Sources for CSE 410 Course Notes

- Most of the materials in these course notes were developed by Prof. Philip McKinley at Michigan State University

- Other sources of materials include:
- Operating Systems, Internals and Design Principles, 8th Edition, Stallings
- The Linux Kernel, Rusling (online), D. Rusling
- Modern Operating Systems, 3rd Edition (2008), A. S. Tanenbaum.
- Mac OS X Internals, A Systems Approach (2007), A. Singh.
- Linux Kernel Internals, 2nd Edition (1998), M. Beck et al.
- Inside Windows NT, 2nd edition, Solomon, Microsoft Press, 1998.
- Operating Systems: Three Easy Pieces, Arpaci-Dusseau[2], (free text)
- Course notes of Prof. K. R. Joshi at Columbia University

- Thanks to all these great writers and instructors!

1-2: Review of Computer Architecture

- Stallings, Chapter 1
- The operating system and the architecture work hand-in-hand to support user applications.
- Some parts of the OS interact directly with HW, e.g.,
  - Boot procedures, switching among processes
  - Configuring memory manager
  - Interacting with I/O devices
- Other parts are heavily influenced by the arch, e.g,
  - Implementation of threads, scheduling, synchronization
  - Memory and file-related data structures/routines
  - All sorts of device- and interface-related software

(Very) Basic Computer Organization

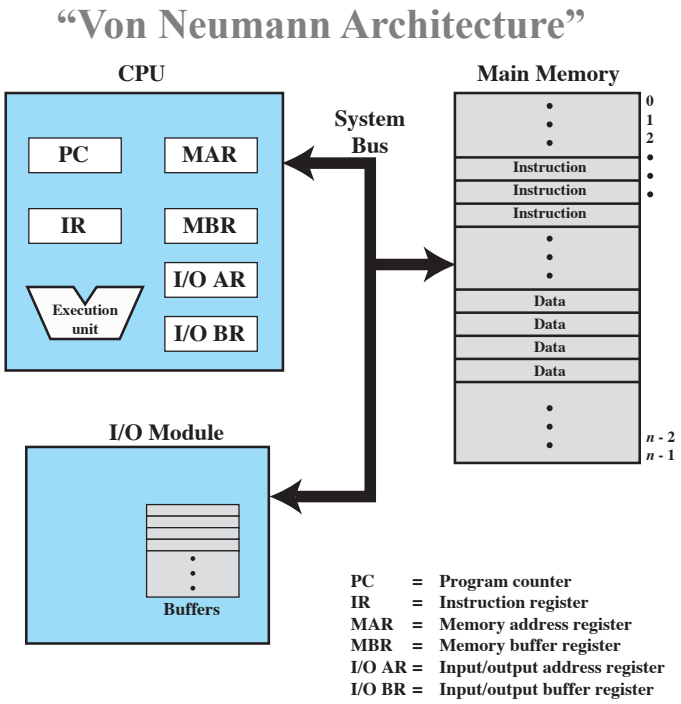
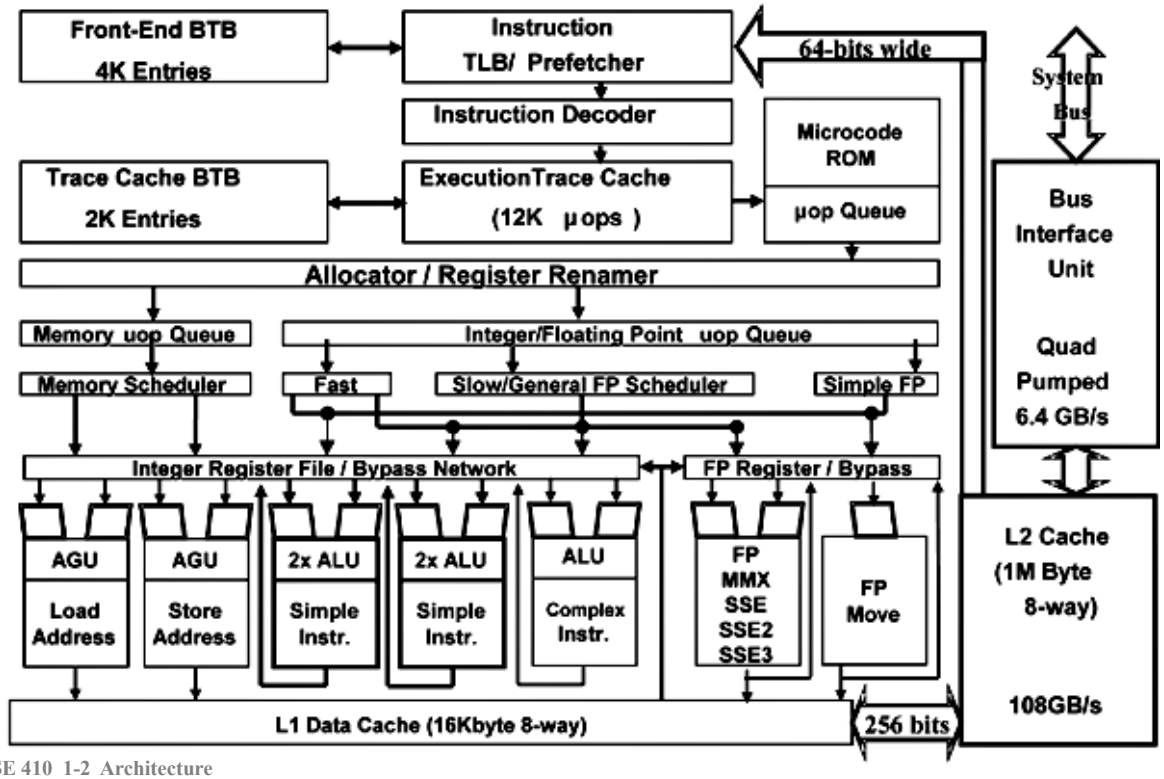
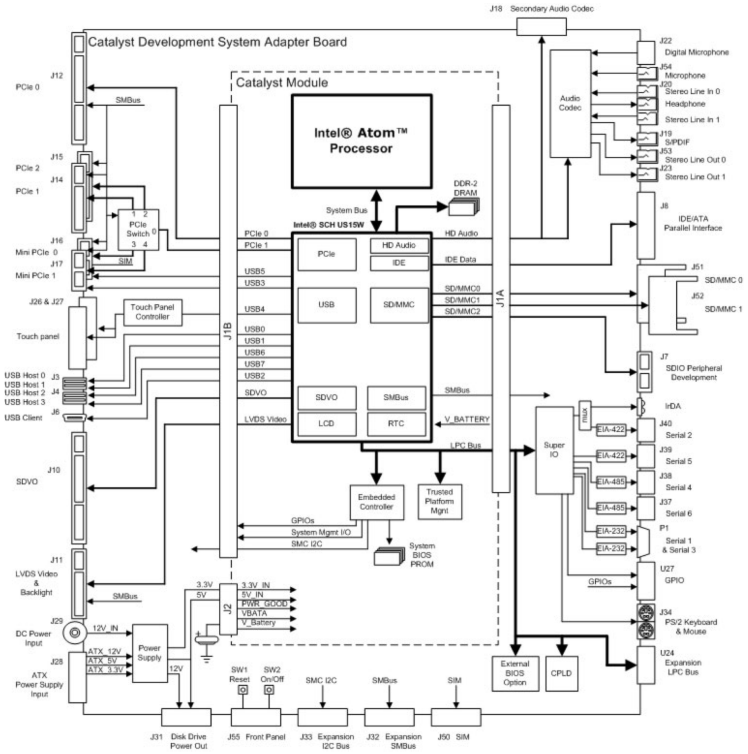


Figure 1.1 Computer Components: Top-Level View

Pentium 4 (2000-2008)



Intel Atom (2010, for embedded systems)



Basic Components of a Computer

- Processor (CPU)
- Main Memory (as opposed to secondary storage)
  - holds program code and data
- I/O modules
  - hardware (controllers, buffers, and registers called I/O ports) that moves data between CPU and peripherals such as:
    - secondary memory devices (e.g., hard disks), printers
    - keyboard, display, network cards
- System interconnection (i.e., buses)
  - provides communication among processors, memory, and I/O modules

Major OS-Related Architectural Issues

- Instruction Execution and CPU Registers
  - Execution mode, 32- vs. 64-bit, protected registers
- Interrupts
  - Hardware interrupts from peripherals
  - Clock, power-off, inter-processor interrupts
  - Traps and other “software interrupts” (e.g., divide-by-zero)
- Memory hierarchy
  - Minimizing data access time, all levels
  - Affects design of OS (buffer cache, scheduling)
  - OS has to work with architecture to accomplish this
  - OS has extra tasks as a result (e.g., flushing cache)

Major OS-Related Architectural Issues

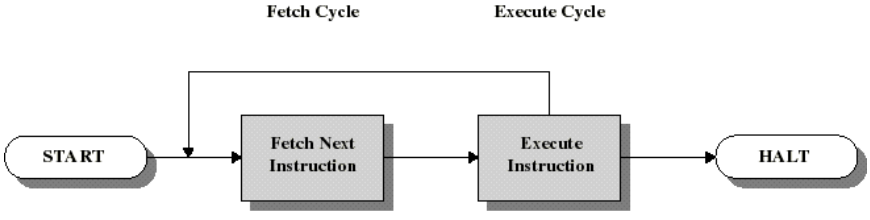
- Memory management
  - Demand-paged **virtual** memory
  - Address translation, address space protection, paging and page replacement
- Communication with peripheral devices
  - Programmed I/O vs direct memory access
  - Block vs. character devices
- Multiprocessors and multiple cores
  - Multiple kernel instances executing **in parallel**!
  - How to schedule processes, protect shared resources.

Instruction Set

- Also referred to as Instruction Set Architecture (ISA)
- Set of instructions executed by the particular processor
- Defines opcodes, operand formats, addressing modes, use of registers, etc.
- Examples:
  - x86 – usually refers to 32-bit Intel architecture (earlier, 16)
  - x86_64 (aka AMD64) – 64 bit version of x86, backward compatible with 32 (and 16) bit instructions

The Basic Instruction Cycle

- The CPU fetches the next instruction (and possibly operands) from memory.
- Then the CPU executes the instruction
- Program counter (PC) holds address of the instruction to be fetched next
- Program counter is automatically incremented after each fetch



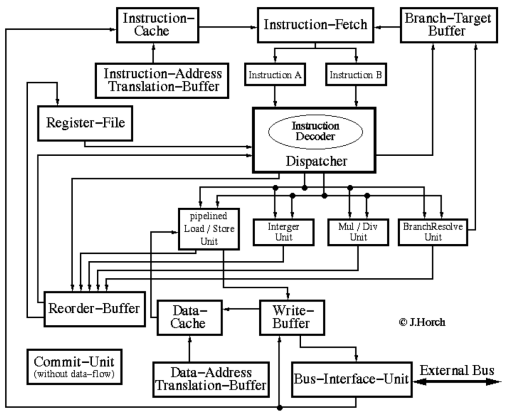
Basic Instruction Cycle

Types of Instructions

- Load/Store: Transfer data from memory to processor or vise versa
- Data Processing: Perform arithmetic or logic operation on data
- Control: alter the execution sequence of the program. Examples?
- Processor-I/O: transfer data to/from a peripheral device
- “Atomic” instructions for synchronization, mutual exclusion (e.g., test-and-set, compare-and-swap)

Note: Modern CPUs are more complicated…

- Superscalar – multiple execution units (e.g. ALUs, load units) can work on instructions in parallel
- Instruction pipelining – overlap fetch, decode, etc.
- Data pipelining – operations on vectors
- Speculative execution – processor works ahead on calculations that might not be needed
  (e.g., branch prediction, prefetching data, etc.)



CPU Registers (fast storage on CPU)

- Play multiple roles in instruction execution
- 30-100 times faster than accessing main memory
- User-Visible Registers
  - available to both OS and user programs
  - hold operands, results, addresses, and condition codes
  - access time is very fast relative to that of main memory
- Control & Status Registers
  - generally not available to user programs
  - some used by CPU to control its own operation
  - others used by OS to control program execution

User-Visible Registers

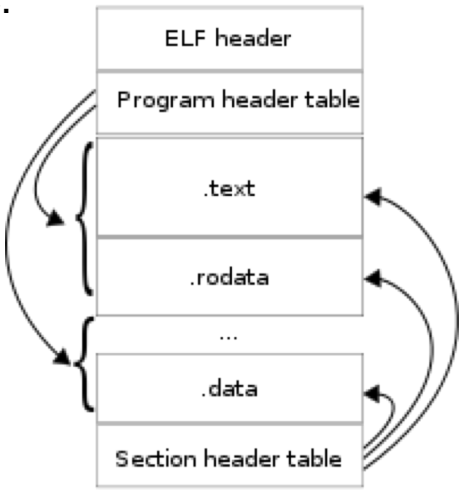
- Data Registers
  - can be assigned by the compiler to hold variables
  - CPU operates on them, stores results in them
  - advantage of placing data in registers?
- Address Registers
  - contain memory address of data and instructions
  - may contain a portion of an address that is used to calculate the complete address

Examples of Control & Status Registers

- Program Counter (PC)
  - Contains the address of the next instruction to be fetched
- Instruction Register (IR)
  - Contains the instruction most recently fetched
- Stack Pointer (SP)
  - Points to top of stack
- Program Status Word (PSW)
  - A register or group of registers containing:
    - condition codes and status info bits (zero flag, sign flag, etc.)
    - interrupt enable/disable bit
    - supervisor/user mode bit
- Can user program modify PC directly?

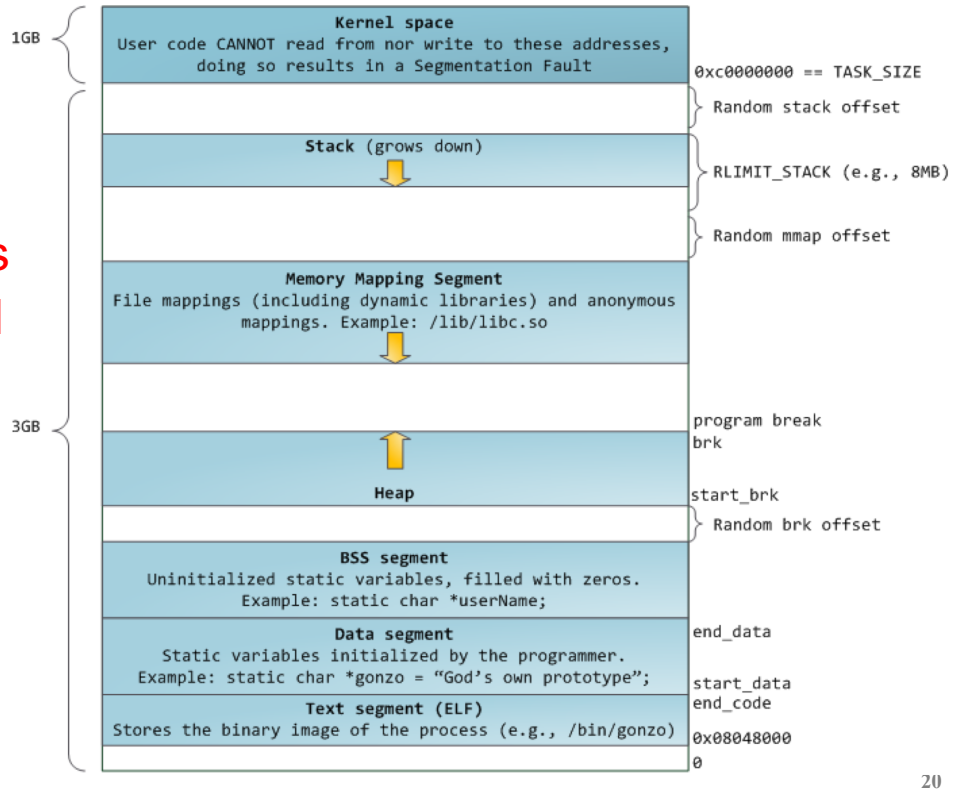
Loading and Linking Executables

- A program is compiled for a particular architecture
  - Object code comprises instructions for that architecture
  - Object file arranged in a particular format, defining location and size of text, data, etc.
    - Examples: ELF, Mach-O, PE
- Operating system loader
  - Invoked when spawning process
  - Reads executable file, lays out memory for process
  - Allocates system data structures
  - Creates thread of control for process



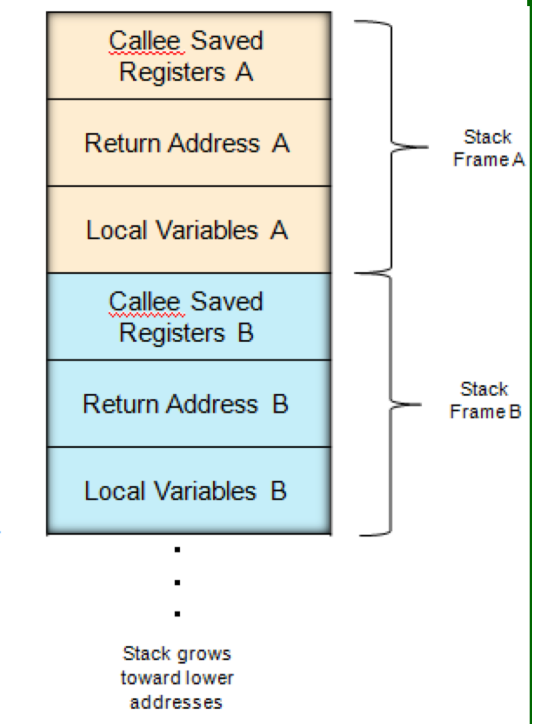
Process Memory (Address Space)

- Example: 32-bit Linux
- Nowadays ASLR and KASLR



Reminder: Stack-based Execution

- When a program calls a function, a new activation record is placed on the stack and the PC and SP are updated
- (Real stacks grow **down**)
- So, your program happily runs along, pushing and popping activation records
- Exception to this flow?

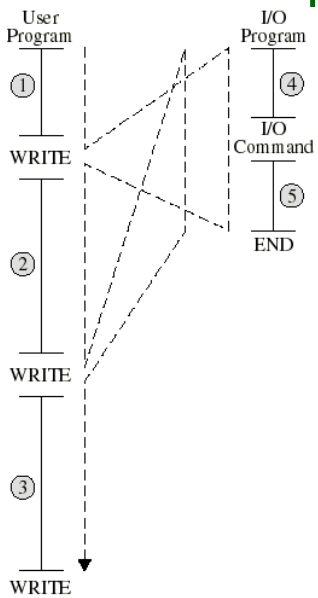


Interrupts

- **Fundamental to the operation of modern computers and an integral part of OS design**
- Example: enables I/O modules to interrupt execution of the CPU.
  - The I/O module asserts an interrupt request line on the control bus
  - Then the CPU transfers control to an Interrupt Handler Routine (normally part of the OS)
  - When the handler exits, execution resumes at interrupted point

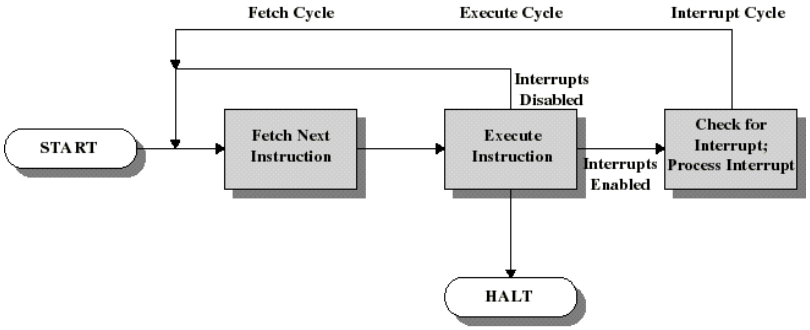
Example: I/O without Interrupts…

- WRITE system call invokes device driver (called I/O program in your text)
- Driver prepares I/O module for task (e.g., printing or writing to hard drive) (4)
- CPU has to WAIT for I/O command to complete
- Long wait…
  - E.g., Processor at 10^9 instructions/sec
  - 7200 rpm disk, half revolution is 4.2 milliseconds
  - Disk is 4 million times slower than processor
- Driver finishes in (5) and report status of operation



Modified Instruction Cycle

- CPU checks for interrupts after each instruction
- If no interrupts, then fetch the next instruction for the current program
- If an interrupt is pending, then suspend execution of the current program, and execute the corresponding interrupt handler

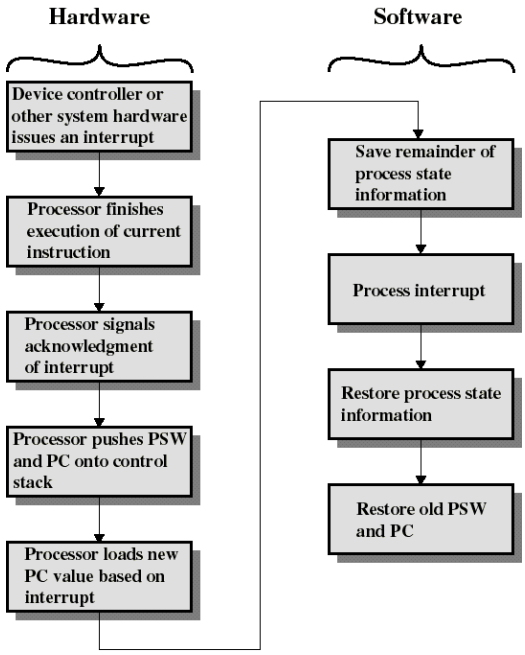


Interrupt Handler

- Definition: A piece of code that determines nature of the interrupt and performs whatever actions are needed
- Interrupt handler addresses (interrupt vector table) configured at boot time, addition of peripherals, etc.
- The point of interruption can occur anywhere in the program
- Thus, the state of the interrupted process (content of PC + PSW + registers + ...) must be saved so the process can be resumed later as if nothing had happened

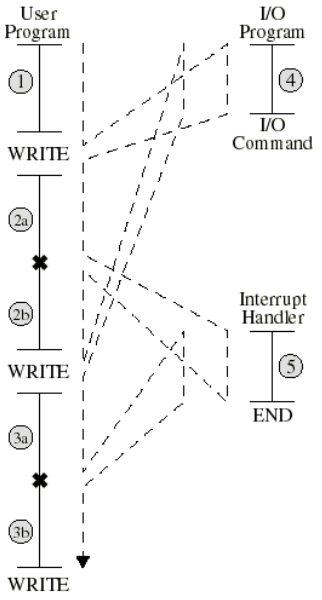
Simple Interrupt Processing

- Interrupt handling involves combination of hardware and software
- The interrupted program is unaware of the interrupt
- Can operating system be interrupted?
- Interrupt handling should be fast. Why?



Interrupts Improve CPU Usage

- Driver called, prepares the I/O module and issues the I/O command (eg: write to disk)
- System call returns
- User code* gets executed during I/O operation: no waiting
- User program interrupted (x) when I/O operation is done. Interrupt handler examines status of I/O module, exits.
- Execution of user code resumes
- might be a different process

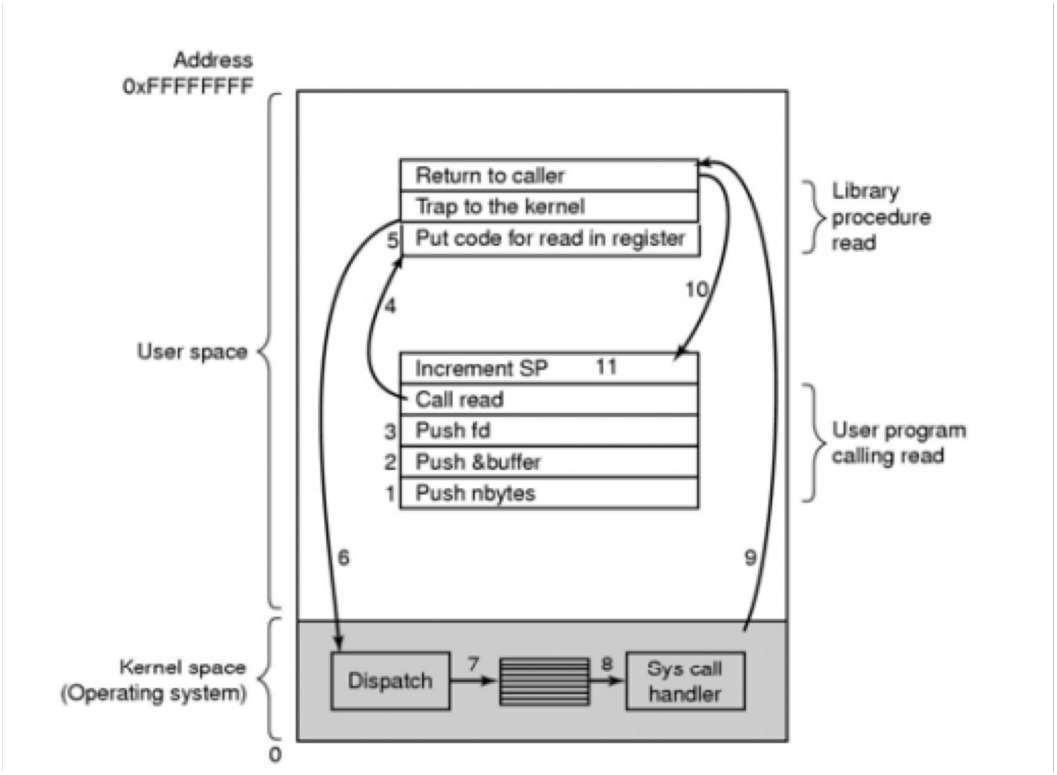


- **But I/O is only one reason for interrupts!**

Classes of Interrupts

- I/O
  - signals normal completion of operation or error
  - (disk read/write, packet transmission reception, etc.)
- Program Exception
  - overflows, divide-by-zero
  - try to execute illegal instruction
  - reference outside user's memory space
- Clock – how used by OS?
- Hardware failure (eg: memory parity error)
- Traps – how used?

System Call Steps

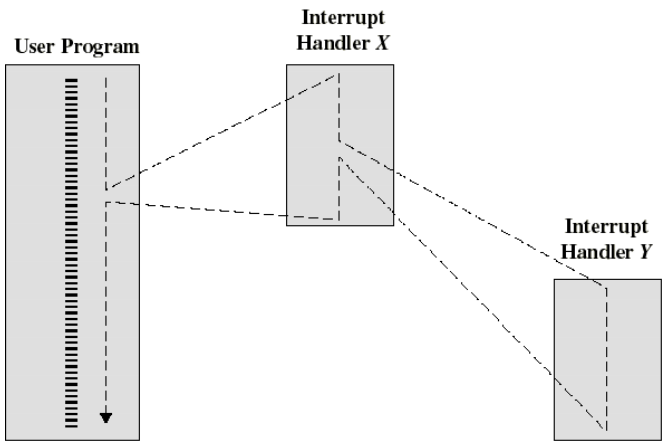


BTW, system calls and stacks…

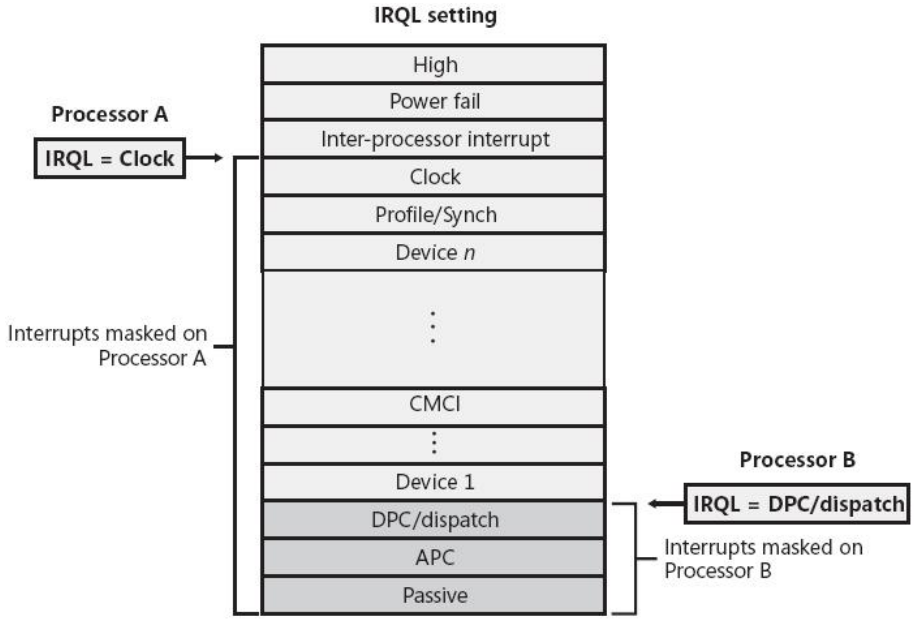
- When a process traps to the kernel, it *could* continue executing on its regular stack, but just with the processor in kernel mode.
- But instead the process executes on a different, kernel stack. In Linux, one per process (thread).
- Why? Security

Can an interrupt handler be interrupted?

- Yes. Interrupts have priorities. Higher priority interrupts can interrupt lower-priority handler.
- Also, new lower-priority interrupts have to wait.
- Example: power-failure interrupt takes priority over others



Interrupt ReQuest Levels (IRQL)



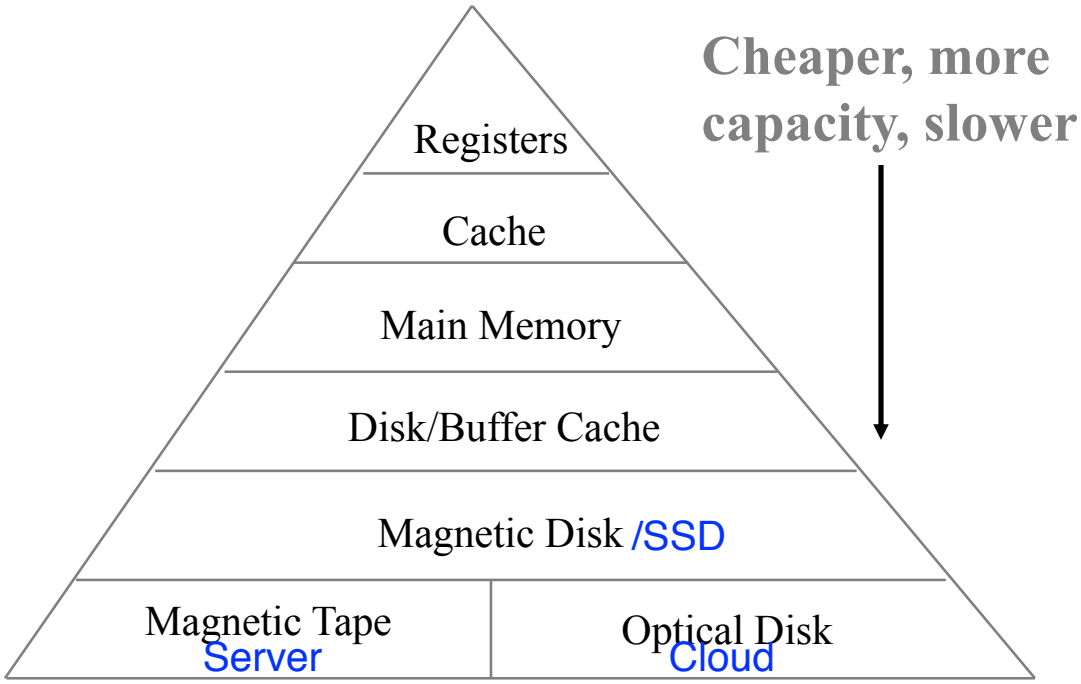
Fundamental Concepts from 1st Lecture…

- Limited Direct Execution
  - when your process is running, it runs directly on hardware
  - OS needs way to gain back control
- Virtual Memory
  - your process does not see physical memory
  - OS and HW implement address space for each process
- OS code (usually) runs in context of a user process
  - processor switches to kernel mode on system call trap
  - kernel code and data are in the address space of the process, just not accessible when executing at user level

A broader fundamental concept…

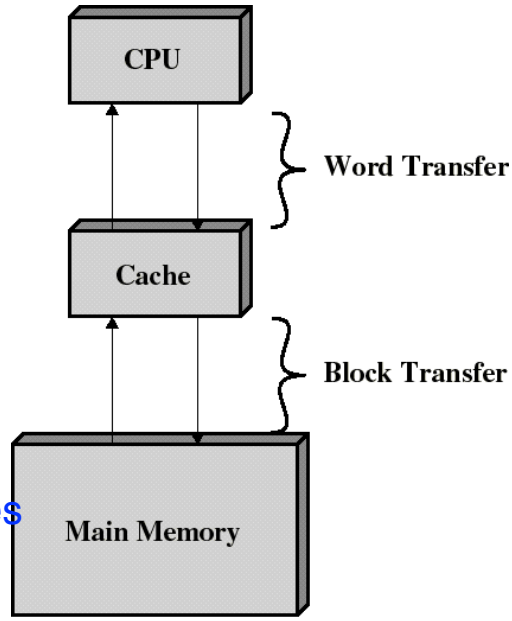
- The *entire* execution of a computer comprises:
  - code sequences
  - interrupts that "jump" to another code sequence
  - returns from interrupt that jump back to a prior sequence
  - one possible side effect: switching the processor mode
- Sometimes referred to as interrupt-driven execution

Memory Hierarchy

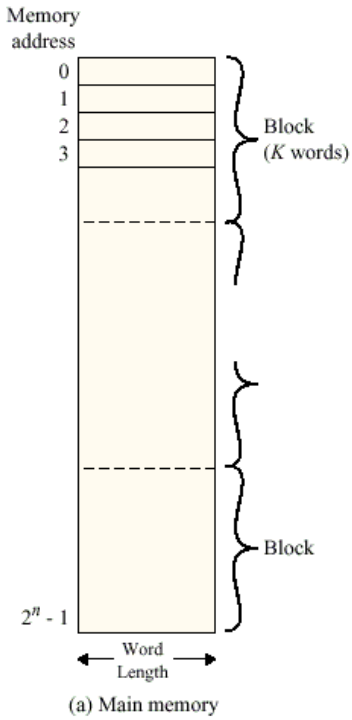


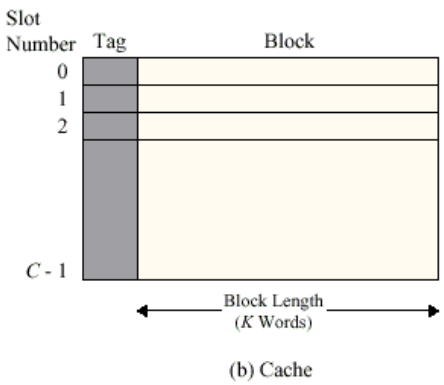
(Hardware) Cache Memory

- Small cache of expensive but very fast memory interacting with slower but much larger memory
- Invisible to OS and user programs but interacts with other memory management hardware
- Processor first checks if word referenced to is in cache
- If not found in cache, a block of memory containing the word is moved to the cache
- Typical block size? 64 Bytes 128 Bytes
- Typical cache size? 8k Bytes

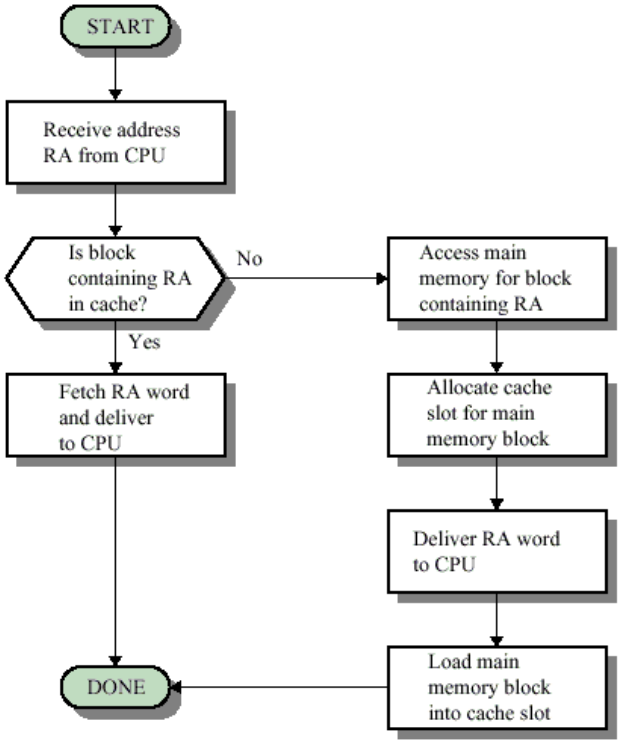


Cache Memory Configuration



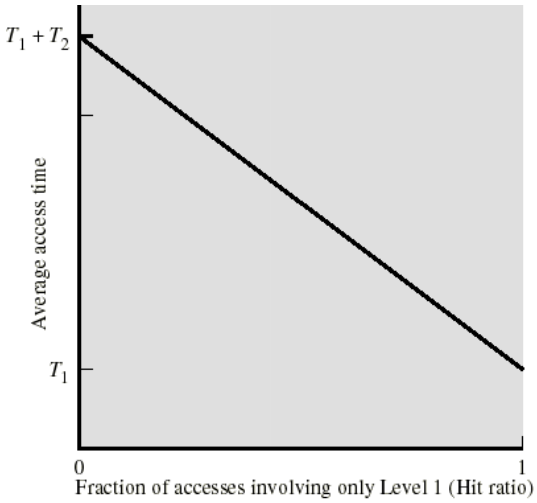


Cache Read Operation



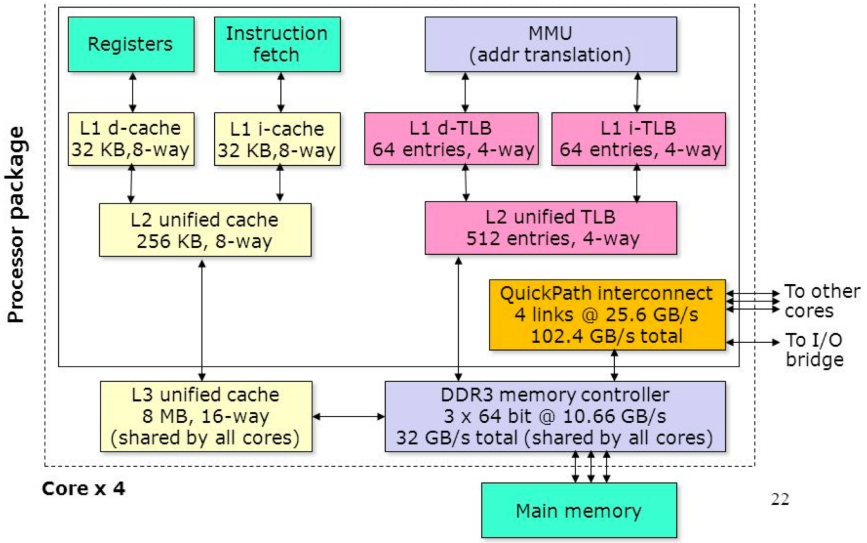
The Hit Ratio

- Hit ratio = fraction of access where data is in cache
- T1 = access time for fast memory
- T2 = access time for slow memory
- T2 >> T1
- When hit ratio is close to 1 the average access time is close to T1



Cache issues in modern CPUs…

- Multiple cores (sharing/consistency?)
- Address translation (virtual or physical addresses?)
- Speculative instruction execution !!
- Example:
  Intel Core i7



22

Locality of Reference

- Memory reference for both instruction and data tend to cluster over a long period of time.
- Example: once a loop is entered, there is frequent access to a small set of instructions.
- Hence: once a word gets referenced, it is likely that nearby words will get referenced often in the near future.
- Phenomenon extensively studied by Peter Denning

Implications for Architecture and OS?

Small cache leads to large improvement in performance

Disk (Buffer) Cache

- A portion of main memory used as a buffer to temporarily to hold data from the disk
- Needed due to memory/disk access time disparity
- Maintained by OS
- Used in a similar manner to the processor cache
- Locality of reference also applies here: once a disk block gets referenced, it is likely that nearby blocks will get referenced often in the near future.
- If a block referenced is not in the disk cache, the disk sector containing the block is read into the disk cache.

Virtual Memory

- A **major** aspect of modern computing systems
- Application processes do not reference physical memory addresses, but rather virtual memory
- Each application process executes in its own address space, (e.g., defining text/data/stack regions of the memory of the process)
- The architecture and the operating system collaborate to dynamically translate virtual addresses to physical addresses.
- This mapping is done on a page basis (e.g., 4KB)

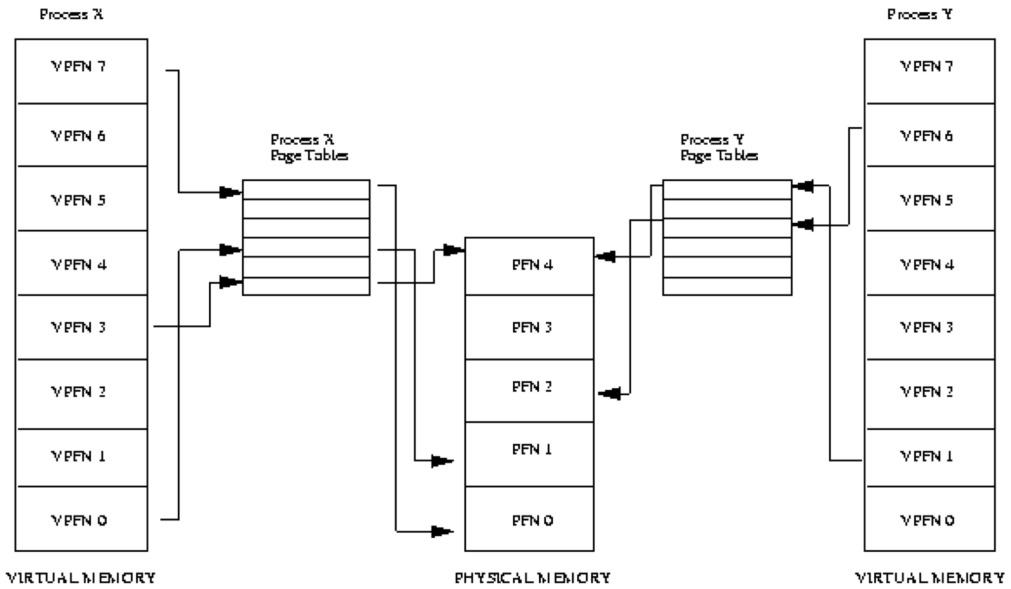
Paging

- Divide virtual and physical memory into fixed size units, called pages and page frames, respectively.
- Mapping virtual address to physical address requires
  - mapping the encompassing page of virtual memory to a page frame in physical memory
  - then adding the corresponding offset within the page
- Demand paging – bring into main memory parts of program only as they are needed
- All modern general-purpose computing systems are demand paged.

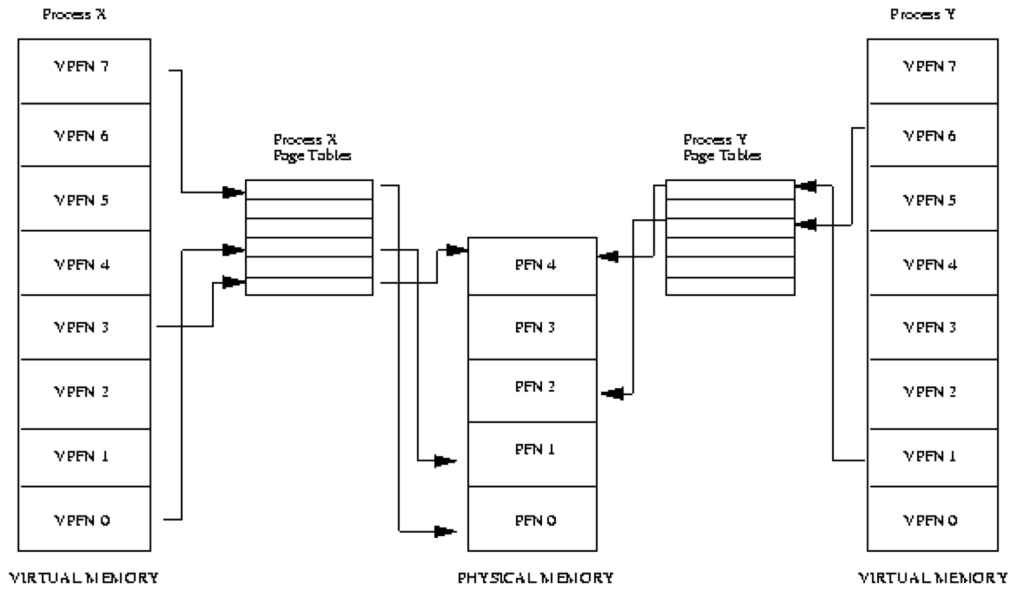
Address Translation

- For each application process, the OS maintains a set of page tables mapping virtual pages to physical pages.
- For every memory reference, the virtual address needs to be translated to a physical address
- Note: some pages may not be memory-resident. If referenced, a page fault is generated, and the OS loads the page from disk, possibly requiring some other page to be written back to disk.

Illustration



Illustration

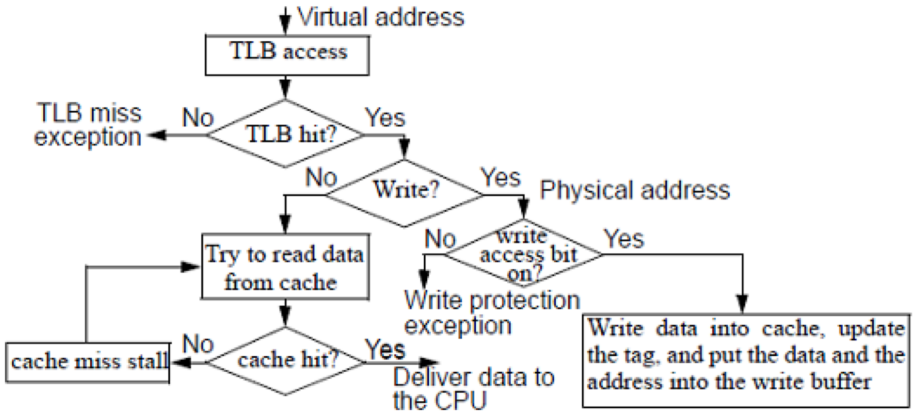


Translation Look-Aside Buffers

- Hardware associated with the processor
- Basically a cache of virtual-to-physical address translations
- The processor checks the TLB concurrently with accessing main memory, aborting the latter if the TLB holds the desired translation.
- If the translation is not present in the TLB, memory is accessed and the translation is loaded into the TLB

Logic of a memory access

- Memory management unit (MMU) – hardware that manages address translation for the processor



Memory Management Roles of the OS

- Maintaining page tables for each process, which typically has multiple areas within its address space
- Supporting sharing of areas among processes Example? Sharing text
- Enforcing address space protection among processes
- Implementing demand paging/managing swap space
- Implementing page replacement policy
- Maintaining set of free page frames

I/O Communication Techniques

- Three techniques are possible for I/O operations
  - Programmed I/O
    - Does not use interrupts: CPU has to wait for completion of each I/O operation
  - Interrupt-driven I/O
    - CPU can execute code during I/O operation: it gets interrupted when I/O operation is done.
  - Direct Memory Access (DMA)
    - A block of data is transferred directly from/to memory without going through CPU

Multiprocessor Issues

- Many operating systems have been written to run on uniprocessors
- In years past, multiprocessor operating systems were the exception
- Now they are the norm.
- How does a multiprocessor OS differ from a uniprocessor OS?

Summary

- Although most parts of the operating system are written in code that is independent of the architecture, the design and operation of the OS are heavily influenced by key features of the HW.
  - processor and interrupt architectures
  - memory hierarchy
  - memory management mechanisms
  - I/O subsystem
  - multiprocessor and multicore platforms
- As architectures evolve, so do operating systems.
- However, many basic concepts remain the same.

Moving Forward

- Review some key developments in operating systems over the years
- Briefly introduce the (current) three main players:
  - Unix/Linux
  - Windows
  - Mac OS X
- Then move on to the details of implementing processes and threads.