System Programming Project 3

담당 교수 : 박성용

이름 : 김택림

학번 : 20201574

1. 개발 목표

- 주식 서버에서는 client를 맞을 준비를 하고 client가 보낸 명령에 따른 응답을 client로 보내고 client에서는 server에 연결을 요청하고 명령어를 server로 보내는 주식 서버를 구현한다. 이때 주식 서버를 Event-driven Approach, Thread-based Approach 두 가지 방식으로 구현해 봄으로서 concurrent programming, synchronization에 대한 전반적인 이해도를 높인다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Pool을 만들어 client가 연결될 때마다 read_set에 연결된 표시를 하고 client로부터 명령어가 들어온다면 명령어가 들어온 위치를 select를 통해 1로 바꿔주고 순서대로 명령어들을 처리해준다.

   show : client가 주식 서버로 show명령어를 보내면 server는 client로 현재 주식 서버에 있는 주식의 ID, 남은 주식 수, 주식의 가격을 보여준다.

   buy [주식 ID] [살 주식 개수] : client가 주식 서버로 buy 명령어와 함께 사고 싶은 주식의 ID, 살 주식의 개수를 보내면 서버에서 주식 tree에서 해당 주식의 남은 개수가 살 주식 개수보다 많거나 같은지 확인한다. 적다면 Not enough left stock을 client에게 보내고 같거나 많다면 [buy] success를 보내고 주식 tree에 해당 내용을 반영한다.

   sell [주식 ID] [살 주식의 개수] : client가 주식 서버로 sell 명령어와 함께 팔고 싶은 주식의 ID, 팔 주식의 개수를 보내면 서버에서 [sell] success를 보내고 주식 tree에 해당 내용을 반영한다.

2. Task 2: Thread-based Approach

   Master thread에서 client를 accept하면 thread를 생성해 connfd를 보내줘 해당 thread와 client와 상호작용을 하도록 한다.

   show : client가 주식 서버로 show명령어를 보내면 server는 client로 현재 주

식 서버에 있는 주식의 ID, 남은 주식 수, 주식의 가격을 보여준다.

buy [주식 ID] [살 주식 개수] : client가 주식 서버로 buy 명령어와 함께 사고 싶은 주식의 ID, 살 주식의 개수를 보내면 서버에서 주식 tree에서 해당 주식의 남은 개수가 살 주식 개수보다 많거나 같은지 확인한다. 적다면 Not enough left stock을 client에게 보내고 같거나 많다면 [buy] success를 보내고 주식 tree에 해당 내용을 반영한다.

sell [주식 ID] [살 주식의 개수] : client가 주식 서버로 sell 명령어와 함께 팔고 싶은 주식의 ID, 팔 주식의 개수를 보내면 서버에서 [sell] success를 보내고 주식 tree에 해당 내용을 반영한다.

3. Task 3: Performance Evaluation

Server가 client와 연결을 한 뒤부터 모든 client와의 연결이 끊길 때까지의 시간을 측정한다. 이때 multiclient파일의 configuration을 바꿔가면서 여러가지 동작방식에 따른 시간 차이, 주어진 명령어에 따른 시간 차이 등을 비교해보면서 수업시간에 배운 내용과 일치하는지 또는 불일치하는지 분석을 해본다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Pool을 만들어 main함수에서 client를 accept할 때마다 clientfd에 어떤 file descriptor에 연결됐는지 적어주고 Rio buffer를 init해준다. 그 다음 연결된 file descriptor의 위치를 read_set에 표시를 해준다. client에게 요청이 온다면 select를 통해 어느 client에게 왔는지 ready_set에 표현을 해주고 그렇게 들어온 명령어들을 select의 return값만큼 처리를 해준다.

✓ epoll과의 차이점 서술

Task1에서는 select를 사용했다. select의 경우는 받을 수 있는 파일 디스크립

터의 최대 수가 FD_SETSIZE로 정해져 있고 호출할 때 이벤트가 발생한 것을 알려주기 위해 fd_set에 표시를 해준다. 대부분의 Unix 운영체제에서 사용 가능하면 사용하기 비교적 쉽다. 하지만 파일 디스크립터의 수를 FD_SETSIZE로 최대 수가 정해져 있기 때문에 대규모 server-client에서는 사용하기 힘들다. 또한 매번 모든 파일 디스크립터를 커널로 복사하기 때문에 연결되어 있는 개수가 많을수록 성능이 저하된다.

이와 반대로 epoll의 경우 select와 비슷하지만 최대 파일 디스크립터의 수가 정해져 있지 않기 때문에 대규모 server-client에서 적합하다. 또한 커널 내부에서 직접 관리하기 때문에 매 호출마다 파일 디스크립터를 복사할 필요가 없어 select에 비해 성능저하가 적다. Select의 경우 대부분의 Unix 운영체제에서 사용가능 했지만 epoll은 리눅스에서만 사용할 수 있다.

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

  ✓ Master Thread의 Connection 관리

  Open_listenfd를 통해 소켓을 생성하고 해당 소켓을 client에서 연결 요청을 받을 수 있는 수신 대기 상태로 만들어준다. 그 다음 accept를 통해 client가 요청을 할 때까지 대기하고 요청이 온다면 새로운 소켓 파일 디스크립터를 생성해 해당 파일 디스크립터를 통해 연결 요청한 client와 정보 교환을 할 수 있도록 한다. 이때 accept하는 부분은 while문으로 묶어 계속 client와 연결될 수 있도록 한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  sbuf라는 스레드 풀을 생성하고 초기화를 해준다. 그 다음 sbuf의 수 만큼 thread를 미리 생성해준다. Master thread에서 client의 연결 요청을 받아 accept한다면 sbuf에 연결한 파일 디스크립터를 넣어준다. 각 thread에서는 sbuf에서 제거하기 위해 대기하고 있는데 이렇게 들어온다면 생성한 thread 들 중 하나는 sbuf에서 파일 디스크립터를 제거해 연결한 client와 소통하게 된다. 각 thread는 pthread_detach를 통해 자원이 자동으로 회수되도록 한다.

- **Task3 (Performance Evaluation)**

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

  측정방법은 모두 동일하다. gettimeofday함수를 사용해 client와 연결하기 전에 starttime으로 시작하고 client와 연결이 전부 끊길 때 endtime으로 해 둘의 차이로 시간을 측정했다.

  이 실험을 통해 측정하고자 하는 metric은 response time과 throughput이다. Response time은 클라이언트가 요청을 보내고 응답을 받기까지 걸리는 시간으로서 client입장에서 정보를 server로 보냈을 때 정보를 받는데 체감시간을 의미하고 throughput은 server입장에서 들어오는 명령어의 처리 능력을 보여주는데 탁월하기 때문에 두 가지에 대해 측정을 할 것이다.

  이 두 가지 metric을 측정하기 위한 configuration은 다음과 같다. Event-based approach, Mutex, Readers-First Semaphore에 대해 측정을 할 것인데 client의 수를 10, 20, 50, 100, 200, 500, 1000으로 점점 늘려갈 것이다. 1000까지 한 이유는 Event-based approach에서 사용한 FD_SET_SIZE가 최대 1024기 때문에 1000으로 제한한 것이다.

  측정을 할 때 다음과 같은 세 가지 시나리오로 나누어서 성능을 측정할 것이다.

  1. 랜덤 명령어

     다양한 명령어를 무작위로 실행한다. 명령어가 어느 한 쪽으로 치우쳐지지 않고 일관된 성능을 보여주는 지 알 수 있다.

  2. write 명령어(buy, sell)

     Write 명령어만 주어진 경우다. 쓰기 작업이 많은 상황에서의 성능을 평가할 수 있다.

  3. Read 명령어(show)

     Read 명령어만 주어진 경우다. Readers-first semaphore는 읽기를 동시에 진행할 수 있기 때문에 다른 방법과 어느 정도 다른지 평가할 수 있다.

  측정을 할 때는 인터넷 상태, random 명령어시 일관되지 않은 명령어 같은 변수들이 존재하기 때문에 5번을 측정해 평균값으로 비교를 할 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

이전 알고리즘 수업에서 n이 늘어날 경우에 실행 시간이 어떻게 변하는지에 대해 측정한 적이 있었는데 n이 2배 3배 늘어날 때마다 실행시간은 기하급수적으로 증가했었기 때문에 client개수가 늘어날 때 실행시간은 기하급수적으로 증가할 것으로 예상된다.

Mutex, readers first semaphore는 thread based인 것과 달리 event based approach의 경우 동작들을 parallel하게 처리하지 않기 때문에 thread based인 세 가지 방식보다 느릴 것으로 예상되며 수업시간에서 배운바에 의하면 mutex를 사용한 경우가 가장 빠르고 그 다음 semaphore 그 다음이 event based순으로 빠를 것으로 예상이 된다. 그렇기 때문에 처리량의 경우는 응답시간이 짧은 경우에 처리량이 더 많을 것으로 예상이 된다.

명령어를 어떻게 주냐에 따라 또 결과가 달라질 것 같은데 데이터를 쓰는 경우 데이터를 레지스터에 불러오고 데이터 처리과정을 거친다음 다시 메모리에 올리기 때문에 write 명령어가 read 명령어보다 느릴 것으로 예상된다.

Random하게 명령어를 주는 경우와 write명령어만 주는 경우는 비슷할 것으로 예상되지만 read 명령어만 주는 경우 readers first semaphore에서는 read 명령어를 thread끼리 막지 않고 동시에 진행할 수 있도록 하기 때문에 이 부분에서는 reader first semaphore가 다른 방식들 보다 훨씬 빠를 것으로 예상된다.

C. 개발 방법

일단 공통적으로 주식의 data는 tree형태로 관리되고 data는 txt파일로부터 가져온다. 그렇기 때문에 server에서 client와 소통하기 전에 txt파일로부터 읽는 과정이 필요하다. fopen으로 연 다음 한 줄씩 읽어주고 각 줄로부터 얻은 정보는 tree에 넣어준다. 이를 위해 tree에 넣어주는 함수가 필요하다. 그리고 buy, sell, show 각각은 tree를 traverse하기 때문에 이를 위한 함수도 필요하다. 각각은 처리하는 것이 다르기 때문에 함수는 각각 만들어줘야 한다. Client와 소통이 다 끝난 다음 tree구조의 정보를 txt파일에 update해줘야 하므로 tree를 traverse하면서 정보들을 한 줄 씩 입력하는 함수도 필요하다.

Task 1의 경우 event_based로 main함수에서 client로부터의 연결 요청을 처리하고

어떤 client가 어떤 명령어를 요청했는지 알기 위해 pool 구조체가 필요하다. 처음에 client의 요청을 받기 위해 listenfd를 만들어야 한다. Listenfd를 만든 다음 pool 구조체에 연결을 해 해당 파일 디스크립터를 pool 구조체의 clientfd 배열에 저장하고 read_set에 해당 파일 디스크립터를 저장해 select함수를 불렀을 때 해당 파일 디스크립터에 요청이 들어왔는지 확인을 할 수 있도록 한다. 이후 select 함수를 해줬을 때 listenfd 부분에 요청이 들어왔다면 accept를 해주고 해당 client와 연결된 file descriptor 또한 pool구조체에 넣어 연결한 client와 소통을 할 수 있도록 한다. select함수를 해줬을 때 client쪽에서 뭔가 왔다면 어떤 client에서 왔는지 확인하고 해당 client에서 보낸 명령어를 읽은 뒤 명령어에 맞는 동작을 수행하여 해당 client에 다시 정보를 보내는 식으로 정보교환을 하게 된다.

Task 2의 경우 thread-based로 master thread에서 client로부터의 연결 요청을 처리한 다음 client와 상호작용은 thread가 하게한다. Worker thread pool을 만들기 위해 sbuf구조체가 필요하다. Sbuf 구조체를 다루기 위해 여러 함수가 필요한데 먼저 sbuf를 초기화 해주는 함수, sbuf를 사용한다음 free해주는 함수, sbuf안에 connfd를 넣어주는 함수, sbuf안의 connfd를 빼주는 함수가 필요하다. 먼저 thread 동적 생성시 오버헤드를 최소화하기 위해 미리 thread들을 생성해준다. 그 다음 sbuf구조체에는 master thread가 client의 연결 요청을 받고 만든 connfd를 넣어준다. 이렇게 넣어주면 미리 만들어둔 thread중 하나가 connfd를 하나 remove를 해 해당 thread와 client가 통신하게 된다. 이때 통신할 때 synchronization시 read우선하도록 하였는데 show의 경우 read이므로 이때는 pthread_rwlock_rdlock을 통해 read끼리는 lock하지 않고 write할때만 lock하도록 한다. Read가 끝나면 pthread_rwlock_unlock으로 lock을 해제한다. Buy와 sell의 경우 ID가 일치하는 것을 찾았을 때 buy의 경우 남아있는 주식 수를 확인하고 명령어 반영, sell의 경우 명령어 반영할 때만 lock을 해주면 된다. 이때는 write이기 때문에 해당 명령어 수행시 pthread_rwlock_wrlock을 통해 write하는 동안 다른 기타 동작들을 하지 못하도록 하고 write가 끝나면 pthread_rwlock_unlock으로 lock을 풀어주는 식으로 synchronization문제를 해결하였다.

3. 구현 결과

Task1과 task2는 client와 연결하여 처리하는 방식만 다를 뿐 하는 동작은 똑같다. 먼저 client의 요청을 받기 전에 stock.txt에 들어있는 data를 tree에 저장한다. 이

후 main함수에서 client가 연결 요청하는 것마다(최대 연결개수 이하로만) 받아주고 client가 show를 요청하면 tree에 저장되어 있는 정보들을 traverse해 buf에 저장한 다음 client한테 보내준다. Buy를 요청하면 traverse를 통해 요청한 ID를 찾고 사고 싶은 개수보다 남은 개수가 작으면 Not enough stock left를 buf에 저장한 다음 client한테 보내고 그렇지 않다면 산 개수만큼 남아있는 개수에서 뺀 다음 buf에 [buy] success를 저장해 client한테 보낸다. Sell을 요청하면 traverse를 통해 요청한 ID를 찾고 판 개수만큼 남아있는 개수에서 더한 다음 buf에 [sell] success를 저장해 client한테 보낸다. Client가 종료되면 server는 tree에 저장되어 있는 주식 data를 stock.txt에 저장한다.

4. 성능 평가 결과 (Task 3)

측정은 gettimeofday를 사용하여 측정하였으며 server를 키고 client와 연결하기 전부터 시작하여 모든 client와의 연결이 끊어질 때까지의 시간을 측정하였다. 또 측정시간의 경우 측정시간끼리의 차이가 크지 않아 그래프를 그릴 시 sec가 아닌 milisecond 단위로 하였다. 이때 각 client별로 주어진 명령어의 수는 10개다.

Response time

Random 명령어

Event_based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	10.176	10.179	10.290	10.130	10.209	10.197
20	10.163	10.257	10.161	10.171	10.177	10.186
50	10.200	10.209	10.187	10.216	10.230	10.208
100	10.255	10.240	10.284	10.359	10.254	10.278
200	11.217	11.237	11.213	11.236	11.392	11.259
500	13.239	13.271	13.221	13.222	13.274	13.245
1000	13.521	13.316	13.485	13.273	13.260	13.371

Mutex

Client 수	1차	2차	3차	4차	5차	평균
10	10.111	10.110	10.123	10.107	10.127	10.116

20	10.111	10.136	10.149	10.121	10.153	10.134
50	10.193	10.174	10.169	10.178	10.164	10.176
100	10.211	10.179	10.196	10.176	10.219	10.196
200	11.158	11.178	11.216	11.249	11.216	11.203
500	13.197	11.569	11.618	11.494	11.503	11.876
1000	13.283	13.215	13.215	13.240	13.243	13.239

Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	10.158	10.191	10.149	10.181	10.123	10.16
20	10.183	10.160	10.183	10.172	10.180	10.176
50	10.211	10.231	10.211	10.189	10.220	10.212
100	10.289	10.282	10.279	10.253	10.313	10.283
200	11.242	11.267	11.205	11.205	11.261	11.236
500	13.259	11.534	11.657	13.310	11.538	12.26
1000	13.432	13.270	13.337	13.279	13.301	13.324

Write 명령어(buy, sell)

Event-based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	10.200	10.171	10.232	10.180	10.198	10.196
20	10.218	10.220	10.266	10.239	10.328	10.254
50	10.280	10.285	10.225	10.236	10.304	10.266
100	10.343	10.297	10.348	10.299	10.324	10.322
200	11.242	10.337	11.293	11.308	11.239	11.084
500	11.652	11.697	11.645	13.331	13.280	12.321
1000	13.308	13.296	13.416	13.337	13.309	13.333

Mutex

Client 수	1차	2차	3차	4차	5차	평균
10	10.145	10.133	10.120	10.143	10.152	10.139

20	10.152	10.127	10.157	10.153	10.182	10.154
50	10.161	10.175	10.210	10.235	10.190	10.194
100	10.234	10.239	10.218	10.244	10.243	10.236
200	11.245	11.205	11.214	11.195	11.200	11.212
500	13.235	11.494	11.491	11.512	11.499	11.846
1000	13.230	12.161	13.322	13.241	13.297	13.05

Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	10.127	10.124	10.129	10.125	10.135	10.128
20	10.134	10.153	10.125	10.160	10.132	10.141
50	10.153	10.156	10.149	10.167	10.151	10.155
100	10.259	10.247	10.200	10.213	10.220	10.228
200	11.233	11.301	11.221	11.195	11.312	11.252
500	13.339	11.513	11.650	11.530	11.566	11.92
1000	13.251	13.256	13.213	13.251	13.3107	13.2563

Read 명령어(show)

Event-based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	10.153	10.149	10.161	10.170	10.124	10.151
20	10.177	10.172	10.170	10.146	10.285	10.19
50	10.266	10.212	10.193	10.173	10.210	10.211
100	10.304	10.309	10.341	10.346	10.288	10.318
200	12.117	11.340	11.261	11.292	11.284	11.459
500	11.694	13.262	13.288	11.644	13.242	12.626
1000	13.336	13.361	13.318	13.339	13.375	13.346

Mutex

Client 수	1차	2차	3차	4차	5차	평균
10	10.113	10.114	10.141	10.142	10.122	10.126

20	10.120	10.116	10.130	10.146	10.144	10.131
50	10.172	10.168	10.176	10.181	10.165	10.172
100	10.202	10.204	10.244	10.193	10.240	10.217
200	11.176	11.206	11.212	11.207	11.206	11.201
500	11.478	13.187	13.174	11.487	11.467	12.159
1000	13.259	13.213	13.227	13.222	13.235	13.231

Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	10.144	10.129	10.128	10.128	10.117	10.129
20	10.136	10.144	10.173	10.157	10.135	10.149
50	10.150	10.140	10.161	10.181	10.173	10.161
100	10.219	10.215	10.188	10.199	10.189	10.202
200	11.210	11.196	11.183	11.210	11.172	11.194
500	13.288	11.556	11.502	13.194	11.496	12.207
1000	13.289	13.247	13.245	13.289	13.243	13.263

Throughput

Random 명령어

Event-based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	9.807	9.823	9.717	9.871	9.794	9.802
20	19.677	19.498	19.682	19.663	19.650	19.634
50	49.017	48.972	49.081	48.942	48.873	48.977
100	97.508	97.652	97.234	96.532	97.521	97.289
200	178.295	177.979	178.350	177.997	175.560	177.636
500	377.64	376.742	378.169	378.137	376.653	377.468
1000	739.565	750.955	741.550	753.391	754.104	747.913

Mutex

Client 수	1차	2차	3차	4차	5차	평균

10	9.890	9.890	9.878	9.893	9.874	9.885
20	19.778	19.730	19.705	19.760	19.698	19.734
50	49.049	49.140	49.167	49.122	49.191	49.134
100	97.933	98.239	98.070	98.264	97.849	98.071
200	179.230	178.919	178.309	177.788	178.312	178.512
500	378.864	432.186	430.350	434.990	434.645	422.207
1000	752.817	756.707	756.681	755.272	755.113	755.318

Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	9.843	9.812	9.852	9.822	9.877	9.841
20	19.639	19.683	19.640	19.661	19.645	19.654
50	48.965	48.870	48.966	49.069	48.921	48.958
100	97.184	97.249	97.280	97.527	96.964	97.241
200	177.894	177.498	178.489	178.482	177.593	177.991
500	377.088	433.473	428.893	375.655	433.316	409.685
1000	744.441	753.554	749.768	753.023	751.779	750.513

Write 명령어(buy, sell)

Event-based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	9.803	9.831	9.773	9.822	9.805	9.807
20	19.571	19.568	19.481	19.532	19.363	19.503
50	48.636	48.614	48.895	48.846	48.521	48.702
100	96.675	97.113	96.633	97.096	96.852	96.874
200	177.892	176.398	177.094	176.854	177.939	177.235
500	429.081	427.437	429.363	375.038	376.485	407.481
1000	751.390	752.051	745.337	749.787	751.359	749.985

Mutex

Client 수	1차	2차	3차	4차	5차	평균

10	9.856	9.868	9.880	9.858	9.849	9.862
20	19.699	19.748	19.690	19.698	19.642	19.695
50	49.205	49.137	48.971	48.849	49.064	49.045
100	97.707	97.664	97.858	97.610	97.621	97.692
200	177.843	178.491	178.344	178.642	178.557	178.375
500	377.785	434.980	435.108	434.314	434.816	423.401
1000	755.800	822.244	750.593	755.190	752.009	767.167

Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	9.874	9.877	9.871	9.875	9.866	9.873
20	19.734	19.697	19.752	19.684	19.738	19.721
50	49.243	49.231	49.263	49.174	49.253	49.233
100	97.468	97.582	98.038	97.907	97.839	97.767
200	178.042	176.971	178.233	178.637	176.798	177.736
500	374.819	434.288	429.153	433.629	432.279	420.834
1000	754.607	754.344	756.777	754.624	751.269	754.324

Read 명령어(show)

Event-based Approach

Client 수	1차	2차	3차	4차	5차	평균
10	9.848	9.852	9.841	9.832	9.877	9.85
20	19.651	19.661	19.664	19.710	19.445	19.626
50	48.701	48.959	49.048	49.146	48.969	48.965
100	97.047	97.002	96.701	96.648	97.191	96.918
200	165.044	176.354	177.104	177.104	177.241	174.569
500	427.562	377.011	376.268	429.380	377.578	397.56
1000	749.817	748.399	750.862	749.627	747.617	749.264

Mutex

Client 수	1차	2차	3차	4차	5차	평균

10	9.888	9.886	9.860	9.859	9.879	9.874
20	19.761	19.770	19.742	19.710	19.714	19.739
50	49.153	49.173	49.134	49.110	49.186	49.151
100	98.013	97.991	97.611	98.105	97.653	97.875
200	178.950	178.467	178.370	178.451	178.460	178.54
500	435.611	379.138	379.531	435.240	436.011	413.106
1000	754.196	756.795	755.993	756.140	755.569	755.739

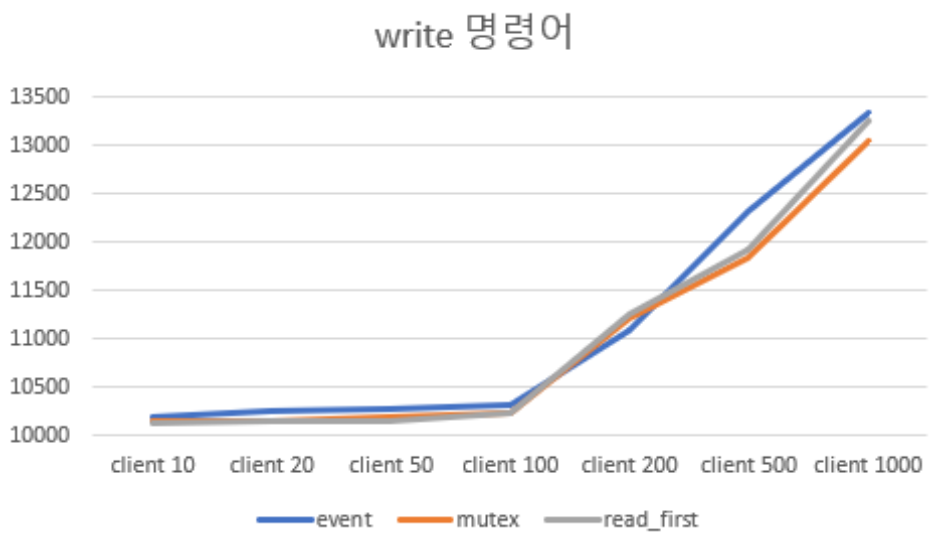
Readers-first Semaphore

Client 수	1차	2차	3차	4차	5차	평균
10	9.857	9.872	9.873	9.872	9.884	9.872
20	19.730	19.714	19.659	19.690	19.732	19.705
50	49.256	49.307	49.205	49.109	49.147	49.205
100	97.856	97.886	98.148	98.039	98.137	98.013
200	178.409	178.621	178.834	178.399	179.017	178.656
500	376.251	432.643	434.703	378.937	434.928	411.492
1000	752.446	754.860	754.990	752.48	755.112	753.978

Response time

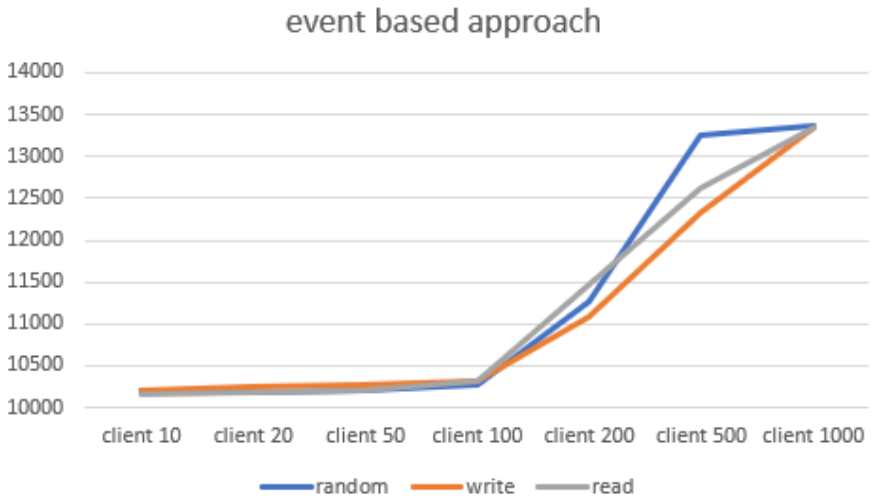
client수가 늘어남에 따라 시간도 같이 늘어났는데 예상했던 바와 달리 Response time이 기하급수적으로 증가하지는 않았다. 알고리즘의 경우는 sequential하게 처리가 되기 때문

에 n의 증가에 따른 증가량이 기하급수적으로 늘어나는 반면 서버의 경우 서버 자원을 병렬적으로 처리하기 때문에 n의 증가에 따른 response time이 크게 증가하지는 않은 것 같다. 각 방식에 따른 response time의 경우는 수업시간에 배운 대로 mutex가 가장 빨랐으며 그 다음이 read_first semaphore, 그 다음이 event-based순으로 빨랐다. Client를 소규모로 받는 server의 경우 어느 방식을 해도 상관이 없지만 대규모 client를 받는 server의 경우 mutex나 read_first semaphore방식을 사용하는 server가 더 효율적일 것으로 보인다. mutex가 read_first semaphore보다 좀 더 빨랐다. 이는 mutex의 경우 단순한 lock, unlock 메커니즘을 사용하지만 semaphore의 경우 여러 개의 스레드의 동시 접근을 가능하게 하는 더 복잡한 메커니즘을 사용했기 때문에 semaphore가 오버헤드가 더 커서 발생한 현상으로 보인다. 비교적 간단한 동작을 원하는 경우에는 mutex를 사용하는 것이 옳아 보이고 여러 스레드를 동시에 특정 동작을 가능하게 하는 등의 복잡한 작업을 하는 server는 semaphore를 사용하는 것이 옳바른 것으로 보인다.



write명령어 또한 위의 random 명령어에서 설명한 대로 나온다. 한 가지 다른 점은 중간에 client 200일 때 event-based approach가 더 빠르게 나온다. 이는 write할 때마다 thread의 경우 lock을 하면서 context switching 오버헤드가 두드러지게 발생하여 생기는 문제로 보인다. Client가 더 커지면 다시 mutex와 read_first semaphore가 더 빠른데 이때는 context switching으로 인한 overhead보다 parallel하게 실행하면서 생기는 영향이 더 크기 때문인 것으로 보인다.

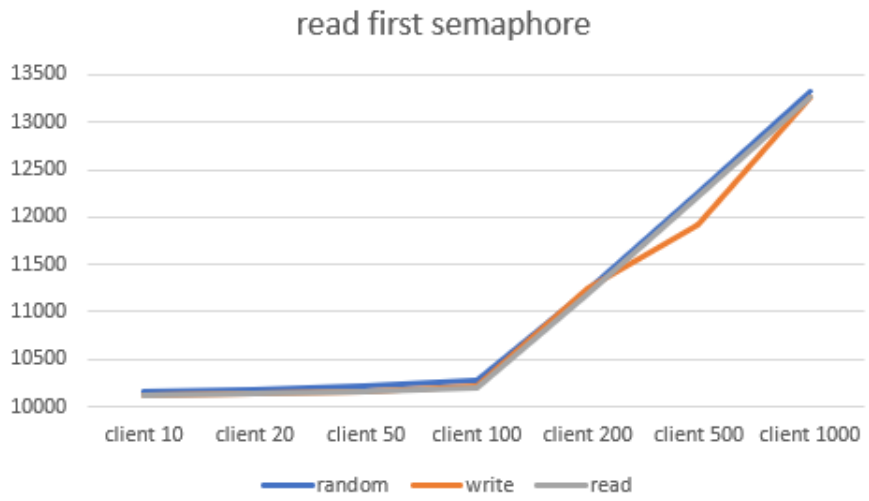
read명령어 또한 위의 random 명령어에서 설명한 대로 나온다. read명령어의 경우 read-first semaphore가 mutex보다 빠를 것으로 예상했는데 그러지는 않았다. Read의 경우 read first semaphore가 동시에 read 명령어를 처리할 수 있기 때문에 더 빠를 것이라고 생각했는데 아마 lock할 시 context switching 오버헤드가 mutex보다 read_first semaphore가 더 크기 때문에 semaphore가 더 느리게 측정이 된 것 같다. 하지만 이전 명령어들에서 차이를 꽤 보였던 것과 달리 read_first semaphore의 response time이 mutex와 거의 동일하게 빠른 것으로 보인다. 이는 read-first semaphore의 경우 Read 명령어가 주어지면 parallel하게 명령어들을 수행하기 때문에 이전 명령어와 달리 더 빨라진 것으로 보인다.



Read가 write보다 빠를 것으로 예상했지만 예상과는 달랐다. 이에 대한 이유를 생각해보

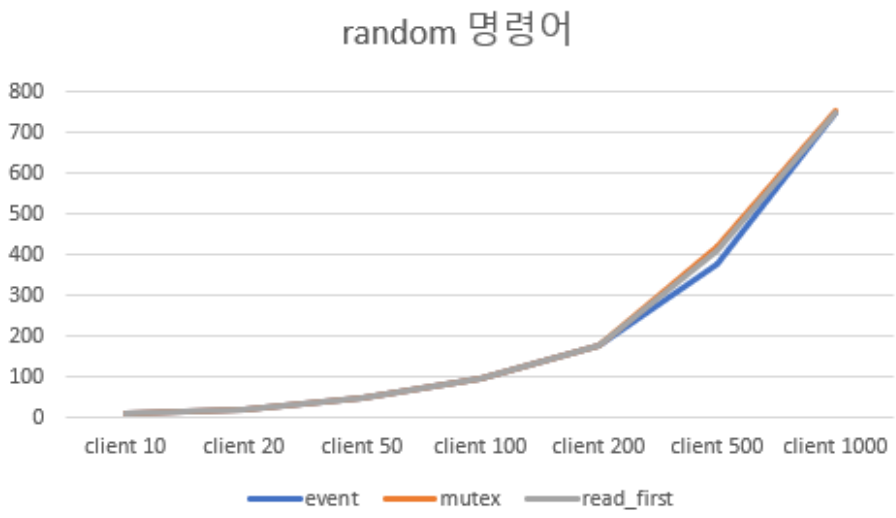
니 주식 data가 tree구조로 되어있다는 것을 간과했기 때문이었다. tree에서 read를 할 때는 모든 node를 읽어들여 buffer에 해당 내용을 누적해야 하지만 write의 경우는 원하는 노드를 찾으면 이후 tree를 traverse할 필요가 없기 때문에 모든 경우에 있어서 write가 read보다 빠를 수 밖에 없었다. 초반에는 write가 read보다 느린데 이는 write할 때 데이터를 register에 불러들이고 계산과정을 거친 뒤 다시 메모리에 저장하기 때문에 이 과정이 tree traverse하는 것보다 시간에 더 영향을 끼쳤기 때문이라고 생각한다. client수가 일정 이상되면 random이 가장 느린데 이는 read와 write가 혼합되어 캐시 효율이 저하됐기 때문인 것 같다.

Mutex 또한 event based approach와 비슷하다. 하지만 한 가지 다른 점이 있다면 random이 가장 느렸던 event based approach와 달리 read가 가장 느리다. 이 이유를 살펴보니 mutex의 경우 lock을 하면서 명령어들을 처리하는데 lock의 획득과 해제 비용에 tree traverse하는 시간까지 겹쳐지니 read가 가장 느리게 나온 것 같다.



Read first semaphore의 경우도 lock을 사용하긴 하지만 read를 수행하는 중에는 read만 parallel하게 사용할 수 있게 하니 mutex와 다르게 random하게 명령어가 주어진 경우보다 read만 주어진 경우가 response time이 더 빨라진 것으로 보인다.

Throughput



처리량은 처리 개수/ 응답시간으로 응답시간이 빠르면 빠를수록 처리량 또한 크다고 예상을 했었는데 예상과 같이 mutex, read_first semaphore, event_based approach순으로 처리량이 많았다. Client의 개수가 적을 때는 client가 2배 늘어난 만큼 그에 따라 처리량도 비슷하게 증가하였지만 client가 개수가 늘면 늘수록 처리량의 증가량은 2배에 못 미치게 증가했다. 이는 여러 요인에 의해 처리량 증가가 둔화되는 것으로 보인다. 그 이유는 다음과 같다. 클라이언트들은 동시에 cpu, 메모리와 같은 시스템 자원을 소비할텐데

client가 늘어나면서 각 client가 사용할 수 있는 시스템 자원의 양이 상대적으로 줄어들어 처리량이 둔화됐다. 이외에도 client가 증가함에 따라 동기화나 context switching시에 overhead가 발생하기 때문에 처리량이 client수가 증가한 만큼 증가하지 않은 것으로 보인다.

Write 명령어의 경우 random 명령어가 주어졌을 때와 마찬가지로 client수가 2배 증가하면 처리량의 증가는 2배에 못 미쳤다. 처리량 순서를 보니 mutex, read_first, mutex 순인데 read_first와 mutex가 event_based approach보다 빠른 것은 thread를 통해 병렬적으로 처리하기 때문에 처리량이 더 높은 것으로 보이고 mutex가 read_first semaphore보다 빠른 것은 mutex를 사용해 lock하는 것이 semaphore를 사용해 lock하는 것보다 lock하는 과정이 좀 더 단순해 더 빠른 것으로 보인다.

read 명령어의 경우 random 명령어가 주어졌을 때와 마찬가지로 client수가 2배 증가하면 처리량의 증가는 2배에 못 미쳤다. Read의 경우 처리량에 있어서 방식의 차이가 거의 나지 않는데 이는 캐시의 효율성, 동기화 오버헤드 때문인 것으로 보인다. 읽기 작업의 경우 데이터의 상태를 변경하지 않기 때문에 대부분의 동기화 메커니즘에서 상대적으로 가벼운 동기화가 필요하기 때문에 이로 인한 오버헤드가 적게 발생한다. 또 캐시 메모리에서의 히트율이 높은 편이기 때문에 읽기 작업은 동기화 방식에 큰 영향을 받지 않는 것 같다. 이런 요인들로 인해 방식에 따른 처리량 차이가 크게 두드러지지 않는 것으로 보인다.