Operating Systems

CSC-4320/6320 –Summer 2014

Dustin Kempton (dkempton1@cs.gsu.edu)

Why are we studying this?

- Unless you land an interesting job, you probably won't be developing an OS
- It is still important to understand what you are using
  - The key thing to remember is, understanding how the OS works helps you develop (better) programs, understand what you can and cannot do, understand performance, and why some OS may be better/worse than another in some situations
- OS concepts are fundamental and re-usable when implementing programs that are not operating systems
  - The OS is the most interesting complex software system and lessons from OSes can be applied in many other contexts

What is an Operating System?

- **Definition:** Software that creates an environment so computer can perform useful tasks
  - **Generally speaking:** Everything that ships with the computer
  - **More specifically:** Software that runs at all times (the kernal). Everything else is either a system utility or an application program.

What is an Operating System?

- It is a resource abstractor and allocator
  - It defines a set of logical resources that correspond to hardware resources, and a set of well defined operations on logical resources
    - e.g., physical resources: CPU, Disks, RAM
    - e.g., logical resources: processes, files, arrays
  - The OS determines who (which program) gets what resource and when

Goals of an Operating System

- May include some or all of
  - Providing a good user experience
  - Responsive to user commands
  - Minimize idle time and maximize throughput
  - Respond to real time constraints
  - Maximize connectivity
  - Efficiently sharing distributed resources

Types of Operating Systems

- **Batch:** Small resident kernel, sequences jobs for one or more batch streams using a job control interface. *Goal:* minimize setup time, minimize idle time.
- **Real time:** Often part of special purpose embedded systems with no user interface. *Goal:* satisfy well-defined hard or soft time constraints
- **Time shared:** Responds to multiple users at workstations. Scheduling is based on real time clock "slices." *Goal:* Responsive to users
- **Desktop systems:** Single users, with multitasked and multi-programming capabilities. *Goal:* user friendly and responsive
- **Multiprocessor:** Tightly coupled processors sharing memory and peripherals. *Goal:* maximize throughput and reliability (redundancy)
- **Multi-computer:** Loosely coupled, often heterogeneous systems working cooperatively. *Goal:* efficient and secure client-server or peer-to-peer inter-computer communication.
- **Grid-based:** loosely coupled, heterogeneous, geographically separated. Communicate through LAN or WAN; peer-to-peer or client-server. *Goal:* utilize distributed resources to address grand challenge problems
- **Handheld systems:** Slow, limited in processor speed, available power and user interfaces. *Goal:* support user-friendly portable applications maximize wireless connectivity, optimize battery life, optimize execution efficiency

Real Time System Constraints

- **Hard** (System fails if constraints not satisfied)
  - Guarantee that real time constraints complete on-time
  - Not supportable by general purpose operating systems
  - Data is stored in ROM (or flash, or EEPROM) or in short term memory; no secondary storage, virtual memory, or advanced operating system featurs.
  - *Examples:* Sensor control inputs on wepon systems, medical imaging, robotics (industrial and otherwise), automatic breaking and traction control systems in cars, etc.
- **Soft** (Best effort made to meet constraints)
  - Priority-based, delays are bounded
  - Not useful for hard restraints like when a robotic arm movement cannot be stopped on time
  - Examples: streaming of audio and video, web cams, stock market activity

Parallel Processing

- Multiprocessors (tightly coupled)
  - Can be less expensive because they share peripherals
  - Can be more reliable:
    - Service continues upon processor failure (*graceful degradation*)
    - One processor picks up functionality upon failure (*fault tolerant*)
  - Operation can be
    - *Asymmetric:* Each processor performs a separate task
    - *Symmetric:* All processors perform the same tasks simultaneously
- Clustered systems (loosely coupled via local LAN)
  - High availability service
  - *Asymmetric:* one machine is in hot-standby mode
  - *Symmetric:* multiple host running and monitoring each other
- Grids: loosely connected and geographically seperated

Multi-Programming

- Multi-Programming: Modern OSes allow multiple “jobs” (running programs) to reside in memory simultaneously
  - The OS picks and begins to execute one of the jobs in memory
  - When the job has to wait for “something”, then the OS picks another job to run
  - This is called a context-switch, and improves productivity
- We are used to this now, but it wasn’t always the case
  - Single-user mode
    - Terrible productivity (while you “think”, nobody else is using the machine)
  - Batch Processing (jobs in a queue)
    - Low productivity (CPU idle during I/O operations)

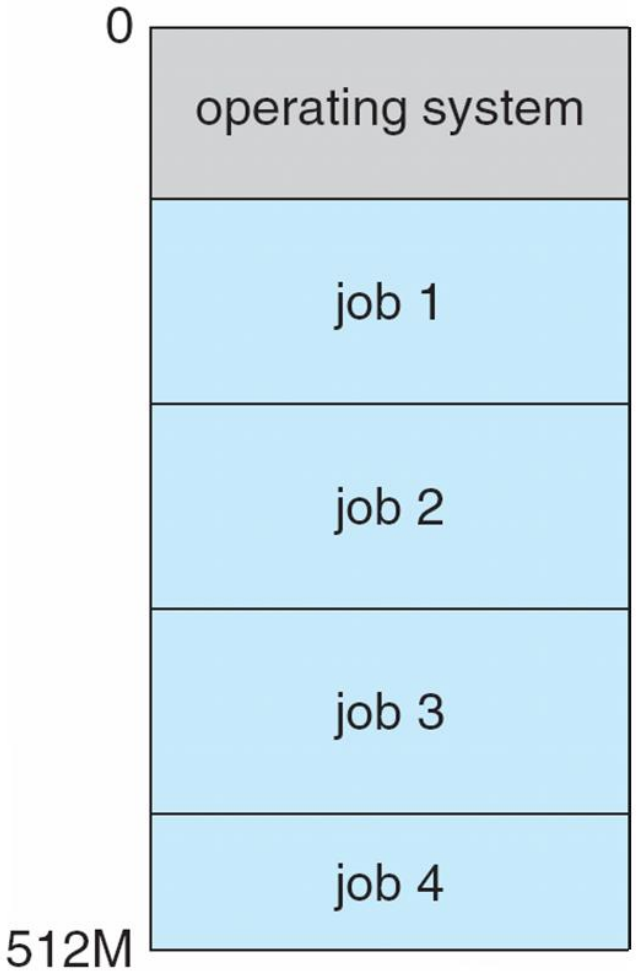
Time-Sharing

- Time-Sharing: Multi-programming with rapid context-switching
- Jobs cannot run for “too long”
- Allows for interactivity
  - Response time very short
  - Each job has the illusion that it is alone on the system
- In modern OSes, jobs are called processes
  - A process is a running program
- There are many processes, some of which are (partly) in memory concurrently (depends on the size of your memory and size/number of active processes)

Starting the OS

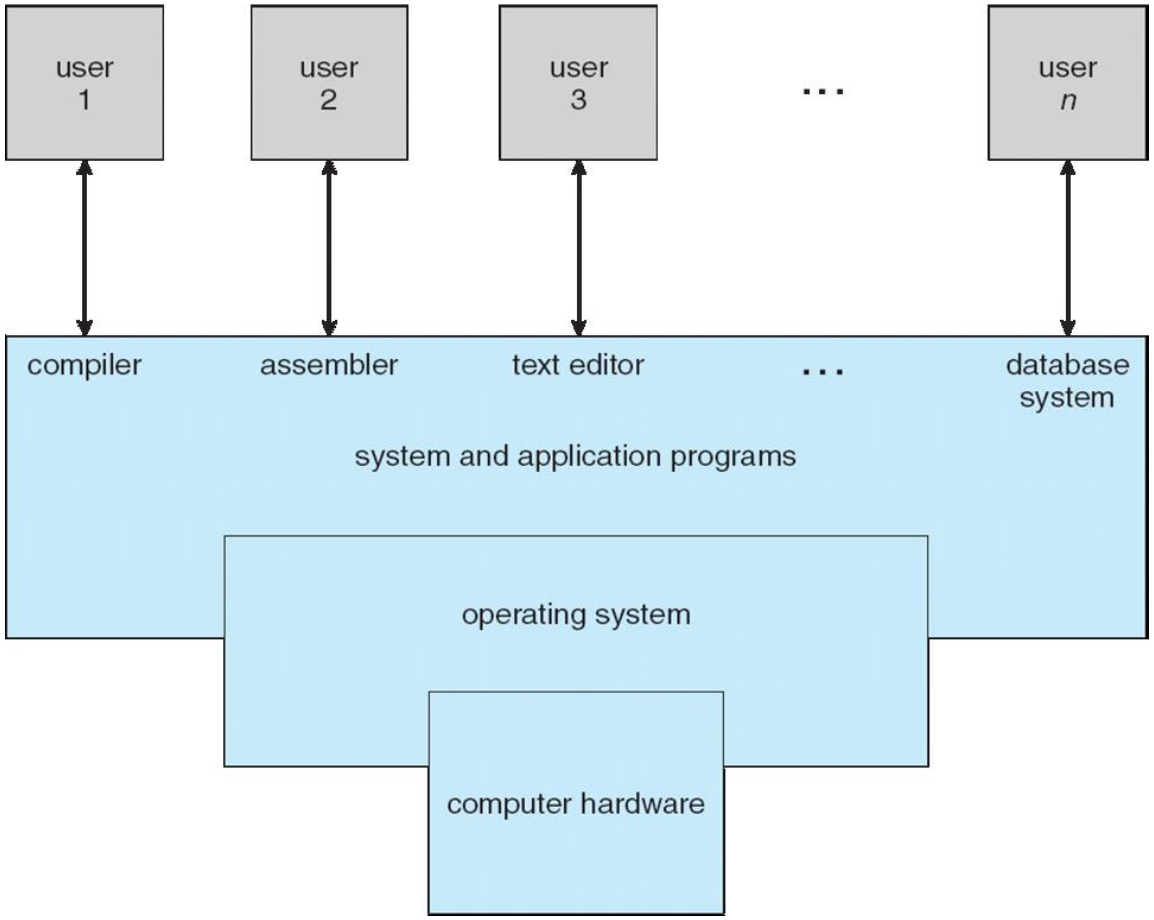
- When the computer boots, it needs to run a first program: the bootstrap program
  - Stored in Read Only Memory (ROM) or most machines today use Flash as it is able to be reprogrammed when updates occur, though not usually from the os
  - This software is called “firmware”
- The bootstrap program initializes the computer
  - Registers content, device controller contents, etc.
- It then locates and loads the OS kernel into memory
- The kernel starts the first process (called “init” on Linux, “launched” on Mac OS X)
- And then, nothing happens until an event occurs
  - More on events later

The Running OS



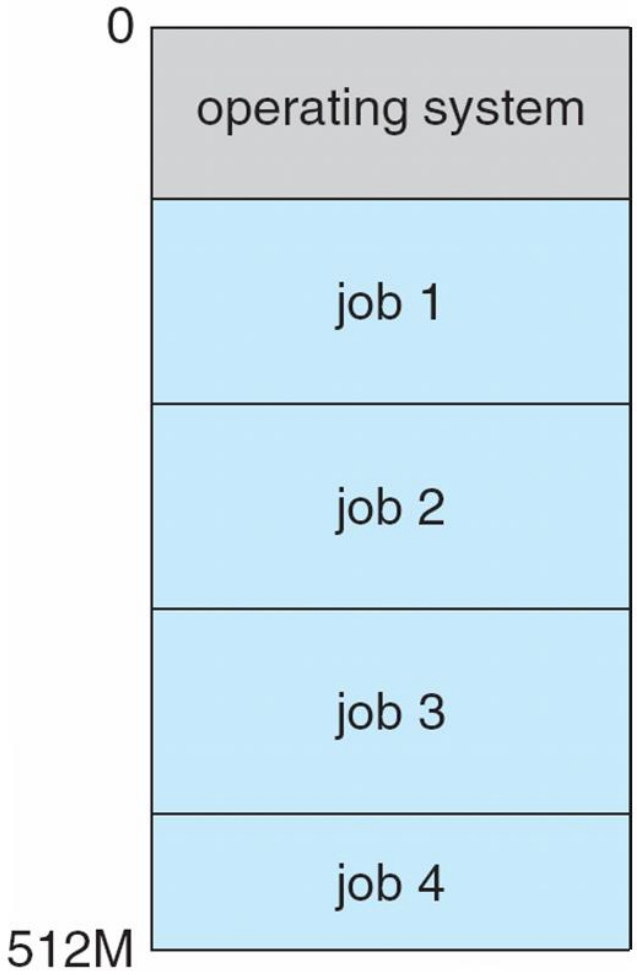
- The code of the operating system resides in memory at a specific address, as loaded by the bootstrap program
- At times, some of this code can be executed by a process
  - Branch to some OS code segment
  - Return to the program's code later
- Each process is loaded in a subset of the memory
  - Code + data
- Memory protection among processes is ensured by the OS
  - A process cannot interfere with another process

The Computing System Components



Running the OS Code?

- The kernel is **NOT** a running job
- It is code (i.e., a data and a "text" segment) that resides in memory and is ready to be executed at any moment
  - When an event occurs
- It can be executed on behalf of a job whenever requested
- It can do special/dangerous things
  - Meaning hardware related commands



Note about Kernel Size

- You saw in the previous figures that the kernel uses some space in memory
- As a kernel designer, you do not want to use too much memory
  - Leads to fights about whether new features are truly necessary in the kernel
  - Leads to the need to write lean/efficient code
- There is no memory protection within the kernel
  - The kernel is the one in charge of saying to process "segmentation faults"
  - Nobody is watching over the kernel
- You must be very carful when developing kernels

Dual-Mode Control

- **Mode bit:** Provided by hardware for dual-mode operation
  - **User mode:** Executing privileged instructions will automatically trap to the operating system
  - **Kernel mode:** Full access to the system is enabled
- **Processing**
  - **At boot time:** System is automatically in kernel mode
  - **During System call:** Mode automatically changes to kernel mode
  - **After responding to system call:** Restores application environment, set mode to user, and reschedules the application
  - **Illegal operation:** A trap (exception) occurs. Control immediately transfers to the operating system, which creates a data dump and terminates the executing program
- **Note:** Early PC's had no mode bit; applications could (and did) often crash the system

User vs. Kernel Mode

- All modern processors support (at least) two modes of execution:
  - **User mode:** in this mode protected instructions cannot be executed
  - **Kernel mode:** in this mode all instructions can be executed
- The CPU checks the status bit in a protected control register before executing a protected instruction
- Setting the mode bit is, obviously, a protected instruction

Protected Instructions

- There are a subset of instructions of every CPU that are restricted in usage: meaning only the OS can execute them
  - These are protected or privileged instructions
- Examples of these instructions include
  - Directly access I/O devices (printer, disk, etc.)
  - Manipulate memory management states
  - Execute the halt instruction that shuts down the processor

OS Events

- An event is an “unusual” change in the control flow
  - A usual change is some “branch” instruction within a user program, for example
- An event stops execution, changes mode, and changes context
  - Meaning it starts running in kernel mode
- The kernel defines a handler for each event type
  - A piece of code executed in the kernel mode
- Once the system is booted, all entries to the kernel occur as the result of an event
  - Essentially, the OS is one big event handler

OS Events

- There are two types of events: interrupts and traps (exceptions)
  - The two terms are often confused (even in the text)
  - The term fault often refers to an unexpected event
- Interrupts are cause by external events
  - Hardware-generated
  - Some device controller says “something happened”
- Traps are caused by executing instructions
  - Software-generated interrupts
  - The CPU tries to execute a privileged instruction but is not in kernel mode
  - A division by zero is also an example

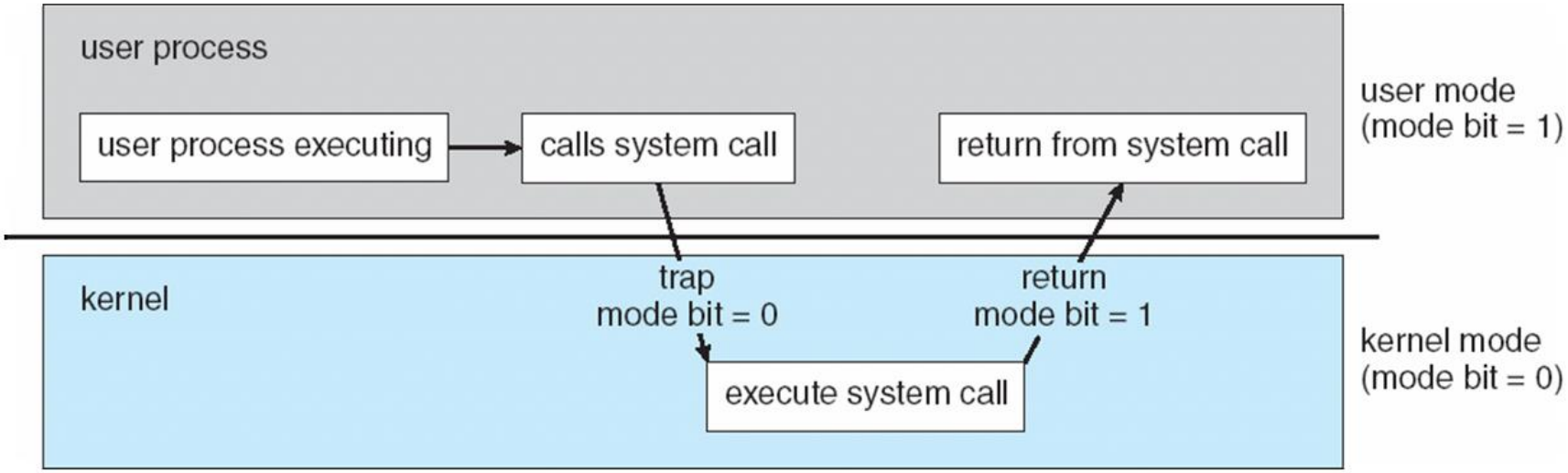
OS Events

- When the CPU is interrupted, it immediately stops what it was doing and transfers execution to a fixed location in the kernel code (the event handler code)
- This could result in:
  - The kernel doing some work
  - A user process being terminated (segmentation fault)
  - Notifying a user process of the event
- What happens when the kernel “faults”?
  - Perhaps a dereferencing of a NULL pointer, or divide by zero
  - This is a fatal fault
  - UNIX Panic, Windows blue screen of death
    - Kernel is halted, state is dumped to a core file, and machine is locked up

System Calls

- When a user program needs to do something privileged, it places a system call
  - For instance, to create a process, write to disk, read from the network card, etc.
- Every Instruction Set Architecture (ISA) provides a system call instruction which
  - Causes a trap, which “vectors” to a kernel handler
  - Passes a parameter determining which system call to place
  - Saves the caller state (PC, registers, etc.) so it can be restored later

System Call Processing



Note: The OS must save the application environment, process the request, and then restore the environment when the application is rescheduled.

System Timer

- The OS needs to keep control of the CPU
  - Programs cannot get stuck in infinite loops and lock the computer
  - Programs cannot can not be allowed to take an unfair share of the computer
- One way in which the OS retrieves control is when an interrupt occurs
- To ensure that an interrupt will occur reasonably soon, the system employs a timer
- The timer interrupts the computer regularly
  - The OS ensures the timer is set before turning control over to a user process
- Modifying the timer is a privileged instruction

OS Services

- The OS provides some main services which include
  - Process Management
  - Memory Management
  - Storage Management
  - I/O Management
  - Protection and Security

Process Management

- A process is a program that is executing
  - Programs are passive (blocks of executable instructions/data)
  - Processes are active (A program together with its active state)
- The OS is responsible for:
  - Giving process the resources they need
  - Providing an API so processes can “talk” to the OS
  - Protecting process from one another
  - Allowing processes to communicate with one another
  - Implement fair share scheduling algrithms
  - Terminate and reclaim reusable resources
  - Create and manage execution of threads
- A typical system as many processes running. These include applications, system processes, and background processes

Scheduling

- Short Term (Must be extremely efficient)
  - Look at queue of process ready to run
  - Decide which has the highest priority
  - Restore its runtime environment
  - Set the system to user mode
  - Transfer control to the process
- Long Term (Low priority background processes)
  - Decide if system load is at an acceptable level
  - Launch processes waiting to enter the system

Process Control Block

- This is the object that maintains the state of a process
  - A program counter and registers for all threads in a process
  - Pointers to the resources allocated to a process
    - File and device handles
    - Mutexes
    - Connections to remote resources
  - Physical and Logical memory allocated to this process (which could be contiguous, or not, in memory, or on disk)
  - Initialization data
  - Runtime statistics and accounting information
  - Links to shared memory and connections to other processes

Memory Management

- Memory management determines what is in memory and when
  - The kernel is ALWAYS in memory
- The OS is responsible for:
  - Tracking which parts of memory are currently in use
  - Transferring processes (or parts thereof) and data into and out of memory
  - Allocating and releasing memory space as needed
  - Maintaining sections of memory for caching disk operations
  - Control shared memory accessible to multiple processes
  - Map logical program addresses into physical memory and maintain portions that are on disk and not contiguous
  - Prevent processes from overwriting each other's memory
- The OS is not responsible for memory caching, cache coherency, etc.
  - These are managed by the hardware

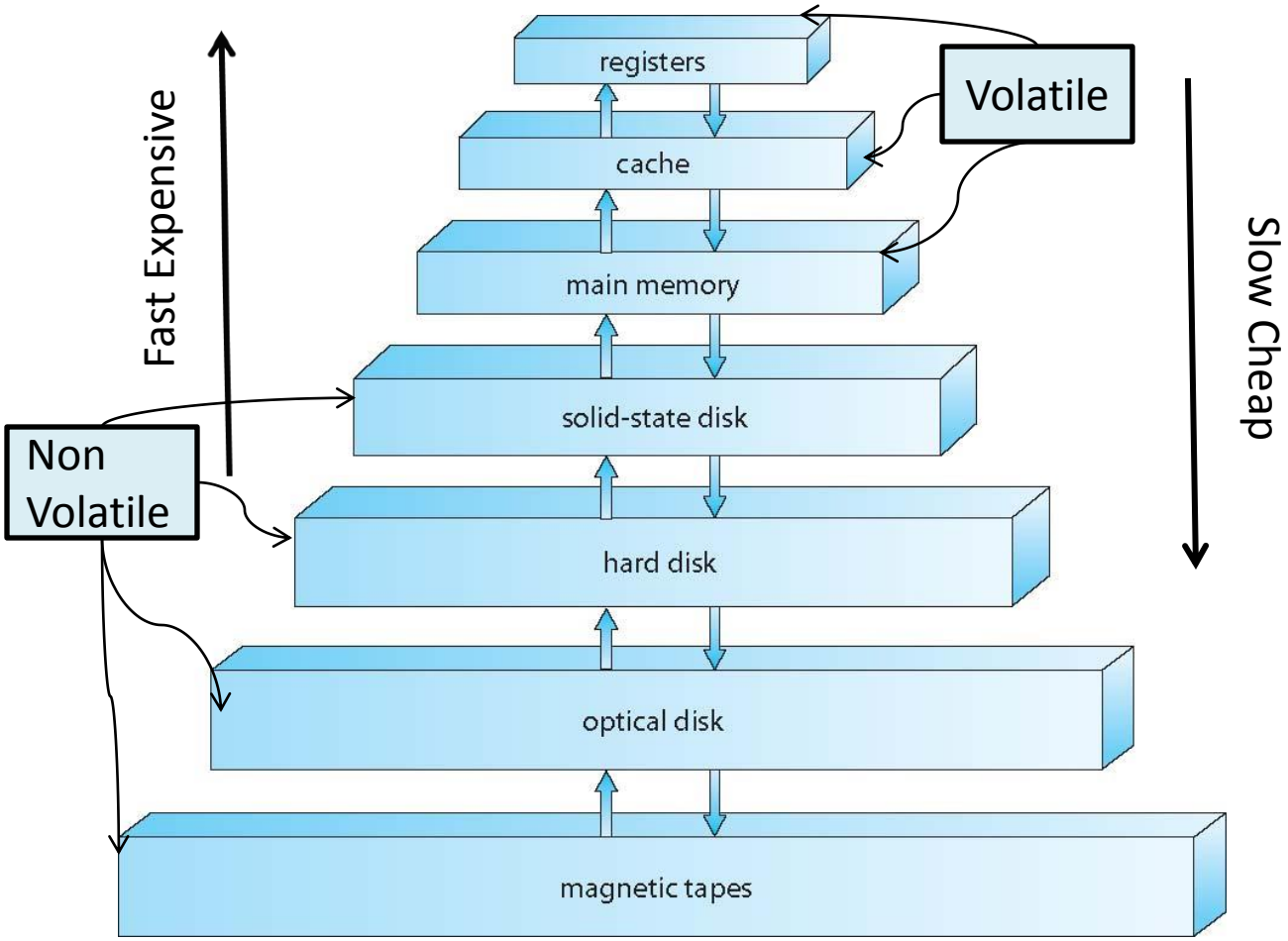
Storage Management

- The OS provides a uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit
- OS operates File-System management
  - Creates and deletes files and directories
  - Mounts and dismounts devices
  - Maintains secondary storage by mapping files onto available areas of the system medium
  - Backup files onto stable (non-volatile) storage medium
  - Establish transparent connections to remote data
  - Establish access control mechanisms
  - Schedule writes and reads for maximum efficiency

Storage Hierarchy

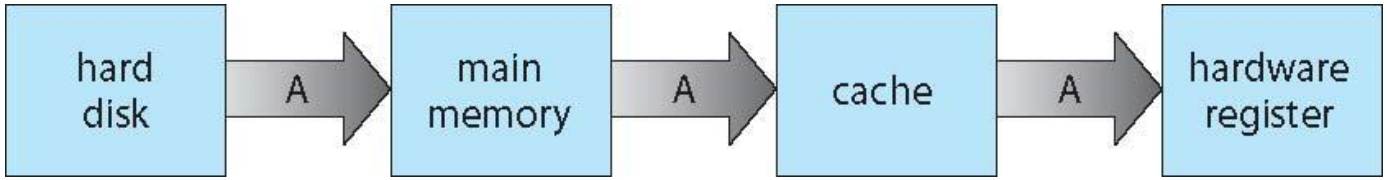
- Storage systems are organized in a hierarchy
  - Higher levels are faster, smaller, and are more likely to be volatile
  - Lower levels are slower, larger , and are less likely to be volatile
- *Cache:* Temporarily copy data into faster storage
  - **Locality principle:** Data accessed as clusters in local areas
  - **Load policy:** Retrieve the data needed plus surrounding data
  - **Replacement policy:** Choice of cache items to expel when the cache fills
  - **Processing:**
    - The system looks to a faster cache before accessing the slower hierarchy levels
    - Higher levels in the storage hierarchy can serve as caches for the lower levels
    - On updates, lower levels are either immediately updated (*write through*) or updated only when an item is expelled (*delayed write*) from the cache
    - Cache coherency (consistency of multiple caches) is an important consideration in parallel systems which contain multiple caches
    - Caching can be explicitly OS controlled or implicitly done in hardware (depending upon which level we are discussing)

Secondary Storage Hierarchy



Cache Coherence

- We must always use the most recent value, no matter where it is in the storage hierarchy



- A Multiprocessor environment must provide cache coherency. All CPUs must have the most recent value in their local cache
- Distributed environments are even more complex

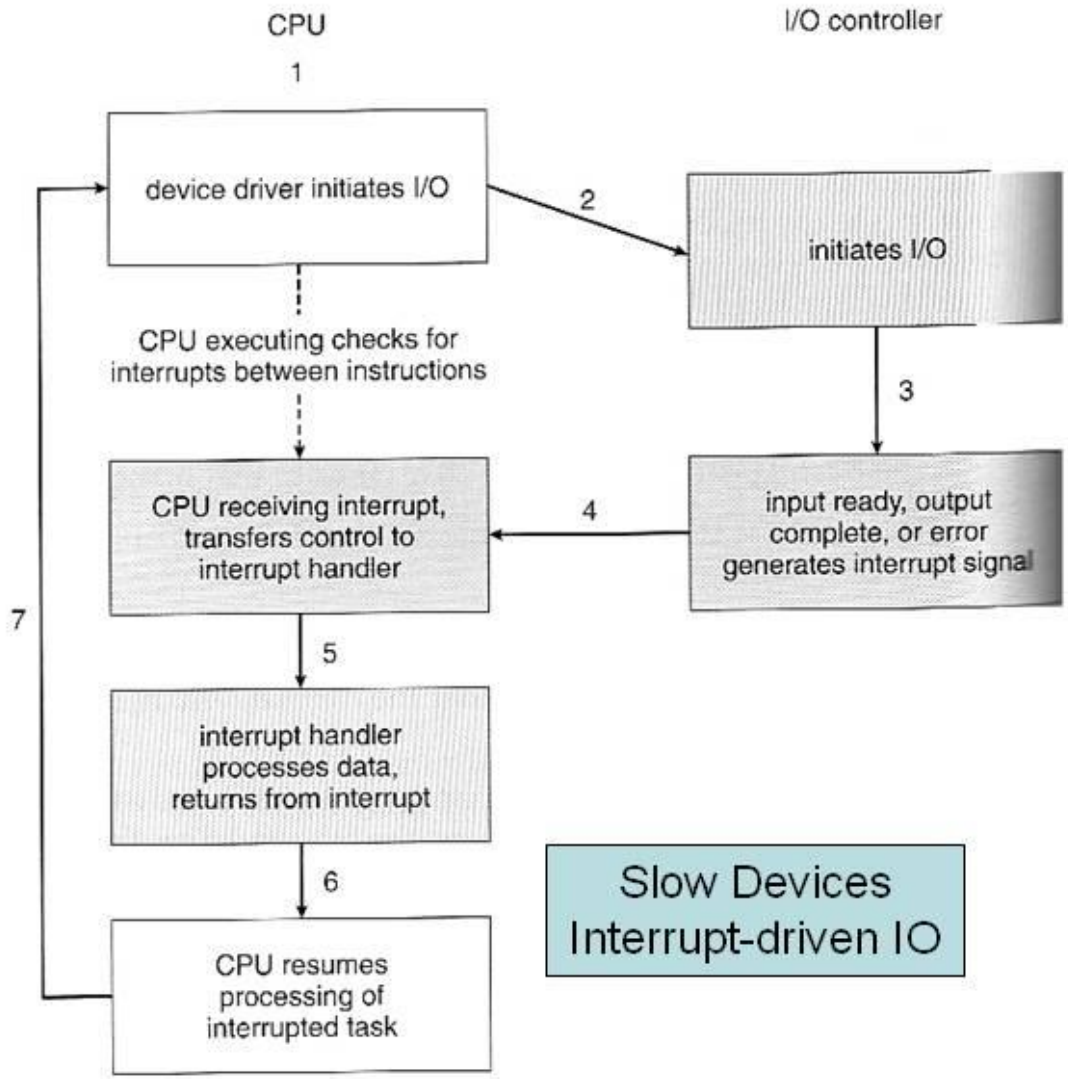
I/O Management

- The OS hides peculiarities of hardware devices from the user
- The OS is responsible for
  - Buffering (storing data temporarily while it is being transferred)
  - Caching (storing parts of data in faster storage fro performance, but only as far as main memory, the OS does not control CPU cache)
  - Spooling (the overlapping of output of one job with the input of other jobs)
  - Uniform device-driver API to abstract device specifics
  - Managing drivers(software modules) for specific hardware devices
- Device Controllers
  - Operate simultaneously and autonomously with CPU
  - Are limited instruction set processors to solely manage devices
  - Are responsible for a particular type of device
  - Contain local memory buffers and hardware registers to control I/O

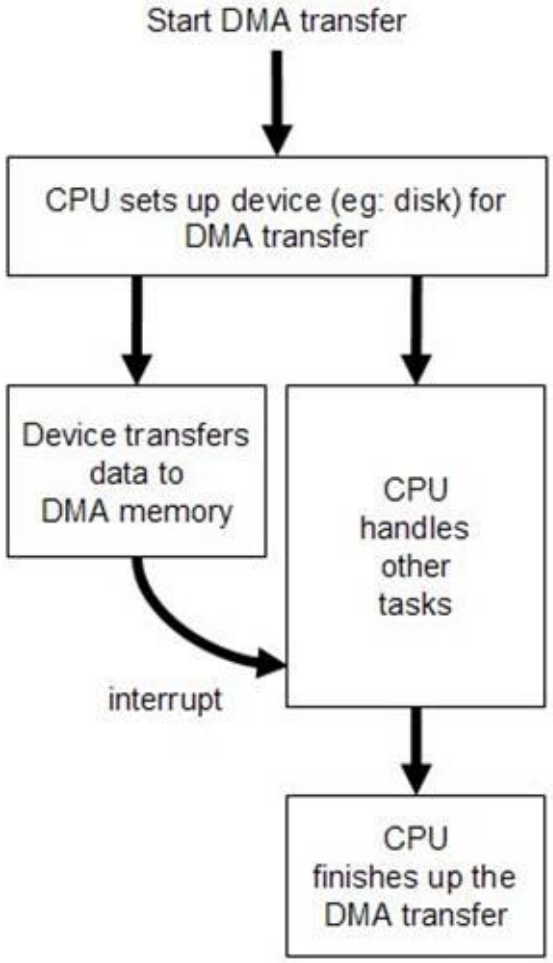
Approaches for Handling Devices

1. **Device driver API:** a uniform mechanism to abstract hardware operation from the rest of the system
2. **Programmed I/O:** computer starts an operation and then waits in a tight loop for the operation to complete
3. **Polling:** A loop that inquires and responds to each device
4. **Interrupts** (An I/O signal raised by a device)
   a) Computer starts an operation
   b) Computer does other things
   c) Computer transfers data, piece by piece, when interrupts occur
5. **Direct Memory Access** (DMA)
   a) Computer starts an operation
   b) Computer does other things
   c) Device transfers a block using main memory by cycle stealing
   d) Computer responds to interrupt when device I/O is done

Device Controllers



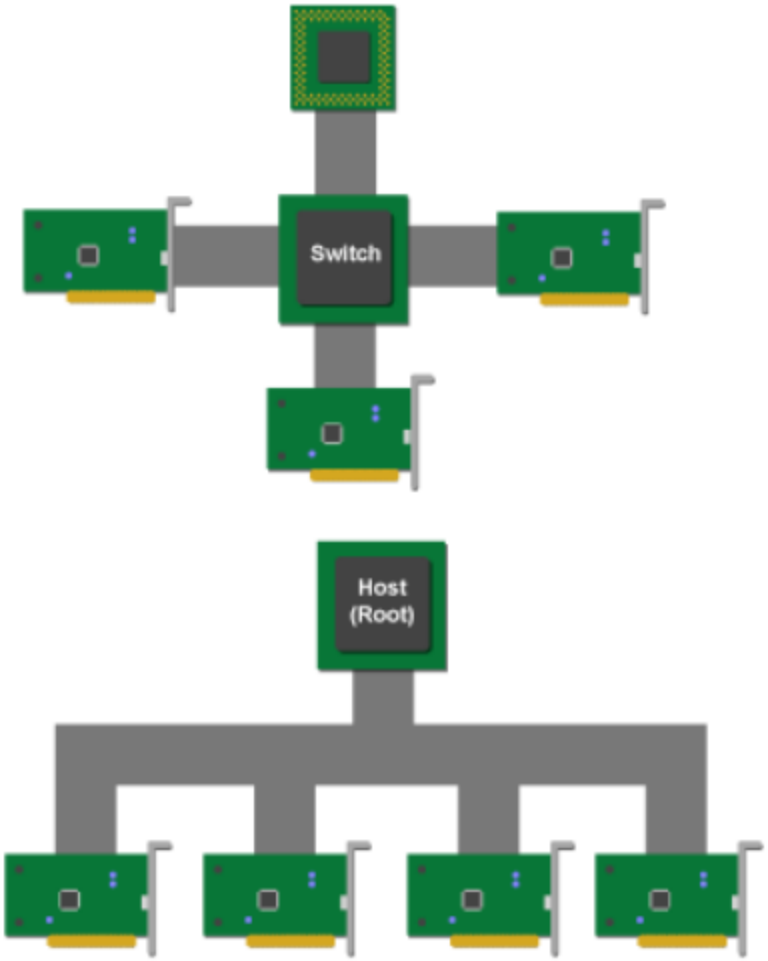
Slow Devices
Interrupt-driven IO



Fast Devices
Direct Memory Access,
possible cycle stealing

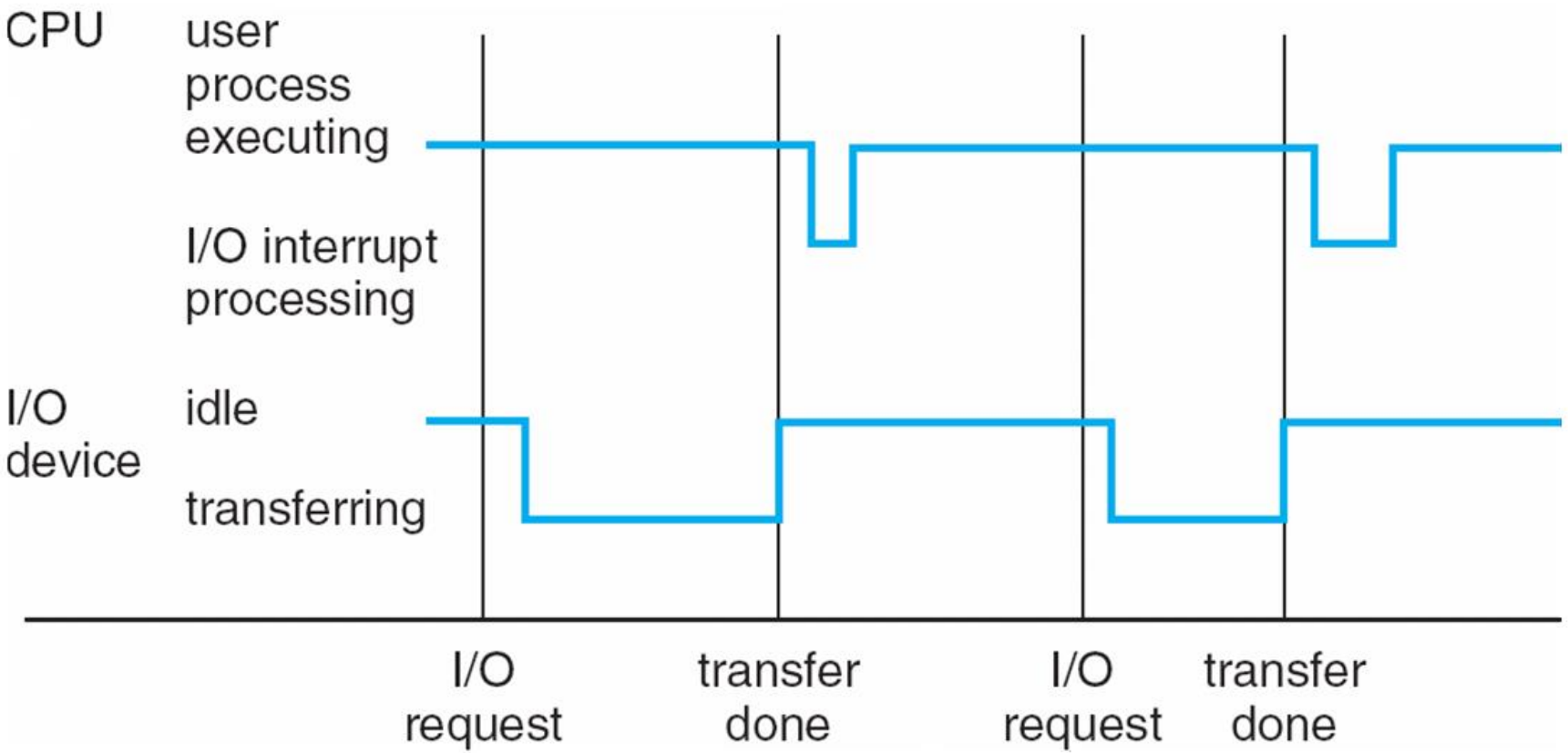
Bus vs. Switched Architecture

Using switched architecture (as in PCIe) helps avoid some cycle stealing as compared to a simple buss architecture

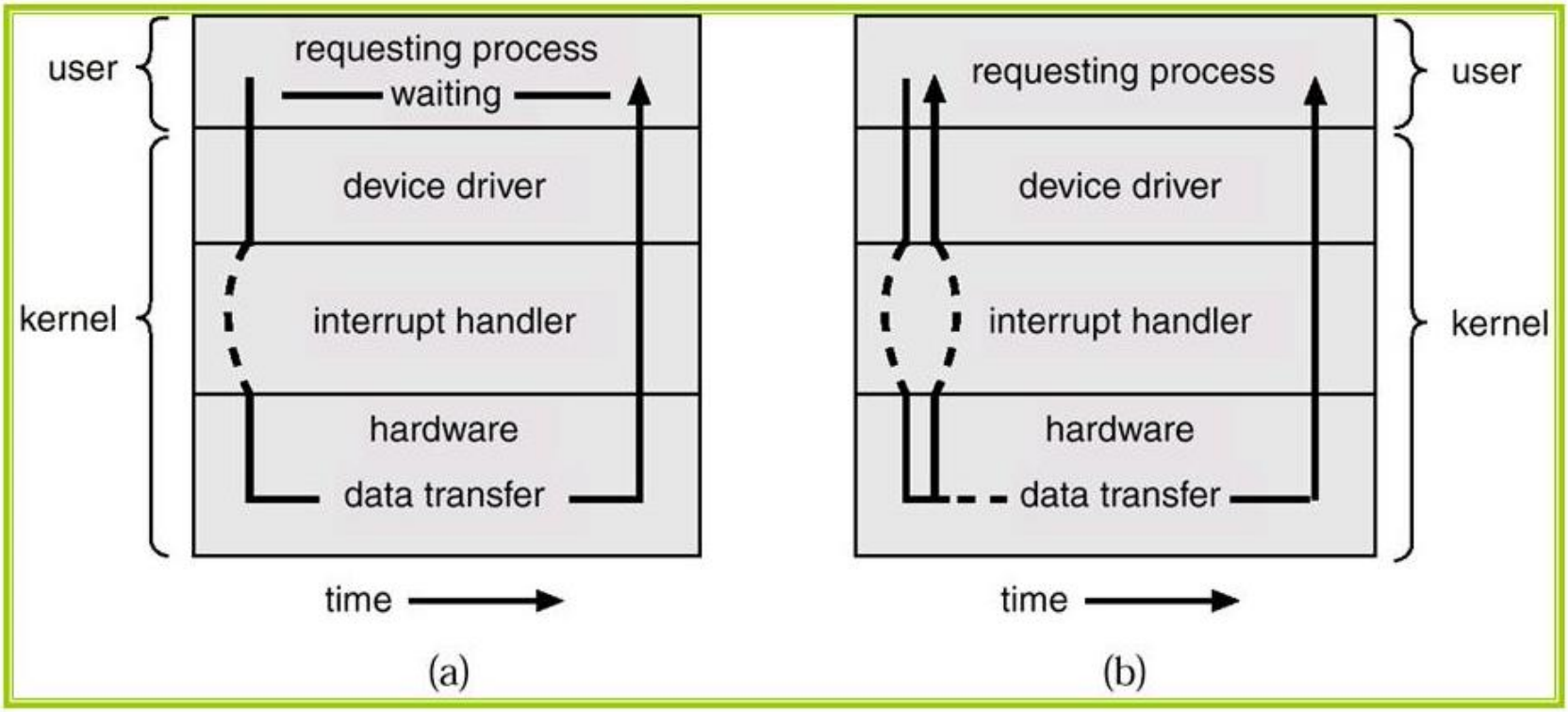


Intro to PCI Express:
http://www.testbench.in/introduction_to_pci_express.html

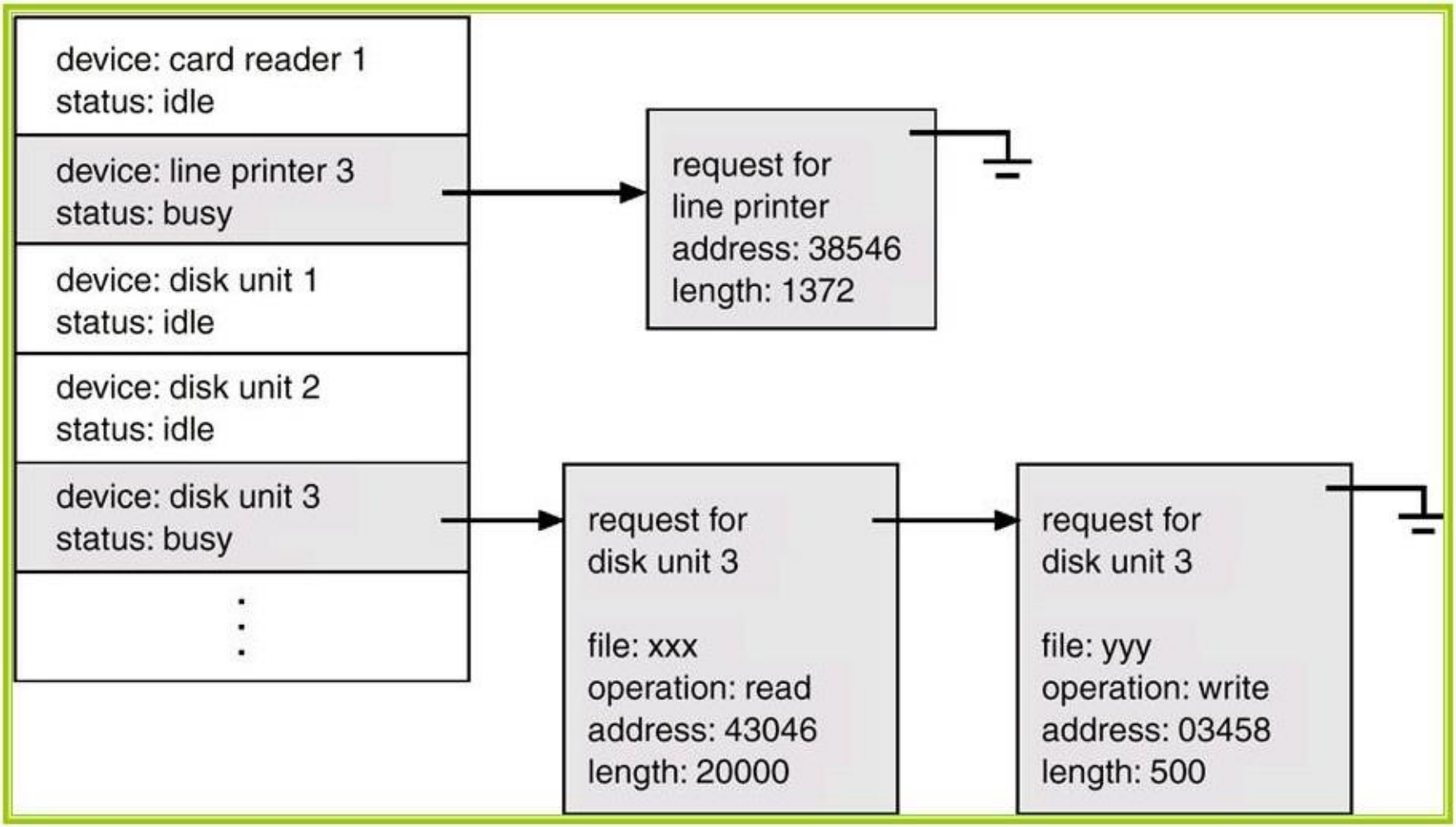
Interrupt Timeline



Synchronous & Asynchronous I/O



Device-Status Table



Protection and Security

- **Protection:** mechanisms for controlling access of process to resources defined by the OS
  - Specifies and implements the controls available
  - Distinguish between authorized and unauthorized use
  - Examples: infinite loop, malicious processes, disk corruption
- **Security:** defense of the system against internal and external attacks
  - Operating system middleware level approaches
  - Examples: denial-of-service, worms, viruses, identity theft
- **Techniques to enforce protections and security**
  - Distinguish between groups of users: user ID and group ID
  - Privilege escalation: process request additional rights (setuid call)
  - Access Control: rwx rights (chmod)

Privileged Instructions

- In class discussion: which of these instructions should be privileged, why?
  - Set value of the system timer
  - Read the clock
  - Clear memory
  - Issue a system call instruction
  - Turn off interrupts
  - Modify entries in device-status table
  - Switch from user to kernel mode
  - Access I/O device

Privileged Instructions

- In class discussion: which of these instructions should be privileged, why?
  - Set value of the system timer
  - Read the clock
  - Clear memory
  - Issue a system call instruction
  - Turn off interrupts
  - Modify entries in device-status table
  - Switch from user to kernel mode
  - Access I/O device

Sections 1.11 and 1.12

- The text has sections on “Computing environments” and “Open-Source Operating Systems”
  - Read them!
  - They talk about a number of topics that are part of the general culture that you should have, if you don’t already

Conclusion

- Reading assignment: Chapter 1-2
  - You should have really already read Chapter 1
  - Start Chapter 2 we will
  - It's always a good idea to read and try to quickly do the "Practice Exercises" at the end of each chapter, it helps reinforce what you learned and expose what you may have missed