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

**Group : Pint-us**

**Allen Daniel Yesa <allendan@buffalo.edu>**
**Bharadwaj Parthasarathy <bparthas@buffalo.edu>**
**Nimisha Philip Rodrigues <nr57@buffalo.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.**

- **struct list waiting_list** – to maintain threads that are sleeping
- **int64_t ticks –** added in thread structure to maintain the sleep time of each thread

## ---- ALGORITHMS ----

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

in timer_sleep():
- Validating ticks for negative and zero values
- Block the current thread and assign computed tick value to it
- Add the thread element which has to sleep to the waiting_list in a sorted order such that the thread to be awakened first will be in front of the list
- Schedule the next thread to run

in timer_interrupt():
- check for all threads that have elapsed their sleep time
- Remove the threads whose sleeping time has elapsed from the waiting_list
- Unblock the threads

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

**Solution:**
In timer interrupt handler, when finding which all threads need to be awakened we need to iterate through the waiting_list each time an interrupt handler occurs. This may not be a good solution as it may take some time to iterate through the list.

**Optimised solution :**

In order to overcome the iteration during interrupt handling, we insert the thread to the list in sorted order of elapse time. This means that the thread whose sleep time is to elapsed soon will be in the front of the list.

---- SYNCHRONIZATION ----

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

In timer_sleep() we disable interrupts because we are manipulating the current thread values.

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

No race condition occurs

---- RATIONALE ----

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

Initially we decided to iterate the waiting_list each time then interrupt occurs. This is a solution of O(n). Hence we decided to sort the list during insertion which means the complexity reduces to O(1).

## 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.**

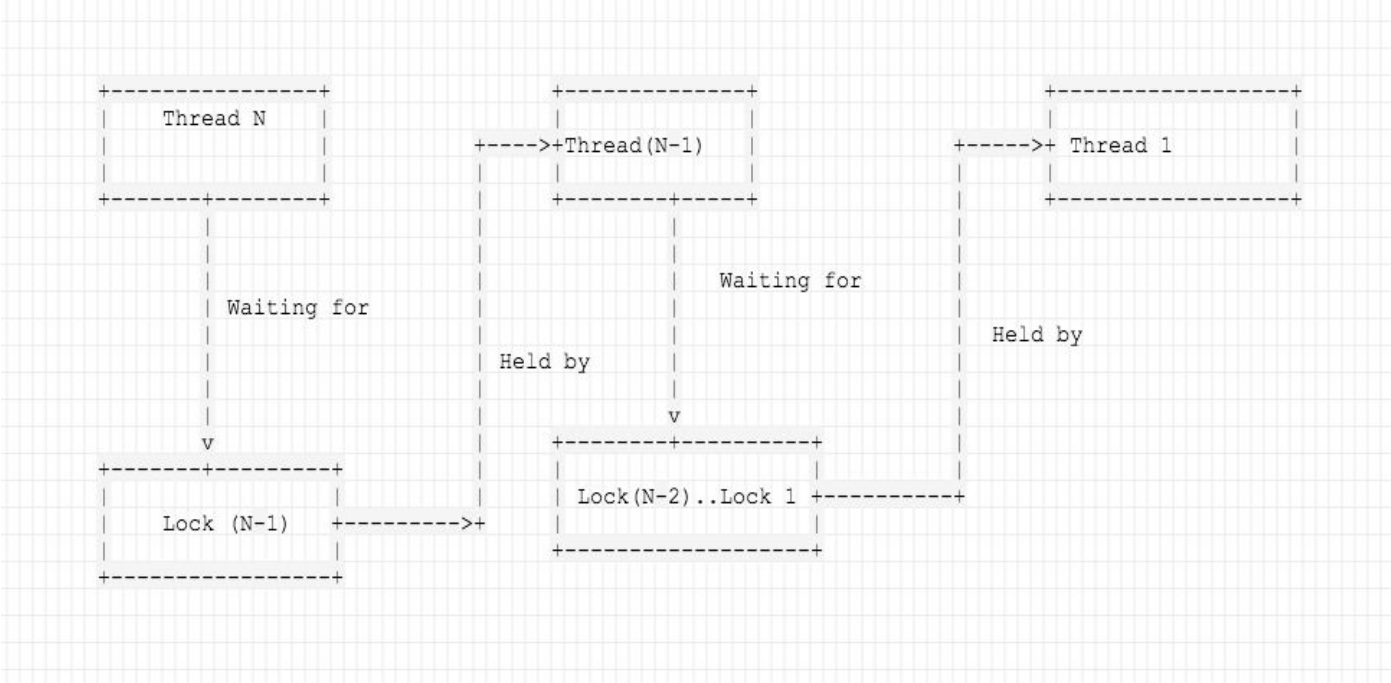
**int orig_priority** added in thread structure to maintain the original priority given to the thread.

**struct list donor_list** added in thread structure to know which threads are the priority donors

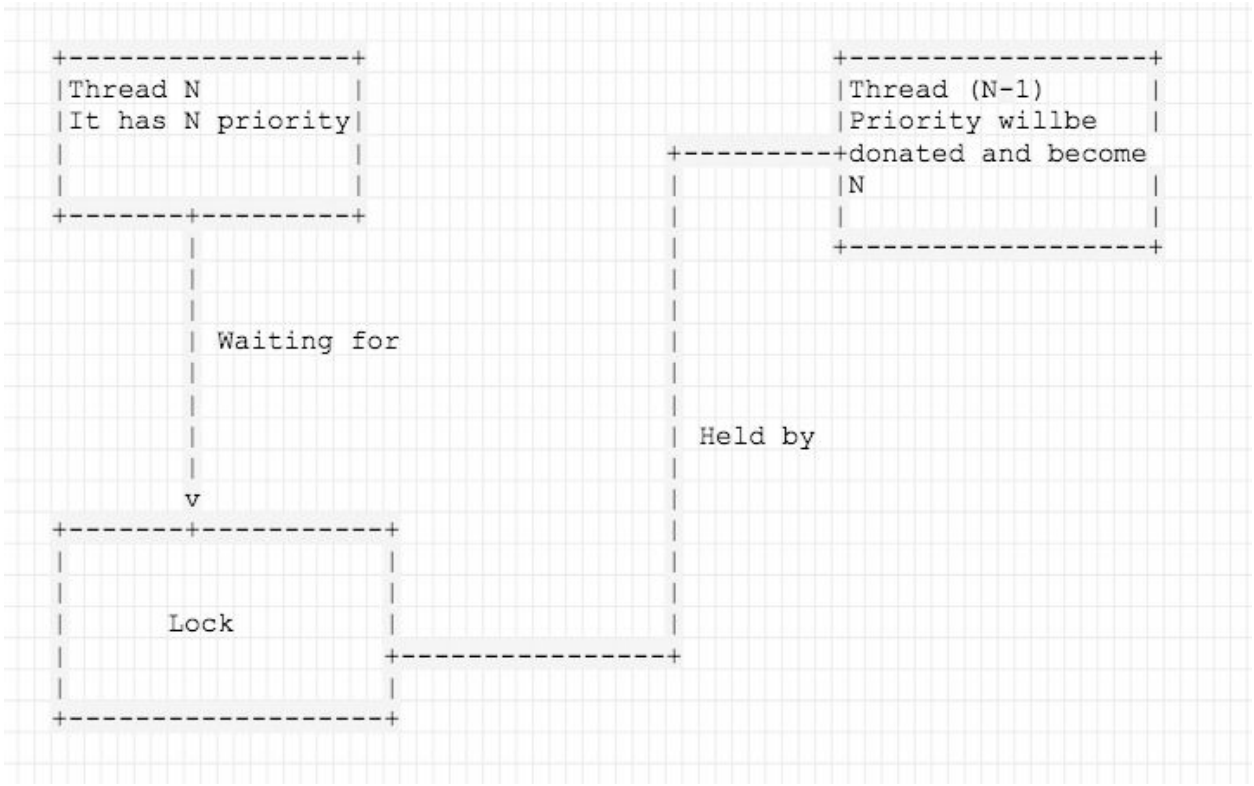
**struct lock lock_thread** added in thread structure to know which lock the thread is waiting for

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

The below ASCII diagram is shown for N threads and corresponding N locks.According to the ASCII diagram below every thread is waiting for a lock which is held by a thread having low priority value,because of which none of the threads can complete its execution




The diagram below is the implemented concept of priority donation, thread N donates its priority to thread (N-1)which has new priority of N, because of which it will be executed first the lock will be released and given to Thread N for further execution.This will be done for all i and i-1 until priority of i-1 is greater than priority of thread N or until we have no more threads waiting for lock.




---- ALGORITHMS ----

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

When threads are added to the waiting_list, they are sorted in descending order based on the priority. So when a lock or semaphore is released the high priority thread gets the lock and starts execution

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

- Thread A is executing and requires a lock "a" held by thread B which is waiting and has less priority than A
- Thread A donates its priority to Thread B
- Thread B executes
- Thread B while executing requires a lock "b" held by Thread C
- Thread B denotes its priority (priority donated to it by A) to C
- C executes
- Each thread will have a donor list to maintain its donors

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

- When a lock is released the priority donation is stopped
- The thread that releases the lock gets its original priority back
- The high priority thread acquires the lock and start executing

---- 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?**

While changing the thread's priority interrupts are disabled since we are changing a thread's priority value.

No, locks cannot be used since lock cannot be used in an interrupt context

---- RATIONALE ----

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

Maintaining a high priority sorted donor list makes it easy during scheduling the threads when releasing the locks. Or else it will be a overhead to find which threads are associated with the locks and to schedule them.

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.**

thread.h

**int nice_value** - added to the thread structure to maintain the nice value of the thread

**int recentcpu_value** - added to the thread structure to maintain the recent cpu of the threads after recalculating during timer ticks

**int loadaverage_value** - global variable to maintain load average required to calculate priority and recent_cpu

---- 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, at some stage of the timer tick the priorities of two threads were equal so we faced an ambiguity in resolving which thread must run. In this case the thread that has had recent cpu time or whose time slice has expired yields. This has been handled in the method used to schedule the next thread based on priority

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

The Scheduler deals with recent cpu time and load average to calculate priority. Since these values change on every timer interrupt, the recalculation happens inside 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?**

We are extending our priority scheduler approach (which deals with sorting the priorities in descending order in a list) to the advanced scheduler. The advantage may be in the fact that our next executing thread is already handled and so the only focus is on calculating the priorities.

The disadvantage on the other hand maybe there are better Data Structures other than list to handle this. Given extra time, would possibly consider other efficient data structures that might handle the priority queue better

**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?**

We defined methods for each computation involving fixed point numbers. As mentioned in the pintos guide considered the fixed point numbers to be in the 17.14 format. So each computation involves multiplying or dividing by a constant (say f) that is equivalent to the bits shifted.

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?

**Priority donation and Advanced scheduler was indeed hard to start with**

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

**When working on alarm clock and mlfqs got major insights on timer interrupts and timer ticks.**

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?

**No, none was misleading. Maybe giving students a heads up on few of these programming concepts may actually help them to start working on their project early**

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

**NO**