PintOS Lab1 Design Report

20210398 최준호, 20210509 정윤재

1. 개요

본 과제는 thread, synchronization, scheduler에 대한 개념과 코드 구현에 대해 이해하고 실습을 통해 주어진 코드를 개선하고자 한다. Busy-wait 형식의 alarm clock을 최적화하고, round-robbin scheduler를 priority를 고려할 수 있는 scehduler로 개선한다. 그리고 이후 발생하는 priority inversion 문제를 해결한다. 마지막으로 cpu starvation을 해결할 수 있는 advanced priority scheduler를 구현하면 된다.

2. Analyzing the current implementation

Thread는 실행 흐름의 단위로 생각하면 하나의 thread는 하나의 executable context를 갖고 cpu를 할당 받게 되면 executable context를 실행한다. Pintos는 thread들을 관리하는 함수들을 제공하고 thread.c에 선언되어 있다. main함수에서 thread를 다루는 방법을 설명하고 이와 관련 있는 thread 함수들을 분석한다.

2-1. Thread

struct thread

struct thread는 4KB 크기의 page 형태로 구성된다. thread structure는 최대 1KB 정도로 page의 시작 부분(offset 0) 부터 저장되고, 나머지 부분은 thread의 stack으로 사용되고, page의 마지막 부분부터 아래 방향으로 커진다. thread stack은 malloc(), palloc_get_page() 등으로 동적할당 된다.

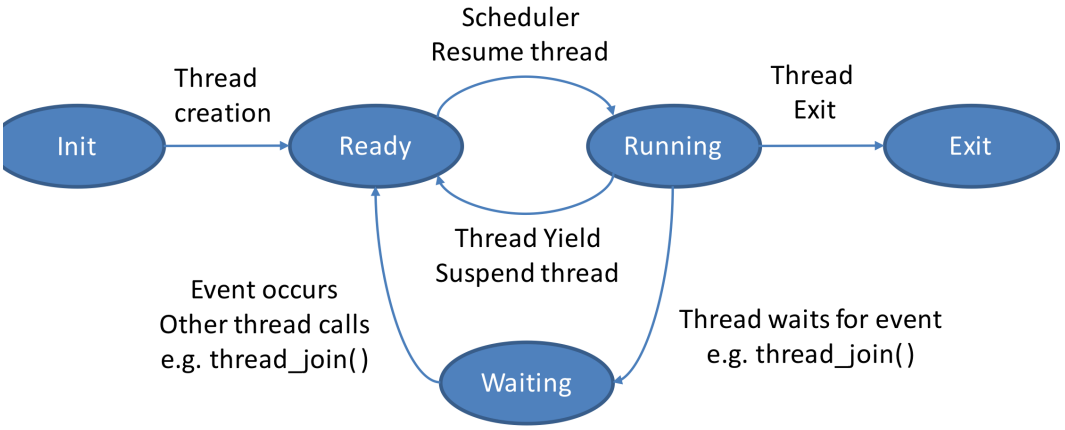
Thread는 다른 thread에게 switch 될 때 이전 실행 흐름을 불러올 수 있어야 한다. 따라서 여러 정보들을 저장해야 하는데 이를 위해 TCB를 사용한다. TCB의 역할을 thread 구조체가 하게 된다.

구성요소들의 역할과 정의는 아래와 같다.

- **tid_t tid**

- thread identifier로 모든 thread는 kernel의 전체 lifetime을 통틀어서 하나의 tid를 갖고 있다. thread가 새로 생길때마다 1부터 시작해서 1씩 증가하는 tid를 갖게 된다.

- **enum thread_status status**



- thread의 life cycle은 총 위의 그림과 같이 ready, running, waiting(blocked), exit(dying) 총 네단계로 구성된다.
  - **THREAD_RUNNING**: 1개의 thread만이 running state로 존재
  - **THREAD_READY**: thread가 run할 준비가 완료된 상태, ready_queue에 저장
  - **THREAD_BLOCKED**: lock이 release되거나 interrupt가 호출되기를 기다리는 상태, thread_unblock에 의해 ready state가 되기 전까지는 scheduling되지 않는다.
  - **THREAD_DYING**: thread_exit 호출 이후 dying state를 갖게 된다.
- **unit8_t *stack**
  - 모든 thread는 각자의 stack을 갖고 있다. thread가 running state일 때, CPU의 stack pointer register (%esp)가 stack의 top을 가리키게 된다. Thread가 switch 될 때, register의 값들은 stack 위에 저장된다.
- **int priority**
  - 0부터 63까지 총 64개의 priority가 존재한다. 이 priority를 기준으로 priority schedular를 구성한다.
- **struct list_elem allelem**
  - list_elem struct는 linked list의 before pointer와 next pointer를 갖고 저장한다. allelem은 모든 thread를 관리하는 all list를 구성하는데 사용된다.

thread_foreach() 함수에 의해서 모든 thread들을 iterate할 수 있다.

- **struct list_elem elem**
  - thread.c의 run을 하기 위해 준비된 ready list 또는 sync.c의 semaphore wait list의 element로 사용된다. ready list나 wait list에 변동사항이 있을 때 elem을 이용하여 관리해준다.
- **unsigend magic**
  - 항상 thread.c에 정의된 THREAD_MAGIC 값을 갖고, stack overflow를 알기 위해서 사용된다. thread_create() 함수는 항상 running thread의 magic 값이 THREAD_MAGIC 값을 갖는지 확인하여 stack over flow를 확인한다. (이 값이 struct의 가장 높은 주소에 위치하므로 stack overflow 발생시 가장 먼저 값이 바뀐다)

main()를 보면 thread system에 관여하는 함수들이 많지만 주요한 함수로 thread_init()과 thread_start() thread_exit()이 있다.

Thread functions

- **void thread_init (void)**

  threads/init.c의 main 함수에서 호출된다. 시스템이 시작될 때, 여러 list들을 초기화하고, running thread()를 호출하여 현재 실행중인 thread를 알아내고 이를 init_thread() 함수를 사용해서 main thread로 초기화하고, state를 THREAD_RUNNING으로 바꾸고 tid를 할당하는 과정을 수행한다. thread는 다른 thread가 호출해야 생성되기 때문에 가장 시작 thread가 필요하다. 이는 kernel이 thread로 변환되면서 해결된다.

- **void thread_start (void)**

  main 함수에서 호출된다. thread_create() 함수를 호출하여 idle thread를 생성한다 (idle thread는 다른 ready thread가 없을 때 호출되는 thread). 이후 intr_enable() 함수를 호출해서 interrupt를 활성화하고, timer interrupt에서 return 될 때까지 schedular가 실행된다. 그리고 Idel thread가 initialize 될 때까지 wait한다.

- **void thread_tick (void)**
  매 timer tick마다 timer_interrupt ()에 의해 호출된다. 현재 실행중인 thread의 tick을 증가시키고 schedular를 호출하는 역할을 한다.

- **tid_t thread_create (const char *name, int priority, thread_func * func, void *aux)**
  ”name”, “priority”를 인자로 받아 새로운 thread를 생성하고 시작한다. thread의

kernel stack에 다양한 stack frame을 저장하기 위한 구조체들을 생성한다 (thread switching 시 필요한 정보들을 저장). 마지막으로 새롭게 생성한 thread의 tid를 return해준다.
palloc_get_page() 함수를 이용해서 thread structure의 메모리를 할당해준 뒤, init_thread() 함수를 호출해서 thread를 초기화한다. 이후 tid를 할당하고, kernel thread frame, switch entry frame, switch thread frame을 할당해준다. 자세히 보면 assert 함수로 생성하려는 thread가 executable context를 가지고 있는지 체크하고 만약 있다면 page 할당을 해준다. page 할당이 끝나면 새로운 thread를 parameter들을 이용하여 초기화 해준다.

kernel thread frame은 kernel thread가 처음 실행될 때, func(aux)를 실행한다. 다음으로 switch entry frame은 thread switch시 switch_entry() (switch.S)로 집입할 수 있게하는 frame이다. 이를 통해 thread switch의 초기 작업을 수행하고 stack pointer를 조정한다. 마지막으로 switch thread frame은 thread가 switch되었을 때, swirch_entry에 필요한 stack frame이다.

- **void thread_block (void)**
  running thread의 상태를 THREAD_BLOCKED로 바꿔준다. block state의 thread는 thread_unblocked가 호출되기 전까지 scheduling 되지 않는다. 현재 진행중인 thread가 block 되었기 때문에 새로운 thread를 결정해주기 위하여 schedule()함수를 호출하여 cpu를 할당해줄 thread를 결정한다.
- **void thread_unblock (struct thread *t)**
  THREAD_BLOCKED 상태의 thread를 THREAD_READY 상태로 바꾼 뒤, 다시 ready_list에 넣어준다. interrupt를 disable 해줘서 running thread를 preempt하지 않는다.
- **void thread_exit (void)**
  all_list에서 thread를 제거하고 THREAD_DYING 상태로 전환한다. 이 함수는 return 되지 않는다.
- void thread_yield (void)

  thread를 yield할 때 실행하는 함수이다. 현재 thread가 idle thread가 아니라면 ready list에 해당 thread를 넣어주고 ready 상태로 바꿔준다. 그리고 cpu를 할당해줄 thread를 찾기 위해서 schedule()를 호출한다. old_level은 함수 실행 이전의 interrupt level로 해당 함수가 끝나면 다시 원래의 interrupt level로 돌려 놓는다.
- **thread * next_thread_to_run (void)**

  schedule()함수 안에서 호출하는 함수로 context switching이 일어날 때 만약 ready list에 thread들이 존재한다면 thread를 pop 해준다. 만약 존재하지 않는다면 idle thread를 return해준다.

- **static void schedule (void)**

  schedule 함수는 다음 실행될 thread를 cpu에게 할당 받을 수 있도록 해준다. 우선 interrupt가 꺼져있는 것을 확인하고 현재 실행중인 thread가 running state가 아닌지 체크한다. 이후 스위치 될 thread와 현재 thread를 parameter로 하는 switch_thread()를 실행하여 thread를 바꾸어준다.

  - ASSERT (intr_get_level () == INTR_OFF) : scheduling 도중에는 인터럽트가 발생하면 안 된다.
  - ASSERT (cur→status != THREAD_RUNNING) : CPU 소유권을 넘겨주기 전에 running 스레드는 running state가 아니어야 한다.
  - ASSERT (is_thread (next)) : next_thread_to_run() 에 의해 올바른 thread 가 return 되었는지 확인한다.

- **Switch_threads()**

  실행 중이던 thread의 레지스터 값과 스택 포인터를 stack에 저장하고 다음 cpu를 할당받을 thread의 레지스터와 스택 포인터를 cpu에 할당해주는 역할을 한다. 이전에 실행하던 context를 그대로 이어서 실행할 수 있도록 하여 context switching이 원할하게 진행될 수 있도록 도와준다.

- **thread system initialized에 대한 추가 설명**

  - 위에 설명한 함수들과 설명으로 thread system initializing에 대한 설명이 충분하지 않은 것 같아 부연 설명을 하면 우선 thread는 struct 구조체로 memory에 저장되게 된다. 메모리 페이지 주소 첫 부분에 thread 구조체가 위치하고 남은 부분에는 stack이 저장된다. stack은 그동안 배운 개념과 같이 반대 부분부터 쌓이게 된다.
  - 우선 thread system을 구축하기 위하여 ready list와 wait list와 같은 queue가 필요하다. 그리고 위에서 설명하였듯이 프로세스는 적어도 thread가 하나씩은 있어야 한다. 그렇기 때문에 main thread가 필요하게 되고 thread_init()을 통하여 이 과정을 수행하게 된다. thread_init()은 lock, ready list, all list 를 초기화 해주고 이름을 main으로 하고 priority default 값을 가지는 thread를 초기화 해준다. 그리고 해당 thread를 running state로 바꿔주고 tid를 할당해준다. 이후 thread들이 create함수를 이용하여 계속 생성되게 된다. create 함수는 name, priority, function, aux를 parameter로 받게 되고 초기화 된다. 이 thread의 excutable context 들의 함수가 function이 되고 이를 이용하여 aux를 넘긴다. 추가로 thread 구조체와 stack을 위한 page를 할당하고 관련한 멤버들을 initialize해준다. thread는 block state로 초기화 되지만 이후 함수 리턴 이전에 unblock 되어 context switching이 가능해진다.

- **Switch_thread()에 관련하여 추가 설명**
    - switch_thread는 stack에 있는 register를 저장하고 현재 thread의 stack number에 있는 CPU의 현재 stack pointer를 저장한다. 이후 새 thread의 stack을 CPU의 stack pointer에 복구하고 stack으로부터 register를 복구한 뒤 return 된다. 여기서 추가로 생각해봐야 할 점이 thread를 처음 run할 때이다. thread_create()를 통하여 새로운 thread를 만들면 switch_thread가 위의 동작을 그대로 실행할 수 없다. 이때 kernel thread frame, switch entry frame, switch threads frame을 사용한다. stack frame 상단에는 switch tread frame이 있다. 이 stack frame의 eip에는 switch_entry()가 있다. 요약하면 stack frame이 return하면 switch_entryr가 호출되었고 이 함수는 switch 함수가 argument를 받는 부분을 뛰어 넘어 원만한 진행을 할 수 있게 해준다. 다음 stack frame은 switch entry frame이다. 이 frame은 thread_schedule_tail을 부르고 return 할 때 kernel_thread를 호출한다. 이 kernel_thread를 위하여 kernel_thread_frame이 존재한다. interrupt를 활성화하고 함수들이 실행될 수 있도록 하고 return할 때 thread_exit()을 호출한다. 정리하면 예외 처리를 잘하기 위하여 fake stack frame이 존재한다.
- **Thread state change에 관련하여 추가 설명**
    - thread가 running 중일 때 다양한 이유로 thread가 더이상 실행되지 않고 기다려야 하는 상황이며 block state로 변하게 된다. scheduler는 ready 상태의 thread들 중 하나를 running state로 바꿔주게 된다. 만약 block 상태인 thread가 특정 조건이 만족된다면 다시 ready 상태로 변경되어 scheduler의 선택을 기다린다. Timer interrupt가 발생하면 thread는 yield하여 ready 상태로 변경될 수 있다.
    - ready list에서 scheduling을 통하여 running할 state를 정하는 것이 중요한데 현재 pintos는 rounnd-robin 방식으로 구현되어 있고, 어떤 thread가 ready state가 되어서 ready list에 들어갈 때 이 ready list는 queue의 형태이고 맨 뒤로 들어가게 된다. 또한 scheduling을 해야 하여 next_thread_to_run() (schedule() 함수가 호출) 에서 다음으로 cpu를 할당받을 thread를 골라줄 때 queue의 맨 앞의 thread를 골라주는 방식이다

  scheduling이 일어나는 상황을 살펴보면 아래와 같은 3가지 경우가 있다.

  thread_yield(), thread_block(), thread_exit() 함수가 실행되면 다른 thread로 cpu가 할당되게 된다. 그런데 만약 현재 thread가 저 함수를 호출하지 않으면 계속 cpu를 독점하게 되고 이것을 방지하기 위하여 timer interrupt가 존재한다.

  timer interrupt는 전역변수인 ticks을 증가시켜 자체 시간을 카운트한다. 이렇게 ticks이 계속 증가하는데 이 ticks이 TIME_SLICE보다 커지면 인터럽트가 실행되어 thread_yield()를 실행시킨다. 따라서 일정 시간이 지나면 3가지 함수가 호출되지 않아도 알아서 자동으로 schedule된다.

2-2. Synchronize variable

Semaphore

1. struct semaphore
    - semaphore는 음이 아닌 정수를 이용하여 여러 thread가 shared object에 동시에 접근하는 것을 예방한다. 0 또는 그 이상의 값으로 초기화되고, seam_down, sema_up 함수들을 이용해 공유자원의 사용을 조정한다.
    - struct semaphore는 unsigend type의 value와 semaphore에 접근하기 위해 block state로 대기하는 waiter list를 포함한다.
    - sema 변수가 0이면 이용 가능한 공유 자원이 없다는 뜻이고 더 크다면 이용 가능한 공유 자원의 개수를 알려준다.
2. sema_init (struct semaphore *sema, unsigned value)
    - sema_init 함수는 argument의 sema의 value를 argument의 value 값으로 초기화하고 waiting list를 초기화한다.
3. sema_down (struct semaphore *sema)
    - sema_down 함수의 sema가 null이 아니고, interrupt handling 도중 호출된 것이 아니라면, old level 변수를 이용해 interrupt를 disable 해준다. 이후, sema value가 1이 아니라면, sema의 waiter list에 현재 thread를 추가해준 뒤, thread를 block한다. 그리고, sema의 value가 1이면, 1을 빼준뒤, old level을 해제한다.
    - 자세하게 설명하면 wait 할 때에는 함수가 sleep 상태에 빠지게 되므로 interrupt handler 내에서 호출되면 안된다. intr_disable함수는 interrupt를 비활성화 시켜주고 이전 interrupt state를 반환해준다. 그리고 함수 마지막 부분을 보면 intr_set_level()함수가 있는데 level에 따라 interrupt를 on이나 off로 바꿔준다. 그리고 이전 interrupt state를 반환한다. 즉 sema_down함수를 실행하는 동안 interrupt를 고려하지 않고 함수가 끝나게 된다면 이전 interrupt state로 돌려준다.
    - while문을 보면 sema value가 0이 될 때까지 현재 실행중인 thread를 waiters list에 넣어주고 block처리 해준다. 현재 공유자원을 사용할 수 없을 때 그 thread를 해당 자원을 사용할 수 있을 때까지 대기시키는 역할을 해준다. 따라서 공유자원이 사용가능 할 때까지 계속 해서 thread들을 wait list에 넣어준다. 이후 sema up이 호출되면 while문을 빠져나와 실행된다.

```
void sema_down (struct semaphore *sema)
{
```

```
  enum intr_level old_level;

  ASSERT (sema != NULL);
  ASSERT (!intr_context ());

  old_level = intr_disable ();
  while (sema->value == 0)
    {
      list_push_back (&sema->waiters, &thread_current ()
-> elem);
      thread_block ();
    }
  sema->value--;
  intr_set_level (old_level);
}
```

4. sema_up (struct semaphore *sema).
    - thread가 사용중이던 shared object를 돌려줄 때 실행된다. if문을 통하여 sema waiter를 확인하고 리스트가 비어있지 않다면 리스트에서 하나의 thread를 pop해주고 unblock해준다. 즉 ready list로 보내주어 해당 thread가 그 공유자원을 사용할 수 있게끔 해준다. 그리고 sema value를 1씩 증가 시킨다.

```
void sema_up (struct semaphore *sema)
{
  enum intr_level old_level;
  ASSERT (sema != NULL);

  old_level = intr_disable ();
  if (!list_empty (&sema->waiters)) {
    thread_unblock (list_entry (list_pop_front (&sema->w
aiters), struct thread, elem));
  }
  sema->value++;
  intr_set_level (old_level);
}
```

Lock

1. struct lock
    - Lock은 같은 shared object에 접근하는 thread들의 mutal exculsion을 위해 사용되는 synchronize variable이다. 기본적으로 lock이 1이 된다면 waiter list에 저장된 thread들 중 가장 우선 순위가 높은 thread가 실행되어 critical section의 코드들을 실행한다. 이때, lock은 0이 되고, 그 동안 다른 thread들은 같은 shared object에 접근할 수 없게 된다.
    - struct lock은 현재 lock을 소유하고 있는 holder를 저장하는 thread holder와 lock을 정의하는 semaphore를 포함하고 있다.
    - semaphore는 value로 1보다 큰 값을 가지지만 lock은 최대 하나의 thread만이 소유할 수 있고 value가 1과 0만 가능하다.
2. lock_init (struct lock * lock)
    - lock의 holder를 NULL로 설정하고, sema_init (&lock→semaphore, 1)을 통해 lock을 초기화한다.
3. lock_acquire (struct lock *lock)
    - 현재 thread가 lock을 소유하고 있지 않을 때 실행된다. 이후 sema_down 함수를 실행하고 lock→holder를 current thread로 초기화한다. 앞서 설명하였듯이 sema_down함수에 갔을 때 sema value값이 1이면 해당 thread가 lock을 소유하게 되지만 0이면 wait list에 들어가서 이미 lock을 소유하고 있는 thread가 끝날 때까지 기다려준다. 즉 sema down의 while문 안에 머물러있는다.

```
void lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  sema_down (&lock->semaphore);
  lock->holder = thread_current ();
}
```

4. lock_release (struct lock *lock)
    - lock holder는 lock_release 함수를 호출한다. 이후, lock_holder를 NULL로 초기화하고, sema_up 함수를 호출한다. 즉 현재 thread가 소유하고 있는 lock을 해제한다. 이후 sema up함수에서 semaphore의 wait list를 체크하여 만약 thread

가 존재한다면 unblock 해준다. 만약 비어있다면 정해진 규칙에 따라 scheduling 된다.

```
voidlock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  lock->holder = NULL;
  sema_up (&lock->semaphore);
}
```

Condition

1. struct condition
   - condition은 여러 lock들을 동시에 관리하는 기능을 수행한다. 기본적으로 lock을 바탕으로 동작하고, condition value를 이용해 조건이 참이 될 때까지 기다리는 동작을 수행한다.
   - condition structure는 waiter들의 list로 구성된다. 즉, 해당 condition이 완료되길 기다리는 thread들에 대한 정보를 담아둔 list이다.
2. cond_init (struct condition *cond)
   - cond가 NULL이 아닐 경우, cond의 waiter list를 초기화한다.
3. cond_wait (struct condition *cond, struct lock *lock)
   - 특정 조건을 만족할때까지 thread를 block해두는 함수이다.
   - 우선 새로운 structure인 semaphore_elem을 확인하자

     elem과 semaphore로 이루어져있다. list_elem은 thread를 doubly linked list에 넣는데 사용된다. linked list에는 ready_list와 sema_down에 waiter list가 있다.
   - semaphore elem 변수 waiter를 선언한다. 그리고 함수를 호출한 thread가 lock을 소유하고 있는 경우, sema_init 함수가 호출되고 waiter의 semaphore를 0으로 초기화한다. 이후 cond의 waiters list에 waiter를 elem을 이용하여 추가해준다. 즉, waiter는 조건의 만족에 대한 semaphore를 의미한다. 해당 thread가 wait을 호출하면 semaphore(value가 0이다)을 만들어주고 waiters list에 들어가게 된다. 이후에는 현재 thread가 갖고 있던 lock을 release하고 sema_down을 통해 waiter의 seamphore가 0인 경우 thread를 block하고 semaphore

wait list에 들어가고 대기한다. 이때 해당 condition을 만족시켰다는 signal을 받기 전까지 block된 상태로 sema_down 함수에 머물러 있는다. 그리고 cond_signal을 통해 cond이 만족되었음을 확인했다면, sema_down함수가 return되고 다시 lock_acquire 함수를 호출하여 전에 갖고 있던 lock을 다시 소유한다.

```
void cond_wait (struct condition *cond, struct lock *loc
k)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  sema_init (&waiter.semaphore, 0);
    list_push_back (&cond->waiters, &waiter.elem);

  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acqwwuire (lock);
}
```

4. cond_signal (struct condition *cond, struct lock *lock UNUSED)
   - 특정 조건이 만족 되었음을 전달해주는 함수이다.
   - thread가 lock을 소유하고 있고, cond의 waiters list가 비어있지 않은 경우, list의 가장 앞에 있는 waiter의 semaphore에 대해서 sema_up 함수를 실행해서 cond_wait 함수의 sema_down 함수에서 block되어있는 thread를 unblock해준다.

```
void cond_signal (struct condition *cond, struct lock *loc
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));
```

```
  if (!list_empty (&cond->waiters)) {
    sema_up (&list_entry (list_pop_front (&cond->waiters),
                          struct semaphore_elem, elem)->
  }
}
```

5. cond_broadcast (struct condition *cond, struct lock *lock)
   - con_signal은 waiters list의 맨 앞의 semaphore에 대해서만 sema_up을 해주었다면, cond_broadcast는 waiters list에 존재하는 모든 semaphore에 대해서 sema_up을 실행해준다.

```
void cond_broadcast (struct condition *cond, struct lock
*lock)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);

  while (!list_empty (&cond->waiters))
    cond_signal (cond, lock);
}
```

3. Proposed design

3-1. Alarm clock

1. Logic
   a. 현재의 alarm clock은 busy-wait 방식으로 구현되어있다. timer.c의 함수를 보면 timer_sleep 함수는 각 thread를 깨울 시간을 입력 받은 뒤, 계산한 tick의 크기에 따라 sleep 함수를 실행하거나 delay 함수를 실행한다. 입력받은 ticks가 0보다 클 경우, sleep 함수가 실행되면서, 입력받은 tick에 다다를 때까지 계속해서 thread_yield를 통해 CPU를 넘겨준다. 반대의 경우에는 timer_delay 함수가 실행이 된다.
   b. 예를 들어, Thread T가 10 tick 후에 특정 작업을 실행하길 원한다면, 매 tick마다 thread T가 실행되어 10 tick이 되었는지 확인하는 작업을 가진다. 따라서, thread는 running state와 ready state가 계속 반복되는 것을 확인할 수 있다. ready

queue에 있는 thread들은 매 tick마다 running state가 되어 일어날 시간이 되었는지 확인한 뒤, 일어날 시간이 되지 않았다면 다시 ready sate로 전환된다. 이러한 방식은 계속해서 context switching이 일어나게 되기 때문에 CPU의 자원이 낭비되고 성능이 저하된다.

2. Design Goal
    a. 특정 tick에 도달하지 못한 thread가 다시 ready queue에 들어가 계속 tick을 확인 하지 않고, thread를 block 시킨 후 sleep_list에 넣어준 뒤, 특정 tick에 도달하면 unblock 해주는 방식을 사용하여 design하고자 한다. 즉 thread가 sleep 함수를 받으면 ready가 아니라 block state로 들어가게 된다.
    b. thread를 깨울 tick을 저장하기 위해 thread structure에 wakeup이라는 정수 변수를 추가한 뒤, timer_sleep이 실행되며 입력받은 n tick을 wakeup에 저장해준 뒤, n tick이 될때까지 thread를 block 해둘 것이다. 그리고 timer_interrupt 함수를 통해서 n tick에 도달했음을 확인해준 뒤, thread를 unblock 해주고 ready list에 넣어준다.
3. Functions
    a. timer_sleep (int64_t ticks)
        i. thread는 running 될 때마다 timer_sleep의 while 문이 실행되고, thread start 부터 몇 tick이 실행되었는지 확인한다. 그리고 그 값이 argument로 들어온 ticks의 값보다 작을 경우, thread_yield를 호출하여 다시 ready_state로 전환된다.
    b. timer_interrupt (struct intr_frame *args UNUSED)
        i. interrupt를 발생시켜 tick을 update 해준다.

3-2. Priority scehduling

1. Logic
    a. 현재 Pintos는 Round-Robin scheculer를 사용한다. 따라서, CPU가 ready list에 들어간 순서대로 실행되는 (FIFO)방식을 따르게 된다.
    b. thread_unblock, thread_yield이 호출될 경우 ready list에서 가장 prioriy가 높은 thread가 CPU를 할당 받아 running state로 전환된다. 두 함수는 list.c의 list_push_back 함수를 사용한다.
    c. thread_block, thread_yield, thread_exit이 호출될 경우 running thread가 priority를 고려하여 CPU를 양보한다. 세 함수들은 마지막에 schedule 함수를 호

출하는데, 이 함수는 switch_thread 함수를 호출하게 되는데, 전환될 thread는 next_thread_to_run 함수에 의해서 결정된다.

d. next_thread_to_run 함수는 ready_list의 맨 앞에 위치하는 thread를 return하는 함수이다.

2. Design goal

a. priority scheduler를 구현하기 위해서는 우선 순위가 높은 threard가 먼저 실행이 될 수 있도록 해야한다. 또한, 현재 CPU를 점유중인 thread의 priority보다 ready list의 thread의 priority가 가장 높으면 CPU를 양도하도록 해야한다. 즉 현재 진행 중인 thread의 priority가 바뀌거나 ready list에 변동사항이 생겨 ready list의 thread들 중 가장 큰 prioirty값의 변화가 생기는 경우에 가장 큰 priority를 가진 thread가 cpu를 할당받을 수 있게 해줘야 한다.

b. 여러 thread들이 shared object에 접근하고자 하는 경우, semaphore를 얻어서 실행되어야한다. 그리고 priority scheduler는 그 중 가장 priority가 높은 thread가 semaphore를 받아 먼저 실행될 수 있도록 구현해야한다. 기존의 코드에서는 semaphore의 wait list에 thread를 priority와 관련없이 넣어준 순서대로 나오게 구현되어 있다. 이 부분을 위에서 설명한 대로 수정해야 한다.

c. Inversion

만약 위의 문제점을 고려하여 코드를 구현했을 때(즉 pintos 1 pdf 21페이지의 4가지 조건을 모두 만족 했을 때) thread 간 서로 semaphore를 주고 받을 때, 더 높은 priority를 가진 thread가 semaphore를 위해 대기하는 동안 priority가 낮은 다른 thread가 실행되는 문제가 발생할 수 있다. 따라서, semaphore를 주고받을 때 priority donation을 구현함으로써, 이러한 문제가 생기지 않도록 구현하고자 한다. 자세히 설명하면 thread(High) 즉 높은 priority를 가진 thread가 lock A를 acquire 하고 thread(Low)가 lock A를 갖고 있다고 하자. thread(H)는 lock_acquire에 의하여 sema_down에 의하여 A의 wait list에 들어가고 block state에 돌입할 것이다. 이때 block이 후에 scheduling이 될 때 만약 thread(H)와 thread(L) 사이의 priority값을 가진 thread가 ready list에 있다면 thread(L)보다 먼저 cpu를 점유할 것이고 결론적으로 A의 release가 늦어져 thread(H)보다 thread(M)이 먼저 실행될 것이다.

d. Donation

위와 같은 inversion 문제를 donation 방법을 이용하여 해결한다. 위와 같은 예시에서 thread(H)가 lock A를 acquire할 때 lock을 갖고 있는 thread에게 자신의 priority를 전달한다. 그리고 lock이 풀리면 자신의 priority를 철회하는 식으로 구현한다. 철회하는 표현이 적합하진 않고 자신의 priority 영향력이 사라진다고 보면 된다. 자세한 방법은 아래에 설명한다.

- Multiple donation

  하나의 thread가 여러개의 lock을 소유하고 있는 경우이다. 나머지 thread들이 하나의 thread에 묶여있는 상황이다. 따라서 lock을 소유하고 있는 thread는 해당 lock들을 원하는 thread들에게 모두 prioirty를 donate 받게 된다. 이때 만약 소유하고 있던 lock A,B 중 A가 release 되었어도 A를 ~~donate~~ acquire한 thread들의 priority 영향력이 없어지는 것이고 B를 원하는 thread들에게 donate 받은 priority는 여전히 유지된다.

- Nested donation

  체인처럼 thread들이 엮여있는 형태이다. thread(L)이 lock A를 소유하고 있는데 A를 thread(M)이 원한다. 더하여 thread(M)은 lock B를 소유하고 있고 thread(H)는 lock B를 원한다. 따라서 thread(H)의 lock을 풀기 위해선 thread(L)까지 priority가 전달되어야 한다.

3. Functions
   a. Control ready_list
      i. ready_list가 priority 순서대로 저장되고 관리되도록 하고자 한다.
         - list_insert_ordered (lib/kernel/list.c) 함수를 사용하면 linked list의 원하는 부분에 elem를 삽입할 수 있다. 우리는 thread의 priority를 비교하는 함수를 새롭게 생성하여 argument로 전달해준다면, ready_list를 priority descending order로 정렬할 수 있다.
         - 이후, thread_unblock, thread_yield 함수에서 list_push_back 대신 list_insert_ordered 함수를 사용한다.

      ii. 실행 중 thread의 priority가 바뀌는 경우 thread를 switch 해줘야한다.
         - thread_create의 경우 thread_unblock에 의해 ready_list가 수정된 뒤, 실행중인 thread와 ready_list의 thread의 priority를 비교해야한다. 그리고 thread_set_priority에 의해 현재 실행 중인 thread의 priority가 바뀔 경우, priority를 비교한뒤, 필요한다면 CPU를 양도할 수 있도록 해야한다.
         - running thread의 priority와 ready_list의 가장 앞에 위치하는 thread의 priority를 비교한 뒤, ready 중인 thread의 priority가 더 높다면 thred_yield를 호출해 CPU를 양도해야한다.

   b. Control synchronize variable

i. a에서 했던것과 마찬가지로 sema_down 함수가 실행될 때 current thread가 semaphore의 waiter list에 추가되고 해당 thread를 block하는데, 이때 list_push_back 함수를 사용한다. 그리고 sema_up 함수가 다음 thread를 고르는 과정은 waiters lsit의 맨 앞에 위치하는 thread가 unblock 된다. 따라서, sema_down의 경우 a와 마찬가지로 list_insert_ordered 함수를 사용해주면 된다. 이후, sema_up 함수는 thread_unblock 이전에 waiters list를 priority에 대해서 정렬주고 unblock 해줘야한다. 이후 unblock 된 thread와 ready list에 있는 thread들간의 비교를 해주고 cpu가 할당된다. 공유 자원에 대하여 thread가 접근하고 싶을 때 접근하는 thread들 역시 가장 높은 priority를 설정해줘야 한다.

ii. lock은 sema_down, sema_up 외에 추가적으로 lock holder만 설정해주면 되기 때문에 추가적인 수정이 필요하지 않을 것 같다.

iii. cond_waiter, cond_signal 함수의 경우 semaphore들의 waiters list들을 관리해야 한다. 그렇기 때문에 각 semaphore의 waiter list에 저장된 thread의 priority가 가장 높은 순서대로 실행될 수 있도록 해야한다. 이외의 과정은 a,b와 유사하게 작성해주면 된다.

c. Priority donation

i. 우선, thread structure에 original_priority, wait_lock, donation_list, donation_elem 변수를 추가하여 준다. 이를 통해 thread가 가진 lock을 기다리는 thread들에 대한 정보와 원래의 priority 에 대하 정보를 추가한다. 그리고 init_thread에서 추가된 변수들에 대한 초기화도 함께 진행해준다.

각각 priority를 donate 받기 이전 자신의 고유의 priority, 해당 thread가 원하는 lock, 자신에게 donate 해준 thread와 그 thread들에 대한 정보를 갖고 있는 list등을 나타낸다.

ii. lock_acquire 함수는 holder가 존재하는 lock에 대해서 wait을 하게 되면 다음의 과정을 수행할 수 있어야한다. 먼저, wait_lock에 기다리는 lock의 정보를 저장한다. 그리고 해당 lock의 donation list에 정렬해서 삽입해 주어야한다. 그리고 그 과정이 끝나면 해당 lock의 holder가 priority를 donate 받을 수 있도록 하는 함수를 실행한다. 이때, priority donation을 수행하는 함수는 lock_acqurie를 호출한 함수의 wait_lock의 donation_list를 확인한 뒤, 그 중 가장 높은 priority를 donate 받으면 된다. 단, nested donation을 고려해줘야하므로, 반복문을 통해서 donation list를 확인해줘야한다.

iii. 만약, 여러개의 lock을 가진 thread가 lock을 하나 해제했을 때, 해제된 lock의 waiter는 더이상 priority를 donate 해주지 않아도 된다. 대신, 다른 lock의 waiter가 자신의 priority를 donate해줘야 한다. 따라서, lock_release 함수는 donation list에서 해제된 wait_lock이 해제된 lock과 같은 thread들을

지워준다. lock A가 release 되었다면 lock A를 acquire해서 donate 받은 priority는 고려하면 안된다. 우선, original_priority로 복귀해준 뒤, donation list에 thread가 남아있다면, 그중 가장 높은 priority를 doante 받는다.

iv. 마지막으로 thread_set_priority에 의해서 실행중인 thread의 priority가 변경될 경우 original_priority를 초기화 해준 뒤, lock_release 호출하는 priority donate 함수를 사용하면 변화하는 priority에 대해서도 대응할 수 있을 것이다.

3-3. Advanced scheduler

1. Logic
    a. priority schduler가 실행될 경우 priority가 낮은 thread는 CPU를 점유하기 어려운 환경이 형성된다. 이 경우 전체 thread의 response time이 늘어나는 문제가 생길 수 있다. prioriity donation을 통해 high priority를 끌어와 사용할 수 있지만, 일부 한정적인 thread에 한정된다.
2. Design goal
    a. priority를 실시간으로 계산해준다.

    이러한 방식을 개선하기 위해 advanced scheduler 는 multi-level feedback queue scheduling 방식을 취한다. priority를 매 띡마다 조절하여 여러 개의 ready queue를 관리하고 priority 가 가장 높은 ready queue 에서 round-robin 로 schedule 한다.

    1. nice
        - min = -20, max = 20, default = 0
        - nice 값이 작을수록 CPU를 양보하려하고, 클수록 CPU를 점유하려 한다.
    2. priority
        - 정수값
        - min = 0, max = 63, default = 31
        - priority = pri_max - (recent_cpu / 4) - (nice * 2)의 식을 갖는다.
    3. recent_cpu
        - 실수값
        - recent_cpu는 최근 cpu 사용량을 의미한다.

        - 최근에 사용된 thread일 수록 큰 값을 가져, 낮은 priority를 갖게됨
    - recent_cpu = (2*load_avg) / (2*load_avg +1) * recent_cpu + nice
    - 매 tick마다 running thread recent_cpu 값이 1 증가한다. (idle thread 가 아닐 때)
4. load_avg
    - 실수값
    - 최근 1분 동안 수행 가능한 thread의 평균 개수를 의미한다.
    - load_avg가 커질 수록 큰 recent_cpu 값을 갖게된다. → 클수록 priority 가 낮아진다.
        - 수행 가능한 thread이 수가 많을 때에는 CPU를 고르게 배분해야하기 때문에 priority를 천천히 증가시킨다.
    - default = 0
    - load_avg = (59/60) * load_avg + (1/60) * (# of ready thread + # of running thread)
    - 매초마다 load_avg 값을 계산

이러한 방식을 개선하기 위해 advanced scheduler 는 multi-level feedback queue scheduling
방식을 취한다. priority 를 실시간으로 조절하며 여러 개의 ready queue 를 준비하여 priority 가
가장 높은 ready queue 에서 round-robin 로 schedule 한다.

이러한 방식을 개선하기 위해 advanced scheduler 는 multi-level feedback queue scheduling
방식을 취한다. priority 를 실시간으로 조절하며 여러 개의 ready queue 를 준비하여 priority 가
가장 높은 ready queue 에서 round-robin 로 schedule 한다.

1. Fixed-point arithmetic
    a. priority 계산에 사용하는 recent_cpu, load_avg는 실수형 값인데, pintos kernel에서는 floating point 기능을 지원하지 않는다. 따라서, 우리는 fixed point 방법을 사용해서 실수형 변수들을 계산한다. fixed point 실수형 표현은 아래와 같이 구현된다.

    0 (sign bit, 1) + 00000000000000000 (integer, 17) + 00000000000000 (decimal, 14)

b. fixed_point.h 파일을 새롭게 생성한 뒤, fixed point 함수들을 만들어 주어야 한다.

- int int_to_fp (int n)
  - n * 1024 (= 2^{14})
- int fp_to_int (int x)
  - n / 1024 (= 2^{14})
- int fp_to_int_round (int x)
  - fp를 반올림하여 정수로 변환한다. (소수 부분이 0.5 이상인지 이하인지 판단하여 결정)
- int add_fp (int x, int y)
  - 두 fp를 더해주는 함수로, 그냥 더해주면 된다.
- int sub_fp (int x, int y)
  - 두 fp를 빼주는 함수로 그냥 빼주면 된다.
- int add_mixed (int x, int n)
  - 정수 n과 fp X을 더해주는 함수로, 정수를 fp로 변경한 뒤, 더해준다.
- int sub_mixed (int x, int n)
  - 정수 n과 fp X을 빼주는 함수로, 정수를 fp로 변경한 뒤, 빼준다.
- int mult_fp (int x, int y)
  - 두 fp를 곱해주는 함수이다.
  - fp를 그냥 곱할 경우 decimal 부분이 사라지기 때문에 1024를 곱해준다.
  - overflow가 생길 수 있으므로 32bit int 대신 64bit int64를 사용해야한다.
- int mult_mixed (int x, int n)
  - fp와 정수를 곱하는 함수로, 그냥 곱해주면 된다.
- int div_fp (int x, int y)
  - 두 fp간 나눗셈을 구현하는 함수이다.
  - fp를 그냥 나눌 경우 decimal 부분이 두배가 되기 때문에 1024를 나눠준다.

        - overflow가 생길 수 있으므로 32bit int 대신 64bit int64를 사용해야한다.
    - int div_mixed (int x, int n)
        - fp x를 정수 n으로 나눠주면 되는 함수로, 그냥 나눠주면 된다.

3. Functions
    a. 우선, pintos kernel은 floating point 연산을 지원하지 않기 때문에 직접적으로 계산할 수 없기 때문에 floating point 연산에 사용할 함수들을 작성해주어야한다.
    b. 먼저 pintos pdf에서 설명되어있는 함수들을 구현해줘야한다. 아래 함수들의 경우, 값을 변경하는 함수들이기 때문에 interrupt의 발생을 막아 방해 없이 값을 가져올 수 있도록 해야한다.
        i. int thread_get_nice(void)
            - curent thread의 nice value를 return 하는 함수이다.
        ii. void thread_set_nice(int new_nice)
            - curent thread의 nice value를 new_nice로 설정하는 함수이다.
            - new_nice로 인해 priority가 변경되면 CPU를 yield 해줘야한다.
        iii. int thread_get_recent_cpu(void)
            - recent_cpu 값을 return한다.
            - return_cpu가 fp이므로 100을 곱한 뒤, 반올림한 정수 값으로 return해준다.
        iv. int thread_get_load_avg(void)
            - load_avg를 return한다.
            - load_avg가 fp이므로 100을 곱한 뒤, 반올림한 정수 값으로 return해준다.
    c. 이후 mlfqs에서 사용되는 변수들을 계산하는 함수들을 구현한다.
        i. void mlfqs_cal_priority(struct threadt *t)
            - thread t의 priority를 계산해주는 함수이다.
            - recent_cpu, nice를 fp 연산 함수를 활용해 priority를 계산한다.
        ii. void mlfqs_cal_recent_cpu(struct thread *t)
            - thread t의 recent_cpu를 계산하는 함수이다.

- load_avg, nice를 fp 연산함수를 활용해 recent_cpu를 계산한다.

iii. void mlfqs_cal_load_avg(void)

- load_avg 값을 계산하는 함수이다.
- load_avg는 thread 고유의 값이 아니라 시스템 단위로 활용되는 값이기 때문에 idle thread인 경우에도 계산해줘야한다.

iv. void mlfqs_recent_cpu_increment(void)

- current thread가 idle thread가 아닐 경우, recent_cpu의 값을 1 증가 시켜주는 함수이다.

v. void mlfqs_update_recent_cpu(void)

- 모든 thread의 recent_cpu 값을 업데이트 해주는 함수이다.

vi. void mlfqs_update_priority(void)

- 모든 thread의 priority 값을 업데이트 해주는 함수이다.

d. Additional design

a. thread 별로 계산되는 nice와 recent_cpu를 struct thread에 추가해주고 init_thread에서 초기화해준다. 시스템 단위로 활용되는 load_avg는 전역 변수로 선언해주고 main 함수에서 실행되는 thread_start 함수에서 초기화 해주었다.

b. mlfqs scheduler는 thread의 prioity를 매 tick 마다 계산해주게 되는데, 이는 매 tick마다 호출되는 timer_intterupt 함수에 의해서 실행된다. 따라서, thread_set_priority 함수를 주석처리해야한다. 그리고 mlfqs에 사용할 변수들을 계산하는 함수들을 구현해주고, 조건에 맞게 update 해주면 된다. 각 변수는 아래의 상황에서 update 된다.

- recent_cpu는 thread가 running 중이라면 매 tick마다 1 증가한다.
- priority는 4 tick마다 update 된다.
- 매 tick 마다 load_avg를 계산해준다.

c. mlfqs scheduler는 priority를 시간에 따라 계산해주기 때문에, priority donation을 사용해주지 않아도 된다. 따라서, lock_acquire, lock_release 에서 구현해준 priority donation 기능을 주석처리 해줘야한다.