

Re: Maybe we should stop "Paging Beth Stone" already...

## Re: Maybe we should stop "Paging Beth Stone" already...

---

*Source:* <http://coding.derkeiler.com/Archive/Assembler/alt.lang.asm/2005-09/msg00986.html>

---

- *From:* "Richard Cooper" <[spamandviruses@xxxxxx](mailto:spamandviruses@xxxxxx)>
  - *Date:* Tue, 20 Sep 2005 12:14:11 GMT
- 

On Mon, 19 Sep 2005 16:54:25 -0400, wolfgang kern <[nowhere@xxxxxxxxxxxxx](mailto:nowhere@xxxxxxxxxxxxx)> wrote:

My OS already decodes everything when a key is hit and put this info into a global structure and into a buffer if configured that way.

What else would be needed ?

Plug a second keyboard into your PS/2 mouse port, then tell me what's needed to utilize it in your OS. Can it be set up from userland, or do you have to modify the kernel or the drivers?

In mine, the kernel would have a PS/2 port driver, which would export two IPC devices, one for each PS/2 port. So normally, you'd load a keyboard driver which attaches to the first PS/2 port, and a mouse driver which attaches to the second PS/2 port. So if you plugged a keyboard into the second PS/2 port, all you would have to do is unload the mouse driver, and load a second keyboard driver, telling it to use the second PS/2 port instead of the first. Since the IPC protocol it uses to connect to the second PS/2 port is identical to the IPC protocol it uses to connect to the first PS/2 port (and indeed, identical to the IPC protocol it would use to connect to anything and everything), it will happily connect to it instead. So that's it, that's all you have to do to get the OS to support the second keyboard.

Now to get the applications to support the second keyboard, you have to do a little more, but there are several options:

One option is to also install a second video card at the same time. Then you load one copy of the multiplexer on the first video card and keyboard, and another copy on the second video card and keyboard. Then, because programs that run will decide which multiplexer to connect to via something like an environment variable, you've basically got two computers

Re: Maybe we should stop "Paging Beth Stone" already...

## Re: Maybe we should stop "Paging Beth Stone" already...

in one now.

No program needs to know that it's on such a system, as they all just look at their environment variables and find some which read "VIDEO=[multiplexer\_1][video]" and "KEYBOARD=[multiplexer\_1][keyboard]", or they find some that read "VIDEO=[multiplexer\_2][video]" and "KEYBOARD=[multiplexer\_2][keyboard]". Additionally, you could use only one multiplexer, and use the second video/keyboard pair for a single application, in which case that program's environment variables would be "VIDEO=[video\_2]" and "KEYBOARD=[keyboard\_2]".

Note also that the IPC names are chosen by the user as well, so that for example if the two display/keyboard pairs were in different rooms, they could be named "[multiplexer\_in\_living\_room]" and "[multiplexer\_in\_kitchen]". The IPC names are just text strings which no program ever tries to make sense of, they're solely for the benefit of the user.

Another option is to write a simple little program that connects to both keyboard drivers and combines them into a single device, and then tell the (one) multiplexer to connect to this device. Then both keyboards effectively act as one, and you can use them interchangeably. The downside to this is that it requires the user to write a program, however they still at least don't have to write a new keyboard driver, or worse yet, modify the kernel.

Yet another option is to use a program that knows how to access additional keyboards. For example, a game with a two player mode might know how to access a second keyboard, so that each player could have their own keyboard. In this case, you would go into the program's setup and give it the IPC name of the second keyboard, and then it would simply connect to it the same way as it connected to its first keyboard, and then it could read from both devices.

It's this kind of flexibility that I'm looking for. I always want to do things in new ways that the system designer never thought of, so I think that a design that's extremely flexible is a good design.

OS post-development in user-space ? :)  
Much better is an OS who already does what and how we want it.

That assumes that you can predict the future. You never know what you will want to do tomorrow.

## Re: Maybe we should stop "Paging Beth Stone" already...

I myself think that things should be done one way one day, and no matter how sure I was that I had the best idea for how they should be done, it's always possible the next day that I either find something wrong with my previous idea, or at least come up with a better idea. So I thought I'd design a system where changing things is easy, that way it never has to get stuck doing things one way simply because that's the best idea I had when I designed the system.

| So in my OS, you'd have a keyboard port, mouse port, and  
| video card | driver which just did the actual hardware  
| interfacing, and nothing| else.

Now here (in the hardware section) while in PL0 can be already done a lot of useful things, saving the user from libraries and detours.

Yes, but I'd rather not be saved from those things.

| Then I'd have a program that I would call the multiplexer,  
| which connects to the keyboard and mouse drivers,  
| and the video card driver.

?? I don't need a multiplexer to display the mouse, the IRQ12(or IRQ3,4,10,11) does this on its own if enabled. If taskchanges then just the mouse's image may be updated, and if there is only one mouse (I can have up to three), then only one global structure (x,y,dx,dy,r/m/l-button and some more) is needed. So the active thread can read it anytime without a task switch.

The multiplexer is designed to provide a generic interface, so it doesn't make any assumptions about how the mouse or any other hardware will be used by the applications. Instead, it simply performs the task of making a single display, keyboard and mouse appear to be many different displays, keyboards and mice. That's where it fits into the design, so that's all that it does.

Certainly things would be faster if the mouse driver drew directly to the screen, but that requires the mouse driver to know about the screen, and that's not what I want. I want total configurability, so that if I feel so inclined I can do something as ridiculous as have a mouse attached to computer #1 and run the mouse driver on that computer, and run the mouse driver's data over a TCP connection to computer #2 and use the mouse on

## Re: Maybe we should stop "Paging Beth Stone" already...

computer #1 with computer #2's display, without having the mouse's cursor fail to appear because the mouse driver is drawing the cursor to computer #1's display.

In my OS, the mouse driver would simply read from whatever device it was told to read from, and take whatever mouse protocol the mouse uses and convert it to the standard mouse protocol used by the OS. Aside from that, it wouldn't do anything else.

| It then makes available a service to which other programs  
| can connect to,  
| just like they would connect to the real  
| keyboard/mouse/video, and it| does the multitasking switching  
| between them, | so when you type a key combination on the  
| keyboard to switch to a | different task, the multiplexer  
| starts paying attention to a different | task, and it either  
| ignores or buffers the output of the other tasks.  
| Then if, for example, you're making a dedicated system, you  
| can leave out  
| that multiplexer and connect the programs to the real  
| keyboard/mouse/video  
| drivers, so that task switching is no longer available.

I wouldn't direct connect the user to the hardware section, just making the IRQ-result structs global accessible for all user threads may do it.

Huh? I don't see how that applies to what it was in reply to.

(BTW, is my computer really wrapping text that poorly, or is that something on your end? I can't tell because if the text isn't quoted then apparently Opera goes and unwraps it so that it can rewrap it to fit the width of the window, so I only see the wrapping when I see other people quoting me.)

I'm not familiar with abstract layers, for me every piece of hardware has to be treated as it needs to and all hardware drivers are connected via (user shared if granted) structures (including buffer pointers) to communicate with any software.

Yes, the above I'd assume to be 'user-system' stuff, except I have no idea what this IPC actually does :)

## Re: Maybe we should stop "Paging Beth Stone" already...

IPC is inter-process communication.

I didn't get all of the details ironed out in my design before I decided to do something else, but basically there is a function to tell the kernel that you want to offer a service via IPC, and then there's a function to tell the kernel you want to connect to some other process' IPC service, and then there are functions to do the actual inter-process communication.

The IPC just does basic communications, it can do stream or block access, with either seeking allowed or no seeking allowed. Any information that doesn't fit into these basic communication types has to be communicated some other way, usually as additional parameters to the IPC connection such as in "[sound][pcm\_44100\_stereo\_16\_signed]", as there's no equivalent of the ioctl() that Linux uses. The result of that is that I can just "copy sound.raw [sound][pcm\_44100\_stereo\_16\_signed]" instead of having to use a sound program that knows how to set up the sampling rate on the sound card. With everything working that way, it's easier to combine things in ways in which they were not intended to work, and have them work anyway.

My kernel routines know only how to read/write sectors and parts of it, so the filesystem (also core routines) can be made flexible.

In my OS the IDE driver would only work in sector sizes, if something that connected to it wanted to work in bytes instead of sectors, it would refuse. However, you could load a buffer program in between them, and the buffer program would make the IDE driver appear to have a one byte block size, as well as implement some caching if desired.

I see it dangerous to let a user play with  
FileSystem-internals...

It's perfectly safe as long as they know what they're doing. Now if they want to unload the driver that's providing disk access for the whole OS, you might make your driver unload command comment that doing so isn't such a good idea, but if they really want to do it, go ahead and let them. Maybe they've thought ahead and already have a system in place to load a different driver once that one unloads.

Additionally, if I don't feel that partitions are useful, in my OS I could just forget about the partition driver, and connect the filesystem driver directly to the IDE driver, effectively using the entire drive as a single partition, except that there is no partition table.

.