Question 8.1–1: (Solution, p 3) Name at least two generic purposes of an operating system.

Question 8.1–2: (Solution, p 3) One way of receiving a response from a device is to **poll** it periodically about whether it has any additional information. What is wrong with this technique, and how do most modern computer systems circumvent it? Explain the alternative.

Question 8.1–3: (Solution, p 3) When the CPU receives an interrupt, how does it determine what instruction to execute next?

Question 8.1–4: (Solution, p 3) Because other users' information is stored on the disk, a multiuser computer system cannot allow a process to access the disk directly. Yet programs often need to read from the disk. Explain how a computer system can allow this without compromising security.

Question 8.1–5: (Solution, p 3) For each of the following CPU instructions, which of the following applies?

A. The instruction should be permitted whether the CPU is running in user mode or kernel mode.

B. The instruction should be allowed only when the CPU is running in kernel mode.

a. Switch from user mode to kernel mode.

b. Send a software-initiated interrupt to the CPU.

c. Disable interrupts by clearing the interrupt flag.

d. Send information to an I/O device.

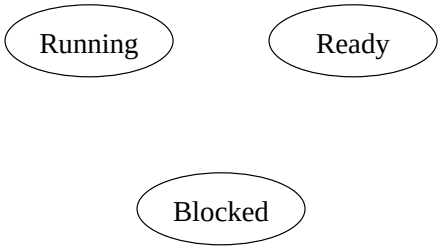
e. Call a subroutine.

Question 8.1–6: (Solution, p 3) What distinguishes system calls (like `write`) from library functions (like `printf`)?

Question 8.2–1: (Solution, p 4) Outline the details of what happens when a process makes a system call necessitating communication with a device.

Question 8.2–2: (Solution, p 4) Because the CPU can run only one thread of execution at once, when a user program is running, the operating system cannot be. How, then, can it manage to preempt a running program that runs for a long period of time and attempts to run beyond its time slice?

Question 8.2–3: (Solution, p 4) In the following diagram illustrating the various states a process can be in, draw arrows connecting each pair of states that a preemptive operating system may move a process between. Label each arrow with a brief description of a situation where the operating system would move the process as indicated.



Question 8.2–4: (Solution, p 4) Explain why the operating system would move a process in the *Running* state to the *Blocked* state instead.

Question 8.2–5: (Solution, p 4) Describe a situation where the operating system would move a process from the *Blocked* state to the *Ready* state.

Question 8.3–1: (Solution, p 4) What does Linux do when requested to perform the `execve` (or `execvp`) system call?

Question 8.3–2: (Solution, p 4) Explain what a Unix shell does so that the output of a process is redirected into a file instead of to the screen, as it is told to do in the following shell command.

```
unix% wc words > count
```

Question 10.1–1: (Solution, p 5) In an operating system, how are threads and processes different?

Question 10.1–2: (Solution, p 5) Suppose we are writing a Web server. Explain how threads would be useful for our program, and describe why they are useful in this context.

Question 10.1–3: (Solution, p 5) Name and explain two reasons that threads are a useful in programming.

Solution 8.1–1: (Question, p 1) There were three that we examined in class, but potentially there are others too.

- Abstract complex resources.
- Provide hardware compatibility.
- Protect the system from untrustworthy programs.

Solution 8.1–2: (Question, p 1) Polling wastes precious processor time repeatedly querying devices. Modern computer systems provide a way for devices to send "interrupts" to the CPU, allowing the device to send a signal to the CPU, whereupon the CPU would enter its "interrupt handler" to handle the fact that the device is now ready.

Solution 8.1–3: (Question, p 1) Each interrupt has an associated identifying number. The CPU uses this as an index into the exception table, where it finds the address of the first instruction in the exception handler for that identifier. It executes the instruction at this address next. (The OS would have registered its exception handler into the exception table as the computer started.)

Solution 8.1–4: (Question, p 1) The CPU implements two modes, kernel mode and user mode, and prevents a process from accessing the disk directly while in user mode. The operating system enters user mode every time it exits into a user program, so that the user program runs in user mode and cannot access the disk. But the user program can initiate an interrupt using a CPU instruction, whereupon the CPU will switch into kernel mode as part of the interrupt process and jump into the interrupt handler, implemented by the operating system. The operating system's handler, running in kernel mode, can access the disk on the process's behalf (after it verifies that the request is legitimate) before returning back into the user program in user mode.

Solution 8.1–5: (Question, p 1)

a. B

b. A

c. B

d. B

e. A

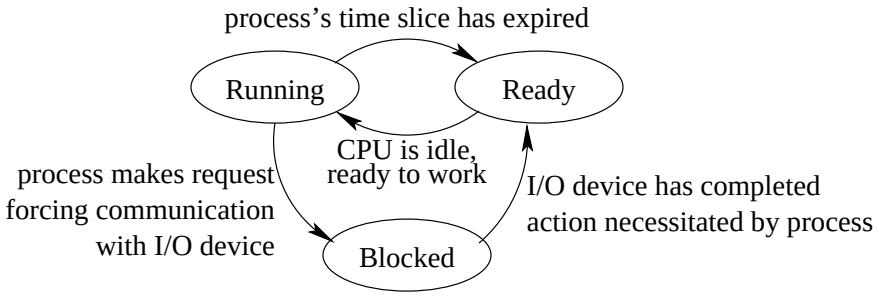
Solution 8.1–6: (Question, p 1) Code for system calls are included in the operating system, while code for library functions are included with the running program. Thus, a system call's code runs in kernel (unprotected) mode, while a library function's code runs in user mode. A program enters a system call by initiating a software interrupt, while a program enters a library function by calling a subroutine included with the program.

Solution 8.2–1: (Question, p 1)

1. The running process sends a system call via an interrupt.
2. The CPU looks up the location of the interrupt handler in the exception table and jumps there.
3. The interrupt handler saves all the registers of the running process into that process's entry of the process table.
4. If another process is already waiting for the device to respond, the interrupt handler places the process into a waiting queue for that device. Otherwise, the process's request is sent to the device.
5. The interrupt handler selects the next process to execute from the ready queue.
6. The interrupt handler restores the registers to the values saved in the next process's entry of the process table.
7. The interrupt handler returns to the address stored in the next process's entry of the process table.

Solution 8.2–2: (Question, p 1) Before beginning to run the user program, the OS schedules the clock device to send an interrupt. When this interrupt occurs, the CPU automatically jumps into the exception handler for the clock's interrupt, at which time the OS is running and can preempt the process.

Solution 8.2–3: (Question, p 1)



Solution 8.2–4: (Question, p 1) When a running process requests interaction with a device that cannot immediately respond, the operating system moves it into the *Blocked* state so that it will not occupy CPU time while the device is working.

Solution 8.2–5: (Question, p 1) A process is typically in the *Blocked* state when it is waiting for some response from a I/O device (such as the keyboard or hard drive). The operating system would move it into the *Ready* state when it has received a response for the I/O device for the blocked process.

Solution 8.3–1: (Question, p 2) It replaces the currently running process with the program specified in the parameters to `execve`. The new process keeps the state of the replaced process, but its memory changes to the memory image required by the program, and execution enters the started program. Execution will not return to the program calling `execve` unless an error prevents the OS from executing the request.

Solution 8.3–2: (Question, p 2) Since the *wc* command writes to file descriptor 1, the shell must insure that file descriptor 1 for the process running *wc* will refer to the `count` file. To accomplish this, it closes file descriptor 1 in the child process it creates for running *wc* and then designates descriptor 1 to refer to the `count` file instead. Thus, when *wc* runs and writes to file descriptor 1, the output will go into the file instead of to the screen.

Solution 10.1–1: (Question, p 2) Threads are individual execution sequences occurring within the same process. The operating system allocates resources (such as memory and file descriptors) to a process as a whole, and individual threads within that process share these resources. Thus, a process is a resource allocation unit with at least one thread, while a thread is an execution sequence working within the resources allocated to the process.

Solution 10.1–2: (Question, p 2) For good performance, a Web server must communicate with many browsers simultaneously, and it is natural to have a single thread handle each individual browser connection. This technique is useful for three reasons. [One is sufficient for the solution.]

- Implementing the Web server as a single process without threads requires the programmer to write code that switches between discussions among browsers, which is difficult and conducive to programmer errors.
- If the Web server were implemented to create a new process for each Web browser connection, the high overhead of process creation would damage the program's efficiency.
- Also in the multi-process solution, having multiple processes would interfere with the possibility that the responding processes might want to share information between themselves.

Solution 10.1–3: (Question, p 2) Any two of the following would be good.

- A programmer using threads can conceptualize a process as doing one thing at a time, even though the process will actually be doing many things simulataneously.
- Threads use less system resources than separate processes.
- A process can continue perform computation during long I/O tasks (even when a system call is blocked).