Multicore Programming Project 2

담당 교수: 박성용

이름: 한석기

학번: 20211606

1. 개발 목표

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

미리 만들어진 주식 데이터를 바탕으로 주식 서버를 열고 여러 client가 해당 주식 서버에 연결하여 Concurrent하게 통신하는 것이 주된 개발 목표이다. 주식 서버는 최초 주식 정보를 저장해야 하고 다수의 client와 연결해, 충돌없이 show, buy, sell등의 요청을 처리해 주식 정보를 update해야 한다. 주식 클라이언트는 server에 연결하고 show, buy, sell등의 요청을 해야 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   stockserver를 포트 넘버와 같이 열고 stockclient혹은 multiclient를 실행시킬 시, 맞는 명령어(show, buy, sell, exit에 따라 결과를 client쪽에 출력시켜준다. 그리고 server는 몇 bytes의 명령어를 받았는지, 그리고 연결된 connfd가 몇 번인지를 출력하고 update한다. 이 경우 I/O Multiplexing 방법을 사용해 connfd를 관리한다.

2. Task 2: Thread-based Approach

   Task 1과 구현결과 자체는 비슷하고 과정이 조금 다르다. stockserver를 포트 넘버와 같이 열고 stockclient혹은 multiclient를 실행시킬 시, 맞는 명령어(show, buy, sell, exit에 따라 결과를 client쪽에 출력시켜준다. 그리고 server는 몇 bytes의 명령어를 받았는지, 그리고 연결된 connfd가 몇 번인지를 출력하고 update한다. 하지만 이 경우 thread 방법을 사용해 읽고 쓰기를 관리한다. 이 과정에서 P함수와 V함수를 통해 control해야하는 부분이 있지만 잘 처리하면 충돌없이 구현할 수 있다.

3. Task 3: Performance Evaluation

Task1과 Task2에서 구현한 결과를 주어진 multiclient를 변형해 비교하는 것이다. multiclient 내에서 시간을 측정하는 함수를 사용해 Task 1, Task 2별 각각 명령어의 client 수에 따른 동시처리율을 구하고 성능을 평가한다.

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 관리하기 위한 pool을 초기화하고 반복문을 돌며 listenfd를 확인하고 pool의 ready_set을 통해 pool에 넣고 활성화시킨다. 그리고 반복문을 돌 때마다 Select로 클라이언트의 준비된 요청이 몇개인지 확인하고 활성화된 클라이언트를 찾는 함수를 실행한다. 활성화된 클라이언트의 요청을 요청 케이스에 따라 나눠 실행하고 해당 실행 내용을 클라이언트에게 보내준다. 그리고 요청사항에 따른 주식의 변화를 저장한다.

✓ epoll과의 차이점 서술

위에서 사용한 Select는 선형적으로 작동하기 때문에 많은 수의 소켓을 처리할 때, 성능이 저하될 수 있다. 반면, epoll은 커널 이벤트 기반 매커니즘을 사용하기 때문에 많은 수의 소켓을 처리할 때, 더 나은 성능을 발휘한다. 그리고 Select는 소켓 수의 제한이 있지만, epoll은 소켓 수의 제한이 없고 Select는 주기적으로 반복문을 통해 event 발생여부를 확인하지만 epoll은 이벤트 발생 시 알림을 받고 작업을 수행해 epoll의 자원낭비가 더 적다. 하지만 epoll은 리눅스에서만 지원된다.

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

  ✓ Master Thread의 Connection 관리

  동적으로 할당한 connfdp 파일 디스크립터 포인터를 통해 Accept으로 클라이언트 연결을 수락하고 client의 수를 global 변수로 관리한다. 이 때, 세마포어의 관리가 필요하고 Pthread_create함수를 이용하여 thread를 생성하고 서버와 클라이언트의 통신을 처리한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  새로운 클라이언트가 연결될 때마다 Thread Pool에 새로운 Thread를 생성하여 작업을 처리하고 각 Thread에서 오는 요청을 반복문을 통해 확인한다. 그리고 요청에 대한 작업을 할 때, Synchronization의 작업이 필요하다. 이 때, 세마포어를 이용해 쓰기 및 읽기 작업을 관리하고 Conflict가 일어나지 않게 한다.

- **Task3 (Performance Evaluation)**

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

  얻고자 하는 metric은 client의 수가 늘어남에 따라 Task1, Task2 각각 동시처리율이 어떻게 변하는지, 그리고 buy, sell 또는 show를 처리 시 동시처리율의 차이가 client 수에 따라 변하는지, 확인하는 것이다. 앞서 말한 내용을 확인함으로써 Task1, Task2의 client 수에 따른 성능을 확인할 수 있고 어떤 작업을 처리할 때, 가장 시간을 많이 요하는지도 확인할 수 있다. 측정은 multiclient.c 파일에서 sys/time.h 헤더의 gettimeofday 함수를 이용해 측정한다.

✓ Configuration 변화에 따른 예상 결과 서술

Task1은 비동기적 방식으로 동작하기 때문에 Client의 수가 증가해도 구성 변화가 없다. 따라서 Task2에 비해 상대적으로 안정적일 것 같다. 하지만 Task1은 최대 연결 가능한 Client수가 있고 똑같이 많은 수의 Client가 연결되면 성능 저하가 발생할 것 같다. Task1은 I/O작업이 비동기적 처리이기 때문에 Show, Buy, Sell 등의 작업을 하더라도 높은 처리율을 유지할 수 있을 것이다. 따라서 Client 수가 무수히 많을 때, Task2보다 성능이 높을 것 같다.

Task2는 Client 수가 증가할수록 Thread의 수도 증가한다. 따라서 Client 수가 증가하면 Thread 관리가 어려워지고 Context Switching에 대한 Overhead가 발생할 수 있다. 그래서 처리율이 처음엔 증가하다가 어느 순간부터 성능 저하가 발생할 것 같다. Task2는 병렬로 처리하기 때문에 Client수가 적을 때, Show, Buy, Sell의 작업 수 증가에 따라 처리율은 증가할 것이다. 하지만 위에서 설명한 Overhead 문제때문에 Thread의 수가 너무 많아지면 성능이 저하된다.

C. 개발 방법

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

Task1은 주어진 stockserver.c의 소스코드에 주식을 저장할 트리 구조체, 그리고 주식의 정보에 대한 구조체, client에 Rio write해줄 char 배열을 global로 생성하고 구조체는 root 정보를 통해 관리한다. 그리고 Connected client를 관리할 pool을 내용이 유지되게 static으로 main에 선언하고 while문을 돌며 클라이언트의 요청을 확인한다. 요청은 연결요청, 그리고 show, exit, buy, sell이 있다. 연결 시 pool에 추가하고 매 loop마다 check한다. 주식 정보는 file descriptor와 txt파일을 통해 tree에 저장하고 show, buy, sell마다 tree구조를 이용해 읽고 쓰기를 한다. server에 연결된 client의 수가 0일 때, update된 tree와 file descriptor를 이용해 읽었던 txt파일에 쓴다. client에게 표시될 Rio write는 check시에 수행한다.

Task2는 Task1에서 설명한 트리 구조체를 대부분 유지한다. tree에 들어갈 주식 정보에서 주식 정보 synchronization을 위한 세마포어를 주식 정보 구조체에 추가한다. 그리고 pool을 선언하지 않고 sem_t를 이용해 세마포어를 두 개 global로 생성한다. main에서 세마포어를 초기화하고 task1과 마찬가지로 주식정보를 트리에 넣는다. 반복문을 돌며 클라이언트의 연결요청에 따라 thread를 생성하고 client의 수를 업데이트한다. thread생성 후, 동적 할당한 connection descriptor 포인터를 integer 변수에 저장하고 동적할당한 것을 free한다. 그리고 이후 과정은 Task1과 비슷하다. Task1에서 client를 반복문마다 check했듯이 각 Thread에서 client에서 명령어가 들어올 때마다 check를 해, 명령어를 처리한다. 명령어를 처리하는 과정은 Task1과 비슷하다. Task2의 모든 과정에서 global 변수, global 변수에 대한 pointer 혹은 thread 생성 시 접근할 수 있었던 변수는 모두 공유되기 때문에 변동 시, P함수와 V함수에 의한 관리가 필요하다.

3. 구현 결과

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

Task1은 Tree의 생성, Tree에 추가, Tree에서 ID를 통해 주식을 찾는 것, 주식을 사는 것, 주식을 파는 것, Tree를 모두 search해서 보여주는 것, pool의 초기화, pool에 client를 추가하는 것, client의 요청이 있는지 확인하는 것, update된 주식 정보를 저장하는 것의 함수를 구현했다. 그리고 main에서는 pool의 초기화, 주식 정보를 받는 과정을 진행하고 Client의 연결요청을 받으며 pool에 추가하는 함수를 호출한다. 이 과정에 Select, Accept등의 함수가 사용되며 반복문을 돌면서 client의 상태에 대해 check한다. check과정에서 show, exit, buy, sell의 명령어를 수행하고 client에 표시한다. 따라서 client가 포트 넘버와 IP address를 통해 연결요청 시, 연결시켜주고 show, sell, buy, exit 명령어를 요청 시 server에서 처리 후 그 결과를 client에게 보여준다. show는 tree에 저장된 주식 정보, 즉 update된 최신 주식 정보를 보여주고 sell은 client가 가지고 있는 주식을 팔아 server 주식의 잔고를 늘려준다. buy는 client가 server에 있는 주식을 사고 만약 없을 시, 남은 주식이 없다고 알려준다. exit는 client와의 연결을 종료한다.

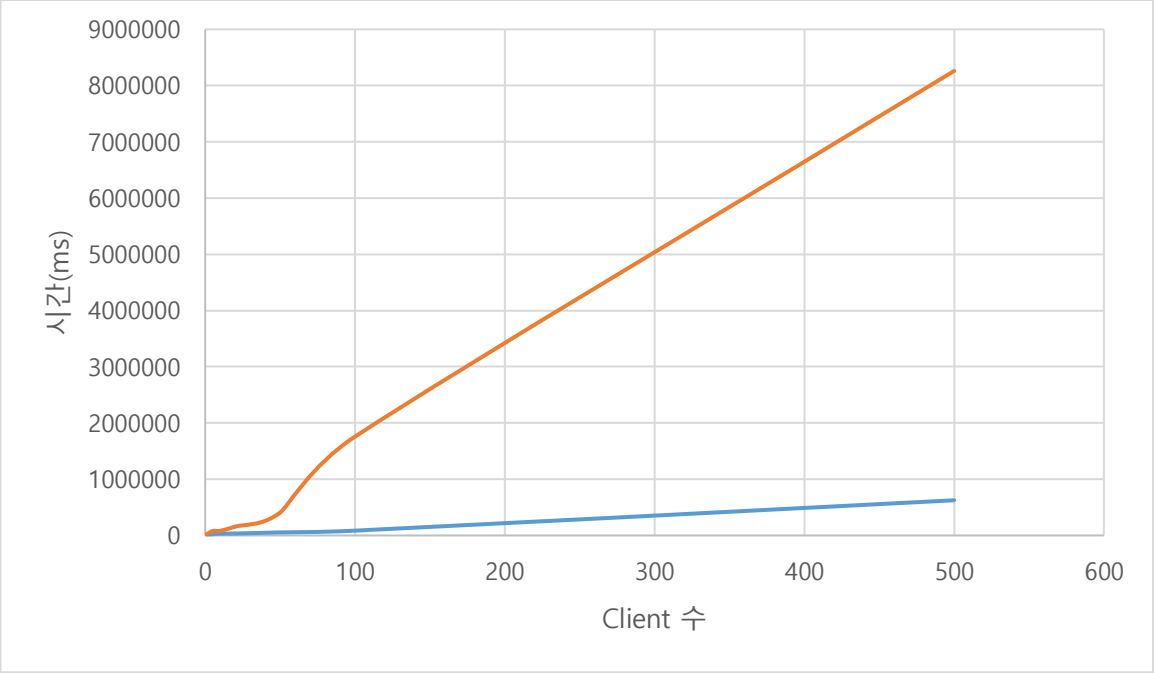
Task2는 Task1과 비교했을 때, pool이 없다는 것과 세마포어가 있다는 것이 다르다. Task2는 thread를 관리하는 함수가 있고 main에서 이 thread를 생성한다. 세마포어는 main에서 초기화하며 client 연결 시 생성되는 connection descriptor는 malloc을 통해 동적할당해 받으며 thread 생성 후, free한다. 이는 서로 다른 영역을 사용해 관리하기 위함이다. 만일 main쪽의 stack영역을 그대로 사용해 connection descriptor를 관리했다면 Conflict가 일어났을 것이다. 따라서 malloc을 이용해 heap 영역에 새로 동적할당을 하고 thread 함수 내에서 값을 주고 동적할당한 포인터는 free해버린다. P함수와 V함수를 통해 세마포어를 관리하고 Synchronization 작업을 한다. global한 부분, 즉, Conflict가 일어날 수 있는 부분을 해당 작업을 통해 처리하고 thread함수의 마지막에 연결을 종료하고 NULL을 리턴한다. 나머지 부분은 task1의 구현부분과 비슷하다. 실제 실행시켜 봤을 때, 모든 동작은 task1하고 동일했다.

4. 성능 평가 결과 (Task 3)

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

-show, buy, sell 모든 명령어를 실행시켰을 때

(1) Client 수에 따른 수행시간 그래프



(2) Task1 Task 2/ Client 수: 시간

1: `elapsed time: 18879 microseconds` `elapsed time: 21423 microseconds`

5: `elapsed time: 21423 microseconds` `elapsed time: 79080 microseconds`

10: `elapsed time: 32067 microseconds` `elapsed time: 82439 microseconds`

20: `elapsed time: 35283 microseconds` `elapsed time: 156002 microseconds`

50: `elapsed time: 53970 microseconds` `elapsed time: 405569 microseconds`

100: `elapsed time: 85523 microseconds` `elapsed time: 1757376 microseconds`

500: `elapsed time: 625678 microseconds` `elapsed time: 8260783 microseconds`

(3) 측정 시점

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

int main(int argc, char **argv)
{
    gettimeofday(&start, 0);
```

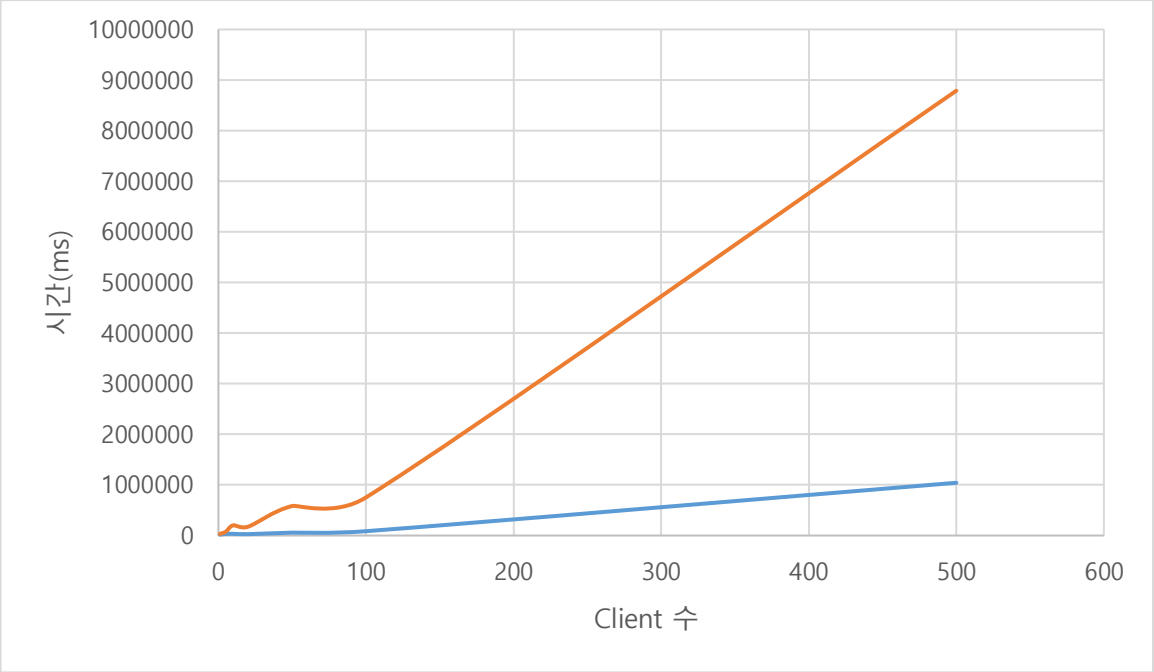
```
    gettimeofday(&end, 0);
    e_usec = ((end.tv_sec * 1000000) + end.tv_usec)
        - ((start.tv_sec * 1000000) + start.tv_usec);
    printf("elapsed time: %lu microseconds\n", e_usec);
    return 0;
}
```

(4) 분석

측정은 주어진 multiclient.c의 처음과 마지막 부분 구간을 측정했다. Task1은 그래프를 확인했을 때, 기울기가 일정한 편이며 따라서 처리율 또한 일정하다. 반면 Task2는 기울기가 완만하다가 클라이언트가 약 50인 시점부터 기울기가 급격하게 변해 처리율이 task1에 비해 떨어진다.

- buy, sell 명령어들만 실행시켰을 때

(1) Client 수에 따른 수행시간 그래프



(2) Task1 Task 2/ Client 수: 시간

1 : elapsed time: 18799 microseconds elapsed time: 32071 microseconds

5: elapsed time: 31445 microseconds elapsed time: 75564 microseconds

10: elapsed time: 34216 microseconds elapsed time: 202084 microseconds

20 : elapsed time: 27355 microseconds elapsed time: 173895 microseconds

50: elapsed time: 57312 microseconds elapsed time: 583878 microseconds

100: elapsed time: 85844 microseconds elapsed time: 752827 microseconds

500: elapsed time: 1042847 microseconds elapsed time: 8788987 microseconds

(3) 측정 시점

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

int main(int argc, char **argv)
{
    gettimeofday(&start, 0);
```

```
    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;
}
```

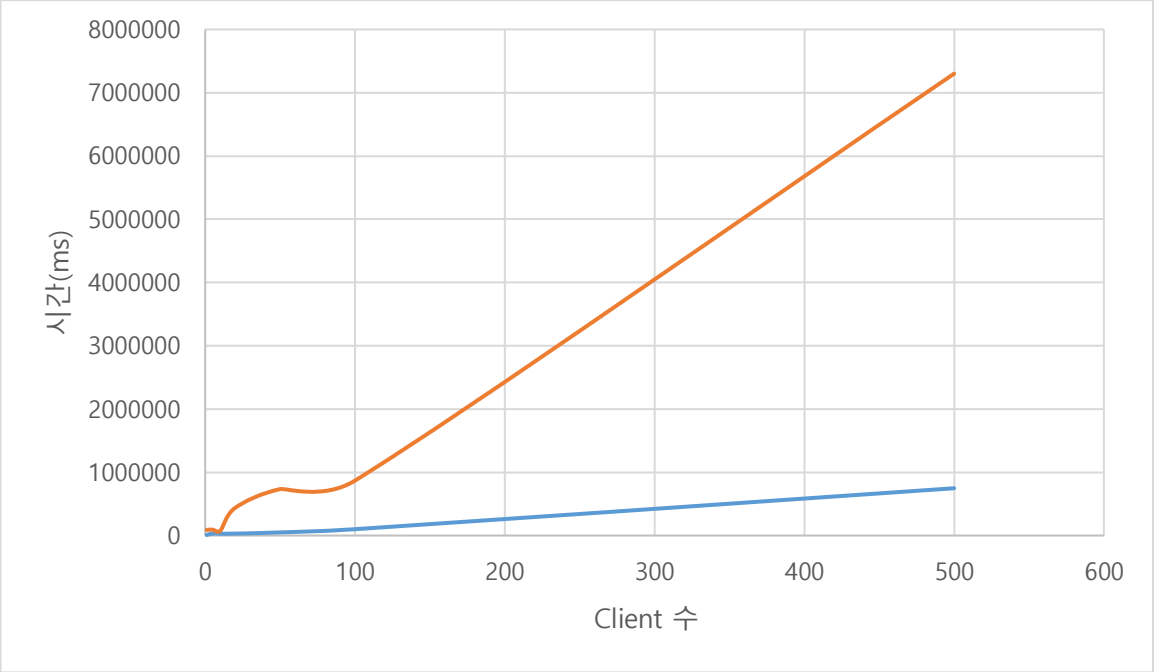
(4) 분석

```
int option = rand() % 2;
//int option = rand() % 3;

if(option == 2){//show
    strcpy(buf, "show\n");
}
else if(option == 0){//buy
    int list_num = rand() %
    int num_to_buy = rand()

    strcpy(buf, "buy ");
    sprintf(tmp, "%d", list_
    strcat(buf, tmp);
    strcat(buf, " ");
    sprintf(tmp, "%d", num_t
    strcat(buf, tmp);
    strcat(buf, "\n");
}
else if(option == 1){//sell
```

위와 같이 multiclient.c 파일을 client가 buy와 sell만 요청하게 변경하고 측정했다. show까지 모두 수행하는 것에 비해 task1과 task2 둘 다 오래 걸렸고 task2의 기울기 변화 시점은 조금 다르지만 Client 수가 무수히 많아지면 똑같이 task1에 비해 처리율이 떨어진다.

- show 명령어만 실행시켰을 때

(1) Client 수에 따른 수행시간 그래프



(2) Task1 Task 2/ Client 수: 시간

1 : elapsed time: 9121 microseconds elapsed time: 88821 microseconds

5: elapsed time: 31542 microseconds elapsed time: 93811 microseconds

10: elapsed time: 29353 microseconds elapsed time: 74363 microseconds

20 : elapsed time: 30953 microseconds elapsed time: 436715 microseconds

50: elapsed time: 49583 microseconds elapsed time: 734343 microseconds

100: elapsed time: 100338 microseconds elapsed time: 868228 microseconds

500: elapsed time: 747741 microseconds elapsed time: 7300892 microseconds

(3) 측정 시점

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

int main(int argc, char **argv)
{
    gettimeofday(&start, 0);
```

```
    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;
}
```

(4) 분석

```
int option = 2;
//int option = rand() % 3;

if(option == 2){//show
    strcpy(buf, "show\n");
}
else if(option == 0){//buy
    int list_num = rand() %
    int num_to_buy = rand()

    strcpy(buf, "buy ");
    sprintf(tmp, "%d", list_
    strcat(buf, tmp);
    strcat(buf, " ");
    sprintf(tmp, "%d", num_t
    strcat(buf, tmp);
    strcat(buf, "\n");
}
else if(option == 1){//sell
```

위와 같이 multiclient.c 파일을 client가 show만 요청하게 변경하고 측정했다. task2는 client수가 적을 때, show에 대한 처리가 buy, sell보다 오래 걸렸지만 client 수가 늘어날 수록 buy, sell에 대한 처리보다 처리율이 높았다. 그리고 기울기의 급격한 변화에 대한 것은 위에서 봤던 바와 동일하다.

- 최종 분석

Task1의 처리율이 Task2의 처리율보다 높다는 것은 위에서 실험 결과로 확인했다. 하지만 Task1은 pool 구조체와 정해진 크기를 통해 관리하기 때문에 엄청 많은 수의 client가 연결요청을 하면 감당을 못할 수 있다는 단점이 있다. Task2는 Task1에 비해 처리 속도가 느리지만 Client 수의 정해진 크기가 없고 Client의 수가 적을 때, 처리율이 괜찮다.

show만 처리할 때, buy, sell만 처리할 때에 따른 것도 또한 Client수가 적을 때의 기울기보다 Client수가 높을 때의 기울기가 더 컸고 task1의 show명령어에 대한 처리시간이 buy, sell보다 적었다. task2는 show명령어에 대한 처리시간이 Client 수가 증가함에 따라 buy, sell보다 적었지만 처음에는 높았다.

결론적으로, Client 수가 적을 때, 혹은 적당히 많을 때는 task1을 사용하는 것이 낫지만 Client 수가 무수히 많을 때는 task2를 사용하는 것이 맞다. 앞서 설명했듯이 task1은 해당 크기를 감당할 수 없기 때문이다. 이렇게 서버 및 클라이언트의 환경에 따라 사용에 유리한 방법이 다르다.