**Pintos Project 3: Threads**

담당 교수 : 박성용

조 / 조원 : 20161566\_권형준

개발 기간 : 11/16~12/8

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

프로젝트3에서는 alarm clock기능과 priority scheduling을 구현한다. 우선 alarm clock은 이미 구현이 되어있지만, busy waiting을 하는 매우 비효율적인 구조로 구현이 되어 있다. 이를 수업시간에 배운 semaphore라는 synchronize 기술을 사용하여 더 효율적으로 구현한다. 다음 scheduling 관련하여, pintos manual에는 pintos가 기본적으로 round-robin scheduling을 사용한다고 나와있다. Round-robin은 모든 process(thread)들이 각각 같은 시간만큼 돌아가면서 처리하는 것이다. Round-robin은 평균적인 response time이 높지만, 모든 process, thread가 끝나는데 걸리는 평균 시간인 turn around time이 매우 높다는 단점이 있다. 이번 프로젝트에서는 priority를 기준으로 scheduling을 실행하는 priority scheduling과, priority scheduling 기법을 사용하면 발생할 수 있는 starvation을 해결하기 위한 aging을 구현한다. 이후, priority scheduling의 aging을 더 recent\_cpu, nice value등의 값으로 priority를 변경함으로써 구현한 BSD scheduler을 구현한다. BSD scheduler는 multi-level ready queue를 가지는 구조이다. 같은 priority 내에서는 round-robin방식으로 scheduling이 된다. 하지만 어차피 4ticks마다 모든 thread의 priority가 변경됨으로 하나의 queue를 사용하는 priority scheduling으로 생각할 수 있다. 이러한 점을 염두하고 구현을 진행하였다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

기존의 timer\_sleep함수에서는 busy waiting, 즉 while 반복문을 사용하여 thread를 sleep 시켰다. 이는 cpu를 계속 사용하여 전체적인 프로그램의 효율을 떨어뜨린다. 하지만 이를 queue를 사용하여 block시켜놓으면 매번 tick할 때 한 번만 확인하면 되기 때문에 cpu사용 효율이 올라간다.

이번에 사용하는 test로는 alarm clock을 제대로 구현했는지 확인하기 힘들다. 이미 pintos상에서 busy waiting으로 구현하여 제대로 동작하기 때문이다. 제대로 구현했는지를 알기 위해서는 성능을 검사해야 되는데, 이 또한 sleep을 시켜 놓는 thread들이 매우 많아야 차이가 발생한다. 하지만 제대로 구현하였다면, 정상적으로 시간이 지난 뒤 sleep\_queue에서 ready\_queue로 이동이 되어 test를 당 통과하게 될 것이다. 또한 cpu 효율도 늘어날 것이다. (test를 통과하는 시간이 더 빨라지지 않을까 예상한다)

* 1. Priority Scheduling

Priority scheduling이 필요한 이유는 프로그램을 만드는 사람의 목적에 따라서 각 thread의 중요도가 달라지기 때문이다. 예를 들어서 프로그램이 최단시간에 끝나는 것이 중요하다면 priority = cpu burst가 될 것이다. 만약 먼저 끝나야 되는 thread가 있다면 그 thread의 priority를 더 높게 프로그래머가 줄 수 있을 것이다. 이처럼 priority scheduling은 다양한 장점이 있기 때문에 필요하다. 한가지 단점이 있다면 starvation이 발생할 수 있다. 이는 priority가 높은 thread만 계속 실행되면서 priority가 낮은 thread는 제대로 실행되지 못하는 상황을 뜻한다. 이를 해결하기 위해서 필요한 것이 바로 aging이다. 시간에 따라서 또는 최근 cpu사용 내역에 따라서 priority를 조정하는 것이다.

Priority scheduling(with aging)을 잘 구현하게 되면, starvation이 일어나지 (거의)않고 설정한 priority순서대로 thread가 실행될 것으로 예상된다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Priority scheduling을 multi-level feedback queue를 사용하는 구조로 구현한 것이다. 또한 aging이 nice, recent cpu use등의 요소로 priority를 계속 update하여 구현된다. 이 scheduler가 단순한 priority scheduler의 aging에 비하여 nice 변수를 사용하여 aging의 정도를 조절할 수 있고, recent cpu를 사용하여 더 균등한 기회를 주는 형식으로 scheduler를 구현할 수 있다는 장점이 있다. 이를 구현하게 되면 앞에서 말한 것과 같은 단순한 priority scheduler에 비한 장점들을 기대할 수 있을 것이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

우선 busy waiting을 사용하지 않으려 고한다면, sleep을 시키는 thread들을 저장할 queue가 필요할 것이다. 또한 queue에 저장한 뒤, 깨워야 되는 시간을 저장하는 변수가 각 thread내부에 있어야 될 것이다. 이를 위하여 변수 wakeup\_time, list sleep\_list를 사용한다. Timer\_sleep이 호출되면, 이 queue에 넣고 깨워야 될 시간을 tick을 사용하여 기록한다. 이후에 매 tick마다, 즉 timer\_interrupt마다 이 queue에 저장되어 있는 모든 thread들을 탐색하여 일어날 시간이 현재 tick과 같으면 깨워주게 된다. 이때, thread\_unblock을 사용하는데 이는 해당하는 thread를 ready list에 넣은 뒤 thread\_yield를 사용하여 다시 scheduling을 한다.

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

Ready list에 thread가 priority가 높은 thread가 추가되는 경우는 2가지가 있다.

* + - 현재 존재하는 thread의 priority가 높아져서 running thread보다 priority가 높아지는 경우
    - 현재 running thread보다 priority가 높은 thread가 create되는 경우

따라서 priority를 update한 후, 새로운 thread를 생성한 후에 running thread와 priority를 비교하여 더 높을 경우 thread\_yield를 호출한다. Thread\_yield는 현재 running thread를 ready상태로 바꾼 뒤(ready queue에 추가한다), ready queue 전체에서 priority가 가장 높은 것을 추출하여 running 상태로 만들어준다.

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

Advanced scheduler의 priority는 4tick마다 갱신이 된다. 이때, priority를 계산하는 식은 다음과 같다.

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

* + - PRI\_MAX

thread가 가질 수 있는 priority의 최대 값이다. 프로젝트에서는 이 값을 63으로 설정하고 있다.

* + - Recent\_cpu

Recent\_cpu는 각 thread가 마지막으로 cpu를 사용한 시간을 의미한다. 매초마다 모든 thread의 recent\_cpu값이 update된다. 현재 running thread의 경우 매 tick마다 추가로 1씩 증가한다. 매초마다 update 되는 recent cpu의 식은 다음과 같다.

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

* + - Load\_avg

Load\_avg는 ready queue에 있는 thread들 수의 평균이다. 이 값 또한 매초마다 갱신이 된다. 기존의 값에 59의 비율을, 현재 ready queue에 들어있는 thread 수의 값에 1의 비율을 할당하여 갱신하며, 식은 다음과 같다.

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

* + - nice

각 thread마다 고유하게 가지고 있는 값이다. 초기에는 0의 값을 가지거나 부모에게서 상속을 받는다. 가질 수 있는 값의 범위는 -20~20사이의 정수이다.

이제 위에 정리한 각각의 요소들이 어떻게 갱신이 되고, 어떠한 성질이 있는지 알아봤으니, priority를 계산할 때 어떤 식으로 영향을 주는지 알아보자. 우선 nice의 경우 식에 나온 것처럼 값이 클수록 priority는 낮아진다. 따라서 priority를 높여주고 싶은 thread일수록 nice값은 작게 하면 될 것이다. Recent\_cpu의 값도 작을수록 priority가 높아지는데, recent\_cpu값과 nice의 값이 비례함으로 이 경우에도 nice값은 priority와 반비례할 것이다. Recent\_cpu는 오랫동안 사용하지 않을수록 더 작은 값을 가진다. 식을 보면 priority와는 반비례하는 관계를 가진다. 즉, 오랫동안 사용하지 않은 thread의 priority를 높여주어 aging을 구현한 것이다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

|  |  |
| --- | --- |
| **날짜** | **개발 범위** |
| **11/16~11/23** | **Alarm clock** |
| **11/23~11/30** | **Priority scheduling** |
| **11/30~12/4** | **Advanced scheduler** |
| **12/4~12/8** | **보고서** |

* 1. **개발 방법**
* II. B.의 개발 내용을 구현하기 위해 각각에 대해 다음 사항들을 포함하여 설명

1. **Alarm clock**
   * 수정해야하는 소스코드

* **<src/threads/thread.h>**: thread내부에 sleep을 끝나는 시간인 wakeup time이라는 변수를 추가해야 된다.
* **<str/devices/timer.c>**: sleep 중인 thread들을 저장하는 sleep queue와 sleep, awake를 해주는 함수를 수정해야 된다.
  + 수정하거나 추가해야 하는 자료구조
* **Wakeup\_time**:thread 구조체 내부에 언제 thread를 깨워야 되는지에 대한 정보를 담고 있는 변수를 저장한다. 처음에 thread를 생성할 때 0으로 초기화를 해주며, timer\_sleep을 하게 되면 sleep시작 시간의 tick과 sleep할 시간의 tick을 더한 값을 저장한다.
* **Sleep\_list**: sleep하는 thread들을 저장한다. 현재 실행 중인 thread의 wakeup\_time변수에 일어날 시간을 기록하고, sleep\_list에 추가하고 이후에 timer\_interrupt가 발생할 때마다 이 queue를 돌면서 일어날 시간이 된 thread를 깨워준다.
  + 수정하거나 추가해야 하는 함수
* **Timer\_sleep**: 기존에 busy waiting(while문)으로 구현되어 있는 sleep을 thread\_block, sleep\_list를 사용하여 최적화를 해준다. 현재 돌고 있는 thread를 sleep시킨 이후, thread\_block을 호출하며, 이는 자동으로 schedule이라는 함수를 호출하여 다음 running할 thread를 정해준다.
* **Timer\_interrupt**: 매 timer\_interrupt가 호출될 때마다, tick이 하나씩 증가한다. 이렇게 증가한 tick과 현재 sleep중인 thread들의 wakeup\_time을 비교하여(sleep\_list를 통하여) 일어날 시간이 된 thread들을 깨워준다. 깨워준다는 의미는 sleep\_list에서 제거한 이후 thread\_unblock을 호출하여 ready\_list에 다시 해당 thread를 추가해주는 것을 의미한다.

1. **Priority scheduling(aging X, advanced scheduler에 설명)**
   * 수정해야하는 소스코드

* **<src/threads/thread.c>**: 이전에 존재하던 ready queue를 priority based ready queue로 바꿔준다. 또한 priority가 바뀌었을 때 running thread가 최고의 priority를 가지는지, 그렇지 않다면 바꾸어 준다.
* **<src/devices/timer.c>**: aging을 구현하기 위해 수정해야 된다. 이번 프로젝트에서 aging은 BSD Scheduler에서 사용하는 nice, priority, recent\_cpu등을 사용하여 구현하였기 때문에 이는 뒤에 “3)advanced scheduler”에서 다룬다.
* **<src/threads/synch.c>**: semaphore를 사용할 때, 하나의 semaphore에 막혀 있는 thread들 중 priority가 가장 높은 thread가 먼저 unblock이 되어야 된다.
  + 수정하거나 추가해야 하는 자료구조
* **Ready\_list**: list인 점은 변하지 않는다. 하지만 새로운 thread를 추가할 때 단순히 push\_back을 하지 않고 insert\_ordered를 사용하여 priority queue로 유지한다. “list\_insert\_ordered”를 위하여 각각의 thread들의 priority를 비교해주는 함수가 필요하다.
  + 수정하거나 추가해야 하는 함수
* **Thread\_yield**: ready list에 새로운 thread를 추가할 때 단순히 push\_back을 하지 않고 insert\_ordered를 하여 priority를 기준으로 정렬된 상태로 유지한다. Running thread의 priority가 바뀐 이후, 새로운 thread가 ready list에 추가된 이후 이 함수가 호출되어 현재 돌고 있는 thread의 priority가 가장 높은지 확인해준다.
* **Thread\_unblock**: block상태의 thread를 ready list에 추가해준다. 이때, yield와 같이 insert\_ordered를 사용한다. Yield와 한가지 다른 점은 ready list에만 추가하고 schedule함수를 호출하여 현재 running thread를 preempt를 하지 않는다.
* **Priority\_compare**: 두개의 thread의 priority를 비교하여 순서를 정해주는 함수이다. “list\_insert\_ordered”의 인자 중 비교 함수로 사용되며 priority queue를 유지하는 핵심 함수이다.

1. **Advanced scheduler**
   * 수정해야하는 소스코드

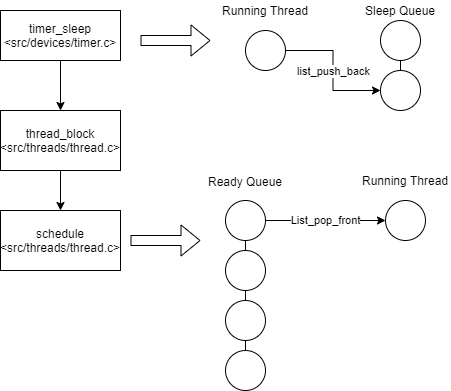
* **<src/threads/thread.c>**: 앞에서 생성한 priority queue를 그대로 사용하면 된다. 하지만 aging을 할 때 사용하는 nice, recent\_cpu등을 조절할 수 있는 함수들을 추가해주어야 한다.
* **<src/threads/thread.c>**: BSD scheduler는 aging등의 이유로 인하여 주기적으로 priority와 priority를 계산하기 위한 요소들을 update해준다. 이를 timer\_interrupt에 추가해주어야 한다.
* **<src/threads/float\_tool.h>**: pintos에서는 float연산을 지원해주지 않기 때문에 float와 int, float와 float사이의 연산을 간편하게 해줄 수 있는 함수들을 작성한다.
  + 수정하거나 추가해야 하는 자료구조
* **Nice**: 가 thread마다 가지고 있는 값이다. Priority를 update할 때 사용되며, 초기에는 0을 가지거나 부모 thread에게서 상속을 받는다. 따로 사용자에 의하여 set될 수 있다.
* **Recent\_cpu**: 가장 최근에 사용한 cpu의 시간을 상대적으로 저장하고 있는 변수이다. 이 값이 작을수록 오랫동안 사용하지 않은 thread이며 이 또한 priority를 계산할 때 사용된다.
* **Load\_avg**: ready list안에 있는 thread의 수들이 가지는 평균이다. 계속 ready queue의 thread 수는 변하기 때문에 이 값 또한 주기적으로 update된다.
  + 수정하거나 추가해야 하는 함수
* **Update\_priority**: priority를 주기적으로 update해주는 함수이다. 모든 thread들은 4 tick마다 priority의 update가 이루어져야 하며, 여기에 사용되는 recent\_cpu, nice등은 바로 밑의 함수에서 update, set을 해줄 수 있다. Update가 끝난 이후에는 새로운 priority들을 각각의 thread가 가지고 있기 때문에 list\_priority\_compare함수를 기준으로 ready list를 다시 sort해주어야 된다.
* **Update\_load\_avg\_and\_recent\_cpu**: 매초마다, recnent\_cpu, load\_avg를 update해준다. 이 과정에서 필요한 float연산은 구현한 함수를 사용하여 간편하게 해결한다.
* **Timer\_interrupt**: 앞의 두 함수가 매초마다, 4tick마다 실행될 수 있게 해주는 코드를 timer\_interrupt에 추가한다. 또한 추가로 현재 running thread의 경우 추가적으로 current\_cpu의 값에 1을 더해준다.
* **Thread\_set\_nice**: 특정 thread의 nice값을 update, set해준다. 새로운 nice를 가지고 priority를 계산하고 이에 따라서 만약 running thread보다 priority가 더 높아진다면 preempt를 시킨다.
* **Thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu**: nice, load\_avg, recent\_cpu값들을 핀토스 명세서에 나와있는 것처럼 반환하는 함수들이다.
* **Max\_priority**: 현재 ready list의 thread들 중 가장 높은 priority를 가지는 thread의 priority를 반환한다.
* **Int\_plus\_float, int\_minus\_float, 등등**: pintos에서는 float 자료형, 연산을 지원해주지 않는다. 하지만 recent\_cpu와 load\_avg등을 계산할 때 소수점 계산이 필요하기 때문에 이러한 연산을 구현하여야 한다. pintos에서는 fixed-point format을 사용함으로 예를 들어서 정수화 실수를 계산하는 경우 정수에 fixed point만큼 shift를 시킨 뒤 연산을 수행하는 형식으로 모든 연산을 구현해 놓는다.

1. **연구 결과**
   1. **Flow Chart**

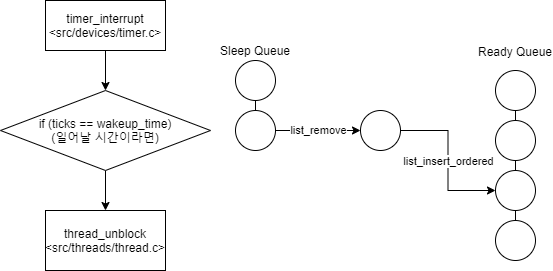
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)
* Alarm Clock

(추가적으로 흐름도는 아니지만, 함수가 하는 일의 이해를 돕기 위하여 간단한 그림을 추가하였습니다)

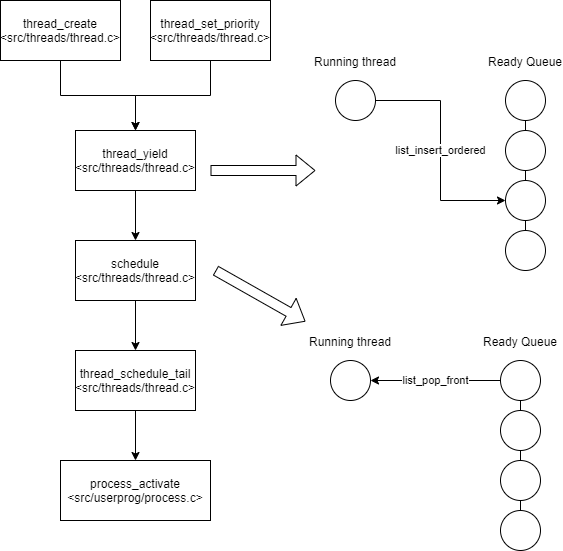
* + - sleep하게 되는 과정



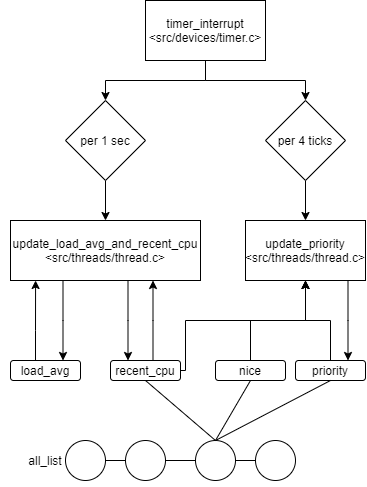
* + - awake하게 되는 과정



* Priority Scheduler
  + - Priority scheduling

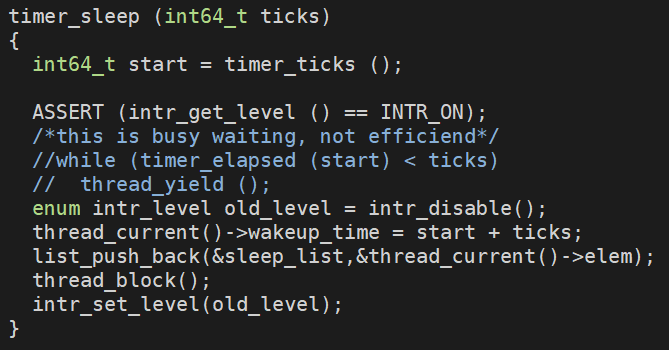


* + - Aging(추가 구현에서 사용되는 aging)

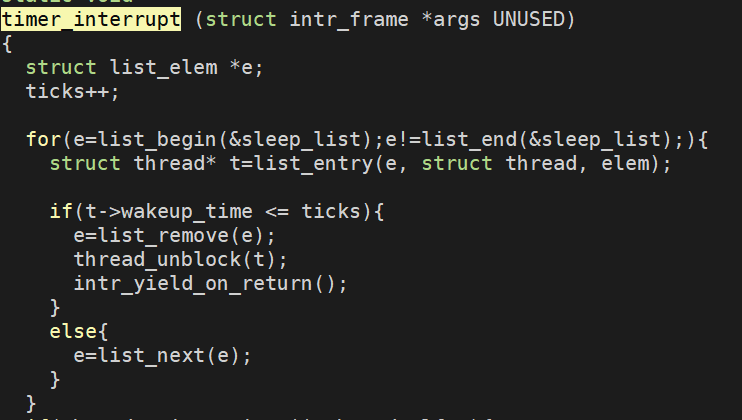


* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명

1. **Alarm Clock**



Alarm clock은 크게 두개의 부분으로 나누어 진다. 현재 돌고 있는 running thread를 sleep시키는 timer\_sleep함수와, 자고 있는 thread들 중 일어날 시간이 된 thread를 깨워서 running list에 추가해주는 timer\_interrupt이다. 우선 timer\_sleep부터 살펴보자. 우선 현재 tick을 알기 위하여 timer\_ticks()함수를 호추한다. 현재가 몇 tick인지 알아야 원하는 tick이후에 일어날 시간에 해당하는 tick을 계산할 수 있다. 다음 interrupt를 disable시킨 다음(여기서 interrupt가 발생하면 timer\_ticks의 반환 값이 달라지거나 running thread가 바뀔 수 있기 때문이다.) 현재 running thread의 wakeup\_time에 일어날 시간을 저장한다. 다음 sleep\_list에 이 thread를 추가하고 thread를 block시킨다. 이 일이 끝난 뒤 interrupt를 다시 enable 시킨다. 쉽게 정리하면 running state에서 blocked state로 바꿔주고 이 blocked state의 thread들을 추적하기 위하여 sleep\_list에 추가한 것이다.



이제 두번째 부분인 timer\_interrupt함수를 살펴보면 다음과 같다. 매번 interrupt가 발생할 때마다, tick이 하나씩 증가한다. 증가한 tick을 가지고 sleep\_list를 돌면서 해당하는 thread의 wakeup\_time과 현재 tick을 비교한다. 만약 일어날 시간을 지났거나 같다면 sleep\_list에서 해당 thread를 제거하고 thread\_unblock을 호출한다. Thread\_unblock을 통하여 해당 thread는 READY상태가 되었고 ready list에 추가되었다. 프로젝트의 alarm clock을 구현할 때는 필요 없는 단계이지만 intr\_yield\_on\_return()을 호출하였다. 이는 현재 external interrupt가 발생하는 도중에 yield를 호출할 수 없기 때문에 interrupt가 끝나기 직전에 yield를 호출해주는 함수이다. 여기서 yield가 필요한 이유는 sleep에서 ready상태로 돌아온 thread 중, 현재 running thread보다 priority가 높은 thread가 존재할 수 있기 때문이다.

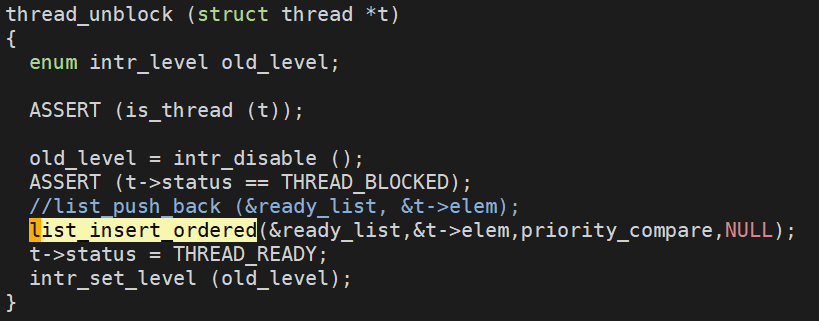
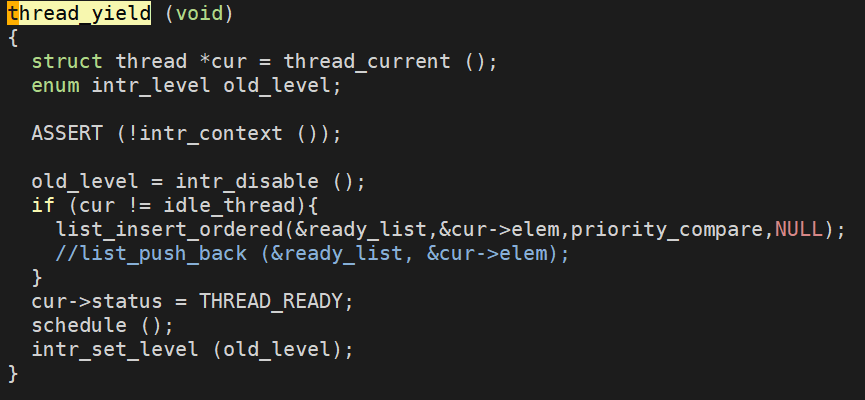
1. **Priority Scheduling**

Priority scheduling은 단순히 ready list를 우선 순위 큐(priority queue)로 구현하면 된다. pintos내부의 함수 중 이를 도와주는 두개의 함수는 다음과 같다.

**List\_insert\_ordered**: 인자로 들어온 비교 함수를 사용하여 list를 특정 기준을 가지고 정렬된 상태로 유지할 수 있게 insert를 실시한다.

**List\_sort**: 인자로 들어온 비교 함수를 기준으로 list의 내용을 특정 기준을 가지고 정렬한다.

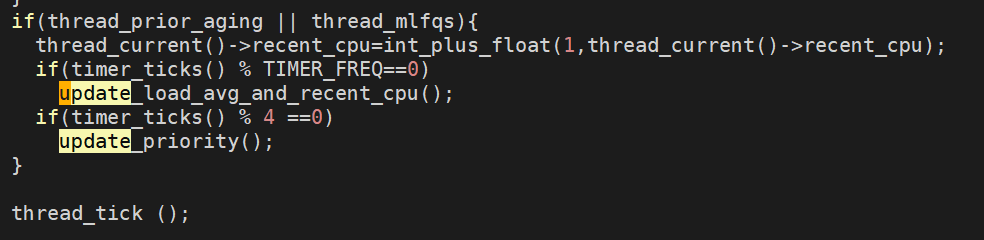
Ready queue에 새로운 thread가 추가될 때 기존에 사용하던 list\_push\_back 대신 list\_insert\_ordered를 사용함으로써, list\_pop\_front만 하여도 priority가 가장 높은 thread를 알 수 있다. Ready queue에 새로운 thread가 생성되어 추가되거나, 기존에 존재하던 thread의 priority가 변하였을 때 list는 재정렬이 필요할 수 있다. 이 경우 list\_sort를 사용하여 ready list를 정렬된 상태로 유지할 수 있다. Thread\_yield는 현재 실행중인 thread를 ready\_list에 넣고 schedule함수를 호출하여 가장 높은 priority를 가진 thread를 running thread로 설정하여 준다. 또한 thread\_unblock함수는 해당 thread를 ready\_list에 priority를 기준으로 삽입하는 함수이다. 이 두곳에서만 list\_insert\_ordered를 사용하여 주면 ready\_list를 우선순위로 유지할 수 있다.



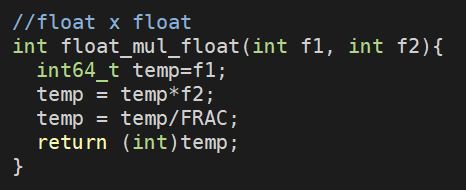
이후에 aging을 적용하게 되면 모든 thread들의 priority가 한 번에 변하는 경우가 발생한다. 이런 경우 list\_sort를 사용하여 새롭게 할당된 priority를 기준으로 새로운 running thread를 결정한다.

한가지 추가로 구현해야 되었던 것은 바로 semaphore사용시의 priority이다. Sema\_up이 호출되는 순간, 해당 semaphore에 대기하고 있던 모든 thread들이 풀려나서 wait list의 가장 앞의 thread가 나와 실행되게 된다. 즉, sempahore에 기다리고 있는 thread들을 저장하는 sema->waiter들이 priority를 기준으로 정렬이 되어있지 않다. 이 방법을 해결하는 방식은 두가지가 있다. 하나는 이 list또한 priority를 기준으로 정렬을 하는 것이다. 나머지 하나는 sema\_up을 할때마다 list를 전체 검색하여 가장 높은 priority를 가진 thread를 깨워주었다. 실제로는 첫번째 경우가 자료구조 측면에서 더 유리할 것이다. (탐색시간, 삽입시간 등등)하지만 코드 구현상으로는 후자가 더 간편하여 후자로 구현하였다.

1. **Advanced Scheduler**



위에서 구현한 priority scheduler을 그대로 사용하여도 된다. 실제로는 multi-level feedback queue scheduler로 각 priority에 해당하는 queue가 존재해야 된다. 그리고 같은 priority사이에서 round-robin이 이루어지는 형식이다. 하지만 BSD scheduler의 경우 매 4tick마다 priority를 update하는 과정이 발생한다. 즉, 매 4tick마다 모든 priority에 해당하는 queue의 내용이 계속 바뀐다는 뜻이다. 따라서 하나의 priority queue를 사용하여 scheduling을 하여도 문제가 없다. Mlfq의 가장 높은 priority queue에서 4tick의 주기를 가지는 round-robin scheduling을 하는 것과 같은 결과를 보인다고 할 수 있다(정확히 같지는 않지만). 따라서 추가로 구현하는 것은 priority와 priority를 계산하는데 사용되는 요소들을 update하는 코드 밖에 없었다. 위의 사진에 나와 있는 코드는 timer\_interrupt의 일부분으로, 주기적으로 load\_avg, recent\_cpu, prioiryt를 update하기 위함이다. 각각의 함수에서 pintos manual에 나온 식을 가지고 update를 한다. 여기서 한가지 주의할 것은 pintos에서 실수형 연산, 자료형을 제공하지 않는다는 것이다. 매번 연산을 할 때마다 14bit만큼 shift하는 것이 번거롭기 때문에 float와 int, float와 float사이의 연산들을 미리 함수로 구현하였다. Over flow가 발생할 수 있기 때문에 int64\_t자료형을 사용한 뒤 다시 int형으로 바꾸는 형식을 취하였다. 하나의 예를 들어보면,

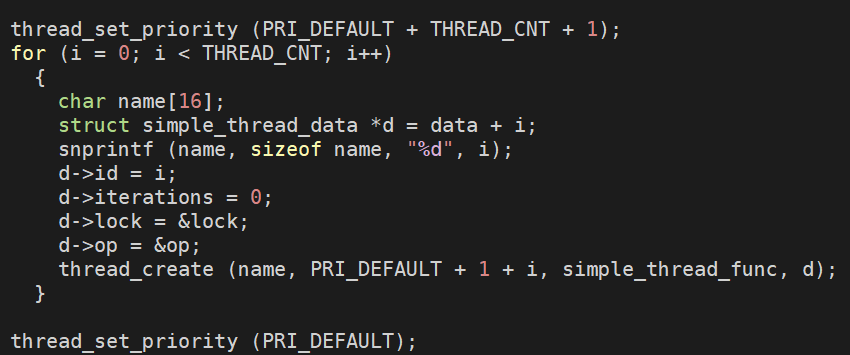


위의 코드는 float자료형과 float연산을 곱하는 것을 구현한 것이다. 이때 FRAC는 (1<<14)를 define한 것으로 pintos에서 사용하는 고정 소수점의 fraction을 위한 bit의 수에 해당한다. 즉 초기에 f1과 f2는 (실제로 저장하는 수\*fraction)의 수를 표현하고 있는 것이다. 따라서 두 수를 곱하게 되면 (실제 f1 \* 실제 f2 \*fraction \*fraction)이 된다. Float를 표현하기 위해서는 fraction을 하나만 곱해야 되기 때문에 FRAC으로 한번 나눠주는 과정이 필요하다. 이런 식으로 recent\_cpu, load\_avg등을 계산하는데 필요한 수식을 작성하고 update\_\*\* 함수들에서 이를 update해주었다. 앞에서 priority scheduling에서 말한 것처럼 한 번에 모든 priority들이 update되었기 때문에 ready\_list를 다시 sorting해준 다음 intr\_yield\_on\_return함수를 사용하여 ready\_list에서 가장 priority가 높은 thread를 running thread로 돌려준다.

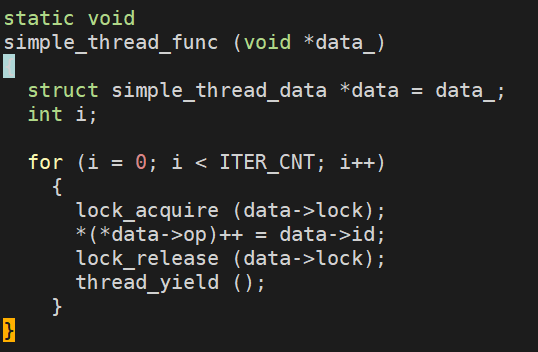
마지막으로 priority scheduler에서 priority를 바꿀 수 있었던 것처럼, BSD scheduler에서는 nice값을 바꿀 수 있는 함수 thread\_set\_nice함수를 구현하여야 된다. nice값이 바뀌면, nice를 사용하여 계산하는 recent\_cpu, priority값들이 즉시 바뀌게 된다. 따라서 이 값들 또한 update를 하고 마지막으로 thread\_yield를 호출하여 가장 높은 priority의 thread가 running thread가 되도록 설정한다.

* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
  1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

priority-lifo.c코드를 들어가보면 설명으로 같은 priority를 가진 몇 개의 thread를 생성하여 round-robin으로 제대로 동작하는지를 확인하는 코드라고 나와있다. Aging을 제대로 구현한 scheduler라면 priority가 같다고 하더라도 cpu를 사용하지 않은 thread들은 priority가 높아져 round-robin의 형태를 띄게 된다. Test 코드를 앞에서부터 살펴보면 다음과 같다.



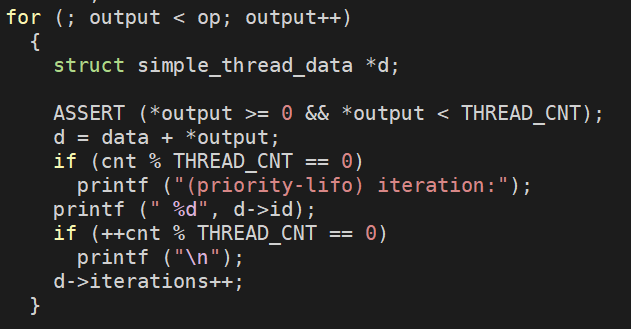
먼저 해당 프로그램의 priority를 PRI\_DEFAULT+THREAD\_CNT+1로 설정하였다. 이는 thread\_create를 하면서 새로 생성한 thread들이, 돌고 있는 thread보다 priority가 높아져서 thread create하는 과정이 중간에 멈추지 않게 하기 위함을 유추할 수 있다. 모든 thread가 생성된 이후(총 16개) 다른 thread들을 생성한 thread의 우선순위를 PRI\_DEFAULT로 하여 다른 thread들보다 우선순위를 낮추었다. 여기서 한가지 이상한 점을 발견하였다. **앞의 주석과는 다르게 각각의 thread들의 priority들이 달랐다**. 즉, (PRI\_DEFAULT+1, PRI\_DEFAULT+2, …, PRI\_DEFAULT+15)의 값을 각각 가지는 thread들이 생성되었다. 처음 결과가 나왔을 때 하나의 thread만 반복되어 코드가 잘못된 줄 알았다. 하지만 priority가 애초부터 모두 다르다면 이야기가 달라진다. 이제 simple\_thread\_func을 살펴보자.



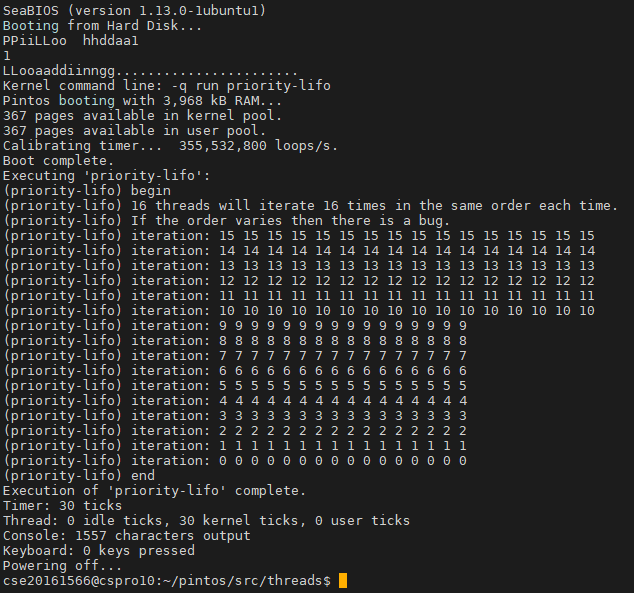
Simple\_thread\_func은 각각의 thread내부에서 수행되는 함수이다. 여기서 사용되는 data->lock은 총 16개의 모든 thread에서 공통으로 사용하는 semaphore(lock)이다. 따라서 \*(\*data->op)++ = data->id는 synchronized 되어 동시에 2개의 thread가 이 부분을 실행하는 일은 없게 된다. 이 작업이 끝나면 thread\_yield()를 하여 priority가 높은 thread에게 순서를 넘겨준다. 이제 각각의 thread들은 data->op 즉, output buffer에 자신의 id를 순서대로 기록하게 된다. Output buffer을 간단하게 정리하면 배열로, thread들이 총 16개 있으며 각각 16번의 반복문을 돈다. 따라서 16\*16개의 수를 저장할 수 있는 배열이라고 생각하면 된다. 정확하게는 이중 포인터로 모든 thread들이 하나의 buffer에 쓸 수 있게 구현한 것이다.

Priority scheduler에 의하여 가장 priority가 높은 (thread\_id=)15가 먼저 실행될 것이다. 그럼 배열은 [15]가 될 것이다. 이제 thread\_yield()를 호출하여 다음 priority가 높은 것을 고르는데 아직 15라서 [15,15]가 될 것이다. 하지만 이는 정확하게 모른다. 우리가 구현한 priority aging 상으로 아직 15가 priority가 가장 높을 수 있지만, 극단적인 aging으로 구현하게 된다면 다음 몇번의 반복문 이후에 14가 나왔다가 나중에 다시 15가 나올 수 있다. 하지만 결과를 보니 겨우 15번의 반복으로는 priority를 역전시키기는 힘들었던 것 같다. 따라서 순서대로 15가 16번, 14가 16번, 13이 16번, …, 0이 16번 등장한 것을 알 수 있다. 또한 aging을 하지 않은 scheduler의 경우에도 다음과 같은 결과가 나왔을 것이다.

코드의 나머지 부분은 단순히 output buffer을 읽어서 해당하는 data의 id를 출력하고 16번(THREAD\_CNT)마다 개행을 해주는 것이다.



결과는 다음과 같이 나왔다.



* make check 수행 결과를 캡처하여 첨부

