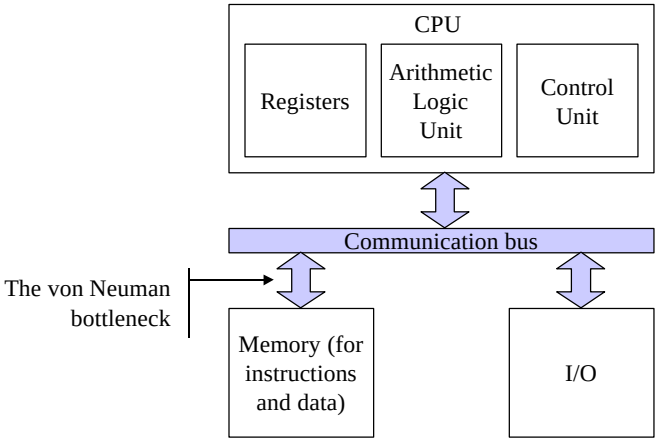
Lecture Overview

- Computer hardware introduction
  - Structure of a simple computer
  - Introduction to interrupts
  - Input/Output (I/O) structure
  - Storage structure
  - Hardware protection
  - Initial introduction to system calls

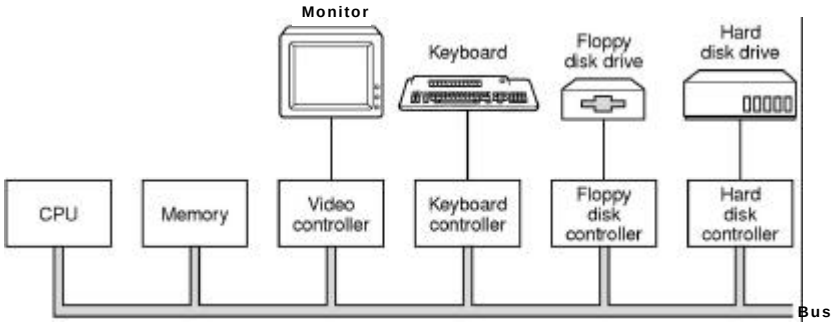
Structure of a Simple Computer

The abstract *von Neumann architecture* is used in most computers...



An alternative is the *Harvard architecture* that uses separate buses and memory for program instructions and data...

Structure of a Simple Computer



An example of the von Neuman architecture in a simple computer; all components are connect together by the "system bus," which is a communication channel that allows the CPU and other components to talk to each other and to access memory

Structure of a Simple Computer

- Devices and the CPU can execute concurrently
- Devices are not directly connected to the system bus, instead a device controller is connected
  - Each controller is in charge of a particular device type
  - Each controller has a local buffer and special purpose registers
  - Special software interfaces to the controller, *device driver*
- CPU moves data between main memory & local buffers
- I/O is from the device to local buffer of controller
- Device controller informs CPU that it has finished its operation by causing an *interrupt*

What is an Interrupt?

- An event that transfers control to an interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines
- A *trap* is a software-generated interrupt caused either by an error or a user request
- Modern operating systems are interrupt-driven

Consider Simple CPU Usage

- Initially, the OS is the only program running (started via a bootstrap program)
- A typical uniprocessor computer can only run one program at a time
  - We want the operating system to share the CPU with any other programs that the user wants to run
- In a single-tasking system, this is not so difficult, essentially make a procedure call (i.e., main()) and wait for it to return (it might not)
- How does the OS share the CPU in a multitasking system and prevent runaway programs?
  - The OS sets a hardware timer interrupt

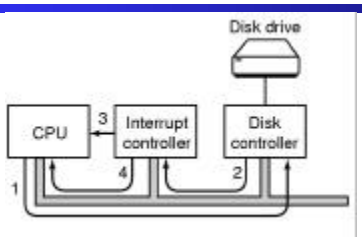
Interrupt Handling

- The interrupt handler preserves the state of the CPU by storing registers
  - The address of the interrupted instruction must be saved so that interrupt handler can return to it
- The handler determines the type of interrupt that occurred
  - There are a fixed number of interrupts for a CPU associated with specific devices
- Separate segments of code determine what action should be taken for each type of interrupt
  - The OS stores interrupt handling routine addresses in an interrupt vector generally indexed by device number
- Generally, interrupts are disabled while an interrupt is being processed to prevent lost interrupts

The I/O Structure

- I/O devices are accessed via their device driver for their controller
  - A device controller may have more than one device (e.g., a SCSI controller)
  - Controllers have special registers and a local buffer
  - Controllers are responsible for moving data between devices and the local buffer
- Controllers and the CPU coordinate with interrupts

Performing I/O

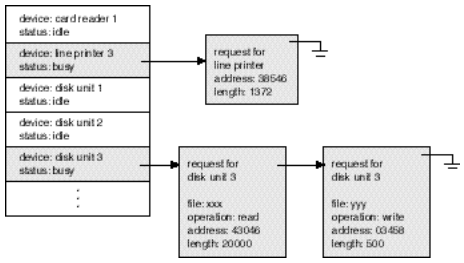


- To start an I/O operation, the CPU loads the appropriate instructions and values into the registers of the device controller via the device driver
- The device controller examines the registers
  - The request may be a read or write instruction
  - The controller performs the appropriate actions
- Once finished, the controller triggers an interrupt
- The interrupt handler services the interrupt once it occurs

I/O Handling Alternatives

- The CPU can start an I/O request and then wait for it to complete; this is *synchronous* I/O
  - In this case the user program always blocks when performing any I/O operation
  - The main advantage of synchronous I/O is its simplicity
- Or the CPU can start an I/O request and then do other work until the I/O completes; this is *asynchronous* I/O
  - In this case the user program may or may not block depending on how it sent the I/O request to the OS
  - The main advantage of asynchronous I/O is the increased system efficiency
  - This is the typical way I/O is handled in an OS

Asynchronous I/O Handling



- The OS maintains a *device-status table*
  - Keeps track of many I/O requests at the same time
  - Contains an entry for each I/O device; indicates the device's type, address, state, and request queue
- I/O requests are queued if the device is busy

I/O Interrupt Servicing

- When the CPU receives an I/O interrupt, it (generally) stops whatever it is currently doing and services the interrupt immediately
  - The I/O interrupt handler is called via the interrupt vector
  - CPU state is saved, including the return instruction address
  - The OS determines which device caused the interrupt
    - The OS looks in the device-status table and modifies the device entry to update its states
    - If there are more I/O requests for the device, the OS starts the next one
    - If there was a process waiting for the I/O, it can be scheduled
  - When the interrupt handler is finished, control is returned to the address of the instruction when the interrupt occurred

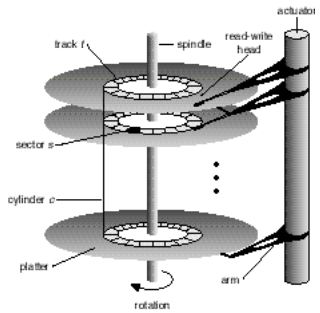
Direct Memory Access (DMA) I/O

- Consider a slow I/O device, like terminal input
  - A character may arrive every 1000 microseconds, perhaps
  - An interrupt service/handler requires 2 microseconds
  - This leaves 998 microseconds out of 1000 to do work
  - A high speed device could seriously eat into CPU time
- DMA is required for high-speed devices
  - The CPU sets up buffers, pointers, and counters for the I/O device
  - The device controller transfers blocks of data from buffer storage directly into main memory without CPU intervention
  - Only one interrupt is generated per block, rather than the one interrupt per byte or word
  - Device contends with CPU for access to memory on the bus

Storage Structure

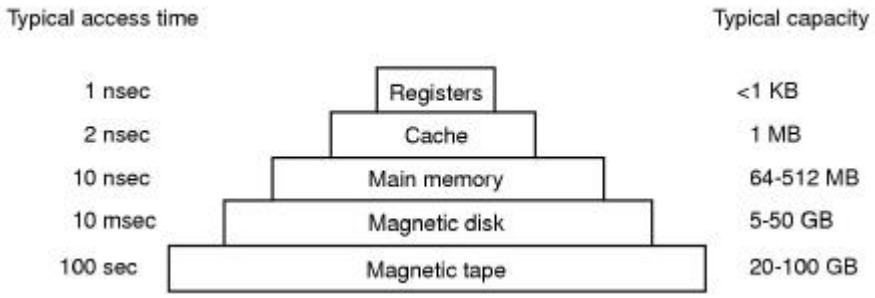
- Main memory is the only large storage media that the CPU can access directly
- Secondary storage is an extension of main memory that provides large nonvolatile storage capacity
  - Magnetic disks are rigid metal or glass platters covered with magnetic recording material
    - Disk surface is logically divided into tracks, which are subdivided into sectors
    - The disk controller determines the logical interaction between the device and the computer

Hard Disk Mechanism



- A typical hard disk is made up of multiple magnetic platters on which digital information is recorded
- Read/write heads sit above each platter surface on a movable arm
- A track is a ring on one platter and is divided into sectors, the stack of homologous tracks for a cylinder
- Issues of transfer rate, seek time, and rotational latency are applicable

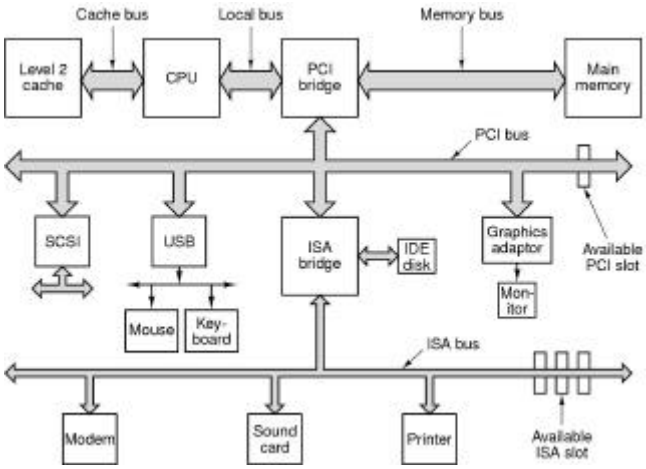
Storage Hierarchy



- The top of the storage hierarchy is very fast, but very expensive
- The bottom of the storage hierarchy is very inexpensive, but very slow
- The types of storage also differ because the upper layers are volatile (not persistent) where the lower layers are nonvolatile (persistent)
- In general, each lower layer of the hierarchical is used as a cache for the layer above it

Structure of a Typical Computer

The following diagram depicts a more complex computer system

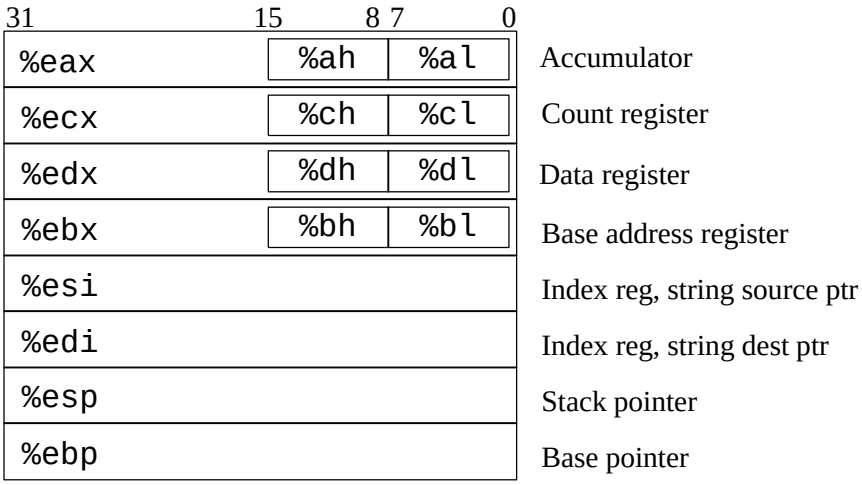


A Typical Intel Pentium Processor

- An x86 processor has multiple data types
  - byte = 8-bit data type (e.g., char)
  - word = 16-bit data type (e.g., short)
  - double word = 32-bit data type (e.g., int, long, pointer, float)
  - quad word = 64-bit data type (e.g., double)
- An x86 processor has various registers
  - Essentially 8 integer registers, 6 of them are general purpose and 2 are special purpose; some instructions also use specific registers
  - A condition code register
  - 8 floating point
- The x86 instruction set has various instructions that operate on the different data types and with some combination of registers, literals, and memory

Intel Pentium Registers

Pentium integer registers



Intel x86 Assembler Example

Consider converting the following trivial C code to assembler

```
int main(int argc, char *argv)
{
    int i = 12;
    int j = 2;
    if ((i % j) == 0)
        i = i / j;
    else
        i = j;
}
```

Intel x86 Assembler Example

```
main:
        pushl %ebp          # Save base pointer on stack.
        movl %esp,%ebp      # Use stack pointer as our new
                            # base pointer.
        subl $24,%esp       # Allocate some space on the
                            # stack for our variables.
        movl $12,-4(%ebp)   # Initialize variable i.
        movl $2,-8(%ebp)    # Initialize variable j.
        movl -4(%ebp),%ecx  # Retrieve value of i.
        movl %ecx,%eax      # Put value of i into %eax;
                            # the division command 64 bit
                            # %edx is high-order bytes and
                            # %eax is low-order bytes.
        cltd                # This sign extends %eax
                            # into %edx.
        idivl -8(%ebp)      # Divide the value of i in %edx
                            # and %eax by j.
# continued on next slide
```

Intel x86 Assembler Example

```
                            # The divide command puts the
                            # remainder in %edx and the
                            # quotient into %eax.
        testl %edx,%edx     # Bit-wise AND on remainder.
        jne .L3             # Jump if testl result not zero.
        movl -4(%ebp),%ecx  # This is the 'if' portion of
        movl %ecx,%eax      # our program; we are simply
        cltd                # doing the division again
        idivl -8(%ebp)      # and storing the quotient into
        movl %eax,-4(%ebp)  # the variable i.
        jmp .L4             # Jump to the end.
.L3:
        movl -8(%ebp),%eax  # This is th 'else' portion.
        movl %eax,-4(%ebp)  # Put j into i, cannot do memory
                            # to memory on Intel so we use a
                            # register.
.L4:
        leave
        ret
```

Hardware Protection

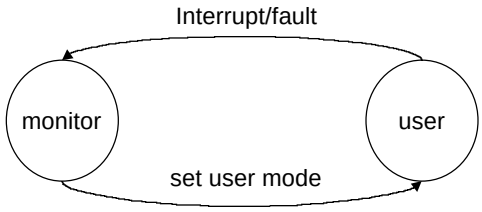
- Early OSs dealt with one program at a time and were not largely concerned with protection
- As OS sophistication increased so did the need to protect program from one another
  - Dual-Mode Operation
  - I/O Protection
  - Memory Protection
  - CPU Protection

Dual-Mode Operation

- The OS must ensure that an incorrect program cannot cause other programs to execute incorrectly
- The main approach to enable protection is to provide hardware support to differentiate between at least two modes of operations
  1. User mode – execution done on behalf of a user
  2. Monitor mode (also supervisor mode or system mode) – execution done on behalf of operating system

Dual-Mode Operation

- A mode bit was added to the CPU to indicate the current mode of operation: monitor (0) or user (1)
- When an interrupt or fault occurs hardware switches to monitor mode



- Certain CPU instructions are defines as “privileged”
- Privileged instructions execute in monitor mode only

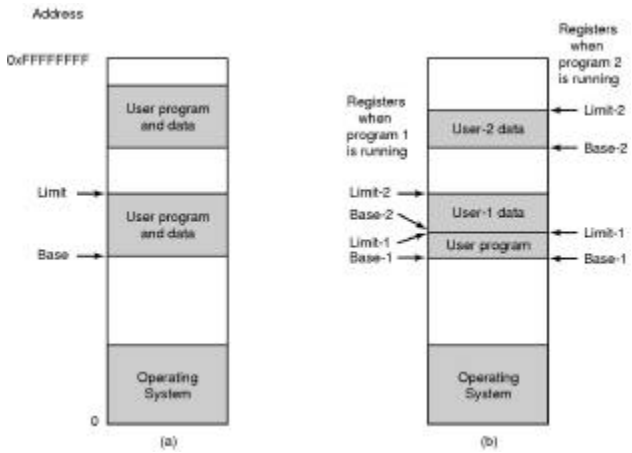
I/O Protection

- I/O instructions may disrupt normal computer operations and must be protected
- As a result, all I/O instructions are privileged
- Must ensure that a user program could never gain control of the computer in monitor mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector)

Memory Protection

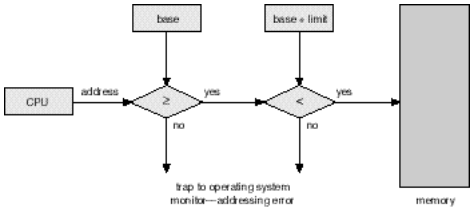
- Must provide memory protection at least for the interrupt vector and the interrupt service routines
- In order to have memory protection, two registers are added to the CPU that determine the range of legal addresses a program may access
  - The *base register* holds the smallest legal physical memory address
  - The *limit register* contains the size of the memory range
- Memory outside the defined range is protected

Memory Protection



The left side shows a single base/limit register set, it is also possible to have two sets of registers for the program text (instructions) and the program data; this allows sharing of program text

Memory Protection



- When executing in monitor mode, the operating system has unrestricted access to both monitor and user's memory
- The load instructions for the *base* and *limit* registers are privileged instructions

CPU Protection

- *Timer* – interrupts computer after specified period to ensure operating system maintains control
  - Timer is decremented every clock tick
  - When timer reaches the value 0, an interrupt occurs
- Timer commonly used to implement time sharing
- Time also used to compute the current time
- Load-timer is a privileged instruction

System Calls

- Given the I/O instructions are privileged, how does the user program perform I/O?
- System call is the method used by a process to request action by the operating system
  - Usually takes the form of a trap to a specific location in the interrupt vector
  - Control passes through the interrupt vector to a service routine in the OS, and the mode bit is set to monitor mode
  - The monitor verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call
- More on these in next lecture...