Multicore Programming Project 2

담당 교수 : 박성용 교수님

이름 : 김승원

학번 : 20182186

1. 개발 목표

여러 Client들의 동시 접속 및 요청들을 처리하기 위한 Concurrent stock server를 구축하는 것이 이번 프로젝트 목표이다. 주식 서버는 Client들의 show, buy, sell 요청에 따라 해당하는 command에 맞는 작업을 수행하고 Client에게 결과를 응답한다.

데이터의 관리는 Binary tree 자료구조를 사용한다. 주식은 stock.txt 파일에 테이블 형태로 관리한다. 주식 단가 변동은 없고, 잔여 주식만 변동되도록 한다. 또한 남은 주식보다 많은 주식을 사려고 시도할 시 잔여 주식이 부족하다는 메시지만 출력하고 요청을 처리하지 않도록 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1) Task 1: Event-driven Approach

Event-driven Approach 방식을 사용해서 구현하면, 서버는 비동기적으로 요청을 처리하는 것이 가능하다. 즉, 서버는 한 client의 요청을 처리하면서 다른 client의 요청을 기다리지 않아도 되고, 시스템의 처리 용량을 크게 향상시킨다. 또한 하나의 thread에서 여러 Client의 요청을 동시에 처리할 수 있어서 thread 관리에 대한 overhead를 줄일 수 있고, 동시 처리 능력을 높일 수 있다는 장점이 있다. 따라서 Event-driven server는 client의 요청을 신속하게 처리하고, client들 또한 응답을 빠르게 받을 수 있다.

2) Task 2: Thread-based Approach

Thread-based Approach를 사용하면, Client 연결마다 별도의 thread를 할당하여 여러 client들의 요청을 동시에 처리할 수 있다. 그러나 client 수가 많아지면 thread를 생성하고 관리하는 overhead가 커질 수 있고, 더 많은 리소스를 사용하게 된다. 그리고 여러 thread가 공유 리소스에 접근할 때, 동기화 문제가 발생할 수 있다. 이로 인해 데이터 integrity가 손상될 수 있고, deadlock과 같은 문제가 발생할 수 있다. 따라서 Mutex와 semaphore를 활용하여 구현하여 문제를 막도록 한다.

3) Task 3: Performance Evaluation

Event-driven approach와 thread-based approach를 활용해서 client 개수에 따라 실행

시간을 비교하고, 동시 처리율이 어떤 지 비교해본다. 또한 thread-based approach에서 thread 개수에 따라 실행시간이 어떻게 달라지는지 확인해본다.

B. 개발 내용

- 아래 항목의 내용만 서술

- (기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)

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

1) Multi-client 요청에 따른 I/O Multiplexing 설명

서버는 수많은 client의 요청을 처리한다. 단일 thread를 통해서 모든 client의 요청을 처리하기 위해서 사용되는 event-driven approach에서 select 함수를 사용해서 처리한다. Select() 함수는 여러 socket의 상태를 확인하고, 한 개 이상의 I/O 연산이 완료될 때까지 프로그램을 블록 한다. 이를 I/O multiplexing이라고 한다.

I/O multiplexing은 하나의 process나 thread가 여러 개의 I/O 요청을 동시에 처리한다. 서버는 각 client의 상태를 계속 확인하고, client로부터 데이터가 도착하면 그것을 읽고 적절하게 응답을 제공한다. 이를 위해서는 서버는 모든 client socket을 감시하며, 데이터가 준비된 socket을 찾아서 읽는다.

2) epoll과의 차이점 서술

select()와 epoll() 모두 I/O multiplexing을 위한 메커니즘이다. Select는 file descriptor 배열을 하나씩 탐색하는 loop를 돌도록 되어 있다. select함수를 호출할 때마다 descriptor를 저장하는 전체의 정보를 확인 하기 위하여 매번 순회하기 때문에 file descriptor의 개수가 증가하면 효율성이 크게 저하된다. 그러나 epoll()은 file descriptor를 효과적으로 관리하기 위하여 커널에서 특별히 최적화 시킨 데이터 구조를 사용한다. 따라서 epoll은 대량의 file descriptor를 더 효율적으로 처리하고, 확장성이 뛰어나다는 장점이 있다.

즉, epoll()이 대규모의 network application에 훨씬 더 적합하고, 더 높은 성능과 확장성을 제공한다. 그러나 select는 API가 간단하고, 호환성이 광범위하여 epoll의 장점이 명확함에도 여전히 많이 사용하는 방식이다.

- Task2 (Thread-based Approach with pthread)

1) Master Thread의 Connection 관리

Master thread를 통해서 client의 요청을 받아들이고 관리한다. 새로운 client가 연결을

요청하면 master thread는 해당 연결을 받아들이면서 새로운 socket을 생성한다. 이후 이 socket를 처리할 때 worker thread를 선택하고 할당한다. 이렇게 하여 master thread는 다수의 동시 연결을 관리하고, 각 연결을 적절한 worker thread에 분배한다.

2) Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread들은 일반적으로 미리 생성하여 pool에 보관한다. Master thread는 client 로부터 요청을 worker thread에게 할당하고 요청을 처리한다. 각 worker thread는 할당된 작업을 처리하고 난 뒤에 다시 thread pool로 돌아가서 다음 할당을 기다린다. 이러한 방식으로 thread생성과 제거를 최소화하고, 또한 각 요청을 병렬적으로 처리하여 서버의 처리 성능을 향상시킨다.

즉, thread pool의 크기는 서버의 성능과 목적에 따라 조정이 필요할 것이다. 만약 너무 많은 worker thread가 있다면, context switching 비용이 커질 수 있어서 적절한 수의 worker thread를 유지하는 것이 중요하다.

- Task3 (Performance Evaluation)

1) 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

성능 테스트를 위해서 여러가지 상황과 요구사항을 고려하고, 여러 번의 테스트를 통해서 결과를 측정한다. 본 프로젝트에서는 metric을 다음과 같이 결정했다.

→ Elapsed time

Client들의 요청을 처리하는데 걸리는 전체 시간을 측정한다. 이러한 시간을 측정하여 서버의 처리 능력과 성능을 측정하는 데 중요한 척도가 될 수 있다. 또한 실제로 주식서버를 구현하고 서비스 할 때 client들의 경험에 직접적으로 연관되어 있고, 성능에 따라 만족도가 달라질 수 있어 중요한 지표가 될 수 있다.

→ Concurrent throughput(동시 처리율)

이는 서버가 동시에 처리할 수 있는 요청의 수이다. 이 지표는 서버의 멀티태스킹 능력을 평가할 수 있고, 서버가 client의 요청을 동시에 처리할 수 있다면 이는 더 높은 처리량을 가짐을 의미한다. 이는 서버의 확장을 얼마나 할 수 있을지 평가하는 데 중요한 지표이고, 더 많은 사용자 요청을 처리할 수 있는 서버가 더 성능이 우수하다고 볼 수 있다.

2) Configuration 변화에 따른 예상 결과 서술

본 프로젝트에서는 multiclient.c 파일에서 gettimeofday 함수를 활용하여 elapsed time을 측정했다. Client 개수가 증가함에 따라서 시스템이 어떻게 반응하는지 측정하고, client 수를 점진적으로 증가시키면서, 시스템의 concurrent throughput이 어떻게 변화하는지 살펴본다. 이를 통해서 평소 시스템의 상태나 과부화 상황에 대해서 어떠한 성능을 보이는지 체크할 수 있다.

또한 워크로드에 따른 분석으로 서로 다른 client 요청에 다라서 성능이 어떻게 변화하는지 측정한다. 가령, 모든 client가 buy나 sell을 요청한 경우, show만 요청하는 경우, 그리고 모든 요청이 섞이는 경우에 대한 시나리오를 구성하여 각 시나리오에 대한 시스템의 반응을 측정하고 분석한다.

Event-driven approach의 경우 client들이 복잡한 연산만 요구하면, 성능저하가 예측된다. 모든 요청을 한 thread가 처리하기 때문에, 한 요청의 처리 시간이 길어진다는 건 다른 요청의 대기시간이 증가하기 때문이다. 반면에 show와 같이 단순한 연산만 한다면, event-driven 모델은 성능이 thread-based approach에 비해서 상대적으로 좋을 수도 있다.

Thread-based approach의 경우 buy, sell과 같은 client들의 요청이 많더라도 thread 기반의 approach의 경우 각 thread가 독립적으로 작동하여 성능 저하가 덜할 것으로 예상된다. 그러나 단순한 연산만을 요구하는 show와 같은 요청이 많을 경우에는 thread 생성과 관리에 따른 overhead로 인하여 성능 저하가 발생할 수 있다고 예상된다.

Concurrency는 프로그램이 여러 작업을 독립적으로 처리하거나 중복해서 처리할 수 있다는 것이다. 그러나 모든 작업이 동시에 실제로 실행됨을 뜻하는 것은 아니다. 단일 프로세스 시스템에서도 구현할 수 있고, 여러 작업이 상호 교차 실행된다는 것을 의미한다.

Event-driven approach는 기본적으로 concurrency를 지원한다. 이러한 방식은 여러 client의 요청을 감지하고 해당 요청을 처리한다. 한 번에 하나의 이벤트만 처리하지만, 이 처리는 비동기적이고 다른 event간에 중첩될 수 있다. 그러나 event-driven approach가 parallelism을 의미하지는 않는다. 여러 작업이 동시에 실행되는 것이 아니라는 것이다. Thread-based approach와 같이 여러 thread를 사용할 수 있을 때만 동시에 여러 작업이 실행될 수 있다. 따라서 서버가 클라이언트의 모든 요청을 처리하는 데 걸리는 전체 시간은 thread-based 방식이 더 짧을 것으로 예측된다. Event-driven approach 실제로는 동시에 처리되는 것이 아니라 차례로 요청을 처리하기 때문에 실제로 동시에 실행되는 thread-based가 더 빠를 것으로 예측된다. 그러나 worker thread수에 따라서 thread-

based는 실행시간이 변동될 것이다. client수가 만약 worker thread 수보다 많다면, 성능 저하가 예상된다. 또한 너무 많은 worker thread는 context switching에 의해서 overhead 가 증가됨에 다라 성능이 저하될 수 있을 것으로 예상된다.

C. 개발 방법

- B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)

우선 event-driven approach이다.

구조체로 주식 아이템을 나타내는 구조체와 file descriptor의 pool을 나타내는 pool 구조체를 구현한다. 동시에 서비스 할 수 있는 client 수를 제한하고, 각 client의 상태를 추적하는데 사용한다.

Savestock 함수는 서버가 종료될 때 호출하여 주식상태를 파일에 저장하는 함수로 구현한다. 서버가 다시 시작될 때 이전상태를 복원하기 위함이다.

Request_show 함수는 모든 주식 항목의 현재상태를 문자열 버퍼에 기록하는 함수이다. 문자열 버퍼의 시작위치에 대하여 포인터와 새롭게 추가된 문자열 길이를 저장할 수 있는 변수를 초기화 하고, 버퍼에 show₩n을 기록한 뒤 포인터를 새롭게 추가된 문자열의 끝으로 이동시키는 방식으로 구현한다. 또한 request_buy와 request_sell은 특정 주식의 구매와 판매를 처리하기 위함 함수이다. ID를 사용하여 주식을 찾고, 주식의 구매가 가능하면 1을 반환하고, 만약 주식 수량이 현재 재고보다 많이 요청하면 0을 반환하도록 구현한다. Sell은 주식 ID를 사용해서 판매할 주식을 찾고, 주식이 존재하지 않으면 아무것도 하지 않고, 존재하면 주식의 재고를 판매수량만큼 늘려준다.

Load_tree는 서버 시작 시에 주식 상태를 파일에서 load한다. 먼저 동적 메모리 할당을 사용해 새로운 주식 항목을 위해서 메모리 공간을 만들고, stock.txt 파일을 읽기 모드("r")로 열고, 파일의 각 줄을 읽는다. 각 줄은 하나의 주식 항목을 나타내고, ID와 잔여주식 그리고 주식가격 정보를 순서대로 포함한다. Ex) 1 3 5000. 각 항목에 대해서 새로운 주식 노드를 만들고, 이를 주식 트리에 추가해준 다음 마지막으로 파일을 닫는다.

Add_client는 연결된 client를 pool에 추가하는 역할을 하는 함수로 구현한다. pool에서 사용가능한 slot을 찾고, 사용가능한 slot에 connfd를 추가하고, descriptor 세트에도 추가한다. 만약 추가한 discriptor가 현재의 최대 discriptor보다 큰 경우에는 최대 discriptor를 업데이트한다. 만약 추가한 slot의 index가 현재의 가장 높은 index보다 큰 경우, 가장 높은 index를 업데이트 한다. 만약 사용 가능한 slot이 없는 경우에는 add_client error : Too many clients를 출력한다.

Init_pool은 pool을 초기화하는 함수다. 모든 client discriptor를 -1로 초기화 하여 사용가능하게 만들도록 구현한다. Listen fd를 최대 fd로 설정하고, descriptor set를 초기화하

고 listenfd를 set에 추가한다.

Check_connection함수는 pool에 연결된 client가 있는지 확인한다. Pool에 있는 모든 client discriptor를 검사하고, 하나라도 연결되어 있으면 1을 반환하도록 구현한다. 만약 없는 경우 0을 반환하도록 구현한다.

Close_connection은 client와의 연결을 닫고 pool을 업데이트 하도록 구현한다. 연결을 닫고 descriptor set에서 connfd를 제거한다. Client discriptor를 -1로 설정한다. 만약 pool에 연결된 client가 없는 경우 현재의 주식상태를 저장한다.

Process_client_request함수는 주식의 구매나 판매 요청을 처리한다, buy요청은 요청을 처리하고 구매가 성공하면 성공 메시지를 반환하고, 만약 주식이 충분하지 않은 경우 Not enough left stock을 출력하여 에러메세지를 반환한다. Sell은 요청을 처리하고 실패하는 경우 없이 성공 메시지를 반환하도록 구현한다.

Check_clients는 pool에 있는 모든 client를 체크하고, 각 client로부터의 입력을 처리하도록 구현한다. pool에 있는 client를 순회하면서, client가 입력을 준비하고 있는지 확인한다. Client가 입력하면 해당 입력을 읽고, show면 현 주식상태를 보여주도록 하고, exit인 경우 해당 client와 연결을 종료한다. show, exit가 아닌 경우 buy와 sell요청으로 간주하고 처리한다. 요청을 처리한 후 client에게 적절한 응답을 보낸다. 만약 client가 더 이상 데이터를 제공하지 않는 경우, 해당 client와 연결을 종료한다.

마지막으로 main함수에서 client의 연결을 수락하고, 요청을 처리하는데 필요한 초기화 작업을 수행하도록 구현한다. 먼저 프로그램이 올바른 수의 인자를 받았는지 확인하고, 만약 못받은 경우 사용자에게 올바른 인자를 입력하도록 안내한다. 이후 stock.txt에서 주식 상태를 load하고, 주어진 포트에서 listen socket을 열고, client pool을 초기화한다. 그 다음 루프를 통해서 client의 연결을 계속 수락하고, 각 client의 입력을 처리한다. 만약 이 과정에서 client가 연결을 요청한다면, client를 pool에 추가해주는 작업을 하도록 구현한다. 모두 완료되면 이후 프로그램이 종료된다.

다음은 thread-based approach이다.

THREADNUM과 SBUF_SIZE, MAX를 선언한다. 차례대로 프로그램에서 사용할 thread 수를 나타내고, 공유 버퍼의 크기, 주식 아이템의 최대 개수를 나타낸다.

Item 구조체는 동일하고, sbuf_t 구조체를 추가로 구현한다. 이 구조체는 공유 버퍼를 나타내고, thread간 데이터를 안전하게 전달할 수 있도록 사용된다.

Subf_init 함수는 공유 버퍼를 초기화하는 함수로 구현한다. N의 크기를 가진 공유 버퍼를 생성하고, 버퍼의 front와 rear을 0으로 초기화해서 버퍼가 비어있음을 나타내도록 한다. 그리고 semaphore를 사용해서 버퍼에 대한 동시접근을 제어하고, mutex semaphore, slot semaphore, item semaphore를 초기화한다.

Subf_deinit 함수는 공유 버퍼를 삭제하는 함수다. Free()를 사용해서 동적으로 할당한 버퍼를 해제한다.

Subf_insert는 공유 버퍼에 아이템을 삽입하는 함수로 구현한다. Slot semaphore를 감소시켜서 사용 가능한 칸이 있는지 확인한 다음 mutex semaphore를 감소시켜서 버퍼에 대한 접근을 잠군다. 이후 삽입할 아이템을 rear 위치에 넣고 rear를 증가시키고, mutex semaphore를 증가시켜 잠금을 해제하도록 한다. 이후 item semaphore를 증가시켜서 아이템이 새롭게 추가된 것을 확인한다.

Sbuf_remove는 아이템을 버퍼에서 제거하는 함수로 구현한다. 먼저 item semaphore를 통하여 사용할 수 있는 아이템이 있는지 확인하고, 아이템을 제거하기 위해 버퍼에 대한 접근을 동기화한 다음 아이템을 제거한 후 mutex를 해제하고 slot semaphore를 증가시켜서 빈 슬롯이 생겼음을 반환한다.

Savestock, request_show, request_buy, request_sell은 기능적으로 event-driven approach와 동일하다. 그러나 동시성을 고려해서 buy와 sell은 mutex lock을 걸어서 수행하도록 구현한다. Load_tree 또한 기능적으로 동일하나 Sem_init 함수를 호출하는 부분을 추가하여 구현한다.

Thread 함수는 thread가 처리할 작업을 정의할 수 있도록 구현한다. Thread는 연결이 수립된 socket에서 데이터를 읽어오고 응답한 후 소켓을 닫는다. 모든 연결이 완료될 경우 connfdcnt가 0이 되고, savestock함수를 이후에 호출해서 주식 정보를 저장하고 thread를 종료한다.

Init_echo_cnt는 동시성 제어를 위한 semaphore를 초기화하고, bytecnt를 0으로 설정할 수 있도록 한다.

Handle_show_cmd, Handle_buy_cmd, Handle_sell_cmd, Handle_invalid_cmd는 각각 resquset_show, request_buy, request_sell를 처리하는 함수들이다. Show는 서버에 현재 상태를 표시하도록 하고, buy와 sell는 구매를 성공할 시, ret_buf에 성공메세지를 저장하고, 만약 buy의 경우 그렇지 않으면 재고 부족 메시지를 저장한다. Handle_invalid_cmd는 Invalid 명령어를 받은 경우 호출한다. Ret_buf에 Invalid command 메시지를 저장한다.

Echo_cnt는 client로부터 받은 명령을 처리하고 응답한다. 각 4가지 명령어를 처리하도록 하고, 명령이 유효하지 않은 경우 잘못된 명령 메시지를 반환하도록 한다. 각 명령은 handle 함수들을 사용해서 처리하고, 처리된 결과를 client에게 응답한다. Pthread_once 함수를 통해서 semaphore를 초기화 하는 작업도 한 번 추가한다.

main함수는 sbuf_init으로 공유된 버퍼를 초기화하는 작업과 Threadnum의 개수에 따라 thread를 생성하고, Sem_init 함수를 통해서 fd에 대한 동시 접근을 제어하는 semaphore를 초기화하는 작업을 추가한다. While loop를 통하여 client의 연결 요청을 계속 수신하도록 하고, 연결되면 sbuf_insert를 통해서 connfd를 공유된 버퍼에 삽입한다. 이렇게 하면 thread가 공유 버퍼에서 연결을 가져와서 client와 통신할 수 있게 된다. Connfdcnt를 통해서 연결 수를 추적하고 새 연결이 추가될 때마다 증가하도록 구현한다.

3. 구현 결과

- 2번의 구현 결과를 간략하게 작성

- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

Task1은 client가 요청할 때 select함수로 요청받은 connfd를 check_clients 함수에서 Rio_readlineb를 통해 읽어오도록 하고, 요청한 명령에 따라 명령을 수행한 후 출력결과를 리턴 버퍼에 담아서 Rio_writen함수를 통해 client에게 응답한다. Show를 입력받은 경우 현재 주식 정보를 출력하여 주고, sell과 buy는 적절한 명령 양식이 입력된 경우 성공했다고 응답한다. 만약 buy의 경우 주식 수보다 많은 수를 구매하려고 시도할 시 Not enough left stock 메시지를 client에 응답한다. 구현결과 client수가 증가하면, client요청을 하나씩 처리하는 것을 확인했다.

Task2는 worker thread를 생성하고, 이후에 accept로 생성된 connfd를 sbuf에 넣은 다음 sbuf에서 thread들이 각각 connfd를 꺼내어서 연결하고 각 client와 통신을 시작하고 request를 요청받는다. Thread 함수에서 각각 thread들이 명령어를 받고 결과를 client에 응답한다. 구현결과 한 번에 여러 client 요청을 수행하는 결과를 확인했다. 단 sell buy의 경우 semaphore방식을 사용해서 동시에 진행될 수 없는 경우가 있어 delay가 발생하는 경우를 확인했다.

4. 성능 평가 결과 (Task 3)

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

성능 평가를 위해서 gettimeofday 함수를 사용해서 multiclient 실행부터 생성된 child process들이 모두 exit한 시점(모든 명령어에 대한 응답을 받고 종료된 시점)까지 시간을 측정했다. 동시 처리율 값은 초당 client 요청 처리 개수로 사용했다. 시간당 client 요청 처리 개수를 구하기 위해서 ORDER_PER_CLIENT값을 10으로 고정시키고, Client 수만 변화를 주었다. 그리고 thread-based의 경우 thread 값도 변화를 주었다. 이후 Elapsed time으로 나누어서 동시처리율 값을 구했다. 이후 단위를 10^7을 곱해주어서 비교를 편하게 확인한다. Thread-based approach(n)에서 n은 thread 수를 의미한다.

<Table 1. Client 수에 따른 Elapsed time 출력 결과 캡쳐>

	Event-driven approach	Thread-based approach(10)
4	elpased time = 10025227 microseconds	elpased time = 10033224 microseconds
10	elpased time = 10027179 microseconds	elpased time = 10033943 microseconds
20	elpased time = 10019173 microseconds	elpased time = 20028302 microseconds
30	elpased time = 10043291 microseconds	elpased time = 30026072 microseconds
40	elpased time = 10034664 microseconds	elpased time = 40029195 microseconds
100	elpased time = 10035789 microseconds	elpased time = 100113686 microseconds

각 행은 client 수이고, ORDER_PER_CLIENT = 10, Thread 개수는 10이다. Buy, sell, show 모든 명령어가 요청 가능한 상태이다.

위의 표를 토대로 Table2과 Table3를 작성했다.

<Table 2. Client 수에 따른 Elapsed time 결과와 동시처리율(Event-driven approach)>

	4	10	20	30	40	100
Elapsed time(μs)	10025227	10027179	10019173	10043291	10034664	10035789
동시처리율	39.8993459	99.72895	199.6173	298.7069	398.6182	996.4339

각 열은 client 수를 의미하고, ORDER_PER_CLIENT = 10, Elapsed time 단위는 microsecond이다.

<Table 3. Client 수에 따른 Elapsed time 결과와 동시처리율(Thread-based approach(10))>

	4	10	20	30	40	100
Elapsed time(μs)	10033224	10033943	20028302	30026072	40029195	100113686
동시처리율	39.8675441	99.66172	99.85869	99.91317	99.92707	99.88644

각 열은 client 수를 의미하고, ORDER_PER_CLIENT = 10, Elapsed time 단위는 microsecond이다. 또한 모든 결과의 worker Thread 개수는 10개로 고정시켜서 진행했다.

<Table 4. Thread 수에 변화를 준 후 Client 수에 따른 Elapsed time 출력 결과 캡쳐>

	Thread-based approach(5)
4	elpased time = 10029159 microseconds
10	elpased time = 20034497 microseconds
20	elpased time = 40043457 microseconds
30	elpased time = 60031292 microseconds
40	elpased time = 80113083 microseconds
100	elpased time = 200105138 microseconds

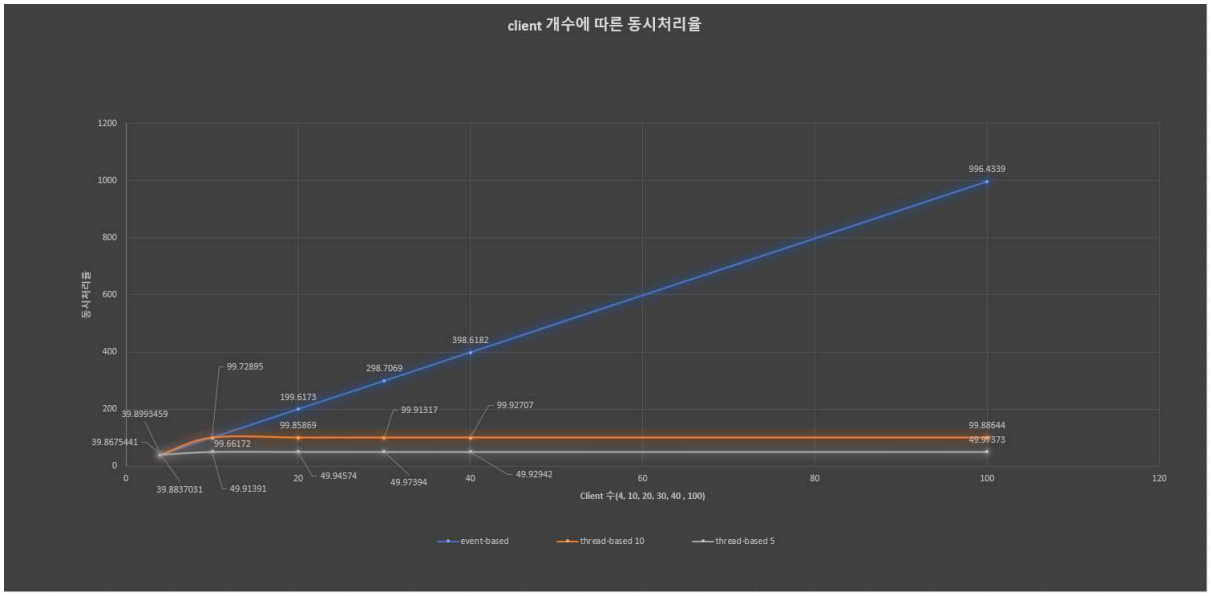
각 행은 client 수이고, ORDER_PER_CLIENT = 10, Thread 개수는 5로 줄였다. Buy, sell, show 모든 명령어가 요청 가능한 상태이다.

위의 표를 토대로 Table5과 Table6를 작성했다.

<Table 5. Client 수에 따른 Elapsed time 결과와 동시처리율2(Thread-based approach(5))>

	4	10	20	30	40	100
Elapsed time(μs)	10029159	20034497	40043457	60031292	80113083	200105138
동시처리율	39.8837031	49.91391	49.94574	49.97394	49.92942	49.97373

각 열은 client 수를 의미하고, ORDER_PER_CLIENT = 10, Elapsed time 단위는 microsecond이다. 또한 모든 결과의 worker Thread 개수는 5개로 바꿔서 진행했다.



<Figure 1. Client 개수에 따른 동시 처리율 그래프>

원래 예상은 event-driven approach가 각 요청을 한 thread가 처리하기 때문에, 하나의 요청 처리 시간이 길어질수록 다른 요청들이 대기하는 시간이 길어질 것으로 예측했다. 이에 반해, thread-based approach는 독립적인 여러 thread가 동시에 처리할 수 있으므로, 이 방식이 성능상 우위에 있을 것으로 생각했다.

하지만 실험 결과는 다르게 나타났다. event-driven approach에서는 client 수가 증가함에 따라 요청 처리율도 비례해서 증가했다. 반면 thread-based approach에서는 thread 수가 제한되어 있어, 클라이언트 수가 증가할 때마다 처리 시간도 같은 비율로 증가했습니다. 더욱이, thread 수를 5개로 제한하면 처리 시간은 더욱 증가하고, 처리율은 반으로 줄어든 것을 확인했다. Event-driven approach에서 처리율이 높다는 것을 확인했다.

다만, Thread-based approach에서 thread 수가 클라이언트 수와 같거나 많을 경우에는 event-driven approach와 비슷한 처리율을 보였다. 그러나 thread 수가 클라이언트 수보다 적을 경우에는 thread의 수가 충분하지 않아 처리율이 증가하지 않고 일정한 것으로 확인됐다.

조금이라도 event-driven approach에서 조금 더 높은 처리율을 보였다. event-driven approach는 대기 시간을 최소화하고, 효율적인 리소스 사용을 하고, thread control overhead 문제가 없다고 학습했고 이러한 부분에서 처리율에 이점이 있다고 생각된다. 반면, thread-based approach는 thread간의 context switching overhead 문제와 동기화 문제로 인해서 성능 저하의 가능성이 있다고 생각된다. 세마포어를 통해 공유 변수에 대한 접근을 관리하다 보니 시간이 더 걸릴 것이다. 또한 정해준 Worker thread만큼 메모

리가 사용되고, 대부분의 경우 select보다 메모리 사용량이 많을 것으로 생각되고, 새로운 client에게 추가로 리소스가 필요하여 메모리 관점에서 봤을 때 성능 저하의 가능성이 있을 것이다. 즉, 이러한 이유로 Select 방식 서버가 처리율이 조금 더 높게 결과가 나타났다. 따라서 전체적으로 처리율이 높은 event-driven approach으로 구현한 서버가 더 나은 확장성을 제공할 것으로 생각된다.

위는 show와 buy sell 명령어를 모두 요청한 경우이고, 다음으로 워크로드에 따른 분석을 진행한다. show명령어만 요청하는 경우와 buy, sell 명령어만 요청하는 경우에서 elapsed time과 동시처리율을 비교했다.

<Table 6. Client 수에 따른 Elapsed time 출력 결과 캡쳐>

	Event-driven approach(show)	Event-driven approach(buy, sell)
4	elpased time = 10102700 microseconds	elpased time = 10022086 microseconds
10	elpased time = 10193219 microseconds	elpased time = 10047303 microseconds
20	elpased time = 10143173 microseconds	elpased time = 10033744 microseconds
30	elpased time = 10179296 microseconds	elpased time = 10033933 microseconds
40	elpased time = 10205964 microseconds	elpased time = 10031136 microseconds
100	elpased time = 10247031 microseconds	elpased time = 10034812 microseconds
	Thread-based approach(show)	Thread-based approach(buy, sell)
4	elpased time = 10033973 microseconds	elpased time = 10034424 microseconds
10	elpased time = 10033364 microseconds	elpased time = 10030717 microseconds
20	elpased time = 20029434 microseconds	elpased time = 20027950 microseconds
30	elpased time = 30033071 microseconds	elpased time = 30039111 microseconds
40	elpased time = 40076631 microseconds	elpased time = 40031765 microseconds
100	elpased time = 100147771 microseconds	elpased time = 100060107 microseconds

위의 출력결과를 토대로 table7-10 까지 작성했다.

<Table 7. show명령만 요청할 경우 Elapsed time 결과와 동시처리율(Event-driven approach)>

	4	10	20	30	40	100
Elapsed time(μs)	10102700	10193219	10143173	10179296	10205964	10247031
동시처리율	39.593376	98.10444	197.177	294.7159	391.9277	975.8924

<Table 8. Client가 buy와 sell명령만 요청할 경우 Elapsed time 결과와 동시처리율(Event-driven approach)>

	4	10	20	30	40	100
Elapsed time(μs)	10022086	10047303	10033744	10033933	10031136	10034812
동시처리율	39.9118507	99.5292	199.3274	298.9855	398.7584	996.5309

Order_PER_CLIENT =10, Elapsed time 단위는 micro second이다.

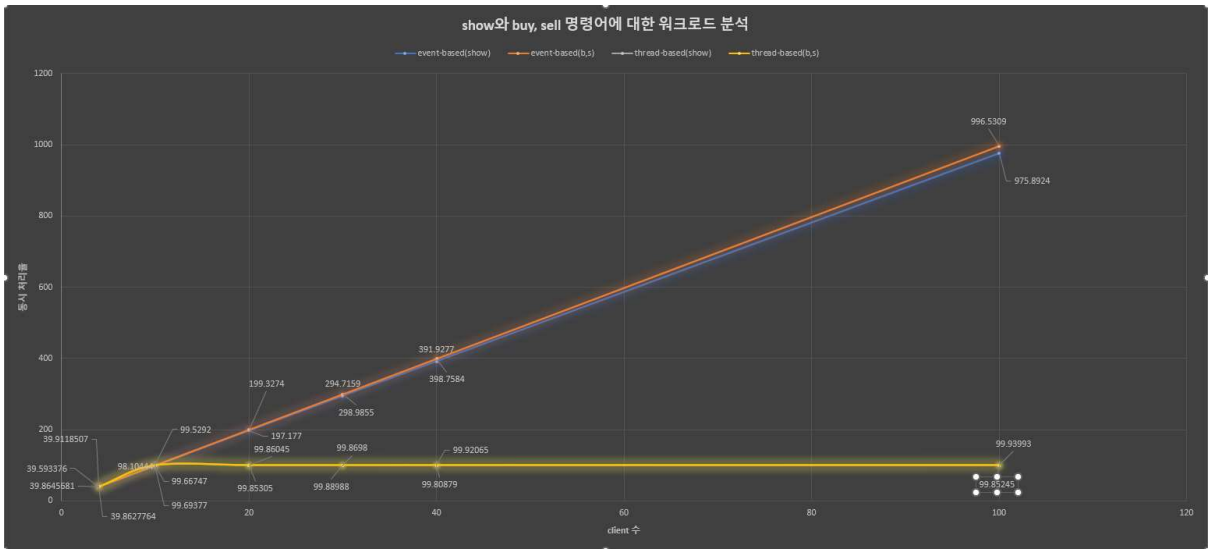
<Table 9. Client가 show명령만 요청할 경우 Elapsed time 결과와 동시처리율(Thread-based approach)>

	4	10	20	30	40	100
Elapsed time(μs)	10033973	10033364	20029434	30033071	40076631	100147771
동시처리율	39.8645681	99.66747	99.85305	99.88988	99.80879	99.85245

<Table 10. Client가 buy와 sell 명령만 요청할 경우 Elapsed time 결과와 동시처리율(Thread-based approach)>

	4	10	20	30	40	100
Elapsed time(μs)	10034424	10030717	20027950	30039111	40031765	100060107
동시처리율	39.8627764	99.69377	99.86045	99.8698	99.92065	99.93993

Order_PER_CLIENT =10, Elapsed time 단위는 micro second이다. 그리고 Thread 총 수는 10으로 고정하여 진행했다.



<Figure 2. Client 개수에 따른 동시 처리율 그래프>

Show와 buy, sell 명령어에 따라서 event-driven approach와 Thread-based approach에 특별히 변화를 주는 것은 없었다. 이전과 같이 비슷한 처리율을 보였다. 다만 워크로드 분석을 해보았을 때, show 명령어가 거의 모든 경우에서 응답시간이 길었고, 처리율이 낮았다. 이는 show 명령어의 경우 모든 주식 정보에 접근하고, 모두 출력을 해주는 부분에서 실행하는데 필요한 시간이 조금 더 길어졌을 것으로 생각된다. 각 명령어의 로직이 복잡한 경우에는 시스템 부하가 좀 더 심해질 수 있다는 것을 알 수 있다.

정리하면, event-driven approach의 thread의 개수가 적음에도 불구하고 많은 수의 client를 처리할 수 있고, 각 client에 대한 thread가 필요하지 않아 client 수에 대해서 더욱 확장성이 좋다는 장점이 있다. 또한 I/O 작업 동안 thread가 차단되지 않아 시스템 리소스를 효율적으로 사용할 수 있고, I/O작업이 완료될 때 까지 기다리는 대신 다른 event를 처리할 수 있다는 장점이 있다. 그렇지만 코드 내용을 확인했을 때, thread-based에 비해 적은 내용이지만 코드가 복잡하다는 것을 확인했다. Thread-based는 순차적으로 각 thread가 작동하고, 코드 또한 순차적이어서 구현하기가 쉽다고 판단된다. 또한 thread를 이용하면 동시에 여러 작업을 처리할 수 있어 시스템 효율성과 반응성을 향상시킬 수 있다고 생각된다. 하지만 thread기반 서버는 client 수가 증가함에 따라 성능이 저하됨을 알 수 있다. 많은 수의 thread는 context switching에 대한 overhead를 증가시킬 수 있다는 단점이 있다.