CSE 521
PROJECT 1: THREADS
DESIGN DOCUMENT

---- GROUP ----

Deepika Ghodki <dghodki@buffalo.edu>
Kartikeyan Lakshminarayanan Vignesh <klakshmi@buffalo.edu>
Palek Naithani <paleknai@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.

In `thread` struct:

```
int64_t wake_time;
```
>> Time when the thread should wake up since start i.e. OS boot

```
struct semaphore sleep_sema;
```
>> Semaphore to keep track of sleeping

```
struct list_elem sleep_elem;
```
>> List element for sleeping thread list */

In `timer.c`:

```
static struct list sleeping_threads;
```
>> List of processes that are currently sleeping i.e. in BLOCKED state

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.
When timer_sleep is called on the running thread, we initialise the semaphore (mutex) of thread with 0 and set the absolute wake time (from the OS boot time) of the thread. Then we insert the thread into the **sleeping_threads** list in the non decreasing order of its **wake_time**. During this step, the interrupts are disabled to ensure mutual exclusion. Hence we are not interrupted by the timer interrupt while updating the list.

>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?
As threads are sorted in the increasing order of their wake times, the first thread in the list would be the one that wakes up first. At each tick, we compare the first wake_time with current and if not equal, we know that there is no other thread that will wake before this. Hence we save time by not iterating over the entire list of sleeping threads.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
When timer_sleep is called, the threads are added to the sleeping_threads list. As this list is shared between timer_sleep and thread_wake, there could be a race condition. To handle this, the interrupts are disabled when we add a thread to the sleeping_threads list. This prevents any other thread from being able to access the list, and hence prevents them from going to sleep simultaneously.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
The sleeping_threads list is accessed by timer_sleep and thread_wake (called by timer interrupt). As interrupts are disabled while adding to list, a timer interrupt will not occur until a thread has been added to the list. Further, interrupts are disabled when it is being put to sleep (i.e. moved to BLOCKED state).

---- RATIONALE ----

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

- Keep track of all threads that are sleeping
- Keep track of the time at which each thread must wake up
- Blocking a thread when it sleeps (no busy waiting)

To keep track of sleeping threads, we created a global list which was updated every time a thread went to sleep. The absolute wake time was tracked by storing **wake_time** at thread level. And in order to block the thread, a semaphore was added to the thread (initialised to 0 so that it blocks the thread on first sema_down and unblocks on first sema_up).

Further the sleeping list was ordered in the increasing order of their wake time, so that we save the overhead of iterating to find the earlier wake time. In our first design we considered iterating over the entire list as opposed to sorting it, and later implemented sorting to optimize the solution.

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.

In thread.h

```
struct thread {
      struct list acquired_locks;
      struct thread *blocked_on;
      int initial_priority;
}
```

```
acquired_locks: a list of locks that each thread has acquired
blocked_on: thread that is currently holding the lock that current
thread is trying to acquire (for nesting)
initial_priority: to keep track of priority before donation, is set
to priority value before donation
```

In synch.h

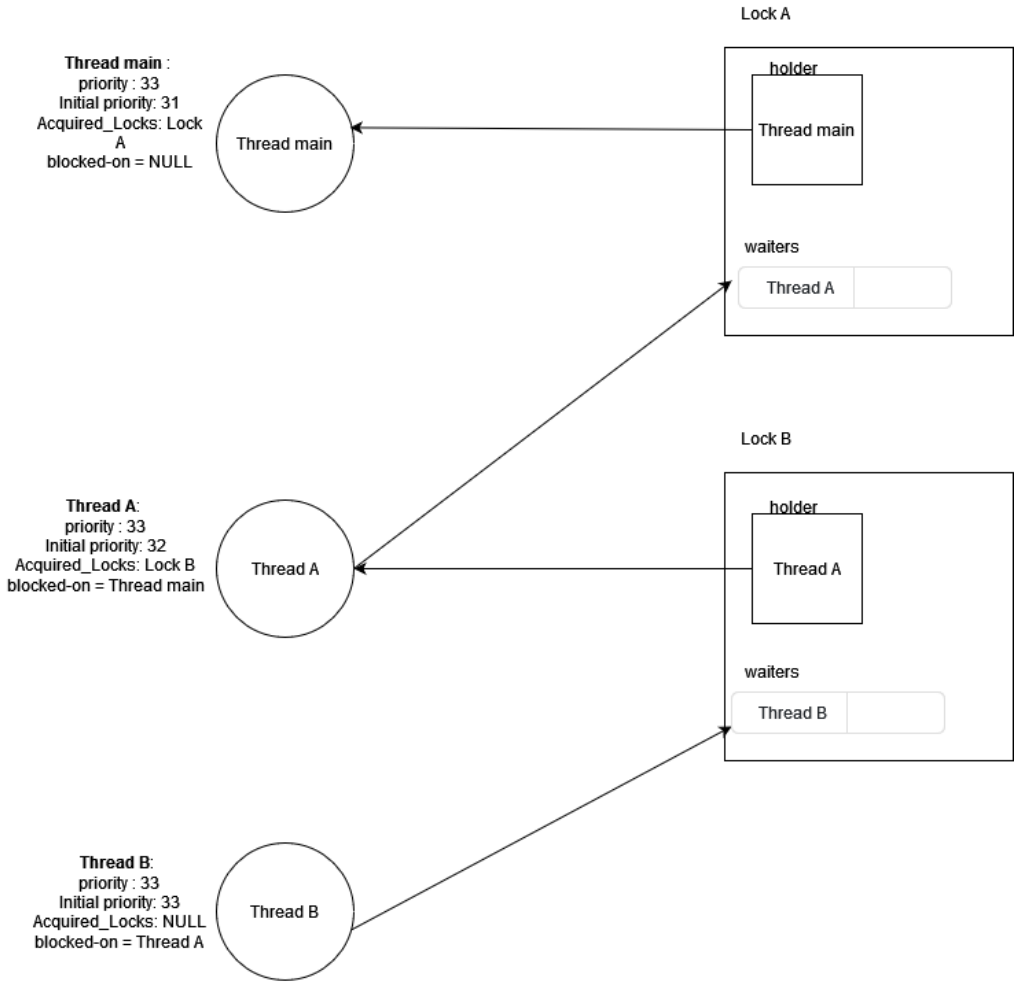
```
struct lock {
    struct list_elem elem;
}
```

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)

We are using a list of locks to track donation of priorities. Each thread will have a list of acquired locks. Further we are using the **waiters** list in lock to identify the threads that have previously donated to the current holder. If the priority of waiting threads is higher than the current thread, that means it will qualify for donation.

Nested donation occurs when a thread is trying to acquire a lock that is held by another thread which is in turn seeking for another lock (and so on).
In the given diagram below we have 3 threads - Thread main, Thread A and Thread B.
Thread main acquires Lock A and then becomes the holder for the lock.
Next Thread A with priority 32, and tries to acquire Lock B and acquires it. Now thread A tries to acquire Lock A but as it's currently held by another thread, Thread A is added to the waiters list of Lock A. As holder thread is having a lower priority, Thread A donates its priority to Thread Main hence its priority becomes 32.

Now consider another thread, Thread B (priority 33) wants to acquire lock B, as lock B is held by another thread (Thread A) and its having lesser priority 32, Thread B is added into the waiters list of Lock B and the priority is donated to Thread A (making its priority 32 to 33) and we check if Thread A is having blocked-on value as Lock A, The priority is recursively donated if the holder is having a lesser priority than Thread B. Hence Thread main is having a priority becomes 33 until its holding the aforementioned locks



---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
We ensure that the **waiters list** of locks, semaphores, condition variables are sorted in non decreasing order of their priorities. While waking (unblocking), we get the first element in the list, which would be of the highest priority.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?

1. Thread calls lock_acquire()
2. If the lock has no holder, the current_thread is made as the holder.
3. If the lock is already acquired (holder is not null) then the current thread priority would be compared with the holder's priority.
4. If the current thread's priority is higher, we set priority of the holder to the current thread's priority. If this was the first thread (i.e. the waiters list of lock is empty) that was called after lock was acquired, current lock is added to the acquired_lock list of holder thread. `blocked_on` for current thread is set to holder thread.
5. Current thread is added to lock's waiter list

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.
1. Check if the lock's waiters list has elements and current thread's priority != initial
    a. If yes, get the highest priority thread (i.e. the first thread) of the list
        i. If highest thread priority is equal to the current thread priority then, we find the next highest element in the waiters list of all locks and set the current thread's priority to that value
        ii. If there's only one element in the waiters list, we set the priority to initial value (**initial_priority**)
2. If the current thread's blockedOn != null, check if it has the same lock
3. Lock 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?
As we are not using thread_set_priority() during donation, we do not run into a scenario where a thread's priority could be changed by two threads at the same time. Further we have implementation of thread_set_priority() is present in lock_acquire() and lock_release() functionalities, here priority of a thread is recalculated based on the number of threads in the waiter's list.

---- RATIONALE ----

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

- To be able to keep track of all threads that have donated their priority to a thread
- Given a blocked thread, on which thread is it blocked on (i.e. the thread that acquired the lock first), so that we are able to reverse priority of the latter even when it has released the lock
- Initial priority of a thread before donation to be able to switch back to original once donation has to be reversed

To keep track of threads that have donated, we are maintaining a list of locks at thread level. The waiters list of lock's semaphore would give all the threads that have donated to the current thread. We considered keeping a list of threads at first, but that design did not hold good for the case of multiple locks i.e. when a thread acquired two locks and got donations from different threads. In this case we would have had to keep track of donor-lock relationships using a separate struct. In order to simplify the implementation we chose this.

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.
In `thread` struct:

```
fixed_point_t recent_cpu;
```

>> Recent CPU value for each thread

```
int nice;
```

>> Nice value for each thread
In thread.h

```
static fixed_point_t load_avg; //global variable
```

In thread.c
All of the constants for fixed point computation are defined globally and are initialised in `thread_init` so that they don't have to be computed repeatedly (at every timer tick)

```
fixed_point_t one;
fixed_point_t two;
fixed_point_t zero;
fixed_point_t frac59_60;
fixed_point_t frac1_60;
```

---- 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 ticks	recent_cpu A	B	C	priority A	B	C	thread 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	A
12	12	0	0	60	61	59	B
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	A
28	20	8	0	58	59	59	C
32	20	8	4	58	59	58	B
36	20	12	4	58	58	58	B

>> 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?

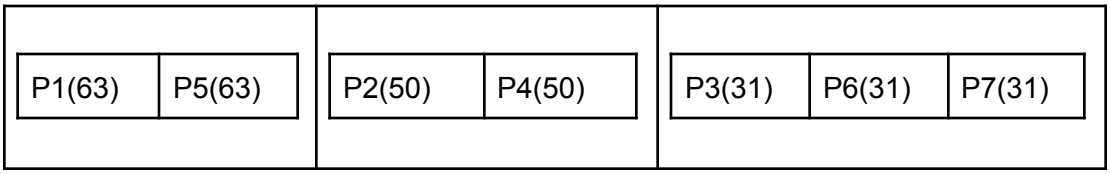
The specifications in Pintos documentation don't specify the order of function invocation for the running thread . Considering `priority_update()` uses `recent_cpu` to set new priorities for all the threads we have incremented the `recent_cpu` at every timer tick first. In our implementation at every second - first priority is updated and then `load_avg` and `recent_cpu` is updated - this is done so that the extensive calculation of `load_avg` and `recent_cpu` do not delay the update of priority.This does match the behaviour 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?
All of the updates to priorities, recent_cpu and load_avg happen within the interrupt context which is why keeping the update functions as optimal as possible was necessary so as to reduce the amount of computation that needs to be done every tick,every 4 ticks or every second.

---- 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?
In our design we are not using any new explicit data structures to redefine the mlfqs scheduling,we have implemented our scheduler by using the pre implemented priority queue with fifo (for same priority threads).
In our implementation, as we sort the ready list in the non ascending order of their priorities, the list automatically behaves as a collection of sorted lists of the same priority. Hence all of the same priority threads of highest priority run in FIFO first and only after their execution is the next list of threads of lower priority (in this case same list just lesser priority) invoked. The `update_priority()` which is called every 4th time tick updates all of the threads priorities (using nice, recent_cpu and therefore pushes the threads above and below in the priority queue.

Currently all of the update functions are called in the interrupt context - given time we would split the scheduling code and try to implement more of it outside the interrupt context for faster execution.
Following is a snapshot of ready list:



>> 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?
For implementing the fixed point computations we used the provided class fixedpoint.h which abstracted all of the fixed point operations. Fixedpoint.h has a struct fixed_point_t that emulates a fixed-point number. To optimise the code we have precomputed all of the constants in fixed point so that they don't have to be computed every timer tick.

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?