**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

학번 / 이름 : 20165181 / 문상영

개발 기간 : 10 / 15 ~ 11 / 04

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

기존의 busy waiting으로 구현되었던 timer\_sleep()을 수정한다. sleep 된 thread의 tick을 확인해서 깨워주는 방식을 사용해 inefficiency가 보완된 Alarm Clock을 구현한다.

priority를 고려하지 않고 Round Robin으로 구현되어 있는 현재의 Pintos scheduler를 수정해서 Priority 를 반영한 Scheduler로 구현한다.

추가 구현으로 BSD Scheduler를 구현해 priority를 64개의 priority level 을 가진 multilevel Feedback Queue 를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

timer 가 expire 되지 않으면 thread\_yield() 를 이용해 반복적으로 ready state로 되돌리는 busy waiting 방법 대신, timer가 expire 되지 않았다면 sleep 시켜 놓았다가, tick의 증가에 따라 thread 를 깨우는 방식으로 Alarm Clock 을 구현해서 cpu자원을 효효율적으로 사용하는 timer sleep 방식을 구현할 수 있다.

* 1. Priority Scheduling

기존의 pintos에서 scheduling 하는 방식은 priority를 고려하지 않는 Round Robin 방식으로, thread\_yield() 또는 thread\_unblock() 이 호출되면 현재 thread 또는 unblocked 된 thread 가 ready queue 의 끝에 push 된다. 이러한 현재 pintos scheduler를 priority를 반영해 thread를 수행하게 만들면, resource를 좀 더 효율적으로 사용하는 scheduler를 구현할 수 있다. 또한 priority 가 낮은 thread의 starvation 문제를 해결하기 위해서 aging을 사용해서 scheduler를 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Priority Scheduler를 구현하고 나서 nice, recent\_cpu, load\_avg 등을 고려해 Advanced 된 Scheduler를 만들기 위해 64 level의 priority를 가진 multi level feedback queue를 구현한다. 이를 이용해 다양한 scheduling needs를 균형있게 만족시킬 수 있고 따라서 general purpose scheduling을 할 수 있다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

기존의 timer\_sleep() 경우, busy waiting을 이용해서 반복문으로 tick을 체크하고 만약 아직 tick이 충분히 지나지 않았다면 thread\_yield() 함수를 호출해 Readyqueue 에 다시 push 하는 방식으로 timer\_sleep() 구현했다. 하지만 좀 더 효율적으로 thread를 sleep 시키기 위해서 다른 방법이 필요하다. 해결책으로 tick 이 증가하는 시점에, Blocked 상태의 thread들의 wake up time 을 체크해 해당 thread가 깨어나야할 시간이 지나면 그때 ready list에 추가하고, Ready 상태로 변경해 주는 방식을 사용한다. 이를 위해서 sleep된 thread를 따로 list로 관리해 주기 위해서 blocked\_list를 선언하고 각 thread의 wake up time 을 계산해 현재와 비교한다음, 만약 wake up time 이 지났다면 thread를 unblock 하고 ready list에 insert 해서 thread를 깨운다. 매 tick 마다 이 과정을 반복한다.

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

우선 readylist에 priority에 맞게 thread가 insert되어야 priority 순으로 정렬된다. multilevel feedback queue로 구현할 경우, priority 에 해당하는 64 level의 ready queue에서 thread의 priority에 해당하는 ready queue에 thread가 push된다.

priority scheduling에 따르면 Running thread가 현재 가장 priority 가 높은 thread 여야 하므로 running thread 보다 높은 priority를 가진 thread 가 들어올 경우, 현재 Thread는 cpu에서 deschedule 되고 priority가 가장높은 thread가 다시 cpu에 schedule 된다.

이를 위해서 만약 running thread 보다 priority 가 높은 thread가 readylist 에 들어온다면 thread\_yield()를 사용해 running thread의 수행을 멈추고 thread 가 새로 schedule 될 수 있게끔 한다. 이로 인해 기존의 running thread 의 priority가 더 낮은 경우에는 해당 thread가 다시 readystate가 되고 ready list에 추가된다.

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

Advanced Scheduler에서 priority 계산에 필요한 각 요소는 recent\_cpu, nice 이고, recent\_cpu를 계산하기 위해서 load\_avg 가 필요하다.

nice 의 경우 값이 -20~20의 범위를 가지고 값이 양의 방향으로 클수록 priority를 낮춘다. 그리고 recent\_cpu는 thread의 CPU time 을 추정하는 요소로, 마찬가지로 값이 클수록 priority를 낮춘다.

따라서 priority를 계산하기 위해 필요한 요소에는 recent\_cpu, nice, load\_avg가 있고, 이를 update 하는 계산식은 다음과 같다.

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

recent\_cpu=(2\*load\_avg + (1/60) \* number\_of\_ready\_threads

load\_avg = (59/60)\*load\_avg + (1/60)\* number\_of\_ready\_threads

이 때, pintos kernel 은 floating-point arithmetic을 지원하지 않기 때문에, fixed pointer arithmetic을 implement 해야하고, 이를 사용해서 priority와, priority를 구하기 위해서 필요한 recent\_cpu, nice, load\_avg를 계산해야한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10/15 ~10/16: 프로젝트 분석, 코드 분석 및 설계

10/17 ~ 10/18 : Alarm Clock 구현

10/19 ~ 10 /29 : Priority Scheduler 구현 (시험기간)

10/30~11/1 : Advanced Scheduler 구현

11/2 ~ 11/4 : 코드 검증

11/5 : 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.(Alarm Clock) :

기존의 timer\_sleep() 의 경우, thread의 wake up time을 체크해 만약 아직 wake up time 이 지나지 않았다면 Ready queue 에 다시 thread를 push 하는 방식으로 구현되어 있었다.

이는 반복문을 이용하는 busy waiting 방식으로, thread를 sleep 시키기 위한 목적으로는 비효율적이다. 따라서 이를 개선하기위해서 tick이 증가할 때 마다, Blocked 상태의 thread들의 wake up time 을 체크해 해당 thread가 깨어나야할 시간이 된다면 그 때 Ready 상태로 변경해 주는 방식을 사용하기로 했다.

이를 위해서 sleep된 thread를 따로 list로 관리해 주기 위해서 **blocked\_list를 만들고** 각 thread의 wake up time 을 현재와 비교해야한다. 따라서 thread.h 에 선언된 **thread 구조체에 계산된 wakeup time을 저장하는 변수를 추가한다( ticks\_to\_wake )**. 그리고 timer.c 에서 sleep 상태의 thread를 관리하기 위해서 **blocked 된 thread로 구성된 list 자료구조를 선언**한다. list 자료구조는 initialize가 필요하므로 새롭게 선언한 list를 **list\_init을 이용해 초기화 한다.**

Alarm clock을 구현하기 위해서 src/device/timer.c 에 정의된 **timer\_interrupt() 함수를 수정해서** 매 tick마다 주기적으로 sleep 상태의 thread를 검사하기위해 blocked\_list를 순회하면서 thread의 wake up time을 확인한다. 만약 wake up time 이 지났다면 blocked\_list에서 해당 thread를 제거하고 unblock 시켜 ready state로 만든다. 이후 schedule() 이 호출되어 ready list에 있는 thread 중에 하나가 policy에 맞게 cpu에 assign 된다.

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

우선 ready list를 총 64개의 level을 가진 multilevel feedback queue를 구현하기 위해서, 기존의 **ready list를 64개의 list로 이루어진 array로 수정**했다. 이에 따라서 priority를 index로 해서 각 list에 접근할 수 있고, next\_thread\_to\_run() 함수에서는 **priority 가 높은 list 부터 thread를 확인해서 반환하도록 코드를 수정**했다.

이러한 자료구조에서 새로운 thread가 Ready list에 들어올 경우, 우선 thread 는 priority 에 알맞은 list( ready\_list[ priority ] )에 push 된다. 이 때, running thread 가 현재 가장 높은 priority를 가져야 하므로, 만약 새롭게 insert 되는 thread 가 더 높은 priority를 가지면, thread\_yield()를 호출해 running thread를 ready state로 전환하고 다시 cpu에 thread를 schedule 한다.

이렇게 되면 ready list에는 thread\_yield()로 인해 다시 ready state가 된 running thread와 새롭게 추가된 thread가 있는데, 새롭게 추가된 thread의 priority 가 더 높다고 했으므로, schedule 함수에서 호출되는 next\_thread\_to\_run() 함수는 더 높은 priority를 가진, 새로 추가된 thread를 반환한다. 따라서 cpu에는 새로 추가된 thread가 schedule 된다.

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

Advanced Scheduler에서 priority 계산에 필요한 각 요소는 recent\_cpu, nice 이고, recent\_cpu를 계산하기 위해서는 load\_avg, nice가, load\_avg를 계산하기 위해서는 ready\_thread의 개수가 필요하다.

**nice 와 recent\_cpu**는 각 thread 마다 존재하는 요소로, thread가 처음 생성된다면 0으로 초기화되고, 아니라면 부모 thread로부터 상속받는다. load\_avg는 시스템이 boot 될 때 0으로 초기화되는 system wide value 이다. 따라서 **nice와 recent\_cpu는 thread 구조체 요소로 선언**되고, **init\_thread() 에서 값을 상속**받을 수 있도록 함수를 수정한다.

이렇게 priority를 계산하기 위해 필요한 요소들은 모두 recent\_cpu, nice, load\_avg가 있고, 이를 update 하기 위한 계산식은 다음과 같다.

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

recent\_cpu=(2\*load\_avg + (1/60) \* number\_of\_ready\_threads

load\_avg = (59/60)\*load\_avg + (1/60)\* number\_of\_ready\_threads

이 때, pintos kernel 은 floating-point arithmetic을 지원하지 않기 때문에, fixed pointer arithmetic을 implement 해야한다. **이를 위해서 fixed point 연산을 도와주는 함수를 작성한다.**

int f\_add\_i(int i, int f);

int i\_sub\_f(int i, int f);

int i\_mul\_f(int i, int f);

int f\_div\_i(int f,int i);

int f\_add\_f(int f1, int f2);

int f\_sub\_f(int f1, int f2);

int f\_mul\_f(int f1,int f2);

int f\_div\_f(int f1, int f2);

해당 함수들을 이용해서 fixed pointer 연산을 수행하고 priority와 그에 필요한 recent\_cpu, load\_avg를 update 한다.

명세서에서 recent\_cpu 와 load\_avg는 매 초마다 update 시키고, priority는 4 ticks 마다 update하라고 했으므로 (recent\_cpu는 매 tick 마다 1 증가 ) **timer\_interrupt가 호출될 때 이를 확인하고 update 하도록 함수를 수정**한다.

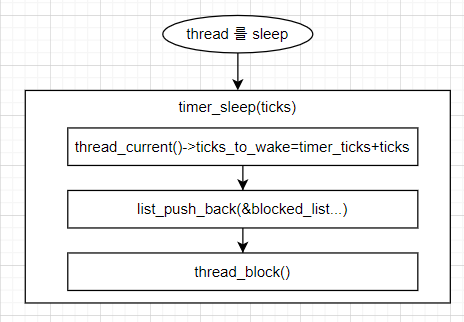
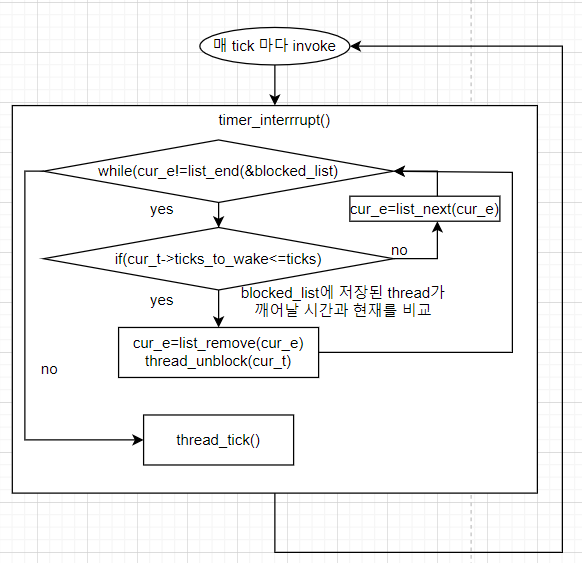
명세서에 명시된 **thread\_set\_priority(), thread\_get\_priority(), thread\_get\_nice(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu(), thread\_set\_nice() 함수를 구현**한다. 또 모듈화를 위해 **load\_avg와 recent\_cpu를 갱신해주는 함수를 정의**하고, 갱신된 load\_avg와 recent\_cpu 정보를 사용해 priority를 계산, 갱신해주는 **update\_priority() 함수도 작성한다**.

Advanced Scheduler의 경우 -aging, -mlfqs 옵션에 따라 수행될 수 있도록 flag 선언하고, src/threads/init.c 의 **parse\_options() 함수에 내용을 추가**해서 command 의 옵션에 따라 해당 flag가 set 되도록 한다.

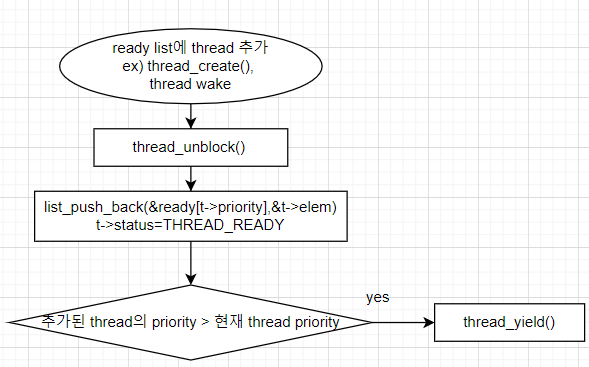
1. **연구 결과**
   1. **Flow Chart**

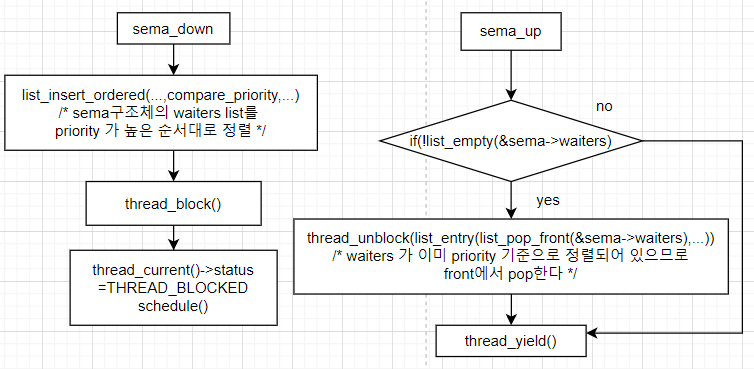
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)

1) Alarm Clock

2) Priority Scheduling





* 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

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술**.( Alarm Clock )**

효율적으로 thread를 sleep 시키기 위해, Blocked 상태의 thread를 wake up time 을 체크해 해당 thread가 깨어나야할 시간이 된다면 그때 Ready 상태로 변경한다.

따라서 Blocked 상태의 thread를 관리해주기 위해 src/device/timer.c에 list 자료구조를 새로 선언했다.

----------------------------------------------------------------------------------------------

/\*blocked queue(list)\*/

static struct list blocked\_list;

----------------------------------------------------------------------------------------------

그리고 앞서 말했듯, blocked\_list를 순회하면서 저장된 thread의 wake up time을 확인해야 하므로 이를 저장하는 변수를 thread 구조체에 추가한다.

----------------------------------------------------------------------------------------------struct thread{

...

int64\_t ticks\_to\_wake;

...

};

----------------------------------------------------------------------------------------------

timer\_sleep() 에서 sleep 되는 시간을 정하게 되므로, timer\_sleep() 내부에서 ticks\_to\_wake 에 thread 가 깨어나야할 시간을 저장하고 blocked\_list에 해당 thread를 push 한뒤, thread\_block()을 이용해 block 상태로 만든다.

---------------------------------------------------------------------------------------------

void

timer\_sleep (int64\_t ticks)

{

...

old\_level=intr\_disable();

thread\_current()->ticks\_to\_wake=start+ticks;

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

thread\_block();

intr\_set\_level(old\_level);

}

----------------------------------------------------------------------------------------------

주의 해야 할 점으로 list에 원소를 추가하거나 제거할 때, shared data에 대한 접근이므로interrupt를 disable 시켜야한다. 위의 함수에서 start는 timer\_ticks() 값으로 현재 tick의 값이다. 따라서 ticks\_to\_wake은 timer\_ticks()+ticks 이므로 thread를 깨워야하는 시점이 된다.

이제 timer\_interrupt() 함수에서 sleep 상태의 thread 들이 저장된 blocked\_list를 반복문을 이용해 순회하고 ticks\_to\_wake와 현재 tick을 비교한다. 만약 현재 tick이 ticks\_to\_ wake를 이미 지났다면 해당 thread를 blocked\_list에서 제거하고 thread\_unblock() 을 호출해 ready 상태로 만든다.

----------------------------------------------------------------------------------------------

static void

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

{

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

ticks++;

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

struct thread\* cur\_t=list\_entry(cur\_e,struct thread, elem);

if(cur\_t->ticks\_to\_wake<=ticks){

cur\_e=list\_remove(cur\_e);

thread\_unblock(cur\_t);

}

else{

cur\_e=list\_next(cur\_e);

}

}

...

thread\_tick ();

}

----------------------------------------------------------------------------------------------

timer\_interrupt 함수내에서 위의 작업이 완료되고 thread\_tick()를 호출하는데, thread\_tick()함수는 내부에서 schedule()을 호출한다. schedule() 함수는 다음으로 수행할 thread를 next\_thread\_to\_run() 에서 반환값으로 받아 switch를 수행한다.

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

새로운 thread 가 ready list에 들어오는 경우에 해당 thread의 priority에 따라 ready list 상에서 적합한 list에 위치해야한다. 따라서 thread는 해당되는 priority level의 list에 인덱싱해서 push 되고, 각 priority는 round robin으로 스케쥴링 되도록 했다.

thread가 ready list에 추가되는 경우는 크게 두가지로 볼 수 있는데 예를 들면 첫번째는thread\_create() 에서 thread가 생성될 때 thread\_unblock() 이 호출되는데, 이 때, thread가 ready state로 바뀌면서 ready list에 추가되게 된다.

또 thread\_yield() 의 경우에도 현재 thread가 idle\_thread 가 아닌경우에는 current thread를 다시 ready list에 추가하게 되는데 이 때도 다음과 같이 thread를 ready state로 바꾸고 ready list에 thread를 추가한다.

----------------------------------------------------------------------------------------------

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->priority], &cur->elem);

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

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->priority], &t->elem);

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

----------------------------------------------------------------------------------------------

이렇게 thread가 ready list에 추가되는 경우인 thread\_create() 와 timer\_interrupt() 함수에서 thread\_unblock() 을 수행할 때, 해당 thread의 priority 가 현재 수행중인 thread의 prioroity 보다 큰지 확인하고 만약 더 크다면 **thread\_yield()를 호출해서 priority 가 큰 thread 가 수행될 수 있도록 한다.**

이 때 next\_thread\_to\_run() 함수가 priority 가 높은 thread를 return 하므로 자동적으로 추가된 thread 가(priority가 높은 thread) 수행되게 된다.

tid\_t

thread\_create (const char \*name, int priority,

thread\_func \*function, void \*aux)

{

...

/\* Add to run queue. \*/

thread\_unblock (t);

if(priority>thread\_get\_priority()){

thread\_yield();

}

return tid;

}

static struct thread \*

next\_thread\_to\_run (void)

{

for(int i=PRI\_MAX;i>=0;i--){

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

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

}

}

return idle\_thread;

}

----------------------------------------------------------------------------------------------

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

(Advanced Scheduler)

Advanced Scheduler에서 priority 계산에 필요한 각 요소는 recent\_cpu, nice 이고, recent\_cpu를 계산하기 위해서 load\_avg, nice 가 필요하다.

load\_avg는 시스템이 boot 될 때 0으로 초기화되는 system wide value 이고, nice 와 recent\_cpu는 각 thread의 property 이다. nice와 recent\_cpu는 thread가 처음 생성된다면 0으로 초기화되고, 아니라면 부모 thread로부터 상속받기 때문에 이를 위해서 **thread\_init() 함수와 init\_thread() 함수에서 이 값들을 다음과 같이 초기화** 시켰다.

----------------------------------------------------------------------------------------------

void

thread\_init (void)

{

...

load\_avg=0; // average of the number of thread in READY state

/\* Set up a thread structure for the running thread. \*/

initial\_thread = running\_thread ();

initial\_thread->nice=0;// if it is created first, the value is 0.

initial\_thread->recent\_cpu=0; // if it is created first, the value is 0.

...

}

static void

init\_thread (struct thread \*t, const char \*name, int priority)

{

...

t->recent\_cpu=running\_thread()->recent\_cpu; //inherits value of the parent thread;

t->nice=running\_thread()->nice; //inherits value of the parent thread;

...

}

----------------------------------------------------------------------------------------------

이렇게 priority를 계산하기 위해 필요한 요소들은 모두 recent\_cpu, nice, load\_avg가 있고, 이에 대한 계산식은 위에서 말했듯 다음과 같다.

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

recent\_cpu=(2\*load\_avg + (1/60) \* number\_of\_ready\_threads

load\_avg = (59/60)\*load\_avg + (1/60)\* number\_of\_ready\_threads

pintos kernel 은 floating-point arithmetic을 지원하지 않기 때문에, fixed pointer arithmetic을 구현하기위해 이를 수행하는 함수를 작성했고, 해당 함수들을 이용해서 priority 계산을 fixed pointer 연산으로 수행한다.

----------------------------------------------------------------------------------------------

#define INT\_PART (1<<14)

int f\_add\_i(int f, int i){

return f+i\*INT\_PART;

}

int i\_sub\_f(int i, int f){

return i\*INT\_PART-f;

}

int i\_mul\_f(int i, int f){

return i\*f;

}

int f\_div\_i(int f,int i){

return f/i;

}

int f\_add\_f(int f1, int f2){

return f1+f2;

}

int f\_sub\_f(int f1, int f2){

return f1-f2;

}

int f\_mul\_f(int f1,int f2){

int64\_t temp=f1;

temp=temp\*f2/INT\_PART;

return (int)temp;

}

int f\_div\_f(int f1, int f2){

int64\_t temp=f1;

temp=temp\*INT\_PART/f2;

return (int)temp;

}

int get\_max\_priority(void){

for(int i=PRI\_MAX;i>=0;i--){

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

return i;

}

return -1;

}

----------------------------------------------------------------------------------------------

INT\_PART 는 하위 14bit 가 FRACTION 부분이므로 14만큼 bit shift 한것이다. f\_div\_f 나 f\_mul\_f 함수의 같은 경우, 두 소수의 곱이 들어가므로 overflow 가 발생하지 않도록 int64\_t 자료형에서 곱하고 다시 나누어 값을 반환했다.

다음으로 위의 함수들을 이용해 **recent\_cpu, load\_avg, priority 를 계산하는 함수를 작성**한다. 이 함수들은 위의 계산식을 참고해서 작성했다.

----------------------------------------------------------------------------------------------

int

calc\_load\_avg(int R\_threads){

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

}

int

calc\_recent\_cpu(struct thread\* t){

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

return f\_add\_i(f\_mul\_f(f\_div\_f(tmp,f\_add\_i(tmp,1)),t->recent\_cpu),t->nice);

}

int

calc\_priority(struct thread\* t){

return 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)))/INT\_PART;

}

----------------------------------------------------------------------------------------------

다음으로 R\_threads를 구해서 load\_avg를 구하고 recent\_cpu를 update하는 과정을 모듈화해서 함수 update\_recent\_cpu\_load\_avg()를 작성한다. 또한 priority를 계산하는 과정도 따로 모듈화해서 update\_priority() 함수를 작성한다.(둘이 update되는 주기가 다르기 때문에 따로 모듈화한다.)

----------------------------------------------------------------------------------------------

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

int R\_threads=0;

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

R\_threads+=1;

for(int i=0;i<=PRI\_MAX;i++){

R\_threads+=list\_size(&ready\_list[i]);

}

load\_avg=calc\_load\_avg(R\_threads);

//printf("\nLOAD AVG: %d\n",load\_avg);

struct list\_elem\* cur\_e;

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

struct thread\* cur\_t=list\_entry(cur\_e,struct thread,allelem);

if(cur\_t==idle\_thread){

continue;

}

cur\_t->recent\_cpu=calc\_recent\_cpu(cur\_t);

}

}

void update\_priority(){

struct list\_elem\* cur\_e;

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

struct thread\* cur\_t=list\_entry(cur\_e,struct thread, allelem);

int calc=calc\_priority(cur\_t);

if(calc>PRI\_MAX){

cur\_t->priority=PRI\_MAX;

}

else if(calc<PRI\_MIN){

cur\_t->priority=PRI\_MIN;

}

else{

cur\_t->priority=calc;

}

}

}

----------------------------------------------------------------------------------------------

마지막으로 timer\_interrupt() 함수에서 mlfqs flag와 thread\_prior\_aging을 check 해서 만약 flag가 true 로 set 되어있다면 위의 함수를 이용해서 각 요소들을 update시킨다. init.c 의 parse\_options() 함수에서 -mlfqs, -aging 옵션에 대한 flag set을 처리해준다. 이 때, 명세서에 recent\_cpu 와 load\_avg는 매 초마다 update 시키고, priority는 4 ticks 마다 update하라고 명시했으므로 (recent\_cpu는 매 tick 마다 1 증가 ) 이를 모듈러 연산을 통해 검사하여 update\_recent\_cpu\_load\_avg()를 호출하고 update한다.

----------------------------------------------------------------------------------------------

static void

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

{

...

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

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

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

if(timer\_ticks()%TIMER\_FREQ==0){ // load\_avg and recent\_cpu updates every second

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

}

if(timer\_ticks()%4==0){ // priority is recalculated every 4 ticks

update\_priority();

if(get\_max\_priority()>cur\_t->priority)

intr\_yield\_on\_return();

}

}

thread\_tick ();

}

----------------------------------------------------------------------------------------------

명세서에 명시된 thread\_set\_priority(), thread\_get\_priority(), thread\_set\_nice(), thread\_get\_ nice(), thread\_ get\_load\_avg(), thread\_get\_recent\_cpu()함수를 구현한다

이중에 thread\_set\_priority() 와 thread\_set\_nice() 의 경우 값이 변경되면 priority 가 변경되므로 만약 변경한 값이 더 작다면, 현재 running thread의 priority가 가장 큰 상태가 아닐 수 있으므로 thread\_yield()를 호출해서 다시 scheduling을 했다.

----------------------------------------------------------------------------------------------

void

thread\_set\_priority (int new\_priority)

{

if(thread\_mlfqs)

return;

int cur\_priority=thread\_current()->priority;

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

if(new\_priority<cur\_priority){

thread\_yield();

}

}

/\* Returns the current thread's priority. \*/

int

thread\_get\_priority (void)

{

return thread\_current ()->priority;

}

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

void

thread\_set\_nice (int nice)

{

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

cur\_t->nice=nice;

update\_priority();

if(get\_max\_priority()>cur\_t->priority)

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.

Rounded to the nearest integer.

\*/

int

thread\_get\_load\_avg (void)

{

return i\_mul\_f(100,load\_avg)/INT\_PART;

}

/\*

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

Rounded up the nearest integer.

\*/

int

thread\_get\_recent\_cpu (void)

{

return i\_mul\_f(100,thread\_current()->recent\_cpu)/INT\_PART;

}

----------------------------------------------------------------------------------------------

* **개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명**

개발중 발생한 이슈로, ready list, blocked list 자료구조가 shared data이므로 접근해서 값을 변경하거나 할 때 interrupt 가 걸리면 data가 손상될 수 있다. 이를 해결하기 위해서 intr\_disable() 을 이용해서 interrupt를 끄고 list 에 작업을 한 다음 다시 interrupt를 복구하는 방법을 사용했다.

또한 priority-sema 테스트에서 priority 와 sema를 동시에 적용하는데, 이 때, sema가 관리하는 waiters list는 priority를 고려하지 않은채 thread가 push 되어 있어 이를 위해서 sema\_down 으로 waiters list에 thread 가 추가되는 경우, compare 함수를 이용해서 정렬 삽입하는 **list\_insert\_ordered라는 기존의 함수를 사용**했고, priority 내림차순으로 정렬하기 위해서 **compare\_priority 라는 함수를 정의**했다.

----------------------------------------------------------------------------------------------

void

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\_push\_back(&sema->waiters,&thread\_current()->elem);

list\_insert\_ordered (&sema->waiters, &thread\_current ()->elem, compare\_priority, NULL);

thread\_block ();

}

sema->value--;

intr\_set\_level (old\_level);

}

bool

compare\_priority(const struct list\_elem\* s, const struct list\_elem\* e,void\* aux)

{

struct thread\* thread\_s=list\_entry(s,struct thread, elem);

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

return thread\_s->priority > thread\_e->priority;

}

----------------------------------------------------------------------------------------------

마지막으로 발견한 이슈중 하나는 timer\_interrupt() 함수가 수행 중일 때, 내부에서 thread \_yield()를 호출해야하는 경우, 바로 호출하는 것이 아니라, interrupt\_handler가 return 된 후에 yield() 가 발생도록 해야했는데, 이를 수행하기 위해서 **기존의 함수 중 intr\_yield\_on\_return()** 이라는 함수를 사용해서 문제를 해결했다.

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

텍스트, 모니터, 검은색, 화면이(가) 표시된 사진

자동 생성된 설명

priority-lifo.c 의 코드의 내용중 위의 반복문을 보면 반복을 진행하면서 i를 1씩 증가시키고, priority가 PRI\_DEFAULT +1 + i 인 thread를 thread\_create()를 이용해 생성하는 것을 확인 할 수 있다. 다시말해 priority가 높은 thread가 더 늦게 추가되게 된다. 올바른 실행은 priority 가 높은 thread가 더 빨리 수행되어야 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

두번째 반복문에서 d->id 를 출력하는데 여기에는 i 값이 저장되었으므로 높은 priority를 가지면 i의 값이 크기 때문에 내림차순으로 출력 되어야 한다.

* 텍스트이(가) 표시된 사진

  자동 생성된 설명

priority-lifo 테스트 결과에 대한 코드 분석으로는 위에서 예상한 바와 같이 이후에 추가된 thread들이 더 높은 priority를 가졌을 때, 더 먼저 수행되는 것을 확인할 수 있다.

* make check 수행 결과를 캡처하여 첨부

텍스트이(가) 표시된 사진

자동 생성된 설명