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?

An invalid bit in the page table entry indicates that the page indexed by the virtual page number is not a valid page in that process's address space. An invalid bit in the translation lookaside buffer means that the particular translation of the virtual page to physical page is not in the set of fast registers that make up the TLB (a cache). If an address translation attempts to use the entry in the page table, the process will trap into the OS and receive a segmentation fault. If an address translation attempts to use that entry in the TLB, we will get a TLB miss. This means that the translation is not in the TLB, so we must trap into the OS to retrieve the translation from the page table, or on disk if the page has been swapped out from the page table. It is possible for both to be set in the case that the page is not a valid page in the process's address space. The page table would indicate this by setting the invalid bit, and since this translation cannot be in the TLB, the invalid bit in the TLB would be set as well.

8 +2

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 interrupts does not work for all cases of uncontrolled executions and there are dangerous effects that can occur by disabling interrupts. First, if we always turn off interrupts, a piggy process can take control of the CPU and starve all other processes, or a buggy process can freeze the system by going into an infinite loop. By turning off interrupts always, we put too much trust in the processes to yield and thus share the CPU. Also, turning off interrupts for too long can lead to them becoming permanently disabled. Furthermore, if we turn off interrupts always, we can slow down processes that need certain resources and can only continue after the interrupt has been handled. Even if we turn off interrupts for all critical sections of code, this does not fix the issue for multicore processors. Threads can run of instructions on different cores of processor simultaneously, so disabling interrupts has no effect on controlling their concurrent execution.

5 +2

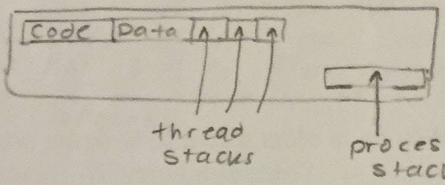
preemptive, i/o?

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?

In the management of thread stacks, the sizes of the stacks are not as easily and readily changeable. Process stacks can grow fairly large due to the free space between the stack and heap. However, because there ~~are~~ may be multiple threads in a process, thread stacks cannot grow as easily since they all grow towards a single hole. To handle these issues, the thread stacks are placed after the data segment such that they grow in the opposite direction. The disadvantage of this approach is the fact that the thread stacks may grow and collide with the process stack. However, this approach is used despite the disadvantage because thread stacks are usually not too large.

6

fixed sized stacks.



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?

If a process is moved from a queue with a short time slice to a queue with a long time slice, then the process ran during the entire duration of the short time slice and was blocked at the end. This means that the process is a long-running process rather than a more interactive process. Therefore, once the OS sees that the process ran during the entire time slice without yielding, it decides to move the process to a queue with a longer time slice in order to increase its throughput and reduce the amount the context switches. Since the process is not interactive, it does not need to be blocked as often and should be given more time to run during each time slice.

x2

8

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 suffers from internal fragmentation. For binary buddy allocation, we divide up memory into sizes that are powers of two and allocate chunks to processes. We start with memory of size n and divide it into two pieces of size n/2. We then divide each of the two pieces into two pieces, thus having a total of four pieces of size $n/2^2 = n/4$. This continues until the pieces cannot be divided any more. When a process requests for memory, this allocation method determines the first chunk of size $n/2^x$ (where x is zero or a positive integer) whose size is greater than or equal to the requested amount of memory. Therefore binary buddy allocation can suffer from internal fragmentation, only if the requested chunk is not a size that is a power of 2. If the size requested is a power of 2, there is no internal fragmentation since the all of the memory in the allocated chunk is used. If the size is not a power of 2, then the allocated chunk is of size greater than the requested amount, so some memory is wasted. Binary buddy allocation does not suffer from external fragmentation because the chunks are not carved out, so there is no wasted space between the address space of each process.

5

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?

In an operating system, a trap table is used to handle errors during process execution or system calls, both of which require privileged instructions performed by OS intervention. When a process has some exception (ex: ^C, segmentation fault) or a process uses a system call, the process blocks and traps into the OS. The trap is used to index into the trap table, which contains the PC and PS. The trap table is loaded during boot time.

9

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).

Correctness: Spin locks correctly provide mutual exclusion. When a thread tries to acquire a free lock, it is able to acquire it and enter the critical section. When another thread attempts to acquire this same lock, it will begin spinning until the lock becomes free. Only at that point can this thread enter the critical section. Therefore, spin locks prevent multiple threads from entering the critical section and allow one thread in the critical section at a time. Thus, spin locks do correctly provide mutual exclusion.

[redacted] 3 +3 7

Fairness: Spin locks are not necessarily fair in all cases. If a thread acquires a lock and never releases it, another thread can endlessly spin and never acquire the lock. However, in cases that all threads eventually release the lock this approach is fair since a thread acquires a lock once it is free and thus stops spinning. → random

[redacted] 1

Performance: A spin lock is very costly, so the performance is not ideal. When a thread is waiting for an already acquired lock, it continues spinning and spinning until the lock is free. If the lock remains acquired for a long period of time, the thread will spin for a while, thus wasting CPU cycles. Therefore there is a lot of overhead with using spin locks.

[redacted] 3

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 used to determine the least recently used page so that it can be swapped out. Each page is given a reference bit, which is set to 1 when the page is accessed and otherwise set to 0. When a page needs to be replaced, the clock algorithm starts at page P and checks the reference bit. If it is set to 1, the reference bit is changed to 0 and the clock checks the next page (in a circular fashion). Once it finds a page with a reference bit of 0, this page is swapped out and the clock remains pointing to this page so that the algorithm can start checking from this page the next time a page needs to be swapped out. This algorithm solves ~~the~~ problem of modeling the LRU page replacement algorithm efficiently. Instead of noting the time each page was referenced and checking all of the time entries every time a page is replaced (which is costly), we only need to keep a reference bit on each page and check pages only until we find a page with this bit set as 0. Thus, we accurately model LRU in a more efficient manner.

8 [redacted]

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?

A user level thread implementation cannot use privileged instructions while a kernel level thread implementation can use privileged instructions. This problem arises because a user level thread is limited to only instructions that can be performed in user mode, however, this problem does not arise in kernel level thread implementation since we are able to access privileged mode. Another problem is the fact that user level threads must yield while kernel level threads can be preempted by the OS. This occurs because the OS kn

(see back) 6

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

Segmentation is the idea of providing a virtual-to-physical translation of contiguous memory segments, whereas paging performs a virtual-to-physical translation of pages. In the case of segmentation, each segment of memory is contiguous, and there is an associated pair of base and bounds registers for each segment. The base register holds the virtual address of the start of the segment and the bounds register indicates the maximum size of the segment. Using these virtual addresses, we can translate from the virtual address space to the physical address space where the segment actually resides in memory. In the case of paging, these individual segments are broken into small fixed sized blocks, or pages. We can perform a virtual to physical translation using the virtual page number to index into the virtual table and determine the physical page number. This physical page number, along with the offset specified by the virtual page, can be used to retrieve the physical page. Both of these may not be used in a single system because segmentation limits us to keeping each segment of memory within contiguous addresses; however paging allows us to break up these segments and relocate the pages into different areas of memory.

6

X

9. In a user level thread implementation, when a thread is blocked, all threads in that process is blocked. This occurs because the OS does not know of the existence of user level threads. However, since the OS knows of the existence of kernel threads, when a thread blocks, the remaining threads can continue running.

Another problem is that user level threads must yield the CPU while kernel level threads can be preempted by the OS. Again, the OS does not know of the ~~ex~~ existence of user level threads, so it cannot preempt or block them, unlike kernel level threads. So, user level threads must yield in order to give up the CPU.

↪ multiprocessor!