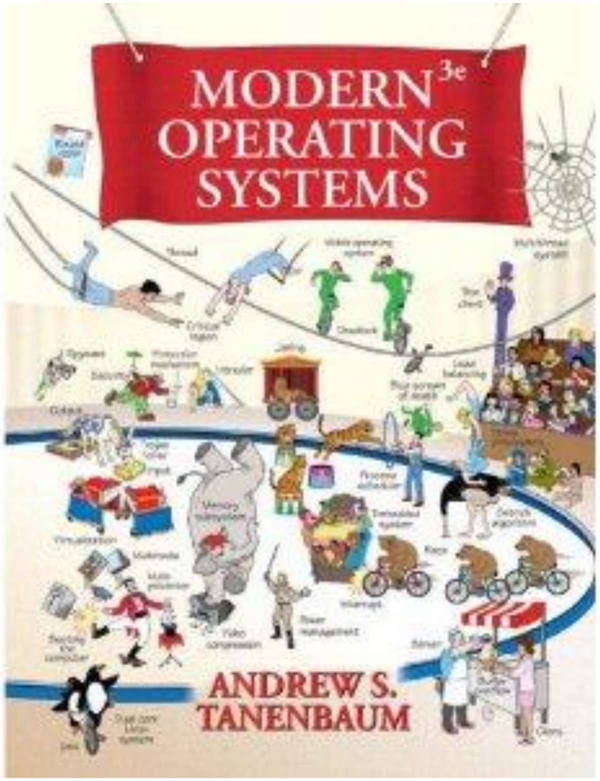
CMPT 300
Introduction to Operating Systems

Course Organization

CMPT 300: Operating Systems

- Time: Wed 5:30-8:20pm
- Location: RCB 8100
- Textbook:
  - Modern Operating Systems, **Third edition**, Andrew S. Tanenbaum, Prentice Hall 2008



Contact Information

- Instructor: Zonghua Gu
- Office: TASC 1, Room 9215
- Office Hours: Wed 3:00-5:30 pm or by appointment
-
- Email/MSN: zonghua(at)gmail.com
- QQ: 59331972

Web-site

- All the information discussed today and more can always be found on the class web-site
- To find the class web site go to http://www.cs.sfu.ca/CourseCentral
  - Select Course Home pages
  - Select the Homepage for CMPT 300
    - 2 sections: Dr. Evans (300D) and Dr. Gu (300E)
    - http://www.cs.sfu.ca/CourseCentral/300/csguestk is 300E

Topics

- History, Evolution, and Philosophies
- The User's View of Operating System Services
- Tasking and Processes
- Inter-process Communication, Concurrency Control and Resource Allocation
- Scheduling and Dispatch
- Physical and Virtual Memory Organization
- File Systems
- Security and Protection

CMPT 300
Introduction to Operating Systems

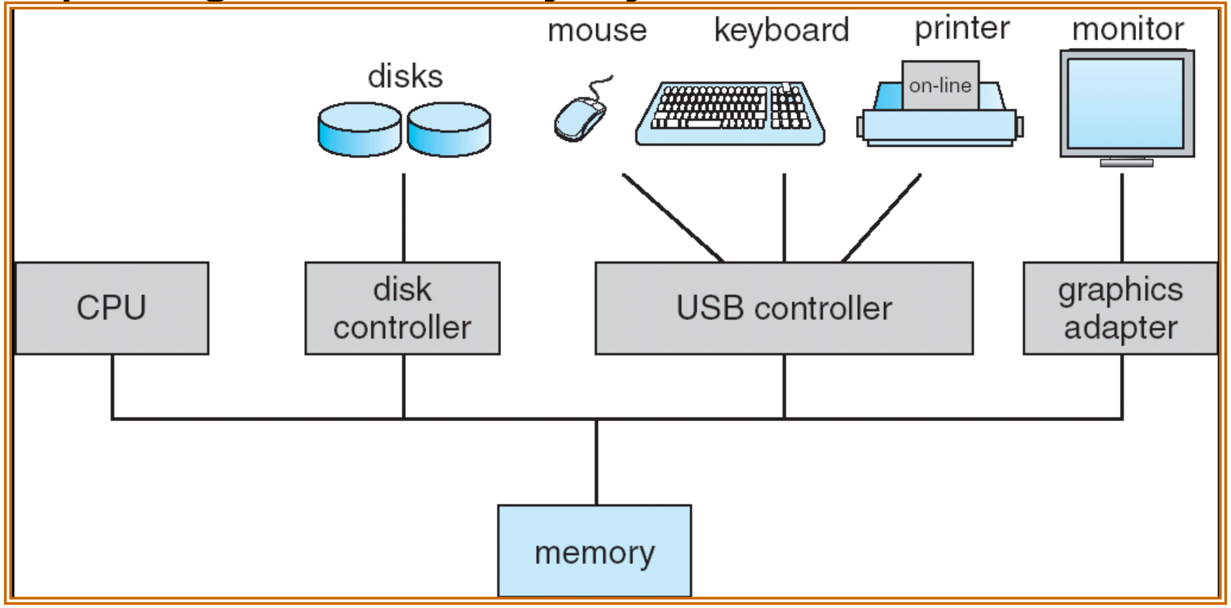
Introduction / Review

Hardware and Software

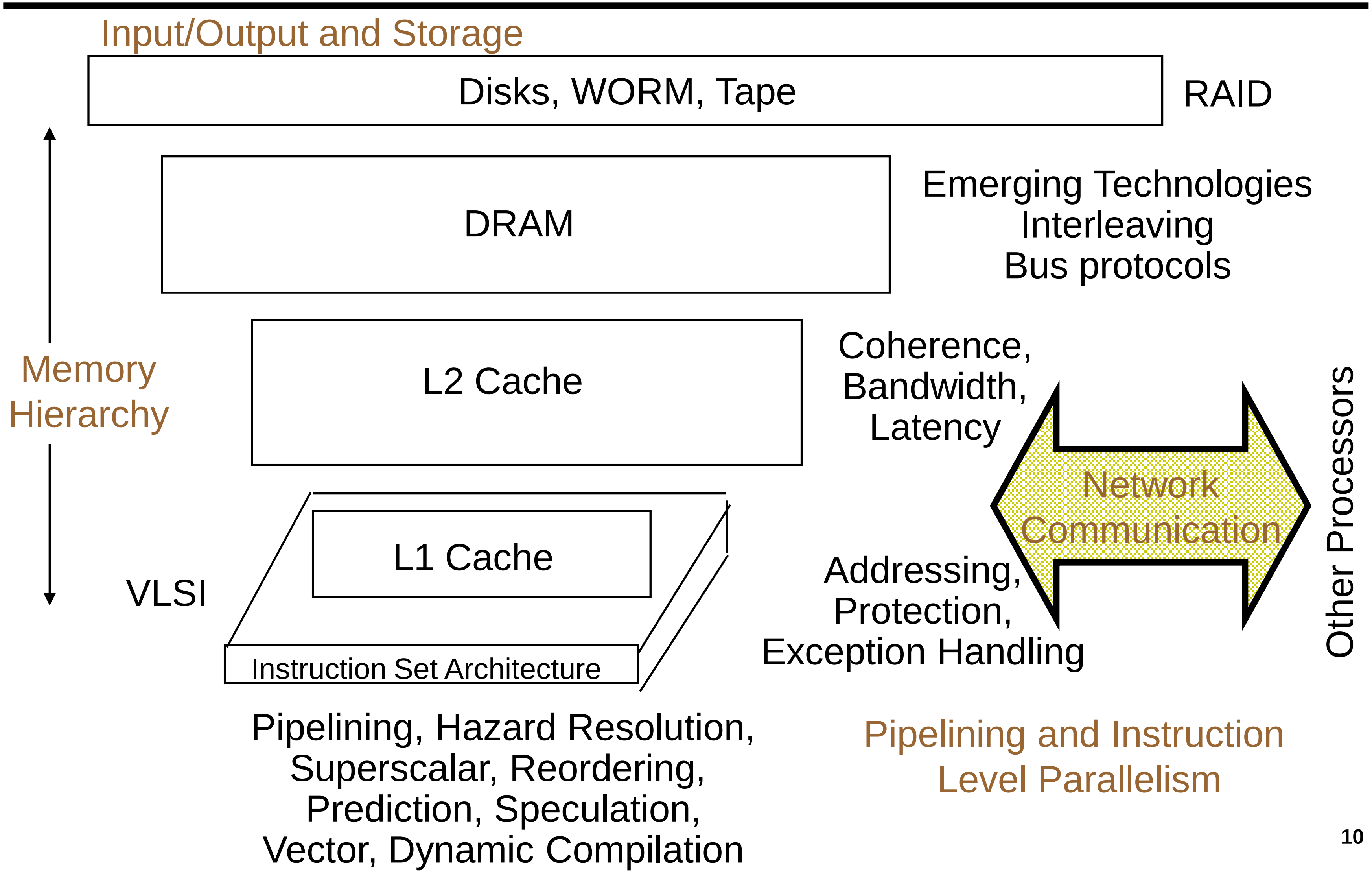
- A **computer** is a machine designed to perform operations specified with a set of instructions called a **program.**
- **Hardware** refers to the computer equipment.
  - keyboard, mouse, terminal, hard disk, printer, CPU
- **Software** refers to the programs that describe the steps we want the computer to perform.
- The software that manages the hardware and shares the hardware between different application programs is called the **operating system.**

Computer Hardware

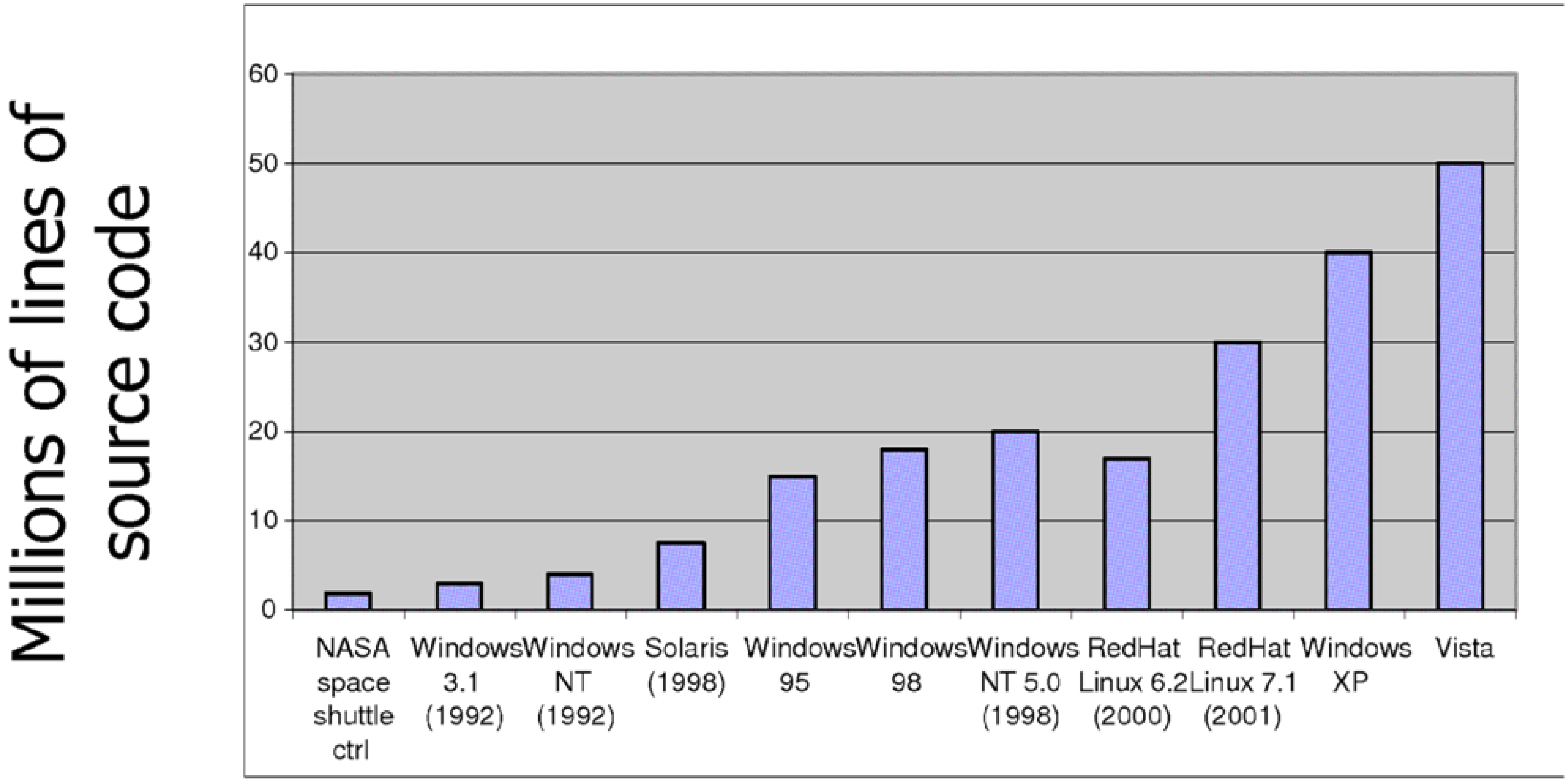
- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles



Sample of Computer Architecture



Increasing Software Complexity



From MIT's 6.033 course

How do we tame complexity?

- Every piece of computer hardware different
  - Different CPU
    - Pentium, PowerPC, ColdFire, ARM, MIPS
  - Different amounts of memory, disk, …
  - Different types of devices
    - Mice, Keyboards, Sensors, Cameras, Fingerprint readers, touch screen
  - Different networking environment
    - Cable, DSL, Wireless, Firewalls,…
- Questions:
  - Does the programmer need to write a single program that performs many independent activities?
  - Does every program have to be altered for every piece of hardware?
  - Does a faulty program crash everything?
  - Does every program have access to all hardware?

Virtual Machine Abstraction

Application

Virtual Machine Interface

Operating System

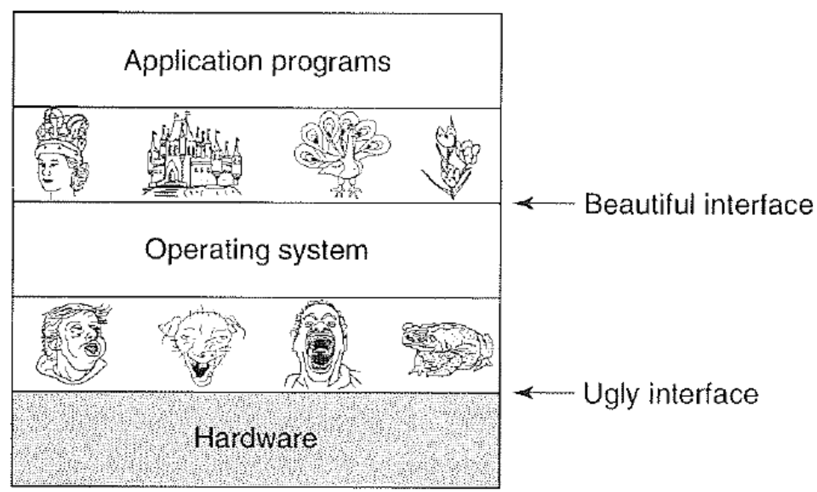
Physical Machine Interface

Hardware

- Software Engineering Problem:
  - Turn hardware/software quirks ⇒ what programmers want/need
  - Optimize for convenience, utilization, security, reliability, etc…
- For any OS area (e.g. file systems, virtual memory, networking, scheduling):
  - What's the hardware interface? (physical reality)
  - What's the application interface? (nicer abstraction)

Virtual Machines

- Software emulation of an abstract machine
  - Make it look like hardware has features you want
  - Programs from one hardware & OS on another one
- Programming simplicity
  - Each process thinks it has all memory/CPU time
  - Different Devices appear to have same interface
  - Device Interfaces more powerful than raw hardware
    - Bitmapped display ⇒ windowing system
    - Ethernet card ⇒ reliable, ordered, networking (TCP/IP)
- Fault Isolation
  - Processes unable to directly impact other processes
  - Bugs cannot crash whole machine



What does an OS do?

- Silberschatz and Gavin: “An OS is Similar to a government”
  - Begs the question: does a government do anything useful by itself?
- Coordinator and Traffic Cop:
  - Manages all resources
  - Settles conflicting requests for resources
  - Prevents errors and improper use of the computer
- Facilitator (“useful” abstractions):
  - Provides facilities/services that everyone needs
  - Standard Libraries like Windowing systems
  - Make application programming easier, faster, less error-prone
- Some features reflect both tasks:
  - File system is needed by everyone (Facilitator) …
  - … but File system must be protected (Traffic Cop)

What is an Operating System,… Really?

- Most Likely:
  - Memory Management
  - I/O Management
  - CPU Scheduling
  - Synchronization / Mutual exclusion primitives
  - Communications? (Does Email belong in OS?)
  - Multitasking/multiprogramming?

- What about?
  - File System?
  - Multimedia Support?
  - User Interface?
  - Internet Browser? ☺

Operating System Definition (Cont'd)

- No universally accepted definition
- "*Everything a vendor ships when you order an operating system*" is good approximation
  - But varies wildly
- "*The one program running at all times on the computer*" is the OS kernel
  - Everything else is either a system program (ships with the operating system) or an application program

Summary

- Operating systems provide a virtual machine abstraction to handle diverse hardware
- Operating systems coordinate resources and protect users from each other
- Operating systems simplify application development by providing standard services and abstractions
- Operating systems can provide an array of fault containment, fault tolerance, and fault recovery

Machine language

- Each type of processor (like Pentium 4, Athalon, Z80, …) has its own instruction set
- Each instruction in an instruction set does a single thing like access a piece of data, add two pieces of data, compare two pieces of data …
- Each instruction is represented by a unique number .This # may be different for different instruction sets, but no two instructions in the same instruction set should have the same #

19

Machine Language programs

- A machine language program is a list of instructions
  - Each instruction is represented by a number
  - Inside the memory of the computer, each number is represented in binary (as a string of 1’s and 0’s)
  - The long string of 0’s and 1’s is easy for the computer to understand, but difficult for a human to read or write

Assembly

- Assembly languages make it easier for the programmer.
  - Assembly is easier for humans to read/write
  - Use mnemonics like ADD, CMP, … to replace the numbers that identify each of the instructions in the instruction set
  - The code for an Assembly program is written into a text file, which is translated into machine language program and executed.

Computer Software: Languages

- Some Computer Languages
  - Machine language (machine instruction set)
  - assembly language
  - high level languages (Compilers/Interpreters)
    - C, C++, Ada, Fortran, Basic, Java
    - Do YOU know of any others?
    - mathematical computation tools (MATLAB, Mathematica, ...)
- Application software is written using computer languages.
- Operating systems are also written using computer languages (often C, some assembly)

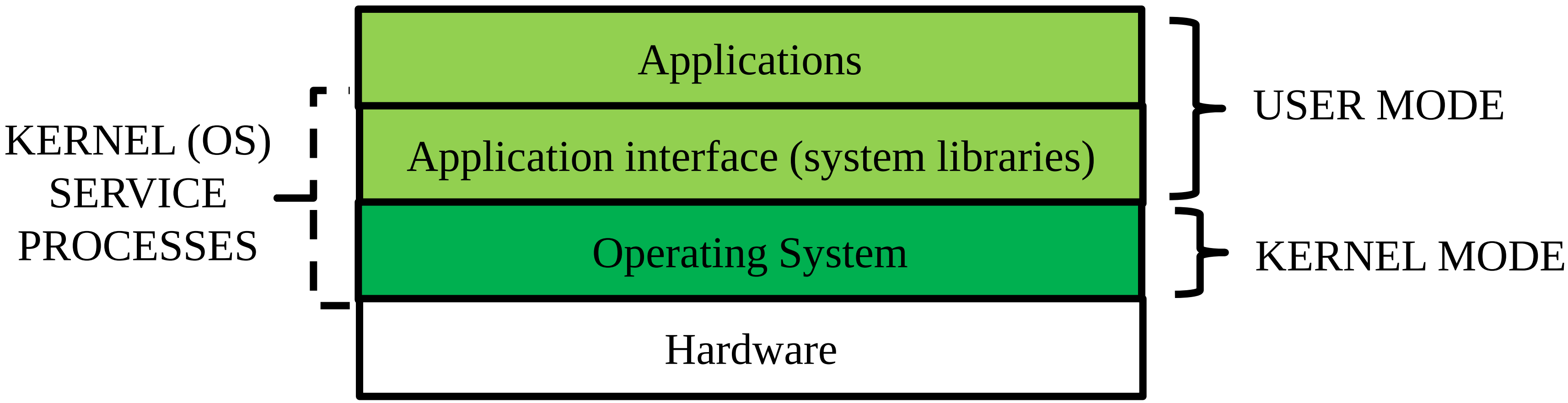
Computer Software: Applications

- Application Software (Software Tools)
  - Word processors (Microsoft Word, WordPerfect, ...)
  - Spreadsheet programs (Excel, Lotus1-2-3, ...)
  - Computer games
  - Communication software (email, chat, web browser…)
  - Telecommunication software (VOIP, …)
  - Integrated programming environments

User mode / kernel mode

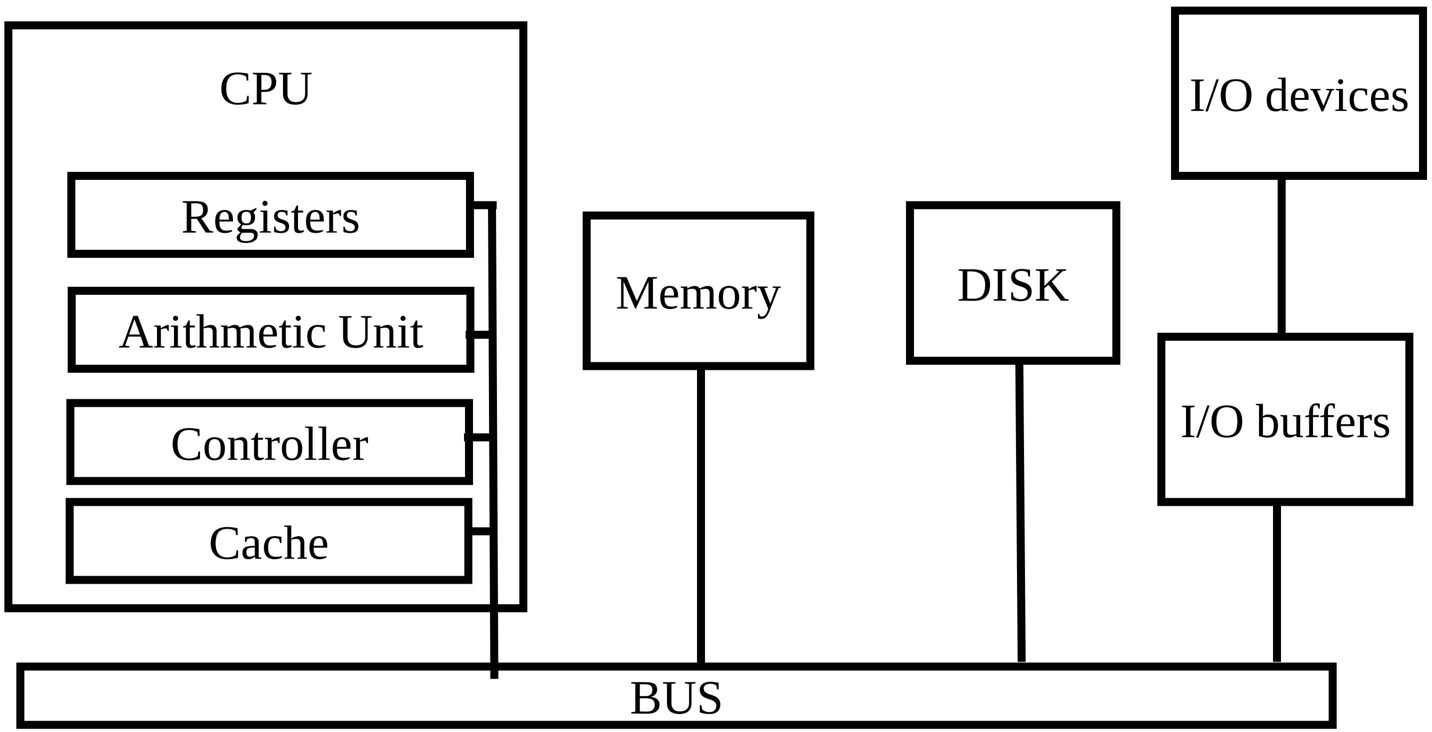
- Most application software runs in user mode. Applications have access to a subset of the instruction set that does not include most direct hardware access
- Operating systems run in kernel mode (supervisor mode) and have access to the complete instruction set, including the instructions used to directly manage the hardware
- Application software running in user mode can used system calls to access hardware managed by the Operating System
- User mode programs may perform duties for the OS

Modes



For some operating systems there may not be a separation between kernel mode and user mode (embedded systems, interpreted systems)

Basic computer configuration



Registers in CPU

- Data registers
- Accumulator
- Address registers
- Control/Status registers
  - Program counter
  - Stack pointer
  - Instruction register
  - Status registers

Controller

- Fetch, Decode, Execute cycle (each instruction)
  - Fetch next instruction: Instruction contains op-code and possibly data
  - Decode op-code
  - Execute op-code (using data if necessary)
- Instructions
  - access data, moving it from memory (or disk or cache) to/from registers, and between registers
  - Complete arithmetic and logical manipulations of data in registers

Executing an instruction (1)

- Examine program counter
- Fetch instruction indicated by program counter

N binary digits		integer
opcode	Address(es) of data	instruction

- Increment program counter to point at next instruction to be executed
- Place fetched instruction in instruction register

Executing an instruction (2)

- Decode the instruction

opcode	Address(es) of data

instruction

- Determine what is to be done
- If needed, load address into an address register

Executing an instruction (3)

- Execute the instruction in the instruction register, may result in one or more of the following
  - Fetch any data from memory (locations given in instruction) and place into the appropriate data registers
  - Place results from a data register or the accumulator into a memory location indicated in the instruction
  - Operate (arithmetic or logical operation) on data in data registers and save the result in the indicated register
  - Control the flow of the program (for example change the value in the program counter register)

Adding 2 numbers (Z=X+Y)

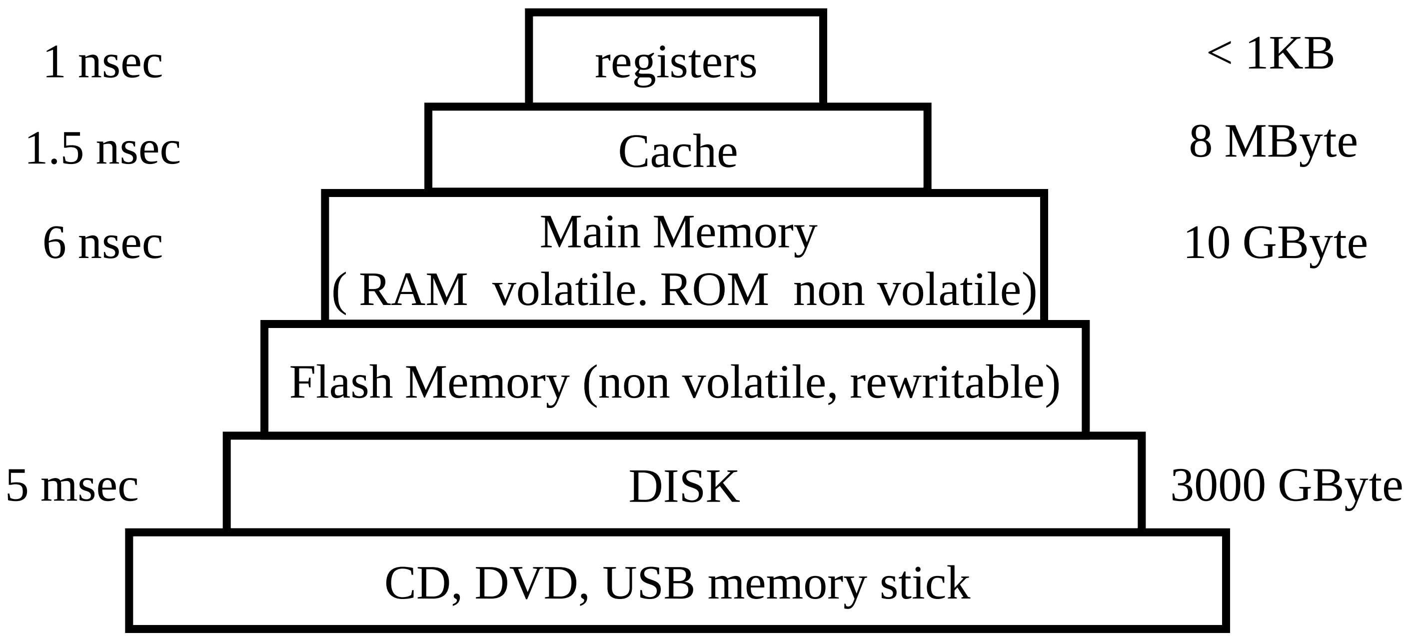
- Program counter points to instruction to load value at location X
- Instruction is fetched into the instruction register, decoded and executed to load the first number into data register A
- Program counter is incremented and now points to an instruction to load value at location Y
- Instruction is fetched into the instruction register, decoded and executed to load the second number into data register B
- Program counter is incremented and now points to an instruction to add the values in data register A and B
- Instruction is fetched into the instruction register, decoded and executed to add the two numbers and place the result in the accumulator register.
- Program counter is incremented and now points to an instruction to place the value in the accumulator register into memory location Z
- Instruction is fetched, decoded and executed to place result in Z
- (See animation)

Memory Hierarchy

- Different types of memory have different access speeds and costs
- Faster access speed implies higher cost
- Greater capacity often implies lower access speed
- From fastest access to slowest access
  - Registers
  - Cache
  - Memory
  - Disk
  - Tapes

Memory

- Modern computers use several kinds of storage



Memory Hierarchy

- As you go down the pyramid
  a) Decreasing cost per bit, Increasing capacity
  b) Increasing access time, Decreasing frequency of access

  - Note that the fastest memory, sometimes referred to as **primary memory**, is usually **volatile** (register, cache, main memory)
  - Non-volatile (continues to store information when the power is off) memory is usually slower. Referred to as **secondary** or **auxiliary memory**. Examples flash memory (flash that holds the BIOS, or removable flash), internal and external hard drives, CD, tape, …

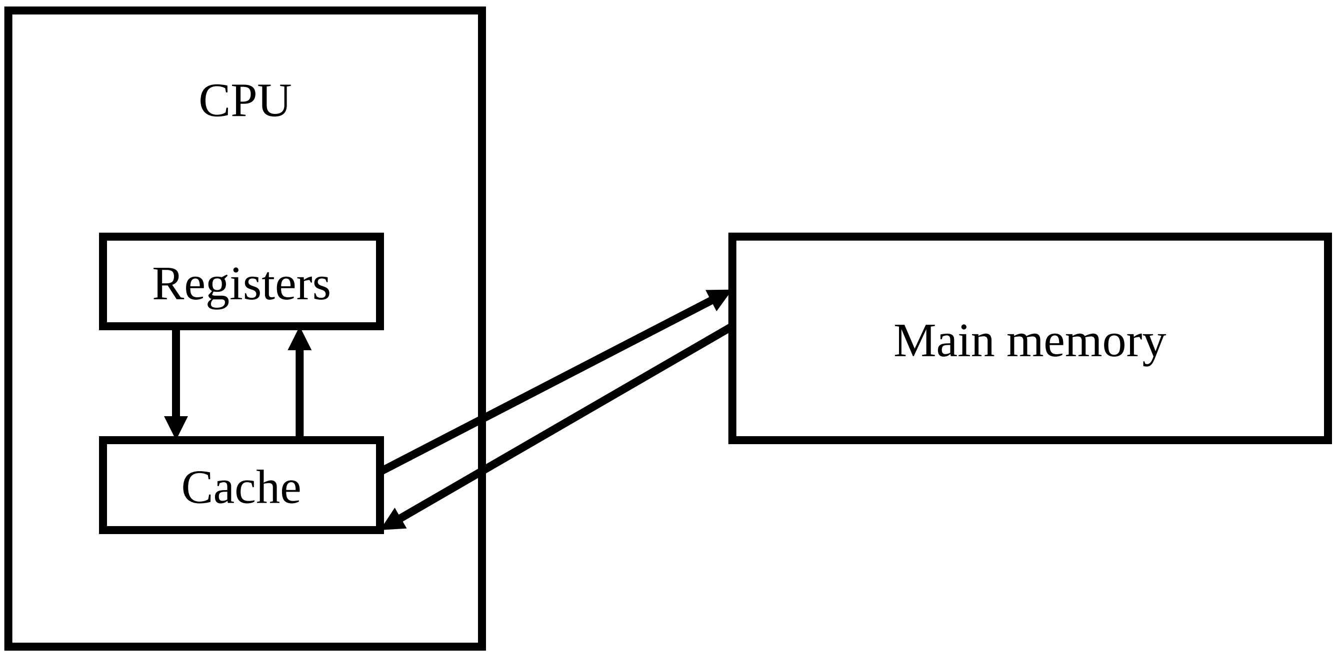
Registers and cache

- Parts of the CPU
- Register access speed comparable to CPU clock speed
- Cache memory may be as fast or as much as several times slower
- Registers
  - Usually 64x64 for 64-bit machine, 32x32 for 32-bit machine
  - Usually < 1 Kbyte
- Cache
  - As much as 8Mbytes

Concept of Cache

- Provide memory on the CPU that is slower that the registers, cheaper and larger than registers, but can be accessed much faster than main memory
- The next few instructions, and data that will be needed will be loaded into cache in anticipation of faster access by the processor.

Cache and main memory



Using Cache

- Instructions (and data) are generally moved from main memory to cache in blocks; one such block (N bytes of memory) is called a cache line
- Cache has a series of slots, each N byes long, and can hold a copy of one cache line.
- The main memory can contain many more cache lines than there are slots in the cache.
- Each time a copy of a new cache line is loaded into a cache slot, the original content of that slot is overwritten

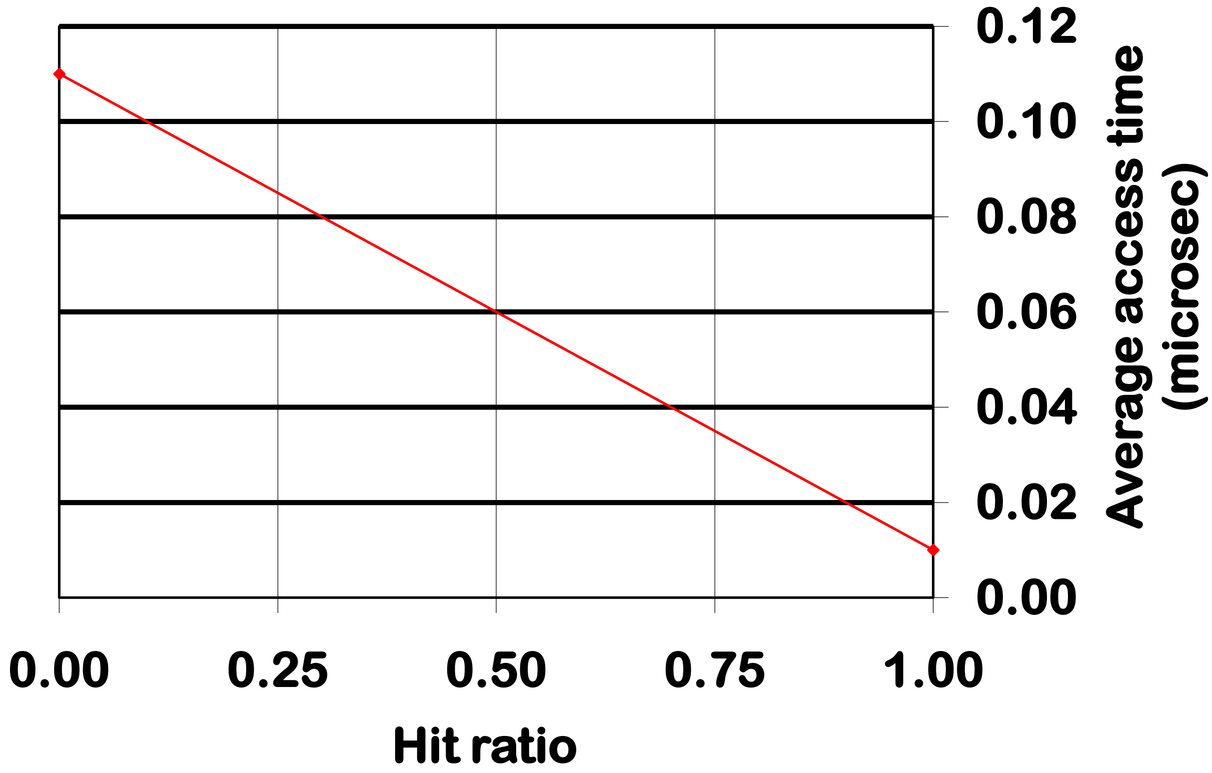
Cache design

- Cache size and Cache line size
  - Determined to optimize access time
- Mapping function
  - Which cache lines may be loaded into which cache slots
    - can any line go in any slot, or is there a mapping function to define rules governing which line can be place in which slot
- Replacement algorithm
  - When is a cache line in a cache slot replaced by another cache line

Hit ratio

- A hit occurs when a memory access finds its information in the cache.
- A miss occurs when it is necessary to access the slower main memory (or lower level cache) to find the information.
- The proportion of accesses that are hits is called the hit ratio
- Consider an example,
  - Assume that any access to the cache takes .01μs, any access to main memory takes 0.1μs (100 nsec)
  - Any instruction or piece of data not in cache will have to be accessed in main memory and moved to cache before being accessed (0.1+0.01)μs
  - For a hit ratio of x%, the average memory access time
    - 0.01*x%+0.11*(1-x%)

Hit ratio

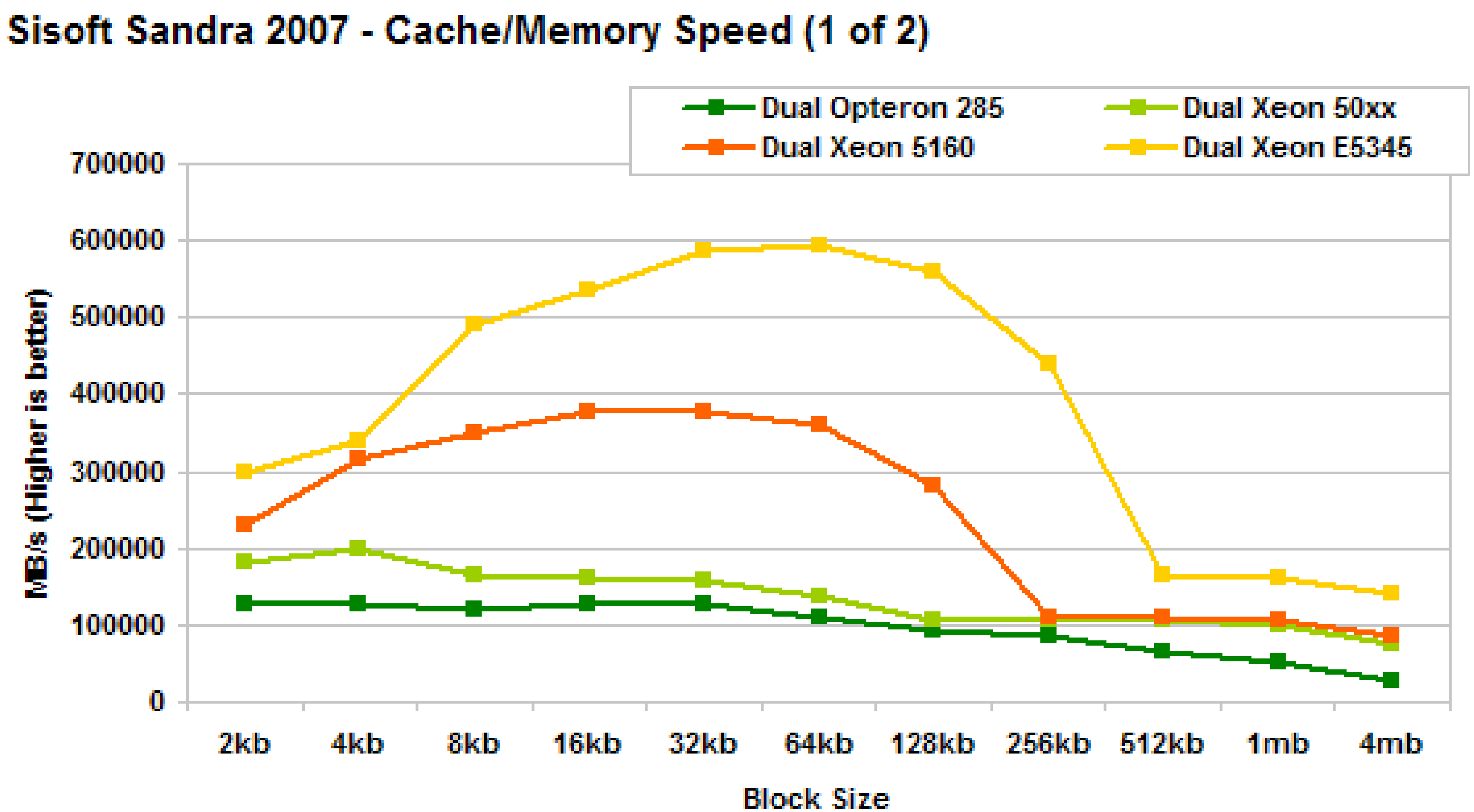


0.12
0.10
0.08
0.06
0.04
0.02
0.00
Average access time (microsec)
0.00
0.25
0.50
0.75
1.00
Hit ratio

Cache Line size

- As cache line size increases from a single byte, the hit ratio will increase at first.
  - It is very likely that bytes near a needed byte will be accessed in the near future (principle of locality (spatial & temporal))
- But as cache line size increases, the number of lines decreases
  - As cache line size increases past it's optimal value then the hit ratio will begin to decrease
  - This happens when it becomes more probable that the next access will involve the cache line that was just removed, i.e., the useful cache line was kicked out prematurely.
- Performance heavily depends on the application workload, mapping function and replacement algorithm, hence difficult to generalize.

Effect of Line size (example)

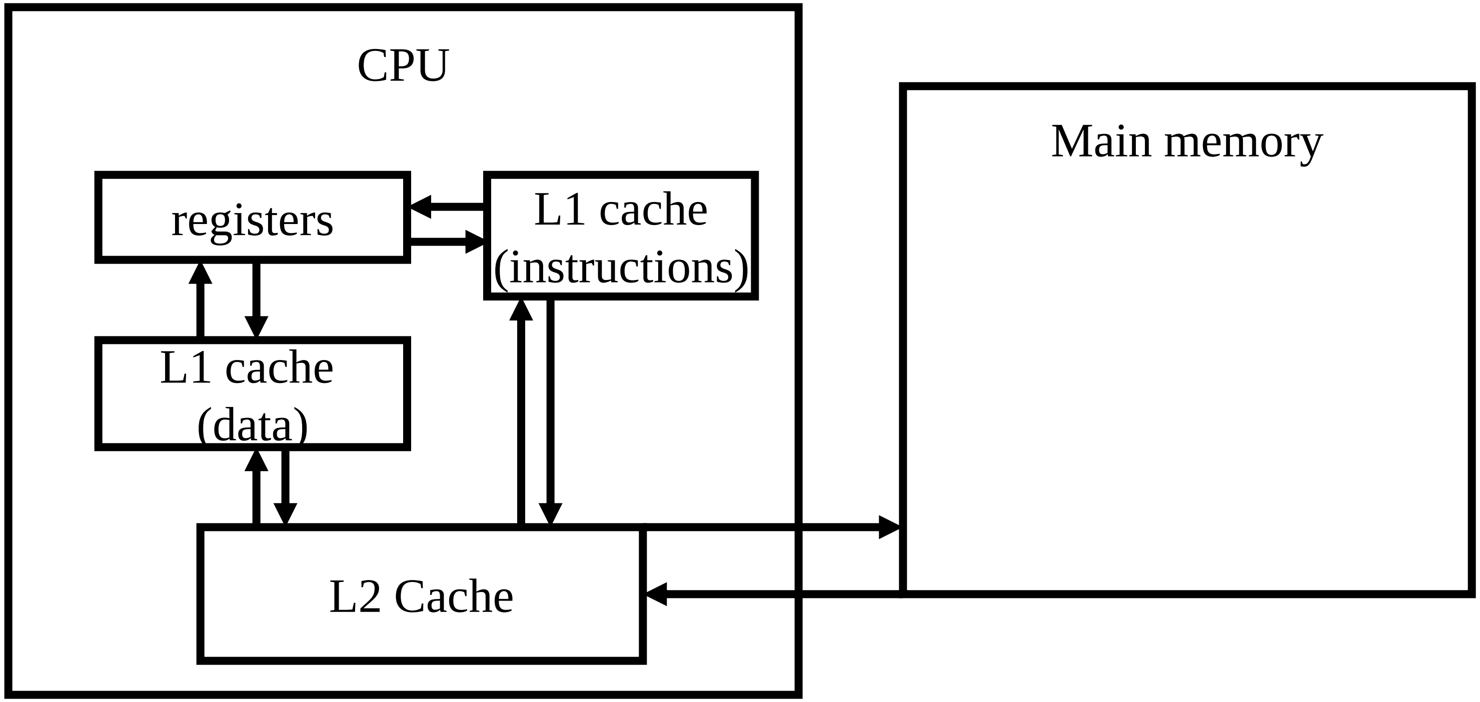


NOTE: MB/s = 1/nsec * 1000

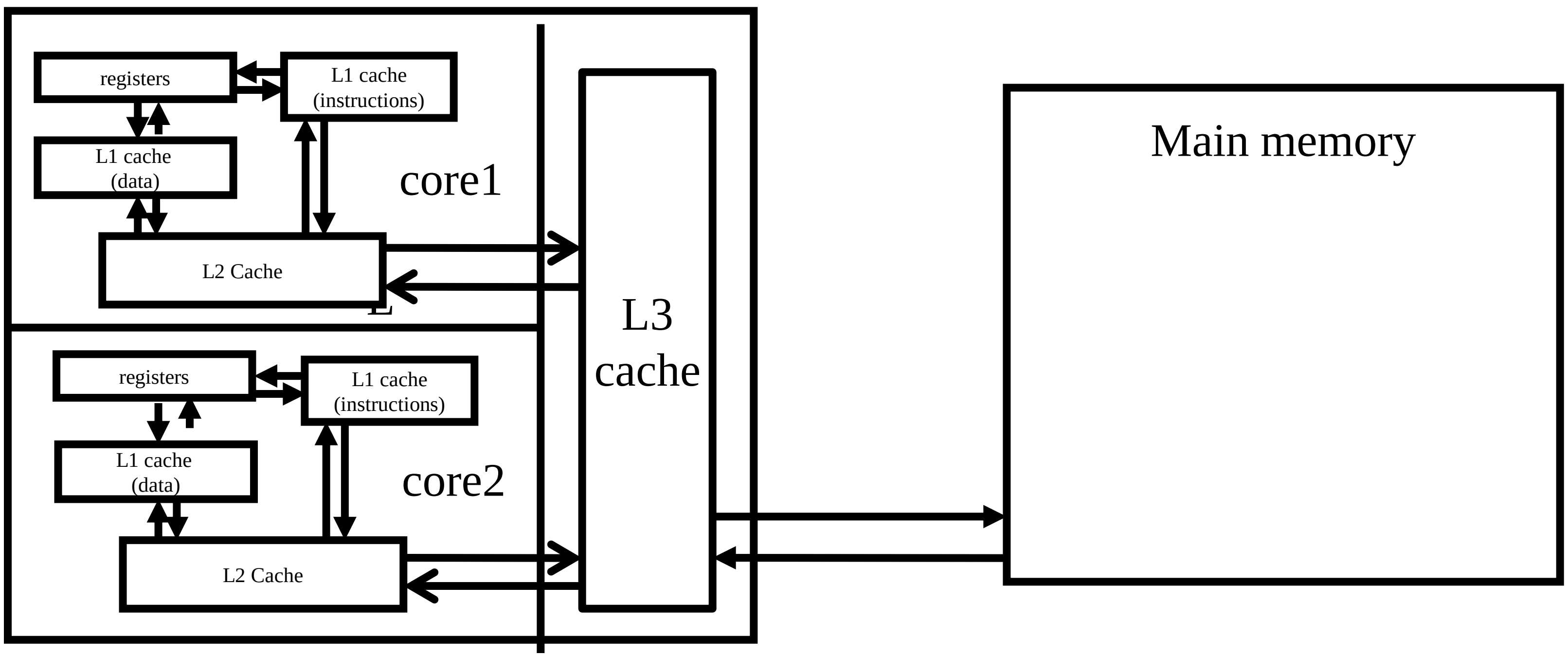
Cache specifications on common systems

- Most modern CPUs have an on-chip cache called an L1 cache
- Many modern systems also have a 2nd , and even 3rd level of cache between the L1 cache and the main memory called the L2 (L3) cache.
  - L2 cache can be on-chip or off-chip (connected to the CPU via a bus)
  - L3 cache is typically off-chip

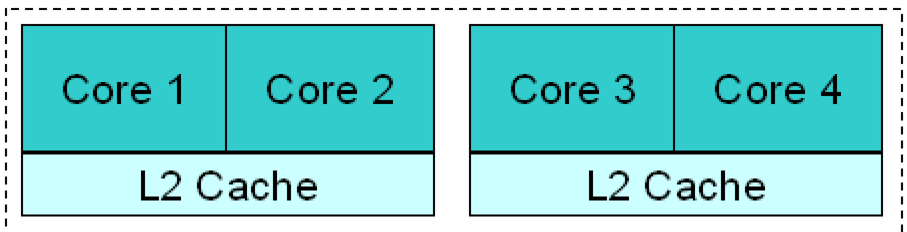
Multiple levels of cache: L2



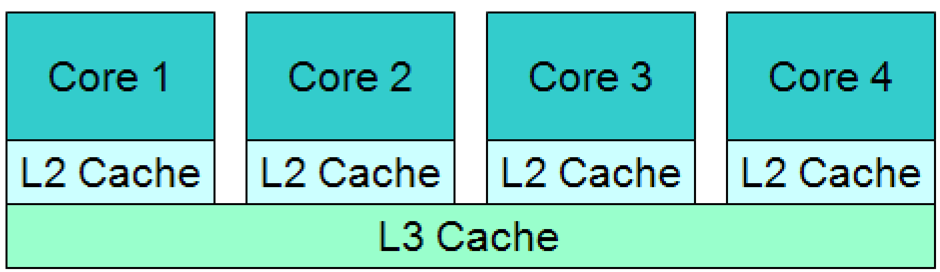
Multiple levels of cache: L3



Modern Cache Architectures



Core 2 Duo Core2 Quad



Nehalem (i5, i7) AMD K10 (Phenom 9)

On-chip L1 caches are omitted from the figure

Shared cache requires more complicated cache controller

Individual caches are more difficult to keep coherent (properly synchronized)

Memory

- Main memory is typically DRAM (Dynamic Random Access Memory)
- Cache is typically SRAM (Static Random Access Memory)
  - Smaller and faster than DRAM
- Both are volatile: contents lost when power is turned off

Disk

- Hard disk
- CD, DVD, Blu-Ray

Disk storage is much cheaper that memory

- (3GB memory or 2000GB disk about the same cost)
- Access time for disk is at least one order of magnitude slower than for memory

Input / Output

- Reading or writing data from a perepheral device is not simple
  - Each device has its own controller (hardware)
  - Each device is managed by a device driver (software to use the controller)
    - Device drivers are specific to hardware and to the operating system using the device
  - Input and output is SLOW in comparison to CPU operations.

Busy Waiting

- Reading or writing data to a device is SLOW in comparison to the time it takes to complete one CPU operation
- The CPU must send one or more instructions to the controller to make the I/O device begin to read or write. These instructions tell the controller where to find the data and/or where to store the data
- The CPU can then wait until the I/O operation is finishes.
  - While it waits the CPU will be in a loop
  - Each time through the loop the CPU will check a register in the controller to see if the I/O operation is complete
  - This is called **busy waiting.**

Alternatives to Busy waiting

- Busy waiting does not use CPU resources efficiently
- Want to use the CPU to execute other instructions while the I/O operation is being completed by the I/O device (instead of executing the busy waiting loop)
- To use the CPU for other calculations while to I/O device controller completes the I/0 operation, we need to use interrupts (a mechanism to tell the CPU when the controller completes the I/O)

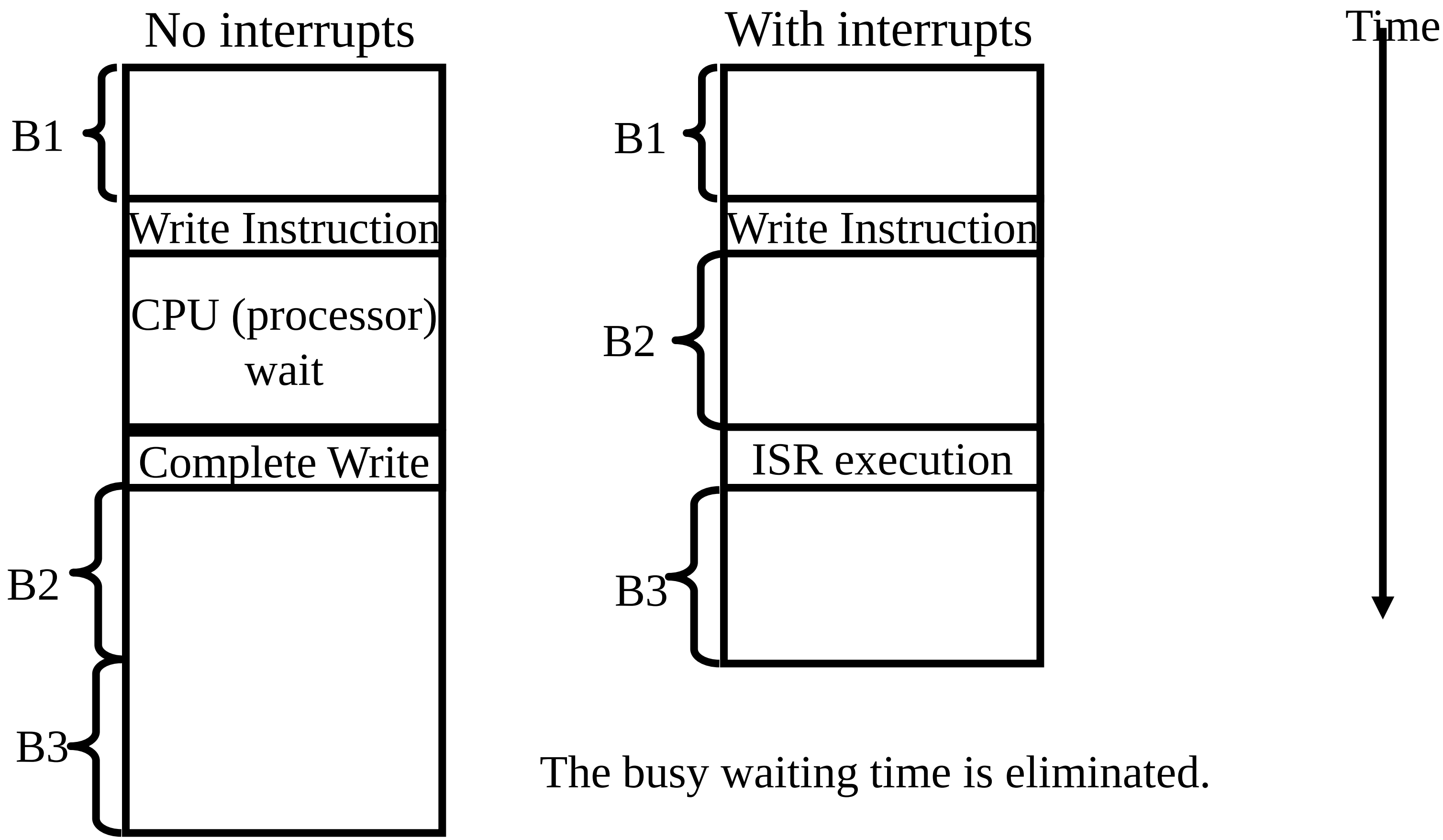
Interrupts

- Interrupts are a mechanism by which other modules (memory, I/0, timers …) may interrupt the normal sequence of instructions being executed by the processor
- Interrupts are a critical component of the operation of spooling and multiprogramming (more later).
- Interrupts allow the transfer of control between different programs (remember the OS is also a program)
- Interrupts are generated by hardware (asynchronous)
  - Exceptions are generated by particular instructions in software (synchronous), e.g., divide by 0, overflow, illegal instruction or address…

Some types of interrupts

- I/0
  - Signaling normal completion of an operation (read or write)
  - Signaling error during operation
- Timer expiry
  - Begin regularly scheduled task
  - End task that has exceeded allocated time
- Hardware failure

Increase in efficiency



The busy waiting time is eliminated.

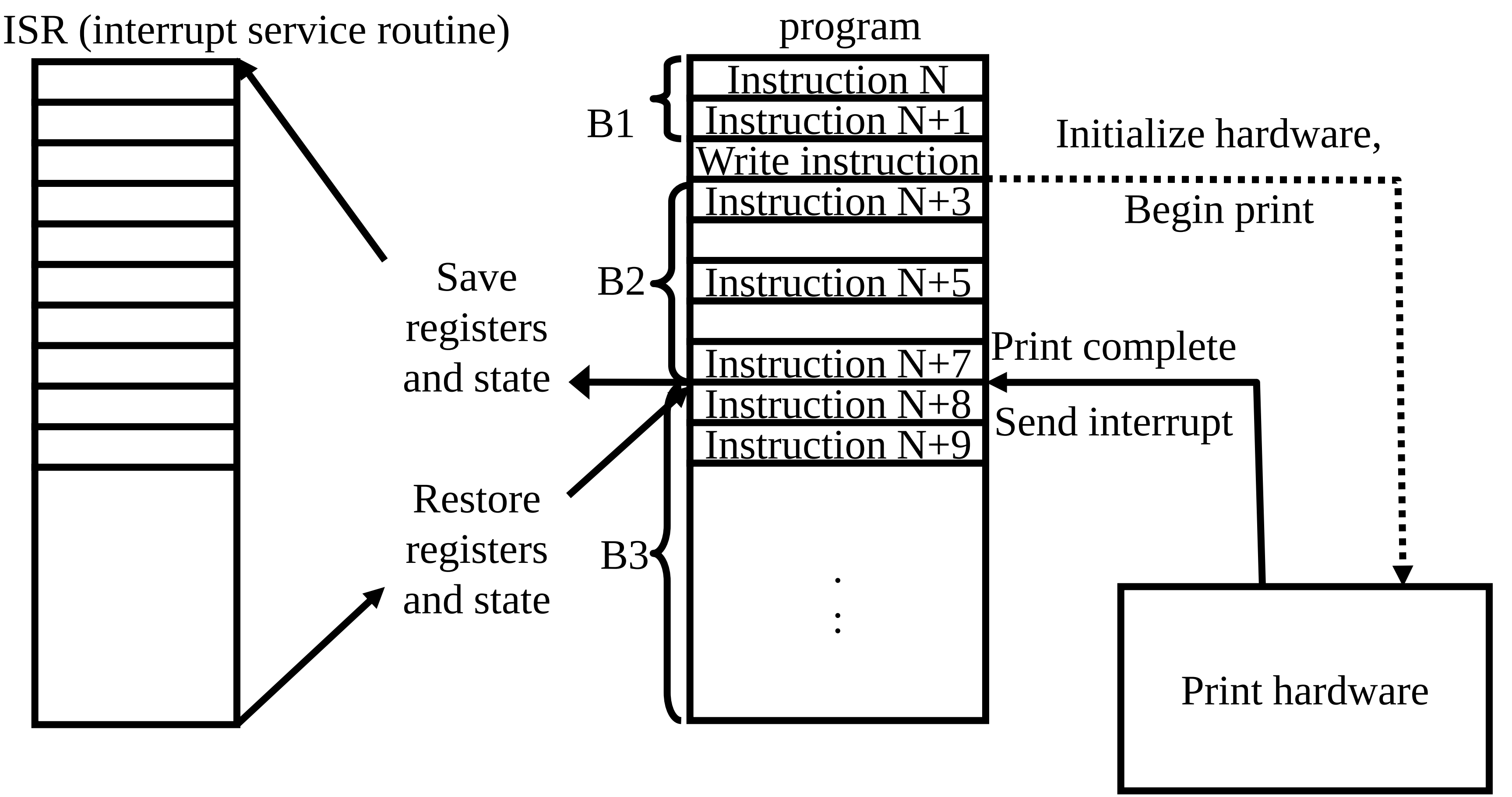
Interrupt example: Output (1)

- Program executes until it reaches a write instruction
- The write instruction sets up the hardware output device operation then leaves the output device controller (not the CPU) processing the output
- The program continues execution of additional instructions (N+3 to N+7 next slide)
- When the hardware device completes the output operation, it generates and sends an interrupt to the CPU, signaling normal completion of output

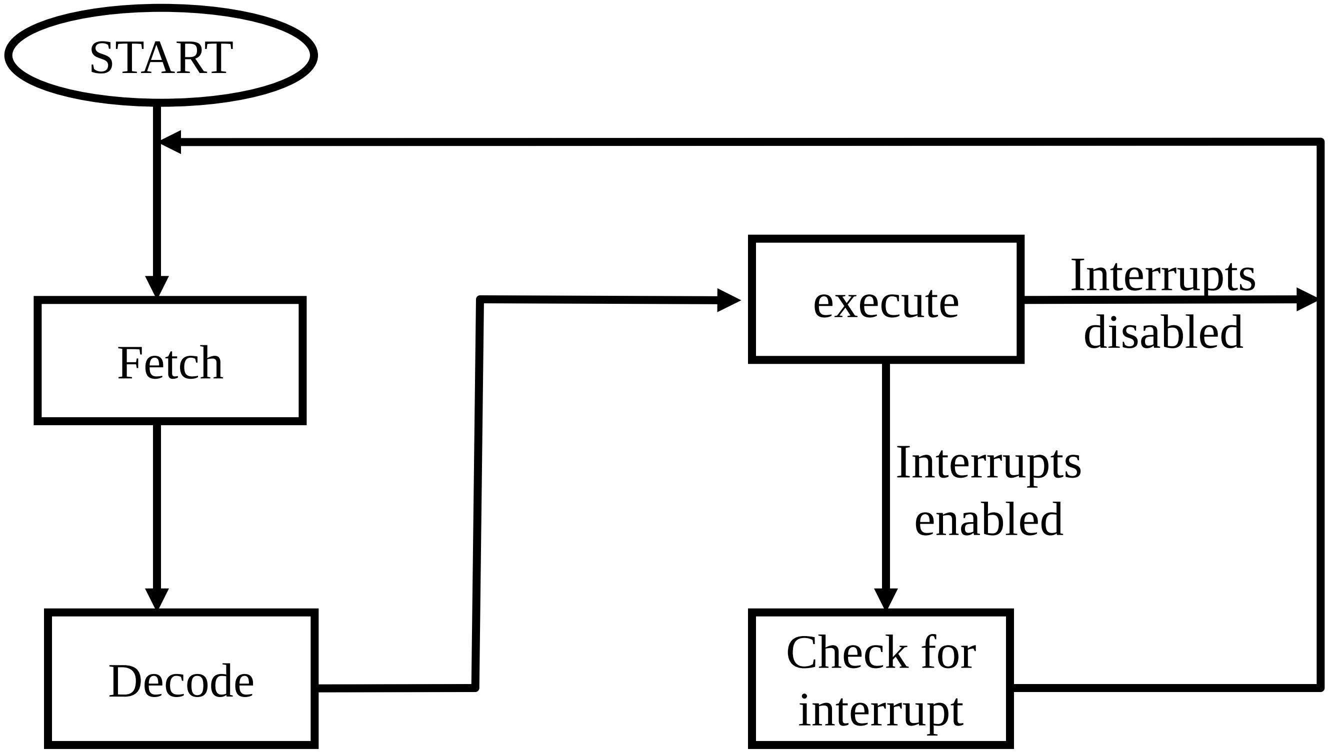
Interrupt example Output (2)

- When the currently executing instruction completes, the program is interrupted
- The program's registers and state are saved
- An ISR (interrupt service routine) completes the output operation
- Rgisters and state are restored
- The original program continues executing

Interrupt operation



Instruction cycle with interrupts



Interrupt processing (1)

- A device issues an interrupt request
- The CPU (processor) finishes execution of the present instruction
- The CPU tests to see if there is a pending interrupt, determines there is, and sends a acknowledgement to the device requesting the interrupt
- The CPU saves registers and state to the stack (including the program counter register value)
- The CPU loads the address of the appropriate ISR into the address register

Interrupt processing (2)

- The CPU executes the ISR
- When the ISR finishes, the saved register and state information is restored to the CPU registers
- The program counter is reset to point to the next instruction
- The original program continues execution

REMEMBER: the time when an interrupt occurs is not known in advance!! Interrupts are asynchronous

CMPT 300
Introduction to Operating Systems

Operating Systems
Overview Part 2: History

History of Operating Systems

- First generation 1945 - 1955
  - vacuum tubes, plug boards
- Second generation 1955 - 1965
  - transistors, batch systems
- Third generation 1965 – 1980
  - ICs and multiprogramming
- Fourth generation 1980 – present
  - personal computers

The earliest computers (1945-55)

- Built of relays, vacuum tubes
- Very large, Very slow by today's standards
- Built, programmed and maintained by the same people
- Programmed by using switches, paper tape, etc
- No operating system, single operation, single problem, sequential access

The next generation (1955-65)

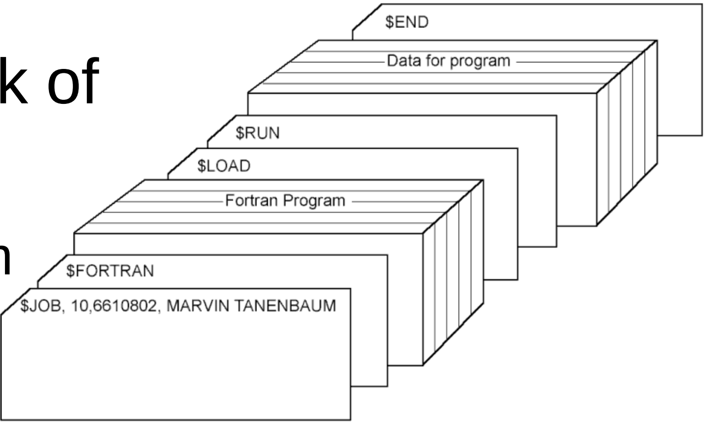
- Transistor-based, increased reliability
- The first commercial mainframes, still very large and very expensive
- Used assembly or even early high level languages like Fortran or ALGOL
- Rudimentary operating system, one program at a time, with control commands to compile, load, execute, terminate, basic compilers
- Input using cards, paper tape, magnetic tape …

Single Job to Batch

- Earliest machines had very rudimentary OS.
  - To run a job, needed to load the compiler as well as the code for the job.
  - Required a great deal of operator intervention
  - CPU not efficiently used
- Batch processing evolved to reduce the amount of time wasted setting up single jobs

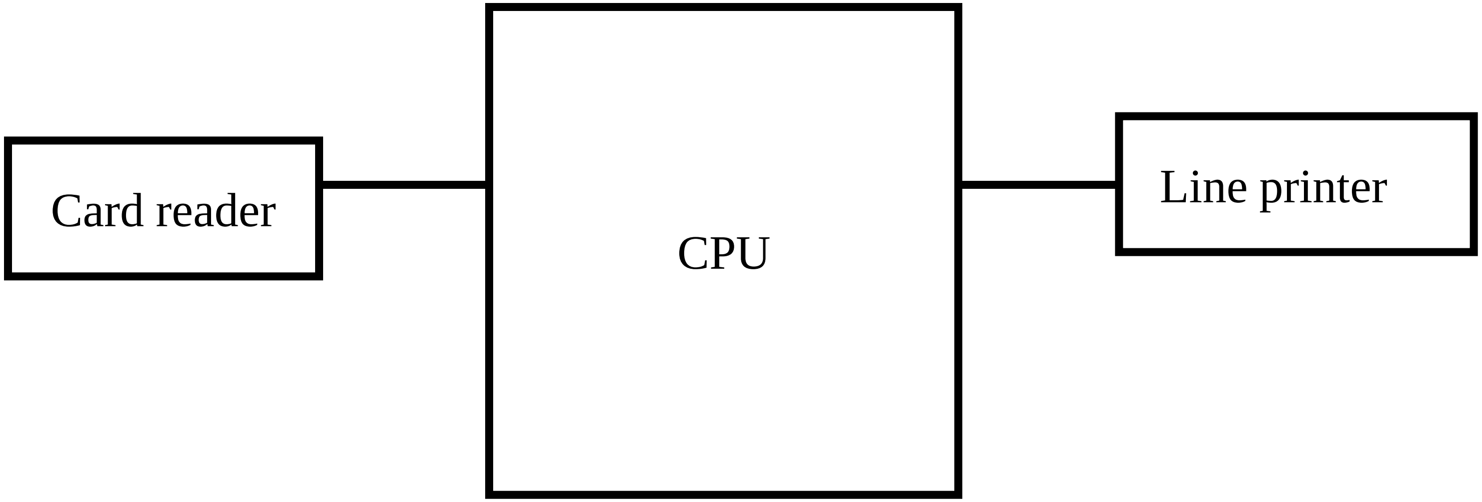
Early Batch processing

- Collect a group of jobs
  - Each job was submitted as a stack of punched cards.
    - Job card, language definition card
    - One card per line of code in program
    - Cards for load and run instructions
    - Cards containing data for program
    - End card indicating end of job



- A group of jobs was submitted as a batch to the card reader
- Each job was read in, executed, produced it output, terminated, then the next job took over the machine

Early Batch Processing



Operating System

- Commands to
  - Read a single card to memory
  - Compile to machine language
  - Place machine language code in memory
  - Start execution (load address of first instruction in program in Program counter then begin execution)
  - Write output to the line printer (or other output device)
  - Trap condition switches control from program to OS
    - END card being executed
    - Illegal opcode, divide by zero, …

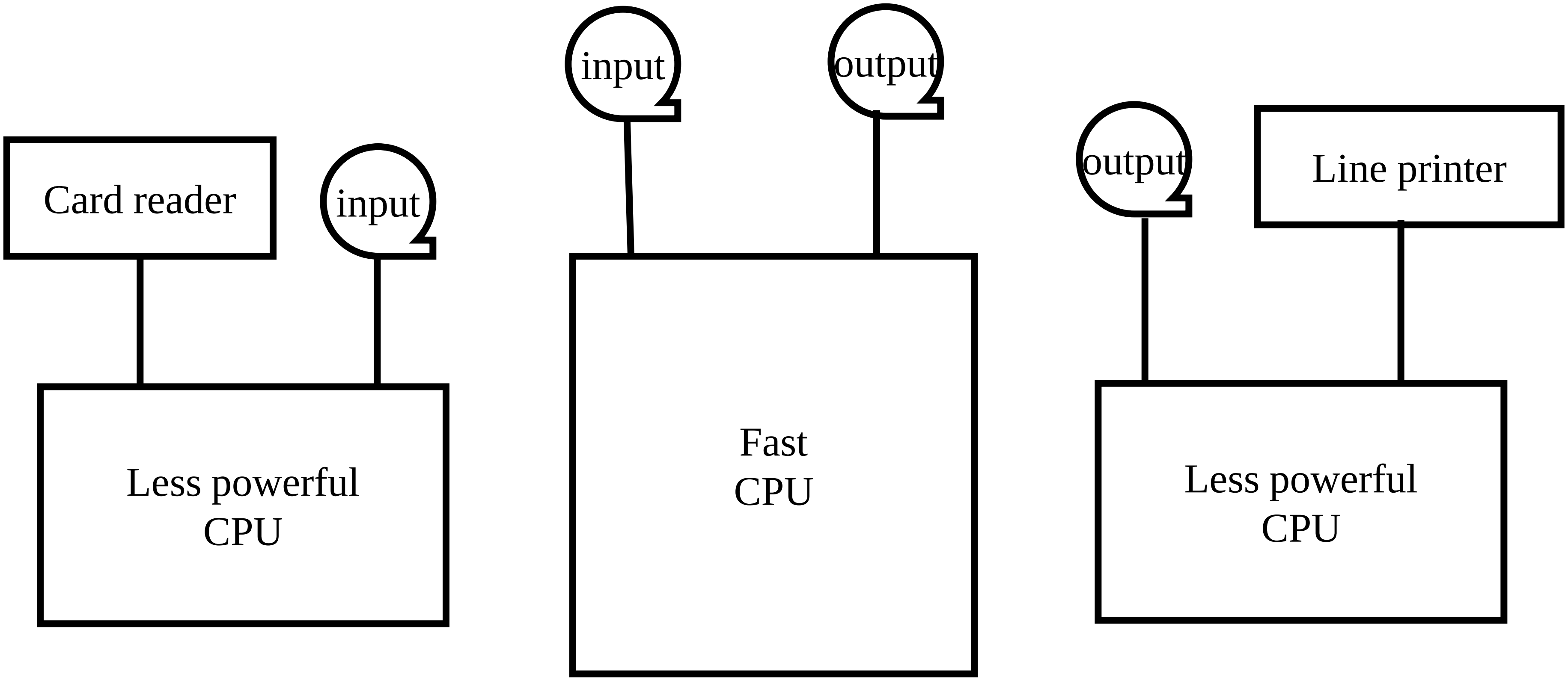
Problems with early batch processing

- Input and output, particularly from peripheral I/0 devices (card reader, line printer), are very slow when compared to the execution of other instructions
- When input and output is happening the CPU is mostly idle. An expensive resource is being wasted
- A program trapped in a infinite loop would never terminate

Improving batch processing

- Offload the slow I/O tasks to less costly and powerful computers
- Use faster I/O media (like tapes) for input to fast powerful machine
- Add timers, if your time runs out an interrupt is generated which terminates your program (deals with infinite loops)
- Adds complexity, improves efficiency

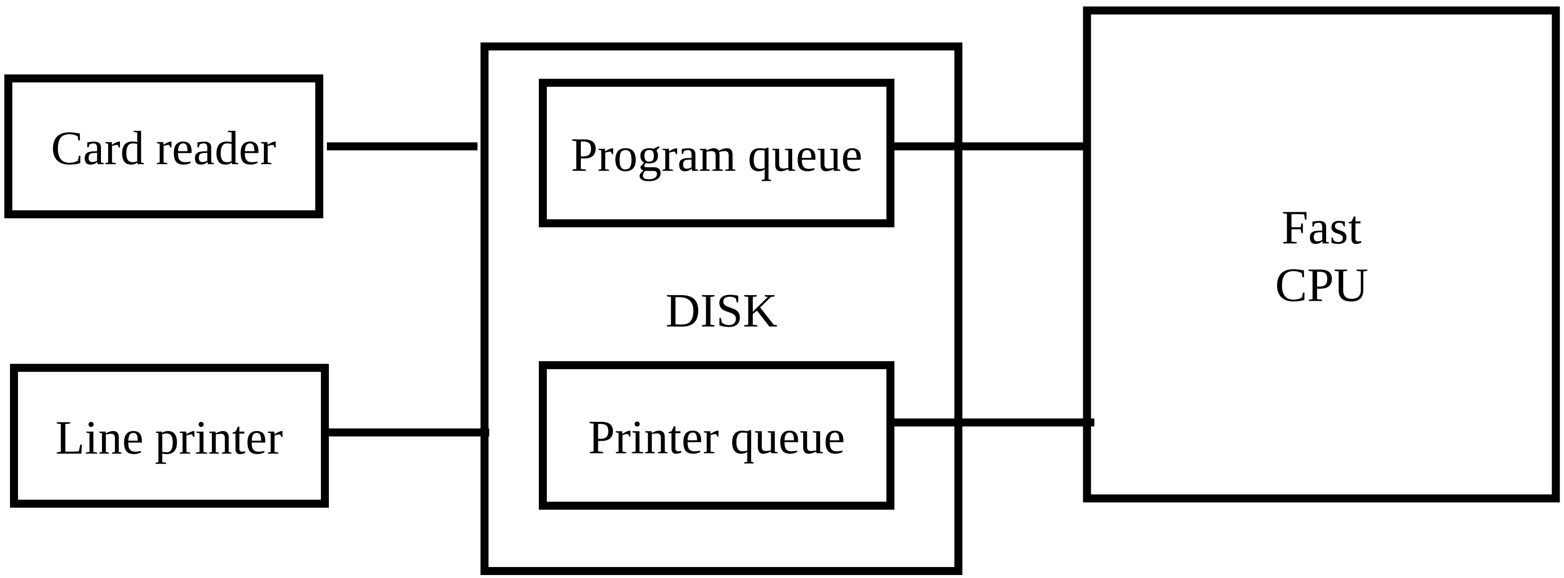
Improving batch processing



The next generation (1965-1980)

- More complicated OS
  - Deal both with I/O intensive and CPU intensive jobs efficiently
  - Multi-programming (with partitioned memory)
  - Switches between tasks
    - Load and run jobs
    - Read cards to job queue on disk (whenever card reader is ready)
    - Print results to printer from printer queue on disk (only when printer is available and job is complete)
  - Changes enabled by going from tape to disk
    - Tape is sequential-access
    - Disk is random-access, hence faster for general workload

The next generation (1965-1980)

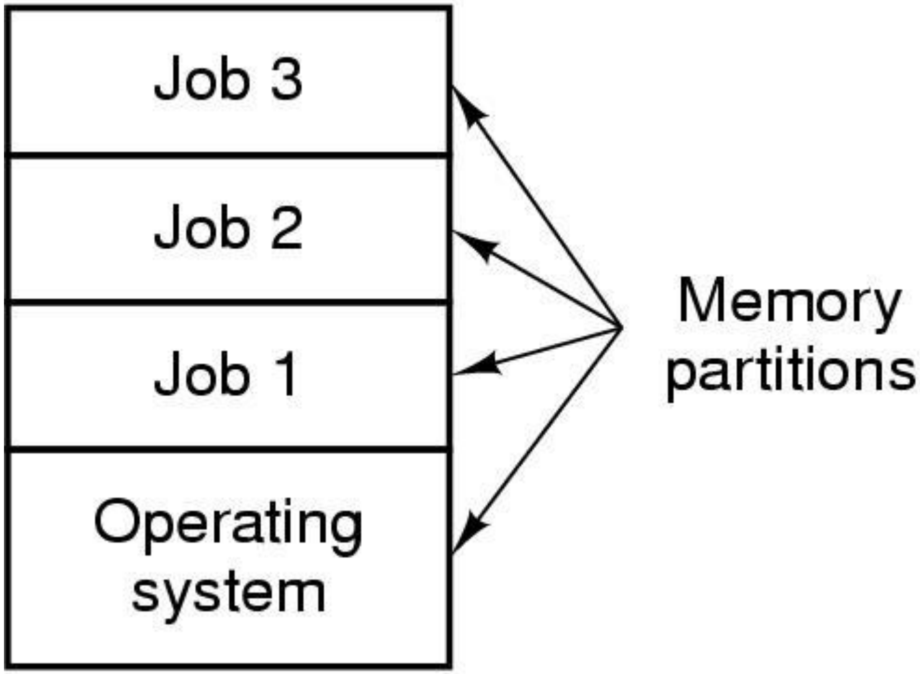


Simultaneous Peripheral Operation On Line (Spooling)

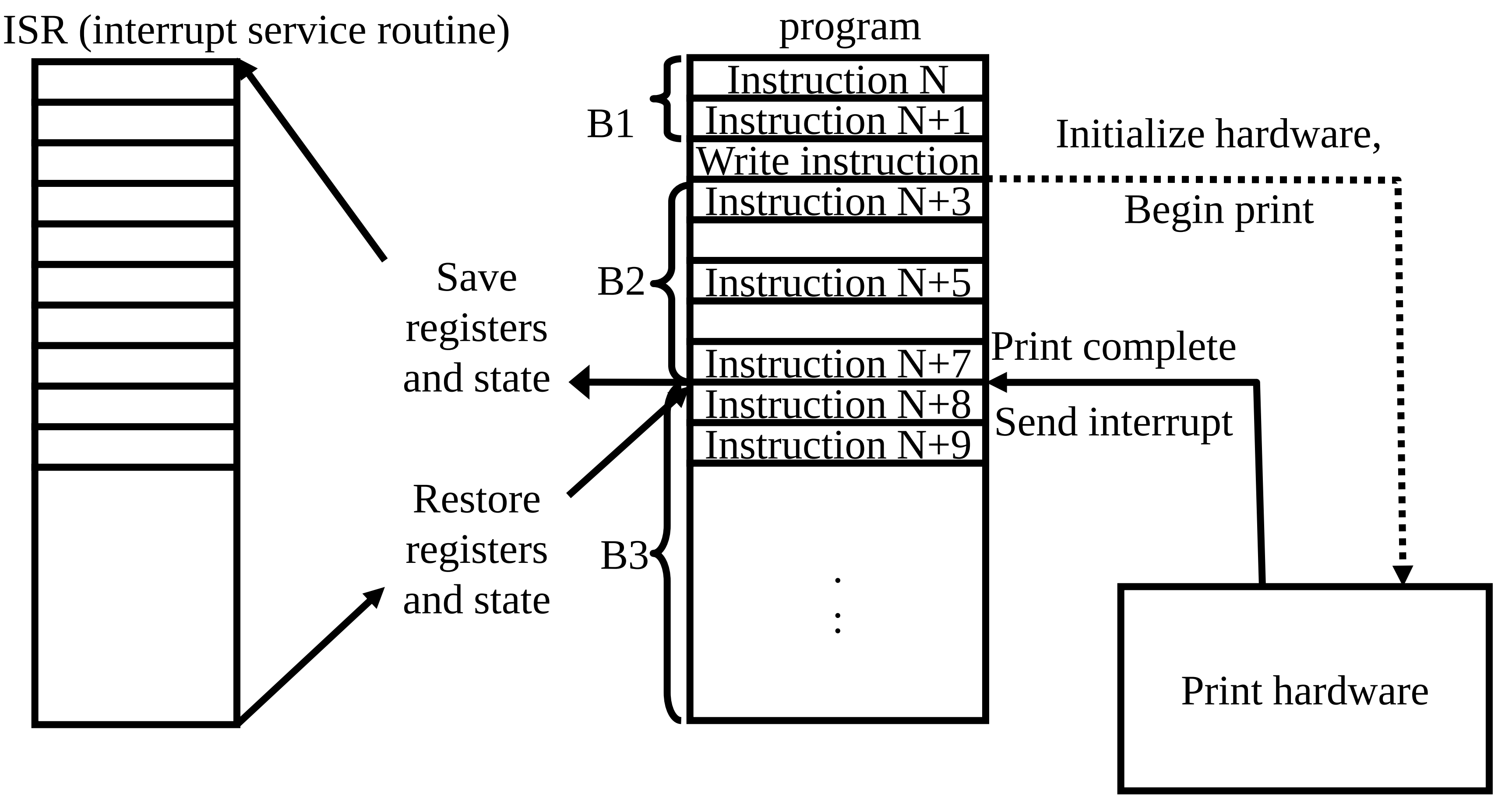
- **Spooling** refers to the process of placing data in a temporary working area for another program to process.
  - Send jobs from card reader to program queue on disk
  - Send job outputs into printer queue on disk
  - When card reader or printer are available, it can add to/print from queue
  - When a job is finished, the next job is loaded from the queue

Multiprogramming

- Partition memory into pieces (often of different sizes)
- Load one job into each partition (choose partition according to needs of job)
- Have multiple jobs executing simultaneously, one in each partition
- When one job is doing I/O, another can be using the CPU



Recall: Interrupt operation



Interrupts for input

- We have seen an example of using an interrupt to facilitate output
- What about input? Can we do the same?
- If we do the same we have problems
  - If we execute the code following the read, the value being read may be used in that code
  - If the value being read has not yet been placed in the variable, results will not be correct
- How to solve the problem?
  - Instead of executing the next block of code in the same program we let a different program run until the I/O has completed.

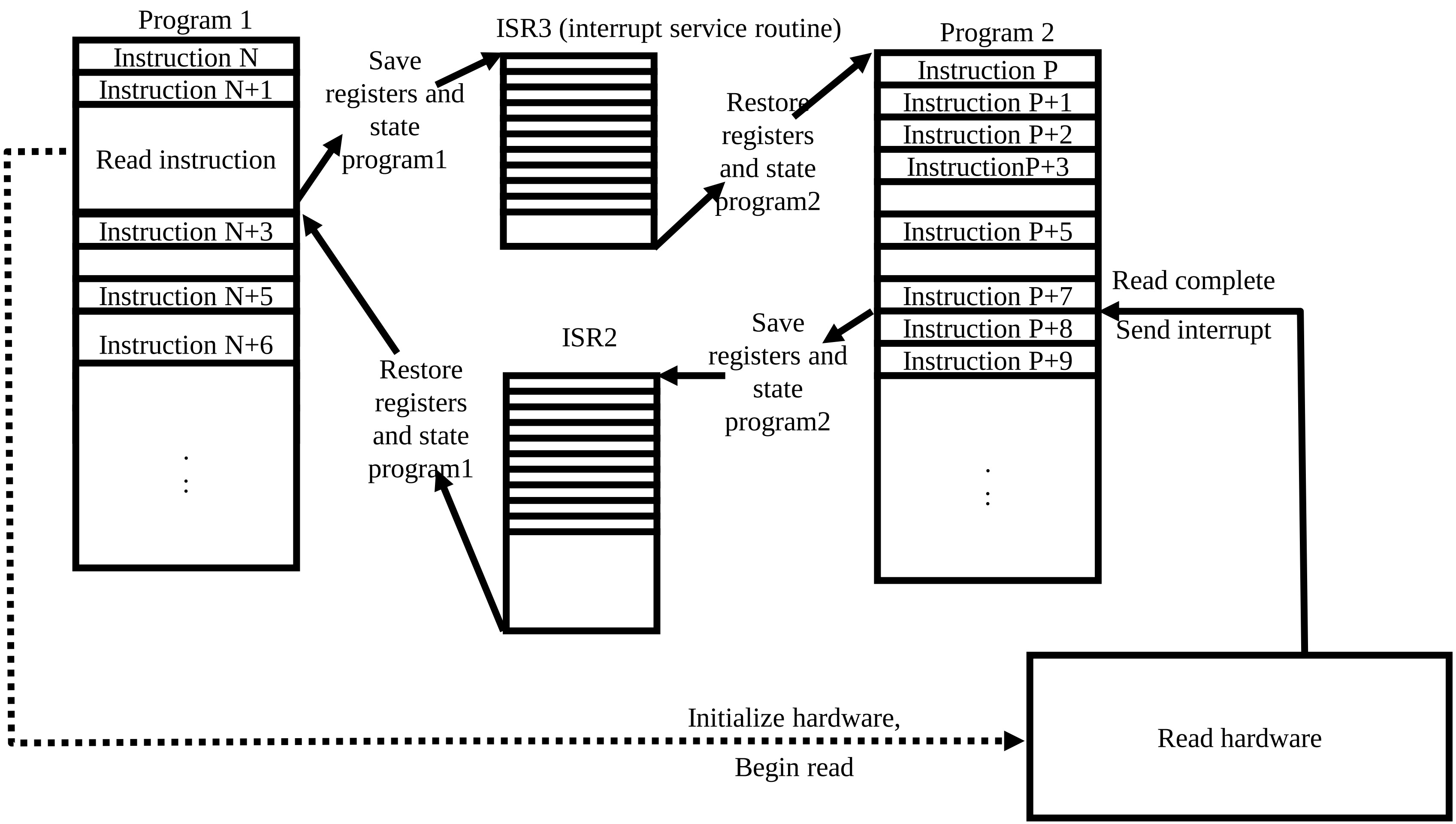
Multiprogramming: Interrupt example (1)

- Program executes until it reaches a read instruction
- The read routine sets up the input operation, then returns leaving the input hardware device (not the CPU) processing the input
- Because successive instructions in the program may use the input value, the program itself cannot continue until the read is complete.
- The read routine then interrupts program 1, saving its registers and state. Then the ISR loads the registers and state for program 2 and continues execution of its instructions (P to P+7 next slide)

Multiprogramming: Interrupt example (2)

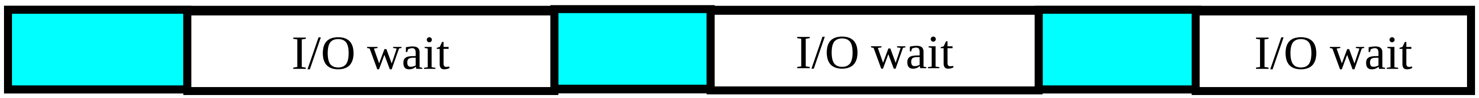
- When the input hardware device completes operation, it generates and sends an interrupt to the system (signaling normal completion)
- When the currently-executing instruction in program 2 completes, program 2 is interrupted
- The registers and state of program 2 are saved
- An interrupt service routine completes the input operation
- The registers and state of program 1 are restored
- The original program continues executing

Multiprogramming operation

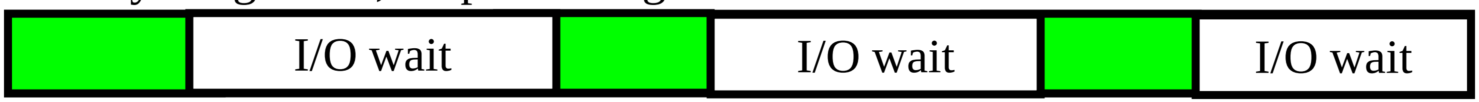


Multiprogramming example

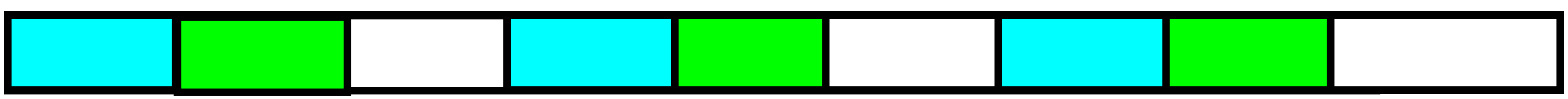
Program A, uniprocessing



Followed by Program B, uniprocessing



multiprocessing

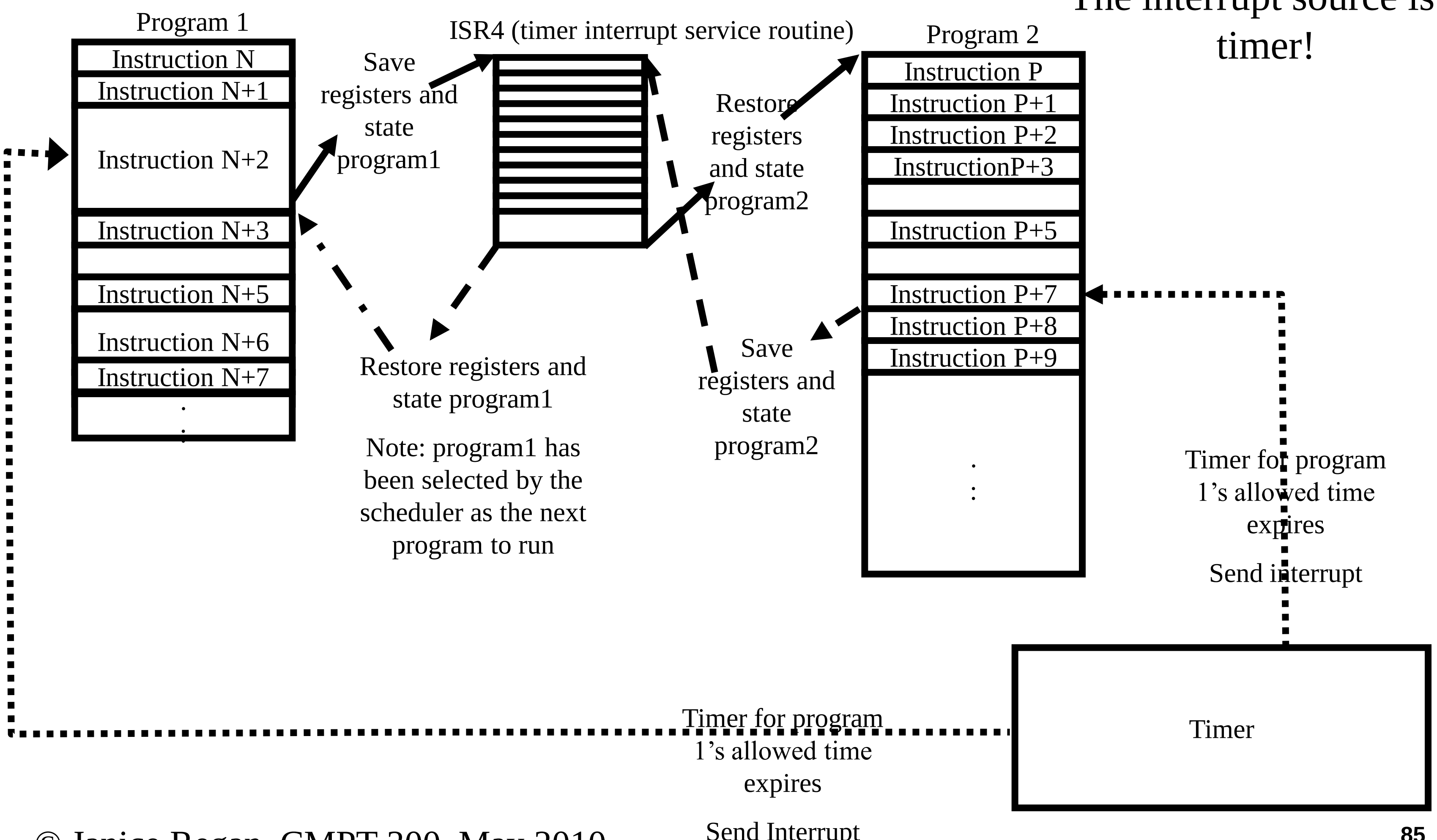


Time sharing: Scheduling, fair sharing

- Once multiple jobs can share the CPU (sequentially, not at the same time) it becomes necessary to determine how time is shared between the processes
- The simplest approach to time sharing(taken by CTSS (Compatible Time-Sharing System) and some later OS's) is time-sliced round-robin
  - Each process is given N seconds of CPU, after N seconds the next processes takes its turn
- There are many other variants of scheduling, some of which we will discuss later.

Simple time sharing operation

The interrupt source is a timer!



Time sharing and scheduling

- In the previous example the simplest case (only two programs) was considered
- In a real system there will be many programs running.
  - Each time an ISR runs, at the end of the interrupt servicing, the OS scheduler must run to determine which program to give the CPU to next, and/or what to set the timer to

Minicomputers

- Near the end of this generation (1965-1980) small, less expensive machines came into common use (for example the DEC PDP and VAX series)
- Costs were reduced from millions to 100's of thousands (about a factor of 20).
- Memory in Kbytes
- Used to develop UNIX operating system (multi-user)
- Problems as mini-computers proliferated: each vendor had their own flavor of UNIX (BSD, system 5, POSIX …) or their own proprietary OS (VMS …), compatibility was an issue
- Mainframes and supercomputers were still necessary for computationally intensive applications.

The next generation(1980- now)

- Use VLSI (very large scale integrated circuits) to build microcomputers
- Reduced price (thousands not 100’s of thousands)
- First used early operating systems like CP/M (control program for microcomputers) or DOS (disk operating system)
- First uses of user friendly GUIs

Commonly used OSs

- Versions of commonly used OSs like Windows, Unix, Linux are available for different types of platforms (PCs, Handhelds, Embedded systems)
- Specialized OSs for purposes such as real time embedded systems or large servers