Swasti Mishra
Professor Beck
COSC 361
10 October 2022

Midterm 1

1. **Trapdoor Mechanisms**

   Code executing in a privileged mode can invoke functions that are cannot be invoked by code executing in nonsupervisory mode. When a trapdoor mechanism is invoked by code executing in a privileged mode it initiates the execution of code in nonsupervisory mode. It is not possible for code executing in nonsupervisory mode to reenter supervisory mode except through another protected mechanism. There are two privileged modes in which code can execute in Linux: 1) in kernel mode and 2) in user mode but with the user identify of the "supervisory user" also known as "superuser" or "root".

   a. Explain in detail the trapdoor mechanism that code running in kernel mode implements to enter user mode, how the reverse transition occurs and how it is protected.

   If code is running in kernel mode and then needs to enter user mode, the OS will use a "return from trap" instruction.

   Usually what happens is that an OS will boot in kernel mode, set up interrupt vectors, start the first process, initialize a trap table, and then switch to user mode. At this point, programs can be run. If the running program or code requires kernel mode, then the trap handler will be run when a trap is triggered. The trap handler is something installed by the OS, essentially just a hard-coded memory address of code to run when the trap is triggered. This handler will run the code at the memory address, which immediately interrupts the program. This interruption is important- it allows each of the three system layers to be protected from one another while ensuring that they can still work together (the three layers being the program which is running in a protected mode, the operating system which is running in a privileged mode, and the hardware which is keeping the code that is running in protected mode from running in privileged mode). Once the kernel is done executing whatever the code asked for, the OS will use the return from trap instruction to resume the program.

   b. Explain in detail the trapdoor mechanism that user code running as superuser implements to begin executing without supervisory privileges, how the reverse transition occurs and how it is protected.

   The trapdoor mechanism is related to the mode bit, which determines whether the user is in supervisory mode or not. This is also known as the Processor Control Register and can be changed using the "sudo" command. Sudo will allow a user to use superuser commands. Typing "exit" will allow a user to leave superuser permissions. There are few permissions that prevent a user from doing this, besides having the correct password and administrator permissions.

2. **Multitasking and I/O**
   User process A performs a read() system call to accept a character of input from a terminal device, but no such character is available. User processes B and C both perform long computations making no system calls until long after a character is entered. The processor is single core, and there are no other processes. What are all the possible states of A, B and C (INITIAL, RUNNING, READY, BLOCKED or ZOMBIE) just before the character is entered? Explain your answer.
   Process A is waiting for an input that it will never receive. Because the processor is single core, process B and C will not be able to run until process A completes. For this reason, process B may be running, which will mean that process C may be ready. Process C may also be running, and process B may be ready. Either situation works, it just depends on which process needs to run first after process A.

3. **Cooperative vs. Preemptive Multitasking**
   a. What is the difference between cooperative and preemptive multitasking?
   The difference between cooperative and preemptive multitasking is when the operating system decides to let another process occur. In cooperative multitasking, for example, the operating system will never initiate a context switch. It will allow the running process to determine when other processes can happen, which means that processes must be designed in such a way that they cooperate otherwise the scheduling system will not work. In preemptive multitasking, however, the operating system can choose to context switch. For example, the operating system may choose to switch to a higher priority process, and lower priority processes may take longer to initiate. In this type of multitasking, processes do not have to "collaborate".

   b. In both cooperative and preemptive multitasking, a clock tick will occur 60 times per second. Which form of multitasking has a higher overhead in handling clock ticks?
   Preemptive scheduling has a higher overhead when it comes to handling clock ticks.

   c. Explain the reason for your answer in part b – what causes the difference in overhead between the two forms of multitasking?
   Preemptive scheduling has a higher overhead when it comes to handling clock ticks because switching to different processes is not instantaneous- it often consumes clock ticks. Also, because the operating system is interrupting processes, there must be a context switch (storing and restoring a task), which also takes time. Context switching requires switching registers, updating tables, and switching the stack pointer. All of this is computationally expensive.

4. **Scheduling**
   A high priority process executing in user mode enters an infinite loop from which it will never exit, and during which it will not make any system calls or cause any error conditions. Explain when/whether the looping process will run and when/whether other processes will run under each of the following scheduling policies:

a. First In First Out

The looping process will run, but other processes will not get to run under First In First Out. This is because the high priority process must complete entirely before other jobs that follow it can run. Because the high priority process has entered a loop that it will never complete, the other processes will never begin.

b. Shortest Remaining Job First

Shortest Remaining Job First will result in the same situation as First In First Out in this situation. This is because the highest priority process has already begun. If the other processes had arrived at the same time as the high priority process, they would have run first, depending on how short they were. But they cannot run because the looping process has already started, and that process must be resolved before the next jobs can begin. The looping process will run endlessly.

c. Round Robin

Round Robin will behave differently from First In First Out and Shortest Remaining Job First. This is because Round Robin will cut each job into a time slice. Regardless of the infinite looping or inability to exit, the scheduler will only run the high priority process for a few seconds before switching over to another process. In this way, the other processes will be able to run under this scheduling policy. However, the looping process will still run forever once the other processes are complete.

d. Priority Scheduling with Aging

In this scheduling system, the lower priority processes will eventually get to run, and it is because the "higher" priority process takes so long! In Priority Scheduling with Aging, the lower priority processes will get an incrementally higher priority the longer they wait. The eternally looping process will run until the lower priority process get a higher priority that the looping process, and then they will run and complete. Once they have completed, the operating system will go back to running the eternally looping process.

5. **Process Creation**

A direct (single level) page table defines a virtual address space that contains code, data and program stack as distinct sets of page table entries (pointers to memory frames). The page table itself is an array of page table entries that resides in physical memory that is accessible only in kernel mode. A pointer to the base of the page table is stored as a field in the kernel's task struct.

In the implementation of the fork() system call the kernel constructs the page table for the child process and populates it with page table entries that point to memory frames. The parent and child share the memory frames that hold the code.

a. Explain which frames pointed to by entries in the child's page table are generated in each of the following ways and why:

i. Copied from the parent's page table entry.

Registers, address spaces, and all the content that is shared with the parent (the task control block) is copied.

ii. New frames allocated by the kernel and filled with data from the parent's frames

The parent's address spaces and the child being 0 instead of the pid for the return statement are new frames.

b. Assume that the kernel communicates the return value of a system call using processor register zero. How can the parent and child processes determine whether which one they are once the fork() system call returns?
The child process will return 0, and the parent will return the pid of the child.

c. When the child terminates the parent can access its exit code (the argument to the exit() system call or -1 if the kernel terminated the process due to an error). Where is this code stored after the child has terminated?
When the child process ends, a signal called "sigchild" is created, and an entry is added to the process table. The process table keeps track of, in, part, children's exit codes.