**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171628 김현우

개발 기간 : 10/28 ~ 11/6

1. **개발 목표**

thread 관련한 함수를 구현한다. Alarm clock을 구현해서 thread를 효율적으로 사용할 수 있게 하고, priority를 계산해서 스케쥴을 조절하는 것을 구현

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock

이전의 루프를 지속적으로 돌아서 시간이 다 될 때까지 기다렸다가 대기상태인 스레드를 깨우는 방식과 달리 시간이 되면 깨워주는 방식을 선택해서 효율을 높혀준다.

* 1. Priority Scheduling

이전에는 프로세스의 우선순위가 존재하지 않아서 들어온 순서대로 fifo로 실행 되었지만 구현을 하면 중간에 바뀌는 우선순위에 따라 프로세스 실행 순서가 달라지게 된다,

* 1. Advanced Scheduler (추가구현을 한 경우)

소수점 계산을 통해 priority, nice, cpu\_recent를 계산 할 수 있게 된다. 핀토스는 소수점 계산을 지원하지 않으므로 직접 int를 고정 소수점으로 활용해서 소수처럼 계산을 해준다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Block 상태로 들어간 thread list를 저장하기 위한 sleep\_list를 선언하고 저장할 때 wakeup 할 시간을 저장한다. 이후에 한 tick이 지날 때 마다 sleep\_list에 깨울 thread가 있는지 확인을 한다. 만약 깨울 thread가 존재한다면 sleep\_list에 있는 원소를 ready\_list로 옮기고 thread의 상태를 blocked 에서 ready로 바꿔준다.

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

기본적으로 ready\_list는 우선순위가 높은 순서대로 정렬이 된다, 그러므로 가장 높은 우선순위를 가진 스레드가 제일 앞에 가게 되는데, 이제 ready\_list의 맨앞의 원소와 현재 실행중인 스레드의 우선순위를 비교한다. Ready\_list에 running thread보다 높은 우선순위를 가진 thrread가 들어오면 현재 스레드에서 thread\_yield를 발생시켜서 현재 스레드를 중단시킨다. 이후에 현재 실행중인 thread는 ready\_list로 우선순위에 따라 들어가게 되고 가장 우선순위가 높은 ready\_list에 있는 스레드가 실행중인 스레드가 되게된다.

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

fixed\_point를 통해 소수의 사칙연산을 구현한다. 또한 소수와 정수 사이의 사칙연산도 구현한다. 이후로는 recent\_cpu, nice, priority값을 구하는 함수와 연산을 담당하는 함수를 만든다. 또한 주기적(4틱마다 혹은 1초마다)으로 모든 스레드의 recent\_cpu, nice, priority 값을 구하는 함수를 만든다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 기간 | 내용 |
| 1주차 (10/28 ~ 11/2) | * 매뉴얼 분석 * 자료구조 및 알고리즘 설계 * Alarm Clock 구현 |
| 2주차 (11/3 ~ 11/6) | * Priority Scheduling구현 * 테스트 및 코드 수정 * 보고서 작성 |

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명
  + 수정해야하는 소스코드
    1. Alarm clock 관련해서 수정한 함수

Timer.c에서

timer\_sleep()는 block된 스레드를 busy-waiting 방식이 아닌 wake-up 방식으로 구현 하는 것이 목적이고, timer\_interrupt()에서는 매번 깨울 스레드를 점검하는 것이 목적이다.

timer\_interrupt()에서 매 tick마다 sleep\_list에서 깨어날 thread가 있는지 확인을 해주어서 만약에 존재한다면 해당 스레드를 깨우는 함수를 호출한다.

Thread.c에서

깨울 스레드의 목록인 sleep\_list를 초기화 하기 위해 thread\_init 함수에서 sleep\_list를 초기화 해줄 함수를 구한다.

* + 1. Prirority scheduling 관련해서 수정한 함수

Thread.c에서

thread\_create()에서는 thrread를 초기화 하면서 필요한 추가적으로 생겨난 변수들을 초기화 하기 위해서 수정을 하였다. 또한 스레드를 생성하면서 우선순위에 변화가 생겼을 수 있으므로 마지막에 ready\_and\_running\_priority로 우선순위 계산을 해준다.

Thread\_yield() 에서는 스레드를 양보한 이후에 원래 실행되던 스레드가 우선순위에 따라 삽입되게 바꾸었다.

Thread\_unblock()에서는 우선순위로 정렬되어서 unblock된 list가 ready\_list에 순서대로 삽입되도록 수정하였다.

Thread\_set\_priority()

스레드의 우선순위가 변경 되었을 때 우선순위에 따라 선점이 발생하도록 한다.

Synch.c에서

Sema\_up 함수 수정을 통해 스레드가 waiters list에 있는 동안 우선순위가 변경 되었을 경우를 고려 하여 waiters\_list 를 우선순위로 정렬 한다. 이후에는 우선순위가 변경 되었을 수도 있으니 스레드 우선순위를 비교해준다.

Cond\_wait 함수에서 condtion variable이 waiters list에 우선순위 순서로 삽입되도록 수정한다.

* + 1. mlfq관련해서 수정한 함수

thread.c

init\_thread() 함수에서 nice 값과 recent\_cpu 값을 수정해준다.

Timer.c 에서

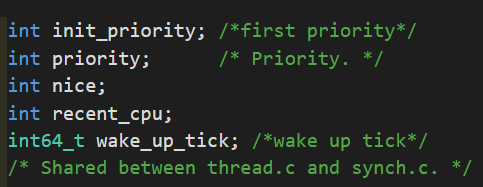
Timer\_interrupt()

Thread\_mlfqs 옵션이 true라면 매틱 마다 recent\_cpu 를 늘려주는 함수를 실행

4틱 마다 우선순위를 계산하는 함수를 호출 1초 마다 load\_avg를 계산 모든 thread 의 recent\_cpu를 계산해주는 함수를 호출

* + 수정하거나 추가해야 하는 자료구조

Thread.에 우선 순위를 추가하고 일어날 시간인 wake\_up\_tick추가하고 nice 값과 wake\_up\_tick을 추가함



* + 추가해야 하는 함수

1,alarm clock 관련한 함수

Thread.c

Thread\_sleep()

sleep list의 모든 entry 를 순회하며 다음과 같은 작업을 수행한다. 현재 tick이 깨워야 할 tick 보다 크거나 같다면 슬립 큐에서 제거하고 unblock 한다.

Thread\_awake()

현재 스레드가 idle 스레드가 아닐경우 thread의 상태를 BLOCKED로 바꾸고 깨어나야 할 ticks을 저장, 슬립 큐에 삽입하고, awake함수가 실행되어야 할 tick값을 update

2. priority scheduling 관련 추가한 함수

Thread.c

Priority\_setup()

list\_insert\_ordered() 함수에서 사용 하기 위해 정렬 방법을 결정하기 위한 함수 작성

max\_priority()

현재 readty\_list에 있는 thread 중에서 가장 우선순위가 높은 thread의 우선순위를 리턴한다.

Ready\_and\_running\_priority()

현재 running중인 스레드와 ready\_list에 있는 스레드를 비교를 해서 ready\_list에 있는 스레드의 우선순위가 높다면 thread\_yield를 통해 현재 running중인 스레드를 중단하고 우선 순위가 제일 높은 스레드를 running thread로 만든다.

3, mlfqs 관련 추가한 함수

Thread\_set\_priority()

현재의 스레드에 새로운 우선순위를 부여한다.

Thread\_get\_priority()

현재 스레드의 우선순위를 리턴한다.

3. mlfqs 관련한 함수

Mlfqs\_priority()

해당 스레드가 idle\_thread가 아니라면 소수점 계산을 통해 priority 를 리턴

Mlfps\_recent\_cpu()

인자로 주어진 스레드의 recent\_cpu 를 업데이트

Mlfqs\_load\_avg()

시스템의 load\_avg를 업데이트

Mlfqs\_increment()

현재 실행중인 스레드의 우선순위를 1 증가 시킴

mlfqs\_recent\_cpu()

인자로 주어진 스레드의 recent\_cpu를 업데이트

Recent\_cpu, priority 계산 구현을 위해서 floating point 계산을 하는 함수를 구현

int int\_to\_fp(int i);

integer를 fixed point로 전환

int fp\_to\_int\_round(int f);   
FP를 int로 전환(반올림)

int fp\_to\_int(int f);

FP를 int로 전환(버림)

int add\_fp(int f1, int f2);

FP의 덧셈

int add\_mixed(int f1, int i);

FP와 int의 덧셈

int sub\_fp(int f1, int f2); /\* FP의 뺄셈(f1-f2) \*/

int sub\_mixed(int f, int i)

FP와 int의 뺄셈(f-i)

int mult\_fp(int f1, int f2)

FP의 곱셈

int mult\_mixed(int f1, int f2);

FP와 int의 곱셈

int div\_fp(int f1, int f2);

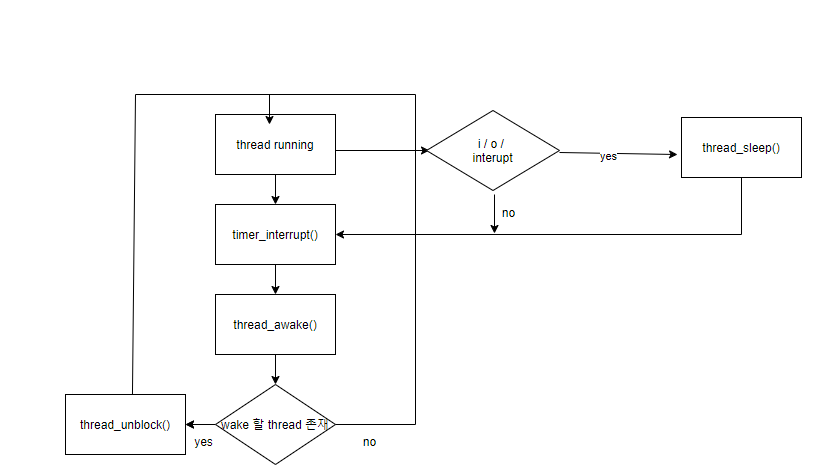
FP의 나눗셈(f1/f2)

int div\_mixed(int f, int i);

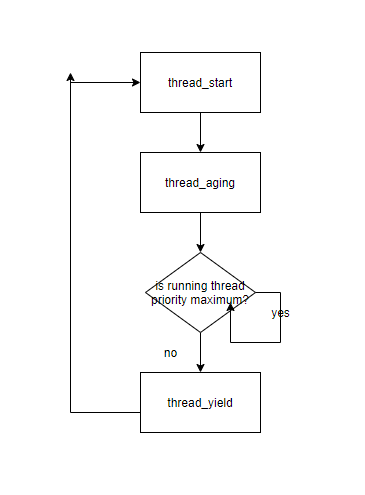
FP와 int 나눗셈(f/i)

1. **연구 결과**
   1. **Flow Chart**

Alarm clock 구현



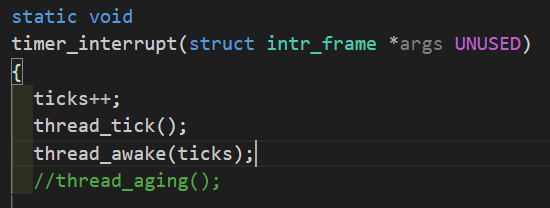
Priority 구현



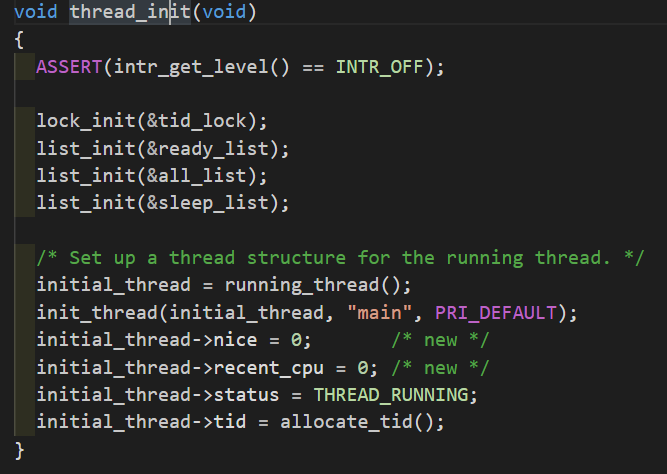
* 1. **제작 내용**
     1. **Alrmclock**

Sleep\_list => 아직 깨어날 시간이 안된 스레드들을 ready\_list에 넣는대신 모아 놓은 공간

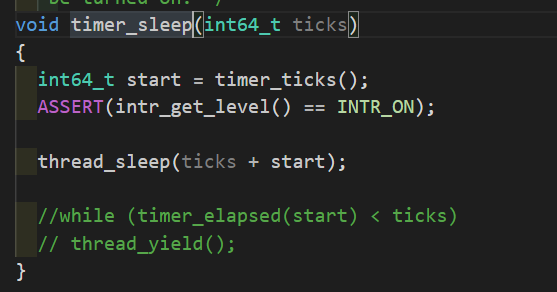
Wake\_up\_tick => 스레드가 깨어날 시간, 이전의 timer\_sleep 무한 루프 대신 넣어주는 thread구조체의 변수다.



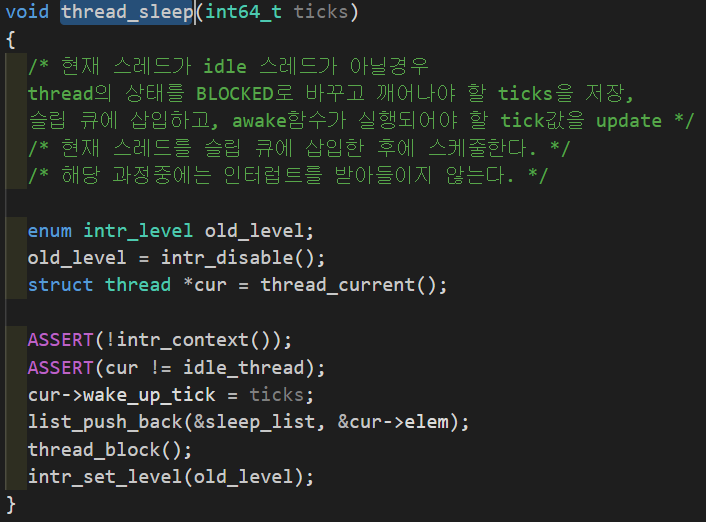
이전의 timer\_interrupt 에서는 thread\_tick 까지만 존재하였지만, 수정한 함수에는 thead\_awake 항목을 추가하여 interrupt마다 sleep\_list에 깨울 thread가 있는지 확인을 한다.



새로 생긴 sleep\_list를 맨처음 thread 생성시 생성해준다.

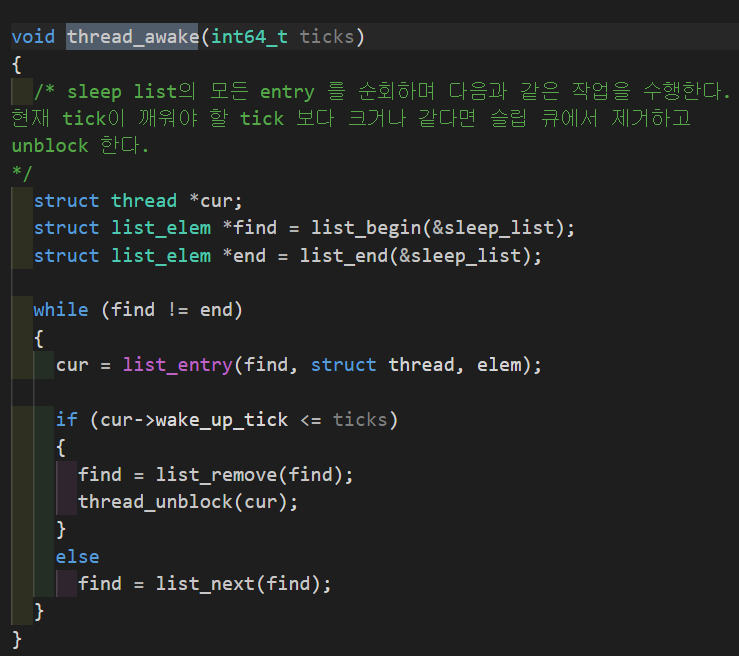


수정 전의 timer\_sleep 에서는 주석에 나온 것 처럼 지난 시간이 지정된 시간을 초과하기 전까지 계속 thread\_yield를 통해 ready\_list로 보내 버리는 방식을 사용하였다. 이렇게 되면 1틱이라도 시간이 덜 지난 경우에는 ready\_list 끝 편으로 밀려나서 자신의 차례가 올 때까지 다시 기다린다는 점이다. 이런 비효율적인 방법 대신에 thread\_sleep를 호출해서 sleep\_list에 넣어서 별도로 관리해준다.



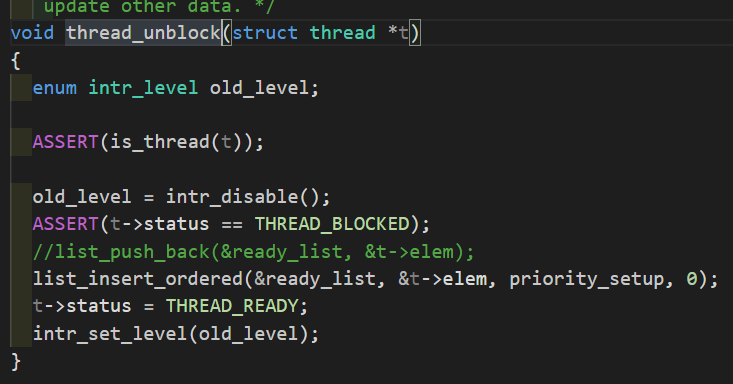
Thread\_sleep()

이 함수에서는 현재의 스레드가 idle\_thread가 아니라면 wake\_up\_tick을 기록해주고 sleep\_list로 보내준다. 그리고 thread\_block 을 통해 현재 스레드의 상태를 blocked로 만들어 준다.



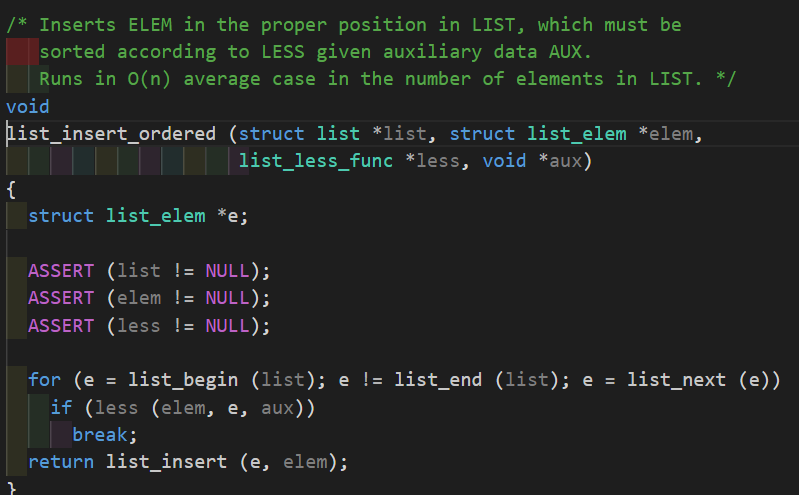
이전의 timer\_interrupt에서 호출을 하면 sleep\_list를 전체 순회하면서 만약에 께울 스레드. 즉, 자신의 wake\_up\_tick 이 tick보다 작다면 sleep\_list에서 제거하고 thread\_unblock을 불러서 ready\_list로 이동한다. 그리고 여기서 주의할 점은 sleep\_list에서 제거하므로 list\_next를 호출하면 모든 원소를 점검해보지 못하는 문제가 발생하므로 제거를 하고 그냥 루프를 돈다.

* + 1. **Priority scheduling**

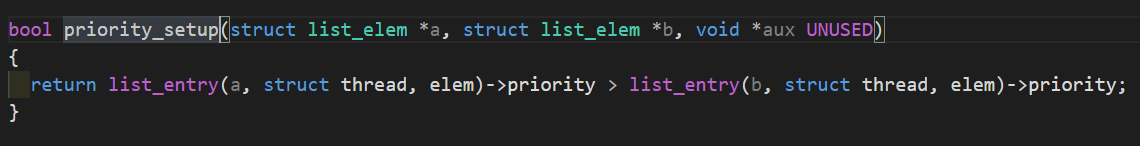


unblock을 할 원소가 들어오면 일단 interrupt를 중단하고 이전에는 그냥 ready\_list뒤에 넣었으나 지금은 list\_insert\_order를 통해서 priority\_setup함수에 따라서 우선순위에 따라서 ready\_list에 들어가게 된다.

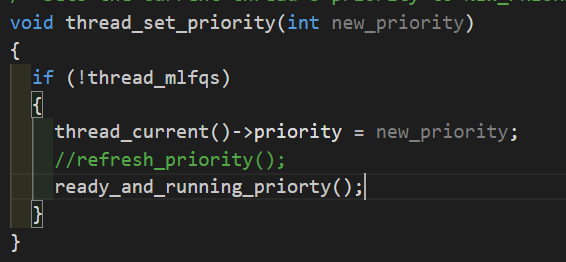
list\_insert\_order는 아래와 같은 함수로



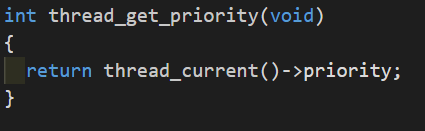
핀토스 내부 구현 함수로 들어온 less 함수의 정의에 따라 순서대로 배열을 진행해주는 함수다.



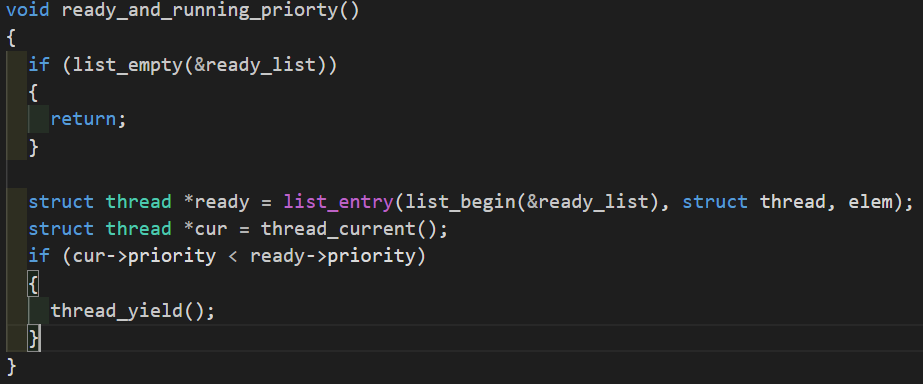
Priority\_setup내부에서는 priority 비교를 수행한다.



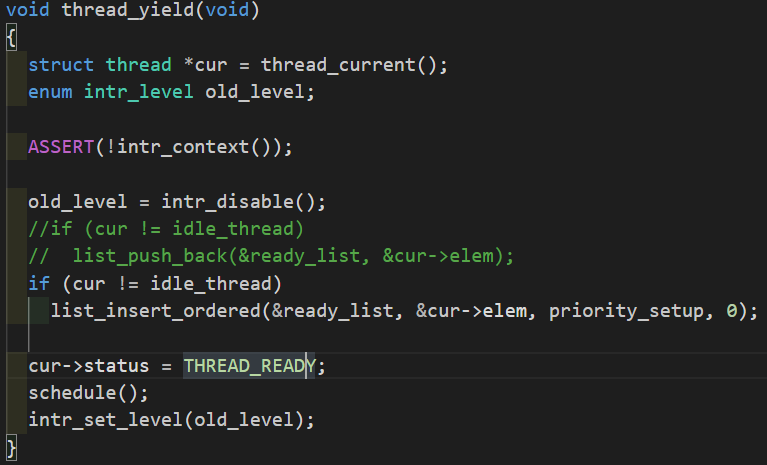
현재 thread의 priority를 인자로 받은 priority로 바꿔준다.



현재 thread의 priority 를 리턴한다.



현재 ready\_list 맨앞에 있는 thread와 현재 running하는 thread를 비교해서 우선순위가 높은 스레드를 running하게 만든다.



원래는 주석과 같이 새로 들어온 원소를 ready\_list 뒤에 넣었으나 list\_insert\_ordered를 통해 우선순위에 따라 삽입을 해서 정렬된 상태를 유지한다.

* + 1. **mlfqs**

mlfps 에서는 소수점 계산이 필수이나 핀토스에서는 기본적으로 소수점 계산이 지원되지 않는다. 그러므로 사용자가 직접 소수 연산을 구현해줘야 한다. Ppt에 나온대로 비트연산을 활용해서 소수 계산을 진행해준다.

**각각의 계산식은 다음과 같다**







소수 연산이 겹치는 부분이 있으므로 fixed\_float.c 와 fixed\_float.h를 만들어서 소수연산을 함수로 구현해준다.

int int\_to\_fp(int i)

{

    return i \* F;

}

/\* FP를 int로 전환(반올림) \*/

int fp\_to\_int\_round(int f)

{

    if (f >= 0)

        return (f + F / 2) / F;

    else

        return (f - F / 2) / F;

}

/\* FP를 int로 전환(버림) \*/

int fp\_to\_int(int f)

{

    return f / F;

}

/\* FP의 덧셈 \*/

int add\_fp(int f1, int f2)

{

    return f1 + f2;

}

/\* FP와 int의 덧셈 \*/

int add\_mixed(int f, int i)

{

    return f + i \* F;

}

/\* FP의 뺄셈(x-y) \*/

int sub\_fp(int f1, int f2)

{

    return f1 - f2;

}

/\* FP와 int의 뺄셈(x-n) \*/

int sub\_mixed(int f, int i)

{

    return f - i \* F;

}

/\* FP의 곱셈 \*/

int mult\_fp(int f1, int f2)

{

    return ((int64\_t)f1) \* f2 / F;

}

/\* FP와 int의 곱셈 \*/

int mult\_mixed(int f, int i)

{

    return f \* i;

}

/\* FP의 나눗셈(x/y) \*/

int div\_fp(int f1, int f2)

{

    return ((int64\_t)f1) \* F / f2;

}

/\* FP와 int 나눗셈(x/n) \*/

int div\_mixed(int f, int i)

{

    return f / i;

}

다음과 같이 구현하고 계산을 해주는 함수를 구현한다. 정수에서 핀토스에서 원하는 소수로 변환하려면 2^14를 곱해야 하므로 정수에서 소수로 바꿀때에는 F = 1<< 14 를 곱해주면서 생각을 한다. 정수에서 한번만 F를 곱하는 연산을 수행해야 소수를 구할 수 있으므로 곱셈이나 나눗셈은 F를 곱해주는 연산이 중복으로 되는 것을 위해서 소수끼리 연산을하면 F로 나눠주는 연산을 하는 것이다.

이렇게 소수점 연산을 구하고 recent\_cpu, load\_avg, priority를 구하는 연산을 수행한다.

void mlfqs\_priority(struct thread \*t)

{

  /\*priority계산식을 구현 (fixed\_point.h의 계산함수 이용)\*/

  t->priority = fp\_to\_int(sub\_fp(sub\_fp(int\_to\_fp(PRI\_MAX), div\_mixed(t->recent\_cpu, 4)), mult\_mixed(int\_to\_fp(t->nice), 2)));

  if (t->priority > PRI\_MAX)

  {

    t->priority = PRI\_MAX;

  }

  if (t->priority < PRI\_MIN)

  {

    t->priority = PRI\_MIN;

  }

}

Mlfqs 계산을 할때의 우선순위를 구현해준다.

void mlfqs\_recent\_cpu(struct thread \*t)

{

  /\* 해당 스레드가 idle\_thread 가 아닌지 검사 \*/

  /\*recent\_cpu계산식을 구현 (fixed\_point.h의 계산함수 이용)\*/

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

    t->recent\_cpu = add\_mixed(mult\_fp(div\_fp(mult\_mixed(load\_avg, 2), add\_mixed(mult\_mixed(load\_avg, 2), 1)), t->recent\_cpu), t->nice);

}

해당 스레드의 recent\_cpu를 구한다. 각각 적절하게 소수와 정수를 분리해서 계산한다.

void mlfqs\_load\_avg(void)

{

  /\* load\_avg계산식을 구현 (fixed\_point.h의 계산함수 이용) \*/

  /\* load\_avg 는 0 보다 작아질 수 없다.\*/

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

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

    ready\_size++;

  load\_avg = div\_mixed(add\_mixed(mult\_mixed(load\_avg, 59), ready\_size), 60);

}

현재의 load\_avg를 구한다. ready\_list는 list\_size라는 list.c에 있는 함수를 활용해서 원소의 개수를 구하고 만약에 idle\_thread가 아니라면

void mlfqs\_increment(void)

{

  /\* 해당 스레드가 idle\_thread 가 아닌지 검사 \*/

  /\* 현재 스레드의 recent\_cpu 값을 1증가 시킨다. \*/

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

    thread\_current()->recent\_cpu = add\_mixed(thread\_current()->recent\_cpu, 1);

}

해당 스레드가 idle\_thread가 아니라면 recent\_cpu값을 1증가 시킨다.

void mlfqs\_recalc\_recent\_cpu(void)

{

  /\* 모든 thread의 recent\_cpㅕ 재계산 한다. \*/

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

  struct list\_elem \*end = list\_end(&all\_list);

  while (cur != end)

  {

    struct thread \*temp = list\_entry(cur, struct thread, allelem);

    mlfqs\_recent\_cpu(temp);

    cur = list\_next(cur);

  }

}

All\_list를 순회하면서 모든 thread의 recent\_cpu값을 다시 계산한다.

void mlfqs\_recalc\_priority(void)

{

  /\* 모든 thread의 recent\_cpu와 priority값 재계산 한다. \*/

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

  struct list\_elem \*end = list\_end(&all\_list);

  while (cur != end)

  {

    struct thread \*temp = list\_entry(cur, struct thread, allelem);

    mlfqs\_priority(temp);

    cur = list\_next(cur);

  }

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

  {

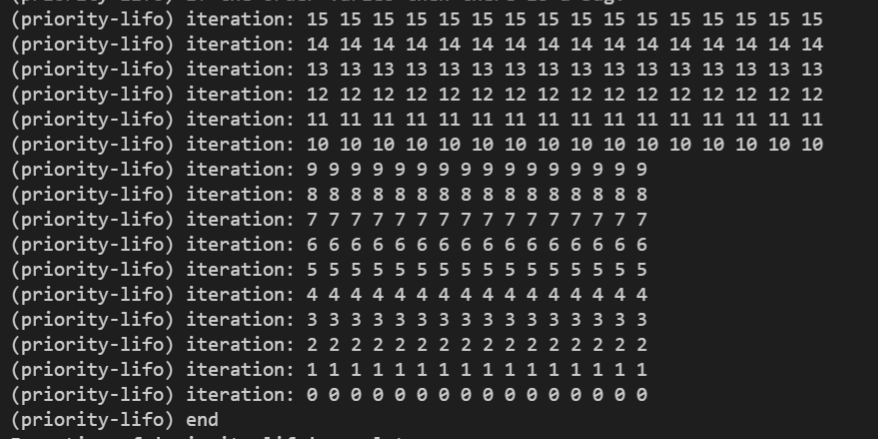
    intr\_yield\_on\_return();

  }

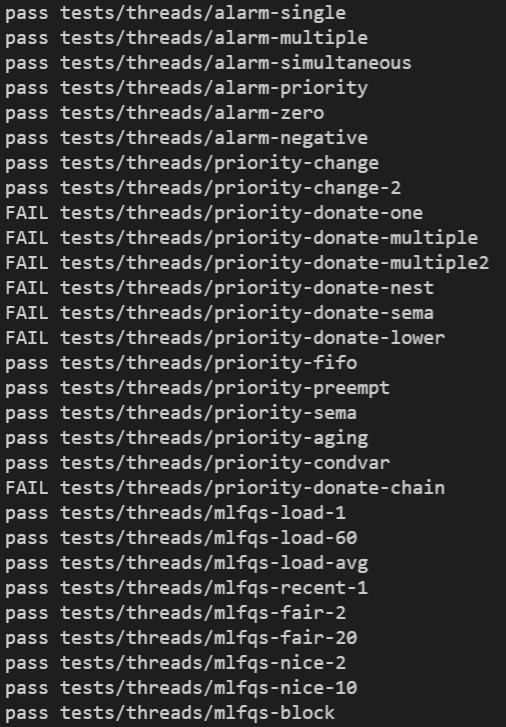
}

모든 cpu의 priority를 재계산한다.

* 1. **시험 및 평가 내용**



Make check



Make-grade