CS162
Operating Systems and Systems Programming
Lecture 12

Kernel/User, I/O

October 14, 2013
Anthony D. Joseph and John Canny
http://inst.eecs.berkeley.edu/~cs162

Quiz 12.1: Paging

- Q1: True _ False _ Inverse Page Tables (IPT) table size grows with virtual memory allocation.
- Q2: True _ False _ IPTs get slower when physical memory is mostly allocated.
- Q3: True _ False _ Increasing the number of frames for LRU page replacement gives the same or lower miss rate.
- Q4: True _ False _ Increasing the number of frames for Second Chance page replacement gives the same or lower miss rate.
- Q5: True _ False _ The Clock Algorithm requires the OS to keep track of page accesses as well as faults .

Quiz 12.1: Paging

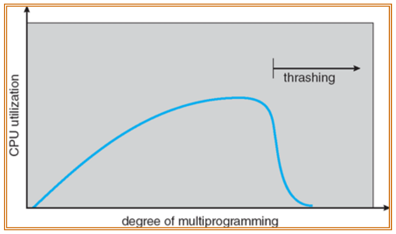
- Q1: True _ False X Inverse Page Tables (IPT) table size grows with virtual memory allocation.
- Q2: True X False _ IPTs get slower when physical memory is mostly allocated.
- Q3: True X False _ Increasing the number of frames for LRU page replacement gives the same or lower miss rate.
- Q4: True _ False X Increasing the number of frames for Second Chance page replacement gives the same or lower miss rate.
- Q5: True X False _ The Clock Algorithm requires the OS to keep track of page accesses as well as faults .

Goals for Today

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

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.

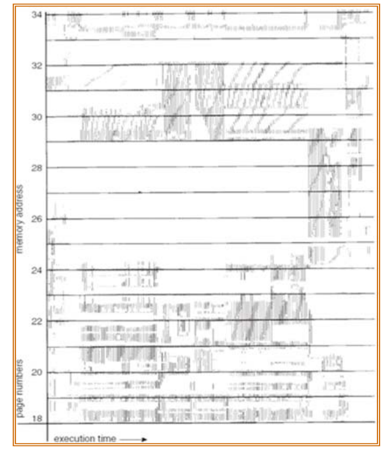
Thrashing



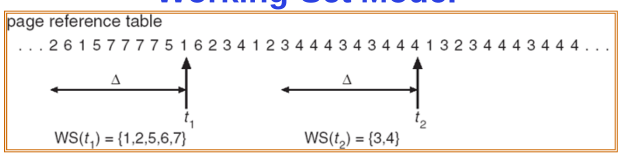
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  – low CPU utilization
  – operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  – How do we detect Thrashing?
  – What is best response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

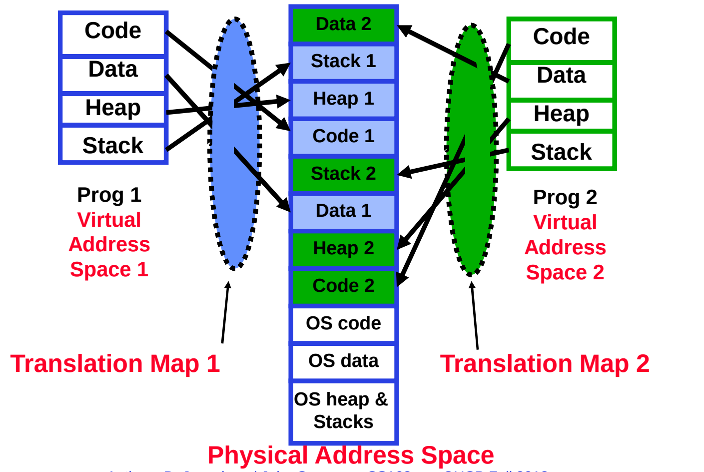


- Δ ≡ working-set window ≡ fixed number of page references
  – Example: 10,000 accesses
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  – if Δ too small will not encompass entire locality
  – if Δ too large will encompass several localities
  – 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?

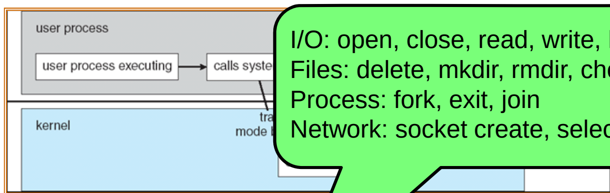
- A couple of issues
  - How to share CPU between kernel and user programs?
  - How does one switch between kernel and user modes?
    - » OS → user (kernel → user mode): getting into cell
    - » User→ OS (user → kernel mode): getting out of cell

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?
  - 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?



I/O: open, close, read, write, lseek
Files: delete, mkdir, rmdir, chown
Process: fork, exit, join
Network: socket create, select

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

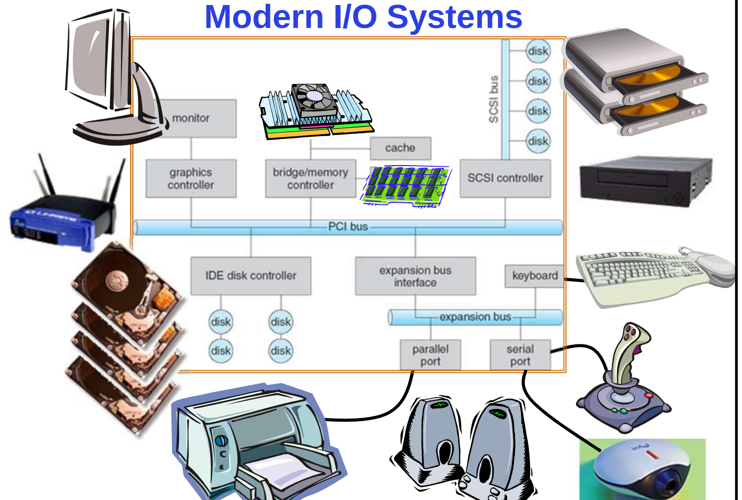
System Call (cont'd)

- 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

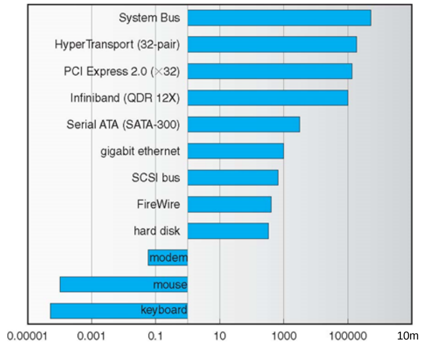
- Midterm #1 is Monday Oct 21 5:30-7pm in 145 Dwinelle (A-L) and 2060 Valley LSB (M-Z)
  - Closed book, double-sided ***handwritten*** page of notes, no calculators, smartphones, Google glass etc.
  - Covers lectures #1-13 (Disks/SSDs, Filesystems), readings, handouts, and projects 1 and 2
  - Review session **390 Hearst Mining, Fri October 18, 5-7 PM**
- Project 2 design docs due **Thursday, 11:59pm**
- Course Survey is online: https://www.surveymonkey.com/s/FSW3HVJ

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 in Mb/s (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

Kernel vs User-level I/O

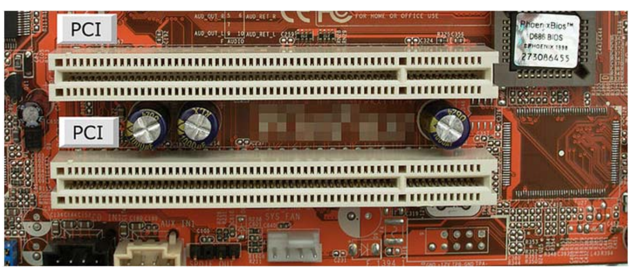
- Both are popular/practical for different reasons:
  - Kernel-level drivers for critical devices that must keep running, e.g. display drivers.
    - » Programming is a major effort, correct operation of the rest of the kernel depends on correct driver operation.
  - User-level drivers for devices that are non-threatening, e.g USB devices in Linux (libusb).
    - » Provide higher-level primitives to the programmer, avoid every driver doing low-level I/O register tweaking.
    - » The multitude of USB devices can be supported by Less-Than-Wizard programmers.
    - » New drivers don't have to be compiled for each version of the OS, and loaded into the kernel.

Kernel vs User-level Programming Styles

- Kernel-level drivers
  - Have a much more limited set of resources available:
    - » Only a fraction of libc routines typically available.
    - » Memory allocation (e.g. Linux kmalloc) much more limited in capacity and required to be physically contiguous.
    - » Should avoid blocking calls.
    - » Can use asynchrony with other kernel functions but tricky with user code.
- User-level drivers
  - Similar to other application programs but:
    - » Will be called often – should do its work fast, or postpone it – or do it in the background.
    - » Can use threads, blocking operations (usually much simpler) or non-blocking or asynchronous.

PCI Bus evolution

- PCI started life out as a bus
  - 32 physical bits double for address/data
- But parallel busses have many limitations
  - multiplexing address/data for many requests
  - Slowest device must be able to tell what's happening
  - ➔ Bus speed is set to that of the slowest device



PCI Express "Bus"

- No longer a parallel bus
- Really a **collection of fast serial channels** or "lanes"
- Devices can use as many as they need to achieve a desired bandwidth
- Slow devices don't have to share with fast ones.
- Both motherboard slots and daughter cards are sized for the number of lanes, x4, x8, or x16
- Speeds (in an x16 configuration):
  - **v1.x**: 4 GB/s (40 GT/s)
  - **v2.x**: 8 GB/s (80 GT/s)
  - **v3.0**: 15.75 GB/s (128 GT/s)
  - **v4.0**: 31.51 GB/s (256 GT/s)

  3.0+ Speeds are competitive with **block memory-to-memory** operations on the CPU.

PCI Express Bus

In practice PCI is used as the interface to many other interconnects on a PC:

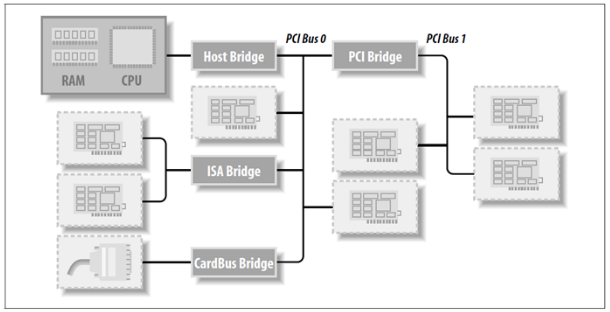


Figure 12-1. Layout of a typical PCI system

Figure from "Linux Device Drivers," 3rd Ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

PCI Express Interface (Linux)

- One of the successes of device abstraction in Linux was the ability to migrate from PCI to PCI-Express.
- Although the physical interconnect changed completely, the old API still worked.
- Drivers written for older PCI devices still worked, because of the standardized API for both models of the interface.
- PCI register map:

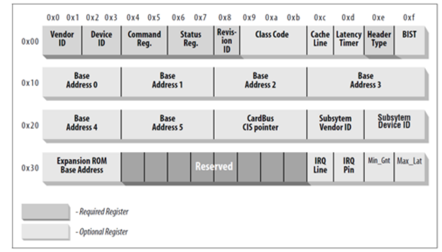
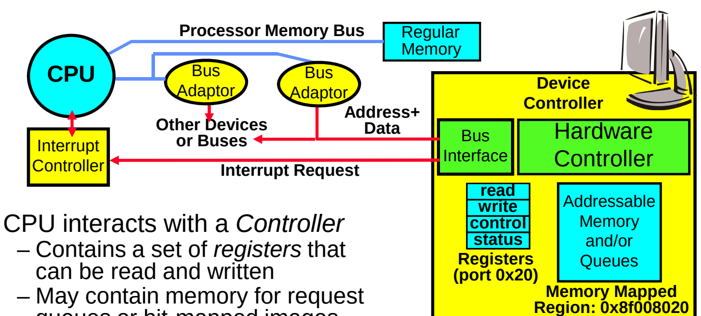


Figure 12-2. The standardized PCI configuration registers

Figure from "Linux Device Drivers," 3rd Ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

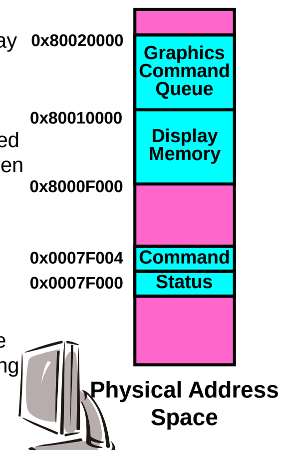
How Does the Processor Talk to Devices?



- 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

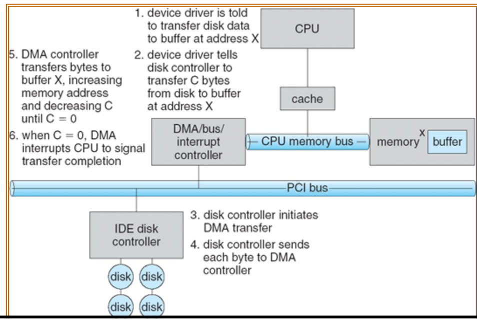
Example: Memory-Mapped Display Controller

- 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):

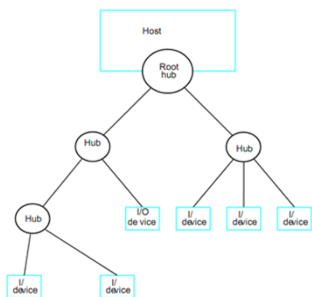


I/O Device Notifying the OS

- 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

USB Topology and Mastering

- USB is a complex standard with a simple communication model.
- It's a complex (tree) topology, but the CPU is always the master.



USB Topology and Mastering

- Each device exposes one or more "endpoints" for communication, control, or interrupts.

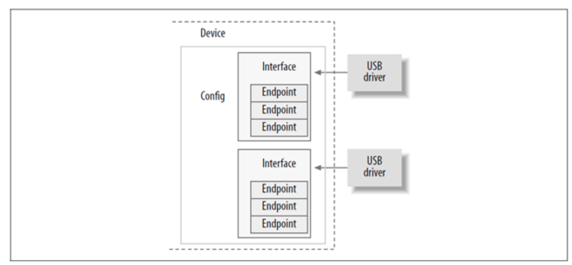


Figure 13-2. USB device overview

Figure from "Linux Device Drivers," 3rd Ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

USB Topology and Mastering

- Each device exposes one or more "endpoints" for communication, control, or interrupts.
- The driver infrastructure (libusb) takes care of actual communication and provides a high-level (blocking) bulk communication primitive.

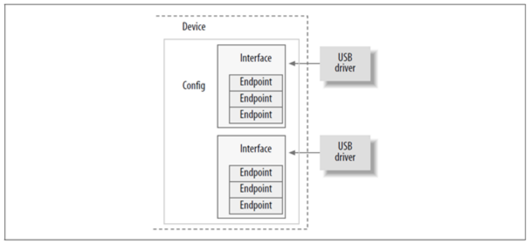
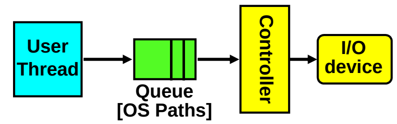


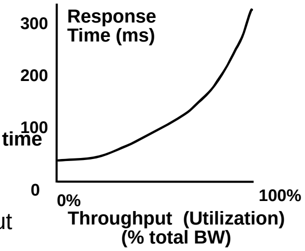
Figure 13-2. USB device overview

Figure from "Linux Device Drivers," 3rd Ed, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman

I/O Performance

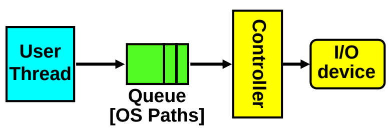


Response Time = Queue + I/O device service time

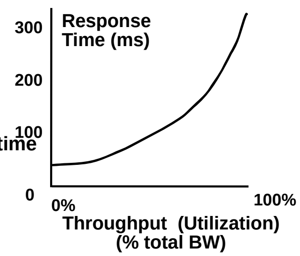


- Performance of I/O subsystem
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » I/O device service time
- Queuing behavior:
  - Can lead to big increases of latency as utilization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

I/O Performance



Response Time = Queue + I/O device service time



- Solutions?
  - Make everything faster ☺
  - Decouple systems
    - » multiple independent buses
    - » or tree-structured buses with higher root bandwidth
  - Buffering (as long as you don't have to wait for it) and spooling
    - » Give the processor something to do that gets the data "closer" to its endpoint.

Quiz 12.2: 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 12.2: 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