System Programming Project 3

담당 교수 : 김영재 교수님

이름 : 송지훈

학번 : 20201595

1. 개발 목표

강의 시간에 배운 Event-driven 방식을 사용한 서버와 Thread-based 방식을 사용한 서버를 만들고 그에 따른 동작을 확인한다. 각각의 주식 서버는 show, buy, sell 등의 명령을 통해 주식 차트를 확인하고 주식을 구매, 판매할 수 있다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   I/O multiplexing을 사용하여 Event-driven program을 설계한다. Select 함수를 이용하여 구현한다. Multi-client가 서버에 연결되고 그에 따라 bit vector에 연결된 client를 표시한다. 이후 서버 프로세스에서 select 함수를 통해 client가 보내는 요청을 확인하고 그에 따른 응답을 순차적으로 진행한다. 이후 모든 client들이 종료되면 stock.txt 파일을 업데이트하고 서버를 종료한다.

2. Task 2: Thread-based Approach

   Thread를 사용한 Thread-based program을 설계한다. 기능은 위와 같지만 하나의 프로세스를 이용하는 것이 아닌 복수의 process를 사용하여 기능을 수행한다. 강의 시간에 배운 Producer-Consumer 방식과 Reader-Writer 방식을 사용한다. 우선 미리 정해 놓은 개수의 Thread를 생성해 Worker Thread pool을 만든다. 그 후 새로운 connected descriptor가 생성되면 공유하고 있는 buffer인 sbuf에 connfd를 추가하고 Worker가 작동해 해당하는 connfd에 대한 작업을 처리한다.

3. Task 3: Performance Evaluation

   Task 1과 Task 2에서 구현한 Concurrent server를 평가한다. 평가하는 기준은 시간당 client 처리 요청 개수인 동시 처리율이다. 동시 처리율의 관점에서 확장성, 워크로드에 따른 분석을 진행한다. 확장성이란 각 Task에 대한 client 수의 변화에 따른 동시처리율 변화이고 워크로드는 client의 요청 즉, show, buy, sell에 따른 동시 처리율에 대한 변화이다. 또한 Thread-based server에

대해서는 추가적으로 미리 생성된 worker thread의 개수에 따른 동시 처리율 변화를 살펴보겠다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Select() 함수를 사용한 Event-driven Approach server는 한 개의 server 프로세스로 concurrent한 작동을 하는 server이다. 우선 listenfd를 통해 connfd를 관리하는 array가 존재한다. Client가 server측에 연결을 시도하면 listenfd가 이를 받고 connfd를 리턴하는데 이를 array에 저장하고 select 함수에 전달할 pending bit vector에 connfd에 해당하는 값을 set 해준다. Select 함수는 관찰하고자 하는 file descriptor들에 대한 정보를 받고 해당하는 FD에 입력이 발생하면 이를 알리는 방식이다. 이런 방식으로 여러 client들에 대한 정보를 저장하고 select 함수를 실행하면 해당하는 bit vector에 대한 요청이 발생했을 때 해당 비트를 1로 변경하고 그 bit vector를 return 한다. Server 측에서는 그 bit vector에 대한 값을 확인하며 순차적으로 1인 비트에 대항하는 connfd에 대해 요청을 수행한다. 위와 같은 작업을 모든 client들이 종료할 때까지 반복한다.

✓ epoll과의 차이점 서술

우선 위에서 설명했던 것과 마찬가지로 select는 관찰하고자 하는 FD에 대한 bit vector를 kernel에 넘기고 I/O event가 발생한 FD에 대항하는 bit vector를 리턴하는 방식이다. 이 과정에서 매번 bit vector에 대한 복사와 전달이 발생한다. Epoll은 이 과정을 최대한 간소화 시킨 버전이다. Select와 마찬가지로 event가 발생한 FD에 대한 값을 알려주는 것은 동일하지만 epoll은 관찰하고자 하는 FD를 처음에 한 번 복사해 kernel에 넘겨 저장하고 그 저장 장소에 대한 FD를 받는다. 이후 해당하는 FD에 event가 발생하면 kernel에서 user에게 알려준다. 즉 처음부터 모든 bit vector에 대한 반복문을 수행하며 탐색할 필요가 없이 event가 발생한 FD만을 알 수 있다. 이러한 특성 때문에 epoll은 select보다 더 빠르고 많은 multiplexing을 지원하지만 select와 비교해 다

른 OS와 호환성이 떨어진다는 단점이 있다.

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

  ✓ Master Thread의 Connection 관리

  Thread-based server에서는 n개의 worker에 해당하는 thread를 미리 만들어 놓고 이를 sleep 상태로 유지한다. 만약 event가 발생한다면 thread가 깨어나 작업을 수행한다. event에 대한 정보를 저장하기 위해서 공유 버퍼를 선언하고 사용한다. Client가 연결해 connfd가 생성되면 master thread는 이를 sbuf의 buf에 저장한다. 기존에 아무것도 존재하지 않던 buf에 새로운 입력이 발생하면 thread가 sleep 상태에서 깨어나 item slot에서 connfd를 삭제하고 해당하는 fd에 대한 작업을 수행한다. 이후 connfd에 대한 연결이 종료되면 해당 thread는 다시 sleep 상태로 돌아가 새로운 event가 발생할 때까지 대기한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  위에서 설명했던 것처럼 master thread는 client가 연결되면 해당하는 connfd 값을 받아 공유 버퍼의 buf에 저장한다. 미리 생성된 후 sleep하고 있던 thread들은 buf의 맨 앞에 있는 connfd 값을 가져와 해당하는 입력에 대한 작업을 수행한다. thread들은 buf에 값이 새로 들어오는 것을 인지하면 해당 값을 제거하고 가져와 connfd에 해당하는 client들에 대해 작업을 수행한다. 이후 client의 연결이 종료되면 Close 함수를 통해 connfd를 닫고 다시 sleep 상태로 들어가 새로운 buf가 들어오기를 기다리고 있는다.

- **Task3 (Performance Evaluation)**

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

  우선 가장 먼저 생각해야 되는 부분은 '언제부터 시간을 측정할 것인가'와 '서버와 client 중 시간을 측정하는 곳을 어디로 해야 하는가' 이다. 언제부터 시간을 측정할 것인지에 대해서는 client 측에서 server 측으로 연결이 되기 직전에 측정을 시작해 모든 client들이 최종 요청에 대한 응답을 받았을 때 종료하도록 설정하였다. 그 이유는 서비스를 제공하는 서버 측에서 마지막으로 송신한 부분이 client에게 전달되었을 때가 서비스를 마치는 마지막이라고 판단했다. 또한 모든 평가의 실험에 있어서 수행하는 명령의 개수는 제공된 multiclient.c의 초기 개수인 10개로 진행하였다. 모든 측정 결과는 네트워크

상태에 결과가 변동될 수 있으므로 이를 최소화 하기 위해 5번의 수행을 진행한 평균값으로 측정한다. 또한 제공된 multiclient.c 파일에는 usleep(1000000)이 호출되어 매 시행마다 1초를 기다리지만 딜레이가 없는 환경에서 정확한 측정을 수행하기 위해 이를 제거하고 측정했다.

1) 확장성 분석

-metric: 동시에 접속한 client의 개수에 따른 동시 처리율

확장성 분석을 진행하기 위해 client의 개수를 1, 10, 50, 100, 200개로 늘려가며 진행하도록 하였다. 이렇게 늘려가며 진행한 이유는 client가 늘어날수록 동시처리율의 변화폭이 더 크게 확인될 것이라 생각해 이렇게 설정하였다. Thread-based server에서 수행할 때 thread의 개수는 max client의 수보다 크게 설정한 것으로 가정한다.

2) 워크로드에 따른 분석

-metric: client 요청 타입에 따른 동시 처리율

워크로드에 따른 분석을 진행하기 위해 케이스를 총 3개로 구분하였다. 첫 번째는 buy 혹은 sell만 진행하는 경우, 두 번째는 show만 진행하는 경우 세 번째는 랜덤으로 진행되는 경우이다. Buy와 sell, show를 나눈 이유는 write를 진행하는 과정에서 mutex lock이 발생하기 때문이다. 이 때 client의 개수에 영향을 받지 않도록 client의 개수는 100개로 고정했다. Thread-based server에서 수행할 때 thread의 개수는 100개보다 크게 설정한다.

3) Worker thread의 개수에 따른 분석

-metric: Task2에서 Worker thread의 개수에 따른 분석

위의 분석을 진행하기 위해서 우선 multiclient를 100개로 고정하고 thread의 개수를 1, 10, 50, 100, 500개로 늘려가며 진행하도록 하였다. 위와 같이 진행한 이유는 고정된 client에 대해 thread의 개수가 늘어남에 따라 변화하는 수치를 더 자세히 확인하기 위해서 위와 같이 설정했다.

✓ Configuration 변화에 따른 예상 결과 서술

우선 확장성에 대해서는 event-driven 보다 thread-based 의 동시처리율 증가폭이 더 클 것이라고 예상한다. 그 이유는 event-driven 은 모든 array 를 탐색하며 수행하고 thread- based 는 각각의 thread 가 독립적으로 시행되기 때문이다.

워크로드에 따른 분석에서는 명령어 show 만 사용했을 때 더 높은 동시처리율을 보일 것 이라 생각한다. Thread-based 에서는 공유 변수에 대한 접근이 이뤄질 때 P 와 V 를 통해 통제하므로 통제가 필요하지 않고 모두 접근할 수 있는 show 로만 이뤄졌을 때 더 높은 효율을 보일 것이라 생각한다.

마지막으로 thread 의 개수에 대해서는 늘어날수록 더 동시 처리율이 증가하지만 100 개와 500 개의 차이는 미비할 것이라고 예상한다. 왜냐면 이미 100 개의 client 에 대한 처리를 진행하고 있으므로 thread 가 더 늘어난다고 변화할 것 같지 않기 때문이다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1) Event-driven server

Event-driven server를 구현하기 위해서 우선 추가적으로 주식을 저장하기 위한 구조체 ITEM과 select와 multiplexing을 구현하기 위한 구조체 stockpool을 추가로 구현했다. 각각의 구조체는 아래와 같다.

```
typedef struct item{
    int ID;
    int left_stock;
    int price;
    int readcnt;
    struct item *left_node;
    struct item *right_node;
}ITEM;
```

```
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];
}stockpool;
```

ITEM은 주식의 id, 남은 주식의 수량, 주식의 가격을 정보로 가지고 있고 Binary search tree를 구현하기 위한 left와 right node 포인터가 존재한다.

Stockpool은 가장 큰 fd, 확인할 fd를 저장할 bit vector read_set, 그 bit vector를 임시로 저장할 bit vector ready_set, event가 발생한 개수인 nready, iteration을 돌 최대 숫자인 maxi, client의 connfd를 저장할 clientfd, rio를 저장할 clientrio가 존재한다.

```
char tempbuf[MAXLINE];
ITEM *list = NULL;
```

또한 전역변수로 임시로 출력 값을 저장할 tempbuf, 만들어진 주식 리스트를 저장할 ITEM 포인터 list를 선언한다. Stockpool을 관리하기 위한 함수들은 다음과 같다.

```
void init_pool(int listenfd, stockpool *p);
void add_client(int connfd, stockpool *p);
void check_clients(stockpool *p, ITEM* list);
```

Init_pool 함수는 stockpool을 초기화하는 함수이다. Listenfd와 stockpool p를 인자로 받아 p의 인자를 초기화하고 maxfd를 listenfd로 설정한다.

Add_client 함수는 connfd와 p를 인자로 받아 클라이언트 목록에 connfd를 추가하는 함수이다. 관찰할 클라이언트인 clientfd에 connfd를 집어넣고 다른 값들을 업데이트 한다.

Check_client 함수는 p 와 list를 인자로 받아 ready_set bit vector가 1인 connfd에 대해 요청을 확인하고 그에 따른 기능을 수행한다. 수행할 수 있는 요청에는 show, buy, sell이 존재한다.

추가적으로 구현한 함수는 다음과 같다.

```
ITEM *makebinary (ITEM *root, int id, int m, int price);
void showlist(ITEM *list);
ITEM *search (ITEM* root, int id);
```

우선 makebinary 함수는 ITEM root, 주식 id, 주식 수량, 가격을 받아와 binary tree의 구조로 자료를 저장하는 함수이다. 리턴값으로 root의 주소를 가진다.

Showlist 함수는 구성된 item 리스트를 출력하는 함수이다. 인자로 ITEM *list를 가진다. 리턴값은 없다.

Search 함수는 item 리스트에서 id에 해당하는 주식을 찾는 함수이다. 리턴값으로 해당하는 노드의 주소를 리턴한다.

```
void updatetxt(ITEM *list, FILE* update);
```

모든 client가 접속을 종료하면 바뀐 주식 데이터를 업데이트하는 함수이다.

Event-driven server에서는 main에서 stock.txt를 열어 binary tree를 만들고 client와 연결할 준비를 한다. 이후 무한 루프를 돌면서 client가 보내는 요청에 해당하는 작업을 수행한다. 만약 새로운 client가 도착한다면 add_client 함수를 통해 추가하고 check_client함수를 통해 명령을 수행한다. 만약 Ctrl+c를 통해 Sigint가 발생하면 updatetxt함수를 호출해 stock.txt 파일을 수정한다.

2) Thread-based server

Thread-based server를 구현하기 위해 위에서 사용했던 ITEM 구조체와 공유 버퍼에 대한 구조체 sbuf_t를 선언했다. Sbuf는 다음과 같다.

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

Buffer를 저장하기 위한 buf, buf의 크기를 저장하기 위한 n, buf의 앞과 뒤를 가리키기 위한 front와 rear, sem_t type으로 선언돼 P와 V 에 사용될 mutex, slots, items가 존재한다. ITEM 구조체는 sem_t type 변수 mutex와 w가 추가로 선언되었다. 이 둘이 추가된 이유는 후술하겠다. Sbuf를 관리하기 위한 추가적인 함수들은 다음과 같다.

```
void sbuf_init(sbuf_t *sp, int n);
void sbuf_deinit(sbuf_t *sp);
void sbuf_insert(sbuf_t *sp, int item);
int sbuf_remove(sbuf_t *sp);
```

Sbuf_init은 sbuf_t* 와 int를 인자로 받아 sbuf를 초기화하는 함수이다. 리턴값은 없다

Sbuf_deinit은 sbuf_t*를 인자로 받아 buf를 free 시키는 함수이다. 리턴값은 없다.

Sbuf_insert는 sbuf_t*와 int를 인자로 받아 item을 buf에 rear가 가리키는 곳에 넣고 rear를 업데이트하는 함수이다. 리턴값은 없다

Sbuf_remove는 sbuf_t*를 인자로 받아 front에 있는 connfd값을 리턴하고 front를 업데이트하는 함수이다. 새롭게 선언된 전역변수는 다음과 같다.

```
ITEM *list=NULL;
sem_t mutex;
sbuf_t sbuf;
```

List는 주식의 정보를 저장하기 위한 포인터이다. Mutex는 P와 V 연산에 사용되는 인자들이다. Sbuf는 공유 버퍼로 사용된다.

```
void *thread(void *vargp);
void init_set();
void checkcommand(int connfd);
```

Thread 함수는 pthread_Create함수로 thread가 만들어 졌을 때 수행되는 함수이다. Buf에 값이 존재한다면 그것을 가지고와 checkcommand함수로 connfd를 보내 받은 요청을 수행한다. Connfd가 종료되면 totalclient를 1 감소하고 만약 totalclient가 0이라면 updatetxt함수를 통해 stock.txt를 업데이트한다.

Init_set 함수는 사용하는 전역변수들을 초기값으로 초기화하는 함수이다.

Checkcommand 함수는 connfd를 인자로 받아 해당하는 fd에서 보낸 요청을 확인하고 이에 따른 기능을 수행한다.

Thread-based는 event-driven과는 다르게 다른 thread에서 공유 변수에 동시에 접속하면 적절하게 작업이 수행되지 않는다. 이를 방지하기 위해 P와 V를 사용해 공유변수에 접근할 때 다른 thread가 접근하고 있다면 접근하지 못하도록 한다. 다음은 reader-writer problem을 적용한 예이다.

```
if(root!=NULL){
    P(&(root->mutex));
    root->readcnt++;
    if(root->readcnt==1)
        P(&(root->w));
    V(&(root->mutex));

    sprintf(temp, "%d %d %d\n", root->ID, root->left_stock, root->price);
    strcat(tempbuf,temp);

    P(&(root->mutex));
    root->readcnt--;
    if(root->readcnt==0)
        V(&(root->w));
    V(&(root->mutex));
}
```

위의 코드는 showlist 함수에서 수행되는 코드의 일부이다.

공유 변수를 변경할 때는 P(&mutex)를 통해 다른 thread가 접근하지 못하게 하고 변경을 완료하면 V(&mutex)를 통해 lock을 해제한다. 위에서는 show를 통해 읽고 있을 때 값이 변화하면 안되므로 읽으려고 시도하는 thread가 1개라면 P(&w)를 통해 write에 해당하는 작업을 멈추고 readcnt가 0일 때 write에 대한 작업을 수행하도록 한다. 이 때 lock이 발생하는 것을 최소화 하기 위해 전체 list에 대한 lock을 발생시키는 것이 아니라 해당하는 항목에 대한 readcnt를 증가시켜 최소한의 lock이 발생하게 만들었다.

Event-driven과 Thread-based에 서버 종료를 SIGINT로 통일했으므로 두 함수에 추가되는 sigint_handler를 선언한다. Sigint_handler 함수의 내부는 다음과 같다.

```
void sigint_handler(int sig){
    int olderr =errno;
    FILE *update = fopen("stock.txt", "w");
    updatetxt(list, update);
    fclose(update);

    errno=olderr;
    exit(0);
}
```

코드를 보면 알 수 있듯이 sigint가 발생하면 file을 열어 업데이트하고 프로그램을 종료한다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

작성한 코드를 통해 select를 사용한 event-driven server와 thread를 이용한 thread-based server는 모두 주어진 명령에 concurrency issue 없이 구현되었다. Thread-based server의 경우 multiclient.c를 실행해 여러 client가 동시에 접속하게 만들었을 때 정해 놓은 thread의 숫자가 multiclient의 숫자보다 적을 시 다소 시간이 걸리는 것을 확인할 수 있었다. 또한 reader-writer problem을 사용해 구현한 결과 writer에는 starvation이 발생한다. 실제로 연산이 많아졌을 경우 writer 작업을 수행하는 함수에서는 대기시간이 길어질 수 있다. 이를 해결하기 위해서는 reader에 제한을 두어 일정 횟수 이상 read를 완료했을 시 writer가 작동할 수 있게 대기하도록 하는 코드를 추가할 수 있다. 또한 실제 주식 시장에서는 현재 stock의 개수와 가격이 실시간으로 변하지만 실습에서 구현한 주식 서버는 그렇지 않다. 이 부분과 전체 주식에 대한 가격 변화도 추가적으로 구현해볼 수 있다.

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

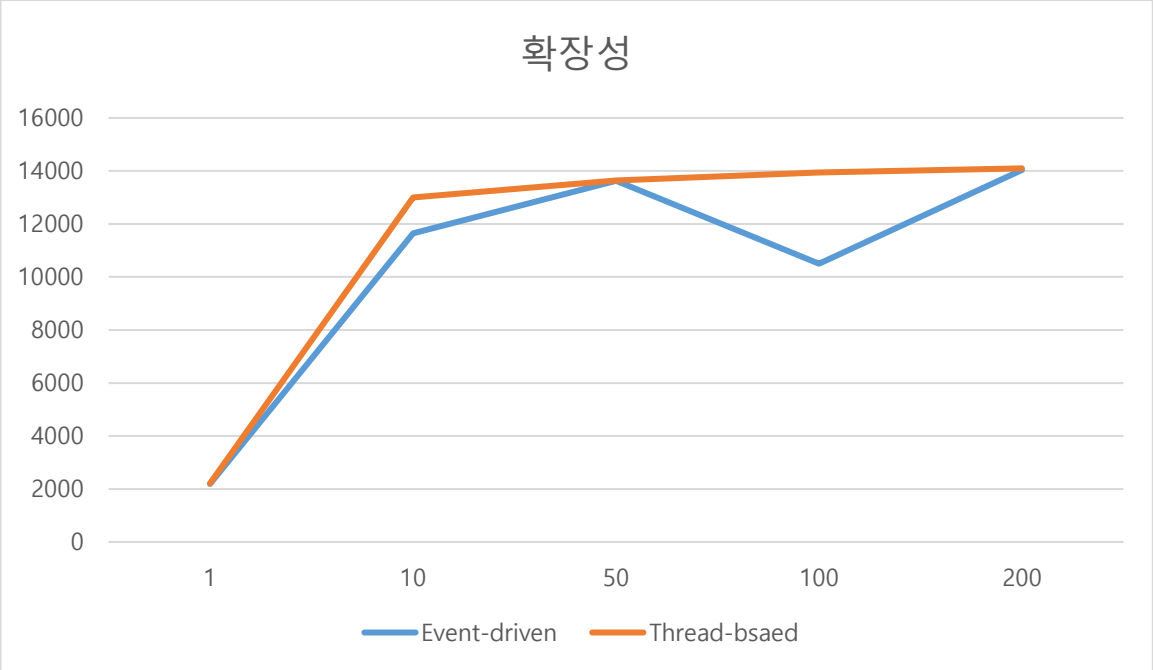
성능 평가에서 시간 측정은 모두 multiclient 측에서 진행되었다.

(1) 확장성 분석

확장성 분석은 event-driven server와 Thread-based server의 동시 접속자 수에 따른 동시처리율 분석이다. Client의 수는 1, 10, 50, 100, 200명으로 증가시키면서 진행했다. 또한 client의 요청은 buy, sell, show 중 랜덤으로 선택되었다. 모든 측정을 진행할 때에는 multiclient.c에 있는 sleep을 포함하지 않고 진행했고 client당 order의 개수는 10개로 고정했다. 각각의 경우에 대해서는 5번의 수행을 한 후 그 평균으로 계산하였다. 다음은 결과를 정리한 표이다.

client 수	Event-driven		Thread-based	
	average	동시처리율	average	동시처리율
1	0.004567	2189.621196	0.004523	2210.726445
10	0.008587	11645.51066	0.007392	13000.182
50	0.036672	13634.23174	0.036639	13646.51168
100	0.095191	10505.17275	0.071743	13938.5648
200	0.14217	14037.62589	0.141862	14098.24787

위의 표를 그래프로 나타내면 다음과 같다.



표를 살펴보면 알 수 있듯이 client가 1일 때를 제외하고 Thread-based가 모든 영역에서 Event-driven을 미세하게 앞지르고 있음을 알 수 있다. 이는 하나의 프로세스로 작업을 하는 Event-driven 방식과는 다르게 Thread-based는 여러 개의 프로세스가 각각의 작업을 수행하기 때문이다. 하지만 예상한 것과는 다르게 두 방식 간의 차이가 크지 않게 측정되었다. 이는 reader-writer problem을 설정할 때 가중치를 설정하지 않고 코드를 작성했기 때문이라고 판단했다. 만약 read하려는 명령이 많이 존재해 write하는 작업에 lock이 발생하고 이에 따른 작업 진행이 지연돼 발생한 것으로 보인다.

(2) 워크로드 분석

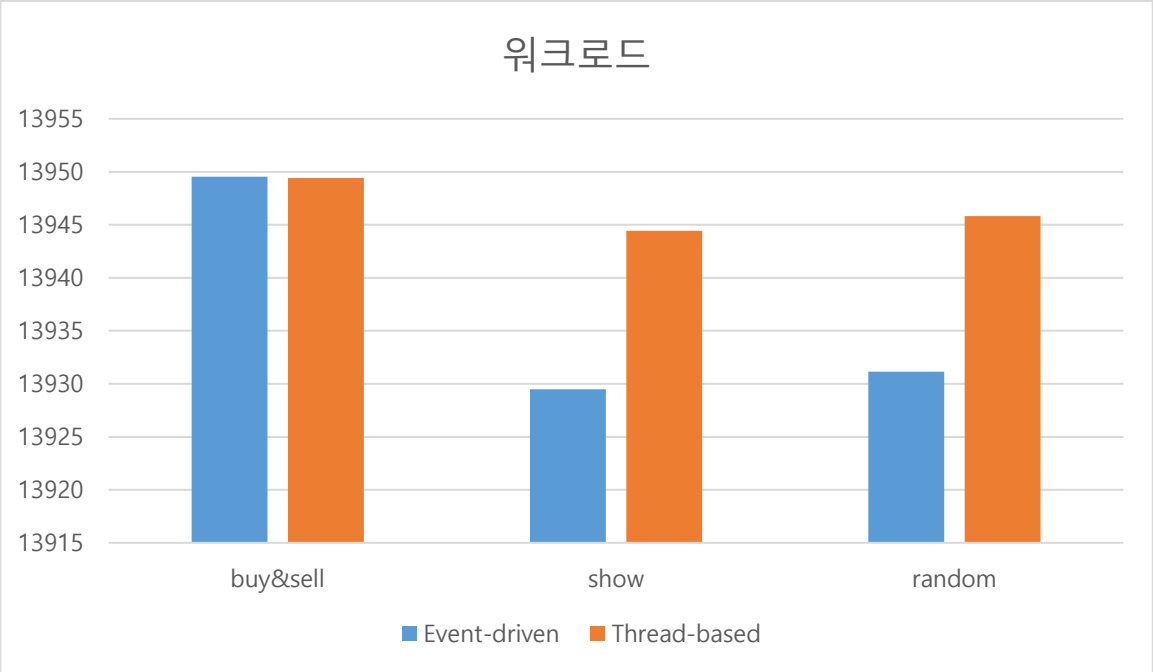
워크로드 분석은 buy와 sell만 있을 때, show만 있을 때, random한 요청이 있을 때 3가지로 나눠진다. 다른 조건에 영향을 받지 않기 위해 client의 수는 100개로 고정하고 client당 요청 수는 10개로 고정했다. 위의 측정에서와 마찬가지로 5번의 측정에 대한 평균값을 구하였으며 그 결과는 다음과 같다.

client 수	Event-driven		Thread-based	
	average	동시처리율	average	동시처리율
buy&sell	0.071687	13949.5306	0.071688	13949.41385
show	0.07179	13929.47784	0.071713	13944.43422
random	0.071782	13931.1467	0.071706	13945.83438

위의 표를 보면 알 수 있듯이 처음에 한 예상과는 다른게 show에서 더 많은 워크로드가 발생하는 것을 확인할 수 있었다. 이는 구현한 showlist함수의 특성 때문이라고 판단했다. 구현한 showlist 함수를 살펴보면 전체 list를 살피며 buf에 문자열을 이어 붙이는 구조를 가지고 있다. 따라서 특정한 id를 찾으면 끝나는 buy와 sell과는 다르게 전체 list를 전부 탐색하기 때문에 더 높은 워크로드를 가진다고 판단된다. Random한 경우에 대한 동시처리율이 show와 buy&sell의 중간에 위치하는 것 또한 이 영향이라고 볼 수 있다.

추가적으로 1번 확장성 측정에서와 다르게 오히려 Event-driven의 경우가 Thread-based 보다 미세하지만 조금 더 좋은 동시처리율을 가졌다. 이는 네트워크 환경이 수시로 변하기 때문에 그에 대한 영향과 최적화되지 못한 mutex lock으로 인한 결과라고 판단했다. 또한 item의 개수가 10개로 매우 작아 동시에 같은 id를 가진 item에 접근하려는 client가 많을수록 시간이 더 느려진다는 단점이 존재한다고 판단했다.

위의 표를 그래프로 나타내면 아래와 같다.



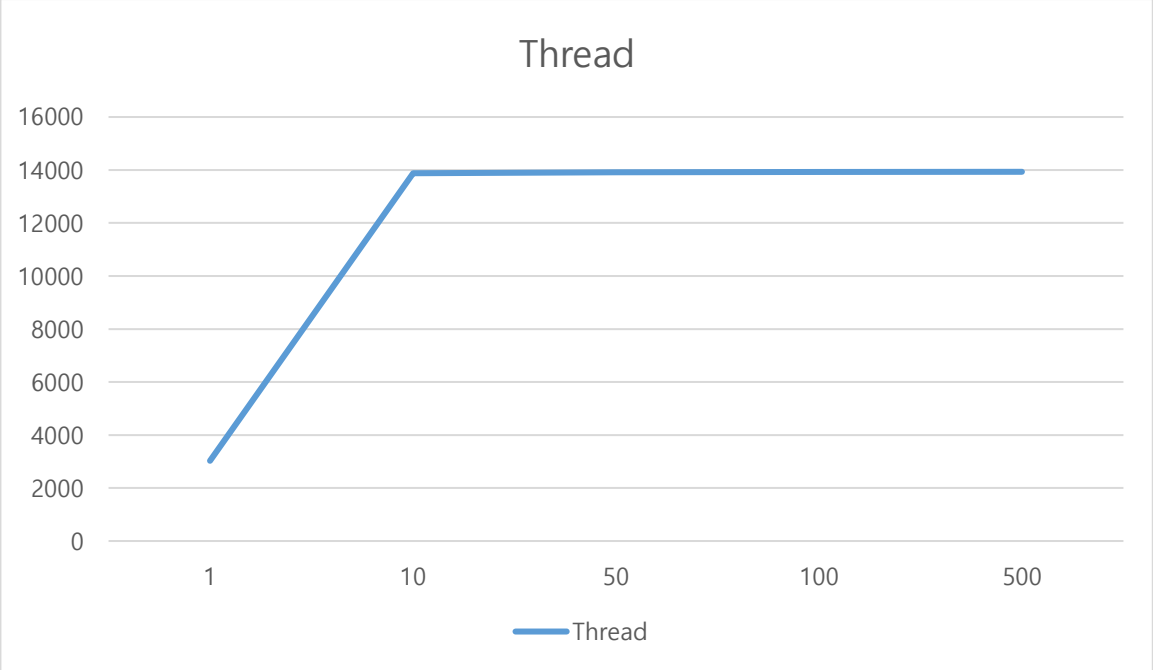
위의 그래프에서 보면 알 수 있듯이 buy와 sell에 있어서는 큰 차이를 가지지 않지만 show와 random의 경우에 thread-based server가 더 높은 동시처리율을 보이는 것을 알 수 있다. Thread- based와 Event-driven이 show명령어의 동시처리율 차이가 큰 이유는 그 구조에 있다. Thread-based는 각각의 Thread가 개별적을 돌면서 list에 접근해 값을 읽어오지만 Event-driven의 경우 한 client의 작업이 끝나야 다음 작업을 수행할 수 있기 때문에 더 오랜 시간이 걸려 동시처리율이 감소한다.

(3) Thread의 개수에 따른 동시처리율

Thread-based server는 Worker에 해당하는 Thread를 미리 생성한 후 작업을 진행한다. 따라서 Thread의 개수인 NTHREAD의 크기의 변화가 동시처리율에 어떤 영향을 미치는지를 파악하기 위해 진행하였다. 동일한 환경에서 작업을 수행하기 위해 client는 100개, client별 order 수는 10개, sbufsize는 1000으로 고정하고 진행하였다. 변수인 NTHREAD는 1개, 10개, 50개, 100개, 500개로 진행했다. 위와 마찬가지로 5번의 측정에 대한 평균값으로 도출하였다. 결과를 정리한 표는 다음과 같다.

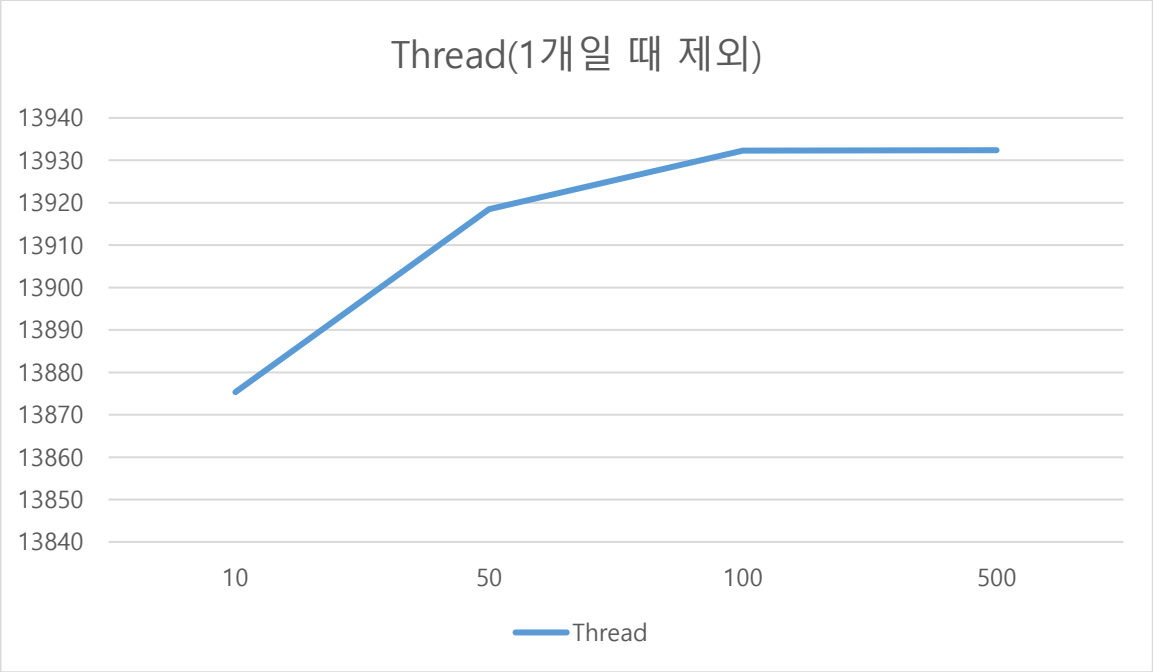
NTHREAD	average	동시처리율
1	0.33027	3027.822051
10	0.07207	13875.32191
50	0.071847	13918.46563
100	0.071776	13932.27244
500	0.071775	13932.3889

우선 NTHREAD가 1 일 때는 1개의 프로세서가 모든 처리를 도맡아 하는 것과 다르지 않다. 따라서 가장 높은 시간이 걸렸고 동시처리율 또한 매우 낮았다. 이후 thread를 10개로 늘린 후에는 앞에서 진행했던 다른 측정 결과 값과 비슷하게 나왔다. 하지만 10개의 thread만 작동하므로 나머지 90개의 접속은 sbuf에서 대기하고 있다. 이는 50개도 마찬가지이다. 둘 사이에 동시처리율 차이가 크게 발생하는 이유는 한 개의 thread가 평균적으로 몇 개의 client를 담당하는지에 따른 차이라고 분석했다. Thread가 10개일 때는 각각 10개의 client를 담당하지만 thread가 50개일 때는 각각 2개의 client만 담당한다. 이러한 차이는 100개와 500개일 때의 차이를 보고도 판단할 수 있다. Client의 수가 100명이기 때문에 실직적으로 100개의 thread로 작업하는 것과 500개의 thread로 작업하는 것은 동일하다. 위에서 설명했던 것처럼 sbuf에 요청이 들어오지 않으면 남은 thread는 sleep 하고 있기 때문이다. 위의 표를 그래프로 나타내면 다음과 같다.



Thread가 1개일 때를 포함한 그래프르 살펴보면 10, 50, 100, 500간의 차이가

거의 보이지 않는다. Thread가 1개일 때를 제외한 그래프는 다음과 같다.



위의 표를 보면 100개 전까지는 점차 늘어나가 100개와 500개 간의 차이는 거의 존재하지 않는 것을 확인할 수 있다.