**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : 최준수 20151619

개발 기간 : 11/14~11/26

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

현재 timer\_sleep 함수는 일정 시간동안 thread를 재우는 역할을 하지만 사실 정해진 시간까지 계속 CPU를 양보하는 busy waiting 기법의 코드로 이루어져 있다. 이러한 CPU의 낭비를 막기 위해 Alarm Clock 기능을 구현해야 하며, 이는 일정시간 시간동안 thread를 Block 상태로 만든 다음 예정된 시간이 지나면 다시 thread를 Ready 상태로 만들어 주는 방식으로 구현할 수 있다.

현재 Thread scheduling 방식은 FIFO방식으로 새로운 thread가 입력되면 ready\_list의 맨 끝에 추가하는 구조로 되어 있다. 이에 각 thread마다 우선순위를 부여하여, scheduling이 우선순위에 의해 수행되도록 수정한다. priority의 범위는 0~63이며 우선순위가 높은 thread부터 수행한다. 이때 scheduling방식은 preemptive이다. 즉, 실행 중간에 priority가 바뀌거나, 새로운 priority를 가진 thread가 추가가 되었을 경우 실행하던 thread를 내보내고 rescheduling 해야 한다.

단순히 priority로 scheduling 했을 때, 낮은 priority를 가진 thread들의 starvation을 야기한다. 따라서 thread 생성 후 흐른 시간에 비례해서 priority를 높여주는 Aging 기법이 필요하다. 이를 위하여 Aging 기법을 적용한 BSD scheduler를 구현한다. 이때, priority 재계산을 위하여 float point 연산이 필요한데 pintos에서는 제공되고 있지 않다. 따라서 fixed point number형식으로 float point 연산을 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* 아래 각 항목 개발의 필요성 또는 개발 시 기대되는 결과를 간략히 서술
  1. Alarm Clock
  2. Priority Scheduling
  3. Advanced Scheduler (추가구현을 한 경우)
* **Alarm Clock**Alarm Clock의 기능을 구현하려면 해당 thread를 정해진 시간동안 Block 상태로 만들어줘야 하고 Block된 thread들을 관리할 수 있도록 list로 연결해주어야 한다. 그러기 위해 Block 상태에서 깨어나 ready 상태로 다시 돌아가야 하는 시간인 wakeup\_time을 thread 구조체에 추가해줘야 한다.

threads/thread.h  
struct thread {…}에 wakeup\_time 변수 추가

그 다음 thread의 sleep과 wakeup\_time이 일어날 수 있도록 devices/timer.c의 timer\_sleep 함수와 timer\_interrupt 함수를 수정해야 한다. 이외에 추가로 구현한 함수는 다음과 같다.

threads/thread.c

void thread\_sleep() // ticks 시간까지 thread를 Block 상태로 바꿈

devices/timer.c

void timer\_interrupt(): 매 틱마다 thread의 wakeup\_time을 검사하여 깨워줘야되는 thread가 있는지 확인한다.

* **Priority Scheduling**

Pintos에서는 이미 thread에 priority가 부여되어 있다. 하지만 thread를 생성시에 priority를 넣어주지 않았다. 따라서 이 부분을 추가하여야 한다. priority scheduling을 위해서는 ready\_queue는 multi level queue처럼 관리해야 하는데 이를 single queue로서 구현한다. 따라서 priority 내림차순으로 single ready queue를 sorting 된 상태로 유지하고, 맨 앞에서부터 scheduling을 수행한다. preemptive 방식을 구현하기 위해서는 현재 돌고 있는 thread를 내보내고 rescheduling해야한다. 따라서 각 thread의 priority가 변할 때, 새로운 thread가 ready queue로 들어올 때 그리고 block 되어있는 thread가 unblock 될 때 등의 경우를 고려하여 preemptive 방식을 구현한다. 따라서 수정해야하는 범위는 다음과 같다.

(threads/thread.c)

void init\_thread() //thread에 priority를 저장한다.

void thread\_yield() //running 상태에서 ready queue로 내보낼 때 sorting을 유지하도록 한다.

void thread\_unblock() //block상태에서 unblock 상태가 될 때, ready queue는 sorting 상태를 유지하도록 한다.

tid\_t thread\_create() //새로 들어오는 thread가 있을 때 priority에 따라 re-scheduling 해야 한다.

void thread\_set\_priority() //기존 thread의 priority를 변화시에 priority에 따라 re-scheduling 해야한다.

void thread\_get\_priority() //현재 thread의 priority를 반환한다.

bool compare\_priority() //두 thread의 priority를 비교하며 왼쪽이 크면 true를 반환한다.

(threads/synch.c)

void sema\_up() // sema up을 통해 unblock 되는 thread가 존재할 때, 가장 priority가 큰 thread를 unblock 해야 하며, 이후 re-scheduling 한다.

* **Fixed-Point Real Arithmetic**현재 pintos는 실수연산 기능이 구현되어 있지 않기 때문에 실수 값을 가지는 load\_avg나 recent\_cpu는 부동소수점 방식으로 저장할 수 없다. 따라서 fixed-point 형식으로 표현된 실수의 연산에 필요한 함수들 또한 구현이 필요하다
* **Advanced Scheduler**

Aging을 구현하기 위하여 필요한 변수인 nice, recent\_cpu, load\_avg를 추가한다. 이를 일정시간마다 update 시켜야 하며, 이에 따라 priority 역시 일정시간 마다 재계산한다. 이때, priority scheduling scheme을 유지하기위해 priority update마다 re-scheduling한다. 따라서 수정해야하는 범위는 다음과 같다.

(threads/thread.h)

struct thread{...} //nice, recent\_cpu 변수 추가

(threads/thread.c)

static int load\_avg // ready 상태 thread수 평균 추정

void init\_thread() //생성되는 thread의 nice, recent\_cpu값 할당한다.

void thread\_init() //initial thread의 nice, recent\_cpu 초기값 할당한다.

thread\_set\_nice() //현재 thread의 nice값 새로운 값으로 설정한다.

thread\_get\_nice() //현재 thread의 nice값 반환한다.

thread\_get\_load\_avg() //load\_avg \* 100 을 정수형으로 변환하여 반환한다.

thread\_get\_recent\_cpu() //현재 thread의 recent\_cpu \* 100을 정수형으로 변환하여 반환한다.

void update\_load\_and\_recent\_cpu() //exponentially weighted moving average 기법에 따라 load\_avg를 update하고 이를 사용하여 recent cpu를 update한다

void update\_priority() // update된 load\_avg와 recent\_cpu 그리고 nice에 따라 priority를 계산하여 update한다. ready\_queue를 sorting한 후 re-scheudling 한다.

(devices/timer.c)

void timer\_interrupt() // aging값이나 mlfq값이 true일때, 일정 시간마다 load\_avg, recent\_cpu, priority가 재계산되도록 수정한다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.
2. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.
3. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)

* **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술**

이번 프로젝트의 구현 범위에서는 timer\_sleep()을 통해 thread\_block()이 호출되어 쓰레드가 block되게 되는데 그러면 thread.h에 선언된 wakeup\_time에 해당 thread가 깨어나야되는 시간을 기록하고 sleep\_list에 추가하고 thread\_block()이 호출된다.

block된 thread는 timer\_interrupt()가 호출될 때마다 증가되는 변수인 ticks와 thread의 wakeup\_time을 비교하여 ticks가 더 작을 경우, sleep\_list에서 제거되고 thread\_unblock()이 호출된다. thread\_unblock()은 block되어있던 thread를 다시 ready\_list에 추가하여 schedule이 될 수 있도록 한다.

* **Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.**
* 각 프로세스마다 우선순위를 부여하여 우선순위가 높은 process부터 수행하는 scheduling이다. 우선순위는 내부적/외부적 조건에 따라 정의될 수 있다. 예를 들어 시간 제한, 메모리 요구량, 열린 파일의 수, 프로세스의 중요성, 자원사용의 비용 등을 고려하여 프로세스의 우선순위를 책정하며 이에 따라 컴퓨터는 scheduling을 진행한다. 우선 순위 스케줄링에는 preemptive/non-preemptive 방식이 있는데 현 project에서는 preemptive방식을 사용한다. 따라서 실행되는 thread는 실행가능한 thread 중에서 가장 높은 우선순위를 가지고 있어야 하므로 ready queue 상황에 따라 실행중인 thread를 강제로 내려야하는 상황이 올 수 있다. 즉, 실행 중간에 priority가 바뀌는 경우, 새로운 thread가 추가되는 경우 마지막으로 thread가 unblock 되는 경우 실행중인 thread보다 높은 priority를 가진 thread가 존재한다면 실행중인 thread를 내리고 새로 scheduling한다. priority scheduling의 문제점으로 starvation을 뽑을 수 있다. starvation 현상은 높은 priority thread에 밀려 낮은 priority thread가 계속해서 우선순위에서 밀려 사실상 실행되지 못하는 상황을 말한다. 이에 대한 해결책으로 Aging 기법이 있다.
* **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술. (추가구현을 한 경우)**
* BSD scheduler는 General-purpose 스케줄러로서 Thread의 Scheduling Needs를 균형적으로 충족시키는 것이 목적이다. 따라서 priority scheduler에서 발생한 starvation 문제를 Aging 기법을 통해 해결한다. Aging 기법이란 시간이 지날수록 priority를 점차 낮추어 일정시간이 지나면 priority가 낮았던 thread들도 scheduling이 될 수 있게 한다. 이러한 Aging을 위해 다음과 같은 변수들이 필요하다.
* Niceness : nice는 다른 thread에게 얼마나 nice한지 판단하는 척도이다. 즉 nice값이 높을수록 priority가 떨어지는 속도가 빨라 이타적이며 nice 값이 적을수록 priority가 떨어지는 속도가 느려 이기적이라고 할 수 있다.
* Recent cpu : recent\_cpu는 최근에 cpu를 사용한 시간을 추정하는 변수이다. recent\_cpu 가 클수록 오래 cpu를 점유했다는 뜻으로 Aging scheme에 따라 priority가 더 줄어 다른 thread에게 cpu를 양보해야 한다.
* Load average : load average는 현재 실행을 대기하는 thread의 수를 평균적으로 추정한 것이다. 이 load average는 recent cpu값을 추정하는데 필요하며 load average가 높을 수록 cpu 점유경쟁중인 thread가 많다는 것을 의미한다. load average의 계산은 60초간 ready 상태인 thread의 수로 계산하며, 속도를 위해 exponentially weighted moving average 방식으로 계산한다.
* 이 변수들을 이용하여 다음과 같이 priority를 재계산한다.
* **priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)**
* 따라서 recent cpu 시간이 많을수록, nice가 클수록 priority가 더 낮게 책정되며, 이를 통해 starvation을 해결 하여 Thread의 Scheduling Needs간 균형조정을 효과적으로 할 수 있다

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11/14~16: 과제 이해 및 ppt, pintos 매뉴얼 정독

11/17~11/24: 과제 구현

11/25~11/26: 레포트 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**

devices/timer.c  
timer\_sleep() : 원래 timer\_sleep 함수는 ticks 시간만큼 threads를 재워야 하는 함수이지만 현재는 busy waiting의 기법으로 구현되어 있어 경과된 시간과 계속 비교하면서 CPU를 양보하게 된다. 따라서 정말로 thread를 Block 상태로 만들기 위해 thread\_sleep 함수를 호출하도록 수정한다.  
timer\_interrupt() : timer\_interrupt 함수는 매 tick이 발생할 때 wakeup 시간에 다다른 Block 상태의 thread들을 깨워줘야 한다. Block 상태의 thread들이 존재하는 list를 순회하면서 thread의 wakeup 시간과 현재 경과된 시간을 비교한다. 만약 wakeup이 시간이 더 작으면 리스트에서 thread를 끄집어낸 후 thread\_unblock()함수를 통해 다시 ready 상태로 만든다.

threads/thread.c  
void thread\_sleep() : 현재 thread를 ticks 시간까지 재워야 하는 함수이므로 thread의 wakeup 변수를 ticks로 설정한 뒤 Block상태인 thread들의 list에 삽입한다. 그 다음 thread\_block()을 통해 해당 thread를 Block시킨다.

* **Fixed-Point Real Arithmetic**fixed-point 방식으로 표현된 실수를 연산하는 과정에서 필요한 함수를 다음과 같이 작성하였다.  
  int int\_to\_fixed(int i): integer representation으로 저장되어있는 i를 fixed point로 변환해준다

int fixed\_to\_int(int f): fixed point로 저장되어있는 F를 integer representation으로 바꾸어준다

int fixed\_mult(int f1, int f2) fixed point로 저장되어있는 f1과 f2를 곱하여 fixed\_point로 반환한다.

int fixed\_div(int f1, int f2) fixed point로 저장되어있는 f1과 f2를 나누어 fixed\_point로 반환한다.

나머지 int와 fixed\_point간의 연산은 int와 int끼리 연산하는 것처럼 동일하게 수행하면 되므로 따로 작성하지 않았다.

* **Priority Scheduling**

(threads/thread.c)

void init\_thread() : 초기화 하려는 thread에 parameter로 전달된 priority를 대입한다.

void thread\_yield() : ready\_queue에는 현재 priority가 내림차순으로 sorting되어있는 상태이다. 따라서 thread\_yield를 통해 ready queue로 내보낼 때, sorting을 유지시키기 위하여 list\_insert\_ordered 함수를 이용한다.

void thread\_unblock() : ready\_queue에는 현재 priority가 내림차순으로 sorting 되어 있는 상태이다. 따라서 thread\_unblock을 통해 unblock 상태가 될 때 sorting 상태를 유지하기 위하여 list\_insert\_ordered 함수를 이용한다.

tid\_t thread\_create() : 새로 들어오는 thread가 현재 running 중인 thread보다 priority가 높은 경우가 발생한다. 이 때, thread\_yield를 호출하여 re-scheduling을 한다.

void thread\_set\_priority() : 기존 thread의 priority를 바꾸고 현재 running 중인 thread보다 priority가 높은 경우 thread\_yield를 통해 re-scheduling을 한다.

void thread\_get\_priority() : 현재 thread의 priority를 반환한다.

bool compare\_priority() : 두 thread의 priority를 비교하며 왼쪽이 크면 true를 반환한다.

(threads/synch.c)

void sema\_up() : sema up을 통해 unblock 될 때, sema->waiters의 thread들을 순회하면서 가장 priority가 높은 thread를 찾아 waiters list에서 제거하고 unblock해 준다..

* **BSD scheduler**

(threads/thread.h)

struct thread{...} : Aging을 위하여 thread 구조체에 nice와 recent\_cpu 변수를 추가한다.

(threads/thread.c)

void init\_thread() : 생성되는 thread의 nice, recent\_cpu값은 부모 thread의 값을 상속한다.

void thread\_init() : initial thread의 nice, recent\_cpu 초기값 0을 할당한다.

thread\_set\_nice() : 현재 thread의 nice값 새로운 값으로 설정한다.

thread\_get\_nice() : 현재 thread의 nice값을 반환한다.

thread\_get\_load\_avg() : load\_avg는 fixed point 형식이므로 load\_avg \* 100 을 정수형으로 변환하여 반환한다.

thread\_get\_recent\_cpu() : 현재 thread의 recent\_cpu는 fixed point 형식이므로 recent\_cpu \* 100을 정수형으로 변환하여 반환한다.

void update\_load\_and\_recent\_cpu() : idle thread가 아니면서 READY 상태인 thread의 개수를 count하여 ready\_threads를 구하고, exponentially weighted moving average 기법에 따라 다음과 같이 load\_avg를 업데이트 한다.

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

exponentially weighted moving average 기법에 따라 load\_avg를 이용하여 다음과 같이 thread들의 recent\_cpu를 업데이트 한다.

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

단 업데이트되는 thread는 idle thread와 DYING 상태가 아니어야 한다.

void update\_priority() : update된 load\_avg와 recent\_cpu 그리고 nice에 따라 모든thread 들의 priority를 다음과 같이 계산하여 update한다.

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

이때 새로운 priority가 최댓값을 넘어갈 경우 최댓값으로 고정해주고, 최솟값을 넘어갈 경우 최솟값으로 고정해준다. 이때 ready\_queue의 정렬성을 유지하기 위해 list\_sort를 호출해주며 새로운 priority가 running thread보다 큰 경우 thread\_yield를 통해 re-scheduling을 진행한다.

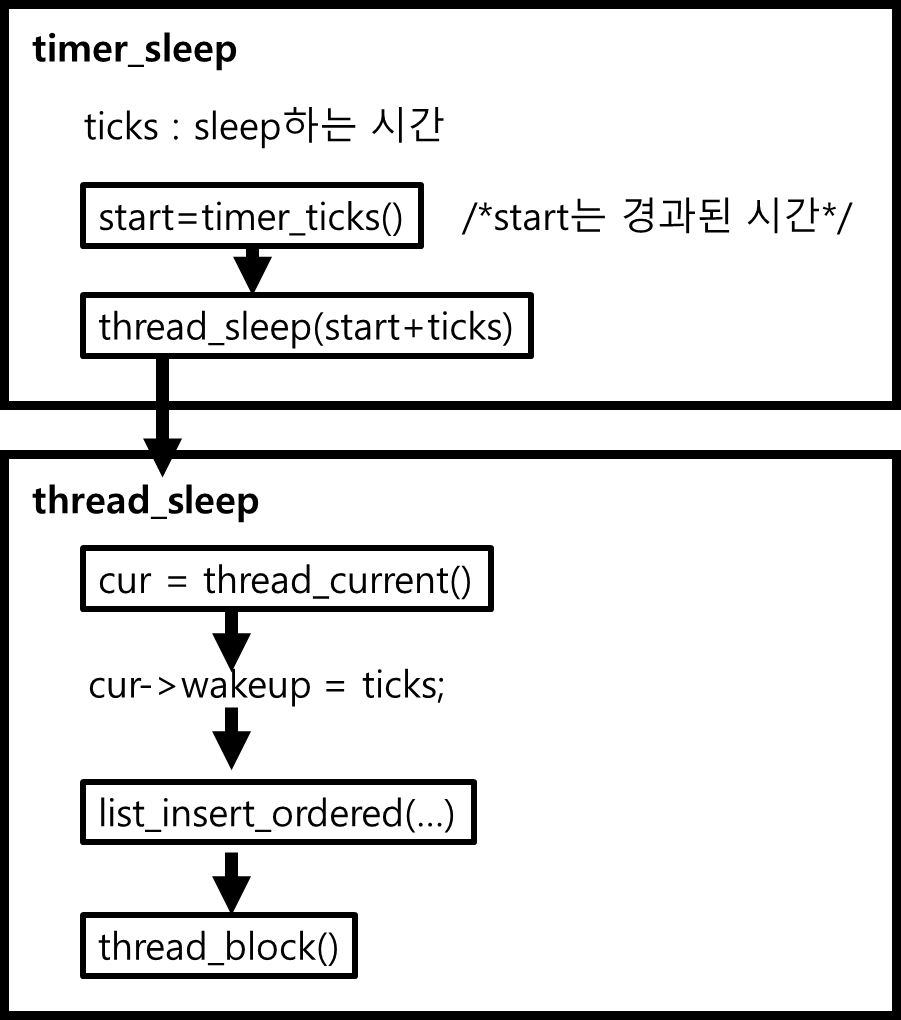
bool isidle() : 받은 thread가 idle thread이면 true를 반환한다.

(devices/timer.c)

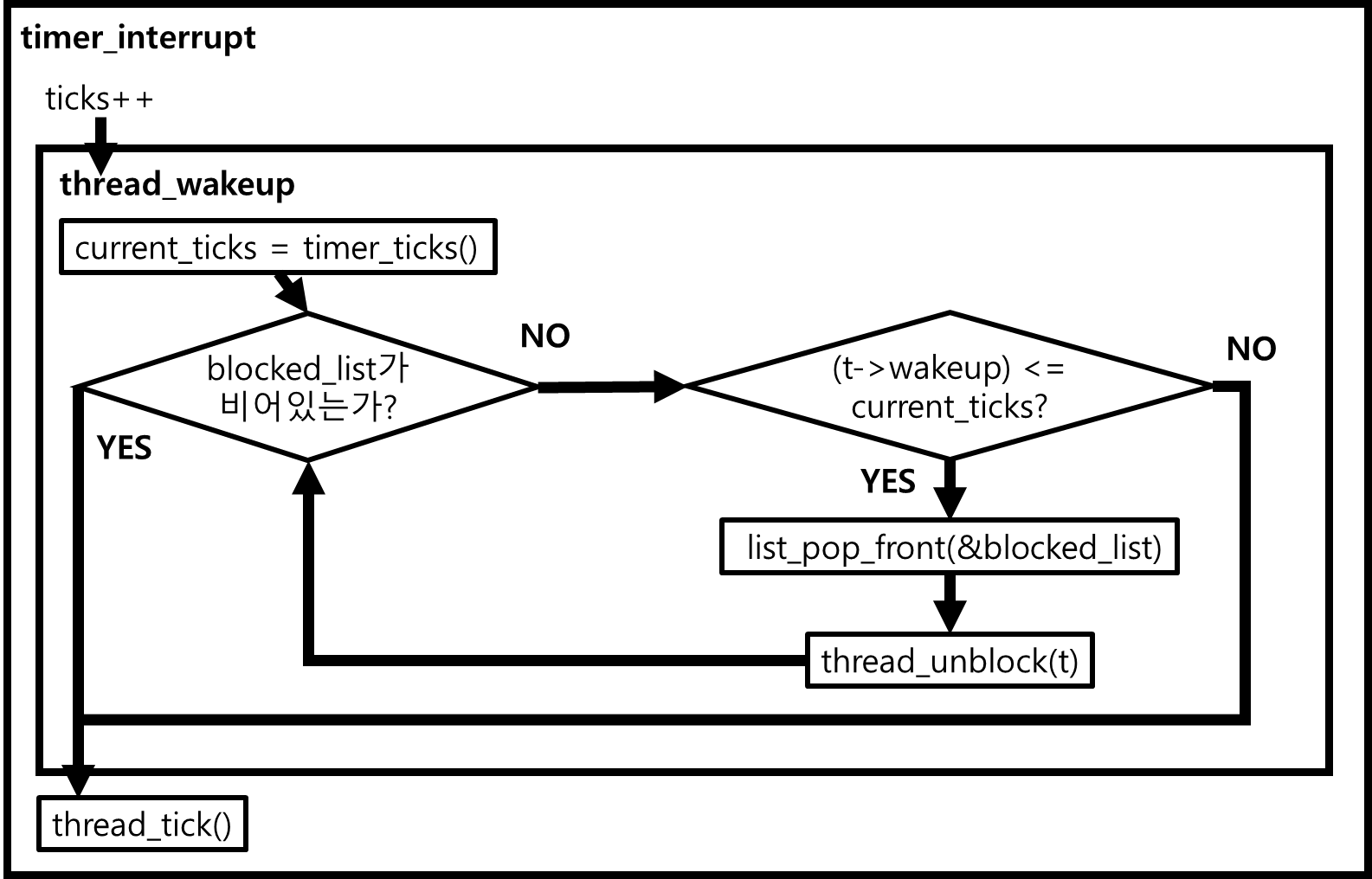
void timer\_interrupt() : aging값이나 mlfq값이 true일때, 일정 시간마다 load\_avg, recent\_cpu, priority가 재계산해야 한다. 따라서 1초(timer\_ticks() % TIMER\_FREQ == 0)마다 update\_load\_avg와 update\_recent\_cpu를 호출해야 하며 4틱(timer\_ticks() % 4 == 0)마다 priority를 재계산해야 한다.

1. **연구 결과**
   1. **Flow Chart**

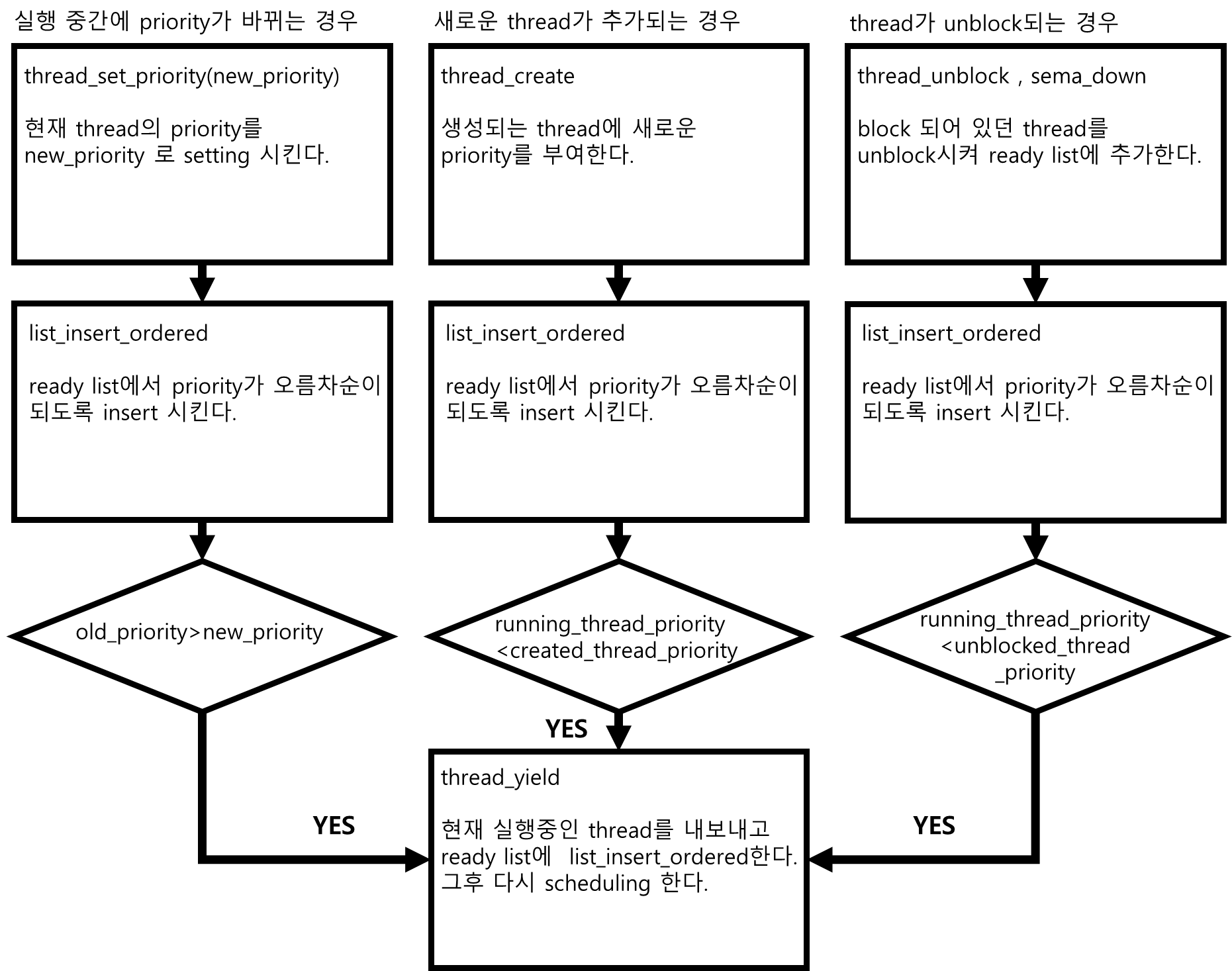
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* **thread의 sleep 과정**

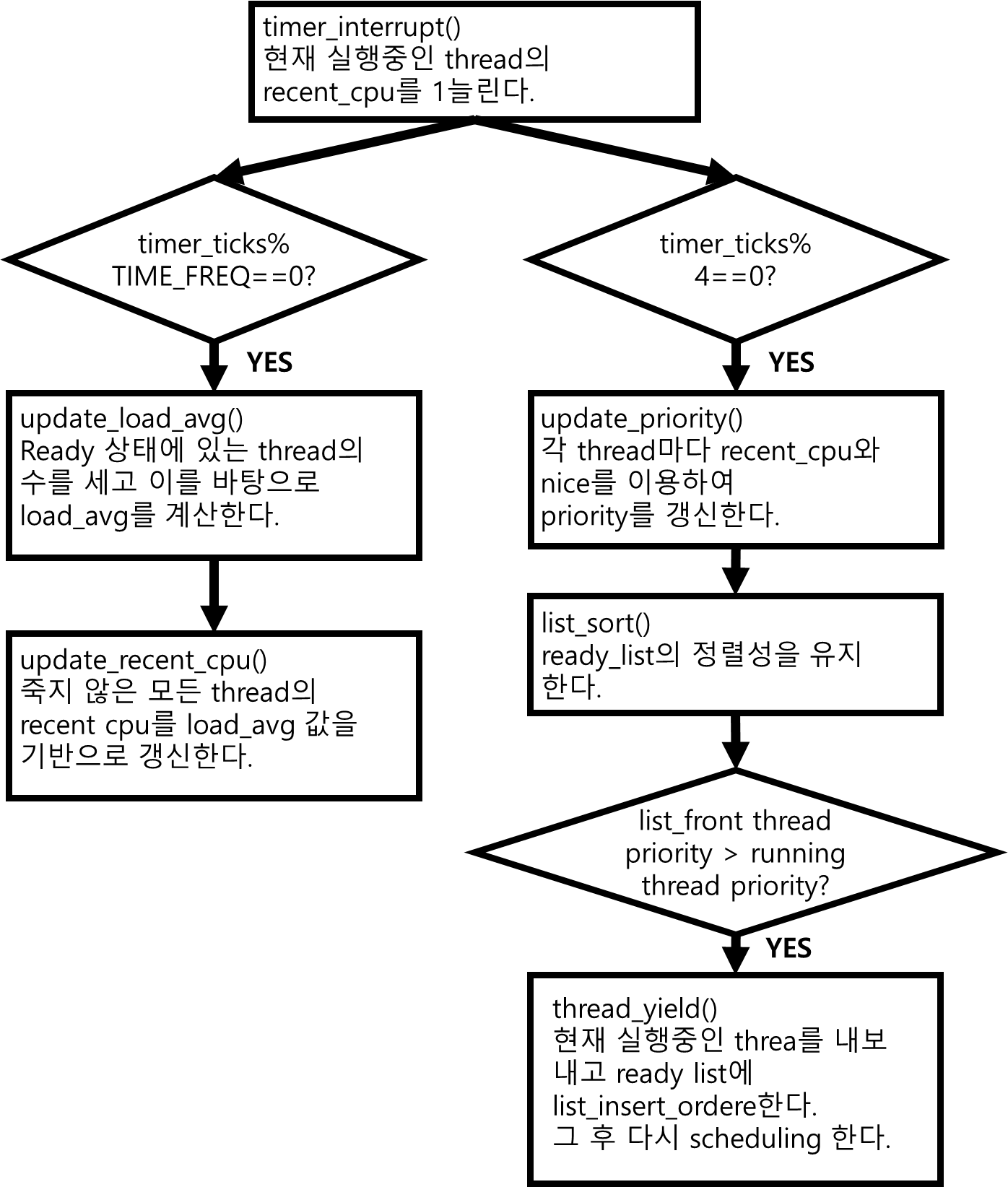
****

* **thread의 wake up 과정**

****

* **Priority Scheduling 구현**

****

* **Aging In BSD Scheduler**
* ****
  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
* **Alarm Clock**Alarm Clock 기능을 구현하기 위해서는 현재 busy waiting의 기법으로 구현되어 있는 devices/timer.c 의 timer\_sleep 함수를 먼저 수정해야 한다. 현재 이 함수는 thread가 Block 상태에 있지 않고 Ready 상태에 있으면서 자신의 차례가 될 때마다 계속 CPU의 권한을 양보한다. 이는 진정한 sleep이 아니기 때문에 thread\_block을 통해 thread를 Block상태로 바꿔줘야 한다. 이 부분은 threads/thread.c에 thread\_sleep이라는 별도의 함수를 통해 구현해주었다. thread\_sleep함수는 parameter로 sleep이 끝나는 시간이 들어가기 때문에 timer\_sleep함수에서 호출할 때 thread\_sleep(start+ticks)의 형태로 호출해주어야 한다. thread\_sleep 함수에선 우선적으로 interrupt가 걸리는 걸 방지하기 위해 intr\_disable 함수를 호출해줘야 하고 block 상태의 thread가 깨어나야 하는 시간인 wakeup을 parameter인 ticks로 저장해주어야 한다. 그 다음엔 해당 thread를 Block된 thread들의 list인 sleep list에 추가해주었다. 이런 방법으로 thread가 Block 상태가 되어 sleep\_list에 들어가게 되면 wakeup 시간에 도달했을 때 해당 thread를 다시 Ready 상태로 만들어줘야 한다. 이 wakeup 과정은 timer\_interrupt함수가 호출될 때마다 sleep\_list를 순회하면서 이루어져야 한다. timer\_interrupt함수가 호출되면 sleep\_list를 처음부터 순회하며 현재의 시간인 tick과 thread의 wakeup\_time을 비교하여 tick보다 작을 경우 sleep list에서 제거하고 thread\_unblock()을 호출하여 ready list에 다시 추가해준다.
* **Fixed-Point Real Arithmetic**fixed-point 방식으로 표현된 실수를 연산하는 과정에서 필요한 함수를 다음과 같이 작성하였다.  
  int int\_to\_fixed(int i): int형인 i를 1<<14인 FRACTION과 곱해 fixed point로 변환해준다

int fixed\_to\_int(int f): fixed point로 저장되어있는 f를 1<<14인 FRACTION으로 나누어 integer representation으로 바꾸어준다

int fixed\_mult(int f1, int f2) fixed point로 저장되어있는 f1과 f2를 곱하여 fixed\_point로 반환한다. overflow를 방지하기 위해 int64\_t로 변환한뒤 두 fixed\_point를 곱한다. FRACTION이 두번 곱해졌으므로 한번 나누어준다.

int fixed\_div(int f1, int f2) fixed point로 저장되어있는 f1과 f2를 나누어 fixed\_point로 반환한다. overflow를 방지하기 위해 int64\_t로 변환한 뒤, 두 fixed\_point를 나누어주는데 FRACTION이 사라지기 때문에 한번 우선적으로 곱하고 진행한다.

나머지 int와 fixed\_point간의 연산은 int와 int끼리 연산하는 것처럼 동일하게 수행하면 되므로 따로 작성하지 않았다.

* **Priority Scheduling**현재 Thread scheduling 방식은 FIFO방식으로 새로운 thread가 입력되면 ready\_list의 맨 끝에 추가하는 구조로 되어 있다. 이에 각 thread마다 우선순위를 부여하여, scheduling이 우선순위에 의해 수행되도록 수정해야한다. 따라서 init\_thread()에서 초기화하려는 thread에 priority를 설정한다. priority의 범위는 0~63이며 우선순위가 높은 thread부터 수행한다. 이를 multi level queue를 이용하여 우선순위에 따라 ready queue를 구분해도 되나, 현 프로젝트에서는 single queue에서 priority 내림차순으로 정렬된 상태를 유지함으로써 구현하였다.

이때, scheduling방식은 preemptive이다. 즉, 실행 중간에 priority가 바뀌거나, 새로운 priority를 가진 thread가 추가가 되었을 경우 또는 block되어 있던 thread가 unblock 되었을 경우 running하던 thread를 내보내고 re-scheduling 해야 한다.

<실행 중간에 priority가 바뀌는 경우>

thread\_set\_priority 함수는 현재 thread의 priority를 바꾼다. running thread는 READY 상태에 있는 thread들 보다 priority가 높아야 하는데 만약 새로운 priority가 현재 가진 priority보다 낮아진다면 thread\_yield를 호출하여 running중인 thread를 빼내고 re-scheduling을 진행한다.

<새로운 thread가 추가 되는 경우>

thread\_create함수는 새로운 thread를 ready queue에 넣어주는 함수이다. 생성되는 thread는 고유 priority를 가지고 있다. ready queue는 priority가 내림차순으로 정렬되어 유지되어야 하므로 기존 FIFO 방식처럼 맨 뒤에 넣어준다면 sorting 상태가 유지될 수 없다. 이에 대한 해결책으로 compare\_priority를 비교 함수로 같는 list\_insert\_ordered를 통해 삽입한다면 ready\_list의 정렬 상태가 유지된다. 또한 삽입 되는 thread의 priority가 현재 진행중인 thread보다 높아 질 수 있는데 preemptive 정책에 따라 thread\_yield를 통한 re-scheduling을 진행한다.

<thread가 unblock 되는 경우>

block 상태에 있던 thread가 thread\_unblock함수를 통해 unblock 된다면, 이는 scheduling 될 수 있는 상태가 된다. 따라서 priority가 높았지만 block 상태에 놓여있어 scheduling 될 수 없었던 thread들은 unblock이 되면서 ready list에 추가 되어야 한다. 이 때 ready\_list의 priority는 내림차순이 유지되어야 하므로 list\_insert\_ordered를 이용하여 삽입한다 또한 현재 running thread보다 unblock되는 thread의 priority가 높다면 preemptive 정책에 따라 thread\_yield를 통한 re-scheduling을 진행한다.

sema\_up의 경우, semaphore의 wait\_list에서 thread를 하나 unblock 해준다. 이때, Priority scheduling scheme을 지키려면 아무나 unblock 하는게 아닌 가장 높은 우선순위를 가진 thread를 unblock 시켜야 한다. 만약 wait\_list가 ready\_queue 처럼 priority 내림차순으로 정렬되어 있다면, 가장 앞의 list\_element가 가장 큰 priority를 가졌다고 할 수 있다.. 따라서 sema\_down 함수를 통해 thread가 block 되어 wait\_list에 추가 될 때, list\_insert\_ordered 함수를 통하여 삽입하면 wait\_list는 정렬된 상태를 유지할 수 있다.

thread\_yield 함수는 현재 running 되는 thread를 ready queue로 내보내고 scheduling 함수를 통해 re-scheduling을 진행한다. ready queue는 항상 priority가 내림차순으로 정렬되어 있어야 하므로 내보낼 때, list\_insert\_ordered 함수를 사용한다.

* **BSD scheduler**

단순히 priority로 scheduling 했을 때, 낮은 priority를 가진 thread들의 starvation을 야기한다. 따라서 thread 생성 후 흐른 시간에 비례해서 priority를 높여주는 Aging 기법이 필요하다. Aging이란 시간이 지날 수록 priority를 줄여, 낮은 priority를 가진 thread도 schedule 될 수 있게 하는 것이다. 따라서 매 4틱마다 update\_priority를 호출하여 Aging을 구현하였다. 이때, priority는 다음과 같이 재계산 된다.

**priority = PRI\_MAX – (recent\_cpu / 4) – (nice \* 2) //매 4틱**

**recent\_cpu = recent\_cpu + 1 //매 1틱**

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

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

재계산에 필요한 변수가 바로 recent\_cpu와 nice그리고 load\_avg이다. recent\_cpu와 nice는 thread마다 가지고 있어야 하므로 thread 구조체에 변수를 추가해주었으며 load\_avg는 ready\_list의 상태이므로 전역으로 처리한다. 이 변수에 대한 interface를 제공하기 위하여 thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 를 구현하였다. 각 변수들의 초기값은 load\_avg의 경우 시작 시 0이므로 thread\_init 함수에서 초기화 시켜주었고, nice와 recent\_cpu의 경우 initial thread는 thread\_init함수에서 0으로 나머지 thread는 init\_thread 함수에서 부모의 값을 상속받도록 하였다.

매 틱마다 timer\_interrupt 함수가 호출되어 load\_avg, recent\_cpu, priority의 update 시점을 고정할 수 있다. timer\_ticks는 한 틱이 1이고 TIME\_FREQ만큼이 1초이므로 매 틱마다 running thread의 recent\_cpu를 1초씩 늘리고, load\_avg와 recent cpu는 timer\_ticks % TIME\_FREQ == 0 일때 update\_load\_avg, update\_recent\_cpu를 호출하고 priority는 timerticks % 4 == 0 일 때 update\_priority를 호출한다면 요구시점에 맞게 update를 수행 할 수 있다.

update함수는 총 3가지가 존재한다.

<update\_load\_avg>

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

ready\_thread를 구하기 위하여 all\_list를 순회하여 idle thread가 아니면서 READY 상태인 thread의 개수를 count한다. load\_avg update formula에 따라 load\_avg를 계산한다.

<update\_recent\_cpu>

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

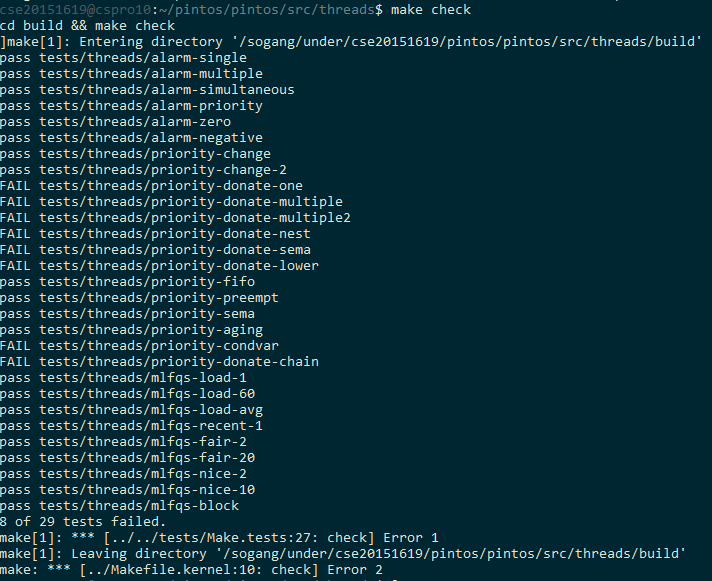
idle thread가 아니면서 DYING 상태가 아닌 모든 thread들의 recent cpu를 업데이트 해야 하므로 all\_list를 순회하여 해당 조건에 맞는 thread를 찾고 recent\_cpu update formula에 따라 recent\_cpu를 계산한다.

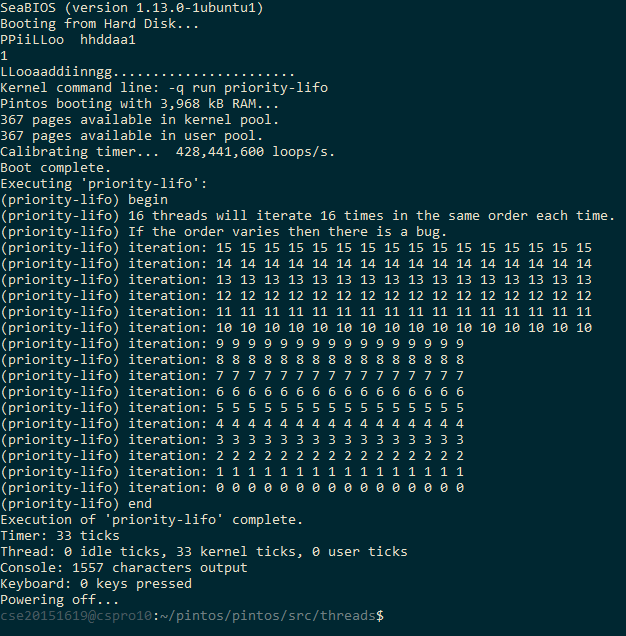
<update\_priority\_cpu>

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

Aging의 가장 최종 단계로서 모든 thread들의 priority를 재계산해야 한다. 따라서 all\_list를 순회하여 priority update formula에 따라 새로운 priority를 계산한다.

* 이때 새로운 priority가 범위(0~63) 을 벗어 날 수 있는데, 최댓값을 넘어갈 경우 최댓값으로 고정해주고, 최솟값을 넘어갈 경우 최솟값으로 고정해줘야 한다. 또한 priority가 재계산되면서 ready\_queue의 정렬성이 어긋날 수 있다. 그러므로 list\_sort를 호출하여 정렬성을 유지시킨다. 정렬 후 ready queue에 가장 앞에 있는 thread priority가 running thread보다 큰 경우 preemptive 정책에 의해 intr\_yield\_on\_return을 통해 ready queue로 내보내고 re-scheduling을 진행한다. 이때 intr\_yield\_on\_return을 호출한 이유는 update 함수를 호출한 것이 timer\_interrupt 이기 때문이다.
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부





priority-lifo는 priority가 오름차순인 16개의 thread를 순차적으로 삽입한다. Priority Scheduling 기법에 따라 높은 우선순위의 thread가 생성되면 낮은 우선순위를 가지고 있는 thread를 내보내고 높은 우선순위의 thread를 실행시켜야 한다. 따라서 이번 testcase에서는 가장 늦게 들어온 thread가 가장 높은 우선순위를 가지고 있기 때문에 가장 먼저 실행이 종료된다.