---- GROUP ----

Ahmed Khalil ElZeiny (20010087)

Ahmed Samir Elsayed (20010107)

Moahmed Wael Fathy   (20011752)

Youssef Hossam AboElwafa (20012263)

Youssef Mohamed ElMedany (20012293)

ALARM CLOCK

=============

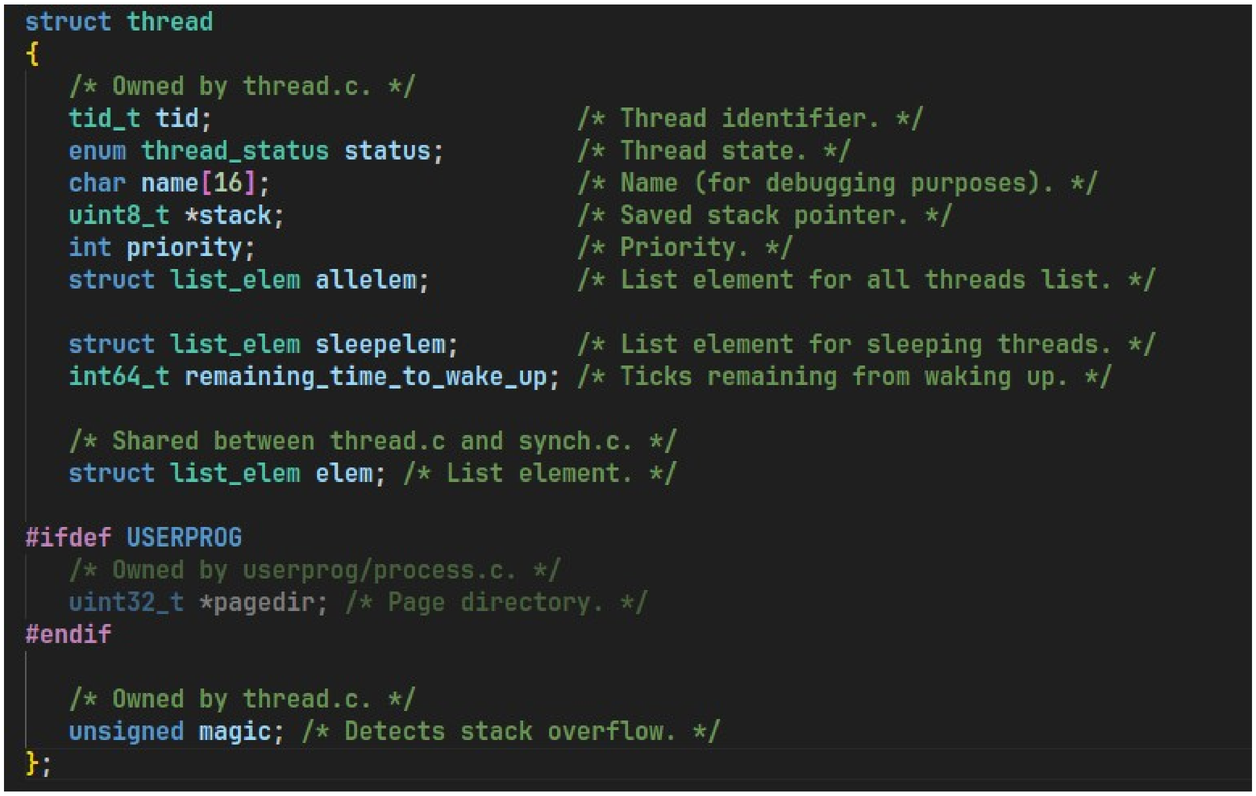
---- DATA STRUCTURES ----

In (thread.h):

Add two new attributes to represent the remaining ticks before waking up the thread.

1. **sleepelem** is the list element for sleeping threads.

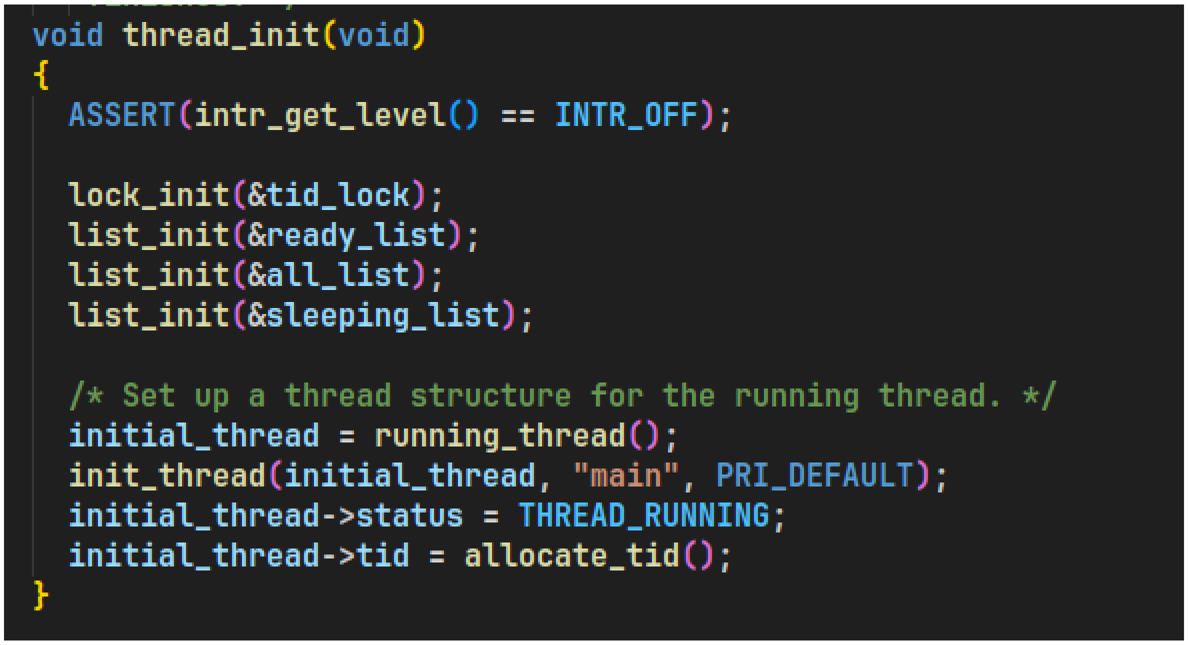
2. **remaining_time_to_wake_up** is ticks remaining from waking up, with an initial value 0.



```
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. */

   struct list_elem sleepelem;       /* List element for sleeping threads. */
   int64_t remaining_time_to_wake_up; /* Ticks remaining from waking up. */

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

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

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

---- ALGORITHMS ----

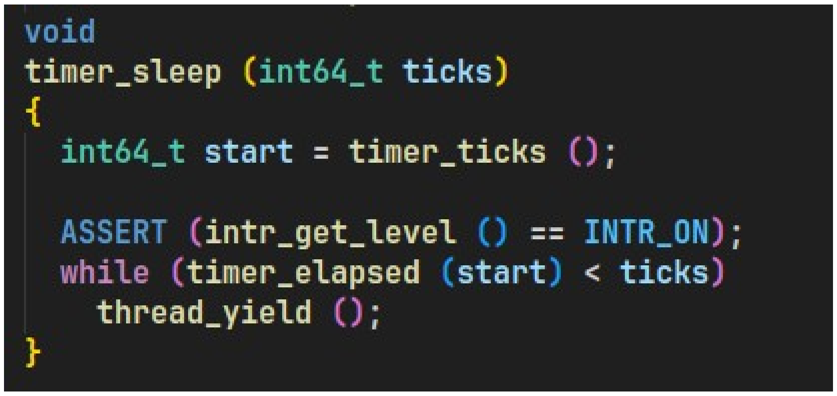
Another list is created to hold the sleeping threads and is initialized with thread init function as well as ready_list and all_list



```
void thread_init(void)
{
  ASSERT(intr_get_level() == INTR_OFF);

  lock_init(&tid_lock);
  list_init(&ready_list);
  list_init(&all_list);
  list_init(&sleeping_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread();
  init_thread(initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid();
}
```

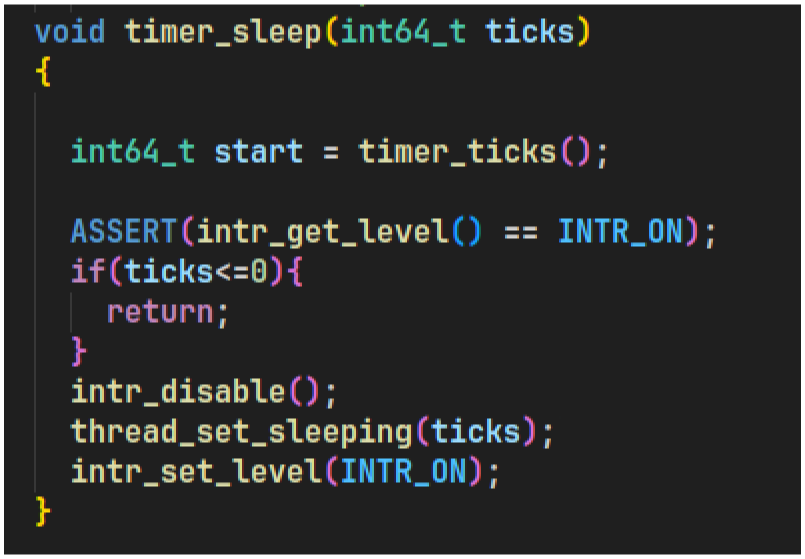
When calling `timer_sleep(int64_t ticks)`function it had a busy waiting so it is removed and replaced with another function `thread_set_sleeping(ticks)` when a thread is put to sleep for a given number of ticks

before (busy waiting):



```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

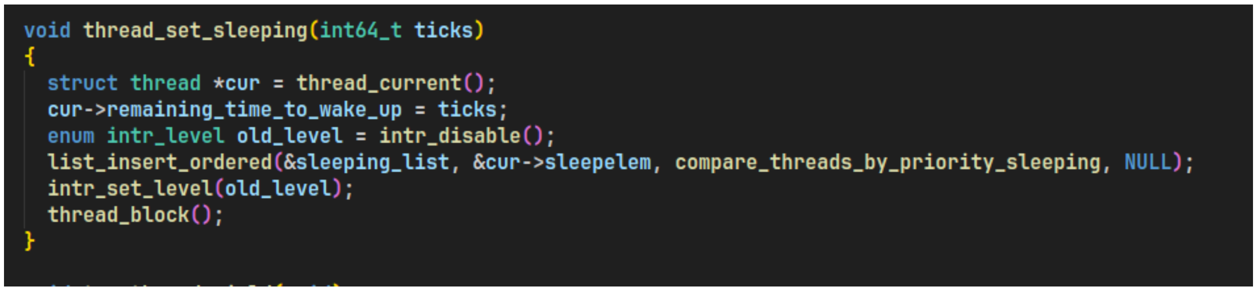
after:



```c
void timer_sleep(int64_t ticks)
{

  int64_t start = timer_ticks();

  ASSERT(intr_get_level() == INTR_ON);
  if(ticks<=0){
    return;
  }
  intr_disable();
  thread_set_sleeping(ticks);
  intr_set_level(INTR_ON);
}
```

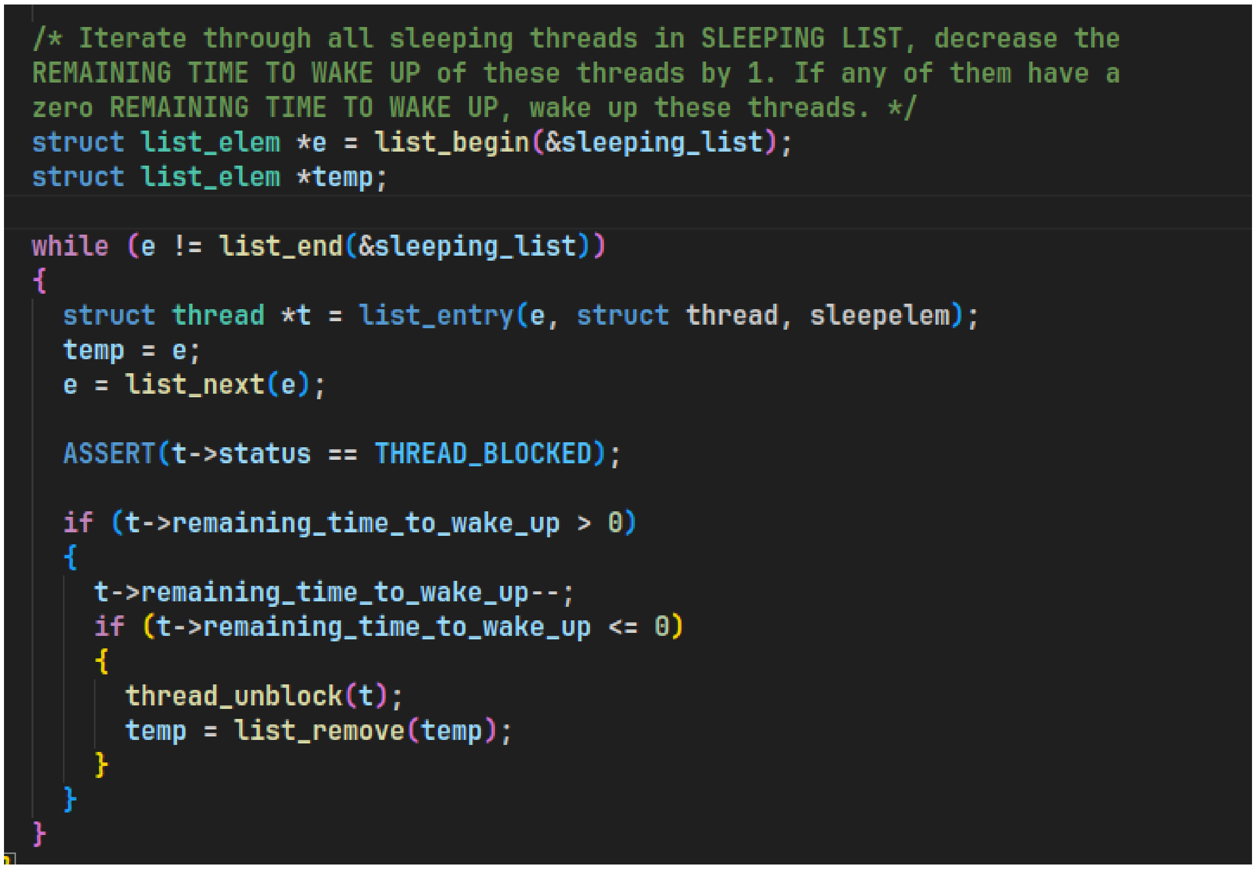
when calling thread_set_sleeping(int64_t ticks) the thread is inserted in descending order w.r.t its priority into the sleeping_list instead of push_back so if there is 2 threads with (remaining_time_to_wake_up) =0

→ then the thread with the highest priority is unblocked first



```c
void thread_set_sleeping(int64_t ticks)
{
  struct thread *cur = thread_current();
  cur->remaining_time_to_wake_up = ticks;
  enum intr_level old_level = intr_disable();
  list_insert_ordered(&sleeping_list, &cur->sleepelem, compare_threads_by_priority_sleeping, NULL);
  intr_set_level(old_level);
  thread_block();
}
```

This while loop in inserted in (thread_tick) that is called in the (timer_interrupt) handler



```c
/* Iterate through all sleeping threads in SLEEPING LIST, decrease the
REMAINING TIME TO WAKE UP of these threads by 1. If any of them have a
zero REMAINING TIME TO WAKE UP, wake up these threads. */
struct list_elem *e = list_begin(&sleeping_list);
struct list_elem *temp;

while (e != list_end(&sleeping_list))
{
  struct thread *t = list_entry(e, struct thread, sleepelem);
  temp = e;
  e = list_next(e);

  ASSERT(t->status == THREAD_BLOCKED);

  if (t->remaining_time_to_wake_up > 0)
  {
    t->remaining_time_to_wake_up--;
    if (t->remaining_time_to_wake_up <= 0)
    {
      thread_unblock(t);
      temp = list_remove(temp);
    }
  }
}
```

It iterate through all sleeping threads in sleeping_list , decrease the

REMAINING TIME TO WAKE UP of these threads by 1 with every tick.

If any of them have a zero REMAINING TIME TO WAKE UP, wake up this thread.

---- SYNCHRONIZATION ----

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

With the interrupt disabled during the thread operation, this function is almost "atomic".

```
intr_disable();

...

intr_set_level(INTR_ON);
```

---- RATIONALE ----

I have thought about just making all_list to replace sleeping_list, because all alive threads are naturally in all_list, then if this design is taken, there will be no needs to insert put sleeping threads in another place.

But the iteration over all_list to find the sleeping thread and check the (remaining_time_to_wake_up) will be time consuming.

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.

In struct thread:

```
int real_priority; // stores the real (original) priority of the thread

struct list locks_held; // the list occupies the locks held by the thread

struct lock *locked_by; // it points to the lock which the thread is currently waiting for
```

(real priority) member stores the original priority of the thread when priority donation takes place. (locks_held) list stores locks elements held by the thread. (Locked_by) member indicates the lock that the thread is waiting for (initially NULL).

in struct lock

```
struct list_elem elem; // pointer for a lock in a list
int max_priority; // the max priority of all threads waiting for the lock
```

(elem) member is a list_element data type that is used to indicate the presence of the lock in a list.

(max_priority) member stores the maximum priority among the threads that are locked by that lock.

In struct semaphore_elem:

```
int priority; // the max priority of the waiting threads on the semaphore
```

priority element stores the maximum priority among the threads waiting on the semaphore.

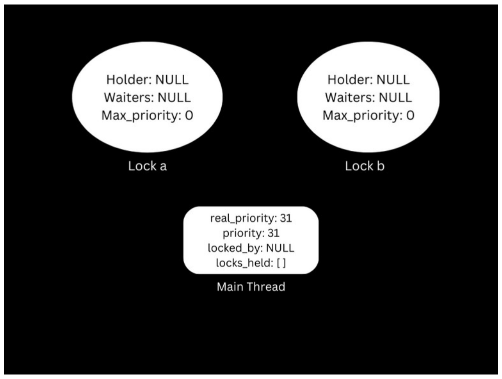
>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation. (Alternately, submit a

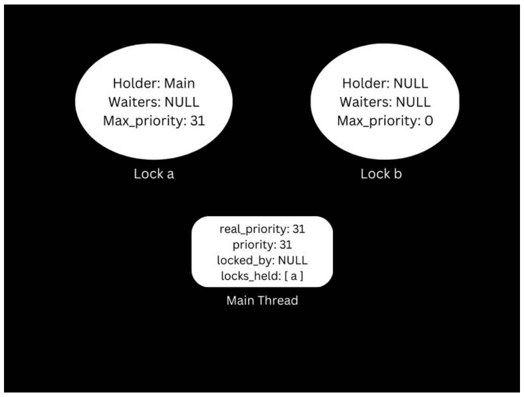
>> .png file.)

let we simulate the test priority-donate-nest to explain the nested donation:

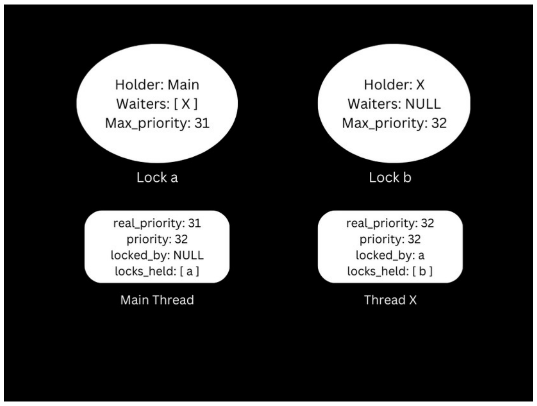
I. main thread starts and initialize two locks a and b:



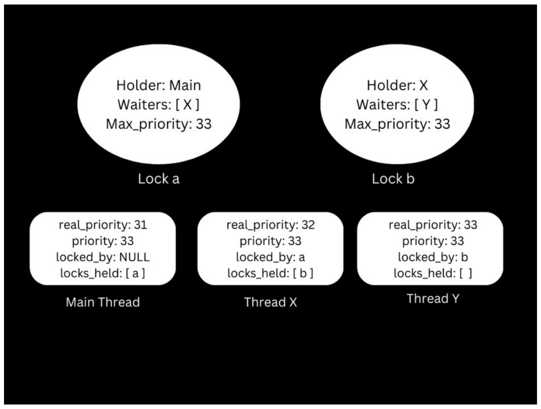
II. main thread then acquires lock a:



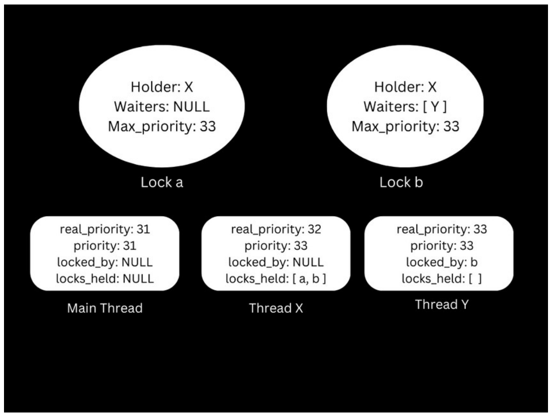
III. a new thread with medium priority (x) is created and acquires lock b and then tries to acquire lock a so it will be listed to wait for lock a and it will donate its priority to the main thread.



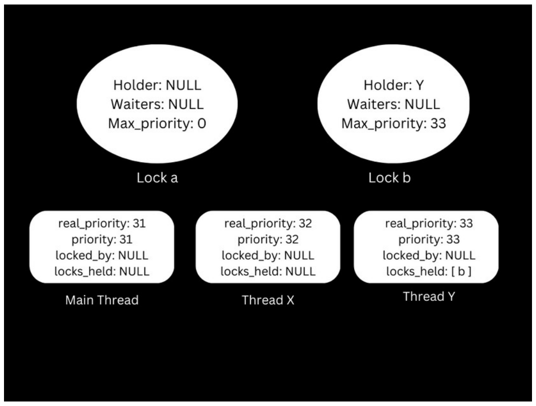
IV. Another new thread Y will be created with a higher priority and it will try to acquire lock b which is held by thread X so it will be blocked waiting for the lock in addition to recursively donating its priority till it reaches the main thread.



V. Returning to main thread until it releases lock a. when it releases a, it will return to its real priority and thread X will be pushed to the ready list then thread_yield will be called to check for threads with higher priority to run. So thread X will hold the CPU.



VI. by returning to X, X will then release lock a then thread_yield will be called then X will return to execute then it releases lock b and then it will return to its real priority. After that thread_yield will be called which will give the CPU to thread Y to start execution.



VII. After thread Y runs to completion, Thread X will be the next one to run and after it runs to completion, the main thread will be the next one to run and after that main thread will run to completion.

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

On calling the signaling function (sema_up) that is used by semaphores, locks (as it contains a binary semaphore) and condition variables (concerned with a lock), the sema_up function extracts the maximum thread (with respect to priorities) from the list of waiters and then awakes it up by removing it from the waiting list then calling the (thread_unblock) function to be added to the ready list.

>> B4: Describe the sequence of events when a call to lock_acquire()

>> causes a priority donation. How is nested donation handled?

For example on a sequence that causes a priority donation let we assume we have two locks a and b. let we assume the main thread having priority 31 and it acquire lock a. let we assume a new thread (x) is created with priority 32 and it successfully acquires lock b then try to acquire lock a then it will be blocked waiting for lock a. before being blocked the main thread priority should have been updated to 32 as it holds lock a which thread (x) with priority 32 is waiting for. Assume that now another new thread (y) is created with priority 33 and then it tries to acquire lock b. as lock b is held by thread(x) then thread(y) will be blocked waiting for the release of lock b but before being blocked, priority of thread(x) will be updated to 33 as thread(y) is waiting for lock b which thread (x) is holding. Recursively, the main thread's priority will be also updated to 33 as at that moment thread(x) which is waiting for lock a held by the main thread has a priority of 33. after the main thread releases lock a, it will return to its original priority which is 31 and then thread(x) can acquire the lock. At that moment thread(x) will continue having priority of 33 as it holds lock b which thread(y) is waiting for. When thread(x) releases lock b it will return to its original priority which is 32 and now thread(y) can acquire lock b. and then keep running (as it will have the highest priority) until it runs to completion then thread(x) will run (as it have priority of 32) until it runs to completion and finally the main thread will run as it has the least priority among the threads.

How lock acquire is implemented to handle priority donation?

If a thread tries to acquire a given lock, it will check first if the lock is held by any other thread if no, then it will acquire the lock. If yes, then a temporary lock will be created and initialized by that lock. A recursive check will take place to check if the temporary lock holder is locked by another lock or not if yes, temporary lock will be updated to that lock and the temporary holder will be the holder of that temporary lock. This sequence will keep gong until the temporary holder will not be locked by any other lock. Priority donation to lock holders and locks will take place during this recursive call. When at last this call finishes, the thread that tried to acquire the lock at the beginning will wait on the lock that tried to acquire until it is released.

>> B5: Describe the sequence of events when lock_release() is called

>> on a lock that a higher-priority thread is waiting for.

As listed in the previous example in B4, when lock_release() is called, the thread that was holding the lock will return to its original priority(as a higher priority thread was waiting for that lock) and the highest_thread in the list of waiters on that lock (if there was more than one) will be awakened to acquire the lock and will hold the CPU as it will be at that moment the highest priority thread among the ready threads. When that higher priority thread runs to completion, the thread that was holding the lock at the beginning can take place to run.

---- 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?

A potential race in thread_set_priority occurs when the priority of the current thread is smaller than that of any threads in the ready_list, therefore thread_yield should take place to yield the CPU so that we always make sure that the highest priority thread is running . Our implementation avoided that by disabling the interrupts whenever we set the current thread’s priority or yielding the CPU. As these operations can be considered atomic so we faced no problems.

Yes we could use locks to overcome or avoid the race by having a lock for accessing and setting the current thread’s priority. Whenever trying to set the priority of the thread, thread should first acquire the lock to have access to set its priority and releases the lock after setting it so that it ensures that there is no race can take place while setting its priority.

---- RATIONALE ----

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

>> another design you considered?

The design I made to handle this part of the code was relatively simple to implement and in addition effective. To handle the priority donation, I added a list of locks to each thread in addition to a lock that is locked_by and waiting for (if exists). This approach removed the headache of searching in each lock individually to check if the current thread presents in its list or not. In addition It eases the way of accessing all the locks that a thread is holding. The efficiency of this design appeared clearly especially during the implementation of lock_acquire function. When a thread tries to acquire a lock it first checks if the lock is held by any thread or not. If not, then just acquire it but if not?

It will keep recursively update the priority of lock holders by the maximum priority of the lock (which is an attribute in the struct lock) until there exists a holder thread which is not locked by any other lock and hence the recursion stop. In that way the priority is transferred among the threads holding the locks (if nested) an the thread return to its real priority after releasing all the locks it holds.

Regarding the priority, our design always makes sure that on signaling either a semaphore, a lock or a condition variable associated with a lock, the highest priority thread waiting for any of the previous events is awakened as the waiters are compared each time signaling takes place to extract the thread of the highest priority.

ADVANCED SCHEDULER

==================

---- DATA STRUCTURES & FUNCTIONS ----

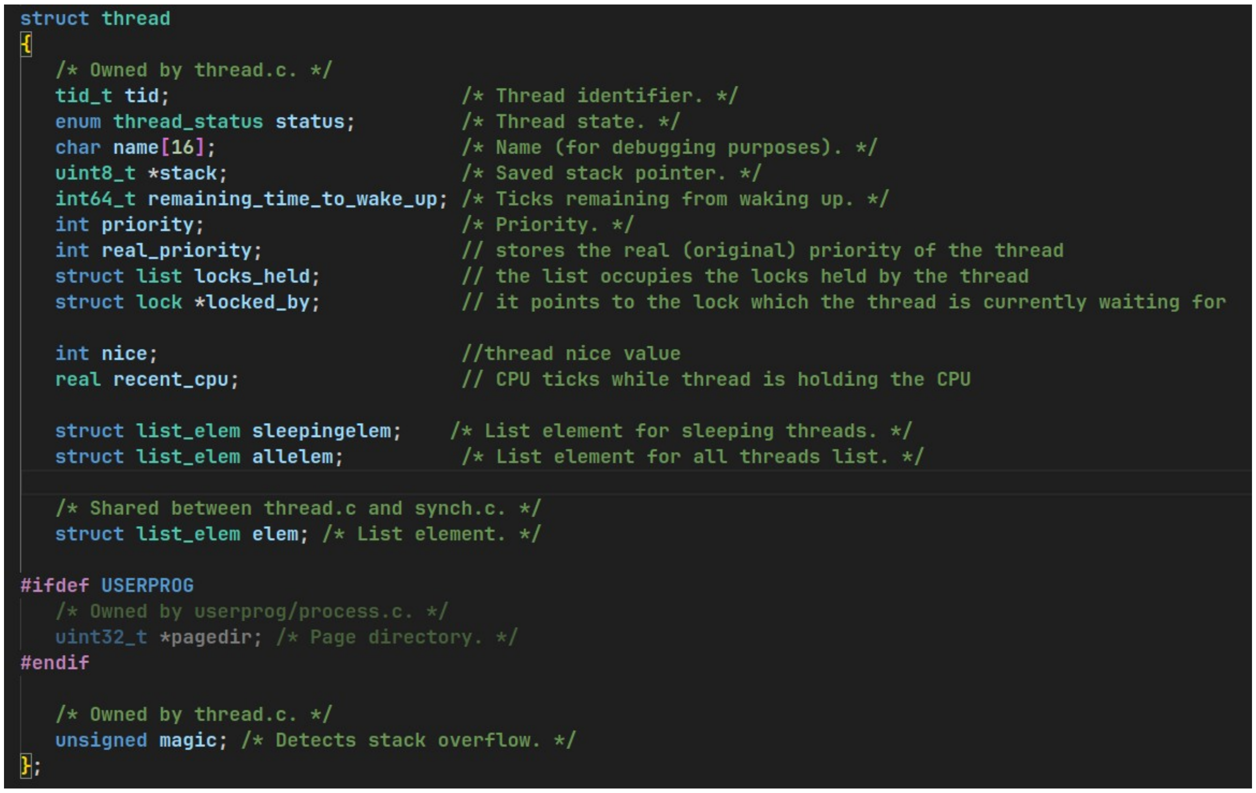
In (thread.h):

Add these attributes.

In the **thread struct**:

1. **int nice** to detect how nicer the thread is to calculate priority.

2. **real recent_cpu** save the time the thread was running on the cpu the last time.



```
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. */
    int64_t remaining_time_to_wake_up;  /* Ticks remaining from waking up. */
    int priority;                       /* Priority. */
    int real_priority;                  // stores the real (original) priority of the thread
    struct list locks_held;             // the list occupies the locks held by the thread
    struct lock *locked_by;             // it points to the lock which the thread is currently waiting for

    int nice;                           //thread nice value
    real recent_cpu;                    // CPU ticks while thread is holding the CPU

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

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

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

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

The rest are added as global variables and macros.

3.**NICE_MAX** ,**NICE_MIN**,**NICE_DEFAULT** macros to indicate the margin of nice value (-20 to 20).

4.Add **real load_avg** to track load average.

In (fixed_point.h):

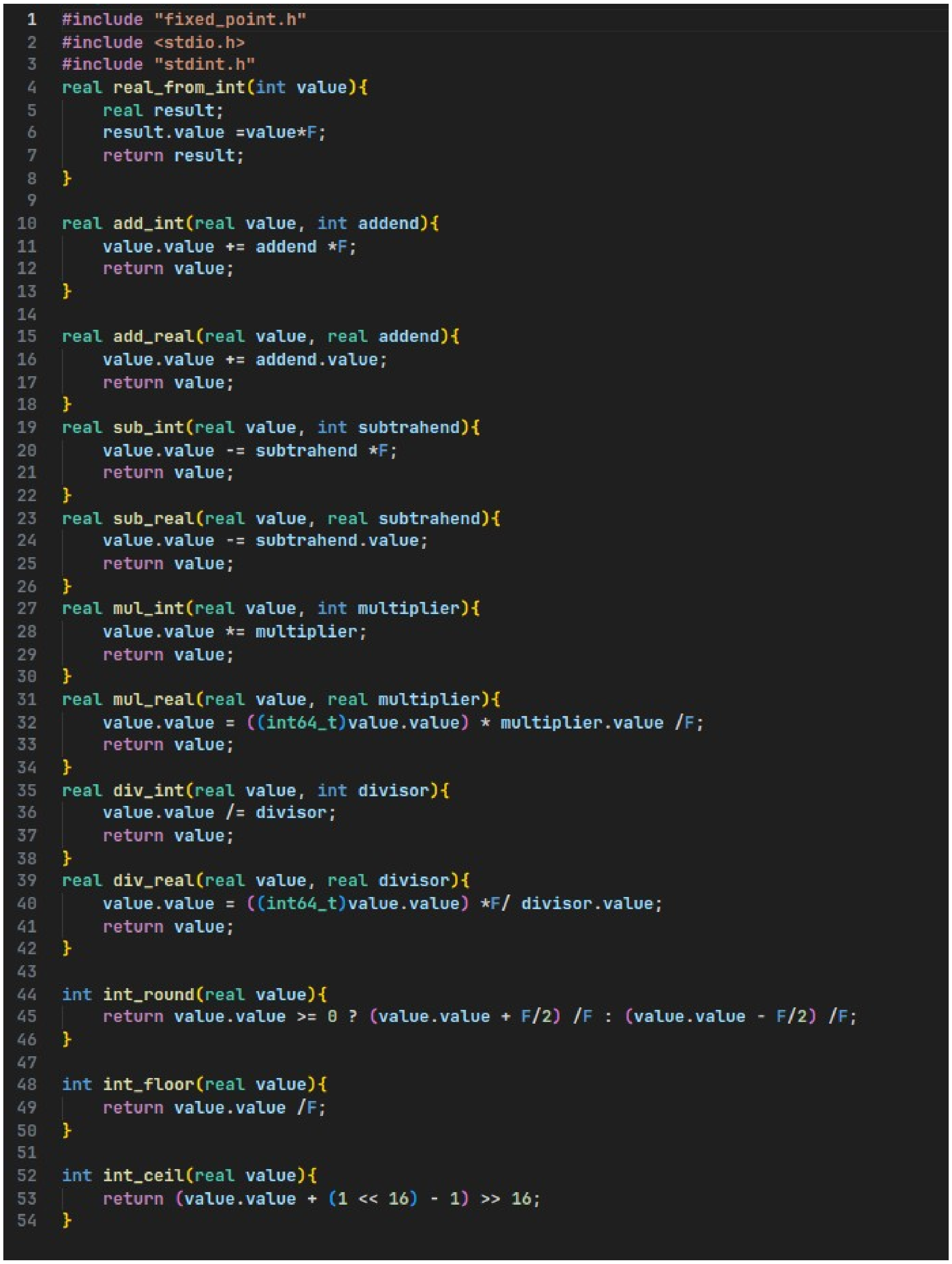
1.Added new **type_def real** ,which is a struct to be able to make real numbers calculations .

2.Added **macro F** which is equal 2^14 such that the float numbers has precision up to 14 bits.

3.**add**,**mult**,**div**,**sub** functions that do the real numbers operations ,with two versions on for operation on real and real ,and the second is for operation on real and int.

4.**real_from_int** ,**int_round** , **int_floor** functions used to alter from fixedpoint numbers to int

In



```
#include "fixed_point.h"
#include <stdio.h>
#include "stdint.h"
real real_from_int(int value){
    real result;
    result.value =value*F;
    return result;
}

real add_int(real value, int addend){
    value.value += addend *F;
    return value;
}

real add_real(real value, real addend){
    value.value += addend.value;
    return value;
}
real sub_int(real value, int subtrahend){
    value.value -= subtrahend *F;
    return value;
}
real sub_real(real value, real subtrahend){
    value.value -= subtrahend.value;
    return value;
}
real mul_int(real value, int multiplier){
    value.value *= multiplier;
    return value;
}
real mul_real(real value, real multiplier){
    value.value = ((int64_t)value.value) * multiplier.value /F;
    return value;
}
real div_int(real value, int divisor){
    value.value /= divisor;
    return value;
}
real div_real(real value, real divisor){
    value.value = ((int64_t)value.value) *F/ divisor.value;
    return value;
}

int int_round(real value){
    return value.value >= 0 ? (value.value + F/2) /F : (value.value - F/2) /F;
}

int int_floor(real value){
    return value.value /F;
}

int int_ceil(real value){
    return (value.value + (1 << 16) - 1) >> 16;
}
```

(thread.c):

1.Added <u>**recent_inc**</u> function that increases recent_cpu for running thread every tick.

2.added **recent_calc** function which Calculate new recent_cpu time for all threads, and updates associated thread fields using this equation:

recent_cpu = (2 * load_avg)/(2 * load_avg + 1) * recent_cpu + nice

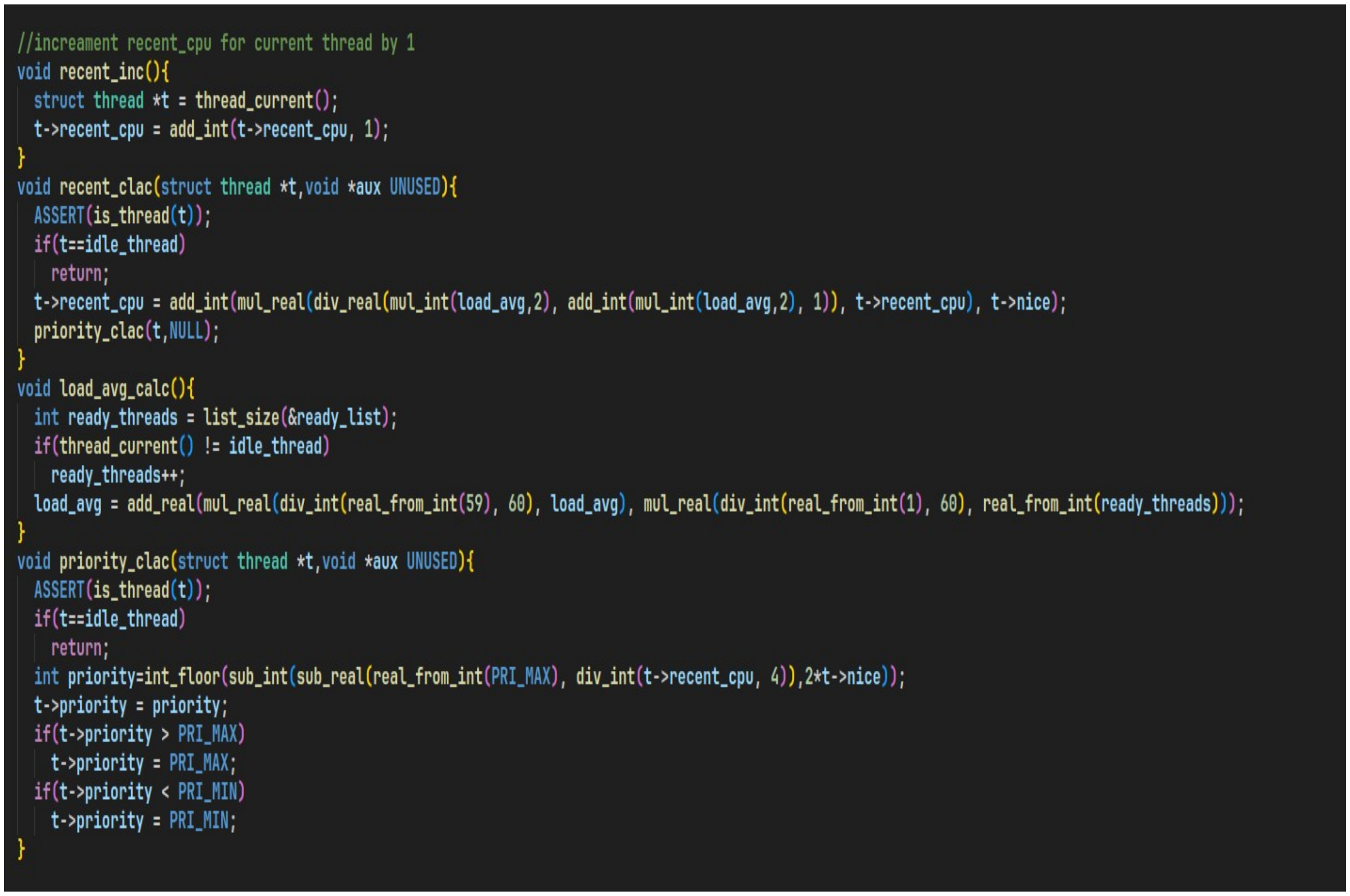
As spec says, may have to compute coefficient separetely to avoid overflow.

3.Added **priority_calc** function to calculate priority using nice value and recent cpu with this equation

Priority =PRI_MAX-(recent_cpu/4)-(nice*2)

4.added **load_avg_calc** function which Calculate new system-wide load average using this equation

load_avg = (59/60) * load_avg + (1/60) * ready_threads.

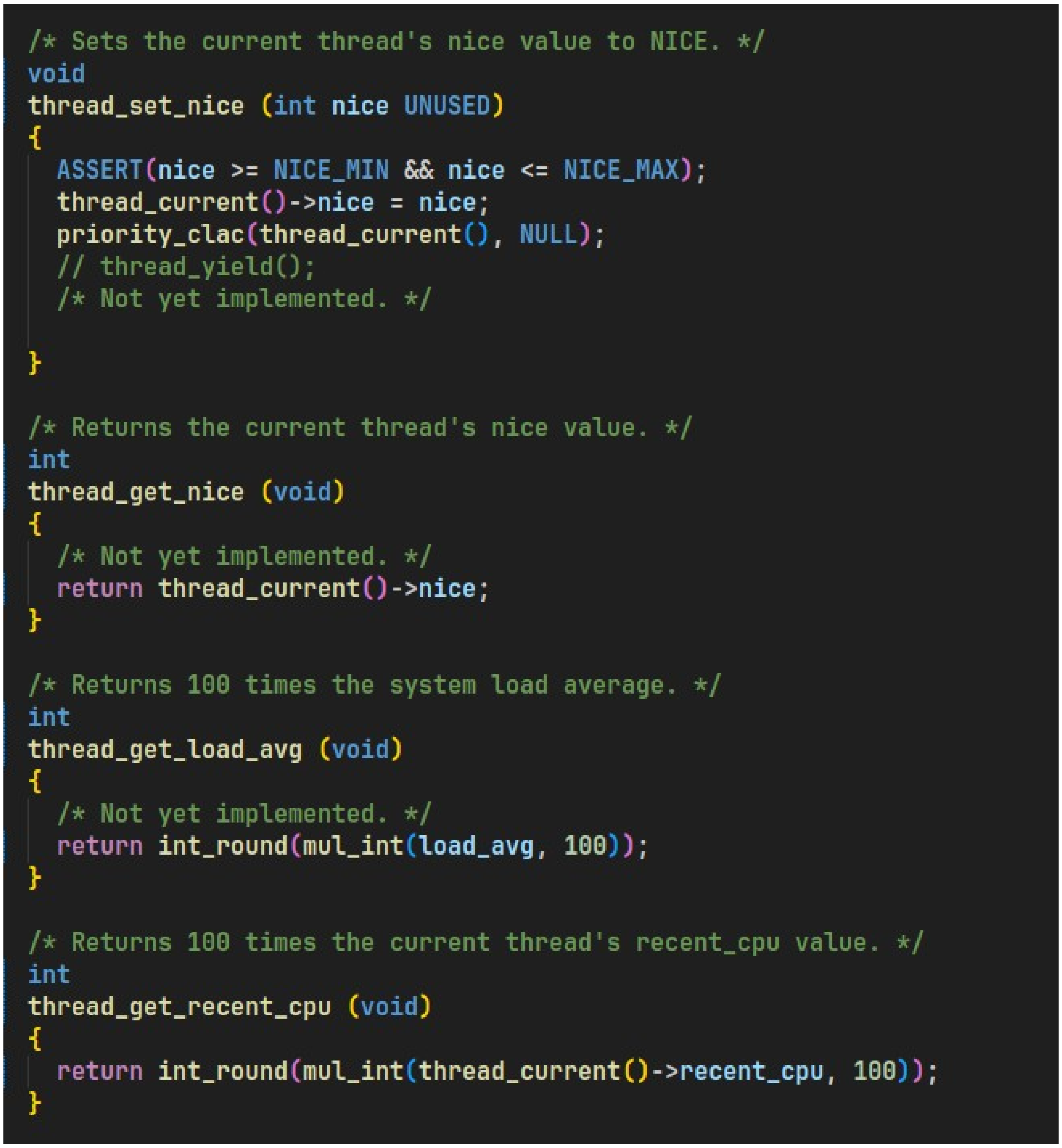


```c
//increament recent_cpu for current thread by 1
void recent_inc(){
  struct thread *t = thread_current();
  t->recent_cpu = add_int(t->recent_cpu, 1);
}
void recent_clac(struct thread *t,void *aux UNUSED){
  ASSERT(is_thread(t));
  if(t==idle_thread)
    return;
  t->recent_cpu = add_int(mul_real(div_real(mul_int(load_avg,2), add_int(mul_int(load_avg,2), 1)), t->recent_cpu), t->nice);
  priority_clac(t,NULL);
}
void load_avg_calc(){
  int ready_threads = list_size(&ready_list);
  if(thread_current() != idle_thread)
    ready_threads++;
  load_avg = add_real(mul_real(div_int(real_from_int(59), 60), load_avg), mul_real(div_int(real_from_int(1), 60), real_from_int(ready_threads)));
}
void priority_clac(struct thread *t,void *aux UNUSED){
  ASSERT(is_thread(t));
  if(t==idle_thread)
    return;
  int priority=int_floor(sub_int(sub_real(real_from_int(PRI_MAX), div_int(t->recent_cpu, 4)),2*t->nice));
  t->priority = priority;
  if(t->priority > PRI_MAX)
    t->priority = PRI_MAX;
  if(t->priority < PRI_MIN)
    t->priority = PRI_MIN;
}
```

5.modified t**hread_set_nice** functions checks the validity of the value nice and if it is valid sets new val ue of nice and updates priority in the current_thread.

6.modified **thread_get_load_avg** and **thread_get_recent_cpu** functions to return the value of load_avg and recent_cpu respectively after multiplication by 100 and rounding .

7. modified **thread_get_nice** function which returns the value of nice.

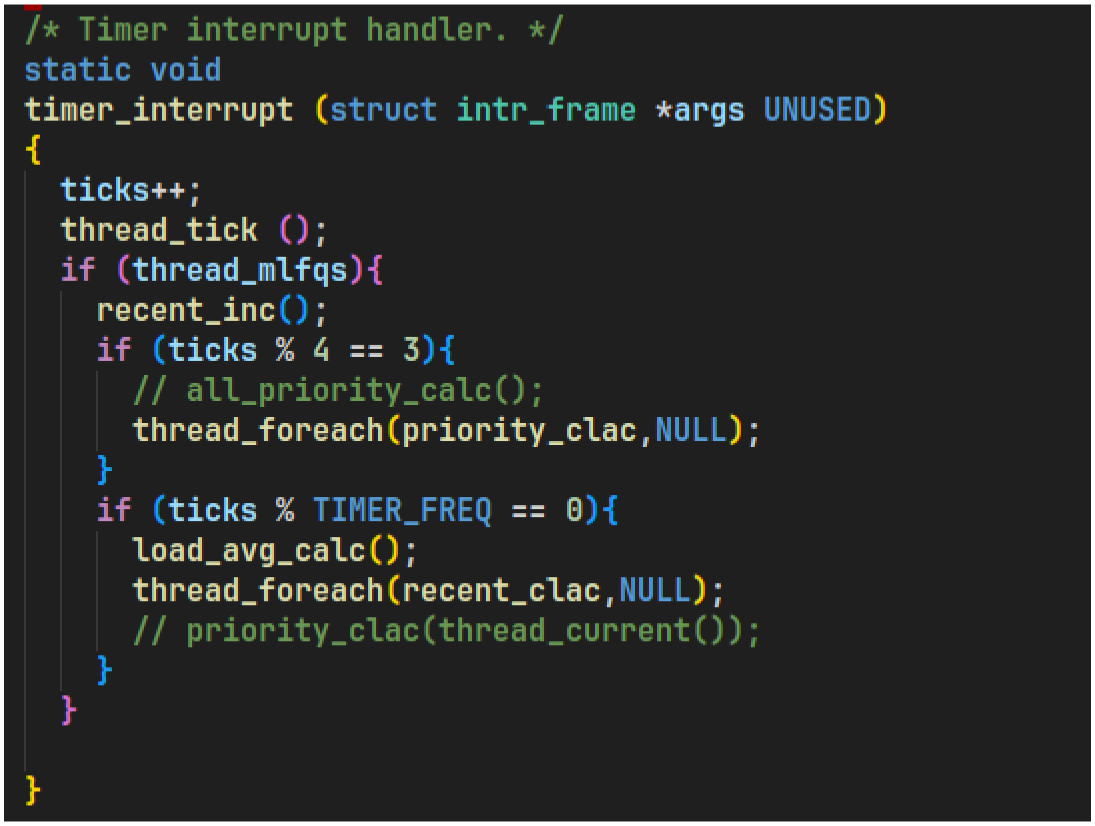


```c
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int nice UNUSED)
{
  ASSERT(nice >= NICE_MIN && nice <= NICE_MAX);
  thread_current()->nice = nice;
  priority_clac(thread_current(), NULL);
  // thread_yield();
  /* Not yet implemented. */

}

/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
  /* Not yet implemented. */
  return thread_current()->nice;
}

/* Returns 100 times the system load average. */
int
thread_get_load_avg (void)
{
  /* Not yet implemented. */
  return int_round(mul_int(load_avg, 100));
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu (void)
{
  return int_round(mul_int(thread_current()->recent_cpu, 100));
}
```

In (timer.c)

1.Modify **timer_interrupt** function we are checking if mlfqs==1 every 100 tick calls load_avg , iterate over all threads and calls recent_calc , and increment the recent_cpu every tick

And if ticks is divisible by 4 iterate over all threads and calls priority_calc.



```c
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  if (thread_mlfqs){
    recent_inc();
    if (ticks % 4 == 3){
      // all_priority_calc();
      thread_foreach(priority_clac,NULL);
    }
    if (ticks % TIMER_FREQ == 0){
      load_avg_calc();
      thread_foreach(recent_clac,NULL);
      // priority_clac(thread_current());
    }
  }

}
```

----ALGORITHMS ----

The main logic is handled in or revolves around schedule(). We decide to work the normal ready list ordered with threads' priorities descending . The thread with maximum priority simply be the first thread so the next thread to run is be the first thread in the running list when schedule. And if there a thread that has much priority than the running thread , we schedule and the running thread inserted in its correct position corresponding to its priority.

In **timer_interupt:**

If we are updating on a TIMER_FREQ step, we:

-Calculate load_avg.

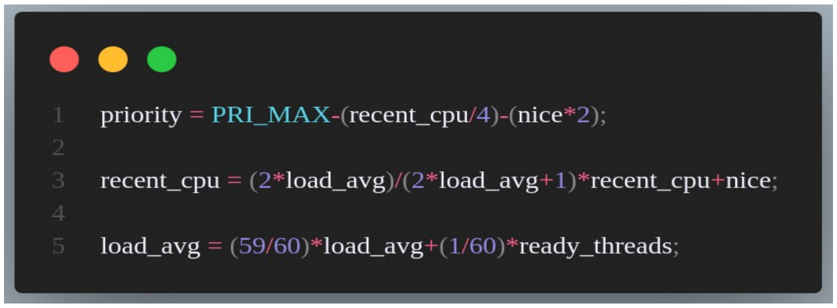
-Loop through every thread using **thread_foreach** function and compute its recent_cpu using **recent_clac** function.

If we are updating on every fourth timer tick, we:

-Loop through every thread using **thread_foreach** function and compute its priority using **priority_calc** function.

All these calculation performed on all threads except idle thread

The equations that used for these calculation are:



```
priority = PRI_MAX-(recent_cpu/4)-(nice*2);

recent_cpu = (2*load_avg)/(2*load_avg+1)*recent_cpu+nice;

load_avg = (59/60)*load_avg+(1/60)*ready_threads;
```

Question2:

-At first the priority for the three threads is calculated using their given nice values and recent_cpu values. Every timer tick the running thread's recent_cpu is incremented by one . And every 4 timer ticks whole thread's priority is calculated and updated then check if there is a thread that has much priority than current thread ,schedule and make the thread with higher-priority run.

timer	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	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

Question3:

the numbers of timer ticks in the table haven't yet reached a hundred so that the only updated recent_cpu is that belongs to current running thread so the priorities for the threads decreases slowly.

The priorities for the threads are close so most of the time they are equal, to resolve these ambiguous situation the thread still running until there is another one that has priority more than running thread so that the thread swapped and the yielded thread inserted in the ready list ordered with the priority descending. It completely match the behavior of our schedule.

Question4:

Every interrupt we preform some small functions such that increment recent_cpu for running thread every tick , update all thread's priorities every four ticks and update load_avg and all thread's recent_cpu every second "hundred ticks". Although , the logic of the algorithm significantly depends on these calculation the big rest of the logic is that insertion of yielded threads ordered in ready list and pop up the thread with maximum priority. This approach preserve the performance of the scheduler as it doesn't provide much overhead every timer interupt.

---- RATIONALE ----

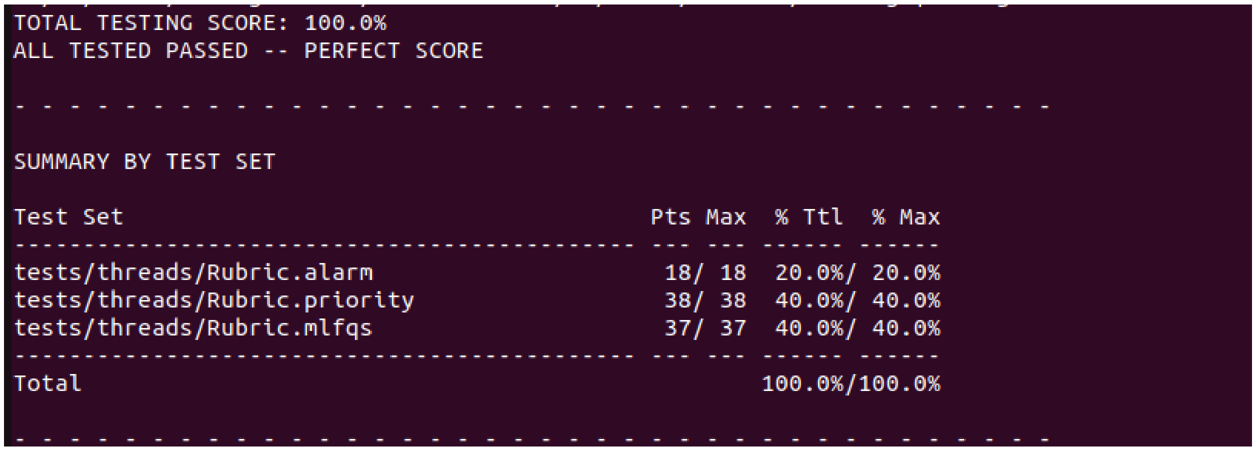
Question5:

we realize that is there are two ways to implement the advanced scheduler one of them which we decide to use and the another is with using list of 64 queue each on of them carry threads that have the same priorities equal to queue index. We decide to implement the first approach as it easy in implementation but has little more overhead rather than the second approach because of in first approach we needed to iterate on ready list every time we swap threads to put ready thread in its right position to keep the list ordered, but in the second approach the only thing you want to do when schedule is to iterate on the queues from max to min checking if they non empty and pop the first thread in max non empty queue. If we have much time so may we implement second approach to improve design or think about more efficient approach to improve the design

Question 6

We decide to implement the fixed-point math as normal integer but it first 14 bit as the float number and the rest of the integer is the int. As we see that 14 bit is enough for the float numbers appear on our calculation. We created a new abstract data type “**real**” which is a struct that has one attribute “int value” ,which simulate the real number and define a macro F indicate the float point position. We define collection of functions that manipulate fixed-point numbers such as conversion from int to real and vice versa and addition, subtraction, multiplication and division on real with real numbers or real with integer numbers.

Tests Check:



```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                      Pts Max  % Ttl  % Max
--------------------------------------------- --- --- ------ ------
tests/threads/Rubric.alarm                     18/ 18  20.0%/ 20.0%
tests/threads/Rubric.priority                  38/ 38  40.0%/ 40.0%
tests/threads/Rubric.mlfqs                     37/ 37  40.0%/ 40.0%
--------------------------------------------- --- --- ------ ------
Total                                                 100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```