NACHOS PROGRAMMING ASSIGNMENT 1

CS 370 – OPERATING SYSTEMS, LUMS

PRIORITY SCHEDULING

DUE DATE: WEDNESDAY OCT 5, 2016, 11:59PM

This programming assignment has been adapted from the Nachos project OS Course at Washington University and other universities

Before you start this programming assignment copy the file "threadtest.cc" from the directory /home/nachos into your threads directory by following the steps below:

1. **Go to the "threads" directory by typing the following command at the shell**
   cd nachos-3.4/code/threads/
2. **For copying "threadtest.cc" type:**
   cp ~nachos/threadtest.cc .
3. **Goto nachos-3.4/code/ directory and type "make clean"**
4. **Compile he code again by typing "make"**
5. **Goto the threads directory and run nachos by typing ./nachos –rs 200.**

Invoking nachos with "-rs #" makes calls to the scheduler for context switching at random time instants.

PART-1

1. Scheduler in Nachos

In the installed Nachos, Scheduler uses Round Robin scheduling. The scheduling is implemented as an object of class Scheduler. Its code can be found in threads/scheduler.h and scheduler.cc. The method of this class provides all the functions to schedule threads. When Nachos is started, an object of class Scheduler is created and referenced by a global variable scheduler. The class definition of Scheduler is as follows:

```
class Scheduler {
  public:
    Scheduler();                         // Initialize list of ready threads
    ~Scheduler();                        // De-allocate ready list
    void ReadyToRun(Thread* thread); // Thread can be dispatched.
    Thread* FindNextToRun();    // Remove the first thread on the readylist, if any, and return thread.
    void Run(Thread* nextThread, bool finishing);  // Cause nextThread to start running
    void CheckToBeDestroyed();   // Check if thread that had been running needs to be deleted
    void Print();                        // Print contents of ready list
                                           // SelfTest for scheduler is implemented in class Thread
  private:
    List<Thread *> *readyList;      // queue of threads that are ready to run but not running
    Thread *toBeDestroyed;          // finishing thread to be destroyed
                                           // by the next thread that runs
};
```

The private data member readyList is a pointer to a List object defined in lib/list.h and list.cc. This list is the ready queue to hold all the threads in the READY state.

Function ReadyToRun(Thread* thread) puts a thread at the end of ready queue by calling Append(thread).

```
void
Scheduler::ReadyToRun (Thread *thread)
{
   ASSERT(kernel->interrupt->getLevel() == IntOff);
   DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
   thread->setStatus(READY);
   readyList->Append(thread);
}
```

Function FindNextToRun () returns the next thread to be scheduled onto the CPU. If there are no ready threads, return NULL. The thread is removed from the ready list.

```
Thread *
Scheduler::FindNextToRun (){
   ASSERT(kernel->interrupt->getLevel() == IntOff);
   if (readyList->IsEmpty()) { return NULL;} else {
        return readyList->RemoveFront();
   }
}
```

Function Run dispatches the CPU to nextThread, saves the state of the old thread, and loads the state of the new thread, by calling the machine dependent context switch routine, SWITCH.

2. How to Implement Priority Scheduling in Nachos

In the function ReadyToRun(Thread* thread), each thread is put at the end of ready queue when it is ready to run. In the function FindNextToRun (), a new thread is chosen to run by removing it from the front of the ready queue. Such an order does not support priority scheduling.

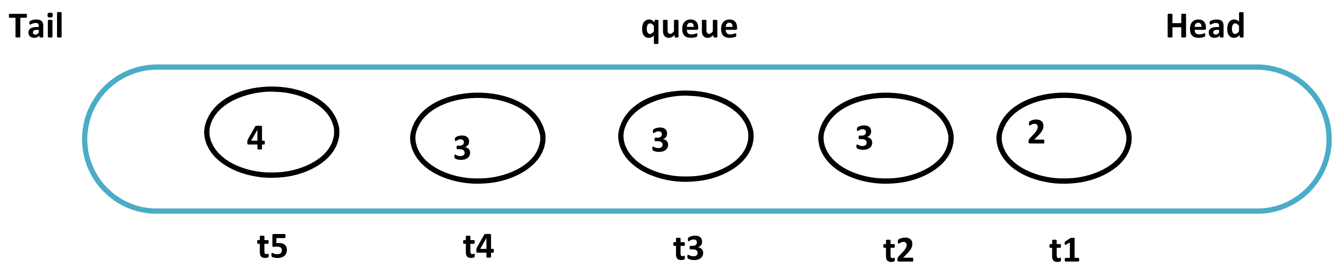
In order to perform Priority scheduling, the scheduler must select the next thread from the readyList that has the highest priority of all the threads in the queue. If two or more threads are of same priority, then Round Robin scheduling is followed.

Note: In the Unix and Linux Operating system, Priority Value of "1" means the HIGHEST Priority, followed by "2", and so on.

Priority scheduling can be implemented by including a private variable **priority** in Thread class and initialized in the Thread constructor. You can overload the Thread constructor to implement priority scheduling. Two functions are provided to set and get the value of priority Get_Priority() returns the newly assigned priority of the thread.

The scheduler picks up the thread from the front of the queue for execution. Different to the Round Robin scheduling, the thread should be inserted into the queue by "List<T>::SortedInsert(T item)" You will have to implement it by using the already implemented "List<T>" class, which makes sure that the threads are added to the ready queue based on their priority so that RemoveFront() always returns the thread with highest priority. The thread with the highest priority (e.g., value = 1) is placed at the front of the queue whereas the thread with the lowest priority (e.g., value = 5)is added at the end of the queue.

Here is an example to better understand the problem: suppose currently queue holds 5 threads as shown in the figure.



In this particular scenario context switch won't happen until the thread with priority 2 finishes its execution or another thread comes with the same or higher priority (i.e., priority value <= 2). Once the t1 has completed its execution t2 will be put to run but on the next interrupt you will have to context switch t2 with t3. If t2 has not yet finished its job put it back in the queue but in between t4 and t5. It implies scheduling among the threads with same priority will be in Round Robin fashion. Don't forget you have to insert new incoming threads in to the queue appropriately depending upon their priority.

3. What Modifications Should Be Made?

A) threads/threads.h:

A private variable priority should be declared as of type integer. The functions Set_Priority(int p) and Get_Priority() should be declared for setting and retrieving the priority of the current thread respectively.

```
class Thread {
private:
    int priority;                    // the priority of a thread
public:
/* implementation of priority scheduling */
  int Get_Priority();
  void Set_Priority(int p);
...
};
```

B) threads/threads.cc:

The priority of a newly created thread should be initialized in thread's constructor . The implementation of Get_Priority() is given below.

```
int Thread::Get_Priority() {
  return priority;
}
```

C) threads/scheduler.cc:

The round robin scheduler scheduling is modified in order to accommodate priority scheduling. In the function ReadyToRun(Thread* thread), the readyList->Append(thread) should be changed to List<T>::SortedInsert(T item) so that threads in the readyList is arranged in the increasing order of priority.

```
void

Scheduler::ReadyToRun (Thread *thread)

{

   ASSERT(kernel->interrupt->getLevel() == IntOff);

   DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());


   thread->priority = Get_Priority();

   thread->setStatus(READY);

   --- insert the thread so that threads in the readyList are sorted in decreasing order by "priority"

}
```

PART-2

In Part 1, you implemented basic priority scheduling in NACHOS. This basic priority scheduling may cause starvation which means, it is highly probable that threads with lower priority will stuck in the queue indefinitely and may never be executed.

To avoid this situation, you will implement AGEING which essentially means that every time a context switch occurs you need to increase the effective priority of all the threads by one (i.e., decrease the priority value by 1).

Be very careful! effective_priorty must never cross the maximum priority limit that is “1” in our case. You will have to declare another variable in Thread named, “effective_priorty”. At the time of thread creation this will be set to the original priority “priority” provided by user. Keep in mind once the thread has been created, the original priority will never change.

As a result of AGEING the effective priority of the threads that are waiting in the ready queue will keep on increasing and hence starvation wouldn’t happen.

Another important point is, when a thread is placed back in the ready queue its effective priority is set to the value of original priority. Be aware of the fact, as the effective priorities are changing dynamically so you have to make careful insertion and removals from the queue while performing context switching. The rest of scheduling algorithm will remain the same as discussed in part-1.

For setting the effective priority you will implement Set_ Effective _Priorty(int p) function.

```
void Thread::Set_Effective_Priority(int p) {

        if( p > 0 and p <= 5 )

                effective_priority  = p;

}
```

```
void Thread::Get_Effective_Priority(int p) {

                return effective_priority;

}
```

THE END