**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

이름 / 학번 : 백인찬 / 20150195

개발 기간 : 2020.11.14 ~ 2020.12.08

1. **개발 목표**

현재 thread의 Alarm Clock은 busy waiting으로 구현되어 있고 Running 상태와 Ready 상태 사이를 계속 loop를 통해 순회하기 때문에 비효율적이다. 이를 non-busy waiting 방식으로 변경해 효율성을 얻는다.

또 현재 thread scheduling은 round-robin 알고리즘으로 구현되어 있는데 이를 priority scheduling 알고리즘으로 변경한다.

더불어 BSD scheduling 알고리즘 또한 구현해 multi-level feedback queue를 통해 schedule할 수 있도록 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

src/threads/ thread.h thread.c init.c sync.c

src/devices/ timer.c

* 1. Alarm Clock

현재 Alarm clock은 현재 thread가 어느 정도 시간이 흐를 때까지(tick이 증가할 때까지) loop에서 벗어나지 못하는 busy waiting 방식으로 구현되어 있다. 따라서 tick이 일정 수준이 되었는지 확인하고, 그렇지 않다면 해당 thread를 Blocked 상태로 만드는 non-busy waiting을 통해 효율성을 높인다.

* 1. Priority Scheduling

round-robin으로 구현되어 있는 scheduling은 priority를 일체 고려하지 않은 상태이다. 이를 보완하여 각 thread에 priority를 부여해 높은 priority를 가진 thread가 CPU를 우선적으로 선점할 수 있도록 한다. 또 starvation 문제를 방지하기 위해 ready queue 내에서 일정 시간이 지날 때마다 priority를 증가시키는 priority aging을 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

BSD Scheduling 알고리즘을 통해 고유의 priority를 가진 Multi-level feedback queue를 두고 해당 queue 내에선 round-robin으로 schedule하여 더욱 범용적인 Scheduler를 구현한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

잠들어 있는, Blocked 상태인 thread를 별도의 queue에 저장하고 timer interrupt가 발생해 tick이 증가한 경우 해당 queue를 순회하며 깨어날 때가 된 thread를 깨운다.

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

각 thread에는 priority가 존재하는데, 이를 이용해 priority가 높은 thread가 들어온다면 CPU를 먼저 선점해 할당 받을 수 있도록 한다. 이를 구현하기 위해 ready list에 thread를 삽입할 때 priority를 비교하는 함수 포인터를 이용해 정렬된 상태로 삽입할 수 있도록 한다.

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

Priority를 계산하기 위해서는 어떤 thread가 얼마나 점유했는지를 나타내는 recent\_cpu의 값과 priority를 낮춰 다른 thread가 CPU를 점유할 수 있도록 하는 nice값이 필요하다. 또 recent\_cpu의 값을 계산하기 위해서는 최근 1분 동안 수행 가능한 프로세스의 개수를 나타내는 load\_average의 값이 필요하다.

또 핀토스는 실수의 계산을 지원하지 않는다. 따라서 4 bytes의 정수를 14bits는 뒷부분의 소수부로 사용하기 위한 shift 연산이 필요하다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

11/14(토) ~ 11/27(금) : ppt 및 핀토스 매뉴얼 분석

11/28(토) ~ 12/01(화) : Alarm Clock 구현

12/02(수) ~ 12/04(금) : Priority Scheduling 구현

12/05(토) ~ 12/06(일) : BSD Scheduler 구현

12/07(월) ~ 12/08(화) : 디버깅 및 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

Block 상태인 thread를 별도로 관리하기 위해 sleeping\_threads라는 list를 선언하였다. timer interrupt를 통해서 tick이 증가할 경우 thread\_wake\_up()을 호출해 해당 tick에서 sleeping\_threads를 순회하며 깨어나야 할 thread가 존재하는지 검사하고, 존재한다면 해당 thread를 sleeping\_threads에서 제거하며 block 상태를 해제한다.

* Priority Scheduling

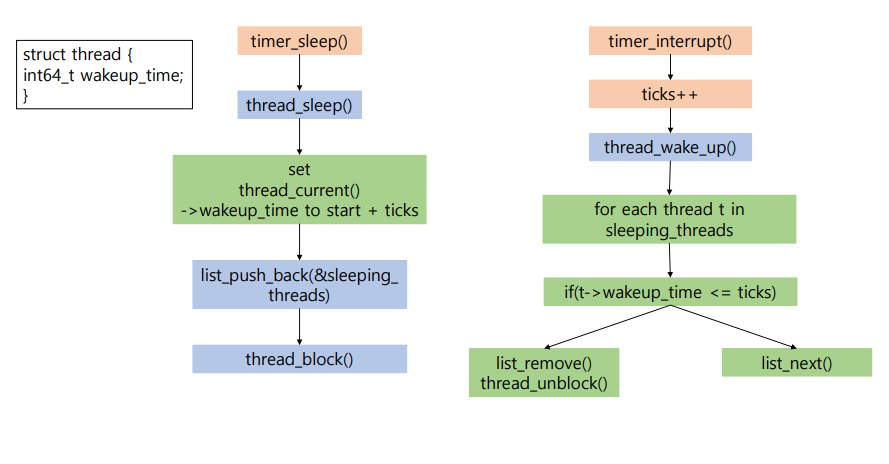
기존에 round-robin으로 구현되어 있던 scheduling에서는 thread를 ready list에 삽입할 때 list\_push\_back()을 통해 맨 뒤에 위치하도록 한 것을 list\_insert\_ordered()을 통해 정렬된 상태로 삽입될 수 있도록 한다. list\_insert\_ordered()의 정렬을 위한 함수 포인터로 prioirty를 비교하는 함수인 compare\_priority()를 함수 포인터로 넘겨준다. 정렬된 ready list에는 가장 높은 priority의 thread가 맨 앞에 위치하므로 맨 앞의 thread보다 삽입할 priority가 높을 경우 CPU를 선점할 수 있도록 한다.

* BSD Scheduler

BSD Scheduler를 구현하기 위해 timer\_interrupt 내에서 recent\_cpu를 증가시키고, 4 tick마다 priority를 재계산하고, 매초마다 load\_avg와 recent\_cpu를 재계산한다. 이를 위해 각 변수들을 재계산하는 함수들을 정의하여 사용한다. 또 BSD Scheduler를 통해 작동할 때에는 임의로 priority를 변경할 수 없도록 thread\_set\_priority()함수를 변경한다.

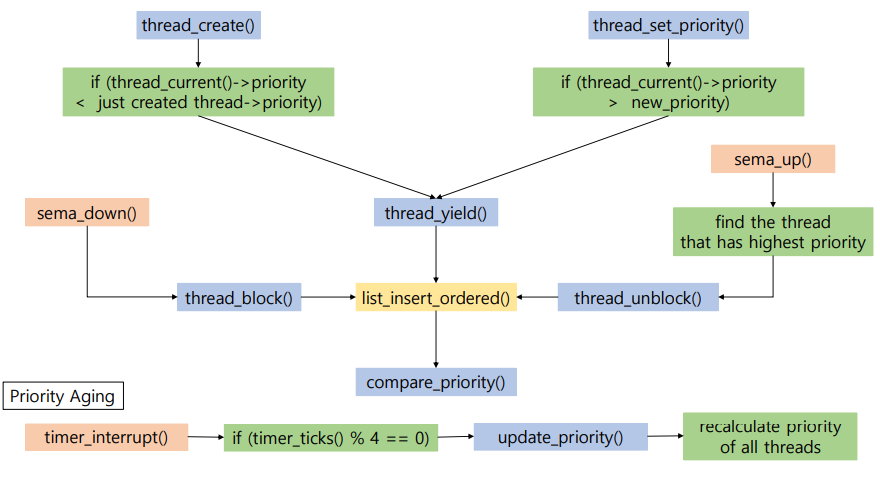
1. **연구 결과**
   1. **Flow Chart**

* Alarm Clock



timer\_sleep()에서 thread\_sleep()을 호출해 현재 thread의 wakeup\_time을 set한다. 이후 timer\_interrupt에서 ticks을 증가시킨 후 thread\_wake\_up()을 호출해 깨어날 thread가 있는지 검사하고 있다면 thread\_unblock()을 호출해 unblock상태를 해제한다.

* Priority Scheduling

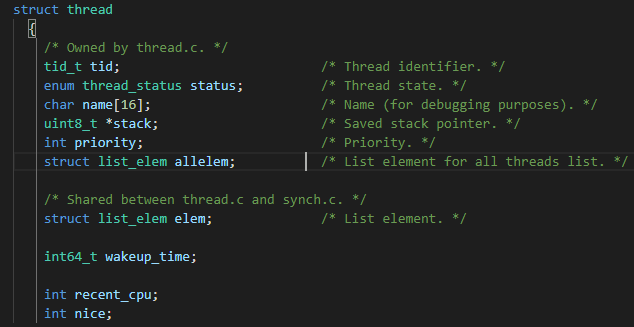


Priority aging을 통해 starvation 문제를 해결하기 위해 timer\_interrupt()에서 4 ticks마다 모든 thread의 priority를 update한다.

* 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

**/threads/thread.h**

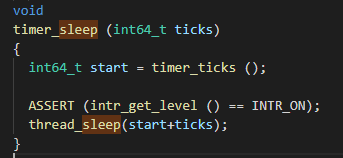
thread 구조체 내에 각 thread의 wakeup\_time을 설정하기 위한 멤버 변수를 추가하였다. ticks와의 비교를 위해 ticks와 동일한 type인 int64\_t로 선언하였다.



**/devices/timer.c**

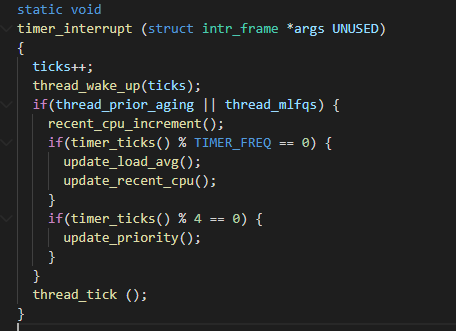
**void timer\_sleep(int64\_t ticks)**

현재 ticks와 parameter로 받은 ticks를 더한 값을 thread\_sleep()을 호출해 thread의 wakeup\_time으로 설정한다.



**static void timer\_interrupt(struct intr\_frame \*arg UNUSED)**

전역 변수인 ticks를 증가시키고 해당 ticks에 깨어나야 할 thread가 있는지 thread\_wake\_up()을 호출해 검사하고 깨운다.



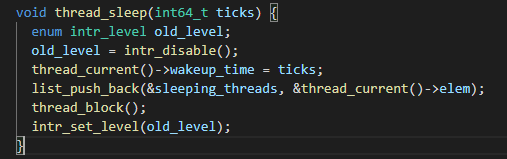
**/threads/thread.c**

**static struct list sleeping\_threads**

Blocked 상태인 thread를 한 곳에 모아 관리하기 위해 list를 선언하였다.

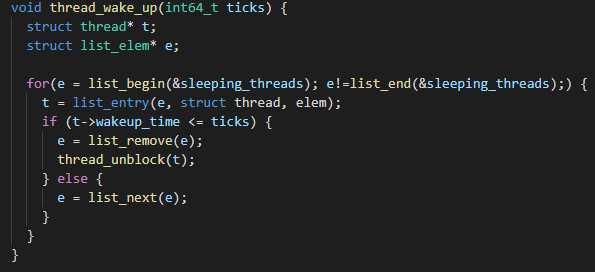
**void thread\_sleep(int64\_t ticks)**

timer\_sleep()로부터 parameter로 넘겨받은 ticks를 현재 thread의 wakeup\_time으로 설정하고 blocked 상태인 thread를 관리하기 위한 queue인 sleeping\_threads에 push\_back한다. 이후 thread\_block()을 호출해 현재 thread를 blocked 상태로 바꾼다.



**void thread\_wake\_up(int64\_t ticks)**

timer\_interrupt()로부터 parameter로 넘겨받은 ticks를 이용해 sleeping\_threads를 순회하며 thread의 wakeup\_time이 지났을 경우(wakeup\_time <= ticks), 해당 thread를 sleeping\_threads에서 제거하고 thread\_unblock()을 수행해 다시 ready list에 넣는다.



* Priority Scheduling

**/threads/thread.h**

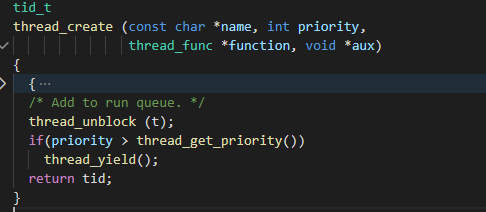
**#define SHIFT (1 << 14)**

핀토스는 실수의 계산을 허용하고 있지 않기 때문에 실수의 계산을 위해 4 bytes(32 bits)인 정수를 소수점으로 쪼갠다고 가정하기 위해 SHIFT 매크로를 선언했다. 핀토스에서는 최상위 비트를 부호 비트로, LSB부터 14개의 비트를 소수부로 나누었기 때문에 14비트만큼 왼쪽으로 shift 연산을 한다.

**/threads/thread.c**

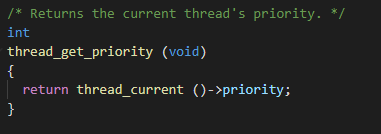
**tid\_t thread\_create(const char\* name, int priority, thread\_func \*function, void \*aux)**

thread를 생성할 때 생성된 thread의 priority가 현재 thread의 priority보다 크다면, thread\_yield를 호출해 CPU를 preempt할 수 있도록 한다.



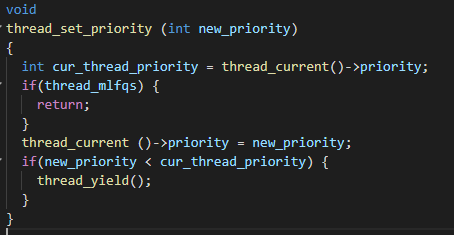
**int thread\_get\_priority(void)**

현재 thread의 priority를 반환한다.



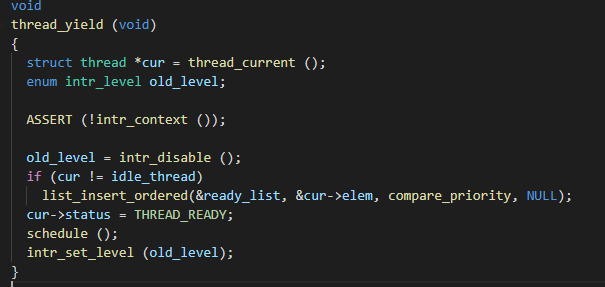
**void thread\_set\_priority(int new\_priority)**

현재 thread의 priority를 new\_priority로 설정한다. 새롭게 설정된 priority가 기존의 priority보다 낮다면 thread\_yield()를 호출해 ready\_list를 다시 schedule한다. 현재 thread의 기존 priority가 가장 높은 priority이므로 ready\_list에서 CPU를 할당 받았을 것이므로, 기존의 priority보다 높은 경우는 CPU를 다른 thread에게 넘겨줄 필요가 없다.



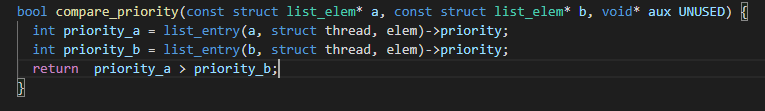
**void thread\_yield(void)**

기존에 list\_push\_back()으로 ready\_list에 맨 뒤에 삽입하던 것을 list\_insert\_ordered를 통해 priority를 기준으로 정렬하여 삽입하도록 한다. 이를 위해 compare\_priority()를 함수 포인터로 넘긴다.



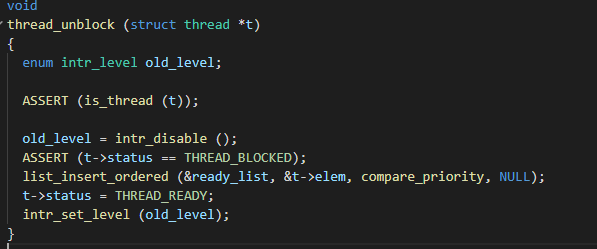
**bool compare\_priority(const struct list\_elem\* a, comst struct list\_elem\* b, void\* aux UNUSED)**

list\_insert\_ordered에서 사용하는 비교를 위한 함수로, a의 thread의 priority와 b의 thread의 priority를 비교한다.



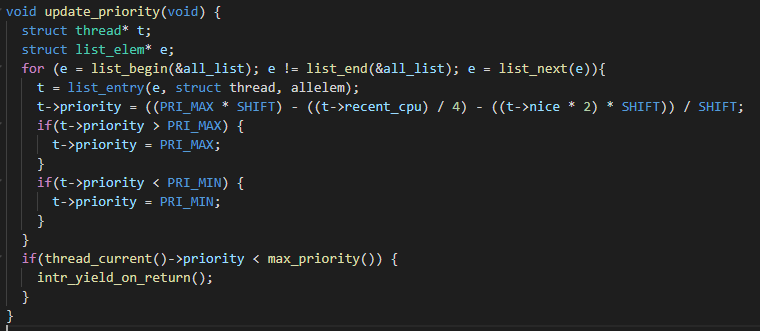
**void thread\_unblock(struct thread\* t)**

thread를 다시 ready\_list로 삽입하며, 해당 thread의 상태를 READY로 변경한다. thread\_yield()에서와 마찬가지로 compare\_priority를 비교 함수로 사용하여 list\_insert\_ordered()를 호출한다.



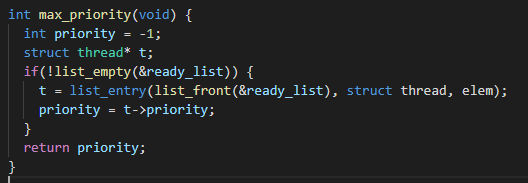
**void update\_priorty(void)**

모든 thread의 priority를 다시 계산한다. priority의 계산은 핀토스 매뉴얼에 안내된 priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)을 따른다. priority의 범위를 검사해 범위를 넘어가지 않도록 한다. update\_priority()는 interrupt 도중에 호출된 것이므로 intr\_yield\_on\_return()을 호출해 다시 schedule한다.



**int max\_priority(void)**

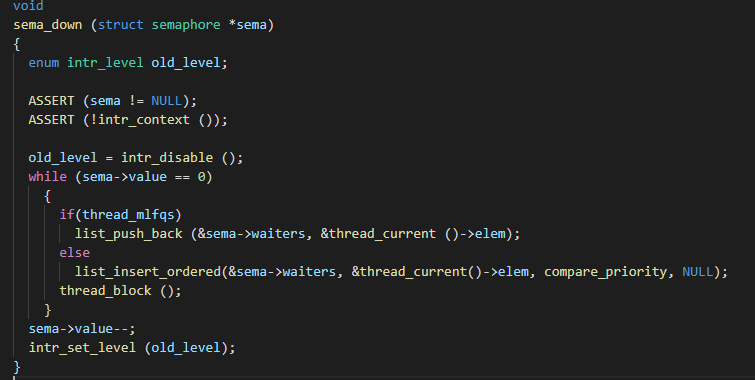
ready\_list의 가장 첫 thread는 가장 높은 priority를 갖고 있을 것이므로 첫번째 thread의 priority를 반환한다. 만약 ready\_list가 비었다면 -1을 반환한다.



**/threads/synch.c**

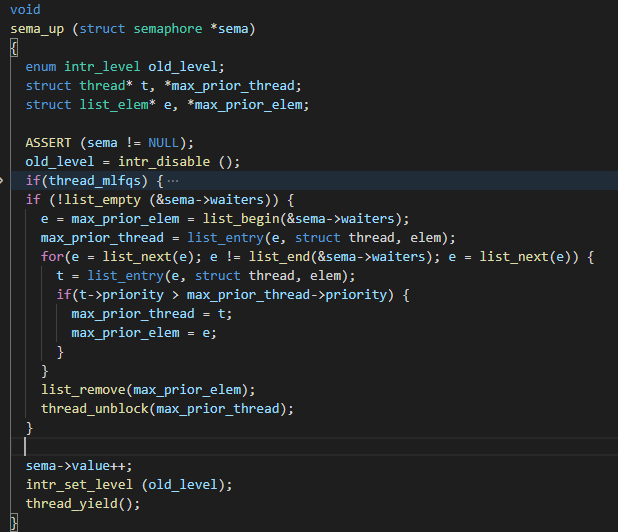
**void sema\_down(struct semaphore \*sema)**

sema\_down으로 Blocked 상태가 되는 thread들도 priority를 기준으로 정렬하기 위해 waiters에 삽입할 때에도 list\_insert\_ordered를 통해 삽입하도록 한다.



**void sema\_up(struct semaphore \*sema)**

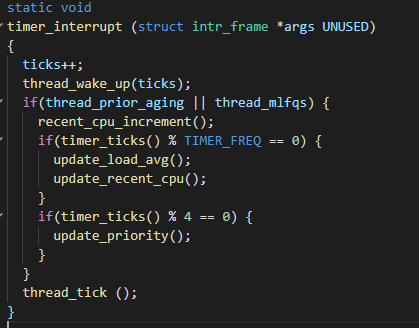
sema\_down()을 통해 이미 정렬된 상태로 waiters에 삽입되었지만 오류를 최소화하기 위해 sema\_up()에서도 waiters를 순회하며 가장 높은 priority를 가진 thread를 골라 waiters에서 제거하며 thread\_unblock()을 통해 Blocked 상태를 해제한다.



**/devices/timer.c**

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

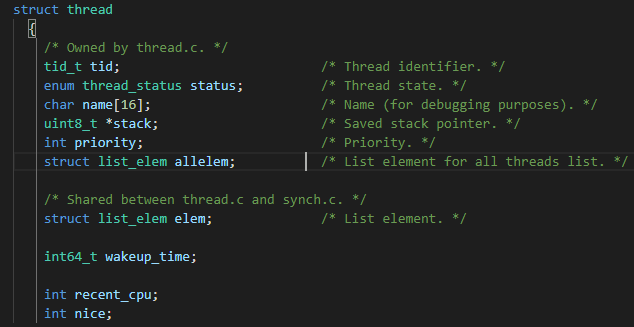
thread.c에 선언된 전역 변수인 thread\_prior\_aging이 true라면 4 ticks마다 update\_priority()를 호출해 thread의 priorty를 재계산할 수 있도록 하여 priority aging을 구현하며 starvation 문제를 해결하도록 한다.



* BSD Scheduler

**/threads/thread.h**

priority 계산에 필요한 recent\_cpu와 nice를 thread 구조체 내에 추가하였다.



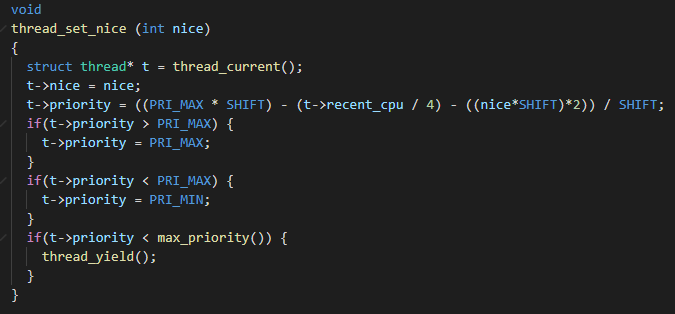
**/threads/thread.c**

load\_avg를 전역변수로 선언하고 thread\_init에서 0으로 초기화한다.



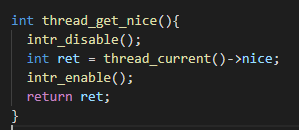
**void thread\_set\_nice(int nice)**

현재 thread의 nice를 parameter로 받은 nice로 설정한다. nice가 새로 설정되었기 때문에 priority도 다시 계산한다. priority가 새로이 설정되었으므로 thread\_yield()를 호출해 다시 schedule할 수 있도록 한다.



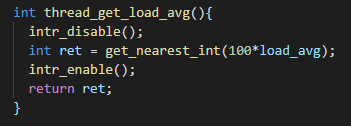
**int thread\_get\_nice()**

현재 thread의 nice를 반환한다. 이 함수가 호출되는 도중 interrupt는 비활성화한다.



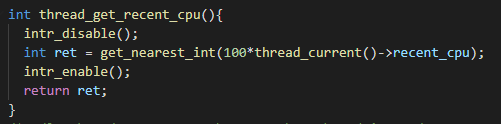
**int thread\_get\_load\_avg()**

load\_avg를 100을 곱하여 반환한다. load\_avg가 온전한 정수가 아닐 수 있으므로 가장 get\_nearest\_int()를 호출해 가장 근접한 정수를 반환한다. 이 함수가 호출되는 도중 interrupt는 비활성화한다.



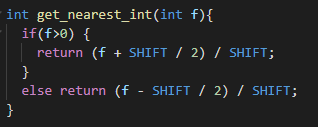
**int thread\_get\_recent\_cpu()**

현재 thread의 recent\_cpu를 100을 곱하여 반환한다. recent\_cpu가 온전한 정수가 아닐 수 있으므로 가장 get\_nearest\_int()를 호출해 가장 근접한 정수를 반환한다. 이 함수가 호출되는 도중 interrupt는 비활성화한다.



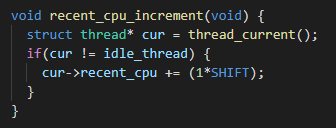
**int get\_nearest\_int(int f)**

parameter로 들어온 f를 반올림하여 정수로 바꾸어 반환한다.



**void recent\_cpu\_increment(void)**

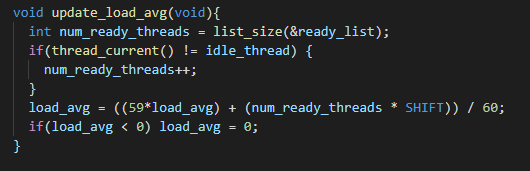
현재 thread의 recent\_cpu를 1만큼 증가 시킨다.



**void update\_load\_avg(void)**

load\_avg를 새로 계산한다. load\_avg의 계산은 핀토스 매뉴얼에 안내된

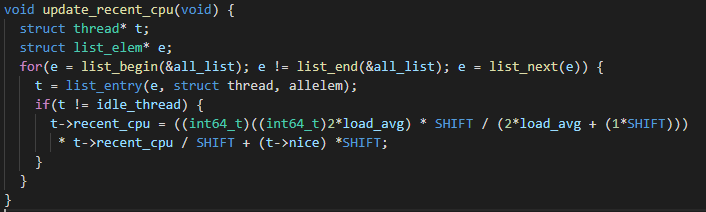
load\_avg = (59/60)\*load\_avg + (1/60)\*ready\_threads 계산식을 따른다. load\_avg는 0보다 작을 수 없으므로 0보다 작다면 0으로 설정한다.



**void update\_recent\_cpu(void)**

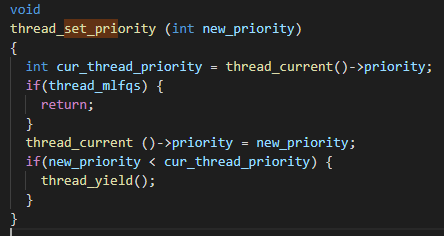
모든 thread를 순회하며 idle\_thread가 아니라면 해당 thread의 recent\_cpu를 재계산한다. recent\_cpu의 계산은 핀토스 매뉴얼에 안내된

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



**void thread\_set\_priority(int new\_priority)**

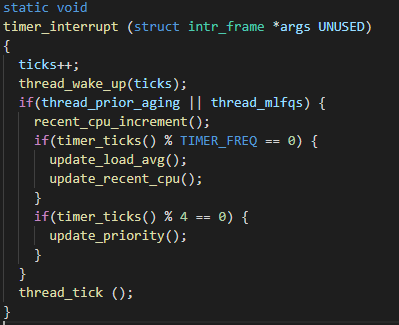
BSD scheduler로 동작한다는 것을 의미하는 thread\_mlfqs가 true라면 임의로 priority를 설정할 수 없도록 한다.



**/devices/timer.c**

**static void timer\_interrup(struct intr\_frame \*args UNUSED)**

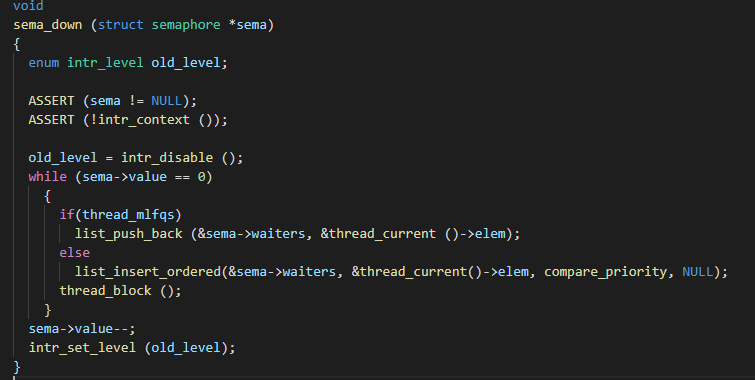
thread\_mlfqs가 true라면 매 ticks마다 recent\_cpu\_increment()를 호출해 recent\_cpu를 1만큼 증가시킨다. 또 매초마다(TIMER\_FREQ) update\_load\_avg()와 update\_recent\_cpu()를 호출해 load\_avg와 recent\_cpu를 재계산한다.



**/threads/synch.c**

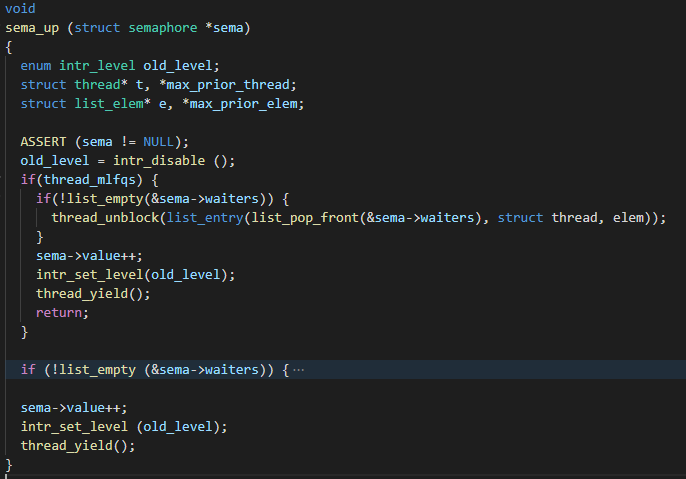
**void sema\_down(struct semaphore \*sema)**

BSD scheduler로 동작할 경우(thread\_mlfqs가 true) priority scheduling으로 동작하는 것이 아니므로 list\_push\_back()을 통해 waiters에 삽입한다.



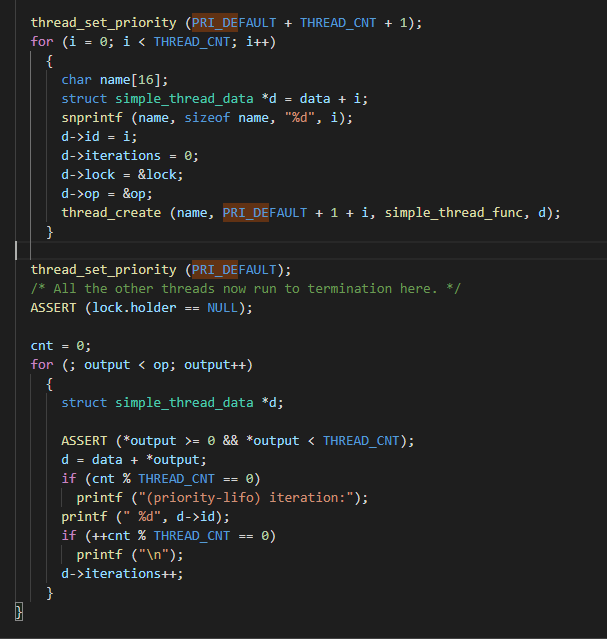
**void sema\_up(struct semaphore \*sema)**

thread\_mlfqs가 true인 경우 waiters의 가장 첫 thread부터 unblock한다.

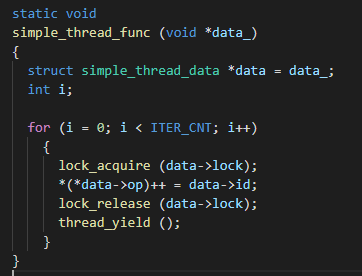


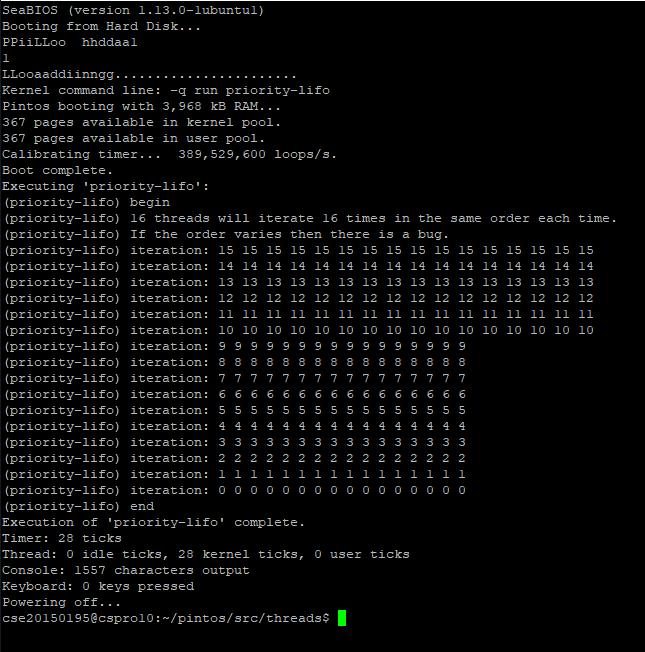
* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

아래의 그림은 priority-lifo.c의 test\_priority\_lifo()의 일부이다. 먼저 현재의 thread의 priority를 PRI\_DEFAULT(31) + THREAD\_CNT(16)+1으로 set하여 이후에 생성하는 thread의 priority보다 낮아져 yield되는 일이 없도록 한다. 이후 for문에서 THREAD\_CNT(16)만큼 반복하며 thread를 생성하고 해당 thread의 priority를 PRI\_DEFAULT + 1 + i(index)로 설정한다.



thread\_create()에 아래의 simple\_thread\_func을 함수포인터로 넘겨 output buffer를 ITER\_CNT(16)만큼 해당 thread의 id로 설정한다. 즉, 16개의 thread는 각각 id가 0부터 15까지의 값을 갖게 되며 priority 또한 각각 32부터 47까지의 값을 갖게 된다. Priority Scheduling에 의해 ready list 내에서의 thread의 순서도 id가 15, 14, 13, ,,, 1, 0 의 역순으로 정렬되며, output buffer에 설정된 thread의 id도 역순이 된다.





위의 그림은 priority-lifo의 출력 결과이다. test\_priority\_lifo()의 두번째 for문에서 output buffer를 끝까지 순회하며 해당 id를 출력하는데 위의 정렬된 순서로 인하여 15, 14, 13,, 1의 id가 각 16번씩 출력된다. 이러한 특징은 Last-In First-Out(LIFO)를 나타낸다.

* make check 수행 결과를 캡처하여 첨부

