Multicore Programming Project 2

담당 교수 : 박성용

이름 : 고동헌

학번 : 20191564

1. 개발 목표

이번 프로젝트는 2가지 방식으로 Conrruent Stock Server를 만드는 것이 목표이다.

Concurrent 한 Server를 만들기 위해선 client가 connection request를 할 때마다 그에 대응 되는 logical control flow를 만들고, 이 개별적인 flow 안에서 client-server간의 통신이 이루어져야 한다.

첫 번째 방식인 Event-driven Server에선 logical flow를 만들기 위해 state machine 개념을 이용한다. Client로부터 Connection request 를 받을 때마다 connfd 를 생성해주고, 이를 fd_set 자료형의 read_set에 추가해준다. 이후 select 함수를 호출해 kernel이 I/O multiplexing을 해주면서, event 가 발생한 client에 대해 check를 한 뒤, client의 요청에 맞는 작업을 수행해준다. 특정 client에 대한 작업을 수행할 때 client 가 보기엔 concurrent 하게 돌아가는 것처럼 보이지만 실제론 event가 발생한 client를 한 명 한 명 처리해주는 식으로 sequential하게 동작한다.

두 번째 방식인 Thread-Based Server에선 logical flow를 만들기 위해 thread를 이용한다. 특히 이번 Server에선 client로부터 connection request가 올 때마다 thread를 만들고 종료 시키는 방식이 아닌, 미리 특정 개수만큼의 thread를 만들어 두는 식으로 구현된다. 미리 만들어 놓은 thread들은 worker thread로, 실제로 client와의 통신을 수행하는 thread 이다. server가 실행되면 Thread를 미리 지정한 개수만큼 만들고, main thread는 무한 루프에 들어간다. Connection request가 오면 Accept 함수를 호출해 connfd 를 받고, 그 값을 main thread와 worker thread가 공유하고 있는 buffer에 계속 넣어준다. 생성된 각각의 thread 들도 역시 무한 루프를 돌면서, buffer에서 connfd 를 가져올 수 있으면 가져와서 그 descriptor를 통해 client와 통신한다. connection 이 종료되면 다시 loop의 초반부에서 connfd 를 가져올 수 있는 상태가 될 때까지 block 상태가 된다. Concurrent 한 Thread 간의 shared variables인 buffer에서 읽고, 쓰는 행동이 이루어지므로 mutex를 통한 protection 이 필요하고, item과 slot의 개수를 semaphore로 조절해 provider-consumer problem을 해결하였다.

stock 에 대한 정보를 저장할 땐 두 방식 모두 Binary Search Tree를 이용하였다. Server가 실행돼서 무한 루프에 들어가기 전 makeTable() 함수를 호출해 stock.txt를 읽어 자료구조에 저장한다. 전자는 tree의 각 node에 mutex를 사용하지 않지만, 후자는 synchronization을 위해 mutex를 사용한다는 것이 둘 사이의 다른 점이다.

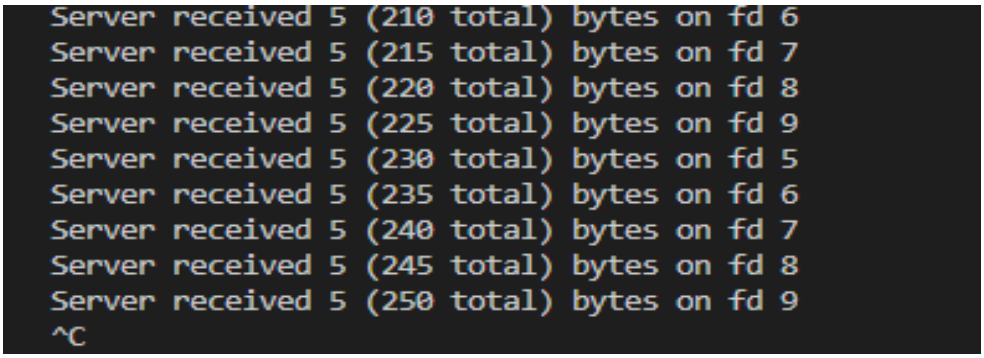
2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

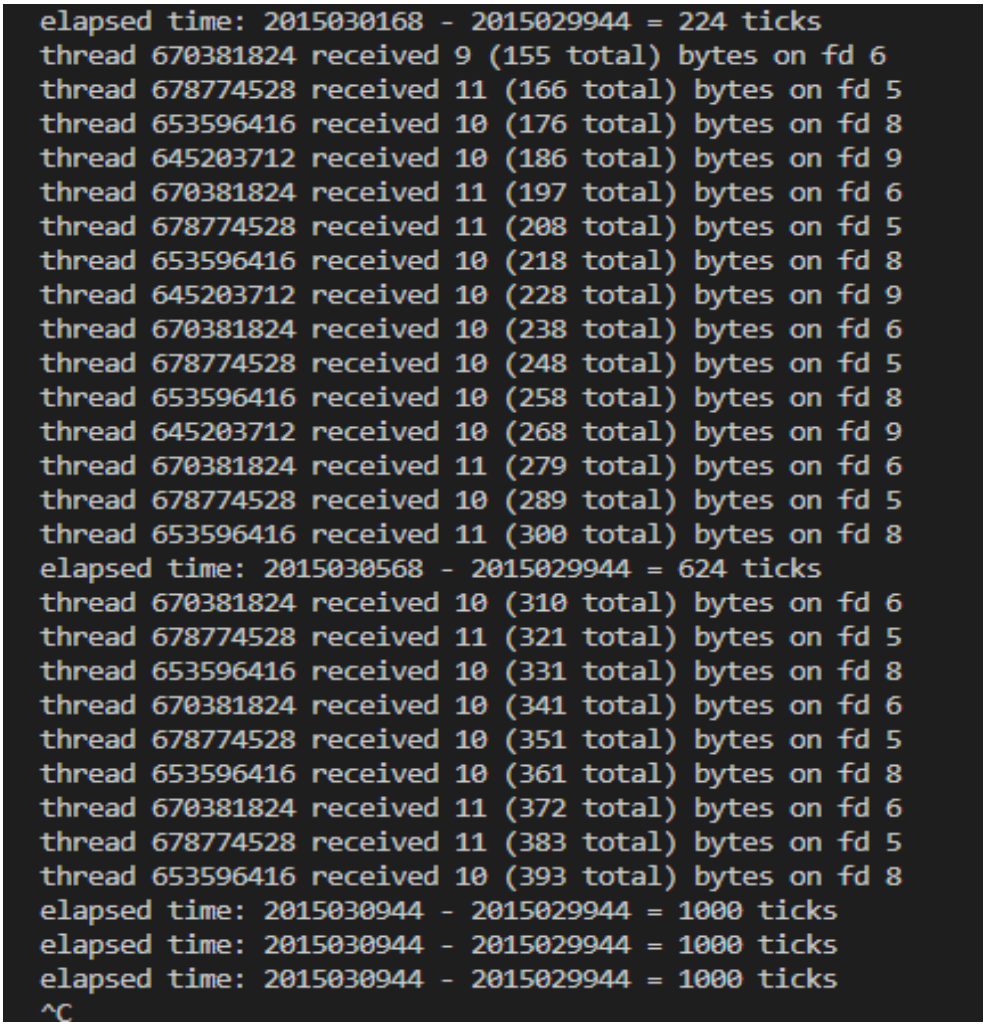
event-driven 방식으로 Server를 구현했을 경우, main에서 무한 루프를 돌면서 "Select 함수 호출 -> Connection Request가 있을 시 client pool 에 추가 -> check_clients 함수 호출 해 event 가 발생한 client의 connection socket descriptor를 통해 순차적으로 통신" 과정을 반복한다. client가 show 명령어만 입력하도록 설정했을 시 보이는 결과는 다음과 같다.



multiclient를 실행했을 때는 중간에 동적으로 어떤 client가 나가고 들어오지 않는다. 따라서 clientfd array에는 fd의 값이 오름차순으로 들어가있고, check_clients() 함수에서는 clientfd의 0번 index부터 event 가 발생했는지 순차적으로 확인하므로, 대부분의 경우 위 사진처럼 file descriptor 의 값이 낮은 것부터 반복적으로 처리된다.

2. Task 2: Thread-based Approach

Thread-Based 방식으로 Server를 구현했을 경우, main thread 에서 무한 루프를 들어가기 전에 MACRO 로 설정한 NTHREADS 값만큼 thread를 미리 생성한다. 그 후 무한 루프에 들어가서 connection request가 있을 경우 main thread와 worker thread 가 공유하는 buf에 connfd 값을 넣어준다. 그리고 worker thread 들은 buf에 들어있는 connfd 를 가져와서 client와 통신을 한다. thread 방식의 경우 kernel 에 의한 scheduling 순서에 따라 concurrent 하게 수행된다.



위의 예시는 5명의 client 가 buy, sell 명령어만 입력하도록 설정하도록 하였을 때의 실행 결과 예시이다. "elapsed time" 은 하나의 thread가 한 명의 client와의 통신이 끝났을 때마다 출력되는데, 예시를 보면 concurrent하게 진행된다는 것을 알 수 있다.

Task 3: Performance Evaluation

Event-Driven Server와 Thread-Based Server의 client 수 및 워크로드에 따른 동시 처리율을 비교한다.

EventDriven Server는 내부적으로 client들의 처리를 sequential 하게 처리하기 때문에 client 수가 늘어남에 따라 수행시간이 sequential 하게 증가할 것이고, Thread-Based Server는 client의 수가 thread pool의 수보다 적을 경우 요청을 concurrent하게 처리할 수 있기 때문에 client수가 늘어날수록 동시 처리율이 증가할 것이다. 하지만 client의 수가 thread pool의 수를 넘는 순간 초과되는 client들은 어떤 thread의 일이 종료될 때까지 기다려야 하기 때문에 동시 처리율이 매우 떨어질 것이다.

client의 명령어는 {buy, sell}, {show} 의 두 개의 그룹으로 나뉠 수 있다. 첫 번째 그룹의 명령어들은 stock tree를 보면서 특정 ID를 가진 주식을 찾아와 재고를 변경해주고, 두 번째 그룹의 명령어는 stock tree의 모든 노드를 보면서 주식의 상태를 client에게 알려줘야 한다. 모든 명령어를 sequential 하게 처리하는 Event-driven server에선 show 명령어에 대한 동시 처리율이 가장 낮을 것이다. show 명령어는 단순히 노드를 읽기만 하기 때문에, 명령어를 Concurrent하게 처리하는 Thread-Based Server에선 reader-reader 간 concurrent한 접근이 가능해 show 명령어에 대한 동시 처리율이 가장 높을 것이다. 시간 측정 구현의 결과 예시는 다음과 같다.

```
18 202 11
19 77 7180
20 23 99
elapsed time : 100030320 microsec
cse20191564@cspro10:~/multiCore/proj2/threadBased$
```

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**

- EventDriven Server 에서는 I/O multiplexing 을 통한 concurrent Server를 위해 active한 clients 들을 pool struct에서 관리한다. pool struct 의 변수로 read_set, ready_set이 있는데, read_set은 현재 active 한 connections 의 descriptor를 저장하고 있고, ready_set은 select() 함수가 호출되고 return 됐을 때 read_set에 있는 descriptor들 중 event가 발생한 descriptor 들을 저장하고 있다.

  read_set에는 client의 connection request의 end point인 listenfd도 저장돼 있다. 만약 Select 함수를 호출한 뒤 ready_set의 listenfd에 해당하는 bit가 1이라면, connection request가 있다는 뜻이므로 Accept 함수를 호출해 connection을 받아준다. 이 때 accept 의 return 값으로 connected fd 가 들어오는데, add_client 함수를 호출해 pool struct에 새로운 client를 추가해준다. add_client에서 read_set에 대한 갱신도 일어난다.

또한, pool struct의 변수로 integer array인 clientfd 가 있는데, 이 array의 특정 index의 값이 -1 이 아니라면 해당 index에는 connected descriptor의 값을 저장하고 있다.

새로운 client가 있을 경우 앞에서 설명한 것처럼 client를 pool에 추가해주고, check_clients 함수를 호출한다. 이 함수에서 cilentfd array 를 maxi 까지 하나씩 보면서 clientfd array의 i번째 index의 값인 cilentfd[i]가 -1이 아니라면, ready_set에 해당 clientfd[i] 가 들어있는지 확인한다. 이때 ready_set에 들어있다면, 해당 client로부터 무언가 요청이 왔다는 소리이므로 그에 대한 처리를 해준다. 이렇게 순차적으로 확인하면서 client에 대한 작업을 수행해준다.

여러 명의 client가 요청을 해도 FD_SETSIZE 크기이상의 client가 동시에 요청하지 않는 이상, I/O multiplexing을 통해 event가 발생한 client들에 대한 작업을 처리 해줌으로써 concurrent 한 Server를 구현할 수 있다.

- ✓ **epoll과의 차이점 서술**

Select 함수는 사용자가 넘겨준 관심 있는 Descriptor들의 Set을 fd_set 자료 구조에 담아 kernel에게 전달하고, Kernel 이 fd_set 중 Event가 발생한 descriptor를 return 해준다. 사용자는 관심 있는 모든 descriptor들을 FD_ISSET으로 살펴보면서, Event가 발생했으면 그에 대한 처리를 해준다. 하지만 일반적으로 FD_SETSIZE가 1024로 제한돼있고, 관심있는 모든 Descriptor들에 대해 FD_ISSET으로 하나씩 확인하는 것도 비효율적이다.

epoll은 Select의 단점을 보완한 I/O multiplexing 방법이다. Select 처럼 호출할 때마다 fd_set과 같은 관심 있는 descriptor 들의 set을 kernel에게 전달하지 않고, kernel이 descriptor들의 set을 직접 관리한다. 또한, Select 함수가 관리할 수 있는 descriptor의 Size가 1024인 반면, epoll을 사용하면 parameter로 넘겨주는 size값에 맞는 descriptor 저장소가 만들어져 kernel이 관리해준다. Event에 대한 발생 처리를 체크할 때도 모든 descriptor들을 순회하며 check 하지 않고도 event 가 발생한 descriptor들을 파악할 수 있어 효율적이다.

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

  ✓ **Master Thread의 Connection 관리**

- 이번 Thread-Based Server에서 masterThread는 main thread 이다. main thread는 무한 루프에 들어가기 전 NTHREAD의 값만큼 thread를 미리 생성한다.(prethread) 그 후 무한 루프에서 Accept함수의 호출을 반복하는데, 이때 connection request 가 있어 accept가 return 된 경우, return 값인 connected fd 를 sbuf struct의 buf 에 넣어준다. 그리고 만들어진 worker thread는 이 buf에 있는 connfd 를 가져와 client와 통신한다.

  이때 buf 는 concurrent한 thread들이 share 하는 variable 이고, main과 worker 는 각각 provider – consumer 관계에 있으므로 mutex 를 이용해 buf에 대한 mutually exclusive access을 보장해줘야 한다. 또한, main thread 에서 buf에 connfd 를 넣을 수 있는 경우는 buf에 빈 공간이 있는 경우이고, worker thread에서 buf에서 connfd를 가져올 수 있는 경우는 buf에 connfd 가 하나라도 들어있는 경우이다. 따라서 단순히 mutex를 통한 mutually exclusive access 뿐만 아니라, counting semaphores를 통해 main thread와 worker thread가 buf의 slot과 connfd(item)의 수에 따라 올바르게 동작할 수 있도록 만들어야 한다.

  결론적으로 Master Thread는 connection request가 들어올 때마다 semaphore에 따라 buf에 connfd를 넣어주고, 각 worker thread 들이 이를 하나씩 가져가는 형태로 Connection 이 관리된다.

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

- main thread(master thread)에서 MACRO 값인 NTHREAD 의 값만큼 thread를 미리 생성한다. client의 connection request가 올 때마다 thread를 생성하면 overhead가 생기기 때문이다. thread를 생성할 때 parameter로 넘겨주는 것은 따로 없다. 이 프로그램에서 thread로 실행되는 함수는 void* thread(void* vargp) 이다. thread의 초반부에 pthread_detach 함수를 호출해 해당 thread를 다른 thread와 independant 하게 수행되도록 만든다. 그 후 worker thread 역시 무한 루프를 돌아가면서, 한 번의 loop에 한 명의 client와 통신을 한다. loop 의 초반에서 sbuf_remove 함수를 호출해 buf 에 만약 connfd 가 있고 이를 가져올 수 있다면 이 descriptor를 가져와 해당 client 와 통신하고, 가져올 수 없다면 가져

올 수 있는 상태가 될 때까지 해당 위치에서 block 된다.

성공적으로 connfd 를 가져왔다면 processingCommand() 함수를 호출해 client와 통신을 수행한다. processingCommand 함수에서 호출하는 함수 들은 모두 local variables 만 사용하거나, stock tree 등과 같이 shared variables 을 사용하는 경우 mutex를 통해 protection 하였기 때문에, thread-safe 한 함수들이다.

각 thread가 한 명의 client와 통신이 끝나면 다시 loop로 돌아와 가져온 connfd 를 close하고, 다시 connfd 를 buf에서 가져올 때까지 block 된다. 만약 main thread에서 미리 생성하는 thread의 개수를 늘리고 싶다면 NTHREAD의 MACRO 의 값을 변경하면 된다.

- **Task3 (Performance Evaluation)**

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

  - 얻고자 하는 Metric은 다음과 같이 크게 네 가지로 나뉜다.

  1. Event-Driven Server 에서 각각의 세 가지 워크로드 유형에 대해서, client 수가 변함에 따른 동시 처리율 변화

  2. Thread-Based Server에서 세 가지 워크로드 유형에 따른 동시 처리율 변화

  3. Thread-Based Server에서 Client 수에 따른 동시 처리율 변화

  4. client수 변화에 따른 Event-Driven Server와 Thread-Based Server의 동시 처리율 비교

  1번 Metric의 경우, Event-Driven Server에서 Client 수 및 각 워크로드 유형에 따라 달라지는 동시 처리율 변화를 확인하기 위해 설정하였다. Event-Driven Server에선 내부적으로는 Concurrent 하게 돌아가지 않고, event가 발생한 descriptor에 대해 순차적으로 처리하는 형식으로 구현되므로 client 수가 늘어날수록 처리 속도도 sequential하게 증가할 것이라고 생각하였다.

  워크로드 유형에 따른 동시 처리율 변화를 확인하는 이유는 buy, sell, show 명

령어에 대한 처리를 구현하는 방식이 다르기 때문이다. buy, sell 같은 경우 주어진 자료구조를 살펴보며 명령어로 들어온 stock ID에 해당하는 node가 있을 시 해당 stock의 재고를 바꾸고 즉시 return 될 수 있는 반면, show 명령어는 반드시 자료구조의 모든 노드를 살펴보면서 client에게 현재 있는 주식의 상태를 제공해야 한다. 따라서 client의 모든 명령어가 show일 때 동시 처리율이 가장 낮아질 것이라고 생각하였다. 1번은 이 두 가지 내용을 검증하고자 정의한 Metric이다.

Thread-Based Server 같은 경우 실제 각 thread 들이 concurrent 하게 동작하므로, shared variables인 stock tree의 각 노드를 semaphore로 protection 해주어야 한다. 이 때 stock tree에 write를 하는 buy, sell 명령어와 달리, show 명령어는 stock tree의 특정 node를 읽기만 한다. 이번 Thread-Based Server에선 readers-writers problem을 해결하였기 때문에, sell, buy로 구성되거나 또는 모든 명령어들이 랜덤하게 들어오는 query를 처리할 때에 비해 show로만 구성된 명령어를 처리할 때의 처리율이 더 높을 것이다. 2번 Metric을 통해 이 내용을 검증해 볼 수 있다.

이번 프로젝트에서 만드는 Thread-Based Server는 preThreading 방식을 이용한다. 즉, client로부터 connection request가 올 때마다 thread를 만들어서 통신하도록 하는 것이 아닌, 미리 정해진 수(NTHREAD)만큼 thread를 만들어 놓고 만든 worker thread를 통해 client와 통신한다. 이 방식은 overhead가 줄어든다는 장점이 있지만, 만약 client가 NTHREAD보다 많이 들어온 경우 NTRHEAD를 초과하는 client들은 어떤 한 thread가 종료되기 전까진 서버로부터 아무런 응답을 받지 못한다는 단점이 있다. Metric3 는 이러한 단점을 확인하기 위해 정의하였다.

Event-Driven Server, Thread-Based서버는 client 수의 증가에 따른 동시 처리율이 다른 양상으로 변화할 것이다. 위에서 설명하였듯이 두 서버의 구현 방식이 다르기 때문이다. 4번 Metric을 통해 이를 검증할 수 있다.

✓ Configuration 변화에 따른 예상 결과 서술

1번 Metric은 Event-Driven Server에서 client 수와 워크로드에 따른 동시 처리율을 측정하기 위한 것이다. Event-Driven Server에서 client의 수가 증가함에 따라 처리 속도가 조금씩은 더 걸리긴 하겠지만, client의 증가율에 비해선 굉장히 미미한 양일 것이기 때문에 client의 수가 증가할 수록 동시 처리율이 높아질 것

이라고 예상할 수 있다. 또한, 워크로드 측면에선 show 명령어의 동시 처리율이 가장 낮을 것이다. 주식의 상태를 제공해주기 위해 반드시 모든 노드를 살펴야 하기 때문이다.

2번 Metric의 경우 show 명령어로만 구성된 query의 동시 처리율이 가장 높을 것이다. show 명령어의 경우 모든 thread 들이 stock tree에 대한 read만 수행해 block 되지 않고 concurrent하게 진행될 수 있기 때문이다. 이와 반대로 buy, sell 명령어로만 구성된 query의 동시 처리율이 가장 낮을 것이다. 두 명령어는 stock tree에 대한 write를 수행하기 때문에, 각 thread 들이 P operation에 의해 block 되는 시간이 많아질 것이기 때문이다.

만약 client의 수가 NTHREAD보다 작거나 같다면, client 수가 증가함에 따른 동시 처리율이 계속 올라갈 것이다. concurrent하게 동작하므로 client의 수가 NTRHEAD 범위내에서 증가해도 처리 시간이 비슷할 것이기 때문이다. 하지만 client의 수가 NTHREAD 수보다 많아지는 순간 동시 처리율이 매우 떨어질 것이다. 초과되는 client에 대한 처리는 block 될 것이기 때문이다.

Event-Driven Server, Thread-Based서버는 client 수의 증가에 따른 동시 처리율이 다른 양상으로 변화할 것이다. 전자는 client의 요청을 sequential 하게 처리하고, 후자는 정해진 thread-pool 내에선 concurrent하게 처리하기 때문이다. 만약 client의 수가 NTHREAD보다 작다면, thread-based server의 동시 처리율이 높을 것이고, NTHREAD보다 커지면 event-driven Server의 동시 처리율이 높을 것이다.

C. 개발 방법

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

task1, task2에선 둘 다 binary search tree를 이용해 stock list를 저장한다. stock tree를 구성하고, 명령어들을 실질적으로 처리해주는 부분은 모두 stockManager.c, stockManager.h 파일에 정의 및 구현 돼있다. 공통적으로 사용되는 구조체, 변수 및 함수는 다음과 같다.

1. task1, task2에서 공통적으로 사용되는 변수 및 함수

- **구조체 및 변수**

- **item Struct**

: 멤버로 ID, cnt, price, left, right를 가지며, 각각은 주식의 ID, 재고, 가격, 해당 node의 left-right child pointer를 나타낸다.

- **FILE*readFile, writeFile**

: readFile은 stock.txt를 읽어 stock tree를 구성할 때 쓰이고, writeFile은 client에 대한 처리가 끝난 후 stock.txt를 update 할 때 사용된다.

- **item* stockTable**

: server의 stock tree의 첫 번째 노드를 가리키고 있는 pointer 변수이다. buy, sell, show등의 명령어를 수행할 때 이 변수를 parameter로 넘겨줘 노드에 접근해서 명령어에 대한 처리를 해준다.

- **long long byte_cnt**

: server 가 client로부터 받은 byte의 수를 누적해서 저장하고 있는 변수이다. 이를 통해 server 가 client로부터 제대로 데이터를 받고 있는지 확인할 수 있다.

- **SHOW, BUY, SELL, EXIT macro**

: client로부터 받은 명령어를 parsing하고 이에 따라 작업을 처리할 때 사용하는 매크로 값이다.

- **함수**

1. item* makeTable() : main 함수에서 무한 루프에 들어가기 전 호출되는 함수이다.

stock.txt를 읽으면서 insertStock함수를 호출해 stock tree를 구성하여 tmptable 변수에 저장하고, 이를 return 해준다.

2. void insertStock(int ID, int cnt, int price, item price)**

: makeTable 함수에서 stock.txt를 한 줄씩 읽을 때마다 호출되는 함수로, parameter로 넘어온 ID를 갖는 주식을 stock tree에 넣어준다.

3. item* findStock(int ID, item* cur)

: buy, sell에 대한 명령어를 처리할 때 특정 ID를 가진 stock node가 필요하다. findStock 함수는 parameter로 넘어온 ID를 가진 stock node를 찾아서 caller에게 return 해준다. 재귀적으로 구현돼있다.

4. int buyStock(int ID, int buyNum, item* stockTable)

: buy 명령어를 처리해주는 함수이다. parameter로 넘어온 ID를 가진 stock을 findStock 함수를 통해 찾고 해당 stock의 재고를 갱신해준다. 만약 사고자 하는 stock의 수가 재고보다 많다면 0을 return 해서 server 가 "Not enough left stock" message를 client 에게 보낼 수 있도록 한다.

5. void sellStock(int ID, int sellNum, item* stockTable)

: sell 명령어를 처리해주는 함수이다. parameter로 넘어온 ID를 가진 stock을 findStock 함수를 통해 찾고 해당 stock의 재고를 갱신해준다.

6. void getTable(item* cur, char (*buf)[MAXLINE])

: show 명령어를 처리해주는 함수이다. 재귀적으로 모든 노드를 보면서 주식의 ID, cnt, price를 buf에 concatenate에 시켜주고, server에서는 buf에 담긴 주식의 상태를 client한테 보내준다.

7. void updateTable(item* cur)

: client에 대한 처리가 끝나고 서버를 종료시킬 때, client와의 통신 결과를 stock.txt에 반영해 저장해주는 함수이다. 재귀적으로 stock tree를 순회하면서 각 노드의 ID, cnt, price를 stock.txt에 적어준다.

8. int parseCommand(char* buf)

: buf에는 client로부터 들어온 명령어를 저장하고 있다. 이 함수에선 이 명령어가 무엇인지 파악하고, 그에 따른 매크로값을 리턴해준다.

9. int getID(char* buf)

: 명령어가 show나 exit이 아닐 경우 호출되는 함수이다. buf에 저장된 명령어를 받아서 명령어에 포함된 주식의 ID를 parsing해 return 해준다.

10. int getStockNum(char* buf)

: 명령어가 show나 exit이 아닐 경우 호출되는 함수이다. buf에 저장된 명령어를 받아서 명령어에 포함된 주식의 ID를 parsing해 return 해준다.

11. void buy(int connfd, int ID, int buyNum)

: 명령어가 buy 일 경우 호출되는 함수이다. 내부적으로 buyStock 함수를 호출하고, return 값이 0이면 parameter인 connfd를 이용해 "Not enough left stock" message를 client에게 보내고, 1이라면 "[buy] success" message를 보낸다.

12. void sell(int connfd, int ID, int sellnum)

: 명령어가 sell 일 경우 호출되는 함수이다. 내부적으로 sellStock 함수를 호출하고 connfd를 이용해 "[sell] success" message를 보낸다.

13. void show(int connfd)

: 명령어가 show일 경우 호출되는 함수이다. 내부적으로 getTable함수를 호출해 local variables인 buf에 주식의 상태를 담아온 뒤, connfd 를 통해 client에게 주식의 상태를 보낸다.

14. void sigint_handler(int sig)

: Server를 종료시킬 때 ctrl+c를 통해 종료 시키는데, 이때 종료되기 전 stock.txt를 update하기 위해 작성한 handler이다. 내부에서 stock.txt를 open하고, updateTable함수를 호출해 stock.txt를 갱신해준다.

2. Event-Driven Server

- **구조체 및 변수**

1. pool structure

- **함수**

1. void init_pool(int listenfd, pool *p)
2. void add_client(int connfd, pool* p)
3. void check_clients(pool* p)

- 공통적으로 사용되는 변수 및 함수를 제외하고 추가된 구조체와 함수는 위와 같다.

  pool struct의 경우 Event-Driven Server에서 connection 이 맺어진 client들과 I/O multiplexing을 통해 통신하기 위한 정보들을 저장하고 있는 구조체이다. 멤버인 read_set, ready_set, maxfd 는 Select 함수를 호출할 때 필요하다. read_set에는 connection이 맺어져 active한 descriptor들 및 connection request의 end point인 listenfd가 bit vector 형태로 저장 돼있다. ready_set 에 read_set을 대입하고 Select함수를 호출하면 이 descriptor들의 목록 중 event가 발생한 descriptor들이 ready_set에 저장된다. clientfd 배열은 active descriptor들의 값을 int array 형태로 저장하고 있다. clientrio 배열은 buffered I/O를 사용하기 위한 rio_t 구조체들을 clientfd array index에 대응되도록 저장해, client와 통신 시 발생할 수 있는 short count 문제를 해결한 Rio_readlineb 함수를 사용할 수 있도록 해준다.

  init_pool 함수는 무한 루프에 들어가기 전, pool structure의 초기 세팅을 해준다.

pool의 clientfd array의 특정 index의 값이 -1이면, 해당 index에 대응되는 client가 없다는 뜻이다. 서버를 처음 시작했을 땐 연결된 client가 없으므로, 이 함수에서 clientfd의 모든 값을 -1로 set 해준다. 또한 read_set에 listenfd 가 들어있어야 Select 함수를 통해 connection request가 왔는지 알 수 있기 때문에 read_set에 listenfd를 set 해준다.

add_client함수는 connection request가 온 client를 client pool 에 추가해주는 함수이다. rio_t 구조체를 초기화해주고, read_set에 accept 함수로부터 return 받은 connfd를 넣어준다. 또한, 필요 시 maxi의 값도 갱신해준다.

check_clients 함수는 event가 발생한 client들로부터의 요청을 처리해주는 함수이다. main의 무한 루프에서 Select함수가 끝나면 ready_set에 event가 발생한 descriptor들이 1로 set 돼있다. check_clients 함수는 loop의 가장 마지막에 호출되고, ready_set의 정보를 이용해 요청을 처리해준다. client pool의 clientfd array를 index 0부터 maxi 까지 하나씩 살펴보면서 해당 배열에 들어있는 descriptor의 값이 ready_set에 들어있으면, Rio_readlineb 함수를 통해 명령어를 받아들인다. 이후 명령어를 parsing하고 그에 맞는 함수를 호출해서 명령어를 처리한다. 이 때 client가 connection을 close해서 EOF가 오거나, client가 exit 명령어를 입력했을 경우 해당 connfd 를 read_set 및 clientfd 에서 제거하고, Close를 통해 connection을 종료한다.

3. Thread-Based Server

- **구조체 및 변수**

1. sbuf struct
2. sem_t s_for_byte_cnt
3. item struct의 변경

- **함수**

1. sbuf_init(sbuf_t* sp, int n)
2. sbuf_deinit(sbuf_t *sp)
3. sbuf_insert(sbuf_t *sp, int item)
4. sbuf_remove(sbuf_t *sp)

5. void* thread(void* vargp)

6. void processingCommand(int connfd)

7. static void init_byte_cnt(void);

8. getTable, buyStock, sellStock, updateTable 함수의 변경

- Thread-Based Server에선 Event-Driven Server와 달리, I/O multiplexing 을 사용하지 않는다. 따라서 read_set, ready_set과 같은 multiplexing을 위한 정보를 저장하던 변수들은 더 이상 필요가 없다. 이에 따라 active한 connection을 관리하는 방식도 달라지기 때문에, Event-Driven Server에서 사용하던 함수들은 모두 사용할 수 없다.

  Thread-Based 방식의 Server를 만들기 위해 새롭게 추가된 구조체는 sbuf 이다. 이번 프로젝트에선 connection을 관리해주는 master thread인 main thread가 있고, 실제 client와의 통신을 하며 명령어를 처리해주는 worker thread 가 있다. main thread에선, connection을 accept하며 return 받는 connfd를 각 thread가 공유하는 buffer에 넣어주고, worker thread는 이 buffer에서 connfd 를 하나씩 가져와서 일을 수행한다. 즉, Producer-Consumer 방식으로 동작하게 되고 sbuf 는 이 방식을 구현하기 위한 struct 이다. 내부적으로 buffer는 circular queue 로 관리하며, buffer를 protection하기 위한 mutex, main thread와 worker thread를 scheduling 하기 위한 counting semaphore인 items, slots가 있다.

  s_for_byte_cnt 는 semaphore 변수로, 모든 thread가 공유하는 byte_cnt 변수의 protection을 위해 선언한 변수이다.

  sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove함수는 모두 sbuf structure 와 관련 있는 함수들이다. init 함수에선 buffer 및 mutex들을 초기화 해주고, deinit은 sbuf를 free해주는 함수이다. insert, remove함수는 buffer에 connfd 를 넣어주고, buffer로부터 connfd를 하나 가져와 return 해주는 함수이다. 내부적으로 모두 mutex와 counting semaphore를 이용하고 있다.

  thread 함수는 생성된 worker thread들이 실행하는 부분이다. 무한 루프를 돌며 connfd 를 buffer에서 가져오고, 가져온 connfd를 통해 client와 통신한다. 통신이 끝나면 Close 함수를 호출해 connection을 종료하고, 다시 connfd 를 가져올 수

있을 때까지 block된다.

processingCommand 함수는 worker thread에서 호출하는 함수로, 실질적으로 client의 요청을 처리해주는 함수이다. Event-Driven Server의 check_clients 함수와 유사해 보이지만, event가 발생한 client에 대해 하나의 작업만 수행하고 다음 client의 작업을 수행하는 것과 달리 이 함수에선 connfd를 통해 통신하는 client와의 connection이 종료될 때까지 while loop를 돌면서 명령어들을 처리해준다. 또한 shared variables인 byte_cnt를 update할 때 P와 V operation으로 protection을 해주어야 한다는 점도 다르다. 그 외의 부분은 유사하다. 명령어를 parsing해서 그에 맞는 함수를 불러 요청을 처리해준다. 이 때 processingCommand 함수와 이 함수에서 호출하는 buy, sell 등의 함수들은 모두 local variables을 사용하거나 mutex에 의해 protection이 돼있으므로 thread-safe한 함수들이다.

init_byte_cnt 함수는 Pthread_once 함수에 의해 프로그램 실행 중 단 한번만 실행되는 함수이다. 내부에서 byte_cnt를 0으로 초기화하고 s_for_byte_cnt를 초기화해준다.

Thread-Based Server에선 각 thread 들이 concurrent 하게 실행되고, 모든 thread들이 함수를 통해 stock tree의 모든 노드에 접근할 수 있다. 따라서 synchronization이 필요하고, 이를 위해 item struct에 mutex 변수를 추가하였다. 각 node 가 tree에 insert될 때 이 mutex의 값은 1로 초기화 되며, 해당 노드에 대한 read와 write가 interleaving 되지 않도록 만든다.

node의 protection을 위한 mutex만 아니라 readers-writers 문제도 해결해야 한다. cnt_mutex와 readcnt는 이를 위한 변수들이다. cnt_mutex는 readcnt를 protection 하기 위한 mutex이다. readcnt의 초기값은 0이고, 어떤 thread가 해당 node를 read할 때 값이 증가되고 read가 종료되면 값이 감소한다.

stock tree 에 접근하는 모든 함수도 semaphore를 사용함에 따라 변경됐다. sellStock 함수와 buyStock 함수는 node에 write를 해야 하기 때문에, write 부분 앞뒤로 P, V operation을 사용하였다. getTable 함수의 경우 stock tree를 read만 하기 때문에, show 명령어와 같이 read 만 하는 thread간에는 concurrent하게 수행될 수 있도록 하면서, 해당 node에 write 되지 못하도록 해야 한다. 이는 위에서 추가한 readcnt와 cnt_mutex에 의해 구현된다. 첫 번째 reader인 경우에만 node에 대한 P(&mutex) operation을 수행하고, 마지막 reader인 경우에만 V(&mutex) operation을 수행한다.

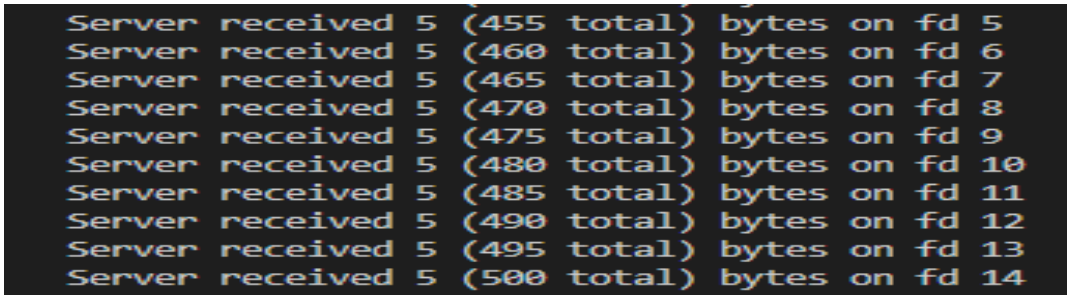
UpdateTable 함수는 프로그램 수행 중 SIGINT signal을 받을 때 수행되는데, 혹시 잘못된 결과를 stock.txt에 쓰지 않도록 하기 위해서 P와 V Operation으로 protection을 해주었다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**

1. Task1

- Event Driven Server의 경우, ready set에 들어있는 event 가 발생한 descriptor를 순차적으로 하나씩 살펴보면서 요청을 처리해준다. 따라서 이번 multiclient 실행 파일과 같이 중간에 어떤 client가 나가거나 들어오지 않고 계속해서 요청만 하는 경우, 대부분은 descriptor의 번호가 낮은 것부터 하나씩 처리된다는 점을 알 수 있다. 실행 예시는 다음과 같다.



2. Task2

- Thread-Based Server의 경우, 각각의 worker thread들이 kernel 에 의해 scheduling 되어 랜덤하게 수행된다. 따라서 어떤 thread가 다른 thread에 비해 일찍 종료될 수도 있고, scheduling 에 따라 모든 thread가 같이 종료될 수도 있다. 실행 예시는 다음과 같다.

```
thread -1073809664 received 5 (215 total) bytes on fd 8
thread -1048631552 received 5 (220 total) bytes on fd 5
thread -1082202368 received 5 (225 total) bytes on fd 9
thread -1057024256 received 5 (230 total) bytes on fd 6
thread -1065416960 received 5 (235 total) bytes on fd 7
thread -1073809664 received 5 (240 total) bytes on fd 8
thread -1048631552 received 5 (245 total) bytes on fd 5
thread -1082202368 received 5 (250 total) bytes on fd 9
```

3. Client 실행예시

- Client가 buy 명령어를 통해 주식의 구매를 성공한 경우, "[buy] success" 가 출력되고, buy가 실패한 경우 "Not enough left stock" message가 출력된다. sell 명령어가 성공할 경우 "[sell] success" message가 출력된다. show 명령어를 실행하면 현재 주식의 상태가 리스트의 형태로 출력된다. 실행 예시는 다음과 같다.

```
10 434 4445
11 1444 4443
12 124 400
13 322 3999
14 444 499
15 488 300
16 200 100
17 33 322
18 202 11
19 77 7180
20 23 99
[sell] success
[buy] success
[buy] success
```

4. 성능 평가 결과 (Task 3)

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

● 시간 측정 방법

- elapsed time은 multiclient에서 gettimeofday 함수를 사용하여 측정하였다. 모든 client의 요청이 시작하고 끝나는 지점은, multiclient 에서 fork를 해서 client를 생성하고 waitpid를 통해 모든 child process 들의 reaping이 끝난 시점이다. 따라서 fork를 통해 client를 생성하기 전 gettimeofday(&start, 0) 을 호출하고, reaping이 끝난 후에 gettimeofday(&end, 0) 을 호출한 후, end와 start의 차이를 통해 elapsed time을 microsecond 단위로 구하였다. multiclient를 수정하지 않기 위해 실험할 땐 별도로 mymulti.c 라는 파일을 만들어서 Configuration을 바꿔가면서 실험을 진행하였다. 함수 사용 예시는 다음과 같다.

-
```
    gettimeofday(&start, 0);    /* mark the start time */
/*  fork for each client process    */
    while(runprocess < num_client){
```

-
```
for(i=0;i<num_client;i++){
    waitpid(pids[i], &status, 0);
}

gettimeofday(&end, 0);  /* mark the end time */
unsigned long e_usec;

e_usec = ((end.tv_sec * 1000000) + end.tv_usec)-((start.tv_sec * 1000000) + start.tv_usec);
printf("elapsed time : %lu microsec\n",e_usec);
```

● 1번 Metric

- *Event-Driven Server 에서 각각의 세 가지 워크로드 유형에 대해서, client 수가 변함에 따른 동시 처리율 변화*

우선, 1번 Metric을 측정할 때 사용한 configuration은 다음과 같다.

```
#define MAX_CLIENT 100
#define ORDER_PER_CLIENT 10
#define STOCK_NUM 10
#define BUY_SELL_MAX 10
```

또한, 각 워크로드 별 동시 처리율을 측정할 때 client 측 option 다음과 같이 바꾸면서 실험

을 진행하였다. 이 option은 각 Metric에서 워크로드 유형에 따른 실험을 할 때 공통적으로 적용되는 사항이다.

(i) buy, sell, show 명령어를 랜덤으로 요청하는 경우

```
int option = rand()%3;
```

(ii) sell, buy 명령어만 요청하는 경우

```
int option = rand()%2;
option += 1;
```

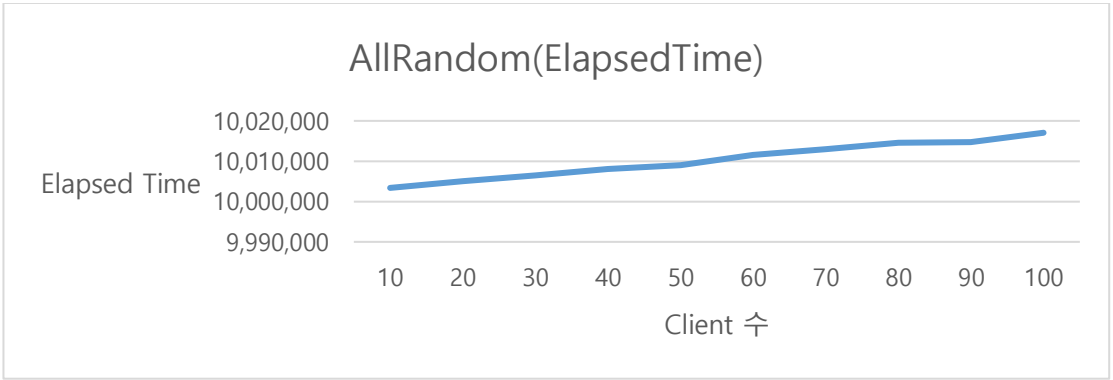
(iii) show 명령어만 요청하는 경우

```
int option = 0;
```

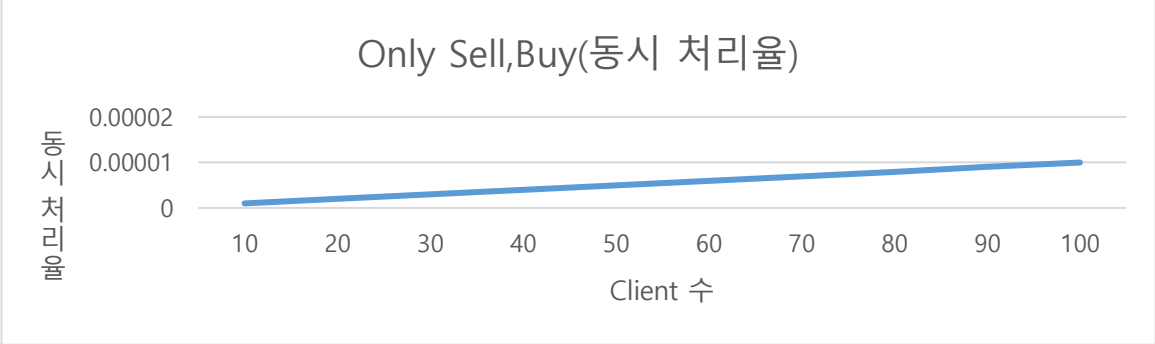
Client의 수는 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 으로 바꿔가며 실험하였다. elapsed time 측정 단위는 ms(마이크로세컨드) 이고,

동시 처리율은 (client의 수 / Elapsed_Time) 로 정의된다. 실험 결과 아래와 같은 그래프가 나왔다.

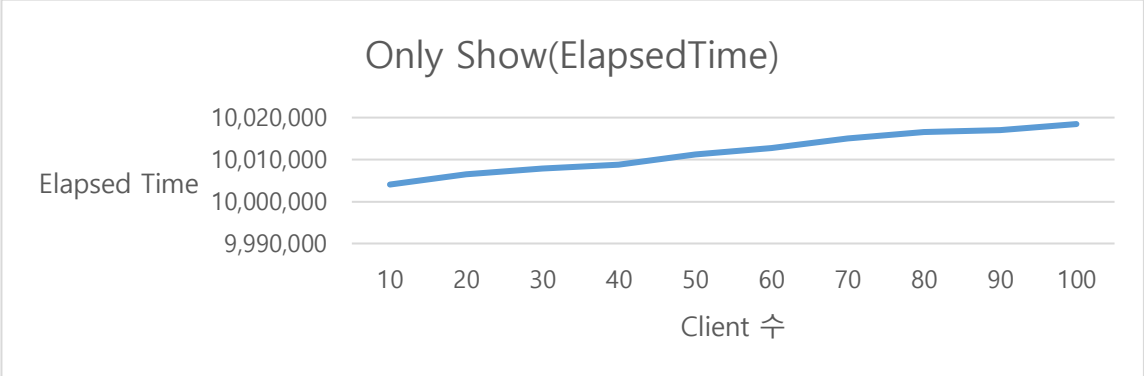
1. buy, sell, show 명령어를 랜덤으로 사용하는 경우 elapsed time 및 동시 처리율 변화

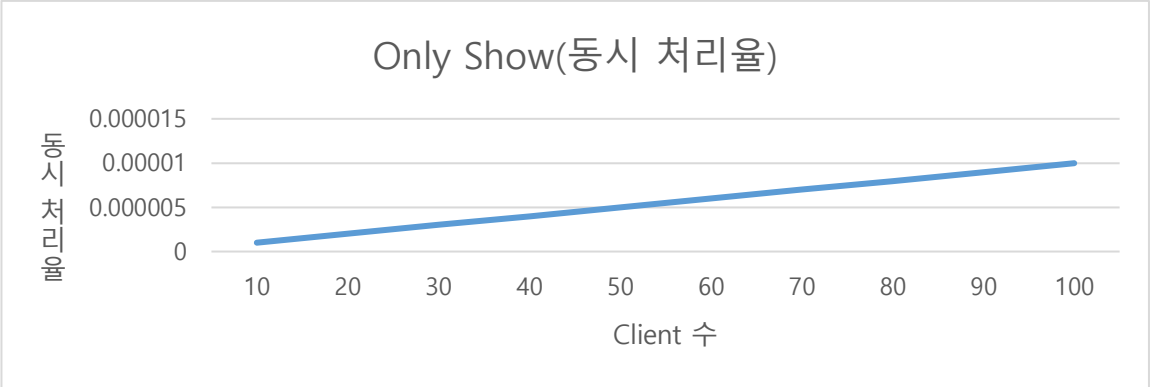


2. sell, buy 명령어만 요청하는 경우 elapsed time 및 동시처리율 변화



3. show 명령어만 요청하는 경우 elapsed time 및 동시처리율 변화





- 그래프를 통해 공통적으로 확인할 수 있는 건 client의 수가 늘어남에 따라 Elapsed Time이 sequential 하게 늘어나고, 앞에서 예상한 바와 같이 elapsed time의 증가율이 client의 증가율에 비해 굉장히 미미하기 때문에 동시 처리율도 올라간다.

- 이번 프로젝트는 자료구조의 크기가 제한돼 있어 각 워크로드 별 동시 처리율의 변화는 눈으로 확인할 만큼 두드러지게 나타나지는 않고 있다. 하지만 각 워크로드 별 Elapsed Time 그래프를 클릭해 각 점들의 값을 확인해 보면, Elapsed Time은 Show 명령어, 모든 명령어, buy/Sell 순으로 줄어든다. buy, sell 명령어에 비해 show 명령어는 모든 노드를 확인해야 하기 때문이다.

  같은 Client수에서 ElapsedTime이 증가했다는 것은 동시 처리율이 그만큼 낮아졌다는 뜻이므로, Event-Driven Server에선 예상대로 Show 명령어의 동시 처리율이 가장 낮게 나왔다.

- **2번 Metric**

- *Thread-Based Server에서 세 가지 워크로드 유형에 따른 동시 처리율 변화*

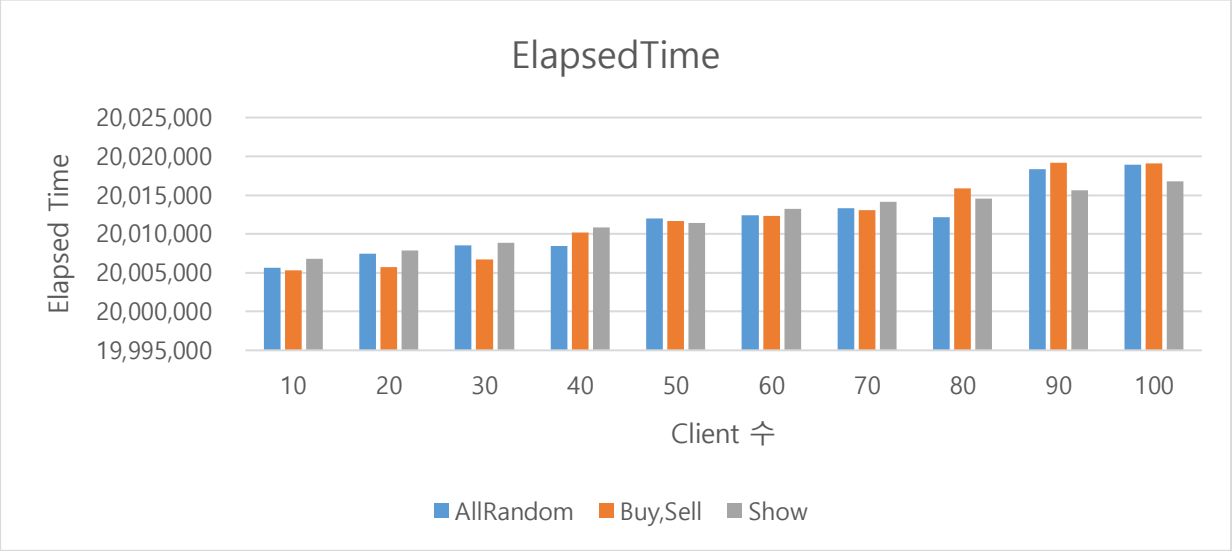
우선, 2번 Metric을 실험할 때 사용한 Configuration 은 다음과 같다.

```
#define MAX_CLIENT 100
#define ORDER_PER_CLIENT 50
#define STOCK_NUM 10
#define BUY_SELL_MAX 10
```

각 워크로드 별 Elapsed Time의 변화를 더욱 명확하게 확인하고자, ORDER_PER_CLIENT를 50으로 증가시켰다. client수는 위와 같이 10명부터 10 단위로

100까지 증가시키면서 실행하였다. 2번 Metric은 워크로드 유형에 따른 동시 처리율 변화를 확인하는 것이 목표이기 때문에, 중간에 block 되는 client가 생기지 않도록 NTRHEAD의 값도 client의 최댓값인 100으로 설정하였다.

모든 워크로드 유형 별 ElapsedTime의 막대 그래프를 그린 결과는 다음과 같다.



client의 수에 상관없이 Show 명령어의 ElapsedTime이 가장 작을 것이라고 예상했던 것과 달리, 오히려 client의 수가 적을 때에는 show 명령어의 ElapsedTime 이 가장 길었다. 왜 이런 결과가 나온 것인지 고민 해보았는데, 우선 stock tree의 크기가 작고, 유의미한 차이를 내기 위한 client의 수가 적어서 그런 것 같다는 결론을 내렸다.

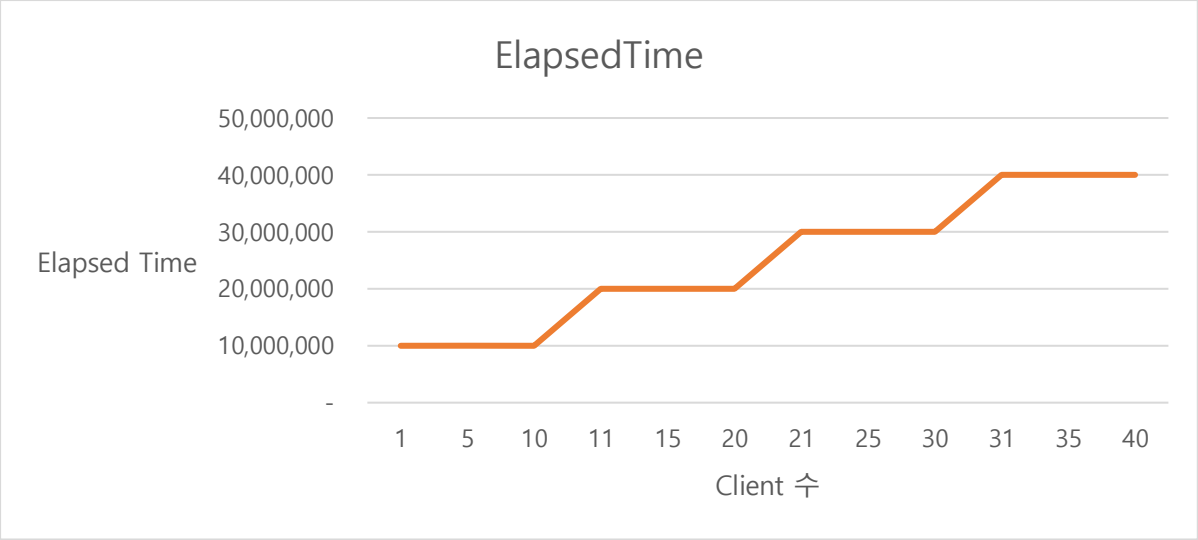
buy, sell 명령어는 write를 수행하기 때문에 n개의 thread가 동시에 똑같은 node에 접근할 수 없다. 이와 반대로 show 명령어는 모든 node에 접근해야 하긴 하지만, n 개의 reading thread가 concurrent 하게 하나의 노드에 접근할 수 있다. client의 수가 적을 때는 모든 thread들이 해당 노드에 concurrent하게 접근해 얻을 수 있는 이득보단, buy/sell 명령어처럼 특정 하나의 node에만 접근해 작업을 수행하는 것의 이득이 더 커서 결과가 위처럼 나온 것 같다. 하지만 그래프를 보면, client의 수가 많아질수록 buy/sell 명령어의 Elapsed time이 길어지고, Show 명령어의 Elapsed Time이 다른 종류의 워크로드 보다 작아진다는 것을 알 수 있다. 즉, Readers-writers Problem 을 해결했을 때의 성능차이는 concurrent하게 돌아가는 thread의 수가 많아질수록 커진다고 분석할 수 있다. 모든 명령어를 랜덤하게 사용하는 워크로드는 show 와 buy/sell 명령어의 비율에 따라 수행속도가 달라 수 있지만, client 의 수가 많아질수록 buy/sell 보단 Elapsed Time이 짧고, show 명령어보단 Elapsed Time이 길어질 것이라고 분석할 수 있다.

- **3번 Metric**

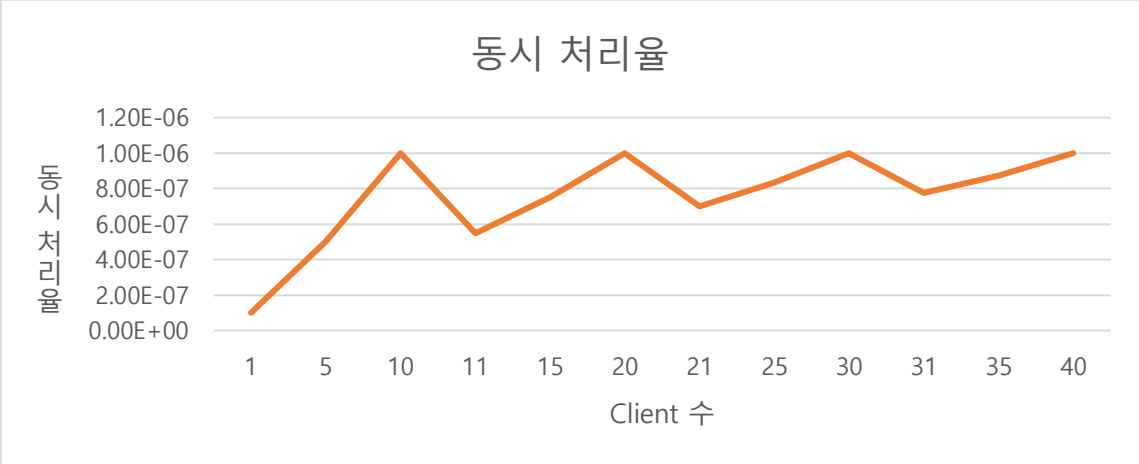
- *Thread-Based Server에서 Client 수에 따른 동시 처리율 변화*

Thread-Based Server의 장점은 client의 수가 Thread의 개수보다 적을 때 concurrent하게 동작해 동시 처리율이 높다는 것이다. 하지만 Client의 수가 NTRHEAD 보다 많아지면 초과하는 client들은 다른 Thread가 종료될 때까지 대기해야 하기 때문에, 동시 처리율이 낮아진다. 이를 확인하기 위해 NTRHEAD의 값을 10으로 설정하고, client의 수를 1, 5, 10, 11, 15, 20, 21, 25, 30, 31, 35, 40 으로 변경해가며 Elapsed Time을 계산해보았다. client의 Configuration은 위와 동일하고, Client 측에선 모든 명령어를 Random하게 사용하였다. 결과는 다음과 같다.

1. Elapsed Time



2. 동시 처리율



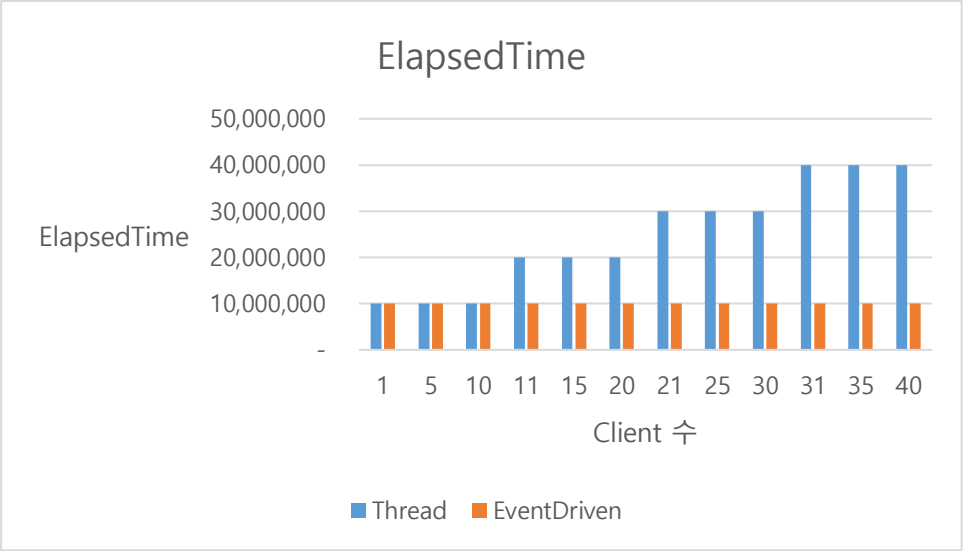
ElapsedTime을 보면 client의 수가 NTHREAD인 11이 되는 순간, 2배가 된다. 그리고 21이 되는 순간 3배가 되고, 31이 되는 순간 4배가 된다. Client의 수가 1~10, 11~20, 21~30, 31~40 일 때의 각 구간별 ElapsedTimed은 거의 동일하다. 이는 Thread-Based Server에서 최대로 처리할 수 있는 client의 수는 Thread의 최대 개수인 NTRHEAD 이기 때문이다. client의 수가 11, 21, 31 이 됐을 때 Elapsed Time이 증가하는 이유도 초과되는 1명의 client들은 10개의 Thread중 한 개의 Thread가 종료될 때까지 기다려야 하기 때문이다.

동시 처리율 그래프는 위의 결과를 잘 나타내고 있다. Client의 수가 1~10 일 땐, client 수가 증가해도 Elapsed Time은 거의 똑같기 때문에 동시 처리율이 증가한다. 이 때 한 명의 client가 새로 들어오는 순간 동시 처리율이 뚝 떨어지는 것을 볼 수 있는데, 이는 NTRHEAD가 10이라 10 단위로 Client를 처리하기 때문이다. 따라서 Client의 수를 N 이라고 했을 때, N % 10 == 1 일 때 동시 처리율이 가장 낮아지고, N % 10 == 0일 때 동시 처리율이 가장 높아진다.

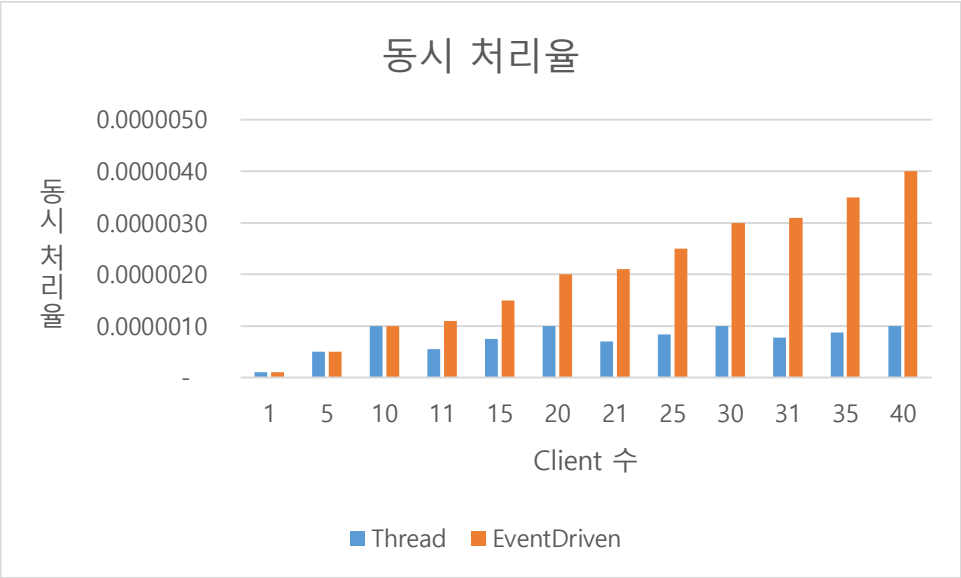
- **4번 Metric**

- *client수 변화에 따른 Event-Driven Server와 Thread-Based Server의 동시 처리율 비교*

3번 Metric에서 사용한 Thread-Based Server의 ElapsedTime을 사용하고, EventDriven-Server의 Client 수 1, 5, 10, 11, 15, 20, 21, 25, 30, 31, 35, 40 명일 때의 ElapsedTime을 측정해서 비교해보았다. 모든 명령어는 랜덤하게 사용하도록 설정하였고, NTRHEAD의 값은 10으로 세팅하였다. 결과는 다음과 같다.



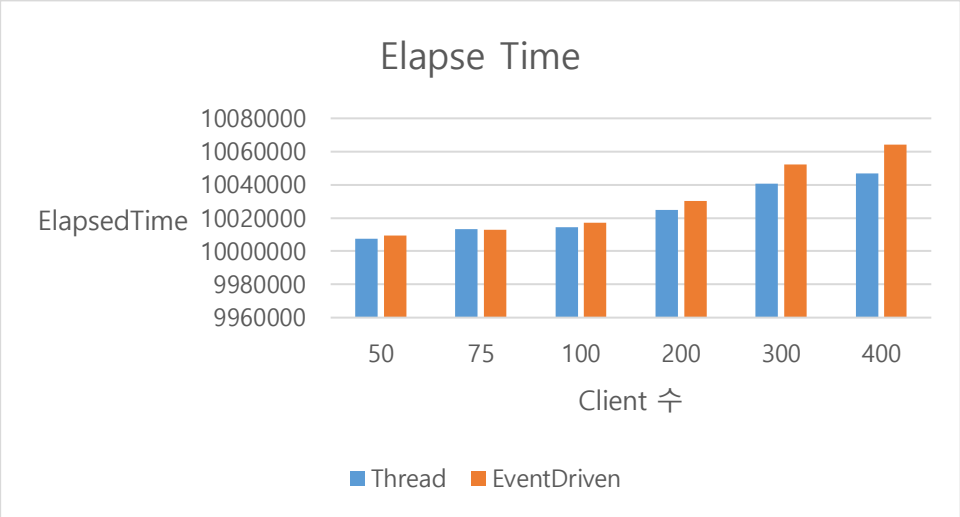
이 그래프를 보면, client 수가 증가함에 따른 ElapsedTime의 증가는 Thread-Based Server 가 EventDriven의 Server보다 훨씬 크다는 것을 알 수 있다. NTHREAD의 배수를 초과하는 Client수가 들어올 때마다, 나머지 Client는 Worker Thread가 종료될 때까지 계속 대기해야 하기 때문이다. 동시 처리율은 다음과 같다.



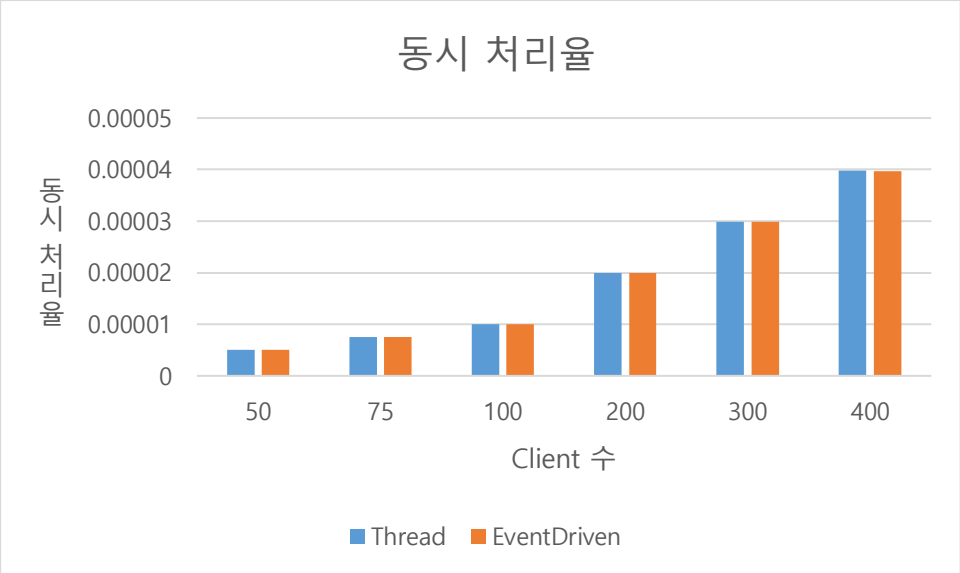
Thread-Based Server의 동시 처리율은 3번 Metric에서 살펴본 것처럼 NTREHAD 단위로 증가, 감소를 반복한다. 이는 Thread-Based Server의 단점을 보여준다. 정해 놓은 Thread 의 수보다 많은 수의 Client가 접속하면, 그만큼 동시 처리율이 낮아지는 것이다. 이와 반대로 Event-Driven Server는 하나의 thread에서 I/O Multiplexing을 기반으로 처리하기 때문에, 동시 처리율이 높아질 수 있다.

위와 같은 결과는 NTRHEAD의 값이 작아서 생긴 문제이다. NTREHAD 값에 따른 결과 비교를 위해, 10이었던 NTHREAD의 값을 400으로 늘리고, Client의 수를 50, 75, 100, 200, 300, 400 으로 변경해가며 ElapsedTime 및 동시 처리율을 측정해보았다.

1 . Elapsed Time



2. 동시 처리율



NTHREAD의 값을 400으로 설정하니, NTHREAD의 값이 10일 때와는 완전히 다른 결과가 나왔다. ElapsedTime도 EventDriven Server가 더 걸렸고, 이에 따라 동시 처리율도 미약하긴 하지만 Thread-Based Server가 더 좋았다. Thread-Based Server는 Client의 수가 NTRHEAD의 값보다 작게 들어오면, 실제로 Concurrent하게 Scheduling해서 Client의 요청을 처리할 수 있기 때문에, Event-Driven Server

보다 성능이 더 좋을 수 있다. 즉, 성능은 NTHREAD의 값을 얼마나 크게 할 수 있는지에 따라 달려있다고 분석할 수 있다.

하지만 THREAD의 값을 무한정 늘릴 수는 없을 것이다. Process에 비해 Context switch의 overhead가 작다고 하더라도, Thread의 수를 늘릴 수록 Overhead가 더욱 계속해서 커지기 때문이다. 따라서 Client의 수를 파악하고, pre thread의 수를 적당히 설정해야 최적의 성능을 낼 수 있을 것이다.