Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 임해진

학번 : 20180223

1. 개발 목표

본 프로젝트에서는, 주식을 사고 팔 수 있는 stock server을 구현했다. 이때, 여러 client와의 동시 접속 및 concurrent한 서비스를 제공하는 concurrent한 stock server을 구현하였다. stock server는 주식 정보 조회, 주식 매입, 주식 매수 총 3가지의 서비스를 제공한다. event-base, thread-base approach로 두 개의 stock server를 구현한 후, 다양한 기준점에서 성능 측정을 실행했다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   이벤트 기반으로 여러 클라이언트의 서비스를 동시에 처리한다. 클라이언트의 요청이 들어오면, 이벤트 풀에 그 요청을 저장하고 차례대로 들어온 요청을 처리하며 concurrent하게 서비스를 제공한다.

2. Task 2: Thread-based Approach

   스레드 기반으로, 미리 스레드 풀을 생성한 후에 클라이언트 요청이 들어오면 스레드 중 하나가 그 클라이언트와 연결을 맺는다. 스레드는 클라이언트에게서 들어온 요청을 처리하고 결과값을 반환해준다.

3. Task 3: Performance Evaluation

   클라이언트 요청 수에 따른 분석, 워크로드에 따른 분석, 클라이언트 별 요청 수에 따른 분석 등 다양한 관점에서 서버의 성능을 측정한 후 분석한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  I/O Multiplexing 기법은 여러 client의 요청을 event로 간주해서, concurrent하게 처리하는 방법 중 하나이다. select()라는 함수를 사용해서, I/O 요청이 들어온 fd를 찾은 후, 차례대로 들어온 요청을 처리해주는 방식이다. client fd를 관리하는 배열 자료구조와 fd_set 자료구조를 사용한다.

먼저 listen_fd에서 요청이 들어왔다면, client fd를 자료구조에 추가해준다. 이후 for문을 사용해서, 모든 client fd를 순회하며 I/O요청이 들어왔다면 해당 요청을 받아서 처리해준다. 이때 요청은 명령어 하나씩만 수행하도록 구현한다. 그 후 커넥션이 끊기면 다시 client fd를 자료구조에서 제거해 처리하지 않도록 한다.

✓ epoll과의 차이점 서술

select()함수와 epoll()함수 모두 fd에 I/O 이벤트가 발생했는지 감지해주는 함수이다. select()함수는 fd_set 자료구조를 사용해서, I/O 이벤트가 발생한 fd의 정보를 넘겨준다. 그렇기 때문에 fd_set자료구조를 직접 관리해야하고, 반복문을 사용해서 모든 fd의 상태를 직접 체크해서 처리해주어야 한다. 또한 select()함수를 호출해 매번 fd_set을 넘겨주어야 한다.

select()함수의 번거로움을 보완한 것이 epoll()함수이다. 프로그램에서 직접 fd_set을 관리하지 않고, 커널이 이 일을 대신 해준다. fd_set과 같이 fd를 관리하는 자료구조를 커널에게 만들어달라고 요청하면(epoll_create), epoll_fd를 리턴받고 이 값으로 이후의 요청을 호출한다. 새로운 fd를 등록하거나 제거하는 과정도 함수 호출(epoll_ctl)로 처리된다. epoll_wait을 호출하면, 이벤트가 발생한 fd만 반환이되어, select()함수와 다르게 모든 fd를 순회하면서 이벤트 발생을 체크하지 않아도 되는 장점이 있다. 또한 커널레벨에서 fd를 관리하기 때문에 사용자 프로그램에서 fd_set등의 자료구조를 넘겨주지 않아도 된다.

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

✓ Master Thread의 Connection 관리

master thread는 클라이언트 요청이 생기면, 그 요청을 worker thread에 넘겨주는 역할을 한다. while()문을 통해 무한 순회하면서, listen_fd에 클라이언트 요청이 생기면 클라이언트 fd를 공용 메모리인 buffer로 넘겨준다. 이때 buffer는 여러 worker thread가 모두 공유하는 자료구조이기 때문에 synchronization기법을 사용해서 race를 방지해주고 남은 버퍼 공간, 새로 들어온 fd 개수를 업데이트해주어야 한다. 그래서 새로운 요청이 들어오면, buffer에 남은 공간이 생길 때까지 대기한 후에, 남은 공간을 사용해서 buffer를 업데이트 해주고 다른 thread가 감지할 수 있도록 새로 들어온 fd의 개수를 업데이트 해준다. 또한 buffer 에 값을 업데이트 할 때는 뮤텍스를 이

용해서, 상호배제적으로 메모리에 접근하는 것을 보장한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

server프로그램이 시작되면 worker thread를 여러개 생성해서 pool로 만들어 놓은 후에, 클라이언트 요청이 들어오면 pool에 있는 thread 중 하나가 해당 클라이언트 요청을 처리하게 된다. 매번 새로운 client요청이 들어올 때 thread를 생성하게 되면 thread생성과 삭제에 overhead가 생기고 response time이 늦어지기 때문에, 미리 pool에 스레드를 만들어놓고 사용한 후 다시 반환하도록 구현한다.

모든 worker thread는 새로운 fd의 개수가 올라갈 때까지 기다리다가, main thread가 buffer에 새로운 fd를 저장하면, thread중 하나가 그 fd를 받아서 처리하게 된다. 클라이언트의 모든 요청을 처리하고 커넥션이 해제되면 thread는 다시 처음처럼 새로운 fd를 기다리게 된다.

- **Task3 (Performance Evaluation)**

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

1. 클라이언트 수에 따른 분석

클라이언트 수에 따른 분석은 event-based방식과 thread-based방식 간의 처리율을 비교하기 위해 수행한다. 클라이언트 수가 작을 때랑, 많을 때 각 방식 중 어느 방식이 효율적인지 그리고 클라이언트 수가 늘어남에 따라서 처리시간이 어떻게 증가하는지 비교할 수 있다. 이는 이후 많은 클라이언트가 접속하는 확장된 서버를 구축할 때 더 효율적인 아키텍처를 고르는 근거가 될 것이다. 이를 위해서 클라이언트 수가 1, 5, 10, 20, 40, 60, 80, 100일 때 모든 클라이언트의 요청을 처리하는데 얼마나 걸리는지 측정한다. 이때 네트워크 상황에 따라 처리 시간이 달라질 수 있기 때문에 10번 측정 후 최댓값과 최소값을 제외한 데이터로 평균값을 내 비교한다.

2. 워크로드에 따른 분석

워크로드에 따른 분석은, 클라이언트의 요청이 read위주인지 update 위주인지에 따라서 처리율이 얼마나 달라지는지 알기 위해 수행한다. event-based방식은 lock을 하지 않고, thread-based방식은 node단위의 lock을 하기 때문에 어느 방법이 read 위주 그리고 update 위주 요청에 효율적인지 알 수 있다.

이를 통해서 read요청과 update요청에 따라 처리 서버를 분리하게 되었을 때 어떤 처리 방식을 사용해야 할지 판단할 수 있다. 모든 실험에는 50개의 클라이언트를 생성하고, 각 클라이언트는 10개의 요청을 전송한다. show, buy, sell 모두를 랜덤하게 고르게, 또는 show만 요청하게, 또는 buy&sell만 요청하도록 multiclient파일을 수정해서 실험을 진행한다. 마찬가지로 네트워크 상황에 따른 오차를 줄이기 위해 10번을 측정하고 최댓값과 최솟값을 제외한 데이터로 평균을 낸다.

3. 같은 요청 수 내의 클라이언트 수에 따른 분석

같은 개수의 요청을 받더라도, 클라이언트 개수에 따라 처리 시간이 달라질 수 있다. event-based는 클라이언트 fd를 따로 저장하고, 이를 순회하면서 i/o를 처리한다. 또한 thread-based방식은 각 클라이언트를 thread가 context switching하면서 처리하기 때문에 overhead가 발생한다. 그래서 이 실험은, 각 방법의 overhead를 비교하고, 여러 client의 요청을 concurrent하게 실행하는 것이 얼만큼의 성능 개선을 가져오는지 알 수 있다. 모든 실험은 총 500번의 요청을 보낸 시간을 측정한다. 각 1개의 클라이언트가 500개의 요청, 5개가 100개, 10개가 50개, 20개가 25개, 100개가 5개의 요청을 보내는 경우에 따라 처리시간을 측정한다. 마찬가지로 네트워크 상황에 따른 오차를 줄이기 위해 10번을 측정하고 최댓값과 최솟값을 제외한 데이터로 평균을 낸다.

✓ Configuration 변화에 따른 예상 결과 서술

1. 클라이언트 수에 따른 분석

당연히 클라이언트 수가 늘어날 수록 두 방법 모두다 처리 시간은 늘어날 것이다. 클라이언트 수가 적을 때는, thread로 얻는 concurrency가 상대적으로 덜 중요하고 context switch의 overhead로 인해 event-based방식이 조금 더 빠르거나 두 방법 사이의 시간차가 없을 것으로 예상된다. 하지만 클라이언트 수가 많아지면 event-based방법에서는 모든 클라이언트의 fd를 관리하면서 순회하며 처리하기 때문에 속도가 느려질 것이고, 반면 thread-based 방법은 multi-core 환경에서 parallel하게 실행되어 처리 속도가 크게 느려지지 않을 것으로 예상된다. 따라서 중간 시점부터 thread-based처리 시간이 event-based 처리 시간보다 크게 작아지게 될 것으로 예상된다.

2. 워크로드에 따른 분석

event-based방식에서는 동기화를 하지 않기 때문에 show나 buy&sell요청에 따라 처리 시간의 차이가 크게 없을 것으로 예상된다. 다만 show는 tree의 모든 노드를 탐색해서 읽는 작업이기 때문에 시간 복잡도가 O(N)이고 buy&sell은 노드를 찾기만 하면 되기 때문에 평균적으로 시간 복잡도가 O(logN)이다. 따라서 buy&sell 위주의 요청이 더 빠르게 실행될 것으로 예상된다.

thread-based방식에서는 show는 read작업이어서 모든 요청이 lock되지 않고 실행된다. 하지만 buy&sell은 write작업으로 lock이 되기 때문에 동시 처리율이 떨어질 것이다. 따라서 buy&sell 작업이 더 늦게 실행될 것으로 예상된다.

두 방법 다 show&buy&sell 모두 요청하는 경우에는, show위주와 buy&sell위주의 경우의 중간 값을 가질 것으로 예상된다.

3. 같은 요청 수 내의 클라이언트 수에 따른 분석

먼저 클라이언트 수가 작은 경우에는 concurrent한 실행의 장점을 가지지 못하기 때문에, 처리 속도가 상대적으로 느릴 것으로 예상한다. 반면 클라이언트 수가 커지면 concurrent하게 실행이 되기 때문에 처리 속도가 빨라질 것이다. 하지만 thread-based 방법같은 경우 클라이언트 수가 많을 때 스레드 간의 context switch와 thread pool의 모든 thread를 다 쓴 경우 기다려야하기 때문에, 특정 시점에서 처리 속도가 느려질 것으로 예상된다. event-based 방법 같은 경우에는, 결국 여러 요청을 하나씩 처리하게 때문에 client개수가 큰 경우와 작은 경우가 큰 차이가 없을 것으로 예상된다.

C. 개발 방법

[Task1]

clientfd를 관리하기 위해 pool 구조체를 사용했다. pool구조체에는 clientfd를 저장하는 배열과 read_set, ready_set등 fd_set 자료구조와 최대 인덱스 값 등이 저장된다. init_pool()함수를 정의해서, pool 구조체를 초기화해준 다음에 add_client()함수를 정의해 새로운 클라이언트와 커넥션을 맺게 되면 pool구조체에 client fd를 추가하는 로직을 구현했다. 또한 클라이언트와 커넥션이 해지되면 pool구조체에 해당 client fd를 제거하는 close_client()함수도 구현했다. 모든 fd를 순회하면서 io 이벤트가 발생한 경우 명령어를 받아서 처리해주는 check_clients()함수도 구현하였다.

각 명령어를 처리하기 위해 execute_command()함수를 구현했다. 이 함수에서는 받은 명령어를 parse_input()함수로 파싱해서 적절한 함수를 호출해준다. show, buy, sell명령어를 처리하기 위한 함수를 구현하고, 각 함수에서는 BST관련 함수들을 호출해서 클라이언트의 요청을 수행하고, 다시 응답해준다.

주식 정보를 저장하기 위해 binary tree node 구조체를 정의했다. 각 노드는 주식 정보과 left, right child에 대한 포인터가 있다. 프로그램이 시작되는 시점에 stock.txt를 읽어서 BST를 구성하는 make_tree()함수를 구현했다. make_tree()에서는 한 줄을 읽어서 BST의 알맞은 위치에 삽입하는 add_node()를 호출한다. buy&sell 명령어에서 사용되는 노드를 이분 탐색하는 로직은 find_node()함수로 구현했다. print_tree()함수는 모든 노드를 preorder순서로 탐색해서 문자열에 concatenate하게 된다. 모든 connection이 종료되면 save_tree()함수를 호출해서 업데이트된 주식 정보를 stock.txt에 쓰게 된다. signal_handler를 구현해서 SIG_INT 시그널이 오는 경우에도 save_tree()가 호출되도록 구현했다.

[Task2]

BST 로직과, 명령어 처리 부분은 Task1과 동일하게 구현하였다. main thread와 worker thread간에 producer consumer problem을 구현하기 위해 sbuf_t 구조체를 구현했다. sbuf_t에는 버퍼, 버퍼의 앞부분, 뒷부분, 크기 등의 정보가 저장되고 비어있는 칸, 사용되지 않은 칸를 저장하는 세마포어와 버퍼의 접근을 제어하는 뮤텍스가 있다.

sbuf_t 구조체를 초기화하는 sbuf_init()함수를 구현했다. 또한 main thread에서 클

라이언트 요청이 들어오면 버퍼에 fd를 추가해주는 sbuf_insert()함수를 구현했다. sbuf_insert()에서는 버퍼에 빈칸이 생기면 새로운 fd를 추가하고 이를 worker thread에 알리는데 이때 뮤텍스를 사용해서 상호배제를 보장한다. worker thread가 버퍼에 새로운 fd가 추가되면 이를 가져가서 사용하는 sbuf_remove()함수를 구현했다. sbuf_remove()는 버퍼에 새로운 아이템이 추가되면 이를 가져가 사용하고 빈칸의 존재를 알린다. 이때도 마찬가지로 뮤텍스를 사용해서 상호배제를 보장한다. 프로그램이 시작되면 worker thread를 생성해서 모든 스레드가, 새로운 fd를 기다리도록 하고 while()을 돌면서 새로운 클라이언트가 생기면 fd를 버퍼에 추가해 worker thread중 하나가 처리하도록 구현했다.

node에 접근할 때 readers-writers problem를 구현하기 위해서 각 node에 대한 동기화를 구현했다. tree node에 뮤텍스, readcnt, w뮤텍스를 추가했다. buy or sell 명령어로 node를 수정할 때는 w_mutex값을 사용해서 상호배제를 보장했다. show명령어로 node를 읽을 때는 readcnt를 증가해주고, 노드를 읽는 thread가 없었다면 wmutex를 lock해서 수정 작업이 중간에 실행되지 않도록 했다. 또한 읽기 작업을 모두 마치면 wmutex를 해제해서 수정 작업이 실행되도록 구현하였다. 이때 readcnt는 여러 thread가 공용으로 접근하기 때문에 뮤텍스를 사용해서 상호배제적으로 readcnt를 수정하도록 구현하였다.

[Task3]

실험을 위해서 multi client파일을 수정하였다. gettimeofday함수를 사용해서 클라이언트 생성 전에 시간을 측정하고, 모든 클라이언트 요청이 끝나고 모든 child process가 ripping된 시간을 측정했다. 두 시간차를 계산해서 모든 요청을 보내고, 응답받는데 걸린 시간을 계산하였다. 이때 실험의 편의를 위해서 for문을 사용해서 걸린 시간을 10번 계산하고, 최대값과 최소값을 구해서 두 값을 제외한 평균값을 출력하는 로직을 추가하였다.

3. 구현 결과

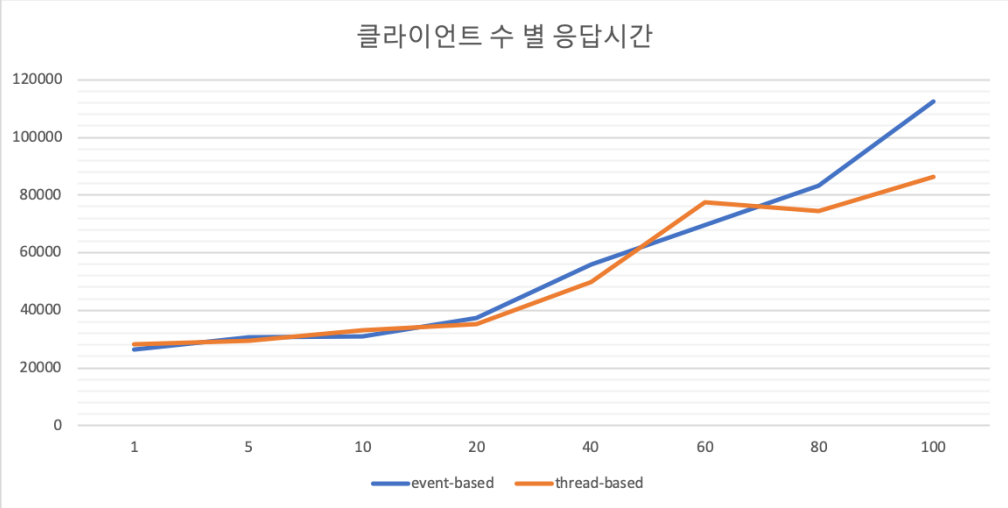
Task1, Task2를 모두 구현한 후 stockclient 프로그램을 사용해서 요청을 보내 구현 결과를 확인했다. show 요청에는 차례대로 모든 node를 순회해서 모든 주식 정보를 잘 전송해주었다. 또한 buy와 sell 요청에는 이를 처리한 후 “success” 메세지를 잘 전송해주었다. 클라이언트 연결을 해제한 경

우에는, 서버에서 연결이 해제되었고 모든 연결이 해제되었을 때 stock.txt의 값이 성공적으로 업데이트 되었다. Task3를 구현한 후 여러 테스트를 성공적으로 진행할 수 있었다. 테스트 결과를 예측한 결과와 비교하면서 여러 시사점을 찾을 수 있었다.

4. 성능 평가 결과 (Task 3)

1. 클라이언트 수에 따른 분석

클라이언트 수가 1, 5, 10, 20, 40, 60, 80, 100일 때 모든 클라이언트의 요청을 처리하는데 얼마나 걸리는지 측정했으며 결과는 아래 표와 같다. 자세한 데이터와 출력값 캡쳐 결과는 부록(a)에서 확인 가능하다.

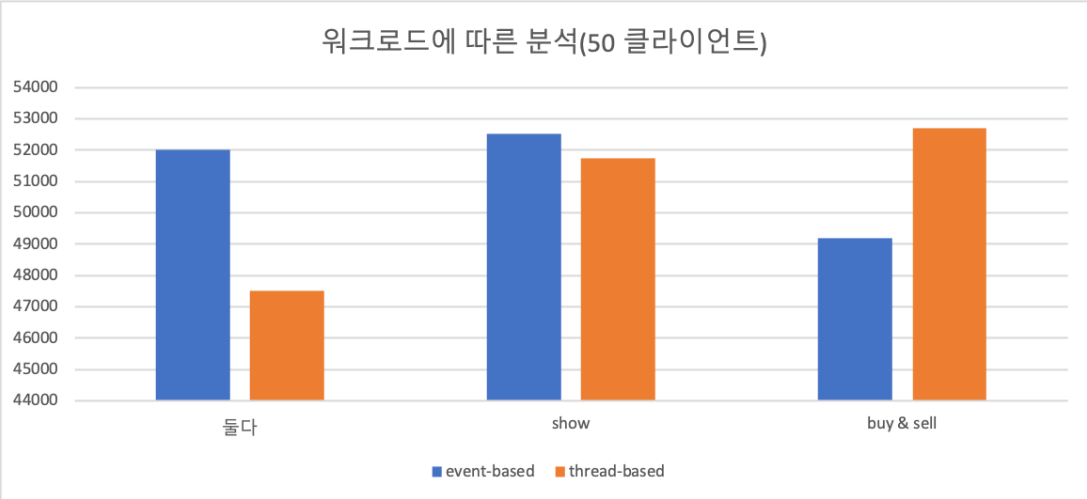


클라이언트 수가 작을 때는 두 방법 모두 응답 시간이 비슷하다. 하지만 클라이언트 숫자가 많아질 수록 thread-based방법이 event-based방법보다 효율적이었고, 클라이언트 숫자가 커질 수록 그 격차가 늘어나는 경향이 있다. 이는 클라이언트 수가 많아질 수록 multicore에서 thread가 parallel하게 실행되어 효율성이 증가한 것으로 보인다. 클라이언트 수가 적은 경우에는 thread의 overhead로 multicore로 얻는 효율성이 부각되지 않았지만 클라이언트 수가 많아질 수록 parallelism의 장점이 뚜렷해진 것이다. 다만 그래프에서 보여지듯이, 클라이언트 수가 큰 경우에도 큰 격차가 보이지는 않았기 때문에, thread-based의 장점이 보여지기 위해서는 더 많은 클라이언트가 접속하는 대규모 서버에 적용되어야 한다.

2. 워크로드에 따른 분석

50개의 클라이언트가 10개의 요청을 보내는 상황에서 그 요청이 read인지,

update인지 또는 둘다인지에 따라 성능을 측정하였다. 결과는 아래와 같다. 자세한 데이터와 출력값 캡쳐 결과는 부록(b)에서 확인 가능하다.

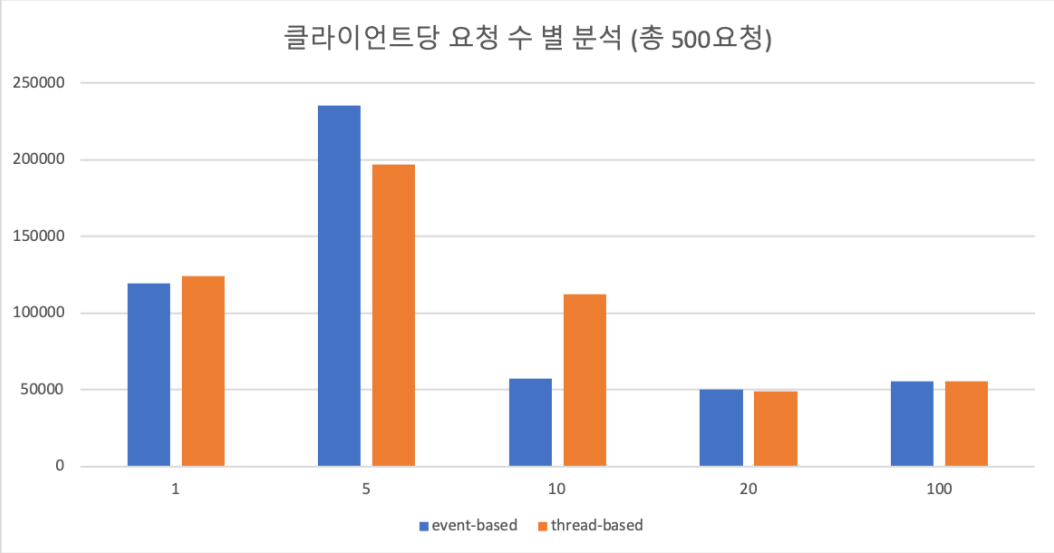


event-based 방법의 경우에는 show보다 buy&sell 요청의 경우 더 빠른 시간안에 수행되었다. 이는 예측한대로 show는 모든 node를 순회해서 출력하는데 반해 buy&sell은 이분탐색으로 노드를 찾아서 수정하는 작업이기 때문에 시간복잡도가 작아 빠르게 수행된 것으로 보여진다. 마찬가지로 둘 다 포함하는 요청을 보낸 경우에는 중간 값이 나온 것도, show와 buy&sell의 명령어가 둘 다 실행되었기 때문이라고 볼 수 있다.

thread-based 방법의 경우에는, 여러 thread가 read를 할 수 있도록 구현해서 show의 수행이 더 빠른 것을 알 수 있다. buy&sell의 경우에는 node단위로 lock이 되어서 다른 thread의 수행이 기다리는 상황이 발생할 수 있지만 show는 모두 read작업인 경우에 lock되지 않기 때문에 빠르게 실행되었다. 다만 show는 위에서 언급한 것처럼 모든 node를 탐색하는 작업이기 때문에 큰 격차가 보여지지 않았던 것으로 예상된다. 예상과 다르게, 모든 요청을 보낸 경우에는 가장 빠르게 요청이 수행된 것을 볼 수 있다. 이는 아마도 buy&sell에서 중복되는 node에 대한 작업이 동시에 일어나 기다리는 경우가 많지 않아서 buy&sell only보다 더 작은 시간이 걸린 것으로 예상된다.

3. 같은 요청 수 내의 클라이언트 수에 따른 분석

각 1개의 클라이언트가 500개의 요청, 5개가 100개, 10개가 50개, 20개가 25개, 100개가 5개의 요청을 보내는 경우에 따라 처리시간을 측정했고 결과는 아래와 같다.



클라이언트 수가 5였을 때 두 방법 모두 가장 긴 응답시간이 소요되었으며, 10, 20, 100인 경우에는 응답속도가 가장 작았다. 클라이언트 수가 많은 경우에는 concurrent하게 요청이 실행되어서 빠른 응답속도를 보인 것으로 예상된다. 클라이언트 수가 5인 경우에는 concurrency의 장점보단 overhead가 더 많았기 때문에 응답 시간이 가장 느리다. 클라이언트 수가 1인 경우에는 concurrency하게 처리되지는 않지만, overhead또한 0에 가깝기 때문에 5인 경우보다 더 빠르게 실행되었던 것으로 보여진다. event-based 방법 같은 경우에는 결국 single core로 모든 요청을 한줄 씩 읽어 처리하는 방식이기 때문에 모든 경우에도 응답시간이 차이가 없을 것으로 예상했다. 하지만 event-based 방식에서도 client수가 작을 때는 앞 요청의 응답을 받은 후 요청을 보내기 때문에 속도가 더 느려지는 것으로 판단된다. 반편 client 수가 많으면 기다려야 하는 응답의 수가 적어지기 때문에 multi core의 장점을 사용하지 못해도 concurrency의 효율성이 보여진다. 따라서 두 방법 모두다 클라이언트 수가 10개 이상인 경우에 concurrency로 인한 장점이 overhead보다 커진다고 볼 수 있다.

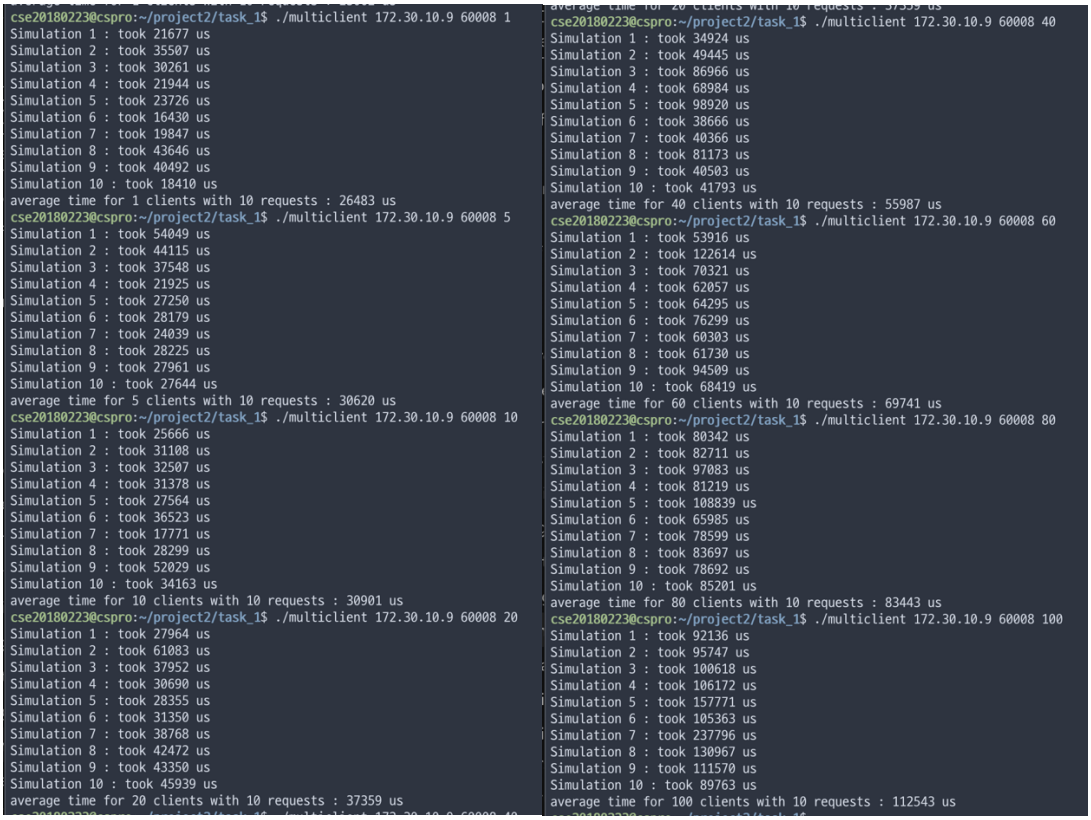
5. 부록

(a) 클라이언트 수 별 분석

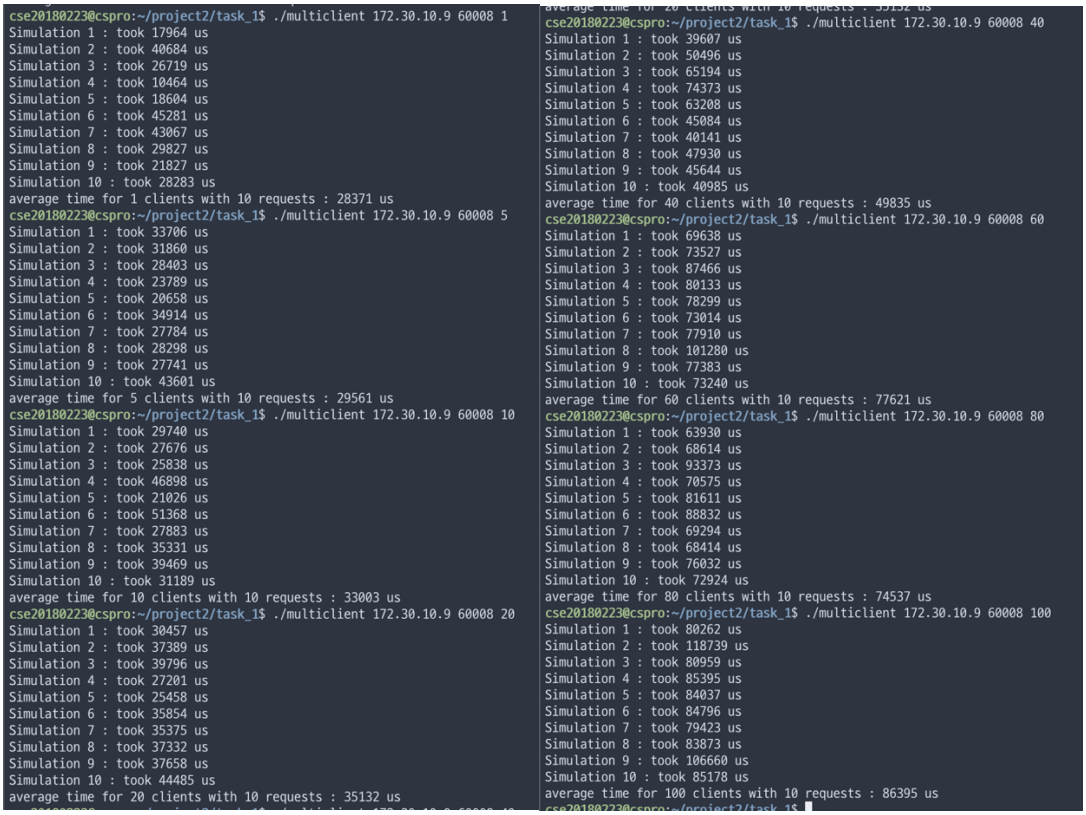
(a)-1 표 데이터

클라이언트 수 별 응답시간, 각 10요청씩								
	1	5	10	20	40	60	80	100
event-based	26483	30620	30901	37359	55987	69741	83443	112543
thread-based	28371	29561	33003	35132	49835	77621	74537	86395

(a)-2 event-based 실험 캡쳐



(a)-3 thread-based 실험 캡쳐

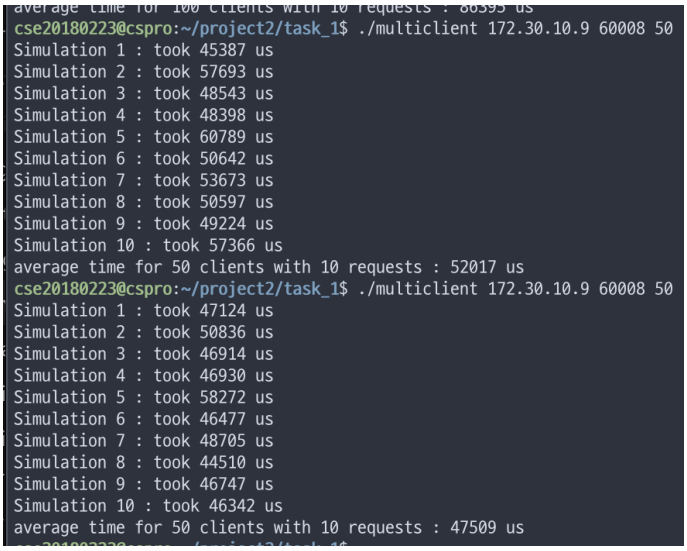


(b) 워크로드에 따른 분석

(b)-1 표 데이터

워크로드에 따른 분석 50개의 클라이언트, 각 10요청			
	둘다	show	buy & sell
event-based	52017	52513	49181
thread-based	47509	51751	52697

(b)-2 둘다



(b)-3 show 요청만

```
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 50
Simulation 1 : took 45612 us
Simulation 2 : took 52247 us
Simulation 3 : took 57690 us
Simulation 4 : took 47944 us
Simulation 5 : took 61229 us
Simulation 6 : took 55246 us
Simulation 7 : took 48139 us
Simulation 8 : took 57499 us
Simulation 9 : took 52960 us
Simulation 10 : took 48383 us
average time for 50 clients with 10 requests : 52513 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 50
Simulation 1 : took 43714 us
Simulation 2 : took 52267 us
Simulation 3 : took 50043 us
Simulation 4 : took 71053 us
Simulation 5 : took 46073 us
Simulation 6 : took 43699 us
Simulation 7 : took 88201 us
Simulation 8 : took 52397 us
Simulation 9 : took 54284 us
Simulation 10 : took 44178 us
average time for 50 clients with 10 requests : 51751 us
```

(b)-4 buy&sell 요청만

```
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 50
Simulation 1 : took 46636 us
Simulation 2 : took 50994 us
Simulation 3 : took 50432 us
Simulation 4 : took 48638 us
Simulation 5 : took 47677 us
Simulation 6 : took 55432 us
Simulation 7 : took 50874 us
Simulation 8 : took 47285 us
Simulation 9 : took 48131 us
Simulation 10 : took 49419 us
average time for 50 clients with 10 requests : 49181 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 50
Simulation 1 : took 48105 us
Simulation 2 : took 50657 us
Simulation 3 : took 64585 us
Simulation 4 : took 56421 us
Simulation 5 : took 45426 us
Simulation 6 : took 49182 us
Simulation 7 : took 46281 us
Simulation 8 : took 59254 us
Simulation 9 : took 66447 us
Simulation 10 : took 47091 us
average time for 50 clients with 10 requests : 52697 us
```

(c) 같은 요청 수 내의 클라이언트 수에 따른 분석

(c)-1 표데이터

요청 수 대비 클라이언트 수 -> 500번의 요청					
	1	5	10	20	100
event-based	119559	235004	57392	50192	55420
thread-based	124019	197091	112135	49161	55677

(c)-2 event-based 실험 캡쳐

```
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 1
Simulation 1 : took 128535 us
Simulation 2 : took 119332 us
Simulation 3 : took 107145 us
Simulation 4 : took 156189 us
Simulation 5 : took 117348 us
Simulation 6 : took 116833 us
Simulation 7 : took 124543 us
Simulation 8 : took 115296 us
Simulation 9 : took 120198 us
Simulation 10 : took 114391 us
average time for 1 clients with 500 requests : 119559 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 5
Simulation 1 : took 335200 us
Simulation 2 : took 108507 us
Simulation 3 : took 304871 us
Simulation 4 : took 345642 us
Simulation 5 : took 125043 us
Simulation 6 : took 349334 us
Simulation 7 : took 368369 us
Simulation 8 : took 230547 us
Simulation 9 : took 94282 us
Simulation 10 : took 238506 us
average time for 5 clients with 100 requests : 254706 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 5
Simulation 1 : took 280829 us
Simulation 2 : took 170936 us
Simulation 3 : took 251014 us
Simulation 4 : took 238584 us
Simulation 5 : took 256392 us
Simulation 6 : took 264441 us
Simulation 7 : took 179347 us
Simulation 8 : took 165412 us
Simulation 9 : took 239115 us
Simulation 10 : took 280203 us
average time for 5 clients with 100 requests : 235004 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 10
Simulation 1 : took 53794 us
Simulation 2 : took 60221 us
Simulation 3 : took 51751 us
Simulation 4 : took 54294 us
Simulation 5 : took 62800 us
Simulation 6 : took 78363 us
Simulation 7 : took 61669 us
Simulation 8 : took 62356 us
Simulation 9 : took 49882 us
Simulation 10 : took 52252 us
average time for 10 clients with 50 requests : 57392 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 20
Simulation 1 : took 45824 us
Simulation 2 : took 66078 us
Simulation 3 : took 50278 us
Simulation 4 : took 53871 us
Simulation 5 : took 49635 us
Simulation 6 : took 50098 us
Simulation 7 : took 50832 us
Simulation 8 : took 47426 us
Simulation 9 : took 50138 us
Simulation 10 : took 49259 us
average time for 20 clients with 25 requests : 50192 us
```

```
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 100
Simulation 1 : took 50690 us
Simulation 2 : took 64558 us
Simulation 3 : took 52869 us
Simulation 4 : took 53099 us
Simulation 5 : took 54948 us
Simulation 6 : took 63004 us
Simulation 7 : took 68568 us
Simulation 8 : took 53647 us
Simulation 9 : took 50473 us
Simulation 10 : took 50549 us
average time for 100 clients with 5 requests : 55420 us
```

(c)-3 thread-based 실험 캡쳐

```
average time for 100 clients with 5 requests : 53420 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 1
Simulation 1 : took 108449 us
Simulation 2 : took 139076 us
Simulation 3 : took 106863 us
Simulation 4 : took 119832 us
Simulation 5 : took 122153 us
Simulation 6 : took 129790 us
Simulation 7 : took 150250 us
Simulation 8 : took 152763 us
Simulation 9 : took 106933 us
Simulation 10 : took 115672 us
average time for 1 clients with 500 requests : 124019 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 5
Simulation 1 : took 192501 us
Simulation 2 : took 208848 us
Simulation 3 : took 189016 us
Simulation 4 : took 196030 us
Simulation 5 : took 211552 us
Simulation 6 : took 194756 us
Simulation 7 : took 210559 us
Simulation 8 : took 104353 us
Simulation 9 : took 194277 us
Simulation 10 : took 190741 us
average time for 5 clients with 100 requests : 197091 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 10
Simulation 1 : took 154696 us
Simulation 2 : took 88766 us
Simulation 3 : took 132834 us
Simulation 4 : took 112772 us
Simulation 5 : took 96323 us
Simulation 6 : took 128881 us
Simulation 7 : took 129210 us
Simulation 8 : took 113818 us
Simulation 9 : took 94480 us
Simulation 10 : took 71515 us
average time for 10 clients with 50 requests : 112135 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 20
Simulation 1 : took 51046 us
Simulation 2 : took 49646 us
Simulation 3 : took 47626 us
Simulation 4 : took 50019 us
Simulation 5 : took 47829 us
Simulation 6 : took 48225 us
Simulation 7 : took 50711 us
Simulation 8 : took 57460 us
Simulation 9 : took 48191 us
Simulation 10 : took 45314 us
average time for 20 clients with 25 requests : 49161 us
cse20180223@cspro:~/project2/task_1$ ./multiclient 172.30.10.9 60008 100
Simulation 1 : took 49828 us
Simulation 2 : took 79019 us
Simulation 3 : took 56948 us
Simulation 4 : took 49175 us
Simulation 5 : took 64659 us
Simulation 6 : took 48321 us
Simulation 7 : took 60368 us
Simulation 8 : took 63553 us
Simulation 9 : took 51843 us
Simulation 10 : took 49044 us
average time for 100 clients with 5 requests : 55677 us
cse20180223@cspro:~/project2/task_1$
```