```
+--------------------+
|       CS 162       |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT  |
+--------------------+
```

---- GROUP ----

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

Andrew Blatner <ablatner@berkeley.edu>
Bryan Tran <btacju@gmail.com>
Kevin Lam <kel018@berkeley.edu>
Noah Ruderman <noahc66260@domain.example>

---- 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.

```
/* Queue of threads put to sleep by timer_sleep */
static struct list sleep_list;


struct thread
  {
      ...
    struct list_elem sleepelem;         /* List element for sleep list. */
    int64_t sleep_until_tick;           /* Wake up no sooner than this tick*/
      ...
  };
```

---- ALGORITHMS ----

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

timer_sleep() sets the thread's sleep_until_tick and inserts it into sleep_list in sorted order. Then it disables interrupts, sets the thread's status to THREAD_BLOCKED, calls schedule(), and enables interrupts.

timer_interrupt() now additionally calls a function that checks the front of sleep_list for elements with a wake up time that has passed. It adds each to ready_list, removes them from sleep_list, and sets their statuses to THREAD_READY.

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

The amount of time spent in timer interrupt handler can be minimized by having the waiting list be in sorted order. There would be no need to search through and check each thread for the number of ticks remaining until wakeup -- only the front of the list would have to be checked in the sorted version. Finding a thread to wake up is an O(1) operation for a sorted list, as opposed to the O(n) search time for an unsorted list.

---- SYNCHRONIZATION ----

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

We can use locks to avoid race conditions during accesses to sleep_list.

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

Locks can solve this, as well, since a lock would make it so only the current timer_sleep() thread could access the waiting queue.

---- RATIONALE ----

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

We considered a design where the queue and timer handling would occur in timer_sleep and schedule. However, another method was chosen because there are many other functions that call schedule and changing it could produce many errors. Also, a design where an unsorted waiting list was considered. An unsorted list would have O(1) insertion time but O(n) search time. A sorted list would have O(n) insertion time and O(1) search time. Since we want to search the waiting list on each timer interrupt (once per tick), we chose the sorted list to make a very frequent operation (checking for and finding a task to wake up) as efficient as possible.

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

---- DATA STRUCTURES ----

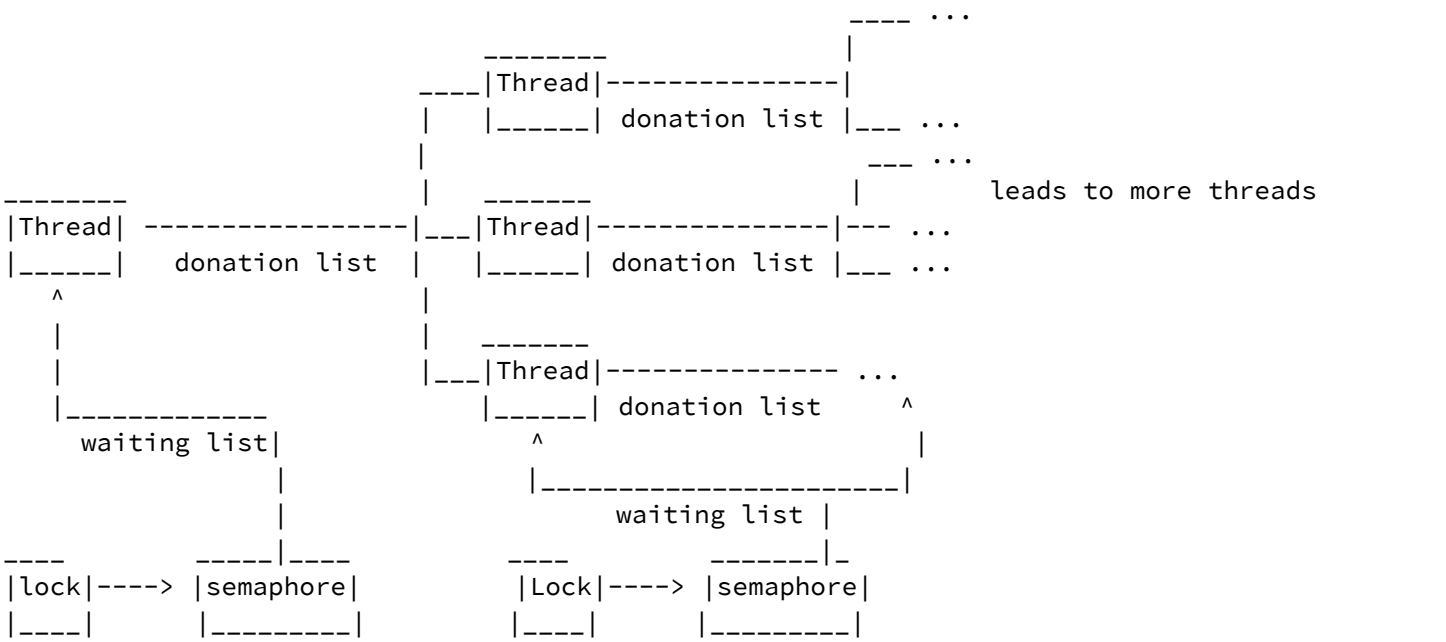
>> B1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

Struct thread
{
      ...
      struct list donations;
      ...
}
   - to keep track of the donations given. Each element in donations is some thread that must finish before the current thread can continue.

Struct list ready_list
   - the ready_list is going to be sorted by priority from high to low.

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

Priority donation is determined by keeping track of all the threads each thread is waiting on a lock from, in other words, all the threads a thread should be donating to. We also keep a list of all the threads waiting on a lock in the semaphore.



There could be many different locks' semaphores pointing to the same threads, since a thread could be waiting on more than one lock. The lock's semaphore simply points to the threads that are waiting on that lock. Threads don't necessarily have to be waiting on other threads that are waiting for the same lock; a thread could be waiting on a thread that acquired the lock originally needed while still waiting on a different lock.

---- ALGORITHMS ----

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

For each shared resource, there is a list of waiting threads. When the resource is available, the thread with the highest priority is removed from the waiting list and is inserted into the ready list. The waiting list will be sorted by priority; This way we can always pull the thread off the front of the waiting list, and it will be the highest priority thread.

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

When a call to lock_acquire is made, the current thread is added into the lock's semaphore's waiting list. While the lock exists, if the priority of current thread is larger than the priority of the lock holder, set the lock holder's priority to the current thread's priority.

To handle nested donation, we modify thread_set_priority() to check the priorities of all the threads it should donate to, which we kept track with a list. If the priority of the current thread is higher any of the threads it should donate to, it would raise those priorities as well, which would recursively raise the all the necessary priorities.

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

When lock_release() is called on that lock, the lock will check its semaphore, which contains a list of the waiting threads. The list of waiting threads will be ordered, by priority, so the first thread on that list will then be woken up and able to acquire the lock.

---- 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?

Thread_set_priority accesses the ready list through thread_yield(), since if we ever lower the priority of the current thread and there is something else with higher priority, we must

yield. But at the same time, our code in thread interrupt to handle timer_sleep also modifies the ready list, which could create a potential race. We can use a lock on the ready list modifying part of thread_yield(), so that the interrupts will not be able to modify the ready list while thread_yield() is.

Potential lock issue: there may be an issue with using the lock since thread_ticks() accesses both the ready list and waiting queue from part one, but if the lock is currently acquired for those, the threads may pile up waiting for the lock since a new process will be requesting that lock every tick of the OS.

---- RATIONALE ----

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

Initially, we chose to call thread_yield after donating which would put the thread at the end of the ready list but we chose not to use it because we are going to keep the ready list sorted. We can still call thread_yield, but we have to modify thread_yield so that rather than pushing the thread to the back of the ready_list, it inserts the thread by priority into the ordered list.

Rather than keeping track of which lock a thread was waiting on, we decided to keep track of all the threads that either have or are waiting on a lock the current thread is waiting on, so it would be easier to keep track of donations. This makes it so it’s easy to actually do the donations whenever a thread tries to acquire a lock, or it’s priority gets raised itself.

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.

```
/* Keeps track of number of running and ready_threads for load_avg calculation.
   Any change to the length of ready_list must update this. */
static int ready_threads;

/* Contains the 64 statically allocated ready lists.
   Each is initialized in thread_init. */
static struct list mlfqs_ready_list[PRI_MAX+1];

struct thread
  {
```

```
    ...
    int nice; /* Nice value for scheduling */
    ...
  }
```

We will also want to precalculate per-thread values used repeatedly in the advanced scheduler such as PRI_MAX-nice*2 (for calculating priority)

---- 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	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?

Yes, the scheduler specification does not say how to break tied priorities. However, we decided to break ties by choosing the thread with the minimum recent_cpu.

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

As the project specification says, the scheduler updates should occur before any ordinary kernel thread has a chance to run, so much of the advanced scheduler’s work should be done in the timer interrupt context. Most of the work done in the timer interrupt is calculating priorities, which happens every fourth tick for each thread because calculating the load_avg and recent_cpu’s happens only once each second. However, moving threads between the 64 queues can happen efficiently if we keep them sorted.

Outside interrupt contexts, the major things we can do are keeping track of the number of ready and running threads and precalculating values for the simulated floating-point arithmetic.

---- 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:
   - Keeping the ready lists sorted makes searching for high priority threads to be more
     efficient
Disadvantages:
   - Inserting threads and keeping them sorted is not as efficient

If we had more time, we could have implemented a heap or used binary search to search through the lists for better efficiency.

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?

>> 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 semesters 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 semesters or the remaining projects?

>> Any other comments?