# System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 노요셉

학번 : 20171631

## 1. 개발 목표

이번 프로젝트에선 여러 client들이 동시에 접속했을 때 해당 주식 정보들을 service 해줄 수 있는 concurrent stock server를 구축해보게 된다. 각 client는 server에 주식 사기, 팔기, 정보 보기 등의 요청을 하게 된다. server는 주식 정보를 저장하고 여러 client들과 소통을 해준다. server에서 여러 client를 service해주는 방법에는 수업시간에 배운 바에 의하면 3가지가 있다. process-based, event-based, thread-based 들이 그것들이다.

이번 프로젝트에선 이중 2개, event-based와 thread-based 방법을 사용해 concurrent stock server를 구현하였다. event-based 방법의 경우 여러 개의 connected clientfd를 저장하는 pool을 만든 뒤 select 함수를 사용하여 pool에 추가해주며 여러 개의 client를 service해준다. client들의 입장에선 concurrent하게 수행하는 것처럼 보이지만 사실 수행부분에선 여러 client의 여러 명령을 sequential하게 수행하는 것일 것이다. thread-based 방법의 경우 여러 개의 thread를 만들어 놓은 뒤 client에서 connect요청이 올 때마다 해당 thread가 connect를 받은 뒤 concurrent하게 service를 해주게 된다.

먼저 stock.txt로부터 주식 정보들을 읽어온 뒤, 주식 정보들을 이진 트리에 저장해주게 된다. server는 해당 tree의 정보들을 저장하고 있고, 다수의 client가 server의 connect를 요청하였을 때 event-based, thread-based 방법을 사용해 connect를 해주고 buy, sell, show, exit등의 명령들을 concurrent하게 service해주게 된다.

## 2. 개발 범위 및 내용

### A. 개발 범위

1. select

   event-based의 방법으로 다수의 client들과의 연결을 handling할 때 사용되는 방법이다. pool이라는 자료구조를 사용하며 client로부터 연결이 올 때마다 select를 하여 ready_set에 insert를 시켜주고, 수행 부분에서 ready_set에 들어가 있는 모든 client의 명령을 하나씩 수행하여 concurrent한 service를 구현해낸다. 여기서 주의할 점은, 다수의 client가 보기엔 concurrent한 service겠지만, 함수 구현 상으론 sequential 한 logical flow를 가질 것이다.

2. pthread

   thread-based의 방법으로 다수의 client들과의 연결을 handling할 때 사용되는 방법이다. sbuf라는 자료구조를 사용하며 client로부터 연결이 올 때마다 sbuf에 저장해준 뒤 미리 생성해놨던 thread가 해당 client와 연결되게 된다. thread의 성질을 이용해 context-switch를 하며 concurrent하게 명령을 수행해주게 되어 server는 모든 client의 명령을 concurrent하게 service해주게 된다.

### B. 개발 내용

- **select**

  ✓ select 함수로 구현한 부분에 대해서 간략히 설명

  먼저 연결이 온 fd들은(listenfd의 역할) read_set에 추가해준다. 그리고 현재 연결된(ready_set) 모든 client의 명령을 하나씩 수행해주는 echo 함수를 수행한 후에 read_set을 ready_set으로 업데이트 해주고 select함수를 불러 ready된 file descriptor를 connect해준다. 그 후 계속 반복문이 돌며 echo 함수가 수행되고, 다시 ready_set을 업데이트 해주고를 반복하며 service해주게 될 것이다.

  ✓ stock info에 대한 file contents를 memory로 올린 방법 설명

  server가 시작이 될 때 server는 stock info가 저장이 되어있는 stock.txt로부터 stock info를 읽어온다. 그리고 해당 정보들을 읽을 때마다 한 노드씩 동적할당 해주며 이진 트리에 저장하게 된다. 저장 기준은 stock의 ID가 된다. 마지막 client와의 connection이 끝나는 순간에 stock.txt를 업데이트해주게 된다. 그렇게 stock.txt를 업데이트해주는 이유는 server가 종료될 때 persistent한 info를 유지하기 위함이다.

- **pthread**

  ✓ pthread로 구현한 부분에 대해서 간략히 설명

  먼저 client들로부터 연결을 받기 전에 20개의 thread를 생성해놓는다. 각각의 thread는 thread함수를 수행하는 중이다. 그리고 connection이 올 때마다 해당 fd는 sbuf에(listenfd의 역할) 추가했다가 thread 함수가 sbuf에서 fd를 remove하며 connect를 하고, 명령들을 service하게 된다.

### C. 개발 방법

### 1. load_stock_table 함수

```
int load_stock_table(){//stock.txt로부터 tree를 세팅한다
    FILE* input;
    char line[100];
    input=fopen("stock.txt","r");
    head=NULL;
    while(fgets(line,100,input)!=NULL){//한 줄씩 읽어서 node setting
        int ID,left_stock,price;
        line[strlen(line)-1]='\n';
        sscanf(line,"%d %d %d\n",&ID,&left_stock,&price);
        insert_node(ID,left_stock,price);//tree에 node insert
    }
    fclose(input);
    return 0;
}
```

stock.txt 파일로부터 stock의 정보를 읽어온 후 이진 트리에 업데이트해주는 함수이다. 한 줄씩 읽은 후 해당 정보를 저장한 뒤 insert_node함수를 통해 tree에 ID를 기준으로 insert해주게 된다.

### 2. pool 구조체 (event-based)

```
typedef struct{//connfd들 저장하는 자료구조
    int maxfd;//largest descriptor
    fd_set read_set;//active한 descriptor들
    fd_set ready_set;//읽을 준비 된 descriptor들
    int nready;//ready 된 fd 갯수
    int maxi;//highest index
    int clientfd[FD_SETSIZE];//connfd 값 저장하는 배열
    rio_t clientrio[FD_SETSIZE];//connfd당 buffer 저장하는 배열
}pool;
```

event-based server의 구현을 위해 file descriptor들의 정보를 저장하게 되는 구조체이다. read_set은 listenfd의 역할을 하며 connect요청을 한 fd들을 저장하게 된다. select 함수를 통해 read_set의 fd들은 ready_set으로 들어가게 되고, ready_set 안에 있는 fd들은 connect되어 각 client별로 echo함수에서 명령을 받아 수행해주게 된다.

- ### 3. sbuf 구조체

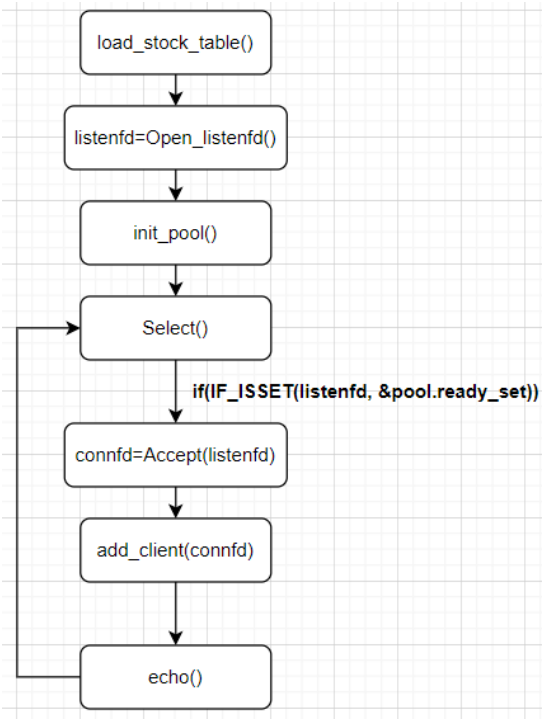
```
typedef struct{
    int *buf;//fd값 저장하는 buf
    int n;
    int front;
    int rear;
    sem_t mutex;
    sem_t slots;//buf 빈 공간
    sem_t items;//buf안의 fd 개수
}sbuf_t;
```

thread-based server의 구현을 위해 listenfd의 역할을 하게 되는 구조체이다. buf 배열에 connect요청을 한 clinet들의 sbuf_insert 함수를 통해 fd값이 저장되게 되며 미리 생성되어있던 thread 함수에서 buf배열에서 하나씩 sbuf_remove함수를 통해 remove되며 연결이 되게 된다. concurrent한 수행, producer-consumer방식의 구현을 위해 세마포어들을 둔다.

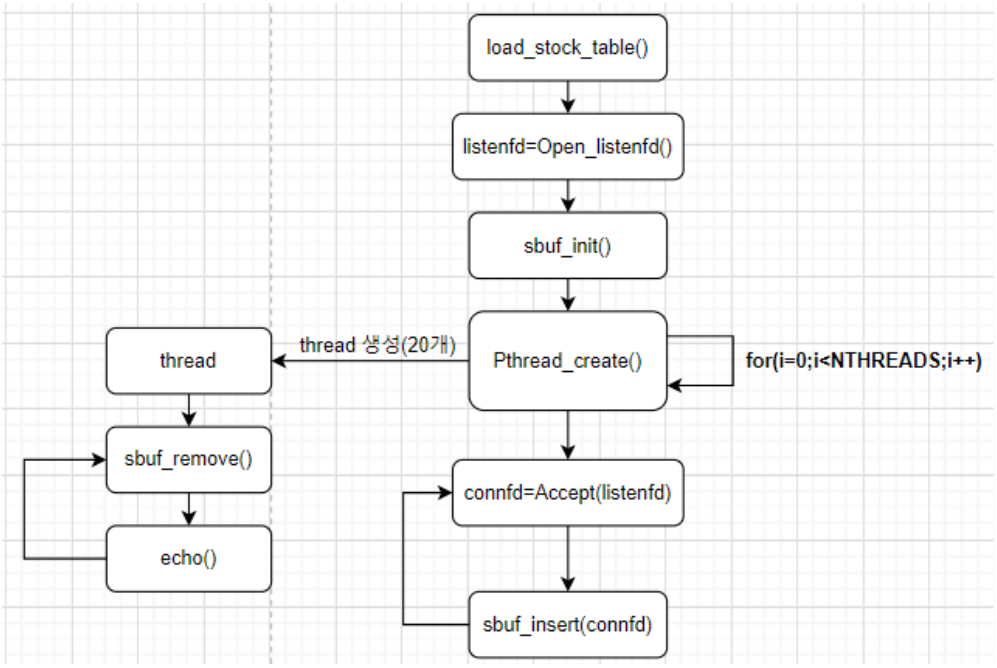
## 3. 구현 결과

### A. Flow Chart

#### 1. select




#### 2. pthread



## B. 제작 내용

### 1. select

event-based concurrent server의 경우 뼈대를 구성해놓고 나니 flow 자체도 sequential하기 때문에 명령 구현의 부분이 그렇게 어렵진 않았다.
위에서 설명한대로 pool구조체를 이용해 미리 생성되어있는 clientfd 배열에 connfd들을 accept받아서 연결 시키면 될 것이다. 이 문장이 한번 수행 될 때마다 echo함수가 호출되어 maxi와 nready 변수에 따라 i=0~maxi까지 p->nready>0이라면 해당 connfd로부터 Rio_readnb함수를 호출하여 명령을 읽어온 후, 명령들을 수행하게 된다. 각 client마다 명령은 하나씩만 수행하게 되며, 모든 conneceted client에서 명령을 한번씩 수행하였으면 다시 ready_set을 update하고 echo함수를 호출하는 과정을 반복하게 된다.

```
if(strcmp(cmd,"show")==0){//show
    InorderSP(head,show);
}
```

show 명령이 들어오면 InorderSP함수를 호출해주었다. event-based의 경우 sequential한 수행이므로 context-switch도 고려해줄 필요가 없었다.
InorderSP함수는 tree를 중위순회하며 stock의 정보를 show 문자열에 update해준다.

```
else if(strcmp(cmd,"buy")==0){//buy
    sscanf(buf,"%*s %d %d\n",&SID,&SCNT);
    if(InorderFIND(head,SID,SCNT,0)==1)//성공
        sprintf(show,"[buy] success\n");
    else//실패
        sprintf(show,"Not enough left stock\n");
}
else if(strcmp(cmd,"sell")==0){//sell(sell 명령은

    sscanf(buf,"%*s %d %d\n",&SID,&SCNT);
    InorderFIND(head,SID,SCNT,1);
    sprintf(show,"[sell] success\n");
}
```

buy, sell 명령이 들어오면 InorderFIND함수를 호출해주었다. buy, sell 명령간의 구분은 flag를 parameter로 두어 0이면 buy, 1이면 sell이 되도록 하였다. 마찬가지로 context-switch를 고려할 필요가 없으므로 InorderFIND함수에서 tree를 중위순회하며 해당 ID의 tree node가 나오면 알맞게 남은 주식 수를 빼고 더하게 하였다. buy일 경우 남은 주식 수가 빼야 하는 주식 수보다 적다면 0을 return해 buy 명령이 수행되지 않도록 구현하였다.

```
else if(strcmp(cmd,"exit")==0){//exit
    Close(connfd);
    client_cnt--;
    FD_CLR(connfd, &p->read_set);
    p->clientfd[i]=-1;
    if(client_cnt==0)
        update_stock_txt();//stock.txt 업데이트
    return ;
}
```

exit 명령의 경우 곧바로 connfd를 close하고, read_set에서 connfd를 빼준 후 pool에 fd값도 -1로 설정해주었다. 그리고 마지막 client 일 경우 stock.txt를 업데이트해주도록 구현하였다. 이런 식으로 구현하면 모든 client가 종료 되었을 때 persistent하게 주식 정보가 유지될 것이다.
모든 명령이 수행 된 이후에(exit)제외 update 하였던 show 문자열을 client에 Riowriten함수를 통해 전달하도록 구현하였다.

```
Rio_writen(connfd,show,sizeof(show));
```

2. **pthread**

thread-based concurrent server의 경우 뼈대 자체는 event-based concurrent server과 거의 비슷하다. 하지만 pthread의 경우 이번엔 thread를 사용해 context switching을 하며 concurrent하게 logical flow를 구성하기 때문에, binary tree에 접근하고 값을 수정할 때 필요하다면 세마포를 사용해 lock을 걸어 더 정확한 concurrency를 보장해줄 필요가 있었다.
함수 호출이나 구성 같은 부분은 위에서 모두 설명하였기에, 세마포어 부분을 중점적으로 설명해보겠다.

```
void *thread(void *vargp){
    Pthread_detach(pthread_self());
    while (1){
        int connfd=sbuf_remove(&sbuf);//sbuf로부터 connfd 읽어온다
        echo(connfd);//client 명령 처리
        Close(connfd);
        /*connection이 종료되면 update_stock_txt함수를 호출해 stoc
k.txt를 update 해준다. 이 때 이 과정을 여러개의 thread가 수행하면 >
안 되므로 lock을 걸어준다.*/
        P(&stocktxt);
        client_cnt--;
        if(client_cnt==0)
            update_stock_txt();
        V(&stocktxt);
    }
    return NULL;
}
```

먼저 시작할 때 배정된 20개의 thread들이 concurrent하게 수행하게 될 thread 함수이다. sbuf엔 listenfd로부터 accept가 된 connfd들이 저장되어 있다. 먼저 이를 sbuf 구조체에서 remove시키며 해당 fd값을 return해주는 sbuf_remove함수를 호출하고, connfd를 통해 client를 service해주는 echo함수가 수행된다. exit명령을 받아 connect가 종료되면 connfd가 끊기고 마지막 client일 경우 stock.txt를 update해주어야 하는데 여기서 세마포를 사용해 해당 함수에서 lock을 걸어준다. 동시에 여러 개의 thread 함수가 종료

될 때, stock.txt의 interity를 보장하기 위함이다. 이렇게 구현할 경우 모든 connect가 종료 됐을 때 persistent하게 주식 정보가 유지될 것이다.
다음은 echo함수에서 show 명령을 받을 경우 실행되게 될 코드이다.

```
if(strcmp(cmd,"show")==0){//show
    InorderLOCK(head);//tree의 모든 node를 lock 해준다.
    InorderSP(head,show);//tree의 node를 출력해준다. 출력할
 node의 lock을 풀어준다
}
```

event-based의 경우와 달리 InorderLOCK부분이 추가 되었다.

```
void InorderLOCK(stock_pointer move){//show할 문자열을 세팅해주는 >
함수인 InorderSP를 호출하기 전에 모든 node에 대해 readcnt++를 해주>
고 필요하다면 w lock을 걸어주는 함수
    if(move!=NULL){
        InorderLOCK(move->left);
        P(&(move->mutex));
        move->readcnt++;
        if(move->readcnt==1)
            P(&(move->w));
        V(&(move->mutex));
        InorderLOCK(move->right);
    }
}
```

show는 Reader-Writers Problem에서 read를 수행한다고 생각할 수 있다. 따라서 show를 하게 될 경우 show가 호출 된 후 해당 node가 출력 되기 전에 buy나 sell등으로 node의 값이 바뀌면 안 된다. 따라서 해당 node의 정보를 출력하는 InorderSP함수가 호출 되기 전 모든 node에 readcnt값에 따라 lock이 걸리도록 하였다.

```
void InorderSP(stock_pointer move,char* output){//show할 문자열 세팅해주는 함수, 중위
e를 탐색해가며 수행한다.
    if(move!=NULL){//현재 node는 w lock이 걸려있는 상태
        InorderSP(move->left,output);
        sprintf(output,"%s%d %d %d\n",output,move->ID,move->left_stock,move->price);
 show에 다음 문장을 넣어준다(개행까지 같이 넣어준다)
        P(&(move->mutex));
        move->readcnt--;//readcnt는 한 thread만 해야하므로 lock을 걸어준다
        if(move->readcnt==0)
            V(&(move->w));//read하는 thread가 없다면 w lock을 풀어준다
        V(&(move->mutex));
        InorderSP(move->right,output);
    }
}
```

InorderSP함수에선 lock이 걸렸을 node가 해당 정보를 sprintf한 후 readcnt를 감소시키며 해당 값에 따라 w lock을 풀어준다. 만약 buy나 sell에서 P(w)로 write하길 기다리고 있었다면 여기서 풀린 후에 write을 할 수 있을 것이다. readcnt 역시 concurrency를 보장받아야하므로 readcnt의 값을 바꿀 때마다 lock을 걸어주었다. 나머지 구현은 event-based server와 동일하다.

```
int InorderFIND(stock_pointer move,int SID,int SCNT,int flag){//flag 0이면 buy, flag 1이면 sell
, 중위순회로 tree 탐색하면서 수행한다.
    int F=0;
    if(move!=NULL){
        F=InorderFIND(move->left,SID,SCNT,flag);
        if(move->ID==SID){
            F=1;
            P(&(move->w));//write는 한 thread만 할 수 있어야 하므로 lock을 걸어준다
            if(flag==0){//buy
                if(move->left_stock<SCNT)//buy 불가능 할 경우
                    F=0;
                else//buy 수행
                    move->left_stock-=SCNT;
            }
            else if(flag==1){//sell
                move->left_stock+=SCNT;
            }
            V(&(move->w));
            return F;//F=0이면 실패, F=1이면 성공
        }
        F=InorderFIND(move->right,SID,SCNT,flag);
        return F;
    }
    return 0;
}
```

buy나 sell이 들어올 경우 node의 값을 바꾸게 되므로 w lock 여부에 따라 바꿔야할 것이다. 만약 show가 호출 되었는데 아직 출력을 하기 전이라 w lock이 걸려있는 상태라면 lock이 풀릴때까지 기다렸다가 수행이 될 것이다. write 하는 중에 또 write를 못 하도록 구현이 될 것이고, 나머지는 event-based server와 구현이 동일하다.

이외에도 sbuf를 insert하고 remove할 때에 세마포를 사용하는 경우들이 있지만 이 경우는 교과서를 참고하였으므로 생략하도록 하겠다. (강의자료에도 설명 되어있음)

thread-based server에서 exit 명령의 경우 event-based server과는 다르게 return을 하여 echo함수만 종료시켜주면 된다. 이후의 수행들이 thread함수에 정의되어있기 때문이다. 어떤 수행을 하는지는 위에 설명이 되어있기에 생략하도록 하겠다.

select와 pthread 방법 모두에 해당되는 부분인데, server, client에서 명령을 읽어오고 결과를 쓸 때 readlineb로 하면 개행이 고려가 되어버려서 show 명령과 같은 경우에 올바르게 출력이 되지 않았다. 따라서 readlineb를 readnb로 고쳐주었고, MAXSIZE로 설정되어있던 parameter도 sizeof(문자열)형식으로 고쳐주어 오류 없이 잘 수행되도록 구현하였다.

```
(n = Rio_readnb(&rio, buf, sizeof(buf))
```

서버에서 client로부터 입력받는 부분

```
Rio_writen(connfd, show, sizeof(show));
```

서버에서 client로 출력하는 부분

## C. 시험 및 평가 내용

select와 pthread에 대해 구현상의 차이점은 select의 경우 multiple client를 concurrent하게 service하는 것처럼 보이나 실제 logical flow는 sequential하기 때문에 concurrent하게 명령을 처리해주는 pthread방법보단 느릴 것이라고 생각하였다. 하지만 사실 client가 몇 명 안될 경우엔 별 차이 없을 것이라고 생각하였고, 실제 사용되는 여러 서버에서처럼 정말 많은 수의 client들을 동시에 처리해주어야 할 경우 시간적인 부분에서(서버의 CPU가 많은 수의 core를 가진다면) pthread 방법이 많은 이득을 볼 수 있을 것이라고 생각하였다.

생각해보면 context-switch로 concurrency를 구현하기에, event와 pthread방법이 큰 차이가 날 것 같지는 않았다. A, B, C의 일을 두 가지 방법이

event: A1->B1->C1->A2->B2->C2->A3->B3->C3

thread: (A1,B1,C1)->(A2,B2,C3)->(A3,B3,C3) (실제론 이렇게 돌아가진 않을 것)

rough하게 보자면 이런 식으로 수행하는 원리지만 사실 core가 여러 개여서 machine level의 concurrency를 사용할 수 있는 것이 아니라면 thread 방법에서 여러 개를 수행하는 것도 context-switch를 통해 일어나는 것이기 때문에 시간은 비슷할 것이기 때문이다. 따라서 매우 많은 양의 client가 아닌 이상 그렇게 큰 차이는 안 날 것이라고 예상하였다.

따라서 실험을 하여서 결과를 비교해보았다.

실험은 1) multiclient 코드를 사용해 client 1, 5, 10, 20개와 connect가 될 경우를 실험하여 확장성을 비교해보았다.

그리고 2) 각 경우에서 모든 client가 write할 경우(buy, sell), read할 경우(show), write과 read를 섞어서 할 경우(buy, sell)로 나누어 워크로드에 따른 분석을 해보았다

좀 더 정확한 분석을 위해 각 경우마다 5번씩 시간을 잰 뒤 평균값을 구하여 값을 측정하였다.

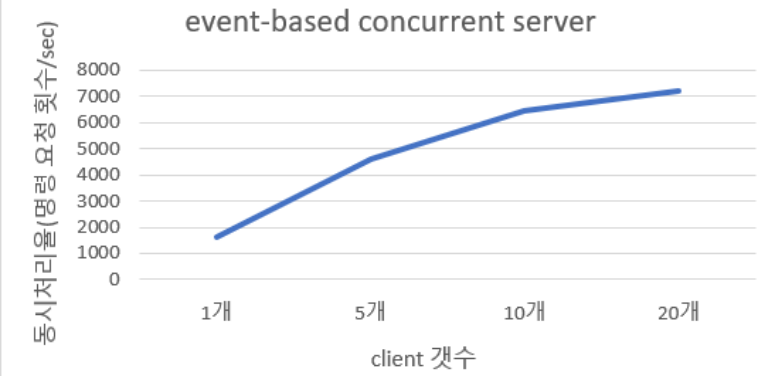
각 client는 10개의 명령을 수행하게 된 후 종료되게 된다. 지정한 client 개수에 대하여 multiclient함수에서 각 client가 10개씩 명령을 수행하고 난 후까지의 시간을 측정하였다.

1) 확장성 비교

먼저 함수 수행한 후 종료까지의 시간을 잰 뒤, client 개수 * 10을 해주고 (client마다 10개의 명령을 요청한다)그것을 수행 시간으로 나누어서 동시처리율을 구해주었다.

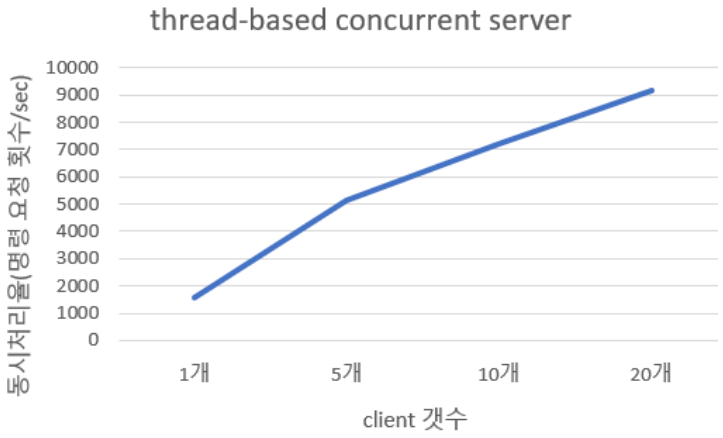
**-event-based concurrent server**

| event-based | | | | |
|---|---|---|---|---|
| client 개수 | 1개 | 5개 | 10개 | 20개 |
| 동시처리율(명령 요청 횟수/sec) | 1613 | 4613.7 | 6453.2 | 7205.2 |
| 수행 시간(msec) | 6.199625 | 10.837367 | 15.496272 | 27.757788 |




**-thread-based concurrent server**

| thread-based | | | | |
|---|---|---|---|---|
| client 개수 | 1개 | 5개 | 10개 | 20개 |
| 동시처리율(명령 요청 횟수/sec) | 1568.3 | 5117.8 | 7191.1 | 9166.8 |
| 수행 시간(msec) | 6.376223 | 9.769795 | 13.906114 | 21.817817 |




각 경우에 따라 요청하는 client의 개수를 1개부터 20개까지 늘려가며 5번씩 시간을 재었다.

예상한대로 client 개수가 적을 때엔 시간이 거의 비슷하였다. 그러나 요청하

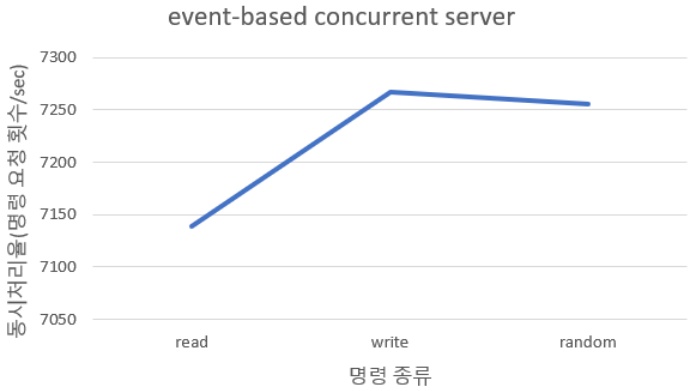
는 client의 개수가 커질수록 수행 시간이 줄어드는 것을 확인 할 수 있었다. 이를 동시처리율로 바꾸어보니 훨씬 더 유의미한 차이인 것을 볼 수 있었다. event-based는 20명의 client를 service할 때 1초에 7200개의 요청을 처리하는 효율을 내지만, thread-based는 1초에 9100개의 요청을 처리하는 효율을 내는 것을 확인할 수 있었다.그래프를 보아도 알 수 있는데, thread-based의 그래프가 훨씬 가파르게 증가하는 것을 볼 수 있었다. 다만 확실히 명세서에 나와있듯이 로그함수의 모양으로 증가하는 것을 볼 수 있었다. 해당 실험을 진행하며 client가 만명, 혹은 더 많은 수의 client가 요청이 오고 해당 client들을 service해야되는 경우 효율적인 면에선 thread-based server가 훨씬 많은 양의 요청을 처리할 수 있을 것임을 알 수 있었다.
이러한 유의미한 차이가 생기는 이유는 event-based는 많은 client을 service할 수 있긴 하지만 logical flow가 sequential한 수행이기에 동시에 여러 client를 정말 동시에 수행해주는 thread-based와 차이가 날 수 밖에 없음을 확인 할 수 있었다.

2) 워크로드에 따른 분석

이번 프로젝트에선 총 4개의 명령을 처리할 수 있게 구현하였다. show, buy, sell, exit이 그것들이다. exit은 server와의 연결을 끊는 것이니까 고려하지 않겠다. buy, sell은 모드 tree의 node에 값을 변경시키는 것이므로 비슷한 종류의 명령(write)으로 보았다. show는 모든 node의 값을 read하기만 하므로 show, (buy, sell)간의 수행 시간, 동시처리율 차이를 분석하여 보았다. client하나당 10개의 요청을 보내게 될 것이고, read만 할 경우, write만 할 경우, read와 write를 random하게 할 경우 총 3가지의 경우로 분석하여 보았다. client 개수는 20개로 지정하였다.

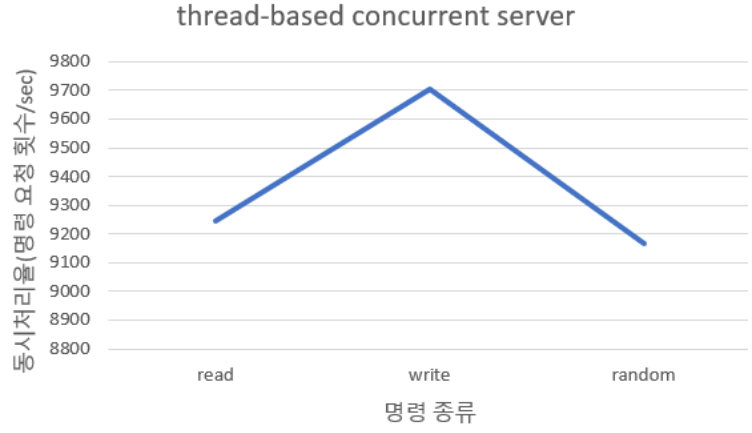
**-event-based concurrent server**

| event-based | | | |
|---|---|---|---|
| 명령 종류 | read | write | random |
| 동시처리율(명령 요청 횟수/sec) | 7138.93 | 7267.68 | 7256.06 |
| 수행 시간(msec) | 28.015399 | 27.5191 | 27.56317 |




**-thread-based concurrent server**

| thread-based | | | |
|---|---|---|---|
| 명령 종류 | read | write | random |
| 동시처리율(명령 요청 횟수/sec) | 9243.06 | 9701.8 | 9166.82 |
| 수행 시간(msec) | 21.637861 | 20.614732 | 21.817817 |



사실 event-based의 경우 코드 구현 상의 부분에 있어서 read와 write의 payload가 달라야할 큰 이유는 찾지 못 했었고, 결과 역시 예상했던 대로 별 유의미한 차이 없이 나온 것으로 해석이 가능하다.
반면, thread-based의 경우 readers-writers problem 때문에 read는 여러 명이 할 수 있어도 write는 한 명 밖에 못 한다. 이로 인해 write 명령이 동시 처리율이 더 작을 것으로 예상했으나, 의외로 read 명령이 더 작은 것을 볼 수 있었다. 세마포 때문인가 생각했지만 random은 그럼에도 더 작은 것을 볼 때에 세마포 때문은 아닌 것 같고, 그냥 함수 자체가 payload가 큰 것 같다. write의 경우엔 값만 한번 대입하면 끝이지만, read의 경우에(show)모든 node를 두번 씩 traverse하며 sprintf를 수행해야한다. 이 때문에 그래프와 같은 차이가 만들어지는 것이라 해석할 수 있었다.

3) 기타 분석
수업 시간에 배운 내용과 거의 일치했던 것 같다. 2번 분석의 경우 예상과는 다르긴 했지만, 함수 구현의 문제였다. 이번 프로젝트를 통해 알게 된 중요한 점은 event-based와 thread-based 의 차이점을 분명힣 알았고 각각의 장단점을 직접 코딩하며 느낄 수 있었다는 점이다.
처음엔 전체적인 함수를 짤 때 모든 connection이 종료 될 때마다 stock.txt를 업데이트하도록 짰었다. 이렇게 하였을 때 client의 개수가 늘어날수록 3배, 5배 정도의 차이로 event-based의 경우가 느려서 왜 그런가를 생각해보았더니, stock.txt를 업데이트를 할 때에 thread-based의 경우 세마포를 사용하긴 하지만 concurrent한 logical flow를 갖기 때문에 stock.txt을 업데이트하는 행동만 못하지 show, buy, sell등 모든 명령들은 수행해줄 수가 있다. 하지만 event-based의 경우 logical flow가 sequential 하기 때문에 그럴 수가 없이 stock.txt 업데이트만 수행할 수 있을 것이다. 이 수행을 매 connection마다 했기 때문에 event-based에서 훨씬 큰 수행 시간이 나올 수 밖에 없었다. 이 오류를 통해 event-based와 thread-based의 수행간의 차이를 분명하게 확인할 수 있는 예시였다고 생각한다. 이후엔 마지막 connection이 종료될 때만 stock.txt를 업데이트하도록 구현하니 합리적인 결과가 나오는 것을 확인 할 수 있었다.