CS330: Operating Systems
Quiz#4

Name:
Roll No.:

1. Consider the following function. The semaphores `a`, `b`, `c`, `d` are all initialized to one.

```
sem_t a, b, c, d;
void f (sem_t *s1, sem_t *s2, sem_t *s3, sem_t *s4)
{
   sem_wait(s1); sem_wait(s2); sem_wait(s3); sem_wait(s4);
   Critical section
   sem_post(s1); sem_post(s2); sem_post(s3); sem_post(s4);
}
```

Two threads call the function `f` concurrently. One of these threads invokes `f` as f(&c, &b, &a, &d). Write down all possible correct invocations of function `f` by the other thread. **(2 points)**

Solution: All invocations that share a common prefix with f(&c, &b, &a, &d) will be correct. So, these are f(&c, &b, &a, &d), f(&c, &b, &d, &a), f(&c, &a, &b, &d), f(&c, &a, &d, &b), f(&c, &d, &a, &b), f(&c, &d, &b, &a).

Grading policy: 0.5 mark for writing any one of the six correct invocations. One mark for writing at least two correct invocations. 2 marks for writing all six. Zero mark for writing anything wrong.

2. The following program segment is executed by ten threads. The initial value of the shared variable `x` is zero. The semaphore `s` is initialized to eight. What are the possible values of `x` after all threads complete execution? **(3 points)**

```
sem_wait(&s);
x++;
sem_post(&s);
```

Solution: At any point in time, there can be at most eight threads executing `x++` concurrently. Thus during the execution of the first batch of eight threads, the value of `x` can vary from 1 to 8. Therefore, when the remaining two threads execute, the value of `x` can go to 9 or 10. Also, it is possible that the thread, which read the value of `x` as zero during the execution of the first batch of threads, finishes last and writes 1 to `x`. Thus, the possible values of `x` are 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. To see how `x` can attain a final value of k for $1 \leq k \leq 10$, consider a thread T that reads `x` after $k-1$ threads have read, incremented, and updated `x` one by one maintaining mutual exclusion. It is always possible to construct such a schedule as long as the initial value of the semaphore is positive. So, the value of `x` read by T is $k-1$. Now, you need to construct a schedule where T is context switched right after reading `x` and is scheduled again after all other threads have completed execution. Construction of such a schedule requires the semaphore value to be at least two. So, T is the last thread to update `x`. Therefore, the final value of `x` will be k.

Grading policy: One mark less for each missing possible value. Thus, if you miss three or more possible values of `x`, you get zero.

3. Consider two semaphores `s1` and `s2`. The variable `count` is shared between two threads and initialized to `x`. The following program segments are executed by two threads. If the final value of `count` is always `x-1` after two threads have completed execution, what are the initial values of the semaphores `s1` and `s2`? **(3 points)**

```
Thread A                                  Thread B
----------                                ----------

sem_wait(&s1);                            sem_wait(&s2);
if (count == x) count--;                  if (count == x) count -= 2;
else count++;                             else count += 2;
sem_post(&s2);                            sem_post(&s1);
```

Solution: For `count` to have a final value of `x-1`, Thread B must execute first and then Thread A executes. So, `s2` should have a positive initial value and `s1` should have a zero initial value.

Grading policy: 1.5 marks for each correct answer. Zero mark if you write multiple possible initial values for `s1`.

4. Consider the following program segments executed by two threads. What possible values can `Thread B` print? **(2 points)**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
int x = 0;
int y = 0;
Thread A                                  Thread B
----------                                ----------

y = 2;                                    while (x == 0) {
pthread_mutex_lock(&lock);                   pthread_mutex_lock(&lock);
x = 1;                                       pthread_cond_wait(&cv, &lock);
pthread_cond_signal(&cv);                    pthread_mutex_unlock(&lock);
pthread_mutex_unlock(&lock);              }
                                          printf ("%d\n", y);
```

Solution: If Thread A executes first or starts executing while Thread B is sleeping on the condition variable, the printed value will be 2. If Thread B executes first, checks the while loop condition, and then switches context before acquiring the lock, Thread A may execute and complete. Now, when Thread B is scheduled, it will keep sleeping on the condition variable forever. Thus, the second possibility is a deadlock with Thread B printing nothing.

To avoid the second possibility, Thread B should acquire the lock before checking the while loop condition. By not doing so, it has introduced a data race between the read of `x` and the write of `x`.

Grading policy: One mark for clearly mentioning each possibility.