Multicore Programming Project 2

담당 교수 : 박성용 교수님

이름 : 장준영

학번 : 20211584

1. 개발 목표

이번 프로젝트에서는 Concurrent server를 Event-based와 Thread-based 두 가지 방법으로 구축한다. server는 stock 정보를 파일로부터 읽어와 BST(AVL)에 저장한다. 이후 정보를 바탕으로 client들에게 서비스를 제공하고, server가 종료될 때마다 stock 정보를 다시 파일에 저장한다. 서비스에는 주식을 구매하는 sell, 판매하는 buy, 확인하는 show, 접속을 종료하는 exit이 있다. Concurrent server이기 때문에 여러 client가 동시에 서비스를 요구해도 순차적으로 모든 요청을 해결할 수 있어야 한다.

Event-based concurrent server에서는 concurrency를 확보하기 위해 connected fd를 저장하는 pool을 사용한다(I/O Multiplexing). connect가 되면 connected fd를 pool에 넣고, 지속적으로 check_client 함수를 통해 pool에 있는 fd를 관리한다. check_client에서는 connected fd가 현재 들어온 만큼만 읽어들여 처리한다.

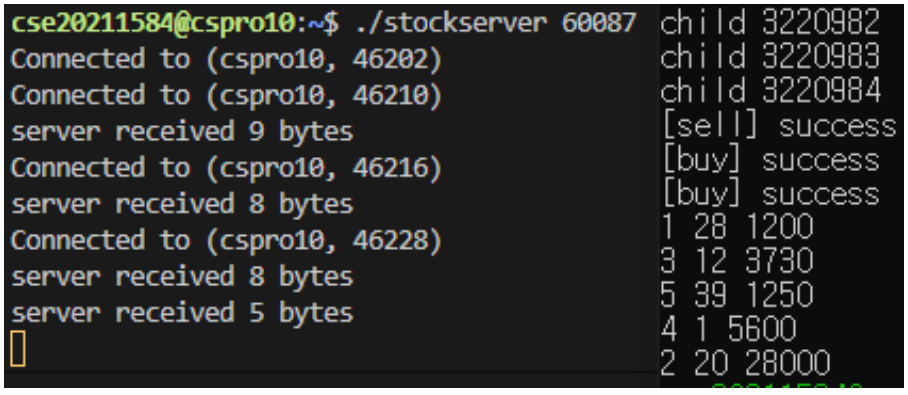
Thread-based concurrent server에서는 concerrency를 확보하기 위해 multi thread를 사용한다. 나의 코드에서는 connect마다 thread를 생성해 발생하는 overhead를 줄이기 위해 main에서 thread를 여러 개 미리 생성해두고, shared buffer에 connected fd가 들어올 때마다 thread가 처리할 수 있도록 구현하였다. thread 함수는 마찬가지로 connected fd를 읽어들여 처리한다.

두 가지 방법으로 구현한 concurrent server를 동일한 client 조건에 대한 워크로드나 소요 시간 등의 요소로 비교하고 분석한다.

2. 개발 범위 및 내용

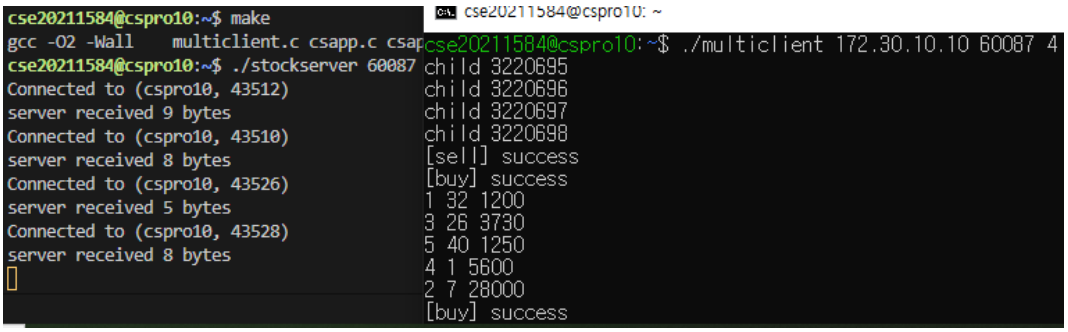
A. 개발 범위

1. Task 1: Event-driven Approach



client가 보내는 show, sell, buy, exit 네 가지 요청을 server가 제대로 수신하고 알맞은 명령을 수행한다. I/O Multiplexing을 통해 동시에 여러 client가 server에 접속해도 concurrent하게 명령을 수행할 수 있다.

2. Task 2: Thread-based Approach



client가 보내는 show, sell, buy, exit 네 가지 요청을 server가 제대로 수신하고 알맞은 명령을 수행한다. Multi thread의 producer-consumer 방식을 통해 동시에 여러 client가 server에 접속해도 concurrent하게 명령을 수행할 수 있다. 또, 성능을 위해 readers-writers 방식을 이용해 suspend되는 thread를 최소화하였다.

3. Task 3: Performance Evaluation

성능 분석을 위해 multiclient.c를 이용한 클라이언트의 개수나 워크로드에 따른 서버의 클라이언트 요청 동시 처리 속도 등을 확인하였다. 분석 결과, event-based server보다 thread-based server가 다방면에서 효율이 좋은 것을 확인할 수 있었다. 이때, 변인을 확실히 통제하기 위해 stock의 정보를 저장하는 binary tree가 height가 최소인 균일한 BST(AVL)가 되도록 구성하였다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Event-based server는 concurrency를 위해 연결되어있는 connected fd를 주기적으로 확인하고, 만약 연결된 client가 작업을 하지 않더라도 꾸준히 다른 client의 요청을 수행해야 한다. 이렇게 하나의 프로세스에서 여러 I/O Event를 처리하는 것을 pool 구조체를 이용한 I/O Multiplexing으로 구현할 수 있다.

```
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;
```

Pool은 connected fd를 저장하는 file descriptor의 set의 정보를 담은 구조체이다. fd_set은 file descriptor를 저장하는 set으로, FD_SET, FD_ISSET 등의 명령어로 집합의 상태를 조절하거나 점검할 수 있다. server는 main의 infinite while loop에서 connection과 listen을 점검한다. 만약 listening fd에 pending된 client의 입력이 있다면 이를 accept로 connect하고 pool에 넣어준다. 또, pool 안에서 connected fd에 pending된 입력이 있다면 이를 처리한다. 이렇게 pool에 connected fd를 삽입하고 client의 요청을 반복적으로 처리하면 kernel에서 스케줄링을 하듯이 한 프로세스에서 여러 클라이언트의 요청을 동시다발적으로 처리할 수 있다.

✓ epoll과의 차이점 서술

epoll 함수도 select와 마찬가지로 file descriptor의 변화를 관찰하여 알려주는 함수이다. kernel을 통해 I/O Event를 감지하면 다시 프로세스로 돌아와 알려주는 역할을 한다.

select와 epoll의 차이점은 작동 방식이다. select 함수는 호출되면 유저 프로세스에서 사용하던 file descriptor set을 kernel에 모두 복사한다. 이후 복사된 set을 kernel이 탐색하며 I/O Event를 감지한다. 하지만, epoll 함수는 호출되면 I/O Event를 감지하고 싶은 descriptor들을 위한 메모리 공간을 따로 할당하여 kernel로 감지한다. 따라서, select 함수는 호출 시마다 kernel로 file descriptor set을 복사해 탐색하고, epoll 함수는 file descriptor마다 호출 시에 한 번만 메모리를 할당한다. 이런 방면에서는 select보단 epoll 함수가 overhead가 적은 효율적인 함수이다.

- Task2 (Thread-based Approach with pthread)

✓ Master Thread의 Connection 관리

내가 구현한 코드에서 master thread는 main thread이다. Main thread에서는 미리 여러 개의 worker thread를 생성해두고 sleep시킨다. 이는 shared buffer(이하

sbuf)의 counting semaphore를 통해 관리된다. sbuf는 모든 thread가 공유하는 buffer로, 여기에 connection이 이루어진 fd를 넣어 buffer가 채워진 만큼 worker thread를 깨워 처리한다. 이는 counting semaphore인 slots, items 두 개로 이루어진다. slots은 sbuf에 비어있는 칸의 수를 의미하며, 최초에는 buffer의 크기로 초기화된다. items는 sbuf에 채워진 칸의 수를 의미하며, 최초에는 0으로 초기화된다.

새로운 connected fd가 발생하면 main thread는 sbuf에 이를 삽입하고 slots에 p operation을, items에 v operation을 실행한다. 그럼 slots는 하나가 줄고, items는 하나가 늘어난다. 이를 통해 sbuf에 삽입된 fd가 있다는 것을 worker thread가 감지하고, sbuf에서 fd를 하나 꺼내와 slots에 v operation, items에 p operation을 실행한다. 꺼내온 fd는 worker thread에서 처리된다. 만약 slots가 0이라면 main thread가 sbuf에 삽입을 진행할 수 없도록, items가 0이라면 worker thread들이 sleep하도록 처리해두었다. 이런 방식을 producer-consumer 방식이라고 한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

앞서 설명한 sbuf를 이용해 worker thread pool을 관리하므로, 이에 대해 조금 자세히 설명하겠다. sbuf_t 구조체는 fd를 저장하기 위한 정수 buffer, buffer의 크기 n, circular queuing을 위한 front, rear, critical section 관리를 위한 mutex, counting semaphore인 items, slots로 이루어져있다. buffer가 가득 찼을 때 main thread가 fd를 삽입하려 하면 overflow가 일어나고, buffer가 비어있는데 worker thread가 suspend 되지 않고 작동한다면 overhead가 발생할 수 있다. 이를 효율적으로 막기 위해 counting semaphore를 사용할 수 있다. 앞서 slots가 0이면 main thread가 buffer에 fd를 삽입하지 못하도록 막는다고 했는데, 과정은 다음과 같다.

```
void sbuf_insert(sbuf_t *sbufp, int elem) {
    P(&sbufp->slots);
    P(&sbufp->mutex);
    sbufp->buf[(++sbufp->rear) % (sbufp->n)] = elem;
    V(&sbufp->mutex);
    V(&sbufp->items);
}
```

main thread가 connection을 감지하면 실행하는 sbuf_insert 함수이다. 첫 번째 줄과 마지막 줄을 보면 p, v operation으로 함수 전체가 묶여있는 것을 볼 수 있다. 만약 buffer가 가득 차 slots가 0이 된다면 맨 처음의 p에 의해 insert가 suspend된다. 그리고, insert가 성공적으로 실행되고 나서는 마지막의 v에 의해 buffer에 채워진 칸의 수를 의미하는 items가 1 증가한다.

```
int sbuf_remove(sbuf_t *sbufp) {
    int elem;

    P(&sbufp->items);
    P(&sbufp->mutex);
    elem = sbufp->buf[(++sbufp->front) % (sbufp->n)];
    V(&sbufp->mutex);
    V(&sbufp->slots);

    return elem;
}
```

sbuf에서 하나의 요소를 꺼내는 sbuf_remove 함수이다. 역시 첫 번째 줄과 마지막 줄에 p, v operation으로 묶여있다. buffer에 채워진 칸의 수를 의미하는 items가 만약 0이라면 꺼낼 요소가 없는 것이고, 그렇다면 맨 첫 줄의 p에 의해 suspend 된다. 성공적으로 요소를 꺼내고 나면 빈 칸이 한 칸 늘어나므로 slots에 v를 적용한다. 이런 과정으로 sbuf와 worker thread가 충돌 및 오류 없이 작동할 수 있다.

- Task3 (Performance Evaluation)

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

※ multiclient를 통해 client를 복제하고 요청을 실행하면서, multiclient 프로세스가 시작하고 끝날 때 까지의 시간을 측정하여 이를 기준 시간으로 사용하였다. 이는 sys/times.h 헤더의 함수들을 사용한다.

1) 확장성 : client의 개수 변화에 따른 동시 처리율을 측정하였다. client가 5개, 10개, 20개, 40개로 변함에 따라 동시 처리율이 어떻게 변동하는지 측정하였다. client는 40번의 요청을 한다. 동시 처리율은 요청의 개수가 늘어나면 자연히 커지게 되는데, 세 자리 수가 넘어가는 너무 많은 요청을 하게 될 경우 client의 수가 변동함에 따라 바뀌는 동시 처리율이 눈에 잘 띄지 않기 때문에 40번의 요청을 선택하였다. 또, client의 수가 너무 많이 늘어나면 fork가 수반하는 overhead가 너무 커지기 때문에 40개의 client를 최대값으로 설정하고 네 개의 축으로 나누었다.

2) 워크로드 : client가 어떤 요청을 하는지에 따른 동시 처리율을 측정하였다. 5개, 10개, 20개, 40개의 client가 show 요청만 보낼 경우와 sell/buy 요청만 보낼 경우, 섞어서 보낼 경우 세 가지에 대해 조사하였다. sell과 buy는 구조적으로 완전히 동일한 동작을 수행하므로, 같은 변인으로 묶었다. 또, client의 수가 늘어남에 따라 두 종류 요청의 동시 처리율의 차이가 늘어나는지, 혹은 줄어드는지를 확인하기 위해 client의 수도 변인으로 책정했다.

3) 트리 구현 방법 : 트리가 기존의 BST일 때와 height가 최소인 AVL일 때의 성능 차이를 측정하였다. 측정을 많이, 그리고 간편하게 하기 위해 무작위의 순서를 가지는 2000개의 node를 생성해 BST와 AVL에 넣고, 1부터 2000까지의 검색을 수행하여 걸리는 시간을 체크하였다.

4) readers-writers : thread-based server의 show 요청을 수행할 때, mutex suspend로 인한 시간 지연을 줄이기 위해 readers-writers 방식을 사용하였다. 일반적인 show 요청과 방식을 도입한 show 요청을 비교하기 위해, 40개의 client가 무작위 40개의 요청을 보낸 결과를 측정하였다.

✓ Configuration 변화에 따른 예상 결과 서술

1) Thread-based server는 thread를 사용해 multicore CPU가 여러 작업을 동시에 처리할 수 있지만, Event-based server는 하나의 process에서 client마다 한 번씩 작업을 진행하므로, 확장성 면에서는 Thread-based server가 우수한 모습을 보일 것이다.

2) show 요청은 buy/sell 요청과 달리 strcat, c11_itoa(custom) 등의 overhead가 큰 함수를 사용하고, 트리의 모든 노드를 탐색해야 하기 때문에 동시 처리율이 더 떨어질 것으로 보인다. 또, event-based server에서는 구조 상 하나의 process에서 하나의 connected fd를 조회한 이후 다음 fd를 조회할 수 있지만, thread-based server에서는 값을 조정하지 않는 경우 많은 fd에 대해 동작을 동시에 수행할 수 있다. 이런 thread-based server의 이점은 show 명령어를 수행할 때 더욱 장점이 될 것이기 때문에 thread-based server가 더 효율적으로 작동할 것이다.

3) 평균적으로 많은 노드를 무작위로 넣으면 높은 확률 BST도 AVL과 비슷한 모양을 갖기 때문에, 성능 상 아주 큰 차이는 없지만 AVL이 근소우위를 점할 것이다.

4) readers-writers 방식이 thread가 동시다발적으로 수행되게 해주기 때문에, 어좋은 성능을 보일 것이다. 하지만, client의 수와 요청의 수가 그렇게 많지 않기 때문에 큰 차이는 없을 것으로 보인다.

C. 개발 방법

서버를 구현하기 위해서, 강의 자료의 concurrent echo server 코드를 그대로 사용한다. 그 코드에서 서버가 요청을 수행하는 부분을 내가 구현하고자 하는 요청으로 바꾸면 된다.

먼저 서버의 종류와 무관하게 stock의 정보를 관리하기 위해 AVL Tree를 이용하였다. AVL tree는 BST(Binary Search Tree)의 문제점을 개선한 자료구조이다. BST는 자식 노드가 왼쪽이나 오른쪽에 쏠리게 되면 시간 복잡도 면에서 트리의 이점을 살릴 수 없게 된다. 이런 문제를 없애기 위해, AVL에서는 노드의 삽입과 삭제 시마다 불균형이 발생하면 회전을 통해 자동으로 균형을 맞춘다. 물론 AVL 트리는 노드의 변화가 있을 때마다 상태를 체크해 회전을 진행하기 때문에 그 과정에서 정말 작은 overhead가 발생할 수는 있다. 하지만, 이번 프로젝트에서는 stock 종류의 변화가 없어 노드를 삽입하거나 삭제할 일이 없고, 보통의 서버는 요청에 의해 트리를 탐색하는 연산을 훨씬 많이 하기 때문에 AVL 트리를 사용하는 것이 좋다고 생각했다.

```
typedef struct _Elem {
    int ID, price, height, left_stock;
    struct _Elem *right;
    struct _Elem *left;
} Elem;
```

(Thread-based server의 Elem(node)에는 critical section을 보호하기 위한 mutex, readers-writers 방식을 위한 w도 존재함.)

```
/* AVL functions */
Elem* AVL_insert(Elem* root, int id, int left_stock, int price);
Elem* AVL_search(Elem* node, int id);
int AVL_height(Elem *node);
Elem* AVL_L(Elem *node_2);
Elem* AVL_R(Elem *node_1);
Elem* AVL_LL(Elem *node);
Elem* AVL_RR(Elem *node);
```

위 사진에선 AVL 트리를 관리하는 함수들을 나열하였다. 프로젝트에서 필요한 삽입, 탐색 등의 기능을 제공한다. (코드에 관한 자세한 설명은 본 프로젝트의 문맥에 맞지 않으므로 작성하지 않는다.) 서버 프로세스가 실행되면 가장 먼저 stock_load() 함수를 통해 stock.txt에서 정보를 읽어 AVL 트리를 세운다. 그리고 서버 프로세스가 종료될 땐 stock_save() 함수를 통해 변환된 정보를 stock.txt에 저장한다. show, buy, sell 등의 요청은 AVL 트리를 적절히 탐색하여 값을 확인하거나 변경하면 된다.

다음은 concurrent echo server에서 요청 수행 부분인 echo(connfd)를 수정한 함수들

이 있다.

```
void exec_command(int connfd, char *buf, int n);
command command_check(char *buf, int *id, int *amount);
void show_command(int connfd);
void sell_command(int connfd, int id, int amount);
void buy_command(int connfd, int id, int amount);
void exit_command(int connfd);
```

요청을 수행하기 위해 exec_command를 호출하면, 우선 command_check를 통해 show, sell, buy, exit 중 어떤 요청을 보내왔는지를 확인하여 return한다. 이후 command의 종류에 따라 아래 네 개의 함수 중 하나를 호출해준다. show_command에선 현재 stock의 상태를 모두 적어 client에게 전송한다. sell_command에선 AVL 트리에서 id에 따라 stock을 찾아 amount만큼 개수를 더한다. buy_command에선 AVL 트리에서 id에 따라 stock을 찾아 amount만큼 개수를 빼는데, stock이 amount보다 작은 경우 값을 변경하지 않고 오류 메시지를 전송한다. exit_command에선 connected fd에 "exit"을 적어 client가 연결을 종료할 수 있도록 해준다.

Event-based concurrent server를 구현하려면 finer granularity를 확보해야 한다. 이를 위해서 앞서 설명한 pool을 사용하여 이벤트 루프로 작업을 비동기적으로 처리한다. pool 구조체는 다음과 같이 이루어져 있다.

```
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;
```

Pool은 여러 client의 connected fd를 관리하는 자료구조이다. maxfd는 현재 pool에 있는 fd 중 가장 큰 값을 저장하여 탐색 범위를 한정한다. read_set은 select 함수에 전달되는 fd_set으로, 읽고자 하는 fd를 전달하기 위해 사용한다. ready_set은 read_set의 값을 보존하기 위해 실제로 사용하는 fd_set으로, 작동 전 read_set의 값을 복사한다(select 함수에 실제로는 ready_set이 전달됨). nready는 루프마다 I/O Event가 발생한

fd의 수를 저장한다. maxi는 최대 인덱스를 저장하는 값이다. clientfd는 연결된 fd를 저장하는 배열이다.

```
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);
        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_client(&pool);
}
```

```
void add_client(int connfd, Pool *p) {
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) {
        if (p->clientfd[i] < 0) {
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);
            FD_SET(connfd, &p->read_set);
            if (connfd > p->maxfd)  p->maxfd = connfd;
            if (i > p->maxi)    p->maxi = i;
            break;
        }
    }
}
```

```
void check_client(Pool *p) {
    int n, connfd;
    char buf[MAXLINE];
    rio_t rio;

    for (int i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                printf("server received %d bytes\n", n);
                exec_command(connfd, buf, n);
            }
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
            }
        }
    }
}
```

main의 루프에선 pool의 ready_set과 select를 통해 I/O Event를 감지하고, 그 수를 nready에 저장한다. 만약 listenfd에 이벤트가 발생한다면 accept로 connection을 진행하고, 그 fd를 add_client로 pool에 저장한다. 루프의 마지막에서 check_client로

connected fd에 대한 요청을 처리한다. add_client에선 clientfd에서 비어있는 칸을 찾아 새로운 fd를 넣고, read_set의 값을 1(true)로 바꾼다. pool에 추가하며 clientrio에 소켓의 요청을 읽어 넣어둔다. 만약 새로 들어간 fd가 최댓값이라면 maxfd 값을 바꾸고, 새로 들어간 칸이 최대 인덱스라면 maxi 값을 바꾼다. check_client에선 clientfd를 필요한 만큼 순차 탐색하며 정보를 읽어들인다. connected fd로 설정된 fd에 대해서, rio buffer에 값이 있을 시 읽어들여 요청을 수행한다. 이와 같은 I/O Multiplexing으로 event-based server에서 concurrency를 확보할 수 있다.

Thread-based concurrent server를 위해서는 producer-consumer, readers-writers 방식을 사용하였다. producer(main thread)와 consumer(worker thread)가 공유하는 shared buffer(sbuf_t) 구조체는 다음과 같다.

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

buf는 n의 크기를 갖는 buffer이고, front와 rear은 queuing을 위한 값이다. mutex는 buffer에 쓸 때 발생하는 critical section을 보호하기 위한 semaphore이고, slots와 items는 위에서 설명한(2. B.) 역할을 하는 counting semaphore이다.

```
void sbuf_init(sbuf_t *sbufp, int n);
void sbuf_insert(sbuf_t *sbufp, int elem);
int sbuf_remove(sbuf_t *sbufp);
```

sbuf_init에선 buf에 n 크기의 메모리를 할당하고, semaphore의 값들을 초기화한다. sbuf_insert와 sbuf_remove는 위에서 설명한(2. B.) 것과 같은 역할을 수행한다.

```
for (int i = 0; i < 500; i++)   Pthread_create(&tid, NULL, thread, NULL);   //spawn 500 threads
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);
}
```

```c
void *thread(void *vargp) {
    Pthread_detach(pthread_self());

    while (1) {
        int n, connfd = sbuf_remove(&sbuf);
        char buf[MAXLINE];
        rio_t rio;

        Rio_readinitb(&rio, connfd);
        while ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
            printf("server received %d bytes\n", n);
            exec_command(connfd, buf, n);
        }

        Close(connfd);
    }
}
```

thread가 생성될 때의 overhead를 앞으로 당기기 위해 우선 thread를 500개 미리 생성해두고, sleep 시켜둔다. main thread가 connected fd를 sbuf에 넣으면, sleep 상태인 worker thread가 sbuf에 item이 있다는 것을 인식하여 동작을 실행한다. thread 함수에선 우선 kernel에게 reaping을 요청하는 Pthread_detach 함수를 호출하고, connected fd가 저장된 sbuf에서 하나를 꺼내 요청을 처리한다. 이런 방식으로 thread-based server에서 효율적으로 concurrency를 얻을 수 있다.

```c
void show_command(int connfd) {
    char buf[MAXLINE] = "";

    for (int i = 0; i < show_size; i++) {
        char s1[100], s2[100], s3[100];

        P(&(show_elem[i]->mutex));
        (show_elem[i]->readcnt)++;
        if (show_elem[i]->readcnt == 1) P(&(show_elem[i]->w));
        V(&(show_elem[i]->mutex));

        c11_itoa(show_elem[i]->ID, s1, 10);
        c11_itoa(show_elem[i]->left_stock, s2, 10);
        c11_itoa(show_elem[i]->price, s3, 10);

        strcat(buf, s1);
        strcat(buf, " ");
        strcat(buf, s2);
        strcat(buf, " ");
        strcat(buf, s3);
        strcat(buf, "\n");

        P(&(show_elem[i]->mutex));
        (show_elem[i]->readcnt)--;
        if (show_elem[i]->readcnt == 0) V(&(show_elem[i]->w));
        V(&(show_elem[i]->mutex));
    }

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

```c
void sell_command(int connfd, int id, int amount) {
    Elem *temp = AVL_search(root, id);

    P(&(temp->w));
    temp->left_stock += amount;
    V(&(temp->w));

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

thread를 사용할 경우 모든 동작에 대해 mutex가 동기적 작동을 막기 때문에 효율이 떨어질 수 있다. 이런 문제를 최소화하기 위해서 readers-writers 방식을 사용할 수 있다. show 요청은 read만 하고, buy/sell 요청은 write를 하기 때문에 적절히 동작 순서를 바꾸면 효율을 최적화할 수 있다. w semaphore의 관점에서 read와 write의 flow를 확인해보면, 만약 read 요청이 와서 readcnt가 1이 되면 p에 의해 write 요청이 막히고, 현재 queue에 있는 read 요청들만 동작할 수 있도록 된다. 이후 read 요청이 모두 수행되어 readcnt가 0이 되면 v에 의해 다시 모든 요청이 동작에 진입할 수 있다. 동시에 수행할 수 있는 read 연산을 우선적으로 모두 처리해 병렬화를 하고, 효율을 증대할 수 있다. (Read-Priority)

3. 구현 결과

이번 프로젝트에서 요구하는 두 가지 타입의 concurrent server는 모두 구현하였다. multiclient.c를 통해 concurrency issue 없이 많은 client의 요청을 모두 수행할 수 있다는 것을 확인하였다. 또, thread-based server의 요청 처리 과정에서 read-priority를 도입해 효율성을 증가시켰다. 모든 구현 사항 하에 두 서버가 오류 없이 클라이언트의 요청을 concurrent하게 처리한다.

4. 성능 평가 결과 (Task 3)

※ 기존 multiclient.c에는 요청 간 구별을 위해 1초를 sleep한다. 하지만 중첩된 1초 대기가 요청 실행 시간에 비해 상당히 dominant한 요소가 될 것으로 보아 성능 평가 중에는 그 라인을 삭제하였다. elapsed time을 측정하는 부분은 sys/times.h 헤더의 gettimeofday 함수를 사용하였다. multiclient.c의 시작 부분에서 start 시간을 측정하고, parent process가 wait을 모두 마친 마지막 부

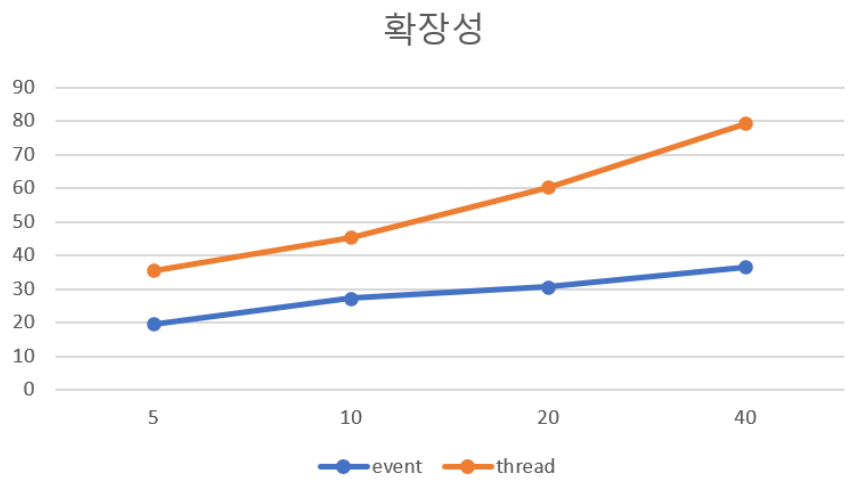
분에 end 시간을 측정하였다.

```
    struct timeval start, end;
    gettimeofday(&start,NULL);
/*  fork for each client process    */
    while(runprocess < num_client){
        //wait(&state);
        pids[runprocess] = fork();

        if(pids[runprocess] < 0)
            return -1;
        /*  child process       */
        else if(pids[runprocess] == 0){
                Fputs(buf, stdout);

                //usleep(1000000);
            }

            Close(clientfd);
            exit(0);
        }
        runprocess++;
    }
    for(i=0;i<num_client;i++){
        waitpid(pids[i], &status, 0);
    }
    gettimeofday(&end,NULL);
    unsigned long e_usec = ((end.tv_sec * 1000000) + end.tv_usec) - ((start.tv_sec * 1000000) + start.tv_usec);
    printf("elapsed time : %lu microseconds.\n",e_usec);

    return 0;
}
```

해당 코드로 multiclient를 실행하면 다음과 같은 출력이 나온다. 5개의 client가 40개의 명령을 thread-based server에서 실행했을 때의 출력이다.

```
4 35 5600
2 253 28000
[buy] success
elapsed time : 5312 microseconds.
```

1) 확장성



확장성 측정의 결과는 위와 같다. x축은 client의 수이고, y축은 동시 처리율(= number of clients * order per client * 1000 / elapsed time)이다. 결과는 두 가지로 해석

할 수 있다.

(1) event-based server보다 thread-based server가 훨씬 높은 동시 처리율을 가진다. 따라서, thread-based server가 많은 client의 요청을 동시에 더 빠르게 처리할 수 있다.

(2) client의 수가 늘어남에 따라 동시 처리율이 늘어나는 비율이 event-based server보다 thread-based server가 더 높다. 따라서, client의 수가 많으면 많을수록 상대적인 thread-based server의 효율이 더 좋아진다.

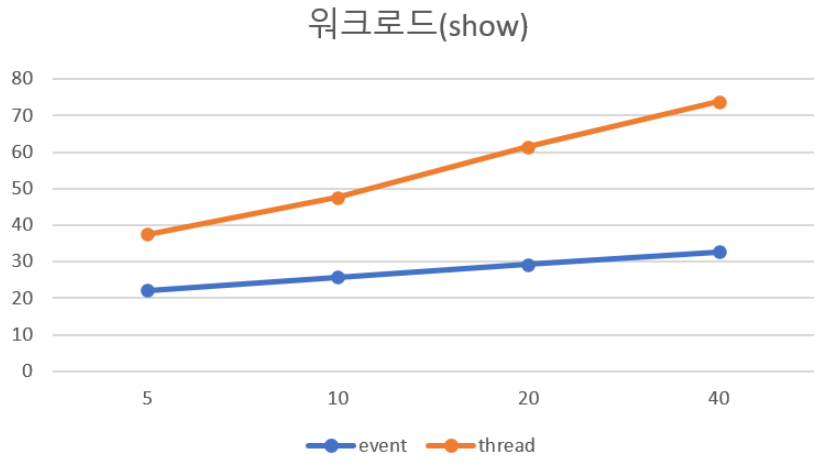
이런 결과가 나오게 된 이유는 역시 thread-based server의 동기성, 병렬성을 꼽을 수 있다. thread-based server는 동시에 들어오는 많은 client의 요청을 동기적으로 처리할 수 있지만, event-based server는 비동기적으로 처리하기 때문에 동시처리율이 조금 떨어지고, 확장성도 떨어지는 것이다.

2) 워크로드

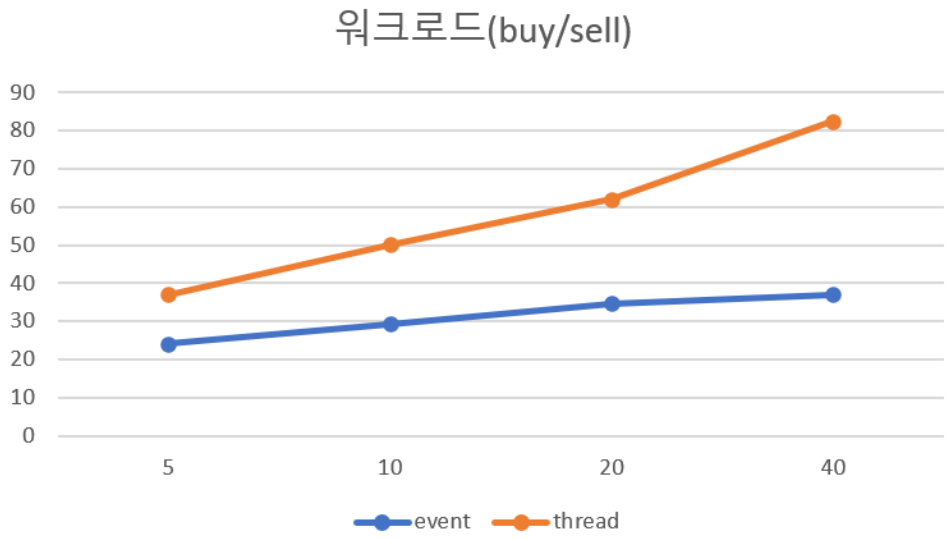
워크로드 측정을 위해선 multiclient.c를 조금 변형해야 한다. 원래는 난수를 이용해 요청의 종류를 결정하는데, 이 실험에선 한 종류의 명령만을 수행하도록 바꿔야 한다.

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

이를 위해 option을 정하는 라인을 수정했다. show의 경우 option의 rvalue를 1~2 중 무작위로 고르게 한 후 이후 branch를 모두 show로 바꾸었고, buy/sell의 경우 1~2 중 무작위로 고르게 하여 1에는 buy, 2에는 sell을 요청하도록 하였다. 또, show 명령어를 buffer에 넣는 과정에서 s, h, o, w를 strcat하여 show를 만들었다. 이는 strcat, rand 함수를 호출할 때나 if~else branch를 실행할 때 발생하는 overhead를 비슷하게 통일하기 위한 것이다. 측정 결과는 다음과 같다.



show 요청을 수행할 때 thread-based server가 event-based server보다 훨씬 높은 동시 처리율을 보인다. 이는 (1)에서 확장성 차이의 원인으로 꼽은 것과 같은 원인을 가지는 것으로 보인다.



buy/sell 요청을 수행할 때 thread-based server가 event-based server보다 훨씬 높은 동시 처리율을 보인다. 이 역시 (1)에서 확장성 차이의 원인으로 꼽은 것과 같은 원인을 가지는 것으로 보인다. 명령어가 혼합된 경우의 워크로드 분석은 (1) 확장성에서 측정한 데이터를 그대로 사용하면 된다. 역시 워크로드에 따라 큰 차이는 보이지 않지만, show와 buy/sell의 두 경우에서 근소한 차이를 확인할 수 있다.

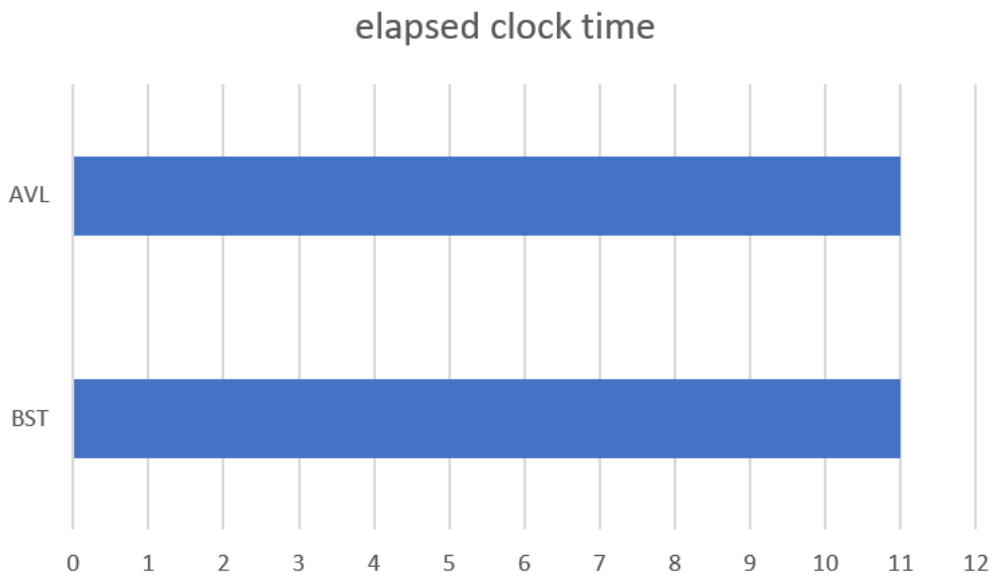
show와 buy/sell의 측정 결과를 비교해보자. show의 동시 처리율보다 buy/sell의 동시 처리율이 약간 더 높은 것을 확인할 수 있다. 이는 예상한대로 show 요청의 overhead가 buy/sell 요청보다 더 크기 때문이다. 트리를 전부 탐색해야 하고, c11_itoa, strcat 등의 함수를 사용하기 때문에 이런 결과가 나온 것으로 보인다.

3) 트리 구현 방법

트리 구현 방법에 따른 처리 속도를 체크하는 과정은 server와 client 사이의 통신에서 구현하지 않고, 임의의 트리를 만들어 전 과정을 처리하는데 걸리는 시간을 측정하였다. 개략적인 코드는 다음과 같다.

```
int* array=(int*)calloc(500,sizeof(int));
random_array(array);

clock_t start1=clock();
for(int i=0;i<2000;i++)
    root1=BST_insert(root1,array,i);
BST_search_all();
clock_t end1=clock();
printf("BST time : %lf\n",(double)(end1-start1));

clock_t start2=clock();
for(int i=0;i<2000;i++)
    root2=AVL_insert(root2,array,i);
AVL_search_all();
clock_t end2=clock();
printf("AVL time : %lf\n",(double)(end2-start2));
```

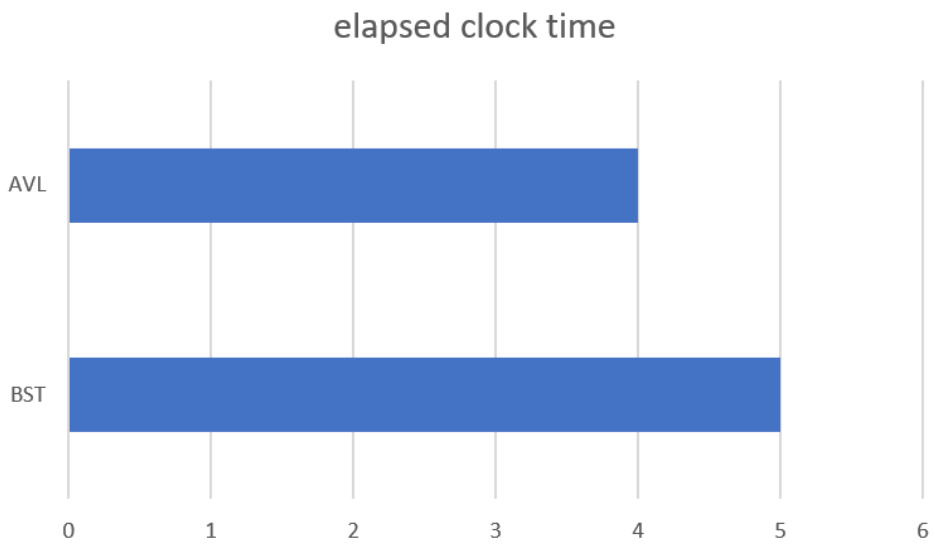
1부터 2000까지의 숫자를 무작위로 array에 배열하고, 이를 각각의 트리에 insert한 후, 다시 1부터 2000까지 search를 진행하는 방식을 채택하였다. 결과는 다음과 같다.



둘의 clock time이 11으로 완전히 동일하게 측정되었다. AVL이 아주 조금 더 빠를 거라고 예상했는데, 그렇지 않은 결과가 나왔다. 원인으로 예상되는 점은, AVL은 insert를 진행하면서 계속해서 균형을 맞추는데, 그 과정에서 BST보다 큰 overhead가 발생한 것이다. insert에서 조금 더 걸리고, search에서 조금 덜 걸려서 같은 시간이 걸린 것으로 보인다. 프로젝트에서 구현한 stock server의 경우 insert를 통한 트리 생성은 프로세스 실행시에 모두 마치고, search를 훨씬 많이 하기 때문에 코드를 조금 수정해보았다.

```
int* array=(int*)calloc(500,sizeof(int));
random_array(array);

for(int i=0;i<2000;i++)
    root1=BST_insert(root1,array,i);
clock_t start1=clock();
BST_search_all();
clock_t end1=clock();
printf("BST time : %lf\n",(double)(end1-start1));

for(int i=0;i<2000;i++)
    root2=AVL_insert(root2,array,i);
clock_t start2=clock();
AVL_search_all();
clock_t end2=clock();
printf("AVL time : %lf\n",(double)(end2-start2));
```

이렇게 하면 search에 걸리는 시간만 확인할 수 있다. 결과는 다음과 같다.



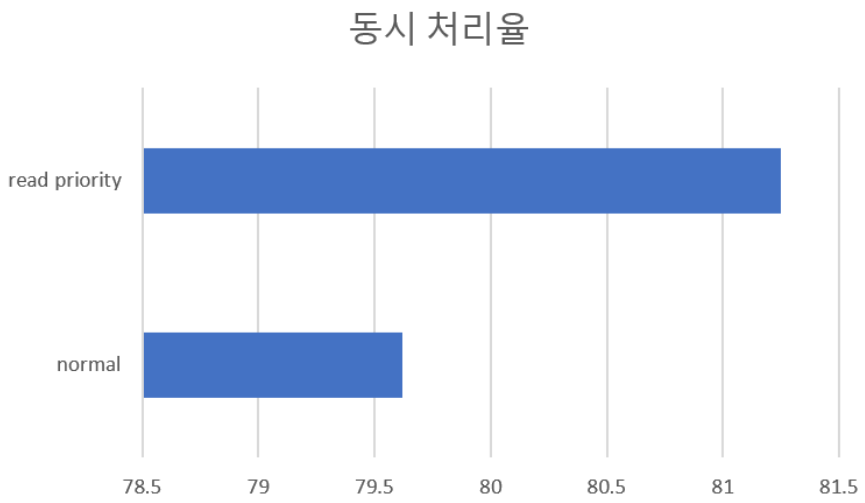
BST보다 AVL이 아주 약간 빠른 속도를 가지는 모습을 보인다. BST보다 AVL이 height가 약간 작기 때문에 이런 결과가 나온 것으로 보인다. 따라서, search 연산을 훨씬 많이 하는 stock server의 경우 AVL Tree를 사용하는 것이 탁월한 선택임을 알 수 있다. 하지만, 해당 프로젝트가 아닌 실제 stock server에서는 insert, delete 등의 연산을 많이 하게 될 수 있으므로, 상황에 맞는 자료구조를 선택해야 한다.

4) readers-writers (read priority)

```c
void show_command(int connfd) {
    char buf[MAXLINE] = "";

    for (int i = 0; i < show_size; i++) {
        char s1[100], s2[100], s3[100];

        P(&(show_elem[i]->mutex));
        (show_elem[i]->readcnt)++;
        if (show_elem[i]->readcnt == 1) P(&(show_elem[i]->w));
        V(&(show_elem[i]->mutex));

        c11_itoa(show_elem[i]->ID, s1, 10);
        c11_itoa(show_elem[i]->left_stock, s2, 10);
        c11_itoa(show_elem[i]->price, s3, 10);

        strcat(buf, s1);
        strcat(buf, " ");
        strcat(buf, s2);
        strcat(buf, " ");
        strcat(buf, s3);
        strcat(buf, "\n");

        P(&(show_elem[i]->mutex));
        (show_elem[i]->readcnt)--;
        if (show_elem[i]->readcnt == 0) V(&(show_elem[i]->w));
        V(&(show_elem[i]->mutex));
    }

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

```c
void show_command(int connfd) {
    char buf[MAXLINE] = "";

    for (int i = 0; i < show_size; i++) {
        char s1[100], s2[100], s3[100];

        P(&(show_elem[i]->w));
        c11_itoa(show_elem[i]->ID, s1, 10);
        c11_itoa(show_elem[i]->left_stock, s2, 10);
        c11_itoa(show_elem[i]->price, s3, 10);
        V(&(show_elem[i]->w));

        strcat(buf, s1);
        strcat(buf, " ");
        strcat(buf, s2);
        strcat(buf, " ");
        strcat(buf, s3);
        strcat(buf, "\n");
    }
    Rio_writen(connfd, buf, MAXLINE);
}
```

첫 번째 사진은 read-priority 방식이 사용된 코드이고, 두 번째 사진은 그렇지 않은 사진이다. 두 코드의 동시 처리율 차이를 측정해보았다.



read-priority를 도입한 코드가 아주 근소하게 높은 동시 처리율을 보인다. read 연산을 동시에 처리할 수 있기 때문에 이런 차이가 생긴 것으로 보인다.