+--------------------+

| CS 140 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

>> Fill in the names and email addresses of your group members.

Shivaprakash Hiremath hiremath@buffalo.edu

Sachin Venugopal sachinve@buffalo.edu

Amrata Anant Raykar amrataan@buffalo.edu

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the

>> TAs, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while >> preparing

your submission, other than the Pintos documentation, course

>> text, lecture notes, and course staff.

ALARM CLOCK

==========

---- DATA STRUCTURES ----

>> A1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread {
        .
        .
        //Added
        int64_t wake_up_ticks; /* Stores time after which the thread needs to be woken up - ticks
        + timer_ticks()*/
        .
        .
}
```

```
static struct list wait_list; /*Stores a list of all sleeping threads that need to be woken up*/
```

```
static struct semaphore wait_sema; /*Used to handle synchronization during timer_sleep()*/
```

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),

>> including the effects of the timer interrupt handler.

When the timer_sleep() function is called with the number of clock ticks for which the thread sleeps, the value of ticks is first checked to determine if it is lesser than or equal to zero, in which case the function returns.

Otherwise, the parameter 'wake_up_ticks' of the thread is set to the time at which the thread needs to be woken up. The thread is then pushed into an ordered list which keeps track of all the sleeping threads. A semaphore is initiated and downed for the above operation to ensure synchronization. The semaphore is then up()-ed and thread_block() is called which in turn schedules another thread to process.

The timer interrupt handler checks at each timer tick if any thread needs to be woken up. If threads need to be woken up, they are popped from the wait_list and the threads are unblocked.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

Sleeping threads are pushed to the wait_list in the increasing order of wake_up_ticks. This minimizes the amount of time spent in searching for the threads to be woken up in the interrupt handler.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer_sleep() simultaneously?

Race conditions are avoided by the use of semaphores during the addition of threads to the wait_list.

Whenever multiple threads call thread_sleep(), they are put into sema->waiters and those threads are blocked until sema_up() is called by the initial thread which called sema_down().

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer_sleep ()?

As explained in the above question, semaphore will eventually block the other threads from execution until the first thread, which acquired the lock (sema_down ()), finishes.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to

>> another design you considered?

This design takes lesser time when the timer_interrupt handler is called, than our initial design, which used an unordered list to keep track of the sleeping threads. In our current design, we do not check our list further once we encounter a thread with higher wake up ticks than the current ticks, whereas in the other design we checked the complete list to find threads to awaken.

PRIORITY SCHEDULING

==========

---- DATA STRUCTURES ----

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread {
                .
                .
                //Added
                int init_priority; /* Keeps track of the initial priority of the thread*/
                struct list lock_list; /* list of locks acquired by thread*/
                struct lock *lock_to_acquire /* lock information that the thread tries to acquire */
                .
                .
        }
struct lock {
                .
```

```
        .
        //Added
        int max_priority; /*holds the priority of the maximum priority thread currently waiting for
        the lock*/
        struct list elem; /* Used to add lock as a member of lock_list*/
        .
        .
}
```

>> B2: Explain the data structure used to track priority donation.

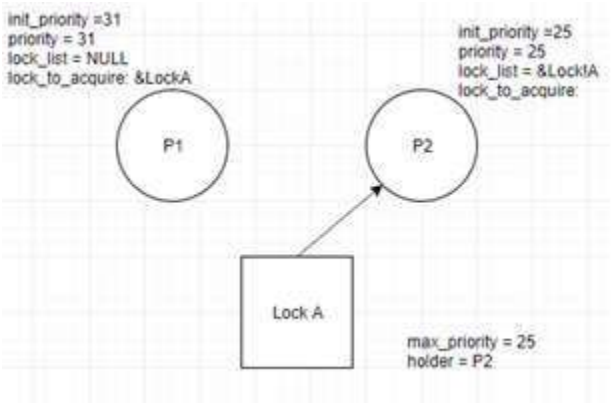
>> Use ASCII art to diagram a nested donation. (Alternately, submit a

>> .png file.)

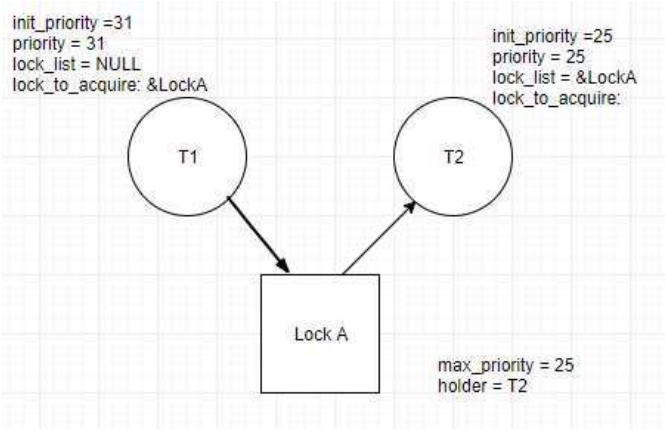
We use a lock pointer, lock_to_acquire, which points to the lock each thread in a nested lock dependency is trying to acquire.

Consider a thread H of high priority trying to acquire lock L1, which is currently held by thread M of lower priority. Thread M in turn is waiting to acquire lock L2 which is currently held by a lower priority thread L. Starting from thread H, we can use the lock pointer lock_to_acquire to traverse the list of dependencies till we reach a thread whose lock_to_pointer value is NULL. With each level of priority donation, we can use the pointer as a way to step into nested level to update priorities of lower priority threads.

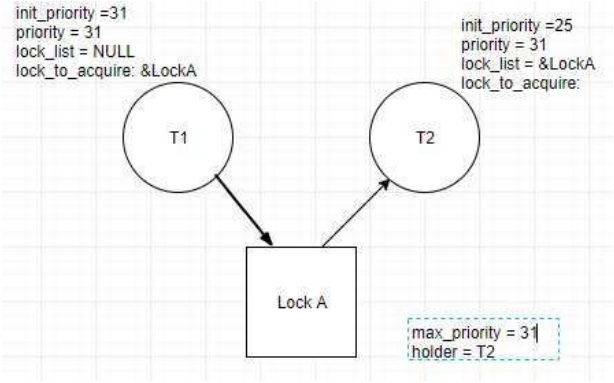
1. Consider lock A currently held by thread T2 of priority 25.



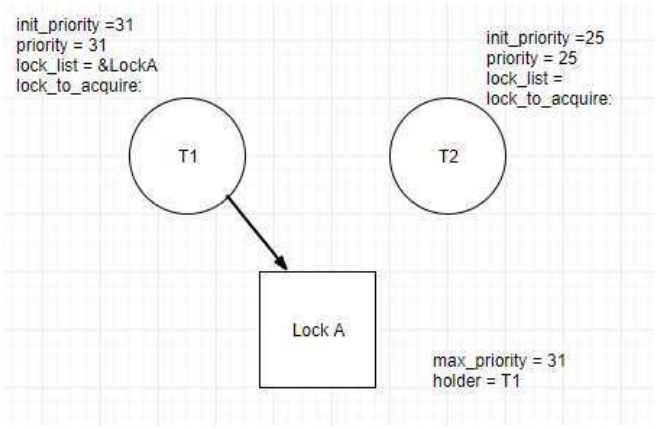
2. Thread T1 calls acquire_lock(&Lock A)



3. Priority donation: The value of max_priority is set to 31 and T2's priority is set to 31.



4. T2 completes its execution, changes its priority back to init_priority and frees LockA. T1 now acquires lockA and set's its holder to T1.



---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

The waiters list, used to keep track of the threads waiting to acquire a semaphore, lock or condition variable, respectively, is sorted according to the descending order of their priorities. When the semaphore, lock or condition variable is freed, the first thread in the list (thread with highest priority) is unblocked. This ensures that the highest priority thread is awakened first.

Updated: In sema_up(), we sort the list before the element is picked because priorities can be changed during priority donation (to handle the test case priority_donate_sema).

>> B4: Describe the sequence of events when a call to lock_acquire()

>> causes a priority donation. How is nested donation handled?

Consider the following sequence:

1. Thread H with higher priority calls lock_acquire()
2. If lock does not have a holder
    i. the current thread becomes the holder.
    ii. max_priority of the lock is set to current thread's priority
3. Else, if lock has a holder L, with lower priority than thread H, then,

i. DONATION: Priority of holder(L) is set to current thread's (H) priority.

ii. The value of max_priority is set to the priority of thread H.

iii. Thread H is blocked.

/*Holder completes execution and releases the lock. The higher priority thread H is unblocked. */

4. The thread H becomes the holder of the lock.

5. Thread->lock_to_acquire is set to NULL for H.

Consider thread L(lowest priority) holds lockA, which is needed by thread M(medium priority) which currently holds lockB, lockB in turn is needed by thread H(highest priority). Using thread->lock_to_acquire ->holder thread H identifies M as the holder of lockB and donates its priority to it. Thread M similarly uses thread->lock_to_acquire->holder to identify thread L and donates its new priority to L. The three threads thus have the same priority as thread H and lockA is first released to thread M, which then releases lockB to thread H.

>> B5: Describe the sequence of events when lock_release() is called

>> on a lock that a higher-priority thread is waiting for.

1. The holder removes the lock from its lock_list.
2. If the lock_list is empty, then the thread's priority is set to the value of its init_priority.
3. Else, the priority of the thread is set to the maximum value of max_priority of all the locks in the lock_list.
4. The lock's holder is set to NULL.

---- SYNCHRONIZATION ----

>> B6: Describe a potential race in thread_set_priority() and explain

>> how your implementation avoids it. Can you use a lock to avoid

>> this race?

Updated: Setting the priority is a critical section problem since there are shared variables being accessed and we have used a semaphore to ensure mutual exclusion.

---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to

>> another design you considered?

Our design uses a pointer to the lock in the thread struct to keep track of lock the thread wishes to acquire and stores acquired locks in a list. This approach is an improvement over our initial consideration of using a pointer to the thread holding the lock the current thread is trying acquire since in the case of multiple threads trying to acquire the same lock, the owner of the lock may change multiple times before a lower priority thread can acquire it. This would mean changing the pointer value multiple times which is infeasible.

ADVANCED SCHEDULER

==========

---- DATA STRUCTURES ----

>> C1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

```
struct thread {
        .
        .
        //Added
        int recent_cpu /* An exponentially weighted moving average to calculate recent CPU t
        time*/
        .
        int nice /* Nice value of the thread */
```

}

//in thread.c

int load_avg; /* exponentially weighted moving average of the number of ready threads to run*/

---- ALGORITHMS ----

>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each
>> has a recent_cpu value of 0. Fill in the table below showing the >>
scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:

timer	recent_cpu			priority			thread ticks
A	B	C	A	B	C		to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	58	58	58	A
36	20	12	4	58	58	58	C

>> C3: Did any ambiguities in the scheduler specification make values

>> in the table uncertain? If so, what rule did you use to resolve

>> them? Does this match the behavior of your scheduler?

If the running thread and a thread in the ready list have the same priority after a time slice is elapsed, the scheduler specification does not specify whether the running thread has to be preempted. Similarly, the scheduler specification does not mention which thread must be chosen for execution next, when there are multiple threads of the same priority.

When two or more threads of the same priority exist in the queue, we choose the threads to execute using the order in which they arrived into the ready queue.

Updated:

This matches the behavior of our scheduler.

>> C4: How is the way you divided the cost of scheduling between code

>> inside and outside interrupt context likely to affect performance?

Process scheduling involves two major operations. The first is to calculate the priorities of the threads at each second, incrementing the recent_cpu value at each timer tick for the running thread and calculating the new value every TIMER_FREQ ticks for all the threads. The second is to schedule the thread with the highest priority and order the other threads according to their priority.

While placing the code inside the interrupt context ensures the operations are performed without interruption, it uses a significant portion of the operating time allocated for the scheduled thread. Our design, therefore, executes only the most essential parts of the scheduling operations within the timer interrupt handler to ensure optimal performance.

---- RATIONALE ----

>> C5: 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?

Updated:

We have implemented our ready queue as a preemptive priority scheduled queue where the threads automatically have their priorities set and positions changed in the queue based on time spent in waiting for the CPU and number of ready threads.

Our design implements the scheduler by making very few changes to the existing thread and ready queue structure. We reduce the amount of computation time required to calculate thread priorities by using macros instead of defining functions to perform them.

One disadvantage of our design is that the ready queue is implemented as a single preemptive priority-scheduled queue.

If we had more time on the project, we would have liked to implement the ready queue as a true multilevel feedback queue, with two or more queue levels, each using a different scheduling algorithm. This would have allowed us to separate CPU-bound and I/O-bound threads to allocate CPU time optimally.

>> C6: 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?

Since Pintos does not support float point arithmetic, we have used fixed-point arithmetic to calculate recent CPU time and load average needed to compute thread priority. Our design implements the fixed-point operations using macros. Since macros are expanded when the code is preprocessed, rather than compiled, it is faster than performing the operations using functions called at runtime.

SURVEY QUESTIONS

==========

Answering these questions is optional, but it will help us improve the

course in future quarters. Feel free to tell us anything you want--these

questions are just to spur your thoughts. You may also choose to

respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in
>> future quarters to help them solve the problems? Conversely, did you
>> find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?

>> Any other comments?