**Pintos Project 3: Threads**

담당 교수 : 박성용 교수님

조 / 조원 : 남현준

개발 기간 : 10/30 ~ 11/19

1. **개발 목표**

* 해당 프로젝트에서 구현할 내용을 간략히 서술.

현재의 pintos는 timer\_sleep()함수를 사용하며 thread\_yield()를 사용하여 thread가 RUNNING state와 READY state를 오가는 비효율적인 방법으로 alarm clock이 작동되고 있다. Project #3에서는 alarm clock의 비효율적인 작동을 timer\_interrupt()를 통해 시간을 재어 BLOCKED state로 전환된 thread를 ready queue에 다시 추가하는 효율적인 방식으로 작동될 수 있도록 alarm clock을 개선하게 될 것이다.

thread들의 우선도에 대해서도 현재의 pintos는 round-robin 방식의 scheduling을 채용하고 있고, 이러한 방식 때문에 priority가 높아 당장 처리되어야 하는 I/O request 등의 waiting time과 turnaround time이 길어지는 문제점이 있다. 이러한 문제를 해결하기 위해 priority를 기준으로 다음에 실행될 thread를 선택할 수 있는 priority scheduling을 구현하고, 동시에 이로 인해 발생할 starvation 문제를 막기 위한 aging 기법을 함께 구현하게 될 것이다.

추가적인 구현으로는 BSD scheduler를 구현하게 될 것이다. 해당 scheduler는 Multi-Level Feedback Queue (MLFQ)를 통해 구현될 것이며 nice, recent\_cpu, priority 값을 주어진 자료를 기반으로 계산할 수 있도록 할 것이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

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

Thread가 RUNNING, READY state를 오가던 기존의 방식과 달리, BLOCKED(SLEEP) state로 thread를 전환시키고 일정 시간이 지났다면 thread를 READY state로 전환하여 ready queue에 추가되는 방식으로 운영체제가 실행될 것이다.

* 1. Priority Scheduling

Thread의 우선 순위를 기준으로 높은 우선 순위의 thread가 먼저 실행되고, 낮은 우선 순위의 thread가 나중에 실행될 것이다. 하지만, 이 과정에서 starvation issue가 생길 수 있으므로 이를 보완하기 위해 aging을 통해 일정 time period동안 processor를 할당받지 못했다면 priority boost를 통해 실행될 것이다.

* 1. Advanced Scheduler (추가구현을 한 경우)

Multi-Level Feedback Queue를 통해 구현될 것이며 각 queue에는 priority가 설정되어 있고 가장 높은 priority에 있는 thread부터 선택되어 실행되는 방식이다.

* 1. **개발 내용**
* 아래 항목의 내용만 서술

1. Blocked 상태의 스레드를 어떻게 깨울 수 있는지 서술.

BLOCKED 상태의 thread를 저장할 list를 선언하여 해당 list에 BLOCKED 상태의 thread를 저장한다. 이 때, 저장되는 thread들에는 언제 unblock이 되는지 알 수 있게 후에 설명할 wakeupTime 변수를 통해 관리하게 된다. Timer\_interrupt() 함수를 통해 clock tick마다 thread가 BLOCKED 상태에 들어간지 얼마나 지났는지 확인하게 되고, unblock이 될 시간이라면 해당 thread를 unblock 시킨 후 ready list에 삽입하는 것으로 thread를 깨우게 된다.

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

현재의 Ready list에 running thread보다 높은 priority의 thread가 들어오게 될 경우, 실행 중인 thread를 ready list에 넣고, ready list에서 가장 priority가 높은 thread가 실행되게 된다.

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

BSD scheduler에서는 -20 ~ 20의 범위를 가지는 nice 변수가 존재한다. Nice는 0일 때 아무런 영향을 주지 않고, 양수의 값을 가진다면 priority를 낮춰 다른 thread가 processor를 사용할 수 있도록 한다. Nice는 thread가 생성될 때 0의 값으로 초기화가 진행되며 thread가 생성되는 경우가 아니라면 parent thread에서 nice 값을 상속받아 사용하게 된다.

Pintos에서는 floating-point arithmetic을 지원하지 않고 fixed-point format을 사용한다. 따라서 priority, recent\_cpu, load\_avg 등을 계산하게 될 때 자료에서 주어진 식을 그대로 사용하지 못하는 경우가 생길 수 있다. 이를 위해, 특정 계산에서는 fixed-point 형식으로 바꾼 다음 해당 식에 맞춰 계산이 진행될 수 있도록 하였다.

BSD scheduler에서 priority의 경우 PRI\_MAX – (recent\_cpu / 4) – (nice \* 2)의 식으로 계산이 진행된다.

매 4 tick마다 priority의 재계산이 이루어지며 해당 식에서 recent\_cpu는 최근에 thread가 사용한 CPU time을 예측한 값이고 nice는 thread가 가지고 있는 nice 값을 의미한다.

Recent\_cpu는 다음과 같은 방식으로 계산된다. (2 \* load\_avg) / (2 \* load\_avg + 1 ) \* *recent\_cpu* + nice. 여기서 load\_avg는 READY 상태에 있는 thread의 평군 개수를 의미한다.

Load\_avg는 다음과 같은 식으로 계산된다. (59/60) \* *load\_avg* + (1/60) \* *ready\_threads.* Ready\_threads의 경우 idle thread를 제외한 READY나 RUNNING state인 thread의 개수를 의미한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* II. A. 개발 범위를 포함하여 구현 내용에 대한 일정 작성

2023.10.30 ~ 2023.11.03 : Alarm clock 구현

2023.11.03 ~ 2023.11.08 : Priority scheduling 및 aging 구현

2023.11.08 ~ 2023.11.19 : BSD scheduler 구현

2023.11.15 ~ 2023.11.19 : 보고서 작성

* 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
* Alarm clock

Blocked 상태의 thread들을 관리하기 위한 list를 timer.c 파일에 struct list sleepList를 통해 선언했다. 해당 list를 timer가 초기화될 때 같이 초기화되도록 설정했다.

Thread.h 파일에서 alarm clock에서 thread가 언제 unblock될지 알 수 있도록 wakeupTime 변수를 선언하고, thread.c 파일에서 thread의 초기화를 진행 시 wakeupTime 변수를 0으로 설정하도록 한다.

Timer.c 파일의 timer\_interrupt() 함수와 timer\_sleep() 함수를 수정한다. Timer가 설정한 tick이 될 때까지 thread\_yield()를 하던 기존의 방식에서 wakeupTime을 시작시간 + ticks로 설정한 후, 앞서 선언한 sleepList에 현재 thread의 정보를 push한다. 그 후, thread\_block()을 통해 blocked 상태로 전환시킨다.

Thread\_interrupt() 함수는 tick이 증가될 때마다 thread\_wake\_up() 함수를 통해 unblock을 진행할 thread가 있는지 없는지 체크한 후, unblock을 해야하는 경우 해당 thread를 unblock한후 ready\_list에 넣어주게 된다.

* Priority scheduling

Priority를 기준으로 thread를 실행하게 된다. 새로 생성된 thread의 priority가 현재 실행 중인 thread의 priority보다 높다면 실행 중인 thread를 yield한 후, priority가 높은 thread가 우선적으로 실행될 수 있도록 코드의 구조를 바꿔주었다.

Thread\_yield() 함수가 호출될 때, ready\_list에 list\_push\_back을 통해 단순히 push를 하던 기존의 방식과 달리, priority scheduling을 구현하기 위해서는 ready\_list에 thread가 priority의 순위대로 push 될 필요성이 있다. 따라서, thread.c의 thread\_yield() 함수에서 ready\_list에 priority 순으로 정렬되면서 push될 수 있도록 함수를 수정했다.

Thread\_set\_priority()와 thread\_get\_priority() 함수를 완성시켰다. 기본 구조만 정의되어 있는 소스코드에서 실제로 작동될 수 있고 코드를 추가했다. Thread\_set\_priority()에서는 thread의 priority를 parameter로 주어지는 new\_priority로 설정하게 된다. 이 때, 새롭게 설정된 new\_priority의 값이 기존의 priority 값보다 작은 경우, 해당 thread의 priority는 더 이상 ready\_list에 있는 thread들 중 크지 않을 수 있기 때문에 thread\_yield()를 통해 ready\_list에서 priority가 가장 높은 thread가 선택되어 실행될 수 있도록 한다. Thread\_get\_priority()의 경우 현재 실행되고 있는 thread의 priority를 반환할 수 있도록 thread\_current()->priority를 return하도록 했다.

Aging이 실행될 수 있도록 자료에 나와있는 thread\_prior\_aging 변수를 필요한 곳에다 자료를 보며 추가를 진행했다. 이렇게 추가한 thread\_prior\_aging을 통해 -aging이 명령어에 들어오는 경우 priority, load\_avg, recent\_cpu의 재계산이 이루어질 수 있도록 timer\_interrupt() 함수에 해당 로직을 추가했다.

Priority\_sema의 구현을 위해서는 semaphore로 block된 thread가 sema\_up을 통해 다시 실행을 진행하게 될 때, priority의 우선 순위대로 실행되지 않을 수 있다는 점을 해결해야 한다. 해당 부분의 구현을 위해, synch.c의 sema\_up 함수에서 sema->waiters list를 순회하며 list의 가장 앞에 있는 wait 상태의 thread보다 큰 priority를 가진 thread가 있는지 확인한다. 있을 경우, sema->waiter는 priority의 우선순위대로 실행되지 않는다는 것을 의미한다. 따라서, 해당 경우에는 list\_sort()를 통해 priority 순으로 정렬될 수 있도록 한다. 이후, thread\_unblock()을 통해 sema->waiters에 저장된 첫 번째 thread를 unblock 시키고 list에서 해당 정보를 삭제한다.

* BSD scheduler

BSD scheduler를 사용하기 위해 load\_avg 전역변수를 선언하고 thread.h에서 thread의 정보에 nice와 recentCPU를 추가했다. 이후, thread가 생성될 때 nice와 recentCPU의 값을 0으로 초기화를 진행한다.

자료에서 주어진대로 BSD scheduler의 구현을 위해 thread\_get\_nice, thread\_set\_nice, thread\_get\_recent\_cpu, thread\_get\_load\_avg 등의 함수와 timer의 tick 진행될 때마다 priority, load\_avg, recent\_cpu의 값을 재계산하는 로직을 추가한다. 재계산 부분은 timer\_interrupt() 함수에서 -mlfqs 옵션으로 실행될 때에만 실행될 수 있도록한다.

Load\_avg, priority, recent\_cpu의 경우 pintos manual에 따라 지원하지 않는 floating point 연산 대신, fixed point 연산을 통해 값을 구할 수 있도록 한다. 이를 위해, 1 << 14를 매크로를 통해 TO\_FIXED\_INT로 선언하였고, 이를 활용하여 계산을 진행한다.

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

스크린샷, 텍스트, 도표, 디자인이(가) 표시된 사진

자동 생성된 설명

* Priority scheduling

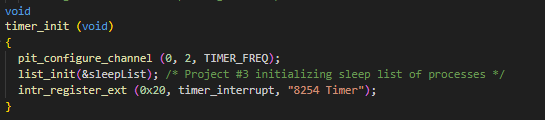
텍스트, 스크린샷, 도표, 평면도이(가) 표시된 사진

자동 생성된 설명

* 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
* Alarm clock

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명



Src/devices/timer.c 파일에서 blocked 상태의 thread들을 관리하기 위해 sleepList를 선언하고 timer의 initialization을 진행 시, sleepList또한 같이 초기화될 수 있도록 코드를 구현했다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

Project 3에서, block된 thread를 unblock하여 ready\_list에 넣어야 할 시간인지 판단할 수 있게 wakeupTime을 thread.h에 선언했다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

그 후, thread가 생성되어 초기화가 진행될 때 호출하는 init\_thread() 함수에서 thread의 wakeupTime을 0으로 초기화하는 작업을 진행했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Unblock을 할 시간이 될 때까지 thread\_yield()를 통해 yield를 시켰던 기존의 방법에서, BLOCKED 상태로 전환시킬 현재의 thread의 wakeupTime을 현재의 tick + blocked 상태를 유지 시킬 tick의 값으로 설정한 후, sleepList에 thread의 정보를 push한다. 이후, 현재 thread\_block() 함수를 통해 thread\_current()를 BLOCKED 상태로 바꿔준다.

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명

Timer\_interrupt() 함수에서 ticks가 증가할 때마다 thread\_wake\_up() 함수를 통해 sleepList에 unblock을 해야 할 thread가 있는지 확인한다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

sleepList를 순회하며 wakeupTime이 된 thread가 있다면 해당 thread의 정보를 sleepList에서 삭제로 thread\_unblock()을 통해 unblock을 진행한 후 ready\_list에 해당 thread의 정보를 insert한다.

* Priority scheduling

Thread의 생성 시 priority를 할당하여 ready\_list에 insert하게된다. 이 때, 새로 생성되는 thread의 priority가 현재 실행 중인 thread의 priority보다 높다면 현재 실행 중인 thread를 yield한 후 ready\_list에서 priority가 가장 높은 thread가 우선적으로 실행될 수 있도록 코드의 구조를 바꿔야 할 필요가 있다.

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명

Thread\_create() 함수에서 priority가 현재 실행 중인 thread의 priority보다 높을 경우, 현재의 thread를 yield하도록 코드를 추가했다. Yield 부분에서도 ready\_list에 thread들이 priority 순으로 정렬되어 insert될 수 있도록 코드를 추가했다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Ready\_list를 탐색하며 priority를 비교하게 되는데 ready\_list에 있는 thread의 priority보다 클 때까지 순회를 진행하며 priority가 큰 지점을 찾았을 경우, 해당 위치에 list\_insert()를 통해 thread의 정보를 insert할 수 있도록 하였다.

텍스트, 스크린샷, 소프트웨어, 디스플레이이(가) 표시된 사진

자동 생성된 설명

Thread\_unblock() 함수의 경우에도 thread가 ready\_list에 priority 순으로 정렬될 수 있도록 priority의 비교를 진행해서 적절한 위치를 찾은 다음 list\_insert()를 통해 thread의 정보를 list에 insert할 수 있도록 했다.

기존에 뼈대만 존재하던 thread\_set\_priority()와 thread\_get\_priority()를 다음과 같이 수정했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Set\_priority에서는 thread의 priority를 새로운 priority 값으로 설정해주게 된다. 이 때, 새로 설정한 priority의 값이 기존 priority 값보다 작다면 thread는 더 이상 가장 큰 priority를 가진게 아닐 수도 있으므로 yield를 시킨 후 ready\_list에서 priority가 가장 높은 thread를 선택하여 실행하게 된다.

Thread\_get\_priority에서는 현재 thread의 priority 값을 return하도록 했다.

Block와 unblock 외에 thread는 semaphore를 통해서도 실행되는 것이 유예될 수 있다. 하지만 이 경우, thread\_sleep()과 thread\_unblock()과는 달리 sema\_up과 sema\_down을 통해서 중단되었던 thread를 다시 실행하게 되고 이러한 경우에는 priority의 우선순위가 높은 순서가 아닌데도 실행이 재개되는 불상사가 발생하게 될 수도 있다.

이러한 문제를 해결하기 위해 synch.c 파일의 sema\_up() 함수를 다음과 같이 수정했다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

Sema->waiters, 즉 semaphore의 작동에 영향을 받는 thread 들을 저장하고 있는 sema->waiters 리스트가 empty가 아닐 경우, 해당 list가 정렬되어 있는 상태인지 is\_sorted() 함수를 통해 검사한다. Is\_sorted() 함수에서는 sema->waiters를 순회하며 priority의 우선순위대로 list가 정렬되었는지 확인하게 되는데 정렬된 상태라면 true, 아니라면 false를 반환하게 된다.

Is\_sorted() 함수의 결과를 토대로 list\_sort() 함수를 통해 sema->waiters의 정렬을 진행한다. cmpPriority의 경우 list를 순회하며 두 thread의 priority 비교를 하게 되는데 왼쪽 thread의 priority가 오른쪽 thread의 priority보다 큰 경우 true를 반환하며 아닐 경우 false를 반환한다. List\_sort() 함수에서 cmpPriority의 값이 false가 나오는 경우, 두 thread는 priority순으로 정렬된 상태가 아님을 의미하기 때문에 두 thread의 위치를 바꿔주는 방식으로 정렬을 진행하게 된다.

Priority scheduling의 경우 starvation issue가 발생할 수 있고 이를 보완하기 위해 aging을 통한 priority boost가 있어야 할 필요가 있다. Aging을 위해서는 -aging option이 명령어로 주어질 경우 thread\_prior\_aging 변수를 true로 set할 필요가 있다. 하지만, 현재의 pintos에는 이러한 변수의 선언이 이뤄지지 않았으므로 해당 변수의 추가를 진행했다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

Thread.h와 thread.c 파일에서 thread\_prior\_aging 변수의 선언 및 정의를 진행한다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

Src/threads의 init.c 파일에서 -aging에 명령어로 들어오게 될 경우, thread\_prior\_aging을 true로 설정하여 aging 관련 작동이 진행될 수 있도록 한다.

Thread\_interrupt() 함수에서 일정 시간이 지날 때마다 recalculatePriority()를 통해 priority의 재계산이 진행되어 priority boost를 통해 priority가 낮아 실행되지 못하고 있던 thread들이 실행될 수 있도록 코드를 수정했다.

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

recalculatePriority() 함수의 구현은 다음과 같다.

* BSD scheduler

-mlfqs가 명령어로 들어오게 될 때, BSD scheduler를 사용하여 priority, load\_avg, nice의 재계산이 이루어져야 한다.

텍스트, 스크린샷, 소프트웨어, 운영 체제이(가) 표시된 사진

자동 생성된 설명

Thread\_mlfqs가 true일 때, recent\_cpu, load\_avg, priority의 재계산이 이루어지도록 코드를 수정했다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

Thread.h 파일에 BSD scheduler를 사용할 때 쓸 변수 recentCPU와 nice를 선언했다.

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

Thread.c 파일에서 load\_avg의 선언과 초기화를 진행한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

이후, thread의 초기화를 진행하며 recentCPU, nice의 값을 초기화하게 된다. 처음 생성되는 경우에는 nice와 recentCPU의 값을 0으로 초기화하고, 처음 생성된 thread가 아니라면 부모의 recentCPU와 nice 값을 계승하여 초기화를 진행한다.



텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Thread\_set\_nice() 함수에서는 thread\_current의 nice 값을 설정하게 된다. Nice의 값을 설정하며 priority의 재계산이 진행된다. 해당 식은 PRI\_MAX – (*recent\_cpu* / 4) – (nice \* 2)로 진행된다. 하지만 pintos에서는 floating point 계산이 지원되지 않으므로 이를 계산하기 위해 TO\_FIXED\_INT를 선언하여 실수 연산이 가능해지도록 했다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Thread\_interrupt()에서 thread의 recent\_cpu는 idle thread가 아닌 경우 1이 증가될 수 있도록 해야한다.

recalculatePriority() 함수에서는 모든 thread에 대해 priority의 재계산이 진행된다. Priority를 주어진 식으로 fixed point 연산을 적절히 사용하여 계산하게 되는데 이 때, 재계산을 통해 구한 priority의 값이 PRI\_MAX보다 크거나 PRI\_MIN보다 작은 경우에는 각각 PRI\_MAX, PRI\_MIN의 값으로 설정해준 후 다음 thread의 priority의 재계산한다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

recalculateRecentCPU() 함수에서는 idle state가 아닌 모든 thread에 대해 recent\_cpu 값의 재계산을 진행한다. 이 때, recent\_cpu의 값을 계산하면서 overflow issue가 생길 수 있기 때문에 계산을 진행할 때는 64bit로 진행한 다음 thread의 recentCPU 값을 갱신할 수 있도록 진행하였다.

Load\_avg의 경우 idle state가 아닌 ready or running 상태의 thread의 개수를 기준으로 재계산이 이루어지게 된다. 현재 실행 중인 thread가 idle thread가 아니라면 ready\_list의 크기 + 1의 값으로 재계산을 진행하고, idle thread라면 ready\_list의 크기로 재계산을 진행할 수 있도록 했다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명

Get\_load\_avg()와 get\_recent\_cpu() 함수에서는 load\_avg \* 100의 값과 thread\_current()->recentCPU \* 100의 값을 return할 수 있도록 했다.

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

Priority-lifo 에서는 thread\_cnt개의 thread가 생성되고 각 thread는 PRI\_DEFAULT + THREAD\_CNT + 1의 priority를 가지게 된다. 이후, ITER\_CNT 횟수만큼 lock과 release를 반복하며 yield하는 식으로 thread가 작동하게 된다. 이러한 과정은 round-robin에서의 작동방식과 유사하고 결과적으로 priority가 높은 순서대로 pintos의 출력으로 찍히게 될 것이다.

텍스트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

실제로도 해당 결과를 출력하는 것을 확인할 수 있다.

* make check 수행 결과를 캡처하여 첨부

텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

Alarm-zero – 출력이 밀림으로 인해 fail이 뜸

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Priority-change – 출력이 밀림으로 인해 fail이 뜸