CS 354 (Park) **Midterm** **March 4 (Mon.), 2019**

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 no 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) What is the role of the upper half in a modern kernel? What are the two main differences of XINU's implementation of its upper half in x86 compared to x86 versions of Linux and Windows? Which of the two impacts reliability/security and which effects kernel responsiveness?

(b) What is the role of the lower half? What has been the main component of the lower half of XINU discussed in the course so far? What are the three chores performed by this component? Which two of the three chores can prompt invocation of XINU's scheduler, and why?

(c) We discussed two hardware primitives and one software primitive for performing mutual exclusion. Why is interrupt disabling, in general, considered preferable to test-and-set (tset)? Why is using counting semaphores, in general, considered preferable to interrupt disabling? What is an example of a Linux system call where interrupt disabling would be especially detrimental for the system as a whole? Why are counting semaphore primitives (e.g., wait() and signal()) not a pure software solution for achieving mutual exclusion?

(d) In the XINU context-switch function, ctxsw(), EBP followed by EFLAGS which is followed by 8 general-purpose registers are pushed onto the stack of the process being context-switched out. When a process is context-switched in, instead of the reverse sequence—popal followed by popfl followed by restoring EBP—the order of EFLAGS and EBP is reversed. What is the reason? In our x86 XINU, is this necessary? Explain your reasoning. In the CDECL caller/callee convention as well as in x86's software interrupt int, the return address EIP is pushed onto a stack so that ret or iret can find their way back. Why is EIP not saved by ctxsw()?

PROBLEM 2 (48 pts)

(a) Suppose you are tasked to select a scheduler for an operating system that is to be a used in a computing system where all processes are CPU-bound. What is the simplest and most efficient solution that accomplishes fair sharing of CPU cycles? Explain its scheduling overhead. In operating environments with a mixture of CPU- and I/O-bound processes, why is the above solution considered insufficient? What rules are imposed by UNIX Solaris to affect improved scheduling? In what way is scheduling improved? As a consequence of the rules, could it happen that some CPU-bound processes end up not receiving CPU cycles for a prolonged period (e.g., 1 second)? What might be such a scenario? What is the basic idea behind Solaris's "safety net" that aims to prevent starvation?

(b) If the data structure being shared by two processes is a producer/consumer queue—in place of using mutual exclusion—a preferred way of protecting the shared data structure from corruption due to concurrent access is to use a pair of counting semaphores. What is the role of the two semaphores? What prologue code should the producer (i.e., writer) execute before writing to the queue? What epilogue code should it execute after completing a write operation? Under what condition is the producer prevented from writing? Given that mutual exclusion primitives suffice to protect a producer/consumer queue, what is gained by using a solution that involves not one, but two, semaphores?

(c) In lab2, you implemented a trapped version of XINU's chprio() that utilized x86's software interrupt support. What was the role of the wrapper function uchprio()? What was the role of the code at _Xint36 in intr.S? What was the role of XINU's legacy chprio()? In ctxsw(), we used pushal/popal to save/restore the 8 general purpose registers. Why does doing the same in _Xint36 lead to a problem? What was the work-around? Compared to trapped system call implementation in x86 Linux or Windows, your implementation did not switch stacks. Why not?

BONUS PROBLEM (10 pts)

Why does performing fair scheduling based on process CPU usage lead to logarithmic scheduling overhead? Rules (i) and (ii) in lab3 (and variants thereof) are needed when implementing fair scheduling in operating systems. What can go wrong if rule (ii) is omitted? How is this related to rule (i)?