Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 이효주

학번 : 20200901

1. 개발 목표

이 Project는 동시 Concurrent stock server를 구축한다. 이 서버는 여러 클라이언트의 동시 접속 및 서비스를 처리하기 위한 기능을 제공한다. 주식 서버는 주식 정보를 저장하고 있는 여러 클라이언트와 통신하며, 주식 클라이언트는 서버에 주식 조회, 매수, 매매, 종료의 request가 가능하다.

동시 주식 서버를 설계하기 위해, Event-driven Approach와 Thread-based Approach를 사용한다. Event-driven Approach에서는 select 함수를 활용해 동시 접속 및 서비스를 관리하고, Thread-based Approach에서는 pthread를 사용해 병렬 처리를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Select 함수를 활용해 여러 client들이 동시 주식 서버에 connection을 요청할 수 있으며, server에게 show, buy, sell 등을 요청할 수 있게 된다. 또한 sever는 select를 통해서 connection request가 들어온 file descriptor를 관리하고, pending input이 들어온, 즉 event가 일어난 file descriptor들만 처리함으로써 동시에 들어오는 client들을 관리할 수 있게 된다.

2. Task 2: Thread-based Approach

   Thread를 활용해 여러 client들이 동시 주식 서버에 connection을 요청할 수 있으며, server에게 show, buy, sell 등을 요청할 수 있게 된다. Main thread는 client로부터 connection request가 들어오면 connection을 수락하고, peer thread가 그 client의 요청을 처리함으로써 동시에 들어오는 client들을 관리할 수 있게 된다.

3. Task 3: Performance Evaluation

   Client 실행파일 내에 configuration들을 바꿔가며 두 가지 Task의 elapse time을 측정하여 분석한다. Client의 개수 변화에 따른 변화와 워크로드의 변화에 따른 분석 2가지를 시행한다.

B. 개발 내용

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

- Select 함수를 활용해 여러 client 들이 주식 서버에 동시 접속이 가능하도록 하였다. Select 함수를 호출해서 file descriptor 의 상태를 감시하고, 준비된 file descriptor 를 확인하도록 하였다. 반복문 내에서 각 fd 에 대해 검사하고, 만약 fd 가 listenfd 와 같다면 새로운 클라이언트의 연결 요청이 있음을 의미하므로 accept 함수를 호출해 새로운 연결을 생성하고 watch_set 에 connfde 를 추가한다. 만약 listenfd 가 아닌 경우, client 의 connection 을 의미하므로 fd 가 준비된 상태인지 확인하고, client 로부터 정보를 읽어온다. 위 과정을 반복해 여러 client 들의 요청을 동시에 처리할 수 있도록 하였다. I/O Multiplexing 은 select 함수를 통해 여러 file descriptor 의 event 를 모니터링하고, 발생한 event 에 따라 처리함으로써 multi-client 의 요청을 동시에 처리할 수 있다.

- Select 함수는 모든 file descriptor 에 대해 반복문을 돌며, 확인하기 때문에 비효율적인 I/O 작업이 발생할 수 있고, 오버헤드가 발생할 수 있다. epoll 함수는 이벤트가 발생할 때까지 기다려, 보다 효율적으로 I/O 작업을 수행할 수 있다.

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

- Master thread는 main 함수가 수행하는 부분으로, listenfd를 사용해 client의 connection을 수락하고, accept를 통해 connfd를 얻는다. 이후 sbuf_insert 함수를 사용해 connfd를 worker thread pool의 작업 큐에 삽입하여, 해당 client의 요청을 처리할 worker thread에게 작업을 할당한다. 그리고 master thread는 다시 accept 함수를 호출하며 다음 client의 connection request를 기다리며, 이를 반복한다. 이러한 방식으로 master thread는 client의 connection을 관리하고, 새로운 connection이 establish되면 해당 connection을 worker thread에게 할당하여 처리한다.

- Thread 함수를 통해 worker thread들은 client의 socket file descriptor를 가져오고, 반복문을 통해 해당 client와 통신하며, 수신한 명령어를 처리한다. Rio_readlineb 함수를 사용하여 적절하게 처리하고, exit 함수인 경우 연결을 종료한다. Worker thread pool은 for loop를 통해 미리 생성된 thread들로 구성된다. 그리고 각 worker thread들이 thread 함수를 실행한 이후, kernel에 의해 reaping될 수 있도록 하였고, shared variable에 접근할 때 synchronization을 통해 (semaphore를 사용하여) 오류가 생기지 않도록 하였다.

- **Task3 (Performance Evaluation)**

  - ✓ Client가 많아질 경우, show만 요청할 경우, buy 혹은 sell만 요청할 경우로 나누어 실험하였다. Client의 동시 접속 수에 따라 task1과 task2의 시간 성능이 궁금하였고, 내 예상은 select 함수가 더 빠르게 작동할 것 같다. 왜냐하면 task2는 write의 동작을 수행할 때는 한 번에 하나의 thread만 접근할 수 있도록 하기 때문에 동기화의 문제, 그리고 어느 정도의 overhead가 발생할 수 있을 것 같다는 생각이 들었다.

  - ✓ 그리고 show만 요청할 경우에는 reader만 요청을 하는 경우 이므로, 둘 다 비슷하거

나 task2의 시간이 더 빠르게 작동할 것 같았다. Thread는 읽기의 경우 동시에 여러 개의 thread가 접근 할 수 있으므로 시간이 빠르게 작동할 것 같았는데, 어느 정도의 overhead를 예상해 비슷한 시간 내에 작동할 수 있을 것이라는 생각도 들었다.

✓ Buy, sell만의 명령어를 입력할 경우 task2는 한 번에 하나의 thread만 접근 할 수 있으므로 (공유 변수에) 순차적으로 진행되기 때문에 task1이 더 빠르게 작동할 것이라는 생각이 들었다.

C. 개발 방법

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

Watch_set에 'connfd'를 추가하고, 연결된 client의 file descriptor를 모니터링 한다. 추가된 file descriptor는 'select'함수 호출 시 해당 file descriptor의 event를 감지할 수 있도록 하였다. 이를 통해 새로운 client의 connection을 처리할 수 있다.

새로운 client가 연결되면, 해당 client의 file descriptr를 watch_set에 추가하고, 이후 이 client의 file descriptor도 'select' 함수로 모니터링 하여 event를 감지할 수 있도록 하였다.

새로운 client의 connection은 accept 함수를 사용해 connection을 establish를 하고, 연결된 client의 file descriptor를 반환하고, 반환된 connfd를 watch_set에 추가하는 것이다. 이를 통해 server는 해당 client의 event를 감지하고 client로부터의 data 수신 및 명령어 처리가 가능하다.

Client의 요청에 대해서는 binary search tree를 순회하며, stock information에 접근해 각 show, buy, sell에 대한 함수를 구현하여 처리할 수 있도록 하였다.

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

Stock information을 저장하고 있는 item에 mutex와 semaphore를 사용해 exclusive한 접근을 가능하게 하였다. 그리고 read를 하고 있는 client의 수를 나타내는 변수도 추가하였다. 그리고 main 함수에서 worker thread pool을 생성하고, client의 요청을 수락하도록 하여 establish된 connection의 connfd는 버퍼에 추가하였다.

Thread 함수는 worker thread가 connfd를 가져와, 해당 client의 요청을 처리할 수 있도록 하였고, binary search tree를 순회하여 show, sell, buy에 대한 명령을 처리할 수 있도록 하였다. 또한 file update 시에도 file_mutex를 통해 exclusiveg게 접근할 수 있도록 하였다.

3. 구현 결과

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

  여러 client들이 동시 주식 서버에 connection을 요청할 수 있으며, server에게 show, buy, sell 등을 요청할 수 있게 된다. Server를 실행하면 포트(나의 경우 60064)에서 client의 connection request를 기다리고, client가 server에 connection request를 보내면, server는 해당 request를 accept하고 file descriptor를 생성한다.

  Server는 watch_set에 connfd를 추가해 해당 client의 file descriptor를 모니터링하고, server는 client로부터 받은 input을 읽고, 처리한다. Client가 'exit'를 입력하면, server는 해당 client와의 connection을 종료한다.

  Server는 계속해서 client의 요청을 받고, 처리하기 위해 무한 루프를 돌며 select 함수를 호출한다. 따라서 server는 동시에 여러 client들과 통신하며, 각각의 client의 connection request를 처리할 수 있게 되고, multi-client가 동시에 server에 접속하고 요청을 처리할 수 있다.

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

  여러 client들이 동시 주식 서버에 connection을 요청할 수 있으며, server에게 show, buy, sell 등을 요청할 수 있게 된다. Server를 실행하면 포트(나의 경우 60064)에서 client의 connection request를 기다리고, client가 server에 connection request를 보내면, server는 해당 clinet의 connfd를 버퍼에 추가한다.

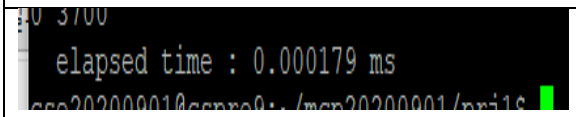
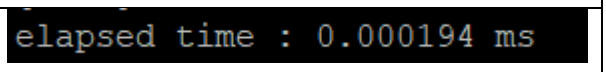
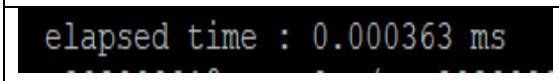
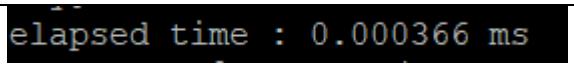
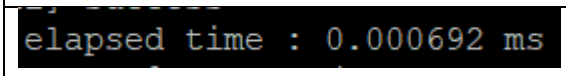
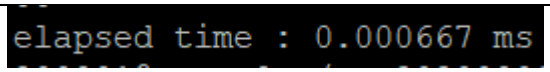
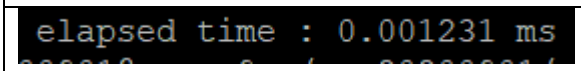
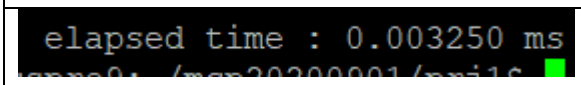
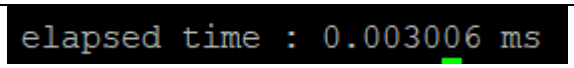
  각 worker thread는 thread 함수에서 작업을 수행하고, 버퍼에서 connfd를 가져와 해당 client의 요청을 처리한다. Sell, buy, show에 대한 명령어는 binary search tree를 순회하며 해당 주식 항목에 대한 작업을 처리하고, 동시에 여러 client가 접근해도 exclusive하게 오류 없이 안전하게 실행된다. 그리고 sell과 buy는 한 번에 하나의 client만 접근할 수 있도록 보호된다.

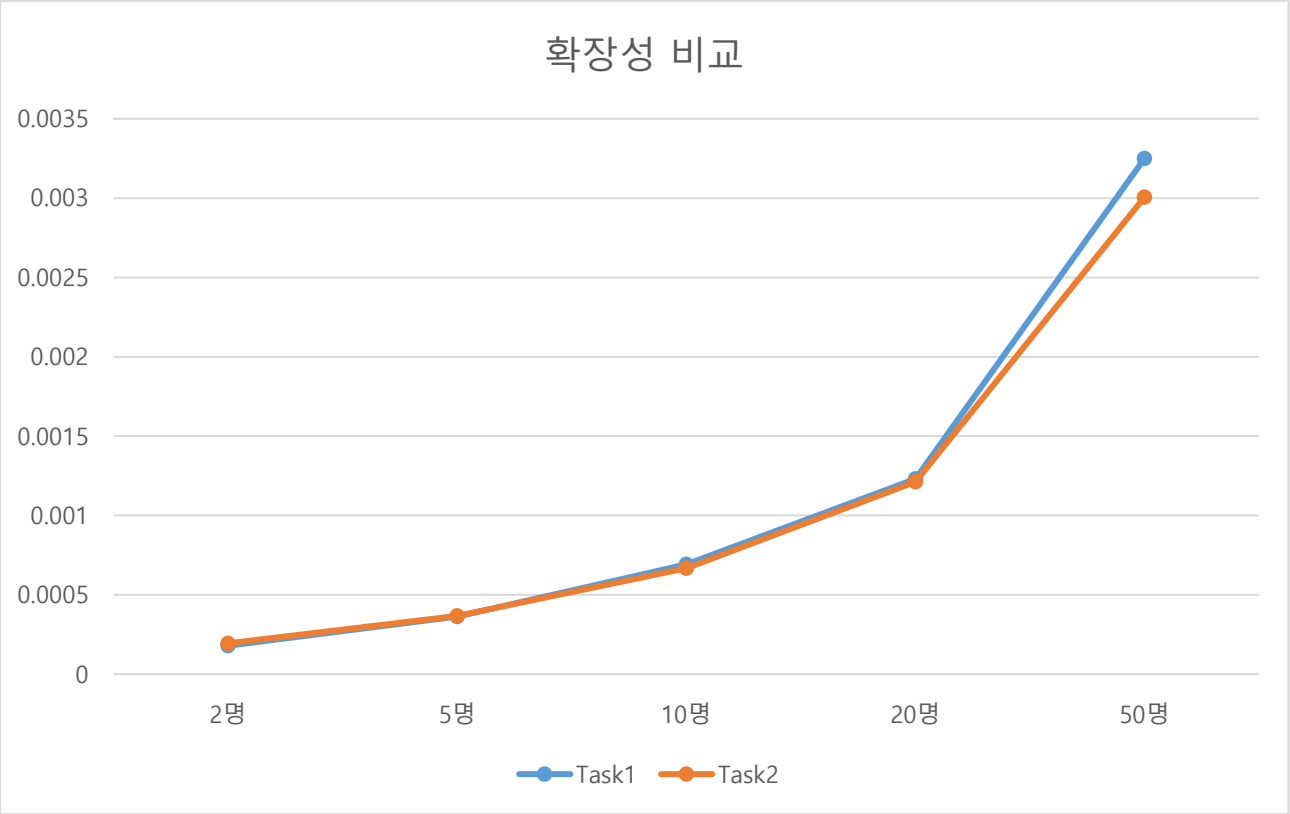
  File update 시에도 exclusive하게 접근하여, 여러 thread가 동시에 file에 접근하여 update하지 않도록 보호된다. 따라서 server는 동시에 여러 client들과 통신하며, 각각의 client의 connection request를 처리할 수 있게 되고, multi-client가 동시에 server에 접속하고 요청을 처리할 수 있음과 동시에 적절한 동기화를 통해 exclusive하게 안전하게 수행된다.

4. 성능 평가 결과 (Task 3)

- Order per client = 10, stock num = 5, buy sell max = 5
- Server: cspro, client: cspro9, port_num: 60064, report time:2023.05.29 18:00 ~ 18:20

- Task1	- Task2

- Client 2명	- Client 2명
	
- Client 5명	- Client 5명
	
- Client 10명	- Client 10명
	
- Client 20명	- Client 20명
	
- Client 50명	- Client 50명
	



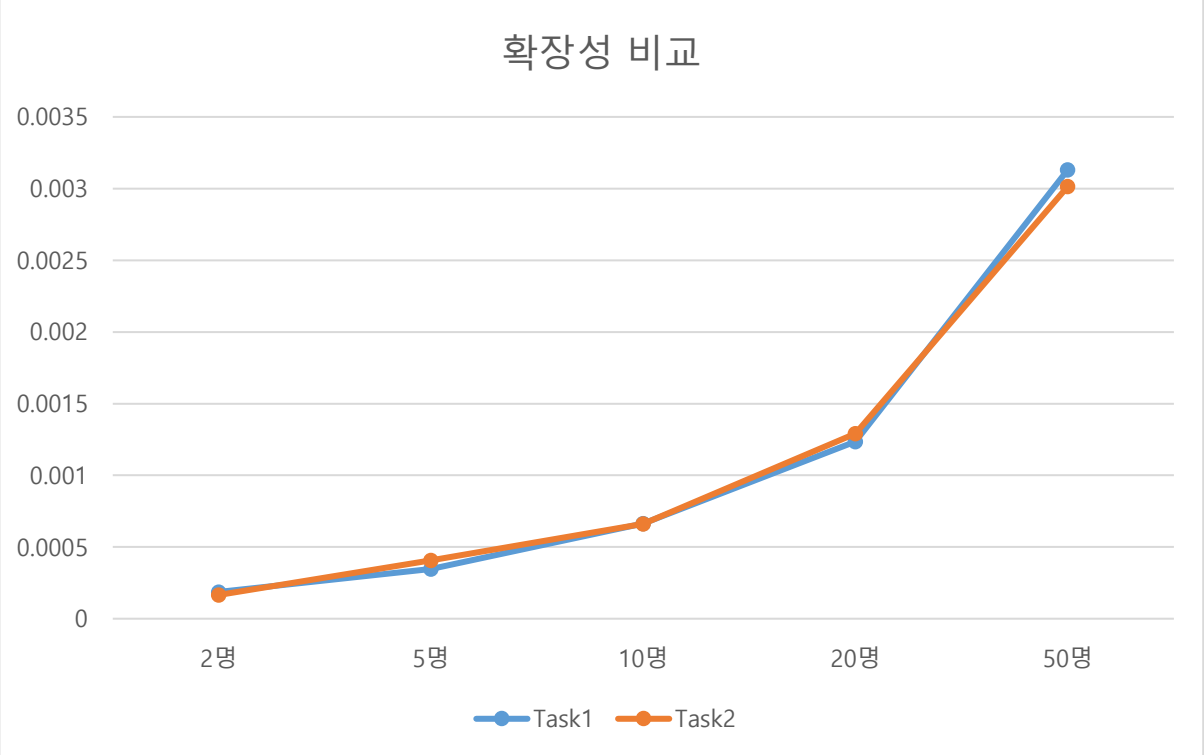
5개 주식 종목에 대해, client의 개수를 늘려가며 비교하였고, client 당 10번의 request를 요청하도록 하였으며 buy, sell, show가 random하게 나왔다.

동시처리율을 보면 client의 수가 많아지면서 Task2가 조금 더 빠른 시간 내에 작동되었는데, buy, sell, show가 random하게 나왔기 때문에 (어느 한 task가 전부 show가 나왔을 수도 있음) 정확하다고는 할 수 없지만, 내가 얻은 결과에서는 Task1보다는 Task2가 더 빠른 시간 내에 동작할 수 있음을 확인할 수 있었다.

- Order per client = 10, stock num = 5, buy sell max = 5
- Server: cspro, client: cspro9, port_num: 60064, report time:2023.05.30 14:58 ~ 15:20
- Show 명령어만 처리하는 경우

- **Task1**	- **Task2**
- Client 2명	- Client 2명
`elapsed time : 0.000187 ms`	`elapsed time : 0.000166 ms`

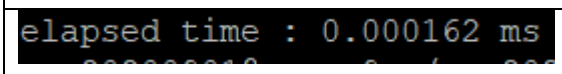
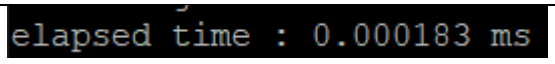
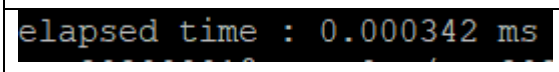
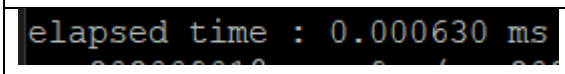
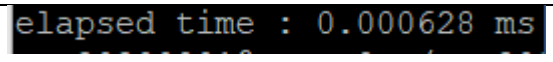
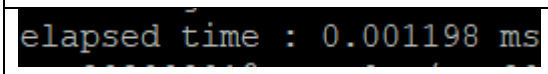
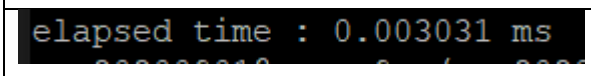
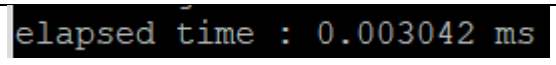
- Client 5명	- Client 5명
elapsed time : 0.000346 ms	elapsed time : 0.000407 ms
- Client 10명	- Client 10명
elapsed time : 0.000663 ms	elapsed time : 0.000661 ms
- Client 20명	- Client 20명
elapsed time : 0.001236 ms	elapsed time : 0.001291 ms
- Client 50명	- Client 50명
elapsed time : 0.003131 ms	elapsed time : 0.003015 ms

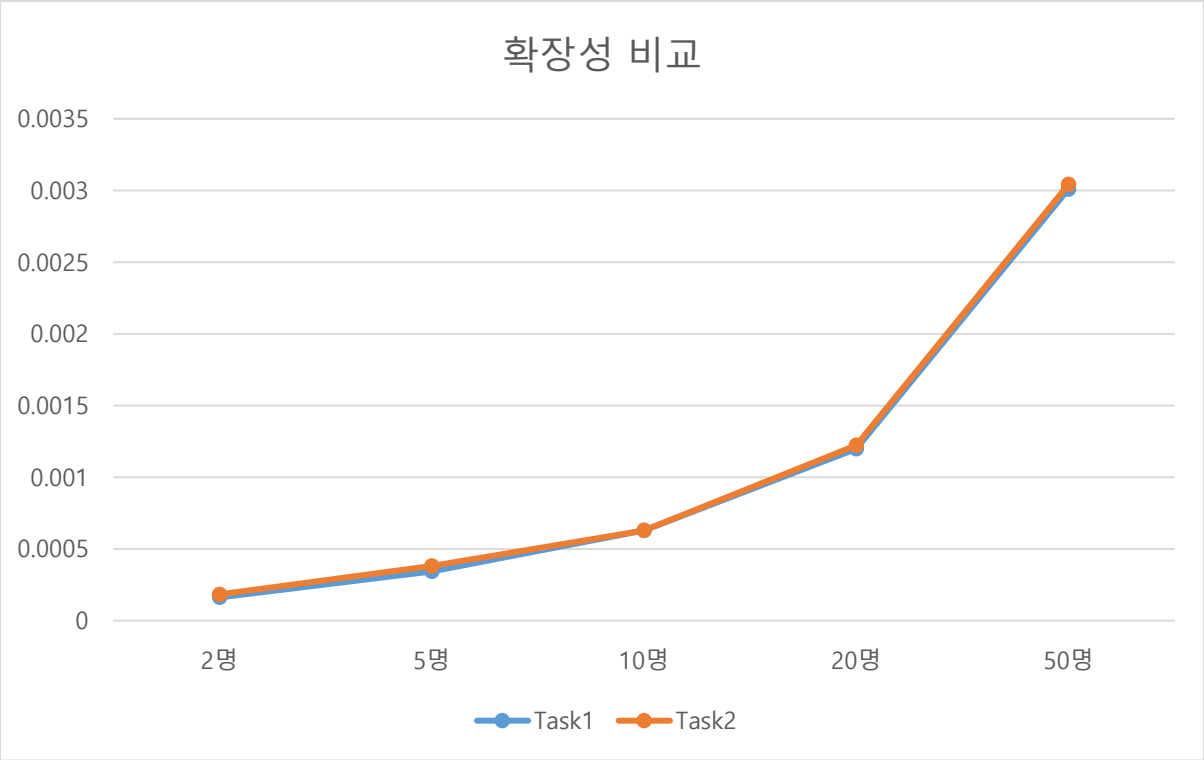


모든 client가 show 명령어만 요청한 경우, 시간의 차이가 별로 나지 않지만 client 수가 많을 때는 task1이 시간이 더 오래 걸렸다. Show의 경우 task2는 한 번에 여러 thread들이 read, 즉 하나

의 주식 종목에 동시에 접근할 수 있으므로 task2가 조금 더 빠르지만, thread의 control overhead 나 synchronization overhead 등의 이유로 시간 상 별로 차이가 나지 않는 것 같다.

- Order per client = 10, stock num = 5, buy sell max = 5
- Server: cspro, client: cspro9, port_num: 60064, report time:2023.05.30 15:48 ~ 16:20
- Buy와 sell 명령어만 처리하는 경우

- Task1	- Task2
- Client 2명	- Client 2명
	
- Client 5명	- Client 5명
	
- Client 10명	- Client 10명
	
- Client 20명	- Client 20명
	
- Client 50명	- Client 50명
	



모든 client가 buy, sell 명령어만 요청한 경우, 시간이 비슷하기는 하지만 client가 많이 있을 때 task1이 조금 더 빨랐다. Buy, sell의 경우 writer로서 task2는 한 번에 하나의 thread만이 주식 종목에 접근할 수 있으므로, 순차적으로 진행되기 때문에 synchronization과 관련한 overhead로 인해 task1이 조금 더 빠른 성능을 보인 것 같다.

Task1과 task2에 대해 시간 차이가 많이 날 것이라고 생각했는데, 시간 차이가 거의 비슷했고, 동기화를 잘 적용했느냐에 따라 task2의 시간 성능이 달려있을 것이라는 생각이 들었다. 그리고 네트워크 상태나, 컴퓨터 조건 등에 따라 시간 성능이 달라질 수 있을 것 같다라는 생각 또한 들었다.