Caleb Faruki
CISC3595 Operating Systems
Dr. Art Werschulz
November 6, 2013

Homework 6

6.6

Starting account balance = $100.00.
Husband calls withdraw(25), wife calls deposit(50). The husband's local balance becomes $75.00. Before this withdrawal is committed, the wife's deposit(50) happens. Making the balance $150.00. The wife's deposit() returns $150.00 and then the husband's withdraw() returns $75. Since deposit() returns last, the final value of the balance is $75.00. This is not the desired result. The new balance should be $125.00.

The way to fix this that once the balance is accessed by a function, lock it to that function until that lock is released, which is after the function terminates.

6.9

Busy waiting is when the process is waiting for a condition to be met in a loop without releasing the CPU. The alternative to this is to release the CPU and then block this process when the condition is eventually satisfied. Busy waiting isn't necessary. But the alternative – pausing a process only to wake it up later after the condition is satisfied – incurs overhead.

6.11

If a user process can disable interrupts, then timer interrupts might be disabled and this will prevent context-switching from occurring. This means that this user process could use the the CPU indefinitely, starving other processes.

6.12

If interrupts are disabled for some processors, then threads on other processors can ignore synchronization primitives and access shared data. If interrupts are disabled for all processors, then interrupts for handling important system tasks are disabled, crippling the computer and preventing further work from being done.

6.14

```
//lock = FALSE by default
do {
      waiting[i] = TRUE; // Pi ready to enter critical section
      key = TRUE; // key belongs to Pi
      while(waiting[i]&&key)
            key = swap(&lock, &key);
      waiting[i] = FALSE; // Pi not ready for crit sec
      j=(i+1) % n; // next process
      // critical section
      while((j!=i) && !waiting[j])
            j = (j+1)%n;
      if(j == i)
            lock = FALSE; // no process in critical section
      else
            waiting[j] = FALSE; // Pj not ready to enter crit sec
      // remainder section
} while(TRUE);
```

6.19

Use an incrementing semaphore initialized to 0. The parent thread would call wait(). When completed, the child thread invokes signal(). Using this method, the

child thread notifies the parent thread when the desired data is available.

6.25
Normally, if the signal() were the last statement, the signaling process has to release the lock and the signal() recipient has to vie for the lock with competing processes before it can make progress. This could be avoided/simplified by simply transferring the lock to the signal() receiver.

6.34
a. Intersections are shared resources and the lines of cars are processes.

Mutual exclusion: only 1 car may use the shared resource at a time.
No pre-emption: The resource can't be used by another process until the prior process has terminated (i.e. line of cars has passed).
Hold-and-wait: Each process is holding a resource and is waiting for the next resource.
Circular waiting: Each process is waiting for process ahead of it to complete.

b. Breaking any one of these conditions will prevent deadlock. One example: a process can't hold a resource indefinitely. That is, a line of cars can't sit in the intersection. They must release the resource after a timeout or after the process is terminated, whichever comes first.