Input/Output (I/O)

- Input and Output devices operate independently of main CPU
  - Speeds vary; often much slower than CPU instructions
  - Device uses interrupts to signal OS when ready/done
    - Have you ever heard of an **IRQ (Interrupt Request Level)**?

- I/O devices are managed by the OS
  - Enforces sharing, protection
  - Requires privileged instructions to initiate I/O
  - Kernel maintains table to map device interrupts to handlers, a.k.a **device drivers**

- Communication handled through memory abstraction
  - Device buffers and registers mapped into address space
  - Data copied into address range send data to I/O device
I/O Example

1) Ethernet receives packet, writes packet into buffer
2) Network Interface Card (NIC) signals an interrupt
3) CPU stops current instruction, saves current context
4) CPU reads handler address from kernel vector table, indexed by IRQ, jumps to appropriate address (Ethernet driver)
5) Ethernet device driver reads packet from buffer, processes packet
6) Kernel restores saved context, reissues interrupted instruction
Interrupt Issues

- Interrupts suspend current process and transfer control to the OS
  - Can the OS itself be interrupted?
  - Sometimes interrupts need to be masked until the OS is ready to handle them

- Interrupts are default model for device interaction
  - What is the alternative?
  - What are the drawbacks of the alternative approach?
  - When might an OS want to avoid interrupts?
Atomicity

- Interrupts are pesky
  - They can occur at any time, including inconvenient ones
  - In particular, a handler may interfere with currently executing code paths
    » Can a handler interrupt itself?

- OS needs some way to ensure certain instructions or sequence of instructions are not interrupted
  - One option is to disable interrupts during critical section
  - Another alternative is to have special atomic instructions
    » Guaranteed not to be interrupted
    » E.g., test and conditionally set a bit based on previous value

- What if the machine has more than one CPU?
Coordinating Execution

- This lecture starts a class segment that covers processes, threads, and synchronization
  - These topics are perhaps the most important in this class.
  - You can rest assured that they will be covered in the exams.

- Today’s topics are processes and process management
  - What are the units of execution?
  - How are those units of execution represented in the OS?
  - How is work scheduled in the CPU?
  - What are the possible execution states of a process?
  - How does a process move from one state to another?
The Process

- The process is the OS abstraction for execution
  - It is the unit of execution
  - It is the unit of scheduling
  - It is the dynamic execution context of a program

- A process is sometimes called a job or a task or a sequential process.

- A sequential process is a program in execution
  - It defines the sequential, instruction-at-a-time execution of a program
  - Programs are static entities with the potential for execution
Process Components

- A process contains all of the state for a program in execution
  - An address space
  - The code for the executing program
  - The data for the executing program
  - An execution stack encapsulating the state of procedure calls
  - The program counter (PC) indicating the next instruction
  - A set of general-purpose registers with current values
  - A set of operating system resources
    » Open files, network connections, etc.

- A process is named using its process ID (PID)
Process Address Space

- Stack
- Heap (Dynamic Memory Alloc)
- Static Data (Data Segment)
- Code (Text Segment)

Address Space:
- 0x00000000
- 0xFFFFFFFF

SP, PC
Process State

- A process has an **execution state** that indicates what it is currently doing
  - **Running**: Executing instructions on the CPU
    - It is the process that has control of the CPU
    - How many processes can be in the running state simultaneously?
  - **Ready**: Waiting to be assigned to the CPU
    - Ready to execute, but another process is executing on the CPU
  - **Waiting**: Waiting for an event, e.g., I/O completion
    - It cannot make progress until event is signaled (disk completes)

- As a process executes, it moves from state to state
  - Unix “ps”: **STAT** column indicates execution state
  - What state do you think a process is in most of the time?
  - How many processes can a system support?
Process State Graph

- New
- Ready
- Running
- Waiting
- Terminated

- Create Process
- Unschedule Process
- Schedule Process
- Process Exit
- I/O Done
- I/O, Page Fault, etc.
- I/O, Page Fault, etc.
Process Data Structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the **Process Control Block (PCB)**
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process’ hardware execution state (PC, SP, regs, etc.) when the process is not running
  - This state is everything that is needed to restore the hardware to the same configuration it was in when the process was switched out of the hardware
PCB Data Structure

- The PCB contains a huge amount of information in one large structure
  - Process ID (PID)
  - Execution state
  - Hardware state: PC, SP, regs
  - Memory management
  - Scheduling
  - Accounting
  - Pointers for state queues
  - Etc.

- It is a heavyweight abstraction
PCBs and Hardware State

- When a process is running, its hardware state (PC, SP, regs, etc.) is in the CPU
  - The hardware registers contain the current values
- When the **OS stops running a process**, it saves the current values of the registers into the process’ PCB
- When the **OS is ready to start executing a new process**, it loads the hardware registers from the values stored in that process’ PCB
  - What happens to the code that is executing?
- The process of changing the CPU hardware state from one process to another is called a **context switch**
  - This can happen 100 or 1000 times a second!
State Queues

How does the OS keep track of processes?

- The OS maintains a collection of queues that represent the state of all processes in the system.
- Typically, the OS has one queue for each state:
  - Ready, waiting, etc.
- Each PCB is queued on a state queue according to its current state.
- As a process changes state, its PCB is unlinked from one queue and linked into another.
State Queues

There may be many wait queues, one for each type of wait (disk, console, timer, network, etc.)
PCBs and State Queues

- PCBs are data structures dynamically allocated in OS memory
- When a process is created, the OS allocates a PCB for it, initialized, and placed on the ready queue
- As the process computes, does I/O, etc., its PCB moves from one queue to another
- When the process terminates, its PCB is deallocated
Process Creation

- A process is created by another process
  - Parent is creator, child is created (Unix: ps “PPID” field)
  - What creates the first process (Unix: init (PID 0 or 1))?
- In some systems, the parent defines (or donates) resources and privileges for its children
  - Unix: Process User ID is inherited – children of your shell execute with your privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)
Process Creation: Unix

- In Unix, processes are created using `fork()`
  ```c
  int fork()
  ```

- `fork()`
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a *copy* of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue

- Fork returns *twice*
  - Returns the child’s PID to the parent, “0” to the child
  - Huh?
What does this program print?
Example Output

alpenglow (18) ~/tmp> cc test.c
alpenglow (19) ~/tmp> ./a.out
My child is 486
Child of a.out is 486
Duplicating Address Spaces

Parent

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}

Child

child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
Divergence

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```

```
child_pid = fork();
if (child_pid == 0) {
    printf("child");
} else {
    printf("parent");
}
```
Example Continued

alpenglow (18) ~/tmp> cc test.c
alpenglow (19) ~/tmp> ./a.out
My child is 486
Child of a.out is 486
alpenglow (20) ~/tmp> ./a.out
Child of a.out is 498
My child is 498

Why is the output in a different order?
Why fork()?

- Very useful when the child...
  - Is cooperating with the parent
  - Relies upon the parent’s data to accomplish its task

- Example: Web server
  ```c
  while (1) {
    int sock = accept();
    if ((child_pid = fork()) == 0) {
      Handle client request
    } else {
      Close socket
    }
  }
  ```
Wait a second. How do we actually start a new program?

```c
int exec(char *prog, char *argv[])
```

**exec()**
- Stops the current process
- Loads the program “prog” into the process’ address space
- Initializes hardware context and args for the new program
- Places the PCB onto the ready queue
- Note: It **does not** create a new process

What does it mean for exec to return?

What does it mean for exec to return with an error?
Creation Conundrums

- fork() is used to create a new process, exec is used to load a program into the address space

- What happens if you run “exec csh” in your shell?
- What happens if you run “exec ls” in your shell? Try it.

- fork() can return an error. Why might this happen?
Process Termination

- All good processes must come to an end. But how?
  - Unix: `exit(int status)`, NT: `ExitProcess(int status)`

- Essentially, free resources and terminate
  - Terminate all threads (next lecture)
  - Close open files, network connections
  - Allocated memory (and VM pages out on disk)
  - Remove PCB from kernel data structures, delete

- Note that a process does not need to clean up itself
  - Why does the OS have to do it?
wait() a second…

- Often it is convenient to pause until a child process has finished
  - Think of executing commands in a shell
- Use `wait()` (WaitForSingleObject)
  -Suspends the current process until a child process ends
  -`waitpid()` suspends until the specified child process ends
- Wait has a return value…what is it?
- Unix: Every process must be reaped by a parent
  -What happens if a parent process exits before a child?
  -What do you think a “zombie” process is?
while (1) {
    char *cmd = read_command();
    int child_pid = fork();
    if (child_pid == 0) {
        Manipulate STDIN/OUT/ERR file descriptors for pipes, redirection, etc.
        exec(cmd);
        panic("exec failed");
    } else {
        waitpid(child_pid);
    }
}
Process Summary

- What are the units of execution?
  - Processes

- How are those units of execution represented?
  - Process Control Blocks (PCBs)

- How is work scheduled in the CPU?
  - Process states, process queues, context switches

- What are the possible execution states of a process?
  - Running, ready, waiting

- How does a process move from one state to another?
  - Scheduling, I/O, creation, termination

- How are processes created?
  - CreateProcess (NT), fork/exec (Unix)
Next time…

- Read Chapter 4
- Homework #1 due
- Project 0 due