**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171646 / 박태윤

개발 기간 : 11/1 ~ 11/6

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

기존에 ticks까지 yield를 하는 비효율적인 timer\_sleep을 개선한, ticks마다 매번 호출이 되는timer\_interrupt를 이용해 마치 실행중인 쓰레드를 잠시 재웠다가 일정 시간에 도달했을 경우 깨우는 alarm clock을 구현한다. 또한 각 쓰레드마다 우선 순위를 부여해 ready상태에 있는 쓰레드들 중 가장 높은 우선순위를 가지는 쓰레드를 먼저 실행할 수 있도록 하는 메커니즘인 priority scheduling을 구현한다. 추가적으로, 여러 큐들이 존재하고 각 큐마다 우선 순위를 가지고 있고 우선 순위가 낮은 쓰레드들이 실행이 안되는 경우 등을 해결할 수 있는 Advanced Scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
  2. Priority Scheduling
  3. Advanced Scheduler (추가구현을 한 경우)

1)

기존에 ticks까지 계속 thread\_yield를 호출하여 sleep과 awake를 지속적으로 반복하여 sleep을 비효율적으로 하는 busy-waiting방식을 벗어나 sleep\_list자료구조에 따로 sleep된 쓰레드들을 추가하여 timer\_interrupt에서 일어나야 하는 쓰레드가 있는지 확인하고 이를 깨워주는 방식으로 구현을 해 비효율적으로 많은 CPU시간을 소비하는 기존의 alarm clock기능을 개선할 수 있다.

2)

기존에는 실행 가능한 쓰레드들을 가지고 있는 ready\_list에 추가를 하는 경우 back에, running쓰레드를 pop하는 경우는 맨 앞에서 뽑는 단순한 방식으로 구현이 되어 있는데, Priority Scheduling을 구현하면 각 쓰레드 마다 실행의 우선 순위를 부여하여 기존에 단순하게 들어온 순서대로 실행을 하는 것이 아닌 우선 순위 기준으로 실행을 할 수 있다.

3)

Priority가 지속적으로 매 시간 마다 change가 되고, 각 우선순위에 따라 여러 큐가 존재하는 multi-level-feedback-queue를 만든 advanced scheduler를 구현함으로써 우선 순위가 낮은 쓰레드는 CPU를 점유하지 못한 채 계속 실행이 되지 않는 등의 문제를 해결할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

1)

재우고자 하는 쓰레드를 devices디렉토리 timer.c파일에 있는 timer\_sleep함수에서 인자로 받은 ticks만큼 재운다(thread의 상태가 blocked가 됨). blocked가 된 쓰레드는 sleep\_list라는 자료구조에 추가가 되며, 1ms마다 실행되는 인터럽트인 timer\_interrupt에서 깨울 쓰레드가 있는지 구현한 thread\_awake()함수에서 확인해준다. 각 쓰레드마다 timer\_sleep함수에서 재울 때 wakeup이라는 변수에 언제까지 재우고자 하는 시간이 저장이 되며 thread\_awake함수에서는 sleep\_list를 탐색하면서 현재 timer\_ticks()가 쓰레드의 wakeup변수보다 크거나 같은 경우 sleep\_list에서 이 쓰레드를 제거하고 thread\_unblock함수를 실행시키는 방식으로 blocked된 쓰레드를 깨운다.

2)

ready\_list에 현재 running쓰레드보다 priority가 높은 쓰레드가 추가되는 경우 쓰레드의 priority순서에 맞게 ready\_list에 추가를 해줘야 한다. ready\_list에 쓰레드가 추가되는 경우는 쓰레드가 생성이 되거나, blocked상태의 쓰레드가 unblock된 경우, yield를 하는 경우 등이 있으며 단순하게 list에 push back하는 것이 아닌 list\_insert\_ordered함수를 호출하여 쓰레드의 priority순서에 맞게 ready\_list에 쓰레드를 추가해준다. 이 함수에서 추가는 thread\_compare\_priority에 따른다. list\_insert\_ordered함수는 lib/kernel디렉토리의 list.c에 구현이 되어 있는데, 인자로 받은 less함수가 true를 나타내는 순간 해당 쓰레드를 리스트에 insert하는 방식이다. 이 less함수가 thread\_compare\_priority함수에 해당하며 추가하고자 하는 쓰레드와 ready\_list리스트 안에 있는 쓰레드들의 priority를 비교하며 ready\_list에 쓰레드들이 priority내림차순으로 insert될 수 있도록 실행을 한다. 결론적으로, ready\_list에 맨 앞에 있는 running thread보다 priority가 높은 쓰레드가 추가가 되면 해당 쓰레드가 ready\_list의 맨 앞에 추가가 되고, schedule함수에 의해 CPU제어권을 ready\_list 맨 앞에 추가가 된 쓰레드가 가지게 된다.

3)

nice : -20 ~ 20사이의 값을 가지는 value로 큰 nice값을 가지는 쓰레드 일수록 다른 쓰레드에 CPU점유권을 양보한다. 쓰레드의 priority계산은 매 4ticks마다 재계산이 되는데, priority = PRI\_MAX – (recent\_cpu / 4) - (nice \* 2) 식으로 재계산이 되며 식에서 볼 수 있듯이 nice값이 클수록 priority가 낮은 것을 확인할 수 있다.

recent\_cpu : 해당 쓰레드가 얼마나 CPU를 점유했는지를 나타내는 value이다. running thread상태에서 쓰레드의 recent\_cpu값은 1tick마다 1씩 증가하며, 값이 클수록 더 많이 CPU를 점유했음을 의미하기 때문에 priority가 재계산이 되는 priority = PRI\_MAX – (recent\_cpu / 4) - (nice \* 2) 식에서 볼 수 있듯이 priority가 낮아진다.

load\_avg : ready상태에 있는 쓰레드의 평균 수를 나타내는 value이다. 즉 수행 가능한 쓰레드의 평균 개수를 나타내는 값인데, 이 값이 크면 클수록 recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* recent\_cpu + nice 로 계속 업데이트가 되는 recent\_cpu가 천천히 감소를 하고, 작으면 빠르게 감소를 한다. 이렇게 되면 priority는 load\_avg가 클수록 천천히 증가하고 작을수록 빠르게 증가를 하는데, 이는 ready상태에 있는 쓰레드가 많을수록 모든 쓰레드가 골고루 CPU제어권을 받을 수 있게 하기 위해서이다.

load\_avg는 매 tick마다 load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_thread 로 재계산이 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11/1 : alarm clock 구현

11/2 ~ 11/4 : busy waiting 구현

11/5 : advanced scheduler 추가 구현

11/6 : 보고서 작성

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
  + 수정하거나 추가해야 하는 자료구조
  + 수정하거나 추가해야 하는 함수
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
  2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
  3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

1)

blocked상태의 쓰레드를 따로 추가할 수 있는 sleep\_list자료구조가 필요하다. 또한 쓰레드를 재우는 timer\_sleep함수에서 쓰레드를 재울 때 언제 일어날 것인지를 저장하는 wakeup변수를 쓰레드 구조체에 추가를 해주어야 하며 timer\_sleep함수에서 wakeup변수에 재우고자 하는 시간인 ticks를 저장하고 sleep\_list에 이 쓰레드를 추가한 뒤 thread\_block하는 부분을 추가하였다. 또한 쓰레드를 깨울 때는, ticks를 인자로 받아 특정 쓰레드의 wakeup변수가 ticks보다 작거나 같은 경우 sleep\_list에서 해당 쓰레드를 제거하고 thread\_unblock시키는 함수인 thread\_awake함수를 구현하여 매 1ms마다 호출이 되는 timer\_interrupt함수에서 이 thread\_awake함수를 호출하는 코드를 추가하였다.

2)

실행 가능한 쓰레드들을 가지고 있는 ready\_list에는 자고 있는 쓰레드가 unblock이 되거나 thread\_set\_priority등에서 thread\_yield를 호출하는 등의 경우에서 추가가 되는데, priority순서에 맞춰서 추가를 해야 하므로 lib/kernel/의 list.c에 구현이 되어 있는 list\_insert\_ordered를 기존에 작성이 되어 있는 list\_push\_back대신에 사용한다. priority비교를 위해 thread\_compare\_priority함수를 따로 구현을 하였으며, 이 함수는 추가를 하고자 하는 쓰레드가 priority내림차순으로 정렬이 되어 있는 ready\_list안의 특정 쓰레드보다 priorty가 높은 경우 true를 반환하고, 이 함수를 list\_insert\_ordered함수에 인자로 넘겨줌으로써 해당 자리에 쓰레드를 ready\_list에 추가가 될 수 있도록 구현하였다.

3)

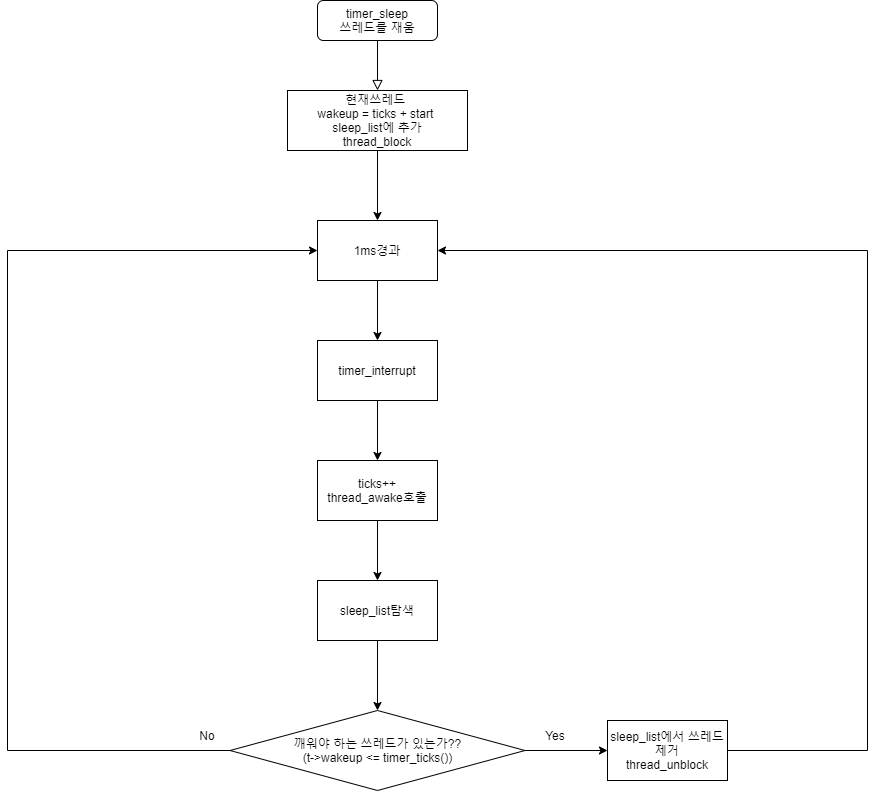
쓰레드 구조체에 int형 변수 recent\_cpu와 nice를 추가하였다. 또한 update\_thread함수를 구현하였는데, 이는 매 1초마다 thread\_tick함수에서 호출이 되며 각 쓰레드의 recent\_cpu변수를 recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* recent\_cpu + nice 와 같은 식으로 update를 해준다. 또한 이 함수에서 thread.c에 전역 변수로 선언이 되어 있는 load\_avg또한 load\_avg = (59/60) \* load\_avg + (1/60) \* ready\_thread 식에 맞춰 update를 해준다. 또한 매 4ms마다 thread\_tick에서 호출이 되는 update\_prioirty를 구현하였는데, 이는 모든 쓰레드의 priority를 재계산 하는 함수로, priority = PRI\_MAX – (recent\_cpu / 4) - (nice \* 2)의 식을 따를 수 있도록 구현하였다.

이 과정에서, 단순한 계산이 아닌 부동소수점 계산을 실행할 수 있어야 하기 때문에 thread.c에 FRACTION을 이용한 fixed\_point연산 관련 함수인 f\_to\_i\_round, i\_sub\_f, i\_mul\_f, f\_add\_i, f\_i, i\_f, f\_div, f\_mul\_f, f\_div\_f, f\_add\_f, f\_sub\_f, f\_div\_i를 구현하였다. 이 함수를 이용하여 load\_avg, priority, recent\_cpu값을 계산하거나 업데이트 하는 과정을 실행하였다.

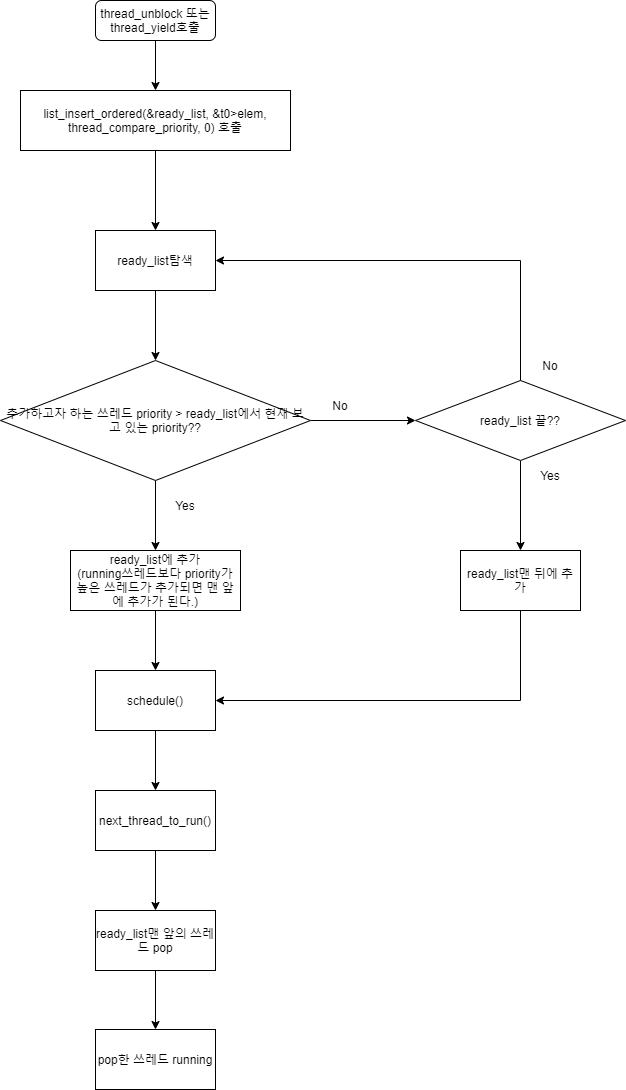
1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
  2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.

1)



2)



* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
  1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
  2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
  3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

1)

쓰레드를 재우는 timer\_sleep함수이다.

void

timer\_sleep (int64\_t ticks)

{

int64\_t start = timer\_ticks ();

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

struct thread \*cur;

enum intr\_level old\_level;

old\_level = intr\_disable ();

cur = thread\_current ();

ASSERT (cur != idle\_thread);

cur->wakeup = ticks + start;

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

thread\_block ();

intr\_set\_level (old\_level);

}

쓰레드 구조체에 선언을 한, 언제까지 재울 것인지를 나타내는 wakeup변수에 ticks + start(timer\_ticks())만큼 쓰레드를 sleep시키는 함수이다. sleep\_list에 해당 재운 쓰레드를 push\_back하고 thread\_block시킨다.

다음은 timer\_interrupt이다.

static void

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

{

ticks++;

thread\_tick();

//thread\_awake (ticks);

thread\_awake();

}

1ms마다 계속 호출이 되는 timer\_interrupt에서 깨워야 하는 쓰레드가 있는지를 확인하고 있다면 해당 쓰레드를 unblock시키는 thread\_awake함수를 호출하는 방식으로 자고 있는 쓰레드를 깨웠다.

thread\_awake함수는 thread.c에 다음과 같이 구현이 되어 있다.

void thread\_awake (void)

{

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

if(e == NULL)

return;

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

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

if(t->wakeup <= timer\_ticks()){

e = list\_remove (e);

thread\_unblock (t);

}

else

e = list\_next (e);

}

}

sleep\_list를 탐색하면서 그 안에 있는 쓰레드 중 wakeup변수가 현재 timer\_ticks보다 작거나 같은 경우 sleep\_list에서 해당 쓰레드를 제거하고 thread\_unblock를 호출하는 방식으로 자고 있는 쓰레드를 깨우는 함수이다.

2)

running쓰레드보다 우선순위가 높은 쓰레드가 들어오는 경우는 실행 가능한 쓰레드들을 가지고 있는 ready\_list에 쓰레드의 우선 순위로 정렬을 하여 insert하는 방식으로 처리를 해주어야 한다. ready\_list에 쓰레드가 추가되는 경우는 unblock이 되거나 yield를 하는 경우인데 다음과 같이 코드를 수정하였다.

oid

thread\_unblock (struct thread \*t)

{

enum intr\_level old\_level;

ASSERT (is\_thread (t));

old\_level = intr\_disable ();

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

list\_insert\_ordered(&ready\_list, &t->elem, thread\_compare\_priority, NULL);

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

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

}

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

기존에 단순히 ready\_list뒤에 삽입하는 list\_push\_back방식 대신, priority순서에 맞춰 삽입이 될 수 있도록 list\_insert\_ordered함수로 변경하였다. 이 함수는 정렬 기준 함수가 필요한데, 이는 thread\_compare\_prioirty로 구현하였다. 이 함수는 다음과 같다.

bool

thread\_compare\_priority (struct list\_elem \*l, struct list\_elem \*s, void \*aux UNUSED)

{

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

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

}

ready\_list를 탐색하면서 추가하고자 하는 thread의 priority와 ready\_list안의 쓰레드의 priority를 비교하며 ready\_list안 쓰레드의 priority가 추가하고자 하는 쓰레드보다 낮은 경우 그 자리에 삽입하고자 하는 쓰레드를 추가하였다.

static struct thread \*

next\_thread\_to\_run (void)

{

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

return idle\_thread;

else

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

}

running쓰레드를 지정하는 함수인데, 위와 같이 priority내림차순으로 정렬하여 ready\_list에 쓰레드를 삽입하면 running thread보다 높은 priority를 가지는 쓰레드가 삽입이 되었을 때, next\_thread\_to\_run함수가 ready\_list에서 맨 앞의 쓰레드를 뽑기 때문에 추가가 된 쓰레드가 running thread가 될 것이다.

3)

해당 부분 구현을 위해서는 부동 소수점 연산이 필요한데, 이 기능을 수행해주는 함수들은 thread.c에 다음과 같이 구현하였다.

int f\_to\_i\_round (int x) {

if (x >= 0)

return (x + FRACTION / 2) / FRACTION;

return (x - FRACTION / 2) / FRACTION;

}

int i\_sub\_f(int x, int y) {

return x\*FRACTION - y;

}

int i\_mul\_f(int x, int y) {

return x\*y;

}

int f\_add\_i(int x, int y) {

return x + y\*FRACTION;

}

int f\_i(int x){

return x / FRACTION;

}

int i\_f(int x){

return x\*FRACTION;

}

int f\_div(int x, int y){

return ((int64\_t) x)\*FRACTION / y;

}

int f\_mul\_f(int x, int y) {

int64\_t temp = x;

temp = temp \* y / FRACTION;

return (int)temp;

}

int f\_div\_f(int x, int y) {

int64\_t temp = x;

temp = temp \* FRACTION / y;

return (int)temp;

}

int f\_add\_f(int x, int y) {

return x + y;

}

int f\_sub\_f(int x, int y) {

return x - y;

}

int f\_div\_i(int x, int y) {

return x / y;

}

이 함수를 이용해, recent\_cpu와 nice 및 쓰레드의 priority를 재계산하고 업데이트를 한다.

void

thread\_tick (void)

{

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

/\* Update statistics. \*/

if (t == idle\_thread)

idle\_ticks++;

#ifdef USERPROG

else if (t->pagedir != NULL)

user\_ticks++;

#endif

else

kernel\_ticks++;

/\* Enforce preemption. \*/

if (++thread\_ticks >= TIME\_SLICE){

intr\_yield\_on\_return ();

}

if(thread\_prior\_aging || thread\_mlfqs){

thread\_aging();

}

}

thread\_tick에서 aging이 필요하거나(thread\_prior\_aging == true) 추가 구현에서 mlfq를 사용하는 경우(thread\_mlfqs == true) thread\_aging함수를 호출한다.

void thread\_aging(void){

thread\_current()->recent\_cpu = f\_add\_i(thread\_current()->recent\_cpu, 1);

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

update\_thread();

if (timer\_ticks() % 4 == 0)

update\_priority();

}

이 함수에서 현재 쓰레드의 recent\_cpu를 1증가시키고, 초 단위로 시간이 경과한 경우 update\_thread를, 4ms가 경과된 경우는 update\_priority함수를 호출한다.

void update\_thread(void) {

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

struct thread\* t;

struct list\_elem\* e;

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

ready\_threads += 1;

}

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

if(e == NULL)

return;

load\_avg = f\_div\_i(f\_add\_i(i\_mul\_f(59, load\_avg), ready\_threads), 60);

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

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

if (t != idle\_thread) {

int two\_load\_avg = i\_mul\_f(2, load\_avg);

int divide = f\_div\_f(two\_load\_avg, f\_add\_i(two\_load\_avg, 1));

t->recent\_cpu = f\_add\_i(f\_mul\_f(divide, t->recent\_cpu), t->nice);

}

e = list\_next(e);

}

}

이 함수에서 매 초마다 load\_avg를 load\_avg = f\_div\_i(f\_add\_i(i\_mul\_f(59, load\_avg), ready\_threads), 60);와 같이 update한다. 또한 모든 쓰레드들의 recent\_cpu도 업데이트 해줘야 하는데, all\_list를 순회하면서 업데이트를 하였다.

void update\_priority() {

struct thread\* t;

struct list\_elem\* e;

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

if(e == NULL)

return;

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

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

t->priority = f\_sub\_f(f\_sub\_f(f\_add\_i(0, PRI\_MAX), f\_div\_i(t->recent\_cpu, 4)), i\_mul\_f(2, f\_add\_i(0, t->nice))) / FRACTION;

if(t->priority < PRI\_MIN)

t->priority = PRI\_MIN;

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

t->priority = PRI\_MAX;

e = list\_next(e);

}

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

intr\_yield\_on\_return();

}

4ms마다 쓰레드의 priority를 업데이트 하는 함수이다. PRI\_MIN, PRI\_MAX바운드를 지키면서 t->priority = f\_sub\_f(f\_sub\_f(f\_add\_i(0, PRI\_MAX), f\_div\_i(t->recent\_cpu, 4)), i\_mul\_f(2, f\_add\_i(0, t->nice))) / FRACTION;와 같이 all\_list를 순회하면서 모든 쓰레드들의 우선순위 값을 업데이트를 하였다.

* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명

priority-lifo.c에서는 우선 i=0에서부터 16을 나타내는 THREAD\_CNT까지 쓰레드를 생성하는데, 쓰레드의 우선 순위 값은 PRI\_DEFAULT + 1 + i이고, 쓰레드의 id는 i이다. 즉 쓰레드 아이디가 클수록 우선 순위 값도 크다고 볼 수 있는데, 다만 먼저 thread\_create되는 것은 쓰레드 id가 낮은(i = 0부터 15까지)쓰레드이다. 즉 ready list에 추가가 되는 것은 우선 순위가 낮은 쓰레드부터 추가가 된다고 볼 수 있다. 이후 쓰레드들을 running하는데 running이 되는 부분이 해당 출력 결과에서 (priority-lifo) iteration: …… 이다. 숫자 출력을 보면 15부터 0까지 내림차순으로 출력이 되는 것을 볼 수 있는데, 이 숫자가 의미하는 것은 바로 앞서 thread\_create를 하는 부분에서 assign을 한 쓰레드의 아이디이다. 즉 ready list에 추가가 된 순서는 0, 1, 2 ….. 15순으로 추가가 되었지만 실행이 되는 순서는 쓰레드 아이디랑 우선 순위가 비례하기 때문에 15, 14, 13 …. 0 순서로 실행이 되는 마치 Last – in – First – Out과 같은 동작을 보인다는 것을 알 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명