Rio: Design of a Concurrent Window System

Rob Pike
Computing Sciences Research Center
Bell Labs
Lucent Technologies
rob@plan9.bell-labs.com

February 4, 2000

History

Topic: the external and, particularly, internal design of Rio, the (new) Plan 9 window system.

Descendant of 8½:

> Pike, Rob, ‘‘8½, the Plan 9 Window System’’, *Proceedings of the Summer 1991 USENIX Conference*, Nashville, 1991, pp. 257-265.

8½ originally written in C, using processes and hand-built coroutines based on `setjmp` and `longjmp` (an interesting exercise).

For Brazil, rewritten from scratch in Alef in 1995, and converted to true threads and procs.

Converted (in an afternoon) in 1999 to use the thread library.

Historical note: although Alef was a fruitful language, it proved too difficult to maintain a variant language across multiple architectures, so we took what we learned from it and built the thread library for C.

Overall Design

Window system provides three things beyond the obvious:

- System interface (to access underlying hardware)
- Program interface (to manipulate windows)
- User interface (mouse, keyboard, menus...)

Then it must multiplex all these things in parallel for multiple clients.

Rio is unusual among window systems (although not among Plan 9 services) because it is implemented as a *file server*.

9P: the Plan 9 File Server Protocol

(Almost) Everything in Plan 9 is a file server or at least represented as one:

- Processes (for debugging)
- Environment variables
- Console
- Graphics
- Network
- ...

Client attaches to a server.
Client requests, server responds.

Requests:

- Attach
- Walk, Open, Read, Write, Close, Create
- Remove, Stat

Can be provided over any reliable channel: TCP/IP, pipe, ...

Name space

A *name space* is the set of files accessible to a process.

In Unix, the name space is the same for all processes. In Plan 9, it is controllable; each process can have a private name space. In practice, name spaces are shared by a set of related processes called a group.

User profile builds name space starting from / and mounting (attaching) services.

Recall all services, including basic devices, are file servers.

That's Plan 9, in a nutshell.

Example devices

Process:

```
/proc/1/...
...
/proc/27/ctl
/proc/27/mem
/proc/27/note
/proc/27/notepg
/proc/27/proc
/proc/27/status
/proc/27/text
...
```

Console:

```
/dev/cons
/dev/time
/dev/cputime
/dev/pid
```

(All textual)

Example devices (cont’d.)

Network:

```
/net/il/clone
/net/il/stats
/net/il/29/ctl
/net/il/29/data
/net/il/29/err
/net/il/29/listen
/net/il/29/local
/net/il/29/remote
/net/il/29/status
...
/net/tcp/...
/net/udp/...
/net/cs
...
```

As with all these examples, no special system calls required. Just open, read, and write files.

Example devices (cont'd.)

Mouse

```
/dev/mouse
/dev/cursor
```

Raster Graphics

```
/dev/screen
/dev/window
/dev/draw
```

Format of `/dev/draw` is RPC over a file. (Nested RPC.)

```
void
draw(Image *dst, Rectangle dr,
    Image *src, Point sp,
    Image *mask, Point mp);

/dev/draw:
    'd'
    2 bytes of destination id
    4x4 bytes of destination rect.
    2 bytes of source id
    ...
```

Structure of the window system (I)

To write a window system,

Read `/dev/cons` for input chars.
Read `/dev/mouse` for mouse position & buttons.
Write `/dev/draw` to update display.

Now, what does a client do?

Read `/dev/cons` for input chars.
Read `/dev/mouse` for mouse position & buttons.
Write `/dev/draw` to update display.

Same thing!

Structure of the window system (II)

So, Rio is just a multiplexer:

- Provide same (appearance of) environment for applications.
- Give each client these file *names*, but each client gets distinct set.

Different `/dev/cons`, `/dev/mouse` in each window. (In fact, 8½ also provided a different `/dev/draw` but Rio doesn't, which makes it much more efficient. Details omitted.)

Compare this to Unix `/dev/tty`, which is same file but different contents. With Rio, `/dev/cons` is actually a different *file*, but same *name* and different contents. (I argue this is much cleaner.)

Summary: Each window mounts a different root of a distinct file system implemented by Rio, containing an identical-looking simulation of the standard device file set for display, mouse, and keyboard.

Intermission: Nice side-effects

Window delete: reference count the open files.

DEL (interrupt)

`/dev/cons`

Remote transparency: if you can export a file (e.g. NFS), you can export the window system.

Recursion:

- Easy debugging: run window system in a window.
- Easy off-loading: run window system on CPU server.
- Easy simulation: can run X in a window.

The problem

Rio must manage the following all at once:

- Input from the mouse
- Input from the keyboard
- Window management (unlike in X, the user interface for window management is part of Rio)
- Output to the display
- I/O requests in 9P from the client to read and write `/dev/cons`, `/dev/mouse`, `/dev/window`, etc.

How to manage all these parallel, simultaneous requirements?

Also, as in any multiplexer, how to avoid locking the critical resource (internal data structures) for every operation? Locking is expensive and deadlock-prone.

Concurrent Design (I): Process allocation

For now, let's not worry about process vs. thread. Give every separate component a process to handle it:

1. Windows. (Each will run an instance of the same `winctl` function with different arguments.)
2. The mouse.
3. The keyboard.
4. The user interface for window management.
5. The file server connection that receives 9P requests.
6. Each incoming client I/O request.

All these processes will communicate entirely by messages on channels.

Note that most of the time, all these processes will be blocked waiting for something. The power of the model is that we can decompose the complex state of the system into independently executing, simple sequences of operations. External events will naturally lead to sequencing of the processes through their actions by scheduling the associated communications.

Process vs. thread

The choice of which to pick is dictated by two factors: (1) blocking system calls and (2) locking protocols.

1. A system call (`read`, `write`, etc.) will block, and that means all threads in the same process will block with it. Therefore, we wish to execute blocking system calls (read the mouse, read the 9P pipe, etc.) in separate processes, and have them pass their data on channels when they get it.

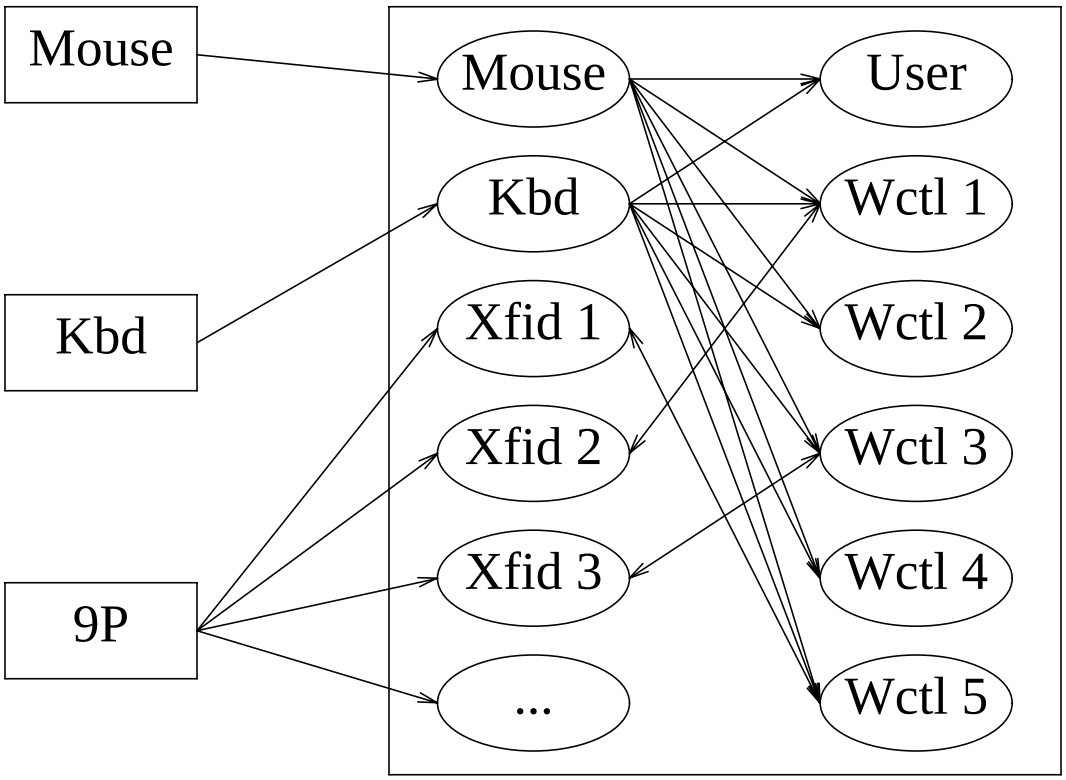
2. Processes executing in parallel have no guarantees about simultaneous access to memory, so we must use locks to protect data structures. However, *threads* within a single process do not execute simultaneously, and scheduling occurs only at communication points, so between communications a thread can access global data without interfering with other threads in the same process, and without needing to set locks.

Our overall design is therefore:

- A single process responsible for all shared data, holding all threads that must access the data.
- A separate process for each blocking call, passing requests for action to threads in the main process.

A Picture

Boxes are processes, ovals are threads.



Other communications exist, e.g. User interface thread also talks to window control threads to signal resize, etc.

Comments

Rio can be thought of as a main program (the big box) with lots of threads within it, and several server processes connecting operating system services to it.

No shared data is touched outside the box, so locking is unnecessary.

Communication synchronizes, schedules, and protects data from modification.

This parallel composition of processes/threads means that the system doesn't block while waiting for something; e.g. window control actions can happen while client is busy writing to window.

It's remarkably easy to write software this way.

Xfids (1)

Xfids carry the state of a single 9P message through the system. They are managed as a pool of anonymous threads that get allocated and assigned to a 9P message dynamically.

Incoming request might be answerable by file server process directly, e.g. a request to read the directory. If it needs access to shared data, though, it hands off the request, like this:

Xfids (2): 9P Process

```
    Xfid *x;

    for(;;){
        read(fd, message);
        if(x == 0){
            sendp(cxfidalloc, nil);
            x = recvp(cxfidalloc);
        }
        convert message into x;
        x = (*fcall[x->type])(x);
    }

extern void xfidread(Xfid*);

static
Xfid*
fsysread(Xfid *x)
{
    if(read of directory){
        handle message;
        return x;
    }
    sendp(x->c, xfidread);
    return nil;
}
```

Xfids (3): Threads in main process

Allocator process is trivial; most of code omitted here. It starts a new Xfid like this:

```
x=emalloc(sizeof(Xfid));
x->c=chancreate(sizeof(void(*)(Xfid*)), 0);
threadcreate(xfidctl, x, 16384);
```

Xfid controller looks like this:

```
void
xfidctl(void *arg)
{
    Xfid *x;
    void (*f)(Xfid*);
    x = arg;
    for(;;){
        f = recvp(x->c);
        (*f)(x);
        if(decref(x) == 0)
            sendp(cxfidfree, x);
    }
}
void
xfidread(Xfid *x)
{
    /* handle read as thread in main proc */
}
```