```
                +--------------------+
                |        CS 140      |
                | PROJECT 1: THREADS |
                |   DESIGN DOCUMENT  |
                +--------------------+
```

---- GROUP 11 ----

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

Sagnik Roy - sagnikr38@gmail.com Roll No: 18CS10063
Debajyoti Kar - debajyoti.apeejay@gmail.com Roll No: 18CS10011

---- PRELIMINARIES ----

>> If you have any preliminary comments on your submission, notes for the
>> TAs, or extra credit, please give them here.
In this assignment we are not expected to pass all the 27 tests given in the tests directory. We need to only pass 17. The code to run only those 17 files are also given in the form of **run_test.sh** that has to be run from /src/threads/build
>> 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.

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.

Sol. **Declarations in thread.h:**

1. Global variables:

- **typedef int fixed_t:** We typedef int because we have implemented fixed-point math.

2. In struct thread():

- **uint64_t sleep_time:** Store the time the thread will wake up
- **int base_priority:** Store base priority of the thread
- **struct list locks_holding:** Locks thread is holding (Necessary for priority-fifo test).
- **int nice:** Store the nice value of the thread
- **fixed_t recent_cpu:**Store the recent_cpu value of the thread

3. Function Declarations :

- **void thread_wake_up():** The core function of wake_up thread
- **void unblock_wake_up():** To unblock the wake_up thread

Declarations in thread.c:

1. #include "devices/timer.h": Necessary to invoke the function unblock_threads_in_timer

2. Global variables:

- **fixed_t load_avg:** Stores the moving average of the number of threads to run. It estimates the average number of threads ready to run over the past minute.
- **struct thread *wake_up:** The wake_up thread to unblock sleeping threads

3. Macros for fixed-point arithmetic: Since pintOS does not support floating point arithmetic and it must be done using integers.

4. Functions:

- **bool thread_cmp_priority ():** To compare priority of 2 threads
- **void thread_update_priority ():** To update priority of thread
- **mlfqs_ functions :** Functions to handle the framework for mlfqs scheduling

Declarations in timer.c:

1. #include "threads/timer.h": So that the non static functions in thread.c may be called from timer.c

2. Global Variables:

- **static int64_t next_wakeup:** Store the time when wake_up thread is to be unblocked.

3. Functions:

- **unblock_thread_in_timer ():** To unblock those sleeping threads whose wake_up time has arrived

---- 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:

Sol.

timer ticks	recent_cpu			priority			thread to run
	A	B	C	A	B	C	
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?
If the running thread has the same priority as some thread in the ready queue, the scheduler will take the one in the ready queue and then in the next time slice will do the same as round-robin. Yes this match with the scheduler as the highest priority one still the one in the running state but this is done to deliver a more responsive system. Also, recent_cpu here is ambiguous here. When we calculate recent_cpu, we did not consider the time that CPU spends on the calculations every 4 ticks, like load_avg, recent_cpu for all threads, priority for all threads in all_list, restore the ready_list. When the CPU does these calculations, the current thread needs to yield, and not run. Thus, every 4 ticks, the real ticks that are added to recent_cpu (recent_cpu is added 4 every tick) is not really 4 ticks -- less than 4 ticks. But we could not figure out how much time it spends. What we did was add 4 ticks to recent_cpu every 4 ticks.

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?
Sol. We disabled interrupts in our code in the thread_yield(), thread_set_priority() and the init_thread() functions. Interrupt disabling in the thread_set_priority() function is unlikely to affect performance by a big margin as it simply involves setting the priority of the current thread to new_priority which takes O(1) time. But performance may be hampered due to interrupt disabling in the thread_yield() and init_thread() functions if the number of threads is very large. Both these functions involve inserting threads into sorted lists (the ready_list and all_list respectively) which takes O(log N) time, where N is the number of threads. Thus other processes may have to wait for considerable time if N is very large.

---- 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?
Part A: Part A is a modification of the implementation we did in last assignment. The part of unblocking the sleeping threads which was earlier done in timer_interrupt, is now in **unblock_threads_in_timer**. The global variable next_wakeup defines the time when the wake_up thread is to be woken up next. The wake_up thread executes the function **thread_wake_up**, that calls the function **unblock_threads_in_timer** in an infinite loop as defined in the problem statement.
Part B: The function **list_insert_ordered () in /lib/kernel/list.h** is able to maintain the ready list in sorted order of priorities. So we define the comparison function thread_cmp_priority. Finally, replace the **list_push_back** function in init_thread, thread_unblock and thread_yield with **list_insert_ordered**, and make similar modifications. This does the task of basic priority scheduling.
It can be found that the experiment requires the implementation of 64-level scheduling queues, and each queue has its own priority, ranging from PRI_MIN to PRI_MAX. Subsequently, the operating system starts scheduling threads from the high-priority queue, and dynamically updates thread priorities over time.

Modify the **timer_interrupt** function first. We know from reference materials that each time the clock is interrupted, the recent_cpu of the running thread will increase by 1, and the system load_avg and recent_cpu of all threads will be updated every TIMER_FREQ ticks, and the thread priority will be updated every 4 ticks. Next, implement the **recent_cpu self-increment function in thread.c** (and declare it in thread.h). Realize the function of **updating system load_avg** and **recent_cpu** of all threads.Finally, we add members **int nice** and **fixed_t recent_cpu** to the thread structure and initialize it in init_thread. Declare global variable **load_avg in thread.c** and initialize it in thread_start and implement the 4 unfinished functions in **get_nice**, **set_nice**, **get_load_avg** and **get_recent_cpu**.

The ready_list was implemented using a single list rather than 64 lists. Although this change was permitted by the course instructor, given more time we would have liked to implement it in the original way intended, although it does not affect performance much but still it would be the ideal MLFQS.

>> 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?
Sol. Fixed-point math provides better performance than floating-point arithmetic because it requires less memory and less processor time as arithmetic and logical operations are performed using integer arithmetic. It is also unambiguous; each numerical value has exactly one representation unlike floating point numbers. Also floating-point operations require a dynamic shift of the exponent used to scale the base number, at run time. But with fixed-point numbers, the exponent value is defined by the fixed-point data type itself and does not require to be computed at run-time.

In our assignment, we needed to deal with several non-integral values like recent_cpu and load_avg. The calculations involving these numbers need to be performed as fast as possible so as to minimise the overhead. Therefore we implemented fixed-point math and defined several macros to perform arithmetic operations on the fixed-point numbers.