**Pintos Project 3 : Threads**

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 김영재 교수님

조 / 조원 : 46조, 20141284, 20141340

개발 기간 : 2019/11/16 ~ 2019/12/8

**프로젝트 제목 : Pintos Project 3 Threads**

**제출일 : 2019년 12월 8일**

**참여 조원 : 이기현, 장주호**

1. **개발목표**

- busy waiting으로 구현되어 비효율적인 Alarm clock 을 non-busy waiting 으로 구현하여 효율적으로 한다.

- thread scheduling이 큐를 이용한 round robin으로 구현되어 있는데 priority를 부여하고 priority가 높은 thread먼저 실행하도록 priority scheduling을 구현한다.

1. **개발 범위 및 내용**
2. **개발범위**

/threads/ thread.h thread.c init.c

/threads/ sync.c

/devices/ timer.c

* **Alarm clock**

기존에 alarm clock은 busy waiting 기법을 이용해 구현되어 있다. timer.c/timer\_sleep 의 함수를 busy waiting이 아닌 코드로 바꾸고 thread.c 에 매 틱마다 깨어나야 할 thread가 unblock이 되도록 도와준다.

* **Priority sceduling**

단순히 round robin으로 되어있는 scheduling을 priority scheduling으로 교체한다. thread.c의 unblock 함수와 yield 함수에서 ready 리스트로 thread를 넣을 때, 우선순위에 맞춰 정렬되어 들어가도록 한다. 또한 sema down 또는 up 을 할 때에도 priority가 맞도록 해준다.

* **Priority aging**

ready list의 thread들이 ready를 할 때 priority가 증가할 수 있도록 thread.c에 구현해준다.

1. **개발 내용**

**1) Alarm clock**

기존에 alarm clock은 busy waiting 기법을 이용하여 깨어날 tick을 기다리는 thread에게 계속해서 CPU를 할당해주기 때문에 비효율적으로 작동한다. 잠들어 있을 thread를 모아 놓는 queue를 만들어 thread를 넣고, 해당 틱이 되면 thread를 깨울 수 있게 구현한다.

**2) Priority scheduling**

pintos에서는 round robin 기법을 이용하여 scheduling이 구현되어 있다. 각 thread에 priority가 존재하고 이를 이용하여 priority가 높은 thread가 먼저 CPU할당을 받도록 구현해준다. 이를 위해 thread를 ready list에 넣을 때, priority가 높은 thread가 앞에 올 수 있도록 정렬된 상태로 넣는다.   
**3) Priority aging**

단순히 priority scheduling을 하면 starvation 현상이 일어날 수 도 있다. 그렇기 때문에 이를 방지하기 위해 ready list 내부의 thread들의 priority를 높여주는 aging을 구현한다.

1. **추진 일정 및 개발 방법**
2. **추진 일정**

11/16 (토) ~ 11/22 (금) : ppt 및 매뉴얼 분석

11/25 (월) ~ 11/28 (목) : Alarm clock 구현

11/30 (토) ~ 12/2 (월) : priority scheduling 구현

12/3 (화) ~ 12/5 (목) : priority aging & priority sema 구현

12/6 (금) ~ 12/7 (토) : 보고서 작성

1. **개발 방법**

* **Alarm clock**

기존에 busy-waiting 방식으로 구현되어있는 timer-sleep를 sleep\_list를 만들어서 일괄적으로 관리를 하게 해주었다.

Busy-waiting 방식에서는 매 tick마다 깨울지 말지 확인을 하지만 sleep\_list에 깨어나야 할 시간을 입력해서 리스트에 깨어나야 할 시간이 빠른 순서로 삽입하고 timer\_interrupt 가 호출될 때 마다 check\_tick()함수를 사용해서 현재 tick에서 깨어나야 할 thread가 있는지 확인한다.  
만약 깨어나야 할 thread가 존재한다면 sleep\_list를 앞에서부터 순회하면서 추가적으로 깨어나야 할 thread가 더 있는지 확인하고 깨운 후 종료한다.

* **Priority scheduling**

round robin으로 구현되어 있는 scheduling에 priority scheduling을 구현하여 준다.

우선 기존에 그냥 queue형식으로 구현되어 있던 ready\_list 를 prioriry\_queue처럼 정렬시켜 사용한다. 앞에서부터 priority가 높게 설정하여 list\_pop\_front로 가장 높은 priority를 갖는 thread를 꺼낸다. 그렇게 하기 위해 thread를 ready\_list에 넣을 때, list\_push\_back이 아닌 list\_insert\_ordered함수를 이용해 넣어주며 compare\_priority 함수를 인자로 넘겨 priority가 높은 thread가 앞에 올 수 있도록 해준다.

각 thread가 생성되면 thread\_create에서 priority를 멤버변수에 저장하는데 이 때, 현재 CPU에 할당된 thread의 priority보다 높다면 바꿔줄 수 있도록 한다.

또는 현재 thread의 priority가 바뀌었을 때 역시 ready list를 확인하여 새로운 priority가 더 낮다면 바꿔줄 수 있도록 한다.

* **Priority aging**

단순히 priority만 기준으로 잡으면 starvation 현상이 일어날 수도 있다.

그렇기 때문에 ready list 내부에 대기하는 thread들의 priority를 높여주도록 한다.

**연구원 역할 분담**

이기현 – priority sema & priority aging 구현

보고서 작성

장주호 – alarm clock & priority sceduling 구현

보고서 작성

1. **연구결과**
2. **합성 내용**

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

1. **제작 내용**

**/devices/timer.c**

**void timer\_sleep (int64\_t ticks)**

parameter로 받은 ticks 과 현재 tick을 더해서 thread\_sleep 함수의 인자로 전달한다.   
thread\_sleep에서는 전달받은 parameter를 해당 thread의 wakeup\_time으로 지정하고 sleep\_list에 삽입 후 block한다.

void timer\_sleep (int64\_t ticks) {

int64\_t start = timer\_ticks();

ASSERT (intr\_get\_level() == INTR\_ON);

thread\_sleep (start + ticks);

}

**void timer\_interrupt (struct intr\_frame \*args UNUSED)**

매 틱마다 check\_tick(ticks)를 호출해서 현재 tick에서 깨워야 하는 thread가 있는지 확인 후 존재시 깨우거나 종료한다.

void timer\_interrupt (struct intr\_frame \*args UNUSED) {

ticks++;

check\_tick (ticks);

thread\_tick ();

}

**/threads/thread.c**

**static struct list sleep\_list;**

잠들어있는 thread를 tick 이 짧은 순으로 저장할 queue를 선언해준다.

**void thread\_init (boid)**

전역으로 선언한 sleep\_list를 초기화 해준다.

list\_init (&sleeep\_list);

**void thread\_unblock (struct thread \*t)**

ready list 에서 thread를 뺄 때, priority 가 큰 priority 를 먼저 빼도록 하기 위해서 반대로 ready list에 넣을 때, priority가 높은 순으로 정렬되도록 넣어준다.

list\_push\_back으로 넣던 thread를 list\_insert\_ordered로 넣어준다.

void thread\_unblock (struct thread \*t) {

…

ASSERT (t->status == THREAD\_BLOCKED);

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

t->status = THREAD\_READY;

intr\_set\_level (old\_leve);

}

**void thread\_yield (void)**

thread\_yield에서도 역시 ready list로 현재 thread를 넣어준다.

마찬가지로 priority 순으로 정렬되도록 해주었다.

void thread\_yield (void) {

…

if (cur != idle\_thread)

list\_insert\_ordered(&ready\_list,&cur->elem,compare\_priority,0);

…

}

**void thread\_set\_priority (int new\_priority)**

thread의 priority를 새로 설정하는 함수이다.

만약 현재 ready\_list에 새로 설정한 priority보다 높은 priority를 갖고있는 thread가 있을 수도 있으므로 확인하여 있다면 교체해주는 check\_thread를 호출해준다.

void thread\_set\_priority (int new\_priority) {

thread\_current ()-> priority = new\_priority;

check\_thread ();

}

**void thread\_wake (int64\_t ticks)**

sleep list를 읽어 깨울 thread가 있다면 깨우는 함수를 새로 구현하였다.

현재 tick을 parameter로 받아 현재 tick보다 일찍 깨어야 했을 thread를 모조리 깨워 ready list로 넣어준다.

void thread\_wake (int64\_t ticks) {

struct list\_elem \*e;

struct thread \*tmp;

e = list\_begin(&sleep\_list);

tmp = list\_entry (e, struct thread, elem);

while (tmp->wakeup\_time <= ticks) {

e = list\_remove(&tmp->elem);

thread\_unblock(tmp);

tmp = list\_entry(e, struct thread, elem);

}

}

**void thread\_sleep (int4\_t ticks)**

block되어야 하는 현재 thread를 재운다.

sleep list에 넣어야 하며 더 일찍일어나야 하는 thread를 sleep list의 앞에 배치해야 하므로 list\_insert\_ordered를 이용하여 넣어준다. 또한, thread의 멤버변수 wakeup\_time에 인자로 받은 깨어야 하는 tick을 넣어준다.

void timer\_sleep (int64\_t ticks) {

struct thread \* cur;

enum intr\_level old\_level;

old\_level = intr\_disable();

cur = thread\_current();

cur->wakeup\_time = ticks;

list\_insert\_ordered(&sleep\_list, &cur->elem, compare\_ticks, 0);

thread\_block();

intr\_set\_level(old\_level);

}

**bool compare\_ticks (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)**

sleep list에 thread를 넣을 때 이용하는 compare function으로 thread의 wakeup\_time이 낮을수록 앞에 올 수 있도록 구현한다.

bool compare\_ticks (const struct list\_elem\* a, const struct list\_elem \*b, void \*){

return list\_entry(a, struct thread, elem)->wakeup\_time

< list\_entry(b, struct thread, elem)->wakeup\_time;

**bool compare\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)**

sleep list에 thread를 넣을 때 이용하는 compare function으로 thread의 wakeup\_time이 낮을수록 앞에 올 수 있도록 구현한다.

bool compare\_priority(const struct list\_elem\* a,const struct list\_elem \*b,void \*){

return list\_entry(a, struct thread, elem)->priority

< list\_entry(b, struct thread, elem)->priority;

**void check\_tick (int64\_t ticks)**

매 tick마다 호출되는 함수로 현재 sleep list의 가장 빨리 일어나야 하는 thread의 wakeup 시간과 현재 tick을 비교한다. 만약 일어나야 하는 thread가 있다면 thread\_wake를 호출해준다.

void check\_tick (int64\_t ticks) {

struct list\_elem \*e;

struct thread \*tmp;

e = list\_begine(&sleep\_list);

tmp = list\_entry(e, struct thread, elem);

if (tmp->wakeup\_time <= ticks)

thread\_wake (ticks);

else return;

}

**void check\_thread ()**

새로운 thread가 생겨 ready list에 들어가거나로 인해 ready list의 front thread의 priority와 CPU할당된 thread 의 priority 를 비교하여 만약 CPU 할당된 thread의 priority가 더 낮다면 thread를 새로 할당해주는 함수이다.

void check\_tick (int64\_t ticks) {

struct list\_elem \*e;

struct thread \*tmp;

e = list\_begine(&sleep\_list);

if (e != NULL) {

tmp = list\_entry(e, struct thread, elem);

if (tmp->priority > thread\_current()->priority)

thread\_yield();

}

**void thread\_aging ()**

starvation을 방지하기 위한 함수이다. ready list내부를 탐색하며 ready 중인 thread들의 priority를 증가시켜준다.

void thread\_aging () {

struct list\_elem \*list\_it;

for (list\_it = list\_begin(&ready\_list); list\_it != list\_end(&ready\_list);

list\_it = list\_next(list\_it)) {

list\_entry(list\_it, struct thread, elem)->priority++;

}

}

**/threads/thread.h**

함수들의 원형을 선언해주었다.

**/threads/sync.c**

**void sema\_down (struct semaphore \*sema)**

sema down 시에 sema의 waiters에 priority 에 맞춰 들어갈 수 있도록 sema의 waiters 에 list\_push\_back하던 걸 list\_insert\_ordered 하였다. 이 때, compare\_priority 라는 priority에 대해 내림차순으로 정렬되도록 compare용 함수를 인자로 넣어주었다.

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\_insert\_ordered (&sema->waiters, &thread\_current()->elem,

compare\_prioriry, 0);

thread\_block();

}

sema->value--;

intr\_set\_level (old\_level);

}

**void sema\_up (struct semaphore \*sema)**

sema\_up을 통해 unblock된 thread의 priority 가 가장 높을 수도 있으므로 sema\_up이 다 끝나고 나서 thread\_yield를 실행시켜주었다.

void sema\_up (struct semaphore \*sema) {

enum intr\_level old\_level;

ASERT (sema != NULL);

old\_level = intr\_disable();

if (!list\_empty(&sema->waiters)

thread\_unblock (list\_entry(list\_pop\_front(&sema->waiters),

struct thread, elem));

sema->value++;

intr\_set\_level (old\_level);

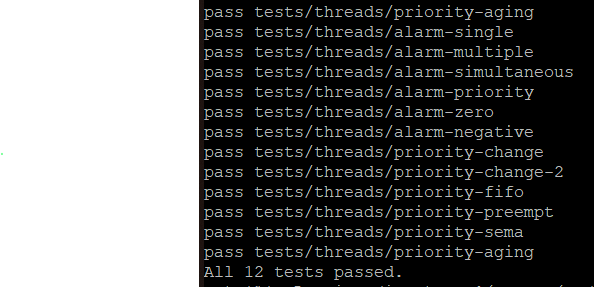
thread\_yield();

}

1. **시험 및 평가 내용**

* test 결과

**13/13**

****

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

1. **기타**
2. **연구 조원 기여도**

이기현 : 50%

장주호 : 50%

1. **소감**

이기현 : 마지막 pintos 프로젝트 threads를 진행했다. 그래도 3차여서 그런지 이해하는데 조금 수월했다. 조교님께서 겁주셔서 무서웠는데 이해를 하고 코드를 짜니 좀 더 편하게 마무리 할 수 있었다.

장주호 : 힘들었던 한 학기였다.   
pintos프로젝트를 처음 시작할 때만 해도 막막함이 매우 컸지만 결국 모든 것을 다 완료했다. 돌이켜보면 학교생활에서 손 꼽을 수 있을 만큼 기억에 남는 시간들이었다.  
여러 시행착오들을 겪으며 밤새어 고민도 해보고 계속해서 실패했던 경험들을 토대로 앞으로 어떤 일을 하더라도 겁 없이 부딪힐 수 있다고 생각한다.