Project 1: Threads

author: 冯宇凡 fengyf2@shanghaitech.edu.cn

杨　斌 yangbin@shanghaitech.edu.cn

date: 2021/10/11

Consulted Sources:

- 如何优雅地完成PintOS (Project1、2)

  https://zhuanlan.zhihu.com/p/343328700
- "Pintos-斯坦福大学操作系统Project详解-Project1"

  https://www.cnblogs.com/laiy/p/pintos_project1_thread.html
- "What is a Semaphore?"

  https://www.baeldung.com/cs/semaphore
- "Harvard/CS61/Lecture19: Semaphores, Condition Variables, and Monitors"

  https://cs61.seas.harvard.edu/wiki/images/1/12/Lec19-Semaphores.pdf
- "CS450: Priority Donation 1 (Pintos Project)"

  https://www.youtube.com/watch?v=nVUQ4f1-roM
- "Haonan_Jia blogs: pintos_project1"

  https://hnjia00.github.io/2019/05/16/pintos-project1/
- Other guys' implementation

  https://github.com/wookayin/pintos

  https://github.com/codyjack/OS-pintos

  https://yuegong.netlify.app/static/files/pintos.pdf

ALARM CLOCK

DATA STRUCTURES

> 1. 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 struct member in `thread/thread.h:struct thread`:

```c
int wakeup_time;                    /* Time to wake up from sleeping. */
```

New static variable in `thread/thread.c`:

```c
/* List of all sleeping threads. Threads are added to this list
   when `timer_sleep ()` called and removed when they stop sleeping.*/
static struct list sleeping_list;
```

ALGORITHMS

> 2. Briefly describe what happens in a call to `timer_sleep()`, including the effects of the timer interrupt handler.

In a call to `timer_sleep()`:

- Get the number of current timer ticks
- Disable interrupts.
- Set the wake up time of current thread (`wakeup_time = start + ticks`).
- Insert current thread into sleeping list.
- Block current thread.

Effects of the timer interrupt handler:

- Update time ticks, thread ticks, idle ticks or user ticks or kernel ticks...
- Check if should perform an external interrupt.
- Check sleeping list and wake up all threads whose `wakeup_time` ⩽ `current_tick`

> 3. What steps are taken to minimize the amount of time spent in the timer interrupt handler?

When a thread is added to the `sleeping_list`, we use `list_insert_ordered` to ensure `sleeping_list` is ordered by threads' priority. So, in the timer interrupt handler, we only have to traverse threads in the front and pop eligible threads.

SYNCHRONIZATION

> 4. How are race conditions avoided when multiple threads call `timer_sleep()` simultaneously?

During the process of get current thread → set wake up time → send thread to wait list, interrupts are disabled.

> 5. How are race conditions avoided when a timer interrupt occurs during a call to `timer_sleep()`?

During the process of get current thread → set wake up time → send thread to wait list, interrupts are disabled.

RATIONALE

> 6. Why did you choose this design? In what ways is it superior to another design you considered?

We also considered implement our own min heap for `sleeping_list` and `ready_list` (also for problems in priority scheduling). However, the time complexity may not be much better since we not only need to access the top element, but also need to make changes to internal elements and resort. Also, the provided list already has functions of sorting and inserting order, so it will be easy to implement.

PRIORITY SCHEDULING

DATA STRUCTURES

> 1. 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 struct member in `thread/thread.h:struct thread`

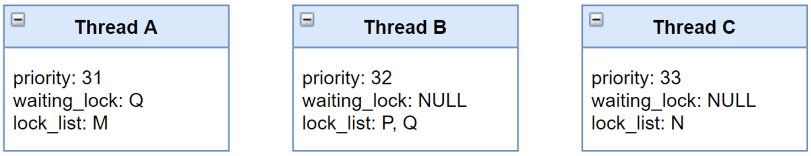
```
int prev_priority;                /* Previous priority. */
struct list lock_list;            /* List of locks held by the thread. */
struct lock* waiting_lock;        /* The lock that blocks the thread. */
```

New struct member in `thread/synch.h:struct lock`

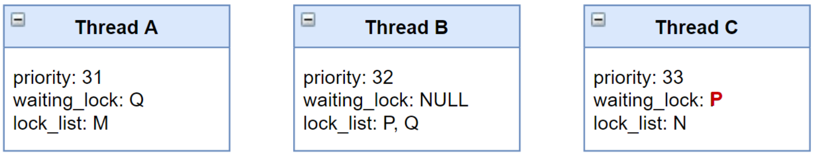
```
struct list_elem elem;           /* List element of the lock. Lock can be treated as a list element. */
int priority;                    /* Priority of the lock. */
```

> 2. Explain the data structure used to track priority donation. Use ASCII art to diagram a nested donation. (Alternately, submit a .png file.)

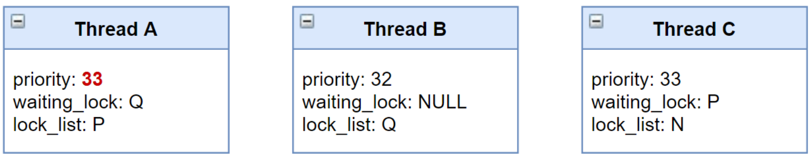
Suppose we have a initial state as bellow.



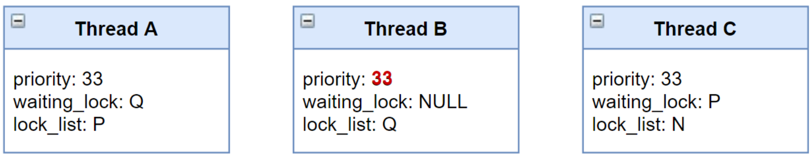
Now, thread C is trying to acquire lock P, it will add P into its `waiting_lock`.



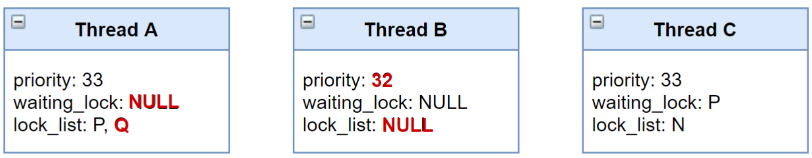
Then we will set priority of P's holder, Thread A, to the same as Thread C.



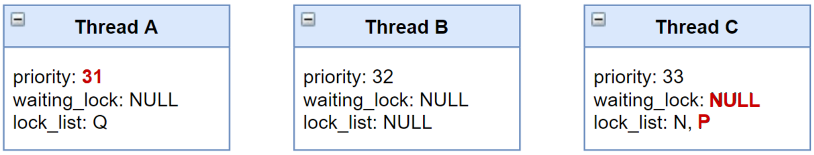
However, A is waiting for Q. We would set priority of Q's holder, Thread B, also to the same as Thread C.



Then Thread B is executed, releases lock Q and set its priority back to 32. Thread A acquires the lock.



For the next, Thread A is executed, releases lock Q and set its priority back to 31. Thread C acquires the lock.



ALGORITHMS

> 3. How do you ensure that the highest priority thread waiting for a lock, semaphore, or condition variable wakes up first?

We always maintain `struct list waiters` in `struct semaphore` as a priority queue. So that when calling `sema_up` and choosing thread to unblock, we always unblock the beginning, which is the thread with the highest priority.

> 4. Describe the sequence of events when a call to `lock_acquire()` causes a priority donation. How is nested donation handled?

Priority donation happens when `lock` already has a holder and `priority` of `lock->holder` < `priority` of current thread.

When priority donation happens:

- set `iter_lock = lock`
- in a while loop:
  - set priority of `iter_lock` same as priority of current thread
  - set priority of `holder` of `iter_lock` same as priority of current thread
  - `iter_lock` = `waiting_lock` of `holder` of current `iter_lock`

  the loop breaks when there the holder of `iter_lock` is not blocked by any other lock.

> 5. Describe the sequence of events when `lock_release()` is called on a lock that a higher-priority thread is waiting for.

When `lock` is released:

- remove `lock` from current thread's `lock_list`

  reset current thread's priority ⇒ max of `prev_priority` and priorities of holding locks
- as for a higher-priority thread is waiting for `lock`,
  - sort waiters of locks
  - unblock the thread with the highest priority
- yield the cpu

SYNCHRONIZATION

> 6. Describe a potential race in `thread_set_priority()` and explain how your implementation avoids it. Can you use a lock to avoid this race?

Thread A is calling `thread_set_priority()`, while another thread is donating its priority and wants to change A's priority, the race against priority of thread A happens.

We set `thread_set_priority ()` to an atomic operation. Disable all interrupts.

Yes. We can use a lock for the thread priority and release the lock after setting priority.

RATIONALE

> 7. Why did you choose this design? In what ways is it superior to another design you considered?

Record all locks the thread holding and the lock it waiting will help us find the donated priority easily. And let the `lock_list`, `sema_list` to be priority list will ensure the `waiting_list` and the running thread always be right.

ADVANCED SCHEDULER

DATA STRUCTURES

> 1. 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 struct member in `thread/thread.h:struct thread`

```
int nice;                           /* Nice. */
int recent_cpu;                     /* Recent CPU. */
```

New static variable in `thread/thread.c`

```
/* Load average. */
static fix_point load_avg;
```

New `typedef` in `thread/fix_point.h`

```
/* Type for fix point numbers. */
typedef int fix_point;
```

ALGORITHMS

> 2. 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` (A)	`recent_cpu` (B)	`recent_cpu` (C)	`priority` (A)	`priority` (B)	`priority` (B)	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	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

> 3. 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?

Threads may have same priority.

We use FIFO rule which match the behavior of our scheduler.

> 4. How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

For every four ticks, we only update the priority of current thread instead of updating all threads' priorities. That is because other threads' `recent_cpu` would keep unchanged before we update `load_avg` every second. In this way, we lower the calculation inside interrupt context and improve performance.

The calculations of `load_avg`, `recent_cpu` and `priority` must be placed in interrupt context, otherwise they would not be set at the correct time and to correct value. However, other operations such as re-sort the `ready_list` are placed outside of interrupt context, so we can minimize time losed when timer interrupted.

RATIONALE

> 5. Briefly critique your design, pointing out advantages an 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?

Instead of building 64 ready-to-run queues for different priority, we still maintain one single ready list. The advantage is that it is fast and easy to implement. However, if we have extra time, we may try implement 64 ready lists since it will save a lot of time of sorting the list.

> 6. 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 used a set of macros to manipulate fixed-point numbers. (Because there will be bugs otherwise.) This is because macro is faster than C functions. Also we used shift operators instead of multiply or dividing since they are faster.

Contribution

We read & search materials separately and understand codes together.

For all tasks, we do pair programming with one coding and another checking.

- Feng Yufan: task 1, reimplement of task 2
- Yang bin: initial implement of task 2, task 3

After finishing coding, we debug seperately and share the ideas of what might be wrong.