

System Programming Project 2

1. 개발 목표

이번 프로젝트의 목표는, 강의 시간에 배운 'Concurrent Server 구축 방법론' 중 'Event-based Concurrent Server'와 'Thread-based Concurrent Server' 개념을 기반으로, 복수의 클라이언트가 동시에 다양한 명령을 요청할 때, 이를 'Concurrency Issue' 없이 처리할 수 있는 '주식 서버'를 구축하는 것이다. 또한, 구축한 두 서버의 성능을 '동시처리율'을 기준으로 직/간접적으로 비교하여 강의 시간에 배운 이론이 현실에선 어떠한 양상으로 나타나는지를 확인하는 것도 목표이다.

이를 위해, 우선 'Concurrent Server'를 구축해야 한다. Task1은 'I/O Multiplexing' 기법을 기반으로 한 'Event-based Concurrent Server' 구축이다. 복수의 클라이언트가 보내는 Connection Request를 서버가 수신할 때마다 생성되는 Connected Descriptor들을 Listening Descriptor와 함께 하나의 Array로 묶는다. 단일의 Main Process는 Iteration을 하면서 매 순회마다 Select 함수로 해당 Array의 File Descriptor를 중 Pending Input이 있는 Descriptor들을 골라내고, 각 Input에 대응하는 Event Handling을 함으로써 'Concurrent Server'를 구축한다.

Task2는 'Thread-based Concurrent Server' 구축이다. 서버의 기능 자체는 Task1의 서버와 동일하다. 다만, 'Concurrency'를 제공하는 방법이 'Thread'인 것으로, 강의 시간에 다룬 'Producer-Consumer Problem' 기법을 적용해 'Thread Pooling'을 구현한다. 물론, 단순히 매 Connection Request마다 새 Thread를 띄우는 식으로도 구축할 수 있지만, 그렇게 하면 Thread 생성 시의 Overhead가 Service의 속도에 개입하게 되므로, 그렇게 하지 않고, Thread Pooling을 통해 미리 Worker Threads(Consumers)를 띄워놓는 방식을 채택한다. 새로운 Connected Descriptor가 생성될 때마다 Shared Buffer에 Item(Connected File Descriptor)을 추가하고, 그 때마다 Worker Thread가 작동하여 Item을 Remove하고 Service하도록 처리한다.

한편, Task1과 Task2에서 개발하는 두 주식 서버는 커다란 시각에서 보면 '주식 정보 로드 -> 각 Client와의 Connection 형성 -> 각 Client에게 Service -> 주식 정보 업데이트'의 흐름을 가진다.

마지막 Task3에서는 Task1과 Task2에서 개발한 두 서버를 동일한 조건 하에 두고 복수의 클라이언트가 동시에 여러 명령(요청)을 각 서버에 보냈을 때, 두 서버의 동시처리율이 어떤 형태로 나타나는지를 확인하고, 그 결과를 다방면으로 분석한다. 그 외에, 프로젝트 수행자 본인이 개발 과정에서 궁금했던 몇 가지에 대해서도 추가 실험을 진행한다. 각 Task에 대한 자세한 내용은 아래에서 설명한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

상술한 것처럼, Select 함수와 'I/O Multiplexing' 기법을 사용하여 단일 프로세스 기반의 'Event-based Concurrent Server'를 구축하였다. 복수 클라이언트의 여러 명령을 문제없이 처리한다.

2. Task 2: Thread-based Approach

'Producer-Consumer Problem' 기법을 적용한 'Thread Pooling'과, 각 Stock Item에 대한 'First Readers-Writers Problem' 기법 적용을 바탕으로 'Thread-based Concurrent Server'를 구축하였다. 역시나 Task1과 마찬가지로 복수 클라이언트의 여러 명령을 문제없이 처리한다.

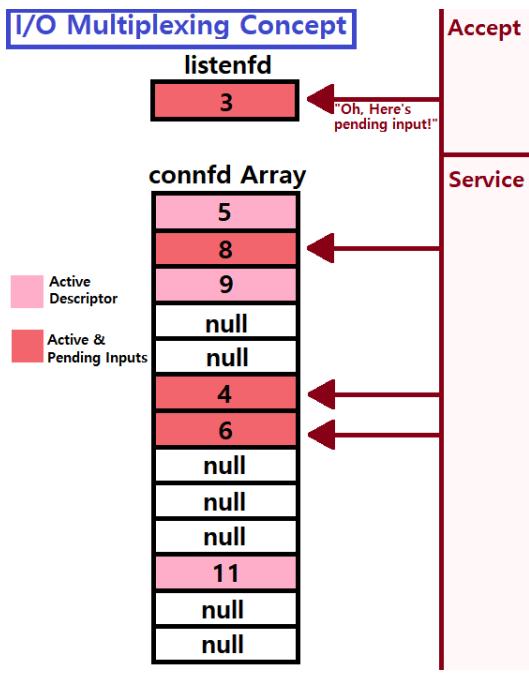
3. Task 3: Performance Evaluation

Task1과 Task2에서 각각 개발한 'Event-based Concurrent Server'와 'Thread-based Concurrent Server'를 '동시처리율(시간당 클라이언트 처리 요청 개수)'을 기준으로 여러 관점에서 평가한다. '관점'에는, '클라이언트 개수', '워크로드' 등이 있다. 우선, 결론만 언급하자면, Thread-based Server의 성능이 'Event-based Server'에 비해 더 우수하다.

본 실험 외에도 추가 실험을 두 가지 진행하였는데, 하나는 '서버의 기반 자료구조 AVL Tree의 효율성 검증'이고, 또 다른 하나는 'Thread-based Server의 동시처리율 Threshold 확인'이다. 모든 실험은 본인의 CSPRO 계정을 통해 모두 '실제로' 진행되고, 마무리된 실험으로, 자세한 실험 과정과 결과 분석은 아래에서 서술한다.

B. 개발 내용

- Task1 (Event-driven Approach with select())
 - ✓ Multi-client 요청에 따른 I/O Multiplexing



본 'Event-based Concurrent Server'의 'I/O Multiplexing'은 강의 시간에 다룬 개념을 적극적으로 도입하여 구축하였다. 'I/O Multiplexing'은 단일 프로세스로 두 종류의 이벤트를 동시에 처리하는 서버를 만들고자 할 때 적용할 수 있는 기법으로, 'Active Connection'에 대한 Array를 통해 구현할 수 있다.

Server는 각 클라이언트로부터 Connection Request가 올 때, Channel을 형성하고, 그 과정에서 생성되는 Connected Descriptor를 'Active Connection Array'에 삽입한다. 참고로 이때, 사전에 Listening File Descriptor를 먼저 Array에 넣어둔다.

이후, Server는 'Active Connection Array'를 순회하면서 '어떤 Descriptor에 Pending Input이 존재하는지'를 지속적으로 체크한다. 이를 select 함수를 이용해 수행한다. Connected File Descriptor에 Pending Input이 있으면 해당 Connection의 Client가 전송한 요청에 대해 Service를 제공하고, Listening Descriptor에 Pending Input이 있으면 새로운 Connected Descriptor를 Array에 삽입하는 식으로 서버를 구축한다.

이때, Linux에서 File Descriptor를 Set의 형태로 관리할 수 있도록 제공하는 fd_set Type을 이용해 read_set과 ready_set이란 Set을 개별적으로 마련한다. read_set은 'Active Connection Array'를 의미하고, ready_set은 read_set의 부분집합, 복사본으로, select에서 read_set을 직접적으로 조정할 경우 발생할 수 있는 데이터 손실을 방지하기 위해 존재한다.

fd_set Type에 대한 함수 select, FD_SET, FD_ZERO, FD_ISSET 등을 이용하여 앞서 설명한 'I/O Multiplexing'을 구현할 수 있다. 본인은 강의 시간에 배운 것처럼, Pool이라는 구조체를 도입하여 이를 구현하였으며, 자세한 Code-Level의 Detail은 후술한다.

✓ epoll과의 차이점

Linux에서 I/O Multiplexing을 구현할 때는 select, poll, epoll이라는 세 함수가 대표적으로 사용된다. 그 중 poll은 주제에서 벗어나므로 논외로 한다.

select 함수는 epoll에 비해 역사가 오래된, 전통적인 기법이라 할 수 있다. 동명의 select라는 System Call을 사용하여 Kernel이 'Ready File Descriptor Bit Vector'를 체크하도록 지시한다. Kernel에게 "I/O Events가 발생하면 다시 프로세스로 돌아가게 해줘!"라고 요청하는 것이다. epoll 함수 역시 마찬가지로 Kernel을 통해 I/O Events를 감지해서 프로그래머에게 알려준다.

이 둘의 차이점은 바로 '내부 구현 방식'에 있다. select는 매번 함수가 호출될 때마다 'File Descriptor Set'을 User-Level에서 Kernel-Level로 복사한다. 이어서 해당 Set을 순차적으로 조회하며 일일이 "이 Descriptor에 Pending이 있나?"하고 살핀다. 반면에, epoll은 '관찰하고자 하는 Descriptors'를 관리하는 하나의 통합적인 공유 메모리 공간과, 그 공간에 대한 별도의 Descriptor를 미리 마련하고, 해당 Descriptor를 통해 Kernel의 Event Table을 추적하여, Event에 대응하는 방식이다.

즉, select 함수는 매번 호출될 때마다 File Descriptor Set의 Copy가 일어나지만, epoll은 관찰하고자 하는 Descriptors를 Kernel에게 최초에 알릴 때에만 한 번 Copy가 진행된다. 따라서, select는 매번 호출 시마다 복사로 인한 Overhead, 순회로 인한 Overhead가 발생하는 것이고, 그렇기 때문에 epoll보다 성능 측면에서 좋지 않다는 특성을 보인다. 또한, select는 epoll과 다르게 File Descriptor 개수에 대한 제한이 매우 작다는 문제점도 있다. 상한의 기본 값이 1024개라고 알려진다.

물론, select에게도 장점은 있다. 아무래도 epoll에 비해 역사가 길기 때문에 대부분의 System에서 이를 지원하고, 따라서 Portability가 높다는 측면도 존재한다. 하지만, 실제 '상용 서버 구축'이라는 측면에서 이는 그다지 주요한 특징은 아닐 것이다. 속도 자체가 느리기 때문이다.

요약하자면, select와 epoll은 '내부 구현 방식'에 차이가 있으며, 그 차이로 인해 select와 epoll의 성능 차이가 발생한다.

- Task2 (Thread-based Approach with pthread)

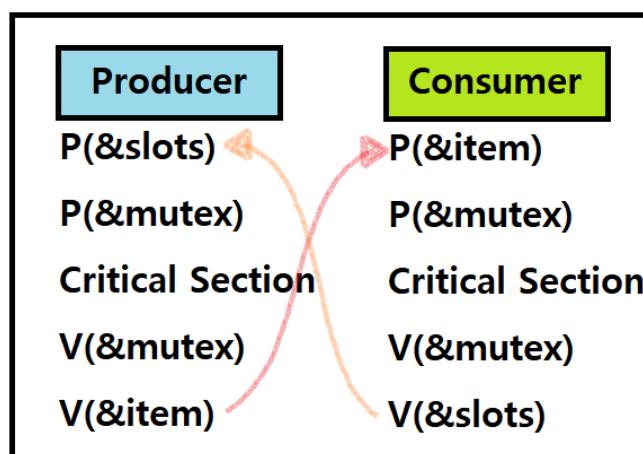
- ✓ Master Thread의 Connection 관리

상기한 것처럼, 'Producer-Consumer Problem'에 기반한 'Thread Pooling' 기법을 적용했다. Main Thread가 곧 Master Thread, Producer이다. main 함수에서 Consumer 역할의 Worker Threads를 미리 만들어놓고, 현재 사용하지 않는 (최초엔 모두 해당) Thread들을 Sleep시킨다. Worker Thread가 필요할 때만 Wake Up 시켜서 일을 시키는 것이다.

이를 위해, Shared Buffer를 구축한다. 이 Buffer에 대해선 Slot, Item이라는 Counting Semaphore가 있다. (Available) Slot은 Buffer의 빈 칸을, (Available) Item은 Buffer의 비어있지 않은 칸에 있는 Descriptor를 의미한다고 보면 된다. Master Thread는 이 Shared Buffer를 이용해 Connection을 관리한다.

새로운 Connected Descriptor가 생성되면, Master Thread는 이를 Shared Buffer에 Item으로 삽입한다. Available Slot이 하나 줄어들고, Available Item이 하나 늘어난 것이다. 이를 Semaphore 관점에서 보면, Slot에 대한 P Operation, Item에 대한 V Operation이 일어난 것이다.

Available Item이 있으면, Worker Thread가 해당 Item을 Buffer에서 제거하고, 일을 수행한다. Item의 제거는 Available Slot이 하나 늘어남을 의미한다. 따라서, Item 제거 시, Item에 대한 P Operation, Slot에 대한 V Operation이 수행된다. 이를 그림을 나타내면 아래와 같다.



즉, Connected Descriptor를 Item으로 하는 'Producer-Consumer Problem'을 적용하여 Master Thread가 Connection을 관리하는 것이다.

✓ **Worker Thread Pool 관리하는 부분**

앞선 항목에서 설명한 'Thread Pooling' 기법을 좀 더 자세히, Worker Thread 관점에서 설명한다. 우선, Buffer를 가리키는 포인터, Slot의 최대 개수, Buffer의 Front를 가리키는 변수, Rear를 가리키는 변수, Shared Data에 대한 Access를 직렬화하는 mutex Semaphore, Available Slot 개수를 Notify하는 slots Semaphore, Available Item 개수를 Notify하는 items Semaphore를 모두 하나의 구조체에 포함시킨다. 해당 구조체를 sbuf_t라 정의한다.

sbuf_t Type의 전역 변수를 하나 둔다. 이 변수가 Shared Buffer 역할을 도맡을 것이다. 해당 구조체 변수를 Master Thread에서 초기화하고, 미리 Worker Threads를 Pthread_create 함수를 이용해 생성해둔다.

Master Thread이자 Main Thread가 Listening Descriptor를 통해 Connection 을 구축할 준비가 되면, 시시각각 Connected Descriptor를 반환 받게 되는데, Descriptor가 반환될 때마다 이들을 Item으로 하여 Shared Buffer에 삽입한다. 이 삽입 과정에서, 앞선 항목에 포함된 그림에서 본 것처럼, Item에 대한 V Operation이 수행된다.

Worker Thread들은 Shared Buffer에 Item이 생기는 것을 상시 대기하고 있다가, V(&items)의 Notification에 의해 Available Item이 있음을 알게 되면, Shared Buffer에서 해당 Item을 제거한 후, 해당 Item(=Connected Descriptor) 을 이용해 Client에게 Service를 제공한다. 이렇게 Worker Thread의 Routine 을 만들면 된다. 서비스를 마치면 Connected Descriptor를 Close하고, 다시 Sleep 상태에 돌입한다. 실제 Code-Level Detail은 첨부 코드를 통해 확인할 수 있다.

- **Task3 (Performance Evaluation)**

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

Task3에서는 Task1과 Task2에서 구현한 두 'Concurrent Server'의 성능을 '동시처리율(시간당 클라이언트 처리 요청 개수)'를 기준으로 비교한다. 본인은 Task3 실험에서 얻고자 하는 Metric을 다음과 같이 분류하여 정의하였다.

1) 첫 번째 실험 : AVL Tree 효율성 검증 실험

~> **Metric** : 단일 Client가 하나의 명령을 보낼 때, Stock 개수와 명령 유형에 따른 명령 처리 소요 시간

(명령 유형 : show, sell, buy / stock 개수 : 5, 100, 10000, 1000000)

~> **설명** : 이 실험은 Main 실험은 아니다. 본인은 Task1, Task2에서 'Concurrent Server'를 구축할 때, 자료구조로 AVL Tree를 도입하였는데, 그 이유는, 일반적인 Binary Search Tree는 데이터 입력이 값의 기준으로 오름차순이나 내림차순으로 들어오는 식의 Linear한 데이터 입력이 있을 시, Tree가 사실상 Array로 작용해버리는 문제점을 해결하고자 함이었다. 이 실험을 통해, 설계한 AVL Tree가 효율적인지를 검증한다.

나아가, 명령의 종류에 따른 단일 명령 처리 시간도 비교해볼 수 있다. 이 실험을 통해, 이어서 진행되는 실험들에서 좀 더 객관적인 지표를 얻을 수 있는 세부 정보들을 알아낼 수 있다고 판단했다.

한편, stock 개수는 최대 1,000,000개까지를 상정하였는데, 물론 더 취급 할 수 있지만, 자료 조사 결과, 국내 총 주식 수는 코스닥 기준 대략 1500개 정도밖에 되지 않기 때문에 너무 큰 숫자에 대한 효율성 검증은 불필요하다고 판단했다(본 프로젝트의 목적과도 부합하지 않는다).

~> **측정 방법** : stockserver.c의 service 루틴의 시작과 끝에 gettimeofday 함수를 설치해 '서버 단의 요청 처리 시간'을 측정한다. 두 시간의 차이가 곧 '(서버 내에서의) 단일 명령 처리 시간'이 된다. 한편, Tree의 효율성 검증을 위해 수많은 주식 정보를 생성해내는 stockgen이라는 프로그램을 따로 설계하였다. 인자로 입력 받은 수 만큼 주식 정보를 생성해낸다. stockgen 프로그램의 자세한 코드는 C번 항목에서 소개한다.

2) 두 번째 실험 : 확장성과 워크로드에 따른 서버 동시처리율 측정 실험

~> **Metric** : Client 개수와 명령 유형에 따른 서버 동시처리율

(Client 개수 : 1, 5, 10, 20, 50 / 명령 유형 : show, sell, buy, Random)

~> **설명** : Task3의 Main 실험이다. 확장성과 워크로드를 동시에 고려해 두 서버의 성능을 비교, 평가할 수 있는 실험이다.

~> **측정 방법** : 제공된 multichannel 프로그램을 이용하여 실험한다. 각 Client Process는 20개의 명령 요청을 서버에게 보내도록 설정하였다. '명령 요청 횟수'를 20으로 설정한 이유는, 동시처리율이 과하게 커지지 않으면서도, 변화가 뚜렷하게 보이는 수치로 20이 가장 적절하다고 판단했기 때문이다. Concurrent Server는 CSPRO의 성능을 고려했을 때, 필연적으로 명령 요청이 많아질수록 동시 처리율이 올라가게 된다. 현대 CPU 성능 상 짧은 시간 내에 많은 처리가 특정 상한까지는 충분히 가능하기 때문이다. 이 말은 즉, multichannel의 각 Client가 너무 많은 명령을 동시에 쏘면, 자연히 동시처리율이 올라가는데, 이때, ORDER_PER_CLIENT 값을 크게 설정해놓으면, Client 개수가 적을 때에도 큰 수치의 동시처리율이 나와버려 Client 개수가 늘어나더라도 (ORDER_PER_CLIENT가 낮을 때와 비교했을 때, 상대적으로) 뚜렷한 차이를 보이지 않게 된다(물론 차이는 벌어지지만 말이다). 따라서, 이를 몇 차례 간단히 확인해보았는데, 20~30 사이의 수치가 실험에 적합하다고 결론지었다. 따라서, multichannel의 각 클라이언트는 20개의 요청을 보낸다.

한편, 동시처리율 측정을 위한 '수행 시간 측정' 역시 multichannel 프로그램을 기준으로 측정하였다. stockserver와 multichannel 중 어느 프로그램에서 시간을 측정하는 것이 합리적인지를 상당 시간 고민하였는데, 두 서버의 정확하고 공정한 비교를 위해선 공통된 multichannel에서 측정하는 것이 옳다고 판단했다.

이유는 크게 두 가지이다. 첫째로는 직전에 언급한 것처럼 실험 환경의 통일성을 주기 위함이고, 둘째는 'Server Service의 진정한 의미'를 고려해서이다. 서버의 서비스라 함은, Client가 Server에게 요청을 보낸 시점부터 시작해서, Server가 요청에 대해 처리한 결과가 최종적으로 Client에게 도달한 순간에 완료된다. 단순히 서버 쪽에서 '요청에 대한 처리'를 시작해서 마무리하는 것만 고려하면, 클라이언트에서 서버로 네트워크를 거쳐 요청을 보내는 것과, 서버에서 네트워크를 거쳐 클라이언트로 요청 처리 결과를 보내는 것이 고려되지 않는다. 물론, 단일 명령의 수행을 따질 때는 이러한 시각은 문제가 없다. 하지만, multichannel 프로그램 수행 환경과, 수많은 명령이 동시다발적으로 특정 시간 간격 동안 서버로 도달하고, 그 결과가 클라이언트로 다시 Write Back하는 흐름을 지니는 이번 실험 상황을 고려하면, multichannel 측에서 네트워크를 고려하여 시간을 측정하는 방

식이 더 합리적이라 판단했다.

서버 쪽에서 동시다발적인 명령에 대한 처리 시간을 측정하면, 일부(특히 최초와 최초에 가까운) 명령에 대해서 네트워크를 거쳐 도달하는 시간을 고려하지 않게 된다. 일부는 고려하는데 말이다. 또한, 결과를 다시 클라이언트로 보내는 시간 역시 마찬가지로 일부는 고려되고 일부(특히 마지막과 마지막에 가까운)는 고려되지 않는다. 이에 대한 명확한 구분이 어려워지는 것이다. 따라서 전체 수행 시간 측정을, 서버 단이 아니라, multiclient 단에서 네트워크 시간까지 충분히 고려하여 측정하는 것이다.

물론, multiclient에서 측정하면 복수의 클라이언트 프로세스를 fork하고 reaping하는 Overhead가 측정 시간에 포함된다. 하지만, 각종 자료조사 결과 현대 Linux System 수준에서 fork와 reaping의 Overhead는 매우 경미하다는 점을 확인했다. 실제로 multiclient에서 측정한 시간과 이벤트 서버에서 측정한 '전체 수행 시간'을 비교해본 결과, 그 간격이 거의 차이 나지 않았다. 반면, 양 Case의 측정 '시작 시각'을 비교해보면, 그 차이는 0.01ms 정도로, 생각보다 컸다. 즉, 네트워크 속도가 더 영향이 큰 것이다. 따라서, 클라이언트가 네트워크를 거쳐서 요청을 전송하고, 네트워크를 거쳐서 결과를 받는 것을 모두 포함한 시간을 측정하기로 결정했다.

앞 문단에서 '이벤트 기반 서버에서 측정한 시간'을 언급한 점에서, "두 Case의 시간 간격이 유사하면, 굳이 multiclient에서 측정할 이유가 있는가?"라고 반문할 수 있다. '네트워크 통신 시간 고려'라는 것이 약간은 불명확한 이유로 들릴 수 있기 때문이다(이 점은 본인도 인정한다).

사실, 이벤트 기반 서버에서는, '전체 수행 시간' 측정이 위에서 언급한 '네트워크 시간 고려가 비-일관적임'을 제외하면 크게 문제는 없다. 하지만, 문제는 '(서버 단에서의) Thread 기반 서버 전체 수행 시간 측정'에서 발생한다. 최초 도달 명령(시간 측정의 시작점)을 식별하는 것은 문제 없지만, 마지막 명령 처리(시간 측정의 종료 시점)를 식별하기 위해선 count 가 필요하고, 문제는 여기서 발생한다. Thread 기반 서버에서 Counting을 하려면 Shared Variable에 대한 추가적인 Synchronization이 필요한 것이다. Semaphore를 추가하면 이벤트 서버에 비해서 Thread 서버에게 불리함을 안기는 행위가 된다. 직렬화가 이루어지기 때문이다. multiclient 프로그램에서 마지막 명령 전송을 알리는 방법도 존재하지만, 이 역시

Concurrency Issue에서 자유롭지 않다.

따라서, 이러한 종합적인 이유들로 인해, multiclient에서 수행 시간을 측정하기로 결정했다. 장황한 내용을 요약하면 다음과 같다.

※ 'Multiclient에서의 수행 시간 측정'을 결정한 이유

1. 실험 환경에 통일성 부여
2. 서비스의 의미를 고려해, 네트워크 통신 시간도 모두 포함
3. 서버 단에서 시간 측정 시, 두 서버에서의 시간 측정 Overhead 크기가 달라 두 서버 비교 간에 불공정함이 발생할 가능성이 높음.

한편, 실험 결과의 정확성을 증대시키기 위해 'exec100'이라는 프로그램도 도입하였다. 이 프로그램은 multiclient 프로그램을 순차적으로 100번 수행하는 프로그램이다. 위에서 언급한 실험 설계에 따라 multiclient 프로그램에서 '시간 측정' 및 '측정 시간에 따른 동시처리율 계산', '계산된 동시처리율을 experiment.txt에 출력'이라는 일들을 수행한다. 'exec100'은 이러한 '실험을 위해 변형된 multiclient 프로그램'을 연속적으로 100번 수행한다. 그 다음, experiment.txt 파일에 기록된 각 수행에서의 동시처리율들을 모두 더하고, 100으로 나누어 평균값을 구하고, 그 값에 소수점 버림을 적용한다.

순차적인 multiclient 수행에는 system 함수를 사용했다. system 함수는 '함수로 인해 수행된 명령/프로그램의 종료'간에 발생하는 SIGCHLD가 Parent Process에 도달하기 전까지 Parent를 Block한다. 즉, system 함수를 Loop를 이용해 순차적으로 호출하면, 그것이 곧 '순차적인 실험 수행'이 되는 것이다. 'exec100' 프로그램의 자세한 코드는 C번 항목에서 설명한다.

한편, 본 실험에서 Client 개수는 최대 50개까지만 설정하였다. CSPRO Server의 과부하를 방지하기 위해 Client 50개까지만 확인하였다.

- 3) 세 번째 실험 : Thread-based Server에서 동일 Shared Variable에 대한 Write명령이 몰릴 경우(Client 수가 늘어날 경우), 서버 동시처리율이 오히려 떨어지기 시작하는 지점이 있는지 확인하는 실험

~> **Metric** : 모든 Client가 '동일 Stock 종목'에 대한 sell 연산만을 수행할 때, Client 개수에 따른 Thread-based Server 동시처리율

(Client 개수 : 1, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100)

~> **설명** : Task3의 번외 실험이다. 본 System Programming 강의에서, Thread Programming 시, 일을 수행하는 Thread가 많더라도 Mutex에 대한 과도한 경쟁이 벌어지면 오히려 성능이 느려질 수 있는 상황에 대해 알아보았는데, 이번 프로젝트에서 개발한 'Thread-based Concurrent Server'에서도 그러한 현상이 발생하는지를, 동일한 Write 명령만을 요청하는 Client들을 상정해, 그들의 개수를 늘려가며 확인해본다. 이를 통해, 궁극적으로 Thread-based Server의 '성능 향상의 Threshold'가 있는지를 확인한다.

한편, 본 실험에서는 본인이 개발한 Thread-based Stock Server가 Concurrency Issue를 발생시키는지도 명시적으로 확인할 수 있다. Write 명령을 동일 Stock Item의 left_stock 변수에 대한 Increment로 한정시켜서 말이다. 대신, 이를 위해 최초 stock.txt 로드 시 해당 Stock Item의 left_stock이 0이어야 한다.

~> **측정 방법** : 측정 방법은 두 번째 실험과 동일하다. 다만, multiclient의 Client Process가 오로지 'sell 1000 1'이라는 명령만 요청하는 것이다 (전체 종목 수를 1000개로 설정하였다). ORDERS_PER_CLIENT는 50개로 설정한다. 최대한 많은 명령이 동시에 서버에 도달하도록 하기 위함이다.

서버의 과부하를 방지하기 위해 클라이언트 개수는 최대 100개까지만 설정하였다(exec100으로 100번씩 검증할 것이므로 수치가 낮아야 한다).

✓ Configuration 변화에 따른 예상 결과 서술

1) 첫 번째 실험 : AVL Tree 효율성 검증 실험

~> **예상** : 개발 과정에서 AVL Tree를 이론적으로 문제 없이 설계하였기 때문에 Stock Item 개수가 많고, 순차적으로 Load 되더라도 좋은 검색 성능을 보일 것이라 생각한다. AVL Tree의 검색 시간 복잡도는 $O(\log N)$ 이므로, N이 1개이든, 1,000개이든, 1,000,000개이든 큰 시간 차이를 보이지 않

을 것이다.

한편, 본인은 서버 개발 과정에서 show 명령에 대해 'Tree Traversal' 방식 대신, 그냥 Array를 사용하였다. AVL Tree에 새 Loaded Item을 삽입할 때, 해당 Node의 주소 값을 'print'라는 Array에 함께 삽입한다. 즉, Array의 각 Element가 Node 그 자체를 가리키는 것이다. Array를 도입한 이유는, Tree Traversal이나, Array나 어차피 전체 조회를 위해선 $O(N)$ 의 시간 복잡도가 필요한데, Array에 대한 Linear Traversal이 코드가 좀 더 간단하고, 동시에 stock.txt 저장 형태 그대로 유지하는 것이 가능하므로 Array의 장점이 더 크다 판단하여 Array를 도입하였다.

본론으로 돌아가자면, 본 실험에서, show 명령에 대한 수행 시간이 sell과 buy에 비해 상당히 비효율적일 것이라 예측하였다. 당연한 것이, 전체 자료구조에 대한 조회가 필요하기 때문이다.

참고로, 본 실험은 Event-based Server에 대해서만 진행한다. 실험 목적 자체가 AVL 자료구조의 효율성 검증이고, 물론, 각 명령 간의 수행 속도 차이를 비교할 것이지만, 단일 명령의 수행 속도를 검증하는 것이기 때문에 굳이 Thread-based에서 추가로 이를 확인할 필요는 없다고 판단했다.

2) 두 번째 실험 : 확장성과 워크로드에 따른 서버 동시처리율 측정 실험

~> 예상 : Event-based Concurrent Server는 기본적으로 단일 프로세스 기반이고, 하나의 Logical Control Flow를 가지므로 현대 Multicore System의 이점을 살리지 못한다. 반면 Thread-based Concurrent Server는 멀티 코어를 비롯한 Hyper-Threading, Parallel Processing의 이점을 충분히 살릴 수 있다.

물론, Thread-based Server는 Semaphore를 사용하므로 Synchronization Overhead가 존재한다. 하지만, 이 정도는 서버 수행 속도에 Critical한 영향을 줄 정도는 아닐 것이라 예상한다. 따라서, 전체적으로 모든 Configuration 상황에서 Thread-based Server가 좀 더 우수한 성능을 보일 것이라 예측했다. 다만, 그 정도가 언급한 Overhead 등으로 인해 그리 크진 않을 것이라 생각했다.

한편, show 명령에 있어서는, Event-based Server에선 그 이론적 구조

때문에 앞선 Client의 show Request가 완료되고 나서야 이후 Client의 show Request가 가능하지만, Thread-based Server에서는 동일 Stock Item 조회 상황을 제외하고는 각 Thread가 동시에 show Service가 가능하므로, 두 서버의 성능 차이가 두드러질 것이라 예상했다.

또한, 앞서 언급한 것처럼 현대 CPU 성능 자체가 우수하기 때문에 Client 개수가 늘어날수록 두 서버 모두 동시처리율이 상승할 것이라 예측하였다.

3) 세 번째 실험 : Thread-based Server에서 동일 Shared Variable에 대한 Write명령이 몰릴 경우(Client 수가 늘어날 경우), 서버 동시처리율이 오히려 떨어지기 시작하는 지점이 있는지 확인하는 실험

~> 예상 : 강의 시간에 학습한 것처럼 Client 개수가 늘어날 때, 어느 순간을 기점으로 Thread-based Server의 성능이 떨어지는 현상이 발견될 것이다. 그 정확한 지점은 실험 이전에는 쉽사리 예측할 수 없었다.

또한, 서버 개발 과정에서 철저한 검증을 거쳤으므로 Concurrency Issue 역시 발생하지 않을 것이다.

C. 개발 방법

(1) AVL Tree Implementation

AVL Tree의 Node는 다음과 같은 구조체 Type으로 이루어진다.

```
typedef struct item {          /* node structure of AVL tree */
    int ID;                      // ID, left_stock, price : attributes of stock item
    int left_stock;
    int price;
    int height;                  // balance factor of node (for AVL operations)
    int readcnt;                 // number of Readers who access the node
    sem_t mutex, w;              // semaphores for 'First Readers-Writers Problem'
    struct item *right;
    struct item *left;
}Item;
```

이때, height 변수는 AVL Tree 조정 연산을 위한 'Balance Factor'이며,

readcnt, mutex, w와 같은 변수는 Thread-based Concurrent Server에서 'First Readers-Writers Problem' 기법을 위한 변수들로, Event-based Server 소스 코드에는 등장하지 않는다.

```
/* Left single rotation */
Item* SingleRotateLeft(Item *nodeB) {
    Item* nodeA = NULL;

    nodeA = nodeB->left;
    nodeB->left = nodeA->right;      // rotating process
    nodeA->right = nodeB;

    nodeB->height = GetGreater(GetHeight(nodeB->left), GetHeight(nodeB->right)) + 1;
    nodeA->height = GetGreater(GetHeight(nodeA->left), GetHeight(nodeB)) + 1;

    return nodeA;
}

/* Right single rotation */
Item* SingleRotateRight(Item *nodeA) {
    Item* nodeB = NULL;

    nodeB = nodeA->right;
    nodeA->right = nodeB->left;      // rotating process
    nodeB->left = nodeA;

    nodeA->height = GetGreater(GetHeight(nodeA->left), GetHeight(nodeA->right)) + 1;
    nodeB->height = GetGreater(GetHeight(nodeB->right), GetHeight(nodeA)) + 1;

    return nodeB;
}

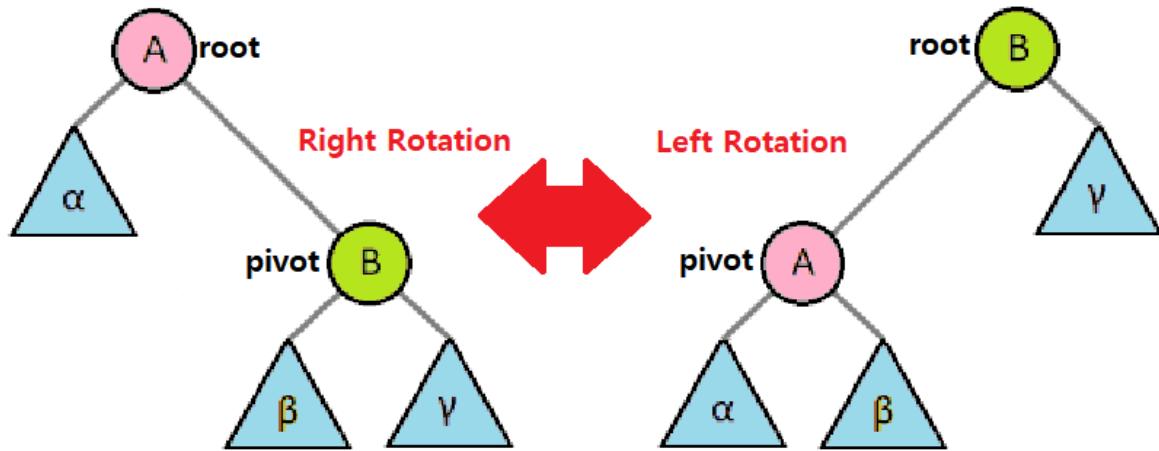
/* Left double rotation */
Item* DoubleRotateLeft(Item *node) {
    node->left = SingleRotateRight(node->left);

    return SingleRotateLeft(node);
}

/* Right double rotation */
Item* DoubleRotateRight(Item *node) {
    node->right = SingleRotateLeft(node->right);

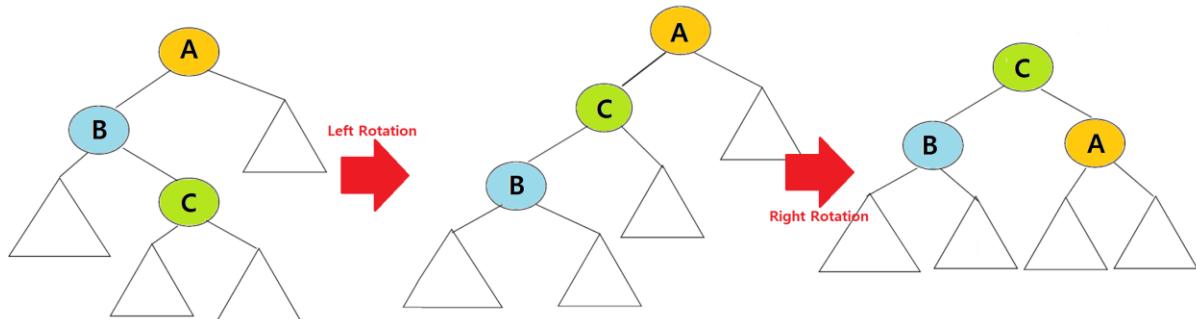
    return SingleRotateRight(node);
}
```

이 함수들은 모두 AVL Tree의 Tree 재구성 연산들이다. 기본적인 Binary Search Tree 코드를 기반으로 '알고리즘 설계와 분석' 강의에서 학습한 내용을 적용하여 코드를 작성하였다. 아래의 그림은 Left와 Right Single Rotation을 표현하였다. AVL Tree는 Node 삽입을 제외하고는 BST와 구성이 다르지 않다. (Node 삭제 기능은 본 프로젝트에서 필요 없다.)



이 연산들을 기반으로 Tree Insertion을 다음과 같이 구현한다.

- 기본적인 흐름은 BST의 In-Order Recursive Insertion과 같다.
- 새로 삽입하는 Node의 Key값(ID)이 현재 조회 중인 Node의 ID 값보다 크면, 해당 Node의 우측 Link로 재귀 흐름을 호출한다(여기까진 일반적인 BST 삽입 논리와 동일).
- 연결 후, 현재 조회 Node의 좌측 링크와 우측 링크의 Height를 비교 한다. 그 차이가 2일 경우, Tree의 균형이 깨진 상황이므로, AVL Operation을 진행한다. 우측 링크 Tree의 Root ID가 새 삽입 Node ID 보다 작으면, 위의 그림에서 우측에서 좌측으로 변환하는 관계를 참고해, Single Right Rotation을 진행한다. 새 삽입 Node ID가 더 작으면, 아래 그림과 같이 Double Right Rotation을 수행한다.



- 반대로, 새 Node Key값이 현재 조회 Node ID보다 작으면, 위의 설명과 반대로 진행하면 된다.

이러한 흐름으로 AVL Tree Insertion을 구성할 수 있다. 나머지 AVL Tree Freeing, Search 과정은 BST의 그것과 정확히 동일하다. AVL Tree 구현 과

정에 대한 추가적인 설명, 이론적인 디테일은 본 프로젝트의 주제와 벗어나므로 서술이 불필요하다고 판단하여 생략하겠다. 나머지는 첨부 소스 코드로 대체한다.

(2) I/O Multiplexing of Event-based Concurrent Server

I/O Multiplexing을 위해 다음과 같은 구조체를 도입한다. 강의 시간에 학습한 내용을 기반으로 한다.

```
typedef struct {                                     /* structure for I/O Multiplexing */
    int maxfd;
    fd_set read_set;                            // bit vector for 'Active Descriptors'
    fd_set ready_set;                           // subset of 'read_set'
    int nready;                                 // num of file descriptors that has pending inputs
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
} Pool;
```

clientfd 배열은 새로 삽입된 'Active Connected File Descriptor'를 기록한다. maxfd는 현재 고려하고 있는 Active Descriptor 중 번호가 가장 큰 Descriptor를 가리키며, read_set, ready_set은 B번 항목에서 설명한 그대로이다. nready 변수는 Pending Input이 있는 Descriptor 개수를 의미한다. 이들을 최초에 init_pool이란 함수를 통해 초기화한다. read_set에 처음에 listenfd를 기록하는 것이 핵심이다.

```
while (1) {
    pool.ready_set = pool.read_set;
    pool.nready = Select(pool.maxfd + 1, &pool.ready_set, NULL, NULL, NULL);

    if (FD_ISSET(listenfd, &pool.ready_set)) {           // if pending at listenfd,
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

        Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);

        add_client(connfd, &pool);                         // add new connfd to pool
    }

    check_client(&pool);                                // check if there's any pendings at connfds
}
```

위의 코드는 Event Server의 main 함수 내부이다. read_set 보존을 위해 ready_set에 read_set을 복사함을 주목하자. 이어서 select 함수를 이용해 Pending Input이 있는 Descriptor 개수를 체크한다.

listenfd에 Pending Input이 있는지 FD_ISSET 함수로 체크 후, Pending이 있다면, 새로운 Connected Descriptor를 add_client 함수로 clientfd 배열에 추가한다. add_client 함수의 내부는 아래와 같다.

```
/* Add new connected descriptors into the pool */
void add_client(int connfd, Pool *p) {
    int i;
    p->nready--; // decrement the available slots

    for (i = 0; i < FD_SETSIZE; i++) {
        if (p->clientfd[i] < 0) {
            p->clientfd[i] = connfd; // insert into the fd array
            Rio_readinitb(&p->clientrio[i], connfd); // ready for using RIO package

            FD_SET(connfd, &p->read_set); // ready for checking pending

            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi) // coordination
                p->maxi = i;

            break;
        }
    }

    if (i == FD_SETSIZE)
        app_error("Error in add_client!\n");
}
```

nready 변수를 Decrement 후, clientfd 배열을 순회하여, 빈 Slot에 새로운 Connected Descriptor를 삽입하고, read_set에 기록하고 있다.

```
/* Check if there are any pending inputs, and provide service */
void check_client(Pool *p) {
    int n, connfd;
    char buf[MAXLINE];
    rio_t rio;

    for (int i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) { // if pending,
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) { // then read!
                printf("server received %d bytes\n", n);
                service(connfd, buf, n); // and service!
            } else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

다시 앞선 main 함수 내부를 보자. listenfd에 대한 확인 절차가 마무리되면, check_client 함수를 호출한다. check_client는, Pending Input이 있는 Connected Descriptor를 순차적으로 확인해가며, 필요한 서비스를 제공하는 함수이다. 내부는 쪼体育投注과 같다.

Client와의 Connection이 종료되면 Descriptor를 Close하고, 이 정보를 clientfd 배열에 업데이트한다. 이렇게 하면 Event-based Server의 I/O

Multiplexing이 구현된다.

(3) Thread Pooling of Thread-based Concurrent Server

'Producer-Consumer Problem'을 기반한 Thread Pooling을 적용하기 위해 다음과 같은 구조체를 도입한다. 강의 시간에 학습한 내용을 기반으로 한다.

```
typedef struct {                                     /* structure for 'Producer-Consumer Problem' */
    int *buf;                                         // shared buffer pointer
    int n;                                            // maximum number of slots
    int front;                                         // buf[(front+1)%n] (pointing the first item)
    int rear;                                          // buf[rear%n] (pointing the last item)
    sem_t mutex;                                       // provides mutual exclusion for accessing buffer
    sem_t slots;                                       // number of available slots
    sem_t items;                                       // number of available items
} sbuf_t;
```

B번 항목에서 설명한 내용과 동일하다. mutex는 Shared Buffer에 대한 Mutual Exclusion을 제공하는 Binary Semaphore이며, slots와 items는 Producer와 Consumer의 접근을 관리하는 Counting Semaphore이다.

```
sbuf_init(&sbuf, SBUFSIZE);
for (int i = 0; i < NTHREADS; i++)
    Pthread_create(&tid, NULL, thread, NULL);    // spawn worker threads (consumer)

while (1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    Getnameinfo((SA *)&clientaddr, clientlen, client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);
    sbuf_insert(&sbuf, connfd);
}
```

위의 코드는 Thread-based Server의 main 함수이자 Master Thread, Producer의 내부 루틴이다. Shared Buffer에 대한 초기화를 진행 후, Worker Threads(Consumers)를 미리 생성해놓는다.

while 문 내부를 보면, 새로운 Connected Descriptor가 생성되면 이를 곧바로 Shared Buffer에 삽입하여 Worker Thread가 이를 Consume하도록 유도한다(sbuf_insert 함수의 V(&items) Operation을 통해 말이다).

sbuf_insert, sbuf_remove 함수는 아래의 코드와 같이 구성되어 있다. B 번 항목 설명 간에 첨부하였던 그림과 정확히 동일한 흐름을 갖추고 있다.

```

/* Insert new item into the 'rear' point of shared buffer */
void sbuf_insert(sbuf_t *sp, int item) {
    P(&sp->slots);                                // waits for available slots
    P(&sp->mutex);                               // lock
    sp->buf[(++sp->rear) % (sp->n)] = item;    // item insertion (produce)
    V(&sp->mutex);                               // unlock
    V(&sp->items);                             // notify that there's new available item!
}

/* Delete the item at the 'front' point of shared buffer, and return it */
int sbuf_remove(sbuf_t *sp) {
    int item;

    P(&sp->items);                                // waits for available items
    P(&sp->mutex);                               // provides serialization
    item = sp->buf[(++sp->front) % (sp->n)];   // item removement (consume)
    V(&sp->mutex);                               // unlock
    V(&sp->slots);                            // notify that there's new available slot!

    return item;
}

```

Worker Thread의 루틴을 확인해보자. 아래의 코드와 같다.

```

/* Thread routine (routine of 'Worker/Consumer Threads') */
void *thread(void *vargp) {
    Pthread_detach(pthread_self());                // reserve the reaping of thread

    while (1) {
        int n, connfd = sbuf_remove(&sbuf);       // consume the item from the buffer
        char buf[MAXLINE];
        rio_t rio;

        Rio_readinitb(&rio, connfd);
        while ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {           // get requests
            printf("server received %d bytes\n", n);
            service(connfd, buf, n);                                // and service!
        }

        Close(connfd);
    }
}

```

Thread의 Reaping을 예약해둔 후, Consume 및 Service를 수행한다.
Shared Buffer에서 Item(Connected Descriptor)을 꺼내 서비스를 제공하고 있다.

(4) Service Routine (with 'First Readers-Writers Problem')

이제 두 서버 모두에 해당하는 서비스 루틴에 대해 알아보자. 이때, 설명 코드는 Thread-based Server를 기준으로 한다. Thread-based Server에

서는 Shared Data Structure에 대한 (First Readers-Writers Problem 기반) Semaphore Synchronization 루틴이 포함되어 있는데, Event-based Server의 경우 서비스 루틴 코드에서 이 Synchronization 작업만 제거하면 되기 때문이다.

```
/* Choose task based on the type of request */
void service(int connfd, char *buf, int n) {
    int id, amount;

    switch (what_command(buf, &id, &amount)) {      // call by reference for id, amount
    case _show_: show_routine(connfd); break;
    case _buy_: buy_routine(connfd, id, amount); break;
    case _sell_: sell_routine(connfd, id, amount); break;
    case _exit_: exit_routine(connfd); break;
    case _error_: error_routine(connfd); break;
    }
}
```

전체 서비스를 관리하는 service 함수이다. 먼저 Client로부터 받은 명령 요청을 what_command 함수로 분석한다. 본 프로젝트 수행 간에 특별한 에러 입력 처리는 필요가 없으므로, what_command 함수 내부에서는 sscanf를 활용해 간단하게 명령 종류를 분석한다. 이때, sell이나 buy 명령의 경우, service 함수의 지역 변수 id, amount에 대한 Call by Reference를 what_command에 적용해 업데이트한다.

```
for (int i = 0; i < print_size; i++) {
    char s1[128], s2[128], s3[128];

    P(&(print[i]->mutex));           // mutual exclusion for the present item
    (print[i]->readcnt)++;           // increment the number of readers
    if (print[i]->readcnt == 1)       // if there's at least one reader, then
        P(&(print[i]->w));          // blocking every writer!
    V(&(print[i]->mutex));

    rio_itoa(print[i]->ID, s1, 10);
    rio_itoa(print[i]->left_stock, s2, 10);    // transform integer into string
    rio_itoa(print[i]->price, s3, 10);

    strcat(printbuf, s1); strcat(printbuf, " ");
    strcat(printbuf, s2); strcat(printbuf, " ");
    strcat(printbuf, s3); strcat(printbuf, "\n");

    P(&(print[i]->mutex));
    (print[i]->readcnt)--;
    if (print[i]->readcnt == 0)        // allow writers to do their tasks only if
        V(&(print[i]->w));          // there's no readers!
    V(&(print[i]->mutex));
}
```

위의 코드는 show 명령 처리 과정이다. 최소 하나의 Reader라도 있으면, 해당 Stock Item에 대한 w Semaphore에 대해 P Operation을 적용해 Writer가 아무런 일을 할 수 없도록 대기시킨다. 이를 readcnt라는 변수를 통해 조절하고 있음에 주목하자. 만약 readcnt가 0이면, w에 대한 V Operation을 통해 Writer의 접근을 허용하고 있음을 주목하라.

한편, rio_itoa라는 함수는 Integer를 String으로 바꾸는 함수로, 실제 csapp 라이브러리에는 없는, 본인이 직접 설계한 함수이다. 일반적인 itoa 와 그 기능은 동일하다.

```
/* Routine for 'buy' service (routine of 'Writer 1') */
void buy_routine(int connfd, int id, int amount) {
    Item *temp = SearchTree(root, id);
    char *buy_msg;

    P(&(temp->w));                                // mutual exclusion for the present item
    if (temp->left_stock < amount)                // only one writer can access at one time
        buy_msg = buy_error_msg;
    else {
        temp->left_stock -= amount;               // update the left_stock
        buy_msg = buy_success_msg;
    }
    V(&(temp->w));

    Rio_writen(connfd, buy_msg, MAXLINE);
}

/* Routine for 'sell' service (routine for 'Writer 2') */
void sell_routine(int connfd, int id, int amount) {
    Item *temp = SearchTree(root, id);

    P(&(temp->w));                                // mutual exclusion for 'writer'
    temp->left_stock += amount;                   // update the left_stock
    V(&(temp->w));

    Rio_writen(connfd, sell_success_msg, MAXLINE);
}
```

Write에 해당하는 sell과 buy 명령은 w에 대한 Mutual Exclusion 과정이 핵심이다. 위에서 Critical Section을 P와 V로 둘러싼 코드를 보면 된다. 이때, Rio_writen은 Critical Section에 포함시키지 않았다. 굳이 Client에 대한 결과 전송까지 직렬화할 필요는 없다고 느꼈다.

exit_routine 함수의 경우, Client로부터 "exit"이라는 문자열을 받는 경우 호출되는데, 그대로 "exit"을 반사한다. 즉, Client는 Server가 "exit"이란 결과를 보내면, 스스로 종료하도록 설계되면 될 것이다.

(5) Auxiliary Programs for Task3 Experiment

B번 항목에서 설명한 'stockgen' 프로그램과 'exec100' 프로그램 코드는 아래와 같다. 'stockgen' 프로그램은 첫 번째 실험을 위한 '다량의 주식 종목 (오름차순 번호) 생성 프로그램'이며, 'exec100' 프로그램은 두, 세 번째 실험을 위한 '100번 연속 동일 실험 수행 프로그램'이다. 실험의 정확성을 제고하기 위한 프로그램이다.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    FILE *fp = fopen("stock.txt", "wt");
    int n = atoi(argv[1]), i;

    if (argc != 2)
        return -1;

    for (i = 1; i <= n; i++)
        fprintf(fp, "%d %d %d\n", i, rand() % 11 + 1, 10000);

    fclose(fp);
    return 0;
}
```

stockgen 프로그램

```
int main(int argc, char *argv[]) {
    char str[200] = "./multiclient 172.30.10.11 60040 ";
    FILE *fp1 = fopen("experiment.txt", "wt"), *fp2;
    long long sum = 0, temp;
    strcat(str, argv[1]);

    fprintf(fp1, "");
    fclose(fp1);

    printf("Execute 'multiclient program' 100 times sequentially!\n");
    printf("-----\n");

    for (int i = 0; i < 100; i++) {
        system(str);
    }

    printf("-----\n");
    printf("----- Execution Ends! -----\n");
    printf("-----\n");

    fp2 = fopen("experiment.txt", "rt");
    while (fscanf(fp2, "%lld", &temp) != EOF) {
        sum += temp;
    }
    printf("\nThe average throughput is %lld\n", sum / (long long)100);
}

return 0;
}
```

exec100 프로그램

ex) "exec100 10" : "./multiclient 172.30.10.11 60040 10"을 100번 순차 수행

3. 구현 결과

2번 항목에서 구현한 'Event-based Concurrent Server'와 'Thread-based Concurrent Server'는 모두 정상적으로 Concurrency Issue 없이 Multiple Clients의 수많은 명령을 정확하게 처리하고 있다(이들이 Concurrency Issue를 보이지 않음을 증명하는 방법은 여러 가지일 수 있지만, 본인은 강의 시간에 배운 Shared Variable Increment 방식으로 검증하였다. 이에 대한 자세한 내용은 4번 항목에서 간단하게 소개한다). 모든 개발 과정은 2017 Visual Studio로 '선 코드 작성' 후, CSPRO로 코드를 옮겨 검증하는 방식으로 진행되었다.

프로젝트에서 요구하는 부분은 모두 구현하였다. 학기가 종료된 후, 추가적으로 구현해보고 싶은 사항들이 있었는데, 그들은 다음과 같다.

- 실제 주식 거래 플랫폼에서 주식을 사고 팔듯이, 좀 더 Real-World에 가깝게 서버 기능을 확장. 예를 들어, 현재 User의 잔고를 고려한다던가, Stock Item의 가격 변동이 일어난다던가 하는 식.
- Stock Item의 Attribute를 더 늘린 후, 실제 주식 거래소의 데이터를 긁어와(크롤링) 이 프로젝트 서버에 적용해보기. Python 코드를 연계해야 할 것이다. 다만, 이는 주식 거래소 사이트의 크롤링 관련 정책을 우선 확인해야 할 것이다.
- 모든 에러 입력을 고려한, 안정적인 서버 프로그램을 구축. what_command 함수에서 다양한 if문을 설치해야 한다.
- 특정 종목의 최근 동향을 알려주는 시스템 구축. 이를 위해선 Transaction에 대한 Log를 기록해야 할 것이다.

4. 성능 평가 결과 (Task 3)

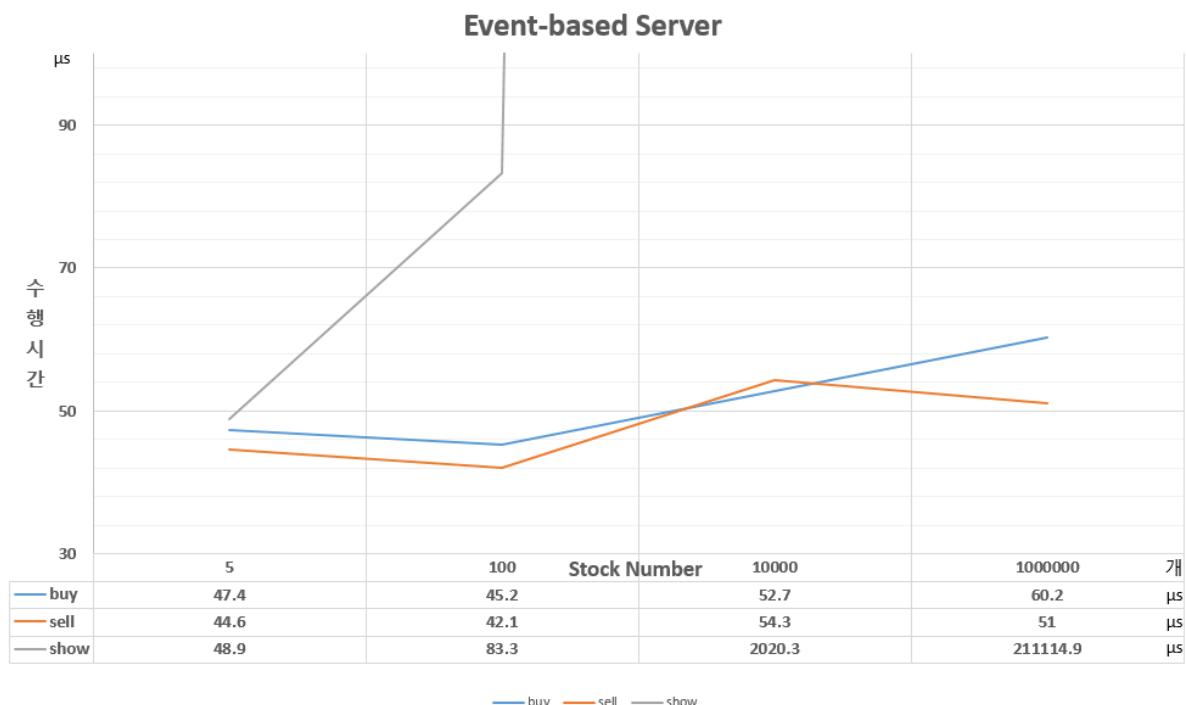
(1) 첫 번째 실험 : AVL Tree 효율성 검증 실험

2번 항목의 B 항목에서 설명한 첫 번째 번외 실험인 'AVL 효율성 검증 실험'에 대해 설명한다. 각 단일 명령이, 주어진 Stock 개수에 대해 얼만큼의 수행 시간을 보이는지를 stockserver 프로그램의 service 루틴 앞 뒤로 시간을 측정해 확인한다.

기본적인 시간 측정은 Event-based Concurrent Server를 기준으로 수행하였다. 본 실험

자체가 단일 명령을 대상으로 진행되기 때문에 두 서버 간에 유의미한 수행 속도 차이가 발생하지 않는다(실제 측정 시 거의 동일하다). 물론, Thread-based Concurrent Server에서는 Synchronization Overhead가 존재하지만, 단일 명령이기 때문에 그 정도가 극미해, 데이터 상에 나타나지 않는다. 따라서, Thread-based Server에 대한 첫 번째 실험은 생략 한다. 이하는 실험 과정에 대한 캡처 이미지이다.

각 명령에 대해 총 10번씩 수행 시간을 측정하였으며, 아래의 그래프는 해당 측정 값들의 평균치를 기준으로 작성하였다. 본 실험의 존재 자체가 본 프로젝트의 메인 실험인 Task 3 '두 번째 실험'을 위한 Sub Experiment이기 때문에, 측정값의 정확도를 위한 다양한 실험 과정이 불필요하다 판단하여 각각 10번씩만 측정하였다.



측정 시간의 단위는 마이크로 초이다. (gettimeofday 함수 사용)

결과는 예상대로 나왔다.

- 1) sell, buy 명령을 보면, Stock Item이 1,000,000개여도 5개일 때와 큰 차이가 없는 수행 시간을 보였다. 일반적인 Binary Search Tree가 순차적인 데이터 입력에 대해 'Array화 위험성'을 가지기 때문에 본 프로젝트 수행 간에 AVL Tree를 구축한 것이었는데, 그러한 목적이 정확히 실현되었다. 데이터가 아무리 많이, 순차적으로 입력되어도 균형 잡힌 이진 트리가 구축되기 때문에 주식 종목이 아무리 많아도 효율적인 수행 속도를 보인다.

이는, 이어지는 두 번째, 세 번째 실험에서 sell, buy 명령이 종목 개수와 상관 없이 효율적으로 처리될 것임을 보장해준다. 즉, 두 서버의 성능 비교 간에 'sell or buy Service의 비효율성으로 인한 데이터 왜곡'은 없음을 시사하는 것이다.

- 2) 한편, show 명령은 예상대로 sell이나 buy에 비해 비교 불가 수준으로 수행이 오래 걸림을 확인할 수 있다. 특히 종목수가 100개만 넘어가도 sell, buy의 두 배 이상의 수행 속도를 보인다. show는 Linear Traversal을 거치기 때문에 아무리 빨라도 $O(\log N)$ 의 sell / buy 명령에 비하면 현저히 느린 것이다.

Stock 개수가 5개일 때, sell, buy 명령과 유사한 수준의 수행 속도를 보였는데, 이 점에서 우리는 두 번째 실험에서 Stock 개수를 5개로 설정하면 'show와 sell, buy 명령이 Random하게 요청되는 상황', 또는 'show 명령만 요청되는 상황'에서, 보다 균형 잡힌 '동시처리율'을 얻을 수 있음을 알 수 있다. 특히, Random 명령 상황에서는 매 실험마다 show가 얼마나 출력될지는 순전히 운에 따라 달라지는 데, Stock 개수를 5개로 설정하면, 운에 의해 동시처리율이 왜곡되는 현상을 방지 할 수 있음을 예상할 수 있다. 따라서, 이를 두 번째 실험에서 반영한다.

결과적으로, 첫 번째 실험을 통해, 두 서버의 기반 자료구조 AVL Tree가 매우 효율적이고, Stock 개수가 5개인 경우 sell, buy, show 명령이 모두 유사한 수준의 수행 속도를 보임을 알 수 있는 것이다.

(2) 두 번째 실험 : 확장성과 워크로드에 따른 서버 동시처리율 측정 실험

Task3의 메인 실험이다. 2번 항목의 B번 항목에서 설명한대로 실험을 수행하였다. 앞서 설명한 것처럼, multiclient.c 소스 코드에서 시간 측정을 진행했다. 아래의 코드를 보자.

```

timer_start_flag = mmap(NULL, sizeof *timer_start_flag, PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
start = mmap(NULL, sizeof *start, PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
end = mmap(NULL, sizeof *end, PROT_READ | PROT_WRITE,
    MAP_SHARED | MAP_ANONYMOUS, -1, 0);
*timer_start_flag = 0;

```

mmap을 이용하여, timer_start_flag(첫 번째 명령 요청을 식별하는 플래그), start(첫 번째 명령 요청 시점을 기록하는 timeval type의 변수), end(서비스가 완료된 시점을 기록하는 timeval type의 변수)라는 변수들을 각 Client Process에서 공유할 수 있도록 설정한다.

```

        strcat(buf, "\n");
        sprintf(tmp, "%d", num_to_sell);
        strcat(buf, tmp);
        strcat(buf, "\n");
    }
    //strcpy(buf, "buy 1 2\n");
    if (*timer_start_flag == 0) {
        *timer_start_flag = 1;
        gettimeofday(start, NULL);
    }

    Rio_writen(clientfd, buf, strlen(buf));
    Rio_readnb(&rio, buf, MAXLINE);
    //Fputs(buf, stdout);

    //usleep(1000000);
}

Close(clientfd);
exit(0);
}
/* parten process */
/*else{
    for(i=0;i<num_client;i++){
        waitpid(pids[i], &status, 0);
    }
}*/
runprocess++;
}
for (i = 0; i < num_client; i++) {
    waitpid(pids[i], &status, 0);
}
gettimeofday(end, NULL);

/*printf("\n\nServer with %d clients, time spent: %0.8f sec\n",
    num_client, time_diff(start, end));
printf("Total Processed Requests : %d\n", total_orders);
printf("The Throughput Measured : %.0f\n",
    (float)(total_orders) / (float)time_diff(start, end));*/
fprintf(fp, "%.0f\n", (float)(total_orders) / (float)time_diff(start, end));
fclose(fp);

```

좌측 코드를 보자. Flag는 최초에 0으로 초기화되어 있다. mmap을 통해 이 플래그를 모든 Client Process에서 Visible하도록 만들어준다.

특정 Process가 최초로 명령을 서버에 요청할 것이며, 이때, 이 Flag가 0임을 파악하여 시간 측정이 시작된다. 시간 측정이 시작되면, 플래그를 1로 설정하여 더 이상의 시작 시간 측정이 이뤄지지 않도록 만들어준다 (MS_SYNC를 적용했다).

이후, 모든 서비스가 완료되면, Client Process Reaping 이후, 시간 측정

을 마무리한다. B 항목에서 설명한 것처럼, Reaping의 Overhead는 경미하다. 기본적으로 Task3 수행 과정에서 프로세스를 최대 100개까지만 생성할 것이기 때문이다(실제 이 Overhead를 따로 측정했는데, 역시나 예상대로 매우 경미한 수준이었다). 한편, Iteration 내에서 if문으로 Flag를 체크하는 과정 역시 Overhead가 매우 경미하다(오히려, 서버 단에서 시간을 측정할 때 발생하는 Counting의 Overhead가 더 크다).

위의 코드에서 하단에 주석으로 처리된 print문들은 실험 설계 과정에서 임시로 두었던 루틴이다. 코드에서 알 수 있듯이, 프로그램이 자체적으로 수행 시간을 측정 후, 측정한 시간과 총 명령 개수를 이용해 자체적으로 동시처리율을 계산한다. 변형 multichannel 프로그램이 자동으로 계산한 동시처리율을 특정 파일(experiment.txt)에 출력한다. 2번 항목의 C 항목에서 소개한 exec100 프로그램이 이러한 multichannel 프로그램을 100번 연속 순차적으로 수행한 후, experiment.txt 파일에 기록된 100개의 동시처리율 측정값을 평균내어 최종 결과를 알려주는 것이다(참고로, 기존 multichannel의 1초 대기 루틴도 제거).

이러한 일련의 과정을 거쳐서 실험이 진행되었다. 2번 항목의 B 항목과 C 항목 설명에서도 자세히 소개한 바 있다. 본인이 이 '실험 설계 과정'을 철저하게 기술하는 이유는, 실험이 실제로 진행되었음을 입증하기 위함이다. 실제 수행 시간 측정 및 동시처리율 계산이, 여러 번 검증을 거친 (변형) multiclient 프로그램과 exec100 프로그램을 통해 자동으로 진행되었음을 재차 강조하는 것이다. 아래는 실제 실험 과정 캡쳐 이미지들이다.

(Event-based Server에 대한 Random 명령 처리 상황의 실험 과정 캡쳐 이미지)

(Event-based Server에 대한 Sell & Buy 명령 처리 상황의 실험 과정 캡쳐 이미지)

```

cse201716438cspc08:/sp2/exp/th/no25 ./exec100 1
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 3247
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 5
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 8290
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 10
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 9681
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 20
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 12058
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 50
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 10020
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 50
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 13384
cse201716438cspc08:/sp2/exp/th/no25

```

실험 중 네트워크 불량 발생하여 폐기한 실험

(Thread-based Server에 대한 show 명령 처리 상황의 실험 과정 캡쳐 이미지)

```

cse201716438cspc08:/sp2/exp/th/no25 ./exec100 1
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 1722
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 5
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 5492
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 10
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 8225
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 20
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 10754
cse201716438cspc08:/sp2/exp/th/no25 ./exec100 50
Execute 'multiclient program' 100 times sequentially!
-----
Execution Ends!! -----
The average throughput is 12388
cse201716438cspc08:/sp2/exp/th/no25

```

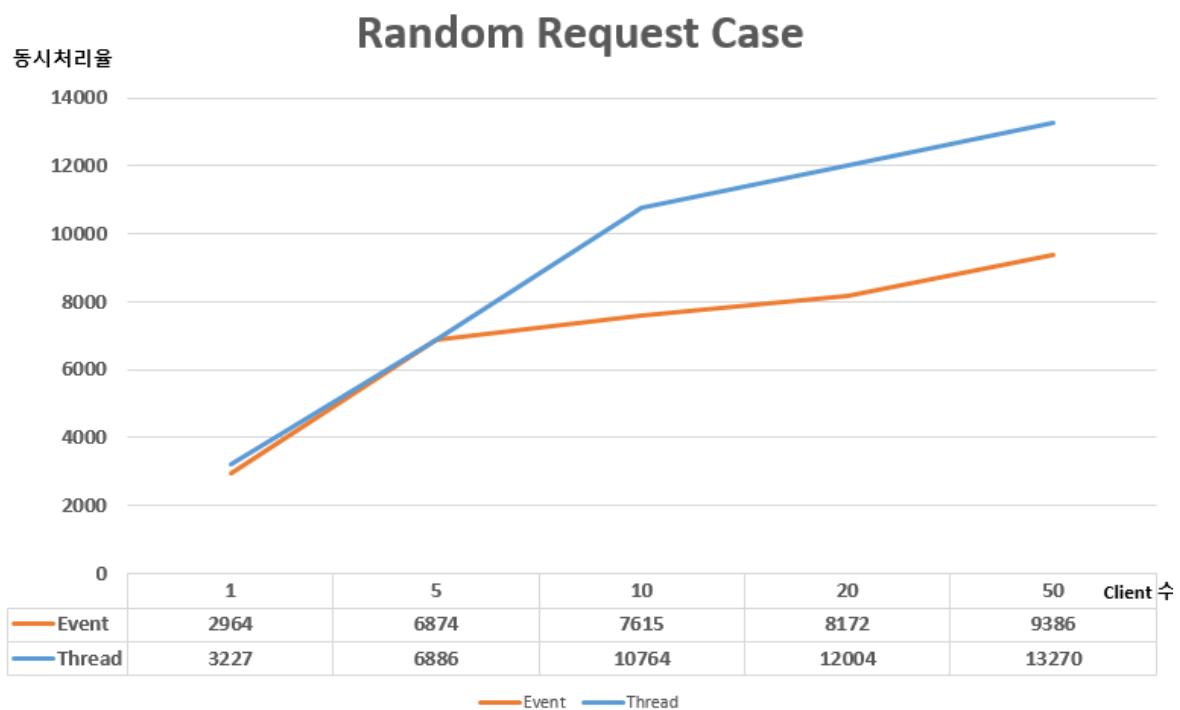
(Thread-based Server에 대한 show 명령 처리 상황의 재-실험 과정 캡쳐 이미지. 재-실험 이유는 후술한다)

실험 과정 증명은 여기까지 한다. 이제 두 번째 실험의 결과 및 결과 해석을 진행한다. 참고로, Thread-based Server의 Worker Thread는 1000개로 두었다. 실제로 Shared Buffer 와 Worker Threads를 다양한 개수로 마련하여 간이 실험을 진행해보았는데, 특별히 이들의 개수와 서버 성능의 연관성이 높지 않다는 점을 발견하였다. 즉, Worker Threads를 많

이 두든 적게 두든, Shared Buffer를 크게 두든 적게 두든, 100개 이상만 두면 Thread-based Server의 성능에 큰 차이가 없던 것이다. 본 실험에서 취급하는 클라이언트 개수가 그리 많지 않아서 이러지 않을까 생각한다. 따라서, 굳이 Worker Threads를 적게 둘 필요는 없을 것이라 판단해 Shared Buffer의 크기를 1000으로, Worker Threads의 개수도 1000으로 두었다. Thread-based Server의 성능을 최대한 끌어올리고자 함이다.

1) Case1 : 복수의 Client가 Random 명령 요청을 동시에 20개씩 전송하는 상황

아래의 그래프는 클라이언트 개수에 따른 Random 명령에 대한 두 서버의 동시 처리율 비교 그래프이다.

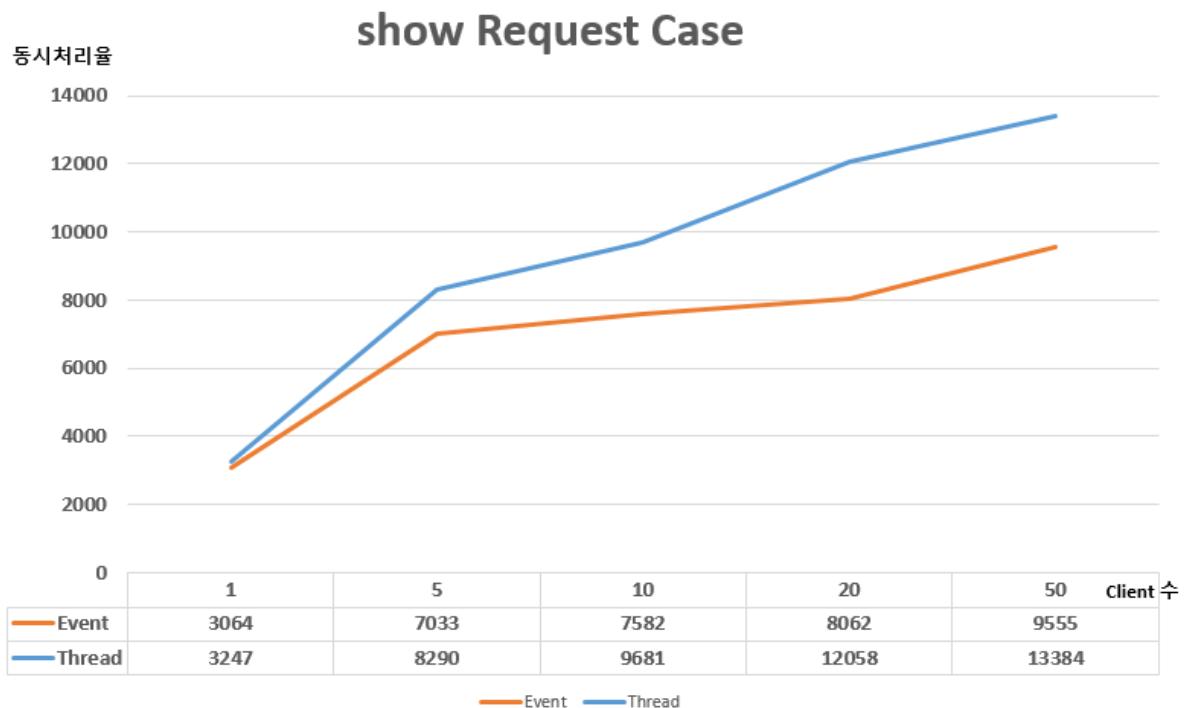


~> **Case1 실험 결과 해석** : 우선 예상대로 Client 개수가 많아질수록 두 서버 모두 동시 처리율이 올라가고 있다. 이때, Client 개수가 10개 미만일 때는 두 서버의 성능 차이가 크지 않음을 알 수 있다. 하지만 Client 개수가 10개인 순간을 기점으로 모든 Client 개수에 대해 약 3000~4000정도의 동시처리율 차이가 발생했다.

기존 예상에서, Thread-based Server는 Multicore의 이점을 충분히 활용할 수 있어 Event-based보다 성능이 좋을 것이지만, Thread-based Server에는 Synchronization을 위한 Overhead가 있으므로 그 차이가 크진 않을 것이라 판단했는데, 실험 결과를 보면 예상보다 그 차이가 훨씬 크다는 것을 확인했다. 이어서 Case2에 대한 결과를 보자.

2) Case2 : 복수의 Client가 show 명령 요청을 동시에 20개씩 전송하는 상황

아래의 그래프는 클라이언트 개수에 따른 show 명령에 대한 두 서버의 동시 처리율 비교 그래프이다.



~> **Case2 실험 결과 해석** : 역시나 Random 명령 상황과 마찬가지로 Thread-based Server가 Event-based Server에 대해 뚜렷한 동시처리율 우위를 보이고 있다. 하지만, 이 실험 결과는 2번 항목에서 언급한 실험 예상을 제대로 확인하지 못한 실험이었다.

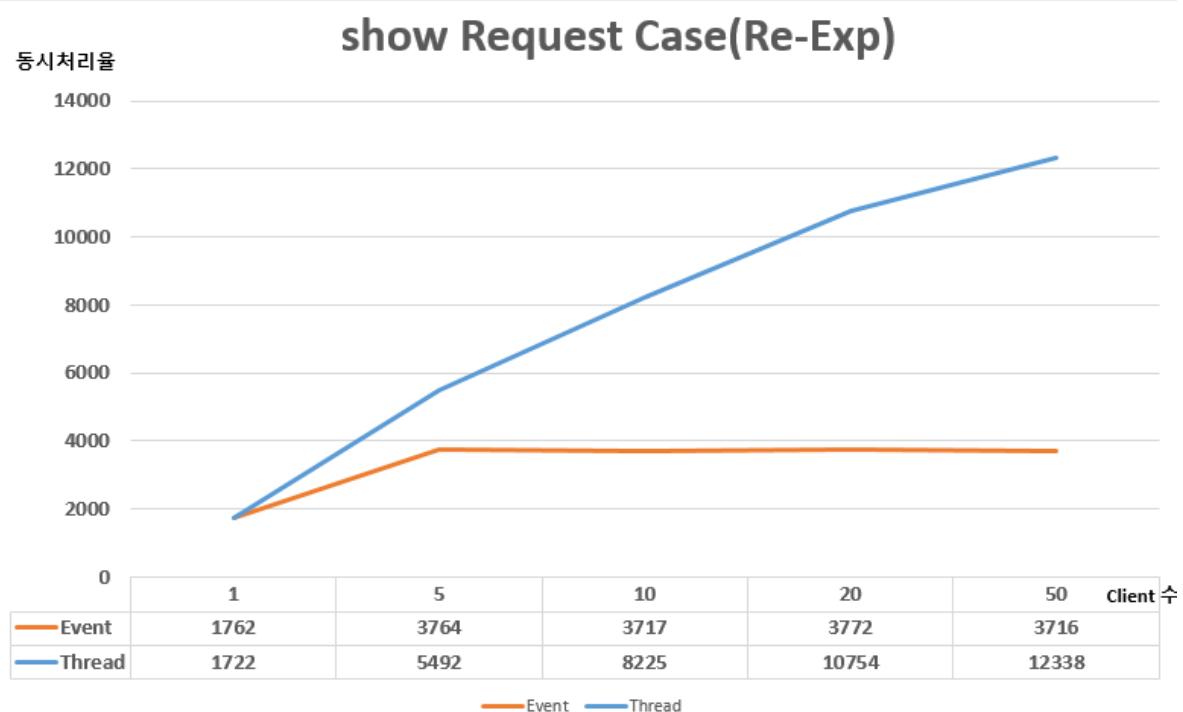
본인은 show 명령의 경우, Event-based Server는 반드시 순차적으로 앞선 Client의 show Request가 마무리되어야 이후 Client의 show Request가 수행될 수 있는데, 그에 반해 Thread-based Concurrent Server에서는 여러 Client가 동일 Stock Item 조회 상황을 제외하고는 모두 동시에 Request Handling이 가능하므로 Thread-based Server가 훨씬 더 빠른 성능을 보일 것이라 예상했었다.

하지만, 위의 그래프를 보다시피 그러한 결과를 내지 못한 것이다(Random Case에서의 두 서버 성능 차이와 비슷하므로). 이에 대한 이유를 추론해보면 다음과 같았다.

- 첫 번째 실험의 결과로서, 본인은 show 명령의 처리 시간이 sell과 buy에 비해 현저히 오래 걸리므로 Stock Item 개수를 5개로 설정하여 show와 sell / buy 간에 큰 차이가 발생하지 않도록 조정했다고 했다. 하지만, 이것이 오히려 Case2 실험에서 의도를 벗어나는 결과가 나오도록 야기한 것이다.

- show Request 처리 시간 자체가, Stock Item 개수가 너무 적어 매우 빨리 걸리는 것이다. 즉, Event-based Server의 단점을 명확히 보여주지 못하는 것이다. 기본적으로 show 명령 수행 시간 자체가 너무 작다 보니 말이다.

따라서, 본인은 이 Case2에 한해서만 Stock Item 개수를 늘려야겠다고 판단했다. 트리 순회 자체가 어느 정도 적당한 시간을 소요해야 Event-based Server의 단점이 명확히 보일 것이라 판단한 것이다. 따라서, 일전에 언급한 바 있는 stockgen 프로그램을 이용해 Stock Item이 1000개가 있는 stock.txt 파일을 새로 만들어 Case2에 대해서만 다시 재-실험을 진행했다. 그 결과는 아래와 같다.



~> **Case2 재-실험 결과 해석** : 다행히 재-실험 결과, 기존의 예상처럼, Stock Item 개수가 많은 경우, Thread-based Server의 성능이 Event-based Server에 대해 show 명령에 한해서 훨씬 더 우수함을 확인할 수 있었다. 예상보다 더 압도적인 차이를 보였다.

Thread-based Concurrent Server의 경우, 여러 클라이언트가 동시적으로 Tree를 순회하면서, 오로지 '현재 조회(Read/Write) 중인 Node'가 겹칠 때에만 'First Readers-Writers Problem'에 의거해 Synchronization이 이뤄지는데, 이 Case에서는 Write 명령이 없으므로 사실상 모든 클라이언트가 (거의) 동시에 일을 수행하는 형태가 되어, 아무리 Linear Traversal을 한다고 해도 기본적으로 동시처리율 자체가 매우 우수하게 나온 것이다.

반면, Event-based Server의 경우, 무조건 앞선 Client의 show Request가 처리 완료되어야 이후 Client의 show Request가 처리되고, 이는 결국 '트리에 대한 접근' 자체가 각

Client에 대해 무조건 배타적으로, 직렬적으로 이루어지는 것을 의미한다. 즉, 사실상 전체 Stock에 대한 중첩 Loop 문 같은 성능을 보이게 되는 것이다.

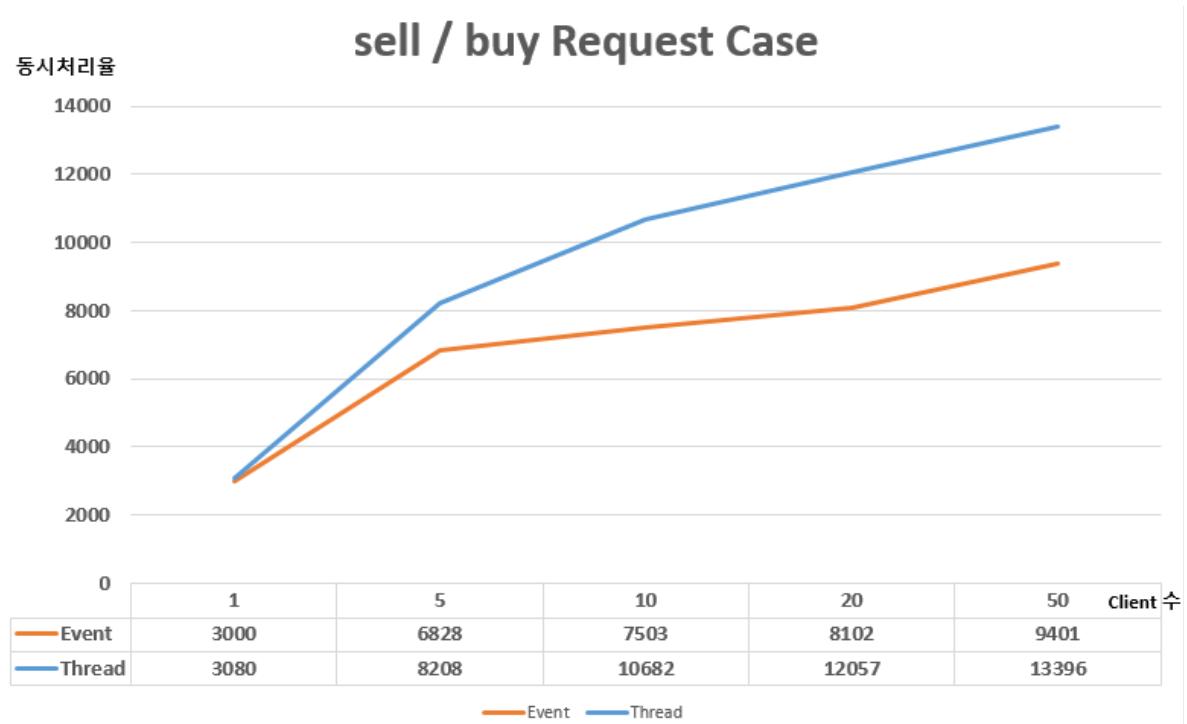
이러한 실험 결과에서 우리는 다음과 같은 사실도 생각해볼 수 있다.

"만일, Thread-based Server 개발 과정에서 'First Readers-Writers Problem'을 적용하지 않고, 단순하게 service 자체에 대한 Mutual Exclusion을 제공했으면 마찬가지로 'Tree에 대한 접근' 자체가 직렬화되어, 이 실험 결과처럼 Thread-based Server도 show Request에 대해 좋지 않은 성능을 보였을 것이다."

한편, Stock Item 개수가 많아서 show 명령 자체가 이전 Case에 비해 오래 걸리고, 그에 따라 전반적인 동시처리율 수치도 약간 낮아졌음도 주목하자.

3) Case3 : 복수의 Client가 sell 또는 buy 명령 요청을 동시에 20개씩 전송하는 상황

아래의 그래프는 클라이언트 개수에 따른 sell or buy 명령에 대한 두 서버의 동시 처리율 비교 그래프이다(buy가 실패하지 않도록 stock.txt의 각 Stock Item의 초기 left_stock 값을 매우 크게 설정하였기 때문에 sell과 buy의 수행 복잡성 차이가 사실상 없고, 따라서 sell과 buy를 굳이 따로 분리해서 Case화하지 않았다).



~> **Case3** 실험 결과 해석 : buy / sell의 경우 Random 명령 Case와 유사한 형태의 결과가 도출되었다. 역시나 Thread-based Server의 동시처리율이 더 높게 측정되었다.

실험을 설계하던 시기에는 Thread-based Server의 Semaphore Synchronization이 있기 때문에 Thread-based Server의 동시처리율 우위에 어느 정도 조정이 이뤄질 것이라고 생각하였다. 하지만 세 Case에 대한 실험 결과를 보면, 예상보다 Synchronization에 의한 Overhead는 경미하고, Fine-Grained Concurrency의 효과는 더 컸다.

좀 더 파고들어 보면, 이것만이 이유는 아닐 것이다. 2번 항목에서 'I/O Multiplexing' 간에 select 함수보단 epoll 함수가 더 우수하다'는 사실을 확인하였는데, 이러한 점도 결과에 영향을 미쳤을 것이다. epoll을 사용한다면 Event-based Server의 성능이 조금 더 개선될 것이다.

하지만, epoll을 사용하더라도 show Request에 대해선 Event-based Server는 어쩔 수 없이 그 근본 구조에 의해 좋은 성능을 보일 순 없다고 예상한다. select보다 더 효율적인 epoll을 사용하더라도 결국 근본 구조 자체는 (사실상) Iterative한 Event Handling임에는 변함이 없기 때문이다. 여기서 본인은 다음과 같은 잠정 결론을 내렸다.

"Event-based Concurrent Server는 Thread-based Server에 비해 시공간적 Overhead는 분명히 더 적고, 디버깅도 쉽지만, (본 프로젝트의 show 명령처럼) 커다란 자료구조를 순회하는 식의, '기본적으로 단일 Request 수행 자체가 오래 걸리는 Service'가 잦게 발생하는 상황에서는 적합하지 않을 것이다."

한편, 본 실험에서는 네트워크 통신 환경의 개입 가능성이 완벽하게 배제되지는 않는다. 실제로 Case2 실험 간에는 실험 중간에 네트워크 이상이 발생해 CSPRO 서버가 잠시 (아주 짧게) Hanging한 순간도 있었다. 따라서, 본 두 번째 실험의 결과가 완벽히 정확하다고 보장할 순 없다. 하지만, 그럼에도 앞서 지속적으로 소개한 것처럼, exec100 프로그램의 도입, 여러 차례의 간이 실험, 서버 수행 시간 측정 기준의 철저한 검증 등을 수행 하였기 때문에 최대한 결과 객관성을 높였다고 판단한다. 따라서 'Client 수가 많아질수록 Thread-based Server의 성능이 우수함'이란 경향성 자체는 의미가 크다고 생각한다.

(3) 세 번째 실험 : Thread-based Server에서 동일 Shared Variable에 대한 Write 명령이 몰릴 경우(Client 수가 늘어날 경우), 서버 동시처리율이 오히려 떨어지기 시작하는 지점이 있는지 확인하는 실험

```

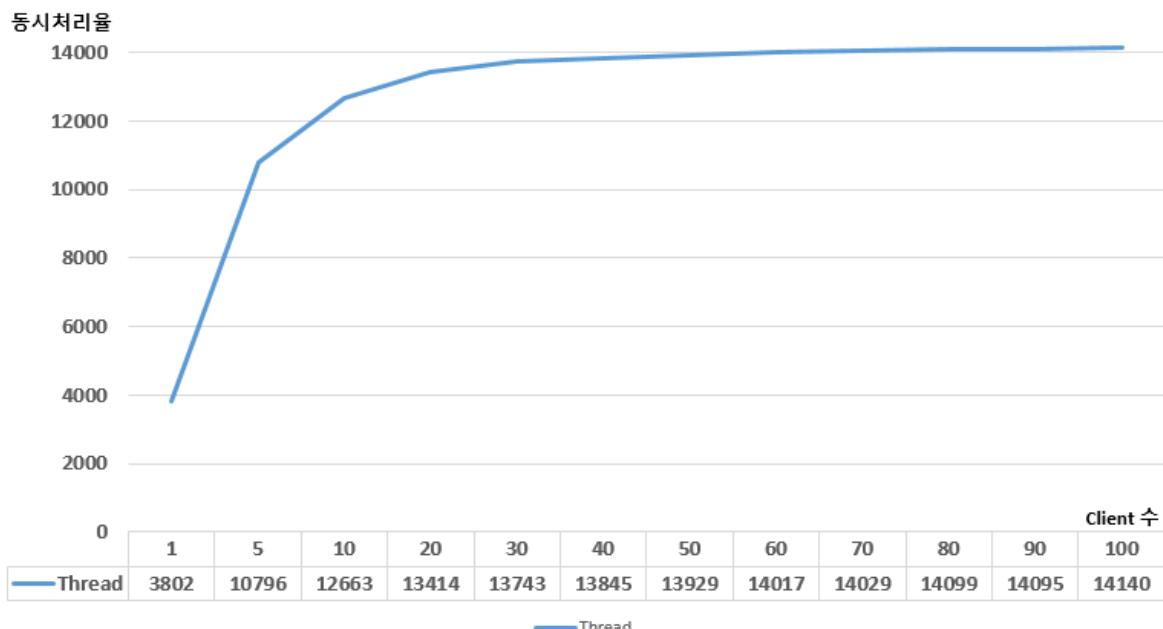
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 13414
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 30
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 13743
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 40
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 13845
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 50
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 13929
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 60
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 14017
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 70
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 14029
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 80
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 14099
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 90
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 14095
cse20171643@cspro8:~/sp2/exp/th/no3$ ./exec100 100
Execute 'multiclient program' 100 times sequentially!
----- Execution Ends!! -----
The average throughput is 14140
cse20171643@cspro8:~/sp2/exp/th/no3$ 

```

Visual Studio Code interface showing the source code for 'multiclient.c'. The code implements a client for a stock trading system, handling sell and buy requests. It uses a thread pool and socket communication.

위는 세 번째 실험 장면 캡쳐 이미지이며, 아래는 실험 결과 그래프이다.

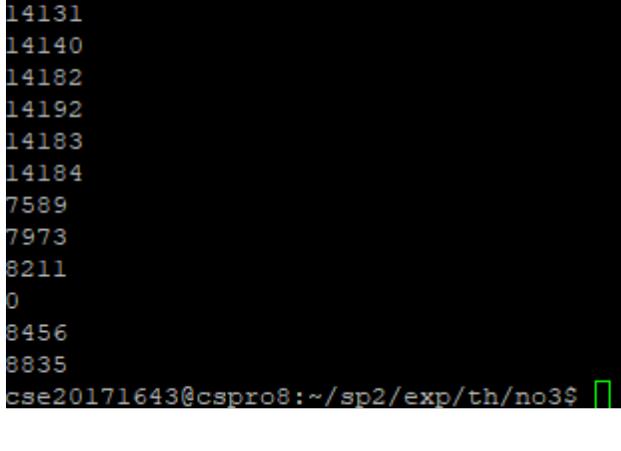
Thread-based Server



~> **실험 결과 해석** : 예상과 다르게, 명확하게 동시처리율이 감소하는 지점은 발견되지 않았다. 동시처리율 14100 정도 부근에서 계속 머무는 식의 Threshold는 발견되었다.

하지만, 이 Threshold가 과연 Thread-based Concurrent Server에서 Write 명령이 몰릴 시에 발견되는 Threshold일까?

- 우선, “**Worker Threads 개수에 따라 이 수치가 변할 수 있지 않을까?**”라는 의문이 들었다. 따라서, Worker Threads와 Shared Buffer Size를 둘 다 10000으로 변경해서 Client가 100개인 Case에 대해서만 추가 간이 실험을 해보았다. 결과는, 동시처리율 수치가 위에서 명시한 14140과 크게 다르지 않았다(13090).
 - ✓ *Worker Threads 개수(및 Shared Buffer 크기)는 영향을 주지 않았다.*
- 그렇다면, “**Client 개수가 더 많아지면, 수치가 꺾이는 지점이 발견되지 않을까?**”라는 생각을 해보았다. Client가 늘어나면 ‘꺾이는 지점’이 명시적으로 나타날 수 있지 않을까?라 생각했다. 이러한 생각에 기반해 multiclient.c의 MAX_CLIENT 제한을 200으로 변경하여 Client 개수가 110, 120, 130, 140, 150, 160, 170, 180, 190, 200개일 때를 기준으로 각각 ‘한 차례’씩 실험을 더 수행하였다.
- ‘한 차례’만 수행하였다는 것은, exec100을 이용하지 않고, “./multiclient 172.30.10.11 60040 N” 명령을 각 N에 대해 한 차례씩만 수행해 동시처리율을 계산했다는 것이다. 앞서 언급한 것처럼, 본인의 변형 multiclient 프로그램은 한 차례 수행 결과에 대해 experiment.txt 파일에 업데이트를 진행한다. 실험 수행으로 기록된 experiment.txt 파일의 내용은 우측 캡쳐 이미지와 같다.
- 위에서부터 순서대로 Client 개수가 110개, 120개, ..., 200개일 때의 동시처리율이고, 중간의 0이란 숫자는 Shell 상에서 command를 잘 못 입력하여 만들어진 결과이다.
- 클라이언트 개수 110개부터는 한 차례씩만 실험을 진행한 이유는 CSPRO 서버에 exec100으로 인한 과부하를 주지 않기 위해서이다. 따라서, 실험 정확성은 떨어진다는 점을 확실히 해두겠다. 하지만, 한 차례만 실험하더라도 대체적으로 평균

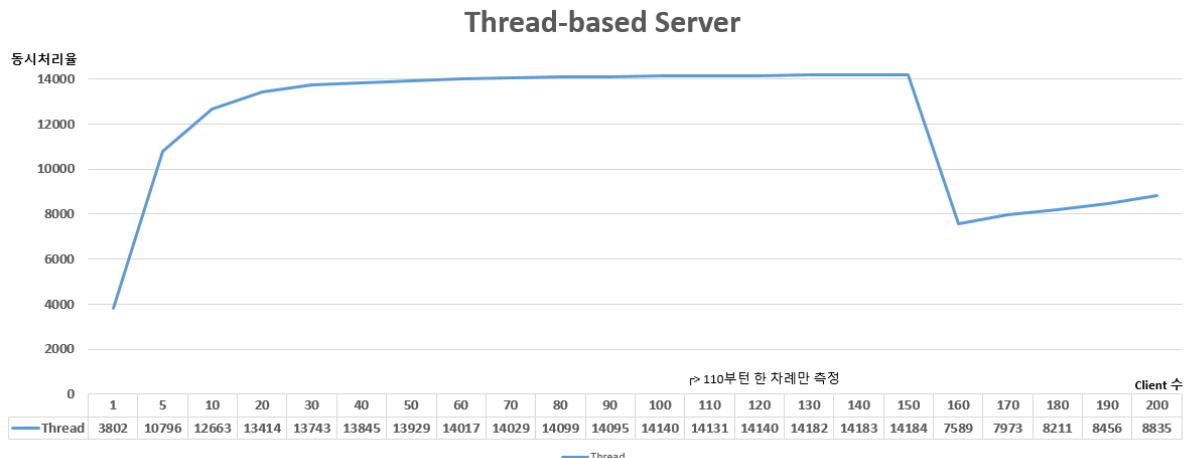


14131
14140
14182
14192
14183
14184
7589
7973
8211
0
8456
8835

cse20171643@cspro8:~/sp2/exp/th/no3\$ █

값에 가까운 수치가 나오므로 어느 정도 경향은 보여줄 수 있다고 생각한다.

- '한 차례 씩 실험한 측정값'을 '기존의 실험 그래프'에 추가하면 다음과 같다. 110부터는 정확성이 떨어진다는 점을 다시 한 번 강조한다.



~> 재-실험 결과 해석 : Client 개수가 150인 순간부터 급격하게 동시처리율이 떨어짐을 확인할 수 있다. 이에 대해선 다음과 같이 두 가지 가능성을 열어두고 해석하였다.

- 1) **해석1** : Thread-based Server에 Write 명령이 몰릴 때 생기는 '성능이 꺾이는 지점'이 발견된 것이다. Worker Thread 개수는 적정 수준 이상일 때부턴 동시 처리율에 크게 영향을 주지 않으므로, 동시처리율을 떨어트릴만한 원인은 오로지 '다량의 Client가 동시에 동일한 Shared Variable에 Write 명령을 쏴서 Synchronization 과정에서 성능 저하가 발생한 것'이다.
- 2) **해석2** : 하지만, 본인이 고려하지 못한 또 다른 요소(이를 테면 네트워크 상황 또는 multichannel의 Process 수행 과정에서 예상치 못한 Overhead 발생 등)가 개입했을 가능성도 결코 무시할 수 없다. 그래프가 너무 급격하게 꺾였다는 점에서, 강의 시간에 배운 성능 저하 양상과는 약간 차이가 있다. 이 부분에서 의심이 생긴다. 해석2에 힘을 실어줄 수 있는 지표도 하나 발견하는데, Event Server에 대해서도 Client 개수를 100개 이상으로 몇 차례 간이 실험을 진행했을 때, 여기서도 동시처리율이 급격히 꺾이는 지점이 존재했다는 사실이다(140개부터 꺾였다).

```
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 110 12088
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 120 12805
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 130 13311
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 140 6814
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 150 7022
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 160 7253
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 170 7447
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 180 7576
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 190 7738
cse20171643@cspro8:~/sp2/exp/th/no3$ ./multiclient 172.30.10.11 60040 200 8215
cse20171643@cspro8:~/sp2/exp/th/no3
```

두 가지 해석이 있는 가운데, 현재 본인의 System Programming 지식 수준에서는 더 이상 '꺾이는 지점이 발생하는 이유'를 명확히 추론할 수 없고, 따라서 어떤 해석이 옳은지 판단할 수가 없었다는 점이 많이 아쉬웠다. 이는 추후 프로젝트 피드백, 또는 학기 종료 이후 추가 학습 등을 통해 확인해보겠다. 어떤 해석이 정답이든, 현 시점에서는 결론적으로 세 번째 실험은 성공하지 못했다고 볼 수 있다.

Task3 총 결론 : 사전에 상정한 세 가지 실험을 모두 정상적으로 마쳤으며, 결론적으로 Thread-based Concurrent Server의 성능이 본 프로젝트 맥락에서는 Event-based Server에 비해 우수함을 확인할 수 있었다. 또한 추가 번외 실험을 통해 서버의 기반 자료구조가 상당히 효율적이며, show 서비스가 sell / buy보다 비효율적임을 알 수 있었다. 마지막 세 번째 실험의 경우 실험 측정 수치 자체는 예상 결론과 어느 정도 부합했지만, 그 결과에서 Thread Program의 Threshold가 발견된 것인지에 대해선, 객관성 / 검증 과정 / 명확한 이론적 설명이 부족하므로 결론을 특정할 수 없다는 한계점이 존재한다.