---- GROUP MESH----

Slim Ghodhbane <ghodhbas@lafayette.edu>
Hiep Le <leh@lafayette.edu>
Emmanuel Jimenez <jimineze@lafayette.edu>
Matthew Gerber <gerberm@lafayette.edu>

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.

In thread.h

We changed thread struct so that each thread element can contain information for when it needed to wake up.

```
struct thread
  {
    /* Owned by thread.c. */
    int64_t wake_up_time;
}
```

We created a doubly linked list of currently sleeping threads. This list is ordered by increasing wake up time.

In timer.c

```
Static struct list sleep_list;
-------------------------------------------------------------------------
------------
void
timer_sleep(int64_t ticks){
if(ticks <= 0) return;
  ASSERT (intr_get_level () == INTR_ON);
  enum intr_level old_intr_level = intr_disable ();
  thread_sleep(ticks);
  intr_set_level(old_intr_level);
}
```

In thread.c

The thread comparator method returns true or false depending on which thread has a large wake_up_time

```
bool
thread_comparator(struct list_elem *a, struct list_elem *b, void *aux){
  struct thread *t1 = list_entry(a, struct thread, elem);
  struct thread *t2 = list_entry(b, struct thread, elem);
  return t1->wake_up_time < t2->wake_up_time;
}
```

This method puts the current thread into the sleep list and sets its wake_up_time. It calls thread_block on the current thread as well.

```
void
thread_sleep(int64_t ticks){
  struct thread* cur = thread_current();
  cur->wake_up_time = ticks + timer_ticks();
  list_insert_ordered(&sleep_list, &cur->elem, thread_comparator, NULL);
  thread_block();
}
```

This method loops through the thread list and wakes up and removed those threads that are ready base on their wake_up_time

```
void
thread_wake (void){
  int64_t curr_time = timer_ticks();
  while(list_begin(&sleep_list) != list_end(&sleep_list)){
    // get head
    struct thread *head = list_entry(list_begin(&sleep_list), struct
thread, elem);

    if(head->wake_up_time > curr_time) break;

    list_remove(list_begin(&sleep_list));
    thread_unblock(head);
  }
}
```

Algorithms

>> A2: Briefly describe what happens in a call to timer_sleep(), including the effects of the timer interrupt handler.

We have modified timer_sleep in timer.c to stop the thread from getting any new interrupts (after making sure that the interrupts are on - intr_get_level () == INTR_ON) and call a thread sleep method. The sleep method then puts that thread into an ordered sleep queue with the time it needs to wake up which is the current tick plus the ticks it needs to sleep. The thread is then blocked and the original interrupt state is reset.

For each tick increase (called by timer_interrupt) we look at the head of the list and if the wake time is the current time we unblock that thread and remove it from the list. We then check the next head and continue until the list is empty or the head of the list has a different wake up time than the current tick.

>> A3: What steps are taken to minimize the amount of time spent in the timer interrupt handler

We have made a method called thread_sleep that blocks the thread instead of yielding and adding it into a queue. The queue is then looked at during each time tick in a different function, instead of having the thread wait inside the interrupt. This helps minimize the time we spend in the timer interrupt handler.

Synchronization

>> A4: How are race conditions avoided when multiple threads call?

Interrupts are turned off when the blocking of a thread occurs, not allowing another thread to be put to sleep at the same time.

>> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

Interrupts are turned off when thread_block is called.

Rationale

>> A6: Why did you choose this design? In what ways is it superior to another design you considered?

We considered building a list that saves the blocked threads and wakes them up when needed. To make this more efficient, instead of looping through the whole list every single time, we just

decided to go with an ordered list and only check the head of the list. This will allow us to save look-up time. The list is ordered by when the threads need to wake up.

PRIORITY SCHEDULING

==========

---- DATA STRUCTURES ----

In thread.c
Update priority is used to ensure that the current thread running is always the thread with the highest priority. Ready_list is ordered with the head being the thread with the highest priority. The function when a thread is created, unblocked or changed in terms of its priority value.

```
void update_priority(void)
{
  if (!list_empty(&ready_list))
  {
      struct thread *head =
      list_entry(list_begin(&ready_list), struct thread, elem);

      if (thread_current()->priority < head->priority)
      {
      thread_yield();
      }
  }
}
```

In thread_yield and thread_unblock, instead of merely pushing the thread to the ready_list

```
//list_push_back (&ready_list, &t->elem);
```

We insert in order into the ready_list such that the head is always the the thread with the highest priority

```
 list_insert_ordered(&ready_list, &t->elem, (list_less_func
*)thread_comparator_priority, 0);
```

A priority comparator is used to compare threads

```
bool thread_comparator_priority(struct list_elem *a, struct list_elem *b,
void *aux)
{
  struct thread *t1 = list_entry(a, struct thread, elem);
```

```
  struct thread *t2 = list_entry(b, struct thread, elem);

  return t1->priority > t2->priority;
}
```

In thread_set_priority, if the new priority set is lower than the old priority, a thread with previously lower priority than the current run might be able to run. As such, update_priority() is called to ensure one with the highest priority is run.

```
void
thread_set_priority (int new_priority)
{
  enum intr_level old_level;
  ASSERT (!intr_context ());
  old_level = intr_disable ();

  int old_priority = thread_current()->priority;
  thread_current()->priority = new_priority;

  if (old_priority > new_priority)
  {
      update_priority();
  }

  intr_set_level (old_level);
}
```

In synch.c, instead of pushing any threads waiting on the semaphores to the back, we order the threads based on the thread priority.

```
// list_push_back (&sema->waiters, &thread_current ()->elem);
list_insert_ordered(&sema->waiters, &thread_current ()->elem,
      (list_less_func *)thread_comparator_priority, NULL);
```

Similar for condition waiters. We compare the priority of the semaphore holder in sema_comparator

```
//list_push_back (&cond->waiters, &waiter.elem);
list_insert_ordered(&cond->waiters, &waiter.elem,
      (list_less_func *)sema_comparator, NULL);
```

```
Bool
sema_comparator(const struct list_elem *a,
  const struct list_elem *b,
  void *axu)
{
  struct semaphore_elem *s1 = list_entry(a, struct  semaphore_elem, elem);
  struct semaphore_elem *s2 = list_entry(b, struct  semaphore_elem, elem);

  return s1->holder->priority > s2->holder->priority;
}
```

Added variable to struct semaphore_elem. This will keep track of the thread that needs to finish before this thread can run.

```
struct thread* holder; // indicate threads waiting on COND to wake up
```

---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?

Insert threads and semaphores into waiting lists in the descending order of priority and pop out the first thread in the lists.

```
Lock -- ready_list
Semaphore -- sema->waiters
Condition -- cond->waiters
```

---- 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?

When changing a thread's priority, another thread can be scheduled before this thread finishes changing its priority. This can mean that the thread's priority is not updated promptly, causing

inaccuracies in the order in which threads are run. As such, we disabled interrupts in thread_set_priority().

---- RATIONALE ----

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

We picked this design because it allows us to order threads, semaphores, and conditions by their priority level (or the priority level of the thread holding them). Then we will always pick the first element in the list to run because it has the highest priority. This design is easier to understand and implement than other designs (like unordered list and pick thread during run time).

ADVANCED SCHEDULER

---- 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.

- Added a new header file fixed_point.h, defined a new type called fixed_point which is used for fixed point calculation. Some of the calculations are included below.

```
typedef int fixed_point;
// Convert a value to a fixed-point value
#define FP_CONVT(A) ((fixed_point)(A*F))

// Add two fixed-point values
#define FP_ADD(A,B) (A + B)

// Add 2 fixed point values
#define FP_ADD_MIX(A,B) (A + (B*F))

// Subtract 2 fixed point values
#define FP_SUB(A,B) (A - B)

// Subtract int and fixed point
#define FP_SUB_MIX(A,B) (A - (B*F))
```

In thread.h, added three new variables. These variables are for the use of calculating the priority of threads.

```
int nice
fixed_point recent_cpu
fixed_point load_avg
```

In timer.c, we change how each timer interrupt functions to account for the recent_cpu increase and priority levels recalculation.

```
increase_recent_cpu ();
if (ticks % TIMER_FREQ == 0)
  update_load_avg_and_recent_cpu ();
else if (ticks % 4 == 0)
  update_mlfqs_priority (thread_current ());
```

In thread.c, added a new global variable for load_avg:

```
fixed_point load_avg;
```

Thread_set_nice is also modified to prompt the current thread to yield. The priority levels of the current thread will be calculated and the scheduler will again pick the thread with the highest priority level to run.

```
void
thread_set_nice (int nice)
{
  thread_current () -> nice = nice;
  update_mlfqs_priority(thread_current ());
  thread_yield ();
}
```

Getting recent_cpu and load_avg numbers will make use of the fixed_point math specified in fixed_point.h

```
Int thread_get_load_avg (void)
{
  return FP_ROUND (FP_MULT_MIX (load_avg, 100));
}

Int thread_get_recent_cpu (void)
{
  return FP_ROUND (FP_MULT_MIX (thread_current () -> recent_cpu, 100));
```

```
}
```

Increase_recent_cpu increments the recent_cpu count of the running thread

```
Void increase_recent_cpu (void)
{
  struct thread *curr = thread_current ();
  if (curr == idle_thread)
      return;
  curr->recent_cpu = FP_ADD_MIX (curr->recent_cpu, 1);
}
```

Update_load_avg_and_recent_cpu(void) will update the corresponding values for all threads. Update_mlfqs_priority(struct thread *t) will update the priority of one specific thread. This is usually called for the currently running thread.

---- 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:

priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)
recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice

Timer ticks	Recent Cpu A	Recent Cpu B	Recent Cpu C	Priority A	Priority B	Priority C	Thread to run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	B
12	12	4	0	60	60	59	A
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	C
28	16	8	0	59	59	58	B
32	16	12	4	59	58	58	A
36	20	12	4	58	58	58	C

>> 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?

The scheduler does not specify which thread to run when there are multiple threads of the highest priority. In our implementation, the current thread with the highest priority will yield if there is another thread with the same priority in the queue. Once there are one or more ready threads with the highest priority as the running thread, the current running thread goes back to the end of the highest non-empty ready queue. Then we run the first thread in the highest non-empty ready queue.

>> C4: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

For a regular timer tick, we only update the recent_cpu count of the current running thread. Every fourth tick, we will update the current thread's priority levels. Doing so minimizes the workload in the three regular ticks.

---- 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?

We use one ready_list that saves threads in a descending level of priority. This simplifies the code, allows easier manipulation of the list and saves us space compared to having queues for each priority. The priority of every thread is recalculated every 4 ticks. Once the priority of a thread changes, we remove the thread from its current ready queue and push it back to the corresponding ready queue.

However, if the priority of a thread changes significantly, it will take more time to find the correct insertion place for the thread assuming the list is relatively full. In that case, the design with multiple queues might be more effective at finding the correct list to insert the thread.

>> C6: The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not?

Fixed-point math is necessary when calculating priority, recent_cpu and load_avg, and is implemented in a new header file called "fixed-point.h". The arguments can be an int or a fixed-point number. The operations supported are: ADD, SUB, MUL, DIV and CONVERT. The fixed-point math is implemented as a macros to help with performance. When using the

fixed-point variables in an application, abstracting functions, and using new variables also provides good readability.