System Programming Project 3

담당 교수 : 박성용

이름 : 이도안

학번 : 20201610

1. 개발 목표

해당 프로젝트에서는 client에게 요청받아 stock을 sell, buy, show 할 수 있는 서버를 구현하는 것이 목표이다. Event-driven 과 Thread-based Approach를 모두 구현하고, client의 요청 사항이 Producer-Consumer, Reader-Writer problem등에 제한되지 않도록 해야 한다.

Task1에서는 select 함수를 활용한 event-driven approach 방식으로 구현했으며, Task2에서는 pthread-library를 사용해 thread-based approach 로 구현한다. 마지막으로, Task3에서는 각 구현 방식에 따른 성능을 비교하고 평가한다.

주식 정보는 이진 트리를 활용해 접근이 효율적이게 관리되며 show, buy, sell 모두 이진 트리를 활용하여 접근할 수 있다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

Event-driven 으로 서버를 구현할 경우, I/O 멀티플렉싱을 사용한다. select함수를 사용해 다중 클라이언트의 요청을 동시 처리하는 것 처럼 보이는 서버를 구현한다. 서버에 요청이 도착할 때 마다 한 줄 씩 텍스트를 처리할 수 있도록 구현한다.

select함수는 다중 클라이언트의 입력인 파일 디스크립터를 모니터링하게 되며, 클라이언트의 정보는 pool형태로 저장되어 관리한다. 단일 스레드로 요청을 처리한다.

2. Task 2: Thread-based Approach

Thread-based로 서버를 구현할 경우, 각 클라이언트 연결에 대한 각각의 스레드를 생성하도록 한다. Pthread를 사용하여 master thread가 parent처럼 클라이언트의 연결 요청을 처리하고, worker thread가 각 클라이언트의 요청을 수행하도록 구현하였다. Task1과 비슷하게 동시 처리하지만, 멀티 쓰레드인 만큼 실제로 동시 처리 효과

가 발생하도록 한다. Semaphore를 적절히 활용하여 여러가지 problem을 제어한다.

3. Task 3: Performance Evaluation

여러 분석 포인트를 설정하여 Task1과 Task2 각각의 방법에 대해 성능을 분석하고 평가한다. 여기서 gettimeofday를 multiclient에 임의로 설정하여 시간을 측정하고 클라이언트들의 요청이 끝나면 시간을 출력하게 한다. thread수와 client수, 워크로드를 조절해가며 concurrent or parallel한 주식 서버 구현이 잘 되었는지 확인하고 더 잘 구현할 수 있는 방법을 고안한다.

B. 개발 내용

-

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  select함수를 이용해 I/O Multiplexing 를 처리한다.

  서버 초기화로 listenfd를 열어두고, pool을 초기화한다. select함수를 활용해 여러 소켓을 감시하며 이벤트가 발생하면 반환해준다. 이벤트 발생 시, FD_ISSET을 통해 확인하고 Accept함수를 호출하여 연결을 진행한다. 클라이언트의 요청으로 인한 데이터를 처리하면서 다른 클라이언트의 요청도 처리하게 된다. 클라이언트가 연결을 종료하면, 소켓을 닫고 배열에서 삭제시킨다.

  ✓ epoll과의 차이점 서술

  epoll은 select함수가 파일 디스크립터를 매번 확인하는 반면, 파일 디스크립터의 상태변화가 있는지만을 확인한다. 따라서, 파일 디스크립터의 수가 많아질수록 select의 성능은 epoll보다 떨어질 수 있다. 현 프로젝트의 경우 파일 디스크립터의 수가 방대하지 않으므로 별 차이가 없을 것으로 보이지만, 결론적으로 epoll은 리눅스 환경에서 select보다 성능적으로 유리할 수 있다.

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

  ✓ Master Thread의 Connection 관리

Master Thread는 최상위 Thread로서 클라이언트의 연결 요청을 관리하고, 요청이 들어온 경우 Accept로 수락하고 worker thread로 클라이언트의 요청을 처리한다.

Worker thread pool을 생성하고, 각 worker thread가 클라이언트들의 요청을 동시에 처리하게 된다. 클라이언트의 연결이 종료되면 Close()함수를 사용하여 connfd를 메모리 해제하고 할당되었던 Worker Thread는 pool로 반환된다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread pool은 클라이언트의 요청을 처리하는 각각의 Worker Thread의 배열이다. Thread는 독립적으로 요청을 처리하는데, 선언한 MAXTHREAD의 수만큼 Thread가 생성되고, sbuf 패키지를 활용하여 관리한다. Sbuf는 여러 스레드가 동시에 sbuf에 접근하는 동시성 문제를 방지한다.

클라이언트의 연결이 종료되면 Close()함수를 사용하여 connfd를 메모리 해제하고 할당되었던 Worker Thread는 pool로 반환된다.

- **Task3 (Performance Evaluation)**

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

분석 방법으로 동시 처리율 비교를 사용한다. 시간당 client 요청 처리 개수를 평가할 수 있도록 metric으로 요청 client 수와 그에 따른 전체 처리량, Thread수, 그리고 총 처리 시간을 사용한다. 클라이언트당 요청 수는 10으로 고정하고, 클라이언트의 개수를 조절하여 전체 처리량을 조절하게 하였다. 특히 Task2는 Task1에 비해 동시성과 병렬성이 모두 보장되는데, 정말 그런지 확인하기 위해 Thread 수와 전체 클라이언트 수를 비교해가며 분석하기 위해 둘을 metric으로 설정하였다.

```
cse20201610@cspro:~/project3-task2$ grep -c processor /proc/cpuinfo
20
cse20201610@cspro:~/project3-task2$
```

Ssh 서버의 cpu 코어 개수가 20개이므로, 20개의 Thread 수를 기준으로 줄이고 늘려가며 수행 시간을 테스트하였다.

✓ Configuration 변화에 따른 예상 결과 서술

Task1 은 단일 Thread가 모든 클라이언트의 요청을 처리하기 때문에 동시성은 갖지만 병렬성은 갖지 못한다. 반면 Task2는 여러 개의 Worker Thread가 각각 하나씩 클라이언트의 요청을 처리하기 때문에 동시성과 병렬성을 모두 가진다. 따라서 이는 CPU의 여러 코어를 동시에 활용할 수 있다는 의미가 된다.

따라서, Event-Based 보다 Thread-Based의 성능이 더 향상되어 보일 것으로 예상된다. 또한 semaphore를 처리하는 부분에 있어서도, 단일 클라이언트를 처리하는 경우에 semaphore의 lock, unlock을 처리하는 부분의 시간은 거의 무시할 정도이기 때문에 Thread-Based의 성능이 더 우수할 것으로 보인다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

Task 1, 2 공통

일단, 주식 클라이언트의 요청에 따라 stock server가 주식을 처리할 수 있도록 각 주식의 정보를담은 item structure와, 그것을 멤버로 가지는 tree node를 구현하였다. 각 node는 binary tree를 구성하며, 서버 내부에서 주식 정보를 저장하는 전역 메모리로서 사용된다.

Load_stock함수는 서버 실행 시 한번 실행되며, stock.txt를 한번 읽어와 binary tree에 저장한다. 또한 sell, buy 함수를 통해 입력받은 수행문을 정상적으로 수행할 수 있도록 한다. 각 함수는 binary tree search를 통해 해당하는 주식 번호에 맞는 노드를 찾고, 그 노드의 정보를 업데이트한다. Print_stock 함수는 파라미터로 전달받은 버퍼에 현재 바이너리 트리에 저장된 stock 정보를 옮기고, save_stock은 stock.txt에 현재 트리에 저장된 정보를 입력한 후 리턴한다.

Task 1

또한 클라이언트의 정보를 저장하고 관리하는 pool 구조체를 만들었다. Read, ready 디스크립터를 생성하고 최대 디스크립터 수, 준비된 디스크립터 수, 디스크립터를 저장하는 배열과 buffer를 읽어오는 rio_t 배열을 추가하였다.

강의자료에 따라서 init_pool로 pool을 초기화하고 select 함수를 호출할 때 앞서 말한 디스크립터 ready를 read_set으로 할당한다. 그리고 maxfd+1을 전달하여 그 수만큼 디스크립터의 수의 확인하고 감시될 파일 디스크립터인 ready_set을 인자로 전달한다. 이 때 나머지 인자는 NULL값을 주게 되는데, select함수는 파일 디스크립터가 있을 때까지 대기하고, 있으면 그 수만큼 nready에 할당한다.

FD_ISSET함수를 통해 listenfd 파일 디스크립터가 ready_set에 있는 지 확인하고, 있다면 Accept를 통해 연결을 해주고 새로운 파일 디스크립터 connfd에 반환한다. 해당 클라이언트를 pool에 추가해주기 위해 add_client 함수를 호출한다.

Check_clients 함수를 호출하여 pool로 들어간 모든 클라이언트의 요청을 처리한다. 이 함수에서 show buy sell 등의 클라이언트 요청을 처리하는 함수를 넣고, 결론적으로 Rio_written을 통해 클라이언트에게 메시지를 전달한다.

Task2

```
typedef struct{
    int ID;
    int left_stock;
    int price;
    int readcnt;
    sem_t mutex;
    sem_t w;
}item;

typedef struct {
 int *buf; /* Buffer array *
 int n; /* Maximum number of
 int front; /* buf[(front+1)
 int rear; /* buf[rear%n] is
 sem_t mutex; /* Protects ac
 sem_t slots; /* Counts avai
 sem_t items; /* Counts avai
} sbuf_t;

typedef struct node{
    item item;
    struct node* left;
    struct node* right;
}node;
```

Task2에서는 task 1의 pool을 사용하지 않는다. 대신에 별도로 sbuf_t 구조체를 선언하였고, sbuf에는 accept된 listenfd의 connfd가 sbuf_insert 함수로 인해 삽입된다. Sbuf_init, Sbuf_remove 함수로 인해 customer problem이 해소된다. 각 스레드에서 echo_cnt를 호출하고 그 안에서 모든 요청 처리가 이루어진다.

Task1과 같이 listen으로 클라이언트 요청을 대기한다. Task1과 다르게 Pthread_create 함수로 NTHREAD개의 Worker Thread를 생성한다. 무한 루프를 돌면서 클라이언트의 연결을 accept 하고 매번 클라이언트가 accept되면 sbuf_insert로 connfd를 전달해준다. Worker thread가 버퍼로부터 클라이언트 요청을 처리하게 하였다.

Worker Thread는 thread함수를 실행하며, thread를 detach상태로 변경하여 스스로를 떼어낸다. 따라서 thread는 종료되면 메모리 해제된다. 그리고 그 안에서 echo_cnt가 호출되어 요청에 대한 처리가 진행되고, close로 인해 메모리 해제된다.

각 item은 w, mutex를 가지고 있는데 각각 write , read mutex로 활용하였다. Buy, sell에서 item의 정보를 수정할 때 write mutex를 semaphore lock 하고 unlock 해주었다. Show와 save_stock에서 활용되는 print_stock에서는 read만 하기 때문에, read mutex로 lock하고 unlock 해주어 Reader-Writer problem을 해소시켜 주었다.

3. 구현 결과

Event-driven Approach 에서는 단일 스레드로 명령을 수행하기 때문에 어려운 문제가 생기지 않았다. Concurrent한 프로세스 요청 처리를 잘 받아들이는 것을 확인할 수 있었다.

Thread-base Approach 에서는 item structure 단위에서의 semaphore를 잘 사용하여, 클라이언트 간 Writer 충돌이나, 간섭 같은 현상은 일어나지 않음을 보였다. 서로 다른 클라이언트가 Show와 Buy를 거의 동시에 요청했을 경우 show의 데이터 값이 변할 수 있음에 대한 의문이 들었다. 만약 show가 먼저 처리되면, buy의 값은 반영되지 않은 채로 출력될 것이고, buy의 값이 먼저 처리되면 show에서 buy의 값이 반영되어 출력될 것이다. 이 현상을 막기 위해서는 show 명령 처리 중에 buy/sell을 block해야 한다. 하지만 show는 이 프로그램의 결과물에 영향을 미치지 않기 때문에 이 프로젝트에서는 문제가 없다.

그 외 한 가지 아쉬운 점은, 데이터를 저장하기 위해 구현한 바이너리 트리가 굉장히 비효율적이라는 점이다. 1~10으로 번호가 순서대로 들어갈 경우, 바이너리 트리의 장점을 하나도 가지지 않는다. 이를 해결하기 위해서는 레드블랙트리를 사용하면 된다.

4. 성능 평가 결과 (Task 3)

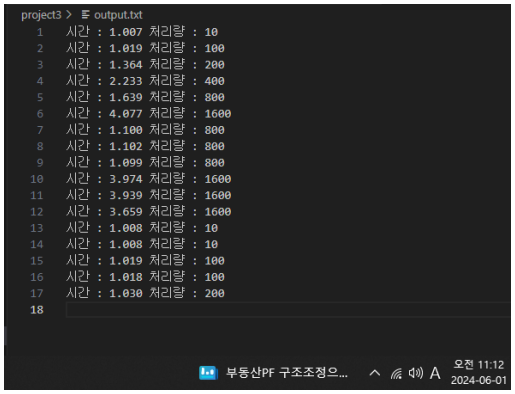
- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

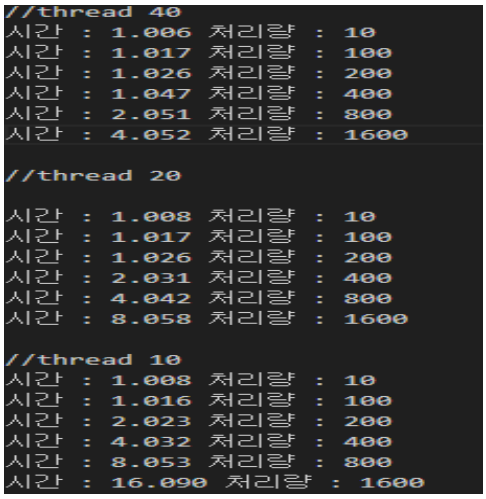
cspro9에서 서버를 실행하고, cspro에서 클라이언트를 실행하여 측정을 진행하였다.

측정 시각은 2024-06-01 오전 11시경이다.

우선, 클라이언트당 요청 처리 수를 10으로 고정하고, multiclient의 sleep 시간을 100000으로 줄여 주었다. 모든 시행은 5번 이상의 실험 후 서버 상 문제가 생긴 것으로 판단되는 특이점을 제외하고 평균을 내었다. 시간 단위는 ms(밀리초)를 사용하였다.

처리량은 client 수 * ORDER_PER_CLIENT 로 설정했고, 동시처리율은 처리량 / 시간으로 설정하였다.





위는 각각 task1, task2의 측정 결과 값이다. (예시)

0. 시행착오

Task 1

시간 : 1.019 처리량 : 100

시간 : 1.030 처리량 : 200

시간 : 1.051 처리량 : 400

시간 : 1.098 처리량 : 800

시간 : 2.052 처리량 : 1600

Task 2

//thread 20

시간 : 1.007 처리량 : 10

시간 : 1.019 처리량 : 100

시간 : 1.026 처리량 : 200

시간 : 2.030 처리량 : 400

시간 : 4.038 처리량 : 800

위와 같이 thread 20인 task2가 거의 두배 정도 시간에 걸리는 의아한 상황이 발생했다. 이 원인을 잡고자 여러 시도를 해본 결과, sleep 시간이 수행시간에 영향을 크게 미치는 것을 확인하였다.

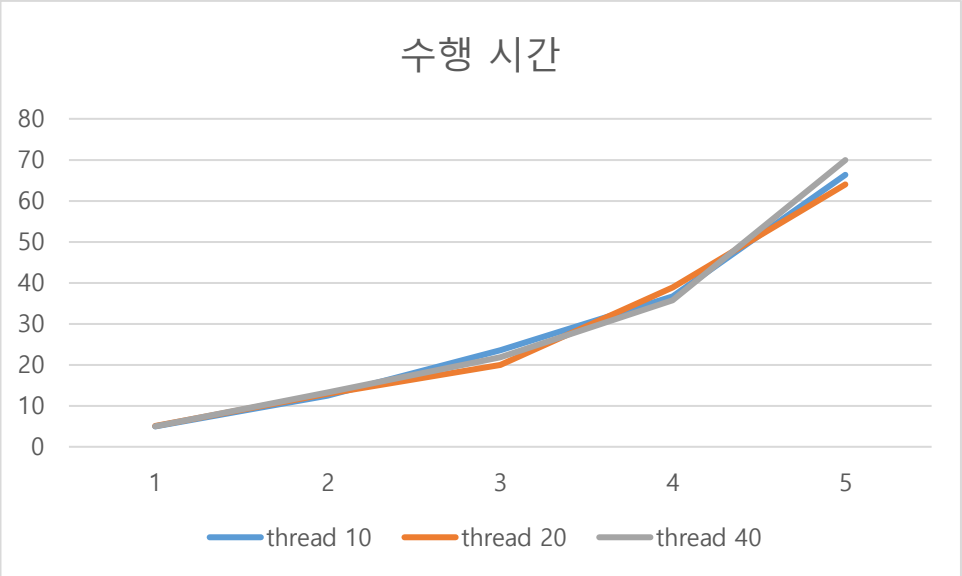
1. Buy/ sell / show 모두 실행하는 workload

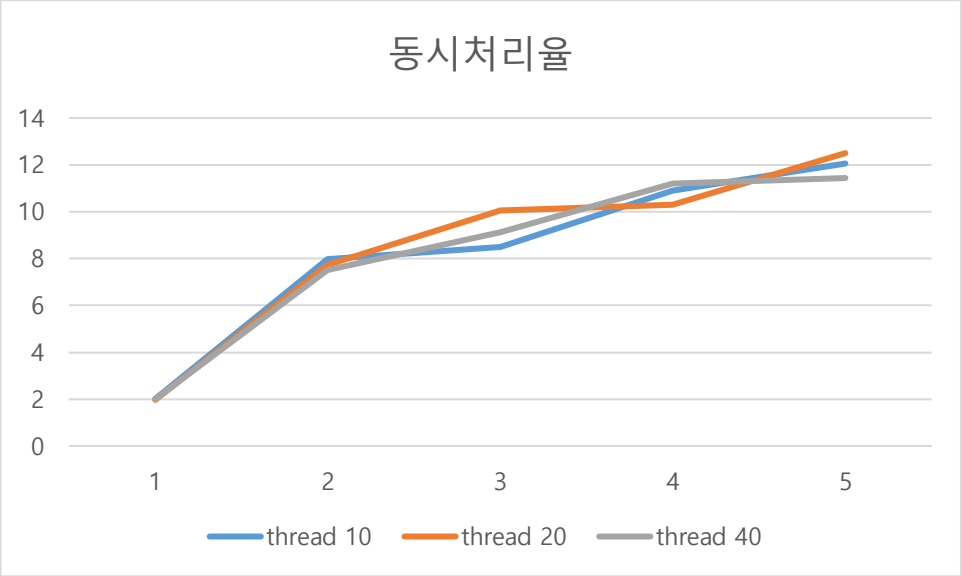
1-1) Task 1

buysellshow					
처리량	10	100	200	400	800
	6.09	20.823	37.847	64.759	109.881
동시처리율	1.642036	4.802382	5.284435	6.176748	7.280604

1-2) Task 2

buysellshow					
처리량	10	100	200	400	800
thread 10	4.971	12.527	23.551	36.682	66.343
thread 20	5.081	12.917	19.906	38.826	63.996
thread 40	4.997	13.307	21.886	35.711	69.936
동시처리율					
thread 10	2.011668	7.982757	8.492208	10.90453	12.05854
thread 20	1.968117	7.741736	10.04722	10.30237	12.50078
thread 40	2.001201	7.514842	9.138262	11.20103	11.43903





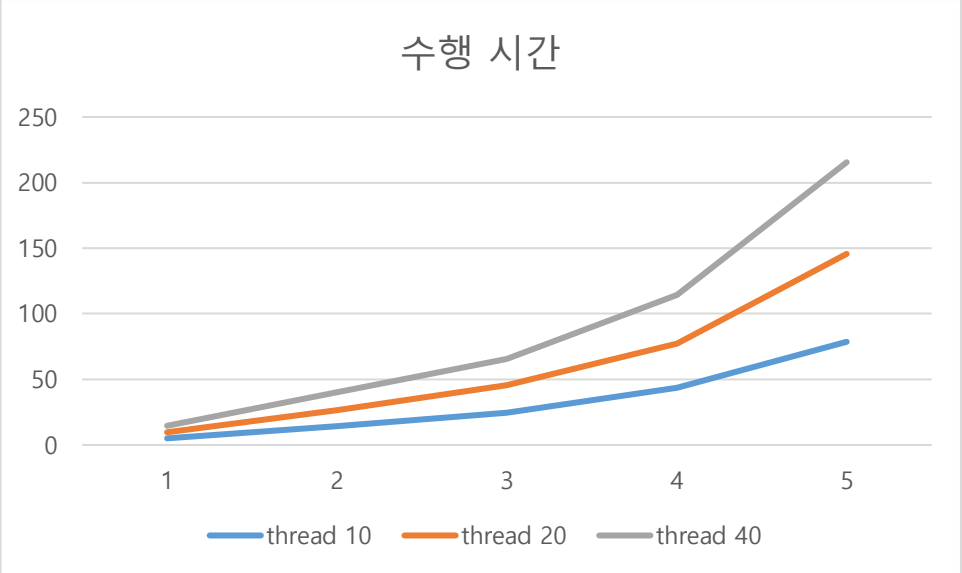
2. Buy/ sell 만 실행하는 workload

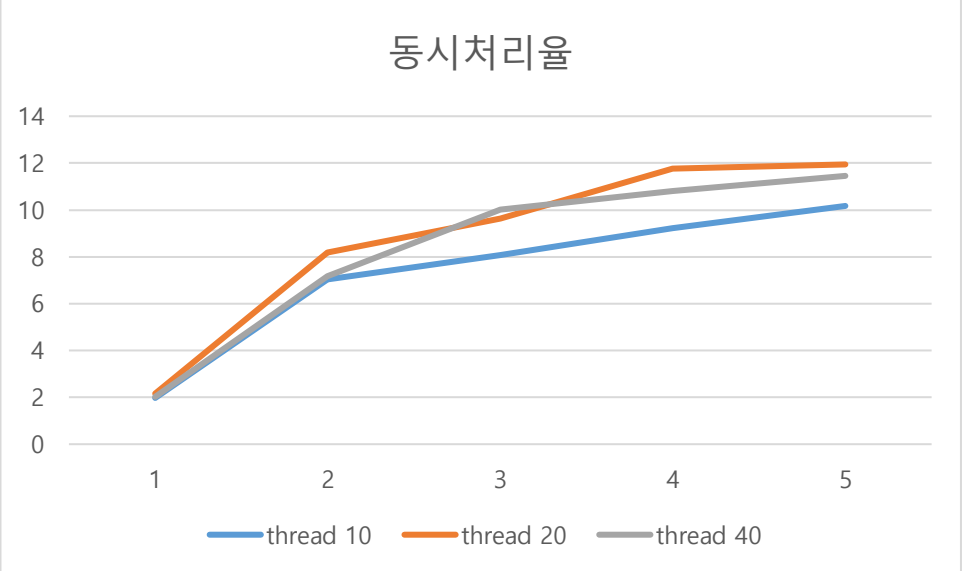
2-1) Task 1

buysell					
처리량	10	100	200	400	800
	6.084	18.927	30.903	58.328	99.265
동시처리율	1.643655	5.283457	6.471864	6.85777	8.059235

2-2) Task 2

buysell					
처리량	10	100	200	400	800
thread 10	5.064	14.204	24.762	43.384	78.663
thread 20	4.628	12.204	20.791	34.006	66.99
thread 40	4.941	13.968	19.996	37.036	69.847
동시처리율					
thread 10	1.974724	7.04027	8.076892	9.219989	10.16997
thread 20	2.160761	8.194035	9.619547	11.76263	11.94208
thread 40	2.023882	7.159221	10.002	10.8003	11.45361





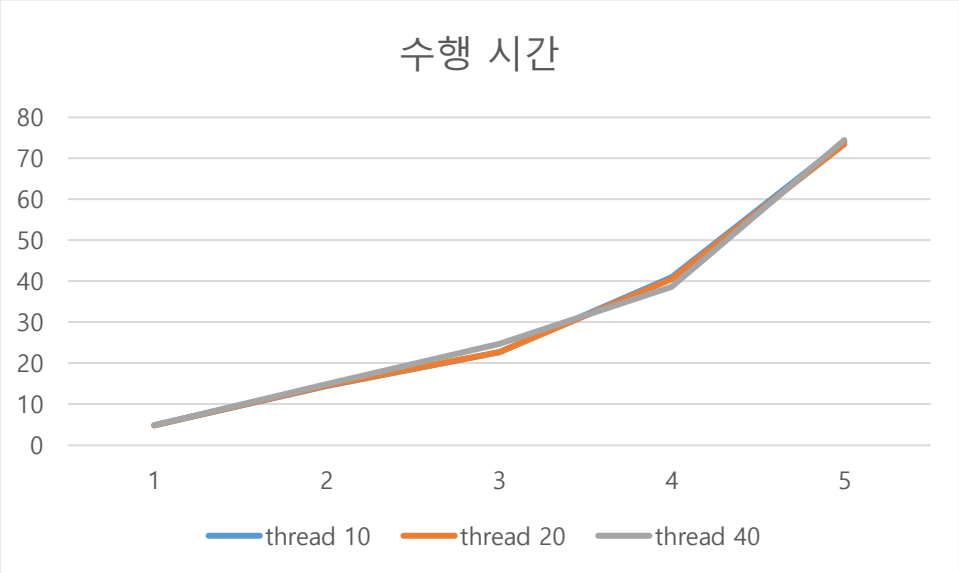
3. Show만 실행하는 workload

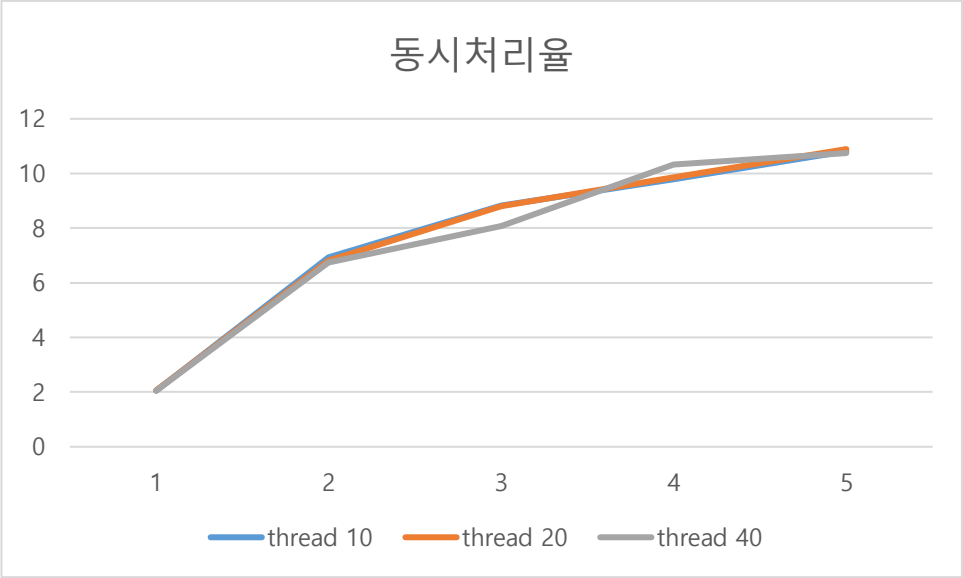
3-1) Task 1

show					
처리량	10	100	200	400	800
	7.433	25.671	38.395	66.798	121.44
동시처리율	1.345352	3.895446	5.209012	5.988203	6.587615

3-2) Task 2

show					
처리량	10	100	200	400	800
thread 10	4.881	14.451	22.641	40.908	73.98
thread 20	4.842	14.661	22.72	40.59	73.453
thread 40	4.872	14.851	24.775	38.69	74.442
동시처리율					
thread 10	2.04876	6.919936	8.833532	9.778039	10.81373
thread 20	2.065262	6.820817	8.802817	9.854644	10.89132
thread 40	2.052545	6.733553	8.072654	10.33859	10.74662





4. Task1과 Task2의 비교

   Task1보다 Task2가 전체적으로 현저히 높은 동시처리율과 짧은 수행시간을 보인다.

   그 차이는 수행하는 처리량이 늘어날수록 커졌고, Task1에서는 처리량이 두배 가량 늘어날 때 수행시간은 1.7~2배만큼 늘어난 반면, Task2는 처리량이 두 배 가량 늘어날 때에도 1.5배가량 늘어나는 모습을 보인다.

   동시처리율도 마찬가지였다. Task1은 처리량이 늘어남에 따라 동시처리율이 1부터 8까지, Task2는 2부터 11까지 늘어나는 모습을 보였다.

5. 워크로드에 따른 비교

   Task1에서는 show 를 수행할 때의 수행시간이 1.2배가량 커졌다. 서버 소스에서 print_server 함수를 호출할 때 재귀 함수를 사용했는데 이렇게 수행시간이 커진 현상은 단일 스레드에서 재귀함수를 호출할 때 다른 수행을 받아들이지 못하는 것 때문이라고 예측된다.

   Task2 에서는 buy/sell을 수행할 때의 수행시간이 더 큰 것으로 분석된다. Reader-Writer problem을 해결하기 위해 write mutex를 따로 선언하고 데이터를 수정 시에 Reader는 들어올 수 없지만, Reader 수행 시에는 Reader가 들어올 수 있기 때문에 show만 수행하거나 show와 buy/sell을 동시에 수행할 때 보다 느려지는 현상이 보이는 것으로 분석하였다.

6. 클라이언트 개수 변화에 따른 동시 처리율 변화 분석

클라이언트 개수 증가에 따라 동시처리율의 증가는 로그함수의 그래프 선형을 보이며 증가하는 것으로 보였다. 150 클라이언트가 넘어가면 시간이 크게 점프하는데, 여기서부터 클라이언트의 fork에서 시간을 많이 잡아먹는 것으로 분석되었다. Thread의 개수별로 동시처리율이 크게 구분될 줄 알았으나 위 실험에서는 그렇지 않고 오차를 제외하면 구분할 수 없을 정도가 되었다. Thread 20에서 가장 큰 동시처리율을 보이는 경향이 있기는 하다. 따라서 sleep을 다시 넣고 실험을 해볼 필요가 있다.

7. 종합 평가

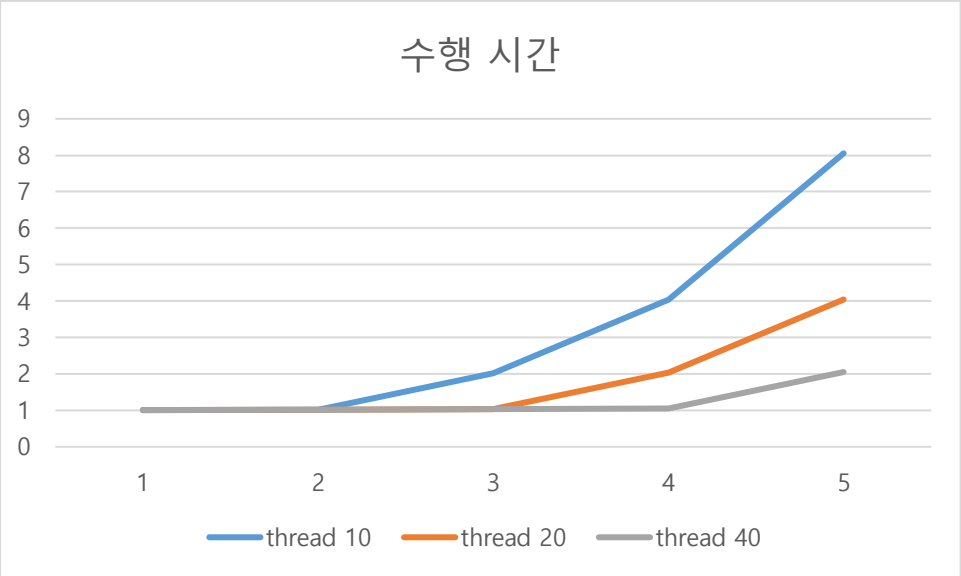
앞선 실험에서 Task1과 Task 2를 비교하고, 워크로드간의 수행 시간과 동시처리율은 잘 확인할 수 있었다. 이렇듯 Sleep을 빼고 실험하니, Task2가 task1보다 좋은 결과를 얻을 수 있었다. 하지만 스레드 개수에 따른 변화를 확인하는 것은 어려웠다. 서버의 수행시간에 따른 오차 범위가 있고, 워낙 짧은 프로세스를 수행하다 보니 thread간의 context switching 시간을 컨트롤하거나 여부를 확인하기 어려웠다.

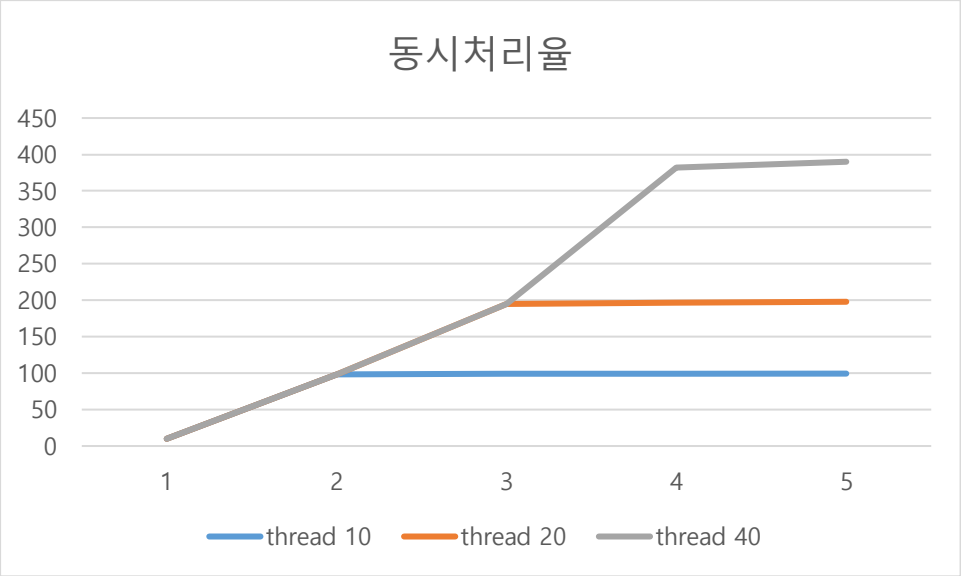
따라서 [0]의 시행착오로 돌아가보면, (sleep 다시 넣기)

처리량	10	100	200	400	800
thread 10	1.008	1.016	2.023	4.032	8.053
thread 20	1.008	1.017	1.026	2.031	4.042
thread 40	1.006	1.017	1.026	1.047	2.051
동시처리율					
thread 10	9.920635	98.4252	98.86307	99.20635	99.34186
thread 20	9.920635	98.32842	194.9318	196.9473	197.9218
thread 40	9.940358	98.32842	194.9318	382.0439	390.0536

위와 같은 동시처리율을 보인다.

위 실험에서는 1s의 시간단위를 사용하였다.





그래프를 확인해보았을 때, client수가 늘어남에 따라 각 Thread의 수의 경우에서 Thread의 수가 가 client 개수와 같을 때의 동시처리율에 머무는 현상을 확인할 수 있었다.