Multicore Programming Project 2

담당 교수 : 박성용

이름 : 이진용

학번 : 20191630

1. 개발 목표

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

동시에 여러명의 client의 request에 대해 response가 가능한 주식서버를 구현한다. 동시성을 구현하기 위해 multi processes, event-driven, multi thread 중 event-driven, multi thread 기반 서버를 별도로 구현한다.

해당 서버는 client의 주식 조회, 구매, 판매의 request를 수행하고 이에 대한 response를 수행한다. 위에 언급한대로 여러 client들의 동시적인 request들을 atomic하게 수행하게 구현한다. 서버 실행 시, 주식 데이터는 stock.txt파일에서 읽어와 실행하는 동안 binary tree 형태로 메모리에 저장하고 서버 종료 시, 다시 stock.txt 파일에 저장한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

Select를 이용하여 다수의 request들을 동시에 처리한다. 이것을 구현하기 위하여 file descriptor의 변경사항 감지하여 새로운 client의 접속을 확인하여 accept하여 새로운 FD를 기존에 select를 통하여 확인하던 fd_set에 추가해준다.

client으로부터의 요청은 마찬가지로 select를 통하여 client socket의 변화를 감지하여 해당 요청을 현재 메모리상의 주식 상태를 기반으로 처리하여 response를 보내준다.

주식 상태는 서버 프로그램 시작 시 stock.txt파일로부터 읽어와 binary search tree상에 저장한다. 이 후 client 에 요청에 따라 해당 주식정보를 다시 string으로 변환하여 사용자에게 전송하고 buy, sell 요청을 처리하고 메모리상에 해당 변경사항을 반영한다. 최종적으로 프로그램을 종료 시에 메모리상의 데이터를 다시 파일에 저장 후 종료한다.

2. Task 2: Thread-based Approach

   task1에서 구현했던 event-driven 기반 서버를 multi thread 기반 서버로 다시 구현한다. 기존의 stock 정보를 저장하던 자료구조의 형태와 로직을 그대로 유지한다. 프로그램 시작시 thread pool을 미리 만들어 놓고 client가 연결되면 thread에서 해당 client의 request를 처리해준다. 이 때 해당 쓰레드는 detach하여 다른 쓰레드들과 독립적으로 해당 작업을 처리한 뒤 종료하게 구현한다.

   thread끼리 주식 정보를 담은 자료구조 외에 공유하는 resource는 없게 구현한다. 다만 주식정보는 모든 thread에서 공유하고 동시에 read, write 시 race condition이 발생하기에, 특정 주식 노드를 update 시에 semaphore lock을 걸어주어 해당 주식에 access할 때, 해당 쓰레드를 기다리게 한다.

3. Task 3: Performance Evaluation

   task1, task2에서 구현했던 두 가지 종류의 서버를 같은 조건에서 실험해본다. 다수의 client가 request를 보낼 때 어느정도의 처리 속도가 나오는지를 확인해보기 위하여 client의 개수와, 요청하는 request의 수, 요청 delay 간격, 요청의 종류에 따른 변화를 관찰한다.

   해당 실험을 진행하기 위해 제공된 test code 즉 multi-client 코드를 변형하여 request type과 대기 속도를 조절하여 같은 조건으로 실험을 진행한다. 실험을 통해 얻은 결과를 각각의 서버의 특징을 통해 분석해본다.

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

  다중의 client들이 서버로 연결하거나, 연결된 소켓을 통해 데이터를 전송시

서버사이드에서는 select 함수를 통해 해당 listen socket과 accepted socket들의 변화를 감지한다.

listen socket이 select 함수에 걸린다면 새로운 client의 연결을 의미하기 때문에 해당 연결을 accept해주고 연결된 socket을 감지할 fd_set에 추가해주고 다시 select함수를 통해 대기하도록 한다.

반대로 accept socket의 변화가 감지된 것은 client으로부터 request가 도착한 것을 의미한다. 해당 요청이 감지되어 select가 빠져나가면 감지된 fd_set들을 통하여 해당 소켓으로부터 데이터를 read하고 요청에 대한 response을 생성하고 그대로 write해주게 된다.

동시에 다수의 fd들의 select될 수 있는데 이 때 몇개의 fd들이 감지되었는지 확인한 후 request들을 처리해줄 때 for문을 통해 감지된 socket들에가만 response을 해주게 된다.

✔ epoll과의 차이점 서술

epoll은 select 보완하여 대체하고자 나온 것이다. select에서는 user level에서 fd_set을 직접 관리하며 select된 fd_set들도 모두 확인하며 변경된 fd들을 찾아야 하지만, epoll은 해당 fd_set을 kernel level에서 관리하고 또 감지된 fd들만 반환해준다. 그래서 반환된 fd_set을 다시 복사해 넣어줄 필요가 없다. 또한 select fd_set의 최대 개수는 1024로 정해져있지만 epoll은 해당 제한이 없다.

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

✔ Master Thread의 Connection 관리

Master Thread에서는 루프문 안에서 연결된 client의 소켓을 sbuf안에 넣어주는 역할을 진행한다. 이때 sbuf는 counting semaphore를 사용하여 connfd를 가능한 slot에 저장하게 되고 thread pool 중 하나의 thread에서 sbuf으로 부터 connfd를 받아와 request 처리를 담당한다.

✔ Worker Thread Pool 관리하는 부분에 대해 서술

위에서 master thread에서 worker thread로 connfd를 넘겨주었다. 이를 구현하기 위해서는 master thread에서 socket listen 전에 thread pool을 미리

생성해준다. 해당 worker thread에서는 sbuf에서 connfd가 들어올 때까지 대기하고 있는다. 연결이 종료 된 이후에 해당 socket을 close해주고 다시 위의 상태로 돌아가 이 과정을 반복하게 된다.

- **Task3 (Performance Evaluation)**

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

여러 가지 방식으로 구현한 서버들을 client의 수, order type에 따른 동시 처리율 즉 처리 시간을 측정한다. 동시에 client들이 서버에 동일한 수의 request를 보내어 처음 request를 보낼 때부터 마지막 response를 받은 시간을 elapse time으로 측정한다. 이때 gettimeofday 함수를 사용하여 micro sec 단위로 시간을 측정한다. 또한 stdout 출력을 제한하여 불필요한 IO를 줄여 실험의 오차를 줄이도록 한다.

event-driven, thread-pool 서버 두개 외에 연결 시마다 thread를 create 하고 readers write lock를 고려하지 않은 multi-thread 서버의 성능도 같이 측정하여 총 3개의 서버의 성능을 측정해본다.

1. client numbers

   다른 변인들을 통제하고 client 수의 변화에 따른 서버의 elapse time을 측정한다. order type은 random하게 지정하고 client_per_orders 를 100으로 지정하여 어느정도 시간을 길게잡아 데이터간의 유의미한 차이가 나도록 설정한다.

2. order type

   request type 즉 show, buy, sell 중 하나만 보내도록 설정하여 서버 별로 request의 elapse time 차이가 나는지 확인한다. 1번 실험과 마찬가지로 외에 다른 변인들을 모두 통제하여 진행한다.

✔ Configuration 변화에 따른 예상 결과 서술

먼저 client numbers에 따른 성능 비교이다. event-driven 서버는 single process, single thread 서버이다 보니 cpu core를 하나밖에 사용하지 못한다. 반면에 multi-thread , thread-pool 서버는 각각의 thread들을

멀티코어 환경에서는 event-driven 서버보다 너 빠른 처리속도를 보여줄 것으로 기대한다. 또한 연결 시 마다 thread를 create해주고 readers lock를 고려하지 않은 multi-thread 서버가 thread-pool 서버보다 더 낮은 처리속도를 보여줄 것으로 예상할 수 있다.

두번째로 order type에 따른 성능비교이다. 모든 서버의 데이터베이스를 binary search tree로 구현했기 때문에 하나의 주식의 접근(buy,sell)에 대한 시간 복잡도는 O(logn) 이고 모든 node에 대한 접근은 O(n)이다. 때문에 단순히 노드에 접근하여 데이터를 read, write한다면 show의 대한 elapse time이 길 것으로 예상되고 서버끼리의 성능은 thread-pool > multi-thread > select 으로 예상된다. 다만 semaphore lock으로 인해 어느정도 multi thread 환경에서 성능 저하가 예상된다.

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

  해당 방법을 구현하기 위해 감지할 fd_set과 client fd들을 저장할 pool 구조체를 선언한다. 해당 구조체 내에는 max fd, max index, nready 등을 추가로 저장하여 client의 request을 처리할 때 최소한의 overhead가 발생하도록 구현한다. nready에는 select의 return 값을 저장하여 감지된 fd들만 처리하게 한다. client가 연결되고 접속 종료될 때마다 maxi, maxfd도 업데이트 해주며 모든 fd_set을 탐색하는데 overhead를 줄여준다.

  Select 함수 다음에 FD_ISSET 매크로를 사용하여 listen fd가 감지되었나 확인 후 client를 추가해주고 남은 fd_set들을 확인하며 client로부터의 request들을 처리해준다.

  데이터베이스에 저장할 주식정보는 주식 하나당 하나의 노드로 binary search tree 형태로 구현한다. 왼쪽의 child node엔 root보다 작은 id를, 오른쪽 child node엔 큰 id를 저장하여 하나의 주식에 조회할 때, O(logn)안에 조회가 가능하게 한다. 서버 실행 시 stock.txt로부터 해당 노드들을 insert 하여 초기화 해준다. 이후 client의 요청에 따라 show에는 모든 노드들을

dfs로 탐색하여 id order 순으로 상태를 전송하고 sell, buy는 하나의 주식을 search하여 해당 잔여주식량을 업데이트 후 결과를 전송한다.

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

  multi-thread pool을 구현하여 client가 연결할 때마다 thread를 생성하는 것이 아니라 서버 실행 시 미리 만들어두었던 thread pool중 하나에서 sbuf로부터 connfd를 가져와 해당 연결에 대한 request들을 처리하고 연결이 종료될 때, socket을 close하고 대기상태로 돌아간다.

  sbuf 구조체에 fd들을 저장할 buf를 저장하고 해당 buf에 동시에 접근을 막는 mutex와 남아있는 slots을 저장할 counting semaphore, 들어있는 items의 접근을 확인할 counting semaphore을 선언한다. client 연결시 sbuf insert하여 thread pool 중 하나의 thread의 items semaphore의 lock이 풀리면서 buf에서 connfd를 가져오게 된다. master thread에서는 연결된 client들의 socket을 단순히 sbuf에 넣어주는 역할만 해주고 worker thread에서는 sbuf에서 connfd를 꺼내온다.

  task1에서 구현했던 데이터베이스 endpoint 코드는 그대로 사용하고 다만 multi thread 환경에서 작동하도록 semaphore lock을 추가로 구현한다. show 명령어는 단순히 주식에 접근하여 데이터를 read하니 해당 노드에 read에 대한 semaphore lock 변수를 추가하여 해당 노드에 여러 thread들이 read가 가능하게 하고 다만 write만 불가하도록 lock을 걸어준다. 마지막 thread가 read가 끝났을 때 write lock을 풀어주어 다른 thead가 해당 노드에 write 가능하게 한다. 마찬가지로 write 시에 다른 thread가 write불가능하게 write lock을 걸어주어 최종적으로 race condition이 발생하지 않게 한다.

- **Task3 (Performance Evaluation)**

  성능을 평가해볼 서버는 총 3개로 event-driven, multi-thread, thread-pool 서버이다. 해당 서버를 client 수의 변화, order type에 따른 변화를 관찰한다. 해당 과정을 측정하기 위해 제공된 테스트 코드를 수정한다. multiclient 프로세스 당 모든 order 처리 후 시간을 측정한다. 측정에 정확성을 올리기 위해 client에서 stdout에 출력하지 않는다. 시간 측정이 완료되면 해당 실험군의 평균을 통하여 데이터를 집계한다.

3. 구현 결과

- 2번의 구현 결과를 간략하게 작성
- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

event-driven concurrent을 구현하였다. select함수를 통하여 동시에 여러 client의 요청을 받아 해당 요청들을 sequential하게 처리해주어 concurrent하게 처리된다. 하지만 동시에 여러 request들을 처리할 때, 요청을 보낸 순서대로가 아닌 해당 socket의 fd들이 저장된 순서대로 해당 요청을 처리하다보니 완벽한 concurrent한 서버라고 보기엔 힘들다. 또한 fd들을 탐색하는 과정에서 생기는 overhead는 서버의 성능저하로 이어진다.

multi-thread pool 기반 서버를 구현하였다. 해당 서버는 thread pool을 미리 만들어주고 sbuf 구조체를 통하여 모든 worker thread들이 client가 연결되기 전에 buf에서 connfd를 가져올 때까지 wait상태로 기다린 후 이는 semaphore lock을 통하여 하나의 thread에게만 socket fd를 전달하도록 구현했다. 이는 client가 연결될 때마다 새로운 thread를 생성해 해당 요청을 처리하는 서버보다 thread를 생성하는 cost를 줄여 더 나은 성능향상을 기대할 수 있다. 이는 아래 성능 평가 결과에서 추후 서술할 예정이다. 또한 여러 thread에서 하나의 주식에 동시에 read 가능하고 write만 막는 lock을 구현하여 불필요한 lock을 줄여 성능향상을 기대할 수 있다.

주식정보를 저장하기 위해 binary search tree를 구현하여 해당 노드에 주식의 정보를 저장하였다. 다만 주식 id가 insert 시, order되어 있다면 bst의 worst case인 한 줄로 노드들이 생성되는 케이스가 발생할 수 있다. 이렇게 되면 기대했던 select의 시간 복잡도가 O(logn)이 아닌 O(n)으로 되어버려 linear search와 크게 차이가 나지 않게 된다. 추가적인 성능 개선을 위해 실제 데이터베이스에 적용는 자료구조인 b+ tree 구조를 적용해볼 수 있을 것으로 기대한다.

4. 성능 평가 결과 **(Task 3)**

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

실험 환경

- OS : ubuntu 16.04.6
- CPU : Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz
- GCC : 9.4.0

cspro 환경이 다중 이용자가 사용하다보니 동일한 조건에도 상이한 결과가 나와 성능평가만 다른 서버를 이용하여 수행했다. 해당 서버의 코어는 48개로 multi thread 환경에서 더욱 유의미한 결과를 보여줄 것으로 기대한다.

실험군

- event-driven
- multi-thread
- thread pool

실험을 진행할 서버는 위의 세개로, 첫번째와 세번째는 task1, task2에서 구현한 서버이고 추가로 연결시마다 thread를 create하고 read, write모두 동일하게 lock을 거는 서버를 같은 조건에서 비교해본다.

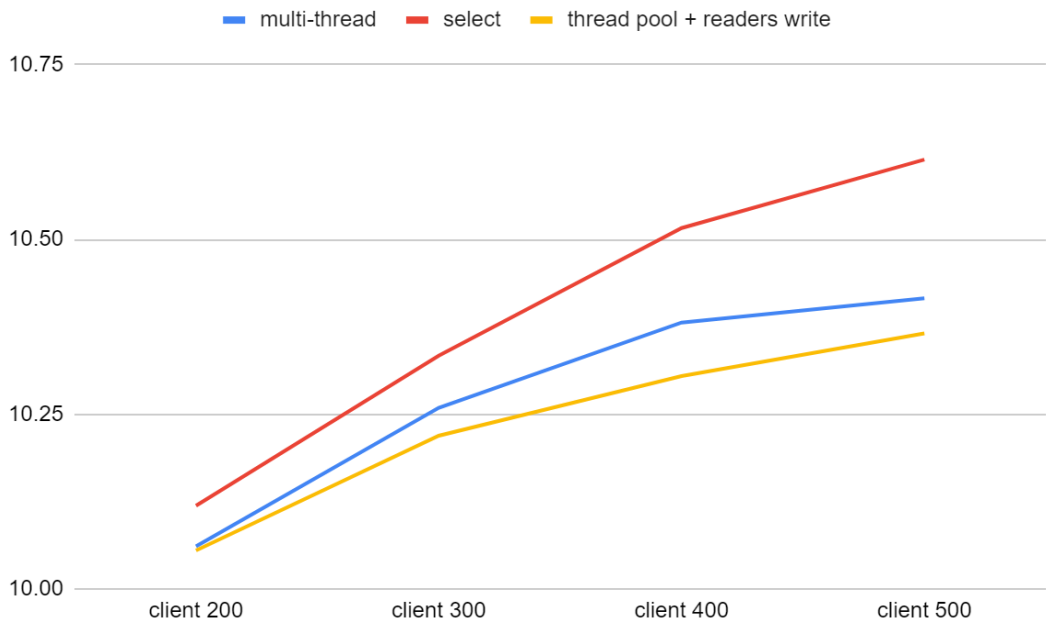
1. **client**의 변화에 따른 성능 비교

해당 실험은 client가 변화함에 따라 즉 동시에 여러 request가 들어옴에 따라 어느정도의 성능이 나오는지 비교하였다. 측정방법은 위에 서술한 방법대로 client에서 측정해본다. 이때 client당 order은 100개 order type은 random으로 진행한다. 결과는 아래와 같다.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	multi thread					select					thread pool			
2	client 200	client 300	client 400	client 500		client 200	client 300	client 400	client 500		client 200	client 300	client 400	client 500
3	10.031	10.03	10.031	10.03		10.049	10.048	10.08	10.068		10.03	10.032	10.028	10.03
4	10.032	10.03	10.031	10.03		10.049	10.048	10.08	10.068		10.03	10.032	10.03	10.03
5	10.032	10.03	10.031	10.03		10.049	10.049	10.08	10.068		10.03	10.032	10.03	10.03
6	10.032	10.03	10.031	10.031		10.049	10.049	10.081	10.07		10.03	10.032	10.03	10.03
7	10.032	10.03	10.031	10.032		10.049	10.049	10.081	10.07		10.03	10.032	10.03	10.03
8	10.032	10.03	10.031	10.032		10.049	10.049	10.081	10.07		10.031	10.032	10.03	10.03
9	10.032	10.03	10.031	10.032		10.049	10.049	10.081	10.07		10.031	10.032	10.03	10.03
10	10.032	10.03	10.031	10.032		10.049	10.049	10.081	10.07		10.032	10.032	10.03	10.03
11	10.034	10.03	10.032	10.034		10.049	10.049	10.081	10.07		10.032	10.032	10.03	10.032
12	10.035	10.03	10.032	10.034		10.049	10.049	10.081	10.071		10.032	10.032	10.03	10.032
13	10.035	10.03	10.032	10.034		10.049	10.049	10.081	10.071		10.033	10.032	10.031	10.033
14	10.035	10.03	10.033	10.035		10.049	10.049	10.081	10.08		10.035	10.032	10.031	10.033
15	10.035	10.03	10.035	10.035		10.049	10.051	10.081	10.08		10.035	10.032	10.031	10.033
16	10.035	10.032	10.035	10.035		10.049	10.051	10.082	10.08		10.035	10.032	10.031	10.033
17	10.035	10.032	10.035	10.035		10.049	10.051	10.082	10.08		10.035	10.032	10.031	10.033
18	10.035	10.032	10.036	10.035		10.049	10.052	10.082	10.08		10.035	10.032	10.032	10.033
19	10.035	10.034	10.036	10.035		10.049	10.052	10.082	10.08		10.035	10.033	10.032	10.033
20	10.035	10.035	10.036	10.035		10.049	10.052	10.083	10.08		10.036	10.033	10.032	10.034
21	10.035	10.035	10.036	10.036		10.049	10.052	10.083	10.08		10.036	10.033	10.032	10.034
22	10.035	10.035	10.037	10.036		10.05	10.053	10.084	10.08		10.036	10.034	10.032	10.034
23	10.035	10.035	10.039	10.036		10.05	10.053	10.085	10.08		10.036	10.034	10.033	10.034
24	10.035	10.036	10.039	10.036		10.05	10.053	10.085	10.08		10.036	10.034	10.033	10.034
25	10.036	10.036	10.039	10.036		10.05	10.053	10.085	10.08		10.036	10.034	10.033	10.034
26	10.036	10.039	10.039	10.036		10.05	10.053	10.085	10.08		10.036	10.034	10.033	10.034
27	10.036	10.039	10.039	10.036		10.05	10.056	10.085	10.081		10.036	10.034	10.033	10.034
28	10.036	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.034	10.033	10.034
29	10.036	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.034	10.033	10.035
30	10.036	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.036	10.034	10.035
31	10.036	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.037	10.035	10.035
32	10.036	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.037	10.035	10.036
33	10.038	10.039	10.039	10.038		10.05	10.056	10.085	10.081		10.037	10.037	10.037	10.036
34	10.038	10.039	10.04	10.038		10.054	10.056	10.086	10.081		10.037	10.037	10.04	10.036
35	10.038	10.039	10.04	10.039		10.054	10.056	10.086	10.081		10.037	10.037	10.04	10.036
36	10.038	10.039	10.04	10.039		10.054	10.059	10.086	10.081		10.037	10.037	10.04	10.036

	client 200	client 300	client 400	client 500
multi thread	10.06087	10.2592	10.3811075	10.415742
select	10.119035	10.3338	10.516355	10.614366
thread pool	10.05506	10.21925	10.304645	10.365644

해당 단위는 sec이다



해당 결과를 해석해보면 다중 코어 환경에서 예상한대로 multi thread 기반 서버가 single thread 기반서버보다 더 좋은 성능을 보여줬다. client수가 증가할수록 더욱 큰 격차로 시간 차이가 발생하였는데, 이는 select기반 서버에서 fd_set이 증가하면 할수록 루프를 돌며 fd를 찾아야 하기때문에 더욱 더 cost가 증가하기에 발생한것으로 예상된다.

또한 같은 multi thread 서버끼리도 thread를 미리 만들고 readers write lock을 구현한 thread pool 서버가 naive하게 구현한 multi-thread서버보다 더 나은 성능 보여주었다. 마찬가지로 client수가 적을 때는 유의미한 차이가 없다가 client수가 더욱 늘어 동시에 lock이 걸리는 thread들이 많아지고 thread 생성시 드는 cost를 감안했을 때, 더욱 큰 차이가 나는 것으로 확인된다.

2. **order type**에 따른 성능비교

해당 실험은 buy, show, sell 이 세가지 request에 따라 서버가 어느정도의 성능 차이가 나는지 확인한다. 이 때 좀더 변수를 줄이기 위해 buy, sell하는 주식의 id를 고정하여 해당 요청이 동시에 들어와 lock이 더 많이 걸리도록 실험을 진행하였다. 해당 결과는 아래와 같다.

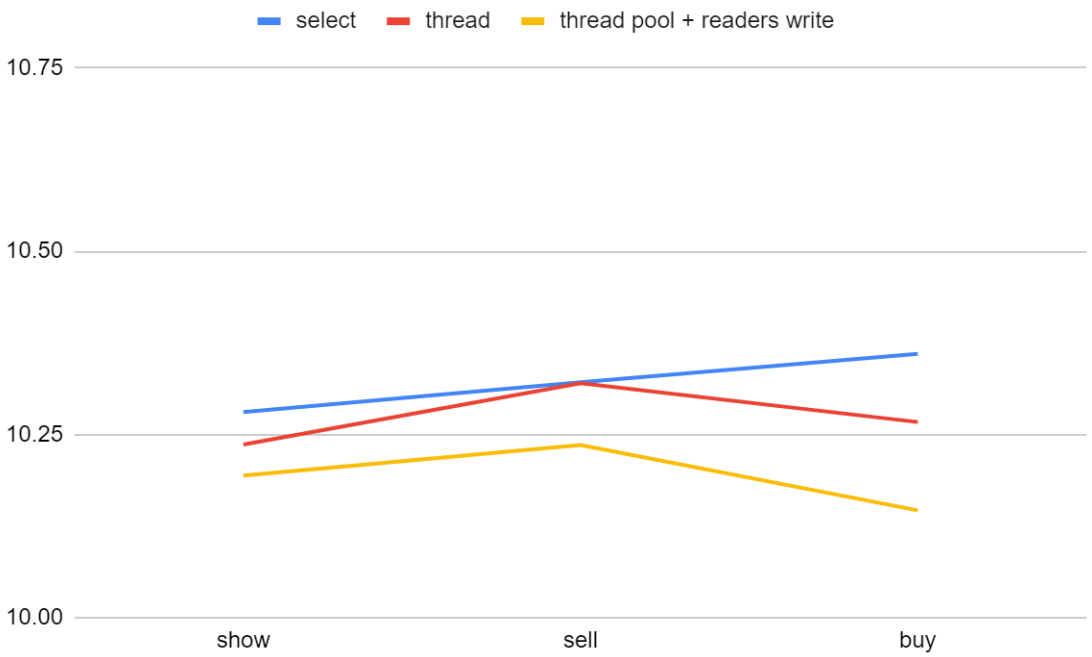
- orders : 100
- clients : 300
- stocks : 10

select		
show	sell	buy
10.05	10.045	10.048
10.05	10.046	10.048
10.05	10.046	10.048
10.05	10.046	10.048
10.05	10.046	10.049
10.05	10.046	10.049
10.05	10.046	10.049
10.05	10.046	10.049
10.05	10.046	10.049
10.05	10.046	10.049
10.05	10.046	10.054
10.05	10.046	10.054
10.051	10.046	10.054
10.051	10.046	10.054
10.051	10.046	10.054
10.051	10.046	10.054
10.051	10.046	10.058
10.051	10.046	10.059
10.051	10.054	10.059
10.051	10.054	10.059
10.051	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.055	10.054	10.059
10.056	10.055	10.062
10.056	10.055	10.062
10.056	10.055	10.062
10.056	10.055	10.062
10.056	10.055	10.062

thread		
show	sell	buy
10.03	10.032	10.03
10.03	10.034	10.032
10.03	10.034	10.032
10.03	10.034	10.033
10.03	10.034	10.033
10.03	10.035	10.033
10.032	10.035	10.034
10.032	10.035	10.034
10.032	10.035	10.034
10.032	10.035	10.034
10.032	10.035	10.034
10.032	10.036	10.035
10.034	10.036	10.036
10.035	10.037	10.036
10.035	10.037	10.036
10.035	10.037	10.036
10.035	10.037	10.036
10.035	10.037	10.036
10.035	10.038	10.037
10.037	10.039	10.037
10.037	10.039	10.037
10.037	10.039	10.037
10.037	10.039	10.039
10.038	10.04	10.039
10.038	10.04	10.039
10.038	10.04	10.039
10.038	10.04	10.039
10.038	10.04	10.04
10.038	10.04	10.04
10.038	10.04	10.04
10.038	10.04	10.04
10.038	10.04	10.04
10.039	10.04	10.04
10.041	10.04	10.041
10.041	10.04	10.041
10.041	10.041	10.041

thread pool		
show	sell	buy
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.029	10.03
10.032	10.031	10.03
10.033	10.031	10.03
10.033	10.031	10.03
10.033	10.031	10.031
10.033	10.031	10.031
10.033	10.031	10.031
10.033	10.033	10.033
10.033	10.033	10.033
10.033	10.033	10.033
10.033	10.033	10.033
10.033	10.033	10.033
10.033	10.035	10.033
10.033	10.036	10.033
10.033	10.036	10.033
10.033	10.036	10.033
10.033	10.036	10.033
10.035	10.036	10.033
10.035	10.036	10.033
10.035	10.038	10.033
10.035	10.038	10.033
10.037	10.039	10.033
10.037	10.039	10.033
10.037	10.039	10.034
10.038	10.039	10.034
10.038	10.039	10.035
10.038	10.039	10.036
10.04	10.039	10.036
10.04	10.039	10.036
10.04	10.039	10.036
10.04	10.039	10.037
10.04	10.043	10.037

	show	sell	buy
select	10.28074667	10.32136667	10.3600598
multi thread	10.23637209	10.31989333	10.26697667
thread poll	10.19389333	10.23562	10.14643854



마찬가지로 해당 결과를 해석해보자, 먼저 show에서는 readers lock을 구현한 thread pool 서버가 multi thread 서버보다 성능이 나음을 확인할 수 있고 두 multi thread 서버가 select 서버보다 더 성능이 좋은 것을 확인할 수 있다.

다음으로 sell에서는 마찬가지로 thread pool서버가 가장 성능이 좋은 것으로 확인할 수있고 multi-thread 와 select 서버의 성능이 비슷한 것을 확인할 수 있다. 이는 multi thread 서버에서 같은 주식에 매번 접근하여 write lock이 걸려 single thread와 비슷한 환경처럼 성능이 저하되어 select서버와 성능이 비슷해지는 것으로 보인다.

마지막으로 buy이다. multi thread에서 buy가 show와 sell보다 성능이 좋게 나오는 것을 확인할 수 있는데, 이는 buy할 때, 먼저 주식에 대한 정보를 read하고 부족하면 write하지 않기에 write lock이 걸리지 않아 더욱 성능이 좋게 나오는 것으로 예상된다. 때문에 두 multi thread 서버의 성능 그래프 모양이 비슷하게 보여진다.