# Operating Systems: Sample Midterm

Summer Semester 2009

1. [25 points] For each of the statements below, indicate in one sentence whether or not the statement is true or false, and why.

   - In a concurrent programming environment that provides non-preemptively scheduled threads (i.e., no involuntary context switches), mutual exclusion synchronization is not required.

     [False. Mutex synchronization is still required (1) on multiprocessors, and (2) on uniprocessors for critical sections that contain blocking operations.]
   - Busy waiting is always less efficient than a blocking wait operation.

     [False. Busy waiting can be more efficient if the expected wait time is shorter than the time it takes to preempt and re-schedule a thread. This is common on multiprocessors.]
   - In a combined segmentation + paging system, there is no internal fragmentation.

     [False. Segment sizes must be a multiple of the page size.]

   Answer each of the following questions as concisely as possible.

   - Can an operating system detect that some of its processes are starving? If so, explain how it can. If no, explain how the system can deal with the starvation problem.

     [Yes and no. Determining starvation precisely would require predicting the future. However, for practical purposes, a process that remains in the ready queue for a "long time" without ever being scheduled can be considered starving.]
   - Can an operating system detect that some of its processes are deadlocked? If so, explain how it can. If no, explain how the system can deal with the deadlock problem.

     [Yes, at considerable expense. E.g., maintain the resource-process graph and check for cycles.]
   - Consider a system that has four memory pages that are shared by three processes, each of which needs at most two pages. The processes can dynamically allocate pages, and must wait for a page to become available. Once a page is allocated to a process, it cannot be reclaimed until the process releases the page. Can this system deadlock?

     [Assuming the processes do not share any other resources, no. A cyclic wait condition cannot occur, because at least one process will always be able to acquire two pages, thus making progress.]

2. [15 points] Condition synchronization in monitors is accomplished with the help of condition variables. Instead of condition variables, a more general form of condition synchronization would be to have a primitive that can suspend execution until a given boolean expression evaluates to the value TRUE. For example, one could write **await** *(active_writers == 0 && waiting_writers == 0)*; This primitive is clearly more general and convenient than condition variables (note that no SIGNAL operation is needed), but it is not used in practice. Explain why.

   [No efficient implementation is known to exist. Conceptually, whenever the shared state changes, the boolean expression of each outstanding await statement must be reevaluated.]

3. [20 points] Customers (represented by threads) repeatedly enter a post-office, draw a number, and wait. $N$ clerks (also represented by threads) repeatedly pick the next customer and service that customer, which takes a random amount of time. Once the service is complete, the customer can proceed.

Customers must be serviced (i.e., service must begin) in the order of arrival, and exactly one clerk must service each customer. Implement a concurrent program that solves this problem. Customers call EnterPostOffice(); the function should return no sooner than the completion of service for the customer. You may assume that the number of customers waiting at any given time is bounded.

```
monitor PostOffice {
    int nClntsWaiting = 0;
    condition idleClerks;
    queue waitingQueue; // queue of instances of condition;

    void enterPostOffice() {
        condition myCond;
        /* signal any idle clerks */
        idleClerks.signal();
        /* wait until my service completes */
        nClntsWaiting++;
        waitingQueue.add(myCond);
        myCond.wait();
    }

    condition clerkReadyToServe() {
        condition myCustomer;
        /* wait for customers */
        while (!nClntsWaiting) idleClerks.wait();
        myCustomer = waitingQueue.RemoveFirst();
        nClntsWaiting– –;
        return myCustomer;
    }

    void clerkFinished(condition myCustomer) {
        myCustomer.signal();
    }
}

void clerk() {
    while(working) {
        condition myCustomer = clerkReadyToServe();
        /* serve customer (takes a random amount of time) */
        clerkFinished(myCustomer);
    }
}

void customer() {
    while(1) {
        enterPostOffice();
    }
}
```

4. [10 points] Assume you have a page reference string for a process running on a system with $m$ physical pages. All of the process's pages are initially non-resident. The page reference string has length $p$ with $n$ distinct pages occurring in it. The process is running uninterrupted. For any page replacement algorithm,

   - what is a lower bound on the number of page faults?
   - what is an upper bound on the number of page faults?

   [Lower bound: $n$. Upper bound: for $m < n$: $p$, for $m >= n$: n.]

5. [15 points] Suppose that a 32-bit virtual address is broken up into four fields, $a$, $b$, $c$, and $d$, to support a three-level page table. Explain the tradeoffs involved in deciding how many bits of the address should be assigned to each field. What type of information would you need about the expected workload on your system to make a good assignment?

   [$d$ determines page size, trading off internal fragmentation and size of page tables. $a$, $b$, $c$ determine size of page tables at levels 1, 2, 3. In the absence of any information about address space layout, assigning an even number of bits to each level is a good guess. In general, need information about typical address space population to make a better choice.]

6. [15 points] Consider a multiprogramming system with combined segmentation and paging hardware. Suppose many different programs run at a given time, and that each of these includes in its executable code copies of the library routines it uses. It is desired that this redundancy be eliminated by having just one copy of the entire library in memory, to be shared by all programs. Describe how this can be done. What restrictions does your solution impose on the library and the programs using the library?

   [There are a number of possible correct answers. Here is a simple solution. Reserve two fixed segments in each process for the shared library: one for the lib's code, one for the lib's data. The linker resolves references to the library as usual, but does not include the library's code segment. The format of the executable is unchanged, except that the lib code segment is missing. The loader proceeds as usual, but sets the shared library code segment's table entry to refer to the (shared) page table that maps the (shared) copy of the lib code in physical memory (loading it when necessary), with read-only permissions. Note that the lib data segment is mapped to separate memory for each process in the usual way (requiring both a private set of pages and a private page table for the data segment). The solution imposes the following restrictions: Two segments must be reserved for the library in each process. If the library changes, all executables must be re-linked. The library code must not be self-modifying.]