CS 354 (Park) **Midterm** **Oct. 11 (Wed.), 2023**

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 (40 pts)

(a) What are the roles of the upper and lower halves of an operating system? For XINU, give an example where the scheduler, resched(), is invoked by the upper half. Do the same for the lower half. Describe a scenario where resched() returns to the caller without calling ctxsw().

(b) Suppose a XINU process is spawned using create() to execute a function abc(). What happens when abc() returns, i.e., its machine code executes the ret instruction? What tasks are performed by create() so that the process can be context-switched in by ctxsw() when it is selected to run for the first time? Provide a logical sketch that includes the IF flag of EFLAGS and the return address of ctxsw().

PROBLEM 2 (40 pts)

(a) A multi-level feedback queue allows constant overhead (i.e., $O(1)$ time complexity) scheduling when implementing the TS scheduler in Solaris UNIX. First, describe what a multi-level feedback queue is, second, explain why enqueue and dequeue operations incur constant overhead. What is the fundamental reason that Linux's CFS scheduler has logarithmic overhead?

(b) receive() is a blocking IPC system call in XINU. Explain what that means in comparison to send() which is nonblocking. Suppose a blocking version of send(), sendb(), were to be implemented. Describe how the internal logic of sendb() would differ from send(). Consider the possibility that multiple senders may call sendb() to send messages to the same receiver who has not had a chance to call receive() yet.

PROBLEM 3 (20 pts)

What were the three software layers of trapped system call implementation in lab2? What were their roles? How did we make use of legacy XINU system calls? How did the clobber list of registers in extended inline assembly help simplify coding? Linux and Windows utilize x86 hardware support to perform user/kernel stack switch in system calls. Our implementation performed stack switching in software. Why did we copy EFLAGS, CS, EIP from user stack to kernel stack upon trapping to kernel code, then copy the values back to user stack before executing iret?

BONUS PROBLEM (10 pts)

What is the main difference between how Linux and Windows configure x86's GDT versus how XINU configures it? What happens to the CS register when Linux and Windows make trapped system calls? What happens to CS when legacy XINU makes a system call?