Misc.

• Unix Session next week – any suggestions for time?
  – Still working on room reservations. Will be posted on class homepage

• Last time:
  – A User-Computer Interface
  – A Resource Manager
  – A Control Program

• A quick walk-through the development of OSs
Last Time – Evolution of OSs

- First Generation – user at terminal
  - CPU expensive; user at terminal inefficient
  - OS Role: Some common functions started to emerge and be shared

- Second Generation – operator queues jobs in a batch
  - Key improvement: Job queue to eliminate “human” scheduling inefficiencies
  - OS Role: A resident monitor program loads each job in turn
    - Monitor passes control to job; job branches back when done
    - Problems? I/O Slow; a lot of wasted CPU time
      - Use off-line processing to pipeline across jobs and hide I/O cost
Third Generation – Batch Multiprogramming

- Problem: but I/O still expensive; can happen in middle of job
- Idea: have a pool of ready jobs in memory, switch to one when another needs I/O
- When one job needs I/O, switch to another while the I/O is done
- OS Role: Need to support multiple jobs
  - How to switch from one job to another?
  - Resources now shared: Need extra functions to share resources (including memory management and CPU scheduling)
Fourth Generation – Time Sharing

• Computing becoming cheap + new applications (e.g. interactive)
• Switch between jobs preemptively (not just at I/O) – why?
• How is this different that Batch Multiprocessing?
  – Batch multiprogramming – maximize throughput; Time Sharing – minimize response time
• OS Support:
  – timer to periodically switch among jobs
  – Concurrency management, deadlock avoidance ...
• Future generations: embedded OS’; distributed OS’ ...
Today

- A quick tour of computer organization

- Operating System Preliminaries
  - Major components of an OS
  - What services does the OS provide to programs
  - How does the OS start execution

- Start process management
A tour of Computer Organization

- A Computer System consists of:
  - Processor: the workhorse
  - Memory: holds instructions and data
  - I/O Devices: interface with the world
Processor Overview

- The processor consists of:
  - Arithematic Logic Unit (the number cruncher)
  - General Purpose Registers (data/address)
  - Control Registers (PC, IR, etc..)
  - Control unit

- Your program is compiled into a machine language program

- The processor executes a sequence of machine instructions

- Fetch – Decode – Execute cycle:
  - fetch an instruction from memory (at location pointed to by program counter) into the instruction register
  - Decode the instruction
  - execute the instruction
  - Repeat until the program halts

- The program counter is automatically incremented to point to the next instruction
Handling Interrupts

- Processor must also handle *interrupts*
- Interrupts are generated by events such as
  - I/O devices completing execution
  - Exceptions (e.g., division by zero, illegal instruction, invalid mem. address)
  - Timers
  - Software traps (most important are system calls)
- When an interrupt happens, save state and go to an *Interrupt Service Routine* (ISR)
- How do we know what is the ISR address to branch to?
  - Polling
  - Vector Interrupt Table
- What support is needed in hardware? (check for the interrupt, saving control registers)
Odds and Ends

- Events: Interrupts, exceptions and traps
- Multiple Interrupts?
  - Disable interrupts while in interrupt service routine (Sequential processing of interrupts)
  - Allow nested interrupts; stop and allow a higher priority interrupt to be processed
- Modern OS’s are event driven
  - A program communicates with the OS using system calls to request a service
  - I/O devices also interrupt OS
A program requests services from the OS using system calls

1. System call causes a trap to the OS
2. OS determines what service is being asked for and branches there
3. Branch back to the program

What is to stop any program from just branching into (or duplicating) the OS code?
Hardware Protection

- Need to ensure that a program does not harm other programs or the OS (intentionally or due to bugs)

- What support in hardware?
  - Need support for a privileged mode – only the OS runs in privileged mode
  - I/O operations only issued by the OS
  - Memory Protection – how to protect against reading/writing someone else’s memory?
  - CPU protection – how to make sure the job eventually leaves the CPU?
The Memory Hierarchy

- Would like very large, very fast, very cheap memory
- Problem: fast memory is expensive
- Solution: Build a Memory Hierarchy – a “pyramid” of different memory levels where each lower level has:
  - Lower cost per bit
  - Larger capacity
  - Slower Response Time
- How does this work? why does this work?
Why Memory Hierarchy Works

• References to memory are not random; they tend to be:
  – spatially localized (if you visit a location, likely to visit neighbors)
  – temporally localized (if you visit a location, likely to visit again soon)

• Think of the last program your wrote; does this hold true?

• Hierarchy Works because of this principle of Locality of Reference

• Example: access time to level 1 is 1; access time to level 2 is 100; Hit ratio (how often you find the item in level 1) is 98%
How Memory Hierarchy Works

• When a memory access is needed, look for it in the fastest level of memory.
• If it is there – hit, we are done
• If it is not there – miss happens
  – look for the item in the next lower level; Repeat until a hit happens.
  – Bring a block from the level where the hit occurs and update all the levels above it (replacing another block if necessary)
• Need mechanisms to: (i) find out if an item is in a particular level; (ii) decide where an item may go (mapping policy); (iii) decide what block to replace if we need a new one; and (iv) write policy
• We will look at an example – Caches
Example: Cache Memory

- Small, expensive Static RAM (on-chip or off-chip)
- access time can be 100 times faster than main memory
Direct Mapped Cache

- Assume memory address is $n$ bits and cache size $2^m$ blocks
  - Low order $m$ bits determine location in cache
  - High order $n - m$ bits stored in tag

- To check if a given address is in the cache
  - Find the entry based on the low order $m$ bits
  - Compare tag against rest of address to determine hit or miss

- Example: suppose memory is 32 words, cache is 4 words. Track the following memory accesses: 0, 3, 8, 7, 8, 2, 3, 2

- Write Policy (write-through vs. write-back)

- Cache coherency
I/O Devices

• An I/O device usually has a controller

• Controller is a small processor with some buffering space

• Actual I/O happens between device and the controller buffering space, it is then copied to memory

• I/O controller registers are usually memory mapped

• The CPU can control the device by writing to its registers or check its status by reading these registers
Communicating with I/O Devices – Programmed I/O

- One of the primary tasks of an OS is to manage I/O resources.

- Programmed I/O (read):
  - Processor issues I/O command to device
  - Keeps checking status until device is done (or failed)
  - Processor copies data from device memory to main memory

- Inefficient; what can be done?
Interrupt Driven I/O

- Instead of hanging around and checking the I/O device frequently, rely on the I/O device to signal an interrupt when it is done.

- When the interrupt is received, the processor carries out the transfer like before (assuming an input).

- Infrequent Polling is sometimes used as well.

- Why is this better than Programmed I/O?
Direct Memory Access (DMA)

- Direct Memory Access: A separate controller (usually on the system bus) carries out the I/O operation for the CPU

- The CPU is interrupted when the data transfer is complete

- Why is this better than Interrupt Driven I/O?

- What if the processor needs the memory while the DMA is underway? (cycle stealing)

- Very efficient, especially for large transfers
Discussion

• What belongs in hardware, what belongs in software?
Operating System Organization

• As with any large system, separate into modules to manage complexity

• System Components:
  – Process Management
  – Memory Management
  – Disk Management
  – I/O System Management
  – File Management
  – Protection
  – Networking
  – Command Interpreter (shells; window system)
  – Other?
Process Management

- What is a process?
  - The animated spirit of a program
  - The unit of resource allocation

- What does the OS need to worry about?
  - Process creation (and destruction)
  - Process scheduling (suspension/resumption)
  - Provide mechanisms for
    * Process synchronization
    * Inter-process communication
Memory Management

• Memory is a linear array of data accessed by address
  – Volatile – data is lost on a reboot or failure
  – Instructions and Data for each process are kept in memory
  – Shared between CPU and I/O devices

• What does the OS need to worry about?
  – Keep track of which part of memory is used and by whom
  – Decide which process to load into memory when space is available
  – Allocate and deallocate space as needed
Disk (Secondary Storage) Management

- Because memory is volatile, secondary storage is needed to keep permanent information – usually a magnetic disk drive.

- What does the OS need to worry about?
  - Free space management
  - Allocating space
  - Disk scheduling

- Why are issues different than memory?
I/O Management

• The OS has to manage the I/O devices efficiently. Furthermore, it should provide a uniform interface for programs to access the devices.

• What does the I/O system consist of?
  – A buffer/caching system (to hide the gap in speed between I/O and memory)
  – A general device-driver interface (to provide a consistent interface for the processes)
  – Drivers for the specific I/O devices (usually provided by the manufacturer)
File Management

• Files are the abstraction provided to the users to manage non-volatile information – users view of secondary storage

• What does the OS need to worry about?
  – File creation, deletion, and manipulation
  – Directories and directory manipulation
  – Mapping files onto disk
  – File backup
Other Components

• Protection
  – How to define protection/access control policies
  – How to enforce them
  – Usually integrated within the other components

• Networking/distributed processing – what if the resources are replicated

• Command-line interpreter
  – A systems program that provides an interface between the user and the OS
    * What is a systems program?
  – Accepts a user command, executes it, then waits for the next command
  – A command is usually a request for an OS service (e.g., start a process to execute a given program)
  – Example – unix (or DOS) shell, Windows
Services Provided by the OS

• The following services are exported to the user (how?)
  – Program execution
  – I/O operations
  – File-system manipulation
  – Communication (same system/with other systems; shared memory vs. message passing)
  – Error detection

• OS also provides the following services to ensure the OS functions correctly and efficiently (services to the owner of the machine)
  – Resource allocation – how to share the resources among the processes
  – Accounting
  – Protection
System calls

• Provides the interface between a running program and the OS
  – how does it compare to a command-line-interpreter?
  – Generally assisted by an assembly language instruction (trap, or syscall)
  – How are they accessed from programming languages?

• Implementation wise, a system call is like a procedure call to an OS procedure
  – Passing the parameters?
    ∗ In registers
    ∗ In a table in memory; pass a pointer as the only parameter
    ∗ Use a stack
  – Need a mode switch to the priviledged mode
  – Can you give specific examples of system calls?
Mode Switch Cost

• Recall: mode switch is switch from user to OS when a system call happens/finishes

• Context switch: mode switch + change the running process

• Mode switch more costly than procedure call?
  – Yes
    ✷ Have to create a kernel stack
    ✷ Have to map kernel memory to user space (or context switch to a kernel process – expensive)
  – Less costly than context switch
Classes of System Calls

- Process control (e.g., load process, end process, allocate memory...)
- File control (e.g., create file or directory, read, close...)
- Device control (e.g., Request/release device, write, read, mount)
- Information maintenance (e.g., get or set time, system data, process attributes, file attributes or device attributes)
- Communication (e.g., create a connection, send/receive messages)
- Other...220+ system calls in linux 2.6 kernel
Systems Programs

• System programs are “tools” for program development and execution

• They include programs for
  – File manipulation (e.g., cp, mv, grep, find)
  – Providing status information (e.g., ps, top)
  – File modification (e.g., editors)
  – Programming language support (e.g., compilers, debuggers, assemblers)
  – Program loading and execution
  – Communication (e.g., telnet, ftp, rsh, ssh)
  – Of course, the command line interpreter (tcsh, bash, command.com, etc.)

• Most users interact with system programs rather than system calls directly

• How does the OS start running?
Bootstrap

- By example – PCs
  - When machine is turned on, it runs Power On Self Test (POST), then
  - It looks for a boot sector in the very first sector of the drive
    * A boot sector is marked by 0xAA55 at offset 510
    * Usually tries the floppy then C:
  - If it is found, the sector is read to memory at location 0x7c00
  - The machine then branches to this location and runs the code in the boot sector (Master Boot Record Program)
  - The MBR loads up the rest of the OS