1.)
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void *printHello(void * x)
{
   printf("Hello World!\n");
   return NULL;
}

int main(int argc, char * argv[])
{
  int nthreads = atoi(argv[1]);
  pthread_t tid[nthreads];
  for(int i = 0; i < nthreads; ++i)
  {
    if(pthread_create(&tid[i], NULL, printHello, NULL) < 0)
    {
      printf("Error creating thread\n");
      exit(1);
    }
  }

  for(int i = 0; i < nthreads; ++i)
  {
    pthread_join(tid[i], NULL);
  }

}
```

2.) Main thread needs to call pthread_join(tid, NULL); to wait for thread to finish.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *thread(void *vargp);

int main()
{
  pthread_t tid;
```

```
  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);

}

void *thread(void *vargp)
{
  sleep(1);
  printf("Hello, world!\n");
  return NULL;
}
```

3.) Replace pthread_exit(NULL) call with the following call:

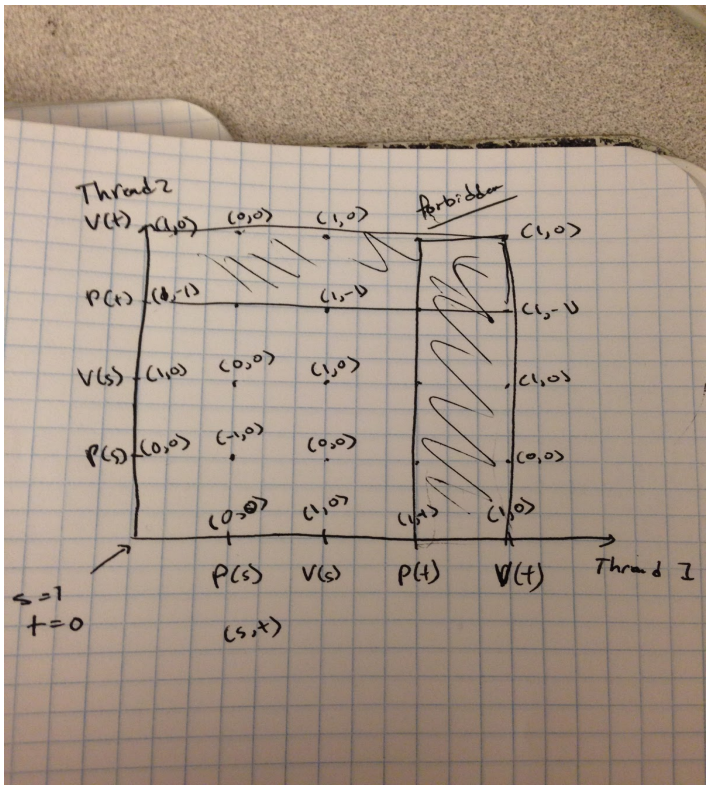
```
for(t = 0; t < NUM_THREADS; ++t)
{
    pthread_join(threads[t], NULL);
 }
```

Fixed code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8

void *PrintHello(void *threadid)
{
  long taskid = (long)threadid;
  sleep(1);
  printf("Hello from thread %ld\n", taskid);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;

  for(t = 0; t < NUM_THREADS; ++t)
  {
    printf("Creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);
```

```
    if(rc)
    {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  for(t = 0; t < NUM_THREADS; ++t)
  {
    pthread_join(threads[t], NULL);
  }
}
```

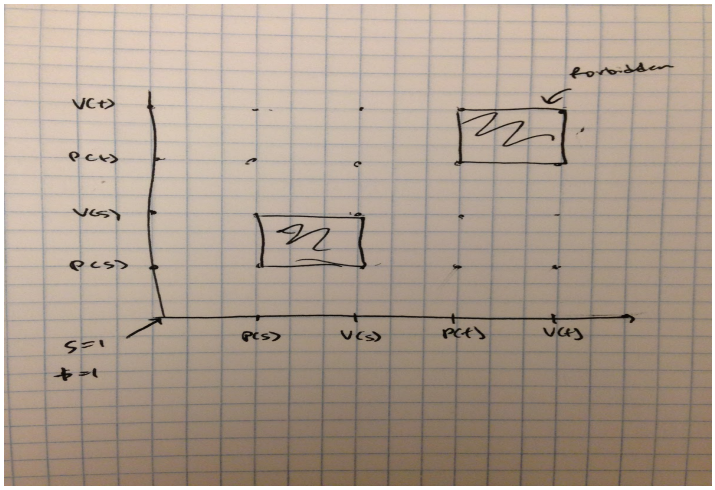
4.) a.)



b.) Yes, because P(t) will always block since t is initialized to 0

c.) Initialize t to 1

d.)



5.) It cannot deadlock because the threads don't hold the same semaphores at the same time.

6.)

a.)

Thread 1: a < b and a < c

Thread 2: c < b

Thread 3: b < a

b.) Thread 2 and Thread 3 violate this rule.

c.)

Thread 1:

N/A

Thread 2:

P(b);
P(c);
V(c);
V(b);
P(a);
V(a);

Thread 3:

P(c);
V(c);
P(a);
P(b);
V(b);
V(a);

7.) a.) The mutex is necessary in both sbuf_insert and sbuf_remove since more than one item can be in the buffer

b.) The mutex is not needed since there can only be one item in the buffer. In this scenario, the items semaphore will make sure that the buffer isn't being read and written at the same time.

c.) The mutex is not needed since there can only be one item in the buffer. In this scenario, the items semaphore will make sure that the buffer isn't being read and written at the same time.

8.)

```
sem_t w = 1;
sem_t count = N;

//Reader
P(count);
//read
V(count);

//Writer
P(w)
for(int i = 0; i < N; ++i)
        P(count);

for(int i = 0; i < N; ++i)
        V(count);
```

9.) The performance of the sequential program is much faster than that of the parallel program. This is because of the way that my matrix multiplication is implemented. When using threads, my program spends more time switching between threads than it does actually doing the computation. Code is included in files: q9.c,

Instructions to run:

1.) Create matrix file

./create_matrices.py

2.) Compile source code

gcc q9.c -pthread

3.) Time program passing matrices file as input. This is the sequential version

time ./a.out matrices

4.) Time program passing matrices file as input with an additional argument. This is the parallel version

time ./a.out matrices -p