

System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 고예성

학번 : 20161385

1. 개발 목표

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

다중 클라이언트들이 서버에 동시에 접속했을 때 하나의 클라이언트에 대해서만 요청을 처리하는 것이 아닌 여러 클라이언트의 요청을 처리할 수 있는 concurrent stock server를 구축한다. 주식 서버는 클라이언트의 요청(show, buy, sell, exit)에 따라 주식 목록을 보여주고 사고 팔고 퇴장 명령을 수행한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

Event-driven방식은 여러 클라이언트 중에서 상태에 변화가 있는 클라이언트에 대해서 서버가 요청을 처리하는 방식이다. 이 방식을 구현하기 위해서는 Select함수를 통해 구현 할 수 있다. Select에 의해 여러 file descriptor들의 변화를 감시할 수 있다. Pending input이 있는 file descriptor들의 요청을 처리하는 것을 통해 여러 클라이언트들의 요청을 concurrently하게 다룰 수 있다.

2. Task 2: Thread-based Approach

Thread-base 방식은 서버에서 multi-thread 방식을 통해 여러 클라이언트들로부터 온 요청을 처리하는 방식이다. 이때 사용하는 것이 Pthread_create()함수이다. Pthread_create()에 의해 thread pool을 생성해 놓고 클라이언트로부터 연결이 들어오면 연결을 해서 thread pool의 thread들 중에서 클라이언트가 주식목록, 구매, 판매, 퇴장 요청을 하게 되면 그러한 요청을 처리한다. 여기서 thread는 하나가 아니라 여러 개이기 때문에 여러 클라이언트들에 대해 concurrent하게 요청을 수행할 수 있다. 또 공유데이터인 주식데이터를 쓰는 것에 대해 critical section에 mutex와 semaphore를 구현해서 주식의 데이터가 잘 못 되는 경우가 발생하지 않게 만들어 준다.

3. Task 3: Performance Evaluation

다중 클라이언트의 접속을 통해 event-driven방식의 서버와 thread-base방식의 서버 성능을 분석할 수 있게 한다. 클라이언트의 개수를 정해줄 수 있으며 명령은 무작위이지만 클라이언트 하나 당 몇 번의 명령을 내릴지 정해 줄 수 있다. 또한 주식의 개수가 몇 개인지와 사고 팔 때 최대 개수를 지정해 줄 수 있다. 이러한 다중 클라이언트의 접속과 명령에 의해 처리 시간이 얼마나 차이나는지를 event-driven방식과 thread-base방식을 서로 비교해서 측정해야 한다.

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

I/O multiplexing을 위해서 Select함수를 이용해야 한다. Select함수는 접속된 클라이언트들의 file descriptor를 감시할 file descriptor의 집합을 표현하는 변수에 넣어주었다. 그리고 반복문 안에서 select함수를 통해 이 변수의 변화가 있는지 감시하도록 하였다. 만약 이 집합 변수 안에서 특정 file descriptor의 변화가 생기면 반복문에서 file descriptor를 돌다가 변화를 감지하게 되고 이에 따라 요청을 수행하게 된다. 또 만일 클라이언트가 exit명령을 요청하게 되면 서버는 이에 따라 해당 클라이언트와의 연결을 종료하고 감시목록 집합에서 file descriptor를 삭제한다. 이렇게 되면 Select함수에 의해 삭제된 file descriptor는 감시대상에서 벗어나게 된다. I/O multiplexing방식은 Synchronization이 필요가 없다. 왜냐하면 여러 요청이 동시에 처리 되는 것처럼 보이지만 사실 하나의 클라이언트로부터 온 요청을 처리하기 전까지는 다른 클라이언트의 요청을 처리할 수 없기 때문이다. 즉, fine grained concurrency는 아닌 것이다.

✓ epoll과의 차이점 서술

select함수는 동작환경에 따라 다르지만 일반적으로 검사할 수 있는 file descriptor의 개수가 최대 1024개로 제한된다. 그리고 검사영역에 포함되는 모든 file descriptor를 순회하면서 FD_ISSET으로 체크를 해야 하기 때문에 불필요한 연산이 들어가게 된다. 실제로 상태 변화가 생긴 file descriptor의 목록이 있다면 더 빠르게 작동할 수 있을 것이다. 그리고 select함수를 통해 관찰 대상에 대한 정보를 계속해서 전달하는 것도 오버헤드를 일으킨다. 관찰 대상이 새로 추가되

거나 삭제되는 경우에만 관찰대상을 전달한다면 오버헤드를 덜게 될 것이다. epoll은 이러한 select의 단점을 보완한 I/O기법이다. 전체 file descriptor에 대한 반복문을 사용하지 않고 커널에게 정보를 요청하는 select와 같은 함수를 호출할 때마다 전체 관찰 대상의 정보를 모두 넘기지 않아도 된다. 계속해서 정보를 넘기지 않기 위해 관찰 대상이 되는 file descriptor의 목록을 운영체제가 담당한다. 운영체제에게 목록을 만들어 달라고 요청하면 그 목록에 해당하는 file descriptor를 반환한다. 관찰영역이 변경되면 변경요청을 할 수 있고 관찰 대상의 변경사항을 체크할 수도 있다.

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

- ✓ **Master Thread의 Connection 관리**

Master thread에서는 먼저 여러 클라이언트의 연결을 가능하게 하기 위해 클라이언트가 접속을 하면 접속한 클라이언트에 대한 file descriptor를 버퍼에 저장해 놓는 방식을 사용하고 있다. 먼저 initFdbuf에 의해 fdbuf을 초기화해서 buffer의 자료구조를 만든다. Accept에 의해 connfd에 연결된 클라이언트의 file descriptor가 반환되면 그것을 file descriptor를 관리하는 버퍼에 삽입하기 위해 insertFdbuf함수를 통해 삽입을 한다. 버퍼에 file descriptor를 삽입하고 삭제하는 과정은 producer와 consumer의 synchronization개념을 가지고 구현하였다. Main thread는 클라이언트의 연결요청이 있으면 file descriptor를 버퍼에 삽입하는 producer역할이고 thread는 suspend상태에 있다가 클라이언트가 연결되면 깨어나서 요청을 수행하는 consumer의 역할을하게 된다. Thread는 parsing함수를 처리하게 되는데 여기서 master thread는 하나의 thread가 일을 끝냈을 때 자원을 회수하는 pthread_join을 실행할 필요가 없다. 왜냐하면 pthread_detach를 통해 하나의 thread가 개별적으로 일을 끝냈을 때 운영체제에 의해 알아서 자원이 회수되도록 만들었기 때문이다. 따라서 별도로 thread자원 회수에 관한 관리를 할 필요는 없고 connection이 끝나면 file descriptor에 대해 close를 하면 된다.

- ✓ **Worker Thread Pool 관리하는 부분에 대해 서술**

Worker thread pool을 만들기 위해 for문을 돌면서 FBBUFSIZE인 1024개만큼 Pthread_create함수를 통해 thread를 만들어 주었다. 이렇게 만들어진 Thread는 서버에 접속한 connected fd가 있을 때 활성화되어 parsing함수를 처리하게 된다. Parsing에서 Pthread_detach함수는 thread가 parsing함수를 모두 처리했을 때 운영체제가 thread의 자원을 회수하도록 만든 것이다. Connfd는 removeFdbuf함수

를 통해 file descriptor의 값을 받는데 이것은 서버에 연결된 클라이언트 하나의 file descriptor 값을 반환한다. Thread가 처리할 file descriptor가 생겼기 때문에 하나의 thread는 클라이언트의 연결이 끊기기 전까지 계속해서 클라이언트의 요청을 받아서 일을 한다.

- **Task3 (Performance Evaluation)**

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

첫 번째로 같은 클라이언트의 수, 같은 명령 개수에 관해 주식 종목의 개수를 늘렸을 때의 경우, 두 번째로 같은 클라이언트 수, 같은 주식 종목 개수에 대해 명령의 개수를 늘렸을 때, 세 번째로 같은 명령의 개수, 같은 주식 종목의 개수에 대해 클라이언트의 수를 늘렸을 때의 속도를 비교해봐야 한다. 왜냐하면 명령의 개수, 다중 클라이언트의 접속 수, 주식 종목의 개수에 따라 연산하는 양이 달라지기 때문이다. 이에 대한 시간 측정은 <time.h>를 multichoice에 include해서 프로세스가 실행되고 끝날 때까지의 시간을 측정하였다.

- ✓ Configuration 변화에 따른 예상 결과 서술

일반적으로 클라이언트 수가 많아 질수록 pthread로 구현된 thread-base 서버의 효율이 Select로 구현된 서버에 비해 좋을 것이다. 왜냐하면 Select로 구현된 서버의 경우 진정한 의미에서의 fine grain한 방식이 아니기 때문이다. 결국 parallel하게 일이 처리되는 것이 아니라 순차적으로 처리되는 것이다. 따라서 요즘같이 cpu가 multi-core가 일반적인 경우에는 select로 구현할 경우 장점을 살릴 수 없다. 하지만 마냥 event-base서버가 thread서버보다 이점을 갖을 수 있다고 생각되는 부분은 buy와 sell, 그리고 exit명령에 있어서 더 빠른 속도를 보여줄 것이라고 생각된다. 왜냐하면 thread-base서버의 경우 위와 같은 명령을 처리하기 위해서는 synchronization을 위해 mutex와 semaphore를 사용해야 한다. 이렇게 되면 어쩔 수 없이 event-driven방식보다 오버헤드가 더 발생하게 된다. 하지만 select로 구현한 서버는 그럴 필요가 없다. 어차피 순차적으로 요청을 처리하기 때문이다. 반면 thread-base의 경우 show명령에 있어서 select로 구현된 서버보다 훨씬 좋은 성능을 보일 것이다. Thread-base서버는 show명령에 있어서 여러 thread가 접근해서 읽을 수 있도록 구현이 되어있다. 따라서 여러 클라이언트를 병렬처리 할 수 있는 thread-base서버가 장점을 갖을 것이다. 하지만 결국 접속하는 클라이언트의 수가 많으면 많을 수록 다른 연산에 있어서도 multi-thread의 이점이 더 커질 것이다. 따라서 event-driven방식은 적은 트래픽의 간단한 연산을

수행하는 서버에 적당한 방식이라 생각이 들고 thread-base방식은 대량 트래픽을 처리하는데 있어서 적당한 방식이라 생각이 듈다.

개발 방법

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

공통적으로 stock.txt의 주식목록을 메모리에 올리는 방법으로 자료구조인 binary tree를 이용하고 있다. Binary tree를 만들기 위해서 stock의 정보를 저장하는 node를 구조체로 만들었다. StockItem은 주식의 ID, left_stock, price를 int형으로 저장하고 있고 binary tree의 구조를 만들기 위해 pItem은 StockItem의 포인터형이고 이것을 leftChild와 rightChild를 가리키는 포인터를 만들어 주었다. Binary tree중에서도 binary search tree 구조로 만들어 찾고자 하는 ID를 빠르게 찾을 수 있도록 만들어 주었다.

- Task1 (Event-driven Approach with select())

I/O multiplexing을 위해서는 먼저 여러 connection에 대한 정보를 저장해야 된다고 생각했다. 따라서 여러 클라이언트들이 접속을 요청하게 되면 여러 File Descriptor가 반환되기 때문에 이러한 File Descriptor를 저장할 자료구조가 필요했다. 물론 자료구조를 select가 감시할 수 있는 file descriptor의 개수에 제한이 있기도 해서 array를 통해서 간단하게 구현해도 되지만 LinkedList방식을 이용해서 클라이언트 연결요청에 최대한 제한이 없게 만들었다. 또한 순차적으로 file descriptor가 처리되어야 하고 삽입, 삭제 또한 순차적으로 수행되기 때문에 Array를 쓸 때와 비교해서 시간복잡도에 별 차이가 없다. FD_Node 구조체는 이러한 LinkedList를 구현하기 위한 node 구조체이다. fd값을 저장하는 변수와 다음 node를 가리키는 포인터인 pFD_Node 변수로 구성되어 있다. 이러한 fd가 linkedlist로 연결되어있기 때문에 for문을 통해서 fd를 감시할 수 있다. 만약에 새로운 클라이언트가 연결된다면 new_fd함수를 통해 새로운 노드를 추가한다. 또 클라이언트가 exit명령에 의해 종료를 요청하면 서버는 그 클라이언트에 해당하는 file descriptor를 close하기 전에 linkedlist에서 fd를 제거해야 한다. 이 명령을 수행하는 함수는 del_fd함수에 의해서 제거할 수 있다. 또한 fd의 변화를 관찰하기 위한 목록원본인 watch_set에서도 fd를 없애야 한다. 따라서 FD_CLR함수를 통해 watch_set에서 fd의 인덱스에 해당하는 비트를 clear한다.

- Task2 (Thread-based Approach with pthread)

Multi-thread를 이용한 다중 클라이언트의 요청처리를 하기 위해서는 thread pooling을 하는 것도 중요하지만 공통변수를 통제하는 것도 중요하다. 공통 변수에 대해 여러 스레드가 접근해서 값을 읽고 쓰게 된다면 데이터가 프로그래머가 원하는 방향으로 나오지 않을 수도 있다. 따라서 이러한 공통 변수를 제어하는 것이 중요한데 일단 읽고 쓰는 작업을 실행하는 부분이 어디인지를 알아야 한다. 주식을 조회하는 것에서 읽기작업이 실행되고 사고 파는 과정 그리고 서버의 종료나 클라이언트의 종료에서 쓰기작업이 실행된다. 따라서 이 부분에 대해 mutex나 semaphore를 걸어줘야만 정확한 읽기, 쓰기 작업이 완료된다. 구조체의 자료구조를 보면 binary tree를 만들기 위한 것에서 writer와 mutex라는 변수를 sem_t형으로 만들어 주었다. 또한 여러 개의 file descriptor를 목록으로 만드는 과정 즉, 버퍼에 추가하는 과정에서 여러 스레드에 의해 받은 file descriptor가 데이터가 덮어씌워지지 않도록 semaphore와 mutex를 이용해서 만들어야 한다. 이를 위해 fdbuf_t라는 구조체에 sem_t형으로 mutex, push, pop이라는 변수를 만들어 주었다. Push는 스레드에 의해 버퍼에 fd가 추가될 때 값을 감소시키는 변수 즉, 버퍼에 fd를 얼마나 추가할 수 있는지 나타내주는 것이고 pop은 버퍼에 들어있는 fd의 양을 나타내기 때문에 fd가 버퍼에서 제거 될 때마다 값이 감소하게 된다. 버퍼의 생산자는 master thread가 되고 소비자는 worker thread가 된다. 또 binary tree의 값을 txt파일에 업데이트하는 과정에서도 mutex가 필요하므로 fileMutex라는 global mutex를 만들어 주었다. 이러한 방법을 통해 Synchronization을 해주었다.

3. 구현 결과

- 2번의 구현 결과를 간략하게 작성
- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가
- Task1 (Event-driven Approach with select())

```

C stockserver.c ×

C stockserver.c > ⌂ show(int)
49
50     int main(int argc, char **argv)
51     {
52         /* file descriptor를 event driven 방식에 따라
53             바꿔서 처리할 수 있도록 구현 */
54         int listenfd, connfd;
55         socklen_t clientlen;
56         struct sockaddr_storage clientaddr;
57         /* Enough space for any address */
58         // line:netp:echoserveri:sockaddrstorage
59         char client_hostname[MAXLINE], client_port[MAXLINE];
60         // file descriptor관련 변수들
61         pFD_Node pIter;
62         int fd_range;
63
64     if (argc != 2)
65     {
66         fprintf(stderr, "usage: %s <port>\n", argv[0]);
67         exit(0);
68     }
69
70     // 서버가 종료 할 때 수행해야할 명령은 따로 interrupt handler를 통해 설정
71     // sigint handler등록
72     Signal(SIGINT, sig_int_handler);
73
74     // 메모리에 주식목록 로딩
75     load_stocks();
76     FD_ZERO(&watch_set);
77
78     // 만일 client가 접속하면 FileDescriptor List에 추가
79     listenfd = Open_listenfd(argv[1]);
80     fd_range = listenfd;
81     FD_SET(listenfd, &watch_set);
82     new_fd(listenfd);

```

처음 main함수에서는 서버가 클라이언트와 통신하기 위해 필요한 데이터를 로딩하는 과정이 주를 이룬다. Signal함수를 통해 SIGINT 핸들러를 등록하는 과정을 거쳐 binary search tree에 stock.txt에 있는 주식데이터를 프로세스에 로딩한다. 그리고 open_listenfd함수를 통해 서버가 listening을 하도록 만든다. 또한 watch_set은 fd를 감시목록에 넣는 과정을 거친다.

```

C stockserver.c ×
C stockserver.c > ⌂ main(int, char **)
82     new_fd(listenfd);
83
84     while (true)
85     {
86         pending_set = watch_set; // watch_set은 원본이므로 pending_set에 옮겨서 저장
87         // FileDescriptor중에 입력 가능한 것이 있는지 감시한다.
88         Select(fd_range + 1, &pending_set, NULL, NULL, NULL);
89
90         // List에 등록된 FileDescriptor를 돌면서 변화를 감지한다.
91         for (pIter = pFD_head; pIter != NULL;)
92         {
93             int currentFD = pIter->fd;
94             pFD_Node nextPtr = pIter->nextLink;
95
96             // fd가 설정되어 있는데 서버로의 access 요청이 아닌 명령을 내리는 경우
97             if (FD_ISSET(currentFD, &pending_set) && (currentFD != listenfd))
98             {
99                 parsing(currentFD);
100            }
101
102             // fd가 설정되어 있고 서버로의 access 요청인 경우
103             else if (FD_ISSET(currentFD, &pending_set))
104             {
105
106                 clientlen = sizeof(struct sockaddr_storage);
107                 connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
108                 Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
109                 printf("Connected to (%s, %s)\n", client_hostname, client_port);
110                 FD_SET(connfd, &watch_set);
111                 new_fd(connfd);
112
113                 if (fd_range < connfd)
114                     fd_range = connfd;
115
116             pIter = nextPtr;
117         }

```

서버 동작의 핵심이 되는 부분이다. While문을 계속해서 돌면서 pending_set에 watch_set의 값을 복사하는 과정을 거치는데 이 과정을 루프의 처음에 수행하는 이유는 또 다른 클라이언트가 접속했을 때 새로운 감시 대상을 추가해야 하기 때문이다. 그래서 만약에 다른 클라이언트가 접속하게 된다면 그 다음에 Select함수에 의해 감시대상에 추가가 된다. 그 다음 안쪽의 반복문은 linkedlist를 돌면서 fd를 체크한다. If문에서 fd가 감시대상에 있으면서 listenfd와 값이 같다면 새로운 클라이언트의 접속이기 때문에 Accept를 통해 새로운 클라이언트의 정보를 받고 fd값을 받은 후 new_fd함수를 통해 새로운 fd를 linkedlist에 추가하게 된다. 만일 listenfd와 currentFD와 값이 같지 않은데 감시대상이면서 상태가 변한 것이 확인이 됐다면 parsing이라는 함수를 통해 해당 클라이언트의 fd로부터 명령을 읽어와 명령을 수행한다. While문이 무한 루프이기 때문에 for문 또한 linkedlist를 모두 돌았더라도 계속 처음으로 돌아가 다시 검사를 하게 된다.

```

C stockserver.c ×

C stockserver.c > ⚡ parsing(int)
125     char ConvTmp[MAXLINE]; // client로 받은 명령어의 파라미터의 값을 integer로 바꾸기 위한 임시변수
126     int ID, stockNum;      // client로부터 받은 주식의 ID와 개수(num)를 저장하는 변수
127     char *argv[3];         // 명령어와 파라미터를 저장하는 문자열 배열
128     char buf[MAXLINE];
129     rio_t rio;
130
131     Rio_readinitb(&rio, connfd);
132
133     // sell id number; buy id number; show; exit
134     if ((receivedByte = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
135     {
136         printf("server received %d bytes\n", receivedByte);
137         // 명령어와 파라미터를 구분화
138         for (int i = 0; i < 3; i++)
139         {
140             if (i == 0)
141                 argv[i] = strtok(buf, " ");
142             else
143                 argv[i] = strtok(NULL, " ");
144         }
145
146         // 파라미터를 string에서 integer로 바꿈
147         if (argv[1] != NULL && argv[2] != NULL)
148         [
149             strcpy(ConvTmp, argv[1]);
150             ID = atoi(ConvTmp);
151             strcpy(ConvTmp, argv[2]);
152             stockNum = atoi(ConvTmp);
153         ]
154
155         // client의 command에 따른 동작 수행
156         if (!strcmp(argv[0], "show\n"))
157             show(connfd);
158         else if (!strcmp(argv[0], "buy"))
159             buy(connfd, ID, stockNum);
160         else if (!strcmp(argv[0], "sell"))
161             sell(connfd, ID, stockNum);
162         else if (!strcmp(argv[0], "exit\n"))
163             clientExit(connfd);

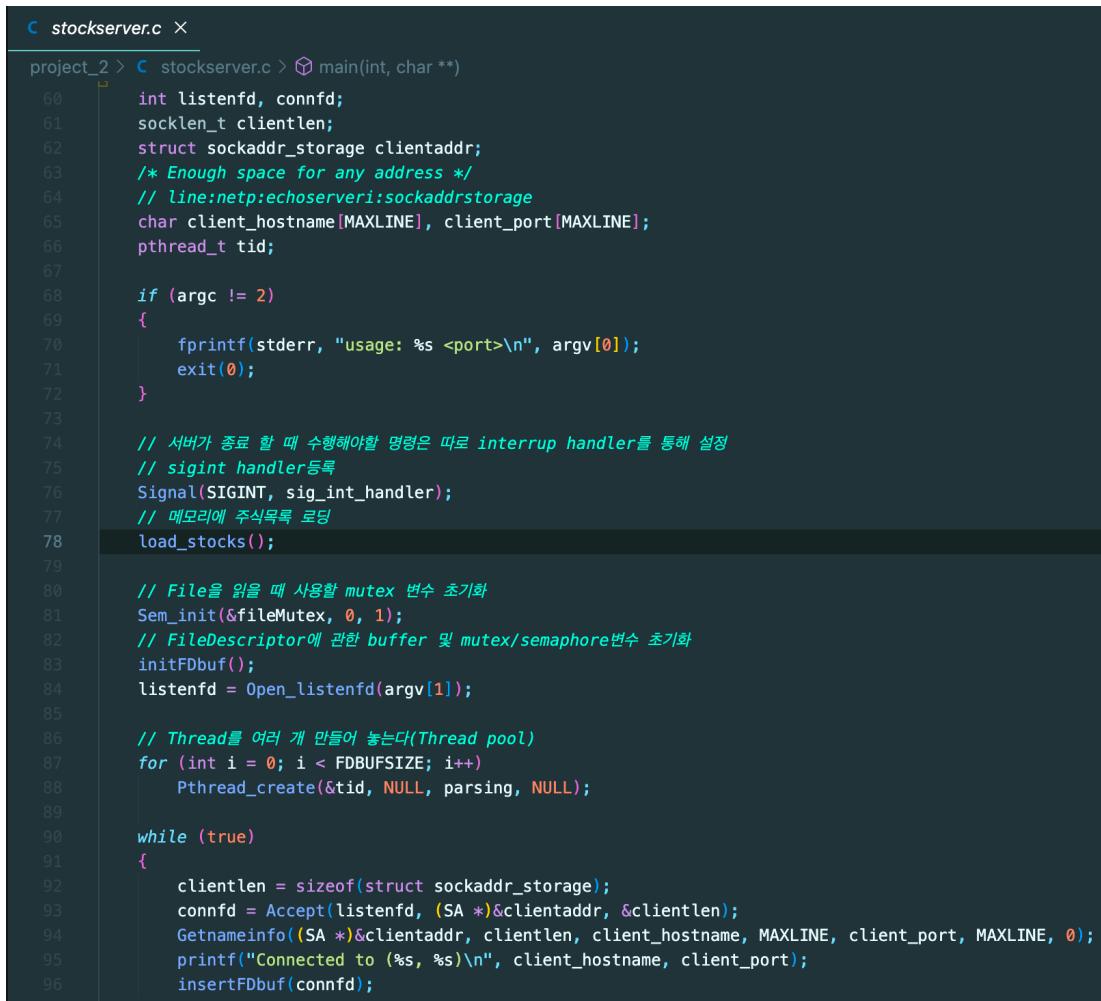
```

다음은 parsing함수이다. Rio_readinitb함수를 통해 특정 클라이언트로부터 데이터를 받을 준비를 한다. 그리고 요청(데이터)를 받게 되면 얼마큼 받았는지 서버쪽에 띄우고 그 요청을 parsing해서 요청에 맞게 명령을 수행한다. 여기서 show는 recursion을 이용해 inorder방식으로 tree를 돌면서 주식데이터를 클라이언트에게 보낸다, buy, sell은 클라이언트의 명령에 따라 tree에 있는 데이터를 변경한다. 이 경우에는 binary search tree의 규칙을 따라 tree를 search해서 데이터를 변경한다. exit은 클라이언트와의 연결을 끊는 건데 끊기 전에 클라이언트에 의해 변경된 모든 데이터를 저장하고 서버에 있는 클라이언트의 정보를 모두 삭제한다. parsing이 정확하게 됐으면 위의 요청을 정확하게 수행할 것이다. 하지만 만일 client가 아무 요청도 하지 않고 비정상종료가 된 경우(ex. sigint에 의한 종료)에는

서버가 자신이 그동안 처리했던 결과를 업데이트하기 위해 updateFile함수를 수행하고 FD_CLR, del_fd, close 함수들로 클라이언트 정보를 모두 삭제해준다.

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

Thread-base서버의 경우 많은 부분에서 위의 서버와 일치하지만 main함수에서 thread pooling하는 부분과 mutex와 semaphore로 동기화처리를 하는 부분에서 차이가 있다. 차이가 있는 부분에 대해서만 보도록 하자.



The screenshot shows a code editor with the file 'stockserver.c' open. The code is written in C and implements a server using threads. It includes comments explaining various parts of the code, such as handling signals (SIGINT) and setting up mutexes and semaphores. The code uses standard C libraries like `accept`, `getnameinfo`, and `pthread_create`. The file is part of a project named 'project_2'.

```
c stockserver.c x
project_2 > c stockserver.c > main(int, char **)
60     int listenfd, connfd;
61     socklen_t clientlen;
62     struct sockaddr_storage clientaddr;
63     /* Enough space for any address */
64     // line: netp:echoserveri:sockaddrstorage
65     char client_hostname[MAXLINE], client_port[MAXLINE];
66     pthread_t tid;
67
68     if (argc != 2)
69     {
70         fprintf(stderr, "usage: %s <port>\n", argv[0]);
71         exit(0);
72     }
73
74     // 서버가 종료 할 때 수행해야할 명령은 따로 interrupt handler를 통해 설정
75     // sigint handler등록
76     Signal(SIGINT, sig_int_handler);
77     // 메모리에 주식목록 로딩
78     load_stocks();
79
80     // File을 읽을 때 사용할 mutex 변수 초기화
81     Sem_init(&fileMutex, 0, 1);
82     // FileDescriptor에 관한 buffer 및 mutex/semaophore변수 초기화
83     initFDbu();
84     listenfd = Open_listenfd(argv[1]);
85
86     // Thread를 여러 개 만들어 놓는다(Thread pool)
87     for (int i = 0; i < FDBUFSIZE; i++)
88         Pthread_create(&tid, NULL, parsing, NULL);
89
90     while (true)
91     {
92         clientlen = sizeof(struct sockaddr_storage);
93         connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
94         Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
95         printf("Connected to (%s, %s)\n", client_hostname, client_port);
96         insertFDbu(connfd);
```

먼저 main함수에서 Pthread_create함수를 통해 여러 개의 thread를 만들어 놓는다. 이 thread는 parsing함수를 처리하게 되며 새로운 클라이언트가 접속해서 fd가 생기게 되면 하나씩 활성화된다. While문 안에 보면 새로운 클라이언트가 접속 했을 때 insertFDbu라는 함수가 실행되고 이 함수에 의해 connfd가 fd를 저장하는 buffer에 추가된다.

```
27
28     typedef struct _fdbuf_t
29     {
30         int *buf;
31         int size;
32         int front;
33         int rear;
34         sem_t mutex;
35         sem_t push;
36         sem_t pop;
37     } fdbuf_t;
38
39     fdbuf_t *fdbuf;
40
41     void initFdbuf();
42     void insertFdbuf(int connfd);
43     int removeFdbuf();
44     void freeFdbuf();
45 //---Mutex END---//
```

```
C stockserver.c ×
project_2 > C stockserver.c > ...
418 }
419
420 void initFdbuf()
421 {
422     // fdbuf을 동적할당으로 초기화
423     fdbuf = (fdbuf_t *)malloc(sizeof(fdbuf_t));
424     fdbuf->buf = (int *)calloc(FDBUFSIZE, sizeof(int));
425     fdbuf->size = FDBUFSIZE;
426     // 버퍼의 front와 rear index 초기화
427     fdbuf->front = fdbuf->rear = 0;
428     // 버퍼에 대한 뮤텍스 초기화
429     Sem_init(&(fdbuf->mutex), 0, 1);
430     // 버퍼에 대한 세마포어 초기화
431     Sem_init(&(fdbuf->push), 0, FDBUFSIZE);
432     Sem_init(&(fdbuf->pop), 0, 0);
433 }
```

```

435 void insertFdbuf(int connfd)
436 {
437     P(&(fdbuf->push));
438     P(&(fdbuf->mutex));
439     fdbuf->buf[+(fdbuf->rear) % FDBUFSIZE] = connfd;
440     V(&(fdbuf->mutex));
441     V(&(fdbuf->pop));
442 }
443
444 int removeFdbuf()
445 {
446     int removed_fd;
447
448     P(&(fdbuf->pop));
449     P(&(fdbuf->mutex));
450     removed_fd = fdbuf->buf[+(fdbuf->front) % FDBUFSIZE];
451     V(&(fdbuf->mutex));
452     V(&(fdbuf->push));
453
454     return removed_fd;
455 }
```

잠시 fd를 관리하는 함수들에 관해 보면 fdbuf_t 구조체는 buf와 그에 관한 mutex 그리고 semaphore 변수를 가지고 있다. main thread는 insertFdbuf을 통해 생산자의 역할을 하게 된다. 반면 working thread들은 parsing 함수 안에서 removeFdbuf 함수를 이용해 소비자의 역할을 한다. 따라서 insertFdbuf과 removeFdbuf은 세마포어와 뮤텍스를 이용해 synchronization을 해야 한다. 그래야 fd가 데이터가 덮어씌워지는 경우가 없이 혹은 잘못 삭제되는 경우가 없이 작동한다. 버퍼는 circular queue로 구현되어 있다.

```

C stockserver.c ×

project_2 > C stockserver.c > ⚙ initFdbuf()
101 void *parsing(void *vargp)
102 {
103     int receivedByte;
104     char ConvTmp[MAXLINE]; // client로 받은 명령어의 파라미터의 값을 integer로 바꾸기 위한 임시변수
105     int ID, stockNum;      // client로부터 받은 주식의 ID와 개수(num)를 저장하는 변수
106     char *argv[3];          // 명령어와 파라미터를 저장하는 문자열 배열
107     char buf[MAXLINE];
108     rio_t rio;
109
110     // 스레드를 독립적으로 동작시키기 위해 사용
111     // -> detach로 생성된 스레드는 일이 끝난 후 자원을 스스로 반환한다
112     Pthread_detach(pthread_self());
113
114     while (true)
115     {
116         int connfd = removeFdbuf();
117
118         while (true)
119         {
120             // sell id number; buy id number; show; exit
121             Rio_readinitb(&rio, connfd);
122             if ((receivedByte = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
123             {
124                 printf("server received %d bytes\n", receivedByte);
125                 // 명령어와 파라미터를 구분함
126                 for (int i = 0; i < 3; i++)
127                 {
128                     if (i == 0)
129                         argv[i] = strtok(buf, " ");
130                     else
131                         argv[i] = strtok(NULL, " ");
132                 }
133
134                 // 파라미터를 string에서 integer로 바꿈
135                 if (argv[1] != NULL && argv[2] != NULL)
136                 {
137                     strcpy(ConvTmp, argv[1]);
138                     ID = atoi(ConvTmp);

```

다음의 코드는 parsing함수이다. 각 thread에 의해 처리가 되며 소비자역할을 한다. Pthread_detach는 thread의 일이 끝나면 즉, 클라이언트와의 연결이 끝나게 되면 master thread에서 자원을 수거할 필요 없이 커널에 의해 thread의 자원이 회수되도록 만든 것이다. removeFdbuf를 통해 connfd가 fd값을 받게 된다. 그 밖에는 모두 이전과 같은 방식으로 작동한다.

```

C stockserver.c project_2      C stockserver.c ~.../20161385 proj-2... X
Users > ysng_ysng > Desktop > 서강대 > 22년 1학기 과제 > 시스템프로그래밍 > 20161385 proj-2 > project_2 >
240     }
241 }
242
243 void inOrder(char *stockList, pItem pIter)
244 {
245     //-- inOrder는 show명령어를 수행하는데 사용된다. --//
246     char tmpList[100];
247
248     if (treeHead == NULL)
249     {
250         printf("there is no stock_list!\n");
251         return;
252     }
253
254     //-- recursion을 이용해 inOrder를 구현한다 --//
255     if (pIter != NULL)
256     {
257         inOrder(stockList, pIter->leftChild);
258
259         P(&(pIter->mutex));
260         pIter->readCount++;
261         if (pIter->readCount == 1)
262             P(&(pIter->writer));
263         V(&(pIter->mutex));
264
265         //크리티컬 섹션 -> 읽기
266         sprintf(tmpList, "%d %d %d ", pIter->ID, pIter->left_stock, pIter->price);
267         strcat(stockList, tmpList);
268         //크리티컬 섹션 -> 읽기
269
270         P(&(pIter->mutex));
271         pIter->readCount--;
272         if (pIter->readCount == 0)
273             V(&(pIter->writer));
274         V(&(pIter->mutex));
275
276         inOrder(stockList, pIter->rightChild);
277     }
278 }

```

다음은 synchronization이 적용된 함수들의 코드들이다. Show함수에서 사용되는 Inorder함수에서 tmpList에 대해서 뮤텍스가 적용된 것을 볼 수 있다. 왜냐하면 이 함수는 주식목록데이터를 트리를 돌면서 수집해서 클라이언트에게 전송해야 한다. 만일 특정 주식종목에 대해 다른 스레드가 sell이나 buy명령을 진행하고 있다면 그 진행이 끝난 후에야 데이터를 써야한다. 그래야 정확한 주식목록 데이터를 클라이언트에게 전송해줄 수 있다. 또 만일 먼저 show명령에 의해 목록을 쓰고 있고 그 후에 sell, buy명령이 다른 스레드에 의해 진행된다면 반드시 show명령이 모두 끝난 후에야 진행되어야 한다. 만일 둘이 같이 진행되버리면 데이터가 꼬여버리고 예측하지 못한 값이 전송될 수 있다. 그래서 임시 버퍼에 트리 노드에 있는 주식목록을 쓰는 과정이 위와 같이 두 개의 뮤텍스가 적용되어 바뀐 것

이다.

```
C stockserver.c project_2      C stockserver.c ~.../20161385 proj-2/... ×
Users > ysng_ysng > Desktop > 서강대 > 22년 1학기 과제 > 시스템프로그래밍 > 20161385 proj-2 > project_2 > C
287     Rio_writen(fd, stockList, strlen(stockList));
288 }
289
290 void buy(int fd, int ID, int stockNum)
291 {
292     // buy명령을 통해 메모리에 로딩된 주식 정보를 업데이트
293     pItem pIter = treeHead;
294
295     // bst이기 때문에 규칙에 맞게 key값인 ID를 서칭을 한다.
296     while (pIter != NULL)
297     {
298         if (ID == pIter->ID)
299             break;
300         else if (ID > pIter->ID)
301             pIter = pIter->rightChild;
302         else
303             pIter = pIter->leftChild;
304     }
305
306     if [pIter == NULL]
307     {
308         Rio_writen(fd, "stock doesn't exist\n", strlen("stock doesn't exist\n"));
309         return;
310     }
311
312     P(&(pIter->writer));
313     //--잔여 주식 수와 비교한 후 buy명령을 실행해야 한다--//
314     if (pIter->left_stock < stockNum)
315         Rio_writen(fd, "Not enough left stock\n", strlen("Not enough left stock\n"));
316     else
317     {
318         pIter->left_stock -= stockNum;
319         Rio_writen(fd, "[buy] success\n", strlen("[buy] success\n"));
320     }
321     V(&(pIter->writer));
322 }
```

그 다음은 buy함수이다. 여기서 주식의 개수를 감소시키는데 뮤텍스가 적용된다. 여러 스레드가 한 번에 이 연산에 접근해서 값을 바꾸게 되면 값은 예측하지 못하게 바뀔 수 있다.

```

323
324     void sell(int fd, int ID, int stockNum)
325     {
326         // sell명령을 통해 메모리에 로딩된 주식 정보를 업데이트
327         pItem pIter = treeHead;
328
329         // bst이기 때문에 규칙에 맞게 key값인 ID를 서칭을 한다.
330         while (pIter != NULL)
331         {
332             if (ID == pIter->ID)
333                 break;
334             else if (ID > pIter->ID)
335                 pIter = pIter->rightChild;
336             else
337                 pIter = pIter->leftChild;
338         }
339
340         if (pIter == NULL)
341         {
342             Rio_writen(fd, "stock doesn't exist\n", strlen("stock doesn't exist\n"));
343             return;
344         }
345
346         P(&(pIter->writer));
347         pIter->left_stock += stockNum;
348         Rio_writen(fd, "[sell] success\n", strlen("[sell] success\n"));
349         V(&(pIter->writer));
350     }
351

```

마찬가지로 sell함수에서도 writer뮤텍스가 적용되었다. Sell 또한 여러 스레드가 한 번에 이 연산에 접근해서 값을 바꾸게 되면 예측하지 못한 값으로 바뀔 것이다. 따라서 뮤텍스를 이용해 하나의 스레드만이 접근해서 연산을 할 수 있도록 해야 한다.

```

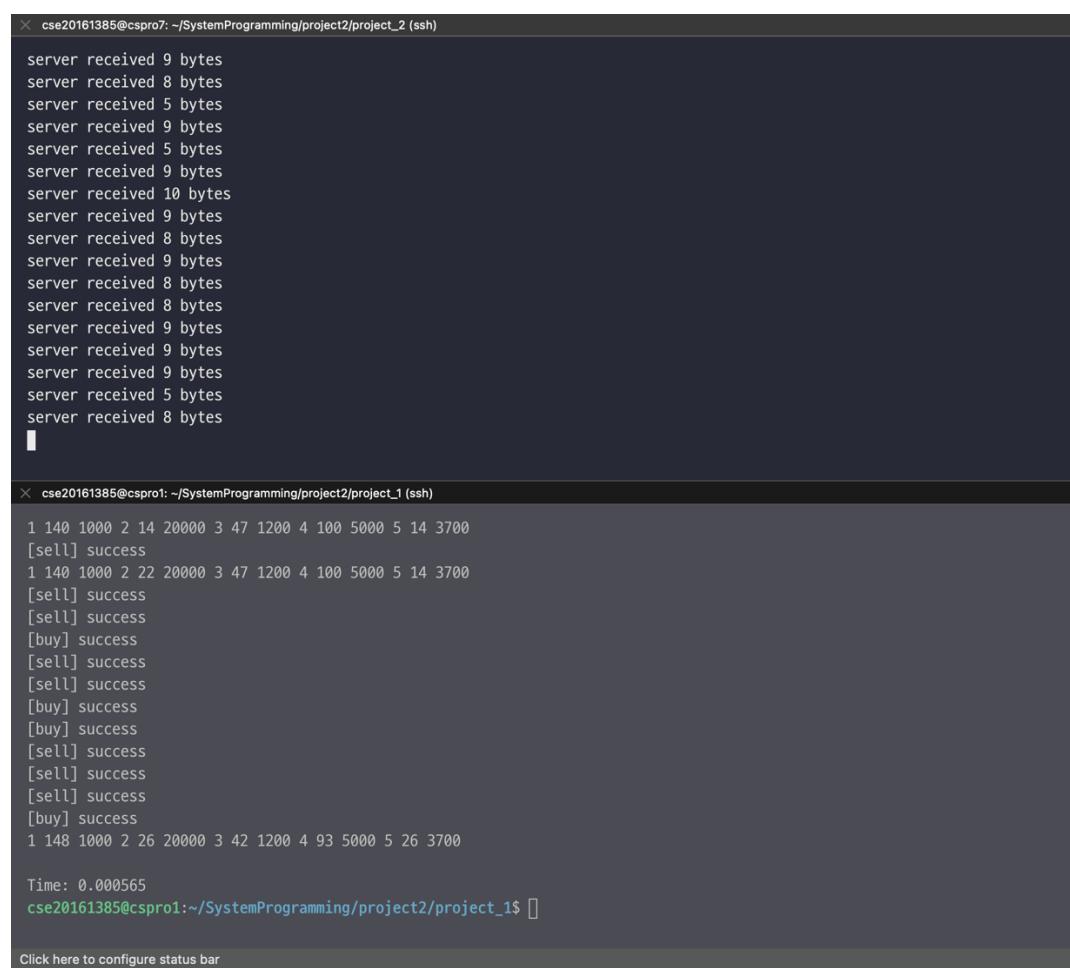
C stockserver.c project_2          C stockserver.c ~.../20161385 proj-2/...
Users > ysng_ysng > Desktop > 서강대 > 22년 1학기 과제 > 시스템프로그래
373
374         P(&(pIter->mutex));
375         pIter->readCount--;
376         if (pIter->readCount == 0)
377             V(&(pIter->writer));
378         V(&(pIter->mutex));
379
380         inorderUpdate(fp, pIter->rightChild);
381     }
382 }
383
384 void updateFile()
385 {
386     P(&fileMutex);
387     //--tree를 돌면서 stock.txt에 write를 한다--//
388     FILE *fp = fopen("stock.txt", "w");
389
390     if (fp == NULL)
391         fprintf(stderr, "can't open stock.txt\n");
392
393     inorderUpdate(fp, treeHead);
394     fclose(fp);
395     V(&fileMutex);
396 }

```

마지막으로 clientExit함수 안에 있는 updateFile함수에서 바뀐 점을 살펴보면 당연하게도 이 부분은 txt파일에 데이터를 업데이트 하는 것이기 때문에 여러 스레드가 한 번에 접근하게 해서는 안된다. 따라서 file mutex를 통해 하나의 스레드만이 값을 바꾸도록 만든다. 이 함수 안에 있는 inorderUpdate함수는 트리를 inorder로 돌아서 기록한다. Inorder함수와의 차이점은 쓰기 연산을 하는 대상만 차이가 있을 뿐 mutex를 적용하는 방식과 recursion을 이용하는 것 마저 똑같다.

4. 성능 평가 결과 (Task 3)

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



```
cse20161385@cspro7: ~/SystemProgramming/project2/project_2 (ssh)
server received 9 bytes
server received 8 bytes
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 9 bytes
server received 10 bytes
server received 9 bytes
server received 8 bytes
server received 9 bytes
server received 8 bytes
server received 8 bytes
server received 9 bytes
server received 9 bytes
server received 9 bytes
server received 5 bytes
server received 8 bytes

cse20161385@cspro1: ~/SystemProgramming/project2/project_1 (ssh)
1 140 1000 2 14 20000 3 47 1200 4 100 5000 5 14 3700
[sell] success
1 140 1000 2 22 20000 3 47 1200 4 100 5000 5 14 3700
[sell] success
[sell] success
[buy] success
[sell] success
[sell] success
[buy] success
[buy] success
[sell] success
[sell] success
[sell] success
[buy] success
1 148 1000 2 26 20000 3 42 1200 4 93 5000 5 26 3700
Time: 0.000565
cse20161385@cspro1:~/SystemProgramming/project2/project_1$
```

다음의 결과들은 성능 분석에 관한 것이다. Cspro1에서는 멀티 클라이언트 프로세스를 실행하였으며 cspro7에서는 주식서버를 실행하여 측정하였다.

먼저 주식 종목이 5개, 요청은 50번으로 고정된 상태에서 클라이언트의 개수를 늘려가는 쪽으로 측정을 하였다. 클라이언트의 개수는 1, 5, 10, 20, 50, 100으로 늘린다. 먼저 event-driven서버에 대해서 클라이언트 수를 늘리며 측정한 결과 시간은 아래의

표와 같이 나타난다.

Event-driven Server	
클라이언트 수	시간
1	0.000173
5	0.000535
10	0.000971
20	0.001672
50	0.004173
100	0.008131

그 다음 thread-based server에 대해서 똑같은 환경과 조건을 가지고 클라이언트 수를 늘려가며 시간을 측정하였다. 그래서 아래와 표와 같이 결과가 나타났다.

Thread-based Server	
클라이언트 수	시간
1	0.000177
5	0.000569
10	0.000917
20	0.001588
50	0.004447
100	0.008012

위의 두 표를 분석했을 때 예측했던 것처럼 thread-based server가 접속하는 클라이언트 수가 많으면 많을 수록 event-driven server보다 조금 더 빠르다는 것을 알 수 있다. 그도 그럴 것이 thread-based server는 multi-thread를 이용하기 때문에 많은 요청을 parallel하게 처리할 수 있고 event-driven server는 fine grained concurrency가 아니기 때문에 클라이언트 수가 많아질수록 처리시간이 늘어날 수 밖에 없다.

다음은 클라이언트의 수를 5개, 요청은 50개로 고정시키고 주식 종목의 수를 5, 10, 20, 50개로 늘려서 실험을 진행해보았다.

Event-driven Server		Thread-based Server	
종목 수	시간	클라이언트 수	시간
5	0.000551	5	0.000548
10	0.000492	10	0.000513
20	0.000552	20	0.000542
50	0.000536	50	0.000566

위와 같이 결과가 나왔는데 주식 종목의 수를 늘리는 것은 두 서버 다 별 차이가 없었다. 그도 그럴 것이 결국 요청하는 양과 클라이언트 수에 따라 처리 시간에 차이가 나지 주식의 수가 늘어난다고 해서 요청하는 양이 늘어나는 것은 아니기 때문이다. 따라서 두 서버 다 일정하게 값이 나온다.

다음은 클라이언트 수 5개, 주식 종목 수 5개로 고정시킨 다음에 클라이언트 당 요청 수를 늘리는 방법으로 두 서버의 처리속도를 아래와 같이 표로 만들어 보았다.

Event-driven Server		Thread-based Server	
요청 수	시간	요청 수	시간
5	0.000520	5	0.000524
10	0.000529	10	0.000529
20	0.000537	20	0.000538
50	0.000559	50	0.000562
100	0.000509	100	0.000546
200	0.000593	200	0.000565

예상과 비슷하게 요청 수에 따라 전체적으로 시간이 늘어나는 추세를 보여준다. 중간에 시간이 약간 줄어드는 부분은 아마 요청의 종류에 따라 조금씩 달라져서 그런 거 같다. 어째튼 전체적으로 보았을 때는 크게 증가하는 건 아니지만 요청 수가 올라가면 올라갈 수록 처리시간이 늘어나며 요청 수가 커지면 커질수록 thread-based server가 조금 더 빠르게 요청을 처리하는 것을 볼 수 있다.

두 서버의 처리속도가 생각보다 차이가 나지 않는 것이 좀 의외였다. Event-driven 방식은 어찌 보면 결국 sequential하게 명령을 처리하는 거나 다름이 없는 것이고 thread-based방식은 다중 스레드를 이용해 parallel하게 일을 처리하는 것이라 결과적으로 병렬 처리되는 방식이 훨씬 빠를 것이라 생각했는데 어찌 보면 인간이 느끼지 못할 정도의 시간 차이 밖에 안나는 게 의외였다. 뮤텍스나 세마포어를 적용하는 부분을 좀 더 디테일하게 주게 된다면 혹은 속도를 더 증가시킬 수 있는 부분이 있다면 속도가 더 차이가 날지도 모른다는 생각이 들지만 지금 내가 적용할 수 있는 부분에서는 이 정도가 최선인 것 같다.