Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 나호영

학번 : 20211531

1. 개발 목표

여러 client들의 동시 접속 및 서비스를 위한 Concurrent Stock Server를 Event-driven 방식과 Thread-based 방식을 이용해 설계하고 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   stock.txt의 주식 정보를 저장할 자료구조인 Binary Tree를 구현한다. 그 후 Event-driven 방식을 이용해 클라이언트들의 동시 접속과 서비스를 제공하기 위해 각각의 클라이언트들에게 배정될 파일 디스크립터들을 만들고 이 파일 디스크립터들을 select 함수를 이용해 모니터링하게 하여 접속과 show, buy, sell, exit 서비스를 제공하는 함수를 작성한다. 또한 주식 정보가 저장되어 있는 트리를 접근할 때 동기화 문제를 해결하기 위해 서비스 처리를 진행하는 함수에 Semaphore를 이용하여 read-write problem을 해결한다.

2. Task 2: Thread-based Approach

   마찬가지로 stock.txt의 주식 정보를 저장할 Binary Tree를 구현한 뒤 Thread-based 방식을 이용해 클라이언트들의 동시 접속과 서비스를 제공하게 한다. 이를 위해 각각의 클라이언트들에게 배정될 Thread들을 미리 생성해놓은 뒤 버퍼를 이용하여 접속을 확인해 서비스 요청들을 각각의 Thread 내에서 처리하게 한다. 또한 버퍼가 올바르게 작동하여 각각 Thread들이 겹치지 않게 sbuf Package를 이용한다. Task1과 마찬가지로 처리 함수 내부 서비스 동작 과정에서 데이터를 접근할 때 Semaphore를 이용한다.

3. Task 3: Performance Evaluation

   성능 평가를 위해 Client 파일 내부에서 configuration을 바꾸어 가면서 수행한다. Configuration은 MAX_CLIENT, ORDER_PER_CLIENT, STOCK_NUM, BUY_SELL_MAX 로 MAX_CLIENT와 STOCK_NUM, BUY_SELL_MAX는 실험을 위해 디폴트로 고정한 후 ORDER_PER_CLIENT를 변경하고 multiclient 프로그램을 실행할 때 클라이언트의 수를 적절히 변경하며 실험을 수행한다. 실험에서 측정해야할 것은 동시 처리율로써 처리량을 수행 시간으로 나누어 측정한다. 분석 포인트는 확장성, 워크로드에 따른 분석을 중점적으로 진행하고 Semaphore 사용 유무에 따른 차이와 자료 구조에 따른 차이를 바탕으로 추가적인 분석을 진행한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  기본적으로 Single Process에서 진행된다. 파일 디스크립터 배열을 이용하여 Select 함수로 Active Descriptor들을 모니터링하여 이벤트가 발생한 파일 디스크립터를 처리한다. 이

때, Select 함수는 이벤트가 발생한 파일 디스크립터에 해당하는 곳을 체크하여 out parameter set에 저장하므로 모니터링 하고있는 read_set이 변경되지 않도록 반복문 내에서 항상 read_set을 이벤트가 발생한 파일 디스크립터들을 의미하는 ready_set에 할당하도록 한다. Select 함수가 이벤트를 감지하여 보내준 ready_set에서 FD_ISSET으로 listenfd에서 발생한 이벤트이면 Accept를 이용해 접속을 처리하고 add_client를 이용하여 파일 디스크립터 배열을 초기화한다. 다른 Active 디스크립터에서 발생한 이벤트이면 check_client 함수가 호출되어 해당하는 요청을 처리한다. 이 때, single process이기 때문에 하나의 클라이언트에 대한 요청을 지속적으로 기다리는 문제를 방지해야 한다. 따라서 클라이언트로부터 발생한 요청을 반복문이 아닌 조건문으로 처리하여 한 줄의 요청만 받아서 처리한 후 다시 Select로 이벤트 발생을 모니터링하는 과정을 반복하여 Multi-client의 요청을 Concurrent하게 처리할 수 있다.

✓ epoll과의 차이점 서술

select와 epoll 모두 I/O Multiplexing을 지원하는 함수이다. 이번 과제에서는 select함수로 구현했기 때문에 epoll로 구현했을 때의 차이점을 위주로 설명하자면 대표적으로 select는 블로킹 방식이고 epoll은 기본적으로 논블로킹 방식이다.(epoll도 블로킹 방식을 지원하기는 한다.) 이로 인해 select는 이벤트가 발생할 때까지 프로세스를 블로킹 상태로 유지시키다가 이벤트 발생 후 반환되고 epoll은 이벤트가 발생하기 전에도 계속해서 프로세스를 실행시킬 수 있다. 또한 epoll은 이벤트가 발생한 파일 디스크립터만 리턴하기 때문에 많은 파일 디스크립터들을 감시하고 있을 때 select보다 효율적이지만 파일 디스크립터의 수가 많지 않을 때는 select가 더 효율적인 경우도 있다.

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

✓ Master Thread의 Connection 관리

마스터 스레드는 클라이언트 요청을 처리하기 위해 미리 생성해놓은 스레드들과 클라이언트들 사이에서 연결을 조율하는 역할을 수행한다. 마스터 스레드는 반복문에서 Accept 함수를 이용해 클라이언트들의 접속 요청을 받아서 버퍼에 해당하는 파일 디스크립터를 삽입하고 다시 접속 요청을 기다린다. 이 때 버퍼에 들어가있는 파일 디스크립터들을 worker 스레드들이 버퍼에서 제거한 후 해당 파일 디스크립터에 해당하는 클라이언트의 요청을 각각 독립적으로 처리한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

worker 스레드들은 클라이언트들의 요청을 처리하기 위하여 main에서 미리 생성되는데 이 때 Pool의 크기는 시스템의 자원과 작업의 효율성을 고려하여 NTHREADS만큼 생성된다. 처리해야 하는 작업들을 담고 있는 버퍼에 마스터 스레드가 파일 디스크립터를 넣고 worker 스레드가 버퍼에서 처리해야할 파일 디스크립터를 꺼내서 진행한다. 이를 해당 과제에서는 sbuf를 이용하여 구현하였다. worker 스레드가 클라이언트의 요청을 받아서 처리하다가 접속이 종료되면 Pthread_detach(pthread_self()) 로 스레드를 detach 상태로 변경시켰기 때문에 마스터 스레드가 join할 필요없이 독립적으로 자원을 해제시킨다. 또한 접속 종료 후에 다시 thread 함수로 돌아가 파일 디스크립터를 닫은 다음 다시 버퍼에서 파일 디스크립터를 가져온다. 이렇게 Worker Thread Pool 내부에서 워커 스레드들이 각각 독

립적으로 버퍼에서 꺼내와서 수행한 뒤 닫는 동작을 반복적으로 수행시킨다. Pool 내부에서 스레드들이 버퍼에서 꺼낼 때 같은 파일디스크립터를 꺼내는 오류는 sbuf 구조체를 활용하여 방지하고 각각의 작업에서 주식 데이터를 수정할 때 동기화 문제는 stock 구조체 내부에서 정의한 sem_t mutex와 w로 방지하는 방식으로 Pool 내부의 스레드들이 오류 없이 잘 수행되도록 관리된다.

- **Task3 (Performance Evaluation)**

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

얻어야 하는 가장 중요한 metric은 동시 처리율, 즉 시간당 요청 처리 개수이다. 이유는 concurrent programming을 활용해 여러 클라이언트들을 처리할 때 클라이언트의 수나 클라이언트들의 요청 개수의 증가에 따라 동시 처리율을 계산하면 어느 정도의 동시 처리 요청을 어느 지점에서 적절히 효율적으로 처리할 수 있는지 알 수 있고 이로 알게된 정보들을 참고하여 성능상의 개선을 위해 코드를 수정할 때 도움이 될 수 있기 때문이다. 측정 방법은 클라이언트의 수를 5, 10, 15, 20 으로 변경시키며 처리한 요청의 총 개수와 실행 소요 시간을 서버 측에서 측정하여 처리한 개수를 실행 소요 시간으로 나누어 동시 처리율을 계산하고 client process의 개수에 따라 비교할 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

Configuration은 MAX_CLIENT, ORDER_PER_CLIENT, STOCK_NUM, BUY_SELL_MAX 가 있다. 이 때, MAX_CLIENT와 BUY_SELL_MAX는 이 값들을 변경한다고 해서 그 자체로 성능 측정에 영향을 주지 않는다. STOCK_NUM은 stock.txt에 저장되어 있는 값을 기준으로 설정한다. ORDER_PER_CLIENT의 변경은 한 클라이언트 당 서버에 보내는 요청의 개수가 변하기 때문에 이를 변경하면서 수행할 것이다. ORDER_PER_CLIENT가 증가할수록 서버가 받는 요청의 개수가 많아지기 때문에 이를 처리하기 위한 시간이 늘어나고 따라서 동시처리율이 줄어들 것으로 예상된다.

C. 개발 방법

- 공통

stock.txt 파일을 읽어와서 데이터를 저장할 자료구조인 이진 트리를 구성할 node 구조체를 멤버로 stock, left, right를 두어 구현하고 해당 이진 트리를 활용하기 위해 삽입, 검색, 순회와 노드 정보들을 stock.txt 파일 초기화하는 함수, 모든 노드 가져오는 함수를 구현하였다.(insert, serachById, pre, writeFile, getAllNode)
주식 데이터를 저장하는 구조체 stock_item을 구현하였다. 멤버는 ID, 남은 주식을 의미하는 left_stock, 가격인 price와 read_write problem을 해결하기 위한 readcnt 변수와 sem_t mutex, w로 구성하였다.
main 함수가 종료되면 writeFile 함수가 호출되어 변경된 주식 데이터들을 stock.txt에 저장된다. 이 때, control + c 로 종료되는 경우에도 변경된 주식 데이터들을 저장해야 하기 때문에 sigint 시그널을 처리할 sighandler를 install하여 sighandler에서 writeFile을 호출하여 마찬가지로 주식 데이터들을 저장하도록 구현하였다.

- Task1

select 함수로 모니터링하는 파일 디스크립터 set을 효율적으로 구성하기 위해 pool 이라는 구조체를 구현하였다. 멤버로 maxfd, read_set, ready_set n_ready, maxi, clientfd 배열, clientrio 배열로 구성되어 있다. Open_listenfd 함수로 포트넘버를 할당하여 프로세스가 클라이언트 접속을 받을 준비가 되면 init_pool 함수를 이용하여 모든 멤버를 초기화하고 listenfd를 set에 넣는다. 그 후 반복문을 이용하여 Select 함수로 지속적으로 모니터링을 한다. 이벤트가 발생하면 Select 함수가 반환되고 listenfd에 이벤트가 발생하면 add_client 함수를 호출하여 모니터링할 파일 디스크립터 set에 넣은 후 접속을 처리한다. 다른 파일 디스크립터들의 이벤트 발생은 check_clients에서 처리된다. 이 함수에서 클라이언트로부터 들어온 요청 문자열을 파싱하기 위해 checkBuf 함수로 show, buy, sell, exit을 구분하여 option 변수로 반환한다. 그후 check_clients에서 조건문으로 각각의 요청을 구분하여 처리한다. buy, sell 요청은 주식의 id와 수량을 따로 파싱해야하므로 strtok를 이용하여 저장한다. show 함수가 호출될 때, buy, sell로 인한 데이터를 변경을 방지하기 위하여 stock_item의 멤버 readcnt, mutex, w를 이용해 readcnt가 0일 때만 buy, sell 이 수행되도록 하고 buy, sell도 동시에 데이터를 변경하면 안되므로 w를 이용해 한번에 하나씩만 접근하도록 구현하였다.

- Task2

마스터 스레드와 워커 스레드 사이에서 처리해야 하는 파일 디스크립터들을 저장하고 있는 버퍼를 만들기 위해 sbuf 구조체를 정의하였다. 이 구조체는 버퍼 그 자체 뿐만 아니라 mutex, slots, items sem_t 변수를 멤버로 가지고 있어 워커 스레드들이 동시에 하나의 파일 디스크립터를 remove 시키는 오류를 방지할 수 있다. sbuf를 다루기 위해 subf_init, sbuf_deinit, sbuf_insert, sbuf_remove 함수를 정의하였다. 메인 함수에서는 미리 Pthread_create를 이용하여 워커 스레드들을 생성해놓은 뒤 마스터 스레드가 Accept 함수를 이용해 접속을 받으면 sbuf_insert를 호출하여 버퍼에 처리할 파일 디스크립터들을 담는다. thread함수는 Pthread_detach(pthread_self())를 호출하여 마스터 스레드가 join을 하지 않아도 알아서 자원을 해제하도록 만들어 주고 무한 루프를 내부에서 버퍼에서 파일 디스크립터를 remove한 뒤 handleStocks를 호출한다. handleStocks 함수는 show,

buy, sell, exit와 같은 주요한 클라이언트의 요청을 처리하는 control 함수이다. handleStocks 함수에서는 Task1과 마찬가지로 checkBuf를 이용하여 요청을 파싱한 뒤 조건문으로 해당 요청을 처리한다. Task1과 다른 점은 Task1은 Rio_readlineb를 if문을 이용해 한 줄만 처리하는 반면에 Task2에서는 while을 이용하여 계속해서 클라이언트의 요청을 해당 스레드가 받아서 처리할 수 있도록 구현하였다. 접속이 종료되면 thread함수로 돌아가 파일 디스크립터를 닫는데 무한 루프 내부에 있으므로 다시 버퍼에서 작업을 가져와서 처리하는 동작을 수행한다.

3. 구현 결과

주식 서버 프로그램을 포트 넘버를 할당하여 실행하면 서버는 클라이언트의 요청을 기다릴 수 있게 된다. 클라이언트 프로그램을 주식 서버의 IP 주소와 포트 넘버를 할당하여 실행하면 주식 서버 프로그램에 연결된다. 클라이언트 프로그램으로부터 show 를 입력하면 저장되어 있는 주식 정보들을 보여준다. buy #1 #2 를 입력하면 #1 id를 가진 주식 #2개를 사는 동작을 수행하는데 수량이 부족하면 메세지를 출력하고 성공적으로 수행되면 해당 주식 데이터 수량을 변경한 후 매수완료 메세지를 출력한다. sell #1 #2 를 입력하면 #1 id를 가진 주식 #2개를 파는 동작을 수행하는데 성공적으로 수행되면 해당 주식 데이터 수량을 변경한 후 매도완료 메세지를 출력한다. exit 을 입력하면 연결 종료 메세지를 출력한 뒤, 서버로부터 연결이 해제된다. 서버 프로그램이 종료될 때는 변경된 주식 데이터를 stock.txt에 업데이트한다.
위와 같은 동작을 여러 클라이언트들이 동시에 수행되더라도 event-driven 방식과 Thread-based 방식으로 Concurret하게 작동되도록 구현하였기 때문에 정상적으로 작동된다.

4. 성능 평가 결과 (Task 3)

(공통)

시작 시간은 첫번 째로 접속 요청을 한 클라이언트의 접속 요청을 처리할 때로 설정하고 종료 시간은 마지막으로 서버로부터 접속 종료를 요청한 클라이언트의 요청을 처리한 후로 설정하였다. clock() 함수를 활용하여 ms 단위로 측정할 수 있게 하였고 종료 시간 변수에서 시작 시작 변수를 빼서 실행 소요 시간을 측정하고 처리한 요청 수를 소요 시간으로 나누어 동시처리율을 계산하게 하였다. 위에서 설명한 것처럼 개념 상 시작, 종료 시간 측정 방식은 같으나 코드 상에서 check_clients 내부에서 마지막 종료 요청 뒤 handling 한 후에 end = clock()을 호출한 Task1과 달리 Task2는 Thread 기반이기 때문에 마지막으로 연결 종료된 클라이언트를 담당하는 쓰레드가 thread 함수 내부에서 handleStocks가 호출되고 리턴된 후로 end = clock()을 호출하였다.

- 측정 시작 시 출력

```
측정 시작
연결된 클라이언트 : 1
thread -1085401344 received 9 (9 total) bytes on fd 4
```

- 측정 종료 시 출력

```
모든 연결 종료
측정 완료
```

(각 결과에 대한 그래프는 분석 관점에 따른 Task1과 Task2의 차이를 비교하기 위해 Task2의 해석 파트에 넣었습니다.)

(Task1)

- 확장성 관점의 분석 포인트

접속 연결과 접속 종료에 대한 요청 처리는 제외하고 show, buy, sell, exit 과 같은 클라이언트로부터 받는 서비스 요청에 대한 처리만 개수를 카운트해서 동시처리율을 계산하였다. 동시처리율은 1ms당 처리 개수로 unit을 설정하였다.

client의 개수는 순서대로 각각 5, 10, 15, 20
각 클라이언트의 주문 개수는 10로 고정

```
------ 결과 ------
연결된 클라이언트 : 5
처리된 요청 개수 : 50
실행 소요 시간 : 3005.000000 ms
동시처리율(processing per second) : 0.016639 p/ms

------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 6250.000000 ms
동시처리율(processing per second) : 0.016000 p/ms


------ 결과 ------
연결된 클라이언트 : 15
처리된 요청 개수 : 150
실행 소요 시간 : 9347.000000 ms
동시처리율(processing per second) : 0.016048 p/ms


------ 결과 ------
연결된 클라이언트 : 20
처리된 요청 개수 : 200
실행 소요 시간 : 13208.000000 ms
동시처리율(processing per second) : 0.015142 p/ms
```

위의 측정 결과를 살펴보면 클라이언트의 개수가 증가함에 따라 실행 소요 시간이 선형으로 증가하는 것을 확인할 수 있다. 동시처리율은 클라이언트의 개수가 증가함에 따라 감소하는 경향을 보이기는 하지만 클라이언트의 개수가 10일 때와 15일 때를 보면 오히려 증가했음을 확인할 수 있다. 이 측정 결과에서는 클라이언트의 개수가 적기 때문에 클라이언트의 개수에 따른 동시처리율이 완벽한 우하향을 그리지는 않았다. (클라이언트의 개수가 100개, 1000개와 같이 크게 증가한다면 동시처리율은 낮아질 것이라고 예측된다.)

- 워크로드 관점의 분석 포인트

클라이언트의 수는 10으로 고정하고 각 클라이언트의 주문 요청 개수는 10으로 고정하여 실험을 진행하였다. 다음과 같은 세가지 경우를 같은 환경에서 비교 분석을 실시하였다.

1. 모든 클라이언트가 show 요청만 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 6762.000000 ms
동시처리율(processing per second) : 0.014789 p/ms
```

2. 모든 클라이언트가 buy/sell 요청만 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 5859.000000 ms
동시처리율(processing per second) : 0.017068 p/ms
```

3. 모든 클라이언트가 show/buy/sell 요청을 섞어서 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 6134.000000 ms
동시처리율(processing per second) : 0.016303 p/ms
```

위의 측정 결과를 살펴보면 show 요청만 한 첫번 째 경우가 제일 실행 소요 시간이 크고 동시처리율이 낮다. 그 다음으로 show/buy/sell을 섞어서 요청한 세번 째 경우가 실행 소요시간이 크고 동시처리율이 낮고 buy/sell 요청만 한 두번 째 경우는 세가지 경우에서 제일 실행 소요 시간이 적고 동시처리율이 크다. 측정 결과 예상에서는 write를 진행해야할 buy/sell 경우가 가장 동시 처리율이 낮을 것이라고 예측했다. 하지만 실제로는 read를 진행하는 show를 요청 받을 때 클라이언트에게 주식 결과 정보를 보여줄 때 , buy/sell 과 같은 write 과정을 처리하기 위해 read_cnt에 관련된 mutex 세마포어, write 시작을 위한 w 세마포어 두가지 모두 작동하기 때문에 show 요청만 받을 때 가장 동시 처리율이 낮게 나왔다고 볼 수 있다.

(Task2)

- 확장성 관점의 분석 포인트

Task 1과 마찬가지로 접속 연결과 접속 종료에 대한 요청 처리는 제외하고 show, buy, sell, exit 과 같은 클라이언트로부터 받는 서비스 요청에 대한 처리만 개수를 카운트해서 동시처리율을 계산하였다. 동시처리율은 1ms당 처리 개수로 unit을 설정하였다.

client의 개수는 순서대로 각각 5, 10, 15, 20
각 클라이언트의 주문 개수는 10로 고정

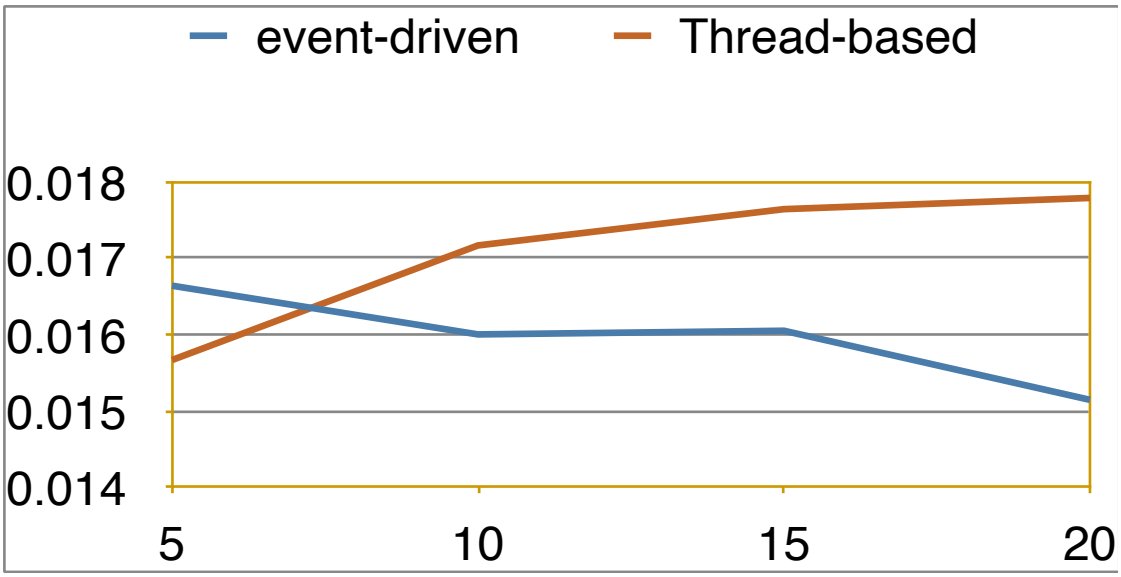
```
------ 결과 ------
연결된 클라이언트 : 5
처리된 요청 개수 : 50
실행 소요 시간 : 3192.000000 ms
동시처리율(processing per second) : 0.015664 p/ms

------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 5825.000000 ms
동시처리율(processing per second) : 0.017167 p/ms

------ 결과 ------
연결된 클라이언트 : 15
처리된 요청 개수 : 150
실행 소요 시간 : 8502.000000 ms
동시처리율(processing per second) : 0.017643 p/ms

------ 결과 ------
연결된 클라이언트 : 20
처리된 요청 개수 : 200
실행 소요 시간 : 11242.000000 ms
동시처리율(processing per second) : 0.017790 p/ms
```

클라이언트의 수가 증가하면 실행 소요 시간이 증가하는 것을 확인할 수 있다. 동시처리율은 클라이언트의 수가 증가할수록 증가하는 것을 볼 수 있는데 이는 Task1과의 차이를 나타낸다. 싱글 스레드로 event-driven 방식으로 처리하는 Task1과 달리 Task2는 미리 워크 스레드를 생성한 뒤에 각각의 클라이언트의 요청을 지속적으로 각각의 워크 스레드들이 처리하기 때문에 요청이 증가할수록 오히려 동시처리율이 올라가는 것이라고 해석할 수 있다. 이를 그래프로 나타내면 아래와 같다.



- 워크로드 관점의 분석 포인트

마찬가지로 클라이언트의 수는 10으로 고정하고 각 클라이언트의 주문 요청 개수는 10으로 고정하여 실험을 진행하였다. 다음과 같은 세가지 경우를 같은 환경에서 비교 분석을 실시하였다.

1. 모든 클라이언트가 show 요청만 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 6078.000000 ms
동시처리율(processing per second) : 0.016453 p/ms
```

2. 모든 클라이언트가 buy/sell 요청만 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 5724.000000 ms
동시처리율(processing per second) : 0.017470 p/ms
```

3. 모든 클라이언트가 show/buy/sell 요청을 섞어서 한 경우

```
------ 결과 ------
연결된 클라이언트 : 10
처리된 요청 개수 : 100
실행 소요 시간 : 5887.000000 ms
동시처리율(processing per second) : 0.016987 p/ms
```

측정 결과, Task1의 워크로드에 따른 분석과 마찬가지로 show 요청만 한 첫번 째 경우가 제일 실행 소요 시간이 크고 동시처리율이 낮다. 그 다음으로 show/buy/sell을 섞어서 요청한 세번 째 경우가 실행 소요시간이 크고 동시처리율이 낮고 buy/sell 요청만 한 두번 째 경우는 세가지 경우에서 제일 실행 소요 시간이 적고 동시처리율이 크다. 이에 대한 이유도 Task1과 같은데 read를 진행하는 show를 요청으로 클라이언트에게 주식 결과 정보를 보여줄 때, buy/sell write 과정을 처리하기 위해 read_cnt에 관련된 mutex 세마포어, write 시작을 위한 w 세마포어 두가지 모두 작동하기 때문에 show 요청만 받을 때 가장 동시 처리율이 낮게 나왔다고 해석할 수 있다. Task1과 Task2의 각 요청의 종류에 따른 경향은 같으나 Thread-based 방식이 동시처리율이 더 높게 나왔는데 이는 single thread 기반인 event-driven 방식보다 각 쓰레드들이 클라이언트의 요청을 지속적으로, 독립적으로 처리하는 Thread-based 방식이 좀 더 효율적으로 요청을 처리한다고 볼 수 있다. 이에 대한 그래프는 아래와 같다.