CSCI 350 Fall 2021 Project 2
Due October 1st by 11:59pm

Objectives:

In this project, you will extend xv6 to support:

1. Kernel level threads.
2. a Synchronization Primitive: mutex

Sounds like fun? Well, it should. Because you are on your way to becoming a real kernel hacker. And what could be more fun than that?

Task 1: Kernel level threads (KLT)

In this task, you will make changes to xv6 to add support for kernel threads to the OS. Once this part is done you will need to create the system calls that will allow the user to create and manage new threads. Kernel level threads (KLT) are implemented and managed by the kernel. A single process may execute multiple kernel threads, which may operate concurrently on a multi-core machine. Each process contains its own static array of threads. The maximum number of threads each process can hold is defined in *kthread.h*.

Clone your Project 2 repo to your 32 bit Ubuntu VM. DO NOT USE xv6 from MIT repo.

The xv6 code for this assignment already includes some significant refactoring. However, the current thread implementation is missing some important functionality. Your job is to add additional thread management facilities and enforce correct thread semantics on existing system calls.

Supporting multiple threads per process means that you will have to think about handling shared resources. For example: ***growproc*** is a function in *proc.c* which is responsible for retrieving more memory when the process asks for it. If not protected, 2 running threads of the same process calling this function may override the *sz* parameter and cause issues in the system. You need to synchronize all shared fields of the thread owner when they are accessed. Be prepared to explain why you did or didn't synchronize those accesses in your report.

1. Thread system calls

In this part of the assignment you will add system calls that will allow applications to create KLTs. Use the file named *kthread.h* to contain the prototypes of the KLT API functions but implement them in *proc.c*.

The API for thread system calls is as follows:

```
int kthread_create(void*(*start_func)(), void* stack, int stack_size);
```

Calling *kthread_create* will create a new thread within the context of the calling process. The newly created thread state will be *TRUNNABLE*. The caller of *kthread_create* must allocate a user stack for the new thread to use (it should be enough to allocate a single page i.e., 4K for the thread stack). This does not replace the kernel stack for the thread.
start_func is a pointer to the entry function, which the thread will start executing. Upon success, the identifier of the newly created thread is returned. In case of an error, a non-positive value is returned.
The kernel thread creation system call on real Linux does not receive a user stack pointer. In Linux the kernel allocates the memory for the new thread stack. You will need to create the stack in user mode and send its pointer to the system call in order to be consistent with current memory allocator of xv6.

```
int kthread_id();
```

Upon success, this function returns the caller thread's id. In case of error, a non-positive error identifier is returned. Remember, thread id and process id are not identical.

`void kthread_exit();`

This function terminates the execution of the calling thread. If called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process. If it is the last running thread, process should terminate. Each thread must explicitly call *kthread_exit()* in order to terminate normally.

`int kthread_join(int thread_id);`

This function suspends the execution of the calling thread until the target thread (of the same process), indicated by the argument *thread_id*, terminates. If the thread has already exited, execution should not be suspended. If successful, the function returns zero. Otherwise, -1 should be returned to indicate an error.

2. Adding threads to the kernel

In the previous part you made the system calls for kernel level threads, but in order to integrate them we need to update existing system calls to allow for kernel level threads.

Here is an ***inexhaustive*** *list* of the expected behavior of existing system calls after you have finished this part (*you may not have to make modifications to all of them*):

Fork – should duplicate only the calling thread, if other threads exist in the process they will notexist in the new process.

Exec – should start running on a single thread of the new process. Note that in a multi-threaded environment a thread might be running while another thread of the same process is attempting to perform *exec*. The thread performing *exec* should "tell" other threads of the same process to destroy themselves and only then complete the *exec* task.

Exit – should kill the process and all of its threads, remember while a single threads executing *exit*, others threads of the same process might still be running.

Task 2: Synchronization Primitive: Mutex

In this task you will implement two synchronization primitives: mutex and condition variable.

Task 2.1 Preparation (Very Important):

1. Read about mutexes and condition variables in POSIX, preferably before you begin coding for this part of the assignment. Your implementation will emulate the behavior and API of *pthread_mutex* and *pthread_cond*. A nice tutorial is provided by Lawrence Livermore National Laboratory. **Read this carefully and give yourself some time to digest it. You will have a far easier time with the rest of the assignment if you do so.**

2. Examine the implementation of spinlocks in xv6's kernel (for example the scheduler uses a spinlock). Spinlocks are used in xv6 in order to synchronize kernel code. Your task is to implement the required synchronization primitives as a kernel service to applications, via system calls. Locking and releasing can be based on the implementation of spinlocks to synchronize access to the data structures which you will create to support mutexes and condition variables.

Quoting from the pthreads tutorial by Lawrence Livermore National Laboratory:

> "Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur. A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex, as used in pthreads, is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful, and all other threads will get blocked until the mutex becomes available. Mutex differs from a binary semaphore in its unlock semantics - only the thread which locked the mutex is allowed to unlock it, unlike binary semaphores on which any thread can perform up."

Task 2.2 Implementation:

Examine *pthreads*' implementation, and define the type *kthread_mutex_t* inside the xv6 kernel to represent a mutex object. You can use a global static array to hold all the mutex objects in the system. The size of the array should be defined by the macro *MAX_MUTEXES=64*.

The API for mutex system calls is as follows:

```
int kthread_mutex_alloc();
```

Allocates a mutex object and initializes it; the initial state should be unlocked. The function should return the ID of the initialized mutex, or -1 upon failure.

```
int kthread_mutex_dealloc(int mutex_id);
```

De-allocates a mutex object which is no longer needed. The function should return 0 upon success and -1 upon failure (for example, if the given mutex is currently locked).

```
int kthread_mutex_lock(int mutex_id);
```

This function is used by a thread to lock the mutex specified by the argument *mutex_id*. If the mutex is already locked by another thread, this call will block the calling thread (change the thread state to *TBLOCKED*) until the mutex is unlocked.

```
int kthread_mutex_unlock(int mutex_id);
```

This function unlocks the mutex specified by the argument *mutex_id* if called by the owning thread, and if there are any blocked threads, one of the threads will acquire the mutex. An error will be returned if the mutex was already unlocked. The mutex may be owned by one thread and unlocked by another!

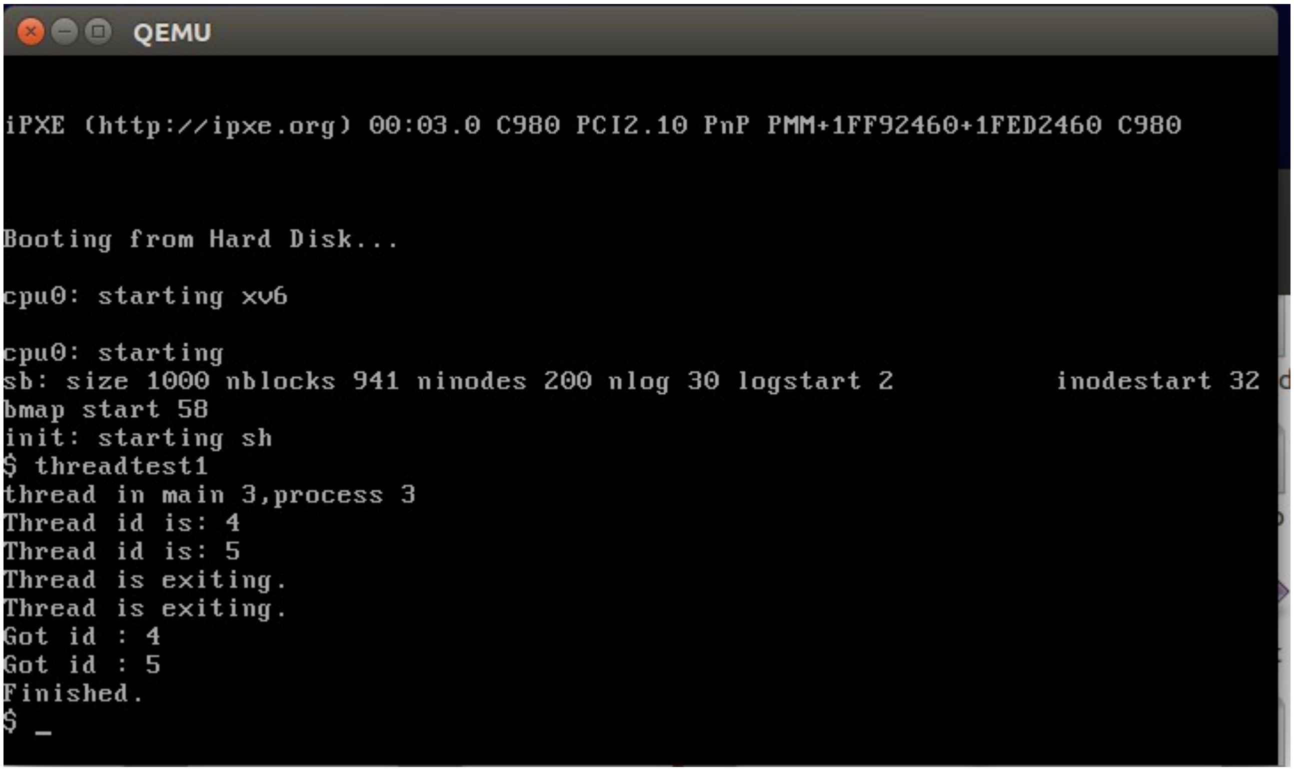
Implementation notes:

- In all the above functions, the value 0 should be returned upon success, and a negative value upon failure.
- No busy waiting should be used whatsoever, except for synchronizing access to the mutex array using short-time waiting in spinlock.

Testing your code

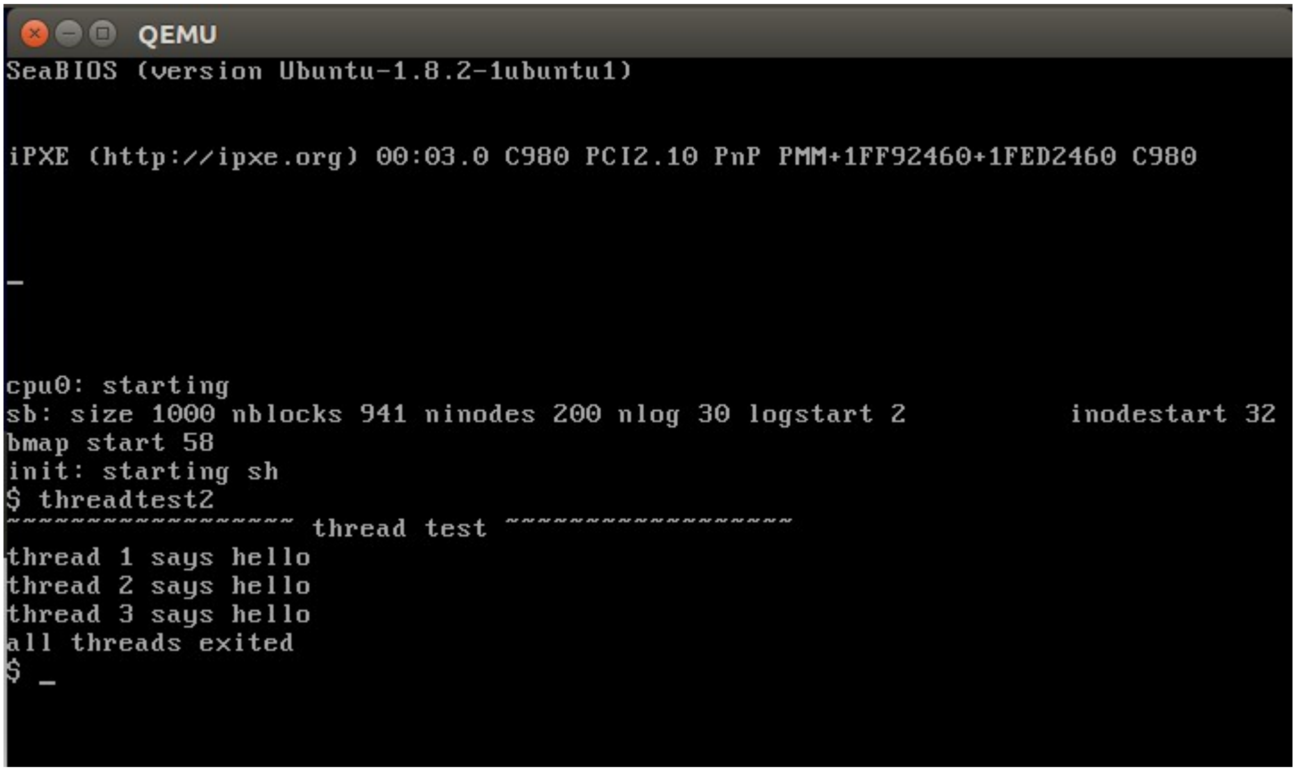
The code distribution contains 5 test files: *threadtest1.c, threadtest2.c, threadtest3.c, mutextest1.c* and *mutextest2.c.* Most of the points on this assignment will be based on your code matching the expected outputs provided below.

threadtest1:



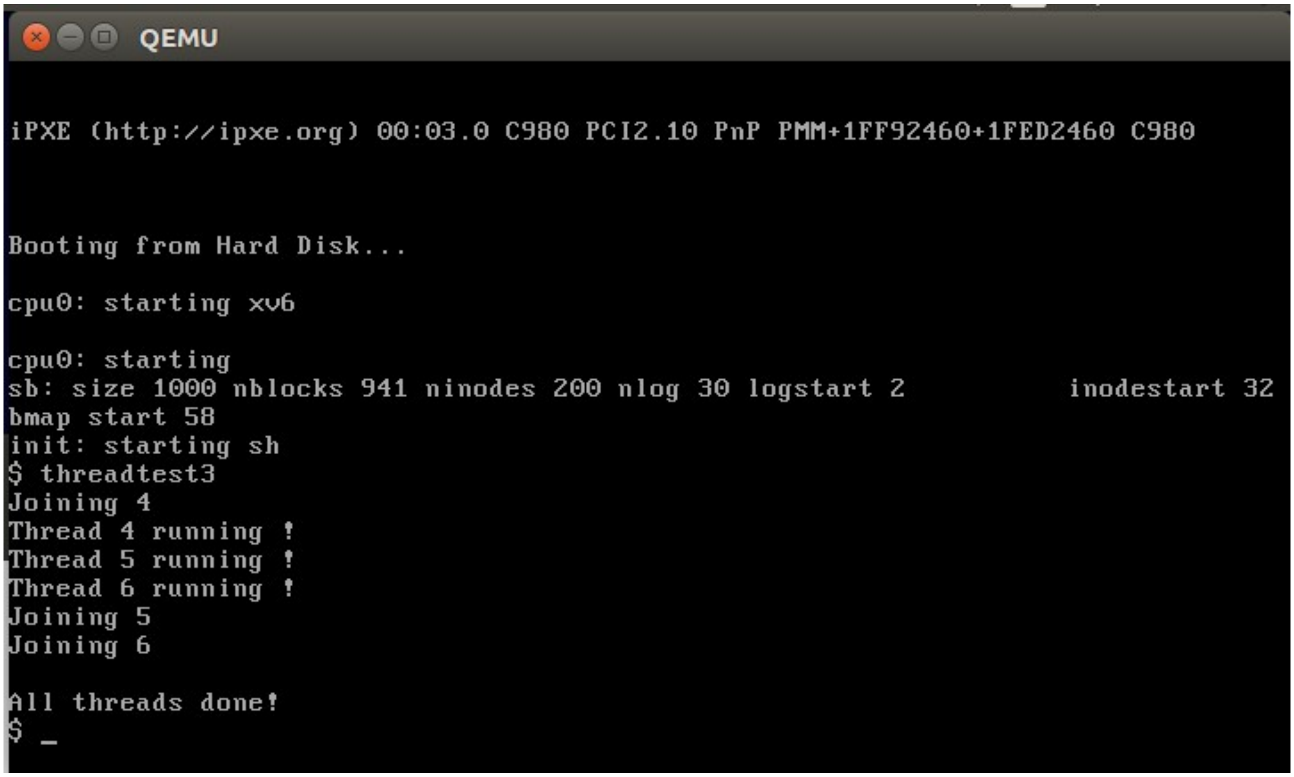
```
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980



Booting from Hard Disk...

cpu0: starting xv6

cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32
bmap start 58
init: starting sh
$ threadtest1
thread in main 3,process 3
Thread id is: 4
Thread id is: 5
Thread is exiting.
Thread is exiting.
Got id : 4
Got id : 5
Finished.
$ _
```

threadtest2:



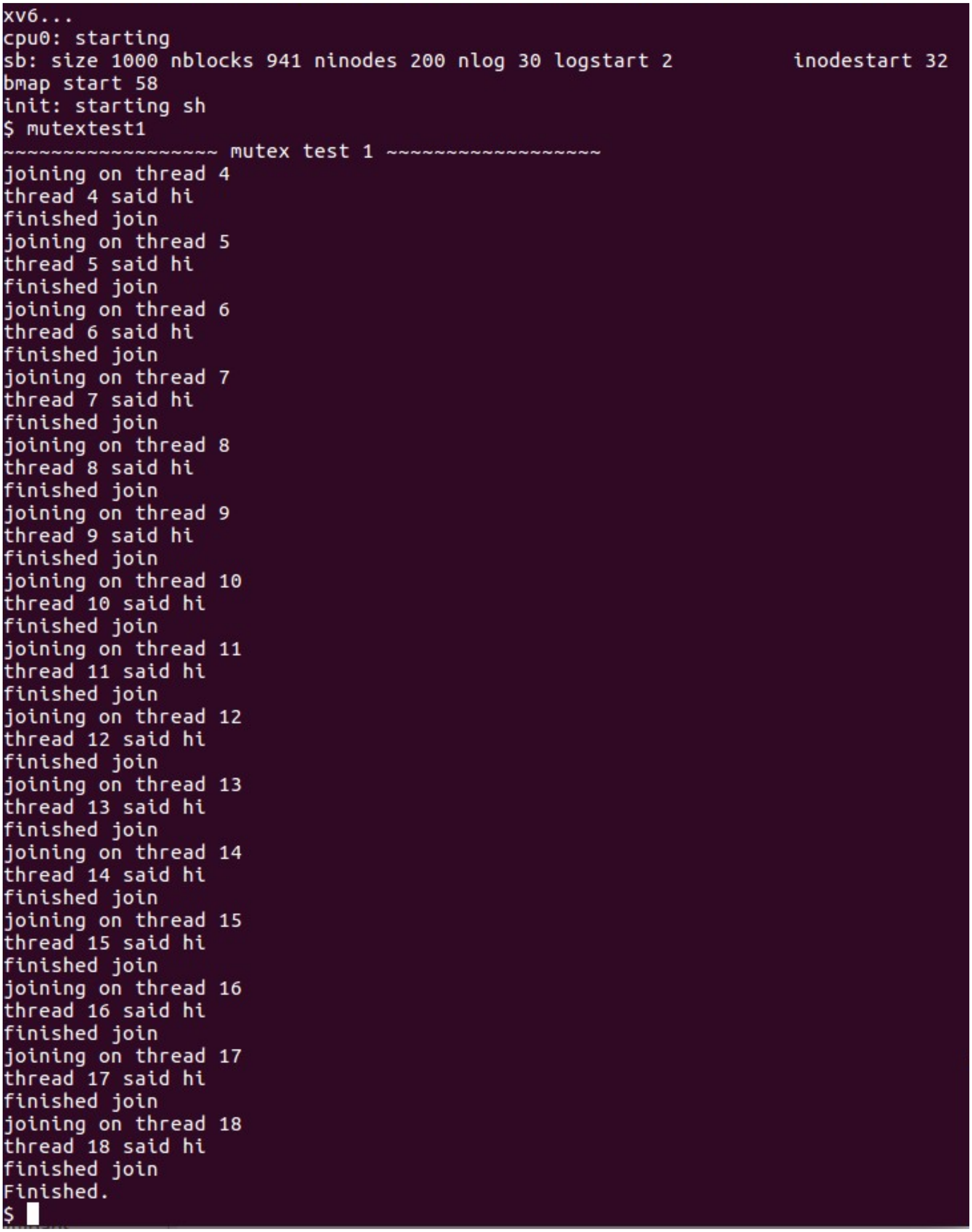
```
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980



_



cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32
bmap start 58
init: starting sh
$ threadtest2
~~~~~~~~~~~~~~~~~~ thread test ~~~~~~~~~~~~~~~~~~
thread 1 says hello
thread 2 says hello
thread 3 says hello
all threads exited
$ _
```

threadtest3:



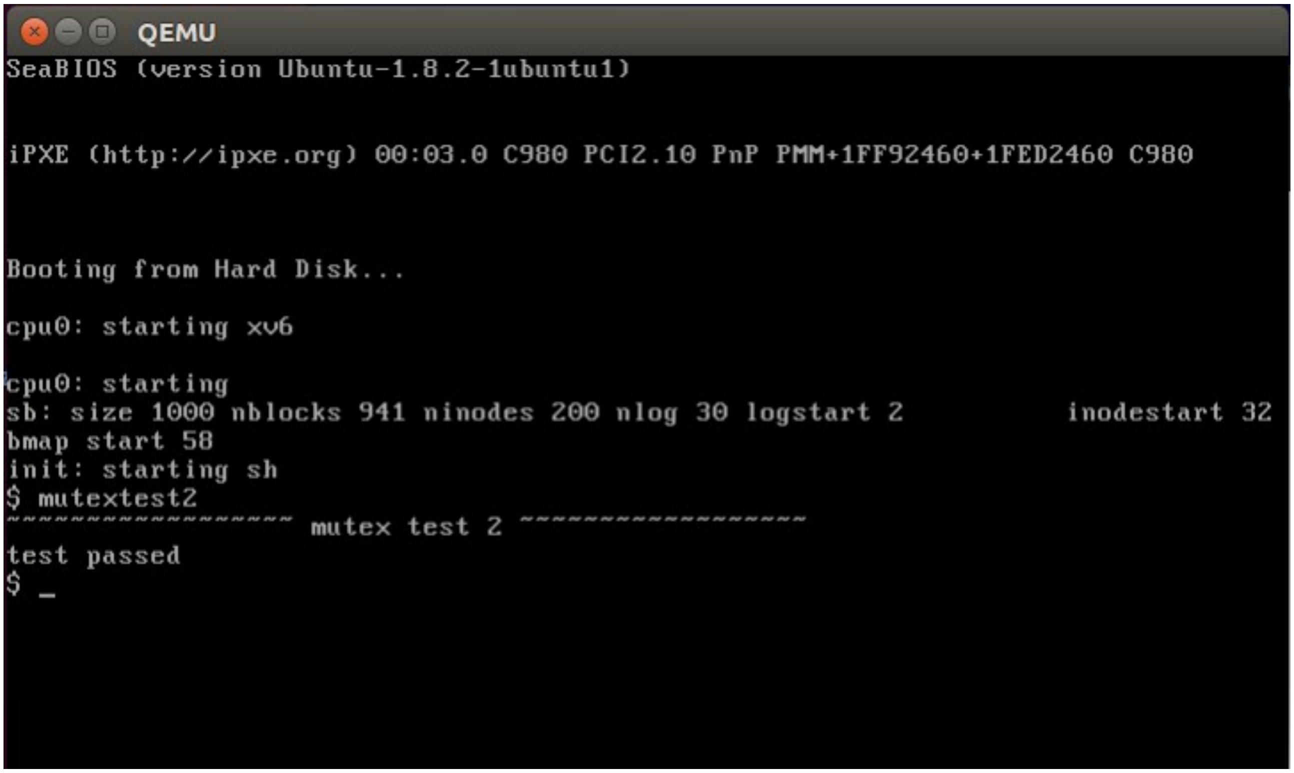
```
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980



Booting from Hard Disk...

cpu0: starting xv6

cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32
bmap start 58
init: starting sh
$ threadtest3
Joining 4
Thread 4 running !
Thread 5 running !
Thread 6 running !
Joining 5
Joining 6

All threads done!
$ _
```

mutextest1:



```
xv6...
cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32
bmap start 58
init: starting sh
$ mutextest1
~~~~~~~~~~~~~~~~~~ mutex test 1 ~~~~~~~~~~~~~~~~~~
joining on thread 4
thread 4 said hi
finished join
joining on thread 5
thread 5 said hi
finished join
joining on thread 6
thread 6 said hi
finished join
joining on thread 7
thread 7 said hi
finished join
joining on thread 8
thread 8 said hi
finished join
joining on thread 9
thread 9 said hi
finished join
joining on thread 10
thread 10 said hi
finished join
joining on thread 11
thread 11 said hi
finished join
joining on thread 12
thread 12 said hi
finished join
joining on thread 13
thread 13 said hi
finished join
joining on thread 14
thread 14 said hi
finished join
joining on thread 15
thread 15 said hi
finished join
joining on thread 16
thread 16 said hi
finished join
joining on thread 17
thread 17 said hi
finished join
joining on thread 18
thread 18 said hi
finished join
Finished.
$ █
```

mutextest2:



```
SeaBIOS (version Ubuntu-1.8.2-1ubuntu1)


iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+1FF92460+1FED2460 C980



Booting from Hard Disk...

cpu0: starting xv6

cpu0: starting
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2          inodestart 32
bmap start 58
init: starting sh
$ mutextest2
~~~~~~~~~~~~~~~~~~ mutex test 2 ~~~~~~~~~~~~~~~~~~
test passed
$ _
```

Free Response Questions (15 pts)

Please submit answers to the free response questions to Blackboard individually. This part will be more hands on this time, so assess your time carefully. Please refer to FRQ document under Piazza/ Resources or Blackboard.

Grading and Submission

Total Points: 100

- Free Response : 15 points
- Report: 10 points
  - Describe your changes to the xv6 code. Name the files your modified and briefly describe the changes
    - Clearly explain the changes you made as a part of Task1.1 to support:
      1. synchronization to shared fields
      2. expected behavior of existing system calls
- Correct implementation of synchronization of access to shared fields and, correct implementation of expected behavior of existing system calls: 15 points
- Passing test cases for threads: 30 points
  - threadtest1.c: 10 points
  - threadtest2.c: 10 points
  - threadtest3.c: 10 points
- Passing test cases for mutex: 30 points
  - mutextest1.c: 15 points
  - mutextest2.c: 15 points

Please do not modify any files you so not have to. This will only make grading harder.