과제2 결과보고서

IEEE Code of Ethics

[출처: http://www.ieee.org]

We, the members of the IEEE, in recognition of the importance of our technologies in affecting the quality of life throughout the world, and in accepting a personal obligation to our profession, its members and the communities we serve, do hereby commit ourselves to the highest ethical and professional conduct and agree:

1. to accept responsibility in making decisions consistent with the safety, health and welfare of the public, and to disclose promptly factors that might endanger the public or the environment;

2. to avoid real or perceived conflicts of interest whenever possible, and to disclose them to affected parties when they do exist;

3. to be honest and realistic in stating claims or estimates based on available data;

4. to reject bribery in all its forms;

5. to improve the understanding of technology, its appropriate application, and potential consequences;

6. to maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations;

7. to seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others;

8. to treat fairly all persons regardless of such factors as race, religion, gender, disability, age, or national origin;

9. to avoid injuring others, their property, reputation, or employment by false or malicious action;

10. to assist colleagues and co-workers in their professional development and to support them in following this code of ethics.

위 IEEE 윤리헌장 정신에 입각하여 report를 작성하였음을 서약합니다.

학 부: 전자공학과
제출일: 2021. 11. 7.
과목명: 전자공학운영체계
교수명:
분 반:
성 명: 윤건

전자공학운영체계 과제2 결과보고서

201721045 윤건

□ 소스 코드

○ timer.c

```
#include "devices/timer.h"
#include <debug.h>
#include <inttypes.h>
#include <round.h>
#include <stdio.h>
#include "devices/pit.h"
#include "threads/interrupt.h"
#include "threads/synch.h"
#include "threads/thread.h"

/* See [8254] for hardware details of the 8254 timer chip. */
#if TIMER_FREQ <19
#error 8254 timer requires TIMER_FREQ >=19
#endif
#if TIMER_FREQ >1000
#error TIMER_FREQ <=1000 recommended
#endif
/* Number of timer ticks since OS booted. */
static int64_t ticks;
/* Number of loops per timer tick.
   Initialized by timer_calibrate(). */
static unsigned loops_per_tick;
static intr_handler_func timer_interrupt;
static bool too_many_loops (unsigned loops);
static void busy_wait (int64_t loops);
static void real_time_sleep (int64_t num, int32_t denom);
static void real_time_delay (int64_t num, int32_t denom);
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}
/* Calibrates loops_per_tick, used to implement brief delays. */
```

```
void
timer_calibrate (void)
{
  unsigned high_bit, test_bit;
  ASSERT (intr_get_level () == INTR_ON);
  printf ("Calibrating timer...  ");
  /* Approximate loops_per_tick as the largest power-of-two
     still less than one timer tick. */
  loops_per_tick = 1u <<10;
  while (!too_many_loops (loops_per_tick <<1))
    {
      loops_per_tick <<=1;
      ASSERT (loops_per_tick !=0);
    }
  /* Refine the next 8 bits of loops_per_tick. */
  high_bit = loops_per_tick;
  for (test_bit = high_bit >>1; test_bit != high_bit >>10; test_bit >>=1)
    if (!too_many_loops (high_bit | test_bit))
      loops_per_tick |= test_bit;
  printf ("%'"PRIu64" loops/s.₩n", (uint64_t) loops_per_tick *
TIMER_FREQ);
}
/* 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, which
   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. */
/* Edited for assignment 1 */
void timer_sleep (int64_t ticks) {
        int64_t start = timer_ticks ();
        ASSERT (intr_get_level () == INTR_ON);
        thread_sleep(start+ticks);
}
```

```c
/* Sleeps for approximately MS milliseconds.  Interrupts must be
   turned on. */
void
timer_msleep (int64_t ms)
{
  real_time_sleep (ms, 1000);
}
/* Sleeps for approximately US microseconds.  Interrupts must be
   turned on. */
void
timer_usleep (int64_t us)
{
  real_time_sleep (us, 1000 *1000);
}
/* Sleeps for approximately NS nanoseconds.  Interrupts must be
   turned on. */
void
timer_nsleep (int64_t ns)
{
  real_time_sleep (ns, 1000 *1000 *1000);
}
/* Busy-waits for approximately MS milliseconds.  Interrupts need
   not be turned on.
   Busy waiting wastes CPU cycles, and busy waiting with
   interrupts off for the interval between timer ticks or longer
   will cause timer ticks to be lost.  Thus, use timer_msleep()
   instead if interrupts are enabled. */
void
timer_mdelay (int64_t ms)
{
  real_time_delay (ms, 1000);
}
/* Sleeps for approximately US microseconds.  Interrupts need not
   be turned on.
   Busy waiting wastes CPU cycles, and busy waiting with
   interrupts off for the interval between timer ticks or longer
   will cause timer ticks to be lost.  Thus, use timer_usleep()
   instead if interrupts are enabled. */
void
timer_udelay (int64_t us)
{
  real_time_delay (us, 1000 *1000);
}
/* Sleeps execution for approximately NS nanoseconds.  Interrupts
   need not be turned on.
   Busy waiting wastes CPU cycles, and busy waiting with
```

```
   interrupts off for the interval between timer ticks or longer
   will cause timer ticks to be lost.  Thus, use timer_nsleep()
   instead if interrupts are enabled.*/
void
timer_ndelay (int64_t ns)
{
  real_time_delay (ns, 1000 *1000 *1000);
}
/* Prints timer statistics. */
void
timer_print_stats (void)
{
  printf ("Timer: %"PRId64" ticks₩n", timer_ticks ());
}

/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
/* Returns true if LOOPS iterations waits for more than one timer
   tick, otherwise false. */
static bool
too_many_loops (unsigned loops)
{
  /* Wait for a timer tick. */
  int64_t start = ticks;
  while (ticks == start)
    barrier ();
  /* Run LOOPS loops. */
  start = ticks;
  busy_wait (loops);
  /* If the tick count changed, we iterated too long. */
  barrier ();
  return start != ticks;
}
/* Iterates through a simple loop LOOPS times, for implementing
   brief delays.
   Marked NO_INLINE because code alignment can significantly
   affect timings, so that if this function was inlined
   differently in different places the results would be difficult
   to predict. */
static void NO_INLINE
busy_wait (int64_t loops)
```

```
{
  while (loops-->0)
    barrier ();
}
/* Sleep for approximately NUM/DENOM seconds. */
static void
real_time_sleep (int64_t num, int32_t denom)
{
  /* Convert NUM/DENOM seconds into timer ticks, rounding down.

        (NUM / DENOM) s
     ---------------------- = NUM * TIMER_FREQ / DENOM ticks.
     1 s / TIMER_FREQ ticks
  */
  int64_t ticks = num * TIMER_FREQ / denom;
  ASSERT (intr_get_level () == INTR_ON);
  if (ticks >0)
    {
      /* We're waiting for at least one full timer tick.  Use
         timer_sleep() because it will yield the CPU to other
         processes. */
      timer_sleep (ticks);
    }
  else
    {
      /* Otherwise, use a busy-wait loop for more accurate
         sub-tick timing. */
      real_time_delay (num, denom);
    }
}
/* Busy-wait for approximately NUM/DENOM seconds. */
static void
real_time_delay (int64_t num, int32_t denom)
{
  /* Scale the numerator and denominator down by 1000 to avoid
     the possibility of overflow. */
  ASSERT (denom % 1000 ==0);
  busy_wait (loops_per_tick * num /1000 * TIMER_FREQ / (denom
/1000));
}
```

해당 코드에서 우선 timer_sleep 함수가 수정되었다. 이는 thread_sleep 함수를 호출하고 (현재 시간 + 설정 시간)을 전달하여 해당 시간까지 잠들어있도록 한다.

○ thread.h

```
#ifndef THREADS_THREAD_H
#define THREADS_THREAD_H
#include <debug.h>
#include <list.h>
#include <stdint.h>
/* States in a thread's life cycle. */
enum thread_status
  {
    THREAD_RUNNING,     /* Running thread. */
    THREAD_READY,       /* Not running but ready to run. */
    THREAD_BLOCKED,     /* Waiting for an event to trigger. */
    THREAD_DYING        /* About to be destroyed. */
  };
/* Thread identifier type.
   You can redefine this to whatever type you like. */
typedef int tid_t;
#define TID_ERROR ((tid_t) -1)          /* Error value for tid_t. */
/* Thread priorities. */
#define PRI_MIN 0                       /* Lowest priority. */
#define PRI_DEFAULT 31                  /* Default priority. */
#define PRI_MAX 63                      /* Highest priority. */
/* A kernel thread or user process.
   Each thread structure is stored in its own 4 kB page.  The
   thread structure itself sits at the very bottom of the page
   (at offset 0).  The rest of the page is reserved for the
   thread's kernel stack, which grows downward from the top of
   the page (at offset 4 kB).  Here's an illustration:
        4 kB +---------------------------------+
             |          kernel stack           |
             |                |                |
             |                |                |
             |                V                |
             |         grows downward          |
             |                                 |
             |                                 |
             |                                 |
             |                                 |
             |                                 |
             |                                 |
             |                                 |
             |                                 |
             +---------------------------------+
             |              magic              |
```

```
             |                    :                    |
             |                    :                    |
             |                   name                  |
             |                  status                 |
        0 kB +-------------------------------------+
   The upshot of this is twofold:
      1. First, `struct thread' must not be allowed to grow too
         big.  If it does, then there will not be enough room for
         the kernel stack.  Our base `struct thread' is only a
         few bytes in size.  It probably should stay well under 1
         kB.
      2. Second, kernel stacks must not be allowed to grow too
         large.  If a stack overflows, it will corrupt the thread
         state.  Thus, kernel functions should not allocate large
         structures or arrays as non-static local variables.  Use
         dynamic allocation with malloc() or palloc_get_page()
         instead.
   The first symptom of either of these problems will probably be
   an assertion failure in thread_current(), which checks that
   the `magic' member of the running thread's `struct thread' is
   set to THREAD_MAGIC.  Stack overflow will normally change this
   value, triggering the assertion. */
/* The `elem' member has a dual purpose.  It can be an element in
   the run queue (thread.c), or it can be an element in a
   semaphore wait list (synch.c).  It can be used these two ways
   only because they are mutually exclusive: only a thread in the
   ready state is on the run queue, whereas only a thread in the
   blocked state is on a semaphore wait list. */
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;              /* List element. */
#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
    /* Owned by thread.c. */
```

```
    unsigned magic;                     /* Detects stack overflow. */

                              /*      added for assignment 1        */
    int64_t timeout;                      /* time that thread should return
*/

                              /*      added for assignment 2        */
    int      orig_priority;             /* keeps the original priority */
    struct   list donation_list;        /* keeps the list of donations */
    struct   list_elem donation_elem;     /* element of that list */
    struct   lock *lock_waiting;          /* waiting because of lock */
  };
/* 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;
void thread_init (void);
void thread_start (void);
void thread_tick (void);
void thread_print_stats (void);
typedef void thread_func (void *aux);
tid_t thread_create (const char *name, int priority, thread_func *, void *);
void thread_block (void);
void thread_unblock (struct thread *);
struct thread *thread_current (void);
tid_t thread_tid (void);
const char *thread_name (void);
void thread_exit (void) NO_RETURN;
void thread_yield (void);
/* Performs some operation on thread t, given auxiliary data AUX. */
typedef void thread_action_func (struct thread *t, void *aux);
void thread_foreach (thread_action_func *, void *);
int thread_get_priority (void);
void thread_set_priority (int);
int thread_get_nice (void);
void thread_set_nice (int);
int thread_get_recent_cpu (void);
int thread_get_load_avg (void);

                                        /*      added for assignment 1        */
void thread_sleep (int64_t ticks);
void thread_wakeup (void);
bool timeout_cmp (const struct list_elem *cmp1, const struct list_elem
*cmp2, void *aux UNUSED);

                                        /*      added for assignment 2        */
```

```
void priority_donate (void);
void priority_max (void);
void priority_remove ( struct lock *lock );
void priority_update (void);
bool priority_cmp (const struct list_elem *cmp1, const struct list_elem *cmp2, void *aux UNUSED);
#endif /* threads/thread.h */
```

thread.c에서 사용할 함수의 원형들과 구조체의 선언이 있다. 구조체의 내부에 과제 1을 위한 종료시간을 저장하는 int64_t를 추가하였고 과제 2를 위한 orig_priority, donation_list, donation_elem, lock_waiting을 추가하였다. 우선 orig_priority는 donation이 일어날 때 기존 우선순위를 저장하기 위해 사용한다. donation_list는 해당 thread에 우선순위를 넘겨주는 thread들을 저장하는 리스트이다. donation_list는 해당 리스트를 관리하기 위한 element이다. lock_waiting은 해당 thread가 얻기 위해 기다리고 있는 lock을 저장한다.

이 외에도 두 과제에 사용되는 함수들의 원형을 미리 선언해두어 호출에 문제가 없게 하고 필요에 따라 다른 .c에서 사용할 수 있게 한다.

○ thread.c

```
#include "threads/thread.h"
#include <debug.h>
#include <stddef.h>
#include <random.h>
#include <stdio.h>
#include <string.h>
#include "threads/flags.h"
#include "threads/interrupt.h"
#include "threads/intr-stubs.h"
#include "threads/palloc.h"
#include "threads/switch.h"
#include "threads/synch.h"
#include "threads/vaddr.h"
#include "devices/timer.h" /////////////////////////////////////////////////////////////////// assignment 1 : for timer_ticks()
#ifdef USERPROG
#include "userprog/process.h"
#endif
/* Random value for struct thread's `magic' member.
```

```
   Used to detect stack overflow.  See the big comment at the top
   of thread.h for details. */
#define THREAD_MAGIC 0xcd6abf4b
/* List of processes in THREAD_READY state, that is, processes
   that are ready to run but not actually running. */
static struct list ready_list;
static struct list timeout_list; ////////////////////////////////////////////////////////// assignment 1 : list for sleep, 'timeouted'
#define depth_lim 8 //////////////////////////////////////////////////////////////////////// assignment 2 : max nesting depth
/* List of all processes.  Processes are added to this list
   when they are first scheduled and removed when they exit. */
static struct list all_list;
/* Idle thread. */
static struct thread *idle_thread;
/* Initial thread, the thread running init.c:main(). */
static struct thread *initial_thread;
/* Lock used by allocate_tid(). */
static struct lock tid_lock;
/* Stack frame for kernel_thread(). */
struct kernel_thread_frame
  {
    void *eip;                  /* Return address. */
    thread_func *function;      /* Function to call. */
    void *aux;                  /* Auxiliary data for function. */
  };
/* Statistics. */
static long long idle_ticks;    /* # of timer ticks spent idle. */
static long long kernel_ticks;  /* # of timer ticks in kernel threads. */
static long long user_ticks;    /* # of timer ticks in user programs. */
/* Scheduling. */
#define TIME_SLICE 4            /* # of timer ticks to give each thread. */
static unsigned thread_ticks;   /* # of timer ticks since last yield. */
/* If false (default), use round-robin scheduler.
   If true, use multi-level feedback queue scheduler.
   Controlled by kernel command-line option "-o mlfqs". */
bool thread_mlfqs;
static void kernel_thread (thread_func *, void *aux);
static void idle (void *aux UNUSED);
static struct thread *running_thread (void);
static struct thread *next_thread_to_run (void);
static void init_thread (struct thread *, const char *name, int priority);
static bool is_thread (struct thread *) UNUSED;
```

```
static void *alloc_frame (struct thread *, size_t size);
static void schedule (void);
void thread_schedule_tail (struct thread *prev);
static tid_t allocate_tid (void);
/* Initializes the threading system by transforming the code
   that's currently running into a thread.  This can't work in
   general and it is possible in this case only because loader.S
   was careful to put the bottom of the stack at a page boundary.
   Also initializes the run queue and the tid lock.
   After calling this function, be sure to initialize the page
   allocator before trying to create any threads with
   thread_create().
   It is not safe to call thread_current() until this function
   finishes. */
void
thread_init (void)
{
  ASSERT (intr_get_level () == INTR_OFF);
  lock_init (&tid_lock);
  list_init (&ready_list);
  list_init (&all_list);
  list_init                                                    (&timeout_list);
////////////////////////////////////////////////////////////// assignment 1
: initializing added list 'timeout'
  /* 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 ();
}
/* Starts preemptive thread scheduling by enabling interrupts.
   Also creates the idle thread. */
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);
}
/* Called by the timer interrupt handler at each timer tick.
   Thus, this function runs in an external interrupt context. */
```

```
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++;
  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE)
    intr_yield_on_return ();
  thread_wakeup(); /////////////////////////////////////////////////////////////////////////// assignment 1 : wakes up thread that needs to be woken */
}
/////////////////////////////////////////////////////////////////////////////////////////////////// added for assignment 1
void thread_sleep (int64_t ticks) { ////////////////////////////////////////////////////////////////// make it sleep for given time
  enum intr_level old_level;
  if (ticks <=0) /////////////////////////////////////////////////////////////////////////// So Obvious
    return;
  // else than run these
  old_level = intr_disable();
  struct thread *timeouted = thread_current();
  timeouted->timeout = ticks; ////////////////////////////////////////////////////////////////////////// write when time to get up
  list_insert_ordered( &timeout_list, &timeouted->elem, &timeout_cmp, NULL); /////////////////////// insert in order for sake of efficiency so that while wakeup you only check the fisrt element
  thread_block(); /////////////////////////////////////////////////////////////////////////// block it.
  intr_set_level(old_level);
}
/////////////////////////////////////////////////////////////////////////////////////////////////// added for assignment 1
```

```
void thread_wakeup (void) { ////////////////////////////////////////////////////////////////// make it wake up at pre-written time
  struct list_elem *chk;
  struct thread *thd;
  enum intr_level old_level;
  if ( list_empty (&timeout_list) ) ////////////////////////////////////////////// Obvious, end if list is empty
    return;
  while ( !list_empty ( &timeout_list ) ) { ////////////////////////////////////////////// while everything is clear, or breaked by if
    chk = list_begin ( &timeout_list ); ////////////////////////////////////////////// list_pop_front at later on makes it move; no need for list_next
    thd = list_entry ( chk, struct thread, elem ); ////////////////////////////////////////////// thread for chk
    if ( thd->timeout > timer_ticks() ) ////////////////////////////////////////////// if fastest timer off time is yet to come
      b r e a k ; ////////////////////////////////////////////// just end the while loop
    // else than runs these
    old_level = intr_disable ();

///////////////////////////////////////////////////////////////////////////////////////// this makes the fastet timeout thread pop out from list

///////////////////////////////////////////////////////////////////////////////////////// also makes e = list_begin ( &timeout_list ) to the next level, changing if-break condition
    list_pop_front ( &timeout_list );
    thread_unblock ( thd ); ////////////////////////////////////////////////////////////////// then unblock that thread poped out of timeout list
    intr_set_level (old_level);
  }
}
// compare logic for passing into list_insert_ordered
// why that third "void *aux UNUSED?"
// because 'list_insert_ordered' func uses that third parameter
// can be found at Definition at line 454 of file list.c
bool timeout_cmp (const struct list_elem *cmp1, const struct list_elem *cmp2, void *aux UNUSED) {
  // for unusall input
  ASSERT (cmp1 !=NULL);
```

```
  ASSERT (cmp2 !=NULL);

  const struct thread *cmp11 = list_entry (cmp1, struct thread, elem);
  const struct thread *cmp22 = list_entry (cmp2, struct thread, elem);
  return cmp11->timeout < cmp22->timeout;
}
/* Prints thread statistics. */
void
thread_print_stats (void)
{
  printf ("Thread: %lld idle ticks, %lld kernel ticks, %lld user ticks₩n",
          idle_ticks, kernel_ticks, user_ticks);
}
/* Creates a new kernel thread named NAME with the given initial
   PRIORITY, which executes FUNCTION passing AUX as the argument,
   and adds it to the ready queue.  Returns the thread identifier
   for the new thread, or TID_ERROR if creation fails.
   If thread_start() has been called, then the new thread may be
   scheduled before thread_create() returns.  It could even exit
   before thread_create() returns.  Contrariwise, the original
   thread may run for any amount of time before the new thread is
   scheduled.  Use a semaphore or some other form of
   synchronization if you need to ensure ordering.
   The code provided sets the new thread's `priority' member to
   PRIORITY, but no actual priority scheduling is implemented.
   Priority scheduling is the goal of Problem 1-3. */
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;
  enum intr_level old_level;
  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 ();
  /* Prepare thread for first run by initializing its stack.
     Do this atomically so intermediate values for the 'stack'
```

```
      member cannot be observed. */
  old_level = intr_disable ();
  /* 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;
  intr_set_level (old_level);
  /* Add to run queue. */
  thread_unblock (t);
  priority_max();
/////////////////////////////////////////////////////////////////// assignment 2
: at creation of thread, it is added to ready_list, so check if that is the
highest priority of that time
  return tid;
}
/* Puts the current thread to sleep.  It will not be scheduled
   again until awoken by thread_unblock().
   This function must be called with interrupts turned off.  It
   is usually a better idea to use one of the synchronization
   primitives in synch.h. */
void
thread_block (void)
{
  ASSERT (!intr_context ());
  ASSERT (intr_get_level () == INTR_OFF);
  thread_current ()->status = THREAD_BLOCKED;
  schedule ();
}
/* Transitions a blocked thread T to the ready-to-run state.
   This is an error if T is not blocked.  (Use thread_yield() to
   make the running thread ready.)
   This function does not preempt the running thread.  This can
   be important: if the caller had disabled interrupts itself,
   it may expect that it can atomically unblock a thread and
   update other data. */
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_cmp, NULL );
////////////////////// assignment 2 : for keeping it high priority at front
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
/* Returns the name of the running thread. */
const char *
thread_name (void)
{
  return thread_current ()->name;
}
/* Returns the running thread.
   This is running_thread() plus a couple of sanity checks.
   See the big comment at the top of thread.h for details. */
struct thread *
thread_current (void)
{
  struct thread *t = running_thread ();

  /* Make sure T is really a thread.
     If either of these assertions fire, then your thread may
     have overflowed its stack.  Each thread has less than 4 kB
     of stack, so a few big automatic arrays or moderate
     recursion can cause stack overflow. */
  ASSERT (is_thread (t));
  ASSERT (t->status == THREAD_RUNNING);
  return t;
}
/* Returns the running thread's tid. */
tid_t
thread_tid (void)
{
  return thread_current ()->tid;
}
/* Deschedules the current thread and destroys it.  Never
   returns to the caller. */
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 ();
}
/* Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
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, &priority_cmp, NULL );
 ////////////////////// assignment 2 : for keeping it high priority at front
    //list_push_back (&ready_list, &cur->elem);
  }
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
/* Invoke function 'func' on all threads, passing along 'aux'.
   This function must be called with interrupts off. */
void
thread_foreach (thread_action_func *func, void *aux)
{
  struct list_elem *e;
  ASSERT (intr_get_level () == INTR_OFF);
  for (e = list_begin (&all_list); e != list_end (&all_list);
       e = list_next (e))
    {
      struct thread *t = list_entry (e, struct thread, allelem);
      func (t, aux);
    }
}
/* Returns the current thread's priority. */
int thread_get_priority (void) {
```

```
  return thread_current ()->priority;
}
void thread_set_priority (int new_priority) { /////////////////////////////////////////////////// assignment 2 : Sets the current thread's priority to NEW_PRIORITY.
  thread_current()->orig_priority = new_priority; ///////////////////////////////////////////// new priority applied
  priority_update(); ////////////////////////////////////////////////////////////////////// priority changed, so update
  priority_max(); //////////////////////////////////////////////////////////////////////////// priority changed, validate that update
}
void priority_max (void) { //////////////////////////////////////////////////////////////////// assignment 2 : check if current thread priority IS THE HIGHEST
  if ( list_empty( &ready_list ) ) { ////////////////////////////////////////////////////////////// Obvious
    return;
  }
  int cur_pri = thread_current()->priority; //////////////////////////////////////////////////// current running priority
  struct thread *thd = list_entry( list_front(&ready_list), struct thread, elem ); ////////////// the highest amog ready_list <- this is possible scince list_insert_ordered
  if ( cur_pri < thd->priority ) { ///////////////////////////////////////////////////////////////// compare which one is higher
    thread_yield(); ////////////////////////////////////////////////////////////////////////// if highest of running list higher than current running, then yield current thread
  }
}
void priority_donate (void) { ///////////////////////////////////////////////////////////////// assignment 2 : the donation of priority
  int depth =0; ////////////////////////////////////////////////////////////////////////// Max time of nesting
  struct thread *cur = thread_current();
  while ( depth <= depth_lim ) { ///////////////////////////////////////////////////////////////// until max
    if ( cur->lock_waiting ==NULL ) { //////////////////////////////////////////////////////////// if there's nothing on
```

```
currnt running thread, then end the donation
         break;
      }
////////////////////////////////////////////////////////////////////////////////
//// BUT else if there's sth on lock waiting list, then
      struct thread   *hold = cur->lock_waiting->holder;
      hold->priority                              =                              cur->priority;
//////////////////////////////////////////////////////////// pass the priority
      cur                                         =                                       hold;
////////////////////////////////////////////////////////////////////////////////
and move on to the next nest
      depth++;
   }
}
void                       priority_update                        (void)                    {
//////////////////////////////////////////////////////////////// update priority to
the highest available
         struct           thread           *cur           =            thread_current();
/////////////////////////////////////////////////////// for current thread
         cur->priority                         =                        cur->orig_priority;
/////////////////////////////////////////////////////////    change    priority    to
original
         if     (     list_empty(     &cur->donation_list     )     ==false     )     {
///////////////////////////////////////////  if donated
      list_sort(       &cur->donation_list,       &priority_cmp,       NULL       );
//////////////////////////////////////  make  sure  donation  list  is  sorted  to
make highet is at beginning
      struct  thread  *high  =  list_entry(  list_front(  &cur->donation_list  ),
struct thread, donation_elem); // for highest peiority from donation list
      if         (         high->priority         >         cur->priority         )         {
/////////////////////////////////////////////////   compare   if   priority   from
donation might be higher
        cur->priority                             =                          high->priority;
/////////////////////////////////////////////////////// if so, change
      }
   }
}
void       priority_remove       (       struct       lock       *lock       )       {
///////////////////////////////////////////////////  check  &  remove  entry  from
donation_list that holds specific lock given
   struct              thread              *cur              =              thread_current();
/////////////////////////////////////////////////// for currently running thread
   struct      list_elem      *donated      =      list_begin(&cur->donation_list);
///////////////////////////////////  will  check  that  donation_list  from  start  to
end
   struct                              thread                                      *thd;
```

```
///////////////////////////////////////////////////////////////////////////////////        tmp thread
        while   (   donated   !=   list_end(   &cur->donation_list   )   )   {
///////////////////////////////////////// until the end of donation_list, repea
                thd = list_entry( donated, struct thread, donation_elem );
////////////////////////////////// tmp thread is entry from donation_list
                if     (     thd->lock_waiting     ==     lock     )     {
////////////////////////////////////////////////////////// if that entry have lock
                        list_remove(&thd->donation_elem);
////////////////////////////////////////////////////////////        remove        from donation_list
     }
        donated                          =                          list_next(donated);
//////////////////////////////////////////////////////////////// next in the list
        }
}
bool priority_cmp (const struct list_elem *cmp1, const struct list_elem *cmp2, void *aux UNUSED) { // assignment 2 : simple compare logic
  ASSERT (cmp1 !=NULL);
  ASSERT (cmp2 !=NULL);
  const struct thread *cmp11 = list_entry (cmp1, struct thread, elem);
  const struct thread *cmp22 = list_entry (cmp2, struct thread, elem);
  return cmp11->priority > cmp22->priority;
}
/* Sets the current thread's nice value to NICE. */
void
thread_set_nice (int nice UNUSED)
{
  /* Not yet implemented. */
}
/* Returns the current thread's nice value. */
int
thread_get_nice (void)
{
  /* Not yet implemented. */
  return 0;
}
/* Returns 100 times the system load average. */
int
thread_get_load_avg (void)
{
  /* Not yet implemented. */
  return 0;
}
/* Returns 100 times the current thread's recent_cpu value. */
int
```

```
thread_get_recent_cpu (void)
{
  /* Not yet implemented. */
  return 0;
}

/* Idle thread.  Executes when no other thread is ready to run.
   The idle thread is initially put on the ready list by
   thread_start().  It will be scheduled once initially, at which
   point it initializes idle_thread, "up"s the semaphore passed
   to it to enable thread_start() to continue, and immediately
   blocks.  After that, the idle thread never appears in the
   ready list.  It is returned by next_thread_to_run() as a
   special case when the ready list is empty. */
static void
idle (void *idle_started_ UNUSED)
{
  struct semaphore *idle_started = idle_started_;
  idle_thread = thread_current ();
  sema_up (idle_started);
  for (;;)
    {
      /* Let someone else run. */
      intr_disable ();
      thread_block ();
      /* Re-enable interrupts and wait for the next one.
         The `sti' instruction disables interrupts until the
         completion of the next instruction, so these two
         instructions are executed atomically.  This atomicity is
         important; otherwise, an interrupt could be handled
         between re-enabling interrupts and waiting for the next
         one to occur, wasting as much as one clock tick worth of
         time.
         See [IA32-v2a] "HLT", [IA32-v2b] "STI", and [IA32-v3a]
         7.11.1 "HLT Instruction". */
      asm volatile ("sti; hlt" : : : "memory");
    }
}
/* Function used as the basis for a kernel thread. */
static void
kernel_thread (thread_func *function, void *aux)
{
  ASSERT (function !=NULL);
  intr_enable ();         /* The scheduler runs with interrupts off. */
  function (aux);         /* Execute the thread function. */
  thread_exit ();         /* If function() returns, kill the thread. */
```

```
}

/* Returns the running thread. */
struct thread *
running_thread (void)
{
  uint32_t *esp;
  /* Copy the CPU's stack pointer into `esp', and then round that
     down to the start of a page.  Because `struct thread' is
     always at the beginning of a page and the stack pointer is
     somewhere in the middle, this locates the curent thread. */
  asm ("mov %%esp, %0" : "=g" (esp));
  return pg_round_down (esp);
}
/* Returns true if T appears to point to a valid thread. */
static bool
is_thread (struct thread *t)
{
  return t !=NULL && t->magic == THREAD_MAGIC;
}
/* Does basic initialization of T as a blocked thread named
   NAME. */
static void
init_thread (struct thread *t, const char *name, int priority)
{
  ASSERT (t !=NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name !=NULL);
  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;

////////////////////////////////////////////////////////////////////////////////
///// assignment 2 : initializing struct elements
  t->orig_priority = priority;
  list_init(&t->donation_list);
  t->lock_waiting =NULL;
  t->magic = THREAD_MAGIC;
  list_push_back (&all_list, &t->allelem);
}
/* Allocates a SIZE-byte frame at the top of thread T's stack and
   returns a pointer to the frame's base. */
static void *
alloc_frame (struct thread *t, size_t size)
```

```
{
  /* Stack data is always allocated in word-size units. */
  ASSERT (is_thread (t));
  ASSERT (size % sizeof (uint32_t) ==0);
  t->stack -=size;
  return t->stack;
}
/* Chooses and returns the next thread to be scheduled.  Should
   return a thread from the run queue, unless the run queue is
   empty.  (If the running thread can continue running, then it
   will be in the run queue.)  If the run queue is empty, return
   idle_thread. */
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);
}
/* Completes a thread switch by activating the new thread's page
   tables, and, if the previous thread is dying, destroying it.
   At this function's invocation, we just switched from thread
   PREV, the new thread is already running, and interrupts are
   still disabled.  This function is normally invoked by
   thread_schedule() as its final action before returning, but
   the first time a thread is scheduled it is called by
   switch_entry() (see switch.S).
   It's not safe to call printf() until the thread switch is
   complete.  In practice that means that printf()s should be
   added at the end of the function.
   After this function and its caller returns, the thread switch
   is complete. */
void
thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();

  ASSERT (intr_get_level () == INTR_OFF);
  /* Mark us as running. */
  cur->status = THREAD_RUNNING;
  /* Start new time slice. */
  thread_ticks =0;
#ifdef USERPROG
  /* Activate the new address space. */
  process_activate ();
```

```
#endif
  /* If the thread we switched from is dying, destroy its struct
     thread.  This must happen late so that thread_exit() doesn't
     pull out the rug under itself.  (We don't free
     initial_thread because its memory was not obtained via
     palloc().) */
  if (prev !=NULL && prev->status == THREAD_DYING && prev !=
initial_thread)
    {
      ASSERT (prev != cur);
      palloc_free_page (prev);
    }
}
/* Schedules a new process.  At entry, interrupts must be off and
   the running process's state must have been changed from
   running to some other state.  This function finds another
   thread to run and switches to it.
   It's not safe to call printf() until thread_schedule_tail()
   has completed. */
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);
}
/* Returns a tid to use for a new thread. */
static tid_t
allocate_tid (void)
{
  static tid_t next_tid =1;
  tid_t tid;
  lock_acquire (&tid_lock);
  tid = next_tid++;
  lock_release (&tid_lock);
  return tid;
}

/* Offset of `stack' member within `struct thread'.
   Used by switch.S, which can't figure it out on its own. */
```

```
uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

우선 과제 1을 위해 timer.h를 include하고 ready와는 별개로 sleep = timeout을 관리하기 위한 리스트를 선언한다. 이어서 thread_init 안에 해당 timeout 리스트를 list_init 함수로 초기화한다.

강의노트에 의해 주어졌듯이 매 tick마다 호출되는 함수 thread_tick 안에 thread를 호출하도록 하였다. 이는 새로운 함수 thread_wakeup으로 구현하였다. 해당 함수는 위에서 선언한 timeout 리스트를 체크하여 현재 시간과 저장돼있는 호출 예정 시간과 비교하여 조건에 맞는 thread를 unblock한다. 함수 내부적으로 다양한 조건을 체크한다. 우선 timeout된 리스트가 비어 있는 경우 함수를 종료한다. 리스트에 무언가 들어있다면 조건에 의해 종료될 때 까지 다음 동작들을 반복한다. 이때 timeout 리스트의 시작 부분과 해당 thread에 대해 동작하는데 이는 해당 timeout 리스트에 thread를 추가할 때 오름차순으로 추가되도록 해 두었으므로 리스트의 가장 처음에 있는 thread가 가장 빠른 wakeup tick을 가지고 있다. 이에 대해 현재 tick이 가장 빠른 wakeup tick보다 이르다면 while은 종료되고 아무런 행동 없이 호출이 종료된다. 만약 현재 tick이 리스트에 있는 가장 이른 tick과 같거나 넘어갔다면 앞에서부터 list_pop_front -> unblock하여 차례로 리스트를 비운다. 이때 리스트가 비거나 가장 빠른 wakeup 시간보다 현재 tick이 이른 두 가지의 경우 while이 종료된다. 리스트에서 빠져나와 unblock할 때 interrupt는 disable한다.

thread_sleep 함수는 인자로 넘겨받은 wakeup 예정 시간을 thread에 기록하고 timeout으로 보내는 함수이다. 이때 list_insert_ordered를 사용하여 가장 빠른 wakeup을 가진 thread가 가장 앞으로 오게 삽입한다. 이후 block한다. 이 과정에서도 interrupt는 disable한다. 삽입에 사용하는 list_insert_ordered는 비교 결과를 알려주는 함수를 인자로서 넘겨야 한다. 따라서 timeout_cmp함수를 선언하여 두 thread를 넘겨받은 두 인자의 쓰레드로 한당하고 비교연산식을 return하여 앞에 오는 것이 더 작다면 연산식의 결과인 true를 반환하고 뒤에 오는 것이 더 크다면 연산식의 결과인 false를 반환할 것이다. 함수의 세 번째 인자는 unused인데 이는 list_insert_ordered 함수가 세 개의 인자를 사용하는 것으로 구현되어 있으므로 이를 맞추기 위해 사용되는 일종의 더미 인자이다. thread_sleep 뿐만 아니라 이후에도 계속 list_insert_ordered 함수에서 해당 인자는 NULL로 처리한다.

과제 2를 위하여 우선 nesting 깊이를 제한하는 depth_lim을 define하였다.

우선순위 비교를 위해 priority_max 함수를 선언하여 현재 실행 중인 thread와 ready_list에서 대기중인 첫 번째 thread의 우선순위를 비교하고 만약 ready_list에 있는 우선순위가 더 높다면 현재 thread는 yield를 통해 ready_list로 보낸다. 이때 첫 번째 thread를 비교하는 것은 해당 ready_list를 우선순위 우선 정렬로 유지시켜 두었기 때문에 가장 앞에 있는 thread가 리스트의 가장 높은 우선순위이기 때문이다. 이 priority_max 함수는 readpy_list에 변동이 생기거나 thread의 우선순위에 변동이 되는 thread_unblock 뒤에 붙어서 호출된다. 즉 우선순위를 바꾸는 함수 thread_set_priority와 새로운 thread를 만들어 리스트에 추가하는 함수 thread_create, 그리고 후술할 synch.c에 있는 sema_up함수에서 호출된다.

priority_donate함수는 우선순위의 donation이 발생할 때 호출되는 함수로 주어진 nesting 한계까지 현재 실행되고 있는 thread를 시작으로 해당 thread가 기다리고 있는 lock을 가진 thread로 이동하여 우선순위를 넘겨주는 것을 반복한다. 해당 한계치는 상기한 define에 의해 가로막힌다. 이 함수 또한 후술한 synch.c의 lock_acquire 함수에 의해 호출된다.

priority_update 함수는 현재 실행되고 있는 thread에 대해 donation_list에 저장된 우선순위 donation을 준 thread들 중 가장 높은 우선순위로 맞춰주는 기능을 한다. 이때 list_sort를 사용하여 가장 높은 우선 순위가 리스트의 가장 앞으로 오게 하여 사용한다.

구현한 리스트 중 donation_list 같은 경우 리스트에 삽입하거나 값을 볼 때 내림차순으로 굳이 정렬하여 자료구조를 유지하지 않고 값을 읽어올 때만 list_max를 사용해서 더 간단하게 구현할 수 있다. 다만 이렇게 하지 않은 것은 다른 리스트 자료구조들이 항상 오름차순 혹은 내림차순 자료구조를 유지하는 것으로 논리적인 이득을 취하고 있으므로, 굳이 두 가지 형태의 리스트 자료구조를 생각하는 것 보다 하나의 통일성 있는 자료구조 형태로 유지하는 것이 더 논리적으로 자연스럽기 때문이다.

이어서 priority_remove 함수는 현재 thread의 donation_list에 대해 인자로 전달받은 lock을 가지고 있는 thread를 제거한다. 이를 list_next를 통해 계속 나아가며 donation_list의 마지막까지 반복한다.

마지막 추가 함수 priority_cmp는 입력받은 두 인자에 대해 thread의 우선순위를 비교하여 앞이 더 크면 true, 뒤가 더 크면 false를 반환한다.

이후 init_thread에서 struct 구조체의 변수들을 초기화한다.

○ synch.c

```
/* This file is derived from source code for the Nachos
   instructional operating system.  The Nachos copyright notice
   is reproduced in full below. */
/* Copyright (c) 1992-1996 The Regents of the University of California.
   All rights reserved.
   Permission to use, copy, modify, and distribute this software
   and its documentation for any purpose, without fee, and
   without written agreement is hereby granted, provided that the
   above copyright notice and the following two paragraphs appear
   in all copies of this software.
   IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO
   ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR
   CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS
SOFTWARE
   AND ITS DOCUMENTATION, EVEN IF THE UNIVERSITY OF
CALIFORNIA
   HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
   THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
   WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
   WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
PARTICULAR
   PURPOSE.  THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS
IS"
   BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION
TO
   PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
   MODIFICATIONS.
*/
#include "threads/synch.h"
#include <stdio.h>
#include <string.h>
#include "threads/interrupt.h"
#include "threads/thread.h"
bool priority_cmp_sema (const struct list_elem *a, const struct list_elem
*b, void *aux UNUSED); /////////////// assignment 2 : func that used
only here
/* Initializes semaphore SEMA to VALUE.  A semaphore is a
   nonnegative integer along with two atomic operators for
   manipulating it:
   - down or "P": wait for the value to become positive, then
     decrement it.
   - up or "V": increment the value (and wake up one waiting
     thread, if any). */
```

```c
void
sema_init (struct semaphore *sema, unsigned value)
{
  ASSERT (sema !=NULL);
  sema->value = value;
  list_init (&sema->waiters);
}
/* Down or "P" operation on a semaphore.  Waits for SEMA's value
   to become positive and then atomically decrements it.
   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but if it sleeps then the next scheduled
   thread will probably turn interrupts back on. */
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) { /* so if semaphore is not avilable, than BLOCK & wait */
    // list_push_back (&sema->waiters, &thread_current ()->elem);
    list_insert_ordered(&sema->waiters, &thread_current()->elem, &priority_cmp, NULL); ////////////////////////////// assignment 2 : for keeping high pri at front
    thread_block ();
  }
  sema->value--; /* now semaphore is avilable, so get it */
  intr_set_level (old_level);
}
/* Down or "P" operation on a semaphore, but only if the
   semaphore is not already 0.  Returns true if the semaphore is
   decremented, false otherwise.
   This function may be called from an interrupt handler. */
bool
sema_try_down (struct semaphore *sema)
{
  enum intr_level old_level;
  bool success;
  ASSERT (sema !=NULL);
  old_level = intr_disable ();
  if (sema->value >0)
    {
      sema->value--;
```

```
        success =true;
      }
    else
      success =false;
    intr_set_level (old_level);
    return success;
}
/* Up or "V" operation on a semaphore.  Increments SEMA's value
   and wakes up one thread of those waiting for SEMA, if any.
   This function may be called from an interrupt handler. */
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_cmp,               NULL); /////////////////////////////////////////////////// assignment 2 : for keeping high pri at front
    thread_unblock (list_entry (list_pop_front (&sema->waiters), struct thread, elem));
  }
  sema->value++;                                    /* semaphore now occupied */
  priority_max();  //////////////////////////////////////////////////////////////////////////////// assignment 2 : unblock happened -> ready_list changed -> check priority
  intr_set_level (old_level);
}
static void sema_test_helper (void *sema_);
/* Self-test for semaphores that makes control "ping-pong"
   between a pair of threads.  Insert calls to printf() to see
   what's going on. */
void
sema_self_test (void)
{
  struct semaphore sema[2];
  int i;
  printf ("Testing semaphores...");
  sema_init (&sema[0], 0);
  sema_init (&sema[1], 0);
  thread_create ("sema-test", PRI_DEFAULT, sema_test_helper, &sema);
  for (i =0; i <10; i++)
    {
      sema_up (&sema[0]);
      sema_down (&sema[1]);
```

```
    }
  printf ("done.₩n");
}
/* Thread function used by sema_self_test(). */
static void
sema_test_helper (void *sema_)
{
  struct semaphore *sema = sema_;
  int i;
  for (i =0; i <10; i++)
    {
      sema_down (&sema[0]);
      sema_up (&sema[1]);
    }
}

/* Initializes LOCK.  A lock can be held by at most a single
   thread at any given time.  Our locks are not "recursive", that
   is, it is an error for the thread currently holding a lock to
   try to acquire that lock.
   A lock is a specialization of a semaphore with an initial
   value of 1.  The difference between a lock and such a
   semaphore is twofold.  First, a semaphore can have a value
   greater than 1, but a lock can only be owned by a single
   thread at a time.  Second, a semaphore does not have an owner,
   meaning that one thread can "down" the semaphore and then
   another one "up" it, but with a lock the same thread must both
   acquire and release it.  When these restrictions prove
   onerous, it's a good sign that a semaphore should be used,
   instead of a lock. */
void
lock_init (struct lock *lock)
{
  ASSERT (lock !=NULL);
  lock->holder =NULL;
  sema_init (&lock->semaphore, 1);
}
/* Acquires LOCK, sleeping until it becomes available if
   necessary.  The lock must not already be held by the current
   thread.
   This function may sleep, so it must not be called within an
   interrupt handler.  This function may be called with
   interrupts disabled, but interrupts will be turned back on if
   we need to sleep. */
void lock_acquire (struct lock *lock) {
  ASSERT (lock !=NULL);
```

```
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));
  struct              thread              *thd            =             thread_current();
//////////////////////////////////////////////////////////////////////   assignmet
2
  if(              lock->holder               !=NULL               )             {
///////////////////////////////////////////////////////////////////////////      if
lock holder thread EXISTS
    thd->lock_waiting                         =                                 lock;
///////////////////////////////////////////////////////////////////////////    put
that lock in currnet thread's waiting list
    list_insert_ordered            (            &lock->holder->donation_list,
&thd->donation_elem, &priority_cmp, NULL ); /////////////// also put it in
LOCK HOLDER's donation elem list, keeping 'the order'
    p  r  i  o  r  i  t  y  _  d  o  n  a  t  e  (  )  ;
///////////////////////////////////////////////////////////////////////////   and
donate priority to that lock holder
  }
  sema_down                                          (&lock->semaphore);
///////////////////////////////////////////////////////////////////////////   call
sema down, anyway. after this means it got semaphore & lock
  thd->lock_waiting                                                      =NULL;
////////////////////////////////////////////////////////////////////////////////
// got that lock, so nothing on lock_waiting list
  lock->holder              =              thread_current              ();
////////////////////////////////////////////////////////////////////////////////
current lock holder is currently running thread
}
/* Tries to acquires LOCK and returns true if successful or false
   on failure.  The lock must not already be held by the current
   thread.
   This function will not sleep, so it may be called within an
   interrupt handler. */
bool
lock_try_acquire (struct lock *lock)
{
  bool success;
  ASSERT (lock !=NULL);
  ASSERT (!lock_held_by_current_thread (lock));
  success = sema_try_down (&lock->semaphore);
  if (success)
    lock->holder = thread_current ();
  return success;
}
/* Releases LOCK, which must be owned by the current thread.
   An interrupt handler cannot acquire a lock, so it does not
```

```
   make sense to try to release a lock within an interrupt
   handler. */
void lock_release (struct lock *lock) {
  ASSERT (lock !=NULL);
  ASSERT (lock_held_by_current_thread (lock));
  enum intr_level old_level = intr_disable ();
  lock->holder =NULL;     /* letting go of lock */
  priority_remove(lock);
/////////////////////////////////////////////////////////////////////////////////
///////// assignment 2 : remove of that lock
  priority_update();
/////////////////////////////////////////////////////////////////////////////////
///////// assignment 2 : that might change priority of some threads
  sema_up (&lock->semaphore);
  intr_set_level (old_level);
}
/* Returns true if the current thread holds LOCK, false
   otherwise.  (Note that testing whether some other thread holds
   a lock would be racy.) */
bool
lock_held_by_current_thread (const struct lock *lock)
{
  ASSERT (lock !=NULL);
  return lock->holder == thread_current ();
}

/* One semaphore in a list. */
struct semaphore_elem
  {
    struct list_elem elem;              /* List element. */
    struct semaphore semaphore;         /* This semaphore. */
  };
/* Initializes condition variable COND.  A condition variable
   allows one piece of code to signal a condition and cooperating
   code to receive the signal and act upon it. */
void
cond_init (struct condition *cond)
{
  ASSERT (cond !=NULL);
  list_init (&cond->waiters);
}
/* Atomically releases LOCK and waits for COND to be signaled by
   some other piece of code.  After COND is signaled, LOCK is
   reacquired before returning.  LOCK must be held before calling
   this function.
   The monitor implemented by this function is "Mesa" style, not
```

```
    "Hoare" style, that is, sending and receiving a signal are not
    an atomic operation.  Thus, typically the caller must recheck
    the condition after the wait completes and, if necessary, wait
    again.
    A given condition variable is associated with only a single
    lock, but one lock may be associated with any number of
    condition variables.  That is, there is a one-to-many mapping
    from locks to condition variables.
    This function may sleep, so it must not be called within an
    interrupt handler.  This function may be called with
    interrupts disabled, but interrupts will be turned back on if
    we need to sleep. */
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);
  list_insert_ordered(&cond->waiters,  &waiter.elem,  &priority_cmp_sema,
NULL); ////////////////////////////////////////// assignment 2 : keep the
order inthe list
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
/* If any threads are waiting on COND (protected by LOCK), then
    this function signals one of them to wake up from its wait.
    LOCK must be held before calling this function.
    An interrupt handler cannot acquire a lock, so it does not
    make sense to try to signal a condition variable within an
    interrupt handler. */
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))
{///////////////////////////////////////////////////////////////////////////////////
////// assignment 2 : until the end of waiting list
    list_sort(    &cond->waiters,    &priority_cmp_sema,     NULL    );
////////////////////////////////////////////////////////////////////// same, making
```

```
sure of the order
     sema_up    (&list_entry    (list_pop_front    (&cond->waiters),    struct
semaphore_elem, elem)->semaphore);
  }
}
/* Wakes up all threads, if any, waiting on COND (protected by
   LOCK).  LOCK must be held before calling this function.
   An interrupt handler cannot acquire a lock, so it does not
   make sense to try to signal a condition variable within an
   interrupt handler. */
void
cond_broadcast (struct condition *cond, struct lock *lock)
{
  ASSERT (cond !=NULL);
  ASSERT (lock !=NULL);
  while (!list_empty (&cond->waiters))
    cond_signal (cond, lock);
}
bool  priority_cmp_sema  (const  struct  list_elem  *cmp1,  const  struct
list_elem *cmp2, void *aux UNUSED) { ////////////////////// assignment 2
: semapore compare logic
        struct    semaphore_elem    *cmp1_s    =    list_entry(cmp1,    struct
semaphore_elem, elem);
        struct    semaphore_elem    *cmp2_s    =    list_entry(cmp2,    struct
semaphore_elem, elem);
        struct              list_elem              *cmp1_se              =
list_begin(&(cmp1_s->semaphore.waiters));
        struct              list_elem              *cmp2_se              =
list_begin(&(cmp2_s->semaphore.waiters));
        struct thread *cmp1_set = list_entry(cmp1_se, struct thread, elem);
        struct thread *cmp2_set = list_entry(cmp2_se, struct thread, elem);
        return (cmp1_set->priority) > (cmp2_set->priority);
}
```

우선 semaphore를 비교할 함수의 원형을 선언한다. 이후 cond_wait 함수에서 list _insert_ordered 함수에서 비교함수로서 사용한다. 이는 최하단에 구현되어 있다. 인자로 list_elem을 받아서 semaphore_elem을 선언, 이에 해당하는 waiter의 첫 번째 list_elem을 선언, 마지막으로 이에 해당하는 thread의 우선순위를 비교하여 앞피 더 크면 true, 뒤가 더 크면 false를 반환한다. 이 함수는 바깥에서 사용되지 않으므로 헤더에 추가하지 않았다.

sema_down 함수를 수정하여 sema->waiter에서 대기자 리스트에 추가하는 것을 리스트의 뒤에 추가하는 것이 아닌 list_insert_ordered를 사용해서 우선순위가 더 큰 것이 앞으로 오도록 한다.

sema_up 함수를 수정하여 semaphor waiters 중 가장 높은 우선순위를 가지는 thread를 unblock한다. 이때 우선순위가 변동되었을 수 있으니 list_sort로 정렬하여 확인한다. 또한 상기하였듯이 ready_list에 변동사항이 생기므로 priority_max 함수를 호출하여 가장 높은 우선순위를 가진 thread가 실행되도록 한다.

lock_acquire 함수를 수정하여 lock을 현재 thread에게 lock을 준다. 이때 lock의 holder가 없다면 바로 sema_down을 호출하여 넘어갔다 돌아오고 lock을 얻는다. holder가 이미 있다면 현재 thread의 lock_waiting 리스트에 lock을 추가하고 현재 lock의 holder의 donation_list에 현재 thread를 추가한다. 이후 sema_down하는 것은 동일하다. 상황 종료되고 현재 thread의 lock_waiting은 없어지고 lock holder는 현재 thread가 된다.

lock_release에서 lock을 놓아주고 해당 lock이 사라졌으므로 priority_remove로 관련된 우선순위를 빼준다. 이에 따라 우선순위가 변동되었으므로 상기한 priority_update로 우선순위를 새롭게 업데이트한다.

cond_wait 함수를 수정하여 cond->waiters 리스트에 후순으로 추가하던 것을 우선순위에 맞게 추가하는 list_insert_ordered로 변경한다. 이후 sema_down으로 thread를 처리한다. 마찬가지로 cond_signal도 cond->waiters 리스트에서 하나를 뺄 때 list_sort로 정렬한 후 가장 높은 우선순위를 가장 앞에서 빼서 sema_up한다.

□ 실행 결과

○ 주어진 alarm, priority 테스트들에 대해 모두 통과한 것을 확인할 수 있다.