**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

학번 / 이름 : 20201635 / 전찬

개발 기간 : 2022.11.01 – 2022.11.12

1. **개발 목표**

이번 프로젝트 3은 Thread에 관련된 여러 기능을 구현해야 한다. 첫 번째로, alarm clock을 busy waiting이 아닌 효율적인 방식으로 구현해야 하며, 두 번째로 pintos의 기본적인 round-robin scheduler에서 벗어나 priority을 적용한 scheduler을 구현해야 한다. 이를 위해 priority scheduling, priority aging 등을 구현해야 한다. 더 나아가 MLFQ 형태의 scheduler 또한 구현해야 한다.

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

현재 pintos의 alarm clock은 busy waiting 방식으로, 비효율적이다. 따라서 계속해서 해당 thread을 running하며 확인하는 것이 아닌, timer tick마다 interrupt을 바탕으로 kernel에서 확인해주는 방식으로 alarm clock을 구현하면 sleep하는 동안 running 상태가 되지 않기 때문에 효율적으로 alarm clock을 수행할 수 있다.

* 1. Priority Scheduling

초기의 pintos는 모든 ready 상태의 thread에 대해 round-robin 방식으로 running thread을 결정한다. 이러한 round-robin 방식은 간단하게 scheduling을 수행할 수 있지만, 원하는 thread을 바로 실행시킬 수 없으며, time quantum에 따라 time quantum이 커지면 FCFS 형태로 작동하며, 작아지면 context switch overhead가 증가한다는 비효율성이 존재하기도 한다. 따라서 각 thread마다 priority을 매겨서 priority가 높은 thread을 먼저 실행시키며, priority aging 기법을 적용하면 효율적인 scheduling을 구현해낼 수 있다.

* 1. Advanced Scheduler

위와 같이 priority scheduling을 적용한 scheduler은 round-robin 방식에 비해 효율적이다. 이때 priority aging의 방식을 적용하여 starvation 문제를 해결할 수 있으며, advanced scheduler은 nice, recent\_cpu, load\_avg 값을 토대로 aging을 구현한다. 이를 통해 실제 priority aging이 적용되며, 일정 ticks 마다 priority을 재설정하여 starvation을 방지하며, 효율적인 scheduler을 구현해낼 수 있다.

* 1. **개발 내용**

1. Blocked 상태의 thread을 깨우는 방법

blocked list에 존재하는 thread에 대해, 매 tick에서 timer interrupt가 발생할 때마다 kernel에서 blocked list의 모든 thread을 확인하며, 해당 thread의 wake up time 이상이 경과했으면, 해당 thread을 thread\_unblock 함수를 통해 ready list에 다시 삽입해주는 형식으로 구현할 수 있다. 이를 위해 해당 thread의 wake up time을 저장할 수 있는 thread 구조체 변수를 추가해 주어야 하며, timer\_sleep 함수가 실행될 때, 해당 변수에 wake up time을 계산해서 저장해 주어야 한다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling 방법

ready list에 현재 running thread보다 높은 priority을 가진 thread가 새롭게 추가되는 경우, thread\_yield 함수를 통해 가장 priority가 높은 thread에게 thread을 yield 해 주어야 한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소

aging 단계에서 각 thread의 priority을 계산할 때 3가지 요소가 필요하다. 각각은 아래와 같다.

1. nice : 양수이면 priority가 감소하며, 음수이면 priority가 증가하도록 하는 값이다.

2. load\_avg : ready state에 있는 thread의 평균적인 수를 의미하는 값이다.

3. recent\_cpu : 각 thread의 CPU time을 추정하는 값이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11.01 – 11.02 : alarm clock 구현

11.03 – 11.04 : priority scheduling 구현

11.05 – 11.07 : priority aging 구현

11.08 – 11.08 : BSD scheduler 구현 및 make check 바탕으로 구현 수정

11.09 – 11.12 : 구현 레포트 작성 및 make check 바탕으로 구현 수정

* 1. **개발 방법**
* Blocked 상태의 thread을 깨우는 방법

blocked 상태의 thread을 timer\_interrupt 함수로 깨우기 위해, 우선 blocked 상태의 thread을 저장할 list인 blocked\_list가 필요하다. 이는 threads/thread.c 에 전역 변수로 존재하는 형태로, ready\_list, 혹은 all\_list와 유사하게 thread.c의 모든 함수에서 접근할 수 있도록 한다.

위의 blocked\_list에 timer\_sleep 함수를 call 하며 blocked 된 thread들에 대해, 각 thread가 일어날 시간을 저장해야 한다. 이는 threads/thread.h 의 thread 구조체 정의에서 wake\_time을 추가해 주는 형식으로 구현해야 한다. 또한 thread\_sleep 함수를 call 할 때, 해당 thread에 wake\_time인 현재 시간 + sleep 하는 시간 을 저장해 주어야 한다.

위를 통해 구현된 blocked\_list의 각 thread에 대해, 마지막으로 각 tick마다 모든 list의 원소를 확인하며, 현재 시간(tick) >= wake\_time 인 경우에 해당 thread을 thread\_unblock을 통해 다시 ready\_list에 추가해 주어야 한다.

위와 같은 과정을 통해 Blocked 상태의 thread을 깨울 수 있다.

* Ready list에 running thread보다 높은 priority을 가진 thread가 들어올 경우 priority scheduling 방법

ready list에 running thread보다 높은 priority을 가진 thread가 새롭게 들어오는 경우, 새로운 thread는 thread\_create 함수를 통해 만들어진다. 이렇게 thread\_create을 통해 만들어진 thread는 input priority에 의해 priority을 결정하는데, 해당하는 priority가 현재 실행중인 thread\_current() -> priority 보다 큰 경우에 thread\_yield 함수를 실행시켜 다시 scheduling할 수 있다. 이 경우에 새롭게 만들어지는 thread가 가장 큰 priority이기 때문에, 해당 thread가 실행되며, 기존에 실행되던 thread는 ready\_list의 마지막에 추가될 것이다. 이러한 방식으로 priority scheduling을 수행할 수 있다.

* Advanced Scheduler에서 priority 계산에 필요한 각 요소

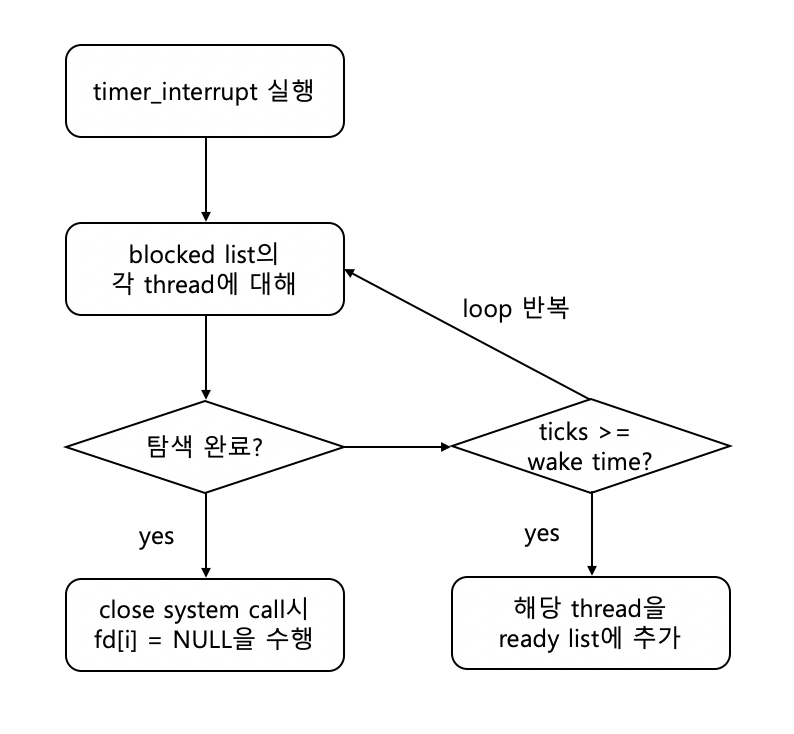
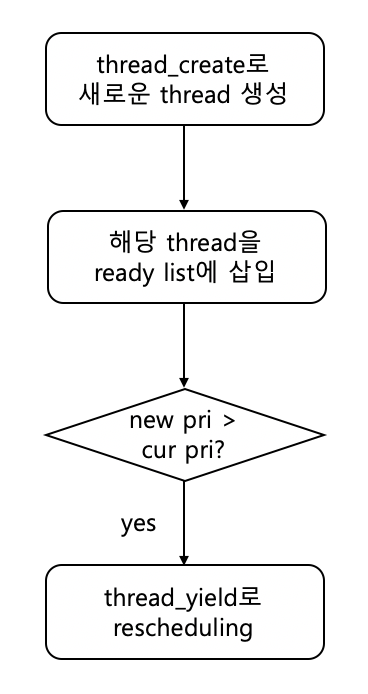
1. nice : 각각의 thread 구조체에 새로운 변수로 추가해준다.

2. load\_avg : threads/thread.c의 전역 변수를 사용해서 구현할 수 있다.

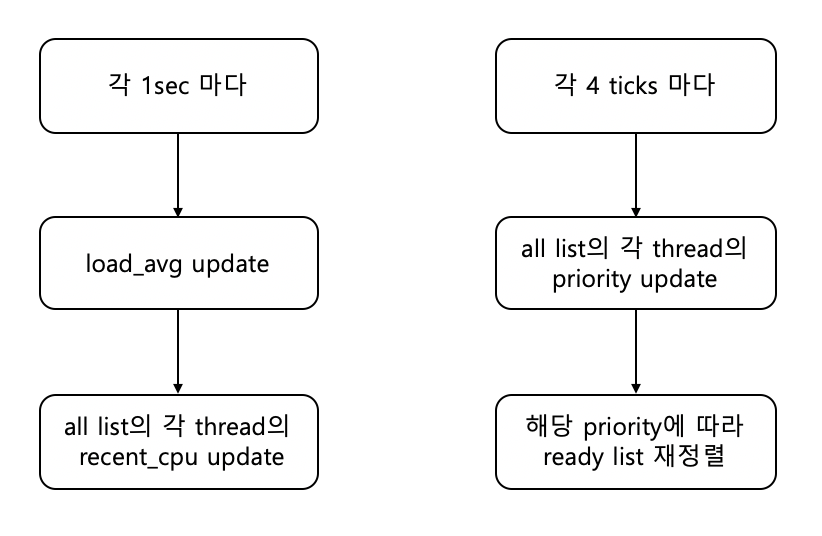
3. recent\_cpu : 각 thread 구조체에 새로운 변수로 추가해 구현할 수 있다.

추가로 fixed pointer와 관련해서 연산을 수행할 때 int / fixed pointer 사이의 연산을 수행할 수 있도록 #define FRACTION 을 추가해 주었다.

1. **연구 결과**
   1. **Flow Chart**

** **

<개발 내용 1의 flowchart> <개발 내용 2의 flowchart>

****

<개발 내용 2의 flowchart>

* 1. **제작 내용**

첫 번째로 alarm clock을 구현하기 위해 threads/thread.h의 thread 구조체에 wake\_time, 그리고 threads/thread.c에 전역 변수로 blocked\_list 추가를 수행하며, threads/thread.c의 thread\_init에서 blocked\_list에 대한 list\_init을 추가해 주었다. 각각은 아래와 같다.

int64\_t wake\_time;

<threads/thread.h의 구조체 추가 부분>

static struct list blocked\_list;

<threads/thread.c의 전역 변수 추가>

list\_init(&blocked\_list);

<threads/thread.c의 thread\_init 함수 추가 부분>

이후에는 alarm clock을 적절한 형태로 수행할 수 있도록 구현해 주었다. blocked\_list가 threads/thread.c 상의 전역 변수이기 때문에 blocked\_list에서의 삽입, 삭제 등은 thread.c에서 새로운 함수를 추가해주는 형식으로 구현해 주었다. 따라서 devices/timer.c의 timer\_sleep, timer\_interrupt 함수는 코드를 유지하되, thread\_sleep, thread\_awake 함수를 실행하는 형태로 코드를 추가해 주었다. 각각은 아래와 같다.

void

timer\_sleep (int64\_t ticks)

{

int64\_t start = timer\_ticks();

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

thread\_sleep(start + ticks);

}

<코드를 추가한 devices/timer.c의 timer\_sleep 함수>

static void

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

{

ticks++;

thread\_tick ();

thread\_awake(ticks);

}

<코드를 추가한 devices/timer.c의 timer\_interrupt 함수>

또한 실제적인 구현 코드인 thread\_sleep, thread\_awake 함수는 threads/thread.c에 새롭게 구현해 주었다. 각각은 아래와 같다.

void thread\_sleep(int64\_t ticks)

{

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

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

ASSERT(cur != idle\_thread);

cur->wake\_time = ticks;

list\_push\_back(&blocked\_list, &cur->elem);

thread\_block();

intr\_set\_level(old\_level);

}

<구현한 threads/thread.c의 thread\_sleep 함수>

위 함수는 현재 thread에 대해, input ticks까지 blocked을 수행할 수 있도록 구조체의 wake\_time 변수에 값을 저장하며, 해당 thread을 blocked\_list에 추가하는 형태이다.

void thread\_awake(int64\_t ticks)

{

struct list\_elem \*e = list\_begin(&blocked\_list);

while (e != list\_end(&blocked\_list)){

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

if(check->wake\_time <= ticks){

e = list\_remove(e);

thread\_unblock(check);

}

else{

e = list\_next(e);

}

}

}

<구현한 threads/thread.c의 thread\_awake 함수>

위 함수는 timer\_interrupt가 실행되는 매 ticks마다 call되며, blocked\_list에 존재하는 모든 thread에 대해, 해당 thread의 wake\_time이 현재 ticks보다 작거나 같으면 다시 ready\_list로의 추가를 수행하는 형태이다. 위와 같은 구현을 통해 alarm clock을 구현할 수 있다.

이후에는 기존의 round-robin 방식에서 벗어나, 0~63의 priority에 따라 우선순위를 정해 실행할 수 있는 pintos scheduler을 구현해야 한다. 각 thread의 priority을 저장할 변수는 이미 threads/thread.h의 thread 구조체 정의에 포함되어 있으므로 수정할 필요가 없다. 따라서 따라서 첫 번째로 priority scheduler을 구현하기 위해 각 priority에 해당하는 thread들을 저장할 64개의 list 가 필요하며 이는 기존의 ready\_list을 ready\_list\_array[64] 로 변경하는 형식으로 구현했다. 이는 아래와 같다.

static struct list ready\_list\_array[64];

<threads/thread.c 의 ready\_list\_array[64]의 구현>

init해주는 부분 또한 threads/thread.c의 thread\_init 함수에 추가해 주었다. 이는 아래와 같다.

for(int i=0; i<64; i++){

list\_init(&ready\_list\_array[i]);

}

<threads/thread.c의 thread\_init 함수 추가 부부>

또한 ready\_list에서 ready\_list\_array로 변경되었기 때문에, 기존에 ready\_list에 연관되어 있는 함수인 threads/thread.c의 thread\_unblock, thread\_yield, next\_thread\_to\_run을 형식에 맞게 수정해 주어야 한다. thread\_unblock, thread\_yield 함수는 기존의 ready\_list로의 추가가 아닌, ready\_list\_array[priority] 로의 추가를 수행할 수 있도록 수정해주며, next\_thread\_to\_run은 empty list가 아닌 가장 priority가 높은 ready\_list\_array[i] 을 찾아 해당 list의 첫 번째 thread을 return 해주는 형식으로 수정해주어야 한다. 각각은 아래와 같다.

list\_push\_back (&ready\_list\_array[t->priority], &t->elem);

<thread\_unblock, thread\_yield에서 변경된 부분>

for(int i=63; i>=0; i--){

if (!list\_empty(&ready\_list\_array[i]))

{

return list\_entry(list\_pop\_front(&ready\_list\_array[i]), struct thread, elem);

}

}

return idle\_thread;

<threads/thread.c의 next\_thread\_to\_run 함수 변경 부분>

이를 통해 적절한 형태로 scheduling을 수행할 수 있다.

또한 추가적으로 사용할 수 있는 기능을 위한 threads/thread.c 상의 thread\_get\_priority, thread\_set\_priority 함수 또한 구현해 주어야 한다. thread\_set\_priority 함수는 priority가 변경되기 때문에, 해당 thread을 기존의 priority list에서 삭제 시키며, 새로운 priority list에 삽입하는 작동을 포함시켜야 한다. thread\_get\_priority 함수는 현재 thread의 priority을 return 해주면 된다. 각각은 아래와 같다.

void

thread\_set\_priority (int new\_priority)

{

// mlfqs-load-60 통과 ,, manual 더 읽어봐야 할듯

if(thread\_mlfqs){

return;

}

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

int save\_priority = cur->priority;

cur->priority = new\_priority;

// 이전에 있던 priority가 새로운 priority보다 큰 경우, 다시 scheduling 수행

if(save\_priority > new\_priority){

thread\_yield();

}

}

<threads/thread.c thread\_set\_priority 구현 코드>

int

thread\_get\_priority (void)

{

return thread\_current ()->priority;

}

<threads/thread.c thread\_get\_priority 구현 코드>

이후에는 semaphore에서도 priority을 고려할 수 있도록 구현을 수행해 주어야 한다. 기존의 semaphore에서 threads/synch.c의 sema\_up 함수는 기존의 sema을 waiting 하고 있는 하나의 thread을 깨워주는 함수인데, priority의 고려 없이 sema\_waiter list에서 첫 번째 thread을 unblock 해주는 형식으로 구현되어 있다. 하지만 priority가 존재하는 scheduler에서는 각 waiting thread들 사이에서도 priority가 존재하기 때문에, sema\_waiter list을 전부 탐색하며, priority가 max인 thread을 찾아 해당 thread을 unblock하며, return해 주어야 한다. 이를 구현한 코드는 아래와 같다.

void

sema\_up (struct semaphore \*sema)

{

enum intr\_level old\_level;

int max\_priority;

struct list\_elem\* max\_e;

struct list\_elem\* e;

struct thread\* max\_t;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

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

e = list\_begin(&(sema->waiters));

max\_e = e;

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

max\_priority = max\_t->priority;

e = list\_next(e);

while(e != list\_end(&(sema->waiters))){

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

// 현재 저장되어 있는 priority 보다 클 경우에,

if(check->priority > max\_priority){

max\_e = e;

max\_t = check;

max\_priority = check->priority;

}

e = list\_next(e);

}

list\_remove(max\_e);

thread\_unblock (max\_t);

}

sema->value++;

intr\_set\_level (old\_level);

thread\_yield();

}

<thtreads/synch.c 의 sema\_up 함수 구현 코드>

이를 통해 semaphore 또한 priority을 고려할 수 있도록 구현하며, priority-sema test을 통과할 수 있었다.

이후에는 priority aging 부분을 구현해 주었다. 구현해야 할 priority aging 함수는 매 tick마다 threads/thread.c의 thread\_tick 함수가 실행될 때 전역 변수인 thread\_prior\_aging == ture인 경우에 수행되는 함수이다. 따라서 해당 함수에서 현재까지의 tick을 파악하고, 4 ticks, 혹은 1 second 마다 priority, 혹은 load\_avg, recent\_cpu 값을 수식에 따라 update 하는 형식으로 구현할 수 있다. 이때 선행되어야 하는 것은 fixed point에 대한 이해이다. 우선 간단하게 fixed point는 int값을 14bit만큼 shift한 형태로 이해할 수 있다. 또한 fixed point와 integer 사이에서 연산을 수행하는 것이 필요한데, 이번 프로젝트에서 필요한 fixed point 관련 연산은 아래와 같다.

fixed point1 + fixed point2 : 덧셈을 수행한다.

fixedpoint1 – fixed point2 : 뺄셈을 수행한다.

fixed point1 \* fixed point2 : 64bit 변수에서 곱을 시행하고, FRACTION으로 나누어 주어야 한다.

fixed point1 / fixed point2 : 64bit 변수에 fixed point1 \* FRACTION을 나눗셈을 수행한다.

fixed point + integer : integer \* FRACTION을 통해 integer을 fixed point로 변환한 후 연산한다.

fixed point \* integer : 곱셈을 수행한다.

fixed point / integer : 나눗셈을 수행한다.

이러한 연산들을 토대로 일정 ticks마다 load\_avg, recent\_cpu, priority을 update할 때, 위 연산들을 사용해서 계산을 수행할 수 있다. priority aging에서 사용할 전역 변수, macro는 아래와 같다..

#ifndef USERPROG

bool thread\_prior\_aging;

#endif

<threads/thread.c 전역 변수 thread\_prior\_aging 추가>

static int load\_avg;

< threads/thread.c 전역 변수 load\_avg 추가>

#define FRACTION (1<<14)

< threads/thread.c fixed point 관련 macro 추가>

또한 thread\_init 첫 thread에서 사용할 load\_avg, recent\_cpu, nice 값을 정해 주었으며, init\_thread에서 각 thread을 생성할 때 해당 thread을 생성하는 thread의 recent\_cpu, nice 값을 상속받을 수 있도록 구현해 주었다.

load\_avg = 0;

initial\_thread->recent\_cpu = 0;

initial\_thread->nice = 0;

init\_thread (initial\_thread, "main", PRI\_DEFAULT);

initial\_thread->recent\_cpu = 0;

initial\_thread->nice = 0;

< threads/thread.c thread\_init 함수 추가 부분>

t->recent\_cpu = running\_thread()->recent\_cpu;

t->nice = running\_thread()->nice;

< threads/thread.c init\_thread 함수 추가 부분>

또한 프로젝트 설명 ppt에 나와있는 것처럼, 각 timer\_tick 마다 실행되는 함수 thread\_tick 에서 조건문을 추가해 주었다. 해당 조건문의 경우 구현을 완료한 이후의 조건문이기 때문에, thread\_mlfqs에 관련된 조건문도 추가되어 있다. 이는 아래와 같다.

#ifndef USERPROG

if (thread\_prior\_aging == true || thread\_mlfqs == true)

thread\_aging ();

#endif

< threads/thread.c thread\_tick 함수 추가 부분>

위에서 매 tick마다 실행할 thread\_aging 함수는 다음과 같다.

void thread\_aging(void)

{

// 현재 실행중인 thread에 대해서는, tick마다 recent\_cpu 값을 1씩 추가해준다.

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

thread\_current()->recent\_cpu = thread\_current()->recent\_cpu + 1 \* FRACTION;

}

// 1초마다 recent\_cpu, load\_avg 값을 update 해준다.

if (timer\_ticks() % TIMER\_FREQ == 0){

update\_recent\_cpu\_load\_avg();

}

// 4 ticks마다 priority을 변경해주며, ready\_list\_array을 다시 정렬한다.

if (timer\_ticks() % TIME\_SLICE == 0){

update\_priority();

}

}

<threads/thread.c thread\_aging 함수 추가 부분>

해당 함수를 보면, 우선 현재 thread에 대해, idle\_thread가 아니라면 recent\_cpu 값을 1만큼 추가해주는 형태이다. 이후에는 현재까지의 timer\_ticks() 값에 따라 recent\_cpu와 load\_avg, 혹은 priority을 update 한다. 추가로 아래는 최대 priority을 파악하기 위한 함수이다.

int pri\_max(void)

{

for(int i=63; i>=0; i--){

if(!list\_empty(&ready\_list\_array[i])){

return i;

}

}

return -1;

}

<threads/thread.c pri\_max 함수 구현>

또한 각각의 update 함수는 다음과 같다.

void update\_recent\_cpu\_load\_avg(void)

{

int ready\_threads = 0;

struct list\_elem\* e;

struct thread\* check;

// ready\_threads 의 개수 파악

for(int i=0; i<64; i++){

for(e = list\_begin(&ready\_list\_array[i]); e != list\_end(&ready\_list\_array[i]); e = list\_next(e)){

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

if(check != idle\_thread){

ready\_threads += 1;

}

}

}

// 현재 thread가 idle\_thread인 경우, thread count을 하지 않는다.

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

ready\_threads += 1;

}

// load\_avg update

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

// 순서에 따라서 결과가 달라질 수 있나?

load\_avg = calculate\_load\_avg(ready\_threads);

// all list에 있는 모든 thread에 대해 idle\_thread을 제외하고 update 을 수행

for(e = list\_begin(&all\_list); e != list\_end(&all\_list); e = list\_next(e)){

check = list\_entry(e, struct thread, allelem);

if(check != idle\_thread){

check->recent\_cpu = calculate\_recent\_cpu(check);

}

}

}

<threads/thread.c update\_recent\_cput\_load\_avg 함수 구현 코드>

우선 모든 ready\_list\_array의 각 thread와 현재 thread을 탐색하며 ready\_threads 개수를 파악하며, 이를 토대로 load\_avg을 update 한다. 또한 all\_list에 존재하는 모든 thread에 대해 각 thread의 recent\_cpu 값을 update 해주는 형식으로 코드가 진행된다.

priority update는 다음과 같다.

void update\_priority(void)

{

struct list\_elem\* e;

struct thread\* check;

int save\_priority;

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

// ready\_list\_array의 각 list들을 탐색하며 search 을 수행한다.

//idle\_thread는 ready list 상에 존재할 수 없으며, running thread인지 확인하여 예외처리를 수행한다.

// fixed\_point -> integer은 FRACTION으로 나누어주면 된다.

for(int i = 63; i >= 0; i--){

e = list\_begin(&ready\_list\_array[i]);

while(e != list\_end(&ready\_list\_array[i])){

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

save\_priority = check->priority;

check->priority = calculate\_priority(check);

if (check->priority > PRI\_MAX){

check->priority = PRI\_MAX;

}

if (check->priority < PRI\_MIN){

check->priority = PRI\_MIN;

}

if(check->priority == save\_priority){

e = list\_next(e);

}

else{

e = list\_remove(e);

list\_push\_back(&ready\_list\_array[check->priority], &check->elem);

}

}

}

// 현재 running thread에 대해서도 priority update 을 수행한다.

// fixed\_point -> integer은 FRACTION으로 나누어주면 된다.

if(cur != idle\_thread){

cur->priority = calculate\_priority(cur);

if (cur->priority > PRI\_MAX){

cur->priority = PRI\_MAX;

}

if (cur->priority < PRI\_MIN){

cur->priority = PRI\_MIN;

}

}

// 해당 switch을 추가하려면, 위에서 context switch 위에서 thread\_aging을 먼저 적용해야 한다.

// timer\_interrupt 을 통해 실행되는 함수이므로, thread\_yield가 아닌 intr~ 로 수행해야 한다.

//if (thread\_current()->priority < pri\_max()) {

// intr\_yield\_on\_return();

//}

}

<threads/thread.c의 update\_priority 함수 구현>

이를 통해 ready\_list\_array의 각 thread에 대해, priority update을 수식에 맞추어 계산할 수 있다.

또한 위 update 함수에서 fixed point와 integer 사이의 계산을 통해 update 값을 구하는 함수는 아래와 같다.

int calculate\_load\_avg(int ready\_threads){

return 59 \* load\_avg / 60 + ready\_threads \* FRACTION / 60;

}

int calculate\_recent\_cpu(struct thread\* cur){

int64\_t save = (2 \* load\_avg) \* FRACTION;

save = save / (2 \* load\_avg + 1 \* FRACTION);

save = save \* cur->recent\_cpu / FRACTION;

save = save + cur->nice \* FRACTION;

return (int)save;

}

int calculate\_priority(struct thread\* cur){

return (PRI\_MAX \* FRACTION - cur->recent\_cpu / 4 - cur->nice \* FRACTION \* 2) / FRACTION;

}

<threads/thread.c의 calculate 함수 구현>

마지막으로 threads/thread.c 상에서 구현한 set\_nice, get\_nice, get\_load\_avg, get\_recent\_cpu 함수는 다음과 같다.

/\* Sets the current thread's nice value to NICE. \*/

void

thread\_set\_nice (int nice UNUSED)

{

// nice 을 바꾸며, 바뀐 nice에 따라 priority 또한 재설정한다

// nice 에 의해 recent\_cpu 도 변경할 수 있지 않나?

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

cur->nice = nice;

cur->priority = calculate\_priority(cur);

cur->recent\_cpu = calculate\_recent\_cpu(cur);

if (cur->priority > PRI\_MAX){

cur->priority = PRI\_MAX;

}

if (cur->priority < PRI\_MIN){

cur->priority = PRI\_MIN;

}

if(cur->priority < pri\_max()){

thread\_yield();

}

}

/\* Returns the current thread's nice value. \*/

int

thread\_get\_nice (void)

{

return thread\_current()->nice;

}

/\* Returns 100 times the system load average. \*/

// rounded을 수행해야 하나?

int

thread\_get\_load\_avg (void)

{

return 100 \* load\_avg / FRACTION;

}

/\* Returns 100 times the current thread's recent\_cpu value. \*/

int

thread\_get\_recent\_cpu (void)

{

return 100 \* thread\_current()->recent\_cpu / FRACTION;

}

<추가로 구현한 함수들>

여기에서 thread\_set\_nice을 수행할 때, 해당 nice 값에 따라 priority와 recent\_cpu값을 추가로 변경해 주어야 하며, 변경한 priority가 ready\_list\_array에 존재하는 priority max 값보다 작은 경우 다시 scheduling 하는 형태임을 파악할 수 있다.

위와 같은 구현들을 사용해서 priority aging과 함께 BSD scheduler을 구현해낼 수 있었다.

추가로 test 관련해서 어려움이 존재했는데, 첫 번째로 priority update을 수행하며, mlfqs-load-60 test을 통과하지 못했다. 이를 찾아보니 thread\_set\_priority에서 thread\_mlfqs인 경우에는, priority set을 수행하지 못하도록 막아주면 해결되는 문제였다. 따라서 아래 코드를 추가해 주었다.

if(thread\_mlfqs){

return;

}

<threads/thread.c의 thread\_set\_priority 추가 부분>

이를 통해 해당 test을 통과할 수 있었다.

마지막으로 mlfqs-load-avg test을 통과하지 못했는데, 원인으로 각 load\_avg 의 update가 synchronize되어 있지 않아서 그런 것이라고 생각을 했다. 따라서 처음에는 각 load\_avg update사이에서 struct lock을 적용하여 구현을 해 보았다. 하지만 semaphore은 sema\_up, sema\_down 등을 수행할 때, 수행이 이루어지는 코드가 interrupt 상의 코드라면 에러를 발생하게 된다. 따라서 두 번째 방법으로 devices/timer.c 상의 timer\_interrupt 함수를 수행할 때 해당 함수를 다음과 같이 변경해 보았다.

static void

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

{

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

ticks++;

thread\_tick ();

thread\_awake(ticks);

intr\_set\_level(old\_level);

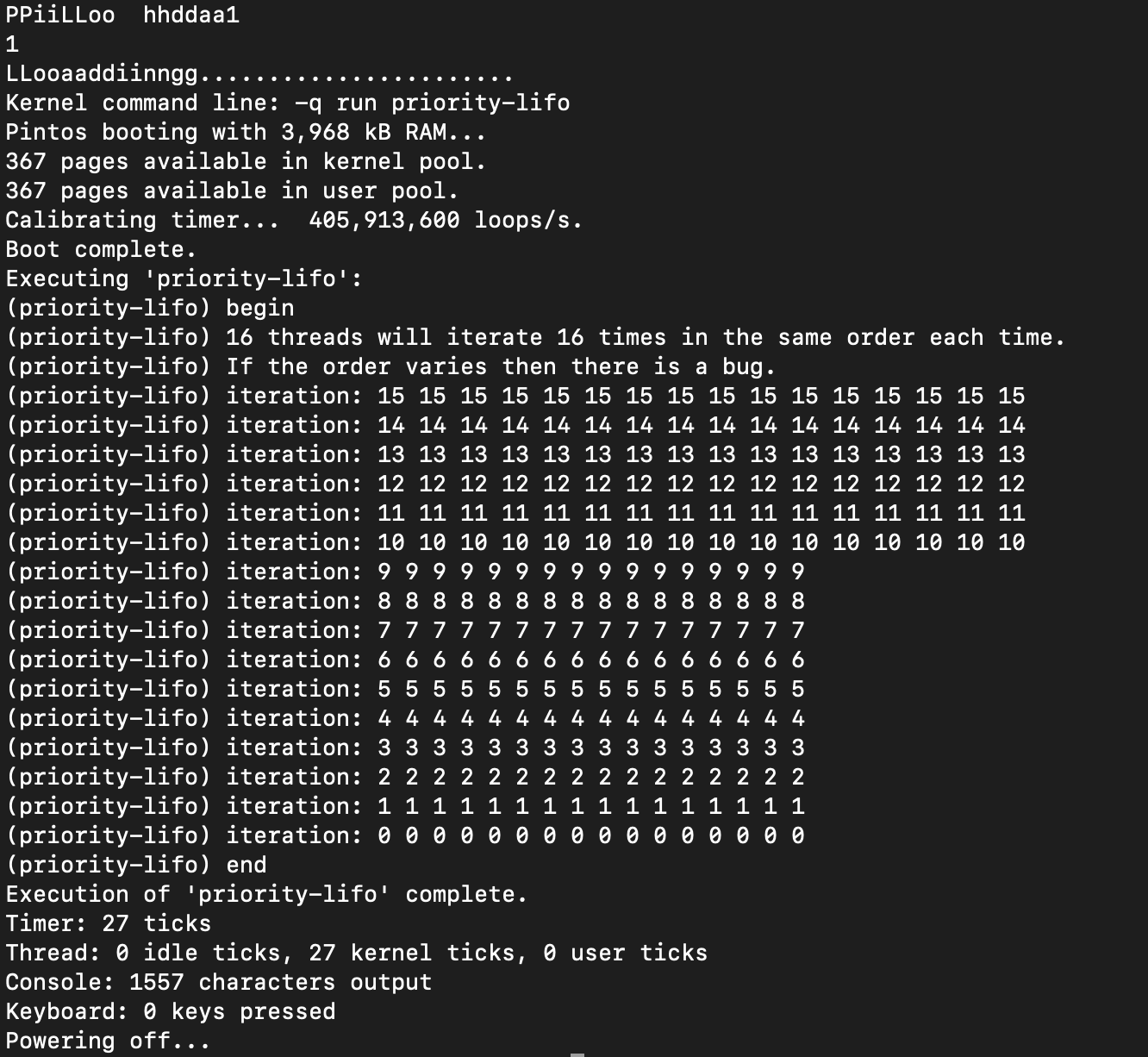
}

<devices/timer.c timer\_interrupt 함수 수정>

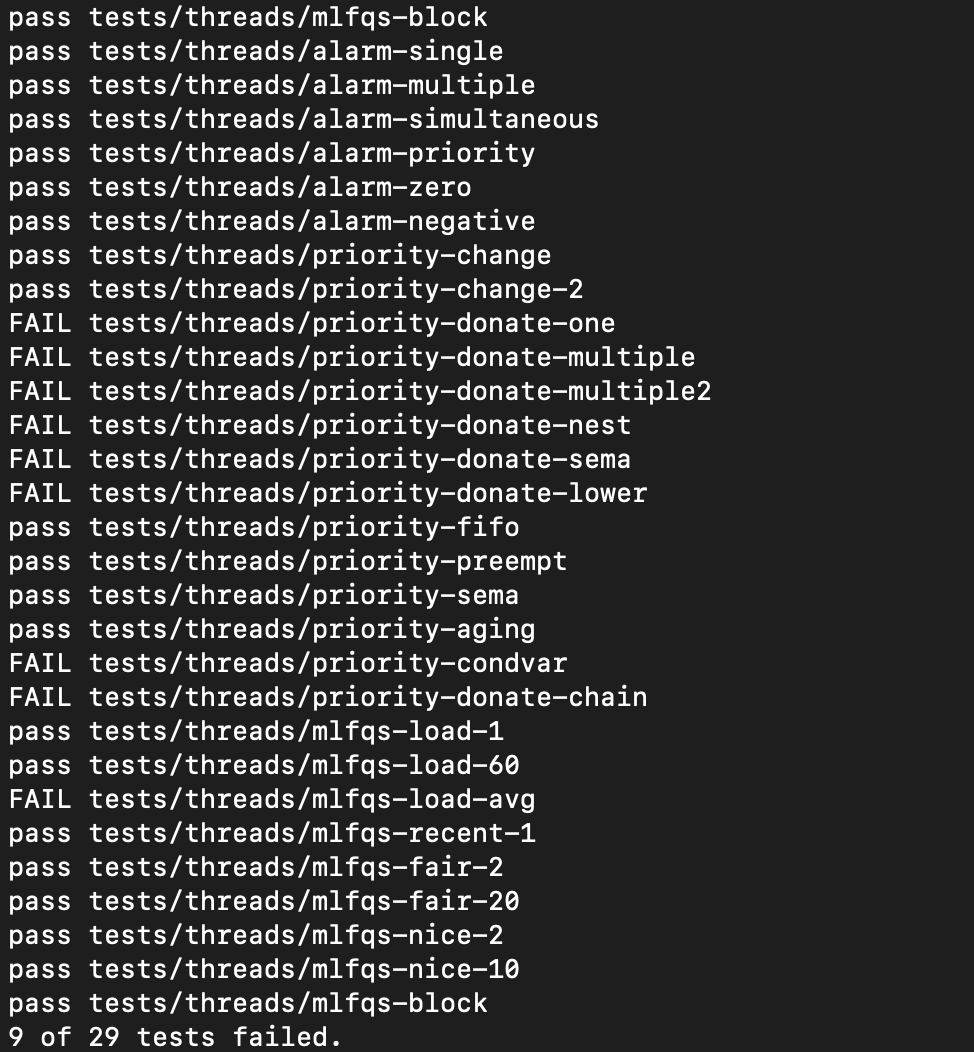
이를 통해 intr\_disable을 수행하면, 각 load\_avg 값을 재설정할 때 synchonize될 것이라 생각했지만, 이러한 구현을 통해서도 해당 test는 통과할 수 없었다. 값에 큰 차이가 있는 것이 아니기 때문에, 구현에 큰 문제는 없을 것이라 생각하여 더이상 코드를 수정하지는 않았다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority-lifo.c 코드는 16개의 thread가 각각 서로 다른(1씩 차이나는) priority을 가지면서 실행되는 형태이다. 이때, 첫 번째로 실행되는 thread priority = PRI\_DEFAULT + 1, 두 번째 = PRI\_DEFAULT + 1 + 1, 세 번째 = PRI\_DEFAULT + 1 + 2 와 같이 마지막으로 실행되는 thread가 가장 우선순위가 높은 형태이다. 따라서 현재 실행되는 thread의 id을 확인하면 15 부터 내림차순으로 수행되어야 함을 파악할 수 있다. 이를 실행한 결과는 아래와 같다.



* make check 수행 결과



make check 에서는 mlfqs-load-avg test을 제외하고 모두 통과할 수 있었다.