# Multicore Programming Project 2

담당 교수 : 박성용

이름 : 이혜인

학번 : 20191426

## 1. 개발 목표

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

이번 프로젝트에서는 concurrent한 주식 서버를 만들기 위하여 I/O multiplexed Event driven server 기법과 multithread server를 만들었다. Event-driven server는 select 함수를 이용하여 listenfd로부터 들어오는 connfd를 pool 구조체의 fd_set 형식인 비트맵과 cliendfd 배열을 이용해 관리하고, multithread server에서는 thread를 먼저 생성해 놓은 다음, client의 연결 요청으로부터 생성된 connfd를 sbuf_t 구조체의 buf 배열에 저장하여 각각의 thread가 하나씩 connfd를 가져와 client의 요청을 수행할 수 있게 하였다.

## 2. 개발 범위 및 내용

### A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

: Client의 연결 요청 받아 명령(Show, Buy, Sell, Exit)에 대한 서비스 제공

2. Task 2: Thread-based Approach

: Client의 연결 요청 받아 명령(Show, Buy, Sell, Exit)에 대한 서비스 제공

3. Task 3: Performance Evaluation

: Task1과 Task2가 명령을 수행함에 있어서 걸리는 소요 시간을 측정, 동시 처리율 비교

### B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

: Multi-client 의 요청을 Concurrent 하게 처리하기 위해서 하나의 Process 안에

서 Accept에 성공된 connfd를 하나의 배열에 저장하고, 루프를 돌면서 해당 배열을 확인하여 pending된 connfd이 있다면 이에 맞는 서비스를 제공한다. 루프에 들어가기 전에는 하나의 listenfd를 생성하여 client 로부터 들어오는 연결 요청을 Accept할 수 있게 한다. 이후 connfd만을 모아놓은 배열을 확인하여 연결된 connfd로부터 요청이 있다면 이를 읽어들여 command(show, sell, buy, exit)에 맞는 결과를 해당 connfd에 write하여 원하는 결과를 서비스한다. 만약 client로부터 연결 종료 요청(입력 바이트가 0이거나 exit 명령어인 경우)이 들어온다면 connfd를 close하고, connfd만을 모아놓은 배열에서 해당 connfd를 제거하여 더 이상의 요청을 받지 않도록 한다. 서버를 종료하는 경우에는 Sigint handler를 설치하여 exit()으로 서버를 종료하도록 한다.

✓ epoll과의 차이점 서술

: Select 함수는 listenfd나 connfd에 요청이 들어오는지를 iterative하게 계속 확인해야 하고, 확인할 fd에 변화가 생기기 전까지 무한정으로 대기해야 한다는 단점이 있다. 이에 반해, epoll 함수는 운영 체제에게 요청이 들어올 대상에 대한 정보를 제공하고, 이 대상의 변화가 생길 때 변경 사항을 제공받을 수 있게 한다. 따라서, listenfd나 connfd의 변화를 확인하기 위해 매번 반복문을 통해 이를 체크할 필요가 없으며, 이들에 대한 정보도 매번 제공할 필요가 없다.

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

✓ Master Thread의 Connection 관리

: Master Thread에서는 초기 NTHREADS 값 만큼 미리 thread들을 생성해놓는다. 이후 listenfd로부터 연결이 들어오면 이로부터 생성된 connfd를 sbuf의 buf에 저장해놓고, 미리 생성해놓은 thread 중 하나가 sbuf에서 하나의 connfd를 꺼내어 connfd로부터 들어오는 요청을 읽어들여 command에 맞는 결과를 해당 connfd로 write한다. 이 때 connfd에서 연결이 끊어지면 해당 thread도 connfd를 닫는데, 이 때 thread를 reaping 하기 위해서 pthread_detach() 함수를 이용해 운영체제가 이를 reaping하도록 한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술:

: Thread pool을 관리하는 부분에 있어서, connfd를 sbuf의 buf에 집어넣고 가져오는 과정에 있어서 producer-consumer 문제가 발생하므로, mutex와 counting

semaphore를 이용하여 thread간의 race condition이 발생하지 않게 한다. 이는 sbuf_insert() 함수와 sbuf_remove() 함수를 이용하여 해결하였다. sbuf_insert() 함수에서는 맨 처음 P(&sp->slots)를 이용하여 초기 slot = n개만큼에 대해서 insert를 가능하게 하고, 직접 buf에 connfd를 집어넣는 부분은 critical 한 부분이므로 mutex lock 으로 이를 보호한다. 마지막으로는 V(&sp->items)으로 connfd를 넣어준 만큼 item을 증가시킨다. 이로써 slot이 n개보다 많을 경우에는 slot에 자리가 날 때 까지 P(&sp->slots)에서 기다릴 수 있게 한다.

반대로, sbuf의 buf에서 connfd를 제거하는 sbuf_remove()에서는 맨 처음 P(&sp->items)를 이용하여 꺼낼 수 있는 item의 갯수만큼 remove 를 가능하게 하고, 직접 buf에서 connfd를 꺼내는 부분은 critical 한 부분이므로 mutex lock으로 이를 보호한다. 마지막으로는 V(&sp->slots)으로 connfd를 삭제한 만큼 slot을 증가시킨다. 이로써 item이 0개일 경우에는 slot에 이용 가능한 connfd가 들어올 때 까지 P(&sp->items)에서 기다릴 수 있게 한다.

마지막으로, Event-based

- **Task3 (Performance Evaluation)**

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

  : 얻고자 하는 metric은 동시 처리율로, 이는 (client process 수) * (process 당 request 수 = 10) / 총 시간 으로 정의하였다. 이 때, 총 시간은 server가 listenfd로부터 처음으로 client의 연결 요청을 accept하는 순간부터, 마지막 요청을 받아들이는 순간 까지를 clock() 함수를 이용하여 측정하였다. 이후, SIGINT를 발생시켜 server를 종료시킬 때, clock() 함수를 이용해 측정한 시작 시각과 종료 시각의 차이를 이용해 elaped time을 구하고, 위의 공식에 값을 대입하여 구한 동시 처리율을 time.txt 파일에 기록하게 하였다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  : 만약 show 명령어만을 출력한다면, 주식 정보들로 구성되는 tree를 모두 순회하여 출력해야 하므로, 단일 노드를 검색해 정보를 업데이트하는 buy, sell 명령어보다는 근소하게 처리율이 낮을 것으로 예상된다. 그렇지만, buy 혹은 sell 명령

을 수행할 때, 최악의 경우에는 모든 트리를 순회하여 찾고자 하는 노드를 검색해야 한다. 따라서, 명령어의 종류에 따른 차이는 근소할 것으로 예상된다.

만약 client process의 수를 증가시킨다면, concurrent하게 구현된 서버는 이 명령들을 동시다발적으로 수행할 수 있을 것이므로, 총 시간은 비슷하나 처리된 명령어 수가 늘어나므로 동시처리율이 증가할 것을 기대해 볼 수 있다.

마지막으로, task1에 비해 multithread 방식으로 구현된 task2에서는, 멀티코어 CPU에서 각각의 thread가 여러개의 프로세서에서 병렬적으로 수행될 수 있으므로, task1에 비해 동시 처리율이 증가할 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)**

- **Task 1.**

(1) Pool 구조체 설정: csapp.h 파일에서 pool 구조체를 선언하였다. 이는 select 함수에 사용하기 위해 필요한 비트맵인 read_set과 ready_set을 가지고 있으며, 반복문을 효율적으로 사용하기 위해 필요한 int형의 maxfd와 maxi, 그리고 client로부터 연결된 fd와 이에 해당하는 rio를 관리하기 위해 필요한 int clientfd 배열과 rio_t clientrio 배열을 선언하였으며, 마지막으로는 select의 결과값을 받아들이기 위한 int형의 nready를 선언하였다.

(2) Select 함수 사용 : Select 함수를 통해 read_set으로부터 연결 요청이 들어온 connfd만을 select하고, 이를 선별하여 ready_set에 저장하도록 한다. select의 반환값은 event 가 생긴 connfd의 갯수인데, 이 값이 -1일 경우에는 오류이므로 이를 handling 하는 코드를 추가하였다.

(3) add_client : 만약 listenfd에 event가 발생해 연결 요청이 들어왔다면, 새 connfd를 생성하여 연결 요청을 받아들이고, 이 connfd를 pool 구조체의 clientfd에 저장하고, 이에 맞는 새 rio 를 할당한다. 만약 할당할 배열이 꽉 찬 경우에는 에러 메세지를 리턴한다.

(4) check_client : connfd로부터 event가 발생한 경우를 체크하고, 그렇다면 해당

connfd의 rio로부터 들어오는 입력을 읽어들여 command_process 함수에 이 명령어를 인자로 넘겨준다. 만약 들어오는 입력의 byte 수가 0이거나, command_process 함수의 결과값이 -1(exit 명령)인 경우에는 connfd를 종료하고, connfd와 관련된 값을 초기화한다.

(5) command_process : 이는 client의 명령을 읽어들여 이 명령어가 show, buy, sell, exit 명령어중 어느 명령어에 속하는지를 확인한 후, 이에 걸맞는 코드를 수행한다. 만약 show 명령어가 입력되었다면, tree의 모든 노드를 순회하여 이를 buf에 저장하고, 해당 내용을 connfd에 작성하여 보낸다. 만약 buy 나 sell 명령어일 경우, tree에 타깃 ID를 검색한 후 이 노드의 주식 개수 정보를 수정하고, buy나 sell 명령어가 성공(혹은 실패)되었다는 내용을 buf에 옮겨 connfd에 작성하여 보낸다. 마지막으로 exit 명령의 경우, -1을 return하여 해당 connfd를 close할 수 있게 한다.

(6) sigint_handler : 서버를 종료시킬 때 Ctrl-C를 눌러 종료시키므로, SIGINT handler를 설치하였다. 이는 여태까지 수정시킨 주식 정보(tree)를 다시 stock.txt 함수에 작성하고, exit()을 통해 주식 서버를 종료시킨다.

- **Task2.**

(1) sbuf 구조체 : sbuf 구조체는 thread pool 의 관리를 위해 사용되는 구조체이다. 이는 thread들 간의 connfd를 배정하기 위해 사용되는 int * 형의 buf 배열과, queueing을 위한 인덱스들(front, rear, n), 그리고 mutex lock과 counting semaphore를 위한 변수들이 존재한다.

(2) sbuf_init: mutex lock에 사용되는 sem_t mutex의 값을 1로, counting semaphore에 사용되는 sem_t slots과 items를 각각 n, 0으로 초기화한다.

(3) sbuf_insert: 비어있는 slot에 connfd를 넣기 위해 바깥으론 counting semaphore를 이용해 slot을 관리하고, buf에 직접 connfd를 넣기 위해 안에서는 mutex lock을 걸어 보호한다.

(3) sbuf_remove: 차있는 slot에 connfd를 빼기 위해서 counting semaphore를 이용해 item을 관리하고, buf에 직접 connfd를 빼기 위해 안에서는 mutex lock을 걸어 보호한다.

(4) Pthread_create : 먼저, 정해진 NTHREADS 만큼의 thread를 미리 create해 놓는다.

(5) thread: 생성된 thread는 먼저 추후 reaping을 위해 pthread_detach함수를 이용해

OS가 reaping 하게끔 하고, sbuf의 slot에 connfd가 있다면 이를 sbuf_remove를 통해 빼내어와 해당 connfd로 client 의 요청을 수행한다.

(6) echo: thread가 가져온 connfd를 받아 client의 명령을 Rio_readlineb로 읽어온 다음, 해당 명령을 command_process(task1과 동일)로 넘겨 client로 결과를 넘겨준다.

## 3. 구현 결과

- **Task 1 & 2.**

**:** Task1과 Task2에서는 구현된 방법만 다를 뿐, 같은 작업을 수행한다. 먼저 Server에서 주어진 포트 번호로 listening socket을 열면, client에서 server의 IP 주소와 포트 넘버를 이용하여 해당 listening socket으로 연결 요청을 한다. 만약 정확하게 연결이 맺어진다면, connfd를 생성한다. server에서는 먼저 client가 connfd에 작성한 명령을 읽어들이고, 이를 분석 후 알맞는 요청을 다시 connfd에 쓴다. Client는 이후 server로부터 온 정보를 다시 connfd로부터 읽어들여 요청에 대한 답을 얻는다. 따라서, Client가 show 명령을 내리면, server는 주식 정보에 대한 트리를 읽어들여 이를 순회하면서 client 에게 주식 정보를 제공하고, Buy나 Sell 명령어를 내리면 server는 client가 원하는 주식 ID를 가진 노드를 방문하고, 그 정보를 변경하여 주식 정보에 반영한다.

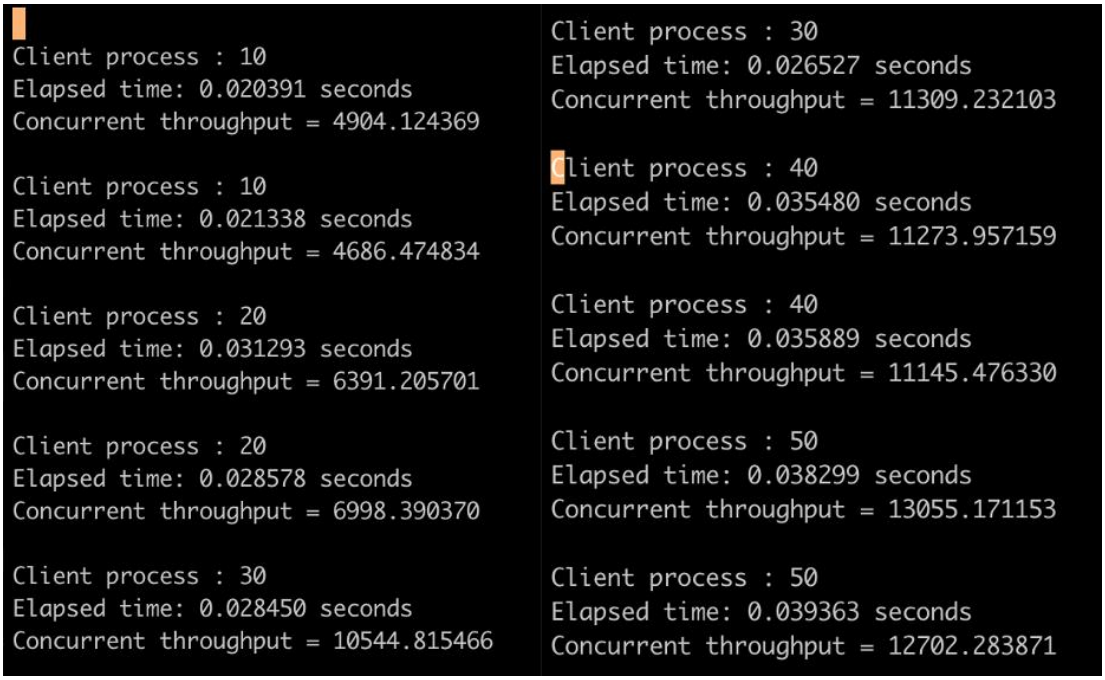
- **Task 3.**

**:** Task2에서는 clock() 함수를 이용하여 시작 시간과 종료 시간을 측정하고, 동시 처리율의 공식에 값을 대입하여 그 결과를 time.txt에 출력하여 결과에 대한 분석을 가능하게 하였다.

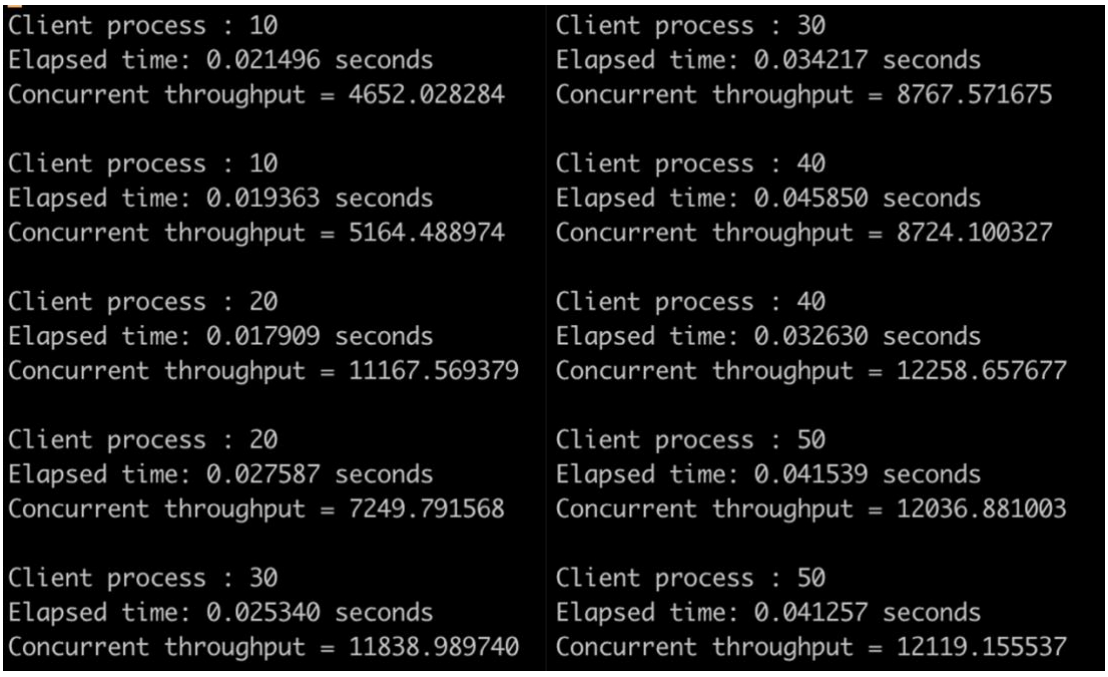
## 4. 성능 평가 결과 (Task 3)

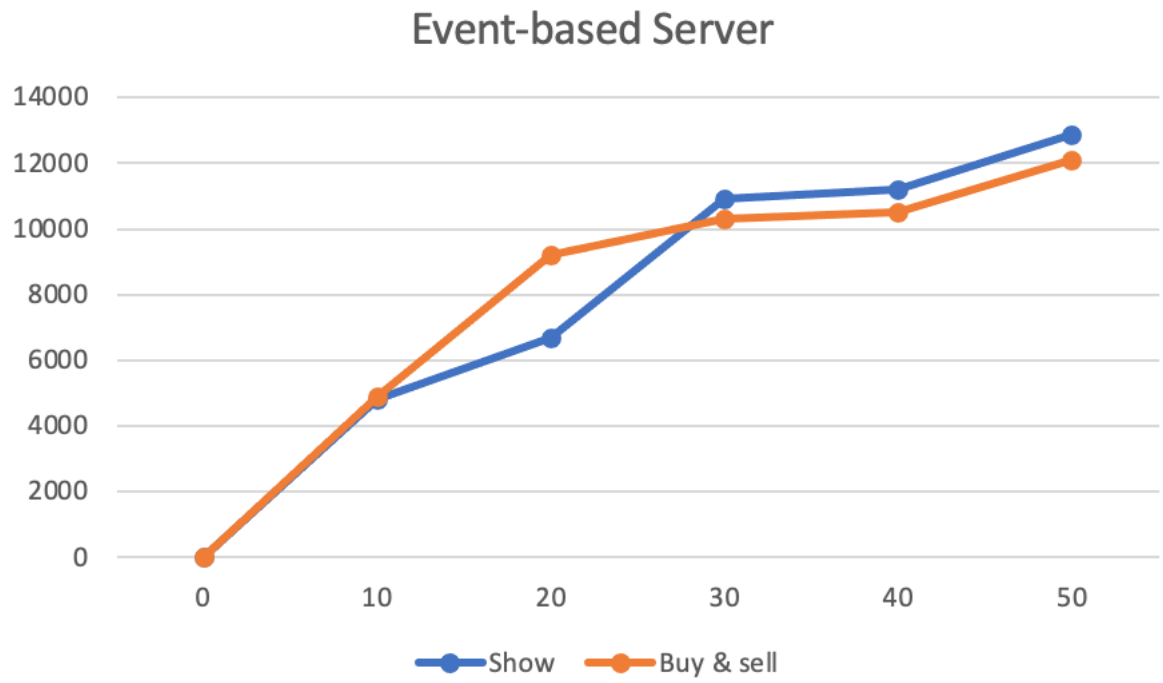
- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡쳐 포함)**

- **(1) 여러가지 command를 섞었을 경우 - Task 1 :**




- (2) Buy, Sell command만 수행하는 경우 - Task 1






-

- 두 command 그룹 모두, client 갯수 30개까지는 linear에 근사한 모습을 보였지만, 이후에는 그 기울기가 살짝 줄어든 모습을 볼 수 있다. 이는 client 가 많아질수록 이를 iteratively하게 처리하게 되므로 인해서 발생하는 overhead로 추측된다.
- 그러나, command에 따라서 동시처리율이 변화한다고 보기 어려우며, client process 50개가 되었을 때 이 둘은 모두 1만 2천대의 동시처리율을 보이고 있다.
- 따라서, Task1에서 의미있는 지표는 client process의 수에 따른 동시처리율의 변화이다.

- **(3) 여러가지 command를 섞었을 경우 - Task 2 :**

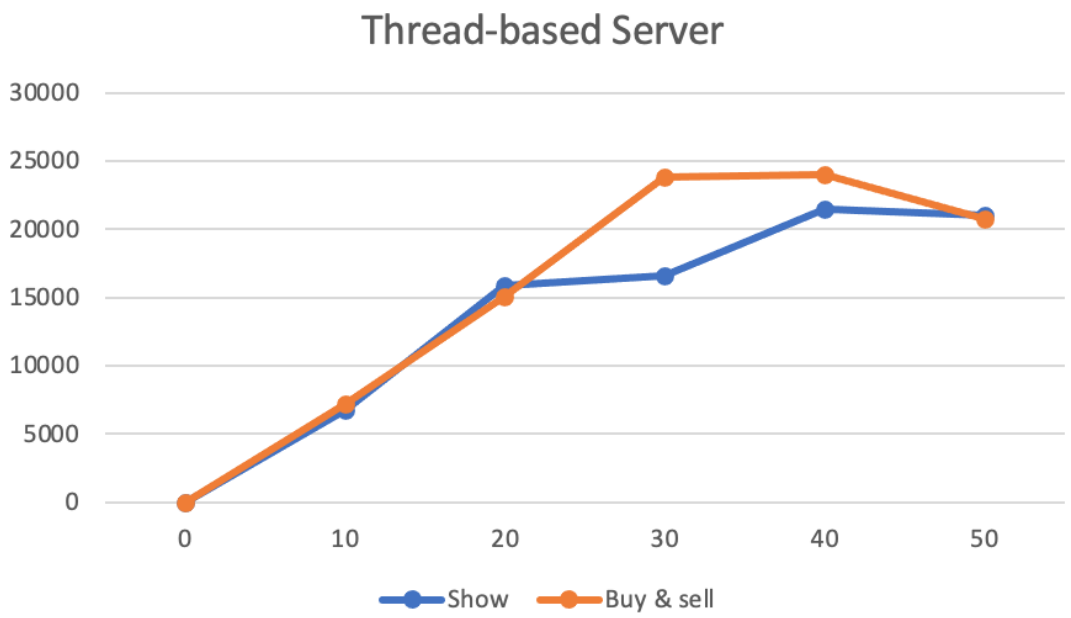
```
Client process : 10
Elapsed time: 0.014471 seconds
Concurrent throughput = 6910.372469

Client process : 10
Elapsed time: 0.014997 seconds
Concurrent throughput = 6668.000267

Client process : 20
Elapsed time: 0.013965 seconds
Concurrent throughput = 14321.518081

Client process : 20
Elapsed time: 0.011448 seconds
Concurrent throughput = 17470.300489

Client process : 30
Elapsed time: 0.017371 seconds
Concurrent throughput = 17270.162915
```

```
Client process : 30
Elapsed time: 0.018921 seconds
Concurrent throughput = 15855.398763

Client process : 40
Elapsed time: 0.019483 seconds
Concurrent throughput = 20530.719088

Client process : 40
Elapsed time: 0.017847 seconds
Concurrent throughput = 22412.730431

Client process : 50
Elapsed time: 0.023231 seconds
Concurrent throughput = 21522.965004

Client process : 50
Elapsed time: 0.024333 seconds
Concurrent throughput = 20548.226688
```

- **(4) Buy, Sell command만 수행하는 경우 - Task 2**

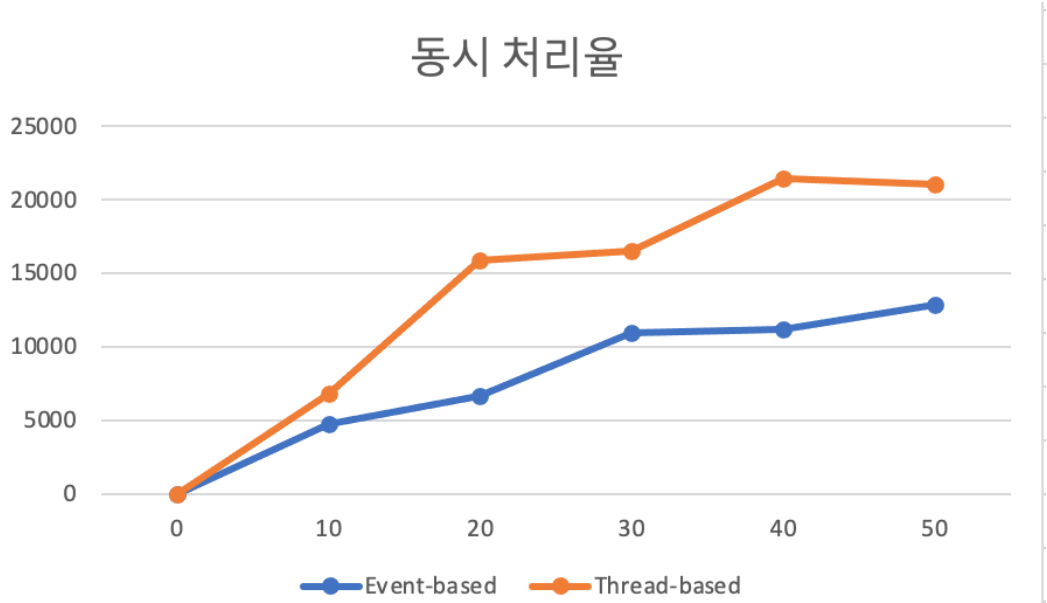
```
Client process : 10
Elapsed time: 0.011435 seconds
Concurrent throughput = 8745.080892

Client process : 10
Elapsed time: 0.017435 seconds
Concurrent throughput = 5735.589332

Client process : 20
Elapsed time: 0.013189 seconds
Concurrent throughput = 15164.151945

Client process : 20
Elapsed time: 0.013279 seconds
Concurrent throughput = 15061.375104

Client process : 30
Elapsed time: 0.011676 seconds
Concurrent throughput = 25693.730730
```

```
Client process : 30
Elapsed time: 0.013659 seconds
Concurrent throughput = 21963.540523

Client process : 40
Elapsed time: 0.016087 seconds
Concurrent throughput = 24864.797663

Client process : 40
Elapsed time: 0.017288 seconds
Concurrent throughput = 23137.436372

Client process : 50
Elapsed time: 0.029883 seconds
Concurrent throughput = 16731.921159

Client process : 50
Elapsed time: 0.020127 seconds
Concurrent throughput = 24842.251702
```




-

- Show command와 Buy&sell command 그룹 모두 30-40개의 client process까지는 동시처리율이 linear하게 증가하는 양상을 보이다, 40개-50개의 process가 들어왔을 때에는 그 기울기가 꺾인 것을 확인할 수 있다. 이는 mutex lock의 사용으로 인한 thread overhead 등의 문제 때문인 것으로 추측된다.
- 그러나, Event-driven server와 마찬가지로 command에 따라서 동시처리율이 변화한다고 보기 어려우며, client process 50개가 되었을 때 이 둘은 모두 2만대의 동시처리율을 보이고 있다.
- 따라서, Task2에서 의미있는 지표는 client process의 수에 따른 동시처리율의 변화이다.
- **(4) Event-driven server와 Thread-based server의 비교**




-

- Event-based 와 Thread-based Server를 무작위 command로 작동시켰을 때에는 2배정도의 동시 처리율 차이를 보였다. 이는 Thread-based server가 multicore processor의 장점을 잘 살린 것으로 해석할 수 있으며, 수업시간에 배운 내용대로 thread-based server가 성능이 더 우수하다는 증거가 된다.