**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 20181617 김채연

개발 기간 : 2020.11.14~2020.12.8

1. **개발 목표**

이전 프로젝트 2에서 구현한 프로그램에 추가로 thread와 관련된 기능을 구현한다. alarm clock, priority scheduling, advanced scheduler을 구현하는 것을 목표로 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

이전 프로젝트 2에서 구현한 프로그램에 추가로 thread와 관련된 기능을 구현한다. alarm clock, priority scheduling, advanced scheduler을 구현하는 것을 목표로 한다.

1. Alarm Clock

thread가 얼마 동안 작동을 하지 않을 때 timer\_sleep이라는 함수를 이용하여 sleep 시킨다. 이때 특정 시간이 지나지 않았을 때 while문을 이용하여 thread\_yield() 함수를 계속 호출하는데 thread가 running과 ready상태를 계속해서 반복하게 되므로 효율적이지 않다. 따라서 이러한 비효율적인 과정을 새로운 방법을 통해 해결하도록 한다.

1. Priority Scheduling

기존 핀토스에서는 round-robin 스케쥴링을 사용하였다. 따라서 thread\_yield()나 thread\_unblock()함수가 호출되었을 때 현재 thread나 unblocked thread는 우선순위에 관계없이 ready list의 가장 끝에 insert 되었다. 이번 프로젝트에서 priority Scheduling을 새롭게 구현하여 우선순위를 고려하여 삽입되도록 한다. 즉 현재 thread보다 우선 순위가 높은 새로운 thread가 있다면 해당 thread를 실행된다.

또한 priority scheduling은 낮은 우선순위의 경우 starvation이 생기므로 이를 방지하기 위해 aging까지 고려를 한다. 따라서 시간이 지남에 따라 우선순위를 점차 높아지도록 한다.

1. Advanced Scheduler

BSD 스케쥴러를 구현한다. multi-level feedback/ready queue를 사용하여 각 우선순위마다 자신의 ready queue를 가지게 한다. schedule()함수가 호출이 되면 priority가 가장 높은 queue에서 thread가 선택이 되도록 한다. 이 때 각 priority의 ready queue는 round-robin을 따르도록 한다. nice, priority, recent-cpu, load-avg 값을 사용하여 priority를 계산할 수 있다.

* 1. **개발 내용**

1. Blocked 상태의 thread를 깨울 수 있는 방법

wake up time이 아직 오지 않은 thread는 blocked 상태이다. 이런 blocked 상태의 thread를 관리하기 위해 list를 새로 만들어서 해당 thread를 모두 저장한다. 그다음 매 tick 마다 이 list에 있는 thread를 검사하며 wake up time이 된 thread를 list에서 제거를 하고 상태를 unblock으로 바꿔주면서 ready queue에 넣으면서 깨워준다.

1. Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따른 처리 방법

ready list에 높은 priority를 가진 thread가 들어오면 해당 thread가 실행되어야 한다. 따라서 먼저 현재 thread를 ready 상태로 바꾸고 ready list에 priority 순서대로 정렬되도록 running thread를 넣는다. 그다음 ready list에서 가장 priority가 높은 thread를 실행시킨다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소

nice, recent\_cpu, load\_avg가 필요하다.

먼저 thread가 처음 만들어지면 nice와 recent\_cpu값이 0으로 초기화된다. 자식 thread인 경우에는 부모 thread의 값을 받는다. 이때 nice는 -20에서 20까지의 값을 가진다. positive한 nice값은 priority를 낮춰주기 때문에 따라서 nice 값이 새롭게 변경이 된다면 thread의 priority를 다시 계산해주어야 한다.

recent\_cpu는 thread의 CPU time을 추정한다. time interrupt가 일어나면 running상태의 thread의 recent\_cpu 값은 1이 증가하도록 하고 매초마다 running, ready, blocked의 모든 thread의 recent\_cpu가 다시 계산되도록 한다. 그 식은 아래와 같다.

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

그리고 load\_avg는 전역변수이고 ready상태의 thread의 수 평균을 추정한다. 시스템이 실행되면 0으로 초기화 되어있다. load\_avg는 매초마다 update되고 그 식은 아래와 같다. (ready\_threads는 ready나 running상태의 thread 수이다.)

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

이를 통해 priority는 아래와 같이 계산이 된다. (PRI\_MAX는 priority의 maximum인 63이다.)

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

이때 계산은 소수점 연산인데 pintos에서는 floating-point 연산이 불가능 하므로 fixed point 연산으로 구현한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2020.11.14~2020.12.08

11/14 ~ 11/21 alarm clock 구현

11/22 ~ 11/28 priority scheduling 구현

11/29 ~ 12/6 advanced scheduling 구현

12/07 ~ 12/8 오류 수정 및 보고서 작성

* 1. **개발 방법**

timer.c에서 먼저 sleep\_list를 새로 선언하여 block된 list를 저장하도록 한다. 그다음busy waiting을 하는 timer\_sleep 함수의 코드를 수정한다. thread의 wakeuptime을 계산하고 앞에서 선언한 sleep\_list를 이용하여 block 상태의 thread를 관리한다. 그리고 timer\_interrupt에서 wakeuptime이 된 thread를 찾아서 깨워주는 코드를 추가한다.

thread.h, thread.c에서 thread구조체에 wakeuptime을 추가로 선언해준다. 그리고 thread\_yield와 thread\_unblock 함수에서 thread를 ready queue에 넣고 스케쥴링을 하므로 list를 priority 순서대로 삽입되도록 코드를 수정한다. 이때 사용할 함수로 priority를 비교하는 함수를 새로 만든다.그리고 thread\_create 함수에서 현재 수행중인 priority와 새로 생성되는 thread의 priority를 비교하여 새로 생성되는 thread가 더 크면 새롭게 스케쥴링을 하도록 코드를 추가한다. 마지막으로 thread\_set\_priority 함수에서 새로운 priority를 thread에 넣고 다시 스케쥴링하도록 코드를 추가한다.

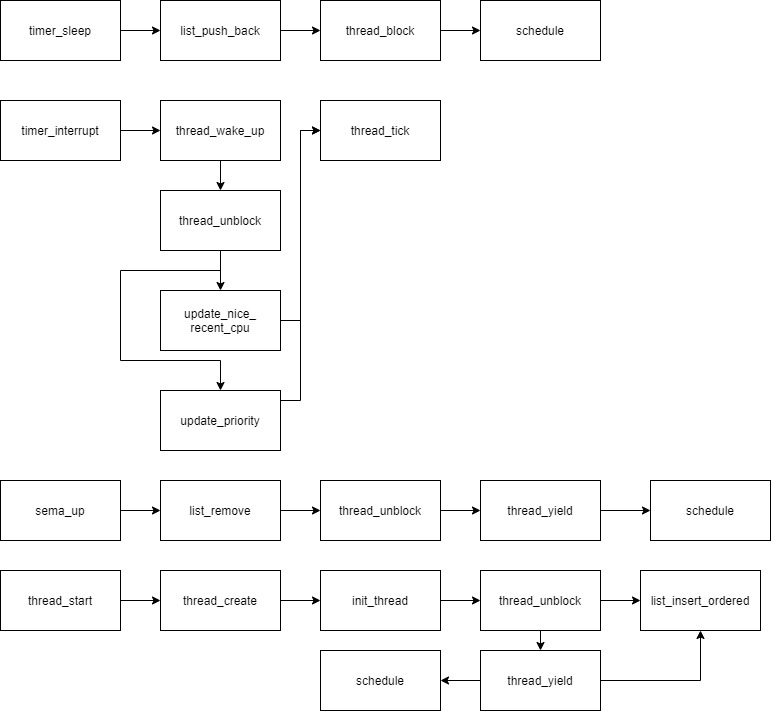
synch.c에서 priority에 대한 sema를 수정해준다. sema\_up 함수에서 sema를 기다리고 있는 list인 waiters 중 priority가 가장 큰 것을 찾아서 sema\_up을 해준다. 그리고 다시 scheduling을 하도록 코드를 추가한다.

thread.h와 init.c, thread.c에 priority aging을 위한 코드를 ppt를 보고 추가를 해준다.

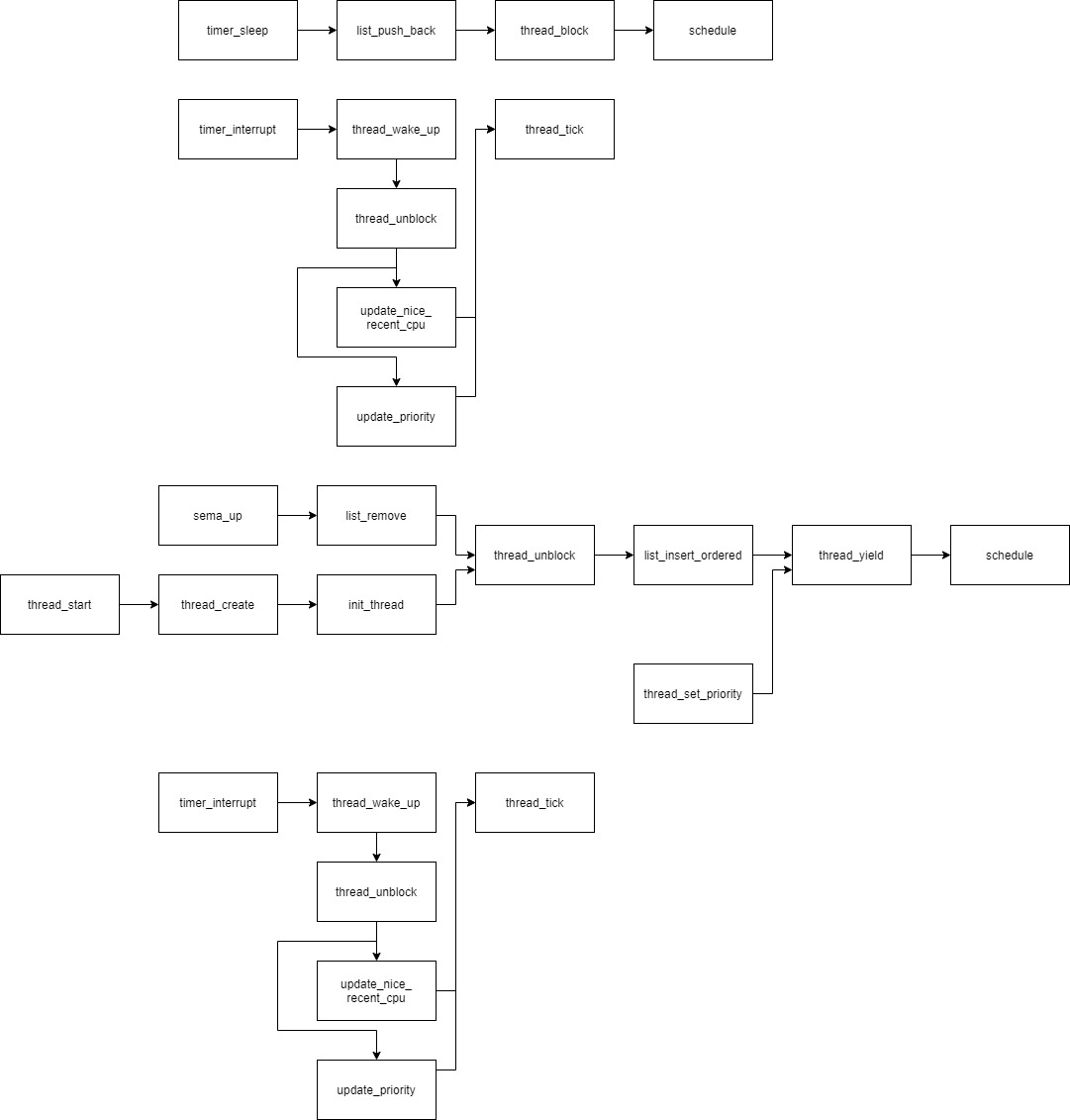
thread.h, thread.c에서 thread구조체에 recent\_cpu와 nice를 추가로 선언해준다. 그리고 load\_avg를 전역변수로 새롭게 선언한다. 초기값은 0으로 설정하고 자식 thread인 경우에는 부모의 값을 받도록 코드를 수정한다. 그리고 fixed point연산을 위해 FRACTION을 새로 define을 하고 nice, load\_avg, priority, recent\_cpu관련 함수를 모두 추가한다. 계산은 fixed point연산을 사용한다.

timer.c에서 timer\_interrupt 함수에서 thread\_prior\_aging과 thread\_mlfqs에 대한 코드를 수정한다. 그리고 4번째 tick마다 모든 thread의 priority를 update를 해주고 매초마다 timer interrupt 수인 timer\_FREQ만큼 tick이 흘렀을 때 load\_avg와 recent\_cpu를 update하도록 한다.

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

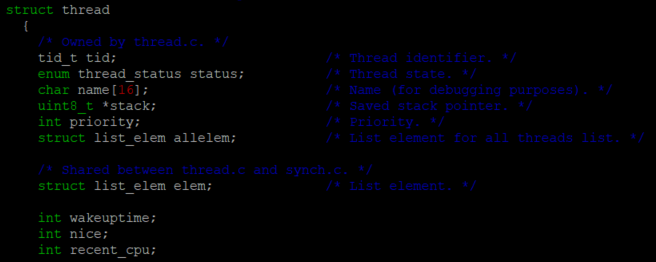


* + - 1. Priority scheduling

****

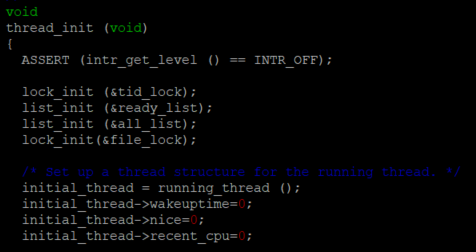
* 1. **제작 내용**

1. Alarm Clock
2. thread/thread.h



thread 구조체에 thread의 wake up time을 저장하는 wakeuptime을 새로 선언한다.

1. thread/thread.c

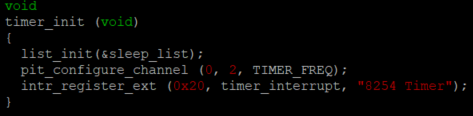


thread\_init 함수내에서 wakeuptime을 0으로 초기화한다.

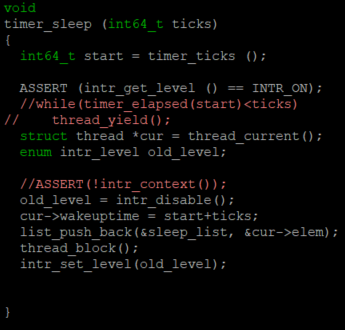
1. devices/timer.c



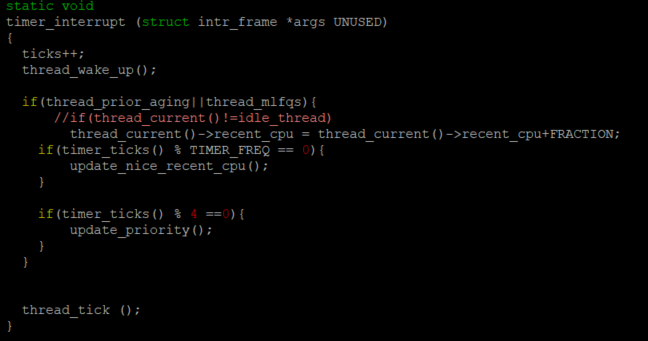
block된 list를 저장하는 sleep\_list를 전역변수로 선언한다.



timer\_init 함수에서 새로 선언한 sleep\_list에대해 list\_init 함수를 사용하여 초기화를 한다.

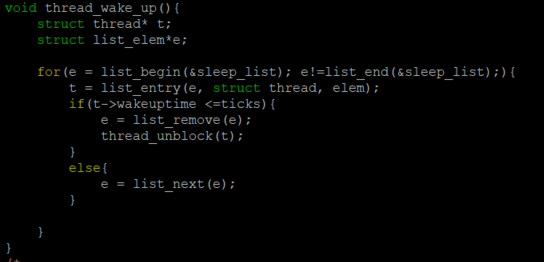


이전의 while문을 이용한 busy waiting 코드를 지우고 현재 thread의 wakeuptime을 현재 시간과 전달받은 ticks를 더해서 구한 다음 sleep\_list에 추가한다. 그러고 thread\_block() 함수를 통해 thread를 blocked 상태로 바꾼다.



timer\_interrupt 함수는 ticks을 증가시키는 함수이다. 따라서 thread\_wake\_up함수를 호출하여 ticks이 변할때마다 wakeuptime이 된 thread를 찾아서 깨워준다.

timer\_interrupt에서 사용하는 thread\_wake\_up 함수는 아래와 같다.



sleep\_list의 처음부터 끝까지를 탐색하여 thread의 wakeuptime이 ticks보다 작을 때 sleep\_list에서 삭제한다음 thread\_unblock함수를 호출하여 thread를 unblock 상태로 변경한다.

1. priority scheduling & advanced scheduling
   * 1. threads/thread.h

priority aging을 위해 bool type의 thread\_prior\_aging을 선언해준다.

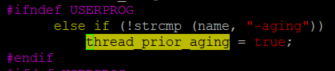


load\_avg를 전역변수로 선언해주고 fixed point 연산을 위해 FRACTION을 define한다.

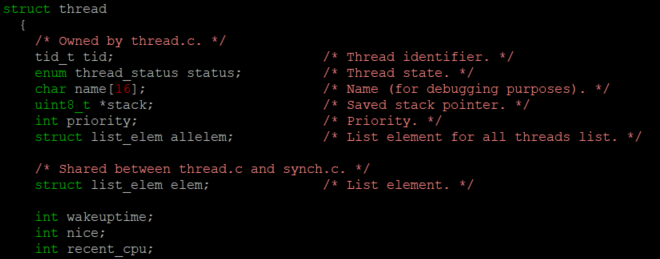


* + 1. threads/thread.init.c

thread.h에서 선언한 thread\_prior\_aging을 true로 값을 설정해준다.

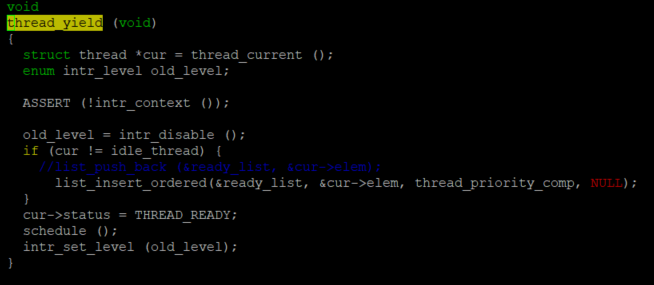


thread 구조체에 nice와 recent\_cpu를 저장하도록 2개의 값을 추가해준다.



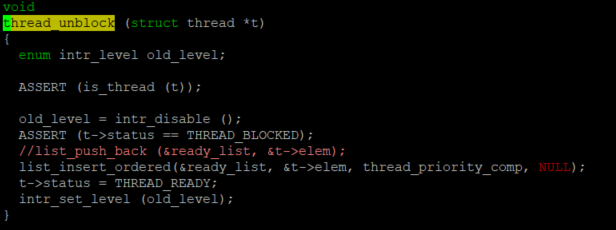
* + 1. thread/thread.c

thread\_yield 함수는 실행시킬 새 thread를 선택하는 scheduler에 CPU를 주는 함수이다.



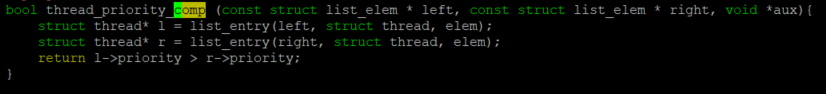
현재 thread가 idle\_thread가 아닐 때 ready\_list에 현재 thread의 elem을 list\_insert\_ordered함수를 이용하여 넣는다. 이때 priority 순서대로 넣기 위해 thread\_priority\_comp 함수를 사용한다. 그리고 현재 thread를 ready 상태로 바꾼 후 schedule()함수를 호출한다.

thread\_unblock 함수에서 blocked thread를 ready-to-run state로 바꾸기 때문에 ready\_list에 priority 순서대로 들어가도록 수정한다.

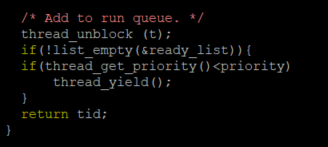


list\_insert\_ordered 함수를 이용하여 ready\_list에 thread\_priority\_comp 함수 순서대로 thread의 elem을 넣는다. 그리고 status를 thread\_ready상태로 바꿔준다.

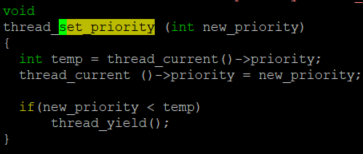
thread\_priority\_comp함수는 아래와 같다.



전달받은 두개 thread의 priority를 비교하여 결과를 bool type으로 return한다.



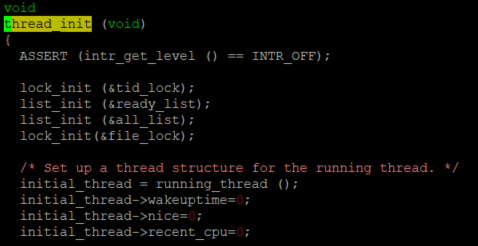
thread\_create 함수의 끝에서 새로 생성되는 thread의 priority가 수행중인 thread의 priority보다 더 크면 새롭게 스케쥴링을 하도록 한다. 따라서 ready\_list가 비어있 않을 때 비교를 하여 thread\_yield 함수를 호출하여 스케쥴링을 다시한다.



thread\_set\_priority함수에서는 현재 thread의 priority를 수정하는 일을 한다. 따라서 현재 thread의 priority에 전달받은 priority를 새로 저장하고 만약 새 priority가 더 작다면 thread\_yield함수를 호출하여 다시 스케쥴링한다.



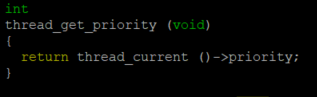
priority aging을 위해 thread\_prior\_aging을 bool type으로 전역변수로 선언한다.



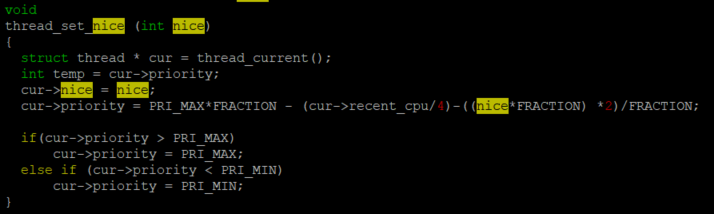
thread\_init 함수에서 nice와 recent\_cpu를 0으로 모두 초기화한다.



init\_thread 함수에서 recent\_cpu와 nice를 부모의 값을 받도록 running thread의 값으로 초기화한다.



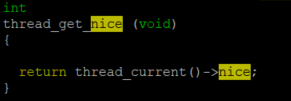
현재 thread의 priority를 return하는 함수이다.



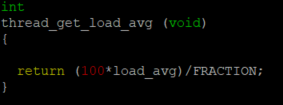
현재 thread의 nice값을 전달받은 nice로 설정하는 함수이다. 따라서 nice값을 변경하고 priority를 새로 계산해준다. 계산식은 아래의 계산식을 이용하였다.

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

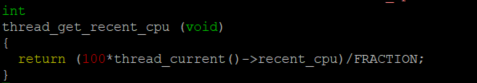
또한 fixed point 연산으로 해야되기 때문에 핀토스 매뉴얼의 연산방법을 사용하였다. priority를 새로 계산하였으므로 MAX, MIN 체크를 해야한다. 따라서 MAX보다 크거나 MIN보다 작은 값이 나왔을 경우 각각 MAX, MIN으로 다시 설정해준다.



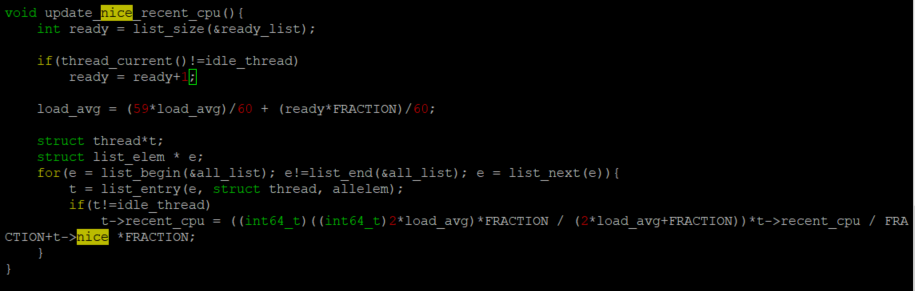
현재 thread의 nice값을 return하는 함수이다.



현재 시스템 load average를 100곱하여 return해주는 함수이다. 이것 또한 fixed point 연산으로 한다.



현재 thread의 recent\_cpu 값을 100 곱하여 return해주는 함수이다. 이것 또한 fixed point 연산으로 한다.

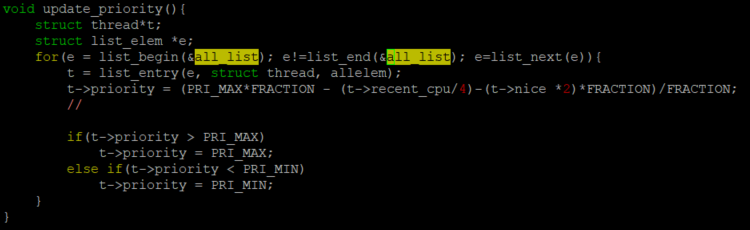


모든 process의 nice와 recent\_cpu를 update해주는 함수이다. load\_avg를 ready\_list에 현재 thread가 idle이 아니라면 현재 thread까지 합친 크기를 ready로 사용하여 값을 다시 계산한다. 식은 아래와 같다.

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

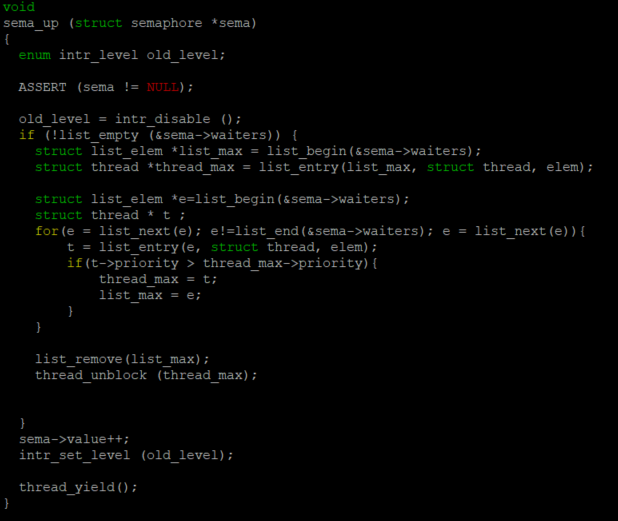
그다음 all\_list에 있는 모든 thread에 대해서 idle\_thread가 아니라면 recent\_cpu를 다시 계산한다. all list는 모든 process가 저장되어 있는 list이다. 식은 아래와 같다.

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



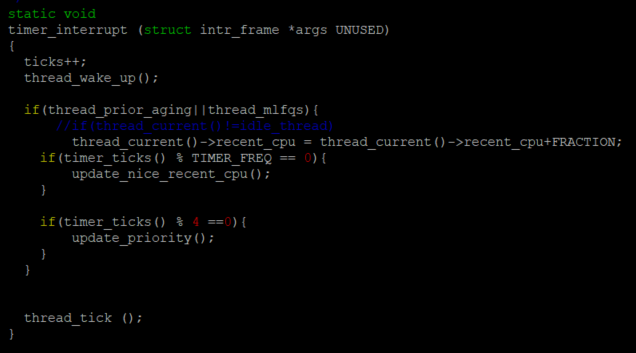
update\_priority 함수는 모든 process의 priority를 update해주는 함수이다. all\_list의 모든 thread에 대해서 priority를 계산해서 다시 값을 변경해준다. 이때 priority가 MAX와 MIN을 벗어난다면 각각 MAX와 MIN으로 설정해준다. 계산식은 위의 thread\_set\_nice 함수에서 사용한 식과 같다.

* + 1. threads/synch.c



sema\_up 함수에 priority관련 처리를 해준다. waiters에는 sema를 기다리고 있는 thread가 저장되어 있다. 따라서 waiters의 내부 thread를 탐색하여서 가장 큰 priority를 가지고 있는 thread를 찾는다. 그다음 list에서 list\_remove 함수를 통해 해당 list를 waiters에서 제거하고 thread를 thread\_unblock 함수를 통해 unblock 상태로 변경한다. 그다음 thread\_yield함수를 호출하여 다시 스케쥴링을 한다.

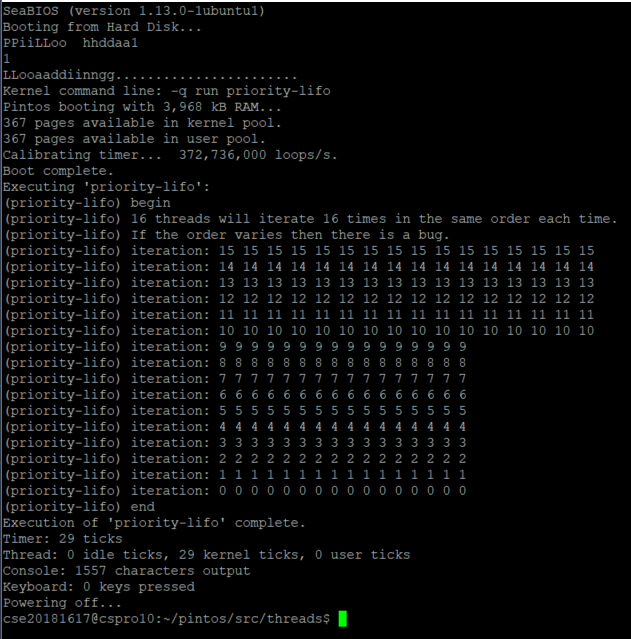
* + 1. devices/timer.c



timer interrupt에서 priority\_aging과 mlfqs에 대한 처리를 해준다. 따라서 먼저 tick이 증가함에 따라 thread의 recent\_cpu값도 1씩 증가시킨다. 그다음 timer\_ticks가 TIMER\_FREQ만큼 흘렀으면 모든 process의 nice와 recent\_cpu값을 update해주도록 update\_nice\_recent\_cpu 함수를 호출한다. timer\_ticks가 4만큼 흘렀다면 모든 process의 priority를 update해주도록 update\_priority 함수를 호출한다

* 1. **시험 및 평가 내용**

1. priority-lifo 결과



먼저 priority-lifo.c 코드를 분석해보면 동일한 우선순위의 thread를 여러 개를 만든 다음 동일한 round-robin 순서로 일관되게 실행하도록 하는 코드이다. 따라서 16개의 thread가 last in first out즉 lifo대로 16번 반복된다. 실행 결과를 보면 0부터 15까지의 이름을 가진 thread 16개가 모두 동일한 round-robin 순서로 실행되고 있는 것을 볼 수 있다.

1. make check 결과

