Jeffrey Everett
CSCI 3753
3/3/2017

Problem Set 2

Question 1:
We will use the following semaphores:

```
semaphore westmutex = 1;
semaphore eastmutex = 1;
semaphore west = 1;
semaphore east = 1;
```

The following program is for the west-bound cars:

```
int westcount = 0;

while (1) {
        wait(westmutex);
        westcount++;
        if (westcount == 1) wait(east);
        signal(westmutex);
        // do actual driving here
        wait(westmutex)
        westcount--;
        if (westcount == 0) signal(east);
        signal(westmutex);
}
```

And the below program is for east-bound cars:

```
int eastcount = 0;

while (1) {
        wait(eastmutex);
        eastcount++;
        if (eastcount == 1) wait(west);
        signal(eastmutex);
        // do actual driving here
        wait(eastmutex)
        eastcount--;
        if (eastcount == 0) signal(west);
        signal(eastmutex);
}
```

The semaphore eastmutex protects access to the eastcount variable, westmutex protects access to the westcount variable, west prevents west-bound cars from driving when east-bound cars are on the bridge, and east prevents east-bound cars when west-bound cars are on the bridge.

There is possible starvation on both west-bound and east-bound cars because they must wait indefinitely if there is an indefinite stream of east-bound or west-bound cars, respectively.

Note that this code has been adapted from the 1st Readers/Writers Solution in the slides for Section 5.3.

Question 2:
The condition variable would be defined as follows:

```
condition wait_on_T1;
lock mutex;
int t1_completed = 0;
```

The code for C1 is as follows:

```
// complete task 1
acquire(mutex);
t1_completed = 1;
wait_on_T1.signal();
release(mutex);
```

And the code for C2 is below:

```
acquire(mutex);
if (t1_completed == 0) wait_on_T1.wait();
release(mutex);
// complete task 2
```

This works because C2 will only wait if T1 has not yet been completed, and in that case there will still be a signalling incoming to take it out of its waiting state.

Question 3:
The semaphores in the 3rd Readers/Writers problem have the following roles:

- mutex: protects access to the readcount variable
- wrt: prevents writers from writing while readers are reading
- readBlock: queues up readers and writers in the same queue.

The wrt semaphore works by preventing writers from writing until all readers that have gotten past readBlock to performed their operation. This is done by keeping track of the readcount, and waiting wrt when readcount is equal to 1 (the first reader begins reading) and signalling wrt when readcount is equal to 0 (the last reader stops reading).

The most complicated semaphore is the last one listed, readBlock. Readers signal this semaphore before performing their actual function, while writers signal this semaphore after performing their write operations. This means that readers can read concurrently but writers cannot write concurrently. The semaphore also has the function of putting readers and writers into the same queue. If a writer shows up while the readers are reading, they will not signal readBlock until after it has completed writing, which it cannot begin until all of the existing readers have finished reading. Thus, if new readers emerge, they will get stuck at wait(readBlock), which means that they will not begin reading until the writer has finished writing. This is what prevents starvation of the writers. If a reader shows up while a writer is writing, it will be queued at the next position of the semaphore queue. New writers that show up will not have priority over those readers, so eventually the queue position of the reader will be reached and it will begin to read. This is what prevents starvation of the readers.

Thus, the solution is starvation-free.

Question 4a:
No, this function is not thread-safe. This is because there is no mutual exclusion around the global variable temp.

As an example of where it could be messed up, imagine that Thread A and Thread B are running the swap function. If Thread A sets temp and then context switches to Thread B before temp is used again, and Thread B runs part of the swap function to where temp is changed but not restored, then temp is overridden and when the processor returns to Thread A, temp will contain a different value than intended, so when the line “*z = temp” is reached, z will refer to the incorrect value.

Question 4b:
It is reentrant because the swap function restores the state of the global variable and does not use any locks or semaphores. Thus, if the first function is interrupted to run the second function, and that function runs completely, the first function will not be affected.

Question 5:
Shared memory would be the best form of IPC to allow P2 to read the file written by P1. This is because the 10MB file is quite large and it would be inefficient to use other forms of IPC such as message passing over UNIX or Internet sockets.

To inform P2 that P1 has finished reading, the best form of IPC would be signalling. This is because signalling is relatively simple and there is no data required in the message, as all the data resides in shared memory.

Question 6:
The monitor for V1 is as follows:

```
monitor mon_V1 {
       V1 declaration

       function increment() { V1++; }
       function decrement() { V1++; }
}
```

The monitor for V2 is:

```
monitor mon_V2 {
       V2 declaration

       function square() { V2 = V2*V2; }
       function squareroot() { V2 = sqrt(V2); }
}
```

And the monitor for V3 is:

```
monitor mon_V3 {
       V3 declaration

       function sin() { V3 = sin(V3); }
       function cos() { V3 = cos(V3); }
}
```

There are three monitors created because that allows one variable to be edited (by exactly one process) at the same time that another variable is edited, thus increasing efficiency.