**Pintos Project 3: Threads**

담당 교수 :

조 / 조원 :

개발 기간 :

1. **개발 목표**

Alarm Clock 구현, Priority scheduling 구현.

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

핀토스의 디폴트 스케줄러는 busy-waiting으로 되어 있다. 이렇게 비효율적으로 구현되어 있는 핀토스의 스케줄러를 tick단위로 계산되는 시간을 활용해 일정 시간마다 thread가 동작하지 않으면 sleep 시키는 스케줄링으로 구현한다.

1. Priority Scheduling

핀토스의 디폴트는 Round-Robin 방식으로 스케줄링이 구현되어 있다. 이는 실행중인 thread가 ready\_list로 들어갈 때 항상 ready\_list의 맨 뒤로 fifo 방식으로 push되는 방식이다. 이는 각 thread의 우선순위를 따지지 못하고 항상 번갈아 실행해야하는 단점이 있다. 이 단점을 보완하기 위해 prirority scheduling을 구현하여 preemptive scheduling을 할 수 있도록 한다. 추가적으로 기아 문제를 해결하기 위해 aging도 구현한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Tick단위로 계산되는 시간을 활용해 timer\_tick 보다 thread의 시간이 흐르면 thread\_unblock() 함수를 호출하여 thread를 ready\_list에 추가하는 thread\_wakeup 함수를 구현한다. Blocked 된 상태의 thread들을 리스트 형태로 저장하고 있는 sleep\_list를 추가적으로 생성하고, 이를 주기적으로 순환하면서 thread\_wakeup을 호출한다.

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

Ready\_list에 running thread보다 높은 priority를 가진 thread가 들어온 경우, ready\_list의 가장 앞에 들어올 확률이 높다. 또는 이미 priority가 더 높은 thread가 맨 앞에 존재할 수도 있다. 위의 기능을 구현하기 위해서, push\_back이 아니라, push할 thread의 우선순위에 따라 정렬된 삽입을 해주어야 한다. 결론적으로 맨 앞의 thread의 우선순위가 현재 실행중인 thread의 우선순위보다 크면 현재 thread는 cpu를 yield함으로써 자신을 새로 ready\_list에 삽입하고 schedule을 실행하게 된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성
  + - 11/10 ~ 11/10: Alarm Clock
    - 11/12 ~ 11/15: Priority Scheduling
    - 11/24 ~ 11/24: 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
* Alarm Clock

Thread.c 파일에 Struct list sleep\_list 추가 선언하여 sleep중인 스레드의 목록을 관리함.

Timer\_interrupt 애서 thread\_wakeup(ticks);를 추가하여 sleep list에 있는 thread들 중 wakeup tick 보다 tick이 커진 thread들을 깨운다.

Thread\_sleep 함수를 추가해서 Timer\_sleep() 에서 thread\_sleep을 호출하여 sleep list에 push하고, block 하는 코드를 구현한다.

Thread\_init에서 새로 추가된 list인 ready\_list와 sleep\_list를 초기화한다.

* Priority Scheduling

Priority를 설정할 때, 새로 생성된 스레드의 우선순위가 현재 실행 중인 스레드보다 높으면 thread\_yield()를 호출해 선점하도록 한다.

[threads/thread.c]에서 우선순위를 비교하는 함수 cmp\_priority()를 구현하여 list\_insert\_ordered()에서 정렬 기준 함수로 활용한다.

기존에 사용된 list\_push\_back()을 list\_insert\_ordered()로 변경해, 스레드가 yield되거나 unblock될 때 정렬된 상태로 추가되도록 한다.

[aging]

[threads/thread.c]에 bool thread\_prior\_aging 변수를 추가해 aging 기능을 활성화할 수 있도록 한다.

[threads/init.c]에 strcmp로 "-aging" 옵션 유무를 확인하는 조건문을 추가하여 aging 기능을 처리할 수 있게 한다.

[threads/thread.c]에 thread\_aging 함수를 추가해 모든 thread의 priority가 일정 시간마다 상승할 수 있도록 구현한다.

1. **연구 결과**
   1. **Flow Chart**

* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성

**-Alarm Clock**

도표, 텍스트, 라인, 기술 도면이(가) 표시된 사진

자동 생성된 설명

* **Priority Scheduling**

도표, 기술 도면, 평면도, 라인이(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

* struct *list* sleep\_list;

thread.c 에 sleep\_list를 선언하여 sleep 상태로 존재하고 있는 thread들의 목록을 저장할 수 있도록 한다. 이를 thread\_init에서 초기화한다.

void

thread\_init (void)

{

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

  lock\_init (&tid\_lock);

  list\_init (&ready\_list);

  list\_init (&all\_list);

  list\_init (&sleep\_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 ();

}

위 언급하였던 ready\_list와 sleep\_list를 thread\_init 함수에서 추가적으로 초기화해주는 구현 모습이다.

void

timer\_sleep (int64\_t ticks) //project 3

{

  int64\_t start = timer\_ticks ();

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

  // while (timer\_elapsed (start) < ticks)

  //   thread\_yield ();

  if(timer\_elapsed (start) < ticks) thread\_sleep(start + ticks);

}

**기존 방식이었던 busy\_waiting 방식의 yield를 반복문 대신 if를 활용하여 thread\_sleep을 호출하면, 아래 추가 구현된 함수가 실행되며, thread가 sleep하게 된다. 이 과정을 통해 효율성을 더 높일 수 있다.**

static void

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

{

  ticks++;

  thread\_tick ();

  thread\_wakeup(ticks);

}

**기존 timer\_interrupt에서 thread\_wakeup()함수를 추가하였다. 인터럽트는 Tick을 증가시키면서, thread\_wakeup함수를 호출하게 된다. 이 함수는 아래 추가 설명하였다.**

void thread\_sleep(int64\_t ticks){//project 3

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

  if(cur != idle\_thread){

    //disable interrupt

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

    // local tick to wake up

    cur->wakeup\_tick = ticks; // ticks = timer\_ticks + ticks

    // sleep list push

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

    thread\_block();

    intr\_set\_level(old\_level);

  }

}

/\*

  project 3

\*/

void thread\_wakeup(int64\_t ticks){

  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->wakeup\_tick <= ticks) {

          e = list\_remove(e);

          thread\_unblock(t);

      } else {

          e = list\_next(e);

      }

  }

}

**Thread\_sleep함수는 현재 thread가 다음에 wakeup될 수 있게 local tick을 저장해주고, sleep list에 push, 마지막으로 thread를 block 함으로써 sleep시킨다.**

**Intr\_set\_level과 intr\_disable() 함수를 사용해 스레드의 리스트를 관리할 때 interrupt를 차단하여 그에 따른 버그 발생을 차단하였다.**

1. **Priority Scheduling**

void

thread\_set\_priority (int new\_priority)

{

  int64\_t old\_prior = thread\_current()->priority;

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

  if (old\_prior > new\_priority) thread\_yield();

}

현재 running thread의 프라이어리티를 바꾸는 함수이다. Preemptive를 위해서 priority가 낮아지면 다른 thread\_yield 한다.

bool cmp\_priority(const struct list\_elem \*elem\_a, const struct list\_elem \*elem\_b, void \*aux UNUSED) {

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

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

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

}

아래 list\_insert\_ordered에 들어가야 하는 비교 함수이다. 우선순위가 큰 것부터 정렬할 수 있도록 한다.

thread\_yield와 thread\_unblock 함수에서 thread를 ready\_list에 추가할 때 priority의 순서를 고려하여 삽입할 수 있도록 삽입 함수를 아래와 같이 변경하였다.

list\_insert\_ordered(& ready\_list, & t-> elem, cmp\_priority, NULL);

다음은 aging을 위한 함수들이다.

[Aging]

bool thread\_prior\_aging;

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

        thread\_prior\_aging = true;

실행 옵션에 aging이 있으면 aging 플래그를 활성화한다.

void thread\_aging(){ //priority aging

  struct thread \*t = NULL ;

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

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

    t->priority++;//tick마다 1씩 priority aging

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

  }

}

일정 tick마다 실행되는 thread\_aging 함수이다.

List에 존재하는 모든 thread에 대해서 우선순위를 상승시킨다. 우선순위가 이미 정해진 MAX\_PRIORITY를 못 넘게 보정하였다.

void thread\_priority\_yield ()//priority sema, sema up에 사용.

{

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

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

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

    thread\_yield();//aging에 따라서 priority 밀려나면 yield 해야함.

  }

}

Thread\_yield와 다르게 주기적으로 현재 running thread보다 높은 우선순위를 가지는 thread가 있는지 확인하고, 있다면 thread\_yield 해준다. 이 함수는 아래 sema\_up 에서만 사용된다.

void

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

{

  enum intr\_level old\_level;

  ASSERT (sema != NULL);

  old\_level = intr\_disable ();

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

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

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

                                struct thread, elem));

  }

  sema->value++;

  thread\_priority\_yield();

  intr\_set\_level (old\_level);

}

위 함수에서 thread\_priority\_yield를 사용하여 priority\_sema를 해결하였다.

세마포어를 상승시킬 때 위 thread\_priority\_yield를 호출해서 현재 thread보다 높은 priority에게 cpu를 양도한다.

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

priority\_lifo.c는 같은 우선순위를 가진 여러 스레드를 만들어서 Round-Robin 방식으로 스레드의 스케줄링이 진행되는지 확인할 수 있도록 하는 코드이다.

메인 스레드 우선순위는 기본값(PRI\_DEFAULT = 15)이고, 우선순위가 15인 스레드들부터 실행된다.

아래 사진을 보면 Ready 리스트에 정렬된 순서대로 스레드가 실행되는 것을 확인할 수 있다.

* make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 흑백, 모노크롬이(가) 표시된 사진

자동 생성된 설명

추가 구현을 제외한 모든 test에서 pass를 확인하였다.