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

---- GROUP ----

ARUN SURESH<arunsure@buffalo.edu>
HARIPRASTH PARTHASARATHY<hparthas@buffalo.edu>
PRANJAL JAIN <pjain5@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.

ALARM CLOCK

---- DATA STRUCTURES ----

>> A1:
struct thread
the following variables have been added

```
        int wake_time //to store the wake up time of the thread
        struct semaphore semaphore_timer //to have a semaphore for the timer
        struct list_elem timer_element   //list elem struct for timer_list
```

the following global variables are added

```
        struct list timer_list //store the list of threads waiting for timer
        struct semaphore timer_sema //semaphore used for locking and accessing
the list
```

---- ALGORITHMS ----

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

Inside the timer_sleep() method, we calculate the time at which the thread needs to be woken up,add value to the thread and push the thread to a wait list in sorted ordered using list_insert_ordered. Then we call sema_down()function to put block the thread.

We use a semaphore to lock the list timer_list and the semaphore value is checked in the timer interrupt handler.

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

The wait list where all the threads are put to sleep is sorted on based on the time to be woken up. The first entry in the list is the thread that needs to be woken up. This saves amount of time sent by timer interrupt handler searching for the thread to be woken up.

---- SYNCHRONIZATION ----

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

Since we are making use of semaphores, only one thread can access the list at a given point of time. This is why no race condition is encountered when more than one thread calls timer_sleep() function simultaneously.

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

Since semaphores cannot be used in an interrupt and considering the fact that interrupts are not preempted, we check the value of timer_list_semaphore and if it is zero we wait till the next tick.

---- RATIONALE ----

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

The design we have chosen has many advantages. The amount of time spent by timer interrupt handler is very minimal as the list is already sorted based on the time to be woken up. As we use semaphores, the timer_sleep () method is synchronized void of race conditions when multiple threads call timer_sleep (). We chose to have a separate list for waiting threads sorted based on the time to be woken up.

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

---- DATA STRUCTURES ---

>> B1:
```
struct donation_info
{
        struct list_elem elem;
        int priority_donated;
        struct thread *recipient;
        bool flag;
};
```
This struct is maintained for every donation done. It is placed in the thread making a donation. Since a thread at a time can wait for only on lock and make only one donation.
The elements include :
```
priority donated // which is the priority donated to the recipient thread.
thread_recipient // pointer to the recipient thread
flag            // which stored if this donation is currently used.
```

the structure thread is altered by adding the following variables
list donation_list; //which stores the list of all donations received by the thread in sorted order
donation_info waiting_for; //if the thread has donated to another thread its stored here
int original_priority; // stores the value of the original priority of the thread

struct thread *lookup_high_priority_thread(struct list *thread_list);
-The above function looks up on the list of threads that are ready to run and chooses the thread with the highest priority so it can compare it with the other threads and yield the appropriate thread.
struct thread *get_high_priority_thread(struct list *thread_list);
-The above function searches the highest priority thread in the waiters list and returns this thread by removing it from the list. This thread is then allocated to the semaphores and locks which then increment their value.

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

```
  Thread1       Thread2         Thread 3
+-------+       +--------+      +---------+
|       |       |       |       |       |
|       |       |       |       |       |
|       |       |       |       |       |
|Waiting|       |Waiting +-->   |Waiting+----> +----+
|for    +--->   |for    |       |for    |       +NUll+
|       |       |       |       |       |
|       |       |       |       |       |
+-------+       +--------+      +---------+
```

---- ALGORITHMS ----
L-Low priority thread
M-Medium priority thread
H-High Priority thread
1.|L|-requests lock
2.|L|-acquires lock
3.|M|-requests lock (currently used by L)
M is unable to acquire lock since L is holds it. Now M donates its Priority to L making L's lower priority equal to Medium's,
4.|M|->|L|->|M|(L gets M's priority and becomes M)(M makes entry in the donation list)
L continues using the lock as its priority is now M and H has not yet requested for the lock.
5. |H|-requests lock
H cannot acquire the lock since L holds it. To enable L to continue using the lock, H gives its priority to L thereby making L's priority from M->H.
6.|H|->|L|(M)->|H|(H makes entry in the donation list)
This is how a thread even with the minimum or lowest priority gets to complete execution with lock even if a higher priority thread requests the lock at the same time.

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
We have implemented a new function get_high_priority_thread() that returns the thread with the highest priority out of all the other threads that are blocked, by removing it from the list waiters. We check if a thread has the highest priority and remove it from the list while increasing the value of the semaphore since it got used by another thread.
In the lookup_high_priority_thread(), we are looking up for the thread with the highest priority.By using this function, we yield the thread with the highest priority in the ready_list.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
When lock acquire is called the holders priority is checked with the current threads priority. If it is lower we do priority donation and make an entry in donation list of the recipient thread and **waiting for** variable in the donor thread. The donation info added to the list of recipient is in the donor's structure.This is done to prevent memory allocation considering that a thread can wait for only one lock.

Nesting is checked by checking the waiting for variable in the recipient thread, and passing on the priority down- which is done by updating the donation info of the lower threads

>> 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 a lock, we check in the list of threads waiting for this lock and yield the lock to the thread which has the highest priority among the list of waiting threads.
We remove the donation done by the donor thread and update the priority of the recipient thread with the next highest in the donation list.If there are no waiters. We replace it with the original 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?
Yes we lock the ready list while setting priority which prevents access from the other threads.
---- RATIONALE ----

>> B7: Why did you choose this design? In what ways is it superior to
>> another design you considered?
We considered storing all the information in the lock itself. But when multiple locks come into picture.It is impossible to synchronize between multiple locks thus we decided to store all the donation info as a list in the recipient thread.

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.

int nice;
This variable is part of the thread structure which will hold the nice value of the corresponding thread.

int recent_cpu;
This variable is part of the thread structure which will hold the nice value of the corresponding thread.

static int load_average;
This is a global variable which will hold the current load average of the 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 ticks	recent CPU A	recent CPU B	recent CPU C	Priority A	Priority B	Priority 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?

Yes.When 2 threads have same priority there is no defined rule as to which thread should be scheduled. So here, we have decided to schedule the thread with least recent_cpu value in such cases.

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

Currently we are doing all the calculations for multi level feedback queue (recent cpu, priority, load average) inside the timer interrupt handler. This does affect the performance but to very small scale.

---- 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 advantage of this design is when there are multiple threads with the same priority, it is always ensured that the thread with minimum recent cpu value is scheduled. The ensures fairness among all threads. The disadvantage of this design is the use of list data structure. We feel when the number of threads increases, it might be more efficient to use other data structures like heap to maintain the ready list based on priority. If given more time, we would like to implement it.

>> 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 have used macros defined in thread.c file. We decided to follow the description as given in pintos manual to handle the fixed point calculations.We decided to use macros instead of functions since macros are much faster than functions and moreover all these calculations are done inside timer interrupt where all the operations must be time efficient.

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?

The way list is implemented inside pintos is very different and it would me very helpful if its implementation is in the book

We were able to debug using GDB-Merged along with VS code and we found it very helpful. So please share this configuration change to add support for GDB debugging.It would be very helpful for the future students taking up this project.

1.install visual studio code
2. Install native debug extension in visual studio code - and restart the IDE
3.select debug->edit_configuration and replace the current configuration with this

```
"configurations": [
{
"type": "gdb",
"request": "attach",
"name": "Attach to gdbserver",
"executable": "${workspaceRoot}/src/threads/build/kernel.o",
"target": "localhost:1234",
"remote": true,
"cwd": "${workspaceRoot}"
}]
```

run test in the terminal with (pintos --gdb -- run alarm-multiple)
*test name should be changed as per your requirements
then press F5 on VS code and debugger is connected

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