Multicore Programming Project 2

담당 교수 :최재승

이름 :김연수

학번 :20180501

1. 개발 목표

concurrent programming에 대한 숙련도를 높이기 위하여, 여러 client들의 동시 접속및 서비스를 가능하도록 하는 concurrent stock server를 구축한다.

주식 서버는 client의 show, buy, sell, exit의 요청을 받아서 주식을 팔거나 사거나, 주식 현재 보유량에 관한 정보를 제공해주는 동작을 수행한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

event-based approach를 이용해서 concurrent stock server를 구현한다. event-based approach를 사용하는 경우 동작은 다음과 같다. client에게 동시에 요청이 들어왔을 때, 요청이 들어온 client들의 file descriptor가 모두 select함수 호출로 인해 ready_set에 기록되고, select는 pending input이 존재하는 fd의 개수를 리턴한다. 리턴된 숫자 만큼 loop를 돌아 각 client에 맞는 서비스를 제공하게 된다.

2. Task 2: Thread-based Approach

thread-based approach를 이용해서 concurrent stock server를 구현한다. thread-based approach를 사용하는 경우 동작은 다음과 같이 진행된다. server에서 미리 적정 수의 thread를 생성해 놓는다. 그 이후 client에게 동시 다발적으로 요청이 들어왔을 때, 각 client의 fd를 배열에 넣는다. 배열에 넣어놓으면, 각 thread는 배열에서 임의의 fd를 가져와서 서비스를 수행하게 된다.

3. Task 3: Performance Evaluation

performamce Evaluation을 구현하기 위해서 gettimeofday함수를 이용한다. client의 동작 이후 한 줄 띄우고 performance Evaluation을 나타내는 정보를 화면에 띄워준다. 화면에 띄워줄 정보는 running time, 동시처리율(request/time), number of request 정도로 한다.

나오는 정보를 취합해서 테이블과 그래프를 작성한다. 작성한 테이블과 그래프를 바탕으로 event-based approach와 thread-based approach의 performance를 분석한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

상술한 바와 같이 multi-client요청이 들어왔을 때, select함수를 호출해서 이를 탐지할 수가 있다. select함수가 한 번에 여러 개의 클라이언트를 감지할 수 있기 때문에, pending input이 들어온 file descriptor를 찾아서 ready_set에 표시를 해주고 들어온 pending input의 개수를 리턴해준다.

우리는 리턴 받은 개수와 ready_set에 담긴 fd정보를 활용하여 서비스를 제공하면 되는데, 여기서 listenfd에 pending input이 들어온 경우와 connfd에 pending input이 들어온 경우를 나눌 수가 있다.

listenfd에 pending input이 있다는 것의 의미는 지금 client에서 connect함수를 사용하여 server에 접속을 시도하고 있음을 뜻한다. 따라서 이 경우 FD_ISSET함수를 이용. listenfd가 ready_set에 있는 지 확인해서 있는 경우 아래와 같은 작업을 수행한다.

먼저, Accept함수를 호출해서 connection file descriptor를 받아온다. 이 connfd를 통해 서버와 클라이언트가 통신한다. add_client 함수를 호출해서 pool struct에 받은 connfd를 저장할 것이다. 좀 더 구체적으로, connfd를 read_set에 세팅해주고, clientfd 배열에 넣는다. clientfd는 connfd만을 따로 모아놓은 배열이다. 따로 모아놓는 이유는 check_client에서 이 모아진 connfd만을 이용해서 클라이언트에게 서비스를 제공할 것이기 때문이다.

add_client에서는 nready의 값을 하나 감소시키게 되는데, 그 이유는 다음과 같다. nready는 현재 connfd와 listenfd로 들어온 pending input에 대한 값이다. add_client에서 listenfd에 대한 처리를 이 함수에서 해줄 것이니 nready의 값을 하나 감소시키는 것이다.

다음 check_client함수에서는 드디어 connfd로 들어온 pending input에 대한 처리를 해줄 것이다. nready의 개수 만큼 loop를 돌면서, 하나하나씩 connfd로 들어온 pendin input을 확인한다. 확인 후 client에게 입력받은 메세지를 가지고 그에 맞는 서비스를 제공하게 된다.

  ✓ epoll과의 차이점 서술

epoll은 select의 단점을 보완하기 위해서 linux kernels 2.5.44 혹은 그 이상 버전에서 제공하는 i/O event notification이다. select의 단점은 많은 수의 fd를 검사하지 못한다는 점이다. 보통 1024개로 제한된다고 알고 있다. 또한 모든 fd에 대해서 FD_ISSET으로 체크

하는 것 또한 불필요한 작업이다. FD_SET을 계속 호출하는 것도 부하를 낳는다.

epoll은 select가 아니라 epoll_wait으로 event가 발생하기를 기다리게 된다. 여기서 select와 차이점은 다음과 같다. select는 관찰하는 fd에 대한 정보를 kernel에게 넘긴다. 하지만 epoll의 fd를 담은 저장소는 운영체제가 직접 관리한다. epoll_create를 호출하면 저장소에 해당하는 epoll Fd를 리턴해준다.

epoll_ctl은 관찰 대상이 되는 fd들을 등록 또는 삭제한다. 인자값으로 epoll fd, operate_enum, enroll fd, event, 가 들어간다. 각각 인자의 의미를 살펴보면 다음과 같다. epoll fd는 epoll create에서 얻은 fd를 의미한다. operate_enum은 수정 또는 삭제, 등록의 명령을 의미하는 인자다. EPOLL_CTL_ADD, EPOLL_CTL_Del, EPOLL_CTL_MOD 세가지중 하나의 값이 인자로 들어간다. enroll_Fd는 등록할 fd를 의미한다. event는 관찰 이벤트 유형을 의미한다.

이런 식으로 select에서의 불필요한 작업들을 없애줄 수 있다. 따라서 epoll은 더 많은 file descriptor에 대해 select 보다 효과적이고, 고도의 성능을 필요로 하는 프로그램에서 널리 쓰인다.

Task2 (Thread-based Approach with pthread)

- ✓ Master Thread의 Connection 관리

Pthread_create함수를 호출해 worker thread를 다수 생성한다. 생성한 이후 master thread에서는 while loop을 돌면서 반복해서 새로운 client를 기다린다. while문 안에서는 Accept 함수를 호출한다. Accept는 listenfd로 client의 pending input이 들어오기를 기다렸다가 들어온 경우에 connection file descriptor를 리턴하며 종료된다. 그리고나서 Accept함수에게서 리턴 받은 connfd를 sbuf에 넣어준다.

sbuf는 connfd를 저장해두기 위해서 생성한 배열이다. sbuf는 buf array를 포함해 여러 field를 가진다. 대표적으로 buf array에서는 들어온 connfd를 저장한다. 그리고 이 배열의 무결성을 지키기 위해서 mutex, slots, items 세 가지 semaphore가 field로 있다. slots은 insert를 하기위해 마련한 semaphore이다. items는 remove를 위해 마련한 semaphore이다. 만약 slots가 0이라면 그 때는 insert를 할 수 가 없고, item이 0이라면 remove를 할 수 없는 상태가 된다. 따라서 초기화 할 때 slots은 BUFSIZE만큼 item은 0으로 sbuf_init에서 선언해준다.

sbuf_insert 이후에 while문을 계속 반복하다가 sigint signal이 도착하면 종료한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread들은 처음 프로그램이 실행될 때, 대량으로 한번에 생산된다. Worker thread에서 sbuf_remove를 호출한다. 따라서 이 Worker thread들은 프로그램 실행 이후에 계속해서 sbuf에 connfd가 들어오길 기다리는 상태가 된다. master thread에서 sbuf_insert함수 호출로 인해 저장된 sbuf들을 worker thread에서 remove해서 가져간다.

여기서 중요한 것은 Worker thread와 master thread 간 data race가 발생할 수 있기 때문에, semaphore를 적절히 사용해줘야 한다는 것이다. 그렇기 때문에 위에서 말했듯 sbuf에 mutex, slots, items의 semaphore를 sbuf의 field로 두는 것이다. insert작업을 수행할 때는 slots을 하나 빼고 item을 넣는다. remove 작업을 수행할 때는 item을 빼고 slots을 하나 뺀다. insert를 하려고 하는데 slot이 없거나 remove하려고 하는데 item이 없는 경우는 수행되지 못한다.

- **Task3 (Performance Evaluation)**

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

metric : 각 방법(event-based, thread-based)에 대한 client 개수에 따른 동시처리율 변화를 분석

이 실험을 하는 목적은 multiclient의 concurrent request를 처리할 수 있도록 하는 server를 구현하는 것이기 때문에, concurrent request를 얼마나 잘 처리할 수 있는가가 관건이다. 여기서 event-based approach와 thread-based approach의 성능 차이를 분석해 볼 수 있도록, client의 개수에 따라 변화하는 동시처리율을 분석한다. 본 실험에서 동시처리율은 client request 처리량 / running time으로 계산했다.

동시처리율 측정을 server에서 측정할 수도 있고, client에서 측정할 수도 있겠지만, client에서 측정하는 것이 비교적 쉽고 정확하다는 판단 하에 본 실험에서는 client측에서 측정하는 것으로 한다.

client에서 running time을 측정하는 방법으로 본 실험에서는 gettimeofday함수를 사용한다. multiclient 파일에서 while문 시작하기 바로 직전에 gettimeofday(&start, 0)으로 시작하는 시점의 시간을 측정, 모든 child process가 종료되고 난 이후 gettimeofday(&end, 0)으로 종료 시간을 측정한다. 측정한 종료시간과 시작시간의 차이를 계산해 running time을 구한다. 구한 runningtime으로 클라이언트 요청개수(num_client * oderperclient)를 나누어 주면 server의 동시처리율을 계산할 수 있다.

✓ Configuration 변화에 따른 예상 결과 서술

client의 요청타입은, show, buy, sell 세가지다. show는 read기능, buy와 sell은 write기능이다. read는 한 번에 여러 개의 작업이 수행될 수 있지만, write는 그렇지가 않다. write는 read와도 겹칠 수 없고, 다른 thread가 write하는 도중에도 접근할 수 없다. 이런 점을 고려해서 show만 요청으로 주는 경우, buy/sell만을 요청으로 주는 경우를 나누어 실험할 것이다. 그렇게 되면 분석 해봐야할 것은 세 가지 경우다. random request, show request, buy/sell request 이 세 가지 경우에 대한 동시처리율이 어떻게 나오는 지를 분석한다.

show request만 존재하는 경우, thread-based approach 서버는 여러 개의 thread가 동시에 reader역할을 하여 show작업을 수행할 수가 있기 때문에 매우 빠른 동시처리율을 보일 것으로 예상한다. 반면 event는 각각의 요청을 하나하나 반복문에서 처리하므로 thread에 비해 많이 뒤쳐지는 성능을 보일 것으로 예상한다.

buy/sell request만 존재하는 경우, thread-based approach 서버는 여러 개의 thread가 이미 생성되어 request요청이 들어오기를 기다리고 있다고 할지라도, 한번에 한 개의 요청밖에 처리하지 못한다. writer는 동시에 여러 개가 작동될 수 없도록 설계되기 때문이다. event-based approach 또한 for loop를 돌면서 한 번에 한 개의 요청을 처리한다. 따라서 buy/sell request만 존재하는 경우에는, 두 approach 모두 비슷한 성능을 가질 것으로 예상한다.

뿐만 아니라, 한 개의 stock만 존재할 때, write 명령 만을 내리는 경우도 살펴본다. thread-based의 경우 다른 stock에 대한 write작업을 동시에 수행할 수는 있지만, 같은 stock에 대해서는 동시에 작업을 수행할 수 없다. 따라서 여러 개의 stock이 존재할 때보다 더 낮은 성능을 가질 것이라 예상한다.

C. 개발 방법

- task_1

일단 stock item들을 담는 구조체는 아래와 같이 필요한 정보 들로만 구성했다.

```
typedef struct {
    int id;
    int left_stock;
    int price;
} stock_item;
```

event-based approach를 구현하기 위해서는 file descriptor들을 담아 놓을 구조체가 필요하다. 구조체를 아래과 같이 선언한다. 각 field에 대한 설명은 주석을 참고하면 된다.

```c
typedef struct {    //Represent a pool of connected descriptors
    int maxfd;      //Largest descriptor in read_set
    fd_set read_set;    //Set of all active descriptors (client_fd + listen_fd)
    fd_set ready_set;   //Subset of descriptors ready for reading
    int nready;         //Numver o fready descriptors from select
    int maxi;           //High water index into client array
    int clientfd[FD_SETSIZE];       //Set of active descriptors
    rio_t clientrio[FD_SETSIZE];    //Set of active read buffers
} pool;
```

main 함수는 아래와 같다.

```c
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;  /* Enough space for any address */  //line:netp:echoserveri:sockac
    char client_hostname[MAXLINE], client_port[MAXLINE];
    static pool pool;

    if (argc != 2) {…

    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool, FILENAME);
    Signal(SIGINT, sigint_handler);

    while (1) {
        /* Wait for listening/connected descriptor(s) to become ready*/
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
        /* If listening descriptor ready, add new client to pool*/
        if (FD_ISSET(listenfd, &pool.ready_set)) {
            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);
        }
        check_clients(&pool);
    }
    update_stock_file(FILENAME);
    exit(0);
}
```

main에서 하는 역할을 간단히 설명하자면, 다음과 같다.

listenfd를 받고 init pool을 호출해 pool 구조체 초기화를 시켜준다. init_pool 코드는 아래와 같다.(함수 아래 파일에서 정보 가져오는 부분 생략)

```c
void init_pool(int listenfd, pool *p, char *filename) {
    /*Initially, there are no connected descriptors*/
    int i;
    p->maxi = -1;
    num_stocks = 0;

    for(i=0;i<FD_SETSIZE;i++)
        p->clientfd[i] = -1;

    /*Initially, listenfd is only member of select read set*/
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
```

sigint_handler를 세팅한다. sigint를 받으면 update_stock_file을 호출하고 종료. 서버가 종료될 때 file에 item을 업로드 한다.

```c
void sigint_handler(int signum){
    update_stock_file(FILENAME);
    exit(0);
}
```

다음 while loop를 돌면서 select로 pending input감지하고 listenfd인 경우와 connfd의 경우에 다르게 처리를 해준다. listenfd로 pending input이 들어온 경우, FD_IFSET에서 감지하여 클라이언트와 소통할 수 있는 connfd를 받아 add_client로 pool에 저장해준다. connfd에서 pending input이 들어온 경우에는 check_clients함수를 호출해 들어온 명령에 맞는 서비스를 제공한다. 자세한 코드는 아래와 같다.

```c
void add_client(int connfd, pool *p) {
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) {      //Find available slot
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the poll */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool high water mark */
            if (connfd > p->maxfd)
                p->maxfd = connfd;
            if (i > p->maxi)
                p->maxi = i;
            break;
        }
    }
    if (i == FD_SETSIZE)
        app_error("add_client error: Too many clients");
}
```

add_client 함수에서 중요한 부분은 p->clientfd[i] = connfd, FD_SET(connfd, &p->read_set) 이다.

connfd를 read_set에 추가해주고, clientfd에도 추가해준다. read_set에 넣어주는 이유는 select에서 감지하기 위해서이고, clientfd에 추가하는 이유는 pending input이 들어온 경우, 저장한 fd를 빼내와서 서비스를 제공해주기 위해서이다. 나머지 설명은 주석을 참고하자.

```c
void check_clients(pool *p) {
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i=0;(i<=p->maxi) && (p->nready > 0);i++) {     //nready는 처리해야할 event 개수
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        //If the descriptor is ready, echo a text line from it
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                printf("Server received %d bytes on fd %d\n", n, connfd);
                handle_client_request(connfd, buf);
                // Rio_writen(connfd, buf, MAXLINE);
            }

            //EOF detected, remove descriptor from pool
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

check_clients에서는 nready만큼 loop를 돌아 client에게 message를 받고 그에대한 서비스를 제공한다. 서비스는 handle_client_request함수를 호출해서 제공하도록 한다. 서비스 제공함수의 구체적인 코드와 자세한 설명은 그다지 중요하지 않으니 넘어가도록 하자.

- task2

thread-based approach에서는 stock_item에 semaphore를 추가적으로 넣어줘야한다. 아래에서 mutex와 w를 field로 넣어줬다. w는 write을 할 수 있는 권한이다. mutex는 readcnt변수를 조작할 때 가지고 있어야하는 권한이다.

```c
typedef struct {
    int fd;
    int id;
    int left_stock;
    int price;
    int readcnt;
    sem_t mutex, w;
} stock_item;
```

sbuf_t는 connfd를 넣어줄 구조체다. connfd를 구조체 내에 존재하는 배열에 insert, remove할 때마다 semaphore가 필요하다. 따라서 구조체 내에 mutex, slots, item을 선언한다. 각각의 역할은 앞서 설명한 바와 같다.

```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 acceesses to buf
    sem_t slots;       //Count available slots
    sem_t items;       //Count available items
}sbuf_t;
```

sbuf와 관련된 함수의 구체적인 코드는 아래와 같다.

```c
void sbuf_insert(sbuf_t *sp, int item) {
    P(&sp->slots);                          //비어있는 슬롯
    P(&sp->mutex);
    if(sp->rear == 0)
        gettimeofday(&start, 0);            // 시간측
    sp->buf[(++sp->rear)%(sp->n)] = item;
    V(&sp->mutex);
    V(&sp->items);                          //item을 1증
}

int sbuf_remove(sbuf_t *sp) {
    int item;
    P(&sp->items);
    P(&sp->mutex);
    item = sp->buf[(++sp->front) % (sp->n)];
    V(&sp->mutex);
    V(&sp->slots);
    return item;
}
```

main함수의 코드는 아래와 같다. 상술한 바와 같이 master-thread에서 accept로 connfd를 받아서 sbuf_insert를 호출해 구조체 내의 배열에 넣어준다. loop를 돌면서 pending input이 생길때마다 반복한다.

```c
int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    if (argc != 2) {…

    listenfd = Open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);

    Signal(SIGINT, sigint_handler);
    get_stock_from_file();

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

    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        //connfd를 버퍼에 넣어줄 것임. connfd가 버퍼에 들어가 있다가 thread가 하나씩 꺼내감.
        sbuf_insert(&sbuf, connfd);
    }
    update_stock_file(FILENAME);
    exit(0);
}
```

worker-thread에서는 배열에 원소가 들어오기를 기다렸다가 sbuf_remove를 호출해 connfd를 받아와 서비스를 제공한다.

```
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf);
        handle_client_request(connfd);
        Close(connfd);
    }
    return NULL;
}
```

중요한 것은 Worker thread간에 충돌이 일어나지 않도록, data race가 일어나지 않도록 적절히 semaphore를 사용해야하는 것인데, 그렇다면 아래 handle_show_request와 sell_request를 보자. 전자를 reader, 후자를 writer 역할로 해석할 수 있다. 따라서 전자에서는 중복으로 접근이 가능하고 후자에서는 불가능하다. readcnt에 접근할 때는 항상 mutex를 닫고 열고, 첫 번째와 마지막 reader가 들어올 때와 나갈 때 w를 닫고 연다.

```
void handle_show_request(int connfd) {
    int i;
    char buf[MAXLINE] = "show\n";
    char temp[MAXLINE];

    for (i = 1; i <= num_stocks; i++) {
        stock_item *item = &stocks[i];
        P(&item->mutex);
        item->readcnt++;
        if (item->readcnt == 1)
            P(&item->w);
        V(&item->mutex);

        sprintf(temp, "%d %d %d\n", item->id, item->left_stock, item->price);
        strcat(buf, temp);

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

    Rio_writen(connfd, buf, MAXLINE);
}
```

handle_sell_request는 shared item에는 동시 접근이 불가능하다. item을 건드리기 전에 w를 닫고 값을 수정하고 나서 w를 열어준다.

```c
void handle_sell_request(int connfd, int id, int num) {
    int i = 1;
    char buf[MAXLINE];
    char temp[MAXLINE];
    sprintf(buf, "sell %d %d\n", id, num);
    while(1) {
        if(i > num_stocks) break;
        stock_item *item = &stocks[i];
        if(item->id > id) i = (2*i);
        else if(item->id < id) i = (2*i) + 1;
        else {
            P(&item->w);
            item->left_stock -= num;
            sprintf(temp, "[sell] success\n");
            strcat(buf, temp);
            V(&item->w);
            break;;
        }
    }
    Rio_writen(connfd, buf, MAXLINE);
}
```

task_3

성능을 평가하기 위해 multiclient를 이용했다. multiclient로 성능을 측정해보기 위해서 코드를 수정한 부분이 있는데, 자세한 내용은 다음과 같다. 일단 orderperclient를 20으로 설정했다. client당 request가 너무 적으면 thread와 event간 성능차이가 확연하게 드러나지 않을 가능성이 있고, 너마 많으면 프로그램에 부하가 발생할 수 있으므로, 20정도가 적당하다고 판단했다. 또한 usleep(1000000) 코드를 주석처리 해줬다. 시간 측정은 gettimeofday함수를 이용해 아래와 같이 코드로 구현했다. while문 시작하기 전에 gettimeofday(&start, 0)을 호출 자식 프로세스가 모두 끝나고 나서 gettimeofday(&end, 0)을 호출한다.

```c
gettimeofday(&start, 0);
fork for each client process    */
while(runprocess < num_client){
```

```c
// usleep(1000000);
```

```c
gettimeofday(&end, 0);
e_usec = ((end.tv_sec * 1000000) + end.tv_usec) - ((start.tv_sec * 1000000) + start.tv_usec);
printf("\ntotal process time: %6.2f seconds\n", (e_usec/1000000.0));
printf("concurrent workload (request / time): %6.2f per second\n",
(num_client*ORDER_PER_CLIENT)/(e_usec/1000000.0));
printf("number of request: %d request\n", num_client*ORDER_PER_CLIENT);
```

또한 random request, show request, buy/sell request의 각 case를 비교 분석하기 위해서, while loop안에 존재하는 for문 안에 아래과 같은 코드를 추가했다.

```
for(i=0;i<ORDER_PER_CLIENT;i++){
    int option = rand() % 3;
    // int option = 0;          //show
    // int option = rand() % 2 + 1;
```

task_3를 수행하기 위해서 stockserver에서 따로 작업한 부분은 없다. multiclient를 수정하고 client측에서 시간을 측정하는 것으로 충분하다고 판단했다. 나머지는 직접 client의 개수를 변화시키면서 확인해본다.

구현 결과

event-based approach thread-based approach 둘 다 정상 작동하는 것을 확인했다. client에서 명령을 입력하면 stock item 제고 현황을 업데이트 하고, client에게 성공 메시지를 전달한다.

동시에 client에서 접근하는 경우에도 server에서 정상 처리하는 것을 확인했다. concurrent server를 thread-based approach로 구현할 때 가장 우려 했던 부분인 data race가 발생하지 않는 것을 확인했다.

multiclient로 4개의 client에서 server에 동시요청을 보냈을 때 캡쳐:

```
cse20180501@cspro9:~/task_1$ ./multiclient 127.0.1.1 60009 4
child 2243528
buy 2 1
[buy] success
sell 10 8
[sell] success
sell 6 8
[sell] success
show
4 387 493
2 778 6916
6 363 28
7 691 60
9 541 3427
3 794 8336
10 173 5737
1 384 887
5 650 1422
8 764 3927
```

3. 성능 평가 결과 (Task 3)

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

1. show와 buy/sell을 섞어서 random하게 request를 보내는 경우

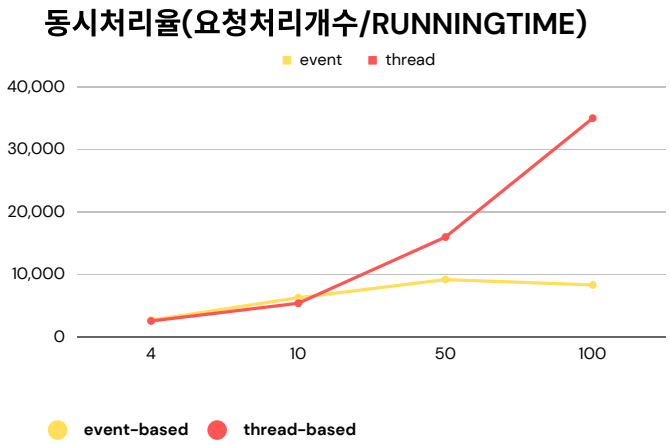
ex) event-based approach client 100

```
total process time:   0.29 seconds
concurrent workload (request / time): 6984.80 per second
number of request: 2000 request
```

ex) thread-based approach client 100

```
total process time:   0.06 seconds
concurrent workload (request / time): 32946.76 per second
number of request: 2000 request
```

graph)



random한 request를 보낸 결과 위와 같은 그래프가 도출되었다. 가로는 동시 접속client의 수, 세로는 동시처리율을 의미한다. client의 개수가 적은 경우, thread-based approach와 event-based approach간 성능차는 그리 크지 않지만, 동시 접속 client가 많아지는 경우 성능 차가 벌어진다. event의 경우 동시처리율이 쭉 증가하는 추세를 보이다가 50부근에서 꺽이는 반면, thread-based의 경우 꺾이지 않고 계속 증가하는 것을 확인할 수 있었다. thread의 경우에도 event와 같이 동시 접속 client가 많아지면 동시처리율이 꺾이는 지점이 있을 것으로 예상된다. 하지만 이 그래프를 보면, 100까지는 계속 증가하는 추세를 이어간다는 것을 확인할 수 있다.

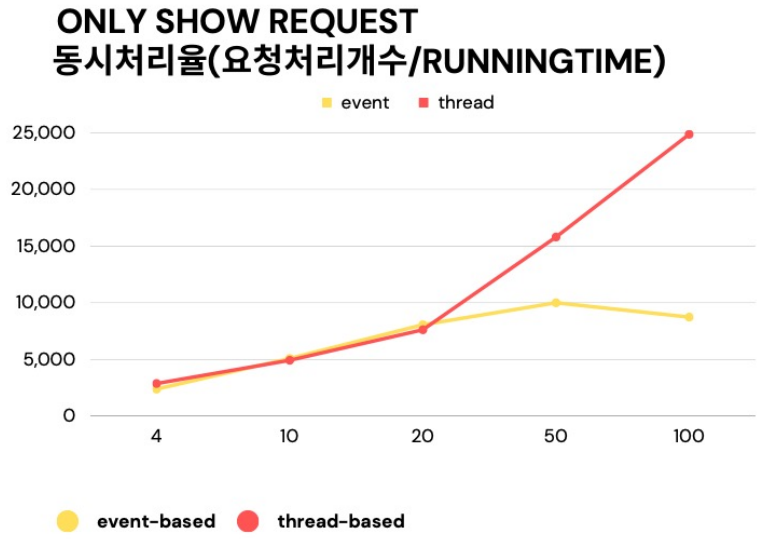
2. show request만 보내는 경우

ex) event-based approach: client 100

```
total process time:   0.22 seconds
concurrent workload (request / time): 9062.20 per second
number of request: 2000 request
```

ex) thread-based approach: client 100

```
total process time:   0.09 seconds
concurrent workload (request / time): 23267.18 per second
number of request: 2000 request
```



show request만 server로 들어오는 경우, thread-based 그래프를 보면 현격하게 동시처리율이 줄어든 것을 확인할 수 있다. 그리고 event-based 그래프는 random과 유사한 양상을 보인다. 자세한 비교분석은 buy/sell 그래프를 보고나서 하도록 하겠다.

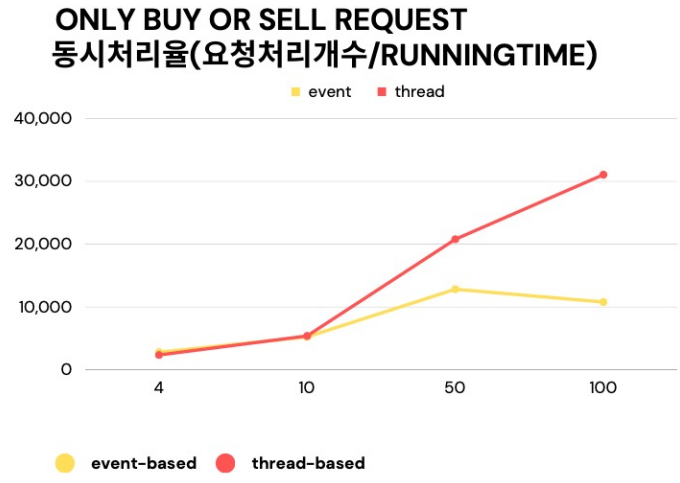
3. buy/sell request만 보내는 경우

ex) event-based approach: client 100

```
total process time:   0.19 seconds
concurrent workload (request / time): 10640.62 per second
number of request: 2000 request
```

ex) thread-based approach: client 100

```
total process time:   0.06 seconds
concurrent workload (request / time): 33988.17 per second
number of request: 2000 request
```



buy/sell request만 server에 입력으로 들어오는 경우, thread-based 그래프와 event-based 그래프 둘 다 random과 유사한 양상을 보인다.

여기서 의문점이 생겼다. 수업시간에 배운 내용으로 따지면, 분명 show request만 주어졌을 때가 동시처리율이 더 높아야 한다. 하지만 실험 결과는 그렇지 않은 것으로 밝혀졌다. buy/sell request case에서 오히려 동시처리율이 더 높은 것으로 드러났다. 그 이유에 대해 고민해본 결과는 다음과 같다.

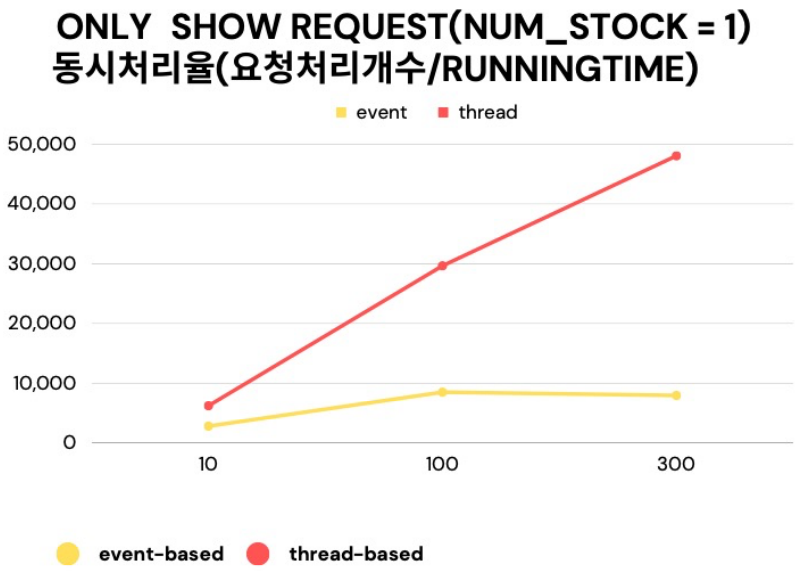
첫 번째로, show 자체의 시간 복잡도 때문이다. show는 buy/sell과 달리 n개의 stock의 input에 대해서 O(n)만큼의 시간 복잡도를 가진다. 하지만, buy/sell에서 명령으로 입력받은 id를 찾는 데 소요되는 시간 복잡도는 O(logn)이다. 그 이유는 show의 경우 모든 stock을 조회해야 하나, buy/sell의 경우 binary tree를 이용한 search가 가능하기 때문이다. 이 때문에, 동시처리율의 계산에서도 차이가 날 수 있다.

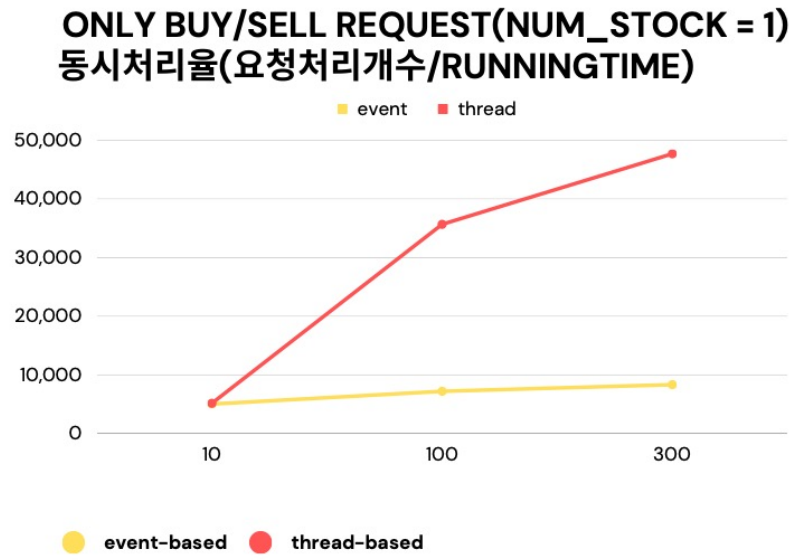
두 번째로, reader와 writer의 작동방식은 semaphore에 의존한다. semaphore는 일련의 코드가 원자성을 가지도록 해줌으로서, global variable이 훼손되는 것을 막아준다. 이 때 worker thread가 buy/sell의 서비스를 수행속도가 client에서 들어오는 입력 텀(즉, 배열에 connfd가 들어오는 텀)보다 빠르다면, semaphore를 사용하는 의미가 없어진다. thread가 중복되어 실행되는 순간이 없어지기 때문이다. 이런 경우에, buy/sell연산의 동시처리율은 client의 증가에 영향을 받지 않는다. 하지만, 그래프에서 보다시피 client의 동시접속이 증가할 수록 동시처리율이 증가한다는 것 자체가 thread가 중복으로 실행되고 있음을 나타내므로 이것은 틀렸다.

세 번째로, client의 동시접속 수가 적어서 제대로 확인이 안된 경우다. 그래서 client의 개수를 더 늘려서 확인해본 결과 300쯤에서 thread와 event 둘 다 동시처리율이 증가하지 않는 것을 발견했다. 하지만 client가 300개인 경우에서도 show의 케이스에서 buy/sell보다 동시처리율이 낮은 것을 관찰할 수 있었다.

네 번째로, 동시처리율은 stock의 개수에 따라서도 영향을 받는다. reader끼리는 중복으로 실행이 가능하지만, writer는 reader 또는 writer 둘 중 하나라도 존재한다면 끝날 때까지 순서를 기다려야 한다. 따라서 stock의 개수가 적어지는 경우 buy/sell request case에서 동시처리율이 감소한다. item당 w semaphore가 존재하기 때문에 2가지 다른 item에는 동시 접근이 가능하지만, 1가지 item에는 동시 접근이 불가능하기 때문이다. 하지만 동시에 show request에서도 stock의 개수가 적어지는 경우 동시처리율을 감소시키는 요인이 존재한다. stock의 개수가 감소할 수록 buy/sell보다 show의 case에서 시간 복잡도가 비교적 많이 줄어들기 때문이다.

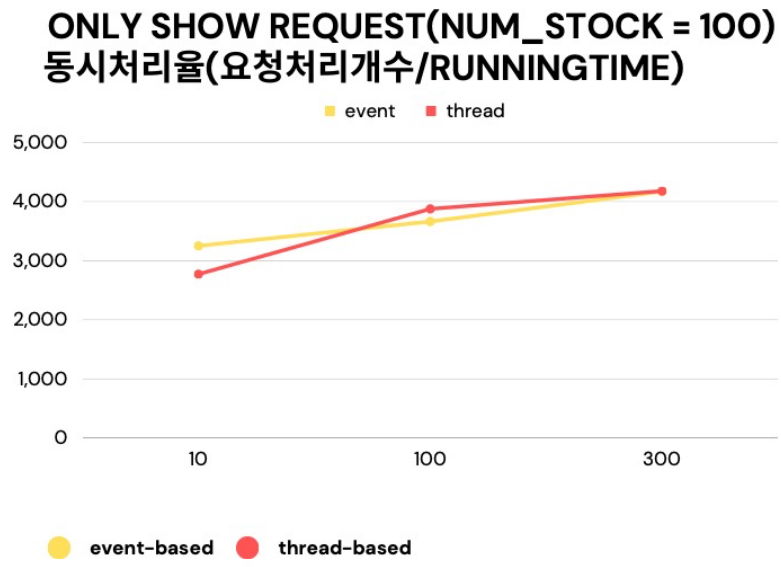
따라서 아래 실험에서는 stocks의 개수를 변화시켰을 때 두 case에 미치는 각각의 요인 중 어떤 것이 더 치명적으로 작용하는 지를 확인해 본다. stock의 개수를 변화시켜서 event-based approach와 thread-based approach 둘 사이 어떤 변화가 생기는 지 관찰해보도록 한다.





stock의 개수가 1인 경우 각 case별로 multiclient의 개수를 변경해 동시처리율을 관찰한 결과 위와 같은 그래프가 도출되었다. 위 그래프는 show case, 아래의 그래프는 buy/sell의 case다. 이번에는 더 명확하게 보기 위해서 client의 개수를 300개까지 늘려봤다.

결과적으로 두 그래프가 비슷한 양상을 보였다. stock의 개수를 10으로 놓고 설정한 앞선 실험과 유사한 결과를 확인할 수 있다. stock의 개수가 10이든 1이든 두 case 모두 동시처리율이 client의 숫자에 대해 큰 변화를 보이지 않았다.



(buy/sell request(num_stock = 100)의 event-based approach를 실험하는 도중 multiclient가 100일 때 프로그램이 중단되는 오류가 생겨 event-based case의 그래프를 측정하지 못했다.)

stock의 개수를 100으로 놓고 동시처리율을 관찰한 결과, 유의미한 차이를 관찰할 수 있었다. show case의 경우 stock의 개수가 늘어남에 따라서 두 케이스 모두 동시처리율이 현격하게 낮아지는 것이 확인되었다. 반면 buy/sell request의 경우 thread case만을 따져봤을 때 앞서 stock의 개수가 1인 경우와 달라진 것이 없으므로, 자료구조의 영향을 매우 많이 받는 다는 것을 발견할 수가 있었다.

결론적으로, 오직 show request만을 보낸 case와 buy/sell request만을 보낸 case와 동시처리율 차이는 reader와 writer사이에 중복 접근이 불가능한 요소로 인한 것 보다는 stock의 개수에 더

많은 영향을 받는 다는 것이 밝혀졌다. 이는 buy/sell의 서비스를 처리할 때보다 show 서비스를 처리할 때 모든 stock을 조회해야 하는 필요성에 따라 시간 복잡도가 더 크기 때문에 발생하는 것이다.