System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 조이준

학번 : 20161647

1. 개발 목표

일반적으로 단일 프로세스/쓰레드 기반의 주식 서버는 한 번에 하나의 클라이언트와 소통한다. 이러한 주식 서버를 여러 명의 클라이언트와 연결하여 동시에 소통할 수 있는 동시 주식 서버를 설계한다.

동시 주식 서버는 이벤트 기반과 쓰레드 기반의 두 가지 타입으로 설계 및 구현한다.

각각의 주식 서버는 stock.txt 라는 텍스트 파일로부터 주식에 대한 정보를 받고 이를 이진 트리로 구성하여 서버에서 유지한다.

최종적으로 두 가지 타입의 주식 서버의 구현이 완료되면 두 개의 주식 서버의 성능을 예측하고 비교 분석해본다.

2. 개발 범위 및 내용

A. 개발 범위

1. Select

   Select 함수를 구현하면 pending bits가 들어온 connfd 수를 반환하고 연결이 준비된 connfd는 listenfd와 모두 연결하여 pending bits가 들어온 connfd만 요청을 처리하게 된다. 요청이 처리되면 요청 받기를 기다리는 pool.nready를 감소시켜 connfd에 대한 처리를 종료한다.

2. Pthread

   Pthread를 통해서 connfd를 처리하는 쓰레드를 생성한다. 쓰레드 함수는 client로부터 온 show, buy, sell 요청에 대한 처리를 하도록 설계한다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **select**
  - ✓ select 함수로 구현한 부분에 대해서 간략히 설명

    Select 함수는 Listening descriptor와 연결하여 listen descriptor에 들어온 요청의 수를 반환한다. 그리고 들어온 요청을 file descriptor set에 저장하여 요청 하나 하나를 수용한다(accept). 이후에 요청에 대한 처리를 일괄적으로 한

다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

stock.txt에 저장된 주식 정보를 한 줄 씩 읽고 서버의 트리 자료구조에 저장하여 이를 유지하고 사용한다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

Pthread_create를 통해 클라이언트가 서버에게 요청을 보내면 스레드를 생성하여 해당 요청을 처리했다. 이 때, 매번 스레드를 생성하고 종료하는 것을 꽤나 큰 오버 헤드를 요구하기 때문에 스레드 풀을 구현해서 풀에서 스레드를 가져다 쓰고 반납하는 형식으로 구현했다.

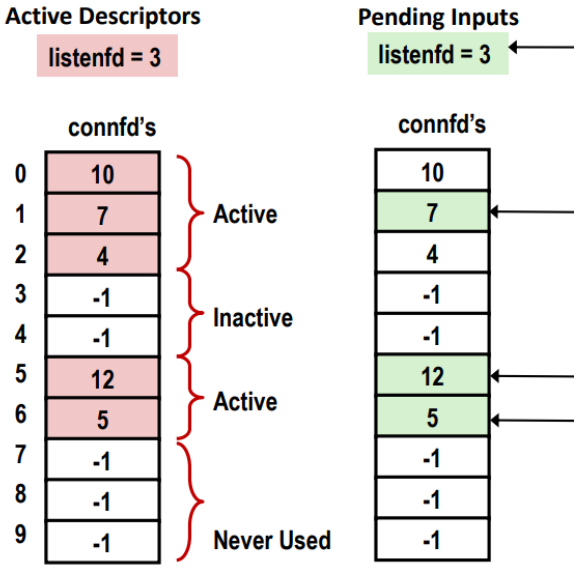
C. 개발 방법

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

Select

Select 함수는 아래 그림과 같은 자료 구조를 필요로 한다. 따라서 pool이라고 하는 구조체를 생성하여 아래의 자료구조를 구현했다. Pool 구조체는 현재 최고로 큰 파일 디스크립터 번호, 현재 가장 큰 배열 인덱스, 현재 펜딩 비트가 대기중이 클라이언트의 수, 클라이언트 배열, 요청에 대한 처리를 저장할 버퍼 등으로 구성된다.

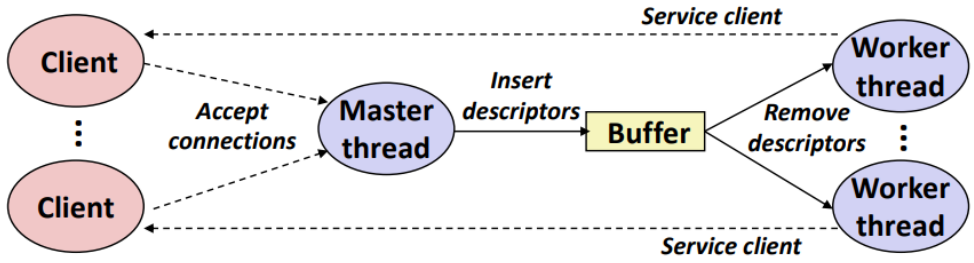
이진 탐색 트리를 만들기 위해 트리의 노드 구조체를 선언했고 트리의 초기화, 트리에 노드 삽입, 트리의 노드에 대한 정보 처리(buy, sell), 트리에서 원하는 노드를 찾는 함수를 추가 구현했다. 노드는 주식의 ID, 잔여 주식의 수, 가격, 주식을 읽는 클라이언트의 수, 왼쪽 자식 포인터, 오른쪽 자식 포인터로 구성했다.



Pthread

스레드 생성 및 종료에 대한 부담을 줄이기 위해 스레드 풀을 이용했다. 스레드 풀은 프로그램 시작 시에 미리 최대 정해진 크기만큼의 스레드를 생성하고 스레드가 필요할 때 마다 해당 풀에서 connfd를 가져다 스레드에 할당하여 사용하는 형식이다. 스레드 풀을 위해 sbuf라는 구조체를 이용했다. Event-based에서는 mutex가 필요하지 않아 사용하지 않았지만 thread-based에서는 필요하기 때문에 tree 노드에 sem_t mutex, w를 추가하였다.

Sbuf는 아래 그림의 buffer에 해당한다.

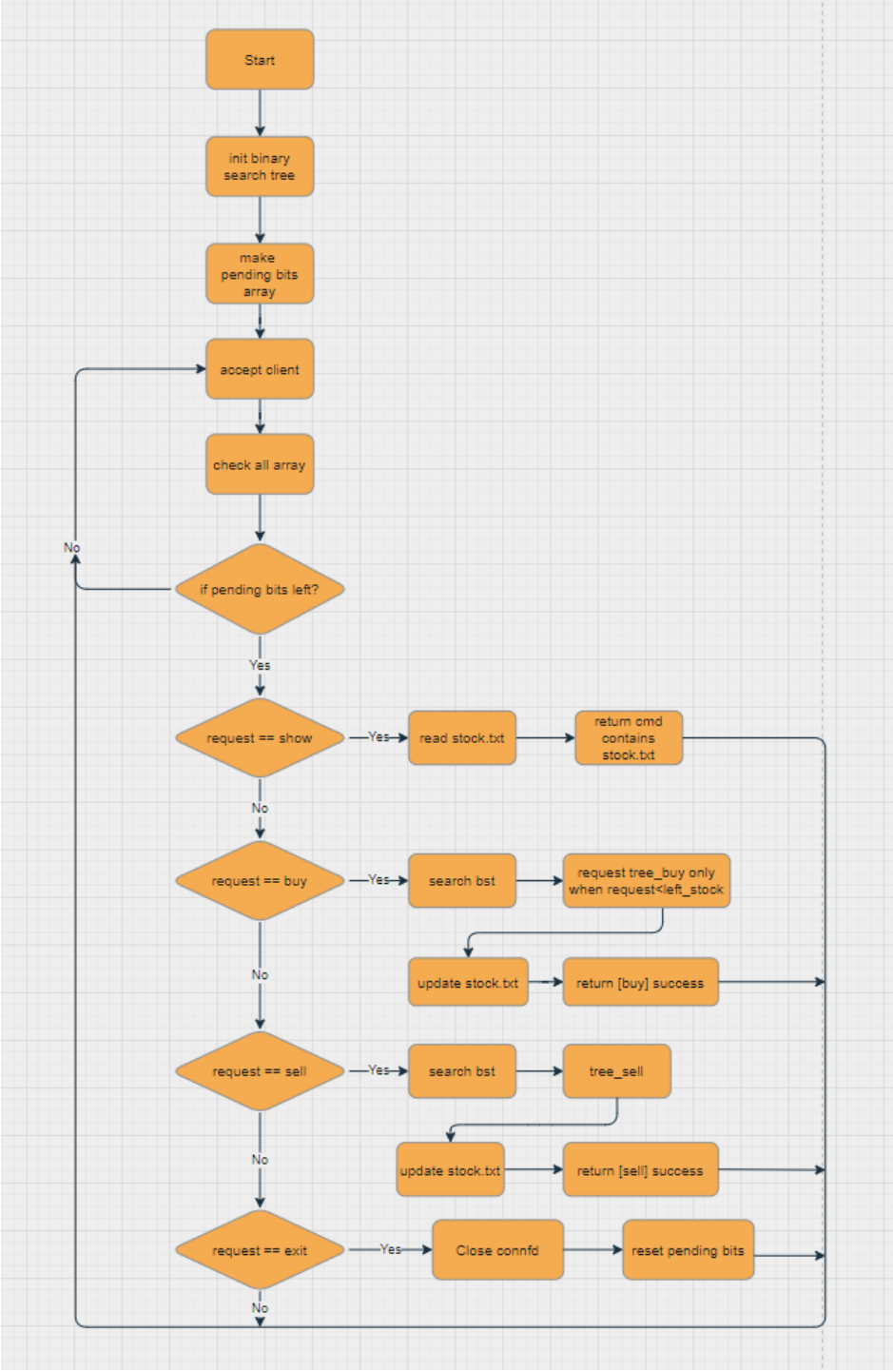


3. 구현 결과

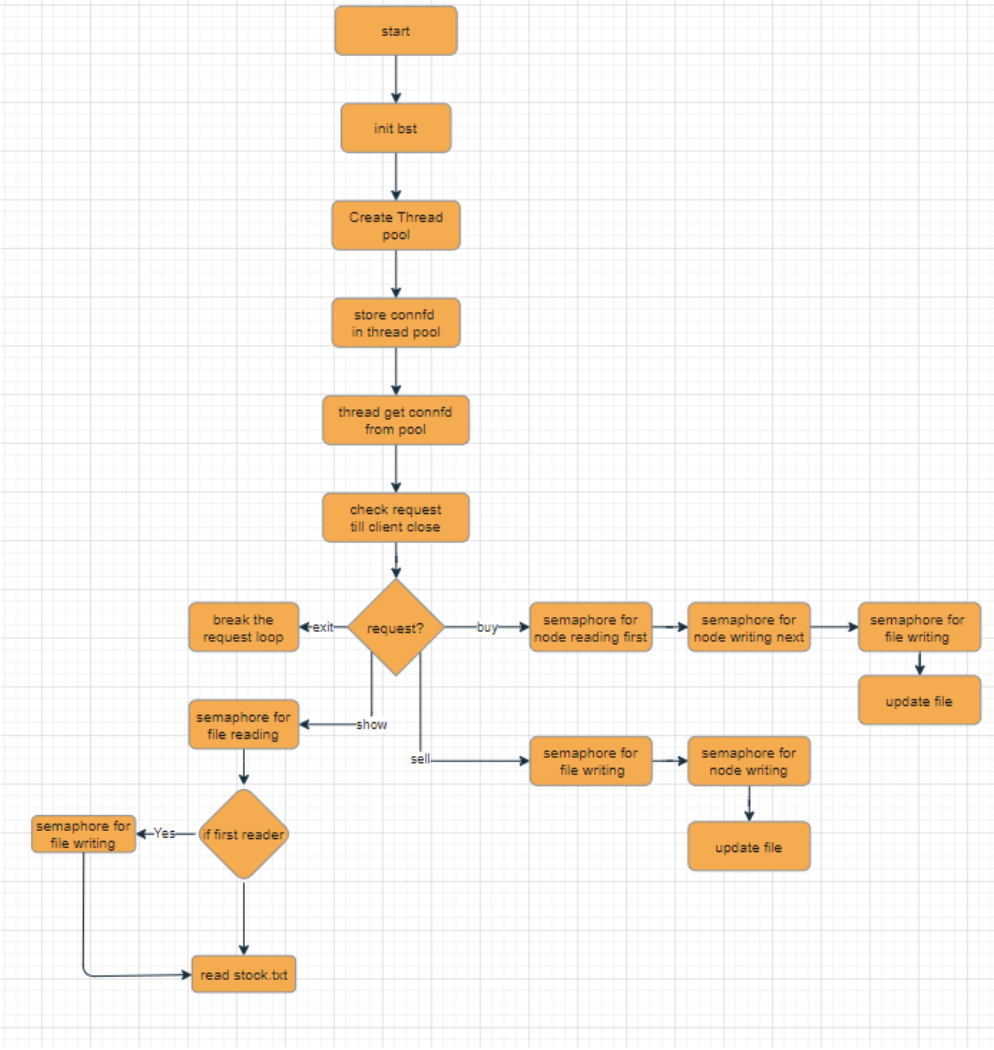
A. Flow Chart

- 2.B.개발 내용에 대한 Flow Chart를 작성.
- (각각의 방법들(select, phtread)의 특성이 잘 드러나게 그리면 됨.)

1. Select



2. Pthread



B. 제작 내용

- II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.
- 개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.

1. Select

Stock.txt에 저장되어 있는 주식 정보를 서버의 메모리에 올려야 하는데 이 때, 이진 탐색 트리를 이용했다. 파일에서 정보를 한 줄 씩 읽어와서 트리의 루트부터 시작하여 서로의 주식 ID 값을 비교해서 ID 값이 root보다 작으면 왼쪽 서브 트리로 ID가 root보다 크면 오른쪽 서브 트리로 저장했다. 이렇게 이진 탐색 트리를 이용하면 buy 또는 sell과 같이 특정한 주식에 대한 정보를 요구할 때 일반적인 탐색 알고리즘보다 더 빠른 접근과 처리가 가능하기

때문에 이진 탐색 트리를 선택했다.

pool은 요청이 들어온 connect descriptor에 대해서 서버의 클라이언트로 추가해주고 풀에서도 배열에 connfd의 값을 저장하는 것으로 해당 클라이언트를 활성화해준다. 반대로 요청에 대한 처리가 종료되면 배열 값을 -1로 변경하여 처리가 종료되어 비활성화 되었음을 알린다. 이 과정은 init_pool, add_client, check_clients의 세 개의 함수를 통해 진행한다. Add_client는 말 그대로 요청이 들어올 때, 파일 디스크립터 배열에 connfd를 저장하여 현재 pending bits가 있음을 알리는 함수이다. Check_clients는 클라이언트로부터 들어온 요청을 처리하는 함수이다.

Buy, Sell에 대한 요청은 각각 tree_buy, tree_sell로 처리하는데 조건(요청한 개수가 현재 남은 개수보다 적은지)을 통과하면 해당 주식 ID 노드의 남은 주식의 수를 변경하는 단순한 함수로 구현했다. 노드를 찾는 함수는 주식 ID를 가지고 트리를 구성했기 때문에 ID를 가지고 트리를 탐색했다.

탐색을 재귀로 할 수 없기 때문에 IDarr이라는 모든 주식 ID를 저장한 배열을 전역으로 선언하여 이용했다. Buy와 Sell을 통해 주식에 대한 정보가 변경되면 매 순간 이를 파일에 업데이트 했는데 이 때, 트리의 모든 노드를 탐색해야 하여 IDarr을 이용했다. Stock.txt 파일을 쓰기용으로 열고 또한 모든 부분을 지우고 새로 쓰는 방식으로 작성했다.

2. **Pthread**

Sbuf는 producer-consumer의 방식으로 설계되어 있는 구조체로 클라이언트의 요청이 들어오면 listen fd와 연결하여 connfd를 생성하고 producer가 item(connfd)을 저장하고 consumer는 이 아이템을 처리 즉, 클라이언트의 요청을 알아차려 처리하도록 한다. 스레드 처리 함수 thread에서는 sbuf에 저장된 connfd를 꺼내서 요청을 처리하도록 한다. 요청이 종료되면 연결을 끊는다.

스레드 처리함수에서는 readers-writers problem을 적용했다. 따라서 tree node에는 mutex 뿐 아니라 w도 추가하여 쓰기에 대한 처리를 추가적으로 진행했다.

주식 정보를 읽으려는 경우가 쓰려는 경우보다 더 많이 일어나기 때문에(주

식을 사고 파는 경우에도 주식 정보를 읽어야 가능하게 설계되었다.) favors readers로 작성했다.

Show 명령어에 경우에 stock.txt file을 읽어오도록 구현했기 때문에 tree의 각 node에 존재하는 mutex와 w뿐 아니라 전역으로 file에 대한 mutex, w를 선언하여 이용했다.

Semaphore를 이용하는 부분을 제외하면 event based에서 트리와 파일에 대한 데이터 처리를 하는 것과 동일하게 작성했다.

3. **문제**

Stockclient와 multiclient가 server로부터 받는 메시지를 Rio_readlineb로 받게 되어 있었는데, readlineb에 경우에 줄바꿈 문자가 도착하면 읽기를 중단한다. 따라서 show 요청이 여러 라인을 보냈을 경우에 한 번에 전부 출력하지 못하기 때문에 해당 부분을 Rio_readnb로 수정하여 작성했다.

Show 명령어에 대해 구현하는 과정에 readnb가 EOF가 들어오는 경우가 아니면 종료되지 않아 프로그램이 해당 위치에 멈춰버리는 경우가 발생했다. 일반적으로 버퍼에 writen을 통해서 보내는 경우에 EOF 입력이 되지 않았고 따라서 show시에 매번 파일을 열어서 읽고 쓰는 작업을 해야 했다.

C. 시험 및 평가 내용

- **select, pthread에 대해서 각각 구현상 차이점과 성능상에 예측되는 부분에 대해서 작성. (ex. select는 ~~한 점에 있어서 pthread보다 좋을 것이다.)**

Select는 pthread와 다르게 스레드를 열고 닫는 컨트롤 작업이 필요하지도 않고 mutex를 통한 컨트롤도 필요하지 않기 때문에 스케쥴링을 하는데 모든 컨트롤이 쓰인다 따라서 프로그램 성능이 pthread보다 좋을 것으로 예상된다. 반면 pthread에서 sbuf 또한 semaphore를 통한 컨트롤이 필요하기 때문에 이 역시 성능의 차이를 만들 것이다.

이외의 트리에 대한 컨트롤은 두 방식 모두 완벽하게 동일하게 작성했기 때문에 서버의 기반으로 인한 차이만이 성능의 차이를 만들 것이다.

- **실제 실험을 통한 결과 분석 (그래프 삽입)**

실험은 주어진 multiclient를 기반으로 본인이 수정한 multiclient로 진행했다.

시간의 단위는 ms이다.

Client의 수	요청 수	Event-based			Thread-based		
		Show	Buy&sell	All	Show	Buy&sell	All
1	10	0.000116	0.000090	0.000121	0.000107	0.000129	0.000086
	15	0.000166	0.000158	0.000140	0.000085	0.000093	0.000124
	20	0.000145	0.000101	0.000081	0.000109	0.000118	0.000130
2	10	0.000140	0.000201	0.000150	0.000195	0.000254	0.000200
	15	0.000221	0.000184	0.000169	0.000179	0.000279	0.000266
	20	0.000187	0.000203	0.000187	0.000193	0.000148	0.000214
3	10	0.000352	0.000254	0.000258	0.000353	0.000222	0.000270
	15	0.000350	0.000354	0.000344	0.000205	0.000219	0.000241
	20	0.000234	0.000292	0.000315	0.000233	0.000279	0.000280
4	10	0.000365	0.000382	0.000272	0.000245	0.000297	0.000309
	15	0.000332	0.000268	0.000471	0.000276	0.000367	0.000287
	20	0.000203	0.000396	0.000406	0.000229	0.000448	0.000351

성능 분석은 c library time.h의 clock 함수와 CLOCK_PER_SEC를 이용하여 클라이언트의 요청을 처리하기 전의 시간부터 요청을 마치가 난 시간의 값을 multiclient에서 계산했다.

실제 성능 분석을 진행 한 결과 예상과는 조금 다른 결과가 발생했다.

1. 요청의 수에 크게 영향을 받지 않는다. 아마도 주식 정보가 상당히 작은 크기이고 클라이언트의 수도 작기 때문인지 요청의 수를 늘렸을 때 무조건적으로 시간이 증가하지는 않는다.

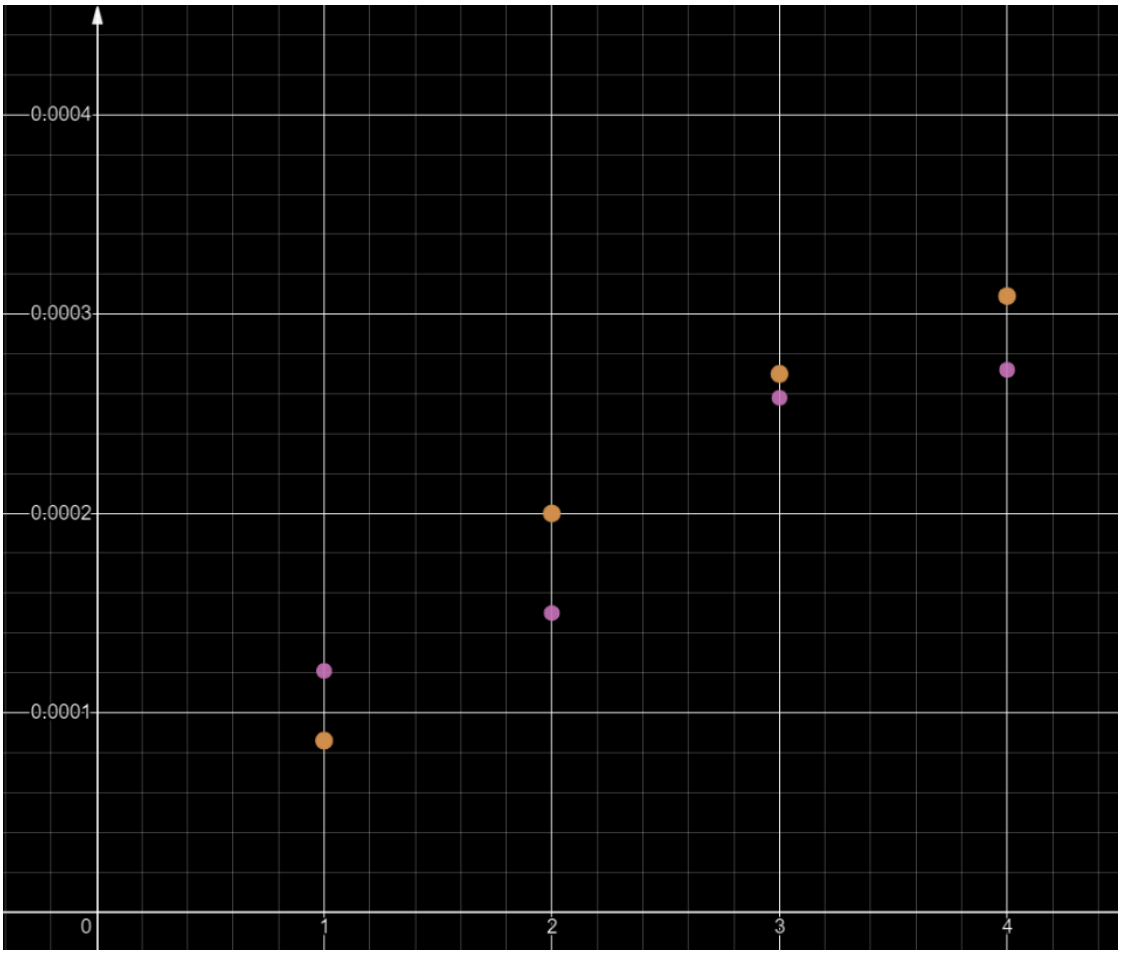
2. 또한 sell, buy 요청은 요청이 올바르지 않은 경우에 파일에 대한 접근이 없기 때문에(stock이 부족하다는 메시지를 전달하는 것으로 종료) 이에 따라서 많은 요청이 있더라도 시간 자체는 훨씬 작을 수 있다는 걸 발견했다.

3. 클라이언트의 수가 늘면 처리 시간이 증가하는 추세이긴 하지만 이 또한 100%는 아니다. 아마도 클라이언트가 1~4까지 늘려도 아주 큰 데이터의 차이가 아니

어서 그런 것 같다. 하지만 증가하는 추세를 보이고 있기 때문에 클라이언트의 수가 매우 커진다면 시간이 크게 차이 날 것으로 예상된다.

4. **스레드 베이스가 이벤트 베이스보다 느리지 않다. 100% 정확한 이유는 예측할 수 없지만 아마도 이벤트 베이스는 멀티 코어 cpu의 장점을 살리지 못하고 스레드 베이스는 멀티 코어의 장점을 살릴 수 있는 것에 차이인 것 같다. 현재 본인이 사용중인 cpu는 6cores 6threads 라서 아마도 이런 차이에 의해서 스레드 베이스가 이벤트 베이스와 비교했을 때 성능의 차이가 나지 않는 것 같다. 오히려 스레드 베이스가 더 좋은 성능을 보이는 경우도 있다.**

5. **그래프는 각 클라이언트의 수와 서버 기반 그리고 요청의 수가 10 모든 요청을 다 받아들일 때를 기준으로 해서 작성했다.**



주황색이 스레드 베이스 서버, 핑크색이 이벤트 베이스 서버이다. 둘다 시간이 증가하는 추세를 보이지만 스레드 베이스 서버가 미세하게 좋은 성능을 보였다. 사실 시간적을 따졌을 때는 더 좋은 성능이라는 표현보다는 동일한 성능이라고 말하는 것이 더 정확할 것 같다.