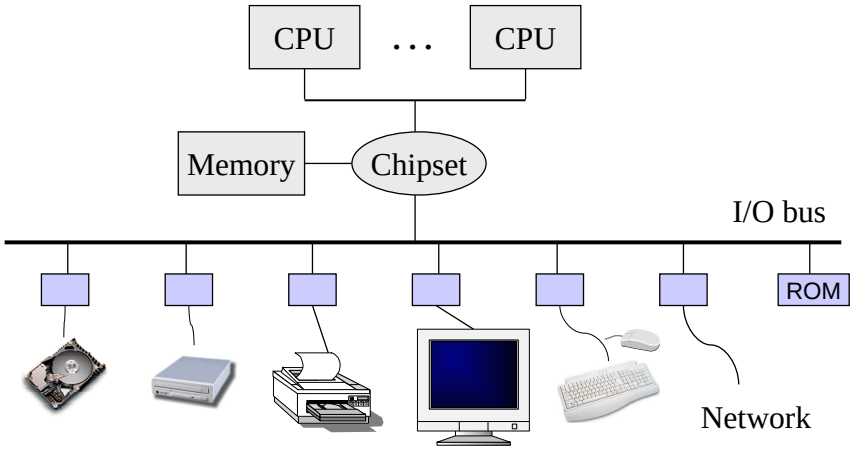
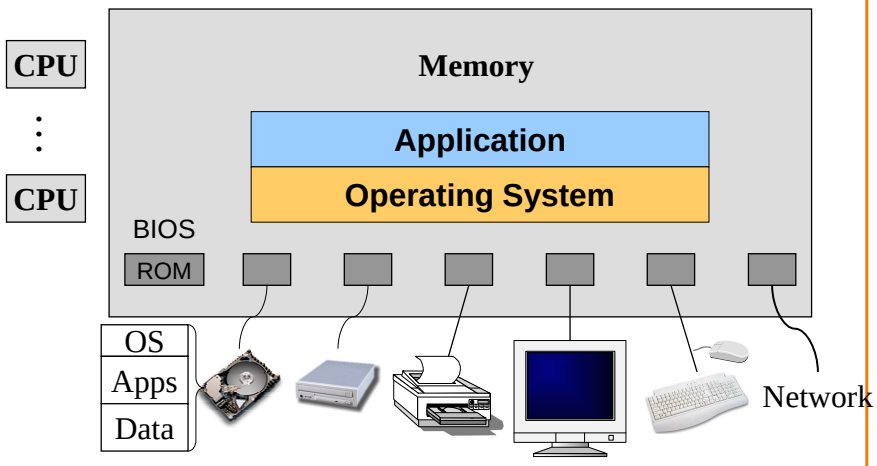
An Overview of Computer Architecture

CS 217

A Typical Computer



A Typical Computer System



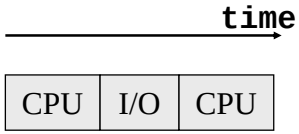
OS Service Examples

- Examples that are not provided at user level
  - System calls: file open, close, read and write
  - Control the CPU so that users won't get stuck by running
    - while (1) ;
  - Protection:
    - Keep user programs from crashing OS
    - Keep user programs from crashing each other
- Examples that can be provided at user level
  - Read time of the day

Processor Management

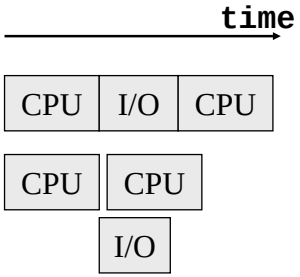
- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free

Processor Management



- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free

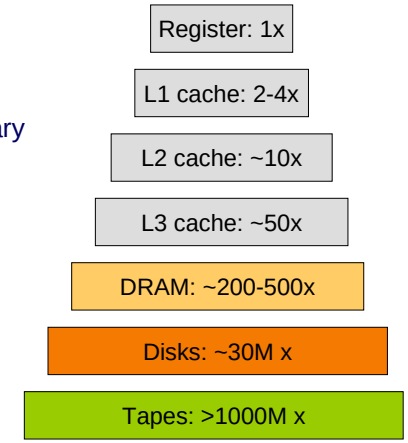
Processor Management



- Goals
  - Overlap between I/O and computation
  - Time sharing
  - Multiple CPU allocations
- Issues
  - Do not waste CPU resources
  - Synchronization and mutual exclusion
  - Fairness and deadlock free

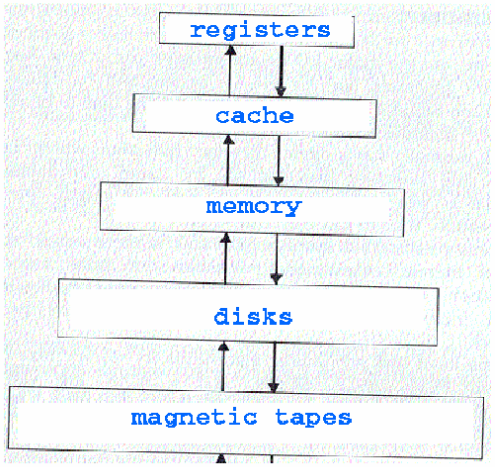
Memory Management

- Goals
  - Support programs to run
  - Allocation and management
  - Transfers from and to secondary storage
- Issues
  - Efficiency & convenience
  - Fairness
  - Protection



Memory Management

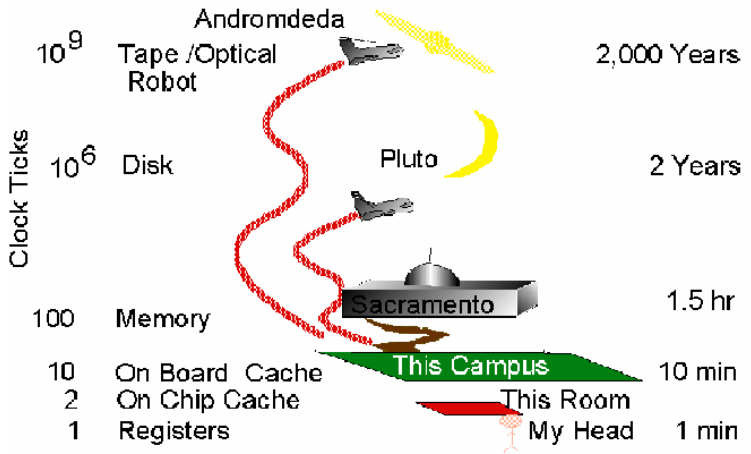
Storage Hierarchies



- Each lower level is
  - slower,
  - bigger,
  - farther away, and
  - cheaper
- Who manages what
  - registers: compiler
  - cache: hardware
  - memory: OS
  - disk: OS
- The performance of lower level is becoming increasingly important

Memory Management

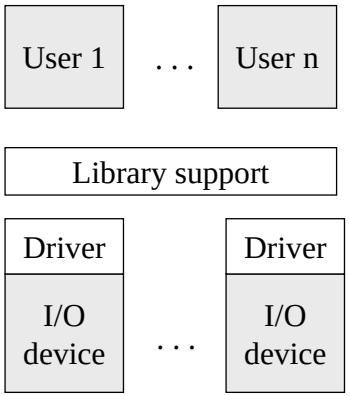
Storage Hierarchy Latency (by Jim Gray)



- And the "universe" is expanding -- farther things are getting farther faster!

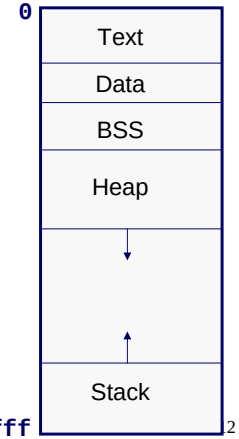
I/O Device Management

- Goals
  - Interactions between devices and applications
  - Ability to plug in new devices
- Issues
  - Efficiency
  - Fairness
  - Protection and sharing

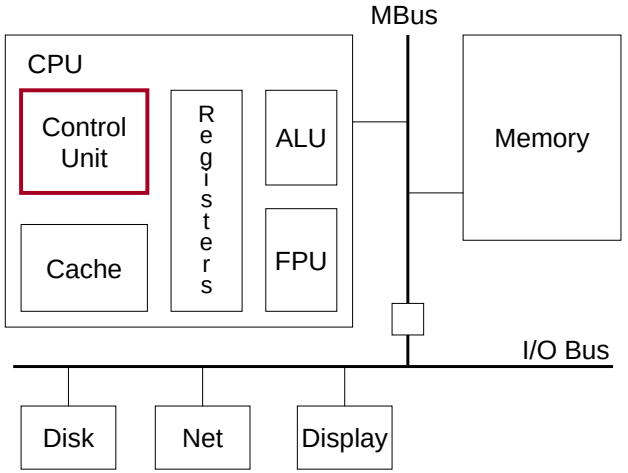


What Is An Application?

- An application has its "own" CPU, memory, and I/O
- "Own" CPU is virtual CPU
- "Own" memory is virtual memory
  - Text = code, constant data
  - Data = initialized global and static variables
  - BSS = (Block Started by Symbol) uninitialized (zero) global & static variables
  - Stack = local variables
  - Heap = dynamic memory
- "Own" I/O devices are virtual
- I/O and CPU may overlap



General Computer Architecture



General Instruction Execution

- CPU's control unit executes a program

```
PC ← memory location of first instruction
while (PC != last_instr_addr) {
    i = fetch(MEM[PC++]);
    execute(i);
}
```

- Multiple phases…
  - Fetch: instruction fetch; increment PC
  - Execute: arithmetic instructions, compute branch target address, compute memory addresses
  - Memory access: read/write memory
  - Store: write results to registers

Fetch	Execute	Memory	Store	Fetch	Execute	Memory	Store

Concept of Instruction Pipelining

- A simple pipeline

Fetch	Execute	Memory	Store				
	Fetch	Execute	Memory	Store			
		Fetch	Execute	Memory	Store		

- What about branch instruction?
- Modern CPUs usually have deep pipelines
  - Pentium II has a 10-stage pipeline
  - Pentium 4 has a 20-stage pipeline
  - They all have sophisticated branch prediction mechanisms

Instructions

- High-level language

```
x = a + b;
```

- Assembly language

```
movl  12(%ebp), %eax
addl  8(%ebp), %eax
```

Symbolic Representation

- Machine code

```
00000011000011000100010 1
11001001000010000100010 1
```

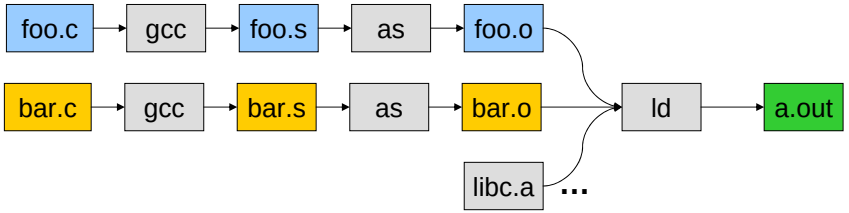
Bit-encoded Representation

Machine Code

- IA32 has variable-sized instructions
- Example:

```
push   %ebp              0x8B
mov    %esp,%ebp         0xE589
```

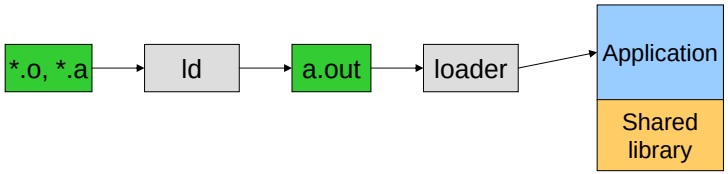

Pipeline of Creating An Executable File



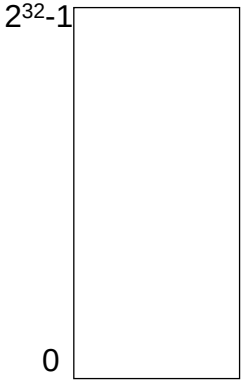
- gcc can compile, assemble, and link together
- Compiler part of gcc compiles a program into assembly
- Assembler compiles assembly code into relocatable object file
- Linker links object files into an executable

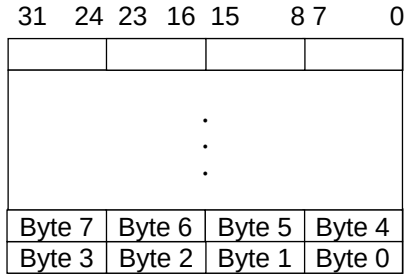
Execution (Run An Application)

- On Unix, “loader” does the job
  - Read an executable file
  - Layout the code, data, heap and stack
  - Dynamically link to shared libraries
  - Prepare for the OS kernel to run the application



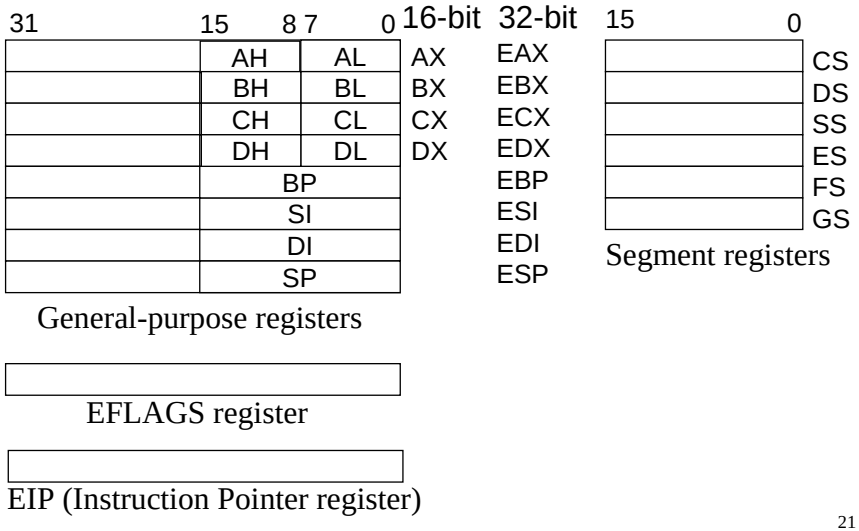
IA32 Memory





Byte order is little endian

IA32 Architecture Registers

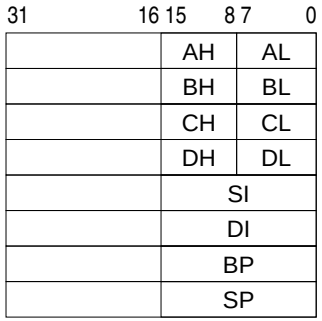


Upcoming Lectures ...

- Mode, registers and addressing
- Arithmetic and logic Instructions
- Control transfer instructions
- Assembly directives
- Assembler

Revisit IA32 General Registers

- 8 32-bit general-purpose registers (e.g. EAX)
- Each lower-half can be addressed as a 16-bit register (e.g. AX)
- Each 16-bit register can be addressed as two 8-bit registers (e.g AH and HL)



AX EAX: Accumulator for operands, results
BX EBX: Pointer to data in the DS segment.
CX ECX: Counter for string, loop operations.
DX EDX: I/O pointer.
ESI: Pointer to DS data, string source
EDI: Pointer to ES data, string destination
EBP: Pointer to data on the stack
ESP: Stack pointer (in the SS segment)

EIP Register

- Instruction Pointer or “Program Counter”
- Software change it by using
  - Unconditional jump
  - Conditional jump
  - Procedure call
  - Return

Segment Registers

- IA32 memory is divided into segments, pointed by segment registers
- Modern operating system and applications use the (unsegmented) memory mode: all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.