Operating Systems – Project 1: Threads

Final Report

Khau Lien Kiet

Team 10

2022/10/1

Department of Computer Science & Engineering

Pohang University of Science and Technology (POSTECH)

Table of Contents

I. Alarm Clock ... 3

a. Data structure ... 3

b. Algorithm ... 3

II. Priority Scheduling ... 6

i. Priority Scheduling ... 6

a. Data Structure ... 6

b. Algorithm ... 6

ii. Priority Donation & Extended Donation ... 11

a. Data Structure ... 11

b. Algorithm ... 11

III. Advanced Scheduler ... 15

a. Data Structure ... 15

b. Algorithm ... 15

IV. Discussion ... 22

<u>Introduction:</u> This report is to demonstrate the implementation (i.e. data structures, algorithm, function added/modified, etc.) for the coding section. In addition, I would explain in detailed the incentives/reasons to modify or add functions as well as illustrate the mechanism if there are any changes in its procedure. Indeed, there are three main sections to describe in the report as (I) Alarm Clock, (II) Priority Scheduling, (III) Advanced Scheduler, and (IV) Discussion. In terms of the Discussion section, it is required to account for my learning retention during the Project 1 implementation. Also, under the final report, I would provide images included with the implementation procedure during my coding section. For a further elaboration, it is more well-explained while reading above description of each part.

I. Alarm Clock

a. <u>Data structure</u>

- **threads/thread.c/sleeping_list:** This list contains the thread after finish executing in the running state. Its main functionality is to manage a currently sleeping thread (performed as the blocked state). For a further description, whenever it exceeds the sleeping duration, it is required to wake up and transfer back to the *ready_list*.

```
/* Alarm Clock: List of containing the threads after executing the running state to sleep in the blocked state.
Whenever it exceeds the sleeping time, it would be transfered */
static struct list sleeping_list;
```

Figure 1: Added sleeping_list data structure in thread.c

- **threads/thread.h/struct thread/local_tick:** It is defined in the thread structure. For each thread, it would have their own *local_tick*, which is represented as the expiration time to wake up and get out of the *sleeping_list*.

```
/* threads/thread.h/struct thread */
struct thread
  {
    //...
    int64_t local_tick;                /* Alarm Clock: Store an expiration time on each thread to wake up in the sleeping_list*/
  };
```

Figure 2: Added local_tick variable in thread.h/struct thread

b. <u>Algorithm</u>

- **Initilize sleeping_list**: It is initialized in the **threads/thread.c/thread_init().** Whenever the thread is created, *sleeping_list* would be initialized to perform as a list of the thread system.

```
/* threads/thread.c/thread_init() */
void
thread_init (void)
{
  //...
  /* When the thread is initialized, simultaneouly
  it is initialized the sleeping_list as a list in its structure*/
  list_init (&sleeping_list);

  //...
}
```

Figure 3: Initialization of the sleeping_list in thread.c/thread_init()

Functions Added/Modified

- Sleep a thread
  - **Add function threads/thread.c/thread_sleep(ticks):** This function is used to put the thread to sleeping state. First, it disables the interrupt to avoid race condition during manipulating the thread list. Then, it receives the *ticks* variable as an argument and also stores the *ticks* value (*ticks* = the beginning of sleeping time + the sleeping duration) to its *local_tick*. If the current thread is not an idle thread, *list_insert_ordered()* function would put the thread into *sleeping_list* in the correct order of *local_tick* with a comparator *sleeping_list_order()* argument *(described below)*. When it goes to the *sleeping_list*, it is necessary to change the status of the thread to *THREAD_BLOCKED* and call *schedule()* to context switch for other threads. Finally, enable the interrupt back to the original state.

```
/* threads/thread.c/thread_sleep() */
/* Set the thread started to blocked and wait in the sleeping_list when inseting it to sleeping_list*/
void thread_sleep(int64_t ticks) {
 struct thread *curr = thread_current();
 enum intr_level old_level;
 // When modified the thread's property, disable interrupt in order to prevent the race condition.
 ASSERT (!intr_context());
 // Ensure the thread is not an idle thread as it was executed in the running state.
 ASSERT (curr != idle_thread);

 old_level = intr_disable();

 // Store in its thread's local_tick about the starting time and the duration of sleeping process.
 curr->local_tick = ticks;

 // Put the thread in sleeping_list with its order (which having a small sleeping time would be the top of the list)
 list_insert_ordered(&sleeping_list, &curr->elem, sleeping_list_order,NULL);

 // Change the state of the thread to THREAD_BLOCKED and call schedule() - Perform as thread_block()
  curr->status = THREAD_BLOCKED;
  schedule ();

 intr_set_level(old_level);
}
```

Figure 4: Added thread_sleep() function in threads/thread.c

  - **Add comparator threads/thread.c/sleeping_list_order(…):** This function is a descending comparator to compare two local ticks from two threads. Therefore, thread having a smallest *local_tick* would at the beginning of the *sleeping_list*.

```
/*ADDED COMPARATOR*/
/* (Descending) Comparator to consider the local_tick for each thread in sleeping_list.
If the thread is contained the smaller local_tick, it would be on the head of the sleeping_list.*/
/* threads/thread.c/sleeping_list_order() */
bool
sleeping_list_order(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED)
{
  struct thread *t1 = list_entry (a, struct thread, elem);
  struct thread *t2 = list_entry (b, struct thread, elem);
  return t1->local_tick < t2->local_tick;
}
```

Figure 5: Added sleeping_list_order() function in threads/thread.c

  - **Modify devices/timer.c/timer_sleep(ticks):** Instead of yielding the thread by *thread_yield()*, the main idea of this modification is to remove the thread from the *ready_list* and put it into the *sleeping_list* if there is a remaining time for its sleeping. Indeed, the argument passed to the *thread_sleep()* would be the starting of sleeping time and sleeping duration.

```c
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
/* devices/timer.c/timer_sleep() */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);

  /* Alarm Clock: If the starting time of sleeping does not exceed the duration of sleeping,
  it still maintains the sleeping state*/
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();
  if (timer_elapsed(start) < ticks) {
    thread_sleep(start + ticks);
  }
}
```

Figure 6: Modified timer_sleep() function in devices/timer.c

- Wakeup a thread
  - **Add function threads/thread.c/thread_wakeup(ticks):** It is called when there is a thread that need to wake up from the sleeping state (so-called the blocked state). It will iterate through all the *sleeping_list* to check if there is any thread with *local_tick* smaller or equal to the current tick. If it satisfies this condition, remove this thread from *sleeping_list* using *list_remove()* and wake up this thread by *thread_unblock()*. Otherwise, it would go to the next thread.

```c
/* Select the thread to wake up from sleeping_list with the local_tick comparison and thus wake it up*/
/* threads/thread.c/thread_wakeup() */
void thread_wakeup(int64_t ticks){
  struct list_elem *e = list_begin(&sleeping_list);
  struct thread *t;

  // Traverse through all the thread elements in the sleeping_list. If there is a thread having a tick smaller
  // than the current tick, it need to wake up by removing out of the sleeping_list and unblock the thread in
  // order to go back to the ready_list. Otherwise, traverse for the next thread to consider.
  while (e != list_end(&sleeping_list)){
    t = list_entry(e, struct thread, elem);

    if (t->local_tick <= ticks){
      e = list_remove(e);
      thread_unblock(t);
    }
    else{
      e = list_next(e);
    }
  }
}
```

Figure 7: Added thread_wakeup() function in threads/thread.c

  - **Modify devices/timer.c/timer_interrupt(…):** On every timer interrupt, it is required to check some threads after sleeping them. If there is any thread needed to wake up, we need to wake up that thread. Indeed, the argument passed through the *thread_wakeup()* would be the *ticks*.

```c
/* Timer interrupt handler. */
/* devices/timer.c/timer_interrupt */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  //...

  /* Alarm Clock: When ever the thread needs to wake up, timer_interrupt would be the caller the unblock that thread
  /to get outside the sleeping_state and wake up to the ready_list.*/
  thread_wakeup(ticks);
}
```

Figure 8: Modified timer_interrupt() function in devices/timer.c

II. Priority Scheduling

i. Priority Scheduling

a. Data Structure

Using the same data structure (no parameter is added in this implementation section)

b. Algorithm

Function added/modified in implementing priority scheduling for the ready state:

- Insert to the *ready_list* with the priority
  - **Add comparator threads/thread.c/thread_priority_compare(…):** This function is considered as an ascending comparator that compare two thread's priority when inserting in this comparator. Therefore, thread having a higher priority would be at the beginning of the list. For a further usage, comparator *thread_priority_compare()* would be the function argument inside the *list_insert_ordered()* to put back the *ready_list* with the correct order, which is particularly modified in *thread_unblock()* and *thread_yield()*.

```
/* (Ascending) Comparator to consider the priorirty of each thread in ready_list for example.
If the thread is contained the higher priority, it would be on the head of the list.*/
/* threads/thread.c/thread_priority_compare() */
bool
thread_priority_compare(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED){
  struct thread *t1 = list_entry (a, struct thread, elem);
  struct thread *t2 = list_entry (b, struct thread, elem);
  return t1->priority > t2->priority;
}
```

Figure 9: Added thread_priority_compare() function in threads/thread.c

  - **Modify threads/thread.c/thread_unblock(thread *t):** To implement the priority scheduling when unblocking one thread, we would use *list_insert_ordered()* to put thread back into *ready_list* in the correct order of priority instead of list_push_back (which simply pushing the thread at the end of the *ready_list*). Indeed, comparator *thread_priority_compare()* would be the function argument inside *list_insert_ordered()* to compare among thread priority. Therefore, when a thread gets unblocked, it would get put back in ready state with the order of priority which makes the *ready_list* be sorted in the correct priority order.

```
/* threads/thread.c/thread_unblock() */
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  /* Priority Scheduling modification*/
  //list_push_back (&ready_list, &t->elem);
  /* To implement the priority scheduling, instead of using the old function to perform FIFO mechanism,
  use the list_insert_ordered with the comparator function thread_priority_compare to insert in the ready_list
  within the priority order. */
  list_insert_ordered(&ready_list, &t->elem, thread_priority_compare, NULL);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

Figure 10: Modified thread_unblock() function in threads/thread.c

- **Modify threads/thread.c/thread_yield(void):** Its modification idea is the same with *thread_unblock()*. To implement priority scheduling when yielding the thread to the *ready_list*, we also replace the *list_push_back()* function with *list_insert_ordered()* with the comparator *thread_priority_compare()*. Thus, if the current thread is not an idle thread, it would perform putting the thread back at *ready_list* in the accurate priority order.

```
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
/* threads/thread.c/thread_yield() */
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    /* Priority Scheduling modification*/
    //list_push_back (&ready_list, &cur->elem);

    /* To implement priority scheduling, instead of using the old function, use list_insert_ordered
    with the comparator thread_priority_compare function is to put back the ready_list with the
    consideration of priority order.*/
    list_insert_ordered(&ready_list, &cur->elem, thread_priority_compare, NULL);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

Figure 11: Modified thread_yield() function in threads/thread.c

- Preempt a thread with the priority when creating a thread and setting priority for thread
  - **Add threads/thread.c/thread_condition_preempt(void):** This function is a condition to preempt one thread in terms of comparing the newly inserted thread and the current running thread (condition (*) in design report). If the priority of the new thread (inserted in the *ready_list*) is higher than the current running thread, it immediately calls *thread_yield()* in order to let the current running thread give up the CPU and yield to the new thread. However, if there is no thread in the *ready_list* to compare within the current running thread, no action is needed to perform.

```
  /* Condition (*):Compare the new thread and the current running thread in terms of the priority.
  If priority of new thread is higher, yield the CPU */
  /* threads/thread.c/thread_condition_preemp() */
void thread_condition_preempt(void){
  //If there is no thread in the ready list to compare with the current running thread, there would be
  // no execution to yield for another thread to run.
  if (list_empty (&ready_list)){
    return;
  }
  struct thread *cur = thread_current();
  struct thread *new_thread = list_entry(list_front(&ready_list), struct thread, elem);
  if (cur->priority < new_thread->priority){
    thread_yield();
  }
}
```

Figure 12: Added thread_condition_preempt() function in threads/thread.c

- **Modify threads/thread.c/thread_create(…):** When create a thread, there would be some changes toward the priority setting. Therefore, add the *thread_condition_preempt()* function is to check whether the condition of preemption and solve the previous issue by the condition (*).

```
/* threads/thread.c/thread_create() */
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  //...

  /* Priority Scheduling section*/
  /* Condition (*):Compare the new thread and the current running thread in terms of the priority.
  If priority of new thread is higher, yield the CPU */
  //To implement priority scheduling, we have to put on the priority comparison mechanism
  // when the thread is created
  thread_condition_preempt();
  return tid;
}
```

Figure 13: Modified thread_create() function in threads/thread.c

- **Modify threads/thread.c/thread_set_priority(new priority):** When the priority of a thread is altered, preemption would happen. Thus, add the *thread_condition_preempt()* is to check for condition (*) to preempt and set priority for the current thread.

```
/* Sets the current thread's priority to NEW_PRIORITY. */
/* threads/thread.c/thread_set_priority() */
void
thread_set_priority (int new_priority)
{
  //...

  /*Priority Scheduling: Since the priority would be changed in a further stage,
  use this function is to check for the condition preemption.*/
  /* Condition (*):Compare the new thread and the current running thread in terms of the priority.
  If priority of new thread is higher, yield the CPU */
  thread_condition_preempt();
}
```

Figure 14: Modified thread_set_priority() function in threads/thread.c

Function added/modified in implementing priority scheduling for synchronization primitives:

- Semaphore
  - **Modify threads/synch.c/sema_down(semaphore *sema):** When it comes to the *sema_down()*, it would check the semaphore value. If the *sema value* is equal to 0, the thread should be pushed back toward the *waiters* list of the semaphore with the correct priority order and get blocked through *thread_unblock()*. Therefore, we would use *list_insert_ordered()* with the comparator *thread_priority_compare()* instead of using *list_push_back()*. However, if it can acquire the semaphore, the *sema value* would decrement by 1.

```c
/* threads/synch.c/sema_down() */
void
sema_down (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      /* Priority Scheduling modification */
      //list_push_back (&sema->waiters, &thread_current ()->elem);

      /* To implement priority scheduling in semaphore, instead of using the old function, use the
      list_insert_ordered function is to insert the thread into its waiters list in the priority order*/
      list_insert_ordered (&sema->waiters, &thread_current()->elem, thread_priority_compare, NULL);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

Figure 15: Modified sema_down() function in threads/synch.c

- **Modify threads/synch.c/sema_up(semaphore *sema):** In terms of the *sema_up()* function, if there exists a waiting thread in the semaphore *waiters* list, before unblocking the front thread or removing that thread out of the *waiters* list, sorting mechanism should be applied in the ascending order with *list_sort()*. The thread with a higher priority would be unblocked directly. Then, it can release the semaphore and the *sema value* increment by 1. Afterwards, to go back to the *ready_list*, condition preemption is necessary to compare between the current running thread and unblocked one in terms of priority. Therefore, we would call *thread_condition_preempt()* to check its condition (*).

```c
/* threads/synch.c/sema_up() */
void
sema_up (struct semaphore *sema)
{
  enum intr_level old_level;

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    /* Priority Scheduling: Before unblocking the thread back to the ready_list, use the sorting mechanism
    in order to sort with the ascending order. Thread has a higher priority would be unblocked first*/
    list_sort(&sema->waiters, thread_priority_compare, NULL);
    thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
  }
  sema->value++;
  /* Priority Scheduling: After unblocking to go back to the ready_list, condition preemption
  would be required to compare the current executing thread and the new unblocked thread in terms of priority.*/
  thread_condition_preempt();
  intr_set_level (old_level);
}
```

Figure 16: Modified sema_up() function in threads/synch.c

- Condition Variable
  - **Add function threads/synch.c/sema_priority_compare(…):** This function is considered as an ascending comparator to compare the priority of the front waiting thread in the semaphore *waiters* list, which is also contained in the condition variable. The priority of front waiting thread in *waiters* list of its semaphore, which is higher than the priority of another, would be performed first. For a further usage, this is a function argument for *list_insert_ordered()* when we put that thread in the *waiters* list of conditional variable with the correct priority order.

```
/*ADDED COMPARATOR*/
/* Priority Scheduling: (Ascending) Comparator to compare the priority of waiting threads in
the waiter list from semaphore, which is contained in the condition variable*/
/* threads/synch.c/sema_priority_compare() */
bool sema_priority_compare(const struct list_elem *a, const struct list_elem *b, void *aux UNUSED){
  struct semaphore_elem *sema_a = list_entry(a, struct semaphore_elem, elem);
  struct list *sema_a_waiter_list = &(sema_a->semaphore.waiters);

  struct semaphore_elem *sema_b = list_entry(b, struct semaphore_elem, elem);
  struct list *sema_b_waiter_list = &(sema_b->semaphore.waiters);

  struct thread *t1 = list_entry (list_begin(sema_a_waiter_list),struct thread, elem);
  struct thread *t2 = list_entry (list_begin(sema_b_waiter_list),struct thread, elem);

  return t1->priority > t2->priority ;
}
```

Figure 17: Added sema_priority_compare() function in threads/synch.c

- **Modify threads/synch.c/cond_wait(…):** Regarding the *cond_wait()*, its main function is to wait for signal by the condition variable. When inserting the waiting thread of semaphore to the *waiters* list of condition variable before release the lock, it is needed to ensure that we would insert that thread in the correct priority order. To perform this, instead of using *list_push_back()*, we will use *list_insert_ordered()* with the comparator *sema_priority_order()* argument. Afterwards, it would execute normally as *cond_wait()* does.

```
/* threads/synch.c/cond_wait() */
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  /*Priority Scheduling: To implement the priority scheduling in the condition varilable,
  instead of using the old function, use the list_insert_ordered with the comparator
  sema_priority_compare function is to insert the thread in its waiters list by the priority order*/

  //list_push_back (&cond->waiters, &waiter.elem);
  list_insert_ordered(&cond->waiters, &waiter.elem, sema_priority_compare, NULL);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

Figure 18: Modified cond_wait() function in threads/synch.c

- **Modify threads/synch.c/cond_signal(…):** As regard to the *cond_signal()*, Before call *sema_up()* to signal and unblock the thread, if there exists a thread in the condition variable's *waiters* list, we should apply sorting mechanism through *list_sort()* in order to maintain correct order of priority since there would occur some changes in the priority of its *waiters* list. After successfully sorting, it would perform *sema_up()* as normally to wake up its front thread.

```
/* threads/synch.c/cond_signal() */
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters)){
    /* Priority Scheduling: Before unblocking the thread through the semaphore mechanism,
    it is required to sort the priority in the waiters list of condition variable through
    the usage of list_sort with the comparator sema_priority_compare function.*/
    list_sort(&cond->waiters, sema_priority_compare, NULL);
    sema_up (&list_entry (list_pop_front (&cond->waiters), struct semaphore_elem, elem)->semaphore);
  }
}
```

Figure 19: Modified cond_signal() function in threads/synch.c

ii. Priority Donation & Extended Donation

a. Data Structure

- **threads/thread.h/struct thread/init_priority:** This integer variable is defined in the structure of thread. Thus, it stores the initially unique priority value. After that, the thread can restore the initial priority itself.
- **threads/thread.h/struct thread/wait_on_lock:** This lock-type variable is defined in the structure of thread. Thus, it is a lock that the thread tends to wait for its usage.
- **threads/thread.h/struct thread/donation_list:** This list is defined in the structure of thread. Thus, it contains the threads donated their priority for this thread.
- **threads/thread.h/struct thread/donation_element:** This list element is defined in the structure of thread, where it is contained inside the *donation_list*, thus letting it control the *donation_list*

```
/* threads/thread.h/struct thread */
struct thread
  {
    //...
    /* Priority Donation: Variable used to perform the donation or inheritance*/
    int init_priority ;                 /* Store or Restore the initially unique priority value. */
    struct lock *wait_on_lock;          /* Lock type variable that the thread waits for */
    struct list donation_list;          /* List contains the other threads donated its priority for this thread*/
    struct list_elem donation_element;  /* List element inside the donation_list*/

   //...
  };
```

Figure 20: Added init_priority, wait_on_lock, donation_list, donation_element variables in threads/thread.h

b. Algorithm

Function Added/Modified

- **Modify threads/thread.c/init_thread(…):** While initialize the thread, *init_priority* and *wait_on_lock* variable should be initialized for implementing priority donation. Therefore, it is essential to set *init_priority* be the given priority as well as the *wait_on_lock* to NULL (since newly initialized thread is not waiting for any lock). In addition, for each thread, it would have its own *donation_list*, thus initializing the *donation_list* through *list_init()* to further maintain other threads donated their priority to this thread.

```
/* threads/thread.c/init_thread() */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  //...

  /* Priority Donation: When initialize the thread, it is required to set the initial priority from its given
  priority and set NULL for its wait_on_lock's parameter. Also, initialize the donation_list for each thread.*/
  t->init_priority = priority;
  t->wait_on_lock = NULL;

  list_init(&t->donation_list);

  //...
}
```

Figure 21: Modified init_thread() function in threads/thread.c

- Implementation for Nested Donation
  - **Add comparator threads/thread.c/thread_donate_priority_compare(…):** This function is served as an ascending comparator to compare the priority of *donation_element(s)* inside the *donation_list*. The thread (*donation_element*) having a higher priority in the *donation_list* would be at the front of the *donation_list*.

```c
/*Priority Donation: (Ascending) Comparator to consider the priority of each donation_element in the donation list.
If the thread contained a higher priority during the donation stage, it would stand forward.*/
/* threads/thread.c/thread_donate_priority_compare() */
bool
thread_donate_priority_compare(const struct list_elem *a,const struct list_elem *b, void *aux UNUSED){
  struct thread *t1 = list_entry(a, struct thread, donation_element);
  struct thread *t2 = list_entry(b, struct thread, donation_element);
  return t1->priority > t2->priority;
}
```

Figure 22: Added thread_donate_priority_compare() function in threads/thread.c

- **Add threads/thread.c/donate_priority(void):** This function is used to donate priority for the thread that holds the lock (lock holder). To implement priority inheritance in the extended donation, use a while loop is to iterate through all threads in *donation_list* within an assumption of maximum space *(8)*. If the current thread is waiting for a lock (*wait_on_lock* != NULL), they would perform priority donation if the priority of the current thread is higher than the holder. The mechanism is to assign the holder's priority with the current thread's priority. However, if current thread is not waiting for lock (*wait_on_lock* == NULL), break out of the while loop that not performing any priority donation. Thus, move to the next thread by setting the holder to be the current thread.

```c
/* Use for donating priority for the thread holding the lock (lock holder).*/
/* threads/thread.c/donate_priority() */
void donate_priority (void)
{
  int count;
  struct thread *cur = thread_current ();
  int cur_priority = cur->priority;
  //Assumption: In the Nested Donation, the maximum depth to perform donation is 8.
  while (count < 9){
    count++;
    if (cur->wait_on_lock == NULL) {break;} //If the thread not waiting for any lock (wait_on_lock is NULL), no priority donation
      /* Ortherwise, if there exists a thread waiting for the lock, set the cur_priority for the priority of lock holder.
      Thus, move to next holder to consider.*/
      struct thread *holder = cur->wait_on_lock->holder;
      holder->priority = cur_priority;
      cur = holder;
  }
}
```

Figure 23: Added donate_priority() function in threads/thread.c

- **Modify threads/synch.c/lock_acquire(struct lock *lock):** Before acquiring the lock, it is necessary to consider this condition. If there exists holder holding a lock that is requested from the current thread, we need to put the current thread into the holder's *donation_list* through *list_insert_ordered()* within the comparator *thread_donate_priority_compare()* to guarantee the accurate priority order in the *donation_list*. Afterwards, perform the priority donation through *donate_priority()* is to donate the current thread's priority toward to the lock holder thread's priority. If there is no holder holding the lock, the current thread could grab the lock and indicate that the lock is being held through defining the lock holder thread to be the current thread.

```c
/* threads/synch.c/lock_acquire() */
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  //...

  /* Priority Donation: (Implemented for Nested Donation) If there exists a current thread requested
  for a lock which is grabbed by lock holder, we put that thread in the donation_list with the comparator
  thread_donate_priority_compare function as well as consider that thread as the donation_element.
  Afterwards, perform the donate_priority() to donate the priority of the current thread to the lock holder.*/
  struct thread *cur = thread_current();
  if (lock->holder){
    cur->wait_on_lock = lock;
    //list_push_back(&lock->holder->donation_list, &cur->donation_element);
    list_insert_ordered(&lock->holder->donation_list, &cur->donation_element, thread_donate_priority_compare, NULL);
    donate_priority();
  }

  /* Then, indicate the lock is being held and thus the current thread acquires the lock, which define
  the lock holder for the current thread.*/
  sema_down (&lock->semaphore);

  cur->wait_on_lock = NULL;
  lock->holder = cur;
}
```

Figure 24: Modified lock_acquire() function in threads/synch.c

- Implementation for Multiple Donation
  - **Add threads/thread.c/lock_remove(struct lock *lock):** This function is to remove a thread holding the lock from the *donation_list*. To implement it, we use for loop to iterate all *donation_element* in *donation_list* of that current thread. If there is a thread waiting for a lock (donating its priority to the holder) that is the same as the lock needed to remove thread from *donation_list*, it directly removes that *donation_element* from the *donation_list*.

```c
/* Remove the thread holding that lock on the donation_list*/
/* threads/thread.c/lock_remove() */
void lock_remove(struct lock *lock){
  struct thread *cur = thread_current();
  struct list_elem *e;

  /* Iterate through the donation_list of the current thread. If the thread is waiting the same lock as
  the one needed to remove thread from donation_list, that thread should be removed out of the donation_list directly*/
  for (e = list_begin(&cur->donation_list); e != list_end(&cur->donation_list); e = list_next(e)){
    struct thread *t = list_entry(e, struct thread, donation_element);
    if (t->wait_on_lock == lock){
      list_remove(&t->donation_element);
    }
  }
}
```

Figure 25: Added lock_remove() function in threads/thread.c

  - **Add threads/thread.c/reset_priority(void)**: This function is to ensure the current thread has the accurate priority. Before checking if *donation_list* is empty (no thread in *donation_list*), reset or restore the thread priority to initial priority with *init_priority*. If there exists the thread in the current thread's *donation_list*, it is required to sort the *donation_list* through using *list_sort()* function with *thread_donate_priority_compare()* as a function argument. Then, if the remaining thread among the *donation_list* has higher priority than current priority, the current priority should be altered toward that higher priority. The main idea of this implementation is that after removing the thread from *donation_list*, the holder priority (current thread in this case) should be set toward the highest priority among those remaining threads in the *donation_list*.

```c
/* Ensure the current thread has the proper priority.*/
/* threads/thread.c/reset_priority */
void reset_priority(void)
{
  struct thread *cur = thread_current ();
  cur->priority = cur->init_priority;
  /* After remove the thread holding the lock, it is required to sort the donation_list in the right order of its priority.
  Then, set the current thread with the highest priority among the remaining threads in the donation_list.*/
  if (!list_empty (&cur->donation_list)) {
    list_sort (&cur->donation_list, thread_donate_priority_compare, NULL);
    struct thread *top = list_entry (list_front(&cur->donation_list), struct thread, donation_element);
      if (top->priority > cur->priority){
        cur->priority = top->priority;
      }
    }
}
```

Figure 26: Added reset_priority() function in threads/thread.c

- **Modify threads/synch.c/lock_release(struct lock *lock)**: When the lock is released, we would remove the thread waiting for the lock in the *donation_list* by using the *lock_remove()* function. In order to maintain the accurate priority order in the *donation_list*, it is essential to reset the priority of the lock holder and sort remaining threads in the *donation_list* by using *reset_priority().*

```c
/* threads/synch.c/lock_release()*/
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  lock->holder = NULL;
  //...

  /* Priority Donation: (Implemented for Multiple Donation) When the lock is released, it is essential to
  remove the thread waiting for the lock in the donation_list. This leads to the change in the donation_list
  in terms of the priority. Therefore, reset_priority is the way to set the priority for the
  remaining threads in the donation_list properly.*/
  lock_remove(lock);
  reset_priority();

  sema_up (&lock->semaphore);
}
```

Figure 27: Modified lock_release() function in threads/synch.c

- Examination and modification of priority
  - **Modify threads/thread.c/thread_set_priority(new priority)**: While performing the priority donation, the priority of the current thread could be altered. To solve it, it is necessary to set the current thread's *init_priority* (initial priority) to be a *new_priority*. Then, its *init_priority* would be updated through *reset_priority()* in order to maintain the accurate priority during continuously changing priority from Priority Donation.

```c
/* threads/thread.c/thread_set_priority() */
void
thread_set_priority (int new_priority)
{
  //...
  /* Priority Donation: Before reset priority, it is essential to set its
  initial priority to the new_priority since there are some cases that change
  the priority value of the current thread.*/
  thread_current ()->init_priority = new_priority;
  reset_priority();

  //...
}
```

Figure 28: Modified thread_set_priority() function in threads/thread.c

  - **Modify threads/thread.c/thread_get_priority()**: It does not need to modify since we did solve with the *thread_set_priority()* function

III. Advanced Scheduler

a. Data Structure

- **threads/thread.h/struct thread/nice**: This integer variable is defined in the structure of thread. It represents how nice the thread is. The more "niceness" means that it had more tendency to give up the thread to others. (Range from -20 to 20).
- **threads/thread.h/struct thread/recent_cpu**: This real number variable is defined in the structure of thread. It represents how much of CPU cycle being used by a that thread. The more CPU cycle being used, the higher *recent_cpu* it would have.
- **threads/thread.c/load_avg**: This real number variable is defined in the src/threads/thread.c since it is considered as the thread system-level. Therefore, itself would be the global variable. However, its main function is to show the average number of threads that are possible to run at the last minute, thus inferring that how busy the system is. The more threads wait for their turn to execute, the higher value in the *load_avg*.
- Default value defined
    - **threads/thread.h/PRI_MAX** (63): Already defined in the naïve PINTOS
    - **threads/thread.h/PRI_MIN** (0): Already defined in the naïve PINTOS
    - **threads/thread.h/NICE_DEFAULT** (0): This default value is defined in the src/threads/thread.h, which would be assigned default for the *nice* value
    - **threads/thread.h/RECENT_CPU_DEFAULT** (0): This default value is defined in the src/threads/thread.h, which would be assigned default for the *recent_cpu* value.
    - **threads/thread.h/LOAD_AVG_DEFAULT** (0): This default is defined in the src/threads/thread.h, which would be assigned default for the *load_avg* value.

```
/* threads/thread.h */
/* Advanced Scheduler: Define three default values for three parameters (nice, recent_cpu, load_avg)*/
#define NICE_DEFAULT 0
#define RECENT_CPU_DEFAULT 0
#define LOAD_AVG_DEFAULT 0
```

Figure 29: Define NICE_DEFAULT, RECENT_CPU_DEFAULT, LOAD_AVG_DEFAULT variables in threads/thread.h

b. Algorithm

Function added/modified for MLFQS

- Initialization of variables:
    - **Modify threads/thread.h/struct thread**: In the struct thread, it is required to initialize two thread-level integer parameters (*nice* and *recent_cpu*)

```
/* threads/thread.h/struct thread */
struct thread
  {
    //...

    /* Advanced Scheduler: Variable to perform Multilevel-Feedback Queue Scheduling (MLFQS)*/
    int nice;                         /* Represent how nice the thread is*/
    int recent_cpu;                   /* Represent how much of CPU cycle that a thread has used*/


   //...
  };
```

Figure 30: Added nice and recent_cpu variables in threads/thread.c/struct thread

- **Add variable threads/thread.c/load_avg**: Since *load_avg* is the system-level parameter, it is necessary to be initialized as the global variable

```c
/* threads/thread.c */
/* Advanced Scheduler: Add a system-level variable*/
int load_avg;                 /* Show the average number of threads that are possible to run at the
                                 last minute. Thus, represent how busy the system is.*/
```

Figure 31: Added load_avg variable in threads/thread.c

- **Modify threads/thread.c/init_thread(…)**: When initializing the thread, under the thread-level, *nice* and *recent_cpu* would be assigned for their default value, which are *NICE_DEFAULT* and *RECENT_CPU_DEFAULT*, respectively.

```c
/* threads/thread.c/init_thread() */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  //...

  /* Adavanced Scheduler: Initialize nice and recent_cpu variable with its defined default value.*/
  t->nice = NICE_DEFAULT;
  t->recent_cpu = RECENT_CPU_DEFAULT;


  //...
}
```

Figure 32: Initialize nice and recent_cpu variables in threads/thread.c/init_thread()

- **Modify threads/thread.c/thread_start(void)**: When a thread is started, under the system-level, *load_avg* is needed to declare for it default, which is *LOAD_AVG_DEFAULT*.

```c
/* Starts preemptive thread scheduling by enabling interrupts.
   Also creates the idle thread. */
/* threads/thread.c/thread_start() */
void
thread_start (void)
{
  //...

  /* Advanced Scheduler: Since the load_avg is a system-level variable, when the thread is started,
  set the load_avg with its defined default value.*/
  load_avg = LOAD_AVG_DEFAULT;

  //...
}
```

Figure 33: Declare load_avg variable in threads/thread.c/thread_start()

- Implementation Fixed-Point Real Arithmetic
- According to the Appendix B, PINTOS does not support floating-point arithmetic in Kernel so that we need to implement a fixed-point arithmetic
- Its mechanism is based on the 2’s complement convention
  - *f is represented as a fixed-point number (1 bit for sign, 17 bits for integer section and 14 bits for decimal section), which is 17.14 format.*
  - *n is represented as integer*
  - *x and y are both represented as the fixed-point number*

No.	Operation	Implementation	Function used
1	Convert n to fixed-point number	$n * f$	int_to_fp (int n)
2	Convert x to integer (rounding toward zero)	$\frac{x}{f}$	fp_to_int_zero (int x)

3	Convert x to integer (rounding to nearest)	$\frac{x+\frac{f}{2}}{f}\ if\ x \geq 0$ $\frac{x-\frac{f}{2}}{f}\ if\ x \leq 0$	fp_to_int_round(int x)
4	Add x and y	$x+y$	add_two_fp (int x, int y)
5	Subtract y and x	$x-y$	sub_two_fp (int x, int y)
6	Add x and n	$x+n*f$	add_fp_int (int x, int n)
7	Subtract n from x	$x-n*f$	sub_fp_int (int x, int n)
8	Multiply x by y	$\left(\left(int64\_t\right)x\right)*y/f$	mul_two_fp (int x, int y)
9	Multiply x by n	$x*n$	mul_fp_int (int x, int n)
10	Divide x by y	$\left(\left(int64\_t\right)x\right)*f/y$	div_two_fp (int x, int y)
11	Divide x by n	x/n	div_fp_int (int x, int n)

- All the function used would be defined in the created file – fixed_point_arithmetic.h

- Implement MLFQS through calculation and recomputing
    - **Add threads/thread.c/mlfqs_priority(thread *t)**: When passing a thread pointer as an argument, this function would calculate the priority of that thread based on its *recent_cpu* and *nice* value with the usage of the *fixed-point arithmetic*'s operation if the thread is an idle thread. The main idea of this function is according to the equation:

$$priority = PRI\_MAX - \frac{recent\_cpu}{4} - nice * 2$$

```
/* Calculate the priority with nice and recent_cpu
Equation: priority = PRI_MAX - (recent_cpu)/4 - (nice)*2 */
/* threads/thread.c/mlfqs_priority() */
void mlfqs_priority(struct thread *t){
  if ( t == idle_thread){ return;}
  t->priority =  fp_to_int_zero(add_fp_int(div_fp_int(t->recent_cpu, -4), PRI_MAX - (t->nice)*2));
}
```

Figure 34: Added mlfqs_priority() function in threads/thread.c

- **Add threads/thread.c/mlfqs_recent_cpu(thread *t)**: When passing a thread pointer as an argument, this function would calculate the *recent_cpu* used in that thread based on the *nice* and *load_avg* value with the usage of the fixed-point arithmetic's operation if the thread is an idle thread. The main idea of this function is according to the equation:

$$decay = \frac{2 * load\_avg}{2 * load\_avg + 1}$$

$$recent\_cpu = decay * recent\_cpu + nice$$
$$= \frac{2 * load\_avg}{2 * load\_avg + 1} * recent\_cpu + nice$$

```
/* Calculate or update the recent_cpu with the decay determined by load_avg
Equation: recent_cpu = decay * recent_cpu + nice
    where decay = (2*load_avg)/((2*load_avg)+1) */
/* threads/thread.c/mlfqs_recent_cpu() */
void mlfqs_recent_cpu(struct thread *t){
  if ( t == idle_thread){ return;}
  int decay = div_two_fp(mul_fp_int(load_avg,2), add_fp_int(mul_fp_int(load_avg,2),1));
  t->recent_cpu = add_fp_int(mul_two_fp(decay,t->recent_cpu),t->nice);
}
```

Figure 35: Added mlfqs_recent_cpu() function in threads/thread.c

- **Add threads/thread.c/mlfqs_load_avg(void)**: This function is to calculate the *load_avg* in every second. Since *load_avg* would be demonstrated as the system-level variable, it would calculate differently by two cases. However, in general, the main idea is according to this equation:

  $$load\_avg = \frac{59}{60} * load\_avg + \frac{1}{60} * ready\_threads$$

  If the current thread is an idle thread, the *ready_threads* would be the size of the *ready_list*, which is the number of threads in the *ready_list*.
  If the current thread is not an idle thread (executable one), the *ready_threads* would be the size of the *ready_list* and the thread itself.

  Therefore, ready_threads = size of the ready_list + 1

```c
/* Calculate or update the load_avg
Equation: load_avg = (59/60)*load_avg + (1/60)*ready_threads
where ready_theads: number of threads in the ready_list and threads in the executing stage at the updated time*/
/*  threads/thread.c/mlfqs_load_avg() */
void mlfqs_load_avg (void){
  int ready_threads = list_size(&ready_list);
  struct thread *cur = thread_current();
  if (cur != idle_thread){
    ready_threads = ready_threads + 1;
  }
  load_avg = add_two_fp(mul_two_fp(div_two_fp(int_to_fp(59),int_to_fp(60)),load_avg),
              mul_fp_int(div_two_fp(int_to_fp(1),int_to_fp(60)),ready_threads));
}
```

Figure 36: Added mlfqs_load_avg() function in threads/thread.c

- **Add threads/thread.c/mlfqs_recent_cpu_increment(void)**: In every timer interrupt, the *recent_cpu* value would increment by 1 to indicate that thread has been used the CPU cycle. After that thread uses 4 ticks on the CPU, it would lessen its priority. The equation would simply be:

  $$recent\_cpu = (current) recent\_cpu + 1$$

```c
/* Increment the recent_cpu by 1 in every timer_interrupt
Equation: recent_cpu = (current thread) recent_cpu + 1*/
/*  threads/thread.c/mlfqs_recent_cpu_increment() */
void mlfqs_recent_cpu_increment(void){
  struct thread *cur = thread_current();
  if (cur != idle_thread){
    cur->recent_cpu = add_fp_int(cur->recent_cpu,1);
  }
}
```

Figure 37: Added mlfqs_recent_cpu_increment() function in threads/thread.c

- **Add threads/thread.c/mlfqs_priority_recompute(void)**: In every 4 ticks, it is required to calculate its priority since there would be some changes in *recent_cpu* values. In terms of the implementation, iterate through every thread in the list from *all_list* in order to recompute each priority of every thread by *mlfqs_priority()* function.

```c
/* Recompute the priority of all threads in every 4 ticks.
Mechanism: Iterate all the threads from all_list and recalculate the priority by the mlfqs_priority() function*/
/*  threads/thread.c/mlfqs_priority_recompute() */
void mlfqs_priority_recompute(void){
  struct list_elem *e;

  for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e)){
    struct thread *t = list_entry(e, struct thread, allelem);
    mlfqs_priority(t);
  }
  list_sort(&ready_list, thread_priority_compare, NULL);
}
```

Figure 38: Added mlfqs_priority_recompute() function in threads/thread.c

- **Add threads/thread.c/mlfqs_recent_cpu_recompute(void)**: In every second, it is essential to calculate the *recent_cpu* value. Regarding the implementation, iterate through every thread in the list from *all_list* in order to recompute the *recent_cpu* of every thread by *mlfqs_recent_cpu()* function.

```c
/* Recompute the recent_cpu of all threads in every 1 second
Mechanism: Iterate all the threads from all_list and recalculate the recent_cpu by the mlfqs_recent_cpu() function*/
/* threads/thread.c/mlfqs_recent_cpu_recompute() */
void mlfqs_recent_cpu_recompute(void){
  struct list_elem *e;

  for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e)){
    struct thread *t = list_entry(e, struct thread, allelem);
    mlfqs_recent_cpu(t);
  }

}
```

Figure 39: Added mlfqs_recent_cpu_recompute() function in threads/thread.c

- **Modify threads/thread.c/timer_interrupt(…)**:

- Before interrupting the thread to wake up, it is vital to check the condition of the MLFQS usage. If the system is implemented with MLFQS, start to increment the *recent_cpu* by 1 through *mlfqs_recent_cpu_increment()* to indicate that the thread is using the CPU cycle.
- Also, check the ticks are passed to regularly update or recompute some MLFQS's parameters
- In every 1 second (determined by mod of *TIMER_FREQ*), it is required to calculate the *load_avg* as well as recalculate its *recent_cpu* value.
- In every 4 ticks (determined by mod of *4)*, it is required to recompute the priority since there would be a change in *recent_cpu* before then.

```c
/* Timer interrupt handler. */
/* devices/timer.c/timer_interrupt */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  /* Advanced Scheduler: Before interrupting the thread to wake up, check some condition in terms of
  MLFQS usage. If implemented MLFQS, increment recent_cpu by 1 in every timer interrupt.*/
  if (thread_mlfqs){
    mlfqs_recent_cpu_increment();

    // In every 1 second (determined by TIMER_FREQ), calculate the load_avg and recompute recent_cpu of all threads.
    if (ticks % TIMER_FREQ == 0){
      mlfqs_load_avg();
      mlfqs_recent_cpu_recompute();
    }
    // In every 4th ticks, recompute the priority of all threads.
    if (ticks % 4 == 0){
      mlfqs_priority_recompute();
    }
  }

  //...
}
```

Figure 40: Modified timer_interrupt() function in devices/timer.c

Function added/ modified for Priority Scheduling in MLFQS

- No implement priority donation in MLFQS:
  - **Modify threads/synch.c/lock_acquire()**: When implementing MLFQS option, it is required to disable the priority donation. In the implementation, use the *thread_mlfqs* boolean to check the usage of MLFQS. If being used, *sema_down()* to indicate the lock is held as well as let the current thread holds a lock immediately and return.

```
/* threads/synch.c/lock_acquire() */
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  /* Advanced Scheduler: While using the MLFQS, disable the priority donation*/
  if (thread_mlfqs){
    sema_down(&lock->semaphore);
    lock->holder = thread_current();
    return;
  }
  //...
}
```

Figure 41: Modified lock_acquire() function in threads/synch.c

  - **Modify threads/synch.c/lock_release()**: When implementing MLFQS option, it is required to disable the priority donation. In the implementation, use the *thread_mlfqs* boolean to check the usage of MLFQS. If being used, *sema_up()* to indicate the lock is free and immediately return.

```
/* synch.c/lock_release() */
void
lock_release (struct lock *lock)
{

  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  lock->holder = NULL;

  /* Advanced Scheduler: While using the MLFQS, disable the priority donation*/
  if (thread_mlfqs){
    sema_up (&lock->semaphore);
    return;
  }

  //...
}
```

Figure 42: Modified lock_release() function in threads/synch.c

  - **Modify threads/thread.c/thread_set_priority()**: When implementing MLFQS option, it is required to disable the priority setting by using *thread_mlfqs* boolean to check. If being used, return immediately.

```
/* Sets the current thread's priority to NEW_PRIORITY. */
/* threads/thread.c/thread_set_priority() */
void
thread_set_priority (int new_priority)
{
  //* Advanced Scheduler: When using MLFQS, setting a priority for a thread is disabled.*/
  if (thread_mlfqs){
    return;
  }
  //...
}
```

Figure 43: Modified thread_set_priority() function in threads/thread.c

- Implementation Priority Scheduling for MLFQS option:
    - **Implement threads/thread.c/thread_set_nice(int nice)**: This function is to set the *nice* value for the current thread to calculate its priority. While changing the value, it is safer to disable the interrupt in order not to let internal and external interrupt intervene in the process. Then, it is required to pass the current *nice* value to calculate the priority of a thread through *mlfqs_priority()*. Also, it is necessary to consider the condition to preempt the thread if the current is not the idle thread. When finishing the change value state, we can enable the old level of interrupt to its original state.

```
/* Sets the current thread's nice value to NICE.*/
/* threads/thread.c/thread_set_nice() */
void
thread_set_nice (int nice UNUSED)
{
  /* Firstly, disable the interrupt for not allowing any internal or external interrupt.
  If getting a nice value, calculate the priority of the current thread. If the current thread is
  not idle thread, it would preempt. Lastly, return the interrupt level to the orginal state.*/
  enum intr_level old_level = intr_disable();
  struct thread *cur = thread_current();
  cur->nice = nice;
  mlfqs_priority(cur);

  if (cur != idle_thread){
    thread_condition_preempt();
  }
  intr_set_level(old_level);

}
```

Figure 44: Modified thread_set_nice() function in threads/thread.c

    - **Implement threads/thread.c/thread_get_nice(void)**: This function is to return the *nice* value of the current thread. The mechanism of disable and enable the interrupt is maintained since values are changing. To implement, assign the current thread's *nice* value to the parameter *nice* and thus return itself.

```
/* Returns the current thread's nice value. */
/* threads/thread.c/thread_get_nice() */
int
thread_get_nice (void)
{
  /* Mechanism of disable and enable interrupt is the same of an above function. This function is to
  return the current thread's nice value.*/
  enum intr_level old_level = intr_disable();
  struct thread *cur = thread_current();
  int nice = cur->nice;
  intr_set_level(old_level);
  return nice;
}
```

Figure 45: Added thread_get_nice() function in threads/thread.c

    - **Implement threads/thread.c/thread_get_load_avg(void)**: This function is to return 100 times the current system *load_avg* (rounded to the nearest integer). To implement, with the usage of operations in *fixed_point_arithmetic*, we could multiple the original *load_avg* by 100 times and return to the nearest integer. Indeed, we still apply the mechanism of disable and enable interrupt for a safer modification.

```c
/* Returns 100 times the system load average. */
/* threads/thread.c/thread_get_load_avg() */
int
thread_get_load_avg (void)
{
  /* With the usage of operation in fixed-point arithmetic, multiply the load_avgin 100 times
  and transform back to the integer number */
  enum intr_level old_level = intr_disable();
  int new_load_avg = fp_to_int_round(mul_fp_int(load_avg,100));
  intr_set_level(old_level);
  return new_load_avg;
}
```

Figure 46: Added thread_get_load_avg() function in threads/thread.c

- **Implement threads/thread.c/thread_get_recent_cpu(void)**: This function is to return 100 times the current thread's *recent_cpu* value (rounded to the nearest integer). To implement, with the usage of operations in *fixed_point_arithmetic*, we could multiple the original *recent_cpu* by 100 times and return to the nearest integer. Indeed, we still apply the mechanism of disable and enable interrupt for a safer modification.

```c
/* Returns 100 times the current thread's recent_cpu value. */
/* threads/thread.c/thread_get_recent_cpu() */
int
thread_get_recent_cpu (void)
{
  /* With the usage of operation in fixed-point arithmetic, multiply the current thread's recent_cpu
  100 times and transform back to the integer number */
  enum intr_level old_level = intr_disable();
  struct thread *cur = thread_current();
  int new_recent_cpu = fp_to_int_round(mul_fp_int(cur->recent_cpu,100));
  intr_set_level(old_level);
  return new_recent_cpu;
}
```

Figure 47: Added thread_get_recent_cpu() function in threads/thread.c

IV. Discussion

During the project 1, I have simultaneously applied the knowledge from the lecture toward the implementation and learned from the solution itself. In terms of the *Alarm Clock* section, from the naïve PINTOS, instead of applying the *busy waiting* – which could be wasted for our resource, the strategy tends to let the thread sleep directly (performed as the blocked state) within the certain amount of sleeping time and wake it up afterwards. Regarding the *Priority Scheduling* part, *priority donation* is not meticulously introduced in the lecture. However, I did acquire this knowledge from the project within two ways to resolve the priority inversion, which are *Nested Donation* and *Multiple Donation*. By these strategies, it is guaranteed that the high priority thread could be executed instead of being preempted by another lower priority thread. As regards to *Advanced Scheduler*, I could understand rigorously the mechanism behind *Multi-level Feedback Queue Scheduling (MLFQS)* and thus apply these main ideas in the implementation. For instance, to implement it, there are introduced with three more variable (*nice*, *recent_cpu*, *load_avg*) to determine the priority for each thread at one priority level. As the priority is changing dynamically under *MLFQS*, the target threads have more tendency to use the CPU and get executed during its turn.