

Homework 5

1. Exercise 6.8 (10 points) 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.

2. Exercise 6.16 (10 points) Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

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

The value `(available == 0)` indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

3. Exercise 6.19 (10 points) Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where the waiting process sleeps while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

4. Exercise 6.22 (10 points) 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 `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).

5. Exercise 6.26 (10 points) Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.