Multicore Programming Project 2

담당 교수 : 최재승

이름 : 이동석

학번 : 20191619

1. 개발 목표

이번 프로젝트의 주요 목표는 동시에 여러 클라이언트의 요청을 처리할 수 있는 concurrent한 주식 서버 구축이다. 기존에 배운 echo서버에서 기본 구조를 확장해 구현할 수 있다. Task1에서는 select 함수를 활용한 event-driven approach 방식으로 구현했으며, Task2에서는 pthread-library를 사용해 thread-based approach 로 구현한다. 마지막으로, Task3에서는 각 구현 방식에 따른 성능을 비교하고 평가한다.

주식 서버는 show, buy ,sell, exit 4가지 명령어를 수행하며, 사용자로부터 입력받는 명령어는 항상 올바른 형식으로 들어온다고 가정한다. 데이터는 stock.txt의 파일을 통해 주식의 고유 ID, 수량, 가격의 정보를 이진트리 형식으로 효율적으로 저장하고 관리 한다. 여러 client의 동시접속이므로 readers-writers problem 을 고려해 개발을 진행한다. 특히, thread-based에서는 semaphore를 활용해, race-condition, deadlock, starvation 과 같은 케이스에 유의하며 진행했다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Event 기반 서버는 I/O 멀티플렉싱을 활용했으며, select()함수로 하나의 스레드에서 다중 클라이언트의 요청을 동시처리하는 서버를 구현한다. 수업시간에 배운 서버 루프로 요청이 도착할 때마다 한번에 한줄씩 텍스트를 처리하도록 세밀하게 구현한다. select 함수로 다중 클라이언트의 입력인 파일 디스크립터를 모니터링 하고, 클라이언트의 정보를 pool 구조체에 저장해 관리한다. 단일 스레드로 요청을 처리하므로, readers-writers problem을 해결할 수 있다.

2. Task 2: Thread-based Approach

   Thread 기반 접근법에서는 각 클라이언트 연결에 대해 별도의 스레드를 생성한다. Pthread library를 활용하여 master thread가 클라이언트의 연결 요청을 관리하고, worker thread가 각 클라이언트의 요청을 처리하도록 구현한다. Task1과 마찬가지로 클라이언트의 요청을 동시에 처리할 수 있다. semaphore

를 활용하여 각 주식노드를 관리하고 이를 통해, readers-writers problem을 해결할 수 있다.

3. Task 3: Performance Evaluation

몇가지 분석 포인트를 설정하고, Task1과 Task2 각각의 방법에 대해 시간당 client 처리 요청 개수를 통해 성능을 평가하고 비교한다. 이때, gettimeofday 함수를 사용해 시간을 측정하며, multicilent 파일을 이용해 여러 상황을 가정하고 실험을 진행한다. client의 수, client의 요청 수, 워크로드 변경, thread 수와 sbuf 사이즈의 변경 등을 통해 concurrent 한 주식서버를 구현하기 위해 더 적합한 방법을 탐구한다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Event 기반 접근법에서는 앞서 말했듯 select 함수를 활용해 I/O Multiplxing을 처리한다. 개발 내용은 다음과 같다.

1. 서버 초기 설정
2. 클라이언트 연결대기 : select 함수를 통해 여러 소켓을 모니터링한다. 이때, read나 write 같은 유형의 이벤트를 모니터링하며, 이벤트가 발생할 시 반환한다.
3. 이제, 서버는 FD_ISSET을 통해 확인하고 accept 함수를 호출하여 연결을 수락한다.
4. 데이터를 처리하며 동시에 다른 클라이언트의 요청을 처리한다.
5. 만약 클라이언트가 연결을 종료하면, 소켓을 닫고 배열에서 제거한다.

✓ epoll과의 차이점 서술

select와 epoll은 모두 I/O Multiplexing에서 사용된다. 하지만, select 함수는 모든 file descriptor를 매번 확인하지만, epoll은 file descriptor의 상태변화만을 추적한다. 따라서, file descriptor의 수가 많아질수록 select 함수의 성능은 epoll에 비해 많이 저하될 수 있다. 이번 프로젝트의 경우 대량의 클라이언트 연결을 처리하지 않기 때문에 select로 진행해도 문제는 없을 것으로 예상된

다. 결론적으로, epoll은 리눅스 환경에서 select의 단점을 보완하여 만든 기법이다.

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

  ✓ Master Thread의 Connection 관리

  Thread 기반에서는 master thread가 클라이언트의 연결 요청을 관리하며, 요청이 들어오면 수락하고 worker thread로 클라이언트의 요청을 처리한다. 데이터 공유가 쉬워진다.

  1. 서버 초기 설정
  2. Master Thread 및 Worker Thread 설정 : 서버는 Master Thread를 설정하여 클라이언트의 연결 요청을 감지한다. 이후 Worker Thread Pool을 생성하여 해당 요청들을 처리한다.
  3. 동일하게 accept 함수로 연결을 수락하고 Worker thread에 할당한다. (sbuf 에 insert 함)
  4. 각 Worker thread는 독립적으로 요청을 수행하므로, 다른 클라이언트의 요청 역시 동시에 처리 가능하다.
  5. 클라이언트의 연결이 종료되면 반드시 Close(connfd)를 하여 메모리 누수를 방지하고 할당 되었던 Worker thread를 Thread Pool로 반환한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Wokrer Thread Pool은 클라이언트 요청을 처리하기 위한 Worker Thread의 집합으로 볼 수 있다. 각 Thread는 독립적으로 요청을 처리한다. 지정된 개수만큼 thread를 생성하고, 이때, 위에서 간략하게 설명했듯, sbuf 패키지를 활용해 관리하게 된다. Task1에서 pool구조체와 비슷한 역할을 수행하여 클라이언트의 파일 디스크립터를 저장하며 동시에 여러 스레드가 동시에 subf에 접근하는 동시성 문제를 관리한다. 이후, 연결이 종료되면 connfd를 close한다.

- **Task3 (Performance Evaluation)**

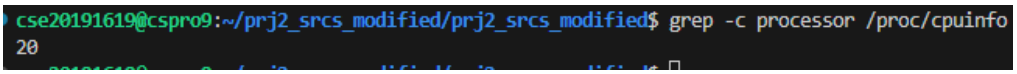
  ✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

  우선 분석 방법이 동시 처리율로 주어졌으므로, 시간당 client 처리 요청개수를 평가할 수 있는 metric을 정의해야한다. 우선, 동시 처리율은 해당 서버

프로그램이 동시에 처리할 수 있는 요청의 수를 의미한다. 따라서, metric을 다음과 같이 정한다.

동시 처리율 = (Throughput) / Time

즉, 동시 처리율은 (처리량) / 시간(s) 로 정의한다. 처리량은 client 수 * 각 클라이언트의 요청 수 로 정의한다. 기본적으로, task1의 event-driven은 하나의 스레드에서 수행하지만, thread-based는 동시성과 병렬성 모두 가질 수 있다.



현재 putty 서버에서 CPU 코어수는 20개이므로 20개를 기준으로 여러 값을 NTHREADS 값으로 하여 실험을 진행할 예정이다. sbuf 사이즈 역시 값에 영향을 줄 것이라 생각하지만, 1024로 놓고 진행한다.

클라이언트의 수가 1, 10, 100, 200, 300 일 때와 NTHREADS의 수가 10, 20, 40 일때 그리고, 각 워크로드에 따른 동시처리율을 측정하고 결과 그래프로 분석한다. 이는 muticlient 파일을 이용해 측정한다.

✓ Configuration 변화에 따른 예상 결과 서술

앞서 말했듯, Event-Based Approach 에서는 단일 스레드가 모든 클라이언트의 요청을 처리한다. CPU 를 병렬적으로 활용하는 데에 제한이 있다. 따라서 이 접근법은 CPU 코어의 수가 많을수록 동시처리율의 차이가 심할 것으로 예상된다.

반면에, Thread-Based Approach 에서는 각 클라이언트 요청을 처리하는 별도의 스레드를 생성한다. 이로 인해, CPU 의 여러 코어를 동시에 활용할 수 있어, 병렬 처리를 할 수 있다. 또한, 각 스레드는 독립적으로 실행되므로, 하나의 요청 처리가 다른 요청 처리에 영향을 미치지 않는다. 하지만, 스레드 생성과 관리에는 비용이 들며, readers-writers problem 을 위한 semaphore 의 관리를 하게 된다.

결론적으로, Event-Based Approach 는 Thread-Based 에 비해 더 낮은 성능을 보일 것으로 예상하며, 특히 CPU 코어의 수가 많은 시스템일 수록 이 차이가 더욱 명확해질 것이다.

그러나, show 와 같은 명령어로 수행할 경우 event-based 에서 더 좋은 성능을 보일 수도 있다. 각 노드를 읽을 때 마다 semaphore 를 관리하기 때문에 시간적인 측면에서 thread 가 더 오래걸릴 수 있을것이라 예상된다. 즉, 주어진 stock.txt 의 데이터의 수도 영향을 미칠 것이라 예상된다.

더불어, sbuf 의 사이즈가 너무 작으면, 스레드간의 동기화 비용이 높아질 수 있다. 예를 들어, 매우 작은 버퍼를 사용하면, 생산자나 소비자 스레드가 버퍼에 접근할 때마다 동기화를 위한 락(lock)을 획득하고 해제해야 하고, 이로 인해 오버헤드가 증가하며, 전체 시스템 성능이 저하될 수 있다. 따라서, 필요시 sbuf 의 사이즈도 조정해가며 실험할 필요가 있다.

C. 개발 방법

우선 task에서 공통적으로 주식 데이터를 저장 및 관리 하기 위한 이진트리가 필요하다. 아래 그림과 같은 구조체를 사용했으며, 각 이진트리의 왼쪽 자식노드와 오른쪽 자식노드의 연결을 위해 링크드 리스트 형식으로 구현했다. 추가적으로, 각 이진트리에 정보를 저장하기 위한, insert함수와 사용자가 원하는 주식을 찾기 위한 search 함수를 구현했다.

```c
/* 바이너리 트리 구현 부분 */
typedef struct item {
    int ID;
    int left_stock;
    int price;
    // int readcnt;
    // sem_t mutex;
    // thread에서 사용하므로 필요없음
}item;

typedef struct tree_node {
    item data;
    struct tree_node* l_node, *r_node;
}tree_node;

tree_node* root = NULL;
```

```c
tree_node* insert_tree(tree_node* node, item data) {
    if (node == NULL) {
        node = (tree_node*)malloc(sizeof(tree_node));
        node->data = data;
        node->l_node = NULL;
        node->r_node = NULL;
    } else if (data.ID < node->data.ID) {
        node->l_node = insert_tree(node->l_node, data);
    } else if (data.ID > node->data.ID) {
        node->r_node = insert_tree(node->r_node, data);
    }
    return node;
}

void free_(tree_node *node) {
    if (node == NULL) return;
    free_(node->l_node);
    free_(node->r_node);
    free(node);
    return;
}

// 탐색
tree_node* search_tree(tree_node* node, int id) {
    if (node == NULL || node->data.ID == id) {
        return node;
    } else if (id < node->data.ID) {
        return search_tree(node->l_node, id);
    } else {
        return search_tree(node->r_node, id);
    }
}
```

- Task 1

```
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;
```

위는 클라이언트의 정보를 저장하고 관리할 pool 구조체이다. 해당 구조체는, 최대 디스크립터 수를 나타내는 maxfd, active 디스크립터인 read_set과 ready_set, 그리고 준비된 디스크립터 수를 나타내는 n_ready (n_ready는 select 함수의 반환값이다.), cientfd는 디스크립터를 저장하는 배열이고 clientrio는 read buffer를 의미한다.

이제, listen함수로 클라이언트 요청을 대기하고, 해당 pool을 init_pool 함수로 초기화한다. 다음으로, 다음 Select 함수 호출에서 사용될 준비된 파일 디스크립터로 ready_set을 read_set의 값으로 할당한다. 이제, Select 함수를 호출하여, maxfd+1 만큼 파일 디스크립터의 수를 확인하고 감시될 파일 디스크립터인 ready_set을 인자로 전달한다. 나머지 인자로 NULL을 주게되면, Select 함수는 파일 디스크립터가 있을 때 까지 대기한다. 만약, 파일 디스크립터가 있다면 그 수만큼 반환하고 nready에 할당한다.

FD_ISSET 함수를 통해, listenfd 파일 디스크립터가 ready_set에 있는지 확인하고, 있다면 새로운 연결의 수신을 의미하므로 Accept 함수를 호출한다. 클라이언트와 서버의 연결을 수락했으므로 이를 위한 새로운 파일 디스크립터 connfd에 반환한다. 다음으로, 해당 클라이언트를 pool에 추가하기 위해 add_client 함수를 호출한다.

마지막으로 check_clients 함수를 호출하고 pool에 존재하는 모든 클라이언트에 대해 요청을 처리한다. 이 함수에서 사용자가 입력하는 show/buy/sell 명령어를 수행한다.

- Task 2

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

```c
typedef struct {
    int *buf; /* Buffer array */
    int n; /* Maximum number of slots */
    int front; /* buf[(front+1)%n] is first item */
    int rear; /* buf[rear%n] is last item */
    sem_t mutex; /* Protects accesses to buf */
    sem_t slots; /* Counts available slots */
    sem_t items; /* Counts available items */
} sbuf_t;
```

위는 Task2에서 사용되는 item 구조체와 sbuf_t 구조체이다. thread의 readers-writesr problem을 위해 semaphore를 사용했다. 해당 구조체와 관련된 sbuf package 함수들을 사용했다.

listen함수로 클라이언트 요청을 대기하고, subf_init 함수로 sbuf를 초기화한다. 이후, NTHREADS 만큼 Worker thread를 Pthrread_create 함수를 이용해 생성한다. 이제 while로 무한루프를 돌면서 클라이언트의 연결을 수락하고 처리한다. Task1과 마찬가지로, Accept함수를 사용하여 수락하면 connfd를 sbuf_insert함수로 sbuf 버퍼에 삽입한다. 따라서, Worker thread 가 sbuf 버퍼로부터 클라이언트 요청을 처리한다.

Worker thread는 void *thread 함수를 실행한다. 현재 실행 중인 thread를 detached 상태로 변경하여, 스레드가 종료되었을 때 자동으로 자원을 해제하도록 한다. 다시 while 무한루프를 돌며 클라이언트의 요청을 받고 수행한다. sbuf_remove 함수를 사용하면, sbuf에서 클라이언트와의 연결을 위한 connfd를 가져오고 제거한다. 이후, task1과 동일한 역할을 수행하는 check_clients 을 호출하여 요청을 처리한다. 이후 Close(connfd)를 함으로서 메모리 누수 없이 클라이언트와의 연결을 종료한다. Worker thread는 끝나면 다시 thread pool로 반환된다. 이때 각 thread는 요청을 독립적으로 수행하므로, concurrent하게 요청을 처리할 수 있게 된다.

Readers-writers problem을 해결하기 위해 바이너리 트리의 노드단위로 관리하였다. 우선 데이터를 삽입할 때 Sem_init 함수로 노드의 mutex와 w semaphore를 초기화한다. 다음으로, 사용자가 show를 입력하게 되면 print_tree함수를 호출한다. 이때 각 노드는 P(&(node->data.mutex)) 함수로 현재 노드에 대한 뮤텍스 잠금을 우선 획득하고 readcnt를 증가시킨다. 다음으로 read하는 동안 write를 막기 위해 w semaphore도 잠근다. 이제 V함수로 잠금을 해제하고, sprintf 작업을 수행한다. 이제 다시 P를 통해 노드의 mutex 잠금을 획득하고 readcnt 감소, 읽기 작업이 마지막이라면, w 잠금을 해제한다. 그리고 다시 최종적으로 mutex잠금을 해

제한다. 비슷한 방식으로 buy/sell/search_tree 에서도 각 노드에 동시 접근을 제어하도록 코드를 추가하였다.

결론적으로, 어떤 노드에 대해서 읽기 작업은 동시에 여러 개 수행되어도 상관없지만, 쓰기 작업은 반드시 한번에 하나만 수행되어여 하며 해당 작업중에는 어떠한 작업도 수행되지 않아야한다. 이를 통해 readers-writers problem을 해결할 수 있다.

- Task 3

task1 과 task2의 성능 평가를 위해서 주어진 시간 측정 함수를 muticlient에 추가하여 사용했다. 또한, 실험을 위해 muticlient 파일을 수정하여 여러가지 상황에 따른 실험을 진행했다. 또한, NTHREADS를 변화시켜가며 실험을 진행한다.

3. 구현 결과

- 결과

Event-driven 방식은 예상했던 대로 단일 스레드에서 진행되므로 readers-writers problem이 전혀 발생하지 않았다. 또한, concurrent하게 mutliclient의 요청을 잘 수행함을 볼 수 있었다.

마찬가지로 thread-based는 semaphore를 통해 노드 단위의 관리 (fine-grained locking)을 잘 활용하여, 클라이언트간의 읽기와 쓰기의 충돌이나, race, deadlock, starvation과 같은 우려할 만한 사항들은 전혀 발생하지 않았다. 다만, node로 lock을 관리하기 때문에 critical section에 진입하는 순서에 따라 애매한 상황이 연출될 수 있다. 예로, A가 서버에 2번 ID에 buy를 요청했고, B가 서버에 show를 요청했다고 가정하자. B가 1번 주식에 접근하는동안, A의 요청이 처리되었다면 B에게 보여지는 show는 A의 요청사항이 반영된 이후이지만, B가 마지막 주식에 접근하는 동안 A가 뒤늦게 요청을 처리했다면 반영되지 못한다. 이를 해결하기 위해서는 show를 진행하는 동안에는 buy/sell 요청을 block하는 것이지만, 이는 동시성을 크게 제한한다.

- 고려할만한 점

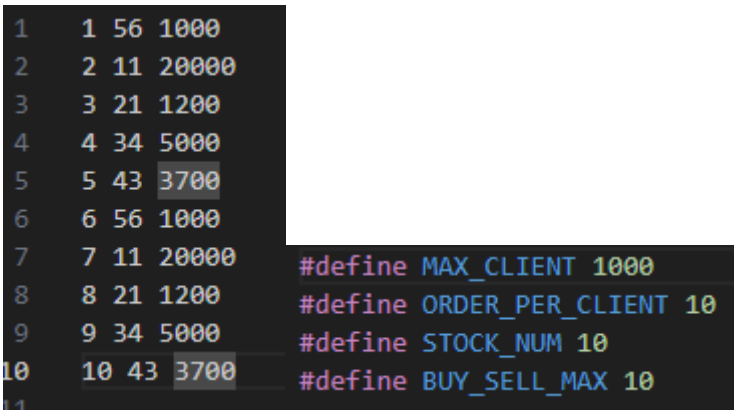
현재 주식 정보는 이진트리로 저장되고 관리된다. 이때, stock.txt에서 한줄씩 정보를 읽고 삽입한다. 그런데, 만약 파일에 주식 ID가 1, 2, 3, 4, 5 … 순서로 정렬된 상태라면 편향된 이진트리가 생성된다. 편향된 이진트리의 탐색과 삽입의 시

간복잡도는 O(N)으로, 기대 시간복잡도 O(log N) 보다 비효율적이게 된다. 따라서 일반 이진트리보다, 레드-블랙 트리와 같은 자가 균형 이진 검색트리를 사용하는게 더욱 합리적일 것 이다. 이러한 트리는 데이터의 순서에 상관없이 트리의 높이가 log N을 유지한다.

4. 성능 평가 결과 (Task 3)

cspro9에서 서버를 실행하고, cspro에서 클라이언트를 실행하여 실습했다.

우선, SBUFSIZE = 1024, stock.txt가 다음 형식으로 저장되어있다고 가정하고(ID의 순서는 동일하지만, 가격 및 수량은 변할 수 있음) 진행하였다. 원할한 실험을 위해 multiclient의 sleep은 주석처리 후 진행하였다. 모든 시간 측정은 5번 실행 후, 평균을 내었다. ORDER_PER_CLIENT는 항상 10으로 진행한다. 처리량은 클라이언트 수 * ORDER_PER_CLIENT로 여기서는 10으로 고정이다.



```
1 56 1000
2 11 20000
3 21 1200
4 34 5000
5 43 3700
6 56 1000
7 11 20000
8 21 1200
9 34 5000
10 43 3700
```

```
#define MAX_CLIENT 1000
#define ORDER_PER_CLIENT 10
#define STOCK_NUM 10
#define BUY_SELL_MAX 10
```

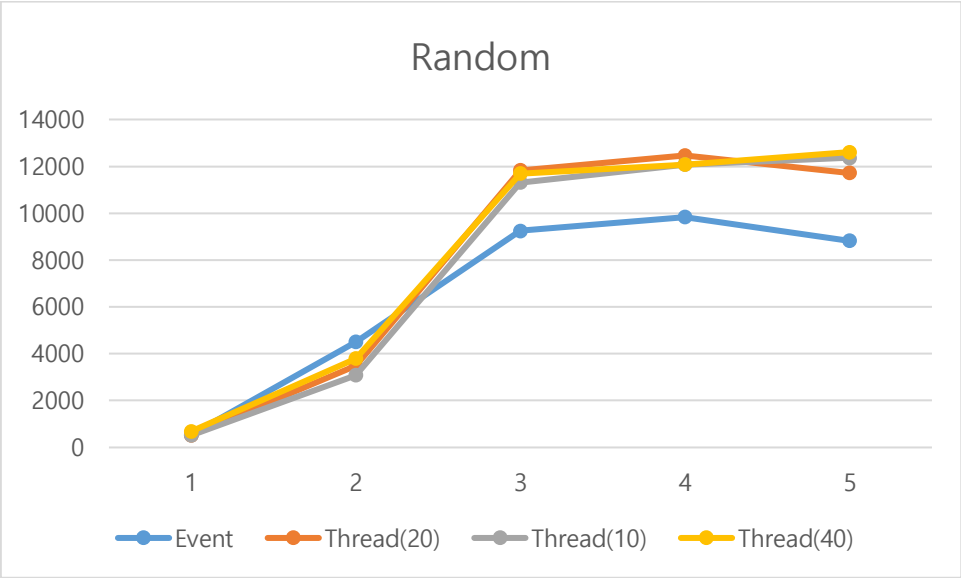
우선 워크로드는 show/sell/buy 모두 가능하게 하고 실험을 진행하였다.

처리량	10	100	1000	2000	3000
Event	18220	22174	108205	203475	339961
Thread(20)	18994	28697	84544	160455	255907
Thread(10)	19323	32431	88426	165541	242530
Thread(40)	14571	26312	85493	165641	237912

위 실험은 NTHREAD의 수와 EVENT의 각 client 가 1, 10, 100, 200, 300 명일 때 걸린 시간을 나타냈다. 이때 단위는 micro seconds 이다. 처리량이 적을 때는 각 5번 실행결과의 시간 편차가 심하게 나타났다. 또한, 처리량이 적은 경우에는 Event와 Thread간의 유의미한 차이가 나지 않았다. 하지만, client의 수가 100이 넘어갈수록 유의미한 차이를 보였다. Thread의 개수에 대한 차이는 거의 나지 않

았으며 해당 차이는 아무래도 명령어에 show의 개수에 대한 차이로 난다고 봐도 무방하다. 아래 표는 초당 처리량을 계산한 값이며, 그래프로 나타냈다.

Event	548.8474204	4509.786	9241.7171	9829.2173	8824.5416
Thread(20)	526.482047	3484.685	11828.16	12464.554	11723.009
Thread(10)	517.5179837	3083.47	11308.891	12081.599	12369.604
Thread(40)	686.2946949	3800.547	11696.864	12074.305	12609.704

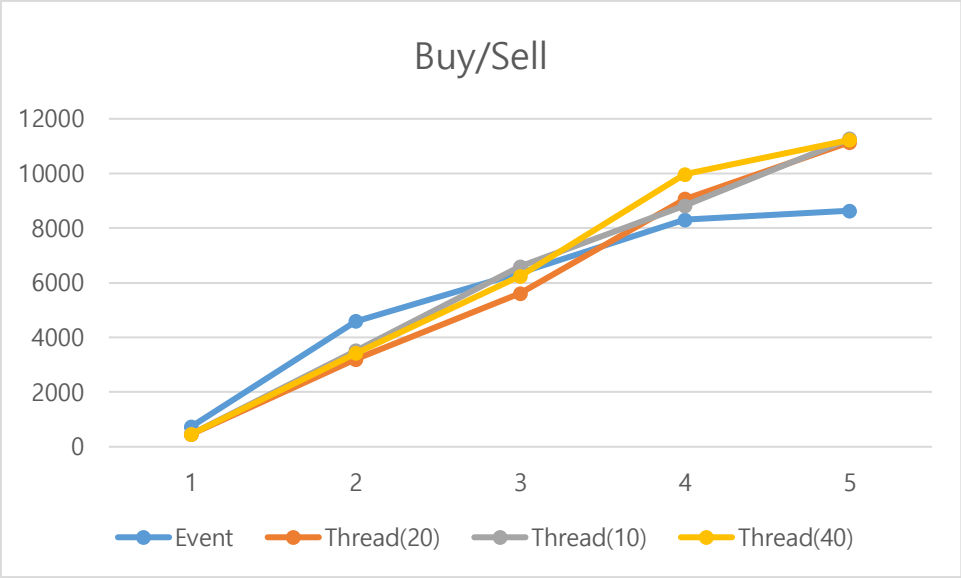


다음으로 워크로드를 바꿔보았다. 우선 sell과 buy의 수행은 비슷하므로 둘을 묶어 진행하고, show만 따로 진행한다. 아래 표는 buy/sell 만을 워크로드로 하여 진행했다.

처리량	10	100	200	400	600
Event	13579	21784	31361	48184	69458
Thread(20)	21890	31292	35609	44120	53840
Thread(10)	21503	28447	30315	45320	53257
Thread(40)	21774	29201	32013	40124	53445

마찬가지로 초당 처리량으로 나타낸 표와 그래프는 아래와 같다.

Event	736.4313	4590.525	6377.348	8301.511	8638.314
Thread(20)	456.8296	3195.705	5616.558	9066.183	11144.13
Thread(10)	465.0514	3515.309	6597.394	8826.125	11266.12
Thread(40)	459.2633	3424.54	6247.462	9969.096	11226.49

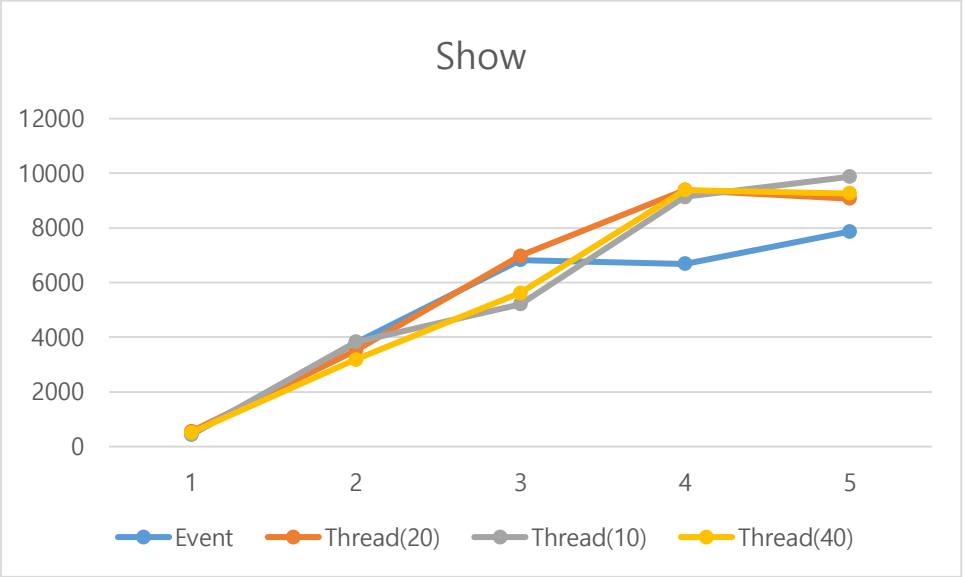


확인해보면, client 수가 적을수록 Event의 성능이 좋은 것을 볼 수 있다. 반면, client의 수가 20이 넘어 처리량이 200이 되는순간 thread의 성능이 좋아진다. 이는 아무래도, client의 수가 20 보다 적으면 thread의 장점인 병렬성을 살리지 못해서라고 생각한다. 현재 서버의 cpu 수는 20인데, 20보다 client가 작다면 모든 cpu를 활용하지 못한다. 따라서, 20을 기점으로 thread가 event보다 성능이 좋아진다고 생각한다. 그러나, 앞선 결과와 유사하게 thread의 수가 10, 20, 40일 때는 확연한 차이를 보이지 못하고 거의 비슷한 값을 보이는 것으로 보인다. 이는 context switching 비용과, 동기화 비용이 서로 trade of 관계에 있기 때문이라고 예상한다. thread 수가 많아지면 context 스위칭 비용이 증가하여 실행 성능은 크게 좋아지지 않을 수 있으며, 반대로 thread 수가 작으면 context 스위칭 비용은 작아지지만, 모든 코어를 사용하지 않아 성능이 저하될 수 있다. 동기화 비용도 마찬가지로 스레드 수가 증가할수록 동기화에 필요한 작업이 더 많아져 동기화 비용이 증가한다.

이제 동일한 조건으로 show만을 수행하도록 하고 측정을 진행한다.

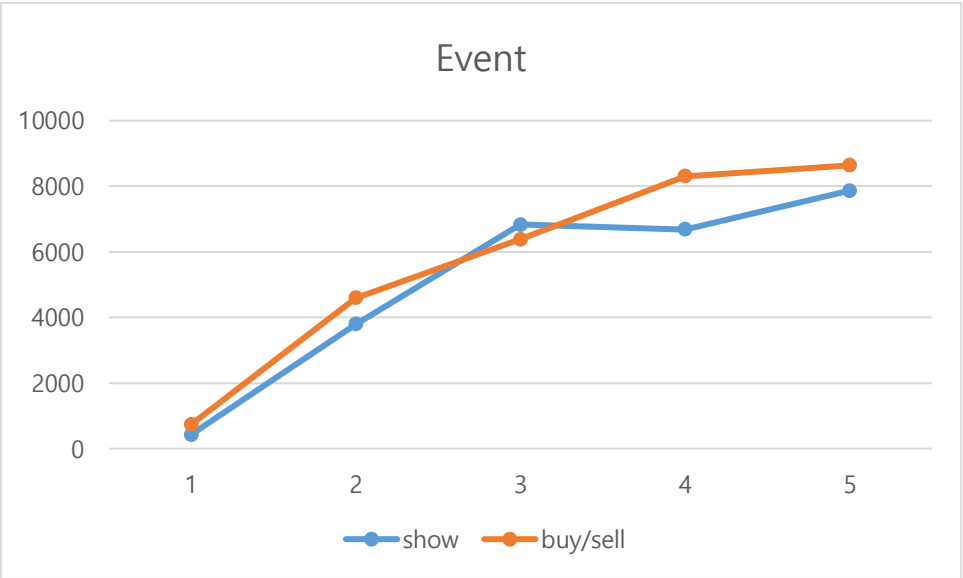
처리량	10	100	200	400	600
Event	23149	26379	29275	59833	76265
Thread(20)	18156	28331	28631	42628	66129
Thread(10)	22174	26132	38411	43763	60784
Thread(40)	19482	31366	35546	42633	64805

Event	431.9841	3790.894	6831.768	6685.274	7867.305
Thread(20)	550.7821	3529.702	6985.435	9383.504	9073.175
Thread(10)	450.9786	3826.726	5206.842	9140.141	9871.019
Thread(40)	513.2943	3188.166	5626.512	9382.403	9258.545

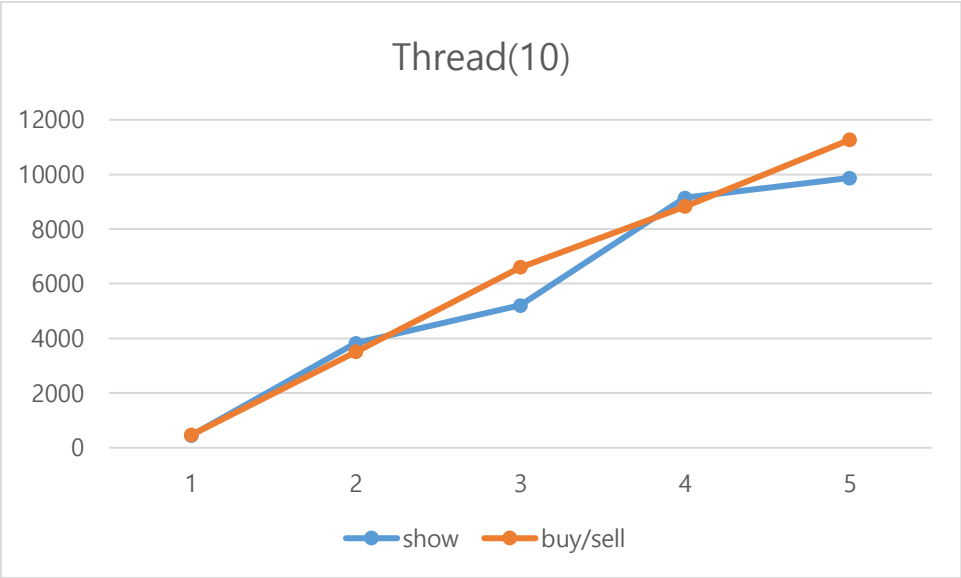


Show가 더 오래걸릴 것이라는 예상과는 달리, 동시처리율은 show가 더 빠르거나 비슷하게 나타났다.

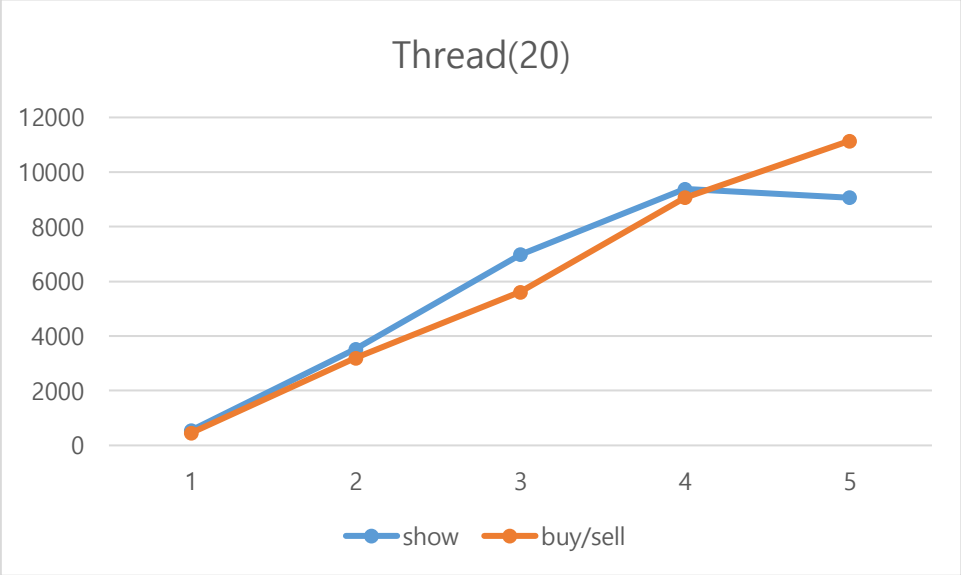
show	431.9841	3790.894	6831.768	6685.274	7867.305
buy/sell	736.4313	4590.525	6377.348	8301.511	8638.314



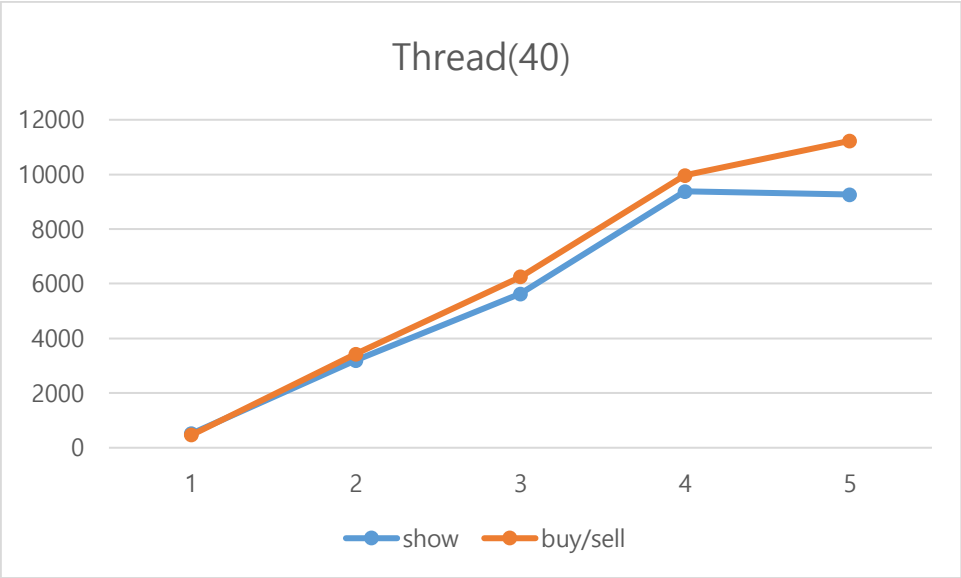
show	450.9786	3826.726	5206.842	9140.141	9871.019
buy/sell	465.0514	3515.309	6597.394	8826.125	11266.12



show	456.8296	3195.705	5616.558	9066.183	11144.13
buy/sell	550.7821	3529.702	6985.435	9383.504	9073.175



show	513.2943	3188.166	5626.512	9382.403	9258.545
buy/sell	459.2633	3424.54	6247.462	9969.096	11226.49



예상과 다르게 나타난 이유로, stock.txt의 ID가 정렬되어있음을 들 수 있을 것 같다. 현재, stock.txt의 파일은 1~10으로 정렬 되어있는데, 이는 편향된 이진트리를 만들게 된다. 따라서, 이진트리의 장점이 사라지고, 탐색과 순회 모두 시간복잡도 O(N)을 가지게 된다.

실험 결과를 분석해보면, 결론적으로는 thread가 처리량이 늘어날수록 압도적인 퍼포먼스를 보였다. 또한, thread의 수 10,20,40은 성능차이를 내기에는 차이가 너무 작았다고 판단된다. 해당 thread들의 성능이 비슷한 이유로는 앞서 말했듯이 실행환경의 cpu수가 20개이기도 하며 context switching 비용과, 동기화 비용이 서로 trade of 관계에 있기 때문이라고 예측된다. 또한, stock.txt에 저장되어있는 데이터의 형식에도 실험결과에 영향을 주었을 것이다. 그럼에도 불구하고 이번 실험은 2-B의 task3에서 예측한 것과 어느정도 유사한 결과를 얻을 수 있었다는 점이 다행이라고 생각된다.

아쉬운점은, 네트워크의 불안정한 연결이 실험결과에 영향을 많이 미쳤을 것 같다라는 점이다. 이번 실험은 서강대학교 로욜라 도서관에서 진행했는데, 무선 와이파이는 종종 끊기며 일정하지 않고 불안정하다. 실제로도 각 케이스에 대한 측

정값의 편차가 너무 심했었다. 또한, 마지막에 가서야 여러가지 실험방법들이 떠오른 것이다. 실제로 시간에 영향을 주는 요인은 task3에서 진행한 요인외에도 분석할 점이 굉장히 많았다. client의 요청 수를 증가시켜 실험하거나 stock의 개수를 늘리면서 실험하거나, 편향된 이진트리가 아닌 균형이진트리로 실험하는 등의 방법들이 존재할 것이다.