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) Describe a simple example of user process deadlock using binary semaphores. Explain the overhead of deadlock detection. What is the rationale followed by kernels when dealing with deadlocks? How is this related to isolation/protection? Above and beyond providing deadlock detection as a service, a kernel needs to use semaphores for its own data structures. What method can kernel programmers use to prevent kernel deadlocks?

(b) Why do kernels allocate part of the virtual address space to themselves? Give an example for 32-bit x86 kernel such as Linux. Suppose in one instance physical memory is 3 GB, in another 4 GB. How does this impact the usable address space of an app process? How is the Meltdown exploit related to kernels occupying part of the virtual address space? Why is providing two page tables per process—one for user mode, the other for kernel mode—considered a possible software solution for the exploit? What is a fundamental hardware fix to the problem?

(c) What are the pros/cons of implementing the bottom half of the lower half of a kernel by context-borrowing versus as a kernel thread? We discussed performance of isochronous webcam streaming in Linux where the bottom half is implemented by context-borrowing. We observed that data loss in the shared lower half/upper half kernel buffer lead to discarded frames at the application layer. What caused the losses in the 4 MB kernel buffer? What is the relationship between kernel buffer loss, copy overhead, and video compression? Does implementing the bottom half as a kernel process help improve performance? Explain.

(d) What is memory thrashing? What are its symptoms? Does increasing CPU speed alleviate the problem? Does increasing main memory help? What about increasing disk space that implements swap space? Explain your reasoning. We observed that in Linux, during thrashing, page fault rate does not indefinitely increase. Why is that? What options are available to a kernel when it notices onset of thrashing?

PROBLEM 2 (32 pts)
(a) Consider input from an I/O device without DMA support. The kernel is organized into upper and lower halves where the latter is further split into top and bottom halves. We are interested in what happens to three buffers: hardware buffer of the I/O device, kernel buffer shared by lower and upper halves, and user buffer that belongs to a process performing a read system call. First, describe from a producer/consumer perspective what happens to the three buffers when data arrives on the I/O device and an interrupt is raised. Be detailed in your description of what processes read/write what buffer. Assume context borrowing is used to implement the entire lower half. Second, what, if anything, needs to be done to protect the three buffers from concurrent access and potential data corruption? Third, how is output to the same device different from input?

(b) In part A of lab5, the page fault handler was implemented so that backing store support was not needed by restricting to test cases that did not deplete the total number of free frames. This meant that a page faulting process was guaranteed to find a free frame which prevented context-switching out by blocking on backing store I/O. How is this different for part B where backing store support is invoked upon page fault? Describe the overall structure of XINU’s backing store support and how blocking comes into play. Be specific by pointing out what XINU function call triggers blocking in the page faulting process. What tasks are carried out after the context-switch?

PROBLEM 3 (16 pts)
What are the pros/cons of tickless vs. tickful kernels? What is the overhead of a delta list for managing sleep events in XINU? What is a unified event queue? How would it simplify handling of sleep and alarm events (e.g., SIGALRM in Linux) for a kernel? How would it simplify managing the current process's time slice? What other kernel events can be integrated into the unified event queue? Describe a modification to XINU that makes it tickless with the help of a unified event queue. Has XINU’s overhead been impacted by the mod? What might be a design of a hybrid kernel that incorporates the benefits of both tickless and tickful kernels?

BONUS PROBLEM (10 pts)
Endowing a kernel with real-time capability goes beyond implementing scheduling algorithms such as RMS and EDF. What are non-algorithmic considerations—systems and otherwise—that are required to provide adequate real-time assurances by a kernel?