Multicore Programming Project 2

담당 교수 : 박성용

이름 : 이용욱

학번 : 20191626

1. 개발 목표

이번 과제는 concurrency가 가능한 주식 서버를 구축하는 것이 목표이다.

Concurrent Server의 구현 방식 중 Event-based 방식, Thread-based 방식을 선택해서 각 방식의 실행에서 다수의 클라이언트 들로부터 서버에의 명령을 입력받아 이를 토대로 서버의 주식 리스트 관리를 구현한다. 그리고 최종적으로 두 방식의 성능을 비교하는 것이 이번 프로젝트의 내용이다. 이 때 프로젝트에서 Client는 주식의 잔여 수량을 보는 show, 주식을 사고파는 buy, sell, 그리고 서버와 연결을 종료하는 exit의 4가지 기능을 수행할 수 있다. 주식서버는 무한 루프에서 새로운 client와의 연결을 대기하고, 여러 명의 client들이 네트워크 상에서 전송한 command를 concurrent하게 수행한다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   Event-based approach를 통해서 concurrent 주식 서버를 구현한다. 이 때 서버는 하나의 logical-flow를 가진다. 무한 loop에서 연결을 시도하는 client들을 연결하여 file-descriptor를 통해 client와 연결시켜준다. 이 후 연결된 client에서 명령어가 입력되면 한 줄씩 명령어를 처리한다. 사실 하나의 logical flow를 가지고, 여러 client들의 명령을 하나씩 순차적으로 처리하기 때문에 concurrent하다고 말할 수는 없지만 , 그래도 이 방식을 통해서도 multi-client들의 명령을 처리할 수 있다. 그리고 하나의 전역 변수에 대해 one at once로 한 번에 한 명만 접근 가능하므로 data corruption을 방지한다.

2. Task 2: Thread-based Approach

   Thread-based approach에서는 master thread가 미리 정의한 개수만큼 worker thread를 구축해서 작동한다. 루프를 돌면서 master thread가 연결을 시도하는 client들을 입력받으면 이들에게 연결한 file descriptor들을 버퍼에 입력한다. 각 Worker thread 들은 각자 버퍼에서 descriptor를 가져와서 해당 client가 보낸 명령어들을 처리한다. 이 과정에서 client의 수를 세는 변수가

오염되는 것을 방지하려면 semaphore의 개념을 활용한다. 또한 각각의 주식 item 들에서 readers-writers 문제와 수량의 update 문제를 해결하기 위해 item 구조체에 readcnt와 mutex 를 추가한다. 이를 통해서 하나의 프로세스 내부에서 여러 개의 thread를 통해 multi-control flow를 구현할 수 있고, concurrent server를 구현한다.

3. Task 3: Performance Evaluation

위의 두 과정에서 구현한 concurrent server의 성능을 비교하고 분석한다. 성능을 비교할 때는 동시 처리율의 개념을 이용한다. 이 값은 단위 시간 당 서버가 처리하는 명령어의 개수를 나타낸다. 그래서 throughput의 개념과 유사하게 이용할 수 있다. 이를 위해서 명령어의 수를 세는 전역 변수를 선언하고, 시간을 구하기 위해 gettimeofday() 함수를 사용한다. 쓰레드의 수를 실제보다 크게 증가시켜 테스트를 진행하도록 한다. 테스팅을 진행하여 얻은 결과를 분석한다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

서버가 pool을 이용해서 이전 연결된 client들의 read_set과 현재 연결된 ready_set을 저장한다. 루프를 통해서 서버는 Accept를 통해서 연결하고자하는 client들을 pool에 넣어서 접근하도록 한다. 이 후에 ready_set에 있는 client들을 FD_ISSET을 활용해서 loop에서 순회하면서 client들 각각이 보낸 명령어를 처리해준다. 이런 방식은 최대 1024개의 descriptor에서만 사용할 수 있고, FD_SET을 계속 운영체제에 전달하면서 생기는 overload 또한 크게 발생한다.

✓ epoll과의 차이점 서술

epoll은 위에서 서술한 overhead를 줄이는 방법이다. 운영체제가 직접 descriptor 영역을 관리해주도록 한다. 이 때 반복문을 사용하지 않고 set을 전달해줄 필요도 없어서 overhead를 줄일 수 있다.

- **Task2 (Thread-based Approach with pthread)**

  ✓ Master Thread의 Connection 관리

  Master thread가 루프를 돌면서 새로운 연결을 시도하는 client들을 공유 버퍼에 추가한다. 이 버퍼는 client들이 연결된 File descriptor를 저장해주는데, worker thread들이 자신들이 cpu에서 선택되었을 때 버퍼에서 file descriptor들을 가져와서 처리한다. 이 때 여러 thread들이 하나의 file descriptor에 접근하여 데이터 오류가 일어나는 것을 방지하기 위해 semaphore를 사용한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Worker Thread Pool에서 Ptherad_create() 함수를 통해서 생성된 thread들이 Client의 요청을 처리하고 요청이 끝나면 Pthread_detach 함수를 통해서 독립적으로 종료할 수 있도록 한다.

- **Task3 (Performance Evaluation)**

  ✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

  $$P = 10 * N / S$$

  Event-based concurrent server와 thread-based concurrent server 의 성능 비교를 해본다. 성능 비교는 동시 처리율을 기준으로 한다.

  P: 동시 처리율, N: client의 수 S: 모든 client들의 명령어를 모두 처리하는 시간

  multiclient.c 파일에서 sleep()을 제거하여 측정하도록 한다. 전체 client의 수 *10 을 적용하여 10 * N으로 설정하였다.

  client가 show만 하는 경우, buy/sell을 하는 경우, randomly하게 여러 서비스를 요청하는 경우 등의 경우의 수로 나누어서 테스트 한다.

위 과정을 세 번씩 반복하여 평균값을 구해서 비교값으로 채택하기로 한다

✓ Configuration 변화에 따른 예상 결과 서술

client 수 가 늘어감에 따라 두 서버의 동시처리율이 높아질것으로 예상된다. 그러나 계속 client수가 늘어나면 그 기울기가 낮아지는 로그함수 의 양상이 나올 것으로 예상된다. 또한 thread-based 서버가 event-based 서버보다 성능이 우수할 것이라 예상된다

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Task1(Event-driven Approach)

우선 자료구조 에서 주식 정보를 담는 구조체를 추가하고, client 명령 정보를 담는 구조체를 추가했다. 주식 구조체는 주식 고유 ID, 잔여 수량, 가격. 자식 노드를 가리키는 left, right 포인터를 담고있다. client 명령 정보 구조체에서는 명령어 종류의 cmd, 그리고 ID, num 값을 담는 변수를 멤버로 가진다.

Binary search Tree는 자료구조 강의에서 배운 연결리스트로 구현하였다. 탐색, 삽입, 메모리 해제 등의 연산은 개별 함수로 담아서 구현하였다.

File descriptor를 다루는 pool 구조체와 init_pool, add_client, check_clients 등의 함수는 멀티코어프로그래밍 강의에서 배운 코드를 활용했다.

이렇게 기본 Event-driven server의 틀을 잡고 client가 요청하는 show, buy, sell, exit 등의 서비스를 check_clients 내부에서 분기를 통해 구현하고, stock.txt 최신화 함수인 writeData()를 구현하여 개발하였다.

Task2(Thread-based Approach)

Task1에서의 구조체와 함수를 기반으로 구현하였다. 달라진 점은 pool 구조체 대신 thread를 사용하여 client들의 명령어를 처리해준다. 그리고 item 구조체에 sem_t mutex, w 를 추가하여 semaphore를 구현한다. 또한 sbuf_t 구조체 sbuf_t sbuf 전역변수 등을 사용한다. 그리고 readers -writers문제를 해결하기 위해 semaphore의 사용에 유의한다.

3. 구현 결과

- 2번의 구현 결과를 간략하게 작성
- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

다음은 task1 에서 1개의 client가 명령어를 실행하였을 때의 결과이다.

```
cse20191626@cspro:~/cse4100/project2$ ./stockclient 172.30.10.10 60042
buy 1 100
Not enough left stock
buy 1 7
sell 1 10
exit
```

```
1    1 37 17500
2    2 25 49000
3    3 73 1500
4    4 67 26000
5    5 34 22000
```

```
1    1 40 17500
2    2 25 49000
3    3 73 1500
4    4 67 26000
5    5 34 22000
```

왼쪽이 이전, 오른쪽이 이후 stock.txt이다.

buy, sell 명령어가 잘 수행되고, 그 결과가 stock.txt에 잘 기록됨을 알 수 있다.

다음은 task1 에서 multiclient가 명령어를 실행했을 때의 결과이다.

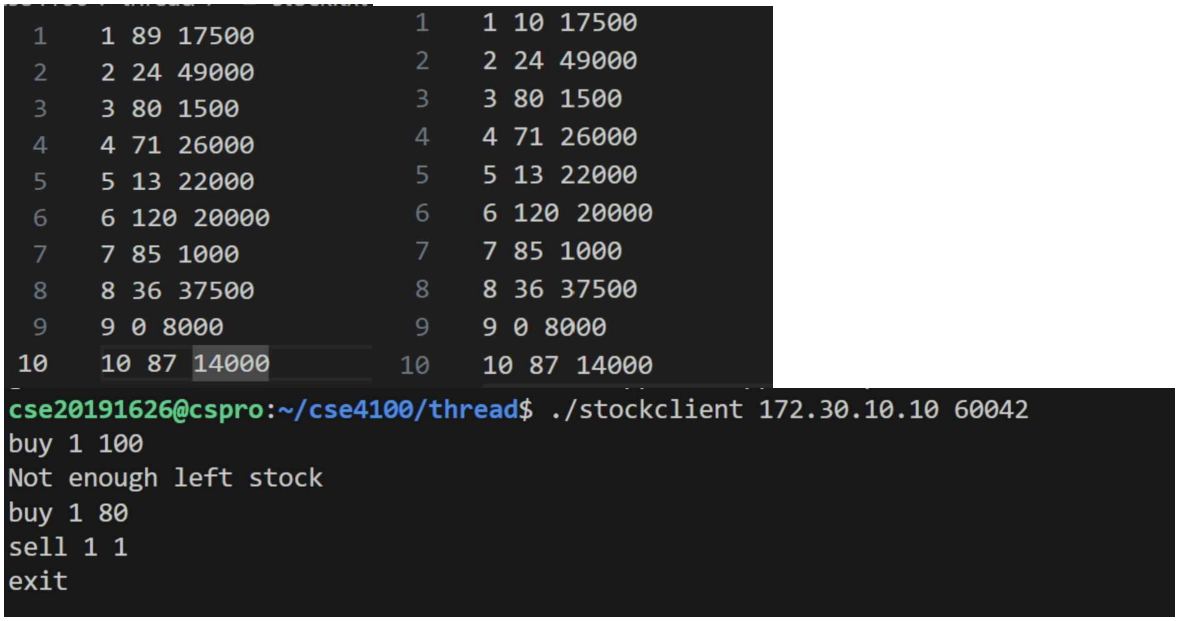
```
[sell] success
1 32 17500
2 1 49000
3 75 1500
4 66 26000
5 80 22000
6 96 20000
7 50 1000
8 42 37500
9 24 8000
10 74 14000
[sell] success
[buy] success
[sell] success
[buy] success
[sell] success
[buy] success
[buy] success
1 32 17500
2 1 49000
3 75 1500
4 66 26000
5 80 22000
6 96 20000
7 50 1000
8 42 37500
9 24 8000
10 75 14000
1 32 17500
2 1 49000
3 75 1500
4 66 26000
5 80 22000
6 96 20000
7 50 1000
8 42 37500
9 24 8000
10 75 14000
[sell] success
[buy] success
```

```
1    1 32 17500
2    2 1 49000
3    3 83 1500
4    4 59 26000
5    5 89 22000
6    6 96 20000
7    7 50 1000
8    8 42 37500
9    9 24 8000
10   10 75 14000
```

```
1    1 37 17500
2    2 1 49000
3    3 59 1500
4    4 62 26000
5    5 61 22000
6    6 91 20000
7    7 72 1000
8    8 36 37500
9    9 14 8000
10   10 77 14000
```

multiclient의 명령어가 잘 입력되고 그 결과가 stock.txt에 최신화 됨을 알 수 있다.

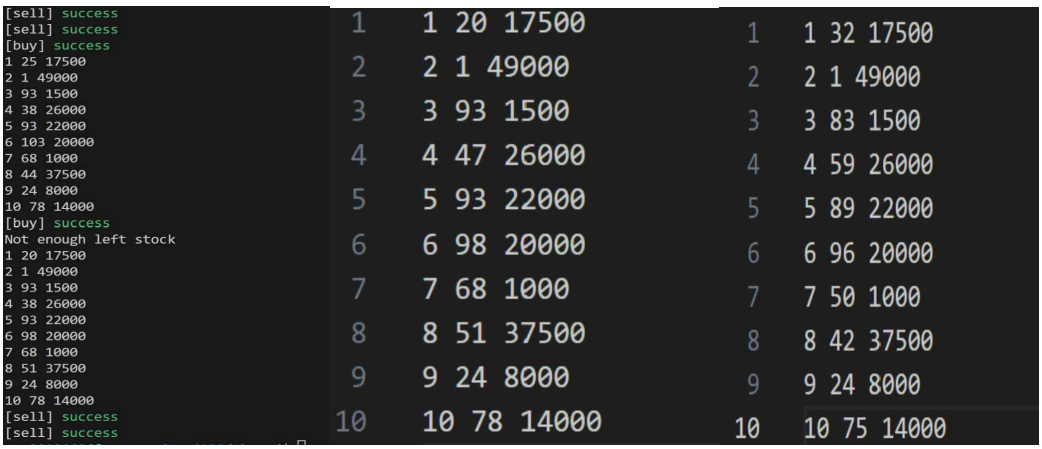
다음은 task2 에서의 1개의 client가 명령어를 수행했을 때의 결과이다.



왼쪽이 이전, 오른쪽이 이후 stock.txt이다.

buy, sell 등의 명령어가 잘 입력 되고, 그 결과가 stock.txt에 최신화 됨을 알 수 있다.

다음은 task2에서 multiclient가 명령어를 입력했을 때의 결과이다.



buy, sell 등의 명령어가 잘 수행되고, 그 결과가 stock.txt에 최신화 됨을 알 수 있다.

4. 성능 평가 결과 (Task 3)

task3 에서는 multiclient.c 파일에서 MAX_CLIENT값을 100, ORDER_PER_CLIENT 값은 10, STOCK_NUM을 10, BUY_SELL_MAX 값을 10으로 설정하고 원하는 출력 값을 얻기 위해 child 출력 부분과 Fputs, 그리고 usleep을 주석처리한 후에 진행한다.

1. Show 만 요청하는 경우

1.1 Event-driven

```
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.021975 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.040592 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.052631 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.071244 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.081817 sec
```

1.2 Thread-based

```
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.035091 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.048266 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.064122 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.079385 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.091739 sec
```

2. Buy or Sell만 요청하는 경우

2.1 Event-driven

```
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.030105 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.040400 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.057216 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.068246 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.081596 sec
```

2.2 Thread-based

```
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.036092 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.048115 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.060759 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.081197 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.094071 sec
```

3.. Randomly하게 요청하는 경우

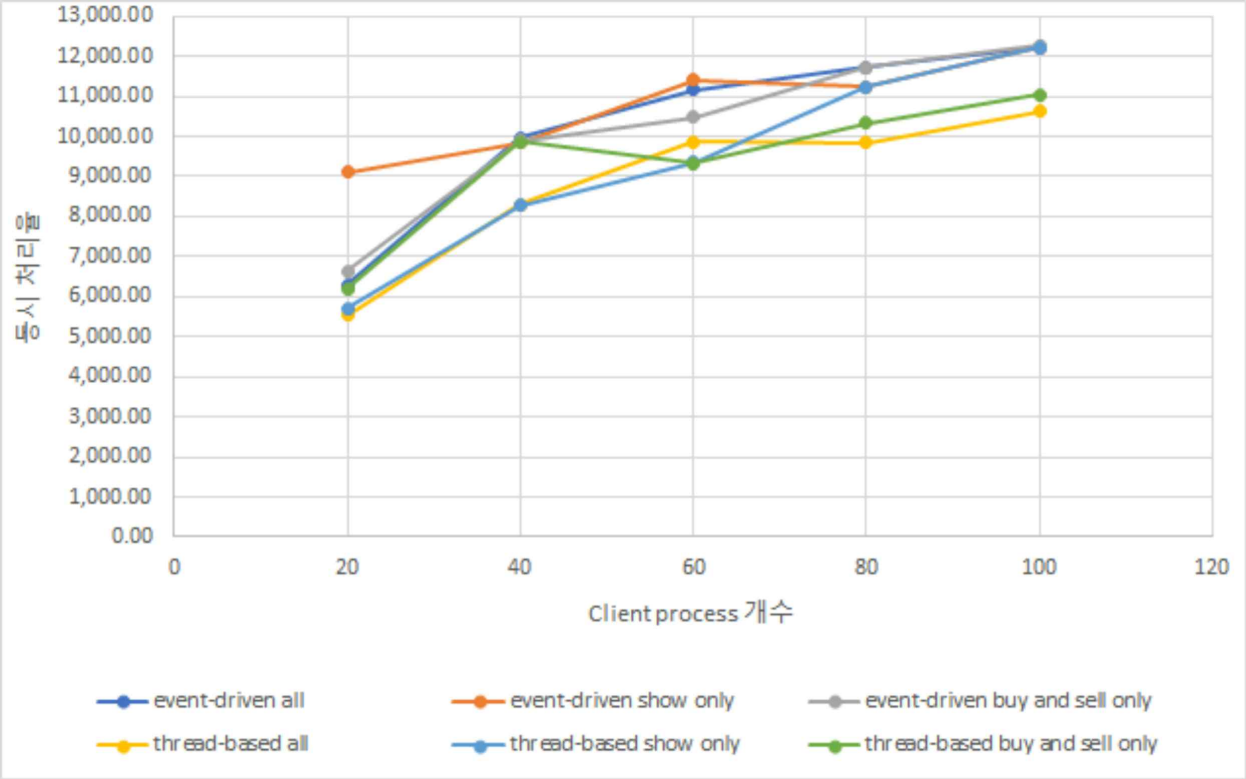
3.1 Event-driven

```
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.031707 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.040161 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.039365 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.053790 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.068252 sec
cse20191626@cspro:~/cse4100/project2$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.081668 sec
```

3.2 Thread-based

```
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 20
elapsed time : 0.032314 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 40
elapsed time : 0.040507 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 60
elapsed time : 0.064355 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 80
elapsed time : 0.077354 sec
cse20191626@cspro:~/cse4100/thread$ ./multiclient 172.30.10.10 60042 100
elapsed time : 0.090620 sec
```

client number	event-driven			thread-based		
	all	show only	buy and sell only	all	show only	buy and sell only
20	6,307.76	9,101.25	6,643.41	5,541.39	5,699.47	6,189.27
40	9,959.91	9,854.16	9,900.99	8,313.42	8,287.41	9,874.84
60	11,154.49	11,400.13	10,486.58	9,875.08	9357.162908	9,323.28
80	11,721.27	11,229.02	11,722.30	9,852.58	11,229.02	10,342.06
100	12,244.70	12,222.40	12,255.50	10,630.27	12,222.40	11,035.09



서로 다른 방식의 두 서버의 능률을 비교해보자.

Event- driven 서버는 평균적으로 client process의 개수가 증가할 때 동시처리율이 미미한 변화를 보이는 것을 확인할 수 있었다. 이는 한 번에 하나의 명령만을 처리할 수 있는 싱글 쓰레드의 구현이기 때문에 client process의 수의 변화가 큰 의미를 갖지 않는 것이다.

그리고 Event-driven 서버에서는 semaphore를 사용하지 않아 show만 입력받는 경우, buy or sell만 입력받는 경우 , 모든 명령어를 입력받는 경우 모두 큰 차이를 보이지 않는다.

반면에 thread-based 서버의 경우 4개의 thread가 각자 client 로부터 명령을 전

달받아 그 결과를 concurrent하게 처리하기 때문에 client process의 개수 변화에 큰 영향을 받는다. client의 수가 증가함에 따라 동시처리율이 크게 변화함을 알 수 있었다. 또한 data corruption을 방지하기 위해서 semaphore를 사용하는 점, readers - writers problem을 해결하기 위해 read_cnt를 사용하는 점이 차이이다. 이런 차이로부터 event-driven 서버와 달리 randomly하게 모든 명령어를 입력받는 경우와, show만을 입력받는 경우들의 동시처리율 편차가 유의미하게 나타난다. 그 중에서 모든 명령어를 처리하는 서버가 동시처리율이 가장 낮게 나타남을 확인할 수 있었다.