Final report

SID: 11812318 Name: 张嘉兮

Task 1 Efficient Alarm Clock

Dara structures and functions

- *thread.h*:
  - Add field `blocking_tick` into *struct thread*. The field will record the ticks this thread will be blocking, and then unblock.

    ```
    int64_t blocking_tick;              /* The ticks this thread will be blocking. */
    ```

  - Add function ***thread_blocking_check*** to check `blocking_tick` for each thread and check whether unblock it.

    ```
    void thread_blocking_check (struct thread *, void *);
    ```

Algorithms

This task is to avoid the 'busy waiting' in `timer_sleep`, where the thread will constantly check whether sleep time is finish.

So I use a 'sleep-being woken up' mechanism to avoid the loop for checking.

1. Assume thread **t** calls the ***timer_sleep(n)*** with **n** time ticks.
2. Then I will block the thread **t** and set **n** to its field `blocking_tick`.
3. In order to wake up **t**, I modify the function ***timer_interrupt***, where I call ***thread_blocking_check*** for all thread.
4. And in the ***thread_blocking_check***, if the `status` of thread input is *BLOCKING* and the `blocking_tick` is greater than 0, then the `blocking_tick` will be minus by 1.
5. And if the `blocking_tick` reaches 0, then I will unblock the thread input.
6. So the `blocking_tick` of **t** will continuous minus by 1 until reach 0 (for **n** time ticks), when is the time for waking it up. And **t** will be unblocked.

Synchronization

Potential concurrent accesses to shared resources

- The accesses to field `blocking_tick`, which will be accessed by function ***timer_sleep*** and function ***thread_blocking_check***. And the ***thread_blocking_check*** will be called every ***timer_interrupt***.
- The call of ***thread_block*** and ***thread_unblock***, which will access some shared resources including `status` of thread, the `ready_list` and etc.

Strategy

- I use the code described below to guarantee the access of field `blocking_tick` is atomic, unable to be interrupted.

  ```
  enum intr_level old_level = intr_disable ();
  /* Codes */
  intr_set_level (old_level);
  ```

  Then the "codes" inside will exclusive access the resources.

- Function ***thread_block*** and ***thread_unblock*** has been guaranteed the correct synchronization by pintos using the same mechanism.

Rationale

- Shortcomings: If there are too many threads, then the execution of ***timer_interrupt*** will be time-consuming.
- Time complexity: O(n) for each execution of ***timer_interrupt*** where n is the number of threads in `all_list`. As each execution of ***timer_interrupt*** will check the threads one by one, and the operation of check for one thread is O(1).
- Space complexity: O(n) where n is the number of threads in `all_list`. As we use a field `blocking_tick` for each thread to save the number of ticks it may be blocking.

Task 2 Priority Scheduler

Data structures and functions

- *thread.h*:
  - Add field `lock_waiting_for` in *struct thread*, for saving the lock that thread acquire for but occupied.

    ```
    struct lock * lock_waiting_for;      /* The lock this thread is waiting for. */
    ```

- Add field `locks_holding` in *struct thread*, for saving a list of locks that thread hold.

```
struct list locks_holding;          /* The locks this thread is holding. */
```

- Add field `undonated_priority` in *struct thread*, for saving the priority set.

```
int undonated_priority;             /* The origin priority of the thread. */
```

- Add function ***thread_donate_priority*** for a thread donating its priority for a hold lock recursively.

```
void thread_donate_priority (struct thread *,struct thread*);
```

- Add function ***thread_check_priority*** for a thread to calculate the real priority it should have after donation.

```
void thread_check_priority (struct thread *);
```

- Add function ***thread_priority_comparator*** as a ***list_less_func*** that compare two *threads* by the `priority` of the thread.

```
bool thread_priority_comparator (const struct list_elem *, const struct list_elem *, void *);
```

* *synch.h*
    - Add field `elem` into *struct lock* let *struct lock* becoming a type of list element.

    ```
    struct list_elem elem;      /* For used in list as lock can only held by one thread. */
    ```

    - Add function ***lock_priority_comparator*** as a ***list_less_func*** that compare two *locks* by the max priority of threads waiting for the lock.

    ```
    bool lock_priority_comparator (const struct list_elem *, const struct list_elem *, void *);
    ```

    - Add function ***locks_max_priority*** to find the maximal priority of *threads* in the waiting list of any *lock* which is in the list of locks.

    ```
    int locks_max_priority (const struct list *);
    ```

    - Add function ***cond_sema_priority_comparator*** as a ***list_less_func*** that compare two *conditions* by the max priority of threads waiting for the *semaphore* that *condition* has.

```
bool cond_sema_priority_comparator(const struct list_elem
*,const struct list_elem *, void *);
```

Algorithms

This task is to implement the priority schedule and priority donation.

In this part, I will separately introduce my design for priority schedule and priority donation.

Priority schedule:

1. For <u>**next_thread_to_run**</u>, I modify ***next_thread_to_run*** as using ***list_max*** with ***thread_priority_comparator*** to get the *thread* that has maximal `priority` in the ready_list, and return this *thread*. As ***schedule*** will call ***next_thread_to_run*** to find the next thread should run, I have guarantee ***schedule*** will choose the *thread* has maximal `priority`.
2. For <u>***semaphore***</u>, I modify ***sema_up*** as using ***list_max*** with ***thread_priority_comparator*** to get the *thread* that has maximal `priority` in the `waiters` of the *semaphore*, and ***unblock*** this *thread*.
3. For <u>***condition***</u>, I modify ***cond_signal*** as using ***list_max*** with ***cond_sema_priority_comparator*** to get the *semaphore_elem* which has the *thread* with maximal `priority` in <u>all</u> *semaphore*'s `waiters` from the `waiters` of the *condition*, and then ***sema_up*** the *semaphore* I selected.
4. For <u>***lock***</u>, as I have modified the ***sema_up***, and ***lock_release*** call ***sema_up***, so I have implemented this.

Priority donation:

1. When <u>**acquire**</u> <u>***lock***</u>, I need to check whether there is a holder for this lock, if so, then I need try to donate priority recursively along the lock requirement chain. I implement this by ***thread_donate_priority***. Otherwise, it will execute the ***sema_down***.

2. ***thread_donate_priority***:

    1. Let we call the *lock*'s `holder` as **l_holder**. I will call ***thread_check_priority*** on **l_holder** to find the correct priority after donation.
    2. ***thread_check_priority***: I will search all the `waiters` of the `semaphore` of the `locks_holding` (a list containing locks holding by **l_holder**) of the **l_holder**, and find the maximal priority all the threads in `waiters` have. Let we call it **max_pri**. Then I will let the `priority` of the **l_holder** become the maximal one between **l_holder**'s `priority` and **max_pri**.
    3. Then **l_holder**'s `priority` has been donated. Finally, if **l_holder** is requiring a *lock* **l2**, then I will recursively call ***thread_donate_priority*** from **l_holder** to **l2**'s `holder`. Recusing like this until a *lock*'s `holder` no longer has the requested lock.
3. After priority donation, it will call ***sema_down***.

4. After **sema_down**, it get the *lock* and become the `holder` of the *lock*, then I need to add the *lock* into the `locks_holding` of the current thread.
5. When **release** ***lock***, I need to remove the *lock* from `locks_holding` of the current thread. Then I call ***thread_check_priority*** to let the `undonated_priority` become the `priority` of the current thread.
6. When **changing thread's priority**, I will change the `undonated_priority` of the thread, then call ***thread_check_priority*** to evaluate the correct `priority` after donation. Finally, call the ***thread_yield*** to do a schedule.

Synchronization

Potential concurrent accesses to shared resources

- Operation of the `locks_holding` of *struct thread* as a *lock* may be required concurrently.
- Modification of the `priority` of a thread (can be modified by itself and other thread by donating, even other threads may concurrently donate to the same thread).
- Original operations about *semaphore*, *condition*, *lock* and other sync objects.
- `lock_waiting_for`, `undonated_priority` are safe because they can only modified by the thread itself.

Strategy

- I use the code described below to guarantee the access of the fields in 'Codes' is atomic, unable to be interrupted.

  ```
  enum intr_level old_level = intr_disable ();
  /* Codes */
  intr_set_level (old_level);
  ```

  Then the "codes" inside will exclusive access the resources.
- I assume the original operations about sync objects are safe implementation by pintos. In fact, the implementation is the same as I mentioned above.

Rationale

Advantages

My implementation is easy to maintain the `priority` of a *thread*, because it can be calculated easily by call ***thread_check_priority***.

Shortcomings

The dynamic computing mode for priority donation may take more time in especial the number of threads is large.

The priority schedule can be improved by a heap, which will let the complexity reduced to O(log**n**).

Time complexity

- O(**n**) for **next_thread_to_run**, ***semaphore***, ***condition***, ***lock*** finding the next thread to switch to or to take over the sync object, where **n** is the number of threads in `all_list`.
- O(**n**) for priority donation, as one thread can only require for one lock in a time. Then the complexity to find the maximal priority of donators is O(**n**), where n is the number of threads in `all_list`.

Space complexity

- O(**n+m**) for the field `locks_holding` and `undonated_priority` in *struct thread* and field `elem` in *struct lock*, where **n** is the number of threads in `all_list` as each thread has a `locks_holding` and **m** is the number of exist locks as each *lock* has a `elem`.

Task 3 Multi-level Feedback Queue Scheduler (MLFQS)

Data structures and functions

- *thread.h*:
  - Add field `nice` into *struct thread* to save the value of 'nice' in the formula.

    ```
    int nice;                          /* Nice value   for
    mlfqs. */
    ```

  - Add field `recent_cpu` into *struct thread* to save the value of 'recent_cpu' in the formula.

    ```
    fixed_t recent_cpu;                /* The recent   cpu for
    mlfqs. */
    ```

  - Add function ***threads_update_mlfqs*** for ***timer_interrupt*** to call to deal with the updating of values about mlfqs.

    ```
    void threads_update_mlfqs (int, int);
    ```

  - Add function ***thread_update_recent_cpu_pri*** for updating the `recent_cpu` and `priority` for a thread.

    ```
    void thread_update_recent_cpu_pri (struct thread *, void *);
    ```

  - Add function ***thread_update_pri*** for updating the `priority` for a thread.

```c
void thread_update_pri (struct thread *, void *);
```

- *thread.c*:
  - Add static variable `load_avg` for 'load_average' in formula.

```c
/* The load_average for mlfqs. */
static fixed_t load_avg = FP_CONST (0);
```

Algorithms

In this task, I just implement the formulas given to me.

1. I call ***thread_update_mlfqs*** in ***timer_interrupt***.
2. In ***thread_update_mlfqs***, I check that the `thread_mlfqs` is enabled, and then I increase the `recent_cpu` of the current thread by 1.
3. Then if the time ticks is 4's multiples, I update the priority of all thread by
$$priority = PRI\_MAX - (recent\_cpu/4) - (nice \times 2)$$
4. Then if the time ticks is TIMER_FREQ's multiples, I update the `load_avg` by
$$load\_avg = (59/60) \times load\_avg + (1/60) \times ready\_threads$$
And update the `recent_cpu` by
$$recent\_cpu = (2 \times load\_avg)/(2 \times load\_avg + 1) \times recent\_cpu + nice$$

The priority of threads changed in time increasing. The running thread having the highest priority will get increasing `recent_cpu` more, then its `priority` will decrease. Therefore, MLFQS guarantees the bounded-waiting and no starvation.

Synchronization

Potential concurrent accesses to shared resources

- the `priority` of a thread, as it can be modified by both ***timer_interrupt*** and ***thread_set_nice***, where may occur synchronization problem.

Strategy

- I use the code described below to guarantee the access of the fields in 'Codes' is atomic, unable to be interrupted.

```c
enum intr_level old_level = intr_disable ();
/* Codes */
intr_set_level (old_level);
```

  Then the "codes" inside will exclusive access the resources.

Rationale

Time complexity

- O(**n**) where **n** is the number of threads in `all_list`. As each thread should be update.

Space complexity

- O(**n**) where **n** is the number of threads in `all_list`. As each thread should save the field `recent_cpu` and `nice`.

3.1.2

timer ticks	R(A)	R(B)	R(C)	P(A)	P(B)	P(C)	thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	8	4	0	61	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	4	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

Yes, there are ambiguities make value uncertain as we can implement the selecting of thread among threads having same priority in different way.

According to my codes, I choose the thread enqueued firstly when there are some threads have the same priority.

And if I change the strategy for choosing the thread among threads having same priority, the result will be different.