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

---- GROUP : 10 ----

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

Sumit Kumar Yadav sumitkumar18cs.iitkgp@gmail.com
Avijit Mandal avijitmandal2001@gmail.com

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.

1. Addition of variable int load_avg in threads.h to hold load average in fixed point notation
2. Addition of variable int nice in threads.h to store nice value
3. Addition of variable int recent_cpu to store recent cpu in fixed point notation
4. Addition of function update_priority(struct thread *) to update priority for a given thread inmlfqs
5. Addition of comparator function cmp_priority(list_elem*,list_elem*) to return true if threadbelonging to 1st parameter has higher priority than second
6. Addition of following functions in threads/fixed_point.h:
    a. int tofxpt(int n): Convert int to fxpt_t
    b. int tointfloor(int x): Convert fixed point value to int and truncate thedecimal part
    c. int tointround(int x): Convert fixed point value to int followed byrounding off
    d. int addin(int x,int y): Add 2 fixed point values
    e. int subin(in t x,int y): Subtract one fixed point value from another
    f. int addfx(int x,int n): Add an int to a fixed point value
    g. int subfx(int x,int n): Subtract an int from a fixed point value
    h. int mulin(int x,int y): Multiply 2 fixed point values
    i. int mulfx(int x,int n): Multiply a fixed point value by an integer
    j. int divin(int x,int y): Divide one fixed point value by another
    k. int divfx(int x,int n): Divide a fixed point value by an int

---- ALGORITHMS ----

- Modified thread_tick() to calculate load_avg and recent cpu. Priority of each thread isalso updated
- Modified thread_create() to pre-empt execution of current thread if new thread hashigher priority
- Modified thread_yield() to sort ready_list according to priority
- Modified thread_get_recent_cpu() to return recent_cpu of currently running thread
- Modified thread_get_nice() to return nice value of currently running thread

● Modified thread_get_load_avg() to return load_avg of system
● Modified set_nice(int) to update the nice value of current thread. The function callsupdate_priority( ) in current thread and subsequently calls thread_yield( )
● Modified thread_set_priority() to modify priority of currently running thread. Itsubsequently calls thread_yield()

>> 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?

The ambiguity was choosing which thread of the same priority to run first. We used the round robin rule to determine the next running thread.
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, resort the ready_list. When CPU does these calculations, the current thread needs to yield, and not running. Thus, every 4 ticks, the real ticks that is added to recent_cpu (recent_cpu is added 1 every ticks) is not really 4 ticks –less than 4 ticks. But we could not figure out how much time it spends. What we did was adding 4 ticks to recent_cpu every 4 ticks.
Our implementation of BSD scheduler is the same as this. We just count the ticks since system boots, and does all the above calculations every 4 ticks.
The cmp_priority( ) function ensures this as thread A is put before thread B on the list only ifthread A’s priority is greater than that of thread B. Otherwise it comes after thread B in thesorted list.

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

If the CPU spends too much time on calculations for recent_cpu, load_avg and priority, then it takes away most of the time that a thread before enforced preemption. Then this thread can not get enough running time as expected and it will run longer. This will cause itself got blamed for occupying more CPU time, and raise its load_avg, recent_cpu, and thereefore lower its priority. This may disturb the scheduling decisions making, thus, if the cost of scheduling inside the interrupt context goes up, it will lower 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?

Our design did not apply the 64 queues. We used only one queue – the ready_list that Pintos originally have. But as the same as what we did for the task 2, we keep the ready_list as an priority oriented descending order at the every beginning -- that is, whenever we insert a thread into the ready_list, we insert it in order. The time complexity is O(n). Every fourth tick, it is required to calculate priority for all the threads in the all_list. After this, we need to sort the ready_list, which will take O(n lgn) time. Since we need to do this job every 4 ticks, it will make a thread's running ticks shorter than it is expected. If n becomes larger, thread switching may happen quite often. If we use 64 queues for the ready threads, we can put the 64 queues in an array with index equaling to its priority value. When the thread is first inserted, it only need to index the queue by this thread's priority. This will take only O(1) time. After priority calculation for all threads every fourth tick, it takes O(n) time to re-insert the ready threads. But our implementation is better than this situation -- ready_list is not ordered. Like pintos originally did, for every new unblocked thread, just push back to the ready_list. When it need to find next thread to run, it has to reorder the ready_list. Sorting takes O(n lgn) time. It needs to repeat this sorting whenever we need to call thread_next_to_run, and after calculating all threads' priorities every fourth tick.
Thus, we'd like to implement 64 queues instead of 1 queue for ready threads.
One disadvantage is that The order of individual threads of same priority in the ready queue depends upon howthe sort function is implemented and may not always guarantee the ordering betweendemoted threads and other threads in the level to which the threads have beendemoted. In our case, a stable sorting algorithm ensures that threads with the samepriority are scheduled using Round Robin scheduling

>> 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 created a set of functions to manipulate fixed-point numbers as it:

- Reduces the complexity of the arithmetic expressions involved in thread.c and improvesreadability of the code

- Helps to distinguish between integer and fixed-point parameters which in turn minimizesthe chances of an error