Assn1 Final Report

Pintos Assn1 Final Report

Team37 20200229 김경민, 20200423 성치호

구현

목표 달성을 위해 추가로 구현, 수정한 코드에 대한 설명으로 Design Report과 달라진 점도 포함하고 있다.

Alarm Clock

design report에서 계획한 구현에서 크게 벗어나지 않았다.

`sleep_list` in `devices/timer.c`

```
/* List of sleep processes.  Processes are added to this list
   when they start sleeping due to timer_sleep() and removed when they wake up. */
static struct list sleep_list;
```

`list` struct 변수인 `sleep_list`를 `timer.c`에 추가하였다. 해당 변수는 현재 sleep하고 있는 `thread`에 대한 리스트로 아래에서 추가한 `struct thread`의 `sleep_elem`을 이용해 list를 구성하고 있다. 기존 계획에는 `thread.c`에 추가하려고 하였으나 기구현된 연관된 함수인 `timer_sleep`이 `timer.c`에 있기에 `sleep_list` 또한 `timer.c`에서만 사용하는 static 변수로 추가하였다.

```
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
  /* Initialize for timer_sleep() */
  list_init(&sleep_list);
}
```

task를 수행하기 전 초반 커널 초기화 때 `timer_init`을 호출하는데 이 때 함께 `sleep_list` 변수를 초기화하기 위해 `timer_init`에 `list_init(&sleep_list);`을 추가하였다. 계획과 달리 `sleep_list`를 `timer.c`에 추가하였기 때문에 초기화도 `thread_init`이 아닌 `timer_init`에 추가하였다.

`thread`

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    int64_t wake_up_tick;               /* Tick time thread need to wake up */

    ...
    struct list_elem allelem;           /* List element for all threads list. */

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

    struct list_elem sleep_elem;        /* List element for sleep threads list */
        ...
    unsigned magic;                     /* Detects stack overflow. */
  };
```

`struct thread`에 속하는 변수로 `wake_up_tick`을 추가하였다. 이는 해당 스레드가 sleep에서 벗어나야 할 tick에 대한 변수이다. 또한 `list_elem sleep_elem`을 추가하였다. 이는 `sleep_list` 리스트를 구성할 때 사용하기 위해 추가한 변수이다.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ...
  t->stack = (uint8_t *) t + PGSIZE;
  t->wake_up_tick = 0;
  t->priority = priority;
  ...
}
```

`init_thread`로 thread 생성 후, `thread`를 초기화하는 함수에도 `wake_up_tick` 초기화 로직을 추가해주었다. `thread` 내 변수인 `wake_up_tick`를 `init_thread`에서 다른 변수를 초기화할 때 함께 0으로 초기화해준다. `sleep_list`에 해당 스레드가 포함되지 않는 이상 참조할 일이 없지만 안전, 통일성을 위해 추가해주었다.

`timer_sleep`

기존에 busy wait으로 구현된 `timer_sleep`을 thread block, unblock을 이용한 구현으로 개선하기 위해 기존 `yield`를 제거하고 `timer_sleep`을 다음처럼 변경하였다.

```
void
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (intr_get_level () == INTR_ON);

  /* Don't have to sleep */
  if (ticks <= 0)
    {
      return;
    }
```

design report에서 계획한 것에 따라 구현하였다. 먼저 sleep 해야 하는 tick이 0보다 작거나 같다면 thread를 sleep할 필요가 없으므로 아무것도 하지 않고 리턴하면 된다. (어떻게 보면 잘못된 인수를 받은 것으로 생각할 수 있지만 테스트 내 0보다 작거나 같은 값을 이용한 호출이 있기에 Assert가 아닌 if문으로 예외 처리)

```
  /* Thread block need Interrupt disabled */
  old_level = intr_disable ();
  cur->wake_up_tick = start + ticks;
  list_insert_ordered (&sleep_list, &(cur->sleep_elem), wake_up_tick_less, NULL);
  thread_block ();

  intr_set_level (old_level);
}
```

기존에는 현재 tick이 sleep 시작 시간 + tick보다 작으면 계속 `thread_yield`를 하는 구현이였다. sleep이 완료될 수 있는(wake up할 수 있는) tick (start+ tick)을 미리 계산한 이후 각 스레드의 변수 `wake_up_tick`에 설정해 집어넣어 주었다. 이후 `wake_up_tick_less`을 사용한 `list_insert_ordered`를 이용해 해당 스레드를 `sleep_list`에 `wake_up_tick`을 오름차순으로 정렬하였을 때 기준 올바른 순서에 넣어주었다.

- 이처럼 `wake_up_tick`을 기준으로 오름차순으로 정렬한 순서에 맞게 `sleep_list` 리스트에 집어넣는 이유는 처음에 O(N)의 시간복잡도로 리스트에 집어 넣게 되면 `sleep_list`는 `wake_up_tick`이 가장 빠른, 가장 작은 값의 스레드가 올 것이 보장되기에 이후 틱마다 `check_wake_up`에서 `sleep_list`를 순회하며 일어나야할(unblock해야 할) 스레드를 정할 때 앞에서부터 확인하고 현재 시간보다 늦은 시간의 스레드가 나오면 바로 멈출 수 있다. 왜냐하면 정렬되어있기에 그 뒤의 스레드들은 순회를 멈춘 스레드보다 큰 `wake_up_tick`을 가짐이 보장되기 때문이다. 또한 삭제시에는 항상 맨 앞만 삭제할 것이 보장되기에 삭제에 O(1) 시간복잡도를 가지는 등 insert_ordered는 여러 이점이 있다.

마지막으로 현재 스레드(`timer_sleep`을 호출한 함수)를 block 함으로써 이 후 `check_wake_up`에 의해 wake up하기 전까지는 스케줄링되지 못하도록, cpu를 점유할 수 없도록 하였다.

- block 되기에 `ready_list`에서도 제거되기에 스케줄 및 cpu 점유할 가능성이 전혀 없다.
- 이로 인해 기존의 busy wait보다 효율적인 wait이 가능해진다.

`sleep_list`에는 thread가 추가되었지만 thread는 block 되지 않는다면 문제가 생기므로 해당 작업들을 수행할 때는 interrupt를 비활성화하고 sleep으로 인한 block 이후 unblock되어 되돌아올 스레드를 위해 마지막에 interrupt level을 되돌려 주었다.

`wake_up_tick_less`

```
static bool
wake_up_tick_less (const struct list_elem *a_, const struct list_elem *b_,
            void *aux UNUSED)
{
  const struct thread *a = list_entry (a_, struct thread, sleep_elem);
  const struct thread *b = list_entry (b_, struct thread, sleep_elem);

  return a->wake_up_tick < b->wake_up_tick;
}
```

`timer.c`에 `list_insert_ordered`에서 비교하는 함수로 사용하는 `wake_up_tick_less`를 추가하였다. 해당 함수는 2개의 `list_elem`인 `a_`, `b_`를 입력받아 `a_`가 포함된 `thread`의 `wake_up_tick`이 `b_`가 포함된 `thread`의 `wake_up_tick`보다 작을 때 true를 반환하고 그 반대일 때 false를 반환한다. 해당 함수를 이용해 `list_insert_ordered`를 하여 `sleep_list`가 항상 `wake_up_tick` 오름차순으로 list를 유지하게 한다.

- 해당 함수의 구조는 `list_less_func`와 동일하며 해당 함수는 A가 B보다 앞서야 할 때 True, 반대일 때 False를 반환하게 설계해야 한다.

`check_wake_up`

```
void
check_wake_up (void)
{
  int64_t now = timer_ticks ();

  while (list_empty(&sleep_list) == false)
  {
    struct thread *t = list_entry (list_front (&sleep_list), struct thread, sleep_elem);
    if (t-> wake_up_tick > now)
    {
      return;
    }
    list_pop_front (&sleep_list);
    thread_unblock(t);
  }
}
```

`sleep_list`의 thread 중 일어나야 할 스레드를 검사하고 wake up해야하는 스레드들을 unblock하는 함수인 `check_wake_up`을 `timer.c`에 정의를 추가하고 `timer.h`에도 선언을 추가하여 다른 파일들에서 사용할 수 있도록 하였다. 먼저 현재 시간을 얻은 뒤, `sleep_list`가 빌 때까지 다음을 반복한다.
현재 `sleep_list`의 맨 앞 `list_elem`이 가리키는 스레드의 `wake_up_tick`이 현재 시간보다 크다면 아직 일어날 시간이 되지 않았으므로 함수를 리턴하여 while 및 함수 작동을 종료한다.

- `sleep_list`의 맨 앞 아이템만 확인하고 조건에 만족하지 않으면 리턴하는 이유는 `sleep_list`는 `list_insert_ordered`를 사용해 insert했으므로 `wake_up_tick` 오름차순으로 정렬되었음이 보장되기 때문에 뒤의 아이템들의 스레드 또한 wake_up 조건에 만족하지 않을 것이 보장되기 때문이다.
  만약 확인한 `list_elem`이 가리키는 스레드의 `wake_up_tick`이 현재 시간보다 작거나 같다면 이제 일어나야 할 시간이므로 `list_pop_front`를 통해 해당 `list_elem`인 `sleep_list`
  맨 앞 아이템을 제거하고 `list_elem`이 가리키는 스레드를 unblock한다. 이후 이를 반복한다.

```
void
thread_tick (void)
```

```
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Check the thread that needs to wake up. */
  check_wake_up();

  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
}
```

다음처럼 Timer interrupt가 발생하면 핸들러 함수인 `timer_interrupt` 에 의해 실행되는 `thread_tick` 에 `check_wake_up()` 을 추가하여 **틱마다** 현재 wake up 해야할 스레드가 있는지 확인하고 wake up할 수 있도록 하였다.

Priority Scheduler

기존 Design report에서 변경된 점이 있는데, 우선순위를 기준으로 스레드를 나열해야 하는 `thread/ready_list` , `synch/sema->waiters` 에서 **우선순위 기준 정렬을 유지한 채로 관리하지 않았다.** 따라서 가장 높은 우선순위의 스레드를 고르려면 매번 리스트를 처음부터 끝까지 순회하며 해당하는 스레드를 찾아야 하는데, 이와 같이 구현한 이유는 **최악의 경우 Priority Donation이 일어날 때마다 임의의 리스트에서의 정렬 상태가 깨질 가능성이 있기 때문**이다. 따라서 이와 같이 구현할 경우 Priority Donation이 일어날 때마다 해당 스레드들과 관여된 모든 리스트들을 재정렬해줘야 하는데, 이를 피하는 것이 구현에 비교적 용이할 것이라 판단하여 스레드를 정렬하지 않은 채로 관리하는 것으로 결정했다.

Basic implementation

`[threads/thread.c]`

```
/* Compares priority of thread a and b, return true if priority
   of a is smaller than b. */
bool
compare_thread_priority(const struct list_elem *a,
                        const struct list_elem *b,
                        void *aux UNUSED)
{
  return list_entry(a, struct thread, elem)->priority
  < list_entry(b, struct thread, elem)->priority;
}
```

`ready_list` 내에서 우선순위에 따른 스레드 비교를 위한 함수. 인자로 주어진 `list_elem` 이 들어있는 스레드가 무엇인지 `list_entry` 를 이용해 찾아낸 뒤 `priority` 를 참조하여 비교한다.

`[threads/thread.c]`

```
/* Checks if the priority of current thread is lower than the
   largest one in the ready list. If it's true, thread should
   be yielded. */
void
thread_preempt (void)
{
  if(list_empty(&ready_list)) return;
  if(thread_current()->priority < list_entry(
    list_max(&ready_list, compare_thread_priority, NULL),
    struct thread, elem)->priority){
      thread_yield();
```

```
  }
}
```

현재 실행 중인 스레드 혹은 대기중인 스레드의 우선순위가 바뀌어 실행 중인 스레드의 우선순위가 가장 높지 않을 가능성이 생겼을 경우 CPU를 넘겨주는 함수이다. `thread_create`와 `thread_set_priority`, `sema_up`의 마지막에서 호출해준다.

`[lib/kernel/list.c]`

```
/* Removes the element in LIST with the largest value according
   to LESS and returns it. Undefined behavior if LIST is empty
   before removal. */
struct list_elem *
list_pop_max (struct list *list, list_less_func *less, void *aux)
{
  struct list_elem *max = list_max (list, less, aux);
  list_remove (max);
  return max;
}

/* Removes the element in LIST with the smallest value according
   to LESS and returns it. Undefined behavior if LIST is empty
   before removal. */
struct list_elem *
list_pop_min (struct list *list, list_less_func *less, void *aux)
{
  struct list_elem *min = list_min (list, less, aux);
  list_remove (min);
  return min;
}
```

리스트 연산의 편의를 위해 추가한 함수들. 해당 리스트에서 `less` 비교함수에 의해 결정된 최댓값/최솟값을 가지는 원소를 리스트에서 삭제 후 반환한다. 구현 시 유의해야 했던 점으로 `list_remove` 함수는 삭제한 원소가 아닌 삭제한 원소의 **다음 원소**를 반환하므로 `list_remove`의 반환값을 그대로 반환하면 예상치 못한 동작이 수행될 수 있다.

`[threads/thread.c]`

```
static struct thread *
next_thread_to_run (void)
{
  if (list_empty (&ready_list))
    return idle_thread;
  else
    /* choose a thread with maximum priority and pop it from
    ready_list */
    return list_entry (
      list_pop_max (&ready_list, compare_thread_priority, NULL),
      struct thread, elem);
}
```

다음 스케줄할 스레드를 뽑는 `next_thread_to_run` 함수는 `list_pop_front`를 이용하여 리스트의 가장 앞에서 FIFO 형태로 고르는 기존 방식 대신 위에서 구현한 `list_pop_max` 함수를 이용하여 우선순위가 가장 높은 스레드를 pop해 반환해주도록 수정하였다.

`[threads/synch.c]`

```
void
sema_up (struct semaphore *sema)
{
  ...
  if (!list_empty (&sema->waiters))
    thread_unblock (list_entry (
      list_pop_max (&sema->waiters, compare_thread_priority, NULL),
                    struct thread, elem));
  sema->value++;
  thread_preempt();
  intr_set_level (old_level);
}
```

Semaphore 역시 멤버 변수로 현재 `sema_down` 을 시도한 스레드들을 담아 관리하는 `waiters` 리스트가 있고 이 역시 스레드 우선순위에 따라 관리되어야 하므로, 리스트에서 스레드를 꺼낼 때 `list_pop_max` 함수를 통해 우선순위가 가장 높은 스레드를 pop해야 한다.

`[threads/synch.c]`

```
/* Compares priority of semaphore a and b, return true if
   priority of the biggest thread in the waiters list of a is
   smaller than the one of b. */
bool
compare_sema_priority(const struct list_elem *a,
                            const struct list_elem *b,
                            void *aux UNUSED)
{
  struct semaphore_elem
  *sema_a = list_entry(a, struct semaphore_elem, elem),
  *sema_b = list_entry(b, struct semaphore_elem, elem);

  struct list_elem
  *max_a = list_max(&(sema_a->semaphore.waiters),
                    compare_thread_priority, NULL),
  *max_b = list_max(&(sema_b->semaphore.waiters),
                    compare_thread_priority, NULL);

  return list_entry(max_a, struct thread, elem)->priority
  < list_entry(max_b, struct thread, elem)->priority;
}
```

Conditional Variable 역시 앞의 경우와 마찬가지로 `cond_signal` 에서 우선순위에 따라 pop할 원소를 선택해야 하지만, 위의 경우들과는 달리 이번에는 비교할 원소가 스레드가 아닌 Semaphore이다. 각 Semaphore 역시 앞서 살펴봤듯이 스레드의 리스트를 관리하고 있으므로, `cond_signal` 에 필요한 `list_pop_max` 의 동작은 **해당 Conditional Variable의 waiters 리스트에 담긴 Semaphore들이 관리 중인 모든 스레드들 중 가장 우선순위가 높은 스레드가 담겨 있는 Semaphore를 pop**하는 것으로 정리할 수 있다. 따라서 이를 구현하기 위해 두 Semaphore에 담긴 스레드들의 최대 우선순위를 비교하는 `compare_sema_priority` 비교함수를 구현하였다.

`[threads/synch.c]`

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ...
  if (!list_empty (&cond->waiters))
    sema_up (&list_entry (
          list_pop_max (&cond->waiters, compare_sema_priority, NULL),
          struct semaphore_elem, elem)->semaphore);
}
```

위의 비교함수를 이용하여 가장 우선순위가 높은 스레드가 들어있는 Semaphore를 pop하도록 `cond_signal` 함수를 수정하였다.

지금까지의 구현 사항을 적용함으로써 기본적인 스레드 우선순위 동작을 구현할 수 있었다.

Priority Donation

Priority Donation을 구현하기 위해선 `lock_acquire` 시 해당 lock을 보유 중인 스레드에게 자신의 우선순위를 전해주고 이를 재귀적으로 일정 깊이만큼 반복하는 동작 (Nested Donation), `lock_release` 시 자신이 해당 lock을 이유로 Donate 받았던 우선순위를 반납한 뒤 남은 우선순위들을 토대로 자신의 우선순위를 갱신하는 동작 (Multiple Donation)을 구현해야 한다.

`[threads/thread.h]`

```
struct lock *lock_to_acquire;
struct list donors;
int base_priority;
struct list_elem donation_elem;
```

스레드 구조체에 다음과 같은 멤버 변수들을 추가해줬다. `*lock_to_acquire` 은 `lock_acquire` 를 통해 자신이 acquire를 시도한 lock을 나타낸다. `donors` 리스트는 자신에게 우선순위를 donate해준 스레드들의 리스트를 나타내고, `donation_elem` 은 `donors` 리스트에서의 연결관계를 저장한다. 마지막으로 `base_priority` 는 우선순위를 donate받기 전 스레드 자신이 가지고 있던 우선순위를 나타낸다.

이때 기존의 `elem`을 재사용하지 않고 `donation_elem`을 새로 정의하는데, Priority Donation을 구현함으로써 스레드가 `ready_list`(혹은 동일한 `elem`을 사용하는 `sema->waiters`)와 `donors`에 동시에 들어있을 수 있기 때문에 리스트의 연결관계를 나타내는 `list_elem` 역시 따로 선언해줘야 한다.

`[threads/thread.c]`

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ...
  t->lock_to_acquire = NULL;
  list_init(&t->donors);
  t->base_priority = priority;
  ...
}
```

추가한 멤버 변수들은 `init_thread`에서 초기화한다. `donation_elem`을 비롯한 `list_elem`들은 초기값이 크게 중요하지 않고 리스트에서 관리되기 때문에 따로 초기화하지 않았다.

`[threads/thread.c]`

```
/* Compares priority of thread a and b, return true if priority
   of a is smaller than b. */
bool
compare_donation_priority(const struct list_elem *a,
                          const struct list_elem *b,
                          void *aux UNUSED)
{
  return list_entry(a, struct thread, donation_elem)->priority
  < list_entry(b, struct thread, donation_elem)->priority;
}
```

`donation_elem`을 비교하기 위한 비교함수 역시 재사용이 불가하므로 별개로 선언해줘야 한다.

`[threads/synch.c]`

```
void
lock_acquire (struct lock *lock)
{
  ...
  struct thread *current = thread_current();
  if(!thread_mlfqs){
    if(lock->holder != NULL){
      current->lock_to_acquire = lock;
      list_push_back(&lock->holder->donors, &current->donation_elem);
      nested_donation();
    }
  }

  sema_down (&lock->semaphore);
  if(!thread_mlfqs)
    current->lock_to_acquire = NULL;
  lock->holder = thread_current ();
}
```

`[threads/thread.c]`

```
/* Performs a nested priority donation from current thread. */
void
nested_donation(void)
{
  struct thread *current = thread_current();
  for(int iter = 0; iter < MAX_NEST_DEPTH; iter++){
    if(current->lock_to_acquire == NULL) break;
    current->lock_to_acquire->holder->priority = current->priority;
    current = current->lock_to_acquire->holder;
```

```
    }
  }
```

다음과 같이 Nested Donation을 구현하였다. `lock_acquire`를 실행할 시 이미 해당 lock을 다른 스레드가 보유 중이라면 자신의 `lock_to_acquire`를 갱신해주고 해당 lock을 보유 중인 스레드의 `donors` 리스트에 자신을 추가한다. 그 뒤 `nested_donation`을 실행해 재귀적으로 자신의 우선순위가 전해지도록 만든다.

`nested_donation`은 `lock_acquire`를 실행한 현재 스레드인 `thread_current()`부터 시작하여 `lock_to_acquire->holder`를 바탕으로 다음 스레드를 찾아 우선순위를 전파한다. 현재 보고 있는 스레드가 더이상 lock을 요청하지 않을 때까지, 혹은 명시적으로 선언된 최대 전파 깊이인 `MAX_NEST_DEPTH`만큼 전파되었을 경우 Nested Donation을 종료한다.

추가로 구현한 Advanced Scheduler는 Priority Donation 기능을 사용하지 않으므로 `thread_mlfqs` 플래그를 이용하여 해당 기능의 실행을 막았다.

`[threads/thread.h]`

```
/* Maximum depth of nested priority donation. */
#define MAX_NEST_DEPTH 8
```

위에서 사용된 `MAX_NEST_DEPTH`는 다음과 같이 정의하였다.

`[threads/synch.c]`

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));
  /* Delete the threads from donors who required this lock */
  lock->holder = NULL;
  if(!thread_mlfqs)
  {
    struct list *donors  = &thread_current()->donors;
    struct list_elem *iter;
    for(iter = list_begin(donors);
        iter != list_end(donors);
        iter = list_next(iter))
    {
      if(list_entry(iter, struct thread, donation_elem)
        ->lock_to_acquire == lock)
      {
        list_remove(&list_entry(iter, struct thread, donation_elem)
        ->donation_elem);
      }
    }
    thread_update_priority();
  }
  sema_up (&lock->semaphore);
}
```

`lock_release`를 실행하게 되면 현재 스레드에게 우선순위를 donate해준 스레드들의 리스트인 `donors`를 순회하며 해당 lock을 이유로 우선순위를 donate해준 스레드들을 리스트에서 모두 삭제해준다. 그 후 `thread_update_priority`를 호출해 현재 스레드의 우선순위를 donate받은 우선순위를 반납한 후의 새로운 값으로 갱신해준다.

Advanced Scheduler는 앞서 설명한 바와 같이 Priority Donation이 일어나지 않으므로 해당 기능 역시 `thread_mlfqs` 플래그를 통해 실행을 막았다.

`[threads/thread.c]`

```
void
thread_update_priority(void)
{
  struct thread *current = thread_current();
  int base = current->base_priority;
  int max_priority_in_donors = PRI_MIN;
  if(!list_empty(&current->donors)){
    max_priority_in_donors = list_entry(
```

```
      list_max(
        &current->donors,
        compare_donation_priority,
        NULL),
      struct thread,
      donation_elem)->priority;
  }

  current->priority = base > max_priority_in_donors ? base
                    : max_priority_in_donors;
}
```

위의 `lock_release`에서 호출한 `thread_update_priority`는 현재 스레드의 원래의 우선순위인 `base_priority`와 `donors`들에게서 받은 우선순위들의 최댓값(`max_priority_in_donors`) 중 더 큰 쪽을 사용하도록 구현하였다.

`[threads/thread.c]`

```
void
thread_set_priority (int new_priority)
{
  thread_current ()->base_priority = new_priority;
  thread_update_priority();
  /* Try to yield if the priority of current thread is not
  the largest one */
  thread_preempt();
}
```

마지막으로 현재 스레드의 우선순위를 변경하는 `thread_set_priority` 함수까지 수정해줘야 하는데, 새로운 우선순위가 `priority`가 아닌 `base_priority`를 변경하도록 한 다음 앞서 구현한 `thread_update_priority`를 호출하여 대소비교에 따라 `priority`가 변경되도록 만들어야 Priority Donation과 관련된 모든 기능이 정상적으로 동작한다.

Advanced Scheduler

모든 구현은 핀토스 레퍼런스 B. 4.4BSD Scheduler을 기반으로 하였다.
디자인 레포트에 작성한 구현 계획에서 크게 변경된 부분은 없으나(`load_avg` 제외) design report에 모호하게, 불분명하게 작성한 부분에 대해서 구체적으로 구현하였다.

Fixed-Point Real Arithmetic

Advanced Scheduler에서 사용하는 `load_avg`와 `recent_cpu`는 실수 값을 가지는데, 일반적으로 사용되는 floating-point number는 연산이 느려 커널의 성능에 악영향을 줄 수 있기 때문에 고정 소수점 표기 방식을 이용해야만 한다. 이를 위해 `fp32` 자료형 및 이를 위한 연산 함수들을 구현하였다.

`fp32`는 디자인 레포트에서 채택한 다음 32비트의 fixed point 자료형 구조를 동일하게 채택하였으며 그 구현 또한 디자인 레포트에 명시된 바와 같다.

```
#include <stdint.h>
/* FP32: Fixed-point number type
    0    00000000000000000 00000000000000
    |    |    Decimal    | | Fractional |
    sign +---(17 bits)---+ +-(14 bits)--+
*/
```

첫번째 비트는 sign을 표기하게 하고, 이후 17비트는 decimal, 이후 14비트는 소수점 이후 부분을 표현하게 하였다. 쉽게 생각한다면 32비트로 표현된 어떠한 이진 수에 대해 소수점이 맨 뒤에 있다고 생각한 다음, 소수점을 한 칸씩 앞으로 이동시켜 14번 이동 시킨 결과를 표현한 것이며 동일한 원리로 구현하였다. 즉 기존 수를 2^{14}로 나누었다고 생각하면 되며 어떠한 bit sequence가 fp32에서 의미하는 값은 int에서 의미하는 값을 2^{14}로 나눈 값이다.

```
/* 1 << 14 = 16384 */
#define FIXED_POINT_F 16384

typedef int32_t fp32;
```

Fixed-Point 자료형의 명칭은 `fp32`으로 정하였고 이는 fixed-point 32bit을 줄인 것이다.
부호가 있는 int 자료형인 int32_t를 base로 하였다. 이는 어떠한 32bit sequence가 `int32_t`에서 의미하는 수를 2^{14}로 나눈 수가 fp32가 의미하는 수와 동일하기 때문에 이것만 고려한다면 사칙 연산을 구현하기 편리하기 때문이다.
`FIXED_POINT_F`는 fp32에서 1.0을 의미하는 수이자 위에서 말한 int와 fp32 간의 관계에서 사용되는 상수이다.

```
#define FP32_TO_FP(N)          (N * FIXED_POINT_F)
#define FP32_TO_INT(F)         (F / FIXED_POINT_F)
#define FP32_TO_INT_ROUND(F)   (F >= 0 ? \
    ((F + FIXED_POINT_F / 2) / FIXED_POINT_F) : ((F - FIXED_POINT_F / 2) / FIXED_POINT_F))
#define FP32_FP32_ADD(A, B)    (A + B)
#define FP32_FP32_SUB(A, B)    (A - B)
#define FP32_INT_ADD(A, B)     (A + FP32_TO_FP(B))
#define FP32_INT_SUB(A, B)     (A - FP32_TO_FP(B))
#define FP32_FP32_MUL(A, B)    (((int64_t) A) * B / FIXED_POINT_F)
#define FP32_INT_MUL(A, B)     (A * B)
#define FP32_FP32_DIV(A, B)    (((int64_t) A) * FIXED_POINT_F / B)
#define FP32_INT_DIV(A, B)     (A / B)
```

다음과 같이 int와 fp32 간 변환, fp32와 fp32간 사칙연산, fp32와 int간 사칙연산 매크로를 구현하였다.
이는 Pintos Reference(B. 4.4BSD Scheduler, B.6 Fixed-Point Real Arithmetic)에서 명시한 방식을 참고하여 동일하게 구현하였다.

load_avg

```
/* BSD Scheduler */
/* Average number of threads that were in the "ready" and "running" states
   over the past minute. */
static fp32 load_avg;
```

`thread.c`에 위에서 선언한 fixed point인 `fp32` 자료형의 `load_avg` 변수를 추가하였다. `load_avg`는 지난 1분 동안 ready하거나 running한 thread의 이동 평균으로 1초마다 재계산하여 갱신한다. 해당 변수는 BSS 초기화 때 0으로 함께 초기화된다.
design report에는 해당 변수를 각 `thread`에 속하는 변수로 생각하였으나 해당 변수는 global variable로 모든 스레드가 공통으로 사용하는 공통 변수로 구현하였다.

`load_avg`는 다음 공식으로 계산된다.

$$\text{load\_avg} = (59/60) * \text{load\_avg} + (1/60) * \text{ready\_threads}$$

여기서 ready_threads는 계산하는 시점에 `status`가 `THREAD_READY` 또는 `THREAD_RUNNING`인 스레드의 개수로 이 때, `idle_thread`는 제외이다.

```
static void
refresh_load_avg ()
{
  ASSERT (intr_get_level () == INTR_OFF);

  fp32 load_avg_ratio;
  fp32 ready_threads_ratio;
  fp32 load_avg_part;
  fp32 ready_threads_part;
  int ready_threads_number;
  ready_threads_number = list_size (&ready_list) + (thread_current()!=idle_thread?1:0);
  load_avg_ratio = FP32_INT_DIV (FP32_TO_FP(59), 60);
  ready_threads_ratio = FP32_INT_DIV (FP32_TO_FP(1), 60);
  load_avg_part = FP32_FP32_MUL (load_avg_ratio, load_avg);
  ready_threads_part = FP32_INT_MUL (ready_threads_ratio, ready_threads_number);

  load_avg = FP32_FP32_ADD (load_avg_part, ready_threads_part);
}
```

`load_avg`를 재계산하는 유일한 함수인 `refresh_load_avg`를 `thread.c`에 추가하였다.

load_avg 및 MLFQS 관련 함수는 interrupt가 disable된 상태에서 업데이트되길 기대하므로 interrupt disabled에 대한 assert를 추가하였다. `ready_threads_number = list_size (&ready_list) + (thread_current()!=idle_thread?1:0);`를 통해 공식 내 ready_threads를 구한다. `ready_list`의 길이를 얻어 현재 ready 중인 스레드의 개수를 얻는다.

- `idle_thread`는 running이 아니면 block 상태이기 때문에 현재 실행하고 있는 스레드가 idle_thread인지 확인하여 그렇다면 ready_threads에 포함시키지 말고 idle_thread가 아니라면 ready_threads에 포함시켜준다.
  먼저 공식의 두 항에 곱할 계수를 앞서 구현한 `fp32`의 사칙연산 매크로를 호출하여 `load_avg_ratio`, `ready_threads_ratio`에 계산한다.
  이후 공식의 각 항을 `load_avg_part`, `ready_threads_part`에 계산하고 이를 더해줘 나온 값을 `load_avg`에 덮어 씌운다. 이로써 `load_avg`를 업데이트하였다.

recent_cpu

```
struct thread
  {
    ...
    int nice;                           /* Nice (for MLFQS) */
    fp32 recent_cpu;                    /* The CPU ticks recently used by the thread. */
    struct list_elem allelem;           /* List element for all threads list. */
    ...
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

다음과 같이 `struct thread`에 `fp32` 자료형의 `recent_cpu`를 추가해 주었다.
`recent_cpu`는 각 개별 thread에 종속되는 변수로 해당 스레드가 최근에 cpu를 사용한 tick에 대한 moving weighted average이다. **해당 스레드가 cpu를 점유하고 running 할 때, tick마다 1씩 증가시킨다.(1씩 증가하는 코드는 뒤에서 설명하도록 하겠다)** 또한 1초 (`TIMER_FREQ` tick)마다 아래 계산에 의해 재계산된다.

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

```
static void
refresh_recent_cpu (struct thread *t)
{
  ASSERT (intr_get_level () == INTR_OFF);

  fp32 coefficient;
  coefficient = FP32_INT_MUL (load_avg, 2);
  coefficient = FP32_FP32_DIV (coefficient, FP32_INT_ADD(coefficient, 1));
  t->recent_cpu = FP32_INT_ADD (FP32_FP32_MUL(coefficient, t->recent_cpu), t->nice);
}
```

위의 공식을 적용해 입력 받은 `thread t`의 `recent_cpu`를 업데이트하는 함수인 `refresh_recent_cpu(t)`를 `thread.c`에 추가하였다.
overflow을 방지하기 위해 `recent_cpu`의 계수 먼저 계산하여 `coefficient`에 넣어준다.
`coefficient`를 해당 스레드의 `recent_cpu`에 곱한 뒤 해당 스레드의 `nice` 값을 곱해 스레드 `t`의 `recent_cpu`에 덮어 씌운다.
해당 함수 역시 계산 중 다른 스레드로 넘어가서는 안되기 때문에 interrupt 비활성화 여부를 확인한다.

nice

```
struct thread
  {
    ...
    int nice;                           /* Nice (for MLFQS) */
    fp32 recent_cpu;                    /* The CPU ticks recently used by the thread. */
    struct list_elem allelem;           /* List element for all threads list. */
    ...
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

다음과 같이 `struct thread`에 `int` 자료형의 `nice`를 추가해 주었다.
`nice`는 각 개별 thread에 종속되는 변수로 해당 스레드가 다른 스레드에 cpu를 얼마나 잘 양보하는지에 대한 정보를 담고 있다. 값이 클수록 더 양보를 잘하는 것을 의미한다. 해당 nice 값은 내부적으로 계산하는 값이 아닌 테스트 및 프로그램에서 `thread_set_nice` 함수를 호출하여 직접 설정해주는 것이다. 이렇게 설정된 `nice` 값은 `priority`, `recent_cpu` 을 계산하는데 함께 사용된다.

priority

앞서 priority scheduling을 위해 변경한 구현들에 의해 priority scheduling이 적용되어 스케줄링된다. 하지만 앞서 구현한 priority scheduling과 다르게 MLFQS는 priority donation을 하지 않고 `nice`, `recent_cpu` 에 따라 직접 `priority` 를 결정, 변경하여 스케줄링을 조절한다.
`priority` 는 4tick마다 주기적으로 아래 공식에 따라 계산되어 업데이트된다.

$$priority = PRI\_MAX - (recent\_cpu/4) - (nice * 2)$$

```
static int
thread_refresh_mlfqs_priority (struct thread *t)
{
  int nice = t->nice;
  fp32 recent_cpu = t->recent_cpu;
  t->priority = PRI_MAX - FP32_TO_INT(FP32_INT_DIV(recent_cpu, 4)) - nice * 2;
  return t->priority;
}
```

이처럼 해당 공식에 맞추어 매개변수로 주어진 스레드의 `priority` 를 업데이트하고 이를 반환하는 함수인 `thread_refresh_mlfqs_priority` 를 추가하였다. 입력받은 스레드 `t` 의 `recent_cpu`, `nice` 를 사용해 공식에 맞추어 fp32의 사칙연산 매크로를 이용해 priority를 계산하여 해당 스레드 `t` 의 `priority` 에 업데이트해준다. 그리고 이 값을 반환한다.

- 이 때 주의할 점은 block된 스레드, 또는 생성 중인 스레드가 아닌 이상 해당 함수는 interrupt를 비활성화한 상태로 실행되어야 한다. 생성 중인 스레드에 대한 priority 연산은 interrupt disabled가 필수가 아니므로 interrupt disabled을 별도로 검사하지는 않았다.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ...
  t->priority = priority;

  if (thread_mlfqs)
  {
    t->nice = 0;
    t->recent_cpu = 0;
    thread_refresh_mlfqs_priority (t);
  }

  t->magic = THREAD_MAGIC;
  ...
}
```

새로 생성된 스레드의 정보를 초기화하는 `thread.c` 의 `init_thread` 에서 `thread_mlfqs` 가 참이라면 mlfqs 관련 스레드 변수를 초기화하는 함수를 추가하였다.
`thread_mlfqs` 는 커맨드에 의해 설정된 MLFQS 사용 여부를 담은 변수이다. 즉 참일 때는 MLFQS를 사용하는 경우, 거짓이면 MLFQS를 사용하지 않는 것을 의미한다.
`thread_mlfqs` 가 참이라면 초기화하려는 스레드 `t` 의 `nice`, `recent_cpu` 값을 0으로 초기화하고 `thread_refresh_mlfqs_priority` 를 통해 스레드 priority를 업데이트해준다. 만약 `thread_mlfqs` 가 거짓이라면 `init_thread` 의 매개변수로 넣어준 값을 사용하여 초기화한다.

`thread_set_nice`

위에서 본 것처럼 priority는 스레드의 `recent_cpu`, `nice` 값에 의존한다. 만일 `nice` 값이 변경된다면 `priority` 또한 변경되게 되고 `priority` 변경에 의해 스위칭이 일어나야할 수도 있다. 이를 고려한 함수가 `thread_set_nice` 이다.

```
void
thread_set_nice (int nice)
{
  enum intr_level old_level;
  old_level = intr_disable ();

  struct thread *cur = thread_current ();
  int priority;
  int first_ready_priority;
  cur->nice = nice;
  priority = thread_refresh_mlfqs_priority(cur);
  first_ready_priority = list_entry (list_max (&ready_list, compare_thread_priority, NULL),
```

```
    struct thread, elem)->priority;
  if(priority < first_ready_priority){
    thread_yield();
  }
  intr_set_level (old_level);
}
```

`thread.c`에 `thread_set_nice(int nice)`를 구현하였다. 해당 함수는 현재 해당 함수를 실행한 스레드의 `nice` 값을 설정하는 함수이다. 이 때 현재 실행 중인 스레드의 `nice`를 변경하는 것 뿐만 아니라 변경된 `nice`로 인한 `priority`를 재계산하고 이에 의해 priority 순위가 변경되어 스레드 스위칭이 일어나야한다면 현재 스레드를 yield하여 이를 진행하는 함수이다.

먼저 해당 함수는 `nice`, `priority` 값을 변경하고 switching 또한 야기할 수 있다. 이 과정 중 interrupt에 의해 다른 스레드로 변경되면 안되기에 먼저 `interrupt`를 비활성화한다. 이후 매개변수로 받은 `nice` 값으로 현재 스레드의 `nice`를 변경해주고 `thread_refresh_mlfqs_priority`를 통해 현재 실행 중인 스레드의 `priority`를 업데이트해준다. 이 후 ready 상태인 스레드들을 담은 `ready_list` 중 가장 큰 값을 가진 스레드의 priority르 얻는다. 이후 ready thread 중 가장 큰 priority와 현재 스레드의 priority를 비교해 현재 스레드가 더 작다면 `thread_yield`를 통해 cpu를 양보한다. yield된 이후 다시 현재 스레드로 switch된다면 interrupt level을 되돌리는 코드부터 실행한다.

`thread_mlfqs_tick`

mlfqs를 사용할 때, timer interrupt가 발생한 tick마다 호출되어 MLFQS 관련 변수를 업데이트하는 함수인 `thread_mlfqs_tick`을 추가하였다.

```
void
thread_mlfqs_tick (int64_t ticks)
{
  enum intr_level old_level = intr_disable ();

  struct thread *cur = thread_current ();

  /* idle thread must not update recent_cpu */
  if(cur != idle_thread)
    cur->recent_cpu = FP32_INT_ADD(cur->recent_cpu, 1);

  /* update mlfqs priority per 4 ticks */
  if(ticks % 4 == 0)
  {
    thread_foreach (thread_refresh_mlfqs_priority, 0);
  }

  /* update load_avg, recent_cpu priority per 1 second */
  if(ticks % TIMER_FREQ == 0)
  {
    refresh_load_avg ();
    thread_foreach (refresh_recent_cpu, 0);
  }

  intr_set_level (old_level);
}
```

먼저 interrupt를 disable한다.

- 해당 함수 실행 중 다른 스레드로 전환되어 다른 스레드의 실행으로 인해 `nice`, `priority` 값 등이 변경시 정상적인 처리가 되지 않을 수 있다.
  이후 현재 해당 함수를 실행 중인 스레드의 `recent_cpu`를 1 증가 시켜준다. *이 때 해당 함수를 실행한 스레드가 `idle_thread`라면 증가시키지 않는다.*
  입력 받은 `ticks`이 4로 나누어 떨어진다면(입력 받은 tick은 현재 시간 tick임을 가정한다. 4tick 마다) 모든 스레드(running, ready 중인 모든 스레드, 즉 `all_list`에 있는 스레드 모두)에 대해서 `thread_refresh_mlfqs_priority`를 통해 `priority`를 업데이트 한다.
  이후 `ticks`가 `TIMER_FREQ`로 나누어 떨어지면, 즉 정확히 1초 간격이라면 `refresh_load_avg`를 통해 `load_avg`를 업데이트하고 모든 스레드에 대해 `refersh_recent_cpu`를 통해 `recent_cpu`를 업데이트한다.

`recent_cpu`, `priority`, `load_avg`의 업데이트 순서는 여러 번의 테스트 시도를 통해 모든 테스트를 통과하는 순서로 결정하였다.

- 문서 내 명확히 명시된 순서가 없다.
  마지막으로 interrupt level을 복구한다.

```
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;

  // mlfqs related info update
  if(thread_mlfqs)
    thread_mlfqs_tick (ticks);
  thread_tick ();
}
```

Timer interrupt handler 함수인 `timer_interrupt` 에서 `thread_mlfqs` 가 참일때, MLFQS를 사용할 때 `thread_mlfqs_tick` 를 호출하여 tick마다 MLFQS 관련 정보를 업데이트하도록 하게 하였다.

그 외 interface 함수들

```
int
thread_get_nice (void)
{
  return thread_current ()->nice;
}
```

현재 스레드의 `nice` 값을 출력하는 함수

```
int
thread_get_load_avg (void)
{
  return FP32_TO_INT_ROUND(FP32_INT_MUL (load_avg, 100));
}
```

fp32 매크로를 이용하여 현재 `load_avg` 에 100을 곱한 값을 int로 round하여 반환하는 함수

```
int
thread_get_recent_cpu (void)
{
  return FP32_TO_INT_ROUND(FP32_INT_MUL(thread_current ()->recent_cpu, 100));
}
```

fp32 매크로를 이용하여 현재 스레드의 `recent_cpu` 에 100을 곱한 값을 int로 round하여 반환하는 함수

발전한 점

기존 busy wait으로 구현된 `timer_sleep` 을 block, unblock을 이용한 구현으로 변경하면서 thread swtich 과정을 하나 감소킬 수 있었다.
busy wait이였던 기존 구현의 경우 스레드B가 sleep중일 때 다음처럼 작동하였다.

```
Thread A --> Timer Interrupt Handler --> Thread B --(yield)--> Thread C
```

block, unblock을 이용한 개선한 현재 구현의 경우 다음처럼 작동한다.

```
Thread A --> Timer Interrupt Handler --> Thread C
```

즉 스위칭이 1회 감소한다.
이는 단발성으로 스위칭이 1회만 감소하는 것이 아니다. 만일 round-robin scheduler을 사용하고 스레드가 2개라고 하고 한 스레드가 sleep 중일 때, 기존 구현이라면 4 tick 마다 sleep하고 있는 thread로 switch 되므로 4tick마다 쓸모없는 스위칭이 발생한다. 하지만 현재 변경한 구현에서는 그렇지 않다.
기존 구현은 priority scheduler 사용시에도 큰 문제다. 큰 priority를 가진 thread가 sleep 중이라면 sleep 중인 동안 계속 cpu를 점유할 것이다. 하지만 변경된 구현에서는 sleep시 block되므로 cpu를 점유하지 않아 해당 문제가 발생하지 않는다.

또한 모든 스레드를 동일한 시간의 CPU를 사용하여 돌아가며 처리하던 기존의 FIFO 방식의 스레드 스케줄링을 Priority Scheduling을 통해 각 스레드에 우선순위를 부여하여 중요한 작업이 비교적 먼저 실행될 수 있도록 변경하였고, 이를 구현하며 발생하는 부가적인 문제인 Multiple Donation과 Nested Donation에 대한 처리 역시 성공적으로 마칠 수 있었다. 이를 구현함으로써 유저 프로그램 혹은 커널이 매긴 스레드 우선순위에 기반하여 이전보다 더 효율적으로 CPU 자원을 사용하는 것이 가능해졌다.

MLFQS, Advanced scheduler는 기존 round-robin 스케줄러와 다르게 `nice`를 통해 해당 스레드가 cpu를 얼마나 잘 양보해줘도 되는지 설정하여 적절하게 스레드가 스케줄링되도록 조절할 수 있었다. 또한 `recent_cpu`를 통해 어떤 스레드가 실행한 시간이 길어질수록, cpu를 점유한 시간이 길어질수록 priority를 내려가 공평하게 스케줄링되도록 조절되었다. 이는 어떤 작업을 하든 상관없이 모든 스레드를 번갈아가며 실행하는 기존 round-robin 구현에 비해 합리적인 작동 방식이다.

한계

`struct thread`에 `wake_up_tick`, `sleep_elem`을 추가하며 Device 중 하나인 Timer와 Thread가 강하게 엮여 있는 코드 의존성을 가지게 된다. 이런 코드 의존성은 timer가 변경되었을 때 `thread`의 변화까지 야기한다. `thread`로부터 해당 변수들을 완벽히 분리하여 alarm clock 및 sleep을 구현할 수 있으면 더 좋았을 것이다.

Priority Donation의 도입부에서 design과는 다르게 의도적으로 스레드 리스트를 우선순위에 따른 정렬 상태로 관리하지 않았다는 점을 언급했다. 정렬 상태를 유지하여 관리할 때와 하지 않을 때의 시간 복잡도를 나타내면 다음과 같다.

	Insert	**Find**	**Delete**	**Modify**
정렬O	`O(n)`	`O(1)`	`O(1)`	`O(nlogn)`
정렬X	`O(1)`	`O(n)`	`O(n)`	`O(1)`

따라서 두 구현 방식의 성능 차이는 어느 한 쪽이 절대적으로 높고 낮다고 하기보단 리스트 삽입, 탐색, 삭제, 변경 연산이 각각 얼마나 일어나는지에 따라 달라진다고 할 수 있다. 이번 Assn 1의 경우 만들고자 하는 OS의 일부 기능들이 제대로 구현되었는지를 시험하는 테스트들이 대부분이었으므로 어느 한 쪽의 구현 방식이 절대적으로 좋다고 결론 짓기에는 어렵고, 추후 조금 더 일반적인 사용 상황을 가정하는 테스트 케이스가 있을 시 이를 이용하여 벤치마킹을 수행할 때 두 구현 방식의 성능이 어떻게 달리지는지 비교해볼 계획이다.

배운 점

과제 목표를 달성하기 위해 코드를 작성하고 디버깅하는 과정에서 Assertion을 이용한 디버깅이 매우 유용함을 깨달았다. Assertion을 추가한다면 test에서도 어떤 assertion에 의해 실패하였는지 명확히 보여주기에 디버깅에 이점이 있었다. 그렇기에 어떤 작업을 수행하는 함수를 구현하기 전에 해당 함수가 interrupt를 비활성화한 상황에서 실행되어야 할지, 매개변수에 대해서는 어떤 검증을 해야할지 등을 생각하고 함수를 만드는 습관이 생기게 되고 이는 각 함수의 기능을 명확히 설계하고 이해하고 이후 디버깅하는데 도움이 되었다.

또한 gdb를 이용한 동적 디버깅 역시 오류를 해결하고 실패한 테스트의 원인을 찾는 데 큰 도움이 되었다. list 관련 함수를 일부 잘못 사용하여 `ready_list`에 스레드가 없는 상태로 무한 루프를 돌거나 커널 패닉이 발생하는 버그를 겪었는데, pintos 바이너리를 실행할 때 `--gdb` 플래그를 추가해 다른 터미널에서 gdb를 이용해 프로세스를 붙여 디버깅을 하였다. 심볼이 모두 남아있어 특정 변수를 찍어보는 것도 편리했고, 특정 메모리의 값이 변경되었을 때 프로세스를 일시정지 시키는 `watchpoint` 기능도 매우 유용했다. 또한 일부 테스트의 경우 한 번 실행 시 실제 시간으로 약 1~2분을 기다려야 다음 로직이 수행되는 경우가 많았는데, 디버깅 중 `ticks` 변수를 강제로 변경해 불필요한 대기 시간을 줄일 수 있었다.