+-----------------------------------+
| CS 330 |
| PROJECT 1: THREADS |
| DESIGN DOCUMENT |
+-----------------------------------+

---- GROUP ----

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

Geonho Koh ghkoh97@kaist.ac.kr
Hyeongseop Kim gudtjqdl789@kaist.ac.kr

---- PRELIMINARIES ----

# of tokens to use: 1

>> If you have any preliminary comments on your submission, notes for the TAs, usage of tokens, or extra credit, please give them here.

>> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

---- REFERENCES ----

Alarm Clock : Hanyang University Embedded Software Systems Lab. *Pintos 6. Alarm System Call*
Lock, Semaphore, and Monitor : https://about-myeong.tistory.com/34

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.

-thread.h

```
Struct thread {
        …
        int64_t wake          //해당 thread가 일어나야할 시간
        …
}
```

-thread.c

```
Static struct list sleeping_list                    //thread_sleep에 의해서 block된 thread들의 list
Static struct int64_t min_wake=INT64_MAX            //sleeping_list에 있는 thread들의 wake 값 중 최솟값
```

---- ALGORITHMS ----

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

timer_sleep()함수는 원래 구현에서는 while 문을 반복하면서 start이후로 sleep해야하는 시간이 지날때까지 running->ready->schedule->ready가 반복되는 busy_waiting으로 구현되었다.

새로운 timer_sleep()함수는 wake를 계산하여, thread_sleep()함수에 전달한다.
thread_sleep()에서 현재 thread가 idle 이 아닐경우, sleeping_list에 현재 thread를 넣고, 만약 min_wake보다 현재 thread의 wake값이 더 작다면, min_wake의 값을 업데이트한다. 그리고 해당 스레드를 block 시킨다.

Timer-interrupt handler는 매 tick마다 tick을 1씩 증가시키고 증가된 후 만약 현재tick이 min_wake보다 크다면, thread_wake()함수를 호출하여 wake가 지난 thread들을 전부 깨운다.

thread_wake()함수는 sleeping_list의 thread들을 전부 봐서 현재 Ticks보다 작은 wake들을 가진 thread들을 깨우고, sleeping_list에서 해당 thread들을 remove해준다음 thread_unblock()함수를 호출해 해당 thread를 ready_list에 넣는다.

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

원래는 timer_sleep()의 while문으로 매tick마다 thread_yield()를 시키는 busy waiting 알고리즘을 사용했다.

하지만 수정한 알고리즘은 min_wake라는 변수를 사용하여, tick이 min_wake값을 초과할 때에만 thread_wake() 함수를 호출하는 효율적으로 알고리즘으로 작성했다.

---- SYNCHRONIZATION ----

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

pintos에서는 running thread가 하나로 이기 때문에 timer_sleep()이 동시에 호출되지 않는다.

Sleeping_list를 통해 timer_sleep()을 호출한 thread들을 한번에 관리하기 때문에 multiple thread들을 처리 할 수 있다. 또 thread_sleep()함수 안에서 intt_disable()을 했기 때문에 한 thread가 sleep되는 동안 문제가 발생하지 않는다.

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

thread_sleep()함수가 실행되는 중 timer interrupt가 발생하면 thread가 sleepinglist에 들어가나 block전에 reschedule되는 문제가 발생할 수 있다. 따라서 Thread_sleep()에서 intr_disable()을 사용하여 timer interrupt가 일어나지 않도록 했다.

---- RATIONALE ----

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

처음에는 단지 thread_block()이라는 함수를 사용해 단일 thread인 경우 해당 wake후에 thread_unblock()을 해주는 알고리즘을 생각했다.
하지만 여러개의 thread가 동시에 들어와서 sleep하는 thread들을 동시에 관리하기 위한 sleeping_list를 구상하였고, 매 tick마다 sleeping_list의 모든thread들의 wake값을 확인할 수 없으니,

min_wake라는 전역변수를 생각하여 min_wake가 지날 시 thread들을 깨우는 것을 design했다. 그리고 wake를 할때, wake가 지난 얘들을 다 깨우기위해 while loop를 사용했다.

생각되는 another design으로는 sleeping_List에 넣을때 List_push_back()이 아니라 list_insert_ordered()를 사용하여, wake순으로 list에 insert하여, wake를 할때 전체 list를 확인할 필요없이 sort된 순서대로 앞쪽의 thread들만 확인하여 wake를 하면 좀 더 효율적일 것 같다.

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.h

```
Struct thread{
        …
        int old_priority            //donation을 하기전에, 초기 priority를 저장하기 위함
        struct list locks           //해당 thread가 holder인 lock들의 List
        Struct lock *wait_lock      //해당 thread가 wait하고 있는 lock
        ...
}
```

-sync.h

```
Struct lock{
        ...
        struct list_elem lock_elem        //thread의 locks list에 넣어 추적하기 위한 list_elem
        ...
}
```

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

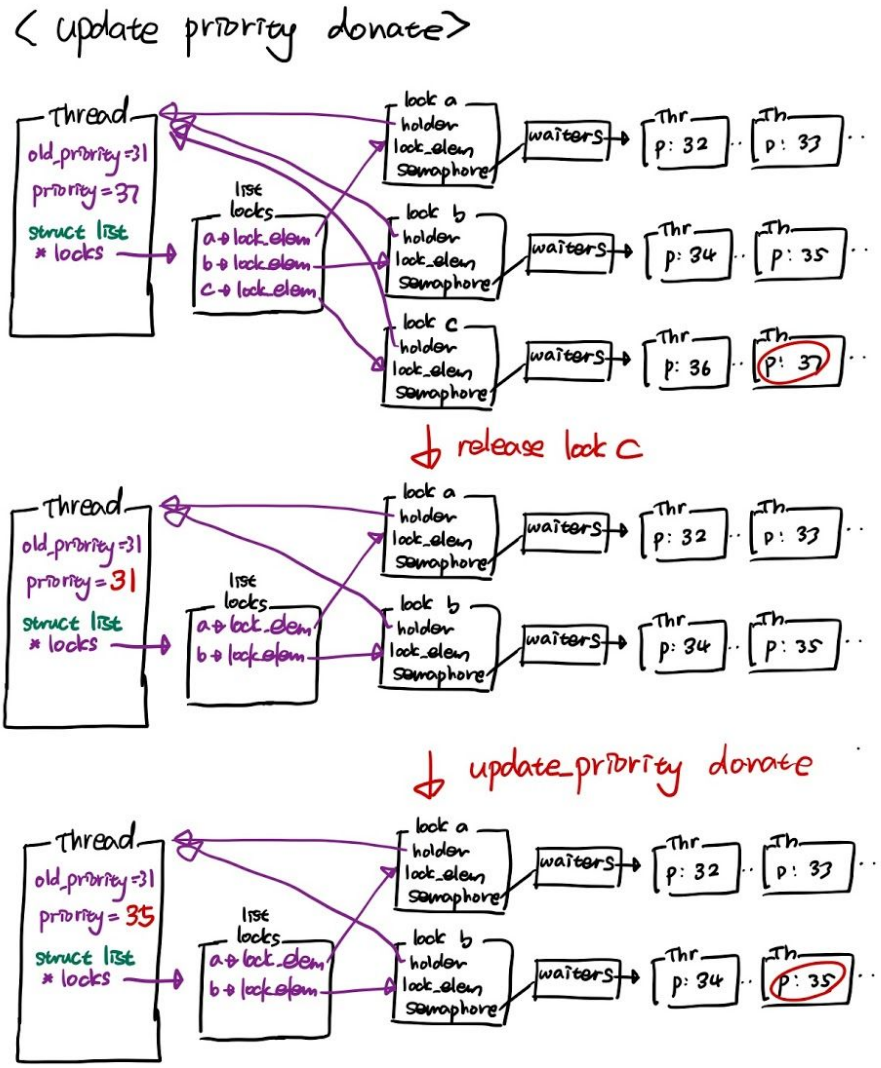

그림 1. update_priority_donate 자료구조와 알고리즘

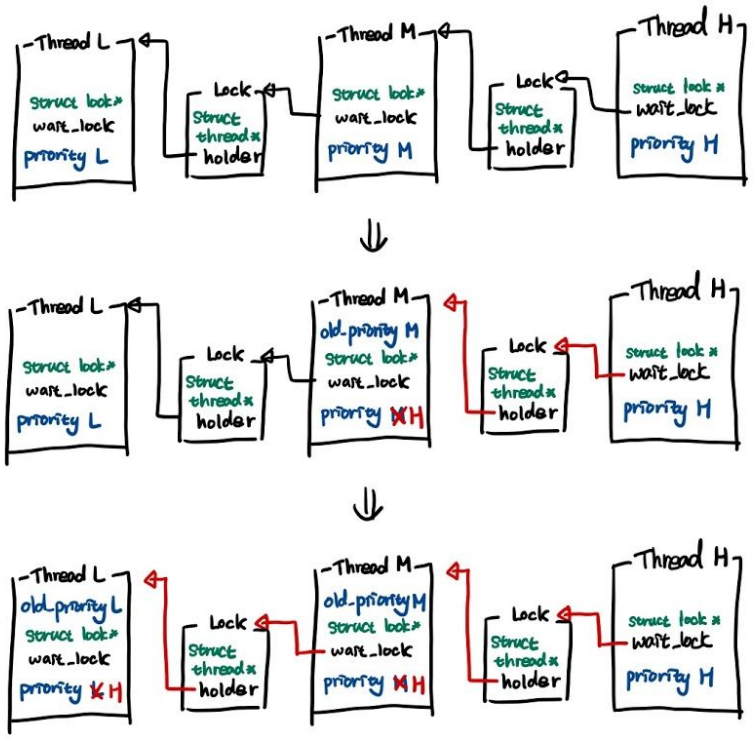

그림2. nested donation 자료구조

---- ALGORITHMS ----

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

먼저 priority가 새롭게 추가되거나 변경되는 순간(set_priority() or thread_create()), update_max_priority(){가장 높은 priority 값을 update해주는함수} 를 실행하여 가장 높은 priority를 가진 thread가 running thread가 되도록 하였다.

해당 lock을 기다리는 thread들은 lock->semaphore->waiters에 들어가게 된다. 이때, sema_down()에서 list_insert_ordered()를 통해 priority가 높은 thread가 가장 앞쪽에 들어가게 된다. 그리고 lock을 waiters의 가장 앞쪽 thread부터 가지게 하여 priority 순서대로 lock을 가지게 된다.

condition에서는 cond_signal에서 list_sort함수를 통해 waiters를 정렬한 후 실행했기 때문에 가장 높은 priority를 가진 thread가 가장 먼저 실행되게 된다.

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

lock_acquire(lock)함수를 실행하면, 해당 lock을 hold하고 있는 thread를 찾아 해당 Lock의 waiters와 함께 비교하여, 가장 큰 priority를 Lock의holder에게 donation한다. (thread_dontate_priority(lock)함수) 이때, 해당 thread는 자신이 기다리는 lock을 wait_lock에 저장한후, block된 상태로 기다리게 된다.

nested_donation인 경우에는 lock_acuqire()을 할때, 이미 그 lock에 hoder가 있고, 그 holder가 wait_lock이 있을 경우, nested_donation이라는 재귀함수를 사용해 (depth최대 8) 처음 Priority를(만약 사이에 더 높은 priority가 있다면 그 priority로) 계속 donation해주게 된다. 마찬가지로 donation을 해준 thread는 sema_down에서 block상태가 된다.

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

Lock_release(lock)가 실행되면 주어진 lock의 lock_elem을 list에서 제외한다. (list_remove(&lock->lock_elem))

그후 sema_up에서 waiters중 가장 priority가 높은 thread를 unblock한다.
그리고 현재 thread는 자신의 원래 Old_priority로 복구시킨 후에, update_priority_donation을 통해서 그 thread가 holder로 있는 모든 Lock의waiters를 탐색하여 가장 높은 priority를 가진 thread를 찾아 그 thread의 priority를 max_priority로 설정한다.
해당 thread가 아직도 lock을 가지고 있어 donation이 일어나면 그 max_priority로 다시 donation이 일어나게 된다.

---- 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
main thread의 priority = 31, lock a를 가짐
1. priority 41의 thread a가 lock a에 acquire를 시도하며 main thread에 priority donation을 함
2. thread_set_priority()를 통해 main thread의 priority를 21로 설정
이 때 1, 2번의 실행 순서에 따라 최종 priority와 old_priority의 값이 달라질 수 있음.

- implementation to avoid
thread_set_priority()에서 priority 와 old_priority를 바꿔준 후 update_priority_donation() 함수를 실행하여 정상적인 donation이 일어났는지 다시 한 번 확인함을 통해 실행 순서에 상관없이 동일한 값을 가지도록 하였다.

그리고 그후 update_max_priority()함수는 readylist에 있는 thread들을 정렬 + thread_yield()를 호출하여 바뀐 priority값들을 반영하게 다시running_thread를 정하게 된다.

또한 interrupt가 일어날 수 있는 부분에 intr_disable()을 사용해 critical section에서 interrupt가 일어나지 않도록 했기 때문에 race condition을 피할 수 있다.

---- RATIONALE ----

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

처음에는 priority순서대로 함수가 실행되도록, update_max_priority()함수로 가장 높은 priority가 생기면 새롭게 thread_yield()가 실행되도록 디자인하였다. 이때 list_insert_ordered()를 통해 ready_list에 priority 순으로 insert를 하고, list_begin()으로 가장 앞쪽의 thread를 먼저 실행되도록 한 것이다.

또한 priority 값의 변화가 있으면(thread_create(), thread_set_priority()) 새롭게 sort를 해줌의 필요성을 알게 되었다.

그후 donation과 synchronizaiton 문제를 해결하기 위해 위의 새로운 struct들을 추가해 주었다.
lock_acquire()에서 donation함수를 실행시켜주었고, lock_release(sema_up)안에서 donation을 해지시키는 개념에서 출발하였다.
waiter에 thread들을 추가할때 list_insert_ordered()함수 사용을 이전 priority 비교함수를 사용하여 (priority_bigger_than())insert해주었다.

condtion은 위의 세마포어와 락 해결관점으로 디자인을 하였고, sorting시 문제가 발생하여 list_sort()함수를 추가해주었다.

old_priority의 경우, 원래 이전 priority로 설정해주었으나 같은 lock을 여러 쓰레드가 acquire했을때 문제가 발생하여
가장처음의 init_priority로 설정해주었다.
또한 locks list를 추가하여, lock_release()를 할때 어떠한 lock을 release해주어야 하는지 결정하였다.
wait_lock의 경우, nested_donation을 저격하여 한것인데 lock의 holder가 다른 lock을 acquire할때 그 lock을 쉽게 찾기 위해서 추가 해주었다.

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?

문제자체는 어렵지 않았으나, 처음 핀토스를 접한 과정에서 이해하는데 시간이 오래 걸렸다.

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

솔직히 donation같은 경우에는 과제를 하면서 새롭게 알게된내용은 없었고, 타이머 인터럽트나 특정 테스트 케이스에서 고생하는 동안 핀토스 시스템이 어떻게 동작하고 어떻게 실행되는지를 깨달았다.

>> 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?

괜찮았다. 테스트케이스를 하나씩 보면서 정복해나가는 느낌으로 가이드 하면 좋을 것 같다.
그리고 각 테스트 케이스가 어떠한 문제를 해결하면 통과되는지(ex, scheduling) 안내해주면 좋을 것같다.

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

slack을 통해 충분히 빠르게 답해주었다.

>> Any other comments?

NO