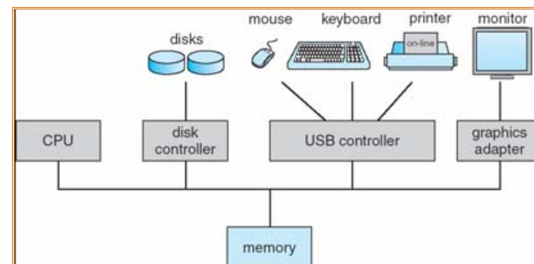
Architectural Support for Operating Systems

Announcements

This Lectures

- I/O subsystem and device drivers
- Interrupts and traps
- Protection, system calls and operating mode
- OS structure
- What happens when you boot a computer?

Computer System Architecture



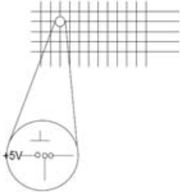
I/O operations

- I/O devices and the CPU can execute concurrently.
- I/O is moving data between device & controller's buffer
  - CPU moves data between controller's buffer & main memory
- Each device controller is in charge of certain device type.
  - May be more than one device per controller
    - SCSI can manage up to 7 devices
  - Each device controller has local buffer, special registers
- A device driver for every device controller
  - Knows details of the controller
  - Presents a uniform interface to the rest of OS

Accessing I/O Devices

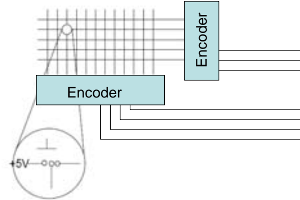
- Memory Mapped I/O
  - I/O devices appear as regular memory to CPU
  - Regular loads/stores used for accessing device
  - This is more commonly used
- Programmed I/O
  - Also called I/O mapped I/O
  - CPU has separate bus for I/O devices
  - Special instructions are required
- Which is better?

Building a Keyboard Controller



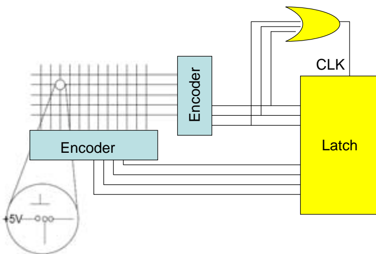
Mesh detects which key is pressed

Building a Keyboard Controller



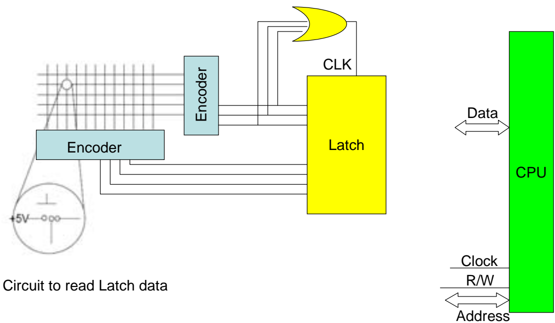
Encoders convert lines to binary

Building a Keyboard Controller



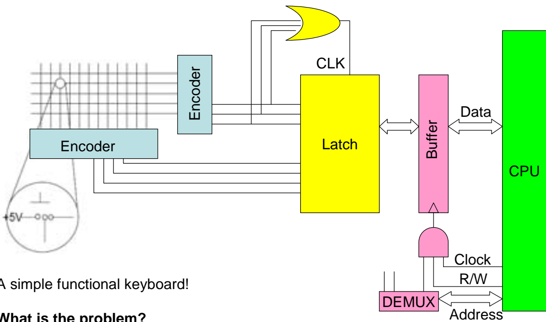
Latch stores encoding of pressed key

Building a Keyboard Controller



Circuit to read Latch data

Building a Keyboard Controller

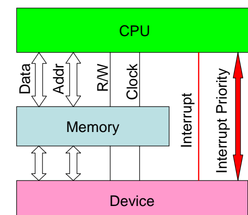


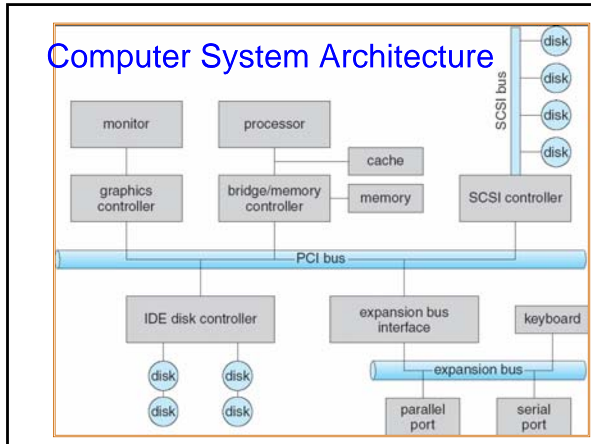
A simple functional keyboard!

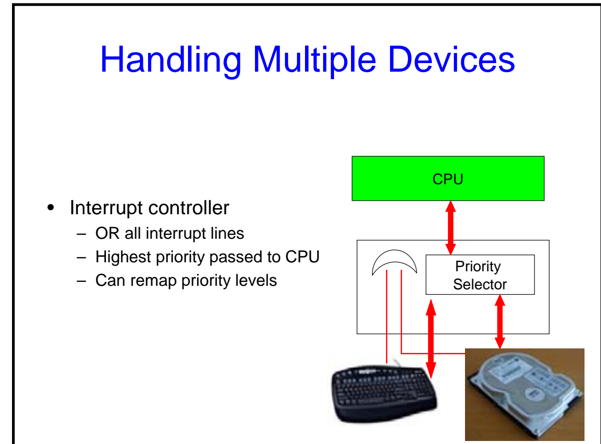
What is the problem?

Interrupts

- Mechanism required for device to *interrupt* the CPU
  - Alternatively, CPU could poll. But this can be inefficient
- Implementing interrupts
  - A line to interrupt CPU
  - Set of lines to specify priority







How does I/O work?

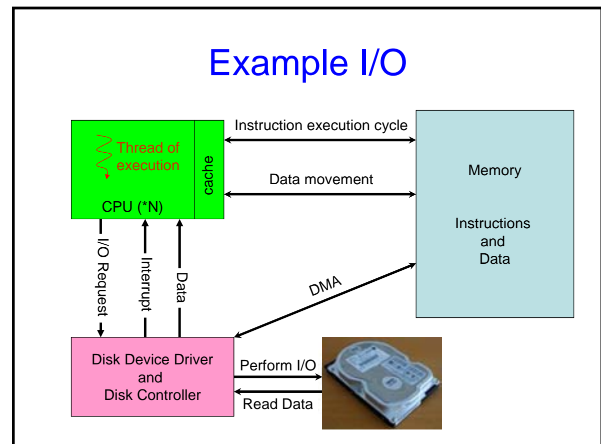
- Device driver loads controller registers appropriately
- Controller examines registers, executes I/O
- Controller signals I/O completion to device driver
  - Using interrupts
- High overhead for moving bulk data (i.e. disk I/O)

Do Examples

- Windows device drivers:
  - Network and PCI Bus
    - IRQ, I/O Range, and Memory Range
  - System Timer, Numeric Processor
    - IRQ and I/O Range only
  - Speaker
    - I/O Range only
  - Disk and Keyboard
    - Nothing! (But they have other ways to interrupt.)
  - DMA is special

Direct Memory Access (DMA)

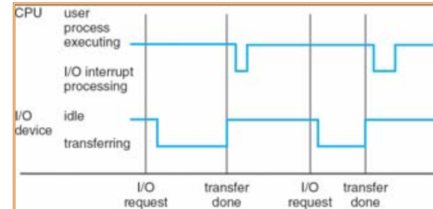
- Transfer data directly between device and memory
  - No CPU intervention
- Device controller transfers blocks of data
- Interrupts when block transfer completed
  - As compared to when byte is completed
- Very useful for high-speed I/O devices



Interrupts

- Notification from interface that device needs servicing
  - Hardware: sends trigger on bus
  - Software: uses a **system call**
- Steps followed on receiving an interrupt:
  - Stop kernel execution
  - Save machine context at interrupted instruction
  - Commonly, incoming interrupts are disabled
  - Transfer execution to Interrupt Service Routine (ISR)
    - Mapping done using the Interrupt Vector (faster)
  - After ISR, restore kernel state and resume execution
- Most operating systems are interrupt-driven

Interrupt Timeline



Traps and Exceptions

- Software generated interrupt
  - Exception: user program acts silly
    - Caused by an error (div by 0, or memory access violation)
    - Just a performance optimization
  - Trap: user program requires OS service
    - Caused by system calls
- Handled similar to hardware interrupts:
  - Stops executing the process
  - Calls handler subroutine
  - Restores state after servicing the trap

Why Protection?

- Application programs could:
  - Start scribbling into memory
  - Get into infinite loops
- Other users could be:
  - Gluttonous
  - Evil
  - Or just too numerous
- Correct operation of system should be guaranteed

  ⇒ A protection mechanism

Preventing Runaway Programs

- Also how to prevent against infinite loops
  - Set a timer to generate an interrupt in a given time
  - Before transferring to user, OS loads timer with time to interrupt
  - Operating system decrements counter until it reaches 0
  - The program is then interrupted and OS regains control.
- Ensures OS gets control of CPU
  - When erroneous programs get into infinite loop
  - Programs purposely continue to execute past time limit
- Setting this timer value is a privileged operation
  - Can only be done by OS

Protecting Memory

- Protect program from accessing other program's data
- Protect the OS from user programs
- Simplest scheme is base and limit registers:

- Virtual memory and segmentation are similar

Protected Instructions

Also called privileged instructions. Some examples:

- Direct user access to some hardware resources
  - Direct access to I/O devices like disks, printers, etc.
- Instructions that manipulate memory management state
  - page table pointers, TLB load, etc.
- Setting of special mode bits
- Halt instruction

Needed for:

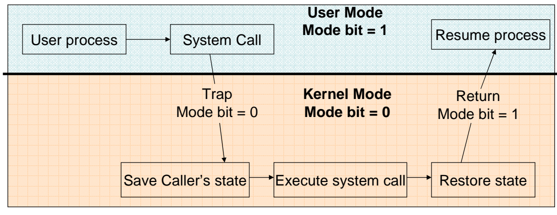
- Abstraction/ease of use and protection

Dual-Mode Operation

- Allows OS to protect itself and other system components
  - *User* mode and *kernel* mode
- OS runs in kernel mode, user programs in user mode
  - OS is god, the applications are peasants
  - Privileged instructions only executable in kernel mode
- Mode bit provided by hardware
  - Can distinguish if system is running user code or kernel code
  - System call changes mode to kernel
  - Return from call using RTI resets it to user
- How do user programs do something privileged?

Crossing Protection Boundaries

- User calls OS procedure for “privileged” operations
- Calling a kernel mode service from user mode program:
  - Using *System Calls*
  - System Calls switches execution to kernel mode



Types of System Calls

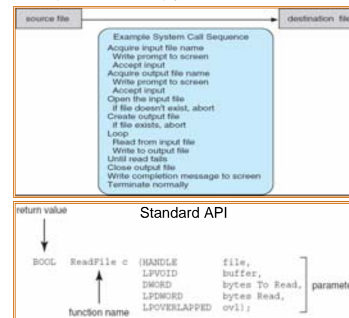
- Process control
- File management
- Device management
- Information maintenance
- Communications

System Calls

- Programming interface to services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs using APIs
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, Mac OS X)
  - Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?
  - Easier to use

Why APIs?

System call sequence to copy contents of one file to another



Reducing System Call Overhead

- **Problem:** The user-kernel mode distinction poses a performance barrier
  - Crossing this hardware barrier is costly.
  - System calls take 10x-1000x more time than a procedure call
- Solution: Perform some system functionality in user mode
  - *Libraries (DLLs)* can reduce number of system calls,
    - by caching results (getpid) or
    - buffering ops (open/read/write vs. fopen/fread/ fwrite).

Real System Have Holes

- OSes protect some things, ignore others
  - Many will blow up when you run:
  - Usual response: freeze
  - To unfreeze, reboot
  - If not, also try touching memory

```
int main() {
        while (1) {
                fork();
        }
}
```

- Duality: Solve problems technically and socially
  - Technical: have process/memory quotas
  - Social: yell at idiots that crash machines
  - Similar to security: encryption and laws

Fixed Pie, Infinite Demands

- How to make the pie go further?
  - Resource usage is bursty! So give to others when idle.
  - Eg. When waiting for a webpage! Give CPU to idle process.
    - 1000 years old idea: instead of one classroom per student, restaurant per customer, etc.
- BUT, more utilization ⇒ more complexity.
  - How to manage? (1 road per car vs. freeway)
  - Abstraction (different lanes), Synchronization (traffic lights), increase capacity (build more roads)
- But more utilization ⇒ more contention.
  - What to do when illusion breaks?
  - Refuse service (busy signal), give up (VM swapping), backoff and retry (Ethernet), break (freeway)

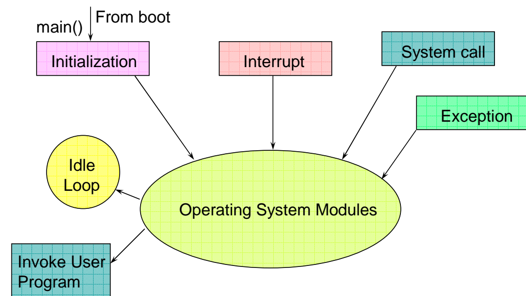
Fixed Pie, Infinite Demand

- How to divide pie?
  - User? Yeah, right.
  - Usually treat all apps same, then monitor and re-apportion
- What's the best piece to take away?
  - OSes are the last pure bastion of fascism
  - Use system feedback rather than blind fairness
- How to handle pigs?
  - Quotas (leland), ejection (swapping), buy more stuff (microsoft products), break (ethernet, most real systems), laws (freeway)
  - A real problem: hard to distinguish responsible busy programs from selfish, stupid pigs.

Operating System Structure

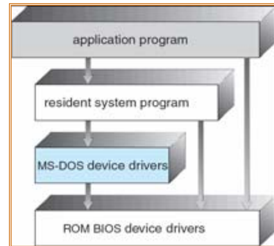
- An OS is just a program:
  - It has main() function that gets called only once (during boot)
  - Like any program, it consumes resources (such as memory)
  - Can do silly things (like generating an exception), etc.
- But it is a very strange program:
  - "Entered" from different locations in response to external events
  - Does not have a single thread of control
    - can be invoked simultaneously by two different events
    - e.g. sys call & an interrupt
  - It is not supposed to terminate
  - It can execute any instruction in the machine

OS Control Flow

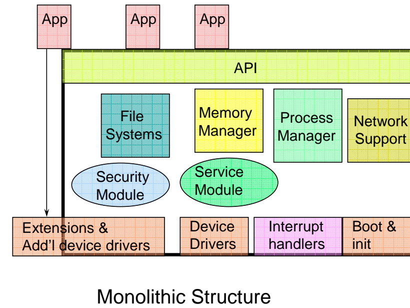


Operating System Structure

- Simple Structure: MS-DOS
  - written to provide the most functionality in the least space
- Disadvantages:
  - Not modular
  - Inefficient
  - Low security



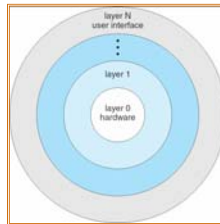
General OS Structure



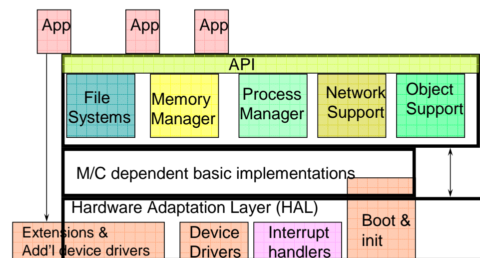
Monolithic Structure

Layered Structure

- OS divided into number of layers
  - bottom layer (layer 0), is the hardware
  - highest (layer N) is the user interface
  - each uses functions and services of only lower-level layers
- Advantages:
  - Simplicity of construction
  - Ease of debugging
  - Extensible
- Disadvantages:
  - Defining the layers
  - Each layer adds overhead



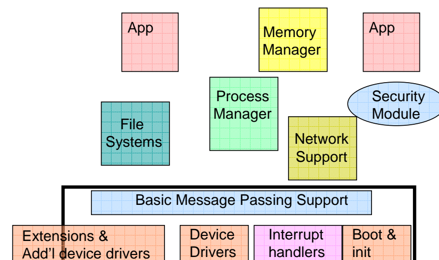
Layered Structure



Microkernel Structure

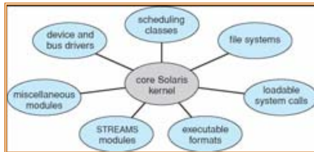
- Moves as much from kernel into "*user*" space
- User modules communicate using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
  - Example: Mach, QNX
- Detriments:
  - Performance overhead of user to kernel space communication
  - Example: Evolution of Windows NT to Windows XP

Microkernel Structure



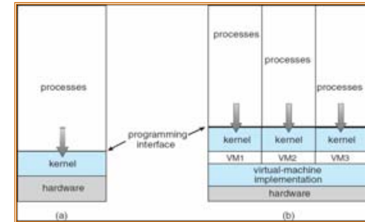
Modules

- Most modern OSs implement kernel modules
  - Uses object-oriented approach
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible
- Examples: Solaris, Linux, MAC OS X



Virtual Machines

- Abstract single machine h/w as multiple execution envs
  - Abstraction has identical interface as underlying h/w
- Useful
  - System building
  - Protection
- Cons
  - implementation
- Examples
  - VMWare, JVM



Booting a System

- CPU loads boot program from ROM
  - BIOS in PCs
- Boot program:
  - Examines/checks machine configuration
    - number of CPUs, memory, number/type of h/w devices, etc.
  - Small devices have entire OS on ROM (firmware)
    - Why not do it for large OSes?
  - Read boot block from disk and execute
- Find OS kernel, load it in memory, and execute it
- Now system is running!

Operating System in Action

- OS runs user programs, if available, else enters idle loop
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (notebooks)
  - OS computes some function (DEC's VMS on VAX computed Pi)
- OS wakes up on:
  - interrupts from hardware devices
  - traps from user programs
  - exceptions from user programs

UNIX structure

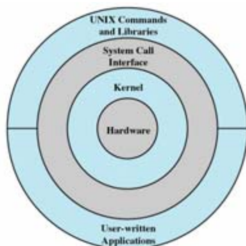


Figure 2.14 General UNIX Architecture

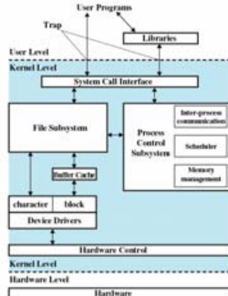


Figure 2.15 Traditional UNIX Kernel [BACH86]

Windows Structure

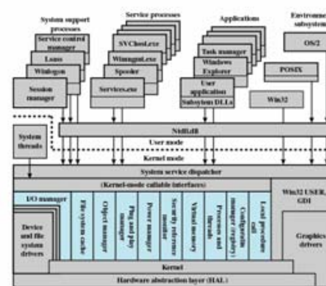


Figure 2.13 Windows 2000 Architecture [SOLO00]

Modern UNIX Systems

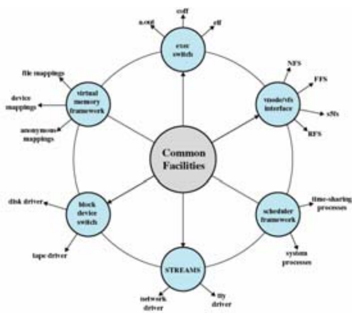
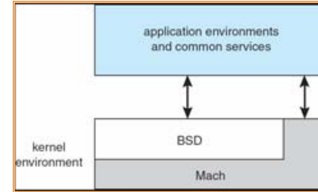


Figure 2.16 Modern UNIX Kernel [VAHA96]

MAC OS X



VMWare Structure