System Programming Project 2

담당 교수 :김영재

이름 [redacted]

학번 [redacted]

1. 개발 목표

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

개발의 주 목표는 주식 서버가 concurrency를 보장하는 주식 서버라는 것에 있다. 프로세스 기반, 이벤트 기반, 스레드 기반이 있는데, 이 중 이벤트 기반과 스레드 기반으로 이루어진 서버를 구축한다. tcp socket과 그 주소, connect, listen, accept등에 대해 이해한 상태로, 베이스코드 내에서 동시성을 구현한다.

주식 서버가 하는 일은 사용자와 연결되어 사용자의 입력을 받고, 해당하는 요청을 메모리의 주식 정보와 비교하여 수행한다. 주식 정보를 조회하고, 판매하고, 구매할 수 있도록 한다. 단, 앞서 밝힌 대로 이는 I/O Multiplexing, thread를 통해 구현되어야 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   고객의 입장에서는 여러 고객이 접속하여, 각각 조회, 구매, 판매, 연결 끊기 등을 진행한다. 이는 아래의 task2도 다르지 않다.

   한편, 이를 서버에서 보면 사용자들의 연결에 대해 pending bit vector를 관리하여 어떤 fd에 내용이 들어왔는지(listenfd 포함), 즉 읽을 event의 발생을 포착하여 순차적으로 진행해주는 것이 필요하다. 다만, 악성 코드나 큰 작업량에 의해 다른 사용자들의 요청에 응답을 하지 못하는 경우가 발생해서는 안 된다. 이것까지 고려하여 구현한다.

   즉, 한 개의 프로세스 내에서 I/O multiplexing을 통해 요청과 응답이 concurrent하도록 서버를 작성한다.

2. Task 2: Thread-based Approach

   고객의 결과는 위와 같다.

   Thread 기반 서버는 미리 thread를 생성하여 pool을 관리하는 thread

pooling을 하고, listen을 통해 사용자를 항상 기다리고, shared_buffer에 가능한 공간이 있는 이상 사용자의 접속을 Accept하여 buffer에 담아준다. 이 버퍼의 connfd는 기다리던 thread, 혹은 연결이 끝나 일을 마치고 돌아온 thread가 꺼내 가게 된다. 이러한 공유된 버퍼의 구현이 중요하다.

한편 thread간에 공유되는 주식 데이터 tree가 있는데, 이는 synchronization에 해가 되는 critical section이다. 그래서 thread의 진입을 잘 관리해줘야 한다. 이를 위해 쓰이는 것이 semaphore의 mutex와 counting semaphore이다.

즉, 스레드 풀의 스레드들과 메인 스레드의 listen 부분은 각각 소비자-생산자 관계에 있다. 그래서 sio_buf 패키지를 이용한다. 한편, 스레드 간 주식 정보 접근의 문제는 reader-writer 문제라고 할 수 있다.

3. Task 3: Performance Evaluation

성능 분석은 다음과 같이 진행한다. 첫 번째로 확장성을 분석한다. client의 개수가 늘어날 때 어떤 방법이 throughput, 즉 동시처리량이 가장 높은지를 보는 것이다.

두 번째로, 워크로드에 따른 분석이다. 각 client가 보내는 요청은 임의적인 면이 있어서, 워크로드 별로 분리하여 수행 및 비교하면 무엇이 더 동시 처리율이 높은지를 분석해 낼 수 있다.

마지막으로 강의에서 배운 내용과 어떤 부분이 일치하는지, 그 외 기타 탐구를 통해 분석결과를 얻어본다.

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
  - ✓ epoll과의 차이점 서술

1. multi-client 요청에 따라 어떻게 I/O Multiplexing을 하도록 작성하였는지 밝혀본다. 먼저 fd의 pool, 즉, clientfd의 pool에 대해 관리를 해 주어야 한다.

먼저 필요한 초기화를 진행하면서 listenfd를 읽도록 pool에 담아준다. 이렇게 하는 것 이후에 반복문을 계속 돌면서 listenfd에 I/O event가 있으면 accept 하여 add_client를 실행하고, 여기서 새로운 connfd를 pool에 저장하여 이 connfd에 대해서도 read하도록 read_set을 설정해 준다.

반복문의 다음 부분은 다음과 같다. 모든 활성화된 connfd에 대해서, 그 중에서도 I/O event가 있었던 connfd에 대해 각각을 모두 읽지 않고, 각각 한 라인씩 읽어 처리한 후, 그에 대해 응답해준다. 이렇게 진행하는 것을 통해 어느 큰 작업량, 혹은 악성 사용자에 의해 concurrency가 방해 받는 일 없이 I/O Multiplexing이 가능해지게 된다.

2. epoll과 select의 차이는 다음과 같다. select에서 커널은 각각의 fd, connection 모두의 waitlist와 프로세스를 연결해준다. 프로세스가 깨어나면 waitlist에서 모두 제거해준다. 이렇게 대기 구조를 붙이고 모두 free하는 오버헤드가 존재한다. 반면에 epoll은 한 waitlist에 대해서만 연결되고, 하나의 대기 구조만 free하면 된다. 또한 select는 입출력이 발생하지 않은 socket에 대해서도 발생을 확인해야 하는 overhead가 있지만, epoll은 그러한 발생이 있으면 해당 epoll 소켓에 그 내용이 전달된다는 이점이 있다.

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

  ✓ Master Thread의 Connection 관리

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

1. master thread, main thread의 connection 관리

먼저 master thread는 모든 connection요청에 대해 반복적으로 처리하고 있는 구조를 가진다. 항상 listen을 통해 connect 요청을 기다리고 있다가, connect를 받으면 accept하여 connected_fd를 shared buffer에 넣어준다. 단, 이 과정에서는 sbuf package를 이용하여 semaphore를 활용한 synchronization을 꼭 진행해준다. 만약 shared buffer가 가득 차면, 더 이상 client의 요청을 승인하지 않고, sbuf_insert의 P부분에서 suspend 상태에 들어간다. 이후 thread가 shared buffer에서 item을 꺼내가면, V에 의해 깨어나게 된다. 이렇게 연결을 관리한다.

2. Worker Thread Pool 관리

worker thread pool은 서버가 사용자를 받기 이전에, 이미 서버가 만들어 둔 thread들의 모음이다. 이러한 thread pool에서 thread들은 서버 시작 전에

생성이 되었으므로, 보통 sbuf_remove의 P에 걸려서, 즉 가져올 아이템 connected_fd가 없어서 진행을 하지 못하고 있다. 이후 메인 스레드가 connfd를 넣고 V로 깨워주는 것에 따라서, 사용자와의 연결을 한 스레드가 맡게 된다. thread pool에 생성된 만큼의 사용자와 동시에 소통할 수 있고, 연결이 끊기면 반복문에 의해 다시 다음 connfd를 기다리게 된다. (혹은 바로 수행된다.) 이렇게 thread의 풀이 관리되고 있다.

- **Task3 (Performance Evaluation)**

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

  ✓ Configuration 변화에 따른 예상 결과 서술

  1. metric, 즉 측정법은 다음과 같다. 동시 처리율은 시간당 처리 요청의 개수로 계산하는데, (client 당 요청의 개수 X client 의 수 / 시간)으로 표현할 수 있을 것이다. 즉 단위 시간당 얼마나 많은 요청을 처리하고 있는지를 '동시 처리율'이라고 본다. 개발 방법은 다르지만, 비슷한 요청을 받고 비슷한 입출력을 받는다는 점에서 이렇게 '동시 처리율'로 성능을 비교해 볼 수 있을 것이다. 측정 방법은 기술서에 명시된 gettimeofday 를 이용하여 시간을 재고, client 측에서 보낸 요청을 처리하여 응답하는 것까지가 요청의 처리라 생각하여, client 측에서 시간을 재도록 작성할 것이다. 워크로드의 종류에 따른 분석과 스레드 기반 서버의 분석에 대해서도 뒤에서 더 다뤄볼 예정이다.

  2. configuration 의 변화에 따라 어떻게 내용이 바뀔지를 예상하기가 힘들다고 생각한다. 이벤트 기반서버는 select 의 오버헤드와 FD_SET 설정 등의 오버헤드가 있지만, 한편으로 스레드 기반서버는 thread 의 스케줄링에 따라 context change 에 오버헤드(프로세스 기반 서버보다는 작더라도, 존재함, 병렬 처리를 위해 kernel 이 하는 영역)가 존재하기 때문에, 딱히 분석을 진행하지 않은 상황에서는 예상이 어려운 것 같다.

  하지만, 이해한 내용 상 이벤트 기반서버는 하나의 process 위에서만 돌아가서 멀티프로세서의 이점을 받지 못하기 때문에, 아마 thread 기반 서버가 더 빠르지 않을까 예상할 수 있다. 또한 워크로드의 종류에 따라 둘의 차이가 크게 달라질 것 같지는 않다.

C. 개발 방법

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

-task1

앞서 B에서 이미 함수와 패키지를 조금 언급하였다. 강의자료, 책, 강의에서 언급된 connected file descriptor pool 구조체를 구현하여, 배열을 관리하고, 보다 효과적으로 select 함수 호출 및 진행을 할 수 있게 한다. 관련된 init_pool, add_client, check_clients 함수가 수정되어 사용되는데, check_clients가 입출력을 처리하는 함수인 만큼 많은 수정이 가해져야 할 것이다. 또한, tree 구조체를 선언하고, 파일을 읽어와 노드를 생성하고, 주식을 사고, 팔고, 그리고 그 과정에서 생긴 노드의 변경사항을 파일에 저장하는 등의 함수가 필요해진다.

-task2

앞서 파일을 읽어오고, 저장하는 부분은 수정하지 않는다. thread가 하나의 경우여서 논리 상 차이가 없기 때문이다. 다만, thread가 여럿 실행되고 있어서, 공유된 data structure에 대해 어떻게 조회(read)와 수정(write)이 이루어질지를 고려하여 show, buy, sell등에 따른 함수 실행이 되어야 할 것이다. 그래서 task1의 tree의 노드를 읽고 수정하는 부분들을 많이 바꿔줘야 하게 된다. 여기서 사용되는 것이 reader-writer problem 강의자료의 내용인데, 이러한 구현을 위해 task1과는 다르게 tree의 노드에 mutex와 추가로 write에 대한 mutex가 필요해진다. 이를 추가해준다.

또한 sbuf package의 구조체와 함수를 통해 thread pool의 관리가 원활해지므로, 이 또한 추가하여 semaphore의 count와 mutex를 통해 synchronization의 문제 없이 동작하도록 작성해준다.

-task3

gettimeofday의 문서 내용에 따라 시간이 얼마나 걸리는지를 확인하고, 추가적으로 가능하다면, 동시 처리율이 계산 가능하도록 수정한다.

multiclient.c의 내용에 대해서, maxclient의 개수를 바꾸어 실험이 이루어질 수 있도록 하고, 이후 워크로드에 따른 변화를 진행할 때에는 option값을 고정시켜주거나, rand 값에 대한 조작을 통해 원하는 워크로드의 set이 나오도록 조작해준다.

확장성을 분석하기 위해 5 10 20 30 50 70 100 200 500 1000까지 내용을 진행해본다.

3. 구현 결과

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

(구현 결과에 대해서는 앞서 말한 부분과 반복되는 부분이 많다고 봅니다. 또한 언급한 내용 중 미처 구현하지 못한 부분을 없습니다.)

간략히 작성하자면, task1의 내용은 event driven based server를 유저의 입출력을 각 소켓에 대해 select를 통해 커널과 소통하여 확인하는 것을 통해 '이벤트'를 확인하고, 상응되는 응답을 각 모든 이벤트에 concurrent하게 보내줄 수 있도록 I/O multiplexing을 작성했다. 실행 결과는 이후 4번의 내용 혹은 명세서와 같다.

다음으로 task2의 내용은 thread based server를 만들어서 thread pooling을 통해 thread를 효과적으로 관리하고, 마스터 스레드는 모든 요청을 accept하려고 listen하고, 요청을 받으면 thread pool의 thread가 이후 쓸 connected_fd를 공유 버퍼를 통해 전달해주게 된다.

모든 연결에 대해 최대 thread개수만큼 동시다발적으로 유저와 소통할 수 있고, 다만 공유된 자료구조에 대해서는 노드 단위로 쪼개어 노드에 접근할 때 락을 걸되, reader가 있는 경우에는 readcnt가 0이 될 때까지 계속 reader가 늘어날 수 있는, 즉 reader-favored한 주식 서버를 작성하게 되었다.

마지막으로 task3의 경우는 다음의 4번이 결과 항목이므로 4번에서 작성하도록 한다.

4. 성능 평가 결과 (Task 3)

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

첫 번째로, 확장성에 대해 고려하여 두 서버의 성능을 비교하였다. 측정 시점은 서버가 모두 준비된 이후 multiclient를 실행하되, multiclient내의 fork를 띄우는 while 문 이전부터 측정을 시작한다. 측정이 끝나는 시점은 모든 child process가 waitpid에 의해 reap되고 난 후로 한다.

바로 다음의 캡쳐화면과 같이 시작지점, 종료지점이 각각 설정된다.

```
        port = argv[2];
        num_client = atoi(argv[3]);

        // time measure start
        struct timeval start;   /* starting time */
        struct timeval end;     /* ending time */
        unsigned long e_usec;   /* elapsed microseconds */
        gettimeofday(&start, 0);        /* mark the start time */


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

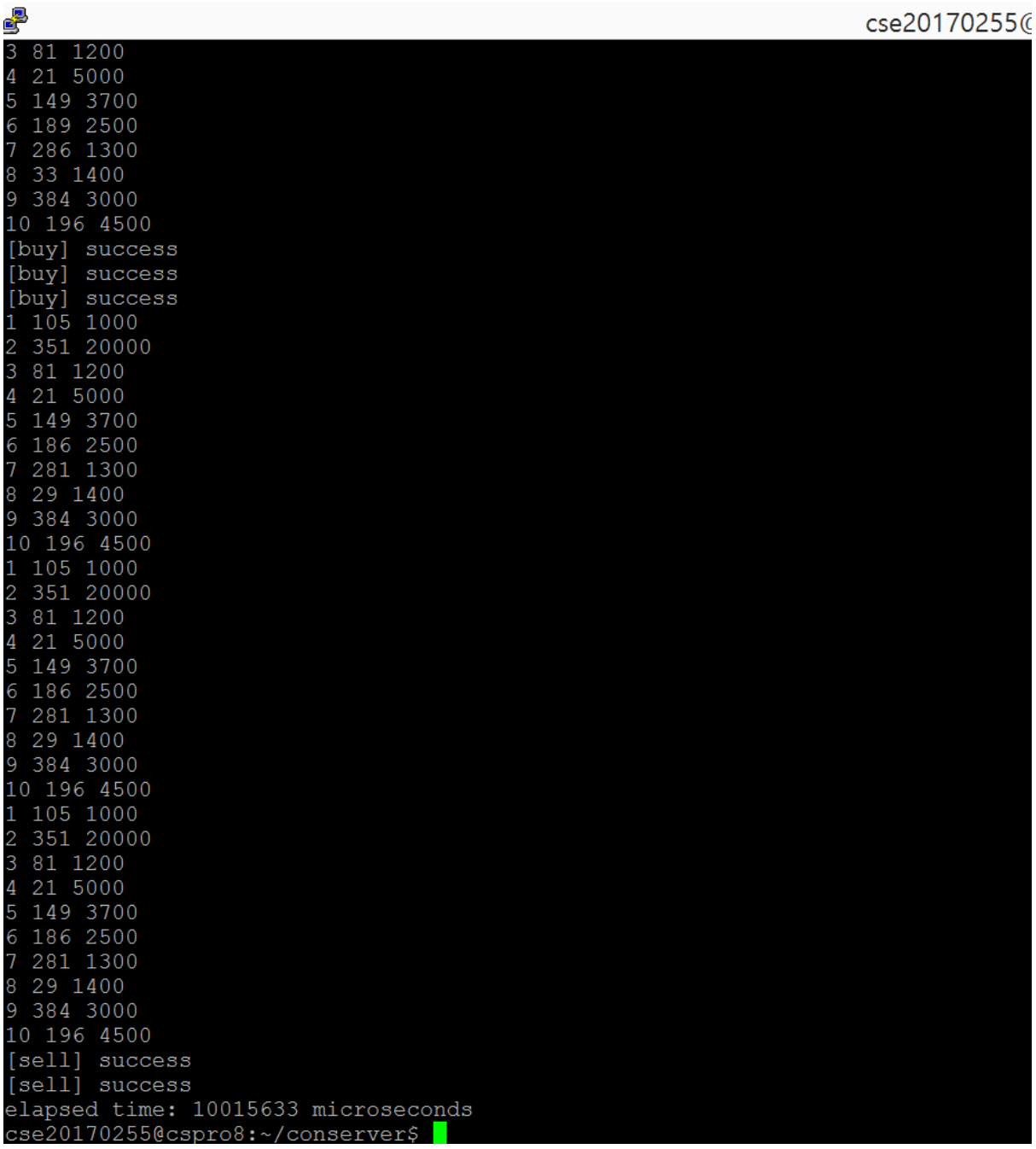
```
  }
  for(i=0;i<num_client;i++){
          waitpid(pids[i], &status, 0);
  }

  // time measure end
  gettimeofday(&end, 0);          /* mark the end time */
  /* now we can do the math. timeval has two elements: seconds and microseconds */
  e_usec = ((end.tv_sec * 1000000) + end.tv_usec) - ((start.tv_sec * 1000000) + start.tv_usec);

  printf("elapsed time: %lu microseconds\n", e_usec);
```

1 5 10 25 50 75 100 250 500 750 1000 에 대해 수행하였다. 이 각각에 대한 시간을 다음과 같이 엑셀로 옮겨 적었다. 그리고 client당 수행 작업 수는 10개였다. μs 단위가 알맞은 정도였다.

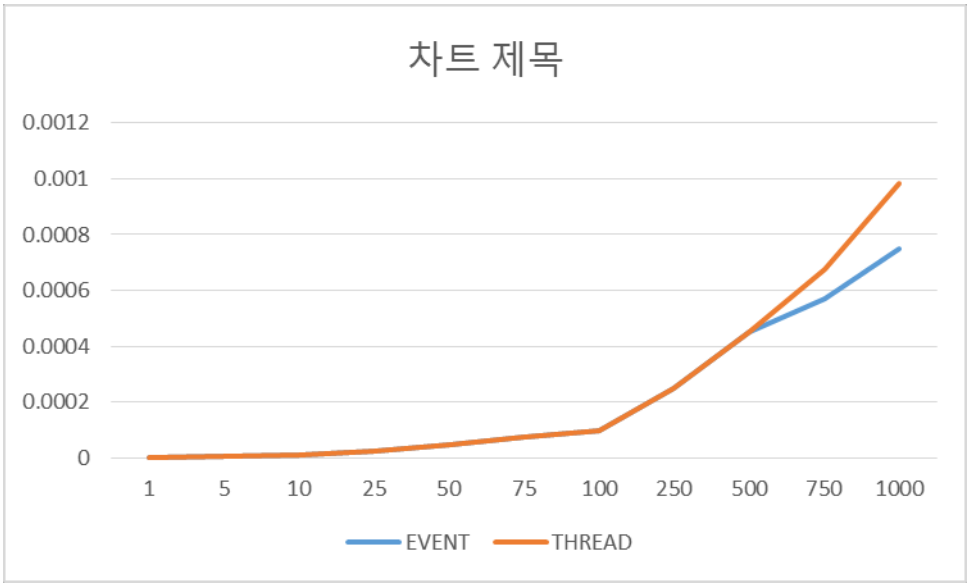
(다음은 75개의 실행 예시)



이와 같이 1000까지 수행한 후 다음과 같은 결과가 나왔다.

process개수(event)	1	5	10	25	50	75	100	250	500	750	1000
수행개수	10	10	10	10	10	10	10	10	10	10	10
총 처리 개수	10	50	100	250	500	750	1000	2500	5000	7500	10000
걸린시간(μs)	10005436	10006894	10006974	10008839	10012951	10015633	10019283	10046085	11084044	13115731	13323605
동시 처리율	9.9946E-07	4.9966E-06	9.993E-06	2.4978E-05	4.9935E-05	7.4883E-05	9.9808E-05	0.00024885	0.0004511	0.00057183	0.00075055
process개수(thread)	1	5	10	25	50	75	100	250	500	750	1000
수행개수	10	10	10	10	10	10	10	10	10	10	10
총 처리 개수	10	50	100	250	500	750	1000	2500	5000	7500	10000
걸린시간(μs)	10006073	10006877	10007226	10009363	10096880	10011663	10014445	10028538	11048754	11068036	10192457
동시 처리율	9.9939E-07	4.9966E-06	9.9928E-06	2.4977E-05	4.952E-05	7.4913E-05	9.9856E-05	0.00024929	0.00045254	0.00067763	0.00098112

이러한 상황에서 EVENT와 THREAD의 차트를 분석해 보았을 때,



이와 같이 분석이 되었다. 즉, THREAD 기반 서버는 동시 처리율이 EVENT 기반 서버보다 500 혹은 500 초과부터 높아지기 시작한다는 것을 알 수 있다.

즉, 500 정도 이상의 접속에서는 더 많은 양을 동시에 처리할 수 있다는 것이다. 이를 위해 추가적으로 그 이상 실험해보려 했으나, 너무 많은 fd를 열려고 하면 오류가 나서 실험을 진행하지 못하였다.

두 번째로, 워크로드에 따른 분석을 진행해 보았다. 진행하고자 한 예시는

1) 모든 client가 sell만을 요청하는 경우이다. buy는 수량이 부족하면 걸리는 시간이 달라질 거라 예상하여 이렇게 진행하였다.

2) 모든 client가 show만 요청하는 경우이다.

3) buy와 sell만 섞어서 요청하는 경우이다.

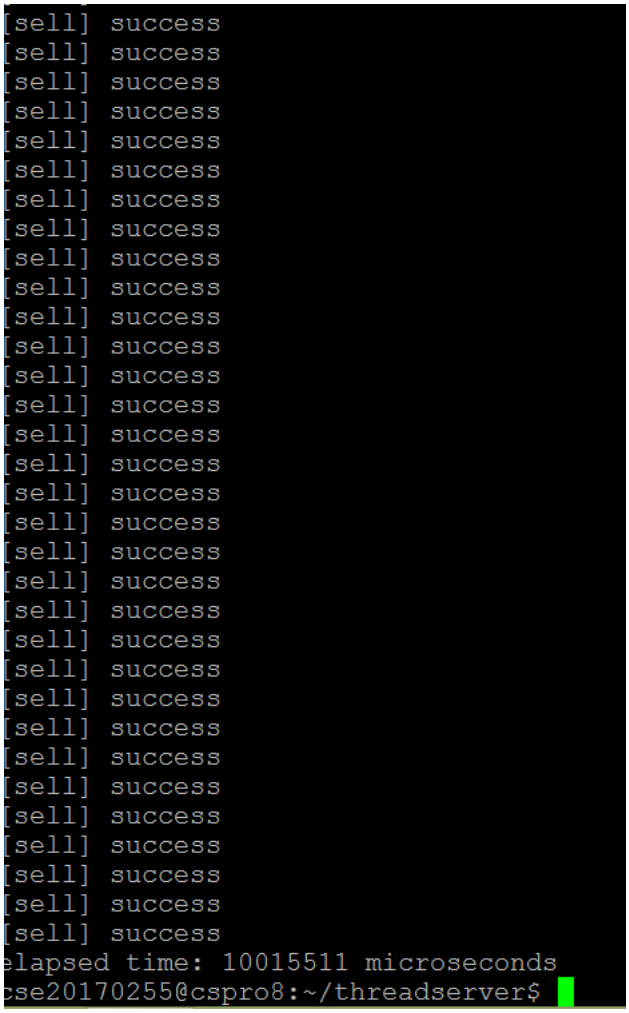
1) 간소화 및 단적인 비교를 위하여, 프로세스를 100개, 1000개씩 띄워 두 번 진행하도록 하였다. 단, sell의 개수는 1로 제한하였다.

```
#define MAX_CLIENT 1000
#define ORDER_PER_CLIENT 10
#define STOCK_NUM 10
#define BUY_SELL_MAX 1
```

```
for(i=0;i<ORDER_PER_CLIENT;i++){
        int option = 2;

        if(option == 0){//show
                strcpy(buf, "show\n");
        }
        else if(option == 1){//buy
                int list_num = rand() %
                int num_to_buy = rand()

                strcpy(buf, "buy ");
                sprintf(tmp, "%d", list
                strcat(buf, tmp);
                strcat(buf, " ");
                sprintf(tmp, "%d", num_
                strcat(buf, tmp);
                strcat(buf, "\n");
        }
        else if(option == 2){//sell
```



이렇게 sell만 100번 1000번 진행한 결과는 다음과 같다.

event에서 10018362 13144477

thread에서 10015511 13111848

걸린 시간을 비교하여, 시간이 적게 걸린 쪽이 처리율이 높다고 생각할 수 있다. 따라서 100일 때, 1000일 때 모두 'sell과 같은 단순한 명령을 10번 처리하는 워크로드에 대해' thread기반의 서버가 더 빠른 것을 알 수 있다. 즉, 더 많이 처리할 수 있었다.

2) 이번에는 show에 대해 똑같이 100번, 1000번 수행하는 워크로드를 비교해 보았다.

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

```
1 1587 1000
2 1706 20000
3 1232 1200
4 1122 5000
5 1332 3700
6 1392 2500
7 1619 1300
8 1081 1400
9 1344 3000
10 1124 4500
1 1587 1000
2 1706 20000
3 1232 1200
4 1122 5000
5 1332 3700
6 1392 2500
7 1619 1300
8 1081 1400
9 1344 3000
10 1124 4500
1 1587 1000
2 1706 20000
3 1232 1200
4 1122 5000
5 1332 3700
6 1392 2500
7 1619 1300
8 1081 1400
9 1344 3000
10 1124 4500
elapsed time: 10017931 microseconds
cse20170255@cspro8:~/conserver$
```

이렇게 show만 진행한 결과는 다음과 같다.

event에서 10017931 13182764

thread에서 10014669 13090873

이와 같이, 걸린 시간을 비교하여, event 기반의 서버가 더 빠른 것을 확인할 수 있다. P와 V의 synchronization을 고려하는 오버헤드가 있는 thread 서버의 영향일 것이라 생각하였다. 100개, 1000개 모두에서 event가 더 많은 show를 처리할 수 있었다.

3) buy와 sell을 섞어서 요청하는 경우를 살펴본다. 위와 같이 100개, 1000개를 띄워 진행하고, 이 워크로드에서는 무엇이 더 빠른지 확인해본다.

```
for(i=0;i<ORDER_PER_CLIENT;i++){
        int option;
        while (1) {
                option = rand()%3;
                if (option != 0) break;
        }
```

```
[buy] success
[sell] success
[buy] success
[sell] success
[buy] success
[buy] success
[sell] success
[sell] success
[buy] success
[buy] success
[buy] success
[buy] success
[sell] success
[buy] success
[sell] success
[buy] success
[sell] success
[buy] success
[sell] success
[sell] success
[buy] success
[buy] success
[buy] success
[sell] success
[buy] success
[buy] success
[buy] success
[sell] success
elapsed time: 10014121 microseconds
cse20170255@cspro8:~/threadserver$
```

이와 같이 진행하여서 다음과 같은 결과가 나왔다.

event에서　10016561 13195130

thread에서　10014121 13095075

이 경우에는 앞서 sell만 진행하는 경우가 그랬던 것처럼, thread 기반 서버의 처리속도가 더 빠르게 나왔다. buy와 sell이 섞인 경우에, 더 빠른 동시처리율을 가지고 있다고 볼 수 있다.

워크로드 분석에 대해 요약하면, 모두 분류를 하여 진행해보니, sell, buy와 같은 경우는 thread가, show의 경우는 event가 빨랐다. 그리고, 앞서 진행한 3가지 명령어 모두 섞어서 사용한 경우에는 thread가 더 빨랐다는 점을 볼 때, 기본적으로 thread가 주식 서버에 적합하다고 분석해 볼 수 있다. 다만, event 기반이 빠른 경우도 있기 때문에 특정 수행 목적에 따라 잘 분석하여 서버를 짜야 한다는 것을 생각해 볼 수 있었다.

또한 주의해야 할 점은, 앞서 차트에서는 thread가 1000개 프로세스에서 매우 빠른 속도로 진행되었지만, 실행 상황에 따라 event와 거의 차이 없는 정도로도 나오기도 한다는 점을 주의해야 할 것 같다. (하지만 평균적으로 thread가 빨랐다.)

특정한 목적에 따라 서버 구현 선택이 달라진다는 강의의 내용을 실제로 실험을 통해서도 확인할 수 있었으며, thread가 일반적으로 빠르다고 강의 중에 들었던 것 같은데, 그것 또한 확인할 수 있었다. 또한, 이렇게 진행되려면 thread pooling과 미리 thread를 생성해 두는 것이 중요함을 깨달을 수 있었다.

(이상으로 보고서를 마칩니다. 감사합니다.)