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 (32 pts)

(a) Throughout the CS 503 course, the context of a process has expanded and so has the overhead of context switching between processes. What are the main components of the context switching cost in modern computing systems? How does Linux (or Windows) allocating a significant part of a process’s virtual address space for itself help reduce context switching cost? Does user space multithreading have an advantage over multithreading implemented with kernel support with respect to context switching overhead?

(b) Increase of page fault rate is a key symptom of memory thrashing. What are the other telltale signs? In a case study of thrashing in Linux for a memory intensive application, we noticed a tendency for the page fault rate to level off, i.e., not keep increasing indefinitely. Why is that so? To help mitigate thrashing, suppose you have the option of increasing CPU speed, RAM, or disk I/O bandwidth. Which is the most effective and which the least?

(c) When a file system is implemented over flash memory (i.e., solid state drives), what are the key issues that must be addressed? In what way are log-structured file system (LFS) and flash memory a good match? What was the original motivation for LFS aimed at disks, and does it still hold for flash memory?

(d) What is external fragmentation and how is it addressed in modern computing systems and their kernels? Suppose a process makes a number of getmem() and freemem() system calls in its virtual address space that results in external fragmentation of its virtual address space. For example, a single getmem() that requests 9 MB of memory from kernel and a single freemem() call that frees up the middle third of the 9 MB virtual memory area leaving a 3 MB “hole” in heap space. Does the external fragmentation solution implemented in modern computing systems help mitigate the external fragmentation of a process’s virtual address space?

PROBLEM 2 (32 pts)

(a) What is the motivation behind dividing the lower half of a kernel into top and bottom halves? What are the two ways of implementing the bottom half in a modern kernel such as UNIX/Linux or Windows? What are their pros/cons? What is the job of the top half assuming DMA is not involved in device I/O? What is its job when DMA is activated? Suppose on a high-speed I/O device a continuous, high rate data stream arrives for a prolonged duration. Which of the two designs for the bottom half would you prefer and why? Does the top half/bottom half division still provide benefits in this persistent, high data rate scenario? Explain your reasoning.

(b) Describe the logic behind the design of Linux’s completely fair scheduler (CFS) and how it differs from Solaris’s TS scheduler in its approach. Why is the time complexity of CFS not $O(1)$ anymore? Despite their significant difference in design, why are the behaviors of CFS and Solaris TS with respect to scheduling CPU- and I/O-bound processes not too dissimilar at the end of the day?

(c) We discussed counting semaphores as an important synchronization primitive for inter-process communication (IPC) where two or more processes talk to each other. When discussing synchronization primitives for device I/O where a device controller (i.e., hardware) needs to send/receive data to/from processes, we observed that the same counting semaphore primitives used in IPC could be employed for device I/O. Why is that the case? When using counting semaphores, in what way is writing to a device different from reading from a device?

(d) What are the pros/cons of tickful vs. tickless kernel design? Which design may be more suited for kernels of mobile devices where battery power is a key concern? Does this still hold if a mobile device is primarily used for high-speed communication where kernel events are closely spaced? What is the main modification to XINU (and for that matter, most other kernels) required to make it into a tickless kernel?
PROBLEM 3 (26 pts)

(a) Explain the memory saving benefit of using a multi-level page table using the x86 2-level page table used in the XINU VM lab. For what types of processes is the benefit most significant and for what types is it the least? Is locking the first-level page table (i.e., page directory in x86) in RAM always an option in today’s systems with several gigabytes of RAM? What about locking in second-level page tables? Suppose virtual memory support is implemented not over disk but flash RAM as persistent storage. What prevents a kernel from putting all components of a multi-level page table onto NAND flash memory which is much more plentiful than RAM? For example, a laptop may have 4 GB of RAM but 128 GB of flash memory.

(b) We discussed video streaming using a web cam connected to FireWire 400/USB 2.0 in Linux/Windows running on x86 where the lower half utilized DMA. In Linux, we observed that video frames went missing at the sender kernel due to one or more pieces of a frame being dropped before reaching the application process. Describe where and how the bottleneck arises. How is the problem related to producer/consumer queues, and who are the producer (i.e., writer) and consumer (i.e., reader), respectively? All else being equal, would increasing CPU speed help alleviate the bottleneck? What about increasing the size of the producer/consumer buffer, say, 10-fold? What about using two CPUs (or cores) instead of one? Explain your reasoning.

BONUS PROBLEM (10 pts)

What is the chief design simplification of XINU when compared to operating systems such as UNIX/Linux and Windows? Note that it was not one of the features incorporated in the CS 503 labs. Sketch how you would modify x86 XINU to implement this feature.