Operating Systems Laboratory (CS39002)

Spring Semester 2020-2021

Assignment 4: Implementation of multilevel feedback queue scheduling (MLFQS) in PintOS

Assignment given on: February 11, 2021

Assignment deadline: February 25, 2021, 1:00 PM

This assignment is an extension of previous assignment where you improved the thread implementation supported by PintOS. Specifically, in the previous assignment you improved the timer_sleep(), defined in devices/timer.c and avoided busy waiting. Now a kernel scheduler needs to ensure *fairness* and *performance*.

The performance requirement ensures that the periodic requirement of determining the resources used by each thread and changing the thread priorities to reflect the cpu usage is efficient. Fairness criteria ensures that threads with similar demand on the resources get similar access to the processor time. So, your lower priority threads should not miss the processor time completely.

To that end, while removing the busy wait some of you might have already created a separate wakeup thread (Of course you might have completed the assignment without the wakeup thread too). However, we are providing a sketch using which you can create the wakeup thread below. This assignment depends on mechanisms similar to the wakeup thread.

Part 1: Implementing a wakeup thread (30 marks)

> The sole purpose of the wakeup thread is to unblock the threads blocked on alarms. The thread becomes active when the current time (timer tick count) matches the wakeup time for the (next) earliest wakeup time of a sleeping thread. The thread will unblock all threads in the waiting-for-timer-alarm queue (let's call it sleepers) with the same wakeup time as the current time.
>
> Wakeup thread then uses list sleepers to determine the wakeup time for its next action. The thread can then block itself until the time so determined for the next wakeup phase. Function thread_tick() will unblock this wakeup thread at the right time.
>
> Since the wakeup thread is a managerial thread and not among the threads in list sleepers, its action code is simple and very easy to write. Interference or likely parallel access to the list of waiting threads is avoided by ensuring that the (wakeup) thread is non-preemptive and has high priority. One advantage of this is that we do not have to disable interrupts while the threads waiting for timer alarms are being unblocked. Once created the thread enters an infinite loop, where it is blocked to be woken up when some sleeping threads are to be unblocked from their timer wait. It will insert the released threads in ready_list and block itself again.
>
> In this arrangement, threads call timer_sleep() to begin waiting for the timer alarm. All these threads are inserted in sorted list sleepers. However, the wakeup managerial thread calls a separate function (timer_wakeup()) to unblock the waiting threads.

Your task is to implement the wakeup thread-based modification of previous assignment, where the wakeup just unblocks a sleeping thread when it's time to wake it up.

Part 2: Implement MLFQS (70 marks)

Problem:

The second part of this assignment is to add an MLFQS (multilevel feedback queue scheduler) to Pintos. The resulting operating system should be able to run either the basic priority scheduler or the MLFQ scheduler. The choice is made when Pintos boots. By default, it will run the basic priority scheduler. To make it use the MLFQ scheduler, you need to add the kernel option -mlfqs to the command that is used to run Pintos. For example:

```
pintos -v -- -mlfqs run mlfqs-load-1
```

The kernel has a global variable named thread_mlfqs that it sets to true if the -mlfqs option is present. Otherwise, thread_mlfqs is false. The code that implements MLFQS should only run if thread_mlfqs is true. Furthermore, when thread_mlfqs is true, the function thread_set_priority should have no effect, since under MLFQS, threads do not set their own priorities. However, it is still true under MLFQS that a higher priority thread should always run in preference to a lower priority thread.

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.

Before you start coding, go through this link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_2.html. it will provide you an understanding of the gist and functionalities of each relevant file.

Tasks:

You will implement an MLFQS that is similar to the one used in BSD Unix, version 4.4. The details for MLFQS are discussed here in the Pintos documentation: https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html

In this case the threads cannot set their own priority, but they can set own nice value. Below we are giving a summary of tasks from the webpage above, however you need to read the webpage to know the exact implementation details.

Task 1: Each thread has an integer nice value that determines how "nice" the thread should be to other threads. A nice of zero does not affect thread priority. A positive nice, to the maximum of 20, decreases the priority of a thread and causes it to give up some CPU time it would otherwise receive. On the other hand, a negative nice, to the minimum of -20, tends to take away CPU time from other threads.

The initial thread starts with a nice value of zero. Other threads start with a nice value inherited from their parent thread. You must implement the functions described below, which are for use by test programs. The skeleton definitions for them are in *threads/thread.c*.

Function: int thread_get_nice (void)

Returns the current thread's nice value.

Function: void thread_set_nice (int new_nice)

Sets the current thread's nice value to new_nice and recalculates the thread's priority based on the new value. If the running thread no longer has the highest priority, yields.

Task 2. Implement *thread_get_recent_cpu(),* for which there is a skeleton in *threads/thread.c*. We wish *recent_cpu* to measure how much CPU time each process has received "recently." Furthermore, as a refinement, more recent CPU time should be weighted more heavily than less recent CPU time. So, use the *exponentially weighted moving average* method to calculate it. The details are in section B.3 of the following link: https://web.stanford.edu/class/cs140/projects/pintos/pintos_7.html

Function: int thread_get_recent_cpu (void)
Returns 100 times the current thread's recent_cpu value, rounded to the nearest integer.

Task 3. Calculate priority of a thread using

priority = PRI_MAX - (recent_cpu / 4) – (nice * 2)

You should Implement your scheduler thread using this priority value. This type of scheduler maintains several queues of ready-to-run threads, where each queue holds threads with a different priority. At any given time, the scheduler chooses a thread from the highest-priority non-empty queue. If the highest-priority queue contains multiple threads, then they run in "round robin" order.

Your MLFQS scheduler should have 64 priorities and thus 64 ready queues, numbered 0 (PRI_MIN) through 63 (PRI_MAX). Lower numbers correspond to lower priorities, so that priority 0 is the lowest priority and priority 63 is the highest. Thread priority is calculated initially at thread initialization. It is also recalculated once every fourth clock tick, for every thread using the formula above.

This formula gives a thread that has received CPU time recently lower priority for being reassigned the CPU the next time the scheduler runs. This is key to preventing starvation: a thread that has not received any CPU time recently will have a *recent_cpu* of 0, which barring a high nice value should ensure that it receives CPU time soon.

Submission Guideline:

You need to upload a zip containing the files you changed along with a design document in Moodle. There should be one submission from each group. Name your zip file as "**Assgn4_<groupno>.zip**". The zip file should contain:

- The files that you changed.
- A design document. You can find the template for design document here: https://web.stanford.edu/class/cs140/projects/pintos/threads.tmpl. Fill the part related to "ADVANCED SCHEDULER" up according to your implementation and include it in your zip file.