Homework Problems on Concurrency

1. Race conditions are possible in many computer systems. Consider an online auction system where the current highest bid for each item must be maintained. A person who wishes to bid on an item calls the bid(amount)function, which compares the amount being bid to the current highest bid. If the amount exceeds the current highest bid, the highest bid is set to the new amount. This is illustrated below:

```
 void bid(double amount) {
   if (amount > highestBid) {
     highestBid = amount;
   }
}
```

Describe how a race condition is possible in this situation and what might be done to prevent the race condition from occurring.

There is a race condition on the variable highestBid. Suppose that two people wish to bid on a computer, and the current highest bid is $1,200. Two people then bid concurrently: the first person bids $1,300, and the second person bids $1,400. The first person invokes the bid() function, which determines that the bid exceeds the current highest bid. But before highestBid can be set, the second person invokes bid(), and highestBid is set to $1,400. Control then returns to the first person, which completes the bid() function and sets hightestBid to $1,300. The easiest way of fixing this race condition is to use a mutex lock:

```
void bid(double amount) {
    acquire(mutex);
    if (amount > highestBid)
        highestBid = amount;
    release(mutex);
}
```

2. Consider the code example for allocating and releasing processes.

```
 #define MAX_PROCESSES 255

 int number_of_processes = 0;

 /* the implementation of fork() calls this function */
```

```
int allocate_process() {
  int new_pid;
  if (number_of_processes == MAX_PROCESSES) {
    return -1;
  } else {
    /* allocate necessary process resources */
    number_of_processes++;
    return new_pid;
  }
}


/* the implementation of exit() calls this function */
void release_process() {
  /* release process resources */
  number_of_processes--;
}
```

a. Identify the race condition(s).
b. Assume you have a mutex lock named mutex with the operations lock() and unlock(). Indicate where the locking needs to be placed to prevent the race condition(s).

a. There is a race condition on the variable number_of_processes.

b. A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.

3. Assume a network is shared among different services, each of which wants to have multiple concurrently open TCP connections. The network sets a limit on the total number of open TCP connections. The purpose of the methods below is to keep track of the number of open connections so that the limit is never exceeded. For example, if a service want to open 5 new connections it must first call `request(5)`.

Implement the methods (pseudocode is fine) using only **locks** and **condition variables** to manage the concurrency.

```
/* 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);

/* release n connections */
release(int n);

cond_t cond;
mutex_m mutex;
int available;
```

```
void
init(int n) {
  // assuming this is called before the other threads are
  // created, it that is not then case then mutex lock needs to
  // be acquired before updating available
  available = n;
}

void
request(int n) {
  pthread_mutex_lock(&mutex);
  while (n > available) {
    pthread_cond_wait(&cond, &mutex);
  }
  available -= n;
  pthread_mutex_unlock(&mutex);
}

void
release(int n) {
  pthread_mutex_lock(&mutex);
  available += n;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mutex);
}
```

4. Consider the following sushi bar problem. There is a sushi bar with one chef and N seats. When a customer arrives, they take a seat at the bar if there is an empty one or they leave if no seats are empty. After taking a seat, they wait for the sushi chef to be available and then give their order. After the sushi chef serves them, they eat and leave. Write a solution to the sushi bar problem that uses only **semaphores** to manage the concurrency. You should write three procedures: init(), chef() and customer().

This problem shares some similarities to a classic problem we didn't cover in class, the sleeping barber problem.

https://en.wikipedia.org/wiki/Sleeping_barber_problem

```
sem_t customersReadyToOrder = 0; /* can be 0 up to N */

sem_t chefTakingOrder; /* will always be 0 or 1 */

sem_t customerGivingOrder; /* will always be 0 or 1 */

sem_t chefServingFood; /* will always be 0 or 1 */

sem_t seatsMutex = 0; /* only one thread at a time allowed

                         to read/write seatsAvailable */

int seatsAvailable = N; /* can be 0 up to N */


void

init() {

    sem_init(customersReadyToOrder, shared, 0);
    sem_init(chefTakingOrder, shared, 0);

    sem_init(customerGivingOrder, shared, 0);

    sem_init(chefServingFood, shared, 0);
    sem_init(seatsMutex, shared, 0);
    int seatsAvailable = N;
}


void chef() {

    while (true) {

        sem_wait(customersReadyToOrder); /* wait for at least one

                                             customer to be ready */
        printf("Welcome to the sushi bar.\n");
        /* take order */
        sem_post(chefTakingOrder);
        printf("What will you have?\n");
        sem_wait(customerGivingOrder);
        /* serve customer */
        printf("Here is your food.\n");
        sem_post(chefServingFood);
    }
}

void customer() {
    sem_wait(seatsMutex); /* get exclusive access to seatsAvailable
                             to prevent race conditions */
    if (seatsAvailable == 0) {
        sem_post(seatsMutex); /* release the lock */
        printf("I'm not waiting around, bye.\n");
        return;
    }
    printf("This seat looks good.\n");
    seatsAvailable--; /* customer takes an available seat */
    sem_post(seatsMutex); /* release the lock */

    sem_post(customersReadyToOrder); /* customer signals they are
                                        ready to order */
    sem_wait(chefTakingOrder); /* customer waits, the chef might be
                                  serving other customers */

    printf("Hello.\n");

  printf("I will have one roll.\n");
    sem_post(customerGivingOrder);
    sem_wait(chefServingFood);
    printf("That was good.\n");

    set_wait(seatsMutex); /* get exclusive access to seatsAvailable
                             to prevent race conditions */
    seatsAvailable++;
    sem_post(seatsMutex); /* release the lock */
}
```