System Programming Project 2

담당 교수 : 김영재

이름 : 정지석

학번 : 20181687

1. 개발 목표

해당 프로젝트는 여러 client들이 동시에 접속할 수 있는 concurrent stock server를 event-based와 thread-based, 이 두가지 방법으로 구현하는 것이 주된 목표이다.

Stock.txt에 들어온 주식 데이터들은 binary tree를 통해 관리한다. Client가 server에게 요청할 수 있는 명령은 show, buy, sell, exit가 있다.

모든 client들의 요청을 처리하고 종료가 되면, 바뀐 주식 정보를 stock.txt에 다시 저장한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Server와 client를 실행한 뒤 show, buy, sell, exit 4개의 명령어를 실행하면 그에 해당하는 결과가 화면에 제대로 출력되는 것을 확인할 수 있다. 또한 명령어 실행이 모두 끝난 뒤 client를 종료하면, 명령어를 수행한 결과가 stock.txt에 저장되는 것 또한 확인할 수 있었다.

   Multiclient인 경우에도 client가 하나인 경우와 동일하게 문제없이 작동하는 것을 확인할 수 있었다.

2. Task 2: Thread-based Approach

   위의 경우와 같이 마찬가지로, Server와 client를 실행한 뒤 show, buy, sell, exit 4개의 명령어를 실행하면 그에 해당하는 결과가 화면에 제대로 출력되는 것을 확인할 수 있다. 또한 명령어 실행이 모두 끝난 뒤 client를 종료하면, 명령어를 수행한 결과가 stock.txt에 저장되는 것 또한 확인할 수 있었다.

   Multiclient인 경우에도 client가 하나인 경우와 동일하게 문제없이 작동하는 것을 확인할 수 있었다.

3. Task 3: Performance Evaluation

   Event-based server에 비해 thread-based server의 실행 속도가 전체적으로 빠르다는 사실을 확인할 수 있었다. 이는 client의 개수가 늘어날수록 더욱 극명하게 드러났다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  I/O Multiplexing은 하나의 process가 Multiclient의 요청을 빠르게 처리하여 마치 여러 process가 concurrent하게 실행되는 것처럼 보이게 해준다.

  이를 실제로 구현하기 위해서는 select 함수를 사용하여 하나 혹은 여러 개의 client에서 요청이 들어왔을 때 이를 돌아가면서 처리한다.

  ✓ epoll과의 차이점 서술

  select의 경우 ready_set에 read_set의 값을 받은 뒤, 함수를 실행하여 어떤 connfd를 통해 pending input이 들어왔는지를 확인하여 입력을 처리한다. 그런데 이때 어느 fd에 들어왔는지를 확인해야 하기 때문에, 이를 반복문을 사용한다. 이렇게 될 경우 overhead가 증가한다는 문제가 생기게 된다.

  그리고 이것을 해결하기 위해 나온 것이 바로 epoll이다. Epoll의 경우 일단 전체 fd를 확인하기 위한 반복문을 사용하지 않기 때문에 불필요한 overhead가 발생하지 않는다. 대신 운영체제가 직접 이를 확인하는 식으로 input을 처리한다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 client와의 connection을 받아들여 connfd를 얻는다. 이렇게 얻은 connfd를 buffer에 저장하여 미리 만들어 놓은 thread들과 echoing을 해준다.

  Sbuf_t type의 구조체, sbuf를 선언해준 뒤 client로부터 넘겨 받은 connfd를 sbuf에 insert 해준다. 이렇게 connfd를 sbuf(buffer)에 넣어주는 것을 통해 client와의 connection을 관리하는 것이 가능해진다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  이 경우 worker thread들이 client들의 input을 처리한 후 종료되게 되면, 해

당 thread를 Pthread_detach 함수를 사용하여 kernel이 reaping을 하게 해준다.

- **Task3 (Performance Evaluation)**

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

  Client 개수의 경우 100개 단위로 나누어 100-500 까지의 범위 내에서 측정을 진행할 것이다. Client 개수에 따라 이를 처리하는 데에 걸리는 시간을 구하고, 이를 통해 성능을 측정한다.

  이는 즉, 실행 시간이 짧을수록 더 성능이 좋다는 의미이다. 보통 프로그램의 성능을 측정하는 가장 좋은 방법이 바로 running time 측정이기에 이를 기준으로 하였다.

  Multiclient.c를 통해 여러 개의 client의 요청을 받을 수 있으므로 이를 활용하면 프로그램의 성능을 측정할 수 있다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  Buy, sell 명령어의 경우 실행에 있어 큰 차이가 없기 때문에 비슷한 결과를 낼 것으로 예상된다.

  하지만 show의 경우 stock.txt의 모든 내용을 출력하는 명령어 이므로, buy와 sell에 비해 실행 시간이 더욱 오래 걸릴 것으로 예상된다.

  3개가 섞여 있는 경우에는 위의 명령어 중 실행시간이 가장 오래 걸리는 show의 개수가 얼마인지에 따라 실행 시간이 영향을 받을 것으로 예상된다.

  또한 하나의 process에서 명령어를 처리하는 event-based server 보다는 thread-based server의 실행 시간이 더 짧을 것으로 예상된다.

C. 개발 방법

일단 event-based, thread-based 둘 모두 stock.txt 의 data 를 관리하는 데에는 binary tree 를 사용하였다. 각 주식 종목에 대한 정보가 들어있는 struct 를 item 이라 정의한 뒤, server 가 실행되었을 때 stock.txt 의 정보를 읽어 insert 함수를 통해 읽어 들인 정보를 binary tree 에 저장하였다. 또한 buy 나 sell 이 실행 되면 binary tree 에서 사거나 판 주식의 id 를 찾아 값을 변경해 주었다. Show 명령어를 실행하거나 client 를 종료할 경우에는 print

함수를 통해 binary tree 의 모든 정보를 저장한 뒤, 이를 출력하거나(show), stock.txt 에 저장하였다(종료된 경우).

*event-based

Pool struct 를 선언하여 여러 client 와의 connection 을 관리하는 것이 기본적인 구조이다. Pool 을 관리하기 위해 init_pool 함수를 통해 pool 을 설정해주고, add_client 함수를 통해 client 와의 connection 을 pool 에 추가해준다.

또한 check_client 함수를 통해 show, buy, sell, exit 과 같은 명령어들을 처리하는 식으로 event-based server 를 구현하였다.

*thread-based

먼저 sbuf 라는 struct 를 선언하여 connfd 를 관리하는 것이 기본적인 구조이다. Pthread_create 함수를 실행하여 일단 thread 들을 미리 만든 후, client 의 connection 이 들어왔을 때에 이를 연결해준다.

Buffer 에 들어온 connfd 의 삽입/삭제를 담당하는 sbuf_insert/sbuf_remove 함수의 경우 semaphore 를 사용하여 thread 간의 충돌을 방지한다.

Echo_cnt 함수에서는 명령어로 들어온 show, buy, sell, exit 명령어를 실행하며, 여기세서도 semaphore 를 사용하여 thread 간의 충돌을 방지한다.

3. 구현 결과

2번을 모두 구현한 결과, event-based, thread-based server에서 모두 show, buy, sell, exit 명령어가 문제 없이 실행되는 것을 확인할 수 있었다. 또한 모든 실행이 종료되면 명령어 실행 결과가 stock.txt에 모두 저장된 것도 확인할 수 있었다.

또한 multiclient.c를 통해 여러 개의 client에게서 입력을 받아 성능을 평가하였는데, 2번에서 예상했던 대로 thread-based server의 실행 속도가 더

빠른 것을 확인할 수 있었다.

하지만 위에서 예상했던 것과 달리 show 명령어의 실행시간이 오히려 buy, sell 명령어의 실행시간 보다 짧은 것을 확인할 수 있었다.

성능 평가 결과 (Task 3)

(1) buy만 실행하는 경우

*event-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 100
[buy] Elapsed seconds : 0.063271 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 200
[buy] Elapsed seconds : 0.135768 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 300
[buy] Elapsed seconds : 0.199019 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 400
[buy] Elapsed seconds : 0.269465 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 500
[buy] Elapsed seconds : 0.322812 s
```

*thread-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 100
[buy] Elapsed seconds : 0.016210 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 200
[buy] Elapsed seconds : 0.066962 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 300
[buy] Elapsed seconds : 0.091124 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 400
[buy] Elapsed seconds : 0.176279 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 500
[buy] Elapsed seconds : 0.211558 s
```

모든 명령어가 buy인 경우의 elapsed time은 위와 같이 나오는 것을 확인할 수 있다. 이를 통해, thread-based server의 실행 시간이 event-based server에 비해 훨씬 빠르다는 것을 확인할 수 있다.

위의 결과를 그래프로 나타내면 이를 더 명확하게 확인할 수 있다.

(2) sell만 실행하는 경우

*event-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 100
[sell] Elapsed seconds : 0.068476 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 200
[sell] Elapsed seconds : 0.123493 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 300
[sell] Elapsed seconds : 0.188721 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 400
[sell] Elapsed seconds : 0.251175 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 500
[sell] Elapsed seconds : 0.316214 s
```

*thread-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 100
[sell] Elapsed seconds : 0.022547 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 200
[sell] Elapsed seconds : 0.034668 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 300
[sell] Elapsed seconds : 0.095764 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 400
[sell] Elapsed seconds : 0.161854 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 500
[sell] Elapsed seconds : 0.200223 s
```

모든 명령어가 sell인 경우에도 elapsed time은 위와 같이 나오는 것을 확인할 수 있다. 이를 통해, thread-based server의 실행 시간이 event-based server에 비해 훨씬 빠르다는 것을 확인할 수 있다.

위의 결과를 그래프로 나타내면 이를 더 명확하게 확인할 수 있다.

(3) show만 실행하는 경우

*event-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 100
[show] Elapsed seconds : 0.057742 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 200
[show] Elapsed seconds : 0.109777 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 300
[show] Elapsed seconds : 0.162077 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 400
[show] Elapsed seconds : 0.212599 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 500
[show] Elapsed seconds : 0.275184 s
```

*thread-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 100
[show] Elapsed seconds : 0.039933 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 200
[show] Elapsed seconds : 0.066062 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 300
[show] Elapsed seconds : 0.080760 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 400
[show] Elapsed seconds : 0.166206 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 500
[show] Elapsed seconds : 0.193235 s
```

모든 명령어가 show인 경우의 elapsed time은 위와 같이 나오는 것을 확인할 수 있다. 이를 통해, thread-based server의 실행 시간이 event-based server에 비해 훨씬 빠르다는 것을 확인할 수 있다.

다만 처음에 예상했던 것과 달리 오히려 show의 실행속도가 buy, sell에 비해 빠르다는 사실을 확인하여, 당초의 예상이 틀렸음을 알 수 있었다.

위의 결과를 그래프로 나타내면 이를 더 명확하게 확인할 수 있다.

(4) 3개를 random으로 실행하는 경우

*event-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 100
[random] Elapsed seconds : 0.071875 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 200
[random] Elapsed seconds : 0.134826 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 300
[random] Elapsed seconds : 0.192294 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 400
[random] Elapsed seconds : 0.263585 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task1$ ./multiclient 127.0.0.1 60000 500
[random] Elapsed seconds : 0.352701 s
```

*thread-based

```
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 100
[random] Elapsed seconds : 0.034747 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 200
[random] Elapsed seconds : 0.046216 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 300
[random] Elapsed seconds : 0.114534 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 400
[random] Elapsed seconds : 0.129265 s
jiseok99@LAPTOP-0TK0MJB5:~/sp-prj2/task2$ ./multiclient 127.0.0.1 60000 500
[random] Elapsed seconds : 0.181169 s
```

명령어가 random인 경우의 elapsed time은 위와 같이 나오는 것을 확인할 수 있다. 이를 통해, thread-based server의 실행 시간이 event-based server에 비해 훨씬 빠르다는 것을 확인할 수 있다.

위의 결과를 그래프로 나타내면 이를 더 명확하게 확인할 수 있다.

*결과

예상했던 대로 thread-based server의 실행 속도가 전체적으로 event-based server에 비해 빠르다는 사실을 확인할 수 있었다. 다만 show 명령어가 buy, sell에 비해 실행속도가 느릴 것이라고 예상한 부분은 틀린 것으로 드러났다. 실험을 통해 확인한 결과, 오히려 show 명령어의 실행 시간이 buy, sell에 비해 빠른 것을 확인할 수 있었다.