Operating System Project 2

Team #2

20165167 정현준. 20185125 이상범

Problem 1. Implementing Alarm Clock

1. Problem definition

Problem 1 은 busy waiting 을 없애기 위해 timer_sleep() 함수를 재구현하는 것이 목적이다. 기존 Pintos 에 프로그래밍된 timer_sleep()을 실행한다면 sleep 시킨 thread 가 loop 을 통해 CPU 를 계속 점유하는 busy waiting 이 발생한다. Busy waiting 이 발생할 경우 대기해야 할 thread 가 CPU 를 점유해 resource 가 낭비되는 상황이 발생한다. 따라서 이를 해결하기 위해 다른 방식으로 알고리즘을 변경해야 한다.

2. Algorithm design

기존의 알고리즘에서는 timer_sleep()함수에서 while loop 를 이용해 지정된 tick 까지 thread_yield() 함수 실행시켜 sleep 을 구현한다. 하지만 이는 busy waiting 을 일으키기 때문에 우리는 while loop 를 제거하였다. 그리고 지정된 tick 까지 thread 를 block 하고 ready_list 에서 제외한 후 sleep_list 에 push 하는 thread_sleep()함수를 구현하여 이를 timer_sleep()함수에 대신 사용하였다. 또한 block 된 thread 를 unblock 할 수 있는 thread_awake()함수를 만들어 timer_interrupt 에 사용해 현재 tick 과 sleep_list 내 thread 의 다음 wakeup tick 과 비교하여 현재의 tick 이 더 클 경우 thread_awake 를 실행시키도록 하였다.

3. Implementation

-Modify pintos/src/devices/timer.c

```
…
void timer_sleep(int64_t) {
…
        //replace while loop sleep algorithm
        thread_sleep(start+ticks);
}
…
static void timer_interrupt (struct intr_frame *args UNUSED){
…
        if(get_next_tick() <= ticks) {thread_awake(ticks);}
…
}
```

…

-Modify pintos/src/threads/thread.h

…

```
struct thread {
…
        int64_t wakeup_time;        // next wakeup tick
…
}
…
// timer.c function
void thread_sleep(int64_t ticks);
void thread_awake(int64_t ticks);
int64_t get_next_tick(void);
void update_next_tick(int64_t ticks);
```

…

-Modify pintos/src/threads/thread.c

…

```
// Variable of alarm clock
static int64_t next_tick;
static struct list sleep_list;
…
void thread_init (void){
…
        list_init (&sleep_list);
…
}
…
// Alarm clock function
void update_next_tick(int64_t ticks){
        if(next_tick > ticks) next_tick = ticks;
}
int64_t get_next_tick(void){
        return next_tick;
}
void thread_sleep(int64_t ticks){
```

```
        struct thread *thr;                 //declare thread

        // ban interrupt and save old_level
        enum intr_level old_level;
        old_level = intr_disable();

        // idle thread must not sleep
        thr = thread_current();
        ASSERT(thr != idle_thread);

        // update tick and push to sleep list
        uppdate_next_tick(thr->wakeup_time=ticks);
        list_push_back(&sleep_list, &thr->elem);

        // block thead until reschedule
        thread_block();

        // reactivate interrupt
        intr_set_level(old_level);
}

void thread_awake(int64_t awake_tick){
        struct list_elem *e;                // declare list elem for loop

        for(e=list_begin(&sleep_list); e != list_end(&sleep_list);{   // until sleep list end
                // get thread inside of sleep list
                struct thread *thr = list_entry(e, struct thread, elem);

                // determine let thread sleep or awake
                if(awake_tick < thr->wakeup_time){              // if thread isn't time for awake
                        e= list_next(e);                        // find next element
                        update_next_tick(thr->wakeup_time);   // update next closest tick
                }else{                                  // if thread isn't time for awake
                        e = list_remove(&thr->elem);// remove thread from sleep list
                        thread_unblock(thr);            // unblock the thread
```

```
                    }
                }
            }
```

Problem 2. Implementing Priority Scheduling

Requirement 1.

1. Problem definition

   Problem 2 는 thread 에 priority(우선순위)를 부여하여 scheduling 될 수 있도록 하는 것이 목적이다. 기존 pintos 의 scheduling 기법은 ready_list 맨 끝에 thread 를 단순히 추가하는 구조로, priority 가 고려되지 않았다. 이미 정의된 상수인 PRI_MIN(0) 부터 PRI_MAX(0) 사이의 priority 를 부여하고, baseline 은 PRI_DEFAUIT(31)이다. 항상 ready 상태의 thread 중 priority 가 가장 높은 thread 가 실행되어야 하며 priority 는 언제든 바뀔 수 있다.

2. Algorithm design

   thread_yield () 함수는 새로운 scheduling 이 필요할 때 사용하는 함수이다. 호출시 현재 thread 를 running 상태에서 ready 상태로 바꾸고 ready 큐에 추가, schedule 함수를 호출하여 thread switching 을 하고 next_thread_to_run () 함수의 리턴값으로 제공되는 thread 를 실행한다.

   thread_yield ()가 ready 큐의 thread 중 priority 가 가장 높은 thread 를 실행하도록 하기 위해 next_thread_to_run ()가 ready 큐 가장 앞의 thread 를 리턴하기 전에 ready 큐를 한번 priority 에 따라 정렬하도록 한다. 또한 priority 에 따라 정렬할 수 있도록 thread 간 priority 를 비교할 수 있도록 하는 함수를 구현하여 사용한다.

   thread_yield ()사용시 가장 priority 가 높은 thread 가 실행되도록 하였으니 다음의 경우를 고려해야한다. 첫 번째, thread 가 새로 생성되는 경우(thread_create ()) 생성된 thread 의 priority 가 높으면 thread_yield () 되어야 함. 두 번째, thread 가 unblock 되어 ready 큐에 들어오는 경우(thread_unblock ()) priority 에 따라 다시 정렬되어야 함. 세 번째, 현재 thread 의 priority 가 변경된 경우(thread_set_priority ()) 생성된 thread 의 priority 가 높으면 thread_yield () 되어야 함.

   또한 lock 되어있는 block 상태의 thread 가 해제되는 sema_up ()으로 인해 semaphore 의 waiter list 에서 thread 가 빠져나오며 unblock 되는데, 가장 높은 priority 의 thread 가 unblock 되게 하기 위해 waiter list 를 priority 에 따라 정렬한다. monitor 구현 부분의 condition variable 에 대해서도 마찬가지이다(cond_signal ()). condition variable 의 정렬시에는 각 semaphore waiter list 의 첫 번째 thread 의 priority 에 따라 정렬되게 해야 하며, 이를 위한 compare_prioirty_sema ()함수를 구현해야 한다.

3. Implementation

-Modify pintos/src/threads/thread.c

```
…
static bool
compare_priority (const struct list_elem * a_, const struct list_elem * b_, void* aux UNUSED) {
        const struct thread * a = list_entry(a_, struct thread, elem);
        const struct thread * b = list_entry(b_, struct thread, elem);
        return a->priority > b->priority;          // return 1 if priority of a is larger
}
…
static struct thread *next_thread_to_run (void) {
        if (list_empty (&ready_list))
                return idle_thread;
        else {
                 list_sort(&ready_list, compare_priority, NULL); //sort ready list before return
                return list_entry (list_pop_front (&ready_list), struct thread, elem);
        }
}
…
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);
        list_sort(&ready_list, compare_priority, NULL);          //sort ready queue
        t-> status = THREAD_READY;
        intr_set_level (old_level);
}
…
tid_t thread_create (const char *name, int priority, thread_func *function, void *aux) {
        …
        // If priority of new thread is larger than priority, we need to schedule again. Call thread_yield
        if (priority > thread_current()->priority) thread_yield();
        return tid;
```

```
}
…
void thread_set_priority (int new_priority) {
        int thread_priority = thread_current()->priority;
        thread_current() -> priority = new_priority;
        if (new_priority < thread_priority) thread_yield();
}
```

-Modify pintos/src/threads/synch.c

```
void sema_up (struct semaphore *sema) {
        enum intr_level old_level;

        ASSERT (sema != NULL);

        old_level = intr_disable ();
        sema->value++;
        //unblock biggest waiter
        if (!list_empty (&sema->waiters)){
                list_sort (&sema->waiters, compare_priority, NULL);
                struct thread *t = list_entry (list_pop_front (&sema->waiters),struct thread, elem);
                thread_unblock (t);
                if (t->priority > thread_current ()->priority)
                        thread_yield ();
        }
        intr_set_level (old_level);
 }

void cond_signal (struct condition *cond, struct lock *lock UNUSED) {
        ASSERT (cond != NULL);
        ASSERT (lock != NULL);
        ASSERT (!intr_context ());
        ASSERT (lock_held_by_current_thread (lock));

        //sort waiters queue before unblock
        list_sort(&cond->waiters, compare_priority_sema, NULL);
```

```
        if (!list_empty (&cond->waiters))
                sema_up (&list_entry (list_pop_front (&cond->waiters), struct semaphore_elem,
                elem)->semaphore);
}

//same with compare_priority () in thread.c
static bool
compare_priority (const struct list_elem * a_, const struct list_elem * b_, void* aux UNUSED) {
        const struct thread * a = list_entry(a_, struct thread, elem);
        const struct thread * b = list_entry(b_, struct thread, elem);
        return a->priority > b->priority;
}

static bool
compare_priority_sema (const struct list_elem * a_, const struct list_elem * b_, void* aux UNUSED) {
        const struct semaphore_elem * ta = list_entry(a_, struct semaphore_elem, elem);
        const struct semaphore_elem * tb = list_entry(b_, struct semaphore_elem, elem);

        const struct semaphore * tta = &ta->semaphore;
        const struct semaphore * ttb = &tb->semaphore;

        const struct thread * a = list_entry(list_front(&tta->waiters), struct thread, elem);
        const struct thread * b = list_entry(list_front(&ttb->waiters), struct thread, elem);

        return a->priority > b->priority;
}
```

Requirement 2.

1. Problem definition

   priority 를 통한 scheduling 도중 lock 을 사용하면 priority inversion 이 생길 수 있다. look 을 실행중인 thread 와 lock 을 필요로 하지 않는 ready 큐의 thread 때문에 lock 을 필요로 하며 이들보다 더 높은 priority 의 thread 가 앞서 lock 을 실행중인 thread 에 막혀 실행되지 않는 문제이다. 이는 lock 에는 한 thread 만 실행될 수 있으며, 이미 lock 에서 실행중인 thread 의 priority 가 lock 필요없이 ready 중이며 priority 가 높은 thread 에게 CPU 사용을 넘겨주어 일어나는 일이다. 때문에 이는 lock 에서 실행중인 낮은 priority 의 thread 에게 lock 을 필요로 하는 높은

priority 의 thread 가 자신의 priority 를 lock 이 풀릴 때 까지만 넘겨줌으로서 해결될 수 있는 문제이다. 이렇게 하면 lock 때문에 높은 priority 의 thread 가 자신보다 낮은 priority 의 thread 보다 늦게 실행되지 않을 것이다.

2. Algorithm design

lock 을 필요로 하는 높은 priority 의 thread 가 lock 을 점유중인 낮은 priority 의 thread 에 자신의 priority 를 넘겨주는 것을 priority donation 이라고 한다. priority donation 이후 lock 의 점유가 끝나고 다시 원래의 priority 로 돌아가기 위해 thread 는 자신의 원래 priority 를 알고 있어야 한다. 또한 한 하나 이상의 thread 로부터 donation 받거나 중첩으로 donation 받을 수 있으므로 다음 자료형은 struct thread 에 추가해야 한다.

-int old_priority: donation 이전에 갖고 있던 원래 priority

-struct lock *wait_locks : 당장 이 thread 를 block 되게 만든 lock. 이 lock 의 holder 를 통해 nested priority donation 문제를 해결할 수 있다.

-struct list locks: 여러 개의 priority donation 에 대응하기 위해 이 thread 에 donation 한 모든 thread 를 저장. 이를 통해 multiple priority donation 문제를 해결할 수 있음.

또한 struct lock 에는 다음 자료형을 추가해야 한다.

-struct list_elem lockelem : 위에서 선언한 thread->locks 에 lock 을 저장하기 위한 element

-int priority: 이 lock 을 holding 중인 thread 중 가장 높은 thread 의 priority 를 저장

priority donation 을 위해 우선 donation 받고자 하는 thread 와 donation 될 priority 를 인자로 받아 입력된 thread 의 priority 를 바꾸는 thread_donation () 함수를 선언한다. donation 후에도 ready 큐에 priority 가 더 높은 thread 가 있으면 thread_yield ()를 호출해 다시 스케쥴링 한다. 단순이 입력된 thread 의 priority 를 바꾸는 함수이므로 lock_acquire ()과 lock_release ()에서 신중하게 호출되어야 한다.

-Lock acquire 의 처리

우선 donation chain 을 구현해야 한다. 한 lock 을 점유하려는 여러 thread 에 대해 donation 되어야 할 priority 를 보이는 것으로 thread 구조체의 wait_locks->holder thread 가 또 어떤 lock 을 점유하여 할 때 이러한 일련의 donation chain 이 생길 수 있다. 이후 어떤 thread 가 lock 을 점유하려 하면 그 thread 의 priority 를 lock->holder 의 priority 와 비교하여 priority donation 을 결정한다. 만약 donation 하기로 결정되었다면 lock->holder 의 다음 thread 로부터도 donation 되어야 할지 결정해야한다. 즉, donation chain 을 따라 priority 를 donation 해야 한다. 이를 위해 위에서 말한 어떤 thread 가 있을 때 그 priority 가 thread->wait_locks->holder 의 priority 보다 커질 때까지 donation 받아야 한다.

lock relese 시 priority 를 되돌리기 위해 thread 가 점유하려 했던 lock (donation chain 가장 앞의 lock)에 대해 sema_down ()하고 lock->holder->locks 를 업데이트 하여 priority 를 donation 한

thread 를 저장하고 지속적으로 업데이트 해야 한다.

-Lock release 의 처리

sema_up ()하고 donation chain 을 따라 priority 를 원래로 돌려놓아야 한다. 이를 위해 점유중인 lock->holder->locks 를 보고 release 하려는 thread 에 donation 한 thread 들을 파악해야 한다. 이 과정은 다음과 같다. 기부를 해준 thread 가 있다면 그 중 가장 높은 priority 를 다시 donation 하여 donation chain 을 업데이트 한다. 더이상 list 에 기부를 해준 thread 가 없다면 old_priority 로 저장한 원래 priority 로 priority 를 복구해준다.

3. Implementation

-Modify pintos/src/threads/thread.h

```
…
Struct thread {
        …
        // priority donation
        int old_priority;
        struct lock *wait_locks;
        struct list locks;
        …
}
…
```

-Modify pintos/src/threads/thread.c

```
static void init_thread (struct thread *t, const char *name, int priority) {
        SSERT (t != NULL);
        SSERT (PRI_MIN <= priority && priority <= PRI_MAX);
        SSERT (name != NULL);

        emset (t, 0, sizeof *t);
        ->status = THREAD_BLOCKED;
        trlcpy (t->name, name, sizeof t->name);
        ->stack = (uint8_t *) t + PGSIZE;
        ->priority = priority;
        ->magic = THREAD_MAGIC;

        // piority donation
        t->old_priority = priority;
```

```
        t->wait_locks = NULL;
        list_init (&t->locks);

        list_push_back (&all_list, &t->allelem);
}
…
void thread_donation(struct thread *L, int donated) {
        L->priority = donated;
        if (L == thread_current () && !list_empty (&ready_list)){
                struct thread *top = list_entry(list_begin(&ready_list), struct thread, elem);
                if (top != NULL && top->priority > donated){
                        thread_yield (); }
        }
}
…
void thread_set_priority (int new_priority) {
        struct thread *curr = thread_current ();
        //dont need priority donation
        if (curr->priority == curr->old_priority) {
                curr->priority = new_priority;
                curr->old_priority = new_priority;
        }
        //need priority donation
        else{
                curr->old_priority = new_priority;
        }

        //yield when ready thread priority is bigger than new
        if (!list_empty (&ready_list)){
                struct thread *top = list_entry(list_begin(&ready_list), struct thread, elem);
        if (top != NULL && top->priority > new_priority){
                thread_yield (); }
        }
}
…
```

-Modify pintos/src/threads/synch.h

```
struct lock {
        struct thread *holder;      /* Thread holding lock (for debugging). */
        struct semaphore semaphore; /* Binary semaphore controlling access. */

        //priority donation
        struct list_elem lockelem;
        int priority;
};
```

-Modify pintos/src/threads/synch.c

```
…
void lock_acquire (struct lock *lock) {
        ASSERT (lock != NULL);
        ASSERT (!intr_context ());
        ASSERT (!lock_held_by_current_thread (lock));

        //priority donation
        struct lock *currlock = lock;
        struct thread *holder = lock->holder;
        struct thread *curr = thread_current ();

        curr->wait_locks = lock;
        if (holder == NULL)
                currlock->priority = curr->priority;

        while (holder != NULL && holder->priority < curr->priority){
                thread_donation(holder, curr->priority);
                if (currlock->priority < curr->priority)
                        currlock->priority = curr->priority;

                currlock = holder->wait_locks;
                if (currlock == NULL) break;
                        holder = currlock->holder;
        }
```

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

        lock->holder->wait_locks = NULL;
        list_insert_ordered(&(lock->holder->locks), &(lock->lockelem), compare_priority_lock, NULL);
    }
    …
    void lock_release (struct lock *lock) {
        ASSERT (lock != NULL);
        ASSERT (lock_held_by_current_thread (lock));

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

        //priority donation
        struct thread *curr = thread_current ();
        list_remove (&lock->lockelem);

        if (list_empty(&curr->locks)){
            thread_donation(curr, curr->old_priority);}
        else{
            list_sort(&(curr->locks), compare_priority_lock, NULL);
            struct lock *top = list_entry(list_front(&(curr->locks)), struct lock, lockelem);
            thread_donation(curr, top->priority);
        }
    }
    …
```

Problem 3. Implementing Advanced Scheduler

1. Problem definition

Problem 3 는 Multi-Level Feedback Queue scheduler (BSD 와 유사, mlfqs 라 서술함)를 구현하는 것에 목적을 두고 있다. 이 scheduler 는 priority 에 기반하여 thread 를 schedule 하도록 구현되어야 하지만 priority donation 을 하지 않아야 한다. Pintos material 의 4.4 BSD 에 따른 mlfqs 의 구현은 다음을 만족시켜야 한다.

첫 번째, thread_set_priority 를 수행하지 않음(priority donation 을 하지 않기 때문). 두 번째, time

interrupt 가 발생할 때마다 recent_cpu 를 1 증가시킴. 세 번째, TIMER_FREQ 마다 recent_cpu 를 업데이트 한다. 네 번째, 4 tick 마다 priority 를 update 한다. 다섯 번째, nice 값이 변경되면 priority 에 따라 scheduling 을 다시 해야함.

2. Algorithm design

mlfqs 의 구현을 위하여 4.4 BSD material 에 묘사된 다음 수식대로 priority 와 recent_cpu 를 문제정의에서의 규칙대로 update 해야한다.

$$priority = PRI_{MAX} - \left(\frac{recent\_cpu}{4}\right) - (nice * 2)$$

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

하지만 위의 계산을 위해서는 fixed point operation 이 필요하지만 pintos kernel 에서 지원하지 않으므로 직접 구현한다. fixop.c, fixop.h 파일을 만들어 구현하였으며, 구현 방법은 4.4 BSD material 에 자세히 설명되어 있다.

이전에 구현했던 priority donation 이 thread_mlfqs flag 가 true 인 경우 실행되지 않도록 조건문을 추가하고 위 문제정의에서의 조건에 따라 priority 와 recent_cpu 를 조정할 수 있도록 update_recent_cpu()와 update_priority() 함수를 구현하였으며, devices/timer.c 파일의 timer_interrupt() 함수에 문제정의에 따라 함수가 사용될 수 있도록 하였다. update 의 경우 all_list 가 담고 있는 모든 thread 에 대해 recent_cpu 와 recent_cpu 의 update 에 필요한 load_avg, priority 를 update 하도록 하고 priority 업데이트시 다시 scheduling 하도록 구현하였다. 이를 위해 thread.h 의 thread 구조체에 int recent_cpu 와 int nice 값을 선언하였다.

3. Implementation

-create pintos/src/threads/fixop.c

```
#include “threads/fixop.h”
#include <stdint.h>
int int_mis_flt (int i, int f) {
        return i*FRA - f;
}

int int_mul_flt (int i, int f) {
        return i*f;
}

int flt_pls_int (int f, int i) {
        return f + i*FRA;
}
```

```
int flt_div_int (int f, int i) {
	return f / i;
}

int flt_pls_flt (int f, int f_) {
	return f + f_;
}

int flt_mis_flt (int f, int f_) {
	return f - f_;
}

int flt_mul_flt (int f, int f_) {
	int64_t temp = f;
	temp = temp * f_ / FRA;
	return (int)temp;
}

int flt_div_flt (int f, int f_) {
	int64_t temp = f;
	temp = temp * FRA / f_;
	return (int)temp;
}
```

-create pintos/src/threads/fixop.h

```
#ifndef FIXOP_H
#define FIXOP_H
#define FRA (1<<14)

int int_mis_flt (int, int);
int int_mul_flt (int, int);
int flt_pls_int (int, int);
int flt_div_int (int, int);
int flt_pls_flt (int, int);
int flt_mis_flt (int, int);
```

```
int flt_mul_flt (int, int);
int flt_div_flt (int, int);

#endif
```

-modify pintos/src/threads/thread.h

```
…
struct thread {
        …
        int wakeup_time;            // declare at Problem 1
        //priority donation
        int old_priority;
        struct lock *wait_locks;
        struct list locks;

        //mlfqs problem
        int recent_cpu;
        int nice;
}
…
```

-Modify pintos/src/threads/thread.c

```
…
#include “threads/fixop.h”
…
static int load_avg;
…
void thread_init (void) {
        …
        initial_thread->nice = 0;
        initial_thread->recent_cpu = 0;
        …
}
…
static void init_thread (struct thread *t, const char *name, int priority) {
```

```
	ASSERT (t != NULL);
	ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
	ASSERT (name != NULL);

	memset (t, 0, sizeof *t);
	t->status = THREAD_BLOCKED;
	strlcpy (t->name, name, sizeof t->name);
	t->stack = (uint8_t *) t + PGSIZE;
	t->priority = priority;
	t->magic = THREAD_MAGIC;

	//priority donation
	t->old_priority = priority;
	t->wait_locks = NULL;
	list_init (&t->locks);

	//mlfqs
	t->recent_cpu = running_thread ()->recent_cpu;
	t->nice = running_thread ()->nice;

	list_push_back (&all_list, &t->allelem);
}
…
int get_max_priority (void) {
	int priority = -1;
	struct thread *t;
	if (!list_empty(&ready_list)){
		t = list_entry(list_front(&ready_list), struct thread, elem);
		priority = t->priority;
	}
	return priority;
}
…
void update_recent_cpu(void){
	int ready_threads = list_size(&ready_list);
```

```
        struct thread *t;
        struct list_elem *e;

        if (thread_current () != idle_thread){
                ready_threads += 1;}

        load_avg = flt_div_int(flt_pls_int(int_mul_flt(59, load_avg), ready_threads), 60);

        for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e)){
                t = list_entry(e, struct thread, allelem);
                if (t != idle_thread){
                        t->recent_cpu = flt_pls_int(flt_mul_flt(flt_div_flt(int_mul_flt(2, load_avg), flt_pls_int(int_mul_flt(2, load_avg), 1)), t->recent_cpu), t->nice);
                }
        }
}
…
void update_priority (void){
        struct thread *t;
        struct list_elem *e;

        for (e = list_begin(&all_list); e != list_end(&all_list); e = list_next(e)){
                t = list_entry(e, struct thread, allelem);
                t->priority = flt_mis_flt(flt_mis_flt(flt_pls_int(0, PRI_MAX), flt_div_int(t->recent_cpu, 4)), int_mul_flt(2, flt_pls_int(0, t->nice))) / FRA;
                if (t->priority > PRI_MAX){
                        t->priority = PRI_MAX;}
                if (t->priority < PRI_MIN){
                        t->priority = PRI_MIN;}
        }

        if (thread_current ()->priority < get_max_priority ()){
                intr_yield_on_return ();}
}
…
```

```
void thread_set_priority (int new_priority) {
        if (thread_mlfqs){ return; }    //mlfqs

        struct thread *curr = thread_current ();
        //dont need priority donation
        if (curr->priority == curr->old_priority){
                curr->priority = new_priority;
                curr->old_priority = new_priority;
        }
        //need priority donation
        else{
                curr->old_priority = new_priority;
        }
        //yield when ready thread priority is bigger than new
        if (!list_empty (&ready_list)){
                struct thread *top = list_entry(list_begin(&ready_list), struct thread, elem);
                if (top != NULL && top->priority > new_priority){
                        thread_yield (); }
        }
}
  …
  void thread_set_nice (int nice) {
        struct thread *t = thread_current ();
        t->nice = nice;
        t->priority = flt_mis_flt(flt_mis_flt(flt_pls_int(0, PRI_MAX), flt_div_int(t->recent_cpu, 4)),
        int_mul_flt(2, flt_pls_int(0, t->nice))) / FRA;
        if (t->priority > PRI_MAX){
                t->priority = PRI_MAX;}
        if (t->priority < PRI_MIN){
                t->priority = PRI_MIN;}
        if (t->priority < get_max_priority ()){ thread_yield (); }
  }
  …
  int thread_get_nice (void) {
        return thread_current ()->nice;
```

```
}
…
int thread_get_load_avg (void) {
        return int_mul_flt(100, load_avg) / FRA;
}
…
int thread_get_recent_cpu (void) {
        return int_mul_flt(100, thread_current ()->recent_cpu) / FRA;
}
…
```

-Modify pintos/src/threads/synch.c

```
…
void lock_acquire (struct lock *lock) {
        ASSERT (lock != NULL);
        ASSERT (!intr_context ());
        ASSERT (!lock_held_by_current_thread (lock));

        if(!thread_mlfqs){
                //priority donation
                struct lock *currlock = lock;
                struct thread *holder = lock->holder;
                struct thread *curr = thread_current ();

                curr->wait_locks = lock;
                if (holder == NULL) currlock->priority = curr->priority;
                while (holder != NULL && holder->priority < curr->priority){
                        thread_donation(holder, curr->priority);
                        if (currlock->priority < curr->priority)
                                currlock->priority = curr->priority;

                        currlock = holder->wait_locks;
                        if (currlock == NULL) break;
                        holder = currlock->holder;
                }
```

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

                lock->holder->wait_locks = NULL;
                list_insert_ordered(&(lock->holder->locks), &(lock->lockelem),
                compare_priority_lock, NULL);
        } else {
                sema_down (&lock->semaphore);
                lock->holder = thread_current ();
        }
}
…
void lock_release (struct lock *lock) {
        ASSERT (lock != NULL);
        ASSERT (lock_held_by_current_thread (lock));

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

        if(!thread_mlfqs){
                //priority donation
                struct thread *curr = thread_current ();
                list_remove (&lock->lockelem);

                if (list_empty(&curr->locks)){
                        thread_donation(curr, curr->old_priority);}
                else{
                        list_sort(&(curr->locks), compare_priority_lock, NULL);
                        struct lock *top = list_entry(list_front(&(curr->locks)), struct lock,
                        lockelem);
                        thread_donation(curr, top->priority);
                }
        }
}
…
```

-Modify pintos/src/devices/timer.c

…

```
static void timer_interrupt (struct intr_frame *args UNUSED) {
        ticks++;
        if(get_next_tick() <= ticks){thread_awake(ticks);}
        // New
        if (thread_mlfqs){
                thread_current ()->recent_cpu = flt_pls_int(thread_current ()->recent_cpu, 1);
                if (timer_ticks () % TIMER_FREQ == 0){
                        update_recent_cpu ();}
                if (timer_ticks () % 4 == 0){
                        update_priority ();}
        }
        thread_tick ();
}
```

…

-Modify pintos/src/Makefile.build

…

```
Threads_SRC += threads/fixop.c          # fixed point operations.
```

…

Appendix. Result picture