
SOLUTIONS TO CS 354 MIDTERM, SPRING 2013 (PARK)

P1(a) 12 pts

Disabling interrupts guarantees that a system call runs uninterrupted which ensures integrity of shared kernel data structures through mutual exclusion.

4 pts

create() has to update the kernel's process table. Without interrupt disabling, the process table may get corrupted if two processes executing create() try to access it concurrently.

4 pts

Interrupt disabling is a drastic action that can prevent other important events, such as handling packets arriving at a network interface card and clock interrupts, from being processed by the kernel in a timely manner.

4 pts

P1(b) 12 pts

Three hardware features: privileged/non-privileged instruction set, kernel/user mode, memory protection.

6 pts

System calls (i) change a process's status from user mode to kernel mode, (ii) branch to a specific kernel function to handle the user request in kernel space. The kernel function may include privileged instructions for accessing shared hardware resources which can now be executed without triggering a fault. When a system call returns the process is switched back to user mode.

6 pts

P1(c) 12 pts

Four pieces: the caller's frame pointer (ebp), flags register bits, 8 general purpose registers, stack pointer.

6 pts

The stack pointer of a context-switched out process needs to be easily accessible when context-switching in the process at a later time. The process table serves this purpose well.

3 pts

Accessing the process table entails several instructions for memory indirection. Using hardware supplied instructions (push/pop variety for stack manipulation) is more efficient.

3 pts

P2(a) 16 pts

Processes that make blocking system calls and give up CPU voluntarily are treated as I/O-intensive and their priority is increased while their time slice is decreased. Processes that deplete their time slice and are preempted by the clock interrupt handler are treated as CPU-intensive processes and their priority is decreased and their time slice increased.

6 pts

The rationale is to give I/O-intensive processes preference over CPU-intensive processes when they are in ready state, since by their nature, they consume less CPU cycles. A small time slice is given to I/O-intensive processes, just in case a CPU-intensive process was misclassified.

6 pts

A potential issue is that for real-world processes that are a mix -- exhibit both CPU- and I/O-intensive behavior over time -- how they fare relative to CPU- and I/O-intensive processes is less clear.

2 pts

Solaris monitors how long a ready process has not received CPU cycles, and if this time period exceeds a threshold, its priority is increased.
2 pts

P2(b) 16 pts

Pro: Thread creation and management does not entail system calls which tend to be slow.
6 pts

Cons: (i) User space threads cannot make blocking system calls (e.g., for I/O) since a single thread blocking ends up blocking all user space threads. (ii) Multiple CPUs (or cores) cannot be utilized.
8 pts

Due to (ii), multithreading with kernel support is more suited for multicore processors.
2 pts

P3(a) 16 pts

Xinu's default scheduler requires linear overhead (i.e., $O(n)$ where n is the number of processes) since inserting a process into a priority queue, in the worst case, requires traversing the whole list.
6 pts

With k priority levels, where k is treated as a constant: (i) Dequeue requires finding the highest priority level at which there are one or more ready processes enqueued. At worst, this requires k comparisons. After finding the highest non-empty priority level, we can pick the first process (constant cost) since all processes at the same level are serviced round-robin. (ii) Enqueue requires going to a process's priority level (constant since multi-level feedback queue is an array of linked lists indexed by priority), which is constant, then inserting at the end of the list (also constant) by round-robin property.
10 pts

P3(b) 16 pts

`tset` does not disable interrupts, thus important events such as packet arrivals can continue to be processed by a kernel's interrupt handling routines.
6 pts

`tset` wastes CPU cycles by waiting on `tset` to return false in an infinite loop (busy waiting). A process that uses semaphores does not busy wait but gets blocked (and context-switched out) by the kernel which avoids wasting CPU cycles.
6 pts

Yes, semaphore calls `wait()` and `signal()`, internally, must disable interrupts to ensure that their code is run atomically. Since the codes of `wait()` and `signal()` are relatively short, the disruption to kernel due to interrupt disabling is kept small.
4 pts

Bonus 10 pts

Without a kernel stack, two processes A and B that share memory where their run-time stacks reside can collude and run their own code in kernel mode. For example, A makes a blocking system call, say, `read()`. `read()` may entail internal kernel function calls which are managed (push/pop) using A's user space run-time stack. `read()` blocks and A is context switched out while in kernel mode. Process B is scheduled next, B has access to A's run-time stack, overwrites one of the return addresses (EIP) of `read()`'s callees so that it points to B's own code. When A

unblocks and is context switched in again, returning from read()'s nested function calls will cause a jump to B's code while A is still in kernel mode.

This study resource was
shared via CourseHero.com