**Pintos Project 3: Threads**

담당 교수 : 김영재 교수님

학번 / 이름 : 20210428 / 정현정

개발 기간 : 2024.11.16 ~ 2024.11.18

1. **개발 목표**

이번 프로젝트에서는 Alarm Clock, Priority Scheduling, Advanced Scheduler를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

1) Alarm Clock

현재 Alarm Clock은 비효율적인 방식으로 동작되고 있다. 현재 timer\_sleep 함수는 start로부터 ticks만큼의 시간이 지날 때까지 thread\_yield()를 실행시켜 스레드가 running 상태가 되면 바로 다시 ready 상태로 되돌리는 방식으로 waiting을 수행하고 있다. 이것을 해결하기 위해 ticks만큼의 시간이 지날 때까지 해당 스레드를 block 상태로 만들고, 새로운 리스트를 사용하여 그것들을 관리할 것이다.

2) Priority Scheduling

현재 Pintos의 스케줄링은 round robin 방식을 채택하고 있다. 즉, 우선순위에 관계 없이 thread\_yield() 또는 thread\_unblock()이 호출되면 실행 중인 스레드가 종료되고 교체된다. 이와 다른 방식으로 priority scheduling을 구현하기 위해 스레드마다 우선순위를 부여하고, 이를 기반으로 스케줄링 되도록 구현할 것이다. 또 Aging을 구현하여 starvation을 방지한다.

3) Advanced Scheduler

Multi Level Feedback Queue를 구현하여 performance를 향상시킨다. MLFQ를 구현하기 위해서는 우선순위를 정해진 공식에 따라 계산해주어야 하고, 이때 소수 연산이 필요하다. Pintos는 부동소수점을 지원하지 않기 때문에 우선순위를 계산할 때 고정소수점 방식을 채택한다.

* 1. **개발 내용**

1) Blocked 상태의 스레드를 어떻게 깨울 수 있는가

Timer interrupt가 발생했을 때, blocked 상태에 있는 스레드를 관리하는 리스트를 순회하면서 지정된 tick만큼 시간이 지났는지 확인한다. 만약 지났다면 thread\_unblock()을 호출하여 blocked 상태의 스레드를 깨우고 다시 ready list에 삽입한다.

2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우, priority scheduling에 따르면 어떻게 해야 하는가

Running thread는 즉시 yield를 수행하고 우선순위 기반의 스케줄링 정책에 의해 더 높은 우선순위를 가진 스레드가 실행되어야 한다.

3) Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술

계산해야할 요소들은 다음과 같다.

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

recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice

load\_avg = (59 / 60) \* load\_avg + (1/60) \* ready\_threads

여기서 nice는 각 스레드마다 정해져 있는 nice value를 의미한다.

recent\_cpu는 해당 스레드가 최근에 사용한 CPU 시간을 측정한 값을 의미한다. 최근에 CPU 사용량이 높았던 스레드는 우선순위를 낮춰주어야 한다.

load\_avg는 시스템 전체의 CPU 부하 상태를 측정하는 값이다. 최근 CPU 대기열에 있는 스레드 수를 지수 가중 이동 평균으로 나타낸다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11/16 : Alarm Clock, Priority Scheduling 구현

11/17 : Aging, Advanced Scheduler 구현

11/18 : 보고서 작성

* 1. **개발 방법**

1) Alarm Clock

해당 스레드를 깨워야할 시간을 저장할 필드가 스레드 구조체에 필요하다. 따라서 wakeup\_time 필드를 추가해야 한다. 또 timer\_sleep 함수를 수정하여 wakeup\_time 필드에 값을 설정하고, blocked 상태의 스레드들을 관리하기 위해 새로 선언한 blocked\_list에 삽입한다. 그리고 timer\_interrupt 함수를 수정하여 blocked\_list를 관리하고 필요 시 thread\_unblock을 호출한다.

2) Priority Scheduling

우선순위 기반의 스케줄링을 위해 ready queue에 삽입하는 방식 자체를 수정한다. list\_insert\_ordered()를 사용하여서 ready queue가 항상 우선순위 기준 내림차순을 유지하도록 한다. 이를 위해 thread\_unblock(), thread\_yield()를 수정한다. 또 sema\_down()에서도 waiter 리스트를 관리할 때, 우선순위 기준 내림차순으로 정렬되도록 기존의 list\_push\_back()을 list\_insert\_ordered()로 수정한다. sema\_up()에서는 sema\_up()으로 인해 ready list에 삽입된 스레드가 현재 스레드가 우선순위가 높은 경우에 yield가 발생할 수 있도록 우선순위를 확인하는 코드를 추가한다. 또 starvation을 막기 위해 thread\_tick()이 호출될 때마다 모든 스레드의 우선순위를 높이는 thread\_aging()을 구현한다.

3) Advanced Scheduler

MLFQ를 구현하기 위해서는 2)에서 사용하였던 thread\_aging() 함수를 앞에서 언급한 공식에 맞게 수정해야 한다. 또 고정소수점 연산을 위한 함수 또한 개발의 편의를 위해 구현한다. load\_avg, recent\_cpu, priority값들이 적절한 주기로 업데이트 될 수 있도록 thread\_tick 함수에 추가한다.

1. **연구 결과**
   1. **Flow Chart**

**텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명**

* 1. **제작 내용**

1. Alarm Clock

1) thread 구조체 수정

threads/thread.h

struct thread

  {

    /\* 생략 \*/

#ifdef USERPROG

    /\* Owned by userprog/process.c. \*/

    uint32\_t \*pagedir;                  /\* Page directory. \*/

    struct semaphore wait\_exit; //자식 프로세스 종료 대기

    struct semaphore wait\_free; //메모리 해제 대기

    struct semaphore wait\_load; //자식 프로세스 load 대기

    bool exec\_flag; //자식 프로세스가 정상적으로 exec 되었는지 표시하는 flag : true = 정상 실행

    int exit\_status; //종료 상태

    struct list child\_threads; //자식 프로세스 모아두는 링크드리스트

    struct list\_elem child\_elem; //내가 자식 프로세스일 때 부모 프로세스의 child\_threads에서의 포인터 정보

    struct file \*fd\_table[130]; //fd 테이블 (0, 1 + 128개)

    struct file \*execute\_file; //현재 실행 중인 file

#endif

    /\* Owned by thread.c. \*/

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

    int wakeup\_time; //깨워야 할 시간 = start + ticks

};

우선 스레드 구조체에 wakeup\_time 필드를 추가한다. 해당 필드에 스레드를 얼마나 blocked 상태로 둘지 그 시간을 저장한다. 추후에 설정한 시간만큼 시간이 지났음을 체크할 때 timer.c에 전역변수로 선언되어 있는 ticks 변수를 사용하기 위해 (blocked 상태로 전환된 시작 시간) + (block 상태로 두고자 하는 설정 시간)을 저장한다.

2) blocked\_list 선언 및 초기화

devices/timer.c

struct list blocked\_list;

void

timer\_init (void)

{

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

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

  list\_init(&blocked\_list);

}

blocked 상태로 바뀐 스레드를 관리할 리스트가 필요하다. 이를 timer.c에 전역변수로 추가한다. 타이머가 초기화되는 시점에서 해당 리스트를 초기화 하기 위해 timer\_init 함수에 list\_init(&blocked\_list)를 추가한다.

3) timer\_sleep 함수 수정

devices/timer.c

void

timer\_sleep (int64\_t ticks)

{

  int64\_t start = timer\_ticks ();

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

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

  enum intr\_level old\_level = intr\_disable(); //인터럽트 비활성화

  current\_thread->wakeup\_time = start + ticks;

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

  thread\_block();

  intr\_set\_level(old\_level); //인터럽트 활성화

}

timer\_sleep 함수는 스레드가 일정 시간 동안 CPU를 사용하지 않고 대기하도록 요청할 때 동작한다. 기존의 timer\_sleep 함수를 수정하여 스레드 구조체의 wakeup\_time 필드를 초기화하고, blocked\_list에 삽입한 후 thread\_block 함수를 사용하여 blocked 상태로 전환한다. 이때 wakeup\_time 필드의 값을 start(현재 tick) + ticks(blocked 상태에 두고자 하는 시간)로 초기화 한다.

여기서 사용하는 thread\_block 함수는 아래와 같다.

void

thread\_block (void)

{

  ASSERT (!intr\_context ());

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

  thread\_current ()->status = THREAD\_BLOCKED;

  schedule ();

}

현재 스레드의 상태를 blocked 상태로 바꾸고, schedule 함수를 호출하여 스케줄링을 진행한다. schedule 함수는 내부적으로 next\_thread\_to\_run()을 사용하여 현재 스레드를 blocked 상태로 만든 후 다음에 실행시킬 스레드를 찾는다. 이때 next\_thread\_to\_run()은 ready list의 첫 번째 스레드를 선택하므로, ready list의 첫 번째 스레드가 그 다음으로 실행된다.

4) timer\_interrupt 함수 수정

devices/timer.c

static void

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

{

  ticks++;

  struct list\_elem\* elem;

  for(elem=list\_begin(&blocked\_list); elem != list\_end(&blocked\_list);){

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

    if(tmp -> wakeup\_time <= ticks){

      //blocked time만큼 시간이 지나 리스트에서 제거해주어야 하는 경우

      struct list\_elem \*next = list\_next(elem);

      list\_remove(elem);

      thread\_unblock(tmp);

      elem = next;

    }

    else{

      elem = list\_next(elem);

    }

  }

  thread\_tick ();

}

이제 timer\_interrupt 함수를 수정하여 blocked\_list를 관리해주어야 한다. timer\_interrupt 함수가 매 타이머 인터럽트마다 blocked\_list를 점검하여 wakeup\_time이 지나간 스레드를 다시 unblock 상태로 만들도록 수정한다. list를 순회하며 만약 wakeup\_time이 지나 깨워야 하는 스레드인 경우에는 blocked\_list에서 제거하고 thread\_unblock 함수를 사용하여 다시 ready list에 삽입한다. 여기서 thread\_unblock 함수는 스레드를 다시 ready 상태로 바꾼 후 ready list에 삽입하는 역할을 수행한다. 만약 아직 시간이 지나지 않았다면 blocked\_list의 다음 원소로 넘어간다.

이후 thread\_tick 함수를 호출한다. thread\_tick 함수는 내부적으로 관련 통계를 업데이트 하고 TIME\_SLICE가 초과되었다면 intr\_yield\_on\_return()을 호출하여 yield를 수행한다. 이때 intr\_yield\_on\_return()은 인터럽트 핸들러가 동작 중인 상황에서 스케줄링을 예약하는 함수로 결과적으로 thread\_yield()와 동일한 동작을 한다.

2. Priority Scheduling

기존의 Pintos 스케줄링은 round robin을 채택하고 있었다. 이를 수정하여 우선순위 기반의 priority scheduling이 이루어지도록 해야 한다. 우선순위는 스레드 구조체 안에 priority 필드에 저장되어 있으며 0~63까지의 값을 갖는다. 숫자가 높을수록 더 높은 우선순위를 가짐을 의미한다. thread.h에 아래와 같은 매크로가 정의되어 있는 것을 확인할 수 있다. 또 thread\_start()를 보면 초기 스레드의 우선순위 값을 PRI\_MIN으로 설정하고 있는 것도 확인할 수 있다.

/\* Thread priorities. \*/

#define PRI\_MIN 0                       /\* Lowest priority. \*/

#define PRI\_DEFAULT 31                  /\* Default priority. \*/

#define PRI\_MAX 63                      /\* Highest priority. \*/

우선순위 기반의 스케줄링을 구현하기 위해서는 새로운 스레드가 만들어지는 경우, thread가 yield되는 경우, thread가 unblock되는 경우, 중간에 우선순위가 변경되는 경우, 세마포어를 사용하는 경우로 나누어 총 다섯 가지 경우를 고려해야 한다.

1) thread\_create 함수 수정

threads/thread.c

tid\_t

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

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

{

  /\* 생략 \*/

  thread\_unblock (t);

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

    thread\_yield();

  }

  return tid;

}

새롭게 스레드가 생성되는 경우, 현재 실행되고 있는 스레드와 새로운 스레드의 우선순위 비교가 필요하다. 만약 새로운 스레드의 우선순위가 현재 실행 중인 스레드보다 높다면 즉시 thread\_yield 함수를 호출하여 새로운 스레드를 실행시켜야 한다.

2) thread\_yield 함수 수정

threads/thread.c

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);

  }

  cur->status = THREAD\_READY;

  schedule ();

  intr\_set\_level (old\_level);

}

기존의 thread\_yield 함수를 수정하여 우선순위 기준 내림차순으로 ready list에 삽입되도록 한다. 이를 통해 추후에 schedule()에서 다음에 실행시킬 스레드, 즉 ready list의 front에 해당하는 스레드가 ready list에서 우선순위가 가장 높은 스레드가 되도록 할 수 있다. 이때 list\_insert\_ordered()를 사용하고 새롭게 비교함수를 선언한다. 비교함수인 compare\_priority()는 다음과 같다.

//우선순위 기준 내림차순 정렬을 위한 비교 함수

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

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

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

  return (thread\_a->priority) > (thread\_b->priority);

}

3) thread\_unblock 함수 수정

threads/thread.c

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);

  t->status = THREAD\_READY;

  intr\_set\_level (old\_level);

}

thread\_unblock 함수도 ready list에 기존에 blocked\_list에서 관리되던 blocked 상태의 스레드를 삽입하고 blocked 상태였던 스레드를 깨우는 역할을 하기 때문에, list\_insert\_ordered()를 사용하여 ready list가 정렬 상태를 유지하도록 한다.

4) thread\_set\_priority 함수 수정

threads/thread.c

void

thread\_set\_priority (int new\_priority)

{

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

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

  //새로운 우선순위가 기존의 우선순위보다 낮다면 rescheduling

  if(old\_priority > new\_priority){

    thread\_yield();

  }

}

thread\_set\_priority 함수는 현재 실행 중인 스레드의 우선순위를 바꾸는 역할을 한다. priority scheduling은 preemptive scheduling이므로, 새로 설정된 우선순위가 기존의 우선순위보다 낮다면 thread\_yield()를 호출하여 우선순위가 더 높은 스레드가 실행되도록 구현했다.

5) sema\_down, sema\_up 함수 수정

threads/synch.c

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);

}

Pintos 매뉴얼에 따르면 priority scheduling를 구현할 때 세마포어의 waiters 리스트를 우선순위에 따라 관리해야 한다. 따라서 sema\_down 함수에서 waiters 리스트에 삽입하는 부분을 우선순위에 따라 삽입하도록 수정하였다.

void

sema\_up (struct semaphore \*sema)

{

  enum intr\_level old\_level;

  ASSERT (sema != NULL);

  old\_level = intr\_disable ();

  if (!list\_empty (&sema->waiters)){

    list\_sort(&sema->waiters, compare\_priority, NULL);

    thread\_unblock (list\_entry (list\_pop\_front (&sema->waiters), struct thread, elem));

  }

  sema->value++;

  //sema\_up으로 ready\_list에 삽입된 스레드가 현재 스레드보다 우선순위가 높은 경우

  check\_priority();

  intr\_set\_level (old\_level);

}

sema\_up 함수도 마찬가지이다. 우선 스레드가 waiters 리스트에 있는 동안 우선순위가 바뀌었을 수 있으므로 list\_sort()로 정렬을 해준다. thread\_unblock()을 수행한 다음, sema\_up()으로 인해 ready 상태가 된 스레드가 현재 스레드보다 우선순위가 높다면 CPU를 선점할 수 있도록 해당 로직을 갖고 있는 check\_priority 함수를 실행시킨다. check\_priority 함수는 다음과 같다.

//ready list의 맨 앞과 현재 스레드의 우선순위 비교

void check\_priority(void){

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

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

  struct thread\* front\_thread = list\_entry(ready\_list\_front, struct thread, elem);

  if(front\_thread->priority > thread\_get\_priority()){

    if (intr\_context()) {

      intr\_yield\_on\_return(); // 인터럽트 컨텍스트에서 스케줄링 예약

    }

    else {

      thread\_yield(); // 일반 컨텍스트에서 즉시 스케줄링

    }

  }

}

check\_priority 함수는 ready list의 맨 앞 스레드 즉, ready list에서 우선순위가 가장 높은 스레드와 현재 실행 중인 스레드의 우선순위를 비교하고, 우선순위가 더 높은 스레드를 실행시킨다. 인터럽트 핸들러 동작 도중 체크가 이루어질 수도 있으므로, 이에 대비하여 intr\_context()를 기준으로 실행시키는 함수를 달리 하였다.

6) Aging

지금까지 구현된 코드에 따르면 우선순위 스케줄링은 이루어질 수 있지만 낮은 우선순위를 가진 스레드가 장시간 CPU time을 할당받지 못하는 starvation이 발생할 수 있다. 이를 해결하기 위해 aging을 구현해야 한다. aging을 통해 스레드들의 우선순위가 시간에 비례하게 높아져야 한다.

threads/thread.c

void thread\_aging(void){

  struct list\_elem\* elem;

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

    struct thread\* tmp = list\_entry(elem, struct thread, allelem);

    tmp->priority++;

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

    else if(tmp->priority > PRI\_MAX) tmp->priority = PRI\_MAX;

  }

}

thread\_aging 함수를 새로 선언하여 aging을 구현하였다. thread\_aging 함수는 모든 스레드의 우선순위를 1씩 높여주는 간단한 함수이다. 이 함수를 thread\_tick()에서 호출하도록 수정하여 매 틱마다 스레드들의 우선순위를 높이도록 하였다. 수정된 thread\_tick 함수는 다음과 같다.

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 ();

  /\* Project #3 \*/

  #ifndef USERPROG

  if(thread\_prior\_aging==true){

    thread\_aging();

  }

  #endif

}

3. Advanced Scheduler

Advanced Scheduler는 MLFQ로 우선순위를 기준으로 스케줄링을 하며, 만약 최상위 우선순위를 가진 스레드가 여러 개라면 round robin 방식으로 스케줄링을 한다. MLFQ이므로 기본적으로 aging과 관련된 부분이 존재한다. 즉, 이전에는 단순히 우선순위를 1씩 높였다면 MLFQ에서는 우선순위를 특정 계산식에 맞게 갱신해주어야 한다.

1) priority 계산에 필요한 요소

priority 계산에 필요한 계산식들은 다음과 같다.

* priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)
* recent\_cpu = (2 \* load\_avg) / (2 \* load\_avg + 1) \* recent\_cpu + nice
* load\_avg = (59 / 60) \* load\_avg + (1/60) \* ready\_threads

priority는 항상 PRI\_MIN(= 0) ~ PRI\_MAX(= 63) 사이의 값으로 조정되며 스레드를 초기화할 때와 매 4 tick마다 모든 스레드에 대해 다시 계산된다. 여기서 nice는 스레드마다 지정된 nice value를 의미한다. nice value가 클수록 우선순위가 낮다. 스레드가 처음 생성될 때 nice값은 0으로 초기화 되며, 부모가 있는 경우 부모의 nice value를 상속받는다.

recent\_cpu는 최근 CPU 사용량을 의미한다. 즉, 특정 스레드가 최근에 사용한 CPU 시간을 말한다. 최근에 CPU 사용량이 높은 스레드의 우선순위를 낮추기 위해 priority 계산에 사용된다. 1초마다 모든 스레드에 대해서 recent\_cpu값이 다시 계산되어야 하며, 타이머 인터럽트가 발생할 때마다 실행 중인 스레드에 대해 recent\_cpu값이 1씩 증가한다. 스레드 생성 시 recent\_cpu는 0으로 초기화 된다. 부모 스레드가 있는 경우, 부모의 값을 상속받는다.

load\_avg는 시스템 전체의 CPU 부하 상태를 측정하는 값이다. 이때 read\_threads라는 값이 필요한데, 이는 현재 ready 상태인 스레드의 수와 현재 실행 중인 스레드의 수를 더한 값을 의미한다. ready\_threads에 idle\_thread는 포함되지 않는다. load\_avg 값은 시스템 부팅 시 0으로 초기화 되며, 1초마다 계산되어야 한다.

정리하자면, 1초마다 모든 스레드의 recent\_cpu값과 전역변수로 선언된 load\_avg값을 업데이트 해주어야 한다. 또 priority는 4 tick마다 재계산 되고, running thread들의 recent\_cpu값은 1 tick마다 1씩 증가시켜야 한다. 주의해야할 점은 해당 요소들의 숫자 타입이 다르다는 점이다. load\_avg와 recent\_cpu값은 소수이고, ready\_threads, nice, priority는 정수이다. Pintos는 부동소수점 연산을 제공하지 않기 때문에 고정소수점으로 연산을 해주어야 한다. 즉, 같은 int type의 변수라고 할지라도 실제로 할당된 값을 고정소수점 연산 방식으로 다루어야 한다.

2) 고정소수점 연산 구현

threads/thread.c

//정수를 고정소수점으로 변환

int int\_to\_fixed(int x){

  return x \* (1 << 14);

}

//고정소수점을 정수로 변환 : 버림

int fixed\_to\_int(int x){

  return x / (1 << 14);

}

//고정소수점을 정수로 변환 : 반올림

int fixed\_to\_int\_round(int x){

  if(x >= 0){

    return (x + (1 << 13)) / (1 << 14);

  }

  else{

    return (x - (1 << 13)) / (1 << 14);

  }

}

//고정소수점 곱셈

int mult\_fixed(int x, int y){

  return ((int64\_t)x \* y) / (1 << 14);

}

//고정소수점 나눗셈

int div\_fixed(int x, int y){

  return ((int64\_t)x \* (1 << 14)) / y;

}

부동소수점 연산의 일관성과 편의성을 위해 부동소수점 연산을 수행하는 함수들을 새롭게 작성하였다.

3) thread 구조체 수정 및 계산에 필요한 요소 선언과 초기화

threads/thread.h

struct thread

  {

    /\* 생략 \*/

    /\* Owned by thread.c. \*/

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

    int wakeup\_time; //깨워야 할 시간 = start + ticks

    int nice;

    int recent\_cpu;

  };

먼저 thread 구조체를 수정한다. 스레드마다 nice값과 recent\_cpu값을 가져야 하므로 이를 저장할 필드들을 새롭게 추가한다.

threads/thread.c

int load\_avg = 0;

void

thread\_init (void)

{

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

  lock\_init (&tid\_lock);

  list\_init (&ready\_list);

  list\_init (&all\_list);

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

  initial\_thread = running\_thread ();

  init\_thread (initial\_thread, "main", PRI\_DEFAULT);

  initial\_thread->status = THREAD\_RUNNING;

  initial\_thread->tid = allocate\_tid ();

  initial\_thread->nice = 0;

  initial\_thread->recent\_cpu = 0;

}

load\_avg는 전역변수로 선언하고 0으로 초기화 한다. 또 thread\_init()에서 각 스레드의 nice값과 recent\_cpu값을 0으로 초기화 한다. nice값과 recent\_cpu값은 부모가 있는 스레드의 경우 부모의 값들을 상속받아야 하므로 thread\_create() 안에서 호출되는 init\_thread 함수를 수정한다. 수정된 init\_thread 함수는 다음과 같다.

threads/thread.c

static void

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

{

  enum intr\_level old\_level;

  ASSERT (t != NULL);

  ASSERT (PRI\_MIN <= priority && priority <= PRI\_MAX);

  ASSERT (name != NULL);

  memset (t, 0, sizeof \*t);

  t->status = THREAD\_BLOCKED;

  strlcpy (t->name, name, sizeof t->name);

  t->stack = (uint8\_t \*) t + PGSIZE;

  t->priority = priority;

  t->magic = THREAD\_MAGIC;

  t->nice = running\_thread()-> nice;

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

  /\* 생략 \*/

}

부모 스레드가 곧 현재 실행 중인 스레드이므로, 현재 실행 중인 스레드의 nice값과 recent\_cpu값을 새로운 스레드에 할당한다.

4) load\_avg

threads/thread.c

//ready\_threads 개수 구하는 함수

int ready\_threads\_cnt(void){

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

    return list\_size(&ready\_list) + 1;

  }

  else{

    return list\_size(&ready\_list);

  }

}

//load\_avg 업데이트

void update\_load\_avg(void){

  int ready\_threads = ready\_threads\_cnt();

  int fixed\_59\_60 = div\_fixed(int\_to\_fixed(59), int\_to\_fixed(60));

  int fixed\_1\_60 = div\_fixed(int\_to\_fixed(1), int\_to\_fixed(60));

  load\_avg = mult\_fixed(fixed\_59\_60, load\_avg) + mult\_fixed(fixed\_1\_60, int\_to\_fixed(ready\_threads));

  if(load\_avg < 0) load\_avg = 0;

}

앞에서 설명한 load\_avg 계산식을 구현했다. 이때 load\_avg는 고정소수점 연산이 필요하므로 연산에 사용되는 모든 숫자를 고정소수점 방식으로 바꿔준 후, 연산을 수행한다.

5) recent\_cpu

recent\_cpu와 관련해서는 크게 두 가지를 구현해야 한다. 먼저, 모든 스레드의 recent\_cpu값을 앞에서 설명한 계산식에 맞게 업데이트 해주어야 한다. 이를 구현한 코드는 다음과 같다. 여기서도 고정소수점 연산을 사용해야 한다.

threads/thread.c

//모든 스레드의 recent\_cpu값 업데이트

void update\_recent\_cpu(void){

  struct list\_elem\* elem;

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

    struct thread\* tmp = list\_entry(elem, struct thread, allelem);

    //idle thread인 경우

    if(tmp == idle\_thread) continue;

    //recent\_cpu값 업데이트

    int fixed\_2 = int\_to\_fixed(2);

    int fixed\_1 = int\_to\_fixed(1);

    int fixed\_nice = int\_to\_fixed(tmp->nice);

    int term1 = mult\_fixed(fixed\_2, load\_avg);

    int term2 = mult\_fixed(fixed\_2, load\_avg) + fixed\_1;

    tmp->recent\_cpu = mult\_fixed(div\_fixed(term1, term2), tmp->recent\_cpu) + fixed\_nice;

  }

}

두 번째로, running thread에 대해서만 recent\_cpu값을 1 증가시켜야 한다. 이를 구현한 코드는 다음과 같다.

threads/thread.c

//running thread의 recent\_cpu값 += 1

void increment\_recent\_cpu(void){

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

  if(current\_thread == idle\_thread) return;

  else{

    int fixed\_1 = int\_to\_fixed(1);

    current\_thread->recent\_cpu += fixed\_1;

  }

}

6) 우선순위 업데이트

threads/thread.c

//모든 스레드의 우선순위 업데이트

//4 tick마다 수행되어야 함 -> timer\_interrupt

void thread\_aging2(void){

  struct list\_elem\* elem;

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

    struct thread\* tmp = list\_entry(elem, struct thread, allelem);

    if(tmp == idle\_thread) continue;

    int fixed\_PRI\_MAX = int\_to\_fixed(PRI\_MAX);

    int fixed\_4 = int\_to\_fixed(4);

    int fixed\_nice = int\_to\_fixed(tmp->nice);

    int fixed\_2 = int\_to\_fixed(2);

    int tmp2 = fixed\_PRI\_MAX - div\_fixed(tmp->recent\_cpu, fixed\_4) - mult\_fixed(fixed\_nice, fixed\_2);

    tmp->priority = fixed\_to\_int(tmp2);

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

    else if(tmp->priority > PRI\_MAX) tmp->priority = PRI\_MAX;

  }

  check\_priority();

}

마지막으로 모든 스레드의 priority를 업데이트 해준다. 계산식에 맞게 고정소수점 연산을 수행한 후, 마지막에 고정소수점 연산 결과를 정수로 바꾸어야 한다. priority는 정수이기 때문이다. 우선순위를 업데이트 한 후, CPU 선점이 발생해야 할 수 있으므로 check\_priority()를 호출한다. 타이머 인터럽트가 발생할 때마다 thread\_tick()이 호출되는데, 여기에서 4 tick마다 thread\_aging2()가 실행되므로 check\_priority()의 intr\_yield\_on\_return()이 호출될 것이다.

threads/thread.c

void

thread\_tick (void)

{

  /\* 생략 \*/

  /\* Project #3 \*/

  #ifndef USERPROG

  if(thread\_prior\_aging==true){

    thread\_aging();

  }

  #endif

  if(thread\_mlfqs == true){

    //업데이트

    increment\_recent\_cpu();

    int64\_t ticks = timer\_ticks();

    if(ticks % TIMER\_FREQ == 0){

      //1초마다 업데이트

      update\_load\_avg();

      update\_recent\_cpu();

    }

    if(ticks % 4 == 0){

      thread\_aging2();

    }

  }

}

thread\_mlfqs가 true인 경우에만 계산이 되도록 thread\_tick 함수를 수정한다. thread\_tick은 매 tick마다 timer\_interrupt 함수에서 호출하므로 1 tick마다 increment\_recent\_cpu()를 호출한다. 또 1초마다 load\_avg와 모든 스레드의 recent\_cpu값을 갱신하고, 4 tick마다 우선순위를 다시 계산한다.

7) 기타 구현 함수

명세서에 나와있는 기타 함수들을 구현하였다.

threads/thread.c

void

thread\_set\_nice (int nice UNUSED)

{

  thread\_current()->nice = nice;

  //우선순위 재계산 과정 필요!

  thread\_aging2();

}

int

thread\_get\_nice (void)

{

  return thread\_current()->nice;

}

int

thread\_get\_load\_avg (void)

{

  return fixed\_to\_int\_round(load\_avg \* 100);

}

int

thread\_get\_recent\_cpu (void)

{

  return fixed\_to\_int\_round(thread\_current()->recent\_cpu \* 100);

}

thread\_set\_nice()는 현재 스레드의 nice값을 업데이트 하한다. nice값이 바뀜에 따라 우선순위도 다시 계산되어야 한다.

thread\_get\_nice(), thread\_get\_recent\_cpu(), thread\_get\_load\_avg() 함수들은 단순한 getter 함수들이다. 이때, recent\_cpu값과 load\_avg값은 명세서에 따라 100배로 반환하고 소수점을 포함하지 않도록 반올림을 하였다.

* 1. **시험 및 평가 내용**

make check 수행 결과

텍스트, 스크린샷, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

priority-lifo 테스트 케이스는 높은 우선순위를 가진 스레드가 낮은 우선순위를 가진 스레드보다 먼저 실행되는지를 체크하는 테스트 케이스이다. 출력 결과에서 볼 수 있듯이 잘 실행되고 있음을 확인할 수 있다.