System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 이동건

학번 : 20181664

1. 개발 목표

여러 클라이언트가 주식서버에 동시에 접속하여 요청하는 request를 처리하는 concurrent stock server를 구현하고자 하였다. 서버는 client가 요청한 명령(show, buy, sell)에 해당하는 작업을 수행하고 그에 맞는 출력값을 client에 전달해준다. 요청이 모두 수행되고 client와의 접속이 모두 끊어지면 주식테이블의 정보는 disk에 저장하는 방식으로 진행된다. 이는 stock.txt로 저장한다.

. 이때 여러 client가 동시에 접속하여 동시에 request를 요청했을 때, concurrent하게 처리를 하도록 구현하는데 이때 두가지 방법으로 구현한다. 하나는 event-driven 의 방식으로, 하나는 thread-based 방식으로 concurrent 처리를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

select 함수를 이용하여 I/O multiplexed Event Processing을 하게 되도록 구현한다. concurrent 하게 보이지만 select 함수로 pending input이 있는 connfd를 확인하고 차례로 처리하는 방식으로 구현되게 되는 것이다. 해당 방식으로 구현하게 되면 thread나 process overhead가 발생하지 않게 되므로 client개수에 따라 실행시간이 크게 변하지 않게 된다. 대신 single-core만 사용하여 multi-core를 사용했을 때의 더 개선되는 부분은 없다는 단점이 있다.

2. Task 2: Thread-based Approach

코드에 지정되어있는 nthread 변수에 값에 따라 일하는 thread을 개수에 맞게 생성하여 주고 해당 thread들이 계속 일하게 되는 방식으로 구현하게 된다. 따라서 일하는 thread의 개수에 따라 실행시간이 크게 바뀌고 work thread 개수보다 client 개수가 많아지면 실행시간이 크게 늘어나게 되는 특징이 있다. process를 생성하는 것보다 훨씬 효율적이면서 데이터를 공유할 수 있다는 장점이 있는 반면 공유하기 때문에 의도하지 않은 error 가 생길 위험성도 있다는 단점도 있다.

3. Task 3: Performance Evaluation

event-driven approach, thread-based approach를 client 개수에 따른 실행시간을

비교하고, 동시 처리율을 비교한다. 또한 thread-based에서는 work thread 개수를 변화 시켜가면서 실행시간이 어떻게 되는 지도 확인한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

I/O multiplexing을 구현하기 위해 먼저 connected descriptor 를 저장하고 관리할 pool이라는 구조체를 정의한다. init_pool 함수를 이용하여 listenfd와 함께 pool을 초기화 시켜주고, while 문을 돌며 select 함수로 pending bit이 있는 descriptor를 배열인 pool 구조체의 ready set에 나타낸다. 그리고 만약 listenfd의 pending bit이 set이 되어 있으면 accept 함수를 이용하여 client와 연결하고 해당 connfd를 add_client 함수를 이용하여 pool의 read_set에 추가한다. 이후로는 check_clients 함수를 이용하여 pool의 connfd을 탐색하며 pending bit이 set 되어 있는 connfd를 하나씩 차례로 요청을 하나씩 처리하도록 한다. 이때 exit 을 입력하면 close 함수로 연결을 끊고 is_connect 함수로 서버에 연결된 connection이 있는지 확인한다. connection이 없다면 save_stock 함수로 주식 정보를 stock.txt로 저장한다.

  ✓ epoll과의 차이점 서술

select는 코드를 보면 알겠지만 file descriptor 배열을 하나씩 탐색하는 loop을 돌고 이를 위해 select 함수를 호출할 때마다 descriptor를 저장하는 전체 정보를 넘겨주어야 하는 overhead가 발생한다. 또한 loop을 돌기 때문에 descriptor의 개수가 많아질 수록 처리 시간이 증가한다는 단점이 있다. epoll 함수는 이러한 단점을 보완한 함수로 직접 운영체제에서 fd 관리를 담당함으로써 정보를 매번 함수에 전달할 필요도 없고 커널 공간이 직접 관리하다보니 select 함수보다 실행 속도도 빠르게 처리가 가능하다는 차이점이 있다.

- Task2 (Thread-based Approach with pthread)

  ✓ Master Thread의 Connection 관리

worker thread pool을 관리할 구조체를 먼저 선언해주고 sbuf_init으로 초기화 해준다. 그리고 master thread는 nthread 개수 만큼 worker thread를 생성하여 준다. 이후 main

에서 while 문을 돌며 accept 함수로 connfd를 받아오게 되면 이를 sbuf_insert 함수로 일단 sbuf 에 넣어주게 된다. 즉 정리하면 master thread는 connfd를 받아오면 이를 sbuf에 넣어주고 connfd를 관리하는 것이다. 그리고 이 sbuf에 있는 connfd를 worker thread가 차례로 가져와 통신을 하게 하는 것이다.

- ✓ Worker Thread Pool 관리하는 부분에 대해 서술

각 thread 들은 thread 함수를 실행하게 되는데 이때, sbuf_remove로 connfd를 꺼내어 해당 thread는 해당 connfd로 끊길 때 까지 통신을 하게 된다. 만약 해당 connfd와 통신이 끊기면 while문으로 다시 sbuf_remove함수로 새로운 connfd를 가져와 새로운 통신을 하며 client request를 처리하게 된다. 이때 stock 정보에 concurrent하게 접근하고 정보를 수정할 수 있으므로 semaphore를 활용하여 정보에 오류가 나지 않도록 한다.

- Task3 (Performance Evaluation)

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

performance는 event-driven, thread-based를 각각 client 개수를 차이를 두어 실행시간이 어떻게 되는지 비교해보고 thread-based는 worker thread 개수에 따라 동시처리율에 어떠한 변화가 있는지 확인해본다. 이렇게 정한 이유는 thread-based 와 event-driven은 둘다 concurrent 한 처리가 되는 것처럼 보이는 공통점 있지만 내부적으로는 thread-based 방법은 실제로 concurrent하게 client 요청이 처리가 되지만 event-driven은 하나씩 차례로 처리 되면서 동시에 처리되는 것처럼 보이는 방식이기 때문에 실행시간에 차이가 발생할 것이라고 예상했다. 또한 thread-based 방식은 worker thread 개수를 따로 지정해줄 수 있으므로 해당 개수에 따라 실행 시간에 변화가 있는지 확인해보고자 하였다.

추가로 buy,sell 명령어만 실행했을 때, show 명령어만 진행했을 때의 실행시간을 비교해보고자 한다. buy와 sell은 실제 주식 정보에 접근하여 값을 조작하는 명령어이고 두 명령은 동시에 처리되기 어렵다는 특징이 있고 show는 단순이 값만 출력해 주는 명령어 이므로 명령어들 사이에서의 실행시간 차이도 존재할 것이라고 예상된다.

이를 측정하기 위해서 gettimeofday 함수를 사용한다. 이는 multiclient.c 파일에서 사용함으로서 시간을 측정하였다.

- ✓ Configuration 변화에 따른 예상 결과 서술

일단 event-driven 과 thread-based 방식은 비교하였을 때 thread-based 방식이 실행시간이 더 짧을 것으로 예상된다. 즉 동시처리율이 더 높을 것으로 예상된다. 그 이유는 event-driven 은 동시에 처리되는 것 처럼 보일 지 모르지만 실제로는 동시에 처리되는 것이 아니고 선택된 connfd 가 차례로 요청을 처리하는 것이기 때문에 실제로 concurrent 실행이 이루어지는 thread-based 보다는 처리 시간이 느릴 것으로 예상된다.

다음으로 thread-based 에서 worker thread 가 많을 수록 실행시간이 단축될 것으로 예상된다. worker thread 보다 client 수가 적은 경우에는 시간이 비슷할 것 같지만 worker thread 보다 client 수가 많아지면 event-driven 과 비슷한 방식으로 worker thread 에 할당된 connfd 만 처리가 가능하므로 같은 client 수라면 client 수가 많을수록 실행시간이 단축될 것으로 예상된다.

C. 개발 방법

task 1 : 먼저 item이라는 구조체를 선언해 주어 주식 정보를 저장할 struct를 만들어 준다. 이때 주식의 id, 남은 주식의 개수, 가격 정보가 들어 있다. 또한 이것을 바이너리 트리로 저장할 구조체도 선언해 준다. items_BT 라는 구조체로 이 안에는 주식의 개수 정보와 주식 정보의 구조체를 배열로 갖고 있는 구조체이다.

이어서 descriptor를 관리할 pool 구조체도 선언 해 준다. clientfd 배열은 connfd를 넣을 배열이다.

이어서 함수 구현을 보면 save_stock으로 주식 정보를 stock.txt에 저장하는 함수를 구현해 주었다. create_BT라는 함수를 통해 stock.txt에서 주식 정보를 읽어와 바이너리 트리로 구현하는 함수이다. show 함수는 show 요청이 들어왔을 때 수행하는 함수로 저장되어 있는 주식의 정보를 나타내어 주는 함수이다. 이 때 parameter로 가져온 ret_buf에 정보를 모두 담고 해당 ret_buf를 Rio_writen 함수로 client에 보내준다.

buy 함수는 주식 정보가 저장되어있는 바이너리 트리를 순차적으로 탐색하며 해당 아이디에 일치하는 주식 정보를 찾는다. 그리고 주식 정보를 변경하고 만약 불가능하다면 불가능할 때 출력할 메세지를 ret_buf에 저장하고 가능하다면 가능할 때 출력할 메세지를 ret_buf에 저장하고 Rio_writen 함수로 client에 보내준다.

sell 함수는 buy와 마찬가지로 저장되어 있는 바이너리 트리를 순차적으로 탐색하며 해당 아이디에 일치하는 주식 정보를 찾고 주식정보를 변경한다. sell은 불가능한 경우가 따로 없으므로 정보를 알맞게 변경하고 성공했다는 메세지를 ret_buf에 담고 Rio_writen 함수로 client 에 보내준다.

그리고 서버에 connection이 있는지 없는지 확인해 주는 is_connect 함수를 구현하였다. 이는 pool 구조체의 clientfd 배열을 탐색하며 connfd 값이 모두 -1이면 연결된 client가 없음을 의미하므로 없으면 0을 return 있으면 1을 return 해 주는 함수이다.

task2 : show buy sell 함수, create_BT 함수의 구현은 task1과 동일하다. event는 순서대로 처리하는 반면 thread는 concurrent하게 처리되기 때문에 thread끼리 공유하는 데이터 조작시 조심해야하는 부분이 있다. 이를 semaphore을 이용하여 오류나 버그가 생기는 것을 막아 주어야 한다. 따라서 이를 이용하기 위해 mutex라는 것을 이용한다. show는 mutex에 관련 없는 함수이지만 같은 종목에 대해서 buy 와 sell은 도중 buy, sell을 하면 안되므로 종목마다 buy나 sell을 하고 있으면 하고 있음을 나타내어주는 mutex를 과목마다 추가해주어야 한다. 따라서 buy나 sell을 할 때마다 해당 아이디의 주식이 현재 buy나 sell을 하고 있는지 확인한 다음 그렇지 않을 때에 명령을 수행할 수 있도록 해주었다.

또한 connfd를 저장 해 놓은 sbuf_t에 대한 struct를 나타내어 주고 해당 구조체는 sbuf_init으로 초기화 하고 sbuf_insert 함수로 connfd를 추가하고 sbuf_remove함수로 connfd를 꺼내는 수행을 진행한다. 또한 init_echo_cnt 함수로 mutex 값을 초기화 시켜준다.

echo_cnt 함수에서 명령어를 입력받아 명령어에 맞는 수행을 할 수 있도록 해준다. 그리고 생성된 worker thread 들은 각각 따로 thread 함수를 반복해서 수행하게 된다. 또한 connfd_cnt 변수를 새로 만들어서 현재 서버에 연결된 connfd 개수를 count 하여 연결된 클라이언트가 있는지 확인한다. 연결된 클라이언트가 없으면 stock.txt로 주식 정보를 저장한다.

3. 구현 결과

task1 : 구현 결과는 여러 client 가 요청했을 때 select 함수로 요청받은 connfd를 차례로 check_clients 함수에서 Rio_readlineb를 통해 명령을 읽어 온다. 읽어온 명령에 따라 show buy sell 함수를 실행하여 해당 명령을 수행한 후의 출력 결과를 ret_buf에 담아 Rio_writen 함수로 client에 결과를 넘겨 준다. 실행 하게 되

면 client 수가 작을 때는 동시에 출력되는 것을 보이지만 client 수가 커질 수록 clinet요청을 하나씩 처리되는 것을 확인 할 수 있다. show를 입력하면 현재 주식 정보를 출력하여 주고 sell 1 2 를 입력하면 1 id 값의 주식을 2개를 판매하므로 성공했을 때 [sell] success가 클라이언트에 표시된다. buy 1 2 를 입력하면 1 id 값의 주식을 산다는 명령으로 2개보다 적게 있으면 Not enough left stock 메세지를 클라이언트에 표시하고 사는데 성공했으면 [buy] success를 표시한다.

task2 : 먼저 worker thread를 생성한다. 이후 accept로 생성된 connfd를 sbuf에 넣고 sbuf에서 thread 들이 각각 connfd를 꺼내어 연결을 하고 각 클라이언트와 통신을 시작하고 request를 요청받는다. 이때 thread 함수에서 각각 thread들이 명령어를 받고 task1과 마찬가지로 show buy sell의 명령을 수행하고 결과를 클라이언트에 넘겨준다. 이때 client 수를 크게 했을 때 task1과는 다르게 하나씩 차례로 수행되는 것이 아니라 한번에 여러 client를 수행하는 결과를 볼 수 있다. 단 sell buy를 진행할 때는 semaphore방식을 사용하여 동시에 진행될 수 없는 경우가 있어 delay가 되는 것도 확인 할 수 있었다.

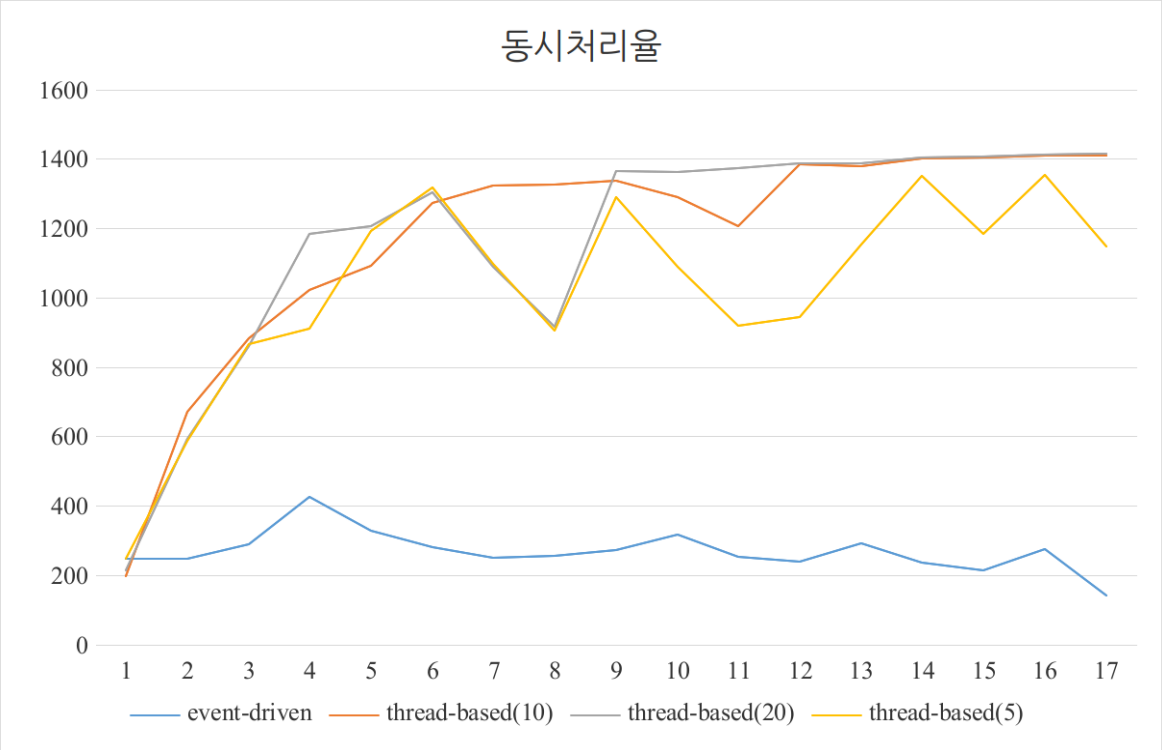
4. 성능 평가 결과 (Task 3)

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

측정 시점은 multiclient가 시작되는 시점에서 multiclient에서 생성된 childprocess 들이 모두 exit 했을 시점까지를 측정하였다.

이는 모든 명령어를 처리하고 종료된 시점이라고 판단했기 때문이다.

먼저 event-driven. thread-based 를 비교해보자. 이때 thread-based는 worker thread 개수에 따라 다르게 결과 값이 나올 수 있으므로 thread-based는 worker thread 개수를 다양하게 하여 측정하였다. 측정한 결과는 다음과 같다.



이때 동시처리율 값은 1초당 client 처리 요청 개수를 나타내었고 이는 client 수 * 10^7 / 측정시간(microsecond)로 나타내었다.

표에는 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17은 차례로 client 개수가 1,3,5,10,20,30,40,50,60,70,80,90,100,200,300,400,500을 의미한다.

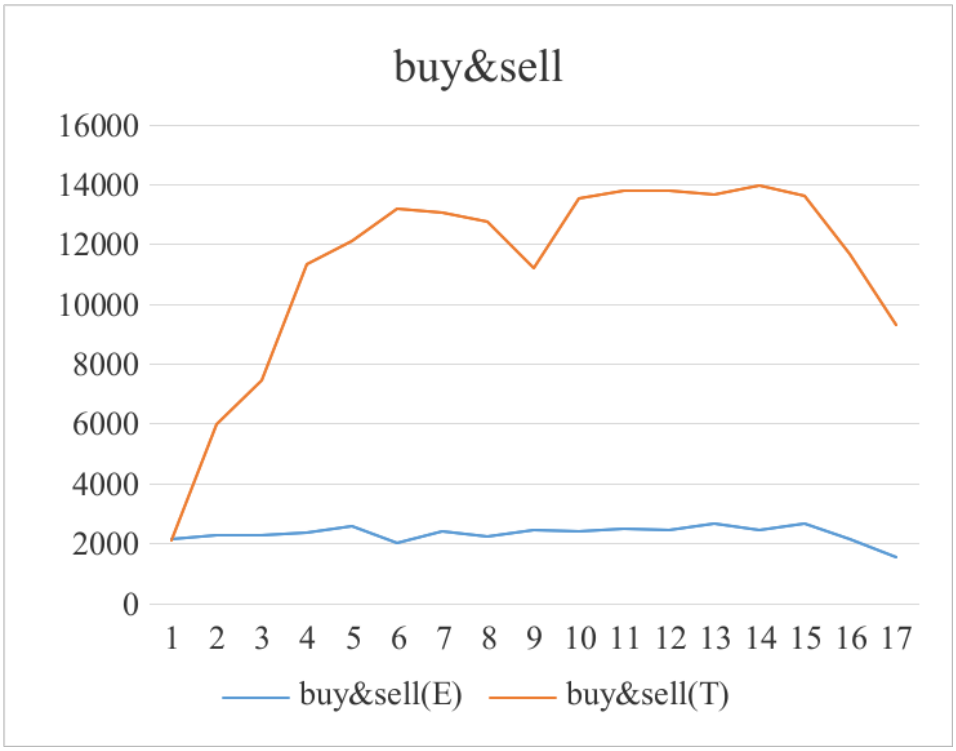
결과를 보게 되면 event driven 접근 방식은 client 개수에 상관없이 동시처리율이 일정한 것을 확인할 수 있다. 이는 측정 이전에 예측했던대로 event driven 방식이 아무래도 concurrent 하게 보이지만 실제로는 concurrent 하지 않고 하나씩 차례로 처리하는 방식이다 보니 동시처리율이 일정한 것이라고 추측해볼 수 있다.

thread-based 접근 방식은 concurrent 하게 request를 처리하다보니 client 개수가 늘어날 수록 동시처리율이 증가하는 것을 확인할 수 있다. 다만 thread worker 개수가 작을 수록 큰 차이는 아니지만 동시처리율이 client 개수가 늘어날 수록 떨어지는 것을 확인할 수 있었다.

이어서 워크로드에 따른 분석을 해보았다. 이는 client 요청 타입 즉 명령어가 무엇이냐에 따라 동시처리율에 변화가 생기느냐를 분석해본 것인데 buy sell 만 요청했을 때와 show 만 요

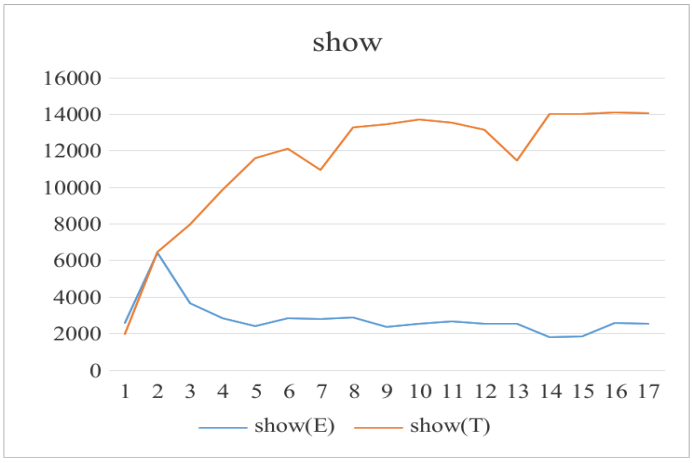
청했을 때 그리고 random으로 요청했을 때를 비교해보았다.

먼저 buy&sell만 요청했을 때 event-driven 과 thread-based 를 비교해보았다.



이때 buy&sell Event-driven 과 thread를 비교했을 때 위의 그래프와 같이 event-driven방식은 크게 변화가 없는 것을 확인할 수 있고 thread-based 방식은 동시처리율이 계속 증가하다가 마지막에 떨어지는 것을 확인 할 수 있다.

show명령만 입력했을 때는 그래프 개형이 buy&sell과 거의 동일함을 확인할 수 있다.

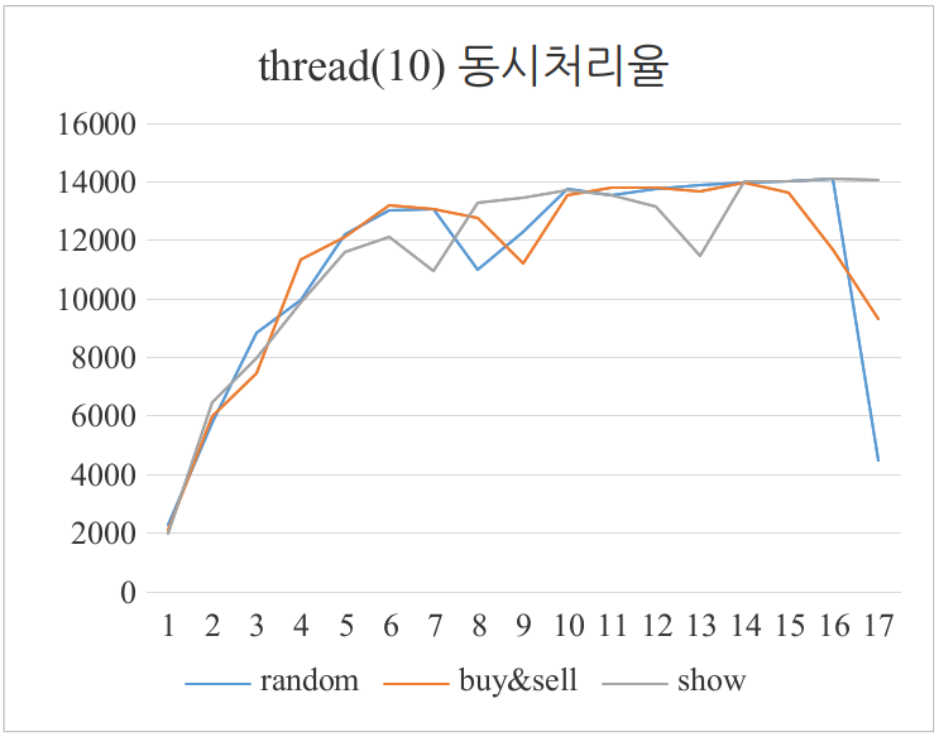


]

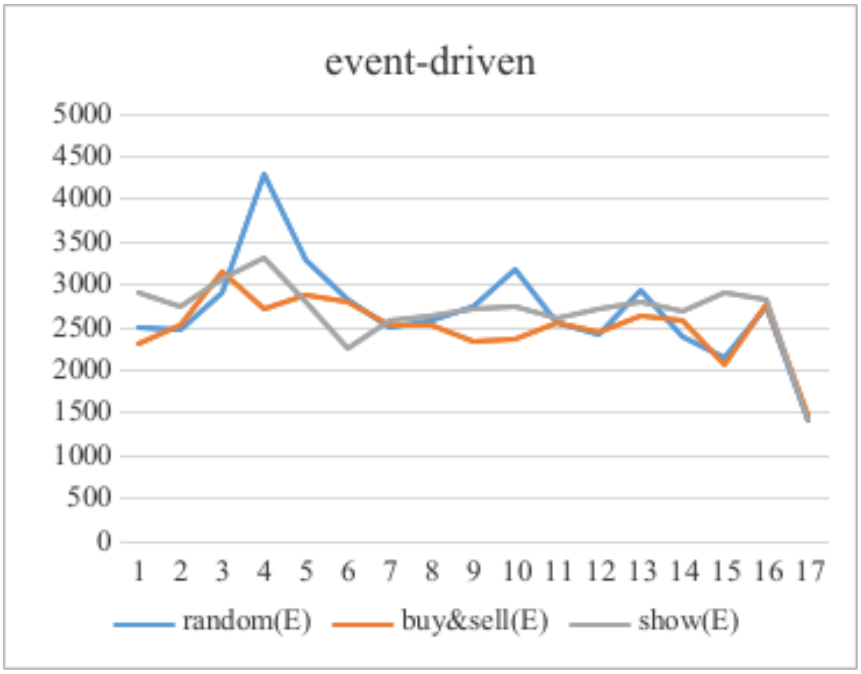
즉 event-driven과 thread-based는 명령어에 의해서는 크게 달라지지 않음을 확인할 수 있다.

그럼 명령어 끼리의 비교를 해보면 어떤지 살펴보면 다음과 같다.

먼저 thread-based 기반에서의 buy sell 명령어만 입력했을 때와 show만 입력했을 때의 결과이다.



거의 비슷한 듯 보이지만 show 명령어보다 buy&sell 명령어가 동시처리율이 조금 더 높은 것을 확인할 수 있었다. 이는 show 명령어는 모든 주식 정보에 접근하여 일일이 모두 출력을 해야하는 부분에서 실행시간이 조금 더 걸려서 그런 것으로 추측된다.
event-driven 방식에서 비교해보면 다음과 같다.



event-driven 방식에서는 buy&sell 보다 show가 더욱 동시처리율이 좋게 나온 것을 확인할 수 있다. 예측 상으로는 buy&sell이 더욱 처리율이 좋을 것이라고 예상했지만 그렇지 않은 결과가 나온것에 대해서는 어떠한 이유인지 추측이 어렵다.

최종적으로 분석한 결과를 정리하자면, event-driven 과 thread-based 방식에서 client 개수가 증가할 수록 thread-based 방식이 동시처리율이 더 높은 것을 확인할 수 있었다. 그 이유를 추측해보자면 event-driven은 실제로는 하나씩 처리되는 방식이고 thread-based는 실제로 concurrent하게 실행되는 방식이기 때문이라고 추측해볼 수 있다. 다만 thread-based 방식은 worker thread 의 개수에 따라 실행시간이 달라지는 것도 확인해 볼 수 있었다. worker thread의 개수보다 적은 경우에는 개수에 상관없이 실행시간이 비슷했으나 그렇지 않은 경우에는 worker thread 개수가 많을 수록 실행시간이 빨라지는 것을 확인할 수 있었다.

추가적으로 thread-based 방식의 경우에는 지정한 sbuf의 크기나 nthread의 개수에 따라 client 가 상대적으로 많이 커지게 되면 작동하지 않는 경우도 확인할 수 있었으나 event-driven 방식은 그런 조건은 전혀 상관없이 작동하는 것을 확인할 수 있어 상당히 많은 client 요청에 유연한 대응은 event-driven 가 더욱 유리할 것 같다는 분석도 해볼 수가 있었다.

그리고 명령어에 대한 분석은 buy&sell만 했을 때와 show만 했을 때가 thread-based와

event-driven 방식이 결과가 각각 달라 명령어는 어떤 것이 특정하게 느린지는 분석하기 어려웠다.