System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 조다민

학번 : 20181297

1. 개발 목표

프로젝트에서 우리는 Concurrent stock server를 구현하여 여러 클라이언트의 동시 접속과 서비스를 제공하는 것을 목표로 한다. 서버에서는 주식에 대한 정보를 저장하고 있는 stock.txt에서 데이터를 받아 binary tree 구조로 저장하고, 클라이언트의 요청에 따라 주식 정보를 출력하거나(show) 사고(buy) 팔(sell) 수 있다.

Concurrent stock server는 1. Event-dirven Approach와 2. Thread-based Approach 두가지 방식 모두를 이용해 구현하고, 마지막에는 두 구현 방식의 성능 차이까지 살펴볼 수 있도록 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   Event-driven Approach에서는 select 함수를 기반으로 프로그램을 작성하였다. 초기 pool 설정을 완료한 뒤, 무한 loop 내부에서 새로운 client와의 연결과 client의 요청을 처리할 수 있도록 check_clients 함수를 호출해 stockclient나 multiclient를 통해 전달되어 지는 client의 show, buy, sell 명령어를 처리하고 해당 결과를 client에게 전달, 출력할 수 있도록 한다.

2. Task 2: Thread-based Approach

   1에서와 비슷하게 먼저 sbuf에 대한 초기 설정을 완료하고 여러 클라이언트의 요청을 동시에 처리하기 위해서 thread를 먼저 생성(pthread_create) 후 무한 loop 내부에서 client와의 연결, discriptor를 sbuf에 삽입한다. 그리고 thread함수 내부에서 stockClient 함수를 이용해서 client에서 넘어온 show, buy, sell 명령어를 처리하고 해당 결과를 client에게 전달, 출력할 수 있도록 한다.

3. Task 3: Performance Evaluation

   걸리는 시간을 측정할 수 있도록 multiclient.c에 instruction을 추가하여 test

를 진행하였다. 또한 multiclient.c에서 option의 번호를 조절해 명령어 별 실행 시간 또한 측정할 수 있었다.

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
  - ✓ epoll과의 차이점 서술
- **Task2 (Thread-based Approach with pthread)**
  - ✓ Master Thread의 Connection 관리
  - ✓ Worker Thread Pool 관리하는 부분에 대해 서술
- **Task3 (Performance Evaluation)**
  - ✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

    측정은 multiclient.c에서 while문 전부터 multiclient를 종료하는 시점까지의 시간을 측정한다.

    1) Client 개수 변화에 따른 동시 처리율

       multiclient를 실행하면서 주는 client 수만을 변화시키며 task1과 task2의 (client*ORDER_PER_CLIENT)/(elapsed time)을 비교한다. 이를 통해 task1,2의 client 개수 증가에 따른 동시처리율 비교를 통해 동시처리율이 좋은 방식은 무엇인지 확인할 수 있을 것이다. Client 수가 10, 20, 30, 40, 50 일 때, ORDER_PER_CLIENT = 20으로 고정하여 시간을 측정하고 엑셀을 이용해 결과를 비교, 분석한다.

    2) Client 요청 타입에 따른 동시 처리율

       multiclient에서 랜덤하게 정해지는 명령어(show, buy, sell)을 제한하여, 한 가지의 명령어만 실행했을 때, 두 가지 명령어의 조합으로 실행할

때, 세 가지 명령어의 조합으로 실행할 때의 동시처리율을 비교한다 .

3) Task2에서 thread 수에 따른 동시 처리율

Task2에서 NTHREAD의 수를 1, 2, 4, 8, 16, 32로 바꿔가며 실행시간을 측정하고, 1)에서와 같이 (client*ORDER_PER_CLIENT)/(elapsed time)를 비교한다. 이를 통해 thread 수에 따른 효율성을 확인할 수 있다.

✓ Configuration 변화에 따른 예상 결과 서술

Event-driven 방식이 control flow를 하나로 실행하기 때문에 control overhead가 없어 더 빠른 동시 처리율을 가질 것으로 예상된다. 또한 show 명령어의 경우 모든 node를 돌면서 출력 멘트를 정하는데, 그 동안 해당 node에 접근을 할 수 없을 것이다. 따라서 show가 다른 명령어보다 시간이 더 오래 걸리고 event-driven과 thread 방식을 비교해봤을 때 thread 방식에서 show를 하는 동안 multi process를 사용할 수 없어 그 실행시간 차이가 더 크게 날 것이라고 생각한다.

C. 개발 방법

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

3. 구현 결과

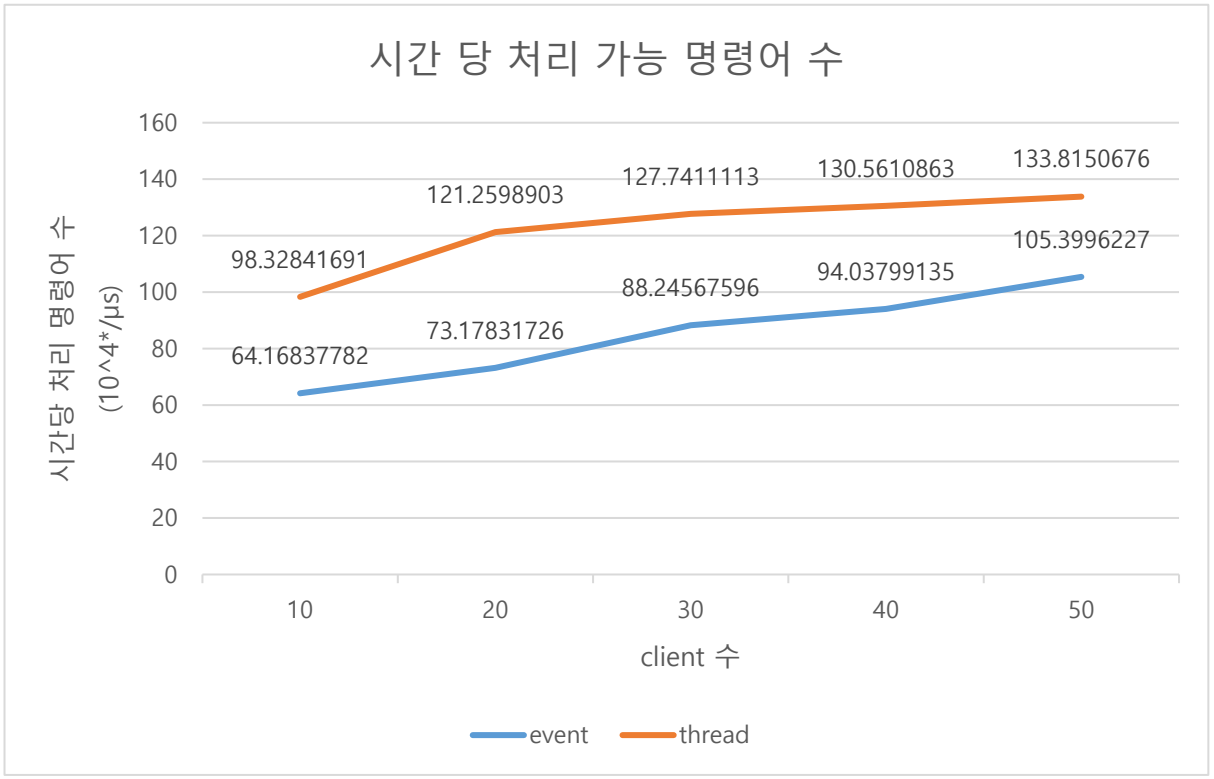
4. 성능 평가 결과 (Task 3)

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

1) Client 개수 변화에 따른 동시 처리율

Table 1. 실행시간 표

Client 수	Event	Thread
10	elapsed time: 31168 microseconds	elapsed time: 20340 microseconds
20	elapsed time: 54661 microseconds	elapsed time: 32987 microseconds
30	elapsed time: 67992 microseconds	elapsed time: 46970 microseconds
40	elapsed time: 85072 microseconds	elapsed time: 61274 microseconds
50	elapsed time: 94877 microseconds	elapsed time: 74730 microseconds

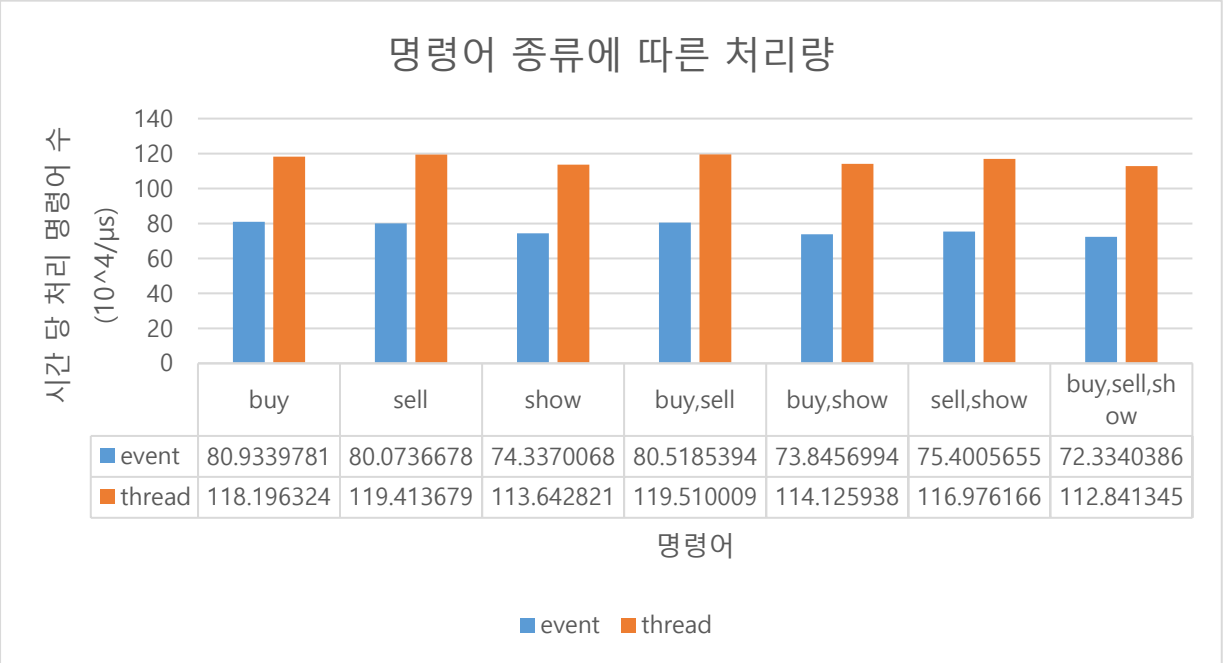


비교를 편하게 하기 위해 결과 값에 10^4를 곱했다. Overhead의 유무가 결과에 영향을 크게 줄 것이라고 예상한 것과는 다르게, thread의 multithreading에 의한 시간 절약이 더 큰 영향을 주어, thread-based 방식이 더 큰 시간 당 명령어 처리 수를 보여주었다.

또한 해당 그래프에서 볼 수 있는 사실로는, thread 방식의 개선 한계점을 들 수 있는데, thread-based 방식의 경우 client 수가 20에서 30으로 넘어감과 동시에 처리량 증가에 대한 기울기가 급격히 감소하는 것을 확인할 수 있다. 하지만 event-driven 방식의 경우 client 수가 증가함에 따라 거의 일정하게 처리량에 대한 기울기가 유지되는 것을 확인할 수 있다.

2) Client 요청 타입에 따른 동시 처리율

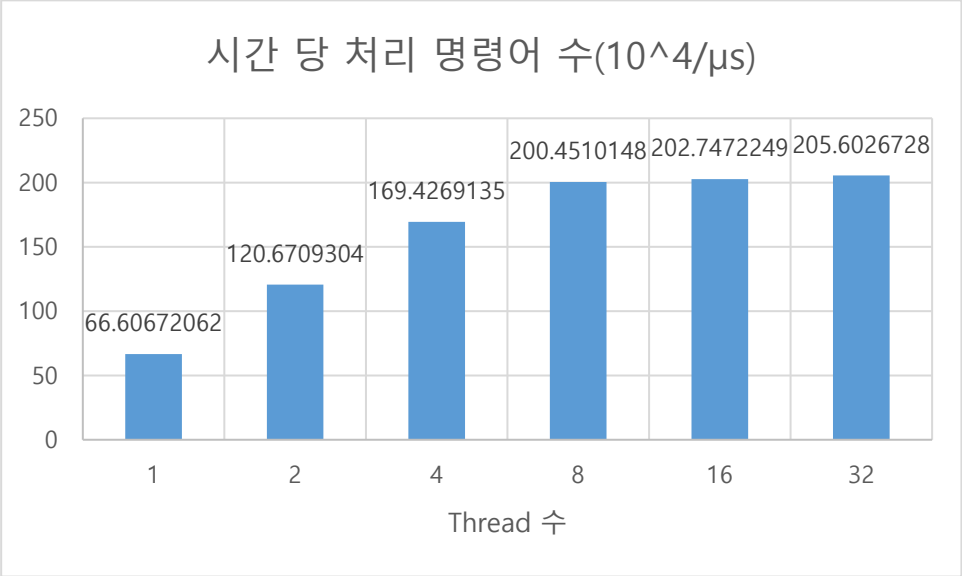
명령어 종류	Event	Thread
buy	49423	33842
sell	49954	33497
show	53809	35198
buy,sell	49678	33470
buy,show	54167	35049
sell,show	53050	34195
buy,sell,show	55299	35448



걸린 시간과 처리량을 봤을 때, show가 들어간 경우에 모든 node를 순회하며 print를 하기 때문에 시간이 더 많이 걸려 처리량도 줄었음을 알 수 있다.

3) Thread 수에 따른 동시 처리율 비교

thread 수	실행 시간(μs)
1	60054
2	33148
4	23609
8	19955
16	19729
32	19455



Thread 수에 따른 동시 처리율을 확인하기 위해 thread 수를 변화 시키며 살펴보았을 때, 8까지는 그 수가 증가하지만, 16, 32로 가면서 더 이상 유의미하게 증가하지 않는 것을 확인할 수 있다. 또한 그 보다 thread 수가 작은 경우에도 살펴보았을 때, thread 수가 2배가 되었음에도, 그 처리 명령어 수는 2배로 늘지 않음을 확인할 수 있는데, 이는 thread간 scheduling나 data sharing을 막기 위해 한 thread에서 데이터에 접근 시 다른 thread에서의 접근을 막는 등의 overhead가 발생하기 때문으로 이해할 수 있다.