90/100

Midterm Examination
CS 111
Fall 2015

Name: Max Chern

Answer all questions. All questions are equally weighted. This is a closed book, closed notes test. You may not use electronic equipment to take the test.

1. One principle of achieving good robustness in a system is to be tolerant of inputs and strict about outputs. Why? Describe an example in the context of operating systems.

modules and subsystems typically depend on the correctness of other modules and subsystems in order to perform their jobs and duties. Being tolerant of inputs ensures that the module can accept most outputs from other modules so that the system doesn't break down completely when an input is slightly off. Being strict about outputs ensures that other modules reading your output can understand and utilize your output. This tends to control and decrease the propagation of errors in a system. One example is when functions depend on the mathematical values of other functions. If the value is "close enough" then the function can output it successfully and the system keeps running smoothly. Strict outputs will ensure that others can use the output. Things will only start to break down if the output is seriously off.

10

2. What is the advantage of using the copy-on-write optimization when performing a fork in the Unix system? -

When performing a fork in the Unix system, a parent process forks a child process that has its own heap and stack. This child process's heap and stack is identical to the parent's - it is as if this process had been running the program from the beginning and has gotten to the same point as the parent process. If the parent process has a huge stack with a lot of used memory, copying it all to the child process's stack can be expensive. In some cases, the child process won't use anything stored in there at all. By using the copy-on-write optimization, the stack won't be copied over until the process uses it. After that, the parent will have the original copy and the child will have its own. This optimization can save time if a child process never needs to touch the memory.

10.

3. What is emulation? What is the main challenge in software emulation?

Emulation is a virtualization abstraction technique in which the OS attempts to emulate a certain interface, so that anything interacting with the emulation will believe it is interacting with the real thing rather than an emulation. The main challenge in software emulation is producing exactly the same interface and function of the desired software when in reality the hardware does not have the same capabilities. The OS must use tricks to make sure that everything appears the same to the user.

performance

5

4. Round Robin, First Come First Serve, and Shortest Job First are three scheduling algorithms that can be used to schedule a CPU. Which one is likely to have the largest overhead? Why?

Round Robin is likely to have the largest overhead. Overhead during scheduling comes from the context switches that occur when the scheduler decides to temporarily halt a process/thread and resume/start another. First Come First Serve (FCFS) and Shortest Job (SJ) have no requirement to switch between processes/threads. Some of the time the processes/threads will run to completion before another is started, in which case throughput is maximized and overhead is minimized. However, Round Robin (RR) by definition involves a requirement that the scheduler must switch to another process/thread after the currently running process/thread has been running for a certain amount of time (this amount of time is called a quantum). Therefore, there will likely be many more switches and therefore more overhead. However, RR is good for preventing starvation and minimizing wait time.

10

5. What is the difference between a first and a second level trap handler? Describe one advantage of using this two-level approach to handle traps.

A first level trap handler saves the state (stack/ program counter of the process, while the second level trap handler actually deals with the problem that occurred. One advantage to this approach is that the first level handler can notify the second handler of certain properties of the trapped process (like its state or mode) and therefore allow the second level trap handler to deal with the problem more specifically. In general, more levels/layers are better because they enforce modularity and also hide grubby details of lower levels/layers from higher levels/layers.

10

6. What is fate sharing? Three common interprocess communications mechanisms are messages, shared memory, and remote procedure calls. For which of these is fate sharing most likely? Why?

Fate sharing is when a process (or a whole system) breaks down because of an error in another process that it depended on. Basically, this process shares the same fate as the process that produced the error and also breaks down. Fate sharing is most likely to occur in shared memory. Messages are communications between processes that have a very specific format. Messages are safe and secure for the most part, as they limit the interaction between modules/processes. Remote procedure calls are also relatively safe. In true remote procedure calls, the processes are on different machines, and so it is impossible for one processes to mess with another process's memory, and so it is less likely to have a fatal error that will also kill the other process. Even in remote procedure calls on the same machine, there is a high level of isolation/protection. However, when processes communicate via shared memory, it is much more likely for a fatal error in one process to bring down another process with it as well. For example, one process could write to the wrong part of memory and fail, and the process depending on its correctness could then also fail due to the incorrect write to memory.

10

7. What is a bus master? Why is a device other than a CPU likely to become a bus master, and what operations will it typically use this role to perform?

A bus master can request data from devices, while bus slaves are only allowed to respond to requests. A device other than a CPU is likely to become a bus master because it can get very expensive when the CPU is tasked with handling device data transfer. Device controllers will often act as bus masters. The CPU will tell the device controller what data it needs, and then go on to process other tasks. Meanwhile, the device controller will deal with data from the device and write that data to memory. Once the requested data is processed, it will signal the bus, and it will result in an interrupt that notifies the CPU of completion.

10

8. What is the asynchronous completion problem? Is a spin lock a good solution for this problem? Why?

The asynchronous completion problem is the problem where one process depends on another and must wait for the other process to perform/finish some action before it itself can go on to do its own job. Classic examples are waiting for another process to WRITE before a READ can be performed and waiting for another process to release a lock before it can be acquired. A spin lock is not a good solution for this problem because it is basically a loop that will cause the process to be blocked while waiting for the other process to complete the task. This may be subpar/wasteful, and in the worst case, could cause a process to be blocked forever because the other process didn't ever complete the task when it was expected to. In the subpar/wasteful case, a whole processor is devoted to spinning, basically wasting time/resources, as it could be running other processes and performing other tasks in the meantime. A better solution would involve putting the waiting process/thread to sleep and running other ones until the one process/thread completed the task at hand.

10

9. In the context of locks, what is the single acquire protocol? Describe a case in which it can be safely relaxed.

The single acquire protocol mandates that only one process/thread can acquire and "have" the lock at any given point in time - only once that process/thread releases the lock can another acquire it. This protocol ensures before-or-after atomicity for the instructions in the locked block of code, as the whole block will be executed by a process/thread completely before or after other executions of the block of code.

Does safely relaxed mean "relax" as in release the lock or "relax" as in relaxing the entire protocol? I will attempt to answer both cases. If it is dealing with releasing the lock ... it is only safe to do so after the entire block of code in the lock has been executed. For example, only after the variable inside the lock has been read and written to, can the lock be relaxed for another to acquire. If it is dealing with relaxing the whole protocol... In the case that different processes had their own physical memory and were isolated, then this protocol can be relaxed because there is no chance of one process's actions externally affecting another (like changing a variable; this may not be true if they are communicating via messages).

10. Why can locks be correctly implemented using assembly language instructions like Compare and Swap or Test and Set?

Locks can be correctly implemented using Compare and Swap or Test and Set because they can correctly make up the lock mechanism. Compare and Swap first compares the value to be written to with its old value. If it is the same, then it is safe to proceed and change the variable. If it is not, then that means another process/thread has changed the variable, and so the current process/thread must loop and try again. Similarly, in Test and Set, the process gets the old flag value, sets the flag value to TRUE, then returns the old value. If the old value was FALSE, then the current process/thread has successfully acquired the lock. However, if it is TRUE, then another process/thread has the lock. These can correctly implement the lock functionality because the probability of interference during these short sections of code is so small that it becomes negligible. A process/thread will rarely loop more than once or twice before it acquires the lock.

7/8

Zero, actually. There is uninterruptable