CPS 110 Midterm

March 9, 1999

There are four questions equally weighted at 50 points each. Answer all questions. Please sign your name and stapling all your answers together. Allocate your time carefully. Your answers will be graded on content, not style. For code answers, any kind of pseudocode is fine as long as its meaning is clear. For written answers to "essay" questions, please do not waste any words. You have 75 minutes.

1. *The Ring Cycle*. This problem asks you to generalize the ping-pong (*Tweedle**) example discussed in class. You are to write a procedure *TakeTurns(int)* that will be called by N threads (N is a constant). The argument to *TakeTurns* is an integer between 0 and N-1 that uniquely identifies the calling thread. *TakeTurns* executes an endless loop, printing out its argument (the calling thread ID) on each iteration. The intended behavior is that the thread IDs [0..N-1] will be printed in order in an endlessly repeating cycle.

   (a) Implement *TakeTurns* using semaphore(s).

   (b) Implement *TakeTurns* using a mutex and a single condition variable. Your solution should work properly whether or not condition variables are fair.

   (c) Briefly discuss the expected performance of your solution for part (b). How many context switches do you expect will occur for each turn? On a uniprocessor? On a shared-memory multiprocessor?

   (d) Implement a more efficient *TakeTurns* solution using condition variables, and briefly explain why it is efficient.

2. *Impatient Wait*. The Java *Object.Wait* primitive is similar to Nachos *Condition::Wait*, except that *Object.Wait* may be optionally bounded by a timeout. The timed variant *Object.Wait(int timeout)* has the following semantics: the calling thread sleeps until either (a) it is awakened by another thread's signal (*Object.Notify*) using ordinary Mesa condition variable semantics, or (b) the period of time specified by *timeout* has elapsed. Show how to implement condition variables with a Java-style bounded wait in a Nachos-like setting. You may assume the existence of familiar Nachos internal primitives and/or an *Alarm* facility as implemented in Lab #3.

3. *Mode and Context*. The following questions concern the handling of processes in a protected kernel-based operating system such as Nachos. Assume that the kernel creates and initializes each process to execute a statically linked program from an executable file whose name is passed as an argument to the process create primitive (as the Nachos *Exec* system call). **Note**: your answers should ignore the issues of page table structure, the virtual address translation mechanism, and argument handling.

   (a) Explain how the kernel initializes physical memory for use by the program. How does the kernel determine the initial values for the memory allocated to the new process?

   (b) Explain how the kernel initializes the program counter register (PC) and stack pointer

register (SP) before switching the CPU to user mode to start the fresh process for the first time. How are the initial values of these registers determined? What about the other registers?

(c) Once the CPU is executing in user mode in the context of the fresh process, what could cause it to switch back into kernel mode? Give three distinct examples and outline how each affects the PC and SP registers on re-entry to the kernel. What about the other registers?

(d) Once the CPU is in kernel mode as a result of the examples in part (c), what could cause it to switch back into user mode? List as many distinct scenarios as you can think of (I can think of six or seven good ones), and outline for each case how the kernel determines values for the PC and SP registers before the switch to user mode. What about the other registers?

(e) Briefly enumerate the kernel data structures involved in representing the new process and the resources allocated to it.

4. *Event*. This problem asks you to implement an *Event* class similar to the fundamental coordination primitives used throughout Windows and NT. Initially, an *Event* object is in an *unsignaled* state. *Event::Wait()* blocks the calling thread if the event object is in the unsignaled state. *Event::Signal()* transitions the event to the *signaled* state, waking up all threads waiting on the event. If *Wait* is called on an event in the signaled state, it returns without blocking the caller. *Event::Reset()* resets an event object to the unsignaled state. It is illegal to call *Signal* twice without an intervening *Reset*, but you need not enforce this.

   (a) Implement *Event* using a single semaphore with no additional synchronization.

   (b) Implement *Event* using a mutex and condition variable.

   (c) Show how to use *Event* to implement the synchronization for a *Join* primitive for processes or threads. Your solution should show how *Event* could be used by the code for thread/process *Exit* as well as by *Join*. For this problem you need not be concerned with deleting the objects involved, or with passing a status result from the *Join*.