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

---- GROUP ----

김우진 <hyung1721@kaist.ac.kr>
김준범 <dungeon12345@kaist.ac.kr>

---- PRELIMINARIES ----

# of tokens to use: 0

Contribution
김우진: 50%
김준범: 50%

>> 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.

Alarm clock implementation을 위해서 thread.c에 다음의 global variable들을 추가하였다.

```
/* Thread's tick which is needed to wake first of all */
static int64_t closest_tick = INT64_MAX;

/* List of sleep threads */
static struct list sleep_list;
```

closest_tick은 sleep된 thread들 중에서 가장 먼저 깨어나야 하는 thread의 tick을 저장한다. sleep_list는 sleep된 thread들이 담긴 list이다.

thread 구조체 내에 다음의 변수를 추가하여 깨어나야 하는 tick을 저장한다.

```
        int64_t tick_to_wake;  /* Ticks at which a thread should wake. */
```

---- ALGORITHMS ----

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

timer_sleep()이 호출되면 현재 thread를 전달받은 인자만큼의 tick동안 thread를 재운다.
기존 PintOS는 busy-waiting 방식을 통해서 tick만큼 시간이 흘렀는지 검사한다. 우리의 implementation에서는 다음의 순서로 작업이 진행된다.

1. sleep_thread_with_ticks(start, ticks)를 call한다.
2. Interruption을 disable한다.
3. 현재 thread의 tick_to_wake 값을 (start + ticks)로 변경한다.
4. 3에서의 값을 closest_tick과 비교하여 필요하면 갱신한다.
5. sleep_list에 thread를 저장한다.
6. interruption을 enable한다.

timer_sleep() call을 통해 재워진 thread들은 timer interrupt handler인 timer_interrupt()에 의해 깨어나야 하는지 검사 받게 된다. 이 handler는 다음의 순서로 작업을 진행한다.

1. PintOS 부팅 이후의 ticks보다 closest_tick가 작다면 wakeup_thread()를 call한다.
2. wakeup_thread()에서는 closest_tick보다 작은 tick_to_wake를 가지는 thread를
unblock하고 sleep_list에서 제거한다.
3. sleep_list를 순회하며 closest_tick를 갱신한다.

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

timer interrupt handler는 매번 call될때마다 tick를 하나씩 올리며 깨울 thread가 있는지 검사해야 한다. 하지만 매번 검사한다면 wakeup_thread()를 호출하여 sleep_list를 무의미하게 순회하는 경우가 존재한다.
우리의 implementation에서는 closest_tick이 ticks보다 작을 때에만 깨울 thread가 있다는 것이 확실하므로, 이 경우에 wakeup_thread()를 call하여 불필요한 시간이 소요되지 않도록 하였다.

---- SYNCHRONIZATION ----

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

여러 thread가 timer_sleep()를 call하여도 sleep_list에 access하는 작업은 interruption이 disable된 상태에서 진행되기 때문에 race conditions을 방지할

수 있다.

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

timer_sleep()이 call되면 우리가 implement한 sleep_thread_with_ticks()가 호출된다.
이 함수 내에서는 disabling interrupt를 통해서 interruption을 막는다. 따라서 timer interrupt에 의한 race conditions을 방지할 수 있다.

---- RATIONALE ----

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

우리의 design를 요약하면 다음과 같다.

1. timer_sleep()이 인자로 받은 tick만큼 현재 thread를 sleep시킨다.
- 여기서 sleep은 blocking + pushing into sleep_list로 정의한다.
2. timer interrupt handler는 closest_tick과 현재 tick을 비교하며 적절히 wakeup_thread()를 호출한다.
- 여기서 wakeup은 unblocking + popping from sleep_list로 정의한다.

기존 PintOS의 busy-waiting이 아니라, 실제로 특정 tick까지 thread를 block해두고 sleep_list를 통해서 따로 관리하는 것은 reasonable한 디자인이라고 생각한다. timer interrupt가 현재 tick과 closest_tick을 비교하면서 필요한 경우에만 wakeup_thread()를 부르는 것은 시스템 resource를 절약할 수 있는 부분이라고 생각한다.

다른 design를 고려해보지는 못했다. 기존 PintOS와 비교한다면, busy-waiting 문제를 해결했고 sleep과 wakeup이라는 작업을 명확히 정의함으로써 디자인을 간결하게 구성하였다.

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.

다음은 수정된 thread 구조체이다.

```
struct thread
{
        /* Owned by thread.c. */
        tid_t tid;                  /* Thread identifier. */
```

```
		enum thread_status status;   /* Thread state. */
		char  name[16];                    /* Name (for debugging
purposes). */
		uint8_t *stack;              /* Saved stack pointer. */
		int priority;                /* Priority. */
		struct list_elem allelem;    /* List element for all threads
list. */

		/* Shared between thread.c and synch.c. */
		struct list_elem elem;       /* List element. */
		struct list_elem elem_d;      /* List element for donation
list. */

		int64_t tick_to_wake;          /* Ticks at which a thread
should wake. */

		/* Priority which holds original value. */
		int old_priority;
		/* Lock on which this thread is waiting. */
		struct lock *wait_lock;
		/* List of threads that donated to this thread. */
		struct list donation_threads;

		#ifdef USERPROG
		/* Owned by userprog/process.c. */
		uint32_t *pagedir;           /* Page directory. */
		#endif

		/* Owned by thread.c. */
		unsigned magic;              /* Detects stack overflow. */
	};
```

old_priority에는 모든 lock을 release한 뒤 본인의 원래 priority를 돌아가기 위해 값을 저장한다.
wait_lock은 nested donation을 implement할 때 현재 thread가 어떤 lock을 wait 하고 있는지 알기 위해 추가하였다.
donation_threads는 현재 thread의 donation에 관여한 모든 thread들이 담긴다. lock release를 한 후 priority를 변경할 때 사용한다.

다음은 수정된 semaphore_elem 구조체이다.

```
	struct semaphore_elem
	{
		struct list_elem elem;        /* List element. */
		struct semaphore semaphore;   /* This semaphore. */
		int priority;                 /* Priority of thread. */
	};
```

기존 구조체에서 priority 멤버를 추가하였다.
cond_wait()에서 semaphore_elem을 insert할 때, cond_wait()을 호출한 thread를 priority 순으로 정렬해 넣기 위해 이 멤버 변수에 현재 thread의 priority 값을 저장한다.

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

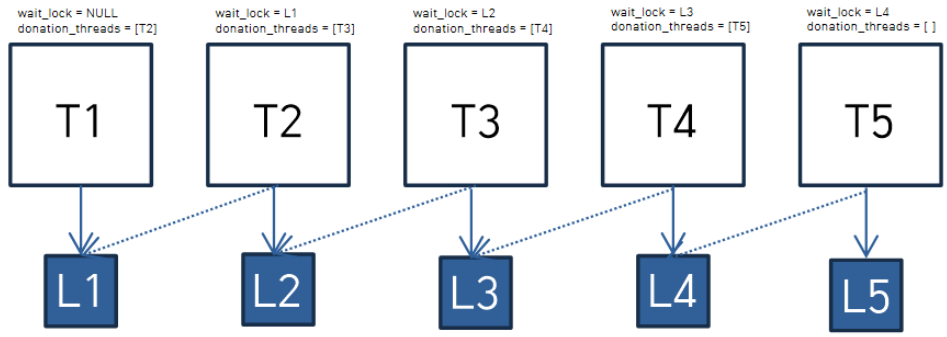
다음은 thread 구조체에 추가하여 priority donation implementation에 사용한 자료구조에 대한 설명이다.

1. struct lock *wait_lock
현재 thread가 waiting을 하고 있는 lock의 주소를 가진 pointer이다. 이 pointer와 lock 구조체의 lock holder을 통해 donated를 할 thread에 대한 priority를 알 수 있다.

2. struct list donation_threads
이 thread의 lock을 기다리고 있는 thread들을 모은 list이다. priority를 donate 받은 thread가 특정 lock을 release하면, 그 다음 변경할 priority를 정할 때 이 list를 활용한다.

3. old_priority
thread가 lock을 release 한 뒤 donate 받기 전의 priority로 돌아가야 하는 경우에 사용한다.
donate를 받기 전 본인의 priority를 저장하는 변수로 사용한다.

nested donation을 도식화한 그림은 다음과 같다.



---- ALGORITHMS ----

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

각 synchronization에 대해서 다음과 같이 설명할 수 있다.

1. semaphore를 기다리는 thread들은 waiters list에 priority에 대해 내림차순으로 정렬되어 넣어진다. 다음 sema_down()을 할 thread를 정할 때, list_pop_front()로 맨 앞의 thread를

꺼내기 때문에 가장 높은 priority를 가진 thread가 깨어나게 된다.

2. lock의 implementation은 semaphore로 되어 있어 같은 원리로 높은 priority를 가진 thread가 먼저 실행된다.

3. condition variable도 비슷하게 waiters list에 priority에 대해 내림차순으로 정렬하여 넣는다. 이때는 B1에 제시한 semaphore_elem의 priority와 sema_comp_function를 활용한다.
cond_signal() 내에서 waiter list의 첫 번째 element를 뽑기 때문에 가장 priority가 높은 것이 먼저 sema_up 되어 깨게 된다.

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

1. lock_aquire()은 내부에서 lock_try_aquire()을 실행한다.
2. 만약 lock을 사용하는 thread가 없다면 이 함수를 통해 lock을 얻게 된다.
3. 만약 lock_try_aquire로 lock을 얻지 못하면 다른 thread가 lock을 aquire한 것이므로
현재 thread의 wait_lock에 lock을 넣고 lock holder의 donation threads에 current_thread를 등록한다.
4. 다음 작업을 진행하는 donation()을 call한다.
    - 현재 thread의 wait_lock이 있고, 그 lock의 lock holder가 존재하는 동안 while 문을 돌면서 현재 thread의 priority가 높은 경우 lock holder에게 priority donation을 한다.

nested donation은 donation() 내에서 다음과 같이 handling한다.

1. current thread가 lock_holder에게 donate를 한 뒤, lock_holder를 새롭게 current_thread로 갱신한다.
2. 갱신된 current_thread의 wait_lock를 가지고 있는 thread가 새로운 lock_holder가 된다.
3. 앞선 과정을 wait_lock이 NULL이거나 wait_lock의 holder가 NULL일 때까지 반복한다.

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

1. 특정 lock에 대해서 lock_release()가 호출된다.
2. 해당 lock의 holder의 donation_threads에서 lock을 기다리고 있는 thread들을 모두 제거한 뒤 남은 thread 중 가장 높은 priority를 현재 thread의 priority로 설정한다.
3. 만약 donation_threads에 남은 thread가 없다면 old_priority에 저장된 값으로 priority를 바꾸고 sema_up()을 실행한다.

---- 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?

thread_set_priority() 함수 안에서 current_thread->priority를 parameter인 new_priority로 바꾸는데 이때 다른 thread가 이 thread의 priority를 바꾸려는 경우 race가 발생할 수 있다.

우리의 implementation에서는 thread_current()로부터 현재 thread의 구조체 포인터 값을 받아서 쓰기 때문에 thread가 바뀌어도 주소로 접근하여 priority를 바꿀 수 있을 것이다.

다만, donation을 구현하는 과정에서 우리는 priority를 수정해야 할 때 thread_set_priority()를 call하지 않고 직접 수정하는 방법을 택했다. 이 선택으로 인해서 발생할 수 있는 potential race를 방지하기 위해서, 우리는 thread_set_priority()를 donation이 발생하지 않는 상황에서만 사용하기로 결정하였고, 관련 주석을 해당 함수 내에 추가하였다.

thread.c에 global variable로 lock를 만든다면, thread마다 thread_set_priority()를 call할 때, 이 lock을 acquire하는 방법을 생각해볼 수 있다. 그러면 현재 thread가 priority를 변경하는 동안, 다른 thread는 해당 lock이 release되기를 기다려야 해서 potential race를 피할 수 있다.

---- RATIONALE ----

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

Priority Scheduling을 위한 design의 요점을 요약하면 다음과 같다.

1. ready_list는 thread의 priority을 기준으로 내림차순으로 정렬되며, schedule() 함수에
의해서 가장 높은 priority를 가지는 thread가 다음 순서에 실행된다.
2. semaphore를 기다리는 thread가 담긴 waiters도 ready_list처럼 priority를 기준으로 내림차순으로 정렬되며, sema_up()을 call하는 과정에서 가장 높은 priority를 가지는 thread 를 unblock한다.
3. condition variable을 기다리는 thread들도 priority에 따라 정렬되어 waiters에 내림차순으로 semaphore_elem의 형태로 저장된다.
4. lock를 기다리는 thread들에 대해서는 donation 작업을 다음과 같이 고안하였다.
    1) 해당 lock을 잡고 있는 thread가 있다면 그 thread의 donation_threads에
    자신을 추가하고, 자신의 wait_lock을 해당 lock으로 갱신한다.
    2) donation()을 call하여 필요하다면 donation 작업을 수행한다.
    3) donation()은 인자로 넘겨받은 thread 구조체 포인터를 통해서 lock으로 연관되어 있는 thread들을 검사하며 priority inversion이 발생한 부분이 있다면 priority donation을 수행한다.
    4) nested donation을 고려하기 위해 wait_lock과 lock 구조체 내에 있는 holder 멤버를 적절히 활용한다.
5. lock를 기다리는 thread들이 여러 개여서 발생하는 multiple donation 문제

는 donation_threads 리스트를 priority 내림차순으로 정렬하여 관리하여 해결한다.

이전에 고려하였던 다른 design은 thread 구조체에 lock_list를 추가하여 각 thread가 어떤 lock을 기다리고 있는지 저장해두는 방식이었다.
하지만 이 방법은 특정 lock이 release될 때 그 lock을 기다리고 있던 thread들에게 정보를 알리기 위해서 여러 번의 list access가 요구되었고, 결정적으로 nested donation을 구현하기 어려웠다.

이 design과 비교하면, 현재의 design은 lock_acquire()나 lock_release()에서 donation
을 좀 더 명확히 구현할 수 있으며, thread 구조체 내의 wait_lock과 donation_threads를 통해 donation 작업이 수행될 때 access해야 하는 변수들에게 더 쉽게 접근할 수 있다는 장점이 있다.

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?

너무 어렵지도, 너무 쉽지도 않은 assignment였다고 생각한다. PintOS를 개선해 나가는 첫 번째 과제인 만큼, PintOS의 부팅 과정, 프로그램 실행 과정 등을 이해해야 했고, thread들이 현재 어떤 방식으로 scheduling되고 있는지 파악해야 했다. 실제로 코딩을 하는 시간보다는 어떤 approach로 접근해야 할 지 고민을 더 많이 했던 것 같고, design를 정립한 뒤 코딩을 진행하니 한결 수월했던 것 같다.

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

특정 insight를 얻은 것 같지는 않지만, 많은 소스 코드 속에서 헤매지 않는 법을 배워가고 있는 것 같다.

>> 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?

Project #1의 경우는 소스 코드 속에서, 특히 init.c와 thread.c로부터 상당히 많은 힌트를 얻을 수 있다고 생각한다. 눈여겨봐야 하는 함수들을 설명해준다면 학생들이 수월하게 project를 진행할 수 있을 것 같다.

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

>> Any other comments?