Gabriel Rojas-Westall
EECS 343
April 22, 2019

Homework 1

1. Running - transition from ready to running means the process has been *scheduled*. A process enters this state when the OS recognizes the CPU is not being used and can run this process.
Ready - running to ready means the process has been d*escheduled*. A process enters this state when it is waiting for the OS to run it. If a process was started while another process was running, if the OS wants to finish the other process before starting this new process, the new process would remain in the ready state until the OS scheduler decides to run it. A process might also be in the ready state after it carries out an I/O operation, but again the OS is running another process, so this process has to wait for the OS scheduler to run it again.
Blocked - initiated by an I/O operation like writing to disk or waiting for a network packet.

2. EMBRYO: related to a race condition. When allocating space for a process in the process table, the EMBRYO state is used int order to initialize the state for a process and thus reserve its space in the stack. Another process might run at some point in between when this process is initialized and when it carries out its instructions, so the EMBRYO state tells other processes that this process in the process table is not unused and its memory cannot be used for another process, thus avoiding the race condition.
ZOMBIE: Unless a parent calls wait(), it may finish before its child. The ZOMBIE state is used so that the child is still accessible to the parent in the process table once it has finished executing. exit() does not return, but the parent process needs to check on the child's exit status before it can continue running if wait() was called. In order for the parent to do this, the child still needs to exist in the process table, so the status is changed to ZOMBIE to allow the parent to check on the exit status as opposed to the child process disappearing from the table.

3. Since the OS can switch between different processes through context switches, the time that this particular deterministic program takes to run could also depend on what other processes are running on the machine at the time the program is run. If many other programs are being run, the program will likely take longer to run because the OS needs to handle more processes at once, performing context switches to make it seem as though all are being run at the same time. If other processes are run while this program is running, data stored in caches may also be changed, possibly causing more read/write time. Lastly, scheduling policies could change the amount of time it takes to run if the policies like SJF or FIFO are being used.

4. If a process were to be using 13% of the CPU according to one of the utilities, it would basically mean that it's using 13% of the OS scheduler's "time units." So really only one process would be being run on the CPU at a time, and the OS may decide that it needs to switch to another process at some point in time, but this process is taking up 13% of the schedule. This gives a better sense of what processes are actually being run (or waiting to actually run)/how large of a process they are.

5. A shell and graphical user interface to make it user friendly :)

6.
- **Read the real-time clock (telling the date and time of day)** not privileged
- **Set the real-time clock** privileged
- **Read memory** privileged for physical memory, not privileged for virtual

- **Write memory** privileged for physical memory, not privileged for virtual
- **Trigger a system call** not privileged
- **Turn off interrupts (make the CPU ignore interrupts)** privileged

7.
- **Exceptions** - division by 0 (arithmetic) and invalid memory access
- **Software Interrupt (traps)** - A call to fork and a call to exec by the user causes a trap instruction to be called
- **Hardware interrupts (generated outside the CPU)** - I/O devices (like typing) or timer interrupt

8. Lots of smartphones implement ARM architecture. One difference between ARM and x86 we saw in lecture was over context switches. When a context switch is made, the mobile OS saves and loads many more registers than x86 for each switch. Because there are more registers to deal with in ARM, context switches are likely faster in x86. This makes sense to me from a user perspective because mobile devices usually are not running as many programs at one time compared to PCs. PCs may possibly need to handle lots of different programs at once while mobile devices likely won't be handling more than a few from the user. However, this might also just have to do with how scheduling policies are implemented so that process execution can be optimized for the policies that are implemented to try to keep energy consumption low (read below).

9. Compared to x86, ARM consists of many more features concerned with battery life (since phones have way less room for battery). Therefore, scheduling policies are altered in ARM to prioritize energy consumption more than x86 might as opposed to trying to complete all processes as quick as possible and consuming a lot of energy.

10. Timesharing OSs might not just need to split up which processes are being run for each user (so that each user feels like they are using the machine by themselves), but the OS also needs to make sure that the resources are properly managed. Multiple users means that resource allocations need to be kept separate. States for multiple users and programs need to be kept in memory and easily switched between. Because it has to keep track of all these states, it could slow down drastically when multiple users are trying to interact with the machine. Additionally, if a system failure or error occurs for one user the OS needs to make sure that no other users are affected by it. Lastly, there are more security risks associated with timesharing because processes from multiple users will have access to memory.

11. The machine can keep a scheduler that handles each virtual machine on the server, running all of their processes with a smart scheduler. If one user is requesting resources at a single point in time, the scheduler will likely prioritize the processes for that virtual machine and schedule those to be run. If multiple users are requesting resources, the scheduler will alternate between processes between the users to make it seem to each user as though the machine is only being utilized by them. The server would behave similarly to a timesharing OS in that the machine would have to allocate its resources and make sure that no one virtual machine can affect another (like a machine level crash). Most cloud computing servers do this using software that makes it so each VM is always executing instructions in user mode as well as checks to make sure privileged instructions made by the user are called correctly and safely so that no VM can affect another.