David Schirduan
03/22/2013
CSCI340 Midterm

1.
   a) Since there are no bugs, and everyone is completely honest and trustworthy, this includes programmers, and thus, their programs. Ideally, all programs would be given equal chance to be scheduled, with each task yielding to other tasks as needed. If no programs used interrupts, and all programs yielded whenever it was best to do so, then the processor could allow programs to self-regulate one another.
   b) Memory should be allocated on a priority basis, as well as on a first come, first serve basis. Priority is needed for necessary applications, such as monitor management, hard disk security, and operating system requirements. However, individual, low-priority programs should be loaded up in a queue for memory, and allocated on a first come first serve basis. If the set of applications do not all fit in memory, then the aforementioned queue would have to be implemented, allowing programs to “wait in line” for memory to be available.
   c) Disk space should be divided up into logical sectors, with certain sectors being lent as needed. As for the instance of a user monopolizing disk space, this would lead to the necessity of doing post-program cleanup. Sort of like the java garbage collector, data that is no longer needed or necessary should be freed as soon as possible.

2.
   a) The best way to prevent memory from being corrupted or accessed by other programs is to lock down the section of memory being used. That way only one program can access certain parts of memory at a time.
   b) Assigning certain parts of memory different permission levels prevents other users from accessing or altering other users' documents. Different tiers of permissions would work best.
   c) The Operating system should closely monitor the ports of a computer. Unless a special exception is made, alterations should not be allowed outside of the local system.

3. I am currently using a minimalist Ubuntu linux distribution. It tries to balance between reliability, performance, and adoption. The operating system is mobile insofar as it is very lightweight and doesn't require as many resources to run efficiently, making it perfect for laptops and netbooks. Adoption and usability are sacrificed for performance, resulting in a complicated OS, but a very fast OS.

4. All three mechanisms are required for smooth operations in an OS. Without privileged instructions, any program could make potentially fatal or dangerous commands directly. Privileged instructions route low-level powerful instructions through the OS, allowing for filtering and modification. Without this safeguard, programs could purposefully or accidentally corrupt the base parts of a system. Without memory protection, programs would be able to alter memory or processes relied upon by other programs, which could even be done while a program is running! By locking down parts of memory, it protects from alteration from other programs. Without timer interrupts, there would be no way to moderate or manage which processes are firing at which times. Timer interrupts allow for changes to be made in the current process/thread in between operations or commands. Without this pause and control, programs would not be able to coordinate or alternate between one another.

5. There are three ways to go from User to Kernel mode. Exceptions, which is an unexpected condition that forces the hardware to stop executing the code; Interrupts, which halts the code because of some external signal or event; System Calls, which allow the program to request changes made through the hardware, kinda like calling the cops to perform a dangerous task.

6. The boostrap loader, also called the boot ROM.

7. There are several different types of interrupts, and the CPU has to decide between divide-by-zero, file read, or timer-interrupt. The hardware saves the pointers and registers on the Interrupt Stack. If another interrupt occurs while the current one is being processed, the interrupt is put onto a queue. After the interrupt is handled, the interrupted code is reloaded from the stack, and the process resumes where it left off.

8. A Unix fork can return an error. The man page specifies errors for lack of memory available for a new process. There are errors returned because fork is a system call, and is routed through the Operating system.

9. No, exec cannot return errors. There are no errors returned because exec is not a system call, and is instead a direct application of the lower level code.

10. When we run this program, children are produced, with each child producing children of it's own. The number of running processes grows exponentially until the computer runs out of memory. Then an error is returned indicating no more available memory.

11. The Unix Wait command is different than most waits. It merely pauses the current process and allows another process to start. It acts more like a thread join than a wait.

12. When we run “exec csh” in the shell, the shell crashes. This is because exec stops the current process to run another command, which is very similar to what 'csh' does. Trying to stop and start nested processes crashes the original process: the shell.

13. When we run “exec ls” in the shell, the shell crashes. This is because exec is meant to run scripts, not commands.

14. It would creates 2^n-1, since we don't count the initial parent. This would result in 31 processes.

15. The first part creates two children, resulting in three different processes with different values of x. These values are 10, 15, and 20.

16. The program shows the output of several concurrent threads. First it shows when each thread starts to run, and prints out the ID of that thread. Then is prints out when the thread has completed and print its ID number + 100. This program has different results every time it is run. When I'm listening to an online radio station, everything is a lot more fragmented, and things don't finish in order.

17. NTHREADS acts as a limiter, and only allows a maximum of NTHREADS+1 threads while running.

18. When we delete that second for loop, we aren't waiting for the other threads to complete. Sometimes, all of the threads won't finish before the main thread completes. This results in only a few threads outputting their IDs before the main closes the program.

19. The variables in go() are in a per-thread state. The variable n is saved in the process, while np is saved in TCB.

20. The variables in main() are in a shared state, since they are all distributed and accessed among threads. These are saved on the heap.