System Programming Project 3

담당 교수 : 이영민

이름 : 김정환

학번 : 20211522

1. 개발 목표

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

여러 client의 동시 접속 및 서비스를 위한 Concurrent stock server를 구축하는 것이 목표이다. Server는 stock.txt 파일에 존재하는 주식의 정보를 binary tree에 저장하여 client의 명령에 따라 정보를 관리하고 모든 client가 접속을 종료했을 때 stock.txt에 파일을 업데이트한다. 프로젝트의 task는 3단계로 구성되어 있는데, task 1에서는 I/O Multiplexing을 이용한 방식으로 여러 client의 요청을 처리한다. Task 2에서는 thread를 이용하여 여러 client의 요청을 처리하고 task 3에서 이 두 가지 방식에 대해서 성능을 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   Event-driven Approch에서는 인자로 받은 port 번호를 이용하여 listen해준다. Client는 IP와 port를 통해서 server에 접속하고, server는 이를 Accept하여 연결시킨다. 이 때 connfd를 clientfd 배열에 저장하여 연결을 관리한다. 이 후에 client로부터 전달 받는 명령어들에 따라서 적절하게 response를 client에 보내고 데이터를 처리하는 방식으로 동작한다. 배열을 순회하여 모든 client가 접속을 종료하면 txt 파일에 주식 데이터를 저장한다.

2. Task 2: Thread-based Approach

   Event-driven Approach와 동작하고자 하는 목표는 같지만 concurrent하게 동작하기 위한 방식이 다르다. Thread를 통해서 구현하기 때문에 thread를 create 시키고 각각의 thread에서 connfd를 처리해준다. Master thread에서 accept한 connfd를 buf에 저장하고 들어온 순서대로 dequeue하여 서비스해준다.

3. Task 3: Performance Evaluation

   Task 3에서는 앞의 task 1과 task 2에서 구현한 두 서버의 elapsed time을 측

정하여 성능을 분석하고 평가한다. 이 때 client의 수를 증가시키며 그 때 명령어를 동시에 처리하는 처리율을 분석하는 과정을 거친다. Elapsed time이 클수록 동시 처리율이 떨어진다고 볼 수 있다.

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

connfd를 관리하는 structure는 강의 자료에 나오는 pool을 그대로 사용하였다. 이는 read_set과 ready_set을 사용하여 입력받는 fd들을 관리하고, clientfd 배열에 현재 접속하고 있는 client의 connfd를 저장해둔다. Select 함수를 통해 들어오는 요청을 connfd에 따라 구분하고 명령을 처리하여 response를 보내기 때문에 한 번에 한 client만을 처리하는 것이 아닌 여러 client의 명령을 concurrent하게 처리한다.

✓ epoll과의 차이점 서술

select는 검사할 수 있는 fd의 개수에 한계가 존재하고 최대 표시가 되어있는 index까지는 모두 순회하여 FD_ISSET을 통해서 확인하는 과정이 존재한다. epoll은 select보다 이후에 나온 기법으로 file descriptor들의 정보를 담은 저장소를 os가 관리하도록 하였다. 따라서 변경 사항이 발생하였을 때만 os에 요청하여 받아오므로 더 향상된 성능을 보인다. 따라서 갱신된 부분만 넘기고 받으므로 더 큰 규모로 운영하는데 있어서 epoll이 유리한 부분을 보인다.

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

✓ Master Thread의 Connection 관리

Master thread에서는 sbuf 구조체를 초기화해준 후에 설정한 thread의 개수만큼 pthread_create 함수로 thread를 create해준다. 그 후에 accept로 들어오는 connfd는 sbuf 구조체에 enqueue하여 저장한 후에 들어온 순서대로 dequeue하고 thread 함수 내에서 처리하도록 한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread에서는 *thread 함수에서 우선 connfd의 값을 sbuf에서 꺼내 지역변수에 저장하였다. 그 후 echo_cnt 함수를 호출하여 서버로부터 rio_readlineb를 이용하여 읽은 후에 show 명령의 경우에는 show 함수를 호출하여 명령을 처리한다. exit의 경우에는 buf만 null로 만들어준 후에 data를 저장하고 종료하도록 하였다. 다음으로 이외의 명령에 대해서는 trade 함수 내에서 명령을 처리하도록 한다. 이 후에 connfd를 close로 닫아주었다.

- **Task3 (Performance Evaluation)**

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

얻고자 하는 metric으로는 client 수의 증가에 따른 task 1과 task 2에서의 elapsed time의 변화, client의 요청 타입에 따른 elapsed time의 변화가 있다. Client의 요청 타입으로는 show, buy, sell 명령에 대해서 각각 한 가지만 보낼 때, 섞어서 보낼 때가 존재한다. 다음으로는 위의 경우들에서 나타나는 task 1과 task 2의 elapsed time의 차이가 있다. 본래의 목표는 동시 처리율을 나타내는 것인데, 동시 처리율은 elapsed time이 클수록 낮아지고, 작을수록 높아진다고 할 수 있다. 따라서 동시 처리율을 (client 수)/(elapsed time)으로 측정하기로 하였다. 측정 방법으로는 multiclient 코드 내에서 서비스 요청을 시작하는 시점부터 해당 프로그램이 종료되는 시점까지를 측정하는 것으로 하였고, 이 시간은 gettimeofday 함수를 사용하여 측정하였다.

✓ Configuration 변화에 따른 예상 결과 서술

우선 client 수가 증가하는 것에 따라서는 task 1, task 2 모두 요청의 수가 증가하므로 elapsed time은 증가할 것이다. task 1에서는 client의 증가에 따라 선형적으로 증가할 것이라 예측하였고, task 2에서는 thread의 개수보다 client가 적을 때는 task 1보다 느리게 증가하다가 thread 개수보다 client가 많아지면 빠르게 증가할 것으로 예측하였다.

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 server의 경우, port 번호는 인자로 전달받는다. 또한 stock.txt에서 data를 가져

와서 binary tree에 저장해야 하므로 binary tree를 구성하기 위한 make_tree 함수를 추가하였다. make_tree 함수는 stock.txt 파일을 읽기 모드로 열고, ID, left_stock, price를 가지고 있는 item 구조체에 data를 넣어준 후, node 구조체 안에 들어가도록 하여 binary tree로 연결하였다. 그리고 signal 중에서 SIGINT signal에 대해서는 handling을 하기위해서 sigint_handler를 추가했다. SIGINT가 왔을 때 바로 종료하는 것이 아닌 메모리를 할당하여 만든 binary tree의 메모리를 해제하기 위해 free_tree 함수를 넣고 exit(0)를 하도록 구현했다. 그 후 open_listenfd를 하며 서버의 기능이 시작되는데, 이 부분부터 task 1과 task 2가 달라진다. Task 1에서는 pool 구조체를 사용한다. 이는 bitmap과 clientfd로 각 connfd를 관리하는데, 이는 초기에 init_pool로 초기화하여 사용한다. 이 후 select로 받아온 후 accept로 들어온 connfd는 add_client 함수를 통해서 추가해준다. 이때 connfd는 clientfd에 저장하고, read_set에 FD_SET으로 표기한다. Client가 추가된 이후, check_client 함수 내에서 client로부터 입력을 받고 명령에 맞는 response를 만들어 보낸다. 이 때 show 명령에서는 show 함수를 실행하여 tree의 정보를 buf에 담아서 보내고, exit의 경우에는 connfd를 Close하고, read_set과 clientfd에서도 각각 FD_CLR과 -1로 삭제를 표기한 후에 함수를 종료시킨다. 이외의 명령어는 trade 함수 내에서 처리하는데, 해당 명령어에서는 buy, sell 명령을 구현하고, 이외의 입력에 대해서는 invalid command를 보내도록 하였다. Task 1에서는 check_client 함수가 종료된 후에 clientfd를 순회하며 연결된 client가 있는지 확인 후에 하나도 없으면 save_data를 통해 stock.txt에 data를 저장한다. Task 2에서는 thread 기반이므로 sbuf_t 구조체를 사용하여 구현했다. 이는 sbuf_init으로 초기화하여 사용하고, 이는 들어온 connfd를 저장하고 들어온 순서대로 제거하여 내보낸다. 초기화 이후에 정의한 NTHREADS만큼 pthread_create로 thread를 만든다. 그 후 실행되는 thread 함수에서 connfd를 sbuf에서 꺼내오고 echo_cnt 함수 내에서 들어온 명령을 수행한 후에 Close로 connfd를 닫아준다. echo_cnt 함수는 task 1에서의 check_client와 유사하지만 P, V로 lock을 걸어주는 부분이 있는 점에 차이가 있다. 또한 exit이 들어올 때마다 save_data를 통해 data를 저장하도록 했다.

3. 구현 결과

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

  Task 1과 task 2에 대해서 각각 concurrent하게 client로부터 받은 명령을 처리하는 server를 구현하였다. 연결을 요청하면 각각 pool, sbuf_t 구조체에 connfd를

저장한다. Client는 show, buy, sell의 명령어를 사용할 수 있고, show는 현재 tree에 저장되어 있는 data를 바탕으로 stock 데이터를 보여주고, buy는 tree 내 data에 존재하는 값을 바탕으로 구매 가능한 경우에 결과를 반영한다. sell은 stock을 판매하므로 tree 내 존재하는 data에 개수를 더해주게 된다. Task 1에서는 모든 client가 접속을 종료할 시에 save_data를 통해 저장하고 task 2에서는 client가 exit을 할 때마다 data를 저장하도록 구현했다.

4. 성능 평가 결과 (Task 3)

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

1. 결과 정리 및 그래프

1) 모든 client가 show만 요청하는 경우

-측정 시점

```
struct timeval start, end;
unsigned long sec;
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
```

```
gettimeofday(&end, 0);
sec = (end.tv_sec * 1000000) + end.tv_usec - start.tv_sec * 1000000 - start.tv_usec;
printf("elapsed time : %lu microseconds\n", sec);
return 0;
```

-출력 결과

<task 1>

Client가 1개일 때

```
elapsed time : 10011080 microseconds
```

Client가 2개일 때

```
elapsed time : 10011302 microseconds
```

Client가 4개일 때

```
elapsed time : 10014084 microseconds
```

Client가 8개일 때

Client가 16개일 때

Client가 32개일 때

Client가 64개일 때

Client가 128개일 때

Client가 500개일 때

<task 2>

Client가 1개일 때

Client가 2개일 때

Client가 4개일 때

Client가 8개일 때

Client가 16개일 때

Client가 32개일 때

```
elapsed time : 10038768 microseconds
```

Client가 64개일 때

```
elapsed time : 10075379 microseconds
```

Client가 128개일 때

```
elapsed time : 20041537 microseconds
```

Client가 500개일 때

```
elapsed time : 50066222 microseconds
```

-elapsed time graph

-동시 처리율 graph

2) 모든 client가 buy나 sell만 요청하는 경우

-측정 시점

```
struct timeval start, end;
unsigned long sec;
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
```

```
gettimeofday(&end, 0);
sec = (end.tv_sec * 1000000) + end.tv_usec - start.tv_sec * 1000000 - start.tv_usec;
printf("elapsed time : %lu microseconds\n", sec);
return 0;
```

-출력 결과

<task 1>

Client가 1개일 때

```
elapsed time : 10010888 microseconds
```

Client가 2개일 때

```
elapsed time : 10012588 microseconds
```

Client가 4개일 때

Client가 8개일 때

Client가 16개일 때

Client가 32개일 때

Client가 64개일 때

Client가 128개일 때

Client가 500개일 때

<task 2>

Client가 1개일 때

Client가 2개일 때

Client가 4개일 때

Client가 8개일 때

Client가 16개일 때

```
elapsed time : 10028365 microseconds
```

Client가 32개일 때

```
elapsed time : 10037112 microseconds
```

Client가 64개일 때

```
elapsed time : 10056337 microseconds
```

Client가 128개일 때

```
elapsed time : 20041570 microseconds
```

Client가 500개일 때

```
elapsed time : 50067233 microseconds
```

-elapsed time graph

-동시 처리율 graph

3) 모든 client가 buy, sell, show를 섞어서 요청하는 경우

-측정 시점

```
struct timeval start, end;
unsigned long sec;
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
```

```
gettimeofday(&end, 0);
sec = (end.tv_sec * 1000000) + end.tv_usec - start.tv_sec * 1000000 - start.tv_usec;
printf("elapsed time : %lu microseconds\n", sec);
return 0;
```

-출력 결과

<task 1>

Client가 1개일 때

```
elapsed time : 10010800 microseconds
```

Client가 2개일 때

```
elapsed time : 10011068 microseconds
```

Client가 4개일 때

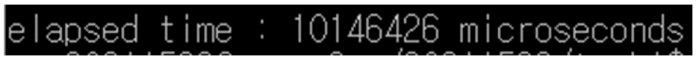
Client가 8개일 때

Client가 16개일 때

Client가 32개일 때

Client가 64개일 때

Client가 128개일 때



Client가 500개일 때

<task 2>

Client가 1개일 때

Client가 2개일 때

Client가 4개일 때

Client가 8개일 때

Client가 16개일 때

```
elapsed time : 10020199 microseconds
```

Client가 32개일 때

```
elapsed time : 10036012 microseconds
```

Client가 64개일 때

```
elapsed time : 10069685 microseconds
```

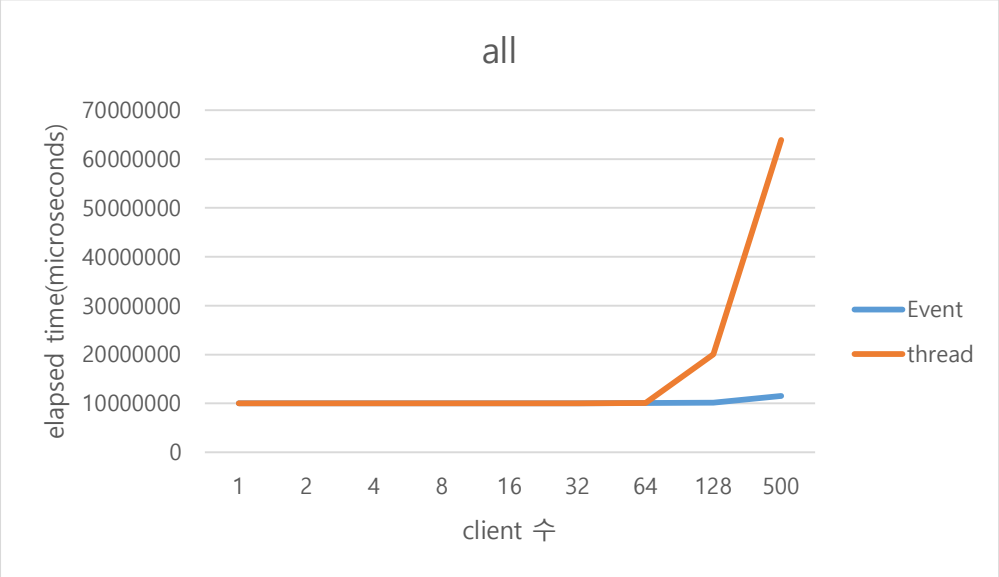
Client가 128개일 때

```
elapsed time : 20035953 microseconds
```

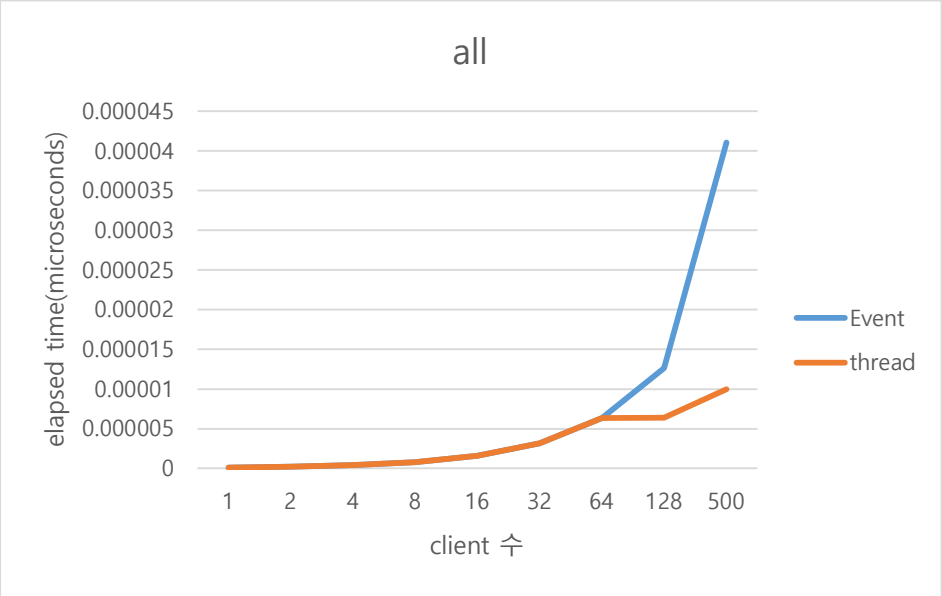
Client가 500개일 때

```
elapsed time : 63914408 microseconds
```

-elapsed time graph



-동시 처리율 graph



2. 결과 분석

측정 시점은 multiclient 코드 내에서 process가 시작되어 while loop에 들어가기 전부터 main 함수가 return 0를 하면서 종료하기 전까지를 측정시점으로 하여 측정하였다. 우선 그래프를 전체적으로 살펴보면, client의 개수가 증가할수록 점차 elapsed time이 증가하는데, Event-based인 task 1의 경우에는 증가가 선형에 가깝게 보이는 점을 확인할 수 있고, Thread-based인 task 2의 경우에는 client가 128, 500으로 갈수록 지수적으로 증가하는 듯한 모습을 확인할 수 있다. 이는 주어진 코드에서 Thread의 개수를 100으로 했기 때문으로, thread의 개수를 넘어가는 client의 수가 되었을 때 elapsed time이 크게 증가하며 동시 처리율 역시 증가폭이 task 1과 달라진다. 반면에 task 1의 경우에는 128, 500 등의 많은 client에도 elapsed time의 증가폭이 thread에 비해서는 덜한 점을 볼 수 있다. 그러나 task 2는 thread의 개수보다 적은 client의 수에서 task 1에 비해서 강점을 가진다. 이는 client의 수가 thread의 개수보다 적은 경우에 대해서 elapsed time을 관찰하였을 때 더 적게 나타나는 것에서 확인할 수 있다.

다음으로 명령어의 종류에 따른 결과이다. 이는 show만 쓰거나 buy, sell만 쓰거나, 모두 섞어서 쓰는 경우로 구분되었는데, 그래프의 개형도, elapsed time도 크게 차이나지 않는 점을 확인할 수 있었다.

결론적으로, thread의 개수가 client의 수와 비교하여 충분한 경우는 thread를 사용하는 방식이, 이외에는 Event-based 방식이 더 유리하다는 점을 확인할 수 있었다.