Multicore Programming Project 2

담당 교수 : 박성용

이름 : 안희원

학번 : 20211550

1. 개발 목표

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

해당 프로젝트에서는 여러 client들의 동시 접속 및 서비스를 위한 Concurrent stock server를 구축하는 것이 목표이다. Concurrent한 server를 구현하기 위한 방법으로는 3가지가 존재하는데, 이번 프로젝트에서는 Select를 이용하는 Event-driven Approach와 Pthread를 이용하는 Thread-based Approach를 사용하여 구현한다. Client는 server에 show, buy, sell, exit 명령을 내릴 수 있고, 주식과 관련된 정보는 stock.txt 파일로 관리한다. 마지막으로 각각의 방식으로 구현한 서버를 여러 기준을 통해 비교, 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   초기에 stock server를 열고, client가 서버와 연결되면 client는 명령어를 사용할 수 있게 된다. Select()를 통해 event가 발생한 connection fd들을 찾는 것이 concurrent한 서버를 구현하는 핵심 부분이다. Client가 사용할 수 있는 명령어인 show, buy, sell, exit은 check_clients()에서 수행되며 그에 맞는 결과가 client에 전달되어 echo값과 동시에 출력된다.

2. Task 2: Thread-based Approach

   Task 1과 실행했을 때의 결과는 거의 같다. 그러나 concurrent 서버를 구현하는 방법에서 차이가 있다. 이번에는 여러 개의 thread를 미리 생성한 후에 client와 연결하는 방식이다. 이때 서버의 명령어들은 func()에서 처리된다. Global 변수를 사용하는 경우 race condition이 일어나는 것을 방지하기 위해 semaphore를 이용하였다. 처리된 명령어의 결과값은 마찬가지로 서버로 들어온 명령을 echo함과 동시에 client server로 출력된다.

3. Task 3: Performance Evaluation

위에서 구현한 event-driven 방식과 thread-base 방식을 client 개수, 워크로드 등의 관점에서 동시처리율을 계산해 이를 비교, 분석한다. 동시처리율을 계산하기 위해 Elapse time을 계산해야 하는데, 이를 위해서 multiclient.c에 clock()을 이용하여 시간을 계산할 수 있는 코드를 추가하였다. 또한, CLIENT_NUM과 각 코드의 option을 조절해가며 실험을 진행하였다.

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

I/O Multiplexing은 while문을 통해 listenfd와 connfd array를 지속적으로 체크하며 select를 이용해 listenfd에 pending input이 있다면 connection을 맺기 위해 accept를 하고, connfd를 생성한다. 이때, listenfd의 active 유무는 read_set으로 관리하며 event 발생 여부는 read_set을 copy한 ready_set으로 관리된다. 이후 connfd array에 pending input이 있는, 즉 ready_set에 pending bit가 1인 connfd에 대해 client의 요청을 서버에서 차례로 제공하도록 한다. 이후, 모든 client에 대해 connection이 끊어지면 sigint_handler()와 save() 함수를 이용해 stock.txt에 새로운 stock 정보를 업데이트해준다.

✓ epoll과의 차이점 서술

이번 프로젝트에서는 pending input의 여부를 확인하는 용도로 Select()를 사용하였는데, select()의 작동 방식은 loop를 이용해 fd set을 하나씩 탐색하며, FD_ISSET으로 pending input이 있는지를 체크하는 것이다. 그러나 이 방식은 fd set 전체를 탐색하기 때문에 불필요한 체크가 일어나 기본적으로 fd 개수가 많아질수록 실행 속도가 느려져 시간적인 측면에서는 비효율적이다. 또한 fd set 전체를 OS에 전달하는 것에 대해서도 오버헤드가 일어난다는 단점이 있다. 반면, epoll()는 select()의 이러한 단점을 보완하여 리눅스 환경에서 사용할 수 있도록 만든 것으로, OS가 직접 fd를 관리해주어 fd set을 넘길 필요가 없다. 또 loop를 이용해 구현되지 않으므로 select에 비해 실행 속도가 빠

르다는 차이점이 있다.

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

  ✓ Master Thread의 Connection 관리

  master thread의 경우, 먼저 define된 NTHREADS 개수만큼의 worker thread를 미리 생성한다. 이후 while문을 돌며 connection이 맺어지기를 기다린다. 이때, client로부터 요청이 들어와 accept를 해 connection이 맺어져 connfd가 생성되면 이를 sbuf에 삽입해 관리한다. 즉, worker thread는 sbuf에 있는 connfd를 차례대로 가져가 서버 간의 통신을 하는 것이다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Connection이 맺어져 worker thread에서 client의 요청을 수행하기 위해서는 우선 thread()가 실행된다. 이때, while문 내에서 sbuf_remove()를 통해 위에서 sbuf에 삽입해 두었던 connfd를 하나 꺼내 이를 이용해 통신을 하게 된다. 만약, 현재 연결되어 있는 connfd가 close되면 while문이 다시 돌아 또 새로운 connfd를 꺼내 요청을 수행하는 방식으로 작동된다. 이때, thread가 전부 종료되면 새롭게 stock.txt를 업데이트해줘야 하므로 현재 연결된 client 수를 표시하는 client_num 변수의 계산 또한 while문 내에서 진행된다. 이때, client_num 변수는 semaphore를 이용하여 다른 thread에 의해 정보가 손상되지 않도록 하였다.

- **Task3 (Performance Evaluation)**

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

  이번 프로젝트의 성능 평가의 기준은 동시처리율이다. 동시처리율은 시간 당 client 처리 요청 개수를 의미하기 때문에, CLIENT_NUM * ORDER_PER_CLIENT / elapse time라고 계산할 것이다. 첫번째 측정으로는 ORDER_PER_CLIENT값은 고정시켜 놓고, CLIENT_NUM을 계속 변화시키며 그에 따른 elapse time을 측정할 것이다. 두번째 측정으로는 워크로드에 따라서 elapse time을 측정할 것이므로, CLIEN_NUM과 ORDER_PER_CLIENT 모두 고정된 값으로 실험한다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  이론에 따르면, event-driven 방식이 thread 방식보다 overhead가 적기 때문

에 더 좋은 퍼포먼스를 보일 것이라 예상된다. 또한, thread 방식의 경우 semaphore를 사용하였기 때문에 이에 따른 elapse time의 증가 역시 고려하여야 할 것이다. 따라서 event-driven 방식의 elapse time이 대체적으로 더 빨라 동시 처리율 역시 event-driven 방식이 더 높을 것 같다고 예상한다. 또한 두번째 측정의 경우에는, show의 경우 모든 tree를 다 돌며 stock 정보를 보여줘야 하지만, buy 또는 sell의 경우에는 tree를 search하다 찾고자 하는 id의 stock을 찾으면 그 즉시 stop하기 때문에 show만 하는 경우에 elapse time이 더 빠를 것으로 보인다. 따라서 동시 처리율은 show만 하는 경우보다 buy 또는 sell만 하는 경우가 더 높을 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)**

Stock은 linked list로 구현한 binary search tree 구조로 관리된다. stock과 관련된 정보(id, left_stock, price 등)는 item 구조체를 통해 저장된다. Linked list로 bst를 구현하기 위해 node 구조체를 선언하였는데, 여기에는 stock의 data를 저장하는 item type의 data와 해당 node의 left child를 뜻하는 left, right child를 뜻하는 right가 있다.

프로그램이 실행되면, stock.txt 파일을 열어 읽어온 stock 정보를 item에 저장해 tree를 구성한다. tree를 구성하는 함수는 insert()를 통해 구현하였다. Insert()는 크게 cur_node에 새로운 data를 붙이는 방식으로 구현되어 있는데, 현재 cur_node에 연결된 data가 없으면 cur_node 자체에 data를 저장하고, 아니라면 id의 크기를 비교해 알맞은 cur_node의 child로 data를 삽입한다. 이때 child의 삽입은 재귀를 사용하였다.

Client와 server가 connect되면 client의 요청을 수행하는 check_clients()가 작동한다. Check_clients()에서는 connfd에 event가 발생하면 client로부터 입력된 것을 읽어오고, 형식에 맞게 각각의 변수에 저장된다. 예를 들어 client로부터 "buy 1 6"이라는 명령어가 들어왔다면 cmd = "buy", id = 1, num = 6 이런 식으로 저장된다. 이때, cmd가 무엇이냐에 따라 요청 수행 방식이 달라진다.

1. show

   show()를 통해 구현하였다. Show()의 경우 tree를 root부터 차례대로 방문하며 저장된 정보를 strcat()을 이용해 하나의 string인 output에 붙여 Rio_writen()을 이용

해 client에 출력하도록 하였다. 이때 tree의 순회를 재귀를 통해 구현하였다.

2. buy

   buy의 경우, 우선 buy를 하고자 하는 id를 가진 stock의 node를 찾아야 한다. 따라서 tree를 탐색하는 search() 함수를 구현하였다. search()는 node 구조체 포인터인 cur_node을 사용해 (cur_node->data).ID가 찾고자 하는 node의 id와 같다면 바로 cur_node를 return하도록 하였고, 만약 아니라면 id의 크기를 서로 비교하여 알맞은 방향의 child를 가리키도록 하였다. 만약 찾고자 하는 node가 없다면 NULL을 return한다.

   찾고자 하는 Id를 가진 Node를 찾았다면 사고자 하는 수인 num과 해당 stock의 left_stock 수를 비교해야 한다. 만약, left_stock이 num보다 작다면 "Not enough left stock₩n"을, 아니라면 left_stock에서 num만큼 빼주어 buy를 구현하였다. 성공적으로 buy가 되었다면 "[buy] success₩n"라는 문구가 출력되도록 하였다.

3. sell

   sell 역시 buy와 마찬가지로 search()를 통해 sell하고자 하는 id를 가진 stock의 node를 찾는다. Sell은 무조건 성공하는 가정을 가진 명령어이므로 node를 찾지 못한 경우만 예외 처리해주고, left_stock에 num을 더해 sell을 구현하였다. 성공적으로 sell이 되었다면 "[sell] success₩n"라는 문구가 client에게 출력되도록 하였다.

4. Exit

   multiclient 서버를 기준으로 구현하였기 때문에 exit에 대한 구현은 따로 하지 않았다.

서버의 종료와 그에 따른 stock.txt 업데이트에 관한 부분은 현재 연결되어 있는 client 서버의 개수를 나타내는 client_num 변수를 이용해 구현하였다. add_client()를 통해 client fd를 pool 구조체에 저장할 때, client_num++;을 해준다. 이후 check_clients()에서 connfd가 close될 때 client_num--;가 된다. 다시 main으로 돌아와 만약 client_num이 0이 되면 sigint_handler()를 수행해 서버가 종료되도록 하였다.

Sigint_handler()에서는 stock.txt 파일을 오픈해 save()를 통해 stock.txt를 업데이트하고, freenode()를 통해 stock 정보를 저장했던 bst를 free해준다. save()는 재귀적으로 tree를 돌면서 한 줄 씩 stock의 정보를 파일에 저장하는 함수이다. freenode() 역시 재귀적으로 tree의 모든 node를 free하는 함수이다.

Task_2의 경우, 위에서 설명한 Task_1의 구조와 거의 동일하지만 thread를 다루는 것에서 차이가 있다. Task_2에서는 명령어에 대한 처리를 func()에서 수행한다. Thread는 서로 데이터를 공유하므로 mutex라는 semaphore를 이용해 이에 따른 문제 상황을 방지하였다. 작성한 코드에서 주의해야 할 데이터는 client_num과 stock 정보를 저장하는 tree의 root를 나타내는 node 구조체 포인터 root이다. 위에서 설명한 insert(), show(), search(), save(), freenode()의 내부의 경우 두 변수 모두 global 변수로써 사용하는 것이 아니라, 직접 parameter로 root를 받아 함수 내부에서는 local 변수로써 사용하였으므로 semaphore를 할 필요가 없다. 그러나, func()에서 if문을 통해 구현한 각각의 명령어의 경우, root를 사용하므로 critical section이 된다. 따라서 각 명령어를 수행하기 전에 P(&mutex)를 해주고, 명령어 수행에 따른 코드가 다 수행되면 V(&mutex)를 해준다. 이외에 client_num을 더하고 빼주는 부분에도 semaphore를 이용하여 구현하였다.

3. 구현 결과

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

(multiclient 기준)

여러 client가 server에 connect를 하게 되면 Rio_readlineb()를 이용해 client의 요청을 읽어 온다. 이후, client에 요청에 맞는 명령어를 수행하게 되는데, client의 요청의 종류는 3가지(exit 제외)가 있다. show를 했을 경우 현재 stock들의 정보를 table 형태로 client로 넘겨주어 출력한다. buy의 경우, buy 3 3을 client로부터 읽어오면 id가 3인 stock을 3개 사고 싶다는 것이므로, 해당 node를 찾아 left_stock값과 3을 비교해 결과값을 client로 보낸다. Sell의 경우도 buy와 비슷하게 돌아간다. 두 경우 모두 성공했다면 "[명령어] success"라는 문구가 client에 출력된다. Task_1과 2 모두 정해진 thread의 수만큼의 thread가 종료되면 client_num 값이 0이 되고 sigint_handler()가 호출되어 stock.txt를 현재 값으로 업데이트한 뒤 서버를 종료한다.

4. 성능 평가 결과 (Task 3)

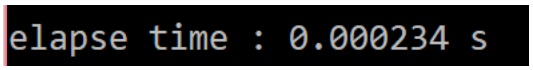
- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

Elapse time의 측정 시점은 multiclient.c 에서 client process를 fork하는 while문 시작 전부터 모든 client가 종료되고 waitpid() 반복문이 종료된 시점까지이다.

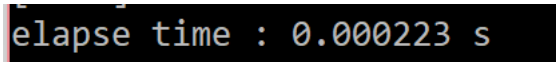
- **Client 개수 변화에 따른 동시 처리율 변화 분석**

  CLIENT_NUM : 4

  Event-driven

  Thread

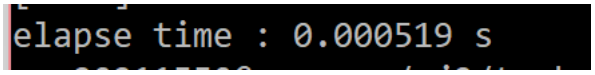
  

  CLIENT_NUM : 10

  Event-driven

  Thread

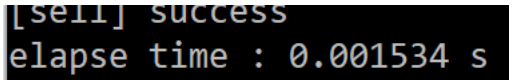
  

  CLIENT_NUM : 30

  Event-driven

  Thread

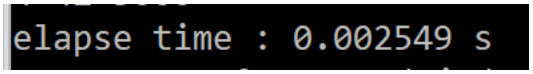
  

  CLIENT_NUM : 50

  Event-driven

  Thread

CLIENT_NUM : 70

Event-driven

```
elapse time : 0.003692 s
```

Thread

```
elapse time : 0.003587 s
```

CLIENT_NUM : 90

Event-driven

```
elapse time : 0.004696 s
```

Thread :

```
elapse time : 0.004742 s
```

CLIENT_NUM : 100

Event-driven

```
elapse time : 0.005258 s
```

Thread

```
elapse time : 0.005152 s
```

CLIENT_NUM : 200

Event-driven

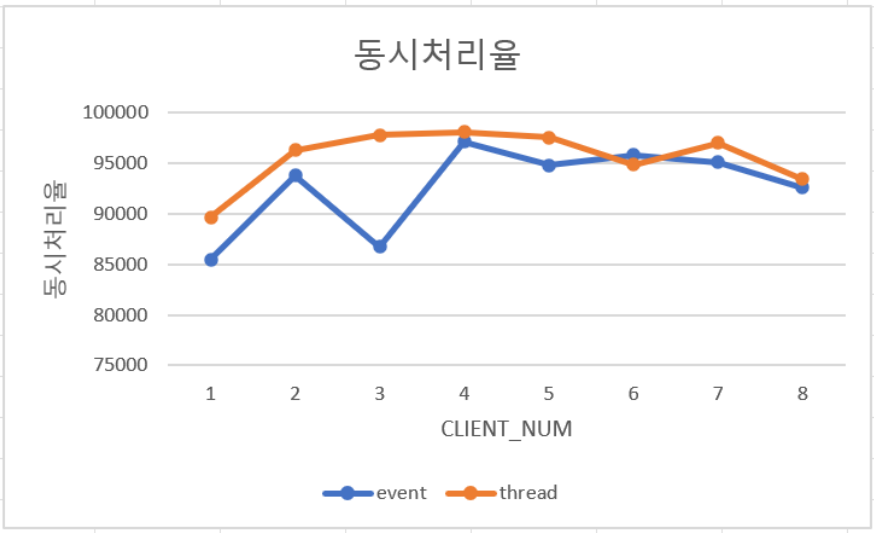
```
elapse time : 0.010797 s
```

Thread :

```
elapse time : 0.010705 s
```

위의 결과를 그래프로 정리하면 다음과 같다.

event	thread
85470.09	89686.1
93808.63	96339.11
86755.35	97783.57
97162.84	98077.68
94799.57	97574.57
95826.24	94896.67
95093.19	97049.69
92618.32	93414.29



동시 처리율 : CLIENT_NUM * ORDER_PER_CLIENT / elapse time

표의 가로축의 1번부터 차례로 THREAD_NUM이 4, 10, 30, 50, 70, 90, 100, 200일 때를 의미한다.

그래프를 보면, event-driven 방식보다 thread 방식의 동시처리율이 대체적으로 높은 것을 확인할 수 있다. 이는, event-driven 방식이 실제로는 concurrent하게 돌아가는 것이 아니라 하나씩 처리하는 방식이기 때문에 그렇다고 볼 수 있다. 두 방식 모두 client의 수가 늘어날수록 동시처리율 역시 증가함을 볼 수 있다. Thread 방식의 경우 실제로 concurrent하게 동작하기 때문에 client의 수가 증가할수록 동시 처리율 역시 증가할 것이다. Event-driven 방식의 경우, concurrent하게 돌아가지 않기 때문에 client의 수와 동시처리율 간의 비례 관계가 없을 것이라 예상하였다. 그런데, 가로축 2번과 3번의 결과를 보면 3번이 client 수는 더 높지만 2번보다 동시처리율이 더 낮은 것을 확인할 수 있다. 따라서 완벽하게 client 수와 동시처리율 간의 비례 관계를 설명하기는 어렵고, 좀 더 정확한 비례 관계를 확인하기 위해서는 여러 번의 실험을 통해 얻은 평균 값을 이용해야 할 것 같다.

- **워크로드에 따른 분석**

CLIENT_NUM*ORDER_PER_CLIENT : 10*5 로 고정

모든 client가 buy 또는 sell을 요청하는 경우

Event-driven

```
elapse time : 0.000522 s
```

Thread

```
elapse time : 0.000523 s
```

모든 client가 show만 요청하는 경우

Event-driven

```
elapse time : 0.000513 s
```

Thread

```
elapse time : 0.000513 s
```

모든 client가 buy, show 등을 섞어서 요청하는 경우

Event-driven

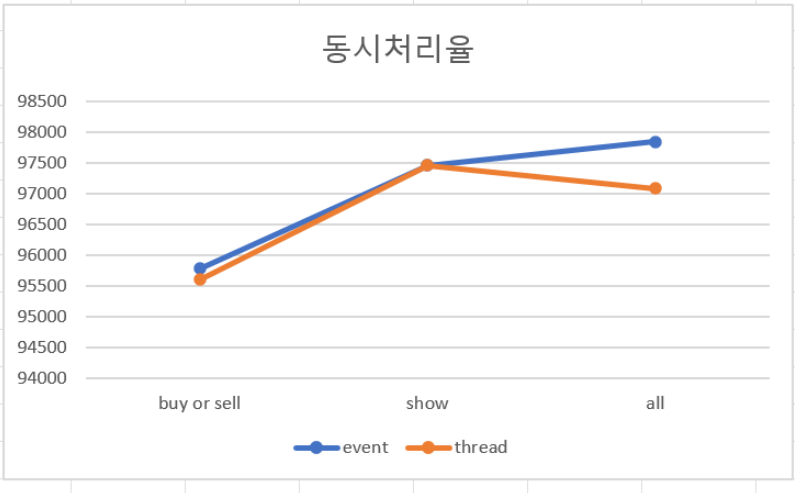
```
elapse time : 0.000511 s
```

Thread

```
elapse time : 0.000515 s
```

위 결과에 따른 동시처리율을 계산하면 다음과 같다.

	event	thread
buy or sel	95785.44	95602.29
show	97465.89	97465.89
all	97847.36	97087.38



위의 결과를 분석해 보겠다. 먼저, 시간 순으로 정렬하면 event-driven 방식은 모

든 요청을 섞어서 하는 경우(0.000511s) < show만 했을 경우(0.000513s) < buy 또는 sell만 했을 경우(0.000522s) 로 결과가 측정되었다. 먼저 구현한 코드 상에서는 show를 하는 경우에도, show() 함수 내에서 tree의 모든 node를 돌고, buy와 sell 역시 search() 함수를 통해 모든 tree를 보는 셈이기 때문에 3가지 경우 모두 시간 차이가 별로 나지 않을 것이라 예상하였다. 실제 결과에서도 별로 큰 차이가 나지 않았지만, show만 하는 경우가 buy 또는 sell만 하는 경우보다 시간이 더 짧은 것은 예상 밖이었다. 이를 분석해보면, 구현한 코드 상에서 show의 경우 재귀적으로 tree를 돌며 정보만 저장하면 끝이지만, search의 경우 root부터 while문을 통해 하나하나 id를 비교하는 구현 방식의 차이로 인해 이러한 결과가 나온 것이라 예측해본다. 또한 해당 실험에서는 STOCK_NUM의 개수가 5개밖에 되지 않았고 만약 STOCK_NUM의 개수를 훨씬 크게 하였다면 무조건 모든 tree를 돌아야 하는 show와 자신이 원하는 stock을 찾으면 search를 stop하는 buy와 sell만 하는 경우의 시간 차이를 정확히 확인할 수 있었을 것 같다. 동시처리율의 측면에서 보면, 한 두가지의 명령어만 실행하는 것보다 세 가지 명령어를 동시에 골고루 수행했을 때 동시처리율이 더 높게 나온 것을 볼 수 있다. 그러나, multiclient.c의 경우 랜덤으로 명령어를 수행하기 때문에 세 명령어를 모두 수행하는 실험에서 show, buy, sell의 비율이 일정하지 않다. 따라서 이 결과는 신뢰도가 낮다고 볼 수 있다.

Thread 방식의 경우에는, 모든 요청을 섞어서 하는 경우(0.000515s) < show만 했을 경우(0.000513s) < buy 또는 show만 했을 경우(0.000523s)로 event-driven 방식과 결과가 같은 것을 알 수 있다. 이 역시 buy 또는 sell만 했을 때의 시간이 제일 크게 나온 것은 의외의 결과였지만 event-driven 방식과 show(), search() 함수는 똑같기 때문에 역시 위와 동일한 이유로 이러한 결과가 나왔을 것이라 생각한다. 동시처리율을 보면 event-driven와 달리 all(세 명령어를 전부 수행하는 경우)일 때가 조금 낮게 나온 것을 볼 수 있는데, 이는 위에서 말한 것 처럼 랜덤으로 명령어를 수행하기 때문에 세 명령어의 수행 비율이 일정하지 않아 이러한 차이가 나온 것으로 보인다.

- **최종 분석 정리 & 이론과 비교**

먼저, client 개수에 따른 동시 처리율은 두 방식 모두 client 개수가 증가할수록 증가하는 추세를 보였다. Thread 방식의 경우, semaphore를 이용하였기 때문에 elapse time이 event-driven보다 길 것이라 예상하였는데, 이보다는 많은 worker thread를 이용하는 것의 이점이 더 커 대체적으로 event-driven보다 빠르게 실행

되는 것을 볼 수 있었다. 또한 thread 방식의 경우, nthread와 sbuf의 크기에 따라 client 개수가 너무 많아지면 작동이 중간에 멈추는 경우도 발생하였다. 그러나, event-driven 방식에서는 이와 같은 문제가 발생하지 않았다. 때문에 thread 방식보다 event-driven 방식이 예상치 못한 error 상황에 유연하게 대처한다고 볼 수 있다. 또한 워크로드에 따른 동시 처리율은, show만 했을 경우와 buy 또는 sell만 했을 경우에는 함수 구현의 차이에 따라 show만 했을 때의 동시처리율이 더 높게 나왔으나, 세 명령어를 모두 처리하는 경우에는 multiclient.c의 특성상 랜덤으로 명령어를 정하기 때문에 세 명령어의 비율이 일정하지 않아 정확한 결과를 비교하기 어렵다.

수업 시간에서는 event-driven 방식은 thread-base 방식보다 구현하기 어렵고, fine-grained concurrency를 제공하기 어렵다는 것을 알 수 있었다. 하지만 thread control overhead가 없기 때문에 좋은 퍼포먼스를 기대할 수 있다고 하였다. 그러나 위에서 분석한 실제 실험 결과를 보면 동시 처리율 측면에서는 오히려 thread 방식이 event-driven 보다 높은 것을 볼 수 있었다. 하지만, client 수가 많아질수록 두 방식 간의 동시 처리율 차이가 줄었기 때문에 더 많은 데이터를 분석해본다면 이론과 같은 결과를 기대해 봐도 좋을 것 같다.