17 Lecture: Canceled

Outline:
- Devices
  - Low level considerations: timing, interleaving, etc.
  - Accessing a device: Device Controllers
- Reading from a device: DMA vs. Programmed IO
  - Goals of IO Software
  - IO Software
- Minix IO Structure
  - Minix IPC
    - Interrupt Handlers
      - Simple
      - complex
- Structure of Minix: Device Tasks (Drivers)
- Driver Behavior
  - Example: Disk seek optimization

• Coming attractions:

<table>
<thead>
<tr>
<th>Event</th>
<th>Subject</th>
<th>Due Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>asgn5</td>
<td>minget and minls</td>
<td>Wed Jun 5</td>
<td>23:59</td>
</tr>
<tr>
<td>asgn6</td>
<td>Yes, really</td>
<td>Fri Jun 7</td>
<td>23:59</td>
</tr>
<tr>
<td>final</td>
<td>stuff</td>
<td>Sat Jun 8</td>
<td>10:10 (in 03-201)</td>
</tr>
</tbody>
</table>

Use your own discretion with respect to timing/due dates.

• Building assignments. I had to build several asgn2 and asgn3s by hand.

• Note labs/assignments

• Exams hopefully Friday, although it is hard to tell.

I got sick

17.1 Devices

The Unix/Minix modes, devices are classified as:

- block devices (e.g. disks)
- character devices (e.g. ttys)


17.1.1 Low level considerations: timing, interleaving, etc.

Isn’t it nice the controller can take care of it?
(under each level is another nice level of abstraction)
17.1.2 Accessing a device: Device Controllers
Mercifully the OS talks to device controllers not the actual devices. Isn’t abstraction great?

Device controllers abstract away much of the complexity. Accessed via:

**Memory-Mapped IO** Device control registers are mapped into memory. (creates “holes” in memory)

**IO Ports** Device control registers must be accessed through special instructions. (convenient, but complicates the CPU).

Either way, we set a value in some register, then the controller does the thing, then it sets a register to tell us that it’s done it.

17.2 Reading from a device: DMA vs. Programmed IO

**Standard (“Programmed IO”)** Controller interrupts, CPU copies data from controller to memory.

**DMA** Controller copies data to memory, then interrupts.

At least we have interrupts (Think about the world if we didn’t. We’d just have to check again and again, called “polling”).

17.2.1 Goals of IO Software

**abstraction** Abstract away the complexity to achieve device independence.

**organization/standardization** It should be easy to name something. “/etc/motd”, rather than /dev/hda1/…, or, worse, (3,1)

**error handling** Fix it ASAP, or report it in a useful way.

**synchronization** Make operations make sense. (e.g. sharing a printer.)

17.2.2 IO Software

As always, it’s all about abstraction: (note the correspondence to the layers in Minix)

1. Interrupt handlers (bottom)
2. Device drivers — knows about the device
3. Device-independent OS software — does not know about the device
4. User-level software (top)

17.3 Minix IO Structure

We discussed principles of resource management, but how does this really work in the Minix environment?

The IO architecture of MINIX is shown in Figure 30

Corresponds to:

- user processes
User Processes
- make IO call; format IO; spooling
Device-independent software
- Naming, protection, blocking, buffering, allocation
Device drivers
- set up device registers; check status
Interrupt Handlers
- wake up driver when IO completed
Hardware
- perform IO

Figure 30: Minix IO

- servers
- tasks
- process management

17.4 Minix IPC

Three primitives of Minix IPC:

**Synchronous**
- send(dest, &message);
- receive(source, &message);
- send_rec(dest_src, &message);

**Asynchronous**
- notify(dest);

Messages (other than notify()) communicated via **rendezvous** semantics: The sender waits until the receiver gets the message. Notify is asynchronous.

Why?

- simplifies buffer management.
- more predictable: there is no question of a program behaving differently given a different buffer size (out of its control)
  (How would one find out the buffer size?)

17.4.1 Interrupt Handlers

- Disk
- Clock

Do a little work, then *tell someone.*
Can be simple or complex: (not always the way you’d think)

17.4.2 Simple

- disk: handler acknowledges interrupt and passes the word on to the device driver.
  why? These calls are infrequent, long term, and involve a lot of work.
17.4.3 complex

- clock: counts ticks `pending_ticks` and only wakes up handler if necessary. (Clock frequency 60hz, quantum 100ms, 6/schedule interval. (See comment in `clock.c`) wakes up driver if tty event or `SCHED_RATE` why? These calls are frequent, high overhead, not much to be done.

```c
* pending_ticks:
  * This is protected by explicit locks in clock.c. Don’t
  * update realtime directly, since there are too many
  * references to it to guard conveniently.
* lost_ticks:
  * Clock ticks counted outside the clock task.
* sched_ticks, prev_ptr:
  * Updating these competes with similar code in do_clocktick().
  * No lock is necessary, because if bad things happen here
  * (like sched_ticks going negative), the code in do_clocktick()
  * will restore the variables to reasonable values, and an
  * occasional missed or extra sched() is harmless.

* Are these complications worth the trouble? Well, they make the system 15%
  * faster on a 5MHz 8088, and make task debugging much easier since there are
  * no task switches on an inactive system.
```

- tty: (keyboard) each key event causes an interrupt (down/up) keep track of events (bochs?)

  More importantly: IO Processing state machine

17.5 Structure of Minix: Device Tasks (Drivers)

Device drivers are independent processes.

Drivers are all structured as:

```c
Initialization
while(TRUE) {
    receive(any,message)
    switch(mess.m_type) {
        ...
    }
    send(mess.m_source,reply);
}
```

17.6 Driver Behavior

Minix’s device drivers are very simple so that we can get out heads around them. What would we do if we were working on a “real” system.

17.6.1 Example: Disk seek optimization

Device driver behavior can be very complex for optimiziation:

Consider the issues involved in disk access: seek time, rotational latency, transfer time....

Candidate algorithms:

- First come, first served
• Shortest seek first
• elevator algorithm
• Further: pick up “blocks of opportunity”; try to predict