---- TEAM ----

**>> Team name.**

**>> Fill in the names, email addresses and contributions of your team members.**

Jae Hyun Lim <ggariongdev@gmail.com> (60%)

FirstName LastName <email@domain.example> (40%)

**>> Specify how many tokens your team will use.**

none

---- PRELIMINARIES ----

**>> If you have any preliminary comments on your submission, notes for the TAs, 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.**

<http://www.embedded.com/electronics-blogs/beginner-s-corner/4023970/Introduction-to-Interrupt-Debugging> (race condition 검색)

<http://yhoon2.blogspot.kr/2013/01/pintos-project-1-threads.html> (project1의 전체적인 그림 파악)

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.**

src / threads/thread.h

-struct thread에서 int64\_t sleep\_ticks 추가. timer\_sleep()를 호출한 쓰레드를 언제 깨울 수 있는지 알 수 있음.

src/device/timer.c

-static struct list sleeping\_list 선언. timer\_sleep()를 호출하는 쓰레드를 저장.

---- ALGORITHMS ----

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

While문을 돌려 깨어날 시간인지 아닌지 계속 감시하는 방법대신, timer\_sleep()를 호출한 현 스레드(thread\_current)의 sleep\_ticks에 ‘현재의 시각 + 잠들어 있어야 할 시간’을 값으로 준다. 그 후 이를 list\_insert\_ordered()로 sleeping\_list에 가장 먼저 깨어나야 할 순으로 저장한다. ready state로 방치되는 yield와 달리 쓰레드가 작동하지 않도록 block을 건다. block을 안전하게 걸기 위해 interrupt는 잠시 해제된다. block이 걸려 잠든 스레드는 그것의 sleep\_ticks만큼 시간이 지났다면, 후에 timer\_interrput가 불릴 때에서야 sleeping\_list에 제외되고 unblock 될 것이다.

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

sleeping\_list에서 잠든 스레드들이 가장 먼저 깨어나야 할 순으로 정렬되어 있기에, 앞에 있는 스레드들이 깨어날 때가 됐는지 확인하고, 아니면 탐색을 중지한다. 즉, 리스트에 있는 모든 스레드들을 검사할 필요 없이 선두의 것만 확인하면 된다.

---- SYNCHRONIZATION ----

**>> A4: How are race conditions avoided when multiple threads call timer\_sleep() simultaneously?**

race condition은 본질적으로 interrupt에 의해 발생할 수 있는 현상이다. 그렇기에 timer\_sleep() 내에 잠시 interrupt를 해제한 사이에 A2의 작업을 수행토록 했다.

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

A4와 같은 이유로 timer\_sleep()에서도 잠시 interrupt를 해제한 사이에 작업을 수행토록 했다.

---- RATIONALE ----

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

timer\_sleep에서 while문을 없애 timer\_interrupt가 호출될 때에만 스레드가 깨어나도록 역할을 분담했다. 그렇기에 처음에 timer\_interrupt에서도 sleeping\_list를 탐색할 때 if문으로만 구현했었다. 그러나 이는 alarm-simultaneous에 fail을 불러왔고, alarm-simultaneous의 코드를 분석한 결과 동일한 sleep\_ticks를 가진 스레드가 다수 있을 수 있다는 점을 감안해야 한다는 것을 깨달았다. 그래서 바깥의 if문을 while문으로 바꾸고, 탐색 중 깨어날 시간이 안된 스레드가 나오면 break 하도록 하였다.

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.**

src /threads/thread.h

-struct thread에서 int original\_locked\_priority 선언. lock을 쥐고 있는 스레드의 priority가 변할 경우에 대비하여 스레드 자체에 lock 이후 돌려받아야 할 priority 값 지정.

- struct thread에서 struct thread \*lock\_holder\_thread 선언. donate-chain의 경우 같이 여러 개의 lock으로 스레드들이 얽혔을 때 donate를 받아야 할 스레드들을 찾기 위해 지정.

src /threads/synch.h

-struct lock에서 int original\_priority 선언. lock을 호출한 스레드가 donation 이후 돌려받아야 할 값을 lock에 저장.

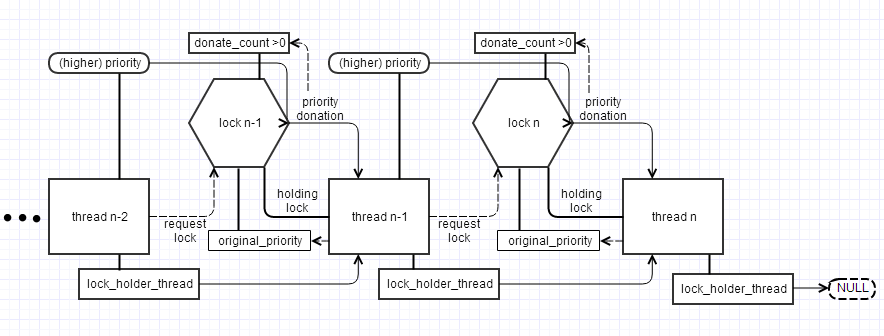
-struct lock에서 int donate\_count 선언. lock을 호출한 스레드가 donation을 받은 여부를 확인하기 위한 변수. 0이면 donation을 받은 적이 없다.

src /threads/synch.c

-struct semaphore\_elem에서 int priority 추가. condition에 의한 동기화에서 priority 순으로 정렬시키기 위해 정렬 기준이 될 변수 추가.

**>> B2: Explain the data structure used to track priority donation. Draw a diagram in a case of nested donation.**

이미 스레드 i가 호출한 lock을 priority가 더 큰 다른 스레드 i+1가 요구할 때 priority donation이 발생한다. 이때 lock이 걸려있는 스레드 i의 priority는 해당 lock의 original\_priority에 저장되고, lock의 donate\_count는 0보다 커지며, 스레드 i는 더 높은 priority로 대체된다. priority donation 이후 priority를 기부한 스레드 i+1에는 자신이 기다리고 있는 lock의 holder인 스레드 i를 스레드 내의 lock\_holder\_thread에 저장한다. 만약 기부를 받은 스레드 i 또한 다른 lock을 기다리고 있으면 (스레드 i의 lock\_holder\_thread가 NULL이 아니면), priority donation은 recursive로 작동하게 된다. 이후 lock이 풀릴 때마다 해당 lock에 저장되어 있던 original\_priority가 lock을 release한 스레드에게 돌아가고, donate\_count는 0으로 초기화 된다.



---- ALGORITHMS ----

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

스레드가 semaphore와 condition을 호출할 때마다 waiters라는 대기리스트에 등록된다. 이때 이 리스트를 스레드들의 priority 순으로 정렬시키면 된다. 이는 다음 run을 기다리는 ready 상태의 스레드들(ready\_list)에게도 적용되는 방식이다.

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

이미 스레드 i가 호출한 lock을 priority가 더 큰 다른 스레드 i+1가 요구할 때 priority donation이 발생한다. 먼저 이 donation이 처음인지 확인하고, 처음이면 lock이 걸려있는 스레드 i의 priority는 해당 lock의 original\_priority에 저장된다. priority를 기부 받을 필요가 있다는 것이 확인되었으므로, 스레드 i는 더 높은 priority로 대체된다. priority donation 이후 priority를 기부한 스레드 i+1는 자신이 기다리고 있는 lock의 holder인 스레드 i를 따로 저장한다. 만약 기부를 받은 스레드 i 또한 다른 lock을 기다리고 있으면, priority donation은 스레드 i가 기다리는 lock의 holder에게로 recursive하게 한다. 마지막으로 해당 lock에 기부가 이루어졌다는 사실을 donate\_count로 저장한다.

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

priority donation 여부를 lock의 donate\_count 값을 통해서 확인한다. priority donation 이후 lock이 풀릴 때마다 해당 lock에 저장되어 있던 original\_priority를 통해 lock을 release한 스레드는 자신의 원래 priority로 돌아가고, donate\_count, original\_priority는 0으로 초기화 된다. 하지만 lock 도중에 스레드의 priority를 수정하는 명령이 진행되었을 수도 있다. 이에 대비하기 위해 lock의 original\_priority 외에 스레드 자체적으로 자신이 갖고 있던 priority를 original\_locked\_priority에 저장한다. lock 도중의 모든 priority 편집은 이 변수에 적용되며, lock을 release할 때 lock과 스레드가 갖고 있는 기존 priority 값을 비교하여 그에 맞게 값을 돌려주고 original\_locked\_priority 를 0으로 초기화한다.

---- 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?**

다른 동작을 수행하고 있는 스레드의 priority를 여러 곳에서 수정하려고 하면 문제가 생기기 마련이다. 예를 들어 lock이 걸려있는 스레드의 priority를 수정해버리면 lock의 우선순위 배정이 엉망이 되고, 해당 스레드가 돌려받아야 할 priority가 수정이 전혀 안 되는 경우가 발생할 수 있다. 이에 대비해 위의 B5에 언급한 것처럼 lock의 original\_priority 외에 스레드 자체적으로 자신이 갖고 있던 priority를 original\_locked\_priority에 저장하도록 했다. lock 도중의 모든 priority 편집은 이 변수에 적용되며, lock을 release할 때 lock과 스레드가 갖고 있는 기존 priority 값을 비교하여 필요하면 수정이 된 priority 값을 돌려줄 수 있다.

---- RATIONALE ----

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

대부분의 디자인은 디버깅 과정에 빚어진 것이다. test 소스와 기존의 함수들을 참고하여 코딩을 하고, 오류가 발생하면 그것을 방지하도록 기존의 것을 수정해온 결과이다. 중도에 포기한 디자인은 priority-donate-lower에 대비하는 코드를 구현할 때인데, 초기에는 스레드 자체에 스레드가 기다리고 있는 lock을 저장하려 하였다. set\_priority() 내에서 스레드에 저장된 lock을 통해 해당 lock의 holder, original\_priority 등에 직접 접근하기 위해서였다. 그러나 lock\_acquire에서 해당 lock을 스레드에 저장하려 해도 그 값이 제대로 넘어가지도 않고 코드 또한 점점 복잡해졌다. 때문에 스레드에서 직접 lock의 변수에 접근하기 보다, 그 변수를 추가로 설정한 스레드의 변수에 저장시키도록 하였다. 이렇게 하니 값도 안정적으로 잘 넘어오게 되고, 값들을 편집하기도 훨씬 쉬워졌다.

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 two problems in it, too easy or too hard? Did it take too long or too little time?**

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

**>> 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?**

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

**>> Any other comments?**