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

---- GROUP ----

Ryan Wilson <rtwilson@stanford.edu>

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 timer.c, added global variable:

static struct list sleep_list;

List that keeps track of the threads which are sleeping.

- In thread.h, added variable to thread struct:

int64_t ticks;

If a thread is sleeping, ticks indicate the tick value when the thread is done sleeping and ready to be unblocked.

---- ALGORITHMS ----

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

In timer_sleep:

1) Check for valid ticks argument (i.e. ticks > 0).
2) Calculate the tick value for the thread as described above. This is calculated by adding the global ticks (ticks since the OS booted) to the ticks argument.
3) Add the current thread to the sleep list. Note that it is added in sorted order such that the front thread element ends its sleep the soonest.
4) Block the thread

In the timer interrupt handler:

1) Get the beginning front list thread.
2) If the thread's ticks value <= the global ticks, the thread is removed from the sleep list and unblocked.
3) Repeat steps 1-2 until the sleep list is empty or the thread's tick value > the global ticks.
4) Test to see if the current thread is still highest priority since other threads may have been unblocked.

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

Note that by keeping the sleep list in sorted order, this minimizes the time in the interrupt handler. Thus, the handler does not have to iterate through the entire sleep list at every interrupt.

---- SYNCHRONIZATION ----

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

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

Interrupts are turned off for steps 2-4 in timer_sleep. Since ready and sleep lists are read / modified in the interrupt handler, steps 3-4 need to be

turned off. Step 2 is turned off as well since if the current thread tick value is calculated, but then the thread is pre-empted or interrupted, the global ticks could eventually be > the tick value calculated and the thread would never had slept!
Step 1 does not allow race conditions since it is a local variable and if the ticks value is invalid, then the thread should not sleep anyways.
---- RATIONALE ----
>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
Before consulting with the TA's, I considered keeping the sleep list unsorted. While this would make adding sleeping threads more efficient, more time would be spent in the interrupt handler. This could cause a problem when there are a large number of sleeping threads. Thus, I felt the current design is superior based on that evaluation.
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, the following variables are added to the thread struct:
int init_priority;
- The baseline (non-donated) priority of the thread
struct lock *wait_on_lock;
- The lock the thread is waiting on (NULL if not waiting on any lock)
struct list donations;
- The list of other threads waiting on locks the thread has. Thus, these are possible priority donors.
struct list_elem donation_elem;
- The list element which can be added to another thread's donation list.
>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation. (Alternately, submit a
>> .png file.)
A, B, C are locks
H1, H2, M, L are threads
A
H1 ---> C
H2 ---> M ---> L
B
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?
In a semaphore, the list of waiters denotes the threads waiting for the semaphore to be upped. Thus, waiters are inserted such that the front thread has the highest priority. Note that this waiters list is also

sorted after the semaphore is released since as best put on the Google Groups page, "things change under the hood between acquiring and releasing."
>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
1) The current thread updates its wait on lock variable to the current lock.
2) The current thread adds itself to the lock holder's donations list.
3) Priority is donated iteratively via the following process (written in pseudocode):
- thread = current thread
- lock = lock current thread is waiting on
while (lock exists)
{
- if there is no holder for lock, return
- if the lock holder has a bigger (or equal) priority than thread's priority, return
- Set the lock holder's priority = thread's priority
- Update thread = lock holder
- Update lock = lock thread is waiting on
}
Note that this iteration has a depth of 8 in order to prevent infinite loops due to deadlocking threads.
>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.
1) The lock holder is updated to NULL.
2) Removes the thread's waiting on the just-released lock off the current thread's donation list. Simply, the priority donation is gone once the lock is released.
3) Refresh the current thread's priority. This means finding the highest priority thread on the donation list.
4) The highest waiting priority thread acquires the lock and is put on the ready queue.
5) Check if the current thread still has max priority. If not, yield the processor.
---- 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?
Interrupts are turned off in thread_set_priority(). This is because we have to read / write to the current thread's priority, which is updated every 4 ticks in the interrupt handler. Because of this, we cannot use locks since the interrupt handler cannot acquire locks.
A potential race condition would be while the thread priority variable is being updated to the new priority, the interrupt handler is writing to the priority variable. Thus, these conflicting writes could mangle the priority variable, therefore being a race condition.
---- RATIONALE ----
>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?
A lot of the reason I chose this design is because I had to add the least amount of variables possible (especially without allocating heap memory). It also made easy use out of the linked list implementation given

to us. I was conflicted with using another design where each thread had a
list of acquired locks, but determined refreshing priority (after a lock
is released), finding the next highest priority would be inefficient as
would sorting a list of locks (with waiters for each lock).
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.h, the following variables are added to the thread struct:
int nice;
- The thread's current nice value
int recent_cpu;
- The thread's most recently calculated recent_cpu value
In thread.c, the following global variable is added:
int load_avg;
- The system's most recently calculated load average value
---- 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:
Assume time slice = 4 ticks.
timer recent_cpu priority thread
ticks A B C A B C to run ready list
----- -- -- -- -- -- -- ------ ----------
0 0 0 0 63 61 59 A B, C
4 4 0 0 62 61 59 A B, C
8 8 0 0 61 61 59 B A, C
12 8 4 0 61 60 59 A B, C
16 12 4 0 60 60 59 B A, C
20 12 8 0 60 59 59 A C, B
24 16 8 0 59 59 59 C B, A
28 16 8 4 59 59 58 B A, C
32 16 12 4 59 58 58 A C, B
36 20 12 4 58 58 58 C B, A
>> 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?
Yes, it is unclear if two threads have equal priority, which thread is
supposed to run. I used the following rules:
1) If the running thread has the highest priority and so does a ready
thread and the running thread reached its time slice, the running thread
continues to run. This is equivalent to all highest priority threads
running round robin.
2) If the scheduler has to choose between multiple ready threads, it
chooses the one that has been run the least recently (i.e. placed first
on the ready list).
This behavior matches my scheduler, as can be seen in test_max_priority().
>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

Due to the precise nature in which the scheduler variables needed to be updated, most of the computation needs to be done within the interrupt handler. However, I found that the currently running thread's priority only needs to be updated every 4 ticks, as it is the only thread that changes values of recent_cpu. Every second, however, the load average, recent_cpu and priority has to be recalculated over all threads, which is expensive. Thus, for a system with a lot of threads, this may be an inadvisable scheduling algorithm as it is likely to affect performance.
The only computation done outside the interrupt handler is when resetting the nice value, but the interrupts have to be turned off. This is because resetting nice also changes the priotity, which is read / written in the interrupt handler.
---- 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?
Advantages:
- Simple to understand since no locking of thread variables (only turn off interrupts).
- Minimal adding of extra variables, kernel threads remain relatively small. Also makes for easier readability.
Disadvantages:
- Performance suffers due to turning off interrupts instead of locking variables.
- Ordered inserting and sorting lists takes O(n) and O(n log n) respectively. Using binary tree / other more efficient data structures may be needed. (Also, bad caching in linked lists)
To refine my design, I would implement the following features:
- Automatic deadlock detection
This is equivalent to finding cycles in the lock donation graph, which is iterated through in the donate_priority function. A maximum iteration depth of 8 is set in case there is a deadlock cycle (without a depth parameter, this would cause an infinite loop).
What would be optimal is to detect deadlock cycles and lower all the deadlocked threads priorities to PRI_MIN / kill the threads.
- Detect overflows in fixed_point.h
The values for priorities and nice are clamped, but the values for recent_cpu and load_avg are not. Hence, this leaves the possibility for overflow, which I currently do not check for. To be correct, the OS needs to check for this.
- Use locks for variables instead of turning off interrupts (if possible)
Since most variables I introduced are used / written to in the interrupt handler, most of my synchronization is essentially done via turning off interrupts. To be correct and improve the speed of the system, a detailed analysis of variables and their reads / writes needs to be done in order to implement locks on variables where possible.
>> 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?
I implemented fixed-point math in a header file. The conversions between integers and fixed-point and arithmetic was abstracted away in this file. I used the standard functions as described in the Pintos documentation and called these functions in my mlqfs calculation functions in thread.c. Abstracting the fixed-point functions allowed for better readability when calculating the mlqfs thread.c functions.

SURVEY QUESTIONS
==========

>> 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?
This assignment was quite challenging and took me 4 days straight to complete. I found figuring out the data structures for priority donation to be quite challenging, but all in all, I enjoyed the assignment despite its difficulty.

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?
I felt dealing with synchronization gave me newfound respect for kernel developers who develop OS's used on multi-processor hardware.

>> 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 would give more hints with regards to the priority donation problem. Luckily, I've had a background in graph theory and mathematics, but even I struggled with figuring out all the donation cases.

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?
More office hours would be greatly appreciated. However, I loved the plethora of Pintos documentation!