System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 김민식

학번 : 20171609

1. 개발 목표

여러 client들의 동시 접속 및 서비스를 위한 Concurrent server를 구축하고, 이를 이용하여 주식을 사고, 팔 수 있는 stock server를 제작한다.

Concurrent한 Stock server를 만들 때 event-driven approach, thread-based approach 두가지 방법을 사용하여 구축한다.

Client는 show, buy, sell, exit 4개의 명령어를 사용할 수 있다. Show 명령어는 현재 주식의 상태를 보여주고, buy와 sell은 주식을 사고 팔 때 쓰이는 명령어이며, 마지막으로 exit 명령어를 입력하면 주식 장을 퇴장하게 된다.

2. 개발 범위 및 내용

A. 개발 범위

1. Select

   Event-driven Approach를 이용하여 concurrent한 서버를 구축하였다. 여러 client와 연결된다고 할 때, 각 client는 fd를 trigger 하고, select 함수를 통해 동작을 시작한다.

   Server는 각각의 fd들을 위한 배열을 만들어 각 client의 명령어들을 병렬적으로 처리한다. 하나의 process가 여러 client를 handling 하기 때문에 시작과 동시에 파일의 내용을 메모리에 올리고, in-memory 프로세싱을 진행한다.

2. Pthread

   Thread-based Approach로 concurrent한 서버를 구축하였다. 이를 위해 pthread 함수를 사용하였다.

   서버는 여러 개의 thread를 생성하게 되고, 각각의 thread마다 서버와 연결시켜주며 concurrency하게 client들의 명령을 수행할 수 있다. 다른 client가 approach 하고 있어도 read, write 가 가능하다.

B. 개발 내용

- **select**

✓ **select 함수로 구현한 부분에 대해서 간략히 설명**

하나의 process가 서버 소켓과 여러 개의 클라이언트 소켓을 관리한다. 관리할 때는 fd_set이라는 구조체를 사용한다.

select함수는 fd에 read, write, exception등이 발생했는지 확인하는 함수이다. 즉, fd_set 의 set을 이용하기 위해 필요한 함수라고 생각할 수 있다. Fd_set을 전달하여 호출하면 변화가 발생한 socket의 디스크립터만 따로 설정한다. 그 후 fd_set에 대한 주소값을 전달하고, 그 결과를 적용한다.

새로운 client가 연결을 시도할 때 select 함수를 통해 알 수 있으며, check_client 함수를 통해 현재 이미 연결되어 있는 client 를 관리 할 수 있다.

✓ **stock info에 대한 file contents를 memory로 올린 방법 설명**

우선, 명세서에 나와있는 대로 고속의 탐색을 위해 binary tree를 사용하였다. Item 구조체를 선언하여 각 stock의 id, price, 그리고 각 left, right item을 저장하여 stock을 표현하고자 하였다.

터미널에서 stock 프로그램을 실행시키면 stock.txt 파일을 불러 읽는다. 각 줄에 있는 주식의 정보를 저장한 후 stock의 id에 맞추어 위에 선언한 구조체에 저장한다. Stock.txt에는 주식의 id가 정렬 되어있지 않기 때문에 첫 줄을 읽고, 이를 기반으로 다음 줄의 id와 비교하여 left item, right item 구조체에 다시 저장시킨다. 이러한 방식으로 stock.txt 에 있는 주식들의 정보를 명세서의 구조체로 저장하여 stock의 정보를 memory로 올릴 수 있다.

- **pthread**

✓ **pthread로 구현한 부분에 대해서 간략히 설명**

thread 기반 소켓 프로그래밍은 프로세스 대신 thread를 이용하여 서버와 클라이언트를 연결한다. Fork() 함수를 대신하여 pthread_create() 함수를 이용하여 새로운 thread를 만든다. 이 thread는 소켓의 소켓 지정 번호를 매개 변수로 받아들

일 수 있다. 따라서 이 소켓을 이용하여 client를 처리하게 된다.

앞서 말했듯이 pthread_create 함수를 함수를 이용하여 thread를 미리 만들고, pool에 저장한다. 그 후 새로운 인풋이 들어왔을 때 실행중인 thread는 잠시 정지 시킨 후 연결되지 않은 thread을 할당하여 처리해주는 방식으로 구현하였다.

서로 다른 client, thread가 공동으로 접근할 수 있기 때문에 counting semaphore와 mutex lock을 이용하여 관리할 수 있도록 하였다.

C. 개발 방법

1. 구조체 선언

```
typedef struct item
{
    int id, left_stock, price;
    struct item *leftitem;
    struct item *rightitem;
} StrItem;
StrItem *StrStock;
```

Stock의 정보를 저장하기 위한 구조체 item을 선언하였다. 주식의 id와 가격, 잔여 주식량, 그리고 각각의 left, right node를 저장할 수 있다.

또한 stock의 정보를 table 형태로 표현하기 위해 strstock이라는 포인터를 하나 선언하여 서버를 구현하는데 사용하였다.

2. Pool 구조체 선언

```
typedef struct{
    int maxfd;
    fd_set read_set, ready_set;
    int nready, maxi, clientfd[FD_SETSIZE];
    rio_t poolrio[FD_SETSIZE];
} pool;


typedef struct
{
    int *buf;
    int n, front, rear;
    sem_t mutex, slots, items;
} sbuf_t;
sbuf_t structBuf;
```

교재 1004p를 참고하여 Select 함수 등을 사용할 때 사용하는 pool 구조체를 선언하였다. 또한 교재 1028p를 참고하여 phread를 사용할 때의 pool 구조체를 선언하였다.

3. Stockclient.c, multiclient.c

클라이언트가 server로 명령문을 입력할 때, server는 stock table을 내용을 보내준다. 이 때, 한 줄 일수도 있지만 여러 줄 일 수도 있기 때문에 stockclient.c와 multiclient.c 의 내용 중 rio_readlineb를 rio_readnb로 수정하였다.

또한 exit 명령문을 입력할 경우 클라이언트와 서버의 연결을 끊어주도록 하였다. 클라이언트가 exit를 입력하여 서버로부터 "exit"를 다시 받을 경우 while문을 break 시켜 연결을 끊도록 하였다.

4. Stockserver.c

Client의 command에 관련된 요소를 추가하였다.

각 명령문 show, buy, sell, exit 마다 각각의 명령문에 대한 함수를 수행하도록 하였다. 먼저 show 명령문의 경우 FuncShow 함수를 호출하였다. Funcshow 함수는 recursion을 이용하여 현재 item 구조체에 있는 모든 stock 의 정보를 출력하는 함수이다.

Buy와 sell의 경우 id와 그 개수를 함께 입력받는데, stock table에서 함께 입력받은 stock의 id를 찾은 뒤 그 개수를 늘리거나 줄여준다. 만약 buy인 경우에 stock의 남은 개수가 구입하려는 개수보다 적다면 에러 문구를 출력한다. 마지막으로 exit일 경우에는 클라리언트와 서버의 연결을 종료시킨다.

event driven approach 를 할 경우 select 함수를 이용하여 listenfd가 준비될 때까지 block 시켰다가 준비가 되면 accept 함수를 통해 fd를 얻는다. 그 후 클라이언트의 명령문을 처리하는 순서로 넘어간다.

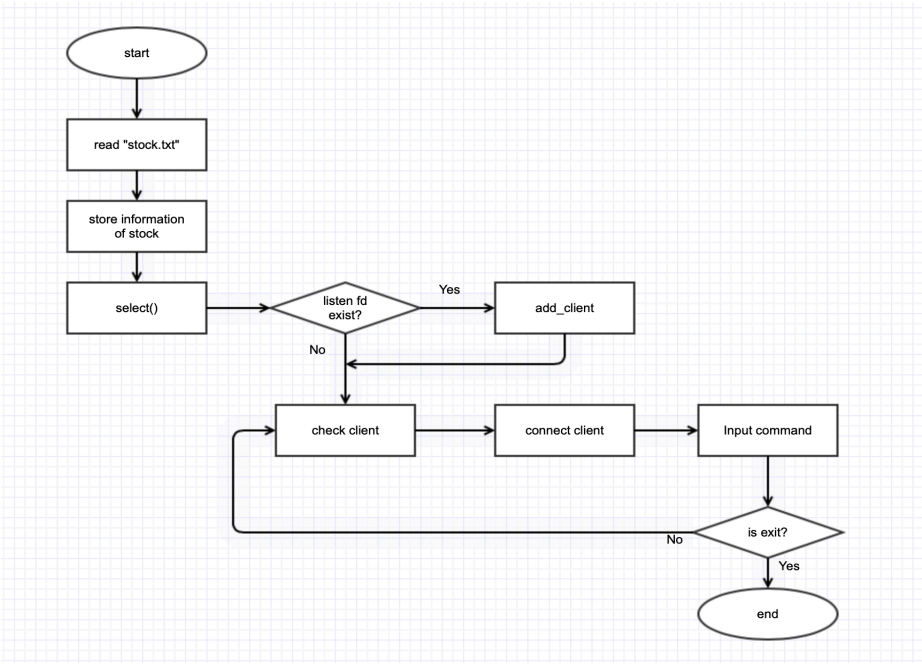
Pthread based approach의 경우 pthread_create 함수를 통해 연결이 되고 위와 똑같이 accept 함수를 통해 fd를 얻은 후 sbuf_insert를 이용하여 해당 fd를 버퍼에 저장시켜준다. 그 후 클라리언트 명령문을 처리한다.

Event driven approach, pthread based approach 두 경우 모두 클라이언트에 대한 실행이 끝나면 buy, sell의 결과물을 다시 stock.txt에 write 한다.

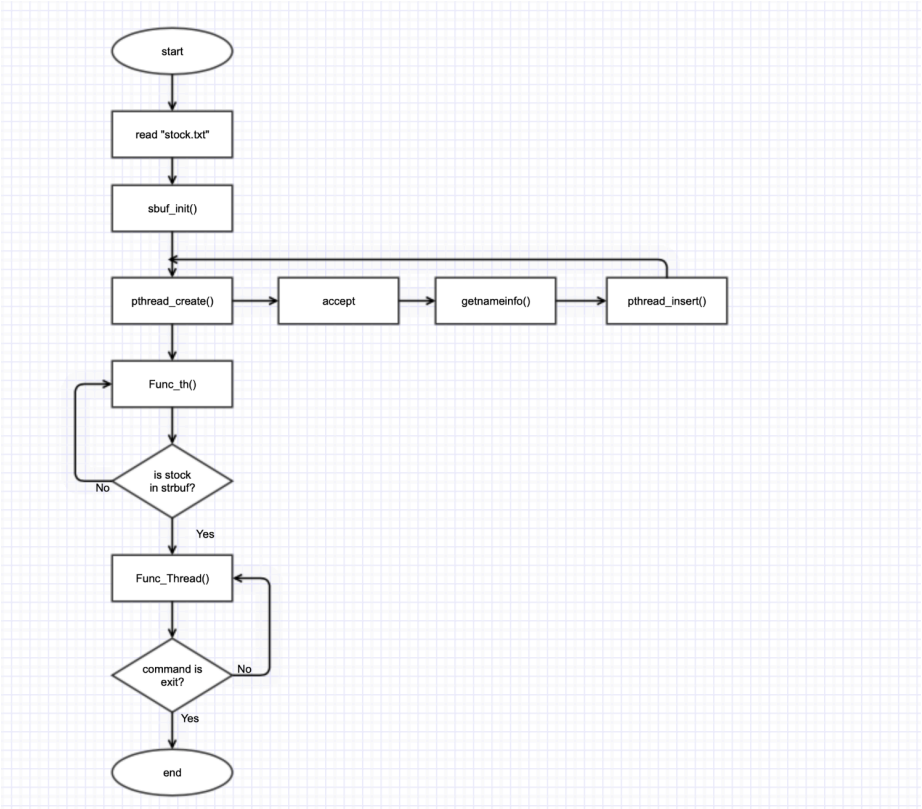
3. 구현 결과

A. Flow Chart

1. Select



2. Pthread



B. 제작 내용

1. Select

1) void FuncShow(StrItem *curItem, char *resItem)

Client가 show 명령어를 입력했을 때 호출되는 함수이다.

Stock의 정보가 담겨있는 구조체 item을 탐색하여 현재의 상태를 화면에 출력한다. 구조체가 이진 트리 형식으로 되어 있으므로 recursion을 활용하여 빠르게 탐색할 수 있다.

2) void FuncRefresh(StrItem *curItem, FILE *fp)

Client의 buy, sell 명령문에 의하여 stock의 개수가 변하였을 때, 이를 stock table에 다시 새로고침 해주는 함수이다.

1번의 Funcshow 함수와 같이 이진 트리인 item 구조체를 탐색하기 때문에 recursion을 이용하여 변경된 정보들을 구조체에 저장한다.

Client가 주식을 사고판 후 함수가 호출된다.

3) void init_pool(int listenfd, pool *p)

Pool을 초기화할 때 사용하는 함수이다. FD_ZERO 를 이용하여 read_set의 디스크립터 값을 모두 0으로 초기화한다. 또한 FD_SET을 이용하여 read_set의 listenfd번째에 위치하고 있는 디스크립터를 1로 바꾸어준다.

4) void add_client(int connfd, pool *funcP)

새로운 클라이언트를 pool에 추가할 때 호출되는 함수이다. Pool을 탐색하며 비어있는 clientfd를 찾고, 만약 있다면 connfd를 저장한다. 그 다음 FD_SET을 이용하여 아까 저장했던 connfd를 readset에 추가한다.

5) void check_clients(pool *funcP)

이 함수는 클라이언트와 서버가 연결할 때 클라이언트를 처리하기 위해 사용되는 함수이다. Readlineb로 client로 부터 명령문을 받는다. 먼저 명령문이 Exit인지 검사한다. 만약 exit라면 바로 서버로부터 연결을 끊고 리턴한다.

만약 Exit가 아니라면 명령문에 대한 결과값을 rio_writen을 통해 클라이언트에게 다시 전달한다. 모든 명령문이 끝나면 stock의 정보가 저장되어있는 구조체를 통해 다시 stock.txt에 그 값을 write 해준다.

3), 4), 5) 함수는 교재 1004 p ~ 1006 p 를 참고하여 사용하였다.

2. **Pthread**

1) char *FuncTran(int p)

정수, 숫자로 저장되어 있는 정보, 즉 stock의 id나 남은 갯수 등을 문자로 바꿀 때 사용하는 함수이다. Strcat을 사용하여 한번에 show 하기 때문에 문자로 바꾸어 사용한다.

2) void FuncShow(StrItem *curItem, char *resItem)

Select에서와 같이 Client가 show 명령어를 입력했을 때 호출되는 함수이다.

Stock의 정보가 담겨있는 구조체 item을 탐색하여 현재의 상태를 화면에 출력한다. 구조체가 이진 트리 형식으로 되어 있으므로 recursion을 활용하여 빠르게 탐색할 수 있다.

3) void FuncThread(int connfd)

클라이언트로부터 받은 명령문에 대하여 행동을 취할 때 사용하는 함수이다. Rio_readlineb로 명령문을 받고, 이때의 명령문에 따라 행동을 다르게 한다.

명령문이 show일 경우에는 Func_show를 호출하고, exit인 경우는 클라이언트와 서버의 연결을 종료한다.

Sell, buy일 경우에는 id와 stock의 개수를 함께 입력받는데, stock table에서

함께 입력받은 stock의 id를 찾은 뒤 그 개수를 늘리거나 줄여준다. 마지막으로 명령문이 종료되면 명령문으로 부터 변경된 stock의 정보들을 다시 stock.txt에 write 시켜준다.

4) void sbuf_insert(sbuf_t *sp, int item)

파라미터 sp->buf의 rear 부분에 파라미터로 받은 item을 넣을 때 사용하는 함수이다. 이때 sp->buf의 자리가 없을 때 item이 저장되지 않도록 설정하였다.

Item을 sp->buf에 저장할 때 다른 코드에 의하여 buf에 다른 값이 저장되는 것을 막기 위해 mutex lock을 사용하여 안전하게 저장될 수 있도록 설정하였다.

5) int sbuf_remove(sbuf_t *sp)

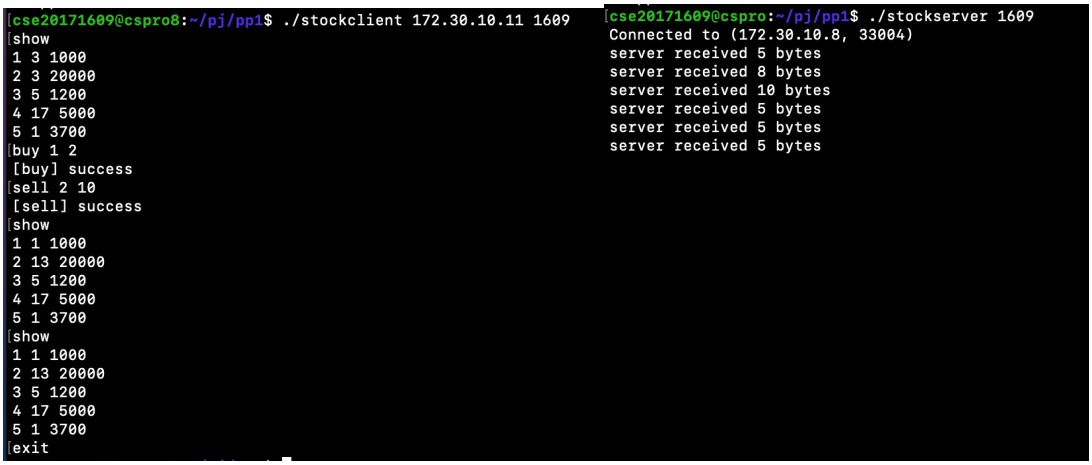
4번의 함수와 반대되는 함수로 sp->buf의 front에 있는 item을 삭제하는 함수이다. Sp->buf가 비어있는 경우 item을 삭제할 수 없도록 예외 설정해주었다.

이 또한 4번과 같이 sp->buf 에 있는 것을 삭제할 때 다른 코드의 영향을 받지 않기 위해서 mutex lock을 사용하여 critical section에 삭제하는 코드를 넣어 안전하게 삭제할 수 있도록 하였다.

5)과 4) 함수는 교재 p 1028 ~ 1029를 참고하여 작성하였다.

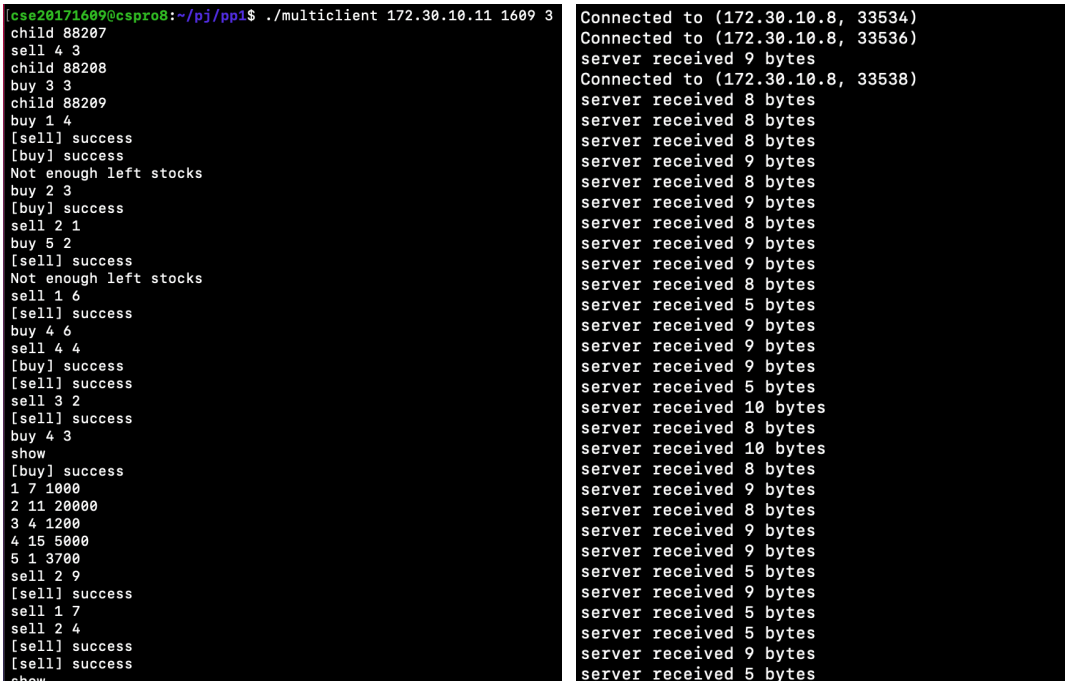
C. 시험 및 평가 내용

1) Project 수행 결과



위 그림과 같이 server와 client를 연결하여 show, buy, sell, exit 명령어가 성공적으로 전달되고, 이에 따라 결과가 도출된다는 것을 알 수 있었다.

그 후, stock.txt 를 확인하여 명령문의 결과가 stock.txt에 성공적으로 write 되었다는 것 또한 알 수 있었다.



Multiclient 또한 서버와 연결되어 성공적으로 실행됨을 볼 수 있었다.

2) 성능 평가 및 분석

클라이언트가 show만 호출할 경우, buy와 sell 만 호출할 경우, 그리고 세가지를 모두 혼합해서 호출할 경우에 대한 결과를 구했다.

또한 각 방법마다 client의 수에 변화를 주어 동시 처리율의 변화를 분석하였다.

이때 multiclient.c 에서 option 변수 값이 각 명령어를 나타낸다. Option이 0이면 show, 1이면 buy, 2이면 sell을 나타낸다, 따라서 option 값을 조절하여 client가 원하는 명령어만 수행시킬 수 있도록 조절하였다.

- Show 만 호출 할 경우 : Option = 0
- 세 명령어 모두 혼합하여 호출 : option = rand()%3
- Buy와 sell만 호출 : option = rand()%2 + 1

위와 같이 설정 후 client의 수를 순차적으로 늘려 실행 시간을 구했다. 각 테스트케이스마다 3번의 시간을 구하여 평균값을 내었고, 이를 실행시간으로 하였다. 또한 이를 바탕으로 동시처리율을 구했다. 이에 대한 결과는 아래의 표를 통해 나타낼 수 있었다.

실행시간(혼합)		
client	thread	select
1	0.020291	0.001176
5	0.029152	0.003492
10	0.038878	0.010274
50	0.069808	0.048352
100	0.137565	0.138838
500	0.407892	0.569118
1000	0.930299	2.484993

실행시간(show)		
client	thread	select
1	0.030254	0.001458
5	0.033986	0.004536
10	0.042344	0.010629
50	0.068097	0.044633
100	0.141671	0.107927
500	0.473479	0.517049
1000	1.069393	3.089026

실행시간(buy sell)		
client	thread	select
1	0.032301	0.001011
5	0.035464	0.004423
10	0.045082	0.012079
50	0.077351	0.041925
100	0.140565	0.117496
500	0.428449	0.489781
1000	1.109562	3.155594

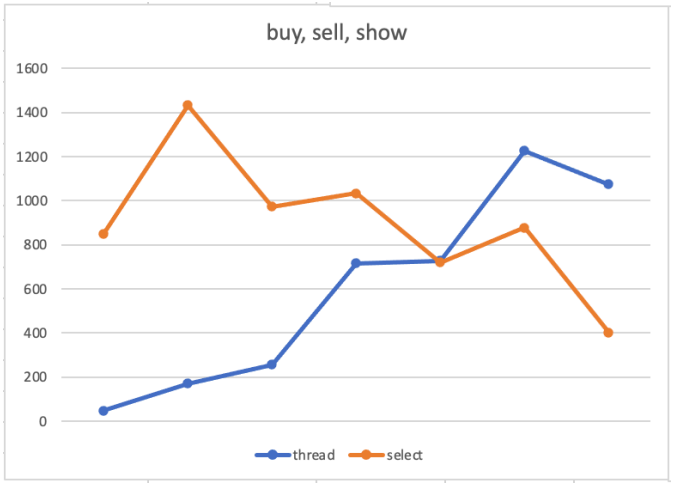
동시처리율(혼합)		
client	thread	select
1	49.28293332	850.3401361
5	171.5148189	1431.844215
10	257.2148773	973.3307378
50	716.2502865	1034.083388
100	726.9290881	720.2639047
500	1225.814676	878.5524267
1000	1074.923224	402.4156205

동시처리율(show)		
client	thread	select
1	33.05348053	685.8710562
5	147.1194021	1102.292769
10	236.1609673	940.8222787
50	734.2467363	1120.247351
100	705.8607619	926.5522066
500	1056.013044	967.026336
1000	935.1099175	323.7266375

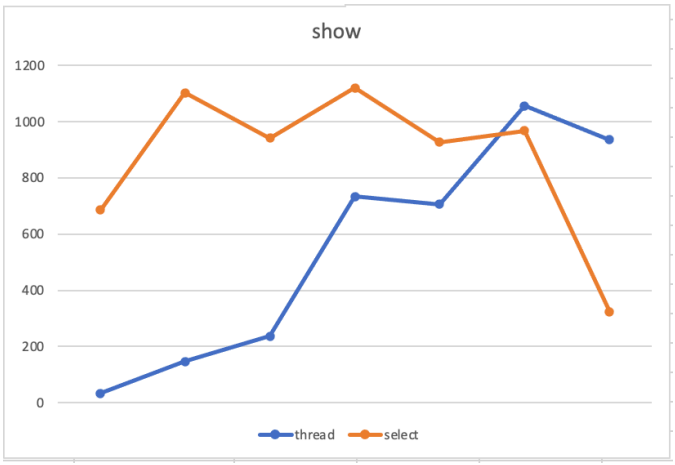
동시처리율(buy sell)		
client	thread	select
1	30.95879385	989.1196835
5	140.9880442	1130.454443
10	221.8180205	827.8831029
50	646.4040542	1192.605844
100	711.414648	851.0928032
500	1167.00004	1020.864427
1000	901.2565319	316.8975477

표를 기반으로 다음과 같은 그래프를 그릴 수 있다.

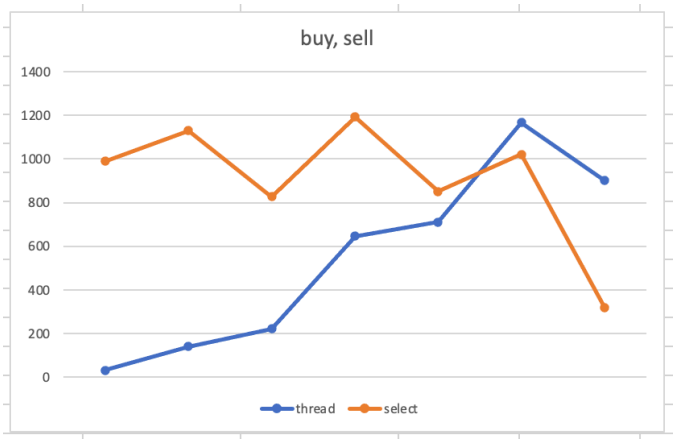
1) 3개의 명령어 모두 사용하였을 때



2) Show 명령어만 사용했을 때



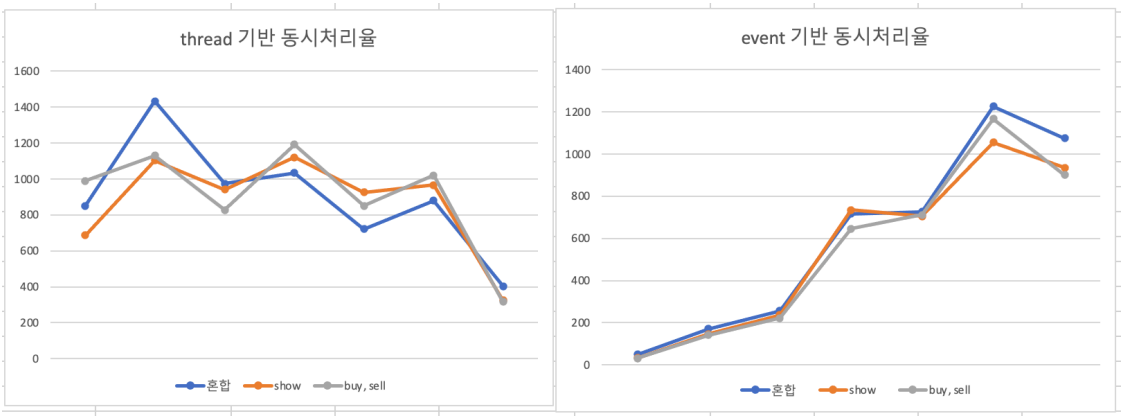
3) Buy, sell 명령어만 사용했을 때



3가지 경우의 결과를 보았을 때 대체적으로 비슷한 그래프의 형태를 보인다는 것을 알 수 있었다. Event driven approach의 경우 client 수가 증가할수록 동시처리율이 점점 감소한다는 것을 알 수 있었으며 thread based approach는 이와 반대로 client수가 증가할수록 동시처리율이 증가한다는 것을 알 수 있었다.

수업 시간에 배운 내용을 통해 thread 기반의 approach가 동시처리율이 좋을 것이라고 예측했고, 나의 예측과 비슷한 결과를 볼 수 있었다.

두번째로 같은 approach에서 각 명령어에 대한 동시처리율을 비교해보았다.



Show 명령어는 read, buy와 sell은 write, 혼합은 read와 write 둘 다 한다고 볼 수 있다. 이에 대하여 비교를 해보았을 때 세 방법 모두 비슷한 결과를 볼 수 있었다.

실행 시간을 측정할 때 컴퓨터 속도, 외부적인 요인들 때문에 정확한 측정이 어려울 것이라 생각했다. 따라서 여러번 측정하여 실생 시간 평균을 구하여 분석에 사용하였다.

하지만 측정했던 client 테스트케이스가 적기도 했고, 실행 시간 평균을 구하기 위한 측정도 테스트 한번에 3번밖에 하지 않았으며, 분석하기 위한 코드가 완벽한 상태가 아니었기 때문에 지금 구한 결과가 절대적인 결과라고 보기에는 어려울 것으로 보인다. 조금 더 명확한 testcase와 프로그램이 있었다면 이론에 가까운, 그리고 더욱 정확한 결과를 얻을 수 있지 않았을까라는 생각이 들었다.