OPERATING SYSTEMS PINTOS PHASE 1

December 3, 2016

Name : Ahmed Eid Abdelmoniem
ID : 10
Email : ahmed.eid.csed18@gmail.com

Name : Shehab Kamal Mohmaed
ID : 33
Email : shehabelsemany@gmail.com

Name : Ahmed Helmi El-wakil
ID : 06
Email : a.helmielwakil@gmail.com

1 ALARM CLOCK

1.1 DATA STRUCTURES

1.1.1 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.

struct sleeping_thread {
struct semaphore sleeping_thread_semaphore;
largeint64_t time_to_wake_up;
struct list_elem auxiliary_list;
};
USAGE : This struct is used to represent thread in sleeping status.
Consist of :

- struct semaphore sleeping_thread_semaphore which is used to block and unblock sleeping thread.
- int64_t time_to_wake_up : To represent the time when the thread should wake up.
- struct list_elem auxiliary_list : To traverse throw the list of sleeping threads.

struct list sleeping_threads_list : This list is used to keep all sleeping thread in order to time of wake up for each thread.

bool sleeping_threads_comparable (struct list_elem *x, struct list_elem *y) :
This method is used to compare between twp sleeping threads and insert them in order to waking up time.

1.2 ALGORITHMS

1.2.1 A2

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

- void timer_sleep (int64_t ticks)
  - Create sleeping thread and calculate when it will be waked up.
  - disable the interrupt.
  - insert the sleeping thread in order into sleeping thread list using sleeping_thread_comparable method to sort the threads in order, The thread with low wake up time comes first than one with higher wake up time.
  - block the sleeping thread after inserting it into sleeping thread list.
  - Enable the interrupt.
- static void timer_interrupt (struct intr_frame *args UNUSED)
  - Disable the interrupt.

  - Actually, Timer interrupt is called each one tick so every time we try to take all threads which have wake up time smaller than or equal current time then unblock the thread.
  - List is already sorted so, Once we have thread with wake up time larger than current time we don't need to iterate throw the rest of list.

1.2.2 A3

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

- We insert sleeping threads in order into sleeping thread list which mean that once we have sleeping thread with wake up time greater than current time that's mean that we are sure that all the rest of list will never wake up on this tick.

1.3 SYNCHRONIZATION

1.3.1 A4

How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

- interrupts are disabled to avoid any race conditions that may take place.

1.3.2 A5

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

- interrupts are disabled to avoid any race conditions that may take place.

1.4 RATIONALE

1.4.1 A6

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

- First design was using list of sleeping threads and insert to it all sleeping threads then every one tick we iterate throw all the list to select the threads which should be waked up. But actually we see that's not efficient because every clock tick we iterate throw sleeping threads which will not be waked up now.
- To solve this problem we also use list of sleeping thread but we insert threads in order which is more efficient when we select which thread to wake up we iterate throw the list until first one will not wake up now we don't need to continue iteration we directly break from this loop because we are sure that all next iteration will be invain.

2 PRIORITY SCHEDULING

2.1 DATA STRUCTURES

2.1.1 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.

- **struct list donating_threads :** In struct thread, used to keep all the threads that is waiting for a lock acquired by that thread.
- **struct list_elem donation_elem :** In struct thread, used to add the thread to the list of donating_threads of another.
- **int base_priority :** In struct thread, used to indicate the base priority of the thread.
  NOTE: int priority now is used to indicate the actual priority of the thread not the base.
- **struct lock *required_lock :** In struct thread, used to indicate the lock that this thread is waiting to acquire.

2.1.2 B2

Explain the data structure used to track priority donation.

To track the priority donation we used two things, first a pointer to the lock that the thread is waiting for it. second a list that keeps all the threads waiting for any lock acquired by that thread. So if any thread tries to acquire a certain lock and that lock is acquired by another thread. the lock pointer in the current thread is set to the required lock and the current thread is added to the list of donating threads in the lock holding thread. That way we have a sequence of threads and locks where each thread points to the lock it needs to acquire and each lock points to the thread holding that lock.

In Figure 1 : shows nested donation where a scenario runs as follows. Thread C ”with lowest priority” acquires Lock C, then comes Thread B ”with higher priority” acquires Lock A and needs to acquire look B, so through the lock holder it reaches to Thread C and donates it’s priority to it and makes it’s lock pointer points to Lock B. Them comes Thread A ”with the highest priority” needs to acquire Lock A so through the lock holder reaches to Thread B and donates it’s priority to Thread B then through the lock pointer in Thread B reaches to Lock B and from it’s Lock holder reaches to Thread C and donates it’s priority to it.

2.2 ALGORITHMS

2.2.1 B3

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

firstly by adding the the threads in the semaphore waiters list in order according to their priority from the biggest priority to the lowest. then by resorting the list again - make sure the sorting wasn’t disturbed by donation - before removing the first thread in the list that - consequently - have the highest priority.

```
+---------------+               +-------------+             +---------------+
|               |   required    |             |   holder    |               |
|   Thread A    +-------------->|   Lock A    +------------>|   Thread B    +---------+
|               |     lock      |             |             |               |         |
+------+--------+               +-------------+             +----+-----+----+         |
       |                                                         |     |              |
       |                                                         |     |              |
       |_____________Thread A donating to Thread B_______________|     |              |
                                                                 |     |              | required
                                                                 |     |              |   lock
                              ____Thread A donating to Thread C__|     |              |
                              |                                        |              |
                              |  _____Thread B donating to Thread C____|              |
                              |  |                                                    |
                        +-----+--+-----+                  +------------+              |
                        |              |      holder      |            |              |
                        |   Thread C   |<-----------------+   Lock B   |<-------------+
                        |              |                  |            |
                        +--------------+                  +------------+
```

Figure 1: nested donation

2.2.2 B4

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

First check if the lock is acquired by checking the thread pointer ”holder” in the lock. If the lock is not acquired acquire it. if it is acquired make the current thread lock pointer points to the lock it needs to acquire and add the current thread to the list of the donating threads of the thread holding the lock. Then by entering a loop - that's how it deals with nested donation - that it's condition that the lock pointer in the thread in hand have a value (does not equal to NULL). Inside the loop checks if the priority of thread holding the required lock by the thread in hand. if the holder's priority is higher than the priority of the thread in hand then break out of the loop. If not then set the priority of the required lock holder by the priority of the thread in hand. By that loop we make sure passing the highest priority up the chain till it finds a thread that does not wait for any lock or a thread that have a higher priority that the current thread priority. Then call sema_down() with the lock.

2.2.3 B5

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

First initialize a variable max_priority to 0 and by iterating on the list of donating threads, if one of those threads are waiting on the lock it is releasing - by checking the required_lock pointer in the thread - then remove it from the list, Else compare its priority to the max_priority, if it is bigger then set max_priority to that thread's priority. After iteration check if the current thread's base priority is less than the value of the max_priority set the current thread priority to max_priority, if base priority is bigger set current thread priority to current thread base priority, call sema_up() with the lock that sorts the list of waiters before removing the highest priority thread. set lock holder to NULL and check if current thread needs to yeild.

2.3 SYNCHRONIZATION

2.3.1 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 condition can happen when using setting the thread priority while a donation is taking place, That may cause a wrong value to be set as the priority of the thread. Our implementation avoids it by disabling the interrupt in the function.
No we can't use a lock to avoid this race condition because another thread can donate it's priority to this thread at the same time while this thread is editing it's priority and we can not use a lock on a lock!

2.4 RATIONALE

2.4.1 B7

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

It was the most effective way with the least amount of add emory to the struct thread that also gives all the data needed efficiently with the best use of the struct lock.
another design that I considered was creating a new struct that carries an integer donated_priority, thread pointer donating_thread, lock pointer required_lock and list_elem. Then adding a list of that struct inside the thread. but of course it was space consuming and had a lot of redundant information.

3 ADVANCED SCHEDULER

3.1 DATA STRUCTURES

3.1.1 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.

new variables in **struct thread:**

- **int nice** : it determines how nice the thread should be to other threads . each thread has a nice value which can be only modified by fun. thread_set_nice();
- **int recent_cpu** :it is a measure of how much CPU time each process has received recently.The initial value of recent_cpu is 0 if it is initial thread or value of it's parent if it has.

new variables in class **thread.c** :

- **int load_avg** :it estimates the average number of threads ready to run over the past minute . this defined once for a system and it is modified very 1 sec.
- **int ready_threads** :it is a variable holds the number of ready threads.Used to calculate load_avg.

new type def in class **fixed-point.h** :

- **#define f (1<<14)** : it defines f to an integer equal to two power 14.
- **#define cnvrt_n_to_x(n) (n*f)** : shifted a value of n 14 bits to the left.
- **#define trunk_x_to_n(x) (x/f)** : shift a number 14 bits to the right and ignores fractions if it has.
- **#define round_x_to_nearest_n(x) (x>0 ? ((x+f/2)/f) : ((x-f/2)/f))** :shift a number 14 bits to the right and round it to nearest integer if it is float.
- **#define add_x_and_y(x,y) (x+y)** : add two numbers each shifted 14 bits to the left.
- **#define sub_y_from_x(x,y) (x-y)** :subtract two numbers each shifted 14 bits to the left.
- **#define add_x_and_n(x,n) (x+n*f)** : add a number shifted 14 bits to the left and (shift a number n 14 bits to left first).
- **#define sub_n_from_x(x,n) (x-n*f)** : subtract (shift a number n 14 bits to left first) from a number shifted 14 bits to the left.
- **#define mul_x_and_y(x,y) ((int)(((int64_t)x)*y/f))** : multiply two numbers each shifted 14 bits to the left.
- **#define mul_x_and_n(x,n) (x*n)** : multiply a number x shifted 14 bits to the left with another number.
- **#define div_x_by_y(x,y) ((int)(((int64_t)x)*f/y))** : divide two numbers each shifted 14 bits to the left.
- **#define div_x_by_n(x,n) (x/n)** : divide a number shifted into 14 bits to the left and an integer. Usage of these type defs : it is used to calculate load_avg and recent_cpu because they can't be floats because operating system deals only with integers.

3.2 ALGORITHMS

3.2.1 A2

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	ready
ticks	A	B	C	A	B	C	to run	list
0	0	0	0	63	61	59	A	(B,C)
4	4	0	0	62	61	59	A	(B,C)
8	8	0	0	61	61	59	B	(A,C)
12	8	4	0	61	60	59	A	(B,C)
16	12	4	0	60	60	59	B	(A,C)
20	12	8	0	60	59	59	A	(C,B)
24	16	8	0	59	59	59	C	(B,A)
28	16	8	4	59	59	58	B	(A,C)
32	16	12	4	59	58	58	A	(C,B)
36	20	12	4	58	58	58	C	(B,A)

3.2.2 A3

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?

- YES, in case of two or more threads have the same priority , we use round robin algorithm in this case (first in first out) .(i.e :if runing thread have the same priority as highest in ready list after it's time slice has finished then runing thread goes to ready list and the thread that was in ready list runs , if after the time slice or runnig thread has finished and two ready threads have same priority then first to enter in ready list first to run.).This behavior matches my scheduler.

3.2.3 A4

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

- due to the need of changing priorities all at same time so we need to do this inside interrupt context also recent_cpu value so most of added code is inside interrupt context.so it reduces the code performance , also the algorithm used to calculate priority for all threads every 4 ticks given that recent_cpu and load_avg are updated every 1 second so much waste of time to update priorities which also reduces the performance.we see that we only need to update current_thread priority only every 4 ticks and all threads every 1 sec.

3.3 RATIONALE

3.3.1 A5

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?

- it is a very simple implementation every second recent_cpu and load_avg are updated . after 4 ticks priority of all threads are updated using the new values of load_avg and recent_cpu and according to these new priorities the scheduler chooses the thread with highest priority to run.
- Advantages : it's very easy to implement , readible,only short code need to be added,few variable.
- Disadvantages : we couldn't used complex data structures in the project , when we upadte variable we need to disable interrupt instead of using lock for each variable.
- we have much time we think we will try to add lock for each variable to use it in stead of disabling interrupts which lead to less performance.also we will try to change algorithm of updating priority of all threads every 4 ticks we will only change current running thread and every 1 second we will update recent_cpu and load_avg and priority of all threads.

3.3.2 A6

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?

- we implemented fixed-point math in a header file. The conversions between integers and fixed-point and arithmetic was abstracted away in this file. we used the standard functions as described in the Pintos documentation and called these functions in my mlqfs calculation functions in thread.c. Abstracting the fixed-point functions allowed for better readability when calculating the mlqfs thread.c functions.