[CSED312] OS project 1 Design Report

- 20180673 하재현
- 20180501 최진수

Analysis of the current thread system

Thread structure

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

    /* Shared between thread.c and synch.c. */
    struct list_elem elem;
  }
```

구조체 thread는 `threads/thread.h`에 아래 코드와 같이 정의되어 있다.

멤버를 하나하나 살펴보자면,

- `tid`는 thread id를 나타내고, status는 thread의 state를 나타낸다.
- `name`은 debugging을 목적으로 thread의 이름을 저장하고 있다
- `stack`은 현재 thread의 stack pointer을 의미한다.
- `priority`는 아래에서 더 구체적으로 다루겠지만 thread의 우선 순위를 나타낸다.
- `elem`는 중의적인 역할을 띠고 있는데, 하나는 run queue에서의 element로서의 역할, 다른 하나는 semaphore wait list에서의 element로서의 역할이다. elem이 이런 두 가지의 역할을 할 수 있는 것은 그 두 역할이 mutually exclusive하기 때문이다. 만약 thread가 ready state에 있다면 그것은 run queue에 있을 것이고, blocked state에 있다면 semaphore wait list에 있을 것이다.

모든 thread structure은 각자 4kB page를 차지한다. 위의 코드에서 보이는 구조체 내 원소들은(structure 자체는) page offset 0에 저장되고, 나머지 page는 offset 4kB(top of the page)로부터 시작해서 아래쪽으로 자라는, thread의 kernel stack에 위치하게 된다.

Initializing threading system

```
void thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);

  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);

  /* Set up a thread structure for the running thread. */
  initial_thread = running_thread ();
  init_thread (initial_thread, "main", PRI_DEFAULT);
  initial_thread->status = THREAD_RUNNING;
  initial_thread->tid = allocate_tid ();
}
```

`void thread_init(void)` 함수는 threading system을 initializing하는 역할을 한다. (첫 번째 kernel thread를 형성하는 역할을 한다)

기본적으로 현재 돌아가고 있는 code를 thread로 transforming하는 것이다.

구체적으론 run queue와 tid lock을 initializing하며, 이 함수를 호출한 이후에는 아래에서 다룰 `thread_create()` 함수를 이용해 thread를 생성하기 전에 page allocator를 initialize해야만 한다.

Thread creation

```
tid_t thread_create (const char *name, int priority,
                thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;
```

```
  /* Stack frame for switch_entry(). */
  ef = alloc_frame (t, sizeof *ef);
  ef->eip = (void (*) (void)) kernel_thread;

  /* Stack frame for switch_threads(). */
  sf = alloc_frame (t, sizeof *sf);
  sf->eip = switch_entry;
  sf->ebp = 0;

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

  return tid;
}
```

`thread_create()`는 새로운 kernel thread를 생성하는 함수이다.

주요 인자로 name(이름), priority(우선 순위), 수행할 function pointer을 넘겨준다.

return value는 creation이 성공적으로 완료되면 thread identifier, creation이 완료되지 못했을 때에는 TID_ERROR이다.

thread가 creating된다는 것은 scheduling이 이루어질 새로운 context를 creating한다는 의미이다.

처음 thread가 scheduled되고 돌아가기 시작하면 인자로 넘겨지는 function으로부터 시작해서 그 context에서 execute된다.

만약 인자로 넘겨진 function이 return 한다면 `thread_create()` 함수도 terminate하게 된다.

Pintos에서 돌아가는 모든 thread는 pintos 내에서 돌아가는 일종의 mini program으로 볼 수 있다.

`thread_create()` 함수의 동작 방법은 다음과 같다. 일단 `struct thread`를 위한 공간을 할당하고, `init_thread()`를 이용해 그 멤버를 초기화한다.

context switching이 일어날 수 있도록 3개의 stack frame(kernel thread, switch entry, switch thread를 위한)을 할당한다.

생성된 thread는 blocked state로 초기화되는데, return 전에 unblocked 상태로 만들어 새로운 thread가 schedule 될 수 있게 해준다.

Thread scheduler

```
void
thread_start (void)
{
  /* Create the idle thread. */
  struct semaphore idle_started;
  sema_init (&idle_started, 0);
  thread_create ("idle", PRI_MIN, idle, &idle_started);
```

```
  /* Start preemptive thread scheduling. */
  intr_enable ();

  /* Wait for the idle thread to initialize idle_thread. */
  sema_down (&idle_started);
}
```

프로그램이 시작할 때 `thread_start()` 함수에 의해 scheduler가 시작된다.

`thread_start()`는 idle thread를 생성하며, 이것은 다른 어떤 thread도 준비되지 않았을 때 schedule되는 thread이다.

그리고 interrupt가 enable되고, 이것은 scheduler를 enable하는 결과를 낳는데 이것은 scheduler가 timer interrupt으로부터의 return에 `intr_yield_on_return()` 함수를 이용해 돌아가기 때문이다.

임의의 시간에 돌아가는 thread는 항상 단 하나여야 한다.

돌아가는 하나의 thread를 제외한 나머지 thread는 inactivate된 상태를 유지해야만 한다.

그리고 다음으로 돌아갈 thread를 결정해주는 것이 thread scheduler이다.

만약 어느 thread도 돌아갈 준비가 되지 않았다면 idle thread가 돌아가게 된다. 즉, thread 사이의 switch가 일어나게 하는 것이 scheduler의 역할이 된다.

`thread_tick()` 함수는 모든 timer tick마다 timer interrupt마다 호출되는데, thread statistics를 tracking하고 time slice가 만료되었을 때 scheduler을 trigger하는 역할을 한다.

`into_yield_on_return()` 함수는 interrupt context로, `thread_yield()` 함수를 interrupt return 직전에 호출한다.

이것은 Timer interrupt handler에서 thread의 time slice가 만료되었을 때 새로운 thread가 호출되도록 하는 역할을 한다.

Thread completion

```
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif

  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();
  list_remove (&thread_current()->allelem);
  thread_current ()->status = THREAD_DYING;
  schedule ();
```

```
    NOT_REACHED ();
  }
```

완료는 `thread_exit()` 함수의 호출에 의해 일어난다.

thread가 task 수행을 완료할 때 `thread_exit()` 함수의 호출을 통해 thread state를 THREAD_DYING으로 설정하고 thread switching을 일어나게 한다.

그리고 page를 free하게 된다.

Analysis of the current synchronization

multi thread 환경을 구현할 때에는 thread간의 자원 공유에 대해서 매우 신경을 써야 한다.

만약 공유 자원에 여러 thread가 동시에 접근하게 되면 race condition이 발생하여 원치 않은 결과를 만들어낼 수 있고, 때로는 시스템 전체를 붕괴시키기도 한다.

이런 불상사를 막기 위해 Synchronization, 즉 스레드들이 수행되는 시점을 적절히 조절하는 것이 필요하다.

1. Semaphores

> Pintos synchronization의 핵심

Semaphore 란 아래 두 개의 연산자를 이용해서 atomically 수정 가능한 nonnegative integer을 의미한다.

- **"Down" or "P"**: 값이 양수가 될 때까지 기다리고, 감소시킴
- **"Up" or "V"** : 값을 증가시킴 (그리고 waiting thread가 있으면 그 중 하나를 실행시킴)

간단한 사용법은 다음과 같다.

thread는 critical section에 진입하기 전 `down`을 해서 semaphore 값을 0으로 만든다.

이후 이 critical section에 진입하는 다른 thread는 `down`을 하지만 양수가 아니므로 기다려야 한다.

첫 번째로 진입한 thread가 끝나면, `up`을 하면서 대기중인 thread를 깨운다.

깨워진 thread는 semaphore 값을 확인하여 양수라면 critical section에 진입한다.

만약 초기값을 1보다 크게 설정하면 그만큼 여러 스레드를 동시에 진입시킬 수 있다.

자세한 원리를 알아보기 위해, Semaphore의 구조체를 알아보자.

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

Semaphore은 현재 값을 저장하는 멤버와, waiting thread list 멤버가 있다.

```
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema != NULL);

  sema->value = value;
  list_init (&sema->waiters);
}
```

Semaphore의 초기화는 초기값의 설정과 list의 초기화로 이루어져있다.

연산 동작을 알아보자.

```
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 ()->elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

down은 semaphore의 값을 확인하여 0이 아니라면 곧바로 값을 1 감소시키지만, 0이라면 waiting list에 넣고, thread를 block한다.

나중에 block이 풀리면, semaphore의 값이 양수임을 확인하고, 값을 1 감소시킨다.

```
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->waiters),
                                struct thread, elem));
  sema->value++;
```

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

up은 priority에 상관없이, **FIFO**로 waiting thread를 선택하여 unblock 한 후 값을 1 증가시킨다.

2. Locks

lock은 1로 초기화된 semaphore에 owner 기능을 추가하고, up, down을 각각 release, acquire로 표현한 것이다.

owner 기능이란, lock을 한 thread를 owner로 지정하여, 이 owner만이 lock을 release할 수 있도록 하는 기능이다.

구체적인 동작을 알아보기 위해 구조체부터 살펴보자.

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

owner를 기록하기 위한 멤버와 semaphore 기능을 사용하기 위해 semaphore 객체를 멤버로 가지고 있다.

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

  lock->holder = NULL;
  sema_init (&lock->semaphore, 1);
}
```

lock의 초기화는 우선 semaphore을 controll 용도로 사용하기 위해 1로 초기화하고, 초기 owner는 없으므로 NULL로 초기화한다.

이제 연산 동작을 알아보자.

```
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 = thread_current ();
}
```

acquire은 recursive한 aquire인지 체크하고, 그렇지 않다면 `semaphore down` 동작을 실행하며 현재 thread를 owner로 등록한다.

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

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

release는 현재 thread가 lock의 owner인지 체크하고, 그렇다면 `semaphore up` 동작을 실행하며 owner를 초기화한다.

3. Monitors

> semaphore나 lock보다 상위 레벨의 synchronization 기법

Monitor의 주요 기능은 특정 condition이 만족될 때까지 thread를 blocking 하는 것이다.

Monitor는 synchronized 되어야 하는 데이터, lock(일명 monitor lock), 그리고 condition variable들로 이루어져 있다.

구조가 조금 복잡하다. 구조체부터 알아보자.

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

```
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
  };
```

`condition` 구조체는 condition variable을 의미하며, 해당 variable에 묶여있는 waiting thread list를 멤버로 가지고 있다.

`semaphore_elem` 구조체는 conditional lock을 구현하기 위한 것이다.

해당 구조체의 `semaphore`을 통해 `cond_wait`과 `cond_signal`이 소통한다.

```
void
cond_init (struct condition *cond)
{
  ASSERT (cond != NULL);

  list_init (&cond->waiters);
}
```

초기화는 각 condition의 waiting thread list를 초기화한다.

```
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);
}
```

`wait`은 lock을 release하고, `semaphore_elem`을 이용해서 condition이 만족될 때까지 blocking 한다.

```
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);
}
```

`signal`은 **FIFO**로 해당 condition이 만족되길 기다리는 thread 중 하나를 선택하여 blocking을 해제한다.

```
void
cond_broadcast (struct condition *cond, struct lock *lock)
```

```
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);

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

`broadcast` 는 해당 condition이 만족되길 기다리는 모든 thread들에 대해 blocking을 해제한다.

Solutions

1. Alarm Clock

2. Priority scheduling

Current Implementation

현재는 단순히 Rount-robin scheduling을 택하고 있다.

```
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);
}
```

```
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);
}
```

```
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);
}
```

구현은 ready_list를 circular FIFO queue로 구현하는 것이 핵심이다.

코드를 분석해보면 thread가 yield되면 `list_push_back` 을 하고, 다음으로 실행할 thread는 `list_pop_front` 를 해서 정하는, queue 구조를 취하고 있음을 확인할 수 있다.

New Implementation

Tests

통과해야 하는 test들은 다음과 같다.

- **priority-change** : Verifies that lowering a thread's priority so that it is no longer the highest-priority thread in the system causes it to yield immediately
- **priority-fifo** : Creates several threads all at the same priority and ensures that they consistently run in the same round-robin order
- **priority-preempt** : Ensures that a high-priority thread really preempts
- **priority-sema** : Tests that the highest-priority thread waiting on a semaphore is the first to wake up
- **priority-condvar** : Tests that cond_signal() wakes up the highest-priority thread waiting in cond_wait()
- **priority-donate-one** : The main thread acquires a lock. Then it creates two higher-priority threads that block acquiring the lock, causing them to donate their priorities to the main thread. When the main thread releases the lock, the other threads should acquire it in priority order
- **priority-donate-multiple** : The main thread acquires locks A and B, then it creates two higher-priority threads. Each of these threads blocks acquiring one of the locks and thus donate their priority to the main thread. The main thread releases the locks in turn and relinquishes its donated priorities
- **priority-donate-multiple2** : The main thread acquires locks A and B, then it creates three higher-priority threads. The first two of these threads block acquiring one of the locks and thus donate their priority to the main thread. The main thread releases the locks in turn and relinquishes its donated

priorities, allowing the third thread to run. In this test, the main thread releases the locks in a different order compared to priority-donate-multiple.c

- **priority-donate-chain** : The main thread set its priority to PRI_MIN and creates 7 threads (thread 1..7) with priorities PRI_MIN + 3, 6, 9, 12, ... The main thread initializes 8 locks: lock 0..7 and acquires lock 0. When thread[i] starts, it first acquires lock[i] (unless i == 7.) Subsequently, thread[i] attempts to acquire lock[i-1], which is held by thread[i-1], except for lock[0], which is held by the main thread. Because the lock is held, thread[i] donates its priority to thread[i-1], which donates to thread[i-2], and so on until the main thread receives the donation. After threads[1..7] have been created and are blocked on locks[0..7], the main thread releases lock[0], unblocking thread[1], and being preempted by it/ Thread[1] then completes acquiring lock[0], then releases lock[0], then releases lock[1], unblocking thread[2], etc. Thread[7] finally acquires & releases lock[7] and exits, allowing thread[6], then thread[5] etc. to run and exit until finally the main thread exits.

- **priority-donate-nest** : Low-priority main thread L acquires lock A. Medium-priority thread M then acquires lock B then blocks on acquiring lock A. High-priority thread H then blocks on acquiring lock B. Thus,thread H donates its priority to M, which in turn donates itto thread L.

- **priority-donate-sema** : Low priority thread L acquires a lock, then blocks downing a semaphore. Medium priority thread M then blocks waiting on the same semaphore. Next, high priority thread H attempts to acquire the lock, donating its priority to L. Next, the main thread ups the semaphore, waking up L. L releases the lock, which wakes up H. H "up"s the semaphore, waking up M. H terminates, then M, then L, and finally the main thread.

Priority scheduling

Data Structure

scheduling을 위해서 별도로 멤버를 추가줄 필요는 없다.

Create

```
/**
* In thread.c
* comparator function, signature of typedef list_less_func
* @return a's priority >? b's priority
*/
bool priority_compare(struct list_elem* a, struct list_elem* b, void* aux){}
```

Change

- `thread_yield()`, `thread_unblock()`

  : `list_push_back (&ready_list, &cur->elem)` -> `list_insert_ordered(&ready_list, &cur->elem, priority_compare, NULL);`

- `thread_create()`

```
{
  if(newThread.priority > currentThread.priority)
    thread_yield();
}
```

- `thread_set_priority()`

```
{
  if(newPriority < currentPriority)
    thread_yield();
}
```

Algorithm

priority 순서로 thread를 선택하기 위해, ready_list에 priority 순으로 thread를 추가하도록 한다.

```
list_insert_ordered(&ready_list, &cur->elem, priority_compare, NULL);
```

`priority_compare()`를 사용하여 priority 순서대로 ready_list에 집어넣는 구문이다.

`thread_yield()`, `thread_unblock()` 의 `list_push_back (&ready_list, &cur->elem)`를 이 구문으로 대체한다.

그리고 scheduling을 다시 해야 하는 시점은

- 새로 만들어진 thread가 priroity가 현재보다 높아짐
- priority가 이전보다 작아지게 set 됨

이므로, 각각의 경우에 맞추어 `thread_yield()`를 호출해준다.

Priority donation

Data Structure

```
struct lock
  {
    ...
    bool is_donated;
  }
```

```
struct thread
  {
    ...
```

```
    int original_priority;
    list donators; //struct thread
    struct lock lock_on_wait;
  }
```

Create

문법 상관없이 수도코드처럼 썼습니다.

```
/**
* In synch.c
* Donate priority from current thread to lock->holder
* Register in current thread's donated_priorities
* @param struct lock
*/
void priority_donate(struct lock* lock){
  thread_current()->lock_on_wait = lock;
  lock->holder->priority = thread_current()->priority;
  lock->holder->donators.INSERT_ORDERED_BY_PRIORITY(thread_current());
  lock->is_donated = true;
}
```

```
/**
* In synch.c
* Restore current thread's priority
* If no donators left, then set to initial value
* Else set as highest priority among donators.
* @param struct lock
*/
void priority_restore(struct lock* lock){
  if(donators.ISEMPTY)
    thread_current()->priority = thread_current()->original_priority;
  else {
    thread_to_remove =  max_priority(donators.filter(() => donator->lock_on_wait
== lock));
    thread_current()->priority = thread_to_remove->priority;
    donators.remove(thread_to_remove);
  }
}
```

```
/**
* In synch.c
* Check whether needs donation
* @return true (If donation needed)
* @return false (If donation is not needed)
*/
bool require_donation(struct lock* lock){
```

```
  return lock->holder != NULL && lock->holder->priority <
thread_current().priority ? true : false;
}
```

```
/**
* In thread.c
* @return current thread's priority
*/
int thread_get_priority(){
  return thread_current().priority;
}
```

```
/**
* In thread.c
* Determine whether call thead_yield is not contained in this code
* See above
* @param new priority
*/
int thread_set_priority(int new_priority){
  thread_current().priority = new_priority;
}
```

Change

- lock_aquire()

```
/**
* In synch.c
* If donation needs, do donation
* Else store original priority
*/
{
  if(require_donation(lock))
    priority_donate(lock);
  else
    thread_current()->original_priority = thread_current()->priority;
}
```

- lock_release()

```
/**
* In synch.c
* If donationed lock, restore donation
* Else do nothing (Except original behavior)
```

```
*/
{
  if(lock.is_donated)
    priority_restore(lock);
}
```

- `sema_up()`

```
/**
* To pass priority-sema test, pop highest priority thread.
*/
{
  //thread_unblock (list_entry (list_pop_front (&sema->waiters), struct
thread, elem));
  list_sort(&sema->waiters, priority_compare, 0);
  thread_unblock (list_entry (list_pop_front (&sema->waiters), struct
thread, elem));
}
```

- `cond_signal()`

```
/**
* To pass priority-condvar test, pop highest priority element.
*/
{
  /*
  if (!list_empty (&cond->waiters))
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
  */
  if (!list_empty (&cond->waiters)) {
    list_sort(&cond->waiters, priority_compare, 0);
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->semaphore);
  }
}
```

Algorithm

알고리즘은 크게 4가지의 흐름으로 나눌 수 있다.

- Donation이 필요하지 않은 `lock_aquire()`

  이 경우, 이후에 donation이 들어올 것을 대비하여 original priority만 저장해두면 된다.

- Donation이 필요한 `lock_aquire()`

  이 경우, lock holder에게 priority를 donate하고, 해당 holder의 donators에 가입한다.

이후 이 donators를 통해 release가 이뤄진다.

또한, donators를 탐색할 때, 어떤 lock에 의해 donating이 된건지 알아야 하므로 lock_on_wait 값을 저장해준다.

- Donation이 되어있지 않은 `lock_release()`

  이 경우, 별다른 처리가 필요하지 않으므로 아무런 추가 동작이 없다.

- Donation이 되어있는 `lock_release()`

  이 경우, priority donation을 roll back 해야 한다.

  그런데, 여러 thread에서 donation을 받았을 가능성이 있으므로, donators의 priority 값들 중에서 가장 높은 값으로 roll back한다.

  그리고 선택된 thread는 donator list에서 삭제한다.

3. Advanced scheduler

Current Implementation

New Implementation