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  (40 pts)
(a) What is the motivating factor that prompted introduction of hardware support features for achieving isolation/protection in early production computing systems? What are these hardware support features? What role does the 2-bit IOPL field of the EFLAGS register play in our x86 architecture? What can happen if app processes are allowed to change IOPL? In what way is the x86 popfl instruction related to the preceding question?
(b) Based on the material covered thus far, what are the main design simplifications of XINU when compared to modern kernels (e.g., UNIX/Linux and Windows)?
(c) What are the pros/cons of RMS vs. EDF in real-time scheduling? Suppose two periodic real-time apps with period and computation time requirements (in msec) \( T_1 = 5, C_1 = 2, T_2 = 14, C_2 = 7 \) request to be scheduled at the same time. Can RMS and/or EDF schedule them? During the time interval from 0 to 12 msec, specify which of the two RT processes gets scheduled when, by emulating the actions of EDF.
(d) Kernels can be viewed as being comprised of two major parts: an upper half and a lower half. What are the roles of these two halves? Where would you place the scheduler, and why? In XINU, the upper half runs with interrupts disabled. Why is that so? What is the drawback of such a design?

PROBLEM 2  (36 pts)
(a) Suppose a kernel with generic isolation/protection hardware support uses full virtualization to concurrently run multiple guest kernels over a hypervisor. Suppose an app process in one of the guest kernels performs only ALU operations (e.g., addition) without any I/O. For this app process, what is the overhead incurred under full virtualization when compared to the case where the guest kernel is run directly over hardware? What is the overhead if the app performs print operations to a terminal device? What complication does x86 introduce? How does binary translation aim to effect full virtualization in x86? Why does locality of reference of typical apps help mitigate the overhead of binary translation?
(b) Why is the time complexity overhead of multi-level feedback queue used in TS scheduling constant? Why is the scheduling overhead of the fair scheduler (i.e., CFS) used by Linux logarithmic? How would you characterize the time complexity of RMS?
(c) What are the two main reasons that justify not providing deadlock detection as a default kernel service to applications? Why is providing selective deadlock detection—the service is provided only to processes that request it (as in lab3)—not harmful to the overall system as long as kernels do not perform system calls with interrupts disabled? Even when interrupts are not disabled by a kernel when performing its upper half chores and semaphores—a software solution—are used instead, why is using semaphores not a “pure” software solution?

PROBLEM 3  (24 pts)
(a) When we studied the context switch code of x86 XINU, we noted that when loading the context of the “new” process, instead of executing popfl right after popal—during context saving pushal was executed right after pushfl—ctxsw() first restored the saved frame pointer ebp by executing movl 4(%esp), %ebp. Why is restoring ebp before restoring EFLAGS necessary for correctness? Describe a scenario where not doing so would result in faulty execution, i.e., kernel bug.
(b) A software support feature that is needed to achieve isolation/protection is a per-process kernel stack that is used when executing kernel functions as part of a system call. Suppose a kernel were to use the per-process user space run-time stack instead. Describe how a hacker, using two user processes, can defeat isolation/protection and run its own code (i.e., malware) with the same privilege as the kernel.

BONUS PROBLEM  (10 pts)
A kernel, for the most part, is a collection of functions that are invoked by system calls and interrupts in the context of user processes. An alternative design is to implement most kernel services as processes. For example, the scheduler is a kernel thread—not a function such as resched() in XINU—that then performs the scheduling function. What are the main drawbacks of the latter approach that puts it at a significant disadvantage?