**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

학번 / 이름 : 20161211 / 김동빈

개발 기간 : 2021. 11. 20 ~ 2021. 12. 04

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

Alarm clock: 먼저 현재의 timer\_sleep은 단순히 busy waiting을 이용하여 thread의 sleep을 구현하고 있기 때문에 이를 tick을 이용하여 sleep list에 추가해 두었다가 wakeup tick이 되면 alarm을 통해 깨워주도록 한다.

Priority scheduling: 현재 pintos는 priority와 관계없이 round robin 알고리즘을 이용해 thread를 scheduling하고 있는데 이를 priority를 고려하여 scheduling할 수 있도록 구현한다.

Advanced Scheduler: nice, recent\_cpu, load\_avg등의 값을 이용하여 priority를 계속 변화하여 효율적인 scheduling이 될 수 있도록 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

현재 timer\_sleep함수를 살펴보면 thread가 정해진 tick만큼의 시간이 지날 때까지 RUNNING state와 READY state를 반복하는 방식으로 동작하고 있다. 하지만 이는 지속적으로 CPU를 낭비하고 있으므로 효율적이지 못하다. 따라서 sleep하는 thread들을 따로 sleep\_list에 저장하고 wakeup tick이 되었을 때 이들을 깨워주는 방식으로 thread의 sleep을 구현하여 더 효율적일 수 있도록 할 수 있다.

* 1. Priority Scheduling

현재 pintos는 단순한 round robin 알고리즘을 이용하여 scheduling을 진행하고 있기 때문에 thread의 priority는 전혀 고려하지 않고 있다. 이는 thread\_yield나 thread\_unblock에서 thread를 push\_back을 통해 ready\_list의 맨 마지막으로만 넣기 때문인데, 이를 priority를 고려해 scheduling이 가능하도록 구현하면 더 효율적으로 thread들이 동작할 수 있다. 또한 priority scheduling과정에서 낮은 priority를 가진 thread는 계속 우선순위에 밀려 동작하지 못하는 starvation현상이 발생할 수 있어 이를 해결하기 위한 aging도 구현한다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Multi level feedback queue를 이용하여 ready\_list를 구성하는 방법으로 각 thread가 가지는 nice, recent\_cpu값과 시스템 전체가 가지는 load\_avg를 이용하여 각 thread의 priority가 결정된다. 이렇게 priority를 결정하는 이유는 어떤 thread는 복잡한 연산이 많아 CPU time이 길고 어떤 thread들은 그렇지 않을 수 있다. 이렇게 다양한 thread를 효율적으로 scheduling하기 위함이다. 따라서 위의 값을 이용하여 시스템이 가장 효율적인 작업을 수행할 수 있도록 각 thread의 priority를 새로 구해주는 것이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

Block 상태에 있는 thread를 따로 sleep\_list라는 list를 구현해 해당 list에 넣어준다. 이때 각 thread의 alarm ticks에다가 깨어나야 할 tick시점을 저장하고 thread\_block 함수를 호출한다. 이후 timer\_interrupt때마다 sleep\_list를 탐색하여 깨어나야 할 thread들을 list에서 제거하면서 thread\_unblock 함수를 호출해 줌으로써 깨울 수 있다.

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

ready list에 현재 running thread보다 더 높은 priority의 thread가 들어오는 경우 현재 running thread의 수행을 멈추고 해당 thread를 실행시켜야 한다. 이 경우 thread\_yield를 통해 현재 running thread를 ready\_list에 넣고 schedule() 함수를 호출해 다음 가장 높은 priority의 thread가 수행될 수 있도록 한다.

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

nice: 각 thread가 다른 thread에 어떤 영향을 미치는지를 의미한다. -20 ~ 20 사이의 정수 값을 가지며 nice > 0 인 경우에는 자신의 priority를 낮춰 CPU를 다른 thread에 양보하도록 한다. 반대로 nice < 0인 경우에는 자신의 priority를 높여 CPU를 더 많이 차지하도록 한다. nice가 0인 경우에는 다른 thread에 영향을 미치지 않는다.

recent\_cpu: 현재 thread의 다음 CPU time을 exponential weighted moving average방법을 이용해 예측한 값으로 이전의 CPU time history에 가중치를 두어 다음 CPU time을 계산한다. 이때 가장 최근의 CPU time일수록 높은 가중치를 가지게 된다.

load\_avg: thread가 아닌 시스템 전체에 귀속된 값으로 최근 1분 동안 수행가능한 thread의 평균 개수를 의미하며 이전의 load\_avg와 ready\_threads 값을 이용하는데 이때 ready\_threads는 현재 시점에서의 ready 혹은 running 상태의 thread의 개수를 의미한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2021. 11. 25 ~ 2021. 11. 27: alarm clock 구현

2021. 11. 28 ~ 2021. 11. 29: priority scheduling 구현

2021. 11. 30 ~ 2021. 12. 04: advanced scheduling 구현

* 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

1. alarm clock

먼저 각 thread가 sleep되었을 때 일어나야하는 tick을 의미하는 alarm\_ticks를 thread구초제에 추가해준다. block thread를 저장할 sleep\_list를 thread.c에 추가해주어야 한다. 그리고 timer.c의 timer\_sleep 함수에서 while문 대신에 새로 thread\_sleep이라는 함수를 추가해서 현재 thread를 sleep\_list에 삽입한다. 이 때 sleep\_list에 alarm\_ticks에 대해 오름차순을 위지하기 위해 thread의 alarm\_ticks를 비교하는 alarm\_ticks\_comp 함수를 구현하고 list\_insert\_ordered 함수를 이용해 sleep\_list에 현재 thread를 집어넣는다.

또한 timer\_interrupt 함수가 호출될 때 마다 깨어나야 할 thread들을 sleep\_list에서 제거해주어야 하는데 이를 위해 thread\_alarm이라는 함수를 추가한다.

2. priority scheduling

먼저 thread가 ready\_list에 들어있는 thread 중 가장 priority가 높은 thread일수록 front에 있어야 한다고 생각했고 thread\_yield 함수와 thread\_unblock 함수에서 ready\_list에 thread를 list\_push\_back 함수를 이용하여 삽입하고 있는데 이를 thread를 priority 순으로 정렬해주기 위해 priority\_comp함수와 list\_insert\_ordered함수를 이용하여 ready\_list가 항상 priority의 내림차순으로 정렬된 상태를 유지하도록 한다.

또한 thread\_create함수에 의해 새로 thread가 생성될 때 새로운 thread의 priority가 현재 thread의 priority보다 높은 경우 preemptive하게 새로운 thread가 수행되어야 하므로 thread\_yield 함수를 호출하여 scheduling을 해준다.

그리고 aging의 경우에는 일단 thread\_prior\_aging이라는 flag를 추가해 aging이 진행되고 있음을 알리는 것이 추가적으로 구현되어야 하고 advanced scheduling이 아닌 일반 aging의 경우에는 ready\_list에 포함된 thread의 prioirity를 1씩 증가시킨다.

3. advanced scheduling

Advanced scheduling을 위해서는 첫번째로 thread 구조체 안에 nice와 recent\_cpu를 추가하고 thread.c 내에 전역 변수 load\_avg를 추가한다. thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu 등의 함수가 현재는 구현되지 않은 상태인데 pintos manual에 따라 이의 구현이 필요하고 fixed point 연산을 위한 함수들도 따로 구현하여 새로운 fixed\_point.h라는 파일을 만들어 준다.

또한 update\_priority 함수를 구현해 pintos manual에 명시되어 있는 것과 같이 모든 thread의 priority를 업데이트 해준다. 마찬가지로 update\_recent\_cpu함수는 모든 RUNNING, READY, BLOCKED 상태의 thread의 recent\_cpu 값을 업데이트 한다. Update\_load\_avg함수는 load\_avg 값을 업데이트 한다.

그리고 timer.c의 timer\_inturrupt 함수에서 pintos manual에 따라 매 interrupt마다 현재 thread의 recent\_cpu는 1 증가시키고, 1초마다 load\_avg와 recent\_cpu를 업데이트 하고, 매 4 tick마다 priority를 업데이트 해준다.

1. **연구 결과**
   1. **Flow Chart**

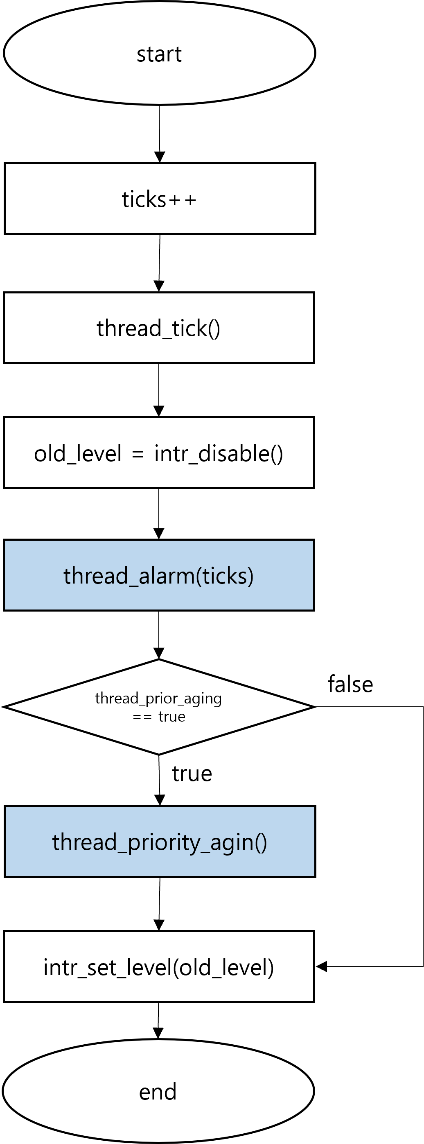
* II. B. 개발 내용의 각 항목에 대하여 Flow Chart 작성  
  (추가구현에 대해서는 flow chart를 작성하지 않아도 됨)

timer\_sleep()의 flow chart는 아래와 같다.

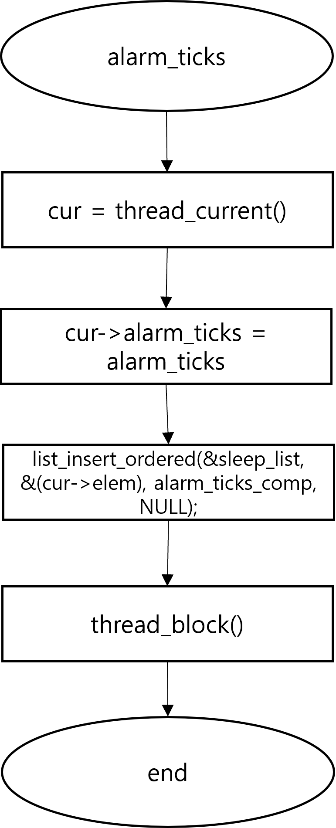
텍스트이(가) 표시된 사진

자동 생성된 설명

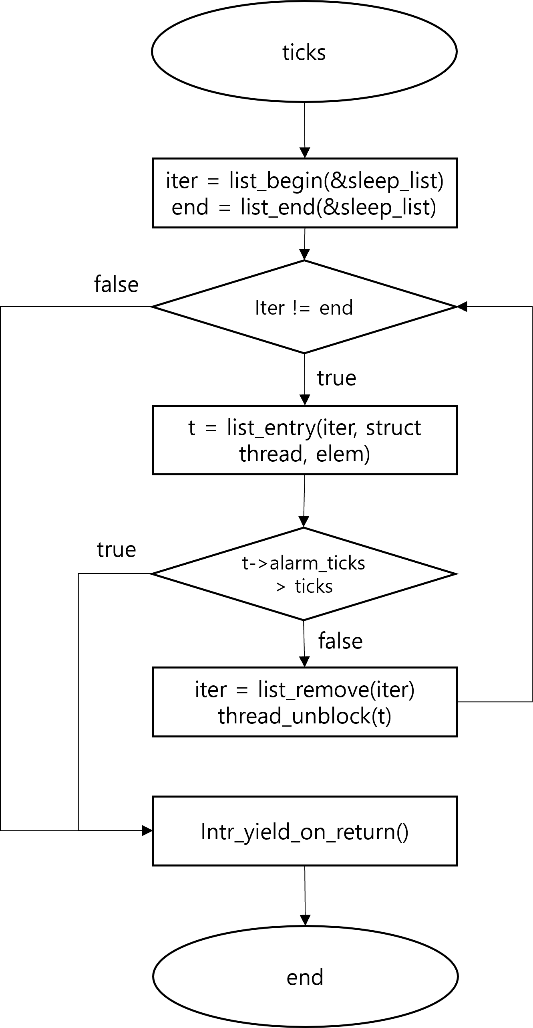
timer\_interrupt()의 flow chart는 아래와 같다. (advanced scheduling을 포함하지 않은 버전이다.)



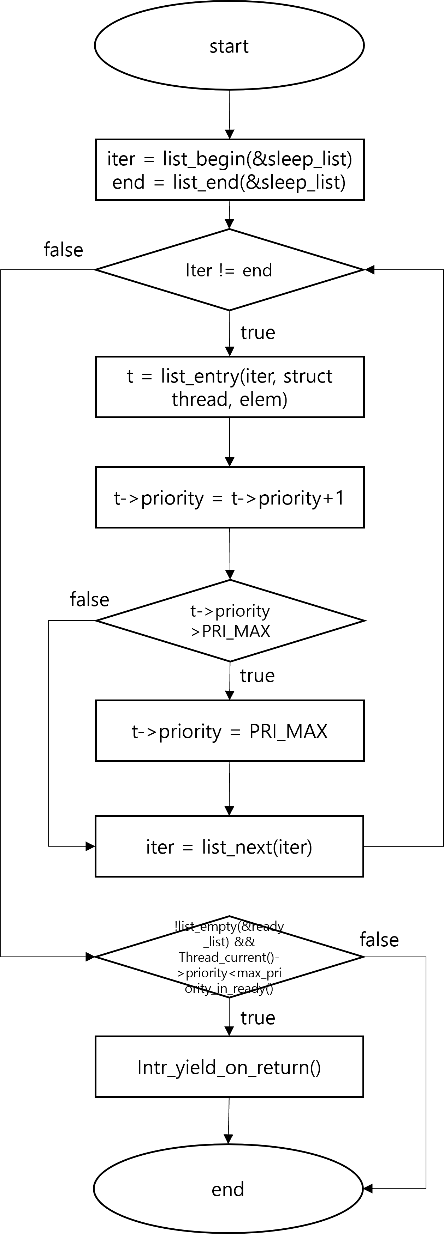
thread\_sleep()의 flow chart는 아래와 같다.



thread\_alarm()의 flow chart는 아래와 같다.



thread\_priority\_aging()의 flow chart는 아래와 같다.



* 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. alarm clock

먼저 timer.c의 timer\_sleep 함수에서 기존의 busy waiting을 없애고 thread\_sleep이라는 함수를 호출한다. 이때 thread\_sleep 함수는 현재 tick을 저장한 start와 timer\_sleep의 인자 ticks를 합한 start+ticks를 인자로 넘겨받는데 이는 현재 thread가 깨어날 alarm\_ticks를 의미한다. 그리고 alarm\_ticks\_comp함수와 list\_insert\_ordered 함수를 이용해 현재 thread를 sleep\_list에 삽입하는데 이때 sleep\_list는 alarms\_ticks의 오름차순 즉 일찍 깨어나야 하는 thread 순으로 정렬된 상태를 유지한다. 또한 thread\_block함수를 호출해 현재 thread를 BLOCKED상태로 바꾸고 새로 scheduling 해준다.

그리고 timer\_interrupt 함수가 호출될 때 마다 깨어나야 할 thread들을 sleep\_list에서 제거해주어야 하는데 이를 위해 thread\_alarm이라는 함수를 호출한다. Thread\_alarm함수는 현재 ticks를 인자로 받는데 sleep\_list내부의 thread를 탐색하면서 alarm\_ticks가 ticks보다 작거나 같은 경우는 list\_remove함수를 이용해 sleep\_list에서 제거하고 thread\_unblock 함수를 이용해 ready\_list로 집어넣어준다. 이때 sleep\_list의 thread는 alarms\_ticks에 대한 오름차순으로 정렬되어 있으므로 alarm\_ticks가 ticks보다 큰 경우를 만나게 되면 break해주면 된다.

2. priority scheduling

먼저 thread가 ready\_list에 들어있는 thread 중 가장 priority가 높은 thread일수록 front에 있어야 하므로 thread\_yield 함수와 thread\_unblock 함수에서 ready\_list에 thread를 priority 순으로 정렬해주기 위해 priority\_comp함수와 list\_insert\_ordered함수를 이용하여 ready\_list가 항상 priority의 내림차순으로 정렬된 상태를 유지하도록 했다.

해당 과정을 수행하면서 ready\_list에 삽입하는 경우에 list\_insert\_ordered를 이용하지 않고 next\_thread\_to\_run 함수에서 ready\_list에서 하나의 thread를 꺼낼 때 priority에 대한 내림차순으로 정렬을 해준 후 list\_pop\_front를 하려고 했으나 aging과 advanced scheduling을 구현하면서 현재 thread와 ready\_list의 가장 큰 priority를 비교해야하는 경우가 발생하여 이를 위해서는 매번 정렬을 하는 것보다 ready\_list에 thread를 삽입할 때 list\_insert\_ordered 함수를 사용해 정렬된 상태를 유지하는 것이 효율적이라고 판단하였다.

그리고 thread\_create함수에서 새로운 thread가 생성될 때 해당 thread의 priority가 현재 thread의 priority보다 높은 경우 새로 생성된 thread가 수행되어야 하므로 thread\_yield 함수를 호출하여 scheduling을 해주었다. 마찬가지로 thread\_set\_priority함수에서도 현재 thread에 새로운 thread를 부여한 경우에 만약 readt\_list에 더 높은 priority를 가진 thread가 있는 경우 해당 thread가 수행되어야 하므로 해당 경우에 thread\_yield를 호출하였다. ready\_list에 현재 thread의 priority보다 높은 priority를 가진 thread의 존재 유무는 ready\_list의 thread가 priority의 내림차순으로 정렬되어 있으므로 front의 thread priority와 현재 thread의 priority를 비교해주면 된다. 이때 ready\_list의 front의 thread priority를 구하기 위해 max\_priority\_in\_ready함수를 추가적으로 구현했다.

aging의 경우에는 일단 init.c에서 parsing 과정에서 “-aging”이 있다면 thread\_prior\_aging을 true로 하여 aging이 진행되고 있음을 알렸고 thread\_priority\_aging 함수에서 ready\_list에 포함된 thread의 prioirity를 1씩 증가시킨다. 이때 priority는 PRI\_MAX를 초과하면 안 되므로 증가시킨 결과가 PRI\_MAX보다 큰 경우에는 priority를 PRI\_MAX로 설정해주었다. 또한 timer.c의 timer\_interrupt에서 thread\_prior\_aging이 true인 경우 thread\_priority\_aging 함수를 호출해 interrupt마다 priority를 증가시켜주었다.

priority\_sema test를 통과하기 위해서는 synch.c의 sema\_down함수와 sema\_up함수의 수정이 필요했다. sema\_down함수에서도 priority\_comp 함수와 list\_insert\_ordered함수를 이용해 sema\_waiters내의 thread가 priority의 내림차순으로 정렬되도록 해주었고, sema\_up 함수에서는 unblock된 thread의 priority가 현재 thread보다 높을 수 있으므로 새로 scheduling 하기 위해 thread\_yield함수를 호출했다.

마지막으로 alarm clock을 위해 구현했던 thread\_alarm에서도 thread\_unblock함수를 호출하는데 깨어난 thread들 중 현재 thread보다 priority가 높은 경우가 있을 수 있으므로 thread\_yield를 호출하려 했으나 interrupt를 disable시킨 도중이므로 interrupt.c의 intr\_yield\_on\_return함수를 호출해 줌으로써 interrupt.c에서 yield\_on\_return값을 true로 바꾸어주어 intr\_handler함수 내부에서 thread\_yield가 진행될 수 있도록 하였다.

3. advanced scheduling

먼저 pintos manual 95p에 따라 fixed point 연산을 위한 함수들을 따로 구현하기 위해 fixed\_point.h라는 파일을 만들고 F는 1<<14로 define한 다음 아래와 같은 함수들을 구현해 주었다.

int\_to\_fp(int n): 정수를 인자로 받아 fixed point number를 만들기 위해 F를 곱한 값을 반환한다.

round\_toward\_zero(int x): fixed point number x를 F로 나누어 정수를 반환한다.

round\_to\_nearest(int x): fixed point number x가 0보다 크거나 같으면 (x+F/2)/F를 0보다 작으면 (x-F/2)/정를 만들어 정수를 반환한다.

fp\_add\_fp(int x, int y): 두 fixed point number x, y를 입력 받아 x+y를 반환한다.

fp\_sub\_fp(int x, int y): 두 fixed point number x, y를 입력 받아 x-y를 반환한다.

fp\_add\_int(int x, int n): fixed point number x와 정수 n를 입력 받아 x+n\*F를 반환한다.

fp\_sub\_int(int x, int n): fixed point number x와 정수 n를 입력 받아 x-n\*F를 반환한다.

fp\_add\_fp(int x, int y): 두 fixed point number x, y를 입력 받아 ((int64\_t)x\*y)/F를 반환한다.

fp\_mul\_int(int x, int n): fixed point number x와 정수 n를 입력 받아 x\*n를 반환한다.

fp\_div\_fp(int x, int y): 두 fixed point number x, y를 입력 받아 ((int64\_t)x\*F)/y를 반환한다.

fp\_div\_int(int x, int n): fixed point number x와 정수 n를 입력 받아 x/n를 반환한다.

update\_priority 함수에서는 모든 thread의 priority를 업데이트 해준다. 이때 priority계산을 위해 new\_priority함수를 구현했는데 new\_priority에서는 priority를 구하는 공식과 위의 fixed\_point.h의 함수를 이용해서 새로운 priority를 계산하고 이를 반환한다. 이때 새로운 priority가 PRI\_MAX보다 크면 PRI\_MAX를, PRI\_MIN보다 작다면 PRI\_MIN을 반환한다. 이후 업데이트된 priority에 의해 ready\_list가 priority의 내림차순으로 정렬되지 않았을 수 있으므로 list\_sort함수를 이용해 정렬해주고, 마지막으로 현재 thread보다 ready\_list의 front의 thread priority가 더 클 수 있으므로 max\_priority\_in\_ready함수를 호출해 해당 경우에 thread\_yield 함수를 호출하고자 했으나 interrupt를 disable한 상황이므로 intr\_yield\_on\_return 함수를 호출했다.

마찬가지로 update\_recent\_cpu함수를 이용해 모든 RUNNING, READY, BLOCKED 상태의 thread의 recent\_cpu 값을 업데이트 한다. 이는 status가 THREAD\_DYING이 아닌 thread의 recent\_cpu를 update하도록 하였다. 이때 idle\_thread의 경우에도 recent\_cpu 업데이트에서 제외시켜주고 new\_recent\_cpu함수를 이용해 새로운 recent\_cpu값을 계산해 업데이트를 진행한다. new\_recent\_cpu함수는 recent\_cpu 공식과 fixed\_point.h의 함수를 이용해 새로운 recent\_cpu를 계산해 반환한다.

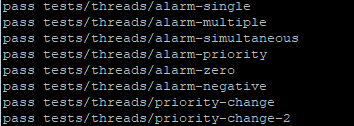
update\_load\_avg함수에서는 new\_load\_avg함수를 이용해 새로운 load\_avg 값을 업데이트 한다. 이때 현재 load\_avg와 ready\_threads가 필요한데, ready\_threads는 현재 READY 상태이거나 RUNNING상태인 thread의 개수를 의미하므로 ready\_list의 원소의 개수에 현재 thread가 idle\_thread가 아닌 경우에만 1을 더한다. 마찬가지로 new\_load\_avg에서도 load\_avg공식과 fixed\_point.h의 함수를 이용해 새로운 load\_avg를 계산해 반환한다.

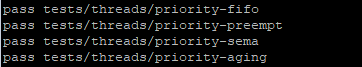
실제로 위의 함수들은 timer.c의 timer\_inturrupt 함수에서 호출되는데 pintos manual에 따라 매 interrupt마다 thread\_mlfqs가 true인 경우 현재 thread의 recent\_cpu는 1 증가시키는 작업이 필요하다. 이는 update\_periodic\_recent\_cpu 함수를 통해 현재 thread가 idle\_thread가 아닌 경우 fd\_add\_int함수를 통해 현재 thread의 recent\_cpu값에 1을 더해준다. 또한 1초마다 load\_avg와 recent\_cpu를 업데이트 해야 하는데 이는 ticks를 TIMER\_FREQ로 나눈 나머지가 0일 때마다 update\_load\_avg와 update\_recent\_cpu함수를 호출함으로써 해결한다. 마지막으로 매 4 tick마다 priority를 업데이트 하기 위해 updupd\_priority 함수를 호출해준다.

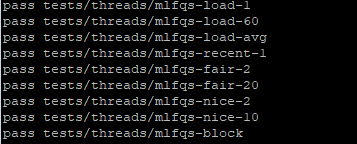
또한 위의 fixed\_point.h의 함수를 이용해 thread\_set\_nice, thread\_get\_nice, thread\_get\_load\_avg, thread\_get\_recent\_cpu등의 함수를 구현해주어야 하는데, 먼저 thread\_set\_nice는 인자로 들어오는 새로운 nice값으로 현재 thread의 nice 값을 설정해주고 pintos manual대로 새로운 priority를 구하는데 new\_priority함수를 이용해준다. 그리고 현재 thread보다 ready\_list의 front의 thread priority가 더 클 수 있으므로 max\_priority\_in\_ready함수를 호출해 해당 경우에 thread\_yield 함수를 호출했다. Thread\_get\_nice는 단순히 현재 thread의 nice값을 반환하였고, thread\_get\_load\_avg, thread\_get\_recent\_cpu 함수는 pintos manual에 명시된 대로 fp\_mul\_int함수를 이용해 각각 load\_avg와 현재 thread의 recent\_cpu 값에 100을 곱한 값을 반환했다.

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

make check 수행 결과는 아래와 같다.







Priority\_lifo.c의 경우에 먼저 output과 op는 처음에는 같은 주소 공간을 가리키도록 메모리를 할당받는다. 그리고 thread\_set\_priority 함수를 이용하여 현재 thread의 priority를 PRI\_DEFAULT + THREAD\_CNT + 1로 설정해준다. 이후 0부터 15까지 하나씩 thread\_create를 통해 새로운 thread를 생성하는데 이들의 priority는 PRI\_DEFAULT+1+i로 현재 thread보다 다 낮다. 따라서 16개의 thread가 모두 생성이 된다. 이후 다시 thread\_set\_priority 함수를 이용하여 현재 thread의 priority를 PRI\_DEFAULT로 설정해주면 앞서 생성해준 새로운 thread들의 priority가 더 높게 되고 thread\_set\_priority 함수 내부에서 thread\_yield가 호출되도록 구현하였기 때문에 thread\_yield에 의해 가장 priority가 높은 thread가 수행되게 된다. 여기서는 15번 thread가 된다.

15번 thread는 simple\_thread\_func을 수행하게 되는데 op가 가리키는 위치에 자신의 id 15를 저장하고 thread\_yield를 호출한다. 그러나 여전히 가장 높은 priority를 가진 thread는 자기 자신이므로 다음 for문의 다음 iteration을 수행하면서 위의 과정을 반복하다 종료된다. 15번 thread가 수행을 종료하면 다음으로 priority가 높은 14번 thread가 수행을 시작하는데 15번 thread와 마찬가지로 매 iteration마다 thread\_yield가 호출되지만 가장 높은 priority를 가진 thread가 자기 자신이므로 종료될 때까지 iteration을 반복한다. 위의 과정을 0번 thread까지 마무리하면 output에는 순서대로 15가 16번, 14가 16번, … 0이 16번의 순서로 저장이 되어있음을 알 수 있다.

이제 마지막으로 output이 op가 될 때까지 output에 저장된, 즉 simple\_thread\_func에서 op에 의해 저장된 값들을 살펴보면 15부터 0까지 16번씩 나타나게 되어있으므로 아래와 같은 결과를 얻을 수 있다.

