Multicore Programming Project 2

담당 교수 : 박성용

이름 : 엄석훈

학번 : 20181536

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

이번 프로젝트에서는 여러 클라이언트가 서버에 접속하여 주식 서비스를 이용할 수 있도록 concurrent 주식 서버를 만든다. 서버는 클라이언트가 요청하는 명령어(show, buy, sell, exit)에 해당하는 명령을 수행하고 해당 명령에 따른 결과를 클라이언트에게 전달한다. 그리고 주식서버는 시작될 때 주식정보를 stock.txt에서 불러와 메모리에 저장하고 있다가 서버가 종료될 때 업데이트된 주식 정보를 stock.txt로 메모리에 저장하여 관리한다. 그리고 주식 서버는 event-driven 방식과 thread-based 방식 총 2가지의 방식으로 만들어 보고 서로 비교해본다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   Select 함수를 사용하여 I/O Multiplexing을 구현한다. 먼저 active되어 있는 클라이언트의 리스트를 pool에 저장하여 유지하며 관리한다. 그리고 서버는 루프를 돌면서 select 함수를 통해 새로운 연결이 요청되거나 pool에 저장된 connfd에서 들어온 명령어를 파악하고 요청을 차례대로 수행한다. 여러 클라이언트로부터 들어온 요청을 차례대로 수행하기 때문에 동시에 명령어가 처리되는 걱정은 하지 않아도 된다.

2. Task 2: Thread-based Approach

   서버가 시작될 때 미리 thread를 만들어 두고 클라이언트로부터 연결이 요청 들어오면 subf에 디스크립터를 삽입한다. 그리고 쓰레드는 해당 sbuf에서 디스크립트를 가져가 명령어에 따른 요청을 수행하고 클라이언트에게 명령어 수행 결과를 보내준다. 여러 클라이언트의 명령을 동시에 수행할 수 있지만 주식 서버의 데이터가 동기화 되도록 세마포어를 이용한다.

3. Task 3: Performance Evaluation

Task 1과 Task 2에서 구현한 Event-driven Approach와 Thread-based Approach를 통해 구현한 서버를 비교해본다. 클라이언트의 수에 따른 동시 처리율을 비교하고 워크로드에 따른 동시 처리율도 분석한다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

다양한 클라이언트의 요청을 받기 위해서 listenfd와 connfd를 저장하는 pool을 만들어 준다. 처음에는 pool에 listenfd만 넣어두고 서버에서 무한루프를 돌면서 select함수를 호출한다. select함수는 pool에 저장된 fd에서 요청이 들어오면 pool의 fd를 순차적으로 확인하면서 요청을 처리해준다. listenfd에 클라이언트로부터 연결 요청이 들어오면 add_client함수를 통해서 pool에 새로운 connfd를 추가해주고 connfd에 명령어 요청이 들어오면 해당 명령어를 확인해 명령어에 따라 동작을 수행해준다.

✓ epoll과의 차이점 서술

select 함수는 fd를 하나씩 차례대로 확인하면서 요청이 들어와 있는지 확인하기 때문에 select함수를 호출할 때 마다 pool의 fd정보 전체를 넘겨주어야 한다. 따라서 관리하는 클라이언트가 많이질수록 넘겨야 하는 데이터도 커지고 수행 시간이 길어진다는 단점이 있다. 반면 epoll함수는 운영체제가 직접 fd를 저장하고 있기 때문에 함수를 호출할 때 fd정보를 넘겨줄 필요도 없고 따라서 fd를 순회하면서 요청이 들어왔는지 확인 할 필요도 없다. 따라서 select함수에 비해서 처리 시간이 짧다는 장점이 있다.

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

  ✓ Master Thread의 Connection 관리

  마스터 쓰레드는 먼저 sbuf_init함수를 통해서 sbuf를 초기화 해준다. 그리고 NTHREADS만큼 worker 쓰레드를 만든다. 이후 무한루프를 돌면서 accept함수로 connfd를 받고 이를 subf_insert함수를 통해서 sbuf에 넣는다. 그럼 sbuf에 저장된 connfd를 worker 쓰레드에서 가져가 요청된 명령어를 처리한다. 즉 마스터 쓰레드는 연결을 요청받고 worker 쓰레드에게 연결해주는 역할을 하게 된다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  각 worker 쓰레드는 thread함수를 수행하며 sbuf_remove함수를 통해 sbuf에서 connfd를 가져간다. 그리고 해당 connfd를 가지고 stock함수를 통해 클라이언트의 명령어를 처리한다. 그리고 connfd와 통신이 끊어지면 다시 sbuf_remove함수를 통해 새로운 connfd를 가져와 클라이언트의 명령을 수행한다. 이때 stock함수에서는 다른 쓰레드와의 동기화를 위해서 적절히 세마포어를 사용한다.

- **Task3 (Performance Evaluation)**

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

  Task 1과 task 2의 두가지 방법에 대해 client수의 변화에 따른 수행 시간을 비교해본다. 이때 시간당 cline의 처리 요청 개수인 동시 처리율을 통해 비교를 한다. Task 1은 여러 client의 요청을 순차적으로 처리하는데 반해 task 2는 여러 쓰레드에서 동시에 처리하기 그 차이를 알아보고자 한다.

  또한 buy, sell 명령어만 실행하였을 때와 show 명령어만 실행하였을 때의 차이도 비교해본다. 왜냐하면 show 명령어는 데이터의 수정이 없기 때문에 readers-writers 문제에 의해 문제가 되지 않기 때문에 show 명령어와 다른 명령어에 따른 차이를 알아본다.

  이때 측정을 위해서는 gettimeofday함수를 이용해서 client의 요청을 처리하는데 소요되는 시간을 측정한다.

✓ Configuration 변화에 따른 예상 결과 서술

먼저 일반적인 상황에서는 client수가 증가할 수 록 task 2가 더 동시 처리율 측면에서 좋은 성능을 보일 것이라고 예상된다. 왜냐하면 task 1은 실제로는 순차적으로 동작을 수행하고 하나의 쓰레드에서 모든 작업을 수행하지만 task 2는 여러 쓰레드를 사용하기 때문에 CPU의 코어가 높아질수록 동시에 처리 가능함으로 더 빠른 동작을 할 것이라고 예상된다.

그리고 show 명령어만 수행할 때가 훨씬 더 빠른 동시 처리율을 보여줄 것이라고 생각된다. 왜냐하면 show 명령어는 각 주식 데이터에 동시에 접근할 수 있는데 반해 buy나 sell을 할 때는 한번에 하나의 데이터만 접근하여 동작을 수행할 수 있기 때문에 show만 수행할 때가 더 동시 처리율이 높을 것이라 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)**

Task 1: 주식서버의 주식 정보를 저장할 구조체인 item을 만들어 준다. 그리고 서버가 시작될 때 read_stock_data함수를 통해서 stock.txt에서 주식 서버에 필요한 데이터를 가져와 바이너리 트리를 만들어 저장한다.

그리고 fd를 관리하기 위해 pool구조체를 만들어 준다. 그리고 메인 함수에서 무한루프를 돌면서 클라이언트로부터 연결 요청이 들어오면 accept함수를 통해서 연결하고 해당 connfd를 add_client함수를 통해서 pool에 추가하여준다. 그리고 check_client함수를 통해서 pool내부에 있는 connfd에서 명령어가 들어왔는지 확인하고 show, buy, sell, exit 명령어에 따라서 동작을 수행해준다.

show명령어의 경우 show_item함수를 호출하여 binary tree를 순회하며 주식서버의 정보를 temp라는 문자열에 저장해주고 rio_writen함수를 통해서 클라이언트에게 주식 정보를 보내준다.

buy 명령어의 경우 명령어에 포함된 ID와 주식수를 구한 뒤 ID를 find_item함수에 넘겨주어 ID가 일치하는 주식 정보에 대한 정보를 받아온다. 그리고 해당 주식 정보

에서 buy하고자 하는 주식 수를 보고 서버에 남은 주식 수를 바꾸고 성공 메시지를 보내주거나 실패 메세지를 보내준다.

sell 명령어도 buy 명령어와 비슷하게 ID와 주식수를 구한 뒤 find_item함수로 주식 정보를 받아오고 남은 주식수를 바꾸고 성공 메시지를 보내준다. 마지막으로 exit 명령어의 경우 해당 connfd의 연결을 끊어준다.

그리고 sigint_handler를 만들어서 주식 서버가 종료될 때 write_stock_data함수를 호출하여 업데이트된 주식 정보를 stock.txt 파일로 저장하고 서버가 종료되도록 만들어준다.

Task 2: 여기서도 task1과 비슷하게 주식 정보를 저장할 구조체인 item을 만들고 서버가 시작될 때 read_stcok_data함수를 통해 stock.txt의 정보를 가져와 주식 서버를 위한 데이터를 바이너리 트리로 만들어 준다. 그리고 sigint_handler에서 write_stock_data함수를 호출하도록 해 서버가 종료될 때 주식 정보를 stock.txt로 다시 저장하도록 해준다.

메인 함수에서 Pthread_create함수를 통해서 미리 worker 쓰레드를 만들어 둔다. 그리고 무한 루프를 돌면서 마스터 쓰레드는 accept를 계속 받아 연결 요청이 들어오면 해당 connfd를 sbuf에 저장한다. 그리고 각 쓰레드는 sbuf에서 connfd를 가져가서 stock함수를 수행시킨다. stock함수는 클라이언트에게서 명령어를 받아서 show, buy, sell 명령어는 각각 show_item, buy_item, sell_item 함수를 호출한다.

show_item함수는 task1과 동일하게 주식정보가 저장된 바이너리 트리를 순회하면서 주식의 정보를 temp 문자열에 저장한다. 이때 여러 쓰레드를 사용할 때 발생할 수 있는 readers-writers 문제를 해결하기 위해서 세마포어를 사용해주었다. Reader가 우선을 동작할 수 있도록 show_item함수에서 하나의 주식 item을 접근할 때 먼저 P(&mutex)를 통해 해당 주식의 readcnt에 다른 함수에서 접근할 수 없도록 해주고 readcnt를 증가시킨다. 그리고 만약 첫번째 read였다면 P(&w)를 통해 writer가 접근할 수 없도록 막아주었다. 그리고 V(&mutex)로 해당 주식에 접근할 수 있게 풀어주고 해당 주식의 정보를 읽어오고 다시 readcnt를 접근하기 위해 P(&mutex)를 하고 만약 마지막 reader였다면 V(&w)로 writer가 접근할 수 있게 해주고 V(&mutex)를 하고 해당 주식의 읽기 동작을 마쳤다. 그리고 left와 right도 재귀적으로 호출해 동일하게 동작하도록 하였다. 그리고 show_item함수를 통해 모든 주식 정보를 temp에 저장하고 나서 stock함수로 돌아와 클라이언트에게 주식 정보를 보내주었다.

buy_item함수는 먼저 주식의 left_stock에 따라 동작이 바뀌기 때문에 show_item과 비슷하게 세마포어를 통해 주식에 접근해 left_stock을 확인하고 요청한 구매 주식 수가 left_stock보다 크다면 connfd에 주식을 구매할 수 없다고 결과를 전달하였다. 구매할 수 있다면 P(&w)와 V(&w)사이에서 주식의 left_stock을 바꿔주고 결과를 connfd에 전달하였다.

sell_item함수는 간단하게 팔려고 하는 주식에 P(&w)로 다른 쓰레드가 접근하기 못하게 하고 주식의 left_stock을 바꾸고 V(&w)를 수행하고 connfd에 수행 결과를 전달하였다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

  Task 1의 event-driven 방식과 task 2의 thread-based 방식으로 주식서버를 구현하였을 때 모두 여러 클라이언트로부터 들어온 명령어에 따라서 show, buy, sell 명령어의 동작을 수행하였다.

  Task 1은 활성화 되어있는 fd를 관리하는 read_set을 바탕으로 ready_set으로 카피하여 select함수를 호출한다. 그 결과로 나온 ready_set을 바탕으로 해당 pool을 순차적으로 확인하면서 요청이 들어온 경우는 accept하거나 show, buy, sell, exit 명령어의 동작을 수행하고 결과를 명령어를 요청한 connfd에서 돌려주었다.

  Task 2는 마스터 쓰레드에서 클라이언트에서 연결 요청이 들어오면 accept를 하고 sbuf에 해당 connfd를 저장한다. 그리고 서버 시작시 생성된 worker 쓰레드에서 sbuf에서 connfd를 가져가서 클라이언트의 명령어를 받아 stock함수를 통해 show, buy, sell, exit동작을 수행하고 그 결과를 해당 connfd에 돌려준다. 이때 동기화를 위해 세마포어를 사용해 데이터에 접근할 수 있는지 없는지 관리해준다.

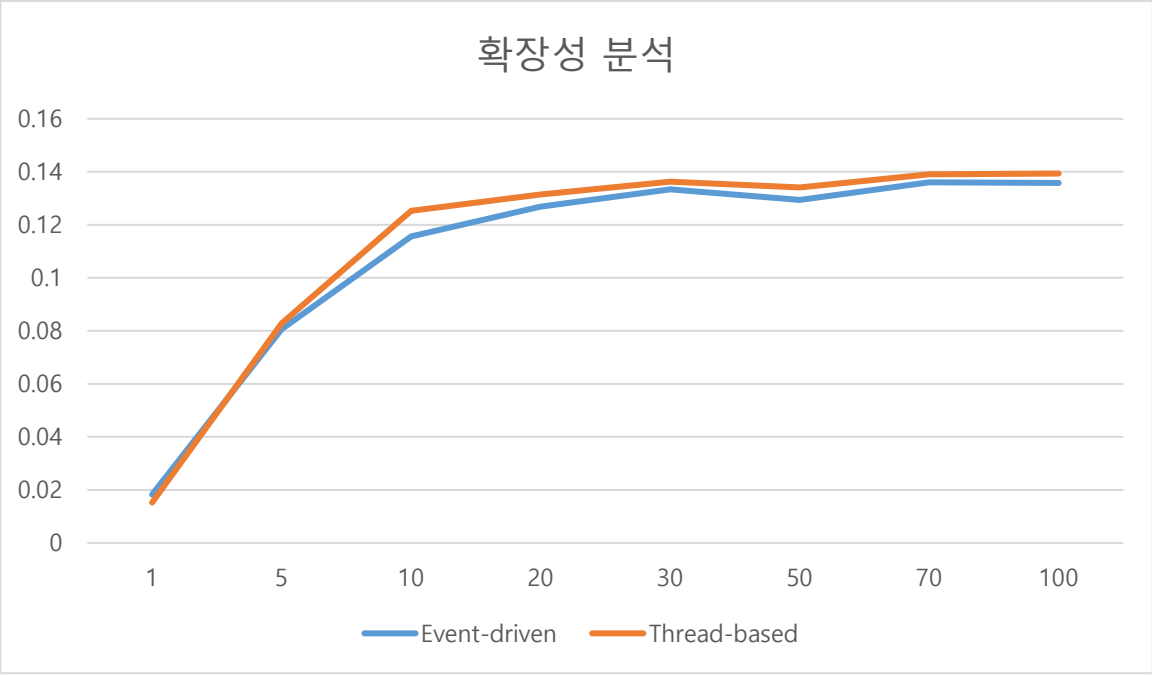
4. 성능 평가 결과 (Task 3)

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

  먼저 두가지 동작 방식의 클라이언트 개수 변화에 동시 처리율의 변화를 통해 확

작성을 확인해보기 위해 multiclient 파일을 클라이언트 수에 변화를 주며 수행해 본 결과 아래 표와 그래프 같은 결과가 나왔다. 기본적으로 두 방식 모두 요청한 명령어는 100개이고 thread-based에서는 thread를 20개를 설정하고 subf 사이즈는 128로 설정하고 측정을 수행하였다. multiclient 파일에서 delay를 삭제하고 실행할 때부터 종료될 때 까지의 시간을 통해 측정하였다.

클라이언트 수	*Event-driven Approach*	*Thread-based Approach*
1	54.890 밀리초	65.655밀리초
5	62.053 밀리초	60.376밀리초
10	86.458 밀리초	79.768밀리초
20	157.602 밀리초	152.102밀리초
30	224.824 밀리초	220.200밀리초
50	386.506 밀리초	372.651밀리초
70	514.511 밀리초	503.114밀리초
100	736.120 밀리초	717.536밀리초

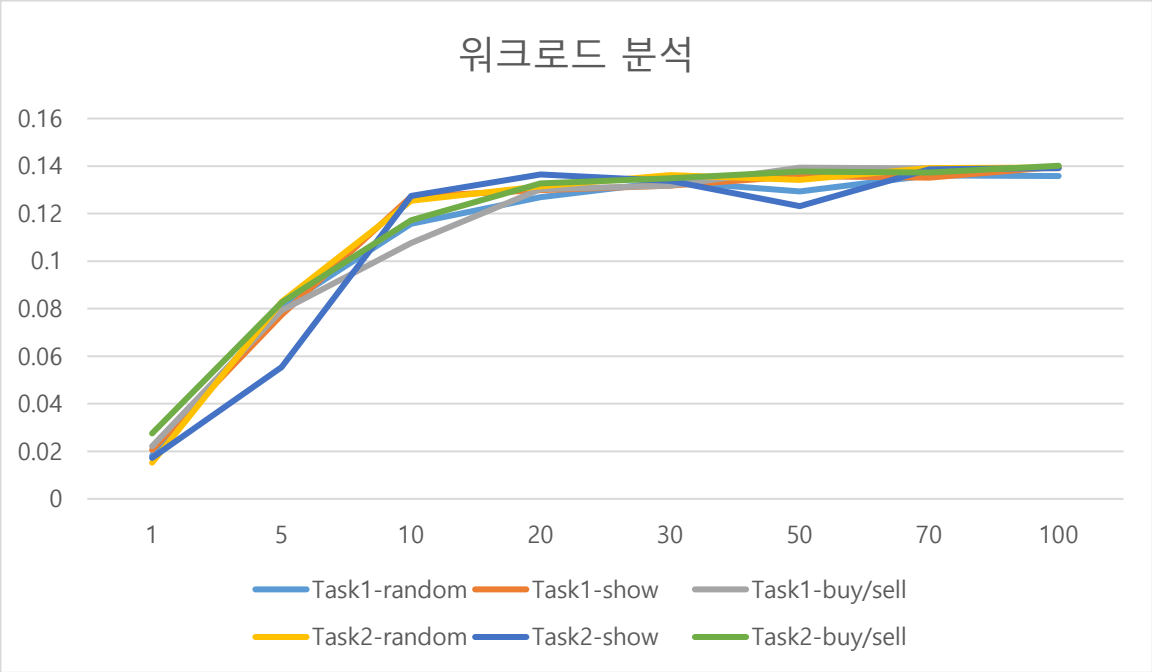


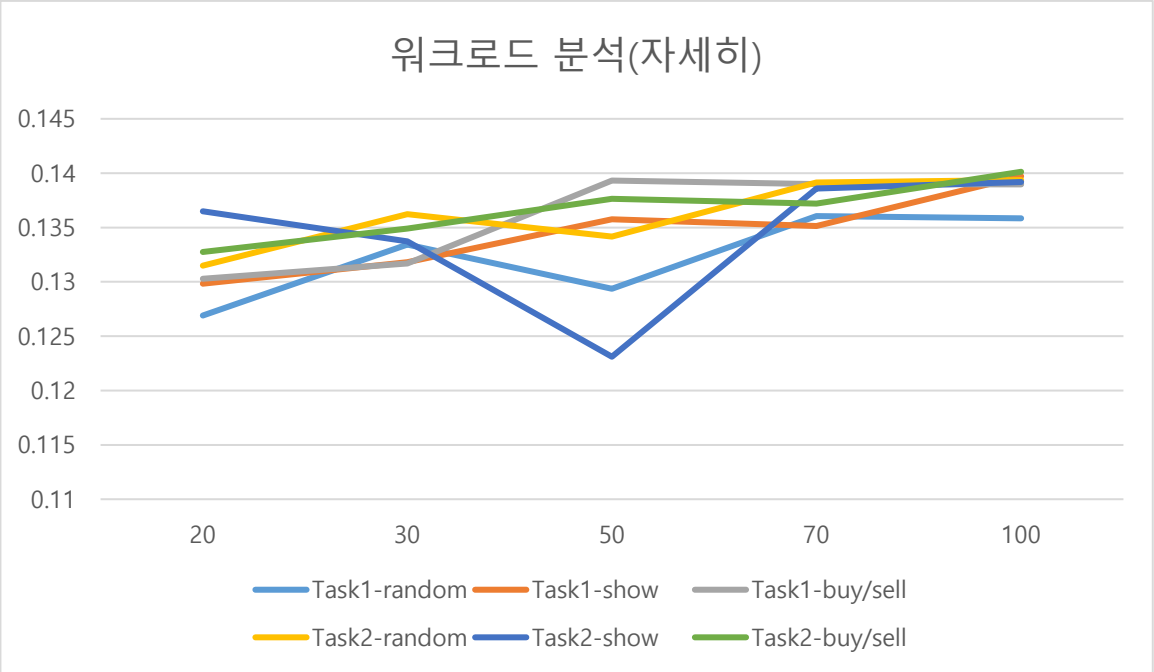
두가지 방법에 대해서 표에서는 클라이언트 수에 따라 소요된 시간을 측정하였고 이를 이용하여 클라이언트 수 / 수요시간을 통해서 동시처리율을 구해서 위의 그래프를 그래주었다. 이를 통해서 확인할 수 있는 것은 두가지 방법 모두 클라이언트수가 증가할수록 동시 처리율이 증가하는 모습을 보였다. 물론 같은 클라이언트 수 더라도 서버의 상황에 따라 소요시간이 다르게 나오는 경우도 있었다. 그리고 클라이언트 수가 증가할 때 어느정도 까지는 동시 처리율이 증가하다가

클라이언트 수가 20을 넘어가기 시작하고 나서는 두가지 방법 모두 비슷한 동시 처리율을 유지하면서 더 이상 증가하지는 못하였다. 이정도의 client 부하에서는 thread를 이용한 방식이 아주 조금 더 높은 동시 처리율을 보여주었다.

다음은 두가지 방식에 대해서 워크로드에 따른 분석을 수행하기 위해서 기존과 동일한 조건에서 명령어의 구성만 다르게 하여 분석을 수행하였다. 그 결과는 아래 표와 같고 show와 buy, sell이 모두 섞인 명령어 구성을 가진 위의 표까지 포함하여 아래 그래프를 그려보았다.

클라이언트 수	*Event-driven Approach (show)*	*Event-driven Approach (buy/sell)*	*Thread-based Approach (show)*	*Thread-based Approach (buy/sell)*
1	48.456밀리초	45.167밀리초	58.013밀리초	36.307밀리초
5	64.617밀리초	63.156밀리초	90.265밀리초	60.516밀리초
10	78.599밀리초	92.873밀리초	78.472밀리초	85.316밀리초
20	154.047밀리초	153.510밀리초	146.535밀리초	150.660밀리초
30	227.617밀리초	227.790밀리초	224.359밀리초	222.392밀리초
50	368.272밀리초	358.888밀리초	406.144밀리초	363.291밀리초
70	517.968밀리초	503.637밀리초	505.131밀리초	510.185밀리초
100	715.739밀리초	719.590밀리초	718.455밀리초	713.639밀리초





두가지 방법에 대해서 random한 명령어가 들어올 때, show명령어만 들어올 때, buy/sell 명령어가 모두 들어올 때에 대해서 분석을 진행하였다. 첫번째 그래프에서는 잘 비교가 되지 않아 두번째 그래프를 그려서 client가 20에서 100일 때의 비교를 좀 더 자세히 진행해보았다. 하지만 사실상 유의미한 차이를 찾을 수 없었다. Show 명령어가 아무래도 readers-writers 문제에서 좀 더 자유롭기 때문에 show 명령어만 수행할 때가 훨씬 높은 동시 처리율을 보여줄 것이라 예상하였지만 실제 측정에서는 모든 경우에서 비슷한 결과를 보여주었다.