System Programming Project 5

담당 교수 : 김영재

이름 : 임정호

학번 : 20171682

1. 개발 목표

여러 client들의 동시 접속 및 서비스를 위한 Concurrent stock server를 구축한다. 주식의 정보는 이진트리로 저장한다. client는 4개의 명령어를 실행할 수 있는데, show, buy, sell, exit이다. show를 하면 현재 server가 관리하고 있는 종목들의 정보를 보여준다. buy는 주식을 매수하겠다는 명령어이고, sell은 주식을 매도하겠다는 명령어이다. exit는 퇴장하겠다는 명령어이고, sever와의 연결을 끊는다. 초기 상태의 주식 정보는 stock.txt에 저장되어 있다. 위 명령어를 수행하는 과정은 Event-drvien Approach와 Thread-based Approach, 2가지 방법으로 구현하여 stock sever의 concurrency를 지원할 수 있게 한다.

2. 개발 범위 및 내용

A. 개발 범위

1. select

   여러 cliente들이 접속을 시도할 때, 서버가 하나의 client와의 접속을 유지할 때까지 다른 client들은 접속을 못 하는 것이 아닌, server가 여ㅉ러 client들과 접속을 유지할 수 있게 한다.

2. pthread

   여러 client들이 접속했을 때, 서버가 각 client마다 미리 할당된 thread를 부여하여 각 thread마다 work routine을 실행시킨다. thread는 semaphore를 이용하여 공유 자원에 대한 concurrent한 접근을 제한한다.

B. 개발 내용

- **select**

  ✓ select 함수로 구현한 부분에 대해서 간략히 설명

connfd을 담고 있는 배열을 생성하고, -1로 초기화한다. -1이라는 의미는 비어있다는 뜻으로 client를 상대할 준비가 되어있는 상태이다. server의 main에서 Accept로 client의 connfd를 받으면 connfd에 add_client()를 호출하여 client를 추가한다. 그 다음 check_clients()를 호출하여 명령어를 요청한 client가 있는지 검사하여 있는 경우 server화면에서는 어떤 file descriptor에서 몇 바이트의 명령

어를 요청한지 출력을 하고, 요청한 명령어에 맞는 처리를 할 수 있게 한다. 그러고 요청한 클라이언트의 file descriptor 버퍼에 Rio_writen()을 이용하여 결과를 보내준다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

stock.txt을 읽어서 binary tree로 저장한다. binary tree의 노드를 구분할 수 있는 것은 주식의 종목 id이므로 id의 값으로 구분하면서 binary tree에 삽입하였다. 개장하면서 bianry tree로 관리되고 있는 주식 종목들의 재고들에 변화가 생길 것이다. 이 변화를 자주 바꾸는 것은 overhead가 있을거로 판단하여 모든 client가 접속을 종료했을 때 stock.txt를 update하는 방식으로 구현하였다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

prethreaded를 구현하기 위해 ntrheads만큼 미리 thread를 create하고, 그 thread를 sbuf가 관리할 수 있게 하였다. event와 동일하게 Accept()로 새로운 client와의 접속을 하면 connfd를 받아서 sbuf에서 일을 하지 않고 있는 thread에게 connfd를 넘겨주어 work을 부여한다. 각 thread는 func()을 실행하며 해당 client가 요청한 명령어를 확인하여, 각 명령어에 맞게 처리를 하고, 그 결과를 Rio_wrien()을 호출하여 connfd의 버퍼에 써준다.

C. 개발 방법

- **select**

✓ select 함수로 구현한 부분

init_pool():

우선 event-driven은 pool 구조로 관리되기 때문에 csapp.h에 따로 struct pool 구조체를 선언해주었다. init_pool() 함수는 client pool을 초기화해주는 함수인데, 초기 clientfd를 -1로 할당해준다. maxfd를 매개 변수로 받아온 listenfd로 설정해주고, FD_ZERO 매크로 함수를 이용해 readset에 있는 비트를 clear해주고, read_set에서 매개변수로 받아온 listenfd를 1로 바꿔준다.

add_client():

매개변수로 connfd와 pool형 변수 p를 받아와서 nready를 줄이고, for loop으로 available한 clientfd를 찾아서 connfd를 할당하고, FD_SET으로 할당된 clientfd를 read_set에서 1로 바꿔준다. maxfd와 maxi 처리를 해주고 종료한다.

감소시킨다.

check_clients():

매개 변수로 pool형 변수 p를 받아온다. for문을 돌면서 client와 연결되어 있는 p의 clientfd를 찾아 들어온 입력을 처리해준다. 요청한 명령어를 받아 각 명령어에 맞게 처리해준다. 들어오는 입력이 없으면 해당 connfd를 close해주고, read_set에서 해당 clientfd를 FD_CLR()를 이용하여 clear한다.

✓ stock info에 대한 file contents를 memory로 올린 방법'

stock.txt -> binary tree

Init()에서 stock.txt을 읽어서 그 정보를 BSTInsert()에 넘겨준다. BSTInsert()에서는 binary tree에 추가하는 일을 수행하는데, 기본적인 내용이라 특별한 부분은 없다.

binary tree -> stock.txt

stock.txt를 update하는 시기에 따라 구현하는 코드가 나뉠 것인데, 나는 모든 client가 접속을 종료할 때 update하였다. 이를 위해 전역 변수로 current_client를 두었고, client가 접속할 때 올리고, 종료될 때 1 감소시키면서, current_client가 0인 경우 update_stock_txt()을 호출하고, stock.txt를 w모드로 열어, 그 파일 포인터를 traverse()에 매개변수로 넘겨서 stock.txt를 update하였다.

- **pthread**

✓ pthread로 구현한 부분

우선 event-driven approach와 다르게 여러 thread가 전역변수를 공유하기 때문에 이에 대한 접근 처리를 할 필요가 있다. 이를 위해 semaphore를 사용하였고, 각 노드에 sem_t 변수 2개를 놓았고, 하나는 node의 정보를 수정할 때 사용하고, 나머지 하나는 show() 수행을 할 때, reader-writer의 문제를 해결하기 위해 사용하였다. prethread 구조를 위해 미리 thread를 create하는데, 이 thread에게 부여할 관리할 배열 sbuf가 필요하다. sbuf를 생성하여 새로운 client가 들어오면 sbuf

에서 쉬고 있는 thread에게 일을 부여할 수 있게 한다.

func()

client가 server에 요청한 명령어를 수행하는 함수인데, 각 thread가 client의 connfd를 부여 받아 해당 connfd의 버퍼에 출력을 한다. event-driven approach 에서 사용했던 방식과 거의 유사해서 check_clients()에 있는 부분과 거의 동일하다.

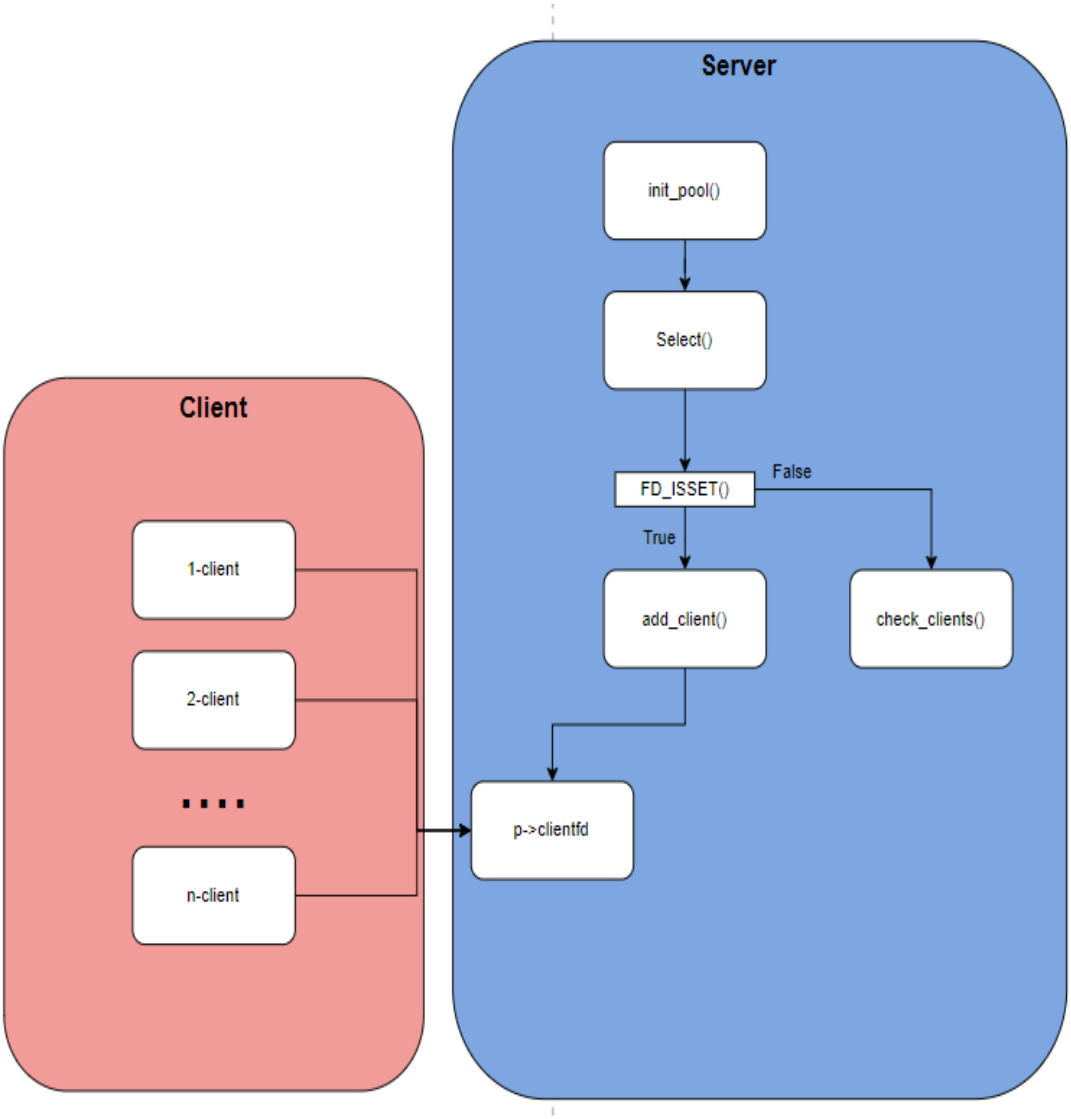
thread()

thread의 work routine이 담겨 있는 함수이다. detatch 모드를 이용하여 explict하게 reap하지 않아도 메모리 누수가 나지 않게 하였고, sbuf에서 connfd를 받으면 current_client를 1 올리고, client가 요청한 작업 처리하고 다시 current_client를 값을 1 감소시킨다. current_client가 0이면 stock.txt를 update한다.

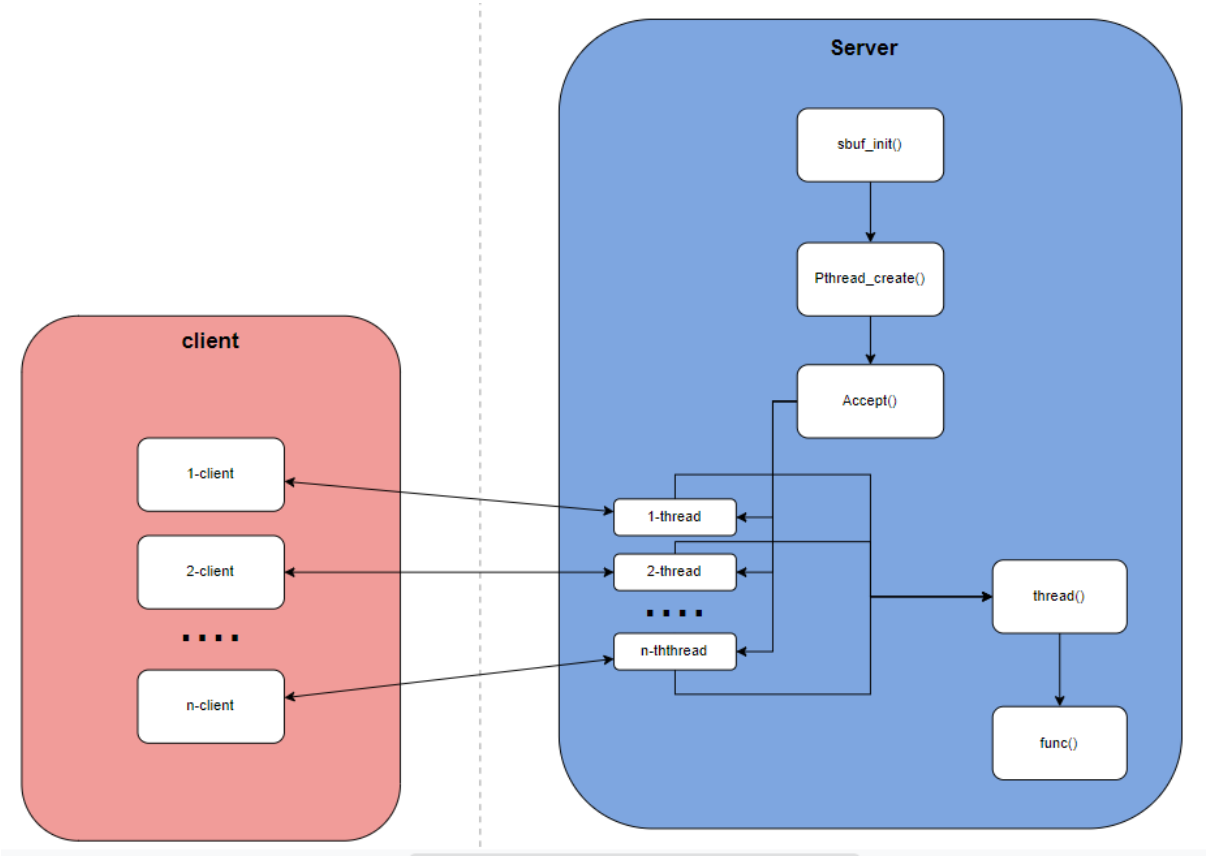
3. 구현 결과

A. Flow Chart

1. select



2. pthread



B. 제작 내용

1. select

select기반 방식에서는 thread기반 방식과는 다르게 전역 변수의 concurrent 한 접근이 없기에 thread기반에서 semaphore변수를 사용해야했던 경우의 문제는 없었다.

2. pthread

전역 변수 current_client에 여러 thread들이 동시에 접근할 수 있는 문제
전역변수로 semaphore 변수를 하나 생성해서 current_client의 값을 올리거나 내릴 때 P(&u), V(&u)로 제한하였다.

show 명령어를 수행할 때 발생할 수 있는 reader-writer문제

traverse()가 node에 접근하려 할 때, 각 node에 있는 sem_t 변수 r을 lock 걸고, readcnt를 올린다. readcnt가 1이면 해당 node의 mutex에 lock을 걸어 sell이나 buy에서 노드의 정보를 수정할 수 없게 하였고, critical section 처리 후 readcnt 값을 내리고, 0인지 확인한다. 0인 경우 앞서 lock했던 mutex변수를 unlock한다. writer보다 reader를 우선시 하였다.

prethreaded 구조에서 발생할 수 있는 consumer-producer 문제

deadlock이 발생하지 않도록, sbuf에 semaphonre변수 m, s, i 를 두어 shared buffer를 구현하였다.

sell과 buy에서 발생할 수 있는 concurrent한 접근 문제

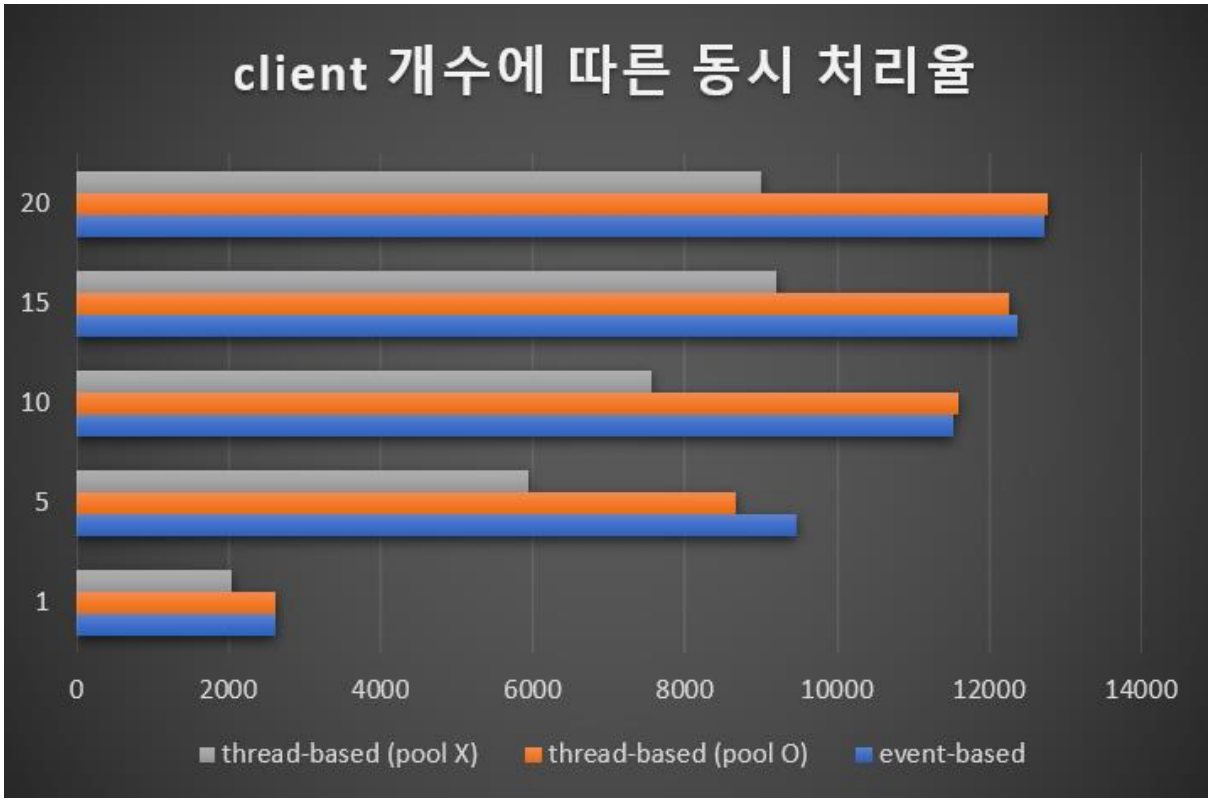
값을 수정하기 전에 해당 node의 mutex변수를 lock 걸고 critical section 작업이 종료되면 unlock하였다.

C. 시험 및 평가 내용

1. client 개수 변화에 따른 동시 처리율

실험은 event기반으로 구현한 server와 thread구조로 구현한 server 2개(pool 구조 o, x)로 진행하였고, 실험환경은 cspro.sogang.ac.kr이다. client당 10개의 requset를 서버에 요청한다. 요청하는 명령어는 show, buy, sell 중 하나의 명령어로 랜덤하게 요청된다. 각 구조당 5번씩 실행하여 그 평균 값으로 동시 처리율을 계산하였다.

실험하기 전에 우선 pool로 구현된 thread 서버가 pool로 구현하지 않은 thread 서버보다는 실행 시간이 빠를 것으로 예측하였다. 그 다음 thread를 create하고 destroy하는 task는 fork()보다는 아니지만 cost가 있는 작업이라고 수업시간에 교수님께서 설명하셨다. event 기반으로 구현된 서버와 prethreaded 구조의 서버를 비교했을 때 event기반으로 구현된 서버가 더 빠를 것으로 예측했다. 그 이유는 thread를 생성하는 overhead 차이와 semaphore 작업이 cost가 있는 수행이라고 수업시간에 들었기 때문이다. 실험 결과는 다음과 같다.



y축은 client의 개수이고, x축은 동시 처리율이다.

동시처리율은 (클라이언트 수 * 10)/실행기간(sec)로 처리하였다. 우선 thread-based에서 pool을 이용한 구조와 이용하지 않은 구조의 실행 시간 차이는 유의미하게 pool을 이용한 구조가 빨랐다. event와 pretreaded의 동시 처리율은 예측했던 것과 다소 달랐는데, client개수가 작을 때는 event가 동시 처리율이 근소하게 앞섰지만, 클라이언트 개수가 커질수록 역전되었는데, 크게 유의미한 차이는 보이지 않았다. stock.txt를 언제 update하는지가 prethreaded와 event의 유의미한 실행시간 차이에 영향을 줄 것으로 생각한다.

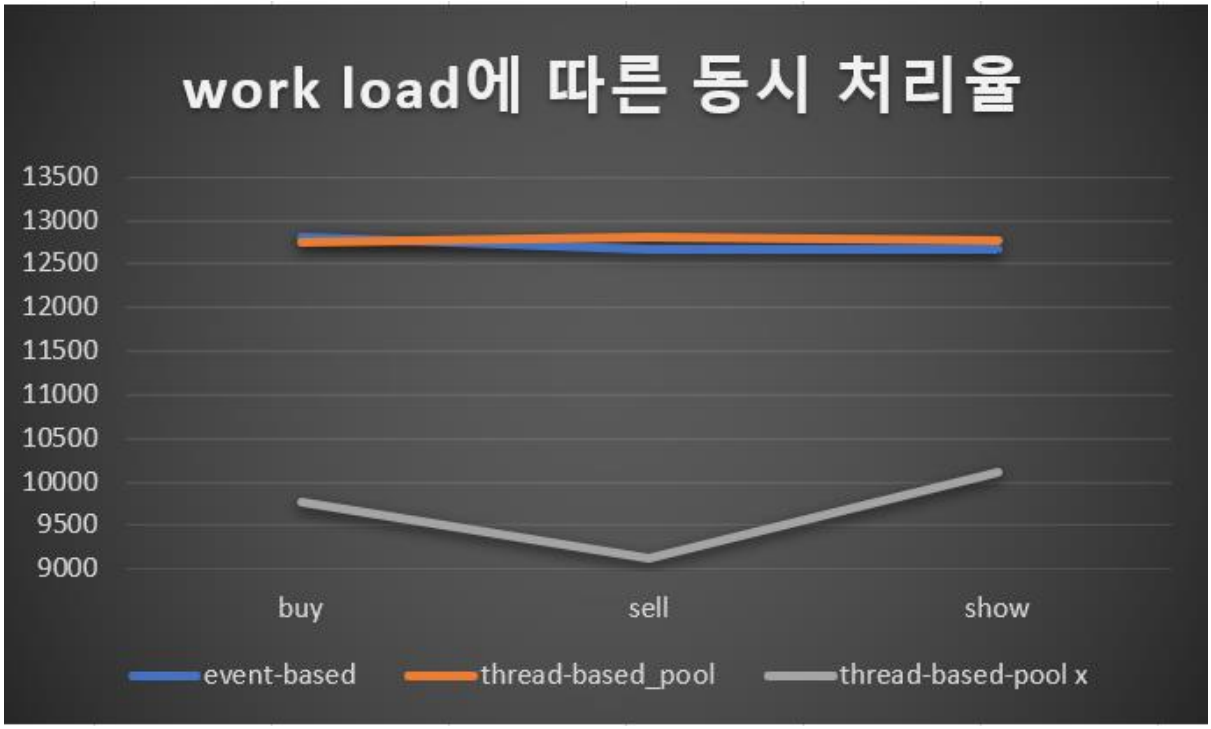
2. work load 변화에 따른 동시 처리율

실험은 event기반으로 구현한 server와 thread구조 (pool O, pool X) 2가지 방식으로 구현한 server로 진행하였고, 실험환경은 cspro.sogang.ac.kr이다. client는 20으로 수행했고, client당 10개의 requset를 서버에 요청한다. show만 요청하는 경우와 buy만 요청하는 경우, sell만 요청하는 경우로 나눴고, event 기반구조와 thread 기반 구조 각 경우마다 5번씩 실행하여 그 평균 값으로 동시 처리율을 계산하였다.

실험결과를 예측하기 전 고려해야할 사항이 하나 있는데, show 명령 수행에서 노드를 탐색하는 방법과, sell, buy에서 node를 탐색하는 방식을 다르게 구현하였다. show 명령을 수행할 때는 recursive하게 binary tree를 탐색하고, sell, buy의 경우는 iterative하게 binary tree를 탐색한다. 이런 방식에서 나오는 차이가 존재할 것이고, 이 차이를 무시하

고 다른 조건을 고려해보겠다.

우선, sell, buy는 최대 O(n)의 시간복잡도를 가지지만, show는 모든 노드를 방문해야하므로 O(n)이다. 그러므로 show가 아무래도 sell, buy보다는 실행시간이 더 크게 나올 것으로 예측했다. 그리고, buy는 binary tree에 있는 개수를 파악하여, 현재 있는 주식의 재고가 client가 매수하려는 수량보다 없는 경우를 처리하므로 buy는 sell보다 근소한 차이로 수행시간이 더 걸릴 것으로 예측했다. 실험 결과는 다음과 같다.



pool을 이용하지 않는 thread 구조가 성능면에서 뒤떨어지는 것을 볼 수 있다. event와 thread-based는 유의미한 결과차이는 없었다. 조건문 하나의 차이는 유의미한 차이를 내지 않는 것을 알 수 있었다. 다만 event는 단일 프로세스의 한계가 있기 때문에, client 수가 월등히 많아지면 thread구조보다 실행시간 면에서 불리할 것으로 생각한다.

실험 표본이 각 경우당 5개 밖에 되지 않아 표본의 크기가 크지 않은 편이다. 랜덤으로 매수 혹은 매도하려는 종목의 id를 결정하는데, 결정된 종목이 binary tree의 leaf node에 위치할 수 있고, 아무래도 stock.txt를 업데이트할 때 중위순회로 binary tree를 순회하기 때문에, stock.txt에 id가 오름차순으로 저장된다. 이는 다음 번의 서버 실행 때 binary tree가 balance있는 트리가 아닌 한 쪽으로 쭉 연결되는 트리로 구축이 되고, 시간복잡도 또한 O(logN)을 보장못하고, O(n)으로 될 것이다. 이런 부분을 보완하며 더 좋은 성능을 발휘할 수 있을 것 같다.