System Programming Project 3

담당 교수 : 박성용 교수님

학번 : 20201654

이름 : 최호진

1. 개발 목표

주식 서버에는 여러 클라이언트가 접속하여 request를 보내고, 서버는 이에 알맞게 작업을 수행한다. 이 프로젝트에서는 주식 서버에 여러 클라이언트가 동시에 작업을 요청할 때 오류(Race condition 등)없이 명령을 처리할 수 있도록 두 가지 방법(Event-driven Approach, Thread-based Approach)을 이용해 서버를 구현한다.

클라이언트는 서버에 4가지 명령어(show, buy, sell, exit)를 요청할 수 있고, 이때 각 명령어는 항상 올바른 입력으로 주어진다고 가정한다. 또한, 서버는 주식에 대한 정보를 stock.txt 파일에서 가져와 이를 바이너리 트리 자료구조 형태로 서버에 저장해 이를 클라이언트의 요청을 처리할 때 사용하고, stock.txt에 변경된 주식 정보를 다시 저장하며 주식을 관리한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

해당 방식으로 구현할 때에는 I/O Multiplexing을 활용하여 구현하였다. Select 함수를 이용해 하나의 process, 하나의 thread에서 여러 클라이언트의 request를 동시에 처리할 수 있도록 한다. pool 구조체를 정의하여 여러 클라이언트의 파일 디스크립터를 관리하고, select 함수로 각 클라이언트에 대하여 새로운 request가 서버에 들어왔는지 확인하며 요청이 들어왔다면 알맞게 이를 처리하여 클라이언트에게 리턴한다.

2. Task 2: Thread-based Approach

해당 방식으로 구현할 때에는 각 클라이언트의 요청을 처리할 때 클라이언트마다 별도의 Thread에서 이를 처리한다. Master Thread에서는 클라이언트들의 connect에 대하여 accept하여 연결을 관리하고, worker thread에서는 하나의 클라이언트에 대해 request를 처리하는 방식으로 구현한다. Thread-based의 경우 하나의 프로세스에서 동일한 변수에 대하여 동시에 요청을 처리하면서 의도치 않은 결과를 불러일으킬 수 있기 때문에 semaphore를 이용하여 한 thread가 클라이언트의 요청을 처리할 때 다른 thread가 critical session에 접근하지 못하도록 적절하게 코드를 작성하여 관리한다.

3. Task 3: Performance Evaluation

Task 1과 Task 2에서 구현한 서버에 대하여, 동시에 들어오는 클라이언트의 수, 요청

하는 명령의 종류들을 변경하며 성능을 평가한다. 평가 방식은 기존의 multiclient를 변경해 multiclient 실행 파일이 시작될 때부터 종료할 때까지의 elapsed time을 측정하여 이를 비교한다. 또한, Task 2에 대해서는 reader-writer problem을 해결하기 위해 구현한 방식과, 이를 고려하지 않고 구현한 방식의 시간 차도 비교해본다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  Select 함수를 이용해 I/O Multiplexing을 하는데, 구현한 주요 내용은 다음과 같다.

  1. pool 구조체를 정의하고 변수를 선언한다. 구조체 내에는 I/O Multiplexing을 위한 file descriptor set(read, ready set), 및 clientfd 배열, clientrio 배열과 개수를 나타내는 변수들이 포함되어있다.
  2. init_pool 함수에서 pool 구조체 변수를 초기화한다.
  3. 서버가 종료될 때까지 while문을 돌면서 select 함수를 이용해 pool 구조체 내의 ready set에 있는 listenfd에 대해 event가 발생했는지(bit이 set 되어있는지) 확인한다.
  4. set 되어 있다면 Accept를 이용해 client와 연결하고 pool 구조체의 read set에 connfd를 추가한다.
  5. 그 다음, check_clients 함수에서 connfd에 새 request가 들어왔는지(ready set에 set된 bit를 가진 connfd가 있는지) 확인하고, 있다면 클라이언트의 요청을 하나씩 처리하며 모든 요청을 다 처리할 때까지 이를 반복한다.
  6. 모든 client의 요청을 처리하면 stock.txt file을 update해준다.

  ✓ epoll과의 차이점 서술

  epoll은 select와 마찬가지로, I/O Multiplexing에 사용된다. 하지만 select 함수가 file descriptor set에 있는 모든 파일 디스크립터에 대해 하나씩 event가 발생했는지 함수가 호출될 때마다 확인하는 반면, epoll은 디스크립터의 상태 변화를 확인한다. 다시 말해, epoll 함수는 이벤트가 발생한 file descriptor만 반환하므로, 모든 file descriptor를 확인해야하는 select 함수보다 더 많은 file descriptor를 처리하는

데 성능적으로 더 뛰어나다고 할 수 있다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 클라이언트의 Connection을 관리하는데, 이때 구현한 방식은 pre-threaded 방식을 이용해 구현하였다.

  1. 먼저 worker thread의 pool을 관리할 구조체인 sbuf 구조체를 정의하고 변수를 선언하고 초기화한다. 이때 구조체 내에는 버퍼 배열과 mutex, 이용 가능한 slot의 개수, item의 개수를 나타내는 변수들을 포함하고 있다.
  2. Master Thread에서는 먼저, 원하는 개수의 worker thread를 먼저 create 한다.
  3. while문을 돌면서 Accept 함수로 client와 연결하고, sbuf_insert 함수로 sbuf에 connfd를 넣어준다. 이후에 client의 요청은 workerd thread가 처리하게 된다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  앞서 말했듯이 Worker thread에서 클라이언트의 요청(show, sell, buy)을 처리한다. Worker Thread Pool은 Thread의 집합으로 sbuf 구조체를 이용해 관리한다. master thread에서 sbuf_insert 함수로 넣어준 connfd를 worker thread들이 thread 함수에서 sbuf_remove 함수를 이용해 하나씩 가져와 check_clients 함수에서 클라이언트의 요청을 처리한다.

- **Task3 (Performance Evaluation)**

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

  Task3에서는 앞에서 구현한 서버의 성능을 측정한다. 이때 서버의 성능을 평가하는 기준으로 서버의 동시처리율이 뛰어난 프로그램을 더 성능이 좋은 서버로 평가한다. 이때 동시 처리율의 metric을 정의하는데, 아래와 같다.

$$동시\ 처리율 = \frac{Client\ 수 * Client\ 당 order\ 수}{Time}$$

  이렇게 정한 이유는 동시 처리율은 같은 시간을 기준으로 서버가 얼마나 더 많은 클라이언트의 요청을 처리할 수 있는지 평가하는 것이 중요하다고 생각했는데 위와 같이 metric을 정의하면 이를 평가할 수 있다고 생각해서 이렇게 정의했다. 이러한 평가 기준을 바탕으로, Task1에 대해서는 클라이언트의 수, 전체 주식

의 수, 그리고 요청하는 명령의 종류(show만 주어졌을 때, 랜덤으로 주어졌을 때 등), 클라이언트 당 명령의 수를 바꿔가며 각각의 성능이 어떻게 변하는 지 평가한다. 이를 측정하는 방법으로 multiclient 프로그램을 이용해서 실험을 진행하기 때문에, 해당 코드의 main 함수의 while loop 시작과 끝 부분에 시간 차를 구하여 시간을 측정하고 이를 동시 처리율을 계산할 때 사용한다. 이렇게 시간을 측정한 이유는 실험을 진행할 때 가장 다른 변수 없이 일정하게 측정할 수 있다고 생각해 이렇게 측정하였다.

Task2에 대해서는 Task1에서 평가한 방법에 더해서, Pre-threaded 방식으로 구현했기 때문에 처음에 생성하는 Thread의 개수와 SBUF 크기를 변경해 성능 차이를 측정한다. 또한 reader-writer 문제를 고려해 구현했을 때와 그렇지 않을 때 성능 차이가 유의미하게 보이는지 평가한다.

✓ Configuration 변화에 따른 예상 결과 서술

먼저 task1(Event-based Approach)과 task2(Thread-based Approach)에서 구현한 방식을 비교했을 때 task2의 방식이 더 뛰어날 것으로 예상된다. 왜냐하면 Event-base Approach는 하나의 프로세스, 하나의 Thread에서 모든 클라이언트의 요청을 처리하기 때문에 Parallel하게 처리하는 것이 불가능하다. 하지만 Task2는 여러 Thread를 이용해 클라이언트의 요청을 처리하기 때문에 멀티코어를 가진 CPU에서는 이를 Parallel하게 처리하는 것이 가능하다. 따라서, CPU의 코어 수가 많은 cspro 서버에서 Task2의 성능이 더욱 뛰어날 것으로 예상했다.

다음으로, ORDER PER CLIENT의 값을 변형하며 실행하면, 동시 처리율에서는 큰 차이가 없을 것으로 예상했는데, 그 이유는 전체 시간이 늘어날 수는 있어도 동시 처리율에 영향을 주는 파라미터가 아니라고 생각했기 때문이다.

또한, 명령어의 종류에 따라 결과의 차이를 예상했을 때 Task1의 경우에는 명령어의 종류에 따른 성능에 큰 차이는 없을 것으로 예상되지만 Task2에서 reader-writer Problem를 고려해서 작성한 코드와 그렇지 않은 코드를 비교했을 때, show 명령어(Reader만 존재하는 경우)만 보내는 경우 전자가 더 좋은 성능을 가질 것으로 예상된다. 왜냐하면, 전자의 경우 reader와 reader(show가 동시에 요청됨)는 동시에 요청을 처리할 수 있지만, 후자의 경우는 이를 고려하지 않고 모든 명령에 대해서 Lock을 걸어 놓기 때문에 전자가 더 좋은 성능을 가질 것으로 예상한다.

다음으로, Task2에서 초기에 생성하는 Worker Thread의 수를 늘릴수록 더 많은

클라이언트의 요청을 동시에 처리할 수 있어 더욱 좋은 성능을 가질 것으로 예상된다. 또한 sbuf의 사이즈가 큰 경우 slot의 개수가 많기 때문에 동시에 처리할 수 있는 명령의 수가 늘어나 더욱 좋은 성능을 가질 것으로 예상한다.

C. 개발 방법

1. 주식 저장 자료 구조 (Binary Tree): Task1, Task2 공통

```c
typedef struct item{
    int ID;
    int left_stock;
    int price;
    int readcnt;
    sem_t mutex;
} ITEM;
```

```c
typedef struct tree_node{
    ITEM item;
    struct tree_node *left;
    struct tree_node *right;
} tree_node;
typedef tree_node *TREE_NODE;
```

Task 1과 2에서 공통적으로 주식 데이터를 저장 및 관리하기 위해 binary tree를 이용하였다. 트리의 노드를 구성하는 주식에 대한 정보는 ITEM 구조체로 정의했고 이를 linked list를 이용해 구현하였다. 처음 stock.txt를 읽어오면 주식 ID를 기준으로 ID값이 작은 것이 Left Child, 큰 것이 Right Child로 들어가도록 정렬하여 tree에 저장하였고, 이 자료구조를 이용해 클라이언트의 요청(show, buy, sell)을 처리하였다.

2. Task1 (Evented-Based Approach)

```c
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;
```

Evented-Based Approach로 클라이언트의 연결을 관리하기 위해 pool 구조체를 선언한다. 구조체 내에는 최대 clientfd 수를 나타내는 maxfd, event가 발생한 디스크립터를 확인하기 위한 fd_set(ready_set과 read_set), set되어 있는 디스크립터의 수를 나타내는 n_ready, clientfd 배열과 buffer로 구성되어 있다.

```c
listenfd = Open_listenfd(argv[1]);
init_pool(listenfd, &pool);

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)){ //new listenfd connection
        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 client to pool
    }
    check_clients(&pool);

    FILE* fp = fopen("stock.txt", "w");
    if(fp == NULL){
        printf("FILE OPEN ERROR: stock.txt\n");
        return 0;
    }
    update_stock_file(fp, root);
    fclose(fp);
}
```

<Event-Based Approach Main Loop>

해당 구조체 변수를 선언하고, listenfd로 클라이언트의 request를 대기하고, init_pool 함수에서 초기화한다. 초기화할 때에는 read_set의 bit를 0으로 초기화해놓는다. 그 다음 while loop을 돌면서 ready_set에 read_set을 복사한 후 Select 함수로 event가 발생한 디스크립터를 확인한다. 이때 set된 디스크립터가 ready_set에 존재한다면, 이는 새 클라이언트의 연결 요청이 들어왔음을 의미하므로 Accept 함수를 이용해 연결하고, 새 클라이언트를 pool에 추가한다.

이후 check_clients 함수를 호출해 pool 내부에 있는 클라이언트들의 요청이 들어왔는지 확인하고, 요청이 들어왔다면 이들을 순서대로 하나씩 처리하여 모든 event를 처리할 때까지 반복하고, 다시 main loop으로 돌아오며 반복하여 실행한다.

주식에 대한 정보는 main loop을 돌면서 계속해서 파일에 저장하고, 마지막에 서버를 종료할 때(SIGINT) signal handler 코드를 작성해 파일에 저장하고 프로세스가 종료될 수 있도록 구현하였다.

3. Task2 (Threaded-Based Approach – Pre-threaded Approach)

```c
typedef struct {
    int *buf;
    int n;
    int front;
    int rear;
    sem_t mutex;
    sem_t slots;
    sem_t items;
} sbuf_t;
```

Pre-threaded Apporach 방식으로 구현하였고, thread pool을 관리하기 위한 sbuf 구조체를 정의하였다. 구조체 내에는 buffer와 mutex, slot, item과 이에 대한 정보를 나타내는 변수들로 구성되어있다.

```c
listenfd = Open_listenfd(argv[1]);
sbuf_init(&sbuf, SBUF_SIZE);

for(i = 0; i < NTHREADS; i++){
    Pthread_create(&tid, NULL, thread, NULL);
}

read_stock_file();
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);

    FILE *fp;
    fp = fopen("stock.txt", "w");
    update_stock_file(fp, root);
    fclose(fp);
}
```

<Thread-Based Approach Main Loop>

Task1과 마찬가지로, listenfd를 open하고 sbuf를 초기화한다. 초기화 할 때 버퍼의 사이즈를 결정하고 mutex와 slot, item을 각각 알맞은 수로 초기화한다. 이후, NTHREADS에 적당한 수를 할당하여 초기에 만들 worker thread 수를 결정한다. (이 값은 Task3에서 적절히 변경해가며 실험을 진행한다.) Pthread_create 함수로 worker thread를 생성한 뒤 while loop을 돌면서 메인 Thread에서는 connfd를 Accept 함수를 이용해 받는다. 이렇게 연결된 새 클라이언트를 sbuf_insert 함수로 sbuf 구조체의 버퍼에 추가한다.

Worker thread에서는 thread 함수를 수행하는데, 이 함수에서는 sbuf 구조체에 추가된 connfd를 하나씩 빼내어 클라이언트의 request를 수행하기 위한 check_clients 함수를 호출하고 이를 수행한다. 클라이언트의 요청을 끝내면 Close(connfd)로 클라이언트와의 연결을 종료하고, thread는 다시 thread pool로 반환된다. 이때 중요한 점은 mutex를 이용해 thread 간에 서로 예기치 못한 상황이 일어나지 않도록 하나의

Thread가 stock의 값을 변경하는 상황에서 다른 thread가 해당 부분에 접근하지 못하도록 lock해야 한다. 또한 Reader-Writer problem을 해결하기 위해 하나의 thread가 show를 수행하는 동안 다른 thread들에서 show를 수행할 수 있도록 하는 반면 다른 case의 경우에는 하나의 thread가 해당 요청을 수행할 때 다른 thread에서는 해당 부분에 Lock이 걸려있을 수 있도록 구현하였다.

4. Task 3

Task 3에서는 Task1과 Task2에서 구현한 서버에 대한 평가를 진행한다. 평가 기준은 앞서 정의한 metric을 이용해 한다. 그 다음, 실험 세팅 시 기본적인 세팅 사항으로 주식 종류 수는 5개로 제한했으며, buy와 sell 명령에서 가질 수 있는 최대 값은 10으로 정의하였다. 또한 동일한 실험을 진행하더라도 서버의 실험 당시 환경에 따라 값의 편차가 클 수 있다고 생각하여 동일한 실험을 10번 반복하여 실행하고, 이때 가장 큰 값과 2번째로 큰 값을 제외한 실험 결과의 평균을 측정해 비교하기로 정하였다. 실험을 할 내용은 아래와 같다.

[1] Task1, Task2 공통

- 동시에 접속하는 클라이언트의 수에 따른 시간 및 동시처리율 변화 측정
- Client의 request의 종류에 따른 성능 차이 측정(show, buy, sell, random)
- ORDER_PER_CLIENT의 변화에 따른 성능 변화

[2] Task2 실험 내용

- NTHREAD의 변화에 따른 성능 변화
- SBUF_SIZE의 변화에 따른 성능 변화
- Reader-writer를 고려해 구현한 코드와 그렇지 않은 코드의 성능 차이 측정

이러한 점들을 중심으로 실험을 진행하였다.

3. 구현 결과

1. Task 1

Event-driven Approach는 앞서 예상했던 것처럼 하나의 Thread에서 진행되므로 reader-writer problem을 고려하지 않아도 되었고, 실제로 결과에서도 여러 번 반복해서 진행했을 때 해당 문제가 발생한 경우가 없었고, 명령어의 종류에 따른 성능에

큰 차이가 있지 않았다. 또한, concurrent하게 mutliclient의 요청을 잘 수행함을 볼 수 있었다.

2. Task 2

Thread-based Approach는 semaphore를 통해 여러 Thread가 하나의 프로세스에서 동작하면서 예기치 못한 상황을 일으키지 않도록 작성했어야 했다. 실제로 클라이언트 간의 읽기(show)와 쓰기(buy, cell)의 충돌로 인한 race condition 또는 deadlock 등이 발생하지 않았다. 또한 Task2는 Reader-Writer 문제를 고려한 코드와 그렇지 않은 코드 두 가지 버전으로 작성하여 보았는데, 성능 차이를 예상했지만 실제로 큰 성능 차이를 보이지 않았다. 이는 실제 프로그램에서 Critical Section이 차지하는 부분이 크지 않기 때문에 Semaphore로 인한 waiting 시간이 그렇게 길지 않아서 이러한 결과를 보였을 것이라 예상된다.

3. 성능 향상 방법

현재 주식을 관리하는 자료구조로 Binary Tree를 선택했지만, stock.txt에서 처음에 자료가 저장된 순서에 따라 Unbalanced한 Tree가 생성되고 Binary Tree에서 기대하는 시간 복잡도는 O(logN)이지만 최악의 경우 시간 복잡도가 O(N)이 될 수 있다. 따라서 Red-Black Tree와 같이 Balance를 맞출 수 있는 자료구조를 선택해 Data에 접근하는 시간을 줄일 수 있다면 더욱 성능이 좋은 서버가 될 것이라 예상된다.

4. 성능 평가 결과 (Task 3)

여러 조건들을 변경하며 실험을 진행했고, 실험은 cspro에서 server를 open하고 cspro9에서 client를 연결해 진행했다. 실험 구현 내용에서 설명했듯이 동일한 실험을 10번 진행해 가장 시간이 오래 걸린 2개의 결과를 제외한 8개의 값에 대한 평균을 내어 시간을 측정했다. 이렇게 시간을 측정한 이유는 서버의 상황에 따라 성능이 급격히 떨어진 값들이 있을 것이라 에상했고, 이러한 오류를 제거하고자 가장 큰 두 값을 제외한 평균을 측정하는 방식으로 진행했다. Multiclient에서 시간을 측정하였고, 마지막 9번 실험을 제외한 나머지의 경우에서 multiclient 내에 sleep 함수를 지우고 측정하였는데, 이렇게 측정한 이유는 클라이언트의 연결 요청과 이외의 요청이 섞여 들어올 수 있도록 하기 위해 sleep을 지우고 진행했다. 실험 진행 시 클라이언트 수는 10개, 100개, 250개, 400, … , 1000개까지 150개 단위로 실험을 진행하였다.

측정 시점은 멀티 클라이언트 파일의 시작과 끝 부분에서 측정을 하였고 아래 사진과 같다.

```
    start = times(&t);


/*  fork for each client process    */
    while(runprocess < num_client){
```

```
105     for(i=0;i<num_client;i++){
106         waitpid(pids[i], &status, 0);
107     }
108
109     end = times(&t);
110
111     printf("elapsed time: %f\n", (double)(end-start)/freq);
112     return 0;
113 }
```

쉘 스크립트를 작성해여 실험을 진행했고 스크립트 내용은 아래와 같다.

```
sp > project3 > task1 > $ test.sh
1   #!/bin/bash
2
3   touch ./output/log_$2.txt
4   for((i = 1; i <= 10; i++))
5   do
6       ./multiclient 172.30.10.11 60048 10 $1 | grep -E "time:" >> ./output/log_$2.txt
7   done
8
9   for ((i=100; i<=1000; i+=150))
10  do
11      for ((j=1; j<=10; j++))
12      do
13          ./multiclient 172.30.10.11 60048 $i $1 | grep -E "time:" >> ./output/log_$2.txt
14          sleep 1
15      done
16  done
```

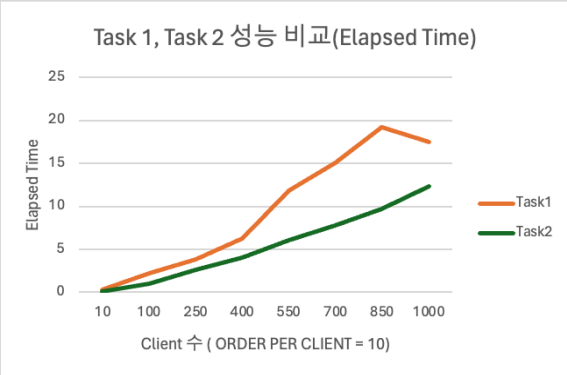
이 log를 이용해 엑셀 파일을 작성해 표와 그래프를 그렸다. 또한 실험 진행 시 클라이언트 수는 10개, 100개, 250개, 400, … , 1000개까지 150개 단위로 실험을 진행하였다. 아래는 로그 파일과 엑셀로 정리한 파일의 예시이다.

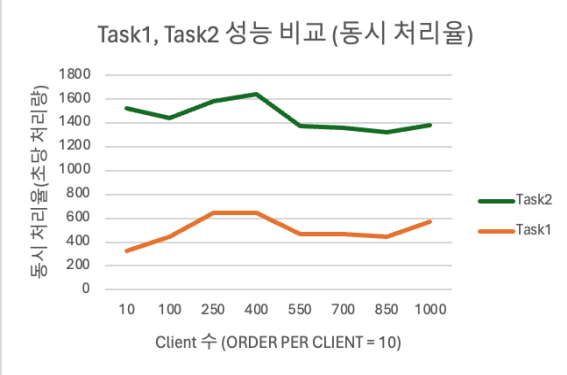
```
sp > project3 > task1 > output > ≡ log_random_10.txt
 1    elapsed time: 0.320000
 2    elapsed time: 0.360000
 3    elapsed time: 0.330000
 4    elapsed time: 0.340000
 5    elapsed time: 0.290000
 6    elapsed time: 0.300000
 7    elapsed time: 0.350000
 8    elapsed time: 0.310000
 9    elapsed time: 0.290000
10    elapsed time: 0.260000
11    elapsed time: 2.020000
12    elapsed time: 2.470000
13    elapsed time: 2.450000
14    elapsed time: 2.550000
```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	elapsed time:	0.100000		0.100000	0.1	ORDER PER CLIENT	10	10	100	250	400	550	700	850	1000
2	elapsed time:	0.080000		0.080000	0.08		1	0.1	1.06	3.9	5.11	14.75	30.22	15.32	16.65
3	elapsed time:	0.140000		0.140000	0.14		2	0.08	1.39	4	6.06	8.12	30.48	55.21	28.82
4	elapsed time:	0.090000		0.090000	0.09		3	0.14	1.19	3.29	7.26	8.56	15.19	16.87	55.33
5	elapsed time:	0.150000		0.150000	0.15		4	0.09	1.28	3.76	6.25	14.39	8.53	28.05	55.22
6	elapsed time:	0.070000		0.070000	0.07		5	0.15	1.13	3	4.81	7.12	15.5	27.77	55.13
7	elapsed time:	0.130000		0.130000	0.13		6	0.07	1.45	3.5	7.42	9.59	8.64	30.31	55.94
8	elapsed time:	0.140000		0.140000	0.14		7	0.13	1.77	3.56	3.8	14.59	14.82	55.88	29.14
9	elapsed time:	0.160000		0.160000	0.16		8	0.14	2.01	3.64	4.59	8.69	14.59	28.04	28.73
10	elapsed time:	0.060000		0.060000	0.06		9	0.16	1.6	3.06	5.18	16.77	8.87	11.58	57
11	elapsed time:	1.060000		1.060000	1.06		10	0.06	1.66	3.58	8.21	17.16	15.47	28.44	16.84
12	elapsed time:	1.390000		1.390000	1.39		MAX	0.16	2.01	4	8.21	17.16	30.48	55.88	57
13	elapsed time:	1.190000		1.190000	1.19		MAX2	0.15	1.77	3.9	7.42	16.77	30.22	55.21	55.94
14	elapsed time:	1.280000		1.280000	1.28		AVEAGE	0.10125	1.345	3.42375	5.3825	10.72625	12.70125	23.2975	35.7325
15	elapsed time:	1.130000		1.130000	1.13		동시 처리율	987.654321	743.4944238	730.1935013	743.1490943	512.7607505	551.1268576	364.8460135	279.8572728
16	elapsed time:	1.450000		1.450000	1.45										
17	elapsed time:	1.770000		1.770000	1.77										
18	elapsed time:	2.010000		2.010000	2.01										
19	elapsed time:	1.600000		1.600000	1.6										

1. Task1, Task2 동시 처리율 비교

앞서 예상한 결과가 맞는지 확인하기 위해 실험을 진행하였다. 주식 수 5개, ORDER PER CLIENT 10개를 동일하게 세팅하고, Task2의 경우 NTHREAD 20개, SBUF 1000개를 기준으로 평가하였다. 결과는 아래와 같다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
Task1	0.305	2.251	3.89	6.228	44.76	15.098	19.198	17.536
Task2	0.084	1.003	2.662	4.0137	6.056	7.798	9.661	12.353

2) 동시 처리율 (초당 처리량)

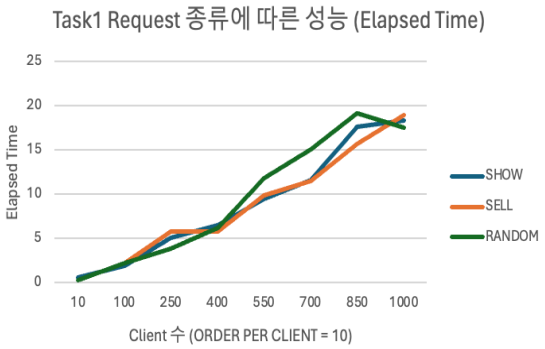
Client 수	10	100	250	400	550	700	850	1000
Task1	327.868	444.197	642.673	642.183	122.877	463.614	442.737	570.247
Task2	1194.02	996.264	938.967	996.574	908.152	897.579	879.803	809.470

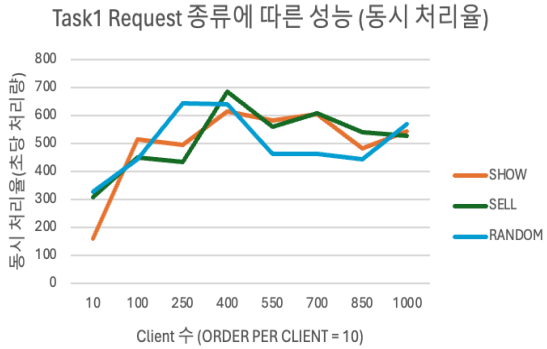
Task1보다 Task2가 더 우수한 성능을 보였고, 이는 앞에서 멀티 코어에서 Task2가 더 뛰어날 것이라고 예상했던 결과와 일치했다. 동시 처리율을 비교하면 Task2가 눈에 뛰게 더 좋은 성능을 가진다는 것을 확인할 수 있었다. 다만, Task1에서 Client 수가 850개일 때1000개일 때보다 더 오랜 시간이 걸렸는데 이는 서버 상황으로 인한 오

차일 것으로 예상된다.

2. Task1의 요청 종류에 따른 동시 처리율 비교

종류가 다르더라도 각 명령이 해야 하는 일이 유사하기 때문에 (자료구조에서 원하는 데이터에 접근하고 출력, 또는 값을 변경) 종류에 관계없이 비슷한 결과를 보일 것이라 예상하고 실험을 진행하였다. BUY와 Sell은 거의 동일한 동작을 하기 때문에 실험에서는 Buy는 진행하지 않았다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
SHOW	0.209	1.536	3.854	6.786	12.189	15.074	16.591	18.713
SELL	0.322	2.228	5.765	5.821	9.825	11.495	15.729	18.972
RANDOM	0.305	2.251	3.89	6.229	11.85	15.098	19.199	17.536

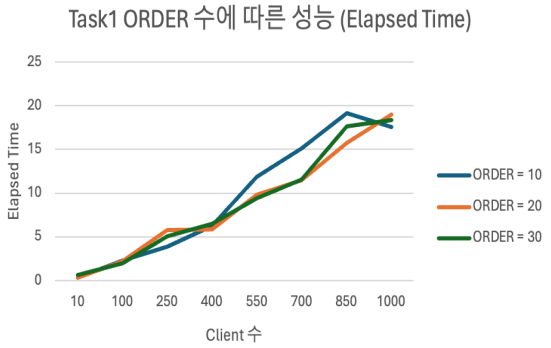
2) 동시 처리율 (초당 처리량)

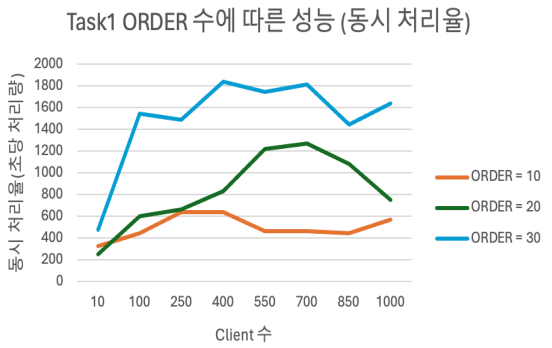
Client 수	10	100	250	400	550	700	850	1000
SHOW	479.042	650.935	648.719	589.4271	451.2358	464.383	512.318	534.366
SELL	310.077	448.933	433.651	687.138	559.796	608.960	540.412	527.079
RANDOM	327.868	444.197	642.673	642.183	464.135	463.615	442.737	570.247

앞선 예상에서는 명령의 종류와 상관없이 비슷한 결과를 보일 것으로 예상했고, 조금의 편차가 있긴 하지만 이는 실험 당시 서버 환경에 따라 생길 수 있는 정도의 오차 볼 수 있다고 생각한다. 다시 말해, 처음의 예상한 결과와 동일하게 결과를 얻을 수 있었다.

3. Task1의 ORDER PER CLIENT 수에 따른 동시 처리율 비교

ORDER PER CLIENT가 변하면 전체 elapsed time은 증가하더라도 클라이언트 개수 당 동시 처리율은 비슷한 값을 유지할 것이라 예상하고 실험을 진행하였다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
Order 10	0.305	2.251	3.89	6.229	11.85	15.099	19.199	17.536
Order 20	0.323	2.228	5.765	5.821	9.825	11.495	15.729	18.973
Order 30	0.626	1.939	5.03	6.515	9.453	11.591	17.646	18.329

2) 동시 처리율 (초당 처리량)

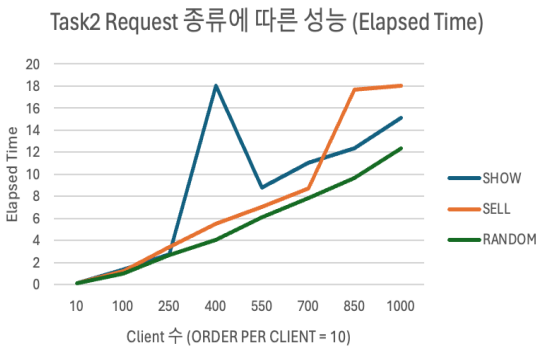
Client 수	10	100	250	400	550	700	850	1000
Order 10	327.869	444.198	642.674	642.183	464.135	463.615	442.737	570.247
Order 20	254.372	600.15	664.894	832.25	1220.189	1270.129	1085.655	753.047
Order 30	479.042	1547.389	1491.054	1841.903	1745.57	1811.711	1445.066	1636.773

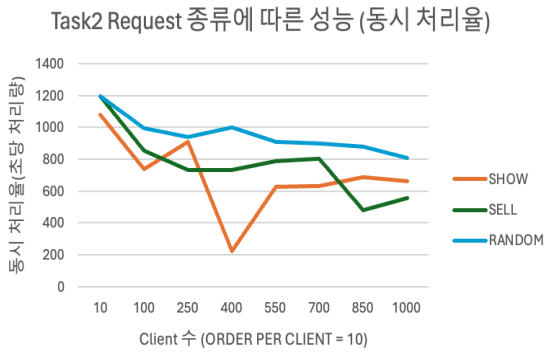
앞선 예상과 달리 실제 결과에서는 ORDER PER CLIENT가 차이가 나더라도 실행 시간에는 큰 차이가 없었고, 따라서 동시 처리율에서 ORDER PER CLIENT가 클수록 더 성능이 뛰어난 결과를 보였다.

결과의 원인을 추측하자면, 첫번째로 실험 변수인 ORDER PER CLIENT에 10, 20, 30을 각각 할당하여 실험했는데 이는 큰 차이가 아니기 때문에 실행시간에 큰 차이가 없었을 것이라 예상된다. 따라서, 다음에 실험을 한다면 더욱 더 차이가 날 수 있도록 1000개 단위로 차이를 두어 실험을 진행해 보아야 알 수 있을 것 같다. 또한, 실제 실행 시간에 큰 부분을 차지하는 것은 클라이언트의 연결을 관리하는 부분이 더욱 큰 영향을 미치는 변수라서 이 정도의 주문 수 차이로는 유의미한 결과를 볼 수 없었을 것이라 생각한다.

4. Task2의 요청 종류에 따른 동시 처리율 비교

2번의 실험과 동일한 이유로 실험을 진행하였다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
SHOW	0.093	1.358	2.759	18.058	8.78	11.063	12.338	15.104
SELL	0.084	1.171	3.406	5.479	6.999	8.728	17.639	18.066
RANDOM	0.084	1.004	2.663	4.014	6.056	7.799	9.661	12.354

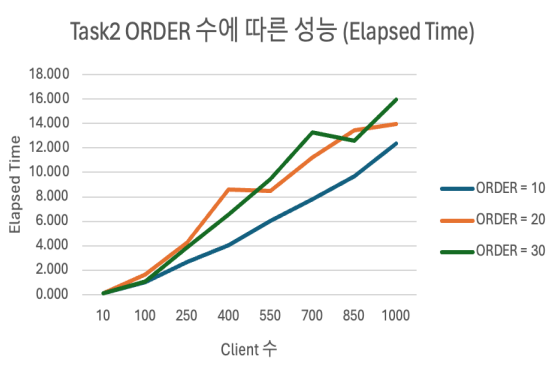
2) 동시 처리율 (초당 처리량)

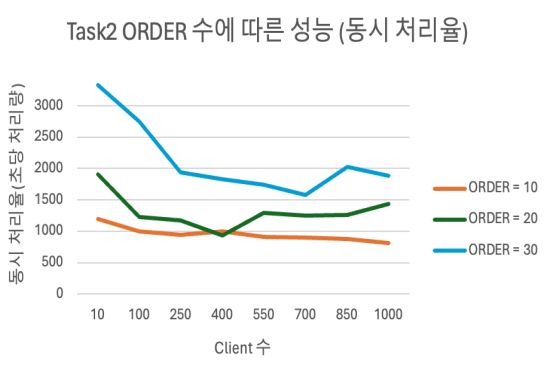
Client 수	10	100	250	400	550	700	850	1000
SHOW	736.648	906.208	221.515	626.424	632.768	688.956	662.087	736.648
SELL	853.789	733.945	730.094	785.855	802.062	481.894	553.518	853.789
RANDOM	996.264	938.967	996.574	908.153	897.58	879.803	809.471	996.264

2번의 실험과 동일하게 예상했고 실제 결과에서도 클라이언트의 개수에 따라 성능의 우위가 조금씩 달라지는 것으로 봐서, 예상과 일치하는 결과라고 볼 수 있겠다. 다만, SHOW를 실험할 때 400개의 클라이언트에서 눈에 뛰게 실행 시간이 길어진 것을 볼 수 있는데, 아마 실험 시 서버 환경 이슈로 값이 눈에 뛰게 차이가 나지 않았을까 예상한다. 다만, 동시 처리율을 보면 클라이언트의 수가 많아질 수록 동시 처리율이 조금씩 우하향 하는 것을 볼 수 있는데, 이는 아마도 CPU 코어의 수가 제한되어 있기 때문에 클라이언트 수가 많아질수록 waiting 해야하는 스레드의 수가 늘어나서 동시 처리율도 떨어진 것이라고 예상된다.

5. Task2의 ORDER PER CLIENT 수에 따른 동시 처리율 비교

3번의 실험과 동일한 이유로 실험을 진행하였다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
Order 10	0.084	1.004	2.663	4.014	6.056	7.799	9.661	0.084
Order 20	0.105	1.63	4.25	8.584	8.494	11.2	13.436	0.105
Order 30	0.09	1.093	3.861	6.578	9.455	13.304	12.579	0.09

2) 동시 처리율 (초당 처리량)

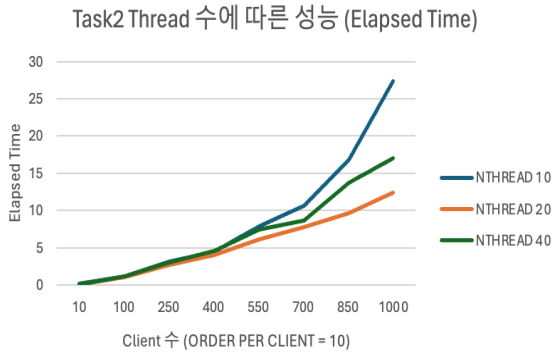
Client 수	10	100	250	400	550	700	850	1000
Order 10	1194.03	996.264	938.967	996.574	908.153	897.58	879.803	809.471
Order 20	1904.762	1226.994	1176.471	931.994	1295.07	1250	1265.234	1433.435
Order 30	3333.333	2745.995	1942.376	1824.401	1745.108	1578.502	2027.228	1880.583

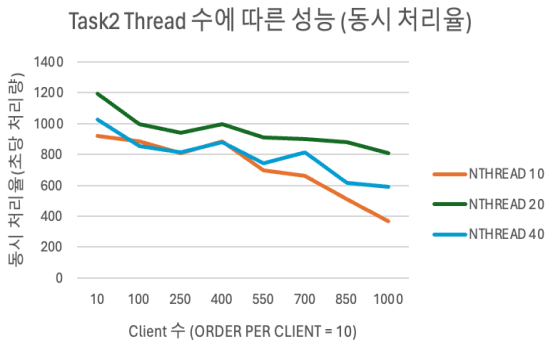
처음에는 3번 실험 전과 마찬가지로 동시 처리율에 큰 차이가 없을 것이라 예상했지만 3번의 실험 후에는 3번과 동일한 결과를 가질 것이라 예상했고, 실제로 3번과 유사한 결과를 볼 수 있었다. 실행 시간에는 큰 차이가 없고, 따라서 동시 처리율이 Order 수가 많을 수록 더 성능이 뛰어났고, 이는 3번과 동일한 이유로 예상한다.

4번의 실험과 마찬가지로 Client 수에 따라 성능이 저하되는 모습을 보였고, 이는 4번에서 설명한 이유와 동일할 것으로 예상한다.

6. Task2의 NTHREAD 수 변화에 따른 동시 처리율 비교

Task2에서 NTHREAD의 수가 많을수록 더 성능이 좋을 것이라 예상했고, 각각 NTHREAD 10, 20, 40개를 만들어 실험해보았다. 이를 제외한 다른 조건 (SBUF SIZE = 1000, ORDER PER CLIENT = 10, Stock 종류 = 5)은 동일하게 세팅하고 실험을 진행하였다.





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
NTHREAD 10	0.11	1.13	3.09	4.51	7.91	10.59	16.75	27.42
NTHREAD 20	0.08	1	2.66	4.01	6.06	7.8	9.66	12.35
NTHREAD 40	0.1	1.17	3.07	4.56	7.41	8.6	13.76	17

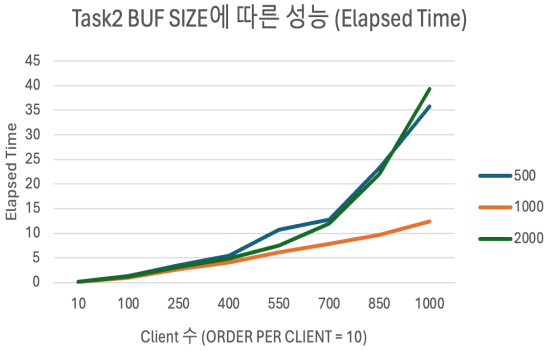
2) 동시 처리율 (초당 처리량)

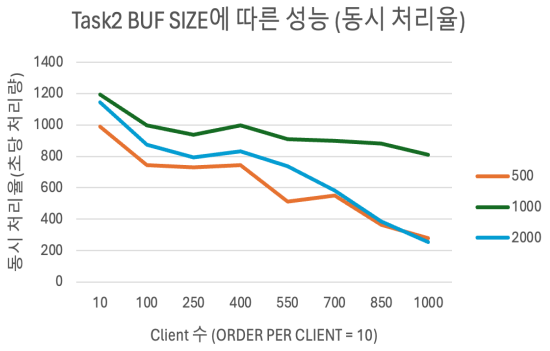
Client 수	10	100	250	400	550	700	850	1000
NTHREAD 10	919.54	885.94	809.72	886.92	695.76	661	507.58	364.73
NTHREAD 20	1194.03	996.26	938.97	996.57	908.15	897.58	879.8	809.47
NTHREAD 40	1025.64	853.79	813.34	878.16	742.62	813.84	617.68	588.32

Thread의 수가 많을 수록 성능이 더 좋을 것이라 예상했는데, 예상과는 조금 다른 결과가 나왔다. Thread가 10개일 때에 비해서는 20개와 40개일 때 더욱 뛰어난 성능을 보여준 것은 맞지만, 20개일 때가 40개보다 더욱 뛰어난 성능을 가지는 것을 확인할 수 있었다. 이러한 결과의 원인을 예상해보자면, 실제 CPU의 코어 수 제한이 있기 때문에 NTHREAD를 크게 한다고 해서 Parallel하게 실행할 수 있는 Thread의 수는 제한되어서 불필요하게 많은 Thread를 가졌기 때문에 이러한 결과를 보이는 것이 아닐까 예상한다.

7. Task2의 SBUF SIZE 변화에 따른 동시 처리율 비교

6에서 실험한 것과 마찬가지로 SBUF의 SIZE가 클수록 더 성능이 좋을 것이라 예상했고, NTHREAD = 20으로 한 상태에서 각각 버퍼의 사이즈를 500, 1000, 2000으로 정하여 실험을 진행했다.



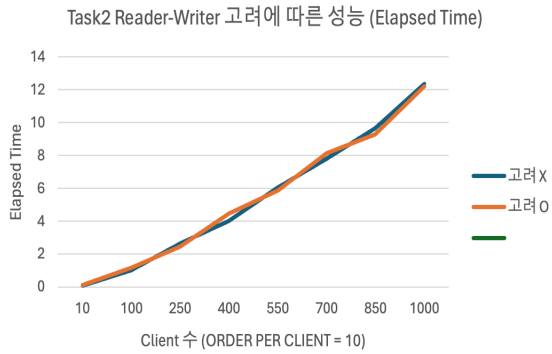


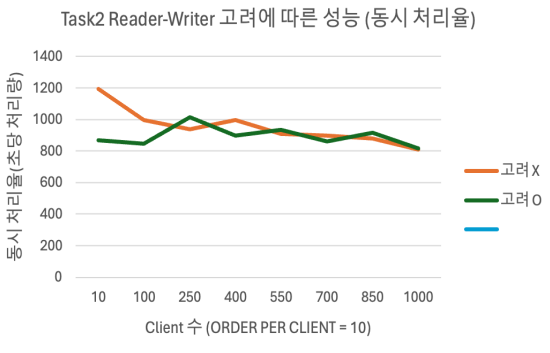
결과는 예상과 달리 1000개일때 가장 뛰어난 성능, 그리고 500개와 2000개에서 성능이 비슷하게 나왔다. 결과의 원인을 예상해보자면 6번의 결과와 마찬가지로 일정 버퍼 크기 이상으로 커지면 하드웨어 적인 제약 때문에 오히려 오버로드가 걸려 성능이 떨어지지 않았을까 예상된다. 다음에 실험을 한다면 더욱 변수 크기의 간격을 좁혀서 성능이 어떻게 변하는지 확인해 볼 필요가 있을 것 같다.

8. Task2의 Reader-Writer 구현 유무에 따른 동시 처리율 비교 (Show 명령어 기준)

기존에 구현한 방식은 reader-writer 문제를 고려하지 않고 진행했는데 이를 고려한 프로그램과 비교했을 때 더욱 성능이 낮을 것으로 예상되었다. 특히 Show 명령어만 들어오는 상황에서 해당 문제를 고려하지 않은 프로그램은 다른 명령어와 동일하게 다른 스레드에서 접근하지 못하도록 막아놓지만, 해당 문제를 고려한 프로그램은 다른 스레드의 접근을 막지 않기 때문에 실험 환경을 show 명령어만 들어오도록 세팅하면 성능의 차이를 확인할 수 있을 것이라 예상했고 실험을 진행했다.

(NTHREAD = 20, SBUF SIZE = 1000)





1) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000

고려 X	0.08	1	2.66	4.01	6.06	7.8	9.66	12.35
고려 O	0.12	1.18	2.46	4.45	5.88	8.12	9.27	12.21

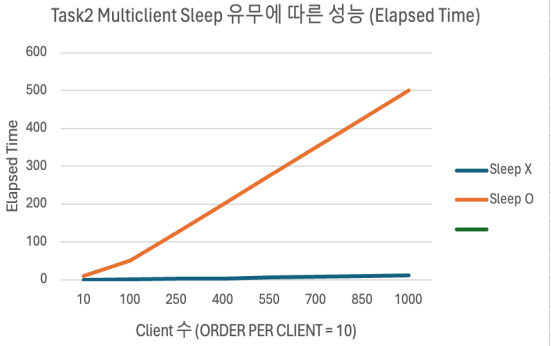
2) 동시 처리율 (초당 처리량)

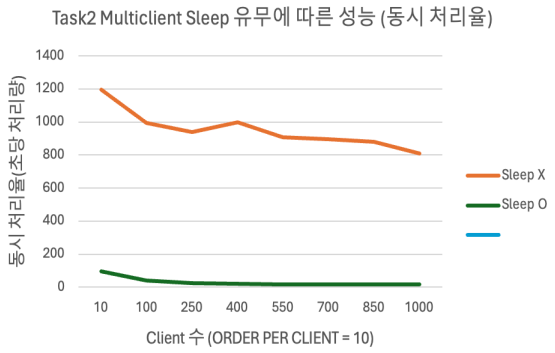
Client 수	10	100	250	400	550	700	850	1000
고려 X	1194.03	996.26	938.97	996.57	908.15	897.58	879.8	809.47
고려 O	869.57	845.67	1015.74	898.12	934.78	861.94	916.57	818.75

실제 예상과 다르게 큰 차이를 보이지 않았다. 이는 예상컨데, Reader-Writer Problem이 일어나려면 Critical Section에 딱 알맞는 타이밍에 접근해야 해당 문제로 waiting하는 스레드가 생기는데 클라이언트 1000개 이하, 클라이언트 당 요청 수 10개인 환경에서는 이러한 충돌이 많이 있지 않기 때문에 결과에 큰 차이를 보이지 않은 것이라 예상한다.

9. Task2의 Multiclient의 Sleep 유무에 따른 동시 처리율 비교

Task2는 Thread를 이용한 방식으로, NTHREAD의 수가 성능에 큰 영향을 미친다고 생각했다. 한편, Multiclient에서 sleep은 하나의 클라이언트가 요청을 하는데 시간 텀을 만들어주기 위해 존재하는데, 이 sleep의 여부에 따라 동시에 연결을 요청하는 클라이언트 수에 차이가 발생할 수 있을 것이라 예상했고, 따라서 결과에 차이를 볼 수 있을 것이라 예상해 이러한 실험을 진행했다. (NTHREAD = 20, BUF SIZE = 1000)





3) 시간 (단위: 초)

Client 수	10	100	250	400	550	700	850	1000
Sleep X	0.08	1	2.66	4.01	6.06	7.8	9.66	12.35
Sleep O	10.25	50.52	125.42	200.39	275.4	350.39	425.41	500.42

4) 동시 처리율 (초당 처리량)

Client 수	10	100	250	400	550	700	850	1000
Sleep X	1194.03	996.26	938.97	996.57	908.15	897.58	879.8	809.47
Sleep O	97.55	39.59	23.92	19.96	18.16	17.12	16.45	15.99

예상했던 것처럼 성능에서 매우 큰 차이를 확인할 수 있었다. 결과의 원인을 더욱 구체적으로 추론해보자면, sleep에 의해서 동시에 접속을 요청하는 클라이언트의 수가 증가한다. 이때 Thread의 수는 제한되어 있기 때문에 동시에 접속하는 클라이언트가 많아질 수록 wait하는 클라이언트의 수가 더욱 많이 늘어나기 때문에 성능에서 매우 큰 차이를 보이지 않았을까 예상한다.

10. 정리

예상과 유사한 결과를 경우도 있었고, 그렇지 않은 경우도 있었다. 예상과 다른 결과의 원인으로는 예측이 잘못된 경우도 있었지만 이외에도 실험 파라미터 값 설정의 문제(변수값 사이의 너무 작은 간격, 또는 너무 큰 간격), 하드웨어적인 제약 (CPU 코어 수 등), 서버 환경의 문제가 있었다. 만약 다시 실험을 진행하게 된다면 이러한 것들을 고려하여 쾌적한 서버 환경에서 다시 실험을 진행해 볼 필요가 있을 것이다.