Principia Softwarica: The Plan 9 Windowing System `rio`

OCaml edition

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike and Yoann Padioleau

March 10, 2026

Contents

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.

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

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

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

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

1.2 The Plan 9 windowing system: `rio`

I will explain in this book the code of the Plan 9 windowing system `rio` [Pik00][1], which contains about 10 000 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 [Pad16b]). In Plan 9, you can run graphical applications with or without `rio`. In fact, `rio` itself is just a graphical application.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. `rio` comes from a series of windowing systems designed by Rob Pike: `8-1/2` [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`). However, an

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

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

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 implemented 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`.

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 [Pad16b], is the most popular open-source implementation of the X Window System [SG86], a windowing system (and a graphics system) designed in the 1980's at MIT. However, its codebase is enormous. In fact, the whole system is divided in hundreds of repositories[6] to better handle its complexity. One of this repository, `xserver`[7], which contains the code of the display server, has already more than 500 000 LOC. This does not even include the code of the window manager (e.g., `twm` with 17 000 LOC), the terminal emulator (e.g., `xterm` with 80 000 LOC), or 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.

[3] For more information, see Section 2.1.6 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

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

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

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.

- 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 (10 000 LOC).

1.4 Getting started

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

[8] https://wayland.freedesktop.org/
[9] https://cgit.freedesktop.org/wayland/weston/
[10] http://www.microwindows.org/

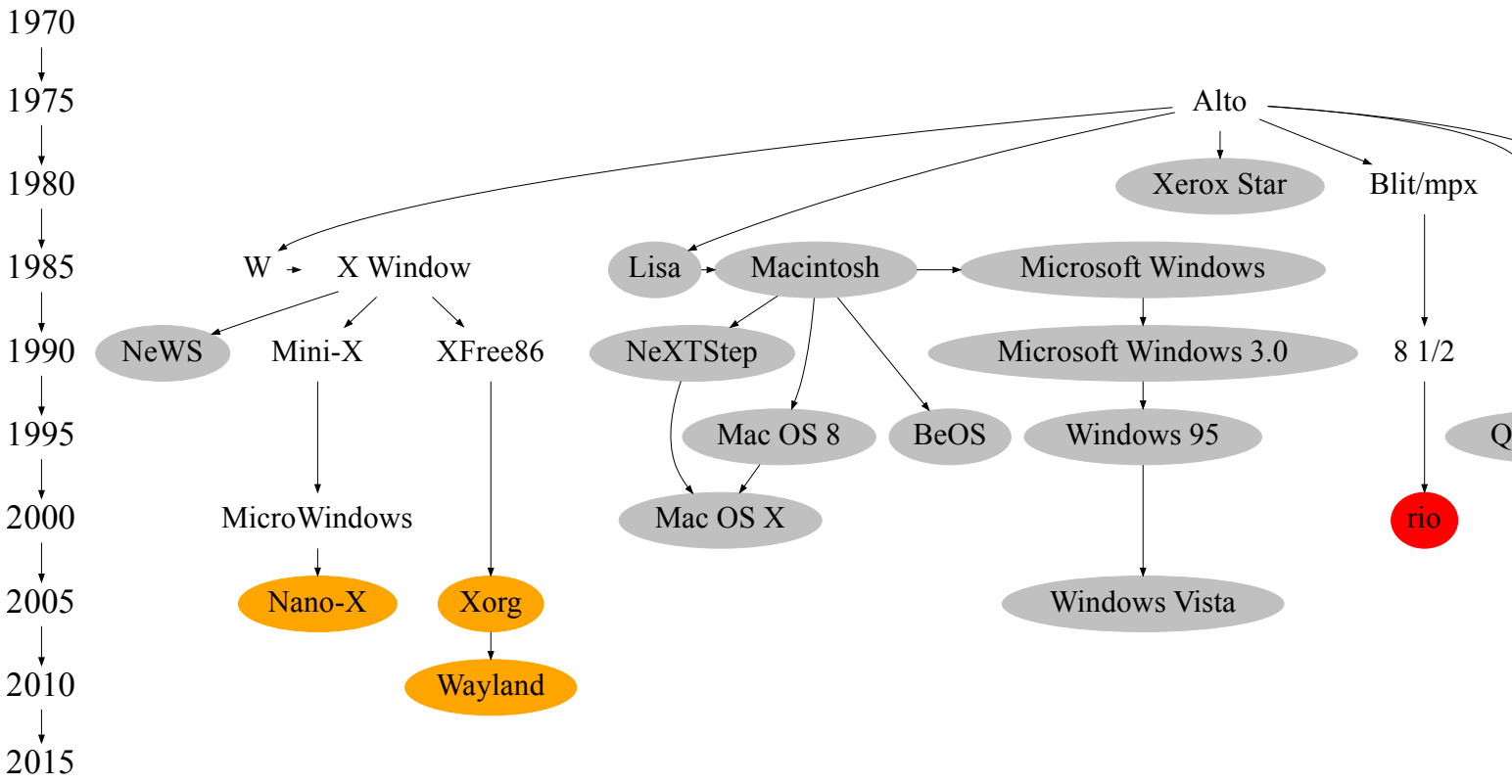

Figure 1.1: Windowing systems timeline

following commands:

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

4   $ 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 3, above, install the graphics system of Plan 9 (`draw`) and configure it to run at the `640x480x8` resolution (See the GRAPHICS book [Pad16b] for more information on `draw`). Line 4 then executes `rio`, which should take over the screen to create the graphics shown in Figure 1.2.

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 [Pad16b] 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 [Pad16c], 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`.

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

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.

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.

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` [Pik00] 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

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/

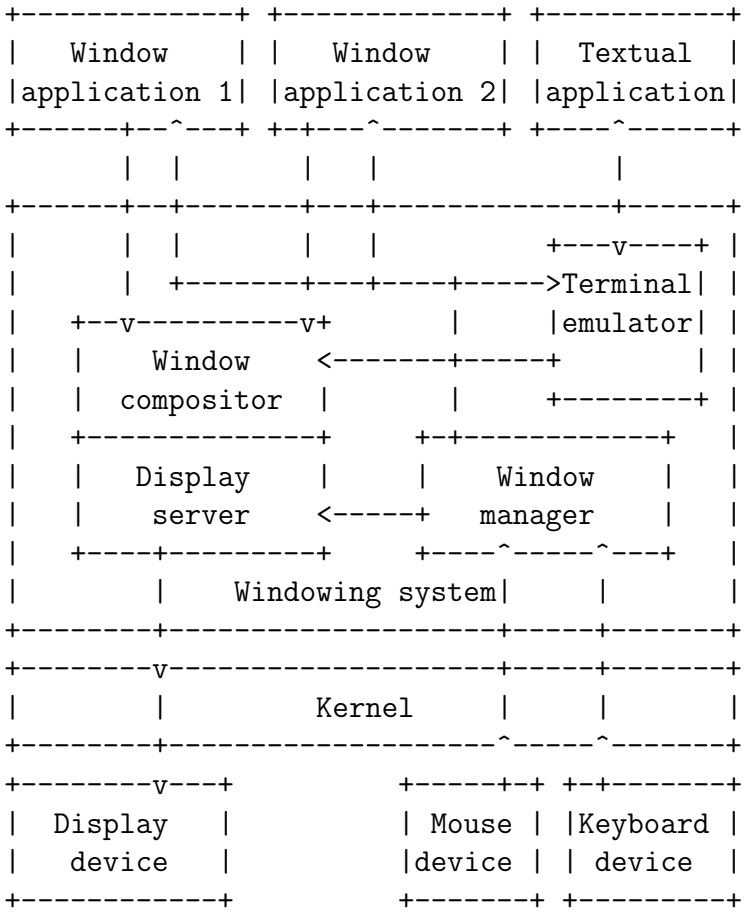

Figure 2.1: Components of a windowing system.

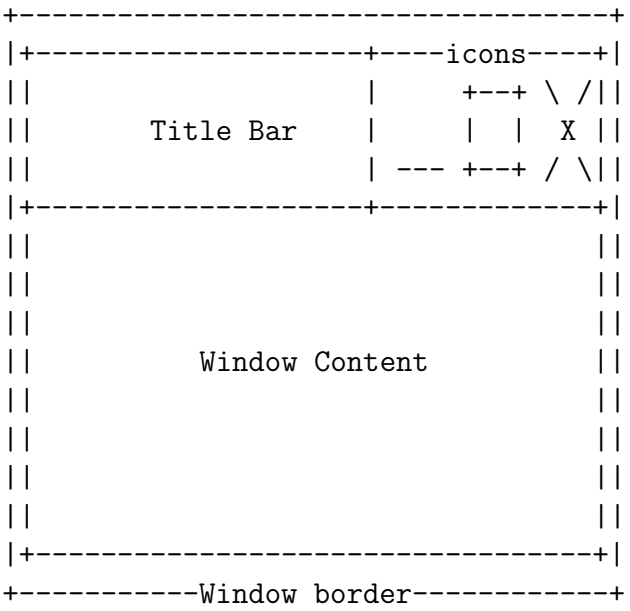

Figure 2.2: Window elements.

- *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.

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 [Pad16b] (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 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,

[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.

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.

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).

2.1.5 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 (eight 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).

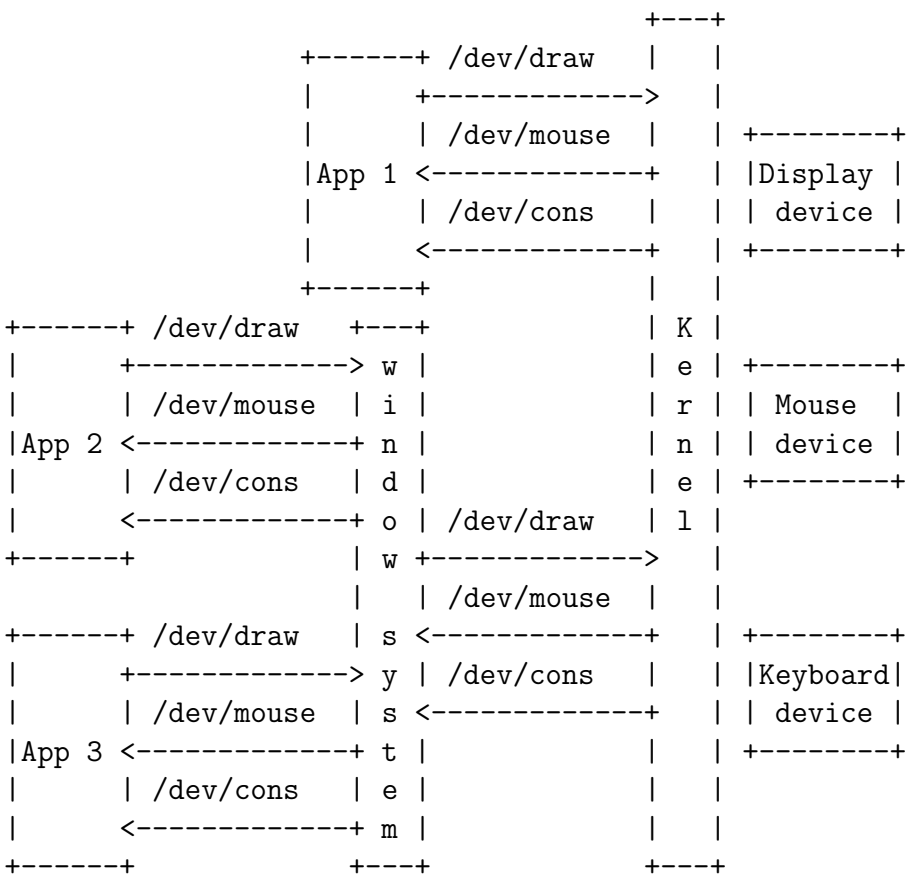

Figure 2.3: Windowing system as a multiplexer.

2.1.6 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 [Pad16b] 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).

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

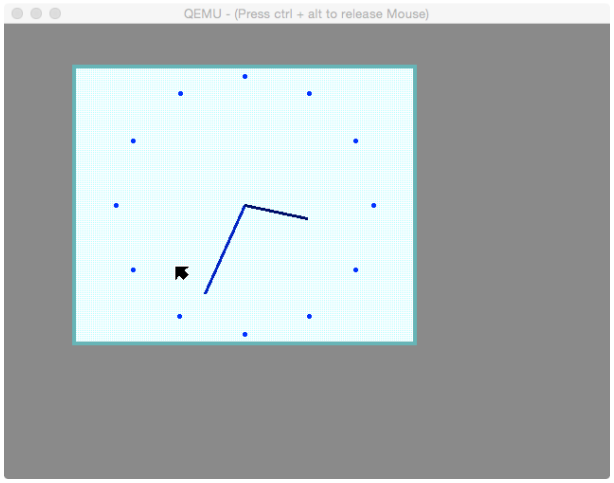

Figure 2.4: `clock` running inside `rio`.

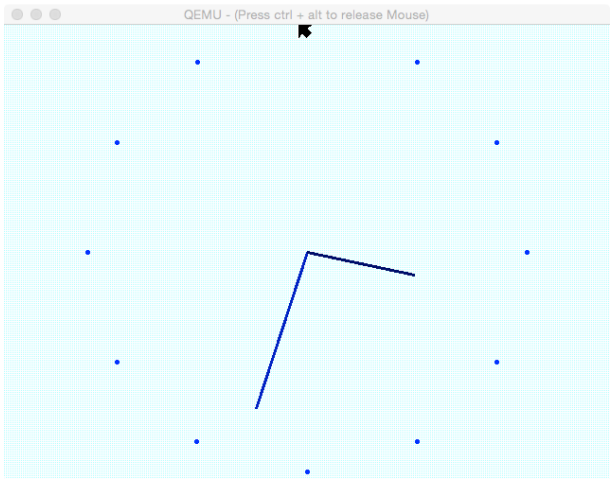

Figure 2.5: `clock` running without `rio`.

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).

2.1.7 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 [Pad16b]). 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 [Pad16b] 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. By using a layer, the programmer of the widget does not have to care about the pixels overlapped by the widget (or the pixels of the widget overlapped by another widget).

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

Resizing windows

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

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

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

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

2.2 `rio` interfaces

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

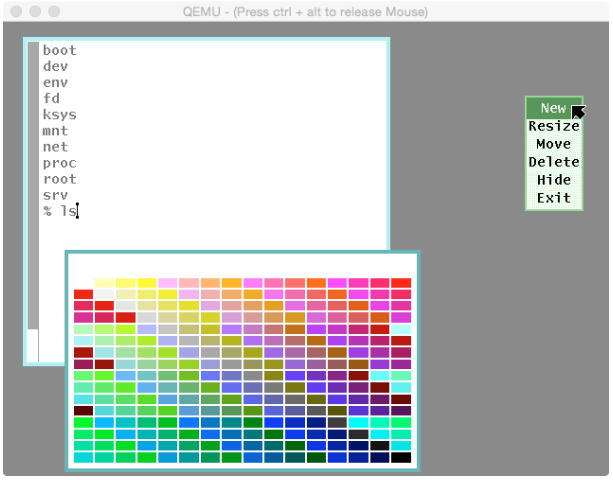

Figure 2.6: Graphical user interface of `rio` after a right-click.

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

2.2.1 Command-line interface

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

⟨*function* `usage` 24⟩≡

```
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 `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

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

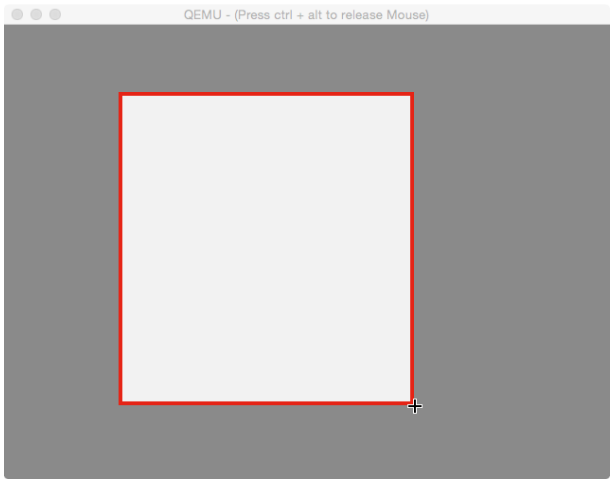

Figure 2.7: Creating a new window.

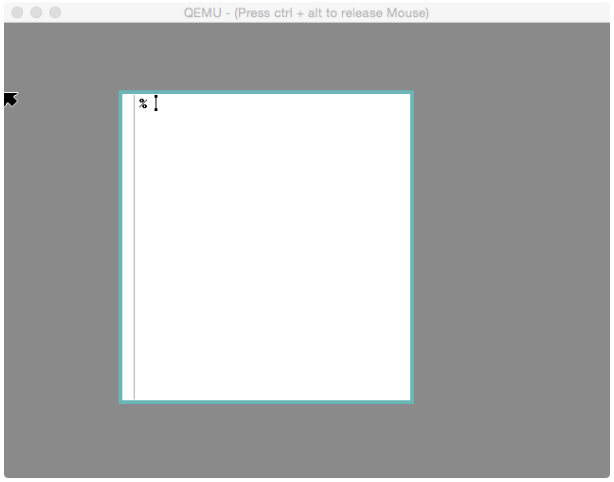

Figure 2.8: Built-in terminal running under `rio`.

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`.

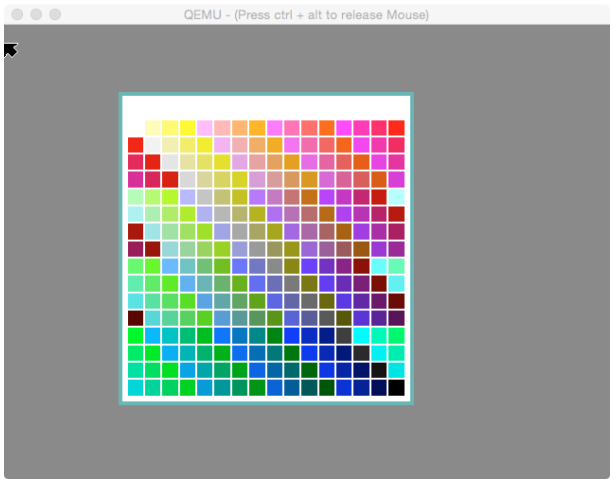

Figure 2.9: `colors` graphical application running under `rio`.

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 2.1.6, `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 2.1.6. 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 [Pad16b] 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 2.1.7 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 [Pad16b]).

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 [Pad16b]) 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 2.1.7, 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 2.1.7, `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 is a function similar to the `gengetwindow()` function I mentioned before. `getwindow()` also reads `/dev/winname`. This file should 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

[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.

hovers a window, the cursor changes according to what the window application wrote in his `/mnt/wsys/cursor` file; if nothing was written, a default cursor is used.

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

The last important information regarding the mouse interface concerns the `mouse.h` header file. A programming alternative to using directly `/dev/mouse` is to use the functions from `mouse.h` such as `initmouse()` (see the GRAPHICS book [Pad16b]). `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, ...)`m 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.

[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.

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

For graphical windows, writing on `/mnt/wsys/cons` should be considered an error. Indeed, the graphical application should instead use the `string()` function from `draw.h` to output text on the screen at a specific location via `/dev/draw` (see the GRAPHICS book [Pad16b]). 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 [Pad16b]). `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.
- `/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 screen at the moment `/mnt/wsys/screen` was read. 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.
- `/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 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.ml

⟨*constant* Hellorio._ 30a⟩≡ (118)
```
let _ =
  Cap.main (fun caps -> thread_main caps)
```

⟨*function* Hellorio.thread_main 30b⟩≡ (118)
```
(* the Keyboard.init() and Mouse.init() below create other threads *)
let thread_main (caps : < Cap.draw; Cap.open_in; Cap.keyboard; Cap.mouse; .. >) =
  let display : Display.t = Draw.init caps "Hello Rio" in
  let view : Display.image = Draw_rio.get_view caps display in

  let kbd : Keyboard.ctl = Keyboard.init caps in
  let mouse : Mouse.ctl = Mouse.init caps in

  let bgcolor : Image.t  =
    Image.alloc display Rectangle.r_1x1 Channel.rgba32 true Color.magenta
  in
  (* if does not use originwindow, then need to add view.r.min *)
  let mousepos = ref (Point.add view.r.Rectangle.min (Point.p 10 10)) in

  redraw display view !mousepos bgcolor;

  while true do
    ⟨Hellorio.thread_main() in event loop 31⟩
    redraw display view !mousepos bgcolor;
  done
```

⟨*function* Hellorio.redraw 30c⟩≡ (118)
```
let redraw (display : Display.t) (view : Display.image) (pos : Point.t)
     (bgcolor : Image.t) : unit =
  Draw.draw view view.r bgcolor None Point.zero;
  (* todo: Text.string *)
  Line.line view pos (Point.add pos (Point.p 100 100))
    Line.EndSquare Line.EndSquare 2 display.black Point.zero;
  Display.flush display
```

⟨*type* Hellorio.event 30d⟩≡ (118)
```
type event =
  | Mouse of Mouse.state
  | Key of Keyboard.key
  (* less: Resize *)
```

⟨Hellorio.thread_main() *in event loop* 31⟩≡ (30b)

```
let ev : event =
  [
    Keyboard.receive kbd |> (fun ev -> Event.wrap ev (fun x -> Key x));
    Mouse.receive mouse |> (fun ev -> Event.wrap ev (fun x -> Mouse x));
  ] |> Event.select
in
(match ev with
| Mouse m ->
  mousepos := m.pos
| Key c ->
  if c = 'q'
  then exit 0
  else Logs.app (fun m -> m "%c" c)
(* less:
 * | Resize -> view := getwindow display
 *)
);
```

2.4 Code organization

2.5 Software architecture

2.5.1 Threads relationships

2.5.2 Trace of a new window creation

2.5.3 Trace of a mouse click

2.5.4 Trace of a key press

2.5.5 Trace of a drawing operation

2.6 Book structure

Chapter 3

Core Data Structures

3.1 Device handles

3.1.1 Output device: `display` and `view`

3.1.2 Input devices: `mouse` and `kbd`

3.2 Desktop

3.3 Windows

3.3.1 `Window.t`

⟨*type* `Window.t` 32⟩≡ (116a)

```
(* The window type! *)
type t = {
  (* ------------------------------------------------------------------ *)
  (* ID *)
  (* ------------------------------------------------------------------ *)
  ⟨Window.t id fields 33b⟩

  (* ------------------------------------------------------------------ *)
  (* Graphics *)
  (* ------------------------------------------------------------------ *)
  ⟨Window.t graphics fields 34a⟩

  (* ------------------------------------------------------------------ *)
  (* Mouse *)
  (* ------------------------------------------------------------------ *)
  ⟨Window.t mouse fields 44a⟩

  (* ------------------------------------------------------------------ *)
  (* Keyboard *)
  (* ------------------------------------------------------------------ *)
  ⟨Window.t keyboard fields 34k⟩

  (* ------------------------------------------------------------------ *)
  (* Commands *)
  (* ------------------------------------------------------------------ *)
  ⟨Window.t command fields 46b⟩

  (* ------------------------------------------------------------------ *)
  (* Process *)
```

```
    (* ------------------------------------------------------------------- *)
    ⟨Window.t process fields 58e⟩

    (* ------------------------------------------------------------------- *)
    (* Config *)
    (* ------------------------------------------------------------------- *)
    ⟨Window.t config fields 86⟩

    (* ------------------------------------------------------------------- *)
    (* Wm *)
    (* ------------------------------------------------------------------- *)
    ⟨Window.t wm fields 34f⟩

    (* ------------------------------------------------------------------- *)
    (* Textual Window *)
    (* ------------------------------------------------------------------- *)
    ⟨Window.t textual window fields 34l⟩

    (* ------------------------------------------------------------------- *)
    (* Graphical Window *)
    (* ------------------------------------------------------------------- *)
    ⟨Window.t graphical window fields 34i⟩

    (* ------------------------------------------------------------------- *)
    (* Concurrency *)
    (* ------------------------------------------------------------------- *)
    (* less:
     * - a Ref (Mutex.t? atomic anyway in ocaml), ref counting
     *   or simply a counter as there is no race issue for rio-ocaml.
     * - Qlock (Condition.t?), needed for?
     *)

    (* ------------------------------------------------------------------- *)
    (* Misc *)
    (* ------------------------------------------------------------------- *)
    ⟨Window.t other fields 46e⟩
  }
```

⟨*type* Window.wid 33a⟩≡ (116a)
```
  (* window id *)
  type wid = int
```

⟨Window.t *id fields* 33b⟩≡ (32) 33d▷
```
  (* visible in /mnt/wsys/winid (and used for /mnt/wsys/<id>/devs) *)
  id: wid;
```

⟨*global* Window.wid_counter 33c⟩≡ (116a)
```
  let wid_counter =
    ref 0
```

⟨Window.t *id fields* 33d⟩+≡ (32) ◁33b 33e▷
```
  (* writable through /mnt/wsys/label *)
  mutable label: string;
```

⟨Window.t *id fields* 33e⟩+≡ (32) ◁33d
```
  (* public named image, visible in /mnt/wsys/winname; change when resize *)
  mutable winname: string;
```

⟨Window.t *graphics fields* 34a⟩≡ (32) 34b▷
```
(* This is most of the time a layer, but it can also be a plain Image.t
 * when the window is hidden.
 * less: option? when delete the window structure and thread is still
 * out there because we wait for the process to terminate?
 *)
mutable img: Image.t;
```

⟨Window.t *graphics fields* 34b⟩+≡ (32) ◁34a 82a▷
```
(* todo: for originwindow and really virtual screen? vs img.r? *)
mutable screenr: Rectangle.t;
```

3.3.2 windows

⟨*global* Globals.windows 34c⟩≡ (99c)
```
let windows: (Window.wid, Window.t) Hashtbl.t = Hashtbl_.create ()
```

⟨*type* Window.topped_counter 34d⟩≡ (116a)
```
type topped_counter = int
```

⟨*global* Window.topped_counter 34e⟩≡ (116a)
```
let topped_counter =
  ref 0
```

⟨Window.t *wm fields* 34f⟩≡ (32)
```
mutable topped: topped_counter;
```

3.3.3 current

⟨*global* Globals.current 34g⟩≡ (99c)
```
(* the man page of rio (rio(1)) uses the term 'current'
 * old: was called 'input' in rio
 *)
let current: Window.t option ref  = ref None
```

⟨*function* Globals.win 34h⟩≡ (99c)
```
(* a bit like cpu(), up() in the kernel, a convenient global *)
let win () =
  !current
```

3.3.4 Graphical windows

⟨Window.t *graphical window fields* 34i⟩≡ (32) 34j▷
```
mutable mouse_opened: bool;
```

⟨Window.t *graphical window fields* 34j⟩+≡ (32) ◁34i 81e▷
```
mutable raw_mode: bool;
```

⟨Window.t *keyboard fields* 34k⟩≡ (32) 42c▷
```
(* see also Window.terminal below for keys when in non-raw (buffered) mode *)
raw_keys: Keyboard.key Queue.t;
```

3.3.5 Textual windows

⟨Window.t *textual window fields* 34l⟩≡ (32)
```
terminal: Terminal.t;
```

⟨*type* Terminal.t 35a⟩≡ (109b 101d)
```
type t = {
  (* the model *)

  ⟨Terminal.t text data fields 35b⟩
  ⟨Terminal.t text cursor fields 35c⟩

  (* the view *)

  ⟨Terminal.t graphics fields 35g⟩

  (* misc *)

  ⟨Terminal.t other fields 36a⟩
}
```

⟨Terminal.t *text data fields* 35b⟩≡ (35a)
```
(* growing array (simpler than a gap buffer).
 * alt: a growing string, like in Efuns.
 *)
mutable text: Rune.t array;
(* number of runes used in text (<= Array.length term.text) *)
mutable nrunes: int;

(* less: lines? like in Efuns? with EOF sentinel to simplify code? *)
```

⟨Terminal.t *text cursor fields* 35c⟩≡ (35a) 35f▷
```
(* where entered text go (and selection start) (old: q0 in rio) *)
mutable cursor: position;
mutable end_selection: position option;      (* old: q1 in rio) *)
```

⟨*type* Terminal.position 35d⟩≡ (109b 101d)
```
type position = {
  i: int;
}
```

⟨*constant* Terminal.zero 35e⟩≡ (109b)
```
let zero =
  { i = 0 }
```

⟨Terminal.t *text cursor fields* 35f⟩+≡ (35a) ◁35c
```
(* Division between characters the host has seen and characters not
 * yet transmitted. The position in the text that separates output from input.
 * old: qh in rio
 *)
mutable output_point: position;
```

⟨Terminal.t *graphics fields* 35g⟩≡ (35a)
```
img: Image.t;

(* img.r without border and some extra space *)
r: Rectangle.t;
(* right side of r, and no bottom rectangle covering a line of
 * text with font below
 *)
textr: Rectangle.t;
(* left side of r *)
scrollr: Rectangle.t;
```

⟨Terminal.t *other fields* 36a⟩≡ (35a) 36b▷
```
  (* first character visible in window from 'text' *)
  mutable origin_visible: position;
  mutable runes_visible: int;
```

⟨Terminal.t *other fields* 36b⟩+≡ (35a) ◁36a 36c▷
```
  font: Font.t;
```

⟨Terminal.t *other fields* 36c⟩+≡ (35a) ◁36b 36d▷
```
  (* this will alter which color to use to render the text *)
  mutable is_selected: bool;
```

⟨Terminal.t *other fields* 36d⟩+≡ (35a) ◁36c
```
  (* alt: mutable colors: colors; *)
```

3.4 Filesystem server

3.4.1 Fileserver.t

⟨*type* Fileserver.t 36e⟩≡ (99)
```
  type t = {
    (* the pipe *)

    (* clients_fd will be shared by all the winshell processes *)
    clients_fd: Unix.file_descr;
    server_fd: Unix.file_descr;

    (* for security *)
    user: string;

    (* the files managed by the server currently-in-use by the client *)
    fids: (File.fid, File.t) Hashtbl.t;

    ⟨Fileserver.t other fields 49a⟩
  }
```

3.4.2 File state and file ids

⟨*type* File.fid 36f⟩≡ (98d)
```
  (* This is maintained by the "client" (the kernel on behalf of a winshell) *)
  type fid = Protocol_9P.fid
```

⟨*type* File.t 36g⟩≡ (98d)
```
  (* fid server-side state (a file) *)
  type t = {
    (* The fid is maintained by the "client" (the kernel on behalf of winshell).
     * It is the key used to access information about a file used by
     * the client (it's redundant in File.t because it is also the key in
     * Fileserver.t.fids)
     *)
    fid: fid;

    (* the state *)
    mutable opened: Plan9.open_flags option;

    ⟨File.t other fields 37a⟩
    (* less: nrpart for runes *)
  }
```

⟨File.t *other fields* 37a⟩≡ (36g) 37f▷
```
  (* for stat (mutable for the same reason than qid) *)
  mutable entry: dir_entry_short;
```

⟨*type* File.dir_entry_short 37b⟩≡ (98d)
```
  (* simpler than Plan9.dir_entry *)
  type dir_entry_short = {
    name: string;
    code: filecode;

    type_: Plan9.qid_type;
    (* just for the user; group and other are 'noperm' *)
    perm: Plan9.perm_property;
  }
```

⟨*type* File.filecode 37c⟩≡ (98d)
```
  (* This is returned by the server (rio) to identify a file of the server *)
  type filecode =
    | Dir of dir
    | File of devid
```

⟨*type* File.dir 37d⟩≡ (98d)
```
    and dir =
      (* '/'
       * old: was called Qdir in rio *)
      | Root
```

⟨*constant* File.root_entry 37e⟩≡ (98d)
```
  let root_entry =
    { name = "."; code = Dir Root; type_ = N.QTDir; perm =  N.rx }
```

⟨File.t *other fields* 37f⟩+≡ (36g) ◁37a 69b▷
```
  (* less: we could also use a wid *)
  win: Window.t;
```

3.5 Devices

⟨*type* File.devid 37g⟩≡ (98d)
```
    and devid =
      | WinName
      | Mouse
      (* todo: Cursor ... *)
      | Cons
      | ConsCtl

      | WinId
      | Text
```

⟨*constant* Threads_fileserver.all_devices 37h⟩≡ (111b)
```
  let all_devices : (File.devid * Device.t) list = [
    File.WinName , Virtual_draw.dev_winname;
    File.Mouse   , Virtual_mouse.dev_mouse;
    File.Cons    , Virtual_cons.dev_cons;
    File.ConsCtl , Virtual_cons.dev_consctl;

    File.WinId   , Dev_wm.dev_winid;
    File.Text    , Dev_textual_window.dev_text;
  ]
```

⟨*constant* Threads_fileserver.toplevel_entries 38a⟩≡ (111b)

```
let toplevel_entries : File.dir_entry_short list =
  all_devices |> List.map (fun (devid, dev) ->
    File.{
      name = dev.D.name;
      code = File.File devid;
      type_ = Plan9.QTFile;
      perm = dev.D.perm;
    }
  )
```

⟨*function* Threads_fileserver.device_of_devid 38b⟩≡ (111b)

```
let device_of_devid (devid : File.devid) : Device.t =
  try
    List.assoc devid all_devices
  with Not_found ->
    raise (Impossible (spf "all_devices is not correctly set; missing code %d"
             (File.int_of_filecode (File.File devid))))
```

⟨*type* Device.t 38c⟩≡ (97a)

```
type t = {
  (* ex: "winname", "cons" *)
  name: string;
  perm: Plan9.perm_property;

  (* called when a process is opening/closing the device *)
  open_: Window.t -> unit;
  close: Window.t -> unit;

  (* called when a process is reading/writing on the device.
   *
   * we need to thread 'read' (and 'write') because this operation may spend
   * some time waiting while receiving and sending information on channels.
   *
   * Note that 'read' takes an offset (and count) because each device
   * can honor or not the offset requirements. For instance,
   * /dev/winname does honor offset but /dev/mouse does not.
   *)
  read_threaded: int64 -> int -> Window.t -> string (* bytes *);
  write_threaded: int64 -> string (* bytes *) -> Window.t -> unit;
}
```

⟨*constant* Device.default 38d⟩≡ (97a)

```
let default = {
  name = "<default>";
  perm = Plan9.rw;
  open_ = (fun _ -> ());
  close = (fun _ -> ());
  read_threaded = (fun _ _ _ -> "");
  write_threaded = (fun _ _ _ -> ());
}
```

Chapter 4

main()

⟨*toplevel* Main._1 39a⟩≡ (100a)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     Exit.exit caps (Exit.catch (fun () -> CLI.main caps (CapSys.argv caps)))
  )
```

⟨*type* CLI.caps 39b⟩≡ (95)
```
(* Need:
 * - draw/mouse/keyboard because rio multiplexes access to those devices
 * - fork/exec/chdir when creating new windows which trigger new rc
 *   processes run possibly from different directories.
 * - open_in: for /dev/winname access
 * - mount/bind: for the window to mount the rio fileserver to /mnt/wsys
 *   and then bind it to /dev for virtual /dev/{cons,mouse,...}
 *)
type caps = <
    Cap.draw; Cap.mouse; Cap.keyboard;
    Cap.fork; Cap.exec; Cap.chdir;
    Cap.open_in;
    Cap.mount; Cap.bind
  >
```

⟨*signature* CLI.main 39c⟩≡ (95a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; Cap.stdout; Cap.stderr; ..> ->
  string array -> Exit.t
```

⟨*constant* CLI.usage 39d⟩≡ (95b)
```
let usage =
  "usage: rio [options]"
```

⟨*function* CLI.main 39e⟩≡ (95b)
```
let main (caps : < caps; Cap.stdout; Cap.stderr; ..>) (argv : string array) :
    Exit.t =
  ⟨CLI.main() locals 92a⟩
  let options = [
    ⟨CLI.main() options elements 89a⟩
  ] |> Arg.align
  in
  (* This may raise ExitCode *)
  Arg_.parse_argv caps argv options (fun _f -> Arg.usage options usage) usage;
  ⟨CLI.main() logging setup 92b⟩
  try
    (* the main call *)
    thread_main caps
```

```
    with exn ->
        ⟨CLI.main() handle exn 93b⟩
```

⟨*signature* CLI.thread_main 40a⟩≡ (95a)
```
  (* main thread which will itself create mouse/keyboard/fs/... threads *)
  val thread_main: < caps; ..> -> Exit.t
```

⟨*function* CLI.thread_main 40b⟩≡ (95b)
```
  let thread_main (caps: < caps; .. >) : Exit.t =

    (* Rio, a graphical application *)
    ⟨CLI.thread_main() graphics initializations 40c⟩

    (* Rio, a filesystem server *)
    let fs = Fileserver.init () in

    (* Rio, a concurrent application *)
    let (exit_chan: Exit.t Event.channel) = Event.new_channel () in
    ⟨CLI.thread_main() threads creation 41b⟩

    (* Wait *)
    let exit_code = Event.receive exit_chan |> Event.sync in
    (* todo: kill all procs? all the winshell processes? *)
    (* todo: kill all threads? done when do exit no? *)
    exit_code
```

4.1 Graphics initialization

⟨CLI.thread_main() *graphics initializations* 40c⟩≡ (40b)
```
  let display : Display.t = Draw.init caps "orio" in
  (* alt: simpler (but does not allow rio under rio in):
   * let view = display.image in
   * less: let viewr save?
   *)
  let view : Display.image = Draw_rio.get_view caps display in
  let font : Font.t = Font_default.load_default_font display in

  ⟨CLI.thread_main() graphics initializations, debug 92i⟩

  let background = Image.alloc_color display (Color.mk2 0x77 0x77 0x77) in
  Globals.red   := Image.alloc_color display (Color.mk2 0xDD 0x00 0x00);
  Globals.title_color       := Image.alloc_color display Color.greygreen;
  Globals.title_color_light := Image.alloc_color display Color.palegreygreen;

  let desktop : Baselayer.t = Baselayer.alloc view background in

  Draw.draw_color view view.r background;
    (* to test: alternative to -test that leverages work done above
    Test.test_display_default_font display view;
    Test.test_display_text display view font;
    *)

  Display.flush display;
```

⟨*constant* Globals.red 40d⟩≡ (99c)
```
  let red               = ref Display.fake_image
```

⟨*constant* Globals.title_color 40e⟩≡ (99c)
```
  let title_color       = ref Display.fake_image
```

⟨*constant* `Globals.title_color_light` 41a⟩≡ (99c)
```
let title_color_light = ref Display.fake_image
```

4.2 Threads creation

⟨`CLI.thread_main()` *threads creation* 41b⟩≡ (40b) 59b▷
```
let mouse : Mouse.ctl = Mouse.init caps in
let kbd : Keyboard.ctl = Keyboard.init caps in

let _kbd_thread   =
  Thread.create Thread_keyboard.thread kbd in

let _mouse_thread =
  Thread.create (Thread_mouse.thread caps)
     (exit_chan,
      mouse, (display, desktop, view, font),
      fs) in

let _fileserver_master_thread =
  Thread.create Threads_fileserver.thread fs in
```

4.3 Filesystem server initialization

⟨*signature* `Fileserver.init` 41c⟩≡ (99a)
```
(* internally creates a pipe between clients_fd/server_fd above *)
val init: unit -> t
```

⟨*function* `Fileserver.init` 41d⟩≡ (99b)
```
let init () =
  let (fd1, fd2) = Unix2.pipe () in
  (* the default threadUnix implementation just set non_block for fd2
   * (the 'in_fd'), but in plan9 pipes are bidirectional so we need
   * to set non_block for fd1 too.
   *)
  Unix.set_nonblock fd1;
  (* todo? record fd2 as close_on exec?
   * Unix.set_close_on_exec? but when it's useful really? just cleaner/safer?
   *)

  { clients_fd = fd1;
    server_fd = fd2;

    (* todo: let user = Common.cat "/dev/user" |> String.concat "" in *)
    user = "pad";
    message_size = 8192 + Protocol_9P.io_header_size;

    fids = Hashtbl_.create ();
  }
```

Chapter 5

Threads

5.1 Keyboard thread

⟨*signature* `Thread_keyboard.thread` 42a⟩≡ (112a)
```
(* Reads from the keyboard and sends the key to the "current" window *)
val thread: Keyboard.ctl -> unit
```

⟨*function* `Thread_keyboard.thread` 42b⟩≡ (112b)
```
let thread (kbd : Keyboard.ctl) : unit =
  (* less: threadsetname *)

  while true do
    let key = Keyboard.receive kbd |> Event.sync in
    (* less:
     *  - do that in other thread? so can start reading more keys?
     *  - have sendp?
     *  - receive array of keys? nbrecv?
     *  - use double array of keys so can send and then receive without
     *    losing anything?
    *)
    Globals.win () |> Option.iter (fun (win : Window.t) ->
      Event.send win.chan_keyboard key |> Event.sync
    )
  done
```

⟨`Window.t` *keyboard fields* 42c⟩+≡ (32) ◁34k 79a▷
```
(* Threads_window.thread <-- Thread_keyboard.thread (<-- keyboard.thread) *)
(* todo: need list of keys? [20]?not reactif enough if buffer one key only? *)
chan_keyboard: Keyboard.key Event.channel;
```

5.2 Mouse thread

⟨*signature* `Thread_mouse.thread` 42d⟩≡ (113a)
```
(* Reads from the mouse and sends the mouse state to the "current" window
 * if the cursor is in the windows rect. Otherwise might call the
 * window manager to change the current window or also display a menu depending
 * on the mouse state. With this menu one can select "New" which will create
 * a new window and process (hence the need for the many Cap.xxx below).
 *)
val thread:
  < Cap.fork; Cap.exec; Cap.chdir; Cap.mount; Cap.bind; .. > ->
  Exit.t Event.channel *
  Mouse.ctl *
```

```
    (Display.t * Baselayer.t * Image.t * Font.t) *
    Fileserver.t ->
    unit
```

⟨*type* `Thread_mouse.event` 43a⟩≡ (113b)
```
  type event =
    | Mouse of Mouse.state
```

⟨*function* `Thread_mouse.thread` 43b⟩≡ (113b)
```
  let thread (caps : < Cap.fork; .. >)
             (exitchan, mouse, (display, desktop, view, font), fs) =
    (* less: threadsetname *)

    while true do
      let ev =
        [
          Mouse.receive mouse |> (fun ev -> Event.wrap ev (fun x -> Mouse x));
        ] |> Event.select
      in
      (match ev with
      | Mouse m ->
        (* less: wkeyboard and button 6 *)
        (* less: loop again *)
        (* less: wkeyboard and ptinrect *)
        (* todo? race on Globals.win? can change between? need store in local?*)

        ⟨Thread_mouse.thread() let sending_to_win 43c⟩
        (match sending_to_win with
        | true ->
          ⟨Thread_mouse.thread() when sending_to_win 43d⟩
        | false ->
          ⟨Thread_mouse.thread() when not sending_to_win 44d⟩
      )
    )
    done
```

5.2.1 Application mouse events

⟨`Thread_mouse.thread()` *let* `sending_to_win` 43c⟩≡ (43b)
```
  let sending_to_win =
    match Globals.win () with
    | Some (w : Window.t) ->
      (* less: logical coordinates with winput.img.r and winput.screenr *)
      let xy = m.pos in
      (* less: goto scrolling if scroll buttons *)
      let inside = Window.pt_inside_border xy w in
      (* todo: set scrolling *)
      (* todo: set moving *)
      (* less: || scrolling *)
      inside && (w.mouse_opened || m.buttons.left)
    | None -> false
  in
```

⟨`Thread_mouse.thread()` *when* `sending_to_win` 43d⟩≡ (43b)
```
  (* could assert that Globals.win() <> None *)
  Globals.win () |> Option.iter (fun (w : Window.t) ->
    (if not (Mouse.has_click m)
    then Wm.corner_cursor_or_window_cursor w m.pos mouse
    (* todo: why if click then not corner cursor? *)
```

```
    else Wm.window_cursor w m.pos  mouse
    );

    (* less: send logical coordinates *)
    Event.send w.chan_mouse m |> Event.sync
  )
```

⟨Window.t *mouse fields* 44a⟩≡ (32) 47g▷
```
  (* Threads_window.thread <-- Thread_mouse.thread (<-- Mouse.thread) *)
  chan_mouse: Mouse.state Event.channel;
```

⟨*signature* Wm.corner_cursor_or_window_cursor 44b⟩≡ (117)
```
  val corner_cursor_or_window_cursor:
    Window.t -> Point.t -> Mouse.ctl -> unit
```

⟨*signature* Wm.window_cursor 44c⟩≡ (117)
```
  val window_cursor:
    Window.t -> Point.t -> Mouse.ctl -> unit
```

5.2.2 Windowing system mouse events

⟨Thread_mouse.thread() *when not* sending_to_win 44d⟩≡ (43b)
```
  let wopt = Globals.window_at_point m.pos in
  (match wopt with
  | Some w -> Wm.corner_cursor_or_window_cursor w m.pos mouse
  | None -> Mouse.reset_cursor mouse
  );
  (* todo: if moving and buttons *)
  (* todo: set corner cursor again part2 *)

  if Mouse.has_click m
  then
    let under_mouse =
      ⟨Thread_mouse.thread() compute under_mouse 45a⟩
    in
    (match under_mouse with
    ⟨Thread_mouse.thread() match under_mouse cases 45b⟩
    | _ -> raise (Impossible "Mouse.has_click so one field is true")
    );
  (* todo: reset moving *)
  (* less: drain *)
```

⟨*function* Globals.window_at_point 44e⟩≡ (99c)
```
  (* old: was called wpointto in rio *)
  let window_at_point pt =
    let res = ref None in
    windows |> Hashtbl.iter (fun _k w ->
      if Rectangle.pt_in_rect pt w.W.screenr && not w.W.deleted
      then
        match !res with
        | None -> res := Some w
        | Some w2 when w.W.topped > w2.W.topped -> res := Some w
        | _ -> ()
    );
    !res
```

⟨*type* Thread_mouse.under_mouse 44f⟩≡ (113b)
```
  type under_mouse =
    | Nothing
    | CurrentWin of Window.t
    | OtherWin   of Window.t
```

⟨Thread_mouse.thread() *compute* under_mouse 45a⟩≡ (44d)
```
match wopt, Globals.win () with
| None, _ -> Nothing
(* less: look if w2.topped > 0? seems useless *)
| Some w1, Some w2 when w1 == w2 -> CurrentWin w1
| Some w, _ -> OtherWin w
```

⟨Thread_mouse.thread() *match* under_mouse *cases* 45b⟩≡ (44d) 45c▷
```
(* TODO: remove; just because hard to right click on QEMU and laptop*)
| Nothing when m.buttons.left ->
  wm_menu caps m.pos Mouse.Left exitchan
    mouse (display, desktop, view, font) fs
```

⟨Thread_mouse.thread() *match* under_mouse *cases* 45c⟩+≡ (44d) ◁45b 45d▷
```
| ((*Nothing |*) CurrentWin _) when m.buttons.left ->
  ()
```

⟨Thread_mouse.thread() *match* under_mouse *cases* 45d⟩+≡ (44d) ◁45c 45e▷
```
| Nothing when m.buttons.middle ->
   middle_click_system m mouse
```

⟨Thread_mouse.thread() *match* under_mouse *cases* 45e⟩+≡ (44d) ◁45d 56a▷
```
| CurrentWin (w : Window.t) when m.buttons.middle ->
  if not w.mouse_opened
  then middle_click_system m mouse
```

⟨*function* Thread_mouse.middle_click_system 45f⟩≡ (113b)
```
let middle_click_system _m _mouse =
  Logs.err (fun m -> m "Todo: middle click")
```

5.3 Window threads

⟨*signature* Threads_window.thread 45g⟩≡ (113c)
```
(* Thread listening to keyboard/mouse/wm events for a particular window
 * and dispatching to the right channel.
 *)
val thread: Window.t -> unit
```

⟨*type* Threads_window.event 45h⟩≡ (114a)
```
type event =
  (* reading from keyboard thread *)
  | Key   of Keyboard.key
  (* reading from mouse thread *)
  | Mouse of Mouse.state
  (* reading from many places *)
  | Cmd   of Window.cmd

  ⟨Threads_window.event other cases 77e⟩
```

⟨*function* Threads_window.thread 45i⟩≡ (114a)
```
let thread (w : Window.t) =
  (* less: threadsetname *)
  ⟨Threads_window.thread() channels creation 77d⟩

  while true do
    let ev = (
    (* receive *)
    [
```

```
      Event.receive w.chan_keyboard |> wrap (fun x -> Key x);
      Event.receive w.chan_mouse    |> wrap (fun x -> Mouse x);
      Event.receive w.chan_cmd      |> wrap (fun x -> Cmd x);
    ] @
      ⟨Threads_window.thread() other select elements 77b⟩
      []
    ) |> Event.select
    in
    (match ev with
    | Key key ->
      key_in w key
    | Mouse m ->
      mouse_in w m
    | Cmd cmd ->
      (* todo: if return Exited then threadsexit and free channels.
       * When answer Exited? when Cmd is Exited?
       *)
      cmd_in w cmd
    ⟨Threads_window.thread() match ev other cases 77f⟩
    );
    if not w.deleted
    then Image.flush w.img;
  done
```

⟨*function* Threads_window.wrap 46a⟩≡ (114a)
```
let wrap f =
  fun ev -> Event.wrap ev f
```

⟨Window.t *command fields* 46b⟩≡ (32)
```
(* Threads_window.thread <-- Thread_mouse.thread? | ?? *)
(* less: also list of cmds? [20]? *)
chan_cmd: cmd Event.channel;
```

⟨*type* Window.cmd 46c⟩≡ (116a)
```
type cmd =
  ⟨Window.cmd cases 46d⟩
```

⟨Window.cmd *cases* 46d⟩≡ (46c) 46f▷
```
| Delete
```

⟨Window.t *other fields* 46e⟩≡ (32)
```
(* todo: why need this? race between delete window and other operation? *)
mutable deleted: bool;
```

⟨Window.cmd *cases* 46f⟩+≡ (46c) ◁46d 66c▷
```
(*
  | Move of Image.t * Rectangle.t
  | Refresh
  | Wakeup
  (* less: RawOff | RawOn? HoldOn | HoldOff *)
*)
```

5.3.1 Keyboard events listening

⟨*function* Threads_window.key_in 46g⟩≡ (114a)
```
(* input from user *)
let key_in (w : Window.t) (key : Keyboard.key) =
  (* less: if key = 0? when can happen? EOF? Ctrl-D? *)
  if not w.deleted then begin
    match w.raw_mode, w.mouse_opened with
    ⟨Threads_window.key_in() match raw_mode and mouse_opened cases 47a⟩
  end
```

⟨Threads_window.key_in() *match* raw_mode *and* mouse_opened *cases* 47a⟩≡ (46g) 47c▷
```
(* todo: if holding *)
| false, false ->
  (* less: snarf *)
  Terminal.key_in w.terminal key
```

⟨*signature* Terminal.key_in 47b⟩≡ (101d)
```
val key_in: t -> Keyboard.key -> unit
```

⟨Threads_window.key_in() *match* raw_mode *and* mouse_opened *cases* 47c⟩+≡ (46g) ◁47a 47d▷
```
| true, true (* less: || q0 == nr *) ->
  Queue.add key w.raw_keys
```

⟨Threads_window.key_in() *match* raw_mode *and* mouse_opened *cases* 47d⟩+≡ (46g) ◁47c 47e▷
```
(* less: in theory we should allow also special navigation keys here *)
| true, false  ->
  failwith "key_in: TODO: raw mode in textual window"
```

⟨Threads_window.key_in() *match* raw_mode *and* mouse_opened *cases* 47e⟩+≡ (46g) ◁47d
```
| false, true  ->
  failwith "key_in: TODO: buffered mode in graphical window"
```

5.3.2 Mouse events listening

⟨*function* Threads_window.mouse_in 47f⟩≡ (114a)
```
let mouse_in (w : Window.t) (m : Mouse.state) =
  w.last_mouse <- m;
  match w.mouse_opened with
  | true ->
    w.mouse_counter <- w.mouse_counter + 1;
    (* less: limit queue length? *)
    if m.buttons <> w.last_buttons
    then begin
      Queue.add (m, w.mouse_counter) w.mouseclicks_queue;
      w.last_buttons <- m.buttons
    end;

  | false -> failwith "mouse_in: mouse not opened todo"
```

⟨Window.t *mouse fields* 47g⟩+≡ (32) ◁44a 47h▷
```
(* we do not queue all mouse states (we queue just the clicks/releases);
 * for the rest (moving the mouse) we just keep the last state.
 *)
mutable last_mouse: Mouse.state;
```

⟨Window.t *mouse fields* 47h⟩+≡ (32) ◁47g 47i▷
```
(* Note that we do not queue all mouse states; just the clicks/releases,
 * otherwise the queue would be too big when you move around the mouse.
 * less: max size = ? mutex around? recent queue.mli says not thread-safe
 *)
mouseclicks_queue: (Mouse.state * mouse_counter) Queue.t;
```

⟨Window.t *mouse fields* 47i⟩+≡ (32) ◁47h 47j▷
```
(* ?? how differ from last_mouse.buttons? *)
mutable last_buttons: Mouse.buttons;
```

⟨Window.t *mouse fields* 47j⟩+≡ (32) ◁47i 77a▷
```
(* less: could have simpler mouse_new_event: bool? *)
mutable mouse_counter: mouse_counter;
```

5.3.3 Control events listening

⟨*function* Threads_window.cmd_in 48a⟩≡ (114a)
```
let cmd_in (w : Window.t) (cmd : cmd) =
  match cmd with
  ⟨Threads_window.cmd_in() match cmd cases 65b⟩
```

5.4 Filesystem server thread

⟨*signature* Threads_fileserver.thread 48b⟩≡ (111a)
```
(*
  will serve all the virtual devices for new windows/processes created by rio.
 *)
val thread: Fileserver.t -> unit
```

⟨*function* Threads_fileserver.thread 48c⟩≡ (111b)
```
(* the master *)
let thread (fs : Fileserver.t) =
  (* less: threadsetname *)

  while true do
    let req : P9.message = P9.read_9P_msg fs.server_fd in

    ⟨Threads_fileserver.thread in loop, debug 92e⟩

    (match req.typ with
    | P9.T x -> dispatch fs req x
    | P9.R _x ->
      (* less: Ebadfcall *)
      raise (Impossible (spf "got a response request: %s" (P9.str_of_msg req)))
    );
    (* for Version first-message check *)
    first_message := false
  done
```

⟨*constant* Threads_fileserver.first_message 48d⟩≡ (111b)
```
(* for Version *)
let first_message = ref true
```

⟨*function* Threads_fileserver.dispatch 48e⟩≡ (111b)
```
let dispatch (fs : Fileserver.t) (req : P9.message) (request_typ : P9.Request.t) =
  match request_typ with
  ⟨Threads_fileserver.dispatch() match request_typ cases 48f⟩
```

5.4.1 `Version` request

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 48f⟩≡ (48e) 69a▷
```
(* Version *)
| T.Version (msize, str) ->
  (match () with
  | _ when not !first_message ->
    error fs req "version: request not first message"
  | _ when msize < 256 ->
    error fs req "version: message size too small";
  | _ when str <> "9P2000" ->
    error fs req "version: unrecognized 9P version";
  | _ ->
    fs.message_size <- msize;
```

```
      answer fs {req with P9.typ = P9.R (R.Version (msize, str)) }
    )
```

⟨Fileserver.t *other fields* 49a⟩≡ (36e)
```
  (* refined after Tversion first message *)
  mutable message_size: int;
```

⟨*function* Threads_fileserver.answer 49b⟩≡ (111b)
```
  let answer (fs : Fileserver.t) (res : P9.message) =
    ⟨Threads_fileserver.answer() debug 92f⟩
    P9.write_9P_msg res fs.server_fd
```

Chapter 6

Cursors

6.1 Cursor graphics

6.1.1 Classic cursors

⟨*signature* Cursors.crosscursor 50a⟩≡ (96a)
```
val crosscursor : Cursor.t
```

⟨*signature* Cursors.boxcursor 50b⟩≡ (96a)
```
val boxcursor   : Cursor.t
```

⟨*signature* Cursors.sightcursor 50c⟩≡ (96a)
```
val sightcursor : Cursor.t
```

⟨*constant* Cursors.crosscursor 50d⟩≡ (96b)
```
(* when create a new window (sweep()) *)
let crosscursor = {
  offset = { x = -7; y = -7; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.boxcursor 50e⟩≡ (96b)
```
(* when move a window (drag()) *)
let boxcursor = {
  offset =  {x = -7; y = -7; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
      |];
  }
```

⟨*constant* Cursors.sightcursor 51a⟩≡ (96b)

```
  (* when select a window (point_to()) *)
  let sightcursor = {
    offset =  {x = -7; y = -7; };
    clr = Cursor.ints_to_bytes
      [| 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;
      |];
    set = Cursor.ints_to_bytes
      [| 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;
      |];
  }
```

⟨*constant* Cursors._whitearrow 51b⟩≡ (96b)

```
  (* for holding mode *)
  let _whitearrow = {
    offset =  {x = 0; y = 0; };
    clr = Cursor.ints_to_bytes
      [| 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;
      |];
    set = Cursor.ints_to_bytes
      [| 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;
      |];
  }
```

⟨*constant* Cursors._query 51c⟩≡ (96b)

```
  (* ?? *)
  let _query = {
    offset =  {x = -7;y = -7; };
    clr = Cursor.ints_to_bytes
      [| 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;
      |];
    set = Cursor.ints_to_bytes
      [| 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

⟨*type* Cursors.corner 52a⟩≡ (96b)
```
type corner =
  | TopLeft    | Top    | TopRight
  | Left                | Right
  | BottomLeft | Bottom | BottomRight
```

⟨*constant* Cursors.tl 52b⟩≡ (96b)
```
let tl = {
  offset =  {x = -4; y = -4; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.t 52c⟩≡ (96b)
```
let t = {
  offset =  {x = -7; y = -8; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.tr 52d⟩≡ (96b)
```
let tr = {
  offset =  {x = -11; y = -4; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.r 53a⟩≡ (96b)

```
let r = {
  offset =  {x = -8; y = -7; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.br 53b⟩≡ (96b)

```
let br = {
  offset =  {x = -11; y = -11; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.b 53c⟩≡ (96b)

```
let b = {
  offset =  {x = -7; y = -7; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
  set = Cursor.ints_to_bytes
    [| 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;
    |];
}
```

⟨*constant* Cursors.bl 53d⟩≡ (96b)

```
let bl = {
  offset =  {x = -4; y = -11; };
  clr = Cursor.ints_to_bytes
    [| 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;
    |];
```

```
    set = Cursor.ints_to_bytes
      [| 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;
      |];
  }
```

⟨*constant* Cursors.l 54a⟩≡ (96b)

```
  let l = {
    offset =  {x = -7; y = -7; };
    clr = Cursor.ints_to_bytes
      [| 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;
      |];
    set = Cursor.ints_to_bytes
      [| 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

6.2.1 riosetcursor()

6.2.2 cornercursor()

⟨*function* Wm._corner_cursor 54b⟩≡ (116c)

```
  let _corner_cursor (w : Window.t) (pt : Point.t) (mouse : Mouse.ctl) =
    if Window.pt_on_border pt w
    then Mouse.set_cursor mouse (Cursors.which_corner_cursor w.screenr pt)
```

⟨*function* Wm.corner_cursor_or_window_cursor 54c⟩≡ (116c)

```
  let corner_cursor_or_window_cursor (w : Window.t) (pt : Point.t) (mouse : Mouse.ctl) =
    if Window.pt_on_border pt w
    then Mouse.set_cursor mouse (Cursors.which_corner_cursor w.screenr pt)
    else window_cursor w pt mouse
```

⟨*function* Window.pt_on_border 54d⟩≡ (116a)

```
  (* old: was called winborder in rio *)
  let pt_on_border pt w =
    Rectangle.pt_in_rect pt w.screenr && not (pt_inside_border pt w)
```

⟨*function* Window.pt_inside_border 54e⟩≡ (116a)

```
  (* old: was not an helper in rio, but should to be consistent with winborder.
   * alt: pt_on_content (window border vs window content in Windows.nw)
   *)
  let pt_inside_border pt w =
    Rectangle.pt_in_rect pt (Rectangle.insetrect window_border_size w.screenr)
```

⟨*signature* Cursors.which_corner_cursor 54f⟩≡ (96a)

```
  val which_corner_cursor: Rectangle.t -> Point.t -> Cursor.t
```

⟨*function* Cursors.which_corner_cursor 55a⟩≡ (96b)
```
let which_corner_cursor rect p =
  let corner = which_corner rect p in
  match corner with
  | TopLeft -> tl
  | Top -> t
  | TopRight -> tr
  | Left -> l
  | Right -> r
  | BottomLeft -> bl
  | Bottom -> b
  | BottomRight -> br
```

⟨*type* Cursors.portion 55b⟩≡ (96b)
```
type portion = Inf | Middle | Sup
```

⟨*function* Cursors.which_corner 55c⟩≡ (96b)
```
let which_corner r p =
  let left_right = portion p.x r.min.x r.max.x in
  let bottom_top = portion p.y r.min.y r.max.y in
  match left_right, bottom_top with
  | Inf, Inf -> TopLeft
  | Inf, Middle -> Left
  | Inf, Sup -> BottomLeft
  | Middle, Inf -> Top
  | Middle, Middle ->
    raise (Impossible "which_corner: pt_on_frame should not generate this")
  | Middle, Sup -> Bottom
  | Sup, Inf -> TopRight
  | Sup, Middle -> Right
  | Sup, Sup -> BottomRight
```

⟨*function* Cursors.portion 55d⟩≡ (96b)
```
let portion x low high =
  (* normalize to low *)
  let x = x - low in
  let high = high - low in
  (* TODO: unused low? *)
  let _low = 0 in

  match () with
  | _ when x < 20 -> Inf
  | _ when x > high - 20 -> Sup
  | _ -> Middle
```

6.2.3 wsetcursor()

Chapter 7

Window Manager

7.1 Overview

7.2 Right-click system menu

⟨Thread_mouse.thread() *match* under_mouse *cases* 56a⟩+≡ (44d) ◁45e 64d▷
```
| (Nothing | CurrentWin _) when m.buttons.right ->
  wm_menu caps m.pos Mouse.Right exitchan
    mouse (display, desktop, view, font) fs
```

⟨*function* Thread_mouse.wm_menu 56b⟩≡ (113b)
```
(* bind to right-click *)
let wm_menu (caps : < Cap.fork; .. >) (pos : Point.t) button
     (exitchan : Exit.t Event.channel)
     (mouse : Mouse.ctl)
     (display, desktop, view, font)
     (fs : Fileserver.t) =
  (* todo: set (and later restore) sweeping to true *)

  let items = [
    ⟨Thread_mouse.wm_menu() items elements 56c⟩
  ] @
    ⟨Thread_mouse.wm_menu() items hidden window elements 68b⟩
  in
  Menu_ui.menu items pos button
    mouse (display, desktop, view, font)
```

⟨Thread_mouse.wm_menu() items *elements* 56c⟩≡ (56b) 56d▷
```
"Exit", (fun () ->
  Event.send exitchan Exit.OK |> Event.sync;
);
```

7.3 Window borders click

7.4 Wctlmesg

7.5 Window creation

⟨Thread_mouse.wm_menu() items *elements* 56d⟩+≡ (56b) ◁56c 65a▷
```
(* less: the first item get selected the very first time; QEMU bug?  *)
"New", (fun () ->
```

```
    let img_opt = Mouse_action.sweep mouse (display, desktop, font) in
    img_opt |> Option.iter (fun img ->
       Wm.new_win caps img "/bin/rc" [|"rc"; "-i"|] None (mouse, fs, font)
      (*
         Wm.new_win img "/tests/xxx/test_rio_graph_app1"
           [|"/tests/xxx/test_rio_graph_app1"|] None (mouse, fs, font)
         Wm.new_win img "/tests/rio/8.out"
           [|"/tests/rio/8.out"|] None (mouse, fs, font)
         Wm.new_win img "/tests/xxx/hellorio"
           [|"/tests/xxx/hellorio"|] None (mouse, fs, font)
         Wm.new_win img "/tests/xxx/test_rio_console_app1"
           [|"/tests/xxx/test_rio_console_app1"|] None (mouse, fs, font)
      *)
    )
  );
```

⟨*signature* `Wm.new_win` 57a⟩≡ (117)

```
  (* This is called from Thread_mouse.ml when right-click and select New.
   * Internally it will:
   * - create a new Window.t and adds it to Globals.windows
   * - create a new window thread
   * - fork a new process, mount /mnt/wsys from fileserver and then binds
   *   /mnt/wsys to /dev so the process will have virtual /dev/{cons,winname,...}
   *   and finally run the cmd in it
   *)
  val new_win:
    < Cap.fork; Cap.exec; Cap.chdir; Cap.mount; Cap.bind; .. > ->
    Image.t -> string (* cmd *) -> string array (* argv (including argv0) *) ->
    Fpath.t option (* pwd option *) ->
    (Mouse.ctl * Fileserver.t * Font.t) ->
    unit
```

7.5.1 Window thread creation

⟨*function* `Wm.new_win` 57b⟩≡ (116c)

```
  (* New window! new process! new thread!
   *
   * less: hideit, pid (but 0, or if != 0 -> use another func), scrolling
   *
   * CLI.thread_main -> Thread_mouse.thread -> Thread_mouse.wm_menu -> <>
   *)
  let new_win (caps: < Cap.fork; Cap.exec; Cap.chdir; ..>) (img : Image.t)
      (cmd : string) (argv : string array) (pwd_opt : Fpath.t option)
      (_mouse, fs, font) : unit =

    (* A new Window.t *)

    (* less: cpid channel?  *)
    (* less: scrollit *)

    let w : Window.t = Window.alloc img font in

    (* less: wscrdraw here? (instead of in alloc, ugly) and draw(cols[BACK])? *)
    (* less: incref? *)

    (* simpler: draw_border w Window.Selected;
     * but done already later in set_current_and_repaint_borders
     *)
```

```
    (* A new window thread *)

    Hashtbl.add Globals.windows w.id w;
    let _win_thread = Thread.create !threads_window_thread_func w in

    (* less: if not hideit *)
    set_current_and_repaint (Some w);
    Image.flush img;

    (* A new window process *)

    pwd_opt |> Option.iter (fun str -> w.pwd <- str);

    (* TODO: Thread.critical_section := true; *)
    (* Logs.warn (fun m -> m "TODO: Thread.critical_section"); *)

    let res = CapUnix.fork caps () in
    (match res with
    | -1 -> failwith "fork returned -1"
    | 0 ->
      (* child *)
      Processes_winshell.run_cmd_in_window_in_child_of_fork caps cmd argv w fs
    | pid ->
      ⟨Wm.new_win() when parent in fork with pid child 58a⟩
    )
```

⟨Wm.new_win() *when parent in fork with* pid *child* 58a⟩≡ (57b) 58b▷

```
  (* parent *)
  (* TODO: Thread.critical_section := false; *)
  w.pid <- pid;
```

⟨Wm.new_win() *when parent in fork with* pid *child* 58b⟩+≡ (57b) ◁58a 58c▷

```
  (* todo: how know if pb in child that require us then from
   * delete the window? need a cpid!
   *)
  (* less: notefd *)
```

⟨Wm.new_win() *when parent in fork with* pid *child* 58c⟩+≡ (57b) ◁58b 58d▷

```
  (* old: was in wsetpid() *)
  w.label <- spf "rc %d" pid;
```

⟨Wm.new_win() *when parent in fork with* pid *child* 58d⟩+≡ (57b) ◁58c

```
  (* less: not too late? race with child to access /dev/winname? *)
  let winname = spf "window.%d" w.id in
  w.winname <- winname;
  Draw_ipc.name_image w.img winname;
  (* less: namecount and retry again if already used *)
```

⟨Window.t *process fields* 58e⟩≡ (32) 58f▷

```
  (* not really mutable, but set after Window.alloc() *)
  mutable pid: int;
```

⟨Window.t *process fields* 58f⟩+≡ (32) ◁58e

```
  (* can be changed through /mnt/wsys/wdir *)
  mutable pwd: Fpath.t;
  (* todo? notefd *)
```

⟨*signature* Wm.threads_window_thread_func 58g⟩≡ (117)

```
  (* for mutual dependencies in new_win *)
  val threads_window_thread_func: (Window.t -> unit) ref
```

⟨*global* `Wm.threads_window_thread_func` 59a⟩≡ (116c)
```
(* will be set to Threads_window.thread in CLI.ml *)
let threads_window_thread_func: (Window.t -> unit) ref = ref (fun _ ->
  failwith "threads_window_thread_func undefined"
)
```

⟨`CLI.thread_main()` *threads creation* 59b⟩+≡ (40b) ◁41b
```
(* To break some mutual dependencies.
 * The mouse right-click and menu will trigger the creation
 * of new windows and new window threads and call this function.
 *)
Wm.threads_window_thread_func := Threads_window.thread;
```

⟨*signature* `Wm.set_current_and_repaint` 59c⟩≡ (117)
```
(* this also used to have a mouse parameter (should be renamed then
 * set_current_and_repaint_borders_content_and_cursor)
 *)
val set_current_and_repaint:
  Window.t option -> unit
```

7.5.2 `Window` allocation

⟨*function* `Window.alloc` 59d⟩≡ (116a)
```
(* ... -> Thread_mouse.wm_menu -> Wm.new_win -> <> *)
let alloc (img : Display.image) (font : Font.t) : t =
  incr wid_counter;
  incr topped_counter;

  let w =
  {
    id = !wid_counter;
    (* will be set in caller *)
    winname = "";
    label = "<unnamed>";

    (* set later in Wm.ml in the caller *)
    pid = -1;
    pwd = Fpath.v (Sys.getcwd ());

    img = img;
    screenr = img.r;

    topped = !topped_counter;

    mouse_opened   = false;
    consctl_opened = false;
    raw_mode       = false;

    deleted = false;

    mouse_cursor = None;

    chan_mouse    = Event.new_channel ();
    chan_keyboard = Event.new_channel ();
    chan_cmd      = Event.new_channel ();

    raw_keys = Queue.create ();
    mouseclicks_queue = Queue.create ();
```

```
    mouse_counter = 0;
    last_count_sent = 0;
    last_mouse = Mouse.fake_state;
    last_buttons = Mouse.nobuttons;

    chan_devmouse_read = Event.new_channel ();

    chan_devcons_read  = Event.new_channel ();
    chan_devcons_write = Event.new_channel ();

    terminal = Terminal.alloc img font;

    auto_scroll = false;
  }
  in
  w
```

⟨*signature* `Terminal.alloc` 60a⟩≡ (101d)
```
val alloc: Image.t -> Font.t -> t
```

7.5.3 Window process creation

⟨*signature* `Processes_winshell.run_cmd_in_window_in_child_of_fork` 60b⟩≡ (101a)
```
val run_cmd_in_window_in_child_of_fork:
  < Cap.chdir; Cap.exec; Cap.mount; Cap.bind; .. > ->
  string -> string array -> Window.t -> Fileserver.t -> unit
```

⟨*function* `Processes_winshell.run_cmd_in_window_in_child_of_fork` 60c⟩≡ (101b)
```
let run_cmd_in_window_in_child_of_fork
    (caps : < Cap.chdir; Cap.exec; Cap.mount; Cap.bind; .. >)
    (cmd : string) (argv : string array)
    (w : Window.t) (fs : Fileserver.t) =

  (*Unix1*)CapUnix.chdir caps !!(w.pwd);
  (* todo? rfork for copy of namespace/fd/env, but ape fork does that? *)

  (*/* close server end so mount won't hang if exiting */*)
  Unix1.close fs.server_fd;
  (* todo: close on exec? *)

  Plan9.mount caps fs.clients_fd (-1) (Fpath.v "/mnt/wsys") Plan9.MRepl (spf "%d" w.id);
  Plan9.bind caps (Fpath.v "/mnt/wsys") (Fpath.v "/dev") Plan9.MBefore;

  (* less: wclose for ref counting *)
  (* todo: handle errors? Unix_error? then communicate failure to parent? *)
  (* bugfix: do not forget the last perm argument below! otherwise partial
   * application and stdin/stdout are never reopened (in fact the dup
   * below even crashes the plan9 kernel)
   *)
  Unix1.close Unix1.stdin;
  let _fd = Unix1.openfile "/dev/cons" [Unix1.O_RDONLY] 0o666 in
  (* TODO? assert fd = 0? possible in OCaml? *)
  Unix1.close Unix1.stdout;
  let _fd = Unix1.openfile "/dev/cons" [Unix1.O_WRONLY] 0o666 in
  (* TODO? assert fd = 1? possible in OCaml? *)
  Unix1.dup2 Unix1.stdout Unix1.stderr;

  (* less: notify nil *)
  (*Unix2*)CapUnix.execv caps cmd argv |> ignore;
```

```
  (* should never reach this point *)
  failwith "exec failed"
```

7.5.4 Namespace adjustments

7.5.5 Public layer: `wsetname()`

7.5.6 Window painting

⟨*function* `Wm.set_current_and_repaint` 61a⟩≡ (116c)

```
(* old: was called wcurrent() in rio.
 * alt: this function also sets the window cursor in rio-C, but this
 * requires then to pass a mouse parameter, which in turn requires to
 * pass the mouse in Reshape, which then requires to pass the mouse
 * parameter to hide_win and show_win and other functions which is
 * not super elegant. This is why I prefer to not pass a mouse parameter
 * which means the user may have to move the mouse to see the cursor
 * correctly updated after certain wm operations. I think this
 * tradeoff is ok.
 *)
let set_current_and_repaint wopt (*mouse*) =
  (* less: if wkeyboard *)
  let old = !Globals.current in
  Globals.current := wopt;
  (match old, wopt with
  | Some w2, Some w when not (w2 == w) ->
    (* bugfix: was doing repaint w, hmm *)
    repaint w2
  | _ -> ()
  );
  wopt |> Option.iter (fun w ->
    repaint w;
    (* TODO: do that in caller? so no need pass mouse? *)
    (* window_cursor w ptTODO mouse;*)

    (* todo: wakeup? why? *)
    ()
  )
```

⟨*function* `Wm.repaint` 61b⟩≡ (116c)

```
(* old: was called wrepaint in rio *)
let repaint (w : Window.t) =
  let status =
    match Globals.win () with
    | Some w2 when w2 == w -> Window.Selected
    | _ -> Window.Unselected
  in
  draw_border w status;
  (* less: wsetcursor again? *)
  w.terminal.Terminal.is_selected <- (status = Window.Selected);
  if not w.mouse_opened
  then Terminal.repaint w.terminal
```

⟨*type* `Window.border_status` 61c⟩≡ (116a)

```
type border_status =
  | Selected
  | Unselected
```

⟨*function* `Wm.draw_border` 62a⟩≡ (116c)
```
let draw_border (w : Window.t) (status : Window.border_status) =
  let img : Display.image = w.img in
  (* less: if holding? *)
  let color =
    match status with
    | Window.Selected   -> !Globals.title_color
    | Window.Unselected -> !Globals.title_color_light
  in
  Polygon.border img img.r Window.window_border_size color Point.zero
```

⟨*signature* `Terminal.repaint` 62b⟩≡ (101d)
```
val repaint: t -> unit
```

7.5.7 Mouse action `sweep()`

⟨*signature* `Mouse_action.sweep` 62c⟩≡ (100b)
```
(* allows to specify a window area *)
val sweep:
  Mouse.ctl -> (Display.t * Baselayer.t * Font.t) -> Image.t option
```

⟨*type* `Mouse_action.sweep_state` 62d⟩≡ (100c)
```
type sweep_state =
  (* start, no buttons *)
  | SweepInit
  (* right click *)
  | SweepRightClicked of Point.t
  (* move while holding right click *)
  | SweepMove of Point.t * Point.t * Image.t option
  (* release right click *)
  | SweepUnclicked of Image.t option
  ⟨Mouse_action.sweep_state other cases 62e⟩
```

⟨`Mouse_action.sweep_state` *other cases* 62e⟩≡ (62d)
```
(* to factorize cornercursor *)
| SweepReturn of Image.t option

(* error management when left or middle click or too small rectangle *)
| SweepRescue of bool (* clicked state *) * Image.t option
(* wait until no buttons *)
| SweepDrain
```

⟨*function* `Mouse_action.sweep` 62f⟩≡ (100c)
```
let sweep (mouse : Mouse.ctl) (display, desktop, font) : Image.t option =
  (* todo: menuing? but not sweeping? *)
  Mouse.set_cursor mouse Cursors.crosscursor;

  let rec transit = function
    ⟨Mouse_action.sweep() match sweep_state cases 62g⟩
  in
  transit SweepInit
```

⟨`Mouse_action.sweep()` *match* `sweep_state` *cases* 62g⟩≡ (62f) 63a▷
```
| SweepInit ->
  let m = Mouse.read mouse in
  if not (Mouse.has_click m)
  then transit SweepInit
  else
    (* less: force exactly right click? *)
```

```
      (* todo: Mouse.Right at some point *)
      if Mouse.has_button m Mouse.Left
      then transit (SweepRightClicked m.Mouse.pos)
      else transit (SweepRescue (true, None))
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 63a⟩+≡ (62f) ◁62g 63b▷

```
  | SweepRightClicked p0 ->
    (* less: onsceen (using clipr) *)
    transit (SweepMove (p0, p0, None))
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 63b⟩+≡ (62f) ◁63a 63c▷

```
  | SweepMove (p0, p1, old_img_opt) ->
    let m = Mouse.flush_and_read display mouse in
    (match () with
    (* less: force exactly right click? *)
    (* todo: Mouse.Right at some point *)
    | _ when Mouse.has_button m Mouse.Left ->
      if m.Mouse.pos = p1
      then transit (SweepMove (p0, p1, old_img_opt))
      else
        (* less: onscreen *)
        let p1 = m.Mouse.pos in
        let r = Rectangle.canonical p0 p1 in
        if Rectangle.dx r > 5 && Rectangle.dy r > 5 then begin
          (* need that? should be transparent anyway no? *)
          let grey = Color.mk2 0xEE 0xEE 0xEE in
          (* todo? RefreshNothing? *)
          let img = Layer.alloc desktop r grey in
          old_img_opt |> Option.iter Layer.free;
          Polygon.border img r Window.window_border_size
            !Globals.red Point.zero;
          Display.flush display;
          transit (SweepMove (p0, p1, Some img))
        end
        else transit (SweepMove (p0, p1, old_img_opt))
    | _ when not (Mouse.has_click m) ->
      transit (SweepUnclicked (old_img_opt))
    | _ ->
      assert(Mouse.has_click m);
      transit (SweepRescue (true, old_img_opt))
    )
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 63c⟩+≡ (62f) ◁63b 64a▷

```
  | SweepUnclicked (old_img_opt) ->
    (match old_img_opt with
    | None -> transit (SweepReturn None)
    | Some (old_img : Display.image) ->
      let r = old_img.r in
      if Rectangle.dx r < 100 || Rectangle.dy r < 3 * font.Font.height
      then transit (SweepRescue (false, old_img_opt))
      else begin
        (* todo? RefreshBackup? *)
        let img = Layer.alloc desktop r Color.white in
        Layer.free old_img;
        Polygon.border img r Window.window_border_size
          !Globals.red Point.zero;
        (* done in caller but more logical here I think *)
        Display.flush display;
        (* finally!! got an image *)
        transit (SweepReturn (Some img))
```

```
    end
  )
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 64a⟩+≡ (62f) ◁63c 64b▷

```
| SweepReturn img_opt ->
  (* todo: cornercursor? pos! *)
  (* less: moveto to force cursor update? ugly ... *)
  (* less: menuing = false *)
  img_opt
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 64b⟩+≡ (62f) ◁64a 64c▷

```
| SweepRescue (clicked_state, old_img_opt) ->
  old_img_opt |> Option.iter Layer.free;
  if clicked_state
  then transit SweepDrain
  else transit (SweepReturn None)
```

⟨Mouse_action.sweep() *match* sweep_state *cases* 64c⟩+≡ (62f) ◁64b

```
| SweepDrain ->
  let m = Mouse.read mouse in
  if not (Mouse.has_click m)
  then transit (SweepReturn None)
  else transit SweepDrain
```

7.6 Window focus

⟨Thread_mouse.thread() *match* under_mouse *cases* 64d⟩+≡ (44d) ◁56a 64e▷

```
| OtherWin w when m.buttons.left ->
  Wm.top_win w
  (* less: should drain and wait that release up, unless winborder *)
```

⟨Thread_mouse.thread() *match* under_mouse *cases* 64e⟩+≡ (44d) ◁64d

```
| OtherWin w when m.buttons.middle || m.buttons.right ->
  Wm.top_win w
  (* todo: should goto again, may need to send event *)
```

⟨*signature* Wm.top_win 64f⟩≡ (117)

```
val top_win:
  Window.t -> unit
```

⟨*function* Wm.top_win 64g⟩≡ (116c)

```
let top_win (w : Window.t) =
  if w.topped = !Window.topped_counter
  then ()
  else begin
    Layer.put_to_top w.img;
    set_current_and_repaint (Some w);
    Image.flush w.img;

    incr Window.topped_counter;
    w.topped <- !Window.topped_counter;
  end
```

7.7 Window deletion

⟨Thread_mouse.wm_menu() items *elements* 65a⟩+≡ (56b) ◁56d 66a▷
```
"Delete", (fun () ->
  let wopt = Mouse_action.point_to mouse in
  wopt |> Option.iter (fun (w : Window.t) ->
    let cmd = Window.Delete in
    Event.send w.chan_cmd cmd |> Event.sync;
  ));
```

⟨Threads_window.cmd_in() *match* cmd *cases* 65b⟩≡ (48a) 66d▷
```
| Delete ->
  (* less: break if window already deleted *)
  (* todo: delete timeout process *)
  Wm.close_win w
```

⟨*signature* Wm.close_win 65c⟩≡ (117)
```
val close_win:
  Window.t -> unit
```

⟨*function* Wm.close_win 65d⟩≡ (116c)
```
let close_win (w : Window.t) =
  w.deleted <- true;
  Globals.win () |> Option.iter (fun w2 ->
    if w2 == w
    then Globals.current := None;
    (* less: window_cursor  ?*)
  );
  (* less: if wkeyboard *)
  Hashtbl.remove Globals.hidden w.id;
  Hashtbl.remove Globals.windows w.id;
  Layer.free w.img;
  w.img <- Image.fake_image;
  ()
```

7.7.1 Mouse action `pointto()`

⟨*signature* Mouse_action.point_to 65e⟩≡ (100b)
```
val point_to:
  Mouse.ctl -> Window.t option
```

⟨*function* Mouse_action.point_to 65f⟩≡ (100c)
```
let point_to (mouse : Mouse.ctl) : Window.t option =
  (* todo: menuing? but not sweeping? *)
  Mouse.set_cursor mouse Cursors.sightcursor;

  let m = ref (Mouse.read mouse) in
  while not (Mouse.has_click !m) do
    m := Mouse.read mouse;
  done;
  let wopt =
    if Mouse.has_button !m Mouse.Right
    then Globals.window_at_point !m.Mouse.pos
    else None
  in
  (* less: wait and cancel option? *)
  while (Mouse.has_click !m) do
    m := Mouse.read mouse;
  done;
```

```
  (* restore cursor state *)
  Globals.win () |> Option.iter (fun w ->
    Wm.corner_cursor_or_window_cursor w !m.Mouse.pos mouse
  );
  wopt
```

7.8 Window move

⟨Thread_mouse.wm_menu() items *elements* 66a⟩+≡ (56b) ◁65a 66b▷

```
"Move", (fun () -> raise Todo);
```

7.8.1 move()

7.8.2 Mouse action drag()

7.9 Window resize

⟨Thread_mouse.wm_menu() items *elements* 66b⟩+≡ (56b) ◁66a 67c▷

```
(* old: was Reshape but here it's really resizing *)
"Resize", (fun () -> raise Todo);
```

⟨Window.cmd *cases* 66c⟩+≡ (46c) ◁46f

```
(* for resize event but also for hide/show *)
| Reshape of
      Image.t (* can be Layer.t or an off-screen Image.t when hidden *)
      (*Mouse.ctl*) (* needed for window_cursor() when repaint border *)
```

⟨Threads_window.cmd_in() *match* cmd *cases* 66d⟩+≡ (48a) ◁65b

```
| Reshape (new_img : Display.image) ->
  (* less: put all of that in Wm.resize_win ? *)

  if w.deleted
  (* less: free new_img if deleted, but when can happen? *)
  then failwith "window already deleted";

  let r = new_img.r in
  w.screenr <- r;
  Wm.resize_win w new_img;
  (* less: set wctlready to true *)
  (* todo: delete timeout proc for old name of window *)

  (match Rectangle.dx r, Globals.win () with
  | 0, Some w2 when w2 == w ->
    Wm.set_current_and_repaint None
  | _n, Some w2 when (w2 == w) ->
    (* less: could Wm.set_current_and_repaint_borders (Some w) mouse,
     * useless opti I think to special case here w2 == w
     *)
    ()
  | _n, (Some _ | None) ->
    Wm.set_current_and_repaint (Some w)
  );
  (* less: Image.flush new_img, but useless cos done in thread () *)
  ()
```

⟨*signature* `Wm.resize_win` 67a⟩≡ (117)
```
val resize_win:
  Window.t -> Image.t -> unit
```

7.9.1 `resize()`

⟨*function* `Wm.resize_win` 67b⟩≡ (116c)
```
(* less: move boolean parameter, useless opti test dx/dy below catch it *)
let resize_win (w : Window.t) (new_img : Display.image) =
  let old_img : Display.image = w.img in
  let old_r : Rectangle.t = old_img.r in
  let new_r : Rectangle.t = new_img.r in
  if Rectangle.dx old_r = Rectangle.dx new_r &&
     Rectangle.dy old_r = Rectangle.dy new_r
  then Draw.draw new_img new_r old_img None old_r.min;
  (* a layer or image, so when hiding this should make disappear the window *)
  Image.free old_img;
  (* less: screenr set in caller, but could do it here *)
  w.img <- new_img;
  (* todo: wsetname *)

  (* todo: textual window update *)
  draw_border w Window.Selected;
  incr Window.topped_counter;
  w.topped <- !Window.topped_counter;

  (* todo: w.W.resized <- true *)
  (* todo: mouse counter ++ so transmit resize event *)
  ()
```

7.9.2 Mouse action `bandsize()`

7.10 Window visibility

⟨`Thread_mouse.wm_menu()` items *elements* 67c⟩+≡ (56b) ◁66b
```
"Hide", (fun () ->
  let wopt = Mouse_action.point_to mouse in
  wopt |> Option.iter (fun w ->
    Wm.hide_win w
  ));
```

⟨*signature* `Wm.hide_win` 67d⟩≡ (117)
```
(* (those actions used to have a mouse parameter to also update the cursor) *)
val hide_win:
  Window.t ->  unit
```

7.10.1 `hidden`

⟨*global* `Globals.hidden` 67e⟩≡ (99c)
```
(* a subset of 'windows' *)
let hidden: (Window.wid, Window.t) Hashtbl.t = Hashtbl_.create ()
```

7.10.2 hide()

⟨*function* Wm.hide_win 68a⟩≡ (116c)
```
let hide_win (w : Window.t) =
  if Hashtbl.mem Globals.hidden w.id
  (* less: return -1? can happen if window thread take too much time
   * to respond to the Reshape command?
   *)
  then raise (Impossible "window already hidden");
  let old_layer : Display.image = w.img in
  let display = old_layer.display in
  (* this is an image! not a layer, so it will not be visible on screen *)
  let img =
    Image.alloc display w.screenr old_layer.chans false Color.white in
  (* less: return 0 or 1 if can or can not allocate? *)
  Hashtbl.add Globals.hidden w.id w;
  let cmd = Window.Reshape img in
  Event.send w.chan_cmd cmd |> Event.sync;
  ()
```

7.10.3 unhide()

⟨Thread_mouse.wm_menu() items *hidden window elements* 68b⟩≡ (56b)
```
(Globals.hidden |> Hashtbl_.to_list |> List.map (fun (_wid, (w : Window.t)) ->
    (* less: could sort by name, or time it was put in hidden hash *)
    w.label, (fun () ->
      Wm.show_win w desktop
    )))
```

⟨*signature* Wm.show_win 68c⟩≡ (117)
```
val show_win:
  Window.t -> Baselayer.t -> unit
```

⟨*function* Wm.show_win 68d⟩≡ (116c)
```
let show_win (w : Window.t) (desktop : Baselayer.t) =
  let old_img : Display.image = w.img in
  (* back to a layer *)
  let layer = Layer.alloc desktop old_img.r Color.white in
  Hashtbl.remove Globals.hidden w.id;
  let cmd = Window.Reshape layer in
  Event.send w.chan_cmd cmd |> Event.sync;
  ()
```

Chapter 8

Filesystem Server

8.1 Attach

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 69a⟩+≡ (48e) ◁48f 70e▷

```
  (* Attach *)
  | T.Attach (rootfid, _auth_fid_opt, uname, aname) ->
    (* stricter: *)
    if Hashtbl.mem fs.fids rootfid
    then failwith (spf "Attach: fid already used: %d" rootfid);

    (match () with
    | _ when uname <> fs.user ->
      error fs req (spf "permission denied, %s <> %s" uname fs.user)
    | _ ->
    (* less: could do that in a worker thread (if use qlock) *)
    (* less: newlymade, qlock all *)
    (try
       let wid = int_of_string aname in
       let w = Hashtbl.find Globals.windows wid in

       let entry = File.root_entry in
       let file_id = entry.code, wid in
       let qid = File.qid_of_fileid file_id entry.type_ in

       let file = File.{
         fid = rootfid;
         qid = qid;
         entry = entry;
         opened = None;
         win = w;
       } in
       Hashtbl.add fs.fids rootfid file;
       answer fs {req with P9.typ = P9.R (R.Attach qid) }
     with _exn ->
        error fs req (spf "unknown id in attach: %s" aname)
    (* less: incref, qunlock *)
    ))
```

⟨File.t *other fields* 69b⟩+≡ (36g) ◁37f

```
  (* The qid is what is returned by the "server" to identify a file (or dir).
   * It is mutable because a fid can be 'walked' to point to another file
   * on the server.
   *)
  mutable qid: Plan9.qid;
```

⟨*type* File.fileid 70a⟩≡ (98d)
```
(* will generate a Qid.path *)
type fileid = filecode * Window.wid
```

⟨*function* File.qid_of_fileid 70b⟩≡ (98d)
```
let qid_of_fileid file_id typ =
  { path = int_of_fileid file_id;
    typ = typ;
    vers = 0;
  }
```

⟨*function* File.int_of_fileid 70c⟩≡ (98d)
```
let int_of_fileid (qxxx, wid) =
  (wid lsl 8) lor
  int_of_filecode qxxx
```

⟨*function* File.int_of_filecode 70d⟩≡ (98d)
```
let int_of_filecode = function
  | Dir Root -> 0
  | File WinName -> 1
  | File Mouse -> 2
  | File Cons -> 3
  | File ConsCtl -> 4
  | File WinId -> 5
  | File Text -> 6
```

8.2 Walk

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 70e⟩+≡ (48e) ◁69a 72a▷
```
(* Walk *)
| T.Walk (fid, newfid_opt, xs) ->
  check_fid "walk" fid fs;
  let file : File.t = Hashtbl.find fs.fids fid in
  let wid = file.win.id in
  (match file.opened, newfid_opt with
  | Some _, _ ->
    error fs req "walk of open file"
  (* stricter? failwith or error? *)
  | _, Some newfid when Hashtbl.mem fs.fids newfid ->
    error fs req (spf "clone to busy fid: %d" newfid)
  | _ when List.length xs > P9.max_welem ->
    error fs req (spf "name too long: [%s]" (String.concat ";" xs))
  | _ ->
    let file =
      match newfid_opt with
      | None -> file
      | Some newfid ->
        (* clone *)
        (* less: incref on file.w *)
        let newfile = { file with
          File.fid = newfid;
          File.opened = None
          (* todo: nrpart? *)
        } in
        Hashtbl.add fs.fids newfid newfile;
        newfile
    in
    let rec walk qid (entry : File.dir_entry_short) acc xs =
```

```
      match xs with
      | [] -> qid, entry, List.rev acc
      | x::xs ->
        if qid.N.typ <> N.QTDir
        (* will be catched below and transformed in an 9P Rerror message *)
        then failwith "not a directory";

        (match entry.code, x with
        | _Qwsys, ".." -> failwith "walk: Todo '..'"
        | File.Dir File.Root, x ->
          let entry =
            toplevel_entries |> List.find (fun (entry : File.dir_entry_short)->
                     entry.name = x)
          in
          let file_id = entry.code, wid in
          let qid = File.qid_of_fileid file_id entry.type_ in
          (* continue with other path elements *)
          walk qid entry (qid::acc) xs

        (* todo: Wsys, snarf *)
        | _ ->
          raise (Impossible "should be catched by 'not a directory' above")
        )
    in
    (try
      let final_qid, final_entry, qids =
        walk file.qid file.entry [] xs
      in
      file.qid <- final_qid;
      file.entry <- final_entry;
      answer fs { req with P9.typ = P9.R (R.Walk qids) }
    with exn ->
      newfid_opt |> Option.iter (fun newfid ->
        Hashtbl.remove fs.fids newfid
      );
      (match exn with
      (* this can happen many times because /dev is a union-mount so
       * we get walk requests also for /dev/draw/... and other devices
       *)
      | Not_found ->
        error fs req "file does not exist"
      | Failure s when s = "not a directory" ->
        error fs req "not a directory"
      (* internal error then *)
      | _ -> raise exn
      )
    )
  )
```

⟨*function* Threads_fileserver.check_fid 71⟩≡ (111b)

```
  (* less: could be check_and_find_fid to factorize more code in dispatch() *)
  let check_fid op fid (fs : Fileserver.t) =
    (* stricter: *)
    if not (Hashtbl.mem fs.fids fid)
    then failwith (spf "%s: unknown fid %d" op fid)
```

8.2.1 Cloning `fid`

8.2.2 `..`

8.2.3 Error management

8.3 Open

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 72a⟩+≡ (48e) ◁70e 72b▷

```
(* Open *)
| T.Open (fid, flags) ->
  check_fid "open" fid fs;
  let file = Hashtbl.find fs.fids fid in
  let w = file.win in
  (* less: OTRUNC | OCEXEC | ORCLOSE, and remove DMDIR| DMAPPEND from perm *)
  let perm = file.entry.perm in
  if flags.x || (flags.r && not perm.r)
             || (flags.w && not perm.w)
  then error fs req "permission denied"
  else
    if w.deleted
    then error fs req "window deleted"
    else
    (* less: could do that in a worker thread *)
    (try
       (match file.entry.code with
       | File.File devid ->
         let dev = device_of_devid devid in
         dev.D.open_ w
       | File.Dir _dir ->
         (* todo: nothing to do for dir? ok to open a dir? *)
         ()
       );
       file.opened <- Some flags;
       let iounit = fs.message_size - P9.io_header_size in
       answer fs { req with P9.typ = P9.R (R.Open (file.qid, iounit)) }
     with Device.Error str ->
       error fs req str
    )
```

8.4 Clunk/Close

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 72b⟩+≡ (48e) ◁72a 73a▷

```
(* Clunk *)
| T.Clunk (fid) ->
  check_fid "clunk" fid fs;
  let file = Hashtbl.find fs.fids fid in
  (match file.opened with
  | Some _flags ->
    (match file.entry.code with
    | File.File devid ->
      let dev = device_of_devid devid in
      dev.D.close file.win
      (* todo: wclose? *)
    | File.Dir _ ->
      ()
    )
```

```
    (* stricter? can clunk a file not opened? *)
    | None ->
      (* todo: winclosechan *)
      ()
    );
    Hashtbl.remove fs.fids fid;
    answer fs { req with P9.typ = P9.R (R.Clunk) }
```

8.5 Read

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 73a⟩+≡ (48e) ◁72b 73b▷

```
  (* Read *)
  | T.Read (fid, offset, count) ->
    check_fid "read" fid fs;
    let file = Hashtbl.find fs.fids fid in
    let w = file.win in

    (match file.entry.code with
    | _ when w.deleted ->
      error fs req "window deleted"
    | File.File devid ->
      (* a worker thread (less: opti: arena of workers? *)
      Thread.create (fun () ->
       (* less: getclock? *)
       (try
         let dev = device_of_devid devid in
         let data = dev.D.read_threaded offset count w in
         answer fs { req with P9.typ = P9.R (R.Read data) }
       with Device.Error str ->
         error fs req str
      )) () |> ignore
    | File.Dir _ ->
      failwith "TODO: readdir"
    )
```

8.5.1 Reading a directory

8.6 Write

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 73b⟩+≡ (48e) ◁73a 74a▷

```
  (* Write *)
  | T.Write (fid, offset, data) ->
    check_fid "write" fid fs;
    let file = Hashtbl.find fs.fids fid in
    let w = file.win in

    (match file.entry.code with
    | _ when w.deleted ->
      error fs req "window deleted"
    | File.File devid ->
      (* a worker thread (less: opti: arena of workers? *)
      Thread.create (fun () ->
       (try
         let dev = device_of_devid devid in
         (* less: case where want to let device return different count? *)
         let count = String.length data in
```

```
        dev.D.write_threaded offset data w;
        answer fs { req with P9.typ = P9.R (R.Write count) }
      with Device.Error str ->
        error fs req str
    )) () |> ignore
  | File.Dir _ ->
    raise (Impossible "kernel should not call write on fid of a directory")
  )
```

8.7 Stats

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 74a⟩+≡ (48e) ◁73b 74b▷

```
  (* Stat *)
  | T.Stat (fid) ->
    check_fid "stat" fid fs;
    let file = Hashtbl.find fs.fids fid in
    let short = file.entry in
    (* less: getclock *)
    let clock = 0 in
    let dir_entry = {
      N.name = short.name;
      (* less: adjust version for snarf *)
      N.qid = file.qid;
      N.mode = (N.int_of_perm_property short.perm,
        match file.qid.N.typ with
        | N.QTDir -> N.DMDir
        | N.QTFile -> N.DMFile
      );
      N.length = 0; (*/* would be nice to do better */*)

      N.atime = clock;
      N.mtime = clock;

      N.uid = fs.user;
      N.gid = fs.user;
      N.muid = fs.user;

      N._typ = 0;
      N._dev = 0;
    }
    in
    answer fs { req with P9.typ = P9.R (R.Stat dir_entry) }
```

8.8 Forbidden operations

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 74b⟩+≡ (48e) ◁74a 74c▷

```
  (* Other *)
  | T.Create _
  | T.Remove _
  | T.Wstat _
      -> error fs req "permission denied"
```

⟨Threads_fileserver.dispatch() *match* request_typ *cases* 74c⟩+≡ (48e) ◁74b

```
  (* todo: handle those one? *)
  | T.Flush _
  | T.Auth _
```

```
->
failwith (spf "TODO: req = %s" (P9.str_of_msg req))
```

Chapter 9

Virtual Devices

9.1 /mnt/wsys/mouse

⟨*signature* Virtual_mouse.dev_mouse 76a⟩≡ (115c)
```
(* a virtual "/dev/mouse" *)
val dev_mouse: Device.t
```

⟨*constant* Virtual_mouse.dev_mouse 76b⟩≡ (115d)
```
let dev_mouse = { (*Device.default with*)
  name = "mouse";
  perm = Plan9.rw;

  ⟨method Virtual_mouse.dev_mouse.open_ 76c⟩
  ⟨method Virtual_mouse.dev_mouse.close 76d⟩
  ⟨method Virtual_mouse.dev_mouse.read_threaded 76e⟩
  ⟨method Virtual_mouse.dev_mouse.write_threaded 78b⟩
}
```

⟨*method* Virtual_mouse.dev_mouse.open_ 76c⟩≡ (76b)
```
open_ = (fun (w : Window.t) ->
  if w.mouse_opened
  then raise (Error "file in use");

  w.mouse_opened <- true;

  (* less: resized <- false? and race comment? *)
);
```

⟨*method* Virtual_mouse.dev_mouse.close 76d⟩≡ (76b)
```
close = (fun (w : Window.t) ->
  (* stricter? check that was opened indeed? *)

  w.mouse_opened <- false;

  (* todo: resized? Refresh message?*)
);
```

9.1.1 Reading part1

⟨*method* Virtual_mouse.dev_mouse.read_threaded 76e⟩≡ (76b)
```
(* a process is reading on its /dev/mouse; we must read from mouse
 * events coming to the window, events sent from the mouse thread.
 *)
read_threaded = (fun _offset count (w : Window.t) ->
```

```
    (* less: flushtag *)
    (* less: qlock active *)
    (* less: qlock unactive after answer? so need reorg this func? *)

    let chan = Event.receive w.chan_devmouse_read |> Event.sync in
    let m : Mouse.state    = Event.receive chan |> Event.sync in

    (* less: resize message *)

    let str =
      spf "%c%11d %11d %11d %11d "
        'm' m.pos.x m.pos.y (Mouse.int_of_buttons m.buttons) m.msec
    in
    (* bugfix: note that we do not honor_offset. /dev/mouse is a dynamic file *)
    Device.honor_count count str
  );
```

⟨Window.t *mouse fields* 77a⟩+≡ (32) ◁47j 77c▷

```
  (* Threads_window.thread --> Thread_fileserver.dispatch(Read).
   * The channel inside the channel will be used to write a mouse state
   * to thread_fileserver.
  *)
  chan_devmouse_read: Mouse.state Event.channel Event.channel;
```

⟨Threads_window.thread() *other* select *elements* 77b⟩≡ (45i) 79b▷

```
  (* sending *)
  (if w.mouse_counter <> w.last_count_sent
   then [Event.send w.chan_devmouse_read chan_devmouse
         |> wrap (fun () -> SentChannelForMouseRead)]
   else []
  ) @
```

⟨Window.t *mouse fields* 77c⟩+≡ (32) ◁77a

```
  mutable last_count_sent: mouse_counter;
```

⟨Threads_window.thread() *channels creation* 77d⟩≡ (45i) 79c▷

```
  let chan_devmouse = Event.new_channel () in
```

⟨Threads_window.event *other cases* 77e⟩≡ (45h) 79d▷

```
  (* producing for thread_fileserver(Read Qmouse) *)
  | SentChannelForMouseRead
```

⟨Threads_window.thread() *match* ev *other cases* 77f⟩≡ (45i) 79e▷

```
  | SentChannelForMouseRead ->
    mouse_out w chan_devmouse
```

⟨*function* Threads_window.mouse_out 77g⟩≡ (114a)

```
  let mouse_out (w : Window.t) (chan : Mouse.state Event.channel) =
    (*/* 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. */
    *)
    let m, counter =
      if Queue.length w.mouseclicks_queue > 0
      then Queue.take w.mouseclicks_queue
      else w.last_mouse, w.mouse_counter
    in
    (* we use last_count_sent later to know if we are ready to send mouse
     * states to someone
     *)
    w.last_count_sent <- counter;
    Event.send chan m |> Event.sync
```

⟨*function* `Device.honor_count` 78a⟩≡ (97a)
```
let honor_count count data =
  let len = String.length data in
  if len <= count
  then data
  else String.sub data 0 count
```

9.1.2 Writing

⟨*method* `Virtual_mouse.dev_mouse.write_threaded` 78b⟩≡ (76b)
```
write_threaded = (fun _offset _str _w ->
  failwith "TODO: virtual_mouse.write_threaded"
);
```

9.2 /mnt/wsys/cons

⟨*signature* `Virtual_cons.dev_cons` 78c⟩≡ (114b)
```
(* virtual "/dev/cons" *)
val dev_cons    : Device.t
```

⟨*constant* `Virtual_cons.dev_cons` 78d⟩≡ (114c)
```
let dev_cons = { Device.default with
  name = "cons";
  perm = Plan9.rw;

  ⟨method Virtual_cons.dev_cons.read_threaded 78e⟩
  ⟨method Virtual_cons.dev_cons.write_threaded 80b⟩
}
```

9.2.1 Reading part1

⟨*method* `Virtual_cons.dev_cons.read_threaded` 78e⟩≡ (78d)
```
(* a process is reading on its /dev/cons; we must read from key
 * events coming to the window (events sent from the keyboard thread).
 *)
read_threaded = (fun _offset count (w : Window.t) ->
  (* less: flushtag *)
  (* less: handle unicode partial runes *)

  let (chan_count_out, chan_bytes_in) =
    Event.receive w.chan_devcons_read |> Event.sync in

  Event.send chan_count_out count |> Event.sync;

  (* less: handle if flushing *)
  (* less: qlock active *)

  let bytes = Event.receive chan_bytes_in |> Event.sync in
  (* we asked for count so honor_count is redundant below (but defensive) *)
  Device.honor_count count bytes
);
```

⟨Window.t *keyboard fields* 79a⟩+≡ (32) ◁42c 80c▷
```
(* Threads_window.thread --> Thread_fileserver.dispatch(Read).
 * The first channel will be used by thread_fileserver to indicate the
 * number of bytes the process wants to read from its /dev/cons. The second
 * channel will be used to send the bytes to thread_fileserver.
 * Note that we send bytes, even though we read keys.
*)
chan_devcons_read: (int Event.channel * string Event.channel) Event.channel;
```

⟨Threads_window.thread() *other* select *elements* 79b⟩+≡ (45i) ◁77b 80e▷
```
(* less: npart *)
(if (w.raw_mode && Queue.length w.raw_keys > 0) ||
    (not w.raw_mode && Terminal.newline_after_output_point w.terminal)
 then [Event.send w.chan_devcons_read
         (chan_devcons_read_count, chan_devcons_read_bytes)
       |> wrap (fun () -> SentChannelsForConsRead)]
 else []
) @
```

⟨Threads_window.thread() *channels creation* 79c⟩+≡ (45i) ◁77d 80d▷
```
let chan_devcons_read_count = Event.new_channel () in
let chan_devcons_read_bytes = Event.new_channel () in
```

⟨Threads_window.event *other cases* 79d⟩+≡ (45h) ◁77e 80f▷
```
(* producing for thread_fileserver(Read Qcons) *)
| SentChannelsForConsRead
```

⟨Threads_window.thread() *match* ev *other cases* 79e⟩+≡ (45i) ◁77f 80g▷
```
| SentChannelsForConsRead ->
  bytes_out w (chan_devcons_read_count, chan_devcons_read_bytes)
```

⟨*function* Threads_window.bytes_out 79f⟩≡ (114a)
```
let bytes_out (w : Window.t) (chan_count, chan_bytes) =
  let cnt = Event.receive chan_count |> Event.sync in
  let buf = Bytes.create cnt in
  let i = ref 0 in

  (match w.raw_mode with
  ⟨Threads_window.bytes_out() match w.raw_mode cases 79g⟩
  );

  let str =
    if !i < cnt
    then Bytes.sub_string buf 0 !i
    else Bytes.to_string buf
  in
  Event.send chan_bytes str |> Event.sync
```

⟨Threads_window.bytes_out() *match* w.raw_mode *cases* 79g⟩≡ (79f) 80a▷
```
| true ->
  while !i < cnt && Queue.length w.raw_keys > 0 do
    Bytes.set buf !i (Queue.take w.raw_keys);
    incr i;
  done
```

⟨Threads_window.bytes_out() *match* w.raw_mode *cases* 80a⟩+≡ (79f) ◁79g
```
  | false ->
    let term : Terminal.t = w.terminal in
    (* "When newline, chars between output point and newline are sent."*)
    while !i < cnt && term.output_point.i < term.nrunes do
      let pos = term.output_point.i in
      Bytes.set buf !i term.text.(pos);
      term.output_point <- { Terminal.i = pos + 1};
      incr i;
    done
```

9.2.2 Writing part1

⟨*method* Virtual_cons.dev_cons.write_threaded 80b⟩≡ (78d)
```
  (* a process is writing on its /dev/cons; it wants to output strings
   * on the terminal
   *)
  write_threaded = (fun _offset str (w : Window.t) ->
    (* todo: partial runes *)

    let runes = Rune.bytes_to_runes str in
    (* less: flushtag *)

    let chan_runes_out = Event.receive w.chan_devcons_write |> Event.sync in

    (* less: handle if flushing *)
    (* less: qlock active *)

    Event.send chan_runes_out runes |> Event.sync;
  );
```

⟨Window.t *keyboard fields* 80c⟩+≡ (32) ◁79a
```
  (* Threads_window.thread --> Thread_fileserver.dispatch(Write).
   * Note that we send full runes, not bytes.
   * The channel inside will be used to read from thread_fileserver(Write)
   * the data the process wrote to its /dev/cons.
   *)
  chan_devcons_write: (Rune.t list Event.channel) Event.channel;
```

⟨Threads_window.thread() *channels creation* 80d⟩+≡ (45i) ◁79c
```
  let chan_devcons_write_runes = Event.new_channel () in
```

⟨Threads_window.thread() *other* select *elements* 80e⟩+≡ (45i) ◁79b
```
  (* less: auto_scroll, mouseopen??
   * todo: qh vs org and nchars *)
  (if true
   then [Event.send w.chan_devcons_write chan_devcons_write_runes
         |> wrap (fun () -> SentChannelForConsWrite);]
   else []
  ) @
```

⟨Threads_window.event *other cases* 80f⟩+≡ (45h) ◁79d
```
  (* producing and then consuming for thread_fileserver(Write Qcons) *)
  | SentChannelForConsWrite
```

⟨Threads_window.thread() *match* ev *other cases* 80g⟩+≡ (45i) ◁79e
```
  | SentChannelForConsWrite ->
    runes_in w chan_devcons_write_runes
```

⟨*function* Threads_window.runes_in 81a⟩≡ (114a)
```
(* output from application *)
let runes_in (w: Window.t) (chan : Rune.t list Event.channel) =
  let runes = Event.receive chan |> Event.sync in
  Terminal.runes_in w.terminal runes
```

⟨*signature* Terminal.runes_in 81b⟩≡ (101d)
```
val runes_in: t -> Rune.t list -> unit
```

9.3 /mnt/wsys/consctl

⟨*signature* Virtual_cons.dev_consctl 81c⟩≡ (114b)
```
(* virtual "/dev/consctl" *)
val dev_consctl : Device.t
```

⟨*constant* Virtual_cons.dev_consctl 81d⟩≡ (114c)
```
let dev_consctl = { Device.default with
  name = "consctl";
  perm = Plan9.w;

  ⟨method Virtual_cons.dev_consctl.open_ 81f⟩
  ⟨method Virtual_cons.dev_consctl.close 81g⟩
  ⟨method Virtual_cons.dev_consctl.write_threaded 81h⟩
}
```

⟨Window.t *graphical window fields* 81e⟩+≡ (32) ◁34j
```
(* can also be used in textual windows, but more rare *)
mutable consctl_opened: bool;
```

⟨*method* Virtual_cons.dev_consctl.open_ 81f⟩≡ (81d)
```
open_ = (fun (w : Window.t) ->
  if w.consctl_opened
  then raise (Error "file in use");

  w.consctl_opened <- true;
);
```

⟨*method* Virtual_cons.dev_consctl.close 81g⟩≡ (81d)
```
close = (fun (w : Window.t) ->
  (* less: if holding *)
  if w.raw_mode
  then begin
    w.raw_mode <- false;
    (* less: send RawOff? but nop in Threads_window anyway *)
  end;
  w.consctl_opened <- false;
);
```

⟨*method* Virtual_cons.dev_consctl.write_threaded 81h⟩≡ (81d)
```
write_threaded = (fun _offset str (w : Window.t) ->
  match str with
  | "rawon" ->
    (* less: holding *)
    (* stricter: set to bool, not increment, so no support
     *  for multiple rawon *)
    (* stricter? exn if already on? *)
    w.raw_mode <- true;
    (* less: send RawOn? *)
```

```
    | "rawoff" ->
      (* todo: send RawOff so terminal can process remaining queued raw keys *)
      w.raw_mode <- false;

    | "holdon" ->
      failwith ("TODO: holdon")
    | "holdoff" ->
      failwith ("TODO: holdoff")

    | _ ->
      raise (Error (spf "unknown control message: %s" str));
  );
```

9.4 /mnt/wsys/cursor

⟨Window.t *graphics fields* 82a⟩+≡ (32) ◁34b
```
  (* writable through /mnt/wsys/cursor *)
  mutable mouse_cursor: Cursor.t option;
```

⟨*signature* Virtual_mouse.dev_cursor 82b⟩≡ (115c)
```
  (* a virtual "/dev/cursor" *)
  (* val dev_cursor: Device.t *)
```

⟨*constant* Virtual_mouse._dev_cursor 82c⟩≡ (115d)
```
  let _dev_cursor = { (*Device.default with*)
    name = "cursor";
    perm = Plan9.rw;

    open_ = (fun _w ->
      raise Todo
    );
    close = (fun _w ->
      raise Todo
    );
    read_threaded = (fun _offset _count _w  ->
      raise Todo
    );
    write_threaded = (fun _offset _str _w ->
      raise Todo
    );
  }
```

9.5 /dev/draw/ and /mnt/wsys/winname

⟨*signature* Virtual_draw.dev_winname 82d⟩≡ (115a)
```
  (* virtual "/dev/winname" *)
  val dev_winname: Device.t
```

⟨*constant* Virtual_draw.dev_winname 82e⟩≡ (115b)
```
  let dev_winname = { Device.default with
    name = "winname";
    perm = Plan9.r;

    read_threaded = (fun offset count (w : Window.t) ->
      let str = w.winname in
```

```
    if str = ""
    then raise (Error "window has no name");
    Device.honor_offset_and_count offset count str
  );
}
```

⟨*function* `Device.honor_offset_and_count` 83⟩≡ (97a)

```
let honor_offset_and_count offset count data =
  let len = String.length data in
  match () with
  | _ when offset > len -> ""
  | _ when offset + count > len ->
    String.sub data offset (len - offset)
  | _ -> data
```

Chapter 10

Graphical Windows

10.1 Graphical window setup

10.1.1 `initdraw()`

10.1.2 `initmouse()`, mouse-open mode

10.1.3 `initkeyboard()`, raw-access mode

10.2 Mouse events

10.2.1 Mouse state queue

10.2.2 `/mnt/wsys/mouse` reading part2

10.3 Keyboard events

10.3.1 Raw keys queue

10.3.2 `/mnt/wsys/cons` reading part2

10.4 Resize events

10.5 `/mnt/wsys/window`

Chapter 11

Textual Windows

11.1 Textual window creation

11.1.1 Scrollbar

11.1.2 Frame

11.2 Frame widget

11.2.1 `Frame`

11.2.2 `frinit()`

11.2.3 Frame colors

11.2.4 Frame tick

11.2.5 Frame boxes

11.2.6 Frame strings

11.2.7 Frame rune position, point, and box number

11.2.8 Frame selection

11.3 Content modification

11.3.1 `winsert()`

11.3.2 Growing array

11.4 Content rendering

11.4.1 `frinsert()`

11.4.2 `frdelete()`

11.4.3 `wshow()`

11.4.4 Drawing the scrollbar: `wscrdraw()`

11.4.5 Drawing the tick: `frtick()`

```
  mutable auto_scroll: bool;
```

11.9 Scroll bar interaction

11.10 Resize

11.11 /mnt/wsys/text

⟨*signature* Dev_textual_window.dev_text 87a⟩≡ (97d)

```
  val dev_text: Device.t
```

⟨*constant* Dev_textual_window.dev_text 87b⟩≡ (98a)

```
  let dev_text = { Device.default with
    name = "text";
    perm = Plan9.r;
    read_threaded = (fun offset count w ->
      let term = w.W.terminal in
      let str = Bytes.create term.T.nrunes in
      for i = 0 to term.T.nrunes - 1 do
        Bytes.set str i term.T.text.(i);
      done;
      Device.honor_offset_and_count offset count (Bytes.to_string str)
    );
  }
```

Chapter 12

Windowing System Files

12.1 /mnt/wsys/winid

⟨*signature* Dev_wm.dev_winid 88a⟩≡ (98b)
```
val dev_winid: Device.t
```

⟨*constant* Dev_wm.dev_winid 88b⟩≡ (98c)
```
let dev_winid = { Device.default with
  name = "winid";
  perm = Plan9.r;
  read_threaded = (fun offset count (w : Window.t) ->
    let str = spf "%11d" w.id in
    Device.honor_offset_and_count offset count str
  );
}
```

12.2 /mnt/wsys/label

12.3 /mnt/wsys/screen

12.4 /mnt/wsys/wsys/

12.5 /mnt/wsys/wctl

12.5.1 Reading

12.5.2 Writing (controlling windows)

Chapter 13

Advanced Topics TODO

13.1 External mount: `/srv/rio.user.pid`

13.2 Command-line control: `/srv/riowctl.user.pid`

13.3 Recursive `rio`

13.4 Advanced terminal editing

13.4.1 Snarf

13.4.2 Plumb

13.4.3 Auto complete

13.4.4 Word selection

13.5 Automatic scrolling: `rio -s`

⟨CLI.main() options *elements* 89a⟩≡ (39e) 89b▷
```
  "-s", Arg.Unit (fun () -> raise Todo),
  " ";
```

13.6 Initial command: `rio -i`

⟨CLI.main() options *elements* 89b⟩+≡ (39e) ◁89a 89c▷
```
  "-i", Arg.String (fun _s -> raise Todo),
  " <initcmd>";
```

13.7 Fake keyboard input: `rio -k`

⟨CLI.main() options *elements* 89c⟩+≡ (39e) ◁89b 90▷
```
  "-k", Arg.String (fun _s -> raise Todo),
  " <kbdcmd>";
```

13.7.1 `rio -k`

13.7.2 `wkeyboard`

13.7.3 `/mnt/wsys/kdbin`

13.7.4 Keyboard hide

13.8 Font selection: `rio -f`

⟨`CLI.main() options` *elements* 90⟩+≡ (39e) ◁89c 92d▷

```
"-font", Arg.String (fun _s -> raise Todo),
" <fontname>";
```

13.9 Holding mode

13.10 Signals, notes

13.11 Timer

13.12 Flushing

13.13 Security

13.14 TODO `Wakeup`

13.15 TODO `Refresh`

Chapter 14

Conclusion

Appendix A

Debugging

A.1 Logging

⟨CLI.main() *locals* 92a⟩≡ (39e) 93a▷
```
let level = ref (Some Logs.Warning) in
```

⟨CLI.main() *logging setup* 92b⟩≡ (39e)
```
Logs_.setup !level ();
Logs.info (fun m -> m "rio ran from %s" (Sys.getcwd()));
```

⟨*constant* Globals.debug_9P 92c⟩≡ (99c)
```
let debug_9P = ref false
```

⟨CLI.main() options *elements* 92d⟩+≡ (39e) ◁90 92h▷
```
(* pad: not in original *)
"-debug_9P", Arg.Set Globals.debug_9P,
" ";
```

⟨Threads_fileserver.thread *in loop, debug* 92e⟩≡ (48c)
```
(* todo: should exit the whole proc if error *)
if !Globals.debug_9P
then Logs.debug (fun m -> m "%s" (P9.str_of_msg req));
```

⟨Threads_fileserver.answer() *debug* 92f⟩≡ (49b)
```
if !Globals.debug_9P
then Logs.debug (fun m -> m "%s" (P9.str_of_msg res));
```

⟨*constant* Globals.debug_draw 92g⟩≡ (99c)
```
let debug_draw = ref false
```

⟨CLI.main() options *elements* 92h⟩+≡ (39e) ◁92d 92j▷
```
"-debug_draw", Arg.Set Globals.debug_draw,
" ";
```

⟨CLI.thread_main() *graphics initializations, debug* 92i⟩≡ (40c)
```
if !Globals.debug_draw
then Display.debug display;
```

A.2 Testing

⟨CLI.main() options *elements* 92j⟩+≡ (39e) ◁92h
```
"-test", Arg.Unit (fun () -> Test.test ()),
" ";
```

Appendix B

Error Management

⟨CLI.main() *locals* 93a⟩+≡ (39e) ◁92a
```
let backtrace = ref false in
```

⟨CLI.main() *handle* exn 93b⟩≡ (39e)
```
if !backtrace
then raise exn
else
  (match exn with
  | Failure s ->
        Logs.err (fun m -> m "%s" s);
        Exit.Code 1
  | _ -> raise exn
  )
```

⟨*function* Threads_fileserver.error 93c⟩≡ (111b)
```
let error fs req str =
  let res = { req with P9.typ = P9.R (R.Error str) } in
  answer fs res
```

⟨*exception* Device.Error 93d⟩≡ (97a)
```
(* This will be catched up by thread_fileserver to transform the
 * exception in an Rerror 9P response.
 *)
exception Error of string
```

Appendix C

Examples of Windowing System Applications TODO

Appendix D

Extra Code

D.1 CLI.mli

⟨CLI.mli 95a⟩≡
```
⟨type CLI.caps 39b⟩

⟨signature CLI.main 39c⟩
⟨signature CLI.thread_main 40a⟩
```

D.2 CLI.ml

⟨CLI.ml 95b⟩≡
```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of rio, the Plan 9 windowing system.
 *
 * Main limitations compared to rio:
 *  - no unicode
 *  - just a basic ASCII font for now
 *  - the terminal does not have many features
 *
 * todo:
 *  - more fonts
 *  - unicode
 *  - need to disable preempt?
 *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)
⟨type CLI.caps 39b⟩

⟨constant CLI.usage 39d⟩

(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)
⟨function CLI.thread_main 40b⟩
```

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* CLI.main 39e⟩

D.3 Cursors.mli

⟨Cursors.mli 96a⟩≡

⟨*signature* Cursors.crosscursor 50a⟩
⟨*signature* Cursors.boxcursor 50b⟩
⟨*signature* Cursors.sightcursor 50c⟩

```
(*
val whitearrow: Cursor.t
val query: Cursor.t
*)
```

⟨*signature* Cursors.which_corner_cursor 54f⟩

D.4 Cursors.ml

⟨Cursors.ml 96b⟩≡

```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common

open Cursor
open Point
open Rectangle

(*****************************************************************************)
(* Classic cursors *)
(*****************************************************************************)
(* See also 'arrow' defined in the kernel and in lib_graphics/input/cursor.ml *)
```

⟨*constant* Cursors.crosscursor 50d⟩
⟨*constant* Cursors.boxcursor 50e⟩
⟨*constant* Cursors.sightcursor 51a⟩

⟨*constant* Cursors._whitearrow 51b⟩
⟨*constant* Cursors._query 51c⟩

```
(*****************************************************************************)
(* Border and corner cursors *)
(*****************************************************************************)
```

⟨*type* Cursors.corner 52a⟩
⟨*type* Cursors.portion 55b⟩

⟨*function* Cursors.portion 55d⟩

⟨*function* Cursors.which_corner 55c⟩

⟨*constant* Cursors.tl 52b⟩
⟨*constant* Cursors.t 52c⟩
⟨*constant* Cursors.tr 52d⟩

⟨*constant* `Cursors.r` 53a⟩
⟨*constant* `Cursors.br` 53b⟩
⟨*constant* `Cursors.b` 53c⟩
⟨*constant* `Cursors.bl` 53d⟩
⟨*constant* `Cursors.l` 54a⟩

⟨*function* `Cursors.which_corner_cursor` 55a⟩

D.5 Device.ml

⟨`Device.ml` 97a⟩≡
```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Helpers to build "virtual devices" (e.g., a virtual /dev/cons) *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)
```
⟨*type* `Device.t` 38c⟩
⟨*constant* `Device.default` 38d⟩

⟨*exception* `Device.Error` 93d⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```
⟨*function* `Device.honor_offset_and_count` 83⟩
⟨*function* `Device.honor_count` 78a⟩

D.6 Dev_graphical_window.mli

⟨`Dev_graphical_window.mli` 97b⟩≡

D.7 Dev_graphical_window.ml

⟨`Dev_graphical_window.ml` 97c⟩≡

D.8 Dev_textual_window.mli

⟨`Dev_textual_window.mli` 97d⟩≡
⟨*signature* `Dev_textual_window.dev_text` 87a⟩

D.9 Dev_textual_window.ml

⟨Dev_textual_window.ml 98a⟩≡
```
open Common

open Device
module W = Window
module T = Terminal

⟨constant Dev_textual_window.dev_text 87b⟩
```

D.10 Dev_wm.mli

⟨Dev_wm.mli 98b⟩≡
```
⟨signature Dev_wm.dev_winid 88a⟩
```

D.11 Dev_wm.ml

⟨Dev_wm.ml 98c⟩≡
```
open Common

open Device

⟨constant Dev_wm.dev_winid 88b⟩
```

D.12 File.ml

⟨File.ml 98d⟩≡
```
(* Copyright 2017-2027 Yoann Padioleau, see copyright.txt *)
open Common

open Plan9 (* for the fields *)
module N = Plan9

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* ?? *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)

⟨type File.fid 36f⟩

⟨type File.filecode 37c⟩
⟨type File.dir 37d⟩
    (* less: '/wsys/' *)
  (* '/xxx' *)
⟨type File.devid 37g⟩

⟨type File.fileid 70a⟩

⟨type File.dir_entry_short 37b⟩
```

⟨*constant* File.root_entry 37e⟩

⟨*type* File.t 36g⟩

```
(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
```
⟨*function* File.int_of_filecode 70d⟩
⟨*function* File.int_of_fileid 70c⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* File.qid_of_fileid 70b⟩

D.13 Fileserver.mli

⟨Fileserver.mli 99a⟩≡
```
(* ?? *)
```

⟨*type* Fileserver.t 36e⟩

⟨*signature* Fileserver.init 41c⟩

D.14 Fileserver.ml

⟨Fileserver.ml 99b⟩≡
```
open Common

module Unix1 = Unix
module Unix2 = ThreadUnix

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)

(* Note that pipes created under plan9 are bidirectional!
 * No need to create 2 pipes for 2-way communication.
 *)
```

⟨*type* Fileserver.t 36e⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Fileserver.init 41d⟩

D.15 Globals.ml

⟨Globals.ml 99c⟩≡

```
(* Copyright 2017, 2025 Yoann Padioleau, see copyright.txt *)
open Common

module W = Window

⟨global Globals.windows 34c⟩
⟨global Globals.hidden 67e⟩

⟨global Globals.current 34g⟩

⟨function Globals.win 34h⟩

⟨function Globals.window_at_point 44e⟩

⟨constant Globals.debug_9P 92c⟩
⟨constant Globals.debug_draw 92g⟩

(* alt: could be in global but better to pass it around so more functional
 * mousectl
 * kbdctl
 * fs
 * let display = ref Display.fake_display
 * let view = ref Display.fake_image
 * let font = ref Font.fake_font
 * let desktop = ref Baselayer.fake_baselayer
 *)

(* this is just too annoying to pass around *)
⟨constant Globals.red 40d⟩
⟨constant Globals.title_color 40e⟩
⟨constant Globals.title_color_light 41a⟩
```

D.16 Main.ml

```
⟨Main.ml 100a⟩≡
  (* Copyright 2025-2027 Yoann Padioleau, see copyright.txt *)
  (* open Xix_windows *)

  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)
  ⟨toplevel Main._1 39a⟩
```

D.17 Mouse_action.mli

```
⟨Mouse_action.mli 100b⟩≡
  ⟨signature Mouse_action.sweep 62c⟩
  ⟨signature Mouse_action.point_to 65e⟩
```

D.18 Mouse_action.ml

```
⟨Mouse_action.ml 100c⟩≡
  open Common
```

```
(* less: maybe the 'ref m' approach of point_to() is simpler for sweep() too *)
⟨type Mouse_action.sweep_state 62d⟩

⟨function Mouse_action.sweep 62f⟩

⟨function Mouse_action.point_to 65f⟩
```

D.19 Processes_winshell.mli

```
⟨Processes_winshell.mli 101a⟩≡
  ⟨signature Processes_winshell.run_cmd_in_window_in_child_of_fork 60b⟩
```

D.20 Processes_winshell.ml

```
⟨Processes_winshell.ml 101b⟩≡
  (* Copyright 2017, 2025 Yoann Padioleau, see copyright.txt *)
  open Common
  open Fpath_.Operators

  module Unix1 = Unix
  module Unix2 = ThreadUnix

  ⟨function Processes_winshell.run_cmd_in_window_in_child_of_fork 60c⟩
```

D.21 Terminal.mli

```
⟨signature Terminal.newline_after_output_point 101c⟩≡ (101d)
  val newline_after_output_point: t -> bool
```

```
⟨Terminal.mli 101d⟩≡
  ⟨type Terminal.position 35d⟩

  ⟨type Terminal.t 35a⟩

  ⟨signature Terminal.alloc 60a⟩

  ⟨signature Terminal.key_in 47b⟩
  ⟨signature Terminal.runes_in 81b⟩

  ⟨signature Terminal.newline_after_output_point 101c⟩

  ⟨signature Terminal.repaint 62b⟩
```

D.22 Terminal.ml

```
⟨type Terminal.colors 101e⟩≡ (109b)
  type colors = {
    mutable background             : Image.t;
    mutable border                 : Image.t;
    mutable text_color             : Image.t;

    mutable background_highlighted : Image.t;
    mutable text_highlighted       : Image.t;
  }
```

⟨*constant* Terminal.default_colors 102a⟩≡ (109b)
```
let default_colors = {
  background             = Display.fake_image;
  border                 = Display.fake_image;
  text_color             = Display.fake_image;
  background_highlighted = Display.fake_image;
  text_highlighted       = Display.fake_image;
}
```

⟨*constant* Terminal.dark_grey 102b⟩≡ (109b)
```
let dark_grey = ref Display.fake_image
```

⟨*constant* Terminal.scrollbar_width 102c⟩≡ (109b)
```
let scrollbar_width = 12
```

⟨*constant* Terminal.scrollbar_gap 102d⟩≡ (109b)
```
(* gap right of scrollbar *)
let scrollbar_gap = 4
```

⟨*constant* Terminal.tick_width 102e⟩≡ (109b)
```
let tick_width = 3
```

⟨*constant* Terminal.scrollbar_img 102f⟩≡ (109b)
```
let scrollbar_img = ref None
```

⟨*constant* Terminal.tick_img 102g⟩≡ (109b)
```
let tick_img = ref None
```

⟨*constant* Terminal.debug_keys_flag 102h⟩≡ (109b)
```
let debug_keys_flag = ref false
```

⟨*constant* Terminal.debug_keys_pt 102i⟩≡ (109b)
```
let debug_keys_pt = ref Point.zero
```

⟨*function* Terminal.debug_keys 102j⟩≡ (109b)
```
let debug_keys term key =
  if !debug_keys_flag
  then begin
    let display = term.img.I.display in
    (* todo: why cant use display.image? does not draw ... cos desktop? *)
    let img = term.img in
    if !debug_keys_pt = Point.zero
    then debug_keys_pt := Point.sub img.I.r.max (Point.p 200 15);
    debug_keys_pt :=
      Text.string term.img !debug_keys_pt display.D.black Point.zero term.font
      (spf "%X" (Char.code key))
  end
```

⟨*function* Terminal.init_colors 102k⟩≡ (109b)
```
let init_colors display =
  if default_colors.background == Display.fake_image
  then begin
    default_colors.background <- display.D.white;
    default_colors.background_highlighted <-
      Image.alloc_color display (Color.mk2 0xCC 0xCC 0xCC);
    default_colors.border <-
      Image.alloc_color display (Color.mk2 0x99 0x99 0x99);
    default_colors.text_color <- display.D.black;
    default_colors.text_highlighted <- display.D.black;

    (*/* greys are multiples of 0x11111100+0xFF, 14* being palest */*)
    dark_grey := Image.alloc_color display (Color.mk2 0x66 0x66 0x66);
  end
```

⟨*function* Terminal.colors_focused_window 103a⟩≡ (109b)
```
let colors_focused_window () =
  default_colors
```

⟨*function* Terminal.colors_unfocused_window 103b⟩≡ (109b)
```
let colors_unfocused_window () =
  { default_colors with text_color = !dark_grey; text_highlighted = !dark_grey}
```

⟨*function* Terminal.scroll_pos 103c⟩≡ (109b)
```
let scroll_pos r p0 p1 total =
  if total = 0
  then r
  else
    let h = Rectangle.dy r in
    let miny = r.min.y +
      if p0.i > 0
      then p0.i * h / total
      else 0
    in
    let maxy = r.max.y -
      if p1.i < total
      then (total - p1.i) * h / total
      else 0
    in
    let maxy, miny =
      if maxy < miny + 2
      then
        if miny+2 <= r.max.y
        then miny+2, miny
        else maxy, maxy-2
      else maxy, miny
    in
    Rectangle.r r.min.x miny r.max.x maxy
```

⟨*function* Terminal.repaint_scrollbar 103d⟩≡ (109b)
```
let repaint_scrollbar term =
  let r = term.scrollr in

  let v = Point.p (Rectangle.dx r) 0 in
  let r1 = Rectangle.sub_pt v r in
  let r2 = scroll_pos r1
    term.origin_visible
    { i = term.origin_visible.i + term.runes_visible }
    term.nrunes
  in
  let r2 = Rectangle.sub_pt r1.min r2 in
  let r1 = Rectangle.sub_pt r1.min r1 in

  let img = Fun_.once scrollbar_img (fun () ->
    let display = term.img.I.display in
    let view = display.D.image in
    (* /*factor by which window dimension can exceed screen*/ *)
    let big = 3 in
    let h = big * Rectangle.dy view.I.r in
    let r = Rectangle.r 0 0 32 h in
    Image.alloc display r view.I.chans false Color.white
  )
  in
  Draw.draw_color img r1 default_colors.border;
  Draw.draw_color img r2 default_colors.background;
```

```
    let r3 = { r2 with min = { r2.min with x = r2.max.x - 1 } } in
    Draw.draw_color img r3 default_colors.border;

    Draw.draw term.img term.scrollr img None (Point.p 0 r1.min.y)
```

⟨*function* Terminal.visible_lines 104a⟩≡ (109b)

```
  let visible_lines term =
    let maxlines = Rectangle.dy term.textr / term.font.Font.height in

    let rec aux acc_lines acc_str nblines_done p =
      if p.i >= term.nrunes || nblines_done >= maxlines
      then Rune.string_of_runes (List.rev acc_str)::acc_lines, p
      else
        match term.text.(p.i) with
        | '\n' ->
          aux ((Rune.string_of_runes (List.rev acc_str))::acc_lines) []
            (nblines_done + 1) { i = p.i + 1 }
        | c ->
          aux acc_lines (c::acc_str) nblines_done { i = p.i + 1 }
    in
    let xs, lastp = aux [] [] 0 term.origin_visible in
    List.rev xs, lastp
```

⟨*function* Terminal.repaint_content 104b⟩≡ (109b)

```
  let repaint_content term colors =
    let xs, lastp = visible_lines term in
    term.runes_visible <- lastp.i - term.origin_visible.i;

    let p = ref term.textr.min in
    xs |> List.iter (fun s ->
      let _endpt =
        Text.string term.img !p colors.text_color Point.zero term.font s
      in
      p := { term.textr.min with y = !p.y + term.font.Font.height };
    )
```

⟨*function* Terminal.newline_after_output_point 104c⟩≡ (109b)

```
  (* "When newline, chars between output point and newline are sent."*)
  let newline_after_output_point term =
    (* todo: more elegant way? way to iter over array and stop until cond? *)
    let rec aux p =
      if p.i < term.nrunes
      then
        let c = term.text.(p.i) in
        if c = '\n'
        then true
        else aux { i = p.i + 1 }
      else false
    in
    aux term.output_point
```

⟨*function* Terminal.move_origin_to_see 104d⟩≡ (109b)

```
  (* assumes term.runes_visible is up to date, so
   * !!do not call this function if you modified term.textr since the last
   * repaint_content!!
   *)
  let move_origin_to_see term pos =
    if pos.i >= term.origin_visible.i &&
       pos.i <= term.origin_visible.i + term.runes_visible
    then ()
    else failwith "TODO: move_origin_to_see out of range"
```

⟨*function* Terminal.point_of_position 105a⟩≡ (109b)

```
let point_of_position term pos =
  if pos.i >= term.origin_visible.i &&
     pos.i <=  term.origin_visible.i + term.runes_visible
  then
    (* some of the logic below is similar in visible_lines *)
    let rec aux pt current_pos =
      if current_pos = pos
      then pt
      else
        let next_pos = { i = current_pos.i + 1 } in
        match term.text.(current_pos.i) with
        | '\n' ->
          aux (Point.p term.textr.min.x (pt.y + term.font.Font.height)) next_pos
        | c ->
          let width = Text.string_width term.font (spf "%c" c) in
          aux { pt with x = pt.x + width } next_pos
    in
    aux term.textr.min term.origin_visible
  else
  (* anything out of textr  *)
  term.textr.max
```

⟨*function* Terminal.repaint_tick 105b⟩≡ (109b)

```
let repaint_tick term colors =
  let img = Fun_.once tick_img (fun () ->
    let display = term.img.I.display in
    let view = display.D.image in
    let font = term.font in
    let height = font.Font.height in

    let r = Rectangle.r 0 0 tick_width height in
    let img = Image.alloc display r view.I.chans false Color.white in
    (*/* background color */*)
    Draw.draw_color img r colors.background;
    (*/* vertical line */*)
    Draw.draw_color img
      (Rectangle.r (tick_width / 2) 0 (tick_width / 2 + 1) height)
      colors.text_color;
    (*/* box on each end */*)
    Draw.draw_color img (Rectangle.r 0 0 tick_width tick_width)
      colors.text_color;
    Draw.draw_color img (Rectangle.r 0 (height - tick_width) tick_width height)
      colors.text_color;
    img
  )
  in
  let pt = point_of_position term term.cursor in
  if Rectangle.pt_in_rect pt term.textr then begin
    (*/* looks best just left of where requested */*)
    let pt = { pt with x = pt.x - 1 } in
    (*/* can go into left border but not right */*)
    let maxx = min (pt.x + tick_width) term.textr.max.x in
    let r = Rectangle.r pt.x pt.y
                        maxx (pt.y + term.font.Font.height)
    in
    Draw.draw term.img r img None Point.zero
  end
```

⟨*function* Terminal.alloc 105c⟩≡ (109b)

```
  let alloc (img : Image.t) (font : Font.t) : t =
    init_colors img.I.display;
    let r =
      Rectangle.insetrect (Draw_rio.window_border_size + 1) img.I.r in
    let scrollr =
      { r with max = { r.max with x = r.min.x + scrollbar_width } } in
    let textr =
      { min = { r.min with x = scrollr.max.x + scrollbar_gap };
        (* can not use last lines if not enough pixels for a character *)
        max = { r.max with y = r.max.y - (Rectangle.dy r mod font.Font.height) };
      } in

    {
      text = [||];
      nrunes = 0;

      cursor = zero;
      end_selection = None;
      output_point = zero;

      origin_visible = zero;
      runes_visible = 0;

      font = font;
      img = img;
      r = r;
      textr = textr;
      scrollr = scrollr;
      is_selected = true;
    }
```

⟨*function* Terminal.insert_runes 106⟩≡ (109b)

```
  (* less: return pos? used only when delete lines in term.text? *)
  let insert_runes term pos runes =
    let n = List.length runes in
    (* stricter: *)
    if n = 0
    then failwith "insert_runes: empty runes";

    (* grow array if necessary *)
    if term.nrunes + n > Array.length term.text then begin
      let old = term.text in
      let oldlen = Array.length old in
      (* todo: high_water, low_water, etc *)
      let increment = 20000 in
      term.text <- Array.make (oldlen + increment) '*';
      Array.blit old 0 term.text 0 oldlen;
    end;

    assert (pos.i <= term.nrunes);
    (* move to the right the runes after the cursor pos to make some space *)
    Array.blit term.text pos.i term.text (pos.i + n) (term.nrunes - pos.i);
    (* fill the space *)
    runes |> List.iteri (fun i rune ->
      term.text.(pos.i + i) <- rune;
    );
    term.nrunes <- term.nrunes + n;

    (* adjust cursors *)
    if pos.i <= term.cursor.i
```

```
    then term.cursor <- { i = term.cursor.i + n };
    term.end_selection |> Option.iter (fun ends ->
      if pos.i <= ends.i
      then term.end_selection <- Some ({ i = ends.i + n});
    );
    (* note the '<' here, not '<=' ! *)
    if pos.i < term.output_point.i
    then term.output_point <- { i = term.output_point.i + n };
    if pos.i < term.origin_visible.i
    then term.origin_visible <- { i = term.origin_visible.i + n };
    ()
```

⟨*function* Terminal.delete_runes 107a⟩≡ (109b)

```
  let delete_runes term pos n =
    (* stricter: *)
    if n = 0
    then failwith "delete_runes: empty delete";
    let pos2 = { i = pos.i + n } in

    Array.blit term.text pos2.i term.text pos.i (term.nrunes - pos2.i);
    term.nrunes <- term.nrunes - n;

    (* adjust cursors *)
    if pos.i < term.cursor.i
    then term.cursor <- { i = term.cursor.i - min n (term.cursor.i - pos.i) };
    term.end_selection |> Option.iter (fun ends ->
      if pos.i < ends.i
      then term.end_selection <- Some ({ i = ends.i - min n (ends.i - pos.i)});
    );

    (match () with
    | _ when term.output_point.i > pos2.i ->
      term.output_point <- { i = term.output_point.i - n };
    | _ when term.output_point.i > pos.i ->
      term.output_point <- pos;
    | _ -> assert(term.output_point.i <= pos.i);
      ()
    );

    if pos2.i <= term.origin_visible.i
    then term.origin_visible <- { i = term.origin_visible.i -n };
    ()
```

⟨*function* Terminal.repaint 107b⟩≡ (109b)

```
  let repaint term =
    let colors =
      if term.is_selected
      then colors_focused_window ()
      else colors_unfocused_window ()
    in

    (* start from scratch with blank image, otherwise tick will be
     * paint all over the place.
     * alt: save what was under the tick image.
     *)
    if not !debug_keys_flag
    then Draw.draw_color term.img term.r colors.background;

    (* repaint_content() needs to be before repaint_scrollbar() because it
     * updates term.runes_visible used by repaint_scrollbar().
```

```
     *)
    repaint_content term colors;
    repaint_scrollbar term;
    repaint_tick term colors;
    ()
```

⟨*constant* `Terminal.previous_code` 108a⟩≡ (109b)
```
  let previous_code = ref 0
```

⟨*function* `Terminal.key_in` 108b⟩≡ (109b)
```
  let key_in term key =
    (match Char.code key with

    (* less: navigation keys are handled in the caller in rio-C, to allow keys
     * navigation even in raw mode, but it's more cohesive to put such
     * code here.
     *)

    (* Left arrow *)
    | 0x91 when !previous_code = 0x80 (* less: and previous_previous = 0xEF *) ->
      let cursor = term.cursor in
      if cursor.i > 0
      then term.cursor <- { i = cursor.i - 1 };
      move_origin_to_see term term.cursor

    (* Right arrow *)
    | 0x92 when !previous_code = 0x80 ->
      (* less: use end_position cursor instead? *)
      let cursor = term.cursor in
      if cursor.i < term.nrunes
      then term.cursor <- { i = cursor.i + 1 };
      move_origin_to_see term term.cursor

    (* End of line ^E *)
    | 0x5 ->
      (* alt: have term.newlines as in Efuns so easier *)
      let rec find_eol_or_end pos =
        if pos.i = term.nrunes || term.text.(pos.i) = '\n'
        then pos
        else find_eol_or_end { i = pos.i + 1 }
      in
      term.cursor <- find_eol_or_end term.cursor;
      move_origin_to_see term term.cursor
    (* Beginning of line ^A *)
    | 0x1 ->
      (* alt: have term.newlines as in Efuns so easier *)
      let rec find_bol_or_start pos =
        if pos.i = 0 || pos.i = term.output_point.i ||
           term.text.(pos.i - 1) = '\n'
        then pos
        else find_bol_or_start { i = pos.i - 1 }
      in
      term.cursor <- find_bol_or_start term.cursor;
      move_origin_to_see term term.cursor


    (* Erase character ^H  (or Backspace or Delete) *)
    | 0x8 ->
      let cursor = term.cursor in
      if cursor.i = 0 || cursor = term.output_point
```

```
      then ()
      else begin
        (* less: adjust if cursor = term.origin_visible *)
        (* this will adjust term.cursor *)
        delete_runes term { i = cursor.i - 1 } 1
      end

    (* todo: should remove once key is directly a rune.
     * less: the sequence we get for Left arrow is really
     * 0xEF 0x80 0x91, but we just check if previous code is 0x80.
     *)
    | 0xEF -> ()
    | 0x80 -> ()

    (* ordinary characters *)
    | _ ->
      debug_keys term key;
      (* this will adjust term.cursor *)
      insert_runes term term.cursor [key]
    );
    previous_code := Char.code key;
    if !previous_code = 0xEF || !previous_code = 0x80
    then ()
    else
    repaint term
```

⟨*function* Terminal.runes_in 109a⟩≡ (109b)

```
  (* "when characters are sent from the host, they are inserted at
   * the output point and the output point is advanced."
   *)
  let runes_in term runes =
    insert_runes term term.output_point runes;
    term.output_point <- { i = term.output_point.i + List.length runes };
    repaint term;
    ()
```

⟨Terminal.ml 109b⟩≡

```
  open Common

  open Point
  open Rectangle

  module I = Display
  module D = Display

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Routines to support a simple terminal emulator.
   *
   * A terminal has many features in common with an editor: you can enter
   * text, move around the cursor, copy, cut, paste, etc. The situation
   * is simpler than in Efuns though; there is no need for a gap buffer because
   * most insertions are at the end of the "file". A growing array
   * is good enough. Like in Efuns, we have also a few "cursors" that need
   * to be updated once you insert text.
   *
   * todo:
   *  - interactive scroll bar
   *  - selection and highlited text
```

```
 * less:
 *  - extract independent stuff and put in lib_graphics/ui/text_ui.ml?
 *  - extract even more independent stuff in editor.ml somewhere?
 *  - take code from zed opam package?
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)

(* The type below is called a 'point' in Efuns, but it would be
 * confusing with the Point.t of lib_graphics/geometry/point.ml.
 * We could also call it 'cursor', but this would be
 * confusing with the Cursor.t of lib_graphics/input/cursor.ml
 *
 * less: make mutable instead of the fields in 't' below?
 *)
⟨type Terminal.position 35d⟩
⟨constant Terminal.zero 35e⟩

⟨type Terminal.t 35a⟩

⟨type Terminal.colors 101e⟩
⟨constant Terminal.default_colors 102a⟩
⟨constant Terminal.dark_grey 102b⟩

⟨constant Terminal.scrollbar_width 102c⟩
⟨constant Terminal.scrollbar_gap 102d⟩

⟨constant Terminal.tick_width 102e⟩

(* temporary images *)
⟨constant Terminal.scrollbar_img 102f⟩
⟨constant Terminal.tick_img 102g⟩

(*****************************************************************************)
(* Debug *)
(*****************************************************************************)

⟨constant Terminal.debug_keys_flag 102h⟩
⟨constant Terminal.debug_keys_pt 102i⟩
⟨function Terminal.debug_keys 102j⟩

(*****************************************************************************)
(* Colors *)
(*****************************************************************************)
⟨function Terminal.init_colors 102k⟩

⟨function Terminal.colors_focused_window 103a⟩
⟨function Terminal.colors_unfocused_window 103b⟩

(*****************************************************************************)
(* Scrollbar *)
(*****************************************************************************)
⟨function Terminal.scroll_pos 103c⟩

⟨function Terminal.repaint_scrollbar 103d⟩

(*****************************************************************************)
(* Text content *)
```

```
(*****************************************************************************)
```
⟨*function* Terminal.visible_lines 104a⟩

⟨*function* Terminal.repaint_content 104b⟩

```
(* helper to know if we should send runes on channel connected to an app *)
```
⟨*function* Terminal.newline_after_output_point 104c⟩

⟨*function* Terminal.move_origin_to_see 104d⟩

```
(*****************************************************************************)
(* Tick (cursor) *)
(*****************************************************************************)
```
⟨*function* Terminal.point_of_position 105a⟩

⟨*function* Terminal.repaint_tick 105b⟩

```
(*****************************************************************************)
(* Entry points *)
(*****************************************************************************)
```
⟨*function* Terminal.alloc 105c⟩

⟨*function* Terminal.insert_runes 106⟩

⟨*function* Terminal.delete_runes 107a⟩

⟨*function* Terminal.repaint 107b⟩

```
(*****************************************************************************)
(* External events *)
(*****************************************************************************)

(* todo: right now 'key' is a byte (a char), but it should be a rune.
 * So for now we need to reconstruct a rune from a series of chars
 * (the utf8 encoding of the rune).
 *)
```
⟨*constant* Terminal.previous_code 108a⟩

⟨*function* Terminal.key_in 108b⟩

⟨*function* Terminal.runes_in 109a⟩

```
(* "When newline, chars between output point and newline are sent."
 * let bytes_out term =
 * but the logic is in Threads_window.bytes_out to factorize code
 * with the logic when in raw-mode.
 *)
```

D.23 Threads_fileserver.mli

⟨Threads_fileserver.mli 111a⟩≡
  ⟨*signature* Threads_fileserver.thread 48b⟩

D.24 Threads_fileserver.ml

⟨Threads_fileserver.ml 111b⟩≡

```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common

module D = Device
module N = Plan9
module P9 = Protocol_9P
module T = P9.Request
module R = P9.Response

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* ??? *)

(*****************************************************************************)
(* Constants *)
(*****************************************************************************)
⟨constant Threads_fileserver.all_devices 37h⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Threads_fileserver.device_of_devid 38b⟩

⟨constant Threads_fileserver.toplevel_entries 38a⟩

⟨function Threads_fileserver.answer 49b⟩

⟨function Threads_fileserver.error 93c⟩

⟨function Threads_fileserver.check_fid 71⟩

(*****************************************************************************)
(* Dispatch *)
(*****************************************************************************)
⟨constant Threads_fileserver.first_message 48d⟩

⟨function Threads_fileserver.dispatch 48e⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Threads_fileserver.thread 48c⟩
```

D.25 Thread_keyboard.mli

```
⟨Thread_keyboard.mli 112a⟩≡
  ⟨signature Thread_keyboard.thread 42a⟩
```

D.26 Thread_keyboard.ml

```
⟨Thread_keyboard.ml 112b⟩≡
  (* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
  open Common

  (*****************************************************************************)
```

```
(* Prelude *)
(*****************************************************************************)
(* Reads from the keyboard and sends the key to the "current" window *)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Thread_keyboard.thread 42b⟩
```

D.27 Thread_mouse.mli

⟨Thread_mouse.mli 113a⟩≡

```
⟨signature Thread_mouse.thread 42d⟩
```

D.28 Thread_mouse.ml

⟨Thread_mouse.ml 113b⟩≡

```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)

⟨type Thread_mouse.event 43a⟩
  (* less: Resize? other? or use other thread and device? (cleaner) in which
   * no need for event type here.
   *)

⟨type Thread_mouse.under_mouse 44f⟩

(*****************************************************************************)
(* Menus *)
(*****************************************************************************)

⟨function Thread_mouse.wm_menu 56b⟩

⟨function Thread_mouse.middle_click_system 45f⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function Thread_mouse.thread 43b⟩
```

D.29 Threads_window.mli

⟨Threads_window.mli 113c⟩≡

```
⟨signature Threads_window.thread 45g⟩
```

D.30 Threads_window.ml

⟨Threads_window.ml 114a⟩≡
```
open Common

open Point
open Rectangle
open Window

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Types *)
(*****************************************************************************)
```
⟨*type* Threads_window.event 45h⟩
```
(*****************************************************************************)
(* In and out helpers *)
(*****************************************************************************)
```
⟨*function* Threads_window.key_in 46g⟩
⟨*function* Threads_window.runes_in 81a⟩

⟨*function* Threads_window.mouse_in 47f⟩
⟨*function* Threads_window.mouse_out 77g⟩

⟨*function* Threads_window.bytes_out 79f⟩

⟨*function* Threads_window.cmd_in 48a⟩

```
(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
```
⟨*function* Threads_window.wrap 46a⟩

⟨*function* Threads_window.thread 45i⟩

D.31 Virtual_cons.mli

⟨Virtual_cons.mli 114b⟩≡
⟨*signature* Virtual_cons.dev_cons 78c⟩
⟨*signature* Virtual_cons.dev_consctl 81c⟩

D.32 Virtual_cons.ml

⟨Virtual_cons.ml 114c⟩≡
```
open Common
open Device

```
⟨*constant* Virtual_cons.dev_cons 78d⟩

⟨*constant* Virtual_cons.dev_consctl 81d⟩

D.33 Virtual_draw.mli

⟨Virtual_draw.mli 115a⟩≡
  ⟨*signature* Virtual_draw.dev_winname 82d⟩

D.34 Virtual_draw.ml

⟨Virtual_draw.ml 115b⟩≡
```
open Common
open Device

(* The ancestor of rio (8 1/2) was serving a virtual /dev/draw,
 * which was more elegant but also more inefficient than the
 * /dev/winname (and associated /dev/draw/x/) approach used by rio.
 *
 * alt: we could also pass the information of winname through
 * the environment instead of through a /dev virtual file.
 *)
```
  ⟨*constant* Virtual_draw.dev_winname 82e⟩

D.35 Virtual_mouse.mli

⟨Virtual_mouse.mli 115c⟩≡
  ⟨*signature* Virtual_mouse.dev_mouse 76a⟩
  ⟨*signature* Virtual_mouse.dev_cursor 82b⟩

D.36 Virtual_mouse.ml

⟨Virtual_mouse.ml 115d⟩≡
```
open Common

open Device
open Point
```
  ⟨*constant* Virtual_mouse.dev_mouse 76b⟩

  ⟨*constant* Virtual_mouse._dev_cursor 82c⟩

D.37 Window.ml

⟨*type* Window.mouse_counter 115e⟩≡ (116a)
```
type mouse_counter = int
```

⟨*constant* Window.window_border_size 115f⟩≡ (116a)
```
(* important convention to follow for rio and draw to cooperate correctly *)
let window_border_size = Draw_rio.window_border_size (* 4 *)
```

⟨Window.ml 116a⟩≡
```
open Common

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The data structure to store all the information about a window! *)

(*****************************************************************************)
(* Types and constants *)
(*****************************************************************************)

⟨type Window.wid 33a⟩

⟨type Window.mouse_counter 115e⟩
⟨type Window.topped_counter 34d⟩

⟨type Window.cmd 46c⟩

⟨type Window.t 32⟩

⟨global Window.wid_counter 33c⟩
⟨global Window.topped_counter 34e⟩

⟨constant Window.window_border_size 115f⟩

⟨type Window.border_status 61c⟩

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Window.pt_inside_border 54e⟩
⟨function Window.pt_on_border 54d⟩

⟨function Window.alloc 59d⟩
  (* less: incref? in caller? *)
```

D.38 Wm.ml

⟨function Wm.window_cursor 116b⟩≡ (116c)
```
(* less: a rio_cursor? with lastcursor opti? and force parameter? *)
let window_cursor (w : Window.t) (pt : Point.t) (mouse : Mouse.ctl) : unit =
  let cursoropt =
    (* less: if img is nil? if screenr is 0? *)
    match Globals.window_at_point pt with
    | Some w2 when w2  == w -> w.mouse_cursor
    | _ -> None
  in
  (* less: if menuing? or use corner_cursor() so no need this global? *)
  (* less: if holding *)
  match cursoropt with
  | Some x -> Mouse.set_cursor mouse x
  | None -> Mouse.reset_cursor mouse
```

⟨Wm.ml 116c⟩≡
```
(* Copyright 2017-2026 Yoann Padioleau, see copyright.txt *)
open Common
```

```
(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The Window Manager *)

(*****************************************************************************)
(* Cursors *)
(*****************************************************************************)

(* less? move those cursor functions in cursors.ml? *)

⟨function Wm.window_cursor 116b⟩

⟨function Wm._corner_cursor 54b⟩

⟨function Wm.corner_cursor_or_window_cursor 54c⟩

(*****************************************************************************)
(* Borders and content *)
(*****************************************************************************)
⟨function Wm.draw_border 62a⟩

(* repaint border, content for textual window, (and todo cursor?) *)
⟨function Wm.repaint 61b⟩

⟨function Wm.set_current_and_repaint 61a⟩

(*****************************************************************************)
(* Wm *)
(*****************************************************************************)

⟨global Wm.threads_window_thread_func 59a⟩

⟨function Wm.new_win 57b⟩

⟨function Wm.close_win 65d⟩

⟨function Wm.top_win 64g⟩

⟨function Wm.hide_win 68a⟩

⟨function Wm.show_win 68d⟩

(*****************************************************************************)
(* Helper for? *)
(*****************************************************************************)
⟨function Wm.resize_win 67b⟩
```

D.39 Wm.mli

```
⟨Wm.mli 117⟩≡
  (* The Window Manager *)

  (* main wm actions *)

  ⟨signature Wm.new_win 57a⟩
  ⟨signature Wm.close_win 65c⟩
```

⟨*signature* `Wm.hide_win` 67d⟩
⟨*signature* `Wm.show_win` 68c⟩

⟨*signature* `Wm.top_win` 64f⟩

⟨*signature* `Wm.resize_win` 67a⟩

```
(* helpers for borders *)
```

⟨*signature* `Wm.set_current_and_repaint` 59c⟩

```
(* helpers for cursors *)
```

⟨*signature* `Wm.corner_cursor_or_window_cursor` 44b⟩

⟨*signature* `Wm.window_cursor` 44c⟩

⟨*signature* `Wm.threads_window_thread_func` 58g⟩

D.40 tests/hellorio.ml

⟨`tests/hellorio.ml` 118⟩≡

```
open Common
```

⟨*type* `Hellorio.event` 30d⟩

⟨*function* `Hellorio.redraw` 30c⟩

⟨*function* `Hellorio.thread_main` 30b⟩

⟨*constant* `Hellorio._` 30a⟩

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.

Bibliography

[FDFH90] James Foley, Andries Van Dam, Steven Feiner, and John Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, 1990. cited page(s) 8, 13

[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) 8, 13

[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) 13

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 14

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 12

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 14

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 10, 12, 20, 26, 28, 30

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 12, 28

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Graphics System draw*. 2016. cited page(s) 9, 10, 12, 18, 20, 22, 23, 26, 27, 28, 29

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Network Stack /net*. 2016. cited page(s) 12

[Pad18] Yoann Padioleau. *Principia Softwarica: The Plan 9 Shell rc*. 2018. cited page(s) 28

[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) 9, 14

[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) 17, 23

[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) 10, 14, 20

[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) 9, 14

[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) 9, 10, 14, 26

[Pik00] 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) 9, 14

[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) 10, 13

[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) 10, 13

[SG86] Robert W. Scheifler and Jim Gettys. The X Window system. In *ACM Transaction of Graphics*, pages 79–109, 1986. cited page(s) 10

[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) 15

[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) 8