System Programming Project 3

담당 교수 : 박성용 교수님

이름 : 윤성민

학번 : 20191264

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

-이번 Stock-server 프로젝트의 목표는 한 명의 client뿐만 아니라 여러 명의 client들이 동시 접속할 수 있고 서비스를 받을 수 있는 concurrent한 주식 서버를 구현하는 것이다. 각 client들이 주식 서버에 show, buy, sell 등의 요청함에 따라 주식 서버가 저장하고 있는 binary tree로 구현된 주식 정보 list를 보여주고 구매와 판매에 대한 동작을 수행하도록 한다. 이렇게 여러 client가 동시에 접속하여 서비스를 받을 수 있는 concurrent한 주식 서버를 event-driven과 thread-based의 2가지 방식으로 접근하여 구현하고 이에 대한 성능을 분석하고 평가한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

-Event-driven 방식의 서버는 select 함수를 통해 하나의 프로세스에서 여러 client의 요청을 동시에 관리 및 처리하도록 구현한다. Pool 구조체를 사용하여 client의 정보를 저장하여 관리하고 select함수를 통해 client의 입력을 모니터링하여 client의 요청이 입력되면 server측에서 check_client함수를 통해 각 요청에 대한 동작을 수행하도록 한다. 이러한 event-driven 방식의 서버는 하나의 루프를 통해 event를 계속해서 감시하므로 client의 요청(event)이 들어올 때 그에 맞는 처리를 해줌으로써 server가 동시에 접속한 여러 client들의 요청을 처리할 수 있게 한다.

2. Task 2: Thread-based Approach

-Thread-based 방식의 서버는 event-driven 방식과 다르게 여러 개의 thread를 이용해 concurrency를 구현한다. Master thread는 client의 connection을 accept하여 Sbuf_t 구조체(buffer)에 connfd를 넣어주고 미리 생성한 worker thread들이 buffer에서 connfd를 받아 그에 맞는 client의 요청을 처리하도록 한다. 이를 통해 여러 client의 요청을 동시에 처리할 수 있다. Thread-based 방식의 경우 여러 thread가 동시에 주식 정보에 접근하기 때문에 semaphore를 사용하여 reader-writer problem을 해결해 준다.

3. Task 3: Performance Evaluation

-multiclient.c 내의 client 개수를 변경하거나 client의 요청 타입을 변경하는 등 client 실행 파일 내의 configuration들을 바꿔가면서 gettimeofday함수를 이용해 두 가지(event-driven, thread-base) 방식의 elapse time을 측정하여 성능을 분석한다.

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 통해 I/O Multiplexing을 처리하게 된다. server는 select함수를 호출함으로써 pending input(event)가 발생한 파일 디스크립터가 있는지 감시한다. event가 발생하게 되면 server는 accept 함수를 통해 connection fd를 return하고 해당 connfd를 add_client함수를 통해 pool에 추가한다. 그 이후에 check_client함수를 호출하여 pool에 저장된 각 connfd에 대해 그에 맞는 client의 요청을 읽어 맞는 동작을 수행한다.

✓ epoll과의 차이점 서술

-Select와 epoll 모두 I/O Multiplexing을 처리하는 방식이다. 이번 프로젝트에서 사용하는 select는 감시할 수 있는 fd의 최대 개수가 제한되어 있다. 일반적으로 최대 1024개의 fd를 감시할 수 있고 이를 넘으면 더 이상 감시할 수 없다. 또한 select는 event를 감지하기 위해 모든 fd를 루프를 통해 순회하면서 선형적으로 감시한다. 따라서 fd의 수가 많아질수록 성능이 떨어지는 단점이 있다. 반면에 epoll은 감시할 수 있는 fd의 개수에 제한이 없고 event가 발생한 fd만 반환하기 때문에 fd의 수가 많아져도 성능 저하가 없다. 따라서 epoll은 대규모 fd set을 감시할 때 높은 성능을 제공한다.

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

✓ Master Thread의 Connection 관리

-Thread-based 방식의 server는 여러 worker thread들을 미리 생성해두고 client들과의 connection을 관리하는 Master thread가 accept한 connfd를 buffer에 넣어주면 미리 생성해둔 worker thread에서 buffer의 connfd를 가져가 그에 맞는 동작을 수행하는 pre-threaded concurrent server로 구현한다. 따라서 각 client의 connection마다 별도의 worker thread가 적용되므로 동시에 여러 client의 요청을 관리 및 처리할 수 있게 된다. client가 연결될 때마다 thread를 생성하여 요청을 처리하게 되면 thread의 overhead는 줄일 수 있지만 시간적 측면에서 delay가 발생하게 된다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

-Worker Thread Pool은 앞서 언급한 worker thread들의 집합이라 할 수 있다. 여러 worker thread들은 미리 생성되어 Master thread에 의해 이루어진 connection을 buffer를 통해 받아 요청을 수행하기 때문에 동시에 여러 client의 요청을 처리할 수 있다. 이러한 여러 개의 worker thread들이 client의 요청을 동시에 처리하는 multi-thread 환경에서는 data race문제가 발생할 수 있다. 어느 한 thread가 show의 요청을 처리하는 도중에 다른 thread가 buy의 요청을 처리하면서 주식 정보에 변화가 생긴다면 show의 요청을 처리하는 thread는 잘못된 정보를 읽을 수도 있다. 따라서 각 thread가 show, buy, sell 등의 요청을 처리하는 과정에서 정보를 읽거나 쓸 때 잘못된 정보의 도출을 막기 위해 P, V 함수를 통해 Read와 Write가 동시에 일어나는 일을 방지한다. 이번 프로젝트에서는 first reader writer problem을 적용하여 read를 write보다 우선시하여 실행하도록 구현한다.

Task3 (Performance Evaluation)

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

-Task3에서는 client 실행 파일 내의 configuration을 바꿔가면서 event-driven과 thread-based 방식의 동시처리율을 분석한다. 여기서 우리가 얻고자 하는 metric은 동시 처리율로 시간당 몇 개의 client의 요청을 처리할 수 있는지를 나타내는 측정값이며 "동시처리율 = (# of client)*(ORDER_PER_CLIENT) / (elapsed time)"으로 나타낼 수 있다. 동시 처리율을 통해 2가지 방식의 서버에서의 시간(초)당 처리 가능한 client의 요청 수를 구해 각 방식의 성능을 측정할 수 있다.

1. 확장성 : Client의 개수 변화에 따른 동시 처리율 변화 분석

-각 방식에서 client의 개수를 변화 시키면서 elapsed time을 구해보고 동시 처리율을 구해 성능을 분석해본다. 각 client의 요청 개수인 ORDER_PER_CLIENT의 값은 10으로 고정한 후 client의 수를 1, 5, 10, 20, 50, 100, 150, 200, 300, 500로 바꿔가면서 비교 분석한다. 해당 분석에서는 Thread-based 서버에서 NTHREAD의 수를 100으로 하여 미리 worker thread를 많이 생성해두는 것으로 초기 설정을 한다.

2. 워크로드에 따른 분석 : client 요청 타입(buy, show, sell)에 따른 동시처리율 분석

-Client의 요청 타입에 변화를 주면서 동시처리율을 분석한다. Client가 buy, show, sell 모두 요청하는 경우(기본), show만 요청하는 경우, buy와 sell만 요청하는 경우를 2가지 방식의 서버에서 측정하여 비교 분석한다. 해당 분석에서는 Client의 개수를 100개로, ORDER_PER_CLIENT는 10으로 고정하고 평균적인 elapsed time과 동시처리율을 확인하기 위해 5번씩 실행한다.

3. 기타 분석 : Thread-based server에서의 thread 수에 따른 동시처리율 분석

-Task1에서 구현하는 Event-driven 방식은 단일 프로세스에서 여러 client를 동시에 처리하는 반면 Task2에서의 Thread-based 방식은 여러 thread들이 동시에 client들을 처리한다. 따라서 reader-writer problem을 방지한 thread-based 방식의 server에서 NTHREAD의 수를 늘리면서 thread 수가 많아질수록 동시처리율이 어떻게 변하는지 분석한다.

각 분석은 gettimeofday 함수를 통해 elapsed time을 구해 동시처리율을 계산하여 진행한다.

✓ Configuration 변화에 따른 예상 결과 서술

-확장성 분석에서는 2가지 방식의 server 모두 client의 개수를 늘릴수록 elapsed time은 늘어날 것이지만 event-driven 방식에 비해 thread-based 방식이 elapsed time이 적게 나올 것으로 예상된다. 하지만 thread-based 서버에서 미리 생성해둔 worker thread의 개수보다 많은 처리량이 들어오게 되면 오버헤드가 발생하기 때문에 어느 시점에서부턴 thread-based 방식에서 event-driven 방식보다 높은 elapsed time 값이 측정될 것으로 예상된다.

-워크로드에 따른 분석에서는 binary search로 모든 주식 정보를 순회하여 탐색해야 하는 show의 요청이 buy와 sell의 요청에 비해 시간이 더 오래 걸릴 것으로 예상된다.

-Thread-based 방식의 서버에서 thread의 수를 늘릴수록 동시 처리율이 높아질 것으로 예상된다.

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-based Server 구현

① pool 구조체와 stock 구조체 구현

```
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;

typedef struct stock{
    int ID;
    int left_stock;
    int price;
    //int readcnt;
    struct stock* left;
    struct stock* right;
}Stock;
Stock *root = NULL;
```

-connected file descriptor를 저장하기 위한 구조체 pool을 선언해주고 binary tree로 구현할 주식 정보를 구조체 Stock으로 선언해준다. 두 구조체 모두 전역 변수로 선언해준다. 또한 주식 정보 binary tree의 root 노드를 NULL값으로 하여 전역변수로 선언해준다.

② init_pool 함수와 add_client 함수 구현

-pool을 초기화해주는 init_pool과 새로운 connected file descriptor를 fd_set에 추가해주는 add_client를 구현하여 listenfd가 준비되면 add_client 함수를 통해 connfd를 pool에 추가해주도록 구현한다. (강의자료에 있는 코드 사용하였습니다)

③ check_clients 함수 구현

```
void check_clients(pool *p) {
    int i, connfd, n;
    char buf[MAXLINE];
    char cmd[10];
    rio_t rio;
    for (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))) {
            p->nready--;
            if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
                int ID, cnt;
                char stock_info[MAXLINE];
                printf("server received %d bytes\n", n);
                sscanf(buf, "%s %d %d", cmd, &ID, &cnt);
                if (!strcmp(cmd, "show")) {
                    stock_info[0] = '\0';
                    showstock(root, stock_info, connfd);
                    stock_info[strlen(stock_info)] = '\0';
                    Rio_writen(connfd, stock_info, MAXLINE);
                }
                else if (!strcmp(cmd, "exit")) {
                    Rio_writen(connfd, buf, strlen(buf));
                }
                else if (!strcmp(cmd, "buy")) {
                    buystock(ID, cnt, connfd);
                }
                else if (!strcmp(cmd, "sell")) {
                    sellstock(ID, cnt, connfd);
                }
            }
            else {
                Close(connfd);
                FD_CLR(connfd, &p->read_set);
                p->clientfd[i] = -1;
                client_cnt--;
                //printf("%d\n",client_cnt);
            }
        }
    }
}
```

-add_client로 connfd를 pool에 추가한 뒤 check_client를 통해 pool에 기록된 event, 즉 client의 요청을 받아 처리해준다. strcmp함수를 이용해 client의 요청이 어떤 타입인지 구분하고 그에 맞는 동작을 수행한다.

④ Stock 관련 함수들

```c
Stock* insertstock(Stock *node, int ID, int left_stock, int price) {
    if (node == NULL) {
        Stock *newnode = (Stock*)malloc(sizeof(Stock));
        newnode->ID = ID;
        newnode->left_stock = left_stock;
        newnode->price = price;
        newnode->left = NULL; newnode->right = NULL;
        return newnode;
    }
    if (ID < node->ID)
        node->left = insertstock(node->left, ID, left_stock, price);
    else if (ID > node->ID)
        node->right = insertstock(node->right, ID, left_stock, price);
    return node;
}
```

-insertstock 함수는 binary tree에 주식 정보를 가진 노드를 넣어준다.

```c
void readstocks() {
    FILE *fp = fopen("stock.txt", "r");
    if (fp == NULL) {
        perror("Error : open file\n");
        exit(0);
    }
    int ID, left_stock, price;
    while (fscanf(fp, "%d %d %d", &ID, &left_stock, &price) != EOF) {
        root = insertstock(root, ID, left_stock, price);
    }
    fclose(fp);
}

void writestocks(Stock *node, FILE *wfp) {
    if (node == NULL) return ;
    writestocks(node->left, wfp);
    fprintf(wfp, "%d %d %d\n", node->ID, node->left_stock, node->price);
    writestocks(node->right, wfp);
}
```

-readstocks함수는 stock.txt에서 주식 정보를 읽어오는 함수이고 writestocks는 update된 주식정보를 stock.txt에 updata된 정보로 다시 적어주는 함수이다.

```c
void freestocks(Stock *node) {
    if (node == NULL) return;
    freestocks(node->left);
    free(node);
    freestocks(node->right);
}

void signal_handler(int sign) {
    freestocks(root);
    exit(0);
}
```

-freestocks 함수는 주식 정보를 저장하기 위해 동적 할당했던 binary tree를 free해주는 함수이며 이 함수는 server의 종료를 위해 SIGINT가 입력되면 signal_handler가 호출되면서 실행하게 된다.

⑤ Client의 요청 타입 관련 함수들

```c
void showstock(Stock *node, char* stock_info, int connfd) {
    if (node == NULL) return;
    char buffer[MAXLINE];
    showstock(node->left, stock_info, connfd);
    sprintf(buffer, "%d %d %d\n", node->ID, node->left_stock, node->price);
    strcat(stock_info, buffer);
    showstock(node->right, stock_info, connfd);
}
```

-showstock 함수는 show의 요청을 처리하기 위한 함수로 recursive하게 구현되어 주식 정보 binary tree 전체를 탐색하여 문자열에 저장할 수 있도록 구현하였다.

```c
Stock* searchstock(Stock *node, int ID) {
    if (node == NULL || node->ID == ID)
        return node;
    if (ID < node->ID)
        return searchstock(node->left, ID);
    return searchstock(node->right, ID);
}
```

-searchstock 함수는 아래에 나오는 buystock과 sellstock 함수에서 사용되는 함수로 parameter로 받은 ID 값을 가진 주식 정보를 찾기 위한 함수이다.

```c
void buystock(int ID, int cnt, int connfd) {
    Stock *stock = searchstock(root, ID);
    char buffer[MAXLINE];
    if (!stock) return;
    if (stock->left_stock < cnt) {
        strcpy(buffer, "Not enough left stocks\n");
        Rio_writen(connfd, buffer, MAXLINE);
    }
    else {
        stock->left_stock -= cnt;
        strcpy(buffer, "[buy] success\n");
        Rio_writen(connfd, buffer, MAXLINE);
    }
}

void sellstock(int ID, int cnt, int connfd) {
    Stock *stock = searchstock(root, ID);
    char buffer[MAXLINE];
    if (!stock) return;
    stock->left_stock += cnt;
    strcpy(buffer, "[sell] success\n");
    Rio_writen(connfd, buffer, MAXLINE);
}
```

-buystock 함수와 sellstock 함수는 searchstock 함수를 호출하여 사거나 팔기를 원하는 ID의 주식 정보를 갖는 노드를 받아 그에 맞는 동작을 수행한다. buystock에서는 해당 주식의 남은 개수를 사고 싶은 개수만큼 빼고 sellstock에서는 주식의 남은 개수에 팔고 싶은 개수를 더해준다. 그 이후 해당 동작이 성공적으로 진행되면 "[buy] success" 또는 "[sell] success"를 출력해주고 buy의 경우 사고 싶은 주식의 개수가 부족하면 "Not enough left stock"을 출력해준다

2. Thread-based Server 구현

① SBUFSIZE와 NTHREAD 수 정의

```c
#define SBUFSIZE 1000
#define NTHREADS 100
```

② sbuf_t 구조체와 Stock 구조체 선언

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

typedef struct stock{
    int ID;
    int left_stock;
    int price;
    int readcnt;
    struct stock* left;
    struct stock* right;
    sem_t mutex, w;
}Stock;
Stock *root = NULL;
```

-sbuf_t 구조체를 선언하여 Master Thread와 worker thread가 공유하는 shared buffer를 구현해주고 Stock 구조체에는 reader-writer problem을 해결해 주기 위한 int readcnt, sem_t mutex와 sem_t w를 추가해 주었다.

shared buffer를 sbuf_t sbuf로 하여 전역 변수로 선언해 sbuf_init 함수를 통해 sbuf에 메모리를 할당해주고 sbuf_insert 함수를 통해 sbuf 맨 뒤에 새로운 item을 삽입한다. Thread-based 방식에서는 worker thread가 shared buffer에 있는 item을 가져가 client의 요청을 처리하므로 shared buffer에서 item을 꺼내 주기 위한 함수 sbuf_remove를 호출하여 sbuf 맨 앞의 item을 빼서 worker thread에게 부여해준다.

③ Stock 관련 함수들

-insertstock, readstock, writestock, freestock, searchstock 함수들은 Event-driven server와 동일하게 구현하였고 insertstock에서만 새로운 노드를 초기화해줄 때 mutex와 w에 해당하는 semaphore의 값을 1로 초기화해주는 과정을 추가하였다.

④ check_client 함수 구현 : Client의 요청 처리

```c
void check_client(int connfd) {
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;
    char cmd[10];

    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        int ID, cnt;
        char stock_info[MAXLINE];
        P(&mutex);
        printf("server received %d bytes\n", n);
        V(&mutex);
        //printf("1\n"); //확인용
        sscanf(buf, "%s %d %d", cmd, &ID, &cnt);
        if (!strcmp(cmd, "show")) {
            stock_info[0] = '\0';
            showstock(root, stock_info, connfd);
            stock_info[strlen(stock_info)] = '\0';
            Rio_writen(connfd, stock_info, MAXLINE);
        }
        else if (!strcmp(cmd, "exit")) {
            Rio_writen(connfd, buf, strlen(buf));
            break;
        }
        else if (!strcmp(cmd, "buy")) {
            buystock(ID, cnt, connfd);
        }
        else if (!strcmp(cmd, "sell")) {
            sellstock(ID, cnt, connfd);
        }
    }
    FILE *wfp = fopen("stock.txt", "w");
    if (wfp == NULL) {
        perror("Error : open file\n");
        exit(0);
    }
    writestocks(root, wfp);
    fclose(wfp);
}
```

⑤ Client 요청 타입 함수들

```
void showstock(Stock *node, char* stock_info, int connfd) {
    if (node == NULL) return;
    char buffer[MAXLINE];
    showstock(node->left, stock_info, connfd);
    P(&node->mutex);
    node->readcnt++;
    if (node->readcnt == 1)
        P(&node->w);
    V(&node->mutex);

    //P(&node->w);
    sprintf(buffer, "%d %d %d\n", node->ID, node->left_stock, node->price);
    strcat(stock_info, buffer);
    //V(&node->w);

    P(&node->mutex);
    node->readcnt--;
    if (node->readcnt == 0)
        V(&node->w);
    V(&node->mutex);
    showstock(node->right, stock_info, connfd);
}
```

```
void buystock(int ID, int cnt, int connfd) {
    Stock *stock = searchstock(root, ID);
    char buffer[MAXLINE];
    if (!stock) return;
    if (stock->left_stock < cnt) {
        strcpy(buffer, "Not enough left stocks\n");
        Rio_writen(connfd, buffer, MAXLINE);
    }
    else {
        P(&stock->w);
        stock->left_stock -= cnt;
        V(&stock->w);
        strcpy(buffer, "[buy] success\n");
        Rio_writen(connfd, buffer, MAXLINE);
    }
}
```

```
void sellstock(int ID, int cnt, int connfd) {
    Stock *stock = searchstock(root, ID);
    char buffer[MAXLINE];
    if (!stock) return;
    P(&stock->w);
    stock->left_stock += cnt;
    V(&stock->w);
    strcpy(buffer, "[sell] success\n");
    Rio_writen(connfd, buffer, MAXLINE);
}
```

-Reader-writer problem을 해결하기 위해 Event-driven server에서의 showstock, buystock, sellstock 함수에서 semaphore를 사용하여 P함수와 V함수를 추가해 주었다. show의 요

청을 처리하는 thread는 read의 기능만을 하기 때문에 read 도중에 다른 thread가 buy, sell을 통해 주식 정보를 바꿔 버리면 data race가 발생하게 되므로 구조체 Stock안에 선언해준 semaphore mutex, w 그리고 read 횟수를 세는 readcnt를 사용해 여러 thread가 read하는 것은 가능하지만 read 도중에는 주식 정보를 update하지 못하게 구현하였다. buystock과 sellstock에서는 남은 주식의 개수가 변하기 때문에 남은 주식 개수를 update하는 동안 다른 thread의 접근을 막아야 하므로 semaphore w를 사용해서 해결하였다.

3. 구현 결과

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

1. Event-Based Server 결과

-stockserver 결과

```
cse20191264@cspro:~/sp/prj3/20191264/task_1$ ./stockserver 60007
Connected to (172.30.10.9 39070)
Connected to (172.30.10.9 39056)
server received 9 bytes
server received 5 bytes
server received 8 bytes
server received 5 bytes
server received 5 bytes
server received 5 bytes
server received 8 bytes
server received 5 bytes
server received 5 bytes
server received 8 bytes
server received 9 bytes
server received 9 bytes
server received 5 bytes
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 9 bytes
server received 8 bytes
server received 9 bytes
server received 9 bytes
^Ccse20191264@cspro:~/sp/prj3/20191264/task_1$
```

-multiclient 결과

```
cse20191264@cspro9:~/sp/prj3/20191264/task_1$ ./multiclient 172.30.10.11 60007 2
child 3840348
child 3840349
1 987 10000
2 1914 2000
3 2300 3000
4 2183 4000
5 2938 5000
6 39293 6000
7 3019 7000
8 4504 8000
9 2721 9000
10 5085 5100
1 987 10000
2 1914 2000
3 2300 3000
4 2183 4000
5 2938 5000
6 39293 6000
7 3019 7000
8 4504 8000
9 2721 9000
10 5085 5100
[buy] success
[sell] success
[sell] success
[buy] success
[sell] success
[sell] success
[sell] success
1 995 10000
2 1914 2000
3 2301 3000
4 2183 4000
5 2946 5000
6 39293 6000
7 3019 7000
8 4504 8000
9 2721 9000
10 5067 5100
```

(실제 결과는 더 나옵니다)

2. Thread-based Server 결과

-stockserver 결과

```
cse20191264@cspro:~/sp/prj3/20191264/task_2$ ./stockserver 60007
Connected to (172.30.10.9, 40376)
server received 8 bytes
Connected to (172.30.10.9, 40382)
server received 5 bytes
server received 9 bytes
server received 8 bytes
server received 8 bytes
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 5 bytes
server received 5 bytes
server received 8 bytes
server received 8 bytes
server received 8 bytes
server received 8 bytes
server received 5 bytes
server received 5 bytes
server received 9 bytes
^Ccse20191264@cspro:~/sp/prj3/20191264/task_2$
```

-multiclient 결과

```
cse20191264@cspro9:~/sp/prj3/20191264/task_2$ ./multiclient 172.30.10.11 60007 2
child 3848936
child 3848937
[buy] success
1 102 10000
2 1166 2000
3 1606 3000
4 1345 4000
5 2089 5000
6 38504 6000
7 1840 7000
8 4165 8000
9 2602 9000
10 4641 5100
[sell] success
[buy] success
[buy] success
1 102 10000
2 1165 2000
3 1606 3000
4 1345 4000
5 2089 5000
6 38495 6000
7 1840 7000
8 4165 8000
9 2609 9000
10 4641 5100
[sell] success
1 102 10000
2 1165 2000
3 1606 3000
4 1345 4000
5 2089 5000
6 38495 6000
7 1840 7000
8 4171 8000
9 2609 9000
10 4641 5100
```

4. 성능 평가 결과 (Task 3)

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

1. 확장성 : Client 개수 변화에 따른 동시 처리율 변화 분석

→ORDER_PER_CLIENT의 수는 10으로 고정하여 측정하였고 동시처리율은 elapsed time을 초 단위로 바꿔 계산하였다.

-Event-driven Server

# of Client	Elapsed time	동시 처리율(초당, int)
1	6553 microseconds	1526
5	10826 microseconds	4618
10	17971 microseconds	5564
20	32247 microseconds	6202
50	67922 microseconds	7361
100	126403 microseconds	8025
150	234060 microseconds	6408
200	1089518 microseconds	1835
500	1307648 microseconds	539

→Event-driven server의 결과표를 보면 client의 수가 높아짐에 따라 elapsed time이 높아지지만 동시 처리율은 낮아지지 않고 계속해서 증가함을 알 수 있다. 하지만 client의 수가 100을 넘어가게 되면 elapsed time은 급격히 증가하게 되고 동시처리율은 급격히 감소하는 것을 알 수 있다 .

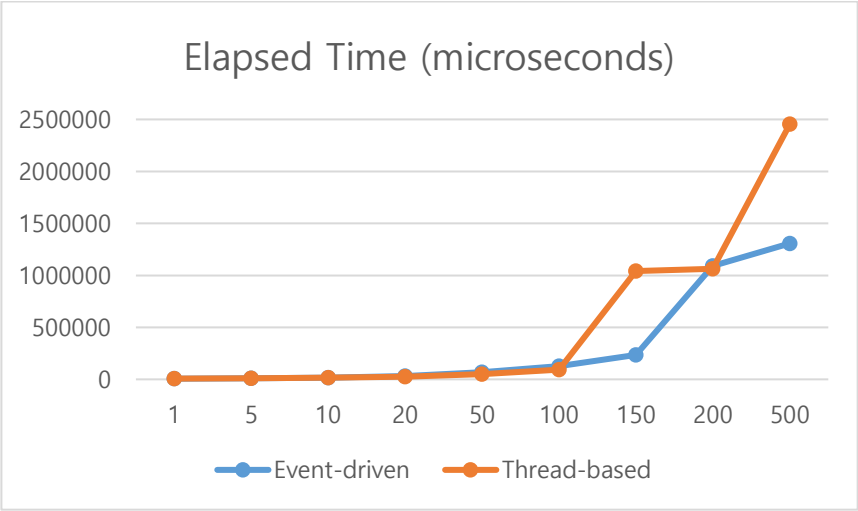
-Thread-based Server

# of Client	Elapsed time	동시 처리율(초당, int)
1	7313 microseconds	1367
5	11218 microseconds	4457
10	15666 microseconds	6383
20	26010 microseconds	7689

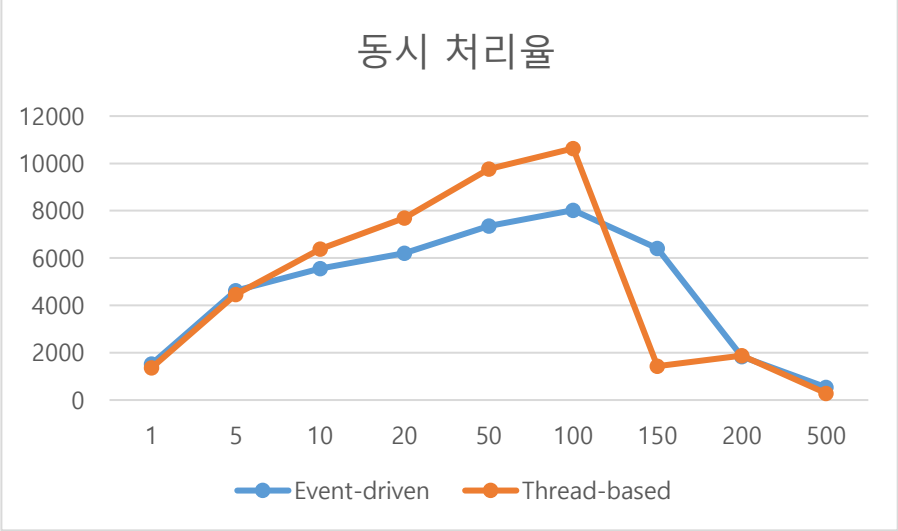
50	51196 microseconds	9766
100	94034 microseconds	10634
150	1043202 microseconds	1437
200	1064303 microseconds	1879
500	2454448 microseconds	287

→Thread-based server의 결과표를 보면 client의 수가 늘어남에 따라 elapsed time 또한 늘어나지만 동시처리율은 계속해서 늘어남을 알 수 있다. 하지만 client의 수가 100개를 넘어가게 되면 elapsed time은 급격히 증가하게 되고 그에 따라 동시처리율은 급격히 감소하는 것을 알 수 있다.

-Elapsed Time



-동시처리율



-위의 결과표를 elapsed time과 동시처리율 별로 그래프로 나타내 보았다. 먼저 elapsed time 그래프를 보면 Event-driven과 Thread-based 모두 client의 수가 늘어남에 따라 elapsed time이 늘어나게 된다. 그래프에서는 시각적으로 확인하기 힘들지만 위의 결과표를 비교해보면 client의 수가 5를 넘어가는 시점부터 thread-based server가 미세하게 더 빠른 elapsed time을 갖는 것을 알 수 있다. 이에 따라 동시처리율은 그래프를 통해 알 수 있듯이 client의 수가 5일 때까지는 event-based가 더 높지만 5를 넘어가는 시점부터는 thread-based가 동시처리율이 더 높은 것을 확인할 수 있다. 하지만 client의 수가 100을 넘어가게 되면 elapsed time과 동시처리율의 양상이 달라지게 된다. Client의 수가 100을 넘으면 양쪽 방식 모두 Elapsed time이 많이 높아지지만 Thread-based 방식에서 그 간극이 더 크게 나타나게 되며 동시처리율에서도 양쪽 모두 동시처리율이 client의 수가 100을 넘으면 많이 감소하지만 Thread-based의 동시처리율 감소의 정도가 event-driven 방식에 비해 큰 것을 알 수 있다.

결과표와 그래프를 보면 여러 thread를 사용하는 Thread-based server가 단일 프로세스를 사용하는 event-driven에 비해서는 전체적으로 좋은 성능을 보이는 것을 확인할 수 있었다. 하지만 client의 수가 어느 정도를 넘게 되면 Thread-based server의 elapsed time과 동시처리율의 변화 폭이 event-based server에 비해 컸던 것으로 보아 Thread-based server는 pthread_create 함수를 통해 미리 생성해둔 worker thread의 수보다 더 많은 client와 client의 요청이 들어올 경우 오버헤드가 발생하면서 성능이 많이 떨어짐을 알 수 있었다. Event-driven server 또한 단일 프로세스를 이용해 select함수를 통해 루프를 돌면서 event가 발생하면 connfd를 통해 요청을 처리하므로 select가 감시할 수 있는 최대 connfd의 개수(일반적으로 1024)를 넘어가게 되면 성능이 떨어짐을 알 수 있었다.

2. 워크로드에 따른 분석 : Client 요청 타입에 따른 동시 처리율 분석

→Client의 수는 100, ORDER_PER_CLIENT의 수는 10으로 고정한 뒤 각 경우마다 5번씩 실행해본다.

-Event-driven server

① Client가 show, buy, sell 모두 요청하는 경우

Elapsed Time	동시처리율(초당, int)
118409 microseconds	8445

119629 microseconds	8359
119112 microseconds	8395
114233 microseconds	8754
114565 microseconds	8728

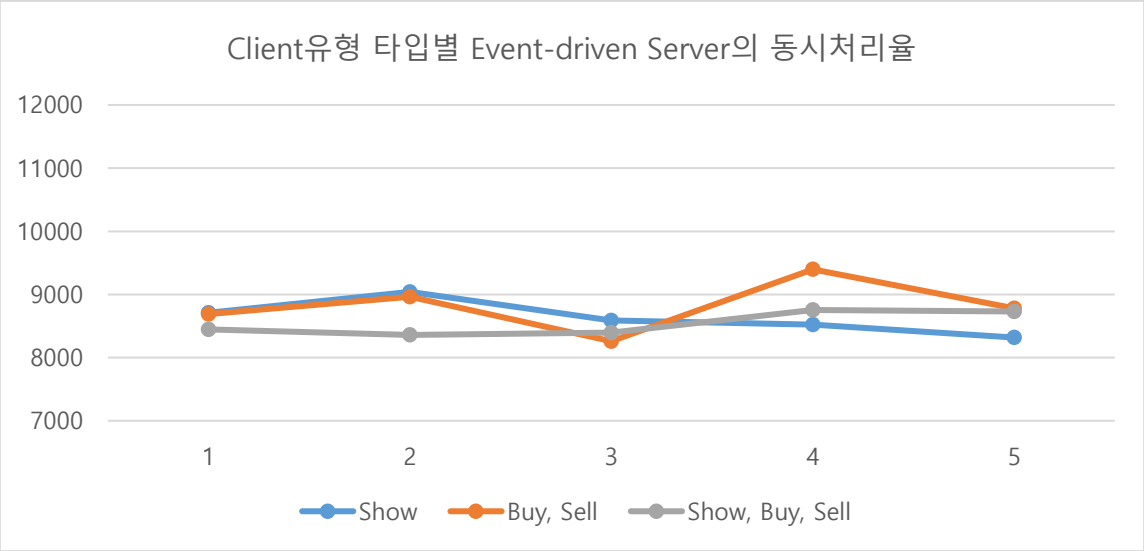
② Client가 show만 요청하는 경우

Elapsed Time	동시처리율(초당, int)
114815 microseconds	8709
110607 microseconds	9041
116415 microseconds	8589
117309 microseconds	8524
120212 microseconds	8318

③ Client가 buy와 sell만 요청하는 경우

Elapsed Time	동시처리율(초당, int)
115075 microseconds	8689
111535 microseconds	8965
121053 microseconds	8260
106411 microseconds	9397
113878 microseconds	8781

-앞에서 결과를 예상해볼 때는 이진 탐색을 통해 모든 주식 정보를 탐색해야 하는 show의 요청만 있을 경우가 다른 경우에 비해 elapsed time이 더 많이 나올 것이라고 예상했다. 하지만 위의 결과표를 보듯이 Event-driven server에서는 client의 요청의 타입에 상관없이 buy, sell, show를 모두 요청하거나 show만 요청하거나 buy, sell만 요청해도 elapsed time과 동시처리율이 모두 유사한 값을 가지는 것을 알 수 있었다. 따라서 client의 요청 타입은 event-driven server의 성능에 영향을 끼치지 않는다고 할 수 있다. 위의 결과표의 동시처리율을 그래프로 나타내면 다음과 같다.



-Thead-based server

① Client가 show, buy, sell 모두 요청하는 경우

Elapsed Time	동시처리율(초당, int)
91543 microseconds	10923
92661 microseconds	10972
94673 microseconds	10562
90329 microseconds	11070
92407 microseconds	10821

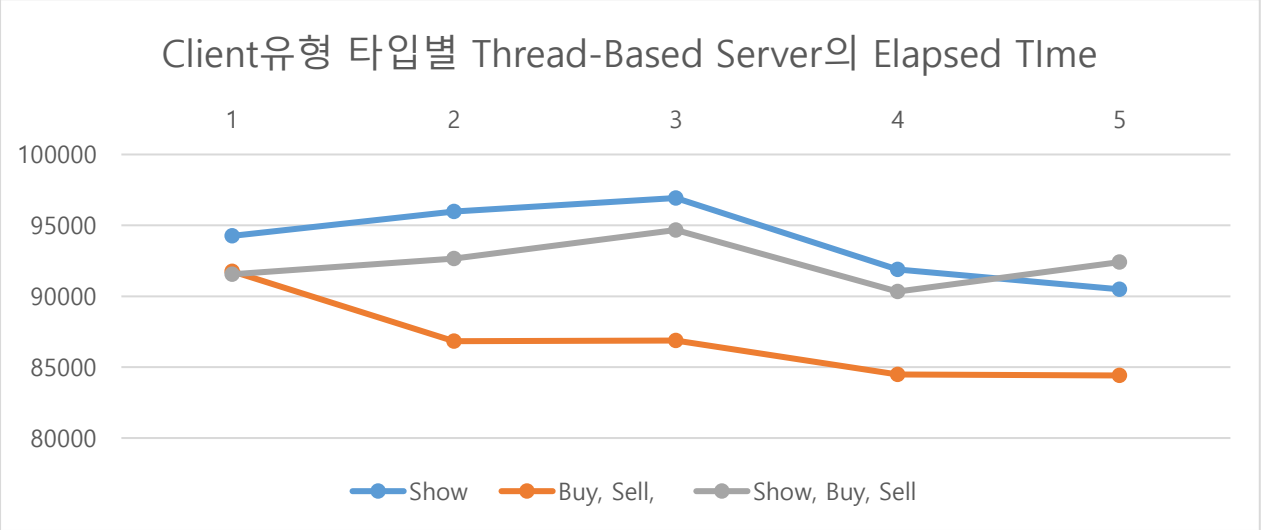
② Client가 show만 요청하는 경우

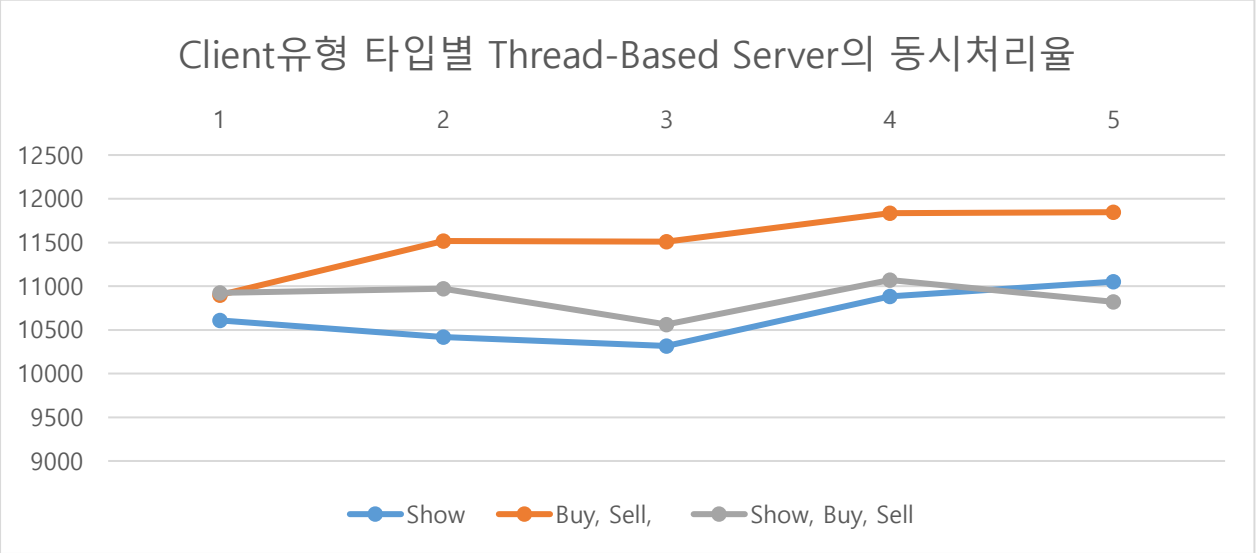
Elapsed Time	동시처리율(초당, int)
94256 microseconds	10609
95976 microseconds	10419
96929 microseconds	10316
91889 microseconds	10882
90489 microseconds	11051

③ Client가 buy와 sell만 요청하는 경우

Elapsed Time	동시처리율(초당, int)
91757 microseconds	10899
86840 microseconds	11515
86873 microseconds	11511
84482 microseconds	11836
84412 microseconds	11846

→위의 결과표들을 그래프로 나타내면 다음과 같다.





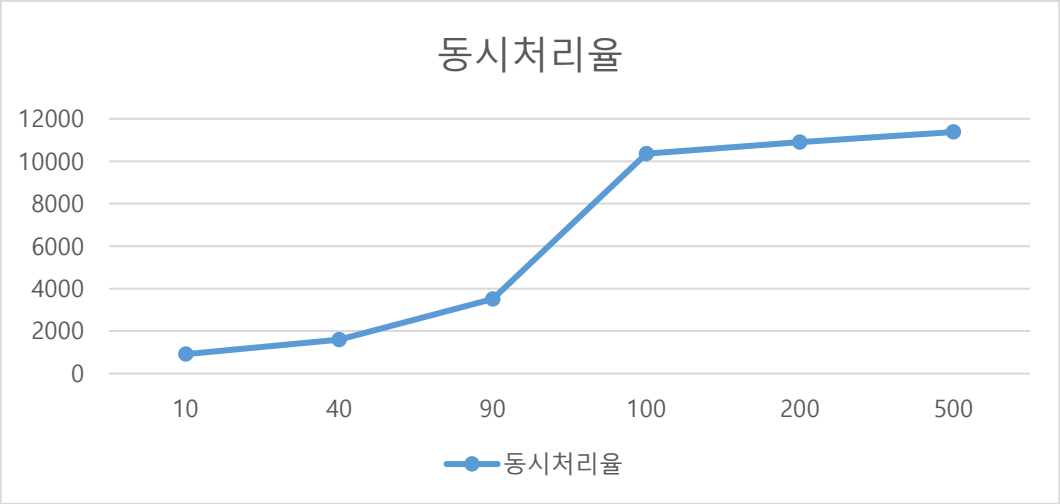
-Thread-based Server 또한 client에게서 show의 요청만 들어올 때 다른 경우에 비해서 elapsed time이 더 많이 늘어날 것으로 예상했다. 위의 결과 그래프로 볼 때 elapsed

time 그래프에서는 확실히 show의 요청만 들어왔을 때가 가장 높은 위치에 있고 동시처리율 그래프에서는 가장 아래의 위치에 있는 것을 알 수 있다. 따라서 show만 요청이 들어왔을 때가 동시처리율이 가장 낮음을 알 수 있다. 하지만 위의 elapsed time의 그래프의 경우 값들의 단위가 microsecond이므로 각 경우의 차이는 엄청난 차이라고 하기 어렵다. 즉, buy, sell만을 요청받았을 때 가장 높은 동시처리율을 보이긴 하지만 그 차이 또한 미세하다고 할 수 있다. 구현 전에는 show의 요청을 처리하기 위해서는 주식 정보를 모두 저장하고 있는 binary tree를 모두 탐색해야 하고 또 P와 V함수를 통한 locking과 unlocking 등으로 소요 시간이 더 클 것으로 예상하였다. 하지만 예상한 것에 비해 각 경우의 측정값들의 차이가 확연하지 않았던 이유는 처음 stock.txt 파일에 적혀 있는 주식의 ID값이 순서대로 정렬되어 있어 편향된 binary tree가 만들어져 buy와 sell 또한 해당 ID를 찾는 search 함수가 있어 show와 elapsed time의 차이가 크지 않았던 것 같다.

3. 기타 분석 : NTHREAD 값에 따른 Thread-based Server의 동시 처리율 분석

→앞에서 확장성 분석을 할 때에는 NTRHEAD 값, 즉 pthread_create를 통해 미리 생성해 두는 worker thread의 개수를 100으로 고정하고 분석을 하였다. 따라서 이번에는 NTHREAD 값을 변화시켜보면서 Thread-based server의 동시처리율을 구하려고 한다. Client의 수는 100으로, OREDER_PER_CLIENT의 수는 10으로 고정하며 SBUFSIZE는 1000으로 설정한 후 진행하였다.

# of NTHREAD	Elapsed Time	동시처리율(초당, int)
10	1093227 microseconds	914
40	624189 microseconds	1602
90	284423 microseconds	3515
100	96569 microseconds	10355
200	91831 microseconds	10889
500	87896 microseconds	11377



-결과를 통해 NTRHEAD의 수가 늘어남에 따라 elapsed time은 확연히 줄어들고 동시처리율 또한 높아지는 것을 알 수 있다. 결과표와 그래프를 볼 때 elapsed time와 동시처리율 모두 NTHREAD가 100을 넘으면 약간의 변화는 있지만 큰 차이가 없는 것은 client의 수를 100으로 초기에 고정시켜 설정했기 때문에 worker thread의 개수가 client의 수에 비해 훨씬 많게 되므로 거의 유사한 측정값이 나오게 된다.