CS 520: Introduction to Operating Systems

Homework Assignment #1

Please complete reading
1) Sections 1.1 through 1.4 of Chapter 1;
2) Sections 2.1 through 2.4 of Chapter 2;
3) Sections 3.1 through 3.3 of Chapter 3; and
4) All of of Chapter 6.

Do as many practice exercises as you can.

Solve the following problems:

1. Consider modifying the *Enter_Critical* procedure (see *Lecture 1*) to assign the *Id* of the executing process (rather than *other*) so that the procedure looks like that:

```
void Enter_Critical (int process) /* 0 or 1 */
 {
    int other;

    other = 1 - process;
    flag[process] = TRUE;     /* show my interest */
    turn = process;            /* grab it! */
    while (flag [other] == TRUE && turn == process);
 }
```

   Will the algorithm still work (i.e., provide the critical section access satisfying all three criteria)? Why or why not? (20 points.)

2. Does the busy waiting solution work when the two processes are running on a shared-memory multiprocessor? (5 points.)

3. When a computer is being developed, it is often first simulated by a program that runs one instruction at a time. (Even multiprocessors are simulated strictly sequentially like this.) Is it possible for a race condition to occur in such situations? (Please explain your answer.) (10 points.)

4. Suppose a queue in a semaphore is implemented not as a *first-in-first-out (FIFO)* queue, but as *last-in-first-out (LIFO)* stack. Show how this can result in *starvation*, that is a situation when a process is indefinitely queued in a semaphore and thus can never progress. (10 points.)

5. One reason semaphores are used is for *mutual exclusion*—that is ensuring that only one process enter a critical session to avoid race conditions. The other reason is *synchronization*, that is ensuring the events happen in certain sequence. The lecture specifies three semaphores *mutex*, *empty*, and *full*. Which ones of these are used for mutual exclusion, and which ones—for synchronization? Explain your answer. (5 points.)

6. Study the *Bounded_Buffer* problem (Section 7.1.1) carefully. Suppose the two *wait* statements of Figure 7.1 were reversed by the programmer [so that *wait(mutex)* is executed before *wait(empty)*]. What will happen when the buffer is full? (Note: The *empty* semaphore is initialized to *buffer_size*; the *full* semaphore is initialized to *0*.) (15 points.)

7. *The Sleeping-Barber Problem*. A barbershop consists of a waiting room with *n* chairs and the barber room containing the barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber.

   Using the semaphore constructs described in the textbook (and used for solving other problems) write a program to coordinate the barber and the customers.That is, write one procedure for the *Barber* process and one procedure for the customer process.) [Hint: define the *customers*, *barbers*, and *mutex* semaphores.]

   The code can be schematic (just the same as in the lecture). You don't have to write the real code. (20 points)

8. ***Artists and Viewers.*** In the new Center for Strange Arts (CESA) in New York City, there is a famous Weird Hall, open 24 hours a day, with a very large painting being perpetually developed by the walk-in artists. Because the artists tend to fight with one another as well as accost viewers, only one artist at a time may enter the Weird Hall, and once an artist is there, no other artists or viewers are allowed in. (They have to wait in front of the Weird Hall.) If no artist is in, the viewers may come in at any time and in any numbers; they are not queued behind the artists. Furthermore, as long as there is at least one viewer is in the Weird Hall, no artist is allowed in. Write the pseudo-code for synchronization of viewers and artists using semaphores. (15 points)

9. **(Bonus problem for extra credit—50 points).** This problem is ***not*** mandatory, You should work on it ***only*** if you have finished the rest of the homework; otherwise no extra credit will be given.

Write the real code for the Sleeping Barber problem. You can write it in C using the Unix or Linux semaphore constructs, in which case you will need to create a process for the barber and a process for each customer. Alternatively, you can use Java and its thread management mechanism.

You can experiment with both the values of customer arrival times and the respective haircut times. Document your experiments.