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 is set when a page is assigned to part of the process memory that is not used, like the region between stack and heap. The invalid bit in a TLB entry is set when that entry does not contain a valid address to page mapping yet, a different characteristic than PTEs.

If an address translation tries to use an entry with invalid bit, it should replace that page or free it up for a process that needs more pages. If an address translation sees an entry in the TLB that is invalid, the OS tries to fill the entry with a valid address to page mapping that exists, usually from the page table.

Both bits can be set, but not for the same page. An invalid TLB entry does not point to any page in the page table, which might have its invalid bit set. TLB entries are only invalid when process running is started (so new page data is inserted into empty entries) or when page entries are removed/replaced from the TLB.

8

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?

Interrupt disabling causes a lot of problems itself. Only kernel level code can use it, since it is privileged, so interrupt disabling does not apply to user mode code. Interrupt disabling also does not even ensure protected sections with parallel processing and multiple cores. If there are any bugs or a critical section takes longer to run than usual, the code will not respond to interrupts that might be important for the rest of the system to continue running smoothly; scheduler cannot preempt the process, and any interrupts from completed disk I/O is lost; longer critical sections are not suited for interrupt disabling.

7

pre-emptive

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?

Each thread has its own private stack, though other parts of the process address space are shared. One issue that could arise is where the thread stacks are stored in the process memory; if they are stored close together, the stacks do not have much space to grow if more functions are called. This shouldn't be a problem though, since stacks grow most with heavy recursion, which threads probably won't do.

Another problem that might happen is updating of information between thread stacks?

→ This doesn't happen for processes since each process only has 1 stack to grow in a certain direction. The issue is figuring out where to store thread stacks in 1 process's mem so they can all be used well.

6

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?

All processes start off with highest priority with MLFQ, which means that they are given the shortest time slices on the assumption that they will be completed quickly. If a process uses a lot of time slices or total CPU runtime cumulatively, it is assumed to be a longer runtime process, its priority decreases and it is assigned to a queue with longer time slices. This both allows short processes to be run first, while not making long processes wait too long, and decreases amount of context switches that normally occur with short time slices given to long processes.

by OS

6

less i/o

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 is when a large chunk of memory is split in halves until a smaller piece can be allocated to a process. It suffers slightly from internal fragmentation; if a process asks for a chunk of memory that is not the same size as any of the halves from splitting, it is allocated memory with some extra space that is not used. BB allocation does not suffer much from external fragmentation, since all memory chunks should stay aligned nicely because they are multiples of each other by factors of 2, but it is possible that external fragmentation still occurs if almost all memory is taken and the chunks left are too small for most processes to use.

7

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?

A trap table handles trap requests to the OS, whenever processes need to give the OS control because of behavior constraints (privileged instructions needed or illegal behavior attempted). It contains a mapping of signals and calls to the correct trap handler that executes the right behavior, so that programs do not know how to access specific traps directly. Whenever user mode or hardware traps to the OS, the table is checked for what instructions to run. It is loaded at boot time, when the OS starts up.

8

The 1st trap handler looks at the trap table to call the 2nd trap handler for the right behavior, after saving process memory and switching mode.

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).

Spin locks are correct in actually giving mutual exclusion to critical sections. When one thread has the lock, all other threads must spin until the lock is freed (the value is changed to "free"). This works even on multiple cores, since there is only one lock value for that section that needs it.

Spin locks are not specifically fair, since they can lead to starvation of threads. When a lock is freed, all the threads spinning for the lock (and there can be hundreds) compete for the lock; some might get it more often by chance, while others might never get it by bad luck and go back to spinning, possibly for a very long time.

8

Spin locks usually have bad performance, since the spinning itself requires CPU control and wastes CPU cycles. They can even delay the releasing of a lock since the thread with the lock might be preempted for other threads to spin, to check if the lock is free; a waste of time. However, spin locks are better on multiple cores, with threads spinning and critical section of thread with lock running at same time.

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?

Clock algorithm is a way to approximate Least Recently Used algorithm for page replacement, which replaces the page that was least recently used in memory (if more pages need to be put into memory). However, this requires the page table to save all the times of page access, and retrieve or update them really quickly, which is expensive with many pages. The clock algorithm just cycles through pages and replaces ones that have not been used since it last checked, if needed; it's not perfect, but it requires much less memory and searching of last access times (also a lot of work) than true LRU. This also works with working set algorithm; the clock allocates least recently used pages from one process to another, minimizing page faults (trapping to OS is expensive) while not using much memory or CPU time to check pages.

15

All page algorithms are designed to reduce page replacements and faults; clock specifically improves on true LRU.

hw bit

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 thread implementation does not have as much free functionality as kernel level threads, and are more susceptible to mismanagement such as race conditions.

At the kernel level, privileged operations can be directly accessed without needing to trap to the OS, and since the kernel is trusted, these operations can actually be run if necessary without limits. User level threads cannot do that; must wait for kernel for any privileged functionality, and might be denied anyway.

At kernel level, it is also easier to ensure atomicity of instructions, by using short assembly instructions. User level thread instructions are usually translated into several assembly ones, leading to indeterminate behavior if the instructions are executed in wrong order vs other threads. Since other threads can interrupt in user mode easily, locks are needed to protect critical sections.

2

multiprocessor thread blocking issues

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 when processes are split into separate segments, such as code, stack, and heap, and placed in different areas in virtual memory. This removes much of external fragmentation and allows sections to grow (stack and heap need this), possibly in both directions of memory.

Paging is the system of giving processes set sized virtual chunks that can be placed in physical frames in real memory. This removes external fragmentation completely, and most of internal fragmentation (only parts of 1 page might be unused).

7

Both might be used if a process changes in size a lot, quickly. Allocating pages into memory is expensive, since a TLB is looked at, then the page table if not present, then the swap space in disk if not present in memory, which leads to traps and I/O reading. Segmentation allows sections like the stack to grow quickly as long as no barriers/boundaries are reached. Paging on the other hand is more effective in memory usage, because of its removal of nearly all fragmentation. Segmentation might still have external fragmentation between segments, chunks that are too small to assign to any process. Relocation can solve this problem but is expensive (with memory rewrites, virtual to physical translations, etc.).