System Programming Project 3

담당 교수 : 이영민 교수님

이름 : 김지섭

학번 : 20201572

1. 개발 목표

Event-driven approach, Thread-based approach의 두 가지 방법을 사용하여 여러 명의 사용자들이 동시 접속 및 사용할 수 있는 주식 서버를 개발하도록 한다. 사용자들은 show, buy, sell, exit 등의 명령어를 통해 서버의 메모리에 올라와 있는 주식들의 정보를 조회 및 수정할 수 있고, 서버 프로그램이 실행 시, 또는 종료 시 stock.txt에 저장된 주식들의 정보를 불러오고 저장하도록 구현하는 것이 목표이다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

각 사용자들이 접속 시 정보를 획득하여 File Descriptor들을 관리하며 I/O Multiplexing을 통해 요청된 작업을 수행하도록 하였다. 반복문 속 Select 함수를 통해 작업 요청이 들어온 File Descriptor들에 대해서만 필요한 작업을 수행하도록 하였다. 이 방식은 thread를 이용하면서 커널 측에서 생기는 overhead가 없어 사용자 수가 증가하여도 수행 시간에 큰 변화가 없다는 장점이자 단점이 되기도 하는 특성을 가진다.

2. Task 2: Thread-based Approach

각 사용자가 접속하여 작업 요청을 받을 시, 해당 작업을 수행하는 worker thread를 생성하여 요청을 처리하도록 한다. 이 방식은 thread 간의 정보 공유가 용이하고, 멀티코어 프로세서에서는 더욱 효율적으로 많은 양의 사용자들을 처리할 수 있다는 장점을 가진다. 하지만 동일한 메모리 지점에 여러 개의 thread가 동시에 접근 시, race condition에 의해 의도치 않은 결과가 발생할 수 있으므로 이를 위하여 mutex, semaphore 등의 추가적인 조치와 그에 의한 overhead가 발생한다는 단점 역시 존재한다.

3. Task 3: Performance Evaluation

두 가지 방식에 대한 성능 측정은 제공된 multiclient 프로그램을 통해 진행하도록 한다. 해당 프로그램은 여러 개의 프로세스를 생성하여 정해진 횟수만큼의 실행을 반복해, 여러 명의 사용자가 동시 접속하는 환경을 구현한다. 이 때 gettimeofday 함수를 이용하여 처리 시간을 계산하고 정리하도록 한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  Client_pool 구조체를 선언하여 연결된 Client들의 File Descriptor를 다루도록 하였다. Client가 연결 시 accept() 함수를 이용하여 read_set에 읽어들일 descriptor들을 삽입한 후, Select() 함수를 이용하여 입력이 들어온 descriptor들을 ready_set에 넣는다. 결과적으로 ready_set에 있는 file descriptor들에 대해 반복문을 통하여 처리를 하도록 하여 I/O Multiplexing을 구현하였다.

  ✓ epoll과의 차이점 서술

  Epoll 함수의 경우 커널이 직접 관리하며 이벤트가 발생한 File Descriptor들을 구조체 배열을 통하여 넘겨준다. 따라서 Select 함수를 사용할 때 반복문과 FD_ISSET을 이용하며 일일히 확인을 해주며 생기는 overhead가 발생하지 않는다는 장점이 있으며, 메모리 관리 역시 커널이 해준다는 차이점이 있다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 반복문을 통하여 새롭게 접속하는 Client들에 대해 accept() 함수로 각 Client들의 connfd를 저장하고, 해당 client의 요청을 처리해주는 worker thread를 pthread_create() 함수를 이용하여 생성해준다. 이때 client의 connfd는 thread 생성 시 인자로 넘어간 후, 각 thread의 스택에 별도로 저장하여 다른 thread의 참조에 의한 문제가 발생하지 않도록 하였다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  각각의 worker thread는 반복문을 통해 EOF가 들어올 때까지 client에게 입력을 받아 필요한 요청을 처리하며, client가 연결 해제, 또는 exit 시 connfd를 닫고 통신을 종료한다. 주식 자료구조에 접근 시 중복 참조 및 접근을 방지하기 위하여 semaphore를 이용하여 critical section에는 한 번에 하나의 thread만 접근하도록 하였다.

- **Task3 (Performance Evaluation)**

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

  성능 평가를 위한 metric은 다음과 같이 정의하였다.

$$동시\ 처리\ 효율 = \frac{총\ request\ 개수 \times 10^6}{총\ 소요\ 시간\ (\mu s)}$$

  해당 방식을 통하여 request 하나 당 평균적인 처리 시간을 구할 수 있으며, Client의 수가 늘어날 경우 효율이 얼마나 떨어지는 지 역시 측정할 수 있게 된다. 측정 방식은 gettimeofday 함수를 multiclient 프로그램의 fork 전과 모든 child process가 종료 후에 삽입하여 그 interval을 측정하여 시간을 마이크로초 단위로 구하고, 총 request 개수는 사전에 정의된 값으로 구하여 계산하는 식으로 이루어진다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  해당 프로젝트에서는 binary tree를 이용하여 stock.txt에서 읽어들인 데이터를 저장하고 사용자의 요청에 의해 값을 수정 시 O(log n) 시간 복잡도로 검색을 수행할 수 있다. 또한 모든 주식의 정보를 출력하는 show 요청의 경우 O(n) 시간 복잡도를 가지게 된다. 따라서 show 요청에 비해 buy, sell 요청이 많은 경우 상대적으로 첫 번째 구현 방식에서는 동시 처리 효율이 높아질 것으로 예상이 된다. 하지만 두 번째 구현 방식의 경우 주식 자료구조에 접근 시 thread의 접근에 semaphore에 의한 overhead가 발생하기 때문에 show 요청에 비해 buy, sell 요청은 처리 효율이 낮아질 것으로 예상된다. 2가지 다른 방식으로 구현된 stockserver를 비교하자면 I/O Multiplexing을 통해 구현한 첫 번째 방법은 multithreading을 활용하지 못하기 때문에 Thread를 통해 구현한 두 번째 방법에 비해 접속하는 client가 많아질수록 처리 효율이 낮아질 것으로 예상된다.

C. 개발 방법

우선 두 가지 구현에서 모두 사용할 stock 구조체와 해당 구조체를 다루는 stock_load, stock_save, insert_tree, print_tree, search_tree, 그리고 process_request 함수를 선언하였다.

프로그램 실행 시 main 함수에서는 stock_load 함수를 통해 stock.txt에서 주식의 정보를 불러와 배열로 구현한 binary tree를 생성하고 초기화한다. 배열로 이를 구현한 이유는 삽입, 삭제보다 search가 많은 해당 프로그램 구조 상, 포인터 참조에 비해 index 연산이 빠르게 수행되고, 상대적으로 entry 개수 (stock의 개수) 가 적어서이다. 이후 process_request 내에서는 입력받은 요청에 따라 search_tree를 이용하여 해당하는 주식을 찾거나 print_tree를 이용하여 모든 주식의 내용을 출력하여 client에게 전송하도록 하였다.

Event-driven approach에서는 Select() 함수로 입력이 들어온 file descriptor를 선택 후, process_request() 함수에서 show, buy, sell, exit 명령에 대한 처리를 하였다. 별도의 client_pool 구조체를 선언하고 해당 구조체에서 client들의 file descriptor들을 관리한다. Client가 exit 명령어를 입력해 종료하거나 연결이 끊어질 경우 해당 client의 file descriptor를 구조체에서도 제거하도록 하였다.

Thread-based approach에서는 main thread가 새로 접속하는 client들에 대해 새로운 worker thread를 생성하고, 각각의 worker thread는 반복문을 통해 client의 접속이 종료될 때까지 필요한 요청을 처리하도록 하였다. 이때 주식 자료구조는 heap에 동적으로 메모리가 할당되어 있기에 모든 thread들이 접근할 수 있으며, 각 주식 구조체에 선언된 semaphore를 mutex처럼 활용하여 여러 개의 thread가 동시에 값에 접근하여 발생하는 문제를 방지하도록 하였다.

3. 구현 결과

사용자는 client 프로그램을 통하여 IP 주소, 포트 번호를 통하여 주식 서버에 접속할 수 있으며, show 명령을 통해 현재 주식들의 잔고를 확인할 수 있으며, buy와 sell 명령을 통해 주식을 사고 팔 수 있다. 또한 exit 명령을 통해 서버와의 접속을 종료할 수 있다. Server 프로그램은 여러 개의 client 요청들을 수행하며 binary tree로 구성된 주식 자료구조를 업데이트하며, SIGINT handler를 통하여 종료 시 현재 메모리에 있는 주식들의 정보를 stock.txt에 작성 후, 동적 메모리를 free한 후 종료된다.

4. 성능 평가 결과 (Task 3)

측정은 multiclient 프로그램의 코드에 gettimeofday 함수를 넣어, 자식 프로세스들이 생성되기 직전, 그리고 모든 자식 프로세스가 종료된 직후 사이의 시간 간격을 측정하며 진행되었다.

```
    host = argv[1];
    port = argv[2];
    num_client = atoi(argv[3]);

    // get start of measuring time
    gettimeofday(&start, NULL);

/*  fork for each client process    */
    while(runprocess < num_client){
        //wait(&state);
        pids[runprocess] = fork();
```

프로세스들이 fork 되기 전의 시간을 우선 측정하는 코드이다.

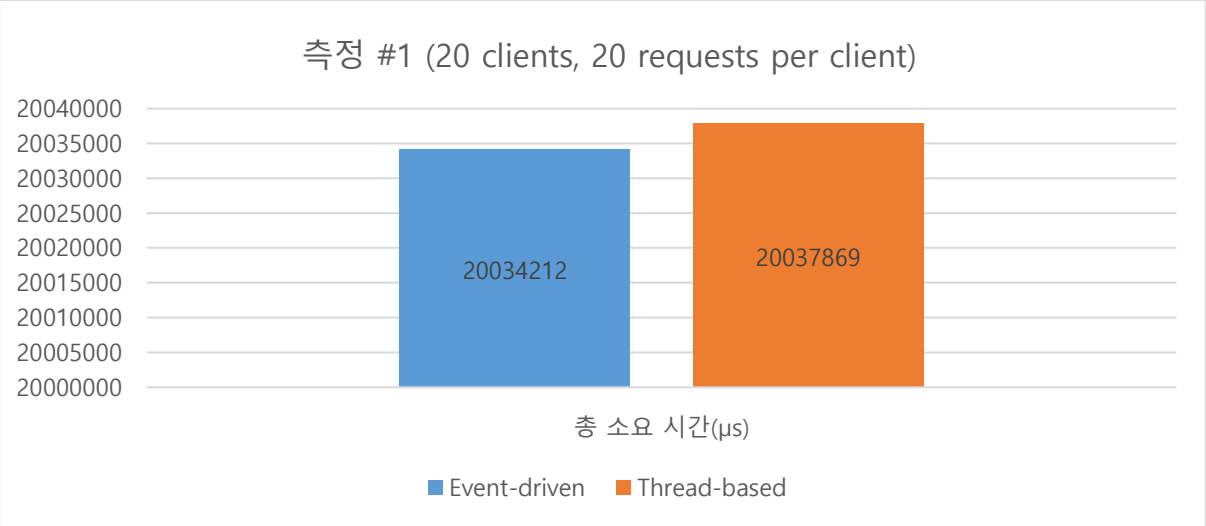
```
for(i=0;i<num_client;i++){
    waitpid(pids[i], &status, 0);
}

// get end of measuring time
gettimeofday(&end, NULL);

long seconds = end.tv_sec - start.tv_sec;
long microseconds = end.tv_usec - start.tv_usec;
long elapsed_time = seconds * 1000000 + microseconds;
float service_rate = num_client * ORDER_PER_CLIENT * 1000000 / (float) elapsed_time;
printf("Elapsed time : %ld microsecs\n", elapsed_time);
printf("Service rate : %.4f (requests per second)\n", service_rate);
```

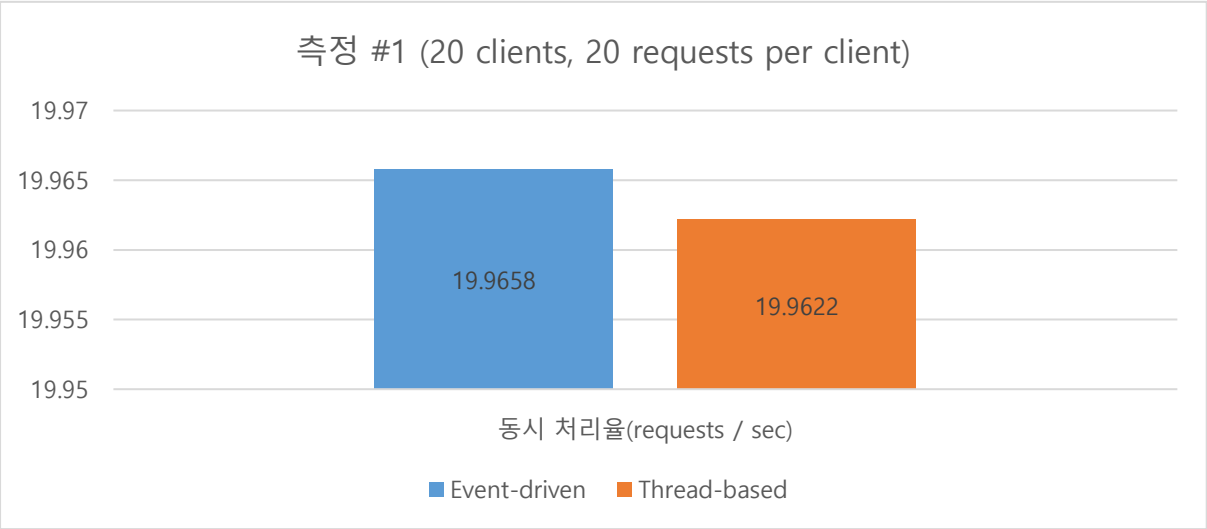
자식 프로세스들이 모두 종료된 후의 시간을 측정 후, 동시 처리율을 계산하여 출력하는 코드이다.

```
[sell] success
1 14 3000
5 89 3700
3 94 1200
2 36 20000
4 27 5000
[sell] success
[sell] success
[sell] success
1 14 3000
5 95 3700
3 102 1200
2 42 20000
4 27 5000
[sell] success
[buy] success
[sell] success
1 14 3000
5 86 3700
3 108 1200
2 42 20000
4 27 5000
Elapsed time : 20034212 microsecs
Service rate : 19.9658 (requests per second)
```

위와 같이 모든 프로세스 종료 후, 총 처리 시간과 앞서 정의한 동시 처리율을 계산하여 출력하도록 하였다.

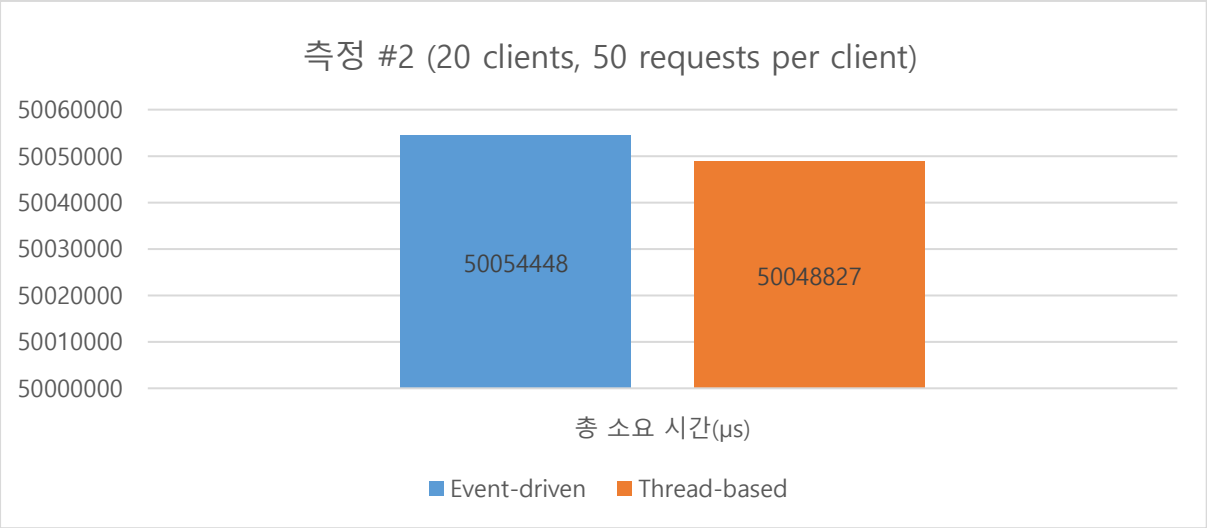
기준이 되는 측정으로는 20개의 client가 각각 20개의 요청을 보내었을 때의 처리 시간과 동시 처리율을 측정하였다.

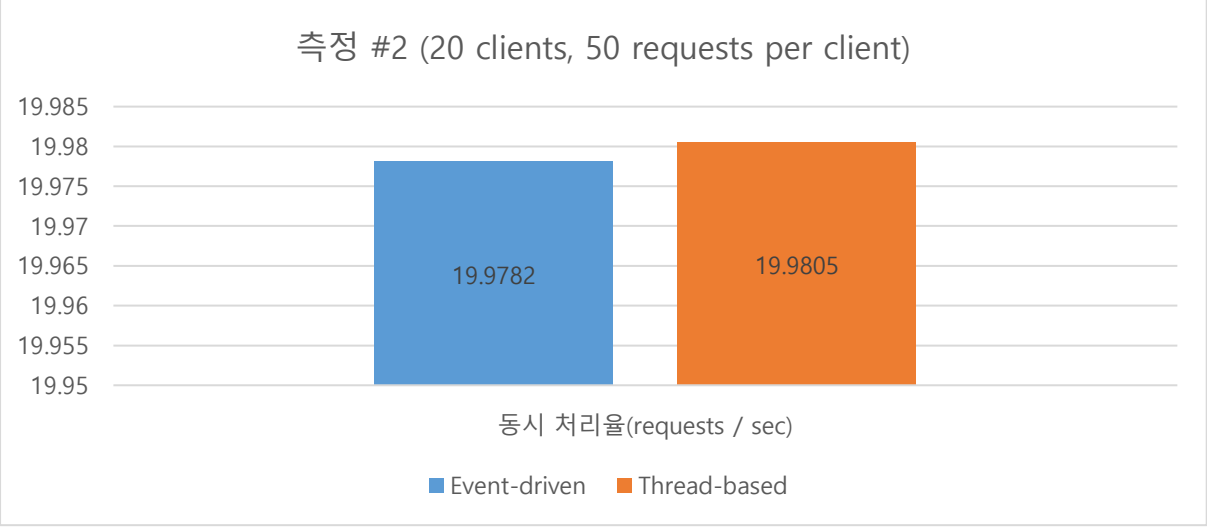




해당 측정에서는 event-driven 방식이 더 높은 동시 처리율을 보였다. 처리하는 request의 양이 적어 상대적으로 thread의 overhead보다 하나의 thread에서 처리하는 방식이 더 높은 효율을 보이는 모습을 볼 수 있었다.

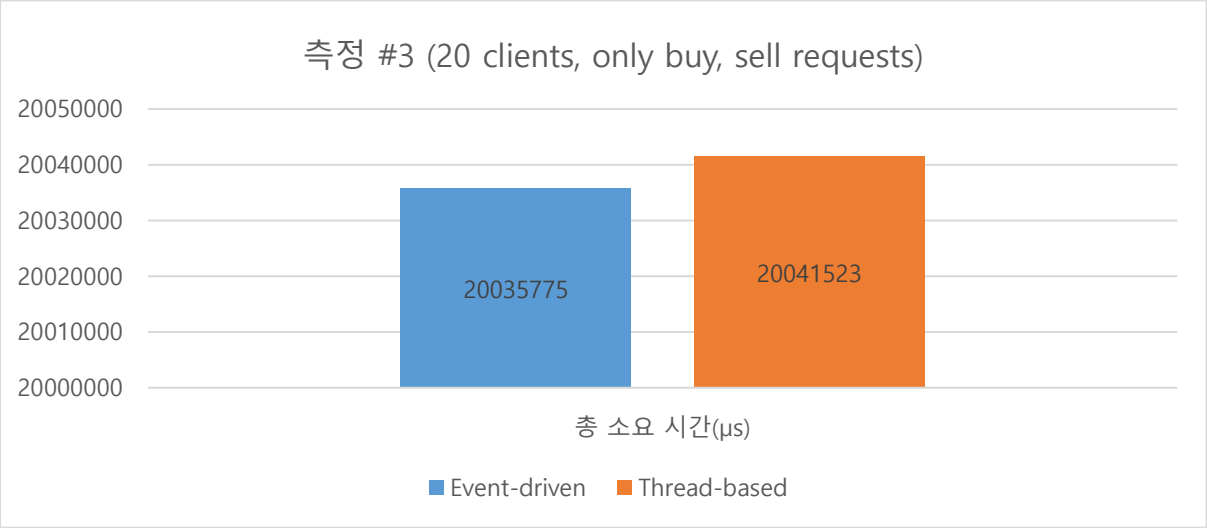
두 번째 측정에서는 각 client 당 요청 수를 50개로 늘렸다.

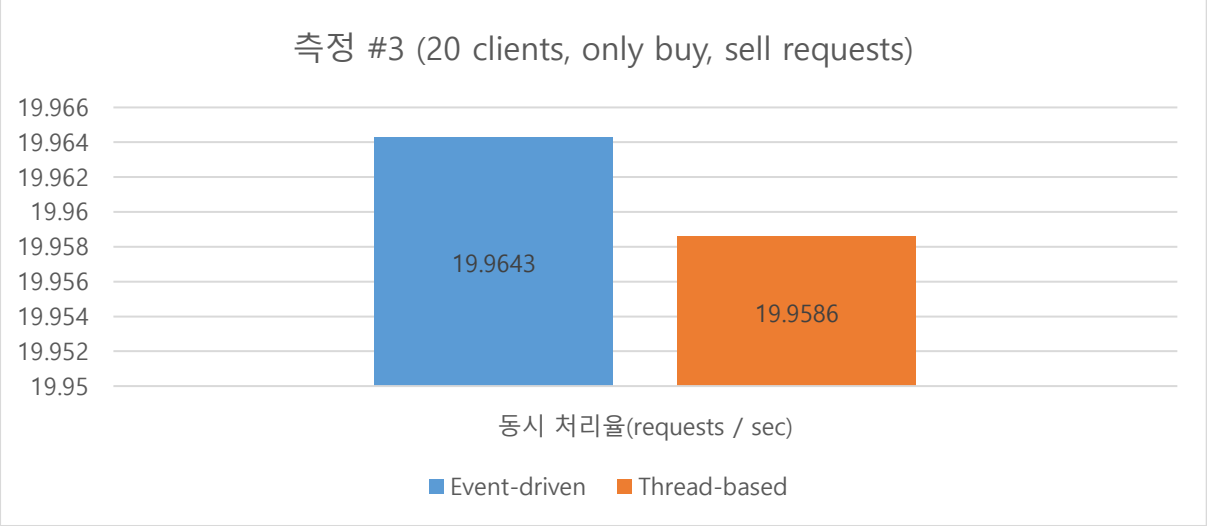




처리하는 request의 수가 많아지자, thread 처리의 overhead보다 처리량이 많아져 multithreading 방식의 동시 처리 효율성 덕분에 동시 처리율이 증가하는 것을 볼 수 있었다.

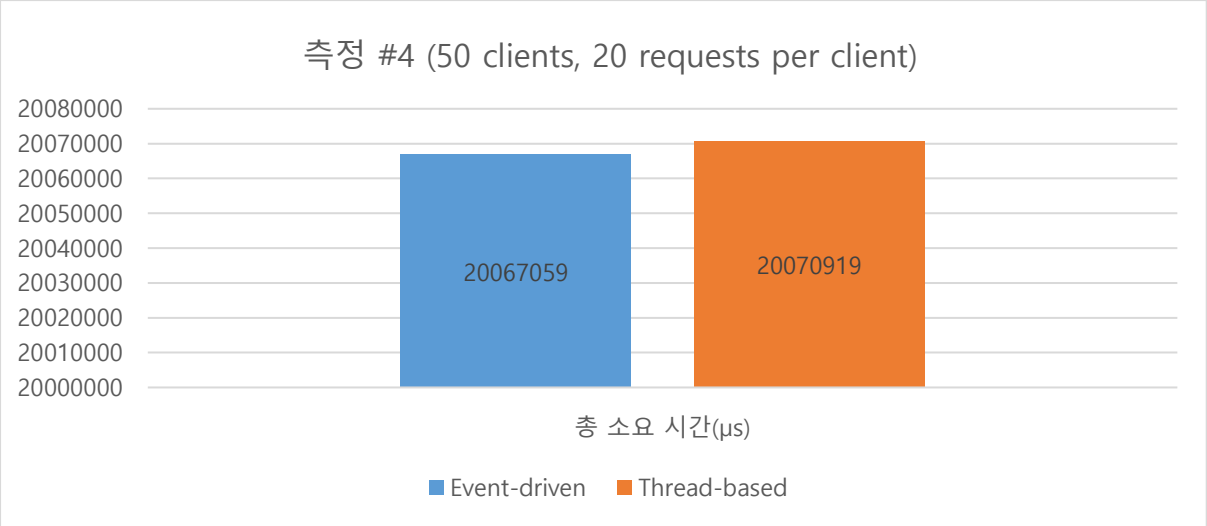
세 번째 측정에서는 request를 오직 buy, sell로만 한정하였다.

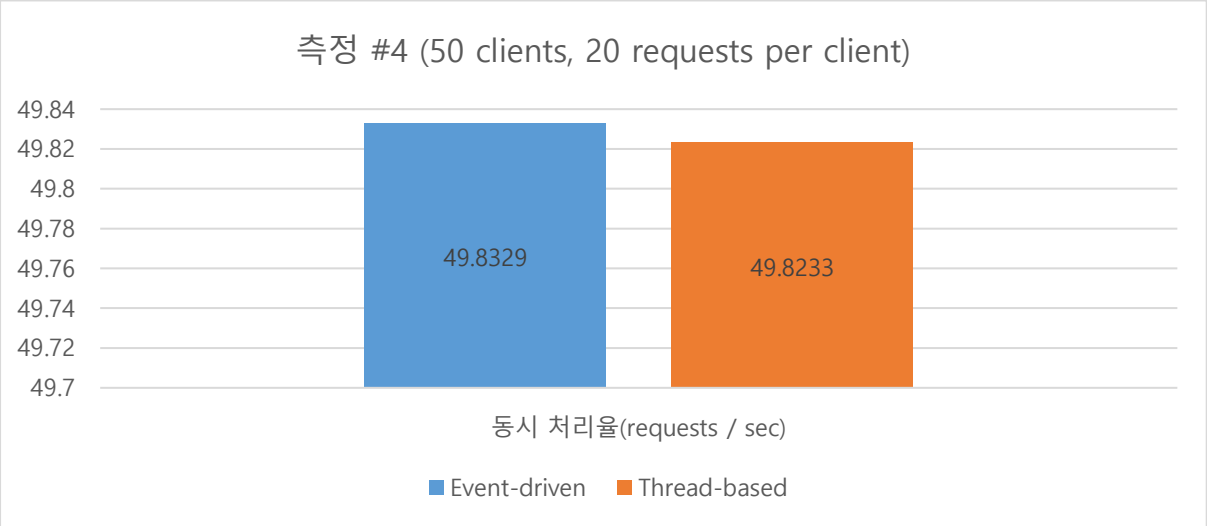




이 측정을 통하여 thread 방식에서의 mutex 활용 시 주식 구조체를 수정하기 위해 접근 시 P, V 함수를 통한 blocking의 overhead가 유의미한 수준으로 작용하여 동시 처리율이 event-based approach에 비하여 하락한 것을 확인할 수 있었다.

마지막 측정에서는 client 개수를 50개로 늘려 측정을 진행하였다.





예상과 다르게 동시 처리율이 높게 나오는 것을 확인할 수 있었는데, 이 결과를 정리하자면, 현재의 동시 처리율 측정 방식에서 여러 개의 자식 프로세스를 생성하고, 이들을 처리하는데 걸리는 시간이 대다수를 차지하기에 상대적으로 소요 시간에 차이가 크게 나지 않아 극단적으로 동시 처리율이 높게 나오는 것으로 추정이 간다.

따라서 각 client에 대하여 자식 프로세스 생성 후 종료까지의 시간을 측정하여 평균을 내는 것이 모든 프로세스에 대하여 포괄적으로 측정하는 것보다 client 수에 대한 서버의 처리율 측정에 더 적합한 metric이라는 점을 확인할 수 있었다.