Pintos Project: 1. Threads

Design Report – team 30

20210054 정하우

20210716 최대현

Context

1. **Introduce & structure**

   1-1. Thread System

   - **Definition**
   - **Thread structure**
   - **Analysis**

   1-2. Synchronization

   - **Definition**
   - **Semaphore & lock structure**
   - **Analysis**

2. **Analysis of the current implementation**

   2-1. Alarm Clock

   2-2. Priority Scheduler

   2-3. Advanced Scheduler

3. **Design Plan**

   3-1. Alarm Clock

   3-2. Priority Scheduler

   3-3. Advanced Scheduler

1. Introduce & Structure

1-1. Thread System

Thread

- 프로그램 내에서 thread는 별도의 실행 경로이다. 운영체제는 thread를 생성하고 관리하며, thread를 생성한 프로그램과 동일한 memory와 resource을 공유한다. 이를 통해 여러 thread가 하나의 프로그램 내에서 공동으로 작업하고 효율적으로 작업할 수 있다.
- thread는 process 내의 단일 sequence stream이고, 각 thread는 정확히 하나의 process에 속한다.
- Pintos 상에서 구현된 thread는 다음과 같은 형태의 structure을 가지고, 구현상 필요한 사항들을 고려하여 이를 수정할 수 있다.

Thread Structure

```
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. */

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

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

- **tid_t** tid: thread의 id를 나타내는 변수이다.
- **enum thread_status** status: thread의 현재 state를 나타내는 변수이다. Thread의 state로는 **THREAD_RUNNING, THREAD_READY, THREAD_BLOCKED, THREAD_DYING** 4가지가 있다.
- **char** name[16]: thread의 name이며, thread를 initialize할 때 이름을 argument로 넣어줄 수 있다.

- **uint8_t ***stack: thread의 switch frame을 가리키는 포인터이다.
- **int priority:** priority scheduler의 기준이 되는 priority이다.
- **struct list_elem** allelem: 모든 thread가 있는 list와 연결되는 변수이다. thread가 생성시 list에 삽입되고 remove시 나가게 된다.
- **struct list_elem** elem: doubly linked list에 연결되고 ready list semaphore 내에 waiting list도 포함된다.
- **unsigned magic:** overflow를 감지하기 위한 변수이다. Thread/thread.c에서 사전 정의된 랜덤한 숫자값인 THREAD_MAGIC으로 설정되며, overflow가 일어나면 해당 값이 변경되고 이를 감지하여 overflow assertion를 발생한다.
- * page directory에 대해서는 추후에 다루도록 한다.

Analysis

Thread system initialization

- **How is the thread system initialized?**
  - Pintos에서는 **thread_init()** 함수를 실행함으로서 thread system을 initialize할 수 있다. thread_init 함수가 실행되면, 우선 lock_init() 함수와 list_init() 함수를 실행하여 lock과 ready_list, all_list를 초기화한다. 그 뒤 init_thread() 함수를 호출하며 current running thread를 'main'이라는 이름과 함께 default priority(31)로 초기화하고, 해당 thread의 state를 THREAD_RUNNING으로 설정한 뒤 allocate_tid() 함수를 호출하여 tid를 할당한다.

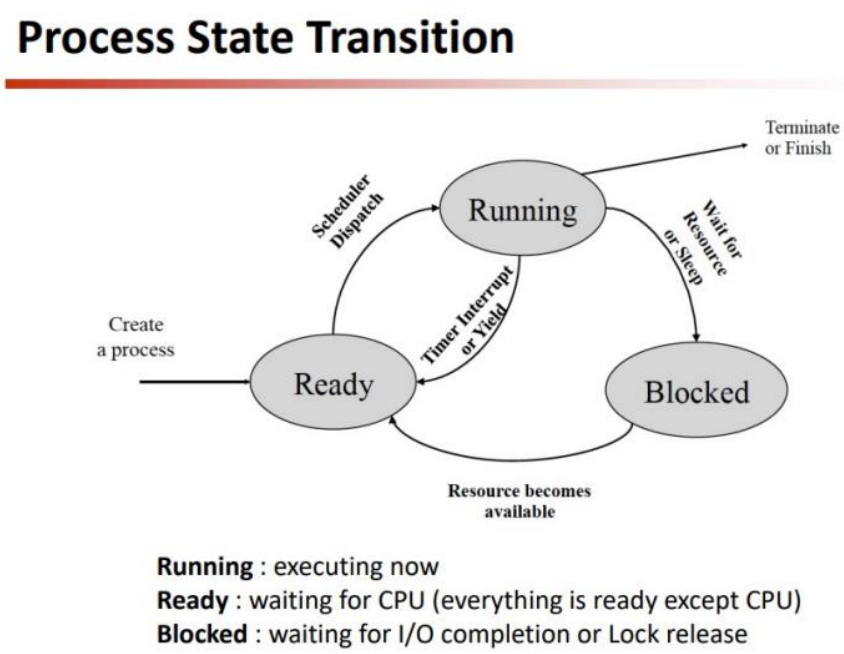
Thread system creation

- **How to create a thread?**
  - Pintos에서는 **thread_create()** 함수를 실행함으로써 thread system을 create할 수 있다. thread_create() 함수에서는 사전 정의된 다른 sub function들을 실행하여thread가 갖춰야 할 것들을 갖춘다. thread_create() 함수가 실행되면, palloc_get_page() 함수를 호출하여 page와 같은, thread가 필요로 하는 메모리를 dynamic allocate한다. 그 뒤, init_thread() 함수를 호출하여 struct의 멤버 변수들을 적절히 초기화하며 initialize를 수행한다. 그 뒤 allocate_tid 함수를 실행하여 tid(thread id)를 할당해준다. 그 뒤 kernel thread, switch entry, switch thread 등 필요한 공간에 대한 stack frame을 alloc_frame 함수를 통해 할당한다. 마지막으로 initialize해줄 때 blocked state로 초기화됐던 해당 thread를

unblock해줌으로써 thread가 ready_list에 들어갈 수 있도록 한다. 마지막으로 할당했던 tid를 return함으로써 thread_create() 함수 호출이 끝난다.

- 다른 예시로는 thread_start() 함수 호출이 있을 수 있다. thread_start()는 새로운 idle thread를 생성한다. Idle thread는 다른 thread를 관리하는 thread로, main thread로 이해해도 무방하다. thread_start() 함수의내부를 보면 thread_create() 함수를 호출하고 sema_init(), sema_down() 함수를 통해 semaphore를 만들어서 idle_thread의 생성 작업동안 다른 thread들의 시작을 차단한다. thread_start()에서의 sema_init()에서는 semaphore의 value를 0으로 initialize한다. 이를 통해 sema_down()에서 0이 아닐 때까지 while loop가 돌게 되므로 semaphore value가 0이 아닐 때까지 대기한다. 그 뒤 idle이 만들어지고 나면 sema_up()을 호출하는데, 이를 통해 sema_down() 함수에서 thread list에 thread가 존재하면 list 맨 처음에 위치한 thread를 ready state로 변경한 뒤 thread_schedule()을 호출하고 이후 while loop를 빠져나온다.

- **When and how does the status of threads change?**



- 위 사진에는 3개의 state가 diagram으로 제시되어 있지만, 공식적으로 pintos에서 지정한 state는 4가지이다. **("READY", "BLOCKED", "RUNNING", "DYING")**

- 각 state 간의 transition이 발생하는 상황은 다음과 같이 다양한 상황을 고려

해볼 수 있다.

- **Thread creation:** thread_create()를 호출하여 새로운 thread가 생성되면 "READY" 상태로 시작한다. 이것은 스레드가 실행될 준비가 되었지만 아직 실행하도록 예약되지 않았다는 것을 의미한다.
- **Thread yield:** thread는 thread_yield()를 호출하여 자발적으로 CPU를 양보할 수 있다. 그렇게 하면 현재 "RUNNING" 상태에 있던 thread가 "READY" 상태로 이동하며, 다른 ready state의 thraed가 running state로 전환될 수 있다.
- **Thread Blocking:** 스레드는 자신을 차단하거나 다른 스레드가 이미 갖고 있는 lock(lock_acquire())을 획득하거나 사용할 수 없는 semaphore(sema_down())에서 대기하는 것과 같은 다른 메커니즘에 의해 차단될 수 있다. 스레드가 차단되면 running state에서 blocked state로 전환된다. 이후 lock이나 semaphore을 이용 가능해지면 unblock을 통해 ready state로 전환될 수 있다.
- **Thread Unblocking:** block된 thread는 조건이 만족되면 다시 ready state가 될 수 있다. 예를 들어, 만약 semaphore에서 대기 중이었는데 다른 thread가 semaphore를 해제했다면, block state의 thread는 unblock 함수 호출을 통해 ready state가 되고, ready list에 들어간 뒤 scheduler에 따라 바로 running state로 전환되거나 다른 thread가 끝나기를 대기할 수 있다.
- **Thread exit:** Pintos에서는 thread가 실행을 완료하거나 thread_exit() 함수를 호출하면 dying state로 이동한다. dying state는 스레드가 exit 중임을 의미하며 실행이 완료되면 해당 리소스가 정리된다.
- **Thread Scheduling:** thread의 state는 thread scheduling 과정에서 변경될 수 있다. scheduler는 자신의 스케줄링 정책에 따라 다음에 어떤 스레드를 실행할지를 결정하고, 그에 따라 ready 상태와 running state 사이에서 thread를 전환할 수 있다.
- **Timer Interrupt:** timer interrupt는 thread의 state 변경을 발생시킬 수도 있다. 예를 들어 timer interrupt가 발생하면 현재 실행 중인 thread를 선점하여 ready state로 이동시켜 다른 thread가 실행되도록 할 수 있다.

Thread switching

- **On demand of switching the execution context, switch_threads() should be called.**
- **In which concrete cases, is it called?**

- Pintos에서는 switch_threads() 함수가 context switch가 일어나는 상황에 호출된다. Context switch가 일어나면 현재 실행되고 있는 thread의 context를 저장하고, 다른 thread를 실행한다. Context switch는 thread yield, thread blocking/unblocking, thread initialization, thread exit 등 다양한 상황에 발생할 수 있다.

- **What it does?**
  - 두 thread 간의 context switching을 발생시킨다.

- **What is the role of kernel_thread_frame, switch_entry_frame, switch_threads_frame?**
  - **kernel_thread_frame:** kernel_thread_frame은 새로 생성된 thread의 초기 상태를 나타내는 데 사용되는 data structure이다. 새로운 thread가 생성되면 초기 실행 컨텍스트가 필요하며, 이 프레임은 스레드가 실행을 시작할 때 가져야 할 register, stack pointer, program counter에 대한 초기 값을 저장한다. 즉, 새로운 thread를 생성할 때 thread의 실행을 위한 initial context를 설정하는 데 사용된다.
  - **switch_entry_frame:** switch_entry_frame은 thread가 user mode에서 kernel mode로 전환하는 context switching 과정에서 사용되는 data structure이다**.** context switch가 개시되는 시점의 CPU의 register 및 state에 대한 정보를 포함한다. Thread가 system call을 하거나 interrupt가 발생할 때 시스템 호출을 하거나 예외가 발생할 때, CPU는 user mode에서 kernel mode로 전환할 필요가 있다. switch_entry_frame은 kernel에 제어권을 전달하기 전에 user mode의 context를 저장하는 것을 돕는다. kernel에 진입할 때 CPU의 state를 포착하는 데 사용되며, user mode와 kernel mode를 연결하는 역할을 한다.
  - **switch_threads_frame:** switch_threads_frame은 두 thread 사이의 context switch에서 사용된다. OS의 scheduler가 하나의 thread를 실행하던 중, 다른 thread로의 context switching을 결정했을 때 context switch 전에 현재 실행 중인 thread의 state를 저장하고 다음 thread의 state를 load해야 한다. switch_threads_frame은 현재 실행 중인 thread의 상태(registers, stack pointer 등)를 저장하며, 나중에 해당 thread가 다시 실행될 때 이 state를 복원하는 데 사용된다.

Fixed Point

- **What is fixed point arithmetic and why is it necessary?**

- priority를 계산하기 위한 변수 recent_cpu, load_avg는 실수이다. 하지만 Pintos는 복잡한 부동소수점 연산을 지원하지 않는다. 따라서 실수의 직접적인 연산은 불가능하므로 고정소수점(fixed-point)연산을 따로 구현해 이 문제를 해결해야 할 필요가 있다.

Priority Scheduler

- **Priority Scheduling**
  - Thread 간의 우선 순위를 고려한 scheduling을 의미한다. Pintos에서는 single thread process만 고려하기 때문에, 만약 한 thread가 실행되고 있다면 다른 준비된 thread들은 해당 thread의 실행이 끝날 때까지 기다린다. 만약 준비가 된 thread가 2개 이상이라면, Operating System은 어떤 thread를 먼저 실행하고 나머지 thread들을 ready list에 둘 지 결정해야 한다. 이를 올바르게 결정하는 알고리즘을 가진 thread scheduler가 priority scheduler이다. 이 때 결정하는 배경이 되는 것이 priority이다. Pintos에서는 0부터 63까지의 priority가 존재하고, 숫자가 클수록 우선순위가 높고 ready list에서 먼저 실행되어야 한다.
- **Priority Inversion**
  - Priority scheduling을 할 때 고려해야 하는 문제이다. 우선 순위가 높은 스레드 H, 중간 스레드 M, 낮은 L이 있는 상황에서, 만약 H가 L을 기다려야 하고(예를 들어 L이 Lock을 가지고 있는 경우), M이 ready_list에 있다면, L은 running thread가 될 수 없을 것이기 때문에 L을 기다리는 H는 CPU를 점유하지 못할 것이다. 이러한 Priority를 해결하는 방법은 **Priority Donation**이다.
- **Priority Donation**
  - 위와 같은 **Priority Inversion** 상황을 해결하기 위해 일시적으로 높은 우선순위를 가진 thread가 낮은 우선순위를 가지고, lock 되어있는 thread에게 일시적으로 Priority를 기부하는 것을 의미한다. 위 상황을 예로 들면, L이 lock을 유지하는 동안 H가 Priority를 L에 기부하고 L이 lock을 해제하면 다시 H로 기부한 Priority를 돌려놓는 것이다.

1-2. Synchronization

Semaphore

- Semaphore는 concurrent programming에서 고려해야 할 요소로, shared resource에 접근 가능한 process, 또는 thread의 수를 관리하는 값이다. Semaphore는 음이 아닌 정수의 값을 가지며 Pintos에서는 semaphore up과 semaphore down이라는 연산을

통해 관리된다. Semaphore down 연산을 수행하면 semaphore을 1 감소시키며, semaphore up 연산을 수행하면 semaphore 값을 1 증가시킨다. 이 때 유의할 점은 semaphore은 음수가 될 수 없기 때문에 만약 semaphore 값이 0이라면, semanphore가 up 연산을 통해 양의 정수가 될 때까지 기다렸다가 down 연산을 수행한다. 이를 관리하기 위해서 semaphore의 structure에는 struct list waiters(waiter list)가 포함되어 있다.

Semaphore structure

```
/* A counting semaphore. */
struct semaphore
  {
    unsigned value;             /* Current value. */
    struct list waiters;        /* List of waiting threads. */
  };
```

Lock

- Lock은 semaphore와 구조가 유사하고, 유사하게 동작하지만 lock의 value는 0과 1의 값만을 가질 수 있다. 즉 하나의 thread가 어떤 lock에 acquire 연산을 통해 들어갔다면 다른 thread들은 해당 lock을 이용하기 위해서는 들어간 thread가 release를 통해 빠져나올 때까지 기다려야만 한다.

Lock structure

```
struct lock
  {
    struct thread *holder;      /* Thread holding lock (for debugging). */
    struct semaphore semaphore; /* Binary semaphore controlling access. */
  };
```

- **struct thread** *holder**: lock_acquire()**를 호출하여 현재 lock을 holding하고 있는 thread의 pointer를 담는 변수이다.
- **struct semaphore** semaphore: Pintos에서 lock structure은 binary semaphore 변수를 멤버 변수로 갖는다. 즉 binary semaphore을 wrapping한 형태이다.

Condition variable

- condition variable(조건 변수)은 thread 간의 synchronization을 효율적으로 관리하기 위한 도구이다. Condition variable은 복잡한 multi-threading 환경에서 race condition

이나 deadlock과 같은 문제를 예방하고, thread 간의 협력을 가능하게 한다.

- Condition variable을 사용하는 주요 목적으로는 thread 간 통신과 thread 간 synchronization을 예로 들 수 있다. condition variable을 사용하면 한 thread가 다른 thread에게 특정 작업을 수행하도록 signal을 보낼 수 있다. 또한, condition variable은 스레드가 공유 자원을 안전하게 사용할 수 있도록 synchronization하는 데 사용된다.

Condition variable structure

```
/* Condition variable. */
struct condition
  {
    struct list waiters;        /* List of waiting threads. */
  };
```

- **struct list** waiters: 대기하고 있는 thread들을 담은 list이다.

Analysis

- **How semaphores and locks differ?**
  - Semaphore과 lock의 가장 큰 차이는 동기화 대상의 수이다. Lock은 synchronization 대상이 하나인 반면 semaphore은 여러 개의 대상에 대해 synchronization을 수행할 수 있다.
  - 또한, Pintos에서 lock structure는 binary semaphore을 멤버 변수로 갖고 holder라는 pointer 변수를 추가적으로 멤버로 갖는다. 즉 lock structure가 semaphore sturcture을 포함하고 있다. 이 holder는 lock_acquire을 진행하고 있는 thread를 담는 pointer이다. lock을 단순히 binary semaphore로도 치환할 수 있지만, 편의 상 이와 같이 구현한 것으로 생각된다.
- **What happens if sema_down() is implemented with if statement instead of the current while loop?**
  - If를 이용한 구현을 사용하면 여러 thread가 semaphore를 위해 경쟁할 경우 if 조건을 동시에 통과하여 비결정적으로 차단된다. 즉, 차단되는 thread의 일관성을 보장하지 못한다. 이로 인해 race condition과 같은 문제가 발생할 수 있다. 반면, while loop는 semaphore의 value가 0보다 클 때만 thread가 진행되도록 보장한다. 따라서, sema_down()에서 while loop 대신 if 문을 사용하면 여러 thread가 semaphore을 위해 경합하는 multi-thread 환경에서 올바른 동작을 보장할 수 없다. 결론적으로,

while loop는 semaphore 구현에서 올바른 synchronization 구현을 위해 꼭 필요하다.

2. Analysis of the current implementation

2-1. Alarm Clock

Problem Description

- 기존 **timer_sleep()** 함수는 모든 Ready State Thread를 확인하는 **Busy Waiting** 방식을 사용한다. 이는 자원을 쓸데없이 낭비하는 방식이기 때문에 수정을 통해 기능을 더 향상시켜야 한다. sleep 상태의 thread를 ready state가 아니라 block state로 두어 깨어나기 전까지는 scheduling에 포함되지 않도록 해 이런 비효율을 해결할 수 있다.

관련 함수 설명

- **1. timer.c/timer_sleep()**

```
timer_sleep (int64_t ticks)
{
  int64_t start = timer_ticks ();

  ASSERT (intr_get_level () == INTR_ON);
  while (timer_elapsed (start) < ticks)
    thread_yield ();
}
```

Ready State thread 중 깨어날 thread가 있는지 확인하고 깨어날 thread의 state를 Running State로 변경하는 함수이다. 현재는 loop를 통해 모든 Ready State thread를 확인하는 Busy Waiting 방식으로 Thread State를 동기화하고 있다.

- **2. timer.c/int64_t timer_ticks()**

```
timer_ticks (void)
{
  enum intr_level old_level = intr_disable ();
  int64_t t = ticks;
  intr_set_level (old_level);
  return t;
}
```

현재 시스템 시간을 tick 단위로 return 하는 함수이다.

- **3. timer.c/timer_elapsed(int64_t then)**

```
timer_elapsed (int64_t then)
{
  return timer_ticks () - then;
}
```

parameter로 받은 then 보다 얼마만큼의 시간이 지났는지 tick 단위로 return 하는 함수이다.

2-2. Priority Scheduler

A. Priority Scheduling

Problem Description

- 기본적으로 Pintos에서 제공하는 scheduler는 Round-Robin 방식으로, thread 간의 Priority를 따지지 않고 순차적으로 ready_list에 thread를 삽입한다. 즉 FIFO 구조의 Queue로 ready_list가 동작하고 있어서, 이를 priority-scheduler로 thread 간의 priority를 고려하여 priority 순으로 실행하게끔 수정해야 한다.
- Pintos의 thread/thread.c 파일에서는 thread가 ready_list라는 이름을 가진, 사전 정의된 struct에 저장된다.
- thread들이 ready_list에 삽입되는 경우는 크게 2가지로 나눌 수 있는데, 1) 새로운 thread가 만들어질 때 2) blocked state였던 thread가 unblock되는 경우가 있다.

관련 함수 설명

- **1. thread.c/void thread_unblock(struct thread *t)**

```
void
thread_unblock (struct thread *t)
{
  enum intr_level old_level;

  ASSERT (is_thread (t));

  old_level = intr_disable ();
  ASSERT (t->status == THREAD_BLOCKED);
  list_push_back (&ready_list, &t->elem);
  t->status = THREAD_READY;
  intr_set_level (old_level);
}
```

- Block state에 있는 thread를 unblock state로 바꿔주는 함수이다. 새로운 thread를 생성하는 **thread.c/thread_create()** 함수에서 thread를 initialize할 때 block state로 초기화했다가, 이 함수를 호출하며 해당 thread의 state를 unblock으로 바꿔주며 동시에 만든 thread를 ready_list의 적절한 위치에 넣어준다.
- Argument인 thread pointer *t가 가리키고 있는 thread의 status member로 접근해서 THREAD_READY state로 바꿔준다.
- list_push_back() 함수는 argument로 삽입할 list와 원소를 받아서 list의 맨 마지막에 해당 원소를 삽입해주는 함수이다. thread의 state를 unblock으로 바꿔주기 전에, list_push_back() 함수를 호출해서 t의 element를 ready_list의 맨 마지막에 삽입해주는 것을 확인할 수 있다.

- **2. thread.c/void thread_yield(void)**

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

- Running thread를 현재 thread에서 다른 thread로 넘기는 함수이다.
- thread_current() 함수를 통해 현재 running되고 있는 thread pointer를 전달받고, 현재 running되고 있는 thread가 idle_thread(main thread)가 아닌 경우 ready_list에 해당 thread의 element를 push_back한다. 그 뒤 threads.c/schedule() 함수를 호출하는데, 이 schedule 함수에서는 switch_threads 함수를 수행하여 running_thread를 cur에서 next로 바꾼다. 이 때 next thread는 thread.c/next_thread_to_run() 함수의 반환값인데, 이 함수 내에서는 결국에 ready_list를 pop_front하여 가져온다. 즉, 현재로서는 우선순위를 고려한 yield가 아니라, 해당 함수가 실행되면 push_back으로 계속해서 쌓인 ready_list를 pop_front하여 가장 먼저 들어온 thread로 CPU 점유권을 넘겨주고 있는 것이다.

B. Priority Donate

Problem Description

- Priority Scheduling을 구현할 때, 위에서 push_back으로 ready_list를 채우던 것을 priority 순으로 running thread를 결정하는 것을 구현 완료했다면 Priority Inversion이라는 상황을 고려해야 한다.
- 현재로서는 Priority scheduling이 진행되고 있지 않기 때문에 priority donate와 관련된 로직이 존재하지 않아서, Design 탭에서 Priority scheduling에서 발생할 수 있는 문제와 개선해야 할 사항을 다루는 것으로 한다.

2-3. Advanced Scheduler

Problem Description

- 위에서 구현한 priority scheduler는 priority가 낮은 thread는 CPU를 점유하기 힘들어 Average response time 이 커질 우려가 있다. priority donation에서 필요한 thread에 대해 priority를 적절하게 수정하긴 하지만 이 마저도 수정하지 못하는 경우가 생길 수 있다.
- 이 문제를 해결하기 위해 Advanced Scheduler는 priority donation을 비활성화 하고 mlfqs(Multi-Level Feedback Queue Scheduler) scheduling 방식을 통해 priority를 실시간으로 수정하려고 한다.

관련 함수 설명

- **1. thread.c/void thread_set_priority(int new_priority)**

  현재 thread의 priority 멤버 변수에 접근하여, 이를 new_priority 값으로 바꾸는 함수이다. 하지만 advanced scheduler에서는 priority 값을 임의로 바꾸지 못한다는 조건이 있어 이를 반영해야 한다.

- **2. synch.c/void lock_acquire(struct lock *lock)**

  - Priority scheduler에서는 기존의 priority_donation()를 통해 priority Inversion 상황을 관리했다. 하지만 mlfqs 기반의 advanced scheduler에서는 priority donation 함수를 이용하지 않으므로 이를 반영해야 한다.

- **3. synch.c/void lock_release(struct lock *lock)**

  - lock_acquire()과 마찬가지로 Priority scheduler에서는 기존의 priority_donation()를 통해 priority Inversion 상황을 관리했다. 하지만 mlfqs 기반의 advanced scheduler에서는 priority donation 함수를 이용하지 않으므로 이를 반영해야 한다.

- **4. timer.c/static void timer_interrupt(struct intr_frame *args UNUSED)**

  timer interrupt를 구현한 함수. 1 tick가 증가할 때마다 실행된다.

3. Design

3-1. Alarm Clock

Algorithm, Rationale

- Thread가 생성되면 Ready State가 아닌 Block State로 설정하고, 각 Thread마다 일어날 시간을 tick단위로 저장한다. 그리고 sleep_list에 일어나야 하는 시간을 기준으로 오름차순으로 정렬되도록 추가한다. sleep_list에 있는 값들을 비교하며 깨어날 시간인 Thread를 Ready State로 변경해 준다.
- 이 방식은 기존과는 다르게 Block State인 sleep_list만 확인할 수 있으며, 매 tick마다 스케줄링을 하지 않아도 되므로 효율성이 올라간다. 또한 sleep_list를 깨어날 시간에 대해 오름차순으로 정렬한다면 더욱 효율적으로 thread를 관리할 수 있을 것이다.
- 이를 디자인하기 위해 기존 함수를 수정한 부분과 새로운 함수를 구현한 부분을 나누어 설명한다.

기존 함수 수정

- **1. thread.h/struct thread**
  - thread struct에 일어날 시간을 tick단위로 저장할 변수인 **int64_t** wake_up_tick를 추가한다.
- **2. thread.c/list sleep_list**
  - sleep 상태인 thread를 저장하는 list인 sleep_list를 추가한다.
- **3. thread.c/thread_init**
  - ready_list, all_list 처럼 새롭게 추가한 sleep_list 역시 list_init 함수를 통해 초기화해 준다.
- **4. timer.c/timer_sleep**
  - 기존에 모든 Ready State Thread를 확인하는 while문을 제거하고, 뒤에 언급할 추가 구현 함수 thread_sleep를 추가해 기존 busy waiting 방식을 효율적인 방식으로 수정한다.

추가 구현

- **1. thread.c/void thread_sleep(int64_t ticks)**
  - 현재 thread를 sleep 상태로 변환하는 함수이다. **thread_current()** 함수로 현재 thread를 받아오고, Block State로 변환해야 한다. 이때 interrupt가 발생하면 안되므로 interrupt를 해제해야 한다. 그리고 sleep state thread 만 모아두는 sleep_list

에 기존 코드에 있는 함수인 **list_insert_ordered** 함수를 통해 오름차순으로 정렬되도록 넣는다. 그리고 thread를 Block State로 변환하고 다시 interrupt를 다시 받을 수 있도록 처리한다.

- **2. thread.c/void thread_awake(int64_t ticks)**
  - sleep_list에 있는 sleep 상태의 thread 중 일어날 시간이 된 thread를 다시 ready state로 변환하는 함수이다. thread_sleep함수에서 이미 sleep_list를 일어날 시간을 기준으로 오름차순 정렬을 하였기 때문에 가장 앞 원소의 일어날 시간만 비교하면 된다. 만약 일어날 시간이 현재 시간보다 작으면 thread를 Ready State로 변환한다

3-2. Priority Scheduler

A. Priority Scheduling

Algorithm, Rationale

- 우선, thread_yield와 thread_unblock 함수에서 단순히 push_back() 함수를 호출하고 있는 부분을 우선순위를 고려하도록 바꿔준다.
- thread 간의 우선순위를 고려하기 위해서, 단순 push_back이 아니라 thread를 삽입할 때 priority의 내림차순 위치로 삽입하는 알고리즘을 디자인했다. 또한, 각 thread들의 priority가 변할 수 있으므로 ready_list나 semaphore의 waiter_list를 순회하면서 thread들을 unblock해주는 경우 thread list를 priority에 대한 내림차순으로 정렬한 뒤 순회하도록 바꿔준다.
- 이를 디자인하기 위해 기존 함수를 수정한 부분과 새로운 함수를 구현한 부분을 나누어 설명한다.

기존 함수 수정

- **1. thread.c/void thread_unblock(struct thread *t)**
  - 기존에 list_push_back 함수를 호출했던 부분을, 아래서 설명할 list_push_priority 함수를 호출하도록 바꿔준다.
- **2. thread.c/void thread_yield(void)**

```
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());

  old_level = intr_disable ();
  if (cur != idle_thread)
    list_push_back (&ready_list, &cur->elem);
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

- Running thread를 현재 thread에서 다른 thread로 넘기는 함수이다.
- thread_current() 함수를 통해 현재 running되고 있는 thread pointer를 전달받고, 현재 running되고 있는 thread가 idle_thread(main thread)가 아닌 경우 ready_list에 해당 thread의 element를 push_back한다. 그 뒤 threads.c/schedule() 함수를 호출하는데, 이 schedule 함수에서는 switch_threads 함수를 수행하여 running_thread를 cur에서 next로 바꾼다. 이 때 next thread는 thread.c/next_thread_to_run() 함수의 반환값인데, 이 함수 내에서는 결국에 ready_list를 pop_front하여 가져온다. 즉, 현재로서는 우선순위를 고려한 yield가 아니라, 해당 함수가 실행되면 push_back으로 계속해서 쌓인 ready_list를 pop_front하여 가장 먼저 들어온 thread로 CPU 점유권을 넘겨주고 있는 것이다.

새 함수 디자인

- **1. void list_push_priority**
  - thread 실행 순서를 관리하는 ready_list를 항상 priority 내림차순으로 정렬된 상태를 유지할 수 있도록 void list_push_priority 함수를 구현한다. 이 때, 기존의 list_push_back 함수가 작동했던 것처럼 아래 언급할 priority_compare 함수를 argument로 넣어주어서 우선순위 비교 알고리즘으로 활용한다.
- **2. bool priority_compare(thread *t1, thread *t2)**
  - 이 과정에서 두 thread pointer를 input으로 받아서, 각 thread의 priority에 접근해 비교하는 priority_compare() 함수가 필요하다. 예를 들어 thread1의 우선순위가 thread2의 우선순위보다 높다면 1(true)를 반환하고, thread1의 우선순

위가 thread2의 우선순위보다 낮다면 0(false)를 반환하는 함수이다. 기존의 list_push_back 함수에서 argument로 함수를 받았던 것처럼, list_push_priority 함수에서는 priority_compare 함수를 argument로 받아서 list에서 각 thread 간의 순서를 적절하게 유지해주는 역할을 하게 될 것이다. 즉 priority_compare 함수가 true인 동안 ready_list를 순회하며 삽입할 위치를 찾는 알고리즘을 제안한다.

B. Priority Donate

- Priority Schedule을 올바르게 구현했을 때 발생할 수 있는 문제인 priority donate를 구현한다.
- semaphore와 lock acquire/release로 인해 일어날 수 있는 상황에 대해서도 고려하여 구현한다.
- 이를 디자인하기 위해 기존 함수를 수정한 부분과 새로운 함수를 구현한 부분을 나누어 설명한다.

기존 함수 수정

- **1. thread.h/struct thread**

```
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. */

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

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

  - Priority donation은 해당 thread가 lock에 걸렸을 때 일어날 수 있는 상황이므로 thread에서 자신이 lock을 기다리고 있는지 확인할 수 있어야 하는데, 현재의 thread structure에는 lock을 고려할 수 있는 방법이 없다. 따라서 이를 고려할 수 있도록 현재 해당 thread가 lock을 기다리는지를 나타내는 **lock***

wait_on_lock을 멤버 변수로 추가한다. 또한, donation이 발생할 경우 priority를 나눠주기 전 상태의 priority를 기억하기 위해 **init_priority**를 멤버 변수로 추가한다.

- 또한, donate한 priority를 나중에 회수하기 위해서는 어떤 thread에 priority를 donate했는지를 기억해야 한다. 이를 위해서 thread sturcture의 멤버 변수로 **struct list** donation_list와 **struct list_elem** donated_elem을 추가한다.

- **2. thread.c/static void init_thread (struct thread *t, const char *name, int priority)**
  - Priority donation을 고려해주기 위해서 **struct thread**에 새로 추가한 멤버 변수들을 초기화한다. (NULL, 0 등, init_priority는 입력받는 priority로 initialize)
- **3. thread.c/void thread_set_priority(int new_priority)**

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
  thread_current ()->priority = new_priority;
}

/* Returns the current thread's priority. */
int
thread_get_priority (void)
{
  return thread_current ()->priority;
}
```

  - 현재 thread의 priority 멤버 변수에 접근하여, 이를 new_priority 값으로 바꾸는 함수이다. 이로 인해 현재 실행되고 있는 thread의 priority가 더 이상 최고가 아닐 수 있으므로, 이를 확인하고 priority_yield 함수를 호출하도록 수정한다.
- **4. thread.c/int thread_get_priority(void)**
  - 현재 thread의 priority를 반환하는 함수이다.
- **5. synch.c/void sema_down(struct semaphore *sema)**
  - 현재 코드에서는 semaphore가 0인 동안 semaphore의 waiter list에 삽입하고 있는데, 기존에 **list_push_back()**으로 삽입하던 것을 앞서 priority scheduler에

서 구현한 **list_priority_push()** 함수로 대신한다.

- **6. synch.c/void sema_up(struct semaphore *sema)**
  - thread들이 semaphore의 waiter_list 안에 있는 동안 Priority가 변경될 수 있으므로 이를 고려하여 각 thread들을 unblock 해 주기 전에 waiters list를 우선순위로 정렬한다.
- **7. synch.c/void cond_wait(struch condition *cond, struct lock *lock)**
  - sema_down 함수와 마찬가지로, 기존에 **list_push_back()**으로 waiter_list에 thread들을 넣어주던 것을 **list_priority_push()** 로 대신한다.
- **8. synch.c/void cond_signal(struch condition *cond, struct lock *lock, aux UNUSED)**
  - cond_wait() 함수와 마찬가지로, 기존에 **list_push_back()**으로 waiter_list에 thread들을 넣어주던 것을 **list_priority_push()** 로 대신한다.
- **9. synch.c/void lock_acquire(struct lock *lock)**
  - 현재의 코드에서는 다른 thread가 lock을 점유 중인 상황을 고려하지 않는다. 이를 argument로 들어오는 lock에 접근하여 lock->holder가 NULL이 아닌지를 확인함으로써 위 상황을 고려하고, 해당 상황일 경우 아래에 새로 언급할 **priority_donation()** 함수를 호출한다. 이후 점유 중이었던 thread가 lock을 release하여 **lock_acquire()**을 호출한 thread가 cpu를 점유하게 될 경우 argument의 lock->holder로 접근하여 이 값을 current thread로 바꿔준다.
- **10. synch.c/void lock_release(struct lock *lock)**
  - lock을 release하기 때문에 donation list에서 해당 thread를 제거하고, 새롭게 빌려주었던 priority를 회수하여 새롭게 우선순위를 정의하는, 후에 언급할 **priority_newly_set()** 함수를 호출한다.

새 함수 구현

- **1. thread.c/void priority_donation(void)**
  - priority donation을 수행하는 함수이다. current_thread에 접근해서, 앞서 수정된 struct thread의 멤버 변수인 wait_on_lock->holder로 접근한 뒤 해당 holder의 priority를 current thread의 priority로 대입해준다. 이 때 current thread의 wait_on_lock이 null이라면 priority donation이 필요하지 않다는 뜻이

므로 함수를 빠져나오면 된다. 위의 과정을 현재 thread가 기다리는 lock의 waiter list에 있는 모든 thread에 대해 진행한다. nested priority를 처리하기 위해 current thread를 holder thread로 바꿔주며 이 과정을 반복하고, 이 때 document에서 이야기한 사항처럼 깊이에 대한 제한을 8로 둘 수 있다.

- **2. thread.c/void remove_lock(struct lock *lock)**
  - lock을 release했을 때 호출되어야 하는 함수이다. current thread의 donation_list를 순회하면서, 만약 donation_list의 원소인 각 thread들이 기다리고 있는 lock이 (즉, thread->wait_on_lock) 방금 release된 lock이라면 해당 thread를 삭제한다.
- **3. thread.c/void priority_newly_set(void)**
  - 다른 thread에 donate했던 current thread의 priorty를 회수한다. list donation_list에 여러 priority가 있을 경우 그 값 중 가장 큰 값을 선택하여 되돌린다.

3-3. Advanced Scheduler

Algorithm, Rationale

- Pintos document를 보면 mlfqs에서 priority를 계산하기 위한 변수들이 제시되어 있다. (niceness / recent_cpu / load_avg). 각 변수들의 계산식은 다음과 같다.

1. Priority

$$priority = PRI\_MAX - (recent\_cpu / 4) - (nice * 2)$$

2. recent_cpu

$$recent\_cpu = (2 * load\_avg)/(2 * load\_avg + 1) * recent\_cpu + nice$$

3. load_avg

$$load\_avg = (59/60) * load\_avg + (1/60) * ready\_threads$$

- 이 중 **recent_cpu**와 **load_avg**는 정수가 아닌 실수 값인데, Pintos에서는 부동소수점 연산을 지원하지 않는다. 이 두 변수를 표현하기 위해 고정 소수점(fixed-point) 연산을 구현해야 한다. Pintos document에 고정소수점을 표현하기 위해 해야 하는 보정식이 있다. 이를 구현해야 한다.

기존 함수 수정

- **1. thread.c/struct thread**
  - priority를 계산하기 위한 nice, recent_cpu, load_avg 변수를 추가한다. mlfqs scheduling 여부를 판별하는 bool 형식의 thread_mlfqs 변수를 추가한다
- **2. thread.c/void thread_set_nice(int nice UNUSED)**
  - nice값을 set하는 함수이다. thread를 set 시 interrupt를 비활성화해야 한다.
- **3. thread.c/int thread_get_nice(void)**
  - 현재 thread의 nice값을 return하는 함수를 구현해야 한다.
- **4. thread.c/int thread_get_load_avg(void)**
  - 현재 thread의 load_avg값에 100을 곱한 값을 return하는 함수를 구현해야 한다.
- **5. thread.c/void thread_get_recent_cpu(void)**
  - 현재 thread의 recent_cpu값에 100을 곱한 값을 return하는 함수를 구현해야 한다.
- **6. thread.c/static void init_thread(struct thread *t, const *name, int priority)**
  - 새롭게 추가해준 struct thread의 멤버 변수 nice와 recent_cpu를 초기화해야 한다.
- **7. thread.c/void thread_start(void)**
  - 새롭게 추가해준 struct thread의 멤버 변수 load_avg를 초기화 해야 한다.
- **8. thread.c/void thread_set_priority(int new_prioirty)**
  - mlfqs 기반 scheduling을 할 때는 priority donation을 비활성화해야 한다.
- **9. sync.c/void lock_acquire(struct lock *lock), void lock_release(struct lock *lock)**
  - mlfqs 기반 scheduling을 할 때는 priority donation을 비활성화해야 한다.
- **10. timer.c/static void timer_interrupt(struct intr_frame *args UNUSED)**
  - mlfqs 기반 scheduling을 할 때 4tick마다 priority를, 1초마다 recent_cpu, load_avg를 다시 계산한다.

새 함수 구현

- **1.thread.c/mlfqs_calculate_priority(struct *thread t),**

mlfqs_calculate_recent_cpu(struct *thread t), mlfqs_calculate_load_avg()

- priority, recent_cpu, load_avg를 계산하는 함수를 구현해야 한다. 계산 로직은 위의 수식을 기반으로 한다.

- **2. thread.c/mlfqs_increment_recent_cpu()**

  - mlfqs 기반의 scheduling을 할 때 recent_cpu를 증가시키는 함수를 구현해야 한다.

- **3. thread.c/mlfqs_priority(struct *thread t), mlfqs_recent_cpu(struct *thread t), mlfqs_load_avg()**

  - mlfqs 기반의 scheduling을 할 때 모든 thread의 priority, recent_cpu, load_avg 값을 다시 계산하는 함수를 구현해야 한다.

- **4. fixed_point.h/int int_to_fp(int N), int fp_to_int(int X), int fp_to_int_round(int X), int add_fp_fp(int X, int Y), int add_fp_int(int X, int N), int sub_fp_fp(int X, int Y), int sub_ fp_int(int X, int N), int multi_fp_fp(int X, int Y), int multi_ fp_int(int X, int N), int div_fp_fp(int X, int Y), int div_ fp_int(int X, int N)**

  - 실수인 recent_cpu, load_avg 값을 계산하기 위해 실수와 실수 계산, 실수와 정수 계산을 구현한 함수와 실수를 정수로, 정수로 실수로 변환하는 함수를 구현해야 한다. 이는 사전 정의된 fixed point와 간단한 arithmetic operation으로 구현 가능하다.