Oct. 27, 2015

95/100

Midterm Examination
CS 111
Fall 2015

Name: Michael Wang

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.

mem interface

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.

Being tolerant on inputs is important because inputs are external (e.g., from a program or user) and may be imperfect. That is, a system should be able to handle a wide range of inputs due to user errors, or communication errors that may occur. A system should be strict on outputs so users can expect consistency and receive predictable results. In an OS, a system call like open(file) should be tolerant when given wrong or extraneous inputs, and always return properly, to either indicate success or error.

10

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

If a section of memory is set to copy-on-write (COW), then that section is not duplicated immediately when a fork is performed. That section is only ever copied (to make a copy for the new process) if the new process decides to ever write to that section of memory. If the new process never does, then that section is never copied, saving memory and time.

5

exec issue

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

Emulation is the process of running a program on one system while (the program thinks it's on another) (for example you can use software to execute) hardware instructions (for a program) as if the (program) were running on hardware.

The main challenge in software emulation is to accurately reproduce the behavior of the emulated system while maximizing performance. There is overhead due to the emulator translating the emulated instructions/code to their native counterpart, and optimizing that translation is challenging.

10

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 most overhead, because it is pre-emptive while the others are not. Round robin pre-empts (interrupts) running processes (to give a fair share of CPU time to every process), and this pre-emption is done often, unexpected by the (running) process, and involves context switching, where one process is "swapped out" of the CPU for another. Context switching has a lot of overhead due to saving process state, and picking and scheduling the next process. First come first serve and shortest job first, on the other hand, only change processes (context switch) when the running process completes, so there are fewer context switches and less overhead.

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.

When handling a software trap, a first level trap handler saves the currently running process's state, which includes the CPU registers, program counter, and processor status, onto the kernel stack. The 1st level handler then calls the 2nd level handler depending on which trap was triggered.

10

The advantage of this approach is that the 1st level handler is handles saving the process state (which is the same procedure for every trap), while the 2nd level handler handles the trap itself. This makes it simpler to add a new trap, or change the process-state-saving procedure.

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 the sharing of end result of some action, i.e., two threads both crashing because one thread crashed.

10

Fate sharing is most likely for shared memory, because two processes physically share the memory, which may be corrupted by one or both processes. It is much less likely in RPC because the server is isolated (often physically) from the client, and client/server interaction is through messages passed over the network. It is less likely with messages, too, because sending and receiving is handled by the OS (as is allocation and storage), and processes only send/receive messages from each other through the OS.

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 is a device that can request control of a bus. The disk controller is likely to become a bus master when it needs to perform DMA (direct memory access) transfers to and from RAM. The reason is that this method of data transfer (hard disk ↔ RAM) is more efficient (for large transfers), because it does not require the CPU to supervise the data going across the bus (and rather the controller does the transfer itself). After the DMA transfer is finished, the controller sends an interrupt on the bus to the CPU to indicate completion, after which the CPU can regain control of the bus.

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 issue of coordinating multiple threads, where one group of threads is waiting for another group of threads to complete (because their task length is different). A spin lock for the waiting threads is not a good solution, since it utilizes CPU for the sole purpose of repeatedly checking a condition. It would be more efficient to block (in the waiting thread) and give that CPU time to non-blocking (non-waiting) threads, which would notify blocked thread(s) when they complete.

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 for locks specifies that a lock can only be acquired once (until it is released). A case in which this can be relaxed is if a thread ~~same thread~~ acquires the same lock twice, because there can be no race conditions or sequence coordination issues (since it's a single thread).

18

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

It is possible because they are atomic instructions that perform a combination of common (e.g., MOV, ADD) instructions in one instruction. Compare-and-swap(val, old_val, new_val) only swaps new_val with val if val == old_val, so a lock could be implemented by letting acquire be: c-a-s(lock, unlocked_val, locked_val) and release be: lock = unlocked_val.

Test-and-set(val) sets a value and returns the old value, so lock acquire can be: success if !test-and-set(lock) and release can be: lock = false.

10