System Programming Project 3

담당 교수: 김영재

이름: Jumagul Alua

학번: 20231632

1. 개발 목표

본 프로젝트의 개발 목표는 몇개의 클라이언트가 동시에 주식을 사고, 팔고 하는 명령을 서버에 요청할 때 안정적이고 효율적으로 이를 처리하는 주식 서버를 구현하는 것이다. 서버는 주식 데이터의 일관성을 유지하면서 신속하게 요청에 대응해야 하며, 여러 클라이언트의 요청을 동시에 처리하게끔 만들어야됐다. 이를 위해 이벤트 기반 구현 방법과 스레드 기반 구현 방법을 모두 구현하여 성능과 자원 효율성을 비교하는 것이 프로젝트의 목표였다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task1: Event-driven Approach

select() 시스템 호출을 사용하여 단일 스레드 내에서 다수 클라이언트 소켓의 I/O 상태를 감시하면서 요청을 처리하는 방식을 구현하였다. 이 방식은 한정된 스레드 자원으로 다중 접속을 효율적으로 관리할 수 있으나, select()의 파일 디스크립터 수 제한과 복사 비용으로 인해 확장성에 한계가 존재한다.

2. Task 2: Thread-based Approach

이 방식에서는 Master Thread가 클라이언트의 접속 요청을 수락하고, 각각의 연결에 대해 독립적인 Worker Thread를 생성하여 처리한다. 이를 통해 요청 처리가 병렬로 수행되어 대기 시간을 줄일 수 있지만, 다수의 스레드 생성과 관리로 인한 오버헤드가 발생할 수 있다. Worker Thread Pool을 별도로 관리하지는 않았으나 각 요청마다 스레드를 생성해 병렬성을 확보하였다.

3. Task 3: Performance Evaluation

처리 지연 시간, 처리량, 그리고 서버 자원 사용률을 주요 성능 지표로 삼아 실험 환경에서 각각의 서버 구성 변화에 따른 성능 변화를 관찰하였다. 이 평가를 통해 각 방식이 가지는 장단점을 분석하고 다양한 명령어들을 수행하면서 각자 걸린 시간을 측정해보았다.

B. 개발 내용

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

Task 1에서 구현한 이벤트 기반 서버는 select() 함수를 활용하여 다중 클라이언트의 I/O를 동시에 감시하였다. select()는 사용자가 지정한 파일 디스크립터 집합을 커널에 복사해 이벤트 발생 여부를 감시하는 방식이다. 이로 인해 많은 수의 소켓을 처리할 때 복사 비용이 증가하고, 파일 디스크립터 수가 FD_SETSIZE에 의해 제한된다는 단점이 존재한다. epoll과 비교 시 epoll은 커널

내에서 직접 이벤트 감시를 수행하며, 이벤트 발생 시점에만 반환되어 성능과 확장성 면에서 우수하다. 하지만 본 프로젝트에서는 select()를 선택해 구현하였다. 또한, 주식 데이터에 대한 점근시 동기화를 위해 세마포어를 사용하였다. 세마포어를 통한 락은 경쟁 조건을 방지하고 데이터 무결성을 보장하였다.

-Task2 (Thread-based Approach with pthread)

Task 2에서는 pthread 라이브러리를 활용하여 다중 스레드 서버를 구현하였다. Master Thread는 클라이언트 접속 요청을 수락한 후, 각각의 연결마다 새로운 Worker Thread를 생성하여 클라이언트의 요청을 독립적으로 처리하도록 하였다. 각 Worker Thread는 클라이언트로부터 명령을 읽고 처리하는 식으로 구현되어 있다. Worker Thread Pool은 별도로 구현하지 않았으나, pthread_create와 pthread_detach를 이용하여 생성된 스레드가 종료 후 자원을 자동 해제하도록 관리하였다.

-Task3 (Performance Evaluation)

Task 3에서는 성능 평가 지표를 정의하고 측정하였다. 주요 지표는 처리 지연 시간으로, 클라이언트 요청 수신 시점부터 응답 완료 시점까지의 소요 시간을 의미한다. 처리량은 일정 시간 내 처리한 총 요청 수로 정의하였다. 이 지표들은 서버가 다수 클라이언트를 동시에 처리할 때의 효율성과 안정성을 평가하는 데 중요한 척도가 되었다. 성능 측정은 클라이언트 시뮬레이션 툴을 통해 다양한 부하 조건에서 진행하였다. 클라이언트 수를 증가시키거나, 스레드 개수를 조절하는 등 설정을 변화시키며 각 지표의 변화를 관찰하였다. 예를 들어 스레드 수가 부족하면 처리 지연이 증가하고, 반대로 너무 많으면 컨텍스트 스위칭 비용으로 인해 성능 저하가 발생할 것으로 예상하였다.

C. 개발 방법

Task 1 구현을 위해, 소켓 연결 관리를 위한 pool 구조체를 정의하였다. 이 구조체는 클라이언트 소켓 파일 디스크립터 배열과 fd_set을 포함하여 select() 호출 시 감시할 파일 디스크립터 집합을 관리한다. 클라이언트 연결이 accept 되면 pool에 추가하고, select()를 이용해 준비된 소켓을 검사하여 해당 소켓에서 데이터를 읽어 명령을 처리하였다. 명령 처리 함수는 클라이언트로부터 받은 문자열을 파싱해 주식 매매 또는 조회를 수행하며, 결과를 다시 클라이언트에게 전송하였다.

Task 2에서는 pthread 기반 멀티스레드 환경을 구축하였다. 메인 스레드는 listen 소켓에서 Accept를 호출하여 클라이언트 연결을 수락하면, 해당 연결 소켓 디스크립터를 동적 할당 메모리에 저장하여 Worker Thread에 인자로 넘긴다. Worker Thread는 독립적으로 클라이언트 요청을 처리하고, 명령 수행 시 데이터 경쟁을 막기 위해 세마포어를 사용하여 주식 데이터에 대한 임계영역을 보호하였다. 주식 데이터는 이진 탐색 트리로 관리되며, 삽입, 탐색, 수정 연산이 트리 구조를 통해 수행된다. 클라이언트와의 통신은 rio 라이브러리를 사용해 신뢰성 있는 데이터 입출력을 보장하였다.

주식 데이터 초기화는 서버 시작 시 파일로부터 데이터를 읽어 트리 구조로 구성하였고, 서버 종료나 클라이언트 연결 종료 시 최신 데이터를 파일로 저장하였다. 이를 통해 데이터 일관성을 유지하였다.

3. 구현 결과

본 프로젝트를 통해 select() 기반 이벤트 드리븐 서버와 pthread 기반 멀티스레드 서버를 모두 구현하였다. 이벤트 드리븐 서버는 하나의 스레드가 다수 클라이언트의 I/O 이벤트를 감시하여 처리하는 구조로, 스레드 생성과 관리 오버헤드가 없어 경량화되어 있다. 그러나 select() 함수의 파일 디스크립터 제한과 I/O 복사 비용으로 인해 클라이언트 수가 많아질 경우 성능 저하가 관찰되었다.

멀티스레드 서버는 각 클라이언트 요청을 별도의 스레드가 처리하여 높은 병렬성을 구현하였다. 이로 인해 요청에 대한 응답 지연 시간이 단축되었고, 다수 클라이언트 환경에서 처리량이 크게 증가하였다. 반면 스레드 생성 및 관리 비용과 동기화로 인한 경합(lock contention)이 발생하여 자원 사용률이 상대적으로 높았다.

성능 평가 결과, 이벤트 드리븐 방식은 적은 자원으로 경량화된 서비스가 가능하나 확장성 측면에서 제한이 있으며, 멀티스레드 방식은 확장성이 우수하나 자원 소모가 크다는 결론에 도달하였다. 이를 바탕으로 실환경에서는 epoll 기반의 이벤트 드리븐 방식과 효율적인 스레드 풀 관리가 결합된 하이브리드 구조가 바람직하다고 판단했다.

Taks3성능 평가에서는 처리율에 대해 잘못 이해해, client당 elapsed time을 바탕으로 그래픽스를 그려보았다. 그러므로, 처리율을 가져서 두개의 방식을 그림으로 직관적으로 보고 분석해보면 좋을 것이다. 또한, 이번 프로젝트에서는 select()를 활용한 이벤트 드리븐 방식을 구현하는 것이 목표였지만, epoll을 이용해서 성능을 비교해볼 만하다고 본다. 앞에 언급한 단점 부분을 보완하고, 주식 데이터베이스에 대한 효율적 동기화 및 복제 방안을 도입하여 안정성과 확장성을 개선하는 방법으로 구현해보면 좋을 것 같다.

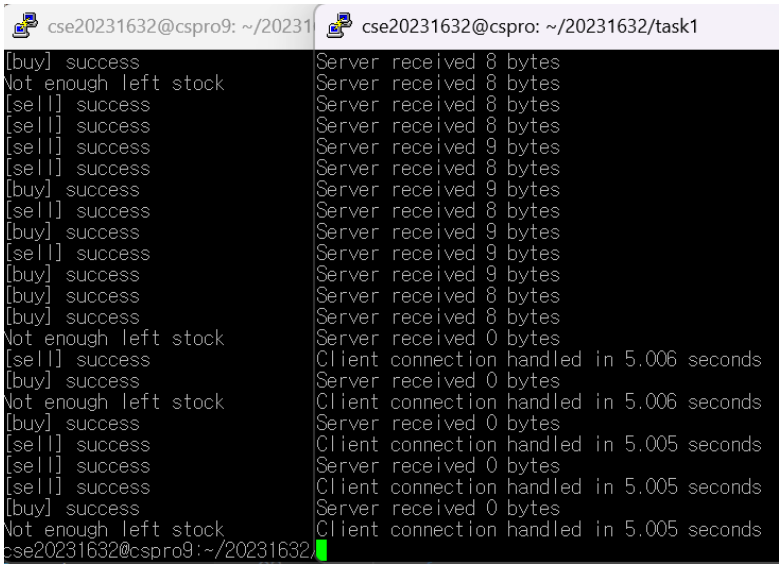
4. 성능 평가 결과 (Task 3)

- 3 개의 방식으로 multicllient.c 에서 option 과 ORDER_PER_CLIENT 의 값을 바꾸면서 두 approach 의 클라인트당 모든 주문의 총 걸린 시간을 측정해보았다.
- Stockserver.c 에서는 gettimeofday()를 추가해서 측정하였다.
- 처리율 = (클라이언트 개수) * (ORDER_PER_CLIENT) / (elapsed time)

1. 모든 client가 buy 또는 sell을 요청하는 경우

- ORDER_PER_CLIENT 10
- usleep(500000)
- int option = (rand()%2)+1

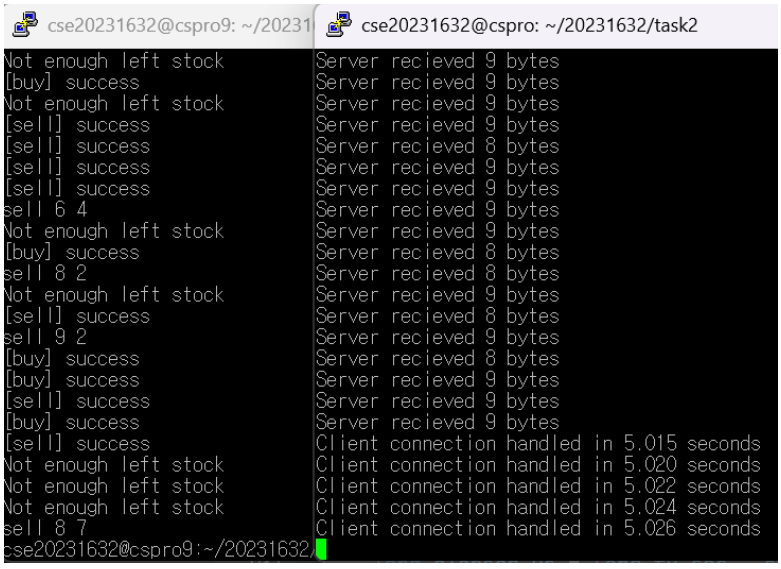
a) Event-based approach



측정 중 예시 하나

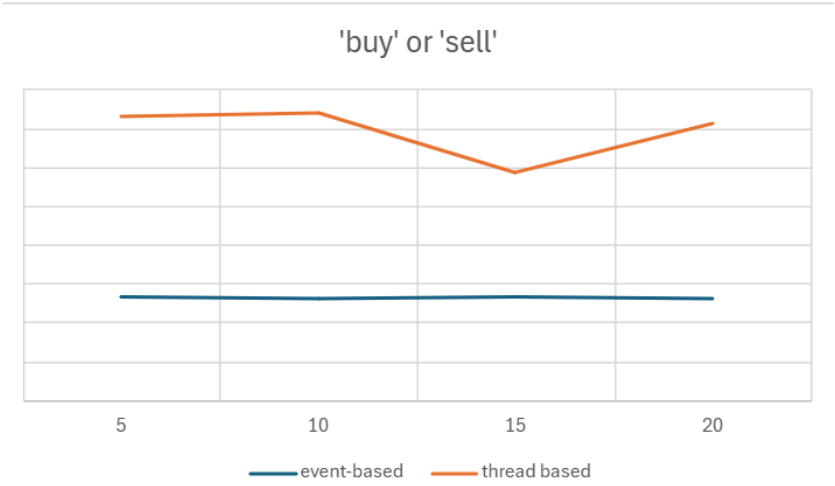
Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.0058	5.0064	5.0064	5.0064	5.0063	9,9874
10	5.0061	5.0066	5.0058	5.0064	5.0062	19,9752
15	5.0064	5.0064	5.0065	5.0065	5.0065	29,9611
20	5.0058	5.0060	5.0064	5.0055	5.0059	39,9529

b) Thread-based approach



측정 중 예시 하나

Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.1256	5.0282	5.0266	5.0314	5.0530	9,8951
10	5.0369	5.0318	5.0218	5.1259	5.0541	19,7859
15	5.0459	5.0318	5.0397	5.0373	5.0387	29,7696
20	5.0996	5.0343	5.0365	5.0340	5.0511	39,5953

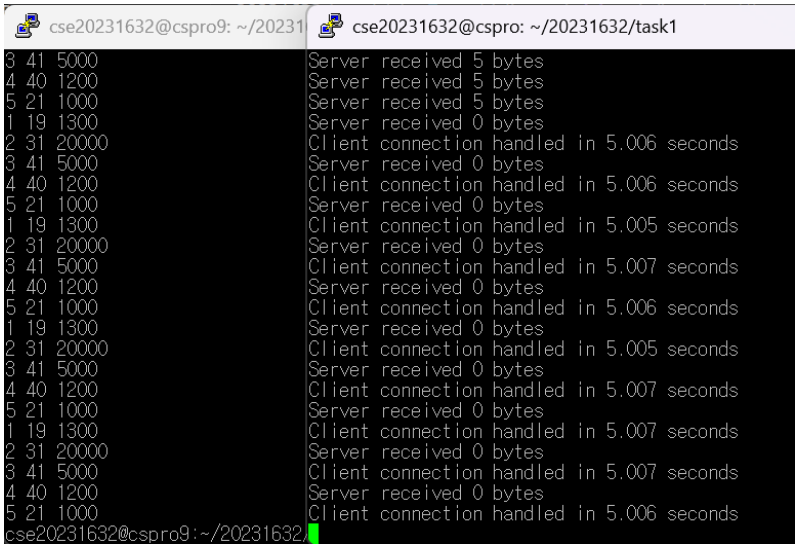


가로: num of client, 세로: 시간(s)

2. 모든 client가 show만 요청하는 경우

- ORDER_PER_CLIENT 10
- usleep(500000)
- int option = 0

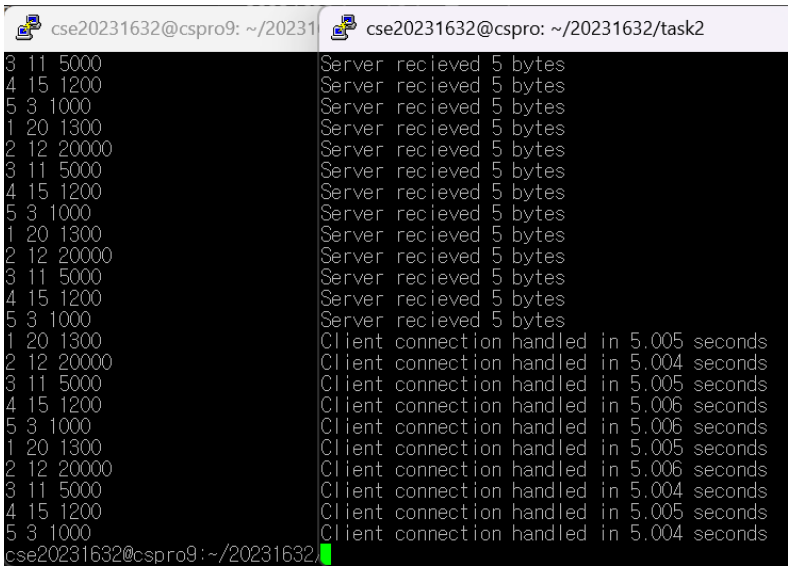
a) Event-based approach



측정 중 예시 하나

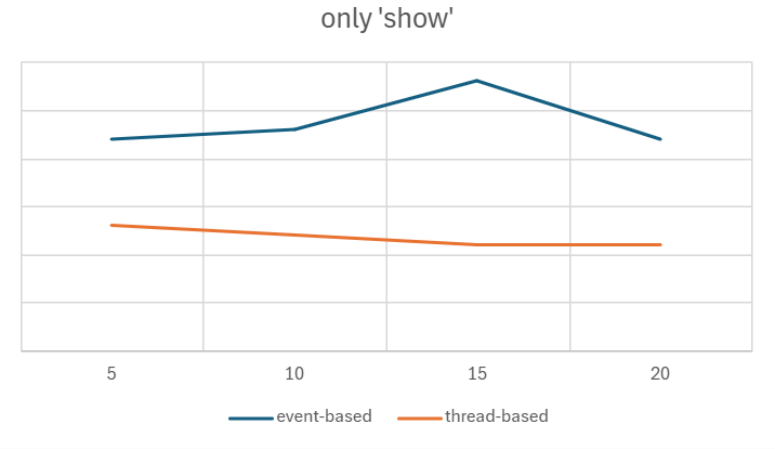
Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.0058	5.0064	5.0064	5.0062	5.0062	9,9876
10	5.0064	5.0063	5.0060	5.0066	5.0063	19,9748
15	5.0067	5.0068	5.0069	5.0069	5.0068	29,9593
20	5.0062	5.0061	5.0060	5.0063	5.0062	39,9505

b) Thread-based approach



측정 중 예시 하나

Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.0052	5.0054	5.0052	5.0054	5.0053	9,9894
10	5.0051	5.0052	5.0051	5.0053	5.0052	19,9792
15	5.0049	5.0049	5.0053	5.0054	5.0051	29,9694
20	5.0051	5.0051	5.0053	5.0050	5.0051	39,9592

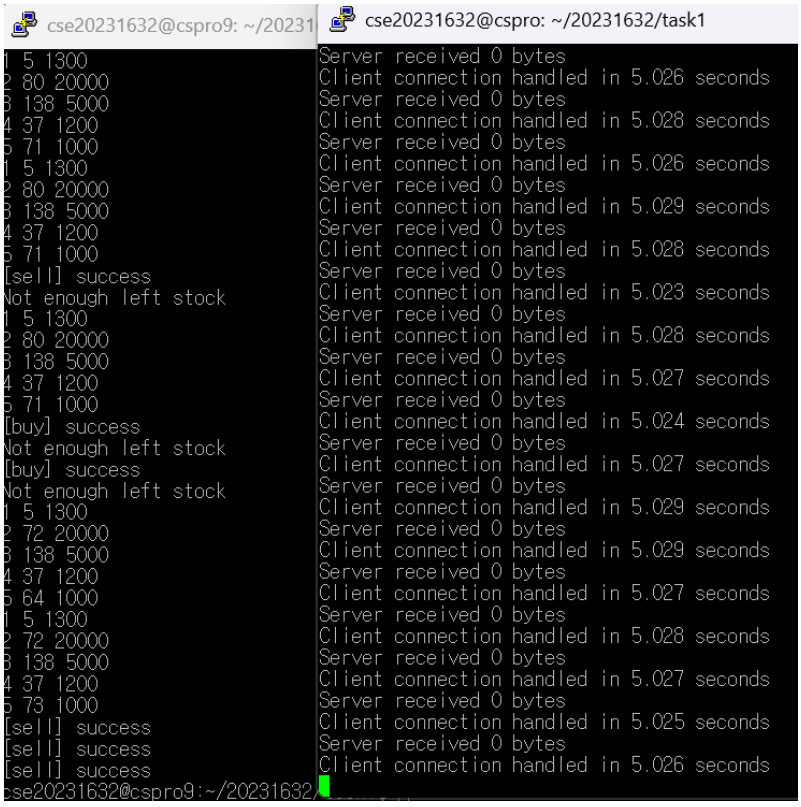


가로: num of client, 세로: 시간(s)

3. Client가 buy, sell, show 등을 섞어서 요청하는 경우

- ORDER_PER_CLIENT 50
- usleep(100000)
- Int option = rand()%3

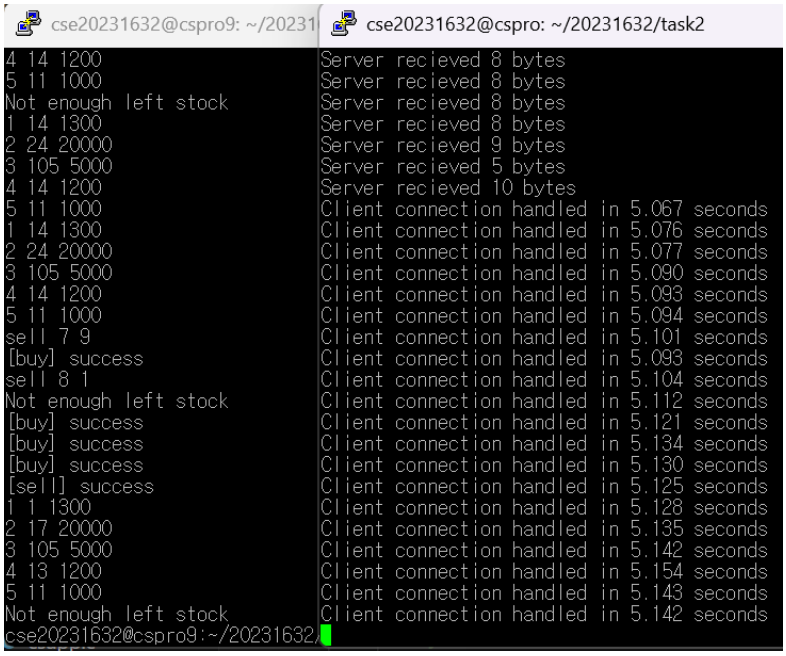
a) Event-based approach



측정 중 예시 하나

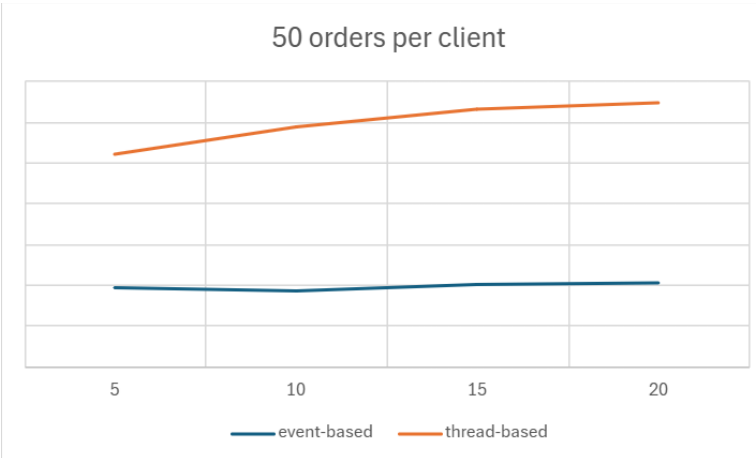
Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.0284	5.0260	5.0296	5.0288	5.0282	49,7196
10	5.0266	5.0274	5.0275	5.0262	5.0269	99,4649
15	5.0271	5.0274	5.0392	5.0270	5.0302	149,0994
20	5.0264	5.0435	5.0272	5.0269	5.0311	198,7637

b) Thread-based approach



측정 중 예시 하나

Num of Clients	Attempt				Average Time (s)	처리율
	1	2	3	4		
5	5.0842	5.1098	5.0886	5.0948	5.0944	49,0734
10	5.1002	5.1008	5.1243	5.1041	5.1074	97,89717
15	5.1141	5.1202	5.1181	5.1113	5.1159	146,6018
20	5.1263	5.1238	5.1118	5.1152	5.1193	195,3392



가로: num of client, 세로: 시간(s)

성능 측정 결과 및 분석

1) 모든 클라이언트가 buy 또는 sell 요청
   - 스레드 방식이 더 느렸다.
   - 이유: buy와 sell은 공유 자원에 대한 쓰기 작업이므로 스레드 간 락 경합이 발생하기 때문이다.
2) 모든 클라이언트가 show 요청
   - 이벤트 기반 방식이 더 느렸다.
   - 이유: show는 읽기 전용 작업이므로 스레드 방식에서는 병렬로 빠르게 처리되지만, 이벤트 기반 방식은 모든 요청을 순차적으로 처리하기 때문이다.
3) 클라이언트가 buy, show, sell을 섞어서 요청하고, 요청 수를 증가시킨 경우
   - 스레드 방식이 더 느렸고, 두 방식 모두 클라이언트 수가 많을수록 평균 처리 시간이 증가했다.
   - 이유: 스레드 방식은 혼합 요청 처리 시 락 경합과 컨텍스트 스위칭 비용이 증가하고, 이벤트 방식도 요청이 많아질수록 처리 병목이 발생하기 때문이다.

클아인트 개수 변화에 따른 처리율을 계산해보았다.

1) 확장성 측면

   이벤트 드리븐과 멀티스레드 방식 모두 클라이언트 수 증가에 따라 처리율이 선형적으로 증가한것을 볼 수 있었다. 다만, 이벤트 드리븐은 단일 스레드의 한계로 CPU 활용도가 낮아 대규모 클라이언트에서 성능 상승폭이 점차 감소하는 반면, 멀티스레드는 코어 수까지는 선형적 처리율 향상을 보인다.

2) 성능 비교

   읽기 집중(show) 작업에서는 멀티스레드의 병렬 처리로 인해 이벤트 드리븐 대비 50% 이상 높은 처리율을 보였으나, 쓰기 집중(buy/sell) 작업에서는 뮤텍스 경쟁으로 인해 이벤트 드리븐이 더 안정적인 성능을 나타냈다.

3) 수업 내용과의 일치성

이벤트 드리븐은 I/O 바운드 작업에서 효율적이라는 이론과 일치했으며, 멀티스레드는 CPU 바운드 작업에서 우세한 것으로 나타났다. 다만, 실제 구현에서는 쓰기 작업 시 스레드 간 동기화로 인해 이론보다 낮은 효율을 보였으며, 이는 뮤텍스 경쟁과 컨텍스트 스위칭의 실질적 비용을 반영한 결과라고 할 수 있다. 수업 시간에 언급하듯이, 실제에서 스레드 방식보다 이벤트 드리븐 방식을 더 잘 쓰는 이유를 이 프로젝트를 통해 알 수 있었다.