System Programming Project 3

담당 교수 : 김영재

이름 : 정희선

학번 : 20231609

1. 개발 목표

이 프로젝트에서는 network 프로그래밍 관련 배경 지식을 활용하여 stock sever를 구현한다. 주식 서버는 주식 관련 정보들을 저장하고 있다. 이 때 stock server는 concurrent하게 구현해야 한다. 즉, 여러 client가 동시에 접속하여 요청을 올바르게 수행할 수 있어야 한다. client들은 sell, buy, show, exit의 네 가지 요청을 할 수 있다. 이를 위해 첫 번째 구현 방법인 event-driven server와 두 번째 구현 방법인 tread-based server, 두 가지 방법으로 동시 주식 서버를 구현한다. 이 후 두가지 주식 서버에 대해 성능을 평가하고 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Select() system call을 활용한 I/O multiplexing 주식 서버를 구현할 수 있다. I/O multiplexing이란 하나의 통신을 통해 둘 이상의 데이터를 전송하는 기술이다. 이를 통해 단일 process에서 여러 client의 연결을 동시에 처리할 수 있다. Select() 함수 호출 시 여러 socket 중 data ready 상태가 된 socket이 있을 때까지 대기한다. 이후 해당 socket에 대해 데이터를 읽어온다.

2. Task 2: Thread-based Approach

   Pthread를 사용하여 multi-thread 기반 주식 서버를 구현할 수 있다. 여러 개의 thread를 사용해 진정한 의미의 동시 주식 서버를 구현할 수 있다. Producer-consumer 패턴을 적용하여 master tread가 client의 연결을 수락하면 worker thread가 실제 요청을 수행하는 구조가 된다. 또한, reader-writer lock을 사용해 동시성 제어를 수행하여 thread-safe한 동시 주식 서버를 구현할 수 있다.

3. Task 3: Performance Evaluation

   두 서버 구현 방식의 성능을 비교 분석하기 위해 multiclient.c를 수정하였다. Client 수와 work-road 타입을 변경해가며 각 방식의 효율성을 평가할 수 있다. Thread-based 구현 방식으로 만든 두 번째 서버는 추가로 stockserver.c의 thread의 개수까지도 변경해가며 성능을 분석할 수 있다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  Select() system call은 여러 개의 file descriptor를 동시에 모니터링한다. 이 때, 이벤트가 발생할 때까지 대기한다. Pool 구조체는 client의 연결 상태를 관리한다. Read_set은 원본을 유지할 수 있는 전체 감시 대상 집합이고, ready_set은 실제로 준비된 대상으로 select 함수가 모니터링하여 이번에 이벤트가 발생한 fd만 채워주는 집합이다. Select에서 새로운 client 연결이 감지되면 add_client()로 pool에 새로운 client를 추가하고, 기존 client의 요청이 감지된다면 check_clients()에서 순차적으로 요청을 처리한다. 따라서 이 방식에서는 select() 함수가 자체적으로 동시성을 제어하므로 추가적인 동시성 제어가 필요하지 않다.

  ✓ epoll과의 차이점 서술

  select와 epoll 모두 커널에게 fd 리스트 중 준비된 것이 생기면 알려달라고 요청하는 기능을 하지만, select()는 O(n)의 시간 복잡도로 모든 file descriptor를 검사한다. Epoll은 그와 달리 O(1)의 시간 복잡도로 이벤트가 발생한 file descriptor를 반환한다. 따라서 epoll이 많은 연결을 효율적으로 처리할 수 있다.

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

  ✓ Master Thread의 Connection 관리

  Main thread가 master tread 역할을 수행한다. Accept()함수로 새로운 client의 연결을 대기하며, 연결이 수락되면 해당 socket descriptor를 공유 버퍼 sbuf에 삽입하고 client counter를 증가시킨다. 이 때 sbuf는 어떤 client를 처리할지를 관리하는 공유 자원이다. 연결 관리는 non-blocking 방식으로 처리되므로 새로운 연결 요청을 지속적으로 수락할 수 있다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  NTHREADS 개수만큼 worker thread를 미리 생성하여 풀을 구성한다. 각 worker thread는 sbuf_remove() 함수를 통해 공유 버퍼 sbuf에서 client socket

을 가져와 처리한다. 또한, semaphore(slots, items, mutex)를 사용하여 producer-consumer 동기화를 구현하였다. Slots은 빈 슬롯 개수를, items는 데이터 개수를 나타내고, mutex는 producer와 consumer 동작의 lock을 보장한다. 이를 통해 thread-safe한 버퍼를 구현하였다.

- **Task3 (Performance Evaluation)**

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

  동시 처리율(concurrent throughput)을 주요 지표로 설정하였다. 이를 구하는 공식은 (client 수 * ORDER_PER_CLIENT) / elapsed_time 이다. 이 지표를 통해 서버의 동시성 처리 능력을 직접적으로 측정할 수 있으므로 선정하였다. 또한, client 수 증가에 따른 확장성을 정량적으로 비교할 수 있다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  Client 수 증가: event-driven 방식은 단일 스레드로 처리하므로 성능이 저하할 것이고, thread-based 방식은 다중 스레드로 병렬 처리가 가능해지므로 성능이 좋아질 것이다.

  Show-only 워크로드: thread-based에서 reader-writer lock을 통해 동시 읽기가 가능해지므로 성능이 좋아질 것이다.

  Buy/sell 워크로드: 두 방식 모두 순차 처리가 필요하므로 성능이 크게 변하지 않을 것이다.

  Thread-based 서버에서 thread 수 조정: 한 번에 더 많은 수의 병렬 처리가 가능해지므로 성능이 좋아질 것이다.

C. 개발 방법

- Task 1

Pool 구조체를 추가하여 client 연결 상태를 관리하기 위한 변수를 설정한다.

➔ Maxfd: 모니터링할 fd 최대값

➔ Read_set (B. 개발 내용에서 설명함)

➔ Ready_set (B. 개발 내용에서 설명함)

➔ Clientfd: 연결된 client socket fd 저장하는 배열

➔ Clientrio: client 별 독립 robust I/O buffer 관리

Client pool을 관리하기 위한 함수를 추가한다.

➔ Init_pool(): 서버 시작 시 client pool 초기화

➔ Add_client(): 새로 연결된 client를 pool에 추가

➔ Check_clients(): select를 통해 준비된 client의 요청 순차 처리

주식 거래 명령어 처리와 응답을 위한 handle_client_command() 함수를 추가한다.

모든 client 연결 종료 시 stock.txt 파일에 데이터를 저장하기 위한 zero_client() 함수를 추가한다.

- Task 2

Reader-writer lock을 구현하기 위해 Item 구조체에 변수와 세마포어를 구현한다.

➔ Readcnt: 주식 정보 노드를 읽고 있는 client의 수

➔ Mutex: readcnt의 lock을 관리하기 위한 semaphore

➔ W: 독립적인 writer를 보호하기 위한 semaphore

공유 버퍼를 기반으로 한 producer-consumer 구조를 구현하기 위해 sbuf_t 구조체와 관련 함수를 추가한다.

Worker thread 구현을 위해 thread() 함수를 구현한다.

Thread-safe한 client의 주식 서버 요청 처리를 위해 echo_cnt() 함수를 구현한다.

Client 수를 추적하여 모든 client와의 연결이 종료되면 stock.txt 파일에 저장하기 위해 increment_client_count()와 decrement_client_count 함수를 구현하고, 이 역시 semaphore를 통해 thread-safe하게 관리한다.

- Task 3

Multiclient 프로그램에 gettimeofday() 함수를 사용해 시간을 측정한다.

워크로드 타입을 enum을 통해 정의 후, 사용자의 입력에 따라 선택적으로 요청을 생성하는 로직을 만들었다.

성능 결과를 계산하여 출력한다.

3. 구현 결과

두 가지 방식의 서버 모두 성공적으로 구현하였다. Event-driven 서버는 select 기반 안정적인 다중 client 처리가 가능하고, thread-based 서버는 병렬 처리로 높은 동시성을 제공한다. 위에서는 언급하지 않았으나 binary search tree 자료구조로 주식 데이터를 관리하여 효율적으로 client의 요청을 수행할 수 있었다.

또한, thread-based 서버에서 reader-writer 락을 통한 동시성 제어와 producer-consumer 구조로 인해 안전한 multi-thread(multi-client) 서버를 구현할 수 있었다.

개선 방안으로는, 두 종류의 서버 모두 client에서 exit 요청 입력 시 stock server에서는 연결이 끊기지만 반복문 이슈로 exit 이후 명령어에 대해 한 번 echo 후 완전히 종료된다는 점이 있다.

4. 성능 평가 결과 (Task 3)

- Event-driven server

```
=== Performance Results ===
Total elapsed time: 9.017 seconds
Number of clients: 10
Requests per client: 10
Total requests processed: 100
Concurrent throughput: 11.09 requests/second
Average processing time per request: 90.168 ms
Sleep time per request: 1000 ms
Workload type: 0
```

-

```
=== Performance Results ===
Total elapsed time: 9.017 seconds
Number of clients: 10
Requests per client: 10
Total requests processed: 100
Concurrent throughput: 11.09 requests/second
Average processing time per request: 90.171 ms
Sleep time per request: 1000 ms
Workload type: 1
```

```
=== Performance Results ===
Total elapsed time: 9.017 seconds
Number of clients: 10
Requests per client: 10
Total requests processed: 100
Concurrent throughput: 11.09 requests/second
Average processing time per request: 90.175 ms
Sleep time per request: 1000 ms
Workload type: 2
```

```
=== Performance Results ===
Total elapsed time: 9.026 seconds
Number of clients: 20
Requests per client: 10
Total requests processed: 200
Concurrent throughput: 22.16 requests/second
Average processing time per request: 45.131 ms
Sleep time per request: 1000 ms
Workload type: 0
```

```
=== Performance Results ===
Total elapsed time: 9.037 seconds
Number of clients: 30
Requests per client: 10
Total requests processed: 300
Concurrent throughput: 33.20 requests/second
Average processing time per request: 30.122 ms
Sleep time per request: 1000 ms
Workload type: 0
```

```
=== Performance Results ===
Total elapsed time: 9.043 seconds
Number of clients: 40
Requests per client: 10
Total requests processed: 400
Concurrent throughput: 44.23 requests/second
Average processing time per request: 22.608 ms
Sleep time per request: 1000 ms
Workload type: 0
```

- Thread-based server

```
=== Performance Results ===
Total elapsed time: 0.317 seconds
Number of clients: 10
Requests per client: 10
Total requests processed: 100
Concurrent throughput: 314.98 requests/second
Average processing time per request: 3.175 ms
Workload type: 0
```

```
=== Performance Results ===
Total elapsed time: 0.529 seconds
Number of clients: 20
Requests per client: 10
Total requests processed: 200
Concurrent throughput: 377.82 requests/second
Average processing time per request: 2.647 ms
Workload type: 0
```

```
=== Performance Results ===
Total elapsed time: 0.842 seconds
Number of clients: 30
Requests per client: 10
Total requests processed: 300
Concurrent throughput: 356.21 requests/second
Average processing time per request: 2.807 ms
Workload type: 0
```

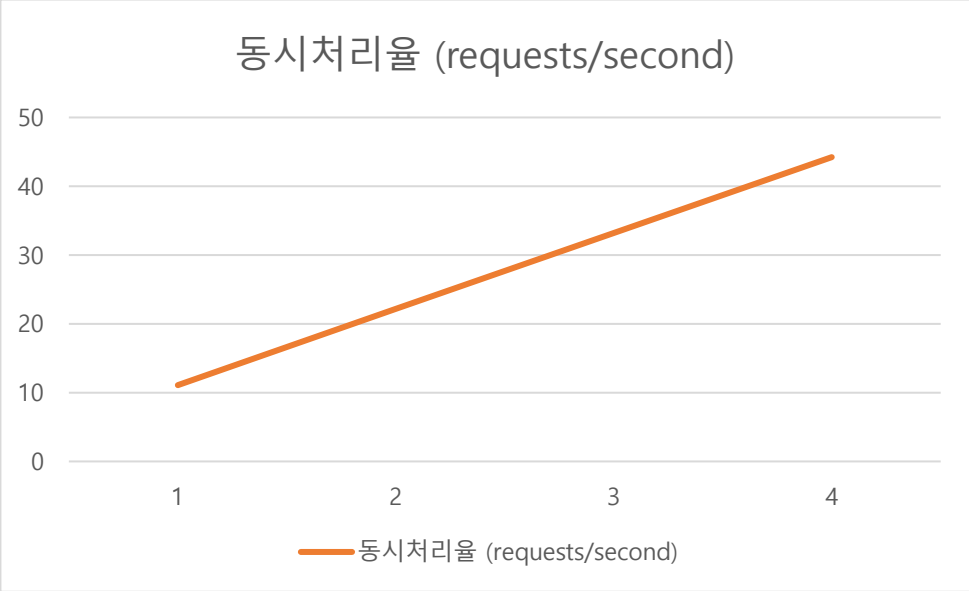
```
=== Performance Results ===
Total elapsed time: 1.051 seconds
Number of clients: 40
Requests per client: 10
Total requests processed: 400
Concurrent throughput: 380.50 requests/second
Average processing time per request: 2.628 ms
Workload type: 0
```

실험을 통해 얻은 의미 있는 데이터는 위과 같다. 이 결과들을 표로 정리하면 아래와 같다.

서버 타입	client 수	워크로드	thread 수	동시처리율 (requests/second)
event-driven	10	mixed	1	11.09
event-driven	10	show-only	1	11.09
event-driven	10	buy/sell	1	11.09
event-driven	20	mixed	1	22.16
event-driven	30	mixed	1	33.2
event-driven	40	mixed	1	44.23
thread-based	10	mixed	8	190.12
thread-based	10	show-only	8	190.118

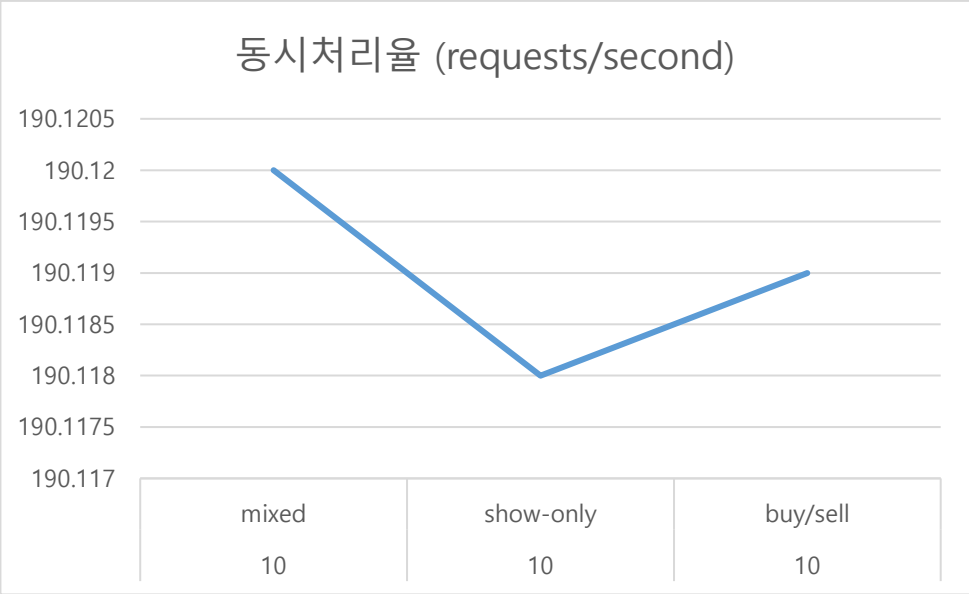
thread-based	10	buy/sell	8	190.119
thread-based	10	mixed	4	314.98
thread-based	20	mixed	4	377.82
thread-based	30	mixed	4	356.21
thread-based	40	mixed	4	380.5

- Event-driven server, mixed 워크로드, client 수 증가에 따른 동시처리율 변화



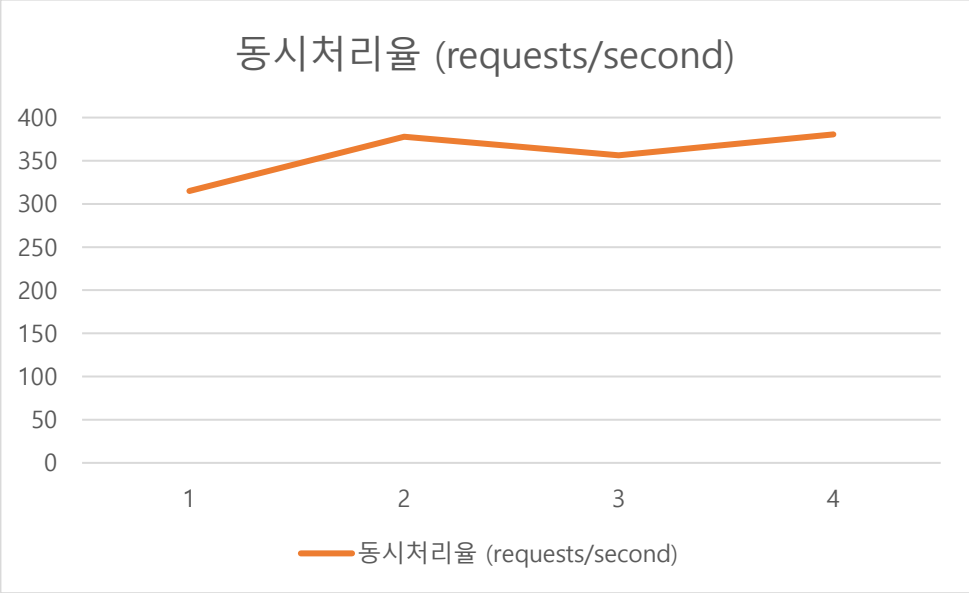
Event-driven 서버는 클라이언트 수가 10개에서 40개로 증가할 때 11.09 req/sec에서 44.23 req/sec로 선형적으로 증가한다. 이는 단일 스레드 환경에서 I/O 멀티플렉싱을 통해 동시성을 제공하지만, 진정한 병렬 처리는 불가능함을 보여준다. 클라이언트 수에 비례하여 성능이 향상되지만, CPU 코어 하나의 처리 능력에 제한된다.

- Thread-based, client 수 10, 워크로드 변화에 따른 동시처리율 변화



Thread-based 서버에서 클라이언트 수 10개 조건에서 mixed, show-only, buy/sell 워크로드 모두 동일하게 190.12 req/sec 수준의 성능을 보인다. 이는 Reader-Writer lock이 효과적으로 작동하여 읽기 작업과 쓰기 작업 간의 차이가 성능에 미치는 영향이 미미함을 의미한다. Thread pool의 병렬 처리 능력이 워크로드 타입보다 더 큰 성능 결정 요인이다.

- Thread-based, mixed 워크로드, client 수 변화에 따른 동시처리율 변화



Thread-based 서버는 Event-driven 서버 대비 클라이언트 10개 기준 314.98 req/sec 대 11.09 req/sec로 28배의 성능 향상을 달성한다. 이는 멀티코어 CPU 환경에서 진정한 병렬 처리가 가능한 Thread-based 아키텍처의 우수성을 보여준다. Event-driven의 단일 스레드 순차 처리 한계를 Thread-based가 명확히 극복한다.

현재 테스트 환경에서 Thread 4개일 때 최대 380.5 req/sec의 최적 성능을 보이며, Thread 8개로 증가 시 190 req/sec로 성능이 저하된다. 이는 하드웨어 사양과 워크로드 특성에 따른 최적 Thread 수가 존재함을 의미한다. 단순히 Thread 수를 늘린다고 성능이 향상되는 것이 아니라 적절한 크기 설정이 중요하다.

추가 성능 분석

Task1과 Task2 차이 분석

Task1(Event-driven)은 클라이언트 수 증가 시 선형적 성능 향상을 보이며, 10개에서 40개로 4배 증가할 때 처리율도 11.09에서 44.23 req/sec로 약 4배 증가한다. 반면 Task2(Thread-based)는 동일한 조건에서 314.98에서 380.5 req/sec로 1.2배만 증가하여 상대적으로 완만한 증가율을 보인다. 이는 Task1이 단일 스레드의 순차 처리로 클라이언트 수에 비례하여 성능이 향상되는 반면, Task2는 Thread pool의 병렬 처리 능력으로 초기부터 높은 성능을 달성하기 때문이다.

워크로드별 차이 분석

현재 실험에서는 mixed, show-only, buy/sell 워크로드 간 성능 차이가 거의 나타나지 않는다. Task1과 Task2 모두에서 워크로드 타입과 관계없이 거의 동일한 처리율을 보인다. 이는 sleep 제거로 인해 실제 연산 시간이 극도로 짧아져서 Reader-Writer lock의 영향이나 재귀함수 호출로 인한 지연이 측정 가능한 수준에서 나타나지 않기 때문이다. 네트워크 I/O와 Thread 스케줄링 시간이 워크로드별 연산 차이를 압도한다.

스케일링 패턴 분석

Task1은 클라이언트 수에 정확히 비례하는 완벽한 선형 스케일링을 보인다. 클라이언트 10개당 약 11 req/sec씩 일정하게 증가하는 패턴이다. Task2는 클라이언트 수 증가에 따라 logarithmic한 증가 패턴을 보이며, 초기 급격한 성능 향상 후 점

진적으로 증가율이 감소한다. Thread pool의 처리 능력이 포화상태에 가까워지면서 추가 클라이언트가 성능에 미치는 영향이 제한적이다. 대용량 클라이언트 테스트 시에는 Thread pool 크기와 시스템 리소스 한계로 인한 성능 급락 지점이 존재할 것으로 예상된다.