Multicore Programming Project 2

담당 교수 : 최재승

이름 : 강상원

학번 : 20191559

1. 개발 목표

Event-driven approach / Thread-based apporoach로 concurrent 주식 서버를 구현한다. 여러 client가 동시에 접속하여 주식을 사거나(buy), 팔거나(sell), 정보를 조회(show)하는 동작을 수행할 수 있게끔 한다. 주식들에 대한 정보는 `stock.txt` 라는 별도의 파일에 저장되며, 주식 서버 실행 시 이 정보를 불러와서 실행한다. 동시성 제어를 통해 여러 클라이언트의 동시 정보 접근, 수정을 알맞게 제어한다. 수정된 주식 정보는 `stock.txt`에 다시 저장된다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   각 클라이언트마다 file descriptor를 관리하여, `Select()`으로 작업 요청을 선택해 처리한다. I/O Multiplexing으로 주식 buy, sell, show 작업을 수행한다. 서로 다른 클라이언트의 작업이 서로 충돌을 일으키지 않게 한다. 2번의 thread-based approach와 다르게 thread마다의 overhead 발생이 없으므로 클라이언트 수가 늘어남에 따라 비약적인 수행 시간의 변화가 이루어지진 않는다. Multi-core를 사용하지 못한다는 비효율성도 존재한다.

2. Task 2: Thread-based Approach

   일정 개수의 thread를 생성하여 1.과 마찬가지의 작업 (동시 여러 서버 접속하여 buy, sell, show 작업)을 Multithreading으로 수행한다. 미리 선언된 thread 개수에 따라 thread를 생성하는데, 접속한 클라이언트 개수 대비 thread의 개수 비에 따라 수행 시간의 차이가 크다. Thread-based approach를 수행하면 thread끼리 메모리 공유와 같은 개발 편의성이 있을 수 있지만, 이 때문에 race와 같은 여러 예상치 못한 수행결과의 발생 가능성이 증가한다.

3. Task 3: Performance Evaluation

   두 가지 수행 방식(Event-driven/Thread-based Approach)으로 구현한 Concurrent stock server를 비교하여 분석한다. 수행 시간, 동시 처리율 측면에서 효율성을 분석하고 클라이언트가 요청하는 타입(buy, show, sell)에 따라 차이도 본다. 클라이언트, thread 개수에 따른 수행시간, 두 방식의 퍼포먼스

를 그래프로 그려 비교 분석한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  현재 연결된 descriptor들을 다루는 pool 구조체를 정의하고, `select()` 반환 값을 이용하여 pending인 file descriptor를 `pool`의 `ready set`에 추가한다. 이후 `accept()`의 반환값 `connfd`를 이용해 클라리언트와 연결하고, 연결 정보 `connfd`를 `pool`에 추가한다. 반복문을 이용해 pending된 `connfd`를 탐색하고, 순차적으로 이를 처리한다. 이처럼 한 event 처리를 할 때는 다른 신호 입력을 처리하지 않고 `Select()` 함수를 사용해 어떤 file descriptor와 연결할지 결정한다.

  ✓ epoll과의 차이점 서술

  `select()`는 호출 시 descriptor의 I/O 정보를 전달하고 저장하는 과정을 거쳐야 하고, `FD_ISSET`으로 체크해야 하는 등 overhead가 많이 발생한다. 또한 반복문으로 순차적으로 file descriptor들을 탐색해 나가야 한다는 비효율성도 존재한다. 반면 `epoll()`은 OS 단에서 file descriptor를 직접 관리하며 함수 호출 시 정보 전달 등의 overhead가 적고 저장 공간도 직접 관리되어 편리하다.

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

  ✓ Master Thread의 Connection 관리

  Master(main) thread는 작동할 thread들을 `NTHREADS` 크기에 맞춰 선언해주고, 초기화한다. 반복문 내 `accept()`로 `connfd`를 받아오고, connection이 성립되면 `sbuf_inser()`로 `sbuf` (공유 버퍼; shared buffer)에 insert한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  공유 버퍼에서 해당 connfd를 받아오고, `EOF` 가 들어올 때까지 통신을 계속한다. `EOF` 시 `Pthread_detach()`를 이용하여 연결을 끊고 이후 다른 연결을 `sbuf_remove()`로 계속해서 받아와서 통신을 진행한다. Semaphore를 이

용하여 concurrent 실행 간 변수 중복 참조, 접근 등을 방지하여 예상치 못한 오류를 막는다.

- **Task3 (Performance Evaluation)**

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

  성능 평가를 함에 있어서 동시 처리율을 다음과 같이 정의하였다:

$$동시\ 처리율 = \frac{10^8 \times client\_num}{elapsed\_time}$$

  이 때 각 클라이언트마다의 작업 개수는 동일하게 유지하여 정량적 평가를 한다.

  먼저 각 방법(Event-based, Thread-based Approach)에 따른 동시 처리율 변화를 분석하고, 클라이언트 개수에 따라의 변화율을 그래프를 그려 한눈에 보기 쉽게 나타내고자 한다.

  또한 각 접근 방법 내, 클라이언트의 작업 요청 타입에 따른 처리율 차이도 분석한다. Buy, Sell 연산과 Show만을 요청했을 때 간의 유의미한 차이가 있는지를 보며 미리 선언된 thread 개수에 따른 동시 처리율 변화, 특히 클라이언트 개수가 증가함에 따라 양상이 어떻게 되는가를 보고자 한다.

  Event-based approach는 앞서 말했듯이 순차적으로 event 처리를 하는 것(즉, multicore를 활용하지 않음)이기 때문에, 작업 양에 따라 Thread-based approach와 어떤 유의미한 차이가 발생하는지를 확인해보고자 하고, Thread-based approach에 predefined thread 개수 또한 들어오는 클라이언트 접속 개수 대비 어떤 효율이 발생하는지를 관측하고자 한다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  Event-based approach는 `select()`를 이용해 `connfd`를 순차적으로 선택하고, 연산을 처리하기 때문에 실제로 concurrent하게 처리하는 Thread-based approach에 비해 처리율이 낮을 것으로 예상된다. 또한 Event based는 순차적이기 때문에 들어오는 클라이언트 개수에 따른 처리율 자체의 변화는 크게 있지 않을 것이라 보지만, Thread based는 실제 동시 처리이므로 처리율의 변화가 있을 것이라 본다. 또한 할당된 worker thread에 비해 들어오는 클라리언트 개수가 늘어나게 된다면 처리율의 효율성이 낮아질 것으로 예상한다.

  Thread-based approach에서 semaphore를 이용하여, 값을 수정하는 연산에선 동시 접근을 막기 때문에 (`P()`, `V()`) 단순히 값을 읽는 show 연산을 했을

때에 비해 처리율이 낮을 것으로 예상한다.

C. 개발 방법

- **Task 1(Event-based Approach):** `Select()`으로 pending file descriptor를 선택하여 클라이언트가 요청한 명령을 받아온다. (`Rio_readlineb()`) 각 연산 요청에 대한 처리는 buy(), sell(), show()로 구현한 함수로 처리한다. `Clientfd pool`을 관리할 `struct pool`을 선언하며 여기서 다루는 descriptor를 추가, 삭제하는 식으로 구현한다.
  (`struct pool` 안에는 ready descriptor, active한 descriptor들의 배열 등이 들어간다.)

  주식 정보는 이진 트리로 처리하는데, stockserver 실행 시 `load_data()`로 `stock.txt`에서 주식 정보를 받아오고 `insert_in_tree(ITEM **tree, int start, int end, ITEM *new_node)` 함수로 `tree[]` 배열에 이진 트리 형식으로 저장한다. (`mid=start+(end-start)/2`), 재귀 호출..

- **Task 2(Thread-based Approach):** 위 Event-based Approach와 마찬가지로 이진 트리를 통해 주식 정보 수정, 입력, 읽기를 처리한다.

  Master thread에서 활성 worker thread를 `NTHREADS` 크기에 맞게 생성해 주고 `sbuf`(shared buffer)에서 `connfd`를 받아내 connect하고 요청된 연산을 수행한다. Shared Buffer 각 정보를 다루는 `sbuf_t` 구조체는 buffer, 전체 크기, `mutex`와 같은 variable이 있다.) 한 동작 수행 중 특정 값 동시 수정과 같은 문제를 막기 위해 semaphore, mutex를 설정해 이를 방지한다.

3. 구현 결과

- Task 1: stockserver를 열고 여러 stockclient로 접속하면 stockserver에서 연결된 connfd에 차례로 접속해 연산을 받아오고 처리한다.

  `show` : 현재 주식의 상태를 보여준다.
  `buy 5 3` : "ID가 5인 주식을 3개 사겠다."

  `sell 2 2` : "ID가 2인 주식을 2개 팔겠다."
  tree[]로 이진 트리를 구현하고, 효율적인 노드 검색 및 접근을 가능케 한다.

- Task 2: Shared buffer(`sbuf`)에 `connfd`를 넣어 관리하고, 활성 worker thread에서

위와 같은 클라이언트 작업 요청을 받아 처리한다. show, buy, sell에서 주식 정보 트리 저장, 읽기, 수정을 할 때의 안전성을 위해 `mutex`를 사용한다.

4. 성능 평가 결과 (Task 3)

측정의 편의를 위해 multiclient에서 main routine이 시작되는 시점에서 시간 측정을 시작하고, exit될 때 측정을 종료하여 elapsed time을 측정한다. 명세서와 같이 `gettimeofday()` 함수를 사용한다.

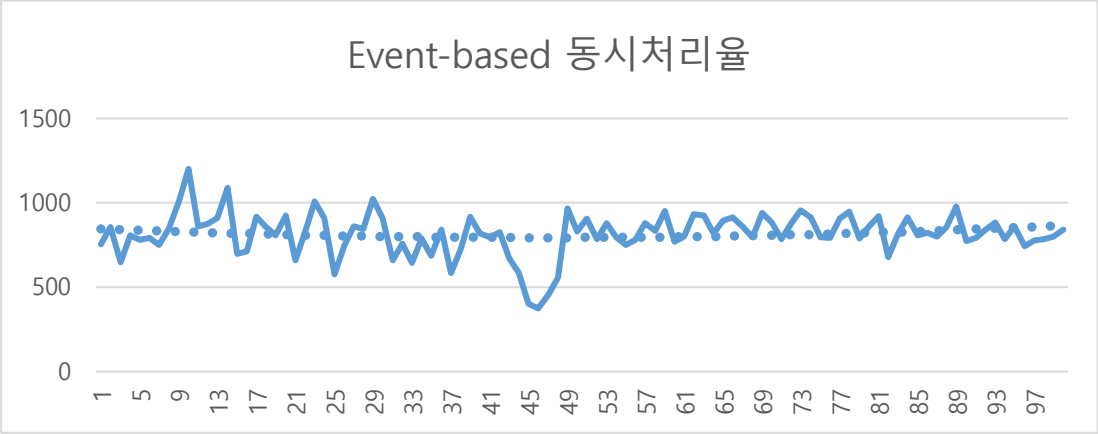
File: run.sh

```
./multiclient 172.30.10.9 60029 1 >> output.csv
./multiclient 172.30.10.9 60029 2 >> output.csv
./multiclient 172.30.10.9 60029 3 >> output.csv
./multiclient 172.30.10.9 60029 4 >> output.csv
./multiclient 172.30.10.9 60029 5 >> output.csv
./multiclient 172.30.10.9 60029 6 >> output.csv
./multiclient 172.30.10.9 60029 7 >> output.csv
./multiclient 172.30.10.9 60029 8 >> output.csv
./multiclient 172.30.10.9 60029 9 >> output.csv
./multiclient 172.30.10.9 60029 10 >> output.csv
./multiclient 172.30.10.9 60029 11 >> output.csv
./multiclient 172.30.10.9 60029 12 >> output.csv
```

```
cse20191559@cspro:~/echoserver$ cat output.csv
1, 132372, 7
2, 233368, 8
3, 463827, 6
4, 495949, 8
5, 639393, 7
6, 758540, 7
7, 930813, 7
8, 944519, 8
9, 891200, 10
10, 832323, 12
11, 1276988, 8
12, 1371803, 8
```

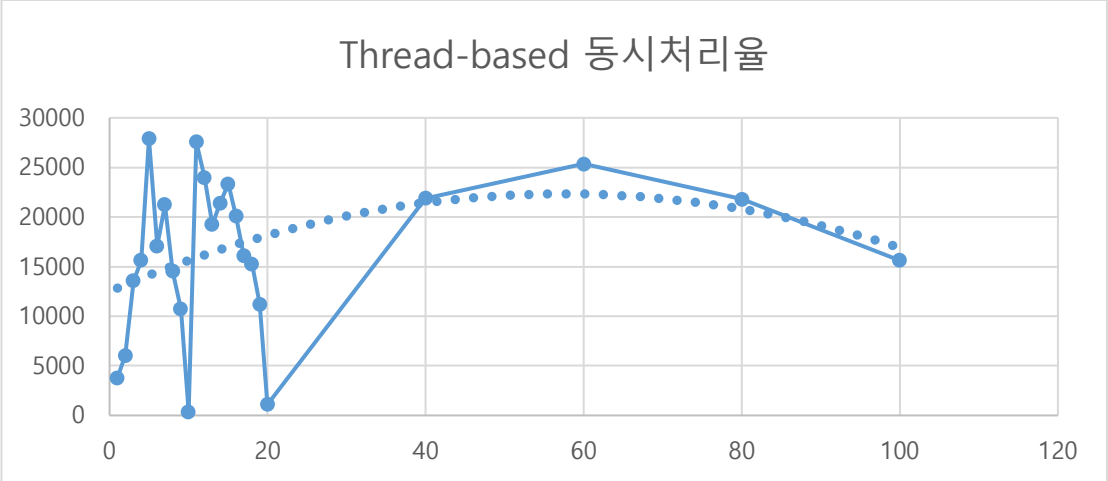
Shell script를 사용하여 클라이언트 개수에 따라 멀티클라이언트 출력값(클라이언트 개수, 시간)을 csv 파일에 순차적으로 출력되게 하여 측정한다. 동시처리율은 앞서 명시한 바와 같이 $\frac{10^8 \times client_num}{elapsed_time}$로 계산한다.

먼저 Event-based approach를 사용하였을 때의 클라이언트 개수에 따른 동시처리율 변화를 측정하면 다음과 같다.

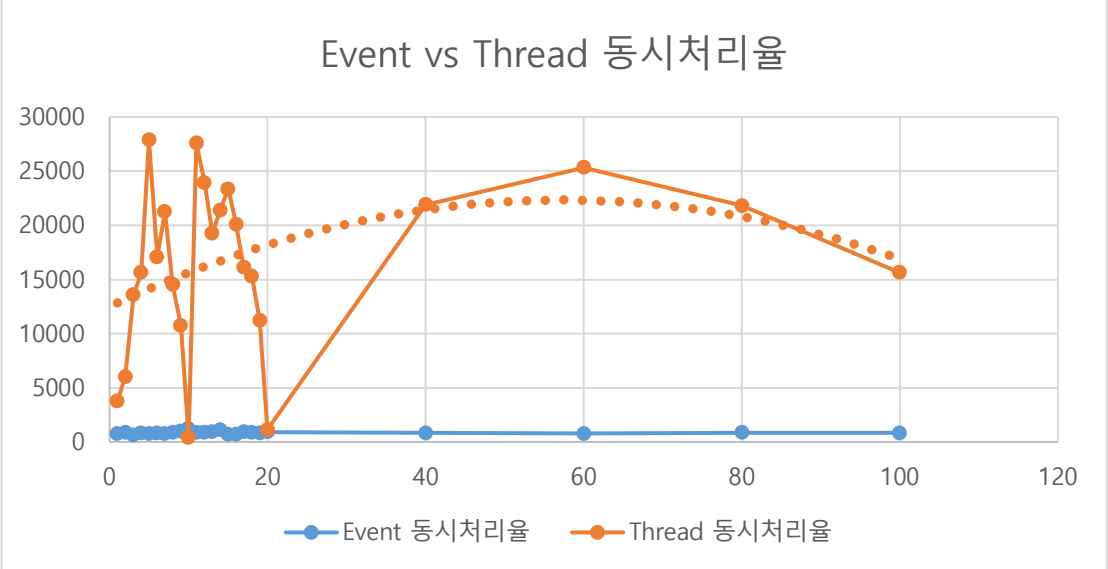


Event-based approach에서, 앞서 예상한 바와 같이 순차적이기 때문에 들어오는 클라이언트 개수에 따른 처리율 자체의 변화는 크게 있지 않았다.

다음은 Thread-based approach를 사용하였을 때의 클라이언트 개수에 따른 동시 처리율 변화를 측정하면 다음과 같다.

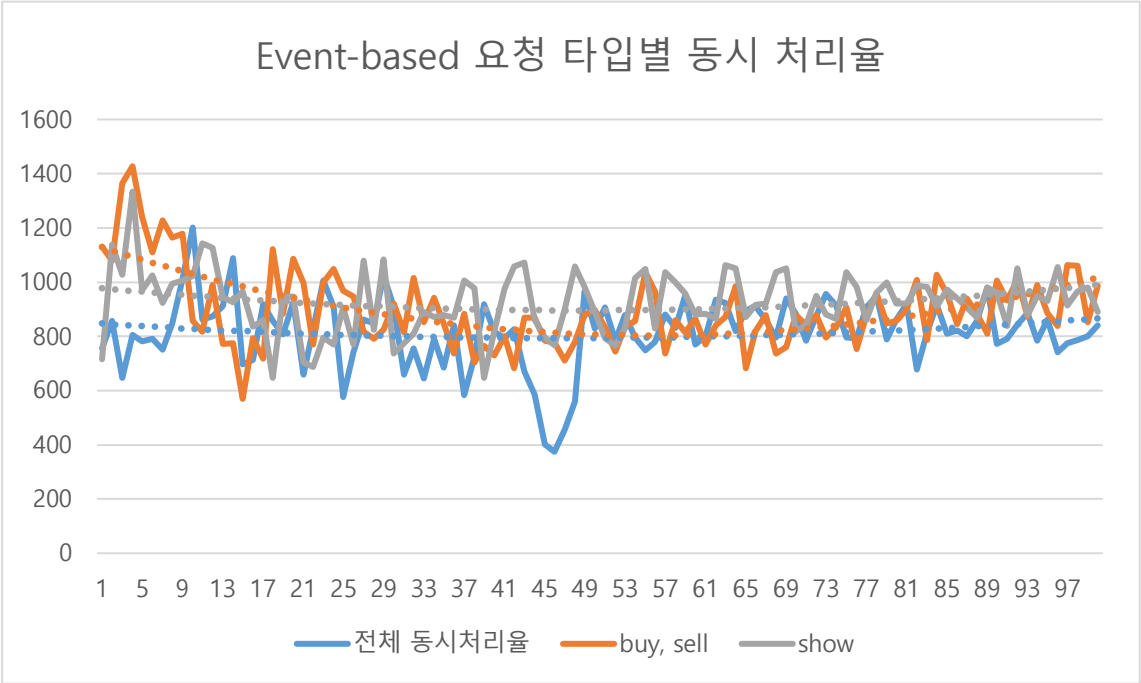


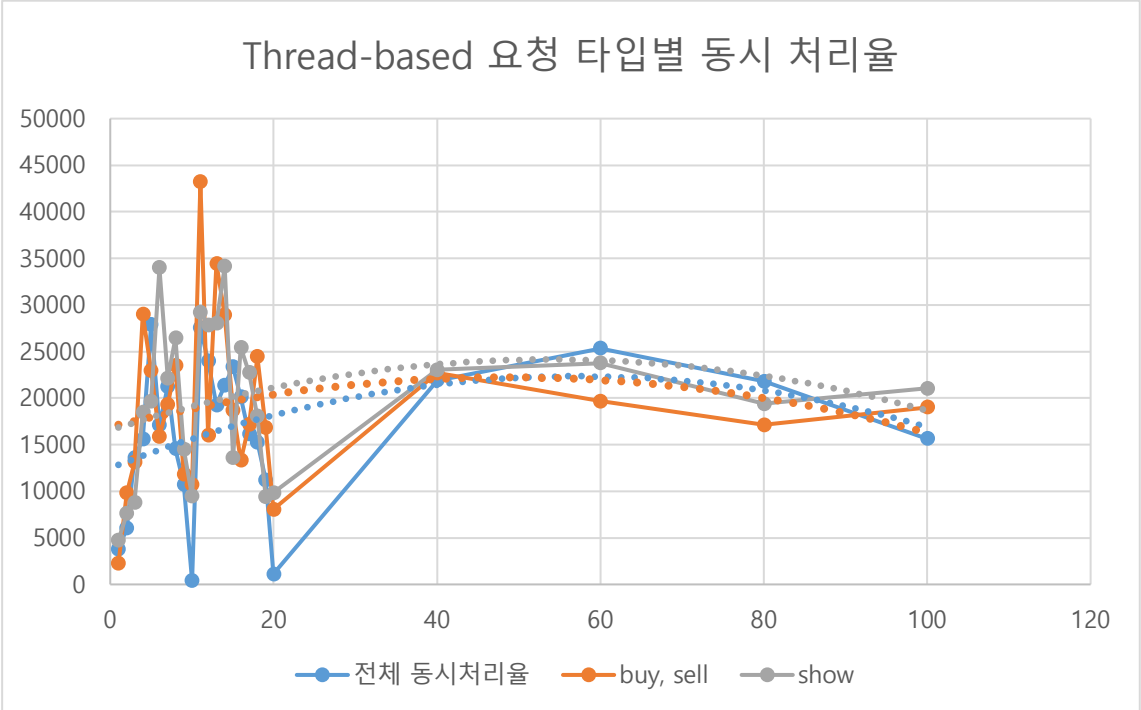
Thread based는 실제 동시 처리이므로 처리율의 변화가 있다. 또한 할당된 worker thread에 비해 들어오는 클라리언트 개수가 늘어나게 된다면 처리율의 효율성이 낮아지고 있다.
두 방식의 처리율을 분석하면, 다음과 같다.



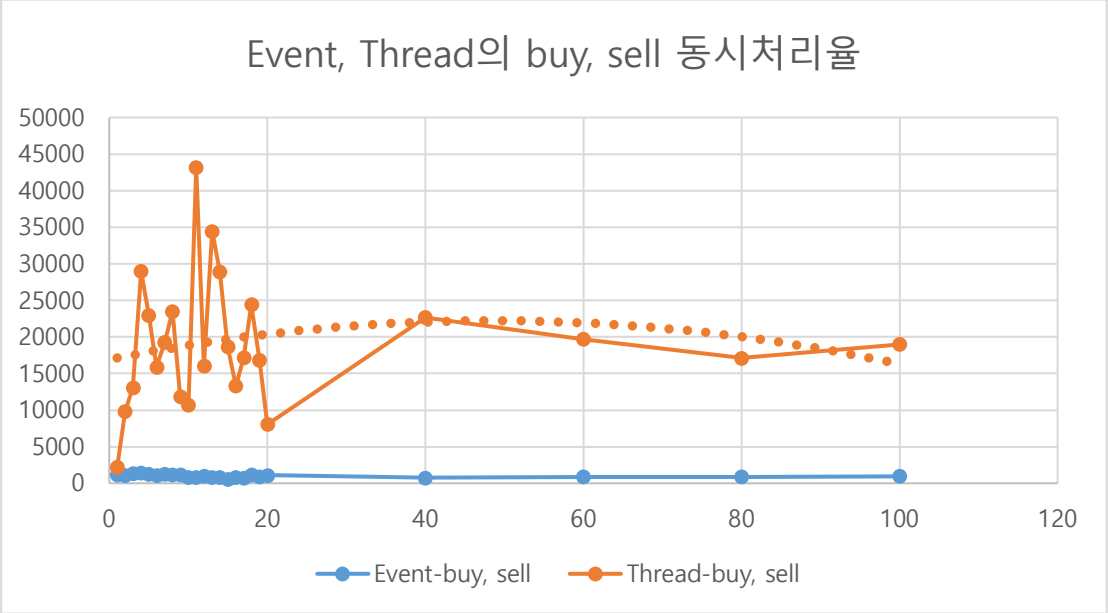
Thread-based approach에서는 멀티코어를 활용해 실제 concurrent하게 작업을 처리해 Evnet-based와 달리 동시처리율이 증가하는 모습을 볼 수 있다.

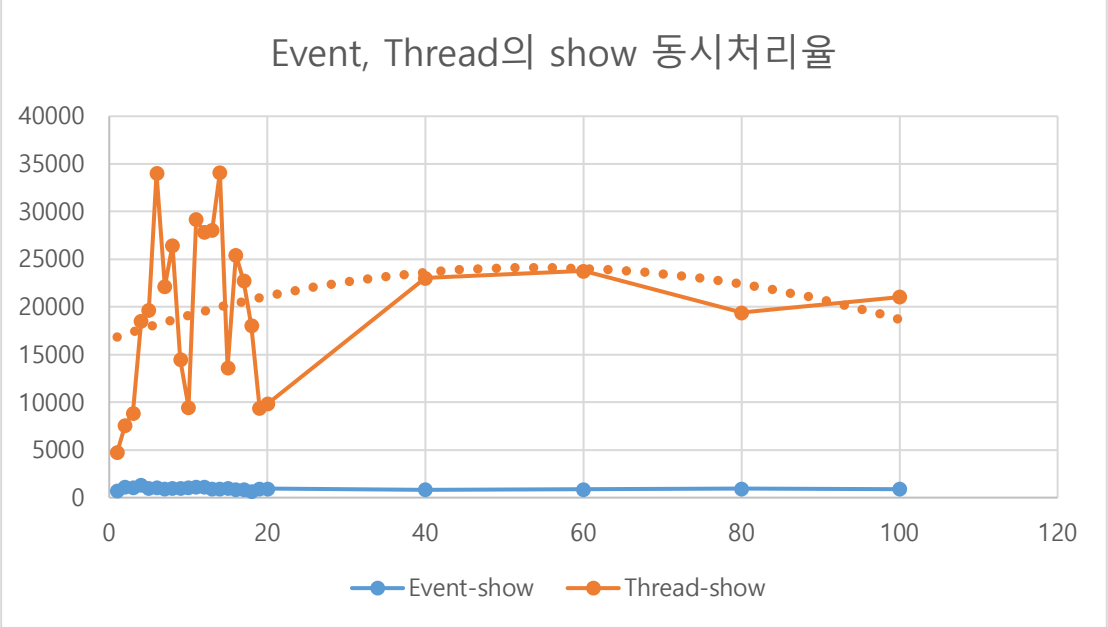
다음은 클라이언트 요청 타입에 따른 동시 처리율 차이이다.





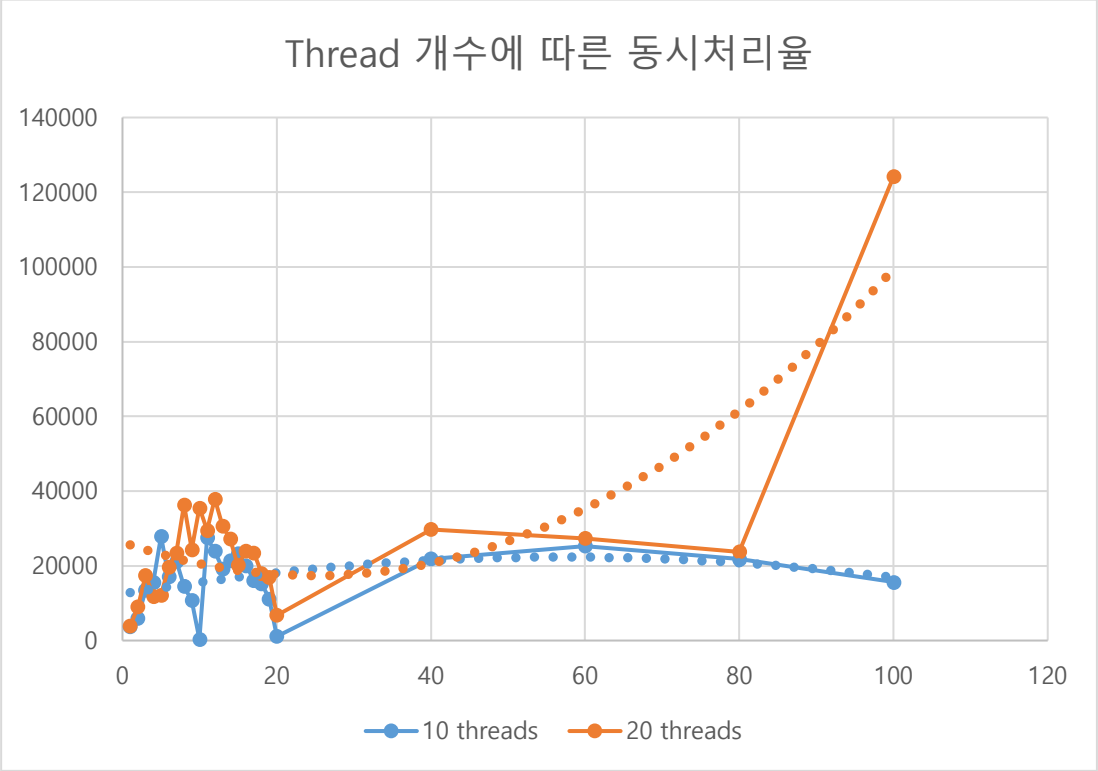
Buy, Sell / Show에 따른 처리율의 유의미한 차이는 보이지 않는다. 이는 stock.txt에 저장된 주식의 정보 개수가 수행 시간 차이를 일으킬 만큼 크지 않기 때문으로 보인다.





위와 같이 요청 타입에 따라 동시처리율을 보면, 둘 다 Event 방식은 처리율에 큰 차이가 없고, Thread 방식은 클라리언트 개수가 증가함에 따라 처리율이 증가하다 감소하는 양상을 보인다.

Thread-based approach일 때, thread 개수에 따른 동시 처리율 변화는 다음과 같다.



Thread의 개수가 증가하니 동시 처리율 또한 증가한 모습을 관측할 수 있다. Worker thread 대비 클라이언트 개수가 더 커질 때부터 동시처리율의 차이가 커지는 모습을 볼 수 있다.