System Programming Project 5

담당 교수 : 김영재

이름 : 남주형

학번 : 20171630

1. **개발 목표**

해당 프로젝트는 여러 client들의 동시 접속 및 서비를 위한 concurrent stock server를 event-driven방식과 Thread-based방식으로 각각 구축하는게 최종 목표이다.

각각의 방식을 구현하기 이전에 자료구조를 작성한다. 자료구조는 이진트리로 작성한다. 주식 관리는 stock.txt 파일로 관리한다. 그리고 show, buy, sell에 대한 기능을 singleclient에서 작동하도록 함수로 작성한다.

그리고 event-driven 방식과 thread-based방식으로 concurrent하게 구현한다.

2. **개발 범위 및 내용**

A. 개발 범위

1. select

   overhead가 발생하지 않기 때문에 client의 개수에 따른 실행시간이 많이 차이나지 않는다. client 여러 개를 동시에 실행하여도 concurrency하게 실행이 된다. 하지만 사실은 I/O Multiplexed Event Processing을 하고 있으므로 실제로 concurrency하게 실행되는 것은 아니다.

2. pthread

   처음 worker threads의 개수에 따라 실행시간이 달라진다. 예를 들면 thread를 8개만 사용하면 client의 개수가 8개 이하에선 모두 비슷한 실행시간이 측정되지만 client의 개수가 16개라면 약 2배의 시간이 32개라면 4배의 시간이 걸린다.

B. 개발 내용

- **select**

  ✓ select 함수로 구현한 부분에 대해서 간략히 설명

먼저 pool에 대한 struct를 정의해주고 init_pool, add_client, check_clients 함수를 정의해준다. 그리고 main에서 먼저 init_pool함수를 통해 pool을 초기화해주고 while문 안에서 pool.ready_set을 pool.read_set으로 세팅해주고 select 함수를 이용해 pool.nready를 세팅해준다. 그리고 FD_ISSET이 set됐다면 client와 server를 연결하고 add_client를 통해 client를 추가해준다. 그리고 check_clients함수를 통해 여러가지 명령어를 주고받으며 기능을 실행한다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

check_clients함수에서 EOF detected가 되면 해당 client와 연결을 종료하는데 이때 연결을 종료한 후 stock.txt 파일에 자료구조에 저장된 정보를 업데이트 해준다. 이때 각 client가 종료될때마다 최신화를 시켜준다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

먼저 sbuf_t에 대한 struct를 정의해준다. 그리고 thread, sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove 함수를 작성한다. main에서 while문에 들어가기전 sem_init으로 mutex를 초기화해주고 sbuf_init을 통해 sbuf를 초기화해준다. 그리고 지정한 thread의 개수만큼 Pthread_create를 통해 thread를 생성해준다. 이제 while문 안에서 client와 연결이 되면 sbuf_insert를 통해 client의 connfd를 sbuf에 넣어준다.

thread함수 안에서 echo함수를 불러주어서 주식 프로그램에 필요한 기능들을 실행시켜준다. 그리고 client의 연결이 종료되면 stock.txt에 자료구조의 정보를 업데이트해준다.

C. 개발 방법

※해당 프로그램은 교재코드를 baseline코드로 사용하여 작성하였다.

- **select**

먼저 자료구조에 대한 내용을 추가해주어야한다. 주식 정보를 저장하는 struct인

ID, left_stock, price의 정보를 담고있는 item을 정의하고 binary tree를 위한 node struct를 추가해준다. 여기에는 item data와 leftChild, rigthChild가 들어있다.

binart tree 자료구조의 기본적인 setItem, setNode, addNode, searchTree, showTree, deleteTree함수를 정의해준다. 그리고 정의한 함수들로 show, buy, sell 기능에 해당하는 각각의 함수를 구현해준다.

먼저 show함수는 재귀함수로 구성되어있다. 흔하게사용하는 show함수의 print대신 Rio_writen을 사용해 client에게 정보를 넘겨준다. 여기서 재귀함수이므로 모든 정보를 매번 넘겨주는데 원래 client에서는 한번만 읽기 때문에 처음 정보만 읽을 수 있었는데 show에 한해서 client에서 'end₩n'신호를 넘겨 받기 전까지 while문을 돌며 Rio_readlineb를 통해 server에서 보내오는 정보들을 받아준다.

buy함수는 위에서 정의한 searchTree함수를 통해 먼저 정보를 읽어오고 그 정보에 따라 조건에 부합할때와 부합하지 않을때를 나누어 Rio_writen으로 client에 정보를 넘겨준다.

sell함수역시 buy함수와 유사하게 구현해주었다.

다음으로 pool 구조체를 정의해주었고 init_pool, add_client, check_clients함수를 추가해주었다. 여기서 check_client함수를 수정해주었다.

Rio_readlineb를 하는 if문안의 내용을 수정해주었는데 입력받은 buf의 내용을 cmd, id, num으로 파싱해준다. 그리고 cmd가 show, buy, sell 중 해당하는 기능이 있다면 그 기능에 해당하는 함수를 호출해준다. 그리고 client와 연결이 종료되었다면storeTxt를 통해 txt에 자료구조를 저장해준다.

- **pthread**

먼저 자료구조와 같은 기본적인 정보는 위의 방식과 유사하다 하지만 순서를 조정하는 방식인 event방식과는 다르게 thread방식에서는 정말 concurrent하게 자료구조에 접근하므로 semaphore를 잘 활용해 주어야한다. 그러기 위해서 정의해둔 자료구조에 sem_t mutex,w를 추가해준다. 그리고 전체적인 자료구조는 같지만 show, buy, sell, storeTxt를 할 때 readers-writers problem이 발생할 수 있기 때문에 first readers-writers problem의 해결방법을 적용해준다. 여기서 show와 stroeTxt는 reader에 해당하고 buy와 sell은 writer에 해당한다.

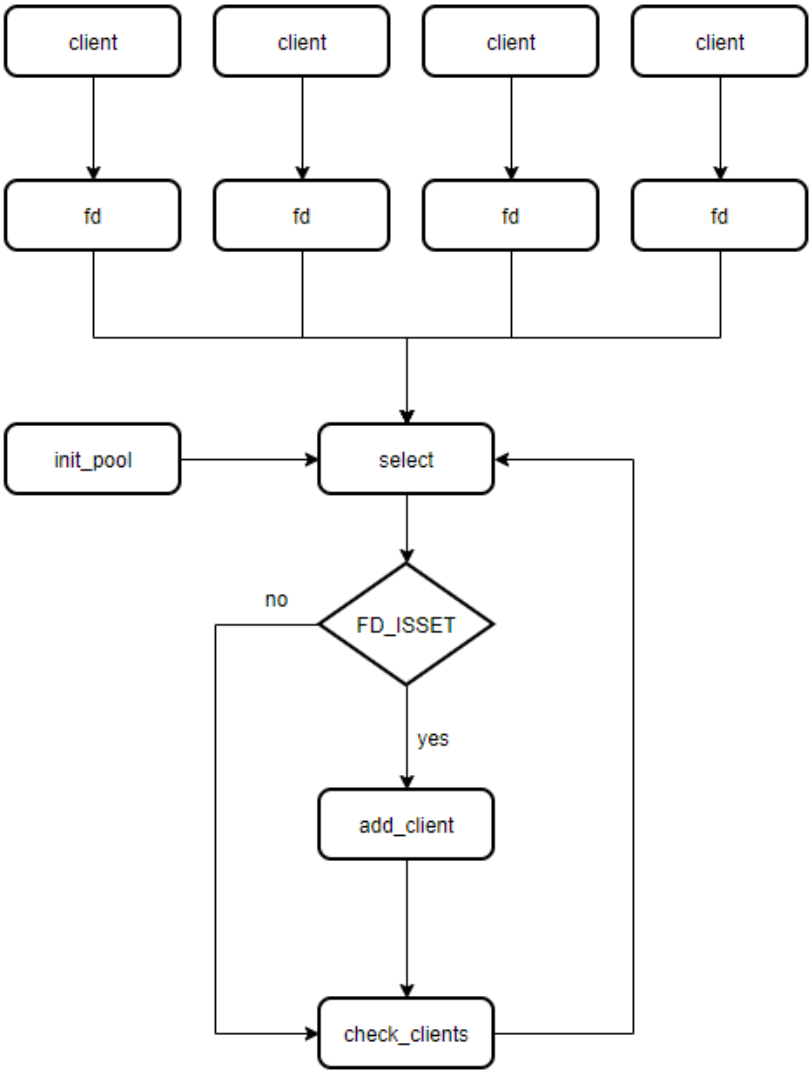
다음으로 sbuf_t에대한 구조체를 정의해주고 thread, sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove함수를 추가해주었다. 여기서 thread함수를 수정해주었다. thread함수에 while문 안에 echo함수를 추가해주었고 client가 종료될 때 자료구조의 정보를 txt파일에 업데이트 해준다.

여기서 echo함수는 sever와 client가 정보를 주고 받을 수 있는 함수이다. while문을 돌면서 show, buy, sell의 기능을 실행한다.

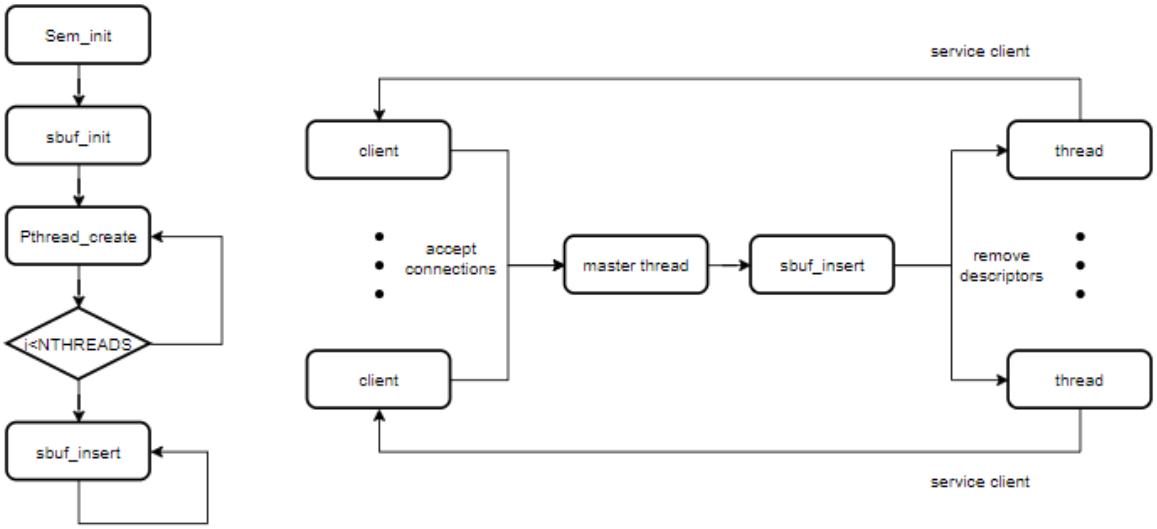
3. 구현 결과

A. Flow Chart

1. select



2. pthread



B. 제작 내용

먼저 아래 방식들을 구현하기 전에 show, buy ,sell에 대한 구현을 해주었는데 여기서 Rio_readlineb를 통해 읽어오기 때문에 '₩n'을 기준으로 읽어온다. 따라서 show를 할때는 예외를 두어 client의 while문안에서 Rio_readlineb를 반복시켜 sever의 Rio_writen으로 보내주는 정보를 계속해서 읽을 수있도록 해주었다. 그리고 정보를 다 보냈다면 sever에서 'end₩n'를 보내주어 show가 끝났음을 알려준다. 그러면 client는 while문을 빠져나온다.

여기에 더해 exit을 입력받았다면 client는 while문을 빠져나와 종료해준다.

1. select

이 방식을 구현하는데 있어 위에서 설명했듯이 check_clients함수를 수정해주었다. 이 방식은 데이터에 동시에 접근하지 않으므로 semaphore를 이용할 필요가 없다. 따라서 check_clients안에서 Rio_readlineb로 받아온 buf를 파싱하여 cmd를 구분해준다. 구분된 cmd에 따라 앞에서 구현해 놓은 show, buy, sell함수를 호출해주어 해당 기능을 실행시켜준다.

만약 EOF detected가 되면 client와의 연결을 종료해주고 stock.txt파일을 쓰기 전용으로 열어 현재 이진트리에 저장된 정보를 저장해준다.

2. **pthread**

이 방식은 위에서 말했듯이 thread들이 concurrent하게 주식 데이터에 접근하므로 read-write 문제가 발생한다. 이를 해결하기 위해 semaphore를 사용해주었다. show, storeTxt는 데이터를 읽어오므로 reader로 buy, sell은 데이터에 새로운 내용을 쓰므로 writer로 설정해주었다. 먼저 reader는 데이터를 직접읽는 부분 즉, critical section 전 후로

```
P(&node->data.mutex);
node->data.readcnt++;
if (node->data.readcnt == 1)
    P(&node->data.w);
V(&node->data.mutex);
```

(전)

```
P(&node->data.mutex);
node->data.readcnt--;
if (node->data.readcnt == 0)
    V(&node->data.w);
V(&node->data.mutex);
```

(후)

위와 같이 처리해주어 데이터를 읽는다. 그리고 writer는 reader보다 우선순위가 높다고 설정해주었으므로 따로 reader를 신경쓰지 않고 다른 writer만 생각해주면서 데이터를 쓴다. critical section전후로 P(&w), V(&w)를 사용한다. 그리고 여기서 fine-grained locking을 위해 자료구조 전체가 아니라 node하나하나에 semaphore방식을 적용했다.

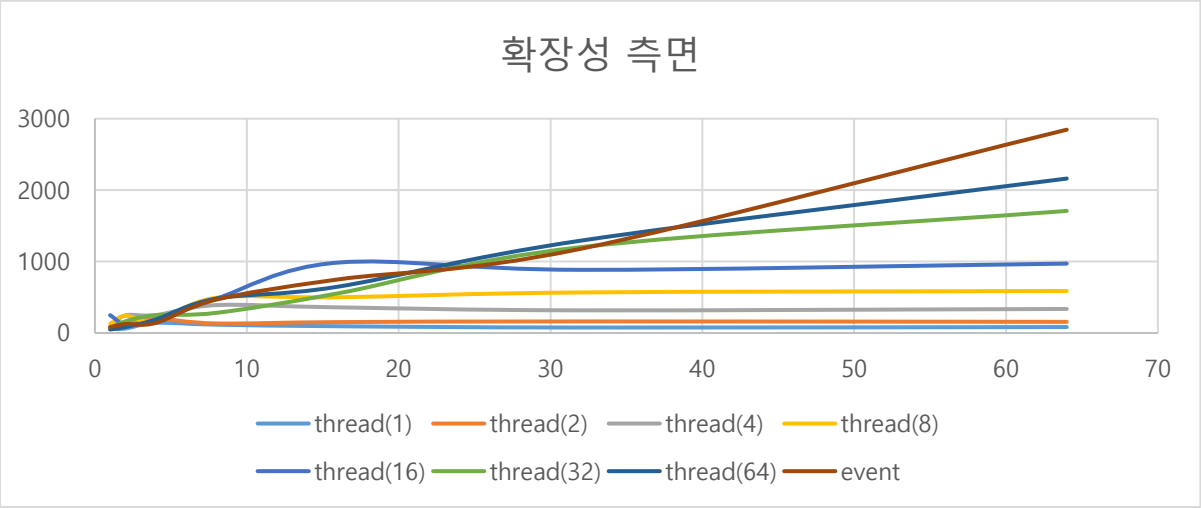
다음으로 thread함수안에 show,buy,sell등의 기능을 수행하는 echo함수를 넣어주고 connfd를 Close하면 즉, client와 연결을 끊으면 데이터를 txt파일에 저장시켜준다.

C. 시험 및 평가 내용

select는 overhead가 매우 적다 그리고 pthread는 medium overhead를 가지고 resource들을 쉽게 공유할 수 있으므로 semaphore방식을 사용하여 서로간의 데이터 침범을 막아줘야하는데 이때 사용하는 semaphore방식의 lock unlock하는데 비용이 많이든다. 따라서 select가 overhead도 적고 semaphore방식을 사용하지 않으므로 pthread보다 좋을 것이다.

먼저 show, buy, sell명령어를 random하게 실행시켜 확장성 측면에서의 분석을 해보았다. x축이 client process의 개수이고 y 축이 동시처리율이다. 동시처리율은 1

초당 client의 요청의 개수로 하였다. 즉, 1초동안 얼마나 많은 요청을 처리할 수 있는지를 측정하였다.



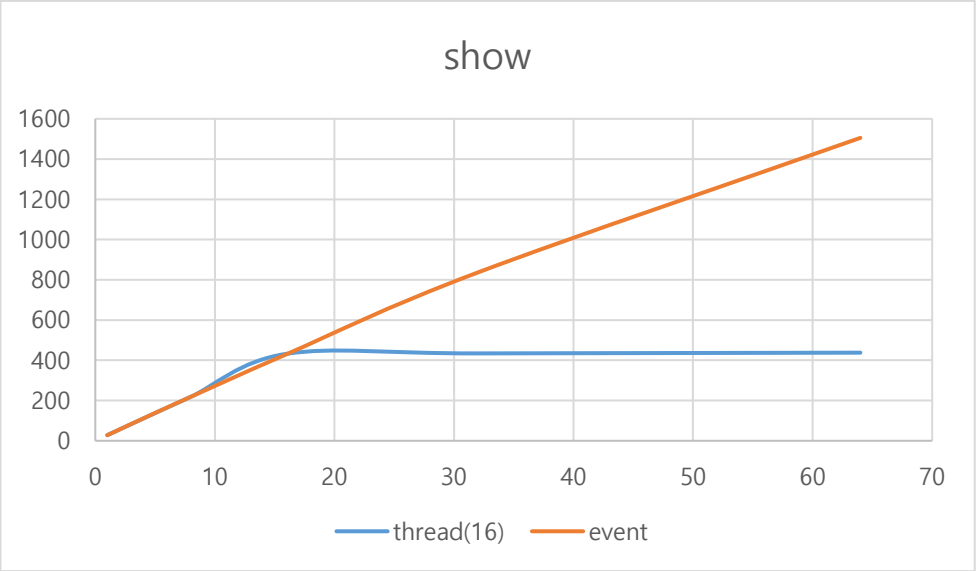
pthread를 처음에 먼저 thread pool을 정해주고 실행시키는 방법으로 구현하였기에 thread 개수별 성능도 같이 추가해주었다.

위 그래프를 보면 thread의 개수가 점점 많아 질수록 client의 개수가 증가함에따라 동시 처리율 또한 높아지는 것을 확인할 수 있다.

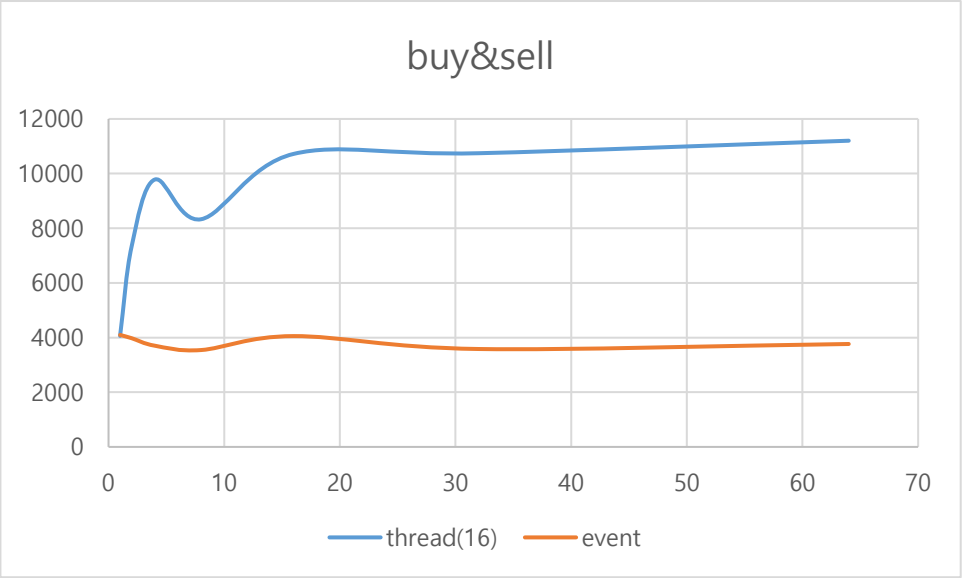
그리고 event방식과 thread방식을 비교해봤을 때 예상했던것과 같이 event방식이 client의 개수가 많아짐에 따라 더 높은 동시처리율을 보여주는 것을 확인 할 수 있다.

다음은 워크로드 측면에서 분석을 해보았다.

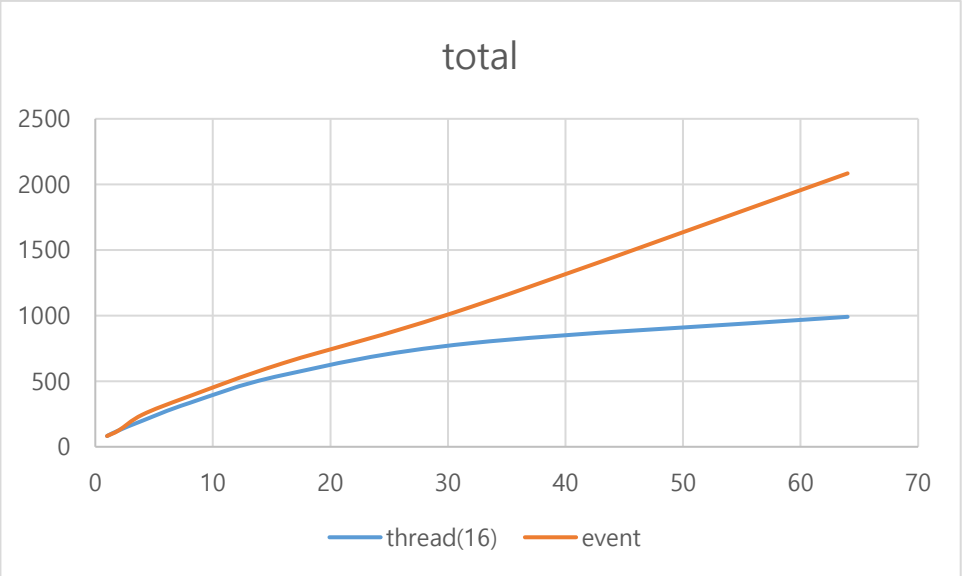
먼저 thread를 16개로 설정하고 show만 실행했을 때, buy&sell만 실행했을 때, 전체를 random하게 실행했을 때로 구분하여 측정해보았다.



위 그래프는 show만 했을 때 16thread와 event를 비교한 모습이다. 16thread라서 client가 16개 이하일때는 event방식과 별 다를 것 없는 성능차이를 보여주었다 하지만 client가 16개 이상일땐 client가 많아짐에따라 event의 동시처리율이 thread방식보다 더 좋게 나오는 것을 알수있다.



위 그래프는 buy&sell만 했을 때 16 thread와 event를 비교한 모습이다. buy와 sell은 특정한 data한 개를 write하는 것이기 때문에 show에 비해 훨씬 가벼운 것을 볼 수있다. 이 그래프에선 오히려 thread의 동시처리율이 더 좋게 나왔다. 하지만 전체적으로 봤을땐 buy와 sell의 비용이 크지 않아서 대세에 영향을 줄 정정도는 아닌 것 같다.



위 그래프는 모든 명령어를 random하게 실행했을때의 결과이다. 이 그래프에서 event방식이 thread방식보다 전체적으로 동시처리율이 좋은것으로 미루어보아 위에서 예상한것처럼 buy와 sell이 전체적인 프로그램에 큰 영향을 주지 못한 것으로 보인다.

다음은 thread의 개수에 따른 performance를 분석해보았다. core의 개수를 조절할 수 없어 다음과 같이 임의로 설정하였다. 또한 client의 개수는 8개로 통일 하였고 명령어는 일관성을 위해 show로 통일하였다.

threads(t)	1(1)	2(2)	4(4)	8(8)	16(8)	64(8)
Running time	2.801702	1.404563	0.723031	0.366607	0.368028	0.368613
Speedup	1	1.99471437	3.874940355	7.642249057	7.612741422	7.600659771
Efficiency	1	99.7%	96.8%	95.5%	95.1%	95.0%

위 표에서 알수 있듯이 speedup이 8까지 증가하다가 16부터는 주춤하는 모습을 확인할 수 있다. 또한 efficiency역시 점점 작아지는 것을 볼수있다. 이론과 잘 부합하는 결과이다.

이러한 분석들을 보았을 때 이론과 실제 구현한 코드가 대부분의 영역에서 일치하는 것을 확인할 수 있다.