**Pintos Project 3: Threads**

담당 교수 : 김영재

학번 / 이름 : 20171700 / 최재원

개발 기간 : 2021/11/1 ~ 2011/11/6

1. **개발 목표**

이번 프로젝트에서 구현하고자 하는 내용은 alarm clock, priority scheduling 그리고 advanced scheduler 이렇게 3가지이다. Pintos 의 기존 scheduling 방식은 다른 thread가 schedule 되었을 때, 자신이 사용할 수 있을때가지 계속 Spin-wait을 하며 기다리는 busy waiting 방식을 사용하기 때문에 cpu 효율면에서 좋지 못하다. 따라서 thread의 state를 바꾸고, priority에 따라서 Scheduling하는 등의 앞 서 말한 방법들을 사용하여 Pintos 내에서의 thread scheduling을 좀 더 효율적으로 바꾸고자 한다.

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

현재 Pintos는 위에 말한대로 busy waiting 방식을 사용하기 때문에, thread들이 ready\_list에 들어가 일정 tick 동안 반복문안에서 아무 일도 하지않고 Spin-wait을 하므로 간편하지만 쓸데없이 Cpu 자원을 사용하여 Cpu 효율은 떨어지는 상태이다. 따라서 이를 개선하기 위해 현재 schedule 되어 있는 thread 외의 thread들을 sleep 상태로 바꾼 후 list에 저장하는 방식을 사용한다. Sleep list에 저장되어 있는 Thread들은 각자 자신이 다시 ready 상태로 되기까지의 tick을 같이 저장하고 있다. 이런 과정을 통해서 thread는 기존과 다르게 Cpu 자원 낭비를 최소화 하여 보다 효율적인 scheduling을 수행할 수 있을 것이다.

* 1. **Priority Scheduling**

1번에서 구현한 scheduling 방식은 우선순위를 고려하지 않고 ready list의 가장 마지막에 계 속해서 thread들이 순차적으로 들어가기 때문에 thread들의 average turn around time이 증가할 가능성이 있다. 따라서 이를 개선하기 위해서 우선순위에 따른 scheduling을 진행하며, 추가적으로 preemptive한 scheduling에서 발생할 수 있는 낮은 우선순위의 threa들에 대한 Starvation을 어느정도 해결해 주기 위해서 aging 방법을 사용한다.

* 1. **Advanced Scheduler**

1번과 2번에서 구현한 scheduler는 priority에만 의존하기 때문에, average turn around time은 감소할 수 있지만 average response time은 증가할 가능성이 있다. Aging은 단순히 priority를 1씩 증가시켜주기 때문에 이를 해소할 완전한 방법이 되지 못한다. 따라서 starvation에 따른 average response time 문제를 해결해 주기 위해서 Multi-level feedback queue 방법을 사용한다. MLFQ는 multi level 관점에서 여러개의 Ready queue가 존재한다는 특징과, feedback 측면에서 Priority를 실시간으로 조절한다는 특징이 있다. 우리는 여기서 두번째 특징에 초점을 맞추어 구현을 하고자 한다.

* 1. **개발 내용**

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

Sleep 상태에 있는 Thread들을 List에 저장하고, 해당 thread들을 다시 깨울 때 새로 정의한 wake\_thread() 함수를 호출한다. Timer\_interrupt() 함수에서 해당 함수를 호출하며, 각 Thread들은 구조체 형태로써, 이에 대해 wake\_t라는 변수를 따로 선언하여 Sleep 상태를 유지할 시간을 저장한다. 따라서 wake\_thread() 함수 내에서 각각의 Thread들의 wake\_t에 따라서 다시 깨운 후 ready\_list에 옮기는 방법대로 구현한다면, sleep list에 있는 Thread는 기다리는 동안 추가적인 Cpu 자원을 사용하지 않고 wake up 되어야 할 때 정확히 깨어날 수 있을 것이다.

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

Priority scheduling은 preemptive한 scheduler를 구현하는 것이기 때문에, Cpu 가 현재 처리하고 있는 Thread 보다 높은 Priority를 가진 Thread가 ready list에 들어온다면 현재 연산중인 thread를 멈추고 우선순위가 더 높은 thread를 먼저 처리해주어야 할 것이다. 따라서 현재 연산중이였던 thread는 다시 ready list로 돌아가며, Thread들의 우선순위에 따라서 재배열 될것이다.

1. **Advanced Scheduler에서 priority 계산에 필요한 각 요소를 서술.**

해당 스케줄러에서 우선순위 계산에 필요한 요소는 Nice value, recent\_cpu, load\_avg 그리고 PRI\_MAX(63) 정도 이다.

우선 Nice는 Niceness, 즉 각 thread가 자신이 아닌 다른 thread에게 CPU를 얼마나 잘 yield 하는지를 나타낸다. I/O bound인 Thread일 수록 해당 nice value가 작을 것이고, 다른 Thread에게 cpu를 더 잘 yield할 것이다. 반대로 cpu intensive한 thread들은 해당 nice value가 클 것이고, 따라서 nice value가 낮을수록 우선순위를 높게 측정해야 할 것이다. Nice value의 범위는 -20 에서 20이다.

다음으로, Recent\_cpu는 thread가 cpu를 얼마나 많은 시간 사용했는지를 나타낸다. 위와 같은 맥락으로 recent\_cpu값은 cpu사용 시간이 길 수록 커지며, 따라서 우선순위는 낮아지게 된다. 여기서 중요한 것은 최근에 사용된 cpu 작업일수록 가중치가 붙는 다는 것이다. 즉, 가장 최근 작업에 대해서는 그대로 반영되고 LRU가 아닐수록 그 값은 점차 감소한다. 이는 Exponentially weighted moving average(지수가중 이동평균 방식)을 따른다고 할 수 있다. Recent\_cpu를 계산하는 식은 이로부터 유도되는데, 다음 식을 살펴보며 간단하게 설명하고자 한다.

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

load\_avg는 현재 Thread들 중 Ready state인 thread들의 개수 평균이다. 이 식에서 recent\_cpu 앞에 곱해지는 (2 \* load\_avg) / (2 \* load\_avg + 1)를 지수 가중 이동평균식에서 사용되는 부패율이라고 생각했을 때, 그 값이 0, nice, a\*nice + nice, a^2\*nice + a\*nice 와 같이 전개되는 것을 알 수 있고, load\_avg에 따라서 각 thread들의 recent\_cpu 값의 부패율 구할수도 있을 것이다.

Load\_avg값은 recent\_cpu와 마찬가지로 지수 가중 이동평균에 따라서 값이 변하고, 다만 recent\_cpu 값은 각 Thread 별로 존재하지만, load\_avg는 System-wide 값으로써, 시스템 내에서 동일한 값을 가진다. 결과적으로 advanced scheduler에서의 priority 값은 다음과 같이 정의된다.

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

PRI\_MAX는 핀토스 내에서의 우선순위 최대값인 63이다. Load\_avg 값과 recent\_cpu갑은 매 초마다 renew되며, priority 값은 4초마다 갱신된다. 우선순위를 계산하는데 있어서 추가적으로 고려해야 할 부분은 실수 계산이다. Pintos는 소수점 연산을 지원하지 않기 때문에, 우리는 모든 실수 연산을 fixed-point real arithmetic을 사용하여 처리해야 한다. 이를 구현하는 방법은 실수에 2^14를 곱하여 고정 소수점 방식으로 표현하는 것이다. 즉, 7 + 3.14와 같은 연산을 7\*2^14 + 3.14와 같은 식으로 계산해야 한다. 모든 계산에 대해서 함수로 정의하여도 되지만, 이번 프로젝트에서는 실수와 정수의 덧셈, 정수와 실수의 뺄셈, 실수와 실수의 곱셈과 나눗셈에 대한 함수를 추가로 정의해주어 사용하였다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

|  |  |
| --- | --- |
| 기간 | 내용 |
| 11/2~11/5 | 중간고사가 연기되는 일이 생겨 본 기간동안  11/2~11/4 :Alarm clock, priority schedule  11/5 : advanced schedule을 진행하였다. |
| 11/6 | 보고서 작성 |

* 1. **개발 방법**

**1) Alarm Clock**

Busy waiting 방법을 고치기 위해서 timer.c 파일을 수정한다. 본 코드에서 timer\_sleep() 함수 내에 있는 반복문을 지우고 thread\_block을 사용하여 해당 Thread를 Sleep상태로 만들고, 다시 Timer interrupt 함수는 매 초마다 호출되므로, 해당 함수에서 새로 정의한 wake\_thread 함수를 호출하고, 함수 내에서 thread\_unblock 함수를 사용하여 다시 thread를 깨워준다. 이 때 thread 구조체에 wake\_t를 새로 정의하여 사용한다. 이 wake\_t변수는 각 thread들이 언제 깨어나야할 지에 대한 정보를 저장하고 있다. Wake\_t 값은 timer\_sleep 함수에서 초기화 해준다. Blocked 된 thread들은 새로 정의한 list 구조체block\_list에 저장되며, 해당 list에 저장된 thread들은 다시 wake\_thread 함수에서 ready\_list로 이동한다. 또한 tick이 가장 조금 남은 thread를 구하기 위한 get\_next\_awake 함수와 smallest\_tick 변수, update\_smallest\_tick 함수 등을 새로 정의한다.

**2) Priority Scheduling**

Thread.c 파일과 semaphore에 사용되는 waiter\_list 또한 우선순위를 적용시키기 위해서 synch.c 를 수정한다. Thread\_set\_priority(), thread\_get\_priority(), thread\_tick() 함수 등을 수정하며, aging을 구현하기 위한 thread\_aging() 함수, 우선순위를 비교하기 위한 is\_prior\_larger() 함수를 새로 정의한다.

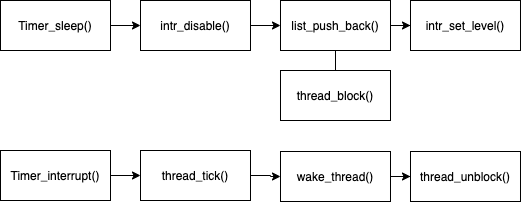
우선순위를 기준으로 정렬하여 block\_list 및 ready\_list에 저장하기 위해서 thread\_yield 함수와 thread\_unblock 함수에서 is\_prior\_larger 함수를 사용하는 list\_insert\_ordered 함수를 호출한다. 또한 Ready list에 running thread보다 높은 priority를 가진 thread가 들어올 경우 더 높은 우선순위의 thread에게 cpu를 양보하기 위해서 thread\_create 함수에서 우선순위를 비교하고, 더 높을 시에 thread\_yield 함수를 호출한다.

기존 pintos는 semaphore를 대기중인 thread들의 waiters list를 FIFO 형식으로 구현하였으므로, 우선순위를 고려하지 않고 삽입되는 순서대로 Lock을 획득하였다. 따라서 이에 대해 우선순위로 정렬하는 list\_insert\_ordered 함수와 is\_prior\_larger 함수를 사용함으로써 이를 해결한다.

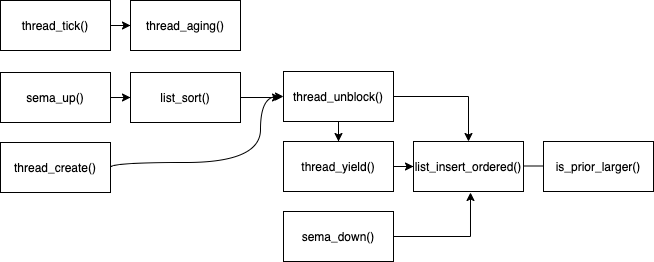
**3) Advanced Scheduler**

Mlfq 방식을 사용하기 위해서 위에서 언급한 load\_avg, nice, recent\_cpu 변수를 선언하고, 해당 값들을 계산하고 최신화할 함수를 새로 정의한다. Fixed point real arithmetic 을 구현하기 위해 사용되는 FRACT 변수 (1<<14), 실수와 정수의 덧셈을 계산하는 FAI 함수,정수에서 실수를 빼는 함수인 ISF 함수, 그리고 실수끼리의 나눗셈과 곱셈을 수행하는 FDF 함수와 FMF함수가 새로 정의되는 함수에 포함된다. Load\_avg 와 recent\_cpu 값은 각각 renew\_recent\_cpu 함수와 renew\_load\_avg함수에서 renew되며, 이 함수들은 thread\_tick 함수에서 매 초마다 호출된다. 또한 recent\_cpu 값은 매초마다 1씩 증가한다. 마지막으로 priority 값은 4초에 한번씩 renew\_prior 함수를 호출하고 해당함수에서 최신화 된다.

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

****

* + 1. **Priority scheduling**

****

* 1. **제작 내용**

텍스트이(가) 표시된 사진

자동 생성된 설명**1) Alarm Clock**

앞에서 언급했던 바와 같이 Busy waiting 방식을 고치기 위해서 우선 Timer\_sleep 함수 내에서 반복문을 고쳐주었다. 해당 함수에서 Thread 구조체에 추가되었던 wake\_t 함수를 ticks + start 값으로 초기화 해주었고, 이 후 가장 Tick이 조금 남은 thread의 시간을 저장하기 위해서 update\_smallest\_tick 함수를 호출하였다. 그리고 List\_push\_back 함수를 호출하여 block\_list 에 block 될 thread를 새로 추가해주고, Thread\_block 함수를 호출하여 해당 thread의 상태를 blocked, 즉 sleep 상태로 바꾸어 주었다. 이 일련의 과정에서 interrupt가 발생하면 안되므로, prec\_level 변수를 사용하여 Instr\_disable 함수를 호출하여 interrupt 를 disalbe 시켰고, 모든 과정이 수행되고 난 후에 intr\_set\_level 함수를 호출하텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명여 다시 interrupt가 가능하도록 돌려놓았다.

위의 두 함수중 먼저 timer\_interrupt 함수에서 thread\_tick() 함수를 먼저 호출하고 난 후에, get\_next\_awake() 함수를 호출하여 가장 작은 wake\_t 값을 불러와 현재 Tick과 비교한다. 만약에 해당 tick이 같아지는 경우 wake\_thread() 함수를 호출하여 해당 함수에서 thread를 깨워주게 된다.

Wake\_thread() 함수를 보면, block\_list에 저장되어 있는 thread들을 while문을 통해서 순회하며 깨워야할 thread를 wake\_t 값을 사용하여 찾아낸 후, 해당 thread를 block\_list에서 지운다. 그리고 thread\_unblock() 함수를 호출하여 thread의 State를 다시 Ready 상태로 만들고, thread\_unblock() 함수에서 ready list에 해당 thread를 다시 집어넣는다.

텍스트이(가) 표시된 사진

자동 생성된 설명

그리고 위 함수들은 앞서 가장 작은 Tick을 유지하고 불러오는데 사용되는 함수들이다. Update\_smallest\_tick 함수는 timer\_sleep 함수에서 호출되며, 해당 thread tick 값과 현재 Smallest\_tick 값을 비교하여 더 작은 값을 저장한다. Get\_next\_awake 함수는 그렇게 저장된 Smallest\_tick 변수값을 return 한다.

이러한 과정들을 거쳐서 thread들을 Sleep 상태로 대기시켰다가 일정 Tick이 지난 이후 다시 깨우는 수행이 가능해지고, 따라서 Cpu 효율을 전보다 더 높일 수 있었다.

**2) Priority Scheduling**

우선순위를 고려하여 ready list( 위의 구현까지는 단순 FIFO으로 작동하기 때문에, ready queue라고 할 수 있었다. ) 에서 thread를 Cpu로 Schedule 해주기 위해서 먼저 thread\_yield 함수와 thread\_unblock 함수에서 ready list에 thread들을 넣어줄 때 list\_insert\_orderd 함수를 사용하여 넣어 주었다. 이는 다음 코드에서 확인 가능하다

**텍스트이(가) 표시된 사진

자동 생성된 설명**본 과정에서 **텍스트이(가) 표시된 사진

자동 생성된 설명**Interrupt가 발생하면 안되기 때문에 마찬가지로 intr\_disable 함수와 Intr\_set\_level 함수 사이에 코드를 작성함으로써, 수행 도중에 interrupt가 발생하는 것을 방지하였다. List\_insert\_ordered 함수에서 is\_prior\_larger 함수가 argument로 사용되는 것을 확인할 수 있는데, 이는 우선순위를 기준으로 하여 ready list 안의 thread 들을 정렬하기 위함이다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**위의 코드에서 2개의 thread에 대해서 각각의 priority를 비교하여 priority값이 더 크다면 true를 return 하고, 더 작다면 false를 return 하는 것을 확인할 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명**텍스트이(가) 표시된 사진

자동 생성된 설명** 다음으로 priority를 설정하는 함수인 thread\_set\_priority 함수이다. 해당 함수는 advanced scheduler인지를 구분하는 thread\_mlfqs flag가 flase인 경우만 적용이 된다. Advanced scheduler에서는 priority가 다른 식에 의해 결정되기 때문이다. 마찬가지로 interrupt가 발생하지 않도록 설정 한 후, 현재의 thread에 new\_priority를 저장하여 그 값을 renew 해준다. 이 때 이전의 priority 값인 prec\_prior 값보다 최신화 해준 new\_priority 값이 더 작다면, 우선순위가 감소한 것이므로, thread\_yield 함수를 호출하여서 다시 scheduling 해 줌으로써 다른 thread가 Preempt될 수 있도록 하였다.

한편 왼쪽에서 볼 수 있듯이 thread\_create 함수에서 새로 생성된 thread의 우선순위가 현재 cpu를 사용하고 있는 thread 보다 더 높을 시에 Thread\_yield 함수를 호출하여 우선**텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명**순위가 더 높은 thread가 preempt 될 수 있도록 한 것을 확인할 수 있다.

synchronization 을 수행하는데 있어서 사용되는 waiter list 내에서도 우선순위에 따라서 Scheduling 될 수 있도록, sema down 함수에서 list\_insert\_ordered 함수를 is\_prior\_larger 함수를 사용하여 우선순위에 따라 해당 thread들을 저장한 것을 볼 수 있고, 이를 바탕으로 sema\_up 에서 우선순위가 가장 높은 thread를 선택할 때, list\_entry 함수를 list\_pop\_front 함수를 사용하여 가져옴으로써, 정렬된 waiter list에서 가장 앞쪽에 있는, 즉 우선순위가 가장 높은 thread를 가져오게 되는 것을 볼 수 있다. Sema\_up 은 Thread를 다시 ready 상태로 바꿔주는 것이므로, waiter list에서 해당 Thread를 삭제하고, thread\_unblock 함수를 호출하여, state를 바꿔줌과 동시에 다시 ready list 에 넣는 수행을 함을 알 수 있다. 마찬가지로 두 함수에서 interrupt가 도중에 발생하는 것을 방지하였다. 또한 여러 Thread가 공유 자원인 condition variable을 얻기 위해 기다리고 있는 상황에서, 다른 thread들은 현재 수행중인 Thread가 끝나기 전까지 기다려주고 있으므로, cond\_wait 과 cond\_signal 함수에서 list\_sort 함수와 list\_insert\_ordered 함수를 사용하여 우선순위를 기준으로 정렬해 주었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

마지막으로, aging 방법을 구현하기 위해서 thread\_tick 함수에서 thread\_prior\_aging flag에 해당될 경우 해당 thread\_aging 함수를 호출 하였고, 왼쪽에서 볼 수 있듯이 모든 ready list에 저장된 thread들을 순회하며 각 thread들의 Priority 값을 1씩 증가시켜주는 것을 확인할 수 있다.

**텍스트이(가) 표시된 사진

자동 생성된 설명3) Advanced Scheduler**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Advanced scheduler에서는 앞 서 언급했던 바와 같이 fixed point 연산을 수행해야 하므로, 해당 연산에 필요한 함수들을 선언한 것을 볼 수 있다. 그리고 load\_avg, recent\_cpu 그리고 이에 따른 priority 값을 최신화 시켜주기 위해서 renew\_load\_avg, renew\_recent\_cpu, renew\_prior 함수를 새로 정의하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명 왼쪽 코드는 thread\_tick 함수이다. 해당 함수에서는 위에서 언급한대로 aging flag에 해당하면 thread\_aging 함수를 호출시키고, thread\_mlfqs Flag에 해당되면 이에 필요한 일련의 과정을 수행하도록 구현하였다. 먼저, Recent\_cpu 값은 매 초마다 1씩 증가하는데, 이는 실수와 정수의 덧셈이므로 위에서 정의한 FAI 함수를 호출하여 계산해 주었고, 이 후 매 초마다 load\_avg와 recent\_cpu 값을 최신화 해주는 함수를 호출하고, 매 4초마다 해당 값들을 사용하여 priority 값을 최신화 시켜준다.

Recent\_cpu 값과 Load\_avg 값은 명세서에 100을 곱해준 값을 반환한다고 되어 있으므로, thread\_get\_recent\_cpu 함수와 thread\_get\_load\_avg 함수를 이를 반영하여 작성해 주었고, Thread\_get\_nice 함수도 현재 thread의 nice 값을 반환하도록 작성하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명아래는 명세서에서 주어진 식에 따라서 작성한 각 nice, load\_avg, recent\_cpu 및 priority에 대한 함수들이다.

Thread\_set\_nice 함수에서는 Nice 값을 저장하며, thread의 priority 값을 초기화 해준다. 해당 값이 범위(0~63)를 벗어나면 최소, 최대 값을 텍스트이(가) 표시된 사진

자동 생성된 설명할당해줌으로써 오류를 방지하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명위는 Recent\_cpu를 최신화 하는 함수이다.

텍스트이(가) 표시된 사진

자동 생성된 설명그리고 위는 load\_avg를 최신화 시켜주는 함수이다. Load\_avg는 앞 서 언급한 바와 같이 system wide 값이고, 이는 ready list에 저장되어 있는 thread들의 갯수, 즉 Ready state인 thread들의 갯수를 반영하기 때문임을 알 수 있다.

마지막으로 Priority를 최신화 시켜주는 함수이다. 본 함수에서는 위의 thread\_set\_nice 함수에서 우선순위 값을 초기화해준대로 모든 상태의 Thread들의 우선순위 값을 식에 따라서 바꿔주는 것을 확인 할 수 있다.

Advanced scheduling을 구현하면서 몇가지 오류가 발생했었는데, 우선 바로 위에 있는 renew\_prior를 작성할때, Idle\_thread에 대한 prority를 제외하고 나머지에 대해서 변경해 주는 실수를 범하여, All\_list의 의미와 전체 Mlfq 의 값 최신화에 대한 개념을 다시 고칠 수 있었다. 그리고, thread\_yield를 전체 흐름을 고려하지 않고 모든 곳에서 해주어서 오류가 생긴적이 있었다. 이는 전체 flow를 고려하여 다시 수정함으로써 해결하였다.

* 1. **시험 및 평가 내용**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명priority-lifo.c 코드 및 priority-lifo 테스트 결과 분석

Priority-lifo 테스트는 priority가 PRI\_Default + Thread\_cnt + 1로 정의되므로, PRI\_Default + 1인 thread부터 +16인 Thread가 Create되고, 해당 수행되는 Thread id를 출력한다. 결과로부터 Thread들이 단순 FIFO가 아닌 가장 마지막에 들어갔지만 우선순위가 가장 높은 thread부터 먼저 출력되는 것을 확인할 수 있다. 따라서 priority scheduling을 정상적으로 구현하였다고 판단할 수 있다.

* 텍스트이(가) 표시된 사진

  자동 생성된 설명make check 수행 결과를 캡처하여 첨부

본 프로젝트에서 요구하는 Test들을 전부 통과한 것을 확인하였다.