Stamatios Morellas
COM S 352 - Homework 5
Due 3/6/20 @ 11:59pm

6.8 (10 points)

Race conditions occur when multiple running processes attempt to access a shared set of data and modify it at the same time. So a situation where a race condition could potentially occur would be if the value of `highestBid = 20` and there were two people that called the `bit(amount)` function, concurrently; Person one's amount is 21 and Person two's amount is 22. The value of `highestBid` would now have two different values. A way to prevent this would be to modify `highestBid` from being a global variable to being a mutually exclusive variable for each process.

6.13 (15 points)

The three requirements for the critical-section problem are *mutual exclusion*, *progress*, and *bounded waiting*. The program is mutually exclusive because only one process is able to execute its critical section at a time. Since the `while` loop in the entry section is one of the only places that could cause a problem, the `if` statement prevents this from happening. A set of processes is said to enter a 'deadlocked' state when every process in the set (except one) is waiting for an event to occur from another process (the remaining process). This is also prevented from happening because turn is always changed after the critical section, eliminating potential deadlocking. For the third requirement, let's assume that P0 is a starving process and P1 is in its critical section. P0 is guaranteed access to its critical section after P1 is finished executing because of turn being changed after the critical section is finished executing, also stated previously, which means that the third requirement is also satisfied.

6.16 (10 points)

```
typedef struct {
    int available;
} lock;

lock *mutex;
mutex->available = 0; // initialize as available

void acquire(lock *mutex) {
    while(compare_and_swap(&mutex->available, 0, 1) != 0);
    return;
}
```

```
void release(lock *mutex) {
    mutex->available = 0;
    return;
}
```

6.19 (10 points)

Lock held for short duration – **Spinlocks** are a better choice for this case because they are more efficient than mutex locks when threads are only being blocked for short periods of time.

Lock held for long duration – A **mutex lock** should be used in this scenario because so that other processes can be scheduled to run after the locked process is finished executing

Thread put to sleep while holding lock – A **mutex lock** should be used so that the lock is able to hold for a longer period of time while the other thread is starting up, so the running thread does not execute at the same time.

6.22 (10 points)

(a) There did not appear to be any identifiable race conditions in the provided section of code.

(b) In order to prevent a race condition, the `acquire()` function should be the first operation called at the beginning of the mutex lock and the `release()` operation should be called right before exiting.