**Pintos Project 3: Threads**

담당 교수 : 박성용

학번 / 이름 : 20181687 / 정지석

개발 기간 : 10/29 ~ 11/12

1. **개발 목표**

기본적으로 구현되어 있는 thread system 부분에 수업에서 다루었던 기능들을 구현해 준다. Alarm clock을 통해 thread들을 좀더 효율적으로 관리하고, priority scheduling을 통해 우선순위가 더 높은 thread가 먼저 실행될 수 있게 해준다. 여기에 추가적으로 multi-level ready queue를 사용하는 BSD scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
   2. Alarm Clock

Thread들은 사용되지 않을 경우 sleep 상태가 되는데, 이전까지는 이 sleep 상태인 thread 들을 깨울 때(wake up) 할당된 시간이 지났는지를 확인하기 위해 반복문을 통해 running state와 ready state를 오가는 방식을 사용하였다. 해당 방식은 대단히 비효율적이기 때문에 sleep 상태에 들어간 thread들을 따로 queue에 저장하여 할당 시간이 지나면 깨워 주는 alarm clock 방식을 사용하여 thread들을 좀더 효율적으로 관리해준다.

* 1. Priority Scheduling

각 thread에 priority를 부여하여 priority 값이 클수록 높은 우선순위를 가지게 된다. Ready list 우선순위가 높은 순서대로 정렬한다. 또한 현재 실행 중인 thread와 ready list 맨 앞에 있는 thread의 priority를 비교하고, 만약 ready list에 있는 thread의 우선순위가 더 높다면, 해당 thread가 먼저 실행된다. 이 때 우선순위가 낮은 thread들은 계속 실행되지 못하는 ‘starvation’ 문제가 발생할 수 있는데, 이를 방지하기 위해 ready list에 있는 시간이 길어질수록 priority 값이 커지게 하는 ‘aging’ 기법을 사용한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Multi-level ready queue를 사용하여 thread들을 관리하여 average response time을 줄여준다. 이를 구현하기 위해 recent cpu, nice, priority, load average 값을 구해야 한다. 이 값들을 구할 때 사용하는 사칙연산들은 fixed point를 통해 따로 함수를 만들어 직접 구현해 주어야한다(pintos에서는 floating point를 지원해주지 않기 때문이다.).

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

먼저 Blocked 상태의 thread들을 따로 저장할 sleep\_list 선언해준다. 이 list에 blocked 상태의 thread들을 넣어주고, 이 때 깨울 시간, 즉 wake up을 해주어야 할 시간도 함께 저장한다. 이렇게 한 후 매 tick 마다 깨워야 할 thread가 있는지 확인해주고, 만약 존재한다면 해당 thread를 sleep\_list에서 ready\_list로 옮겨준 후, thread의 state를 ready 상태로 바꿔준다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.

Ready\_list는 기본적으로 priority 값이 높은 순으로 정렬이 되기 때문에 ready\_list 맨 앞의 thread와 현재 실행 중인 running thread의 priority 값을 비교해주면 위의 문제를 해결할 수 있다. 만약 ready\_list에 running thread 보다 높은 우선순위를 가진 thread가 들어오게 된다면, 먼저 thread\_yield() 함수를 통해 현재 실행 중인 running thread를 중지시킨 후 ready\_list에 넣어준다. 그리고 높은 우선순위를 가진 ready\_list에 있던 thread가 running thread가 된다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

BDS scheduler를 구현하려면 매 tick마다 Multi-level ready queue에 있는 모든 thread들의 priority를 계산해야 한다. Priority를 구하는 공식은 아래와 같다.

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

위의 공식에서 recent\_cpu 값을 구하기 위해서는 아래와 같은 계산이 필요하다.

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

위의 공식에서 load\_avg 값을 구하려면 아래의 계산이 필요하다.

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

여기서 recent\_cpu 값은 해당 thread가 가장 최근에 사용한 CPU time을 의미한다. 위 공식을 통해 확인할 수 있듯이 recent\_cpu 값이 커지면 해당 thread의 priority 값이 작아지게 된다. 이는 다시 말해 가장 최근에 실행된 thread일수록 우선순위에서 밀려나게 된다는 의미이다.

그런데 위의 공식들은 실수&실수, 실수&정수 사이의 사칙 연산을 많이 사용하는 것을 볼 수 있다. 그런데 pintos의 경우 floating point를 제공하지 않는다. 이럴 경우 위의 연산들이 제대로 계산되지 않기 때문에 직접 fixed point를 사용하여 위의 연산들을 함수로 직접 구현해 주어야 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/29 ~ 11/02: alarm clock 구현

11/03 ~ 11/07: priority scheduler 구현

11/07 ~ 11/11: BSD scheduler 구현

11/12: 보고서 작성

* 1. **개발 방법**

수정해야하는 소스코드

1. Alarm Clock

devices/timer.c

timer\_sleep() 함수의 경우 기존에는 반복문(while 문)을 사용하여 running state와 ready state를 오가는 방식을 사용하였다. 이를 수정하기 위해 반복문을 삭제하고 thread.c에 새로 선언한 thread\_sleep() 함수를 통해 thread를 sleep\_list에 넣어 관리한다.

Timer\_interrupt() 함수에서는 매 tick마다 sleep\_list에서 wake up 해야 할 thread가 있는지를 확인해야 한다. 이는 thead.c에 새로 구현한 thread\_awake() 함수를 통해 확인할 수 있다.

threads/threa.h

thread 구조체 안에 각 thread가 얼마동안 sleep 상태에 놓여있어야 하는지를 명시하는 wake\_up 정수형 필드를 선언해준다.

threads/threa.c

sleep 상태의 thread들을 따로 관리해 줄 sleep\_list를 선언해준다. 그 후 thread\_init() 함수에서 sleep\_liset를 initialize해준다.

1. Priority Scheduling

thread/init.c

parse\_option() 함수에 aging 관련 부분을 추가한다.

threads/sync.c

sema\_up() 함수를 수정하여 thread의 priority 값이 변경되었을 경우를 고려하여 waiters를 sorting해준다. 또한 thread.c에서 새로 선언한 ready\_running\_priority() 함수를 통해 실행 중인 thread와 ready 상태인 thread 간의 우선순위를 비교한다.

threads/thread.c

aging 기법을 사용하기 위해 thread\_aging 변수를 추가해 준다.

thread\_create() 함수에서 thread를 초기화할 때 ready\_running\_priority() 함수를 추가하여 실행 우선순위에 변화가 있는지를 확인한다.

thread\_yield() 함수에서는 list\_push\_back 함수를 list\_insert\_ordered 함수로 바꿔 새로운 thread가 들어왔을 때 우선 순위에 맞게 ready\_list안에 정렬될 수 있도록 한다.

thread\_unblock() 함수에서도 thread\_yield()와 마찬가지로 list\_push\_back 함수를 list\_insert\_ordered 함수로 바꿔 unblocked 된 thread가 우선 순위에 맞게 ready\_list안에 정렬될 수 있도록 한다.

thread\_set\_priority() 함수에서는 ready\_running\_priority() 함수를 통해 우선순위에 변화가 생겼을 경우 이에 따른 변경이 있도록 한다.

devices/timer.c

thread\_aging 값이 true일 때, 4 tick마다 thread.c에 선언한 update\_priority() 함수를 통해 priority를 update 해준다.

1. Advanced Schedule

threads/thread.h

thread 구조체 안에 recent\_cpu 값과 nice 값을 추가해 준다.

threads/thread.c

init\_thread() 함수에 nice와 recent\_cpu 값을 초기화하는 부분을 추가한다.

devices/timer.c

thread\_mlfqs가 true일 경우 매 tick마다 recent\_cpu 값을 1씩 늘려주는 부분을 추가한다. 4 tick마다 thread.c에 선언한 update\_priority() 함수를 통해 priority를 update 해준다.

추가해야 하는 함수/자료구조

1. Alarm Clock

threads/thread.c

thread\_sleep() 함수를 통해 현재 thread를 sleep\_list에 넣어준 뒤 block 상태로 만들어 준다.

thread\_awake() 에서는 wake up 해주어야 할 thread를 sleep\_list에서 제거하고 unblock 상태로 만들어 준다.

1. Priority Scheduling

threads/thread.c

ready\_running\_priority() 함수를 통해 실행 중인 thread와 ready 상태인 thread 간의 우선순위를 비교한 후, ready\_list의 thread가 우선순위가 더 높다면, running 중인 thread를 중단하고 ready\_list에 있는 thread를 running 상태로 만들어준다.

priority\_compare() 함수는 list\_insert\_ordered() 함수를 사용할 때 두 thread간에 우선순위를 비교하기 위해 사용되는 함수이다.

max\_priority() 함수는 ready\_list에 있는 thread 중 가장 높은 우선순위를 가진 thread의 priority 값을 return하는 함수이다.

update\_priority() 함수는 모든 thread의 우선순위를 다시 update 해준다.

1. Advanced Schedule

threads/thread.c

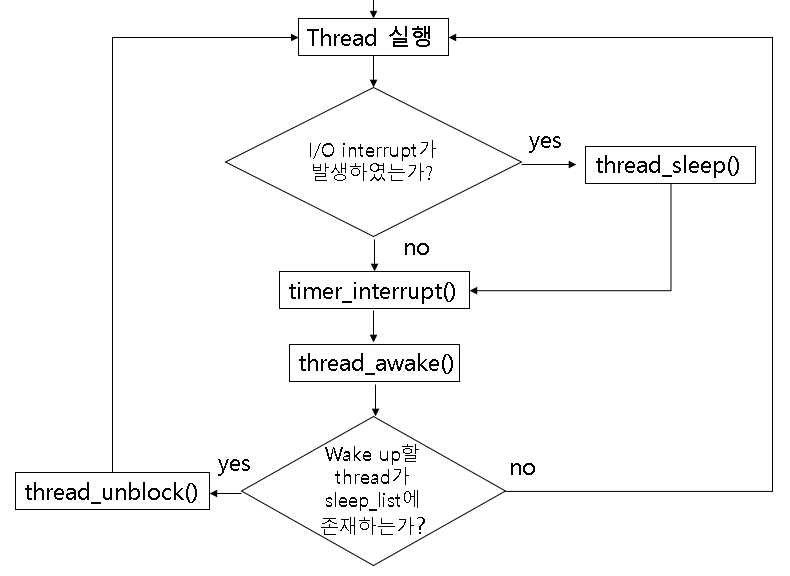
load\_avg\_recent\_cpu() 함수를 통해 load\_avg 값과 recent\_cpu 값을 구한다.

priority() 함수를 통해 위에서 구한 값을 토대로 priority 값을 계산한다.

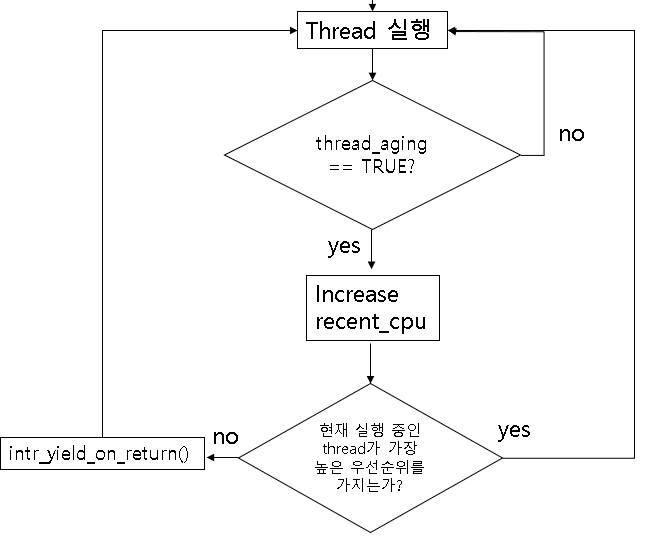
threads/fixed-point.c

pintos에서 floating point를 지원하지 않기 때문에 위의 공식들을 계산하기 위해서는 직접 함수들을 만들어 주어야 한다. fixed-point.h & fixed-point.c 파일을 생성한 뒤, 여러 상황에 맞는 사칙연산 함수들을 작성한다(이는 아래에서 자세히 다룰 예정이다.).

1. **연구 결과**
   1. **Flow Chart**
      1. **Alarm Clock**

****

* + 1. **Priority Scheduling**

****

* 1. **제작 내용**
     1. Alarm Clock

\*threads/thread.h

  int64\_t wake\_up;

thread 구조체에 wak\_up 변수를 추가한다.

\*devices/timer.c

timer\_sleep()

  thread\_sleep(ticks + start);

thread\_sleep()을 통해 block 상태의 thread들을 따로 저장해준다.

timer\_interrupt()

  thread\_awake(ticks);

thread\_awake 함수를 통해 wake up해야 할 thread가 있는지 확인한다.

\*threads/thread.c

static struct list sleep\_list;

block 상태의 thread들을 따로 관리할 list를 정의한다.

thread\_init()

  list\_init(&sleep\_list);

sleep\_list도 initialize 해준다.

thread\_sleep()

void thread\_sleep(int64\_t *t*)

{

  enum intr\_level old = intr\_disable();

  struct thread \*current = thread\_current();

  ASSERT(!intr\_context());

  ASSERT(current != idle\_thread);

  current->wake\_up = t;

  list\_push\_back(&sleep\_list, &current->elem);

  thread\_block();

  intr\_set\_level(old);

}

Idle\_thread가 아닌 thread를 sleep\_list에 넣어주고 block 상태로 만들어주는 함수이다.

thread\_awake()

void thread\_awake(int64\_t *t*)

{

  struct thread \*current;

  struct list\_elem \*ptr = list\_begin(&sleep\_list);

  struct list\_elem \*end = list\_end(&sleep\_list);

*while* (1)

  {

*if*(ptr == end)

*break*;

    current = list\_entry(ptr, struct thread, elem);

*if* (current->wake\_up <= t)

    {

      ptr = list\_remove(ptr);

      thread\_unblock(current);

    }

*else*

      ptr = list\_next(ptr);

  }

}

Wake up time이 tick보다 작은 thread 들을 wake up 한 뒤 sleep\_list에서 제거해주고, unblock 상태로 만들어 ready\_list에 넣어준다.

* + 1. Priority Scheduling

\*threads/threa.c

thread\_unblock() 함수에서 기존에 사용하던 list\_push\_back() 함수 대신에 아래와 같이 list\_insert\_ordered() 함수를 사용한다.

  list\_insert\_ordered(&ready\_list, &t->elem, priority\_compare, 0);

이때 추가적으로 구현한 priority\_compare() 함수를 통해 우선순위에 따라 ready\_list에 넣어주게 된다.

bool priority\_compare(struct list\_elem \**x*, struct list\_elem \**y*, void \*aux *UNUSED*)

{

*return* list\_entry(x, struct thread, elem)->priority > list\_entry(y, struct thread, elem)->priority;

}

void ready\_running\_priority()

{

*if* (list\_empty(&ready\_list))

*return*;

  struct thread \*ready = list\_entry(list\_begin(&ready\_list), struct thread, elem);

  struct thread \*running = thread\_current();

*if* (ready->priority > running->priority)

    thread\_yield();

}

max\_priority() 함수를 통해 ready\_lsit에서 가장 높은 우선순위를 가진 thread의 priority 값을 return한다.

int max\_priority()

{

  int priority = -1;

  struct thread \*tmp;

*if* (!list\_empty(&ready\_list))

  {

    tmp = list\_entry(list\_front(&ready\_list), struct thread, elem);

    priority = tmp->priority;

  }

*return* priority;

}

update\_priorty() 함수를 통해 매 4 tick마다 전체 thread의 우선순위를 update 해준다.

void update\_priority()

{

  struct list\_elem \*ptr = list\_begin(&all\_list);

  struct list\_elem \*end = list\_end(&all\_list);

*while* (ptr != end)

  {

    struct thread \*tmp = list\_entry(ptr, struct thread, allelem);

    priority(tmp);

    ptr = list\_next(ptr);

  }

*if* (thread\_current()->priority < max\_priority())

      intr\_yield\_on\_return();

}

* + 1. Advanced Schecule

\*threads/thread.c

load\_avg\_recent\_cpu() 함수에서는 load\_avg 값과 recent\_cpu 값을 구하는 작업을 수행한다.

void load\_avg\_recent\_cpu(void) {

  int ready\_threads = list\_size(&ready\_list);

  struct thread\* tmp;

  struct list\_elem \*ptr = list\_begin(&all\_list);

  struct list\_elem \*end = list\_end(&all\_list);

*if* (thread\_current() != idle\_thread)

    ready\_threads += 1;

  load\_avg = fp\_div\_int(fp\_add\_int(int\_mul\_fp(59, load\_avg),ready\_threads),60);

*while* (ptr != end) {

    tmp = list\_entry(ptr, struct thread, allelem);

*if* (tmp != idle\_thread)

        tmp->recent\_cpu = fp\_add\_int(fp\_mul\_fp(fp\_div\_fp(int\_mul\_fp(2, load\_avg), fp\_add\_int(int\_mul\_fp(2, load\_avg), 1)), tmp->recent\_cpu), tmp->nice);

    ptr = list\_next(ptr);

  }

}

위에서 구한 값을 토대로 priority() 함수에서 priority 값을 구한다.

void priority(struct thread \**t*)

{

   t->priority = fp\_sub\_fp(fp\_sub\_fp(fp\_add\_int(0, PRI\_MAX), fp\_div\_int(t->recent\_cpu, 4)), int\_mul\_fp(2, fp\_add\_int(0, t->nice))) / fraction;

*if* (t->priority > PRI\_MAX)

    {

      t->priority = PRI\_MAX;

    }

*if* (t->priority < PRI\_MIN)

    {

      t->priority = PRI\_MIN;

    }

}

그런데 위에서 살펴본 3개의 공식을 계산하기 위해서는 소수점 계산을 해주어야 한다. 하지만 pintos의 경우 floating point를 지원하지 않는다.

이를 해결하기 위해 직접 fixed point 연산을 구현해 주어야 한다.

위 3개의 공식에서 사용하는 실수&실수, 실수&정수 연산은

정수 - 실수  
정수 \* 실수  
실수 + 정수  
실수 \* 실수  
실수 / 실수  
실수 + 실수  
실수 - 실수  
실수 / 정수

가 있다. 이를 fixed-point.c라는 파일을 새로 생성해 모두 구현해주면 다음과 같다.

int fp\_add\_fp(int *a*, int *b*)

{

*return* a + b;

}

int fp\_add\_int(int *a*, int *b*)

{

*return* a + (b \* fraction);

}

int int\_sub\_fp(int *a*, int *b*)

{

*return* (a \* fraction) - b;

}

int fp\_sub\_fp(int *a*, int *b*)

{

*return* a - b;

}

int fp\_mul\_fp(int *a*, int *b*)

{

    int64\_t tmp = a;

    tmp = tmp \* b;

    tmp = tmp / fraction;

*return* (int)tmp;

}

int fp\_mul\_int(int *a*, int *b*)

{

*return* a \* (b \* fraction);

}

int int\_mul\_fp(int *a*, int *b*)

{

*return* a \* b;

}

int fp\_div\_fp(int *a*, int *b*)

{

    int64\_t tmp = a;

    tmp = tmp \* fraction;

    tmp = tmp / b;

*return* (int)tmp;

}

int fp\_div\_int(int *a*, int *b*)

{

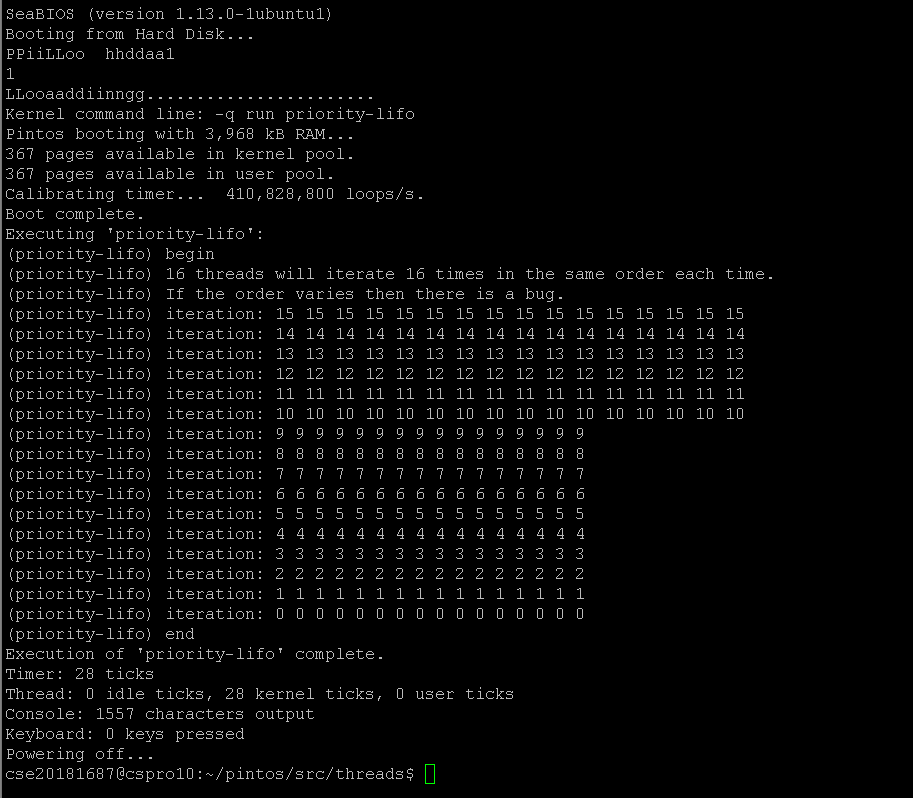
*return* a / b;

}

* 1. **시험 및 평가 내용**

priority-lifo.c 코드는 thread를 여러 개 생성하여 실행시키는 작업을 수행한다. LIFO의 형식으로 실행되기 때문에 우선순위가 높은 값이 나중에 들어왔다 하더라도 가장 먼저 출력된다.

아래의 사진을 통해 확인할 수 있듯이 총 16개의 thread가 각각 16번씩 반복 실행되는데, 우선순위가 높은 순서대로 차례대로 출력되는 것을 확인할 수 있다.



make check 수행결과는 아래와 같다.

