---- GROUP ----
Zachery Brunner <zacherybrunner@k-state.edu>
Jordan Voss <javoss@k-state.edu>
Jackson Carder <jgcarder@k-state.edu>

---- PRELIMINARIES ----
>> If you have any preliminary comments on your submission, notes for the TA, or extra credit, please give them here.

    Finished Part #1
    Attempted Part #2
    Did not attempt EC

>> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation course text, lecture notes, and course staff.

>Additions or modifications to files will be marked by comments and by bordering sets of lines

```
>i.e. //------------------------------
>        /*code*/
>    //-------------------------------
```

--References--
>https://github.com/Waqee/Pintos-Project-1/blob/master/src/threads/thread.c

    >>For use with the bool thread_cmp_wakeup (struct list_elem *first_elem, struct list_elem *second_elem) method.

>https://github.com/meskuwa/pintos-project-1/tree/master/src/threads

    >>Logic assistance

>https://github.com/SignorMercurio/PintOS-Project-1

    >>Logic assistance

ALARM CLOCK
===========

---- DATA STRUCTURES ----
>> A1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

pintos\src\devices\timer.c

Global list that holds all of the threads that are currently asleep (***Line 25***)

```
static struct list sleeping_thread_list;
```

Created sleepingThreads struct which holds information about waking threads up (***Lines 28-31***)

```
struct sleepingThreads {
  int64_t ticksToSleep;           /* This is the param that was passed into the
sleep function */
  struct thread * thread;         /* Holds the information about the thread that
wanted to sleep */
  struct list_elem elem;          /* Needed to be able to make a list of these
structs */
};
```

timer_init now also initializes the global variable ticks and the global sleeping list (***Lines 52-58***)

```
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
  ticks = 0;
  list_init(&sleeping_thread_list);
}
```

---- ALGORITHMS ----
>> A2: Briefly describe what happens in a call to your timer_sleep(),
>> including the effects of the timer interrupt handler.
>> A3: What steps are taken to minimize the amount of time spent in
>> the timer interrupt handler?

(***Lines 109-129***)

```
//--------------------------------------------------------------------------------
-----------------------
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
   be turned on. */
void
timer_sleep (int64_t sleepForThisLong)
{
  ASSERT (intr_get_level () == INTR_ON);

  if(sleepForThisLong > 0)
  {
    struct sleepingThreads sleepy_thread;
    struct thread* t = thread_current();

    /* Grabs the thread's information so we can wake it up later */
    sleepy_thread.thread = t;
    sleepy_thread.ticksToSleep = sleepForThisLong;

    /* Disables interrupts, inserts into list, enables interrupts */
    enum intr_level old = intr_disable();
    list_push_front(&sleeping_thread_list, &sleepy_thread.elem);
    thread_block();
    intr_set_level(old);
  }
}
//--------------------------------------------------------------------------------
-----------------------
```

>A2: timer_sleep first ensures that the thread actually wants to sleep for a period of time. If the value passed in is not greater than zero the method returns. Second step is to create a sleepingThreads struct that will facilitate the information transfer between methods. Finally, interrupts must be disabled so the sleeping_thread_list may be safely modified by adding the new sleeping thread. Interrupts are re-enabled and execution of threads continue.

(***Lines 204-227***)
```
//--------------------------------------------------------------------------------
-----------------------
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
    /* Ticks is being incremented automatically */
    ticks++;

    /* After execution of this line current = first element of list */
    struct list_elem *traversal;
    struct sleepingThreads *thread;

    /* Iterates through the list until the end is reached */
    for(traversal = list_rbegin(&sleeping_thread_list);
!list_empty(&sleeping_thread_list) && traversal !=
list_rend(&sleeping_thread_list); traversal = list_prev(traversal))
    {
      thread = list_entry(traversal, struct sleepingThreads, elem);
      thread->ticksToSleep--;
      if(thread->ticksToSleep <= 0)
      {
        list_remove(traversal);
        struct thread* t = thread->thread;
        thread_unblock(t);
      }
    }
  thread_tick ();
}
//--------------------------------------------------------------------------------
-----------------------
```
>A3: The interrupt function features a for loop that traverses the sleeping_thread_list from the end to the beginning. Threads are not ordered in any way as they are just inserted into the list as they come. The loop decreases the waiting time by 1 tick every iteration while finally removing threads that have a waiting time of 0 or less and 0 remaining.Once threads are removed from the list, they are unblocked and put back onto the ready queue and can possibly be scheduled the next time the scheduler runs.

---- SYNCHRONIZATION ----
>> A4: How are race conditions avoided when multiple threads call
>> timer_sleep() simultaneously?
>> A5: How are race conditions avoided when a timer interrupt occurs
>> during a call to timer_sleep()?

>A4: By disabling interrupts, threads that call timer_sleep simultaneously is not an issue. After entering the method, only thread specific information is collected. Before the critical section of the method we disable interrupts so the thread can work with the shared memory in peace without fear another thread will intervene.

>A5: As stated above, timer interrupts are disabled only during the critical section where the thread is accessing shared memory. Timer interrupts during timer_sleep are okay.

---- RATIONALE ----
>> A6: Why did you choose this design? In what ways is it superior to
>> other designs that you considered?

>A6:
We spent a lot of time in the list and pintos manual pages trying to make a more efficient alarm clock. We tried to implement an ordered list with the wakeup being equal to current timer ticks + ticks to sleep. This would have allowed us to traverse the list only as far as we need to wake up all the threads that need to wake up. In trying to accomplish this method we were running into a variety of exceptions and ended up taking this functionality out due to us not being able to get it to work successfully.

We went with this solution to the alarm clock problem partially due to time constraint. Our code is superior to other designs because of code readability and followable logic.

PRIORITY SCHEDULING
==================
---- DATA STRUCTURES ----
>> B1: Copy here the declaration of each new or changed `struct' or `struct' member, global or static variable, `typedef', or enumeration. Identify the purpose of each in 25 words or less.
>> B2: Explain the data structure used to track priority donation.

pintos/src/threads/thread.h
(***Line 83-111***)

```
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. */

    /* Variables for priority donation */
    int orig_priority;                  /* The original priority of the thread */
    struct list donor_list;             /* List of threads that have donated
priority */
    struct list_elem donor_elem;        /* Element that will be in the donor_list
*/
    struct lock* lock_waiting_for;       /* Lock that the thread is waiting to
acquire */



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

```
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```
>B2: We added a variable called orig_priority which is a placeholder value that holds the original priority given to the thread when it is initialized or when the priority is changed. To handle priority donations we used the variable priority to be considered the active priority level. When the thread gave up the lock its priority was reverted back to the original priority.

---- ALGORITHMS ----
>> B3: How do you ensure that the highest priority thread waiting for
>> a lock, semaphore, or condition variable wakes up first?
>> B4: Describe the sequence of events when a call to lock_acquire()
>> causes a priority donation. How is nested donation handled?
>> B5: Describe the sequence of events when lock_release() is called
>> on a lock that a higher-priority thread is waiting for.

pintos/src/threads/threads.c *Lines: 303-322*
```
//--------------------------------------------------------------------------------
-----------------------
/* 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, thread_cmp_priority, NULL);
    //list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
//--------------------------------------------------------------------------------
-----------------------
```

>B3: The ready_list is a list of all the processes that are ready to run. They are all waiting on the CPU to give them time. Each time that a thread is added to the ready_list we use the function call *list_insert_ordered(&ready_list, &cur->elem, thread_cmp_priority, NULL);* it will insert the current thread struct into the correct position in the ready_list, such that they are ordered with the highest priority first.

pintos/src/threads/threads.c *Lines:343-363*

```
//----------------------------------------------------------------------------------
-----------------------
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  enum intr_level old_level = intr_disable();
  struct thread* cur_thread = thread_current ();
  struct thread* traversal;
  /* If the priority donation list for the current thread is empty
  * or
  * the new_priority is greater than the current priority then set the current
thread priority equal to the new one
  */
  cur_thread->orig_priority = new_priority;
  if (list_empty (&cur_thread->donor_list) || new_priority >
cur_thread->priority)
  {
    cur_thread->priority = new_priority;
    thread_yield();
  }


  intr_set_level(old_level);
}
//----------------------------------------------------------------------------------
-----------------------
```

pintos/src/threads/thread.c *Lines:475-496*

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

  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);

  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;      /* Sets the new thread’s priority to the passed
                                  in priority */
  t->orig_priority = priority; /* Sets the thread’s original priority to the
                                  passed in priority */
  list_init(&t->donor_list);   /* Initializes the list of donors for the new
                                  thread */
  //---------------------------

  t->magic = THREAD_MAGIC;

  old_level = intr_disable ();
  list_push_back (&all_list, &t->allelem);
  intr_set_level (old_level);
}
```

pintos/src/threads/thread.c *Lines:613-623*

```
//---------------------------------------------------------------------------------
------------------------
/* Compare the priorities of two threads and return which one is larger */
bool thread_cmp_priority (struct list_elem *first_elem, struct list_elem
*second_elem)
```

```
{
  struct thread *first = list_entry(first_elem, struct thread, elem);
  struct thread *second = list_entry(second_elem, struct thread, elem);

  //Return if first is greater than second
  return first->priority > second->priority;
}
//--------------------------------------------------------------------------------
------------------------
```

pintos/src/threads/synch.c *Lines:205-232*

```
//--------------------------------------------------------------------------------
------------------------
/* 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));

  enum intr_level old_level = intr_disable();


  struct thread* cur_thread = thread_current();
  if (lock->holder == NULL)
    lock->holder = cur_thread;
  else
  {
    cur_thread->lock_waiting_for = lock;
    struct thread *lock_holding_thread = lock->holder;
    if(cur_thread->priority > lock_holding_thread->priority)
    {
      lock_holding_thread->priority = cur_thread->priority;
      list_insert_ordered(&lock_holding_thread->donor_list,
&cur_thread->donor_elem, thread_cmp_priority, NULL);
      thread_yield();
    }
  }
  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
  intr_set_level(old_level);
}
//---------------------------------------------------------------------------------
------------------------
```

>B4: To handle nested donation we were planning on checking to see if there were any donations to the thread by using the donation list variable. If there are donations then it would iterate through them and find the thread with the highest priority and recursively call itself with the highest priority donated thread until the true donor is reached.

pintos/src/threads/synch.c *Lines:261-329*

```
//---------------------------------------------------------------------------------
------------------------
/* 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();

  struct thread* cur_thread = thread_current ();
  struct thread* traversal;
  /* Revert the lock to its state before donation */
  /*
  while (!list_empty(&cur_thread->donor_list))
  {
    thread_set_priority(cur_thread->orig_priority);
    list_pop_front(&cur_thread->donor_list);
  }
  */
  /* Update the threads priority */

  if (!list_empty(&cur_thread->donor_list))
  {
    list_pop_front(&cur_thread->donor_list);
    cur_thread->priority = cur_thread->orig_priority;
    /*
    for (traversal = list_rbegin(&cur_thread->donor_list); traversal !=
list_rend(&cur_thread->donor_list); traversal = list_prev(traversal))
    {
      if (traversal->lock_waiting_for == lock)
      {
        list_remove(&traversal->elem);
      }
    }
    */

    /* Reference:
https://github.com/meskuwa/pintos-project-1/blob/master/src/threads/synch.c */
    struct list_elem* traversal;
    for (traversal = list_begin(&cur_thread->donor_list); traversal !=
list_end(&cur_thread->donor_list); )
    {
      struct thread *t = list_entry(traversal, struct thread, donor_elem);
      if (t->lock_waiting_for == lock)
      {
        struct list_elem* to_delete = traversal;
        traversal = list_next(traversal);
        list_remove(to_delete);
      }
      else
      {
        traversal = list_next(traversal);
      }
    }

    cur_thread->priority = cur_thread->orig_priority;

    if (!list_empty(&cur_thread->donor_list))
    {
      struct thread* max_donor = list_entry(list_front(&cur_thread->donor_list),
struct thread, donor_elem);
      if (cur_thread->priority < max_donor->priority)
        cur_thread->priority = max_donor->priority;
    }



  }

  sema_up (&lock->semaphore);
  intr_set_level(old_level);
  lock->holder = NULL;

}
//---------------------------------------------------------------------------------
------------------------
```

>B5: When a thread calls lock_release that thread will first revert back to its original priority, unless it holds another lock with donations for that one. Because the new thread that wants the lock is in the waiting list for that lock, and that list is ordered by priority, it will be the next lock to acquire it. The new thread will then call thread_acquire and make sure that it is able to hold the lock. Once the lock is acquired the thread is able to continue execution through the critical section.

---- SYNCHRONIZATION ----
>> B6: Describe a potential race in thread_set_priority() and explain how your implementation avoids it. Can you use a lock to avoid this race?

>B6: When thread donation is taking place. A nested donation can cause a race condition if an interrupt were to be called. A thread may need to donate to another thread’s priority to get a lock.

---- RATIONALE ----
>> B7: Why did you choose this design? In what ways is it superior to another design you considered?

>B7: We chose this design because we thought we had considered what the end goal of the project should have looked like. Our designs were flawed and we will need to spend some time with the gdb debugger for future success in this class. We will need to start on projects earlier. We knew that this project was going to be a lot of work. However, the amount of work was much more than we were anticipating and therefore will be accounted for in the future.

ADVANCED SCHEDULER [EXTRA CREDIT] (DID NOT ATTEMPT)
=================================

---- DATA STRUCTURES ----
>> C1: Copy here the declaration of each new or changed `struct' or
>> `struct' member, global or static variable, `typedef', or
>> enumeration. Identify the purpose of each in 25 words or less.

pintos\src\threads\thread.h

Data structure to hold the thread object (lines 83-111)

---- ALGORITHMS ----
>> C2: Suppose threads A, B, and C have nice values 0, 1, and 2. Each
>> has a recent_cpu value of 0. Fill in the table below showing the
>> scheduling decision and the priority and recent_cpu values for each
>> thread after each given number of timer ticks:

timer	recent_cpu			priority			thread
ticks	A	B	C	A	B	C	to run
-----	--	--	--	--	--	--	------
0							
4							
8							
12							
16							
20							
24							
28							
32							
36							

>> C3: Did any ambiguities in the scheduler specification make values
>> in the table uncertain? If so, what rule did you use to resolve
>> them? Does this match the behavior of your scheduler?
>> C4: How is the way you divided the cost of scheduling between code
>> inside and outside interrupt context likely to affect performance?

---- RATIONALE ----
>> C5: Briefly critique your design, pointing out advantages and
>> disadvantages in your design choices. If you were to have extra
>> time to work on this part of the project, how might you choose to
>> refine or improve your design?