```
+--------------------+
|       CS 140      |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

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

Shawn Bachlet shawn.bachlet@ucdenver.edu
Lewis Sammons lewis.sammons@ucdenver.edu
Jordan Stein jordan.steint@ucdenver.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.

<sleepingList, struct List>

The sleeping list is a used as a queue of all threads that are currently “asleep” or blocked.

Threads are added to the queue in timer sleep and removed in timer interrupt.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),
>> including the effects of the timer interrupt handler.

The timer sleep function blocks the threads, which places the thread in a state of psuedo sleep. This thread is then placed into the sleeping thread queue. Within timer interrupt the threads member of wakeTicks is checked against the amount of ticks that have passed since the member was initialized(i.e. the value). In cases where the mlfqs scheduler is being used, the threads recent cpu value is increased and the load average is updated. The thread is then removed from the sleeping queue if they have “slept” for the right amount of time.

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

The only thread that is checked within the while loop where ticks are checked and threads are removed, is the thread at the front of the sleepingThreads list. If this thread doesn’t need to be woken up yet, the while loop is broken. If the thread does need to be woken the next thread in the sleeping list will be checked.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?

We disable interrupts when we are within timer sleep, this results in only one thread being in the function at a time as no other thread can take the current thread off of the CPU.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

As mentioned above interrupts are disabled, thus there is no possible way for a timer interrupt to occur until the end of the function where we re enable interrupts.

---- RATIONALE ----

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

This decision came mostly down to time. This method was figured out early in development, but we agreed to change to a method using a semaphore. However the semaphore implementation proved to be fruitless over time, and the decision to move back to the original design after closing in on time.

PRIORITY SCHEDULING
==================

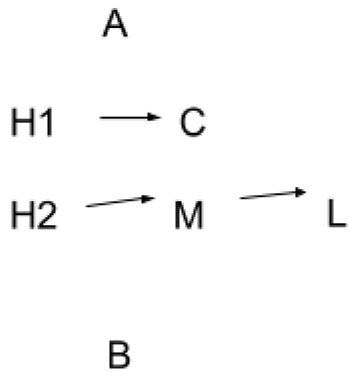
---- DATA STRUCTURES ----

>> 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.

<Members added to threads>

<base_priority, int>
Used within priority donation as the true priority of the thread.
<locks, struct list>
These are locks that are held for priority donation, each lock has it's own thread
<*lock_waiting, struct lock>
This is the thread's lock that is waiting for priority donation.

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



M's donation list: H1, H2 L's donation list: M
M's wait on lock: C
L's wait on lock: NULL
M's current donated priority is max(H1, H2, M).
L's current donated priority is max(L, M).

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
There is a different list added to thread which is a list of locks held for priority donation. each lock has a thread member that has it's own priority. This thread's priority is then donated accordingly, using a function called threadPriorityDonation. This will then call threadPriorityUpdate this will update the thread that is asking for donation, with the

donation.

>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

First a function called threadRemoveLock is called. Within this function the lock is then removed from the thread's list of locks. The thread's priority is then updated and then we return to lock_release(). Back in lock_release, the locks thread value is set to NULL and then we call sema_up() using the locks sema value. This ends lock_release().

---- 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?

We disable interrrupts within this function causing only one thread to be in set_priority. A lock is a safer way to implement this, however like with the alarm this decision came down to time. The way to implement the lock would to have a lock be created at the start of the function and have the first thread to get to set_priority lock it. Any thread that that cannot access the function can be placed on a sleeping threads list until the lock opens.

---- RATIONALE ----

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

Having each thread have an individual lock member is included in the initial code as a comment yet was not initially included in the thread struct, so adding this member was something that was shown by pintos already.

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.

<nice, int, thread.h, member of thread>
The threads “niceness” value. Determines how often it'll donate priority/ let other thread jump it on queue
<recent_cpu, int, thread.h, member of thread>

The recent cpu activity, used in mlfqs scheduling and load_avg.
<load_avg, int, thread.c, global>

---- 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	4	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	59	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 two threads have equal priority choosing which thread to run was slightly ambiguous. The solution was to use the thread that has spent the longest time not running. Often this would be C or B as A always has higher or equal priority, whereas C and B are always lower or equal A, running less often. This does match the behavior of our scheduler as can be seen within mlfqsPriorityUpdate()

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

A large percent of the computation occurs within the interrupt context,
this works well for this particular system as the amount of threads is relatively low compared to many modern thread schedulers, thus not causing too many performance issued. However if this design were to be used in a system with a large amount of threads this algorithm would run into many problems with 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?

Like the rest of the “rationale: portions of this document, much of this critique is centered around the fact that we chose to disable interrupts instead of locking them. Other issues present is the fact that we used linked list’s(kind of given to us anyway) to insert and sort the threads. Using other data structures would be beneficial for performance.

Though our disadvantages are performance based, all of our design decisions allow for easy to understand, easy to implement code. Not locking the threads results in no thread variables being locked saving some headaches. Also less variables are created due to this. This saves a small amount of memory, a very small amount of memory.

The biggest thing we could change for our implementation would be to use locks instead of disabling interrupts. Implementing locks would be the overall “correct” solution to this assignment, yet becomes a hassle with such strict time constraints.

>> 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?

The fixed point arithmetic is implemented within a header file known simply as fixed-point.h. Many of these functions can be taken directly out of the pintos documentation, they just need to be transposed into C code. These functions are used within our mlfqs scheduler, mainly for calculating load_avg and recent_cpu. Using these functions are useful for the readability of the code, as the amount of code going into these calculations is slightly absurd for single lines of code.

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?

The assignment seems to be split into three different sections. The alarm clock, priority scheduler, and MLFQS advanced scheduler. The alarm clock is a fairly easy part of the assignment, the priority scheduler was a fair challenge that can stump you. However the MLFQS is the real difficulty of the assignment. It can easily break your already working priority scheduling causing a headache in terms of having to fix your priority scheduler and in terms of still having to fix your MLFQS.

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

Without a doubt. This assignment really brought understanding into How threads work, priority queue's etc.

>> 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?

I think some hints on how to use the lock and semaphore structs would be helpful. There is one thing to know how one of these works, and another to know how to use them properly.

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

Having a TA would be helpful.

>> Any other comments?