CS 354 (Park) **Midterm** **March 14 (Fri.), 2014**

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 (36 pts)

(a) We discussed that XINU and modern kernels such as UNIX/Linux and Windows follow a design that is broadly divided into two parts: an upper half and a lower half. What are the responsibilities of each half? When kernel code is invoked in each half, which process runs them?

(b) Based on the material covered so far in class, what are the major design simplications of XINU when compared to kernels such as UNIX/Linux and Windows? List at least three based on their importance. Which of the three is most critical, and why?

(c) What method is preferred by XINU, especially in its system calls, to achieve mutual exclusion? Why is this approach problematic in desktop, server, and mobile kernels that we use every day? Whenever possible, what alternative method is utilized instead? What advantage does the alternative method have?

PROBLEM 2 (32 pts)

(a) In XINU, the context switch routine `ctxsw` is invoked by the scheduler `resched`. Who invokes `resched`? Why does `resched` not disable/restore interrupts in its code? What pieces of information does `ctxsw` save in a process's run-time stack in `x86`? What piece of information is stored in the process table?

(b) In general, it is good practice for app developers not to invoke system calls, whenever possible, due to their large overhead. Why do system calls incur high overhead when compared to regular function calls that remain in user space/user mode? When executing kernel code as part of a system call, why can a process's run-time stack not be used to do caller-callee bookkeeping of nested kernel function calls?

PROBLEM 3 (32 pts)

(a) The `receive` system call in XINU and `read` system call in Linux (by default) are blocking. What does this mean? What does it mean to make a system call such as `read` nonblocking? What is asynchronous IPC (in general, I/O) with callback function? When implementing this feature in a kernel, what complications arise that must be carefully handled not to violate isolation/protection?

(b) Most processes in today's operating systems are scheduled by the time share (TS) scheduling component of the kernel's scheduler whose overall aim is to provide "fair" sharing of CPU cycles. How does UNIX Solaris gauge, based on a process's recent behavior, whether it is CPU- or I/O-intensive? What actions does Solaris take when a process is deemed CPU- or I/O-intensive? What is the rationale behind these actions? Is the above sufficient to prevent starvation?

BONUS PROBLEM (10 pts)

Based on the material covered so far, list key hardware support features provided by today's computing systems that kernel programmers must be aware of and utlize when writing kernel code.