Multicore Programming Project 2

담당 교수 : 박성용

이름 : 이건화

학번 : 20191616

1. 개발 목표

- **해당 프로젝트에서 구현할 내용을 간략히 서술.**
- 주식 서버를 구현한다. 주식 서버는 현재 주식 시장에 있는 주식들의 정보들을 가지고 있다. 서버는 클라이언트들의 송신한 요청에 따라 정보들을 수정하거나 제공해줄 수 있어야 한다. 클라이언트가 할 수 있는 요청에는 show, buy, sell이 있다. show는 주식정보들을 보여달라는 요청이고, buy는 원하는 ID의 주식을 몇 개 살 것인지를 알려주는 요청이다. sell은 클라이언트가 팔 주식의 ID와 몇 개를 팔 것인 지를 알려주는 요청이다. 위와 같은 주식의 기능의 구현도 중요하지만 주식 서버의 핵심이라함은 서버가 여러 클라이언트의 요청에 대해 어떻게 처리할 것인 지를 생각해보고 공부해본 후 구현하는 것이다. 여러 클라이언트의 요청을 처리하는 방법 두가지를 구현해보고 다양한 상황에서 실행해보면서 두 방법을 비교분석해본다.
- **(주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)**

  주식 정보는 binary tree로 저장하여 검색의 효율을 높인다. buy와 sell은 우선 거래를 할 주식 노드를 search함수로 찾는다. 찾은 노드에 대한 거래 처리를 완료하고 해당 거래가 성공적이었음을 알려주는 메시지를 클라이언트에게 보낸다. show의 요청을 받았을 때는 주식 노드를 preorder로 순회하면서 주식 노드의 정보를 buf에 저장하여 순회가 끝나면 해당 buf를 client에게 보낸다. exit 요청이 들어오면 클라이언트와의 연결을 종료한다.

  여러 클라이언트의 요청을 두 가지 방법으로 처리해보았다. 첫 번째 방법은 event-driven 방식으로 Select함수를 통해 요청이 들어온 fd를 파악 후, 해당 요청을 순차적으로 처리하는 방식이다. listenfd에서 요청이 들어왔을 경우 새로운 연결 요청이 들어온 것이므로 새로운 연결을 처리하고, 이외의 clientfd에서 요청이 들어왔을 경우 들어온 요청에 따라 주식 정보를 처리한 후, 해당 client에게 보낸다.

  두 번 째 thread-based 방식은 thread들을 미리 만들어놓은 후, 새로운 연결이 생겼을 경우 해당 연결에 대한 정보를 sbuf에 sbuf_insert를 통해 넣는다. 각 thread들은 sbuf에서 새로운 연결이 들어올 경우 sbuf_remove를 통해 해당 연결에 정보를 가져온 후 처리를 수행한다. 즉, 각 thread가 client 하나를 맡은 후 client의 요청을 독립적으로 병렬 처리하는 것이다. 병렬 처리에 따른 데이터 접

근 충돌을 제어하는 것 또한 구현했다. show와 같은 read 요청에 대한 처리는 buy와 sell 같은 write 요청보다 우선순위를 둬 더 빠르게 처리할 수 있도록 하였다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   ■ event-driven 방식을 구현하기 전에는 먼저 들어온 client에 대한 처리를 완료하고 연결을 종료한 후에 다음 client의 연결 요청을 받고 처리할 수 있었다. 굉장히 비효율적이다. 늦게 연결을 요청한 client 같은 경우 이전 client가 연결을 종료할 때까지 기다려야했고, 그 다음 client는 더 많은 시간을 기다려야 했다. Select 함수를 통해 요청이 들어온 fd를 파악한 후, 순회하면서 요청이 들어온 모든 fd의 요청을 차례대로 수행한다. 이로 인해 새로운 client의 연결 요청을 받을 수 있었고, 여러 client가 요청을 보내도 차례대로 모든 client의 요청을 처리할 수 있었다. 즉 나중에 연결된 client가 먼저 들어온 client의 연결 종료를 기다릴 필요가 없어졌다.

2. Task 2: Thread-based Approach

   ■ thread-based 방식을 구현하기 전에는 먼저 들어온 client에 대한 처리가 완료될 때까지 다음 client는 요청을 보내도 server에서 처리를 해줄 수가 없었다. 먼저 들어온 client의 요청을 기다리고 있었기 때문이다. thread-based 방식은 client 하나 당 하나의 thread가 붙어 각 thread가 하나의 client의 요청을 처리한다. 새로운 요청이 들어오면 담당하고 있는 client가 없는 thread가 새로운 client를 받아서 해당 client에 대한 처리를 진행한다. client의 연결이 끊어지면 client와의 연결을 종료하고 새로운 client를 기다린다. 이와 같은 방법으로 여러 client가 연결을 요청하고 각자 요청을 보내도 concurrent하게 처리할 수 있었다. thread-based 방식에서는 특히 여러 thread가 하나의 데이터에 동시 접근하여 충돌을 일으킬 수 있었는데 이 부분에 대해서는 mutex를 사용하여 해당

정보를 처리하고 있는 thread가 없을 때만 접근할 수 있게 하였다. 추가로 read하는 요청을 write하는 요청보다 우선시하여 효율적으로 요청을 처리할 수 있게하였다. 이는 w를 사용하여 구현하였다.

3. Task 3: Performance Evaluation

   ■ 클라이언트의 수가 적을 때는 event-driven과 thread-based의 속도 차이는 거의 없었다. 대개 thread-based가 조금 더 빨랐지만 event-driven이 더 빠른 경우도 적지 않았다. 그러나 클라이언트의 수가 1000명 정도로 많을 때에는 thread-based가 더 빨랐다. 추가로 요청하는 동작의 종류에 따라 thread-based 방식은 속도차이를 보였다. 이는 thread의 read작업끼리 동시 데이터 접근할 때는 충돌을 피하기 위해 대기할 필요가 없기 때문이다. 그러나 다른 경우 대기해야했기 때문에 더 느린 것을 확인할 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

  ◆ Select함수를 사용하여 구현하였다. pool이라는 개념이 존재한다. pool이란 여러 fd에 요청이 들어왔을 때 해당 fd에 요청이 들어왔음을 나타내주기 위해 사용한다. 이를 위해 FD_SET을 사용한다. FD_SET은 하나의 fd의 상태를 하나의 비트로 표현한다. 연결된 fd의 bit의 값을 1로하여 Select함수에 넣어주면 입력받을 준비가 된 fd의 값만 1로하여 return한다. 이를 통해 요청이 들어온 fd를 파악할 수 있다.

  ◆ 이후 요청이 들어온 fd가 listenfd라면 새로운 연결을 pool의에 추가한다. listenfd가 아닌 fd에 들어온 요청들은 이미 연결된 clientfd가 주식 정보에 관한 처리를 요청한 것이다. 해당 요청들을 checkclients함수를 통해 순회하며 처리한다. client가 보낸 요청은 Rio_readline에서 rio를 통해 들어온 데이터를 buf로 받는다.

- ◆ 처리를 완료한 후 결과값을 client한테 보낼 때는 rio_writen함수를 사용하는데 rio_writen함수는 입력으로 들어온 n만큼의 데이터를 모두 성공적으로 보낼 때까지 전송을 계속 시도한다. 이로 인해 안정적으로 데이터를 전송 할 수 있다. client함수로부터 데이터를 받아올 때에는 Rio_readlineb 함수를 사용하는데 해당 client에 연결된 rio를 input으로 받아 rio로부터 데이터를 n만큼 읽어 buf에 저장한다. n만큼의 데이터가 성공적으로 들어올 때까지 계속 대기한다.

✓ epoll과의 차이점 서술

- ◆ epoll함수는 select함수보다 나중에 개발된 함수여서 select의 단점이 보완된 부분이 있다. select함수는 관찰 영역에 대하여 bit별로 event가 발생했는지 검사해야한다. 또한, fd가 1024개로 제한되어있다. 그러나 epoll 함수는 반복문을 통해 관찰영역에 있는 디스크립터를 모두 살펴봐야하는 불필요한 동작을 할 필요가 없다. epoll함수는 이벤트가 발생한 fd와 이벤트의 종류를 반환한다. 이로 인해 이벤트가 발생한 꼭 필요한 event만 순회하면서 처리할 수 있다. select함수가 epoll보다 불필요한 순회를 해야한다는 단점이 있지만 많은 운영체제에서 호환이 가능하다는 장점이 있다.

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

✓ Master Thread의 Connection 관리

- ◆ master thread는 client들의 요청을 처리해줄 peer thread를 먼저 많이 만들어 놓는다. 그리고 sbuf에 새로 들어온 연결에 대한 정보를 삽입한다. 새로운 연결이 들어오면 하나의 peer thread가 해당 연결을 담당하여 처리한다. 연결이 끝나면 다시 새로운 연결을 가져가서 처리한다. master thread는 accept함수를 통해 listenfd로부터 새로운 connection을 받아서 sbuf에 넣는 역할만 한다. peer thread는 새로 생긴 연결을 sbuf로부터 가져와서 thread하나 당 하나의 client를 처리한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

- ◆ worker thead pool은 병렬 작업 처리 비용을 줄이고, 동시성을 효율적으로 처리하는 것을 목적으로 둔다. 이를 위해서 main thread에서는 peer thread를 많이 생성해놓는다. 해당 thread는 하나의 client를 담당하여 처리한다. main thread는 listenfd를 통해 새로운 연결 요청이 들어오면 새

로운 연결 요청에 대한 정보를 sbuf에 넣는다. thread는 sbuf에서 하나씩 요청을 받아 연결이 끝날 때까지 해당 client의 요청을 처리한다. 이를 통해 thread를 불필요하게 생성하고 삭제할 필요가 없다. 노드별로 mutex 변수를 넣어 하나의 노드에 동시 접근하는 thread 간 데이터 충돌이 발생하지 않게 한다. 또한, 해당 server는 first read-writer를 기반으로 하는데 write처리를 할 때 read를 할 수 없으므로 read요청을 우선 처리 후, write를 하는 것이다. read 처리는 동시에 두 개 이상이 접근해도 충돌이 발생하지 않기 때문이다. 이를 통해 동시성을 효율적으로 처리할 수 있다. 이를 구현하기 위해서 노드별로 w라는 변수를 하나 더 추가하였다.

- **Task3 (Performance Evaluation)**

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

  ◆ 프로세스의 걸리는 시간을 측정하였다. 동시처리율은 (클라이언트 수 X 클라이언트 별 요청 개수)/(걸리는 시간)인데 클라이언트 수와 클라이언트 별 요청 개수가 같기 때문에 시간을 비교하면 동시처리율을 비교하는 것과 같이 성능을 비교할 수 있다0. 따라서 같은 요청의 양을 더 짧게 처리하는 것은 성능이 더 좋다고 판단할 수 있다. 이로 인해 performance evaluation에서 시간이 가장 직관적이고, 성능을 판단하기에 간편하고 명확한 지표다. 이를 측정하기 위해 sys/time.h의 gettimeofday 함수를 사용하였다. 시작시간과 종료시간을 측정한 후, 둘의 차를 계산하여 종료 직전 출력하게 했다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  ◆ event-driven 에는 반복문도 있기 때문에 클라이언트의 수가 적을 때에도 thread-based 가 조금 더 빠를 것이라고 예상했다. 클라이언트의 수가 많아질수록 thread-based 가 event-driven 방식의 속도 차이는 더 커질 것이라고 생각했다. thread-based 는 각 thread 가 client 를 병렬적으로 처리하는 반면 event-driven 은 sequential 하게 처리하기 때문이다. 이로 인해 client 의 수가 많아질수록 병렬처리하는 thread-based 의 속도가 느려지는 정도는 event-driven 보다 작을 것이라고 생각했다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할**

것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)

- 우선 task_1과 task_2 모두 주식서버이므로 공통적으로 buy, sell, show 요청을 처리할 수 있어야 한다. 위의 기능을 구현하기 위해 주식 정보는 구조체 stock에 담았다. buy와 sell은 client가 처리할 주식을 해당 주식의 ID를 사용하여 search함수를 통해서 찾는다. search를 효율적으로 사용할 수 있게 주식정보는 binary tree로 담겨있다. 찾은 함수를 처리 후 해당 요청에 대한 처리의 성공여부를 client에게 보낸다. show요청을 처리할 때는 stock_to_buf함수를 사용하여 binary tree에 있는 모든 노드들을 preorder로 순회하며 주식정보들을 차례대로 buf에 담는다. buf를 client에게 전송해 show의 요청을 처리한다. read_stock 과 insert함수는 stock.txt에 있는 주식정보들을 읽어 tree구조로 저장한다. INT_handler는 server가 SIGINT에 의해 종료되었을 때 table에 있는 stock 정보를 stock.txt에 update한다.
- task1에는 pool구조체를 사용하는데 pool구조체는 event가 발생한 fd들을 저장하기 위해 사용된다. 이를 통해 select함수에서 fd별로 event가 발생했는지 모니터링할 수 있다. listenfd를 통해 새로운 연결 요청이 들어올 경우 addclients를 통해 pool에 새로 연결된 fd를 추가한다. 이 외의 clientfd에서 요청이 들어왔을 경우, checkclients 함수를 통해 관찰영역에 있는 fd들을 살피며 event가 발생했다면 해당 event에 맞는 요청을 순회하며 처리해준다.
- task2에는 추가로 sbuf 구조체를 사용하는데 이는 새로 연결된 clientfd가 저장될 queue를 가지고 있다. listenfd를 통해 새로운 client의 연결 요청이 들어오면 sbuf_insert를 통해 queue에 새로운 연결이 들어간다. thread는 sbuf_remove를 통해 새로 들어온 연결에 대해 처리한다. 여러 thread가 sbuf에 동시 접근하여 충돌이 발생하는 것을 방지하기 위해 sbuf에도 mutex가 있다. 추가로 INThandler에서 sbuf에 대한 메모리를 해제해준다.
- task3를 위해서는 gettime함수를 만든다. gettime함수는 gettimeofDay함수를 사용해서 시간을 측정하고 시간을 return한다. 새로 만든 gettime함수를 통해 시작시간과 종료시간을 측정한 후 둘의 차를 종료직전 출력하게 한다.

3. 구현 결과

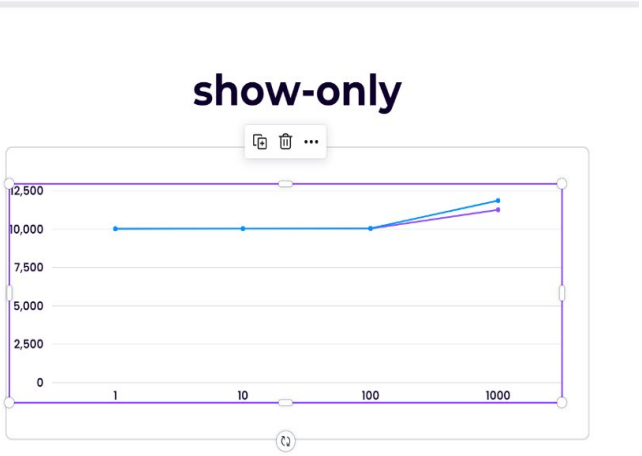
- **2번의 구현 결과를 간략하게 작성**
- 2번의 구현이 완료되면, task1과 task2모두 여러명의 client의 요청을 처리할 수 있게 된다. task2 경우 first reader-writer기능을 지원하므로 show가 많을수록 더 빠르다. task1과 task2 모두 sell, buy, show 기능을 처리할 수 있게 된다. server를 종료하면 해당 정보들이 stock.txt에 반영되는 것을 확인할 수 있다.
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**
- 명령어의 입력에 따른 예외처리를 구현할 수 있다. 예를 들어, buy 1 2는 ID가 1인 주식 2개를 사겠다는 요청이다. buy 1 2도 buy 1 2와 똑같은 요청임을 유추할 수 있지만 에러가 발생한다. 위와 같이 공백의 개수 차이 등 형식에 완전히 부합하지 않는다면 값을 제대로 입력받지 못하여 에러가 발생하게 된다. 이와 같은 경우를 해결하기 위해서는 입력받은 요청을 우선 전처리 해주는 parsing 부분을 구현한다면 해결될 것이다. 추가로 stock.txt에 없는 ID의 값이 들어오거나 stock.txt의 파일에 형식이 맞지 않은 파일이 있다면 readstock()부분에서 에러가 발생하거나 search함수에서 에러가 날 수 있기 때문에 이의 예외처리를 구현하기 위해서는 stock.txt를 read할 때 readstock()함수 안에서 전처리 과정이나 형식을 엄격하게 확인하는 부분을 구현한다면 안정적으로 주식테이블을 관리할 수 있다..

4. 성능 평가 결과 (Task 3)

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

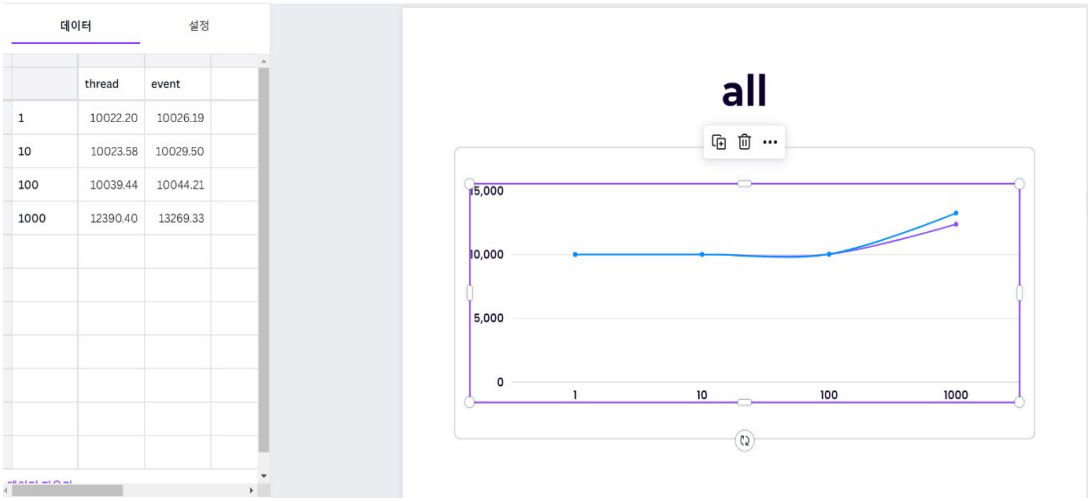
데이터 설정

	thread	event
1	10020.37	10019.65
10	10034.31	10031.52
100	10038.95	10052.42
1000	11259.06	11862.06



- read만 했을 때의 client별로 속도를 측정한 것이다. client가 1개만 있을 때는 event가 더 빨랐다. client가 10개만 있을 때까지도 event가 미세하게 더 빠르다. 그러나 client의 개수가 100개가 되자 event-driven 방식보다 thead-based방식이 지연정도가 작아서 thread-based방식이 더 빠른 것을 확인할 수 있었다. 1000명의 client가 요청을 보냈을 때는 그래프에서 육안으로 구분이 가능할 정도의 차이가 발생하였다. client가 적을 때는 두 방식의 속도 차이가 거의 없지만, client가 많을 때는 thread-based의 각 thread가 client를 병렬적으로 처리함에 따라 sequential하게 처리하는 event-based 방식보다 더 적게 느려지는 것을 확인할 수 있다.

- write만 했을 때의 분석이다. read만 했을 때처럼 client의 개수가 적을 때는 두 방식의 속도차이는 미세하거나 누가 더 빠르다고 말하기에는 다른 변수의 영향이 더 컸다. 실제로 여러 번 실험해보았는데 어떨 때는 event가 빠르고 어떨 때는 thread 방식이 빨랐다. 그러나 client의 개수가 1000개가 되자 thread-based방식이 event-driven방식보다 더 빠른 것을 그래프에서 육안으로 확인할 수 있을 만큼 차이가 났다. 추가로 thread-based 방식에서 write만하는 경우 read만하는 것보다 각 thread가 충돌을 피하기 위해 대기하는 시간이 있어서 read하는 경우보다 느려진 것을 확인할 수 있었다. event-driven 방식도 write만 하는 경우, 더 느려지는 것을 확인할 수 있었다. 이는 search와 update를 모두 해야하는 write의 요청 처리시간이 search만하는 read의 경우보다 더 걸리기 때문으로 보인다.



- read와 write를 모두 하는 경우를 분석해보았다. 클라이언트의 개수가 적을 때는 위와 같이 큰차이가 없었다 그러나 1000명의 클라이언트의 요청을 처리할 때부터는 위의 두 실험과 마찬가지로 event-driven보다 thread-based방식이 더 빨랐다. thread-based방식은 각 thread가 병렬적으로 client를 처리하기 때문에 직렬적으로 처리하는 event-driven방식보다 더 빠른 것을 확인할 수 있었다. thread-based방식에서 read만 했을 경우 보다 read와 write를 같이 하니 더 느린 것을 확인할 수 있었다. 이는 read-write, write-write, write-write와 같이 일이 배열되어 있을 때 충돌을 피하기 위해서 대기해야하는 시간이 있기 때문이다. 그러나 write만 하는 경우보다는 충돌이 덜 발생한다. 왜냐하면 read-read와 같은 경우에는 동시에 접근해도 충돌이 발생하지 않기 때문이다. 그래서 read-write하는 경우가 충돌하는 시간을 피하기 위해 대기하는 시간이 write만 하는 경우보다 조금 더 적어서 최종 처리시간이 덜 걸린 것을 확인할 수 있었다.

위의 실험 결과를 통해 client의 개수가 적을 때는 둘의 차이가 미세하다는 것을 확인할 수 있다. client의 개수가 적을 때는 방식이 성능에 미치는 영향이 적어서 어떤 방식이 항상 더 빠르다고 말하기 애매했다. 그러나 client의 개수가 1000정도로 커지자 대개 thread-based방식이 더 빨랐다. 이는 thread-based 방식은 많은 client들을 concurrent하게 처리하지만 event-driven방식은 많은 client들을 sequential하게 처리하기 때문이다. thread-based방식에서 read만 했을 경우가 가장 빨랐고, 그 다음 read와 write하는 경우가 빨랐다. write만 하는 경우가 가장 느렸는데 이는 read와 read가 동시에 처리될 경우 충돌이 없지만 read-write, write-write, write-read경우 모두 충돌이 발생해서 충돌을 피하기 위해 대기해야하기 때문이다. 따라서 write만하는 경우에 가장 충돌이 많이 발생한다.