CSE 6431: Homework I

1) Suppose a banking system stores all users' balance in an array, and uses the following logic to transfer balance from one user to another. The program was originally designed as a single-threaded program so that transfer is only called by one thread.

```
int balance[NO_USERS];

void transfer(int source, int target, int amount){
    if(balance[source] > amount) {
        balance[source] -= amount;
        balance[target] += amount;
    }
}
```

Now the programmer wants to execute transfer with multiple threads, but finds with a global lock, the performance of this program is not satisfactory, so s/he wants to try fine-grained locking by using a lock for each user's balance. The following is his/her code.

```
int balance[NO_USERS];
pthread_mutex_t locks[NO_USERS];

void transfer(int source, int target, int amount){
    pthread_mutex_lock(&locks[source]);
    pthread_mutex_lock(&locks[target]);
    if(balance[source] > amount) {
        balance[source] -= amount;
        balance[target] += amount;
    }
    pthread_mutex_unlock(&locks[target]);
    pthread_mutex_unlock(&locks[source]);

}
```

Does this code have any problem? If so, can you fix it? Note "source" and "target" are set by a user, so they could be any value at running time. You can assume the system has already checked the validity of source and target, source is not equal to target, and locks are properly initialized.

2) Try to find bugs related to multi-threading. You can assume all locks and condition variables have been properly initialized. Write down your corrected version.

a) The following code asks thread 1 to wait for thread 2 to finish its task first. These two tasks have their own synchronization mechanisms, so you don't need to worry about them.

```
void *thread1(void *arg){
   pthread_mutex_lock(&lock);
   pthread_cond_wait(&cond, &lock);
   pthread_mutex_unlock(&lock);
   //perform thread1's task
}

void *thread2(void *arg){
   //perform thread2's task
   pthread_mutex_lock(&lock);
   pthread_cond_broadcast(&cond);
   pthread_mutex_unlock(&lock);
}
```

b) In the following program, thread 1 releases 3 tickets; thread 2 tries to get 2 tickets; thread 3 tries to get 5 tickets. If the program is correct, then thread 2 should get 2 tickets and thread 3 should be waiting.

```
void *thread1(void *arg){
   pthread_mutex_lock(&lock);
   ticket += 3;
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&lock);
}

void *thread2(void *arg){
   pthread_mutex_lock(&lock);
   if(!(ticket>=2))
      pthread_cond_wait(&cond, &lock);
   ticket-=2;
   pthread_mutex_unlock(&lock);
}

void *thread3(void *arg){
   pthread_mutex_lock(&lock);
   if(!(ticket>=5))
      pthread_cond_wait(&cond, &lock);
   ticket-=5;
   pthread_mutex_unlock(&lock);
}
```

3) Implement a solution with writers' priority to the readers/writers problem using semaphores.

4) Write a monitor-based solution to the reader–writers problem that works as follows: If readers and writers are both waiting, then it alternates between readers and writers. Otherwise, it processes them normally, i.e., readers concurrently and writers serially.

5) A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: The sum of all unique numbers associated with all the processes concurrently accessing the file must be less than n. Write a monitor to coordinate accesses to the file.

6) Using Java support for multithreading (synchronized, wait, and notifyall), write a solution to the producer-consumer problem with a buffer of length N.

7) Using Java support for multithreading (synchronized, wait, and notifyall), write a solution to the readers-writers problem, with exclusive writer access, concurrent reader access, and reader's priority.