Operating Systems

PintOS

phase 1

Submitted by:

Omar Shawky Abd El-Salam	43	Walid Ahmed Zattout	69
Yahia Mohamed El-Saadawy	70	Youssef Mohamed Al-Sharawi	74

Faculty of Engineering, Alexandria University

Computer and Systems Engineering Department

December 2020

GROUP:

Omar Shawky <omaralgamil@gmail.com>
Walid Zattout <wzattout@gmail.com>
Yahia El-Saadawy <yahiaElsaadawy@gmail.com>
Youssef Al-Shaarawi <yfalshaarawi@gmail.com>

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.

- Global:
  ```
  struct list sleeping_threads;          /*List of sleeping threads*/
  ```

- Thread Struct:
  ```
  int64_t time_to_wake_up;          /* time where thread should wake up*/
  ```

ALGORITHMS:

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

- Calculating the time that thread will not wake up before.
- Pushing that thread in the sleeping thread list according to its wakeup time in ascending order.
- Blocking the thread.
- At every tick, checking the list first element time.
- If the current time is greater than or equal the first element time remove it from the list and unblock this thread.
- repeat until the list is empty.

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

- The sleeping thread list is ordered so that the check step is always only on the first element not all elements.

SYNCHRONIZATION:

A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

- By saving the calling thread wakeup time in the thread struct.

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

- By saving the calling thread current time in a local variable in the function’s scope.

RATIONALE:

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

- To minimize the complexity of timer_interrupt() by using insert_ordered() that makes the check step is in O(1).

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.

- Thread Struct:

```
int don_priority;                 /* Donated priority */
struct list my_locks;             /* List of locks the thread holds */
struct list * blocking_sema_list;     /* Pointer to the waiters list for the blocking sema*/
struct thread *lock_holder ;      /* Pointer to the Thread Thats holds the lock that caused the blocking*/
```

- Lock Struct:

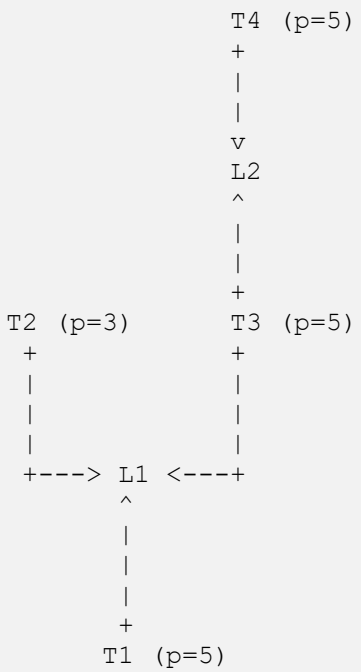
```
struct list_elem lock_elem;          /* List element for the emy_locks list in each thread*/
```

- struct semaphore_elem:

```
int priority;                         /* max priority the sema blocking*/
```

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

- Tree



ALGORITHMS:

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

- They are inserted by using insert_ordered() function that guarantees that the highest priority thread is the first element in the list.

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

- Let current thread that called lock_acquire be (X) and the thread that currently already holding this lock Y. I start checking if the lock wanted by X was previously taken by thread Y: i- if so, I start climbing the tree of locks holders that were each blocked on some specific lock held by their parent thread (lock_holder), changing the donation priority of each if necessary with maximum priority whether it was (X.priority, X.donated_priority,Y .donated priority). (still I want to edit this but I should stop when I reach a lock holder that has priority donation higher than X) ii- if not,then pass step i.
- sema_down is called if lock was held then X will be put to sleep.
- When X wakes up later then, I clear out the lock_holder of thread X to Null as the current holder of lock is X itself and add this lock to the list of X.my_Locks to be later used by lock_release.

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

- let current thread that called lock_release be (X). remove the lock released from the list of X.my_locks check the rest of X.my_locks on all waiters of these locks to get the new priority_donation of thread X.

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?

- Potential race condition: If the current thread calls thread_set_priority(), and the thread has a donated priority, the donated priority could be overwritten incorrectly. We avoided this race by saving only the highest donated priority and the original priority. The original priority is overwritten in any case, but the donated priority is only overwritted when the new priority is higher than the donated priority.

RATIONALE:

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

- It guarantees that every thread has its own donation priority, and no thread will go into deadlock or starvation conditions.

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.

- Global:
  ```
  struct real load_avg;                /*load average*/
  ```
- Thread Struct:
  ```
  int nice;                             /* Nice value for thread */
  struct real recent_cpu;               /* Thread's recent cpu */
  ```
- Real Struct:
  ```
  int val;                              /*The fixed point number*/
  ```

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:

timer	recent_cpu			priority			thread	ready list	ready list
ticks	A	B	C	A	B	C	to run	before	after
0	0.0	0.0	0.0	63	61	59	A	A -> B -> C	B -> C
4	4.0	0.0	0.0	62	61	59	A	B -> C -> A	B -> C
8	8.0	0.0	0.0	61	61	59	B	B -> C -> A	C -> A
12	8.0	4.0	0.0	61	60	59	A	C -> A -> B	C -> B
16	12.0	4.0	0.0	60	60	59	B	C -> B -> A	C -> A
20	12.0	8.0	0.0	60	59	59	A	C -> A -> B	C -> B
24	16.0	8.0	0.0	59	59	59	C	C -> B -> A	B -> A
28	16.0	8.0	4.0	59	59	58	B	B -> A -> C	A -> C
32	16.0	12.0	4.0	59	58	58	A	A -> C -> B	C -> B
36	20.0	12.0	4.0	58	58	58	C	C -> B -> A	B -> A

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?

- It has some ambiguities if two threads have the same priority which of them to run. So, we decided that the running thread yield when it finished its time slice and be pushed back to the end of the ready list, then the scheduler takes the thread of maximum priority, removes it from ready list and runs it, in case many threads have the same priority equals to the maximum priority in the ready list the scheduler will take the first one of them in the ready list. Our scheduler also matches does the same behavior.

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

- Most of the Advanced Scheduler code is between interrupt disable and reenable as it is called as an interrupt service routine for the timer. This can cause some performance leeks; however, we optimized the code inside the interrupt that all thread priorities are recalculated only when load_avg and recent_cpu is changed (every 100 ticks). Every 4 ticks we only update the priority of the current running thread only.

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?

- Advantages are that it is simple and very time efficient.
- Disadvantages are that it uses extra space for pre_computed_priority as a constant rather than calculating it every time the priority of the thread is recalculated.
- We might change fixed point to be a struct to be more readable and errors can find easily.

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?

- First, we implemented it as a set of functions but this implementation takes some in time in initialization, so we decided to change the implementation to be a set of macros to be more efficient in time and memory.