1. 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
   b. What data have a race condition?
   c. What might be done to prevent the race condition from occurring?

2. Define:
   a. Critical-section problem
   b. Race condition
   c. Busy waiting
3. What are the 3 requirements that a solution to the critical-section problem must satisfy?
4. What is a semaphore? What is it used for?

5. 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]
6. Write the code for the two Semaphore operations. Why are they atomic?
7. 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.
   - The lock is to be held for a long duration.
8. Explain how deadlock is possible with the dining-philosophers problem.
9. 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.

10. 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.

11. 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);
```