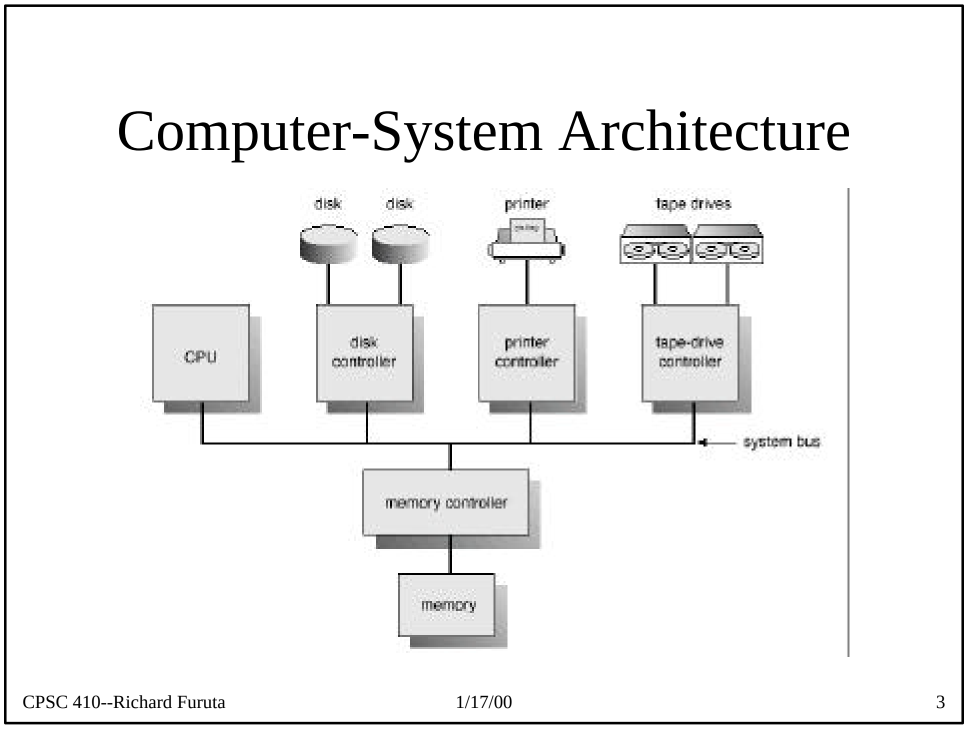
Silberschatz, et al.

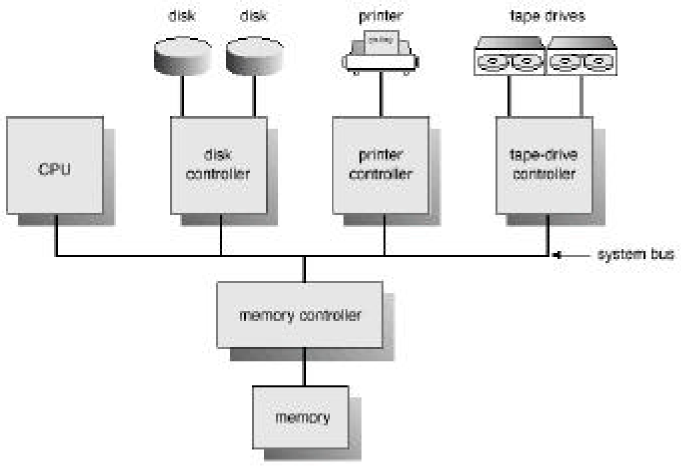
Topics based on Chapter 2
Computer-System Structures

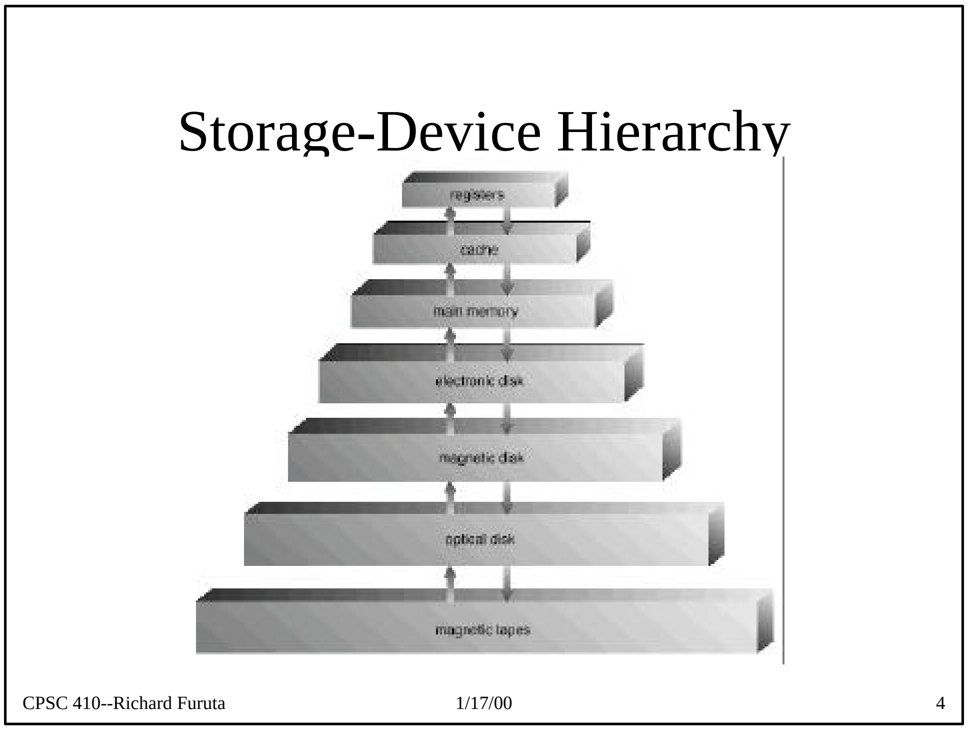
Topics

- CPU/Device interface
- I/O structure
- Storage structure and hierarchy
- Hardware protection

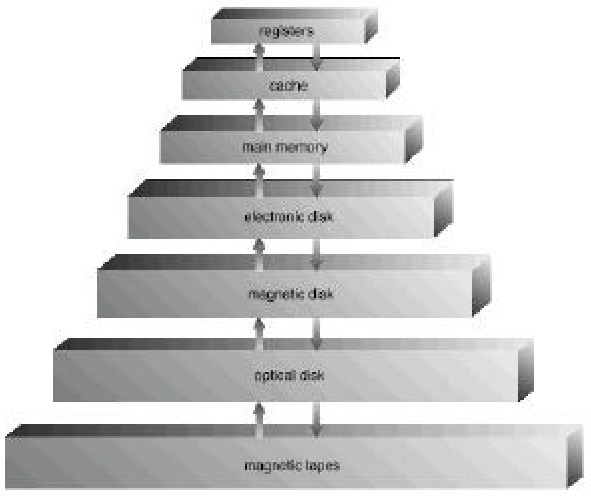


Computer-System Architecture





Storage-Device Hierarchy



CPU/Device Interface

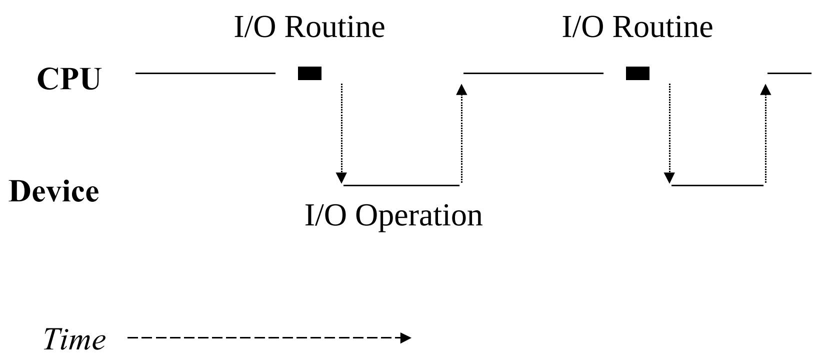
- Assume Von Neumann architecture
- Assume each device has a small hardware buffer (one record)
- Handshaking
  - CPU responsible for data transfer
  - Device raises flag when consistent state (Q: what is *consistent*?)
  - CPU transfers information and notifies device
- CPU control?

CPU Control
Busy Waiting

```
repeat {
   while (device busy) {} ;
   transfer record;
   notify device;
} until (transfer finished)
```

What are the pros and cons?

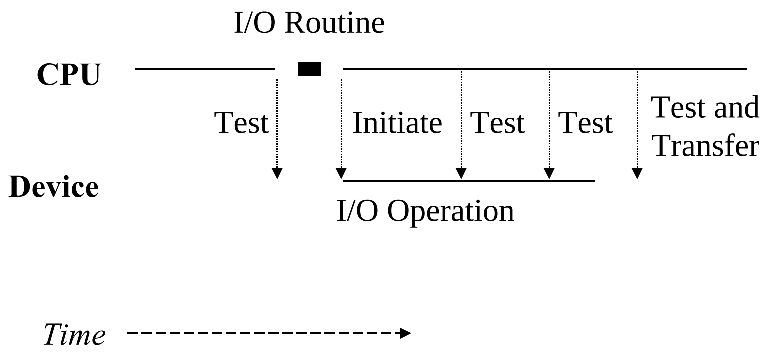
Busy Waiting
Time line



CPU Control
Polling

- CPU loads appropriate registers and initiates operation
- CPU switches to a different activity
- From time to time CPU queries hardware flag to see if device is finished
  - if so, CPU performs transfer
  - if not, CPU returns to other task
- Pros and cons?

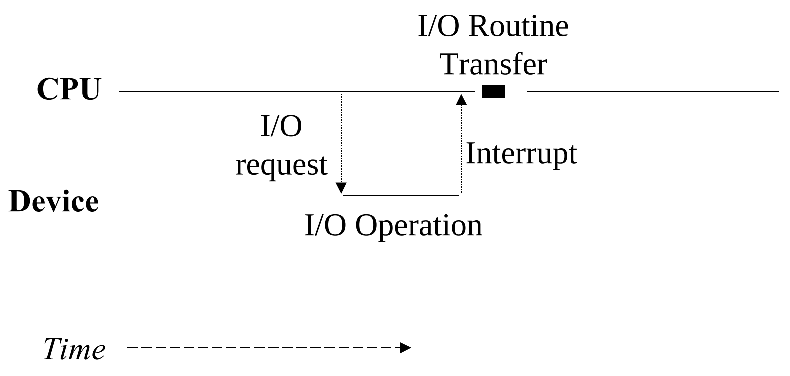
Polling
Time line



Asynchronous Control
Interrupts

- CPU loads appropriate registers and initiates operation
- CPU carries on with another task
- When device completes operation, it informs CPU with interrupt
- CPU stops what it is doing and transfers control to interrupt handler (located at predefined location in memory)
- Pros and cons?

Interrupts Time line



Use of interrupts (AKA traps)

- External events (i/o, timers)
- Internal events:
  - system calls
  - errors (illegal instruction, addressing violation, operand out of range, etc.)
  - page faults
  - ...

Implementation of Interrupts

- Multiple classes of interrupts (perhaps associated with different events)
- **Interrupt handler**: small software routine that determines what caused interrupt and what to do (AKA interrupt service routine)
- Reserved set of locations in low memory that are indices to interrupt handlers (vectors)

Interrupt Implementation

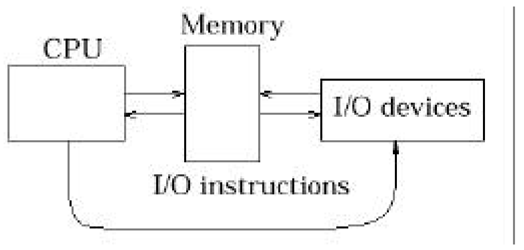
Hardware instruction cycle:

```
while (true)  {
    fetch next instruction
    increment instruction counter
    carry out instruction
    if interrupt pending then  {
        store instruction counter, save state as necessary
        jump to handler (by setting IC from vector)
    }
}
```

Interrupt Implementation

- Where is return address stored?
- How many interrupts can be active at a given time? What if more than one is active?
- What if interrupts arrive too fast? How can this be prevented?

Direct Memory Access (DMA)



Direct Memory Access (DMA)

- High speed I/O devices
- Device transfers block of data to memory directly with no intervention by CPU
  - 128 to 4096 bytes common
- Device notifies CPU that data has been transfered (how?)
- Extension: channels
  - special purpose processors that also offload CPU work by, e.g., handling device errors, code conversion, formatting functions during DMA activity

Other Interrupt Examples

- Timers
- User-generated signals (SIGHUP, SIGQUIT, SIGKILL)
- Dual-mode instructions
  - User mode and monitor mode (AKA system/supervisor)
- Hardware protection implementation
  - Base/limit registers, set in monitor mode (why?)
  - hardware insures address references in legal range and traps if not

CPU Protection

- Timer--interrupts computer after specified period to ensure operating system maintains control
  - Timer decremented every clock tick
  - When it reaches the value 0, an interrupt occurs
- Implements time sharing
- Implements current time clock

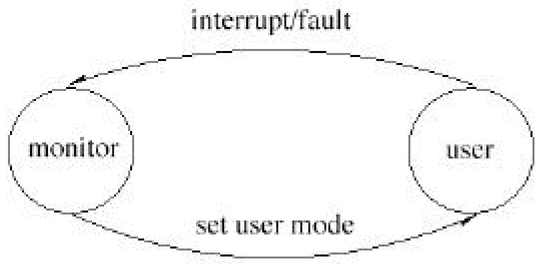
Hardware Protection

- Dual-mode operation
- I/O protection
- Memory protection
- CPU protection (just discussed)

Dual-Mode Operation

- Hardware support differentiates between
  - User mode -- execution done on behalf of a user
  - Monitor mode (aka supervisor mode or system mode) -- execution done on behalf of operating system
- Mode bit shows current mode
- Switches on interrupt or fault
- Privileged instructions only in monitor mode

Dual-Mode Operation



I/O Protection

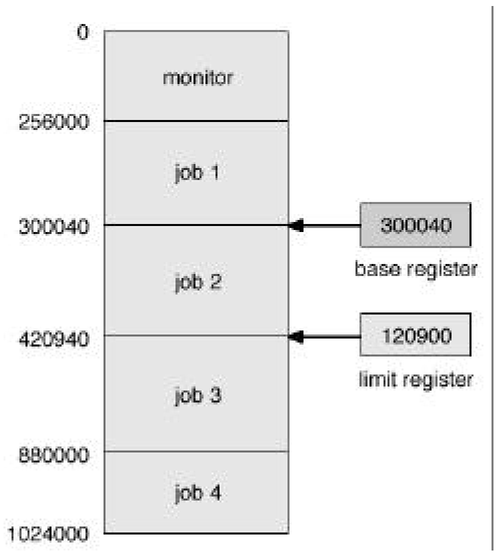
- User program can cause problems with I/O
  - Issuing illegal I/O instructions
  - Accessing memory locations that aren’t under its control
  - Refusing to relinquish the CPU
- All I/O instructions are privileged instructions
- User programs carry out I/O through the operating system

I/O Operations

- Process requests I/O operation through a system call
  - Trap to specific location in the interupt vector
  - Control passes through the interrupt vector to service routine in OS; enters monitor mode
  - Monitor verifies correctness of parameters, executes the request, exits monitor mode, returns control to the user program

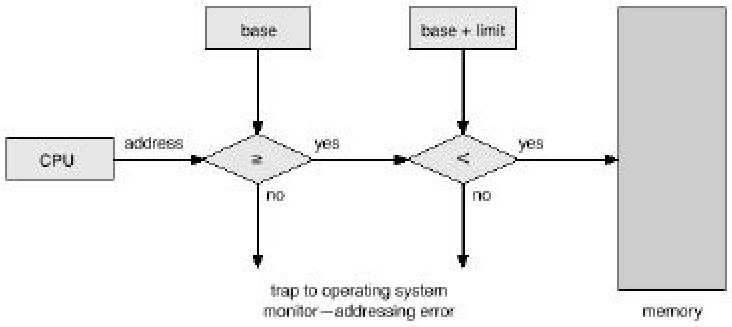
Memory protection

- Must protect interrupt vector and interrupt service routines
- Desirable to protect other users' memory spaces
- Two registers determine the range of legal addresses a program may access
  - Base register (smallest legal physical memory address)
  - Limit register (size of range)



Memory protection

- Accomplished by protection hardware (disabled in monitor mode)



General System Architecture

- Dual modes of execution
  - user mode
  - monitor mode
    - privileged instructions that can only be executed in monitor mode (e.g., halt, turn on/off interrupts, change from user mode to monitor mode, I/O operations)

General System Architecture

- User programs must ask the monitor to perform priviledged operations (e.g., I/O)
  - Request via **system call** (also known as **monitor call**, **operating-system function call**)
  - Invocation varies with architecture
    - usually by a trap to a specific location in the interrupt vector
    - syscall instruction

General System Architecture

- System call
  - treated as software interrupt
  - control passes to service routine (part of the OS) and mode bit set to monitor mode
  - if legal, the operation is executed and result placed in registers, stack, or in memory
  - control then returned to user program