CS 342 Design Document
Project 1B – Priority Scheduling and Donation

Rajat Kateja (r.kateja@iitg.ernet.in)
Maths and Computing
10012338

TA: Lalatendu Behera
Chetti Prasad

---- DATA STRUCTURES ----
Added to struct thread:

```
struct lock *waiting_on_lock;      /* To store the lock on which the thread is waiting */
struct lock *donated_for_lock;     /* To store the lock for which the thread has been
                                      donated the recent-est priority */
struct list pri_list;              /* To store the priorities donated in a list */
```

Created the following data structure for storing priorities donated and the locks associated with them. Elements of these data structure are added in the pri_list of each thread.

```
struct pri
{
        int priority;              /* Stores the priority donated */
        struct lock lock;          /* Stores the lock for which the priority was donated */
        struct list_elem elem;     /* List element to insert in the pri_list */
};
```

Added to struct lock:

```
struct list waiters;               /* List of threads waiting on lock */
```

---- ALGORITHMS ----

Priority Scheduling

Whenever a thread is unblocked using the thread_unblock() function, instead of simply pushing it in the ready list, it is inserted in order according to its priority. And also if the thread being unblocked has a priority greater than the currently running thread's priority, the current thread yields the CPU.

In thread_yield() function, when a thread yields the cpu, instead of simply pushing it in the ready list, it is inserted in order according to priority.

The ready list is always kept ordered, whenever a new thread is added to the list, or priority of a thread in the list changes, the list is re-ordered to maintain the highest priority thread at the front of the queue. This is done because the next_thread_to_run() function which schedules the threads returns the front of the queue. The re-ordering can be avoided by re-implementing the next_thread_to_run() function, and returning the highest priority thread.

For all other queues, like semaphore waiters and lock waiters, re-ordering is not done at each change. Instead the thread to be unblocked from such a list is chosen according to maximum priority as defined in the function priority_max(). Although, still the insertions are done in-order.

Re-implementing thread_set_priority() function

If the thread wants to increase its priority, no restrictions are imposed. If it wants to lower its priority, cases are checked. If it has been donated some priority, then its actual starting priority (i.e. the last element in the thread's priority list), is set to the value. If it has not been donated, its priority is set to new_priority and checked if it is to yield the cpu.

Functioning of the pri_list

The pri_list of a thread stores all the previous priorities it has ever had, including the starting priority on which it is created, Along with the priorities it also stores the lock for which it was donated that particular priority. (The members of pri_list are struct pri, which has these two fields in addition to a list_elem for storing it in the list.) Whenever a thread is donated some higher priority, its previous priority and donated_for_lock are stored in this list, and the new values are set. When a thread is to get some lower priority, (because it has released some lock for which it was donated the priority), the values are set, but the previous priorities are not stored in the list, because those have already been discarded once the lock related to them has been released. Also the next priority to fall back to, after releasing a lock is the front element of the list, because that is the next priority for which it much release a particular lock.

Priority – Donations

Added a new element of list waiters to the struct lock. When a thread does a lock_acquire() on a particular lock, instead of sema_down, sema_try_down() is checked and while it returns false, i.e. the lock cannot be acquired, the thread adds itself to the lock->waiters list and blocks itself. But before blocking, it donates its priority to the thread which is holding the lock currently (identified by lock->holder).

For **nested donations**, the thread which is receiving the priority, is checked. If it is in turn waiting on a lock (waiting_on_lock != NULL), then the thread on which it is waiting (waiting_on_lock->holder) is given the priority. This process is continued in a for loop until a thread which is not waiting on any lock is reached. The donations are done using thread_change_priority() explained later.

Next while releasing a particular lock, it is checked if there are any waiting threads in the lock->waiters list. If the list is non-empty, maximum priority thread from the list is removed and unblocked. Then it is checked that the lock being released is the lock for which the thread was donated its recene-est priority. If the check is true , then the lock and priority of the thread are changed to the next (head) element in the threads pri_list. This is again done by using thread_change_priority() function. If the above check is false, it is further checked if the thread was ever donated priority for this particular lock. The check is done by iterating the pri_list of the thread. If the check is true, then that particular lock and its corresponding priority are removed from the pri_list, as there is no further need for this to be in the thread's pri_list.

void thread_change_priority(struct thread *t, int priority, struct lock *lock)

The function is used to set the priority of thread t to priority and set its donated_for_lock to lock. The functioning is as follows.

If the priority is to be decreased, simply decrease the priority and set the lock. This is done when a thread is releasing a lock.

If priority is to be increased (when a thread is donated some priority), then create a pri structure of previous priority, and if the lock is a new one, push it in the threads pri_list. Then if the thread was donated a priority for this particular lock once earlier as well, then remove that from the list, as this lock is now the current donated_for_lock.

Finally update the threads parameters, lock and priority. If the thread was in ready state re-order the ready list.

As a last check if the thread is in running state, check if it needs to yield the cpu.

---- SYNCHRONIZATION ----

For synchronization, as the codes here are of small enough length, I have used interrupt disabling. Interrupts are disabled whenever any list is being manipulated.

---- RATIONALE ----

The design can support nested donations upto any level because iterative algorithm is used instead of recursive. Also associating a pri_list with each thread allows to keep an easy track of the priorities donated to the thread.

One downside of the design is that it is extra conservative. The elements of waiting lists are added in order, although the removal is done by choosing the maximum priority element. The insertion can be done at the front of the list itself, reducing the overhead.

----- CHANGED CODE SNIPPETS -----

In thread.h

added in struct thread

```
struct lock *waiting_on_lock;      /* To store the lock on which the thread is waiting */
struct lock *donated_for_lock;     /* To store the lock for which the thread has been
                                      donated last priority */
struct list pri_list;              /* To store the priorities in a list */
```

Declared the following data dtructture

```
struct pri
  {
        int priority;
        struct lock *lock;
        struct list_elem elem;
  };
```

In synch.h

Added the following to struct lock

```
struct list waiters;                /* List of threads waiting on lock */
```

in thread.c

code of thread_unblock function

```
void
thread_unblock (struct thread *t)
{
            enum intr_level old_level;
            struct thread *cur;
            ASSERT (is_thread (t));

            old_level = intr_disable ();
            ASSERT (t->status == THREAD_BLOCKED);
            /*****************************************
            code changed for project 1B
            *****************************************/
           /  list_push_back (&ready_list, &t->elem);
            list_insert_ordered(&ready_list, &t->elem, less, NULL);
            t->status = THREAD_READY;
            cur=thread_current();
            if(t->priority > cur->priority && cur!=idle_thread)
            {
                  thread_yield();
            }
             //added code ends----------------------------//
             intr_set_level (old_level);
}
```

code of thread_yield function

```
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)
        {
                /*****************************************
          code changed for project 1B
          *****************************************/
       //  list_push_back (&ready_list, &cur->elem);
          list_insert_ordered(&ready_list, &cur->elem, less, NULL);
          //added code ends----------------------------//
        }
       cur->status = THREAD_READY;
        schedule ();
        intr_set_level (old_level);
}
```

code of thread_set_priority

```
void
thread_set_priority (int new_priority)
{
        struct thread *cur, *top_ready_thread;

        cur=thread_current();
        if(new_priority < cur->priority)
        {
                if(cur->donated_for_lock == NULL)
                {
                  cur->priority = new_priority;
                  top_ready_thread=list_entry(list_begin(&ready_list), struct thread, elem);
                  if(new_priority < top_ready_thread->priority)
                    thread_yield();
                }
          else
          {
                list_entry(list_back(&cur->pri_list), struct pri, elem)->priority = new_priority;
          }
        }
        else
        {
                cur->priority = new_priority;
        }
}
```

code of init_thread function to initialise the newly added data mambers of struct thread

```
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;
	t->magic = THREAD_MAGIC;

	/**********************added for 1A*********************************/
	t->ticks_to_sleep = -1;
	/**********************added for 1B*********************************/
	t->waiting_on_lock = NULL;
	t->donated_for_lock = NULL;
	list_init(&t->pri_list);

	list_push_back (&all_list, &t->allelem);
}
```

added the following two functions

```
/****************************************
Defining functions for project 1B
****************************************/
bool less(struct list_elem *a_, struct list_elem *b_, void *no_use)
{
	struct thread *a, *b;
	a=list_entry(a_, struct thread, elem);
	b=list_entry(b_, struct thread, elem);
	return a->priority>b->priority;
}

void thread_change_priority(struct thread *t, int priority, struct lock *lock)
{
	enum intr_level old_level;
	struct pri *prev_pri;
	struct list_elem *e;
	if(t->priority < priority)
	{
		prev_pri = (struct pri *)malloc(sizeof(struct pri *));
		prev_pri->priority = t->priority;
		prev_pri->lock = t->donated_for_lock;
		old_level=intr_disable();
		if(prev_pri->lock != lock )
			list_push_front(&t->pri_list, &prev_pri->elem);
		else
			free(prev_pri);
		for(e=list_begin(&t->pri_list);e!=list_end(&t->pri_list);e=list_next(e))
		{
			struct pri *pri_elem = list_entry(e, struct pri, elem);
			if(pri_elem->lock == lock)
			{
```

```
                        list_remove(e);
                        break;
                }
        }
        t->priority = priority;
        t->donated_for_lock = lock;
        if(t->status==THREAD_READY)
        {
                list_remove(&t->elem);
                list_insert_ordered(&ready_list, &t->elem, less, NULL);
        }
        else if(t->status == THREAD_RUNNING)
        {
                struct thread *top_ready_thread = list_entry(list_begin(&ready_list), struct
                                                                thread, elem);
                if(priority < top_ready_thread->priority)
                thread_yield();
        }

        intr_set_level(old_level);
    }
    else
    {
        t->donated_for_lock = lock;
        t->priority = priority;
        if(t->status==THREAD_READY)
        {
                list_remove(&t->elem);
                list_insert_ordered(&ready_list, &t->elem, less, NULL);
        }
        else if(t->status == THREAD_RUNNING)
        {
                struct thread *top_ready_thread = list_entry(list_begin(&ready_list), struct
                                                                thread, elem);
                if(priority < top_ready_thread->priority)
                thread_yield();
        }
    }
}
//function defining ends----------//
```

in synch.c

code of function sema_down

```
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);

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

code of sema_up

```
void
sema_up (struct semaphore *sema)
{
      enum intr_level old_level;

        ASSERT (sema != NULL);

        old_level = intr_disable ();
        sema->value++;
        if (!list_empty (&sema->waiters))
        {
          /**********modified for 1B: removal by choosing max priority
                                    element*******************/
              struct list_elem *e=list_max (&sema->waiters, priority_max, NULL);
              list_remove(e);
              thread_unblock (list_entry (e, struct thread, elem));
        }
      intr_set_level (old_level);
}
```

code of lock_acquire

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

      //  sema_down (&lock->semaphore);
/*********************added for 1B**********************************/
/*
Instead of making sema down, new implementation tries to get sema down, and if unsuccessful
goes to waiting state after adding itself to lock->waiters list.
```

```
*/
        old_level = intr_disable();
        while(!sema_try_down(&lock->semaphore))
        {
            thread_current()->waiting_on_lock = lock;

            list_insert_ordered(&lock->waiters, &thread_current()->elem, priority_ordered,
                    NULL);

        /************************added for 1B: nested donations************************/
            struct thread *lock_holders;
            struct lock *lock_held;

            lock_held = lock;
            for(lock_holders = lock->holder; ; lock_holders = lock_holders->waiting_on_lock->holder)
            {
                if(lock_holders->priority < thread_current()->priority)
                {
                    thread_change_priority(lock_holders, thread_current()->priority,
                                lock_held);
                }
                if(lock_holders->waiting_on_lock == NULL)
                    break;
                else
                    lock_held = lock_holders->waiting_on_lock;
            }
            thread_block();
        }
        intr_set_level(old_level);
        lock->holder = thread_current ();
}
```

code of lock_release

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

        struct thread *cur;
        enum intr_level old_level;
        old_level=intr_disable();
        ASSERT (lock != NULL);
        ASSERT (lock_held_by_current_thread (lock));

        sema_up (&lock->semaphore);
        lock->holder = NULL;
 /*****************************************************
 Implementation made for 1B
 If the lock waiters list is non-empty unblock the maximum priority thread.

 If the thread was donated recentest priority for the lock being released, then
```

```
  fall back to the previous priority in the threads priority list.

  If the thread was ever donated priority for the lock, remove that priority from the priority list.
  ****************************************************/
        if(!list_empty(&lock->waiters))
        {

                struct list_elem *e = list_max(&lock->waiters, priority_max, NULL);
                list_remove(e);
                list_entry(e, struct thread, elem)->waiting_on_lock = NULL;
                 thread_unblock(list_entry(e, struct thread, elem));

        }
        cur=thread_current();

        if(cur->donated_for_lock == lock)
        {
                if(!list_empty(&cur->pri_list))
                {
                  struct pri *pri_elem = list_entry(list_pop_front(&cur->pri_list), struct pri, elem);
                  struct lock *prev_lock = pri_elem->lock;
                  int new_priority = pri_elem->priority;
                  free(pri_elem);
                thread_change_priority(cur, new_priority, prev_lock);
                }
        }
        else
        {
                struct list_elem *e;
                for(e=list_begin(&cur->pri_list);e!=list_end(&cur->pri_list);e=list_next(e))
                {
                        struct pri *pri_elem = list_entry(e, struct pri, elem);
                        if(pri_elem->lock == lock)
                        {
                          list_remove(e);
                          free(pri_elem);
                          break;
                        }
                }
        }
        intr_set_level(old_level);

}
```

code of cond_signal

```
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));
/*************************************************************************
Implemented for 1B
signal the highest priority thread waiting on the condition
*************************************************************************/

        if (!list_empty (&cond->waiters))
        {
                struct list_elem *e;
                struct list_elem *max_elem;
                int max_pri = -1;
                for(e=list_begin(&cond->waiters);e!=list_end(&cond->waiters);e=list_next(e))
                {
                        struct semaphore_elem *s_elem = list_entry(e, struct semaphore_elem, elem);
                        if(list_entry(list_front(&s_elem->semaphore.waiters), struct thread, elem)-
                                                >priority> max_pri)
                        {
                                max_pri = list_entry(list_front(&s_elem->semaphore.waiters), struct
                                                                        thread, elem)->priority;
                                max_elem = e;
                        }
                }
                list_remove(max_elem);
                sema_up (&list_entry (max_elem, struct semaphore_elem, elem)->semaphore);
        }

}
```