System Programming Project 2

담당 교수 : 김영재

이름 : 임종환

학번 : 20171683

1. 개발 목표

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

이번 프로젝트의 목표는 Event-driven Approach와 Thread-based Approach를 이용하여 Concurrent한 주식 서버를 만드는 것이다. 주식 서버는 주식 종목들에 대한 정보(주식 ID, 구매 가능한 주식의 수, 가격)를 가지고 있다. 서버가 실행되는 동안 여러 client들의 동시 접속을 지원하고 각 client에서 들어오는 요청들(show, buy, sell, exit)을 처리한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task1: Event-driven Approach

   Event-driven Approach에서 서버는 listening fd, client와 연결된 connected fd를 저장하고 있는 pool을 가지고 있다.

   서버는 Select 함수를 호출하여 pool에 저장되어 있는 fd들을 monitoring한다. 특정 fd에 event가 발생하면 이를 감지하고 해당 event를 처리한다. Listening fd에 event가 발생한 경우 새로운 client로부터 connection 요청이 온 것으로, 이를 Accept하여 fd pool에 추가한다.

   Connected fd에 event가 발생한 경우 client로부터 요청이 들어온 것으로, command 입력에 따라 요청을 처리한다. 서버는 위와 같은 과정을 반복하며 여러 client로부터 들어오는 요청을 concurrent하게 처리한다.

2. Task2: Thread-based Approach

   Thread-based Approach에서 서버는 새로운 client의 connection 요청을 처리하는 Master thread와, 연결된 각 client와 connect되어 client의 요청을 처리하는 Worker thread들이 있다.

   Master thread가 client의 connection을 Accept하여 연결된 connfd를 모든 thread들이 공유하는 Shared Buffer에 삽입한다. Worker thread는 Shared Buffer에 connfd가 삽입될 때까지 wait하다가, connfd가 삽입되면 wait중이던

Worker thread 하나가 Shared Buffer의 connfd를 제거하고 해당 client와 연결되어 communication을 한다.

각 client의 요청은 연결된 Worker thread가 처리되고, 여러 client로부터 들어오는 요청은 각 Worker Thread에 의해 동시에 처리된다. 각 요청을 처리할 때, 모든 thread들이 share하고 있는 자료구조에 접근할 경우 synchronize를 해서 thread safe한 concurrent 서버를 구현한다.

3. Task3: Performance Evaluation

실험의 configuration과 동시 접속하는 client 수를 변화시키면서 Event-driven Approach와 Thread-based Approach로 구현한 서버의 elapse time과 throughput을 측정하여 performance를 비교한다.

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

Task1은 Single process에서 I/O Multiplexing을 통한 concurrent 서버를 구현한다. Client와 연결된 connfd 정보를 저장하고 있는 fd pool을 monitoring하면서 Multi-client로부터 들어오는 요청을 처리한다.

Client가 서버에 요청을 보내면, client와 연결된 connfd의 pending input이 setting된다. 서버는 Select 함수를 통해 connfd로 오는 pending input을 확인한다. Pending input이 도착한, 즉 event가 발생한 connfd는 client의 요청이 들어온 것이다. 서버는 event가 발생한 모든 client의 요청을 처리한다.

✓ epoll과의 차이점 서술

select() 함수를 호출할 때 인자로 monitoring할 fd pool을 넘겨준다. 함수는 fd pool을 모두 순회하며 각 fd의 event 발생 여부를 확인하고 event가 발생한 fd의 개수를 return한다. 서버는 다시 fd pool을 순회하며 event가 발생한 fd들을 처리한다. 이와 같은 방식은 **select**() 함수를 호출할 때마다 fd pool을 순회하며 event를 처리해야 하는 단점이 있다.

epoll은 **epoll_wait**() 함수를 호출할 때 인자로 fd pool과 event가 발생한 fd를 저장할 buffer를 넘겨준다. 함수는 event가 발생한 fd들을 buffer에 추가하고 event가 발생한 fd의 개수를 return한다. 서버는 return 받은 event 개수를 통해 buffer를 순회하며 event가 발생한 fd들을 처리한다. **select**() 함수에서 모든 fd pool을 순회했던 것과는 달리, **epoll**은 event가 발생한 fd들을 순회한다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 client의 connection 요청을 처리한다. Listening fd를 통해 새로운 client로부터 connection 요청이 오면 이를 Accept하여 해당 client와 communication이 가능한 connfd를 생성하고, 이를 Shared Buffer에 삽입한다. Shared Buffer의 모든 slot이 가득 차 있다면 Worker Thread가 buffer의 connfd를 제거하여 빈 slot이 생길 때 까지 대기한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  생성된 Worker Thread들은 Shared Buffer에 connfd가 삽입될 때까지 wait한다. Master Thread가 Shared Buffer에 connfd를 삽입하면, wait중이던 Worker Thread 하나가 깨어나 Shared Buffer에 삽입된 connfd를 제거하고 해당 client와 연결되어 communicate한다.

- **Task3 (Performance Evaluation)**

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

  이번 프로젝트는 Concurrent한 주식 서버를 만들어 여러 client의 동시 접속과 요청을 처리하는 것이다. Concurrent 서버의 performance는 서버로 매우 많은 client의 요청이 들어왔을 때, 이를 얼마나 빠르고 정확하게 처리할 수 있는지에 달려있다.

  서버의 performance를 평가하기 위해 **multiclient** 프로그램을 실행시켜 여러 client의 모든 요청들을 처리하는데 걸리는 시간(**elapsed time(ms)**)을 측정한다. client의 수와 각 client의 주문 개수를 곱하고, 이를 측정한 **elapsed time**으로 나눈 다음 10^6을 곱하여 동시 처리율(**throughput(order/s)**)을 구하여 서버가 시간당 처리하는 client의 요청 개수를 구한다. **Throughput**이 높을수

록 더 높은 performance를 의미한다.

동시 접속하는 client 수, order type(show only, buy only) 등의 configuration을 변화를 주어 다양한 환경에서의 event-based 서버와 thread-based 서버의 performance를 측정한다.

✓ Configuration 변화에 따른 예상 결과 서술

Event-based concurrent 서버는 single-process에서 I/O Multiplexing을 통해 작동한다. **select** 함수를 통해 fd pool을 순회하며 event가 발생한 client의 요청을 순차적으로 처리해준다. 동시 접속하는 client수가 증가할수록 순회하는 fd pool도 커지기 때문에 성능이 낮아질 것으로 예상된다.

그에 반해 Thread-based 서버는 multiple client의 요청을 Worker Thread가 각각 동시에 처리해줄 수 있다. 동시 접속하는 client수가 증가하더라도 성능이 유지될 것으로 예상된다.

따라서 동시 접속하는 client의 수가 증가할수록 Thread-based 서버의 성능이 Event-based 서버보다 더 좋을 것으로 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

- pool 구조체: Listening fd와 client와 연결된 connected fd를 관리하는 구조체

- Select: 서버가 실행되는 동안 반복적으로 호출된다. **pool**을 monitoring하여 listening fd, connected fd의 event 발생을 감지한다.

- init_pool 함수: 서버를 실행하면 listening fd를 생성한 후 처음 호출되어, **pool**에 listening fd를 삽입한다.

- add_client 함수: Select를 통해 listening fd에 event 발생이 감지된 경우 호출된다. 새로운 client로부터 온 connection 요청을 Accept하고 생성된 connfd를 **pool**에 삽입한다.

- check_clients 함수: Select를 통해 listening fd를 제외한 connfd에 event 발생이 감지된 경우 호출된다. **pool**을 모두 순회하면서 event가 발생한 모든 client의 요청을 처리한다. 특정 client가 무수히 많은 요청을 보냈을 때 해당 client의 요청을 한 번에 모두 처리하려고 하면 fine-grain하지 못하다. 이를 보완하기 위해 각 client의 요청을 한 줄씩 처리한다.

✓ Task2

- sbuf_t 구조체 sbuf: Master Thread에서 client의 connection 요청을 Accept하여 생긴 connfd를 저장하고 있는 Shared Buffer.

- sbuf_init 함수: 서버를 실행하면 호출되어 **sbuf**를 초기화한다.

- sbuf_deinit 함수: sbuf를 free한다.

- sbuf_insert 함수: Master Thread에서 새로운 client의 connection 요청을 Accept할 때마다 호출된다. **sbuf**에 빈 slot이 존재할 경우 buffer에 새로운 item(connfd)를 삽입한다. Slot이 꽉 차서 빈 slot이 존재하지 않을 경우 **sbuf_remove** 함수로부터 connfd가 제거되어 빈 slot이 나올 때 까지 대기한다.

- sbuf_remove 함수: 각 Worker Thread에서 호출된다. **sbuf**에 connfd가 존재할 경우 buffer에서 connfd를 제거하고 해당 connfd를 return한다. **sbuf**에 connfd가 존재하지 않을 경우 **sbuf_insert** 함수로부터 connfd가 삽입될 때까지 대기한다.

- *thread 함수: 생성된 worker thread들의 작동 함수. 각 Worker Thread들은 **sbuf_remove** 함수를 호출하여 **sbuf**에 저장된 connfd를 return받아 해당 client와의 communication을 한다. **sbuf**에 저장된 connfd가 존재하지 않을 경우 새로운 connfd가 shared buffer에 삽입될 때까지 worker thread는 대기한다.

- communicate_client 함수: Worker Thread가 연결된 client의 요청을 처리하는 함수.

✓ Task1, Task2 공통

- STOCK binary tree: 주식의 정보를 나타내는 binary tree. 각 node는 주식의 ID, left_stock, price와 left, right child node 정보를 가지고 있다. Tree는 ID

를 효율적으로 찾을 수 있도록 binary search tree 구조로 작성했다.

Task1은 single process에서 순차적으로 client의 요청을 처리하기 때문에 별도의 synchronization이 필요하지 않다.

Task2는 multiple Thread가 share하고 있는 **STOCK**의 정보를 read(show) 또는 write(buy, sell)할 때 interleaving이 발생하지 않도록 synchronization 기법을 적용해야 한다. 따라서 **Task2**는 추가로 각 node를 read(show) 중인 Thread의 개수를 나타내는 readcnt 변수, readcnt의 증감을 synchronize하기 위한 semaphore mutex 변수, 그리고 write(buy, sell)를 synchronize하기 위한 semaphore w 변수가 있다.

- store_stock 함수: 서버가 종료될 때 호출되어 **STOCK** tree에 저장되어 있는 정보를 다시 stock.txt에 저장한다.

- sigint_handler 함수: SIGINT signal이 발생할 때 처리하는 함수. 서버를 종료할 때 Ctrl+C로 signal을 보내 종료한다. **store_stock** 함수를 호출하여 메모리에 저장되어 있는 주식 정보를 파일에 저장한다.

- load_stock 함수: 서버가 실행될 때 호출되어 stock.txt에 저장되어 있는 주식 정보를 메모리에 저장한다.

- insert_stock 함수: load_stock 함수에서 파일의 주식 정보를 read할 때마다 호출되어 새로운 주식 정보를 할당한 node를 **STOCK** tree에 삽입한다.

- show_stock 함수: client의 show 요청 시 호출되는 함수. 주식 tree를 inorder traverse하며 모든 주식의 ID, left_stock, price 정보를 client에게 전송한다. Binary search tree 구조이므로 주식 ID의 오름차순으로 client에게 보여준다.

Task2에서는 추가로 synchronization이 필요하다. 각 Thread에서 **show_stock** 함수를 호출하여 node의 주식 정보를 read할 때, 해당 node의 readcnt를 1 증가시킨다. 증가한 readcnt가 1이라면(read하는 Thread가 생김) semaphore w에 대한 lock을 잡아 다른 Thread에서 해당 node에 write(buy, sell)할 수 없도록 한다. Read 작업이 끝나면 readcnt를 1 감소시킨다. 감소한 readcnt가 0이라면(read중인 Thread 없음) semaphore w에 대한 lock을 풀어 다른 Thread에서 write(buy, sell)이 가능하도록 한다.

이는 First readers-writers problem의 solution으로 Thread들이 특정 node를 read(show)중일 때, 다른 Thread가 write(buy, sell)하지 못하도록 하여 shared variable에 대한 interleaving이 발생하지 않도록 한 것이다.

여러 Thread들이 동시에 같은 node의 정보를 read(show)하는 것은 가능하다. 이는 여러 thread들이 동시에 read를 하여도 shared variable의 값이 변하는 것이 아니기 때문에 문제가 발생하지 않는다. 단, readcnt를 증감시킬 때는 mutex에 대한 lock을 잡아 synchronization을 해주어야 한다.

- buy 함수: client의 buy 요청 시 호출되는 함수. **STOCK** binary search tree를 탐색하여 구매하려는 주식 ID를 찾는다. 해당 주식의 left_stock이 주문 수량 이상인 경우 구매 요청을 승인하여 정상 처리하고, left_stock이 주문 수량 미만인 경우 구매 요청을 거절한다.

Task2에서는 구매 요청을 처리할 때 추가적인 synchronization이 필요하다. 주식의 구매 요청을 처리할 때 semaphore w에 대한 lock을 잡은 후 left_stock의 정보를 변경한 다음 다시 lock을 해제한다. 만약 다른 Thread에서 read(show)중이거나 write(buy, sell)중일 경우 w에 대한 lock이 해제될 때까지 대기해야 한다. Writer(buy, sell)은 reader(show)보다 우선순위가 낮기 때문에 여러 Thread들이 show 요청을 처리 중일 경우 모든 show 요청이 종료된 후에 write가 가능하다.

- sell 함수: client의 sell 요청 시 호출되는 함수. **STOCK** tree를 탐색하여 판매하려는 주식 ID를 찾는다. 해당 주식의 판매 요청을 승인하여 정상 처리한다.

TASK2에서는 판매 요청을 처리할 때 **buy** 함수와 같은 synchronization이 필요하다.

3. 구현 결과

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

✓ Task1

새로운 client의 connection 요청이 들어오면 Accept하여 fd pool에 넣어 Select 함수를 통해 monitoring 하도록 했다.

Select 함수를 통해 Multiple client로부터 오는 요청을 감지하면, event가 발생한 모든 client의 요청을 순차적으로 처리하도록 했다.

Event-driven 방식은 fine-grain한 concurrent 서버를 만들기 어렵다. 한 client의 요청을 모두 처리하고 다음 client의 요청을 처리하는 방식은 fine-grain하지 못하다. 따라서 이를 보완하기 위해 각 client의 요청은 하나씩 처리하여 multiple client로부터 들어오는 요청을 concurrent하게 처리했다.

Client로부터 연결이 종료되면 연결된 connfd를 close하고, fd pool에서 제거했다.

```
cse20171683@cspro8:~/SP/AS2/20171683/task_1$ ./stockclient 172.30.10.11 60055
show
1 20 1000
2 56 20000
3 9 1200
4 17 5000
5 15 3700
6 14 2500
7 90 12000
8 10 7000
9 47 8500
10 15 51000
buy 3 10
Not enough left stocks
buy 3 9
[buy] success
show
1 20 1000
2 56 20000
3 0 1200
4 17 5000
5 15 3700
6 14 2500
7 90 12000
8 10 7000
9 47 8500
10 15 51000
exit
disconnection with server
```

✓ Task2

Master Thread로 client의 connection 요청이 들어오면 Accept하여 Shared Buffer에 삽입하여 wait 중인 Worker Thread를 깨우도록 했다.

Worker Thread는 wait하다가 Shared Buffer에 connfd가 삽입되면 이를 제거하고 해당 client와 연결되어 communication을 하도록 했다. Client의 요청은 연결된 Worker Thread가 처리하도록 하여, 여러 client로부터 동시에 요청이 들어와도 연결된 각 Worker Thread에 의해 동시에 처리할 수 있도록 했다.

Thread-based 방식에서 주식 정보를 저장하고 있는 **STOCK** tree는 모든 Thread에 의해 공유되는 shared variable이다. 각 Thread에서 node의 정보를 read 또는 write할 때 semaphore에 대한 lock을 잡도록 하여 thread-safe한 concurrent 주식 서버를 구현했다.

```
cse20171683@cspro8:~/SP/AS2/20171683/task_2$ ./stockclient 172.30.10.11 60055
show
1 46 1000
2 6 20000
3 72 1200
4 22 5000
5 17 3700
6 28 2500
7 11 12000
8 3 7000
9 23 8500
10 34 51000
buy 8 4
Not enough left stocks
buy 8 3
[buy] success
sell 9 5
[sell] success
show
1 46 1000
2 6 20000
3 72 1200
4 22 5000
5 17 3700
6 28 2500
7 11 12000
8 0 7000
9 28 8500
10 34 51000
exit
disconnection with server
```

4. 성능 평가 결과 (Task 3)

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

multiclient.c의 num_client만큼 fork를 통해 자식 프로세스를 생성하고 서버에 요청을 보내는 while문 앞, 뒤에 다음과 같은 code를 추가하여 성능을 측정했다.

```
        host = argv[1];
        port = argv[2];
        num_client = atoi(argv[3]);

        /* Task 3 */
        struct timeval start;    // starting time
        struct timeval end;      // ending time
        unsigned long e_usec;    // elapsed microseconds

        gettimeofday(&start, 0);
        int iter;
        for(iter=0; iter<ITERATIONS; iter++){
        runprocess=0;
        /* Task 3 */


/*      fork for each client process     */
        while(runprocess < num_client){
                //wait(&state);
                pids[runprocess] = fork();
```

(while문 앞)

```
        for(i=0;i<num_client;i++){
                waitpid(pids[i], &status, 0);
        }

        /* Task 3 */
        }
        gettimeofday(&end, 0);
        e_usec = ((end.tv_sec * 1000000) + end.tv_usec)
                        - ((start.tv_sec * 1000000) + start.tv_usec);

        printf("elapsed time: %lu microseconds\n", e_usec/ITERATIONS);
        printf("order_cnt: %d\n", ORDER_PER_CLIENT * num_client);
        /* Task 3 */
```

(while문 뒤)

또한 server의 처리율을 확인하는 것이 목적이기 때문에 Fputs(buf, stdout); 코드와 usleep(1000000); 코드는 주석을 처리하여 elapsed time을 측정했다. 프로그램은 지정한 ITERATIONS 만큼 반복문을 수행하여 평균 elapsed time을 측정했다.

성능 측정을 위한 **multiclient** 프로그램 실행 시 출력 결과 예시는 다음과 같다.

```
cse20171683@cspro8:~/SP/AS2/20171683/task_1$ ./multiclient 172.30.10.11 60055 32
elapsed time: 272258 microseconds
order_cnt: 3200
```

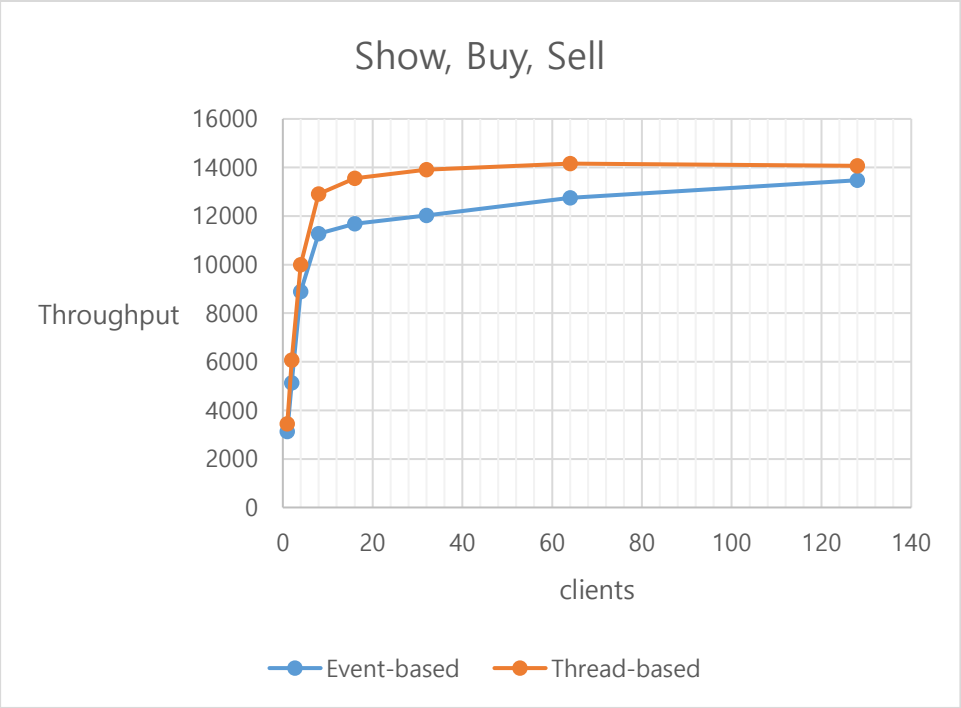
- **모든 명령어 수행(ORDER_PER_CLIENT = 100)**

Task1			
Client_Num	ORDER_PER_CLIENT	Time(ms)	Throughput
1	100	32	3125
2	200	39	5128.205128
4	400	45	8888.888889
8	800	71	11267.60563
16	1600	137	11678.83212
32	3200	266	12030.07519
64	6400	502	12749.00398
128	12800	950	13473.68421

Task2			
Client_Num	ORDER_PER_CLIENT	Time(ms)	Throughput
1	100	29	3448.275862
2	200	33	6060.606061
4	400	40	10000
8	800	62	12903.22581
16	1600	118	13559.32203
32	3200	230	13913.04348
64	6400	452	14159.29204
128	12800	910	14065.93407

client 수에 따른 elapsed time

client 수에 따른 Throughput



분석

Event-based, Thread-based 서버 모두 client의 수가 적을 때에는 client 수의 증가에 따른 Throughput의 증가가 크게 나타났지만, 일정 수준의 동시 접속 client수에 도달할 경우 Throughput의 큰 변화없이 14,000 정도로 수렴하는 것을 확인할 수 있다. 이는 서버가 동시에 처리할 수 있는 요청 개수의 한계치로 보인다.

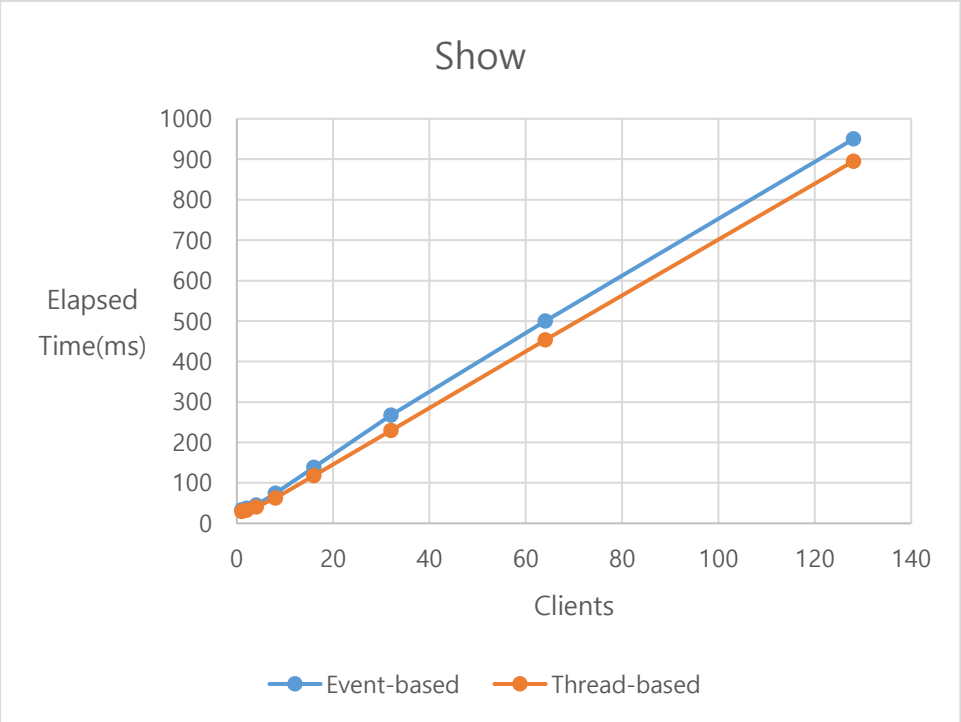
예상했던 대로 Thread-based 서버가 Event-based 서버보다 더 좋은 성능을 보였다. 하지만 큰 차이는 없었는데, Thread-based 서버에서 각 Worker Thread가 요청을 처리할 때 다른 Thread에 의해 interleaving되지 않도록 semaphore에 대한 lock, unlock operation을 통해 synchronization을 한다. 요청을 처리할 때마다 lock operation과 unlock operation을 하는 cost, Thread context switch를 하는 overhead 등의 영향으로 성능의 큰 차이는 나타나지 않은 것으로 보인다.

- **Show 명령만 수행(ORDER_PER_CLIENT = 100)**

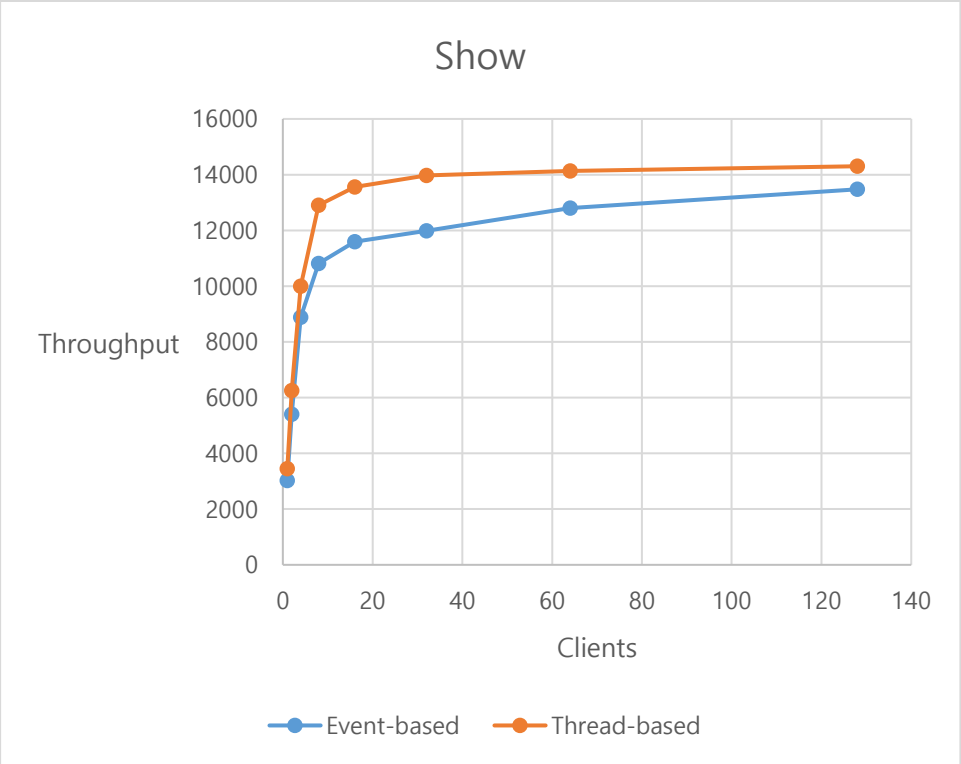
Task1			
Client_Num	ORDER_PER_CLIENT	Time(ms)	Throughput
1	100	33	3030.30303
2	200	37	5405.405405
4	400	45	8888.888889
8	800	74	10810.81081
16	1600	138	11594.2029
32	3200	267	11985.01873
64	6400	500	12800
128	12800	950	13473.68421

Task2			
Client_Num	ORDER_PER_CLIENT	Time(ms)	Throughput
1	100	29	3448.275862
2	200	32	6250
4	400	40	10000
8	800	62	12903.22581
16	1600	118	13559.32203
32	3200	229	13973.79913
64	6400	453	14128.03532
128	12800	895	14301.67598

client 수에 따른 elapsed time



client 수에 따른 Throughput



분석

Event-based 서버와 Thread-based 서버 모두 모든 명령어를 처리할 때와 Show 명령어만 처리할 때의 Performance가 거의 똑같이 나왔다.

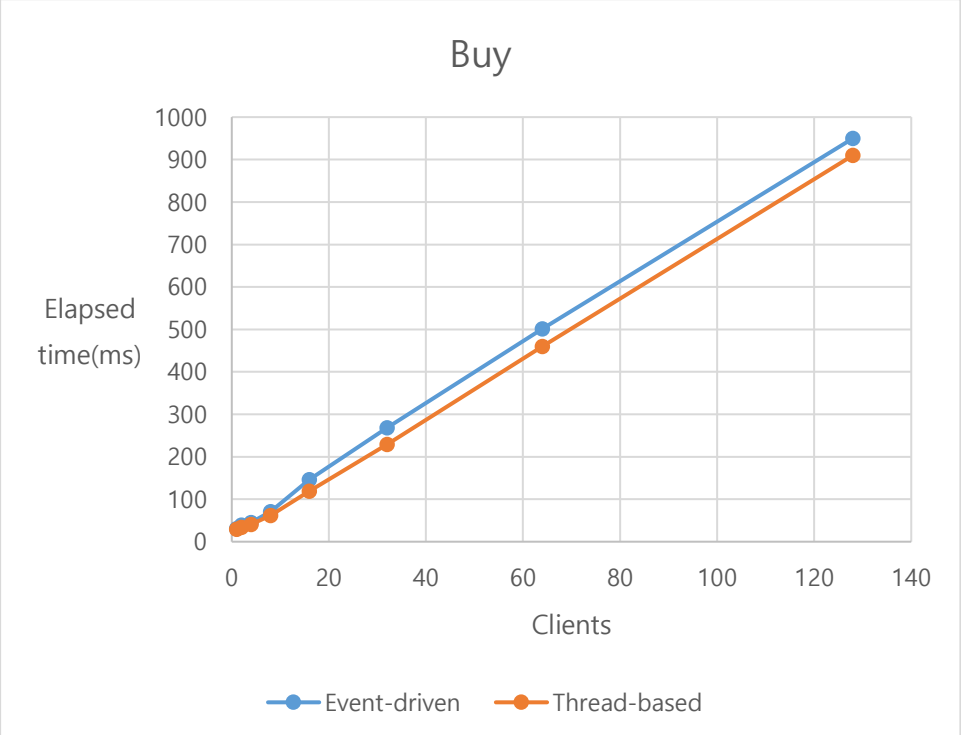
Thread-based 서버는 모든 명령어를 처리할 때보다 Show 명령어만 처리할 때의 Performance가 더 좋게 나올 것으로 예측했지만 큰 차이가 나타나지 않았다. Show 명령어만 처리하는 경우 모든 Thread가 **STOCK** tree의 정보를 read만 하기 때문에 대기없이 동시에 read가 가능하여 더 좋은 성능을 보일 것으로 예측했으나 이 경우에도 각 노드의 readcnt 증감을 위한 lock과 unlock을 수행하기 때문에 큰 차이는 나타나지 않았다.

- **Buy(Sell) 명령만 수행(ORDER_PER_CLIENT = 100)**

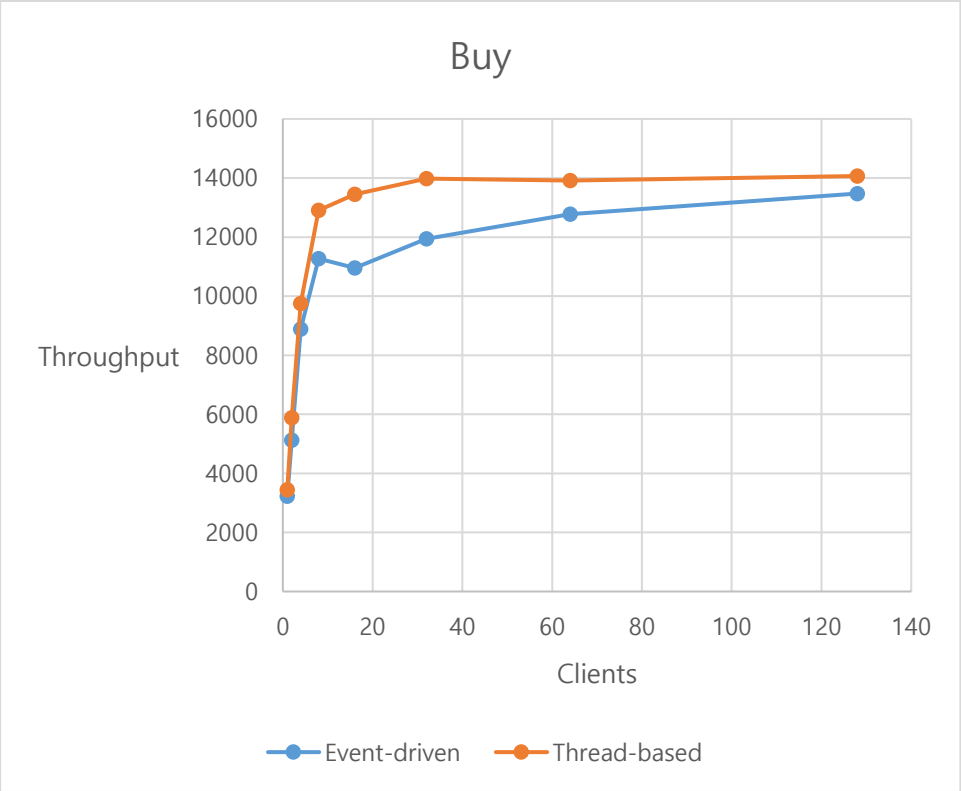
Task1			
Client_Num	ORDER_PER_CLIENT	Time(microsec)	Throughput
1	100	31	3225.806452
2	200	39	5128.205128
4	400	45	8888.888889
8	800	71	11267.60563
16	1600	146	10958.90411
32	3200	268	11940.29851
64	6400	501	12774.4511
128	12800	950	13473.68421

Task2			
Client_Num	ORDER_PER_CLIENT	Time(microsec)	Throughput
1	100	29	3448.275862
2	200	34	5882.352941
4	400	41	9756.097561
8	800	62	12903.22581
16	1600	119	13445.37815
32	3200	229	13973.79913
64	6400	460	13913.04348
128	12800	910	14065.93407

client 수에 따른 elapsed time



client 수에 따른 Throughput



분석

Buy 명령어만 처리하는 경우에도 모든 명령어를 처리할 때와 Show 명령어만 처리할 때와 비슷한 결과가 나왔다.