Rashed Abdulla Alyammahi

1. Race condition is a situation in which concurrently executing process accessing the same snared data item may affect the consistency of the data item.

highestBid variable is snared data item between two or more processes. Here race condition is possible to occur.

Example: suppose current highestBid=50

Bider 1 amount = 51

Bider 2 amount = 52

If Bider1 && Bider2 execute void bid concurrently

If bider1 write finally highest bid =51

If bider2 write finally highest bid =52

=> data inconsistency, race condition occurs

It can be solved by accessing highestBid mutually exclusively. Acquired lock on highestbid.

Modified program:

```
Void bid (double amount){
 acquire();
            If(amount>highestbid){
                 Highestbid = amount;}
 release();
}
```

2.

a) Race condition would come when any process tries to run statements

number_of_processes++; number_of_processes--;

More than one process can execute the above code in interleaved manner, resulting in race condition

b) To prevent race condition

Make sure only 1 process can access number_of_processes variable at a time.

```
pthread_mutex_lock( &mutex )
if (number_of_processes == MAX_PROCESSES) {
return -1;
```

} else {

/ allocate necessary process resources */*

number_of_processes++;

return new_pid;

}

pthread_mutex_unlock(&mutex)

pthread_mutex_lock(&mutex)

number_of_processes--;

pthread_mutex_unlock(&mutex)

3.

```
pthread_mutex_t  mutex= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond  = PTHREAD_COND_INITIALIZER;
int current_connection_num = 0;
#define MAX_CONNECTION 10
/* initialize the network capacity to n connections */
void
init(int n) ;

/* The purpose of this function is to block (not return) until
* n network connections are granted to the caller.
*
* On return, the caller assumes they are allowed to open n
* network connections.
*/
void
request(int n) {
        pthread_mutex_lock( &mutex )
        if (n >  MAX_CONNECTION) {
                pthread_mutex_unlock( &mutex )
```

```
                exit
        }
        while (n >  MAX_CONNECTION- current_connection_num)
                {pthread_cond_wait(&cond, &mutex);}
        init(n)
        current_connection_num +=  new_connection_num
        pthread_mutex_unlock( &mutex )
}

/* release n connections */
release(int n) {
        pthread_mutex_lock( &mutex )
        deinit(n)
        current_connection_num -=  n
        pthread_cond_signal(&cond);
        pthread_mutex_unlock( &mutex )
}
```

4.

```
sem_t seats;
sem_t chef;
void init() {
        /* provide any initialization */
        sem_init(&seats, 0, N); */ init semaphore with N seat */
        sem_init(&chef, 0, 1); */ init semaphore with only 1 chef */
}

void chef() {
   while (true) {
        /* wait for ready customer */
```

```
        sem_wait(&chef);
        printf("Welcome to the sushi bar.\n");
        /* take order */
        printf("What will you have?\n");
        /* serve customer */
        printf("Here is your food.\n");
        sem_post(&chef);
    }
}

void customer() {
    if (sem_trywait(&seats) == 0) {
        /* take seat if available, otherwise leave */
        sem_wait(&seats);
        printf("This seat looks good.\n");
        /* wait for chef */
        printf("Hello.\n");
        /* give order */
        printf("I will have one roll.\n");
        /* eat and leave */
        printf("That was good.\n");
        sem_post(&seats);
     } else {
         printf(“Full slots, leave food store\n”);
     }
}
```

5.

a) The four conditions necessary for deadlock to be possible are:

1. **mutual exclusion** - only one car can occupy each intersection at a time
2. **hold and wait** - cars can hold an intersection while waiting in a line for access to the next intersection
3. **no preemption** - cars cannot be removed from their spot in the traffic flow, except by moving forward
4. **circular wait** - the set of cars in the deadlock situation includes the cars in the middle of the intersection

b)

Use Traffic Lights - only one direction of traffic can move in the intersections for a set interval of time -- the direction alternates.

6.

Suppose the system is in deadlock situation. there are only 3 resources of the same type but there are 2 threads, each thread requires 2 resources. So there will be 1 thread holding 2 resources and 1 holding 1 resource and waiting for resource. So it can’t lead to deadlock.

7.

a. While P0 holds resource S and waits for resource Q to execute, P1 holds resource Q and needs S resource to execute, a deadlock will occur.

b.

FCFS

| P0 | P1 |

0ms 20ms 40ms

RR (quanta = 5ms)

| P0 | P1 | P0 | P1 | P0 | P1 | P0 | P1 |

0 5 10 15 20 25 30 35 40

RR algorithm is more prone to deadlock because unlike FCFS algorithm, process P0 goes in to get 2 resources S and Q and uses it to burst out of burst time, then returns 2 resources for P1 to use for execution. In the RR algorithm, the processes take turns running P0 running for 5ms, then returning resources to P1 to execute, between the turns, there is a chance that there is a problem that causes a deadlock to occur.