Administrivia

• Unix Session – in the G-area classroom 5pm on Friday
  – Not that much material to cover–just some basic unix to get you started

• Last time:
  – Very quick overview of Computer System
    ✴ CPU organization
      · Handling Interrupts
      · System calls
    ✴ Memory Hierarchy
I/O Devices

- An I/O device usually has a controller – 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
- 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
  - What do you think?
Interrupt Driven I/O and DMA

• Interrupt Driven I/O
  – Rely on the I/O device to signal an interrupt when it is done
  – the processor carries out the transfer like before (assuming an input)
  – Why is this better than Programmed I/O?

• 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
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 privileged 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?
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
Today – Process Management

• We already defined a process as
  – A program during execution (animated spirit of a program)
  – Unit of resource management

• A process state includes:
  – A program counter
  – A stack
  – A data section
  – A text (or program) section
  – Others (register values, control information)

• Key concept: processes are completely separate – a process cannot directly affect the state of another
The Life and Travails of a Process

• Birth: Process Creation – How is a process created?
  – Created by the OS to provide a service
  – Spawned (forked) by an existing process

• Death: Process Termination – How/Why does a process die?
  – Normal Termination (its task is done)
  – Errors, abuse of privileges, parent request/parent death, etc..

• In between the process lives in one of the following stages.
  – Running
  – Waiting
  – Ready
The Process State Diagram

- Think about the different transitions
- How is process creation, termination and the transitions implemented?
Process Control Block (PCB)

- OS maintains information about each process in a data structure called a Process Control Block (PCB)

- Process image: program, data, stack and PCB

- PCB includes
  - Process Identification information: process id, etc..
  - Process state
    * Program counter value
    * Registers
  - Process Control Information
    * CPU scheduling information
    * Memory management information
    * Accounting information
    * I/O status information
    * File information
Process Control Information

- Scheduling and state information: what state the process is in, its priority, what event its waiting on, scheduler-specific information

- Data Structuring – Links to PCB’s to form logical relationships. Examples:
  - Links to other PCB’s in the scheduler queues (ready/blocked/etc..)
  - Links to parent/child processes

- Privileges – what operations the process is allowed. Any limitations on resources will also be here.

- Memory Management – pointers to the process page/segment tables

- Resource ownership – what resources does the process own

- Interprocess communication – flags, signals and messages for interprocess communication
**Example PCB**

<table>
<thead>
<tr>
<th>pointer</th>
<th>process state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>process number</td>
</tr>
<tr>
<td></td>
<td>program counter</td>
</tr>
<tr>
<td></td>
<td>registers</td>
</tr>
<tr>
<td></td>
<td>memory limits</td>
</tr>
<tr>
<td></td>
<td>open files list</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
- A “dispatcher” (also called “scheduler”) runs after each process, and decides which process gets to run.
Process Switching

- Process switching:
  - Save the context (register values, PC, etc.)
  - Update the PCB
  - Move the PCB to the appropriate queue (pointer manipulation)
  - Select a new process for execution (scheduling)
  - Update its PCB
  - Update memory management data structures
  - Restore its context to the processor

- Mode switch vs. process (context) switch vs. procedure call
Scheduling Queues

- Job queue – set of all processes in the system
- Ready queue
- Device queues
- How does a process move among queues?
• **Short-Term Scheduler** decides which process to run next

• **Long-Term Scheduler** decides which new process can be brought into the ready queue
  – Controls the degree of multiprogramming of the system
Schedulers (cont’d)

- Short-term scheduler executes very often (every few milliseconds) – it must be fast

- Long-term scheduler executes infrequently (seconds), can be slower
  - What criteria should be taken into consideration by the long-term scheduler?
  - Are there enough resources to satisfy the new process and still operate efficiently?

- Processes can be classified as
  - I/O bound processes – require I/O frequently; short CPU bursts
  - CPU-bound processes – spend most time doing computation; very long CPU sequences

- Suppose all processes are blocked doing I/O...what can we do?
Medium Term Scheduler

- More often than long-term, but less than short-term
- A process that is **swapped out** is removed from memory and is not eligible to run
- A process that is **swapped in** is brought back into memory
- Why/How is this useful?
- How does the process state diagram look now?
Process Creation

- What needs to be done to create a process?
  - Need to create and initialize a new process image
- Assign a unique identifier to the process
- Allocate memory space for program, data and stack
- Initialize the PCB (what values?)
- Set up links to other PCBs
- What about process termination?
Process Creation (cont’d)

• In unix:
  – `fork()` call spawns a new process
  – similar to the standard implementation, except:
    ∗ The child gets a copy of the parent process image (with the exception of shared memory)
    ∗ Both processes start executing at the point in the code where the fork occurred – `fork` returns the child’s process id number to the parent, 0 for child
  – `execve()` can then be used to load a new executable into the new process
    
    ```
    int execve (char *const filename,
                char *const argv[],
                char *const envp[]);
    ```

• Windows NT supports similar mechanism, but also allows a direct “spawn-and-execute”
Tree of Processes on a Typical Linux System

init+-apmd
  |-atd
  |-automount
  |-battery_applet
  |-bdflush
  |-cardmgr
  |-crond
  |-gdm---gdm-+-X
     `-gnome-session
  |-gnome-name-serv
  |-gnome-smproxy
  |-gnome-terminal+-gnome-pty-helpe
     `-tcsh

* pstree command in unix shows the process tree
Process Termination

- When does termination occur?
  1. Process executes the last statement and makes the system call `exit`
     - May need to return data to its parent if the parent had executed a `wait`
  2. Parent may terminate a child process, because:
     - it's no longer needed
     - it exceeded its resources
     - the parent is exiting
       * Many OSs do not allow a child to continue if its parent terminates
       * May have cascading terminations

- Deallocate process resources, and update OS data structures
Cooperating Processes

- Independent processes are not affected by the execution of other processes

- However, an application may be organized as a set of cooperating processes – why?
  - Information sharing (e.g., shared file)
  - Computation speedup – how?
  - Modularity
  - Convenience (e.g., the command line interpreter spawning multiple tasks)

- OS provides several facilities to support this cooperation – Inter Process Communication (IPC)