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

---- GROUP ----

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

Ashish Ramagoni <ashishgo@buffalo.edu>
Wendan Zhao <wendanzh@gmail.com>
Yashwanth Chennu <ychennu.buffalo.edu>

>> Contributions of group members.
Test Case:

Phase 1: priority-fifo, priority-preempt, priority-change -> code implemetations by Wendan and Yashwanth, ideas also by Ashish
Phase 2: alarm-single, alarm-multiple, alarm-simultaneous, alarm-priority,
alarm-zero, alarm-negative -> Code implementation by Yashwanth and Ashish, ideas and debugging by Wendan, Ashish
Phase 3: priority-donate-one, priority-donate-multiple, priority-donate-multiple2, priority-donate-nest, priority-donate-sema,
priority-donate-lower, priority-fifo, priority-preempt, priority-sema, priority-condvar, priority-donate-chain
-> code implementation by Wendan and ashish, ideas and debugging by Ashish and Yashwanth

mlfqs-load-1, mlfqs-load-60, mlfqs-load-avg, mlfqs-recent-1, mlfqs-fair-2,
mlfqs-fair-20, mlfqs-nice-2, mlfqs-nice-10,mlfqs-block -> code implementation by Ashish and Yashwanth, ideas and debugging by Wedan and Yashwanth

---- 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.

ANSWER:

In thread.h file

Modified the struct thread.

```
struct thread{
  /* ------ Omit Members Defined in Pintos Source Code ------ */
    /* ----------------- New Added Members ------------------ */
  bool waiting_status; /*Indicates whether the thread is waiting for a
timer event. Used for sleep implementation.*/
  struct semaphore s; /* Semaphore used for blocking and unblocking
threads during sleep*/
  int64_t time_to_wake; /*Stores the tick count at which the thread
should wake up from sleep*/
}
```

purpose:
Below are the modification made in the timer.c file using the above declared:
- The function timer_sleep() has been modified to use the newly added waiting_status, s, and
time_to_wake members of struct thread to implement sleeping functionality.
It sets the thread to wait until the specified tick count is reached.

Below are the modification made in the thread.c file using the above declared:
-The function is modified to check all threads' time_to_wake against the current tick
count and wake them up by calling sema_up when their wait time is over.

-sema_init(&t->s, 0); in init_thread function in thread.c:
Initializes the semaphore in each thread struct, which is essential

for the sleeping
functionality to work correctly.

---- ALGORITHMS ----

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

Answer:

When a thread invokes `timer_sleep(ticks)`, it transitions to a waiting state and
calculates its wake-up tick (`time_to_wake`) by adding the requested `ticks` to
the current system tick count (`timer_ticks()`). The thread then self-blocks by executing
`sema_down(&t->s)` on its own semaphore `s`.

The system's timer interrupt handler, `timer_interrupt`, is invoked at each timer
tick, incrementing the global `ticks` counter. It then calls `thread_tick()`, which
iterates through all threads. For each thread, it checks if the `waiting_status`
is true and if the current tick count meets or exceeds the thread's `time_to_wake`. If so,
the thread is unblocked with `sema_up(&t->s)`, allowing it to be considered for scheduling.

The blocked thread, having called `sema_down(&t->s)` in `timer_sleep()`, remains
in a non-running state until the timer interrupt handler's actions result in
`sema_up(&t->s)` being called when the system tick count reaches the thread's
`time_to_wake`. This mechanism allows the thread to resume execution after sleeping
for the specified duration, effectively using `timer_sleep()` to pause execution
without engaging in CPU-intensive busy waiting.

This sleep mechanism efficiently conserves CPU resources by ensuring that threads awaiting
their wake-up time do not occupy the CPU. This approach integrates closely with scheduling and synchronization
mechanisms, leveraging the semaphore s and the waiting_status and time_to_wake variables in t
the struct thread to manage sleep durations and wake-up scheduling.

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

Answer:

Below are the steps taken to minimize the amount of the time in the timer interrupt handler:

-Efficient Tick Count Increment: The timer interrupt handler (timer_interrupt) directly increments
the global ticks variable. This operation is very quick and ensures that the handler does not
spend unnecessary time performing basic bookkeeping.
-Minimal Processing in thread_tick(): The thread_tick() function, called by the timer interrupt
handler, focuses on essential operations. It updates system-wide statistics like idle_ticks,
kernel_ticks, and user_ticks depending on the current thread's state. While it does iterate over all
threads to check their waiting_status and time_to_wake, it aims to do so efficiently by quickly determining
which threads need to be woken up and acting on them without complex logic.
-Use of Semaphores for Blocking and Unblocking: The waking up of threads is managed through semaphores (sema_up(&t->s)).
This approach is efficient because semaphore operations are designed to be low overhead.
Instead of performing complex checks or computations, waking a thread involves merely updating its semaphore state, which is a relatively quick operation.
-Conditional Processing: The actual work to wake threads is conditional; that is, it only proceeds to call sema_up(&t->s) if a thread's wake-up condition is met (total_ticks >= t1->time_to_wake). This ensures that during each tick, only necessary actions are taken without needless processing for threads that do not need to be woken yet.

---- SYNCHRONIZATION ----

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

Answer:

Below are how the race conditions are avoided when multiple threads call timer_sleep() simultaneously:

1. Disabling Interrupts: By disabling interrupts at the start of

timer_sleep() and re-enabling them after modifying shared structures or variables, race conditions from concurrent access are prevented.
2. Use of Synchronization Primitives: The semaphore (struct semaphore s;) in the struct thread plays a pivotal role. When a thread calls sema_down(&t->s) in timer_sleep(),
it's effectively put to sleep in a controlled manner that synchronizes access to the CPU.
The sema_down() operation is atomic, ensuring that even if multiple threads invoke timer_sleep() simultaneously, their transition to a blocked state happens without interference.
3. Atomic Access to Global Variables: Access to shared resources, like the global tick count ticks in timer_interrupt(), is managed in a way that prevents race conditions.
For example, ticks is accessed and incremented atomically within the timer interrupt handler, ensuring consistent state updates even when multiple timer_sleep() calls are pending or being processed.
4. Thread-Specific Wake-up Time: Each thread calculates its own time_to_wake based on the current ticks value plus the desired sleep duration.
This calculation and the subsequent checking against the global ticks in thread_tick() do not modify shared state in a way that could cause race conditions.
Each thread's sleep duration is managed independently, based on its own time_to_wake attribute.

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

Answer:

Below are the ways how a race conditions are avoided when a timer interrupt occurs during a call to timer_sleep().
-Firstly, the operation of setting a thread to sleep and calculating its time_to_wake in timer_sleep() involves critical variables like the global ticks counter and thread-specific attributes (waiting_status, time_to_wake).
To ensure atomicity and consistency, such operations are safeguarded by disabling interrupts at critical moments within timer_sleep().
This approach prevents the timer interrupt from preempting the process of setting up sleep conditions, thus avoiding race conditions between setting a thread's sleep parameters and the interrupt-driven tick count increment.

-Furthermore, the semaphore mechanism plays a crucial role in synchronization.
The sema_down(&t->s) call within timer_sleep() not only blocks the calling thread until the sleep duration has elapsed but does so in a

manner that's atomic and safe from interruption.
This ensures that the process of putting a thread to sleep is not disrupted by concurrent timer interrupts, maintaining the integrity of the sleep logic.

-During the timer interrupt, the timer_interrupt handler increments the global ticks variable and calls thread_tick(), which checks if any sleeping thread needs to be woken up.
The design ensures that the increment of the tick count and the subsequent checking of threads' time_to_wake are performed in a controlled, atomic fashion, part of the interrupt handler's execution flow.
This atomicity is crucial for preventing inconsistencies in the system's time tracking and the scheduling of thread wake-ups.

By combining interrupt disabling with the use of synchronization primitives like semaphores, the system effectively ensures that operations within timer_sleep() and actions taken during timer interrupts do not interfere with each other.
This design carefully balances the need for responsiveness to timer interrupts with the requirement to maintain a consistent and race-free environment for managing thread sleep states.

---- RATIONALE ----

>> A6: Why did you choose this design? In what ways is it superior to
>> another design you considered?
Answer: The chosen design, emphasizing interrupt management and semaphore synchronization, excels in efficiency, simplicity, and atomicity, making it superior to alternatives like busy-waiting or complex locking mechanisms.
It optimizes CPU utilization by allowing threads to sleep without consuming resources, a contrast to busy-waiting approaches that degrade system performance.
By disabling interrupts during critical updates and using semaphores for thread blocking and waking, it ensures atomic operations and prevents race conditions, avoiding the complexity and potential deadlock issues of finer-grained locks.
This approach strikes a balance between system responsiveness and the scalability needed to handle numerous concurrent threads, showcasing its effectiveness in a multitasking environment.

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.
Modify struct thread, struct lock
struct thread
  {
    /* ------ Omit Members Defined in Pintos Source Code ------ */
    /* ----------------- New Added Members ----------------- */
    int base_priority;                  /* Remember base priority. */

    struct list locks_holding;          /* locks currently held by
thread. */
    struct thread *lock_holder;          /* lock holder that thread is
waiting on. */
  };

struct lock
  {
    /* ------ Omit Members Defined in Pintos Source  Code ------ */
    /* ----------------- New Added Members ----------------- */
    struct list_elem elem;      /* Link locks that are currently held
by thread. */
  };


>> B2: Explain the data structure used to track priority donation.
>> Use ASCII art to diagram a nested donation.  (Alternately, submit a
>> .png file.)
In struct thread, base_priority is used to keep the thread's original
priority,
while priority is taken as effective priority which is considered when
scheduling
this thread and keeps updated whenever there is higher donated
priority. Only when
the thread releases all the locks, its thread will be recovered to the
original
priority. As struct lock has a member holder which is used to keep the
thread holding
this lock, after the lock being acquired by one thread, other threads
that try to get
this lock will donate their priority if higher than the lock holder
when acquiring that
lock, this secinario can be understood as direct priority donation.
Another secinario is
that whenever lock holder thread gets new higher donated priority, if
it is also waiting
on another lock which is held by a thread, it should immediately pass
or propagate new
higher donated priority to that thread, which is kept by member
lock_holder, similaly,
if that thread is also waiting on some lock holder thread, it needs to
pass the new donated
```

```
priority further.

Take priority-donate-nest as an example.

1, Main thread creates two locks a and b, and acquires a.
+---------------------+    +-----------------+
|     lock a          |    |    lock b       |
+---------------------+    +-----------------+
| holder: Main thread |    | holder: NULL    |
| waiters: empty      |    | waiters: empty  |
+---------------------+    +-----------------+

+-----------------------+
|       Main thread     |
+-----------------------+
|  priority: 31         |
|  base_priority: 31    |
|  locks_holding: lock a|
|  lock_holder: NULL    |
+-----------------------+

2, Main thread creates thread medium with priority 32 which
   is higher than main, so thread medium is scheduled to run
   and it acquires lock b, it gets blocked when trying to acquire
   lock a, it donates priority to main thread.
+---------------------+    +-----------------+
|     lock a          |    |    lock b       |
+---------------------+    +-----------------+
| holder: Main thread |    | holder: medium  |
| waiters: medium     |    | waiters: empty  |
+---------------------+    +-----------------+

+-----------------------+   +-------------------------+
|       Main thread     |   |         medium          |
+-----------------------+   +-------------------------+
|  priority: 32         |   |  priority: 32           |
|  base_priority: 31    |   |  base_priority: 32      |
|  locks_holding: lock a|   |  locks_holding: lock b  |
|  lock_holder: NULL    |   |  lock_holder: Main      |
+-----------------------+   +-------------------------+
3, Main thread gets run again and creates thread high with priority
33 which is higher than main, so thread high is scheduled to run
and it tries to acquire lock b but blocked, so it donates priority to
thread medium, also, priority gets passed to main thread.
+---------------------+    +-----------------+
|     lock a          |    |    lock b       |
+---------------------+    +-----------------+
| holder: Main thread |    | holder: medium  |
| waiters: medium     |    | waiters: high   |
+---------------------+    +-----------------+
```

```
+-----------------------+   +------------------------+
+-----------------------+
|       Main thread     |   |         medium         |    |
high          |
+-----------------------+   +------------------------+
+-----------------------+
|  priority: 33         |   |  priority: 33          |    |  priority:
33         |
|  base_priority: 31    |   |  base_priority: 32     |    |
base_priority: 33    |
|  locks_holding: lock a|   |  locks_holding: lock b |    |
locks_holding: NULL  |
|  lock_holder: NULL    |   |  lock_holder: Main     |    |
lock_holder: medium  |
+-----------------------+   +------------------------+
+-----------------------+
```

4, Main thread gets run again after getting donated priority 33,
it releases lock a and unblock thread medium, and its priority returns to
original priority, thread medium run again and gets lock a.

```
+---------------------+    +-----------------+
|     lock a          |    |    lock b       |
+---------------------+    +-----------------+
| holder: medium      |    | holder: medium |
| waiters: empty      |    | waiters: high  |
+---------------------+    +-----------------+

+-----------------------+   +------------------------+
+-----------------------+
|       Main thread     |   |         medium         |    |
high          |
+-----------------------+   +------------------------+
+-----------------------+
|  priority: 31         |   |  priority: 33          |    |  priority:
33         |
|  base_priority: 31    |   |  base_priority: 32     |    |
base_priority: 33    |
|  locks_holding: empty |   |  locks_holding:lock a&b|    |
locks_holding: NULL  |
|  lock_holder: NULL    |   |  lock_holder: NULL     |    |
lock_holder: medium  |
+-----------------------+   +------------------------+
+-----------------------+
```

5, Thread medium release lock a and b, and recover its priority to
32, thus giving control to thread high.

```
+---------------------+    +-----------------+
|     lock a          |    |    lock b       |
+---------------------+    +-----------------+
| holder: NULL        |    | holder: NULL    |
```

```
| waiters: empty      |    | waiters: empty |
+---------------------+    +----------------+

+-----------------------+   +-----------------------+
+-----------------------+
|       Main thread     |   |         medium        |    |
high          |
+-----------------------+   +-----------------------+
+-----------------------+
|  priority: 31         |   |  priority: 32         |    |  priority:
33         |
|  base_priority: 31    |   |  base_priority: 32    |    |
base_priority: 33    |
|  locks_holding: empty |   |  locks_holding: empty  |    |
locks_holding: NULL  |
|  lock_holder: NULL    |   |  lock_holder: NULL     |    |
lock_holder: NULL    |
+-----------------------+   +-----------------------+
+-----------------------+
```

6, Thread high is scheduled to run and gets lock b.

```
+---------------------+    +----------------+
|     lock a          |    |    lock b      |
+---------------------+    +----------------+
| holder: NULL        |    | holder: high   |
| waiters: empty      |    | waiters: empty |
+---------------------+    +----------------+

+-----------------------+   +-----------------------+
+-----------------------+
|       Main thread     |   |         medium        |    |
high          |
+-----------------------+   +-----------------------+
+-----------------------+
|  priority: 31         |   |  priority: 32         |    |  priority:
33         |
|  base_priority: 31    |   |  base_priority: 32    |    |
base_priority: 33    |
|  locks_holding: empty |   |  locks_holding: empty  |    |
locks_holding: b     |
|  lock_holder: NULL    |   |  lock_holder: NULL     |    |
lock_holder: NULL    |
+-----------------------+   +-----------------------+
+-----------------------+
```

7, Thread high release lock b and finishes, next thread scheduled to run would be medium
as it has higher priority than main thread, after thread medium finishes, main thread gets
to run and finish.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
Threads that are waiting for a lock or semaphore are kept track by struct semaphore member
waiters, it is a list of waiting threads, when trying to wake up a thread with highest
priority, we use list function list_max (struct list *list, list_less_func *less, void *aux),
which will return the element in the list with largest value of priority according to our myless
function, then we use list_entry(LIST_ELEM, STRUCT, MEMBER) to find the corresponding thread.
Threads that are waiting for condition variable are kept track by list of struct semaphore_elem,
to find the highest priority thread, we traverse the list and first use list_entry(LIST_ELEM, STRUCT, MEMBER)
to get struct semaphore_elem, then extract member semaphore within struct semaphore_elem,
then use list_entry(LIST_ELEM, STRUCT, MEMBER) again to convert waiters in that semaphore to be threads,
after getting struct thread we can extract the priority of that thread, so we can keep comparing priority of
all the threads waiting for the condition variable, always keeping the highest one in a variable called
waiter_max and sema_max, using sema_max we can wake up the thread with highest priority.

>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
when a call to lock_acquire(), a pointer to struct lock is passed into that function, the first thing
this function will do is to check whether the lock has holder, if it has holder, then current running
thread keeps the lock holder in its variable called lock_holder which is used to remember who it is waiting
on and also used for further donation, then it will compare its priority with lock holder's priority and
pass its priority to lock holder if its priority is higher than lock holder's. After the lock holder gets new
higher donated priority, it will check whether it is waiting on some lock holder, if it is true then passing
further this new higher donated priority to the lock holder, here a while loop is used to handle nested donation
by keep checking if lock holder is waiting on any lock holder and propagate the higher donated priority.

>> 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, the lock holder i.e. current running thread will first remove that
lock from its locks_holding which is a list of locks, and check whether it is still holding any locks, if it
no longer holds any lock, it will recover to its original priority, if it is still holding some locks then it
will traverse the list of locks, from all the waiters of those locks to get highest priority and update its priority
with that one. Next, it release the lock by assigning the value of this lock holder to be NULL, then it calls sema_up()
to wake up threads with highest priority, using list_max (struct list *list, list_less_func *less, void *aux) and
list_entry(LIST_ELEM, STRUCT, MEMBER) to find that thread and unblock it. After that, it will call intr_context(),
if is not in interrupt context, it will call thread_yield() to schedule next thread to be run. If the higher-priority
thread waiting for the lock has higher priority than the current running thread's new updated priority, it will get
CPU, otherwise it will still be in the ready list.

---- 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?
thread_set_priority(int new_priority) is used to set the current thread's priority to new_priority, and it will invoke
thread_yield() as current thread's priority changes. There are two secinarios current thread may change priority, one is
itself wants to change priority, another is through priority donation. If thread_set_priority() is used for both secinarios,
it may cause some issues. One case is that after the thread releases all the locks and lower priority to its original one, it
will yield to other higher priority threads before it can unblock the thread waiting for that lock. Another case is that when
it is holding some locks and wants to lower its priority, if this lowering takes effect, then it may never get chance to run
and cannot unblock other higher priority threads waiting for that lock. To avoid this issue, In our implementation,
thread_set_priority() is only used by thread itself when it voluntarily wants to change its base_priority.

---- RATIONALE ----

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

our design is logically clear and readable. In phase 1, we use list_insert_ordered() to keep the list of threads in decreasing order whenever there is a new thread added to the ready list, and use list_pop_front to get thread with highest priority in that list, by doing in this way it leads to modifying less code of pintos. However, in phase3 of priority donation, the priority of threads may change all the time, so the ready list may become stale, therefore the first one in the ready list may not be the highest priority thread. Instead we use list_max(), this function will always ensue that we get the highest priority thread, it completely solver the stale issue.

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.

```
struct thread{
  ---old vairable--
  int nice;
  fixed_point_t recent_cpu;
}
```

I have added 2 vairables in thread struct nice, recent_cpu. Assigned default values 0 for both.
nice -> Represents the niceness value of the thread for scheduling purposes.
recent_cpu->Represents the recent CPU usage of the thread, used for advance priority calculation.

I have also added a global varibale load_avg, initializes with 0.
fixed_point_t load_avg; in thread.c -> Represents the system-wide average number of threads ready to run, used in priority calculation.

---- 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?
We have found many times threads having same priority, and this causes uncertanity.
We have used FIFO to over come this, as every thread gets a chance.
Explination->
at ticks 0 - A,B,C(ready in priority order) - A go to running
at ticks 4 - A,B,C(ready) - A(running)
after this A's priority is 61 it enters ready list after threads which have same priority of before thread with less priority.
at ticks 8 - B,A,C(ready) - B(running)

Other uncertainities are we are not considering the time for calculations of recent_cpu, load_avg,priority.

>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?
Inside interrupt -> for every four ticks we have to calculate priority for threads and for every tick we
have to increment the recent_cpu of running thread, so this operations take most of time that to run the process of the thread
and we have to calulate load_average, recent_cpu for every second.
This takes most of the time of running thread and this
makes the thread to run more no of times and every time the thread priority decrease as the recent_cpu increase.
Outsite Interrupt -> we have to switch the thread for every 4 ticks if there is a high priority thread. Even switching takes time.
So if cost of inside and outsise interrupt should be minimal to have a good 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?
With our design we can eleminate starvation of threads with less
priority by decreasing priority of threads
with high priority as time increases.
We can also improve it a tittle by changing few approaches as we are
using insertion sort to insert a thread,
we can use binary search to insert a new thread in list, which reduce
the time for insertion.
We are using "MULTI LEVEL FEEDBACK QUEUEs", so that each queue has
threads of same priority, but we have used
only 1 list. If each there were queues for each thread seperately we
add a new thread to of its priority queue
rather than traversing the entire ready_list to find its position. By
such kind of implementations we might need
more variables and pointers(for each queue) but the performance
increases.

>> 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 used the fixed-point math, as it was simple and every operations
that we need for calculations we
already present. we used thwm to calculate at 3 places(recent_cpu,
load_avg, priority) so, we didnt make it complex
by creating abstraction layer.
Also, by creating abstraction layer over sixed-point might be easy for
undersating but increases the performance time.


               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?

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

>> Any other comments?