System Programming Project 3

담당 교수 : 김영재

이름 : 윤준서

학번 : 20211558

1. 개발 목표

3rd 프로젝트에서 구현할 주제는 "concurrent 주식 서버 개발"이다. 순차적으로 처리를 할 경우 사용자의 대기 시간이 길어지고 서버의 자원을 낭비하게 된다. 이를 해결하기 위해 하나의 서버에서 여러 클라이언트를 동시적으로(concurren) 처리하도록 한다.

해당 프로젝트는 동시성 구현을 Event-driven 방법과 Thread-driven 방법 두 가지로 구현한다. 그리고 클라이언트 수와 요청에 따른 두 구현의 성능 차이를 비교한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

주식 정보를 저장하는 자료를 binary search tree 로 구현한다.

서버에 요청하는 show, buy, sell, exit 메세지 처리를 구현한다.

종료 시 주식 정보를 stock.txt 에 새로 저장한다.

fd_set 과 select 함수를 통해 Event-driven 방법으로 concurrent 를 구현한다.

2. Task 2: Thread-based Approach

주식 정보를 저장하는 자료를 binary search tree 로 구현한다.

서버에 요청하는 show, buy, sell, exit 메세지 처리를 구현한다.

종료 시 주식 정보를 stock.txt 에 새로 저장한다.

semaphore 와 thread 함수를 통해 Thread-driven 방법으로 concurrent 를 구현한다.

3. Task 3: Performance Evaluation

위의 두 가지 방법으로 서버 구현을 완료 후, 각 방법을 여러 조건에서 각각의 성능을 구해 서로 비교한다. 조건은 클라이언트의 수, 요청의 수, 요청의 종류 등이 있으며 이를 각기 다르게 환경을 조성하여 두 구현의 성능을 구하고 서로 비교한다. 이를 통해 Event-driven 과 Thread-driven 방법 중 성능이 더 높은 방법이 무엇인지 조사한다. 그리고 Thread-driven 방법이 성능이 더 높은 것이 맞는지 확인한다.

B. 개발 내용

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

✓ **Multi-client 요청에 따른 I/O Multiplexing**

Event-driven 방법은 fd_set 을 통해 클라이언트를 관리한다. fd_set 에는 클라이언트와의 통신을 읽는 read_set 과, 연결을 통해 들어온 요청을 처리하는 ready_set 두 가지가 있다.

연결된 fd 는 clientfd[]에 저장한다. 그리고 FD_ISSET()을 통해 clientfd 와 ready_set 을 비교하여 요청이 들어왔는지 확인, 요청을 처리한다.

✓ **epoll과의 차이점**

• select 는 이벤트를 감지하기 위해 fd 를 모두 탐색하기에 시간 복잡도가 O(n)이지만, epoll 은 이벤트가 발생한 fd 를 바로 알려주기에 O(1)이다.

• select 는 FD_SETSIZE 에 제한이 있지만, epoll 은 커널 내에서 fd 를 동적으로 생성하기에 개수에 제한이 없다.

• select 는 호출마다 fd_set 을 커널 영역에 매번 복사해야 하지만, epoll 은 커널에 처음 등록한 상태를 유지하므로 복사 과정이 필요없다.

• select 는 상태가 바뀌지 않으면 계속 알려주기에 불필요한 시간이 쓰이지만, epoll 은 트리거를 통해 이벤트 처리를 효율적으로 할 수 있다.

이처럼 epoll 은 select 의 여러 단점들을 보완할 수 있는 방식이다. 따라서 epoll 은클라이언트의 수가 대규모, 고성능 네트워크 서버에 적합하다.

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

  ✓ **Master Thread의 Connection 관리**

  Master Thread 는 클라이언트의 요청(connection)을 Accept()를 통해 connfd 을 받아서 공유 저장소에 저장한다. 이때 저장 공간이 가득 차 있으면 Blocking 을 한다. 저장이 가능하면 공유 저장소에 connfd 를 저장하고 semaphore 를 늘린다.

  ✓ **Worker Thread Pool 관리**

  Worker Thread 는 공유 저장소를 확인해 처리할 수 있는 요청이 있으면 해당 요청을 처리한다. 코드에서 define 한 thread 개수 만큼 thread 를 생성하고 detach()를 실행한다. detach() 함수를 통해 thread 는 종료 시 자동으로 메모리를 free 한다.

  mutex 등의 semaphore 를 통해 thread 간의 충돌을 방지한다.

- **Task3 (Performance Evaluation)**

  ✓ **얻고자 하는 metric 정의와 이유, 측정 방법**

  서버를 구현하는 두 방법의 성능을 비교하기 위해 생각할 수 있는 간단한 평가 요소는 실행 시간이다. 따라서 시간 당 클라이언트의 요청 처리율을 통해 각 방법의 실행 시간을 구하고 이를 비교할 수 있다.

  시간 당 요청 처리율을 얻기 위해, 프로그램 실행 시간을 알려주는 스크립트를 따로 작성하여 실행한다.

  사전에 언급한 여러 조건은 다음과 같다.

  1. show, buy, shell 을 각각 하나만 반복해서 요청하는 경우
  2. 클라이언트의 수를 10, 100, 1000 개로 요청하는 경우
  3. worker thread 의 수를 10, 20, 40, 80, 160, 320 개로 처리하는 경우

  해당 조건들을 조합하여 다양한 환경을 조성할 수 있다.

  ✓ **Configuration 변화에 따른 예상 결과**

- **1. show, buy, shell을 각각 하나만 반복해서 요청하는 경우**

show 의 경우 두 방법의 차이는 없을 것이다. 하지만 buy, sell 의 경우 task2 에서 mutex semaphore 를 통해 stock 의 개수를 바로바로 바꾸기에 추가 연산 시간이 발생할 것이다.

- **2. 클라이언트의 수를 10, 50, 100, 500, 1000개로 각각 요청하는 경우**

task1 의 경우 클라이언트의 요청을 순차적으로 처리하기에, linear 하게 증가할 것이다.

task2 의 경우 Work Thread 의 개수에 따라 동시에 처리 가능한 클라이언트의 수가 결정되므로, 실행 시간은 Work Thread 의 개수에 반비례하게 결정될 것이다. 따라서 해당 조건은 task1 에서만 진행한다.

- **3. worker thread의 수를 10, 50, 100, 500, 1000개로 각각 처리하는 경우**

2 번째 조건에서 언급했듯이 worker thread 의 개수와 동시에 처리 가능한 클라이언트 수는 같다. worker thread 의 개수에 맞춰서 클라이언트 수가 그 이하로 조정된다면, 각 경우의 실행 시간은 유사할 것이다. 하지만 클라이언트 수가 대규모라면, 실행 시간은 worker thread 의 개수에 반비례하게 결정될 것이다. 해당 조건은 task2 에서만 진행한다.

C. 개발 방법

task1

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

item *root = NULL;

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

주식 정보를 저장하는 구조체 item을 binary search tree의 형태로 표현할 수 있게 정의한다. 그리고 root를 생성하여 헤더 역할을 부여한다.

구조체 pool은 Event-driven 방법을 구현하기 위해 사용한다.

```
void init_pool(int listenfd, pool *p)
void add_client(int connfd, pool *p)
void check_clients(pool *p)
```

다음 함수들은 여러 클라이언트를 Event-driven 방법으로 처리하기 위해 사용하는 함수들이다. check_clients의 경우, pool을 탐색하여 각 클라이언트의 요청 유무를 확인한다. 요청이 있을 경우, echo()를 통해 요청을 처리한다.

모든 클라이언트에 echo()를 완료하고 각 connfd를 close를 완료하면, 가장 마지막에 바뀐 전체 주식 정보를 stock.txt에 새로 저장한다.

```c
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)) {
        clientlen = sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = 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(*connfdp, &pool);
    }
    check_clients(&pool);
}
```

main 함수에서 클라이언트를 연결, pool에 connfd를 저장하는 과정이다. read_set으로 ready_set을 설정하고, ready_set의 원소가 활성화될 때까지 select에서 Blocking을 한다. 이후 listenfd에 connection 들어오면 accept를 한다. connection이 들어오지 않으면 check_client를 통해 클라이언트의 요청(echo)을 처리한다.

task2

```
typedef struct item {
    int ID, left_stock, price, readcnt;
    sem_t mutex, w;
    struct item *left, *right;
} item;

item *root = NULL;
sem_t file_mutex;
```

주식 정보를 저장하는 구조체 item을 binary search tree의 형태로 표현할 수 있게 정의한다. 그리고 root를 생성하여 헤더 역할을 부여한다. task1과 달리 몇 가지의 요소가 추가된 것을 알 수 있다.

mutex와 w, file_mutex는 공유 자원이 concurrent에 의해 값이 바뀌는 것을 막기 위한 semaphore 변수다.

```
int conn_queue[QUEUE_SIZE];
int queue_front = 0, queue_rear = 0;
sem_t mutex_queue, slots, items;
```

Worker Thread를 통해 여러 클라이언트를 분산해서 관리하도록 할 때 큐를 사용한다. 하나의 큐를 전역으로 선언하여 creat_thread를 통해 각 Thread들이 큐를 사용하는 방식이다.

```
void queue_init() {
    queue_front = queue_rear = 0;
    Sem_init(&mutex_queue, 0, 1);
    Sem_init(&slots, 0, QUEUE_SIZE);
    Sem_init(&items, 0, 0);
}
```

```c
void enqueue(int connfd) {
    P(&slots);
    P(&mutex_queue);
    conn_queue[queue_rear] = connfd;
    queue_rear = (queue_rear + 1) % QUEUE_SIZE;
    V(&mutex_queue);
    V(&items);
}
```

```c
int dequeue() {
    int connfd;
    P(&items);
    P(&mutex_queue);
    connfd = conn_queue[queue_front];
    queue_front = (queue_front + 1) % QUEUE_SIZE;
    V(&mutex_queue);
    V(&slots);
    return connfd;
}
```

Worker Thread가 사용하는 큐를 초기화, 원소 관리를 하도록 하는 함수들이다. C언어의 경우 별도의 자료구조 라이브러리가 존재하지 않기 때문에 push, pop 등의 기능들을 직접 설정해야 한다.

```c
void *worker(void *vargp) {
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = dequeue();
        char buf[MAXLINE];
        rio_t rio;
        int n;
        Rio_readinitb(&rio, connfd);
        while ((n = Rio_readlineb(&rio, buf, MAXLINE)) > 0) {
            printf("server received %d bytes\n", n);
            fflush(stdout);
            if (!strcmp(buf, "exit\n"))
                break;
            echo(connfd, buf);
        }
        P(&file_mutex);
        write_stock("stock.txt");
        V(&file_mutex);
        Close(connfd);
    }
    return NULL;
}
```

클라이언트를 Thread-driven 방법으로 처리하기 위해 사용하는 worker() 함수다. 클라이언트의 요청 유무를 확인하여 요청이 있을 경우, echo()를 통해 요청을 처리한다.

클라이언트에 echo()를 완료하고, connfd를 close를 완료하여 마무리한다.

이 때 semaphore를 통해 concurrent 중 공유 자원의 변경을 막을 수 있는 P()와 V() 함수를 사용해서 여러 thread가 각각 다룬 주식 정보를 stock.txt에 최신화한다.

```
queue_init();

for (int i = 0; i < NWORKERS; i++) {
    Pthread_create(&tid, NULL, worker, NULL);
}

listenfd = Open_listenfd(argv[1]);
while (1) {
    clientlen = sizeof(clientaddr);
    int 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);
    enqueue(connfd);
}
```

main 함수에서 클라이언트를 연결, 여러 thread를 생성하는 과정이다. task1과 달리 semaphore을 초기화한 후 Pthread_create() 함수를 통해 thread를 생성한다. 이후 연결된 클라이언트의 connfd를 큐에 삽입한다. 그럼 worker()에서 큐의 내용을 꺼내어 요청 처리를 진행한다.

공통 자료구조 & 함수

```
item *arr[100];
int arr_idx = 0;
item *root = NULL;
```

이진 탐색 트리 이외에도 출력을 위해 item의 주소를 저장하는 포인터 배열을 선언한다.

```
item* new_item(int ID, int left_stock, int price) {
    item *node = (item*)Malloc(sizeof(item));
    node->ID = ID;
    node->left_stock = left_stock;
    node->price = price;
    node->left = node->right = NULL;
    return node;
}
```

한 종류(ID)의 주식을 item 구조체에 맞게 생성한다. task2는 Sem_init()을 통해 semaphore 변수인 mutex와 w를 초기화한다.

```c
item* insert_item(item *node, int ID, int left_stock, int price) {
    if (node == NULL) {
        item *NEW = new_item(ID, left_stock, price);
        arr[arr_idx++] = NEW;
        return NEW;
    }
    if (ID < node->ID) node->left = insert_item(node->left, ID, left_stock, price);
    else if (ID > node->ID) node->right = insert_item(node->right, ID, left_stock, price);
    return node;
}
```

new_item()을 통해 생성한 주식 item을 binary search tree에 삽입한다. 해당 프로젝트에서는 주식 ID를 숫자 크기 순서대로 저장한다. 이때 포인터 배열에 함께 item을 저장한다. 이를 통해 stock.txt에 작성한 순서를 유지하는 주식 정보를 저장할 수 있다.

```c
void load_stock(char *filename) {
    FILE *fp = fopen(filename, "r");
    if (!fp) return;
    int id, left, price;
    while (fscanf(fp, "%d %d %d", &id, &left, &price) != EOF) {
        root = insert_item(root, id, left, price);
    }
    fclose(fp);
}
```

stock.txt에 저장된 주식 정보를 binary search tree에 모두 저장한다. 이때 insert_item()을 사용한다.

```c
void write_stock(char *filename) {
    FILE *fp = fopen(filename, "w");
    if (!fp) return;
    for (int i = 0; i < arr_idx; i++) fprintf(fp, "%d %d %d\n", arr[i]->ID, arr[i]->left_stock, arr[i]->price);
    fclose(fp);
}
```

프로그램을 실행하면서 변경된 주식 정보를 stock.txt에 새로 저장한다.

```c
void show_stock(int connfd, char *buf) {
    for (int i = 0; i < arr_idx; i++) {
        char line[MAXLINE];
        sprintf(line, "%d %d %d\n", arr[i]->ID, arr[i]->left_stock, arr[i]->price);
        strcat(buf, line);
    }
}
```

show 명령어를 입력하면 실행되는 함수로, 포인터 배열에 저장된 주식 정보를 순서대로 출력한다. 이는 stock.txt 내용의 원본 순서를 유지하면서 출력할 수 있다.

```
item* search_item(item *node, int ID) {
    if (!node) return NULL;
    if (ID < node->ID) return search_item(node->left, ID);
    else if (ID > node->ID) return search_item(node->right, ID);
    return node;
}
```

buy, sell 명령어를 통해 주식 상품을 다룰 때, 주식 정보에 존재하지 않는 상품을 입력할 수 있다. 이에 입력한 주식 상품이 실제 주식 정보에 존재하는지 확인하기 위한 함수다.

```
void echo(int connfd, char *cmd)
```

클라이언트의 요청, 즉 명령어를 처리하는 함수다. cmd가 show, buy, sell, exit 중 하나인지 판별하고 각각에 알맞은 과정을 실행한다. 마지막 부분에 클라이언트에 end₩n을 보내 계속해서 요청을 보낼 수 있도록 한다.

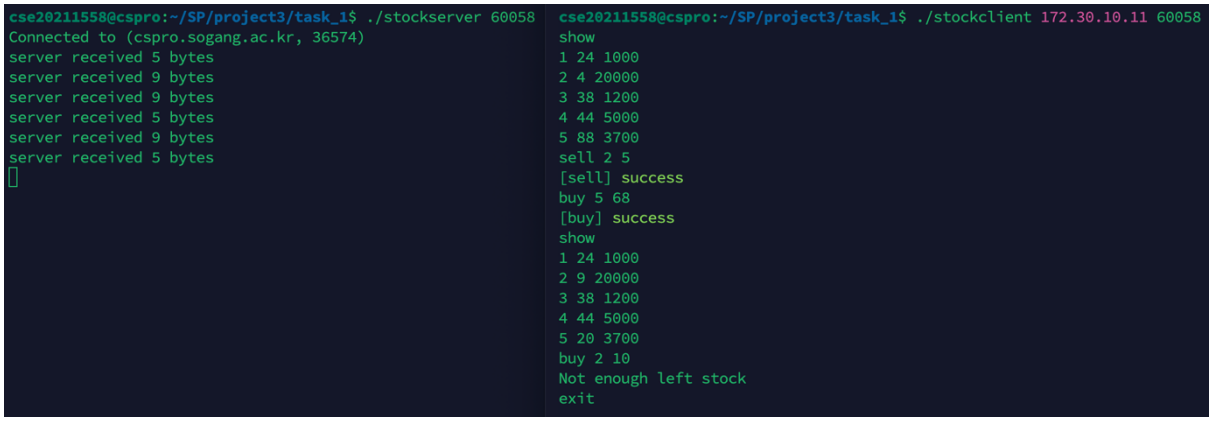
show : show_stock()을 실행하여 주식 정보를 출력한다.

buy : 해당 상품이 존재하는지, 구매 가능한 개수인지 확인 후 구매 여부를 결정한다.

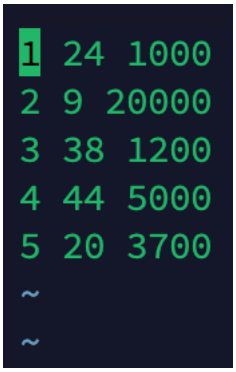
sell : 해당 상품이 존재하는지 확인 후 판매 여부를 결정한다.

exit : 클라이언트의 연결을 종료한다.

3. 구현 결과

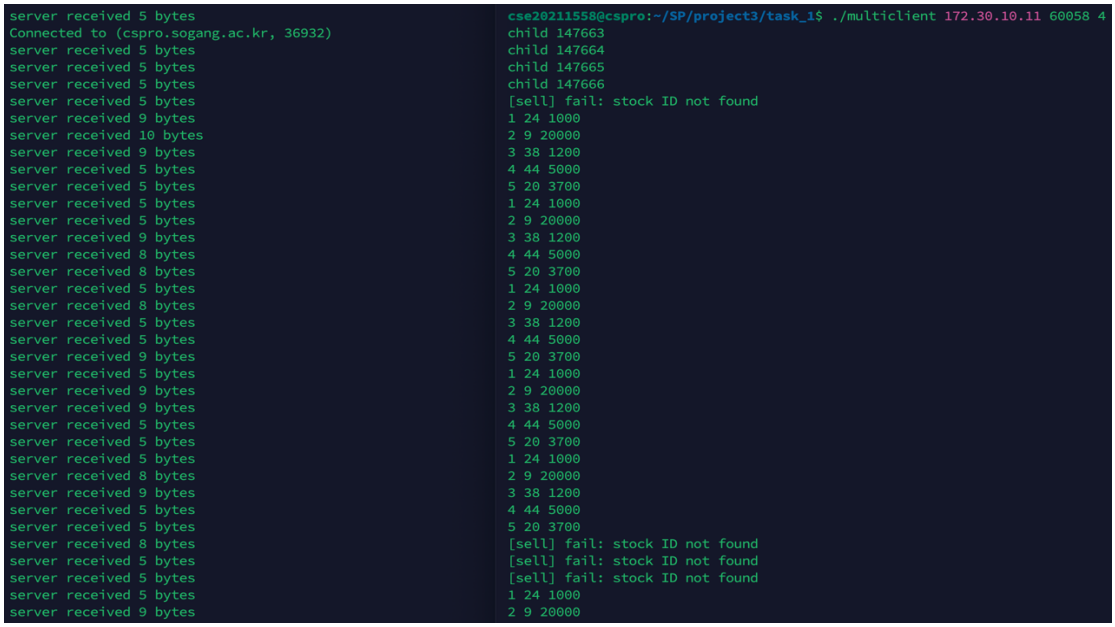


하나의 클라이언트를 서버에 연결하고 요청을 보내는 과정이다. 모든 명령어가 서버에 올바르게 전달되고, 기능도 올바르게 작동한다. 사고자 하는 주식 수가 남은 수를 초과하면 Not enough left stock 을 출력한다.

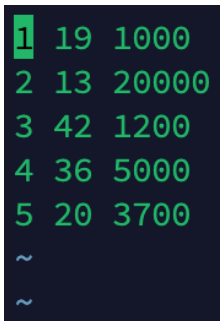


exit 를 통해 연결을 종료한 후 stock.txt 내용을 보면 최신화 된 것을 알 수 있다. 처음에 stock.txt 에 ID 순서 관계없이 작성했더라도 inorder 을 통해 정렬이 된 후 최신화 된 것을 확인할 수 있다.

일련의 과정은 task1 과 task2 모두 같은 결과가 나온다. 단일 클라이언트만을 다루기에 concurrent 구현의 차이가 결과에 거의 반영되지 않기 때문이다.



multiclient.c 를 통해 여러 개의 클라이언트를 생성해서 서버에 요청을 보내는 과정이다. 모든 명령어가 서버에 올바르게 전달되고, 기능도 올바르게 작동한다. 사고자 하는 주식 수가 남은 수를 초과하면 Not enough left stock 을 출력한다. 사거나 팔고자 하는 주식 ID 가 주식 정보에 존재하지 않으면 [sell(buy)] fail : stock ID not found 를 출력한다.



exit 를 통해 연결을 종료한 후 stock.txt 내용을 보면 최신화 된 것을 알 수 있다. 처음에 stock.txt 에 ID 순서 관계없이 작성했더라도 inorder 을 통해 정렬이 된 후 최신화 된 것을 확인할 수 있다.

일련의 과정은 task1 과 task2 모두 같은 결과가 나온다. concurrent 를 구현한다는 목적이 같기 때문이다. 둘의 차이를 확인하기 위해선 다음 항목에서 두 방법의 성능을 측정해야 한다.

4. 성능 평가 결과 (Task 3)

앞서 성능 테스트를 진행할 환경을 다음 세 가지로 나누었다.

1 - show, buy, shell 을 각각 하나만 반복해서 요청하는 경우

2 - 클라이언트의 수를 10, 50, 100, 500, 1000 개로 각각 요청하는 경우

3 - Worker Thread 의 수를 10, 50, 100, 500, 1000 개로 각각 처리하는 경우

다음은 각 환경에 따라 진행한 성능 측정 과정과 결과다.

1 - show, buy, shell 을 각각 하나만 반복해서 요청하는 경우

각 명령마다 걸리는 수행 시간을 task1 과 task2 를 서로 비교한다.

클라이언트의 수는 100 으로, 요청 횟수는 10,000 번으로 진행한다. 이때 task2 의 Worker Thread 의 수는 클라이언트의 수와 똑같이 100 으로 설정한다. 측정 횟수는 각각 3 회로 평균을 내어 비교한다.

(측정 결과)

	수행 시간											
	show				buy				sell			
	1차시	2차시	3차시	평균	1차시	2차시	3차시	평균	1차시	2차시	3차시	평균
task_1	72	71	71	71.3	70	70	69	69.7	71	71	71	71
task_2	71	70	70	70.3	70	70	70	70	71	71	70	70.7

클라이언트 수 : 100

요청 횟수 : 10,000

Worker Thread 수 : 100

세 가지의 명령의 수행 시간은 비슷하게 측정됐다. task1 과 task2 의 차이 또한 비슷하지만, show 와 sell 명령에서 미세하게 task2 가 더 빠른 것을 알 수 있다. 반대로 buy 명령에서 미세하게 task1 이 더 빠르게 나왔다.

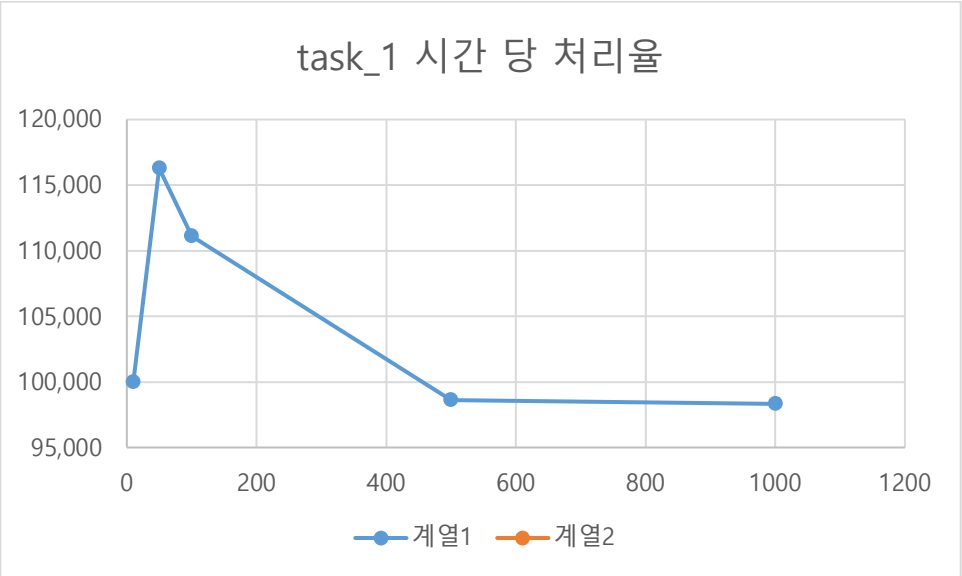
2 - 클라이언트의 수를 10, 50, 100, 500, 1000 개로 각각 요청하는 경우

각 클라이언트 수에 따라 걸리는 총 수행 시간을 케이스별로 나누어 task1 의 결과를 측정한다. 클라이언트 당 요청 횟수는 10,000 번으로 진행한다. 측정 횟수는 각각 3 회로 평균을 내어 비교한다. 그리고 평균으로 시간 당 처리율을 계산하여 각 클라이언트 수에 따라 비교한다.

(측정 결과)

task_1	수행시간			
클라이언트	1차시	2차시	3차시	평균
10	1	1	1	1
50	4	4	5	4.3
100	9	9	9	9
500	51	51	50	50.7
1000	102	102	101	101.7

task_1		
클라이언트	평균	시간 당 처리율
10	1	100,000
50	4.3	116,279
100	9	111,111
500	50.7	98,619
1000	101.7	98,323



클라이언트 수가 증가할 수록 시간 당 처리율이 감소하는 것을 확인할 수 있다. 클라이언트의 수가 10 인 경우 오히려 시간 당 처리율이 낮게 나왔는데, 데이터 측정 각 3 회만 진행했고, cspro 서버 환경에 따라 차이가 나는 것이 원인일 수 있다.

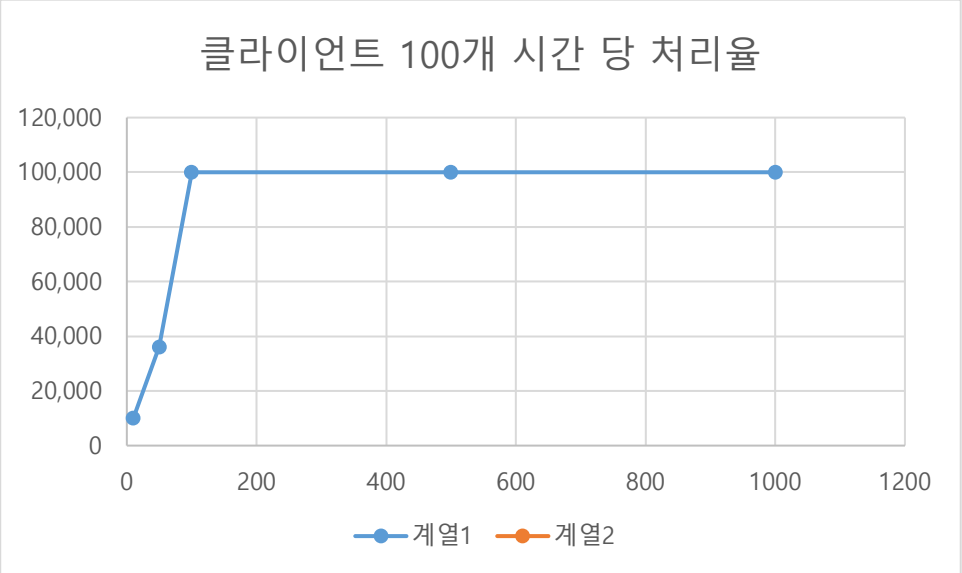
3 - Worker Thread 의 수를 10, 50, 100, 500, 1000 개로 각각 처리하는 경우

각 Worker Thread 수에 따라 걸리는 총 수행 시간을 케이스별로 나누어 task2 의 결과를 측정한다. 클라이언트 당 요청 횟수는 10,000 번으로 진행한다. 측정 횟수는 각각 3 회로 평균을 내어 비교한다. 그리고 평균으로 시간 당 처리율을 계산하여 각 클라이언트 수에 따라 비교한다.

(측정 결과 - 클라이언트 수 : 100)

task_2	클라이언트 100개 수행시간			
W Thread	1차시	2차시	3차시	평균
10	101	100	100	100.3
50	28	28	27	27.7
100	10	10	10	10
500	10	10	10	10
1000	10	10	10	10

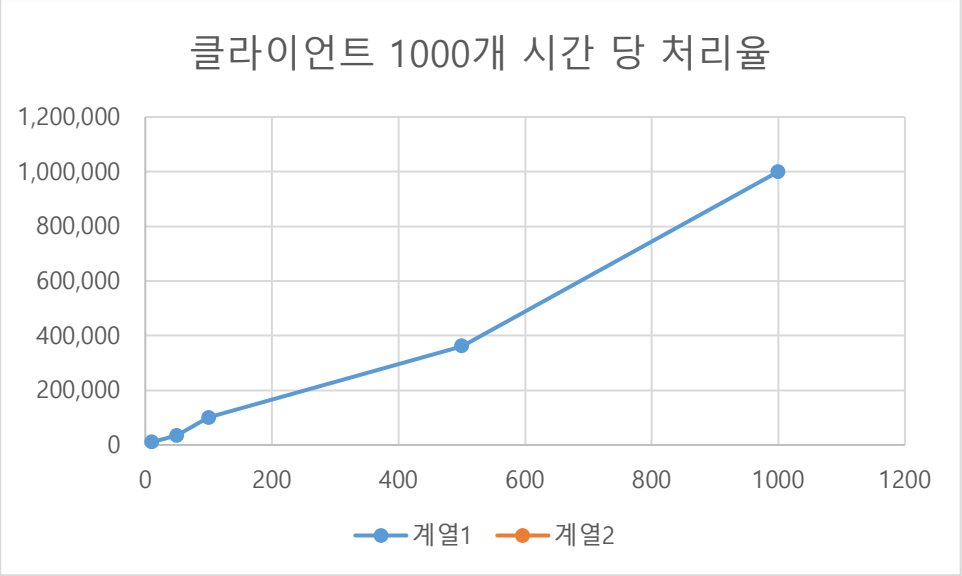
W Thread	평균	시간 당 처리율
10	100.3	9,972
50	27.7	36,101
100	10	100,000
500	10	100,000
1000	10	100,000



(측정 결과 - 클라이언트 수 : 1000)

task_2	클라이언트 1000개 수행시간			
W Thread	1차시	2차시	3차시	평균
10	1000	1001	1000	1000.3
50	283	284	282	283
100	100	100	100	100
500	28	27	28	27.7
1000	10	10	10	10

W Thread	평균	시간 당 처리율
10	1000.3	9,997
50	283	35,338
100	100	100,000
500	27.7	361,007
1000	10	1,000,000



Worker Thread 수가 증가할 수록 시간 당 처리율이 증가하는 것을 확인할 수 있다. 이는 서버의 동시 처리율이 Worker Thread 수에 비례함을 의미한다.

클라이언트 수가 100 인 경우, Thread Worker 는 100 개만 되어도 그 이상은 성능이 같을 수 밖에 없음을 확인할 수 있다. 이는 동시에 처리할 수 있는 최대 클라이언트 수가 Worker Thread 와 같기 때문이다.

결론

task1 과 task2 의 성능 차이는 유의미하게 측정되지 않았다. 또한 명령어의 종류에 따른 성능 차이도 유의미하게 측정되지 않았다.

1. 측정 횟수가 3 회
2. 주식의 규모가 크지 않음
3. 클라이언트, 요청의 규모가 크지 않음

위 3 가지의 이유가 원인이라 예상된다. 미세한 차이가 분명 있었으므로, 실험의 규모를 비약적으로 증가시키면 성능의 차이가 확연히 드러날 것이다.

Event-driven 방법은 클라이언트 수와 성능이 반비례하는 것을 알 수 있었다.

Thread-driven 방법은 Worker Thread 수와 성능이 비례하는 것을 알 수 있었다.