Indian Institute of Technology, New Delhi, India

Assignment 3

COL331, COL633 and ELL405: Operating System

April 4, 2018

Things to keep in mind:

1. Use a fresh copy of XV6.
2. There are no test cases for this. You need to come up with your own test cases for this.
3. We will have demo of the assignment where you will explain your implementation.
4. There might be some other questions to test your understanding.
5. You will write your code in new files only. No existing files need to be touched.
6. You need to submit whole xv6 code after cleaning it (*make clean*).
7. MOSS will be run on the submitted files.
8. Please note that some parts of the assignment are dependent on each other. You should be able to demonstrate each part separately. Keep this in mind while coding.
9. Please follow the Naming Conventions as mentioned in the questions.

Release Date: 4th April 2018
Due Date:25th April 2018: 23:55
Max Marks: 50

1 User Level Threads: 10 marks (no part marking)

We will implement user level threads in XV6. As of now the xv6 does not support threads, which means you will have to come up with the design of the thread structure, keeping in the mind all the features which you need to support for the assignment.

If you are stuck and not sure from where to start, please check this link:
`https://pdos.csail.mit.edu/6.828/2012/homework/xv6-uthread.html`

1.1 Thread Structure

To create this you will need to know all the things thread needs for its operation e.g. context switching, preemption etc. You will also need to implement a stack for the thread, which will be used by the thread for its operations. Figure 1 gives the design for a basic structure of the thread. It can be defined like:

Listing 1: Simple Thread structure

```
struct thread {
  int         sp;                 /* curent stack pointer */
  char stack[STACK_SIZE];         /* the thread's stack */
  int         state;              /* running, runnable, waiting */
};
```

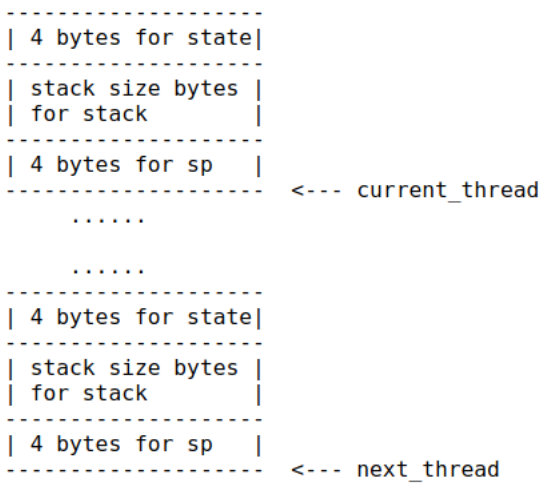

Figure 1: Basic structure of a thread

Please check the above mentioned link to see how a thread stack works.

1.2 Thread Init

You need to come up with some design decision like, how many threads will be allowed for a process. How much stack space will be allocated to each of the thread. Giving too little will not allow for some basic operations, giving too much will be a waste of space.

Listing 2: Design Choices

```
/* Possible states of a thread; */
#define FREE          0
#define RUNNING       1
#define RUNNABLE      2
#define WAITING      3

#define STACK_SIZE   8192
#define MAX_THREAD   4
```

1.3 Thread Creation

You need to have a function with the definition:

Listing 3: Thread Create

```
boolean thread_create(const char * name, *func)
```

which will basically create a thread with the name provided and a function pointer to the function which will be executed by the thread. Assume that the function (executed by the thread) will always return void.

thread_create will return a boolean value, depending upon whether the thread creation was successful or not.

1.4 Scheduler

For the scheduler part, we will follow a co-routine (co-operative) design. Here the thread themselves will *yield* the cpu and allow some other thread to run. This means there is no need to handle interrupts etc.
This will be a basic scheduler, where it will just pick the next *RUNNABLE* thread and schedule it.

Listing 4: Thread yield

```
void
thread_yield(void)
{
  current_thread->state = RUNNABLE;
  thread_schedule();
```

```
}
```

Listing 5: Finding the next runnable thread

```
static void
thread_schedule(void)
{
 /* Find another runnable thread. */
  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == RUNNABLE && t != current_thread) {
      next_thread = t;
      break;
    }
  }
  // Some other stuff
}
```

We will make the scheduler more intelligent in the next part.

1.4.1 Thread Switching

When the scheduler decides to switch two threads, it must *save* some information of the current running thread, so that in the future it can resume from the same point and not from the beginning. You need to decide what information to save and where to save it so that the design remains simple.
It also needs to *restore* the context of the new thread so that it can start executing.

Listing 6: Thread Switching

```
void thread_switch(void);
```

This function works similar to `swtch` function of xv6. When we enter this function, the old thread will be executing and after coming out of this function, the next thread will start executing.

Note: Please make sure that this is correct. You cannot do the rest of assignment till this works.

2 Synchronization: 15 marks

Once you get a thread running, the next part is synchronization. You need to implement a lock structure. You have to come up with a structure for the lock so that you can handle the requirements of the assignment easily.
Note: As this is a co-routine design, the threads can choose to not yield in the critical section. Please design a test case where the thread yields in the critical section.

2.1 Busy Waiting: 3 marks

Implement a busy waiting model: the thread which is trying to acquire the lock fails and yields immediately.

Listing 7: Lock Acquire: Busy Wait

```
lock_busy_wait_acquire(args);
```

2.2 NOT Busy Waiting: 12 marks

Here things will get more interesting. If the thread trying to acquire the lock fails it should change its STATE to *WAITING* before *yielding* so that the scheduler does not select it. However, as it will not be scheduled again its state needs to be changed by some other thread.
You need to decide **which** thread will change its STATE and **when**.
Keep the design general as more than one thread can be waiting for a single lock at a given time.

Listing 8: Lock Acquire: Not Busy Wait

```
lock_acquire(args);
```

Note: You need to have two functions to acquire the lock, one with the busy waiting and another with NOT busy waiting. Keep the functions name same.

3 Thread Priority: 25 marks

In this part we will add priority to the threads. The *thread_create* function will change accordingly:

Listing 9: Thread Create

```
boolean thread_create(const char* name, void *func, int priority)
```

Here, along with the name and the function pointer, we are also passing the priority of the thread.

3.1 Priority Scheduler: 3 marks

We will make changes to the scheduler so that it takes into account the priority of the thread.
Note: This will be a non-preemptive scheduler as this is a cooperative model and the running thread must call yield before it can be swapped out.
It can be designed in the following ways:

1. Maintain separate queues for different threads with different priorities and scan them from highest to the lowest priority.

2. Use a single queue, and while searching the list find the thread with the highest priority which is RUNNABLE.

3. Something else. You are free to design this.

Please note that if the priority is same, the scheduler should use FIFO scheduling policy.

3.2 Starvation: 10 marks

It might be possible that a low priority thread never gets to run because of other high priority threads. This can lead to starvation. You need to come up with a mechanism to prevent this. One solution is *aging*. But, again, you are free to choose whatever suits your design.

3.3 Priority Inversion: 2 marks

This part combines the synchronization scheme and the priority scheme. Recall *Priority Inversion* happens when a high priority process(thread) is waiting for a resource which is held by a process(thread) of low priority.
Design a test case to show that this problem persists in the current design. *Aging* and similar techniques for starvation prevention solves this, where the low priority thread will eventually run and release the resource. However we want something better than this, so that the high priority thread does not have to wait for very long.

3.4 Priority Donation: 10 marks

Priority donation is a technique to solve this. Here, the high priority process or thread donates its priority temporarily to the low priority process so that it gets scheduled, and releases the resource after finishing its operation.
You need to handle the following situations:

1. When the priority donation will happen?

2. How will the high priority thread know, which thread has the lock?

3. As the donation is temporary, the priority of the low priority thread needs to be restored after it is done with its operation.

4. When will the restoration be carried out?

5. Who will do the restoration?

Please keep in mind that as there can be multiple locks; it might be possible that a single thread has donated its priority to multiple threads.

You should be able turn this off, so that the test case generated to check the Priority Inversion part can run.