# Advanced Scheduler

## 1. Introduction

Implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler to reduce the average response time for running jobs on your system.

Like the priority scheduler, the advanced scheduler chooses the thread to run based on priorities. However, the advanced scheduler does not do priority donation. Thus, we recommend that you have the priority scheduler working, except possibly for priority donation, before you start work on the advanced scheduler.

You must write your code to allow us to choose a scheduling algorithm policy at Pintos startup time. By default, the priority scheduler must be active, but we must be able to choose the 4.4BSD scheduler with the ‘-mlfqs’ kernel option. Passing this option sets thread*mlfqs, declared in ‘threads/thread.h’, to true when the options are parsed by parse*options(), which happens early in main().

When the 4.4BSD scheduler is enabled, threads no longer directly control their own priorities. The priority argument to thread*create() should be ignored, as well as any calls to thread*set*priority(), and thread*get_priority() should return the thread’s current priority as set by the scheduler.

## 2. BSD4.4 Scheduler

The goal of a general-purpose scheduler is to balance threads’ different scheduling needs. Threads that perform a lot of I/O require a fast response time to keep input and output devices busy, but need little CPU time.

On the other hand, compute-bound threads need to receive a lot of CPU time to finish their work, but have no requirement for fast response time. Other threads lie somewhere in between, with periods of I/O punctuated by periods of computation, and thus have requirements that vary over time.

A well-designed scheduler can often accommodate threads with all these requirements simultaneously.

You must implement the scheduler described in this appendix. Our scheduler resembles the one described in [McKusick], which is one example of a multilevel feedback queue scheduler. 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.

Multiple facets of the scheduler require data to be updated after a certain number of timer ticks. In every case, these updates should occur before any ordinary kernel thread has a chance to run, so that there is no chance that a kernel thread could see a newly increased timer_ticks() value but old scheduler data values.

The 4.4BSD scheduler does not include priority donation.

### 2.1 Niceness

Thread priority is dynamically determined by the scheduler using a formula given below. However, each thread also 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. We have provided skeleton definitions for them in ‘threads/thread.c’.

**int thread*get*nice (void)** [Function]

Returns the current thread’s nice value.

**void thread*set*nice (int new_nice)** [Function]

Sets the current thread’s nice value to new nice and recalculates the thread’s priority based on the new value (see Section B.2 [Calculating Priority], page 91). If the running thread no longer has the highest priority, yields.

## 2.2 Calculating Priority

Our scheduler has 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. In either case, it is determined by the formula:

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

where recent cpu is an estimate of the CPU time the thread has used recently (see below) and nice is the thread’s nice value. The result should be rounded down to the nearest integer (truncated). The coefficients 1/4 and 2 on recent cpu and nice, respectively, have been found to work well in practice but lack deeper meaning. The calculated priority is always adjusted to lie in the valid range **PRI_MIN** to **PRI_MAX**.

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.

## 2.3 Calculating recent cpu

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. One approach would use an array of n elements to track the CPU time received in each of the last n seconds. However, this approach requires O(n) space per thread and O(n) time per calculation of a new weighted average.

Instead, we use a exponentially weighted moving average, which takes this general form: `java x(0) = f(0), x(t) = ax(t − 1) + (1 − a)f(t), a = k/(k + 1),`

where x(t) is the moving average at integer time t ≥ 0, f(t) is the function being averaged, and k > 0 controls the rate of decay. We can iterate the formula over a few steps as follows:

```
        x(1) = f(1),
        x(2) = af(1) + f(2),
        .
        .
        .
        x(5) = a^4f(1) + a^3f(2) + a^2f(3) + af(4) + f(5).
```

The value of f(t) has a weight of 1 at time t,a weight of a at time t+1, a2 at time t+2,and so on. We can also relate x(t) to k: f(t) has a weight of approximately 1/e at time t + k, approximately 1/e2 at time t + 2k, and so on. From the opposite direction, f (t) decays to weight w at time t + loga (w).

The initial value of recent cpu is 0 in the first thread created, or the parent's value in other new threads. Each time a timer interrupt occurs, recent cpu is incremented by 1 for the running thread only, unless the idle thread is running. In addition, once per second the value of recent cpu is recalculated for every thread (whether running, ready, or blocked), using this formula:

```
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice,
```

where load_avg is a moving average of the number of threads ready to run (see below). If load avg is 1, indicating that a single thread, on average, is competing for the CPU, then the current value of recent cpu decays to a weight of .1 in log2/3 (.1) ≈ 6 seconds; if load avg is 2, then decay to a weight of .1 takes log3/4 (.1) ≈ 8 seconds. The effect is that recent cpu estimates the amount of CPU time the thread has received "recently," with the rate of decay inversely proportional to the number of threads competing for the CPU.

Assumptions made by some of the tests require that these recalculations of recent cpu be made exactly when the system tick counter reaches a multiple of a second, that is, when **timer*ticks () % TIMER*FREQ == 0**, and not at any other time.

The value of recent cpu can be negative for a thread with a negative nice value. Do not clamp negative recent cpu to 0.

You may need to think about the order of calculations in this formula. We recommend computing the coefficient of recent cpu first, then multiplying. Some students have reported that multiplying load avg by recent cpu directly can cause overflow.

You must implement **thread*get*recent_cpu()**, for which there is a skeleton in 'threads/thread.c'.

**int thread*get*recent_cpu (void)** [Function]

Returns 100 times the current thread's recent cpu value, rounded to the nearest integer.

## 2.4 Calculating load_avg

Finally, load avg, often known as the system load average, estimates the average number of threads ready to run over the past minute. Like recent cpu, it is an exponentially weighted moving average. Unlike priority and recent cpu, load avg is system-wide, not thread specific. At system boot, it is initialized to 0. Once per second thereafter, it is updated according to the following formula:

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads,
```

where ready threads is the number of threads that are either running or ready to run at time of update (not including the idle thread).

Because of assumptions made by some of the tests, load avg must be updated exactly when the system tick counter reaches a multiple of a second, that is, when timer*ticks () % TIMER*FREQ == 0, and not at any other time.

You must implement **thread*get*load_avg()**, for which there is a skeleton in 'threads/thread.c'.

**int thread*get*load_avg (void)** [Function]

Returns 100 times the current system load average, rounded to the nearest integer.

## 2.5 Summary[Important]

The following formulas summarize the calculations required to implement the scheduler. They are not a complete description of scheduler requirements.

Every thread has a nice value between -20 and 20 directly under its control. Each thread also has a priority, between 0 (PRI*MIN) through 63 (PRI*MAX), which is recalculated using the following formula every fourth tick:

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

recent cpu measures the amount of CPU time a thread has received "recently." On each timer tick, the running thread's recent cpu is incremented by 1. Once per second, every thread's recent cpu is updated this way:

```
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice.
```

load_avg estimates the average number of threads ready to run over the past minute. It is initialized to 0 at boot and recalculated once per second as follows:

```
load_avg = (59/60)*load_avg + (1/60)*ready_threads.
```

where ready threads is the number of threads that are either running or ready to run at time of update (not including the idle thread).

# 3. Fixed-Point Real Arithmetic

**In the formulas above, priority, nice, and ready threads are integers, but recent cpu and load avg are real numbers. ** Unfortunately, Pintos does not support floating-point arithmetic in the kernel, because it would complicate and slow the kernel. Real kernels often have the same limitation, for the same reason. This means that calculations on real quantities must be simulated using integers. This is not difficult, but many students do not know how to do it. This section explains the basics.

The fundamental idea is to treat the rightmost bits of an integer as representing a fraction. For example, we can designate the lowest 14 bits of a signed 32-bit integer as fractional bits, so that an integer x represents the real number x/2^14. This is called a 17.14 fixed-point number representation, because there are 17 bits before the decimal point, 14 bits after it, and one sign bit.1 A number in 17.14 format represents, at maximum, a value of (2^31 – 1)/2^14 ≈ 131,071.999.

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, x and y are fixed-point numbers, n is an integer, fixed-point numbers are in signed p.q format where p + q = 31, and f is 1 << q:(in this case, p = 17, q = 14)

| Operation | Result |
|---|---|
| Convert n to fixed point | n*f |
| Convert x to integer (rounding toward zero) | x/f |
| Convert x to integer (rounding to nearest) | (x + f / 2) / f if x >= 0,(x - f / 2) / f if x <= 0. |
| Add x and y | x+y |
| Subtract y from x | x-y |
| Add x and n | x+n *f |
| Subtract n from x | x-n *f |
| Multiply x by y | ((int64_t) x) * y / f |
| Multiply x by n | x*n |
| Divide x by y | ((int64_t) x) * f / y |
| Divide x by n | x/n |

# 4. Assignment

## 4.1 Descprition

You need to support Fixed-Point Real Arithmetic in Pintos at first, and use these Fixed-Point Real Arithmetic function to implement a multilevel feedback queue scheduler similar to the 4.4BSD scheduler.

## 4.2 Requirement

You should pass all the tests in **src/threads/** after you have finished this assignment.

Run `make check` in ***src/threads/*** to make sure your design can pass all these tests.

You should download and **fill** the [DESIGNDOC]() and archive your code(**the whole pintos folder**) and DESIGNDOC into a **zip** file **named sid*nameInPinYin*vid.zip(e.g 12330441*zhangsan*v0.zip)** and upload to the ftp.