System Programming Project 3

담당 교수 : 박성용 교수님

이름 : 김명준

학번 : 20201558

1. 개발 목표

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

간단한 주식서버를 구현한다. 이 프로젝트에서 주식서버에 요구하는 기능은 클라이언트의 접속을 감지하여 연결한 후, 클라이언트가 원하는 명령 (sell, buy, show, exit)에 대해 올바른 동작을 하게끔 하는 것이다.

특히 주식서버에 동시에 여러 클라이언트가 접속하는 것을 관리해주기 위해 event-based와 thread-based 두가지로 모두 구현한 후, 여러 경우에 대해 퍼포먼스를 측정 및 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

Concurrent Programming을 구현하기 위해, Event-based 방식을 택한다. Event-based 방식을 사용하여 여러 클라이언트가 서버에 동시에 접속할 수 있도록 돕는다. 클라이언트가 접속하면, 주식을 Binary Search Tree 형태로 저장해두고 클라이언트가 원하는 구매, 판매, show 등의 명령을 수행하게 하게 만든다.

2. Task 2: Thread-based Approach

Thread-based 방식을 택하여 Concurrent Programming을 한다. 서버-클라이언트의 통신이 있다는 점이나, 클라이언트가 주식에 대해 행하는 작업들에 대해서는 Task 1과 기능적으로는 동일하나, 구현 방식에 대해 차이가 존재한다. 또한 Thread-based에서 발생할 수 있는 race문제를 해결하는 방법으로 프로그램을 작성한다.

3. Task 3: Performance Evaluation

주어진 멀티 클라이언트 프로그램을 활용하여 task1, task2의 각 서버의 작동 시간을 측정하여 얻은 동시처리율을 기반으로 성능을 예상하고 평가해볼 것이다.

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

✓epoll과의 차이점 서술

Event-based 방식은, 프로그래머가 수동적으로 자신만의 private address space를 갖는 로지컬 플로우를 끼워 넣는 방식으로 concurrent programming을 하는 것이다. 여기서, 멀티 클라이언트의 I/O Multiplexing이 필요한데, 이번 프로젝트에서는 select()를 활용하여 이것을 구현했다. 우선 큰 틀에서의 이 방법을 설명하자면, listenfd만을 받는 pool을 설정하여, 서버를 동작시킨 후, Select를 사용하여 추가로 접속이 들어오는 것을 감지한다. 만약 어떤 클라이언트와 Accept로부터 연결이 감지되면 add client를 사용하여 pool에 connfd를 추가하는 방식으로 연결한다. 이후 check client에서 pool을 확인하고, pool에서 연결하고자 하는 클라이언트를 확인하여 통신하며, 명령을 입력 받아 클라이언트 요청에 맞는 기능을 수행해준다.

epoll은 우리가 사용한 select대신에 event based 방식에서 유사하게 사용할 수 있는 함수이다. select는 모든 파일 디스크립터를 매번 새로 전달하고, 때문에 파일 디스크립터가 많아지면, 성능이 저하될 수도 있다. 반면 epoll은 확인해야할 파일 디스크립터를 커널 내부에 유지시키기 때문에, 매번 새로 전달받지 않고, 변경사항만을 전달받아 적용한다. 때문에 파일 디스크립터가 많은 경우에도 성능을 비교적 안정적으로 낼 수 있다. 이러한 작업은 epoll_ctl, epoll_wait등의 함수를 사용하여 수행된다. 요약하자면 모든 파일 디스크립터를 매번 스캔하는 select는 O(n), 변화만 감지하는 epoll은 O(1)의 상수 시간 복잡도를 갖는다고 볼 수 있으며, 때문에 select는 파일 디스크립터 최댓값이 정해져있고, epoll은 제한이 없다. 결론적으로 파일 디스크립터 개수에 따라 적절한 함수를 사용해야 할 것이다.

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

✓Master Thread의 Connection 관리

✓Worker Thread Pool 관리하는 부분에 대해 서술

이번 프로젝트에서는 Master-Worker 방식의 Thread-based 방식을 채택했다. Master-Worker 방식에서는 Master Thread가 클라이언트의 연결 요청을 감지하여 수락하고, 각 요청을 Worker Thread에 전달하는 방식으로 concurrent programming을 한다. 각 thread는 process의 fork와 유사하게 개별적을 수행되기 때문에 동시접속을 구현해낼 수 있는 것이다.

만약 클라이언트가 연결을 요청하면, 마스터 쓰레드가 이를 감지하고 버퍼에 디스크립터를 삽입시킨다. 그럼 워커 쓰레드가 버퍼에 디스크립터가 존재하는 것을 감지하여, 버퍼에서 디스크립터를 삭제하고, 해당 디스크립터와 연결한다.

클라이언트 요청을 처리하는 워커쓰레드의 집합을 워커 쓰레드 풀이라고 한다. 워커스레드는 버퍼에서 작업을 꺼내서 연결하고, 기능을 수행한다. 이러한 쓰레드 풀을 사용하기 위해서는, 특정 정해진 쓰레드를 생성하고, 각 쓰레드가 작업을 대기하도록 한다. 이후 만약 버퍼에 새로운 디스크립터가 삽입되며 작업이 추가되면, 그것을 꺼내어 기능을 수행한다. 모든 요청을 수행한 후, 연결을 종료하고 다음 연결이 오는 것을 대기한다.

- **Task3 (Performance Evaluation)**

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

✓Configuration 변화에 따른 예상 결과 서술

각 서버의 성능을 측정하는 task3는 동시 처리율을 기반으로 성능을 평가한다. concurrent programming이기 때문에, 초당 몇 개의 명령을 동시에 처리하는가를 기반으로 측정하기로 결정했다. 서버에서 적절한 방법으로 멀티 클라이언트 프로그램의 시작과 끝 시간을 측정하여 총 걸린 시간을 측정하고, 총 명령 수를 걸린 시간으로 나누어 동시처리율을 측정할 것이다. 또한 클라이언트 개수를 달리할 뿐만 아니라, show명령만 사용하는 only read방식, buy/sell만을 사용하는 only write방식, 두 명령 모두 사용하는 혼합 방식에 대해서도 비교 분석할 것이다.

configuration에 대해서는, task 2에 대해서 주식의 개수 증가에 따른 변화가

있는지 실험할 것이다. 다른 configuration들은 구현방식에 따라서 다른 변화를 기대하기 힘들다고 판단했고, 주식의 개수 변화에 따라 readers-writers 문제에 따른 블락이 많이 일어날 수 있다고 판단하여 주식 개수를 1으로 설정하고, 또한 readers-writers로 인한 블락을 최대로 발생시키고자 writer 명령만 받도록 조작한 뒤, 해당 경우의 결과와 일반적인 명령의 결과를 실험으로 확인해볼 것이다.

C. 개발 방법

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

기본적으로 주식을 Binary Search Tree에 ID를 기준으로 하여 저장하였다. 때문에 ID, 남은 수량, 가격, 자식노드 등을 갖고 있는 item이라는 struct를 만들어 해당 구조체를 아이템으로 하는 Binary Search Tree를 구현하고자 했다. 이를 위해 기본적으로 makeItem, findItem, insertItem, freeItem, makestockStruct라는 함수를 제작했다. 순서대로 makeItem은 ID, 수량, 가격을 인자로 받아 새로운 item struct를 만든 후 반환해주는 함수이고, findItem은 ID를 인자로 받아 해당 ID를 갖는 주식을 찾아 반환해준다. 존재하지 않을 시, NULL을 반환한다. InsertItem은 첫번째 인자를 root로 하는 트리에, 두번째 인자를 삽입한다. Binary Search Tree이기 때문에, 현재 노드(초기에 root)보다 Id가 작으면 left, 크면 right child를 다시 첫번째 인자로 하여 재귀적으로 함수를 호출하고, 만약 첫번째 인자로 사용할 아이템이 존재하지 않는다면, 해당 자리를 두번째 인자로 받았던 Item으로 채워주는 방식으로 Insert를 구현했다. free역시 재귀적으로 자식들을 먼저 free해주고, 마지막으로 해당 노드를 free해주는 방식으로 구현했다. makestockStruct는 인자로 받은 이름의 파일을 열어 줄단위로 분석 후, 위에서 언급했던 makeItem과 insertItem으로 트리를 만들어주는 함수였다.

task 1을 위해서는 init_pool, add_client, check_client, analysisCommand로 서버-클라이언트 통신을 구현해야했으며 이를위해 pool 구조체를 만들었다. pool 구조체는 maxfd, readset, readyset, nready, maxi, clientfd[], clientrio[]으로 구성되어있다. maxfd와 maxi는 각각 fd의 최대, clientfd의 최대를 의미한다. readset은 읽어야 할 set, readyset은 readset이 변하는 것을 막기 위해 사용하는 fd_set이다. nready는 통신을 기다리고있는 클라이언트의 개수라고 보면 되고,

clientfd[i]와 clientrio[i]는 각각 i번째 클라이언트의 fd와 rio를 의미한다. initpool은 pool구조체의 초기화를 담당한다. 각 값들을 적절한 값으로 초기화해주며, 슬롯을 모두 비어있는 것으로 처리하기 위해 client를 -1로 초기화한다. addclient함수는 새로운 클라이언트를 pool에 추가하는 기능을 담당한다. clientfd를 순회하며 -1으로 설정되어있는 즉, 비어있는 위치를 찾아 해당 인덱스에 새로운 fd를 넣어 추가해준다. checkclient함수는 readyset에서 연결되어있는 디스크립터를 찾아서 해당 디스크립터와 rio를 통해 통신하고, 문자열을 읽어온 후, analysisCommand로 읽어온 문자열을 넘겨준다.

analysisCommand는 task1, task2 에 모두 공통되는 함수로, 클라이언트로부터 받은 문자열을 인자로 받아온 후, 분석하여 적절한 명령을 수행한다. 명령에 따라 아래에 언급될 매매 명령을 위한 함수들을 적절히 불러와서 주식 매매 및 확인을 수행해준다.

클라이언트 매매 명령을 위해 printStock, buyStock, sellStock, writeStocktxt라는 함수를 제작했다.

printStock은 show명령을 위해 제작된 함수이다. 아이템노드 포인터와, char*를 인자로 받아서 주식들을 출력해준다. 인자로 받은 아이템 노드가 Null이라면, 바로 return해주고, 아니라면 주식의 내용을 sprintf로 임시 문자열에 받아온 후, 인자로 받은 문자열 끝에 concat해준다. 이후 아이템의 left, right를 첫번째 인자로하는 printStock을 재귀적으로 불러오며 모든 노드를 순회하며 문자열에 주식정보를 계속해서 병합하는 방식으로 문자열을 만들어준다. 이러한 작업이 끝난 후, 생성된 문자열을 클라이언트에게 rio_wirtten해주어 출력해준다. 이는 이후 txt파일에 주식을 다시 써줄때에도 다시 사용할 수 있다.

buyStock은 주식 구매를 구현한다. 클라이언트의 명령을 분석하여 buyStock에 id와 개수를 넘겨주면, 해당 id를 갖는 주식을 찾은 후, 구매가 가능하면 원하는 숫자만큼 구매(db에서 삭제)하고, 구매 성공이라는 의미로 1을 반환, 개수가 모자라면 0을 반환하는 방식으로 구현했다. 이후 analysisCommand에서 반환받은 값에 따라 success혹은 개수가 충분하지 않다는 문자열을 클라이언트에게 출력해준다.

sellStock은 buyStock과 거의 유사하게 작동한다. 이번 프로젝트에서 sell은 무조건 성공하는 방식으로 input이 주어진다고 가정되었기 때문에, id와 num을 인자로 받아서 db에 증가시켜준 후, 함수를 종료하게 제작했다.

writeStocktxt는 stock.txt에 다시 주식정보를 갱신해주는 함수이다. printStock으로 생성한 문자열을 그대로 파일에 작성해주었다. 이 함수는 클라이언트의 접속 이후, 모든 클라이언트가 종료된 상태가 되면 호출되고, CTRL+C로 SIGINT가 발생할 시에도 signal handler를 통해 writeStocktxt후, exit되게끔 설정하여 서버가 닫힐 때, 호출되게 설정해두었다.

추가로 exit명령이 들어오면 즉시 Close(connfd)등의 작업을 수행하며 ss클라이언트의 접속을 끊게끔 코드를 작성했다.

task 2 에서는 event-based 대신 thread-based이기 때문에 조금 다른 함수와 구조체가 필요하다. Master-worker 방식을 구현하기 위해 sbuf_t라는 구조체를 만들었다. 이 구조체가 하는 역할은 마스터 쓰레드와 워커 쓰레드 사이의 통신을 위한 버퍼 역할이다. 버퍼인 int* buf, 개수를 의미하는 n, 버퍼의 시작과 끝을 의미하는 front, rear와, race condition을 방지하기 위한 slots, item, mutex라는 sem_t 타입의 변수를 선언했다. 이 sbuf를 조작하기 위해서 sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove라는 함수를 제작한다. 우선 이 sbuf가 하는일은 버퍼에 통신하고자 하는 클라이언트를 마스터 쓰레드가 넣어주면, 워커쓰레드가 버퍼에서 꺼내가며 커넥팅하도록 돕는 것이다. 이때, 버퍼가 비어있다면, 워커가 접근하지 않게 하기 위해 item이라는 카운터를 사용하고, 버퍼가 가득 찼다면 마스터가 버퍼를 채우는 동작을 하지 않게하려고 slots이라는 카운터를 사용하는것이다.

sbuf_init과 deinit은 단순하게 초기화와 free를 담당하기 때문에 자세한 내용은 생략한다. mutex는 binary semaphore로 동작하기 때문에 1으로, slots은 버퍼의 빈공간을 의미하기 때문에 n, items은 얼마나 버퍼가 차있는지를 의미하기 때문에 0으로 init에서 초기화 해준다. sbuf_init은 main함수에서 미리 정해둔 max값만큼을 인자로 받아 해당 개수만큼의 버퍼를 생성해준다.

sbuf_insert에서는 마스터 쓰레드가 버퍼에 클라이언트를 넣는 것을 구현한다. P(슬롯)으로 슬롯이 이용 가능할때까지 대기하고, P(뮤텍스) 버퍼에 대한 다른 접근을 막고, 버퍼에 아이템을 넣는다(여기선 connfd값). 그리고 V로 뮤텍스를 풀어주고, V(아이템)으로 아이템이 버퍼에 들어옴을 알려준다.

sbuf_remove는 워커 쓰레드가 버퍼에서 connfd를 확인해갈 때 사용한다. insert와 유사하게, P(아이템)으로 아이템이 존재할때까지 대기하고, P(무텍스)로 다른 접근을 막는다. 이후 버퍼에 있던 아이템을 로컬변수에 저장해두고

V(뮤텍스), V(슬롯)으로 슬롯에 빈공간이 하나 증가함을 알려준 후, 저장해둔 아이템 로컬변수를 반환해준다. 이 반환값으로 받은 fd에 쓰레드가 연결하게 된다.

이와 같은 쓰레드 방식에서는 reader-writer 문제가 발생할 수 있다. 때문에 다수의 reader작업은 허용하되, reader와 writer의 동시작업이나 writer-writer동시작업을 막아야한다. 이 프로젝트에서는 show가 reader, sell, buy가 writer에 해당한다. 이를 위해 ptread_wrlock계열의 함수들을 사용했다.

또한 이러한 과정을 위한 sbuffer의 사이즈와, 총 사용할 쓰레드 개수를 미리 설정해줘야하는데, sbuffer는 1000개, 쓰레드는 20개를 기본적인 설정으로 채택했다.

우선 기존 item struct에 pthread_rwlock_t변수를 새로 추가하고, 아이템 생성과정에서 이를 pthread_rwlock_init을 사용해 초기화해주었다. 물론 아이템 free 과정에서 이도 pthread_rwlock_destory를 사용해 삭제했다. show명령을 수행하는 printStock에서는 재귀적으로 아이템을 확인하는데, 해당 아이템의 정보를 문자열에 적기 전에, pthread_rwlock_rdlock으로 read_lock(shared lock)을 걸어주고, 문자열에 적은 후 pthread_rwlock_unlock으로 해제해주었다. buy, sell의 경우에서는 각각 buyStock, sellStock에서 findItem으로 주식을 찾은 후, 개수를 조작하기 직전과 직후에 각각 ptrhead_rwlock_wrlock과 pthread_rwlock_unlock으로 안전하게 조작할 수 있게끔 함수를 수정했다. 이러한 작업을 하면 동일한 주식에 대해서, read작업은 동시에 여러 개 수행하면서도, read와 write가 동시에 일어나거나, writer가 여러 개 동시에 일어나지 않기 때문에 reader-writer 문제를 해결할 수 있다.

3. 구현 결과

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

task1과 task2에 대해서 각각 event, thread based concurrent server를 구현했다. task1은 pool을 사용하는 I/O multiplexing 방법을 사용했으며, task2는 sbuf를 사용하는 master-worker thread 방법을 사용하고, race를 readers-writers lock을 사용하여 관리해주는 방법을 택했다. 각각 다른 방법으로 서버와 클라이언트를

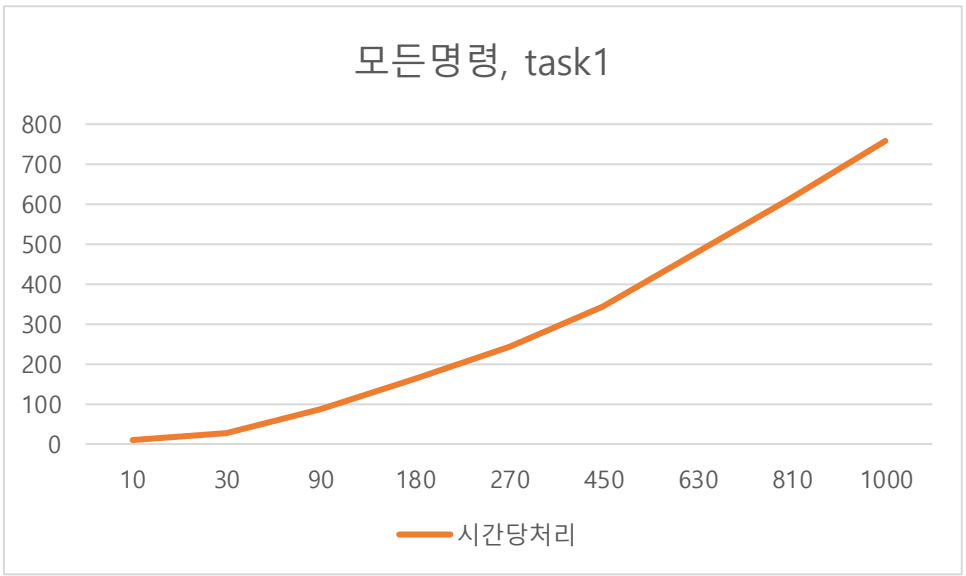
연결한 뒤, 클라이언트가 원하는 명령에 따라 stock.txt로부터 읽어온 값을 변화시켜주고 클라이언트에게 명령의 결과를 반환해주는 방식으로 서버를 제작했다. 그리고 변화한 stock으로 다시 stock.txt를 업데이트해주는 방식으로 프로젝트 코드를 작성했다.

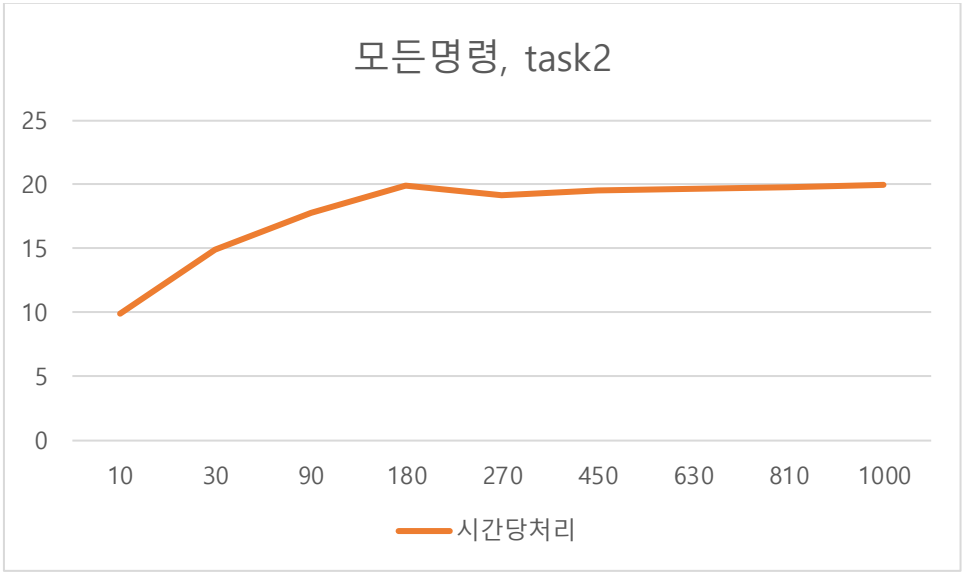
성능 평가 결과 (Task 3)

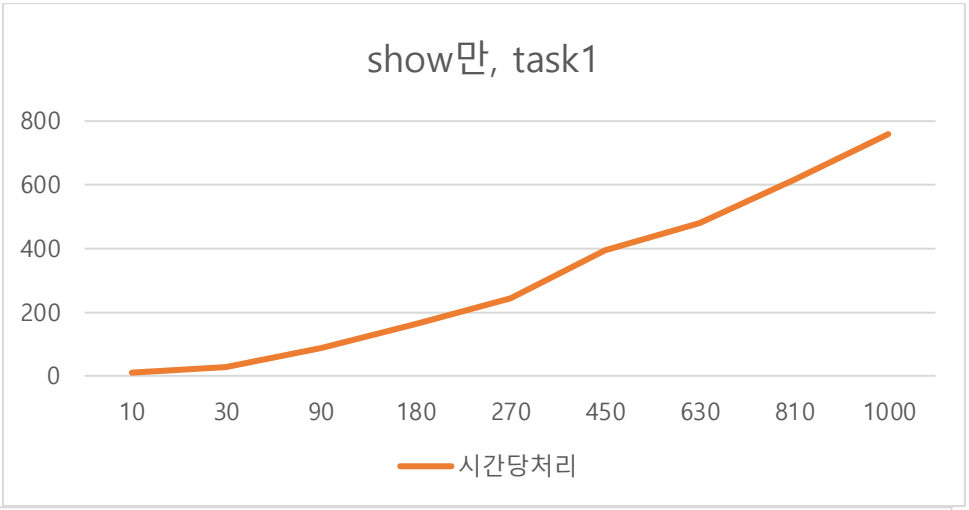
- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

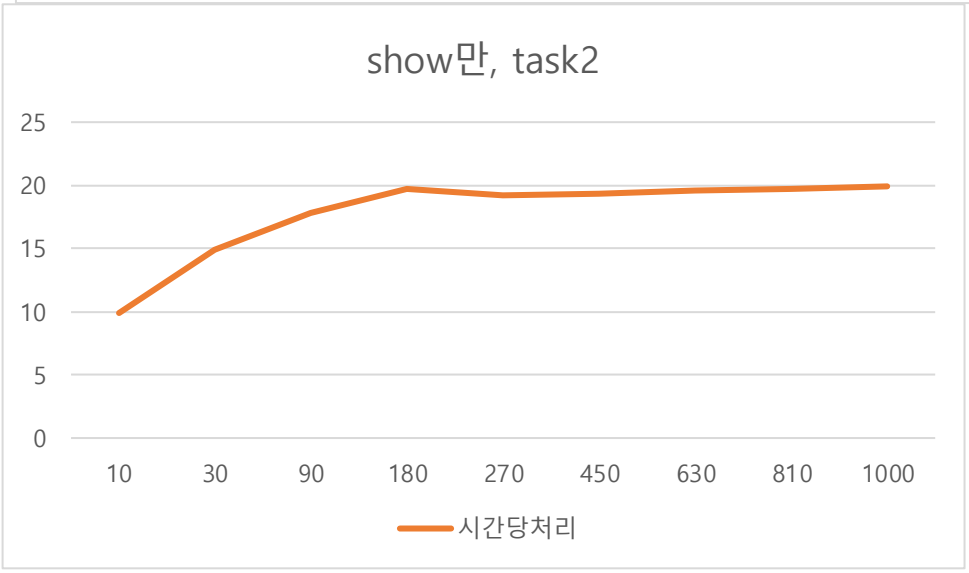
stockserver에서 시간을 측정하는 방식을 선택했다. 처음으로 클라이언트와 접속이 일어나는시점에서 시작 시간을 저장하고, 클라이언트가 연결을 끊을때마다 시간을 측정하여 출력해줬다. 때문에 각 클라이언트가 접속이 종료될때마다 종료시점-시작시점 의 연산을 통해 시간이 측정되며 출력되었고, 마지막 클라이언트가 종료되며 출력한 시간을 총 처리 시간이 되는 것으로 판단하고, 성능을 평가했다.

모든 그래프의 세로축은 시간당 명령 처리율 (명령 개수/걸린 시간)이고, 가로축은 클라이언트 개수이다. 명령은 클라이언트당 10개를 하도록 설정하였으며, 별도의 언급이 없는 그래프는 multiclient.c의 기본 설정대로 실험을 진행했다.




모든명령, task2
25
20
15
10
5
0
10
30
90
180
270
450
630
810
1000
시간당처리


show만, task1
800
600
400
200
0
10
30
90
180
270
450
630
810
1000
시간당처리


show만, task2
25
20
15
10
5
0
10
30
90
180
270
450
630
810
1000
시간당처리

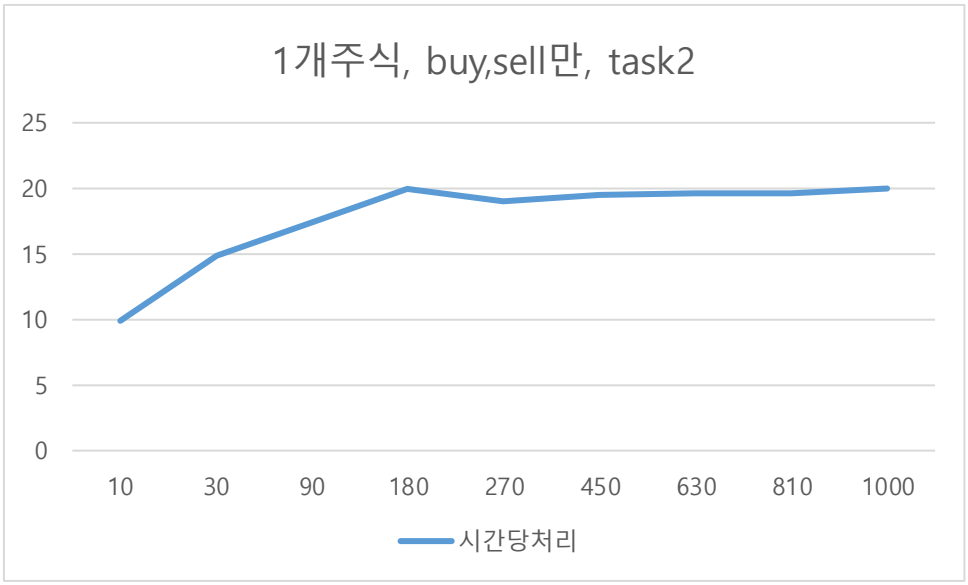
[그래프 1~6 – 내용은 그래프 제목 참고]

위의 6개의 그래프는 각각 그래프 이름과 같은 상황으로 실험을 진행한 결과이다. 명령의 형태를 달리한 세가지 경우에 대해서 모두 거의 유사한 결과가 도출되었다. task1은 계속해서 증가하는 형태의 동시처리율을 보여주었으며, thread기반인 task2는 동시처리율이 20에 수렴하는 형태를 보여줬다. 이러한 결과의 이유를 생각해보자면, task2는 task1과 달리 쓰레드의 개수를 미리 설정하고 시작하는데 이번 프로젝트에서는 그 개수를 20개로 설정해주었기에, 한번에 20개의 쓰레드만이 존재할 수 있어 멀티클라이언트 프로그램이 초당 1개의명령을 전송하는 방식으로 설계되었기에 최대 동시처리율이 20을 넘지 못하는 것으로 판단된다. 반면에 event-based는 하나의 쓰레드에서 작업이 수행되지만, 비동기적으로 멀티 I/O를 사용해 많은 연결을 동시에

처리하기때문에 동시처리율이 계속해서 높아진다. 물론 event-based방식도 시스템 메모리에 한계가 오는 등의 문제에 다다르면 동시처리율이 증가하지 않을 것으로 예상되지만, 이 실험에서는 그정도로 많은 클라이언트를 생성하기에는 제한적이어서 직접 확인은 불가했다.

예상과 다른 결과를 보여준 것은 명령의 형태를 모든명령/show만/buy,sell만으로 바꾸어가며 진행해도 큰 차이가 없음이었다. 이는 readers-writers문제에 대해서 블락을 행하는 것의 영향을 받을 수 있는 task2에서 동시처리율에 문제가 변화를 기대하고 진행한 실험이었다. show들끼리는 블락이 일어나지 않고, writer명령이 동시에 들어오면 블락이 일어나기 때문에 show만 입력되면 동시처리율이 비교적 증가하고, buy,sell만 입력되면 동시처리율이 비교적 감소할 것을 기대했지만, 큰 차이를 확인할 수는 없었다.

이를 확인하기 위해 진행한 추가 실험이 존재한다. 주식 개수를 1개로 제한하고, 오직 buy, sell만을 입력 받으면 모든 클라이언트가 동일한 주식에 대해 명령을 보내기때문에 계속해서 block이 일어나야 하므로 동시처리율이 감소할 것을 기대하고 진행했다.



[그래프 7 – 주식1개, sell/buy만, task2]

하지만 해당 경우에도 위와 동시처리율이 거의 동일하게 나오는 것을 확인할 수 있었다.

이는 크리티컬 섹션에서의 행하는 작업이 단순한 연산 하나이기 때문에 매우

짧은 시간에 block이 풀리고, 클라이언트들의 명령이 정말 block이 걸릴 정도로 매우 짧은 시간 차이로 들어오지 않을 수 있음에 이유가 있을 것이라고 예상했다.

따라서 이번 실험에서는 확인할 수 없었지만, 크리티컬 섹션에서 더욱 복잡한 적업을 수행하거나, 클라이언트 개수가 10만, 100만 등 아주 큰 수로 진행된다면 이를 확인할 수 있을 것으로 기대한다.

그래프로는 확인하기 어려운 자세한 측정값을 위해 표를 아래에 첨부했다. 각 값은 해당 환경에서의 3번의 실험을 평균 낸 값을 작성해주었다.

	task1	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.0455	10.57037	10.116469	11.118196	11.184899	13.08836	13.151073	13.16618	13.19095
		시간당처리	9.9547061	28.38122	88.963847	161.89677	241.396905	343.8169	479.048364	615.2125	758.0958
	task2	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.081668	20.158107	50.438409	90.065544	140.37365	230.6231	320.70402	410.2149	500.31
		시간당처리	9.9189936	14.88235	17.843545	19.985445	19.2343791	19.51235	19.6442814	19.74575	19.98761

[표 1 – 모든명령]

only show	task1	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.07934	10.0832	10.126616	11.074469	11.184749	11.39748	13.157981	13.17349	13.17392
		시간당처리	9.9212845	29.75246	88.874704	162.53601	241.400142	394.824	478.796861	614.8712	759.0757
	task2	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.107416	20.131017	50.538467	91.067095	140.834324	232.4403	321.407031	411.6123	502.0659
		시간당처리	9.8937256	14.902377	17.808217	19.765646	19.1714628	19.35981	19.6013136	19.67871	19.9177

[표 2 – show만]

only sell/bu	task1	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.055213	10.33421	10.843311	11.032181	11.166542	12.3321	13.112366	13.19482	13.19335
		시간당처리	9.9450902	29.029795	83.000478	163.15903	241.793744	364.9014	480.462489	613.8772	757.9578
	task2	CLIENT	10	30	90	180	270	450	630	810	1000
		time	10.103251	20.133447	51.32347	90.123133	141.883312	230.9763	322.41234	411.2218	503.1103
		시간당처리	9.8978042	14.900578	17.535837	19.972674	19.0297221	19.48252	19.540195	19.6974	19.87636

[표 3 – sell/buy만]

주식 한 개만	task2	CLIENT	10	30	90	180	270	450	630	810	1000
onlybuy/sell		time	10.103251	20.119347	51.77347	90.197643	142.431122	231.3424	321.730218	412.1273	501.7794
		시간당처리	9.8978042	14.911021	17.383421	19.956176	18.9565311	19.45169	19.5816235	19.65412	19.92907

[표 4 – 주식1개, sell/buy만]