PROJECT 1: THREADS

DESIGN DOCUMENT

Islam Yousry <islamyousry16@gmail.com > ………………………………………………… 14
Hamza Hassan <hamzahassan835@gmail.com >.. 26
Mahmoud Manfy <mahmoud.manfy159@gmail.com>……………………………………… 57
Andrew Adel <andrewadel2013@gmail.com >.. 17

ALARM CLOCK

DATA STRUCTURES

Thread.c

```
struct thread
 {
   struct list_elem blocked_elem;
   int64_t waited_time;
 };
```

In *timer.c,* we create a list for storing time-blocked threads so we needed a *list-element* in the thread struct.

For the *waited-time* integer, we store the time(in unit ticks) that the blocked thread needs to wait before being unblocked.

***Timer*.c**

```
struct semaphore sema;
struct list blocked_list;
int64_t min_time;
```

- Here we define our list in which we store the blocked threads so that we can unblock them when their time comes.
- In order to reduce the overhead on the timer interrupt, we declared a *min-time* to hold the nearest time at which we have to unblock some thread.
- At each tick, we first check whether the *min_time* is less than or equal to the current time so we can unblock the suitable threads stored in the list. You can see that this reduces the number of times we access the *blocked_list.*
- As long as the list is global, we have to make sure that race conditions will not occur, so we use a semaphore for that purpose.

ALGORITHMS

timer_sleep()

When timer_sleep() is called

- The time at which the thread should be unblocked is calculated
- The value of this time is stored in waited_time of the thread
- The thread is blocked.

After it is **unblocked**, it unblocks another thread (if it wakes up at the same time) which by its turn would unblock another thread and so on.

Time overhead

To reduce the time at each tick we consider the following :

- We store the blocked elements in a sorted manner according to their time and priority, so we can get the thread with minimum time to wake up and maximum priority in a constant time.
- Each time a thread is unblocked, that thread checks whether there is some thread which has the same time to wake it up and so on. This reduces the overhead on the timer interrupt handler.

SYNCHRONIZATION

We use semaphore so that we can guarantee that no more than one thread accesses the *blocked_list* at the same time.

RATIONALE

- When we decided to use a list storing the blocked threads, we had two options:
  - ➢ Insert the thread to be blocked at the end of the blocked_list This wasn’t a good choice by any means as we had to iterate over the whole list each tick to choose the threads which would be unblocked. We found out that this way may consume more than one tick in the *timer_interrupt* which is supposed to take one tick !!.
  - ➢ Insert the threads in a sorted manner according to the time at which they will be unblocked This was more efficient because now at each tick, we don’t have to iterate over the whole list.

- As we described before when we first mentioned the semaphore, we chose using it over disabling the interrupts to get more flexibility and concurrency.

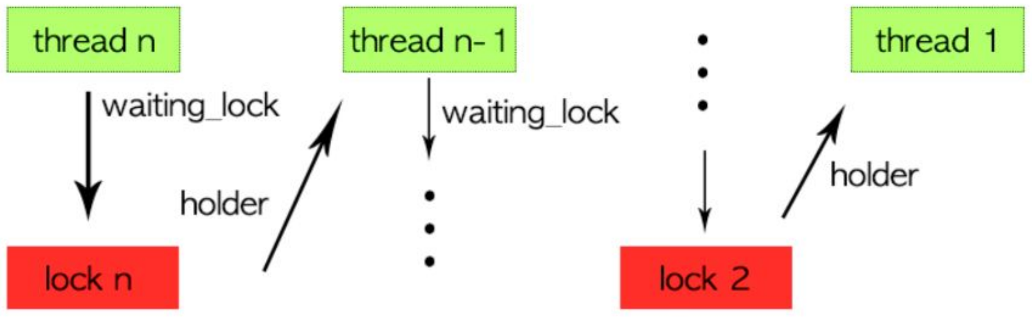
PRIORITY SCHEDULING

DATA STRUCTURES

```
struct lock
 {
   struct list_elem elem; // for the locks list in each thread
 };
struct thread
 {
      struct list acquired_locks;
      struct lock * seeking;
      int donate_priority;
 };
```

To handle multiple and nested donations we had to store a list of acquired locks for each thread so that we can update its donation priority efficiently. We also had to store the donated priority for each thread in a separate variable.
If a thread is waiting for a lock to be released, we store a pointer to that lock in the variable *seeking* which helps in the chain donation procedure.

Use ASCII art to diagram a nested donation.
(Alternately, submit a.png file.)



1) | low |	//a thread with low priority holds a lock.
2) | med | -> | low |	// a thread with medium priority waits on that lock.
3) | med | -> | med |	// that thread donates its priority to the previously-low thread.
4) | high | -> | med | -> | med |	// a high priority thread wants a lock that medium has.
5) | high | -> | high | -> | med |	// high priority thread donates it's priority to the next thread in the donation chain.
6) | high | -> | high | -> | high |	// the newly-high thread donates it's new priority to the first thread.

ALGORITHMS

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

- ➢ When the lock is released, the list of threads waiting for the lock is sorted according to their priority.

Describe the sequence of events when a call to lock_acquire()

- ➢ When a thread calls *lock_acquire(), we get two cases*

  1- No thread is holding the lock, so the thread can easily get the lock

  2- In case that the lock is held, we update the seeking variable for that particular thread and make donations- may be chain donations - if needed.

How is nested donation handled?

- ➢ Once the *lock_acquire* happens and we found out that the lock is held, we then implement the following procedure:
    1. Update the priority of the thread holding the lock (In case that its priority is less than the new thread's priority).
    2. If the thread holding the lock is waiting for other locks to release, we iterate over them and check if we have to update the priority of thread holding each one.
    3. Then we repeat this procedure to the threads holding each lock and so on until no need to update any priority or the thread is ready.

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

- ➢ When a thread releases a lock, we update the list of lock it owns. Then we update its priority.
  As mentioned before, the waiters list in the lock is sorted according to the priority so the higher priority thread will take the lock.

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?

- ➢ A race could affect the donation chain because any change of a priority for one lock holder may affect the priorities of all the chain threads.
- ➢ This is avoided by disabling the interrupt.
- ➢ No, we can't because while setting the priority we should disable the interrupt to ensure that the scheduler will choose the highest priority thread.

RATIONALE

The decisions we take in **part II** were simple and straightforward:

1- Make a list of acquired lock in each thread

2- when a thread is blocked because it is waiting on a lock, we store this lock in the pointer variable *seeking* described before.

Those decisions made the manipulation of nested, multiple and chain donation relatively easy taking into consideration that from a lock, we can get the sorted list of threads waiting for.

ADVANCED SCHEDULER

DATA STRUCTURES

thread.c

```
static struct real load_avg;
struct thread
 {
   int nice;
   struct real recent_cpu;
 };
```

We added the parameters needed to update the values of the load_average,recent_cpu every second so that we can calculate the priority using the formulas described below.

$load\_avg = (59/60) * load\_avg + (1/60) * ready\_threads$

$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$

$recent\_cpu = (2 * load\_avg)/(2 * load\_avg + 1) * recent\_cpu + nice$

ALGORITHMS

```
C1: 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 ticks	**Recent CPU**			**Priority**			**Thread to run**
	A	**B**	**C**	**A**	**B**	**C**	
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	8	4	0	61	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	4	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 behaviour of your scheduler?

- ➢ Yes, because there are some cases in which many threads have the same priority. One way to resolve it is to use the round-robin technique.
- ➢ Yes, our scheduler behaves this way.

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

- ➢ The best way to divide the cost of the scheduling is to create a thread which is responsible for updating the load average and the recent CPU for all the threads. This thread has a high priority and is unblocked each one second to calculate this task. Otherwise, It is blocked.
  This solution guarantees that the interrupt will not disable for a lot of time which increases the performance.
  Unfortunately, We didn’t have enough time to implement this solution and we didn’t have problems with tests when we used ***list_sort()*** and ***thread_foreach()*** functions.

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?

- ➢ In the task of updating priorities and load average, we used the ***list_sort()*** and ***thread_for_each()***.
- ➢ If we had extra time, we would implement the solution described in the previous question because it is more elegant and efficient.

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?

- ➢ Because the kernel deals only with integer numbers, so we had to make our own struct to perform implicitly float arithmetic operations but, communicates with the kernel with only integer numbers choosing rounding or any possible decision according to the operation we make.
- ➢ We decided to implement the fixed-point arithmetic via struct to achieve reusability and add a layer of abstraction.
- ➢ We used functions and macros to make the code more readable.