PintOS Lab1 Final Report

Alarm Clock

Background

a. Timer Interrupt

- 시스템에서 정해진 간격으로 CPU에 신호를 보냄으로써 CPU는 현재 실행중인 작업을 일시 중단하고 scheduler와 같은 시스템 서비스 과정을 수행하게 된다.
- 즉, timer interrupt는 thread간의 시간을 분배하고, event간의 시간 간격을 측정하고, 시스템 성능을 모니터링하는데 사용한다.
- 특정 시간 간격이 지날 때마다 timer는 CPU에 interrupt를 보낸다. 그리고 CPU는 interrupt를 받으면 현재 실행중인 작업의 상태를 저장하고 interrupt 처리 루틴을 실행한다.

b. Timer Sleep

- Thread나 process를 일정시간 동안 대기 상태로 전환하는 기능이다.
- program의 실행 흐름을 일시적으로 중지함으로서 특정 작업의 실행을 중단하거나 CPU 자원의 과도한 사용을 방지하는데 사용한다.
- OS의 Timer에 의해서 구현된다. Timer sleep 함수가 호출하면, 호출된 thread는 지정된 시간동안 실행을 멈추고 대시 상태로 전환된다. 대기 시간 동안에는 CPU는 다른 thread의 작업을 처리할 수 있으므로 효율적인 관리가 가능해진다. 그리고 지정된 대기시간이 지나면, 운영체제의 scheduler는 대기중인 thread를 다시 실행 상태로 변경하고 작업을 재개한다.

Previous Alarm Clock

```
/* Returns the number of timer ticks since the OS booted.
*/
int64_t timer_ticks (void){
  enum intr_level old_level = intr_disable ();
  int64_t t = ticks;
  intr_set_level (old_level);
  return t;
```

```
}

/* Returns the number of timer ticks elapsed since THEN, wh
ich
   should be a value once returned by timer_ticks(). */
int64_t timer_elapsed (int64_t then){
  return timer_ticks () - then;
}

/* Sleeps for approximately TICKS timer ticks.  Interrupts
must
   be turned on. */
void timer_sleep (int64_t ticks){
  int64_t start = timer_ticks ();
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

Improving timer_sleep

- timer_sleep으로부터 thread를 다시 실행시킬 시간을 입력받아야 함
  - struct thread에 `int64_t wakeUp` 를 추가
- 지정된 시간동안 thread를 block해서 thread_yield를 통해 CPU를 받지 못하도록 함
  - thread의 block, unblock에는 `thread_block`, `thread_unblock`을 이용
  - block 상태의 thread를 관리하는 list가 필요 → thread.c에서 `sleep_list` 를 선언
    - `sleep_list`는 `thread_init` 함수에서 all_list, ready_list를 초기화할 때 같이 초기화한다.
  - thread를 sleep 시키고, wakeUp을 저장해주는 함수가 필요 → `thead_sleep`
    - 생성한 `thread_sleep` 함수를 `timer_sleep`에서 실행시킴으로서, `wakeUp`을 설정하고, thread를 block한 뒤, `sleep_list`에 추가할 수 있음
    - `timer_sleep`은 thread가 시작할 때 생성되므로, start는 thread가 block 되는 tick을 의미한다. 그리고 ticks는 OS가 지정해주는 시간으로 `timer_sleep` 함수의 argument이다. 따라서, thread를 깨우는 시간 `wakeUp`은 `start + ticks`가 되어야한다.

- 자세히 설명하면 timer_sleep 함수가 thread_sleep을 호출하게 되고 이때 start 값과 ticks를 받아온다. start는 os에서 관리하는 tick이고 ticks는 해당 thread가 언제 깨어나야하는지에 관한 정보가 들어있다. thread_sleep에서 우선 interrupt를 막아두고 idle이 아닐때에만 실행된다. thread 구조체의 wakeUp 변수를 start_ticks 값으로 저장해준다. 즉 해당 thread가 일어나야 할 시간을 나타내고 해당 thread를 elem을 이용하여 sleep list에 넣어준다. list에 넣어주는 과정과 elem 등은 priority schedule에 자세히 설명할 것이기 때문에 생략할 것이다. sleep list에 넣어줄 때는 가장 뒤에 넣어준다. 그리고 기존과 다르게 해당 thread는 block 상태가 되어 잠들게 된다.

```
// threads/thread.c
static struct list sleep_list;
...

void thread_init (void){
  ...
  list_init (&sleep_list);
  ...
}
...

void thread_sleep (int64_t start , int64_t ticks){
  struct thread *cur;
  enum intr_level old_level;
  int64_t wakeup_time;
  old_level = intr_disable ();
  ////////////////////////////
  cur = thread_current ();
  ASSERT (cur != idle_thread);

  wakeup_time = start + ticks;
  cur->wakeUp = wakeup_time;
  list_push_back (&sleep_list, &cur->elem);
  thread_block ();
    ////////////////////////////
  intr_set_level (old_level);
}
```

```
// devices/timer.c
void timer_sleep (int64_t ticks){
  int64_t start = timer_ticks ();
  ASSERT (intr_get_level () == INTR_ON);
  thread_sleep (start, ticks);
}
```

- wakeUp에 도달하면 다시 ready state로 전환하는 함수가 필요 → `thread_awake`
  - `thread_awake` 함수는 현재의 ticks를 wakeUp과 비교하여, `ticks ≥ wakeUp`이면 thread를 unblock해주는데, 이를 위해서는 매 tick마다 실행되는 함수가 필요하다. 따라서, `thread_interrupt` 함수에서 `thread_awake`를 호출한다.
  - thread_awake 함수에 대해서 더 자세하게 설명해보면 모든 sleep list상에 있는 thread들에 접근하면서 wakeUp과 현재 ticks를 비교하여 wakeUp이 더 작다면, 즉 깨어나도 상관없는 상황일때 unblock해준다. unblock 해준다는것은 sleep list가 아닌 ready list로 옮겨준다는 것을 의미한다.
  - 마지막으로 timer_interrupt에 대해서 자세히 설명하면 매 tick마다전체 os ticks를 1 증가시켜주고 thread_tick을 통하여 thread들의 종류에 따라 tick을 증가시켜준다. 그리고 매 tick마다 모든 sleep list의 원소들에 접근하여 깨어날 수 있는 thread를 찾아 ready list로 옮겨주어 cpu를 할당 받을 수 있게 해준다.
  - 처음 os 코드를 봐서 그런지 timer_sleep, thread_sleep 함수끼리 헷갈려서 코드적으로 많이 꼬였고 thread_awake함수를 timer_interrupt에 포함시켜준다는 생각을 하지 못해서 많은 오류가 생성되어 시간이 오래 지체되었던 것 같다.

```
// threads/thread.c
void thread_awake (int64_t ticks){
  struct list_elem *i;

  for( i  = list_begin(&sleep_list); i!=list_end(&sleep_
list) ;)
    {
        struct thread *t = list_entry(i, struct thread,
elem);
        if (t->wakeUp <= ticks) {
            i = list_remove(i);
```

```
            thread_unblock(t);
        }
        else
            i = list_next(i);
    }
}
```

```
// devices/timer.c
static void timer_interrupt (struct intr_frame *args UNU
SED){
  ticks++;
  thread_tick ();
  thread_awake (ticks);
}
```

Priority Scheduler

Background

Semaphore

- Semaphore는 Shared object에 접근할 수 있는 thread의 수를 제한하기 위해서 사용한다.

Lock

Conditional Variable

Previous Scheduler (Round-Robin Scheduler)

기존의 scheduler는 Round-Robin 방식으로 구현되어있었다.

RR schduler는 모든 process에 동일한 시간을 배분함으로써 공평하게 CPU를 할당 받을 수 있게 하도록 설계되어있다. state 전환을 생각해보면, current thread가 CPU 할당을 종료하는 경우에 다음으로 실행할 thread를 결정하게 된다. 그렇기 때문에 `thread_yield`, `thread_block`, `thread_exit` 함수를 살펴 보아야 한다.

```
// 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);
}

void thread_block (void){
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);

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

void thread_exit (void){
  ASSERT (!intr_context ());
    #ifdef USERPROG
      process_exit ();
    #endif

  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
  NOT_REACHED ();
}
```

세개 함수의 코드를 살펴보면 공통적으로 `schedule` 함수가 실행되는 것을 볼 수 있다. 그렇기에, 이 함수를 통해 current thread 이후에 실행될 thread를 결정하는 함수임을 알 수 있

다. `schedule` 함수의 코드는 아래와 같다.

```
// 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);
}

static struct thread * next_thread_to_run (void){
  if (list_empty (&ready_list))
    return idle_thread;
  else
    return list_entry (list_pop_front (&ready_list), struct
thread, elem);
}
```

`schedule` 함수의 코드를 보면 `next_thread_to_run` 를 이용해 next thread를 결정하고 있음을 확인할 수 있다. 그리고 `next_thread_to_run`는 `ready_list`의 맨 앞에 있는 thread를 pop하여 next thread로 설정하는 것을 알 수 있다. swtich_thread를 통하여 context swtich를 진행해주고 관련된 thread 정보들을 관리한다. 이제 , ready_list에 thread를 추가하는 `thread_unblock`, `thread_yield` 함수를 살펴보아야한다.

```
// 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);
}

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);
}
```

코드를 살펴보면 두 코드 모두 `list_push_back` 함수를 이용해서 `ready_list`의 맨 뒤에 thread를 push하는 것을 볼 수 있다. 이를 통해 현재의 scheduler는 FIFO 방식으로 thread를 할당하는 Round-Robin으로 구현되어있음을 확인할 수 있다. 따라서 thread들의 중요도와 관련 없이 먼저 들어온 thread가 먼저 cpu를 할당 받는다. 이 부분을 thread가 ready list에 들어온 순서가 아닌 priority 기준으로 변경해주어야 한다.

Improving Scheduler

Priority ordering

가장 먼저 해야할 것은 priority에 기반하여 next thread를 선택하는 것이다. priority를 관리하는 방법 중 ready list는 들어온 순서대로 관리하고 선택해줄 때 priority가 큰 것을 선택해주는 방법이 있고 ready list 자체를 내림차순으로 관리하는 방법이 있다. 우리는 두번째 방법을 선택하여 ready_list의 thread들을 **priority descending order**로 관리되도록 해야한다. 이를 위해, `list_push_back` 함수 대신 `list_insert_ordered` 함수를 이용해 ready_list에 thread를 priority order에 맞춰 넣어줘야한다.

```
// lib/kernel/list.c
void list_insert_ordered (struct list *list, struct list_el
```

```
em *elem,
                      list_less_func *less, void *aux){
  struct list_elem *e;

  ASSERT (list != NULL);
  ASSERT (elem != NULL);
  ASSERT (less != NULL);

  for (e = list_begin (list); e != list_end (list); e = lis
t_next (e))
    if (less (elem, e, aux))
      break;
  return list_insert (e, elem);
}

void list_insert (struct list_elem *before, struct list_ele
m *elem){
  ASSERT (is_interior (before) || is_tail (before));
  ASSERT (elem != NULL);

  elem->prev = before->prev;
  elem->next = before;
  before->prev->next = elem;
  before->prev = elem;
}
```

`list_insert` 함수는 elem을 before 앞에 넣어주는 함수이다. less함수 자리에 우리가 thread들의 priority를 비교하여 boolean의 형태로 출력되는 함수를 따로 정의해줘야 한다. 구현한다면 `list_insert_ordered` 함수는 반복문과 priority를 비교하는 함수를 이용해 thread를 ready_list의 원하는 위치에 넣어주게 된다. 따라서, 우리는 priority를 비교하는 `thread_priority_compare` 함수를 작성해줘야한다. 그 코드는 아래와 같다. 처음에 코드 상으로 실수 했었던 부분이 thread에 접근하기 위해서 단순히 ready list에 접근하여 thread를 가져오려했다는 점이다. 잘못된 코드를 따로 저장하지 않아 코드는 첨부할 수 없지만 list의 주소를 이용하여 바로 thread에 접근하려 했는데 잘못 실행되었다. pintos 상에서는 list_elem 구조체를 이용하여 여러가지 list를 동시에 관리할 수 있고 자신이 위치할 곳의 앞 노드 , 뒤 노드 등의 주소를 통하여 접근한다고 한다. 따라서 thread들의 priority를 비교할 때 list_elem 의 new와 r_list를 parameter로 받아주고 list_entry 함수를 이용하여 내가 비교하고자 하는 thread 2개를 받아온다. 여기서 list_elem은 list를 효과적으로 관리하기

위한 방법이다. 실제로 이 list들은 elem이라는 앞 과 뒤 주소만 가지고 있을 뿐 thread를 가지고 있지 않다. list_elem 값을 이용하여 list 상의 자신의 노드의 위치를 알아내고 elem을 이용하여 정보가 담긴 구조체인 thread로 확장시켜 우리가 실제로 원하는 thread에 접근할 수 있게 된다. 앞으로 이런식의 구현이 많이 일어날텐데 모두 비슷한 원리로 함수를 호출하게 된다. 이런 방법으로 비교할 thread를 불러오고 부등호를 이용해서 return하여 위의 list_insert_ordered함수의 if문에 넣어준다.

```
// threads/thread.c
bool thread_priority_compare (struct list_elem *a, struct l
ist_elem *b, void *aux UNUSED){
  int a_priority = list_entry (a, struct thread, elem)->pri
ority;
  int b_priority = list_entry (b, struct thread, elem)->pri
ority;

  return a_priority > b_priority;
}
```

이제 `thread_priority_compare`와 `list_insert_ordered` 함수를 이용하여 `thread_unblock`과 `thread_yield` 함수의 `list_push_back`을 대체하여 아래와 같이 코드를 작성하였다.

그러면 unblock 함수와 yield 함수는 내가 현재 진행하고 있던 thread가 만약 다른 thread로 교체된다면 ready list에 넣어주고 이때 ready list가 priority 내림차순으로 유지될 수 있게끔 넣어준다. yield도 마찬가지로 list_insert_ordered를 적용시켜준다.

```
// 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_insert_ordered (&ready_list, &t->elem, thread_priori
ty_compare, 0);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

```
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_insert_ordered (&ready_list, &cur->elem, thread_pr
iority_compare,0);
  }
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

이제 ready_list의 thread가 prioritiy 순서대로 실행되는 것을 구현할 수 있다. 그런데 `thread_set_priority` 함수에 의해 current thread의 priority가 바뀌거나 `thread_create` 함수에 의해 새로운 thread가 생기는 경우 current thread의 priority보다 ready_list의 front의 priority가 커지는 경우가 생긴다. create 된 thread는 unblock된 상태로 state가 바뀌고 ready list에 들어가게 된다. 따라서 current thread와 ready list의 priority를 비교한다. 이 경우 current thread와 next thread를 바꿔주어야한다. 그렇기 때문에 CPU의 소유권을 바꿔주는 `swap_for_update` 함수를 작성하고 thread_set_priority와 thread_create 함수에 추가해주었다.

swap_for_update를 구현하는 과정에서도 ready list의 첫번째 thread를 불러오는 과정에서 위에서 말한 elem 개념이 필요하였고 list_entry함수를 이용하여 ready list상에서 관리되고 있는 첫번째 thread의 priority를 받아온다. 그리고 만약 ready list의 priority가 현재 진행중인 thread보다 priority가 크다면 cpu 할당을 옮겨줘야 되므로 thread_yield함수를 호출한다.

```
// threads/thread.c
void swap_for_update (void){
  if (!list_empty (&ready_list)){
    int curr_priority = thread_current ()->priority;
    int ready_priority = list_entry (list_front (&ready_lis
t), struct thread, elem)->priority;
    int max_priority = max(curr_priority, ready_priority);
```

```
    if (curr_priority < max_priority)
        thread_yield();
  }
}

tid_t thread_create (const char *name, int priority,thread_
func *function, void *aux){
    ...
  /* Add to run queue. */
  thread_unblock (t);
  swap_for_update ();

  return tid;
}

void thread_set_priority (int new_priority){
  thread_current ()->priority = new_priority;
  swap_for_update ();
}
```

이렇게까지 코드를 작성하였을때, 16개의 test case에서 fail하는 것을 확인할 수 있었다.

Limitatioins

여기까지 구현하였을 때, thread의 실행 순서는 priority를 따르지만, shared object에 접근하는 순서는 최적화 되지 못한 상태가 된다. 따라서, shared object에 접근하는 것도 priority를 따라서 이뤄질 수 있도록 scheduler를 더 개선해야한다.

Synchronization

Synchronization variable을 얻고자하는 경우에도 높은 priority를 갖는 thread가 먼저 synchronization variable을 가질 수 있도록 구현해야한다. 이를 통해 높은 priority의 thread가 먼저 shared object에 접근할 수 있도록 한다. 따라서, semaphore, lock, condition variable의 사용을 관리하는 sema_down, sema_up, lock_acquire, lock_release, cond_wait, cond_signal 함수를 살펴보아야한다.

먼저, `sema_down`과 `sema_up`의 코드를 살펴보면 아래와 같다.

```
// threads/sync.c
struct semaphore
{
     unsigned value;             /* Current value. */
   struct list waiters;        /* List of waiting threads.
*/
 };

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

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0) {
      list_push_back (&sema->waiters, &thread_current ()->e
lem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    thread_unblock (list_entry (list_pop_front (&sema->wait
ers), struct thread, elem));
  }
  sema->value++;
```

```
  intr_set_level (old_level);
}
```

thread는 `sema_down`을 통해 semaphore value가 0인 동안 block된 상태로 while문 안에서 대기하고 semaphore value가 양수가 되면 semaphore를 획득하여 while문을 탈출하여 다음 instruction이 실행되고 shared object에 접근하려고 시도한다. 그리고 shared object에 접근을 성공하고 critical section을 수행하고 shared object 사용이 끝나면 semaphore를 `sema_up` 함수를 통해 semaphore를 놓아준다. 두 함수를 살펴보면 thread가 semaphore를 기다리는 경우 `waiters`라는 list를 통해 관리되는 것을 볼 수 있다. waiters list는 shared object를 사용하고 싶은데 현재 다른 thread가 점유하고 있어 사용하지 못하고 대기(blcok)하고 있는 상황이다. 그러고 만약 shared object를 점유하고 있던 thread가 사용을 끝내 다른 thread를 배당하고 싶을 때는 sema_up함수를 이용한다. sema_down에서 `list_push_back`으로 만들어진 waiters list에서 thread를 pop해주고 해당 thread를 unblock 해준다. 이때에도 priority에 따라 unblock 시켜줄 thread를 선택해야 한다. 그런데 현재 구현된 것은 선입선출 구조이므로 이 구조를 바꿔주자. waiters list를 priority 순서대로 관리하기 위해서는 priority ordering에서 했던것처럼 `list_push_back` 대신 `list_insert_ordered` 함수를 사용하면 된다. 이때 compare 함수로는 `thread_priority_compare`를 사용하면 된다. 정한 코드는 아래와 같다 .

또한, sema_up에서 waiters list의 thread를 unblock 해주기 전에 priority에 변동이 있었을 수 있으므로 waiters list를 list_sort 함수를 이용해서 sorting 해줘야하는데, 이때도 `thread_priority_compare` 함수를 사용해주면 된다. (앞에와 같은 상황은 shared object를 갖고 있던 thread가 sema_up을 호출하면 waiters list에서 pop해주게 되는데 직전에 create등을 통해서 priority가 높은 thread가 ready_list에 추가되어 cpu를 선점하고 이때 그 thread가 만약 해당 shared object를 원해서 sema_Down을 통하여 waiters에 들어간다면 발생할 수 있다.) 추가적으로 우리는 mesa 방법을 사용하기 때문에 thread가 unblock 되면 ready_list에 새로운 thread가 추가된다. 자세히 설명하면 sema_up함수를 통하여 waiters list에서 pop된 thread가 ready list에 들어가게 된다. 따라서 ready list와 current_thread의 priority를 비교하자. 따라서, `swap_for_update` 함수를 실행해서 current thread의 priority와 ready thread의 priority를 비교해 thread yield 실행 유무를 확인해줘야한다. 수정한 코드는 아래와 같다.

```
// threads/sync.c
void sema_down (struct semaphore *sema){
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());
```

```
  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_insert_ordered (&sema->waiters, &thread_current
() -> elem, thread_priority_compare, 0);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}

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

  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    list_sort (&sema->waiters, thread_priority_compare, 0);
    thread_unblock (list_entry (list_pop_front (&sema->wait
ers), struct thread, elem));
  }
  sema->value++;

  swap_for_update ();
  intr_set_level (old_level);
}
```

다음으로 lock을 관리하는 `lock_acquire`, `lock_release` 함수를 살펴보아야한다. 코드는 아래와 같다.

```
// threads/sync.c
struct lock
{
    struct thread *holder;      /* Thread holding lock (for
debugging). */
  struct semaphore semaphore; /* Binary semaphore controlli
ng access. */
```

```
};

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 = curr;
}

void lock_release (struct lock *lock){
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

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

`lock_acquire`과 `lock_release` 함수를 보면 lock→holder를 설정하고 관리해주는것 이외에는 lock 구조체에 있는 semaphore에 대한 sema_down, sema_up 함수를 사용하는 것을 볼 수 있다. 따라서, lock에 대해서는 추가적인 수정이 필요 없다.

마지막으로 condition variable을 관리하는 cond_wait, cond_signal_cond_broadcast 함수를 살펴보아야한다. 코드는 아래와 같다.

```
// threads/sync.c
struct condition
{
    struct list waiters;        /* List of waiting threads.
*/
};

void cond_wait (struct condition *cond, struct lock *lock){
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
```

```
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());

  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}

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));

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

void cond_broadcast (struct condition *cond, struct lock *l
ock){
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);

  while (!list_empty (&cond->waiters))
    cond_signal (cond, lock);
}
```

condition variable은 특정 조건이 만족될때까지 thread를 block 하는 함수이다. cond_wait 함수를 보면 lock을 획득하고 있는 동안 semaphore_elem 객체 waiter를 생성한 뒤, waiter의 semaphore를 0으로 초기화한 뒤, list_push_back을 이용해 struct condition의 waiters list의 맨 뒤에 넣어준다. 즉, conditon variable의 경우 thread의

elem으로 이루어진 waiters list를 갖는 semaphore, lock과 달리 semaphore의 elem으로 이루어진 waiters list를 갖는다. 따라서, conditon variable는 waiters list의 semaphore elem의 semaphore로 접근하여 semaphore가 갖고 있는 waiters list의 맨 앞에 위치하는 thread간의 priority 순서대로 list를 관리해주어야한다. 즉 어떤 조건이 충족되기 위해서 thread들이 잠들게 되는데(block) 나중에 이 조건이 충족되었을 때 가장 먼저 실행할(sema_down) thread를 골라주어야 한다. 그러므로 우리는 추가로 semaphore waiters list의 맨 앞의 thread들의 priority를 비교하는 sema_prioriity_compare 함수를 선언해주었다. 그리고 이 함수를 이용해서 cond_wait의 list_push_back을 list_insert_ordered로 대체해주었다. 이후에는 lock을 release 해주고 cond의 semaphore가 1이 될때까지 thread를 block 해줄 수 있도록 한다. 그리고 다시 lock_acquire를 이용해 lock을 획득해주면 된다.

cond_signal은 특정 조건이 만족되었을 경우, sema_up을 통해 특정 조건에 대한 semaphore 값을 1로 만들어준다. 이렇게 되면 cond_wait 함수는 sema_down 이후의 코드를 실행할 수 있게 된다. 그리고 이 과정은 condition variable의 waiters list의 맨 앞에 위치하는 semaphore 부터 sema_up 해주는 방식으로 실행되고 있었다. 그러나 각 semaphore waiters list의 맨 앞 thread의 priority가 바뀌었을 수 있기 때문에 list_sort 함수로 list를 정렬해주었다. 이때, 비교함수로는 sema_priority_compare 함수를 사용하였다. 그리고 condition variable의 경우에는 ready_list에 thread가 추가되는 경우가 없기 때문에 따로 current thread와 ready thread를 swap 해주지 않아도 된다.

마지막으로 cond_broadcast 함수를 보면 단순히 모든 condition semaphore들에 대해서 sema_up을 해주는 함수이기 때문에 추가적으로 구현해야하는 부분은 없었다.

따라서, 최종적으로 구현된 코드를 보면 아래와 같다.

우선 wait 함수를 통해 waiters list에 semaphore_elem인 waiter를 이용하여 wait을 호출한 thread의 정보가 들어가게 된다. 앞서 말한듯이 waiters list의 semaphore_elem인 waiter의 semaphore waiter list에 해당 thread가 들어가는 과정으로 실행된다. 이때 이미 condition waiters list에 이미 semaphore elem들이 존재한다면 각 semaphore의 waiters list에 접근하여 priority를 비교하여 condition waiters list 를 정렬해줘야 한다. 이때 앞서 사용했던 list_insert_ordered 함수를 이용하는데 parameter들에 맞추어서 sema_priority_compare 함수를 추가로 구현해줘야 한다. 위에서 말했듯이 condition waiters 의 각 semaphore에 접근하여 각 semaphore의 waiters list의 앞에 있는 thread의 priority를 가져와서 비교한다.

우선 마찬가지로 list_elem 꼴인 new와 now를 받아온다. new와 now는 condition waiters에서 elem값을 통해서 현재 위치한 노드를 알아내는 정보를 담고 있다. 앞서 자세히 설명하였다. new와 now를 통해서 condition waiters list의 semaphore_elem을 가져온다. 그것을 new_semaphore 와 now_semaphore로 받아오고 semaphore의 waiters

list에 접근하여 new_waiter와 now_waiter에 그 주소값을 저장한다. 그리고 이미 구현된 thread_priority_compare 함수를 이용해서 new_waiter와 now_waiter이 가장 앞에 있는 thread의 prioirty를 비교한다. 이미 waiter list는 정렬되어 있기 때문에 가장 앞 thread끼리만 비교해주면 된다.

이후 이 부분을 list_push_back 대신 넣어주어 wait 함수를 완성한다.

signal 함수 부분은 신호를 보내서 sema_up을 해주는 과정을 하기 전에 condition waiters list를 priority 순으로 재정렬 해주고 pop을 해주면 된다. 그리고 condition waiters list의 가장 앞에 있는 semaphore를 sema_up 해준다.

wait 도중에 해당 semaphore_elem을 만들고 스스로 semaphore의 waiters list에 들어간 wait을 호출한 thread가 sema_up 해준 semaphore의 waiters list에 있다면 해당 thread가 unblock 될것이고 sema value 가 1인 상태로 while문을 돌고 있는 sema_down 코드로 가서 while문을 탈출시켜주고 다시 sema value가 0이 될것이다.

이후 다시 자신이 갖고 있던 lock을 다시 갖게 된다.

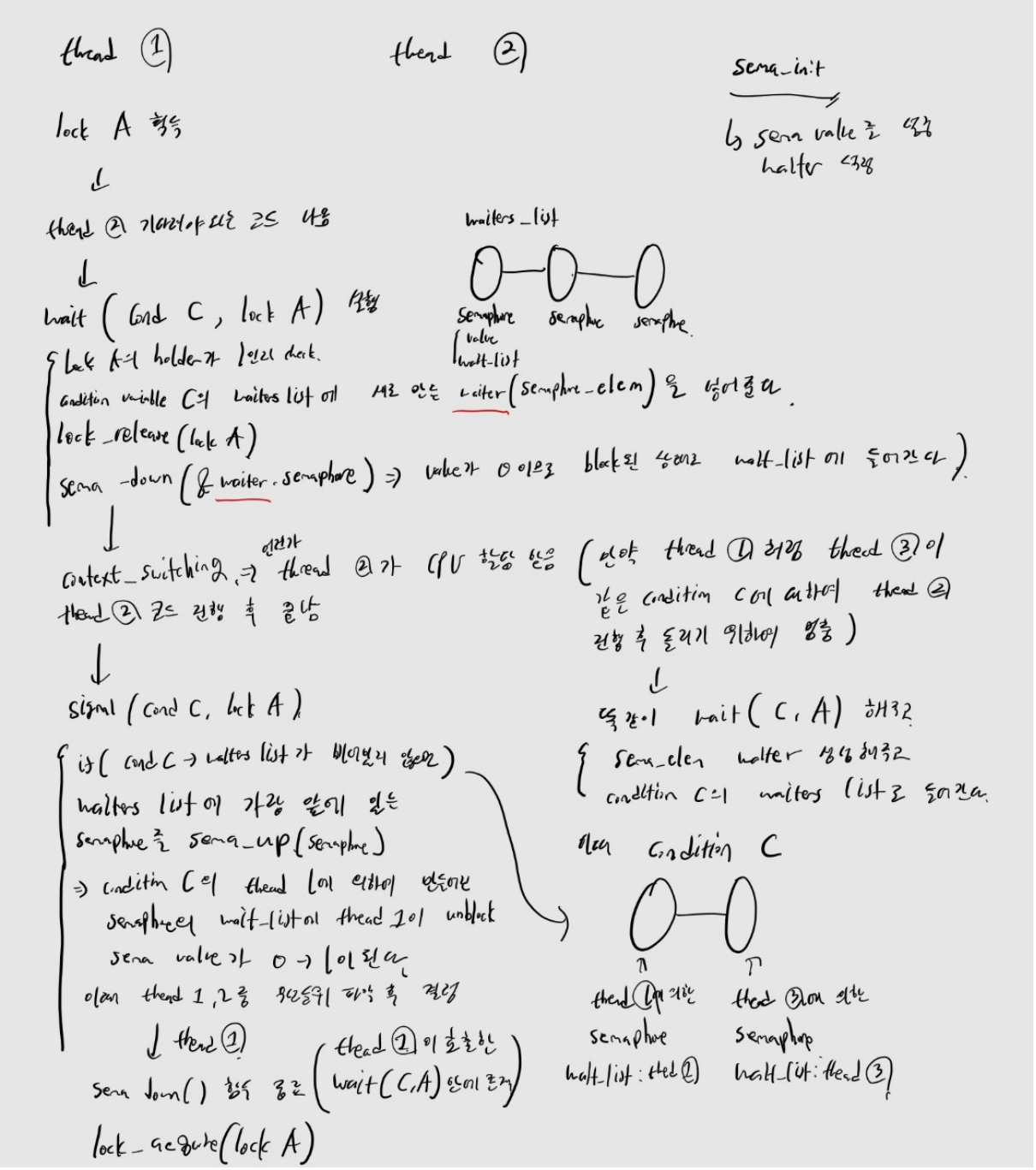
```
list_insert_ordered (&cond->waiters, &waiter.elem, sema_prior
```

```
// threads/sync.c
bool sema_priority_compare (const struct list_elem *a, cons
t struct list_elem *b, void *aux UNUSED){
   struct semaphore_elem *a_semaphore = list_entry (a, stru
ct semaphore_elem, elem);
   struct semaphore_elem *b_semaphore = list_entry (b, stru
ct semaphore_elem, elem);

   struct list *a_waiter = &(a_semaphore->semaphore.waiter
s);
   struct list *b_waiter = &(b_semaphore->semaphore.waiter
s);

  return thread_priority_compare(list_begin(a_waiter), list
_begin(b_waiter), 0);
}
```

```
// threads/sync.c
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());

  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
  list_insert_ordered (&cond->waiters, &waiter.elem, sema_p
riority_compare, 0);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}

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));

  if (!list_empty (&cond->waiters)) {
    list_sort (&cond->waiters, sema_priority_compare, 0);
    sema_up (&list_entry (list_pop_front (&cond->waiters),
struct semaphore_elem, elem)->semaphore);
  }
}

void cond_broadcast (struct condition *cond, struct lock *l
ock){
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
```

```
  while (!list_empty (&cond->waiters))
    cond_signal (cond, lock);
}
```

전체적인 conditino variable의 과정은 아래와 같다.



여기까지 코드를 작성하였을때 14 fail이 나오는 것까지 확인할 수 있다. 그러나 “priority-donate-~” test case에서 여전히 fail하기 때문에 추가적인 구현이 필요하다.

Priority donation

앞선 구현에서 문제가 생기는 이유는 priority가 낮은 thread가 lock을 소유하고 있을때 발생한다. design report에서 설명했듯 priority inversion 문제가 발생할 수 있고, 이는 priority donation이라는 방법을 이용해서 해결해야한다. 그렇기 때문에 우리는 lock을 acquire하고 release하는 과정에서 priority를 donate할 수 있는 코드를 구현해줘야한다.

우선 thread structure에 priority donation이 끝난 후 복귀하기 위한, original_priority, priority를 donate할 lock holder를 찾기 위한 wait_lock, 그리고 donation을 관리하기 위한 donation_list와 donation_elem을 추가해주었다. 그리고 init_thread 함수에서 새롭게 선언한 정보들을 함께 초기화해주었다.

```
// threads/thread.h
struct thread
{
    ...
  int original_priority;
  struct lock *wait_lock;
  struct list donation_list;
  struct list_elem donation_elem;
    ...
};

static void init_thread (struct thread *t, const char *nam
e, int priority){
    ...
  t->original_priority = priority;
  list_init (&t->donation_list);
  t->wait_lock = NULL;
    ...
}
```

가장 먼저 확인해야할 것은 `lock_acquire` 함수이다. lock_acquire 함수에서는 lock holder가 있는 경우 lock_acquire를 호출한 thread의 priority를 lock holder에게 donate 해주어야한다. 이를 구현하기 위해 우리는 먼저, `donation_list`를 정렬할 때 사용할 비교함수인 `donation_priority_compare` 를 생성해주었다. 그리고

자신의 priority를 lock holder에게 donate하는 donate_priority 함수를 생성해주었다.

따라서, 그 코드는 아래와 같다.

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

  ASSERT (!lock_held_by_current_thread (lock));
  struct thread *curr = thread_current ();

  if (lock->holder != NULL) {
    curr->wait_lock = lock;
    list_insert_ordered (&lock->holder->donation_list, &cur
r->donation_elem, donation_priority_compare, 0);
    donate_priority ();
  }
```

```
// threads/thread.c
bool donation_priority_compare (const struct list_elem *a,
const struct list_elem *b, void *aux UNUSED){
  int a_priority = list_entry (a, struct thread, donation_e
lem)->priority;
  int b_priority = list_entry (b, struct thread, donation_e
lem)->priority;

  return a_priority > b_priority;
}

void donate_priority (void){
  struct thread *curr = thread_current ();
  int limit = 0;

  while (curr->wait_lock != NULL && limit < 8) {
    struct thread *holder = curr->wait_lock->holder;
    holder->priority = curr->priority;
    curr = holder;
    limit++;
```

```
    }
  }
```

우선, lock_acquire 함수의 코드를 보면, 내가 원하는 lock을 아무도 소유하고 있지 않다면 기존의 sema_down을 호출하여 lock을 획득한다. 하지만 획득하고자하는 lock의 holder가 이미 있는 경우 진행하고 있는 thread의 priority를 lock을 소유하고 있는 thread에게 전달해야 한다. 현재 thread의 wait_lock에 lock을 넣어준다. 즉 내가 어떠한 lock을 기다리고 있다는 것을 알려주는 표시이다. 그리고 list_insert_ordered를 이용해서 lock_acquire를 호출한 thread를 donation_list에 추가한다. 조금 더 자세히 설명하면 lock holder, 즉 현재 thread가 소유하고자 하는 lock을 실제로 소유하고 있는 thread의 donation list에 가서 자신을 포함시킨다. 이때 마찬가지로 앞서 설명했듯이 elem을 통하여 비교해주고 donation list에 넣어주다. 자세한 설명은 위쪽에서 이미 설명했기 때문에 생략하겠다. 이후 위에서는 thread_compare_priority를 이용했는데 이번에는 같은 역할을 donation list에 대해서 진행해주는 donation_priority_compare 함수를 만들어주었다. donation_elem을 사용한 이유는 위에서는 ready_list, sleep_list, wait_list등을 list_elem등을 이용하여서 현재 thread 구조체 또는 semaphore 구조체 등을 표현하였다. 이제는 독립적인 donation list를 다룰 것이기 때문에 donation_elem을 사용해주었다. 마찬가지로 링크드 리스트 상에서 어디에 위치하는지 체크할 수 있는 길잡이가 된다.

donation_priority_compare를 조금 더 자세히 설명해보면 2개의 list_elem 값을 입력받아서 그 링크드 리스트에서 현재 내가 찾고자 하는 thread list가 존재하는 노드에 접근하고 최종적으로 donation_elem을 이요하여 해당 thread를 정확히 가져와준다. 각각의 thread를 가져오면 priority 값을 저장하고 비교해주고 결과값을 불 형식으로 리턴해준다.

그리고 priority donation을 수행하는 형태로 작성되었다. donate_priority는 말그대로 wait_lock을 결정해주었고 donation_list에 기부해준 thread도 넣어줬으니 마지막으로 실제로 lock을 소유한 thread에게 자신의 thread를 제공할 차례이다.

donate_priority 함수의 코드를 보면 nested donation의 형태를 구현하고 있음을 볼 수 있다. 이 부분에서 연쇄적으로 priority를 전달하는 부분에서 chain 형태로 물려있는 thread를 잘 연결해주지 않아서 해결하는데 시간이 오래걸렸다.

lock을 acqurie한 thread는 획득하고자하는 lock의 hodler에서 priority를 donate하고, 이 과정을 holder가 대기중인 lock이 없거나 최대 8번 반복하도록 작성되어있다. bottom-up 형태로 priority를 donate 해주기 때문에 lock을 여러개 갖고 있는 경우 자연스럽게 여러번 donate 받는 구조이므로 multiple donation도 구현되어있음을 확인할 수 있다.

좀 더 자세히 설명하면 thread A가 원하는 lock을 B가 가지고 있고 B가 원하는 lock을 C가 가지고 있다면 연쇄적으로 priority가 전달되어서 C는 A,B,C의 priority 중 가장 큰 값을 결과적으로 가질 수 있게 코드를 구현해야 한다.

그런데 여기서 holder의 priority가 current thread (lock을 acquire한)의 priority보다 큰 경우는 없는가에 대한 의문이 들었다. lock의 작동 과정을 생각해보면 donate_priority는 lock holder가 있는 경우에만 실행된다. 그런데, lock을 획득하는 것은 sema_down, sema_up에서 priority가 높은 순서대로 획득하도록 되어있다. 따라서,높은 priority를 갖는 thread가 lock_acquire를 호출하는 경우는 lock_hodler가 존재하지 않아 donate_priority가 호출되지 않는다. 그렇다면 donate_priority는 낮은 priority를 갖는 thread가 lock holder일 때, priority가 높은 thread가 생성되어 같은 lock을 acquire하는 경우 실행된다. 따라서, holder의 priority는 항상 donate_priority를 호출하는 thread의 priority보다 낮아지고, 그렇기 때문에 별도의 조건 없이 `holder->priority = curr->priority` 를 통해 priority를 donate할 수 있다.

이런식으로 코드를 구현하면 nested donation 문제를 해결할 수 있고 multiple donation 문제도 자연스럽게 해결된다.

그렇다면 이제는 donate 받은 priority가 lock_release를 통해 lock이 release 될 때 donate 받을 다른 priority가 있다면 donate 받고 없다면 original_priority로 되돌아가도록 구현해주어야한다. 우리는 먼저 donation list를 정리해줘야한다. 어떤 lock을 release하게 되면, 그 lock을 acquire하고 있는 thread들은 더이상 priority를 donation 해주지 않아도 된다. 그렇기 때문에 donation list에서 wait_lock이 release되는 lock과 같은 경우 donation_list에서 제거해줘야한다. 그래서 먼저 이 기능을 구현하는 `lock_release_helper` 함수를 구현하였다. 이 함수는 반복문을 이용해서 donation_list를 순회하면 `wait_lock == lock`인 thread들을 제거한다.

자세히 설명하면 lock A를 갖고 있던 thread가 lock을 release하게 된다면 lock A를 원해서 해당 thread에게 donate해줬던 thread들은 모두 빠져야 한다. 즉 그 thread들이 준 priority들은 취소되야 하고 당연히 donation list에도 빠져야 한다. lock_release_helper는 donation list를 먼저 관리해주는 함수이다.

이후에는 lock을 release하는 current thread의 priority를 다시 설정해줘야한다. 가장 먼저, priority를 original priority로 초기화시켜준다. 이를 통해 priority를 donation 받지 않은 원래의 priority를 확인해줄 수 있다. 이후에는 donation_list를 다시 priority 순서대로 정렬해준 뒤, 그 중 priority가 가장 높은 thread와 original priority를 비교하여 더 큰 값을 priority로 가질 수 있게 해주었다. donation_list를 priority로 정렬해줄때는 list_sort 함수와 donation_priority_compare 함수를 이용하여 먼저 donation list를 priority 순으로 재정렬해주었고, orignal priority와 donation list의 thread의 priority를 비교할 때는 max 함수를 사용해주었다. 추가적으로 max 함수는 int a, b를 비교해서 그 중 더 큰 값을 return하는 함수로, inline 함수로 작성되어 process들의 실행에 영향을 주지 않도록 하였다. 최종적으로 구현된 코드는 아래와 같다. 이때 이미 donation list상에는 release된 lock을 필요로 하여 donate 해준 thread들은 빠져있기 때문에 자신에게 다른 lock을 위하

여 기부한 thread들이나 자신의 original priority들 중 가장 큰 priority를 골라주게 된다. 즉 multiple donation 문제가 자연스럽게 해결된다.

마지막으로 현재 진행중인 thread의 priority가 변경될 수도 있다. thread_set_priority() 함수인데 이때 만약 변경된 priority가 donation list들로부터 받은 priority보다 크다면 이 부분에 대하여 변경된 priority로 바꿔주는 과정이 필요하다. 즉 priority_sort 함수를 호출하여 priority를 변경해주고 swap_for_update 함수를 호출하여 ready list와 비교하여야 한다.

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

  lock->holder = NULL;

  lock_release_helper (lock);
    priority_sort ();

  sema_up (&lock->semaphore);
}
```

```
// threads/thread.c
void lock_release_helper (struct lock *lock){
  struct list_elem *e;
  struct thread *curr = thread_current ();
  struct thread *t;
  for (e = list_begin (&curr->donation_list); e != list_end
(&curr->donation_list); e = list_next (e)){
    t = list_entry (e, struct thread, donation_elem);
    if (t->wait_lock == lock)
      list_remove (&t->donation_elem);
  }
}

void priority_sort (void){
  struct thread *curr = thread_current ();
```

```
  curr->priority = curr->original_priority;

  if (list_empty (&curr->donation_list))
    return;
  else {
    list_sort (&curr->donation_list, donation_priority_comp
are, 0);
    struct thread *front = list_entry (list_front (&curr->d
onation_list), struct thread, donation_elem);

    curr->priority = max(curr->priority, front->priority);
  }
}

static inline int max(int a, int b) {
    return (a>b) : a: b;
}


void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
  thread_current ()->original_priority = new_priority;

  priority_sort ();
  swap_for_update ();
}
```

Limitatioins

priority 만을 이용해서 scheduling을 하기때문에 priority가 낮은 thread들은 CPU를 점유하기 어렵기 때문에 average response time이 급격하게 증가하는 문제가 발생할 수 있다. priority donation이 구현되었기 때문에 priority가 높은 thread들이 synchornous variable들을 이용하는데에는 문제가 없지만 예외적인 경우들이 존재한다. 그리고 이는 프로그램의 average response time을 증가시키게 된다.

Advanced Scheduler

그러나 priority 만을 이용해서 scheduling을 하기때문에 priority가 낮은 thread들은 CPU를 점유하기 어렵기 때문에 average response time이 급격하게 증가하는 문제가 발생할 수 있다. 따라서, 우리는 multi-level feedback queue를 구현함으로서 priority를 실시간으로 조절하며 scheduler를 구현할 수 있었다.

design report를 보면 mlfqs scheduler를 구현할 때, 사용하는 nice, priority, recent_cpu, load_avg 변수 중 recent_cpu와 load_avg는 실수형인데 pintos에서 floating point 방식을 지원하지 않으므로 fixed point를 사용하고 fixed point의 연산 식을 편하게 사용하기 위해서 fixed_point.h라는 새로운 파일을 생성하고 거기에 연산 함수들을 작성해주었다. 코드는 아래와 같다. 코드의 구현은 pintos.pdf의 설명을 참고해서 구현하였다.

이전에 nice와 recent_cpu 값은 thread별로 가지는 값이기 때문에 thread 구조체에 추가해주었고 load_avg는 공통된 하나의 값으로 전역변수로 설정해주었다.

thread 구조체에 2가지가 추가되었다. 따라서 init_thread에서 thread 구조체 내용들을 초기화 해줄 때 각각을 default값으로 초기화 해주었다.

load_avg값은 threa_start함수에서 default값으로 초기화 해주었다.

init.c의 parse_options를 보면 사용자가 mlfqs 옵션을 입력하면 thread_mlfqs값이 true로 바뀌는 것으로 설정되어있다.

```
struct thread
{
   /* Owned by thread.c. */
   tid_t tid;                          /* Thread identifier.
   enum thread_status status;          /* Thread state. */
   char name[16];                      /* Name (for debugging
   uint8_t *stack;                     /* Saved stack pointer
   int priority;                       /* Priority. */
   int64_t wakeUp;// 1                    /* Wake-Up. */
   int nice;//3
   int recent_cpu;//3
   struct list_elem allelem;           /* List element for al

   /* Shared between thread.c and synch.c. */
   struct list_elem elem;              /* List element. */

   /* priority inversion */
```

```
    int original_priority;
    struct lock *wait_lock;
    struct list donation_list;
    struct list_elem donation_elem;
    ///////////////////////

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

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

int load_avg;


init_thread (struct thread *t, const char *name, int priority
{
  enum intr_level old_level;

  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;

  /* mlfqs s*/
  t->nice = 0;
  t->recent_cpu = 0;

  thread_start (void)
```

```
{
  /* Create the idle thread. */
  struct semaphore idle_started;
  sema_init (&idle_started, 0);
  thread_create ("idle", PRI_MIN, idle, &idle_started);
  // load_avg = 0;
  /* Start preemptive thread scheduling. */
  intr_enable ();
  load_avg = 0;
  /* Wait for the idle thread to initialize idle_thread. */
  sema_down (&idle_started);
}
```

```
bool thread_mlfqs;
else if (!strcmp (name, "-mlfqs"))
        thread_mlfqs = true;
```

The following table summarizes how fixed-point arithmetic operations can be implemented in C. In the table, `x` and `y` are fixed-point numbers, `n` is an integer, fixed-point numbers are in signed $p.q$ format where $p + q = 31$, and `f` is `1 << q`:

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`

```
// threads/fixed_point.h
#define F (1 << 14)
#define INT_MAX ((1 << 31) - 1)
#define INT_MIN (-(1 << 31))

int int_to_fp (int n) {
  return n * F;
}

int fp_to_int (int x) {
  return x / F;
}

int fp_to_int_round (int x) {
  if (x >= 0) return (x + F / 2) / F;
  else return (x - F / 2) / F;
}

int add_fp (int x, int y) {
  return x + y;
}

int sub_fp (int x, int y) {
  return x - y;
}

int add_mixed (int x, int n) {
  return x + n * F;
}

int sub_mixed (int x, int n) {
  return x - n * F;
}

int mult_fp (int x, int y) {
  return ((int64_t) x) * y / F;
}
```

```
int mult_mixed (int x, int n) {
  return x * n;
}

int div_fp (int x, int y) {
  return ((int64_t) x) * F / y;
}

int div_mixed (int x, int n) {
  return x / n;
}
```

이제는 priority를 계산하기 위해서 사용되는 변수들을 선언하고 그 변수들의 값을 관리해주는 함수들을 작성해줘야한다. 사용되는 변수들과 식을 다시한번 살펴보면 아래와 같다.

priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)

recent_cpu = (2 * load_avg) / (2 * load_avg + 1) * recent_cpu + nice

load_average = (59/60) * load_avg + (1/60) * ready_threads

이제 mlfqs_calculate_priority, mlfqs_calculate_recent_cpu, mlfqs_calculate_load_avg 함수를 사용해서 위의 변수들을 식으로 변환해주었다. 작성한 코드는 아래와 같다.

```
// threads/thread.c
void mlfqs_calculate_priority (struct thread *t){
  if (t == idle_thread)
    return ;

  int a = div_mixed(t->recent_cpu, 4);
  int b = 2 * t->nice;
  int tmp1 = add_mixed(a, b);
  int tmp2 = sub_mixed(tmp1, (int)PRI_MAX);
  int pri_result = fp_to_int(sub_fp(0, tmp2));

  if (pri_result < PRI_MIN)
      pri_result = PRI_MIN;
  if (pri_result > PRI_MAX)
      pri_result = PRI_MAX;
  t->priority = pri_result;
```

```
}

void mlfqs_calculate_recent_cpu (struct thread *t){
  if (t == idle_thread)
    return ;

  int a = mult_mixed (load_avg, 2);
  int b = add_mixed(a, 1);
  int tmp1 = div_fp(a, b);
  int tmp2 = mult_fp(tmp1, t->recent_cpu);
  int result = add_mixed(tmp2, t->nice);

  t->recent_cpu = result;
}

void mlfqs_calculate_load_avg (void){
  int ready_threads;
  if (thread_current () == idle_thread)
    ready_threads = (int) list_size (&ready_list);
  else
    ready_threads = (int) list_size (&ready_list) + 1;


  int a = div_fp(int_to_fp(59), int_to_fp(60));
  int b = div_fp(int_to_fp(1), int_to_fp(60));

  int tmp1 = mult_fp(a, load_avg);
  int tmp2 = mult_mixed(b, ready_threads);

  int result = add_fp(tmp1, tmp2);

  load_avg = result;
}
```

Nice : [-20, 20] — 정수
Priority : [0, 63] = PRI_MAX − (recent_cpu/4) − (nice × 2) — 4tick
Recent_CPU : 실수값 = (2 × load_avg)/(2 × load_avg + 1) × recent_cpu + nice — 1초마다 — 실수
load_avg = (59/60) × load_avg + (1/60) × ready_threads, — 1초마다 — (전역 변수)

1 | 17 | 14.

F : 1<<14 , x, y : fixed_point 실수 , n = 정수

int_to_fp : n × F

fp_to_int : x/F

fp_to_int_round : (x+F/2)/F   F/2 : 0.5   ∴ 1/2 더하고 F로 나눔 (버림) ∴ 반올림

add_fp : x+y

sub_fp : x−y

add_mixed x+n·F

sub_mixed x−n×F

mult_fp ((int64)x)×y/F : overflow 방지하려고 x를 64bit로 바꿔주고 곱하고 F로 나눔 (x만 32로 연산)

mult_mixed x·n

div_fp : ((int64)x)·F/y :

div_mixed : x/n

① mlfqs_calculate_priority
= PRI_MAX − (recent_cpu/4) − nice × 2,

a = div_mixed (t→recent_cpu, 4) ⇒ recent_cpu = 실수 ∴ div_mixed 이용

b = 2 × t→nice ⇒ nice × 2 계산

tmp1 = add_mixed (a, b) ⇒ recent_cpu/4 (fixed_point) + 2×nice (정수) 계산 ∴ add_mixed 이용

tmp2 = sub_mixed (tmp1, (int) PRI_MAX) ⇒ 아래 가리 → 먼저 입력 받은 Parameter가 fixed_point여서 어쩔 수 없이 거꾸로 했음.

즉 tmp2는 − priority 값 가정

pri_result = fp_to_int (sub_fp(0, tmp2)) ⇒ 다시 원래 priority 값을 sub_fp로 구하고 실제 계산할 때는 int가 필요해서 fp → int로 변환

이후 pri_result가 PRI_MIN 보다 작으면 PRI_MIN, Max 크면 Max ⇒ 최종 priority 결정

② mlfqs_calculate recent_cpu
= (2×load_avg)/(2×load_avg+1) × recent_cpu + nice   (current_thread가 idle thread면 바로 return)

a = mult_mixed (load_avg, 2)   load_avg는 실수라 fp ∴ mult_fixed 이용

b = add_mixed (a, 1)   a는 fp , 2×load_avg+1 값 계산

tmp1 = div_fp(a, b)   a를 b로 나눔 , 둘 다 fp

tmp2 = mult_fp(tmp1, recent_cpu)   tmp1 × recent_cpu , 둘 다 fp.

result = add_mixed (tmp2, nice)   최종 recent_cpu 구하기   tmp2 = fp, nice = 정수 ∴ add_mixed 이용

③ mlfqs_calculate_load_avg
= 59/60 × load_avg + 1/60 × ready_threads

추가 구현 : ready_threads (int 형식) . (i) current thread : idle ⇒ ready_threads = (int) list_size (&ready_list)
(ii) ″ not idle ⇒ ″ +1 (∵ running thread 포함)

a = div_fp(int_to_fp(59) , int_to_fp(60))   (정수 ÷ 정수 X ∴ 둘 다 fp로 바꿔서 나눠줌)
b = ″ (1) (60)

∴ a, b는 59/60 , 1/60 의 fixed point 연산

tmp1 = mult_fp(a, load_avg)

tmp2 = mult_mixed (b, ready_threads)   ready_threads는 int ∴ mult_mixed 이용

result = add_fp (tmp1, tmp2)

다음으로는 각 값들의 변해야할 때, 이를 실행해불 수 있는 코드를 살펴보아야한다. 그 중 가장 먼저 확인해야할 것은 recen_cpu의 update이다. running thread가 CPU를 소유하고 있는 동안 recent_cpu는 1 tick에 1씩 증가하게 된다. 우리는 이 함수를 `mlfqs_increment_recent_cpu`로 구현해주었다. 코드의 구현을 살펴보면 idle thread가 아닌 경우, current thread의 recent_cpu 값이 1 증가하게 된다.

```
void mlfqs_increment_recent_cpu (void){
  if (thread_current () != idle_thread)
    thread_current ()->recent_cpu = add_mixed (thread_curre
nt ()->recent_cpu, 1);
}
```

그 다음으로 mlfqs scheduler는 4 tick 마다 priority를 갱신 해주게 된다. 그렇기 때문에 우리는 4초마다 priority를 재계산 해주는 `mlfqs_recalculate_priority` 함수를 작성해주었다. 이를 통해서 모든 thread의 priority를 재계산해주었기 때문에, 우리는 for문을 사용해서 모든 thread가 저장되어있는 all_list를 순회하면 priority를 계산하는 `mlfqs_calculate_priority` 함수를 호출해주게 된다.

마지막으로 우리는 1초 (100 tick)마다 모든 thread의 recent_cpu와 load_avg 값을 갱신해줘야한다. 따라서 우리는 `mlfqs_recalculate_recent_cpu` 함수를 추가적으로 설정해주었다. `mlfqs_recalculate_recent_cpu` 함수는 `mlfqs_recalculate_priority` 와 같이 all_list를 순회하면서 `mlfqs_calculate_recent_cpu` 함수를 호출하도록 구현되었다. 그리나 load_avg는 thread 하나에 대한 값이 아니라 시스템의 단위로 살펴봐야하는 변수이기 때문에 따로 recalaultate 함수를 구현해주지 않고 100 tick 마다 `mlfqs_calculate_load_avg` 함수를 호출해주면 된다.

```
void mlfqs_recalculate_recent_cpu (void){
  struct list_elem *e;

  for (e = list_begin (&all_list); e != list_end (&all_lis
t); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allele
m);
    mlfqs_calculate_recent_cpu (t);
  }
}

void mlfqs_recalculate_priority (void){
  struct list_elem *e;
```

```
  for (e = list_begin (&all_list); e != list_end (&all_lis
t); e = list_next (e)) {
    struct thread *t = list_entry (e, struct thread, allele
m);
    mlfqs_calculate_priority (t);
  }
}
```

이 함수들을 실행해주기 위해서는 tick 마다 interrupt를 보내 변수들을 계산해줄 수 있어야 한다. 그래서 우리는 `timer_interrupt` 함수를 이용해서 위의 함수들을 호출해줄것 이다. 먼저 생각해줘야할 것은 mlfqs 함수들의 호출은 `-mlfqs` 명령어를 사용해주었을 때만 호출되어야하는 것이다. thread.h 코드를 살펴보면 `thread_mlfqs` 변수를 이용해서 mlfqs scheduler의 실행 여부를 확인해주는 것을 볼 수 있다. 따라서, thread_mlfqs가 참인 경우 4 tick 마다 `mlfqs_recalculate_priority` priority를 갱신해준다. 그리고 1초마다 `mlfqs_calculate_load_avg`, `mlfqs_recalculate_recent_cpu`를 호출해주어야한다. 그리고 recent_cpu를 계산하는데, load_avg가 사용이 되기 때문에 load_avg 값을 먼저 갱신해줘야한다. 변경해준 timer_interrupt 함수는 아래와 같다.

```
// threads/thread.h
/* If false (default), use round-robin scheduler.
   If true, use multi-level feedback queue scheduler.
   Controlled by kernel command-line option "-o mlfqs". */
extern bool thread_mlfqs;
```

```
//devices/tiemr.c
static void timer_interrupt (struct intr_frame *args UNUSE
D){
  ticks++;
  thread_tick ();

  if (thread_mlfqs){
    mlfqs_increment_recent_cpu();

    if (timer_ticks() % 4  == 0) {
      mlfqs_recalculate_priority();

      if (timer_ticks () % 100 == 0){
```

```
        mlfqs_calculate_load_avg();
        mlfqs_recalculate_recent_cpu ();
        }
    }
  }

  thread_awake (ticks);
}
```

다음으로 해주었던 것은 mlfqs가 실행될 때, priority donation이 가 실행되지 않도록 하는 것이다. 따라서, `lock_acquire`, `lock_release`, `thread_set_priority` 함수에서 `thread_mlfqs` 가 참일 경우 priority donation이 실행되지 않도록 해야한다. thread_mlfqs가 참일 경우 lock_acquire의 `donate_priority`, lock_release의 `lock_release_helper`, `priority_sort` 함수가 실행되지 않아야한다. 그리고 이제 priority를 실시간으로 계산해주게 되므로 `thread_set_priotity`가 실행되지 않아야한다. 변경된 코드는 아래와 같다.

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

  ASSERT (!lock_held_by_current_thread (lock));
  struct thread *curr = thread_current ();

  if (lock->holder != NULL) {
    curr->wait_lock = lock;
    list_insert_ordered (&lock->holder->donation_list, &cur
r->donation_elem, donation_priority_compare, 0);
    if (!thread_mlfqs)
      donate_priority ();
  }

  sema_down (&lock->semaphore);
  curr->wait_lock = NULL;
  lock->holder = curr;
}

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

```
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  lock->holder = NULL;

  if (!thread_mlfqs){
    lock_release_helper (lock);
    priority_sort ();
  }

  sema_up (&lock->semaphore);
}
```

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

  thread_current ()->priority = new_priority;
  thread_current ()->original_priority = new_priority;

  priority_sort ();
  swap_for_update ();
}
```

이제 `thread_set_nice`, `thread_get_nice`, `thread_get_load_avg`, `thread_get_recent_cpu` 함수들을 완성해줘야한다.

가장 먼저 `thread_set_nice` 함수를 작성해주었다. 현재 thread의 nice 값을 변경해주는 함수로, interrupt를 비활성화한 뒤, thread의 nice 값 변경해준다. 그리고 변경된 nice 값을 이용해서 priority를 다시 계산해주고 priority가 변경 되었으므로 `swap_for_update` 함수를 호출해서 current thread의 priority가 ready_list의 thread의 priority 보다 클 경우 thread yield가 호출될 수 있게 해주었다. 작성된 코드는 아래와 같다.

```
void thread_set_nice (int nice UNUSED){
  enum intr_level old_level = intr_disable ();
  thread_current ()->nice = nice;
  mlfqs_calculate_priority (thread_current ());
  swap_for_update ();
```

```
  intr_set_level (old_level);
}
```

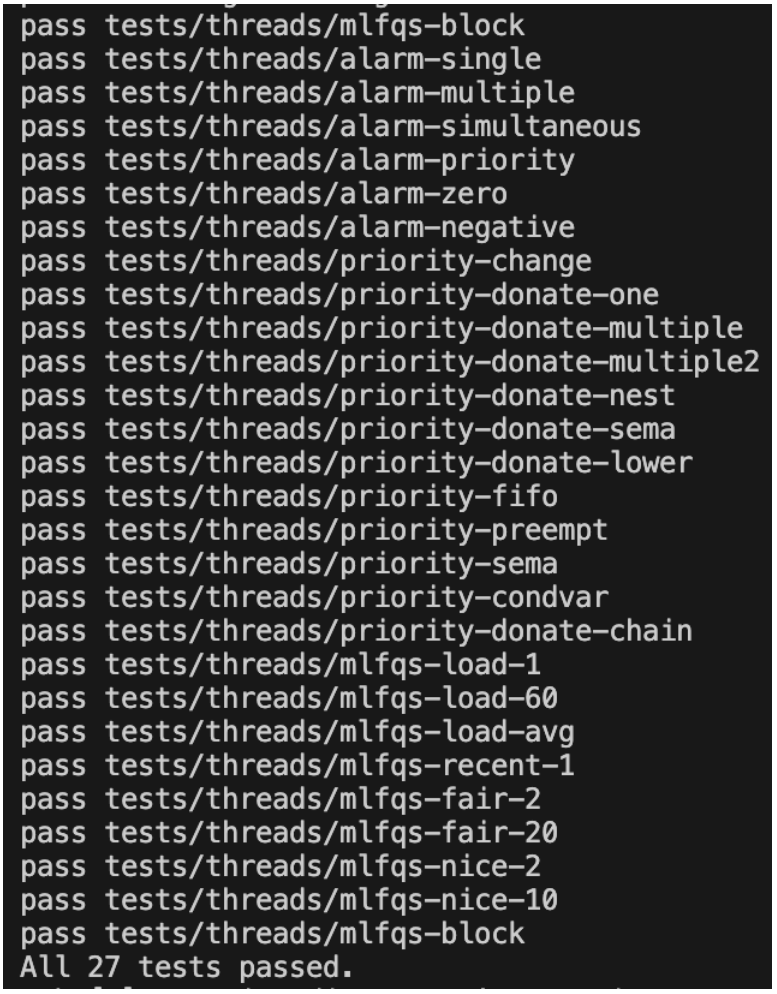
thread_get_nice 함수는 현재 thread의 nice 값을 return 해주는 함수이다. 또한, get 하는 중에 값이 바뀌는 것을 막기 위해 interrupt를 중단 시켜야한다. 작성된 코드는 아래와 같다.

```
int thread_get_nice (void){
  enum intr_level old_level = intr_disable ();
  int nice = thread_current ()-> nice;
  intr_set_level (old_level);
  return nice;
}
```

`thread_get_load_avg`와 `thread_get_recent_cpu`는 각각 `load_avg`와 `recent_cpu`에 100을 곱한 값을 return 해주는 함수이다. 따라서, 실수 값인 laod_avg와 recent_cpu를 `mult_mixed`를 이용하여 100을 곱해준 뒤 `fp_to_int_round`를 이용해 정수로 만들어 return 해주면 된다. 그리고 추가적으로 값을 가져오는 동안 interrupt를 막아줌으로서 get 하는 중에 값이 바뀌는 것을 막아주었다. 작성된 코드는 아래와 같다.

```
int thread_get_load_avg (void){
  enum intr_level old_level = intr_disable ();
  int load_avg_value = fp_to_int_round (mult_mixed (load_av
g, 100));
  intr_set_level (old_level);
  return load_avg_value;
}

int thread_get_recent_cpu (void){
  enum intr_level old_level = intr_disable ();
  int recent_cpu= fp_to_int_round (mult_mixed (thread_curre
nt ()->recent_cpu, 100));
  intr_set_level (old_level);
  return recent_cpu;
}
```

이제 make check 해보면 27개의 test를 모두 통과하는 것을 확인할 수 있다.



Limitaions

Advanced scheduler의 경우 priority scheduling을 효과적으로 구현하고 균등한 CPU 할당이 가능하게 해주지만, priority와 그를 계산하기 위한 변수들을 계산하고 interrupt를 발생시킬때 overhead가 많이 생길 수 있다는 문제가 존재한다.

Discussion

본 과제를 통해 pintos의 구현을 살펴보고 직접 scheduler도 개선해보면서 priority scheduling에 대해 잘 이해할 수 있었고, thread들이 shared object에 접근할 때 그것들이 어떻게 관리 되어야하는지를 코드의 구현을 통해서 보다 자세히 이해할 수 있었다.