Examination 2

COP 4600 Operating Systems
March 27, 1997

Instructions

1. **Read and follow all instructions.**
2. This is a closed-book examination.
3. You are permitted on 8.5 by 11 inch sheet of notes, both sides, that you have prepared.
4. **Answer any three (3) questions**. All questions are of equal value. Points for each part of each question are given in parentheses.
5. **Leave sufficient room in the upper lefthand corner for the staple.**
6. Use exactly one page of paper (both sides is OK) to hold the answer to each question, and please write legibly.
7. Start the answer to each question on a new page (i.e., do **not** put the answer to more than one question on the same page).
8. Assemble your answers in numerical order of the questions when you submit them.
9. **Read and sign the following statement.** You may write this on your exam and sign it there if you wish to take the exam questions home with you today.

On my honor, I have neither given nor received unauthorized aid on this examination.
Signed:

1. (a) (7) Describe and compare shared memory solutions (using the atomic load/store model) to hardware solutions for the critical section problem. Under what circumstances would either of these types of solution fail, and how?

   (b) (8) In what way do semaphores improve over either of the solutions of the previous part of this question? When would semaphores be most useful? When might semaphores be less desirable than the solutions from the previous part? Why?

   (c) (6) How do monitors improve over semaphores for synchronization? Compare and contrast condition variables and semaphores.

   (d) (4) Describe a synchronization primitive provided in Minix. How does it provide synchronization?

2. Consider a system in which there are four processes (P_1, P_2, P_3, P_4) and three resource types (A, B, and C). There are three instances of A, two instances of B, and four instances of C. P_1 holds one instance of A and one of C, P_2 holds one instance of B and two of C, P_3 holds one instance of B, and P_4 holds one instance of A.

   (a) (6) Suppose that P_1 requests one instance of A and one of C, P_2 requests one instance of B, P_3 requests two instances of A and one of C, and P_4 requests one instance of A and two of C. Draw the Resource Allocation Graph for this system state.

   (b) (6) Is the above system in deadlock or not? If so, state why, if not, give an order in which the processes can complete.

   (c) (5) What is the difference between a safe state and an unsafe state in which there is no deadlock? Why can a new request never take a system from a safe state into an unsafe state?

   (d) (8) Give example system states showing how a request can take a system from an unsafe state to a deadlocked state; how an allocation can take a system from a safe state to a deadlocked state; and how an allocation can take a system from a safe state to an unsafe state.

3. (a) (8) What are the necessary conditions for deadlock? Why is each one necessary?

   (b) (9) What are the three ways of handling deadlock (other than ignoring it)? Name, describe and compare them. Which condition(s) does each negate?

   (c) (8) What are resource contention graphs, and how can they be used to provide for greater concurrency when a system constrains the order in which resources may be requested? What type of solution is this and what condition(s) does it negate?

4. Suppose the head of a hard disk is at track 73, where tracks run from 0 to 255, and it just serviced a request at track 51. Suppose there are requests queued (in the order of arrival) for accesses at tracks 3, 60, 95, 85, 123, 17, 90, and 188.

   (a) (10) Give the total head movement required to service the requests in the following orders: FCFS, SSTF, SCAN, C-SCAN, LOOK. Show your work.

   (b) (3) What is the optimal order (assuming no new requests arrive)? Give its total head movement.

   (c) (9) What are the three components contributing to the time required for a disk access? What can be done to improve each of these times? Which component does disk scheduling focus on, and why? Why is disk scheduling often less helpful than it might be expected to be?

   (d) (3) What disk scheduling algorithm does Minix use? Why?

5. (25)

Solve the following synchronization problem using monitors or semaphores by writing the code for `get_free_buffer()`, `get_full_buffers()`, `announce_free()`, `announce_full()` and any additional code needed. Be complete (declare and initialize variables, etc.). You may assume common ADTs such as stacks and queues with the usual access procedures.

There are two types of items: A and B. Corresponding to each item type, there is a producer type that exclusively produces items of its type. Both types of producer share a common buffer pool in which to store the items they produce and where consumers find them. Consumers require simultaneously one item of type A and another of type B fro processing. An item of either type uses one buffer for storage, and there are N buffers total available. Producers only request a buffer when they already have an item ready to store in it, and consumers pass in a pair of pointers to local buffers into which the items they obtain are copied for processing later. You may assume that `copy_buff()` copies the data in the buffer pointed to by the first argument into the buffer pointed to by the second argument.

Your solution must be correct, must not have starvation of either producer type or the consumer, and must allow N to be any value larger than 1. The code for generic producer and consumer processes is given below, with X taking values in A or B. You should assume that there are several processes of each type, each running at its own speed.

```
Producer_X() {
         buffer local_buff, *shared_buff;
         while (TRUE) {
                 produce_X(&local_buff)
                 shared_buff = get_free_buffer(X);
                 copy_buff(&local_buff, shared_buff);
                 announce_full(X, shared_buff);
                 }
         }
Consumer() {
         buffer local_buff_A, local_buff_B,
                 *shared_buff_A, *shared_buff_B;
         while (TRUE) {
                 get_full_buffers(&shared_buff_A, &shared_buff_B);
                 copy_buff(shared_buff_A, &local_buff_A);
                 copy_buff(shared_buff_B, &local_buff_B);
                 announce_free(shared_buff_A);
                 announce_free(shared_buff_B);
                 consume(&local_buff_A, &local_buff_B)
                 }
         }
```