**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 강상원 / 20191559

개발 기간 : 2023. 10. 30. ~ 2023. 11. 5.

1. **개발 목표**

Alarm clock, priority scheduling, advanced scheduler(MLFQS)를 구현한다.

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

기본 구현 전, PintOS의 스케쥴러는 busy-waiting으로 되어 있다. 이는 상당히 비효율적으로, tick을 계산해 일정 시간 동안 thread가 동작하지 않을 때 sleep 시키는 방식으로 바꾼다. 기준이 되는 tick을 알맞게 지정하여 비효율성을 해소한다.

* + 1. Priority Scheduling

이번 프로젝트 구현 전에는 Round-Robin 방식으로 스케쥴러가 작동한다. ready\_list의 끝에 priority에 관계없이 push\_back되는 문제가 있다. 따아서 Priority Scheduling을 구현하기 위하여 현재 thread보다 우선순위가 높은 thread가 들어온다면 이를 preempt한다. 이렇게만 한다면 또 낮은 우선순위 thread가 하염없이 기다리는 starvation 문제가 발생하므로, priority aging(시간에 따라 우선순위 상승)으로 문제를 해결한다.

* + 1. Advanced Scheduler

BSD Scheduler(Multi-Level Feedback Queue; MLFQ)를 구현한다. nice, priority, load-avg 값 등을 이용하여 우선순위를 계산해야 하는데, 소수점 계산을 기본 PintOS가 지원하지 않아서 이를 처리하는 로직 또한 작성해야 한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

앞서 말한 대로 지정한 timer tick보다 thread가 지났으면 thread\_unblock() 함수를 호출하여 wake up 한다. Blocked 상태의 thread들을 저장하는 List에서 후보군을 계속 찾으며, wake할 때 목록에서 제거한다.

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

현재 running thread보다 높은 우선순위를 가지고 있다면 Priority Scehduling의 의의에 따라 yield하고 새로운 thread에 processor를 넘겨준다. 실행 중이던 thread는 ready list에 다시 추가한다. list\_insert\_ordered()를 통해 정렬된 삽입을 한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

과제 명세서 p.38에 따라, 초기 priority는 thread\_create()에서 결정되고, 매 4 틱마다 시스템에 있는 모든 우선순위가 새로 계산된다. 계산식은 다음과 같다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
      * 10/30 ~ 10/30: Alarm Clock
      * 11/01 ~ 11/02: Priority Scheduling
      * 11/03 ~ 11/05: Advanced Scheduler
      * 11/06 ~ 11/06: 보고서 작성
   2. **개발 방법**
      1. Alarm Clock
         * [devices/timer.c]에서 기존 busy waiting으로 되어 있는 scheduling 방식을 바꿔 준다.
         * struct list sleep\_list를 선언하고 timer\_init()에서 list\_init(&sleep\_list)으로 초기화한다.
         * timer\_sleep()에서 wake time을 세서 옮겨 주고 list\_push\_back()으로 sleep list에 thread를 추가하는 로직을 작성한다.
         * timer\_interrupt에서 tick을 증가시키면서, 조건에 맞는 thread들을 wake up시킨다.
      2. Priority Scheduling
         * Priority를 set 하는 부분에서 새로 들어온 thread의 priority가 현재 실행 중인 thread보다 높으면 thread\_yield()를 통해 preempt하게 만든다.
         * [threads/thread.c]에서 bool thread\_aging을 추가하여 aging flag 역할을 하게끔 한다.
         * [threads/init.c]에서 strcmp로 “-aging”을 체크하는 조건문을 추가하여 aging thread를 처리할 수 있게 한다.
         * [threads/thread.c]에서 comp\_priority()라는 비교 함수를 생성하여, list\_insert\_ordered()의 기준 함수로 기능할 수 있게끔 한다.
         * 기존 함수들에서 list\_push\_back()으로 되어 있는 호출을 list\_insert\_ordered()으로 바꿔 주어 yield, unblock할 때 정렬된 상태로 삽입되도록 한다.
      3. Advanced Scheduler
         * 소수점 연산을 할 수 있게끔 각 recent\_cpu, nice 값을 계산하는 함수를 작성 완료한다. fixed-point number의 예시:
         * [threads/thread.h]에서 load\_avg를 선언한다.
         * 초기 priority는 thread\_create()에서 결정되고, 매 4 틱마다 시스템에 있는 모든 우선순위가 앞서 설명한 수식에 따라 새로 계산된다.
         * [threads/thread.h]에서 각 thread마다 필요한 nice 값, recent\_cpu를 저장할 수 있게끔 인자를 추가한다.
         * 모든 thread의 recent\_cpu 값을 갱신하는 update\_recent\_cpu()함수, load avg 값을 갱신하는 update\_load\_avg() 함수를 추가한다.
         * 명세서에 적힌 대로 thread\_get\_nice()와 thread\_set\_nice() 함수를 구현 완료한다.
2. **연구 결과**
   1. **Flow Chart**
      * Alarm Clock

스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* + - Priority Scheduling

스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
     1. Alarm Clock

struct *list* sleep\_list;

struct list sleep\_list를 선언하여 sleep 상태에 있는 thread 관리하는 리스트를 만들고, timer\_init에서 이를 초기화한다.

void

timer\_init (void)

{

pit\_configure\_channel (0, 2, TIMER\_FREQ);

intr\_register\_ext (0x20, timer\_interrupt, "8254 Timer");

list\_init(&sleep\_list);

}

[devices/timer.c]에서 기존 busy waiting으로 되어 있는 scheduling 방식을 바꿔 준다.

void

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

{

// Time to wake up

*int64\_t* end\_time = timer\_ticks() + *ticks*;

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

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

struct *thread* \*current\_thread = thread\_current();

current\_thread->wake\_time = end\_time;

// Add to sleep\_list

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

thread\_block();

intr\_set\_level(old\_level);

}

end\_time을 timer\_ticks() + ticks로 초기화하고, Sleep 할 때 현재 실행중인 thread를 sleep list에 추가한다.

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

ticks++;

thread\_tick();

// Wake up sleeping threads

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

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

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

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

e = list\_remove(e);

thread\_unblock(t);

} else {

e = list\_next(e);

}

}

...

}

timer\_interrupt에서 tick을 증가시키면서, 조건에 맞는 thread들을 wake up시킨다.

* + 1. Priority Scheduling

else if (!strcmp (name, "-aging")) // Added in #Proj 3

thread\_aging = true;

[threads/init.c]에 strcmp로 “-aging”을 체크하는 조건문을 추가하여 aging thread를 처리할 수 있게 한다.

bool thread\_aging; // Added in Proj #3

[threads/thread.c] 에서 bool thread\_aging을 추가하여 aging flag 역할을 하게끔 한다.

bool comp\_priority(struct *list\_elem* \**a*, struct *list\_elem* \**b*, void \**aux* *UNUSED*){

struct *thread* \*t1 = list\_entry(*a*, struct thread, elem);

struct *thread* \*t2 = list\_entry(*b*, struct thread, elem);

return t1->priority > t2->priority;

}

[threads/thread.c]에서 comp\_priority()라는 비교 함수를 생성하여, list\_insert\_ordered()의 기준 함수로 기능할 수 있게끔 한다.

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* \*) &comp\_priority, NULL); // Added in Proj #3

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->elem);

list\_insert\_ordered(&ready\_list, &*t*->elem, (*list\_less\_func* \*) &comp\_priority, NULL); // Added in Proj #3

*t*->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

기존 함수들에서 list\_push\_back()으로 되어 있는 호출을 list\_insert\_ordered()으로 바꿔 주어 yield, unblock할 때 정렬된 상태로 삽입되도록 한다.

void

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

{

enum intr\_level old\_level;

ASSERT (sema != NULL);

old\_level = intr\_disable ();

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

struct list\_elem \*e;

struct thread \*highest\_priority\_thread = NULL;

int highest\_priority = -1;

FOR\_LIST(e, &sema->waiters) {

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

if (t->priority > highest\_priority) {

highest\_priority = t->priority;

highest\_priority\_thread = t;

}

}

if (highest\_priority\_thread != NULL) {

list\_remove (&highest\_priority\_thread->elem);

thread\_unblock (highest\_priority\_thread);

}

}

sema\_up()에서 waiting 중인 thread 중 가장 큰 priority를 찾아 Unblock, 또한 알맞게 yield할 수 있게 수정한다.

* + 1. Advanced Scheduler

struct *thread*

{

...

/\* Added in #Proj 3 \*/

*int64\_t* wake\_time; /\* Time ticks to wakeup \*/

*int64\_t* recent\_cpu; /\* Recent CPU time \*/

*int64\_t* nice; /\* Nice value \*/

};

[threads/thread.h]에서 각 thread마다 필요한 nice 값, recent\_cpu를 저장할 수 있게끔 인자를 추가한다.

static int load\_avg; // Added in Proj #3

또한, load\_avg를 선언하여 해당 값을 저장할 수 있게 한다.

/\* Updates Load Avg

Added in #Proj. 3 \*/

void update\_load\_avg() {

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

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

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

}

/\* Updates recent\_cpu of all threads

Added in #Proj. 3 \*/

void update\_recent\_cpu() {

struct *list\_elem* \*e;

FOR\_LIST(e, &all\_list) {

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

if (t != idle\_thread) {

int load\_avg\_twice = 2 \* load\_avg;

int coeff = (load\_avg\_twice \* FRACTION) / (load\_avg\_twice + FRACTION);

t->recent\_cpu = (coeff \* t->recent\_cpu) / FRACTION + t->nice \* FRACTION;

}

}

}

모든 thread의 recent\_cpu 값을 갱신하는 update\_recent\_cpu()함수, load avg 값을 갱신하는 update\_load\_avg() 함수를 추가한다. update\_load\_avg(), update\_recent\_cpu()에서는 소수점 계산을 처리하는데, II.B.3에서 서술한 바와 같이 계산한다.

/\* Timer interrupt handler. \*/

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

ticks++;

thread\_tick();

...

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

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

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

update\_load\_avg();

update\_recent\_cpu();

update\_all\_priority();

} else if (timer\_ticks() % 4 == 0) {

update\_all\_priority();

}

}

}

timer\_interrupt() 함수 내 thread\_mlfqs에 대한 예외 케이스를 추가한다. load\_avg, recent\_cpu, priority를 수정한다. 매 4틱마다 갱신한다.

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

void

thread\_set\_nice (int *nice* *UNUSED*)

{

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

t->nice = *nice*;

update\_thread\_priority(t);

if (t->priority < get\_max\_priority()) thread\_yield();

}

nice 값을 set하는 함수를 완성한다. 현재 우선순위가 최대 우선순위보다 작으면 yield한다.

최대 우선순위를 구하기 위해 get\_max\_priority() 함수를 선언하여 찾는다. 이는 ready\_list를 traverse하면서 maximum priority 값을 찾아 반환한다.

/\* Get max priority in ready\_list

Added in #Proj. 3 \*/

int get\_max\_priority() {

int max\_priority = -1;

struct *thread* \*t;

// Find max priority in ready\_list by traversing

struct *list\_elem* \*e;

FOR\_LIST(e, &ready\_list) {

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

if (t->priority > max\_priority) max\_priority = t->priority;

}

return max\_priority;

}

thread의 priority를 수정하는 코드에서는, [PRI\_MIN, PRI\_MAX]의 범위에 있도록 제한을 걸어서 올바르게 설정할 수 있도록 구성한다.

/\* Update one thread's priority, check range.

Added in #Proj. 3 \*/

void update\_thread\_priority(struct *thread* \**t*) {

int priority\_base = PRI\_MAX \* FRACTION - *t*->recent\_cpu / 4;

int priority\_nice = 2 \* *t*->nice \* FRACTION;

*t*->priority = (priority\_base - priority\_nice) / FRACTION;

// Priority should be in range [PRI\_MIN, PRI\_MAX]

*t*->priority = *t*->priority > PRI\_MAX ? PRI\_MAX : *t*->priority;

*t*->priority = *t*->priority < PRI\_MIN ? PRI\_MIN : *t*->priority;

}

update\_all\_priority() wrapper 함수를 선언하여 모든 thread들의 priority들의 일시 갱신을 용이하게 한다.

/\* Update priority of all threads

Added in #Proj. 3 \*/

void update\_all\_priority() {

struct *list\_elem* \*e;

FOR\_LIST(e, &all\_list) {

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

update\_thread\_priority(t);

}

if (thread\_current()->priority < get\_max\_priority()) intr\_yield\_on\_return();

}

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

priority\_lifo는 같은 우선순위를 가지는 여러 thread를 생성하고, Round-Robin에 의거하여 실행되는지 확인한다. Main thread의 priority는 PRI\_DEFAULT로, 15 thread부터 실행됨을 볼 수 있다. Ready list에 있는, 정렬된 순서로 thread가 실행된다.

텍스트, 전자제품, 스크린샷, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

* + - make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

과제 명세서에 있는 모든 기본 구현사항과 더불어 추가 구현까지 모두 완료하였다.