Pintos Project: 1. Threads

Final Report – team 30

20210054 정하우

20210716 최대현

Context

1. **Implementation**

   1-1. Alarm Clock

   - Algorithm

   - Data Structure

   - Function Implementation

   1-2. Priority Scheduler

   1-2-1. Priority Scheduler

   - Algorithm

   - Data Structure

   - Function Implementation

   1-2-2. Priority Donation

   - Algorithm

   - Data Structure

   - Function Implementation

   1-3. Advanced Scheduler

   - Algorithm

   - Data Structure

   - Function Implementation

1-1. Alarm Clock

Algorithm

- Alarm Clock의 목적은 기존에 busy-waiting 방식으로 구현되어 있던 Pintos의 Alarm Clock 기능을 수정해 기능을 향상시키는 것이다. 이를 위해 기존의 baseline 코드에서 다음과 같은 구현이 필요하다.

  1. thread struct에 일어날 시간인 wake_up_tick 변수를 추가한다.
  2. sleep state의 thread를 관리하는 sleep_list 를 추가한다.
  3. 이제 thread를 sleep state로 변경하는 thread_sleep()함수를 구현한다. 이 함수에선 thread에 일어날 시간을 저장하고, block state로 state를 변경하며, sleep_list에 해당 thread를 오름차순으로 추가한다.

     - sleep_list를 오름차순으로 정렬 되로록 하여 thread_awake()함수에서 sleep_list를 돌 때 깨어날 시간이 되지 않은 thread를 만나는 순간 검사를 종료하여 성능을 향상시켰다.

  4. 위 thread_sleep()함수는 timer_sleep()함수에서 호출되어 thread를 바로 ready state 가 아닌 block state 로 변경하도록 구현한다.
  5. 이제 thread가 일어날 시간이 되었는지 확인하는 thread_awake()함수를 구현한다. 매 tick마다 실행이 되도록 timer_interrupt()에 함수가 호출이 될 수 있도록 한다.

Data Structure

1. **thread.h/struct thread/int64_t wake_up_tick**

```
struct thread

  {
  /*…*/
    int64_t wake_up_tick;
  /*…*/
}
```

- thread가 일어날 시간에 대한 정보를 담는 변수를 thread struct에 추가한다.

2. **thread.c/static struct thread sleep_list**

```
static struct list sleep_list;
```

```
/*…*/

void
thread_init (void)
{
/*…*/
  list_init(&sleep_list);
/*…*/
}
```

- sleep state thread만을 관리하는 list를 추가한다. 다른 thread를 관리하는 list와 마찬가지로 thread_init()함수에서 초기화시켜준다.

Function Implementation

1. **timer.c/void timer_sleep (int64_t ticks)**

```
void

timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  // while (timer_elapsed (start) < ticks)
  //   thread_yield ();
  thread_sleep(start+ticks);
}
```

- 기존 busy-waiting 방식으로 구현되어 있는 while문 대신 thread_sleep()함수를 추가해 일어나야 할 thread만 처리할 수 있도록 구현한다.

2. **thread.c/bool compare_wake_up_tick(struct list_elem *new_elem, struct list_elem *sleep_list_elem, void *aux)**

```
bool

compare_wake_up_tick(struct list_elem *new_elem, struct list_elem
*sleep_list_elem, void *aux)
{
```

```
  return list_entry(new_elem,struct thread,elem)-
>wake_up_tick<list_entry(sleep_list_elem,struct thread,elem)->wake_up_tick;
}
```

- sleep_list를 오름차순 정렬하기 위한 함수를 구현하였다.

3. **thread.c/void thread_sleep(int64_t ticks)**

```
void

thread_sleep (int64_t ticks)
{
  struct thread *cur_t=thread_current();
  ASSERT(cur_t!=idle_thread);
  enum intr_level old_level=intr_disable();

  cur_t->wake_up_tick=ticks;
  list_insert_ordered(&sleep_list, &cur_t->elem, compare_wake_up_tick, NULL);
  thread_block();

  intr_set_level(old_level);
}
```

- 재워야 할 thread에 일어날 시간을 저장하고, sleep state thread를 관리하는 sleep_list에 thread를 오름차순으로 추가한다.

4. **timer.c/static void timer_interrupt (struct intr_frame, *args UNUSED)**

```
static void

timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  /*…*/

  thread_awake (ticks);
}
```

- 매 tick마다 일어나는 interrupt 함수에서 깨워야할 thread가 있는지 확인하는 thread_awake()함수를 추가한다.

5. **thread.c/void thread_awake(int64_t ticks)**

```
void
thread_awake(int64_t ticks)
{
  struct list_elem *sleep_list_elem;

  for(sleep_list_elem=list_begin(&sleep_list);sleep_list_elem!=list_end(&sleep
_list);)
  {
    struct thread *cur_t=list_entry(sleep_list_elem,struct thread, elem);
    if(ticks<cur_t->wake_up_tick)
    {
      break;
    }
    sleep_list_elem=list_remove(sleep_list_elem);
    thread_unblock(cur_t);
  }
}
```

- thread가 일어날 시간이 됐는지 확인하고 일어나야할 thread는 ready state로 변경한다. sleep_list에서 thread를 지우고 thread_unblock()함수를 통해 ready state로 변경한다. sleep_list는 오름차순 정렬이 되어 있으므로 아직 일어날 시간이 안된 thread를 만나면 그대로 반복문을 나가도 된다.

1-2. Priority Scheduler

1-2-1. Priority Scheduler & Synchronization

Algorithm

- Priority Scheduler의 목적은, 기존에 Round-Robin 방식으로 구현되어있던 Pintos의 thread scheduler을 thread 간의 우선순위 (Priority)를 고려한 Scheduling 방식으로 다시 구현하는 것이다. 이를 위해 기존의 baseline 코드에서 다음과 같은 구현이 필요하다.

1. thread_create() 함수에서 thread를 생성한 뒤 ready_list라는 data structure에 넣어줄 때, push_back()으로 해당 list 맨 뒤에 넣어주던 구현을 Priority 내림차순으로 넣어주는 구현으로 변경한다.

2. thread_yield() 함수와 thread_unblock() 함수에서도 유사하게 thread를 단순히 push_back으로 넣어주던 구현을 Priority 내림차순으로 넣어주는 구현으로 변경한다.

3. ready_list에 thread를 삽입할 때, 즉 ready_list에 원소가 추가될 때마다 현재 실행되고 있

는 thread와 우선순위를 비교하여 필요하다면 running_thread를 교체한다.

4. synchronization과 관련된 lock, semaphore, condition variable들에 대해서도 priority를 고려한 scheduling을 구현한다. 즉, semaphore을 기다리고 있는 thread들의 list가 내림차순으로 항상 유지되게 구현하여 간단한 pop front 연산으로 thread를 꺼낼 수 있게 하고, condition variable에서의 waiter list에서도 input semaphore의 waiter을 적절한 위치에 배치하는 등의 구현이 필요하다.

Data Structure

- 별도로 추가해주어야 하는 data structure는 없다. 다만 기존에 thread structure 속 구현되어 있었던 priority를 활용하기로 한다.

Discussion

- 디자인할 당시에는 list_push_back 함수 대신 list_push_priority 함수를 호출하도록 구현을 계획했다. 하지만, list.c에 list_insert_ordered 함수가 있어서 이 함수에 넣어줄 함수를 디자인하고, 우선순위 기반 삽입은 구현된 함수를 사용하는 것이 낫다고 판단하여 구현 방법을 변경했다. 즉 list_push_priority 함수를 직접 짜서 사용하는 것보다 같은 역할을 하는 list_insert_ordered 함수를 적절히 활용하는 것이 낫다 판단하고 이를 적용했다. 디자인 단계에서 list_push_priority 함수를 호출하도록 디자인한 것들을 모두 list_insert_ordered를 호출하도록 변경해주면 된다.
- 디자인할 당시 현재 thread_set_priority 함수의 동작에 의해 현재 실행되고 있는 thread의 priority가 더 이상 최고가 아닐 수 있으므로, 이를 확인하고 priority_yield 함수를 호출하도록 수정한다는 추상적인 디자인이 있었는데, 이 과정을 명료하게 thread_priority_test 함수로 구현하고 반복적인 작업에 활용하였다.

Function Implementation

Thread Scheduling

thread.c

1. **thread.c/priority_compare(struct list_elem* t1, struct list_elem* t2, void* aux UNUSED)**

```
bool // called by list.c/list_insert_ordered
priority_compare(struct list_elem* t1, struct list_elem* t2, void* aux UNUSED)
{
  struct thread *thread1 = list_entry(t1, struct thread, elem);
```

```
  struct thread *thread2 = list_entry(t2, struct thread, elem);
  return thread1->priority > thread2->priority;
}
```

- 구현 상으로 필요하다고 생각하여 새롭게 구현한 함수이다. list.c에서 list_insert_ordered 라는 함수를 활용하기 위해서 구현했다. list_insert_ordered 함수에서는 input argument 인 elem이 argument로 들어오는 list를 순회하면서 얻는 e라는 원소보다 더 작은지를 검사하는 less function을 input으로 받는다. 따라서 이 less function을 우리가 원하는 형태로, 즉 어떤 것을 기준으로 비교할지를 디자인해서 list_insert_ordered 함수의 argument로 넣어주면 priority scheduler에 필요한 scheduling 부분을 쉽게 구현할 수 있다. 해당 함수의 경우 thread의 priority를 비교하기 위해서 구현한 함수이다. 후에 비슷한 예시로 list_insert_ordered 함수에 적용하기 위해 donated_priority를 비교하거나 semaphore의 원소를 비교하는 함수들도 구현했다.

-

2. thread.c/void thread_unblock (struct thread *t)

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

- 기존에 구현되어 있었던 알고리즘을 개선한 함수이다. Blocked state에 있던 thread를 input으로 받아서 ready state로 바꿔주는 함수인데, 이전의 thread_unblock 함수의 경우 단순히 해당 thread를 ready state로 바꾼 뒤 ready list에 push_back해주고 있었다. 그러나, priority scheduling을 해주어야 하므로 앞서 구현한 priority_compare함수를 argument로 갖는 list_insert_ordered 함수를 호출하여 적절한 위치에 thread를 삽입하게끔 구현했다.

3. **thread.c/void thread_yield (void)**

```
void
thread_yield (void)
{
  struct thread *t_cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (t_cur != idle_thread)
    // list_push_back (&ready_list, &cur->elem);

    // push_back 이 아니라 ready_list 에 order 을 지켜서 삽입하도록 코드 수정.
    list_insert_ordered (&ready_list, &t_cur->elem, priority_compare, NULL);
// less_func
  t_cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

- 현재 실행되고 있는 thread를 ready_list에 넣어주는 함수이다. 비슷한 이유로 기존에는 단순히 현재 thread를 멈추고 ready list의 맨 뒤에 넣어주었지만 이를 list_insert_ordered 함수와 priority_compare 함수를 통해 적절한 위치에 삽입되도록 구현했다.

4. **thread.c/void thread_priority_test(void)**

```
void // compare current thread's priority & front thread of ready list
thread_priority_test(){

  if(list_empty(&ready_list) == true) return; // exception: if ready list is
empty

  struct thread* p_max_thread = list_entry(list_front(&ready_list), struct
thread, elem);
  if (p_max_thread-> priority > thread_current()->priority) thread_yield();

  return;
}
```

- 디자인 때 고려했던 사항대로, running thread의 priority가 바뀌는 예외를 처리한다. 이 때 바뀐 priority가 ready_list의 front (즉, 가장 높은 priority를 갖고 있는 thead)보다 낮다면 점유를 넘겨주어야 하고, 해당 부분을 구현한 함수이다. ready_list에 대해 list_front 함수를 호출하여 p_max_thread를 불러오고, 이를 current thread와 비교하여 priority가 더 크다면 thread_yield 함수를 호출한다. 이 때, running thread의 priority가 바뀔 수 있는 경우는 thread_create() 함수와 thread_set_priority() 함수가 호출될 때이다. 따라서 해당 함수들을 호출할 때 위에서 구현한 thread_priority_test() 함수를 호출하도록 한다.

5. **thread.c/tid_t thead_create(const char *name, int priority, thread_func *function, void *aux)**

```
tid_t
thread_create (const char *name, int priority, // PR_DEFAULT = 31
               thread_func *function, void *aux)
{
  tid_t tid; // thread_id

  ASSERT (function != NULL);

  .
  .
  . (omitted)

  /* Add to run queue. */
  thread_unblock (t);
  thread_priority_test();

  return tid;
}
```

- 앞서 언급한대로 thread_unblock() 함수 아래 thread_priority_test() 함수를 호출한다. 이를 통해, running thread의 priority가 바뀌는 예외를 처리해줄 수 있다.

6. **thread.c/void thread_set_priority (int new_priority)**

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

```
  .
  .
  . (changed by priority donation)
  thread_priority_test();
}
```

- 앞서 언급한대로 thread_priority_test() 함수를 호출한다. 이를 통해, running thread의 priority가 바뀌는 예외를 처리해줄 수 있다.

Synchronization (semaphore, lock, conditional variable)

synch.c

1. **synch.c/void sema_down(struct semaphore *sema)**

```
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
      list_insert_ordered (&sema->waiters, &thread_current ()->elem,
priority_compare, 0);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

- 앞서 thread 관련 함수를 호출할 때도 round-robin 방식의 scheduler가 기본적으로 구현되어있는 Pintos의 구현 상 ready state의 thread를 ready list에 넣을 때 list_push_back 함수를 호출하여 ready list의 맨 뒤에 넣어주는 경우가 많았다. 마찬가지로, semaphore의 진입을 관리하는 sema_down에서도 기존의 list_push_back 함수를 이용한 구현을 list_insert_ordered와 compare_priority 함수를 이용한 구현으로 바꿔준다.

2. **synch.c/void sema_up(struct semaphore *sema)**

```
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, priority_compare, 0);
    thread_unblock (list_entry (list_pop_front (&sema->waiters),
                                struct thread, elem));
  }
  sema->value++;
  thread_priority_test();
  intr_set_level (old_level);
}
```

- waiters list에 있는 thread들의 priority가 변했을 수 있으므로 list.c에 있는 list_sort 함수를 통해 input argument로 들어오는 semaphore의 waiter list를 priority 내림차순으로 정렬해준다. 그 뒤, thread_unblock 함수를 실행하기 때문에 그 결과로 unblock된 thread가 running thread보다 우선순위가 높은 상황을 고려하여 thread_priority_test 함수를 호출해준다.

  c.f) Lock의 출입과 관련된 함수인 lock_acquire과 lock_release 함수의 경우 내부에서 struct 멤버인 semaphore을 그대로 적용하기 때문에 바뀐 sema_down과 sema_up 함수를 그대로 적용할 수 있어서 변경해줄 필요는 없다.

3. **synch.c/bool semaphore_priority_compare (struct list_elem *t1, struct list_elem *t2, void *aux UNUSED)**

```
bool
semaphore_priority_compare(struct list_elem *e1, struct list_elem *e2, void
*aux UNUSED){

  struct semaphore_elem *sema_e1 = list_entry (e1, struct semaphore_elem,
elem);
  struct semaphore_elem *sema_e2 = list_entry (e2, struct semaphore_elem,
elem);
  struct list *sema_e1_waiters = &(sema_e1->semaphore.waiters);
```

```
  struct list *sema_e2_waiters = &(sema_e2->semaphore.waiters);

  struct thread *w1_front = list_entry (list_begin (sema_e1_waiters), struct
thread, elem);
  struct thread *w2_front = list_entry (list_begin (sema_e2_waiters), struct
thread, elem);

  bool ret = (w1_front->priority > w2_front->priority);

  return ret;

}
```

- 앞서 구현한 priority_compare 함수와 로직은 같지만, argument가 thread가 아닌 semaphore_elem type이기 때문에 함수를 따로 처리해주어야 한다. 또한, 여기서는 semaphore_elem에 접근하여 각 element들의 waiter list에 접근하고, 각 리스트들의 맨 앞 thread끼리 priority를 비교한다는 점에서 약간 다르다. 이를 이용해서 condition variable 관련 함수들을 수정한다.

4. synch.c/void cond_wait(struct condition *cond, struct lock *lock)

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

- 기존의 list_push_back 기반 구현을 list_insert_ordered와 위에서 구현한 semaphore_priority_compare 함수들을 이용한 구현으로 바꾸어준다. 즉, condition

variable의 waiter list에 대해서 priority를 가진 thread가 묶여 있는 semaphore가 가장 앞에 위치하도록 알고리즘을 수정한대로 구현했다.

5. **synch.c/void cond_wait (struct condition *cond, struct lock *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, semaphore_priority_compare, 0);
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
  }
}
```

- 앞서 구현한 semaphore_priority_compare 함수를 argument로 넣는 list_sort 함수를 input condition variable의 waiter list에 대해 수행해주고, 이를 통해 정렬을 완료했으므로 sema_up 함수를 호출할 때는 해당 list의 front를 pop하여 넣어준다.

Priority Inversion

Algorithm

- 앞서 Priority Scheduler을 구현함으로써 각 thread 간의 Priority를 고려하여 scheduling하도록 구현하는 데 성공했지만, 현재의 scheduler는 여전히 **Priority Inversion** 문제를 해결하지 못한다는 문제점이 있다. Priority Inversion은 우선 순위가 높은 스레드 H, 중간 스레드 M, 낮은 L이 있는 상황에서, 만약 H가 L을 기다려야 하고(예를 들어 L이 Lock을 가지고 있는 경우), M이 ready_list에 있다면, L은 running thread가 될 수 없을 것이기 때문에 L을 기다리는 H는 CPU를 점유하지 못하는 상황이다. Priority Donation은 일시적으로 높은 우선순위를 가진 thread가 낮은 우선순위를 가지고, lock 되어있는 thread에게 일시적으로 Priority를 기부하는 것을 의미한다. 위 상황을 예로 들면, L이 lock을 유지하는 동안 H가 Priority를 L에 기부하고 L이 lock을 해제하면 다시 H로 기부한 Priority를 돌려놓는 것이다. 이를 위해 기존의 baseline 코드에서 다음과 같은 구현이 필요하다.

1. thread data structre의 수정이 필요하다. Priority donation을 하기 전의 초기 priority를

저장해둔 뒤, donation이 끝난 이후 priority를 회수할 수 있도록 하는 init_priority, 해당 thread가 기다리고 있는 lock을 나타내는 lock pointer wait_on_lock, 그리고 해당 thread에게 priority를 donate한 thread들의 list인 donation_list, 해당 list를 관리하기 위해서 elem과 구분하여 사용하는 donation_list가 인자로 필요하다.

2. 현재 running thread가 lock을 기다리고 있다고 할 때, 해당 thread가 기다리고 있는 lock의 holder thread에게 자신의 priority를 빌려주는 priority_donation 함수를 구현한다. 이 때 nested donation 상황을 고려하여 최대 깊이 8에 대해 반복적으로 donation을 수행한다.

3. 기존에 구현되어 있던 lock_release 함수와 lock_acquire 함수를 수정한다. 기존의 lock_release 함수에서는 무조건 input lock의 semaphore에 대해 sema_down을 호출했는데, donation_inversion 상황을 고려하여 input lock의 holder가 있을 경우 donation_list를 업데이트하고 priority_donation 함수를 호출하여 priority donation을 수행한다.

4. donation한 element을 기준으로 두 thread를 비교하는 donation_compare_priority 함수를 구현한다.

5. current thread의 donation_list를 순회하면서 해당 원소들이 기다리고 있는 lock(wait_on_lock)이 release되는 lock이라면 해당 원소를 donation_list에서 제거하는 lock_remove 함수와, 앞서 구현한 donation_compare_priority 함수를 이용해서 donation list를 정렬한 뒤 priority를 재설정하는 priority_newly_set 함수를 구현한다.

6. 기존에 baseline으로 제공되었던 thread_set_priority 함수에서도 위에서 구현한 priority_newly_set 함수를 추가해줌으로써 함수가 호출될 때 priority를 재설정한다.

Data Structure

```
struct thread
  {
    // Already Implemented
    .
    .
    .


    /* Considering Priority Inversion & Priority Donation */
    struct lock *wait_on_lock;
    int init_priority;
    struct list donation_list;
    struct list_elem donation_elem;
```

1. **struct lock *wait_on_lock**

- thread가 얻기 위해서 기다리고 있는 lock을 나타낸다. 어떤 thread의 wait_on_lock이 NULL이 아닐 경우 thread는 해당 lock이 release될 때까지 기다린다.

2. **int init_priority;**

- 초기 priority를 저장하기 위해 정의된 멤버 변수이다. Priority donation을 하기 전의 초기 priority를 저장해둔 뒤, donation이 끝난 이후 priority를 회수하는 데 사용된다.

3. **struct list donation_list;**

- 해당 thread에게 priority를 나눠준 thread들의 list이다.

4. **struct list_elem donation_elem;**

- donation_list를 관리하기 위한 element이다. 기존에 정의된 elem과 구분해서 사용하기 위해 정의한다.

Discussion

- priority_compare 함수를 구현하여 list.c 내에 내장되어있는 list 관련 함수 (sort, insert)의 인자로 넣어주었는데, 같은 작업을 할 때 thread struct 내의 donation_elem과 semaphore의 element에 같은 함수를 사용할 수 없었다. (인자의 datatype이나, 접근하는 멤버 변수에 차이가 있었다.) 따라서 이를 위해 semaphore_priority_compare 함수와 donation_priorirty_compare 함수를 추가적으로 구현해주었다.

-

Function Implementation

1. **thread.c/static void init_thread(struct thread *t, const char *name, int priority)**

```
static void
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);
.
.
.
```

```
// initialize newly defined data structures.
  t->wait_on_lock = NULL;
  t->init_priority = priority;
  list_init(&t->donation_list);

.
.
.

}
```

- 앞서 thread structure에서 새롭게 정의해준 부분들에 대한 초기화를 진행한다. wait_on_lock의 경우 pointer이므로 NULL로 초기화하고, init_priority의 경우 priority 값과 같게 초기화해주면 된다. 마지막으로 donation_list는 list이므로 list.c의 list_init 함수를 호출하여 초기화한다.

2. thread.c/void priority donation (void)

```
void
priority_donation(void){

  int MAX_DEPTH = 8; // given condition
  struct thread *t_cur = thread_current ();

  int cur_depth = 0;

  while (cur_depth < MAX_DEPTH){
    if (t_cur->wait_on_lock == NULL) return;

    else {
      struct thread *holder = t_cur->wait_on_lock->holder;
      holder->priority = t_cur->priority;
      t_cur = holder;
    }
    cur_depth++;
  }
}
```

- priority_donation을 구현한 함수이다. 우선 thread_current() 함수 호출을 통해 running thread에 접근하고, current thread의 wait_on_lock이 NULL값인지, 즉 기다리고 있는 lock이 있는지를 체크한다. 만약 기다리고 있는 lock이 있다면, 해당 lock의 holder에 접근하여 priority값을 t_cur의 priority 값으로 대입해준다. 즉 이 과정이 priority donation

에 해당하고, 그 뒤 해당 holder을 current thread로 설정하여 donation을 반복적으로 수행하는 nested donation에 대한 예외 처리를 구현한다. 구현 요구 사항에 따라 max depth는 8로 설정하였고 만약 연쇄적으로 설정한 current thread의 wait_on_lock이 없다면 함수를 리턴하여 priority donation을 마친다.

3. **thread.c/bool donation_priority_compare(struct list_elem *t1, struct list_elem *t2, void aux UNUSED)**

```
bool // called by list.c/list_insert_ordered
donation_priority_compare(struct list_elem* t1, struct list_elem* t2, void*
aux UNUSED) {
  struct thread *thread1 = list_entry(t1, struct thread, donation_elem); // it
compares each thread's donation element
  struct thread *thread2 = list_entry(t2, struct thread, donation_elem);
  return thread1->priority > thread2->priority;
}
```

- 앞서 구현한 priority_compare나 semaphore_priority_compare 등과 로직은 같지만, 마찬가지로 donation_elem에 접근하여 priority를 비교하는 함수를 구현하기 위해서 새롭게 작성해야 한다.

4. **synch.c/void lock_acquire(struct lock *lock)**

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

  struct thread* t_cur = thread_current();


.
.
. // (changed by mlfqs exception)

  if(lock->holder){ // if holder exists

    t_cur->wait_on_lock = lock;
    list_insert_ordered(&lock->holder->donation_list, &t_cur->donation_elem,
    donation_priority_compare, 0);
    priority_donation();
```

```
  }

  sema_down (&lock->semaphore);
  t_cur->wait_on_lock=NULL;
  lock->holder = t_cur;
}
```

- priority donation을 해야 하는 상황에 대한 예외를 처리한다. Input argument인 lock의 holder, 즉 lock을 점유하고 있는 thread를 체크하고, 해당 thread의 donation lost를 donation element에 대해 정렬한다. 그 뒤, 위에서 구현한 priority_donation()함수를 호출하여 priority donation을 수행한다. 이 때 less function은 donation_priority_compare 함수이다.

5. synch.c/void remove_lock(struct lock *lock)

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

  struct thread *t_cur = thread_current ();
  struct list_elem *e;

  for (e = list_begin (&t_cur->donation_list); e != list_end (&t_cur-
>donation_list); e = list_next (e)){
    struct thread *t = list_entry (e, struct thread, donation_elem);
    if (lock == t->wait_on_lock) list_remove (&t->donation_elem);
  }
}
```

- input argument로 막 release된 lock을 받고, donation list를 순회하면서 해당 lock에 걸려 있는 donation element를 donation list에서 제거하는 함수이다. 이 때 list에서 제거하는 함수는 list.c/list_remove 함수를 호출하여 사용했다.

6. thread.c/void priority_newly_set(void)

```
void
priority_newly_set(void){

  struct thread *t_cur = thread_current();
  t_cur->priority = t_cur->init_priority;
```

```
  if (list_empty(&t_cur->donation_list) == true) return;

  list_sort(&t_cur->donation_list, donation_priority_compare, 0);
  struct thread *p_max_thread = list_entry(list_front(&t_cur->donation_list),
struct thread, donation_elem);
  if (p_max_thread->priority > t_cur-> priority) t_cur->priority =
p_max_thread->priority;

  return;
}
```

- priority를 재설정하는 함수이다. lock_release에서 priority donation을 고려했을 때 호출해야 하는 함수이다. priority inversion 상황을 해결하고 난 뒤, lower priority를 가졌던 thread에 걸린 lock을 release하고 나면 higher priority를 가진 thread로부터 받았던 priority를 지우고 priority를 다시 정해줘야 한다. 우선 함수에서 current thread의 priority를 initial priority로 재설정해주고, donation list가 비어있는 경우 함수를 그대로 return하여 initial priority를 갖게 한다. donation list를 donation_priority_compare 함수를 argument로 넣어 정렬해준다. 그 뒤 해당 list의 front를 가져와서 priority가 가장 높은 p_max_thread를 불러오고, current thread와 비교해서 current thread보다 더 크다면 p_max_thread의 priority를 대입해준다.

7. synch.c/void lock_release(struct lock *lock)

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

  lock->holder = NULL;

  //it is not complete implementation, it will be changed with mlfqs.
  remove_lock (lock);
  priority_newly_set();


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

- priority donation을 고려하여, 앞서 구현한 remove_lock 함수와 priority_newly_set 함수

를 차례로 호출해준다. 이를 통해 방금 release된 lock을 기다렸던 thread들을 donation list에서 삭제하고. 다른 thread에 donate했던 current thread의 priority를 회수한다.

8. thread.c/void thread_set_priority (int new_priority)

```
void
thread_set_priority (int new_priority)
{
  .
  .
  . // changed with mlfqs

  thread_current ()->init_priority = new_priority;

  priority_newly_set();
  thread_priority_test();
}
```

- 앞서 구현한 priority_newly_set() 함수를 호출해서 다른 thread에 donate했던 current thread의 priority를 회수하고 priority를 재설정한다.

1-3. Advanced Scheduler

Algorithm

- Advanced Scheduler는 실시간으로 priority를 조절하는 mlfqs(multi-level feedback queue scheduling) 방식을 구현하여 좀 더 효율적인 스캐쥴링을 구현하는데 목적이 있다. 이를 위해 기존의 baseline 코드에서 다음과 같은 구현이 필요하다.

  1. fixed_point 기능을 제공하지 않는 pintos에서 실수를 다루기 위해 fixed_point.h 파일에 fixed_point 연산 함수를 구현한다.
  2. thread struct에 각 thread의 고유의 특징을 나타내는 nice, recent_cpu를 추가한다.
  3. 시스템에 하나의 값으로 존재해야하는 load_avg는 thread.c 에 전역변수로 추가한다.
  4. 문서에 나온 priority, recent_cpu, load_avg를 계산하는 공식 그대로 계산을 해 주는 함수를 구현한다.
  5. 이제 위에서 구현한 함수를 이용하여 모든 thread의 priority, recent_cpu를 재계산

하는 함수와 load_avg를 재계산하는 함수를 구현한다.

6. mlfqs 활성화시, priority donation를 구현한 lock_acquire, lock_release 함수와 prioirty를 임의로 변경시키는 thread_set_priority함수를 비활성화 시킨다.

7. 모든 구현을 마쳤으므로 pintos에서 구현이 되어있지 않은 thread_set_nice, thread_get_nice, thread_get_load_avg, thread_get_recent_cpu 함수를 구현한다.

Data Structure

- Advanced scheduler에서 다루는 변수 중 recent_cpu와 load_avg는 실수지만, pintos에서는 부동소수점 연산을 지원하지 않아 실수로 저장을 물론이고 실수와 실수, 실수와 정수간의 연산이 불가능하다. 따라서 직접 fixed point(고정소수점)표현과 연산방식을 직접 구현해야 한다.

  fixed point 방식은 32bit 중 실수가 17bit, 소수가 14bit 그리고 1 bit의 sign으로 이루어져 있다. 우리는 이 값을 정수로 저장해야 하므로, 실수 일 경우에는 2^14를 곱한 값을 저장하고, 연산을 할 때도 실수값에 맞춰 계산을 해야 한다. 구현해야 하는 식은 pintos 문서에 있다. 식은 다음과 같다.

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`

Discussion

- Design Report에서는 load_avg를 계산하는 함수인 **mlfqs_calculate_load_avg** 의

parameter를 받지 않았다. 이는 다른 recent_cpu, priority 변수와 같이 각 thread가 고유로 가지는 값이 아니라 시스템 전체에 딱 하나 존재하는 값이기 때문에 load_avg를 다시 계산할 때 이 함수를 호출하기만 하면 된다고 생각했다. 하지만 현재 thread가 idle thread인지 여부에 따라 running thread의 개수가 달라지므로, running thread 개수를 parameter로 넘겨주도록 수정하였다.

- load_avg 를 다시 계산하는 함수인 **mlfqs_load_avg** 함수에 **mlfqs_calculate_load_avg** 기능을 구현할 수 있었지만 recent_cpu와 priority 변수를 다시 계산하는 함수와 통일성을 주기 위해 따로 분리하여 두 함수를 구현하였다.

Function Implementation

1. **fixed_point.h/ int int_to_fp(int N), int fp_to_int(int X), int fp_to_int_round(int X), int add_fp_fp(int X, int Y), int add_fp_int(int X, int N), int sub_fp_fp(int X, int Y), int sub_ fp_int(int X, int N), int multi_fp_fp(int X, int Y), int multi_ fp_int(int X, int N), int div_fp_fp(int X, int Y), int div_ fp_int(int X, int N)**

```
#include <stdint.h>


#define F (1<<14)

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){return (X>=0)?(X+F/2)/F:(X-F/2);};


int add_fp_fp(int X, int Y){return X+Y;};
int add_fp_int(int X, int N){return X+N*F;};

int sub_fp_fp(int X, int Y){return X-Y;};
int sub_fp_int(int X, int N){return X-N*F;};

int multi_fp_fp(int X, int Y){return ((int64_t)X)*Y/F;};
int multi_fp_int(int X, int N){return X*N;};

int div_fp_fp(int X, int Y){return ((int64_t)X)*F/Y;};
int div_fp_int(int X, int N){return X/N;};
```

- fixed point를 계산하는 함수이다. Pintos document에 있는 알고리즘을 그대로 구현하였다.

2. **thread.h/struct thread/int nice, int recent_cpu**

```
struct thread

  {
  /*…*/
    int nice;
    int recent_cpu;
  };
```

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  /*…*/
  t->nice=0;
  t->recent_cpu=0;
  /*…*/
}
```

- thread struct에 nice, recent_cpu를 추가하고, init_thread에서 초기화 했다. Nice는 CPU time을 다른 thread에게 양보하는 정도를 나타내는 niceness 값을 나타내는 변수로, 각 thread별로 갖는 고유한 값이다. 아래서 구현할 thread_set_nice()함수에서 nice값을 설정한다.

3. **thread.c/int load_avg**

```
/*...*/

int load_avg;


/*…*/

void
thread_start (void)
{
  /*…*/
  load_avg=0;
  /*…*/
}
```

- thread.c 전역변수로 load_avg를 추가하고 thread_start에서 초기화했다. 1분동안 평균적으로 수행 가능한 thread의 개수를 나타내는 변수로, 본 시스템에 하나의 값으로 존재

한다.

4. **thread.c/void mlfqs_calculate_priority(struct *thread t)**

```
void mlfqs_calculate_priority(struct thread* t)

{

  int priority;
  if(t!=idle_thread)
  {
    priority=fp_to_int(add_fp_int (div_fp_int (t->recent_cpu, -4), PRI_MAX -
t->nice * 2));

    if (priority > PRI_MAX)
    {
      t->priority = PRI_MAX;
    }
    else if (priority < PRI_MIN)
    {
      t->priority = PRI_MIN;
    }
    else
    {
      t->priority = priority;
    }
  }
  return;

}
```

- priority 계산 공식을 바탕으로 thread의 priority를 계산하는 함수이다. priority는 Pintos 내 정의에 따라 0-63 사이의 값을 가지도록 설정한다.

5. **thread.c/void mlfqs_calculate_recent_cpu (struct *thread t)**

```
void mlfqs_calculate_recent_cpu(struct thread* t)

{
  if (t == idle_thread) return;
  t->recent_cpu = add_fp_int (multi_fp_fp (div_fp_fp (multi_fp_int (load_avg,
2), add_fp_int (multi_fp_int (load_avg, 2), 1)), t->recent_cpu), t->nice);
}
```

- Pintos document의 recent_cpu 계산 공식을 바탕으로 thread의 recent_cpu를 계산하는 함수이다. 앞서 구현한 소수점 계산 관련 함수들을 사용하였다.

6. thread.c/void mlfqs_calculate_load_avg (int ready_threads)

```
void mlfqs_calculate_load_avg(int ready_threads)

{
  load_avg = add_fp_fp (multi_fp_fp (div_fp_int (int_to_fp (59), 60),
load_avg), multi_fp_int (div_fp_int (int_to_fp (1), 60), ready_threads));
}
```

- load_avg 계산 공식을 바탕으로 load_avg를 계산하는 함수. 실행가능한 thread의 개수를 담는 ready_threads를 parameter로 받는다.

7. thread.c/void mlfqs_increment_recent_cpu()

```
void mlfqs_increment_recent_cpu()

{
  if (thread_current () != idle_thread)
  {
    thread_current () ->recent_cpu = add_fp_int (thread_current ()-
>recent_cpu, 1);
  }
}
```

- 현재 thread의 recent_cpu를 1 증가시키는 함수이다.

8. thread.c/void mlfqs_priority()

```
void mlfqs_priority (void)

{
  struct list_elem *e;

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

- 모든 thread의 priority를 다시 계산하는 함수. 모든 thread가 들어있는 all_list를 돌아 각 thread의 priority를 위에서 구현한 mlfqs_calculate_priority함수를 통해 다시 계산한다.

-

9. thread.c/void mlfqs_recent_cpu()

```
void mlfqs_recent_cpu()

{
  struct list_elem *e;

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

- 모든 thread의 recent_cpu를 다시 계산하는 함수이다. 모든 thread가 들어있는 all_list를 돌아 각 thread의 priority를 위에서 구현한 mlfqs_calculate_recent_cpu함수를 통해 다시 계산한다.

10. thread.c/void mlfqs_load_avg()

```
void mlfqs_load_avg(void)

{
  int ready_threads = list_size (&ready_list);

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

- load_avg를 다시 계산하는 함수이다. idle thread는 실행 가능한 thread에 포함시키지 않으므로, 만약 현재 thread가 idle thread일 경우 ready_list의 개수를, 현재 thread가 idle thread가 아닌 경우는 ready_list개수에 현재 running thread 1개를 더한 값을 실행

가능한 thread 개수로 생각하고, mlfqs_calculate_load_avg 함수에 ready_thread parameter로 전달하여 load_avg를 다시 계산한다.

11. timer.c/static void timer_interrupt (struct intr_frame, *args UNUSED)

```
static void

timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();

  if (thread_mlfqs) {
    mlfqs_increment_recent_cpu ();
    if (ticks % 4 == 0) {
      mlfqs_priority ();
      if (ticks % TIMER_FREQ == 0) {
        mlfqs_load_avg ();
        mlfqs_recent_cpu ();
      }
    }
  }

  thread_awake (ticks);
}
```

- timer interrupt가 일어났을 때 mlfqs가 활성화된 경우 일정 시간마다 각 변수들을 다시 계산하는 함수들을 호출한다. Recent_cpu값은 load_avg 값에 영향을 받으므로, mlfqs_load_avg()함수를 먼저 호출 한 다음 mlfqs_recent_cpu()함수를 호출했다.

12. synch.c/void lock_acquire (struct lock *lock)

```
void

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

  struct thread* t_cur = thread_current();

  if (thread_mlfqs) {
```

```
    sema_down (&lock->semaphore);
    lock->holder = t_cur;
    return;
  }

  if(lock->holder){ // if holder exists

    t_cur->wait_on_lock = lock;
    list_insert_ordered(&lock->holder->donation_list, &t_cur->donation_elem,
    donation_priority_compare, 0);
    priority_donation();
  }

  sema_down (&lock->semaphore);
  t_cur->wait_on_lock=NULL;
  lock->holder = t_cur;
}
```

-

- mlfqs 옵션으로 명령어가 들어올 경우에 priority donation을 비활성화해야 하므로, semaphore down과 input lock의 holder를 현재 thread로 설정해준 뒤 함수를 return하여 뒤의 priority donation 관련 부분을 실행하지 않는다.

13. synch.c/void lock_release (struct lock *lock)

```
void

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

  lock->holder = NULL;

  if (!thread_mlfqs)
  {
    remove_lock (lock);
    priority_newly_set();
  }

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

- 마찬가지로 lock_release 함수에서 priority donation을 비활성화 시킨다. priority

donation을 적용해야 하는 상황에만 관련한 remove_lock과 priority_new_set 함수를 호출해야 하므로 해당 부분에 대해 예외처리를 해준다.

14. thread.c/void thread_set_priority(int new_priority)

```
void

thread_set_priority (int new_priority)
{
  if (thread_mlfqs)
  {
    return;
  }
  thread_current ()->init_priority = new_priority;

  priority_newly_set();
  thread_priority_test();
}
```

- mlfqs가 활성화된 경우 priority를 임의로 변경하는 thread_set_priority함수를 비활성화 한다.

15. thread.c/void thread_set_nice(int nice UNUSED), int thread_get_nice(void), int thread_get_load_avg(void), void thread_get_recent_cpu(void)

```
/* Sets the current thread's nice value to NICE. */

void
thread_set_nice (int nice UNUSED)
{
  /* Not yet implemented. */
  enum intr_level old_level = intr_disable ();

  struct thread *t = thread_current();

  t->nice = nice;
  mlfqs_calculate_priority (t);

  if (t != idle_thread)
  {
    thread_priority_test();
  }
```

```
  intr_set_level (old_level);
}

/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
  /* Not yet implemented. */
  enum intr_level old_level = intr_disable ();

  int return_nice = thread_current ()-> nice;

  intr_set_level (old_level);
  return return_nice;
}

/* Returns 100 times the system load average. */
int
thread_get_load_avg (void)
{
  /* Not yet implemented. */
  enum intr_level old_level = intr_disable ();

  int return_load_avg = fp_to_int_round (multi_fp_int (load_avg, 100));

  intr_set_level (old_level);
  return return_load_avg;
}

/* Returns 100 times the current thread's recent_cpu value. */
int
thread_get_recent_cpu (void)
{
  /* Not yet implemented. */
  enum intr_level old_level = intr_disable ();

  int return_recent_cpu = fp_to_int_round (multi_fp_int (thread_current ()-
>recent_cpu, 100));

  intr_set_level (old_level);
  return return_recent_cpu;
}
```

- 모든 기능을 구현했으니, 구현이 되어있지 않은 위 4함수를 구현한다. nice 값을 set하는 thread_set_nice()함수, nice값을 return 하는 thread_get_nice() 함수를 구현하였다. 그리고 pintos document대로 thread_get_recent_cpu()와 thread_get_load_avg()함수에서는

recent_cpu와 load_avg값에 100을 곱한 값을 return하도록 구현하였다.