System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 이재진

학번 : 20181671

1. 개발 목표

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

해당 프로젝트에서 구현하는 내용은 보유하고 있는 주식의 상태를 열람하거나(show), 주식 장에서 여러 명의 client들이 주식을 사고 파는 작업(buy, sell)을 수행하는 동시적인 주식 서버이다. 여러 명의 사람들이 주식을 사고 팔거나 열람하는 행위는 실시간으로 이루어져야 하기 때문에 이와 같은 과정이 동시적으로 일어나야 한다. 따라서 이런 동시성을 보장하기 위해 이번 프로젝트에서는 select()를 이용한 Event-based 동시서버와 pthread()를 이용한 Thread-based 동시서버 두 가지를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   server가 각각의 client에 대한 connected descriptor를 pool에 저장하면, select()를 통해 준비된 connection들에 대해 client가 보내는 show, buy, sell을 수행하고 그 결과값을 client로 다시 보내준다. (결과값을 보내준다는 점에서 단순한 concurrent echo server와 다르다.)

2. Task 2: Thread-based Approach

   server가 worker thread 여러 개를 생성하고 반복문을 돌면서 connected descriptor들을 sbuf라는 버퍼에 저장한다. 또한 worker thread는 sbuf에 만약 connected descriptor가 있다면 하나를 빼 와서 show, buy, sell을 수행하고 그 결과값을 connected descriptor를 통해 client로 다시 보내준다. 단, 주의해야 될 점은 sbuf에서 connfd를 add, remove하는 과정과 show, buy, sell하는 과정에서 Readers-Writers 문제가 발생할 수 있는데, 이는 mutex를 이용하여 해결해야 할 것이다.

3. Task 3: Performance Evaluation

Task1, Task2에서 제작한 Event-driven 동시서버와 Thread-based 동시서버를 동시처리율을 기준으로 한 여러 가지 항목들에 대해 그 성능을 검사하고 수업 시간에 배웠던 내용과 일치하는지, 일치하지 않는지 직접 확인해본다.

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

Event-driven 동시서버에서는 select()를 이용하여 I/O Multiplexing 기능을 구현하였다. select() 는 다음의 두 가지 이벤트가 들어오기 전까지는 (fd_set에 다음 두 가지의 경우에 대한 file descriptor의 index가 set 되지 않으면) kernel에게 현재 프로세스를 suspend한다.

① listenfd를 통해 client로부터 새로운 connection이 들어왔을 때

② connected descriptor가 show, buy, sell, echo 등의 작업을 수행할 준비가 되었을 때 (더 정확히는, ready set의 해당 connected descriptor에 대한 index가 set 되었을 때)

1번을 통해 server와 client 간의 연결을 Accept 함수로 형성하고, 2번을 통해 연결된 client의 요청을 server가 받고 그 결과값을 Rio_writen()을 통해 connected descriptor, 즉 client에 쓰게 된다.

✓ epoll과의 차이점 서술

select()는 매 반복문마다 fd_set의 모든 내용을 계속 순회하면서 FD_ISSET() 함수를 통해 어떤 특정한 I/O 이벤트가 어떤 file descriptor에서 일어났는지 확인하기 때문에 비효율적인 면이 있다. 반면 epoll() 같은 경우, 커널에 해당 file descriptor에 대한 정보를 단 한번만 알려준다. 또한 select() 처럼 전체 file descriptor들의 정보를 모두 다 확인할 필요 없이, 변화가 일어난 부분만 확인하여 알려준다. 그러나 select()와 달리 epoll()은 리눅스에서만 사용할 수 있는 함수이다.

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

  ✓ Master Thread의 Connection 관리

  구현한 내용에서 Master Thread는 main thread에 해당한다. main thread에서는 N_THREADS 개수만큼의 worker thread을 pthread_create()를 통해 생성하고, 반복문 (while(1)) 에 들어가게 된다. Main thread가 Connection을 관리하는 방법은 반복문에서 client로부터 새롭게 들어온 connection request를 Accept()를 통해 받아들이고, connection을 형성하는 데 성공했을 때 Accept()가 리턴하는 connected file descriptor를 sbuf라는 버퍼에 sbuf_insert()를 통해 추가한다. 또한 바로 후술할 내용이지만, main thread에 의해 처음에 생성된 각 worker thread들은 sbuf로부터 한 개의 connected file descriptor를 빼서 쓰게 되는데, 이 과정에서 sbuf_remove()가 실행된다. 여기서 sbuf_insert()와 sbuf_remove()함수의 실행 순서가 올바르지 않을 수 있는 Readers-Writers 문제가 발생하게 된다. 이 부분은 각각의 sbuf 함수 내에 구현된 counting semaphore들인 sp->slots, sp->items 에 의해 관리된다. P(sp->slots)는 sp->slots의 값이 0일 때, 즉 Producers-Consumers 문제에서 Produce 할 수 있는 남은 slot의 개수가 0이 된 것이다. 이 때 해당 쓰레드는, 먼저 P(sp->slots) 를 실행하여 lock을 가지고 있는 쓰레드의 critical section에서의 작업이 끝나서 V(sp->slots)를 실행해줄 때까지 suspend 된다. 마찬가지로 Consume 할 수 있는 남은 item의 개수가 0이 된 경우, 해당 쓰레드는, 먼저 P(sp->items) 를 실행하여 lock을 가지고 있는 쓰레드의 critical section에서의 작업이 끝나서 V(sp->items)를 실행해줄 때까지 suspend 된다. 이런 식으로 Connection을 관리하여 sbuf에서 발생하는 Readers-Writers 문제를 해결할 수 있다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  우선 main thread와 worker thread pool 간에 발생하는 sbuf에서의 Readers-Writers 문제에 대한 해결책은 앞에서 서술하였다. 발생할 수 있는 다른 문제를 보면 이것도 Readers-Writers 문제이다. 저장된 주식의 정보를 어떤 쓰레드가 읽고 있는 와중에 다른 worker 쓰레드가 새로운 정보를 업데이트(쓰기)해서는 안 되고, 반대로 어떤 쓰레드가 주식 정보에 새로운 정보를 업데이트하고 있는 중간에 다 업데이트가 되지도 않았는데 다른 worker 쓰레드가 주식 정보를 읽어서는 안 된다. 이것은 이진 트리 형식으로 정의된 items 구조체에서 각각의 트리 노드에 대한 mutex 값 (더 정확히는 mutex, mutex_w) 을 제공함으로써 해결한다. 맨

처음에는 mutex 값들을 전부 1로 초기화하고, 만약에 해당 노드(해당 주식)의 mutex_w 값이 0이 되어 있는 상황에서 새로운 쓰레드가 쓰기 작업을 하려고 하거나 mutex 값이 0이 되어 있는 상황에서 새로운 쓰레드가 읽기 작업을 하려고 한다면 각각 P(&mutex_w), P(&mutex)를 실행하여 해당 노드의 resource에 대한 lock을 얻으려고 시도할 것이고, 이 쓰레드는 이미 읽기 작업을 하고 있거나 쓰기 작업을 하고 있는 쓰레드가 V(&mutex_w), V(&mutex)를 해 줄때까지 suspend 되어 기다리게 된다. 이런 식으로 worker thread들 간에 발생하는 Readers-Writers 문제를 해결할 수 있다.

- **Task3 (Performance Evaluation)**

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

얻고자 하는 metric은 과제 설명문에서 제시한 '동시 처리율'이다. 이렇게 정한 이유는 동시 서버가 얼마나 동시적으로 요청을 잘 처리하는지가 각 서버의 성능을 평가하는 중요한 지표가 되기 때문이다. 이 metric을 기준으로 두 가지 서버 프로그램을 여러 가지 기준을 두고 평가하여 각각의 성능과 특징을 분석하려고 한다. 과제 설명문에서 제시한 '동시 처리율'의 정의는 '시간(초)당 client 처리 요청 개수' 인데 이것을 수식으로 표현하면

$$\frac{\text{num(Clients) x num(Requests per Client)}}{\text{Total Elapsed time (s)}}$$ 이다.

미리 제시된 multiclient.c 파일의 설정 들 중 ORDERS_PER_CLIENT와 프로그램을 실행할 때의 인자로 주어지는 Client 수를 변경해가면서 해당 경우에 대한 총 소요 시간(Total Elapsed Time in seconds)를 clock_gettime() 함수를 통해 측정한다.

또한 과제 설명문에서 제시하는 '분석 포인트'에 맞추어 '확장성', '워크로드에 따른 분석'을 진행하려고 한다.

✓ Configuration 변화에 따른 예상 결과 서술

① 기본적인 예측 : 우선 client의 개수나 client당 보내는 요청 수를 증가시킬수록 총 소요 시간이 선형적으로 증가할 것이라고 예상한다.

② '확장성' 항목에 대한 예측 :

Event-driven server의 경우 단일 쓰레드이고 강의에서 들었던 대로라면 thread based server에서는 thread context switching 과 세마포어 변수의 사용에 의해 medium overhead를 가지는 것에 비해 아주 적은 overhead를 가지기 때문에 대부분의 경우에서 같은 조건 하에 총 소요 시간이 더 적을 것이라 예상했다. (성능이 더 좋을 것이다.) 그러나 event-driven server는 멀티코어 CPU를 활용하지 못하고, 매 이벤트가 발생할 때마다 select 함수는 fd_set 전체를 순회하므로 thread based server의 총 소요시간이 더 빠른 경우도 분명히 존재할 것이다.

③ '워크로드에 따른 분석' 항목에 대한 예측 :

1) client가 buy 또는 sell만 요청할 때

2) client가 show만 요청할 때

3) client가 buy만 요청할 때

4) client가 sell만 요청할 때

5) client가 buy 또는 sell 또는 show를 요청할 때

예측하기로 빠른 순서대로 3) ≒ 4) ≥ 1) > 2) > 5)일 것이다. 왜냐하면 sell, buy는 이진 트리 자료구조를 해당 id를 찾을 때까지만 순회하면 되는데 반해 show의 경우는 항상 이진 트리 자료구조를 전부 순회해야 해서 sell, buy와 비교해서 어느 정도 더 시간이 걸릴 것이라 생각한다.

C. 개발 방법

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

① Event-driven 동시서버

1) (struct) item 구조체 : 기본적으로 이진트리 구조체이며, 현재 노드의 각각 왼쪽과 오른쪽을 나타내는 item* left, item* right와 해당 주식의 id, 주식 잔고, 주식 가격에 대한 정보를 int형으로 저장하도록 했다. 정확한 구조는 아래와 같다.

```
typedef struct item{
    int id;
    int left_stock;
    int price;
    struct item* left;
    struct item* right;
}item;
```

2) void handler(int sig) : 서버에 대한 Client들의 Connection이 더 이상 없을 때 서버를 종료하기 위해 CTRL+C를 누르게 되면 시그널 핸들러가 SIGINT 시그널을 catch하면서 reset_tree()를 호출하여 트리 구조에 대한 메모리를 해제하고 서버가 종료된다.

3) (struct) pool 구조체 : active client들의 connected file descriptor들과 그 외 file descriptor들에서 I/O 이벤트가 일어났는지 확인하기 위해서 fd_set에 대한 정보를 가지고 있는 구조체이다. select() 에 의해 그 값이 변경되는 ready_set 값을 매 루프마다 백업하는 read_set과 client들의 connfd들을 저장하는 int 배열 clientfd, 각 client들의 Rio buffer를 저장하는 rio_t 배열 clientrio, 그리고 현재 ready 상태에 있는 file descriptor들의 개수를 나타내는 int 변수 nready와 현재 배열의 크기값을 저장하는 maxi, 가장 큰 fd값을 저장하는 maxfd 변수로 구성되어 있다. 정확한 구조는 아래와 같다.

```
typedef struct {
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready;
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
}pool;
```

3) void init_tree() : 프로그램이 시작되면 stock.txt 파일의 내용을 읽어와 메모리에 이진트리 형태로 로드한다. 이진트리로 로드하는 과정은 inorder binary tree traversal 방식을 이용하여 각 주식 정보를 fscanf로 읽어오고 해당 노드에 읽어온 정보를 저장하는 식으로 수행하였다. recursive한 방식이 아닌 iterative한 방식으로 inorder traversal을 수행하였다.

4) void store_tree() : 어떤 client에 대한 connected file descriptor가 Close 될 때마다 파일에 쓰기 작업을 수행한다. 이 경우에도 역시 inorder binary tree traversal을 하면서 해당 노드의 정보를 파일에 알맞은 형식으로 fprintf로 쓰게 된다. 이 경우에는 recursive한 방식으로 inorder traversal을 수행하였다.

5) void buy(item *items, int id, int num_to_buy) : 파라미터로 받은 id에 해당하는 주식을 num_to_buy개만큼 사는 작업을 수행하는 함수이다. 주의할 것은 item *items는 전역 변수라는 사실이다. (이후의 함수 설명에서도 동일한 조건이 적용된다.) '산다'는 것은 이진트리 자료구조에서 해당 주식의 left_stock 변수를 left_stock에서 num_to_buy개만큼 뺀 값으로 변경하는 작업을 의미한다. 이 경우에는 inorder binary tree traversal을 하면서 파라미터로 받은 id에 해당하는 노드를 찾는다. 그 다음 해당 노드의 left_stock과 num_to_buy의 값을 비교해서 만약 전자가 더 크다면 값을 변경하고, buy가 성공했다는 메시지를 전역변수 buf_temp에 작성한다. 반대로 후자가 더 크다면 주식을 살 수가 없으므로 주식 잔고가 부족하다는 메시지를 buf_temp에 저장하고 리턴한다.

6) void sell(item *items, int id, int num_to_sell) : 파라미터로 받은 id에 해당하는 주식을 num_to_sell개만큼 파는 작업을 수행하는 함수이다. '판다'는 것은 이진 트리 자료구조에서 해당 주식의 left_stock 변수를 left_stock에 num_to_sell만큼 더한 값으로 변경하는 작업을 의미한다. 이 경우에도 5)번과 유사하게 inorder binary tree traversal을 하면서 파라미터로 받은 id에 해당하는 노드를 찾는데, sell의 경우 항상 성공한다는 가정이 있으므로 left_stock의 값을 변경하고 성공 메시지를 buf_temp에 작성하고 리턴한다.

7) void show(item *items) : inorder traversal을 recursive하게 수행하면서 이진트리의 모든 노드들에 대한 정보를 sprintf로 buf_temp에 작성하고 리턴한다.

8) void reset_tree(item *items) : server가 모종의 이유로 종료될 때 메모리를 free해주기 위한 목적으로 이 함수를 작성하였다. 역시 inorder traversal을 recursive하게 수행하면서 모든 노드들에 대한 메모리를 free한다.

9) void service_clients(pool *p) : 구현에서 가장 중요한 함수이다. ready 상태인 모든 connected file descriptor에 대해 반복문을 돌면서 각 client에서 보내는 메시지 (show, buy, sell, exit 중 하나) 를 Rio_readlineb()로 읽어들인다. 그런 다음 버퍼 buf에 들어온 입력이 무엇인지 strcmp로 비교하여 앞에서 언급한 show, sell, buy 함수를 수행하고 그에 따라 변경된 buf_temp의 값을 버퍼 buf에 strcat으로 붙이고 Rio_writen으로 client의 connected file descriptor에 buf의 내용을 쓴다. 이 때 주의해야 할 것은 client에 buf의 내용을 쓴 다음 buf_temp가 전역변수이기 때문에 값을 비워주어야 한다는 것이다. 어떤 connected file descriptor가 close되면 stock.txt에 변경 내용을 write 해 준다.

② Thread-based 동시서버

1) (struct) item 구조체 : 앞에서 언급한 Event-driven 동시서버에서의 item 구조체에 Reader를 위한 semaphore 변수인 mutex, writer를 위한 semaphore 변수인 mutex_w, reader가 critical section에 몇 명 들어가 있는지에 대한 int 변수인

readcnt가 추가되었다. 자세한 구조는 아래와 같다.

```
typedef struct item {
    int id;
    int left_stock;
    int price;
    int readcnt;
    struct item* left;
    struct item* right;
    sem_t mutex; //lock for reading
    sem_t mutex_w; //lock for writing
}item;
```

2) (struct) sbuf_t 구조체 : main thread에서 client의 connected file descriptor를 '생산'하고, worker thread에서 client의 connected file descriptor를 '가져오고' 싶을 때 (Producer- Consumer 개념) 쓰이는 버퍼 배열과 그 배열의 정보를 담고 있는 구조체이다. 앞에서 언급했듯이 Producer-Consumer 문제를 해결하기 위해 sbuf_t 의 함수 sbuf_insert(), sbuf_remove()에서는 counting semaphore인 sem_t slots, sem_t items를 도입하고 있다. 구체적인 구조는 아래와 같다.

```
typedef struct {
    int *buf; //buffer array
    int n; //maximum slots (buffer size)
    int front; //buf[(front+1)%n] is the first item
    int rear; //buf[rear%n] is the last item
    sem_t mutex; //protects accesses to buf
    sem_t slots; //available slots to be produced
    sem_t items; //available items to be consumed
}sbuf_t;
```

3) init_tree(), store_tree(), handler(), reset_tree()의 내용은 Event-driven 동시서버와 동일하다.

4) void show(), void show_read(item* items) :

show()는 show_read()를 수행하기 위한 껍데기 함수이기 때문에 실질적인 내용은 show_read()에 있다. show_read()는 일반적으로 Readers-Writers 문제를 해결하기 위한 코드 구조를 채택하였다. 우선 mutex는 이진트리에서의 읽기 권한에 대한 lock이고, 어떤 worker thread가 처음으로 reader가 되는 순간 items->mutex의 값은 0이 되고 다른 worker thread는 이 thread가 V(&(items->mutex))를 해줄 때까지 기다려야 한다. 현재 lock을 가지고 있는 thread는 주식 정보를 읽고 있는 worker thread가 단 한 명이라도 존재할 경우 Writer가 데이터에 접근하지 못하게 하기 위해 이진트리에서의 쓰기 권한의 lock인 mutex_w에 대한 P 함수를 실행시킨다. 이렇게 하면 다른 쓰레드가 sell이나 buy를 할 때 mutex_w 값은 다른 reader들의 읽기 작업이 완전히 끝나지 않는 이상 0이므로 쓰기를 할 수가 없게 된다. 그 외의 show()의 구현은 앞에서의 Event-driven 동시서버 방식과 완전히 유사하다. (이진트리를 inorder traversal 하면서 buf_temp에 그 내용을 저장하는 방식)

5) void buy(int id, int num_to_buy), void buy_read(item *items, int id, int num_to_buy), void sell(int id, int num_to_sell), void sell_read(item *items, int id, int num_to_sell) :

buy_read()에서는 앞에서 show_read()를 구현한 방식과 유사한 방식으로 구현하였고, buy()에서는 inorder traversal을 통해 찾은 노드가 buy_read()에 의해 return 되면 그 때 write가 가능한지 P(&(items->mutex_w))를 통해 확인하고 아니라면 해당 쓰레드는 suspend되고, 맞다면 앞에서의 Event-driven 동시서버 방식과 유사하게 buy 명령어의 내용을 수행한다. (데이터를 수정(쓰기)한다.)

sell_read()와 sell() 같은 경우도 앞과 동일하게 진행된다.

6) void init_thread() :

맨 처음에 전역변수 mutex의 값을 Sem_init을 통해 1에 해당하는 값으로 초기화해주는 함수이다. Pthread_once() 함수의 파라미터로 사용된다.

7) void *thread(void *vargp) :

쓰레드 함수로, 구현에 가장 중요한 부분이다. 처음에 sbuf_remove() 함수를 통해 main thread에서 추가한 connected file descriptor들 중 하나를 sbuf에서 제거하고 기본적으로 Event-driven 동시서버에서 설명했던 것과 유사한 방식으로 명령어를 받고 (다만 buf_temp, choice를 비우는 작업이 쓰레드별로 순차적으로 시행되도록 mutex 전역변수값을 따로 하나 설정하였다. 이 값은 앞에서 설명한 init_thread() 함수에서 맨 처음에 한번 1로 초기화된다.) connfd에 buf와 buf_temp를 붙인 데이터를 Rio_writen()으로 작성한다. 그런 다음 만약 sbuf가 비었다면 (front == rear) stock.txt 파일에 write한다. 이것이 connection이 모두 끝났다는 얘기는 아니지만, connection이 모두 끝났을 때는 sbuf가 비었다는 것이 보장되므로 마지막 이진 트리의 상태를 무조건 저장하게 되어 있다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

1) 구현하는 과정에서 server에서 client로 보내는 데이터를 읽어들이는 Rio_readnb() 가 어떻게 그 데이터의 바이트수를 얼마만큼 읽을지 정확히 알고 파라미터로 넘겨주어야 할지 고민하게 되었다. 읽어들이는 바이트수를 제대로 지정하지 못해 Rio_readnb error : Connection reset by peer 에러가 생기거나 deadlock 현상이 일어나기도 했다. 이에 대한 해결책은 server에서도 MAXLINE만큼을 write하도록 하고, client측 readnb에서도 MAXLINE만큼 읽어들이게 하는 것이었다.

2) 모든 connection이 끝나고 파일에 언제 이진트리의 내용을 작성해야 하는지에 대해서도 고민을 많이 했다. 결국은 event_based의 경우에는 service_clients() 함수에서 한 fd에 대한 connection을 close 할 때마다 작성하도록 하였고, thread_based의 경우에는 sbuf가 비었을 때마다 작성하도록 하였다. 오버헤드는 좀 생기겠지만 파일에 이진트리의 구조가 올바르게 업데이트되는 건 보장할 수 있게 제작하였다.

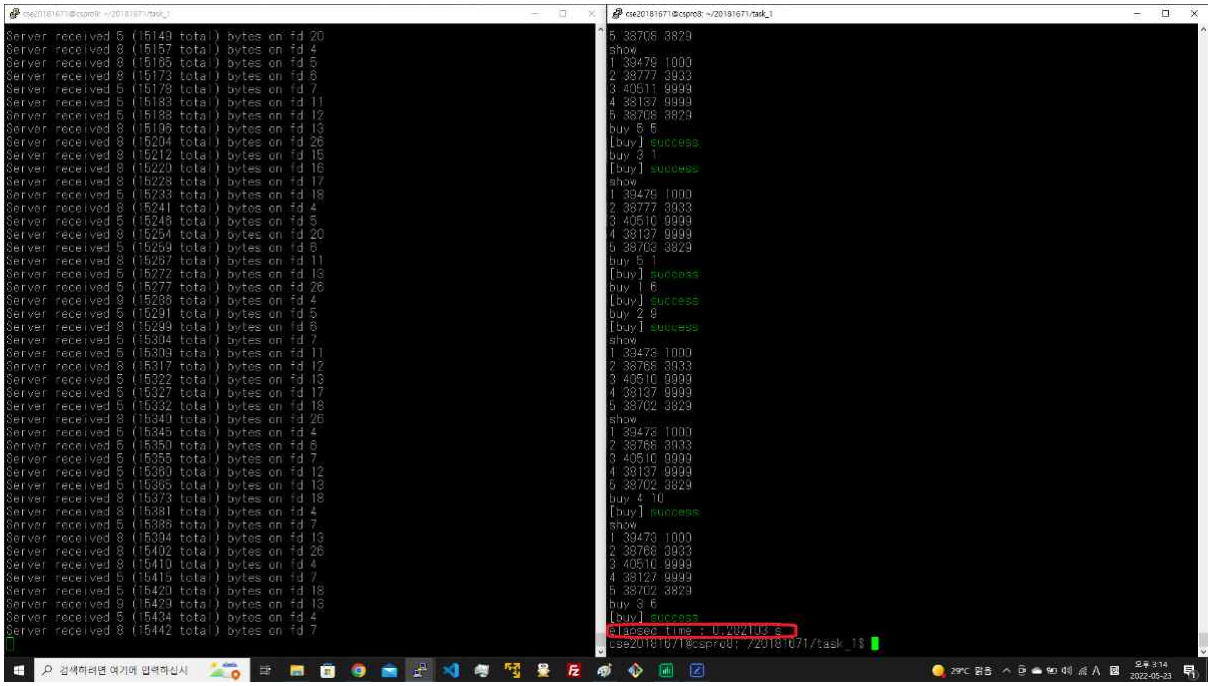
3) Thread-based의 경우 show, buy, sell 요청을 수행할 때 Readers-Writers 문제가 발생해서는 안 되므로, 앞에서 언급했던 approach로 이 문제를 해결하였다.

4. 성능 평가 결과 (Task 3)

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

우선 **B. 개발 내용**의 Task3 부분에 명시한 것과 같이 clock()을 사용하여 elapsed time을 측정하였고, 예시로 cspro에서 multiclient.c를 실행한 후의 output은 아래와 같다.

아래에서 측정한 모든 값들은 변동폭이 어느 정도 있어 신뢰도를 보장하기 위해 각 값마다 10번씩 측정하고 그 평균값을 기록하였다.



예시1. ORDER_PER_CLIENT = 10, client 수 = 100일 때 elapsed time 출력

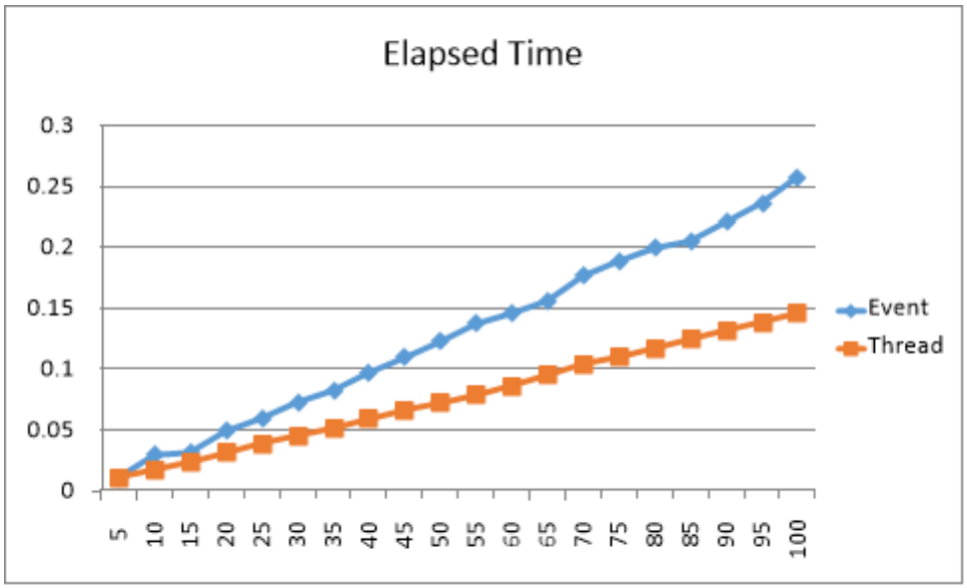
우선 본격적으로 Event-driven 서버와 Thread-based 서버의 성능을 비교하기 전에, **B. 개발 내용**의 Task3 부분에서 '기본적인 예측'에 대한 검증을 해 보도록 하겠다. Client당 보내는 request의 수를 10으로 고정하고 Client의 수에 변화를 주면서 측정한 총 소요 시간 및 동시처리율의 표와 그래프는 각각 아래와 같다.

(동시처리율 = $\frac{\text{num(Clients) x num(Requests per Client)}}{\text{Total Elapsed time (s)}}$ 으로 정의하였다.)

	5	10	15	20	25	30	35	40	45	50
Event	0.010908	0.029392	0.031473	0.048978	0.059532	0.072688	0.082331	0.09695	0.109184	0.122323
Thread	0.010762	0.017109	0.023235	0.030766	0.038805	0.045015	0.0510588	0.058507	0.065924	0.071688

55	60	65	70	75	80	85	90	95	100
0.13707	0.145674	0.155977	0.176458	0.188665	0.199277	0.205088	0.220701	0.23646	0.257603
0.078083	0.086208	0.094762	0.104045	0.109421	0.116845	0.124259	0.131582	0.138322	0.145663

표. 가로방향은 client의 수, 해당값은 총 소요 시간 (s) (Event-driven/Thread-based)

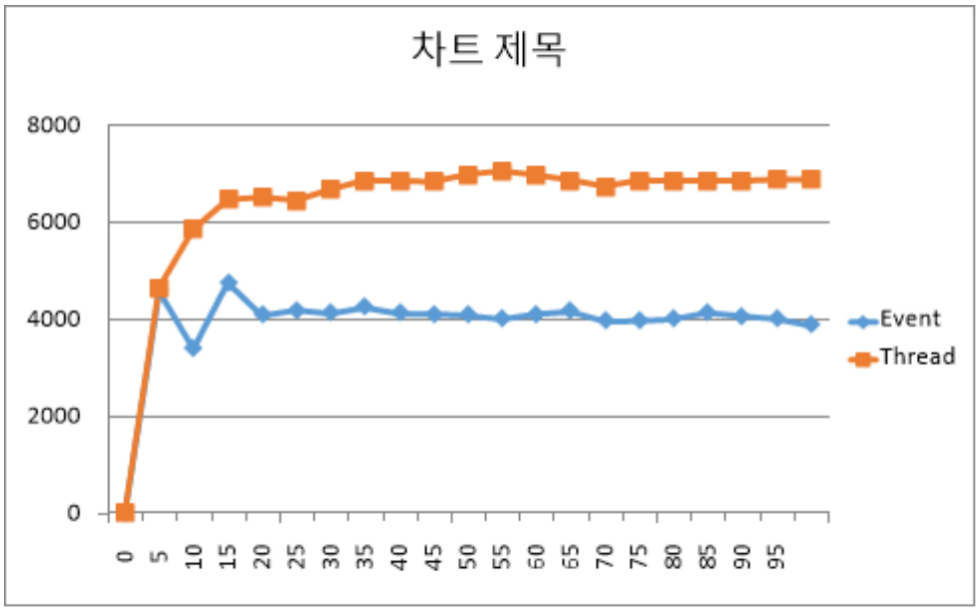


결과 그래프를 살펴보면 총 소요 시간은 Client의 수가 늘어남에 따라 (Client들이 보내는 총 request 수가 증가함에 따라) 선형에 가깝게 증가하고 있음을 알 수 있다. 이는 앞에서의 '기본적인 예측'과 부합한다.

	0	5	10	15	20	25	30	35	40	45
Event	0	4583.791713	3402.286336	4765.9899	4083.466	4199.4222	4127.2287	4251.1326	4125.8381	4121.483
Thread	0	4645.976584	5844.876965	6455.7779	6500.6826	6442.4688	6664.4452	6854.8419	6836.7888	6826.0421

50	55	60	65	70	75	80	85	90	95	100
4087.5387	4012.5483	4118.7858	4167.2811	3966.9496	3975.3001	4014.5125	4144.5623	4077.9154	4017.5928	3881.9424
6974.668	7043.7867	6959.9109	6859.2896	6727.8581	6854.2602	6846.6772	6840.5508	6839.8413	6868.0326	6865.1614

표. 가로방향은 client의 수, 해당값은 동시처리율 (Event-driven/Thread-based)



하지만 동시처리율의 경우는 일정한 'Client의 총 요청 수' 까지는 증가하다가 그 지점을 지나면 더 이상 크게 증가하지 않는 것을 볼 수 있었다. Thread-based와 Event-based 동시서버의 성능 비교와 관한 이야기는 뒤에서 다루도록 하겠다.

위와 같은 경우는 show, sell, buy를 client 측에서 모두 요청하는 경우이므로 뒤의 '워크로드에 따른 분석' 항목에서 또 인용하도록 하겠다.

※ 분석 항목

① 확장성 : 각 방법에 대한 Client 개수 변화에 따른 동시 처리율변화 분석

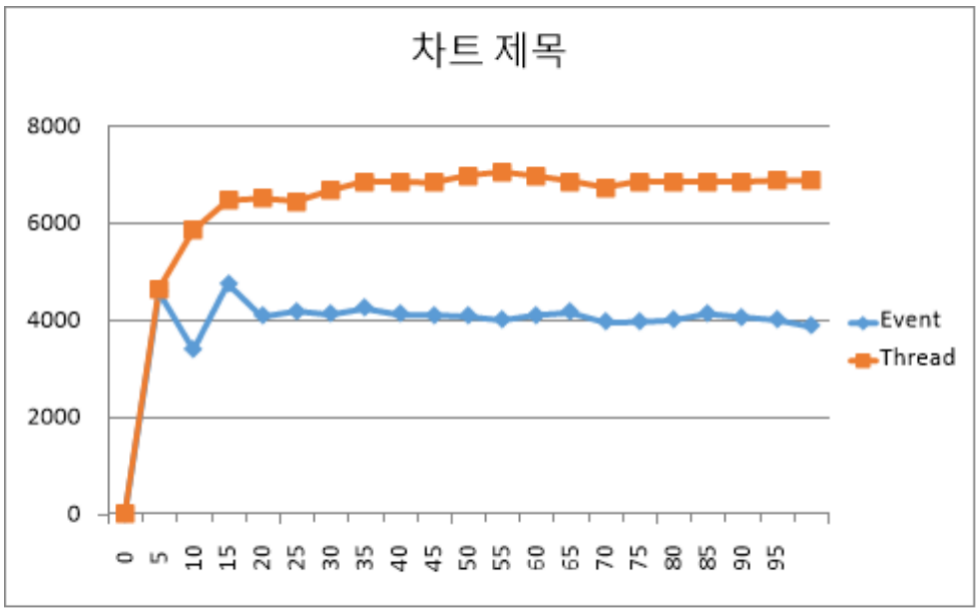
Client당 Request 개수를 10으로 고정시키고, Client 개수를 변화시키면서 측정한 동시 처리율 결과값은 아래와 같다. (바로 앞의 표와 그림과 동일하다.)

[Request 개수를 10으로 고정시켜도 Client 개수를 변화시키면 Client의 총 request의 개수는 늘일 수 있기 때문에 생각해낸 실험 조건이다.]

	0	5	10	15	20	25	30	35	40	45
Event	0	4583.791713	3402.286336	4765.9899	4083.466	4199.4222	4127.2287	4251.1326	4125.8381	4121.483
Thread	0	4645.976584	5844.876965	6455.7779	6500.6826	6442.4688	6664.4452	6854.8419	6836.7888	6826.0421

50	55	60	65	70	75	80	85	90	95	100
4087.5387	4012.5483	4118.7858	4167.2811	3966.9496	3975.3001	4014.5125	4144.5623	4077.9154	4017.5928	3881.9424
6974.668	7043.7867	6959.9109	6859.2896	6727.8581	6854.2602	6846.6772	6840.5508	6839.8413	6868.0326	6865.1614

표. 가로방향은 client의 수, 해당값은 동시처리율 (Event-driven/Thread-based)



결과값을 분석하면, 한눈에 봐도 대체로 동시처리율 측면에서 Thread-based 서버가 더 높은 값을 보임을 알 수 있다. client 수가 적을 때는 Elapsed time이 워낙 적은 수기 때문에 변동폭이 심해서 성능이 비슷하다고 볼 수 있는데, client 수가 늘어날수록 (요청하는 총 request 수가 증가할수록) Thread가 거의 2배 정도 월등한 성능을 보였다. 대체적

으로 Event-driven 서버가 오버헤드가 작아 더 좋은 동시처리율을 가지고, Thread-based 서버가 thread context-switching이나 세마포어의 사용에 의해 더 높은 오버헤드를 가져 Event-driven 서버가 더 월등한 성능을 보일 것이라 생각했는데, 생각보다 그런 요인들이 야기하는 오버헤드가 서버의 성능에 영향을 미칠 만큼은 아닌 것 같다. 또한 Event-based 동시서버에서 select()를 사용하기 때문에 하나의 이벤트가 발생할 때마다 fd_set 전체를 매번 다 순회하게 되는데, 이것도 성능의 지연에 영향을 주었다고 생각한다. 또한 Thread-based 서버의 경우 멀티코어를 지원하기 때문에 이와 같은 성능의 차이가 생겼다고 추측할 수 있다. 또한 이 Thread-based 서버의 경우 pre-threaded 방식으로 되어 있는데 (thread pooling 방식) 만약 thread를 그때그때 생성하는 서버가 된다면 그에 대한 오버헤드 때문에 예상했던 결과와 비슷했을 것이라 생각한다.

또한 두 번째로, 동시처리율이 client에서 보내는 총 요청 수가 늘어남에 따라 계속 증가하지 않음을 볼 수 있다. 이것은 thread-based 서버와 Event-driven 서버 모두에서 나타나는 현상이다. 어느 지점을 지나면 증가율이 점점 완만해지면서 0에 가까워진다. Event-driven 서버의 경우 어느 지점을 지나면 동시처리율이 조금씩 감소하는 것을 볼 수 있다.

② 워크로드에 따른 분석 : Client 요청 타입 (buy, show, sell 등)에 따른 동시 처리율 변화 분석

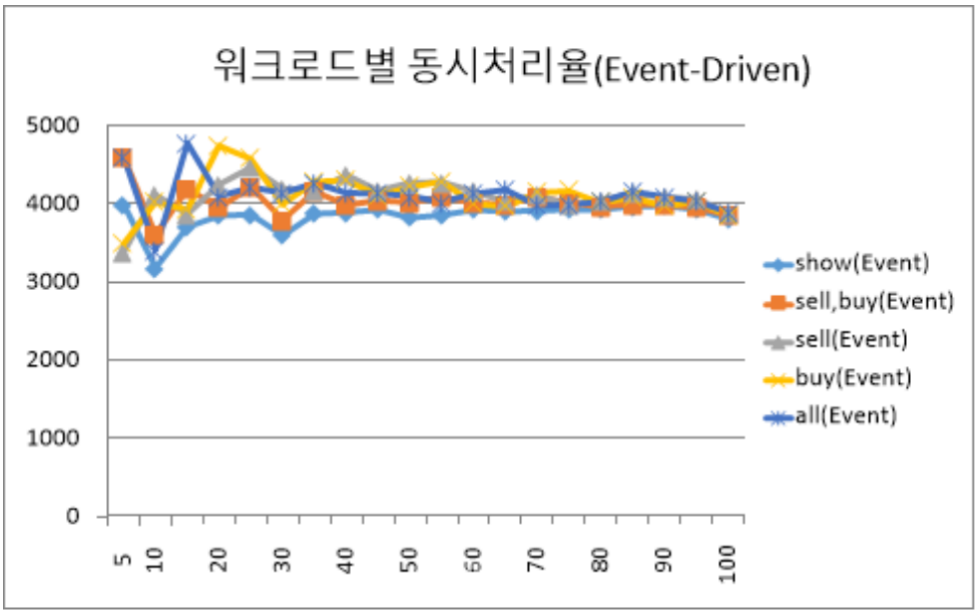
1) Event-Driven 동시서버의 워크로드별 동시처리율 분석

B. 개발 내용의 Task3 부분에 명시한 것과 같이 다섯 가지 항목 (buy+sell, show만, buy만, sell만, buy+sell+show[all]) 을 수행하고 동시처리율을 측정한 표와 그래프는 아래와 같다.

	5	10	15	20	25	30	35	40	45	50
show(Event)	3983.746315	3172.951225	3697.3133	3840.172	3858.501	3587.9159	3864.073	3883.8345	3916.9263	3814.4644
sell,buy(Event)	4587.155963	3590.793206	4176.6442	3937.2404	4201.6101	3781.1472	4145.1023	3969.3566	4030.2357	4002.3694
sell(Event)	3364.511137	4100.713524	3853.5645	4234.6863	4450.4575	4182.5254	4152.8239	4356.729	4153.8973	4258.0008
buy(Event)	3495.770118	4023.982938	3930.9206	4748.3381	4579.3417	4000.6935	4288.0071	4308.0237	4106.9636	4219.9435
all(Event)	4583.791713	3402.286336	4765.9899	4083.466	4199.4222	4127.2287	4251.1326	4125.8381	4121.483	4087.5387

55	60	65	70	75	80	85	90	95	100	평균
3848.226	3914.9925	3896.2985	3906.359	3920.5027	3917.9004	3960.8943	3966.9771	3920.7432	3801.8188	3833.73
4077.0337	3994.6206	3982.9164	4073.1775	3998.6351	3952.0224	3978.3205	3976.0026	3939.1627	3841.8687	4011.77
4266.0462	4145.1636	4014.2537	4104.0313	3966.9526	4035.94	4119.0952	4066.7122	4052.659	3863.0323	4087.09
4291.2762	4008.1767	3963.7772	4147.1897	4168.751	4015.5603	4087.3834	3972.7207	4034.9983	3856.2394	4112.4
4012.5483	4118.7858	4167.2811	3966.9496	3975.3001	4014.5125	4144.5623	4077.9154	4017.5928	3881.9424	4106.28

표. 가로방향은 client의 수, 해당값은 워크로드별 동시처리율 (Event-driven)



그래프를 분석해보면 우선 차이가 근소하지만 위 표의 평균값을 보면 대체로 buy ≥ all > sell > sell+buy > show 순으로 동시처리율이 낮아짐을 볼 수 있다. 앞에서의 예측과 완벽히 일치하지는 않지만, 이와 같은 결과는 show가 sell, buy보다 수행시간이 더 긴 것을 의미하는데, show는 모든 경우에 이진트리를 전부 다 순회해야 하고, sell, buy는 이진트리의 특정 노드를 찾으면 그 값까지만 순회해도 된다는 점에서 생각해보면 어느정도 결과가 예측에 부합한다고 볼 수 있다. 또한 결과로부터 알 수 있었던 점은 buy의 경우 한꺼번에 여러 개의 buy 요청을 보냈을 때 근소한 차이겠지만 주식 트리에서 해당 노드를 inorder search로 찾는 데 성공한 후에 만약 잔고 주식이 사려고 하는 주식보다 더 작으면 잔고 주식을 변경하는 연산을 하지 않고 buy 함수가 리턴하게 되는데, 이 때문에 요청들이 buy 명령어로만 이루어진 경우 sell이나 sell,buy만 있을 경우보다 더 동시처리율이 커질 수도 있겠다는 점이었다.

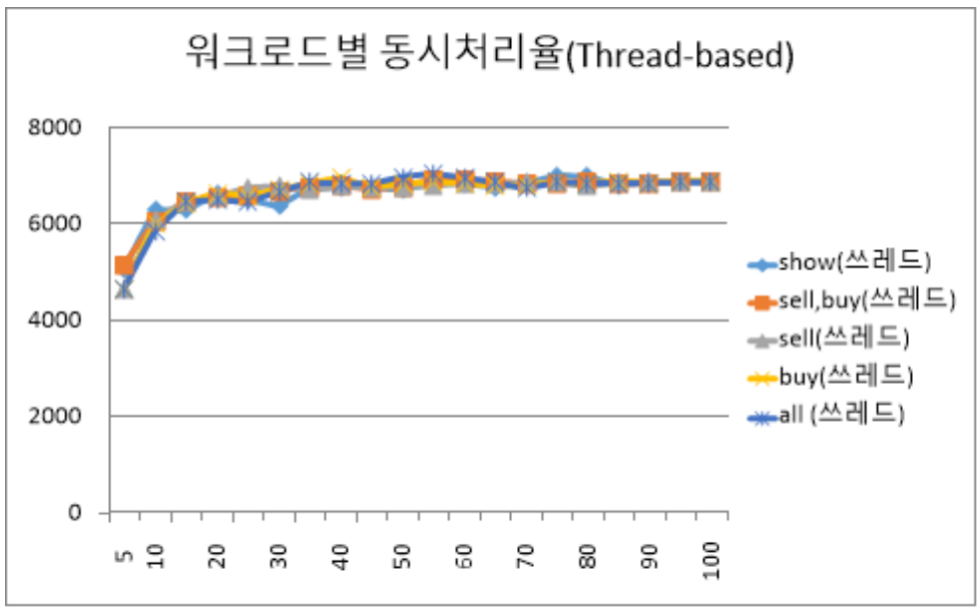
2) Thread-based 동시서버의 워크로드별 동시처리율 분석

B. 개발 내용의 Task3 부분에 명시한 것과 같이 다섯 가지 항목 (buy+sell, show만, buy만, sell만, buy+sell+show[all]) 을 수행하고 동시처리율을 측정한 표와 그래프는 아래와 같다.

	5	10	15	20	25	30	35	40	45	50
show(쓰레드)	5060.728745	6296.436217	6273.2634	6623.6132	6508.2134	6361.458	6740.8806	6765.3277	6756.6553	6703.8507
sell,buy(쓰레드)	5136.106831	6038.282712	6470.8166	6549.6463	6580.1595	6679.5805	6758.1919	6783.2251	6706.4083	6762.1482
sell(쓰레드)	4626.630887	6077.918921	6475.286	6567.2818	6752.1945	6789.7882	6696.0015	6770.8245	6766.7138	6762.4226
buy(쓰레드)	4629.62963	5996.282305	6415.1912	6640.3267	6569.439	6701.9637	6830.7344	6950.8402	6751.1815	6823.051
all (쓰레드)	4645.976584	5844.876965	6455.7779	6500.6826	6442.4688	6664.4452	6854.8419	6836.7888	6826.0421	6974.668

55	60	65	70	75	80	85	90	95	100	평균
6838.9247	6836.3603	6761.3956	6837.9408	7005.2212	6987.4488	6806.0983	6852.1832	6871.9076	6861.4872	6745.47
6911.0237	6899.724	6879.5445	6827.2035	6848.0643	6870.4322	6813.518	6838.0224	6870.9136	6876.4913	6654.98
6799.6934	6849.0805	6813.0601	6865.8415	6856.8294	6779.7759	6851.7444	6883.7337	6873.1009	6857.1585	6635.75
6858.1974	6835.7372	6772.8087	6787.4181	6854.8866	6821.8059	6888.7268	6841.6091	6879.7706	6889.5197	6636.96
7043.7867	6959.9109	6859.2896	6727.8581	6854.2602	6846.6772	6840.5508	6839.8413	6868.0326	6865.1614	6636.16

표. 가로방향은 client의 수, 해당값은 워크로드별 동시처리율 (Thread-based)



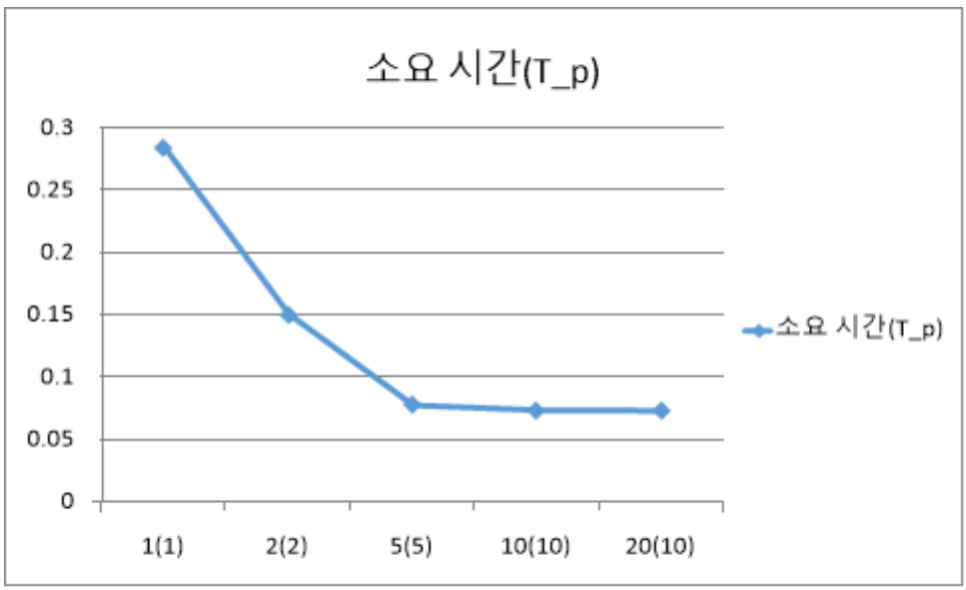
그래프를 분석해보면 거의 그래프들이 겹쳐서 보일 정도로 앞에서의 event-driven server의 경우보다 동시처리율의 차이가 더욱 근소하여 자료상 특징이 있다고 보기는 어렵지만, 위에 표와 같이 각각의 워크로드에 대한 평균값을 봤을 때 대소를 비교해보면 눈에 띄는 것은 show로만 이루어진 요청들에 대한 서버의 동시처리율이 sell, buy보다 더 높다는 것이다. 이는 sell, buy에서 이루어지는 작업들이 세마포어에 의해 통제되고 있기 때문임을 짐작할 수 있다. 이 사실은 앞에서 했던 예측과 부합한다.

③ Thread-based 동시서버에서의 쓰레드 개수에 따른 성능 분석

쓰레드의 개수를 점진적으로 증가시키면서 Thread-based 동시서버의 성능이 어떻게 개선되는지 확인해보려고 한다. 우선 lscpu | grep Thread 명령어를 통해 Threads per core가 2임을 보아 cspro 서버가 쓰는 cpu는 hyperthreading이 되어있음을 알 수 있었다. 또한 추가적으로 확인해본 결과 cpu core의 개수(p)는 10이었다.

```
cse20181671@cspro8:~/20181671/task_2$ lscpu | grep Thread
Thread(s) per core:    2
```

먼저 Client의 수는 100, Client당 요청의 개수는 20으로 고정하고 실험을 진행한다. 이렇게 실행을 진행했을 때 수업시간에 Thread Parrellism에서 배웠던 내용에 기반해 Speedup ($S_p = \frac{T_1}{T_p}$), Efficiency Metric($\frac{S_p}{p}$)을 도입하면 표와 소요시간에 대한 그래프는 아래와 같다.



	1(1)	2(2)	5(5)	10(10)	20(10)
소요 시간(T_p)	0.284062	0.150134	0.076896	0.072941	0.072459
속도 향상(S_p)	1	1.8920564	3.6941063	3.8944078	3.9203136
효율성 (E_p)	100	94.602821	73.882127	38.944078	19.601568

표. 가로 방향은 쓰레드의 수 (실제 cpu core의 수), 해당값은 표 참조

우선 그래프를 보면 처음에는 쓰레드 수를 늘인 배수만큼 소요시간이 그에 거의 비례하여 줄어드는데, 쓰레드 개수 = 5를 기점으로 소요시간이 더 이상 비례해서 감소하지 않는 것을 볼 수 있다. 표를 보면 마찬가지로 속도 향상이 쓰레드 수 = 5를 기점으로 크게 일어나지 않고 있고, 효율성도 쓰레드 수 = 5를 기점으로 크게 감소하고 있다. 하지만 어디까지나 쓰레드 개수를 증가시켰을 때 기대한 만큼의 성능변화가 일어나지 않는다는 것이지, 쓰레드 개수를 증가시켰을 때 어느 정도의 정체는 있겠지만 성능 향상이 아예 없다는 것은 아니다. 그렇지만 더 많은 쓰레드를 다룰 때는 쓰레드를 생성하는 비용과 세마포어의 사용 등에 의해 오버헤드가 발생할 수 있으므로 무조건적으로 쓰레드 수를 늘인다고 성능향상이 이루어질 것이라는 명제 또한 잘못됐다.