CC418 - Operating Systems
Lecturer: Dr. Ayman Adel
TA: Eng. Shereen Oraby
Term: Spring 2012

Sheet #7
PintOS Threads Project

Date Assigned: Week of Sunday, May 13th (Week 13)
Date Due: Week of Sunday, May 27th (Week 15). Late assignments will not be accepted. Submissions should be typed. Be sure to write your name, registration number, assignment number, and lecturer and TA name in the header.

---

http://www.scs.stanford.edu/10wi-cs140/pintos/pintos_2.html#SEC15

Problem Statement:

Implement priority scheduling in Pintos. When a thread is added to the ready list that has a higher priority than the currently running thread, the current thread should immediately yield the processor to the new thread. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first. 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) to PRI_MAX (63). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. The initial thread priority is passed as an argument to thread_create(). If there's no reason to choose another priority, use PRI_DEFAULT (31). The PRI_ macros are defined in threads/thread.h, and you should not change their values.

Implementation Notes:

By default, PintOS stores the threads to be executed in list `ready_list`. In order to implement priority scheduling, you will need to either implement `ready_list` as a priority queue, or implement a *set* of queues, one for each priority level, such that each task can be sent to its respective queue.

Important functions to consider are (also check `thread.h` for the prototypes): `thread_init()`, `thread_unblock()`, and `next_thread_to_run()`.

As these functions all use the default `ready_list` (take a look at `list.h` in "`pintos/src/bin/kernel`" for the list prototypes available). You will need to modify (or re-implement) these functions to use your newly implemented queuing structure, instead. You may also need to implement new `push_queue` and `pop_queue` functions to interact with the new structures, as well.

Testing:

Update the "alarm-zero.c" file with the following code excerpt (**make sure to re-build your "`pintos/src/threads/build`" folder before running any tests!**). The test spawns a set of 64 threads, with matching respective priorities (Thread 0 has priority 0, Thread 1 has priority 1, and so on until Thread 64). If your code runs properly, these threads should run in the opposite order (starting with Thread 64 and ending with Thread 0), as their priorities are inverted, instead of running as issued if priority scheduling is not implemented.

To run it, invoke : **pintos run alarm-zero**

```
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/malloc.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

// creating threads with different priorities to test that the
// scheduler dispatches threads with
// higher priority first

void tester()
{
       int i=0;
       printf("\nfirst run %s\n",thread_name());
       timer_msleep(500);
       printf("\tfinished %s\n",thread_name());
}
void
test_alarm_zero (void)
{
       int i;
       for (i = 0; i < 64; i++)
       {
              char name[16];
              snprintf (name, sizeof name, "thread %d", i);
              thread_create (name, i, tester,NULL );
       }
  pass ();
}
```