System Programming Project 2

담당 교수 : 김영재

이름 : 조창희

학번 : 20200118

1. 개발 목표

이번 프로젝트에서는 event-driven approach와 thread-based approach를 통한 Concurrent server를 구축하고, 간단한 주식 기능이 구현되도록 하는 것을 목표로 한다. server에서는 주식 정보를 가지고 있는 stock.txt 파일에서 정보를 불러와 이진 트리 형식으로 변수에 저장하고 다음과 같은 요청에 대한 기능을 수행한다.

(1) Buy: buy [주식ID] [구매하고자 하는 주식 개수]의 형식으로 이루어지며, 해당 ID가 없거나 현재 주식 개수보다 더 많은 개수를 구매하고자 할 때 에러 메시지를 출력한다.

(2) Sell: sell [주식ID] [팔고자 하는 주식 개수]의 형식으로 이루어지며, 해당 ID가 없을 경우 에러 메시지를 출력한다.

(3) Show: 현재 주식의 상태를 보여준다.

(4) Exit: 주식장을 퇴장한다.

명령어에 대한 예외 사항은 고려하지 않는다. 또한 Readers-writers problem이 있음을 인지하여 노드 단위의 find-grained locking을 통해 해결할 수 있어야 한다. 프로그램이 종료되면 stock.txt에 지금까지 수정된 값들이 모두 반영되어야 한다. 본 프로젝트를 통해 전반적인 Network Programming에 대한 이해와 구현 방법을 숙지할 수 있도록 한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

먼저 stock.txt 파일에서 정보를 불러와 이진 트리로 저장하고, init_pool값을 통해 poll을 초기화한다. 그러한 다음 무한 루프를 돌면서 active한 상태가 있는지 점검한다. 이를 가능하게 해주는 것은 select함수인데, 이 함수를 통해 여러 파일 디스크립터를 동시에 관찰할 수 있게 해준다. 그러면서 활성화된 파일 디스크립터에 대해 check_clients라는 함수를 통해 주식 관련 명령어를 실행한다. 2개 이상의 서버에서 값이 concurrent하게 적용되며 마지막에 파일에 무사히 반영됨을 확인할 수 있다.

2. Task 2: Thread-based Approach

   전반적인 주식 구현 방식은 1과 유사하나, concurrent server을 구축하기 위해 select가 아닌 thread방식을 사용하였다. 먼저 sbuf를 초기화한 후 미리 선언해 둔 NTHREADS 개수만큼 thread를 생성하는 worker thread pool을 만든다. 그러한 다음 무한 로프를 돌면서 sbuf에 connfd를 삽입하고, thread 함수를 통해 주식 요청을 반영한다. 이를 통해 1번과 같이 동시에 여러 서버에 대한 요청사항을 수행하며 최종적으로 stock.txt에 반영되는 것을 확인하였다.

3. Task 3: Performance Evaluation

   결과적으로는 예상과 달리 두 방법 모두 show 명령어만을 수행하였을 때 가장 뛰어난 동시 처리율을 보였다. 모든 명렁어와 buy + sell 명령어를 수행하였을 때는 event 기반 서버가 더 뛰어난 반면 show 명령어에서는 thread 기반 서버가 더 뛰어난 동시 처리율을 보였다. 따라서 show 명령어의 비중이 클 경우에는 thread 방법을, 그 외 경우에는 event 기반 방법을 사용하는 것이 더 효과적일 것이라고 예상된다. 자세한 분석은 3번에서 작성하였다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  I/O Multiplexing이란 하나의 통신 채널을 통해 두 개 이상의 데이터를 처리하는 데에 사용된다. 이를 통해 event 기반의 Concurrent server을 구현할 수 있다. I/O Multiplexing을 사용하는 것에서는 select함수를 이용하는 것이 핵심이다. 위에서 말하였듯 select 함수는 여러 파일 디스크립터를 한번에 관찰하여 pending input이 있는지 관찰할 수 있다. 이를 통해 listenfd에 pending input이 있다면 accept함수를 통해 새로운 fd를 만들어 array에 추가해주고(add_clinet), connfd에 pending input이 있을 경우 읽어서 요처에 맞는 동작을 수행한다. (check_clients) 이러한 과정을 계속해서 반복한다. I/O Multiplexing은 하나의 process(or thread)를 통해 concurrent한 server을 구현할 수 있다는 장점이 있다.

  ✓ epoll과의 차이점 서술

selext함수는 concurrent server을 가능하게 해주는 함수이지만 항상 모든 파일 디스크립터에 대한 반복문을 진행하기에 효율이 떨어진다는 단점이 있었다. epoll은 이러한 select의 단점을 보완하여 리눅스 환경에서 사용하도록 만들어진 I/O 통신 기법이다. epoll은 운영체제에게 관찰 대상에 대한 정보를 한번만 알려주고 변경이 있을 때 변경사항만 알려주는 방식이다. 따라서 select와 같은 반복문이 필요 없다. 기본적인 구조는 select와 유사하다.

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

  ✓ Master Thread의 Connection 관리

  본 코드에서는 Master Thread를 sbuf_t sbuf로 선언하여 connection을 관리한다. 우선 sbuf_int으로 sbuf를 초기화한 다음 for문으로 worker thread pool을 생성하고, 무한 루프를 돌면서 sbuf에 connfd를 삽입하는 형식이다. 각각 생성된 thread 안에서는 기존 sbuf를 remove해서 connfd에 저장하고, 이에 대해 execFunf을 실행시켜 주식 관련 요청을 수행하며 concurrent server을 구축한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

- Thread pool이란 스레드를 그때그때 생성하는 것이 아닌, 제한된 개수의 스레드를 정해놓고 생성한 다음 큐에 들어오는 작업들을 하나씩 스레드가 맡아 처리하는 것을 의미한다. Memory leak를 방지하기 위해 각 thread를 실행할 때 마다 Pthread_detach를 디폴트 값으로 실행시켜준다.

- **Task3 (Performance Evaluation)**

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

- 동시 처리율은 다음과 같이 정의된다.

- client 개수 / 시간

- 위와 같이 정의한 이유는 시간당 클라이언트의 처리 개수가 곧 동시 처리율이기 때문이다. 시간 측정은 다음과 같이 진행하였다. clock함수를 통해 함수 multiclient.c에서 시작부터 끝까지 시간을 측정하고, CLOCKS_PER_SEC으로 나누어 elapsed time을 계산했다. ORDER_PER_CLIENT은 10으로 고정하였으며 client의 개수를 늘려가며 테스트 하였다. 그리고 show 명령어만 수행하였을 때, buy 명령어만 수행하였을 때, sell 명령어만 수행하였을 때, 세 명령어를 섞어서 수행하였을

때 결과를 확인하였다.

✓ Configuration 변화에 따른 예상 결과 서술

- Buy와 sell은 이진트리를 모두 탐색할 필요가 없이 같은 id를 찾을 때까지만 수행하면 되는 반면에, show 명령어는 이진트리 끝까지 탐색해야 하므로 show 명령어만을 실행시켰을 때 동시 처리율이 낮아질 것이라고 예상하였다. 또한 client가 많아질수록 동시 처리율이 낮아질 것이라고 예상하였다.

C. 개발 방법

- Task_1

  이진 트리를 구현하기 위해 구조체인 node struct를 추가하였다. 이 안에 파일을 읽을 정보를 저장하는 변수들과, left, right를 가리키는 노드를 선언하였다. 또한 newNode, insertNode 또한 이진 트리를 구현하기 위한 함수들로 추가하였다.

  그리고 pool 자료구조를 추기화해주는 init_pool, client를 추가해주는 add_client, client를 종료해주는 close_client 함수들은 수업 자료를 참고하여 추가하였다.

  또한 주식 요청을 수행해주는 핵심적인 함수인 check_clients를 추가하였다. 여기서는 p->nready가 0이 될 때까지 반복문을 수행하면서, sell, buy, show, exit 에 대한 각 명령을 수행하였다. 여기서 핵심은 각 텍스트 라인을 알맞게 parsing하는 것인데, 이를 위해 임시변수인 buf을 정의하고, 자체적인 함수인 makeStr을 선언하였다. makeStr 함수 안에서는 int 최댓값을 고려하여 11의 크기로 임시 배열을 선언하여, sprintf함수를 이용하여 parsing하였다. 그리고 memset함수와 sscanf함수를 이용하여 처음 구현한 node 구조체의 값과 비교할 수 있도록 구현하였다. 또한, 이중 같은 id를 찾아내야 하는 findId 함수를 추가하여 인자로 들어온 노드 값을 순회하면서 같은 id값을 찾아내는 기능을 구현하였다.

  또한 sig_handler 함수를 통해 CTRL + c 입력이 들어왔을 때 종료되기 전에 stock.txt에 수정된 값을 반영하도록 하였다. 그 외에 saveStr 함수에서도 같은 기능을 구현하도록 하였다.

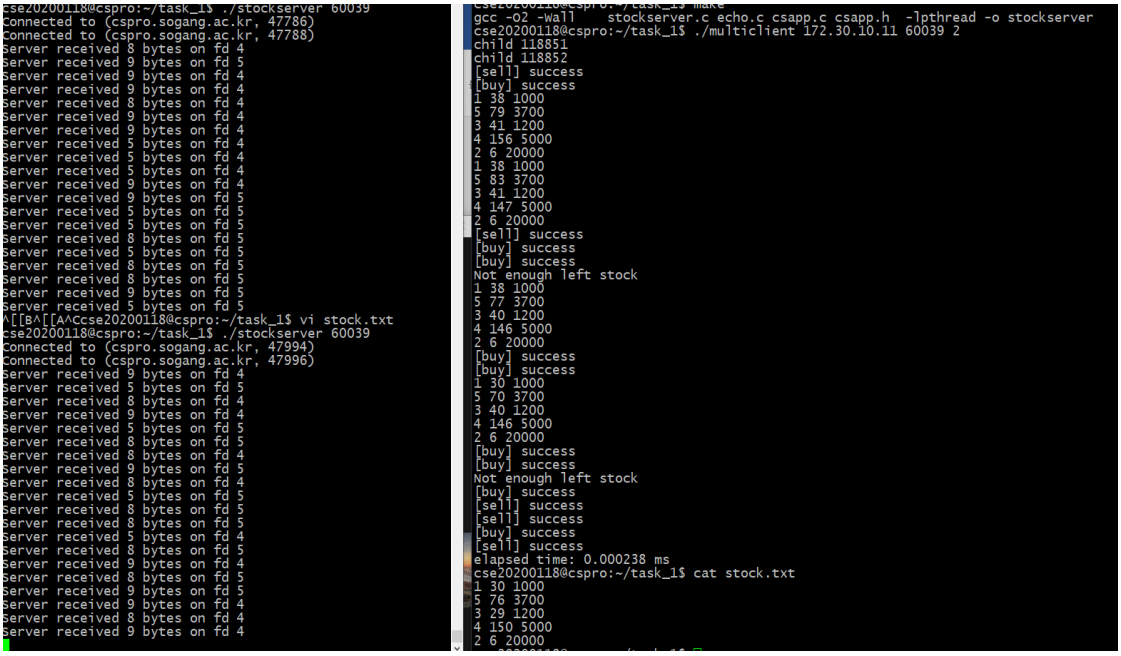
- Task_2

- sbuf_t 구조체를 추가하여 thread들을 관리할 수 있는 변수들을 선언하였다. sbuf와 관련된 기초함수들을 강의자료를 참고하여 추가하였다. 또한 task_1에 있던 node 구조체에 set_t mutex와 set_t w 를 추가하여 node 단위로 fine-grained locking을 수행하도록 했다. 이것은 execFunc에서 buy 명령어와 sell 명령어를 실행시킬 때, 자료의 값이 직접적으로 반영되기 때문에 thread safety를 위해 P와 V 함수를 이용하여 locking을 수행하도록 했다. 이 외에도 파일에 직접적으로 write 해야 하는 경우는 모두 P와 V 함수로 safety가 손상되지 않도록 구현하였다. 또한 thread 함수를 추가하여 이 안에 reaping이 가능하도록 pthread_detach함수를 호출하였고 이 안에서 기존 check_clients를 execFunc으로 이름을 변경하여 호출하였다.

3. 구현 결과

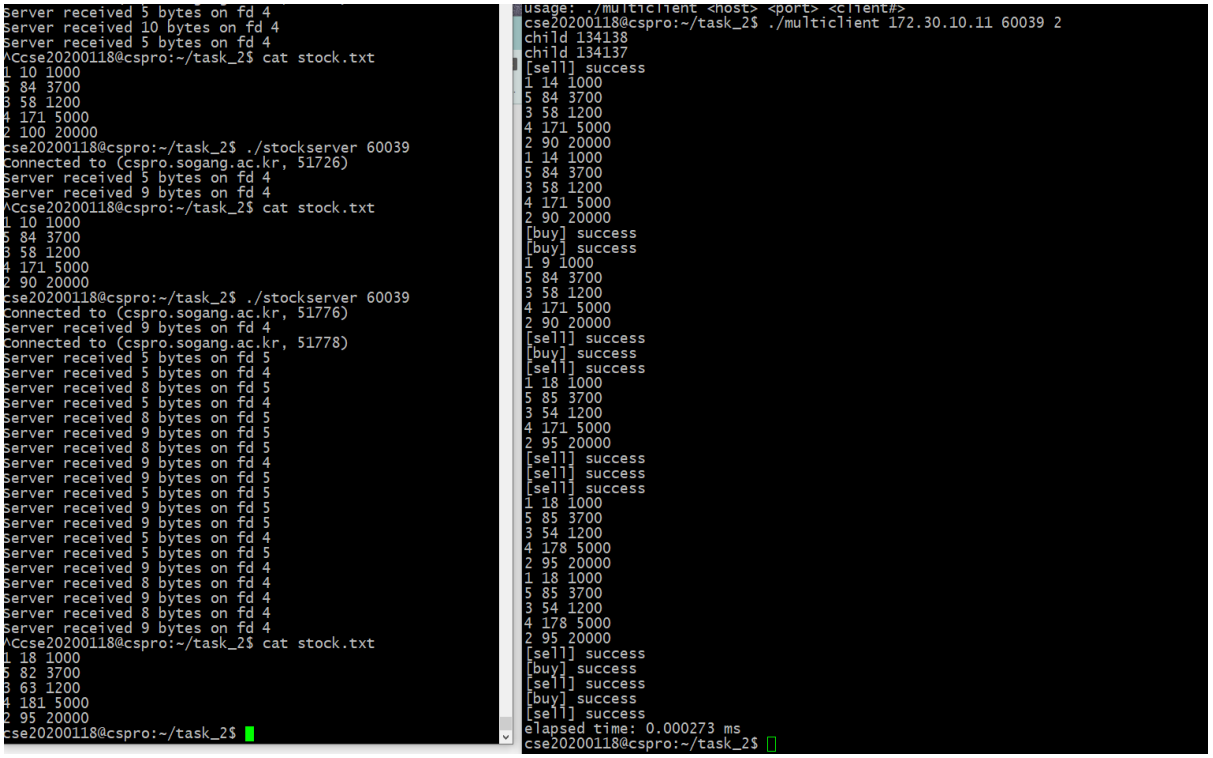
1. Task_1

stock.txt 파일을 읽으면서 이진 트리 값에 노드 형태로 저장한다. Listenfd를 열고 pool을 초기화 한 다음, 무한 루프 안에서 select와 accept 함수를 통해 add_client와 check_client를 호출하는 방식으로 concurrent server을 구현하였다. 마지막에 최종적으로 변경된 값을 다시 stock.txt에 반영한다. 이러한 결과를 다음과 같이 확인할 수 있다.



2. Task_2

stock.txt 파일을 읽으면서 이진 트리 값에 노드 형태로 저장한다. Listenfd를 열고 sbuf를 초기화한다음, 각 thread를 생성한다. 그러한 다음 무한루프를 돌면서 sbuf_insert를 호출하고 이에 대한 서비스를 제공한다. 마지막에 최종적으로 변경된 값을 다시 stock.txt에 반영한다. 이러한 결과를 다음과 같이 확인할 수 있다.



- 딱히 구현하지 못한 부분은 없으나, 처음에는 client만 종료했을 때 stock.txt에 변한 값이 반영되지 않아 혼란이 왔었다. server까지 종료를 해주어야 반영된다는 사실을 알게 되었다. 또한, task_2에서 buy, sell, show 외 명령어를 입력하였을 때 Rio_readlineb error 에러가 뜨는데 이에 대한 해결은 하지 못하였다. 그러나 구현 조건 외에 속하고, 종료 후 반영이 잘 되는 것을 확인하였다.

4. 성능 평가 결과 (Task 3)

- 실험 환경: 서버-cspro, 클라이언트-cspro, 포트-60039

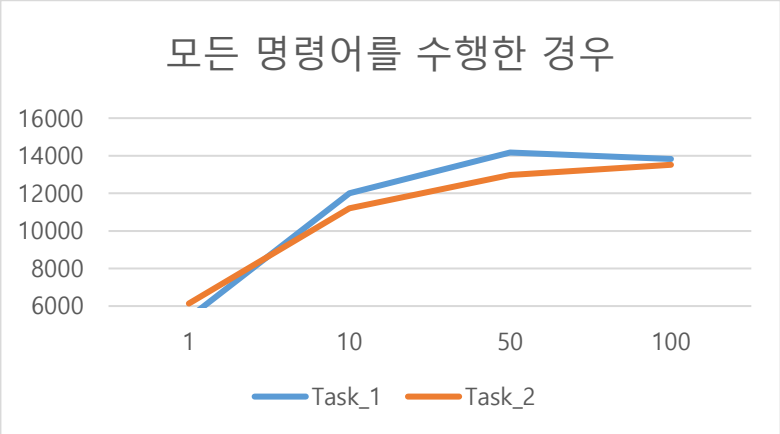
  Client 개수 외에 다른 값은 다음과 같이 고정한다.

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

  Client 개수 변화에 따른 elapse time 변화를 측정하고, 똑같이 변화를 주면서 워크로드를 모든 명령어를 혼합한 경우, show만 요청하는 경우, buy 혹은 sell을 요청한 경우 총 3가지 경우를 각각 변환시켜보며 확인해 볼 것이다. 이는 확장성과 워크로드의 변화에 따른 분석을 하기 위해서다. Buy와 show는 비슷한 구조를 가지고 있기에 한 경우로 묶어서 분석하였다. 가로 그래프는 client 개수를, 세로 그

래프는 동시 처리율을 의미한다. 분석 결과는 다음과 같다.

(1) 모든 명령어를 혼합하여 요청할 경우

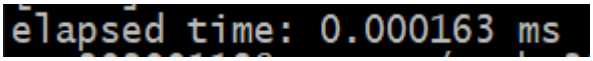


- 미세한 차이지만 전반적으로 task_1이 task_2보다 더 뛰어난 동시 처리율을 보였다. event기반이 하나의 thread 안에서 운영되므로 그럴 것이라고 추측된다. 두 task 모두 1에서 10까지는 큰 기울기의 상승성을 갖다가 점차 기울기가 완만해짐을 볼 수 있다.

Client의 개수 : 1개

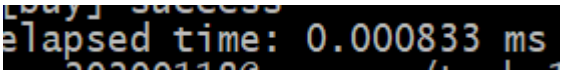
- Task_1

- Task_2

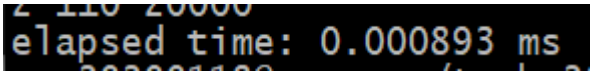


Client의 개수: 10개

- Task_1



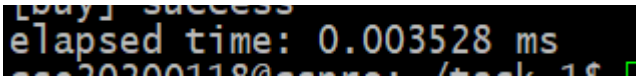
- Task_2



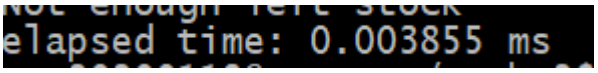
-

- Client의 개수: 50개

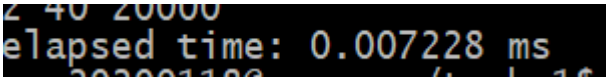
- Task_1



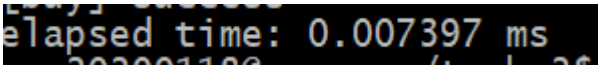
- Task_2



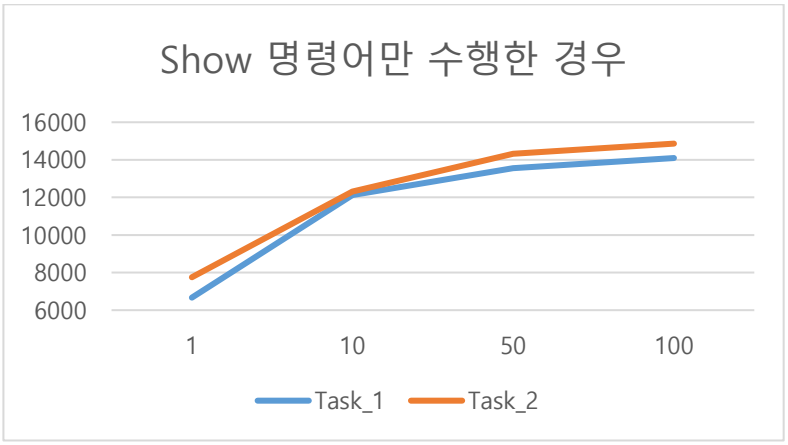
- Client의 개수: 100개
- Task_1



- Task_2



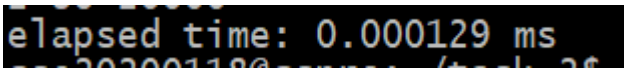
(2) Show 명령어만 수행할 경우

- 

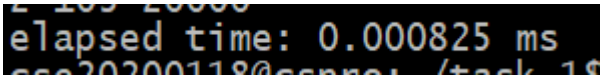
미세하지만 Task_2가 Task_1보다 더 높은 동시처리율을 보였다. 또한 1부터 100까지 기준으로는 client 수가 커질수록 동시 처리율이 점점 커지는 모습을 보여준다.

- Client의 개수 1개
- Task_1

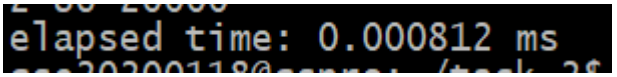
- Task_2



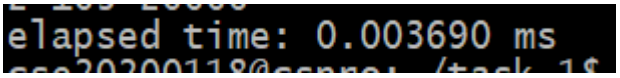
- Client의 개수 10개
- Task_1



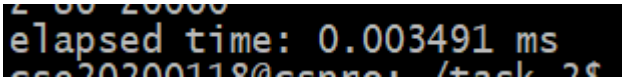
- Task_2



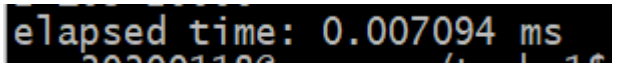
- Client의 개수 50개
- Task_1

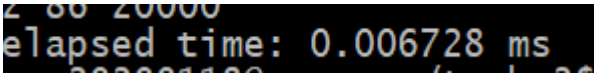


- Task_2

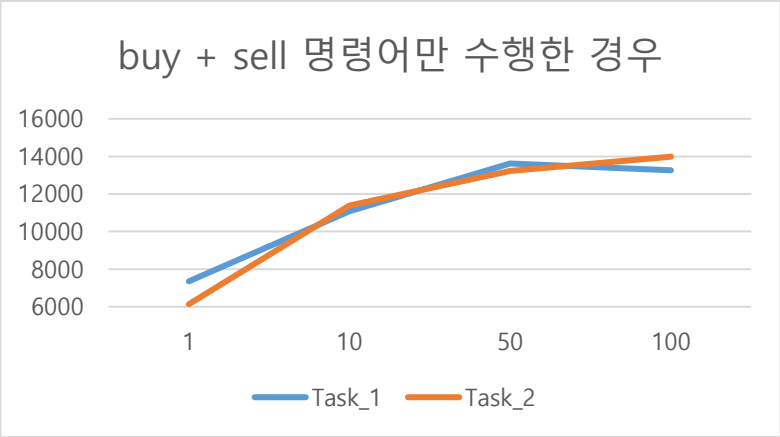


- Client의 개수 100개
- Task_1



- Task_2
- 

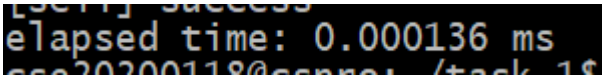
(3) buy + sell 명령어만 수행할 경우



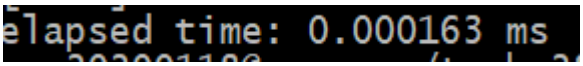
- client 수와 상관 없이 서로 교차되는 모습을 보인다. 초반에는 task_1이 더 뛰어난 동시처리율을, 후반에는 task_2가 더 뛰어난 동시처리율을 보인다.

  Client의 개수 1개

  - Task_1

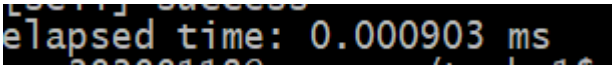


  - Task_2

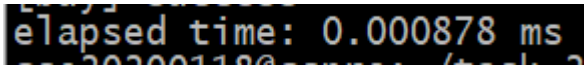


  - Client의 개수 10개

  - Task_1

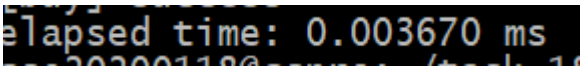


  - Task_2

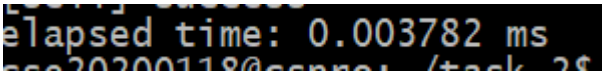


  - Client의 개수 50개

  - Task_1



  - Task_2



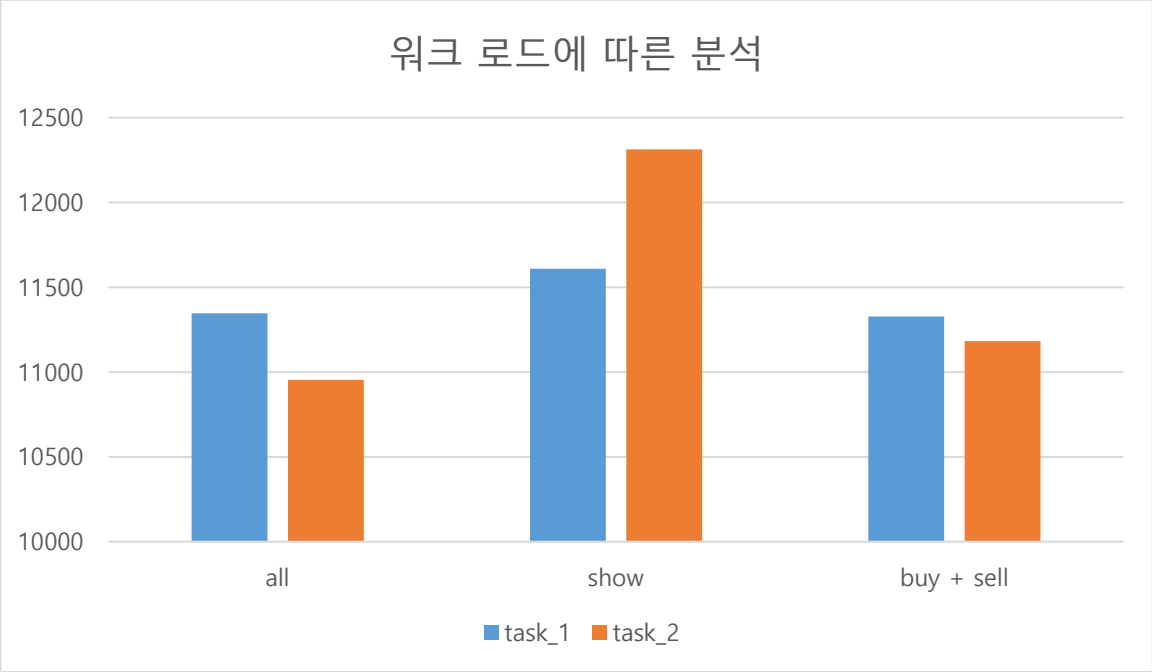
- Client의 개수 100개
- Task_1

```
elapsed time: 0.007541 ms
```

- Task_2

```
elapsed time: 0.007150 ms
```

- 워크로드에 따른 분석



- 결과적으로는 예상과 달리 두 방법 모두 show 명령어만을 수행하였을 때 가장 뛰어난 동시 처리율을 보였다. 모든 명령어와 buy + sell 명령어를 수행하였을 때는 event 기반 서버가 더 뛰어난 반면 show 명령어에서는 thread 기반 서버가 더 뛰어난 동시 처리율을 보였다. 따라서 show 명령어의 비중이 클 경우에는 thread 방법을, 그 외 경우에는 event 기반 방법을 사용하는 것이 더 효과적일 것이라고 예상된다.