# Final Report

### 卢斓 11810935

## Introduction

In this projects, 5 files are modified for realizing task 1, 2 and 3 which are $thread.h$, $thread.c$, $synch.h$ and $synch.c$ in folder "threads" as well as $timer.c$ in folder "devices".

For task 1, files $thread.h$, $thread.c$ and $timer.c$ are modified. And for task 2, $thread.h$, $thread.c$ , $synch.h$ and $synch.c$ are changed while for task 3, $thread.h$, $thread.c$ and $timer.c$ are altered.

## Data structures and functions

### 1. $thread.h$

```
#define RE_TICK_DEFAULT 0
#define NICE_DEFAULT 0
fixed_t load_avg;
```

Define default remaining ticks, default nice value and variable load_avg.

```
struct thread  {
    int64_t remaining_ticks;
    struct lock *waiting_lock;
    int ori_priority;
    struct list locks;
    fixed_t recent_cpu;
    int nice;
}
```

Attributes are added in $struct\ thread$ to represent the number of ticks remained for sleeping, lock the thread is waiting for, original priority, list of all the locks held by the thread, recent CPU time and nice value respectively.

```
void thread_check_wake (struct thread *sleeping_t, void *aux UNUSED);

void thread_insert_for_lock(struct thread *holder);bool thread_priority_compare
(const struct list_elem *x, const struct list_elem *y, void *aux UNUSED);

void thread_update_priority(struct thread *current);

void thread_update_load_avg(void);

void thread_update_recent_cpu(void);

void thread_update_running_recent_cpu(void);
```

Define prototypes for functions in $timer.c$ added where the first one is for task 1 used in $timer.c$, the second and the third are for task 2 used in $synch.c$ while the remaining three are for task 3 used in $timer.c$.

## 2. $thread.c$

```
void thread_check_wake (struct thread *sleeping_t, void *aux UNUSED);
```

Wake up a sleeping(blocked) thread $sleeping\_t$ if the time for sleep is up.

```
void thread_insert_for_lock(struct thread *holder);
```

Re-insert a thread into ready list according to its priority to guarantee $ready\_list$ as an ordered list. Thread insertion is called in $synch.c$ and caused by priority changes related with priority donation.

```
void thread_update_priority(struct thread *current);
```

Update priority for a thread $current$ except the idle thread according to $priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$.

```
void thread_update_load_avg(void);
```

Update load_avg according to $load\_avg = (59/60) * load\_avg + (1/60) * ready\_threads$.

```
void thread_update_recent_cpu(void);
```

Update $recent_cpu$according to $recent\_cpu = (2 * load\_avg)/(2 * load\_avg + 1) * recent\_cpu + nice$

```
void thread_update_running_recent_cpu(void);
```

Update $recent\_cpu$ of the running thread, not idle, by adding one.

```
tid_t thread_create (const char *name, int priority, thread_func *function, void
*aux);
```

Add $thread\_yield$ if the priority of new thread is higher than the currently running thread to realize priority scheduler.

```
void thread_unblock (struct thread *t);
```

Implement $ready\_list$ as an ordered list using $list\_insert\_ordered$ to realize priority scheduler.

```
bool thread_priority_compare (const struct list_elem *x, const struct list_elem
*y, void *aux UNUSED);
```

Comparison function used for threads according to their $priority$.

```
void thread_yield (void);
```

Implement $ready\_list$ as an ordered list using $list\_insert\_ordered$ to realize priority scheduler.

```
void thread_set_priority (int new_priority);
```

Set the current thread's priority to NEW_PRIORITY. Modified to realize priority scheduler by using $thread\_yield$ and handle priority set during priority donation.

```
int thread_get_priority (void);
void thread_set_nice (int nice UNUSED);
int thread_get_nice (void);
int thread_get_load_avg (void);
int thread_get_recent_cpu (void);
```

Methods completed for task 3 with functionality being returning the current thread's priority, setting the current thread's nice value to NICE, returning the current thread's nice value, returning 100 times the system load average and returning 100 times the current thread's $recent\_cpu$ value.

```
static void init_thread (struct thread *t, const char *name, int priority);
```

Modified to initialize other new added attributes of a thread and implement $all\_list$ as an ordered list using $list\_insert\_ordered$ to realize priority scheduler.

## 3. $synch.h$

```
struct lock   {
    int max_priority;
    struct list_elem elem;
}
```

Attributes are added in $struct\ lock$ to represent the maximum priority among its candidates and owner as well as elem for it as a list_elem.

## 4. $synch.c$

```
void sema_down (struct semaphore *sema);
```

Implement $sema->waiters$ as an ordered list using $list\_insert\_ordered$ to realize priority scheduler when allocate a lock to its candidates.

```
void sema_up (struct semaphore *sema);
```

Use $list\_sort$ to guarantee $sema->waiters$ as an ordered list before choosing the thread corresponding to the front element as the one to unblock.

```
void lock_init (struct lock *lock);
```

Modified to initialize other new added attributes of a lock.

```
void lock_acquire (struct lock *lock);
```

Acquire for a lock which is not held by the current thread. Update related locks' and threads' attributes due to priority donation if there is any as well as the status of $SEMA$. Finally, the thread gets this lock.

```
int lock_update_max_priority(struct lock *lock, struct thread *t);
```

Check if there is any new priority donation and update related information especially the $priority$ of threads.

```
void lock_release (struct lock *lock);
```

Release a lock held by the current thread and reset the attributes of this lock and its previous holder, especially recalculating the thread's priority.

```
bool lock_priority_compare (const struct list_elem *x, const struct list_elem
*y, void *aux UNUSED);
```

Comparison function used for locks according to their $max\_priority$.

```
bool cond_priority_compare (const struct list_elem *x, const struct list_elem
*y, void *aux UNUSED);
```

Comparison function used for semaphore_elem according to their $priority$.

```
void cond_signal (struct condition *cond, struct lock *lock UNUSED);
```

Use $list\_sort$ to guarantee $cond->waiters$ as an ordered list before $sema\_up$.

### 5. $timer.c$

```
void timer_sleep (int64_t ticks);
```

Let the thread sleep for approximately TICKS timer ticks and keeps interrupts being turned on. Turn sleep by $thread\_yield$ to sleep by $thread\_block$.

```
static void timer_interrupt (struct intr_frame *args UNUSED);
```

Timer interrupt handler. Modified to check and wake(if time for sleep is up) a sleeping thread and at the same time update $load\_avg$, $recent\_cpu$ and $priority$ for threads.

## Algorithms

### Task 1

When calling $timer\_sleep$, instead of calling $thread\_yield$ , the thread is directly blocked by $thread\_block$ while we use an attribute $remaining\_ticks$ of $thread$  to record how much time remaining for the thread to sleep. Particularly, $remaining\_ticks$ is a new attribute added to the $thread$ structure and  and is set to be $ticks$ when calling $timer\_sleep(ticks)$ to the thread. Then $Pintos'$ own clock interrupt $timer\_interrupt$ which will be executed once per tick is used to detect the status of the thread by calling $thread\_check\_wake$ . Each time $remaining\_ticks$ is decreased by 1 and if it is decreased to 0, the thread will be waken up by $thread\_unblock$.

## Task 2

### Choosing the next thread to run

We choose thread with the highest priority in the $ready\_list$ as the next thread to run. To realize this, we maintain $ready\_list$ as an ordered list according to the threads' priority from the highest to the lowest. In this way, when calling $next\_thread\_to\_run$ in $schedule$, the thread with the highest priority will be returned as the next thread to run.

Besides, to guarantee $ready\_list$ as an ordered list all the time, we will use $list\_remove$ and $list\_insert\_ordered$ to make the thread whose priority changes because of priority donation still in a correct position in $ready\_list$.

Specifically, after a new thread is created, a thread is unblocked or the priority of the thread is changed, check the current conditions and call $thread\_yield$ to let the thread with the highest priority run first to realize priority scheduling.

### Acquiring a lock

A thread can use $lock\_acquire$ to asking for a lock which is not belong to it at this time.

If the lock currently has a holder, then $lock\_update\_max\_priority$ is called iteratively to update threads' $priority$ caused by priority donation. Specifically, it recalculates the efficient priority of the related threads, and use $list\_remove$ and $list\_insert\_ordered$ to move the thread in $ready\_list$ and lock in its holder's $locks$ to right places and thus guarantee $ready\_list$ and $locks$ as ordered lists.

Then $sema\_down$ is called to do the $P$ operation on a semaphore. It waits for $SEMA's$ value by calling $thread\_block$ if the current $SEMA's$ value is not positive, and then atomically decrements it. If the thread is blocked, it can be waken up if and only if $lock\_release$ is called to release a lock and this thread is judged by $sema\_up$ to be the highest priority one among those waiting for $SEMA$.

After all above, the lock is allocated to the thread with status of lock and its holder thread updated.

### Releasing a lock

Lock held by the current thread can be released by calling $lock\_release$.

When a lock is released, the attributes of this lock and its previous holder should be reset. Particularly, for its previous holder, the effective priority is recalculated. Detailed description is given in the next part.

### Computing the effective priority

Effective priority will need to be computed when there occurs a priority donation in $lock\_acquire$ and when the priority donation needs to be updated because of $lock\_release$.

In $lock\_acquire$, $lock\_update\_max\_priority$ is called iteratively to check if there is any new priority donation and update $priority$ for the related threads. While in $lock\_release$, the effective priority of the lock's previous holder is recomputed by choosing the largest one among the thread's $ori\_priority$ and $max\_priority$ of all the locks still held by this thread as its new $priority$. Specifically the lock with highest $max\_priority$ is the front element in the thread's $locks$ since $locks$ is maintained as an ordered list.

### Priority Scheduling for semaphores and locks

As for the priority Scheduling for semaphores, I use $list\_sort$ in $sema\_up$ and $list\_insert\_ordered$ in $sema\_down$ to guarantee $sema->waiters$ as an ordered list before choosing the thread corresponding to the front element as the one to unblock.

In this way, the priority scheduling for locks is also realized since a lock is allocated to a thread through $sema\_down$ and $sema\_up$.

### Priority scheduling for condition variables

To implement priority scheduling for condition variables, I use $list\_sort$ in $cond\_signal$ before calling $sema\_up$ to guarantee $cond->waiters$ as an ordered list according to the priority of threads from the highest the the lowest.

### Changing thread's priority

Priority of a thread can be changed through calling of $thread\_set\_priority$. To handle problems raised because of priority donation, it sets $ori\_priority$ of a thread to be the $new\_priority$ and changes $priority$ to $new\_priority$ only if the attribute $locks$ of this thread is empty or the $new\_priority$ is larger than $priority$. After changing the $priority$ of this thread, call method $thread\_yield$ to realize priority scheduling.

## Task 3

In $Pintos'$ own clock interrupt $timer\_interrupt$ which will be executed once per tick, $thread\_update\_running\_recent\_cpu$ is called to add the $recent\_cpu$ of the running thread by 1 per tick. Besides, for every second, $thread\_update\_load\_avg$ and $thread\_update\_recent\_cpu$ are performed to update the $load\_avg$ and $recent\_cpu$ for threads. What needs to be emphasized there is, due to the dependency relationships of $load\_avg$, $nice$, $recent\_cpu$ and $priority$, $thread\_update\_load\_avg$ should be performed ahead of $thread\_update\_recent\_cpu$ and in method $thread\_update\_recent\_cpu$, $thread\_update\_priority$ should be called after updating the $recent\_cpu$ for every threads, except idle thread, to update threads' $priority$. What's more, $thread\_update\_priority$ should be called every fourth tick for only the current thread since $recent\_cpu$ for only the current thread changes.

In $thread\_set\_nice$, the nice value of the current thread may be changed and will also need $thread\_update\_priority$ to update the priority. In this case, $thread\_yield$ should also be added to guarantee a priority scheduling after this setting.

The criteria for updating are as follows:

1. $Priority$ for every thread except idle thread:

   $priority\ =\ PRI\_MAX\ -\ (recent\_cpu\ /\ 4)\ -\ (nice\ *\ 2)$ for every fourth tick

2. $Recent\_cpu$ for every thread:

   $recent\_cpu\ =\ recent\_cpu\ +\ 1$ for every tick and only for the running thread

   $recent\_cpu\ =\ (2*load\_avg)/(2*load\_avg\ +\ 1)\ *\ recent\_cpu\ +\ nice$ for every second

3. $Load\_avg$:

   $load\_avg\ =\ (59/60)*load\_avg\ +\ (1/60)*ready\_threads$ for every second

# Synchronization

## Task 1

To guarantee synchronization, we could disable the interrupts for setting the $remaining\_ticks$ and calling $thread\_block$ in $timer\_sleep$. Interrupts are also disabled in $thread\_unblock$.

### Task 2

In $lock\_acquire$, interrupts are disabled when $lock\_update\_max\_priority$ is called iteratively to update threads' $priority$ caused by priority donation, when using $sema\_down$ to do the $P$ operation on a semaphore and when the lock is allocated to the thread with status of lock and its holder updated. Also, in $lock\_release$, $thread\_yield$ and other methods which make changes to a group of correlated objects and may interact with the shared objects such as $ready\_list$, interrupts are disabled while updating the status. Through this, we can guarantee a consistent state for synchronization.

### Task 3

For synchronization, interrupts are disabled when $priority$ is recomputed to guarantee the synchronization and coherent of data.

## Rationale

### Task 1

Compared with the original implementation, my design can save the CPU resources which are wasted because of the busy waiting in the original one. In other words, sleeping threads will not be put into the $ready\_list$ in my design.

Besides, I have also considered using an ordered list to hold all the sleeping threads by their $remaining\_ticks$, however since it is more complex and will cause additional memory cost and need more time($O(n)$ time) to insert a new element, it is not as good as my final design.

### Task 2

Compared with the original implementation, my design realized priority scheduling and solved problems caused by priority donation.

Secondly, to guarantee priority scheduling, I designed two approaches for this. The first is to maintain $ready\_list$ as an ordered list all time which requires to change the position of a thread by $list\_remove$ and $list\_insert\_ordered$ in $ready\_list$ when its $priority$ changes because of priority donation, which is used in my final answer. The second is to use $list\_sort$ on $ready\_list$ when we have to get a thread from it as the next thread to run. However, the second one is more costly since every time we use schedule to get the next thread, we call $list\_sort$ while the first one only updates $ready\_list$ when priority donation occurs. Besides, the codes need to be updated is also less in the former approach.

### Task 3

The $MLFQS$ is implemented in this task using fixed point real numbers which is already supported by the $Pintos$. Originally, besides $thread\_update\_priority(struct\ thread\ *t)$, I plan to write another method $thread\_update\_priority(void)$ to visit $all\_list$ and update the priority for all threads which is called in $timer\_interrupt$ every fourth tick. However, later I find that the functionality of the later method can be realized with $thread\_update\_priority(struct\ thread\ *t)$ inserted into $thread\_update\_recent\_cpu$, which is executed once per second for every threads. And during this second, $recent\_cpu$ for only the running thread will change and therefore we only need to call

$thread\_update\_priority(thread\_current())$ every fourth tick. In this way, the complexity for code modification is simplified and time is also saved.

## Design Document Additional Questions

| Initial | A | B | C |
|---|---|---|---|
| nice | 0 | 1 | 2 |
| recent_cpu | 0.00 | 0.00 | 0.00 |

Answers:

$R$ represents $recent\_cpu$ and $P$ represents $priority$.

The $TIMER\_FREQ$ in my program is 100 which means that there are 100 ticks in one second and $TIME\_SLICE$ is 4. Besides, the $load\_avg$ is initialized to 0.

| timer ticks | R(A) | R(B) | R(C) | P(A) | P(B) | P(C) | thread to run |
|---|---|---|---|---|---|---|---|
| 0 | 0.00 | 0.00 | 0.00 | 63 | 61 | 59 | A |
| 4 | 4.00 | 0.00 | 0.00 | 62 | 61 | 59 | A |
| 8 | 8.00 | 0.00 | 0.00 | 61 | 61 | 59 | B |
| 12 | 8.00 | 4.00 | 0.00 | 61 | 60 | 59 | A |
| 16 | 12.00 | 4.00 | 0.00 | 60 | 60 | 59 | B |
| 20 | 12.00 | 8.00 | 0.00 | 60 | 59 | 59 | A |
| 24 | 16.00 | 8.00 | 0.00 | 59 | 59 | 59 | C |
| 28 | 16.00 | 8.00 | 4.00 | 59 | 59 | 58 | B |
| 32 | 16.00 | 12.00 | 0.00 | 59 | 58 | 58 | A |
| 36 | 20.00 | 12.00 | 4.00 | 58 | 58 | 58 | C |

Indeed, there are some ambiguities when the priority of some threads happen to be the same.

However, this can be solved through the following rules. Namely let the thread which enters $ready\_list$ earlier execute first which can be implemented by writing a suitable comparison function for $list\_insert\_ordered$ to put the later object with the same comparison value behind the earlier object.