CS162
Operating Systems and Systems Programming
Lecture 12

Kernel/User, I/O

March 6, 2013

Anthony D. Joseph

http://inst.eecs.berkeley.edu/~cs162

Review: Demand Paging Mechanisms

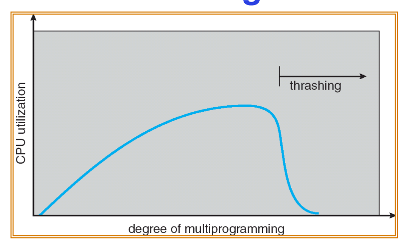
- Demand paging leverages several PTE bits
  - "V": Valid / Not Valid
    - "V = 1": Valid ⇒ Page in memory, PTE points at physical page
    - "V = 0": Not Valid ⇒ Page not in memory; use info in PTE to find page on disk if necessary
  - "D = 1": Page modified ⇒ Need to write it back to disk before replacing it
  - "U = 1": Page referenced ⇒ Give page a second chance before being replaced when using Second Chance algorithm
- Some other PTE bits:
  - "R/W": specifies whether the page can be modified or is read only
  - Page Access Count: implement more accurate LRU algorithms

Goals for Today

- Finish Demand Paging: Trashing and Working Sets
- Dual Mode Operation: Kernel versus User Mode
- I/O Systems
  - Hardware Access
  - Device Drivers
- Disk Performance
  - Hardware performance parameters

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from my lecture notes by Kubiatowicz.

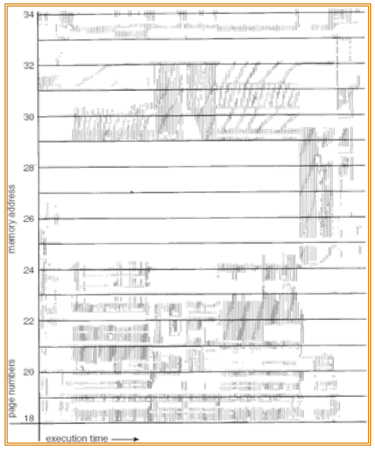
Thrashing



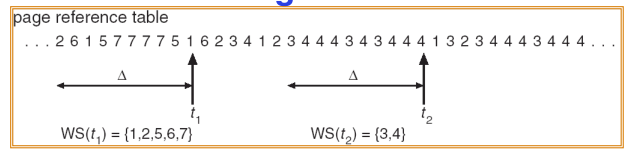
- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model

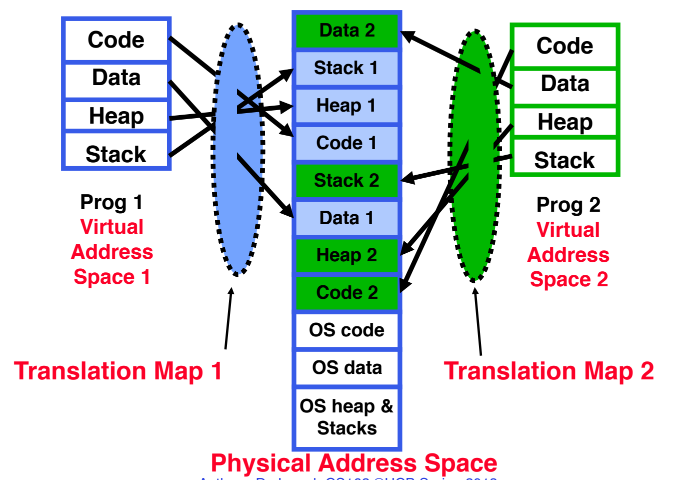


- Δ ≡ working-set window ≡ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if Δ = ∞ ⇒ will encompass entire program
- $D = \Sigma|WS_i|$ ≡ total demand frames
- if $D >$ *physical memory* ⇒ Thrashing
  - Policy: if $D >$ physical memory, then suspend/swap out processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages "around" the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
  - Tradeoff: Prefetching may evict other in-use pages for never-used prefetched pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Review: Example of General Address Translation



Dual-Mode Operation

- Can an application modify its own translation maps or PTE bits?
  - If it could, could get access to all of physical memory
  - Has to be restricted somehow
- To assist with protection, hardware provides at least two modes (Dual-Mode Operation):
  - "Kernel" mode (or "supervisor" or "protected")
  - "User" mode (Normal program mode)
  - Mode set with bits in special control register only accessible in kernel-mode
- Intel processors actually have four "rings" of protection:
  - PL (Privilege Level) from 0 – 3
    - » PL0 has full access, PL3 has least
  - Typical OS kernels on Intel processors only use PL0 ("kernel") and PL3 ("user")

For Protection, Lock User-Programs in Asylum

- Idea: Lock user programs in padded cell with no exit or sharp objects
  - Cannot change mode to kernel mode
  - Cannot modify translation maps
  - Limited access to memory: cannot adversely effect other processes
  - What else needs to be protected?

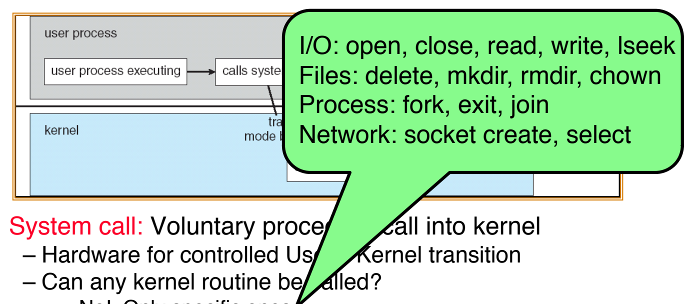
- A couple of issues
  - How to share CPU between kernel and user programs?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): getting into cell
    - » User→ OS (user → kernel mode): getting out of cell

How to get from Kernel→User

- What does the kernel do to create a new user process?
  - Allocate and initialize process control block
  - Read program off disk and store in memory
  - Allocate and initialize translation map
    - » Point at code in memory so program can execute
    - » Possibly point at statically initialized data
  - Run Program:
    - » Set machine registers
    - » Set hardware pointer to translation table
    - » Set processor status word for user mode
    - » Jump to start of program
- How does kernel switch between processes (we learned about this!) ?
  - Same saving/restoring of registers as before
  - Save/restore hardware pointer to translation map

User→Kernel (System Call)

- Can't let inmate (user) get out of padded cell on own
  - Would defeat purpose of protection!
  - So, how does the user program get back into kernel?



- System call: Voluntary proce... call into kernel
  - Hardware for controlled Us... Kernel transition
  - Can any kernel routine be ...alled?
    - » No! Only specific ones
  - System call ID encoded into system call instruction
    - » Index forces well-defined interface with kernel

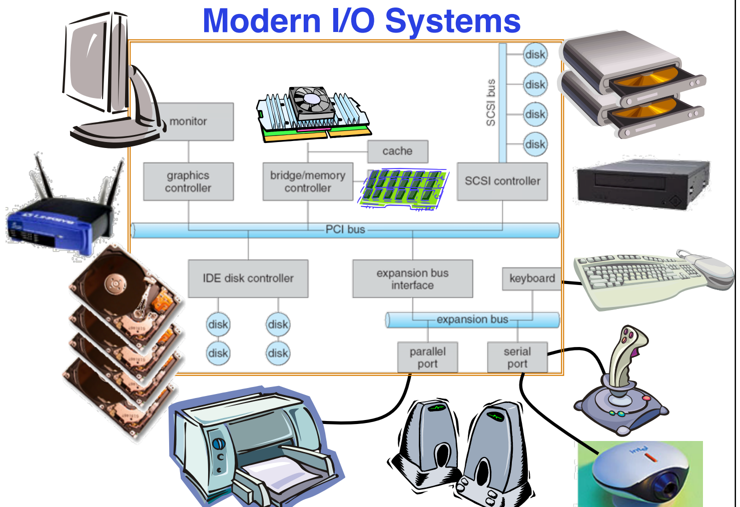
System Call (cont'd)

- Are system calls the same across operating systems?
  - Not entirely, but there are lots of commonalities
  - Also some standardization attempts (POSIX)
- What happens at beginning of system call?
  - On entry to kernel, sets system to kernel mode
  - Handler address fetched from table, and Handler started
- System Call argument passing:
  - In registers (not very much can be passed)
  - Write into user memory, kernel copies into kernel memory
  - *Every argument must be explicitly checked!*

User→Kernel (Exceptions: Traps and Interrupts)

- System call instr. causes a synchronous exception (or "trap")
  - In fact, often called a software "trap" instruction
- Other sources of *Synchronous Exceptions:*
  - Divide by zero, Illegal instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault
- Interrupts are *Asynchronous Exceptions*
  - Examples: timer, disk ready, network, etc.…
  - Interrupts can be disabled, traps cannot!
- SUMMARY – On system call, exception, or interrupt:
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.

Modern I/O Systems



What is the Role of I/O?

- Without I/O, computers are useless (disembodied brains?)
- But… thousands of devices, each slightly different
  - How can we standardize the interfaces to these devices?
- Devices unreliable: media failures and transmission errors
  - How can we make them reliable???
- Devices unpredictable and/or slow
  - How can we manage them if we don't know what they will do or how they will perform?

Administrivia

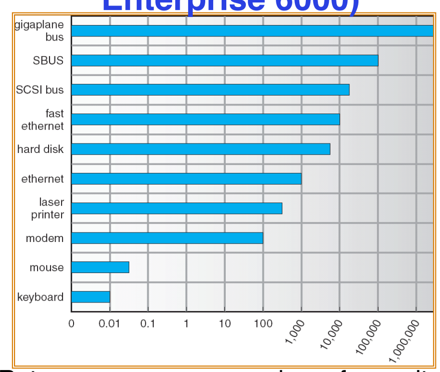
- Quiz #1 was yesterday – we'll have more and drop lowest one
- Project 2 Design Doc due tomorrow Thursday 3/7 at 11:59PM
- Midterm exam next Wednesday 3/13 4-5:30pm **in 2 rooms**
  - 145 Dwinelle for last names beginning with A-H
  - 245 Li Ka Shing for last names beginning with I-Z
  - Closed book, no calculators
  - Covers lectures/readings #1-12 (today) and project one
  - One double-sided ***handwritten*** page of notes allowed
  - Review session: **105 North Gate, Saturday March 9, 1-3PM**
- Please fill the anonymous course survey at https://www.surveymonkey.com/s/9DK2VVJ
  - We'll try to make changes *this* semester based on your feedback

5min Break

Operational Parameters for I/O

- Data granularity: Byte vs. Block
  - Some devices provide single byte at a time (*e.g.,* keyboard)
  - Others provide whole blocks (*e.g.,* disks, networks, etc.)
- Access pattern: Sequential vs. Random
  - Some devices must be accessed sequentially (*e.g.,* tape)
  - Others can be accessed randomly (*e.g.,* disk, cd, etc.)
- Transfer mechanism: Polling vs. Interrupts
  - Some devices require continual monitoring
  - Others generate interrupts when they need service

Example Device-Transfer Rates (Sun Enterprise 6000)



- Device Rates vary over many orders of magnitude
  - System better be able to handle this wide range
  - Better not have high overhead/byte for fast devices!
  - Better not waste time waiting for slow devices

The Goal of the I/O Subsystem

- Provide uniform interfaces, despite wide range of different devices
  - This code works on many different devices:

```
FILE fd = fopen("/dev/something","rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd, "Count %d\n",i);
}
close(fd);
```

  - Why? Because code that controls devices ("device driver") implements standard interface
- We will try to get a flavor for what is involved in actually controlling devices in rest of lecture
  - Can only scratch surface!

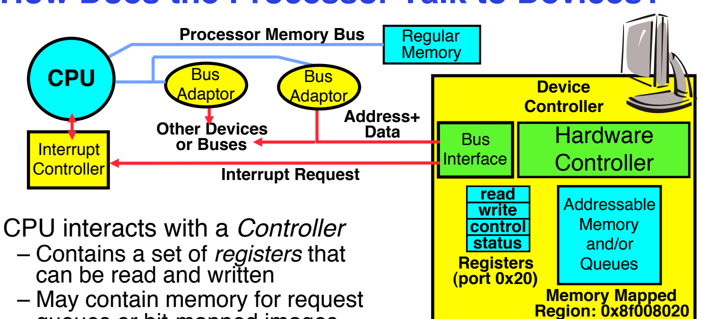
Want Standard Interfaces to Devices

- Block Devices: *e.g.,* disk drives, tape drives, DVD-ROM
  - Access blocks of data
  - Commands include `open()`, `read()`, `write()`, `seek()`
  - Raw I/O or file-system access
  - Memory-mapped file access possible
- Character/Byte Devices: *e.g.,* keyboards, mice, serial ports, some USB devices
  - Single characters at a time
  - Commands include `get()`, `put()`
  - Libraries layered on top allow line editing
- Network Devices: *e.g.,* Ethernet, Wireless, Bluetooth
  - Different enough from block/character to have own interface
  - Unix and Windows include socket interface
    - » Separates network protocol from network operation
    - » Includes `select()` functionality

How Does User Deal with Timing?

- Blocking Interface: "Wait"
  - When request data (*e.g.,* `read()` system call), put process to sleep until data is ready
  - When write data (*e.g.,* `write()` system call), put process to sleep until device is ready for data
- Non-blocking Interface: "Don't Wait"
  - Returns quickly from read or write request with count of bytes successfully transferred to kernel
  - Read may return nothing, write may write nothing
- Asynchronous Interface: "Tell Me Later"
  - When requesting data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user
  - When sending data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

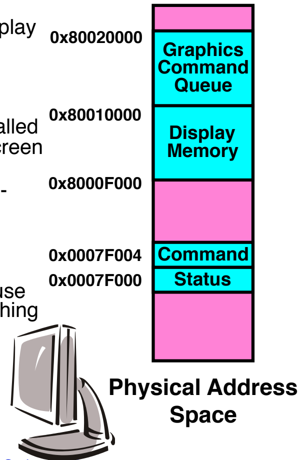
How Does the Processor Talk to Devices?



- CPU interacts with a *Controller*
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images
- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - I/O instructions: in/out instructions (e.g., Intel's 0x21,AL)
  - Memory mapped I/O: load/store instructions
    - » Registers/memory appear in physical address space
    - » I/O accomplished with load and store instructions

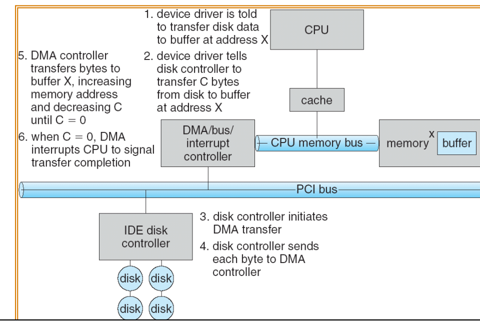
Example: Memory-Mapped Display Controller

- Memory-Mapped:
  - Hardware maps control registers and display memory into physical address space
    - » Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the "frame buffer") changes image on screen
    - » Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - » Say enter a set of triangles that describe some scene
    - » Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - » Say render the above scene
    - » Addr: 0x0007F004
- Can protect with address translation



Transferring Data To/From Controller

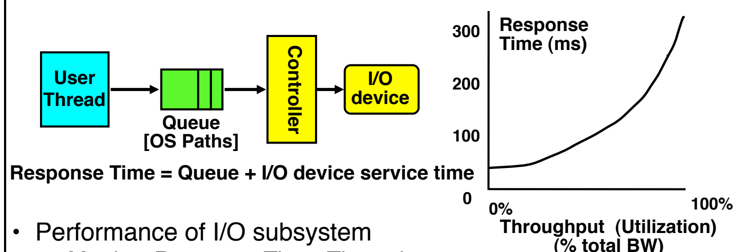
- Programmed I/O:
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
- Direct Memory Access:
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
- Sample interaction with DMA controller (from book):



I/O Device Notifying the OS

- The OS needs to know when:
  - The I/O device has completed an operation
  - The I/O operation has encountered an error
- I/O Interrupt:
  - Device generates an interrupt whenever it needs service
  - Pro: handles unpredictable events well
  - Con: interrupts relatively high overhead
- Polling:
  - OS periodically checks a device-specific status register
    - » I/O device puts completion information in status register
  - Pro: low overhead
  - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations
- Actual devices combine both polling and interrupts
  - For instance – High-bandwidth network adapter:
    - » Interrupt for first incoming packet
    - » Poll for following packets until hardware queues are empty

I/O Performance



- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization approaches 100%

Quiz 12.1: I/O

- Q1: True _ False _ With an asynchronous interface, the writer may need to block until the data is written
- Q2: True _ False _ Interrupts are more efficient than polling for handling very frequent requests
- Q3: True _ False _ Segmentation fault is an example of synchronous exception (trap)
- Q4: True _ False _ DMA is more efficient than programmed I/O for transferring large volumes of data
- Q5: In a I/O subsystem the queuing time for a request is 10ms and the request's service time is 40ms. Then the total response time of the request is ___ ms

Quiz 12.1: I/O

- Q1: True _ False **X** With an asynchronous interface, the writer may need to block until the data is written
- Q2: True _ False **X** Interrupts are more efficient than polling for handling very frequent requests
- Q3: True **X** False _ Segmentation fault is an example of synchronous exception (trap)
- Q4: True **X** False _ DMA is more efficient than programmed I/O for transferring large volumes of data
- Q5: In a I/O subsystem the queuing time for a request is 10ms and the request's service time is 40ms. Then the total response time of the request is **50** ms

Summary

- Dual-Mode
  - Kernel/User distinction: User restricted
  - User→Kernel: System calls, Traps, or Interrupts
- I/O Devices Types:
  - Many different speeds (0.1 bytes/sec to GBytes/sec)
  - Different Access Patterns: block, char, net devices
  - Different Access Timing: Non-/Blocking, Asynchronous
- I/O Controllers: Hardware that controls actual device
  - CPU accesses thru I/O insts, ld/st to special phy memory
  - Report results thru interrupts or a status register polling
- Device Driver: Device-specific code in kernel