System Programming Project 5

담당 교수 : 김영재

이름 : 조보현

학번 : 20160641

1. 개발 목표

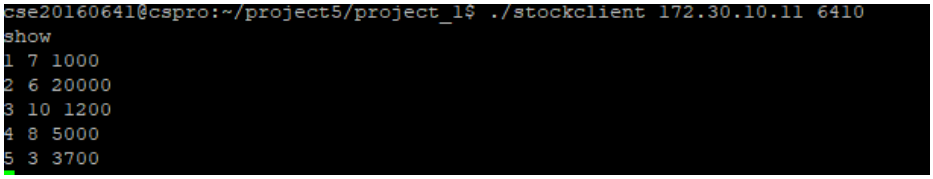
주식 정보를 저장하고 있는 server와 주식 거래, 리스트 출력을 요청하는 client를 구현한다. 특히 Concurrent Stock Server를 구현해 concurrent 하게 주식 거래가 진행될 수 있는 server 구현에 중점을 둔다. Event-driven Approach 와 Thread-based Approach 로 나누어 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

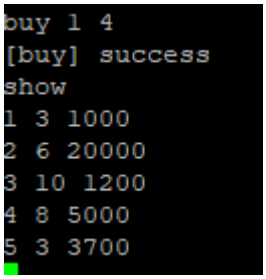
1. select

1) show 명령



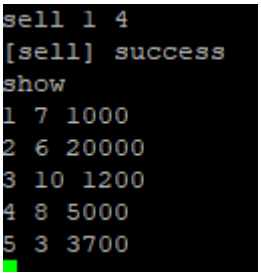
stock list를 적절히 출력하는 것을 확인 할 수 있다.

2) buy 명령



buy 명령이 적절히 수행되는 것을 확인 할 수 있다.

3) sell 명령



sell 명령이 적절히 수행되는 것을 확인 할 수 있다.

4) exit 명령

```
exit
cse20160641@cspro:~/project5/project_2$
```

exit 명령이 적절히 수행되는 것을 확인 할 수 있다.

5) concurrency 확인

```
cse20160641@cspro:~/project5/project_1$ ./multiclient 172.30.10.11 6410 3
child 121364
child 121365
show
sell 1 3
child 121366
sell 2 9
1 7 1000
2 6 20000
3 10 1200
4 8 5000
5 3 3700
[sell] success
[sell] success
show
1 10 1000
2 15 20000
3 10 1200
4 8 5000
5 3 3700
buy 5 1
[buy] success
sell 3 6
[sell] success
sell 5 1
buy 4 7
[sell] success
show
[buy] success
1 10 1000
2 15 20000
3 16 1200
4 1 5000
5 3 3700
buy 2 9
[buy] success
show
1 10 1000
2 6 20000
3 16 1200
4 1 5000
5 3 3700
buy 1 2
[buy] success
show
1 8 1000
2 6 20000
3 16 1200
4 1 5000
5 3 3700
buy 5 7
Not enough left stocks
buy 2 5
[buy] success
```

concurrent 하게 명령들이 실행되는 것을 확인 할 수 있다.

2. pthread

1) show 명령

```
cse20160641@cspro:~/project5/project_2$ ./stockclient 172.30.10.11 6410
show
1 10 1000
2 5 20000
3 6 1200
4 6 5000
5 12 3700
```

show 명령이 적절히 동작하는 것을 확인 할 수 있다.

2) buy 명령

```
buy 1 3
[buy] success
show
1 7 1000
2 5 20000
3 6 1200
4 6 5000
5 12 3700
```

buy 명령이 적절히 동작하는 것을 확인 할 수 있다.

3) sell 명령

```
sell 2 2
[sell] success
show
1 7 1000
2 7 20000
3 6 1200
4 6 5000
5 12 3700
```

sell 명령이 적절히 동작하는 것을 확인 할 수 있다.

4) exit 명령

```
exit
cse20160641@cspro:~/project5/project_2$
```

5) concurrency 확인

```
cse20160641@cspro:~/project5/project_2$ ./multiclient 172.30.10.11 6410 3
child 57544
child 57545
buy 4 8
show
1 16 1000
2 1 20000
3 6 1200
4 8 5000
5 7 3700
child 57546
sell 1 3
[buy] success
[sell] success
buy 4 2
Not enough left stocks
buy 2 9
Not enough left stocks
buy 1 9
[buy] success
buy 2 4
show
Not enough left stocks
1 10 1000
2 1 20000
3 6 1200
4 0 5000
5 7 3700
sell 2 4
[sell] success
sell 5 5
show
[sell] success
1 10 1000
2 5 20000
3 6 1200
4 0 5000
5 12 3700
sell 4 6
[sell] success
show
1 10 1000
2 5 20000
3 6 1200
4 6 5000
5 12 3700
show
1 10 1000
2 5 20000
3 6 1200
4 6 5000
5 12 3700
show
1 10 1000
2 5 20000
3 6 1200
4 6 5000
5 12 3700
```

concurrent 하게 명령들이 실행되는 것을 확인 할 수 있다.

B. 개발 내용

- **select**

✓ select 함수로 구현한 부분에 대해서 간략히 설명

select 함수로 어떤 descriptor(connfd`s or listenfd)가 pending inputs 를 갖는지 결정한다. 만약 listenfd가 input이 있다면 accept connection을 하고 배열에 connfd를 추가한다. 이후 모든 connfd들을 pending input 처리한다. 이때 client service 는 command()함수로 처리한다. command() 함수에 대한 설명은 개발 방법 참고.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

stock info 는 stock.txt 에 저장되어 있다. 이를 tree_init() 이라는 함수를 이용해 이진 탐색 트리 형태로 memory 에 올린다. tree_init() 함수에 대한 설명은 개발 방법 참고.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

define 한 NTHREADS 수 만큼의 thread를 만들어 각 thread가 client 들과 연결되어 처리하도록 한다. 이 때 Reader-Writer 문제를 해결하기 위해 semaphore를 활용한다.

C. 개발 방법

- **select**

(1) stockserver.c

1) pool 구조체

```
typedef struct{  // Represent a pool of connected descriptors
    int maxfd;  // Largest descriptor in read_set
    fd_set read_set;    // Set of all active descriptors
    fd_set ready_set;   // Subset of descriptors ready for reading
    int nready; // Number of ready descriptors from select
    int maxi;   // High water index into client array
    int clienfd[FD_SETSIZE];    // Set of active descriptors
    rio_t clientrio[FD_SETSIZE];    // Set of active read buffers
} pool;
```

Select 함수 구현을 위한 pool 구조체 선언. 자세한 내용은 주석과 같다.

2) Node 구조체

```
typedef struct _Node{
    int ID; // stock ID
    int left_stock; // left stock count
    int price;  // stock price
    struct _Node* lchild;   // left child
    struct _Node* rchild;   // right child
}Node;
```

주식 정보를 저장할 트리를 구성하는 노드 구조체. stock ID를 저장한 ID, 잔여 주식량을 나타내는 left_stock, 주가를 나타내는 price 등이 있다.

3) void init_pool(int listenfd, pool *p)

pool 구조체를 초기화하는 함수.

4) add_client(int connfd, pool *p)

client 를 배열에 추가하는 함수.

5) check_clients(pool *p)

계속해서 연결된 client(connfd)들을 조사하여 input이 있다면 command()함수를 불러 처리하는 함수. EOF를 감지하면 pool에서 descriptor를 제거하는 역할도 맡고 있다.

6) void init_tree()

주식 정보를 저장할 트리를 초기화하는 함수. "stock.txt" 에 저장된 내용을 불러들여 트리 구조로 만든다.

7) Node* insert_tree(Node* root, int ID, int left_stock, int price)

이진 탐색 트리에 Node를 추가하는 함수. Stock ID를 기준으로 추가한다.

8) Node* search_tree(Node* root, int ID)

이진 탐색 트리를 탐색하여 원하는 노드를 반환하는 함수.

9) void command(char* buf, int connfd, int n, pool *p)

client에게서 받은 명령을 처리하는 함수. buf에 명령이 담겨져 파라미터로 주어진다. 각 명령은 예외 없이 제대로 입력된다고 가정한다. 따라서 공백이 하나라도 있는 명령의 경우에는 buy, sell 중 하나 이고 없다면 show, exit 중 하나이다. 이를 토대로 show 의 경우에는 이진탐색트리의 모든 노드의 내용을 형식에 맞춰 char 배열 airplane[MAXBUF] 에 담아 client 에게 전송한다. client 측에서 한 줄 씩 입력을 읽기 때문에 이런 방식을 채택했다. buy 의 경우 buy_stock()함수를 불러 잔여 주식이 있는 경우와 없는 경우를 나누어 결과를 client에게 전송한다. sell 의 경우 sell_stock()함수를 불러 해당 주식이 있는 지 확인하고 잔여 주식량을 증가시킨후 결과를 client에게 전송한다. exit의 경우 client와의 연결이 끊어진다.

10) int buy_stock(int ID, int cnt)

이진 탐색 트리를 조사해 해당 ID를 갖는 주식의 잔여 주식량이 요청한 주문량을 충족시킨다면 해당 잔여 주식량을 주문량만큼 감소시킨후 1을 return 한다. 그렇지 않다면 0을 반환한다.

11) sell_stock(int ID, int cnt)

이진 탐색 트리를 조사해 해당 ID를 갖는 주식의 잔여 주식량을 요청한 주문량만큼 증가시킨후 1을 반환한다.

12) void update(Node* root)

command() 명령을 수행한 후 변경 된 이진탐색트리를 “stock.txt” 에 저장하는 함수.

(2) stockclient.c

exit 명령을 사용자가 입력할 경우 해당 clientfd를 close 하고 종료되도록 변경했다. 또한 한줄로 받은 입력 중에 ‘|’ 이 포함된다면 이를 줄바꿈 ‘₩n’ 으로 바뀌도록 변경했다. 이는 show 명령의 응답으로 트리 전체를 받을 때의 편의성을 위해 임의로 변경한 부분이다.

(3) multiclient.c

한줄로 받은 입력 중에 '|' 이 포함된다면 이를 줄바꿈 '₩n' 으로 바뀌도록 변경했다. 이는 show 명령의 응답으로 트리 전체를 받을 때의 편의성을 위해 임의로 변경한 부분이다. exit 명령은 입력되지 않으므로 따로 구현하지 않았다.

- **pthread**

(1) stockserver.c

1) sbuf_t 구조체

```
typedef struct{
    int *buf;       /* Buffer array*/
    int n;          /* Maximum number of slots */
    int front;      /* buf[(front+1)%n] is first item*/
    int rear;       /* buf[rear%n] is last item*/
    sem_t mutex;    /* Protects accesses to buf*/
    sem_t slots;    /* Counts available slots*/
    sem_t items;    /* Counts available items */
}sbuf_t;
```

Reader-Writer 문제를 해결하기 위한 sbuf_t 구조체. 자세한 설명은 주석과 같다

2) Node 구조체 – select와 동일

3) void sbuf_init(sbuf_t *sp, int n)

empty, bounded, shared FIFO buffer를 n개의 slots으로 초기화하는 함수.

4) void sbuf_deinit(sbuf_t *sp)

sp buffer를 clean up 하는 함수.

5) void sbuf_insert(sbuf_t *sp, int item)

shared buffer sp 의 rear 에 item을 추가하는 함수. 여기서 item은 connfd 가 된다.

6) int sbuf_remove(sbuf_t *sp)

sp 의 첫번째 item을 반환하는 함수.

7) void* thread(void *vargp)

각 thread의 기능을 구현하는 함수. 여기서는 connfd 마다 command() 함수를 불러 처리하게 구현했다.

8) static void init_echo_cnt(void)

mutex를 초기화하는 함수.

9) void init_tree() – select와 동일

10) Node* insert_tree(Node* root, int ID, int left_stock, int price) – select와 동일

11) Node* search_tree(Node* root, int ID) – select와 동일

12) void command(int connfd)

각 thread 에서 client로 부터 입력받은 명령을 처리하는 함수. Readers-writers 문제를 처리하기 위해 semaphore를 활용한다. 이를 제외하고는 select 와 동일하다.

13) int buy_stock(int ID, int cnt) – select와 동일

14) sell_stock(int ID, int cnt) – select와 동일

15) void update(Node* root) – select와 동일

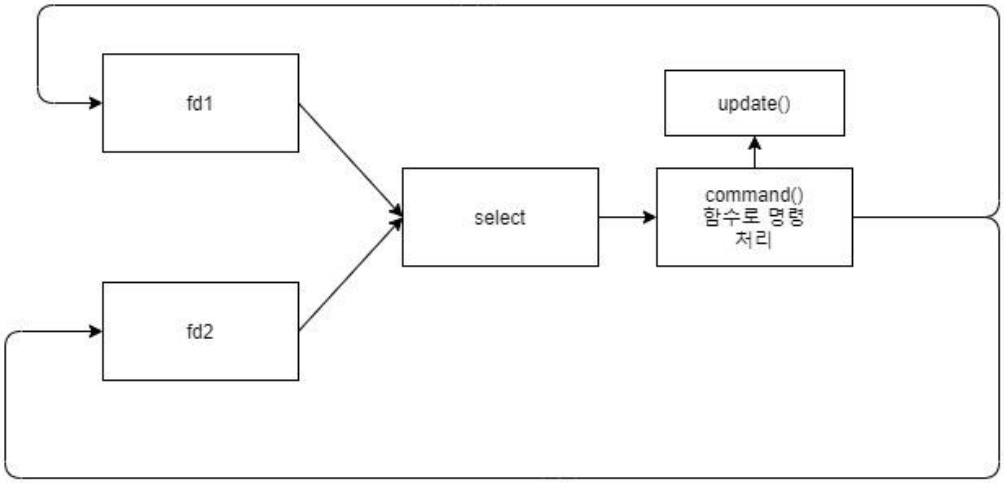
(2) stockclient.c – select와 동일

(3) multiclient.c – select와 동일

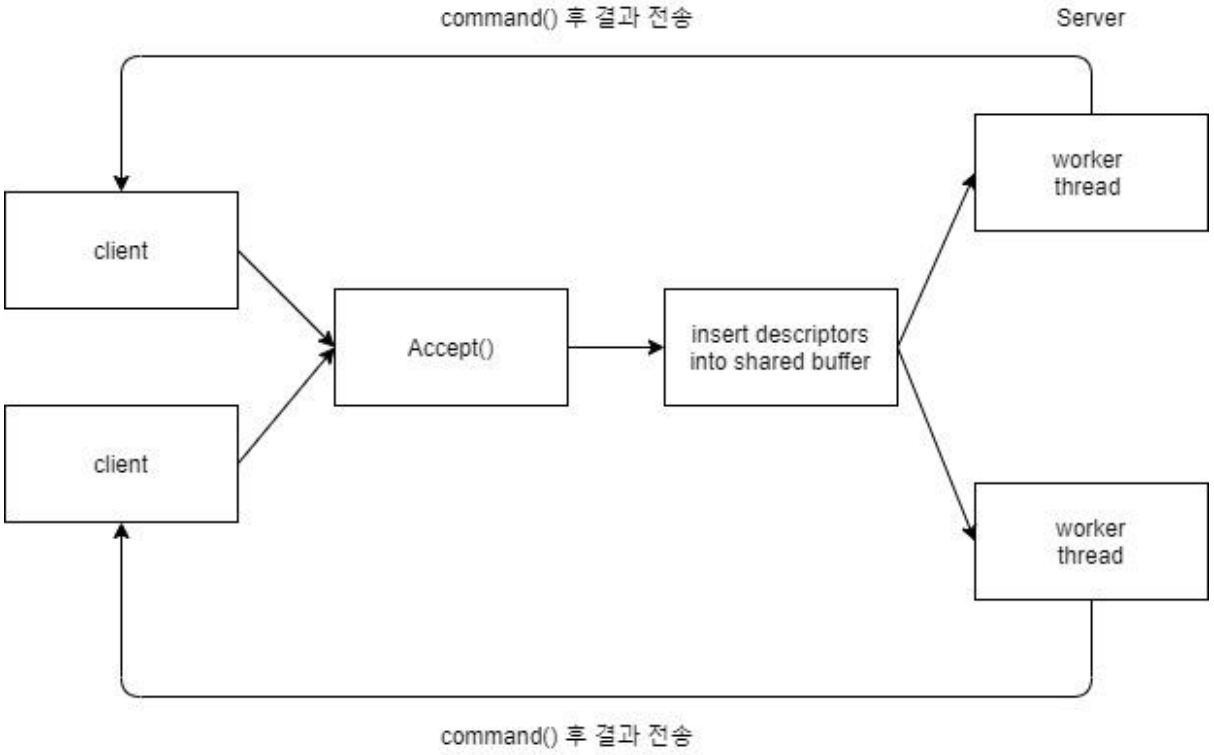
3. 구현 결과

A. Flow Chart

1. select



2. pthread



B. 제작 내용

1. **select**

   event-based server를 구현하는 중 Rio_writen() 함수로 인한 문제가 발생했다. 첫 번째 문제로는 show 명령을 받은 경우 모든 트리 노드의 정보를 전송해야 하는 데 이를 한 줄 씩 보낼 경우 client 측에서 한번에 한 줄씩만 출력하게 되는 문제점이 발생한다. 따라서 이를 해결 하기 위해 airplane 이라는 배열을 버퍼처럼 사용해 모든 정보를 한 줄로 만든 후, client측에서 이를 처리하는 식으로 해결했다. 두 번째 문제로는 Rio_writen() 에서 '₩n' 혹은 '₩0' 이 포함되지 않는 경우 송신 측에서 이를 인식하지 못하는 경우가 발생했다. 이를 해결 하기 위해 모든 전송 되는 메시지의 끝에는 '₩n' 를 포함하도록 설정했다.

2. **pthread**

   pthread의 경우에는 Readers-Writers 문제가 발생했다. 즉 semaphore 설정을 제대로 하지 못해 발생한 문제이다. 이를 해결 하기 위해 적절한 위치에 P, V 함수를 이용해 해결했다. 특히 update() 함수를 실행할 때(즉, shared data에 접근하는 경우) 다른 client가 show 명령 등을 진행하지 못하게 막았다.

C. 시험 및 평가 내용

1. 성능 예측

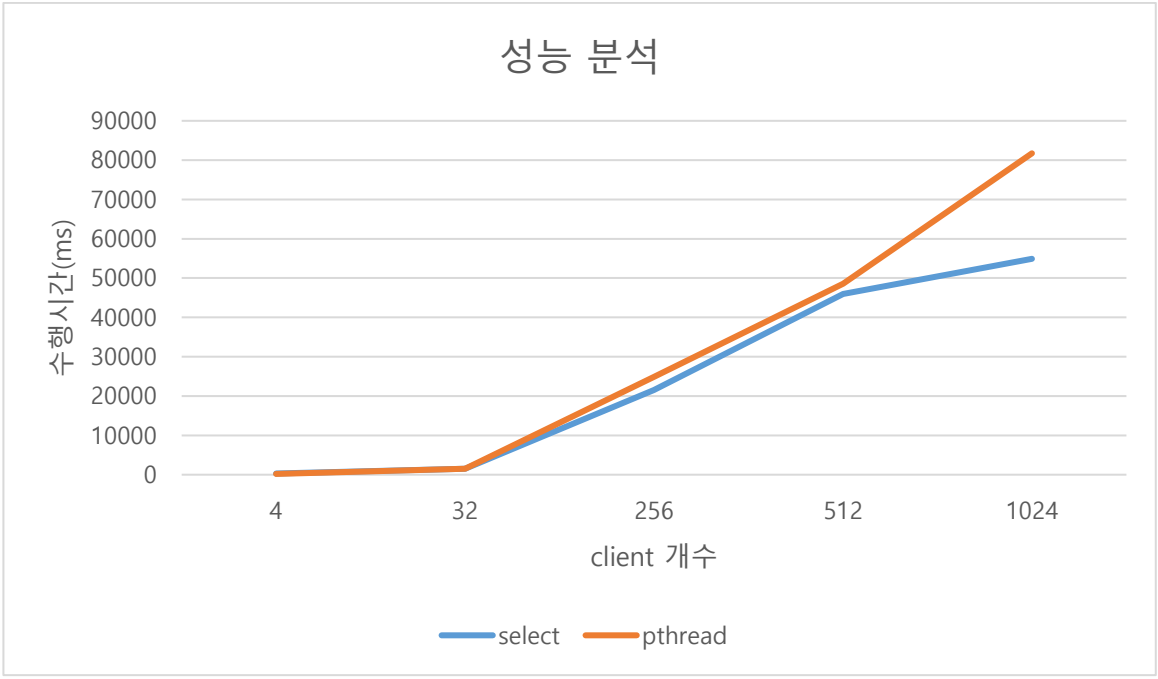
select를 이용한 event-based server는 overhead가 작아 그 성능이 pthread를 이용한 thread-based server보다 좋을 것으로 예측 할 수 있다. 실제로 성능이 중요한 분야에서 많이 사용되는 방식이다. 그러나 event-based server는 multi-core 인 환경을 제대로 활용할 수 없기 때문에 multi-core 환경에서는 thread-based server의 성능이 더 뛰어날 수 있다고 예측할 수 있다.

2. 결과 분석

1) 랜덤 명령에 따른 성능 분석

처음으로 분석해 볼 결과는 client들이 랜덤하게 sell, buy, show 명령을 전송하고 이를 server가 얼마나 빠르게 처리하는지 비교해보는 것이다. client의 개수를 늘려가며 그 결과를 분석해본다. 이를 위해 MAX_CLIENT 수를 100으로 설정하고, ORDER_PER_CLIENT 는 5로 설정한다.

우선, 실행할 때마다 수행시간의 차이가 +-1000ms 정도가 발생하는 것을 고려하여 각각 3번씩 수행하여 평균 수행시간을 사용했다. 하지만 event-based server와 thread-based server의 수행시간에 크게 유의미한 차이는 보이지 않는다. 추측 할 수 있는 원인들로는 비교하는 client 개수가 너무 적기 때문일 수도 있고, cspro 서버 상으로의 network delay, cspro의 서버 상태등이 영향을 미치기 때문일 것이다. 따라서 client의 수를 크게 늘려서 다시 한 번 분석을 진행해 보았다. 이를 위해 MAX_CLIENT 수를 1024로 설정하고, ORDER_PER_CLIENT 는 5로 설정한다.



client의 수를 크게 키우니 차이가 확실히 보인다. 예상대로 select-based server의 성능이 pthread의 성능보다 나음을 확인 할 수 있다. 다만, client의 수가 늘어날수록 각 실행마다 수행시간의 오차가 +-6000ms 까지로 크게 늘어난 것도 확인할 수 있었다.

2) 모든 client가 buy 또는 sell 만을 요청하는 경우

모든 명령을 랜덤하게 보내는 것이 아닌 buy 명령 혹은 sell 명령 만을 보냈을 때의 결과를 분석해본다. buy, sell 명령은 memory의 이진 탐색 트리를 접근하고 수정해야 하기 때문에 show 명령 보다 시간이 오래 걸릴 것이다.

실제로 select 와 pthread 모두 수행시간이 길어진 것을 확인 할 수 있다. 예상대로라면 pthread가 shared data 에 대한 semaphore 처리를 해야하므로 overhead가 커서 수행시간이 확연히 차이 날 것 같았으나 실제 결과는 크게 유의미한 차이는 보이지 않는것으로 확인된다.

3) 모든 client가 show 명령만 요청하는 경우

모든 client가 랜덤하게 명령을 보내는 경우가 아닌 오직 show 명령만 요청하는 경우를 분석해본다. 이 경우 buy, sell 의 경우보다 처리할 연산이 적어 수행시간이 더 짧을 것이다.

예상과는 다르게 buy, sell만 수행하는 경우와 수행시간에 유의미한 차이는 보이지 않는다. 하지만 client 개수가 늘어날 수록 pthred 방식이 수행시간이 더 걸리는 것을 확인 할 수 있다.