**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : 이은지

개발 기간 : 2020/12/1 ~ 2020/12/7

1. **개발 목표**

1) Process Scheduling – 기존의 Round Robin 스케줄링을 개선하여 더 효율적인 스케줄링 구현

2) Threads – Thread에 기능을 추가하여 Thread를 더 효율적으로 관리

3) Aging 및 Priority 추가 연산 – Starvation을 방지 및 효율성 증대.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock – 프로세스를 중지하였을 때 시스템 자원을 낭비하는 것을 방지
  2. Priority Scheduling – 우선순위를 고려하거나 갱신하여 ready\_list에서 관리하고 효율적인 프로세스 처리
  3. Advanced Scheduler (추가구현을 한 경우) – 최근 사용 여부와 빈도를 고려하여 더 효율적인 스케줄러 구현
  4. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

큐로 작동하는 ready\_list에 스레드를 block하여 insert한 뒤, 스레드를 깨울 때는 ready\_list의 모든 entry를 순회하며 현재 tick이 깨울 tick을 지났다면 ready\_list에서 remove한다.

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

thread\_unblock(), thread\_yield() 수정하여 ready\_list 정렬한 뒤 여기서 최우선순위를 가진 스레드와 현재 스레드의 우선순위를 비교하여 만약 현재 스레드의 우선순위가 더 작다면 재스케줄링 후 스위치를 일으킨다.

즉시 현재 스레드를 중지하고 더 높은 priority를 가진 스레드에 CPU를 양보한다.

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

- nice : 다른 스레드에 얼마나 양보하는가(how nice the thread should be to other threads). 최소값은 -20이고 최대값은 20이고, 만약 음수면 우선순위를 증가하고, 양수면 우선순위를 감소하며, 0이면 우선순위에 영향을 주지 않는다.

- recent\_cpu : 해당 스레드가 가장 최근 CPU를 사용한 기간

- load\_avg : 최근 1분 동안 수행 가능한 프로세스의 평균 개수

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

12/1 ~ 12/2 : 이론 공부 및 내용 이해

12/2 ~ 12/4 : Alarm Clock 구현

12/5 ~ 12/5 : Synchronization을 제외한 Priority Scheduling 구현

12/6 : Synchronization 및 Priority aging 구현

12/7 : Advanced Scheduler 구현

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
* **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.**

thread 구조체 내 sleepend\_tick 요소 추가

timer\_sleep(int64\_t ticks)에 파라미터 추가하여 스레드가 자는 시간 지정

thread\_awake(int64\_t current\_tick)에 파라미터 추가하여 스레드가 깨어나도록 함

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

thread\_unblock(), thread\_yield() 수정하여 ready\_list 정렬 및 최우선순위 스레드 추출

priority 비교해주는 함수 구현

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

thread 구조체 내에 recent\_cpu 와 nice 요소 추가

timer\_interrupt에서 시간의 흐름에 따라 priority 재계산 및 recent\_cpu 업데이트

thread\_set\_nice(), thread\_get\_nice(), thread\_get\_recent\_cpu(), update\_recent\_cpu() 구현

이외 부동소수점 관련 연산 함수 추가(nearest\_int)

* + 수정하거나 추가해야 하는 자료구조

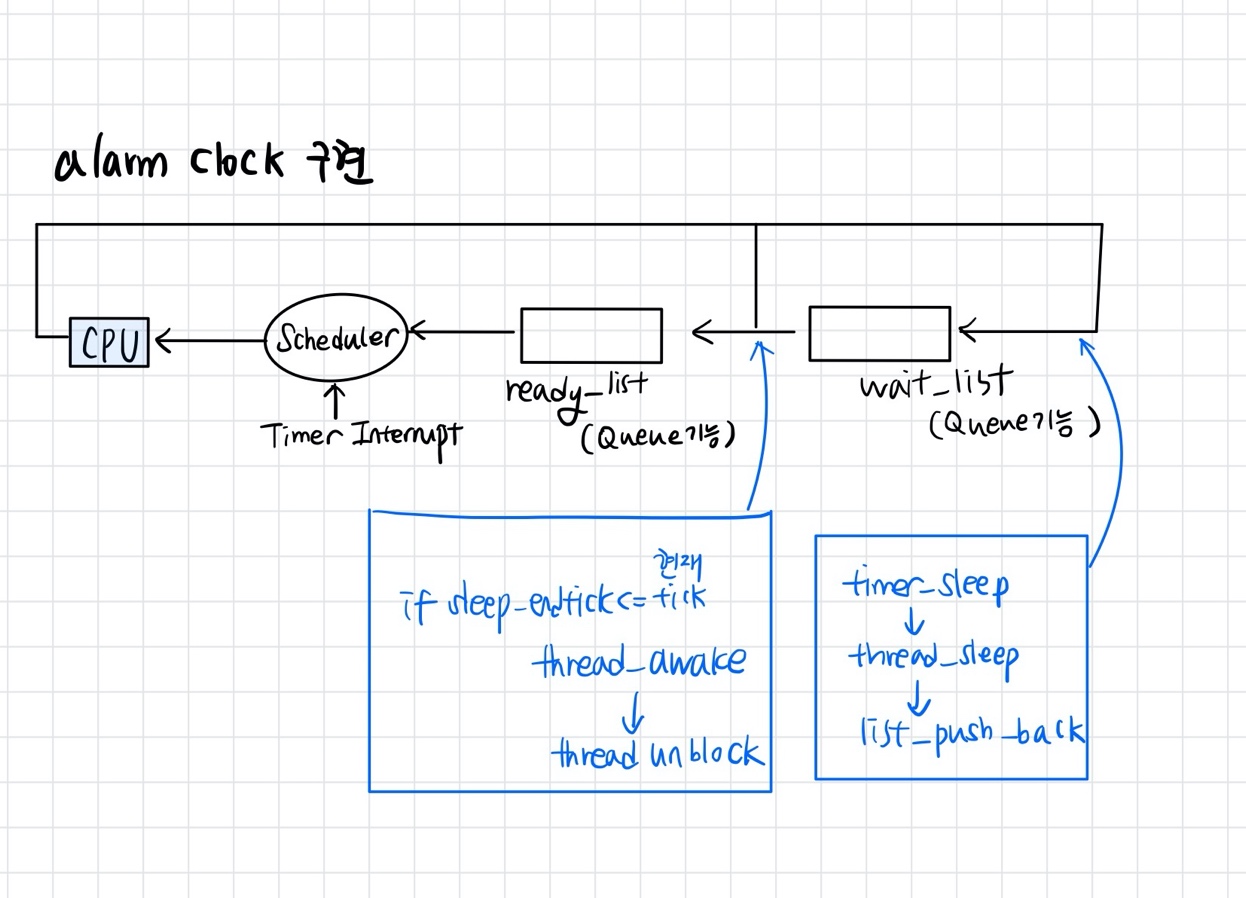
- ready\_list를 위한 Queue 구현

* + 수정하거나 추가해야 하는 함수

위에 이미 서술 완료함.

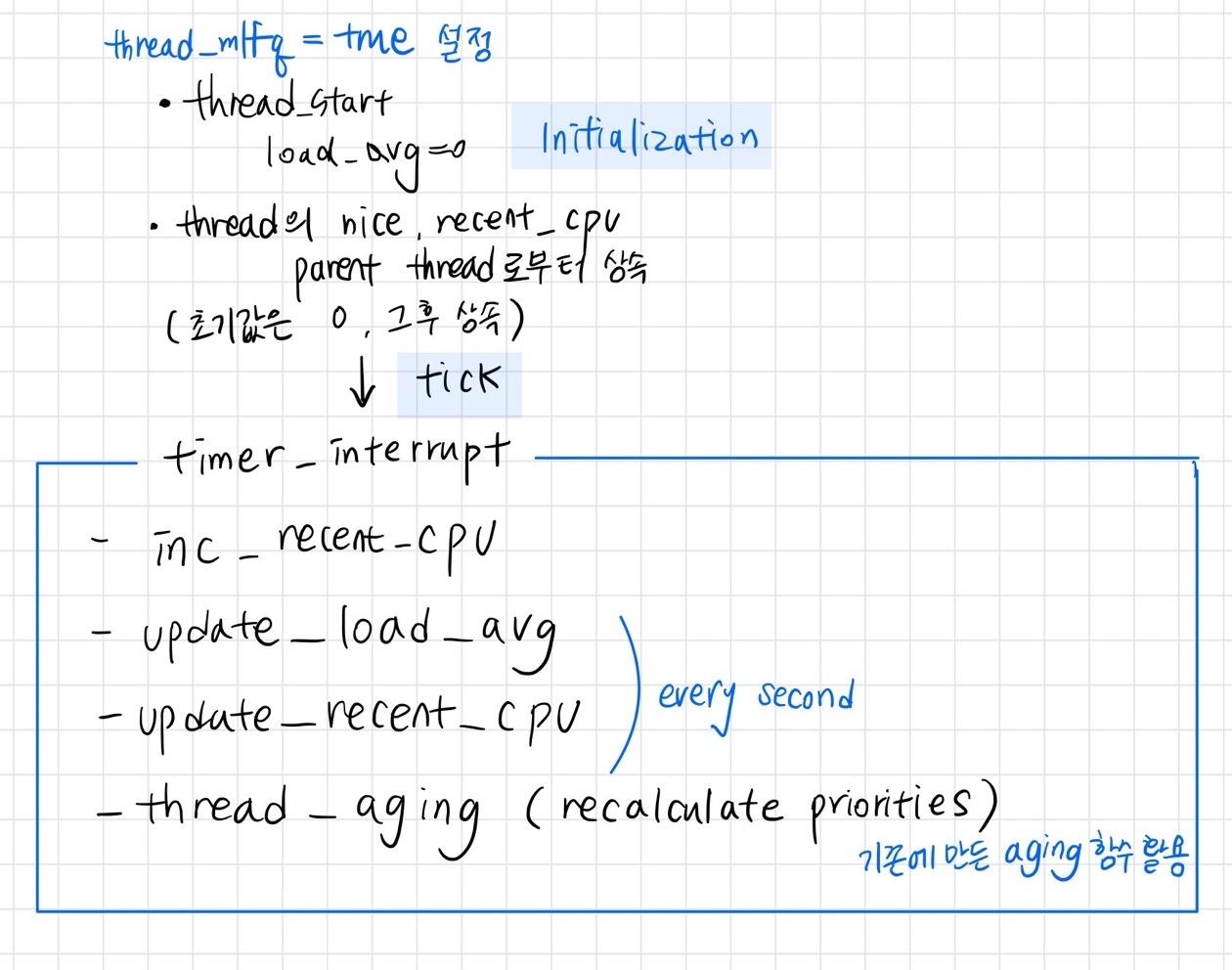
1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.**

****

* **Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.**
* **텍스트, 영수증, 서류이(가) 표시된 사진

  자동 생성된 설명**
* **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.**

****

* 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
* **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.**
* thread 구조체 내 sleepend\_tick 요소 추가
* int64\_t sleep\_endtick;

스레드가 자는 기간을 따로 추가함.

* timer\_sleep(int64\_t ticks)에 파라미터 추가하여 스레드가 자는 시간 지정
* void
* timer\_sleep (int64\_t ticks)
* {
* int64\_t start = timer\_ticks ();
* ASSERT (intr\_get\_level () == INTR\_ON);
* enum intr\_level old\_level = intr\_disable();
* //Busy Waiting
* //while (timer\_elapsed (start) < ticks)
* // thread\_yield ();
* thread\_sleep(start + ticks);
* intr\_set\_level(old\_level);
* }

기존의 busy waiting방식을 alarm clock 방식으로 대체하여 시스템 자원을 최대한 아낀다.

timer.c 내 timer\_sleep -> thread,c 내 thread\_sleep 호출

void thread\_sleep(int64\_t tick){

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

t->sleep\_endtick = tick;

//queue

list\_push\_back(&wait\_list,&t->waitelem);

thread\_block();

}

이 경우 자는 기간을 sleepend\_tick을 통하여 알아내고, 이 스레드를 wait\_list에 추가한다. 이 때 큐를 구현하기 위하여 list\_push\_back 함수를 사용한다.

* thread\_awake(int64\_t current\_tick)에 파라미터 추가하여 스레드가 깨어나도록 함
* /\* IMPORTANT
* Wake up all sleeping threads whose ticks has been expired.
* Remove them from the wait queue and push them into the ready queue \*/
* void thread\_awake(int64\_t current\_tick){
* struct list\_elem \*e;
* ASSERT(intr\_get\_level() == INTR\_OFF);
* //iterate
* for(e = list\_begin(&wait\_list);e != list\_end(&wait\_list);e = list\_next(e)){
* struct thread \*t = list\_entry(e,struct thread,waitelem);
* if(t->sleep\_endtick <= current\_tick){//expire
* //t -> sleep\_endtick = 0;
* list\_remove(&t->waitelem);
* //add it to the ready\_queue
* thread\_unblock(t);
* }
* }
* }

wait\_list에서 순회하며 스레드를 탐색하며 깨어날 시간이 지난 스레드를 깨워주고 이를 wait\_list에서 삭제한 뒤 ready\_list에 추가한다.

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,compare\_priority,NULL);

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

t->status = THREAD\_READY;

intr\_set\_level (old\_level);

}

이 때 ready\_list에 우선순위를 비교하여 list에 삽입한다. 이 때 기존의 list\_push\_back이라는 삽입 형식이아니라 비교함수를 compare\_priority라는 파라미터로 전달하여 list\_insert\_ordered 리스트 내장 함수를 활용한다. 이 함수는 아래에서 더 자세히 설명하도록 한다.

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

thread\_unblock(), thread\_yield() 수정하여 ready\_list 정렬 및 최우선순위 스레드 추출

thread\_unblock함수는 이미 위에 서술되었음.

/\* Yields the CPU. The current thread is not put to sleep and

may be scheduled again immediately at the scheduler's whim. \*/

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, compare\_priority,NULL);

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

}

cur->status = THREAD\_READY;

schedule ();

intr\_set\_level (old\_level);

}

thread\_yield함수에서도 리스트를 정렬하여 삽입한다.

* priority 비교해주는 함수 구현
* bool compare\_priority(const struct list\_elem \*e1, const struct list\_elem \*e2,
* void \*aux UNUSED)
* {
* struct thread \*t1 = list\_entry(e1, struct thread, elem);
* struct thread \*t2 = list\_entry(e2, struct thread, elem);
* ASSERT(t1 != NULL && t2 !=NULL);
* return t1 -> priority > t2 -> priority;
* }

list\_elem으로 연결된 스레드를 list\_entry로 접근하여 두 스레드의 priority를 직접적으로 비교한다.

그리고 priority 에 맞게 재스케줄링하기 위해 aging함수를 이에 적용한다.

void thread\_aging(){

struct thread \*t;

struct list\_elem \*e;

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

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

t->priority = ((PRI\_MAX \* FRACTION) - ((t->recent\_cpu)/4) - ((t->nice \* 2)\* FRACTION)) / FRACTION;

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

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

}

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

intr\_yield\_on\_return();

//thread\_yield();

}

}

모든 스레드의 우선순위를 재계산한다. 그리고 우선순위는 0~63이어야하므로 만약 이 범위를 벗어나면 최소값과 최대값으로 맞춰준다. 그리고 만약 현재 스레드의 우선순위가 더 작다면 CPU를 양보해야한다. 그리고 ready\_list는 이미 우선순위를 고려하여 정렬하여 insert하였으나 혹시 모를 경우를 대비하여 별도의 최우선순위 스레드의 우선순위를 리턴하는 함수를 추가 구현하였다.

int max\_priority(void){

int priority = -1;

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

struct thread \*t = list\_entry(list\_begin(&ready\_list),struct thread, elem);

priority = t->priority;

}

return priority;

}

* **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.**
* thread 구조체 내에 recent\_cpu 와 nice 요소 추가
* /\* Aging \*/
* int nice;
* int recent\_cpu;

그리고 thread\_init 함수에서 두 변수를 초기화한다.

initial\_thread->nice = 0;

initial\_thread->recent\_cpu = 0;

자식 스레드를 실행할 경우에는 init\_thread함수에서 부모 스레드의 값을 상속받는다.

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

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

* timer\_interrupt에서 시간의 흐름에 따라 priority 재계산 및 recent\_cpu 업데이트

timer\_interrupt의 진행흐름은 다음과 같다.

/\* Timer interrupt handler. \*/

static void

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

{

ticks++;

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

//increment recent\_cpu of current running thread

inc\_recent\_cpu();

thread\_awake(ticks);

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

//recent\_cpu is updated in every second

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

update\_load\_avg();

update\_recent\_cpu();

}

//Every 4 ticks, priorities are recalculated

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

thread\_aging();

}

}

thread\_tick(timer\_ticks());

}

mlfqs가 true로 설정되어있을 때, recent\_cpu를 1 증대한다.

이 경우 recent\_cpu는 real number로 별도의 연산 17.14 계산이 필요하다.

그리고 매초마다 load\_avg와 recent\_cpu를 업데이트하고 이에 맞게 매 4ticks마다 priority를 재계산한다. 이러한 매 타이머마다 recent\_cpu, load\_avg, 그리고 nice 입력을 위해 아래의 연산 함수들을 추가 구현하였다. 이 때 대부분 타이머의 영향으로 인한 혼란을 방지하고자 인터럽트를 중지해야 한다.

* thread\_set\_nice(), thread\_get\_nice(), thread\_get\_recent\_cpu(), update\_recent\_cpu() 외 변수 설정 함수 구현
* /\* Sets the current thread's nice value to NICE. \*/
* void
* thread\_set\_nice (int nice UNUSED)
* {
* intr\_disable();
* struct thread \*t = thread\_current();
* t->nice = nice;
* t-> priority = ((PRI\_MAX \* FRACTION) - (t->recent\_cpu / 4) - (nice \* 2 \* FRACTION)) / FRACTION;
* if(t->priority > PRI\_MAX) t->priority = PRI\_MAX;
* if(t->priority < PRI\_MIN) t->priority = PRI\_MIN;
* intr\_enable();
* }

이는 파라미터로 전달받은 nice를 해당 스레드에 설정하고 이에 맞게 priority를 조정한다.

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

int

thread\_get\_recent\_cpu (void)

{

intr\_disable();

int temp = nearest\_int((100 \* thread\_current()->recent\_cpu));

intr\_enable();

return temp;

}

recent\_cpu 관련 c파일을 보면 100을 나누어 값을 출력하는 형태를 띄고 있다. 이를 감안하여 위 함수에서는 100을 곱하여 리턴한다. 이는 int값 반환을 목적으로 하기 때문이다.

void inc\_recent\_cpu(void){

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

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

}

timer\_interrupt 함수에서 호출되는 함수로 recent\_cpu를 1씩 증대한다. 이 때 recent\_cpu가 실수의 경우라는 점을 감안하여 14자리를 올려서 처리해야한다. 간편한 코드처리를 위해 FRACTION 을 1<< 14 매크로 정의를 하였다.

void update\_load\_avg(){

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

if(thread\_current() != idle\_thread) ready\_threads += 1;

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

if(load\_avg < 0) load\_avg = 0;

}

void update\_recent\_cpu(){

struct thread \*t;

struct list\_elem \*e;

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

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

if(t != idle\_thread){

t->recent\_cpu = ((int64\_t)((int64\_t)2 \* load\_avg) \* FRACTION / (2\*load\_avg + (1\*FRACTION))) \* t->recent\_cpu

/ FRACTION + (t->nice) \* FRACTION;

}

}

차례로 매초마다 load\_avg 와 recent\_cpu를 업데이트한다. 이 경우에도 수의 분류를 고려하여 연산하고 FRACTION을 활용하였다.

* 이외 부동소수점 관련 연산 함수 추가(nearest\_int)
* int nearest\_int(int num){
* if(num >= 0)
* return (num + (FRACTION / 2)) / FRACTION;
* return (num - (FRACTION / 2)) / FRACTION;
* }

핀토스 매뉴얼대로 해당 실수의 가장 가까운 정수를 구하도록 하는 함수를 별도로 처리하여 반복적인 코드 작성을 방지하였다.

* /\* Returns 100 times the system load average. \*/
* int
* thread\_get\_load\_avg (void)
* {
* intr\_disable();
* int temp = nearest\_int((100 \* load\_avg));
* intr\_enable();
* return temp;
* }
* /\* Returns 100 times the current thread's recent\_cpu value. \*/
* int
* thread\_get\_recent\_cpu (void)
* {
* intr\_disable();
* int temp = nearest\_int((100 \* thread\_current()->recent\_cpu));
* intr\_enable();
* return temp;
* }
* 위의 nearest\_int 함수를 이용하여 load\_avg와 recent\_cpu를 간편하게 구할 수 있도록 한다.
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority-lifo는 같은 우선순위를 가지는 여러 개의 스레드를 만들어 라운드로빈 형식으로 동시적으로 처리되도록한다. 이는 mlfqs 기능은 false로 설정한 뒤 모든 스레드의 우선순위를 디폴트값인 31로 처리한다. 그리고 총 16개의 스레드를 형성하고, 각 id를 이에 맞게 출력한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* make check 수행 결과를 캡처하여 첨부
* 텍스트이(가) 표시된 사진

  자동 생성된 설명