**Pintos Project 3: Threads**

담당 교수 : 김영재 교수

학번 / 이름 : 20171682 임정호

개발 기간 : 21’.10.27 ~ 21’.10.31

1. **개발 목표**

기존의 pintos가 사용하는 스케줄러를 개선하여, thread가 일을 보다 효율적으로 처리할 수 있게 한다. thread의 Alarm-Clock과 Priority Schedule, BSD Scheduler를 구현한다.

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

기존의 pintos에서 구현되어 있는 timer\_sleep()은 thread가 일어날 시간이 아닌 경우에도 계속 while loop을 돌면서 확인하기 때문에 비효율성이 생긴다. 그래서 일어날 시간이 아닌 경우라면 thread의 state를 blocked\_state로 바꿔두고, 일어날 시간에만 thread를 깨워준다. 불필요한 thread 실행시간을 줄일 수 있다.

* 1. Priority Scheduling

기존의 pintos는 Round-Robin방식으로 thread가 스케줄된다. 이는 thread의 priority를 고려하지 않기 때문에, priority가 높은 thread가 먼저 실행될 수 있도록 한다.

* 1. Advanced Scheduler

Thread의 priority를 priority를 스케줄러가 조절해준다. priority가 끊임없이 변경되기 때문에, priority가 낮은 thread들도 잘 실행될 수 있게 한다.

* 1. **개발 내용**

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

thread\_tick()에서 thread\_wake\_up()이라는 함수를 호출하게끔 한다. 깰 시간이 되지 않은 thread를 관리하는 list, sleeping\_threads를 선언하고, 각 쓰레드는 일어날 시간을 저장하는 wakeup\_time 구조체 변수를 가지게 한다. sleeping\_threads를 wakeup\_time의 값이 작은 thread가 list의 앞에 위치하게끔 priority-queue로 관리한다. 현재 pintos의 ticks 값과 wakeup\_time을 비교하여 ticks값이 더 큰 경우라면 쓰레드 리스트에서 제거하고 해당 thread를 thread\_block()으로 ready\_list로 넣어준다.

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

부모 thread가 새로운 자식 thread를 thread\_create()에서 생성하는데, priority가 부모보다 높은 경우, thread\_yield()를 호출하여 thread 스케쥴이 다시 이루어지게끔 한다. thread\_yield()에서 현재 running\_state였던 thread를 priority에 맞게 ready\_list로 넣어주고, schedule()을 호출한다.

1. Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

쓰레드마다 nice, recent\_cpu변수가 필요하고, 전역 변수로 load\_avg가 필요하다.

**nice :** thread가 create될 때 정해지며, initial\_thread는 0을 가지며, 그 외의 경우는 부모 thread의 nice값을 물려받는다.

**recent\_cpu :** 해당 thread가 최근에 cpu를 얼마나 사용하였는가를 나타내며, thread가 create될 때 정해지고, initial\_thread는 0을 가지고, 그 외의 경우는 부모 thread의 recent\_cpu값을 물려받는다. nice와는 다르게, 1초마다 모든 쓰레드의 recent\_cpu값이 아래와 같은 공식으로 계산되어 조정된다.



**load\_avg :** 지난 1분동안 수행 가능한 thread의 평균 개수를 추정한 값이다. nice와 recent\_cpu와는 다르게, thread-specific하지 않고, system-wide하다. 아래와 같은 공식으로 1분마다 값이 변경된다.



1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10.27 ~ 10.28 : 전체강의 재시청 후, pintos manual 정독, Alarm Clock 구현

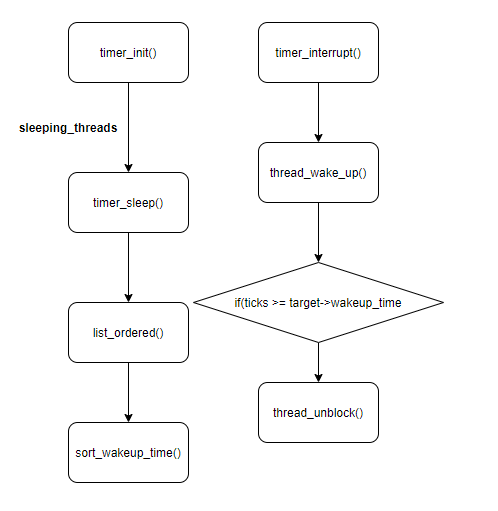
10/29 : Priority Scheduling, aging 구현

10.30 ~ 10.31 : BSD Scheduler 구현

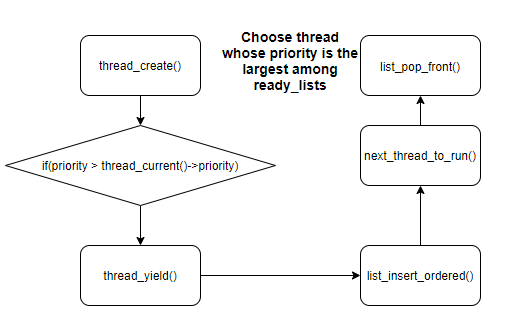
* 1. **개발 방법**
  + **수정해야하는 소스코드 :** thread.c, thread.h, timer.c, init.c, synch.c
  + **수정하거나 추가해야 하는 자료구조 :** struct thread
  + **수정한 함수 :** thread\_tick(), thread\_create(), thread\_unblock(), thread\_yield(), thread\_set\_priority(), thread\_get\_priority(), thread\_set\_nice(), thread\_get\_nice(), thread\_get\_load, init\_thread(), timer\_interrupt(), timer\_init(), timer\_sleep(), sema\_up(), parse\_options()
  + **추가한 함수 :** sort\_priority(), thread\_get\_load\_avg(), thread\_get\_recent\_cpu(), update\_recent\_cpu(), update\_load\_avg(), recalculate\_priority(), sort\_wakeup\_time(), thread\_wake\_up()

1. **연구 결과**
   1. **Flow Chart**

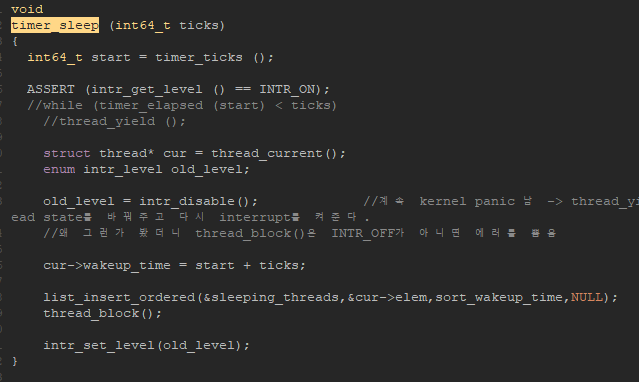
* 1) Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.



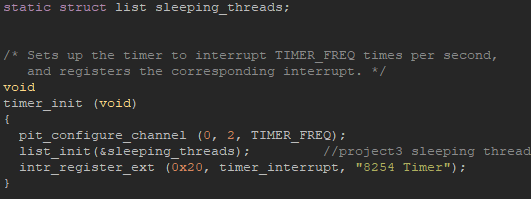
* 2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야 하는지 서술.



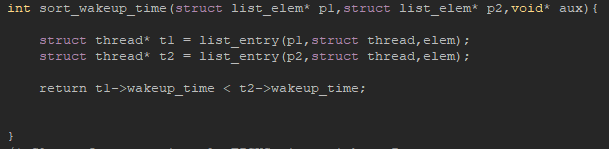
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명
* 1) Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.



기존의 pintos에서 제공하던 timer\_sleep()은 while loop으로 계속 ticks와 경과한 시간을 비교하면서 ready\_state와 running\_state를 반복하는 구조로 구현되어 있다. ready\_list로 갈 시간이 되지 않았으면 확인을 안 하면 되는데, 이런 구조에서 비효율성이 발생한다. 따라서 sleeping\_threads라는 전역 변수를 선언하여, blocked\_state인 thread를 관리한다. 이는 timer\_init()에서 초기화한다.



쓰레드 구조체 변수에 wakeup\_time을 추가하여, 각 thread가 일어날 시간을 저장한다. sleeping\_threads는 각 thread의 wakeup\_time을 기준으로 우선순위 큐로 구현하였고, sleeping\_threads의 맨 앞에 있는 thread는 가장 가까운 미래에 깨어날 thread가 위치한다. 이러한 order는 list\_insert\_ordered()를 사용하여 가능하게 하였다. list\_insert\_ordered()에는 매개변수로 함수를 넘기는데, sort\_wakeup\_time()을 새로 구현하였다.



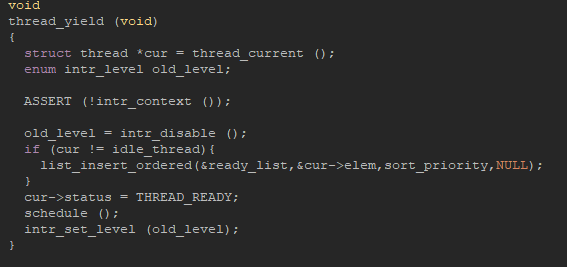
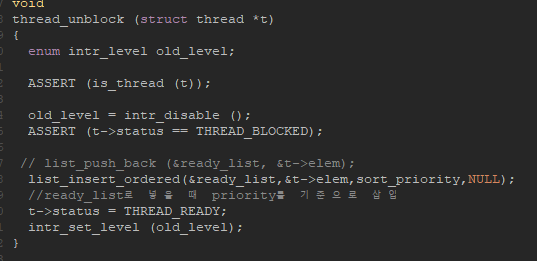
이 함수를 매개변수로 넘김으로써 wakeup\_time이 작은 순으로 list에 삽입될 수 있게 한다. sleeping\_threads에 thread를 insert하고나서 thread\_block()을 호출한다. 이 때 intr\_disable()을 사용하여 interrupt를 끄지 않으면 에러가 났다. 이 에러가 왜 났는지 고민해본 결과, 원래의 코드는 thread\_yield()를 호출한다. thread\_yield()에서도 thead\_block()을 호출하는데, thead\_block()은 interrupt가 꺼진 상태가 아닌 경우 에러처리를 하는 코드가 있기 때문에 에러가 났던 것이다. 이제 thread를 재웠으니, 깨울 일만 남았다. 깨우는 방법은 thread\_tick()에서 thread\_wake\_up()을 호출하게 한다. 이 때, thread\_tick()은 timer\_interrupt()에서 매 tick마다 호출된다.

매 tick마다 sleeping\_threads에서 일어날 시간이 된 thread를 찾는다. 이 때, wakeup\_time이 작은 순으로 sleeping\_threads에 저장했으므로, list의 앞부분부터 깰 시간이 안 되면, 바로 종료할 수 있다.

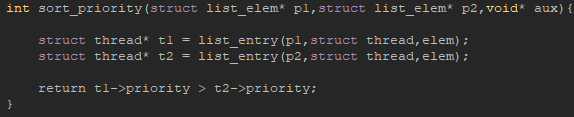
텍스트이(가) 표시된 사진

자동 생성된 설명

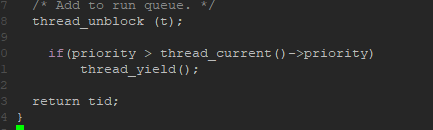
* 2) Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 priority scheduling에 따르면 어떻게 해야하는지 서술.

우선 ready\_list를 priority가 큰 순서로 우선순위 큐로 구현한다. 이를 위해 thread\_yield()와 thread\_unblock()에서 ready\_list에 thread를 추가할 때, list\_insert\_ordered()를 이용하여 삽입한다.

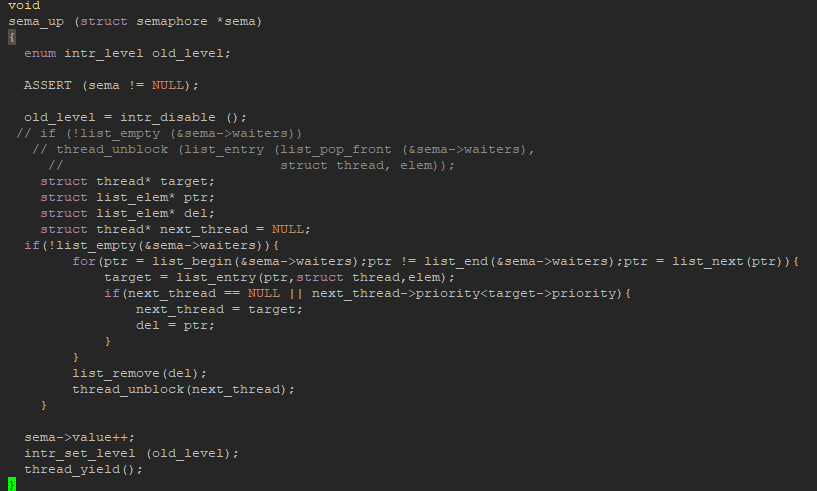
이 때 위에서 했던 것처럼 order의 기준을 알려줄 함수를 매개변수로 넘겨야 한다. priority가 큰 순으로 정렬된 상태로 list에 삽입될 수 있도록 한다.



thread\_create()에서 새로운 thread가 생성하고, thread\_unblock()으로 ready\_list에 삽입하고 나면, 현재 thread의 priority와 새로 생성된 thread의 priority를 비교한다. 비교해서 새로운 thread의 priority가 큰 경우 thread\_yield()를 통해 스케줄링이 다시 실행되게끔 한다. preemptive한 스케줄링을 보장할 수 있다.

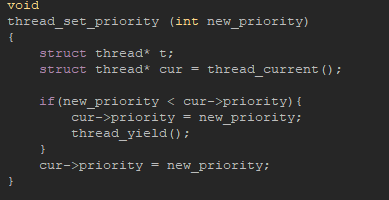


새롭게 추가되는 thread의 priority가 큰 경우에는 현재 실행되던 thread는 cpu를 양보해야 하는 것처럼, threads가 lock이나 semaphore를 위해 기다리는 중에도, 높은 priority를 가진 thread가 먼저 실행되어야 한다. 기존의 pintos는 blocked threads를 깨울 때 그저 sema->waiters의 list의 front를 pop하는 형식이었다. 이런 구조에서 sema->waiters에서 priority가 가장 큰 thread를 먼저 thread\_unblock() 하는 방식으로 바꾼다.



이 때, priority-sema에서 에러가 났다. sema\_up()에서 blocked\_threads 중에서 priority가 가장 높은 thread를 unblock()하는 과정에서 ready\_list와 꼬이는 것 같다. 그래서 sema\_up()에서 현재 thread의 priority보다 next\_thread의 priority가 더 큰 경우, thread\_yield()를 하게끔 했음에도 에러가 발생했다. 그래서 위의 조건을 지우고, 항상 reschedule되게끔 thread\_yield()를 호출하였더니, 에러가 해결되었다.

thread의 priority를 변경하는 경우이다. 이 때도 변경하는 priority가 현재 priority보다 더 큰 경우 reschedule되게끔 thread\_yield()를 호출한다.



* 3) Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.

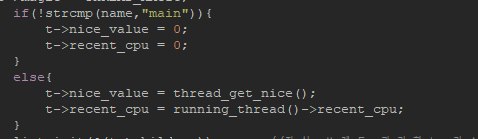
우선 pintos는 floating 연산을 지원하지 않는다. 그렇기 때문에 17.14 format을 이용한다. pintos manual을 참고하여 실수와 실수의 곱셈, 나눗셈 그리고 실수와 정수의 덧셈을 매크로로 정의하였다. 처음부터 매크로로 할 생각은 없었지만, 실수와 실수의 연산은 조금 복잡했기 때문에, 매크로로 정의하였다.

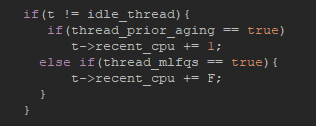
텍스트이(가) 표시된 사진

자동 생성된 설명

ㅇ

모든 thread는 구조체 변수로 recent\_cpu와 nice\_value를 가진다. init\_thread()에서 thread의 이름이 main이면 init\_thread이므로, nice\_value와 recent\_cpu값을 0으로 초기화하고, 그 외의 경우는 부모의 nice\_value와 recent\_cpu를 물려받게 한다.

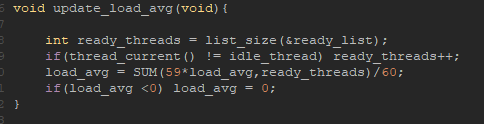


thread\_tick()에서 thread\_prior\_aging이 true이면 recent\_cpu는 1씩 증가시키고, thread\_mlfqs가 true이면 fixed-format 적용으로 F만큼 증가시킨다.

pintos 매뉴얼을 보면, 4tick마다 각 thread의 priority를 재계산해야 한다고 나와있다. 이를 위해서 recalculate\_priority()라는 함수를 구현하여, timer\_interrupt()에서 4초마다 호출되게끔 하였다. 그리고 1초마다 load\_avg와 recent\_cpu값을 재계산해야 한다고도 나와있다. pintos 매뉴얼에 timer\_ticks() % TIMER\_FREQ의 값이 0이면 알 수 있다고 적혀있다. update\_load\_avg()와 update\_recent\_cpu()를 구현하고, timer\_interrupt()에서 1초마다 호출되게금 하였다.

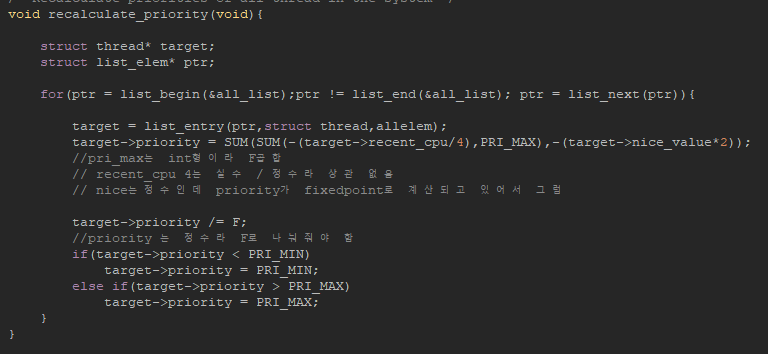
load\_avg는 아래의 공식으로 계산된다.



단순한 식이지만, 17.14 format을 이용하기 때문에, 다소 식이 지저분해진다.

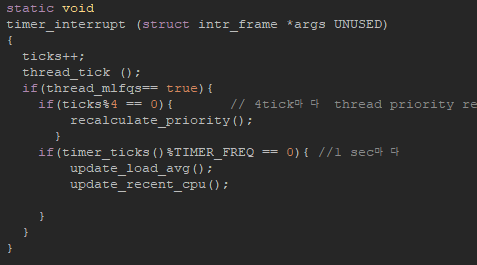
ready\_threads는 ready\_list의 list\_size()로 쉽게 구할 수 있고, 매뉴얼을 읽어보면, 현재 실행되고 있는 thread도 idle\_thread가 아니라면 count에 포함되어야 한다고 한다. 그래서 idle\_thread가 아닌 경우 ready\_threads를 1 증가시킨다. 위에서 정의한 매크로를 이용하여 load\_avg를 계산한다. load\_avg가 음수이면 0으로 초기화해준다.

priority를 재계산할 때는 ready\_list에 있는 thread만 계산하는게 아니라, 모든 thread를 대상으로 해야한다. for문으로 all\_\_list에 있는 thread를 돌면서, 아래의 공식대로 계산해준다.

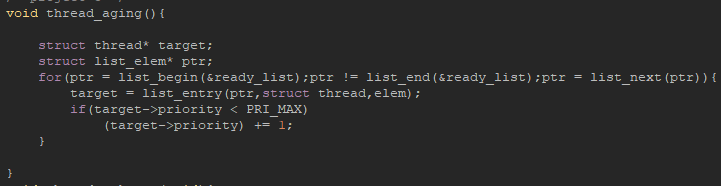
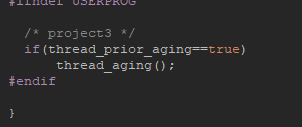


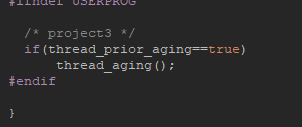
laod\_avg는 실수의 값을 가진다. 따라서 2\*load\_avg와 1을 더할 때는 1에 F만큼 값을 곱해주고 더해야 하고, (2\*load\_avg)와 (2\*load\_avg + 1)을 나눌 때는, 실수와 실수의 나눗셈으로 정의해둔 매크로를 이용한다. 그리고 그 값을 recent\_cpu와 곱할 때는 실수와 실수의 곱셈이므로, 매크로를 이용한다. nice\_value는 정수의 값이므로 F를 곱한 다음 더해주고, priority값은 PRI\_MIN과 PRI\_MAX 사이의 정수값을 가져야 하므로 F로 나눠준다.

계산된 priority값이 PRI\_MIN과 PRI\_MAX보다 작거나 큰 경우, 가까운 값으로 설정해준다.



timer\_interrupt()의 코드이다. thread\_tick()을 호출하고, thread\_mlfqs가 true인 경우에만 priority 계산을 해준다. ticks와 timer\_ticks()는 사실상 같은 의미이다. 매뉴얼대로 따라가다 보니 본의 아니게 달라졌는데, 큰 의미는 없다. mlfq가 아니고 thread\_aging모드인 경우에는 priority aging을 위해 thread\_aging()이라는 함수를 구현하여 thread\_ticks()에서 호출하게끔 하였다.



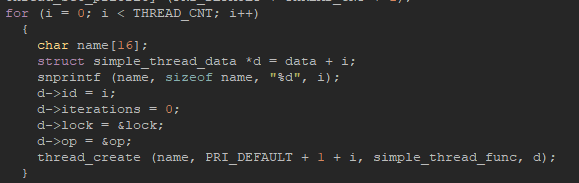


* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

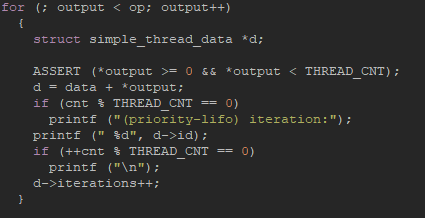
priority-lifo.c 코드의 특징은 늦게 생성되는 thread일수록 그 priority가 높다는 점이다.

늦게 생성되었지만, priority가 높기 때문에 가장 먼저 실행된다.



 우선, thread를 THREAD\_CNT만큼 생성하기 전에 본인의 priority를 PRI\_DEFAULT + THREAD\_CNT + 1의 값으로 설정하는데, 이 의미는 생성하는 다른 thread보다 priority를 높게 두어서, 본인이 자식 thread를 모두 생성할 때까지는 자식 thread가 실행되지 않게끔 하기 위함이다.

for문으로 THREAD\_CNT만큼의 자식 thread를 만드는데, 그 priority는 PRI\_DEFAULT+1+i이다. 이 의미는 늦게 생성되는 thread일수록, i값이 커지므로, 더 높은 priority를 가진다는 말이다. 일찍 생성된다고, 일찍 실행되지 않을 수 있다는 얘기이다.

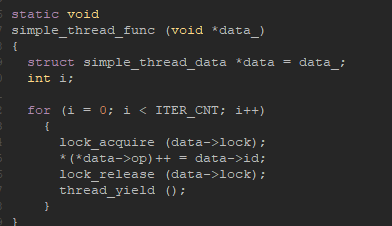


data는 자식 thread 배열의 시작주소를 가리킨다. output을 증가시키면서

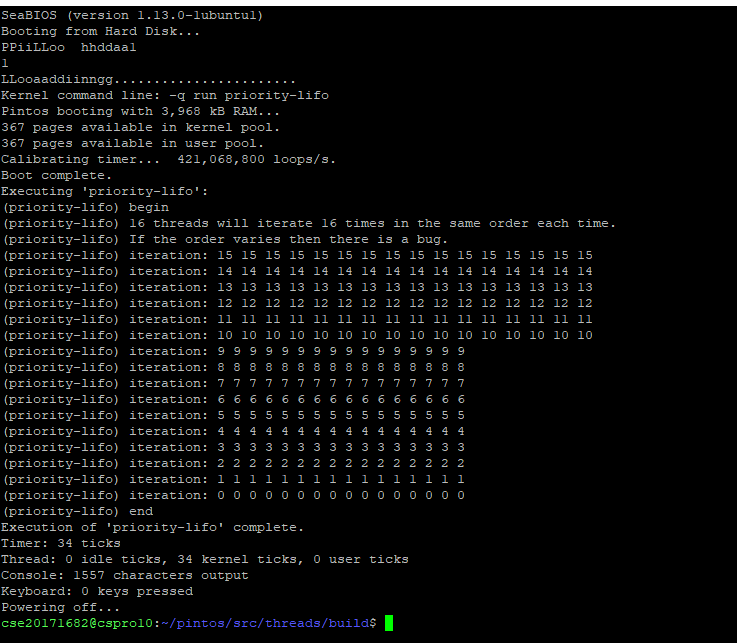
output이 가리키는 곳에 저장된 값 만큼 data에서 떨어진 곳을 d가 가리키게 한다.

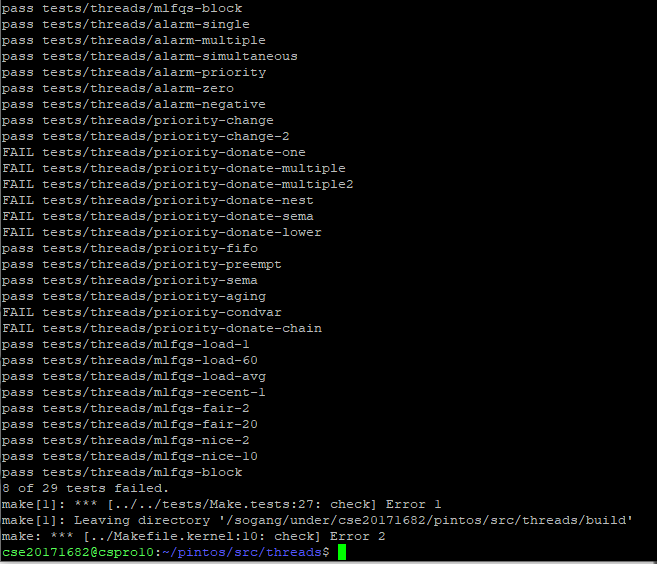


이 때, 초기상태에 output과 op는 같은 곳을 가리킨다.



생성된 각 thread들은 ITER\_CNT만큼 data->op에 본인 id를 남긴다. 남기는 이유는 앞서 d = data + \*output에서 output에 본인 id가 들어가게끔 하기 위함이다. 본인의 id를 남기면 d->id를 출력할 때, 본인의 id가 출력된다. 따라서 priority가 높은 thread부터 ITER\_CNT만큼 본인의 id가 출력될 것이다. 순서대로 priority가 낮은 thread까지 for loop으로 각 thread당 ITER\_CNT만큼 thread의 id가 출력될 것이다. 다음은 출력 결과이다.



* make check 수행 결과를 캡처하여 첨부