**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171705 / 한솔

개발 기간 : 10.15 ~ 11.06

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

alarm clock기능과 priority scheduling, bsd scheduler를 구현한다. alarm clock은 timer\_sleep함수를 통해서 구현하고 일정한 시간동안 스레드를 멈추는 기능을 한다. priority scheduling은 스레드마다 큐에 입력될 때, 가장 뒤에 입력되는 것이 아니라 priority에 따라 입력되게 한다. bsd scheduler는 MLFQ를 구현한다. priority를 주기적으로 새로 계산하게 된다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

alarm clock은 timer\_sleep함수를 이용해서 해당 스레드를 멈출 수 있다. 이를 위해서 기존에 계속해도 yield하는 방법에서 스레드를 block 시키는 방식으로 전환해서 효율성을 높인다.

* 1. Priority Scheduling

큐에 입력되는 스레드가 단순히 큐의 가장 뒤에 입력되지 않고, priority에 따라서 큐의 중간에 입력된다. 이를 통해서 높은 priority를 가진 스레드는 우선적으로 cpu를 배정받고, 낮은 priority를 가진 스레드는 늦게 cpu를 배정받게 할 수 있다.

* 1. Advanced Scheduler (추가구현을 한 경우)

multi level feedback queue를 구현한다. nice값과 최근에 cpu사용량에 따라서 priority가 새롭게 계산된다. cpu사용량이 적으면 더 높은 priroity를 갖게되고, cpu 사용량이 많으면 더 낮은 priority를 갖게된다. 이렇게 함으로써 한정된 cpu를 효율적으로 사용할 수 있게된다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

block을 할 때 스레드를 깨울 시간을 저장해두고, tick마다 block된 스레드의 리스트를 확인해서 스레드를 깨울 시간인지를 확인한 후 꺠워야한다면 unblock하여 상태를 바꾸고 ready\_list에 추가한다.

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

ready list의 가장 앞부분에 새로운 thread를 집어넣은 후에, thread\_yield를 통해서 cpu를 current thread보다 높은 priority를 가진 스레드에게 양보해야한다.

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

load\_avg는 과거에 실행된 ready list에 있는 스레드의 평균적인 개수이다. 이를 이용해서 recent\_cpu를 계산할 수 있다. recent\_cpu는 각 프로세스들이 얼마나 많은 cpu시간을 최근에 사용했는지를 나타낸다. nice값은 각 스레드가 얼마나 자신의 cpu시간을 다른 스레드에게 양보를 하는지를 나타내는 값이다. priority를 계산할 때, recent\_cpu뿐아니라, 이 nice값도 사용된다. nice값은 -20~20사이의 값을 가지고 값이 클수록 다른 스레드에게 cpu를 더 양보한다. 즉, priority = PRI\_MAX - (recent\_cpu / 4) - (nice \* 2)와 같이 계산되므로, nice값이 클수록 새로 계산되는 priority가 작아진다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

10.23~10.25 alarm clock

10.26 ~10.28 priority scheduling

10.29 ~ 10.31 advanced scheduler

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

alarm clock을 구현하기 위해서는 devices/timer.c에 있는 timer\_sleep함수를 수정해야한다. thread\_yield를 계속해서 호출하는 것은 비효율적이므로 block하는 함수를 추가해야한다. 또한 thread.c에 있는 thread\_tick함수를 수정해야한다. block되어 있는 thread를 일정시간이 지난후에 unblock해야하기 때문에 이 함수가 호출될 때마다 unblock할 thread가 있는지를 확인하고 unblock하는 함수를 호출한다.

advanced scheduler를 구현하기 위해서 thread.c에 있는 thread\_tick()를 수정해야한다. 일정한 tick마다 load\_avg와 recent\_cpu를 다시 계산해야하고, 우선순위도 다시 계산하도록 수정해야한다. thread\_init()에서 initial\_thread의 nice와 recent\_cput값을 설정하는 부분을 추가한다. 그리고 init\_thread()에서 새로운 스레드의 nice와 recent\_cpu를 초기화하는 부분을 추가해야한다. thread\_create에서는 새롭게 생성한 스레드의 우선순위가 더 클 경우에 context switch를 하도록 해야한다.

* + 수정하거나 추가해야 하는 자료구조

alarm clock을 구현하기 위해서 thread.h에서 struct thread 구조체에 struct list\_elem block\_elem을 추가한다. 이는 스레드를 block리스트에 집어넣기 위해서 필요하다. 또한 thread.c에 static struct list blocked\_list를 추가한다. 이는 block된 스레드를 저장하는 리스트이다.

advanced scheduler를 구현하기 위해서 struct thread에서 nice와 recent\_cpu라는 변수를 추가해야한다. 스레드마다 이를 저장해야하기 때문이다. 그리고 thread.c에서 load\_avg 변수를 추가해야한다. 실수연산을 할 때, 더하기, 빼기, 곱하기, 나누기를 결정하기 위해서 enum을 추가한다.

* + 수정하거나 추가해야 하는 함수

alarm clock을 구현하기 위해서 thread.c에 thread\_block\_with\_time이라는 함수를 추가한다. 이는 현재 스레드를 block하고 리스트에 추가하는 역할을 한다. 또한thread.c에 block\_check라는 함수를 추가해야한다. 여기서는 현재 tick을 확인하며 blocked\_list에 있는 thread중에서 unblock해야하는 스레드를 찾아서 unblock한다.

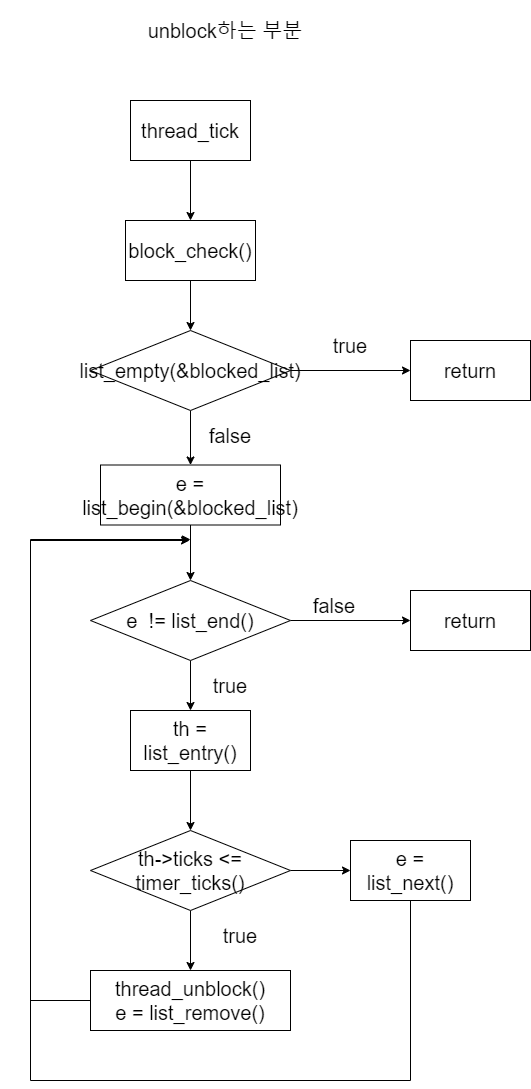
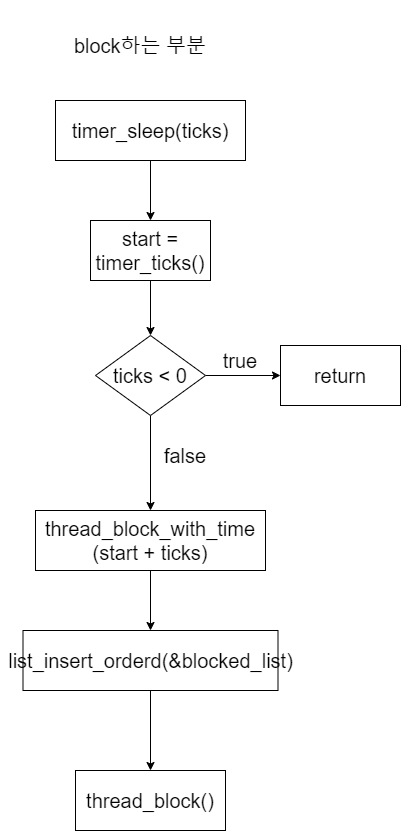
priority scheduling을 구현하기 위해서 thread.c에 list\_compare\_priority라는 함수를 추가한다. 이 함수는 priority값에 따라서 정해진 위치에 list의 node를 insert하기 위해 필요한 비교를 하는 함수이다. 또한 thread.c에 위치한 thread\_unblock함수와 thread\_yield 함수에서 list에 값을 집어넣을 때 list\_push\_back 대신 list\_insert\_ordered 함수로 대체한다. priority에 따라서 리스트의 어느 위치에 insert되어야하는지가 구현되어야하기 때문이다. 그리고 thread.c에서 thread\_set\_priority에서는 현재 스레드의 priority를 재설정하고 필요에 따라 context switch가 일어나도록 수정한다. thread\_get\_priority함수에서는 현재 스레드의 priority값을 반환하도록 한다. 그리고 sema\_down함수를 수정해야한다. sema의 waiter 리스트에 집어넣을 때 priority에 맞게 집어넣어넣어서 sema\_up을 해줄 때, priority가 높은 순으로 unblock되어야 한다. 그리고 매 틱마다 현재 스레드의 recent\_cpu값을 1씩 증가하는 함수도 추가해야한다.

advanced scheduler를 구현하기 위해서 정수를 fixed\_point 실수로 변환하는 함수, 그 반대로 변환하는 함수, fixed\_point 실수와 정수를 계산하는 함수, 실수와 실수를 변환하는 함수를 추가한다. 그리고 load\_avg를 새로 계산하는 함수, recent\_cpu를 새로 계산하는 함수, priority를 새로 계산하는 함수가 필요하다. 또한 현재 thread\_set\_nice함수를 수정해서 변경된 nice값에 따라 priority를 새로 계산하도록 수정한다. thread\_get\_nice함수는 현재 스레드의 nice값을 반환하도록 수정한다. thread\_get\_load\_avg함수는 현재 load\_avg \* 100의 값을 반환하도록 수정하고, thread\_get\_recent\_cpu는 현재 스레드의 recent\_cpu값을 반환하도록 수정한다.

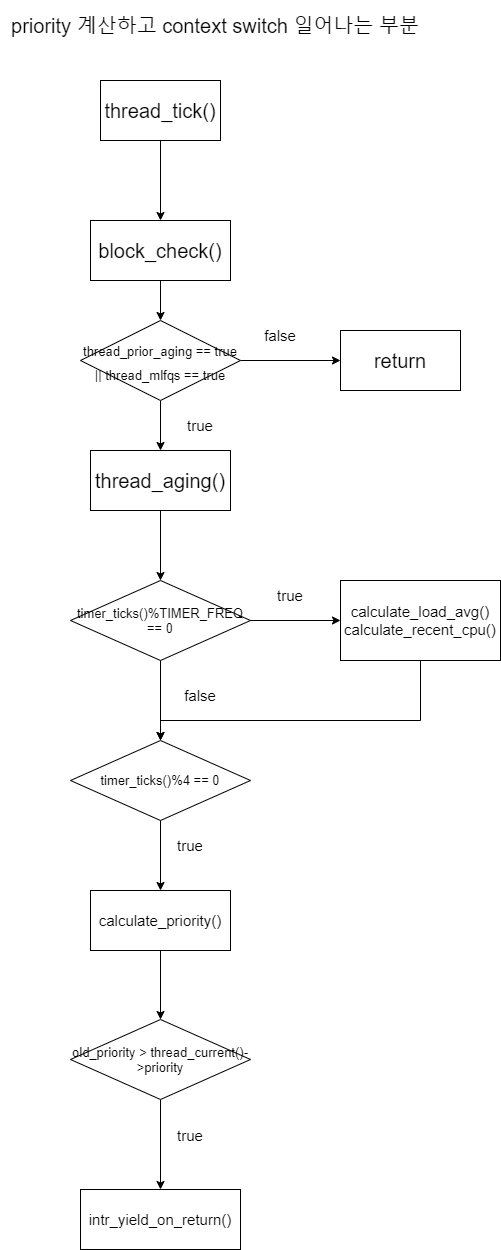
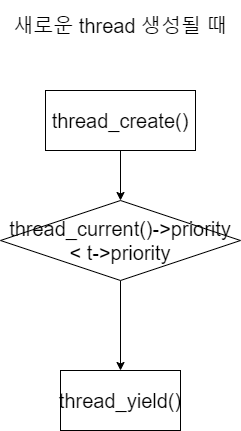
1. **연구 결과**
   1. **Flow Chart**

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

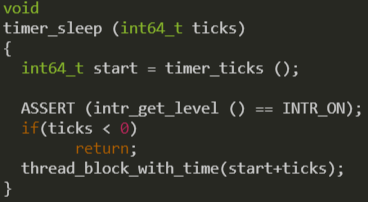
**alarm clock**

****

**priority scheduling**

****

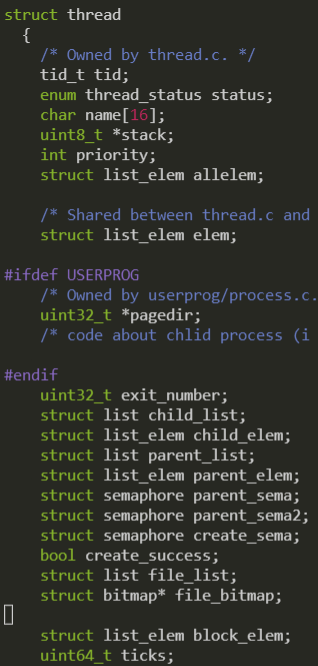
* 1. **제작 내용**
* II. B. 개발 내용의 각 항목에 대하여 실질적으로 구현한 코드의 관점에서 작성 (구현 내용, 알고리즘 등을 명확히 서술할 것)
  + 구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명
* 개발 중 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결한 방식에 대해 설명

****

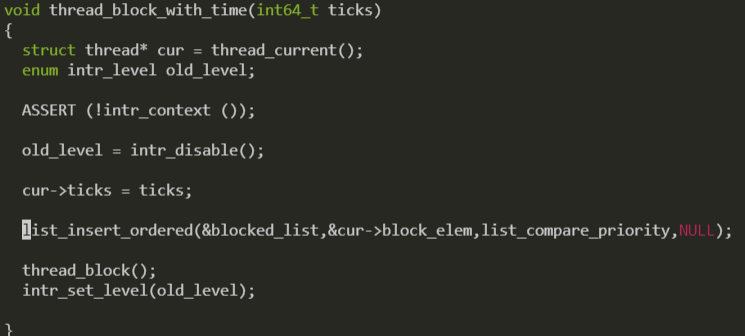
devices/timer.c에서 timer\_sleep함수를 위와 같이 수정했다. thread\_block\_with\_time은 입력된 시간이 될때까지 thread를 block하는 함수이다.



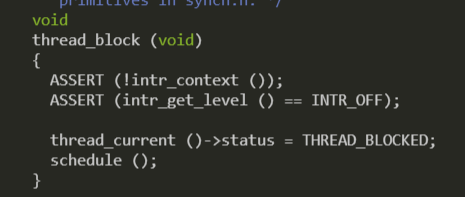
위는 thread.c에 생성한 blocked\_list이다.

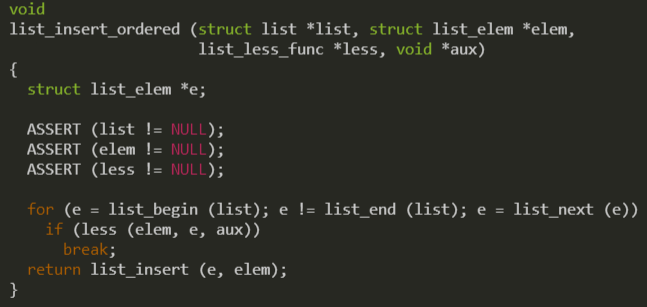


위는 struct thread에 새로 추가한 ticks와 block\_elem이다. block\_elem은 blocked\_list에 스레드를 추가하는데 쓰이고, ticks는 블락된 시간을 저장하는데 쓰인다.

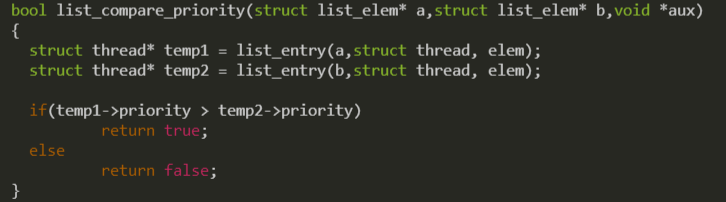


thread.c의 thread\_block\_with\_time함수이다. 위와 같이 우선 blocked\_list에 priority에 맞게 thread를 넣은 후에, thread\_block을 통해서 block한다. thread\_block함수는 현재 스레드의 상태를 THREAD\_BLOCKED로 바꾸고, schedule함수를 호출해서 context switch해준다.

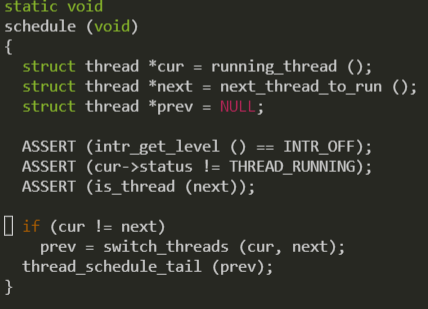




list\_insert\_ordered는 less라는 function에 해당하는 규칙대로 입력된 elem를 list에 삽입한다. 여기서 비교를 위해 추가한 함수가 list\_compare\_priority이고, 다음과 같다.

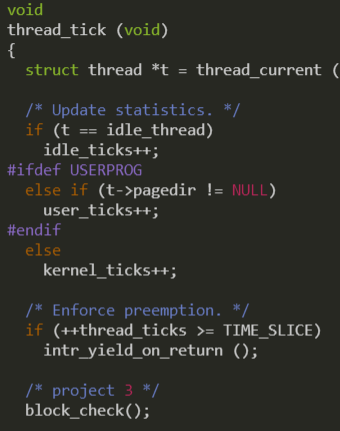


두 thread의 priory를 비교해서 앞의 변수의 priory가 크면true를 아니면 false를 반환한다. 즉, priority에 따른 내림차순으로의 비교를 의미한다.

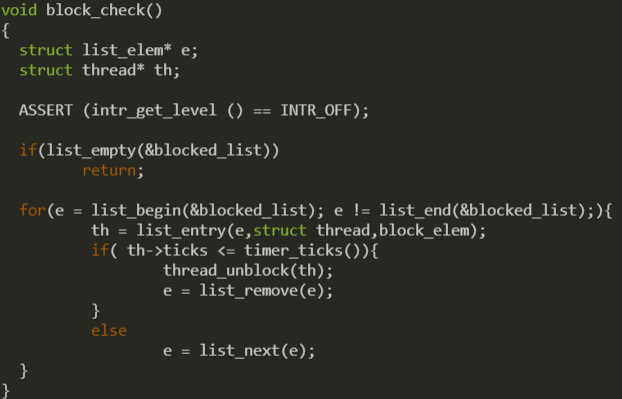


schedule함수는 위와 같이 switch\_thread를 통해서 context switch를 해준다.

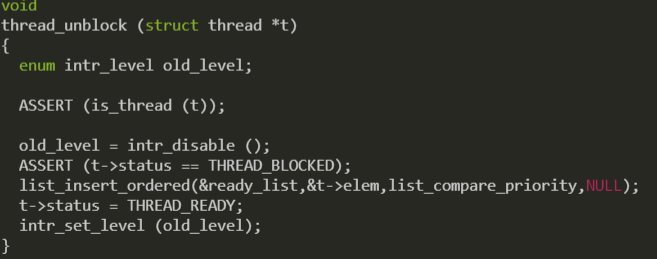
위와 같이 timer에서 thread를 block할 수 있다.



thread를 unblock하기 위해서는 위와 같이 thread\_tick함수에서 매 tick마다 block check()함수를 통해서 unblock해야할 thread를 blocked\_list에서 확인하고 unblock해주게 된다. block\_check함수는 다음과 같다.

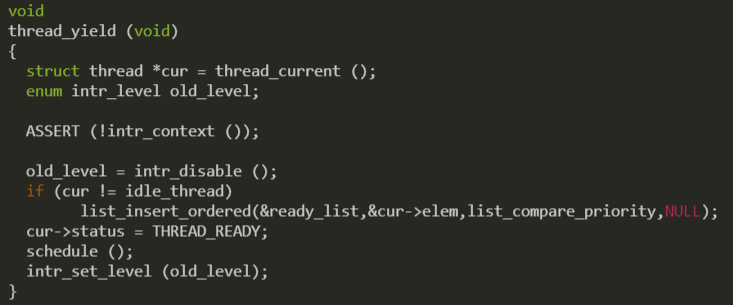


blocked\_list가 비어있으면 그냥 반환한다. 그렇지 않으면, 리스트를 처음부터 따라가면서 리스트의 노드에 저장된 thread의 ticks변수를 확인해서 현재의 ticks을 나타내는 timer\_ticks와 비교후에 현재 ticks이 더 크다면, 해당 thread를 unblock해준다. 그리고 리스트에서 해당 thread를 제거한 후 다음 계속 리스트를 따라가면서 확인한다.

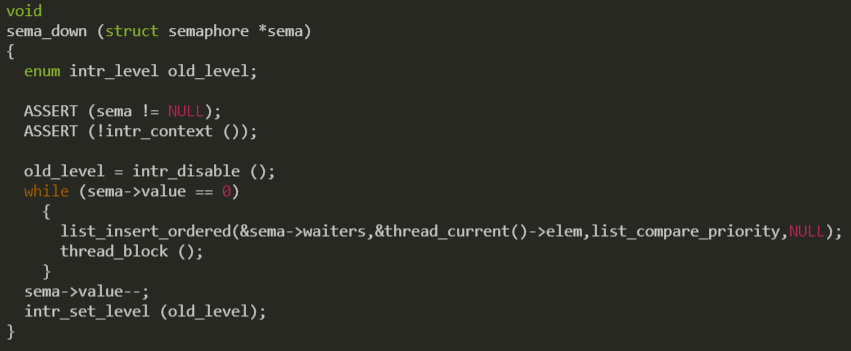


thread\_unblock함수는 위와 같이 unblock할 thread를 입력받아서 ready\_list에 list\_insert\_ordered를 통해서 집어넣는다. 즉 priority에 해당하는 위치에 넣어준다. 그리고 해당 thread의 상태를 THREAD\_READY로 변경한다.



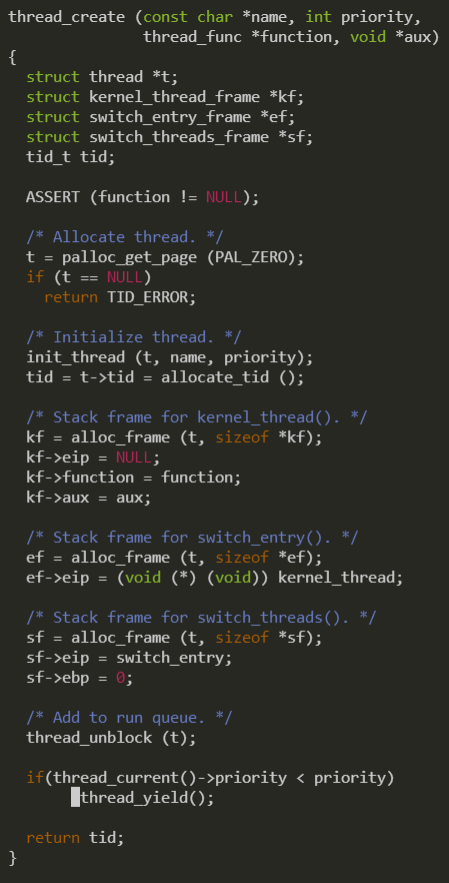


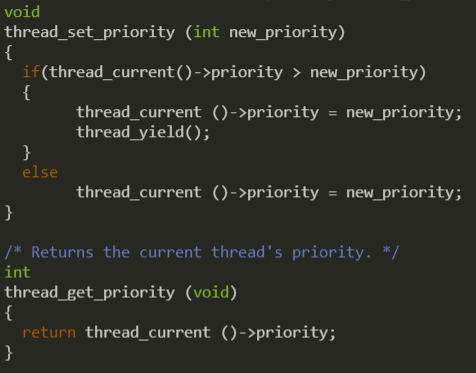
앞에서 설명한 것처럼 위의 두 함수에서 list\_insert\_orderd를 통해서 리스트를 priority에 맞게 추가해줌으로써, priority scheduling을 구현할 수 있다. 또한 sema\_down에 의해서 block되어있다가, sema\_up에 의해 unblock되면 priority가 가장 높은 스레드가 먼저 ready\_list에 추가가 되어야한다. 이를 다음과 같이 구현하였다.



즉, semaphore의 waiter 리스트에 추가할 때부터, priority가 높은 스레드를 앞에 추가하고, 낮은 스레드는 그에 맞게 뒤에 추가하는 방식이다.

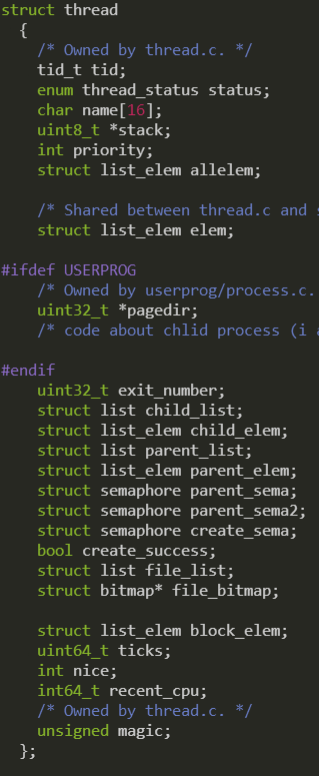
스레드를 새로 생성한 경우에 새로 생성된 스레드의 priority가 현재 thread의 priority보다 크다면 context\_switch를 해준다. 이는 다음 코드와 같다.





추가로 threaed\_set\_priority에서는 현재 스레드의 priority를 새롭게 변경하고, priority가 낮아진 경우에는 thread\_yield를 통해서 context\_switch를 해주고, 높아진 경우에는 가장 높은 priority인 것이 확실하기 때문에 context\_switchf를 하지 않는다.

thread\_get\_priority는 현재 스레드의 priority를 반환한다.





struct thread에서 위와 같이 recent\_cpu와 nice라는 변수를 추가하고 thread.c에는 load\_avg라는 변수를 추가한다. recent\_cpu는 각 스레드가 최근에 cpu를 얼마나 사용했는지를 나타내는 변수이고, nice는 스레드가 다른 스레드에게 priority를 얼마나 양보할 수 있는지를 나타내는 변수이다. load\_avg는 최근에 ready\_list에 존재한 스레드의 평균적인 개수를 의미한다. 이 값들은 priority를 계산하는데 사용된다.



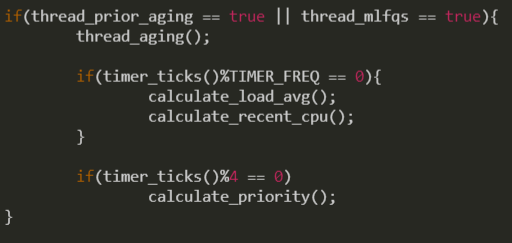
thread.c의 thread\_init함수에서 위와 같이 가장 처음 실행되는 initial\_thread의 nice와 recent\_cpu값을 0으로 초기화한다.

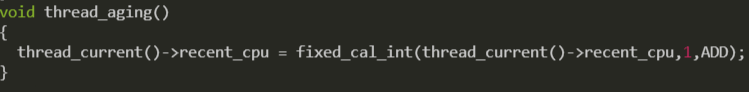
텍스트이(가) 표시된 사진

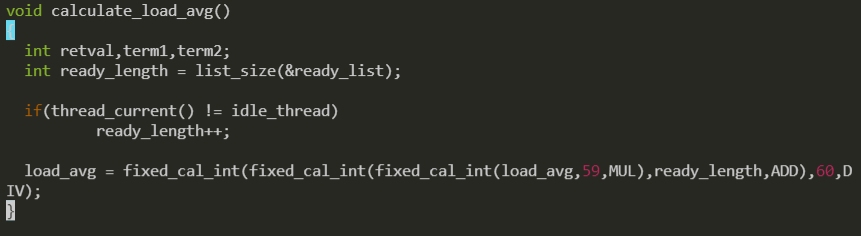
자동 생성된 설명

thread.c의 init\_thread함수에서는 새로 생성된 thread의 nice값과 recent\_cpu값을 그 부모의 nice값과 cpu값으로 초기화한다.

스레드의 우선순위는 cpu를 최근에 많이 사용했다면 낮아지게 된다. 이를 위해서 thread\_aging()등을 이용했다. thread\_aging()은 다음과 같이 현재 thread의 recent\_cpu값을 1증가시킨다. 이 함수는 매 tick마다 실행된다

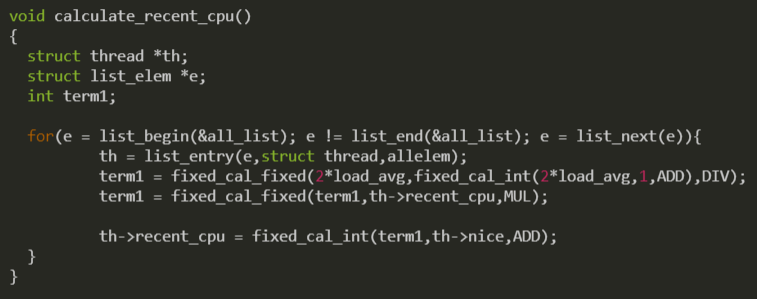
.





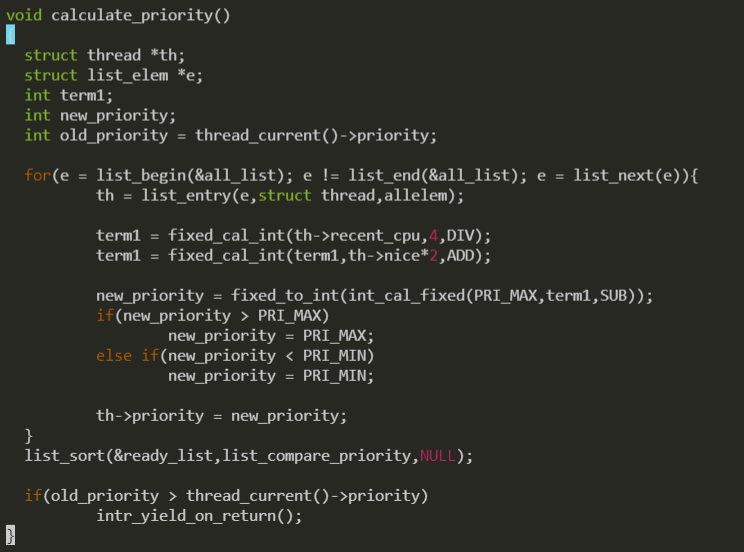
thread.c의 calculate\_load\_avg는 일정 틱마다 실행되는 함수이다. 이는 load\_avg를 다시 계산해주는 함수인데, 현재 ready\_list에 있는 스레드의 개수와 running\_thread의 개수를 ready\_length라는 변수로 하여 계산한다. 이는 다음과 같은 식으로 계산한다.

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



calculate\_recent\_cpu함수도 일정 틱마다 calculate\_load\_avg와 함께 실행되는 함수이다. 이는 모든 스레드의 recent\_cpu값을 다시 계산한다. all\_list라는 모든 스레드를 저장하고 있는 list를 처음부터 따라가면서 각 스레드의 recent\_cpu값을 다음과 같은 식으로 계산한다.

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

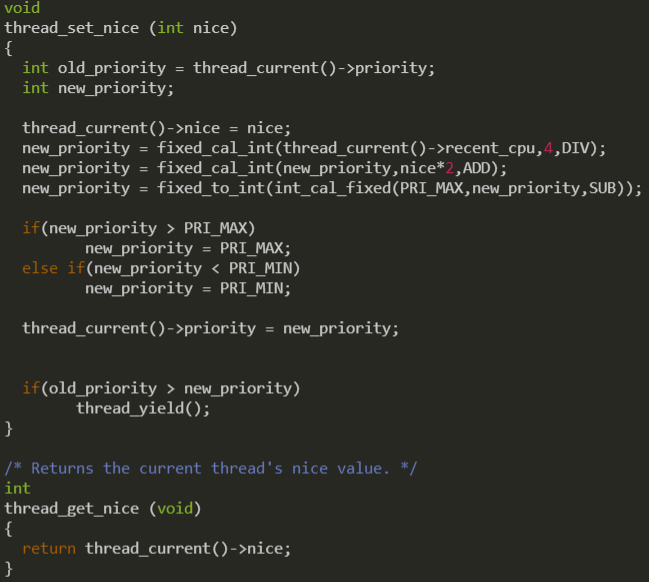


calculate\_priority함수는 4번의 틱마다 한번씩 실행된다. 이는 모든 스레드를 하나씩 따라가면서 다음 식을 통해 priority를 계산해서 변경한다.

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

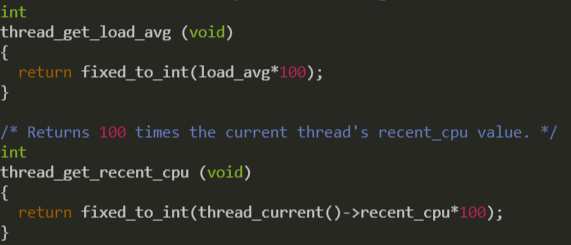
그리고 이렇게 계산된 priority가 0~63범위를 벗어난 경우에 clamp를 해준다. 즉, 63보다 큰 경우에는 63으로 0보다 작은 경우에는 0으로 변경해준다. 그리고 priority가 모두 변경되었으므로 ready\_list를 다시 정렬해야한다. ready\_list를 priority에 따라 내림차순으로 list\_sort함수를 이용해서 정렬해준다. 그리고 현재 스레드의 우선순위가 작아진 경우에 context switch를 해준다.

현재 스레드의 nice값을 설정해주는 thread\_set\_nice함수는 다음과 같다.



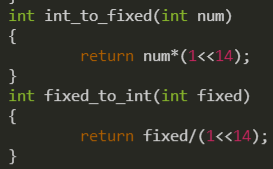
현재 스레드의 nice값을 입력받은 값으로 변경해주고나서, 변경된 nice로 priority를 다시 계산해준후에 priority도 update해준다. 그리고 마찬가지로 낮아진 경우에는 context switch를 해준다.

thread\_get\_nice를 현재 스레드의 nice값을 반환한다.



thread\_get\_load\_avg는 현재 load\_avg값의 100배를 반환하는 함수이다. 그리고 thread\_get\_recent\_cpu는 현재 스레드의 recent\_cput값의 100배를 반환하는 함수이다.

위에서 load\_avg, recent\_cput, priority를 계산할 때 실수 계산이 필요한데, 이를 위해서 fixed\_point연산을 구현했다. fixed\_point 연산은 정수를 고정소수점 실수로 변경하기 위해서 32비트중에서 18비트를 정수부 14비트를 소수부로 이용했다. 정수를 fixed\_point로 변경하려면 정수에 214을 곱해주면 되고, 반대로 고정소수점 실수를 정수로 변경하려면 214를 나눠주면 된다. 이는 다음과 같은 함수로 구현했다.

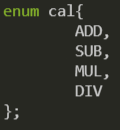


고정소수점으로 나타낸 실수와 정수를 계산하기 위해서 fixed\_cal\_int와 int\_cal\_fixed라는 함수를 추가했다. 이는 각각 “실수 (연산) 정수”, “정수 (연산) 실수”를 나타낸다.

실수와 정수를 더하기 위해서는 정수에 214를 곱해서 더해주면 되고, 뺄 때도 마찬가지이다. 실수와 정수를 곱할 때는 단순히 곱하면 되고 나눌 때도 마찬가지이다.

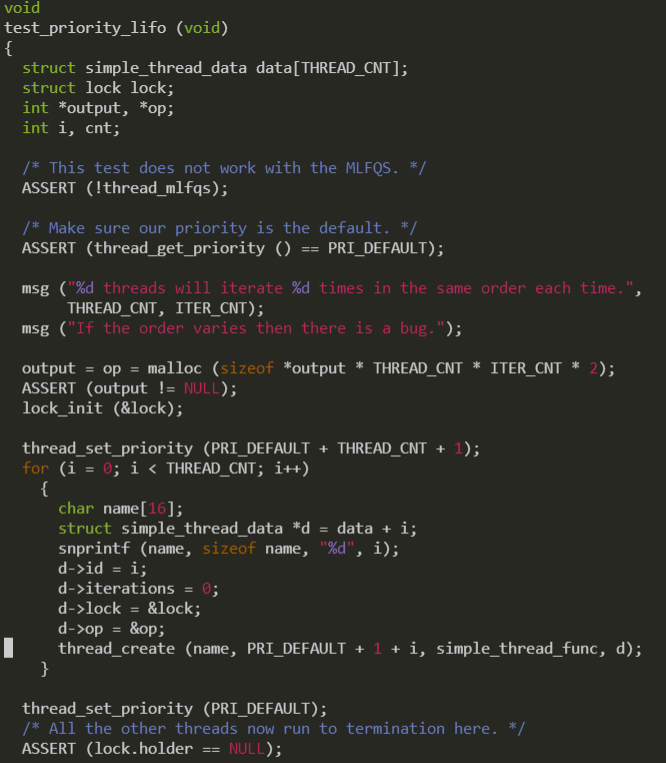
실수와 실수를 연산하기 위해서 fixed\_cal\_fixed라는 함수를 추가했다. 실수끼리의 합과 차는 단순히 더하고 빼면된다. 곱에서는 두 실수를 곱한후 214로 나눠줘야한다. 나누기에서는 두 실수를 나눈후에 214를 곱해주면 된다.

이 함수들에 더하기, 빼기, 곱하기, 나누기 중 어떤 연산을 할지를 결정하기 위해서 enum cal이라는 enum을 이용한다.





* 1. **시험 및 평가 내용**
* priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석
* make check 수행 결과를 캡처하여 첨부

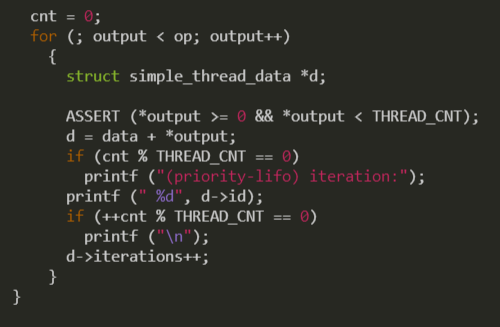


lifo 코드는 위와 같이 우선 현재 스레드의 priority를 새로 생성할 스레드의 priority보다 높게 설정한다. 그리고 PRI\_DEFAULT + 1~ PRI\_DEFAULT + THREAD\_CNT까지의 순서로 차례로 스레드를 생성한다. 그리고 나서 현재 스레드의 priority를 PRI\_DEFAULT로 변경한다. 그러면 context switch가 발생하게 된다. 그러면 thread\_create를 통해서 생성한 스레드들이 모두 차례로 실행된다. 이때 가장 priority가 높은 순으로 실행된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

thread가 실행된 순서대로 thread의 id가 output에 차례로 저장된다.



output의 처음부터 마지막까지 순서대로 읽어들이면 \*output에 저장된 값은 가장 우선순위가 높은 id인 15가 16개 순서대로 저장되어 있으므로 data+\*output은 16번동안 data+15를 나타내고 이의 id는 15이다. 그 다음 output의 배열에는 14가 16개 저장되어있으므로 data+\*output은 16번동안 계속 data+14를 나타내고 이의 id는 14이다. 이렇게 모두 15가 16번 14가 16번, 13이 16번, …, 0이 16번 출력되게 된다. 다음은 priority-lifo.c의 출력결과이다.

