14 Lecture: IO Processing

Outline:
- Announcements
- Managing Multiple Resources
  - Resource Trajectories
- Managing Multiple Resources, cont.
  - Multi-way bankers’
- Wrapping up deadlock avoidance
- Back to I/O
- 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
  - Interrupt Handlers
  - Simple
  - complex

14.1 Announcements
- 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</td>
</tr>
</tbody>
</table>

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

- ABSTRACTION
- Lab03 due Monday
  - no late days
  - I will release lab solns right after
- Exam Wednesday
  - Bring questions Monday; I’ll leave time.

14.2 Managing Multiple Resources
But multiple types of resources are a problem...

14.2.1 Resource Trajectories
The example from Tanenbaum, p.249 is shown in Figure 27.
Yuck.
14.3 Managing Multiple Resources, cont.

14.3.1 Multi-way bankers’

Here we simply do a vectorized version of the Banker’s Algorithm:

- Maintain a table of held resources
- Maintain a table of maximum requests
- Maintain a table of remaining requests
- Maintain a vectors of allocated, free, total resources

If there does not exist a row less than the available vector, the system will deadlock, else:

1. Choose a process whose resource requirements can be satisfied. (It doesn’t matter which, because it always increases the resource pool.)
2. Assume its resources are released (because it’s finished)
3. Repeat until all processes terminate or there are no more satisfiable processes.

If all processes can terminate, the state safe. If not, it is unsafe and the resource request must be delayed.

See the example in Figure 28. Manipulations:
Figure 28: Multi-dimensional Bankers Algorithm
1. Initial situation

2. B requests an instance of $R_2$ (printer?): (safe: D, then A or E, then...)

3. E requests the last $R_2$ printer: **unsafe**

### 14.4 Wrapping up deadlock avoidance

Are any of these any good?

- Processes rarely know their resource requirements in advance.
- Processes come and go.

This is **hard**.

If you have a good idea, you can be famous like Dijkstra. :)

**What does Unix do?**

Not a thing. The Unix way (as with many other operating systems) is to hope it doesn’t happen, and if it does, it expects some higher being (super user) to fix it. Think *Deus extra machina*.

### 14.5 Back to I/O

Without IO, there’s no real point in doing the computation. It’s also complicated.

As always, it’s all about abstraction: keep the dirty machine details hidden.

### 14.6 Devices

The Unix/Minix modes, devices are classified as:

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


#### 14.6.1 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)

#### 14.6.2 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.
14.7 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”).

14.8 Goals of IO Software

<table>
<thead>
<tr>
<th>Goal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>abstraction</strong></td>
<td>Abstract away the complexity to achieve device independence.</td>
</tr>
<tr>
<td><strong>organization/standardization</strong></td>
<td>It should be easy to name something. “/etc/motd”, rather than /dev/hda1/..., or, worse, (3,1)</td>
</tr>
<tr>
<td><strong>error handling</strong></td>
<td>Fix it ASAP, or report it in a useful way.</td>
</tr>
<tr>
<td><strong>synchronization</strong></td>
<td>Make operations make sense. (e.g. sharing a printer.)</td>
</tr>
</tbody>
</table>

14.9 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)

14.10 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 29

| 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 |

Corresponds to:

- user processes
- servers
- tasks
- process management
14.11 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?)

14.11.1 Interrupt Handlers
- Disk
- Clock

Do a little work, then *tell someone.*

Can be simple or complex: (not always the way you’d think)

14.11.2 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.

14.11.3 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.

- `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. 
- `* pending_ticks`: 
  - * 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