1. Race conditions are possible in many computer systems. Consider a banking system with two methods: deposit(amount) and withdraw(amount). These two methods are passed the amount that is to be deposited or withdrawn from a bank account. Assume that a husband and wife share a bank account and that concurrently the husband calls the withdraw() method and the wife calls deposit(). Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

   Answer:

   Assume the balance in the account is $250.00 and the husband calls withdraw($50) and the wife calls deposit($100). Obviously, the correct value should be $300.00. Since these two transactions will be serialized, the local value of balance for the husband becomes $200.00, but before he can commit the transaction, the deposit($100) operation takes place and updates the shared value of balance to $350.00. We then switch back to the husband and the value of the shared balance is set to $200.00 - obviously an incorrect value.

2. Race conditions are possible in many computer systems. Consider the producer–consumer problem, which is representative of operating systems, consisting of cooperating sequential processes, running asynchronously and sharing data using a bounded buffer. An integer variable counter, initialized to 0 is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

   The code for the producer process is as follows:

```
while (true) {
        /* produce an item in next_produced */
        while (counter == BUFFER_SIZE)  ;          /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

   The code for the consumer process is as follows:

```
while (true)
        { while (counter == 0) ;              /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
        /* consume the item in next _consumed */
}
```

   Note that the statement “counter++” may be implemented in machine language (where register1 is one of the local CPU registers) as follows:

$$register_1 = counter$$
$$register_1 = register_1 + 1$$
$$counter = register_1$$

Similarly, the statement "counter--" is implemented as follows:

$register_2$ = counter
$register_2$ = $register_2$ − 1
counter = $register_2$

a. Describe how a race condition is possible Page 259 in book.
b. What data have a race condition? counter
c. What might be done to prevent the race condition from occurring? Page 259 in book

3. Define:
   a. Critical-section problem Page 260.
   b. Race condition page 259
   c. Busy waiting Page 272
4. What are the 3 requirements that a solution to the critical-section problem must satisfy? Page 260
5. What is a semaphore? What is it used for? Page 272, 273
6. Consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2. Suppose we require that *S*2 be executed only after *S*1 has completed. How do you implement this scheme using a semaphore? [Hint: Refer section 6.6.1]

   We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.

   In process P1, we insert the statements
   S1;
   signal(synch);

   In process P2, we insert the statements
   wait(synch);
   S2;

   Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.
7. Write the code for the two Semaphore operations. Page 273 Why are they atomic? For uninterrupted updates of shared data
8. Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—spinlock, or a mutex lock (where waiting processes sleep while waiting for the lock to become available):
   - The lock is to be held for a short duration. Spinlock
   - The lock is to be held for a long duration. Mutex lock
9. Explain how deadlock is possible with the dining-philosophers problem.

   Suppose that all five philosophers become hungry at the same time and each grabs the left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab the right chopstick, deadlock happens.
10. Consider a system consisting of two processes, *P*0 and *P*1, each accessing two semaphores, S and Q, set to the value 1:

```
        P0                  P1
wait(S);            wait(Q);
wait(Q);            wait(S);
   .                   .
   .                   .
   .                   .
signal(S);          signal(Q);
signal(Q);          signal(S);
```

Suppose that P0 executes wait(S) and then P1 executes wait(Q). When P0 executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until P0 executes signal(S). Is the system deadlocked? Explain why or why not.

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q).
When P_0 executes wait(Q), it must wait until P_1 executes signal(Q).
Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S).
Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

11. Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

    signal(mutex);
    ...
    critical section
    ...
    wait(mutex);
    Find the error generated by the use semaphores incorrectly.

    In this situation, several processes may be executing in their critical section simultaneously, violating the mutual-exclusion requirement.

12. The structure of the Producer Process is given below. What are the purposes of the semaphores in the code?

```
do {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
} while (true);
```

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1.
The empty and full semaphores count the number of empty and full buffers.
The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.