

System Programming Project 2

담당 교수 : 김영재

이름 : 20181654

학번 : 오상훈

1. 개발 목표

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

수업에서 배운 echo 서버의 세 가지 구현 방법을 바탕으로 concurrent stock server를 구현한다. task1에서는 event-driven 방식으로, task2에서는 thread-based 방식으로 주식 서버를 설계한다. 본 프로젝트에서 구현하는 주식 서버는 주식 데이터를 바탕으로 여러 client 들과 동시에 통신한다. client는 주식 정보 조회, 주식 판매와 구매, 연결 종료를 주식 서버에 요청할 수 있으며, 주식 서버는 주식 데이터를 바탕으로 여러 client들의 요청을 처리한다. 여러 client들의 동시 접속에 대한 처리를 2가지 방법으로 구현하는 것이 이 프로젝트의 목표이다.

client 가 4개의 명령어로 server와 통신한다.

1. show : 현재 주식의 상태를 보여준다.
2. buy : 주식을 사는 행위를 한다. 만약 server에 있는 주식이 부족하다면 에러 메시지를 받는다.
3. sell : 주식을 파는 행위를 한다.
4. exit : server와의 연결을 종료한다.

주식 데이터는 stock.txt 데이터에 저장되어 있으며, 주식 서버는 이 데이터들을 이진 탐색 트리 형태로 저장한다. 서버가 종료될 때, 변경된 주식 정보들을 stock.txt에 다시 저장한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

event-driven 방식에서는 I/O multiplexing을 사용하여, 하나의 프로세스에서 여러 client와의 통신을 동시에 처리하는 서버를 구현하였다.

2. Task 2: Thread-based Approach

thread-based 방식에서는 thread pool을 생성하고 thread를 관리하여 thread

를 생성 및 삭제하는 데에 필요한 비용을 최소화함과 동시에 여러 client와의 통신을 동시에 처리하는 서버를 구현하였다.

3. Task 3: Performance Evaluation

Task1, Task2에서 사용한 두 가지 방식에 대해서 성능 분석을 위한 metric을 제시하고, 이에 따라 성능 분석을 하였다. client 요청 수, thread 개수 등의 변수에 따라 성능이 달라지는 것을 분석하였다.

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

I/O multiplexing은 process based 또는 thread based와 다르게 서버에서 여러 client의 요청을 받기 위해, 하나의 프로세스 및 control flow에서 모든 처리를 수행하는 방식이다. process 및 thread 생성에 대한 오버헤드가 없다는 장점이 있지만, fine-grained한 동시성을 제공하기 어렵다. 본 프로젝트에서는 client에서 요청하는 명령들(show, buy, sell)이 모두 복잡하지 않은 연산이고, data가 많지 않다는 점에서 fine-grained 한 동시성이 큰 의미를 지니지 않는다.

- ✓ epoll과의 차이점 서술

먼저 select는 fd_set 구조체를 사용하여 read, write, error 에 대한 descriptor 정보를 받을 수 있는데, 이는 FD_ISSET 매크로를 사용하여 read, write, error 에 대한 input이 있는 socket descriptor를 알 수 있다. 하지만 fd_set 구조체는 검사할 수 있는 socket descriptor의 수가 최대 1024개로 한정적이고, input이 있는 socket descriptor를 알기 위해서는 모든 파일 descriptor를 검사해야 한다는 단점이 있다.

epoll은 kernel 레벨의 multiplexing을 지원하는데 epoll을 사용하면 select에 서와 달리 모든 파일 descriptor를 순회하며 FD_ISSET을 하는 문제점이 사라지게 된다. 하지만 epoll의 경우에도 kernel에서 I/O state의 변경을 확인해야 하는 문제는 여전히 존재한다.

- Task2 (Thread-based Approach with pthread)

✓ Master Thread의 Connection 관리

Master thread에서는 getaddrinfo, socket, bind, listen함수를 사용하여, socket descriptor에서 client로부터의 요청을 받을 준비를 한다. listen함수로부터 return받은 listenfd를 accept함수에 전달하여 요청을 기다린다. 요청이 도착하면 client와 연결하고 통신을 시작한다. client에서 요청을 종료하면 master thread에서는 연결을 종료하고 connfd를 close하여 메모리 누수가 일어나지 않도록 한다.

이때, Master thread에서는 multi thread환경에서 발생할 수 있는 producer-consumer문제를 해결하기 위한 sbuf를 만들어 요청이 생기면 sbuf에 insert 한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

master thread는 처음 서버가 실행될 때, 미리 thread들을 생성한다. 이 thread들은 새로운 요청이 생겨, client와의 통신에 할당 받기 전까지 blocked 상태로 기다린다. master thread에서 accept함수를 통해 요청을 받으면, sbuf에 connfd를 insert한다. 이때, sbuf에 connfd가 존재하면 thread들이 해당 connfd를 sbuf에서 remove하면서 해당 client와의 통신을 할당받게 된다. client와의 연결이 종료된 후, sbuf에 새로운 connfd가 존재한다면 해당 connfd로 client와 통신을 하고, 없다면 다시 blocked 상태에서 요청을 기다린다. 또한, 주식 서버 프로그램은 ctrl + c 가 입력되어 sigint signal을 받을 때 종료하게 되는데, thread들은 pthread_detach를 통해 master thread에서 reaping하지 않고, 커널이 reaping한다.

- Task3 (Performance Evaluation)

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

task1과 task2에서 각각 구현한 event driven, thread based server의 perfomace evaluation을 하기 위해, 동시처리율을 정의하였다. 동시처리율 P는 시간당 client 처리 요청 개수로, 다음과 같은 식으로 나타낼 수 있다.

$$P = (N \times ORDERSPERCLIENT) \div T$$

여기서 N은 client 개수, ORDERSPERCLIENT는 한 client에서 server에 보내는

요청 개수, T는 elapse time이다. 위와 같이 동시처리율을 정의한 이유는 단순히 서버에서 요청을 처리하는 시간인 elapse time 만으로는 객관적인 서버의 성능을 측정할 수 없기 때문이다. client에서 보내는 요청의 종류, 개수 등에 따라서 달라질 수 있다. 따라서 Task3에서는 client 개수 변화에 따른 동시처리율, client 요청 타입에 따른 동시처리율에 대한 분석을 하였다. 또한 두 방식의 메모리 사용량을 비교하였다.

elapse time은 N개의 자식 프로세스를 생성하여 각 자식 프로세스가 ORDERSPERCLIENT 수만큼 server에게 요청을 보내는 multiclient 함수를 변경하여 측정하였다.

✓ Configuration 변화에 따른 예상 결과 서술

우선 task1, task2 공통적으로 stock item 의 개수에 따라서 elapse time이 바뀔 수 있다. 따라서 task3에서는 stock item의 개수는 10개로 설정하여 동일하게 elapse time을 측정하였다.

stock.txt:

```
cse20181654@cspro4:~/sp/project2/task_1$ cat stock.txt
1 1 100
3 1214 1200
2 1989 2000
4 4984 5000
9 154 100
8 139 100
6 521 500
5 3724 3700
7 216 300
10 2057 2000
```

task2에서는 몇 가지 고려할만한 configuration이 존재한다. 첫 번째로 thread pool 에 존재하는 thread의 개수이다. 만약 thread pool의 thread 개수가 1개 라면 serial하게 요청을 처리하는 것과 같은 셈이 된다. 반대로 thread 개수가 많다고 하더라도 cpu개수는 한정적이기 때문에 무조건적으로 server의 성능 및 효율이 되는 것은 아닐 것이다. 따라서 적절한 개수의 thread 개수 설정이 필요한데, server를 실행하는 환경(cspro)의 cpu 개수와 같은 개수로 thread 개수를 설정할 때 효율 및 성능이 가장 좋을 것이라고 예측한다. 서버에서 하이퍼 스레딩을 사용할 경우, 서버 cpu 개수의 두 배에 해당하는 thread를 사용할 때 효율 및 성능이 가장 좋을 것이라고 예측한다.

두 번째 configuration은 sbufsize이다. sbuf는 client요청을 받아서 처리가 되기를 기다리는 배열인데, 만약 sbuf의 크기가 thread 개수에 비해 작다면 thread가 lock이 걸리는 상황이 자주 발생할 것이며, 성능이 저하될 것이라고

예측한다.

또한 client의 개수 및 client에서 보내는 요청의 타입에 따라서도 성능이 달라질 것이라고 예측한다. buy 와 sell 명령은 큰 차이가 없지만, show는 stock BST를 순회해야 하기 때문에 BST에 있는 모든 노드를 순회할 필요가 없는 buy 와 sell 보다 동시 처리율이 낮을 것이라고 예측한다.

event-driven과 thread-based 중에서는 event-driven 서버가 더 좋은 동시 처리율을 보일 것이라고 예측한다. 그 이유는 thread-based의 경우, context switching하는 데에 process-based보다는 오버헤드가 적지만 여전히 존재하기 때문이다. 또한 thread-based server는 semaphore를 사용하여 lock을 잡기 때문에 하나의 프로세스에서 lock 없이 처리하는 event-driven가 더 좋은 동시처리율을 보일 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

select 함수를 사용하여 event driven 서버를 구축하였다. 동시성을 제공하기 위해 다음과 같이 connected descriptor pool을 정의하였다.

```
typedef struct {    // connected descriptor pool
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready; // number of ready descriptor
    int maxi;
    int clientfd[FD_SETSIZE]; // active socket descriptors
    rio_t clientrio[FD_SETSIZE]; // read buffers
} pool;
```

fd_set 타입의 read_set 은 active descriptor의 집합을 저장한다.. cliendfd는 client와 통신하기 위해 사용되는 socket descriptor들을 저장하는 배열이다.

한편, 주식 item에 대한 정보를 binary search tree에 저장하기 위해 다음과 같이 Stock과 Node 구조체를 사용하였다.

```

typedef struct {    // stock item
    int ID;        // stock id
    int left;      // number of left stock
    int price;     // stock price
} Stock;

typedef struct _Node{    // node of stock binary search tree
    struct _Node* left;  // left child
    struct _Node* right; // right child
    Stock data;          // stock data
} Node;

```

구현 내용은 accept함수에서 요청을 받으면 add_client 함수에서 pool에 해당 client와 통신하기 위한 정보들을 insert한다. 이후 check_client 함수 내에서 현재 pending input이 있는 모든 client와의 통신을 처리한다. show 명령어에 대해서는 print_stock_data함수를 통해 BST를 순회하며, buf배열에 response내용을 저장한다. buy 명령어에 대해서는 find_stock 함수를 통해 BST에서 pre-order traverse를 통해 해당 id에 맞는 노드를 찾고, reduce_stock 함수를 호출하여 이 노드의 정보를 변경한다. sell 명령어에 대해서는 buy와 마찬가지로 find_stock 함수를 사용하여 노드를 찾고, add_stock 함수를 호출하여 이 노드의 정보를 변경한다.

ctrl+c가 입력되어 sigint 시그널을 받을 경우에는 BST, listenfd등과 같은 모든 할당된 메모리들을 해제해준 뒤, 서버 프로그램을 종료한다.

- Task2

pthread를 사용하는 thread based 서버를 구축하는 경우에는 이전 task1과 다르게 thread pool을 생성하기 위해 다음과 같은 구조체를 선언하였다.

```

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;

```

한편, 주식 아이템에서 task 1과 의 차이점은 multi thread 환경에서 각 주식 아이템(노드)들을 read, write 할 때, race 같은 문제가 발생할 수 있으므로, 다음과 같이 read_mutex, write_mutex, readcnt를 포함하여 구조체를 선언하였다.

```

typedef struct {
    int ID;           // stock id
    int left;         // number of left stock
    int price;        // stock price
    int readcnt;      // number of reading thread
    sem_t read_mutex; // protect reading accesses to buf
    sem_t write_mutex; // protect writing accesses to buf
} Stock;

typedef struct _Node{ // node of stock binary search tree
    struct _Node* left; // left child
    struct _Node* right; // right child
    Stock data; // stock data
} Node;

```

구현 내용은, master thread에서 요청을 받기 이전에 thread들을 미리 생성하여 thread pool을 생성한다. 이 thread들은 sbuf의 자원을 요구하는데, sbuf는 client로부터 요청이 생기기 전까지는 empty 상태이므로, thread들은 blocked상태로 유지된다. client로부터 요청이 생기면 하나의 client에 하나의 thread를 할당하여 통신이 이루어진다. 하나의 thread에서 show, buy, sell명령을 처리하는 것은 이전 task1과 동일하지만, race가 발생할 수 있으므로 주의해야 한다. show, buy, sell에서 호출하는 read 함수인 print_stock_data, find_stock에서는 어떤 노드에 접근하여 읽을 때, P, V operation을 통해 write 연산을 하려는 thread를 block 시킨다. writer thread는 reader가 없을 때 write연산이 가능하다. buy와 sell에서 호출하는 reduce_stock, add_stock함수에서는 다른 writer thread가 해당 노드를 동시에 write하는 것을 방지한다.

- Task3

task3에서는 task1과 task2에서 구현한 서버들의 성능을 측정하기 위해 시간을 측정하는 코드를 삽입하였다. 그리고 프로젝트에서 제공된 multiclient 코드를 변경하여 delay 없는 요청을 하도록 하였다. 분석 방법에 따라서 thread의 개수, client의 개수, 요청 명령어에 따라 코드를 변경하여 성능 측정을 하였다.

3. 구현 결과

- 2번의 구현 결과를 간략하게 작성

task2에서 thread based서버를 구현한 결과 multi thread 환경에서 semaphore를 맞게 사용하여 race 문제가 발생하지 않는 것을 확인할 수 있었다.

하지만 context switching이 일어남에 따라 event-based 서버보다는 동시처리율이 낮을 것을 확인할 수 있었다.

- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

이번 프로젝트에서는 thread를 사용함에 따라 생길 수 있는 여러 가지 문제들을 해결하기위해서 readers-writers problem에서 first readers-writers problem(favors readers)를 채택하여 스레드의 동기화 문제를 해결하였다. 그 이유는 read연산이 write연산보다 많을 것이라고 예상하였기 때문이다. show뿐만이 아니라, buy와 sell 명령어 에서도 주식의 id를 통해 binary search tree를 순회하는 read연산이 수반되기 때문에 reader를 우선시하는 방식으로 문제를 해결하였다. 하지만, 본 프로젝트의 주식 서버는 주식 계정 및 잔고, 주식 발행량 수, 가격 변동 등 여러 가지 조건들을 생략하고 구현한 것이기 때문에, 실제와 유사한 주식 서버를 만드려고 한다면, reader를 우선시하는 방식으로는 문제가 발생할 가능성이 있다. 따라서 thread 동기화를 보장하면서도, 위의 문제들을 해결하기 위해서 논문, "[Faster Fair Solution for the Reader-Writer Problem](#)"에서 제시한 fair reader-writer problem을 적용해볼 수 있다.

4. 성능 평가 결과 (Task 3)

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

4.1. 실험 환경

서버 (cspro.sogang.ac.kr) :

```
top - 08:26:45 up 27 days, 4:10, 58 users, load average: 14.06, 14.11, 14.14
Tasks: 3 total, 1 running, 2 sleeping, 0 stopped, 0 zombie
%Cpu(s): 70.1 us, 0.2 sy, 0.0 ni, 29.7 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 65853364 total, 53046264 free, 3407580 used, 9399520 buff/cache
KiB Swap: 628732 total, 439484 free, 189248 used. 61052536 avail Mem
```

클라이언트 (cspro4.sogang.ac.kr) :

```
top - 08:27:49 up 27 days, 4:11, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 262 total, 1 running, 261 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 0.0 sy, 0.0 ni, 99.9 id, 0.1 wa, 0.0 hi, 0.0 si, 0.0 st
KiB Mem : 65853492 total, 63335316 free, 280008 used, 2238168 buff/cache
KiB Swap: 628732 total, 439372 free, 189360 used. 64441276 avail Mem
```

위의 두 사진은 각각 서버와 클라이언트 환경에서 top명령어를 입력했을 때의 결과이다.

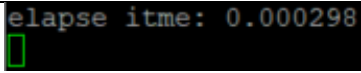
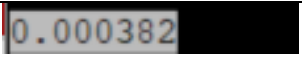
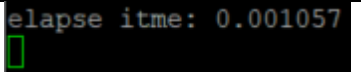

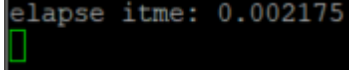

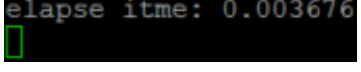

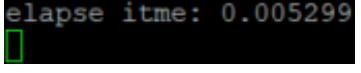

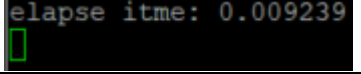

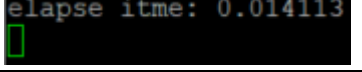

4.2. client 개수에 변화에 따른 동시처리율

아래 모든 그래프들은 각각 5번 elapse time을 측정했을 때의 평균값으로

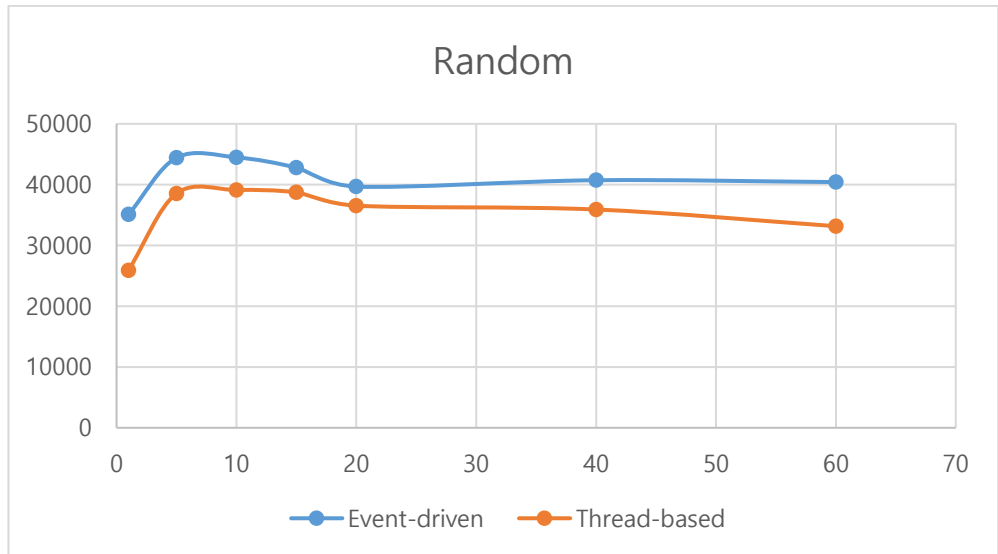
작성하였으며, 사진은 5번 중의 일부이다.

한 client당 요청의 수는 10이며, thread의 수는 20, SBUF의 size는 100이다. 2.B의 task3 부분에 설명한 것과 같이 thread의 개수가 core의 개수 또는 그 2배에 해당할 때 동시처리율 및 성능이 증가할 것이라고 예측하였기 때문에 thread의 개수를 20(cspro 서버의 core 수)개로 설정하여 실험하였다.

아래 표는 event driven 방식과 thread based 방식의 성능 측정 결과 예시이다.

	event driven	thread based
N = 1		
N = 5		
N = 10		
N = 15		
N = 20		
N = 40		
N = 60		

client에서의 요청은 show, buy, sell을 모두 random하게 요청한 것이다. 이를 아래 그래프로 나타내면 다음과 같다.

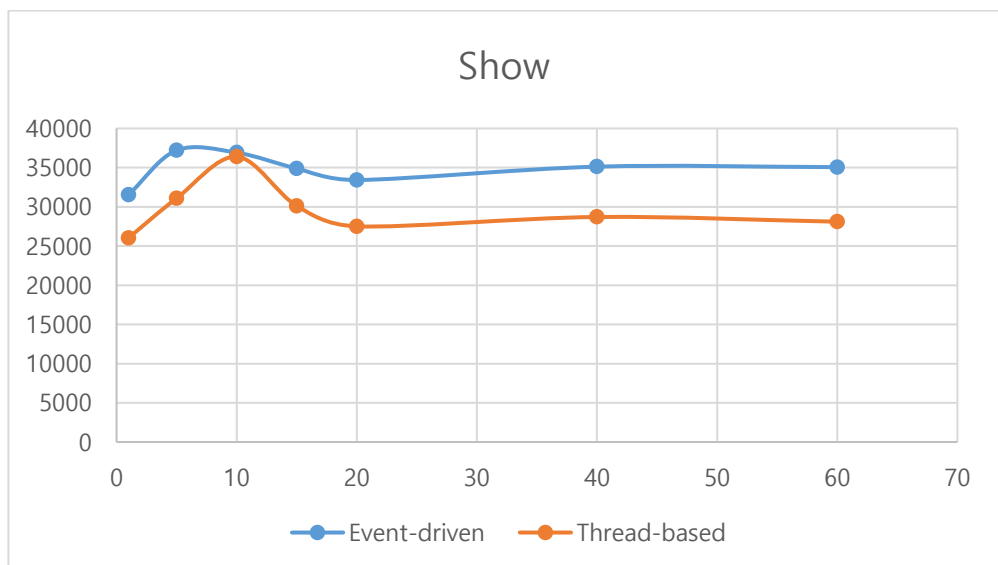


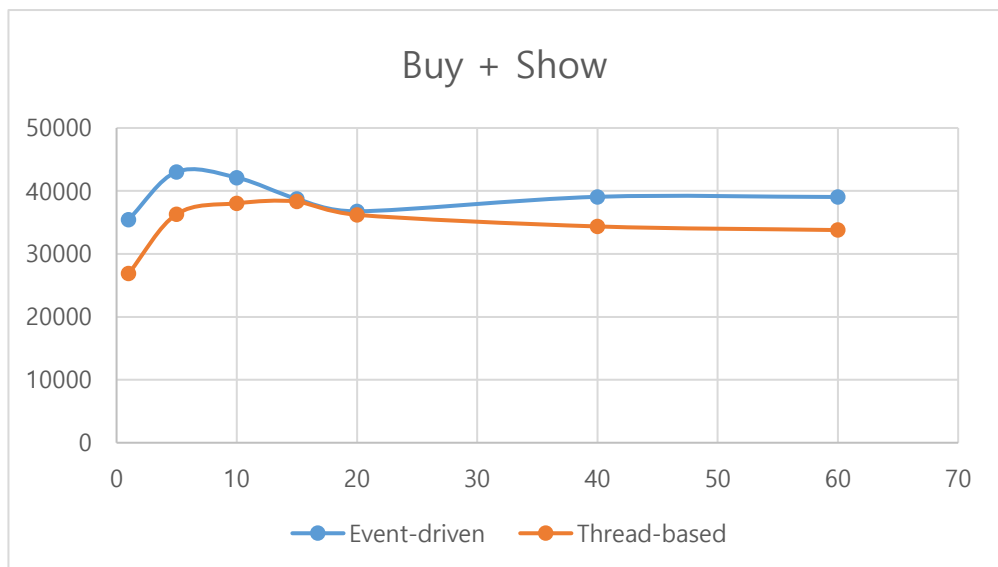
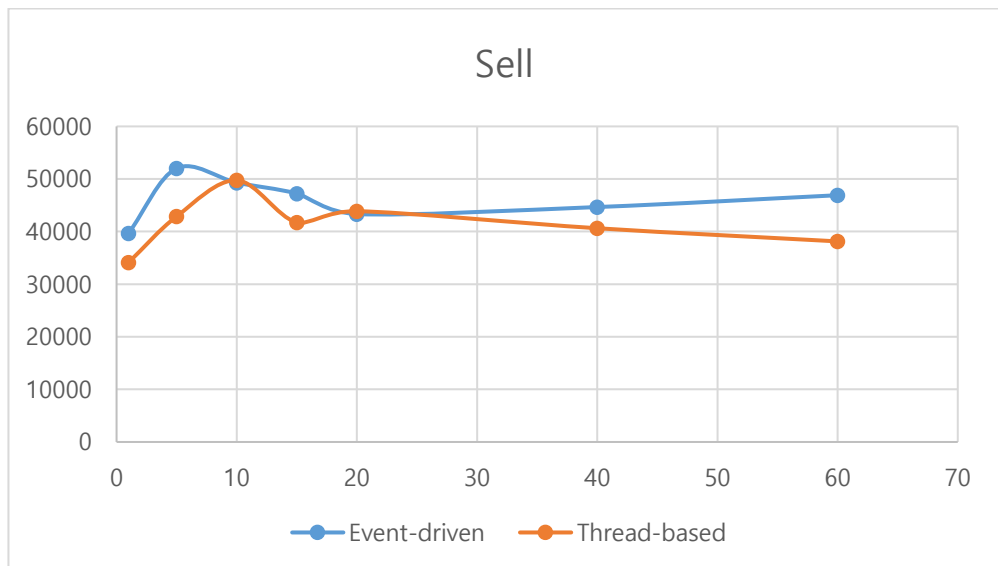
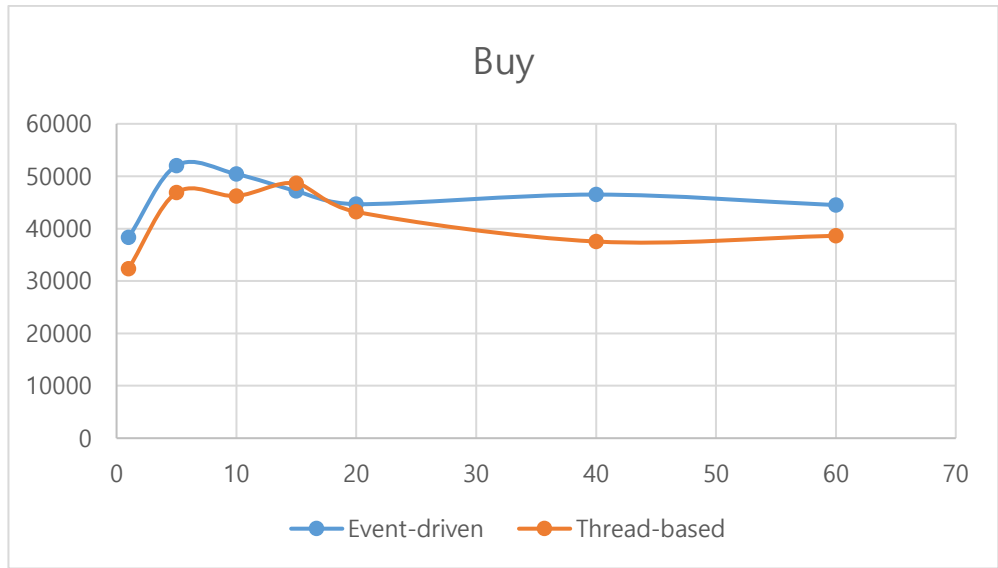
먼저 실험한 client의 개수는 1, 5, 10, 20, 40, 60이다. event-driven과 thread-based 서버 모두에서 client의 개수가 5일 때 동시처리율이 가장 높은 것을 확인할 수 있었다. client의 개수가 5개보다 커지면서 동시처리율이 점점 감소하거나 그와 비슷한 수준을 나타내었다.

또한 예측과 동일하게 전체적으로 event-driven 서버가 thread-based 서버보다 더 좋은 동시처리율을 보이며, event-driven 서버가 더 좋은 확장성을 제공한다는 것을 확인할 수 있었다.

4.3. 워크로드에 따른 동시처리율

아래 그래프들은 show, buy, sell, show + buy 만을 client가 요청하였을 때에 대한 동시처리율 그래프이다.

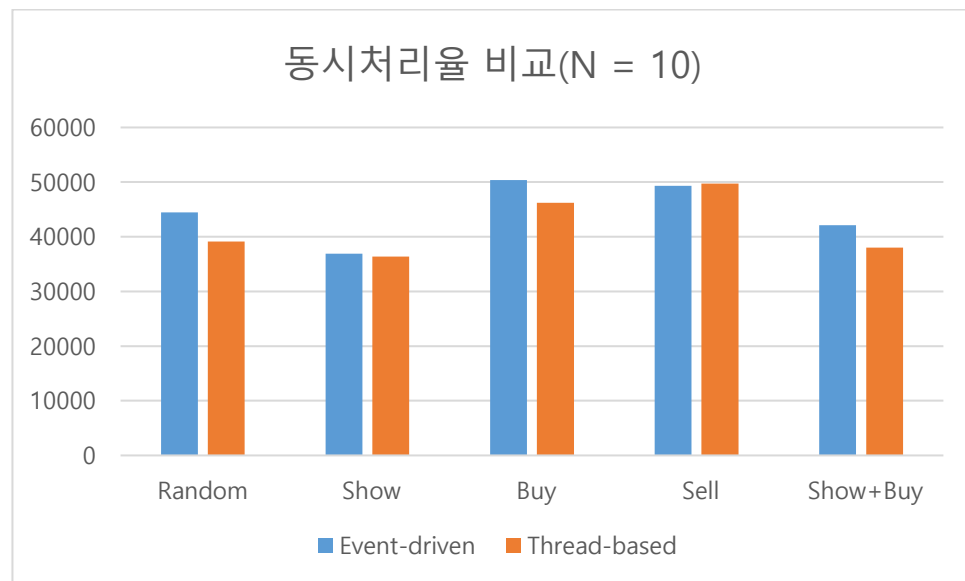




위의 4가지 경우 모두 전체적으로 4.2의 random 경우와 동일하게 event-

driven 서버가 thread-based서버보다 더 좋은 동시처리율을 보이는 것을 확인할 수 있었고, 그래프의 모형 또한 client의 개수가 5일 때 최고점을 보이며 대동소이한 것을 확인할 수 있었다.

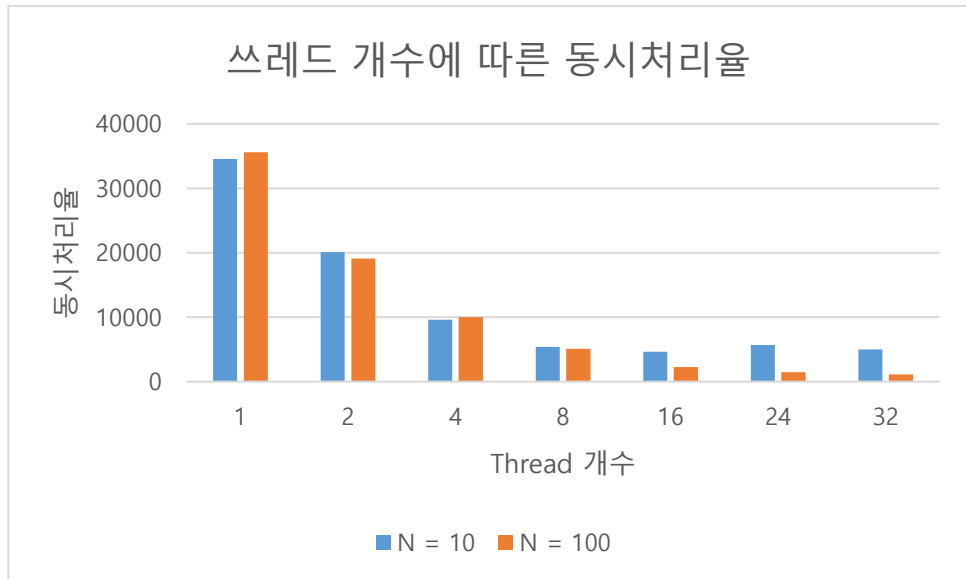
아래는 client개수가 10일 때 각각 워크로드에 따른 동시처리율 비교 그래프이다.



예측한 것과 같이 show는 모든 BST의 노드를 순회해야 하기 때문에 Buy와 Sell보다 동시처리율이 낮을 것을 확인할 수 있었다. Buy와 Sell은 트리에서 ID가 같은 노드를 찾고 값을 변경시키는 작업은 같으므로 동시처리율이 거의 비슷한 것을 확인할 수 있었고, random(show + buy + sell)과 show + buy은 거의 비슷한 것을 확인할 수 있었다.

4.4. 쓰레드 개수에 따른 동시처리율

예측에서 쓰레드 개수가 증가함에 따라 동시처리율이 증가할 것이라고 예측하였다. 아래는 쓰레드 개수에 따른 동시처리율 그래프이다.



client의 개수가 10일 때와 100일 때의 동시처리율을 비교하였다. 예측과는 반대로 쓰레드의 개수가 증가하면 동시처리율이 감소하는 것을 확인할 수 있었다. 그 이유는 task2에서 구현한 thread-based서버는 단순한 구현으로 P와 V함수로 인해 lock이 걸리기 때문에 동시처리율이 점점 감소하는 것을 확인할 수 있었다. 만약 예측처럼 thread 개수가 코어의 수와 비슷할 때 동시처리율 및 speedup이 증가하도록 서버를 구현하기 위해서는, Thread별로 요청 log 리스트를 유지하여 그에 해당하는 response를 보내주는 서버 디자인을 고려해볼 수 있다.