**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20171609 김민식

개발 기간 : 11/14 ~ 12/7

1. **개발 목표**

기존 thread system에서 alarm clock을 개선하고, 기존의 scheduling방식을 각 thread에 priority를 고려하는 scheduling으로 수정한다. 또한 aging과 BSD 스케쥴러를 구현하며, 최종적으로 thread의 동작을 효율적으로 관리하는 방법을 이해한다.

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

**-pintos/src/devices/timer.c**

Project3 이전의 기존 timer\_sleep()의 경우에는 busy waiting기법을 기반으로 구현이 되어있다. Timer\_sleep()함수에서 기다리는 시간동안 while문을 이용해 반복적으로 일을 처리하기 때문에 필요 이상으로 CPU 자원을 낭비하게 된다.

따라서 CPU 자원을 낭비하지 않도록 context switching 방식이 첨가된 preemption 방식을 구현한다. while문을 반복적으로 수행하며 시간을 체크하는 대신 시간 체크 후 설정된 시간이 되지않으면 thread를 block 상태로 만들고, wake up시간이 되었을 때 ready 상태로 만들어 ready list에 push하는 방법을 사용하였다.

* 1. **Priority Scheduling**

**- Pintos/src/threads/thread.c**

**- Pintos/src/devices/timer.c**

기존에는 새로운 thread가 입력되면 ready list의 끝에 push되는 구조로 구현되어 있었다. 하지만 이 scheduling 방법을 사용하면 starvation이 발생한다. 따라서 각 thread마다 priority를 부여하여 scheduling이 priority에 의해 수행이 되도록 수정한다. ready list에 머무는 시간에 비례하여 priority를 증가시키는 Aging 기법도 구현한다.

1. **BSD Scheduling**

**- Pintos/src/threads/thread.c**

기존에는 scheduling이 FIFO 방식으로 구현되어 있었다. 이 스케쥴링 방식을 서로 General-Purpose Scheduler로 변경한다. 이를 일반적으로 MLFQS(Multi-Level feedback Queue Scheduler)이라 한다. MLFQS는 우선순위별로 ready queue가 있고, 가장 높은 우선순위의 ready queue에서의 thread가 먼저 선택된다. 같은 우선순위 내에서는 round-robin방식으로 Scheduling된다.

**4) Fixed Point real Arithmetic**

Pintos Kernel에서는 Floating Point 연산을 지원하지 않는다. 하지만 BSD 스케쥴러 구현에서는 Floating Point 연산이 필요하기 때문에 실수연산이 가능하도록 Fixed Point 형식을 사용한다.

* 1. **개발 내용**
     1. **Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술**

Blocked상태인 thread를 깨우기 위해서는 time interrupt가 필요하다.

먼저 thread를 blocked 상태로 만들 때 ready list에 wake up time을 함께 저장해 준다. 이후 time interrypt가 일어날 때 timer\_interrupt 함수에서 wake up할 thread가 있는지 주기적으로 확인한다.

만약 wake up 할 thread가 있다면 thread를 unblock 해주고, block list에서 thread를 제거한다.

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

높은 priority를 가지고 있는 thread가 생성되는 경우에 생성되는 thread의 priority가 running thread의 priority보다 더 높으면 thread yield() 함수를 수행한다. 이 함수는 현재 running 중인 thread를 ready list에 push하고, 다시 스케쥴링 하는 함수이기 때문에 priority가 높은 create 된 thread가 running 상태로 된다.

또한 running thread의 priority가 변해서 ready list의 thread의 priority보다 작아지는 경우에도 thread yield()함수를 실핸하여 running중인 thread를 ready list에 push하고 다시 스케쥴링 한다.

* + 1. **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술**

Load\_avg : 매 초 마다 계산 되는 ready 상태의 thread 수의 평균

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

이때 ready\_threads는 ready state 뿐만 아니라 running state에 있는 thread의 개수를 의미한다.

Nice : thread는 priority와 관련된 nice value를 가진다. Nice의 범위는 -20 ~ 20이다.

Recent\_cpu : thread의 최근 CPU Time이며 식으로는 아래와 같이 나타낼 수 있다.

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

Priority는 위의 load\_avg, recent\_cpu를 사용하여 구할 수 있으며, priority를 식으로 나타내면 아래와 같다.

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

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

11/14 ~ 11/21 : 매뉴얼 분석 및 자료구조 및 알고리즘 설계

11/22 ~ 12/5 : Alarm Clock 구현, Priority Scheduling, Bsd Schdeler 구현

12/6 ~ 12/7 : 테스트 및 보고서 작성

* 1. **개발 방법**
     + 1. **Alarm Clock**

Timer.c의 timer\_sleep() 함수에서는 현재 tick을 sleep\_timer\_inthread()함수의 parameter로 넘겨주면서 호출한다. 그 후 넘겨받은 tick을 thread에 넣어준 후 현재 thread를 block 시키는 thread\_block()을 호출한다. 그 전에 block된 thread들이 모여 있는 block\_list에 thread를 우선순위를 고려하여 넣어준다.

Block 상태가 된 thread list를 저장할 때 wake up 할 시간도 함께 저장한다. Wake up 할 때는 Timer interrupt 함수를 호출하며 wake up할 thread가 있는지 확인한다. 그 후 Wake up할 때가 된 thread는 ready queue에 삽입해준다. 삽입 할때는 우선순위에 근거하여 우선순위대로 삽입한다.

* + - 1. **Priority Scheduling**

프로젝트3에서는 Ready list나 block list에 삽입 시 priority에 기반하여 삽입하도록 한다. 이를 위해서 list\_insert\_ordered()함수를 사용한다. 또한 ready list 안에서 우선순위가 낮은 thread는 기다리는 time에 따라 aging을 부여하여 starvation을 방지한다.

**<threads/synch.c>**

- sema\_up() : sema\_up을 할 때 우선순위를 정렬시켜준다.

- sema\_down() : thread를 priority에 따라 정렬하여 삽입해준다.

**<threads/thread.c>**

* thread\_yield() : thread의 우선순위가 최고로 큰 우선순위가 아니면 다시 ready\_list에 넣어주는 함수이다.
* thread\_create() : 만약 새 thread의 우선순위가 수행되고 있는 thread보다 그면 다시 스케쥴링 해야하므로 thread\_yield()를 호출해주는 함수이다.
* thread\_set\_priority() : 새로운 thread가 들어오면 우선순위를 현재 thread의 priority에 넣고, 만약 새로운 priority가 최대 priority보다 작아지면 다시 스케쥴링하는 함수이다.
* Thread\_prior\_aging() : aging이 기능이 필요한 test를 확인하기 위한 함수이다.
* Thread\_unblock() : thread의 상태가 block이면, running 상태로 변경해주는 함수이다.
  + - 1. **BSD Scheduler**

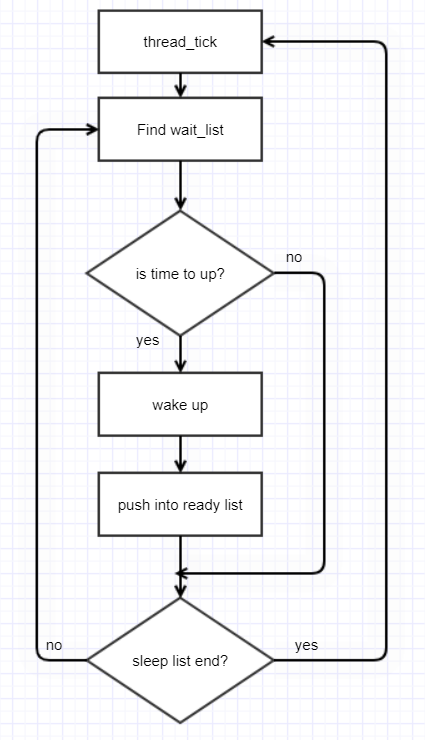
**<threads/thread.c>**

* thread\_init() : 새롭게 선언해준 변수들을 초기화 해준다.
* init\_thread() : parameter로 들어오는 thread의 recent\_cpu에 현재 실행중인 thread의 recent\_cpu 값을, thread의 nice에 현재 실행중인 thread의 nice 값을 저장한다.
* thread\_set\_priority() : 만약 thread\_mlfqs 플래그가 true라면 해당 함수를 수행하지 않도록 설정한다.
* thread\_set\_nice() : 현재 수행중인 thread의 nice값을 설정해주고, 이 값을 기반으로 priority를 설정해준다.
* Thread\_get\_load\_avg() : 함수에서 100 \* load\_avg를 return한다
* Thread\_get\_nice() : running thread의 nice값을 리턴해주는 함수이다.

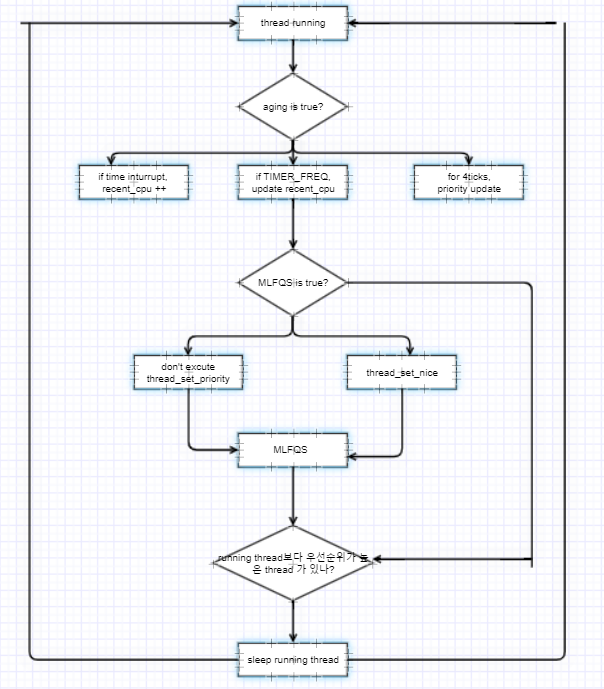
**<devices/timer.c>**

* timer\_interrupt() : thread\_prior\_aging 가 true면, time interrupt가 발생할 때마다, TIMER\_FREQ 마다 recent\_cpu 값을 1씩 더하서 업데이트 한다. 또한 4 ticks마다 우선순위를 다시 계산해주는 함수이다.

1. **연구 결과**
   1. **Flow Chart**
      1. **Alarm Clock**

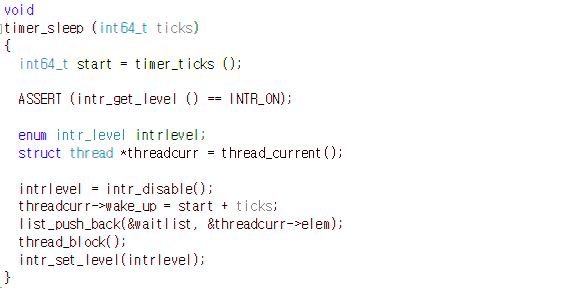


* + 1. **Priority Scheduling**



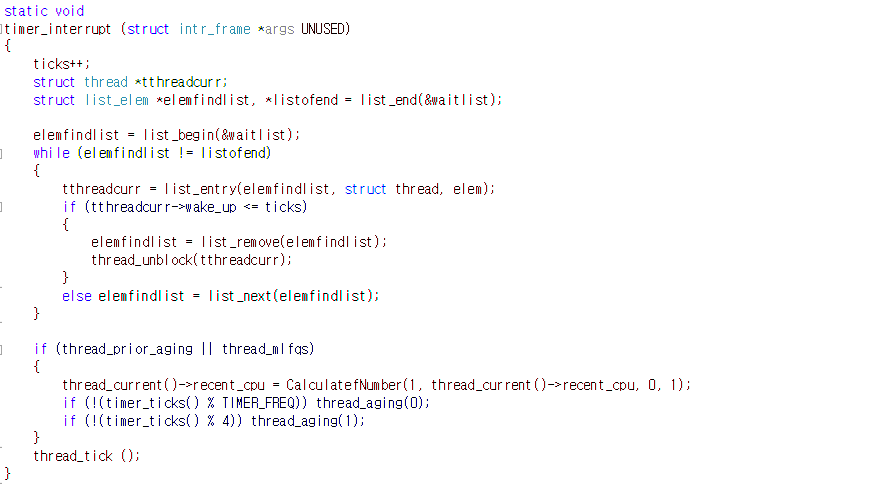
* 1. **제작 내용**
     + 1. **Alarm Clock**

**<timer.c / void timer\_sleep()>**



기존 while문으로 대기하고 있는 방법 대신 thread 별로 처음부터 메모리 상에 존재한 시간만큼 저장시켜준다. 이후 waitlist에 그 thread를 push\_back하고, block을 실시한다.

**<timer.c / void timer\_interrupt()>**

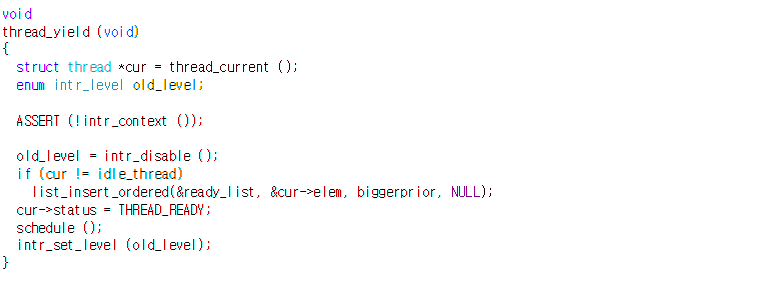


Wait list에 있는 thread를 검색하면서 tick보다 작은 thread를 unblock 시켜준다. 또한 thread\_prior\_aging이나 thread\_mlfqs 변수가 true라면 현재 thread의 recent\_cpu를 새로 계산한다.

이후 tick 값이 timer를 호출한 횟수의 배수이거나, 4의 배수이면 우선순위를 새로 계산해준다. 마지막으로 thread\_tick() 함수를 호출하여 tick 값을 더 계산한다.

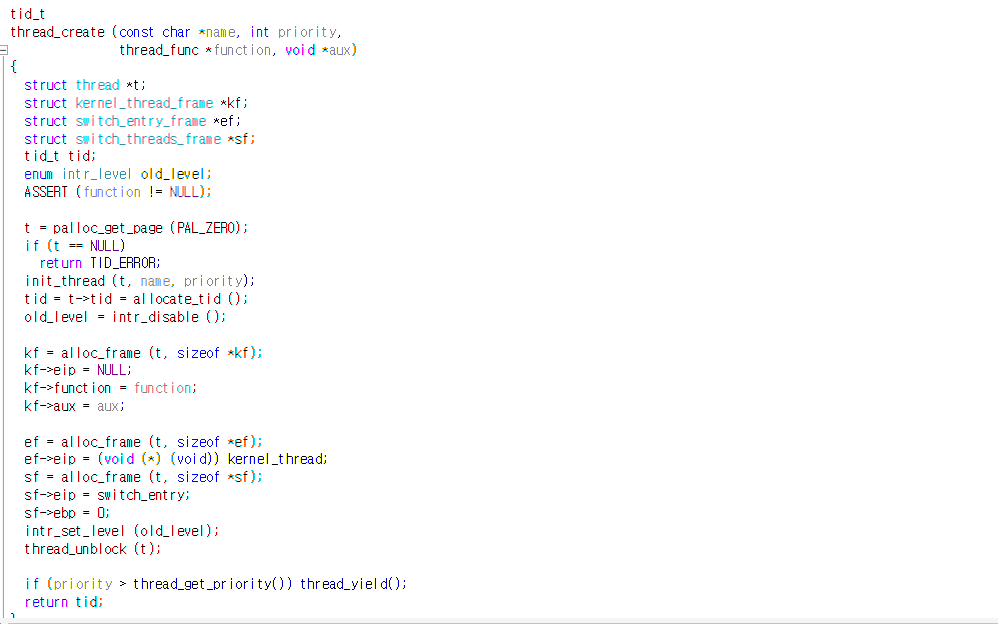
* + - 1. **Priority Scheduling**

**< thread.c / thread\_yield()>**



현재의 thread의 우선순위보다 새로 만든 thread의 우선순위가 더 높을 때 호출되는 함수이다. 현재 thread가 idle\_thread가 아닐 경우 우선순위에 따라 list\_insert\_ordered() 함수를 호출하여 ready queue에 삽입해주는 함수이다.

**< thread.c / thread\_create()>**



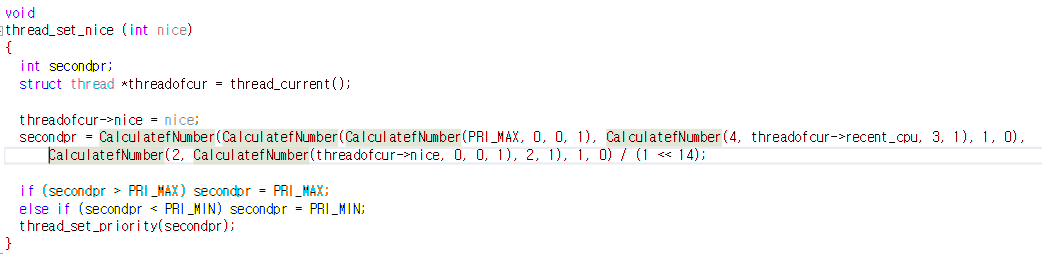
Thread가 만들어지면 thread\_yield()함수를 호출하여 thread를 ready list에 순서에 맞게 넣어주고 스케쥴링 할 수 있게 한다.

**< thread.c / thread\_set\_priority()>**



현재 thread의 우선순위를 set하는 함수로, 파라미터로 넘겨받은 우선순위가 현재 thread의 우선순위보다 작으면 입력받은 우선순위로 set해준 다음 thread\_yield함수를 호출한다.

**< thread.c / thread\_set\_nice()>**

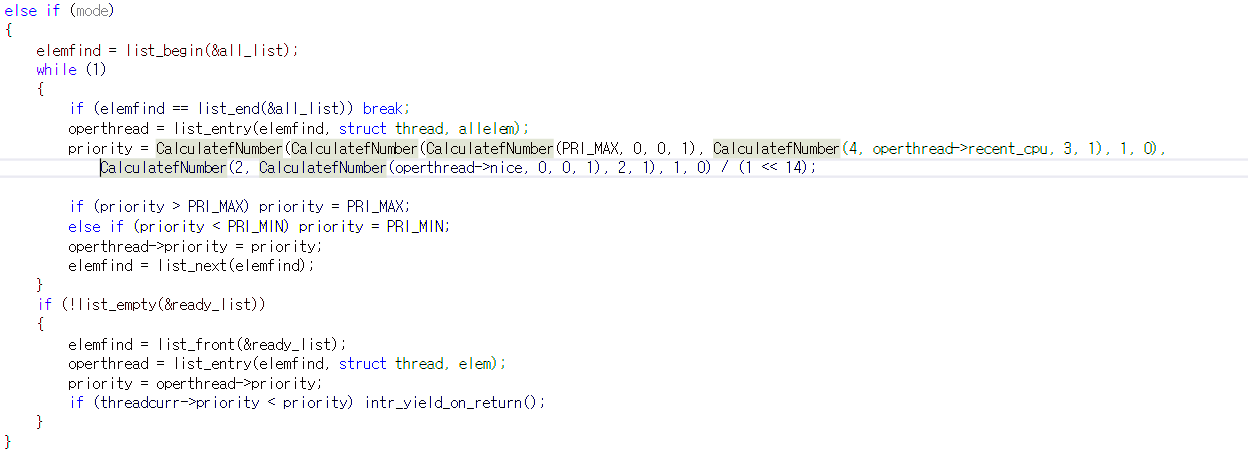


현재 thread의 nice값을 입력 받으면 우선순위를 다시 계산해야 한다. 새로 운 우선순위가 PRI\_MAX보다 클 경우에는 새 우선순위를 PRI\_MAX로 해주고 PRI\_MIN보다 작을 경우에는 PRI\_MIN으로 교체해준다.

이렇게 계산된 우선순위를 thread\_set\_priority의 인자로 넘겨주어 호출해주는 함수이다.

**< thread.c / thread\_aging()>**

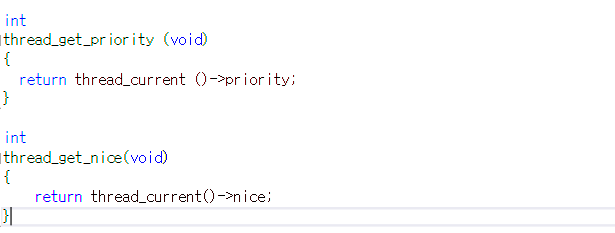




Readylist에 존재하는 모든 원소의 priority를 1씩 증가시키는 함수이다. Parameter로 mode 변수를 받는다. Mode가 0일 경우에는 매초마다 recent\_cpu를 업데이트하고, mode가 1일 때는 4초마다 priority를 업데이트 한다.

또한 ready list에 있는 thread를 탐색하며 우선순위가 정해진 범위를 벗어나지 않도록 해준다. 만약 현재 thread의 우선순위가 가장 높지 않다면 interrupt중 이기 때문에 intr\_yield\_on\_return을 호출해준다

**< thread.c / thread\_get\_nice(), thread\_get\_priority()>**



현재 thread의 nice와 priority 값을 리턴해주는 함수이다.

* + - 1. **BSD Scheduler**

**< thread.c / CalculatefNumber()>**

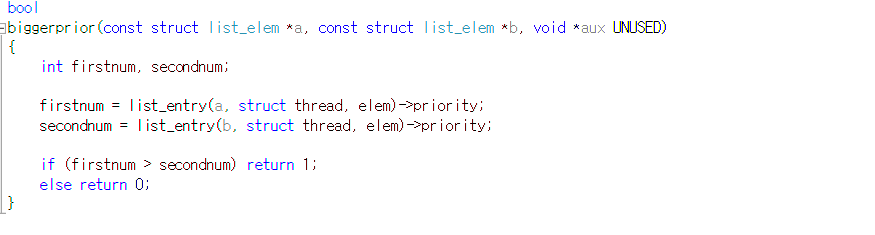


Float 연산을 위해 만들어진 함수이다.

실수와 실수끼리의 사칙연산, 실수와 정수 의 사직연산에 대하여 switch문을 통해 구현하였다. 실수 \* 실수, 실수 / 실수 연산의 경우 손실이 일어날 수 있기 때문에 int64\_t 임시변수를 두어 중간중간 저장해가며 손실을 줄였다.

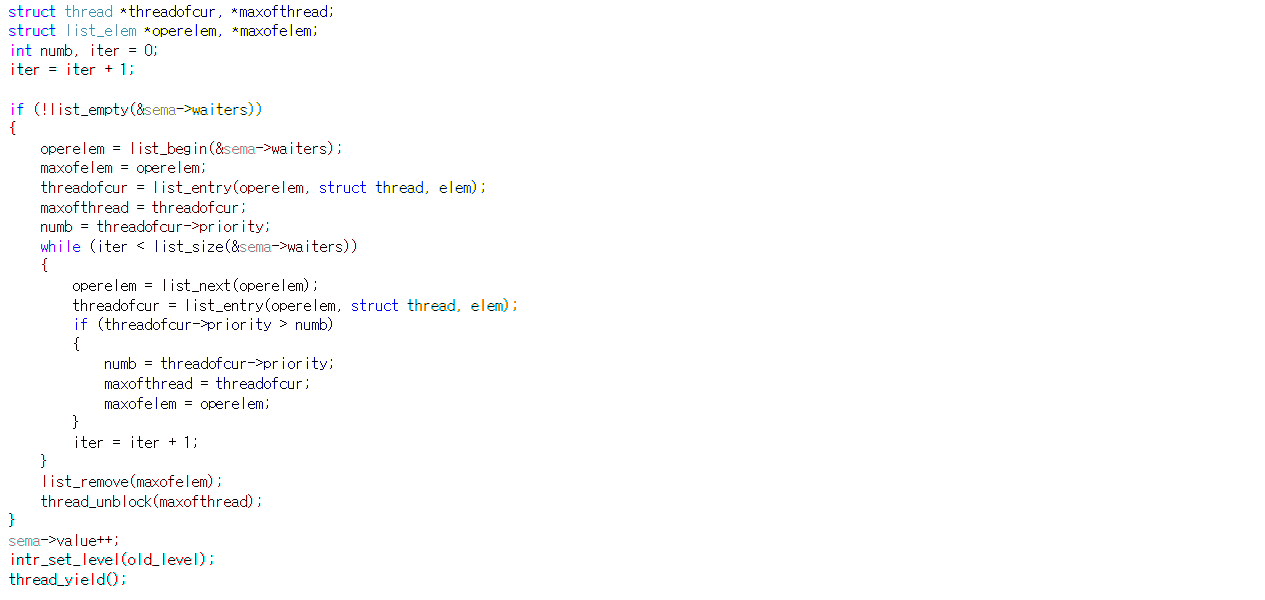
또한 pintos에서는 실수에 대한 표현, 출력을 제공하지 않기 때문에 비트 연산을 통해 구한하였다.

**< thread.c / biggerprior()>**



List\_insert\_ordered를 호출할 때 우선순위에 따라 list를 정렬할 수 있도록하는 함수이다. 두개의 list elem을 인자로 받아 첫번째 elem의 priority가 두번째 elem의 priority보다 크면 1 아니면 0을 반환한다.

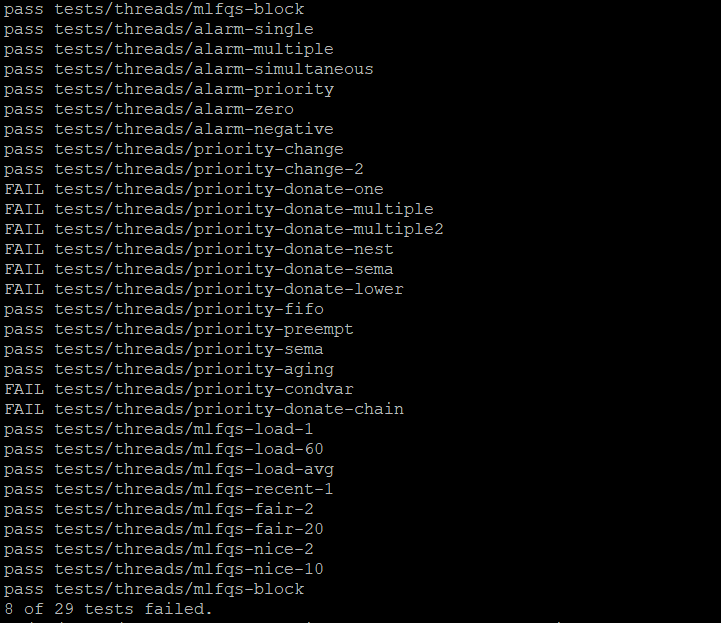
**< synch.c / sema\_up()>**



Sema\_up을 할 때 block된 thread를 unblock 시켜주는 함수이다. 이 때 새로운 thread가 ready list에 추가된 상황과 같은 상황이므로 thread\_yield()함수를 호출하여 스케쥴링 한다.

* 1. **시험 및 평가 내용**

**<make check>**



**<priority-lifo.c 코드 및 priority-lifo테스트 결과 분석>**

반복문을 통해 총 16개의 thread를 만든다. 만들어진 thread의 priority는 늦게 만들어진 thread일수록 우선순위 또한 높다. 이때 priority는 PRI\_DEFAULT + i(0 <= I <= THREAD\_CNT) +1 이다.

모두 만들어진 thread가 실행될 수 있도록 running thread의 우선순위를 PRI\_DEFAULT(31)로 설정한다. 그 뒤에 생성된 thread는 모두 PRI\_DEFAULT 보다 높은 우선순위를 가지기 때문에 순차적으로 실행된다.

아래 그림은 pintos에서 priority-lifo를 실행하였을 때 결과를 캡쳐한 사진이다.

그림을 보면 나중에 만들어진 thread일수록 높은 priority를 가지게 되고, 나중에 ready list에 들어가도 먼저 수행이 된다는 것을 알 수 있다.

