57 +9

Midterm Exam
CS 111, Principles of Operating Systems
Fall 2016

Name: Robert Zalog

Student ID Number: 504-578-962

This is a closed book, closed note test. Answer all questions.

Each question should be answered in 1-3 paragraphs. DO NOT simply write everything you remember about the topic of the question. Answer the question that was asked. Extraneous information not related to the answer to the question will not improve your grade and may make it difficult to determine if the pertinent part of your answer is correct.

1. What is the difference between the invalid bit in a page table entry and the invalid bit in a translation lookaside buffer entry? What happens in each case if an address translation attempts to use that entry? Is it possible for both to be set? Why?

The invalid bit in a PTE tells the CPU that the page does not currently reside in main memory, and thus must be swapped in. The OS then decides what page it is going to evict (through some sort of page replacement algorithm), and swaps the page fault occurs ← The CPU traps to the OS, and ← two. Then the CPU retries the accessing the data specified in the PTE, which should now exist.

4

2. In the TLB, the invalid bit could mean that the entry or translation is no longer valid. The OS will then reallocate that entry to hold a different translation.

4

1 Yes, both can be set, if a page somehow gets wiped from memory (eg. process frees a bunch of memory), both the PTE and TLB entries will be invalid.

2. There are many difficult issues that arise due to uncontrolled concurrent executions. Why do we not simply turn off interrupts to prevent such problems from arising? Why not always? Why not just for all critical sections of code?

Turning off all interrupts always would be very bad for a number of reasons. One, the OS would never be able to preempt processes, so they would continue to hog the CPU until they yield (which could be never). Also, processes could miss interrupts they actually care about, like keyboard interrupts. As for why not just critical sections of code, similar problems arise. There would still not be anything stopping a process from just taking over the CPU for as long as it wanted to, which could be very bad in the case of malicious or buggy code (e.g. infinite loops). Also, in general, anything dealing w/ enabling/disabling interrupts on a system is quite slow, adding extra overhead to code that probably doesn't need it.

+2

5

↑ or forgetting to re-enable interrupts.

3. What issues arise in management of thread stacks that do not arise in management of process stacks? Why do these issues arise and what is typically done to handle them? What are the disadvantages of this approach and why is the approach used despite those disadvantages?

The issue that arises in the ~~case~~ management of thread stacks is that you essentially have a bunch of different threads sharing the same process stack, and ~~they can~~ the different stacks can thus run into each other if they grow too large.

(3)

4. In MLFQ scheduling, processes are moved from one scheduler queue to another based on their behavior. Each such queue has a particular length of time slice for all processes in that queue. Why might a process be moved from a queue with a short time slice to a queue with a long time slice? How can the operating system tell that the process should be moved?

The main reason processes are moved down a priority level (from short time slice → long time slice) is because they are constantly using up their full time slice (i.e., never blocking for I/O, etc.). To the OS, this indicates two things. One, that the process is most likely ~~not~~ not interactive and does not need to run so frequently, and two, that the process may appreciate one longer time slice compared to many shorter ones, so as to not keep getting interrupted while working.

+1

(7)

(9?)

5. Will binary buddy allocation suffer from internal fragmentation, external fragmentation, both, or neither? If it does suffer from a form of fragmentation, how badly and why? If it does not suffer from a form of fragmentation, why not?

Binary buddy allocation distributes variable size chunks and suffers from mostly from external fragmentation. ~~Ete the pro~~ while the memory space will be OK for a while, over time more gaps will emerge as the memory manager cannot find chunks in optimal locations to hand out and allocate. It does not really ~~suffer~~ from internal fragmentation unless a process requests a chunk bigger than it is actually going to use.

2 X

6. What is the purpose of a trap table in an operating system? What does it contain? When is it consulted? When is it loaded?

The trap table tells the CPU an address to jump to whenever a ~~user mode~~ program causes an exception/trap. It contains a list of addresses where exception handler code is located. The trap table is consulted whenever a program causes a trap. The trap table is loaded at boot time by the OS (note that creating and modifying the trap table is a privileged instruction, and can only be done in kernel mode).

+2 6 10?

7. Assuming a correct implementation, do spin locks provide correct mutual exclusion? Are they fair? Do they have good performance characteristics? Explain why for all three of these evaluation criteria (correctness, fairness, performance).

Assuming they are correct, then yes, spin locks will provide mutual exclusion. Eventually one thread will see that the lock is available and claim it, which is done atomically so that two threads do not grab the lock at the same time. However, spin locks are neither fair nor perform well. Regarding fairness, there is nothing in place to prevent threads from spinning forever and starving if they are not lucky enough to happen to get a chance at the lock. Regarding performance, spin locks are very inefficient. If a thread gets interrupted while it has the lock, it may have to wait for every other thread to spin for a whole time slice, as the OS schedules spinning threads equally. This obviously wastes a ton of time and makes spin locks extremely unscalable w/ larger numbers of threads.

8. What is the purpose of using a clock algorithm to handle page replacement in a virtual memory system? How does it solve the problem it is intended to address?

The purpose of the clock algorithm for page replacement is intended to act as a "close enough" version of the LRU page replacement algorithm, greatly lessening the amount of page faults, while also executing much faster than the LRU algorithm, saving time whenever we need to decide what page to evict from memory.

It solves the problem of the slow LRU algorithm by only approximating its results. Rather than searching for the single page that has not been used for the longest in the past, it searches for any page that has not been used fairly recently (which it determines through the specific mechanics of the algorithm, which I will not discuss here). In practice, this performs much better than running a true LRU algorithm, while minimizing page faults to basically the same degree.

hardware bit set not mentioned!

9. What are two fundamental problems faced by a user level thread implementation that are not faced by a kernel level thread implementation? Why do these problems arise in user level implementations? Why don't they arise in kernel level implementations?

One problem is that user mode threads cannot actually utilize multiple cores, b/c to the OS, they are all a part of one process, and thus execute on a single core. In kernel level implementations, the OS actually has access to multiple cores, and can be smart about scheduling different threads from the same process to run on multiple cores.

Another problem is mutual exclusion is more difficult to ~~acquire~~ for user-mode threads. B/c they do not have access to things like mutexes or thread waiting queues, they are forced to use spin locks w/ test-and-set or compare-and-swap for mutual exclusion, which can be very slow compared to the kernel-mode alternatives, and less efficient. Of course, kernel mode threads have more efficient and robust alternatives, including mutexes, condition variable, and semaphores.

6

10. What is the difference for a virtual memory system between segmentation and paging? Why might both be used in a single system?

w/ segmentation, segments do not have to be any fixed size, and are instead specified by an upper and lower bounds. Segments are ~~distributed~~ determined when a process starts up, but cannot expand very easily (if at all) if the process decides it needs more memory. Also, segmentation results in lots of internal fragmentation (average 50%). Also, segments have a very long time swapping from RAM to disk, and cause a lot of external fragmentation. In contrast, paging distributes fixed size pages to a process as needed (i.e. dynamically). Once a process fills up its space, it can simply get more pages allocated for it. Pages are not specified by bounds in the OS, but rather by PFNs (which virtual addresses can be translated to). Paging also suffers from very little internal fragmentation, no external fragmentation, and allows easy swapping on/off disk.

10 ↑ +2

6

this way a process has specific bounds on its segments, when can cause exceptions if stepped over (eg. seg. faults).

In a system, a process' memory space might be divided into large segments (e.g., the code, data, and stack segments), which can then be divided further into pages, allowing easy expansion if need be.