System Programming Project 3

담당 교수 : 김영재

이름 : 김태규

학번 : 20191243

1. 개발 목표

이번 프로젝트의 목표는 Concurrent Stock Server를 Event-driven method와 Thread-based method를 기반으로 Concurrency Issue 없이 구축하는 것에 있다.

Task1에서는 I/O Multiplexing 기법을 통해 Event-driven Concurrent Server를 구축한다. 다수의 client가 보내는 connection request를 server가 받을 때마다 생성되는 connected descriptor를 listening descriptor와 array 하나로 묶는다. Single process는 반복하여 순회하면서 Select 함수를 통해 해당 array의 file descriptor들 중 pending 중인 descriptor를 찾고 각각에 맞는 event를 처리함으로써 Concurrent Server를 구축한다.

Task2에서는 Producer-Consumer problem 기법을 통해 thread pooling을 구현함으로써 Concurrent Server를 구축한다. Worker threads를 미리 띄우고 새로운 Connected Descriptor가 만들어지면 shared buffer에 connected file descriptor를 add하고, add할 때마다 worker thread가 작동함으로써 conn descriptor를 닫고 그에 맞게 동작하도록(show, buy, sell) 한다.

Task3에서는 Task1과 Task2에서 구현한 것을 토대로 client 수와 client 요청 타입에 따른 동시 처리율 변화를 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Select 함수와 I/O Multiplexing을 통해 single process concurrent server를 구축했다. Multi-client 환경에서도 동시다발적인 command를 concurrency issue 없이 잘 처리한다.

2. Task 2: Thread-based Approach

   Producer-Consumer problem을 통한 thread pooling과 stock item 각각에 대해 First Readers-Writers problem을 통해 concurrent server를 구축했다. 마찬가지로 Multi-client 환경에서도 동시다발적인 command를 concurrency issue 없이 잘 처리한다.

3. Task 3: Performance Evaluation

   Client 수와 client의 요청 타입(workload)에 따른 동시처리율을 비교하여 분석한다. Thread-based server가 event-driven server보다 동시처리율 관점에서 더 우수한 성능을 가진다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  각 client가 server에게 connection 요청을 하면 connected descriptor를 connfd array에 추가한다. 그리고 나서 Server는 Select 함수를 통해 pending input에 대해 지속적으로 확인한다. connfd에 pending된 input이 존재하면 해당 Client의 request에 맞는 명령어를 수행하고, Listen descriptor에 Pending된 Input이 존재하면 connected descriptor를 connfd array에 추가하는 방식으로 운영된다.

  File Descriptor는 fd_set data type을 이용해 read_set과 ready_set이란 Set으로 관리한다. read_set은 를 의미하고, ready_set은 read_set의 copy로, read_set을 보존하기 위해 Select 함수는 read_set의 복사본인 ready_set를 조정한다.

  ✓ epoll과의 차이점 서술

  select와 epoll은 둘 다 Linux에서 다중 I/O 이벤트를 처리하기 위한 system call로, 주로 네트워크 프로그래밍에서 여러 file descriptor를 동시에 monitoring하는 데 사용된다. 이 두 가지 방법은 성능과 사용 방식에서 여러 가지 차이점이 존재한다.

  먼저, select는 file descriptor set을 monitoring한다. 이를 위해 read, write, 예외 file descriptor의 세 개의 집합을 설정한 후, 이 집합을 select 함수에 전달한다. select는 이들 중 어떤 descriptor가 ready인지 확인하는 방식이다. 그러나 select에는 몇 가지 제한 사항이 있다.

  먼저, 기본적으로 한 process에서 monitoring할 수 있는 fd의 수는 FD_SETSIZE로 제한되며, 이는 일반적으로 1024개이다. 또한, select는 매번 호출할 때마다 모든 file descriptor set을 다시 초기화하고 커널에 전달해야 하므로, fd 수가 많아지면 성능이 저하된다. 그럼에도 불구하고 select는 거의 대부분의 POSIX 호환 시스템에서 지원되며 사용이 비교적 단순하다.

  반면, epoll은 리눅스에서 제공하는 고성능 event-based interface이다. epoll은 epoll_create로 epoll 인스턴스를 생성하고, epoll_ctl로 관심 있는 file

descriptor를 추가, 수정, 삭제한다. 이후 epoll_wait를 호출하여 준비된 file descriptor를 monitoring합니다. epoll의 성능은 file descriptor의 수와 무관하게 높은 성능을 유지하는데, 이는 내부적으로 준비된 file descriptor만을 추적하여 이벤트를 전달하기 때문이다. 또한, epoll은 level-triggered와 edge-triggered 모드를 지원하여, 이벤트 발생 시점의 유연한 처리가 가능하다. 다만, epoll의 설정은 select보다 복잡하고 리눅스 전용이라는 단점이 있다.

결론적으로, select는 간단하고 범용적인 다중 I/O 처리 방법으로 적은 수의 file descriptor를 다루는 application에 적합하다. 반면, epoll은 대규모 네트워크 서버와 같이 많은 수의 file descriptor를 효율적으로 처리해야 하는 application에 적합하다.

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

  ✓ Master Thread의 Connection 관리

  Main 함수에서 미리 NTHREADS만큼의 worker threads를 생성하고, 사용하지 않는 thread는 sleep으로 멈춰놨다가 client가 연결을 요청하여 worker thread가 필요한 상황이 오면 thread를 깨워 work하도록 한다. 이를 위해서 shared buffer를 구현한다. Master thread는 이 shared buffer(sbuf)를 통해 connection을 관리한다.

  새로운 connected file descriptor가 만들어지면 master thread가 이 descriptor를 sbuf의 item으로 추가한다. 이에 따라 slot이 하나 줄고, item이 하나 늘어난다. 즉, slot에 대한 P operation, item에 대한 V operation이 수행된다.

  Available item이 존재하면 worker thread가 이를 sbuf에서 remove하고(slot 1 증가) 작업을 수행한다. Semaphore 관점에서 item에 대한 P operation과 slot에 대한 V operation이 수행된 것이라고 볼 수 있다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Shared buffer를 전역변수로 선언한다.(sbuf_t sbuf) sbuf를 main 함수(master thread)에서 initialize(sbuf_init 함수)하고 Pthread_create 함수를 통해

NTHREAD만큼 worker thread를 미리 생성한다.

Client가 connection request를 보내면 connected descriptor가 반환되고 이 descriptor들을 item으로 하여 sbuf에 추가한다. 이때 item에 대한 V operation이 수행된다. Worker threads는 sbuf에 item이 추가되는 것을 기다리고 있다가 V(&items)가 호출되면 wake-up하여 item을 sbuf에서 remove한 뒤 item을 통해 client가 요청한 명령어를 수행한다. 명령어를 수행한 후에는 connected descriptor를 닫고 다시 sleep state로 전환된다.

- **Task3 (Performance Evaluation)**

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

  - metric 1: Client 수에 따른 동시처리율 비교

  Client 수가 시간당 처리하는 명령어 수에 어떠한 영향을 주는지, 구현 방식에 따라 얼마나 차이가 나는지 비교하기 위한 실험이다. Client의 수를 1, 5, 10, 20, 50, 100으로 변화를 주어 실험을 진행하였다. 모든 Client가 20개의 명령을 요청하고 이를 모두 처리하는데 걸리는 시간을 측정한 후, 요청의 개수를 처리 시간으로 나누어 계산한다.

  metric 2: Client 요청 workload(buy+sell, show, all)에 따른 동시처리율 비교

  buy+sell workload는 writer만 존재하는 workload이고, show workload는 트리의 모든 노드를 순회해야 하는 workload이다. workload를 달리 줌으로써 구현 방식에 따라 시간당 처리하는 명령어 수가 얼마나 차이가 나는지 비교하기 위한 실험이다. 모든 Client가 20개의 명령을 요청하고 이를 모두 처리하는데 걸리는 시간을 측정한 후, 요청의 개수를 처리 시간으로 나누어 계산한다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  metric 1:

  예상 1: 두 approach 모두 client 수가 많아질수록 동시처리율이 높아질 것이다.

예상 2: Event-driven approach는 single process이기 때문에 Thread-based approach가 동시처리율이 더 높을 것이다.

metric 2

예상 1: show는 트리 모든 노드를 순회하므로 show workload의 동시처리율이 가장 낮을 것이다.

예상 2: 실험 2도 마찬가지로 event-based approach가 thread-based approach보다 동시처리율이 더 낮을 것이다.

C. 개발 방법

(1) Event-driven Approach

```
typedef struct item {
    int ID;
    int left_stock;
    int price;
    struct item *right;
    struct item *left;
} item;
```

Binary Tree는 위와 같은 structure type을 가진 item으로 이루어진다.

```
typedef struct pool {
    int maxfd;
    fd_set read_set;
    fd_set ready_set;
    int nready;
    int maxi;
    int clientfd[FD_SETSIZE];
    rio_t clientrio[FD_SETSIZE];
} pool;
```

I/O Multiplexing을 구현하기 위해 pool이라는 structure를 도입하였다. maxfd는 현재 활성화된 file descriptor 중 가장 큰 번호를 가진 descriptor를 의미하고, clientfd array는 활성화된 connected file descriptor를 기록하고 있는 array이다. nready는 descriptor 중 pending input이 있는 것의 개수를 의미한다. 처음에 listenfd를 read_set에 기록하고 read_set의 변동을 막기 위해 나중에 ready_set을 read_set으로 복사한다. 이 변수들은 init_pool 함수를 통해 initialize한다.

main 함수에서 ready set에 read set을 복사한 후 select 함수를 통해 pending input을 가지고 있는 descriptor 수를 확인한다. FD_ISSET을 통해 pending input이 listen fd에 존재하는지 확인하고, 있다면 add_client 함수를 통해 connected descriptor를 client fd array에 새로 추가한다.

add_client 함수에서는 nready를 1만큼 감소시키고 clientfd array의 빈 곳에 connected descriptor를 추가하고 read_set에 추가한다.

add_client 절차가 끝나면 check_client 함수를 통해 pending input이 존재하는 fild descriptor를 하나씩 순회하여 client의 요청 사항을 들어준다.(connect, show, buy, sell..)

Client와 연결이 종료되면 close(connfd)를 통해 descriptor를 close하고 이를 다시 clientfd array에 반영한다.

이러한 flow로 Event-driven Concurrent Server가 동작한다.

(2) Thread-based Approach

```
typedef struct item {
    int ID;
    int left_stock;
    int price;
    int readcnt;
    sem_t mutex, w;
    struct item *right;
    struct item *left;
} item;
```

Thread-based Server의 트리는 위와 같은 노드들로 구성된다. 해당 노드를 read 하고자 하는 reader의 수를 의미하는 readcnt와 mutex, w(semaphore)가 추가되었다.

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

Thread pooling을 위해 위와 같은 pool이라는 구조체를 도입하였다. 그리고 sbuf_t type의 sbuf라는 전역변수를 선언하고 main 함수 내부에서 sbuf_init 함수를 통해 구조체 내의 변수들을 initialize한다.

전역변수 sbuf를 초기화한 후 worker threads를 NTHREADS만큼 미리 만들어 놓는다. 그 후, connected descriptor가 새롭게 만들어지면 sbuf_insert 함수와 V(&items)를 통해 이 descriptor를 sbuf에 넣어서 worker thread가 이를 바로 consume 하도록 한다.

그 후에는 thread 함수의 flow를 보면 처음에 sbuf_remove를 통해 thread reaping을 reserve 해놓고 client가 요청하는 작업을 수행한다.

이와 같은 flow로 thread-based concurrent server가 동작한다.

(3) Client 요청 작업 수행

```
void OperateCmd(int connfd, char *buf) {
    int ID, num;
    command cmd = parseline(buf, &ID, &num);
    if (cmd == show) {
        ShowStock(connfd);
    }
    else if (cmd == buy) {
        BuyStock(connfd, ID, num);
        return;
    }
    else if (cmd == sell) {
        SellStock(connfd, ID, num);
    }
    else if (cmd == exitt) {
        ExitServer(connfd);
    }
    else if (cmd == errorr) {
        PrintError(connfd);
    }
    return;
}
```

Client가 요청한 작업을 수행하는 단계에 접어들면 OperateCmd라는 함수가 호출된다. 이 함수가 호출되면 parseline 함수를 통해 client가 보낸 메시지에서 원하는 작업(show, buy, sell..)과 주식 종류, 수량에 대한 정보를 얻는다. 이를 통해

client가 요청한 작업을 실질적으로 수행하는 단계로 넘어간다.

```
void ShowStock(int connfd) {
    char buf[MAXLINE] = {0};
    char *ptr = buf;

    for (int i = 0; i < tree_size; i++) {
        P(&(Tree[i]->mutex));
        (Tree[i]->readcnt)++;
        if (Tree[i]->readcnt == 1)
            P(&(Tree[i]->w));
        V(&(Tree[i]->mutex));

        ptr += sprintf(ptr, "%d %d %d\n", Tree[i]->ID, Tree[i]->left_stock,
                                                        Tree[i]->price);

        P(&(Tree[i]->mutex));
        (Tree[i]->readcnt)--;
        if (Tree[i]->readcnt == 0) {
            V(&(Tree[i]->w));
        }
        V(&(Tree[i]->mutex));
    }
    Rio_writen(connfd, buf, MAXLINE);
}
```

위는 show 명령어에 대한 함수이다. (Event-driven의 코드에서 고려해야 하는 부분은 Thread-based의 코드에서 모두 cover 가능하므로 thread-based 기준으로 설명)

buf라는 char type 배열을 선언해 출력하고자 하는 string을 buf에 넣는다. 이 과정에서 반복문과 ptr 변수를 통해 한번에 출력할 수 있도록 한다.

먼저 작업하고자 하는 노드의 mutex에 P operation을 한 뒤 readcnt를 1만큼 increment한다. Reader가 한 명이라도 있으면 해당 노드의 w semaphore에 대해 P operation을 하여 writer가 수정하지 못하도록 대기시킨다. Reader가 없다면 (readcnt == 0) w에 대해 V operation을 수행하여 writer가 해당 노드를 수정할 수 있도록 허용한다.

```
void BuyStock(int connfd, int ID, int num) {

    item *temp = SearchTree(ID);

    P(&(temp->w));
    if (temp->left_stock < num) {
        V(&(temp->w));
        Rio_writen(connfd, "Not enough left stock\n", MAXLINE);
    }
    else {
        temp->left_stock -= num;
        V(&(temp->w));
        Rio_writen(connfd, "[buy] success\n", MAXLINE);
    }
}

void SellStock(int connfd, int ID, int num) {
    item *temp = SearchTree(ID);

    P(&(temp->w));
    temp->left_stock += num;
    V(&(temp->w));
    Rio_writen(connfd, "[sell] success\n", MAXLINE);
}
```

위는 buy와 sell 명령어에 대한 함수이다.

Buy하고자 하는 item을 SearchTree 함수를 통해 temp 변수에 넣고 해당 item의 w semaphore에 대해 P operation을 수행한다. 주식의 재고가 사고자 하는 수량보다 적으면 바로 V operation을 수행하고, 재고가 충분하다면 temp의 left_stock 변수를 num만큼 더해준 뒤 V operation을 수행한다.

Sell 명령어도 동일하게 수행된다.

Exit의 경우, client가 종료를 위해 exit라는 메시지를 server에 보내면 다시 exit라는 메시지를 출력하도록 하여 스스로 종료하도록 구현하였다.

3. 구현 결과

Task1과 Task2에서 구현한 Event-driven server와 Thread-based server 모두 multiple clients의 동시다발적인 명령을 concurrency issue 없이 제대로 수행하고 있다.

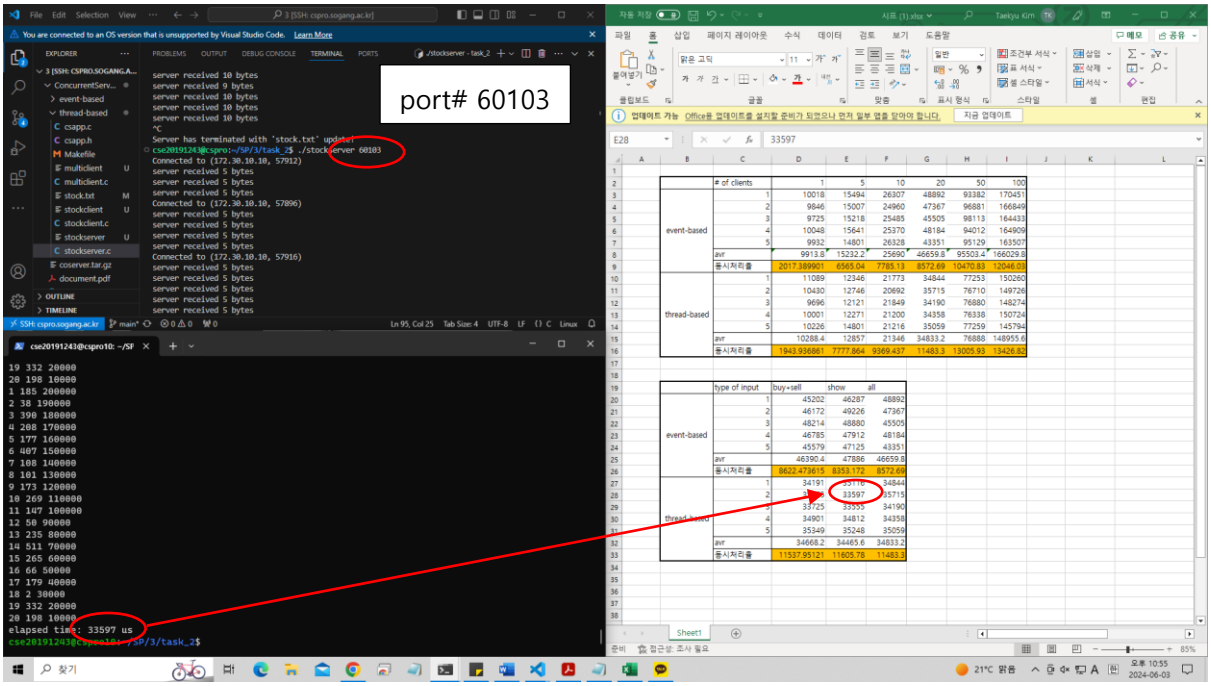
하지만 Tree 구현을 binary search tree로 했는데 좌우 balanced가 보장되는 트리로 구현했다면 속도가 더 빠르지 않았을까 한다.

또한 client의 현재 잔고를 반영하여 구현하면 좀 더 real-world에 가까운 server를 구현할 수 있었을 것 같다.

4. 성능 평가 결과 (Task 3)

```
#define MAX_CLIENT 100
#define ORDER_PER_CLIENT 20
#define STOCK_NUM 20
#define BUY_SELL_MAX 10
```

configuration은 위와 같이 설정하였다.



위는 클라이언트 수가 20이고, 모든 client가 show만 요청하는 경우에 대한 출력 결과 캡처본이다. 이와 같이 각각의 case에 대해 5번씩 실험하여 결과를 도출하였다.

```
    /************/
    struct timeval start;
    struct timeval end;
    unsigned long e_usec;
    gettimeofday(&start, 0);
    /************/

/*  fork for each client process    */
    while(runprocess < num_client){
        //wait(&state);
        pids[runprocess] = fork();
```

```
. . . . . .
. . . . . .

    for(i=0;i<num_client;i++){
        waitpid(pids[i], &status, 0);
    }

    /************/
    gettimeofday(&end, 0);
    e_usec = ((end.tv_sec*1000000) + end.tv_usec) -
                                        ((start.tv_sec*1000000)+start.tv_usec);
    printf("elapsed time: %lu us\n", e_usec);
    /************/
```

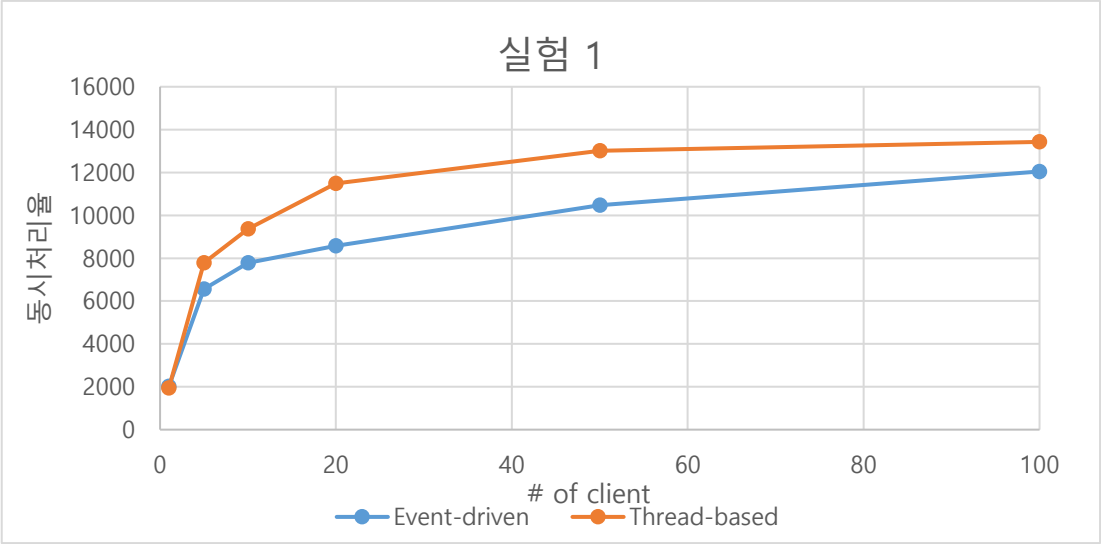
위와 같이 multiclient.c에서 while(runprocess < num_client)를 수행하기 전과 마지막 for문을 수행한 후의 시점에서 gettimeofday 함수를 통해 시간 측정을 한 뒤 elapsed time을 구하였다.

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

그리고 위의 코드에서 int형 변수 option의 값을 0 또는 rand()%2+1로 바꾸면서 workload를 조정하였다.

(1) 실험 1: Client 수 변화에 따른 동시처리율 변화 분석

	# of clients	1	5	10	20	50	100
event-based	1	10018	15494	26307	48892	93382	170451
	2	9846	15007	24960	47367	96881	166849
	3	9725	15218	25485	45505	98113	164433
	4	10048	15641	25370	48184	94012	164909
	5	9932	14801	26328	43351	95129	163507
	avr	9913.8	15232.2	25690	46659.8	95503.4	166029.8
	동시처리율	2017.389901	6565.04	7785.13	8572.69	10470.83	12046.03
thread-based	1	11089	12346	21773	34844	77253	150260
	2	10430	12746	20692	35715	76710	149726
	3	9696	12121	21849	34190	76880	148274
	4	10001	12271	21200	34358	76338	150724
	5	10226	14801	21216	35059	77259	145794
	avr	10288.4	12857	21346	34833.2	76888	148955.6
	동시처리율	1943.936861	7777.864	9369.437	11483.3	13005.93	13426.82



예상 1: 두 approach 모두 client 수가 많아질수록 동시처리율이 높아질 것이다.

예상 2: Event-driven approach는 single process이기 때문에 Thread-based approach가 동시처리율이 더 높을 것이다.

예상 1, 2 모두 실험 결과와 일치하였다.

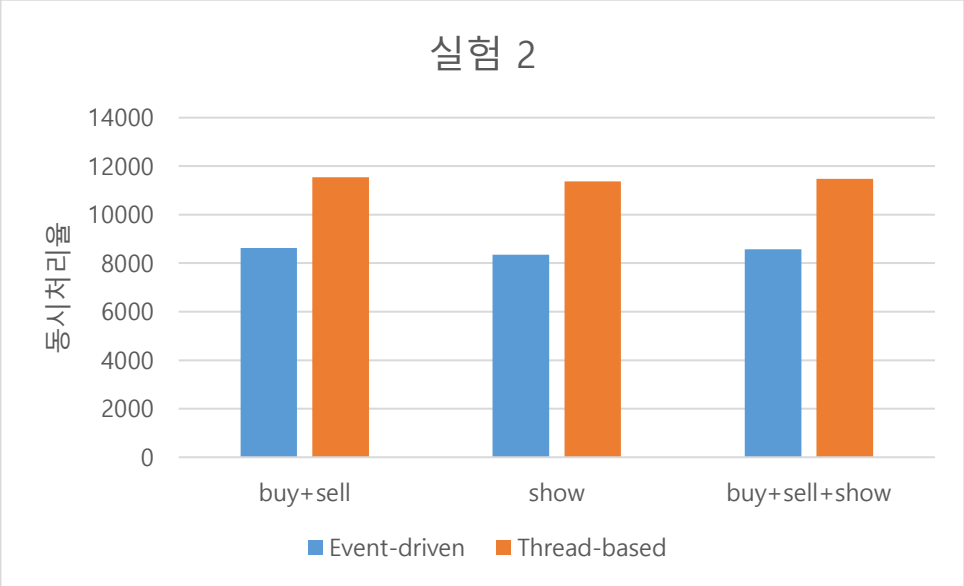
두 방법 모두 client 수가 많아질수록 동시처리율이 높아지고 있고, client 수가 1~5일 때는 동시처리율이 비슷했는데 10명일 때 약 1600 정도 차이가 나기 시작하면서 20~50명일 때 약 3000 정도로 최대 차이를 보였다. client가 100명일 때에는 다시 차이가 1400 정도로 좁혀지고 있다. 이를 통해 client 수가 증가하면 동시 처리율이 높아지지만 일정 수준 이상으로 커지면 증가폭이 감소하는 로그 함수 개형을 가진다는 것을 알 수 있었다.

첫 실험 때는 Thread-based approach에서 thread의 수를 100으로 설정해놓았는데 multi core의 advantage를 높이기 위해 1000으로 올려서 다시 실험해보았지만 실험 결과가 크게 달라지는 점이 없었다. 이를 통해 thread의 수가 무한정 커진다고 해도 계속 증가하는 것이 아니라 일정 수준의 임계값을 가진다는 것을 알 수 있었다.

(2) 실험 2: Client 요청 type에 따른 동시처리율 변화 분석

실험 2는 Client 수를 20명으로 고정시켜놓고 진행하였다.

	type of input	buy+sell	show	all
event-based	1	45202	46287	48892
	2	46172	49226	47367
	3	48214	48880	45505
	4	46785	47912	48184
	5	45579	47125	43351
	avr	46390.4	47886	46659.8
	동시처리율	8622.473615	8353.172	8572.69
thread-based	1	34191	34127	34844
	2	35175	35923	35715
	3	33725	35791	34190
	4	34901	34812	34358
	5	35349	35248	35059
	avr	34668.2	35180.2	34833.2
	동시처리율	11537.95121	11370.03	11483.3



예상 1: show는 트리 모든 노드를 순회하므로 show workload의 동시처리율이 가장 낮을 것이다.

예상 2: 실험 2도 마찬가지로 event-based approach가 thread-based approach보다 동시처리율이 더 낮을 것이다.

예상 2는 예측대로 실험 결과와 일치한다.

값을 비교해보면 예상1과 같이 예측대로 Event-driven approach와 Thread-based approach 모두 show workload가 동시처리율이 가장 낮다. 하지만 각 workload에 따른 동시처리율의 차이가 어느 정도 유의미한 정도로 날 것이라고 예상했는데 실험 결과로부터 봤을 때 매우 미미한 정도이다.

이에 대한 원인을 생각해보았는데 show workload는 reader만 존재하고 writer는 존재하지 않는다. 하지만 buy+sell workload는 writer만 존재하므로 semaphore의 영향을 크게 받는다. 그래서 show workload와 거의 비슷한 동시처리율을 가질 수 있겠다고 생각했다. 또한, 트리 구현을 simple binary tree로 구현했는데 이는 트리의 좌우 balance를 보장해주지 않는다. 그래서 buy와 sell 명령어를 수행할 때 극단적으로는 show와 비슷하게 시간이 걸릴 수도 있다. 그러므로 balance를 보장할 수 있는 트리로 구현했다면 예상1과 더 가까운 결과를 얻을 수 있었을 것이라고 생각한다.

Event-based와 Thread-based approach 모두 동시처리율을 비교해보면 buy+sell > all(random) > show의 대소 관계를 갖는다. 이를 통해 semaphore로 인한 overhead보다 트리의 모든 노드 순회를 함으로써 생기는 overhead의 영향이 더 크다고 가설을 세우고 새로운 실험을 진행해볼 수 있겠다고 생각했다.

위의 실험은 주식 종목의 종류를 20개로 설정했는데, 즉 노드의 개수를 20개로 설정, 종목의 개수를 훨씬 더 큰 숫자로 설정한다면 show 명령어에 걸리는 시간이 더욱 커져 예상 1과 가까운 결과를 도출해낼 수 있을 것이라고 생각한다.