```
                 +--------------------+
                 |        CS 301      |
                 | PROJECT 1: THREADS |
                 |   DESIGN DOCUMENT  |
                 +--------------------+
```

---- GROUP ----

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

Haikoo Khandor <haikoo.ashok@iitgn.ac.in>
Madhav Kanda <madhav.kanda@iitgn.ac.in>
Mihir Sutariya <sutariya.mihirkumar@iitgn.ac.in>

---- 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.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* sleeping implementation */
    int64_t sleep_ticks;
    struct list_elem sleep_elem;

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

1. New attributes sleep_ticks (int64_t) and sleep_elem (struct list_elem) added to the thread struct definition in the thread.h file.
2. One new static variable sleep_list(struct list) is defined in timer.c.

Purpose:

1. int64_t sleep_ticks; - Denotes the total time ticks at which the thread has to wake up.(if the

thread is in sleep)

2. struct list_elem sleep_elem; - Denotes the elements(type:list_elem) to be stored in the sleeping list.
3. static struct list sleep_list; - Contains the list of sleeping threads who will be woken up after sleep_ticks for them passes.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer_sleep(),

1. When any thread calls timer sleep, the thread attribute sleep_elem  will be added to the sleep_list. Threads will be added in non decreasing order on the basis of sleep_ticks variable.
2. Function will disable the interrupts and store its value in the variable, "lev". The current thread is then blocked and the interrupts are set to the previous value again.

>> including the effects of the timer interrupt handler.

1. Timer interrupt handler is keeping track of the ticks starting from when the cpu was turned on, so we can wake up our threads by adding some functionality in the timer_interrupt function.
2. In the timer_interrupt function we are checking if the front thread stored in the "top" variable has sleep_ticks less than ticks global variable. If it is, then we will unblock it, pop it and check for other threads, if not then we will break the loop. It works because if the front element has a higher value then all others will have higher values than the first one because the list sleep_list is sorted.

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

1. Brute force approach should not be implemented because the timer interrupt handler is the most frequently used function. The brute force method is to check all the blocked thread and its sleep_ticks attribute and check the condition. It will take O(number of threads blocked)
2. What we have implemented is we keep track of sorted lists so that every time we need to just check the first element of the list. This will take O(1) time.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer_sleep() simultaneously?
When multiple threads call timer_sleep () function simultaneously, the interrupts are disabled. Hence, whenever we access, add or delete anything from the sleep_list mentioned in A1, the operations i.e. read/write are atomic.

>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?
We disable interrupts in the critical part of the code in the timer_sleep() function. This means that timer_interrupt cannot occur. Thus there are no race conditions between timer_sleep and timer_interrupt due to disabled interrupts, i.e when we are inserting or deleting elements from sleep_list.

---- RATIONALE ----

>> A6: Why did you choose this design?  In what ways is it superior to
>> another design you considered?
Another design was to just block the thread when it calls timer_sleep(). When the global ticks are changing we will check all the threads sleep_ticks and wake_up thread if it is less than ticks. In our solution, this is not required since we are maintaining a sorted list of threads where the comparisons are done on the first thread which has the lowest time of sleep, if it is less than the current ticks, unblock and pop it, else we break the loop. Thus our design is superior in case of number of queries called and the time complexity.

One more design that we thought about was using semaphores. We will add the semaphore attribute in thread.h. We will use this to block the thread so the problem of synchronization is solved beforehand. But, this is a bit complex to understand and write the code. It has the same complexity and performance as our solution.

```
                  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
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */

    struct list_elem allelem;           /* List element for all threads list. */

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

    /* sleeping implementation */
    int64_t sleep_ticks;
    struct list_elem sleep_elem;
    /* priority scheduling elements */
    int priority;                       /* Priority. */
    int donated_prio[20];                 /* Priority list for the donations*/
    int total_donation;                    /* Store the number of donation locks */
    int length;                           /* donated priority list length */
    struct lock *waiting_for;           /* Lock for which a blocked thread waits */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif

    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

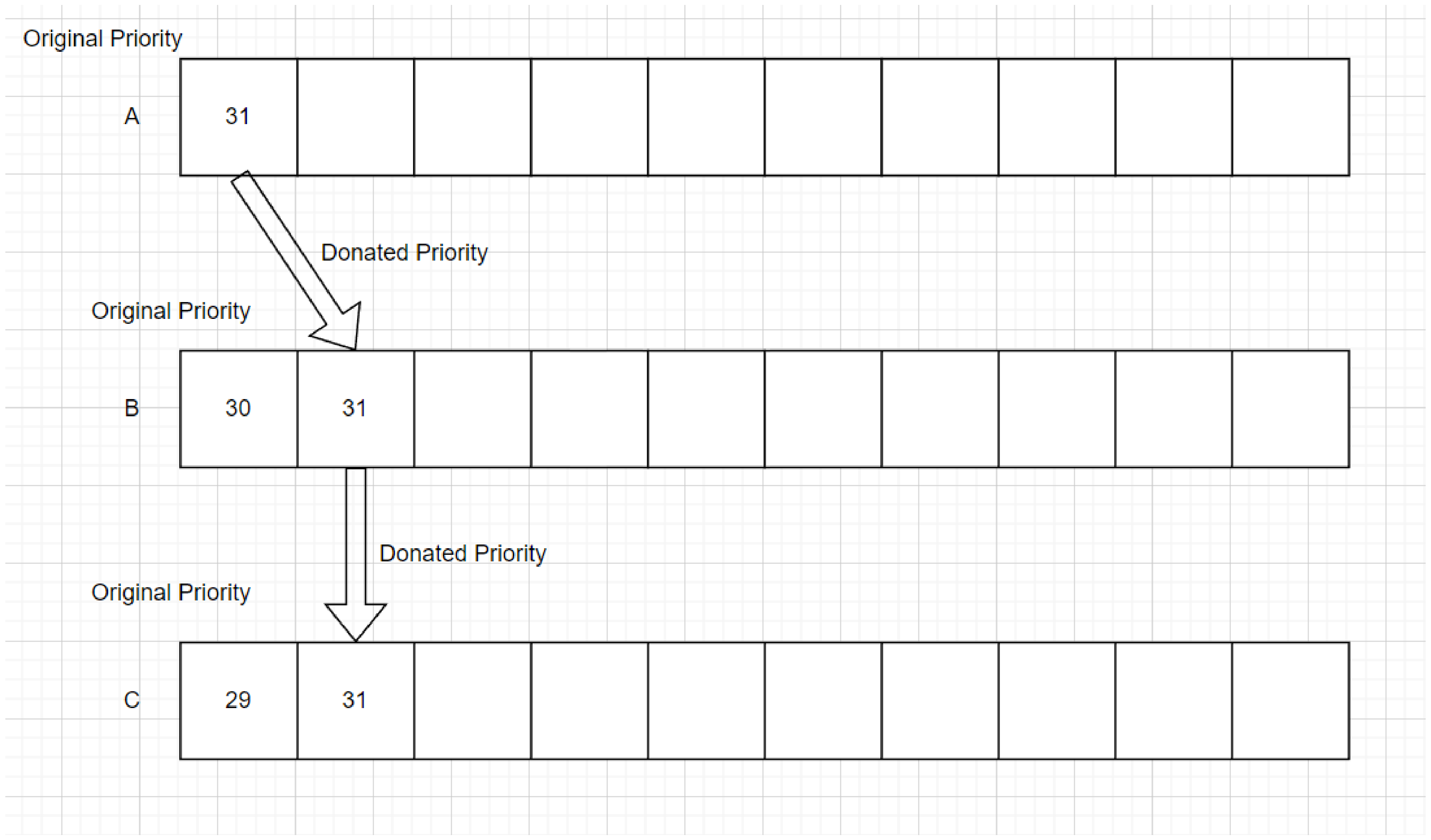
1. In struct thread, 4 new attributes added donated_prio, length, total_donation, waiting_for

Functions:

1. donated_prio - A list for each thread which stores the donated and current priority. In our implementation, we append and delete the donated priority(if any) to this list after the thread is blocked/finished execution.
2. length - It is the length of the donated_prio list. On appending, it increases by one and reduces by one on decrementing.
3. total_donation - This is the number of total donated_priority locks i.e. it
4. waiting_for - A lock for which the thread is in the waiting list. This helps to signify which lock is going to be acquired when the thread comes out of the waiting state.

In the synch.h file in struct lock 1 new attribute is also added which tells us that is this lock donated to the current holder or not.

>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation.  (Alternately, submit a
>> .png file.)
The data structure used in a simple array whose length is changed based on the requirement of the thread. The nested donation diagram is listed below. File



1. Suppose we have 3 threads: A-31 prio, B-30(blocked) prio, C-29 prio.
2. Thus, B and C have acquired 2 different locks, say lock1 and lock2. Now A has the highest priority so it will run first and it is trying to acquire lock1 but it is currently in hold by thread B. So, it will donate its priority, so A(blocked), B(31 donated), C-29. Now, B wants to

acquire lock2 but it is currently in hold by C so, B will also donate its priority so A(blocked), B(blocked) and C(31 donated).

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
Wherever ready_list is updated, we change some functionality like when we are inserting a thread pointer in ready_list or waiting_list we will order it on the basis of its current priority. If you see the code then you will notice that in thread_yield, sema down, etc. functions, we are adding elements in order based on their priority so the first element will have the highest priority.

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

1. Suppose we have 3 threads: A-31 prio, B-30(blocked) prio, C-29 prio.
2. Thus, B and C have acquired 2 different locks, say lock1 and lock2. Now A has the highest priority so it will run first and it is trying to acquire lock1 but it is currently in hold by thread B. So, it will donate its priority, so A(blocked), B(31 donated), C-29. Now, B wants to acquire lock2 but it is currently in hold by C so, B will also donate its priority so A(blocked), B(blocked) and C(31 donated).
3. In the code we have used four variables which will keep track of all data we need. So, when any thread is acquiring a lock, we need to know which thread has acquired this lock and which lock is the thread waiting for so that we can donate priority sequentially. The waiting_for variable will keep track of which lock the thread is waiting for.
4. The variable "donated_prio" is for multiple donations of priority. When a thread is donated by many threads we want to keep track of the original priority and all the donated priorities so that when we release all the lock we can set the thread's priority to original priority.

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

1. We have an attribute is_donated by which we can know that the current lock is donated to the current thread. If it is donated then we have to delete its corresponding donated priority. This

priority is known by choosing the first element of lock's corresponding semaphore waiting list. Because the waiting list is sorted, the first thread will have the highest priority, and this priority would be donated once to the current thread. Because when higher priority threads go to the waiting list we are donating its priority.

---- 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?

1. Suppose thread A has been given priority by thread B and now thread A is changing its priority to the same as thread B so, which should run first. Two cases are possible: thread B first releases the lock and then B completes first and another case is thread A releases a lock and A completes first but in both of these answers are valid because after setting up A's priority it doesn't matter which thread runs first.
2. When thread sets its priority to a new value it may happen that it is in the waiting list of the lock or using some semaphores and that list is not sorted when we call set_priority function. But here set_priority does not mean that we are setting its original priority to its new value. This function will change its priority to new value when all the donation priorities will be deleted so there is no need to change all the lists or sort the lists. We have used thread_yield here which will run the highest priority thread after set_priority function.
3. Global variable changes are inserting elements in the list in order of priority but when we add elements, the interrupts are off. Thus, no need to use locks there.
4. Another change in shared variables is semaphore waiting list but there is also no need to use lock because semaphore functions are atomic and they are turning off interrupts.

---- RATIONALE ----

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

1. We thought that we could define a global list and keep track which thread donated its priority to

which thread. And push when priority is donated and pop when we release a lock. This type of structure is easy to code but it has high time complexity and also there are a lot of synchronization problems in this approach. Whereas in this approach it is simply handled using interrupts.

```
                        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.

---- 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
 4
 8
12
16
20
24
28
```

```
32
36

>> 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?

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

>> 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?

                    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 was too hard and long. It would have been great if prior tutorial sessions were taken introducing us to the PintOS. If we would have been taught about the folder structure and the use of various files in the PintOS directory then it would have been much easier for us.

>> Did you find that working on a particular part of the assignment gave
>> you greater insight into some aspect of OS design?
Working on the first two questions helped us understand the complexity that today's operating systems have. Even in Pintos which is a very old operating system, the scheduling algorithm and the alarm clock has been implemented very neatly. The concept is straightforward but applying it in real life is the challenge here and it helped us understand many things.

>> 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.

>> Do you have any suggestions for the TAs to more effectively assist
>> students, either for future quarters or the remaining projects?
A prior coding session in labs might help. Ungraded exercises may be held to instill the thinking process and the curiosity of the students.

>> Any other comments?
No.

>> Reference
https://github.com/abhivandan1999/pintos-priority-scheduling/blob/master/completion/pintos2/src/threads/thread.c
https://www.youtube.com/watch?v=myO2bs5LMak