Project 1. Threads

Score

Total: 30 points

Requirements

1. Alarm Clock (5 points)

- Reimplement timer_sleep(), defined in devices/timer.c.

A. Current Implementation: **"busy waits,"** that is, it spins in a loop checking the current time and calling thread_yield() until enough time has gone by.

B. New Implementation

i. **void timer_sleep (int64_t x) (5 points)**: Suspends execution of the calling thread for x timer ticks. Unless the system is otherwise idle, the thread need not wake up after exactly x ticks. Just put it on the ready queue after they have waited for the right amount of time.

C. Note

i. When a thread calls timer_sleep(n), the thread will be made ready by OS at some time after n ticks. Therefore, the thread may wait for longer than n ticks.

ii. Timer Interrupt: Pintos generates timer interrupt per ticks(by Intel 8254 programmable interval timer. It is no need to study the applicable chip). By timer interrupt handler, thread_tick() is executed, and thread scheduling is possible in here.

2. Priority Scheduling (15 points)

- Implement priority scheduling in Pintos.

A. Current Implementation: New thread is inserted at the last of ready_list.

B. New Implementation

i. **Priority scheduling (5 points)**: When a thread is added to the ready list that has a higher priority than the currently running thread, current thread should go to ready queue, and the highest priority thread should be executed immediately. Similarly, when threads are waiting for a lock, semaphore, or condition variable, the highest priority waiting thread should be awakened first.

ii. **Priority donation (5 points)**: Implement priority donation. In case of priority inversion, L thread should temporarily get H priority to prevent 'priority inversion'. It should be implemented in case of lock, but it is unnecessary in case of semaphore and condition variable.

1. **Priority inversion**: Consider high, medium, and low priority threads H, M, and L, respectively. If H needs to wait for L (for instance, for a lock held by L), and M is on the ready list, then H will never get the CPU because the low priority thread

will not get any CPU time because M should complete its work. H should wait until L's release of lock. This is priority inversion.

2. **Extended donation**: You must also handle various donation cases. If H is waiting on a lock that M holds and M is waiting on a lock that L holds, then both M and L should be boosted to H's priority. You need to implement an algorithm that satisfy various priority donation cases such as priority donate nest.(refer to tests/threads)

iii. **Examination and modification of priority (5 points)**: 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.

1. **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.
2. **int thread_get_priority (void)**: Returns the current thread's priority. In the presence of priority donation, returns the higher (donated) priority.

3. Advanced Scheduler (10 points)

- Implement a multilevel feedback queue scheduler

A. MLFQS (5 points): Implement MLFQS. It needs ready queues per priority. By using these queues, it manages the priority automatically. From 0 to 63, the total number of ready queue is 64. You should refresh the thread's priority automatically per 4 ticks.

i. For priority calculation, refer to appendix.

ii. 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 MLFQS scheduler with the **-mlfqs** kernel option.(thread_mlfqs, declared in threads/thread.h, execution example : pintos –b -- -mlfqs ...)

B. Priority scheduling in MLFQS (5 points): 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 should calculate the thread's priority per 4 ticks. Through this calculation, the change of thread execution sequence can happen.

i. **Thread_set_priority()** should not modify priority, but thread_set_nice() can modify the priority indirectly.

ii. No Priority donation.

iii. In case of the highest priority, use round robin scheduling method.

APPENDIX

B.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.

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 (see section B.2 Calculating Priority). If the running thread no longer has the highest priority, yields.

B.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 toPRI_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.

B.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:

$$x(0) = f(0),$$
$$x(t) = a*x(t-1) + f(t),$$
$$a = k/(k+1),$$

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:

$$x(1) = f(1),$$
$$x(2) = a*f(1) + f(2),$$
$$\ldots$$
$$x(5) = a**4*f(1) + a**3*f(2) + a**2*f(3) + a*f(4) + f(5).$$

The value of f(t) has a weight of 1 at time t, a weight of a at time t+1, a**2 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/e**2 at time t+2*k, and so on. From the opposite direction, f(t) decays to weight w at time t + ln(w)/ln(a).

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 ln(.1)/ln(2/3) = approx. 6 seconds; if *load_avg* is 2, then decay to a weight of .1 takes ln(.1)/ln(3/4) = approx. 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.

Function: int thread_get_recent_cpu (void)

Returns 100 times the current thread's *recent_cpu* value, rounded to the nearest integer.

B.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`.

Function: int thread_get_load_avg (void)

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

B.5 Summary

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).

B.6 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), where ** represents exponentiation. 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.(7) A number in 17.14 format represents, at maximum, a value of (2**31 - 1)/(2**14) = approx. 131,071.999.

Suppose that we are using a p.q fixed-point format, and let f = 2**q. By the definition above, we can convert an integer or real number into p.q format by multiplying with f. For example, in 17.14 format the fraction 59/60 used in the calculation of *load_avg*, above, is 59/60*(2**14) = 16,110. To convert a fixed-point value back to an integer, divide by f. (The normal / operator in C rounds toward zero, that is, it rounds positive numbers down and negative numbers up. To round to nearest, add f / 2 to a positive number, or subtract it from a negative number, before dividing.)

Many operations on fixed-point numbers are straightforward. Let x and y be fixed-point numbers, and letn be an integer. Then the sum of x and y is x + y and their difference is x - y. The sum of x and n is x + n * f; difference, x - n * f; product, x * n; quotient, x / n.

Multiplying two fixed-point values has two complications. First, the decimal point of the result is q bits too far to the left. Consider that (59/60)*(59/60) should be slightly less than 1, but 16,111*16,111 =

259,564,321 is much greater than 2**14 = 16,384. Shifting q bits right, we get 259,564,321/(2**14) = 15,842, or about 0.97, the correct answer. Second, the multiplication can overflow even though the answer is representable. For example, 64 in 17.14 format is 64*(2**14) = 1,048,576 and its square 64**2 = 4,096 is well within the 17.14 range, but 1,048,576**2 = 2**40, greater than the maximum signed 32-bit integer value 2**31 - 1. An easy solution is to do the multiplication as a 64-bit operation. The product of x and y is then `((int64_t) x) * y / f`.

Dividing two fixed-point values has opposite issues. The decimal point will be too far to the right, which we fix by shifting the dividend q bits to the left before the division. The left shift discards the top q bits of the dividend, which we can again fix by doing the division in 64 bits. Thus, the quotient when `x` is divided by `y` is `((int64_t) x) * f / y`.

This section has consistently used multiplication or division by f, instead of q-bit shifts, for two reasons. First, multiplication and division do not have the surprising operator precedence of the C shift operators. Second, multiplication and division are well-defined on negative operands, but the C shift operators are not. Take care with these issues in your implementation.

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`:

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`