Operating Systems – Project 1: Threads

Design Report

Khau Lien Kiet
Nguyen Duong Tung

Team 10

15/09/2022

Department of Computer Science & Engineering
Pohang University of Science and Technology (POSTECH)

Table of Contents

Introduction: This report is aimed to let us plan and synthesize the idea to design for the improvement of naïve PINTOS. There are three main sections required to result in the new implementation such as (I) Alarm Clock, (II) Priority Scheduling, and (III) Advanced Scheduler. Under this Design Report of Project 1, it is required to illustrate the analysis of both the current thread system and synchronization primitives (lock, semaphore). Indeed, we allocate those sections inside "Analysis of current implementation" in our report. Afterwards, "Problem" toward each specific component would be underlined and resolved by demonstrating our "Solution" section. Moreover, it is essential to achieve each criteria modification that is included in the Project 1 Description. Therefore, our "New implementation" is to satisfy its requirements by providing the main idea implementation as well as the data structures/functions needed to add/modify.

I. Alarm Clock

a. Analysis of current implementation

- Description: Alarm Clock is used to wake up the sleeping threads. After a period of times (ticks), it is considered as the internal kernel function to restart the process, which put the thread in the ready state to the running state.

- Main current structure:
- **ready_list:** The list contains all the threads to get ready for executing

- Main current function:
- **src/devices/timer.c/timer_sleep():** This function puts the current thread to sleep in an amount of ticks and switch between the running state to the ready state.

```
/* devices/timer.c */
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

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

Figure 1: timer_sleep() function in the naïve PINTOS

  - Mechanism:
    - When timer_sleep() function is called, it would record the current time to the timer_ticks(). Indeed, timer_sleep() receives an argument (ticks) that is its sleeping time or scientifically speaking, ticks is considered as the amount of time to put the thread to sleep in the ready_list.
    - Basically, timer_sleep() will continuously check whether its sleeping time through the while loop. In general, when the wake-up time has not been reached, the thread will be pushed back into the ready_list through function thread_yield() which releases the CPU.

- Other subfunctions used:
  - src/devices/timer.c/timer_ticks(): This function returns current tick’s value.
  - src/devices/timer.c/timer_elapsed(start): This function returns the number of ticks passed since the starting point.
  - src/threads/thread.c/thread_yield(): This function puts the thread to the ready_list and yield the control to the CPU.

- **src/threads/thread.c/thread_yield()**: This function yields or releases CPU and puts the thread to the ready_list.

```c
/* threads/thread.c */
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)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

Figure 2: thread_yield() function in the naïve PINTOS

- Mechanism:
  - At the beginning, through the thread_current() function, it would result in the pointer to the current thread.
  - Next, the interrupt is needed to be disabled in order to avoid other internal or external interruptions when accessing shared resources.
  - Simultaneously, if the system is not idle (the thread is idle in which there is not much work/task to get done), that thread needs not to wake up, thus still maintained in the ready_list through list_push_back() to the end of the ready_list.
  - Also, the status of current thread is THREAD_READY, which is represented as in the ready_list
  - Use the schedule() function as the context switch to select one thread from the ready state to the running state.
  - Lastly, the interrupt level would be enabled as in the original state for continuing the process.
- Other subfunctions used:
  - src/threads/thread.c/thread_current(): This function returns the object of current thread.
  - src/threads/interrupt.c/intr_disable(): This function turns interrupts off and thus returns the previous interrupt state.
  - src/threads/interrupt.c/intr_set_level (old_level): This function sets the state of interrupt to the passed argument and returns the previous interrupt state (original state).

- src/lib/kernel/list.c/list_push_back(&ready_list, &cur->elem): This function will insert the passed argument to the end of the ready_list.
- src/threads/thread.c/schedule(): This function works as the context switch.

b. Problem statement

During the current implementation, busy waiting - the process of continuously checking their time- has been used. Therefore, among the sleeping threads, one of those threads could have a chance to use CPU by placing the non-wake-up threads toward the running state. The thread that puts in the ready list by thread_yield() could still quickly push back to the running state by context switch based on scheduler mechanism.

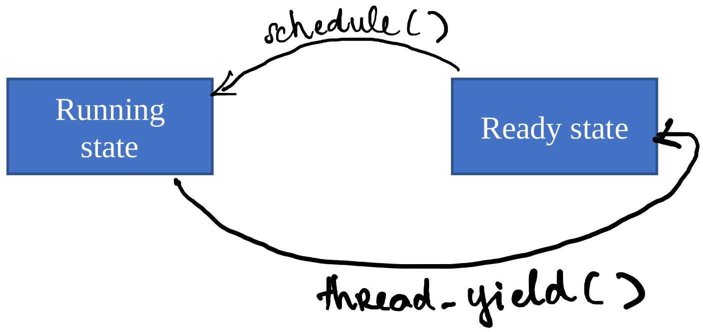

Figure 3: The relation of ready state and running state within functions

The above interpretation would be the main idea of current implementation of alarm clock. Although some threads could wait due to the busy waiting method in the ready state, they continuously use the CPU and Memory. Therefore, CPU resources are certainly wasted.

c. Solution

As mentioned about its problem, to avoid busy waiting while waiting its turn in the ready_list, it is applicable to apply "Sleep – Wakeup" method. After it checks the time in the running state, instead putting straight to the ready_list, the main idea is to put the thread into the sleeping_list. Whenever it is its turn, we wake the thread up, thus putting it in the ready_list. By this change, busy waiting can be solved.

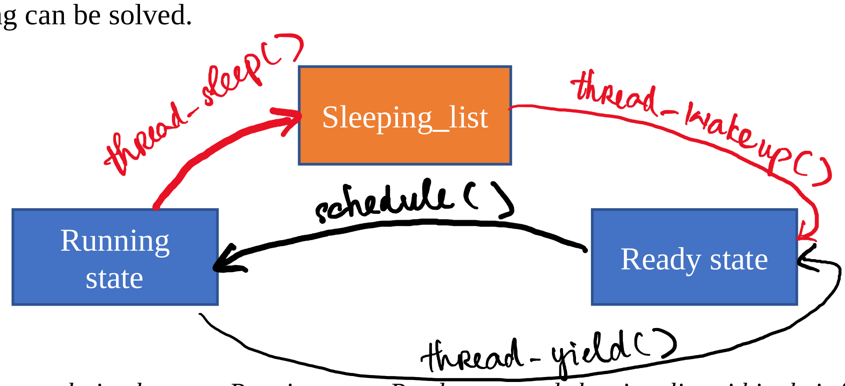

Figure 4: The correlation between Running state, Ready state and sleeping_list within their functions

d. New implementation

- Data Structure added/modified:

- **sleeping_list:** This list contains the threads after executing the running state. Whenever it exceeds the sleeping time or it is time to wake up, it would be transferred to the ready_list.
- **local_tick:** This variable is represented as the expiration time to wake up for each thread.

- Function added:

- **src/threads/thread.c/thread_sleep():** This function is to set the thread state from the running state to a sleeping state (inferred the sleeping_list) and wait after that thread is put to the sleeping_list. This function will be used in timer_sleep() when we want to put thread to sleeping_list.
- **src/threads/thread.c/thread_wakeup():** This function is find the thread which is needed to wake up from the sleeping_list after checking the local_tick. Therefore, that thread would be put back into the ready_list (represented as the ready state). Indeed, this function will be used in the timer_interrupt() function.
- **src/threads/thread.c/sleeping_list_order():** This function acts as a comparator to compare the local_tick of threads in the sleeping_list, which is used as an argument function in list_insert_ordered(). This function will allocate the thread with the smallest local_tick at the beginning of the list. Indeed, the list is sorted in ascending order based on local_tick from each thread in the sleeping_list.

  - Function modified:

- **src/threads/thread.c/thread_init()**: To create a sleeping list, we need to initialize the sleeping_list data structure.
- **src/devices/timer.c/timer_sleep():** Add a new function (thread_sleep()) to place the thread to the sleeping_list.
- **src/devices/timer.c/timer_interrupt():** When time_interrupt() is executed, at every tick, it is mandatory to check whether some threads need to wake up from the sleeping_list. Thus, in terms of new implementation, thread_wakeup() must be called.

II. Priority Scheduling

1/ Priority Scheduling

a. Analysis of current implementation

- Description: In the current implementation, priority scheduling is not implemented since PINTOS uses the FIFO scheduling, so-called the Round-Robin Method which is taking turn after staying in the ready_list in the amount of time. When a thread is gone to the ready state, it would be put at the end of the ready_list. However, it is not sorted by priority instead.

  Likewise, in terms of synchronization primitives (lock, semaphore, condition variable), priority is not considered when lock, semaphore, and condition variable are used in the waiting_list. Also, the lock is implemented with the same mechanism of the PINTOS in general – FIFO.

  To understand its idea, we should understand thoroughly in terms of its basis and functionality. In the existing implementation, one of the most important synchronization primitive methods to solve critical section problems is semaphore. In general, semaphores are nonnegative integer variables that are used to tackle the race condition or deadlock by using two atomic operations that are up and down.

  *Up: increment the value and wake up one waiting thread to claim semaphore value.

  *Down: wait for the semaphore value to be positive then decrease it.

  A semaphore is used when a thread wants to wait for some signals or events that happen first before it continues its execution. The implementation of semaphore in naive Pintos is using two main functions that are sema_up() and sema_down() with each semaphore maintained a list of waiting threads, that used the linked-list implementation in lib/kernel/list.c. For example, a thread T1 creates thread T2 and waits for T2 to finish some execution. T1 can create a semaphore, initialize it to 0, pass it to T2 and call sema_down() to wait. T1 will be in the waiter of the semaphore. When T2 finishes its execution, it would call the sema_up() to let T1 continue the execution.

  The image below is used for easier interpretation of three lists: ready_list, sleeping_list, and waiting list:

Figure 5: The correlation between Running State, Ready_list, Sleeping_list, Waiter_list with functions

- Main current structure:
  - **ready_list**: The list contains all the threads to get ready for executing
  - **waiter_list**: The list contains the threads using the synchronization primitives (where synchronization primitives -lock, semaphore, condition variable- are used to signal the thread in the waiter_list)

- Main current function:

Particularly, in the priority scheduling's aspect, main functions are used when the thread is created through thread_create():

- **src/threads/thread.c/thread_set_priority()**: After created thread, its function would set a priority for a thread with specified value/priority.

```
/* threads/thread.c */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
}
```

Figure 6: thread_set_priority() function in the naïve PINTOS

- **src/threads/thread.c/ thread_get_priority ()**: After set the priority, its function would return the priority of the current thread.

```
/* threads/thread.c */
int
thread_get_priority (void)
{
  return thread_current ()->priority;
}
```

Figure 7: thread_get_priority() function in the naïve PINTOS

- **src/threads/thread.c/schedule()**: This function is used for scheduling and it is used from thread_yield() and thread_block(). When this function is called, the current thread would allocate the next thread to the running state.

```
/* threads/thread.c */
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;

  ASSERT (intr_get_level () == INTR_OFF);
  ASSERT (cur->status != THREAD_RUNNING);
  ASSERT (is_thread (next));

  if (cur != next)
    prev = switch_threads (cur, next);
  thread_schedule_tail (prev);
}
```

Figure 8: schedule() function in the naïve PINTOS

  - Mechanism:
    - Firstly, they initialize three pointers (*cur: pointing to the current running thread, *next: use next_thread_to_run() function for pointing to

the next thread to execute, *prev: pointing to the previous thread finished the previous execution)
        - Some ASSERT checkers is to check the interrupt, its status, and the choice of next thread to execute
        - When the *cur pointer points to the next accurate thread, switch() function is executed. This is a vital function to implement the priority scheduling.
- **src/lib/kernel/list.c/list_push_back()**: This function is used for pushing the thread to the end of the ready_list
- **src/threads/thread.c/thread_yield()**: This function is to yield or release CPU and put the thread in the ready_list.
- **src/threads/thread.c/thread_block()**: This function is to put the thread into the waiter_list. It would be appropriate/efficient when the synchronization primitives are used if there is a thread in waiter_list.

```c
/* threads/thread.c */
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}
```

Figure 9: thread_block() function in the naïve PINTOS

    - Mechanism:
        - When it is called, this function will first change the current thread status to block.
        - After that, it calls the context switch to schedule another thread into the running state.
- **src/threads/thread.c/thread_unblock()**: This function transfers the thread in waiter_list to the ready-to-run state.

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

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

Figure 10: thread_unblock() function in the naïve PINTOS

    - Mechanism:

  - When the function is called, it would disable the interrupt for changing the value and enable the interrupt when the procedure finishes.
  - Afterwards, it would put the thread at the end of the ready_list and change the thread status to READY.

- Functions to address the synchronization primitives' part:
  - **src/threads/synch.c/lock_init()**: This function initializes a new lock that not be owned by any thread.
  - **src/threads/synch.c/lock_acquire()**: This function calls this function to acquire lock for current running thread.
  - **src/threads/synch.c/lock_release()**: This function checks if current thread is holding lock, if it is then release lock for other threads.
  - **src/threads/synch.c/cond_init()**: This function initializes new conditional variable.
  - **src/threads/synch.c/cond_wait()**: When condition variable is signaled by another thread, lock is atomically released. Before returning, this function will reacquire lock after the cond is signaled. This function cannot be called without the lock being held by current thread.
  - **src/threads/synch.c/cond_signal()**: The current thread must hold the lock when calling this function. This function would wake up any thread that is waiting for this conditional variable. If none are waiting, it would return without doing anything.

b. Problem Statement

Previously mentioned, threads in the ready_list as well as threads in waiter_list to use the synchronization primitives (lock, semaphore, condition variables) is not considered about priority. For instance, from the ready state to the running state, when the function schedule() is called, the thread at the top of the ready_list would be executed toward the running state. Also, inside the waiter_list to use its primitives, the thread which is requested first would be the one to be executed immediately. This mechanism comes from the FIFO (Round Robin method).
For a further situation, if the thread in the waiter_list is still applied that mechanism, there would occur the Priority Inversion while we did implement priority.

c. Solution

According to the previous problem statement, it is mandatory to implement priority order in two data structures such as ready_list and waiter_list.

d. New implementation

- Function added/modified in ready state:

When inserting into the ready_list, the priority order should be concerned. Specifically, when the thread is added to the ready_list – so called new thread, it is necessary to compare the priority of the new thread and the priority of the current thread (that has the highest priority compared to the other threads in ready_list), which is executed in the running state. If the new thread has a higher priority than the current thread, the scheduler should yield the control to the CPU and let the new thread executed.

  - **src/threads/thread.c/thread_create()**: To implement the priority in this function, we add on the condition (*) in terms of comparing the new thread and the current running thread. If the priority of new thread is higher, thread_yield() would be added to yield to the CPU.

- **src/threads/thread.c/thread_set_priority()**: Likewise, we also use the condition (*) of priority setting into this function.
- Function added: In src/threads/thread.c, create a comparison function called **thread_priority_compare()** for a further usage inside the list_insert_ordered(). To better describe this new function, it is served as the comparator that compares the priority of two threads inserted to this function, particularly the new thread and the current running thread.
- **src/threads/thread.c/thread_unblock() & thread_yield()**: Instead using the list_push_back() function – simply add the thread in the ready_list without order, we could replace it with list_insert_order() implemented with the thread_priority_compare() function argument since it could add to the ready_list within priority.

- Function added/modified in waiting state for using the synchronization primitives:

The main idea is to wake up the waiting threads to use the synchronization primitives in general compared to its priority. Also, we need to prevent the Priority Inversion problem, which could be solved in the next section – Priority donation.

- **src/threads/synch.c/sema_down()**: This function is used by the thread to require the semaphore. It would check whether the semaphore is available, if it is, then the thread gets the semaphore and makes the semaphore unavailable to others. When the thread could not get the semaphore, it is put back into the waiter_list in the priority order. To do this, instead of using the normal list_push_back() function, we will replace it with the list_insert_ordered() used the comparator function thread_priority_compare() for priority implementing.
- **src/threads/thread.c/sema_up()**: Since there are changes in the priority order of threads in wait_list, list_sort() would sort the thread in the descending order. Within the highest priority, it would unblock. Moreover, if the priority of the unblocked thread has higher priority than the current running thread in the running state, add a condition (*), it would yield to the CPU.
- Function added: In src/threads/synch.c, create function called **sema_priority_compare()**: This function is served as comparator, which receive 2 semaphore elements (semaphore.elem structure). It would compare and return the thread priority located at the top of these two's semaphore waiter lists.
- **src/threads/synch.c/cond_wait()**: In the conditional variable (cond) implementation, the waiter list of cond is the list of semaphore. When inserting the element of semaphore back to waiter list of cond before releasing lock, make sure to insert semaphore element back with the top semaphore waiting contained the highest priority thread. To do this, instead of using list_push_back(), replace it by list_insert_ordered() function but with argument of sema_priority_compare() boolean.
- **src/threads/synch.c/cond_signal()**: Similar to sema_up(), the cond_signal() function also uses list_sort() to sort cond waiter_list in descending order before calling sema_up() to unblock or wake up the thread in the top of the waiter list. This is to ensure the semaphore waiter list that unblock has the thread with the accurate priority order since priority in the wait_list could be changed.

2/ Priority donation & Extended Donation

a. <u>Analysis & Problem Statement of current implementation</u>

- Description

  When using the synchronization primitives, priority is implemented in the previous section, but we should be aware of the issue related to all threads' priority, which is called as "Priority Inversion". To solve it, Priority Donation would be the efficient solution.

  In general, priority inversion occurs when the high priority thread could not be prioritized to execute rather than the lower priority. As mentioned in the project description, assume there are three different priorities of thread, H(high), M(Medium), L(Low) within their priority order H>M>L. While L is now held the lock, H is requested for a lock but since L has acquired it so that H could not acquire the lock. However, if H passes the lock control to L, during the stage, M is on the ready_list. Since M has a higher priority than L, M can acquire the lock. Thus, H would never get the lock since H has to wait for M completed its work.

  Since lock has played a vital role in priority donation method, it is required that we understand its meaning and implementation in naïve Pintos. Lock is a synchronization primitive that is used to provide mutual exclusion and prevent race conditions when working with threads. Once a thread acquires the lock, so-called the lock holder, only this thread can release the lock. Hence, this is to ensure that only one thread can enter the critical section at a time, which is protected by lock. The implementation of lock in naïve Pintos is to use two main functions: lock_accquire() and lock_release(). Particularly, lock_accquire() will be used by the thread that wants to acquire the lock. If lock has already been acquired by the other, then put the current one onto the waiter_list. Otherwise, if lock is free since no thread is requested on it, immediately acquire that lock. In terms of lock_release(), it is simply released the lock and thus this would yield to other thread on the waiter_list.

- Main current function:
  - **src/threads/synch.c/lock_acquire()**: This function is called when a thread is requested a lock for itself. Indeed, this function is used to implement and modify in the Nested Donation case *(description below)*.
    - Mechanism:
      - In naïve Pintos, this function will call the sema_down() function with the semaphore of lock as the argument.
      - In its sema_down() function, if there is a thread that is currently holding the lock (sema_value = 0), the thread that called this function would put in the waiter list of semaphore to wait for the lock to be released and also get blocked by thread_unblock() (or get back to the waiter_list) .
      - If there are none holding the lock (sema_value = 1), decrease the sema_value to indicate the lock is held by this thread and thus immediately acquire the lock.

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

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

Figure 11: lock_acquire() function in the naïve PINTOS

- **src/threads/synch.c/lock_release()**: Indeed, this function is used to implement and modify in Multiple Donation *(description below)*. After check if the current thread is holding the lock, immediately release the lock (set the lock holder to NULL). Afterwards, call the sema_up() to wake up other thread that waiting on this lock.
  - Mechanism:
    - This function would first check if the thread that called is really holding the lock or not, if it is then change the lock holder to NULL (indicating the lock is free).
    - Then, it calls the sema_up() with semaphore of lock as the argument.
    - After that, if there are threads waiting for the lock in the waiter list of semaphore, unblock the thread in the top waiter list to acquire the lock.
    - It also increases the sema_value to indicate that the lock is free.

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

  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

Figure 12: lock_release() function in the naïve PINTOS

b. Solution

After elaborating on its problem status, priority donation is needed to implement as H would transfer the priority to L in the previous situation. Therefore, L would have the same priority as H, finished executing and release the lock for H's execution.
However, the above case is a general one so that there would be an extended donation that is necessary to be taken into account. That would be "nested donation" and "multiple donation".

Extended donation:

- Nested Donation

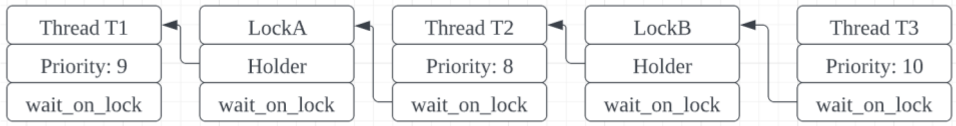

Figure 13: Illustration in the case of nested donation

From the figure above, initially, there are two threads T1 and T2 with priority 9 and 8, respectively. T1 is held a lock A. T2 is held a lock B but requested on the lock A. There are one more thread (T3) added to the waiter_list, which is required to request a lock B and its priority 10. To avoid the priority inversion, nested donation is the efficient solution. To clarify on this point, T3 donates its priority 10 to thread T2. Also, T2 donates its new donated priority 10 to thread T1.

- Multiple donation

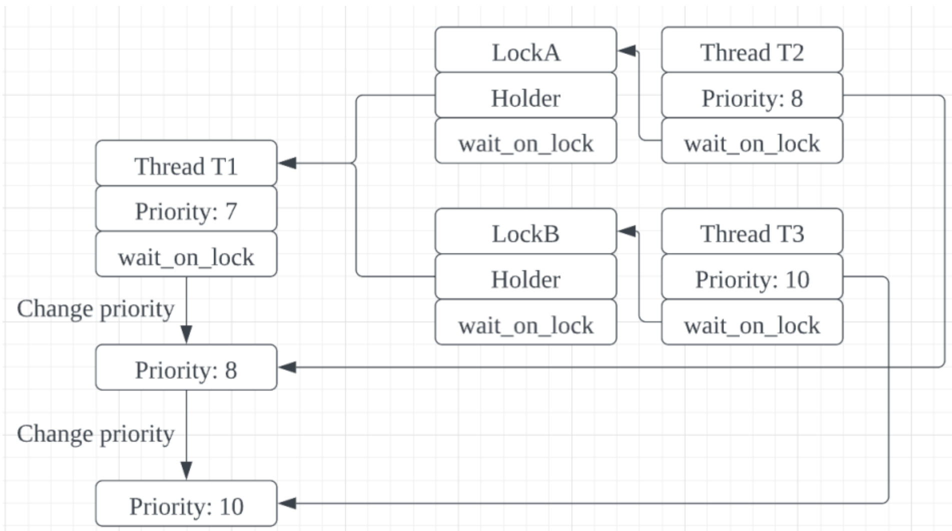

Figure 14: Illustration of multiple donation

From the figure above, there are three block T1, T2, T3 with its priority T3>T2>T1. Assume that T1 is held two locks A and B. Also, T2 requests lock A and T3 requests lock B. For this case, since the priority of T1 is lower than both priority of T2 and T3, T1 needs to yield the lock A for T2 and lock B for T3.

c. <u>New implementation</u>

- Data Structure added
  In terms of the structure of thread, to implement priority donation itself, we should provide four parameters:
  - **init_priority**: This variable stores the initially unique priority value and so after that the thread can restore the initial priority itself.
  - **wait_on_lock**: This variable is a lock that the thread tends to wait for lock’s usage.
  - **donation_list**: The list contains the threads are donated the new priority
  - **donation_element**: This is a parameter to let the element control the donation_list
- Function modified:
  - **src/threads/thread.c/init_thread()**: To implement the priority donation, use the above data structures for initializing for its priority donation such as init_priority (store the original

priority) , wait_on_lock (set to NULL as no thread have given its donation yet) , and donation_list (initialize the donation_list itself).

Implementation for Nested Donation:

- **src/threads/synch.c/lock_acquire()**: There is two types of threads in this modification, which is the current thread and thread held the lock (thread holder). Before sema_down() to acquire a lock, there should be a condition to consider. If there is no thread holder acquired the lock, the current thread would grab the lock immediately. Otherwise, the priority should be passed to the thread holder. However, this depends on priority of current thread and the thread holder. If current thread's priority is larger than the thread holder's priority, donating priority is needed to perform.
  If the thread holder already holds the lock, add this lock to the current thread's wait_on_lock parameter. Also, add the current thread to the donation_list of the thread holder by using list_insert_ordered() with the **thread_donate_priority_compare()** as a comparator function. Then, execute the priority donation through **donate_priority()** function *(description on the donate_priority as below).*
  Finally, sema_down() would be occurred to indicate the lock is held and thus the current thread is acquired the lock. To define some parameters, current thread would become the thread holder holding the lock and set NULL for the wait_on_lock variable.
- Function added
  - **src/threads/thread.c/thread_donate_priority_compare()**: This function has the same purpose as thread_priority_compare() but it would compare the priority on each donation_element.
  - **src/threads/thread.c/donate_priority()**: This function is used to donate priority for the thread that holds the lock. To implement, we would use a while loop to check if the current thread is waiting on a lock (wait_on_lock != NULL). They would perform priority donation if the priority of the current thread is larger than the thread held the lock. However, if thread is not waiting for lock (wait_on_lock = NULL), stop the loop as there is not necessary to perform the priority donation. After that, set the thread holder to be the current thread (therefore, it would go deeper into the thread-lock chain).

Implementation for Multiple Donation

- **src/threads/synch.c/lock_release()**: Before releasing the lock, we must delete the thread that release the lock from the donation list by using the lock_remove() function *(described later)* and reset the priority of the lock holder thread by using reset_priority().
- Function added:
  - **src/threads/synch.c/lock_remove()**: This function uses for loop to iterate all donation_element in donation_list of that current thread. If there is an element in the donation_list has their wait_on_lock pointing to the lock to be released at the time, remove that donation element from the donation_list. Thus, it would have chance to acquire that lock.
  - **src/threads/thread.c/reset_priority()**: This function to make sure 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 are threads remained in donation list, the highest priority in the donation_list

included itself would be chosen. By that, sort the donation_list by their priority order through using list_sort() function with thread_donate_priority_compare() as a function argument. Then, if the thread with highest priority in the remain donation_list is higher than current priority, it would change to the higher one. Otherwise, the current thread is still maintained with initial priority.

d. <u>Examination and modification of priority</u>

- **src/threads/thread.c/thread_set_priority()**: While reset_priority() function is called, the priority of current thread might be altered besides the case of lock or priority donation. To solve it, the initial priority (init_priority) must be updated through the newly created function reset_priority(). Moreover, it would occur that thread has a higher priority than the thread elements in the donation_list. Therefore, we should apply on the priority aspect that if the current thread does not have the highest priority compared to the element in the donation list, condition (*) must be applied which is to yield.
- **src/threads/thread.c/thread_get_priority()**: it does not need to modify since thread_set_priority has been solved the case of priority donation.

III. Advanced Scheduler

a. Analysis of current implementation & Problem Statement

Within the current implementation, normal scheduler would select which thread executed based on its priority. To get further on that point, scheduler mechanism should be re-elaborated when it is not implemented the priority inside.

Thread scheduler mechanism in naïve pintos is the simple First-in-first-on (FIFO) rationale or Round-robin method, which means that which thread got into the ready state first gets to switch to the running state. In the current thread implementation, thread switching is done by the function call schedule(). There are three functions that need to call schedule(), which are thread_exit() – *remove thread from all thread list and while setting its status to be DYING, then call schedule() function to schedule for another thread*, thread_yield() – *straightforwardly used to yield the CPU and thus schedule for another thread*, and thread_block*() – the thread would be scheduled if it is awoken by thread_unblock()*. To analyze thoroughly the function schedule() or the scheduler mechanism itself, before switching threads, it must make sure that the interrupt is disabled and its state different from running. Firstly, the schedule() function will determine the current thread and the next thread by the running thread and next_thread_to_run() function, respectively. In terms of next_thread_to_run(), it would pop and return the thread at the front of the ready_list to further place it into the running state. Afterwards, if the current running thread is not the same as the next thread, schedule() function would call switch_thread() to switch from the current thread to the next running thread. Indeed, switch_thread() would do the low-level job, which is saving the register, CPU's current stack pointer of the current thread and restoring the same information with register and stack pointer of the new thread. Finally, thread_schedule_tail() is to activate the next running thread's property. That provokes the completion of thread switching method from ready state to running state.

After priority scheduling is implemented, there are some changes in the scheduling stage. Indeed, the changes are that instead of choosing the top thread in the ready_list, its mechanism would select the highest priority thread, which makes that thread context switch to run afterwards. We have implemented by considering a priority property for each thread. Whenever a new thread pushes back into the ready_list, it would compare with the current thread and check whether it be inserted in the ready_list or let it execute according to the priority comparison of those two threads.

By then, there is a problem when optimizing the scheduler through the priority comparison. For instance, thread with lower priority is difficult to have its turn to occupy the CPU and execute in the running state. When it comes to the priority donation, although some low-priority thread could be executed due to the donation from other high-priority thread to grab the lock over the low one, there would be no guarantee that some threads have this chance.

b. Solution

According to its problem description, Advanced Scheduler can be implemented for optimizing the existing scheduler, which is implemented as the Multi-level Feedback Queue Scheduling (MLFQS) as 4BSD. The mechanism of MLFQS is indeed based on priority but its priority is not given as in the previous part; however, it is based on four variables to rigorously calculate the priority for each thread.

In terms of the operation of MLFQS, there will be no priority donation and threads would not control their own priorities. This scheduling mechanism would have two main properties which are multiple ready queues represented for one priority number, and the priority of each thread is updated concurrently. In fact, there are 64 priority levels in Pintos from 0 to 63, thus, there are 64 ready queues of threads (each priority level has their own ready queue). Among one ready queue, FIFO method would also be used to select the thread based on its priority. Therefore, this would guarantee the higher priority will get executed first. As the priority is constantly altered after every 4 ticks with consideration of niceness and amount of CPU cycle used *(description below)*, no thread would be waiting for the CPU for too long. For instance, a thread with higher priority is executed first and after a certain amount of time, it uses in the CPU cycle. Thus, its priority will be decreased by the CPU usage rationale and the system switches to another thread with higher priority. This is different from the mechanism of (normal) scheduler with or without priority scheduling since the priority is always fixed, resulting in the issue that lower priority threads can hardly have its turn to be executed.

In the current naive Pintos, thread control by default is using the existing scheduler that has been previously implemented and optimized. However, if users want to use MLFQS in Advanced Scheduler, they run the '-mlfqs' option. When inputting the '-mlfqs' value when running pintos, the variable thread_mlfqs will be set to "true" and thus change the system to use advanced scheduler.

Explanation of four variables
Before getting to new improvement, we should re-explain and introduce four variables used interchangeably under the "New implementation" section. When it comes to the details of each parameter, under the thread-level, each thread consists of their own nice value, priority and recent_cpu. However, when it comes to load_avg, it belongs to the system level.

- nice (niceness value):
  - type: integer. It ranges from -20 to 20.
  - Represents how nice the thread is, more "niceness" equals more willingness to give up the thread to others, which means that increasing this value will decrease the thread's priority.

- priority:
  - Type: integer. It ranges from 0 to 63.
  - Represents the priority value of the thread. The higher the value means the higher priority and the default value when initialized is 31 and get recalculated after 4 ticks.

$$priority = PRIMAX - \left(\frac{recent\ cpu}{4}\right) - (nice * 2)$$

- recent_cpu:
  - Type: real number.
  - The default value is 0 and it represents how much of CPU cycle that a thread has been used. The more CPU cycle the thread has used, the lower priority it will have.

$$Recent\ cpu = \frac{(2 * loadavg)}{(2 * loadavg + 1)} * recent\ cpu + nice$$

  - Increase this value by 1 at every 1 tick when not on idle thread and recalculate the value by every 1 second.

- load_avg:
  - Type: real number. In general, this value shows the average number of threads that are possible to run at the last minute, which means that it represents how busy the system is. The more threads are waiting to be executed, the higher the value of load_avg, which implies the busier the system is.

$$load\ avg = \frac{59}{60} * load\ avg + \frac{1}{60} * ready\ threads.$$

- ready_threads: number of threads in running and ready state.
- It gets recalculated every second.

According to the Appendix B of the Project Description, since we can only perfom the integer arithmetic inside level (Kernel cannot support a floating point), we must implement a fixed-point arithmetic using integer arithmetic. The table below contains all the operations to implement in the arithmetic:

Convert n to fixed point:	n * f
Convert x to integer (rounding toward zero):	x / f
Convert x to integer (rounding to nearest):	(x + f / 2) / f if x >= 0, (x - f / 2) / f if x <= 0.
Add x and y:	x + y
Subtract y from x:	x - y
Add x and n:	x + n * f
Subtract n from x:	x - n * f
Multiply x by y:	((int64_t) x) * y / f
Multiply x by n:	x * n
Divide x by y:	((int64_t) x) * f / y
Divide x by n:	x / n

Figure 15: Table of fixed-point arithmetic

c. <u>New implementation</u>

Based on the above table, it is necessary to transform to the fix-point arithmetic for a better adaptation of Advanced Scheduler. Therefore, we would create a new file under the threads directory, which is called "fixed_point_arithmetic.h". To initialize, we have to define some variables. 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 in 17.14 format. Based on 2's complement convention, MAX would be 0 11111111111111111 11111111111111) and MIN would be 1 00000000000000000 00000000000000). Also, in this file, it contains 11 operation functions directly added such as:

Let x and y be two fixed-point numbers and n is an integer

- **int_to_fp(int n)**: Convert n to fixed point and return integer value
- **fp_to_int_zero(int x)**: Convert x to integer (rounding toward zero) and return integer value
- **fp_to_int_round(int x)**: Convert x to integer (rounding to nearest) and return integer value
- **add_two_fp(int x, int y)**: Add x and y (add two fixed_point numbers) and return integer value
- **sub_two_fp(int x, int y)**: Subtract x and y (subtract two fixed_point numbers) and return integer value
- **add_fp_int (int x, int n)**: Add x and n (add a fixed_point number and an integer) and return integer value

- **sub_fp_int (int x, int n)**: Subtract x and n (subtract a fixed_point number and an integer) and return integer value
- **mul_two_fp (int x, int y)**: Multiply x and y (multiply two fixed_point numbers) and return integer value
- **mul_fp_int (int x, int n)**: Multiply x and n (multiply a fixed_point number and an integer) and return integer value
- **div_two_fp (int x, int y)**: Divide x and y (divide two fixed_point numbers) and return integer value
- **div_fp_int (int x, int n)**: Divide x and n (divide a fixed_point number and an integer) and return integer value

- Data Structure added (previously explained above)
  - nice
  - recent_cpu
  - load_avg
  - Define related variable
    - PRI_MAX:63
    - PRI_MIN: 0
    - NICE_DEFAULT: 0
    - RECENT_CPU_DEFAULT: 0
    - LOAD_AVG_DEFAULT: 0

- Function modified
  - **src/threads/thread.c/struct thread**: In the thread structure, add new data structures, which is nice and recent_cpu
  - **src/threads/thread.c/init_thread()**: Initialize nice and recent_cpu with the default related variable (thread's nice would be a NICE_DEFAULT, thread's recent_cpu would be RECENT_CPU_DEFAULT)
  - **src/threads/thread.c/thread_start()**: Since load_avg is a variable related to the system-level, declare the load_avg as a global variable and thus add the load_avg that is assigned to LOAD_AVG_DEFAULT

- Function added

After initializing every required variable, we have to create six functions to satisfy the calculated idea as (i) the function calculates the priority used nice and recent_cpu (ii) the function calculates/update the recent_cpu + sub_function calculates load_avg (iii) the function calculates and updates load_avg (iv) the function increments the recent_cpu by 1 in every timer_interrupt (v) the function re-calculates the priority for all threads (vi) the function re-calculates recent_cpu and load_avg for all threads

  - **mlfqs_priority(thread *t)**: This function receives a thread pointer as an argument. If a function is an idle thread that its priority is fixed, it would not execute. Otherwise, it would calculate the priority of a specific thread using the fixed_point_arithmetic's operations. In every fourth tick, priority of all threads is recalculated. Therefore, its function would be:

$$priority = PRIMAX - \frac{recentcpu}{4} - nice * 2$$

- **mlfqs_recent_cpu (thread *t)**: This function receives a thread pointer as an argument. If a function is an idle thread, it would return. Otherwise, it would calculate the recent_cpu of a specific thread using the fixed_point_arithmetic's operation. In every second, it updates recent_cpu of each thread. Therefore, its function would be:

$$recentcpu = decay * recentcpu + nice$$

where:

$$decay = \frac{2 * loadavg}{2 * loadavg + 1}$$

Therefore, the calculation of recent_cpu would be:

$$recentcpu = \frac{2 * loadavg}{2 * loadavg + 1} * recentcpu + nice$$

- **mlfqs_load_avg ()**: This function is to calculate the load_avg in every second. Indeed, load_avg is a variable on the system-level and thus it would also compute in the case of idle thread is executed. This following equation is used for computing the load_avg in general case:

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

where ready_threads is the number of threads in the ready_list and threads in the executing at the time of update.
Thus, there are two cases that: If the current thread is an idle thread, ready_threads would be the number of threads in the ready_list. However, if the current one is an executable one, ready_threads is the number of threads in ready_list and also itself (+1).

- **mlfqs_recent_cpu_increment()**: When value changes, this function is served that the executing thread's recent_cpu is added to 1 in every clock tick. Hence, if the thread is executed in the running state, its recent_cpu would be incremented by 1 by using the operation of the fixed_point_arithmetic

$$recentcpu = (current\ thread)\ recentcpu + 1$$

- **mlfqs_priority_recompute()**: In every 4 ticks, it is compulsory to recompute the priority of all threads. During this implementation, use the for loop to iterate every thread in order to calculate their own priority through using mlfqs_priority() function.
- **mlfqs_recent_cpu_recompute()**: In every 1 second, we need to perform the recompute recent_cpu and load_avg of all threads. Its implementation is the same mechanism as mlfqs_priority_recompute() by using iteration of for loop.

- Function modified (continue):
  - **src/devices/timer.c/timer_interrupt()**: Before sleeping the thread to yield it in the sleeping_list, we have to check some conditions in terms of calculated priority in the MLFQS of Advanced Scheduler. To begin, in every timer interrupt, ticks are always incremented to 1. If "mlfqs" option is chosen, start to increment the recent_cpu of that thread through mlfqs_recent_cpu_increment(). Afterwards, in every 4 ticks (means that the number of ticks is divisible by 4), recompute the priority of that threads through mlfqs_priority_compute(). Otherwise, it does not recalculate again. Then, while it satisfied the condition of every 4 ticks,

in every second, recent_cpu and load_avg need to recompute through two functions mlfqs_recent_cput_recompute() and mlfqs_load_avg().

In the context of priority donation, the priority should be fixed in order not to intervene their procedure of donation. Therefore, while implementing the MLFQS of the Advanced Scheduler, it is mandatory to disable priority donation of lock_acquire(), lock_release(), and disable priority setting in thread_set_priority() when in the usage of MLFQS.

- **src/threads/synch.c/lock_acquire()**: use the variable thread_mlfqs to check whether using MLFQS option
- **src/threads/synch.c/lock_release()**: use the variable thread_mlfqs to check whether using MLFQS option
- **src/threads/thread.c/thread_set_priority()**: use the variable thread_mlfqs to check whether using MLFQS option

As in the advanced scheduler, nice, load_avg, and recent_cpu would be used to calculate the priority of all threads. Therefore, modifications for the not yet implemented functions below are necessary:

- **thread_set_nice()**: This function is to set the nice value to the current thread. During the value changing stage, it is needed to disable the interrupt not to let other external or internal interrupt intervene. While having the nice value, it is time to calculate its priority and then apply the condition (*) as described at the initial part in order to check whether its preemptive property. At the end, enable the interrupt again for getting back to the original state.

- **thread_get_nice()**: This function is to return the nice value of current thread. During this modification, disable and enable interrupt's mechanism is still maintained as the thread_set_nice() since we are now on the changing value stage. Then, assign the nice value from the current thread's nice value and return the nice value afterwards.

- **thread_get_load_avg()**: This function is to return the value of load_avg which is multiplied by 100. The mechanism of disable and enable the interrupt is the same as two above functions. However, to return the load_avg value, it is essential to multiple the (normal) load_avg by 100 with the mul_fp_int() function in the fixed_point_arithmetic's operation. Finally, return its load_avg value after calculating and transforming the fixed-point number into integer.

- **thread_get_recent_cpu()**: This function is to return the value of recent_cpu which is multiplied by 100. The mechanism of disable and enable the interrupt is the same as two above functions. However, to return the recent_cpu value, it is essential to multiple the recent_cpu of current thread by 100 with the mul_fp_int() function in the fixed_point_arithmetic's operation. Finally, return its recent_cpu value after calculating and transforming the fixed-point number into integer.