63

Midterm Exam
CS 111, Principles of Operating Systems
Fall 2016

Name: _____

Student: _____

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.

(5) (1)

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 page table entry indicates that the page has been swapped out to the disk's swap space, and that the entry is not currently in memory.

In the translation lookaside buffer, the invalid bit indicates that the entry is not usable in that context and is not allowed to be used, i.e. it belongs to another process and is not shared.

If an address translation attempts to use the invalid page table entry, then a pagefault is generated, and the fault handler is called to retrieve the page from swap space and copy it in, replacing a page if necessary by the page replacement policy specified. On the other hand, if a address translation attempts to use an invalid TLB entry, a segmentation fault is generated. ✗ page table lookup

It is not possible for both to be set because a TLB entry must request to be in the TLB from the page table. If invalid in the page table, it must first be retrieved from disk first before it can be put in the TLB. ✓

(1)

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?

For starters, turning off interrupts is a privileged instruction, and cannot be used by user-mode code. Even if it could, security issues could arise – it is not difficult to imagine a piece of malicious code which would use interrupt disables to monopolize the CPU.

Secondly, turning off interrupts, even just during critical sections of code, does not ensure mutual exclusion. For example, if a CPU has multiple processors, just because interrupts are disabled on one does not ensure a concurrent operation on another processor will not introduce a race condition. And "turning off" all processors but one is obviously not feasible, as it makes having more cores/a more powerful computer pointless.

(5)

We don't want to turn off interruptions always either, or we could have starvation for processes that wish to run after another. Such a policy would be unfair and produce terrible turnaround times, and interactive programs would be potentially unbearably slow. Furthermore, without preemption, we eliminate the benefits of concurrent executions anyway if we "always" turn off interruptions.

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?

One problem that thread stacks face that process stacks do not is that while process stacks are allowed to grow virtually indefinitely (of course, "virtually" is the keyword), thread stacks are confined within process data space and must have their stack size statically set. In order to handle them, when declared, threads are required to give their stack size. The disadvantages of such static-stack size declaration is that stack size is limited, and if the thread runs out of stack space it's out of luck. However, because the programmer has control over thread creation, it is hoped that the programmer will evaluate what stack size is needed and allocate appropriate

(7)

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?

(6)

In MLFQ scheduling, processes are dynamically moved from one queue to another based on their behavior. One way the OS can figure out whether a process should be moved is based on how often the process reaches the end of their time slice without giving up the processor. If a process continuously uses up its time slice, this may be an indication to the OS that this process needs to be moved from a queue with a short time slice to one with a longer time slice.

If a process continuously yields before reaching the end of its time slice, this is an indication to the OS it may need to be moved to a queue with a shorter time slice.

less i/o operations also.

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?

(5)

Buddy allocation likely will suffer from internal fragmentation. Buddy is good at handling limited but efficient coalescing, so external fragmentation is less of an issue.

Buddy allocation can suffer fairly from internal fragmentation, but the benefit of this style of allocation is that internal fragmentation in a block will never be 50% or greater. This is because buddy allocation allocates blocks of size 2^n. However, internal fragmentation can still be serious. Imagine processes request a great deal of, for example, 33 KB blocks. In this case the buddy allocator has to provide 64 KB blocks for each of these requests. That's a lot of memory wasted.

External?

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 in the OS is a part of the OS's 1st level handler to handle interrupts/traps, that holds information on how the trap should be handled. With the program counter and processor status registers of the interrupted program, the trap table will index into what trap handle should be run, and pass this information to the system call handler.

It is consulted when a program calls an interrupt, and is loaded by the OS at startup time. It contains the PC/PS that must be loaded by the system call handler.

(8)

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).

(7)

Assuming correct implementation, spinlocks do provide mutual exclusion. In this regard, spinlocks do have good correctness. This is because spinlocks use an atomic, hardware-supported operation to check whether it may enter a critical section. If not, it spins until it gets a chance to take the lock. The use of an atomic hardware-assisted operation enforces correctness when implemented properly.

Spinlocks are not really fair. They are fair in that every thread will likely get a chance to run (depending on scheduling policy), but most threads will not be doing any thoughput if only one thread has control of the lock, and this thread will likely be starved for a long time by spinning threads before it can run.

On the same note, spinlocks yield terrible performance. Since spinning threads do not yield, they spend their entire time slice spinning and likely blocking the actual thread that can do work with fewer threads than processors. Spinning can be a useful behavior. Otherwise, most CPU cycles are wasted spinning.

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 problem with LRU, despite it being an optimal approach in theory, is that the question of implementing it is much more difficult. How do we record a timestamp for every entry in the page table? The MMU is meant to be fast, so writing such data seems to be counterintuitive and a performance boon. At most, the MMU can keep an "accessed" bit, but not relative time.

The second problem is that the cost of going through the entire list of entries is not conducive to good performance and is in fact counterintuitive to the good performance we are seeking.

The clock algorithm solves these problems while maintaining near-LRU good performance. With the clock algorithm, the MMU need only maintain the "accessed" bit mentioned above. The clock algorithm will simply flush that bit, if assigned, if it finds it while looking for a page to remove, and skip over that page. Also, the clock algorithm solves the problem of looking through the whole list. By simply flushing the first page without the accessed bit set and setting the pointer to start looking at this entry in the circular list, we eliminate the performance boon of having to look through the entire list while achieving similar (but not the same) results to LRU.

(8)

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? (5)

First of all, user-level thread implementations are in fact limited to the same non-privileged instruction set as simply another user process on the system. This means limitation to one processor. Indeed, user-level thread implementation cannot make use of multiprocessor benefits as kernel-level threads can. Kernel-level threads can run for multiple processors without being confined to one core.

Another issue faced by user-level thread implementation is complexity. While kernel-level thread implementation is largely kept "under-the-hood," in order to implement threads on the user-side, threads must be managed by the programmer and thus give arise to a much higher degree of complexity, which is something that we want to avoid. With kernel-level threads, we can just rely on the standard implementation given by the OS while bypassing this complexity.

2) thread blocking

10. What is the difference for a virtual memory system between segmentation and paging? Why might both be used in a single system? (4)

Segmentation is the virtualization of a process into "segments" including the code segment and the data segment, which includes the heap, stack, BSS, etcetera. Segmentation requires the OS to allocate blocks of physical memory corresponding to virtual memory of the process, and uses hardware registers to perform the rapid translation from virtual-to-physical. These blocks may be of fixed or variable size.

On the other hand, paging is the division of memory into "pages" of a set size, which may be used to virtualize memory greater than in physical memory by swapping pages out to disk and using a quick TLB in the MMU to achieve good performance while translating many addresses.

Both may be used by a single system because it is advantageous to have processes allocate their memory in blocks, so that multiple processes can share memory and so processes can differentiate their code, heap, and stack segments (and prevent illegal operations into these segments). However, using paging with segmentation not only allows virtualization of a larger memory space for each of these segments, but also ensures that these segments do not need to be contiguous in memory, providing huge benefits in terms of fragmentation.