System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 김민선

학번 : 20181603

1. 개발 목표

주식 서버는 다수의 client들이 접속했을 때, 각 client의 요청을 동시에 처리해줄 수 있어야 한다. 이 때 만약 주식 서버가 iterative하게 구현되어 있다면, 주식 서버는 한 번에 한 client와의 connection에 대해서만 request를 처리할 수 있다. 따라서 다수의 clients와 connection을 형성한 채로 여러 request를 동시에 처리하기 위해서는 주식 서버를 concurrent하게 구현할 필요가 있다.

본 프로젝트에서는 미니 주식서버인 stockserver가 concurrent하게 운영되도록 하기 위해 두 가지 방식을 이용하여 설계 및 구현한다.

1) select()함수를 이용한 Event-based approach

2) pthread에 대한 함수를 이용한 Thread-based approach

2. 개발 범위 및 내용

A. 개발 범위

1. select

select함수를 통해 event-based approach로 concurrent서버를 구현할 수 있다. 이 때 서버는 다수의 client들과의 connection을 위해 connfd를 위한 하나의 배열을 생성하고, 이를 통해 각 client들의 request를 concurrent하게 처리한다.

2. pthread

pthread_create, pthread_detach 등의 pthread 계열 함수를 통해 thread-based approach로 concurrent 서버를 구현할 수 있다. 이 때 서버는 다수의 client들과의 connection을 위해 여러 개의 thread를 생성하고, 한 thread마다 하나의 connection을 할당하여 각 client들의 request를 concurrent하게 처리한다.

B. 개발 내용

1. select

✓ **select 함수로 구현한 부분에 대해서 간략히 설명**

concurrent 서버를 구현하기 위해 fd_set 타입의 read_set과 ready_set등의 set을 구조체 pool로 유지하였다. 이 때 read_set은 일종의 bit vector라고 생각할 수 있다. 즉, 어

떤 descriptor k가 read해야할 대상이라면 해당 set의 k번째 index의 값을 1로 세팅하면 되는 것이다. ready_set은 read_set의 부분집합으로, ready for reading상태가 된 read_set의 descriptors로 구성되어 있다.

select 함수는 이러한 fd_set 타입의 set을 조작해주는 함수이며, 하나의 read_set이 reading 작업에 대한 준비가 완료될 때까지 block되어있다. 따라서 open_listenfd를 통해 listenfd를 얻어온 후에는 select를 호출하여 listenfd가 ready for reading 상태가 되기 전까지 block시키고, ready for reading 상태가 되면 accept를 호출하여 connfd를 얻어올 수 있도록 하였다.

event-based의 경우 stockserver의 main함수에서 중점적으로 처리해야 할 부분은 1) 새로운 client로부터 connection request가 도착했는지 모니터링, 2) 이미 연결된 client와의 connected descriptor가 ready for reading 상태인지 확인하는 부분이다. 이 중 1)은 앞서 설명한 바와 같이 select와 accept 함수를 통해 처리할 수 있다. 2)에 대한 부분은 이후 check_client()라는 함수를 호출하여 ready for reading 상태의 active clients들에 대해 실제로 echo작업을 수행함으로써 처리할 수 있다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

stock.txt 파일의 내용을 담아두기 위해 item이라는 구조체를 선언하였다. item에는 id(주식 ID), left_stock(잔여 주식량), price(주식 가격), left, right(트리에서 자식노드를 가리킬 포인터)가 선언되어 있으며, 각 주식 종목들은 하나의 트리 구조로 저장된다.

본 프로젝트에서는 서버 실행 시 read_file()이라는 함수를 호출하도록 하였다. read_file()에서는 stock.txt파일을 한 줄씩 읽고, 각 줄에 있는 stock ID, left_stock, price를 item에 저장한다. 만약 현재 읽어들인 item을 cur, 이전에 읽어 이미 트리에 저장된 item을 prev라고 했을 때, cur의 id가 prev의 id보다 작으면 prev의 left로 연결되고, 크면 prev의 right으로 연결된다. stock.txt 파일의 마지막 줄까지 이를 반복함으로써 stock.txt의 내용을 트리 구조로 메모리 상에 올려둘 수 있다.

2. pthread

✓ pthread로 구현한 부분에 대해서 간략히 설명

Thread-based approach를 이용해 concurrent 서버를 구현하기 위해 pthread 계열의 함수들을 사용하였다. 이 때 새로운 connfd가 얻어질 때마다 pthread를 create하고, connection이 끝날 때마다 thread를 반납해주는 행위를 반복하면 비효율적이다. 따라서 pthread_create를 통해 일정 개수의 thread를 미리 create하여 pool에 저장해둔 뒤, 새로

운 connection이 이루어질 때마다 아직 conncection이 이루어지지 않은 thread에 할당해 주도록 하였다. 이를 위해 connection request를 보내는 client들의 connfd를 buffer에 저장해두고, buffer에 새로운 connfd가 들어올 때마다 thread가 하나씩 이를 pop하여 conncection을 형성함으로써 해당 client와의 request를 처리한다. 이 때 buffer가 비어있는 경우 pop을 하는 것을 방지하고, buffer가 꽉 찼을 때 connfd가 저장되는 것을 막기 위해 slot과 item에 대한 counting semaphore를 설정하고, buffer의 맨 처음과 맨 마지막을 가리키는 front와 rear변수에 대한 mutex lock을 설정한다. 또한, 각 thread 종료 시 자동으로 reaped되게 하기 위해 각 thread는 detached mode로 돌 수 있도록 하였다.

C. 개발 방법

<csapp.h>

1) item에 대한 구조체 및 전역변수

stock.txt의 내용을 트리 구조로 저장하기 위해 item에 대한 구조체를 선언하였다. item의 내부에는 id(stock ID), left_stock(잔여 주식량), price(주식 가격), left, right(자식 노드를 가리키는 포인터)를 선언하였다. 또한 item 타입의 stock_table 포인터를 선언하여 트리의 root를 가리키도록 하였고, item_num은 stock_table이라는 트리의 총 노드의 수(주식 종목의 수)를 저장한다.

또한, 현재 stock_table을 read하는 reader의 수를 readcnt로 저장하고, stock_table을 read하는 경우에 대한 semaphore로 mutex, write하는 경우에 대한 semaphore로 w를 선언하였으며, 이 때 mutex와 w는 binary semaphore이다.

2) event-based에서의 pool

```
typedef struct{
    int maxfd; //read_set의 가장 큰 descriptor
    fd_set read_set; //모든 active desciptors의 set
    fd_set ready_set; //ready for reading 상태인 read_set의 descriptors의 subset
    int nready; //select를 통해 얻어진 ready descriptors의 수
    int maxi; //clientfd의 가장 큰 index
    int clientfd[FD_SETSIZE];    // active desciptors의 set
```

```
        rio_t clientrio[FD_SETSIZE];  // active read buffers
    } pool;
```

3) thread-based에서의 pool

```
typedef struct {
        int *buf; //connection request를 보낸 clients들의 descriptors가 저장되는 buffer
        int n; //빈 slot의 최대 개수
        int front; //buf의 시작 index. buf[(front+1)%n]이 first item이다.
        int rear; //buf의 끝 index . buf[rear%n]이 last item이다.
        sem_t mutex; //buf로의 access를 protect해주는 binary semaphore
        sem_t slots; //빈 slot의 개수를 측정하기 위한 counting semaphore
        sem_t items; //buf에 있는 item의 개수를 측정하기 위한 counting semaphore
} sbuf_t;
```

< stockserver.c>

1) event based approach

서버가 실행되면, main함수에서는 read_file()을 호출하여 stock.txt의 내용을 stock_table에 저장하여 메모리에 상주시킨다. 이 stock_table에 대한 semaphore인 mutex와 w를 1로 초기화하고, readcnt는 0으로 초기화한다. 이후 init_pool()을 호출하여 pool을 초기화한 후 open_listenfd()를 통해 listenfd를 얻는다.

다음으로는 loop가 수행된다. loop에서는 select함수를 호출하여 listenfd가 ready for reading상태가 될 때까지 block시킨다. listenfd가 ready for reading 상태가 되면 accept() 함수를 통해 connfd를 얻고, add_client()를 통해 connfd들을 pool에 추가시킨다. 이후 서버에서는 check_client()함수를 호출하여 실질적인 client request를 처리한다. 이후 해당 client와의 connection이 종료되면 현재 stock_table의 내용을 stock.txt 파일에 써주기 위해 write_file()함수를 호출한다.

2) thread-based approach

stock.txt의 내용을 메모리에 상주시키는 과정은 event-based와 같다. 이후 listenfd를 얻고, sbuf_init()을 호출하여 sbuf의 내용을 초기화시킨다. 이후 connection을 형성하여 request를 처리할 thread들을 미리 create해둔다.

다음으로는 loop가 수행된다. loop에서는 accept()함수를 통해 connfd를 얻고, sbuf_insert()함수를 통해 해당 connfd를 sbuf에 저장해둔다. 앞서 pthread_create()를 통해 미리 create된 thread들은 sbuf가 비어있다면 잠시 block되어 있지만, sbuf에 item이 존재하면(connfd) 해당 connfd와 연결을 형성하여 실질적인 client request 처리를 시작한다. 이후 해당 client와의 connection이 종료되면 현재 stock_table의 내용을 stock.txt 파일에 써주기 위해 write_file() 함수를 호출한다.

<echo.c>

echo()함수는 client로부터 온 메시지(buf)에 대한 처리 작업을 수행해주고, 결과 메시지(result)를 다시 client에게 보내준다. echo에서 처리해야 할 메인 작업은 1)buf를 parse하여 command를 추출해주는 작업과 2)command를 실행시키는 작업이다. 1)의 경우, buf를 ' '와 '₩n'를 기준으로 parse해주고, 이 때 가장 앞에 있는 단어가 command이다. command로는 show, buy, sell, exit이 올 수 있다. command가 show일 경우 show()함수를 호출하여 stock_table의 내용을 읽어들인다. command가 buy일 경우 buy()함수를 호출하여 stock_table에서 해당 id를 search한 뒤, 해당 stock의 잔여량을 감소시킨다. command가 sell일 경우 sell()함수를 호출하여 stock_table에서 해당 id를 search한 뒤, 해당 stock의 잔여량을 증가시킨다. command가 exit일 경우 connection을 종료할 수 있도록 한다.

Event-based에서는 stockserver.c의 check_clients()함수에서 rio_readnb(), echo(), rio_writen()이 순서대로 호출된다. 따라서 check_clients()에서는 rio_readnb()를 통해 buf를 받고, 이를 echo()에 parameter로 넘기게 된다. echo()에서는 buf에 대해 처리하고, result를 형성하여 반환해주는 형식으로 구현한다. echo()에서 반환된 result는 check_clients()의 rio_writen()을 통해 client로 전달될 것이다.

반면 Thread-based에서는 echo()함수 내에서 rio_readnb(), buf에 대한 처리, result 형성, rio_writen()이 모두 이루어진다는 점에서 Event-based와 차이가 있다.

<multiclient.c

client가 서버로 보낸 request를 client의 화면에 출력할 수 있도록 하였다. 예를 들어, client가 서버로 'show'라는 request를 요청하면, client는 자신의 화면에 'show'를 출력하도록 한다.

또한 client가 서버로 show request를 보낼 경우, 서버는 stock_table의 내용을 여러 줄로 보내주기 때문에 client는 rio_readlineb가 아니라 rio_readnb를 호출해주어야 한다.

<stockclient.c>

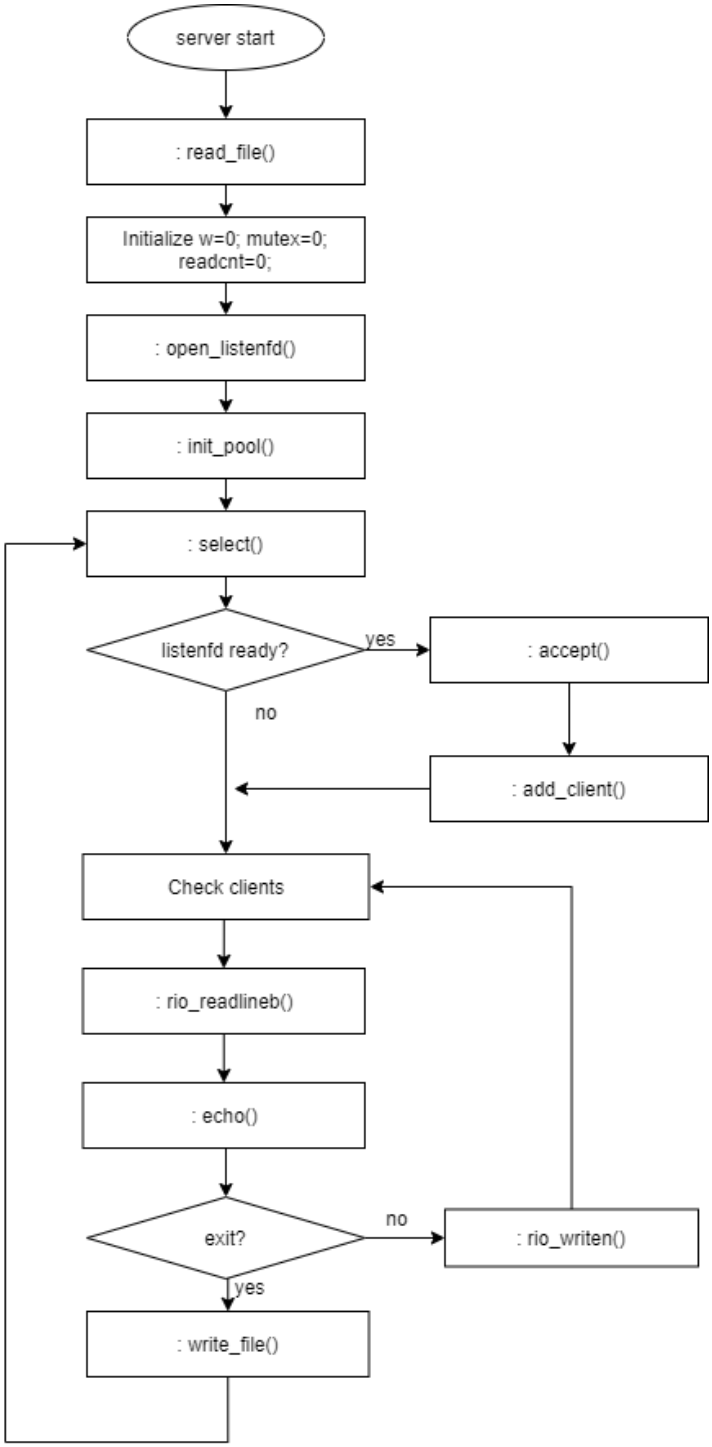
client가 서버로 show request를 보낼 경우, 서버는 stock_table의 내용을 여러 줄로 보내주기 때문에 client는 rio_readlineb가 아니라 rio_readnb를 호출해주도록 하였다.

또한, client가 exit request를 보낸 경우, 서버는 client에게 다시 exit을 반환해주게 되는데, 이 때 client는 자신의 connection을 끊어주는 작업을 수행해주어야 한다. 따라서 서버로부터 받는 메시지가 "exit"일 경우, client는 while루프를 빠져나와 자신의 clientfd를 close해줄 수 있도록 한다.

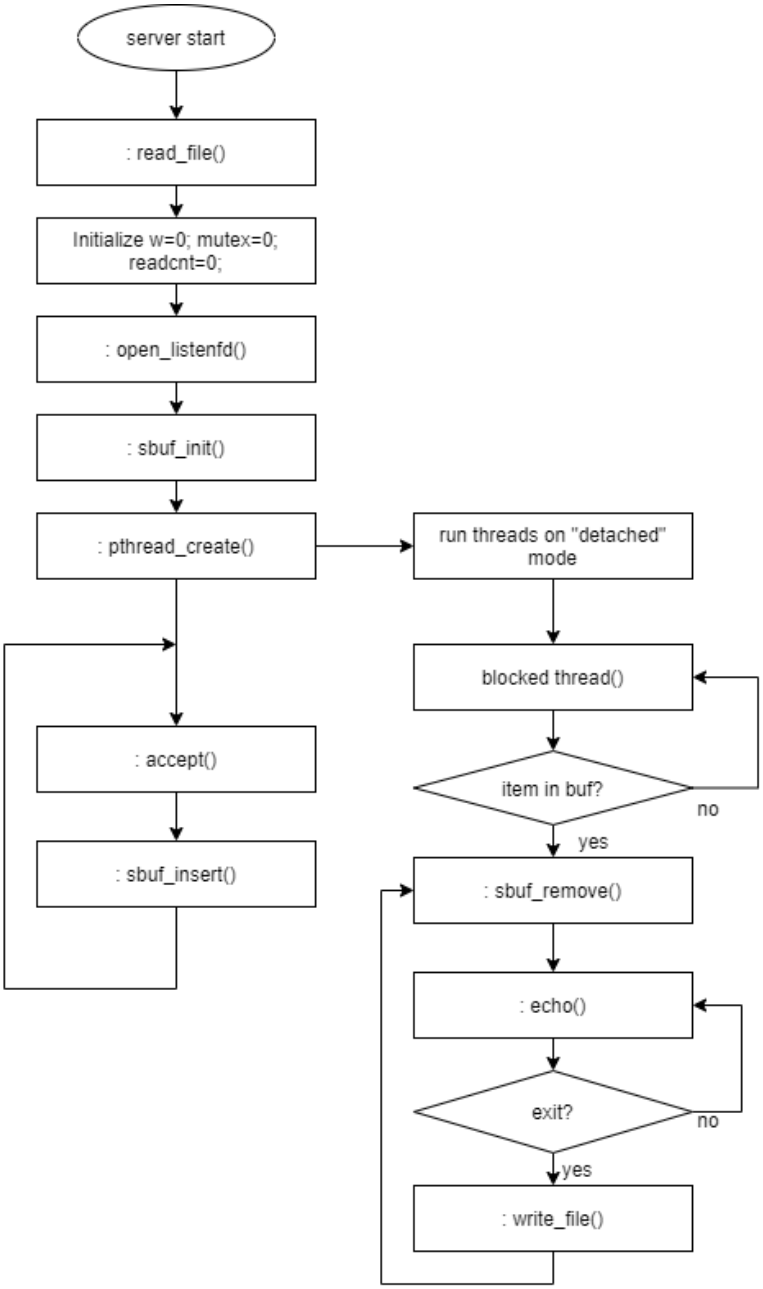
3. 구현 결과

A. Flow Chart

1. select


server start
: read_file()
Initialize w=0; mutex=0;
readcnt=0;
: open_listenfd()
: init_pool()
: select()
listenfd ready?
yes
: accept()
no
: add_client()
Check clients
: rio_readlineb()
: echo()
exit?
no
: rio_writen()
yes
: write_file()

2. pthread



B. 제작 내용

1. select

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

이 함수는 pool을 초기화하는 역할을 수행한다. 우선 pool의 maxi를 -1로 초기화하고, pool의 모든 clientfd[i]의 값 역시 -1로 초기화하여 비어있음을 나타낸다.

또한, FD_ZERO 매크로를 통해 read_set의 descriptor 값들을 전부 0으로 초기화시

킨다. 이 때, 초기에는 listenfd만이 read_set의 멤버이므로 pool의 maxfd를 listenfd(3)으로 지정하고, FD_SET() 매크로를 통해 read_set의 listenfd번째 descriptor를 1로 세팅해준다.

(2) void add_client(int connfd, pool *p)

이 함수는 새로운 client connection을 pool에 추가해주는 작업을 수행한다. pool의 clientfd를 탐색하면서 비어있는 clientfd를 찾는다. 이 때 clientfd[i] == -1인 경우가 비어있는 경우이다. 만약 빈 clientfd를 찾는다면, 해당 clientfd[i]에 connfd 값을 저장하고, 해당 connfd에 대한 buffer(clientrio[i]) 역시 마련해둔다. 이후 FD_SET() 매크로를 통해 해당 connfd를 read_set의 멤버로 추가한다. 이 때 만약 connfd의 값이 pool의 maxfd보다 크다면, maxfd에 connfd를 저장한다. 또한, 만약 현재 i의 값이 pool의 maxi보다 크다면, maxi에 i를 저장한다.

(3) check_clients(pool *p)

이 함수는 ready for reading상태의 client들의 실질적인 connection을 담당하여 request를 처리해주는 함수이다. ready_set의 멤버 descriptors에 대해, readlineb()를 호출하여 client로부터 request를 받는다. 이 때 만약 request가 존재한다면 echo()함수를 호출하여 request를 처리한 후, rio_writen()을 통해 결과 메시지를 client에게로 보내준다. 반면, client로부터 온 request가 없는 경우, client가 연결을 종료하는 것으로 간주하여 해당 client와의 connection을 종료하고, write_file()함수를 호출하여 현재의 stock_table을 stock.txt에 update해준다.

2. pthread

(1) void sbuf_init(sbuf_t *sp, int n)

이 함수는 n개의 slots을 갖는 버퍼를 만들어주는 역할을 수행한다. 우선 parameter로 받은 sp에 대하여 sp->buf에 n만큼의 slots을 할당해준다. 초기에 버퍼의 시작과 끝은 같으므로 sp->front=sp->rear=0으로 초기화해준다. 버퍼에 대한 접근을 제어하는 mutex lock은 1로 초기화하고, 초기에는 모든 slots이 비어있으므로 slot semaphore는 n으로, item semaphore는 0으로 초기화한다.

(2) void sbuf_deinit(sbuf_t *sp)

이 함수는 parameter로 받은 sp에 대하여, sp->buf의 메모리를 해제시켜주는 역할을 수행한다.

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

이 함수는 parameter로 받은 sp에 대하여, sp->buf의 rear에 item을 삽입해주는 역할을 수행한다. 이 때 주의할 점은, buf가 꽉 찬 경우 item이 더 이상 삽입되지 않도록 하기 위해 slots == 0이면 block될 수 있도록 구현해주어야 한다. 만약 slots>0 이라면 slots의 값을 1만큼 감소시킨 후 다음 코드를 수행할 수 있도록 한다. 또한, item을 buf에 삽입하는 동안에는 buf에 대해 다른 작업이 수행되지 않도록 하기 위해 mutex lock으로 'item 삽입을 수행하는 영역(critical section)'을 감싸준다. 이후 buf에 item이 하나 추가되었음을 알리기 위해 items의 값을 1만큼 증가시킨다.

(4) int sbuf_remove(sbuf_t *sp)

이 함수는 parameter로 받은 sp에 대하여, sp->buf의 front에서 item을 뽑아오는 역할을 수행한다. 이 때 주의할 점은, buf가 비어있는 경우 item을 빼갈 수 없도록 하기 위해 items==0이면 block될 수 있도록 구현해주어야 한다. 만약 items>0이면 items의 값을 1만큼 감소시킨 후 다음 코드를 수행할 수 있도록 한다. 또한, item을 buf에서 pop하는 동안에는 buf에 대해 다른 작업이 수행되지 않도록 하기 위해 mutex lock으로 'item 삽입을 수행하는 영역(critical section)'을 감싸준다. 이후 buf에 slot이 하나 늘어났음을 알리기 위해 slots의 값을 1만큼 증가시킨다.

C. 시험 및 평가 내용

1. 예측을 통한 결과 분석

1) Event-based(select)와 Thread-based(pthread)의 성능 비교

Thread-based의 경우, client request가 증가함에 따라 concurrent하게 처리할 수 있는 request양도 함께 증가할 것이므로 동시처리율 역시 점점 증가할 것이라고 예측할 수 있다. 다만, CPU의 개수는 제한되어 있기 때문에 어느 순간부터는 증가하는 비율이 점점 줄어들다가, 동시처리율이 증가하지 않는 구간이 나타날 것이다.

Event-based의 경우 역시 client request가 증가함에 따라 concurrent하게 처리할 수

있는 request양도 함께 증가할 것이므로 동시처리율 역시 점점 증가할 것이라고 예측할 수 있다. 다만, Event-based는 multi-core의 장점을 활용하지 못하고 1개의 CPU에 대한 concurrent programming만 수행한다. 따라서 어느 순간부터는 증가하는 비율이 점점 줄어들고, thread-based와 비교했을 때 동시처리율이 감소하는 구간이 나타날 수도 있을 것 같다.

2) write만 한 경우, read만 한 경우, read&write을 섞어서 한 경우에 대한 성능 비교

서버 측에서는 write에 대한 request를 한 번에 하나씩만 수행할 수 있다. 이는 여러 client가 동시에 stock_table을 write할 경우, 어떤 client가 어떤 순서로 write하느냐에 따라 의도치 않은 결과가 나오는 race문제가 발생할 수도 있기 때문이다. 따라서 write을 수행하기 위해서는 한 번에 하나의 client request만 처리해주어야 한다. 반면, read는 여러 client가 동시에 수행할 수 있다. 이는 read작업은 stock_table의 내용 변화에 아무런 영향도 미치지 않기 때문이다. 따라서 동시처리율은 read>read&write>write순으로 높을 것이라고 예상할 수 있다.

2. 실제 실험을 통한 결과 분석(그래프 포함)

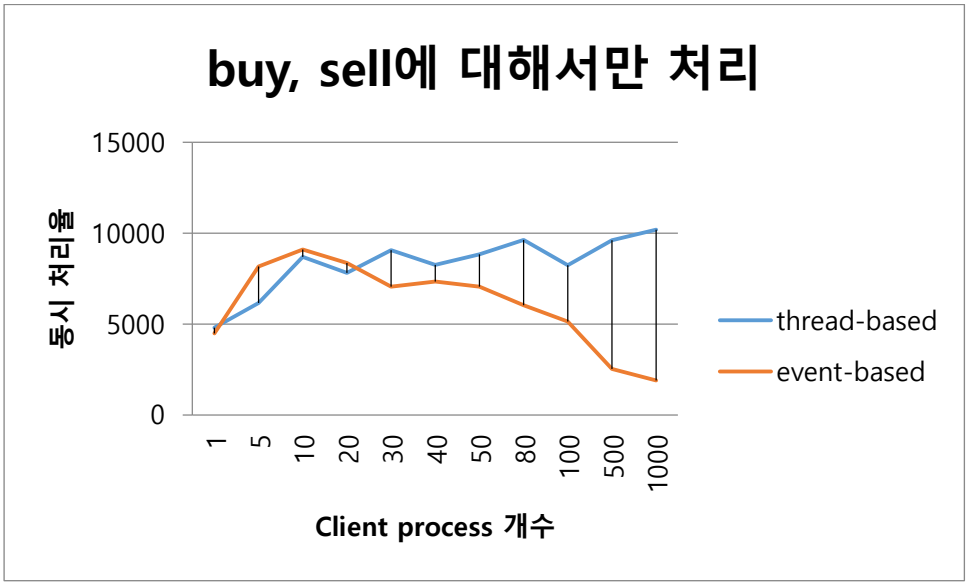
표와 그래프에 대해 분석하기에 앞서, 성능 테스트를 수행한 조건은 다음과 같다.

- ✓ 한 CLIENT 당 요청한 request 수는 10개이다.
- ✓ STOCK_NUM은 5개로 설정하였다.
- ✓ BUY_SELL_MAX는 10개로 설정하였다.
- ✓ 한 CLIENT 당 요청한 request 수는 10개이므로 'client의 처리 요청 개수'는 client수 *10으로 설정하였다. 따라서 동시처리율은 다음과 같이 계산하였다.

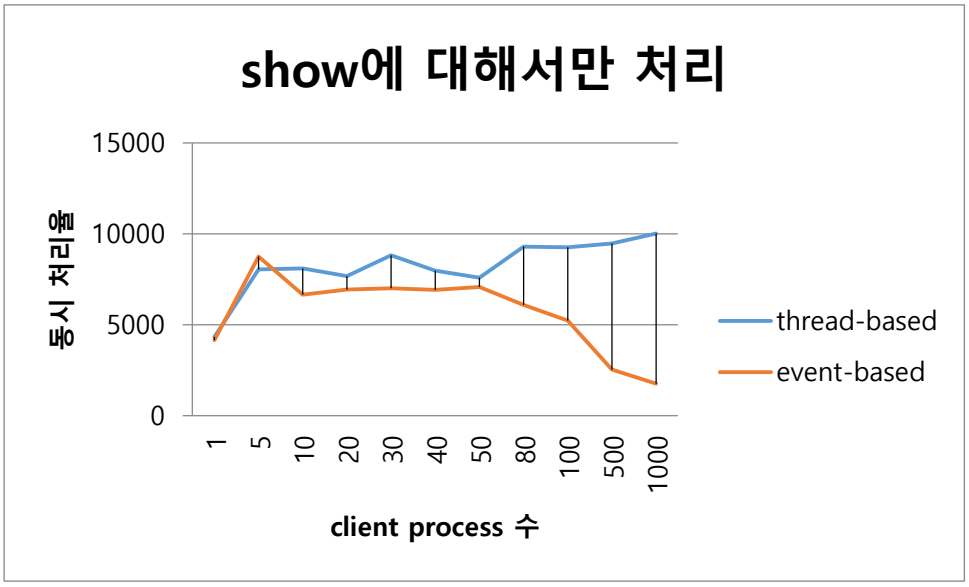
$$\text{동시처리율} = \frac{client\text{수} \times 10}{\text{걸린시간(초)}}$$

1) Event-based (select)와 Thread-based(pthread)의 성능 비교

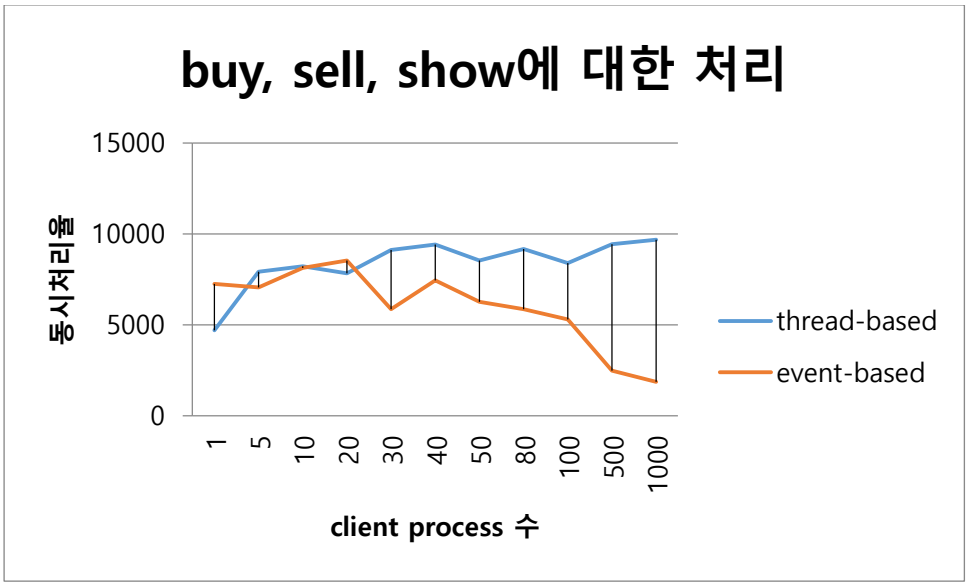
1. buy, sell만 test한 경우					
client수	request수	thread		event	
		시간	1초당 처리 request량	시간	1초당 처리 request량
1	10	0.002081	4805.382028	0.002227	4490.345757
5	50	0.008107	6167.50956	0.006118	8172.605427
10	100	0.011485	8707.009142	0.010981	9106.63874
20	200	0.025598	7813.110399	0.023899	8368.550985
30	300	0.033081	9068.649678	0.042498	7059.155725
40	400	0.048447	8256.445188	0.05449	7340.796476
50	500	0.056533	8844.391771	0.070808	7061.349
80	800	0.083073	9630.084384	0.132523	6036.687971
100	1000	0.121266	8246.334504	0.194295	5146.812836
500	5000	0.519819	9618.732674	1.969977	2538.100699
1000	10000	0.980998	10193.7007	5.291741	1889.737234



2. show만 test한 경우					
client수	request수	thread		event	
		시간	1초당 처리 request량	시간	1초당 처리 request량
1	10	0.002312	4325.259516	0.002411	4147.656574
5	50	0.006214	8046.346958	0.005713	8751.969193
10	100	0.012339	8104.384472	0.015019	6658.232905
20	200	0.026057	7675.480677	0.028862	6929.526713
30	300	0.033988	8826.644698	0.042853	7000.676732
40	400	0.050119	7981.005208	0.057768	6924.248719
50	500	0.065875	7590.132827	0.070577	7084.460943
80	800	0.085981	9304.3812	0.13135	6090.59764
100	1000	0.107949	9263.633753	0.191537	5220.923373
500	5000	0.528157	9466.882007	1.978176	2527.580963
1000	10000	0.998806	10011.95427	5.693042	1756.530164



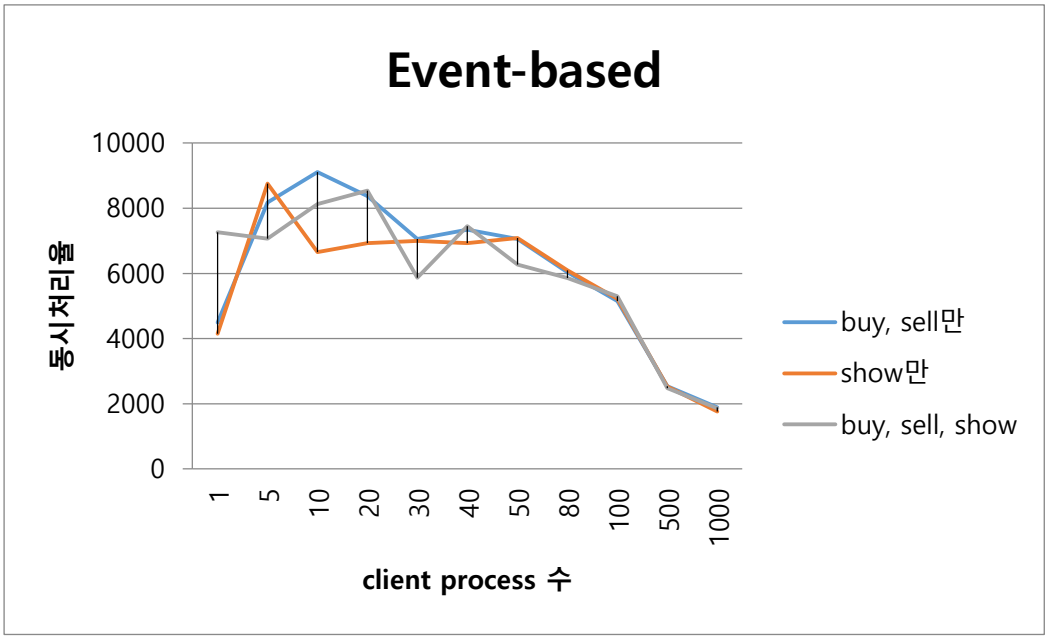
3. show, buy, sell을 섞어서 test한 경우					
client수	request수	thread		event	
		시간	1초당 처리 request량	시간	1초당 처리 request량
1	10	0.002125	4705.882353	0.001377	7262.164125
5	50	0.006312	7921.419518	0.00708	7062.146893
10	100	0.012173	8214.901832	0.012298	8131.40348
20	200	0.025522	7836.37646	0.023414	8541.89801
30	300	0.032864	9128.529698	0.051138	5866.478939
40	400	0.042487	9414.644479	0.053688	7450.454478
50	500	0.05857	8536.79358	0.079732	6271.007876
80	800	0.087266	9167.373318	0.136529	5859.560972
100	1000	0.119031	8401.172804	0.18875	5298.013245
500	5000	0.529255	9447.241878	2.020017	2475.226694
1000	10000	1.033416	9676.645223	5.363585	1864.424634



첫 번째 경우는 client가 write(buy와 sell)하는 request만을 요청했을 경우, event-based와 thread-based의 성능을 표와 그래프로 나타낸 것이다. 두 번째 경우는 client가 read(show)하는 request만을 요청했을 경우, event-based와 thread-based의 성능을 표와 그래프로 나타낸 것이다. 세 번째 경우는 client가 read&write하는 request를 요청했을 경우, event-based와 thread-based의 성능을 표와 그래프로 나타낸 것이다.

위 표와 그래프를 살펴봤을 때, 정확한 값에는 차이가 있었으나 대체로 세 가지 경우 모두 비슷한 형태의 그래프를 나타내고 있음을 확인할 수 있다. Thread-based의 경우, client process수가 증가함에 따라 동시 처리율이 대체로 증가하는 양상을 보이고 있으며, 초반에 비해 후반 증가율이 약간은 감소하였다. Event-based의 경우, 초반에는 client process수가 증가함에 따라 동시 처리율이 증가하는 양상을 보였으나, 어느 순간부터는 client process 수가 증가할수록 동시 처리율이 감소하였다.

2) write만 한 경우, read만 한 경우, read&write을 섞어서 한 경우에 대한 성능 비교



첫 번째는 서버가 thread-based로 운영될 때, read, write, read&write에 대한 성능 비교를 그래프로 나타낸 것이다. 두 번째는 서버가 event-based로 운영될 때, read, write, read&write에 대한 성능 비교를 그래프로 나타낸 것이다.

위 그래프를 살펴봤을 때, read만 한 경우, write만 한 경우, read&write만 한 경우에 사이에 유의미한 차이는 나타나지 않는 것으로 보인다. 따라서 'read>read&write>write 순서로 동시처리율이 높게 나타날 것이다'라는 예측과는 달리 실제 test에서는 세가지 경우의 동시처리율을 명확하게 비교할 수 없었다.