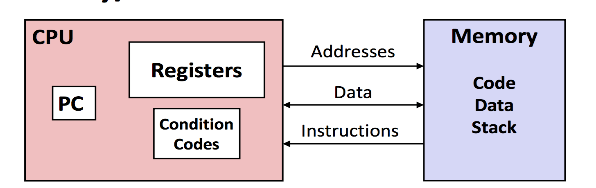
**Lecture 2- C**

* Arrays in C are fixed length and C doesn’t check bounds
* &x : get the address of x
* \*p : access the value of the pointed-to variable (deref)
* **C is a call-by-value language (**Changes made to arguments passed to a function aren’t reflected in the calling function)
* To allocate memory: **p = malloc(size in bytes)** – returns a pointer to a memory region that you can then assign to a variable of whatever type you like
* To free: **free(p)**

**Lec -3 OS Overview**

* **grep** command searches through standard input or files and prints lines that match a pattern
  + grep foo \*
* Linkers – modularity, efficiency (sep comp & space)
* Object Files
  + Relocatable object file (.o file)
  + Executable object file (a.out file)
  + Shared object file (.so file)
* Linker Symbol Rules
  + Given a strong symbol and multiple weak symbols, choose the strong symbol
  + If there are multiple weak symbols, pick an arbitrary one
* Kernel has access to everything. User must trap to the kernel state in order to do anything of privilege
* A system call can interact with the kernel and has more access while a function call is within a user created code
* Machine Code: The byte-level programs that a processor executes
* Assembly Code: A text representation of machine code

**PC: Program counter**- Address of next instruction

**Register file**- Heavily used program data

**Condition codes**- Store status information about most recent arithmetic or logical operation.  Used for conditional branching

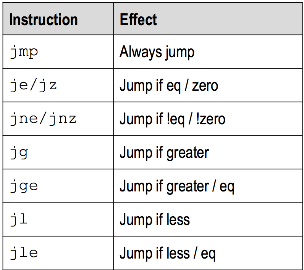
**Memory-** Byte addressable array.

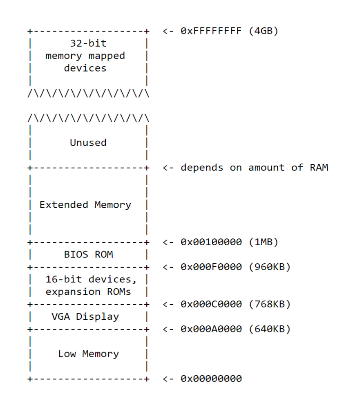
Code and user data

Stack to support procedures

* Stack Frames
  + to use arguments and return information we need to deal with the stack.
  + The portion of memory used in the stack for a single procedure call is called a stack frame.
  + The topmost stack frame is delimited by two register pointers:
    - %ebp is the frame pointer (base)
    - %esp is the stack pointer(points to top of current stack in memory)
* BootSector:
  + First 512 bytes of the boot disk.  Contains the boot loader

**BIOS**

* During the BIOS's hardware initialization, it enabled hardware interrupts
  + - We don't know what its interrupt handlers look like
* Use cli instruction to disable interrupts
* The stack pointer is set to 0x7c00 – the address of the start of the bootloader
* **IA32 Registers**
  + %eax- Accumulate
  + %ecx- Counter
  + %edx- Data
  + %ebx- Base
  + %esi- Source Index
  + %edi- Destination Index
  + **%ebp- Base Pointer**
  + **%esp- Stack Pointer**
* Physical Address Space



**Lec 6 – System Calls**

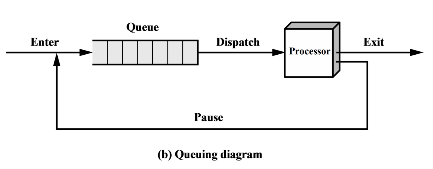
**Trap Frame**

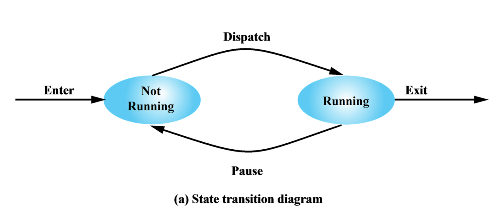
* + When we make a system call from the user mode, we trap the frame so that we can restore the state when returning from the kernel mode
  + By capturing all of this information in the trap frame structure, we can restore the CPU state exactly when we return from the system call

**Lec 6- System Calls**

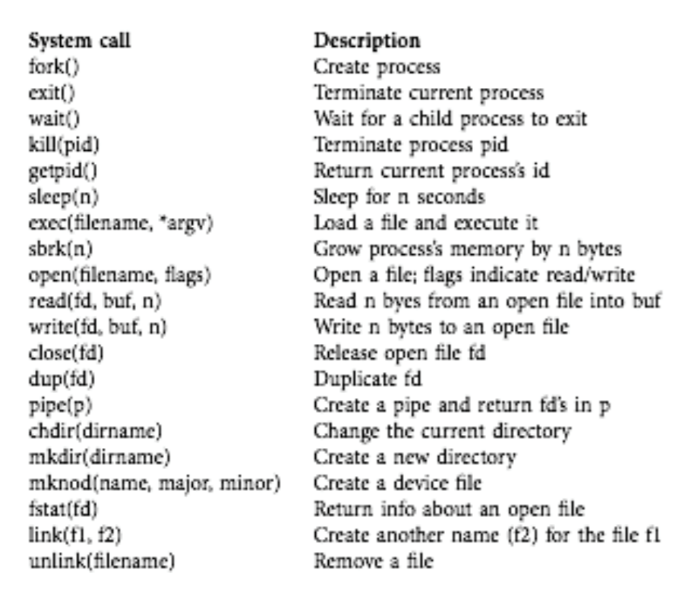
* + When executing the system call, we switch stacks. However the arguments are stored in the user stack. We use the stack pointer value saved in the trap frame to get the arguments (Arguments are at **%esp+4+(4\*arg\_no**) )
  + When fetching arguments, we check to make sure the pointer is not outside ***proc->sz***
  + In user mode, we can rely on the paging hardware to disallow access to anything outside of process's memory
  + But in kernel-mode we must do explicit checks, because the kernel has access to all memory
  + **syscall()** put the return value in ***proc->tf->eax***
  + So when we get back to userspace, we will have our return value in **%eax**

**Lec 7- Process Scheduling**

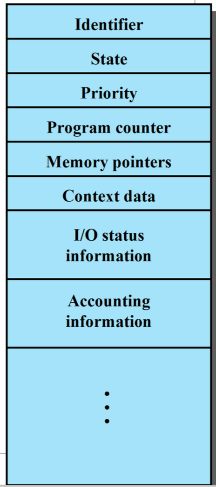
* Process Control Block (PCB)
  + Contains the process elements
  + It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
  + Created and managed by the operating system
  + Key tool that allows support for multiple processes
* Two State Process Model Queuing Diagram

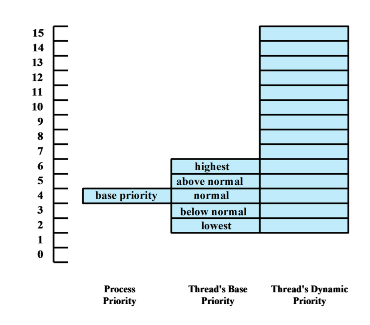
Change of Process State

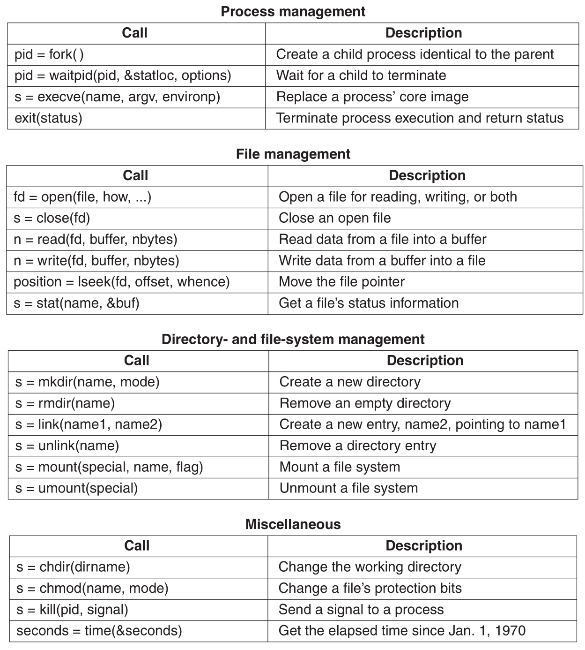
* + Save the context of the processor
  + Update PCB of processor in running state
  + Move PCB of process to appropriate queue
  + Select another process for execution
  + Update PCB of processor selected
  + Update memory management data structure
  + Restore the context of the processor



* Mechanisms for Interrupting the Execution of a Process
* Interrupt
  + Clock Interrupt
  + I/O Interrupt
  + Memory Fault
* Trap
* An error or exception condition (syscall) generated within the currently running process







Suspended Process

* The process is not immediately available for execution
* The process may or may not be waiting on an event
* The process may not be removed from this state until the agent explicitly orders the removal

Reasons for Suspension

* Swapping (OS needs to release memory)
* Timing
* Interactive User Request
* Parent Process Request

Killing Processes

* + Processes can exit on their own using a system call
  + One process can request that the OS terminate (kill) another process

Zombies

* + Zombies are processes that have exited but still have an entry in the process table
  + If a child process exits and the parent has not called wait() to get its exit status, the child becomes a zombie

**2a**.On line 15 of the C code, in the argint function, why do we add 4 to proc->tf->esp?skip over the return address. **b.**On the same line, why do we multiply n by 4? 32-bit architecture, each process is 4 bytes. What could go wrong if xv6 didn’t have the check at line 5 in fetchint? Fault, you could be outside your current address space. **4. swtch() a.** is the program counter (%eip) saved and restored by this code? The program counter is pushed to the stack when swtch is called, and

restored by the ret at the end. **b.** Give an example of one other piece of program state that is not explicitly saved by this code and explain why that doesn't cause problems when performing a context switch. Stack pointers are already on the PCB, does not need to be saved explicitly + all general purpose registers. **5a**. What scheduling algorithm is implemented by this code? Is it fair? Solution: This is round robin. Each process runs until its quantum is up (at which point the timer interrupt \_res) or it yields, and the scheduler then picks the next runnable process. This algorithm is fair: the scheduler picks each process with equal frequency, so each gets the same share of the CPU.**c. swtch:**switches stack programs. Only in assembly you’re allowed to manipulate registers. 6a. 1. Output goes into the pipe 2. Pipe sys call lowest file system. b. you’re creating 2 child processes. Cannot just make 1 child process. 7. (**F)**A process that is not in main mem is imm available for exec. **(T**) responsib of OS to control exec of process**. T** OS may suspend process if detects problem **T** example of app that uses threads is browser **F** segmentation is not visible to programmer. 8. A process is in the runnable state is in main mem and avail. When the system spends most of its time swapping rather than exec it leads to a cond known as trashing. PIC mechanism is pipas

Scheduling

* + To coordinate the running of multiple processes, the OS includes a scheduling algorithm that decides what process will be run when, and for how long
    - There are tons of different scheduling algorithms with different goals:
      * Maximize CPU utilization
      * Maximize I/O throughput
      * Make the system feel responsive to the user
      * Meet real-time deadlines
    - Preemptive Schedulers
      * Take advantage of time interrupt

*Context Switching*

Switchuvm

* + This function does two things:
    - Switches the task state segment to the user- mode one
    - Changes the current virtual address space to the process's

*Task State Segment*

* + The TSS is a special structure defined by x86 intended to help with context switching

*Swtch*

Reasons for Process Termination

* + Normal Completion
  + Over Time Limit
  + Memory Unavailable
  + Bounds Violation
  + Protection error
  + Arithmetic Error
  + Time overrun
  + I/O Failure
  + Invalid Instruction
  + Privileged Instruction
  + Data Misuse
  + Operator or OS Termination
  + Parent Termination
  + Parent request
  + Does the actual work of switching between the kernel scheduler context and the kernel process context
  + When a process gives up the CPU, yield() is called
  + yield() makes the process runnable and then calls sched()

**Lec 8 – Scheduling**

* Process Scheduling
  + **Throughput**= Jobs Completed/Time Interval
* Scheduling Algorithms
  + First Come First Serve
  + Single queue of ready processes
  + The process at the head of the queue runs as long as it likes or until it blocks
  + After it runs, you add it to the back of the queue and let the next one in line run
  + **Turnaround Time= [Time(completed)-Time(submitted)] / N**
* Round Robin Scheduling
  + A simple, fair preemptive scheduling algorithm
  + Run first process until its quantum is used up
  + Move that process to the end and run the next process until its quantum is used up
  + Assumes that all are of the same priority
* Role of Init
  + Init is the first process created
  + It spawns the system shell (sh)
  + After starting the shell, sits in a loop calling wait() in case any zombies get assigned to it
* First Come First Serve : T = Sum(CT – TC)/n