System Programming Project 3

담당 교수 : 박성용

이름 : 조원빈

학번 : 20201644

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

이번 프로젝트에서는 동시 주식 서버(Concurrent Stock Server)를 설계하였다. 우선, 서버를 구현하면서 client로부터 기본적인 명령어 buy, sell, show(주식 상태를 보여줌), exit를 입력으로 받아 수행할 수 있도록 후현하였다. 그리고 서버는 여러 client의 동시 접속 및 서비스를 수행할 수 있어야 하는데 이를 위해 Event-driven Approach와 Thread-based Approach의 두 가지 방법을 사용해보았으며, 각각의 경우의 성능에 대해 분석해보았다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   우선 client로 부터의 여러 명령어를 수행한후 결과를 보낼 수 있도록 함수들을 먼저 구현하였고, Event-driven Approach를 통해 여러 clinet들이 서버에 동시에 접속하고 작업을 수행할 수 있도록 하였다. listen file descriptor를 통해 client의 connect 요청을 accept하고, select 함수를 통해 ready상태의 descriptor를 찾은 후 각 connected descriptor에 대해 명령을 한 줄 씩 차례대로 수행한다.

2. Task 2: Thread-based Approach

   Task 2에서는 여러 개의 thread를 이요하는데, 각각의 thread에서 하나의 client를 관리하면서 server가 여러 client를 동시에 관리하도록 하였다. 또한 thread의 병렬 처리의 특성 덕분에 앞의 구현보다 더 효율적으로 작동하도록 하였다.

3. Task 3: Performance Evaluation

   Event-driven 방식을 사용했을 때와 Thread-based 방식을 사용했을 때 성능의 차이를 분석하였다. 분석은 clinet의 수와 요청의 수를 변화시켜 가면서

시간을 측정하고 동시 처리율을 그래프로 나타내는 등 다양한 관점에서 수행하였다.

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

    여러 client들의 descriptor를 보고 event가 있는 지 확인한다. event가 있는 경우 각 file descriptor는 ready상태가 되고, ready상태가 된 discriptor에 대해 입력으로 받은 client의 명령 수행한다. 이를 통해 client부터 하나 또는 여러 개의 I/O event를 제어할 수 있다.
  - ✓ epoll과의 차이점 서술

    기존에 select에서는 함수 내에서 event가 발생한 descriptor를 ready상태로 설정한 후, 모든 파일 디스크립터를 순회하면서 ready_set에서 비트 값이 set 된 descriptor를 찾아 작업을 수행하기 때문에 비효율적인 면이 있다. 하지만 epoll에서는 fd의 상태 변화를 kernel에서 관리하기 때문에 직접 루프 문을 통해 모든 descriptor를 확인할 필요가 없다. 따라서 select보다 더 효율적으로 작동한다.
- **Task2 (Thread-based Approach with pthread)**
  - ✓ Master Thread의 Connection 관리

    이번 task에서는 pre-threaded Concurrent Server 방식을 이용한다. Master Thread에서 worker thread의 pool을 미리 만들어 놓는데, 인자로 전달된 각 thread 함수에서 sbuf를 통해 connected file descriptor를 전달 받고 client로부터 input을 받도록 구현하였다. 즉 각각의 worker thrad에서 client로부터의 요청을 처리하게 되고, master thread에서는 client로부터의 connect 요청을 입력받아 accept하는 역할을 한다.
  - ✓ Worker Thread Pool 관리하는 부분에 대해 서술

앞서 말한 것과 비슷하게 Pthread_create를 통해 thread를 미리 만들어 놓고, buffer를 통해 client의 connect상태를 전달한다. 구체적으로 Master thread에서 connected file descriptor를 sbuf에 insert하면 thread를 만들 때 인자로 전달한 함수에서 sbuf_remove() 함수를 통해 받아와 client로의 명령을 수행하고 결과를 전달하는 기능을 할 수 있게 된다.

- **Task3 (Performance Evaluation)**

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

  첫 번째로는 수행 시간(elapsed time)을 측정하였다. 시간은 프로그램의 성능을 판단할 수 있는 가장 간단하면서도 중요한 기준이기 때문에 가장 먼저 측정하였고, multiclient 파일 내에서 gettimeofday() 함수를 이용해 main함수 내에서 타이머를 시작하고 모든 명령을 다 수행한 후의 시간과 시작 시간을 비교하는 방법으로 측정하였다. 또한 이를 이용해 동시 처리율을 비교하였다. 동시처리율은 수행한 요청 수를 걸린 시간(millisecond)으로 나누어 측정하였고, 서버의 성능을 판단하는 중요한 지표라고 판단되어 분석하게 되었다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  우선 client 수가 증가함에 따라 thread-based 방식이 event-driven방식보다 더 좋은 성능을 보일 것이라고 예상하였다. 왜냐하면 thread-based 방식을 사용하면 multi-thread를 이용해 좀 더 유연하게 작동할 수 있는 반면, event-driven 방식에서는 한 번에 하나의 client에서만 요청을 수행하며, select 함수 내에서 또는 요청을 처리할 때 전체 client를 순회하는데, client 수가 증가할 수록 loop문에서 잡아먹는 시간이 증가할 것이기 때문이다. 다만 thread-based방식을 사용하더라고 context switch 비용이 증가하므로 client가 증가에 비례해 성능이 좋아지지는 않을 것이다.

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 with select())

```
/* connected된 descriptor를 저장하기 위한 구조체(pool)*/
typedef struct{
    int maxfd;
    fd_set read_set, ready_set;
    int nready;
    int maxi;
    int clientfd[FD_SETSIZE];    // FD_SETSIZE = 1024
    rio_t clientrio[FD_SETSIZE];
} Pool;
```

Pool 구조체를 통해 connected된 file descriptor를 관리하였다. maxfd와 maxi를 통해 가장 큰 file descriptor값과 index값을 저장하고, nready에 ready된 fd의 개수를 관리하여 루프 문을 돌 때 좀 더 효율적으로 작동하도록 구현하였다.

```
void add_client(intconnfd, Pool*p)
```

add_client 함수에서는 새로운 client가 connect요청을 했을 때 이를 accept하고 pool에 추가하는 작업을 한다. 구체적으로 설명하자면, for문을 돌면서 clientfd의 i번 slot이 비어있으면 거기에 descriptor를 추가하고, readset을 갱신하여 앞으로 이 descriptor를 통해 입력을 받을 수 있도록 하였다.

```
void check_clients(Pool*p)
```

이 함수에서는 for문을 돌면서 각 slot에 대해 descriptor가 존재하고 pending input이 있는 경우, 즉 ready 상태인 경우 client로 부터 한 줄 씩 input을 받아 명령을 수행하는 기능을 한다.

- Task2 (Thread-based Approach with pthread)

```c
typedef struct {
    int *buf; /* Buffer array */
    int n; /* Maximum number of slots */
    int front; /* buf[(front+1)%n] is the first item */
    int rear; /* buf[rear%n] is the last item */
    sem_t mutex; /* Protects accesses to buf */
    sem_t slots; /* Counts available slots */
    sem_t items; /* Counts available items */
} sbuf_t;
```

master thread에서 worker thread로 connected file descriptor값을 안전하게 전달하기 위해 Semaphore 방법을 사용하였고 이를 위해 sbuf 구조체를 정의하였다. sbuf_init()함수에서 전달 받은 변수 n만큼으 크기를 slot으로 갖게 되고 fd는 item의 형태로 master thread에서 worker thread로 전달되며, mutex를 통해 buf로의 접근을 통제함으로써 synchronization이 가능하도록 하였다.

```c
void sbuf_insert(sbuf_t*sp, intitem)
```

Master thread에서 client로부터 connect 요청을 받으면 file descriptor를 worker thread로 전달해야 하는데 이 때 sbuf_insert 함수를 이용한다. sbuf_insert 함수 내에서는 item이 생기면 mutex를 통해 buffer를 lock하고 item을 buf배열에 추가한 후 unlock하는 과정을 수행한다.

```c
int sbuf_remove(sbuf_t*sp)
```

worker thread에서는 master thread로부터 connected file descriptor를 받아야 하는데 이 때 sbuf_remove() 함수를 사용한다. sbuf_remove() 함수는 insert 함수와 비슷하게 mutex shared variable 때문에 발생하는 문제를 방지하기 위해 mutex를 이용하여 buf를 갱신하는 동안 다른 thread로부터의 접근을 제한한다.

- Task3 (성능 분석)

우선 시간을 측정하기 위해 sys/time.h헤더에 있는 gettimeofday() 함수를 이용했다.

```c
void start_timer(struct timeval*start)

long end_timer(struct timeval*start)
```

구체적으로는 gettimeofday() 함수를 사용하는 두 가지 함수를 새로 정의했는데,

start_timer 함수 프로그램의 시작부에서 에서는 timeval를 가지는 start를 현재 시간으로 초기화해주고, end_timer에서는 프로그램의 종료 전에 현재 시간을 시작 시간과 비교해 얼마나 시간이 걸렸는 지를 ms단위로 측정해 반환해주도록 함수를 정의하였다.

```
printf("number of clients : %d\n", num_client);
printf("total order is %d\n", ORDER_PER_CLIENT * num_client);
printf("elapsed time: %ldms\n", elapsed_time);
printf("tasks per ms : %0.3lf\n", 1.0 * ORDER_PER_CLIENT * num_client / elapsed_time);
```

또한 결과를 측정한 후 분석을 위해 위와 같이 4개의 측정값(client 수, 요청 개수, 걸린 시간, ms당 처리한 요청의 개수)를 출력하였다.

3. 구현 결과

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

- Task1 (Event-driven Approach with select())

Pool을 통해 connected descriptor를 관리하면서 server에서 여러 client를 관리할 수 있게 되었다. main 함수 내에서 우선 Select 함수를 통해 pending input이 있는 descriptor를 뽑아 ready_set에 체크를 한다. listenfd를 통해 새로운 client로부터 connect 요청을 받으면 add_client 함수를 통해 pool에 추가하였고, check_client 함수를 통해 매 루프문 안에서 각 client로부터 받은 명령을 수행하고 결과를 client에게 전송하였다.

- Task2 (Thread-based Approach with pthread)

Task2에서는 Master thread에서 client의 connect 요청을 관리하고, workerthread에서 client의 명령을 병렬적으로 처리하도록 구현하였다. Task1에서는 connected client 수가 0이 되면 서버가 종료되는 부분도 구현을 했는데, Task2에서는 이 부분을 구현하지 못했다. pre-threaded Concurrent Server방식을 사용해 미리 thread를 create해놓아서 효과적으로 client의 수를 세는 방법을 제대로 찾지 못했는데, sbuf_deinit() 함수를 이용하면 현재 connect된 client의 수를 셀 수 있을 것 같기도 하다.

- Task3

configuration(client 수, client 당 요청의 개수, 주식의 개수, 사고 파는 주식의

최대 개수 등을) 바꿔가면서 여러 경우에 대해 아래와 같이 출력하면서 성능을 비교분석해보았다.

```
number of clients : 200
total order is 2000
elapsed time: 156ms
tasks per ms : 12.821
```

다만, multiclient의 코드를 보면 tmp배열의 길이가 3인데 주식 종류의 개수가 100(세 자리)인 경우 널 문자까지 포함해 총 4자리가 필요해 배열의 범위를 벗어나는데 multiclient code는 수정하지 말라고 하셔서 이런 corner case의 경우 문제가 발생할 수 있을 것 같다

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

분석을 하기에 앞서서 multiclient 파일의 코드를 살짝 수정하였다. 기존에는 여러 client의 요청을 어떻게 처리하는 지 좀 더 가시적으로 보여주기 위해 usleep() 함수가 있었는데 성능을 제대로 분석하기 위해 이 부분을 없애주었고, 입출력 시간을 수행 시간에서 제외하기 위해 결과를 터미널에 출력하는 코드도 없애주었다.

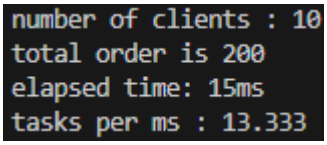
- **확장성**

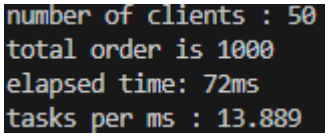
두 가지 방법의 성능을 평가하기 위해 우선 확장성을 분석해보았다. client 당 oreder의 개수는 20개로 고정했으며, 각각의 방법에 대해 각각 client 수가 변함에 따라 동시 처리율(시간 당 client 요청 처리 개수인데 여기서는 ms당 요청 처리 개수를 측정)이 어떻게 변화하는 지 보았다. client의 수를 1부터 시작해 증가시키면서 동시처리율의 변화를 그래프로 나타냈는데, 좀 더 직관적인 이해를 위해 client의 수가 1, 5, 10, 50, 100, 500, 1000일 때의 출력 결과를 첨부하였다.

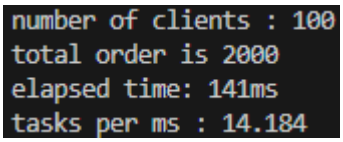
- event-driven Approach에서 결과

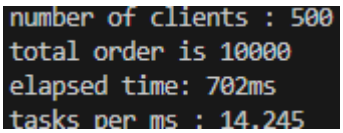
```
number of clients : 1
total order is 20
elapsed time: 8ms
tasks per ms : 2.500
```

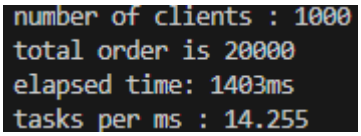
```
number of clients : 5
total order is 100
elapsed time: 9ms
tasks per ms : 11.111
```



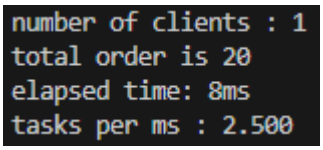


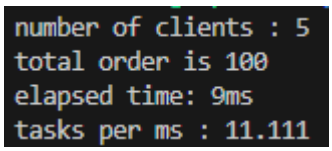


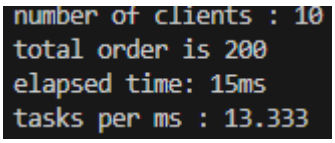


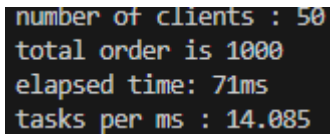


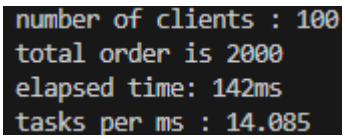
- thread-based Approach

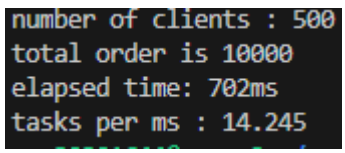


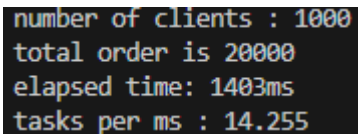




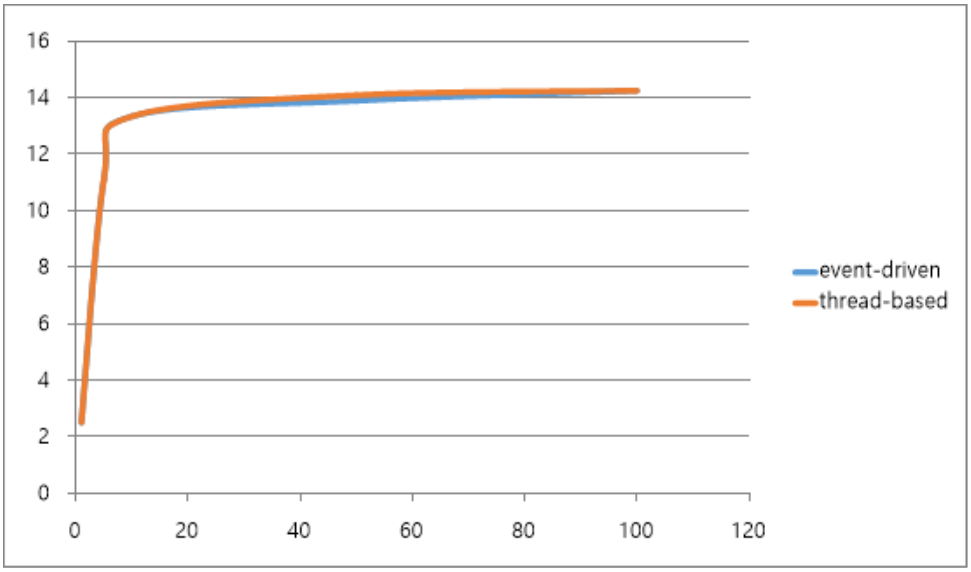








우선 분석을 진행하기 전 예상했던 바와 같이 client 수가 증가함에 따라 동시 처리율이 증가하는 것을 확인할 수 있었다. 다만, client수가 증가함에 따라 thread-based Approach 방식이 event-driven 방식보다 더 좋은 동시 처리율을 보일 것이라고 예상했는데, 예상과는 달리 두 방식에서 차이를 확인할 수 없었다. client 수 변화에 따른 동시처리율의 변화를 아래에 그래프로 나타내보았다.



client가 100 이하일 때 프로젝트 명세서에 있는 그래프와 비슷하게 동시처리율은 계속해서 증가하되 증가율은 감소하는 경향성을 확인할 수 있다. 다만 두 방식의 성능 차이는 확인할 수 없었다. 왜 이런 결과가 나왔는 지 이유에 대해 생각해 보았고, 다음과 같은 가설을 세워보았다.

1. elapsed time에 더 큰 영향을 미치는 다른 요인이 있다.

2. 처리해야 하는 명령이 별로 무겁지 않아 오는 순서대로 하나씩 수행해도 큰 문제가 없다.

우선 첫 번째 가설을 검증하기 위해 elapsed time에 큰 영향을 줄 다른 요인에 대해 생각해보았다. 시간 측정은 thread-based방식을 기준으로 client 수가 100이고, client당 order의 개수가 20개일 때를 기준으로 하였다.

- 아무 것도 수정하지 않은 경우

```
number of clients : 100
total order is 2000
elapsed time: 142ms
tasks per ms : 14.085
```

- server에서 client로 결과를 보낼 때 버퍼의 크기를 줄인 경우(입출력 시간 단축)

```
number of clients : 100
total order is 2000
elapsed time: 141ms
tasks per ms : 14.184
```

- client 당 oreder의 개수를 늘림(initialize하는데 드는 비용의 영향을 줄임)

```
number of clients : 100
total order is 10000
elapsed time: 703ms
tasks per ms : 14.225
```

이외에도 여러 가지 시도를 해봤으나 유의미하게 시간이 줄은 경우는 없었다.

따라서 두 번째 가설로 넘어가 '처리해야 하는 명령이 별로 무겁지 않아 오는 순서대로 하나씩 수행해도 큰 문제가 없다.'가 원인이라고 가정하였다. 이 가설을 검증하기 위해 매 요청마다 아래의 코드를 추가로 수행하도록 구현한 후 결과를 확인해 보았다.

```
usleep(10000); // 한 명령을 수행하는데 10ms의 추가적인 시간이 필요함
```

단, 제대로된 실행 결과를 얻기 위해 thread-based approach에서 mutex로 lock이 되지 않은 code부분에 넣어줘야 한다. lock이 된 부분에 넣으면 unlock이 될 때까지 다른 thread에서 실행할 수 없으므로 multi-thread의 역할을 수행할 수 없기 때문이다.

- event-driven Approach Approach (client 수가 100인 경우까지만 측정)

```
number of clients : 1
total order is 10
elapsed time: 93ms
tasks per ms : 0.108
```

```
number of clients : 5
total order is 50
elapsed time: 541ms
tasks per ms : 0.092
```

```
number of clients : 10
total order is 100
elapsed time: 1096ms
tasks per ms : 0.091
```

```
number of clients : 50
total order is 500
elapsed time: 5553ms
tasks per ms : 0.090
```

```
number of clients : 100
total order is 1000
elapsed time: 11131ms
tasks per ms : 0.090
```

- thread-based Approach

```
number of clients : 1
total order is 10
elapsed time: 93ms
tasks per ms : 0.108
```

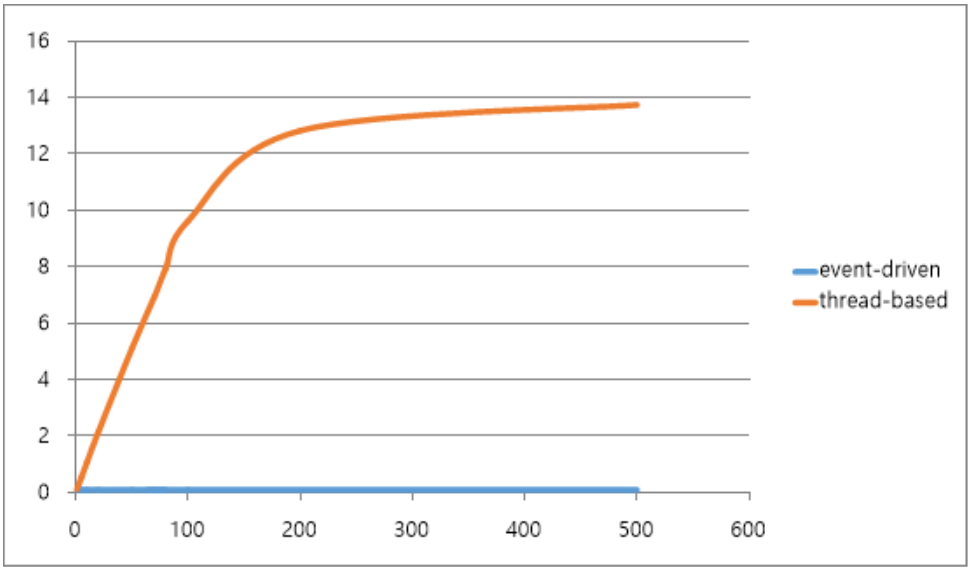
```
number of clients : 5
total order is 50
elapsed time: 94ms
tasks per ms : 0.532
```

```
number of clients : 10
total order is 100
elapsed time: 94ms
tasks per ms : 1.064
```

```
number of clients : 50
total order is 500
elapsed time: 98ms
tasks per ms : 5.102
```

```
number of clients : 100
total order is 1000
elapsed time: 104ms
tasks per ms : 9.615
```

```
number of clients : 500
total order is 5000
elapsed time: 364ms
tasks per ms : 13.736
```



client 수가 500 이하일 때 thread-based 방식으로 구현했을 때 원하는 형태의 그래프를 얻을 수 있었다. 왜 그럴까 이유를 생각해보았는데 **기존의 방식에서는 synchronization을 위해 semaphore를 통해 한 thread에서 stock 정보를 참조할 때 다른 thread의 접근을 막고 있었고, 이러한 이유로 multi-thread의 효율성을 충분히 반영할 수 없었을 것이다**. 하지만 두 번째 가설을 검증하기 위해 코드를 수정한 것처럼 lock이 된 block의 바깥에서 연산을 수행할 경우 multi-thread를 이용하는 것이 훨씬 더 효율적임을 알 수 있다. 다만, 주식 서버의 경우 대부분의 연산이 저장된 stock 정보를 참조하는 것이기 때문에 multi-thread의 장점을 극대화하기에는 쉽지 않은 것 같다.

- **워크로드에 따른 분석**

client가 buy/sell을 요청한 경우와 show를 요청한 경우 동시처리율이 어떻게 변화하는지 분석해보았다. 여기서는 client의 수는 100, clinet당 oreder의 개수는 20개로 고정하였다. 우선 각각의 경우에 대해 thread-based 방식과 event-driven 방식에는 차이가 없었다. 따라서 thread-based 방식의 결과를 정리해보았다.

<buy/sell 명령만 요청한 경우>

```
number of clients : 100
total order is 2000
elapsed time: 142ms
tasks per ms : 14.085
```

<show 명령만 요청한 경우>

```
number of clients : 100
total order is 2000
elapsed time: 141ms
tasks per ms : 14.184
```

여기서 가장 눈에 띄는 부분의 두 명령을 수행했을 때에 수행 시간의 차이가 없다는 것이다. 서버를 어떻게 구현하느냐에 따라 결과가 다르게 나오겠지만, 여기서는 서버에서 client로 전송할 때 client가 Rio_readnb()를 통해 입력을 받는데, 여기서 제대로 입력을 전달하기 위해 매 번 MAXBUF크기 만큼의 문자열을 전송한다. 따라서 buy명령이 대체로 더 많은 값을 출력하지만 시간에는 차이가 없었다. 그렇다면 client에서 결과를 출력하는 부분을 추가하면 결과가 어떻게 달라질까?

<buy/sell 명령만 요청한 경우>

```
number of clients : 100
total order is 2000
elapsed time: 142ms
```

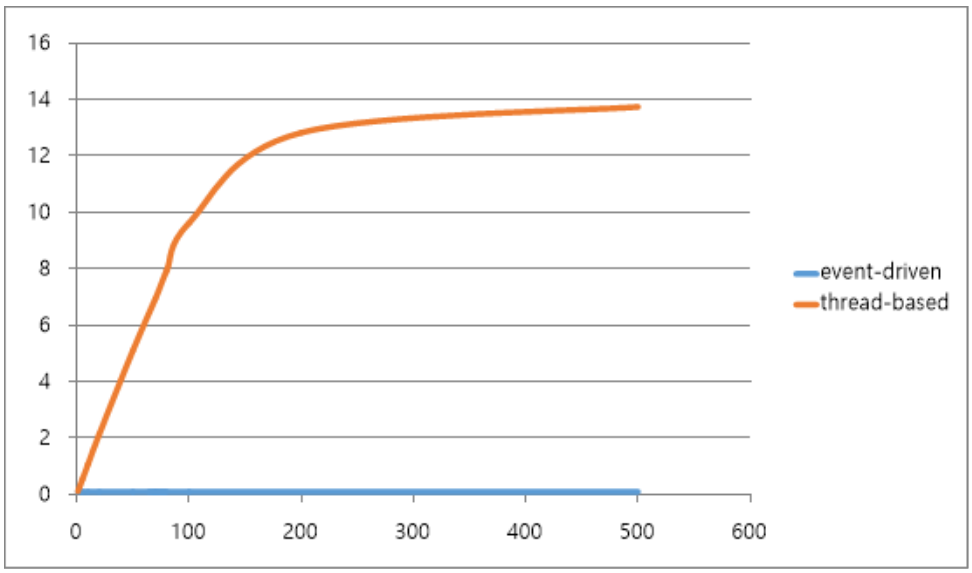
<show 명령만 요청한 경우>

```
number of clients : 100
total order is 2000
elapsed time: 2013ms
```

이 경우 client에서 결과를 출력하는 시간은 서버의 동시처리율을 평가하기에는 적합한 요소가 아니므로 동시처리율의 결과는 제외하였다. 이 경우에는 **buy/sell 명령만 요청한 경우게 show 명령만 요청한 경우보다 약 14배 정도 빨랐다**. 이는 buy명령을 수행하면 모든 주식 정보를 출력해야 하기 때문에 시간이 많이 걸리기 때문이다. 만약 server에서 client로 값을 전송할 때도 문자열의 크기 만큼만 전송한다면 이와 비슷한 결과가 나올

것이라고 예상된다.

- **semaphore의 overhead**



앞서 분석한 내용을 갖고 오면 thread-based 방식을 쓰더라도 client 당 동시 처리율을 그래프로 나타냈을 때 그래프가 꺾이는 것을 확인할 수 있다. 그 이유에 대해 생각해보면 semaphore의 overhead의 영향이 클 것이라고 생각되는데, client 수가 증가함에 따라 context switching이 더 많이 일어나게 된다. 따라서 운영체제가 thread의 상태를 저장하고 다른 thread로 왔다 갔다 하는 과정에서 비용이 발생하게 된다. 또한 mutex를 이용해 여러 thread가 주식 정보에 동시 접근하는 것을 막는 데 이 때문에 lock상태로 인해 기다리는 시간이 많아져 성능이 떨어졌을 가능성도 높아보인다.