System Programming Project 3

담당 교수 : 박성용

이름 : 임나현

학번 : 20211582

1. 개발 목표

- 아래 항목을 구현했을 때의 결과를 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

여러 client가 동시에 접속하여 여러 서비스를 요청할 수 있는 concurrent 주식 서버를 구현하는 것이 목표이다. 주식 서버는 stock.txt 파일에 저장되어 있는 주식 정보를 binary tree 형태로 저장하고, 여러 client가 해당 주식 정보를 보고, 사고 파는 동작을 가능하도록 한다. Client가 요청한 서비스의 결과에 맞게 server에서는 주식 정보를 업데이트해주고 요청에 대한 response를 출력시켜야 하며 최종적으로 stock.txt파일에 주식 정보를 업데이트해야 한다. 일반 server는 하나의 client의 접속 또는 서비스 요청이 종료되기 전까지 다른 client를 처리하지 못하는 반면, concurrent 주식 서버는 여러 client들의 목록을 적은 buffer들을 통해 concurrent하게 처리할 수 있다.

Task1에서는 event 기반의 concurrent 서버를 구현하는 것이 목표이다. Connected descriptor를 관리할 수 있는 구조체를 만들고 그 구조체 안에 client를 관리할 수 있는 clientfd배열을 만들어야 한다. 서버는 select 함수에 의해서 monitoring하며, 만약 connect를 요청하는 listenfd가 존재하는 경우 해당 client를 connfd에 추가하는 과정이 필요하다. 그 후 client로부터 서비스 요청이 들어온 경우를 확인하여 동작을 진행해야 한다.

Task2에서는 thread 기반의 concurrent 서버를 구현하는 것을 목표로, client에서 connection이 들어와 서비스를 요청할 때, main thread가 아닌 peer thread에서 따로 동작할 수 있도록 해야 한다. Connfd를 관리할 수 있는 sbuf를 만들어 thread 함수에서 sbuf에 저장되어 있는 connfd를 하나씩 제거하면서 서비스 요청을 수행하는 과정이 필요하다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

Stockserver에서 listenfd로 connection을 요청하는 client를 받기 위해 port넘버를 인자 값으로 받는다. Stockclient에서는 연결을 하고 싶은 server의 IP address와 port를 인자 값으로 받는다. 해당 client의 연결 요청을 stockserver에서 listenfd로 듣고 있다 accept를 해주면 client의 서비스 요청을 관리할 수 있는 connfd 배열에 포함시킨다. Server에서는 새로운 client 요청을 위한 listenfd와 connfd 배열을 포함한 read_set이라는 bitmap에서 select함수를 사용하여 현재 서버에게 요청되고 있는 서비스들을 concurrent하게 처리한다.

2. Task 2: Thread-based Approach

   Task 1과 마찬가지로 stockserver에서는 listenfd로 connection을 요청하는 client를 듣고, 여러 client에서 들어온 서비스 요청을 처리해야 한다. Task1과의 차이점은 thread를 create시키고 해당 thread에서 connfd를 처리해주는 방식이다. Master thread에서는 client의 connection을 accept해주고, buffer에 connfd를 저장한 후 각각의 thread에서 buffer에 저장되어 있는 connfd를 제거하여 서비스를 처리한다.

3. Task 3: Performance Evaluation

   Task3에서는 even 기반의 concurrent server와 thread 기반의 concurrent server의 elapsed time을 비교하여 성능을 비교해본다. 구현한 thread 기반의 concurrent server는 먼저 생성할 thread의 개수를 다르게 설정하기 때문에 thread개수가 10일 때, 100일 때 두 가지 case에 대해 비교하고 이를 even 기반의 서버와 비교한다. client개수를 1, 5, 10, 50, 100, 500까지 증가시키면서 elapsed time은 어떻게 증가하는지 분석한다.

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

  가장 처음으로, connfd를 관리하는 pool을 초기화 시키고, client로부터의 연

결 요청을 기다리는 listenfd를 포함시킨다. 해당 pool의 bitmap을 select함수를 통해 connfd를 통한 client의 서비스 요청이나 listenfd를 통한 새로운 client의 연결 요청에 대해 모두 처리를 해준다. 각각의 client를 처리할 수 있는 bitmap을 사용하기 때문에 하나의 client가 연결을 요청하고 연결을 끊을 때까지 다른 client가 기다리는 것이 아니라 concurrent하게 돌아갈 수 있는 방식이다. 해당 요청사항을 반영하여 stock.txt에도 새롭게 저장한다.

✓ epoll과의 차이점 서술

select를 사용한 위의 방식과 다르게 epoll은 규모가 큰 파일 디스크립터를 처리할 때 효율적으로 관리할 수 있다. Select는 파일 디스크립터 수의 제한이 있고, epoll은 파일 디스크립터 수의 제한이 있다. Select는 모든 파일 디스크립터에 대해서 매번 검사를 한다는 점에서 디스크립터 수가 많아질수록 선형적으로 성능이 저하되지만 epoll은 event가 발생했을 때만 처리가 이루어지기 때문에 성능이 좋으며 O(1) 시간 복잡도를 가진다.

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

✓ Master Thread의 Connection 관리

Master thread에서 우선 connfd를 관리하는 sbuf를 init한다. sbuf에는 정해진 slot의 개수와 item 등에 대한 정보를 저장하고 있다. 또한, 처음에 설정한 worker thread개수만큼 pthread_create함수로 thread를 create해준다. 그 후, listenfd로 connection이 요청되는 것을 듣고 accept함수로 client의 connection 요청을 수락하여 connfd를 만든다. 이러한 connfd는 반복적으로 sbuf에 저장되어 여러 client에서 connection을 요청한 경우 sbuf에 하나씩 쌓이게 된다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread에서는 먼저 master thread에서 pthread_create로 생성되어 있다가 *thread함수에서 sbuf에 담겨있는 connfd를 제거한다. 해당 connfd에 대해 echo_Cnt함수에서 buy, sell 등의 서비스 요청을 처리한다. 각각의 worker thread마다 sbuf에서 한 개씩의 connfd를 꺼내고 sbuf에는 제거가 되는 것이기 때문에 독립적으로 수행할 수 있다. 또한, 여러 client에서 서비스 요청을 보낸 경우에 각각의 worker thread마다 하나씩 connfd를 concurrent

하게 진행한다.

- **Task3 (Performance Evaluation)**

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

  얻고자 하는 metric의 정의는 client수가 증가할 때 task1과 task2의 elapsed time이 어떻게 증가하는지, 또 task1과 task2의 그래프가 어떻게 다른지 비교하는 것이다. 각각을 test하는 방법은 크게 3가지로 구분하여 show만 했을 때, buy와 sell만 했을 때, show, buy, sell을 모두 했을 때를 비교한다. Pre-thread 방식의 경우 미리 설정한 worker thread의 개수에 따라 처리율이 달라지기 때문에 thread의 개수가 10개일 때 100개일 때를 구분하여 정하였다. 측정 위치는 multiclient파일에서 서비스를 요청하고 response를 기다리는 부분에 대해서 gettimeofday함수를 통해 특정한다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  Task1과 Task2 모두 client수가 증가할수록 서비스 요청이 많아지기 때문에 elapsed time이 증가할 수밖에 없을 것이다. Task1은 client수가 증가할수록 그에 맞게 처리시간이 선형적으로 증가할 것이라고 생각된다. 그러나, Thread 개수보다 작은 client 개수를 가졌을 때에는 비슷한 처리시간을 필요로 하다가 thread 개수를 넘어서는 client 개수를 처리할 때 처리시간이 급격히 증가할 것이라고 예상된다. 또한, thread 개수가 10개일 때에 비해서 thread 개수가 100개이면 동시에 처리할 수 있는 connfd가 더 많아지기 때문에 성능이 높아질 것 같다.

C. 개발 방법

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

Stockserver에서는 가장 먼저 stock.txt파일에 저장되어 있는 주식 정보를 저장할 binary tree 구조체가 필요하다. Main에서 client와의 연결을 처리하기 전에 binary tree를 init하는 함수가 필요하다. Init_tree에서는 stock.txt 파일을 read 모드로 열어 그 안에 있는 item의 ID, 남은 수량, 가격을 node에 item 구조체에 저장한다.

그 후에, 각 item을 ID 순서대로 정렬하여 하나의 binary tree인 node를 완성시킨다. 그 후, task1과 task2는 동일하게 새로운 client의 연결 요청을 처리하는 listenfd를 만든다. Task1에서는 그 이후, pool이라는 구조체를 사용하여 client 요청들을 처리할 수 있는 구조체를 init한다. Pool에는 항상 listenfd가 포함되어 있어야 한다. 그 후, pool에 있는 bitmap에서 select함수를 통해 현재 서비스 요청이 들어온 client들의 clientfd를 처리한다. 만약, listenfd로 새로운 client가 connection을 요구한 경우에는 clientfd에 먼저 해당 connfd를 추가해준다. 그 후, bitmap을 탐색하면서 client의 show, buy, sell 명령어에 맞는 서비스를 처리하고 client가 connect를 끊으면 server 또한 해당 connfd를 close하고 pool에서 삭제한다. 반면에, Task2에서는 main thread에서 여러 client의 서비스 요청을 처리할 수 있는 sbuf를 만들어야 하며, 사용하기 전에 sbuf를 init해줘야 한다. 그 후, 서버가 정의한 nthread개수(worker thread 개수)만큼 pthread를 create하는 pthread_create함수를 사용하며, 해당 인자로 thread라는 함수를 넘겨준다. Thread 함수에서는 sbuf에 저장되어 있는 client 요청 중 하나를 delete하고 처리를 해주는 방식으로 작동하며, main thread는 새로운 client가 connection 요청을 보내는 것을 처리해주는 방식으로 작동된다. 따라서, main thread와 worker thread에서 처리하는 작업이 분리된다. Worker thread에서는 sbuf에 담긴 connfd에 맞게 client가 buy를 요청하는 경우 tree에 저장되어 있는 ID를 찾아 buy를 할 수 있게 하며, sell을 요청하는 경우, show를 요청하는 경우 각각에 맞는 동작을 할 수 있게 된다. Task1과 마찬가지로 client가 요청하는 서비스를 처리한 후에, stock.txt에 다시 바뀐 주식정보를 업데이트해야 한다.

3. 구현 결과

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

server는 port number를 인자로 받고, client는 server의 IP address와 port number를 인자로 받는다. Stock.txt에 있는 주식 정보를 기반으로 binary tree를 구성한다. Client가 우선 연결을 요청하면 server에서 연결 요청을 듣는 listenfd 파일디크립터로 확인하고 sbuf나 pool 등에 client의 connfd를 저장한다. Client는 크게 세 가지 서비스를 요청할 수 있는데, 먼저 show를 한 경우 stock.txt파일에을 저장한 binary tree의 정보를 기반으로 현재 모든 주식 정보를 server가 client에 write한

다. 다음으로, client가 buy를 한 경우 server는 해당 주식 ID를 tree에서 찾아 buy 한 수량만큼 남은 수량에서 뺀 후 다시 binary tree의 node를 수정하고, stock.txt 파일에도 업데이트한다. 만약, buy하고 싶은 주식 ID가 없는 경우, 남은 수량보다 더 많은 수량을 사고 싶어 하는 경우, 에러 메세지를 rio_write로 client에 출력한다. Client의 sell의 경우에도, sell한 만큼 해당 주식 ID를 tree에서 찾아 업데이트 하고 stock.txt파일에도 업데이트 한다. Task1에서는 client가 요청한 서비스를 pool이라는 structure구조에서 clienfd배열에 저장하고 search하면서 concurrent하게 처리할 수 있다. 반면에, task2에서는 main thread에서는 client의 연결 요청을 처리하고 worker thread에서 client가 요청한 서비스를 저장한 sbuf에서 하나씩 제거하여 concurrent하게 처리할 수 있다.

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**
- 모든 client가 show만 요청하는 경우

(1) 측정 시점

```
struct timeval start;
struct timeval end;
unsigned long e_usec;
```

```
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
    //wait(&state);
    pids[runprocess] = fork();
```

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

(2) 출력 결과

<task1>

elapsed time : 10048469 microseconds : client 1개

elapsed time : 10105445 microseconds : client 5개

elapsed time : 10162981 microseconds : client 10개

elapsed time : 10462121 microseconds : client 50개

elapsed time : 11093503 microseconds : client 100개

elapsed time : 18406967 microseconds : client 500개

<task2>

#define NTHREADS 100일 때)

elapsed time : 10007687 microseconds : client 1개

elapsed time : 10021942 microseconds : client 5개

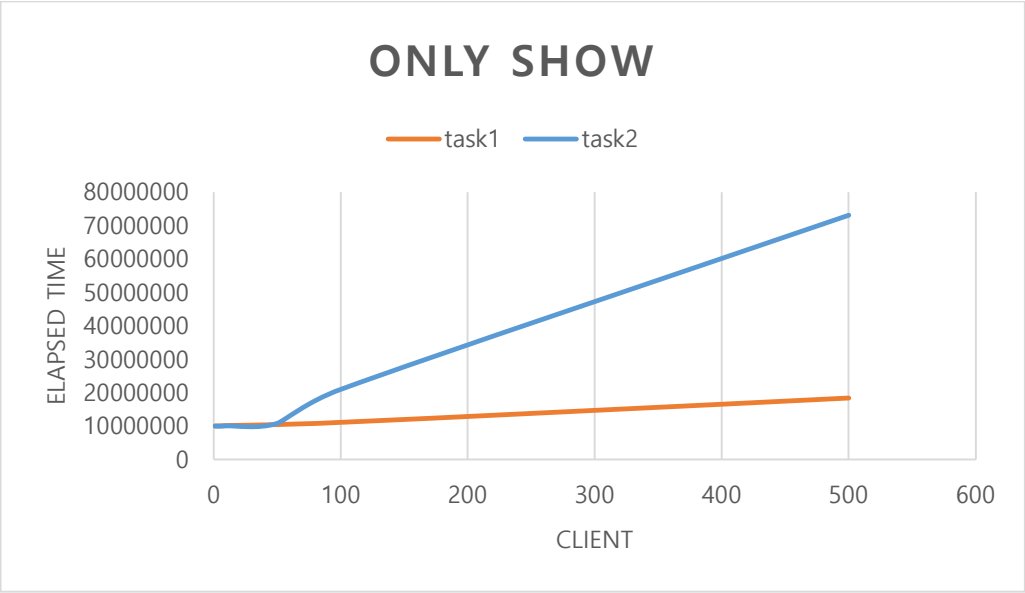
elapsed time : 10045002 microseconds : client 10개

elapsed time : 10836654 microseconds : client 50개

elapsed time : 21036333 microseconds : client 100개

elapsed time : 73140710 microseconds : client 500개

(3) Client 개수 변화에 따른 동시 처리율 graph



(4) 분석

우선, 측정은 multiclient에서 처음부터 끝까지의 시간이 아닌 서버와 연결이 된 후 서비스를 요청하고 connfd를 close한 순간까지의 부분을 측정하였다. 서버 연결을 포함할 경우, 서버의 상태에 따라 연결 시간에 차이가 존재할 것 같아서 오차를 최대한 줄이려고 하였다. Task1의 동시 처리율 graph를 살펴보면 client의 개수가 늘어날수록 clientfd배열에 담겨있는 client 서비스 요청이 많아지기 때문에 선형적으로 시간이 증가한다는 것을 확인할 수 있다. Task2는 기울기 100개 이하에서는 task1과 유사하게 기울기가 일정하다가 급격하게 커진다는 것을 확인할 수 있는데, 이는 work thread 개수를 100으로 설정하였기 때문이라는 것을 알 수 있다. 100개 이하의 client에 대해서는 concurrent하게 처리가 가능하지만 그 이상의 client는 앞선 client 요청이 끝나는 것을 기다렸다가 work thread에서 처리를 해야 하기 때문에 기울기가 급격하게 변한다.

- 모든 client가 buy, shell만 요청하는 경우

(1) 측정 시점

```
struct timeval start;
struct timeval end;
unsigned long e_usec;
```

```
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
    //wait(&state);
    pids[runprocess] = fork();
```

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

(2) 출력 결과

<task1>

elapsed time : 10031159 microseconds : client 1개

elapsed time : 10191914 microseconds : client 5개

elapsed time : 10232921 microseconds : client 10개

elapsed time : 10639539 microseconds : client 50개

elapsed time : 11345488 microseconds : client 100개

elapsed time : 16250297 microseconds : client 500개

<task2>

#define NTHREADS 10일 때)

elapsed time : 10007775 microseconds : client 1개

elapsed time : 20012218 microseconds : client 5개

elapsed time : 30018116 microseconds : client 10개

elapsed time : 130070509 microseconds : client 50개

elapsed time : 250133861 microseconds : client 100개

elapsed time : 460248727 microseconds : client 500개

#define NTHREADS 100일 때)

elapsed time : 10007221 microseconds : client 1개

elapsed time : 10011826 microseconds : client 5개

elapsed time : 10016647 microseconds : client 10개

elapsed time : 10078885 microseconds : client 50개

elapsed time : 11274982 microseconds : client 100개

elapsed time : 73995519 microseconds : client 500개

(3) Client 개수 변화에 따른 동시 처리율 graph

(4) 분석

마찬가지로, 측정시간은 client가 서비스를 요청하고 connfd를 close한 순간까지 설정하였다. Task1에서는 show 명령어만 존재하는 경우와 유사하게 client 개수가 증가할수록 처리시간이 선형적으로 증가한다는 것을 확인할 수 있다. 반면에, task2에 대해서는 thread개수가 10, 100개를 구분하였는데 thread개수가 많을수록 처리율이 증가한다는 것을 우선적으로 확인할 수 있다. Thread 개수가 10개인 경우에는 10개 단위로 concurrent하게 처리가 가능하기 때문에 기울기가 급격하게 증가한다는 것을 확인할 수 있었다.

- 모든 client가 show, but, shell 등을 요청하는 경우

(1) 측정 시점

```
struct timeval start;
struct timeval end;
unsigned long e_usec;
```

```
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
    //wait(&state);
    pids[runprocess] = fork();
```

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

(2) 출력 결과

<task1>

elapsed time : 10042697 microseconds : client 1개

elapsed time : 10123951 microseconds : client 5개

elapsed time : 10132584 microseconds : client 10개

elapsed time : 10757826 microseconds : client 50개

elapsed time : 11705095 microseconds : client 100개

elapsed time : 15958348 microseconds : client 500개

<task2>

#define NTHREADS 10일 때)

elapsed time : 10007542 microseconds : client 1개

elapsed time : 20012539 microseconds : client 5개

elapsed time : 30019680 microseconds : client 10개

elapsed time : 130139419 microseconds : client 50개

elapsed time : 250134833 microseconds : client 100개

elapsed time : 290160545 microseconds : client 500개

#define NTHREADS 100일 때)

elapsed time : 10007396 microseconds : client 1개

elapsed time : 10010887 microseconds : client 5개

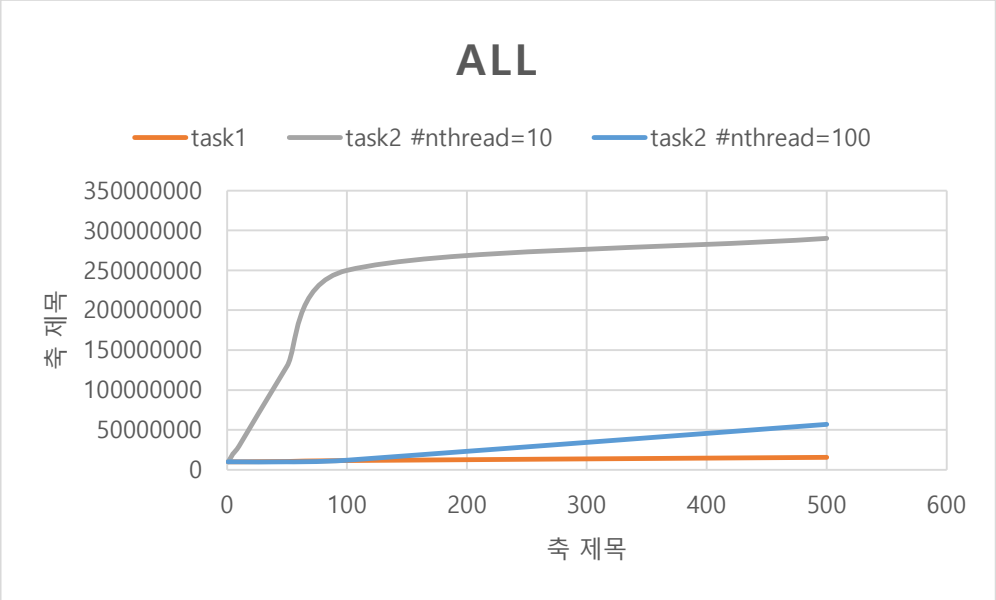
elapsed time : 10017341 microseconds : client 10개

elapsed time : 10058261 microseconds : client 50개

```
elapsed time : 12332195 microseconds
```
: client 100개

```
elapsed time : 57251185 microseconds
```
: client 500개

(3) Client 개수 변화에 따른 동시 처리율 graph



(4) 분석

마지막으로, buy sell show 모든 명령어를 random하게 실행시킬 때에 대해서 분석하였다. 측정시점은 client에서 서비스를 요청한 순간부터 connfd를 close 한 순간까지 설정하였다. Task1의 경우 event 기반의 concurrent server이기 때문에 client의 수가 증가할수록 처리시간이 선형적으로 증가한다는 것을 확인할 수 있다. 마찬가지로, task2에서 thread의 개수가 100개인 경우에도 task1과 유사하게 선형적으로 증가한다. 100개를 넘어가는 client 개수에 대해서는 기울기가 좀 더 가파르게 증가한다는 것을 확인할 수 있다. Thread 개수가 10개인 경우는 처리율이 가장 낮으며 worker thread 개수가 크면 클수록 처리율이 높아진다는 것을 확인할 수 있고, client개수가 thread 개수보다 크면 기울기가 가파르게 증가한다는 것을 알 수 있다.