CIS 415: Operating Systems

I/O, Final Review
Spring 2011
OS role in I/O

- Share the same device across different processes/users
- User does not see the details of how hardware works
- Device-independent interface to provide uniformity across devices.
I/O Peripherals
Talk to Devices

- **Communication**
  - Send instructions to the devices
  - Get the results

- **I/O Ports**
  - Dedicated I/O registers for communicating status and requests

- **Memory-mapped I/O**
  - Map the registers into address space
  - Communicate requests through memory operations

- **Memory-mapped data “registers” can be larger**
  - Think graphics device
Memory-mapped I/O

- Can read and write device registers just like normal memory.
- However, user programs are NOT typically allowed to do these reads/writes.
- The OS has to manage/control these devices.
- The addresses to these devices may not need to go through address translation since
  - OS is the one accessing them and protection does not need to be enforced, and
  - there is no swapping/paging for these addresses.
Consider a disk device ...
Reading sector from disk

Store [Command_Reg], READ_COMMAND
Store [Track_Reg], Track #
Store [Sector_Reg], Sector #

/* Device starts operation */
L:  Load R, [Status_Reg]
    cmp R, 0
    jeq

/* Data now on memory of card */
For i = 1 to sectorsize
    MemTarget[i] = MemOnCard[i]

You don't want to do this!
Instead, block/switch to
other process and let an
interrupt wake you up.

This is again a lot of
overhead to ask the main
CPU to do!
Interrupt Cycle

1. CPU
   - device driver initiates I/O
   - CPU executing checks for interrupts between instructions
   - CPU receiving interrupt, transfers control to interrupt handler
     - interrupt handler processes data, returns from interrupt
     - CPU resumes processing of interrupted task
2. I/O controller
   - initiates I/O
     - input ready, output complete, or error generates interrupt signal
DMA engine

Used to offload work of copying
Lots of different offload engines possible in systems
Store [Command_Reg], READ_COMMAND
Store [Track_Reg], Track #
Store [Sector_Reg], Sector #
Store [Memory_Address_Reg], Address

/* Device starts operation */

P(disk_request);

/* Operation complete and data is now in required memory locations */

Called when DMA raises interrupt after Completion of transfer

ISR() {
    V(disk_request);
    }

Assuming an integrated DMA and disk controller.
Issues to consider

- What is purpose of RAM on card?
  - To address the speed mismatch between the bit stream coming from disk and the transfer to main memory.

- When we program the DMA engine with address of transfer (Store [Memory_Address_Reg], Address), is Address virtual or physical?
  - It has to be a physical address, since the addresses generated by the DMA do NOT go through the MMU (address translation).
  - But since it is the OS programming the DMA, this is available and it is NOT a problem.
  - You do NOT want to give this option to user programs.
  - Also, the address needs to be “pinned” (cannot be evicted) in memory.
I/O Devices

- Block devices:
  - usually stores information in fixed size blocks
  - you read or write an individual block independently of others by giving it an address.
  - E.g., disks, tapes, ...

- Character devices:
  - delivers or accepts streams of characters
  - Not addressable.
  - E.g., terminals, printers, mouse, network interface.
Principles of I/O Software

- Provide device independence:
  - same programs should work with different devices.
  - uniform naming -- i.e., name shouldn't depend on the device.
  - error handling, handle it as low as possible and only if unavoidable pass it on higher.
  - synchronous (blocking) vs. asynchronous (interrupt driven). Even though I/O devices are usually async, sync is easier to program
## Device Characteristics

<table>
<thead>
<tr>
<th>aspect</th>
<th>variation</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>data-transfer mode</td>
<td>character block</td>
<td>terminal disk</td>
</tr>
<tr>
<td>access method</td>
<td>sequential random</td>
<td>modem CD-ROM</td>
</tr>
<tr>
<td>transfer schedule</td>
<td>synchronous, asynchronous</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>sharing</td>
<td>dedicated, sharable</td>
<td>tape keyboard</td>
</tr>
<tr>
<td>device speed</td>
<td>latency, seek time, transfer rate</td>
<td></td>
</tr>
<tr>
<td>I/O direction</td>
<td>read only, write only, read-write</td>
<td>CD-ROM graphics controller disk</td>
</tr>
</tbody>
</table>
I/O Software

- A layered approach:
  - Lowest layer (device dependent): Device drivers
  - Middle layer: Device independent OS software
  - High level: User-level software/libraries
- The first 2 are part of the kernel.
Device Drivers

- Accept abstract requests from device-independent OS software, and service those requests.
- There is a device driver for each “device”
- However, the interface to all device drivers is the same.
  - `Open()`, `close()`, `read()`, `write()`, `interrupt()`, `ioctl()`, …
Device-independent OS Layer

- Device naming and protection
  - Each device is given a (major#, minor#) – present in the i-node for that device
  - Major# identifies the driver
  - Minor# is passed on to the driver (to handle sub-devices)

- Does buffering/caching
- Uses a device-independent block size
- Handles error reporting in a device-independent fashion.
Putting things together (UNIX)

- User calls open("/dev/tty0","w") which is a system call.
- OS traverses file system to find the i-node of tty0.
- This should contain the (major #, minor #).
- Check permissions to make sure it is allowed.
- An entry is created in OFDT, and a handle is returned to the user.
- When user calls write(fd, ....) subsequently, index into OFDT, get major/minor #s.
I/O and Kernel Objects

- File descriptor
  - Per-process open-file table
  - User-process memory

- System-wide open-file table
  - File-system record
    - Inode pointer
    - Pointer to read and write functions
    - Pointer to select function
    - Pointer to ioctl function
    - Pointer to close function
  - Networking (socket) record
    - Pointer to network info
    - Pointer to read and write functions
    - Pointer to select function
    - Pointer to ioctl function
    - Pointer to close function
  - Kernel memory

- Active inode table
- Network-information table
• Copy the bytes pointed to by the pointer given by user, into a kernel “pinned” (which is not going to be paged out) buffer.

• Use the above data structure, to find the relevant driver’s write() routine, and call it with the pinned buffer address, and other relevant parameters.

• For a write, one can possibly return back to user even if the write has not propagated. On a read (for an input device), the driver would program the device, and block the activity till the interrupt.
• This was a character device. In a block device, before calling the driver, check the buffer/cache that the OS is maintaining to see if the request can be satisfied before going to the driver itself.

• The lookup is done based on (major #, logical block id).

• Thus it is a unified device-independent cache across all devices.
• This is all for the user referring to an I/O device (/dev/*).

• Note: It is not very different when the user references a normal file. In that case, we have already seen how the file system generates a request in the form of a logical block id, which is then sent to the driver where the specified file system resides (disk/CD/…).
Life Cycle of an I/O Request

1. **request I/O**
   - user process
   - system call

2. **can already satisfy request?**
   - yes: transfer data (if appropriate) to process, return completion or error code
   - no: send request to device driver, block process if appropriate

3. **send request to device driver, block process if appropriate**
   - device controller commands
   - monitor device, interrupt when I/O completed

4. **device controller**
   - receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

5. **device handler**
   - determine which I/O completed, indicate state change to I/O subsystem

6. **I/O completed, generate interrupt**
   - I/O completed, input data available, or output completed
   - return from system call
Summary

- Input/Output
  - The OS Manages Device Usage
  - Communication
  - I/O Subsystem
Final Exam

- Tuesday, June 7 at 8 AM
- 2 hours
- Structure: similar to previous exam in terms of question layout, may be similar length or possibly a bit longer
- Questions:
  - some technical details
  - some conceptual questions
- Material: slides, text, class discussion, homeworks, projects
What we’ve covered

- Everything from the first part of the course is fair game
  - but probably de-emphasized a bit - bit twiddling probably not required but have a strong idea of the concepts

- Chapter 6 -- Synchronization
- Chapter 7 -- Deadlocks
- Chapter 8 -- Main Memory (Physical)
- Chapter 9 -- Virtual Memory
- Chapter 10 -- File System Interface
- Chapter 11 -- File System Implementation
- Chapter 12 -- Storage
- Chapter 13 -- I/O
Synchronization

- Little’s Law -- Final
- Problems
- Synchronization Requirements
- Disabling Interrupts
- Busy-wait/Spinlock solutions
  - Related to properties
- Hardware-enabled Solutions
- OS-supported
Requirements for Solution

1. Mutual Exclusion - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. Progress - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the N processes.
Synchronization

• Hardware Enabled Solutions
• OS-supported Solutions
  ‣ Mutex
  ‣ Semaphores
  ‣ Condition Variables
• Apply these to code
• Classic Synchronization Problems
Semaphores

- You are given a data-type Semaphore_t.
- On a variable of this type, you are allowed
  - P(Semaphore_t) -- wait
  - V(Semaphore_t) – signal
- Intuitive Functionality:
  - Logically one could visualize the semaphore as having a counter initially set to 0.
  - When you do a P(), you decrement the count, and need to block if the count becomes negative.
  - When you do a V(), you increment the count and you wake up 1 process from its blocked queue if not null.
Deadlocks

- Necessary Conditions
- Safe States
- Resource Allocation Graph
- Deadlock Prevention
  - Safe States
- Deadlock Detection
  - Detection Algorithm
  - Recovery
Necessary Conditions for a Deadlock

- Mutual exclusion: The requesting process is delayed until the resource held by another is released.
- Hold and wait: A process must be holding at least 1 resource and must be waiting for 1 or more resources held by others.
- No preemption: Resources cannot be preempted from one and given to another.
- Circular wait: A set (P0,P1,…,Pn) of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for …. by P2, … Pn is waiting for … held by P0.
Deadlock Prevention Example

5 processes, 3 resource types A (10 instances), B (5 instances), C (7 instances)

<table>
<thead>
<tr>
<th>MaxNeeds</th>
<th>Allocated</th>
<th>StillNeeds</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>A B C</td>
<td>A B C</td>
</tr>
<tr>
<td>P0 7 5 3</td>
<td>P0 0 1 0</td>
<td>P0 7 4 3</td>
</tr>
<tr>
<td>P1 3 2 2</td>
<td>P1 2 0 0</td>
<td>P1 1 2 2</td>
</tr>
<tr>
<td>P2 9 0 2</td>
<td>P2 3 0 2</td>
<td>P2 6 0 0</td>
</tr>
<tr>
<td>P3 2 2 2</td>
<td>P3 2 1 1</td>
<td>P3 0 1 1</td>
</tr>
<tr>
<td>P4 4 3 3</td>
<td>P4 0 0 2</td>
<td>P4 4 3 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
</tr>
<tr>
<td>3 3 2</td>
</tr>
</tbody>
</table>

This state is safe, because there is a reduction sequence <P1, P3, P4, P2, P0> that can satisfy all the requests.

Exercise: Formally go through each of the steps that update these matrices for the reduction sequence.
### Deadlock Detection Example

5 processes, 3 resource types A (7 instances), B (2 instances), C (6 instances)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

This state is NOT deadlocked.

By applying algorithm, the sequence <P0, P2, P3, P1, P4> will result in Done[i] being TRUE for all processes.
Main Memory

- Swapping
- Allocation
  - Contiguous, Non-contiguous (paging)
  - Algorithms
- Fragmentation
  - Internal, External
- Page-tables, TLBs
  - virtual-physical translation
  - Page table structure, entries
Memory Allocation

Queue of waiting requests/jobs

Question: How do we perform this allocation?
• Programs are provided with a virtual address space (say 1 MB).

• Role of the OS to fetch data from either physical memory or disk.
  ‣ Done by a mechanism called (demand) paging.

• Divide the virtual address space into units called “virtual pages” each of which is of a fixed size (usually 4K or 8K).
  ‣ For example, 1M virtual address space has 256 4K pages.
Page Tables

Virtual Address

Virtual Page # | Offset in Page
---|---
Virtual Page # | Offset in Page

<table>
<thead>
<tr>
<th>VP #</th>
<th>PP #</th>
<th>Present</th>
</tr>
</thead>
<tbody>
<tr>
<td>vpi</td>
<td>ppi</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vpi</td>
<td>ppi</td>
<td></td>
</tr>
</tbody>
</table>

Physical Address

Physical Page # | Offset in Page
---|---
Physical Page # | Offset in Page
Virtual Memory

- Page Fault Handling
  - Performance Estimations
- Memory Initialization
- Page Replacement
  - Algorithms
  - Belady’s Anomaly
- Uses of Virtual Memory
  - COW, Shared Pages, Memory-mapped Files
- Thrashing
Page Fault

• If there is a reference to a page, first reference to that page will trap to operating system:
  ‣ page fault

• Operating system looks at another table to decide:
  ‣ Invalid reference -- abort
  ‣ Just not in memory

• Get empty frame

• Swap page into frame

• Reset tables

• Set validation bit = ∨

• Restart the instruction that caused the page fault
File Systems

- File System Concepts
  - Files, Directories, File Systems
  - Operations and Usage
  - Remote File Systems

- File System Implementation
  - What’s on the disk? How’s it formatted?
  - What’s in memory? How’s it represented?

- File System Usage
  - Get a file
  - Caching
  - Free Space
  - Recovery
File System Mounting
Mass Storage and I/O

- Disk scheduling algorithms
- Access and transfer time (and their components)
- Real schedulers and their differences
- Buses
- DMA
Looking forward...

• What is the future of operating system design and research?
  ‣ Mobile phones (Android is a big research platform right now)
  ‣ Clouds (Amazon AWS and other services)
  ‣ Both cases: distributed operation is becoming ever more critical
    ▪ IPC within system and to other components that comprise logical systems
    ▪ ubiquitous computing and prevalent network connectivity

• SECURITY
Planning your UO career

- If you are interested in what you’ve learned in this class and want to consider learning more about systems concepts, you may also think about the following courses:
  - CIS 432: Networking
  - CIS 433: Computer and Network Security

- Also: seminars and reading groups

- Think about your future and what you want
  - Take advantage of resources you have at your disposal while you’re a student
Thanks to...

- Colleagues and mentors here at UO, at the Pennsylvania State University, the University of Pennsylvania, and Columbia University
  - provided basis for many course materials
  - special thanks: Trent Jaeger and Adam Aviv
- Kaveh Kazemi: GTF for this course
- Most importantly:
  - You! Those of you who stuck it out to the bitter end
- Have a great summer! See you Tuesday!