Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 송경호

학번 : 20191599

1. 개발 목표

여러 고객이 접속하여 동시에 작업을 수행할 수 있는 동시성을 가진 주식 서버를 구현한다.

고객의 경우 서버에 주식의 조회, 구매, 판매를 요청하며, 서버의 경우 이 요청들을 받아 처리한다. 이때, 다수의 고객이 동시적으로 서버에 요청을 보낼 수 있으므로 이를 처리하기 위해 **1. Event-Driven Approach**와 **2. Thread-Based Approach**, 두 가지 방식을 통해 주식 서버를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

Event-Driven Approach와 Thread-Based Approach 모두 서버의 시작과 함께 stock.txt를 읽고 이진 트리의 형식으로 이를 저장한다. 또한 서버 종료 시 변경된 정보를 다시 stock.txt에 덮어쓴다.

1. Task 1: Event-Driven Approach

Thread-Based Approach와 달리, 하나의 thread 내에서 모든 고객들의 요청을 처리한다.

select 함수를 통해 다수의 file descriptor를 관리하고, 요청 발생 시 해당 요청을 수행 후 다시 select 함수를 통해 다음 요청을 기다리는 방식으로 동작한다.

때문에 다수의 요청이 동시적으로 들어와도 이를 절차적으로 수행해 나가며 서버를 concurrent하게 동작 시킨다.

2. Task 2: Thread-Based Approach

각 고객의 요청을 각기 다른 thread에서 처리한다.

Overhead를 최소화 하기위해 미리 다수의 thread를 생성한 상태에서 고객의 연결 요청이 들어오면 사용 중이지 않은 thread를 해당 고객과 연결하는 방

식으로 동작한다.

Event-Driven Approach와 달리 다수의 thread에서 작업이 동시적으로 수행되기 multicore의 사용으로부터 이득을 얻을 수 있으며 또한 semaphore을 통해 thread 간 공유하는 데이터를 관리해야 한다.

3. Task 3: Performance Evaluation

multiclient.c를 통해 **확장성**과 **워크로드에 따른 분석**, 두 가지를 수행한다. 이때, 원활한 측정을 위해 usleep를 주석 처리하거나 워크로드의 조건에 따라 특정 코드를 추가하는 등, multiclient.c 파일의 수정이 요구된다.

추가적으로, 각 경우 당 3번의 실험의 평균으로 수행능력을 측정한다.

B. 개발 내용

Task1 (Event-Driven Approach with select())

- ✓ **Multi-client 요청에 따른 I/O Multiplexing 설명**

고객과 정보를 주고받을 file descriptor들이 저장된 fd_set과 select 함수를 통해 I/O Multiplexing을 구현한다.

위에서 언급했듯이, select 함수를 통해 다수의 file descriptor를 관리한다. select 함수의 경우 인자로 받은 fd_set 내의 이벤트가 발생한 file descriptor의 fd_set을 반환한다.

반환 받은 fd_set 내의 이벤트가 발생한 file descriptor가 listenfd일 경우, 새로운 고객의 연결 요청이 들어온 것이므로 이를 fd_set에 저장한다.

만약, 이벤트가 발생한 file descriptor가 listenfd가 아닐 경우, 기존 연결된 고객의 요청이 들어온 것이므로, 요청에 맞게 이를 처리해준다.

다수의 요청이 발생했을 경우, 이벤트가 발생한 file descriptor들을 순서대로 순회하면서 요청을 처리해주며 서버를 concurrent하게 동작 시킨다.

✓ **epoll과의 차이점 서술**

epoll은 select의 단점을 보완한 함수로 보다 효율적인 I/O Multiplexing을 구현할 수 있도록 도와준다.

epoll의 경우 이벤트가 발생한 소켓만을 처리하기 때문에 select보다 향상된 성능을 보장하고 소켓 수에 관계없이 일정한 성능을 유지한다. 또한 이벤트가 발생할 때 까지 블로킹 되지 않아 보다 효율적으로 프로그램을 구현할 수 있게 도와준다.

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

✓ **Master Thread의 Connection 관리**

앞서 말했듯, overhead를 줄이기 위해 thread를 미리 생성한 후 새로운 고객의 요청이 발생할 때마다 thread를 연결시킨다. 자세한 과정은 다음과 같다.

먼저 아직 thread에서 작업중이지 않은 file descriptor를 저장할 배열인 sbuf 배열에 새로운 고객에 대해 생성된 file descriptor를 계속해서 저장한다. 이와 동시에 대기 중인 thread에서는 sbuf내에 thread를 배정받지 않은 file descriptor가 존재하는지 확인한다. 만약 그러한 file descriptor가 존재할 경우 이를 추출하여 해당 file descriptor에 대한 작업을 수행하게 된다. 결과적으로 하나의 thread가 특정 client의 요청에 대해 독립적으로 작업을 수행하게 된다.

그러나 이 경우, 예상치 못한 context switching이나 복수의 thread에서의 데이터 접근으로 인해 여러 thread가 동일한 file descriptor를 추출해가는 문제가 발생할 수 있다.

✓ **Worker Thread Pool 관리하는 부분에 대해 서술**

위에서 언급된 문제를 해결하기 위해 semaphore를 이용한다.

sbuf 내에 존재하는 file descriptor의 개수를 나타내는 items, 그리고 sbuf에 file descriptor를 저장할 수 있는 공간의 개수를 나타내는 slots라는 semaphore 변수를 통해, sbuf의 공간을 관리한다. 또한 mutex라는

semaphore 변수를 통해 두 thread가 sbuf에 동시에 접근하여 하나의 file descriptor를 공유하는 race를 방지한다.

file descriptor 외에도 여러 thread가 동시에 접근하여 그 값을 조회, 수정하는 각 각의 주식 노드 역시 semaphore 변수를 통한 관리가 필요하다. 이때, 무조건적으로 두 thread의 동시 접근을 막아야 하는 위 문제와 달리, 노드를 조회하는 show 명령어의 경우 다수의 thread가 동시에 접근이 가능하지만 노드를 수정하는 buy, sell 명령어의 경우 오직 하나의 thread의 접근만이 가능하므로 두 경우를 분리해야 한다. 이를 위해, 각 노드에 대해 그 노드에 접근하고 있는 thread의 수를 나타내는 readcnt 변수와 readcnt의 관리를 위한 mutex semaphore 변수, 마지막으로 현재 노드 접근이 조회인지 수정인지를 나타내는 w semaphore 변수를 통해 이를 관리한다.

- **Task3 (Performance Evaluation)**

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

확장성의 비교는 client의 수에 따른 Event-Based Approach와 Thread-Based Approach의 동시처리율 비교한다. 이때 client의 수가 1, 10, 50, 100인 경우에 대해 실험을 진행한다.

워크로드에 따른 분석은 client 측에서 show, buy, sell을 모두 요청하는 경우, buy와 sell만을 요청하는 경우, show만을 요청하는 경우 세 가지에 대한 실험을 진행한다. 이 경우에도 client의 수가 1, 10, 50, 100인 경우에 대해 실험을 진행한다.

각 실험의 결과는 3번의 실행의 평균값을 이용했으며, 시간은 gettimeofday 함수를 사용하여 second 단위로 측정하였다.

✓ **Configuration 변화에 따른 예상 결과 서술**

Event-Based Approach 와 Thread-Based Approach 모두 오버헤드가 존재하기 때문에 분석 관점에 상관없이 client 가 증가함에 따라 동시 처리율이 증가할 것으로 예상된다. 이때, Thread-Based Approach 는 다수의 thread 에서

작업을 처리하는 반면, Event-Based Approach 는 하나의 thread 에서 모든 작업을 절차적으로 처리하여 멀티코어 사용의 이점을 얻을 수 있어서, client 의 수가 커질수록 Thread-Based Approach 의 동시처리율이 더 좋을 것으로 예상한다.

워크로드 분석의 결과로는 client 측에서 buy 와 sell 만을 요청하는 경우의 동시처리율이 가장 좋을 것으로 예상한다. 기본적으로 show 명령의 경우 그 비용이 높고, 특히 Thread-Based Approach 에서 buy 와 sell 명령어와 함께 show 명령어를 사용하게 되면 semaphore 변수로 인해 중간에 show 가 블락되는 순간이 발생해 더 오랜 실행시간이 발생하기 때문이다.

이러한 이유로 모든 명령어를 요청 받는 경우와 show 명령어만을 요청 받는 경우의 비교는 show 명령어 그 자체의 비용과 buy, sell 명령어와 show 를 함께 쓰는 경우 발생하는 지연의 차이가 변수로 작용할 것으로 예상된다.

C. 개발 방법

①-ⓐ Stock 구조체

```
typedef struct Stock
{
    int ID;
    int left_stock;
    int price;

    struct Stock *left;
    struct Stock *right;

    int readcnt;
    sem_t mutex;
    sem_t w;
} Stock;
```

주식 정보를 노드 단위로 관리하기 위한 구조체이다. 주식의 ID, 남은 주식 수,

주식의 가격을 담는다. 전체 주식의 이진 트리 저장을 위해 left, right 포인터 변수가 존재한다.

Thread-Based Approach 구현 시 thread의 독립적인 접근을 위한 semaphore 변수와 readcnt 변수도 존재한다.

①-ⓑ Stock 관련 함수

```
void showStock(int connfd);
```

고객으로부터 "show" 명령을 받았을 때 실행하는 함수이다. 이진 트리를 순회하며 주식의 id, 남은 상품 수, 가격을 담은 문자열을 생성한 뒤 다시 고객에게 전송한다.

Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 특정 노드에 접근을 시작하면 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 증가시킨다. 만약 첫번째 접근이었다면 w semaphore 변수를 0으로 감소시킨다. 이때 만약 다른 thread가 해당 노드에 대해 수정을 수행 중이었다면 해당 과정에서 수정이 완료될 때까지 기다리게 된다. 이후 해당 노드에 대한 정보를 문자열에 담은 뒤 해당 노드에 대한 조회가 완료됐으므로 다시 mutex를 이용해 다른 접근을 막은 뒤 readcnt를 감소시킨다. 역시 마찬가지로 해당 thread가 마지막 접근이었다면 w semaphore 변수를 1로 증가시켜 다시 다른 thread의 수정 접근이 가능하게끔 만들어준다.

```
void buyStock(int connfd, int targetID, int num);
void sellStock(int connfd, int targetID, int num);
```

고객으로부터 "buy" 혹은 "sell" 명령을 받았을 때 실행하는 함수이다. 이진 트리를 순회하며 사거나 팔 노드를 검색하고 주어진 수 만큼 해당 주식을 증가시키거나 감소시킨다. 성공하거나 실패할 수 있으므로 그에 해당하는 메시지를 다시 고객에게 전송한다.

Thread-Based Approach의 경우 노드 단위로 발생할 수 있는 Reader-Writer Problem을 해결해야 한다. 수정이 필요한 특정 노드에 접근을 시작하면 w semaphore 변수를 감소시켜 해당 노드의 조회나 수정을 막는다. 만약 다른 thread에서 해당 노드를 조회하거나 수정 중이라면 완료될 때까지 기다리게 된다.

②-ⓐ Pool 구조체

```
typedef struct

{
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready;

    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
} pool;
```

Event-Based Approach를 구현하기 위해 관리중인 file descriptor를 저장하는 구조체이다. 관리중인 file descriptor를 담는 read_set과 select 함수의 반환 값을 저장할 ready_set이 존재하며 고객들과 정보를 주고받기 위한 rio 구조체를 담을 배열이 존재한다.

②-ⓑ Pool 구조체 관련 함수

```
void add_client(int clientfd, pool *p);
```

Select 함수를 통해 이벤트 발생이 확인된 file descriptor 중 listenfd가 존재할 시에 수행되는 함수이다. 새로운 고객의 연결 요청이 발생했으므로 accept를 통해 얻은 file descriptor를 read_set에 추가한다. 또한 해당 file descriptor에 대해 사용할 rio 구조체의 생성 및 pool 내의 관련 변수의 수정이 발생한다.

```
void check_client(pool *p);
```

Select 함수를 통해 이벤트 발생이 확인된 file descriptor 중 listenfd를 제외한 file descriptor에 대한 처리를 수행한다. 이벤트 발생이 확인된 file descriptor에 대해 받아진 명령에 따라 showStock, buyStock, sellStock 함수를 시행한다.

③-ⓐ sbuf 구조체

```
typedef struct
{
    int *buf;
    int n;
    int front;
    int rear;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} sbuf_t;
```

고객의 연결로 생성된 file descriptor를 보관하고 하나의 file descriptor에 대해 복수의 thread에서의 접근을 막기 위한 구조체이다. file descriptor를 저장할 buf 변수, 총 저장 가능 개수, 처음과 마지막 file descriptor의 위치, 그리고 semaphore 변수들을 포함한다.

③-ⓑ sbuf 구조체 관련 함수

```
void sbuf_insert(sbuf_t *sp, int connfd);
```

file descriptor를 sbuf에 저장한다. 이때 slots semaphore 변수를 감소시키며 만약 남는 slot이 존재하지 않는다면 sbuf_remove에서 slots를 증가시켜줄 때까지 기다리게 된다. 이후에 mutex로 동시 접근을 막고 buf에 file descriptor를 추가하며

item semaphore 변수를 증가시킨다.

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

file descriptor를 sbuf에서 제거한다. 이때 items semaphore 변수를 감소시키며 만약 남는 item이 존재하지 않는다면 sbuf_insert에서 items를 증가시켜줄 때까지 기다리게 된다. 이후에 mutex로 동시 접근을 막고 buf에 file descriptor를 제거하며 slots semaphore 변수를 증가시킨다.

④ Thread

```
void *thread(void *vargp)
```

생성된 thread에서 시행될 함수이다. Pthread_detach(pthread_self)를 통해 thread 함수의 종료와 함께 thread가 자동으로 종료된다. sbuf 배열에서 file descriptor를 하나 추출해 해당 file descriptor를 통해 들어오는 요청에 대해 받아진 명령에 따라 showStock, buyStock, sellStock 함수를 시행한다.

3. 구현 결과

① Task1

pool 구조체를 통해 모든 file descriptor를 관리하며, select 함수를 통해 event가 발생한 file descriptor를 처리한다. 이벤트가 발생한 file descriptor가 listenfd일 경우 add_client 함수를 통해 pool에 새로운 file descriptor를 처리하고 그 외에서 이벤트가 발생할 경우 check_clients를 통해 관리하는 모든 file descriptor를 순회하며 동시적인 요청도 절차적으로 처리한다.

② Task2

새로운 고객의 연결로 인해 file descriptor가 생성될 때마다 하나의 thread를 배정해 해당 thread에서 특정 file descriptor를 통해 들어오는 요청을 처리한다. context switching에 의해 concurrent한 server가 구현되며 이때 thread가 공유하는 값들에 대한 처리가 요구된다.

③ Task3

Multiclient.c 내의 값들을 수정하여 고객의 수 및 고객이 요청하는 명령어의 종류를 조정할 수 있다.

4. 성능 평가 결과 (Task 3)

① 확장성

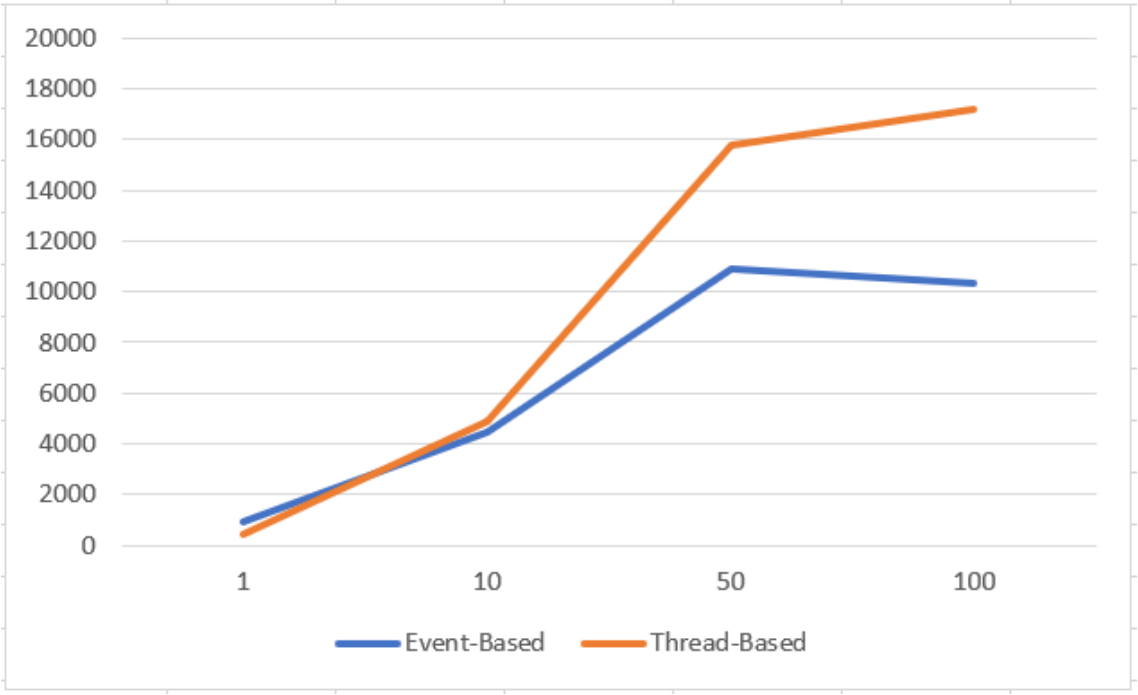
각 방법에 대한 Client 개수 변화에 따른 동시 처리 변화 분석

①-ⓐ 출력결과

Task1 (1 10 50 100)	Task2 (1 10 50 100)
Elapsed Time: 0.021252 seconds Elapsed Time: 0.003196 seconds Elapsed Time: 0.008292 seconds	Elapsed Time: 0.006397 seconds Elapsed Time: 0.000965 seconds Elapsed Time: 0.001973 seconds
Elapsed Time: 0.022821 seconds Elapsed Time: 0.024704 seconds Elapsed Time: 0.019932 seconds	Elapsed Time: 0.020585 seconds Elapsed Time: 0.016597 seconds Elapsed Time: 0.024299 seconds
Elapsed Time: 0.040574 seconds Elapsed Time: 0.043882 seconds Elapsed Time: 0.053017 seconds	Elapsed Time: 0.030914 seconds Elapsed Time: 0.033702 seconds Elapsed Time: 0.030408 seconds
Elapsed Time: 0.105673 seconds Elapsed Time: 0.082834 seconds Elapsed Time: 0.102158 seconds	Elapsed Time: 0.058267 seconds Elapsed Time: 0.057871 seconds Elapsed Time: 0.058655 seconds

①-ⓑ 결과 그래프

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	0.021252	0.022821	0.040574	0.105673
	2nd (s)	0.003196	0.024704	0.043882	0.082834
	3rd (s)	0.008292	0.019932	0.053017	0.102158
	Avg (s)	0.010913	0.022486	0.045824	0.096888
	동시처리율	916.3103	4447.278	10911.23	10321.16
Thread-Based	1st (s)	0.06397	0.020585	0.030914	0.058267
	2nd (s)	0.000965	0.016597	0.033702	0.057871
	3rd (s)	0.001973	0.024299	0.030408	0.058655
	Avg (s)	0.022303	0.020494	0.031675	0.058264
	동시처리율	448.3769	4879.556	15785.49	17163.16



①-ⓒ 해석

예상과 동일하게 client의 수가 증가할수록 구현 방식에 상관없이 동시처리율이 증가하는 경향성을 보인다. Event-Based Approach의 경우 마지막에 동시처리율이 오히려 감소하는데, 이는 3번을 표본으로 잡았기에 발생한 오류로 생각된다.

또한 client의 수가 증가함에 따라 Thread-Based Approach의 동시처리율이 Event-Based Approach의 동시처리율보다 증가함을 볼 수 있는데 이는 Event-Based Approach가 하나의 thread에서 모든 client를 순회하며 절차적으로 작업을 처리하기 때문이다. 이에 반해, event가 발생한 thread에서 자체적으로 작업을 처

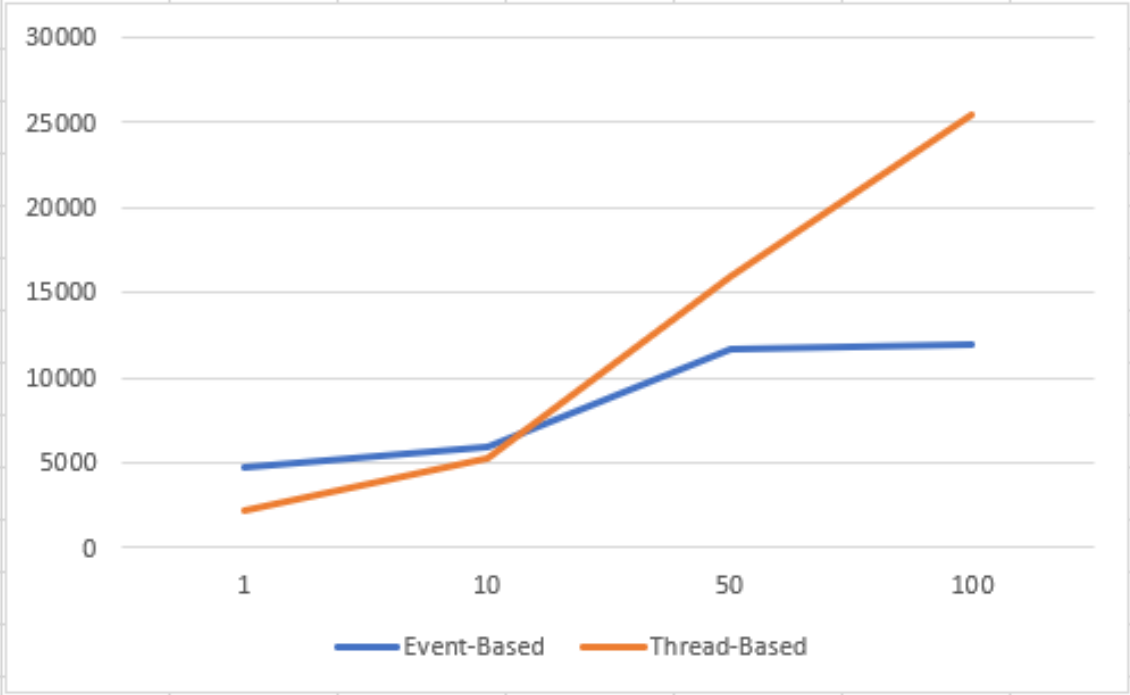
리하여 멀티코어의 이점을 볼 수 있는 Thread-Based Approach가 client의 수에 비례해 성능이 좋아진다.

② 워크로드에 따른 분석

②-ⓐ Client가 buy, sell만 요청하는 경우

Task1 (1 10 50 100)	Task2 (1 10 50 100)
Elapsed Time: 0.002751 seconds Elapsed Time: 0.002868 seconds Elapsed Time: 0.000764 seconds	Elapsed Time: 0.004389 seconds Elapsed Time: 0.008338 seconds Elapsed Time: 0.000780 seconds
Elapsed Time: 0.016748 seconds Elapsed Time: 0.016916 seconds Elapsed Time: 0.016757 seconds	Elapsed Time: 0.020870 seconds Elapsed Time: 0.016378 seconds Elapsed Time: 0.020383 seconds
Elapsed Time: 0.043185 seconds Elapsed Time: 0.044797 seconds Elapsed Time: 0.040317 seconds	Elapsed Time: 0.031887 seconds Elapsed Time: 0.027981 seconds Elapsed Time: 0.034118 seconds
Elapsed Time: 0.086834 seconds Elapsed Time: 0.084789 seconds Elapsed Time: 0.078676 seconds	Elapsed Time: 0.039170 seconds Elapsed Time: 0.037243 seconds Elapsed Time: 0.041373 seconds

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	0.002751	0.016478	0.043185	0.086834
	2nd (s)	0.002868	0.016916	0.044797	0.084789
	3rd (s)	0.000764	0.016757	0.040317	0.078676
	Avg (s)	0.002128	0.016717	0.042766	0.083433
	동시처리율	4699.984	5981.935	11691.44	11985.67
Thread-Based	1st (s)	0.004389	0.02087	0.031887	0.03917
	2nd (s)	0.008338	0.016378	0.027981	0.037243
	3rd (s)	0.00078	0.020383	0.034118	0.041373
	Avg (s)	0.004502	0.01921	0.031329	0.039262
	동시처리율	2221.071	5205.532	15959.82	25469.92



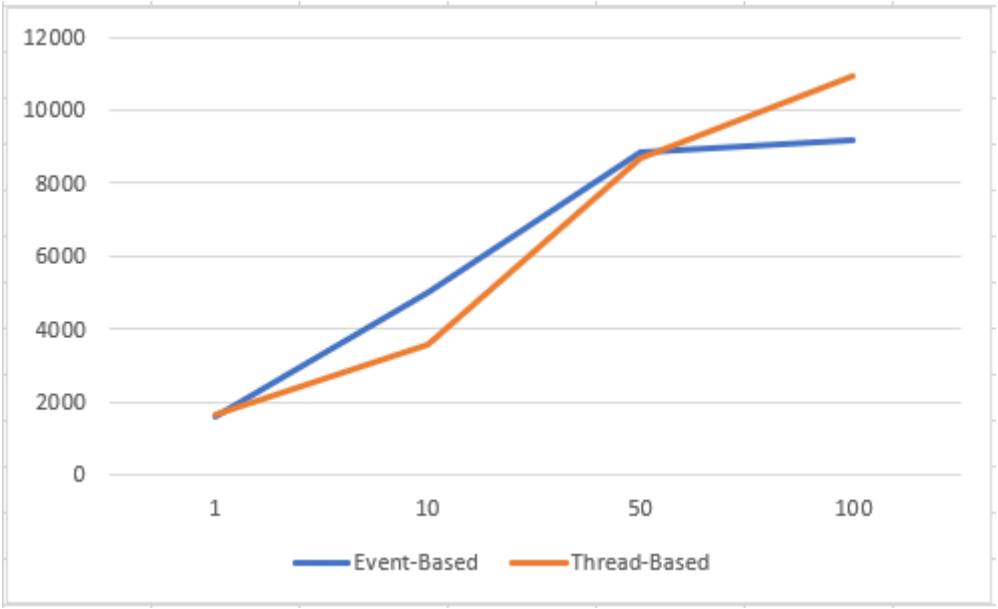
예상과 동일하게, Thread-Based Approach의 경우 buy와 sell 명령어만을 요청 받는 것이 client가 모든 명령어를 요청하는 경우와 show만을 요청 받는 것에 비해 훨씬 높은 동시처리율을 보인다. 이는 주식을 읽는 show 명령어 자체가 모든 주식을 순회하며 문자열에 저장하는 비용이 큰 작업을 수행할 뿐만 아니라 주식을 수정하는 buy, sell 명령어와 동시에 사용되면 각 주식 노드의 semaphore 변수로 인해 show 명령어의 수행이 buy와 sell 명령어가 시행중인 주식 노드에서 진행이 멈추기 때문에 발생하는 지연 때문이다.

②-ⓑ Client가 show만 요청하는 경우

Task1 (1 10 50 100)	Task2 (1 10 50 100)
Elapsed Time: 0.008227 seconds Elapsed Time: 0.008866 seconds Elapsed Time: 0.001401 seconds	Elapsed Time: 0.014509 seconds Elapsed Time: 0.001398 seconds Elapsed Time: 0.002024 seconds
Elapsed Time: 0.016514 seconds Elapsed Time: 0.021631 seconds Elapsed Time: 0.021538 seconds	Elapsed Time: 0.019634 seconds Elapsed Time: 0.028476 seconds Elapsed Time: 0.035157 seconds
Elapsed Time: 0.061690 seconds Elapsed Time: 0.055129 seconds	Elapsed Time: 0.040921 seconds Elapsed Time: 0.077282 seconds

Elapsed Time: 0.052884 seconds	Elapsed Time: 0.054776 seconds
Elapsed Time: 0.107180 seconds Elapsed Time: 0.109116 seconds Elapsed Time: 0.109911 seconds	Elapsed Time: 0.091937 seconds Elapsed Time: 0.089685 seconds Elapsed Time: 0.092560 seconds

Client (10 requests per client)		1	10	50	100
Event-Based	1st (s)	0.008227	0.016514	0.06169	0.10718
	2nd (s)	0.008866	0.021631	0.055129	0.109116
	3rd (s)	0.001401	0.021538	0.052884	0.109911
	Avg (s)	0.006165	0.019894	0.056568	0.108736
	동시처리율	1622.148	5026.557	8838.972	9196.614
Thread-Based	1st (s)	0.014509	0.019634	0.040921	0.091937
	2nd (s)	0.001398	0.028476	0.077282	0.089685
	3rd (s)	0.002024	0.035157	0.054776	0.09256
	Avg (s)	0.005977	0.027756	0.05766	0.091394
	동시처리율	1673.08	3602.868	8671.573	10941.64



show 명령어 만을 요청 받는 것이 모든 명령어를 요청 받는 것에 비해 Event-Based Approach 의 경우 더 좋게, Thread-Based Approach 의 경우 더 낮게 나왔다. 예상했듯이, 해당 비교는 show 명령어 그 자체의 비용과 buy, sell 명령어와 show 를 함께 쓰는 경우 발생하는 지연의 차이가 변수로 작용할 것으로 생각되며, 시행 순간의 buy, sell 명령어와 show 명령어의

비율 혹은 실행 순서 등에 의해 영향을 받아 해당 결과가 발생했다고 예상된다.