System Programming Project 3

담당 교수 : 이영민

이름 : 박지민

학번 : 20231552

1. 개발 목표

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

본 프로젝트의 목표는 다수의 클라이언트가 동시에 접속하여 주식 데이터를 조회하거나 거래할 수 있는 동시성 기반 주식 서버를 구현하는 것이다. 서버는 클라이언트로부터 show, buy, sell, exit과 같은 명령어를 입력 받아, 현재 주식의 잔여 수량 및 가격 정보를 조회하거나, 실제 주식의 수량을 변경하는 작업을 수행한다. 핵심적으로, 본 프로젝트는 다수의 클라이언트 요청을 효율적으로 처리할 수 있는 동시성 모델을 설계하고 구현하는 데 초점을 두며, 이를 위해 select 기반의 event driven 모델과 pthread 기반의 Thread Pool 모델을 각각 실험적으로 구현하고 성능을 비교 평가하였다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

select() 시스템 콜을 이용한 I/O Multiplexing 기반의 구조를 구현하였다. 단일 프로세스 내에서 다수의 클라이언트 소켓을 감시하고, 해당 소켓 중 입력이 준비된 소켓에 대해서만 명령을 처리하는 방식으로, 별도의 스레드나 프로세스를 생성하지 않고도 동시에 다수의 요청을 처리할 수 있다. 클라이언트 연결 관리 및 명령 처리 흐름은 select 루프 내에서 순차적으로 이루어진다.

2. Task 2: Thread-based Approach

pthread를 활용한 고정 크기 Thread Pool 기반 구조를 구현하였다. Master Thread는 클라이언트의 접속을 수락하고, 연결된 소켓을 공유 큐에 적재한다. Worker Thread들은 큐에서 소켓을 꺼내 요청을 병렬로 처리하며, read/write 동기화는 stock 노드 단위의 세마포어로 구현하였다. 이를 통해 다수의 클라이언트가 동시에 요청을 보내더라도 병렬로 처리된다.

3. Task 3: Performance Evaluation

각 서버 구조의 처리 성능을 비교 분석하기 위해 다양한 실험을 수행하였다. 클라이언트 수 및 요청 수 조합을 변화시키며 처리 시간과 처리율(ops/ms)을 측정하였고, 명령어 종류별(show, buy/sell) 성능 차이도 평가하였다.

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

이번 과제에서 구현한 서버는 하나의 프로세스에서 동시에 여러 클라이언트의 요청을 처리할 수 있도록 'select()' 기반의 I/O Multiplexing 기법을 사용하였다. 서버는 'fd_set' 구조를 활용해 감시할 파일 디스크립터들의 집합을 유지하고, select() 시스템 콜을 통해 이들 중 어느 소켓이 읽기 가능한 상태인지 판별한다. 클라이언트가 새롭게 접속하면 'listenfd'를 통해 연결 요청이 감지되고, 'Accept()'를 통해 연결을 수락한 뒤 해당 커넥션 소켓을 'read_set'에 등록하여 다음 루프에서 감시 대상으로 포함시킨다. 클라이언트로부터의 데이터는 'Rio_readlineb()'을 통해 수신되며, 요청에 따라 'handle_command()' 함수에서 show, buy, sell 명령어를 처리한다. 이 과정에서 각 소켓은 독립적으로 감시되고 처리되기 때문에, 블로킹 없이 다수의 클라이언트 요청을 효율적으로 순차 처리할 수 있다. 모든 클라이언트가 종료되면 서버는 stock 정보를 'stock.txt'에 저장하고 종료하여 일관된 데이터 상태를 유지하도록 하였다.

✓ epoll과의 차이점 서술

'select()' 방식은 구현이 간단하지만, 구조적 한계도 분명히 존재한다. 대표적으로 감시할 수 있는 파일 디스크립터 수가 `FD_SETSIZE`로 제한되어 있어, 대규모 서버 환경에는 적합하지 않다. 또한 select()는 이벤트 발생 여부를 판별한 후 'fd_set'이 내부적으로 변경되기 때문에, 매번 'read_set'을 원본으로부터 복사해야 하는 불편함과 오버헤드가 있다. 반면 'epoll'은 커널 내부에서 이벤트를 관리하며, 수천 개의 파일 디스크립터를 효율적으로 감시할 수 있도록 최적화되어 있다. 'epoll'은 Edge-triggered 모드를 지원하여 이벤트가 발생한 시점에서만 반응하게

할 수 있고, Level-triggered도 지원해 유연한 이벤트 모델 구성이 가능하다. 성능 면에서는 클라이언트 수 증가에 따른 부하가 거의 없기 때문에 고성능 서버에 적합하다.

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

  ✓ Master Thread의 Connection 관리

  Thread-based 구조에서의 서버는 하나의 Master Thread가 모든 클라이언트의 연결을 수락하고, 이를 처리할 Worker Thread에게 분배하는 역할을 한다. 본 구현에서는 'main()' 함수가 Master Thread의 역할을 수행하며, 클라이언트가 접속할 때마다 'Accept()'를 통해 연결을 수락하고, 해당 소켓 파일 디스크립터를 bounded buffer인 'sbuf'에 삽입한다. Master Thread는 연결을 직접 처리하지 않고, 오직 새로 들어온 클라이언트 소켓을 큐에 적재하는 역할만 수행함으로써, 병목 없이 빠르게 연결 요청을 받아들일 수 있도록 구성되었다. 이 과정에서 소켓 번호와 함께 클라이언트의 IP와 포트를 출력하여 연결 상태를 확인할 수 있도록 하였다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  서버가 시작될 때, 고정된 개수(NTHREADS)만큼의 Worker Thread를 생성하여 thread pool을 구성한다. 각 Worker Thread는 무한 루프를 돌며 'sbuf_remove()'를 통해 소켓 번호를 큐에서 하나씩 꺼내고, 이를 기반으로 클라이언트 요청을 처리한다. 큐에 저장된 소켓 번호를 기반으로 'handle_command()' 함수를 호출하고, 이 함수 내부에서 클라이언트의 명령을 파싱하여 주식 정보를 조회하거나 갱신한다. Worker Thread는 'pthread_detach()'를 통해 종료 시 자원을 자동으로 회수하도록 설정되어 있으며, 클라이언트와의 통신이 끝난 후에는 'Close()'를 통해 소켓을 닫는다. 또한 공유 자원인 주식 정보를 여러 쓰레드가 동시에 접근할 수 있기 때문에, 각 stock 노드마다 'w' 세마포어를 두어 임계 구역을 보호하고 있다. show 명령어의 경우 다수의 읽기 작업이 동시에 가능하므로 readers-writers lock 패턴을 적용하였고, buy/sell 명령어의 경우에는 단일 쓰기 작업만 가능하도록 하여 데이터를 보장한다.

-Task3 (Performance Evaluation)

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

✓ Configuration 변화에 따른 예상 결과 서술

이번 성능 평가에서는 서버 구조의 차이에 따른 처리 효율성을 정량적으로 비교하기 위해 총 소요 시간과 시간당 처리 요청수(throughput, ops/ms)를 주요 metric으로 정의하였다.

총 소요 시간은 일정 수의 클라이언트가 일정량의 요청을 서버에 보낸 뒤, 모든 요청을 완료하는 데 걸린 시간으로, 서버의 전반적인 응답 능력을 직관적으로 반영한다. Throughput은 총 명령 수를 소요 시간(ms)으로 나눈 값으로, 단위 시간당 서버가 얼마나 많은 요청을 실제로 처리했는지를 나타낸다. 이 두 지표는 단순 평균 응답 시간보다 전체 시스템의 동시성 처리 능력과 구조적 성능 차이를 파악하는 데 적합하다.

측정은 각 Task 서버를 대상으로 동일한 요청 조건 하에 클라이언트 수, 요청 수를 변화시키며 진행되었으며, 실험에서는 show, buy, sell 명령을 랜덤하게 조합하여 각 클라이언트가 고정된 수의 명령을 서버에 요청하는 방식으로 설정되었다. 측정 시점은 첫 번째 요청 직전과 마지막 응답 직후를 gettimeofday()로 기록하여 전체 경과 시간을 계산하였다.

실험 설계 초기 예상으로는, 클라이언트 수가 적을 경우 Task 1과 Task 2 모두 유사한 처리 성능을 보일 것으로 판단하였다. 이는 select 구조 또한 비동기 이벤트 기반으로 여러 요청을 병렬처럼 순차적으로 빠르게 처리할 수 있기 때문이다.

반면, 클라이언트 수가 수백~수천 단위로 증가할 경우, select 방식은 fd_set의 크기 제한(FD_SETSIZE)과 반복 루프마다 read_set 복사 및 전체 순회를 수행해야 하는 구조적 오버헤드로 인해 처리 속도가 점진적으로 저하될 것으로 예상하였다.

한편 Task 2는 Worker Thread를 통한 병렬 처리 구조를 갖추고 있으므로, 요청량이 증가할수록 select 방식보다 더 높은 throughput을 유지할 것으로 예측되었다. 다만 본 실험에서는 Thread 수를 고정(50개)하였기 때문에, 클라이언트 수가 이를 초과하는 시점부터는 큐에 대기하는 요청이 누적되며 성능 저하가 발생할 수 있다는 점도 고려하였다. 또한 명령어의 종류 측면에서도 차이가 예상된다. show 명령은 read-only 연산이므로 readers-writers lock을 통해 다수의 스레드가 동시에 처리할 수 있는 반면, buy/sell은 쓰기 동기화가 필요하므로 병렬성에 제약이 따른다. 따라서 이론상으로는 show 명령에서 더 높은 병렬 처리율을 기대할 수 있으며, 이는 Thread 기반 구조에서 더욱 뚜렷하게 나타날 수 있다.

C. 개발 방법

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

- **Task1**

Task 1의 Event-driven 구조를 구현하기 위해 먼저 전체 연결을 관리할 수 있는 'pool' 구조체를 정의하였다. 이 구조체에는 최대 파일 디스크립터 값을 저장하는 maxfd, 클라이언트들의 연결 상태를 추적하는 clientfd 배열, 클라이언트의 입출력을 위한 rio_t 배열, 그리고 감시할 디스크립터 집합인 read_set, ready_set 등이 포함된다. 이를 통해 서버는 다수의 클라이언트를 효율적으로 관리할 수 있도록 설계되었다.

서버는 main() 함수에서 Open_listenfd()를 통해 리스닝 소켓을 초기화하고, 이후 init_pool() 함수를 호출하여 pool 구조체를 설정한다. init_pool()에서는 모든 clientfd를 -1로 초기화하고, read_set에 listenfd를 등록한다. 서버는 무한 루프 내에서 pool.read_set을 pool.ready_set에 복사한 후, select() 시스템 콜을 호출하여 어떤 소켓에 이벤트가 발생했는지를 확인한다.

select()의 결과로 listenfd가 활성화되었을 경우에는 새로운 클라이언트 연결이 들어온 것으로 판단하고, Accept()를 호출하여 연결을 수락한다. 이후 add_client() 함수를 통해 연결된 소켓을 pool.clientfd에 등록하고, read_set에 포함시키며, Rio_readinitb()를 사용해 해당 소켓의 buffered I/O를 초기화한다. 또한 새로운 소켓이 maxfd보다 큰 경우에는 maxfd 값을 갱신하여 다음 select() 호출 시 정상적으로 감시가 이루어지도록 한다.

기존 클라이언트의 소켓에서 이벤트가 발생했을 경우에는 check_clients() 함수를 통해 각 소켓을 순회하며 FD_ISSET() 매크로로 활성화 여부를 검사한다. 클라이언트로부터 데이터가 수신되었을 경우에는 Rio_readlineb()로 해당 요청을 읽고, handle_command() 함수를 통해 명령어를 분석하여 show, buy, sell, exit 동작을 수행한다. 클라이언트가 연결을 종료하면 해당 소켓을 닫고, read_set에서 제거하며, clientfd 배열의 해당 인덱스를 -1로 초기화한다. 모든 클라이언트가 종료되었을 때는 save_stock_data()를 호출하여 현재 트리 구조의 주식 정보를 stock.txt 파일에 저장한다.

이러한 구조를 통해 서버는 다수의 클라이언트 요청을 병렬적으로 처리

할 수 있으며, blocking 없이 효율적인 연결 관리를 실현할 수 있다. 추가적인 프로세스나 스레드를 생성하지 않기 때문에 오버헤드가 적고, 구조가 단순하며 디버깅이 용이하다는 장점이 있다.

Task2

Thread-based 서버 구조를 구현하기 위해 우선 bounded buffer를 표현하는 sbuf_t 구조체를 정의하였다. 이 구조체는 클라이언트 연결 소켓 번호를 저장할 원형 큐(buf), 큐의 크기(n), front와 rear 인덱스, 그리고 동기화를 위한 세마포어(mutex, slots, items)로 구성된다. 서버가 실행되면 main() 함수에서 sbuf_init()을 호출하여 해당 큐를 초기화하고, 고정 개수의 Worker Thread들을 생성하여 thread() 함수에 진입하도록 한다.

main() 함수는 Master Thread의 역할을 수행하며, 클라이언트가 접속할 때마다 Accept()를 통해 연결을 수락하고, 해당 소켓 번호를 sbuf_insert()를 통해 큐에 저장한다. 이때 Master Thread는 클라이언트 요청을 직접 처리하지 않고, Worker Thread에게 분배하는 데만 집중하도록 설계되었다. Worker Thread는 thread() 함수 안에서 무한 루프를 돌며 sbuf_remove()를 통해 큐에서 소켓을 꺼내고, 해당 소켓에 대해 handle_command()를 호출하여 명령어를 처리한다.

handle_command() 함수는 내부적으로 Rio_readlineb()를 통해 클라이언트 요청을 읽고, 명령어 종류에 따라 처리한다. show 명령은 collect_show()를 호출하여 전체 주식 정보를 문자열로 수집하고, buy와 sell 명령은 find_stock_by_id()를 통해 해당 ID의 주식 노드를 탐색한 뒤, 재고 수량을 수정한다. buy 및 sell 로직은 handle_command() 함수 내부에 직접 구현되어 있으며, 명령 처리 후 결과 메시지를 Rio_writen()을 통해 클라이언트에게 전송한다.

주식 정보를 담은 트리는 동시 접근이 발생할 수 있으므로, 각 노드에 쓰기 보호용 세마포어(w)를 두어 buy 및 sell 작업에서 단독 접근을 보장하였다. show 명령과 같이 읽기 작업의 경우, readcnt와 mutex 세마포어를 활용하여 readers-writers lock 패턴을 구현하였고, 이를 통해 동시 다중 읽기를 허용하였다.

또한, 클라이언트 연결 수를 추적하기 위해 전역 변수 connected_clients를 정의하고, 이 값을 변경할 때는 client_count_mutex 세마포어를 이용해 동기화하였다. 클라이언트가 exit 명령을 통해 종료되거나 연결이 끊기면 소켓을 닫고

connected_clients 값을 감소시키며, 마지막 클라이언트가 종료된 시점에서는 자동으로 stock.txt에 주식 정보를 저장한다.

이러한 방식으로 서버는 동시 접속한 다수의 클라이언트를 Worker Thread에 분산 처리시킬 수 있으며, Master-Worker 구조와 thread-safe한 공유 자원 접근이 결합된 효율적인 멀티스레드 서버 구조를 완성할 수 있었다.

---

3. 구현 결과

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

Task 1의 Event-driven 방식은 select() 시스템 콜을 기반으로 하여 다수의 클라이언트를 하나의 서버 프로세스에서 동시에 처리할 수 있도록 구현되었으며, 예상한 대로 안정적으로 동작하였다. 클라이언트가 순차적으로 또는 동시에 접속하여 show, buy, sell, exit 명령을 보낼 경우, 서버는 각 클라이언트의 요청을 비동기적으로 감지하고 독립적으로 처리할 수 있었다. 예를 들어 하나의 클라이언트가 특정 주식을 구매한 후, 다른 클라이언트가 동일 주식에 대해 잔여 수량을 조회하는 경우에도 수정된 결과가 반영되어 나타났으며, 이는 서버의 동기화 및 상태 관리가 잘 작동하고 있음을 보여주었다.

또한, 마지막 클라이언트가 종료되었을 때 서버는 자동으로 전체 주식 데이터를 stock.txt 파일에 저장하였으며, 저장된 파일을 다시 로딩해도 일관된 데이터가 유지되는 것을 확인하였다. 이로써 연결 종료 시점에 상태 저장이 정상적으로 이뤄지고 있음을 검증할 수 있었다. select 기반 구조의 특성상, 새로운 클라이언트가 접속할 때마다 fd_set에 소켓이 동적으로 등록되고, 최대 파일 디스크립터가 갱신되며, 클라이언트 종료 시에는 해당 소켓이 FD_CLR 되어 메모리와 리소스를 적절히 정리하였다.

전반적으로 Task 1에서 요구된 기능은 모두 구현 및 정상적으로 동작하였으며, 이벤트 기반 구조가 의도한 바대로 동시 접속과 요청 처리의 효율성을 확보할 수 있었음을 확인하였다

Task2

Thread-based 서버는 다수의 클라이언트가 동시에 접속하여 주식 정보를 요청하거나 수정하는 상황에서도 안정적으로 작동하였다. main() 함수에서 클라이언트 연결 요청을 수락하고, 이를 Worker Thread에게 분배하는 구조 덕분에 Master Thread는 연결 수락에만 집중할 수 있었고, Worker Thread들은 독립적으로 명령어를 처리함으로써 요청 처리 속도가 향상되었다. 실제로 클라이언트가 show 명령어를 동시에 요청할 경우에도 출력 결과가 충돌 없이 정확하게 반환되었으며, buy 또는 sell 요청이 동시에 들어온 상황에서도 재고 수량이 정확하게 반영되는 것을 확인하였다.

또한 각 Worker Thread는 pthread_detach()로 분리되어 있어, 명시적인 pthread_join() 없이도 종료 시 자원이 정리되었으며, 접속 종료 후에는 Close()와 함께 소켓이 닫히고 연결 수가 감소하였다. 마지막 클라이언트가 종료되었을 때는 자동으로 stock.txt에 데이터가 저장되었으며, 저장된 파일을 다시 로딩했을 때 상태가 일관되게 유지되는 것도 확인되었다.

구현된 Thread Pool 방식은 select 기반의 구조에 비해 병렬 처리 효율이 높고, 클라이언트 수가 많아질수록 명확한 성능 차이를 보였다. 구현 과정에서 주요 기능은 모두 성공적으로 구현되었으며, 미구현된 부분은 없었다. 추후 개선점으로는 Worker Thread 수를 동적으로 조절하는 기능이나 graceful shutdown 구현 등을 고려해볼 수 있다.

성능 평가 결과 (Task 3)

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

본 실험은 서버 구조와 클라이언트 요청 패턴에 따른 성능 변화를 평가하기 위해 총 명령 수가 동일한 조건 하에서 다양한 클라이언트 수 및 요청 수 조합을 테스트하였다. 측정 지표로는 (1) 총 소요 시간(ms)과 (2) 시간당 처리율(ops/ms)을 사용하였다. 단순한 처리 시간만으로는 서버 구조와 병렬성 차이를 비교하기 어렵기 때문에, 단위 시간당 처리된 명령 수인 처리율을 함께 측정함으로써 효율성을 보다 명확히 평가하고자 하였다. 클라이언트 요청의 시작 시점은 첫 번째 자식 프로세스 또는 스레드가 요청을 전송하기 직전으로 정의하였고, 종료 시점은 마지막 클라이언트가 서버 응답을 받고 종료되었을 때로 설정하였다. 이를 통해 전체 요청 처리에 걸린 실제

경과 시간을 측정하였다. 처리율은 명령 수 / 소요 시간(ms)으로 계산하였다.

```
4 129 500
[sell] success
[sell] success
[sell] success
[sell] success
[sell] success
[sell] success
[buy] success
[sell] success
[buy] success
[sell] success
3 168 500
2 131 200
1 358 300
5 346 400
4 136 500
[sell] success
Elapsed time: 50048.71 ms
```

: 소요시간 캡쳐

```
cse20231552@cspro8:~/project3/task1$ ./multiclient 172.30.10.11 60212 10
child 469740
child 469741
child 469742
child 469743
child 469744
child 469745
child 469746
child 469747
child 469748
child 469749
3 71 500
2 85 200
1 379 300
5 345 400
4 339 500
[sell] success
[sell] success
3 74 500
2 85 200
1 379 300
5 349 400
4 339 500
[buy] success
[sell] success
```

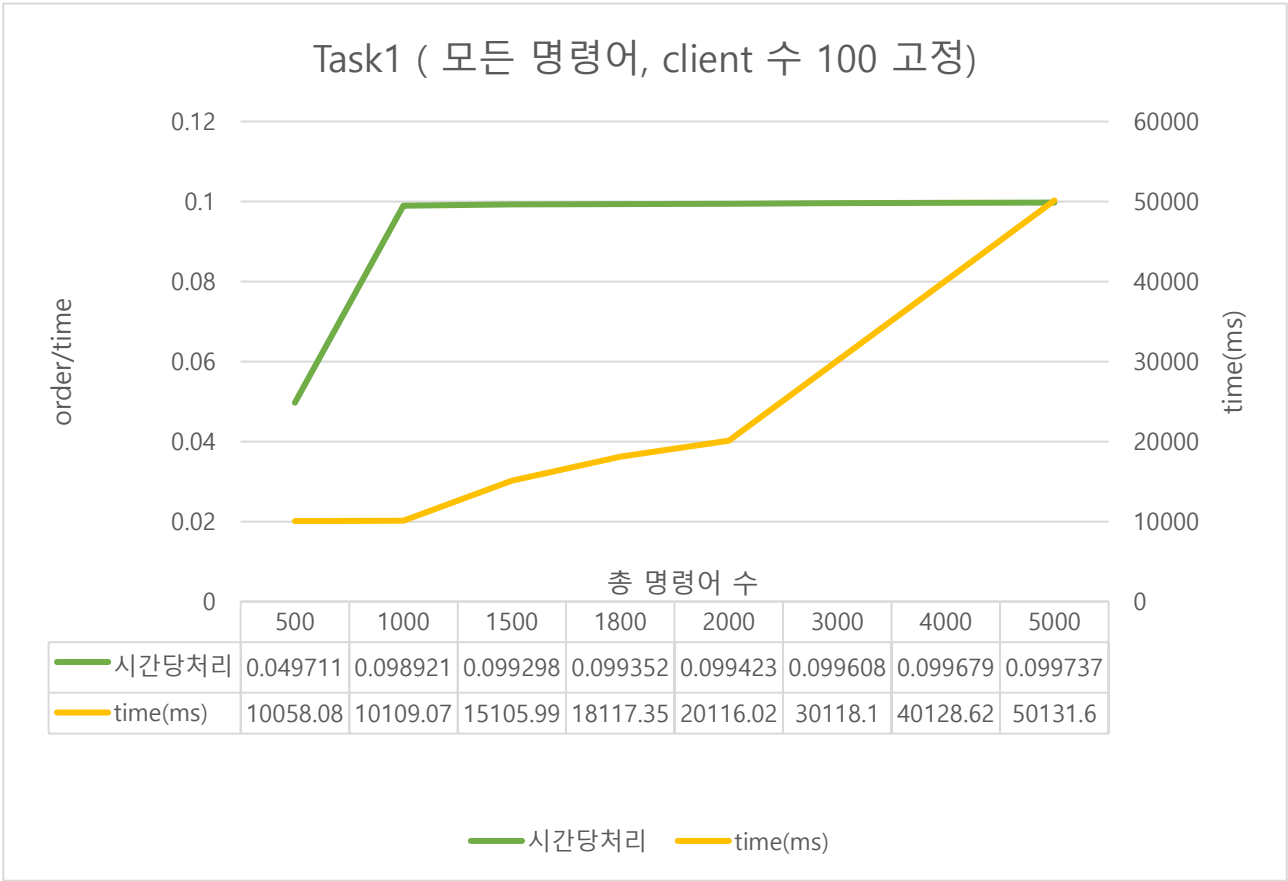
: 실행 과정

1. 모든 명령어 조합에 대한 Task 1 성능 분석 (select 기반)

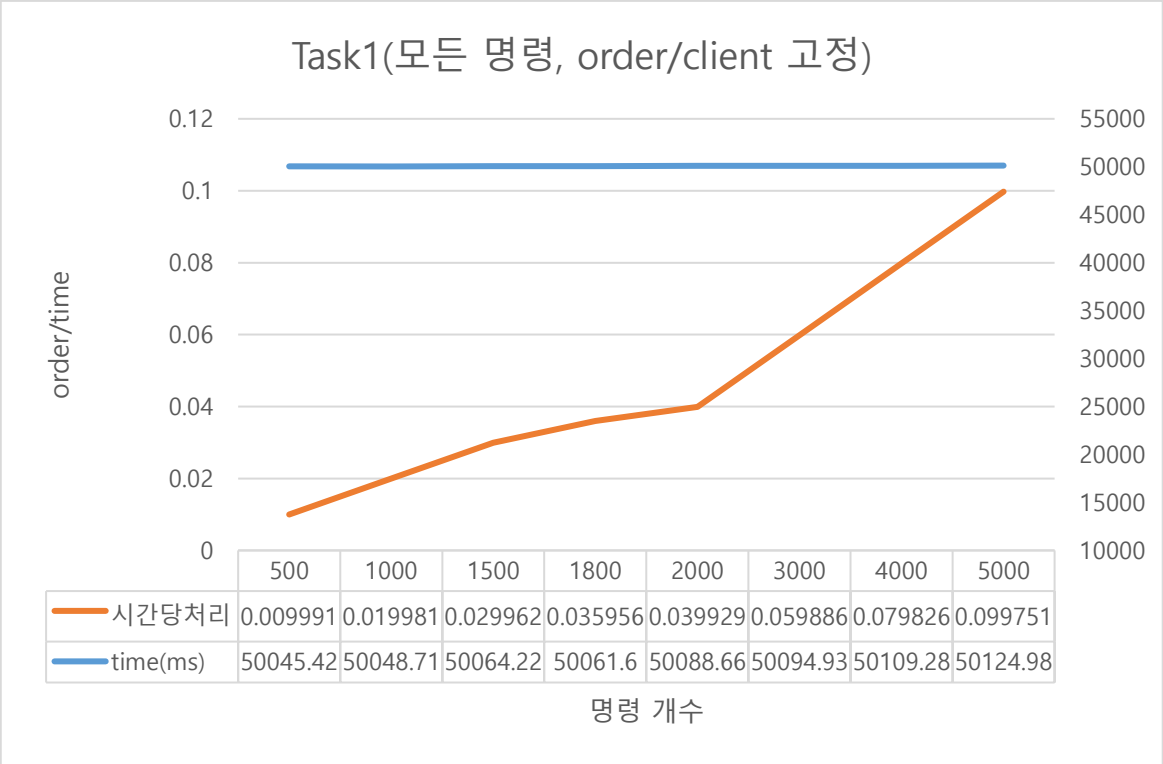
Task 1에서는 클라이언트 수를 고정하고 클라이언트당 요청 수를 증가시키는 실험과, 클라이언트당 요청 수를 고정하고 클라이언트 수를 증가시키는 두 가지 실험을 수행하였다.

첫 번째 실험에서는 클라이언트 수를 100명으로 고정하고, 총 요청 수를 500개에서 5000개까지 증가시켰다. 이 경우 총 처리 시간은 요청 수에 따라 선형적으로 증가하였고, 처리율은 약 0.099 ops/ms 수준에서 일정하게 유지되었다. 이는 select 기반 구조가 순차 처리 방식이지만, 비교적 안정적으로 다수의 요청을 처리할 수 있음을 보여준다.

두 번째 실험에서는 클라이언트 당 요청 수를 50개로 고정하고, 클라이언트 수를 1명에서 100명까지 증가시켰다. 클라이언트 수가 증가할수록 select는 각 요청을 빠르게 감지하고 처리하며 처리율이 꾸준히 증가하는 경향을 보였고, 100명 시점에서는 처리율이 0.09975 ops/ms로 포화되었다. 이는 각 요청이 짧은 시간 내에 서버에 도달하고 순차적으로 처리되기 때문에 발생한 현상으로, 구조적으로 병렬은 아니지만 비차단 처리 모델이 효율적으로 작동했음을 보여준다.



	500	1000	1500	1800	2000	3000	4000	5000
시간당처리	0.049711	0.098921	0.099298	0.099352	0.099423	0.099608	0.099679	0.099737
time(ms)	10058.08	10109.07	15105.99	18117.35	20116.02	30118.1	40128.62	50131.6



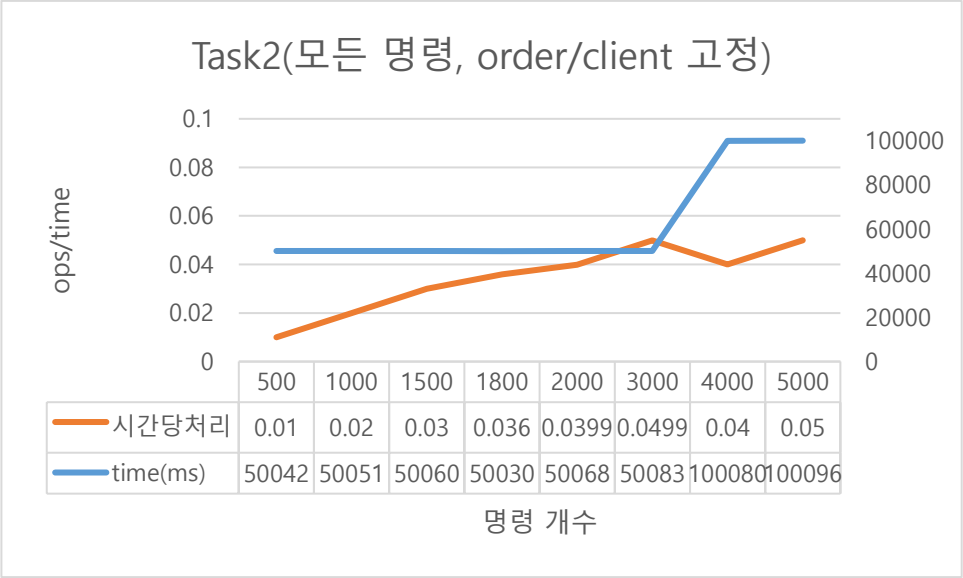
	500	1000	1500	1800	2000	3000	4000	5000
시간당처리	0.009991	0.019981	0.029962	0.035956	0.039929	0.059886	0.079826	0.099751
time(ms)	50045.42	50048.71	50064.22	50061.6	50088.66	50094.93	50109.28	50124.98

2. Task 2 성능 분석 (Thread Pool 기반 pthread 구조)

Task 2는 고정된 50개의 Worker Thread로 구성된 Thread Pool을 사용하여, 클라이언트로부터의 요청을 병렬로 처리한다. 클라이언트 수가 50 이하일 때는 각 요청이 즉시 할당된 스레드에서 처리되므로, Task 1과 유사한 처리 시간을 보였다. 그러나 클라이언트 수가 50을 초과하면, 나머지 요청은 큐에서 대기하게 되며, Thread Pool의 병렬 처리 한계에 따라 처리 지연이 발생하기 시작하였다.

특히 명령 2500개 이후부터는 성능 저하가 본격화되며, 4000개 실험에서는 일시적으로 처리율이 0.0004 ops/ms 수준으로 급락하였다. 이는 Thread Pool의 고정 크기 구조에서 스레드의 처리 속도를 초과하는 요청이 들어올 경우, 큐의 포화 및 병목 현상이 발생할 수 있음을 의미한다. 이후 5000 ops 실험에서는 다시 0.0499 ops/ms 수준으로 회복되었으나, 이는 스레드 부하가 간헐적으로 분산되며 일부 요청이 지연되었기 때문이다.

```
[CLIENT 1137378] Sending (50/50): buy 4 :
[buy] success
Elapsed time: 50060.39 ms
```



4. 명령어 종류별 분석

Task1

task1	명령 개수	500	1000	1500	1800	2000	3000	4000	5000
show만	time	5106.9	10111.28	15112.42	18118.68	20126.01	30127.19	40129.54	50135.21
client 수 100 고정	시간당처리	0.097907	0.098899	0.099256	0.099345	0.099374	0.099578	0.099677	0.09973

task1	명령 개수	500	1000	1500	1800	2000	3000	4000	5000
sell/buy 만	time	5106.8	10109.03	15095.38	18117.03	20123.38	30114.47	40128.93	5012.99
client 수 100 고정	시간당처리	0.097909	0.098921	0.099368	0.099354	0.099387	0.09962	0.099679	0.997409

Task2

task2	명령 개수	500	1000	1500	1800	2000	2500	4000	5000
show만	time	50035.83	50049.93	50057.62	50067.64	50060.32	50078.41	10063.67	10099.66
order/client 고정	시간당처리	0.009993	0.01998	0.029965	0.035951	0.039952	0.049922	0.397469	0.495066

task2	명령 개수	500	1000	1500	1800	2000	2500	4000	5000
sell/buy만	time	50042.37	50046.65	50059	50056.29	50068.56	50083.54	100083.2	100106
order/client 고정	시간당처리	0.009992	0.019981	0.029965	0.03596	0.039945	0.049917	0.039967	0.049947

명령어 유형별로 서버 구조의 영향을 보다 명확히 관찰하기 위해, show 명령어만 실행한 경우와 buy/sell 명령어만 실행한 경우로 나누어 Task 1과 Task 2의 성능을 비교하였다. 실험 결과, 두 명령어 유형 간 처리 시간은 Task 1, Task 2 모두에서 큰 차이가 발생하지 않았으며, 특히 Task 1에서는 거의 동일한 수준의 처리 시간이 측정되었다. 이는 직관적으로는 예상과 다를 수 있으나, 서버 구조적 특성과 명령어 내부 동작 방식에 기인한 것으로 해석된다.

```
[CLIENT 1155335] Sending (50/50): buy 3 3
[buy] success
[sell] success
[buy] success
[CLIENT 1155337] Sending (50/50): sell 4 5
[sell] success
[CLIENT 1155338] Sending (50/50): buy 5 2
[CLIENT 1155339] Sending (50/50): buy 3 2
[buy] success
[buy] success
[CLIENT 1155336] Sending (50/50): sell 5 5
[sell] success
Elapsed time: 50046.65 ms
```

Task 1에서는 모든 명령어—read-only인 show든, write 작업이 포함된 buy/sell이든—서버 루프 안에서 동일한 방식으로 순차 처리되기 때문에 명령의 종류에 따른 처리 시간 차이가 거의 발생하지 않는다. 각 명령어는 해당 클라이언트의 소켓이 read_set에 포함되어 있는지 검사한 후, handle_command() 내부에서 처리된다. 이때 show는 트리를 순회하여 데이터를 읽는 연산이고, buy/sell은 세마포어를 활용해 데이터를 수정하는 연산이지만, select 루프 내에서는 클라이언트당 단일 요청만 처리하므로 전체 루프 입장에서는 처리 비용 차이가 크게 반영되지 않는다. 결과적으로 show와 buy/sell 모두 클라이언트 수에 따라 동일한 속도로 순차적으로 처리되므로, 측정된 시간 차이는 거의 없다.

Task 2에서는 내부적으로는 병렬 처리를 수행하지만, 한 요청이 처리되는 단위 시간은 명령어 간 큰 차이가 나지 않는 구조다. show 명령어는 readers-writers lock을 통해 여러 스레드가 동시에 읽기를 수행할 수 있도록 설계되어 있어 병렬성이 높고, buy/sell은 노드 단위로 세마포어를 이용한 쓰기 동기화가 적용되어 상대적으로 직렬화된다. 이러한 구조적 차이는 명령어 간 성능 차이를 유발할 수 있는 요인이지만, 본 실험에서는 클라이언트 수와 요청 수 증가에 따른 전체 처리 시간의 차이는 크지 않게 나타났다.

이는 각 클라이언트가 단일 명령만을 수행하고 종료되는 구조에서, 명령 간 경합이 충분히 누적되지 않았기 때문으로 보인다. 즉, 구조적으로는 show 명령의 병렬성이 훨씬 높지만, 실험 구성에서는 이러한 차이가 처리율에 뚜렷하게 반영되지는 않았다. 따라서 Task 2에서는 명령어 복잡도보다는 Thread Pool의 포화 여부와 요청량 자체가 성능 결정 요소로 더 크게 작용한 것으로 해석된다.

3. 구조적 비교 및 결론

Task 1은 select 기반의 이벤트 루프 구조로 구현이 단순하고, 소수의 클라이언트

나 낮은 요청량을 처리하는 환경에서는 안정적인 성능을 제공한다. 단일 루프에서 모든 요청을 순차적으로 감시하고 처리하기 때문에, 구조적으로 병렬성이 부재하며, 클라이언트 수나 요청 수가 증가할수록 처리 시간도 선형적으로 증가하는 경향을 보인다. 특히 스레드나 큐 없이 작동하므로 병목은 루프의 I/O 감시 및 처리 속도에 의해 결정된다.

반면 Task 2는 Thread Pool 기반의 구조로, 초기 구현 복잡도는 다소 높지만, 각 요청을 스레드에 분산시켜 병렬로 처리할 수 있다는 장점을 지닌다. 클라이언트 수가 스레드 수 이내일 때는 매우 효율적이며, 멀티코어 환경에서도 스레드들이 독립적으로 실행되어 높은 처리량을 확보할 수 있다. 그러나 실험 결과에서 확인되었듯, 고정된 스레드 수를 초과하는 요청이 동시에 유입되는 경우 큐가 포화되고 처리 지연이 발생하는 한계도 존재하였다.

또한 구조적으로는 show 명령이 buy/sell에 비해 병렬 처리에 유리함에도, 이번 실험에서는 클라이언트당 단일 요청을 처리하는 간단한 시나리오로 인해 명령어 간 성능 차이를 뚜렷하게 관측하기는 어려웠다. 이는 실험 구성에 따른 해석의 한계로 볼 수 있으며, 보다 복잡한 요청 패턴이나 지속적인 명령 스트림 상황에서는 구조적 차이가 더욱 분명히 드러날 수 있을 것이다.