System Programming Project 5

담당 교수 : 박운상 교수님

이름 : 배성현

학번 : 20161595

1. 개발 목표

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

여러 Client들이 동시에 접속할 수 있는 서비스를 제공해주는 Concurrent Stock Server(미니 주식 서버)를 구축한다. 주식 서버는 주식 정보를 저장하고 있으며 여러 Client들과 소통하게 된다. 주식 클라이언트는 주식 서버에게 주식 사기, 팔기, 조회, 주식 장 퇴장을 요청하게 된다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. select

   Select를 구현하게 되면 여러 Client들이 미니 주식 서버에 Connection을 요청할 수 있으며 Server에게 주식 조회, 주식 구매, 주식 판매, 주식 장 퇴장을 요청할 수 있다. 또한 Server는 Select를 통해서 요청이 들어온 fd들을 살펴볼 수 있게 되고, pending input이 있는 fd들만 Manually 요청된 작업(Connection 요청, 주식 조회, 주식 구매, 주식 판매, 주식 장 퇴장)을 처리해줌으로써 여러 Client를 Concurrently 다룰 수 있게 된다.

2. pthread

   pthread를 구현하게 되면 여러 Client들이 미니 주식 서버에 Connection을 요청할 수 있으며 Server에게 주식 조회, 주식 구매, 주식 판매, 주식 장 퇴장을 요청할 수 있다. Main Server thread는 thread pool을 생성해놓고, 어떤 Client로부터 Connection 요청이 들어오게 되면 Connection 요청을 수락하고, thread pool의 thread들 중에서 해당 Client가 요청한 작업(주식 조회, 주식 구매, 주식 판매, 주식 장 퇴장)을 처리해줌으로써 여러 Client를 Concurrently 다룰 수 있게 된다. 또한 여러 thread가 동시에 판매 또는 구매를 요청하더라도 Synchronization을 통해 주식 data가 잘못되게 수정되는 경

우가 없어진다.

B. 개발 내용

- 아래 항목의 내용만 서술
- (기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)
- **select**

✓ select 함수로 구현한 부분에 대해서 간략히 설명

주식서버에서 먼저 listenfd를 select 함수가 감시할 set에 넣어주었다. 그리고 이후 반복문 안에서 select 함수를 통해 해당 set을 감시하도록 하였다. 만약 해당 set안의 listenfd에 해당하는 fd가 셋팅이 된 경우 어떤 Client로부터 Connection 요청이 들어온 것이기 때문에, Connection을 Accept하고 반환 값인 connfd를 감시할 set에 넣어줌으로써 앞으로 해당 Client의 요청(주식 조회, 판매, 구매, 주식 장 퇴장)을 감시할 수 있도록 하였다. 또 만약 해당 set 안의 어떤 client와의 connfd에 해당하는 fd가 셋팅이 된 경우, 해당 fd로부터 요청을 받고 그에 해당하는 작업을 수행할 수 있도록 해줬다. 만약 Client가 Exit을 통해 Connection의 종료를 요청한 경우 해당 connfd를 감시할 set에서 제거해주고, connfd를 close해주었다. 또한 여기서는 Synchronization은 필요 없었는데 그 이유는 하나의 fd의 pending input을 다 처리하기 전까지는 다른 fd의 pending input을 처리할 수 없기 때문에 사실상 Fine grained Concurrency가 아니기 때문이다. 또한 각 Client가 Exit할 때와, Server가 Ctrl+C를 통해 Exit될 때 stock.txt 파일을 업데이트 하도록 하였다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

균형 잡힌 binary search tree 자료구조로 memory에 올렸다. 먼저 파일로부터 stock_id, left_stock, stock_price를 읽어와 list에 저장하였다. list에 저장한 이후에 해당 size만큼의 배열을 만들고 list에 있는 목록들을 배열에 넣어 주었으며 균형 잡힌 binary search tree를 보다 쉽게 만들기 위해서 배열을 오름차순으로 정렬하여 주었다. 그리고 이를 기반으로 divide and conquer알고리즘을 이용하여 균형 잡힌 binary search tree 형태로 memory에 올려주었다.

또한 여기까지의 작업을 위하여 list, 배열을 heap에 할당하여 주었으므로 list, 배열을 나중에 free해주어 불필요한 메모리 낭비를 없앴다.

- **pthread**

  ✓ pthread로 구현한 부분에 대해서 간략히 설명

  먼저 주식 서버의 main thread는 처음에 한번에 많은 worker thread들을 생성하여 worker thread pool을 만들어 놓도록 하였고 이후부터는 Connection을 Accept 하고 connfd를 버퍼에 넣어주는 역할만을 수행해주도록 하였다. 이 때 버퍼는 producer와 consumer의 synchronization 개념을 도입하여, main thread가 connfd에 대한 producer의 역할을, worker thread들이 consumer의 역할을 수행하도록 하였다. 만약 어떤 Client로부터 Connection 요청이 들어온 경우 main thread에서는 이를 accept하고 connfd를 buffer에 넣어주게 된다. 그리고 suspend상태에 있는 worker thread중 하나가 suspend 상태에서 깨어나 해당 Client에게 connfd를 통해 Service를 제공해줄 수 있도록 하였다. 각 worker thread는 thread 함수를 실행하게 되는데, thread가 끝났을 때 kernel에 의해 reaping될 수 있도록, detach모드로 하였다. 또한 thread가 Client의 요청(주식 조회, 판매, 구매, 주식 장 퇴장)을 Deal하게 되는데, 판매와 구매, 조회의 경우에는 Shared Data(주식)에 접근하기 때문에 판매와 구매는 Writer, 조회는 Reader로 하여 각 주식 종목마다 First-Reader-Writers Synchronization을 적용하여 주었다. 이 때 전체 주식 종목(전체 트리를)을 하나의 semaphore로 두게 되면 서로 다른 주식 종목에 접근하고자 하는 Client들에 있어 효율성이 떨어지기 때문에 각 주식 종목마다 semaphore를 만들어 주었다. 또한 Client가 Exit하거나 Server가 Ctrl+C를 통해 Exit할 때에 stock.txt를 업데이트해주도록 하였는데 stock.txt도 공유 자원이기 때문에 Synchronization을 통해 한번에 하나의 Thread만 파일에 접근할 수 있도록 하였다.

C. 개발 방법

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

- **select**

  ✓ select 함수

  먼저 select가 감시하는 fd들을 따로 저장할 수 있도록 하는 fd들의 list를 만들어 줄 것이다. 이를 위해서 fd_item(int fd, fd_item* link)에 대한 구조체를 만들어 줄 것이고, fd list에 감시할 fd를 추가하는 fd_add함수와 fd list에서 connection이 종료된 fd를 제거하는 fd_delete함수를 만들어줄 것이다. 또한 main함수에서 fd_set 타입의 watch_set을 만들어 해당 watch_set에 listen_fd와 각 client의 요청을 accept하여 반환된 connfd들을 감시할 fd로 셋팅하여 주는 코드를 추가할 것이다. 또한 select이후에 fd list를 순회하면서 해당 fd가 fd_set에서 셋팅되어 있는지(pending input이 있는지) 체크하여 셋팅되어 있는 경우 해당 작업을 수행할 수 있도록 할 것이다. 또한 binary search tree를 순회하여 주식종목에 접근 방식의 코드를 통해 각 작업(주식 조회, 판매, 구매)에 대한 함수를 구현할 것이다.

  ✓ stock info에 대한 file contents를 memory로 올릴 방법

  균형 잡힌 binary search tree 자료구조로 memory에 올릴 것이다. 따라서 먼저 각 stock종목을 나타낼 수 있도록 하는 구조체를 만들어줄 것이다. 해당 구조체에는 stock의 id, 남은 stock의 수, stock의 가격 등을 저장 할 수 있도록 할 것이다. 이후 파일로부터 stock_id, left_stock, stock_price를 읽어와 이를 오름차순으로 sorting 하여 줄 것이다. 오름차순으로 sorting하여주는 이유는 sorting 된 array를 통해 보다 쉽게 균형 잡힌 binary search tree를 만들기 위해서이다. 해당 과정을 수행하기 위해서 stock의 전체 개수를 모르기 때문에 먼저 list를 하나 만들어 list에 data를 읽어오고, 해당 list에 있는 값들을 배열로 가져와 qsort함수를 통하여 sorting하여 줄 것이다. 그리고 divide and conquer 알고리즘을 통해 최종적으로 균형 잡힌 binary search tree 자료구조로 메모리에 올려줄 것이다.

- **pthread**

  ✓ pthread

먼저 주식의 종목 하나를 나타내는 item 구조체에 mutex와 writer에 대한 semaphore와 현재 해당 종목을 현재 조회하고 있는 client의 수를 나타내는 변수를 추가하여 줄 것이다. 또한 server가 accept한 connfd들을 저장하는 버퍼와 관련된 구조체를 만들어 줄 것이고, 이러한 buffer를 초기화, buffer에 connfd 추가, buffer에서 connfd 삭제와 관련된 함수를 추가해줄 것이다. 또한 처음에 binary search tree를 통해 주식의 메모리 구조를 만들 때 이러한 semaphore와 변수를 초기화 하여 줄 것이다. 또한 주식 서버의 main함수에 처음에 worker thread pool을 생성하는 코드와 Connection request을 Accept하는 코드, accept 했을 때 connfd를 buffer에 넣어주는 코드를 추가할 것이다. 또한 각 client를 담당하는 worker thread의 동작을 지정하는 thread 함수를 새로 만들어 줄 것인데 해당 thread함수에는 detach 모드로 설정하는 코드와 buffer로부터 connfd를 빼오는 코드, Client의 요청을 받아서 해당하는 작업을 수행하는 함수를 호출하는 코드를 추가할 것이다. 또한 binary search tree를 순회하며 주식종목에 접근하는 방식의 코드를 통해 각 작업(주식 조회, 판매, 구매)에 대한 함수를 구현할 것이다. 이 때 주식 종목의 left_stock 변수는 조회, 판매, 구매를 통해 Race Condition에 빠질 수 있는 변수이기 때문에 주식 조회에서 각 주식 종목의 left_stock에 접근할 때마다 First-Reader-Writer의 Reader에 해당하는 Synchronization 코드를 추가하여 줄 것이다. 마찬가지로 판매와 구매에 해당하는 함수 또한 해당 주식 종목의 left_stock에 접근할 때마다 Writer에 해당하는 Synchronization코드를 추가하여 줄 것이다. 또한 file_mutex라는 Global Semaphore를 추가하여 stock.txt를 업데이트하는 함수에 mutually exclusive하게 파일을 update할 수 있도록 하는 코드를 추가할 것이다.

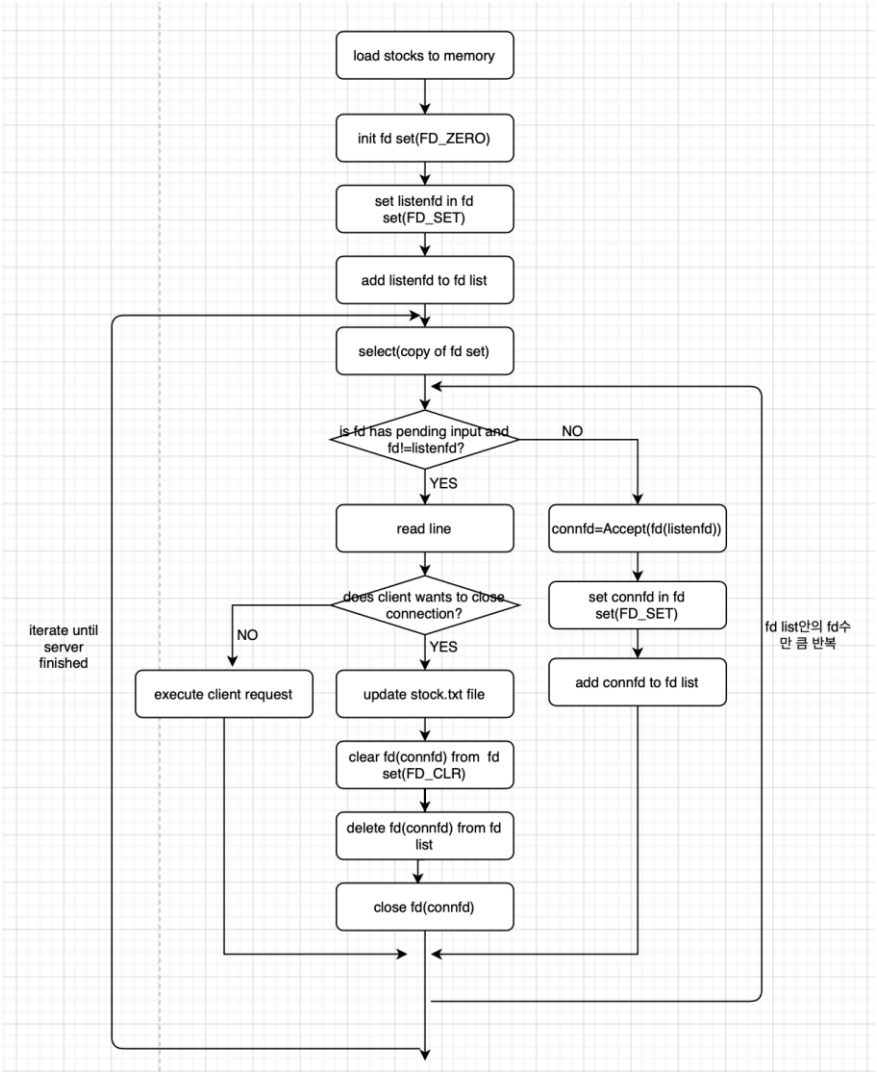
3. 구현 결과

A. Flow Chart

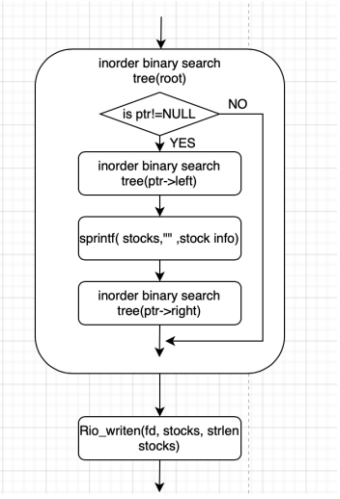
- **2.B.개발 내용에 대한 Flow Chart를 작성.**
- **(각각의 방법들(select, pthread)의 특성이 잘 드러나게 그리면 됨.)**

1. select

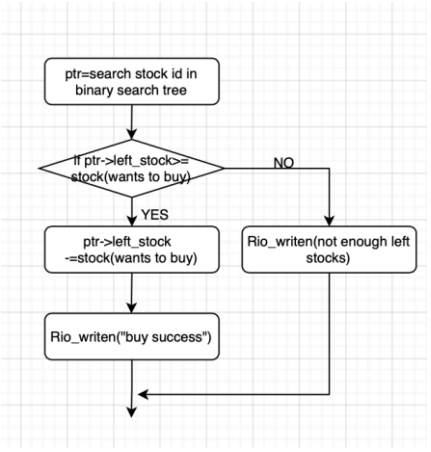
A. select 함수로 구현한 부분



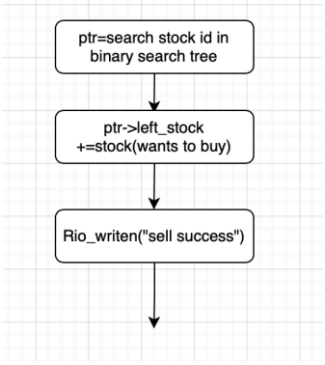
(전체적인 흐름)

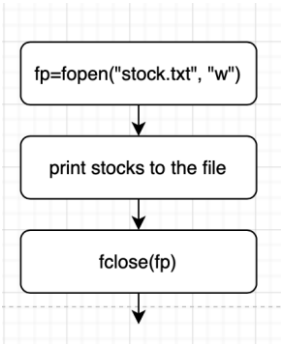


show 함수



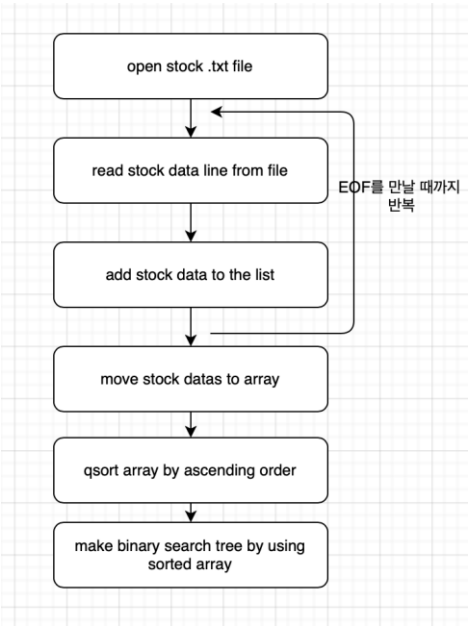
buy 함수



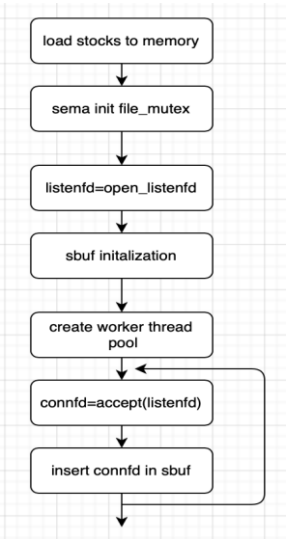


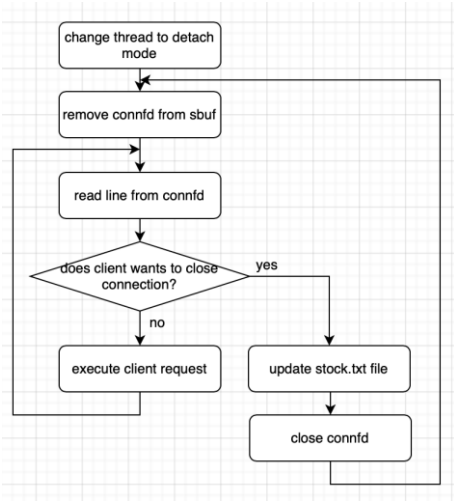
sell 함수 update_file 함수

B. stock info에 대한 file contents를 memory로 올린 방법

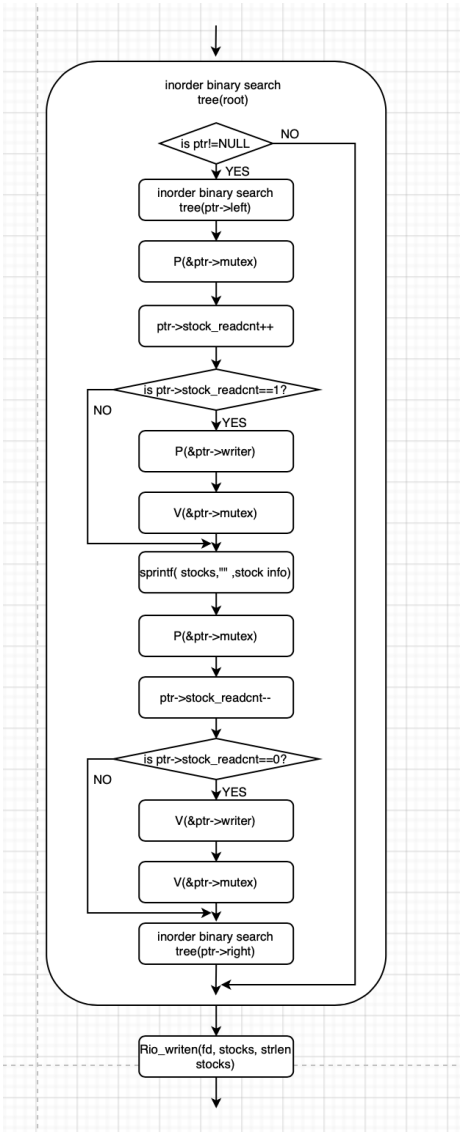


2. pthread

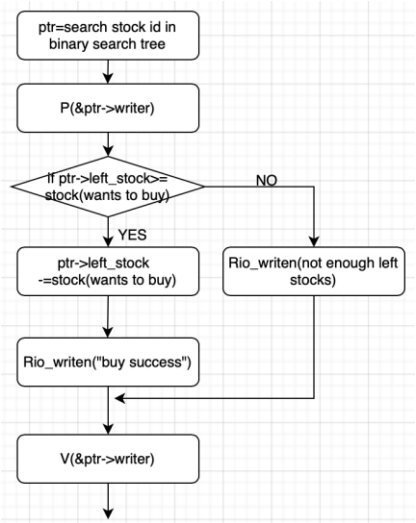




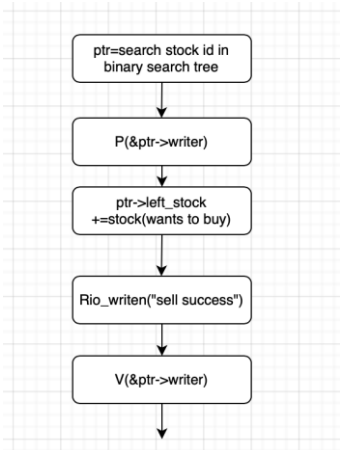
(전체적인 흐름)



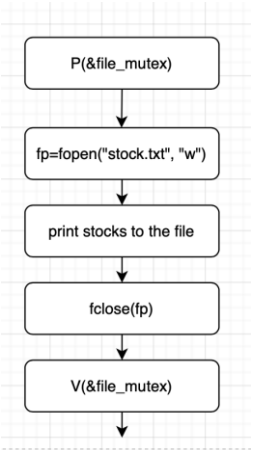
(thread함수)



show함수



buy 함수



sell 함수 update_file 함수

B. 제작 내용

- II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.
- 개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.

1. select

A. select 코드

```
typedef struct fd_item * fd_link;
typedef struct fd_item{
    int fd;
    fd_link next;
}FD_ITEM;

FD_ITEM * fd_head=NULL;//현재 fd set에 관리항목으로 추가된 fd들의 list
FD_ITEM * fd_tail=NULL;
```

먼저 위와 같이 fd_item(int fd, fd_item* link)에 대한 구조체를 만들어 주었다. 그리고 이를 통해 select가 감시하는 fd들을 따로 저장할 수 있도록 하는 fd들의 list를 만들어 주었다.

```
void fd_delete(int fd)
{
    FD_ITEM *ptr;
    FD_ITEM *prev_ptr=fd_head;

    for(ptr=fd_head; ptr!=NULL; ptr=ptr->next)
    {
        if(ptr->fd==fd)//fd list에서 fd를 찾은 경우
        {
            break;
        }
        prev_ptr=ptr;
    }
    if(ptr==fd_head && ptr==fd_tail)//하나밖에 없었던 경우에 이제 fd_list는 비어있게 된다
    {
        fd_head=NULL;
        fd_tail=NULL;
        free(ptr);
    }
    else if(ptr!=fd_head && ptr!=fd_tail)//ptr이 fd list의 중간에 있는 경우
    {
        prev_ptr->next=ptr->next;//ptr의 이전 node가 ptr의 다음 node를 가리키도록함
        free(ptr);
    }
    else if(ptr==fd_head)//ptr이 fd_list의 head인 경우
    {
        fd_head=ptr->next;//fd list의 head가 ptr의 다음 node를 가리키도록 함
        free(ptr);
    }
    else if(ptr==fd_tail)//ptr이 fd의 tail인 경우
    {
        fd_tail=prev_ptr;//fd_list의 tail이 ptr의 이전 node를 가리키도록함;
        fd_tail->next=NULL;
        free(ptr);
    }
}
```

```
void fd_add(int fd)
{
    FD_ITEM * fd_item=(FD_ITEM*)malloc(sizeof(FD_ITEM));
    fd_item->fd=fd;
    fd_item->next=NULL;

    if(fd_head==NULL)//fd list에 아무것도 없으면 head와 tail이 fd_item을 가리킴
    {
        fd_head=fd_item;
        fd_tail=fd_item;
    }
    else
    {
        fd_tail->next=fd_item;
        fd_tail=fd_item;
    }

}
```

그리고 이러한 fd list에 감시할 fd를 추가할 수 있도록 해주는 fd_add함수와 fd list에서 connection이 종료된 fd를 제거하는 fd_delete함수를 위와 같이 만들어 주었다.

```
fd_set watch_set, pending_set;
```

```
pending_set=watch_set;
Select(fd_max+1, &pending_set, NULL, NULL, NULL);
```

main함수의 일부

또한 main함수에서 fd_set 타입의 watch_set과 pending_set을 만들어 감시할 fd들을 watch_set에 넣을 수 있도록 하였고, watch set을 기반으로 select를 통해 pending input을 pending set에 set하도록 하였다.

```
for(ptr=fd_head; ptr!=NULL;)//관리하고 있는 fd_list를 돌면서
```

그리고 감시할 fd의 목록인 fd list를 순회하도록 하였는데

```
if(FD_ISSET(fd, &pending_set) && fd!=listenfd)//fd가 set 되어 있고, access 요
{
    int n;
    Rio_readinitb(&rio, fd);
    command[0]='\0';
    if((n = Rio_readlineb(&rio, command, MAXLINE)) != 0) {//명령이 들어온 경우
        printf("server received %d bytes\n", n);
        if(!strcmp(command, "exit\n"))
        {
            update_file();
            Rio_writen(fd, "exit\n", strlen("exit\n"));
            FD_CLR(fd, &watch_set);//connfd를 watch set에서 없애줌
            fd_delete(fd);//fd list에서 지워줌
            Close(fd);

        }
        else if(!strcmp(command, "\n"))
        {
            Rio_writen(fd, "\n", strlen("\n"));
        }
        else
            execute_command(fd, command);
        //printf("%s", command);
        //Rio_writen(fd, command, n);
    }
    else
    {
        update_file();
        FD_CLR(fd, &watch_set);//connfd를 watch set에 없애줌
        fd_delete(fd);//fd list에서 지워줌
        Close(fd);
        //아무것도 들어오지 않은경우->connection이 종료된 경우
    }
}
```

먼저 특정 fd가 pending_set에 set되어 있고 listenfd가 아닌 경우 특정 client와 연결된 endpoint인 connfd를 의미하는 것이므로, 해당 client의 요청을 fd로부터 읽고 이에 따라 각각 connection종료 작업 또는 execute command(주식 조회, 구매, 판매를 담당하는 함수)을 취해주도록 하였다. 위에서 exit의 경우에는 connection 종료를 의미하므로 stock.txt file을 업데이트 해주도록 하였고, 이제 더 이상 감시하지 않아도 되는 connfd이기 때문에 이를 fd list와 watch_set에서 제거하여 주고 close 해주었다.

```
else if(FD_ISSET(fd, &pending_set)){//fd가 set 되어 있고, access 요청인 경우
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);//요청을 accept
    FD_SET(connfd, &watch_set);//connfd를 watch_set에 set해줌
    fd_add(connfd);//coonfd를 fd list에 추가해줌
    if(fd_max<connfd)
        fd_max=connfd;
}
```

다음으로 fd가 pending_set에 set되어 있고 fd가 listenfd인 경우, 새로운 client가 connection을 request한 경우이므로, 이를 accept하고 감시할 목록인 fd list와 watch set에 connfd를 추가해주도록 하였다.

```c
void execute_command(int fd, char *command)
{
    char order[MAXLINE];//명령
    int stock_id;//주식 id(Char type)
    int stock_num;//주식 개수
    //Rio_writen(fd, command, strlen(command));
    sscanf(command, "%s %d %d", order, &stock_id, &stock_num);
    //printf("%s %d %d\n", order, stock_id, stock_num);
    if(!strcmp(order, "show"))
    {
        show(fd);
    }
    else if(!strcmp(order, "buy"))
    {
        buy(fd, stock_id, stock_num);
    }
    else if(!strcmp(order, "sell"))
    {
        sell(fd, stock_id, stock_num);
    }
    else
    {
        Rio_writen(fd, "invalid command\n", strlen("invalid command\n"));
    }
}
```

execute command함수에서는 위와 같이 명령어의 종류에 따라 각각의 기능(조회, 구매, 판매)을 수행하는 함수로 분기하도록 하였다.

```c
void show(int fd)
{
    char stocks[MAXLINE];
    stocks[0]='\0';
    inorder(stocks, root);
    strcat(stocks, "\n");
    //printf("%s", stocks);
    Rio_writen(fd, stocks, strlen(stocks));
}
```

```c
void inorder(char *stocks, STOCK_ITEM *ptr)
{
    char temp[100];
    if(ptr)
    {
        inorder(stocks, ptr->left);
        //이 사이 thread시에 synchornizae//read
        sprintf(temp, "%d %d %d ", ptr->stock_id, ptr->left_stock, ptr->stock_price);
        strcat(stocks, temp);
        //이 사이 thread시에 synchronize//read
        inorder(stocks, ptr->right);
    }
}
```

주식 조회를 의미하는 show함수의 경우에는 위와 같이 binary search tree를 중위순회하는 방식으로 조회하도록 하였다.

```c
void buy(int fd, int stock_id, int stock_num)
{
    STOCK_ITEM * ptr=root;
    while(ptr)//binary search tree에서 stock_id 검색
    {
        if(ptr->stock_id==stock_id)
        {
            break;
        }
        else if(stock_id<ptr->stock_id)
        {
            ptr=ptr->left;
        }
        else
        {
            ptr=ptr->right;
        }
    }
    if(ptr==NULL)//stock_id가 존재 하지 않음
    {
        Rio_writen(fd, "stock_id not exists\n", strlen("stock_id not exists\n"));
    }
    else//stock_id가 존재하는 경우
    {
        //이 사이 thread 시에 synchronization 필요
        if(ptr->left_stock>=stock_num)//잔여 수량이 남아 있는경우
        {
            ptr->left_stock-=stock_num;
            Rio_writen(fd, "[buy] success\n", strlen("[buy] success\n"));
        }
        else//잔여수량이 남아있지 않는 경우
        {
            Rio_writen(fd, "Not enough left stocks\n", strlen("Not enough left stocks\n"));
        }
        //이 사이 thread 시에 synchronization 필요
    }
}
```

```c
void sell(int fd, int stock_id, int stock_num)
{
    STOCK_ITEM * ptr=root;
    while(ptr)//binary search tree에서 stock_id 검색
    {
        if(ptr->stock_id==stock_id)
        {
            break;
        }
        else if(stock_id<ptr->stock_id)
        {
            ptr=ptr->left;
        }
        else
        {
            ptr=ptr->right;
        }
    }
    if(ptr==NULL)//stock_id가 존재 하지 않음
    {
        Rio_writen(fd, "stock_id not exists\n", strlen("stock_id not exists\n"));
    }
    else//stock_id가 존재하는 경우
    {
        //이 사이 thread 시에 synchronization 필요
        ptr->left_stock+=stock_num;
        Rio_writen(fd, "[sell] success\n", strlen("[sell] success\n"));
        //이 사이 thread 시에 synchronization 필요
    }
}
```

주식 구매, 판매를 의미하는 buy함수와 sell함수는 위와 같이 binary search tree를 stock_id의 대소관계에 따라 비교하여 순회하여 id에 해당하는 node를 찾도록 하였고, 이를 통해 left stock을 업데이트해줄 수 있도록 하였다. 이 때 buy의 경우에는 left_stock이 사려고 하는 것보다 많은 지의 여부를 체크해줄 수 있도록 하였다. 또한 해당 부분에는 synchronization이 필요 없었는데 그 이유는 event based는 fine grained concurrency를 만족하지 못하기 때문에 사실상 순차적으로 도는 것이기 때문이다.

```
void sig_int_handler(int sig)
{
    update_file();
    free_tree(root);
    exit(0);
}
```

또한 server가 종료되는 순간에도 stock.txt 파일을 update 할 수 있도록 하였다.

✓ stock info에 대한 file contents를 memory로 올린 코드

```
typedef struct stock_item * stock_link;
typedef struct stock_item{
    int stock_id;
    int left_stock;
    int stock_price;
    int stock_readcnt;
    sem_t mutex;
    sem_t writer;
    stock_link left;
    stock_link right;
    stock_link next;//임시로 list만들 때 쓰임
}STOCK_ITEM;
```

먼저 각 stock종목을 나타낼 수 있도록 하는 구조체를 위와 같이 만들어 주었다.

```
void load_stock_to_memory()
{
    FILE *fp=fopen("stock.txt", "r");
    int res;
    int stock_id;
    int left_stock;
    int stock_price;
    while(1)
    {
        res=fscanf(fp, "%d %d %d", &stock_id, &left_stock, &stock_price);
        if(res==EOF)
            break;
        stock_add_to_list(stock_id, left_stock, stock_price);//list에 임시로 저장
    }
    stock_list_to_bst();
    fclose(fp);
}
```

이후 stock.txt 파일로부터 stock_id, left_stock, stock_price를 읽어와 이를 각각 stock list에 넣어줄 수 있도록 하였고 또 최종적으로 stock_list_to_bst함수

를 통해 binary search tree로 만들어 주었다.

```c
void stock_list_to_bst()
{
    STOCK_ITEM *stock_arr=(STOCK_ITEM*)malloc(sizeof(STOCK_ITEM)*total_stock_num);
    STOCK_ITEM *ptr=stock_head;
    int i;
    for(i=0; i<total_stock_num; i++)
    {
        STOCK_ITEM* prev_ptr=ptr;
        stock_arr[i].stock_id=ptr->stock_id;
        stock_arr[i].left_stock=ptr->left_stock;
        stock_arr[i].stock_price=ptr->stock_price;
        ptr=ptr->next;
        free(prev_ptr);//array로 옮겨준 뒤 list는 하나씩 제거해줌
    }
    stock_head=NULL;
    stock_tail=NULL;
    //arr를 id 순서대로 sorting 해줌
    qsort(stock_arr, total_stock_num, sizeof(STOCK_ITEM), less);
    //arr를 기반으로 bst 만들어줌
    root=stock_arr_to_bst(stock_arr, 0, total_stock_num-1);

    free(stock_arr);//root(bst)를 만들었으니 array는 free해줌
}
```

```c
STOCK_ITEM* stock_arr_to_bst(STOCK_ITEM *stock_arr, int start, int end)
{
    if(start>end)//재귀를 벗어나는 조건
        return NULL;

    int mid=(start+end)/2;

    STOCK_ITEM *item=(STOCK_ITEM*)malloc(sizeof(STOCK_ITEM));
    item->stock_id=stock_arr[mid].stock_id;
    item->left_stock=stock_arr[mid].left_stock;
    item->stock_price=stock_arr[mid].stock_price;
    item->next=NULL;
    item->stock_readcnt=0;
    Sem_init(&item->mutex, 0, 1);
    Sem_init(&item->writer, 0, 1);

    item->left=stock_arr_to_bst(stock_arr, start, mid-1);
    item->right=stock_arr_to_bst(stock_arr, mid+1, end);

    return item;
}
```

stock_list_to_bst 함수에서는 list를 binary search tree로 만들어 주게 되는데 균형잡힌 tree를 만들기 위해서 이를 array에 넣고 오름차순으로 sorting하게 된다. 그리고 위와 같이 stock_arr_to_bst 함수를 호출하여 해당 sorted array를 divide and conquer 방식을 이용해 binary search tree를 만들게 된다.

2. **pthread**

✓ pthread

```c
typedef struct stock_item{
    int stock_id;
    int left_stock;
    int stock_price;
    int stock_readcnt;
    sem_t mutex;
    sem_t writer;
    stock_link left;
    stock_link right;
    stock_link next;//임시로 list만들 때 쓰임
}STOCK_ITEM;
```

먼저 위와 같이 주식의 종목 하나를 나타내는 item 구조체에 mutex와 writer에 대한 semaphore와 현재 해당 종목을 현재 조회하고 있는 client의 수를 나타내는 변수들을 넣어 주었다.

```c
typedef struct {
    int *buf;
    int n;
    int front;
    int rear;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} sbuf_t;
sbuf_t sbuf;
```

또한 server가 accept한 connfd들을 저장하는 버퍼를 위한 sbuf_t라는 구조체를 위와 같이 만들어 주었다.

```c
void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;
    sp->front = sp->rear = 0;
    Sem_init(&sp->mutex, 0, 1);
    Sem_init(&sp->slots, 0, n);
    Sem_init(&sp->items, 0, 0);
}
void sbuf_deinit(sbuf_t *sp)
{
    Free(sp->buf);
}
```

```c
void sbuf_insert(sbuf_t *sp, int item)
{
    P(&sp->slots);
    P(&sp->mutex);
    sp->buf[(++sp->rear)%(sp->n)] = item;
    V(&sp->mutex);
    V(&sp->items);
}
int sbuf_remove(sbuf_t *sp)
{
    int item;
    P(&sp->items);
    P(&sp->mutex);
    item = sp->buf[(++sp->front)%(sp->n)];
    V(&sp->mutex);
    V(&sp->slots);
    return item;
}
```

그리고 이러한 buffer를 초기화, buffer에 connfd 추가, buffer에서 connfd 삭제와 관련된 함수들을 위와 같이 추가해주었다. buffer는 여러 thread에 의해 접근되고 또 buffer안의 connfd는 maint thread가 produce하고, 여러 worker thread가 consume하기 때문에 buffer에 insert하는 함수(producer)와, remove하는 함수(consumer)에 있어서는 위와 같이 producer-consumer synchronization을 적용하여 주었다.

```
load_stock_to_memory();
Sem_init(&file_mutex, 0, 1);

listenfd = Open_listenfd(argv[1]);
sbuf_init(&sbuf, SBUFSIZE);
for(i=0; i<SBUFSIZE; i++)
    Pthread_create(&tid, NULL, thread, NULL);//처음에 여러개의 thread 만들어놈
while (1)
{
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);//요청을 accept
    sbuf_insert(&sbuf, connfd);
}
```

또한 위의 코드는 main함수의 일부인데 main함수에서는 처음에 stock을 메모리로 올리고, connfd에 대한 버퍼를 초기화 해주고, worker thread pool을 생성해 주도록 하였다. 이후 client의 request를 accept해주고 accept한 이후에 connfd를 buffer에 넣어주는 것을 반복하도록 하였다.

```
void *thread(void *vargp)
{
    int n;
    rio_t rio;
    char command[MAXLINE];
    Pthread_detach(pthread_self());
    while(1)
    {
        int connfd=sbuf_remove(&sbuf);
        while(1)
        {
            int i;
            for(i=0; i<MAXLINE; i++)
                command[i]='\0';
            Rio_readinitb(&rio, connfd);
            if((n = Rio_readlineb(&rio, command, MAXLINE)) != 0)
            {//명령이 들어온 경우
                printf("server received %d bytes\n", n);

                if(!strcmp(command, "exit\n"))
                {
                    update_file();
                    Rio_writen(connfd, "exit\n", strlen("exit\n"));
                    Close(connfd);
                    break;
                }
                else if(!strcmp(command, "\n"))
                {
                    Rio_writen(connfd, "\n", strlen("\n"));
                }
                else
                    execute_command(connfd, command);
            }
            else//client가 connection을 종료한 경우
            {
                update_file();
                Close(connfd);
                break;
            }
        }
    }
}
```

또한 각 client를 담당하는 worker thread의 동작을 지정하는 thread 함수는

위와 같다. 먼저 thread를 detach모드로 설정하도록 하였고, buffer로부터 connfd를 꺼내와 exit이 아닌 경우에는 execute_command함수(select때와 동일)를 호출하여 해당 connfd가 요하는 작업을 수행할 수 있도록 하였다.

```
void inorder(char *stocks, STOCK_ITEM *ptr)
{
    char temp[100];
    if(ptr)
    {
        inorder(stocks, ptr->left);

        P(&(ptr->mutex));
        ptr->stock_readcnt++;
        if(ptr->stock_readcnt==1)//한명이 주식을 읽고 있음
            P(&(ptr->writer));
        V(&(ptr->mutex));

        //Critical Section, Reading
        sprintf(temp, "%d %d %d ", ptr->stock_id, ptr->left_stock, ptr->stock_price);
        strcat(stocks, temp);
        //Critical Section, Reading

        P(&(ptr->mutex));
        ptr->stock_readcnt--;
        if(ptr->stock_readcnt==0)
            V(&(ptr->writer));
        V(&(ptr->mutex));

        inorder(stocks, ptr->right);
    }
}

void show(int fd)
{
    char stocks[MAXLINE];
    stocks[0]='\0';
    inorder(stocks, root);
    strcat(stocks, "\n");
    //printf("%s", stocks);
    Rio_writen(fd, stocks, strlen(stocks));
}
```

show함수의 경우는 위와 같이 inorder를 통해 tree를 순회하면서 주식종목들을 출력하도록 하였다. 각 주식 종목의 left_stock은 다른 thread의 buy나 sell에 의해 수정될 수 있기 때문에 각 주식 종목을 접근할 때마다 다른 buy나 sell을 요청하는 thread들은 접근할 수 없도록 해당 부분을 first reader-writers problem의 reader에 해당하도록 코드를 작성하여 주었다.

```
P(&(ptr->writer));
//Critical Section, Writing
//
if(ptr->left_stock>=stock_num)//잔여 수량이 남아 있는경우
{
    ptr->left_stock-=stock_num;
    Rio_writen(fd, "[buy] success\n", strlen("[buy] success\n"));
}
else//잔여수량이 남아있지 않는 경우
{
    Rio_writen(fd, "Not enough left stocks\n", strlen("Not enough left stocks\n"));
}
//
//Critical Section, Writing
V(&(ptr->writer));
```

buy함수의 일부

```
else//stock_id가 존재하는 경우
{
    P(&(ptr->writer));
    //Critical Section, Writing
    //
    ptr->left_stock+=stock_num;
    Rio_writen(fd, "[sell] success\n", strlen("[sell] success\n"));
    //
    //Critical Section, Writing
    V(&(ptr->writer));
}
```

sell함수의 일부

마찬가지로 buy와 sell에 해당하는 함수 또한 해당 주식 종목의 left_stock을 직접적으로 수정하는 Writer에 해당하기 때문에 해당 작업을 수행하는 동안에 다른 thread에 의해 조회, 판매, 구매가 수행될 수 없도록 first reader-writer problem의 writer에 해당하는 코드를 추가하여 주었다.

```
void update_file()
{

    P(&file_mutex);
    //FILE WRITE, Critical Section
    FILE *fp=fopen("stock.txt", "w");
    inorder_print(root, fp);
    fclose(fp);
    //FILE WRITE, Critical Section
    V(&file_mutex);
}
```

또한 stock.txt를 업데이트 하는 update_file함수의 경우에도, stock.txt를 어떤 thread가 수정할 때 다른 thread에 의해 접근되지 않도록 file_mutex라는 semaphore를 도입하여 해당 부분을 감싸 주었다.

3. client 코드 및 Multiclient코드

```
if(flag==1)//show인 경우
{
    int i;
    for(i=0; i<strlen(buf); i++)
    {
        if(buf[i]=='\n')
            buf[i]='\0';
        if(buf[i]=='\t')
            buf[i]='\n';
    }
    flag=0;
}
if(flag==2)//exit인 경우
{
    Close(clientfd);
    exit(0);
}
```

```
if(option == 0){//show
    strcpy(buf, "show\n");
    Rio_writen(clientfd, buf, strlen(buf));
    Rio_readlineb(&rio, buf, MAXLINE);
    int i;
    for(i=0; i<strlen(buf); i++)
    {
        if(buf[i]=='\n')
            buf[i]='\0';
        if(buf[i]=='\t')
            buf[i]='\n';
    }
}
```

또한 stockclient.c(왼쪽)와 multiclient.c(오른쪽)에 각각 위의 코드들을 추가하여주었는데, show에 대한 명령과 exit에 대한 명령을 수행 해주기 위해서 추가하여 주었다. show의 경우 server에서는 모든 주식 종목의 내용을 한 줄의 버퍼에 넣어서 전송하여 주기 때문에 이를 tab을 기준으로 client단에서 개행을 해주도록 하였다. exit의 경우에는 client에서 server로부터 connection종료에 대한 확인 답을 받은 이후에 clientfd를 close해주도록 하였다.

C. 시험 및 평가 내용

- **select, pthread에 대해서 각각 구현상 차이점과 성능상에 예측되는 부분에 대해서 작성. (ex. select는 ~~한 점에 있어서 pthread보다 좋을 것이다.)**

  client수가 많아짐에 따라 pthread의 효율이 select에 비해 좋아질 것이다.

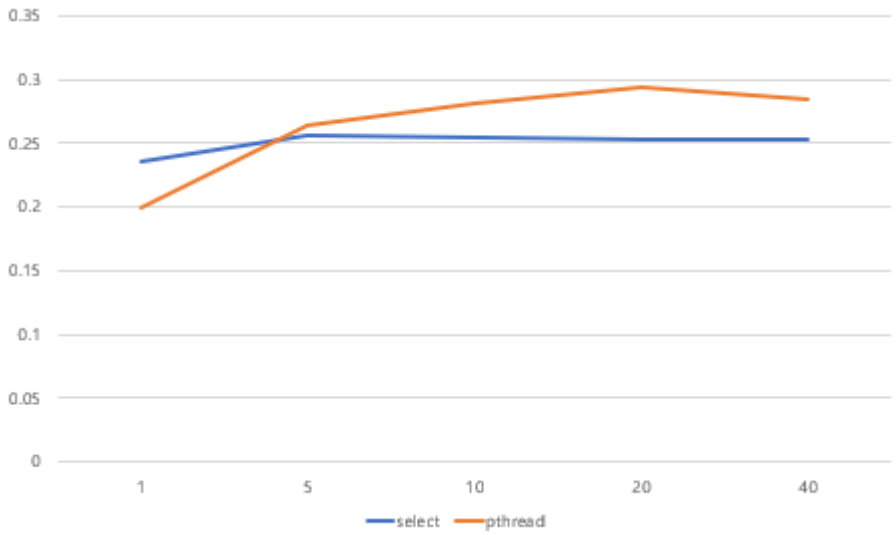
  pthread는 일반적으로 select보다 좋은 성능(시간이 적게 걸릴 것)을 보일 것이다. 왜냐하면 select의 경우에는 fine grained concurrency가 불가능하고 또 multicore의 이점을 살릴 수 없기 때문이다.

  pthread는 select에 비해서 show가 좋은 성능을 보일 것이다. 왜냐하면 show의 경우 reader에 해당하도록 코드를 짰기 때문에 따라서 여러 Client는 동시에 같은 주식 종목에 대한 show에 대해 처리를 받을 수 있어 multicore의 이점을 살릴 수 있는 데에 반해, select의 경우 multicore의 이점을 살릴 수 없고 순차적으로 진행되기 때문이다.

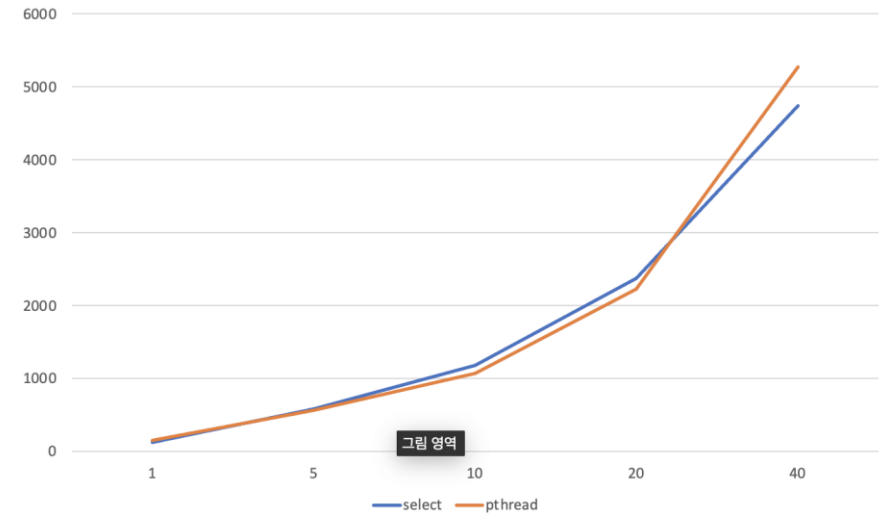
  select는 pthread에 비해서 buy와 sell에 대해 좋은 성능을 보일 것이다. 왜냐하면 pthread의 buy, sell코드의 경우 write에 해당하고 semaphore에 의해 한 번에 하나의 Client만 접근하게 되는 synchronization overhead가 있기 때문이다.

- **실제 실험을 통한 결과 분석 (그래프 삽입)**

  우선 전체 실험의 경우 5개의 주식 종목에, client를 늘려가며 실험(x축)하였고, 이에 대해 전체 결과가 나올 때까지의 시간을 여러 번 측정하여 평균을 냈다. 또한 각 client당 30번의 request를 요청하도록 하였다. 동시 처리율은 아래와 같았다.



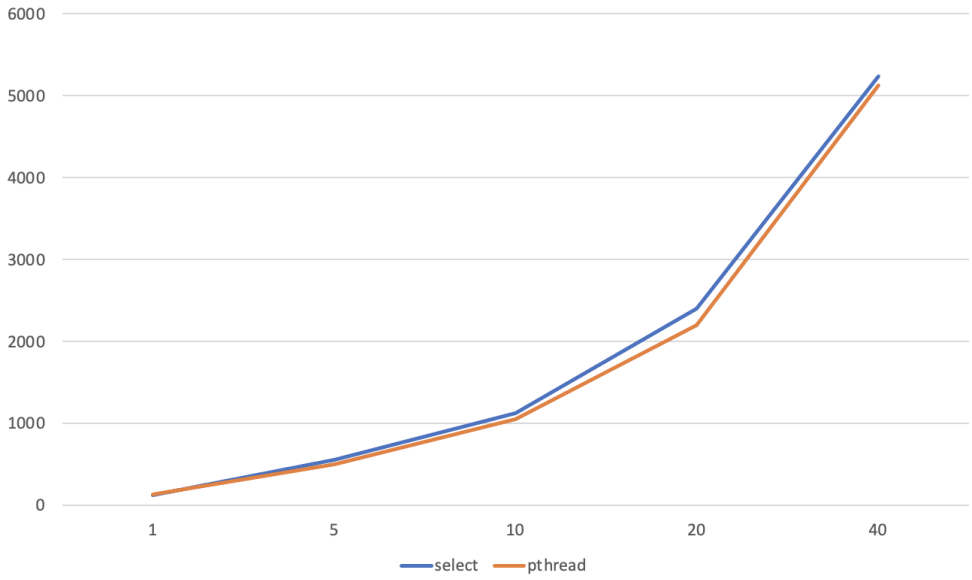
동시처리율(처리한 요청의 개수/시간)을 보면 client수가 많아지면서 pthread의 동시처리율은 조금 향상된 것을 알 수 있다. client의 증가는 thread의 증가를 의미하기 때문에, fine grained concurrency를 통해 pthread의 경우 동시처리율이 조금 향상되었다고 판단할 수 있다. 반면 select의 경우 fine grained concurrency가 아니기 때문에, 순차적으로 각각의 client를 처리해 주어 동시 처리율이 일정한 값을 보이는 것이라고 판단할 수 있다.



	1	5	10	20	40
select	127ms	587	1184	2378	4746
pthread	150ms	567	1070	2226	5278

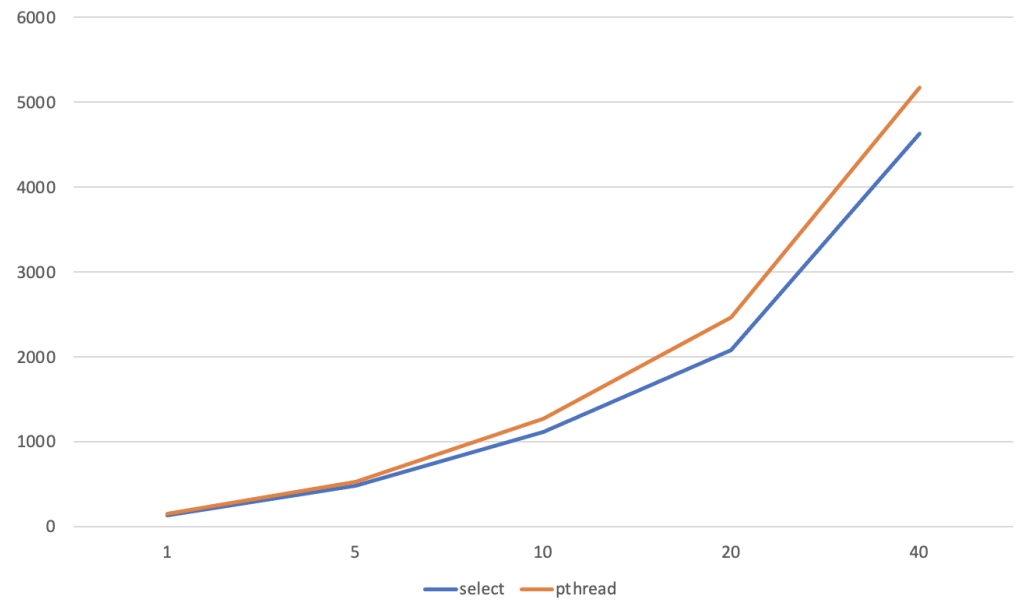
먼저 위의 그래프는 client가 buy, sell, show를 섞어서 요청하는 경우를 실험한 결과이다. 1개의 client만 실행하였을 때에는 select가 thread보다 시간 초가 낮게 나왔는데, 이는 한개의 client만 있을 때에는 thread case의 경우 synchronization에

의한 overhead가 있기 때문이다. 또한 대체적으로 thread가 select보다 시간이 적게 걸린 것을 확인 할 수 있는데, 이는 예측한 결과와 비슷하다고 할 수 있다. 하지만 이는 사실 신뢰할 수 없다고 볼 수 있는데 그 이유는 show, sell, buy를 random하게 뽑았기 때문에, 평균을 내더라도 어떤 case에서는 show가 많을 수도 있고, 어떤 case에서는 buy, sell이 많을 수도 있기 때문이다.



	1	5	10	20	40
select	121	560	1124	2400	5240
pthread	132	503	1057	2201	5123

위의 그래프는 client가 show만을 요청하는 경우를 실험한 결과이다. 전체적으로 보았을 때, 둘 사이의 차이가 별로 나지않는다는 것만 제외하면 예측한 것과 같이 select에 비해 pthread의 시간이 더 적게 걸린 것을 확인할 수 있다. 이는 show의 경우 한번에 여러 thread들이 하나의 주식 종목에 동시에 접근할 수 있기 때문이다. select와 pthread의 시간 차이가 많이 나지 않는 이유에 대해서 생각해보았을 때 thread의 control overhead와 synchronization overhead 등을 생각해 볼 수 있었다.



	1	5	10	20	40
select	133	485	1114	2080	4631
pthread	157	529	1272	2468	5173

위의 그래프는 client가 buy, sell만을 요청하는 경우를 실험한 결과이다. 전체적으로 보았을 때, 예측한 것과 같이 pthread에 비해 select의 시간이 더 적게 걸린 것을 확인할 수 있다. 이는 buy, sell의 경우 writer에 해당하기 때문에 한번에 하나의 thread만이 하나의 주식 종목에 동시에 접근할 수 있어 사실상 순차적으로 진행된다고 볼 수 있는데 이로 인해 synchronization과 관련된 overhead가 생기는데 반해, select의 경우 이러한 overhead가 없기 때문이다.

select와 pthread에 대해 서로 유리한 컨디션(show만 하는 경우, buy, sell만 하는 경우)에 있어서 시간 차이가 많이 날 것으로 생각하였는데 그렇지 아니하였다. 사실 이는 pthread를 구현함에 있어서 synchronization을 적용하는 방법의 차이때문에 생긴 문제라고 판단할 수 있었다. binary search tree를 하나의 변수로 보아 이를 synchronization해주냐와 binary search tree의 각 종목을 하나의 변수로 보아 이를 synchronization을 해주냐의 차이었는데 이번 프로젝트에서 후자의 방법을 적용하였다. 전자의 방법을 적용하게 되면 select와 thread의 차이를 확실히 비교할 수 있을 것인데 그 이유는 하나의 거대한 binary search tree만을 synchronization을 해줌으로써 thread의 synchronization overhead가 훨씬 줄어들게 되기 때문이다. 하지만 이와 같은 처리를 해주게 되면 user가 사거나 팔거나

조회하고 싶은 종목은 a라는 종목인데, 다른 user가 b라는 종목을 사거나, 팔거나, 조회하고 있다는 이유 하나만으로 user는 다른 user의 작업이 끝날 때까지 기다려야한다는 문제가 생기게 된다. 후자의 방법을 이용했을 때 실험에서는 엄청나게 큰 차이를 확인할 수 는 없었는데 이는 실험에서는 random하게 접근할 종목이 선택되므로 각 종목들이 겹칠 확률이 그렇게 크지 않으며, 각 종목에 대해서 모두 synchronization을 처리해주기 때문에 thread의 synchronization overhead가 늘어나게 된다. 하지만 이와 같이 처리를 해주면 user가 사거나 팔거나 조회하고 싶은 종목이 a라는 종목일 때, 다른 user가 b라는 종목을 사거나 팔거나 조회하더라도 user는 a라는 종목에 접근할 수 있게 된다는 장점이 있다.

또한 이번 실험에서 client와 server사이의 요청 처리에 걸리는 시간을 측정하였는데, 실험의 결과는 네트워크 상태(network delay등)에 따라, 또 server를 담당하는 컴퓨터의 상태에 따라 상당히 다른 값을 보일 것이라는 판단 또한 내릴 수 있었다.