CS 33211  
Exam #2  
OS  
Monday 30 October 2006

1. Consider processes and threads.
   a. What process resources does the thread share with other threads? (5 points)
      Address space, program code, global variables, heap, and any resources assigned by the OS such as files or I/O devices. Technically, all the threads also share the stack but they pretend they do not.
   b. What process resources are associated with each thread that it does not share with other threads? (5 points)
      CPU register data, Program Counter, Stack Pointer, and stack (though see note above).
   c. Explain how support for a lightweight process provides a process with the advantages of both user-level and kernel-level threads. (10 points)
      The primary advantage of kernel-level threads is that the kernel has full knowledge of the threads, so the CPU scheduler might choose to give a process with 10 kernel-level threads more CPU time than a process with 1 kernel-level thread. Since one kernel-level thread is associated with each lightweight process, lightweight processes provide this advantage of kernel-level threads.
      The primary advantages of user-level threads is that their simple representation and lack of OS involvement in switching makes context switches much faster than with kernel-level threads. Since lightweight processes allow the user full control over which threads are scheduled onto each lightweight process, lightweight processes provide this advantage of user-level threads.

2. CPU scheduling methods are chosen to meet a variety of goals, one of which is to avoid the starvation of a process.
   a. What is meant by “starvation of a process”? (6 points)
      A process is said to starve if it is indefinitely delayed from being scheduled onto the CPU — it “starves” due to lack of CPU cycles to “consume”.
   b. Consider the First-Come-First-Serve (FCFS), Round-Robin (RR), and Shortest-Job-First (SJF) CPU scheduling algorithms. For each of these algorithms, is starvation possible or not? (Just answer “yes” or “no” for each — no additional text is necessary.) (6 points)
c. **How does process “aging” in CPU scheduling methods such as priority scheduling or multi-level feedback queue scheduling help avoid starvation?** (8 points)

Aging increases the priority of a process based on the length of time it has been waiting to be scheduled onto the CPU. With the proper tradeoff between scheduling priorities and aging, processes will likely avoid starvation.

3. **Write the pseudocode that defines the “P” or “wait” semaphore operation (ideally, the version discussed for most of the class, rather than the “classical” version introduced at the end of the semaphore discussion).** (10 points)

```plaintext
s = s - 1
if (s < 0)
    block the thread that called wait(s) on a queue associated with that semaphore
otherwise
    let the thread that called wait(s) continue into the critical section
```

4. **Consider a message queue shared between two threads. Assume a thread named ReceiveMessage adds messages to the queue, a thread named ProcessMessage removes messages from the queue, and a semaphore named QueueFull is available. Do not worry about what happens when the queue is empty, but explain how the semaphore can be used to insure that no more than 10 messages are added to a message queue at any time.** (15 points)

*Sigh. Professors should make out the key before giving the exam, not after, don’t you think? What I meant to assign was a question like the bounded-buffer producer-consumer Coke Machine example, so what I should have called the semaphore was QueueEmpty, like emptySlot in the Coke Machine example.*

If I had done that, you would initialize semaphore QueueEmpty to 10, meaning that up to 10 messages can be added to the empty queue.

In ReceiveMessage, you would call wait(QueueEmpty) to decrement the number of empty slots in the queue before adding a message to the queue, and to block (wait) if there are no empty slots.

In ProcessMessage, you would call signal(QueueEmpty) to increment the number of empty slots in the queue after adding a message to the queue, possibly signaling some process waiting in ReceiveMessage for an empty slot to become available.

*All of this still works if you call the semaphore QueueFull instead of QueueEmpty, but it doesn’t make as much sense reading the code.*
5. Why is disabling interrupts not a good method to use in implementing semaphores? (10 points)

The primary reason is that if interrupts are disabled by a wait( ) operation, which then waits for the semaphore to be incremented, the signal( ) operation will never be called, so it can never increment the semaphore. Other reasons include the undesirability of disabling the system timer (which requires interrupts to operate), as well as the fact that this method will not work on a multiprocessor system (since disabling interrupts on one processor will have no effect on the other processors).

6. Deadlock can be prevented if one of the 4 conditions for deadlock can be eliminated. Which of the 4 conditions can best be eliminated, and how can it be eliminated? (15 points)

It is easiest to eliminate the “mutual exclusion” condition, at least from the point of view of the processes. If exclusive access is required to a resource such as a printer, so that only one process can print to a particular piece of paper at a time, it is possible to design a “print server” that receives multiple user requests simultaneously and then processes those requests sequentially. From each process’s point of view, it “prints” when it contacts the server, and since it does not have to wait to contact the server until another process finishes, multiple processes can appear to be “printing” at the same time, thus eliminating the “mutual exclusion” condition.

(It isn’t possible to eliminate the “no preemption” except for resources whose state can be restored if they are preempted, eliminating “hold and wait” is difficult since you don’t usually know resource needs in advance, and eliminating “circular wait” is difficult without some awkward method to force all processes to request resources in the same order.)

7. In the Banker’s Algorithm for deadlock avoidance, what does it mean to say the system is in a “safe” state, and what does it mean to say the system is in an “unsafe” state? (10 points)

If the system is in a “safe” state, there are enough resources in the system (counting available resources and currently allocated resources) that all processes can run to completion in some order, assuming that when a process runs it might make its maximum resource request, and when it completes it will return all of its resources back to the system.

If the system is in an “unsafe” state, there are not enough resources in the system that all processes can run to completion under the assumptions above, so it is possible (though not guaranteed since some processes might not make its maximum resource request) that the system might deadlock.
EXTRA CREDIT QUESTION ANNOUNCED IN CLASS ON WEDNESDAY 25 OCTOBER: WHAT ARE THE ORIGINS OF THE TERM “BSS” THAT IS USED TO SPECIFY THE UNINITIALIZED DATA IN AN OBJECT FILE? (5 POINTS)


In computer programming, .bss or bss is used by many compilers and linkers as the name of the data segment containing uninitialized variables. It is often referred to as the "bss section" or "bss segment".

Historically, BSS (from Block Started by Symbol) was a pseudo-instruction in UA-SAP (United Aircraft Symbolic Assembly Program), the assembler developed in the mid-1950s for the IBM 704 by Roy Nutt, Walter Ramshaw, and others at United Aircraft Corporation.

The BSS keyword was later incorporated into FAP (FORTRAN Assembly Program), IBM’s standard assembler for its 709 and 7090/94 computers. It defined a label and reserved uninitialized space for a given number of words.

A similar origin is attributed to Dennis Ritchie in the FAQ at: http://www.faqs.org/faqs/unix-faq/faq/part1/section-3.html