CS 503 (Park) **Midterm** **Oct. 11 (Thu.), 2018**

Remarks: Keep the answers compact, yet precise and to-the-point. Long-winded answers that do not address the key points are of limited value. Binary answers that give little indication of understanding are not good either. Time is not meant to be plentiful. Make sure not to get bogged down on a single problem.

PROBLEM 1 (52 pts)

(a) A multi-level feedback queue is a popular data structure that facilitates $O(1)$ scheduling overhead in operating systems. Explain why time complexity is constant. Fair scheduling such as Linux's CFS incurs logarithmic overhead. Why can CFS not be implemented in constant time?

(b) In Problem 4 of lab2, you were asked to re-implement XINU's getpid() utilizing x86 software interrupt by executing `int 33` within a wrapper function igetpid() instead of the plain old C function call getpid(). How does the former differ from the latter (which is determined by gcc following CDECL)? Point out the main differences. Did a mode switch (i.e., user to kernel mode) occur in either implementation? What about a stack switch? Which implementation incurred more overhead?

(c) Suppose a real-time kernel is presented with two tasks to schedule: $p_1 = (2, 5)$ and $p_2 = (4, 7)$. Draw a time evolution of under RMS. Do the same for EDF. Are both able to schedule the tasks? Would the two tasks be admitted under EDF? What about RMS? From a practical perspective, should they be admitted by a real-time kernel?

(d) When discussing heavy-weight and light-weight process models, we noted that XINU, as a whole, may be viewed as a single heavy-weight process with multiple light-weight processes. In what sense was this comparison meaningful? When context switching between XINU processes, what pieces of information were saved on the stack? Why was it not necessary to save EIP? What is an alternative option to using the run-time stack to checkpoint a process's state? XINU's ctxsw() function did not pop information saved on the stack in the exact reverse order of how information was pushed. What was the reason? Was it necessary?

PROBLEM 2 (32 pts)

(a) What advantage, if any, does asynchronous IPC with callback function for receiving messages have over synchronous nonblocking receive? When coding a callback function, an app programmer has a choice: either make a blocking or nonblocking message receive call. Given that a registered callback function is invoked only after a message has arrived, why is making a blocking message receive call not always sufficient? Why is isolation/protection an issue when implementing kernel support for asynchronous IPC with callback function? What is a solution that preserves isolation/protection? What is a simple solution in the case of XINU?

(b) x86 implements segmentation-based addressing where CS, DS, and SS registers indexing into GDT play an important role. What technique is used by XINU (and similarly for Linux/Windows) to neutralize or disable segmentation addressing in software? What is a minimum number of entries in GDT needed to facilitate user mode/kernel mode operation in Linux/Windows? How many entries are installed in XINU's GDT? Are all of them necessary? Why does XINU's ctxsw() not save the content of CS, DS, and SS onto the stack of a process being context switched out? Should context switching in Linux/Windows save the content of these registers?

PROBLEM 3 (16 pts)

For what types of application processes does full virtualization run close to bare metal speed with minimal overhead? In what scenarios does trap-and-emulation overhead become significant? In full virtualization a guest kernel runs in user mode. That is, even if a process in a guest kernel makes a system call and traps to kernel mode, the true (i.e., hardware) state of the process remains in user mode. Why is it necessary for a hypervisor to track if a process running in a guest kernel "thinks" it is running in user mode or kernel mode? Use the `cli` instruction in x86 as an example. Why do sensitive instructions such as popf render x86 ill-suited for full virtualization? In a simple redesign of x86, what minimal changes are sufficient to make it suited for implementing full virtualization?

BONUS PROBLEM (10 pts)

The Solaris TS scheduler aims to achieve "fair" allocation of CPU cycles to processes by promoting/demoting priorities dynamically based on their behavior. What is aging, and why is it needed for the Solaris TS scheduler? How is aging related to a problem of lab2?