# Project 1: Threads

## 11510602 倪犀子

## Design Document

### Task 1 Design

#### Task 1: Key Idea

In order to sleep a thread without busy waiting, I implement a priority queue (using the wake up time as key, in acending order) for sleeping priority queue. When the thread need to sleep, I calcualte the time tick that it need to wake up and put it to the sleeping priority queue. For every time tick, I search the priority queue, if any thread should be waked up, I wake it up and put it into ready list.

#### Task 1: Data Structure and Functions

1. I changed the thread structure.
    - Add `wake_up_ticks` to store the time tick that the sleeping thread needed to be waked up.
    - Add `blocked_elem` to put the sleeping thread to the priority queue.

```
struct thread
{
/* Owned by thread.c. */
tid_t tid;                          /* Thread identifier. */
enum thread_status status;          /* Thread state. */
char name[16];                      /* Name (for debugging purposes). */
uint8_t *stack;                     /* Saved stack pointer. */
int priority;                       /* Priority. */

// Newly Added
int wake_up_ticks;                  /* The time ticks that the thread
should wake up*/
struct list_elem blocked_elem; /* List element for sleeping priority
queue*/

struct list_elem allelem;           /* List element for all threads list.
*/
/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */

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

/* Owned by thread.c. */
```

```
unsigned magic; /* Detects stack overflow. */
};
```

2. Add some new function to the thread.c file and a static variable

    - Add `add_to_blocked_list` for adding sleeping thread to sleeping priority queue.
    - Add `wake_up_thread` to wake up all the thread in the priority queue that time up for sleeping.
    - Add `blocked_foreach` in order to search the time up thread in the priority queue.
    - `wake_up` is the function that check whether the thread times up, if do, wake it up.
    - `blocked_list` the list to store all the sleeping variable.
    - `blocked_ticks_less_func` use this function to implement priority queue.

```
void add_to_blocked_list(struct thread *blocked_thread);
void wake_up_thread(int cur_ticks);
void blocked_foreach(int64_t cur_ticks);
void wake_up(struct thread *t, int64_t cur_ticks);
static struct list blocked_list; /* List of all the blocked process.
Sort in acending order.*/
static bool blocked_ticks_less_func(const struct list_elem *a, const
struct list_elem *b, void *aux UNUSED);
```

**Task 1: Algorithms**

To sleep a thread without busy waiting:

- Disable interrupt
- Calculate the time tick the thread should wake up
- Adding the thread to the sleeping priority queue (`add_to_blocked_list`)
- Enable interrupt

Wake up the thread when the times up:

- For every `timer_interrupt`
- Search the sleeping priority queue (`blocked_foreach`)
- When the sleeping times up, wake it up (`wake_up`)

**Task 1: Synchronization**

Since the `blocked_list` is a global variable. I disable the interrupt when puting the sleeping thread to the `blocked_list`, and enable the interrupt when finished.

**Task 1: Rationale**

Because I implement `blocked_list` as a priority queue, for every time tick, I do not have to search all the element in the list. Therefore the time for searching and waking up all the thread is less.

## Task 2 & 3 Design

### Task 2 & 3: Key Idea

1. Priority Schedular.

   Rewrite the ready list, change it to the priority queue (using the priority of the thread as key, in desending order). Therefore when the schedular need to know waht is next to run, it can simply pop the head of the queue.

2. Modified three synchronization primitives (lock, semaphore, condition variable).

   For semaphore, I change the waiter list to a priority queue (using the priority of thread as key, in decending order). When `sema_up`, it wake up the thread that have the highest priority. Same as semaphore, I change the waiter list to a priority queue. Since lock is based on semaphore, the lock can be modified when the semaphore are modified correctly.

3. Priority Donation.

   When a lock have already acquired by a thread, when a new thread wants to acquired it, I will examine whether it have higher priority than the holder thread priority. If it have higehr priority, it increase the holder's priority to its priority. When the holder release the lock, it must decrease its priority to its original priority. (This is the original design, it have beed changed. The new version describe in the Task 2 & 3 Implementation)

### Task 2 & 3: Data Structure and Functions

1. Thread

   - Add `donated_status` and `original_priority` to identified whether the thread is in donated mode and its original priority.
   - Add `time_slice` to record the time tick of time slice.

```
struct thread
{
/* Owned by thread.c. */
tid_t tid;                          /* Thread identifier. */
enum thread_status status;          /* Thread state. */
char name[16];                      /* Name (for debugging purposes). */
uint8_t *stack;                     /* Saved stack pointer. */
int priority;                       /* Priority. */
bool donated_status;                /* Thread that donote the priority to
current holder */
int original_priority;              /* The amount of priority that others
give */
int unsigned time_slice;            /* Time Slice*/
int wake_up_ticks;                  /* The time ticks that the thread
should wake up*/
struct list_elem allelem;           /* List element for all threads list.
*/
struct list_elem blocked_elem; /* List element for blocked list*/
/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */
```

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

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

2. Semaphore

   Add `priority` to make condition prefer higher priority.

```
struct semaphore
{
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
    int priority;               /* Priority. */
};
```

3. Lock

   Add donator to identified the donator.

```
struct lock
{
    struct thread *holder;        /* Thread holding lock (for
debugging). */
    struct semaphore semaphore;   /* Binary semaphore controlling
access. */
    struct thread *donator;       /* Thread that donote the priority
to current holder */
};
```

4. Function

   - `release_donate` is used for decrease the priority of holder when the lock released.
   - In order to implement donate machenism, `donate` function can increase the holder's priority.
   - `ready_thread_more_func` and `con_more_func` are used to implement priority queue.

```
void release_donate(struct lock *lock);
void donate(struct lock *lock, struct thread *donator);
static bool ready_thread_more_func(const struct list_elem *a, const
struct list_elem *b, void *aux UNUSED);
static bool con_more_func(const struct list_elem *a, const struct
list_elem *b, void *aux UNUSED);
```

**Task 2 & 3: Algorithms**

1. Priority Schedular.
    - Every time the thread add to `ready_list` (implement as priority queue), it should insert using `list_insert_ordered` function.
    - The `next_thread_to_run` will return the head of the `ready_list`, which is the thread with highest priority.
2. Modified three synchronization primitives (lock, semaphore, condition variable).
    - Disable interrupt
    - Push the waiting thread into the `waiters` (implement as priority queue), using `list_insert_ordered` function.
    - Enable interrupt
    - When call `sema_up`, the `waiters` will return the head, which is the thread in list with highest priority.
3. Priority Donation.
    - `lock_acquire()`
        - If the lock have holder and the thread priority is higher than the holder, it will call the `donate`
        - else, as usual.
    - `lock_release()`
        - If the lock have `donator` is not NULL, it should decrease the priority of the holder to the original one.
        - else, as usual.

**Task 2 & 3: Synchronization**

The static and global variable will be protected by disabling interrupt.

**Task 2 & 3: Rationale**

The lists are all implement as priority queue, therefore it have higher efficiency.

# Final Implementation

The final implementation is roughly same as the initial design, however, the implementation of task 2 & 3 is very different from the original design. Especially in the priority donation.

## Task 1 Implementation

The final implementation is same as the original design. It have been describe clearly in the design document. The key idea is that I implement a priority queue (the time tick to wake up as key, in acending order) to store the sleeping thread. And in every time tick, I wake up the thread that can be waked up.

See the Task 1 Design for the details.

## Task 2 & 3 Implementation

The implementation of priority schdular, semaphore, lock and conditional varible are same as the design document. For the details can refer to Task 2 & 3 Design.

However, it is very naive to design the donation logic like the original design. I did not consider the case that the lock will be acquire by multiply thread and the case that holder of lock is lock by another thread. Therefore, I change the design of donation.

1. Thread
    - Add `lock_list` to record the lock that the donator have been donate its priority. It have been implemented as priority queue (the priority of the lock holder as the key, in desending order).
    - Add `donate_list` to record the lock that the thread have donated its priority.

```
struct thread
{
/* Owned by thread.c. */
tid_t tid;                          /* Thread identifier. */
enum thread_status status;          /* Thread state. */
char name[16];                      /* Name (for debugging purposes). */
uint8_t *stack;                     /* Saved stack pointer. */
int priority;                       /* Priority. */
bool donated_status;                /* Thread that donote the priority to
current holder */
int original_priority;              /* The amount of priority that others
give */
int unsigned time_slice;            /* Time Slice*/
int wake_up_ticks;                  /* The time ticks that the thread
should wake up*/
struct list_elem allelem;           /* List element for all threads list.
*/
struct list_elem blocked_elem; /* List element for blocked list*/
/* Shared between thread.c and synch.c. */
struct list_elem elem; /* List element. */

struct list lock_list;
struct list donate_list;

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

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

2. lock

- `lock_elem` and `donate_elem` are the list elemment to implement the `lock_list` and `donate_list`.

```
struct lock
{
    struct thread *holder;        /* Thread holding lock (for
debugging). */
    struct semaphore semaphore;   /* Binary semaphore controlling
access. */
    struct thread *donator;       /* Thread that donote the priority
to current holder */
    struct list_elem lock_elem;   /* List element for blocked list*/
    struct list_elem donate_elem; /* List element for blocked list*/
};
```

3. New Functions

- Add `recursive_raise` to implement the case that A (Highest Priority) is locked by B (Medium Priority) and B is locked by C (Lowest Priority).
- `lock_more_func` is for the implementation for `lock_list` (Priority queue).

```
void recursive_raise(struct thread *donatee, int new_priority);
static bool lock_more_func(const struct list_elem *a, const struct
list_elem *b, void *aux UNUSED);
```

4. Detail of `donate()`

- Raise exception when the priority of donator is lower than the holder
- If the `lock_list` of the donatee (lock holder) is empty, simply change the status of the donatee thread, raise the priority and change the donator of the lock.
- If the `lock_list` is not empty, change the donator of the lock.
- Add the lock into `lock_list` by using the `list_insert_ordered`.
- Add the lock into `donator_list`.
- Run `recursive_raise`. Raise the priority of the donatee's donatee.

5. Detail of `release_donate()`

- Remove the lock from the `lock_list` and `donator_list`.
- Change the status and the priority of the lock holder.
- If the `lock_list` of the donatee (lock holder) is not empty, change the priority and the status of donatee, by simply see the head of the `lock_list`.

6. Detail of `recursive_raise()`

- Iterate the `donator_list` of a thread.
- If the priority of the lock holder is lower than the newly raised priority. Raise the priority of the lock holder.

## What can be done to improve the project

- I not sure the efficience of the final implementation whether is high enough. If not, maybe I can redesign the algorithms.
- After testing there are not obvious race condition appear. More test is needed to test whether the race condition is exist.