OS Design

This comes from an old alt.os.development news thread. The part that has the ">" on each line was written by Ian Nottingham. The rest of it was written by Alistair J. R. Young.

```
> Hey...I've been trying to get started on a kernel for awhile now, but
> all the examples i look at are so far beyond the extreme early stages
> that i can't figure out where to start and what exactally to do when
> it does....anybody have an example of this early stage of development?
> or perhaps a guide or list of what needs to be done?

The first thing I'd say you need to take into account is that there is
*no* whole subsystem to start with. One of the most fun things about
kernels, I find, is the way that everything has a half-dozen
dependencies on everything else - so you're left with developing
little chunks of a half-dozen subsystems simultaneously.

If you are starting from flat scratch, probably the first thing you
need to develop are a bunch of little macros and functions to put a
prettier face on the more messy bits of the x86 architecture. If
you're really ambitious, you can try and make it a portable or at
least semi-portable face. :)

(I really believe in cheating when it comes to these boring minutiae -
if you're not all that interested in writing lots of low-level
PC-hardware-bashing assembler code, I suggest you get the Utah Oskit
{just released in v. 0.96, woo!} where that bit's all been done for
you.)

You'll also need a bunch of library functions for the kernel (no
strcmp(), no strcpy(), no malloc(), etc. of course, so...?) - plus
debugging functions along the lines of assert() and panic().

At the same time, you *really* need to make your major design
decisions now, otherwise you'll end up in the same situation as I was
- and having to rip the thing out and do it over again is a *major*
pain.

The first big one's the memory model. Segmentation and/or paging? It
seems, these days, that most people go for paging w/o segmentation -
both because the flat memory model's neater, and because paging's
generally portable and segmentation isn't.

And where do you put the kernel in it? Do you allow a separate address
space for each process, or all mixed together? Virtual memory?

Then there's multitasking/threading, if you decide to have
them. Perhaps most crucially at the start, is it possible your code
will ever be running on a SMP machine? If so, put in all the spinlocks
and other synchronisation stuff you'll need right from the start -
doing it in retrospect is an *incredible* nightmare.

What's your process model? Just the basic two-level
processes-owning-threads, or something more sophisticated? (I'm using
a three-level InteractiveSession/Job/Facility owns Process owns Thread
model, myself.) When you do the task-switching, are you going to write
your own code to do it, or use the Intel-provided TSSes to do it. If
the latter, are you going to go the whole way and have one TSS per
thread, or have two and swap the appropriate information into and out
of the inactive one before and after every task switch?

System calls. How do you do them? Several possibilities - Linux, et
al. seem to use software interrupts, so that's quite popular. Other
options - call gates to let a user-mode program directly call the
appropriate kernel functions. Or just have one call to send messages
```

to IPC ports, or some such, and run all communication even with the kernel over IPC.

And speaking of IPC - Message ports? Named pipes? Queues? Local sockets? All of the above? Best to think about it now, because it can influence the rest of the design quite a bit - you may want to have one fundamental one and implement everything else in terms of it (in Laura, for example, everything's pretty much a derivative of the basic message port - which influences the design of the TCP/IP stack (networking - something else to think about).

Ich. Device drivers - how do you talk to them? And the filesystem. This is all one muxed-up topic, really. Simplistically, there's always the Unix model where you treat everything as a file and access it all through the filesystem. Or there are more exotic options waiting out there - again to use my own kernel as an example, the Laura model has a 'namespace manager' which lists every object on the system - devices, filesystems and their files, etc., etc. To access anything, device, file, whatever, you ask the namespace manager and it gives you back an (harking back to above) an IPC port for that object - and each registered thing can support a totally customised set of actions, not just the standard open(), close(), read(), write(), ioctl()...

Security! If you're going to have any, think about it now. It's almost worse than SMP to retrofit, at least if you want to avoid having security holes everywhere. And there are so many different ways to do it - mine uses object ACLs validated against a possible three different tokens depending on what the relevant thread, task and session are doing.

Well, that's all design stuff, but that's the most important in my book. Getting it done first, I mean. Otherwise, you end up spending a lot of time ripping out and rewriting.

Once you actually start putting code to keyboard... well, (unless you're using something that gives you one), about the first thing you need is a crude console driver. It almost certainly won't be the one you use to drive the console in the final revision, but trying to debug anything without printf() is an absolute bastard...

Then - let me see. It depends a lot on how your OS is laid out (design again :)) and what the dependencies are. The first thing you need, obviously, is code that boots the kernel (if you use the MultiBoot boot loader, or something similar, this is rather less painful than otherwise. It might be an idea to use something like it to start with even if you plan to write your own boot-loader later) and does the most basic setup (get the segments set right, allocate a kernel heap and stack to work with, and so on and so forth.)

Then, my first step was to write something that displayed a startup banner and halt (so I could actually see something happening - always a plus!). Then, building on that - let's see, my first step was to rewrite that to display the banner and then enter the system idle loop (the loop that will eventually become the idle thread, that is). That *is* actually more complicated than just 'for (;;) ;' - as I think it's a good idea to halt the processor during it to avoid burning too much power, but...

I think the next thing that's probably a good idea to get working is trap handling - even though, at this stage, about all that you can do is display a message, dump state, and panic the kernel (this should, IMO, be your default handler for everything). The messages will come in handy when debugging later (much more useful than just a crash), and it's a sight quicker to implement the real handlers when you've already got a framework.

The same thing goes for interrupt handling, which I think is probably the next thing that needs to go on the list. I wouldn't panic on an

```
unexpected interrupt, though.

After that - well, after that it gets very dependent on the kernel
you're doing. My next step was to get a handler running for the timer
interrupt - just about everything uses the system timer for
*something*, so it's pretty essential. And, also, I need that
working for the scheduler to work, so...
```

The Utah OSKit can be found at http://www.cs.utah.edu/flux/oskit/