**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

조 / 조원 : 20201654 최호진

개발 기간 : 11/14 ~ 11/22

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

효율적인 thread 관리를 위한 내용을 구현한다. Alarm clock, priority scheduling, 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

PintOS는 default waiting이 Busy-waiting으로 구현되어 있어, while문을 이용해 thread\_yield() 함수를 계속 호출하게 되어, 굉장히 비효율적이라는 문제점이 있다. 이를 tick을 이용하여 thread가 sleep 상태가 되도록 구현함으로써 효율적인 waiting을 할 수 있도록 한다.

* 1. Priority Scheduling

기본적으로 PintOS는 Round-Robin Scheduling 방식을 이용한다. 이렇게 되면, priority를 고려하지 않아 우선순위와 관계없이 새롭게 Ready List에 들어오는 thread는 항상 리스트 마지막에 추가된다. 본 프로젝트에서는 Priority Scheduling 방식을 도입해 우선순위를 고려하여 Preempt할 수 있도록 구현한다. 한편, 우선순위가 낮은 Thread는 Starvation 문제에 발생할 수 있는데 이를 방지하기 위해 Priority aging을 구현하여 시간이 지남에 따라 우선순위가 높아질 수 있도록 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Load\_avg, recent-cpu, nice와 같은 변수를 도입해 BSD Scheduler(MLFQ)를 구현한다. 이 때, 해당 변수를 계산하기 위해서는 소수점 계산이 필요한데 이를 위한 함수들도 같이 구현한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

tick마다 blocked state에 있는 thread를 모두 검사하여 정해진 wakeup\_tick보다 지난 경우 thread\_unblock 함수를 통해 unblock 상태로 바꿔주어 해당 thread를 ready list에 추가한다. 이때, blocked state의 thread를 관리하기 위한 새 list를 만들어준다.

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

현재 running thread보다 높은 priority(우선순위)를 가진 thread가 ready list에 새롭게 들어오면 새롭게 들어온 thread를 실행해야 한다(프로세서를 새 thread에 넘겨줘야 한다). 따라서 현재 running state의 thread를 ready list에 추가하고, 이때 마찬가지로 priority 값에 따라 알맞은 위치에 추가한다. 그 다음 Ready list의 가장 앞에 있는 (Priority가 가장 높은 새롭게 들어온 Thread) thread가 running state가 되도록 한다.

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

Priority 계산을 위해 3가지 변수: load\_avg, recent\_cpu, nice가 필요하다.

load\_avg는 thread.c 내의 전역 변수로 ready state의 thread의 수의 평균을 추정한다. 먼저 0으로 초기화된 상태로 시작하고, 매초마다 load\_avg 값이 업데이트 된다. 계산 식은 아래와 같다.

recent\_cpu와 nice는 thread 구조체 내의 변수로 먼저, 두 값 모두 thread 생성 시 0으로 초기화 된다. 자식 thread의 경우는 부모 thread의 값을 받는다.

recent\_cpu는 thread의 CPU 타임을 의미한다. Time interrupt 발생 시 현재 running 상태의 thread는 recent\_cpu 값이 1 증가한다. 또한, 매초마다 모든 thread의 recent\_cpu 값을 재계산하는데, 이때 계산식은 아래와 같다.

nice는 -20 ~ 20의 값을 갖고, nice가 높을수록 priority가 낮음을 의미하는데, 따라서 nice 값이 업데이트 되면 priority도 재계산 해주어야 한다. priority의 계산식은 아래와 같다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
  1. 11/14 ~ 11/14: Alarm Clock 구현
  2. 11/15 ~ 11/17: Priority Scheduling 구현
  3. 11/17 ~ 11/19: Advanced Scheduler 구현
  4. 11/20 ~ 11/22: 보고서 작성
  5. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명

Thread와 관련한 Alarm clock, priority scheduling, advanced scheduler를 구현한다.

* 1. Alarm Clock

앞서, 기존 PintOS에는 기본적으로 busy waitig 방식으로 구현되어 있다고 하였다. “devices/timer.c”에서 while문을 이용해 busy waiting 방식으로 구현되어 있는 부분의 코드를 수정해야 한다. 이를 위해서 blocked state의 thread를 관리할 새로운 리스트 (struct list sleep\_list)를 선언하고 timer\_init 시 해당 리스트를 초기화한다. 또한 thread\_sleep 함수를 thread.c 파일 내에 선언하여 timer\_sleep 함수에서 이 함수를 호출해 blocked된 thread를 위에서 선언한 list에 추가할 수 있도록 한다. timer\_interrupt 함수에서는 tick을 증가시키면서, 조건에 맞는 blocked threads를 awake 시킬 수 있도록 한다. 해당 코드는 thread.c 함수에 thread\_wakeup 함수를 추가해 timer\_interrupt에서 해당 함수를 호출하여 사용할 수 있도록 구현한다.

* 1. Priority Scheduling

기존에 Round-Robin Scheduling 방식으로 구현된 것을 Priority Scheduling이 되도록 수정하여야 한다. 먼저, 기본적으로 ready list에 새 thread가 추가될 때 기존에는 단순히 push back 했지만, piority를 고려하여 삽입할 수 있도록 수정한다. (list\_push\_back 함수를 list\_insert\_ordered 함수로 교체) Priority를 비교하여 삽입할 수 있도록 priority 비교를 위한 함수를 새롭게 선언한다. Priority를 set하는 함수에서 새로 들어온 thread의 우선순위가 높으면 thread\_yield() 함수를 호출해 preempt 할 수 있도록 한다.

또한, Starvaition을 막기 위해 Boolean type의 변수의 flag를 추가하고, aging 조건을 체크하는 조건문을 추가한다. 또한, aging을 위해 시간이 지남에 따라 priority 값이 증가하도록 구현한다.

* 1. Advanced Scheduler

BSD Scheduler를 구현하는데, 먼저 쓰레드 구조체 내에 recent\_cpu와 nice 변수를 추가, thread.c에 전역변수로 load\_avg 변수를 추가한다. 또한, 이들을 계산하기 위해서는 소수점 연산이 필요한데, 따로 PintOS에서 지원하지 않으므로, fixed\_point.h 헤더파일을 추가해 소수점 연산을 위한 함수들을 구현한다.

구현 명세서에 따라 매초마다 load\_avg와 recent\_cpu값을 재계산, 그리고 4 ticks 마다 priority를 재계산한다. 각각의 변수를 재계산하기 위한 함수들을 선언하고, 마찬가지로 기존에 thread.c 파일에 nice, recent\_cpu와 관련하여 구현이 미완성되어 있는 부분을 구현한다.

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

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

1. Priority Scheduling

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명 스크린샷, 라인, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

* 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. Alarm Clock

Thread.c 파일에 blocked thread를 관리하기 위한 리스트를 선언하고 thread\_init 함수에서 다른 리스트들(Ready list, All list)과 동일하게 초기화 해준다.

/\* List of process in THREAD\_BLOCKED state\*/

static struct list sleep\_list;

void

thread\_init (void)

{

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

lock\_init (&tid\_lock);

list\_init (&sleep\_list);

list\_init (&ready\_list);

list\_init (&all\_list);

thread.c 파일에 blocked state의 thread를 sleep list에 push back 하는 thread\_sleep 함수를 선언 및 구현한다.

void thread\_sleep(int64\_t ticks) {

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

// Disable interrupts

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

if (t == idle\_thread) {

intr\_set\_level(old\_level);

return;

}

t->status = THREAD\_BLOCKED;

t->wakeup\_tick = ticks;

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

schedule();

// Enable interrupts

intr\_set\_level(old\_level);

}

devices/timer.c 파일에서 기존 busy waiting으로 구현된 timer\_sleep을 수정하고 위에서 구현한 함수를 호출하여 사용할 수 있도록 한다.

void

timer\_sleep (int64\_t ticks)

{

int64\_t start = timer\_ticks ();

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

thread\_sleep(ticks + start);

}

또한, blocked state의 thread를 깨우는 로직이 구현된 thread\_wakeup 함수를 thread.c 파일에 구현하고, 이를 timer\_interrupt 함수에서 호출해 사용할 수 있도록 한다.

thread\_wakeup 함수는 특정 시간이 지난 blocked thread들을 sleep list에서 pop 해 unblock 하는 역할을 한다.

void thread\_wakeup(int64\_t ticks){

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

struct thread \*t;

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

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

if (t->wakeup\_tick <= ticks) {

e = list\_remove(e);

thread\_unblock(t);

}

else {

e = list\_next(e);

}

}

}

/\* Timer interrupt handler. \*/

static void

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

{

ticks++;

thread\_tick ();

thread\_wakeup(ticks);

}

* 1. Priority Scheduling

먼저, synch.c 파일에 각 thread의 priority를 비교하기 위한 함수인 cmp\_priority 함수를 선언한다.

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

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

}

이 함수를 이용해 기존에 ready list에 단순히 push back으로 구현(list\_push\_back)된 부분을 모두 list\_insert\_ordered 함수로 수정해 priority 값에 따라 ready list에 정렬되도록 수정한다.

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\_insert\_ordered(&ready\_list, &t->elem, (list\_less\_func \*) &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, (list\_less\_func \*) &compare\_priority, NULL);

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

또한, Starvaition을 해결하기 위해 aging 옵션을 궇현해야 한다. 먼저, init.c에 “-aging” 문자열을 확인하는 조건문을 체크해 aging 모드인지 확인한다.

#ifndef USERPROG

else if (!strcmp (name, "-aging"))

thread\_prior\_aging = true;

#endif

Thread.c 파일에 aging을 위한 Boolean type의 flag를 선언한다.

#ifndef USERPROG

bool thread\_prior\_aging;

#endif

매 tick마다 thread의 priority를 증가시키는 Aging을 위한 함수를 선언한다.

void thread\_aging(void) {

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

struct thread \*t;

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

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

if (t->priority < PRI\_MAX) {

t->priority++;

}

e = list\_next(e);

}

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

t->priority = PRI\_MIN;

}

}

해당 함수는 thread\_tick 함수에서 매 tick마다 호출된다.

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 ();

#ifndef USERPROG

if (thread\_prior\_aging == true) {

thread\_aging();

}

#endif

if (thread\_mlfqs) {

// increment recent\_cpu of current thread

int64\_t ticks = timer\_ticks();

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

thread\_current()->recent\_cpu = add\_x\_to\_n(thread\_current()->recent\_cpu, 1);

}

// recaclulate load\_avg, recent\_cpu of all threads, priority every 1 second

if (ticks % TIMER\_FREQ == 0) {

// recalculate load\_avg

recalculate\_load\_avg();

// recalculate recent\_cpu

recalculate\_recent\_cpu();

}

// recalculate priority of current thread every 4 ticks

if (ticks % TIME\_SLICE == 0) {

recalculate\_priority();

}

}

}

* 1. Advanced Scheduler

먼저, 3-B에서 설명했듯이 소숫점 계산이 필요한데 이를 위한 함수들을 fixed-point.h 헤더파일을 만들어서 해당 파일 내에 작성하고 thread.c에서 해당 헤더파일을 사용할 수 있도록 한다.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <stdint.h>

#define F (1 << 14)

int integer\_to\_fp(int n);

int fp\_to\_integer(int x);

int fp\_to\_integer\_round(int x);

int add\_x\_to\_y(int x, int y);

int add\_x\_to\_n(int x, int n);

int sub\_y\_from\_x(int x, int y);

int sub\_n\_from\_x(int x, int n);

int mul\_x\_by\_y(int x, int y);

int mul\_x\_by\_n(int x, int n);

int div\_x\_by\_y(int x, int y);

int div\_x\_by\_n(int x, int n);

int integer\_to\_fp(int n){

return n \* F;

}

int fp\_to\_integer(int x){

return x / F;

}

int fp\_to\_integer\_round(int x){

if (x >= 0) return (x + F / 2) / F;

return (x - F / 2) / F;

}

int add\_x\_to\_y(int x, int y){

return x + y;

}

int add\_x\_to\_n(int x, int n){

return x + n \* F;

}

int sub\_y\_from\_x(int x, int y){

return x - y;

}

int sub\_n\_from\_x(int x, int n){

return x - n \* F;

}

int mul\_x\_by\_y(int x, int y){

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

}

int mul\_x\_by\_n(int x, int n){

return x \* n;

}

int div\_x\_by\_y(int x, int y){

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

}

int div\_x\_by\_n(int x, int n){

return x / n;

}

Thread 구조체에 Thread 객체마다 필요한 nice, recent\_cpu 변수를 추가하고, thread.c 파일에 전역변수 load\_avg 변수를 선언한다.

[thread.h]

struct thread

{

...

int nice;

int recent\_cpu;

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

[thread.h]

int load\_avg;

그 다음 프로젝트 조건에 따라, 매초마다 load\_avg와 recent\_cpu 값을 갱신해야 하고, 4 tick 마다 priority를 갱신해야 한다. 이를 위한 함수들을 선언한다.

[thread.c – recalculate\_load\_avg/recent\_cpu/priority]

void recalculate\_load\_avg(void){

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

if (thread\_current() != idle\_thread) ready\_threads++;

int temp = mul\_x\_by\_y(div\_x\_by\_y(integer\_to\_fp(59), integer\_to\_fp(60)), load\_avg);

load\_avg = add\_x\_to\_y(temp, mul\_x\_by\_n(div\_x\_by\_y(integer\_to\_fp(1), integer\_to\_fp(60)), ready\_threads));

}

void recalculate\_recent\_cpu(void){

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

struct thread \*t;

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

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

if (t != idle\_thread) {

int recent\_cpu = mul\_x\_by\_n(load\_avg, 2);

int temp = div\_x\_by\_y(recent\_cpu, add\_x\_to\_n(recent\_cpu, 1));

temp = add\_x\_to\_n(mul\_x\_by\_y(temp, t->recent\_cpu), t->nice);

t->recent\_cpu = temp;

}

e = list\_next(e);

}

}

void recalculate\_priority(void){

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

struct thread \*t;

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

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

// if the thread is not idle thread, recalculate priority

if (t != idle\_thread) {

int p = sub\_y\_from\_x(integer\_to\_fp(PRI\_MAX), div\_x\_by\_n(t->recent\_cpu, 4));

p = sub\_y\_from\_x(p, integer\_to\_fp((t->nice) \* 2));

t->priority = fp\_to\_integer(p);

if (t->priority > PRI\_MAX) t->priority = PRI\_MAX;

else if (t->priority < PRI\_MIN) t->priority = PRI\_MIN;

}

e = list\_next(e);

}

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

e = list\_front(&ready\_list);

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

// ready\_list의 제일 앞에 있는 thread의 priority가 더 높으면 yield

if (t->priority > thread\_get\_priority()) intr\_yield\_on\_return();

}

함수 내 계산 식은 2-B의 계산식에 따라 작성하였다. 그리고, 해당 함수들은 thread\_tick 함수에서 호출하게 된다.

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 ();

#ifndef USERPROG

if (thread\_prior\_aging == true) {

thread\_aging();

}

#endif

if (thread\_mlfqs) {

// increment recent\_cpu of current thread

int64\_t ticks = timer\_ticks();

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

thread\_current()->recent\_cpu = add\_x\_to\_n(thread\_current()->recent\_cpu, 1);

}

// recaclulate load\_avg, recent\_cpu of all threads, priority every 1 second

if (ticks % TIMER\_FREQ == 0) {

// recalculate load\_avg

recalculate\_load\_avg();

// recalculate recent\_cpu

recalculate\_recent\_cpu();

}

// recalculate priority of current thread every 4 ticks

if (ticks % TIME\_SLICE == 0) {

recalculate\_priority();

}

}

}

이 함수들 외에 기존에 구현이 완성되어 있지 않았던 핀토스의 함수들의 구현을 완성한다.

void

thread\_set\_priority (int new\_priority)

{

// Reorder the ready list based on the new priority

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

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

if (new\_priority < old\_priority) thread\_yield();

}

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

int

thread\_get\_priority (void)

{

return thread\_current ()->priority;

}

Thread\_set\_priority 함수에서는 old\_priority와 새로 바뀌는 priority를 비교해 만약 새로 바뀐 priority가 기존 값보다 작을 경우 yield 한다.

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

void

thread\_set\_nice (int nice UNUSED)

{

thread\_current()->nice = nice;

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

int p = add\_x\_to\_y(integer\_to\_fp(PRI\_MAX), div\_x\_by\_n(thread\_current()->recent\_cpu, -4));

p = add\_x\_to\_y(p, integer\_to\_fp(-2 \* (thread\_current()->nice)));

thread\_current()->priority = fp\_to\_integer(p);

if (thread\_current()->priority > PRI\_MAX) {

thread\_current()->priority = PRI\_MAX;

}

else if (thread\_current()->priority < PRI\_MIN) {

thread\_current()->priority = PRI\_MIN;

}

}

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

struct list\_elem\* e=list\_front(&ready\_list);

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

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

thread\_yield();

}

}

}

Nice 값을 바꾸는 함수에서는 nice 값이 갱신될 때 마찬가지로 priority도 변경해주어야 한다.

그 외에 값을 반환해주는 함수들도 구현한다. 이때, 핀토스 매뉴얼에 따라 fp를 integer 형태로 바꾸어 반환해줘야 하는 load\_avg, recent\_cpu는 fixed-point.h에서 선언한 함수를 이용해 계산하여 반환한다.

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

int

thread\_get\_nice (void)

{

return thread\_current()->nice;

}

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

int

thread\_get\_load\_avg (void)

{

return fp\_to\_integer\_round(mul\_x\_by\_n(load\_avg, 100));

}

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

int

thread\_get\_recent\_cpu (void)

{

return fp\_to\_integer\_round(mul\_x\_by\_n(thread\_current()->recent\_cpu, 100));

}

* 1. **시험 및 평가 내용**

priority-lifo.c는 동일한 priority를 갖는 thread를 여러개 생성해, Round-Roblin Scheduling 방식으로 실행되는지를 확인한다. Main thread의 priority는 PRI\_DEFAULT 값으로, 15부터 실행되는 것을 확인할 수 있다. Ready list에 있는 Thread들은 정렬된 순서대로 실행된다.

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

프로젝트에서 요구하는 테스트 케이스들을 모두 통과했다.