System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 전용본

학번 : 20181683

1. 개발 목표

동시 주식 서버(Concurrent Stock Server)를 설계하고 구현하는 것이 이 프로젝트의 목적이다. 주식 서버는 주식 정보를 저장하고 있고 여러 client들과 통신하여 요청들을 수행한다. 주식 서버를 Iterative하게 구현하면 서버에 하나의 client 밖에 접근하지 못하고 연결이 종료되기 전까지 다른 client들은 서버와 통신할 수 없기에 주식 서버를 Concurrent하게 구현해야 한다. Concurrent하게 주식 서버를 구현하면 동시에 여러 client들의 요청을 받아들이고 처리할 수 있다.

동시 주식 서버를 1) event-driven approach, 2) thread-based approach 2가지 방식으로 설계, 구현하고 성능 평가 및 분석할 것이다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   select함수를 이용해 주식 서버를 concurrent하게 구현할 수 있다. select함수는 여러 client이 주식 서버에 connection을 요청할 수 있고 buy, sell, show의 request를 server가 수행할 수 있도록 한다. server는 select함수를 통해 request가 들어온 file descriptors를 알 수 있고 이를 수행함으로써 여러 client와의 connection을 concurrent하게 다룰 수 있다.

2. Task 2: Thread-based Approach

   pthread library를 이용해 주식 서버를 concurrent하게 구현할 수 있다. main thread는 accept함수를 이용해 client의 연결 요청을 받아들이고 subf 구조체를 이용해 client와의 connection file dscriptors를 buf에 저장하고 이를 이용해 worker thread들이 각각의 connection을 할당받아 client들의 request를 concurrent하게 수행할 수 있다. 또 자료구조를 공유하고 있는 문제로 동시에 request를 수행할 때 데이터가 잘못 수정되는 경우를 synchronization을 이용해 방지한다.

3. Task 3: Performance Evaluation

   gettimeofday함수를 이용해 task1, 2에서 구현한 동시 주식 서버의 요청 수

행 소요 시간을 측정한다. 총 요청 수 / 소요 시간을 동시처리율이라는 개념으로 정의하고 동시처리율이 높을 수록 client의 요청을 더 concurrent하게 처리할 수 있을 것이다. client의 수의 변화에 따라 달라지는 소요 시간과 동시처리율을 기록하고 분석할 것이다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  select함수와 FD_SET 구조체를 이용해 file descriptors가 I/O할 준비가 되었는지(FD_SET에 변동사항이 있는지)를 간편하게 확인할 수 있다. 하나 이상의 client가 준비가 될 때까지 block하고 준비가 되면 unblock한 후에 accept함수를 통해 준비가 된 client와의 connection을 pool 구조체의 clientfd 배열에 추가해서 관리한다. check_client함수에서 pool 구조체의 clientfd 배열을 반복문으로 순회하며 FD_ISSET을 통해 통신 여부를 확인하고 multi-client들의 요청들을 받아들이고 수행한다.

  ✓ epoll과의 차이점 서술

  epoll은 select와 다르게 전체 file descriptors에 대한 반복문을 사용하지 않고, 커널에게 정보를 요청하는 함수를 호출할 때마다 전체 file descriptors에 대한 정보를 넘기지 않는다. 운영체제가 직접 file descriptors들의 정보를 관리한다. 전체 file descriptors들을 순회하면서 FD_ISSET을 하는 문제가 발생하지 않는다.

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

  ✓ Master Thread의 Connection 관리

  pthread library의 함수들을 이용해서 thread 기반의 주식 서버를 구현한다. Master Thread가 multi-client들의 connection 요청들을 받아 sbuf 구조체의 buf 배열에 connection file descriptors을 저장한다. Worker Thread들을 만들어 지정한 함수를 수행하게 할 것이다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker Thread들은 pthread_create함수를 이용해 원하는 수 만큼 만들어지고 정해놓은 thread함수를 수행한다. Worker Thread들은 Master Thread가 관리하는 subf의 구조체에 buf 배열에 접근한다. synchoronization을 이용해 배열에서 connection file descriptor을 pop해서 하나의 Worker Thread가 중복되지 않는 하나의 client와의 connection을 할당 받고 client의 request를 수행한다. 수행 과정에서 Worker Thread들이 공유하고 있는 자료구조(stock_tree, stock.txt)를 접근, 수정할 때에도 synchronziation을 이용해 문제가 없게 한다.

- **Task3 (Performance Evaluation)**

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

  동시처리율을 # of request / 소요 시간 으로 정의하고 측정할 것이다. 동시처리율이 크다는 것은 많은 양의 client request를 짧은 시간 안에 처리할 수 있다는 것이므로 서버의 concurrency를 측정하기에 좋은 도구이다.

  multiclient.c에서 gettimeofday함수를 이용해 측정 시작 시간, 측정 종료 시간을 정하고 이를 통해 소요 시간을 알아낼 것이다. multiclient의 configuration들을 조정 후에 client의 수를 증가시키며 소요 시간을 측정하고 이를 통해 동시처리율을 알아낼 것이다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  thread-based approach 방식의 동시 주식 서버는 client request가 증가할 때 concurrent하게 처리할 수 있는 request양도 증가할 것이므로 동시처리율 또한 증가할 것이라고 예측할 수 있다. multi-core의 장점을 살려 점점 동시처리율이 증가할 것이다. CPU core의 개수는 한정되어 있으므로 client수가 증가할 때 동시처리율이 낮아지는 구간도 있을 것이라 예상할 수 있다.

  event-driven approach 방식의 동시 주식 서버는 client request가 증가할 때 concurrent하게 처리할 수 있는 request양도 증가할 것이므로 동시처리율 또한 증가할 것이라고 예측할 수 있다. 하지만 multi-core의 장점을 살리지 못하므로 thread-based approach보다 동시처리율이 점점 낮아질 것이라고 예상할 수 있다.

C. 개발 방법

< csapp.h >

1) item(stock) 구조체 및 전역변수

```
/* stock table */
typedef struct item{
    int id; /* stock id */
    int left_stock; /* num of left stock */
    int price; /* stock price */
    struct item *left, *right;
}item;

item *stock_tree;
int num_item;
```

stock.txt에 담긴 주식 내용들을 binary search tree로 구현하기 위해 item 구조체를 선언하였다. item에는 주식의 ID, 잔여 주식량, 주식 가격과 자식 노드를 가르키는 pointer가 있다. item 타입의 stock_tree pointer를 선언하여 트리의 root을 가르킨다. num_item은 stock_tree에 들어있는 총 주식의 수를 저장한다.

2) pool 구조체

```
typedef struct{
    int maxfd;                  /* Largest descriptor in read_set */
    fd_set read_set;            /* Set of all active descriptors */
    fd_set ready_set;           /* Subset of descriptors ready for reading */
    int nready;                 /* Number of ready descriptors from select */
    int maxi;                   /* High water index into client array */
    int clientfd[FD_SETSIZE];       /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE];    /* Set of active read buffers */
}pool;
```

3) sbuf 구조체 및 전역변수

```
sem_t mutex;    /* readcnt seamphore */
sem_t w;        /* stock_table semaphore */
int readcnt; /* num of current readers */

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

#define SBUFSIZE 8192
#define NTHREADS 1024
sbuf_t sbuf;
```

< stockserver.c – 공통 >

1) void read_table()

stockserver 실행 시에 메모리(stock.txt)에서 데이터를 가져와 트리 자료구조(stock_tree)를 구성하는 함수이다. stock_tree의 root을 만들어 준 후에 stock.txt를 한 줄씩 읽으면서 tokenize한 후에 stock_tree에 삽입한다.

2) void write_table()

메모리(stock.txt)를 현재 주식 정보로 최신화하는 함수이다. stock_tree를 중위 순회하면서 주식 정보를 한 줄씩 메모리(stock.txt)에 저장한다

< stockserver.c – event-driven approach >

1) void init_pool(int listenfd, pool *p)

이 함수는 pool 구조체를 초기화한다. maxi를 -1로, clientfd 배열의 모든 값을 -1로 초기화해 비어있게 한다. FD_ZERO를 이용해 read_set의 값들을 0으로 초기화하고 초기 read_set에는 listenfd만 존재하므로 FD_SET을 이용해 read_set에 listenfd를 추가한다.

2) void add_client(int connfd, pool *p)

새로운 client와의 connection을 pool에 추가하는 함수이다. pool의 clientfd 배열을 순회하면서 비어있는 clientfd를 찾는다. 빈 clientfd[i]를 찾은 후 connfd 값을 저장하고 clientrio[i] 구조체도 초기화한다. FD_SES을 이용해 connfd를 read_set에 추가하고 만약 connfd 값이 maxfd보다 크다면 maxfd를 connfd로 바꿔주고, connfd의 index 값이 maxi보다 크다면 maxi를 connfd의 index 값으로 바꿔준다.

3) void check_client(pool *p)

client들과의 connection을 바탕으로 client들의 request들을 처리하는 함수이다. ready for reading 상태인 client들을 찾고 Rio_readlineb함수를 이용해 request를 buf에 담는다. request가 exit일 경우 해당 client와 connection을 종료하고 주식 자료구조(stock_tree)를 메모리(stock.txt)에 적재한다. request가 exit이 아닌 경우 stocktrade함수를 이용해 show, buy, sell 명령을 수행한다.

<stockserver.c – thread-based approach >

1) void sbuf_init(sbuf_t *sp, int n)

sbuf 구조체를 초기화하는 함수이다. 인자로 받은 sbuf 구조체에 buf에 n개의 slots을 생성하고 초기에 비어있으므로 front,rear은 0으로 초기화하고 buf, slots, items에 대한 각각의 semaphore들의 초기값을 할당한다.

2) void sbuf_deinit(sbuf_t *sp)

sbuf 구조체의 buf를 free하는 함수이다.

3) void sbuf_insert(sbuf_t *sp, int item)

sbuf 구조체의 buf에 인자로 받은 item을 삽입하는 함수이다. slots semaphore을 이용해 slot이 꽉 찬 경우에는 자리가 날 때까지 기다린 후에 slot의 값을 감소시키며 buf에 접근한다. buf에 접근해서 삽입할 때에도 semaphore을 이용해 buf에 대한 다른 작업이 동시에 발생하지 않게 한다. 마지막으로 item의 값을 증가시킨다.

4) int sbuf_remove(sbuf_t *sp)

sbuf 구조체의 buf에서 item 하나를 pop해서 반환하는 함수이다. item semaphore을 이용해 item이 없는 경우에는 생길 때까지 기다린 후에 item의 값을 감소시키며 buf에 접근한다. buf에 접근해서 pop할 때에도 semaphore을 이용해 buf에 대한 다른 작업이 동시에 발생하지 않게 한다. 마지막으로 slots의 값을 증가시킨다.

5) void* thread(void *vargp)

Worker Thread들이 수행할 동작을 담은 함수이다. 먼저 pthread_detach함수를 이용해 thread 종료시에 자동으로 reaping이 되게 한다. sbuf_remove함수를 이용해 client와 connection을 할당 받은 후 stocktrade함수를 통해 show, buy, sell 명령을 수행한다. 수행 후에 메모리(stock.txt)를 최신화한다.

< stocktrade.c >

1) void show(item *cur, char msg[])

현재 주식 정보를 client에게 보여주는 역할을 하는 함수이다. stock_tree(binary search tree)를 중위 순회하면서 주식 정보(ID, 잔여량, 가격)을 msg 배열에 담아 client에게 보낸다.

2) void sell(int stock_id, int num)

client의 요청에 따라 주식을 매도하는 함수이다. stock_tree에서 팔고자 하는 주식을 찾은 후에 잔여량을 증가시키고 매도 성공 메세지를 client에게 보낸다 .

3) int buy(int stock_id, int num)

client의 요청에 다라 주식을 매수하는 함수이다. stock_tree에서 사고자 하는 주

식을 찾은 후에 잔여량을 감소시키고 매수 성공 메세지를 client에게 보낸다. 이 때 사고자 하는 양보다 잔여량이 적으면 매수가 실패했음을 client에게 알린다.

4) void stocktrade(char *buf, int connfd)

client의 request를 직접적으로 수행하는 함수이다. client에게서 받은 buf를 tokenize하고 적절한 함수를 실행시키며 명령을 수행한다.

< binarysearchtree.c >

주식 정보를 관리하기 위한 자료구조 (binary search tree)를 구성하기 위한 함수들이 있다. 트리의 생성, 삽입, 삭제 등에 관한 함수들이 작성되어 있다.

3. **구현 결과**

stockserver

```
cse20181683@cspro8:~/Project2/task_1$ ./stockserver 60045
Connected to (172.30.10.11, 44768)
Server received 5 bytes on fd 4
Server received 8 bytes on fd 4
Server received 10 bytes on fd 4
Server received 5 bytes on fd 4
Server received 5 bytes on fd 4
```

stockclient

```
cse20181683@cspro:~/Project2/task_1$ ./stockclient 172.30.10.8 60045
show
1 40 10000
2 20 20000
3 70 30000
buy 1 5
[buy] success
sell 2 10
[sell] success
show
1 35 10000
2 30 20000
3 70 30000
exit
```

stock.txt (거래 전)

```
cse20181683@cspro:~/Project2/task_1$ cat stock.txt
1 40 10000
2 20 20000
3 70 30000
```

stock.txt (거래 후)

```
cse20181683@cspro:~/Project2/task_1$ cat stock.txt
1 35 10000
2 30 20000
3 70 30000
```

stockserver

```
cse20181683@cspro8:~/Project2/task_1$ ./stockserver 60045
Connected to (172.30.10.11, 46896)
Connected to (172.30.10.11, 46898)
Server received 9 bytes on fd 4
Connected to (172.30.10.11, 46900)
Server received 5 bytes on fd 5
Server received 5 bytes on fd 6
Server received 5 bytes on fd 4
Server received 9 bytes on fd 5
Server received 9 bytes on fd 6
Server received 8 bytes on fd 4
Server received 9 bytes on fd 5
Server received 9 bytes on fd 6
```

multiclient

```
cse20181683@cspro:~/Project2/task_1$ ./multiclient 172.30.10.8 60045 3
child 173841
child 173840
child 173839
[sell] success
1 33 10000
2 22 20000
3 68 30000
1 33 10000
2 22 20000
3 68 30000
1 33 10000
2 22 20000
3 68 30000
[sell] success
[sell] success
[buy] success
[sell] success
[sell] success
```

stock.txt (거래 전)

```
cse20181683@cspro:~/Project2/task_1$ cat stock.txt
1 33 10000
2 18 20000
3 68 30000
```

stock.txt (거래 후)

```
cse20181683@cspro:~/Project2/task_1$ cat stock.txt
1 39 10000
2 24 20000
3 69 30000
```

4. 성능 평가 결과 (Task 3)

multiclient configuration

```
#define MAX_CLIENT 100
#define ORDER_PER_CLIENT 10
#define STOCK_NUM 5
#define BUY_SELL_MAX 5
```

1) Event-driven approach, Thread-based approach 성능 비교

동시 처리율 = # of request / 소요 시간

그래프 X축 = # of client

그래프 Y축 = 동시 처리율

A. buy, sell, show test

< event-driven test result >

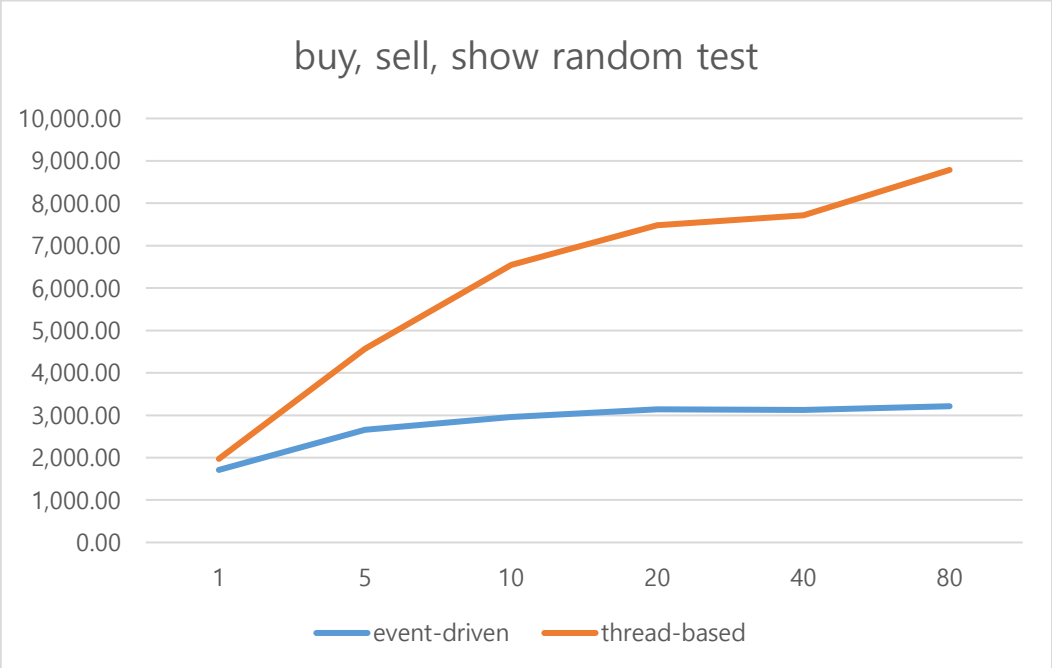
```
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 1
time : 5846 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 5
time : 18792 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 10
time : 33838 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 20
time : 63726 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 40
time : 128071 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 80
time : 248976 microseconds
```

< thread-based test result >

```
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 1
time : 5073 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 5
time : 10954 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 10
time : 15276 microseconds
```

```
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 20
time : 26726 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 40
time : 51865 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 80
time : 91059 microseconds
```

buy, sell, show random test					
# of client	# of request	event-driven		thread-based	
		시간	동시 처리율	시간	동시 처리율
1	10	0.005846	1,710.5713	0.005073	1971.2201
5	50	0.018792	2,660.7066	0.010954	4,564.5426
10	100	0.033838	2,955.2574	0.015276	6,546.2162
20	200	0.063726	3,138.4364	0.026726	7,483.3495
40	400	0.128071	3,123.2675	0.051865	7,712.3300
80	800	0.248976	3,213.1611	0.091059	8,785.5126



위의 그래프는 client가 buy, sell, show를 랜덤하게 섞어서 요청하는 경우를 실험결과이다. 전체적으로 thread-based 방식이 event-driven 방식보다 동시처리율이 높게 나왔는데 이는 thread-based 방식이 multi-core의 장점을 살리기 때문일 것이고 예측과 유사하다. 하지만 show, buy, sell을 랜덤하게 뽑았기에 결과를 무조건적으로 신뢰하기 어렵다.

B. buy, sell test

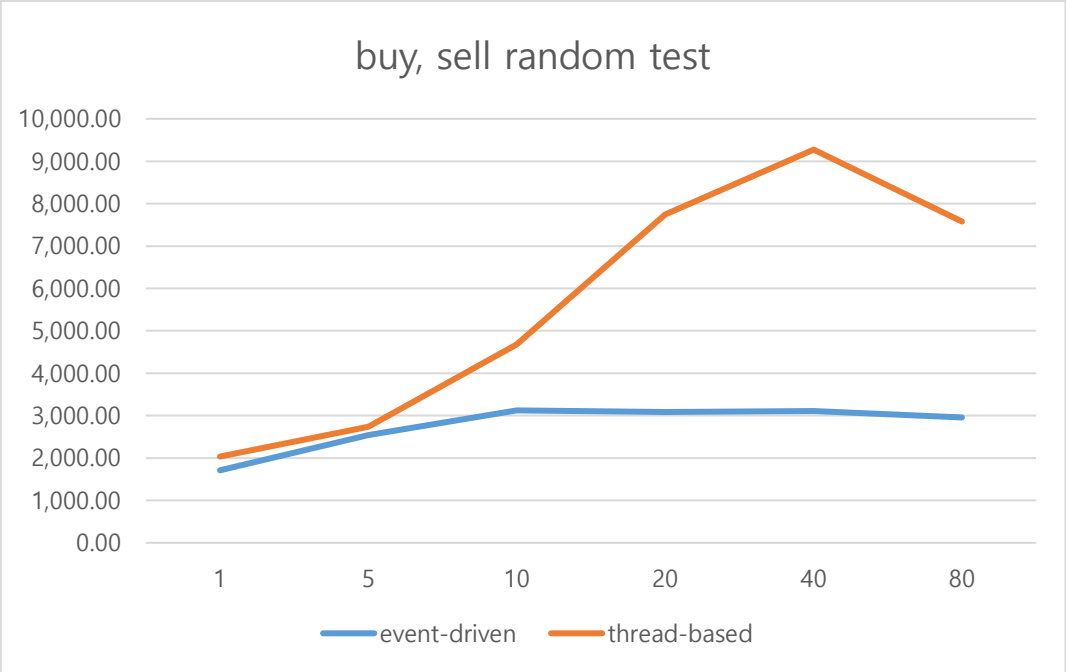
< event-driven test result >

```
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 1
time : 5811 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 5
time : 19696 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 10
time : 32019 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 20
time : 64884 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 40
time : 128834 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 80
time : 248582 microseconds
```

< thread-based test result >

```
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 1
time : 4914 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 5
time : 10681 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 10
time : 16495 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 20
time : 25821 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 40
time : 43135 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 80
time : 105607 microseconds
```

buy, sell random test					
# of client	# of request	event-driven		thread-based	
		시간	동시 처리율	시간	동시 처리율
1	10	0.005811	1,710.8742	0.004914	2,035.0020
5	50	0.019696	2,538.5865	0.010681	2,738.5255
10	100	0.032019	3,123.1456	0.016495	4,681.2096
20	200	0.064884	3,082.4240	0.025821	7,745.6333
40	400	0.128834	3,104.7704	0.043135	9,273.2120
80	800	0.248582	2,959.5255	0.105607	7,575.2554



위의 그래프는 client가 buy, sell만 랜덤하게 요청하는 경우를 실험한 결과이다. buy, sell은 write 행위인데 thread-based 방식에선 한 명의 writer만이 공유 자료구조에 접근 가능하므로 synchronization overhead가 많이 발생해서 thread-based 방식보다 event-driven 방식이 더 시간이 낮게(동시처리율이 높게) 나올 것으로 예상했는데 예상과 다른 결과가 나왔다. 실험 환경의 문제이거나 synchronization overhead를 감안하더라도 thread-based 방식이 multi-core의 장점을 살린 것이 이런 결과로 나타난 것 같다.

C. show test

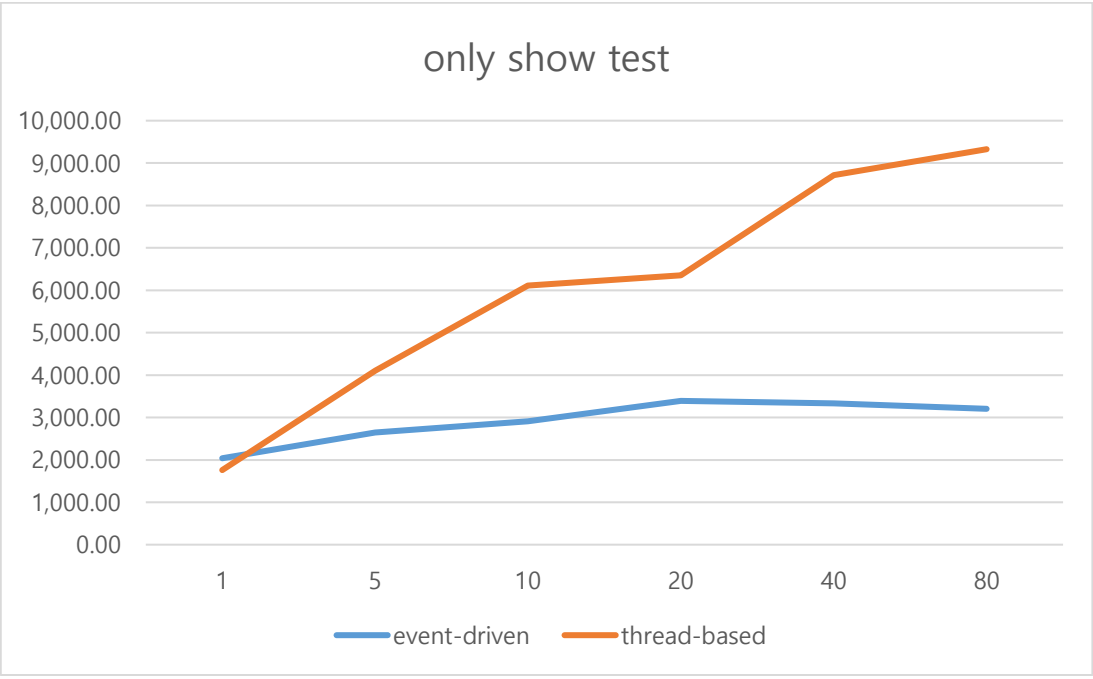
< event-driven test result >

```
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 1
time : 4905 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 5
time : 18889 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 10
time : 34349 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 20
time : 58977 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 40
time : 120144 microseconds
cse20181683@cspro:~/Project2/task_3-1$ ./multiclient 172.30.10.8 60045 80
time : 249633 microseconds
```

< thread-based test result >

```
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 1
time : 5685 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 5
time : 12184 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 10
time : 16371 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 20
time : 31471 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 40
time : 45909 microseconds
cse20181683@cspro:~/Project2/task_3-2$ ./multiclient 172.30.10.8 60045 80
time : 85772 microseconds
```

only show test					
# of client	# of request	event-driven		thread-based	
		시간	동시 처리율	시간	동시 처리율
1	10	0.004905	2,038.7359	0.005685	1,759.0149
5	50	0.018889	2,647.0432	0.012184	4,103.7426
10	100	0.034349	2,911.2929	0.016371	6,108.3623
20	200	0.058977	3,391.1524	0.031471	6,355.0570
40	400	0.120144	3,329.3381	0.045909	8,712.8885
80	800	0.249633	3,204.7045	0.085772	9,327.0531



위의 그래프는 client가 show만을 요청하는 경우를 실험한 결과이다. thread-based 방식이 event-driven 방식보다 시간이 낮게(동시처리율이 높게) 나온 것을 확인할 수 있는데 show는 read 행위인데 thread-based 방식에선 동시에 여러 명의 reader가 공유 자료구조에 접근할 수 있으므로 synchronization overhead가 buy, sell을 요청하는 경우와 다르게 없고 multi-core의 장점을 생각했을 때 예상한 결과가 나왔다.

2) 분석 결과

네트워크 상태, 서버 상태의 차이가 있을 수 있어 완벽한 실험이지는 못했으나 thread-based 방식의 동시 주식 서버가 event-baed 방식의 동시 주식 서버보다 속도, 처리 효율면에서 낫다는 점을 알 수 있었다. 위에서 보인 3개의 그래프에서는 대체적으로 thread-based 방식의 동시처리율이 점점 증가하고 있는데 client수가 더 많아지면 동시처리율이 계속 감소하는 구간이 나타날 것이고 과도한 락(semaphore)으로 인해 병목 현상이 벌어질 수도 있으므로 조심해야 한다.