```
+--------------------+
|    CSE 421/521     |
| PROJECT 1: THREADS |
|   DESIGN DOCUMENT|
+--------------------+
```

**---- GROUP ----**

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

Roopa Chandrashekar<roopacha@buffalo.edu>
Bhupika Gautam<bhupikay@buffalo.edu>
Neeraj Abhyankar<nabhyank@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.

http://stuartharrell.com/blog/2016/12/16/efficient-alarm-clock/

**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.

Add Global Variable in timer.c file home/pintos/src/devices

static struct list waiting_list/*keeps a track of sleeping process*/

Use variable int64_t ticks in thread.h to track the number of ticks a process is sleeping and let variable make_thread_wake_up try to wake up the thread after the end of sleeping process.

**---- ALGORITHMS ----**

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

In timer_sleep() function
a) Check for timer ticks > 0.
b) Calculate the ticks value using int64_t ticks.
c) Put/Add the current thread to the waiting_list. Add it in the sorted order such that the first element can be woken up as soon as the ticks end for sleep.
d) Block the thread if above not true.

In Timer interrupt handler
a) Find the first element thread.
b) Calculate if the thread tick <= global tick, and if true remove thread from waiting_list and also unblock it. Continue this process i.e. a) & b) till waiting_list are empty and are woken up when ready and sent forward to the next process step.
c) Put a test to see if the current thread is of the highest priority in the waiting_list.

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

By maintaining the accuracy of the sorting of the threads, this will minimize the time of each thread in the interrupt handler so that the iteration process won't take place often.

**---- SYNCHRONIZATION ----**

>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
Since interrupts are disabled, only one threads will be allowed to be added to the wait list, even if multiple threads call timer_sleep simultaneously they will have to wait till they are allowed. Thus a race condition is avoided.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
While adding the elements to the waiting list the interrupts are disabled and there occurs no race condition during a call to timer_sleep().

**---- RATIONALE ----**

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

**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.

Added in threads.c
1) static struct list donor_thread_list/* renamed see 11) */
2) void check_and_preempt_thread (void) /*check and preempt the thread before the priority scheduling*/
3) bool compare_lock_priority(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED); /*Compares the priorities of locks assigned to the threads*/
4) void donate_thread_priority(struct thread *); /* Function declaration to handle priority donations in threads */
5) void update_thread_priority(struct thread *); /* Function declaration to handle priority donations in threads */

6) void set_thread_lock(struct lock *); /* Function declaration to handle setting locks on threads especially for priority donation. */

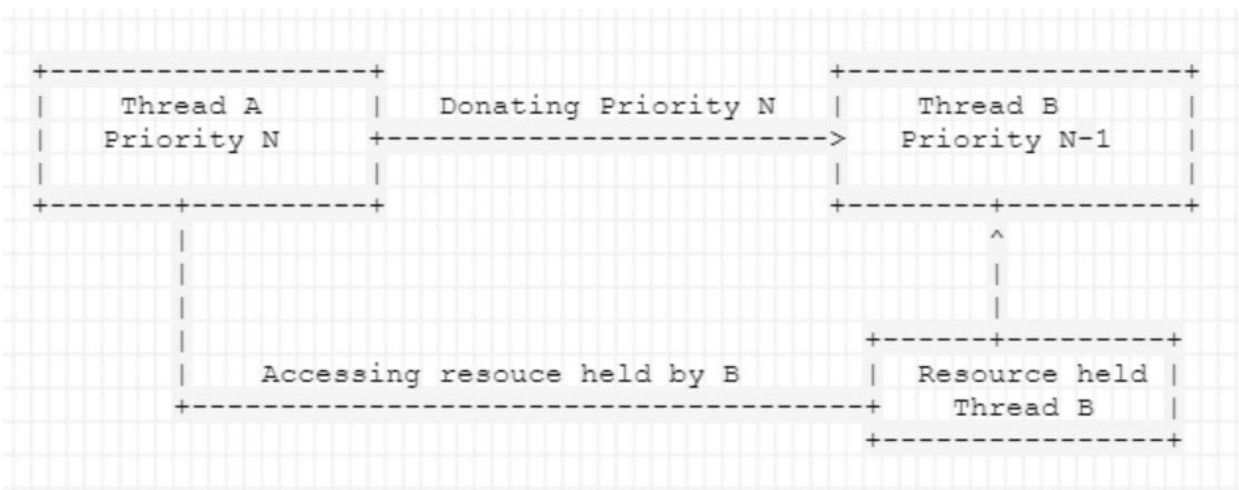
7) void reset_thread_lock(struct lock *); /* Function declaration to handle setting locks on threads especially for priority donation. */

Added in thread.h
9) int base_priority;
10) struct list locks; /*maintains a list of locks*/
11) struct lock *lock_waiting_donated_priority;
/*keeps track of donor threads */

Added in synch.c
12) #define PRIORITY_DONATION_MAX_DEPTH 8

Added in synch.h
13) int max_priority;

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




**---- ALGORITHMS ----**

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
Whenever there is a new entry in the list of waiters, the element is entered in the sorted order of its priority. So, whenever a thread has to be woken up the first thread in the list is signaled. Since the list is being sorted with respect to the order of priority, this ensures the waking up of highest priority thread.

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

Priority Donation: Consider two threads,

Thread A High Priority
Thread B Low Priority

Step 1-
Thread A has higher priority, and is scheduled for execution in the CPU.

Step 2-
Thread A needs some resource to continue its execution, but the resource is held by a low priority Thread B which is either in the ready_list or waiting_list.

Step 3-
To allow Thread A to continue execution, a call to lock_acquire() will donate the priority of Thread A to Thread B by calling donate_thread_priority() for a given TIME_SLICE. During this time Thread A is placed in lock list pointed by lock_waiting_donated_priority.

Step 4-
Thread B executes in the CPU and releases the lock on the resource it held by calling lock_release().

Step 5-
Thread A is popped out of ready_list and is scheduled to execute in the CPU.

Nested Donation: Consider more than 2 threads, where thread c holds a resource required by a higher priority thread b which holds a resource required by another higher priority thread a.

Thread A donates priority(N) → B donates priority(N) → C now holds priority(N)

Execution: (thread C) executes, lock passed to thread B → (thread B) executes, lock passed to thread A → thread A executes

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

Considering a simple priority donation scenario,

Step 1-
Thread B executes in the CPU and releases the lock on the resourceheld by calling lock_release() which calls lock_held_by_current_thread()to reset the value of the lock.

Step 2-
If
Thread B has completed execution then its status is updated and exits the scheduler.
Else
The priority of Thread B is updated and then placed on the ready or wait list.

**---- 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?

Yes we can use locks to avoid this race condition. There can be a potential race condition if the thread has received a donation of priority from another thread and at the same time it tries to set its own priority. To avoid this, before we change the priority of the thread to new priority we check if the thread has received donation and has any locks. If it does, we only update the base priority of the thread so that after releasing the locks the priority can be changed back to new priority.

**---- RATIONALE ----**

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

This design provides a solution for priority inversion problem, using ready list, wait list and donor list makes it easy to schedule the threads from high to low priority.

**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.

Added in thread.h

int nice; /*The nice value for each thread*/

int recent_cpu; /*CPU time received by each thread*/

Added in thread.c

int load_avg; /*Average threads ready to run over the past minute*/

void update_thread_priority_mlfqs(struct thread *t) /*updates and calculates the new priority every four clock ticks*/

void increment_recent_cpu(void)/*increments the recent_cpu value by 1 for the current thread*/

void update_recent_cpu(struct thread *t)/*updates and calculates the recent_cpu value for the given thread*/

void recalculate_per_second(void) /*iterates the calculation of the recent_cpu, load_avg, priority value per second*/

Added fixed-point.h /*Contains the fixed-point arithmetic operations*/

**---- 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 | 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?

When the priority of two or more threads is the same, at that time which thread should run is not certain. And thus causes ambiguity. To resolve this ambiguity, we have used the 'First In First Out' scheduling to decide the next thread to run. This behavior 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?

Most of the coding has to be done inside the interrupt context; this would have an negative impact of the performance of the Operating System.
The resetting of the values can be done outside the interrupt context.

**---- 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?

The 64 queues in 4.4BSD Scheduling is not implemented, but the functionality is the same, if had more time the implementation of the 64 queues would be done.
The design is using first in first out scheduling for the processes with the same priority. The priority scheduling can lead to starvation of the lower priority scheduling hence we can also calculate the aging of the processes.

>> 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?

Fixed point arithmetic is needed to convert the recent_cpu and load_avg of a thread into integers as they are real numbers and pintos does not support floating point arithmetic in the kernel. We are implementing the arithmetic for fixed-point in a separate file fixed-point.h.

## 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?