+--------------------+

| CS 330 |

| PROJECT 1: THREADS |

| DESIGN DOCUMENT |

+--------------------+

---- GROUP ----

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

김건호(Kunho Kim) <kaist984@kaist.ac.kr>

김철환(Cheolhwan Kim) <maxwell98@kaist.ac.kr>

---- PRELIMINARIES ----

>> 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.

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.

In thread.h --------------------------------------------------------

struct thread{

...

int64\_t wake\_ticks; /\* Time to wake up. \*/

...

}

Thread가 일어나야 할 시간을 저장하는 변수입니다.

In thread.c --------------------------------------------------------

int64\_t ticks\_to\_wake; /\* # of timer ticks which represents the fastest time to awake thread. \*/

struct list sleep\_list; /\* List of sleeping processes. \*/

현재 수면중인 thread를 관리하는 sleep\_list 입니다. sleep\_list안에서 가장 빨리 일어나야할 thread가 일어날 시간을 ticks\_to\_wake에 저장합니다.

---- ALGORITHMS ----

>> A2: Briefly describe what happens in a call to timer\_sleep(),

>> including the effects of the timer interrupt handler.

timer\_ticks()를 이용해 timer interrupt에 영향을 받지 않고 현재 시각을 불러옵니다. 그 후 parameter로 얻은 시간간격을 더해 깨워야 할 thread의 시각을 계산 후 thread\_sleep()함수의 parameter로 넣고, thread\_sleep() 함수를 호출합니다. thread\_sleep()함수 역시 method 진행중에 timer interrupt에 의해 시간 정보에 영향을 받을 수 있으므로 interrupt를 막아 두고 진행합니다. Thread 구조체에 추가한 wake\_ticks를 갱신하고, 자체 제작한 함수 update\_ticks\_to\_wake(int64\_t ticks)를 통해 수면 중인 thread들 중 가장 먼저 일어나야 할 thread의 시각을 갱신합니다. 이후 sleep list에 thread를 추가하고 block상태로 만든 뒤 다시 인터럽트를 받을 수 있게 변경하고 함수를 마칩니다.

>> A3: What steps are taken to minimize the amount of time spent in

>> the timer interrupt handler?

기존 timer\_sleep()에서 while loop의 yield를 통해 깨어나기전까지 계속 time\_elapsed()로 시간을 불러오던 방식을 없앴습니다. 대신, 깨어날 시간을 저장해주고 아예 block시켜서 timer\_interrupt() 외에 ticks를 불러오는(timer\_ticks() call) 횟수를 최소화 시켜줬습니다.

---- SYNCHRONIZATION ----

>> A4: How are race conditions avoided when multiple threads call

>> timer\_sleep() simultaneously?

multiple threads라고 하더라도 동시에 실행하는 것처럼 보이는 것이지 결국 cpu에는 하나의 thread만이 실행되기 때문에, 고려하지 않았습니다. 하지만 문제가 정말로 여러 개의 thread가 동시에 timer\_sleep()을 부르는 경우를 물어보는 것이라면, 그때에도 이 방법은 제대로 적용될 것 같습니다. 결국 핵심은 timer interrupt handler에 의해 원하지 않았던 ticks 값이 저장되는 문제인데, 저희가 만든 timer sleep과 thread sleep에서 ticks를 참조하는 건 timer\_ticks()가 불리는 한 번이기 때문에 문제가 없을 것 같습니다.

>> A5: How are race conditions avoided when a timer interrupt occurs

>> during a call to timer\_sleep()?

A4와 같은 맥락입니다. 실시간으로 변하는 global 값을 사용하지 않고, 함수 자체의 parameter를

활용했기 때문에, 문제는 일어나지 않을 것입니다.

---- RATIONALE ----

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

>> another design you considered?

저희가 첫번째로 고려한것은 아무래도 기존의 문제로 지적되어 있던 busy waiting 기법을 피하는 것이었습니다. thread가 cpu를 선점한 후 thread 내부 함수로 timer\_sleep이 실행되고, 모종의 이유로 yield가 실패할 것을 고려해 yield를 while문 안에 넣은 것으로 인해 cpu에 막대한 손실을 낳게 되는 것인데, 이를 피하기 위해 지정된 시간까지 건드리지않고 지정된 시간에만 건드릴 수 있는 방식이 필요했습니다. 따라서 재우는 것과 깨우는 방식을 나눠야 했고, 재우는 방식은 timer\_sleep에서 조금 변형하면 됐지만 깨우는 것이 문제였습니다. 언제 깨울 지에 대한 문제는 ticks\_to\_wake 변수를 만들어 해결했지만 어떻게 깨울 지는 아직 해결하지 못했습니다. 깨울 시간이 되었을 때, 이미 다른 thread가 cpu를 선점하고 있고, 이 thread가 자고있는 thread를 깨워주길 기대할 순 없었기 때문입니다. 따라서 선점 중인 cpu에 관계없이 깨울 수 있는 방식이 필요했고, 그것이 timer interrupt 함수였습니다. 선점 중인 thread에 상관없이 맞춰진 시간마다 자동적으로 발생했기 때문입니다. 따라서 sleep\_list에서 일어날 시간이 된 모든 thread들을 깨우는 thread\_awake()함수를 만들어 time\_interrupt()에 집어넣었습니다.

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.

In thread.h --------------------------------------------------------

struct thread{

...

bool orizin; /\* If origin priority == 0, true. \*/

int origin\_priority; /\* Save original priority of current thread to get donated priority.

0 means current thread doesn't get donated priority yet. \*/

struct list donated; /\* Save all threads who donate his own priority to this thread. \*/

struct list\_elem donate\_elem; /\* Put this in donated list. \*/

struct lock\* lock\_wait\_for; /\* Show which lock this thread wait for. \*/

...

}

bool orizin은 thread의 기존 우선순위가 0이라면 true, 아니라면 false를 갖습니다.

int origin\_priority는 donate 받기 전 thread의 우선순위를 저장합니다. 아직 donate를 받지 않았다면 0을 값으로 가집니다.

struct list donated는 thread가 lock의 holder로 존재할 때, 이 holder에게 우선순위를 donate를 해준 다른 모든 thread들의 list입니다.

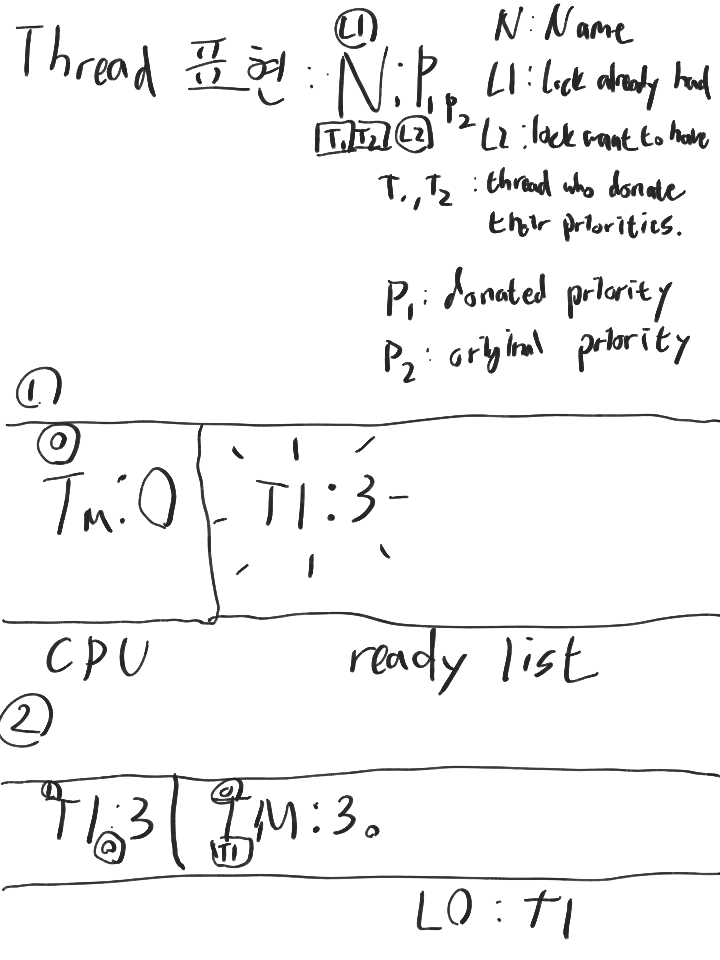
struct list\_elem donate\_elem는 donated에 thread 구조체를 추가하기 위한 원소입니다.

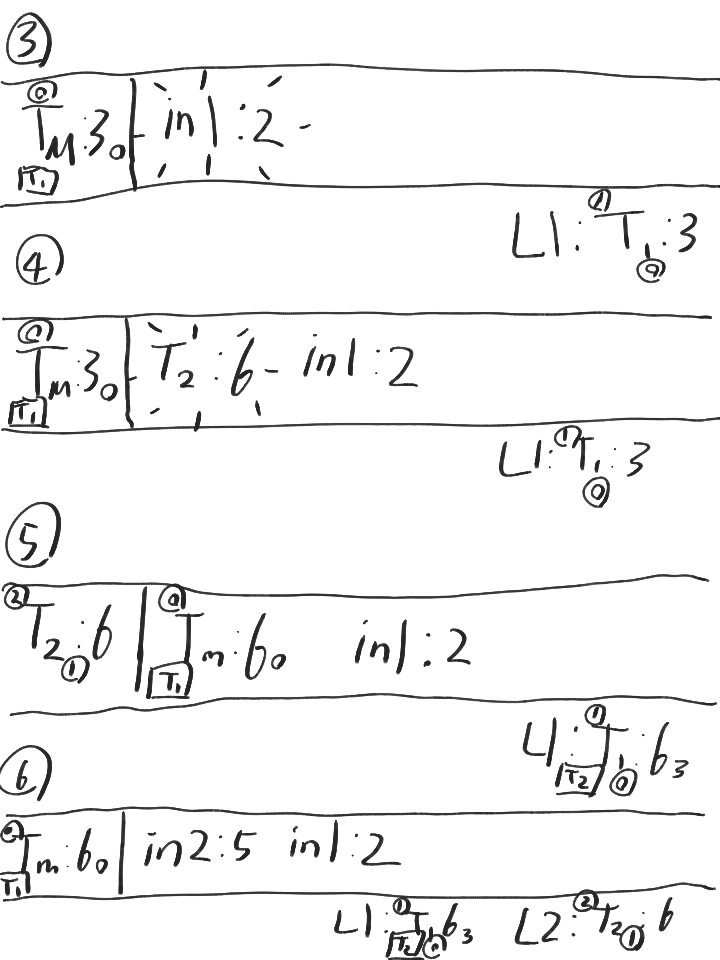
struct lock\* lock\_wait\_for는 thread가 acquire를 요청한 lock을 저장하는 변수입니다.

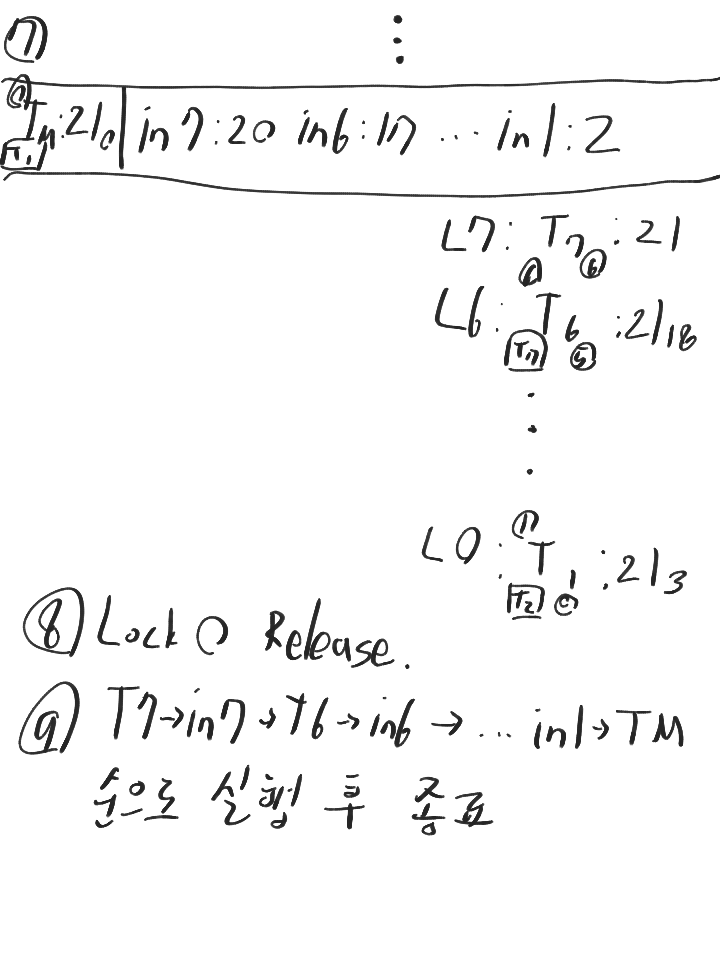
>> B2: Explain the data structure used to track priority donation.

>> Use ASCII art to diagram a nested donation. (Alternately, submit a

>> .png file.)







---- ALGORITHMS ----

>> B3: How do you ensure that the highest priority thread waiting for

>> a lock, semaphore, or condition variable wakes up first?

1. case lock & semaphore

lock, semaphore을 선점하고 있는 thread보다 높은 우선순위의 thread가 ready list에 진입하게 되면, 우선순위 비교를 통해 thread\_yield()가 실행되고, cpu에 들어오게 됩니다. 이때, cpu에서 실행중인 thread가 lock, semaphore의 acquire를 시도하면, 자신의 우선순위를 이미 lock, semaphore을 소유중인 thread에게 donate하고 sema\_down()이후, thread\_block()되어 semaphore의 waiters에 우선순위에 따라 들어가게 됩니다. 이에 추가적으로 더 높은 우선순위를 가진 thread가 ready list에 들어오면, 앞에서 말한 방법과 동일하게 진행됩니다. 이러한 과정이 끝나게 되면, 가장 처음에 lock이나 semaphore를 소유한 thread가 가장 높은 우선순위를 가진 채 cpu에서 실행 중일 것입니다. 이후 lock\_release()나 sema\_up()이 호출되면, lock이나 sema를 소유하고 있던 thread는 자신의 기존 우선순위로 돌아가게 되고, waiters의 가장 앞에 있는(우선순위대로 정렬해서 넣었으므로 우선순위가 가장 높은) thread가 unblock되게 됩니다. ready list에 있던 unblock된 thread의 우선순위는 현재 cpu에 있는 thread의 우선순위보다 당연히 높을 것입니다. 그러므로 sema\_up()에서 thread\_yield()가 호출되어 cpu를 선점하게 됩니다. 위의 과정들을 통해 가장 높은 우선순위를 가진 thread가 가장 먼저 깨어난다는 사실을 알 수 있습니다.

2. case condition variable

condition은 semaphore를 관리하는 list인 waiters를 갖고 있습니다. 현재 실행중인 thread A가 소유하고 있는 lock 때문에 donate를 받은 상황이라고 가정해봅시다. 하지만, 현재 실행중인 thread와 관련 없는 lock을 필요로 하는 가장 높은 우선순위의 thread B가 생성되었다면, thread A는 B와 관련없이 lock release를 실행해 thread A에게 donate 해준 thread중에 가장 우선순위가 높은 thread가 cpu를 선점하도록 만들 것이고, thread B는 기다리게 됩니다. 이때, 현재 사용중인 모든 semaphore들을 관리할 수 있는 condition이 이용되는 것입니다. condition은 lock기법과 상관없이 여러 semaphore을 기다리는 모든 thread들 중에 우선순위가 가장 높은 값의 thread B를 실행하도록 도와줍니다.

>> B4: Describe the sequence of events when a call to lock\_acquire()

>> causes a priority donation. How is nested donation handled?

lock을 선점하고 있는 thread보다 높은 우선순위의 thread가 ready list에 진입하게 되면, 우선순위 비교를 통해 thread\_yield()가 실행되고, cpu에 들어오게 됩니다. 이때, cpu에서 실행중인 thread가 lock의 acquire를 시도하면, lock을 소유한 holder thread가 가진 list(donated)가 비어 있다면, 자신의 우선순위를 이미 lock을 소유중인 thread에게 donate하고, lock을 소유한 holder thread의 origin priority가 0에서 기존 자신의 우선순위로 바뀌게 됩니다. 만약 holder thread의 기존 우선순위가 0이었다면, holder thread가 가진 boolean orizin이 false에서 true로 바뀝니다. 그리고 lock을 소유한 holder thread가 갖고있는 list(donated)가 갖고 있는 원소들과 우선순위를 비교하여 우선순위에 따라 list(donated)에 저장합니다. 그리고 cpu에서 실행중인 thread가 어떤 lock을 기다리고 있는지를 lock\_wait\_for 변수에 저장합니다. nested donation 경우, lock을 소유한 holder thread가 기다리고 있는 lock이 있으면, 기다리고 있는 lock을 소유한 holder thread의 우선순위를 현재 cpu를 선점중인 thread의 우선순위로 바꿔줍니다. 이후, sema\_down()이 호출되고, thread\_block()이 호출되어 semaphore의 waiters에 current thread가 우선순위에 따라 들어가게 됩니다. 이에 추가적으로 더 높은 우선순위를 가진 thread가 ready list에 들어오면, 앞에서 말한 방법과 동일하게 진행됩니다. 이러한 과정이 끝나게 되면, 가장 처음에 lock을 소유한 thread가 가장 높은 우선순위를 가진 채 cpu에서 실행 중일 것입니다.

>> B5: Describe the sequence of events when lock\_release() is called

>> on a lock that a higher-priority thread is waiting for.

앞서 acquire 과정에서 설명했듯이 현재 lock을 소유하고있는 holder thread(A)보다 높은 우선순위를 보유하고 있는 thread(B)가 lock\_acquire()를 호출해 lock을 요청했다면 A는 B와 같은 우선순위를 가지고 있고, A의 원래 우선순위는 origin\_priority에, A의 기존 우선순위에 따라 orizin(기존 우선순위가 0인가?)에 해당하는 T/F정보가 저장, A의 list, donated에 B의 주소값이 저장됨과 동시에 B의 lock\_wait\_for에 B가 acquire을 요청하는 해당 lock의 주소값이 저장되어 있을 것입니다. 이때, lock을 선점중인 A가 해당 lock을 release한다면, 첫번째로 A의 list donated에서 해당 lock을 소유하길 원했던 모든 thread들을 없애줍니다. 더이상 리스트에 보관할 필요가 없기 때문입니다. 그 다음, donated에서 아직 priority를 donate해준 다른 thread들이 있었다면(다른 lock을 원하는 thread), 그 thread들 중 가장 높은 우선순위를 보유한 thread의 우선순위로 A의 우선순위를 갱신해줍니다. 만약 A의 donated가 비었다면, 더이상 우선순위를 donate해주는 다른 thread가 없다는

의미이므로 다시 원래의 우선순위로 돌아갑니다. origin\_priority에 저장되어 있던 기존 우선순위를

입혀주고, 0으로 만들어줍니다. donate와 관련한 변수 설정이 모두 완료되면 lock의 holder를 null로 설정 후 sema\_up()을 실행합니다. sema\_up을 통해 sema의 value를 1 높여주고, block 되어있던 thread들 중 우선순위가 가장 높은 B를 waiters list에서 pop시켜줍니다. 그 후, ready\_list가 비어 있는지 확인한 후, (모종의 이유로 unblock이 실패했을 경우) lock을 원했던 B가 cpu를 다시 선점할 수 있도록 yield를 진행합니다. 이후 lock\_release()는 종료됩니다.

---- 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?

현재 저희의 thread\_set\_priority()코드는 current thread가 우선순위를 donate 받은 상태이면 current thread의 origin priority만 바꿔줍니다. 만약 donate 받지 않았다면 current thread의 priority를 바꿔준 후, ready list의 가장 앞에 있는(우선순위가 가장 높은) thread와 우선순위를 비교하여 list안의 thread가 우선순위가 더 크다면 thread\_yield()를 실행해줍니다. 간단한 로직이지만, 이 과정에서 interrupt로 인해 cpu 양보가 일어나는 상황이라면 레지스터의 값을 침해받기 때문에 원하는 결과를 얻을 수 없을 것입니다. 그것을 막기 위해 우리는 interrupt를 막거나 lock 기법을 이용할 수 있는데, 전자는 처리속도를 낮추기 때문에 lock을 이용하는 것이 더 효과적인 방법입니다. 간단하게 생각하자면 thread.c 에 thread\_set\_priority()를 위한 global변수인 struct lock lock\_for\_TSP를 추가해줍니다. thread\_set\_priority()의 시작 부분에 lock\_acquire(lock\_for\_TSP)와 함수가 끝나기 직전에 lock\_release(lock\_for\_TSP)를 추가합니다. 두 함수 사이에 기존의 thread\_set\_priority() method를 진행하게 되면 lock을 통해 potential race를 해결할 수 있을 것이라 생각합니다.

---- RATIONALE ----

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

>> another design you considered?

기존의 design은 lock 구조체에 origin priority를 저장하고, thread 구조체에 lock을 소유하고 있는 개수 num\_lock을 만들어서 donation문제를 해결하려 했습니다. 하지만 이 design으로는 한 thread가 여러 개의 lock을 소유하고 있고, 순차적으로 lock들이 release될때 남아있는 후원 우선순위 가운데에서 가장 높은 우선순위로 줄어들게 할 수 없었습니다. 이 문제를 해결하기 위해 priority를 소유하고 있는 thread의 주소값을 직접 받아와야 했고, 그 결과 thread 구조체 안에 또다른 list인 donated를 제작했습니다. 이후 release과정 중에 donated에서 thread들을 효과적으로 제거하기 위해 그들이 원하는 lock의 주소값을 원하는 lock\_wait\_for을 추가했습니다. 이 방법으로 대부분의 test case를 통과했지만, priority-donate-chain 과정을 통과할 수 없었습니다. gdb를 통해 저희의 기능적 오류를 확인한 결과, 시도 때도 없이 실행되는 lock\_release로 원하지않던 우선순위의 복구를 막기위해 추가했던 조건문 if(orgin\_priority != 0) 때문에 원래의 우선순위가 0이었던 thread는 원래 우선순위로 복귀할 수 없던 상황이었습니다. 마지막으로 이것을 해결하기 위해 원래 thread의 우선순위가 0이었는지를 확인하는 bool orizin(origin+zero)를 추가하여 문제를 해결했습니다. 이번 priority schduling을 design하면서 저희가 큰 그림을 잡지않고 급하게 코딩에 착수함으로써 이런 번거로운 일이 발생했고, 앞으로 과제를 실행하기 전 기존의 코드와 실행 방식을 정확히 이해하고 그 다음에 저희의 코드를 design 해야겠다는 교훈을 얻었습니다.

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?

솔직히 말해서 많이 버거웠습니다. 평일에는 매일 12시부터 3~4시까지, 주말에는 거의 하루종일 zoom 화상회의를 통해 직접 얘기를 주고받으며 아이디어 design과 코딩을 했습니다.

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?

project1을 통해 os에서 사용되는 기본적인 구조체를 이해할 수 있었고, cpu 선점과 interrupt의 작동 원리, 그 기능들을 새롭게 알게 되었습니다.

>> 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?

test case를 미리 확인하고 design을 시작하면 좋을 것 같습니다.

guidance to be misleading으로는 과제 배포 초반에 make grade를 실행했을 때, 채점 요약 결과가 제대로 출력되지 않았던 점을 발견했습니다.

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

gdb를 활용하는 방법과, 활용했을 때 어떤 이점이 있는지 예시와 함께 설명해주면 좋을 것 같습니다.

Design document 양식을 txt가 아닌 docx로 주셨으면 합니다. 중간에 있는 줄바꿈들이 docx파일에서는 가독성을 떨어뜨리는 것 같습니다.

>> Any other comments?

없습니다.