CS 354 (Park) **Final** **May 1 (Wed.), 2019**

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 (52 pts)

(a) What is the main difference between the XINU and UNIX (i.e., traditional) file system design? How does it impact file access overhead? How has the 90/10 property of real-world file sizes has influenced the latter's design? For a file system implemented on hard disk, why has this property of file systems caused significant performance degradation?

(b) What are the pros/cons of tickful vs. tickless kernels? Suppose Arnold uses his smartphone to mainly check email and text messages. Mira uses her phone to video chat, stream on-line video and music. All else being equal, what kernel design is suited for Arnold? Which is more suited for Mira? Explain your reasoning. What hardware support is important for supporting Mira's activity on her mobile device? What would happen without this hardware support?

(c) Explain what internal and external fragmentation are. What is the computer science technique used to tackle external fragmentation? How is the technique implemented with hardware support in demand paging systems? Why is hardware support essential? Who handles TLB misses: hardware or kernel? In a 32-bit x86 architecture with 4 KB pages, how many entries are there in a single-level page table? Why is compressing single-level page tables using two-level page tables feasible in real-world computing environments?

(d) Solaris UNIX uses a multi-level feedback queue to achieve constant scheduling overhead of TS processes. Explain how $O(1)$ overhead is achieved for insert and extract operations. Compared to Linux's CFS, what is the essential difference that makes constant scheduling overhead possible?

PROBLEM 2 (48 pts)

(a) Suppose the upper and lower halves of a kernel (e.g., Linux/UNIX, Windows) share a kernel buffer for reading from a device hardware buffer (e.g., Ethernet). The content in the kernel buffer—sometimes after further processing by the kernel—is copied to a process's user space buffer. Viewing the kernel buffer as a producer/consumer queue, explain how counting semaphores from IPC are used for copying copying data from hardware buffer through kernel buffer to its final destination in user space buffer. Assume a blocking system call—say read() in Linux—is invoked by the receiver process P_1, and the lower half of the kernel responds to an interrupt indicating packet arrival by borrowing the context of the current process P_2. Your description needs to be detailed.

(b) Virtual memory management using demand paging increases process context-switch overhead. What is the reason for that? In XINU's ctxsw() kernel function, what additional key piece of information would need to be pushed onto the stack? What is a page fault? Who handles page faults: hardware or kernel? What is the rationale behind the design choice? If a kernel experiences a high page fault rate, it may be a sign of memory thrashing. What is memory thrashing? What other characteristic symptoms may be checked to confirm the diagnosis? Would increasing CPU speed alleviate memory thrashing? Explain your reasoning.

(c) In lab5, you implemented callback function support for asynchronous message receive. Although XINU does not implement user mode/kernel separation—processes always run in kernel mode—the implementation in lab5 followed a design compatible with isolation/protection that makes it portable to Linux/UNIX and Windows. For the case where a process that registered a callback function has been context-switched out, describe how XINU in lab5 has to implement asynchronous callback function execution so that callback function execution occurs in user context (user mode if the implementation were ported to Linux/UNIX and Windows). Start with your modifications to clkdisp.S, the role of xruncb_lh(), and the registered callback function.

BONUS PROBLEM (10 pts)

Deadlock detection is typically not considered an essential service that a modern kernel needs to provide. What is the overhead of deadlock detection? What is the rationale behind not providing deadlock detection services? Above and beyond supporting/not supporting deadlock detection system calls, the kernel itself has to be coded so that deadlocks do not arise. What is a straightforward programming method that is guaranteed to prevent deadlocks?