Assignment 6
Operating Systems (PG)

This assignment is divided into 2 parts:

[A] Pintos Priority Scheduling Implementation
[B] Linux Threads and Race Condition

Part A:

Objective:

To implement priority scheduling in Pintos kernel.

Details:

You need to handle two cases:

1. When a thread is added to the ready queue of the scheduler having higher priority than the currently running thread, then the current thread should yield the CPU making way for the newly added thread (i.e. preemptive strategy).
2. A thread may raise or lower its own priority at any time, but lowering its priority such that it no longer has the highest priority must cause it to immediately yield the CPU.

Thread priorities range from PRI_MIN (0 - lowest) to PRI_MAX (63 – highest). Initial thread priority can be set by passing an argument to thread_create(). Default priority is PRI_DEFAULT (31).

To implement:

Implement the following functions that allow a thread to examine and modify its own priority. Skeletons for these functions are provided in `threads/thread.c`.

Function: void **thread_set_priority** (int new_priority)

Sets the current thread's priority to new_priority. If the current thread no longer has the highest priority, yields.

Function: int **thread_get_priority** (void)

Returns the current thread's priority.

You need not provide any interface to allow a thread to directly modify other threads' priorities.

Part B:

Objective:

To understand the Linux threading mechanism and race conditions.

Details:

This assignment consists of 2 exercises. In the first exercise we will set up a race condition and then observe its nasty effects. This exercise will deal with POSIX Pthreads. The second exercise uses threads and mutexes to accomplish a good translation of consumer producer problem.

To implement (Exercise A):

The POSIX standard does not prescribe the exact concurrency requirements among the threads. Let us assume that the implementation supports concurrent threads with RR scheduling. Thus, all threads within a process proceed concurrently at unpredictable speeds.

To set up a race condition, we will create two threads, THREAD0 and THREAD1. The parent thread of the 2 threads – main( ) - defines two global variables ACCOUNT1 and ACCOUNT2. Each variable represents a bank account; it contains a single value – the current balance – which is initially zero. Each thread emulates a banking transaction that transfers some amount of money from one account to another. That means, each thread reads the values in the two accounts, generates a random number amount, adds this number to one account and subtracts it from the other.

```
counter = 0;
do {
temp1 = account1;
temp2 = account2;
amount = rand();
account1 = temp1 – amount;
account2 = temp2 + amount;
counter++;
} while (acount1+account2 == 0);
print(counter);
```

Both threads execute the same code. As long as the execution is not interleaved, the sum of the two balances should remain zero. However, if the threads are interleaved, one thread could read the old value of account1 and the new value of account2, or vice versa, which results in the loss of one of the updates. When this is detected, the thread stops and prints the step (i.e., the counter) at which this occurred. Write the code for main and the two threads. Then measure and report how long it takes to see the effect of the race condition.

To implement (Exercise B):

2 Processes A and B share the variables x and y. Process A writes x and reads y, while process B writes y and reads x. The two processes need to cooperate so that B does not read x until after A has written it, and so that A does not read y until B has written a new value to y. Use threads and mutexes to accomplish a good working translation of the given consumer - producer program.

```
procA {
while (TRUE) {
<compute section A1>
update(x);
<compute section A2>
retrieve(y);
}
}
procB {
while (TRUE) {
retrieve(x);
<compute section B1>
update(y);
<compute section B2>
}
}
```

Hints & Guidelines

1. Read the pthread man pages to become familiar with pthreads and mutexes. In particular, the man pages for pthread_create, pthread_exit, and pthread_join are useful when working with pthreads, and those of pthread_mutex_init and pthread_mutex_lock/unlock are useful when working with mutexes.
2. In order to compile your programs with the pthread library, add -lpthread to the gcc command line.

Deliverables

1. A README file which contains a list of items you turned in + in what fashion they are to be used.
2. All C programs

Upload Instructions:

1. Create a folder and name it as your ROLL NUMBER (2012*****)
2. Inside the above folder, create 2 more folders PART_A and PART_B
3. PART_A will contain your pintos/ folder
4. PART_B should contain 3 files.
    1. Exercise A's C code, <rollno_A.c>
    2. Exercise B's C code, <rollno_B.c>
    3. Read Me file
5. Archive the folder and name the file Assignment6.tar.gz

Example:

```
201106560
        |_________ PART_A
        |                  |__________ pintos/
        |
        |_________PART_B
                           |__________201106560_A.c
                           |__________201106560_B.c
                           |__________ReadMe.txt
```

Deadline: 30th October, Tuesday, 11:59PM