->mc.buttons == 8)
              wkeyctl(w, Kscrolloneup);
          if(w->mc.buttons == 16)
              wkeyctl(w, Kscrollonedown);
          return;
      }

      incref(w);  /* hold up window while we track */
      <wmousectl() goto Return if window was deleted 216c>

      <wmousectl() if pt in scrollbar 218b>
      if(but == 1)
          wselect(w);

      /* else all is handled by main process */
    Return:
      wclose(w);
  }
```
Uses Kscrollonedown 219a, Kscrolloneup 219a, wclose() 111a, wkeyctl() 79e, and wselect() 269.

⟨wmousectl() *goto Return if window was deleted* 216c⟩≡ (216b)
```
  if(w->deleted)
      goto Return;
```

11.9 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 `w->scrolling` 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* 217a⟩≡ (215a)
```
  if(w->scrolling)
      menu2str[Scroll] = "noscroll";
  else
      menu2str[Scroll] = "scroll";
```
Uses Scroll-24 215e and menu2str 215d.

⟨button2menu() *cases* 217b⟩≡ (215a) 247a ▷
```
  case Scroll:
      if(w->scrolling ^= 1)
          wshow(w, w->nr);
      break;
```
Uses Scroll-24 215e and wshow() 190.

11.10 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`[218c] during window creation. When `frselectX` 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* 217c⟩≡ (162a)
```
  void (*scroll)(Frame*, int); /* scroll function provided by application */
```

⟨mousethread() *locals* 217d⟩+≡ (72c) ◁ 95c
```
  bool scrolling = false;
```

The `scrolling` local in `mousethread`[72c] (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* 217e⟩≡ (73c)
```
  if(winput->mouseopen)
      scrolling = false;
  else
    if(scrolling)
      scrolling = mouse->buttons;
    else
      scrolling = mouse->buttons && ptinrect(xy, winput->scrollr);
```
Uses mouse 58a.

⟨mousethread() *goto Sending if scroll buttons* 218a⟩≡ (73c)
```
/* the up and down scroll buttons are not subject to the usual rules */
if((mouse->buttons&(8|16)) && !winput->mouseopen)
    goto Sending;
```
Uses mouse 58a.

⟨wmousectl() *if pt in scrollbar* 218b⟩≡ (216b)
```
if(ptinrect(w->mc.xy, w->scrollr)){
    if(but)
        wscroll(w, but);
    goto Return;
}
```
Uses wscroll() 218c.

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`) 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`[196b] 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), `frcharofptX` 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`[219c] for as long as the button is held.

⟨*function* wscroll 218c⟩≡ (319)
```
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.xy.x<s.min.x || s.max.x<=w->mc.xy.x){
            readmouse(&w->mc);
        }else{
            my = w->mc.xy.y;
            if(my < s.min.y)
                my = s.min.y;
            if(my >= s.max.y)
                my = s.max.y;
            if(!eqpt(w->mc.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.Mouse);
                first = false;
            }
            wscrsleep(w, 100);
        }
    }while(w->mc.buttons & (1<<(but-1))); // >>
    while(w->mc.buttons)
        readmouse(&w->mc);
}
```

Uses `wbacknl()` 196b, `wmovemouse()` 118b, `wscrsleep()` 219c, and `wsetorigin()` 197a.

⟨*enum* `_anon_ (windows/rio/dat.h)` *2* 219a⟩≡ (299b)

```
enum
{
    Kscrolloneup = KF|0x20,
    Kscrollonedown = KF|0x21,
};
```

⟨*function* `freescrtemps` 219b⟩≡ (319)

```
void
freescrtemps(void)
{
    freeimage(scrtmp);
    scrtmp = nil;
}
```

Uses `scrtmp-74` 192h.

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` 219c⟩≡ (319)

```
void
wscrsleep(Window *w, uint dt)
```

```
{
    Timer *timer;
    int y, b;
    static Alt alts[3];

    timer = timerstart(dt);
    y = w->mc.xy.y;
    b = w->mc.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.Mouse;
    alts[1].op = CHANRCV;
    alts[2].op = CHANEND;
    for(;;)
        switch(alt(alts)){
        case 0:
            timerstop(timer);
            return;
        case 1:
            if(abs(w->mc.xy.y-y)>2 || w->mc.buttons!=b){
                timercancel(timer);
                return;
            }
            break;
        }
}
```

Uses `timercancel()` 281e, `timerstart()` 281f, and `timerstop()` 281d.

The `framescroll` callback is invoked by the frame library's `frselectX` 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`. 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`[269]) 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` 220a⟩≡ (312b)

```
/*
 * called from frame library
 */
void
framescroll(Frame *f, int dl)
{
    if(f != &selectwin->frm)
        error("frameselect not right frame");
    wframescroll(selectwin, dl);
}
```

Uses `error()` 292c, `selectwin-27` 268d, and `wframescroll()` 220b.

⟨*function* `wframescroll` 220b⟩≡ (312b)

```
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 `selectq-28` 268e, `wbacknl()` 196b, `wscrsleep()` 219c, `wsetorigin()` 197a, and `wsetselect()` 199.

11.11 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` 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`), re-initialized with the new rectangle (`frinit`), the scrollbar rectangle is recomputed, the background is repainted, visible text is refilled with `wfill`[198b], and the selection and scrollbar are redrawn.

This code is essentially a re-execution of the `wmk()`[98e] 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 221⟩≡                                  (116b)
  ⟨wresize() textual window updates, reset lastsr 192f⟩
  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 211d⟩
      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 Scrollgap 161c, Scrollwid 160c, Selborder 73d, cols-67 164a, wfill() 198b, wscrdraw() 191b, wsetcols() 164e, and wsetselect() 199.

11.12 /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` 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.

⟨qid *cases* 222a⟩+≡ (122d) ◁157a 224a▷

```
  Qtext,
```

⟨*dirtab array elements* 222b⟩+≡ (123b) ◁157b 224b▷

```
  { "text",  QTFILE, Qtext,  0400 },
```

Uses Qtext 222a.

⟨xfidread() *cases* 222c⟩+≡ (134a) ◁157d 224c▷

```
  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 222a, filsysrespond() 124a, and wcontents() 222d.

⟨*function* wcontents 222d⟩≡ (312b)

```
  char*
  wcontents(Window *w, int *ip)
  {
      return runetobyte(w->r, w->nr, ip);
  }
```

Uses runetobyte() 295b.

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. Through these files, programs can discover their own window identity (winid, label), query the screen (screen), access other windows (wsys/), and control windows programmatically (wctl). The full shape of the per-window filesystem that rio serves is the union of all the dirtab 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 ?? (QID(winid, qxxx)) is what lets the server tell them apart. The 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, which is how the `wctl(1)` command and tools like `acme`'s win interact with specific windows.

12.1 /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/N/wctl` when it needs to control a specific window (including its own). The implementation is straightforward—a single `sprint()`:

⟨qid *cases* 224a⟩+≡ (122d) ◁222a 224d▷
```
  Qwinid,
```

⟨*dirtab array elements* 224b⟩+≡ (123b) ◁222b 224e▷
```
  { "winid",  QTFILE, Qwinid,  0400 },
```
Uses `Qwinid` 224a.

⟨xfidread() *cases* 224c⟩+≡ (134a) ◁222c 224f▷
```
  case Qwinid:
      n = sprint(buf, "%11d ", w->id);
      t = estrdup(buf);
      goto Text;
```
Uses `Qwinid` 224a and `estrdup()` 294a.

12.2 /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):

⟨qid *cases* 224d⟩+≡ (122d) ◁224a 225b▷
```
  Qlabel,
```

⟨*dirtab array elements* 224e⟩+≡ (123b) ◁224b 225c▷
```
  { "label",  QTFILE, Qlabel,  0600 },
```
Uses `Qlabel` 224d.

⟨xfidread() *cases* 224f⟩+≡ (134a) ◁224c 225d▷
```
  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` 224d and `filsysrespond()` 124a.

⟨xfidwrite() *cases* 225a⟩+≡ (135a) ◁146g 231a▷

```
  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` 224d, `emalloc()` 293d, and `filsysrespond()` 124a.

12.3 /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 as `/dev/window` (see the GRAPHICS book [Pad16c]). This lets programs discover the full desktop geometry, which is useful for positioning new windows or for utilities like `lens` that operate on the whole screen. The implementation delegates to the same `caseImage` code path used by `/mnt/wsys/window`, passing `display->image` instead of the window's own image:

⟨qid *cases* 225b⟩+≡ (122d) ◁224d 225e▷

```
  Qscreen,
```

⟨*dirtab array elements* 225c⟩+≡ (123b) ◁224e 226a▷

```
  { "screen",  QTFILE, Qscreen, 0400 },
```

Uses `Qscreen` 225b.

⟨xfidread() *cases* 225d⟩+≡ (134a) ◁224f 229c▷

```
  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` 225b and `filsysrespond()` 124a.

12.4 /mnt/wsys/wsys/

`/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/N/` gives access to the same virtual device files (`cons`, `mouse`, `wctl`, etc.) as `/mnt/wsys/`, but for window N instead of the caller's own window. This is how the `wctl(1)` command can create, resize, or delete another window from a script. For example, a status bar program can enumerate all windows, read their labels, and display a window list. The implementation is interesting because it uses `filsyswalk()` 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/`:

⟨qid *cases* 225e⟩+≡ (122d) ◁225b 227a▷

```
  Qwsys,  /* directory of window directories */
```

⟨*dirtab array elements* 226a⟩+≡ (123b) ◁225c 227e▷

```
{ "wsys",  QTDIR, Qwsys,  0500|DMDIR },
```

Uses Qwsys 225e.

⟨*function* idcmp 226b⟩≡ (317)

```
static
int
idcmp(void *a, void *b)
{
    return *(int*)a - *(int*)b;
}
```

⟨filsysread() *other locals* 226c⟩+≡ (133a) ◁133c

```
int i, j, k;
int len;
int *ids;
Dirtab dt;
char buf[16];
```

⟨filsyswalk() *if Qwsys, then goto Accept* 226d⟩≡ (127d)

```
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 225e, Qwsysdir 227a, all 126c, dirtab 123b, numeric() 226e, winclosechan 110a, and wlookid() 127a.

⟨*function* numeric 226e⟩≡ (317)

```
static
int
numeric(char *s)
{
    for(; *s!='\0'; s++)
        if(*s<'0' || '9'<*s)
            return 0;
    return 1;
}
```

⟨filsysread() *cases* 226f⟩+≡ (133a) ◁133d

```
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 225e, all 126c, dostat() 136a, emalloc() 293d, idcmp() 226b, nwindow 61b, and windows 61a.

⟨qid *cases* 227a⟩+≡ (122d) ◁225e 227d▷
```
  Qwsysdir,  /* window directory, child of wsys */
```

⟨filsyswalk() *when in dotdot, if Qwsysdir adjust path* 227b⟩≡ (129c)
```
  if(FILE(qid) == Qwsysdir)
      path = Qwsys;
```
Uses FILE 122c, Qwsys 225e, and Qwsysdir 227a.

12.5 /mnt/wsys/wctl

The `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/N/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.

⟨Window *other fields* 227c⟩+≡ (59) ◁192d 229d▷
```
  bool wctlopen;
  bool  wctlready;
```

⟨qid *cases* 227d⟩+≡ (122d) ◁227a 248c▷
```
  Qwctl,
```

⟨*dirtab array elements* 227e⟩+≡ (123b) ◁226a 248d▷
```
  { "wctl",  QTFILE, Qwctl,  0600 },
```
Uses Qwctl 227d.

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* 228a⟩+≡ (131d) ◁148 248f▷

```
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` 290h, `Qwctl` 227d, `Wakeup` 286a, `filsysrespond()` 124a, and `wsendctlmesg()` 96c.

⟨`xfidclose()` *cases* 228b⟩+≡ (132b) ◁149a 249b▷

```
case Qwctl:
    if(x->f->mode==OREAD || x->f->mode==ORDWR)
        w->wctlopen = false;
break;
```

Uses `Qwctl` 227d.

⟨`wcurrent()` *wakeup w and oi* 228c⟩≡ (106e)

```
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` 286a and `wsendctlmesg()` 96c.

12.5.1 Reading

Reading `wctl` uses the same producer/consumer pattern as `/mnt/wsys/cons`: the worker thread (consumer) blocks on `w->wctlread` until the window thread (producer) sends the current geometry. The read blocks until the geometry *changes* (a resize, move, or focus change sets `wctlready` 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.

Producer

Consumer

⟨*enum* _anon_ (windows/rio/xfid.c)*5* 229a⟩≡ (318)
```
  enum { WCRdata, WCRflush, NWCR };
```

⟨xfidread() *other locals* 229b⟩+≡ (134a) ◁157c
```
  Consreadmesg cwrm;
```

⟨xfidread() *cases* 229c⟩+≡ (134a) ◁225d 249c▷
```
  case Qwctl: /* read returns rectangle, hangs if not resized */
      if(cnt < 4*12){
          filsysrespond(x->fs, x, &fc, Etooshort);
          break;
      }
      ⟨xfidxxx() set flushtag 283c⟩

      alts[WCRdata].c = w->wctlread;
      alts[WCRdata].v = &cwrm;
      alts[WCRdata].op = CHANRCV;
      ⟨xfidread() when Qwctl, set alts for flush 284h⟩
      alts[NMR].op = CHANEND;

      switch(alt(alts)){
      case WCRdata:
          break;
      ⟨xfidread() when Qwctl, switch alt flush case 285a⟩
      }

      /* received data */
      ⟨xfidxxx() unset flushtag 283d⟩
      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 285b⟩

      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` 291c, `NMR-12` 139d, `Qwctl` 227d, `WCRdata-13` 229a, and `filsysrespond()` 124a.

⟨Window *other fields* 229d⟩+≡ (59) ◁227c
```
  // chan<Consreadmesg> (listener = , sender = )
  Channel  *wctlread; /* chan(Consreadmesg) */
```

⟨wmk() *channels creation* 229e⟩+≡ (98e) ◁143b
```
  w->wctlread =  chancreate(sizeof(Consreadmesg), 0);
```

⟨winctl() *other locals* 229f⟩+≡ (78) ◁213c 230f▷
```
  Consreadmesg cwrm;
```

⟨winctl() *channels creation* 230a⟩+≡ (78) ◁212c
```
cwrm.c1 = chancreate(sizeof(Stringpair), 0);
cwrm.c2 = chancreate(sizeof(Stringpair), 0);
```

⟨winctl() *Wctl case, free channels if wctlmesg is Excited* 230b⟩+≡ (81a) ◁212d
```
chanfree(cwrm.c1);
chanfree(cwrm.c2);
```

⟨Wxxx *cases* 230c⟩+≡ (77b) ◁212a
```
WWread,
```

⟨winctl() *alts setup* 230d⟩+≡ (78) ◁212e
```
alts[WWread].c = w->wctlread;
alts[WWread].v = &cwrm;
alts[WWread].op = CHANSND;
```
Uses WWread-90 230c.

⟨winctl() *alts adjustments* 230e⟩+≡ (78) ◁212f
```
if(w->deleted || !w->wctlready)
    alts[WWread].op = CHANNOP;
else
    alts[WWread].op = CHANSND;
```
Uses WWread-90 230c.

⟨winctl() *other locals* 230f⟩+≡ (78) ◁229f
```
char *s;
```

⟨winctl() *event loop cases* 230g⟩+≡ (78) ◁213a
```
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-90 230c, hidden 120a, input 61e, and nhidden 120b.

12.5.2 Writing (controlling windows)

Writing to `wctl` accepts commands like "new", "resize", "move", "delete", "hide", "unhide", "top", "bottom", "scroll", "noscroll", and "set". The `parsewctl` 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 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* 231a⟩+≡ (135a) ◁225a 249e▷
```
  case Qwctl:
      if(writewctl(x, buf) < 0){
          filsysrespond(x->fs, x, &fc, buf);
          return;
      }
      flushimage(display, true);
      break;
```
Uses Qwctl 227d, filsysrespond() 124a, and writewctl() 231b.

⟨*function* writewctl 231b⟩≡ (320b)
```
  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){
      case New:
          return wctlnew(rect, arg, pid, hideit, scrollit, dir, err);
      case Set:
          if(pid > 0)
              wsetpid(w, pid, 0);
          return 1;
```

```
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;
case Scroll:
    w->scrolling = 1;
    wshow(w, w->nr);
    wsendctlmesg(w, Wakeup, ZR, nil);
    return 1;
case Noscroll:
    w->scrolling = 0;
    wsendctlmesg(w, Wakeup, ZR, nil);
    return 1;
case Top:
    wtopme(w);
    return 1;
case Bottom:
    wbottomme(w);
    return 1;
case Current:
    wcurrent(w);
    return 1;
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;
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;
    case Delete:
        wsendctlmesg(w, Deleted, ZR, nil);
        return 1;
    }
    strcpy(err, "invalid wctl message");
    return -1;
}
```

Uses Bottom-40 237b, Current-41 237b, Delete-44 237b, Deleted 108b, Ebadwr 290f, Ewalloc 290g, Hide-42 237b, Move-35 237b, New-33 237b, Noscroll-37 237b, Reshaped 96a, Resize-34 237b, Scroll-36 237b, Selborder 73d, Set-38 237b, Top-39 237b, Unhide-43 237b, Wakeup 286a, desktop 58b, goodrect() 239a, hidden 120a, mouse 58a, nhidden 120b, nwindow 61b, rectonscreen() 240d, red 58d, wbottomme() 244b, wctlnew() 243, wcurrent() 106e, whide() 120c, windows 61a, wsendctlmesg() 96c, wsetpid() 102c, wshow() 190, wtopme() 244a, and wunhide() 121e.

Chapter 13

Advanced Topics

This chapter covers features that go beyond `rio`'s basic operation: external access to `rio`'s filesystem (`/srv/rio.user` command-line window control (`/srv/riowctl.user.pid`), recursive `rio`, and terminal editing features like snarf/paste, plumbing, auto-completion, and word selection. It also covers `rio`'s command-line options (`rio -s`, `rio -i`, `rio -k`, `rio -f`), holding mode, signal handling, the timer subsystem, 9P flush support, and security.

13.1 External mount: `/srv/rio.user.pid`

Until now, we have seen `filsysmount()` 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()` 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` 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 (windows/rio/fsys.c)* 234a⟩≡ (316b)

```
char srvpipe[64];
```

⟨`filsysinit()` *srv pipe* 234b⟩≡ (69a)

```
/*
 * Post srv pipe
 */
sprint(srvpipe, "/srv/rio.%s.%d", fs->user, pid);
post(srvpipe, "wsys", fs->cfd);
```

Uses `post()` 234c and `srvpipe` 234a.

⟨*function* `post` 234c⟩≡ (316b)

```
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() 292c.

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.

⟨xfidattach() *other locals* 235a⟩+≡ (126d) ◁127b
```
Rectangle r;
int pid;
bool hideit = false;
bool scrollit;
char *dir, errbuf[ERRMAX];
Image *i;
```

When an external client attaches with a spec string starting with "new", xfidattach() parses it with the same parsewctl() 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. 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.

⟨xfidattach() *if mount "new ..."* 235b⟩≡ (126d)
```
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 291h, Ewindow 291i, Selborder 73d, desktop 58b, goodrect() 239a, new() 97b, and scrolling 272a.

13.2 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` runs in its own process doing blocking reads on the pipe, and sends the text over a channel to `wctlthread` in the main process where it is safe to manipulate window state.

⟨*global* `wctlfd` 236a⟩≡ (304)
```
int wctlfd;
```

⟨*global srvwctl (windows/rio/fsys.c)* 236b⟩≡ (316b)
```
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* 236c⟩≡ (69a)
```
/*
 * Create and post wctl pipe
 */
⟨filsysinit() create wctl pipe 236e⟩

/*
 * Start server processes
 */
⟨filsysinit() create wctl process and thread 236f⟩
```

⟨`filsysinit()` *other locals* 236d⟩≡ (69a) 285c▷
```
fdt p0;
// chan<??> (listener = ??, sender = ??)
Channel *c;
```

⟨`filsysinit()` *create wctl pipe* 236e⟩≡ (236c)
```
if(cexecpipe(&p0, &wctlfd) < 0)
    goto Rescue;
sprint(srvwctl, "/srv/riowctl.%s.%d", fs->user, pid);
post(srvwctl, "wctl", p0);
close(p0);
```
Uses `cexecpipe()` 69b, `post()` 234c, `srvwctl` 236b, and `wctlfd` 236a.

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* 236f⟩≡ (236c)
```
c = chancreate(sizeof(char*), 0);
if(c == nil)
    error("wctl channel");

proccreate(wctlproc, c, 4096);
threadcreate(wctlthread, c, 4096);
```
Uses `error()` 292c, `wctlproc()` 236g, and `wctlthread()` 237a.

`wctlproc` 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` 236g⟩≡ (320b)
```
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 `emalloc()` 293d, `messagesize` 81b, and `wctlfd` 236a.

`wctlthread` is the dispatching half, running in the main process. It receives command strings, parses them with `parsewctl()`, and currently only handles the `New` command—the other wctl commands (resize, move, top, etc.) are handled through `/dev/wctl` instead.

⟨*function* `wctlthread` 237a⟩≡ (320b)

```
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-33` 237b and `wctlnew()` 243.

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_ (windows/rio/wctl.c)` 237b⟩≡ (320b)

```
/* >= Top are disallowed if mouse button is pressed */
```

```
enum
{
    New,
    Resize,
    Move,
    Scroll,
    Noscroll,
    Set,
    Top,
    Bottom,
    Current,
    Hide,
    Unhide,
    Delete,
};
```

⟨*global* cmds 238a⟩≡ (320b)

```
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_ (windows/rio/wctl.c) *2* 238b⟩≡ (320b)

```
enum
{
    Cd,
    Deltax,
    Deltay,
    Hidden,
    Id,

    Maxx,
    Maxy,
    Minx,
    Miny,

    PID,
    R,

    Scrolling,
    Noscrolling,
};
```

⟨*global* params 238c⟩≡ (320b)

```
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
};
```

`parsewctl()` 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` 239a⟩≡ (308)

```
/*
 * 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` 193a.

⟨*function* `word` 239b⟩≡ (320b)

```
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 240a⟩≡ (320b)
```
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 240b⟩≡ (320b)
```
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()` 293a.

⟨*function* shift 240c⟩≡ (320b)
```
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 240d⟩≡ (320b)
```
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()` 240c.

⟨*function* riostrtol 241a⟩≡ (320b)

```
/* 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;
}
```

⟨*function* parsewctl 241b⟩≡ (320b)

```
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()` 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` 243⟩≡ (320b)

```
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 290f, Ewalloc 290g, Selborder 73d, desktop 58b, emalloc() 293d, goodrect() 239a, new() 97b, and red 58d.

⟨*function* wtopme 244a⟩≡ (311)

```
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-65 61c.

⟨*function* wbottomme 244b⟩≡ (311)

```
void
wbottomme(Window *w)
{
    if(w!=nil && w->i!=nil && !w->deleted){
        bottomwindow(w->i);
        flushimage(display, 1);
        w->topped = - ++topped;
    }
}
```

Uses topped-65 61c.

13.3 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.

⟨Mxxx *cases* 244c⟩≡ (72a)

```
MReshape,
```

In Plan 9, resize events are delivered through /dev/mouse as a special message type (the "r" prefix instead of "m"). This is somewhat ugly—piggybacking geometry changes onto the mouse device—but it means rio just needs to listen on its existing mousectl->resizec channel.

⟨mousethread() *alts setup* 244d⟩+≡ (72c) ◁72d

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

Uses MReshape-93 244c and mousectl 57b.

⟨mousethread() *event loop cases* 244e⟩+≡ (72c) ◁73c

```
case MReshape:
    resized();
    break;
```

Uses MReshape-93 244c and resized() 245.

`resized()` is a standard handler that any graphical application should implement. For `rio`, the logic is more involved: it must rebuild the entire `desktop`, then proportionally scale every window's rectangle to fit the new dimensions. Each window receives a `Reshaped` control message to redraw itself. 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.

⟨*function* `resized` 245⟩≡ (305b)

```
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 246a⟩
    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 246b⟩
        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 96a, background 58c, desktop 58b, error() 292c, freescrtemps() 219b, hidden 120a, nhidden 120b, nwindow 61b, viewr 57a, windows 61a, and wsendctlmesg() 96c.

This function is a nice summary of rio's full state: it iterates over all windows, distinguishes hidden from visible ones, and uses the Reshaped message to trigger each window thread's redraw logic.

⟨resized() *sanity check desktop* 246a⟩≡ (245)

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

Uses desktop 58b and error() 292c.

⟨resized() *continue if window was deleted* 246b⟩≡ (245)

```
    if(w->deleted)
        continue;
```

13.4 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.

13.4.1 Snarf

"Snarf" is Plan 9's term for copy. The cut/copy/paste operations work with a process-local snarf buffer (the snarf 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[79e] 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* 246c⟩≡ (204a)

```
    if(r != 0x7F){ // 0x7F = Interrupt key
        wsnarf(w);
        wcut(w);
    }
```

Uses wcut() 248a and wsnarf() 247b.

Snarf menu

The middle-click menu provides the full set of clipboard operations. Send is the interesting one: it pastes the snarf buffer at the end of the window's text (position w->nr) 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()` instead.

⟨`button2menu()` *cases* 247a⟩+≡ (215a) ◁217b 251a▷

```
case Cut:
    wsnarf(w);
    wcut(w);
    wscrdraw(w);
    break;

case Snarf:
    wsnarf(w);
    break;

case Paste:
    getsnarf();
    wpaste(w);
    wscrdraw(w);
    break;

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 `Cut-19` 215e, `Paste-20` 215e, `Send-23` 215e, `Snarf-21` 215e, `getsnarf()` 250h, `nsnarf` 250d, `snarf` 250e, `waddraw()` 153c, `wcut()` 248a, `winsert()` 175b, `wpaste()` 248b, `wscrdraw()` 191b, `wsetselect()` 199, `wshow()` 190, and `wsnarf()` 247b.

`wsnarf()` copies the selection into the global `snarf` buffer, bumps the version counter, and publishes to `/dev/snarf` so other programs can paste it. `wcut()` simply deletes the selected range. `wpaste()` 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` 247b⟩≡ (312b)

```
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` 250d, `putsnarf()` 250g, `runemove` 294e, `runerealloc` 294d, `snarf` 250e, and `snarfversion` 250f.

⟨*function* wcut 248a⟩≡ (312b)
```
void
wcut(Window *w)
{
    if(w->q1 == w->q0)
        return;
    wdelete(w, w->q0, w->q1);
    wsetselect(w, w->q0, w->q0);
}
```
Uses wdelete() 209c and wsetselect() 199.

⟨*function* wpaste 248b⟩≡ (312b)
```
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 250d, snarf 250e, waddraw() 153c, wcut() 248a, winsert() 175b, and wsetselect() 199.

/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.

⟨qid *cases* 248c⟩+≡ (122d) ◁227d 267d▷
```
Qsnarf,
```

⟨*dirtab array elements* 248d⟩+≡ (123b) ◁227e 267e▷
```
{ "snarf",  QTFILE, Qsnarf,  0600 },
```
Uses Qsnarf 248c.

⟨filsyswalk() *if snarf* 248e⟩≡ (127d)
```
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 250b.

⟨xfidopen() *cases* 248f⟩+≡ (131d) ◁228a 275a▷
```
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` 248c, `ntsnarf-3` 250a, and `tsnarf-2` 249g.

⟨`xfidclose()` *other locals* 249a⟩≡ (132b)

```
  int nb, nulls;
```

⟨`xfidclose()` *cases* 249b⟩+≡ (132b) ◁228b

```
  /* 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` 248c, `cvttorunes()` 294f, `nsnarf` 250d, `ntsnarf-3` 250a, `runerealloc` 294d, `snarf` 250e, and `tsnarf-2` 249g.

⟨`xfidread()` *cases* 249c⟩+≡ (134a) ◁229c 267f▷

```
  /* 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` 248c, `getsnarf()` 250h, `nsnarf` 250d, `runetobyte()` 295b, and `snarf` 250e.

⟨*constant* `MAXSNARF` 249d⟩≡ (318)

```
  #define MAXSNARF 100*1024
```

⟨`xfidwrite()` *cases* 249e⟩+≡ (135a) ◁231a 268a▷

```
  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` 291f, `MAXSNARF-1` 249d, `Qsnarf` 248c, `erealloc()` 293c, `filsysrespond()` 124a, `ntsnarf-3` 250a, `snarfversion` 250f, and `tsnarf-2` 249g.

⟨`dostat()` *adjust vers for snarf* 249f⟩≡ (136a)

```
  if(dir->qid == Qsnarf)
      d.qid.vers = snarfversion;
```

Uses `Qsnarf` 248c and `snarfversion` 250f.

⟨*global* `tsnarf` 249g⟩≡ (318)

```
  static char *tsnarf;
```

⟨*global* ntsnarf 250a⟩≡ (318)
```
static int ntsnarf;
```

⟨*global* snarffd 250b⟩≡ (315a)
```
fdt  snarffd;
```

⟨main() *set some globals* 250c⟩+≡ (66a) ◁211g 276c▷
```
snarffd = open("/dev/snarf", OREAD|OCEXEC);
```
Uses snarffd 250b.

⟨*global* nsnarf 250d⟩≡ (315a)
```
int  nsnarf;
```

⟨*global* snarf 250e⟩≡ (315a)
```
Rune* snarf;
```

⟨*global* snarfversion 250f⟩≡ (304)
```
int  snarfversion; /* updated each time it is written */
```

`putsnarf()` 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 250g⟩≡ (315a)
```
/*
 * /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 250d, snarf 250e, and snarffd 250b.

⟨*function* getsnarf 250h⟩≡ (315a)
```
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()` 294f, `erealloc()` 293c, `nsnarf` 250d, `runerealloc` 294d, `snarf` 250e, and `snarffd` 250b.

13.4.2 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).

⟨`button2menu()` *cases* 251a⟩+≡ (215a) ◁247a

```
case Plumb:
    wplumb(w);
    break;
```

Uses `Plumb-22` 215e and `wplumb()` 251b.

`wplumb()` constructs a `Plumbmsg` 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` 251b⟩≡ (312b)

```
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 `emalloc()` 293d, `estrdup()` 294a, `lastcursor` 90a, `messagesize` 81b, `query` 87d, `riosetcursor()` 90b, and `runetobyte()` 295b.

⟨*struct* `Plumbmsg` 252a⟩≡ (298)

```
struct Plumbmsg
{
    char  *src;
    char  *dst;
    char  *wdir;
    char  *type;
    Plumbattr  *attr;
    int   ndata;
    char  *data;
};
```

⟨*struct* `Plumbattr` 252b⟩≡ (298)

```
struct Plumbattr
{
    char  *name;
    char  *value;
    Plumbattr  *next;
};
```

⟨*function* `plumbsendtext` 252c⟩≡ (346)

```
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);
}
```

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()` 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` 253a⟩≡ (345b)

```
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` 253b⟩≡ (345b)

```
static int
Strlen(char *s)
```

```
{
    if(s == nil)
        return 0;
    return strlen(s);
}
```

⟨*function* Strcpy 254a⟩≡ (345b)

```
static char*
Strcpy(char *s, char *t)
{
    if(t == nil)
        return s;
    return strcpy(s, t) + strlen(t);
}
```

⟨*function* quote 254b⟩≡ (345b)

```
/* 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 254c⟩≡ (345b)

```
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;
}
```

⟨*function* plumblookup 255a⟩≡ (345b)

```
char*
plumblookup(Plumbattr *attr, char *name)
{
    while(attr){
        if(strcmp(attr->name, name) == 0)
            return attr->value;
        attr = attr->next;
    }
    return nil;
}
```

⟨*function* plumbpack 255b⟩≡ (345b)

```
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;
}
```

⟨*function* plumbsend 256a⟩≡ (345b)

```
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;
}
```

⟨*function* plumbline 256b⟩≡ (345b)

```
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 256c⟩≡ (345b)

```
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 257⟩≡ (345b)

```
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 258a⟩≡ (345b)

```
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 258b⟩≡ (345b)

```
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 259⟩≡ (345b)

```
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;
}
```

⟨*function* `plumbunpack` 260a⟩≡ (345b)

```
Plumbmsg*
plumbunpack(char *buf, int n)
{
    return plumbunpackpartial(buf, n, nil);
}
```

⟨*function* `plumbrecv` 260b⟩≡ (345b)

```
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;
}
```

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` structure buffers partial messages until all bytes arrive. `plumbevent()` 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` 260c⟩≡ (345a)

```
struct EQueue
{
    int  id;
    char  *buf;
    int  nbuf;
    EQueue *next;
};
```

⟨*global* `equeue` 260d⟩≡ (345a)

```
static EQueue *equeue;
```

⟨*global* `eqlock` 260e⟩≡ (345a)

```
static Lock  eqlock;
```

⟨*function* partial 261a⟩≡ (345a)
```
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;
}
```

⟨*function* addpartial 261b⟩≡ (345a)
```
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);
}
```

⟨*function* plumbevent 262a⟩≡ (345a)

```
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;
}
```

⟨*function* eplumb 262b⟩≡ (345a)

```
int
eplumb(int key, char *port)
{
    int fd;

    fd = plumbopen(port, OREAD|OCEXEC);
    if(fd < 0)
        return -1;
    return estartfn(key, fd, 8192, plumbevent);
}
```

13.4.3 Auto complete

Pressing Ctrl-F or Insert triggers filename completion, similar to Tab completion in shells. `namecomplete()` works backward from the cursor to extract the current word and its path prefix, resolves relative paths against the window's working directory, 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* 262c⟩+≡ (204a) ◁210c

```
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()` 263b, `winsert()` 175b, and `wshow()` 190.

⟨*struct* Completion 263a⟩≡ (297a)

```
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 */
};
```

⟨*function* namecomplete 263b⟩≡ (312b)

```
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 runemalloc 294c, runemove 294e, showcandidates() 264a, and windfilewidth() 264b.

⟨*function* showcandidates 264a⟩≡ (312b)
```
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() 175b and wsetselect() 199.

⟨*function* windfilewidth 264b⟩≡ (312b)
```
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()` 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` 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 265⟩≡ (341a)

```
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;
}
```

⟨*function* `longestprefixlength` 267a⟩≡ (341a)
```
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` 267b⟩≡ (341a)
```
void
freecompletion(Completion *c)
{
    if(c){
        free(c->filename);
        free(c);
    }
}
```

⟨*function* `strpcmp` 267c⟩≡ (341a)
```
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 `w->dir`, 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.

⟨`qid` *cases* 267d⟩+≡ (122d) ◁248c 274f▷
```
Qwdir,
```

⟨*dirtab array elements* 267e⟩+≡ (123b) ◁248d 274g▷
```
{ "wdir",  QTFILE, Qwdir,  0600 },
```
Uses `Qwdir` 267d.

⟨`xfidread()` *cases* 267f⟩+≡ (134a) ◁249c
```
case Qwdir:
    t = estrdup(w->dir);
    n = strlen(t);
    goto Text;
```
Uses `Qwdir` 267d and `estrdup()` 294a.

⟨xfidwrite() *cases* 268a⟩+≡ (135a) ◁249e 275b▷
```
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` 267d, `emalloc()` 293d, and `estrdup()` 294a.

13.4.4 Word selection

`wselect()` 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.

⟨*global* clickwin 268b⟩≡ (312b)
```
static Window *clickwin;
```

⟨*global* clickmsec 268c⟩≡ (312b)
```
static uint clickmsec;
```

⟨*global* selectwin 268d⟩≡ (312b)
```
static Window *selectwin;
```

⟨*global* selectq 268e⟩≡ (312b)
```
static uint selectq;
```

The `wselect` function handles button 1 click in a textual window. It first calls `frselectX` 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` 269⟩≡ (312b)

```
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.buttons;
    q0 = w->q0;
    q1 = w->q1;
    selectq = w->org + frcharofpt(frm, w->mc.xy);
    if(clickwin==w && w->mc.msec-clickmsec<500)
    if(q0==q1 && selectq==w->q0){
        wdoubleclick(w, &q0, &q1);
        wsetselect(w, q0, q1);
        flushimage(display, 1);
        x = w->mc.xy.x;
        y = w->mc.xy.y;
        /* stay here until something interesting happens */
        do
            readmouse(&w->mc);
        while(w->mc.buttons==b && abs(w->mc.xy.x-x)<3 && abs(w->mc.xy.y-y)<3);
        w->mc.xy.x = x; /* in case we're calling frselect */
        w->mc.xy.y = y;
        q0 = w->q0; /* may have changed */
        q1 = w->q1;
        selectq = q0;
    }
    if(w->mc.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.msec-clickmsec<500){
            wdoubleclick(w, &q0, &q1);
            clickwin = nil;
        }else{
            clickwin = w;
            clickmsec = w->mc.msec;
        }
```

```
    }else
        clickwin = nil;
    wsetselect(w, q0, q1);
    flushimage(display, 1);
    while(w->mc.buttons){
        w->mc.msec = 0;
        b = w->mc.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.buttons == b)
            readmouse(&w->mc);
        clickwin = nil;
    }
}
```

Uses `clickmsec-26` 268c, `clickwin-25` 268b, `framescroll()` 220a, `getsnarf()` 250h, `selectq-28` 268e, `selectwin-27` 268d, `wcut()` 248a, `wdoubleclick()` 271a, `wpaste()` 248b, `wscrdraw()` 191b, `wsetselect()` 199, and `wsnarf()` 247b.

⟨*global* `left1` 270a⟩≡ (312b)

```
static Rune left1[] =  { L'{', L'[', L'(', L'<', L'«', 0 };
```

⟨*global* `right1` 270b⟩≡ (312b)

```
static Rune right1[] = { L'}', L']', L')', L'>', L'»', 0 };
```

⟨*global* `left2` 270c⟩≡ (312b)

```
static Rune left2[] =  { L'\n', 0 };
```

⟨*global* `left3` 270d⟩≡ (312b)

```
static Rune left3[] =  { L'\'', L'"', L'`', 0 };
```

⟨*global* `left` 270e⟩≡ (312b)

```
Rune *left[] = {
    left1,
    left2,
    left3,
    nil
};
```

Uses `left1-29` 270a, `left2-31` 270c, and `left3-32` 270d.

⟨*global* `right` 270f⟩≡ (312b)

```
Rune *right[] = {
    right1,
    left2,
    left3,
    nil
};
```

Uses `left2-31` 270c, `left3-32` 270d, and `right1-30` 270b.

`wdoubleclick()` 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` 271a⟩≡ (312b)

```
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()` 294b, `left` 270e, `right` 270f, `strrune()` 295a, and `wclickmatch()` 271b.

`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` 271b⟩≡ (312b)

```
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;
}
```

13.5 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.

⟨*global* `scrolling` 272a⟩≡ (304)

```
bool  scrolling;
```

⟨`main()` *command line processing* 272b⟩≡ (66a) 272d ▷

```
case 's':
    scrolling = true;
    break;
```

Uses `scrolling` 272a.

13.6 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* 272c⟩+≡ (66a) ◁ 211f 273c ▷

```
char *initstr = nil;
```

⟨`main()` *command line processing* 272d⟩+≡ (66a) ◁ 272b 273d ▷

```
case 'i':
    initstr = ARGF();
    if(initstr == nil)
        usage();
    break;
```

Uses `usage()` 27.

⟨main() *if initstr or kdbin* 273a⟩≡ (292b) 274a▷

```
  if(initstr)
      proccreate(initcmd, initstr, STACK);
```

Uses STACK 68d and initcmd() 273b.

⟨*function* initcmd 273b⟩≡ (305a)

```
  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");
  }
```

13.7 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.

13.7.1 `rio -k`

⟨main() *locals* 273c⟩+≡ (66a) ◁272c 273g▷

```
  char *kbdin = nil;
```

⟨main() *command line processing* 273d⟩+≡ (66a) ◁272d 276b▷

```
  case 'k':
      if(kbdin != nil)
          usage();
      kbdin = ARGF();
      if(kbdin == nil)
          usage();
      break;
```

Uses usage() 27.

⟨*global* wkeyboard 273e⟩≡ (304)

```
  Window *wkeyboard; /* window of simulated keyboard */
```

⟨*global* kbdargv 273f⟩≡ (305a)

```
  char *kbdargv[] = { "rc", "-c", nil, nil };
```

⟨main() *locals* 273g⟩+≡ (66a) ◁273c

```
  Image *i;
  Rectangle r;
```

⟨main() *if initstr or kdbin* 274a⟩+≡ (292b) ◁273a

```
  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");
  }
```

Uses `desktop` 58b, `error()` 292c, `kbdargv` 273f, `new()` 97b, `r` 88e, `scrolling` 272a, and `wkeyboard` 273e.

13.7.2 wkeyboard

⟨mousethread() *if wkeyboard and button 6* 274b⟩≡ (73c)

```
  if(wkeyboard!=nil && (mouse->buttons & (1<<5))){
      keyboardhide();
      break;
  }
```

Uses `keyboardhide()` 275d, `mouse` 58a, and `wkeyboard` 273e.

⟨mousethread() *if wkeyboard and ptinrect* 274c⟩≡ (73c)

```
  /* 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` 58a, `wkeyboard` 273e, and `wtopme()` 244a.

⟨wcurrent() *if wkeyboard* 274d⟩≡ (106e)

```
  if(wkeyboard!=nil && w==wkeyboard)
      return;
```

Uses `wkeyboard` 273e.

⟨wclosewin() *if wkeyboard* 274e⟩≡ (108f)

```
  if(w == wkeyboard)
      wkeyboard = nil;
```

Uses `wkeyboard` 273e.

13.7.3 /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.

⟨qid *cases* 274f⟩+≡ (122d) ◁267d

```
  Qkbdin,
```

⟨*dirtab array elements* 274g⟩+≡ (123b) ◁267e

```
  { "kbdin",  QTFILE, Qkbdin,  0200 },
```

Uses `Qkbdin` 274f.

⟨xfidopen() *cases* 275a⟩+≡ (131d) ◁248f

```
  case Qkbdin:
      if(w !=  wkeyboard){
          filsysrespond(x->fs, x, &fc, Eperm);
          return;
      }
      break;
```

Uses `Eperm` 290a, `Qkbdin` 274f, `filsysrespond()` 124a, and `wkeyboard` 273e.

⟨xfidwrite() *cases* 275b⟩+≡ (135a) ◁268a

```
  case Qkbdin:
      keyboardsend(req->data, cnt);
      break;
```

Uses `Qkbdin` 274f and `keyboardsend()` 275c.

⟨*function* keyboardsend 275c⟩≡ (318)

```
  /*
   * 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()` 294f, `keyboardctl` 57c, and `runemalloc` 294c.

13.7.4 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.

⟨*function* keyboardhide 275d⟩≡ (305b)

```
  /*
   * 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` 58a, `mousectl` 57b, and `wkeyboard` 273e.

13.8 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` 276a⟩≡ (305a)
```
char  *fontname;
```

⟨`main()` *command line processing* 276b⟩+≡ (66a) ◁273d
```
case 'f':
    fontname = ARGF();
    if(fontname == nil)
        usage();
    break;
```
Uses `fontname` 276a and `usage()` 27.

⟨`main()` *set some globals* 276c⟩+≡ (66a) ◁250c
```
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);
```
Uses `fontname` 276a.

13.9 Holding mode

Holding mode freezes a window's output: when `w->holding` 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.

⟨`Window` *config fields* 276d⟩+≡ (59) ◁149b
```
bool holding;
```

⟨`winctl()` *alts adjustments, if holding* 276e⟩≡ (154g)
```
if(w->holding)
    alts[WCread].op = CHANNOP;
```
Uses `WCread-88` 154b.

⟨`wsetcursor()` *if holding* 276f⟩≡ (91d)
```
if(p==nil && w->holding)
    p = &whitearrow;
```
Uses `whitearrow` 87c.

⟨wborder() *if holding* 277a⟩≡ (99c)
```
  if(w->holding){
      if(type == Selborder)
          col = holdcol;
      else
          col = paleholdcol;
  }
```
Uses Selborder 73d, holdcol-71 277c, and paleholdcol-73 277d.

⟨wsetcols() *if holding* 277b⟩≡ (164e)
```
  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 holdcol-71 277c, input 61e, and lightholdcol-72 277e.

⟨*global* holdcol 277c⟩≡ (311)
```
  static Image *holdcol;
```

⟨*global* paleholdcol 277d⟩≡ (311)
```
  static Image *paleholdcol;
```

⟨*global* lightholdcol 277e⟩≡ (311)
```
  static Image *lightholdcol;
```

⟨wmk() *extra colors initialisation* 277f⟩+≡ (164c) ◁164g
```
  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-71 277c, lightholdcol-72 277e, and paleholdcol-73 277d.

⟨Wctlmesgkind *cases* 277g⟩+≡ (96a) ◁149e 286a▷
```
  Holdon,
  Holdoff,
```

⟨xfidwrite() *Qconsctl case* 277h⟩+≡ (146b) ◁149c
```
  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 277g, Holdon 277g, and wsendctlmesg() 96c.

⟨xfidclose() *Qconsctl case, if holding* 277i⟩≡ (146a)
```
  if(w->holding){
      w->holding = false;
      wsendctlmesg(w, Holdoff, ZR, nil);
  }
```
Uses Holdoff 277g and wsendctlmesg() 96c.

⟨`xfidwrite()` *Qconsctl case, if rawon message and holding mode* 278a⟩≡ (149c)
```
if(w->holding){
    w->holding = false;
    wsendctlmesg(w, Holdoff, ZR, nil);
}
```
Uses `Holdoff` 277g and `wsendctlmesg()` 96c.

⟨`wkeyctl()` *if holding* 278b⟩≡ (79e)
```
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()` 107a.

⟨`wctlmesg()` *cases* 278c⟩+≡ (96d) ◁150a 286b▷
```
case Holdon:
case Holdoff:
    ⟨wctlmesg() break if window was deleted 108e⟩
    wrepaint(w);
    flushimage(display, true);
    break;
```
Uses `Holdoff` 277g, `Holdon` 277g, and `wrepaint()` 107a.

13.10 Signals, notes

When `rio` receives a note (signal), `shutdown()` first sends "hangup" to all window processes via `killprocs()`, 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` 278d⟩≡ (305a)
```
char *oknotes[] =
{
    "delete",
    "hangup",
    "kill",
    "exit",
    nil
};
```

⟨*function* `shutdown` 278e⟩≡ (305a)
```
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() 279a and oknotes 278d.

⟨*function* killprocs 279a⟩≡ (305a)
```
void
killprocs(void)
{
    int i;

    for(i=0; i<nwindow; i++)
        postnote(PNGROUP, windows[i]->pid, "hangup");
}
```
Uses nwindow 61b and windows 61a.

13.11 Timer

The timer subsystem provides cancellable one-shot timers, used primarily by `wscrsleep()` for scroll bar repeat delays. The design uses a dedicated `timerproc` that polls at 1ms intervals, decrementing each timer's `dt` countdown. When a timer fires, it sends on the timer's channel; when cancelled, it is recycled into a freelist.

⟨*global* clockfd 279b⟩≡ (317)
```
fdt clockfd;
```

⟨filsysinit() *set clockfd* 279c⟩≡ (69a)
```
clockfd = open("/dev/time", OREAD|OCEXEC);
```
Uses clockfd 279b.

⟨*function* getclock 279d⟩≡ (317)
```
static
uint
getclock(void)
{
    char buf[32];

    seek(clockfd, 0, 0);
    read(clockfd, buf, sizeof buf);
    return atoi(buf);
}
```
Uses clockfd 279b.

⟨*struct* Timer 279e⟩≡ (299b)
```
struct Timer
{
    int  dt;
    int  cancel;
    Channel *c; /* chan(int) */
    Timer *next;
};
```

⟨main() *threads creation* 279f⟩+≡ (66a) ◁110c
```
timerinit();
```
Uses timerinit() 280a.

⟨*function* timerinit 280a⟩≡ (320a)

```
void
timerinit(void)
{
    ctimer = chancreate(sizeof(Timer*), 100);
    proccreate(timerproc, nil, STACK);
}
```

Uses STACK 68d, ctimer-82 281a, and timerproc() 280b.

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 280b⟩≡ (320a)

```
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-82 281a, msec() 281c, and timerstop() 281d.

⟨*global* ctimer 281a⟩≡ (320a)

```
// chan<?> (listener = ?, sender = ?)
static Channel* ctimer; /* chan(Timer*)[100] */
```

⟨*global* timer 281b⟩≡ (320a)

```
static Timer *timer;
```

⟨*function* msec 281c⟩≡ (320a)

```
static
uint
msec(void)
{
    return nsec()/1000000;
}
```

⟨*function* timerstop 281d⟩≡ (320a)

```
void
timerstop(Timer *t)
{
    t->next = timer;
    timer = t;
}
```

Uses timer-83 281b.

⟨*function* timercancel 281e⟩≡ (320a)

```
void
timercancel(Timer *t)
{
    t->cancel = true;
}
```

Timer allocation uses a freelist (`timer`) to avoid repeated `malloc`/`free` for the common case of scroll delays. `timerstart()` either reuses a freed timer or allocates a new one, sets the countdown and sends it to `timerproc` via the `ctimer` channel.

⟨*function* timerstart 281f⟩≡ (320a)

```
/*
 * 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-82` 281a, `emalloc()` 293d, and `timer-83` 281b.

13.12 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` thread handling the original request may be blocked waiting on a channel. The approach uses a combination of reference counting and a `flushc` notification channel. `xfidflush()` 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* 282a⟩≡ (64c)

```
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* 282b⟩≡ (84d)

```
x->flushc = chancreate(sizeof(int), 0); /* notification only; nodata */
x->flushtag = -1;
```

⟨`Xfid` *other fields* 282c⟩≡ (64c)

```
QLock active;
```

⟨`fcall` *other methods* 282d⟩+≡ (65) ◁136b 285e▷

```
[Tflush]   = filsysflush,
```

Uses `filsysflush()` 282e.

⟨*function* `filsysflush` 282e⟩≡ (317)

```
static
Xfid*
filsysflush(Filsys*, Xfid *x, Fid*)
{
    sendp(x->c, xfidflush);
    return nil;
}
```

Uses `xfidflush()` 283a.

⟨*function* xfidflush 283a⟩≡ (307a)
```
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-78` 70b, `error()` 292c, `filsysrespond()` 124a, and `xfid-76` 84a.

⟨*function* filsyscancel 283b⟩≡ (316a)
```
void
filsyscancel(Xfid *x)
{
    if(x->buf){
        free(x->buf);
        x->buf = nil;
    }
}
```

⟨`xfidxxx()` *set flushtag* 283c⟩≡ (229c 144b 142f 139f)
```
x->flushtag = req->tag;
```

⟨`xfidxxx()` *unset flushtag* 283d⟩≡ (229c 144b 142f 139f)
```
x->flushtag = -1;
```

⟨`xfidread()` *when Qmouse, set alts for flush* 283e⟩≡ (139f)
```
alts[MRflush].c = x->flushc;
alts[MRflush].v = nil;
alts[MRflush].op = CHANRCV;
```
Uses `MRflush-11` 139d.

⟨`xfidread()` *when Qmouse, switch alt flush case* 283f⟩≡ (139f)
```
case MRflush:
    filsyscancel(x);
    return;
```
Uses `MRflush-11` 139d and `filsyscancel()` 283b.

⟨xfidread() *when Qmouse, if flushing* 284a⟩≡ (139f)
```
  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() 283b.

⟨xfidread() *when Qcons, set alts for flush* 284b⟩≡ (142f)
```
  alts[CRflush].c = x->flushc;
  alts[CRflush].v = nil;
  alts[CRflush].op = CHANRCV;
```
Uses CRflush-8 142d.

⟨xfidread() *when Qcons, switch alt flush case* 284c⟩≡ (142f)
```
  case CRflush:
      filsyscancel(x);
      return;
```
Uses CRflush-8 142d and filsyscancel() 283b.

⟨xfidread() *when Qcons, if flushing* 284d⟩≡ (142f)
```
  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() 283b.

⟨xfidwrite() *when Qcons, set alts for flush* 284e⟩≡ (144b)
```
  alts[CWflush].c = x->flushc;
  alts[CWflush].v = nil;
  alts[CWflush].op = CHANRCV;
```
Uses CWflush-5 143d.

⟨xfidwrite() *when Qcons, switch alt flush case* 284f⟩≡ (144b)
```
  case CWflush:
      filsyscancel(x);
      return;
```
Uses CWflush-5 143d and filsyscancel() 283b.

⟨xfidwrite() *when Qcons, if flushing* 284g⟩≡ (144b)
```
  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() 283b and runemalloc 294c.

⟨xfidread() *when Qwctl, set alts for flush* 284h⟩≡ (229c)
```
  alts[WCRflush].c = x->flushc;
  alts[WCRflush].v = nil;
  alts[WCRflush].op = CHANRCV;
```
Uses WCRflush-14 229a.

⟨xfidread() *when Qwctl, switch alt flush case* 285a⟩≡ (229c)
```
case WCRflush:
    filsyscancel(x);
    return;
```
Uses WCRflush-14 229a and filsyscancel() 283b.

⟨xfidread() *when Qwctl, if flushing* 285b⟩≡ (229c)
```
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() 283b.

13.13 Security

rio's security model is minimal. It reads the username from /dev/user at startup and stores it in fs->user, 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. As the author notes, even macOS doesn't really improve on this: its security ultimately relies on app review rather than technical isolation.

⟨filsysinit() *other locals* 285c⟩+≡ (69a) ◁236d
```
fdt fd;
char buf[128];
int n;
```

⟨filsysinit() *set fs user* 285d⟩≡ (69a)
```
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);
```
Uses estrdup() 294a.

⟨fcall *other methods* 285e⟩+≡ (65) ◁282d
```
[Tauth]    = filsysauth,
```
Uses filsysauth() 285f.

⟨*function* filsysauth 285f⟩≡ (317)
```
static
Xfid*
filsysauth(Filsys *fs, Xfid *x, Fid*)
{
    Fcall fc;

    return filsysrespond(fs, x, &fc, "rio: authentication not required");
}
```
Uses filsysrespond() 124a.

13.14 TODO Wakeup

⟨Wctlmesgkind *cases* 286a⟩+≡ (96a) ◁277g 286c▷
```
  Wakeup,
```

⟨wctlmesg() *cases* 286b⟩+≡ (96d) ◁278c 286d▷
```
  case Wakeup:
      break;
```
Uses Wakeup 286a.

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).

13.15 TODO Refresh

Refresh is sent when an application closes /dev/mouse, signaling that it is done drawing directly on the window image. The handler redraws the entire text frame and borders, restoring the window's normal appearance. It skips the refresh if the window is deleted, has zero width, or the mouse is still open.

⟨Wctlmesgkind *cases* 286c⟩+≡ (96a) ◁286a
```
  Refresh,
```

⟨wctlmesg() *cases* 286d⟩+≡ (96d) ◁286b
```
  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 286c and wrefresh() 286e.

⟨*function* wrefresh 286e⟩≡ (311)
```
  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 Selborder 73d, Unselborder 107b, input 61e, wborder() 99c, and wscrdraw() 191b.

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 application. More generally, you now understand how a windowing system manages multiple windows, dispatches input, and provides a transparent 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). Each window provides virtual `/dev/cons`, `/dev/mouse`, and `/dev/winname` files, so programs running inside a window see the same interface as programs running on a bare console—they do not even need to know they are running in a window. This multiplexer design has remarkable consequences: windows are composable (you can run `rio` inside `rio`), any program that reads and writes `/dev/` files automatically works in a windowed environment, and `rio` gains network transparency for free through 9P.

14.1 Patterns and techniques

These techniques apply far beyond window managers:

- *Active objects*: one thread per window, communicating by message passing. This is how Erlang processes and Go services are structured: eliminate locks by giving each object exclusive ownership of its state.

- *Master/worker dispatch*: the `fileserver` proc dispatches requests to a pool of reusable workers. The same architecture is used by web servers (Nginx, Apache prefork) and database connection pools: decouple request arrival rate from processing capacity.

- *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.

- *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 like LMAX Disruptor—constant-time enqueue and dequeue with bounded memory.

- *Async IO offloading*: blocking IO is moved to separate procs that forward events through channels. 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).

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].
- WIDGETS book [Pad26]: the widget toolkit runs on top of `rio`, using the virtual devices that `rio` provides. The WIDGETS book [Pad26] covers the higher-level UI components (buttons, scrollbars, menus) that applications use inside `rio` windows.
- 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`.
- 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` composites windows in software using the draw device's layer mechanism.
- *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.
- *Display protocols*: X11 provides network-transparent graphics through a complex client-server protocol with hundreds of request types, extensions (GLX, XRender, XInput2), and a separate window manager process. Wayland simplifies this by having clients render locally and pass buffers to the compositor. `rio`'s approach—serving virtual devices over 9P—achieves network transparency more naturally, since 9P can be mounted across the network.
- *Accessibility*: modern windowing systems provide accessibility APIs (screen readers, magnification, keyboard navigation) as a core feature. `rio` does not address accessibility.
- *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.

`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 (import a remote `rio`), and simplicity (programs are unaware of windowing) from a single mechanism. Modern systems achieve richer visual results, but with far more complex architectures.

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—which is frozen when `rio` misbehaves. Two tricks help. 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 13.3), 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 in `fsys.c`. When enabled, `filsysproc` dumps each incoming 9P message on `STDERR` and `filsysrespond()` 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` 289a⟩≡ (299b)

```
#define DEBUG false
```

⟨`filsysinit()` *install dumper* 289b⟩≡ (69a)

```
fmtinstall('F', fcallfmt);
```

⟨`filsysproc()` *dump Fcall if debug* 289c⟩≡ (81c)

```
if(DEBUG)
    fprint(STDERR, "rio:<-%F\n", &x->req);
```

Uses `DEBUG` 289a.

⟨`filsysrespond()` *dump Fcall t if debug* 289d⟩≡ (124a)

```
if(DEBUG)
    fprint(STDERR, "rio:->%F\n", fc);
```

Uses `DEBUG` 289a.

Appendix B

Error Management

`rio`'s error strings are all lowercase, terse, and stored as global `char` arrays because they cross a 9P protocol boundary: when `filsysrespond()`[124a] is called with an error, the string is copied into an `Rerror` 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. The conventions match those of other Plan 9 fileservers (e.g., "permission denied", "file does not exist"), which means a client program written to expect Plan 9-style error reporting can display rio-produced errors without knowing they came from `rio` specifically.

B.1 Error codes

⟨*global* `Eperm` 290a⟩≡ (321a)
```
char Eperm[] = "permission denied";
```
Uses `Eperm` 290a.

⟨*global* `Eexist` 290b⟩≡ (317)
```
char Eexist[] = "file does not exist";
```
Uses `Eexist` 290b.

⟨*global* `Enotdir` 290c⟩≡ (317)
```
char Enotdir[] = "not a directory";
```
Uses `Enotdir` 290c.

⟨*global* `Ebadfcall` 290d⟩≡ (317)
```
char Ebadfcall[] = "bad fcall type";
```
Uses `Ebadfcall` 290d.

⟨*global* `Eoffset` 290e⟩≡ (317)
```
char Eoffset[] = "illegal offset";
```
Uses `Eoffset` 290e.

⟨*global* `Ebadwr` 290f⟩≡ (320b)
```
char Ebadwr[]  = "bad rectangle in wctl request";
```
Uses `Ebadwr` 290f.

⟨*global* `Ewalloc` 290g⟩≡ (320b)
```
char Ewalloc[]  = "window allocation failed in wctl request";
```
Uses `Ewalloc` 290g.

⟨*global* `Einuse` 290h⟩≡ (318)
```
char Einuse[] =  "file in use";
```
Uses `Einuse` 290h.

⟨*global* `Edeleted` 291a⟩≡ (318)
```
char Edeleted[] = "window deleted";
```
Uses `Edeleted` 291a.

⟨*global* `Ebadreq` 291b⟩≡ (318)
```
char Ebadreq[] = "bad graphics request";
```
Uses `Ebadreq` 291b.

⟨*global* `Etooshort` 291c⟩≡ (318)
```
char Etooshort[] = "buffer too small";
```
Uses `Etooshort` 291c.

⟨*global* `Ebadtile` 291d⟩≡ (318)
```
char Ebadtile[] = "unknown tile";
```
Uses `Ebadtile` 291d.

⟨*global* `Eshort` 291e⟩≡ (318)
```
char Eshort[] =  "short i/o request";
```
Uses `Eshort` 291e.

⟨*global* `Elong` 291f⟩≡ (318)
```
char Elong[] =   "snarf buffer too long";
```
Uses `Elong` 291f.

⟨*global* `Eunkid` 291g⟩≡ (318)
```
char Eunkid[] =  "unknown id in attach";
```
Uses `Eunkid` 291g.

⟨*global* `Ebadrect` 291h⟩≡ (318)
```
char Ebadrect[] =  "bad rectangle in attach";
```
Uses `Ebadrect` 291h.

⟨*global* `Ewindow` 291i⟩≡ (318)
```
char Ewindow[] =  "cannot make window";
```
Uses `Ewindow` 291i.

⟨*global* `Enowindow` 291j⟩≡ (318)
```
char Enowindow[] =  "window has no image";
```
Uses `Enowindow` 291j.

⟨*global* `Ebadmouse` 291k⟩≡ (318)
```
char Ebadmouse[] =  "bad format on /dev/mouse";
```
Uses `Ebadmouse` 291k.

⟨*global* `Ebadwrect` 291l⟩≡ (318)
```
char Ebadwrect[] =  "rectangle outside screen";
```
Uses `Ebadwrect` 291l.

⟨*global* `Ebadoffset` 291m⟩≡ (318)
```
char Ebadoffset[] =  "window read not on scan line boundary";
```
Uses `Ebadoffset` 291m.

B.2 error(), derror()

`error()` has a two-phase behavior controlled by `errorshouldabort`. Before graphics is fully initialized, `errorshoulda` 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) 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.

`derror()`[292d] 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.

⟨*global* `errorshouldabort` 292a⟩≡ (321a)
```
bool errorshouldabort = false;
```
Uses `errorshouldabort` 292a.

⟨`main()` *error management after everything setup* 292b⟩≡ (66a)
```
errorshouldabort = true;/* suicide if there's trouble after this */
⟨main() if initstr or kdbin 273a⟩
threadnotify(shutdown, true);
```
Uses `errorshouldabort` 292a and `shutdown()` 278e.

⟨*function* `error` 292c⟩≡ (321a)
```
void
error(char *s)
{
    fprint(STDERR, "rio: %s: %r\n", s);
    if(errorshouldabort)
        abort();
    threadexitsall("error");
}
```
Uses `errorshouldabort` 292a.

⟨*function* `derror` 292d⟩≡ (321a)
```
void
derror(Display*, char *errorstr)
{
    error(errorstr);
}
```
Uses `error()` 292c.

Appendix C

Utilities

The utility functions here fall into three groups. The first is abort-on-failure wrappers around `malloc`, `realloc`, and `strdup`: `emalloc()`[293d], `erealloc()`[293c], and `estrdup()`[294a] call `error()`[292c] 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()`[294c], `runemove()`[294e], `strrune()`[295a], `cvttorunes()`[294f]): `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`[293a]/`max`[293b]/`isalnum`[294b] — 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.

⟨*function* `min` 293a⟩≡ (321b)
```
int
min(int a, int b)
{
    if(a < b)
        return a;
    return b;
}
```

⟨*function max (windows/rio/util.c)* 293b⟩≡ (321b)
```
int
max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
```

⟨*function* `erealloc` 293c⟩≡ (321b)
```
void*
erealloc(void *p, uint n)
{
    p = realloc(p, n);
    if(p == nil)
        error("realloc failed");
    return p;
}
```
Uses `error()` 292c.

⟨*function* `emalloc` 293d⟩≡ (321b)
```
void*
emalloc(uint n)
{
```

```
    void *p;

    p = malloc(n);
    if(p == nil)
        error("malloc failed");
    memset(p, 0, n);
    return p;
}
```
Uses `error()` 292c.

⟨*function* `estrdup` 294a⟩≡ (321b)
```
char*
estrdup(char *s)
{
    char *p;

    p = malloc(strlen(s)+1);
    if(p == nil)
        error("strdup failed");
    strcpy(p, s);
    return p;
}
```
Uses `error()` 292c.

⟨*function* `isalnum` 294b⟩≡ (321b)
```
//@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;
}
```

⟨*function* `runemalloc` 294c⟩≡ (301)
```
#define runemalloc(n)  malloc((n)*sizeof(Rune))
```

⟨*function* `runerealloc` 294d⟩≡ (301)
```
#define runerealloc(a, n) realloc(a, (n)*sizeof(Rune))
```

⟨*function* `runemove` 294e⟩≡ (301)
```
#define runemove(a, b, n) memmove(a, b, (n)*sizeof(Rune))
```

⟨*function* `cvttorunes` 294f⟩≡ (321b)
```
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` 295a⟩≡ (321b)

```
    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` 295b⟩≡ (321b)

```
    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;
    }
```

Uses `emalloc()` 293d.

Appendix D

Examples of Windowing System Applications TODO

Appendix E

Extra Code

E.1 include/

E.1.1 include/complete.h

```
⟨include/complete.h 297a⟩≡
  #pragma lib "libcomplete.a"
  #pragma src "/sys/src/libcomplete"

  typedef struct Completion Completion;

  ⟨struct Completion 263a⟩

  Completion*  complete(char *dir, char *s);
  void   freecompletion(Completion*);
```

E.1.2 include/frame.h

```
⟨include/frame.h 297b⟩≡
  #pragma src "/sys/src/libframe"
  #pragma lib "libframe.a"

  typedef struct Frbox Frbox;
  typedef struct Frame Frame;

  ⟨enum _anon_ (include/frame.h) 163f⟩

  ⟨constant FRTICKW 165d⟩

  ⟨struct Frbox 167a⟩

  ⟨struct Frame 162a⟩

  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 167b⟩
⟨function NBYTE 183b⟩
```

E.1.3 include/plumb.h

⟨include/plumb.h 298⟩≡

```
#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 252a⟩
```

```
⟨struct Plumbattr 252b⟩

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*);
```

E.2 windows/rio/

E.2.1 windows/rio/dat.h

⟨*enum* _anon_ (windows/rio/dat.h)*3* 299a⟩≡ (299b)

```
enum
{
    ⟨constant Selborder 73d⟩
    ⟨constant Unselborder 107b⟩
    ⟨constants Scrollxxx 160c⟩
    ⟨constant BIG 193a⟩
};
```

⟨windows/rio/dat.h 299b⟩≡

```
// 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_ (windows/rio/dat.h)2 219a⟩

⟨constant STACK 68d⟩

⟨enum _anon_ (windows/rio/dat.h)3 299a⟩

⟨constant DEBUG 289a⟩

⟨enum wctlmesgkind 96a⟩

⟨struct Wctlmesg 96b⟩

⟨struct Conswritemesg 143c⟩

⟨struct Consreadmesg 142b⟩

⟨struct Mousereadmesg 139c⟩

⟨struct Stringpair 142c⟩

⟨struct Mousestate 151d⟩

⟨struct Mouseinfo 151a⟩

⟨struct Window 59⟩

⟨struct Dirtab 123a⟩

⟨struct Fid 63c⟩

⟨struct Xfid 64c⟩

⟨constant Nhash 63a⟩

⟨struct Filsys 62g⟩

⟨struct Timer 279e⟩

//----------------------------------------------------------------------------
// 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 142b, Conswritemesg 143c, Dirtab 123a, Fid 63c, Filsys 62g, Mouseinfo 151a, Mousereadmesg 139c, Mousestate 151d, Stringpair 142c, Timer 279e, Wctlmesg 96b, Window 59, and Xfid 64c.

E.2.2 windows/rio/fns.h

⟨windows/rio/fns.h 301⟩≡

```
// 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 294c⟩
⟨function runerealloc 294d⟩
⟨function runemove 294e⟩
```

```
// error.c
void    derror(Display*, char *); // for main.c
void error(char*);
```

E.2.3 windows/rio/globals.c

```
⟨windows/rio/globals.c 304⟩≡
  #include <u.h>
  #include <libc.h>

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

  #include <frame.h>

  #include <fcall.h>

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

  ⟨global mousectl 57b⟩
  ⟨global mouse 58a⟩
  ⟨global keyboardctl 57c⟩

  ⟨global desktop 58b⟩
  ⟨global background 58c⟩
  ⟨global red 58d⟩

  ⟨global window 61a⟩
  ⟨global wkeyboard 273e⟩
  ⟨global nwindow 61b⟩


  ⟨global exitchan 68b⟩

  ⟨global winclosechan 110a⟩

  ⟨global deletechan 109b⟩


  ⟨global input 61e⟩
  ⟨global all 126c⟩
  ⟨global filsys 62h⟩
  ⟨global hidden 120a⟩
  ⟨global nhidden 120b⟩
  ⟨global scrolling 272a⟩


  ⟨global startdir 100c⟩
  ⟨global sweeping 94a⟩
  ⟨global wctlfd 236a⟩
  ⟨global menuing 104c⟩
  ⟨global snarfversion 250f⟩

  ⟨global maxtab 211e⟩
```

E.2.4 windows/rio/rio.c

```
⟨windows/rio/rio.c 305a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <window.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <plumb.h>

  #include "dat.h"
  #include "fns.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 276a⟩

  ⟨global kbdargv 273f⟩


  ⟨function usage 27⟩


  ⟨function initcmd 273b⟩

  ⟨global oknotes 278d⟩

  ⟨function killprocs 279a⟩


  ⟨function shutdown 278e⟩


  ⟨function threadmain 66a⟩
```

E.2.5 windows/rio/thread_mouse.c

```
⟨windows/rio/thread_mouse.c 305b⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
```

```
#include <fcall.h>
#include <thread.h>

#include <window.h>

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

⟨enum Mxxx 72a⟩

⟨function keyboardhide 275d⟩

⟨global viewr 57a⟩

⟨function resized 245⟩


⟨function mousethread 72c⟩
```

E.2.6 windows/rio/thread_keyboard.c

⟨windows/rio/thread_keyboard.c 306a⟩≡

```
#include <u.h>
#include <libc.h>

// for dat.h
#include <draw.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

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

⟨function keyboardthread 71⟩
```

E.2.7 windows/rio/threads_window.c

⟨windows/rio/threads_window.c 306b⟩≡

```

#include <u.h>
#include <libc.h>

// for dat.h
#include <draw.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

#include <window.h>

#include "dat.h"
```

```
#include "fns.h"

⟨enum Wxxx 77b⟩

⟨function deletetimeoutproc 109a⟩

⟨function wctlmesg 96d⟩

⟨function winctl 78⟩
```

E.2.8 windows/rio/threads_worker.c

```
⟨windows/rio/threads_worker.c 307a⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <thread.h>

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

  ⟨global xfidfree 84b⟩
  ⟨global xfid 84a⟩
  ⟨global cxfidalloc 70a⟩
  ⟨global cxfidfree 70b⟩

  ⟨enum Xxxx 83g⟩

  ⟨function xfidctl 85c⟩

  ⟨function xfidflush 283a⟩

  ⟨function xfidallocthread 84d⟩

  ⟨function xfidinit 70c⟩
```

E.2.9 windows/rio/threads_misc.c

```
⟨windows/rio/threads_misc.c 307b⟩≡
  #include <u.h>
  #include <libc.h>

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

```
#include <window.h>

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

⟨function winclosethread 110d⟩

⟨function deletethread 109e⟩
```

E.2.10 windows/rio/wm.c

```
⟨windows/rio/wm.c 308⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <thread.h>

  #include <window.h>

  #include "dat.h"
  #include "fns.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_ (windows/rio/rio.c) 94d⟩

  ⟨global menu3str 94c⟩

  ⟨global menu3 94b⟩

  //----------------------------------------------------------------------------
  // Helpers
  //----------------------------------------------------------------------------

  ⟨function goodrect 239a⟩


  ⟨function portion 91b⟩

  ⟨function whichcorner 91a⟩

  ⟨function cornercursor 90c⟩

  //----------------------------------------------------------------------------
  // Mouse actions
```

```
//----------------------------------------------------------------------------

⟨function pointto 112a⟩


⟨function onscreen 105a⟩

⟨function sweep 104d⟩



⟨function drawedge 115a⟩

⟨function drawborder 114b⟩

⟨function drag 113b⟩



⟨function cornerpt 118d⟩

⟨function whichrect 118c⟩

⟨function bandsize 117a⟩

//----------------------------------------------------------------------------
// Window management
//----------------------------------------------------------------------------

⟨function delete 108a⟩

⟨function resize 115c⟩

⟨function move 113a⟩

⟨function whide 120c⟩

⟨function wunhide 121e⟩

⟨function hide 119b⟩

⟨function unhide 121c⟩


⟨global rcargv 100f⟩

⟨function new 97b⟩

//----------------------------------------------------------------------------
// Entry point
//----------------------------------------------------------------------------

⟨function button3menu 93b⟩
```

E.2.11 windows/rio/data.c

```
⟨windows/rio/data.c 309⟩≡
  #include <u.h>
  #include <libc.h>
```

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

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

⟨global crosscursor (windows/rio/data.c) 86⟩

⟨global boxcursor (windows/rio/data.c) 87a⟩

⟨global sightcursor (windows/rio/data.c) 87b⟩

⟨global whitearrow (windows/rio/data.c) 87c⟩

⟨global query (windows/rio/data.c) 87d⟩

⟨global tl 88b⟩

⟨global t 88c⟩

⟨global tr 88d⟩

⟨global r 88e⟩

⟨global br 89a⟩

⟨global b 89b⟩

⟨global bl 89c⟩

⟨global l 89d⟩

⟨global corners (windows/rio/data.c) 88a⟩

⟨function iconinit 67c⟩
```

E.2.12 windows/rio/cursor.c

```
⟨windows/rio/cursor.c 310⟩≡
#include <u.h>
#include <libc.h>

// for dat.h
#include <draw.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

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

```
⟨global lastcursor 90a⟩

⟨function riosetcursor 90b⟩
```

E.2.13 windows/rio/wind.c

```
⟨windows/rio/wind.c 311⟩≡
  #include <u.h>
  #include <libc.h>

  #include <draw.h>
  #include <window.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>

  #include <frame.h>
  #include <fcall.h>

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

  ⟨global topped 61c⟩
  ⟨global id 60b⟩

  ⟨global cols 164a⟩
  ⟨global darkgrey 164f⟩


  ⟨global titlecol 99d⟩
  ⟨global lighttitlecol 99e⟩
  ⟨global holdcol 277c⟩
  ⟨global lightholdcol 277e⟩
  ⟨global paleholdcol 277d⟩


  ⟨function wsendctlmesg 96c⟩


  ⟨function wborder 99c⟩

  ⟨function wsetcols 164e⟩


  ⟨function wmk 98e⟩

  ⟨function wsetname 104a⟩

  ⟨function wresize 116b⟩

  ⟨function wrefresh 286e⟩

  ⟨function wclose 111a⟩




  ⟨function wrepaint 107a⟩
```

```
⟨function winborder 90d⟩


⟨function wmovemouse 118b⟩


⟨function wpointto 76a⟩

⟨function wcurrent 106e⟩

⟨function wsetcursor 91d⟩


⟨function wtop 106d⟩

⟨function wtopme 244a⟩

⟨function wbottomme 244b⟩

⟨function wlookid 127a⟩

⟨function wclosewin 108f⟩

⟨function wsetpid 102c⟩
```

E.2.14 windows/rio/processes_winshell.c

```
⟨windows/rio/processes_winshell.c 312a⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <thread.h>

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

  ⟨function winshell 101b⟩
```

E.2.15 windows/rio/terminal.c

```
⟨windows/rio/terminal.c 312b⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
```

```
#include <frame.h>
#include <fcall.h>
#include <thread.h>

#include <plumb.h>
#include <complete.h>

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

⟨enum _anon_ (windows/rio/wind.c) 176b⟩

⟨enum _anon_ (windows/rio/rio.c)2 215e⟩

⟨global menu2str 215d⟩

⟨global menu2 215c⟩

⟨global clickwin 268b⟩
⟨global clickmsec 268c⟩
⟨global selectwin 268d⟩
⟨global selectq 268e⟩

//----------------------------------------------------------------------------
// Completion
//----------------------------------------------------------------------------

⟨function windfilewidth 264b⟩

⟨function showcandidates 264a⟩

⟨function namecomplete 263b⟩

//----------------------------------------------------------------------------
// Editor
//----------------------------------------------------------------------------

⟨function wbswidth 207d⟩

⟨function wfill 198b⟩


⟨function wdelete 209c⟩

⟨function wbacknl 196b⟩

⟨function wsetorigin 197a⟩


⟨function wshow 190⟩


⟨function wsetselect 199⟩

⟨function winsert 175b⟩


⟨function wcontents 222d⟩

//----------------------------------------------------------------------------
```

```
// Cut/copy/paste
//----------------------------------------------------------------------------

⟨function wsnarf 247b⟩

⟨function wcut 248a⟩

⟨function wpaste 248b⟩

//----------------------------------------------------------------------------
// Scrolling
//----------------------------------------------------------------------------

⟨function wframescroll 220b⟩

⟨function framescroll 220a⟩

//----------------------------------------------------------------------------
// Selection
//----------------------------------------------------------------------------

⟨function wclickmatch 271b⟩


⟨global left1 270a⟩
⟨global right1 270b⟩
⟨global left2 270c⟩
⟨global left3 270d⟩

⟨global left 270e⟩
⟨global right 270f⟩

⟨function wdoubleclick 271a⟩


⟨function wselect 269⟩

//----------------------------------------------------------------------------
// Clicking
//----------------------------------------------------------------------------

//----------------------------------------------------------------------------
// Plumb
//----------------------------------------------------------------------------

⟨function wplumb 251b⟩

//----------------------------------------------------------------------------
// Middle click
//----------------------------------------------------------------------------

⟨function button2menu 215a⟩

//----------------------------------------------------------------------------
// Mouse dispatch
//----------------------------------------------------------------------------

⟨function wmousectl 216b⟩

//----------------------------------------------------------------------------
```

```
// Key dispatch
//----------------------------------------------------------------------------

⟨function interruptproc 210d⟩

⟨function wkeyctl 79e⟩
```

E.2.16 windows/rio/snarf.c

```
⟨windows/rio/snarf.c 315a⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <thread.h>

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

  ⟨global snarffd 250b⟩

  ⟨global snarf 250e⟩
  ⟨global nsnarf 250d⟩


  ⟨function putsnarf 250g⟩

  ⟨function getsnarf 250h⟩
```

E.2.17 windows/rio/graphical_window.c

```
⟨windows/rio/graphical_window.c 315b⟩≡
  #include <u.h>
  #include <libc.h>

  // for dat.h
  #include <draw.h>
  #include <mouse.h>
  #include <cursor.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <thread.h>

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

  ⟨function waddraw 153c⟩
```

E.2.18 windows/rio/9p.c

⟨windows/rio/9p.c 316a⟩≡
```
#include <u.h>
#include <libc.h>

// for dat.h
#include <draw.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

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

⟨global messagesize 81b⟩

⟨function filsysrespond 124a⟩

⟨function filsyscancel 283b⟩
```

E.2.19 windows/rio/proc_fileserver.c

⟨windows/rio/proc_fileserver.c 316b⟩≡
```
#include <u.h>
#include <libc.h>

// for dat.h
#include <draw.h>
#include <mouse.h>
#include <cursor.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <thread.h>

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

extern Xfid*  (*fcall[])(Filsys*, Xfid*, Fid*);
extern char Ebadfcall[];
extern bool firstmessage;
extern Fid*  newfid(Filsys*, int);
extern fdt clockfd;

⟨global srvpipe (windows/rio/fsys.c) 234a⟩
⟨global srvwctl (windows/rio/fsys.c) 236b⟩

⟨function filsysproc 81c⟩

⟨function cexecpipe 69b⟩

⟨function post 234c⟩


⟨function filsysinit 69a⟩
```

⟨*function* filsysmount 102e⟩

E.2.20 windows/rio/fsys.c

```
⟨windows/rio/fsys.c 317⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>

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

  ⟨global Eexist 290b⟩
  ⟨global Enotdir 290c⟩
  ⟨global Ebadfcall 290d⟩
  ⟨global Eoffset 290e⟩


  ⟨global dirtab 123b⟩

  static uint  getclock(void);
  Fid*  newfid(Filsys*, int);
  static int  dostat(Filsys*, int, Dirtab*, uchar*, int, uint);

  ⟨global clockfd 279b⟩
  ⟨global firstmessage 83c⟩

  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 65⟩


  ⟨function filsysversion 83b⟩

  ⟨function filsysauth 285f⟩

  ⟨function filsysflush 282e⟩

  ⟨function filsysattach 125⟩
```

⟨*function* numeric 226e⟩

⟨*function* filsyswalk 127d⟩

⟨*function* filsysopen 130c⟩

⟨*function* filsyscreate 136c⟩

⟨*function* idcmp 226b⟩

⟨*function* filsysread 133a⟩

⟨*function* filsyswrite 134b⟩

⟨*function* filsysclunk 132a⟩

⟨*function* filsysremove 137a⟩

⟨*function* filsysstat 135b⟩

⟨*function* filsyswstat 137b⟩

⟨*function* newfid 63f⟩

⟨*function* getclock 279d⟩

⟨*function* dostat 136a⟩

E.2.21 windows/rio/xfid.c

⟨windows/rio/xfid.c 318⟩≡

```
#include <u.h>
#include <libc.h>

#include <draw.h>
#include <marshal.h>
#include <window.h>
#include <thread.h>
#include <cursor.h>
#include <mouse.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>
#include <plumb.h>

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

⟨constant MAXSNARF 249d⟩

⟨global Einuse 290h⟩
⟨global Edeleted 291a⟩
⟨global Ebadreq 291b⟩
⟨global Etooshort 291c⟩
⟨global Ebadtile 291d⟩
⟨global Eshort 291e⟩
⟨global Elong 291f⟩
⟨global Eunkid 291g⟩
```

```
⟨global Ebadrect 291h⟩
⟨global Ewindow 291i⟩
⟨global Enowindow 291j⟩
⟨global Ebadmouse 291k⟩
⟨global Ebadwrect 291l⟩
⟨global Ebadoffset 291m⟩

⟨global tsnarf 249g⟩
⟨global ntsnarf 250a⟩



⟨function xfidattach 126d⟩

⟨function xfidopen 131d⟩

⟨function xfidclose 132b⟩


⟨function keyboardsend 275c⟩


⟨enum _anon_ (windows/rio/xfid.c)2 143d⟩

⟨function xfidwrite 135a⟩

⟨function readwindow 157e⟩

⟨enum _anon_ (windows/rio/xfid.c)3 142d⟩
⟨enum _anon_ (windows/rio/xfid.c)4 139d⟩
⟨enum _anon_ (windows/rio/xfid.c)5 229a⟩

⟨function xfidread 134a⟩
```

E.2.22 windows/rio/scrl.c

```
⟨windows/rio/scrl.c 319⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>

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

  ⟨global scrtmp 192h⟩

  ⟨function scrtemps 193b⟩

  ⟨function freescrtemps 219b⟩

  ⟨function scrpos 192a⟩

  ⟨function wscrdraw 191b⟩
```

⟨*function* wscrsleep 219c⟩

⟨*function* wscroll 218c⟩

E.2.23 windows/rio/time.c

```
⟨windows/rio/time.c 320a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>

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

  ⟨global ctimer 281a⟩
  ⟨global timer 281b⟩

  ⟨function msec 281c⟩

  ⟨function timerstop 281d⟩

  ⟨function timercancel 281e⟩

  ⟨function timerproc 280b⟩

  ⟨function timerinit 280a⟩

  ⟨function timerstart 281f⟩
```

E.2.24 windows/rio/wctl.c

```
⟨windows/rio/wctl.c 320b⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <window.h>
  #include <thread.h>
  #include <cursor.h>
  #include <mouse.h>
  #include <keyboard.h>
  #include <frame.h>
  #include <fcall.h>
  #include <plumb.h>

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

  #include <ctype.h>

  ⟨global Ebadwr 290f⟩
  ⟨global Ewalloc 290g⟩
```

```
⟨enum _anon_ (windows/rio/wctl.c) 237b⟩

⟨global cmds 238a⟩

⟨enum _anon_ (windows/rio/wctl.c)2 238b⟩

⟨global params 238c⟩


⟨function word 239b⟩

⟨function set 240a⟩

⟨function newrect 240b⟩

⟨function shift 240c⟩

⟨function rectonscreen 240d⟩

⟨function riostrtol 241a⟩


⟨function parsewctl 241b⟩

⟨function wctlnew 243⟩

⟨function writewctl 231b⟩

⟨function wctlthread 237a⟩

⟨function wctlproc 236g⟩
```

E.2.25 windows/rio/error.c

```
⟨windows/rio/error.c 321a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>

  ⟨global errorshouldabort 292a⟩

  // could be in 9p.c too
  ⟨global Eperm 290a⟩

  ⟨function error 292c⟩

  ⟨function derror 292d⟩
```

E.2.26 windows/rio/util.c

```
⟨windows/rio/util.c 321b⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <cursor.h>
```

```
#include <mouse.h>
#include <keyboard.h>
#include <frame.h>
#include <fcall.h>

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

⟨function cvttorunes 294f⟩


⟨function erealloc 293c⟩

⟨function emalloc 293d⟩

⟨function estrdup 294a⟩

⟨function isalnum 294b⟩

⟨function strrune 295a⟩

⟨function min 293a⟩

⟨function max (windows/rio/util.c) 293b⟩

⟨function runetobyte 295b⟩
```

E.3 windows/apps/

E.3.1 windows/apps/lens.c

⟨*enum* _anon_ (windows/apps/lens.c) 322a⟩≡ (327)

```
enum {
    Edge = 5,
    Maxmag = 16
};
```

⟨*enum* _anon_ (windows/apps/lens.c)*2* 322b⟩≡ (327)

```
enum {
    Mzoom,
    Munzoom,
    Mgrid,
    Mredraw,
    Mexit
};
```

⟨*global* menustr 322c⟩≡ (327)

```
static char *menustr[] = {
    "zoom",
    "unzoom",
    "grid",
    "redraw",
    "exit",
    nil
};
```

⟨*global menu (windows/apps/lens.c)* 323a⟩≡ (327)
```
static Menu menu = {
    menustr,
    nil,
    -1
};
```

⟨*global* lastp 323b⟩≡ (327)
```
static Point lastp;
```

⟨*global red (windows/apps/lens.c)* 323c⟩≡ (327)
```
static Image *red;
```

⟨*global tmp (windows/apps/lens.c)* 323d⟩≡ (327)
```
static Image *tmp;
```

⟨*global* grid 323e⟩≡ (327)
```
static Image *grid;
```

⟨*global* chequer 323f⟩≡ (327)
```
static Image *chequer;
```

⟨*global* screenfd 323g⟩≡ (327)
```
static int screenfd;
```

⟨*global* mag 323h⟩≡ (327)
```
static int mag = 4;
```

⟨*global* showgrid 323i⟩≡ (327)
```
static int showgrid = 0;
```

⟨*global* screenr 323j⟩≡ (327)
```
static Rectangle screenr;
```

⟨*global* screenbuf 323k⟩≡ (327)
```
static uchar *screenbuf;
```

⟨*function* drawit 323l⟩≡ (327)
```
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);
}
```

⟨*global* bypp 323m⟩≡ (327)
```
static int bypp;
```

⟨*function main (windows/apps/lens.c)* 324⟩≡ (327)

```
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;
	}
}
```

⟨*function* makegrid 326a⟩≡ (327)

```
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);
        }
    }
}
```

⟨*function eresized (windows/apps/lens.c)* 326b⟩≡ (327)

```
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();
}
```

⟨*function* magnify 326c⟩≡ (327)

```
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));
}
```

⟨windows/apps/lens.c 327⟩≡

```
#include <u.h>
#include <libc.h>

#include <draw.h>
#include <window.h>
#include <event.h>
```

⟨*enum* _anon_ (windows/apps/lens.c) 322a⟩

⟨*enum* _anon_ (windows/apps/lens.c)*2* 322b⟩

⟨*global* menustr 322c⟩

⟨*global menu (windows/apps/lens.c)* 323a⟩

⟨*global* lastp 323b⟩
⟨*global red (windows/apps/lens.c)* 323c⟩
⟨*global tmp (windows/apps/lens.c)* 323d⟩
⟨*global* grid 323e⟩
⟨*global* chequer 323f⟩
⟨*global* screenfd 323g⟩
⟨*global* mag 323h⟩

```
⟨global showgrid 323i⟩
⟨global screenr 323j⟩
⟨global screenbuf 323k⟩

void magnify(void);
void makegrid(void);

⟨function drawit 323l⟩

⟨global bypp 323m⟩

⟨function main (windows/apps/lens.c) 324⟩

⟨function makegrid 326a⟩

⟨function eresized (windows/apps/lens.c) 326b⟩

⟨function magnify 326c⟩
```

E.3.2 windows/apps/statusbar.c

⟨*enum* _anon_ (windows/apps/statusbar.c) 328a⟩≡ (334b)

```
enum {PNCTL=3};
```

⟨*global* nokill 328b⟩≡ (334b)

```
static int nokill;
```

⟨*global* textmode 328c⟩≡ (334b)

```
static int textmode;
```

⟨*global* title 328d⟩≡ (334b)

```
static char *title;
```

⟨*global* light 328e⟩≡ (334b)

```
static Image *light;
```

⟨*global* dark 328f⟩≡ (334b)

```
static Image *dark;
```

⟨*global text (windows/apps/statusbar.c)* 328g⟩≡ (334b)

```
static Image *text;
```

⟨*function* initcolor 328h⟩≡ (334b)

```
static void
initcolor(void)
{
    text = display->black;
    light = allocimagemix(display, DPalegreen, DWhite);
    dark = allocimage(display, Rect(0,0,1,1), CMAP8, 1, DDarkgreen);
}
```

⟨*global* rbar 328i⟩≡ (334b)

```
static Rectangle rbar;
```

⟨*global* ptext 328j⟩≡ (334b)

```
static Point ptext;
```

⟨*global* last 328k⟩≡ (334b)

```
static int last;
```

⟨*global lastp (windows/apps/statusbar.c)* 329a⟩≡ (334b)

```
static int lastp = -1;
```

⟨*global* backup 329b⟩≡ (334b)

```
static char backup[80];
```

⟨*function* drawbar 329c⟩≡ (334b)

```
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);
}
```

⟨*function eresized (windows/apps/statusbar.c)* 330a⟩≡ (334b)

```
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();
}
```

⟨*function* bar 330b⟩≡ (334b)

```
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");
}
```

⟨*function usage (windows/apps/statusbar.c)* 331a⟩≡ (334b)

```
static void
usage(void)
{
    fprint(2, "usage: aux/statusbar [-kt] [-w minx,miny,maxx,maxy] 'title'\n");
    exits("usage");
}
```

⟨*function main (windows/apps/statusbar.c)* 331b⟩≡ (334b)

```
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);
}
```

⟨*function* rdenv 332a⟩≡ (334b)

```
/* 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 332b⟩≡ (334b)

```
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;
}
```

⟨*function* screenrect 333⟩≡ (334b)

```
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 334a⟩≡ (334b)

```
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;
}
```

⟨windows/apps/statusbar.c 334b⟩≡

```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <window.h>
#include <bio.h>
#include <event.h>

⟨enum _anon_ (windows/apps/statusbar.c) 328a⟩

static char* rdenv(char*);
int newwin(char*);
Rectangle screenrect(void);

⟨global nokill 328b⟩
⟨global textmode 328c⟩
⟨global title 328d⟩

⟨global light 328e⟩
⟨global dark 328f⟩
⟨global text (windows/apps/statusbar.c) 328g⟩

⟨function initcolor 328h⟩

⟨global rbar 328i⟩
⟨global ptext 328j⟩
```

```
static vlong n, d;
⟨global last 328k⟩
⟨global lastp (windows/apps/statusbar.c) 329a⟩

⟨global backup 329b⟩

⟨function drawbar 329c⟩

⟨function eresized (windows/apps/statusbar.c) 330a⟩

⟨function bar 330b⟩


⟨function usage (windows/apps/statusbar.c) 331a⟩

⟨function main (windows/apps/statusbar.c) 331b⟩


⟨function rdenv 332a⟩

⟨function newwin 332b⟩

⟨function screenrect 333⟩

⟨function postnote 334a⟩
```

E.3.3 windows/apps/winwatch.c

⟨*struct* Win 335a⟩≡ (340)

```
struct Win {
    int n;
    int dirty;
    char *label;
    Rectangle r;
};
```

⟨*global* exclude 335b⟩≡ (340)

```
static Reprog  *exclude  = nil;
```

⟨*global* win 335c⟩≡ (340)

```
static Win *win;
```

⟨*global* nwin 335d⟩≡ (340)

```
static int nwin;
```

⟨*global* mwin 335e⟩≡ (340)

```
static int mwin;
```

⟨*global* onwin 335f⟩≡ (340)

```
static int onwin;
```

⟨*global* lightblue 335g⟩≡ (340)

```
static Image *lightblue;
```

⟨*enum* _anon_ (windows/apps/winwatch.c) 335h⟩≡ (340)

```
enum {
    PAD = 3,
    MARGIN = 5
};
```

⟨*function erealloc (windows/apps/winwatch.c)* 336a⟩≡ (340)

```
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 (windows/apps/winwatch.c)* 336b⟩≡ (340)

```
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 (windows/apps/winwatch.c)* 336c⟩≡ (340)

```
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` 336d⟩≡ (340)

```
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);
}
```

⟨*function* `drawnowin` 337a⟩≡ (340)

```
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);
}
```

⟨*function* `drawwin` 337b⟩≡ (340)

```
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;
}
```

⟨*function* `geometry` 337c⟩≡ (340)

```
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;
}
```

⟨*function redraw (windows/apps/winwatch.c)* 338a⟩≡ (340)

```
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;
}
```

⟨*function eresized (windows/apps/winwatch.c)* 338b⟩≡ (340)

```
void
eresized(int new)
{
    if(new && getwindow(display, Refmesg) < 0)
        fprint(2,"can't reattach to window");
    geometry();
    redraw(view, 1);
}
```

⟨*function* `click` 338c⟩≡ (340)

```
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);
}
```

⟨*function usage (windows/apps/winwatch.c)* 339a⟩≡ (340)

```
static void
usage(void)
{
    fprint(2, "usage: winwatch [-e exclude] [-f font]\n");
    exits("usage");
}
```

⟨*function main (windows/apps/winwatch.c)* 339b⟩≡ (340)

```
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;
        }
    }
}
```

⟨windows/apps/winwatch.c 340⟩≡

```
#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 335a⟩

⟨*global* exclude 335b⟩
⟨*global* win 335c⟩
⟨*global* nwin 335d⟩
⟨*global* mwin 335e⟩
⟨*global* onwin 335f⟩

```
static int rows, cols;
```

⟨*global* lightblue 335g⟩

```
extern Font *font;

⟨enum _anon_ (windows/apps/winwatch.c) 335h⟩

⟨function erealloc (windows/apps/winwatch.c) 336a⟩

⟨function emalloc (windows/apps/winwatch.c) 336b⟩

⟨function estrdup (windows/apps/winwatch.c) 336c⟩


⟨function refreshwin 336d⟩

⟨function drawnowin 337a⟩

⟨function drawwin 337b⟩

⟨function geometry 337c⟩

⟨function redraw (windows/apps/winwatch.c) 338a⟩

⟨function eresized (windows/apps/winwatch.c) 338b⟩

⟨function click 338c⟩

⟨function usage (windows/apps/winwatch.c) 339a⟩

⟨function main (windows/apps/winwatch.c) 339b⟩
```

E.4 windows/libcomplete/

E.4.1 windows/libcomplete/complete.c

```
⟨windows/libcomplete/complete.c 341a⟩≡
  #include <u.h>
  #include <libc.h>
  #include "complete.h"

  ⟨function longestprefixlength 267a⟩

  ⟨function freecompletion 267b⟩

  ⟨function strpcmp 267c⟩

  ⟨function complete 265⟩
```

E.5 windows/libframe/

E.5.1 windows/libframe/frbox.c

```
⟨windows/libframe/frbox.c 341b⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
```

```
#include <mouse.h>
#include <frame.h>

⟨constant SLOP 167c⟩

⟨function _fraddbox 167d⟩

⟨function _frclosebox 168e⟩

⟨function _frdelbox 168c⟩

⟨function _frfreebox 168d⟩

⟨function _frgrowbox 167e⟩

⟨function dupbox 183a⟩

⟨function runeindex 183e⟩

⟨function truncatebox 183c⟩

⟨function chopbox 183d⟩

⟨function _frsplitbox 182d⟩

⟨function _frmergebox 188a⟩

⟨function _frfindbox 182c⟩
```

E.5.2 windows/libframe/frdelete.c

⟨windows/libframe/frdelete.c 342a⟩≡

```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <thread.h>
#include <mouse.h>
#include <frame.h>

⟨function frdelete 188b⟩
```

E.5.3 windows/libframe/frdraw.c

⟨windows/libframe/frdraw.c 342b⟩≡

```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <thread.h>
#include <mouse.h>
#include <frame.h>

⟨function _frdrawtext 174a⟩

⟨function nbytes 195⟩

⟨function frdrawsel 194a⟩

⟨function frdrawsel0 194b⟩
```

⟨*function* `frredraw` 202c⟩

⟨*function* `frtick` 193f⟩

⟨*function* `_frdraw` 185⟩

⟨*function* `_frstrlen` 186c⟩

E.5.4 `windows/libframe/frinit.c`

⟨`windows/libframe/frinit.c` 343a⟩≡
```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <thread.h>
#include <mouse.h>
#include <frame.h>
```

⟨*function* `frinit` 162d⟩

⟨*function* `frinittick` 165e⟩

⟨*function* `frsetrects` 163a⟩

⟨*function* `frclear` 163d⟩

E.5.5 `windows/libframe/frinsert.c`

⟨`windows/libframe/frinsert.c` 343b⟩≡
```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <thread.h>
#include <mouse.h>
#include <frame.h>
```

⟨*constant* `DELTA` 178⟩
⟨*constant* `TMPSIZE` 184a⟩
⟨*global* `frame` 177b⟩

⟨*function* `bxscan` 184b⟩

⟨*function* `chopframe` 187a⟩

⟨*struct* `points_frinsert` 177c⟩

⟨*function* `frinsert` 179⟩

E.5.6 `windows/libframe/frptofchar.c`

⟨`windows/libframe/frptofchar.c` 343c⟩≡
```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <thread.h>
#include <mouse.h>
```

```
#include <frame.h>

⟨function _frptofcharptb 170b⟩

⟨function frptofchar 170a⟩

⟨function _frptofcharnb 172a⟩

⟨function _frgrid 173a⟩

⟨function frcharofpt 172b⟩
```

E.5.7 windows/libframe/frselect.c

```
⟨windows/libframe/frselect.c 344a⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <mouse.h>
  #include <frame.h>

  ⟨function region 202a⟩

  ⟨function frselect 200⟩

  ⟨function frselectpaint 201⟩
```

E.5.8 windows/libframe/frstr.c

```
⟨windows/libframe/frstr.c 344b⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <mouse.h>
  #include <frame.h>

  ⟨constant CHUNK (windows/libframe/frstr.c) 169a⟩
  ⟨function ROUNDUP 169b⟩

  ⟨function _frallocstr 169c⟩

  ⟨function _frinsure 169d⟩
```

E.5.9 windows/libframe/frutil.c

```
⟨windows/libframe/frutil.c 344c⟩≡
  #include <u.h>
  #include <libc.h>
  #include <draw.h>
  #include <thread.h>
  #include <mouse.h>
  #include <frame.h>

  ⟨function _frcanfit 171e⟩
```

⟨*function* _frcklinewrap 171c⟩

⟨*function* _frcklinewrap0 171d⟩

⟨*function* _fradvance 171a⟩

⟨*function* _frnewwid 186a⟩

⟨*function* _frnewwid0 186b⟩

⟨*function* _frclean 187b⟩

E.6 windows/libplumb/

E.6.1 windows/libplumb/event.c

⟨windows/libplumb/event.c 345a⟩≡
```
#include <u.h>
#include <libc.h>
#include <draw.h>
#include <event.h>
#include "plumb.h"

typedef struct EQueue EQueue;

⟨struct EQueue 260c⟩

⟨global equeue 260d⟩
⟨global eqlock 260e⟩

⟨function partial 261a⟩

⟨function addpartial 261b⟩

⟨function plumbevent 262a⟩

⟨function eplumb 262b⟩
```

E.6.2 windows/libplumb/mesg.c

⟨windows/libplumb/mesg.c 345b⟩≡
```
#include <u.h>
#include <libc.h>
#include "plumb.h"

⟨function plumbopen 253a⟩

⟨function Strlen 253b⟩

⟨function Strcpy 254a⟩

⟨function quote 254b⟩

⟨function plumbpackattr 254c⟩

⟨function plumblookup 255a⟩
```

```
⟨function plumbpack 255b⟩

⟨function plumbsend 256a⟩

⟨function plumbline 256b⟩

⟨function plumbfree 256c⟩

⟨function plumbunpackattr 257⟩

⟨function plumbaddattr 258a⟩

⟨function plumbdelattr 258b⟩

⟨function plumbunpackpartial 259⟩

⟨function plumbunpack 260a⟩

⟨function plumbrecv 260b⟩
```

E.6.3 windows/libplumb/plumbsendtext.c

⟨windows/libplumb/plumbsendtext.c 346⟩≡

```
#include <u.h>
#include <libc.h>
#include "plumb.h"

⟨function plumbsendtext 252c⟩
```

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.

all: 126c, 126d, 127c, 226d, 226f
Alloc-79: 83g, 84d
b: 88a, 89b
background: 58c, 67a, 67c, 245
bandsize(): 95d, 117a
bgcolor: 35, 35
BIG: 193a, 193b, 239a
bl: 88a, 89c
Bottom-40: 231b, 237b
boxcursor: 87a, 113b
br: 88a, 89a
button2menu(): 214, 215a
button3menu(): 93a, 93b
Cd-46: 238b
cexecpipe(): 69a, 69b, 236e
clickmsec-26: 268c, 269
clickwin-25: 268b, 269
clockfd: 279b, 279c, 279d
cmds-45: 238a
cols-67: 160a, 161b, 164a, 164c, 221
Consreadmesg: 142b, 299b
Consreadmesg.c1: 142b
Consreadmesg.c2: 142b
Consreadmesg (typedef): 299b
Conswritemesg: 143c, 299b
Conswritemesg.cw: 143c
Conswritemesg (typedef): 299b
ControlMessage: 96a
cornercursor(): 74d, 76b, 76c, 90c, 95d, 104d, 106b, 112a, 112b, 113a, 113b
cornerpt(): 117a, 118d
corners: 88a, 90c
CRdata-7: 142d, 142f
CRflush-8: 142d, 284b, 284c
crosscursor: 86, 104d
ctimer-82: 280a, 280b, 281a, 281f
Current-41: 231b, 237b
Cut-19: 215e, 247a
cvttorunes(): 144b, 249b, 250h, 275c, 294f

CWdata-4: 143d, 144b
CWflush-5: 143d, 284e, 284f
cxfidalloc-77: 70a, 70c, 84d
cxfidfree-78: 70b, 70c, 84d, 85c, 283a
darkgrey-68: 164e, 164f, 164g
DEBUG: 78, 96c, 96d, 97b, 100h, 101b, 102c, 102e, 289a, 289c, 289d
delete(): 107c, 108a
Delete-44: 231b, 237b
Delete-98: 94d, 107c
deletechan: 109a, 109b, 109c, 109e
Deleted: 101a, 108a, 108b, 108d, 231b
deletethread(): 109d, 109e
deletetimeoutproc(): 108d, 109a, 115d
Deltax-47: 238b
Deltay-48: 238b
derror(): 67a, 292d
desktop: 58b, 67a, 67b, 93b, 104d, 113b, 115a, 117a, 121e, 215a, 231b, 235b, 243, 245, 246a, 274a
Dirtab: 123a, 299b
dirtab: 123b, 125, 127d, 129c, 133d, 226d
Dirtab.name: 123a
Dirtab.perm: 123a
Dirtab.qid: 123a
Dirtab.type: 123a
Dirtab (typedef): 299b
dostat(): 133d, 135b, 136a, 226f
drag(): 95d, 113a, 113b
drawborder(): 113b, 114b, 117a
drawedge(): 114b, 115a
Ebadfcall: 82d, 290d, 290d
Ebadmouse: 140b, 291k, 291k
Ebadoffset: 291m, 291m
Ebadrect: 235b, 291h, 291h
Ebadreq: 291b, 291b
Ebadtile: 291d, 291d
Ebadwr: 231b, 243, 290f, 290f
Ebadwrect: 291l, 291l
Edeleted: 131e, 291a, 291a
Eexist: 129e, 290b, 290b
Einuse: 145h, 148, 228a, 290h, 290h
EKey-61: 37a, 37c, 39a
Elong: 249e, 291f, 291f
emalloc(): 63f, 69a, 81c, 84d, 98b, 98e, 100h, 135b, 210c, 225a, 226f, 236g, 243, 251b, 268a, 281f, 293d, 295b
EMouse-60: 37a, 37c, 38e
Enotdir: 130a, 290c, 290c
Enowindow: 157d, 291j, 291j
Eoffset: 290e, 290e
Eperm: 126b, 131c, 136c, 137a, 137b, 275a, 290a, 290a
erealloc(): 97b, 249e, 250h, 293c
EResize-62: 37a, 37c, 39b

error(): 67b, 67d, 68a, 81c, 82a, 85a, 85b, 96d, 98d, 108f, 111b, 124a, 124c, 192g, 193c, 220a, 234c, 236f, 245, 246a, 274a, 283a, 292c, 292d, 293c, 293d, 294a
errorshouldabort: 82a, 292a, 292a, 292b, 292c
Eshort: 140b, 291e, 291e
estrdup(): 97b, 98e, 100a, 100e, 102c, 108d, 115d, 147c, 157d, 224c, 251b, 267f, 268a, 285d, 294a
Etooshort: 229c, 291c, 291c
Eunkid: 127b, 291g, 291g
EventType: 37a
Ewalloc: 231b, 243, 290g, 290g
Ewindow: 235b, 291i, 291i
Exit-100: 94d, 94e
exitchan: 66a, 68b, 68c, 94e
Exited: 81a, 111a, 111d, 111e
fcall: 65, 81c, 82d
Fid: 63c, 299b
Fid.busy: 63e
Fid.dir: 123c
Fid.fid: 63c
Fid.mode: 63c
Fid.next: 63b
Fid.nrpart: 144c
Fid.open: 63c
Fid.qid: 63d
Fid.rpart: 144c
Fid.w: 64a
Fid (typedef): 299b
FILE: 122c, 131d, 132b, 133a, 134a, 135a, 227b
Filsys: 62g, 299b
filsys: 62h, 66a, 66b, 102d
Filsys.cfd: 62g
Filsys.cxfidalloc: 64b
Filsys.fids: 62g
Filsys.sfd: 62g
Filsys.user: 62g
Filsys (typedef): 299b
filsysattach(): 65, 125
filsysauth(): 285e, 285f
filsyscancel(): 283b, 283f, 284a, 284c, 284d, 284f, 284g, 285a, 285b
filsysclunk(): 65, 132a
filsyscreate(): 136b, 136c
filsysflush(): 282d, 282e
filsysinit(): 66a, 69a
filsysmount(): 102d, 102e
filsysopen(): 65, 130c
filsysproc(): 69a, 81c
filsysread(): 65, 133a
filsysremove(): 136b, 137a
filsysrespond(): 82d, 83b, 83d, 83f, 124a, 126b, 126d, 127c, 127d, 129b, 129d, 131c, 131d, 131e, 132a, 132b, 133a, 133b, 134a, 135a, 135b, 136c, 137a, 137b, 139f, 140b, 142f, 144b, 145h, 146b, 146f, 147d, 148, 157d,

222c, 224f, 225a, 225d, 228a, 229c, 231a, 249e, 275a, 283a, 285f
filsysstat(): 65, 135b
filsysversion(): 83a, 83b
filsyswalk(): 65, 127d
filsyswrite(): 65, 134b
filsyswstat(): 136b, 137b
firstmessage: 83c, 83c, 83d, 83e
fontname: 276a, 276b, 276c
framescroll(): 220a, 269
Free-80: 83g, 84d
freescrtemps(): 219b, 245
getclock(): 133a, 135b, 279d
getsnarf(): 247a, 249c, 250h, 269
goodrect(): 117a, 231b, 235b, 239a, 243
hidden: 120a, 120c, 120d, 120e, 121a, 121c, 121e, 230g, 231b, 245
Hidden-101: 94d, 121a, 121c
Hidden-49: 238b
hide(): 119a, 119b
Hide-42: 231b, 237b
Hide-99: 94d, 119a
HiWater-16: 176b, 176c, 176d
holdcol-71: 277a, 277b, 277c, 277f
Holdoff: 277g, 277h, 277i, 278a, 278c
Holdon: 277g, 277h, 278c
iconinit(): 67a, 67c
Id-50: 238b
id-66: 60b, 98e
idcmp(): 226b, 226f
initcmd(): 273a, 273b
input: 61e, 71, 73c, 104d, 106b, 106e, 107a, 108f, 112a, 112b, 113a, 115d, 140b, 164e, 230g, 277b, 286e
interruptproc(): 210c, 210d
isalnum(): 209a, 271a, 294b
kbdargv: 273f, 274a
keyboardctl: 57c, 68a, 71, 275c
keyboardhide(): 274b, 275d
keyboardsend(): 275b, 275c
keyboardthread(): 68e, 71
killprocs(): 66a, 278e, 279a
Kscrollonedown: 206a, 216b, 219a
Kscrolloneup: 206d, 216b, 219a
l: 88a, 89d
lastcursor: 90a, 90b, 251b
left: 270e, 271a
left1-29: 270a, 270e
left2-31: 270c, 270e, 270f
left3-32: 270d, 270e, 270f
lightholdcol-72: 277b, 277e, 277f
lighttitlecol-70: 99c, 99e, 99f
LoWater-17: 176b, 176d

max(): 105a, 176c, 293b
MAXSNARF-1: 249d, 249e
maxtab: 211c, 211d, 211e, 211e, 211g
Maxx-51: 238b
Maxy-52: 238b
menu2: 215a, 215c
menu2str: 215c, 215d, 217a
menu3: 93b, 94b
menu3str: 94b, 94c, 121a
menuing: 91d, 104c, 104d, 112a, 113b
messagesize: 81b, 81b, 81c, 83b, 124a, 124b, 131d, 133a, 135b, 198b, 236g, 251b
min(): 105a, 139f, 176c, 176d, 209e, 240b, 293a
MinWater-18: 176b, 176c
Minx-53: 238b
Miny-54: 238b
MMouse-92: 72a, 72d, 73c
mouse: 58a, 67d, 73c, 74a, 74d, 75b, 76b, 76c, 76d, 91d, 95b, 95d, 104d, 106a, 106b, 106c, 112a, 112b, 113a, 113b, 114a, 117a, 117b, 140b, 217e, 218a, 231b, 274b, 274c, 275d
mousectl: 57b, 67d, 72d, 74c, 77a, 90b, 93b, 98b, 104d, 106b, 112a, 113b, 114a, 117a, 117b, 118b, 215a, 244d, 275d
Mouseinfo: 151a, 299b
Mouseinfo.counter: 151b
Mouseinfo.lastb: 151c
Mouseinfo.lastcounter: 151b
Mouseinfo.qfull: 151a
Mouseinfo.queue: 151a
Mouseinfo.ri: 151a
Mouseinfo.wi: 151a
Mouseinfo (typedef): 299b
mouseloc: 35, 35, 38e
Mousereadmesg: 139c, 299b
Mousereadmesg.cm: 139c
Mousereadmesg (typedef): 299b
Mousestate: 151d, 299b
Mousestate.counter: 151d
Mousestate (typedef): 299b
mousethread(): 68e, 72c
move(): 112c, 113a
Move-35: 231b, 237b
Move-97: 94d, 112c
Moved: 95d, 96a, 113a, 115d
Movemouse: 140b, 141a, 141b
MRdata-10: 139d, 139f
MReshape-93: 244c, 244d, 244e
MRflush-11: 139d, 283e, 283f
msec(): 280b, 281c
N-81: 83g, 84d
NALT-63: 37a, 37b, 38c
NALT-94: 72a, 72b, 72c

namecomplete(): 262c, 263b
NCR-9: 139e, 142d, 142f
NCW-6: 143d, 144a, 144b
new(): 97a, 97b, 235b, 243, 274a
New-33: 231b, 237a, 237b
New-95: 94d, 97a
newfid(): 63f, 81c, 129a
newrect(): 240b
Nhash: 62g, 63a, 63f
nhidden: 120b, 120c, 120d, 120e, 121a, 121e, 230g, 231b, 245
NMR-12: 139d, 139f, 229c
Noscroll-37: 231b, 237b
Noscrolling-58: 238b
nsnarf: 247a, 247b, 248b, 249b, 249c, 250d, 250g, 250h
ntsnarf-3: 248f, 249b, 249e, 250a
numeric(): 226d, 226e
NWALT-91: 77b, 78
NWCR-15: 229a
nwindow: 61b, 76a, 97b, 108f, 127a, 226f, 231b, 245, 279a
oknotes: 278d, 278e
onscreen(): 104d, 105a, 117a
paleholdcol-73: 277a, 277d, 277f
params-59: 238c
Paste-20: 215e, 247a
PID-55: 238b
Plumb-22: 215e, 251a
pointto(): 108a, 112a, 113a, 115c, 119b
portion(): 91a, 91b
post(): 234b, 234c, 236e
putsnarf(): 247b, 250g
Qcons: 141c, 141d, 142f, 144b
Qconsctl: 145e, 145f, 145h, 146a, 146b
Qcursor: 146c, 146d, 146e, 146f, 146g
Qdir: 122d, 123b, 125, 129c, 133d, 226f
QID: 122a, 128a, 136a, 226f
Qkbdin: 274f, 274g, 275a, 275b
Qlabel: 224d, 224e, 224f, 225a
QMAX: 122d
Qmouse: 138a, 138b, 139f, 140b, 148, 149a
Qscreen: 225b, 225c, 225d
Qsnarf: 248c, 248d, 248f, 249b, 249c, 249e, 249f
Qtext: 222a, 222b, 222c
query: 87d, 251b
Qwctl: 227d, 227e, 228a, 228b, 229c, 231a
Qwdir: 267d, 267e, 267f, 268a
Qwindow: 157a, 157b, 157d
Qwinid: 224a, 224b, 224c
Qwinname: 147a, 147b, 147c
Qwsys: 225e, 226a, 226d, 226f, 227b

Qwsysdir: 133d, 226d, 227a, 227b
Qxxx: 122d
r: 88a, 88e, 274a
R-56: 238b
Rawoff: 149c, 149d, 149e, 150a
Rawon: 149c, 149e, 150a
rcargv: 100f, 100h
readwindow(): 157d, 157e
rectonscreen(): 231b, 240d
red: 58d, 67c, 104d, 117a, 231b, 243
redraw(): 35, 35, 38d
Ref (typedef): 299b
Refresh: 149a, 286c, 286d
Reshape-96: 94d, 115b
Reshaped: 95d, 96a, 115c, 115d, 120c, 121e, 231b, 245
resize(): 115b, 115c
Resize-34: 231b, 237b
resized(): 244e, 245
right: 270f, 271a
right1-30: 270b, 270f
RightMenuCommand: 94d
riosetcursor(): 76b, 90b, 90c, 91d, 104d, 112a, 113b, 251b
riostrtol(): 241a
runemalloc: 144b, 197a, 213d, 263b, 275c, 284g, 294c
runemove: 153c, 154a, 155a, 175b, 176d, 197a, 198b, 209c, 213d, 247b, 263b, 294e
runerealloc: 153c, 176c, 247b, 249b, 250h, 294d
runetobyte(): 222d, 249c, 251b, 295b
Scroll-24: 215e, 217a, 217b
Scroll-36: 231b, 237b
Scrollgap: 161b, 161c, 221
scrolling: 97a, 235b, 272a, 272b, 274a
Scrolling-57: 238b
Scrollwid: 160b, 160c, 161b, 221
scrpos(): 191b, 192a
scrtemps(): 191b, 193b
scrtmp-74: 191b, 192h, 193b, 193c, 219b
Selborder: 73c, 73d, 90d, 99b, 99c, 104d, 107a, 116b, 117a, 160a, 160b, 161b, 221, 231b, 235b, 243, 277a, 286e
selectq-28: 220b, 268e, 269
selectwin-27: 220a, 268d, 269
Send-23: 215e, 247a
set(): 240a
Set-38: 231b, 237b
shift(): 240c, 240d
showcandidates(): 263b, 264a
shutdown(): 278e, 292b
sightcursor: 87b, 112a
snarf: 247a, 247b, 248b, 249b, 249c, 250e, 250g, 250h
Snarf-21: 215e, 247a
snarffd: 248e, 250b, 250c, 250g, 250h

snarfversion: 247b, 249e, 249f, 250f
srvpipe: 234a, 234b
srvwctl: 236b, 236e
STACK: 68d, 68e, 70c, 109d, 110c, 273a, 280a
startdir: 100a, 100c, 100e
str: 35, 35, 39a
Stringpair: 142c, 299b
Stringpair.ns: 142c
Stringpair.s: 142c
Stringpair (typedef): 299b
strrune(): 271a, 295a
sweep(): 97a, 104d, 115c
sweeping: 93b, 94a, 95d, 141b, 146g
t-64: 88a, 88c
threadmain(): 35
Timer: 279e, 299b
timer-83: 281b, 281d, 281f
Timer.c: 279e
Timer.cancel: 279e
Timer.dt: 279e
Timer.next: 279e
Timer (typedef): 299b
timercancel(): 219c, 281e
timerinit(): 279f, 280a
timerproc(): 280a, 280b
timerstart(): 219c, 281f
timerstop(): 219c, 280b, 281d
titlecol-69: 99c, 99d, 99f
tl: 88a, 88b
Top-39: 231b, 237b
topped-65: 61c, 98e, 106d, 116b, 244a, 244b
tr: 88a, 88d
tsnarf-2: 248f, 249b, 249e, 249g
unhide(): 121b, 121c
Unhide-43: 231b, 237b
Unselborder: 107a, 107b, 286e
usage(): 27, 272d, 273d, 276b
viewr: 57a, 67a, 245
waddraw(): 153b, 153c, 247a, 248b
Wakeup: 215a, 228a, 228c, 231b, 286a, 286b
wbacknl(): 196a, 196b, 206e, 218c, 220b
wborder(): 99b, 99c, 107a, 116b, 286e
wbottomme(): 231b, 244b
wbswidth(): 207c, 207d, 208a
wclickmatch(): 271a, 271b
wclose(): 101b, 110d, 111a, 132b, 215a, 216b
wclosewin(): 108d, 108f, 111c
wcontents(): 222c, 222d
WCRdata-13: 229a, 229c

WCread-88: 154b, 154f, 154g, 155a, 204d, 276e
WCRflush-14: 229a, 284h, 285a
WCtl-86: 77b, 80g, 81a
wctlfd: 236a, 236e, 236g
Wctlmesg: 96b, 299b
wctlmesg(): 81a, 96d
Wctlmesg.image: 96b
Wctlmesg.r: 96b
Wctlmesg.type: 96b
Wctlmesg (typedef): 299b
wctlnew(): 231b, 237a, 243
wctlproc(): 236f, 236g
wctlthread(): 236f, 237a
wcurrent(): 97b, 106d, 106e, 115d, 231b
wcut(): 246c, 247a, 248a, 248b, 269
WCwrite-89: 212a, 212e, 212f, 213a
wdelete(): 208a, 209c, 213d, 248a
wdoubleclick(): 269, 271a
wfill(): 197a, 198b, 210a, 221
wframescroll(): 220a, 220b
whichcorner(): 90c, 91a, 117a
whichrect(): 117a, 118c
whide(): 119b, 120c, 231b
whitearrow: 87c, 276f
WIN: 122b, 127d, 133d, 135b
winborder(): 76d, 90c, 90d, 95b
winclosechan: 110a, 110b, 110d, 129f, 132a, 226d
winclosethread(): 110c, 110d
winctl(): 78, 97b
windfilewidth(): 263b, 264b
Window: 59, 299b
Window.cctl: 80e
Window.ck: 79a
Window.consread: 141e
Window.conswrite: 143a
Window.ctlopen: 145g
Window.cursor: 91c
Window.cursorp: 91c
Window.deleted: 108c
Window.dir: 100b
Window.frm: 62d
Window.holding: 276d
Window.i: 60c
Window.id: 60a
Window.label: 60e
Window.lastsr: 192d
Window.maxr: 61h
Window.mc: 80a
Window.mouse: 150b

Window.mouseopen: 61f
Window.mouseread: 139a
Window.name: 60a
Window.namecount: 103g
Window.notefd: 102b
Window.nr: 61h
Window.nraw: 153a
Window.org: 62b
Window.pid: 102a
Window.q0: 62a
Window.q1: 62a
Window.qh: 62c
Window.r: 61h
Window.raw: 153a
Window.rawing: 149b
Window.resized: 116a
Window.screenr: 60d
Window.scrolling: 62f
Window.scrollr: 62e
Window.topped: 61d
Window.wctlopen: 227c
Window.wctlread: 229d
Window.wctlready: 227c
Window (typedef): 299b
windows: 61a, 76a, 97b, 108f, 127a, 226f, 231b, 245, 279a
winsert(): 175b, 204c, 213a, 247a, 248b, 262c, 264a
winshell(): 100h, 101b
WKey-84: 77b, 79c, 79d
wkeyboard: 273e, 274a, 274b, 274c, 274d, 274e, 275a, 275d
wkeyctl(): 79d, 79e, 154a, 216b
wlookid(): 126d, 127a, 226d
wmk(): 97b, 98e
WMouse-85: 77b, 80b, 80d
wmousectl(): 216a, 216b
WMouseread-87: 152a, 152e, 152f, 152h
wmovemouse(): 117a, 118b, 141b, 218c
word(): 239b
wpaste(): 247a, 248b, 269
wplumb(): 251a, 251b
wpointto(): 75b, 76a, 91d, 106d, 112a, 140b
wrefresh(): 286d, 286e
wrepaint(): 106e, 107a, 278b, 278c
wresize(): 115d, 116b
writewctl(): 231a, 231b
wscrdraw(): 190, 191a, 191b, 197a, 213a, 221, 247a, 269, 286e
wscroll(): 218b, 218c
wscrsleep(): 218c, 219c, 220b
wselect(): 216b, 269
wsendctlmesg(): 95d, 96c, 101a, 108a, 111a, 113a, 115c, 120c, 121e, 140b, 149a, 149c, 149d, 215a, 228a, 228c,

231b, 245, 277h, 277i, 278a
wsetcols(): 164d, 164e, 221
wsetcursor(): 74d, 90c, 91d, 106e, 107a, 108f, 117a, 146e, 146g
wsetname(): 97b, 104a, 116b
wsetorigin(): 196a, 197a, 206b, 206e, 218c, 220b
wsetpid(): 97b, 102c, 231b
wsetselect(): 197a, 199, 206f, 207a, 207b, 207c, 208a, 213a, 220b, 221, 247a, 248a, 248b, 264a, 269
wshow(): 190, 204c, 205c, 205d, 206f, 207a, 207b, 207c, 210c, 213b, 217b, 231b, 247a, 262c
wsnarf(): 246c, 247a, 247b, 269
wtop(): 106c, 106d
wtopme(): 231b, 244a, 274c
wunhide(): 121c, 121e, 231b
WWread-90: 230c, 230d, 230e, 230g
Xfid: 64c, 299b
xfid-76: 84a, 84d, 283a
Xfid.active: 282c
Xfid.buf: 64c
Xfid.c: 64c
Xfid.f: 64c
Xfid.flushc: 282a
Xfid.flushing: 282a
Xfid.flushtag: 282a
Xfid.free: 84c
Xfid.fs: 64c
Xfid.next: 84c
Xfid.req: 64c
Xfid (typedef): 299b
xfidallocthread(): 70c, 84d
xfidattach(): 125, 126d
xfidclose(): 132a, 132b
xfidctl(): 84d, 85c
xfidflush(): 282e, 283a
xfidfree-75: 84b, 84d
xfidinit(): 66a, 70c
xfidopen(): 130c, 131d
xfidread(): 133a, 134a
xfidwrite(): 134b, 135a
__anon_enum_10: 238b
__anon_enum_11: 83g
__anon_enum_12: 77b
__anon_enum_13: 72a
__anon_enum_1: 143d
__anon_enum_2: 142d
__anon_enum_3: 139d
__anon_enum_4: 229a
__anon_enum_5: 219a
__anon_enum_6: 299a
__anon_enum_7: 176b
__anon_enum_8: 215e

__anon_enum_9: 237b

Bibliography

[FDFH90] James Foley, Andries Van Dam, Steven Feiner, and John Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, 1990. cited page(s) 9, 15

[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) 9, 14

[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. Available at http://www.chilton-computing.org.uk/inf/literature/books/wm/overview.htm. cited page(s) 14

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 15

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 14

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 15

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 11, 14, 23, 30, 32, 33, 36, 40, 45, 50, 58, 61, 288

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 14, 32, 36, 37, 38, 39, 41, 43, 60, 288, 293

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 292

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 10, 11, 14, 20, 23, 25, 26, 30, 32, 33, 34, 35, 36, 38, 39, 41, 52, 53, 55, 58, 60, 103, 156, 225, 234, 288

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Network Stack /net*. 2016. cited page(s) 14, 41, 45, 62

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 32, 44, 288

[Pad26] Yoann Padioleau. *Principia Softwarica: The Plan 9 Widgets libpanel*. 2026. cited page(s) 26, 288

[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, 15

[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) 19, 26

[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) 12, 15, 23

[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, 15

[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, 15, 30, 51

[Pik00a] Rob Pike. Plumbing and other utilities. Technical report, Bell Labs, 2000. Also available at windows/docs/plumb.ps. cited page(s) 39

[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) 10, 15

[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) 11, 14

[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) 11, 14

[SG86] Robert W. Scheifler and Jim Gettys. The X Window system. In *ACM Transaction of Graphics*, pages 79–109, 1986. cited page(s) 11

[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) 17

[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) 9