System Programming Project 3

담당 교수 : 박성용 교수님

이름 : 이민석

학번 : 20201614

1. 개발 목표

- 이번 프로젝트는 여러 클라이언트가 동시에 접속하여 주식 정보를 조회하고, 주식을 사고파는 요청을 처리할 수 있는 Concurrent stock server를 구현하는 것이 목표이다. 서버는 주식 정보를 파일에 저장하고 있으며, 각 클라이언트와 통신하여, 명령을 처리할 수 있어야 한다. 클라이언트는 정해진 명령어를 통해 서버에 요청을 보내게 된다.
- 서버와 클라이언트의 통신을 위해 Socket을 사용한 Network programming 개념이 필요하며, Concurrent stock server는 두 가지 방법으로 구현하게 된다. 첫째로, 'Select()' 함수를 이용하여 각 client와 연결된 socket file descriptor를 monitioring하고, 요청을 처리하는 Event-driven Approach 방법이 있다. 둘째로, Master-Worker thread관계를 이용하며, semaphore, mutex를 이용하여 Synchronization을 해결하는 Thread-based Approach 방법이 있다. 이 때, Readers-Writers 문제 해결 방법을 적용하여 성능을 높일 수 있다.
- 서로 다른 두 Approach의 성능을 Client 요청 타입, Client 개수 변화에 따른 동시처리율 비교를 통해, 다양한 관점에서 평가하고 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   - 우선 active한 connection을 여러 connfd로 구성된 array로 관리한다. 만약 listenfd 또는 connfd에 pending input이 입력되면, 해당 클라이언트의 요청을 처리하게 된다. 'Select()'함수를 사용하여 이를 해결할 수 있으며, I/O Multiplexing을 통해 여러 client의 request들을 Asynchronized하게 처리한다.

2. Task 2: Thread-based Approach

   - 효율적인 Thread-based Approach를 사용하기 위해 Master-Worker thread 개념을 적용하였다. Master thread가 client의 connection을 받아들이고, 각 client의 연결을 'sbuf'(작업 큐)에 추가하여, worker thread가 sbuf에서 꺼내와

client의 요청을 처리하는 방식이다. 또한 Semaphore, mutex의 lock/unlock을 이용하여 여러 thread가 동시에 주식 정보를 접근하지 못하도록 하였다. Client의 request 해결에 따른 주식 정보의 변경이 잘못된 결과를 낳을 수 있기 때문이다. 그러나 여러 thread가 주식 정보를 변경하지 않고 읽기만 한다면 이를 막을 필요는 없다. 따라서 주식 정보를 여러 client의 request에 의해 read, write할 때 발생할 수 있는 synchronize 문제점을 해결하기 위해 First Readers-Writers 문제를 해결하는 코드를 적용한다. 이는 다수의 thread가 동시에 주식정보를 읽고 출력할 순 있으나, 주식정보를 쓰는 동안은 다른 thread가 읽거나 쓰지 못하게 막는 방법이다.

3. Task 3: Performance Evaluation

- Task1, 2에서 구현한 2개의 서로 다른 server의 분석 및 성능 평가를 실시한다. 제공된 'multiclient.c'를 바탕으로 Client의 수, 각 Client당 request 명령 개수, request의 종류('show'만 요청하거나 'buy','sell'만 요청하거나 3개 모두 요청)의 변화에 따른 응답 시간 차이를 분석하여 성능을 평가할 수 있다. 또한 Task 2의 Thread-based Approach server에서 NTHREAD(Worker thread)의 개수를 달리하여, thread 개수 차이에 따른 성능 차이를 확인하고자 한다. 응답 시간을 바탕으로 동시 처리율을 계산하여, 이를 성능 지표로 여겨 성능을 평가하고자 한다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

- I/O Multiplexing은 single process로 서버가 여러 client의 연결 및 request 처리를 Concurrent하게 처리하게끔 한다. 이 때, pool 구조체를 이용하여 연결을 관리하는데, socket file descriptor와 관련된 정보를 저장하고, active descriptor들로 구성된 bitmap을 관리할 수 있다. 'listenfd'는 client의 connection request를 감지하는 socket file descriptor이며, 'Select()'함수는 이를 통해 새로운 client의 연결 요청을 확인한다. 또한 client와 connection을 맺게 되면, connfd를 통해 server와 client가 통신을 하게 되며, client의

request가 있다면, 해당 connfd의 pending input을 통해 알게 된다. 따라서 'Select()'의 여러 socket file descriptor의 pending input checking을 통해 여러 client의 request들을 동시에 처리할 수 있게 된다. request문들은 'check_clients()'를 통해 처리되며, 명령('show', 'buy', 'sell')을 읽고 이에 맞게 처리하여 결과값을 다시 connfd에 결과를 적는다.

✓ epoll과의 차이점 서술

- 'epoll()'은 Linux에서 제공하는 고성능 I/O Multiplexing 구현 함수로, 'select()' 함수에 비해 더 많은 수의 file descriptor를 처리할 수 있다. 'select()'는 매번 호출될 때마다 순차적으로 모든 file descriptor set들을 검사하지만, 'epoll()'은 관찰하고자 하는 file descriptor set들을 통해 이벤트가 발생한 file descriptor만 반환하므로, 대규모 연결 및 매우 많은 file descriptor를 처리할 때 효율적이다. 또한 read, write, exceptional event만 감지하는 'select()'와 달리, 'EPOLLIN', 'EPOLLOUT' 등 다양한 event들을 감지할 수 있다. 따라서 시스템 자원을 더 효율적으로 사용하는 장점도 있다.

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

✓ Master Thread의 Connection 관리

- Master Thread는 Stockserver의 main thread이며, client의 연결을 Accept하고, client와 맺어진 socket file descriptor를 'sbuf'의 공유 버퍼(buf)에' 추가하는 작업을 한다.

Multi thread 환경으로 인해 synchronization문제 중 하나인 Producer-consumer문제가 발생할 수 있다. Producer인 Master thread는 connection이 맺어진 socket file descriptor를 버퍼에 넣어주며, Consumer인 worker thread는 버퍼에서 socket file descriptor를 하나씩 꺼내 처리하는데, 동시에 Producer와 Consumer가 버퍼에 접근할 수 있기에 문제가 발생할 수 있다. 이 때, 'sbuf'는 공유 버퍼로서 여러 개의 thread가 안전하게 connection을 통해 맺어진 socket file descriptor를 접근할 수 있도록 synchronization 제어를

하기 위해 Mutex, Semaphore 변수들을 사용한다. 버퍼의 접근을 제어하는 mutex 변수 'mutex', 빈 slot의 수를 세는 semaphore 변수 'slots', 채워진 slot의 수를 세는 'items' semaphore 변수가 있다. 이를 통해 동시에 Producer와 Consumer가 버퍼를 수정할 수 없으며, Producer는 빈 slot이 생길 때까지 기다리며, Consumer는 채워진 slot이 있을 때까지 기다린다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술(위 내용에 이어서 참고)

- Stockserver는 매우 많은 client에 대해서 각 client의 request들을 처리해야 한다. 이를 효율적으로 처리하기 위해 Worker thread를 미리 n개 생성하고, client의 요청이 있을 때마다 Worker thread가 작업을 수행할 수 있도록 하는 Worker Thread Pool을 이용한다.

Worker Thread는 위 Producer-Consumer 관계에서 Consumer에 해당하며, 생성 시 'thread()'함수를 통해 detached된 상태로 바뀌어, 'sbuf'의 공유 버퍼에서 client와 맺어진 socket file descriptor를 가져와 client의 request들을 처리한다. 모든 request들을 처리한 후, 해당 socket file descriptor를 닫아 connection을 해제한다.

- **Task3 (Performance Evaluation)**

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

- 두 개의 서로 다른 'Stockserver'의 성능을 평가하기 위해 동시 처리율 개념을 도입하였다. 동시 처리율은 시간당 client의 request를 처리하는 개수이며, 이에 따라 metric인 동시처리율을 다음과 같이 계산하였다.

동시 처리율 = { (*client request 명령 수*) x (*client 수*) } / (*수행 시간*)

이 때 수행 시간은 Multiclient에서 측정한 시간이며, '프로그램이 시작된 후 ~종료' 까지의 경과시간을 말한다. Server에서 측정하지 않고, multiclient에서 측정한 이유는, 이번 프로젝트의 개발 목표인 "여러 클라이언트가 동시에 접속하여 주식 정보를 조회하고, 주식을 사고파는 요청을 처리할 수 있는 server 구현"을 고려하였기 때문이다. 결국 수행 시간은 여러 Client가 Server

에 request를 보낸 시점부터 Server의 작업 처리를 거쳐 다시 Client로 결과가 도달하여 connection을 종료하는 시점 까지라 판단하였다. 또한 수행 시간은 각 case마다 5번의 실험을 진행하여 평균을 계산하여 사용하였다.

물론 이 과정에서 여러 client 생성을 위한 child process fork와 reaping 과정의 경과시간이 포함되어 있지만, 이에 대한 Overhead는 초(sec) 단위의 수행시간에 비해 매우 미비하다는 것을 자료조사를 통해 알게 되었다. 오히려 서버의 환경, network 환경이 중요하다는 것을 알게 되었고, 이를 위해 동일 시간대에 cspro의 접속자 수와 process 수를 확인하여 최대한 동일 조건하에 실험을 진행하고자 노력하였다. 또한 주식 ID는 1~20으로 설정하였다.

- 두 개의 서로 다른 'Stockserver'의 성능을 평가하기 위해 4개의 실험을 진행하였다. 우선 Task 3의 분석포인트인 확장성과 워크로드에 대해 살펴보았다. 확장성 분석은 Client 수 변화에 따른 서로 다른 Stockserver의 동시 처리율 성능 분석이며, Workload에 따른 분석은 Client의 request 명령 종류에 따른 동시처리율 분석이다.

첫번째 실험은 client당 request를 15개로 고정하여, Client 개수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정하여 동시처리율을 비교한다. 이 때 Client의 수를 600개 까지만 한정한 이유는, Task 1에서 사용하는 'Select()'함수의 FD_SETSIZE를 고려하였기 때문이다. 일반적으로 FD_SETSIZE는 1024로 정의되어 있으나, 실제 작동은 이보다 적은 개수의 file descriptor만을 확인하고, cspro의 환경에 따라 받아들일 수 있는 client의 수가 조금씩 다름을 사전 모의 실험을 통해 확인하였다.

두번째 실험은 client당 request를 15개로 고정하고 request로 'buy' 또는 'sell'만 요청할 때, Client 개수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정하여 동시 처리율을 비교한다.

세번째 실험은 client당 request를 15개로 고정하고 request로 'show'만 요청할 때, Client 개수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정하여 동시 처리율을 비교한다.

두번째 실험과 세번째 실험은 Task 2의 Thread-Based Approach에 적용한 First Readers-Writers 문제 해결의 성능을 확인해보고자 진행한 실험이다. 또한 1~3실험에서 Task 2의 Stockserver는 NTHREAD = 256로 고정하여 실험하였다.

마지막 네번째 실험은 Task 2의 Thread-Based Approach Server의 NTHREAD 개수(1,32,128,512)를 다르게 했을 때의 동시처리율 차이를 비교하고자 하였다. client당 request를 15개로 고정하여, Client 개수를 1개,10개,100개, 250개, 500개까지 늘리며 진행하였고 각각의 수행 시간을 측정하였다.

✓ Configuration 변화에 따른 예상 결과 서술

- 각 실험에 대한 결과를 예상하기 전, Task 1, 2의 서로 다른 Server의 특징을 분석해야 한다.

Task 1의 Event-driven Approach Server는 단일 프로세스를 기반으로 단일 thread에서 모든 client를 처리하기에, 메모리 사용량은 적으나, CPU의 부하가 심하고 CPU를 병렬적으로 활용하지 못한다. 이러한 Multicore의 이점을 살리지 못하는 부분은 동시처리율의 부정적인 결과를 불러올 수 있다. 그러나 I/O Multiplexing을 활용한 Non-Blocking I/O('Select()') 방식을 채택했기에, 또한 process나 thread control에 의한 overhead가 없기에 이에 따른 긍정적인 동시처리율을 기대할 수 있다.

Task 2의 Thread-Based Approach Server는 각 client의 request를 미리 생성한 Worker thread가 독립적으로 처리하므로, 응답 속도가 빠를 수 있다. 그러나 Client수가 Thread 수보다 많거나, Thread의 개수가 많음으로 인한 overhead가 발생한다면 이는 성능 저하를 불러올 수 있다고 생각한다. 따라서 Client 수가 어느 정도 이상이 된다면, Overhead로 인한 동시 처리율이 감소할 수 있다고 판단된다. 또한 synchronization을 위한 semaphore 변수의 lock/unlock으로 인해 수행 시간이 길어질 수 있다.

위를 고려하여, Client 수 변화에 따른 동시 처리율을 분석하는 실험 1, Client가 'Show' request만 요청하는 실험 3에서, Client 수가 적은 case들은, Event-driven Approach와 Thread-Based Approach의 처리율이 비슷할 것이며, Client 수가 많아질수록 Thread-Based Approach Server가 Event-driven Approach Server에 비해 더 높은 처리율을 보일 것이다. 우선 Client 수가 적은 case의 경우, 서로 다른 2 server의 성능 차이를 확연히 알 수 없고, 짧은 시간 내에 수행될 것이기에 차이를 느끼기 어렵다고 예측하였다. Client 수가 많은 case일수록 성능 차이는 두드러질 수 있다. Event-driven Approach Server는 동시에 여러 client의 'Read' 작업을 수행할 수 없고, 반면에 Thread-Based Approach Server는 NTHREAD 개수만큼

동시에 'Read'작업을 진행할 수 있기 때문이다. 또한 Thread-Based Approach Server는 Multicore의 이점을 살리며, 실제론 Parallel 하진 않으나, 여러 개의 Thread로 인한 비슷한 효과를 불러올 수 있기에, Event-driven Approach Server보다 좋은 성능을 보일 것이다.

앞선 실험 1, 3의 예측과 달리, Client가 'Write('buy' or 'sell')' 작업만 요청하는 경우인 실험 2에선, Thread-Based Approach Server가 Event-driven Approach Server에 비해 성능이 절대적으로 좋을 것이라 예측하지 못한다. Shared Variable에 대한 Synchronization을 위해서, mutex와 Semaphore 변수들을 사용하였고, 이에 따른 lock/unlock 과정이 많은 overhead를 불러올 것이라 예상한다. 따라서 실험 1, 3과 client 수에 따라 어느정도 비슷한 결과를 보이다가, 특정 Client 수를 기점으로 동시처리율이 감소하는 모습을 보일 수 있다.

마지막으로 Thread-Based Approach Server의 NTHREAD 개수(1,32,128,512)를 다르게 했을 때의 동시처리율 차이를 확인하는 실험 4는 절대적으로 NTHREAD의 개수가 많을수록 성능이 좋다고 생각한다. NTHREAD는 Master-Worker thread 관계에서 Worker Thread의 절대적 개수이며, 이는 곧 client와 맺어진 connection socket file descriptor를 처리할 수 있는 worker의 양이기 때문이다. 물론 Worker thread가 생성된 후, 대기 상태로 지내게 되고, 만약 client 수가 worker thread 개수에 비해 현저히 적다면, 대기 중인 Thread가 많아 자원 낭비가 있을 수 있지만, 이는 실험 결과에 미치는 영향이 매우 작다고 이론적으로 생각한다. 결국 Client 수가 많아질수록 자원 낭비가 불러오는 효과는 적고, 많은 Worker thread가 작업을 수행하여 성능을 향상시키는 효과가 더 크기 때문에, NTREAD의 개수와 동시처리율은 비례할 것이라 예측한다.

C. 개발 방법

- 주식 서버는 효율적인 data관리를 위해 binary tree를 사용하여 주식의 정보를 저장한다. 'stock.txt'에 모든 주식의 정보가 담겨있으며, server 실행 시 정보들을 불러오고, 종료 전에 모든 거래 이후 결과를 update한다. 이 과정들을 구현하기 위해, Binary Tree를 생성하는 'createNode()', 'insertNode()', 주식 정보를 탐색하거나 update, 출력하는데에 필요한 'searchTree()', 'update_stock()', 'printTree()'함수가 있다.

- 서버의 종료는 'Ctrl+C'(SIGINT)가 입력되었을 때이며, Sigint_handler 함수를 통해 주식 정보들을 다시 'stock.txt'에 update한다.

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

B에서 설명한 'pool' 구조체는 여러 client 연결을 관리하기 위한 구조체이다. 'read_set'을 통해 모든 socket file descriptor를 관리하며, 'clientfd[]'에 client와 connection을 맺은 'connfd'를 저장한다. 이외에도 필요한 정보들을 저장하는 변수들이 선언되어 있다. 'init_pool()' 함수를 이용하여 pool을 초기화하며, client file descriptor array와 fd_set 변수들을 초기화한다. 이 때, listenfd를 열어 두어 connection request를 받아들일 준비를 하게 된다. Pool의 read_set을 복사한 ready_set을 'Select()'인자로 넘겨주어 만약 새 client의 연결 요청이 발생되면, 'add_client()'로 해당 client와 맺은 connfd를 pool에 추가한다. 이후 'check_clients()'로 pending input이 있는 connfd를 확인하여, client의 request들을 처리한다. 만약 client와의 연결이 끊기면 해당 connfd를 Close하게 된다. 또 connection을 맺은 모든 client와 연결이 끊기면 주식 정보를 업데이트 한 후, 종료한다.

- Task2 (Thread-based Approach with pthread)

B에서 설명하였듯이, Master Thread는 client와 connection이 맺어진 socket file descriptor를 공유 버퍼에 추가하고 Worker thread는 공유 버퍼를 통해 client의 요청을 처리하게 된다. 이 때 공유 버퍼 'sbuf'는 'connfd'들을 저장하는 배열, 배열의 크기, 배열의 앞과 뒤 index, Producer-Consumer 문제를 해결할 수 있는 mutex, Semaphore 변수들로 구성되어 있다. Server의 Main 함수에서

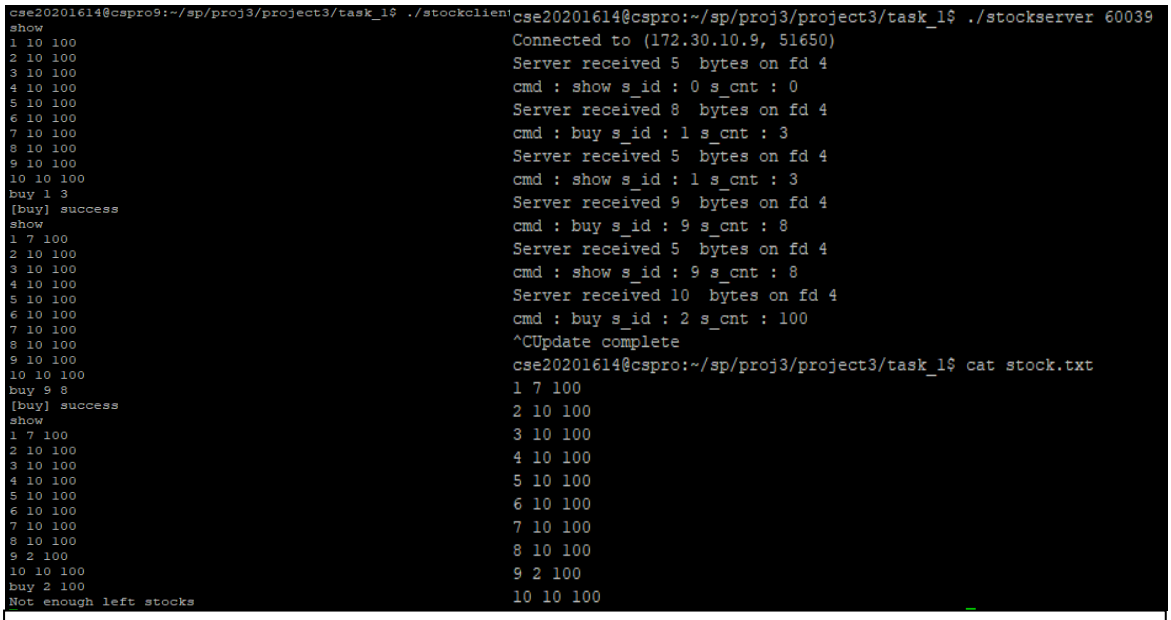
'sbuf_init()'을 통해 'sbuf'를 초기화하며, 'Pthread_create()'를 통해 매크로로 정의한 NTHREADS 개수만큼 Worker thread를 생성하고 대기시킨다. 만약 client와 connection을 맺게 되면, 해당 socket file descriptor를 'sbuf_insert()' 함수를 이용하여 'sbuf'에 추가하고, 대기하고 있던 worker thread는 'Pthread_detach()'함수를 통해 detach 상태로 변하게 된다. 이후 sbuf에서 socket file descriptor 하나씩 처리하여 'echo_cnt()'함수를 통해 client의 request('show', 'buy', 'sell')등을 처리한다.

위와 더불어 First Readers-Writers 문제 해결을 적용하여 코드의 효율성을 높였다. 'show' request는 주식 정보를 읽기만 하기에, 여러 thread가 'Read('show')'에 대한 request를 처리할 수 있도록 하였다. Item의 readcnt 변수 값을 조절하기 전과 후에 적절히 'root->mutex'의 lock/unlock을 수행한다. 또한 thread가 첫 번째 'Read('show')' 작업 수행 시, 'Write('buy 또는 'sell)'을 수행하지 못하도록 'write_mutex'를 lock한다. 여러 thread가 동시에 'Read'를 수행하고 난 후, 마지막 'Read'를 수행하는 thread가 작업이 끝났다면, 다른 thread가 'Write' 작업을 수행할 수 있도록 'write_mutex'를 unlock한다. Write('buy 또는 'sell) 작업을 수행하는 thread의 경우, 동시에 여러 thread가 작업을 수행하지 못하도록, 즉 하나의 thread의 'Write'작업이 끝난 후 다음 thread의 'Write'작업이 이루어지도록 적절히 'write_mutex'를 lock/unlock한다.

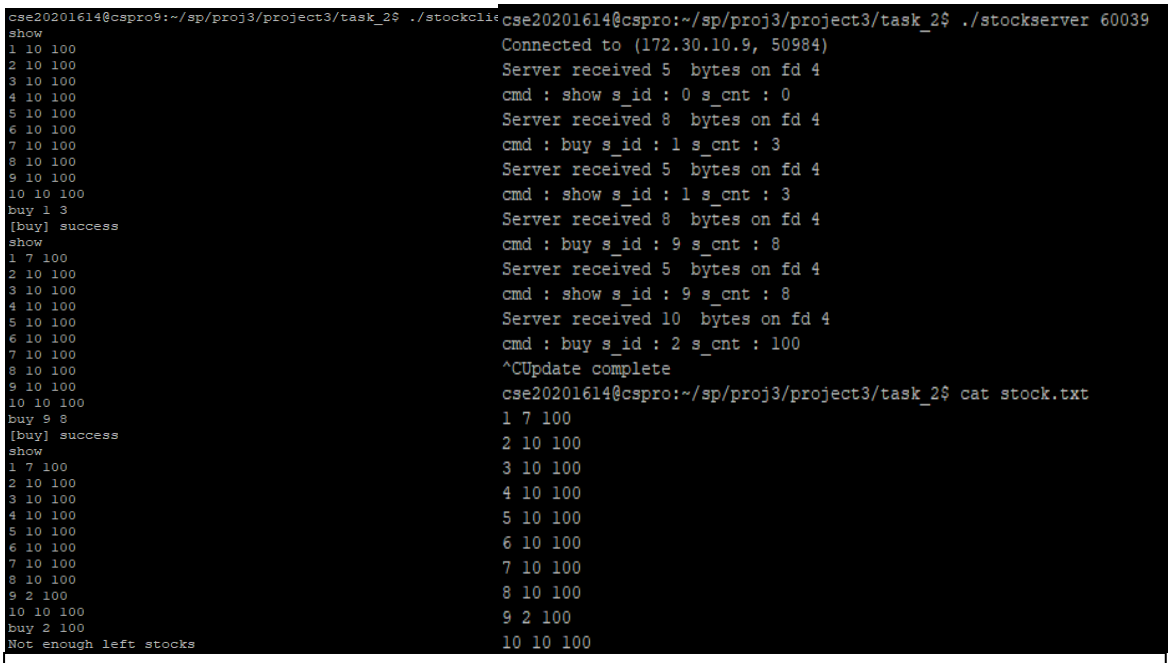
- **Task3 (Performance Evaluation)**

실험 1~4의 client 수와 각 client당 request수 조절, 주식 ID의 총 개수를 조절하는 multiclient.c의 매크로 값을 적절히 변경하여, 사전에 설정한 실험 환경을 진행할 수 있도록 하였다. 또한 실험 4의 경우, stockserver.c의 NTHREAD 매크로 값을 적절히 변경하여 실험을 진행하였다.

3. 구현 결과

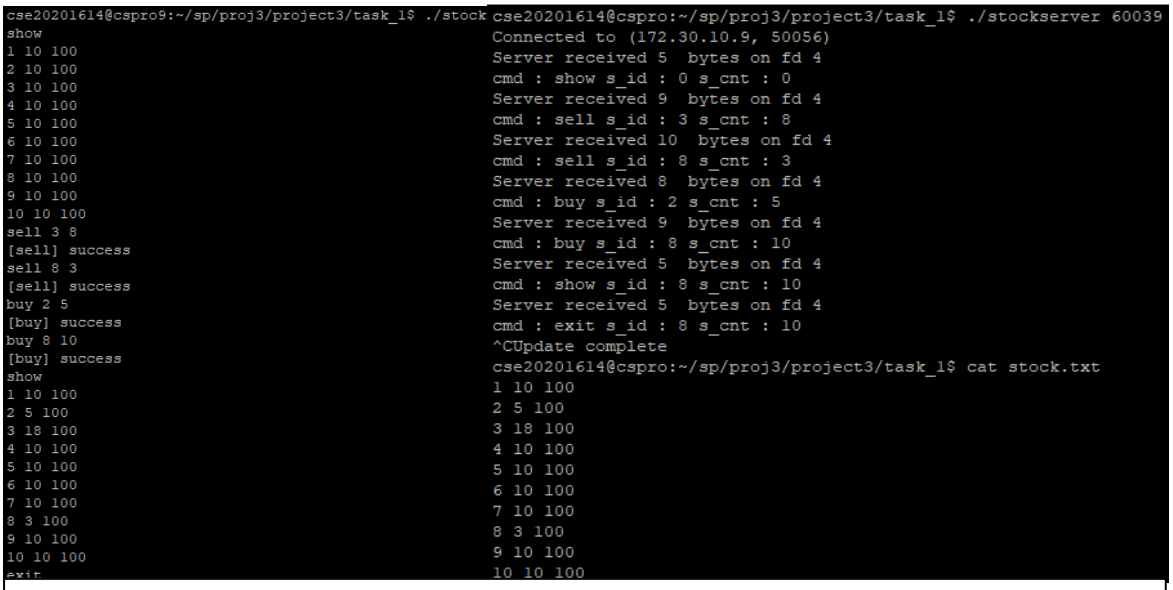


Task 1 Event-driven Approach server의 'show', 'buy' 명령어 실행 장면

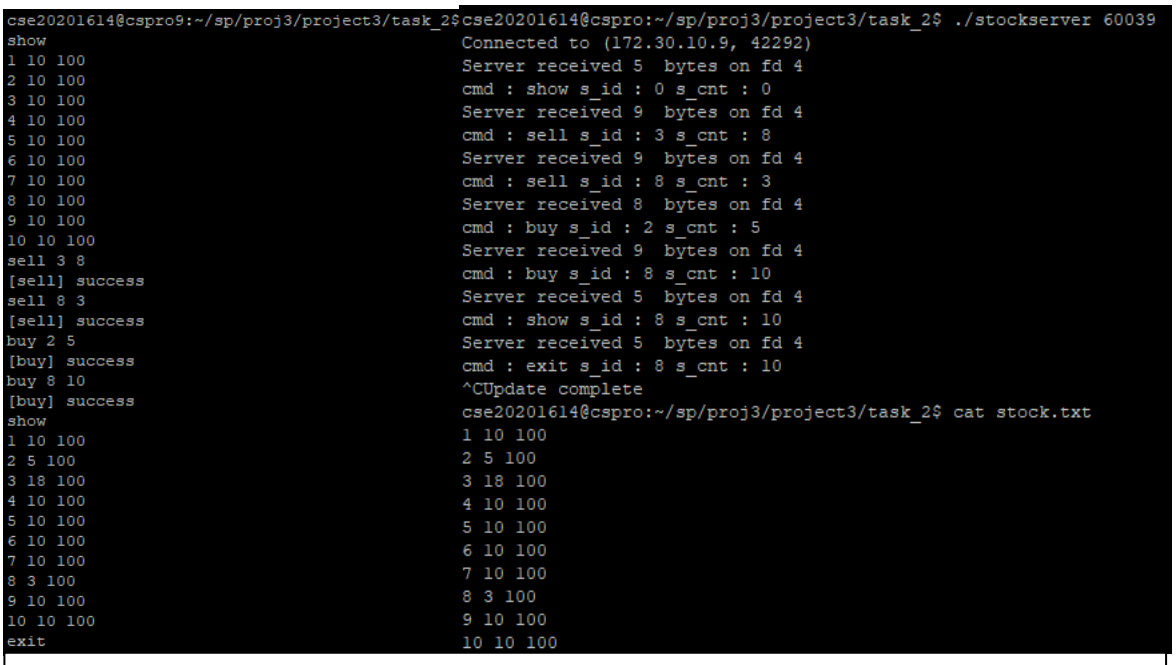


Task 2 Thread-Based Approach server의 'show', 'buy' 명령어 실행 장면

2개의 서로 다른 서버 모두 'stock.txt' 파일에서 주식 정보를 읽어와 저장하고, Client의 요청에 맞게 결과를 출력하고 있다. 만약 잔여 주식 수가 구매하려는 주식 수보다 적을 경우, 적절히 에러문을 출력하며, 종료 후 주식정보를 업데이트 한다.



Task 1 Event-driven Approach server의 'show', 'buy', 'sell' 명령어 실행 장면

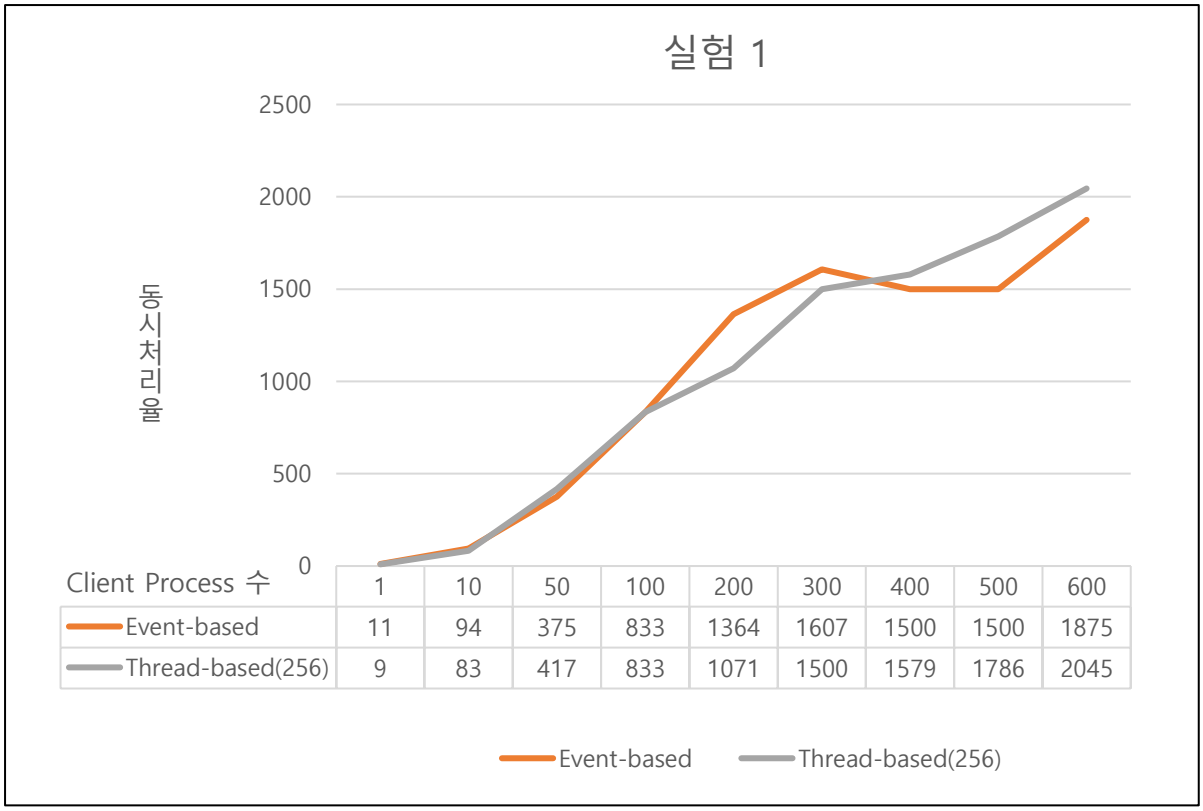


Task 2 Thread-Based Approach server의 'show', 'buy' 명령어 실행 장면

2개의 서로 다른 서버 모두 Client의 'buy', 'sell' request를 수행하고, 이에 맞게 주식 정보들을 새로 업데이트 한다. 이후 'Ctrl+C' 입력으로 서버가 종료되면, "Update Complete"라는 문구와 함께 'stock.txt' 파일에 최종 주식 정보를 저장한다.

4. 성능 평가 결과 (Task 3)

- 실험 1 : Client 수 변화에 따른 동시 처리율



Client Process 수	1	10	50	100	200	300	400	500	600
Event-based	11	94	375	833	1364	1607	1500	1500	1875
Thread-based(256)	9	83	417	833	1071	1500	1579	1786	2045

첫번째 실험은 Client당 request를 15개로 고정하여, Client Process 수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정해 그에 맞는 동시처리율을 비교하였다. 위 결과 그래프를 분석해보자면, Event-driven Approach Server는 Client수가 증가할수록 동시처리율이 선형적으로 증가하지만, 점차 Client 수가 많아질수록, 특히 300개 이상의 구간부턴 처리율의 증가율이 감소하거나 일정수준에서 머무르는 경향을 보였다. Thread-Based Approach Server는 Client 수가 300개 이하인 구간에서 Event-driven Approach Server보다 낮거나 비슷한 동시처리율을 보였다. 그러나 400개 이상의 구간에서는 반대로 더 높은 동시처리율을 보이며, 처리율의 증가 또한 안정적으로 선형적으로 나타나는 모습을 보였다.

이론적 배경을 바탕으로 실험 결과를 분석한다면, Event-driven Approach Server는 단일 프로세스로 동작하며, 여러 Client의 socket file descriptor를 'select()'함수를 통해 살펴보며, 응답이 온 client의 request들을 처리한다. 이러한 I/O Multiplexing 기법은 일정 Client 수 이하에 대해서는 좋은 성능을 보일 수 있으나, 'select()' 함수가 지켜볼 수 있

는 file descriptor의 수에는 한계가 있기에, 일정 Client 수 이상부턴 동시 처리율이 감소할 수 있다. 반면에, Thread-Based Approach Server는 Master-Worker Thread 관계를 사용하여 여러 Client의 request를 병렬적으로 처리할 수 있다. 따라서 일정 Client 수 이상에 대해서 Thread-Based Approach의 장점을 드러내었다고 판단할 수 있다. 그러나 일정 Client 수 이하에 대해서 Event-driven Approach server보다 낮거나 그와 비슷한 동시처리율을 보인 이유는 Synchronization을 위하여 Semaphore, Mutex를 이용한 lock/unlock overhead가 있었기 때문이라고 생각한다.

- **실험 2 : Client 수 변화에 따른 동시 처리율 (buy/sell 요청)**

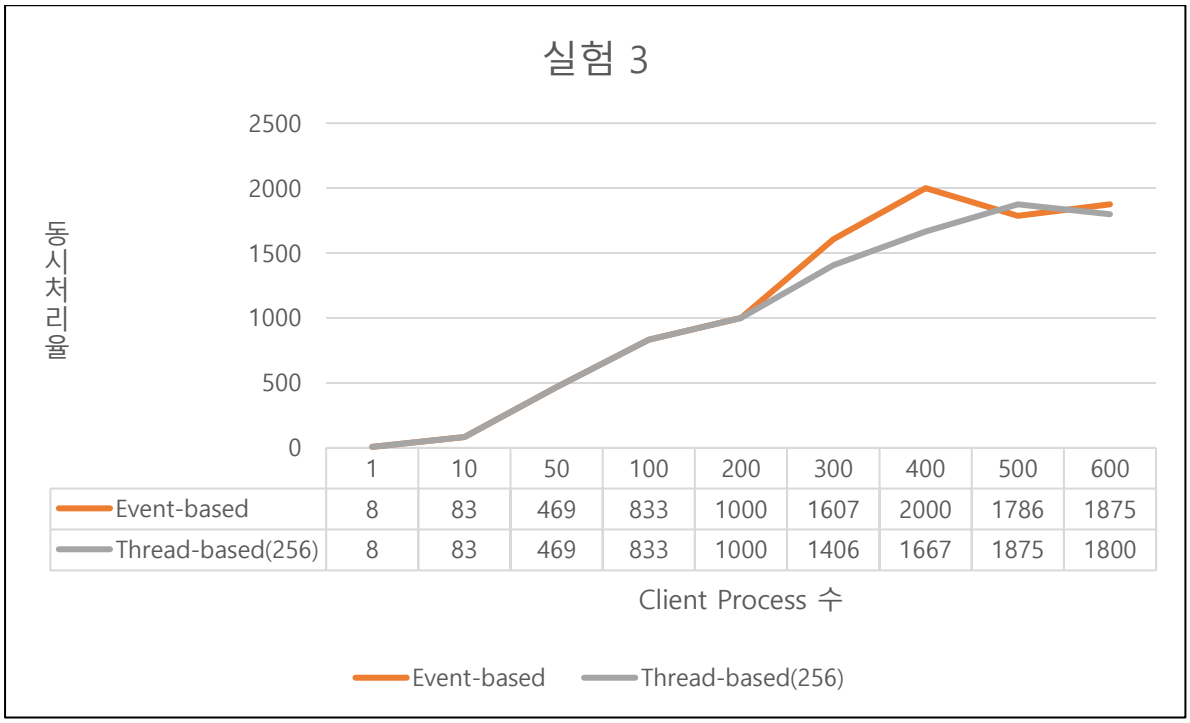
	1	10	50	100	200	300	400	500	600
Event-based	11	94	469	833	1071	1607	1875	1974	2250
Thread-based(256)	9	107	417	833	1071	1607	1579	1786	1957

두번째 실험은 Client당 request를 15개로 고정하고 request로 'buy' 또는 'sell'만 요청할 때, Client Process 수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정해 그에 맞는 동시처리율을 비교하였다. 위 결과 그래프를 분석해보자면, Event-driven Approach Server는 Client수가 증가할수록 꾸준히 동시처리율이 증가하였으나 Client 수가 많아질수록 증가폭이 점차 감소하는 모습을 보였다. 이러한 모습은 Thread-Based Approach Server에서 더 심하게 나타났는데, Client 수가 300개 이하인 구간은 Event-driven Approach Server와 비슷한 동시처리율을 보였으나 Client 수가 300개 이상인 구

간에서 동시처리율이 감소하거나 처리율의 증가폭이 눈에 띄게 감소하는 경향을 보였다.

이론적 배경을 바탕으로 실험 결과를 분석한다면, Event-driven Approach Server는 실험1의 경우와 같이, 'show', 'buy', 'sell'을 I/O Multiplexing을 통해 한 request를 처리한 후 다음 request를 처리하기에, 실험 1과 overhead 측면에서 별 차이가 없는 상황이다. 반면 Thread-Based Approach Server는 First Readers-Writers 문제를 해결하기 위해 Semaphore, Mutex 변수를 사용하여 lock/unlock을 하게 되는데 이 때 'Write' 작업인 'buy' 와 'sell'이 많아질수록 overhead가 커짐을 이번 실험을 통해 알 수 있었다. 또한 여러 thread가 동시다발적으로 'Write'작업을 수행한다면 Synchronization에 의한 overhead도 발생할 수 있으며, 언급한 두 가지 이유로 인해 성능저하가 이번 실험 2에서 발생하였고, 결과가 이를 충분히 보여주었다고 생각한다.

- 실험 3 : Client 수 변화에 따른 동시 처리율 (show 요청)



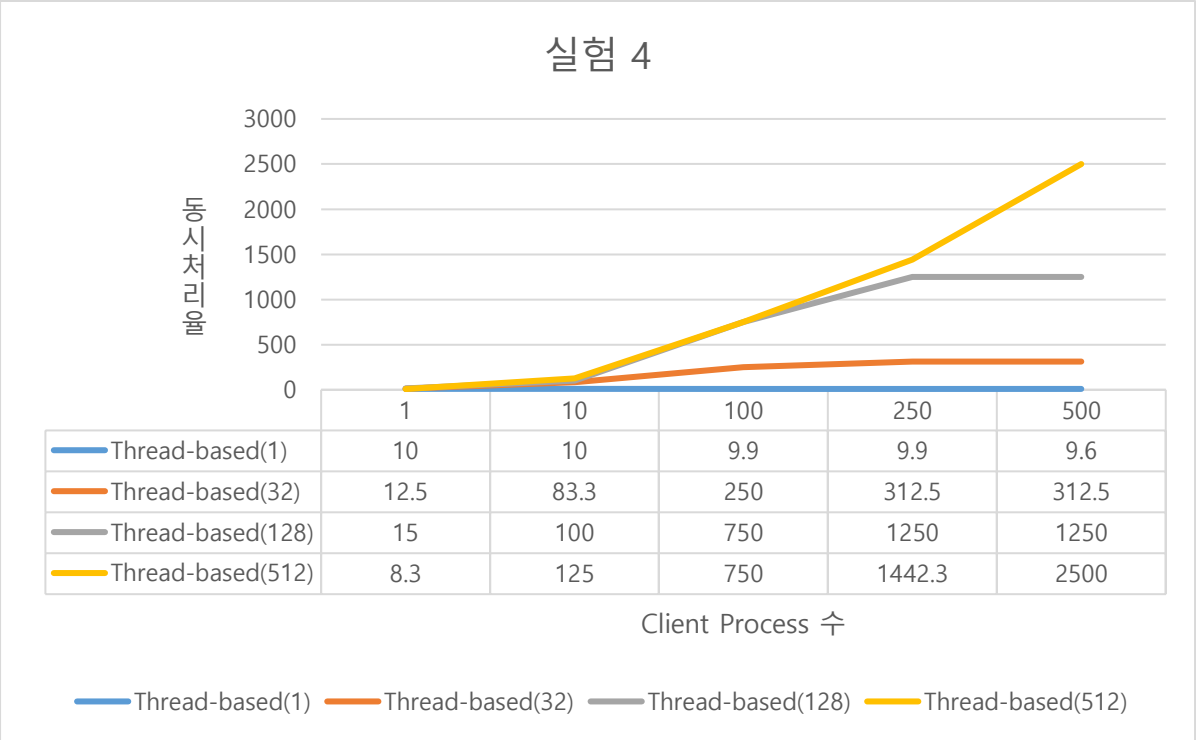
	1	10	50	100	200	300	400	500	600
Event-based	8	83	469	833	1000	1607	2000	1786	1875
Thread-based(256)	8	83	469	833	1000	1406	1667	1875	1800

세번째 실험은 Client당 request를 15개로 고정하고 request로 'show'만 요청할 때, Client Process 수를 1개부터 600개까지 늘리며 이에 따른 수행 시간을 측정해 그에 맞는 동시처리율을 비교하였다. 위 결과 그래프를 분석해보자면, Event-driven Approach

Server는 실험 1, 2와 같이 Client수가 증가할수록 꾸준히 동시처리율이 증가하였으나 Client 수가 많아질수록 증가폭이 점차 감소하는 모습을 보였다. 특히 이번 실험 3에서는 Client 수가 400개 이상일 때 감소하는 폭이 컸다. Thread-Based Approach Server는 실험 2와 다르게 Client 수가 증가할 수록, 동시처리율도 꾸준히 증가하는 모습을 보였으나, Client 수가 300개 이상일 경우 증가폭이 감소하는 모습도 보였다.

이론적 배경을 바탕으로 실험 결과를 분석한다면, Event-driven Approach Server는 실험 1, 2의 경우와 같이, 'show', 'buy', 'sell'을 I/O Multiplexing을 통해 한 request를 처리한 후 다음 request를 처리하기에, 실험 1과 overhead 측면에서 별 차이가 없는 상황이다. 반면 Thread-Based Approach Server는 First Readers-Writers 문제를 해결하기 위해 Semaphore, Mutex 변수를 사용하여 lock/unlock을 하게 되는데 이 때 'Read' 작업인 'show' request는 여러 Worker thread가 동시에 작업을 수행할 수 있으므로, 실험 2에 비해 높은 처리율을 보인다. 이는 Client 수가 400~500구간에서 잘 나타났다. 결국 First Readers-Writers 문제 해결을 적용한 코드가 'Read' 작업이 많은 상황에서 효과를 나타냄을 확인할 수 있었고, Semaphore를 이용한 Synchronization에서의 overhead가 'Write'작업이 많을 때에 비해 적다고 판단할 수 있었다.

- **실험 4 : NTHREAD 개수 변화에 따른 동시 처리율**



	1	10	100	250	500
Thread-based(1)	10	10	9.9	9.9	9.6
Thread-based(32)	12.5	83.3	250	312.5	312.5
Thread-based(128)	15	100	750	1250	1250
Thread-based(512)	8.3	125	750	1442.3	2500

마지막 네번째 실험은 Task 2의 Thread-Based Approach Server의 NTHREAD 개수(1,32,128,512)를 다르게 했을 때의 동시처리율 차이를 비교하고자 하였다. client당 request를 15개로 고정하여, Client 개수를 1개,10개,100개, 250개, 500개까지 늘리며 진행하였고 각각의 수행 시간을 측정하였고, 이에 따른 동시처리율을 비교하였다. 위 결과 그래프를 분석해보자면, NTHREAD = 1 일 때, Client 수가 증가해도 처리율은 거의 10으로 일정하다. NTHREAD = 32 일 때 동시처리율은 Client 수가 증가할수록, 증가하는 모습을 보이나 증가폭이 매우 낮음을 알 수 있다. NTHREAD = 128 일 때, NTHREAD = 32와 같은 모습을 보이나, Client 수가 10개 이상일 때부터, 더 큰 증가폭을 보이다, Client수가 250개 이상일 때 일정한 값을 유지함을 확인할 수 있다. NTHREAD = 512 일 때, Client 수가 증가할수록, 동시처리율도 급증하며, 이러한 증가폭은 Client수가 더 많아질수록 확연히 드러남을 확인할 수 있다.

이론적 배경을 바탕으로 실험 결과를 분석한다면, Task 2의 Thread-Based Approach Server는 Master-Worker Thread를 적용하였기에, NTHREAD의 값, 즉 Worker Thread가 많을 수록 병렬로 처리할 수 있는 Client수가 증가하기에 처리율 또한 높아질 수밖에 없다. 물론 NTHREAD 값이 매우 높아도, Client 수가 이에 비해 적다면, 대부분의 Worker Thread는 대기 상태로 있어 자원을 낭비하고, NTHREAD 값의 차이에 따른 동시처리율 변화를 확인하기 쉽지 않다. 그러나 Client 수가 많아질수록 이러한 overhead

는 감소하고 높은 처리율을 보였다고 판단할 수 있다. 이는 2-B에서 예측한 결과와 비슷하다고 할 수 있다.

- 실험 1~4를 진행한 Task 3의 총 결론 및 종합 분석

Event-driven Approach Server는 단일 process에서 모든 Client의 request를 처리하기에 Client 수가 적다면 효율적이나, Client 수가 많아질수록 처리 성능이 저하됨을 확인할 수 있었다. 그러나 Non-Blocking I/O와 'select()' 함수를 이용한 I/O Multiplexing은 Thread-Based Approach Server와 비슷한 동시처리율을 보이게끔 하는 긍정적 효과를 불러일으켰다. 이는 특히 'Read' 작업인 'show' request 처리율에서 확연히 드러났다.

Thread-Based Approach Server는 여러 Thread가 병렬로 Client의 request를 처리하기에 Client 수가 많아질수록 높은 처리율을 보이며, 그러한 상황에서 효과적임을 보였다. Synchronization을 위한 Semaphore, Mutex의 lock/unlock으로 인한 overhead가 일으키는 동시처리율의 감소도 확인할 수 있었으나, Readers-Writers 문제 해결을 적용한 방법이 이를 어느정도 해소하였다고 판단할 수 있었다.

이번 Task 3를 수행하면서, 실험 4개를 진행하였을 때, 모든 실험의 상황이 일정하게 같게 유지할 수 없었다. 이에 따라 실험 과정의 각 case 결과값인 수행 시간이 매우 낮거나 매우 높게 나온 경우도 존재하였다. 이러한 이상치들을 최대한 제거하고자 5번의 반복적인 실험을 진행하여 수행시간을 평균내고자 하였고, 그럼에도 불구하고 이론적 사실을 기반으로 한 예측과 실제 실험 결과가 많이 달랐다. 그러나 두 개의 서로 다른 접근방식의 장단점과 차이점을 어느정도 확인할 수 있었던 실험이라고 생각한다.