System Programming Project 5

이름 : 박찬우

학번 : 20171645

1. 개발 목표

여러 Client들의 동시 접속 및 서비스를 위해 StockServer를 Concurrent하게 구현하고, 이에 접속하는 Client 및 다중 Client 접속을 돕는 Multiclient 프로그램을 구축한다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. select

   select 함수를 통해 단일 프로세스에서 여러 Input/Output 통신 소켓을 관리해 다중 Client의 입력을 받고 이를 concurrent 하게 처리 할 수 있는 Server Program이 구축된다. select 함수로 등록한 fd_set을 통해 Client가 Server에 메세지를 보내면, select 함수는 어떤 소켓을 통해 Client가 요청을 보냈는지 확인한 후 이를 통해 input을 처리한다. EventServer 관련 함수로 소켓 생성 및 제거를 관리하고, AcceptFunc 및 ReadFunc를 통해 Client로부터 입력받는 과정을 관리한다. 여러 Client로부터 show, buy, sell 명령어를 받으면 이를 Server 프로세스의 Binary Search Tree를 통해 관리하고, 메세지를 Client에 전송하는 식으로 Server-Client간의 통신이 진행된다. 연결된 Client 개수가 0이 되면 해당 BST 내용을 stock.txt에 write하는 과정을 수행한다.

2. pthread

   pthread 함수를 통해 일정 데이터를 공유하는 thread를 생성하고, Server가 listenfd를 통해 Client의 접속을 Accept하게 되면, 해당 thread의 connfd와 연결한 뒤 이를 바탕으로 통신을 진행하는 방법으로 다중 Client의 입력을 받고 이를 처리 할 수 있는 Server Program을 구축한다. Server-Client간의 통신은 앞서 언급한 Select와 동일하게 작동한다.

B. 개발 내용

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

EventServer_Init() 으로 Server의 통신 socket인 select_socket 및 기타 변수를 초기화한 뒤, AcceptFunc 를 통해 socketfd로 client 접속 요청이 오면 EventServer_AddIO() 함수를 통해 Socket을 추가하고 해당 소켓과 ReadFunc() 를 bind한다. 그 뒤 EventServer_LoopStart() 함수에서 종료조건이 되기 전까지 select() 함수를 통해 여러 소켓을 동시에(concurrent하게) 관리한다. Client 접속이 종료되면 EventServer_Remove() 함수를 통해 해당 소켓을 제거한다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

먼저 프로그램 시작시 BST_init() 함수를 통해 stock.txt 파일로부터 내용을 읽어들여 stock data를 갖는 binary tree를 완성한다.

Socket과 연결된 ReadFunc() 에서 read() 함수를 통해 client로부터 메세지를 입력받은 뒤, 해당 메세지에 따른 명령을 수행하는데, 입력받은 ID 를 바탕으로 BST_search() 함수를 통해 해당 BST Node를 찾아 buy면 left_stock 값을 산 수치만큼 낮추고, sell이면 판 수치만큼 높인다. 만약 show라면 BST_print() 함수를 통해 BST 내부의 data 값을 write() 함수를 통해 Client로 보내고, Client에서 readnb() 함수를 통해 Server로부터 입력받은 데이터를 출력하게 된다. buy, sell 기능 역시 적절한 문자열을 write() 함수로 보낸다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

우선 Pthread_create() 함수를 통해 thread를 생성하고, thread 내부에서는 connfd = sbuf_remove() 함수를 통해 sbuf에 내용이 들어오기를 기다리게 된다. main에서 listenfd를 통해 Client의 요청을 Accept() 하게 되면, 해당 connfd 값을 sbuf_insert()를 통해 sbuf에 집어넣고, 그 순간 thread에서 sbuf_remove를 통해 connfd를 입력받아 이를 바탕으로 echo_cnt 함수를 실행해 client의 요청을 처리하게 된다. sbuf_remove 와 sbuf_insert는 각 sbuf의 내용인 slot,buf 에 대해 mutex 처리를 해 주어 여러 thread가 동시에 sbuf에 접근하는걸 막는다. echo_cnt() 함수에서는 Rio_readnb() 를 통해 client의 메세지를 받은 뒤 이를 바탕으로 적절한 데이터 처리(BST_search, BST_print 등)을 해주고, client로부터 exit를 입력받게 되면 해당 connfd를 close() 해준다. 이때 여러 thread가 동시에 stock data를 저장하는 BST에 접근하는걸 막기 위헤 mutex 처리를 해준다.

C. 개발 방법

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

pthread, select 모두 공통적으로 stock의 데이터를 저장할 Binary Search Tree와 관련 함수를 필요로 한다.

BST의 하나의 Node는 stock의 ID, left_stock, price와 leftChild, rightChild의 포인터를 갖게 되고, BST를 관리할 BST_init(), BST_search(), BST_insert(), BST_print(), BST_save() 함수를 통해 각각 BST를 초기화, BST 내부의 값을 탐색, BST에 새로운 Node를 추가, BST의 내용을 전부 print, 변경된 BST를 파일에 저장하는 기능을 수행하게 된다.

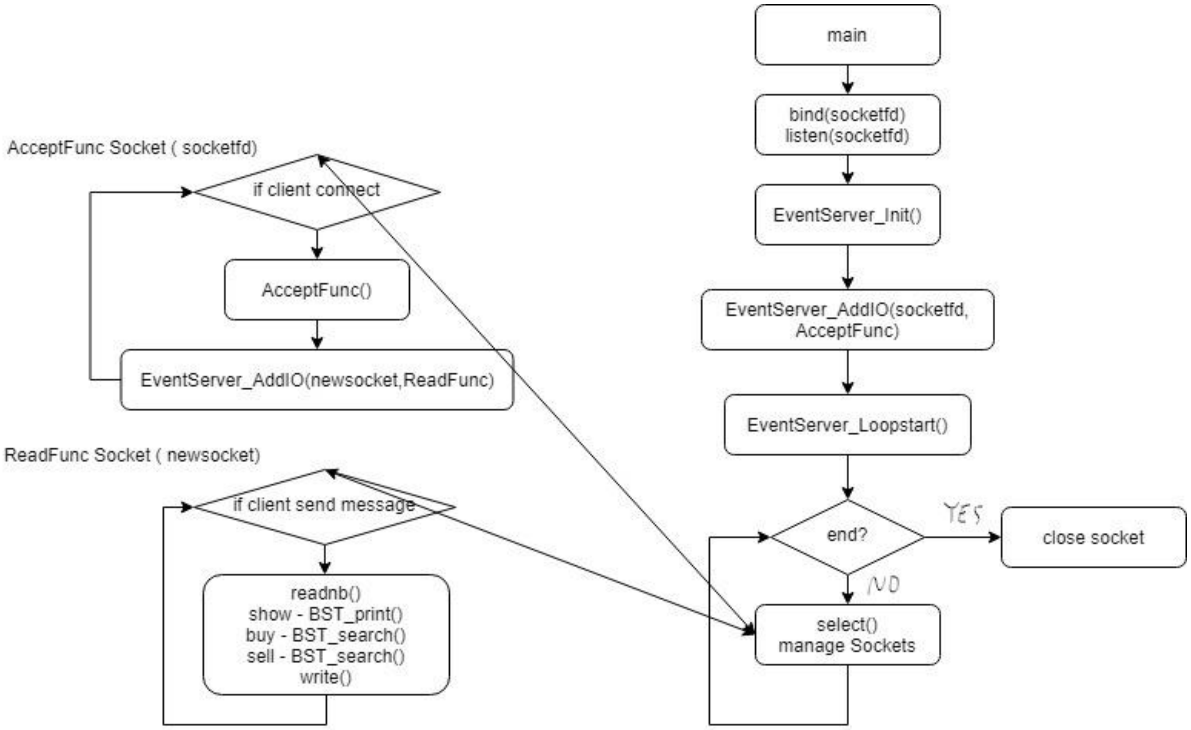
select 기반 server는 앞서 언급한 EventServer 관련 함수 (EventServer_Init(), EventServer_Loopstart(), EventServer_AddIO, EventServer_REmoveIO) 와, select의 대상이 되는 socket과 그 소켓의 자료를 처리하는 함수, write의 대상이 되는 소켓을 동시에 갖는 구조체 SelectSocket을 통해 select 함수에서 사용할 데이터를 관리한다.

pthread는 여러 Client로부터 접속되는것을 염두에 둔 sbuf 구조체를 사용해 sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove 함수를 통해 접속되는 명령을 처리한다.

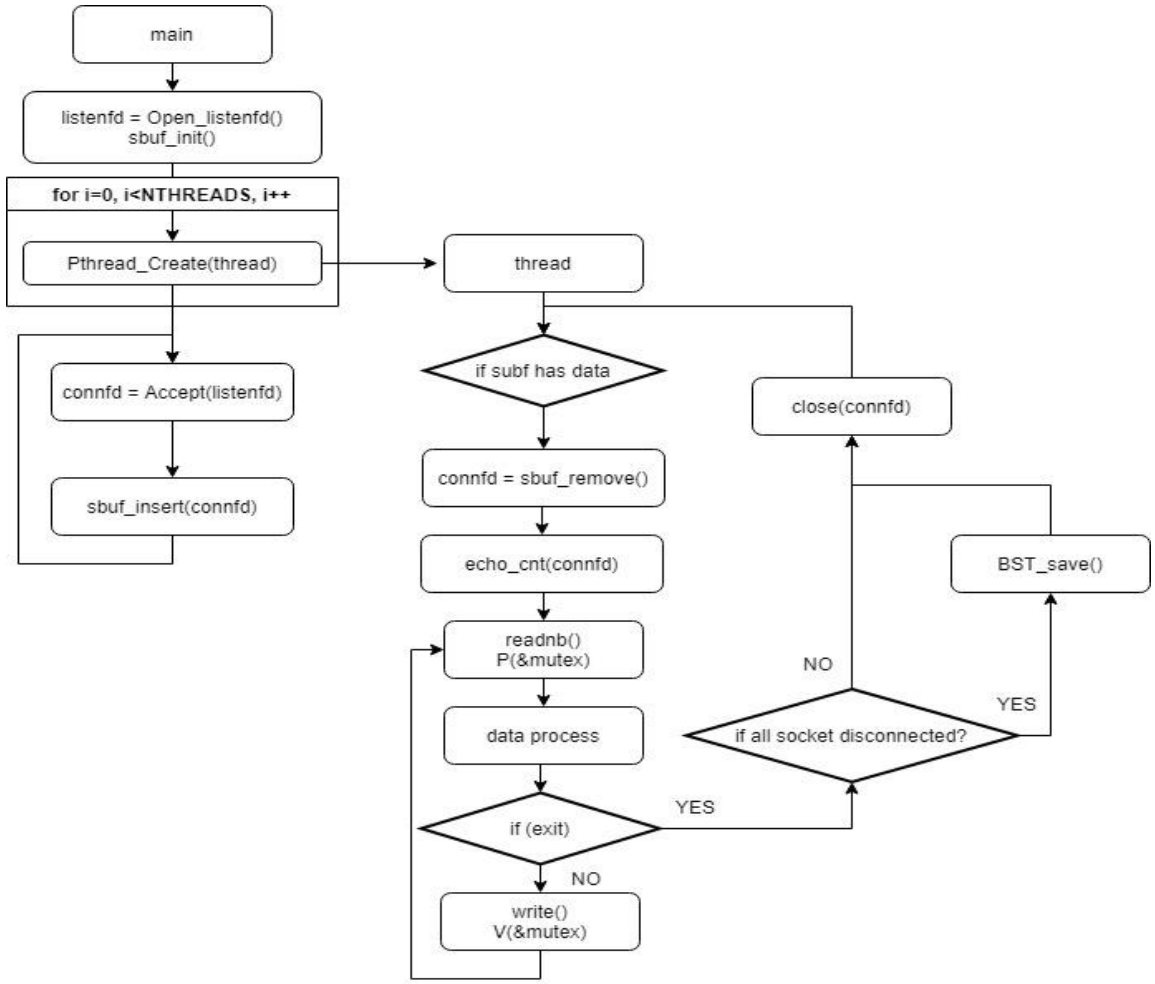
3. 구현 결과

A. Flow Chart

1. select


main
bind(socketfd)
listen(socketfd)
EventServer_Init()
EventServer_AddIO(socketfd, AcceptFunc)
EventServer_Loopstart()
end?
YES
close socket
NO
select()
manage Sockets
AcceptFunc Socket (socketfd)
if client connect
AcceptFunc()
EventServer_AddIO(newsocket,ReadFunc)
ReadFunc Socket (newsocket)
if client send message
readnb()
show - BST_print()
buy - BST_search()
sell - BST_search()
write()

2. pthread


main
listenfd = Open_listenfd()
sbuf_init()
for i=0, i<NTHREADS, i++
Pthread_Create(thread)
thread
connfd = Accept(listenfd)
sbuf_insert(connfd)
if subf has data
connfd = sbuf_remove()
echo_cnt(connfd)
readnb()
P(&mutex)
data process
if (exit)
YES
NO
write()
V(&mutex)
close(connfd)
if all socket disconnected?
NO
YES
BST_save()

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**

   select 함수를 사용하기 위해 선언한 전역변수들 중 static fd_set writeset과 static fd_set readset은 각각 현재 read, write에 쓰이는 I/O소켓을 지정하는 변수고, static int maxsocket은 여러 socket에 연결될 때 가장 나중에 연결된 (가장 큰)소켓 값을 저장한다. int flag_socketend 및 static int flag_loopend는 각각 socket이 끝날때, 프로그램이 끝날때를 확인하는 flag변수이고 static SelectSocket select_socket[MAX_CLIENT + 1]은 select 대상 소켓과 그것과 bind된 callback 함수, write의 대상 소켓을 저장하는 구조체 배열이다.

   EventServer_init() 함수에서 flag변수 및 maxsocket을 초기화 하고, writeset과 readset 역시 FD_ZERO()를 통해 초기화하며 select_socket[].fd와 tfd값은 -1로 초기화한다.

   EventServer_AddIO는 select_socket[]의 fd가 -1인 위치를 찾아 td, fn, tfd를 입력받은 값으로 설정해준뒤, maxsocket 역시 재설정한다.

   EventServer_RemoveIO는 Client의 연결이 종료되면 호출되며, 만약 모든 client가 접속이 종료됐다면 BST_save() 를 통해 stock.txt에 데이터를 저장한다. 그 뒤, select_socket을 탐색해 입력받은 위치의 fd, tfd를 다시 -1로 바꾼뒤 maxsocket 역시 재조정한다.

   AccpetFunc는 새로운 소켓 socketfd_new를 통해 accept() 한뒤 이를 EventServer_AddIO를 통해 ReadFunc과 연결한다.

   ReadFunc는 Client로부터의 입력을 read를 통해 입력받고 그에 맞는 연산을 수행한 뒤 적절한 문자열을 설정해 write를 통해 Client로 다시 전송한다.

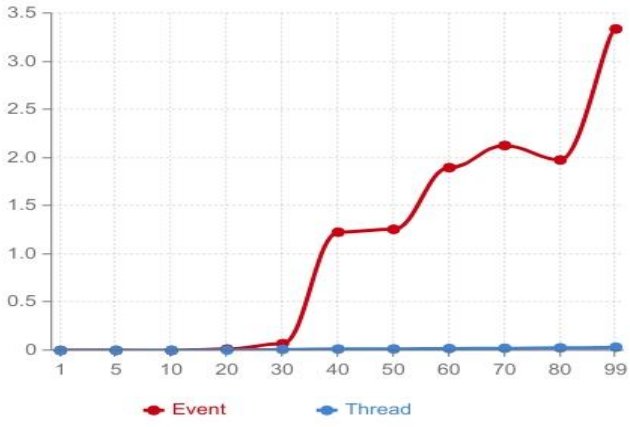
2. **pthread**

   sbuf_init, sbuf_deinit을 통해 thread끼리 공유되는 buffer 구조체인 sbuf를 할당, 초기화 및 해제한다. sbuf_insert는 Server가 Client로부터 연결을 요청받

으면, sbuf_insert를 통해 적절한 mutex 설정(item, mutex, slot)을 통해 여러 thread가 sbuf에 접근하는걸 막으며 sbuf에 연결요청을 저장하고, 각 thread는 sbuf_remove를 통해 sbuf 안의 연결 요청을 입력받아 connfd를 설정하고 이를 통해 echo_cnt를 호출해 입력받는 명령에 대한 처리를 수행한다. 만약 exit를 입력받아 echo_cnt가 종료되면 connfd를 close한 뒤, 다음 connfd가 sbuf_remove를 통해 연결되기를 대기한다.

만약 exit을 입력받았는데 해당 연결을 끊으면 모든 client의 연결이 끊어지게 되면, BST_save 함수를 통해 stock.txt에 데이터를 저장하게 된다.

C. 시험 및 평가 내용

- 우선, pthread가 select보다 효율적일 것이라고 예측했다. 왜냐하면, 비록 pthread를 통해 thread를 생성하는 과정 자체가 costly하지만, select를 통한 concurrent server는 단일 프로세스에서 모든 입력을 처리하게 되며 multi core 효과 역시 누리지 못하게 되는데, 기술의 발전으로 cpu core 기술은 발전하고, thread 생성 과정은 최적화가 잘 되어 있을거라 생각되므로 pthread가 select보다 빠르게 작동할 것으로 예측했다.
- 먼저 thread 기반과 event 기반 Server의 client 개수에 따른 차이를 확인하기 위해 실험했다. stock 종류는 5개, client 당 명령은 10개, 최대 client는 100개로 설정했다.
- 

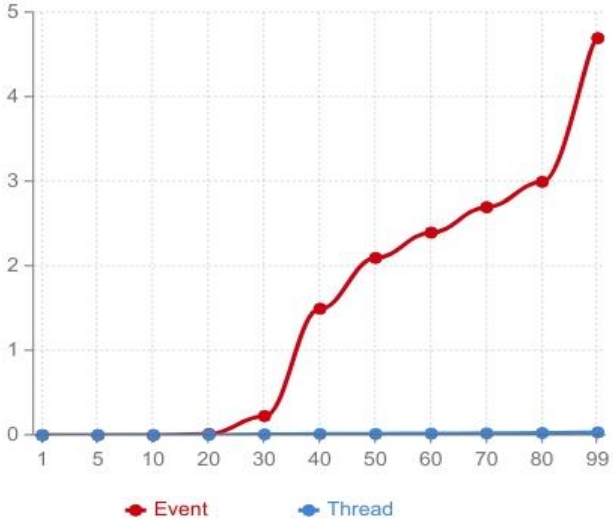
- 각 Client 개수 당 10회 확인한 뒤, 최소값 2개와 최대값 2개를 뺀 뒤 평균을 낸 수치를 그래프로 나타낸 결과이다.
- Client 개수가 적을때는 유의미한 차이를 내지 못했지만, Client 개수가 늘어날수

록 event 기반 서버가 압도적으로 오래 걸리는 걸 확인할 수 있다. 이러한 결과가 나온 이유는 역시 event 기반은 단일 프로세스고, multi core를 활용할 수 없어 Client의 숫자가 늘어날수록 비효율적인 면이 커지는 것으로 보인다.

- 특기할 점으로, thread 기반 Server는 client 개수에 따라 선형적으로 커지는 모습을 확인할 수 있고, 실제 실험값도 거의 일정하게 나왔는데, event 기반 서버는 값이 상당히 출렁이는걸 확인할 수 있다. 측정할 때도 평균은 0.02지만 최대값이 1.02 인 등 특정 상황에 시간이 급격하게 오래 걸리는 현상이 확인됐는데, 이는 select 함수와 그 관련 함수들이 Client 최대 개수인 크기 100 배열을 탐색하는 과정에서 특정 case 에 따라 시간이 급격하게 소요되는 것으로 생각됐다.

- 다음으로, client의 요청 타입에 따른 동시처리율 변화를 분석했다. 먼저, 실험 전에 예측하기로는 show 명령어는 BST의 모든 Node를 탐색해야 되는 반면, buy 및 sell은 특정 Node에 도달하기만 하면 되므로 show 명령이 다른 명령보다 더 오랜 시간이 소요될 것으로 예상했다. 하지만, 기존의 testcase(최대 client 100, stock 종류 5, client당 명령 10회)에서는 유의미한 변화를 관찰할 수 없었다. 위의 실험 결과는 모든 종류의 요청(show,buy,sell)이 랜덤하게 들어오는 경우인데, 이를 show만 들어오게 변경하거나 sell,buy만 들어오게 변경해도 값의 차이는 매우 미비했다.

- 이러한 결과가 나온 이유는 Stock의 종류에 있다고 판단되었다. Stock의 종류가 5종류 밖에 없으니, Binary Search Tree의 깊이가 너무 얕게 형성되어 유의미한 차이가 나오지 않았다고 보고, stock의 종류를 15개로 늘리고 stock.txt를 수정한 뒤 show 명령만 수행하도록 설정한 뒤 다시 측정해 보았다. 아래는 그 결과다.



- 이전의 실험 값보다 전체적인 값이 크게 늘어난 모습을 확인할 수 있고, 그래프 상으로 확인하긴 어렵지만 thread 기반의 Server의 측정 시간도 전 실험 대비 1.2~1.5배 정도로 나타났다. 이로서 show 명령이 다른 명령에 비해 Binary Search Tree를 탐색하는 시간이 오래 걸려 더 오랜 시간이 들 것이라는 예상이 맞았음을 확인할 수 있다.