Midterm Exam
CS 111, Principles of Operating Systems
Fall 2016

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 what was asked. Extraneous information not related to the answer to the question will not improve your grade.

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 the page table entry is set if the owning process does not have a page at that address allocated. Thus, an attempt to translate that address will lead to an exception. The invalid bit in a translation lookaside buffer entry is set if that TLB entry is not currently valid. Translating the address in question may still go ahead, but the actual page table, rather than the translation lookaside buffer entry, must be consulted. Both could be set. The TLB entry invalid bit might be set because the TLB entry refers to a different process' address space, while the page table entry invalid bit might be set because the process has not allocated that page.

Reference: Arpaci-Dusseau chapter 19, page 8
(http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf)

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?

First, on multicore machines and other machines with genuine parallel operations (such as symmetric multiprocessors), concurrent execution can occur even when interrupts are turned off. Also, many system activities require interrupts to proceed properly, such as handling I/O events and performing preemptive scheduling. If interrupts are always off, we cannot run a preemptive scheduler and we cannot guarantee that I/O events like keystrokes, mouse movements, message arrivals, or disk reads will be handled in a timely manner. Some interrupts may be lost while interrupts are disabled. As a result, the system may run very slowly and provide poor responsiveness if interrupts are kept off for a long time. Thus, running with interrupts turned off always is not desirable.

Turning interrupts off for critical section is less harmful, but unless the critical sections are guaranteed to be short, the same undesirable effects can occur. Again, merely disabling interrupts won't prevent all concurrent operations on multicore machines. Finally, turning interrupts off and on is a privileged operation and cannot be performed by user code, unless you provide a system call for that purpose.

Reference: Arpaci-Dusseau chapter 28, page 4 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf), Lecture 7 pages 47-51 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_7.pdf)

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?

A process has its own virtual memory space and only one stack. This stack can be placed at one end of the essentially one-dimensional memory and allowed to grow towards the other end, while the data segment can be located at that other end and grow in the opposite direction. Unless all memory space is used up, they will not run into each other. But threads of a single process share the same address space. Since the address space is still one dimensional, multiple thread stacks cannot all grow towards a single "hole" in the middle of the address space. Therefore, thread stacks might run into each other or other memory segments if they grow too large. The typical solution is to place thread stacks at one place in the virtual address space and limit their maximum size. This solution implies that a thread cannot make deeply nested function calls, as each will use up part of its stack space, and too many will overflow its allocated space. The approach is used largely because there are few alternatives and most threads tend to be relatively simple, with little likelihood of making deep sequences of function calls. So they don't need arbitrarily growing stack sizes.

References: Arpaci-Dusseau chapter 26 pages 1-2 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf), Lecture 7 slides 9-11 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_7.pdf)

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 long time slice? How can the operating system tell that it should be moved?

MLFQ scheduling tries to find a good time slice for each process, where "good" is defined as a time slice that is suited for the process' needs. Processes that perform a lot of I/O need short time slices, since they will probably block on I/O after running for a short time. They will need to be scheduled fairly often, though, so their I/O is handled promptly once it is ready. Processes that perform a lot of computation and little I/O would work best with long time slices, since that will allow them to get much work done before they are context switched out due to time slice expiration. But if we allow them to run for such long slices very often, they will hog the processor. So they get scheduled for longer slices, but less often. A process would move from the short time slice queue to the long time slice queue if it appeared to require more processing time than it got on the short time slice queue. That would be detected by the process frequently running to the end of its short time slice and rarely blocking on I/O requests. The OS can simply count how often the process finishes executing for each reason and move the process to the longer time slice queue if it usually uses up its entire short time slices.

References: Arpaci-Dusseau chapter 8 pages 2-4 (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf), lecture 4 slides 50-52 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_4.pdf)

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?fragmentation, external fragmentation, or both? How badly? Why?

Binary buddy allocation starts with all of free memory in one segment. On request of n bytes, it looks for a free segment of at least n bytes, but no more than 2n. If it has no such segment, it halves the next largest segment and continues halving one of the segments until it has produced a segment in the desired range. That segment is given to the requestor. Since the possible segment sizes are fixed, many allocations will be met with somewhat larger segments, resulting in internal fragmentation. The internal fragmentation will always be less than 50%, since if a segment more than twice the size of the request is available, it will be split in half before granting the request.

External fragmentation is also possible, since segments keep getting split into two smaller segments. As a result, we could end up with lots of very small segments scattered through memory, totaling more than N bytes, but with none of them large enough to fulfill a request for N bytes. That would be external fragmentation. It might not be too bad, since one only splits a segment when one is asked for a segment of a particular size. Thus, while one could have small segments, they are likely to be usable, since segments of those sizes were usable before.

References: Arpaci-Dusseau chapter 17 pages 14-15 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf)

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 is used to specify what actions the operating system should take when a particular exception occurs. It will contain pointers to exception handling routines, indexed by a number specifying the particular exception that occurs. It is consulted whenever an exception actually occurs. It is loaded at boot time and typically will not change until the next reboot.

References:Arpaci-Dusseau chapter 6 pages 4-5 (http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf), lecture 3 slides 45-47 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_3.pdf)

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).

Correctly implemented spin locks do provide correct mutual exclusion. The holder of the lock is permitted to access the critical section and no one else is. When the lock holder releases the lock, if any other process is spinning on that lock, one of the spinning

processes will obtain the lock. So not only will we prevent multiple processes from simultaneously accessing the critical section, but we will ensure that if anyone wants to access the critical section, someone will be able to.

Spin locks are not fair. They give no guarantee that a process spinning on a lock will ever obtain the lock. If multiple processes are all spinning on the lock, only one will be the next to get it. That isn't necessarily the one that has waited longest, and there is no way to ensure that a process' waiting time while spinning is bounded.

Spin lock performance is poor. Spinning burns cycles, so instructions are being spent merely to determine if the lock can be obtained. Further, if the process holding the lock is preempted by someone spinning on the lock, no progress will be made until the process holding the lock is re-scheduled. This is worst on single core systems. On multi-core systems, it's not quite as bad, since the lock-holding process is less likely to be preempted by spinning processes. If the critical section is short, the spinning process might not need to wait long and preemptions might not occur.

References : Arpaci-Dusseau chapter 28 page 9 (http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf), Lecture 8 slide 23 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_8.pdf)

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?

A clock algorithm is intended to approximate an LRU page replacement algorithm without incurring the costs of true LRU. In true LRU, every page access requires saving a clock value of the time of that access, and choosing the least recently used page requires checking the access time values for all pages. (Alternately, one can keep an ordered list of pages by access time, but that requires updating this list on each page access, which is even more expensive than searching the access time values on pages.)

A clock algorithm avoids keeping the timestamp by instead having a single bit that is set when a page is accessed. The bit is set in hardware on each access, so there is little extra cost for this level of record keeping. The clock algorithm avoids the cost of checking all pages to choose the least recently used by simply choosing the first page it checks where this bit is not set.

References:Arpaci-Dusseau Chapter 22 pages 12-13, lecture 6 slides 41-43 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_6.pdf)

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?

User level threads generally cannot take advantage of multicore or other multiprocessor systems. The OS schedules the process containing the user level threads on one processor, not knowing that the process contains multiple potentially parallelizable threads. In kernel level thread implementations, the OS is aware of the existence of

multiple threads and can choose to schedule them on more than one processor simultenously.

User level thread implementations block all threads in the process if any of the threads block, even though the other threads might be able to run properly. From the OS's perspective, the blocking request applies to the entire process, since it does not know that the process contains multiple threads of control. In kernel level thread implementations, the blocking request was issued by a particular thread known to the kernel, so it can block just that thread and allow other threads in the same process to continue execution.

References: Web page on user mode threads (http://lasr.cs.ucla.edu/classes/111_fall16/readings/user_threads.html)

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

Segmentation allows specification of arbitrarily sized ranges of the address space that are valid and are used for a particular purpose, such as holding the process' code or stack. Paging is used to divide allocated memory space into smaller pieces that allow the virtual memory management system to load, relocate, and otherwise manage the space more flexibly.

Segmentation and paging might be used together to both specify the portions of the theoretical virtual memory space that a process is actually to use while allowing more flexible management of RAM use within those portions. Segmentation would specify the set of address ranges that are legally addressable, while paging would allow the OS to locate small sections of those address ranges in arbitrary page frames, or to move them off to disk, as necessary.

References: Arpaci-Dusseau chapter 16 pages 1-4 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-segmentation.pdf) and Arpaci-Dusseau Chapter 18 pages 1-5 (http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf), lecture 5 slides 58-62 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_5.pdf) and lecture 6 slides 6-9 (http://www.lasr.cs.ucla.edu/classes/111_fall16/slides/Lecture_6.pdf)