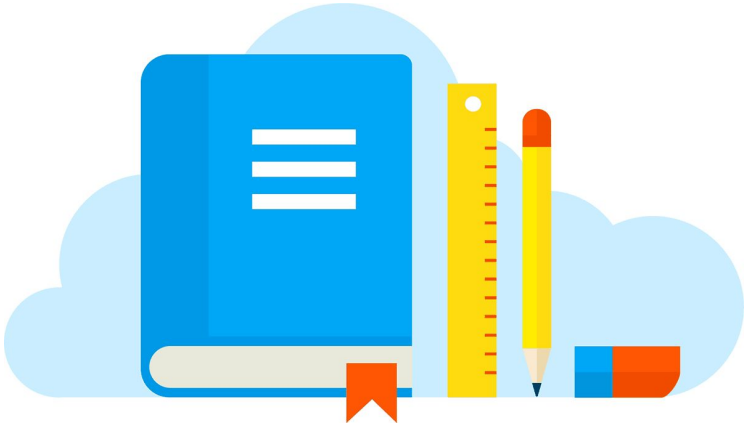


Pintos Phase 1

Team

Sherif Hamdy: 32

Mohamed Ahmed Ramadan: 51

Tarek Mohamed Nawara: 36

Alarm Clock

This part of the project was designed to block the threads for a specific period of time and then wake them after it. We were given an implementation of this part using busy waiting technique and we were asked to change this implementation to use more efficient way for the thread sleep.

Data Structure

We used the given list implementation in an efficient way so first we insert the thread to block in this list in order according to the awake time we should awake it at. We introduced this field to the thread structure. Then using the ***thread_tick*** method that we know it is been called each clock tick to iterate over this list of blocked threads and awake the threads which there awake time has come.

Making the implementation more efficient

By inserting the threads in order of the awake time. When iterating over them in the ***thread_tick*** we will stop the iteration when we reach a thread with awake time greater than the current time.

Algorithm

In the ***thread_sleep*** method we assign to the current thread a value we called the ***awake_time*** , this value indicates when the thread to be awaked, then we insert the thread in a list called ***blocked_list*** this list holds all the blocked threads sorted in an ascending order according to the ***awake_time*** value. In the ***thread_tick*** method we iterate over this list again and unblock the threads that should be awaked, then we call the schedule method to choose the appropriate thread to be running.

Synchronization

In the algorithm race condition could happen in the ***thread_sleep*** method so we avoided it entirely by disabling interrupts before we block the running thread.

Priority donation

Data structures

We introduced new fields to the following

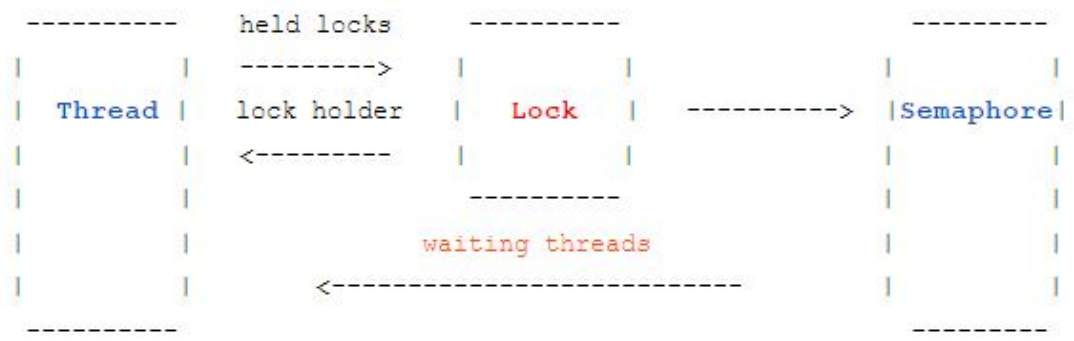
Lock structures

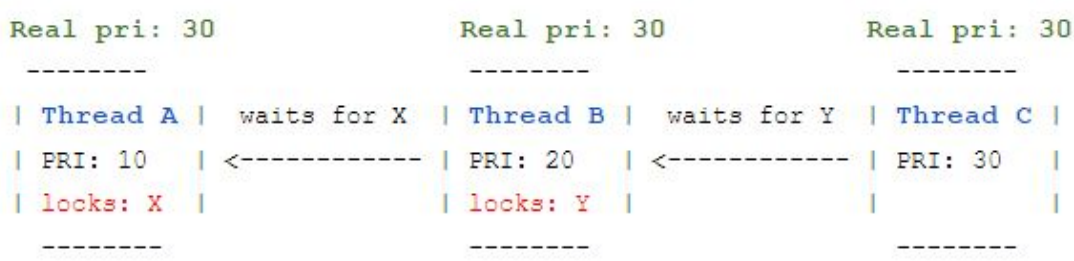
- A new variable called ***max_offered_donation*** this variable indicates the maximum donation offered by all the threads that tried to acquire this lock but failed because there was a thread acquiring the lock
- A new pointer called ***lock_elem*** this variable is responsible for inserting the lock in a list for later use.

Thread structure

- A new pointer called ***recently_tried_lock*** this pointer points to the last lock you tried to acquire but you failed

- A new list called ***acquired_locks*** which holds all the locks the current thread has successfully acquired.
- *Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a *.png file.)*





Algorithm

- *How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?*

  Ans: In the ***sema_up*** function we choose from all the waiting threads the one with the highest priority to enter the critical section.

- *Describe the sequence of events when a call to **lock_acquire** causes a priority donation. How is nested donation handled?*

  Ans: mark the ones that accessed the lock and couldn't acquire it, will mark the accessed lock with that lock and update the max donate priority of the lock and before the thread sleep we update the lock holder.

- *Describe the sequence of events when **lock_release** is called on a lock that a higher-priority thread is waiting for.*

  Ans: The lock holder will update its priority (max between max donation of all locks it hold and the original priority) then sema up in it, it will wake up the maximum one.

Synchronization

- *Describe a potential race in **thread_set_priority** and explain how your implementation avoids it. Can you use a lock to avoid this race?*

  Ans: Update the original priority and update the current priority:

  - if there's no donations .
  - if the new priority higher than the current priority.

  And if the new less than the old one we will yield the running thread.

Advanced scheduler

The main idea is to use a variation of the ***round robin*** algorithm but with the addition of multiple queues each queue indicates the level of priority and the duration that these threads should consume the CPU, but ***BSD*** simplifies the implementation process by making use of only one queue and introducing new fields to the thread structure and applying simple equations.

BSD Scheduler used with ***mlfqs*** kernel option which as we showed use simple mathematical equations to simulate the multi-level system we will show how its components are linked.

Stanford explanation for the BSD 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. For project 1, 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 nonempty 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.

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.

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 * 4)*

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.

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 an exponentially weighted moving average, which takes this general form:

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

x(2) = a*f(1) + f(2),

...

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.

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.

Data structures

int nice was added to thread.h so that we can indicate the thread niceness. Int ***recent_cpu*** in thread.c to indicate the time spent by the thread on the CPU. Int ***load_avg*** this element was added at thread.c. alongside with the elem in the struct thread previously mentioned its job.

Algorithms

We will specify the parts edited in order to collect the parts. Thread create the thread initially created blocked so we will unblock it then test if the current thread will continue to run or not we will disable the interrupts in this part in order not to many threads access the list in the same time. Thread unblock will do the insertion of the blocked thread into the list of the ready threads and maintain the order. Thread yield we will just maintain the thread ready list sorted to maintain the mechanism of the extraction of the

first thread as the running thread. Thread set priority we will edit this part in order to maintain the previous priority as the time slice will pass and the old priority will return. Will current thread continue or yield is a function which we used to determine if the running thread still has the max priority if not another thread will be scheduled to run. Thread update will do the updating of the recent CPU and the load average and the priorities alongside with calling the update priority each time slice for the running thread. So after collecting the parts the following will be done as our algorithm to solve the problem each second calculate the recent CPU of all the threads and the load average each time slice calculate the priority of the running thread and as the yield on return is called at thread tick we don't need to schedule another thread it will be done automatically each time to run the next thread that should run.

Rationale

Briefly critique your design, pointing out advantages and disadvantages in your design choices. If you were to have extra time to work on this part of the project,

- *how might you choose to refine or improve your design?*

  we couldn't used complex data structures in the project due to the date of the delivery as binary trees to maintain the sorting.

Suggestions: detecting overflows in the fixed point operations and using the locks instead of turning off the interrupts. The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not? We used the same concept as the explanation in the website of stanford as it was the most simple way as I think to implement fixed point operations.

SURVEY QUESTIONS

- In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard?

To be honest, we found task two to be very challenging.

- *Did it take too long or too little time?*

  No, it was just perfect for the given time.

- *Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS Design?*

  Yes, the priority donation part took us a lot to configure what do we need to do to make it work correct.