Pintos Project1

20210084 김지민, 20210216 양준영

목표 달성하기 위한 수정 계획

1. Alarm clock

   A. 현재 진행 방식

      i. timer_sleep 함수의 경우 스레드를 재우기 위해서 thread_yield 함수를 사용한다

      ii. 하지만 이의 경우, x초 동안 멈춰야 할 스레드를 레디 큐에 해당 스레드를 지속적으로 삽입하게 만든다는 문제를 가진다.

      iii. 위의 문제로 인해 busy waiting이 되는 것이 주된 문제이다.

   B. 수정할 방식

      i. Yield로 인해 계속 ready_list에 삽입되는 것을 막기 위해선 sleep된, block된 스레드만을 따로 처리하는 스레드를 만들 필요가 있다.

      ii. 이러한 구현을 위해선 리스트를 sleep을 시켜야 하는 시간(어느 tick에 깨어나야 하는지)를 기준으로 정렬된 상태를 유지하는 것이 수행에 유리할 것으로 보인다.

      iii. 정리하면, thread에 새로운 리스트를 만들고, yield를 편집하거나 새로운 함수를 통해 스레드를 잠시 블록하는 함수를 구현할 필요가 있어 보인다.

      iv. 또한, background에서 틱의 증가를 담당하는 timer_interrupt나 thread_tick의 부분에서 해당 곧 깨어나야할 스레드를 모은 리스트를 검사하여야 할 것으로 보인다.

      v. 위에서 깨어나야할 스레드를 파악하였다면, 해당 스레드를 깨워주는 함수 또한 새로이 구현해야 할 것이다.

2. Priority Scheduling

A. 현재 스케쥴링 방식

i. Round-robin scheduler: 이 스케쥴러는 먼저 들어온 순서대로 처리하면서 정해진 시간이 지나면 ready list의 맨 뒤로 보내고, 다른 쓰레드를 실행한다. 현재 핀토스에는 이 방식이 구현되어 있는데, 이것을 우선순위에 따라 쓰레드를 실행시키는 스케쥴러로 수정해야 한다.

➔ Ready list를 우선순위대로 정렬하고, 새로 추가된 쓰레드 중 현재 running 중인 쓰레드보다 우선 순위가 더 큰 것이 있다면 선점하는 기능을 추가하면 될 것 같다.

➔ 우선, Ready list에 쓰레드를 추가하거나 제외시키는 함수들을 찾아 구현 방식을 알아볼 필요가 있다.

ii. Ready list와 관련된 함수들

1. 쓰레드가 ready 상태가 되도록 하는 함수들: thread_create, thread_unblock, thread_yield
2. 쓰레드가 ready에서 벗어나게 하는 함수: schedule.

iii. 현재 pintos에서의 구현 ("threads/thread.h", "threads/thread.c", "lib/kernel/list.h, c")

1. Void thread_unblock (struct thread *t): list_push_back 함수를 통해 unblock 시킨 쓰레드를 우선 순위 상관없이 리스트 맨 끝에 두는 모습을 확인할 수 있다.
2. tid_t thread_create (const char *name, int priority, thread_func *function, void *aux): 이 함수의 마지막을 보면 thread_unblock 함수를 호출함으로써 ready 리스트 맨 끝에 새로 생성한 쓰레드를 추가하는 것을 알 수 있다.
3. Void thread_yield (void): 현재 실행 중인 쓰레드가 CPU를 양

보하고 다시 ready list에 들어가는 함수다. 이 함수 마지막에서 schedule 함수를 호출하는 것을 볼 수 있는데, 이 함수를 알아보도록 하자.

4. static void schedule (void): 이 함수는 next_thread_to_run 함수에 의해 결정된 next에게 CPU를 넘기는 역할을 한다. 그럼 이제는 Next_thread_to_run 함수에서는 어떻게 다음에 실행될 쓰레드를 결정하는지를 보면 된다.

5. static struct thread * next_thread_to_run (void): 이 함수에서는 list_pop_front 함수를 이용하여 ready_list에 제일 앞에 있는 쓰레드를 반환한다.

   ➔ 정리하자면, 쓰레드를 새로 ready list를 추가할 때는 맨 끝에서부터 추가하고 schedule 함수에서는 next_thread_to_run 함수를 호출함으로써 ready list의 앞에서부터 처리하는 것을 알 수 있다.

   ➔ 그렇다면 ready list를 추가할 때 list 정렬하든가, 아님 ready list에서 꺼낼 때 정렬하든가 하면 해결될 것이다. 아무래도 전자의 방식으로 구현해야 list를 항상 내림차순으로 정렬할 수 있을 것 같으니 ready list에 push하는 방식을 바꿔도록 한다.

   ➔ 핀토스 lib/kernel/list.c에는 리스트와 관련된 다양한 함수가 정의되어 있다. 이 중 정렬과 관련된 함수들을 살펴보자.

6. Void list_sort (struct list *list, list_less_func *less, void *aux): 우리가 인수로 주는 less함수에 따라 list를 정렬하는 함수다.

7. Void list_insert_ordered (struct list *list, struct list_elem *elem, list_less_func *less, void *aux): 우리가 인수로 넣는 less에 따라 list가 정렬될 수 있도록 list에 elem를 알맞게 삽입하는 함수다. 이 함수를 보면 less함수의 조건을 만족하는 순간 for문을

빠져나온 후, list_insert 함수에 그 때의 요소(e)와 elem를 넣는 것을 알 수 있다. List_insert를 확인해보면 elem는 e 이전으로 삽입된다.

➔ 적절한 less함수를 만든 뒤 list_insert_ordered 함수에 넣어 호출하면 내림차순 ready list를 만들 수 있을 것이다.

B. 새로운 스케쥴러 설계

i. Ready list 내림차순 정렬: ready list에 삽입할 때마다 내림차순으로 정렬하는 함수를 생성한다.

ii. Less 함수 설계: list_insert_ordered에 넣을 less 함수를 설계한다. List_insert함수에서, less 함수가 true를 반환하게 하는 요소(e)의 앞에 새로운 요소(elem)가 삽입된다. 우리가 만들 list는 내림차순이므로 가장 큰 요소가 제일 앞으로 와야 한다. 즉, 새로운 요소 elem가 기존 요소 e보다 크다면 삽입할 때마다 더 큰 요소가 앞으로 올 수 있도록 할 수 있다. 따라서 less는 elem > e일 때 참을 내놓도록 설계한다. -> 이후 ready list 맨 끝에 쓰레드를 push하는 list_push_back 함수 대신 list_insert_ordered에 우리가 새로 설계할 less를 넣음으로써 내림차순으로 정렬하면 된다.

iii. 현재 running 중인 쓰레드보다 우선순위가 높은 쓰레드가 있는 경우: 이런 경우 우선 순위가 더 높은 쓰레드가 CPU 점유권을 뺏어야 한다. 우선순위를 바꾸는 함수인 thread_set_priorty에서 새로 우선 순위가 바뀐 현재 쓰레드와 ready list의 맨 앞에 있는 쓰레드를 서로 비교하여 만약 현재 쓰레드의 우선순위가 더 낮다면 즉시 thread_yield 함수를 호출하여 양보하도록 하면 될 것이다. (thread_create 함수에도 이 기능을 추가해야 할까? 실제로 수행해봐야 알 것 같다.)

3. Advanced scheduler

A. Priority donation 방식

i. 위의 priority donation의 경우, multiple & nested donation을 처리하는 부분을 가지고 있다.

B. 새로 바꿀 방식

i. 우선 순위를 상위 스레드로 전달해주는 방식이 아니라 우선 순위를 지속적으로 (4틱 마다) 변경하여 주는 방식으로 scheduler를 구현하는 것이 목적이다.

ii. MLFQS를 인자로 받아 해당 설정을 적용할 수 있도록 구현하여야 하므로, 우선 순위를 donation 받아 set해주는 set_priority나 multiple & nested donation을 처리하는 부분에 조건물을 추가하여 MLFQS의 경우 우선순위를 전달해주는 것을 막을 필요가 있을 것으로 보인다.

iii. 우선 순위를 계산하는 식을 볼 때 나눗셈으로 인한 floating_point의 계산과 관련된 문제가 발생할 수 있을 것으로 보이기 때문에, floating point를 안전하게 계산할 수 있는 함수들을 만드는 것이 유리해 보인다.

iv. 또한, 식에 필요한 recent_cpu, nice가 필요하므로 스레드의 nice 값을 처리하는 thread_set/get_nice 함수를 구현하여야 할 것으로 보이며, system load average, get recent cpu의 경우 해당 값의 100배를 연산하여 반환하므로 오버플로우 문제가 발생할 수 있는지도 확인해야할 것이다.

v. 다음으로 수행해야할 스레드가 next_thread_to_run에서 결정되므로 해당 함수가 우선 순위에 맞게 실행할 스레드를 반환하도록 구현하여야 할 것이다.

vi. 위의 변경을 거칠 경우, 스레드의 우선 순위가 지속적으로 변경되므로 ready_list를 전체 탐색하거나, 4틱마다 정렬하여야 할 것이다. 이를 더 효율적으로 탐색하기 위해 priority_queue와 같은 구조체를 사용할 수 있을지 확인할 필요가 있을 것 같다.

현재 스레드 시스템과 synchronization 분석

1. Analysis of thread system

   A. Thread에 사용되는 변수들과 구조체

      i. 변수 설명

         1. tid: 스레드를 구분하는 식별 번호
         2. name: 이름
         3. stack: stack 포인터
         4. priority: 우선순위
         5. pagedir: page 디렉토리를 저장하는 포인터
         6. magic: 오버플로우가 일어났는지를 판단

      ii. ready_list: list 자료형으로, 핀토스의 프로세스의 4가지 상태인 ready, running, blocked, dying state들 중에서 ready state인 프로세스들을 저장하는 리스트이다.

      iii. all_list: 리스트 자료형으로, 모든 프로세스들을 저장해놓는 리스트이다.

      iv. idle_thread: Idle thread이다

      v. initial_thread: init.c의 main()을 실행하는 초기 스레드이다.

      vi. tid_lock: allocate_tid를 위해 사용되는 lock 변수

      vii. kernel_thread_frame: kernel_thread에서 사용되기 위해 선언된 구조체이다. 해당 프레임에는 return address, 호출할 함수, 함수를 위한 보조데이터를 가지고 있다.

   B. 함수 설명

      i. thread_init: 메인에서 스레드 시스템을 초기화하기 위해 불리는 함수이다. 해당 함수는 INTR_OFF인 상태에서 수행되어야 하며

tid_lock, ready_list, all_list를 초기화해준다. initial thread를 running_thread()를 통해 실행 중인 스레드로 받는다. 그 후, 스레드를 초기화하고 상태를 running 상태로 바꾸고 tid를 받는다.

ii. thread_start: 스케쥴러를 시작하기 위해 메인에서 호출되며, idle 스레드를 만든다. 스레드가 시작될 때, semaphore를 생성 및 초기화 하고 스레드를 생성한다. 이후 인터럽트가 intr_enable()을 통해 가능해지게 된다. sema_down을 통해 semaphore를 사용할 수 있도록 다운시켜놓는다

iii. thread_tick: 현재 스레드에 따라 idle, user, kernel의 틱을 증가시키며, ticks 가 TIME_SLICE 보다 커지는 순간에 intr_yield_on_return를 실행한다. 위 인터럽트를 최종적으로 thread_yield() 를 실행시킨다. 일정 시간마다 자동으로 scheduling 이 발생한다.

iv. thread_print_stats : 스레드 출력용

v. thread_create: name으로 이름을 설정하고, priority만큼의 우선 순위를 가지는 스레드를 생성한다. func과 그에 필요한 aux를 받아 alloc_frame 해주어 해당 함수를 스레드가 실행할 수 있도록 한다. 이후 kernel 스레드, switch 엔드리와 스레드를 위한 stack_fame을 할당한다. 그러곤 스레드를 unblock하여 run queue에 추가한다.

vi. thread_block: 해당 함수는 실행 중 interrupt가 되면 안 되므로, INTR_OFF인지 확인하며, intr_context()를 통해 외부 스레드 인터럽트를 처리 중이 아님을 확인한다. 그러곤 현재 스레드의 상태를 Blocked로 바꾸고 schedule을 실행한다.

vii. thread_unblock: 블록할 것이 스레드가 맞는지 확인한다. 인터럽트를 블록하고 스레드의 상태를 block인지 확인한다. 레디 리스트에 해당 스레드를 넣고 상태를 ready로 만든다

viii. thread_name: 현재 스레드의 이름을 반환한다.

ix. thread_current: 현재 실행중인 스레드를 반환하는 스레드이다. 스

레드가 정말 스레드인지, 현재 실행중인지를 확인할 필요가 있다. (스택오버플로우로 인한 문제 방지)

x. thread_tid: 현재 실행중인 스레드의 tid를 반환한다.

xi. thread_exit: 인터럽트를 비활성화하고, 리스트에서 스레드를 삭제한다. 해당 스레드의 상태를 dying으로 변경하고 schdule한다

xii. thread_yield: 현재 실행 중인 스레드를 중단하고 CPU를 스케쥴러에게 넘긴다. 외부 인터럽트를 수행 중이 아님을 intr_context()를 통해 확인한다. 실행 중에 interrupt되는 것을 intr-disable()로 막는다. 만약 현재 실행 중인 스레드가 idle 스레드가 아니라면 레디 리스트에 스레드를 넣고 상태를 ready로 바꾼다. 이후 다시 schedule하고 인터럽트를 받아드릴 수 있도록 한다.

xiii. thread_foreach: all 리스트에 있는 모든 요소에 대해 func을 실행시킨다.

xiv. thread_set_priority: 현재 스레드의 우선순위를 새로운 우선순위로 변경한다.

xv. thread_get_priority: 현재 실행중인 스레드의 우선순위를 반환한다.

xvi. thread_set/get_nice : 스레드의 nice 점수를 받거나 세팅할 함수

xvii. thread_get_load_avg: system load의 평균을 반환할 함수

xviii. thread_get_recent_cpu: 현재 스레드의 cpu 값을 반환할 함수

xix. idle: idle thread를 만드는 것, semaphore를 세팅하고 idle thread를 현재 실행중인 것으로 한다. idle thread의 semaphore는 공유되지 않도록 sema_up 시킨다. 인터럽트를 막고 스레드를 블록시킨다.

xx. kernel_thread: 커널 스레드를 위한 기초 함수. 스케듈러가 중단된 상태에서 실행되어, interrupt를 받아드릴 수 있도록 활성화한다. 함수를 실행하고 스레드를 삭제한다.

xxi. running thread: 현재 실행중인 스레드를 반환한다.

xxii. is_thread: t가 Null이 아니고 t->magic이 스레드 매직이 맞는지를 확인하여 스레드인지를 반환한다.

xxiii. init_thread: 스레드를 초기화한 후, all 리스트에 해당 스레드를 추가한다.

xxiv. alloc_frame: 스레드의 스택 상단에 SIZE-바이트 프레임을 할당하고 프레임의 기본에 포인터를 반환한다.

xxv. next_thread_to_run: 레디 리스트에 있는 것 중 가장 앞에 있는 스레드를 반환하거나 레디 리스트가 비었다면 IDLE 스레드를 반환하여 다음으로 실행할 스레드를 반환한다.

xxvi. thread_schedule_tail: 아래의 스레드 scheduling에서 사용되는 함수. 다음으로 실행될 스레드의 상태를 Running으로 바꾸고 tick을 새로 시작했으므로 0으로 설정한다. 또한 새로운 주소 공간을 활성화하고 이전 스레드 공간을 할당 해제한다.

xxvii. Schedule: 현재 실행 중인 스레드를 종료시키고 다음으로 실행할 스레드로 전환하는 역할을 한다.

xxviii. allocate_tid: 1씩 증가된 tid를 반환하는데, tid를 증가시킬 때, lock을 걸어 tid의 증가가 다른 활동에 의해 영향을 받아 이상한 값을 반환하지 않도록 하였다.

C. 전체 시스템 분석: 스레드를 변경하는 방법

i. Schedule 함수에서 현재 실행 중인 스레드가 실행 중이지 않을 경우, ready_list의 첫 스레드를 받아와서 해당 스레드를 실행하게 된다.

ii. 현재 실행 중인 스레드의 상태를 저장하고 다음으로 실행할 스레드의 상태를 복원하는 식으로 thread switching이 진행되게 된다.

2. Analysis of synchronization

A. Semaphore

i. Meaning: 여러 개의 프로세스 혹은 쓰레드가 공유자원이나 임계구역(공유자원을 접근하는 코드 부분)을 동시에 점유하려 하면 문제가 발생할 가능성이 있다. 따라서 공유자원을 특정 수의 프로세스 혹은 쓰레드만 접근할 수 있도록 제한할 필요가 있는데, 이를 가능하게 하는 장치이다. 세마포어는 nonnegative 정수 1개와 이를 조절하는 atomoic한 함수 2개로 설명할 수 있다. 정수는 이 공유자원을 이용할 수 있는 쓰레드의 개수를 의미한다.

ii. Atomic 함수 Down, Up 설명

1. Down (또는 P로 표시): 쓰레드가 공유자원의 사용을 요청할 때 이 함수가 시행된다. 세마포어의 value가 0에서 양수가 될 때까지 기다리다가 0이 아니게 되는 순간 1 감소해준다. 즉, 다른 쓰레드가 사용하고 있어 0일 때는 기다리고 있다가 반환하여 숫자가 증가하면 이 값을 감소시킴으로써 공유자원을 사용한다.

2. Up (또는 V로 표시): 쓰레드가 공유자원을 다 쓰고 임계구역에서 나올 때 이 함수가 시행된다. 세마포어의 value를 1 증가해주고, 이 공유자원을 기다리고 있는 쓰레드들 중 하나를 깨워주어야 한다.

->이 두 함수 모두 Atomic 하므로 함수가 실행되는 도중 다른 interrupt 등에 의해 방해받지 않도록 주의해야 한다.

iii. 현재 pintos에서의 구현 (“threads/synch.h”, “threads/synch.c”, “lib/kernel/list.h, c”) -> 이번 과제에서 활용될 함수 위주로 살펴본다.

1. struct semaphore: unsigned로 선언된 value 변수와 waiters라는 리스트가 정의되어 있다.

2. Void sema_down (struct semaphore *sema): while문의 조건문

으로 해당 세마포어 sema의 value가 0인지 체크한다. 즉, sema에 지금 접근 가능한지 불가능한지 검사한 후, 불가능하다면 list_push_back 함수로 sema의 waiters 리스트에 현재 쓰레드를 추가한다. (참고로 List_push_back 함수는 list의 마지막 부분으로 요소를 push해주는 함수다.) 그런 다음 thread를 block시킨다. 나중에 sema_up 함수에 의해 unblock되고 sema의 value도 0이 아니게 되면, sema의 value를 1 감소시킴으로써 해당 자원에 접근할 수 있게 한다.

3. Void sema_up (struct semaphore *sema): 이 함수에서는 해당 자원을 기다리고 있는 쓰레드들의 리스트인 &sema->waiters 에서 가장 앞에 있는 쓰레드를 unblock 시킨 후 sema의 value를 1 증가시킴으로써 자원을 반환한다.

   ➔ 즉, 현재 down과 up은 waiters 리스트에 먼저 들어온 순서대로 꺼내는 방식이다. 이번 과제는 priority에 따라 쓰레드를 실행시켜야 하므로 이 함수들을 수정할 필요가 있다. (둘 다 수정할 수도, 아닐 수도..)

   ➔ Waiter 리스트를 Priority에 따라 정렬하면 해결될 문제다.

B. Lock

i. Meaning: 위에서 살펴본 세마포어와 마찬가지로, 공유자원이나 임계구역을 하나의 프로세스 혹은 쓰레드만 접근할 수 있도록 하는 동기화 도구이다. Lock은 초기값이 1인 세마포어의 특별한 경우라고 볼 수 있다. 즉, 공유자원이나 임계구역에 접근할 수 있는 프로세스나 쓰레드가 1개로 제한되는 것이다. Acquire 함수로 lock을 하면 다른 쓰레드는 접근하지 못하고, release 함수로 lock을 풀어야 다른 쓰레드가 접근 가능하다.

ii. 세마포어와 다른 점

1. 세마포어는 1보다 큰 수를 가질 수 있지만, lock은 최대 1을 가질 수 있다. 즉, 한 번에 하나의 쓰레드에 의해서만 접근될

수 있다.

2. 세마포어는 Down 함수를 실행한 쓰레드와 Up 함수를 실행한 쓰레드가 다를 수 있다. 그러나 lock은 같은 쓰레드가 acquire와 release 함수를 모두 실행해야 한다.

iii. 현재 pintos에서의 구현 ("threads/synch.h", "threads/synch.c", "lib/kernel/list.h, c")

1. struct lock: 해당 lock을 실행한(해당 자원을 lock하고 점유하고 있는) 쓰레드를 가리키는 struct thread *holder와 struct semaphore로 구성되어 있다.

2. Void lock_acquire (struct lock *lock): 우선 lock은 한 번 lock한 자원을 다시 lock하는 것을 허용하지 않으므로 ASSERT로 이를 막는다. 또한 lock은 세마포어의 value가 1인 특수한 경우라고 했으므로 sema_down 함수를 이용할 수 있다. &lock->semaphore를 넣음으로써 해당 자원에 대해 접근을 요청하고, lock -> holder를 현재 쓰레드로 설정하여 해당자원을 lock한 쓰레드를 표시한다.

3. Void lock_release (struct lock *lock): lock은 acquire한 쓰레드가 release까지 해야 하므로 ASSERT로 이를 확인한다. 이후 lock -> holder를 NULL로 바꾸고 sema_up 함수를 호출함으로써 해당 자원을 반납한다.

➔ Lock의 경우 holder를 따로 설정하여 두 번 acquire하거나 본인이 release 하는지 체크하는 것 외에는 세마포어와 같기 때문에 나머지 동작은 세마포어의 함수로 대신하는 것을 알 수 있다. 우리가 구현해야 하는 '우선순위에 따른 선택'은 이미 세마포어의 함수들에서 구현할 것이기 때문에 lock에서는 수정할 필요가 없어보인다.

C. Condition variable

i. Meaning: 쓰레드가 특정 조건이 만족될 때까지 계속 대기하도록 함으로써 CPU 낭비를 줄이는 도구이다. 조건이 만족되면 만족한 쓰레드에서 신호를 보냄으로써 대기 중인 쓰레드는 깨운다. 크게 cond_wait 함수와 cond_signal 함수를 이용한다.

ii. 함수 설명

1. Cond_wait 함수: 쓰레드가 대기 상태로 들어갈 수 있게 해준다. 즉, sleep 시킨다.

2. Cond_signal 함수: 다른 쓰레드에서 특정 조건을 만족하여 대기 중인 쓰레드에게 신호를 보냄으로써 깨우는 함수이다.

iii. 현재 pintos에서의 구현 ("threads/synch.h", "threads/synch.c", "lib/kernel/list.h, c")

1. struct condition: 해당 조건이 만족되기를 기다리는 쓰레드들의 리스트인 waiters를 변수로 가진다.

2. Void cond_wait (struct condition *cond, struct lock *lock): sema 함수와 lock함수에 대해선 위에서 이미 처리했으므로 이 함수에서는 list에 어떤 방식으로 넣는 지만 알면 될 것 같다. List_push_back 함수를 호출하여 cond의 waiters 리스트에 요소를 추가하고 있는데, 앞선 경우들과 달리 여긴 쓰레드를 넣는 것이 아니라 세마포어를 넣는다. 즉, condition의 waiters 리크는 세마포들의 리스트이고, 각 세마포어들의 waiters 리스트로 그 세마포를 기다리는 쓰레드들을 다루는 것이다.

   ➔ 각 세마포어의 waiters리스트는 위 세마포어 파트에서 이미 내림차순으로 정렬했으므로 여기선 condition의 waiters 리스트만 내림차순으로 정렬하는 방법만 고려하면 된다.

3. Void cond_signal (struct condition *cond, struct lock *lock UNUSED): 이 함수도 마찬가지로 리스트에서 어떻게 빼는 지만 보면 되는데, list_pop_front 함수로 앞에 있는 요소를 빼내

는 것을 볼 수 있다.

➔ 즉, condition 또한 세마포어의 경우와 마찬가지로 먼저 들어온 순서대로 처리하고 있는데, 리스트를 내림차순으로 정렬해주는 함수를 추가하면 우선순위대로 꺼내져 수행될 것이다.

3. Priority donation

A. Priority inversion

i. 우선순위에 따른 순서가 뒤바뀐 것이다. 더 높은 우선순위에 있는 쓰레드가 더 낮은 우선순위의 쓰레드에게 CPU 점유를 빼앗겨 우선순위에 따른 실행 순서가 지켜지지 않는 것이다. 우선순위가 H > M > L인 세 쓰레드가 있을 때 만약 L이 점유하고 있는 lock을 H을 요청했다고 해보자. 그럼 H는 lock으로 인해 L을 기다려야 한다. 이때 새로운 쓰레드인 M (이 쓰레드는 L이 점유하고 있는 lock을 요청하지 않음)이 등장하면 L은 더 높은 우선 순위의 M에게 점유권을 넘겨주게 되고, 그 결과 M이 H보다 먼저 실행, 마무리된다. 이번 과제에서는 이 문제를 해결해야 한다.

B. Priority donation: 위 문제를 해결하기 위한 방법이다. 말그대로 우선순위를 양도하는 것이다. 위 상황에서 H는 L이 끝날 때까지 기다리다가 M에게 점유권을 빼앗겼는데, 이런 상황을 만들지 않기 위해선 H는 L에게 일시적으로 자신의 우선 순위를 donation할 수 있다. 이렇게 하면 L이 작업을 수행하는 도중 M이 들어오더라도 CPU 점유권을 빼앗기지 않고 마무리 할 수 있고, 작업이 끝난 L은 원래 우선순위로 돌아가면 된다.

두 가지의 donation 경우를 고려해야 한다. -> multiple donation, nested donation

i. Multiple donation: 한 쓰레드가 두 개 이상의 lock을 점유하고 있을 때, 각각 다른 쓰레드가 이 lock들에 접근하는 경우를 말한다. 즉, L이 lock을 A, B 2개를 가지고 있을 때 M은 lock A에 대해, H

는 lock B에 대해 접근 요청을 하는 경우를 말한다. 이 경우 L은 M과 H 모두에게 우선순위를 양도받고, 더 높은 우선순위를 먼저 갖는다. Release 작업이 끝나면 그 다음으로 높은 우선순위를 갖는다. 모든 쓰레드에게 양도받은 우선순위가 끝나면 다시 원래 자신의 우선순위로 돌아간다.

ii. Nested donation: 체인처럼 연결되어 우선순위를 양도하는 경우다. 만약 L이 lock A를, M이 lock B를 점유하고 있고 H는 lock B에 대해 M은 lock A에 대해 접근 요청을 한 상황이라고 가정하자. 이때 H가 M에게 우선순위를 양도하더라도, M이 lock B을 반환하기 위해선 L에게 lock A를 먼저 얻어야 한다. 따라서 L에게도 우선순위를 양도해야 하는데, 이때 우선순위는 H의 우선순위로 양도받는다. 이렇게 되면 L이 lock A를 반환한 후 M이 lock B를 반환하게 될 것이다. 이때, 각 쓰레드는 lock을 반환한 후 본래의 우선순위로 돌아가야 한다.

iii. 현재 pintos에서의 구현 ("threads/thread.h", "threads/thread.c", "lib/kernel/list.h, c")

1. Struct thread: 쓰레드의 state를 나타내는 status 변수 등 여러 변수가 저장되어 있다.

   ➔ Priority donation에서는 여러 쓰레드로부터 우선순위를 양도받고, 다시 원래 자신의 우선 순위로 돌아가야 하므로 초기 우선순위를 기억해두는 변수가 필요할 것이다.

   ➔ 또한 multiple donation의 경우, 우선순위를 양도해준 쓰레드 수가 많으므로 이들을 리스트로 묶어서 저장할 필요가 있다.

   ➔ 현재 쓰레드가 어떤 lock을 요청하고 기다리고 있는지 기록함으로써, 나중에 다른 쓰레드가 lock을 반납할 때 바로 위에서 언급한 리스트에서 이 쓰레드를 삭제할 수 있도록 한다.

➔ Thread 구조체를 바꿈에 따라 thread_init에서 추가된 변수들을 초기화하는 작업이 필요할 것이다.

2. Void lock_acquire (struct lock *lock): 이미 위에서 살펴본 함수로, sema_down 함수를 호출함으로써 해당 자원에 대한 접근 요청을 하고 있다.

   ➔ Sema_down을 호출하기 전에 만약 이 lock을 미리 점유하고 있는 쓰레드가 있다면 자신의 우선순위를 양도해주는 작업이 필요할 것이다. 또한 sema_down이 끝난 후면 해당 lock을 현재 쓰레드가 점유하므로 lock -> holder를 현재 쓰레드로 바꿔주는 작업 등이 필요하다.

3. Void lock_release (struct lock *lock): lock -> holder를 NULL로 바꾸고 sema_up으로 lock을 반납한다.

   ➔ Priority donation 상황에서는 lock을 반납하기 이전에 자신에게 우선순위를 양도해줬던 쓰레드들을 모아뒀던 리스트에서 해당 lock을 기다리고 있는 쓰레드를 삭제해줘야 한다. 그래야 나중에 우선순위를 재정립할 때 올바르게 할 수 있다.

C. 새로운 디자인 설계

i. Struct thread: 초기 자신의 우선순위를 기록할 변수(initial_p), 자신에게 우선순위를 양도해줄 쓰레드들을 모아 놓은 리스트(priority_donation list), 자신이 기다리고 있는 lock을 가리킬 변수(wait_lock) 등이 선언되어야 이후 설계가 간단해질 것이다.

ii. Thread_init: thread에서 새로 선언된 변수들을 초기화하는 코드를 추가한다.

iii. Lock_acquire: sema_down를 호출하기 전, 우선순위를 양도해주는 작업이 필요하다. 그 전에 우선 현재 쓰레드가 요구하는 lock을 점유하고 있는 쓰레드의 priority_donation list에 현재 쓰레드를 추

가해줘야 한다. 그리고 그 list를 내림차순으로 정렬해야 우선순위가 높은 순대로 작업을 수행할 수 있다. 그 뒤에 실제로 낮은 우선순위를 가진 쓰레드에 우선순위를 양도해주는 작업을 진행한다. 이때, nested donation의 경우를 고려해야 하므로, for문을 이용하여 chain에 있는 모든 쓰레드의 우선순위를 현재 쓰레드의 우선순위로 바꿔준다.

iv. Lock_release: 위 작업으로 쓰레드가 높은 우선순위를 양도받은 후 lock을 release할 상황이 오면, release하기 전에 이 lock을 기다리고 있던 (그래서 이 쓰레드에게 우선순위를 양도해줬던) 쓰레드를 priority_donation list에서 제거해줘야 한다. 그래야 더 이상 양도해줬던 우선순위를 가지지 않게 된다. "list.c"에 보면 "list_remove (struct list_elem *elem)"라는 함수가 있어 리스트에서 elem라는 요소를 삭제해줄 수 있다. 이 함수를 이용하도록 한다.

v. Priority 재설정 "Reset_Priority": priority_donation list에서 쓰레드를 삭제하고 난 후, lock을 release할 쓰레드는 list에서 다시 가장 높은 우선순위를 골라 우선순위를 양도받아야 한다. "Reset_Priority"라는 새로운 함수를 생성하여 priority_donation list 중 가장 높은 우선순위로 변경하는 작업을 진행한다.