Filesystems and block devices

Ok, welcome to the last module of the course! We're going to learn about filesystems and block devices.

(redraw usual picture of the CPU, memory, and disk). We have occassionally talked about how the heck the OS accesses the disk. We talked about the CPU "automatically" knowing a particular offset in the disk to go and load the initial operating system when the CPU turns on. We also talked about using mmap to effectuate program loading: can anyone remind me what we said about this before?

(hopefully the class will supply e.g., when we create a new process control block, all we need to do is specify the disk locations where the program resides, and map them to known memory locations in virtual memory, and then let the MMU do the whole process-loading thing).

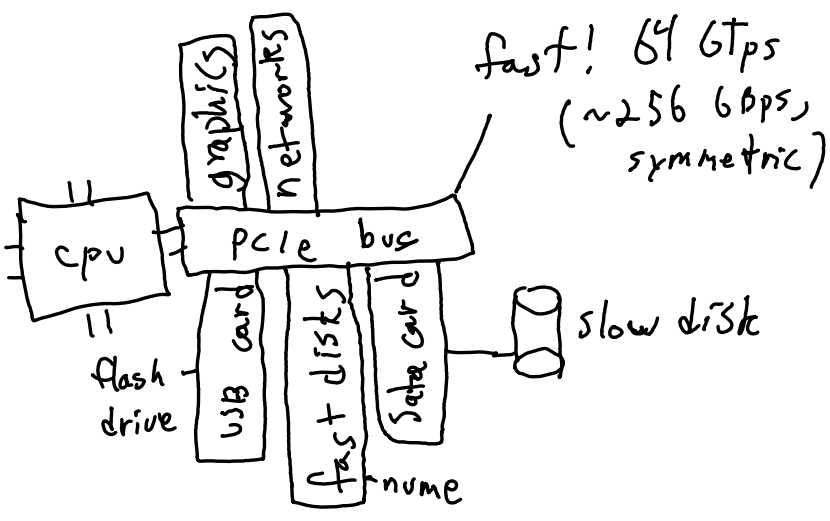
So, this module is going to focus broadly on two questions we still have left over from that time.

(1): how do you communicate with the disk?

(2): how do you find the particular disk addresses that you want to load?

1: communicate with the disk

(draw the IO bus / PCIe stuff / IO controller)



Note that the PCIe bus (and also everything hanging below it) is a *bus*. It's like a primitive network in which everyone is directly connected to everyone else. In order to use it, some low-level logic needs to effectively poll to see if the bus is quiet, and then broadcast its message on the bus if so. If two components broadcast on the bus at the same time, then there's *bus congestion* and the messages get all garbled. This stuff is usually handled by the hardware though.

The devices.

Every single device on this bus---including the CPU, the graphics card, network controller, disk drives, etc---is basically an embedded system all on its own. Each one gets its own OS even! (which we call *firmware* because we're fundamentally CPU-pilled). They can all listen on the bus and broadcast to the bus independently. To make that work *at all* reasonably, its essential that these devices have worked out a *protocol* ahead of time to determine what should be sent or received. On the CPU OS side, this protocol is defined by device-specific **driver code.** Which winds up being most of the code "in" the kernel.

(you can illustrate this; as puzzle pieces maybe?)

The CPU needs fixed mechanisms to communciate with *all* of these devices, since it's not clear (to the OS) which devices *even exist* to start. Note that some devices (e.g. graphics) now get special handling, but we'll focus on the ones that don't.

Communication option 1: registers and instructions.

On x86, there's a specific in/out instruction on the CPU, which broadcasts bytes on the bus. Which bytes? it usually includes a "bus address" for the target devices, but beyond that the format is *basically up to the device*.

(draw this on the board)

Communication option 2: direct memory mapping

Normally we think of memory mapping as something the OS sets up; but in 64-bit systems, we have *so much address space* that it makes sense to let the *devices* register themselves at particular main memory address ranges. Under this setup, writes to those addresses are *automatically redirected* to the devices.

Regardless: interrupts and polling.

Devices have the opportunity to send *interrupts* to the CPU (the CPU effectively has dedicated pins for this), announcing themselves to exist and allowing the OS to react to whatever they have to say. A device interrupting a CPU is kinda symmetric to an x86 CPU sending an in/out instruction: the CPU gets a bus address telling it whence this interrupt comes, and then a bunch of bytes to decode, whose format is left up to the individual devices.

Ok, these are all our primitives! Let's now talk about how to use these primitives to build up various tasks that the OS needs to do with respect to the disks. And let's make this interactive: the OS boots up. What does it *need* to do, in order to make these disks usable? Let's think about some steps; we'll try to ask the audience, but if y'all are the frozen chosen then I'll pair you off again.

- disk discovery
  - option: hw auto-broadcasts a "wakeup" on the bus *after* CPU is ready to receive it (how? races? timing?)
  - option: CPU broadcasts an "identify yourself" on the bus, and gets interrupted a bunch
  - option: hardware uses a series of "known addresses" in memory to specify device information; devices register with hardware (effectively the same (from their perspective) as the interrupt-loop) and get a mapping set up for them
- reading
  - When we talk about reading/writing: disks are *sector/block-addressable*. Blocks are usually 512 bytes, but can be up to 4kb, it depends on the device!
- writing
- status / error / config
  - sometimes this can be fancy. Recall the IOCTL system call

Remember: you also need to talk about DMA. How does the disk actually access data it wants / place data into memory?? It has its own mmu! The CPU needs to set this up.

Remember: you also need to talk about the "device registers" e.g. status, ready, error, data. This is the memory-mapped stuff?

Note: we now have examples of memory contention that *does not go through the CPU*. This is where you really (really!) need to use the volatile keyword!!

Part 2: let's actually use the filesystem on this thing!

We're going to look at this from the *user* side first; which is a shame, because the FS side is more interesting. Ah well, more's the pity. I will need to improv this one!

Note: you spent almost all of your time on part 1. You drew a "realistic" system model. You talked a bit about graphics cards, and how to access them with BLIT. You talked a bit about the PCIe bus, and how it's a full broadcast bus---you also told them how fast it was. You talked about ways that an OS could communicate with a filesystem, by broadcasting on a bus and by receiving interrupts. You hypothesized several ways this could, in practice, work. You spent a bunch of time explaining firmware, and that firmware protocols existed. told them about sectors and blocks. You went over at the very end that inodes exist, and that directories are just mappings from names to inodes.