Team:
Ankur Bhat
Paras Chauhan **PROJECT 1: DESIGN DOCUMENT**
Phani Dharmavarapu

--- GROUP ----

>> Fill in the names and email addresses of your group members.

Ankur Bhat <ankurbt@ccs.neu.edu>
Paras Chauhan<pchauhan@ccs.neu.edu>
Phani Dharmavarapu <phanidv@ccs.neu.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.

Answer:

1. We introduced a new list that consists of threads that are in sleeping state.

static struct list sleep_list;

2. We introduced a new state in thread life cycle as highlighted below. This state is given to threads that must sleep for a required duration and yield the CPU to other threads.

```
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_SLEEPING   /* New state for sleeping threads */
 };
```

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

---- ALGORITHMS ----

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

Core Algorithm :-
A new method call is made :- **thread_sleep(ticks);** (The new method is in thread.c)

Here we do the following:

a) disable interrupt.
b) if the current thread isn't the system idle thread then :
    i. calculate the time when the thread needs to wake up.
    ii. insert the thread into **sleep_list**.
    iii. change thread state to **SLEEPING**
    iv. lastly call **schedule()** , to perform switch.
c) enable interrupt again.

Now, by design, **thread_tick()** is called by timer interrupt handler at each timer tick. Here we have introduced 2 new modules :

a. **wake_sleeping_threads();**
Check if any sleeping threads, then wake them up. Basically ,move threads to ready list if their sleep time is over !

b. **check_thread_priority();**
Check following:
    i. if the time slice for current thread has expired.
    i. if the priority of current thread is less than the anyone waiting in ready list.

Optimization :
Ordered sequencing of threads in sleeping-list and ready-list : Threads are inserted in an ordered fashion in both these lists.

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

1. From efficient coding perspective, we have kept interrupt handler sections minimum with least lines of code within the interrupt handler section.
2. We can see that majority of code that is written inside interrupt handler block is of changing priorities. Also, majority of the times these priorities are received from an element which is present in a list. Hence we have made sure that effort in getting the element from the list should be the minimum. How have we done that? Keeping the list ordered (Highest priority -> Lowest Priority).

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?
Answer: Interrupts are turned off while catering a thread.

>> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?
Answer: Same as A4.

---- RATIONALE ----

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

1. The **sleep_list** keeps all the threads that need to sleep.
2. We have done the optimization as pointed out by professor to keep the **sleep_list** sorted.
3. On slide 30 for project1 ppt, the thread's **wake-up** code is being shown as part of **schedule()**. We find it a bit odd since methods like **thread_block** also calls **schedule() .**

Instead we find **thread_tick** to be a better place, as it is called by timer interrupt handler at each timer tick. **schedule()** seems to a designed as a general purpose method that just schedules a next available thread.

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

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.

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

   The purpose of holding the thread in the semaphore_elem is to ensure that during the cond_wait, the insertion of the waiter elem in the waiters list of the condition variable happens according the priority of the thread.

2. struct thread
   {
   …
   int priority; /* Has the current priority of the thread. */
   int actual_priority; /* NEW FIELD: Introduced to hold the priority before donation.
   This will be used to set the priority back to it's original after the donation task is complete.
   This is also set when the thread_set_priority is called, until the donation is complete. Once the donation process is complete, this will be set to the priority as well.*/
   ...
3. **struct lock *required_lock;** /*NEW FIELD: This holds the lock which is required by this thread (in case it is waiting for it to be released). It will be set to NULL by default and also when a lock is no longer required */
   ...
4. **struct list received_priorities;** /* NEW FIELD: This field holds the threads which have donated priority to this thread as part of the donation process. It has the threads inserted in the order of highest priority. Hence, when the lock is released, the front element would be executed, if it also present in it's waiters list as well. */
5. **struct list_elem recieved_priorities_elem;** /* NEW FIELD: Introduced as an alternative to the using the property elem while iterating/removing as using elem would remove the elements from the ready list as well*/

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

>> B2: Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a .png file.)

The data structure which we've used is a list for holding the threads which have donated priority to this thread as part of the donation process. It has the threads inserted in the order of highest priority. Hence, when the lock is released, the front element would be executed, if it is also present in it's waiters list as well.

Consider the following scenario:
There are three thread, namely, Main thread, thread L(Lower Priority) and another thread H(Higher Priority).
Initially, the main thread runs and acquires lock A.

Main ---> lock A ---> X
L ---> lock B ---> req lock A(Blocked)

Donation 1
Main's priority = L's priority

H ---> req lock B(Blocked)

Donation 2
L's priority = H's priority
Main's priority = H's priority

---- ALGORITHMS ----

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

The solution we have come up is to insert the threads in descending order of their priorities into the waiting list of the semaphore. This way, the thread with the highest priority would be present at the front and would be fetched first when the wake/unblock method is called on the thread..

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

>> B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

The sequence of events have been described as follows:

1. call lock_accquire():
    - 1.1. lock_accquire() checks if the lock has any holder or not. If the lock has a holder then :
        - 1.1.1. set the **required_lock** of current thread = lock
        - 1.1.2. Now since the lock has already been acquired by a low priority thread and current thread is of greater priority, we insert the current thread into **received_piorities** list of the lock holder. This is to make sure that when the donations happen. the lock holder remembers the unwrapping sequence.
    - 1.2. if the lock doesn't have any holder then we skip step 1.1.
    - 1.3. sema_down() is called :
        - 1.3.1. We set a variable called **contention_lock = threaad_current()->required_lock,** which is the lock that current thread tried to acquire.
        - 1.3.2. if **contention_lock == NULL,** which means that there is no one blocking current thread and it runs smoothly.
        - 1.3.3. However, if **contention_lock != NULL,** then current thread donates its priority to the lock holder.
        - 1.3.4. Now we need check for donation nesting. Consider the following scenario :
          A starts with priority =30.
          A does a lock acquire on lock A. Things are fine till here !
          B interrupts and runs as its priority=40. This thread acquires lock B and requires lock A.
          However, lock A has been locked by thread A. So, lets donate. Now,
          Thread A has priority 40 (Running)
          Thread B has priority 40 (Waiting for thread A to finish)
          Now, a new thread comes into picture. Thread C with priority 45 and it requires lock B.

          So. lets donate again. Now,
          Thread B has priority 45 (Waiting for thread A to finish). As B is still waiting for A to finish which has priority 40, we need to pass this newly attained high priority till the last level of chain. So, now final priorities after donation are as follows:
          Thread C has priority 45 (Waiting for thread B to finish)
          Thread B has priority 45 (Waiting for thread A to finish)
          Thread A has priority 45 (Running)

          All set!

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

>> B5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.

The sequence of events have been described as follows:

1. call lock_release() :
   1.1. we set the priority of thread as follows:
   current_thread->priority = current_thread->actual_priority;

   The **actual_priority** is the backup of initial priority on which the thread was created. At **thread_init()** both **actual_priority** & **priority** are set to same value.

   1.2. Now we need to check if we had taken donations ever. How do we know it? Well, we just check if our **received_piorities** list is not empty. if the list is empty then just do a **sema_up()** and everything is fine.
   1.3. Now if the list is not empty then, we need to give the current thread its previous priority. However, before doing this lets remove all the threads in the **received_priorities** list that will be un-blocked after this release.
   1.4. Once that is done, lets check if the received_priorities list still has some threads left in it after the above deletion.
      1.4.1. If the resultant list is empty, just do a **sema_up()** and everything is fine.
      1.4.2. Otherwise, if it is not empty, lets update the priority as of one that it obtained from a thread which requires a different lock and has the highest priority among the new **received_priorities** list. This check ensures that lock releasing thread updates its priority according to donations it received.

   The check is as follows:

```
if (current_thread->priority > head_of_received_piorities->priority){
        current_thread->priority = head_of_received_piorities->priority;
}
```

---- 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?

>> Answer: Consider the following scenario:

There is a thread A which had initial priority of 31. Currently, thread_set_priority() can be called on thread A. Lets call it. Meanwhile we are changing the priority of thread A to a new value, an act of donation happened and A received a priority of say 34 (i.e. greater than initial, 31). As A has received a new priority, we should not allow it to set to new priority till it achieves its initial priority back. This is bad! Lets solve it

We have disabled interrupts when we set the priority of a thread. Hence, no other thread can run in that period of time till we change the priority. All set.

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

---- RATIONALE ----

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

1. We do list insertions in sorted order by priority. That make finding the highest priority thread in a list efficient. We did this with :
   1.1. thread->recieved_priorities list
   1.2. sema->waiters list
   1.3. cond->waiters list

2. Inspired from the way the **Pintos** is designed where each lock has the information about its holder, we have a similar property in thread. We have a property called **required_lock** in struct thread which stores the information about the lock which thread wants to acquire. This property is initialized only if the lock which a thread wants to acquire is not available, otherwise, this property is seE to NULL. This helps us in following two important ways:
   2.1. in **sema_down()**: This is the place where the donation of priorities happen. Now the donating becomes really simple. We are just giving the current thread's priority to current thread->lock->holder's priority. We are done.
   2.2. in **lock_release():** When a lock is released, we delete all the threads from our received_priorities as they can be un-blocked now. Now the lock which is passed as an argument to lock_release can be compared with each thread's required lock and that thread can be removed as:
   if (lock == thread->required_lock)
   delete thread

3. **actual_priority** was used to take a backup of the priority of thread when it is created. This variable held the original priority while the **priority** struct member kept on changing on account of donations. Also , this variable was overridden for test case : **priority-donate-lower**,

4. Explicitly maintaining **received_piorities** list with the thread ,helped in retrieving the sequence in which the priorities should be updated once a lock is released after a thread undergoes priority donations.

5. We had a requirement that the list of conditional waiters should be sorted by priority in descending order (highest_priority to lowest_priority). However, condition waiters is a list of sempamphore_elem which just has semaphore in it. It doesn't have any information about thread. So we had two choices :
   5.1. Either add sempahore_elem's elem to thread and retrieve the thread and then use that thread's priority for comparison during sorting.
   5.2. Or, keep the pointer of thread in semaphore_elem.
   5.3. Using a priority variable in the semaphore_elem.

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

We chose approach 5.2 over 5.1, 5.3. Why?

Flaws in approach 5.1: We have to keep a list_elem in thread's struct that will correspond to semaphore_elem. Then, while sorting the cond->waiters list we can retrieve the thread and use its priority for sorting comparison. However, this approach is bad as every thread may not require lock or the lock that it wants to acquire is readily available. In that case, the list_elem added to thread corresponding to semaphore_elem would be initialized(NULL) for no reason. This a bad programming practice.

Flaw in approach 5.3: We constantly have to update this variable in addition to the priority in the thread, whenever it has been updated.

Why approach 5.2 is good? Semaphore_elem is created only when the cond_wait method is called. Hence, a semaphore_elem corresponds to a single thread and hence having the thread pointer in sempahore_elem is better.

Team:
Ankur Bhat
Paras Chauhan
Phani Dharmavarapu

PROJECT 1: DESIGN DOCUMENT

SURVEY QUESTIONS
==========

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

>> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

>> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

>> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

>> Any other comments?