Multicore Programming Project 2

담당 교수 : 최재승

이름 : 김태곤

학번 : 20191583

1. 개발 목표

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

여러 client들의 동시 접속 및 서비스를 위한 Concurrent stock server를 구축한다. 주식 정보를 저장하고 여러 client들과 통신하여 주식정보 List, 판매, 구매 등의 동작을 수행할 수 있어야 한다. Event driven Approach와 Thread based Approach를 사용하여 여러 client와 통신하며 concurrent한 stock server를 구현하는 것이 이번 프로젝트의 목표이다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   하나의 process를 가지고 여러 개의 client를 처리한다. 여러 개의 connection fd를 생성하고, 각각의 client는 fd를 trigger한다. 서버는 fd들을 monitoring하던 select에 의해 동작을 시작한다.

2. Task 2: Thread-based Approach

   각 client의 요청을 별도의 thread로 실행한다. 서버는 여러 개의 thread를 생성하고, 각각의 client가 접속할 때 Master thread에서 client의 연결을 한다. 이후 연결 된 각각의 thread에서 client의 동작을 수행한다.

3. Task 3: Performance Evaluation

   Event driven와 Thread based 방식을 사용하여 여러 조건의 elapse time을 측정하고 분석한다. 시간당 client 처리 요청 개수를 비교하여 동시처리율을 비교해본다. 또한, Client의 요청 타입에 따른 동시 처리율 변화를 분석해 워크로드에 따른 분석을 진행한다. Event driven 방식은 하나의 control flow와 address sapce를 가지고 overhead가 거의 발생하지 않지만, 멀티 core를 가진 cpu에 대해 성능을 온전하게 활용하지 못할 것으로 예상된다. Thread based 방식은 이와 반대로 멀티 core를 잘 활용한 결과가 나올 것으로 예상한다.

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

서버는 하나의 listenfd를 가지고 여러 개의 clientfd와 연결이 이루어진다. 여러 개의 이벤트 발생 확인은 select 또는 epoll로 이루어지는데 본 프로젝트에서는 I/O multiplexing을 select방법으로 구현하였다. Select함수는 특정 fd집합에서 I/O 이벤트가 발생할 때까지 호출한 프로세스의 실행을 블록시킨다. 그리고 I/O 이벤트가 발생한 fd 중 하나 이상이 일정 조건을 만족하는 경우 select함수는 호출한 프로세스의 실행을 재개한다.

✓ epoll과의 차이점 서술

select와 epoll 모두 I/O multiplexing을 위한 메커니즘이지만 차이가 있다. Select는 예전에 만들어진 방법으로 많은 단점들을 포함하고 있다. Select는 모든 fd를 대상으로 반복문을 진행하기 때문에 느리다. 또한 함수 호출할 때마다 매번 운영체제에 관찰 대상에 대한 정보를 넘겨야 한다. 최대 동접자도 100면 안팍으로 많은 단점을 가지고 있는데, 이러한 단점을 해결하기 위해 나온 것이 epoll이다. Epoll함수는 상태변화의 확인을 위한, 전체 파일 디스크립터를 대상으로 하는 반복문이 필요 없다. 또한 함수 호출 시 관찰 대상의 정보를 매번 전달할 필요가 없다. 운영체제에 관찰 대상에 대한 정보를 딱 한번만 알려주고, 이후 변경이 있을 때에만 다시 알려준다.

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

✓ Master Thread의 Connection 관리

Main함수에서 Master Thread는 클라이언트의 연결 요청을 받고, 해당 연결을 처리하기 위한 Connection 관리를 담당한다., Aceept함수를 호출하여 연결을 수락하고, connfd에 연결된 fd를 저장한다. 이후 sbuf_insert 함수를 호출하여 connfd를 Worker Thread Pool인 sbuf에 추가한다. 이를 통해 Master Thread는 연결된 client와의 통신을 Worker Thread에게 할당하고, 다른 client

의 연결 요청을 처리할 수 있게 된다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

sbuf_t 구조체는 Worker Thread Pool을 나타낸다. sbuf_init, deinit, insert, revmoe를 통해 Worker Thread Pool을 초기화하고, 작업을 처리하도록 한다. sbuf_init은 구조체를 초기화한다. sbuf_insert는 Worker Thread Pool에 작업을 추가하고, sbuf_remove는 Worker Thread Pool에서 작업을 가져온다. 작성한 코드에서는 main함수에서 Worker Thread Pool을 생성하고, sbuf_insert함수를 통해 client의 연결 요청을 Worker Thread에게 할당한다. Worker Thread는 echo_cnt(강의자료에서는 숫자를 세는 함수였지만 본 프로젝트에서는 stock server의 client의 행동에 해당한다) 를 호출하여 처리하고, Close함수를 사용하여 연결을 종료한다.

- **Task3 (Performance Evaluation)**

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

Event driven 방식의 서버와 Thread based 방식의 성능을 비교하고자 한다. 먼저 동시 처리율에 대해 분석은 시간당 client 처리 요청 개수로 "총 client의 요청 수(client 수 * client 당 요청 수) / 요청을 모두 처리하는데 걸린 시간"으로 분석한다. Event driven 방식은 하나씩 처리되면서 동시에 처리되는 것처럼 보이고, thread based 방식은 concurrent하게 처리 되므로 동시처리율에 차이를 보일 것이다.

추가로 워크로드에 따른 분석을 한다. Client의 요청 타입(buy, sell, show)에 따른 동시 처리율 비교한다. Show는 단순히 데이터를 읽고 출력하는데 비해, buy와 sell은 데이터의 정보를 조작한다. 순차적으로 진행하는 event driven방식과 semaphore를 이용해 접근 순서를 관리하는 thread based 방식은 요청 타입에 따라 차이를 보일 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

두 방식의 동시처리율을 비교하면 client수가 작을 때는 event driven 방식이 좀 더 크고, client 수가 많을 때는 thread based방식이 좀 더 클 것으로 예상된다. Event driven 방식은 overhead가 적어 client 수가 적을 때는 thread based보다 성능이 좋을 것이다. 하지만 client 수가 증가하면 Event driven 방식은 하나씩 처리되면서 동시에 처리되는 것처럼 보이지만, thread

based 방식은 여러 thread가 concurrent하게 처리하므로 단위 시간 당 처리하는 client의 요청 수가 더 클 것으로 예상된다.

Client의 요청 타입에 대해 비교를 해보면, show는 buy와 sell에 비해 처리율이 더 낮을 것으로 예상된다. Show의 경우 주식 정보에 대한 노드에 한번씩 접근하여 출력하는 과정이 있으므로 실행 시간이 조금 더 걸릴 것이다.

좀 더 정확한 시간 측정을 위하여 multiclient.c파일에서 usleep(1000000); 이부분을 주석 처리하고 실행한다. 또한 정확한 측정을 위해 각각 9번의 반복 측정 후 중앙값에 해당하는 값으로 선택하였다.

C. 개발 방법

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

두 서버 모두 주식 관리 테이블을 동일하게 구현하였다. Item이라는 구조체를 선언하여 ID, left_stock, price, readcnt, *left,*right를 선언하였고, 추가로 thread based 방식에서는 mutex를 선언하였다. 이 구조체를 바탕으로 linked list를 이용하여 이진 트리 방식을 구현하였다. 먼저 main함수가 실행되면 stock.txt파일을 열어 미리 선언한 insert함수를 사용하여 이진 트리에 주식 정보를 집어넣는다. Insert함수는 주식 종목의 id가 이전 꺼보다 작으면 왼쪽 노드, 크면 오른쪽 노드에 배치하였다. 이후 이루어지는 과정은 두 서버 구조에 따라 차이가 난다.

Event Driven Approach

먼저 client와 관련된 정보와 I/O multiplexing에 필요한 상태를 저장하는 pool 구조체를 만들었다. 이후 반복문을 통해 Select 함수를 사용하여 I/O multiplexing을 수행한다. Select 함수는 ready_set에 있는 fd중 어떤 fd에서 이벤트가 발생했는지 확인하고, nready에 발생한 이벤트의 수를 저장한다. 이후 FD_ISSET을 통해 listenfd가 ready_set에 속해 있는지 확인하여 새로운 client의 요청을 확인한다. 이때 Accept함수를 사용하여 새로운 client와의 연결을 수락하고, connfd에 연결된 fd를 저장한다. 이후 add_client를 통해 pool에 추가한다. Check_client는 pool에 있는 모든 client와 관련된 이벤트를 검사하고 처리한다. Check_client 함수에서

client의 요청에 따른 show, buy, sell 과정이 이루어졌다. 읽어 들인 버퍼로부터 명령어가 show인 경우 show함수를 실행시켰다. Show함수는 전위 순회 방식으로 이진 트리로부터 정보를 읽어 지정된 버퍼에 저장하는 함수이다. 이 외에 buy나 sell 명령어가 올 경우 추가로 index와 주식 수를 읽어와 이에 맞는 action을 취해 주었다. 이후 Rio_writen을 통해 connfd에 버퍼를 전달해 최종적으로 client에 답을 주는 구조를 완성하였다.

Thread Based Approach

Thread Based Approach는 Event driven Approach와 구조가 다르다. Shared buffer를 구현하기 위해 교재에 나온대로 sbuf를 구현하였다. Main함수에서 파일을 읽어들인 후 sbuf를 초기화 시켜 주었다. Prethreaded 방식으로 구현하여 반복문을 진행하기 전 Pthread_create를 통해 thread를 미리 생성해두었다. 이후 incoming client가 있으면 shared buffer에 insert해주고, accept를 통해 connfd를 만들고 sbuf_insert를 통해 shared buf에 connfd를 집어넣었다. Thread함수는 각 thread가 하는 action을 담고 있다. 기존 교재의 코드를 바탕으로 작성하여 echo_cnt에 thread가 하는 행동이 정의되어 있다. 기본적인 과정은 Event driven 방식과 동일하나 Read-writer 문제로 show, buy, sell과정에서 약간의 차이가 있다. Show는 reader로 각 주식 종목에 접근할 때 그 종목에 대한 P(&mutex)를 사용하고 readcnt를 증가시킨다. 이후 readcnt가 1이면 P(&w)를 해주어 writer과정인 buy나 sell이 접근하지 못하도록 하였다. 이후 show와 관련된 작업을 해주고 위의 과정의 역순을 해주었다. Buy와 sell은 기존 Event driven에서 해주었던 과정에서 P(&w)와 V(&w)로 감싸주어 Readers-Writers 문제를 해결하였다.

3. 구현 결과

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

초기 프로젝트에서 요구하였던 Event driven Approach와 Thread based Approach 두가지 방식의 stock server를 모두 구현하였다. 단일 client의 요청도 성공적으로 수행할 수 있었고, multiclient의 요청도 두 서버 모두 concurrent하게 수행되는 것을 확인할 수 있었다.

4. 성능 평가 결과 (Task 3)

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

Multi client에 대한 성능평가는 기본적으로 셋팅 되어 있는 값을 바탕으로 진행하였다. MAX_CLINET는 100, ORDER_PER_CLIENT는 10, STOCK_NUM은 10, BUY_SELL_MAX는 10으로 설정하였다. 또한 thread based 방식에서 기본적으로 thread의 수를 100개로 지정하고 성능 평가를 진행하였다.

1) Event driven Approach와 Thread based Approach

Event driven	1	10	20	30	40	50	60	70	80
측정1	30440	30925	43686	55219	42887	48203	67243	88436	99645
측정2	23314	40706	32470	40017	43413	59872	80183	98968	97959
측정3	25647	39704	38850	47627	46687	53633	49664	79394	105095
측정4	28604	29185	39938	41198	40185	41446	62695	70175	97597
측정5	21747	39656	36182	41548	48207	45840	61851	103359	92933
측정6	14020	39953	37384	40126	49797	54394	75541	90253	84515
측정7	20709	37847	43880	35254	39534	71664	79903	69384	103797
측정8	20877	35911	41470	40420	53158	59441	81996	91598	94458
측정9	32858	31936	38466	36715	36318	51457	71643	86500	90948
측정10	12797	35750	37090	36230	47900	62198	62745	92619	124212
측정11	16894	31874	35953	41176	48978	60899	75106	86114	91679
중앙값	21747	35911	38466	40420	46687	54394	71643	88436	97597
동시 처리율	459.83354	2784.66208	5199.39687	7422.06828	8567.6955	9192.19032	8374.85867	7915.3286	8196.97327

Table 1 Event driven Approach

Figure 1 Event driven Approach

Thread based	1	10	20	30	40	50	60	70	80
측정1	24270	34687	35938	39974	40819	47600	40929	44913	60340
측정2	28379	32635	43936	36021	41546	43751	45495	49645	59575
측정3	28035	38745	47816	37538	40408	42553	47259	49013	52079
측정4	21324	35910	36789	43639	42859	43421	40804	47665	50291
측정5	27507	34824	36061	44342	40686	49484	47157	45744	60897
측정6	28970	28379	36662	40514	39850	44329	38574	47136	53640
측정7	28214	32812	44029	38736	48584	50742	41141	50671	44971
측정8	36171	39979	33486	35526	47517	46510	41932	57124	40900
측정9	28093	32575	33887	43585	40226	43720	46555	41388	46754
측정10	16642	40019	37430	41442	43505	41894	48682	50503	50589
측정11	28456	41105	35755	42122	48081	46254	45478	50594	47369
중앙값	28093	34824	36662	40514	41546	44329	45478	49013	50589
동시 처리율	355.96056	2871.58282	5455.23976	7404.84771	9627.88235	11279.298	13193.1923	14281.9252	15813.7144

Table 2 Thread based Approach

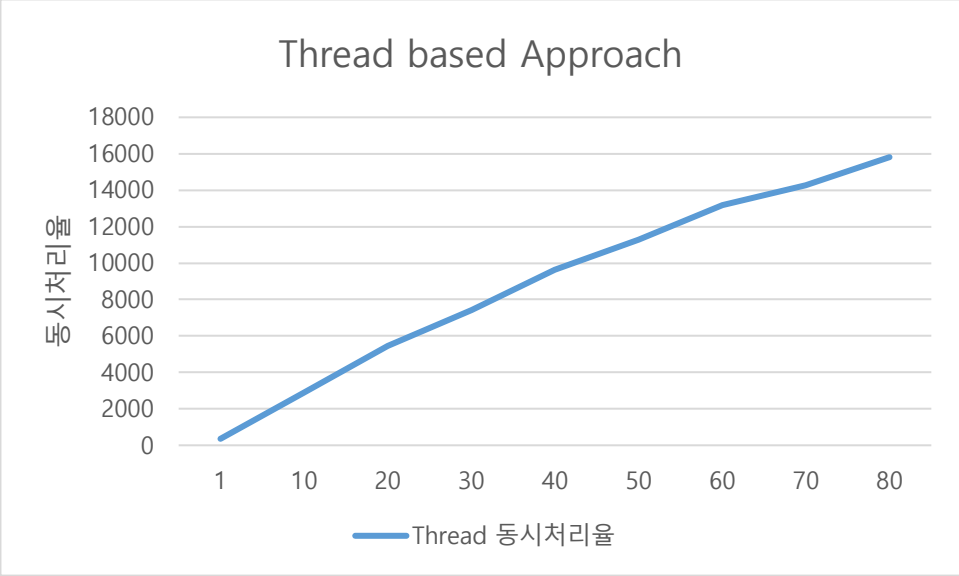

Figure 2 Thread based Approach

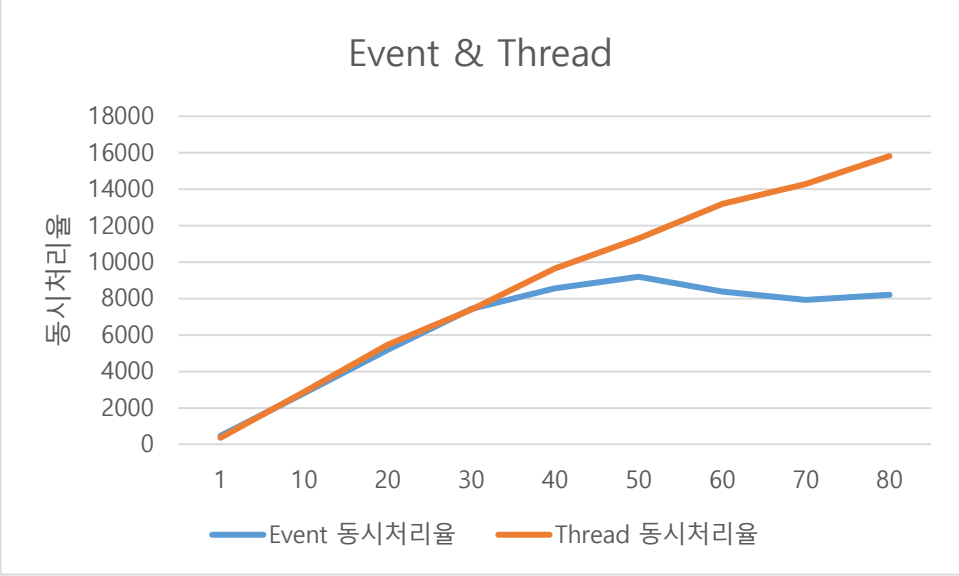

Figure 3 Event & Thread

2) show만 사용한 Event driven Approach와 Thread based Approach

Event-show	1	10	20	30	40	50	60	70	80
측정1	13145	32551	30673	31264	47632	65635	79364	79461	93505
측정2	24621	36975	38280	36297	34246	64084	64586	91501	91882
측정3	28512	39636	39871	51549	37770	56910	62957	61037	76726
측정4	23925	36516	36783	44788	51942	40790	69837	99115	97827
측정5	28658	31010	27849	41980	47798	70733	59139	87531	100778
측정6	16928	28969	43531	36955	36580	73505	61886	87335	92015
측정7	20299	34080	33003	51437	51907	43626	63931	75931	103791
측정8	20977	32012	36613	41680	36307	51652	68089	81387	108272
측정9	21599	29352	38310	41460	33126	55360	66766	79652	99675
측정10	24346	31736	41395	32407	43831	55598	71464	76447	78161
측정11	22965	28551	39777	36204	43870	57087	61046	80292	111398
중앙값	22965	32012	38280	41460	43831	56910	64586	80292	97827
동시 처리율	435.445243	3123.82856	5224.6604	7235.89001	9125.96108	8785.80214	9289.939	8718.17865	8177.70145

Table 3 Event driven Approach(show)

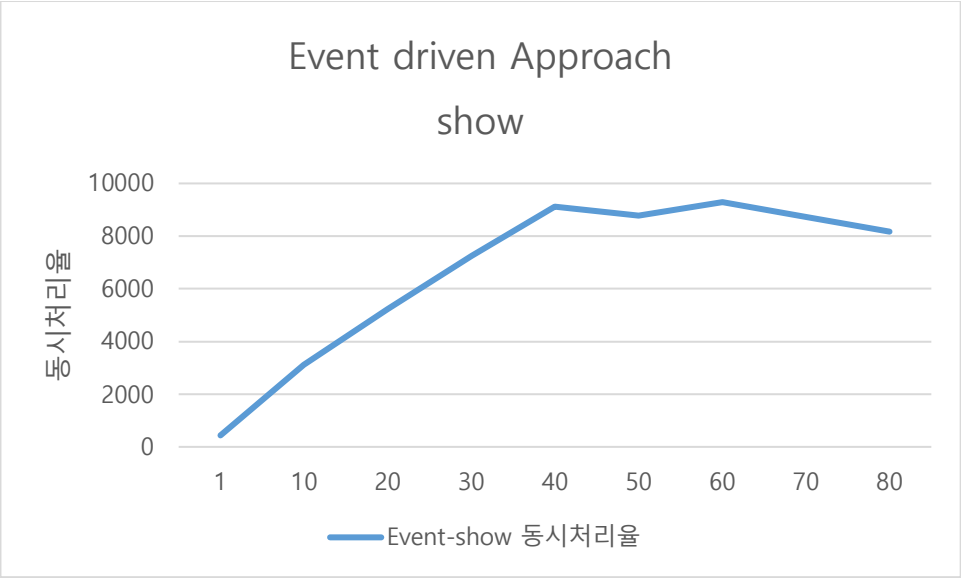

Figure 4 Event driven Approach(show)

Thread-show	1	10	20	30	40	50	60	70	80
측정1	31408	28503	33398	38008	42498	42697	41305	53106	49035
측정2	16828	32866	31005	51964	39541	53440	49452	51386	54849
측정3	20316	31933	38533	39284	37442	51102	48147	43605	44635
측정4	24793	31827	36071	37457	51680	48461	47204	54757	44783
측정5	24457	28428	39748	39421	39934	35692	41114	43841	46601
측정6	28488	28870	32361	40507	34984	48965	50583	47408	58460
측정7	17851	31771	33215	39906	44359	39714	46512	44958	58907
측정8	18820	39966	31132	43146	48481	41915	38084	46890	51900
측정9	14720	32807	33957	35300	38710	50828	40733	41392	51128
측정10	27444	32088	37322	35487	38980	43847	41894	43262	49103
측정11	30103	40018	41088	25927	36344	42010	44143	49897	61547
중앙값	24457	31933	33957	39284	39541	43847	44143	46890	51128
동시 처리율	408.880893	3131.5567	5889.80181	7636.69687	10116.082	11403.2887	13592.189	14928.5562	15647.0036

Table 4 Thread based Approach(show)

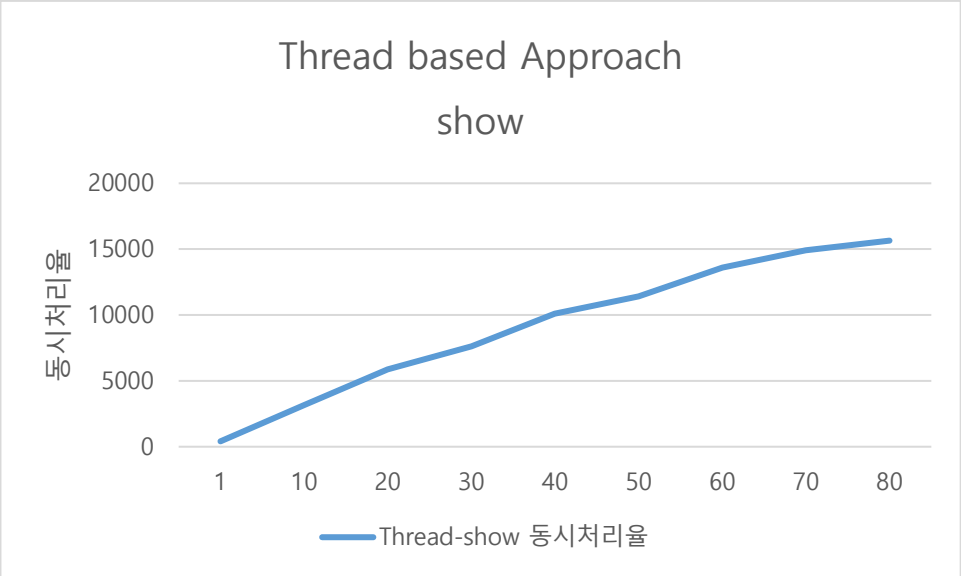

Figure 5 Thread based Approach(show)

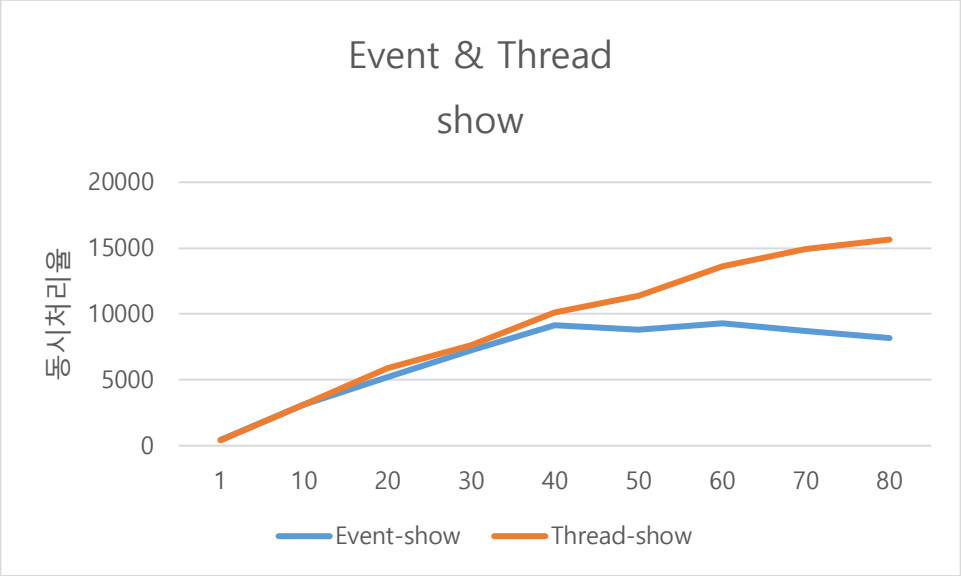

Figure 6 Event & Thread(show)

3) buy, sell만 사용한 Event driven Approach와 Thread based Approach

Event-buysell	1	10	20	30	40	50	60	70	80
측정1	31623	31645	32462	36229	42217	61072	69959	68017	86817
측정2	24690	33125	30493	29844	44555	55595	48619	68017	87432
측정3	24554	28018	39622	38117	47390	59561	65655	86026	82979
측정4	13623	52182	49372	30985	43526	60420	51572	63852	80862
측정5	24555	29312	33110	30088	43878	51785	61491	84834	83491
측정6	28016	32266	35471	41326	36610	76197	62154	67142	81925
측정7	28110	29579	40445	39176	40032	56794	68631	72682	84235
측정8	25262	27834	29499	44331	34294	57020	63588	74774	84089
측정9	32473	32575	48027	51615	36076	60195	61605	84376	95600
측정10	23433	27890	35546	33727	55519	54638	48349	64820	81949
측정11	12472	36561	30573	43072	37800	51657	55155	78644	90314
중앙값	24690	31645	35471	38117	42217	57020	61605	72682	84089
동시 처리율	405.022276	3160.05688	5638.40884	7870.50397	9474.8561	8768.85303	9739.4692	9630.99529	9513.7295

Table 5 Event driven Approach(buy, sell)

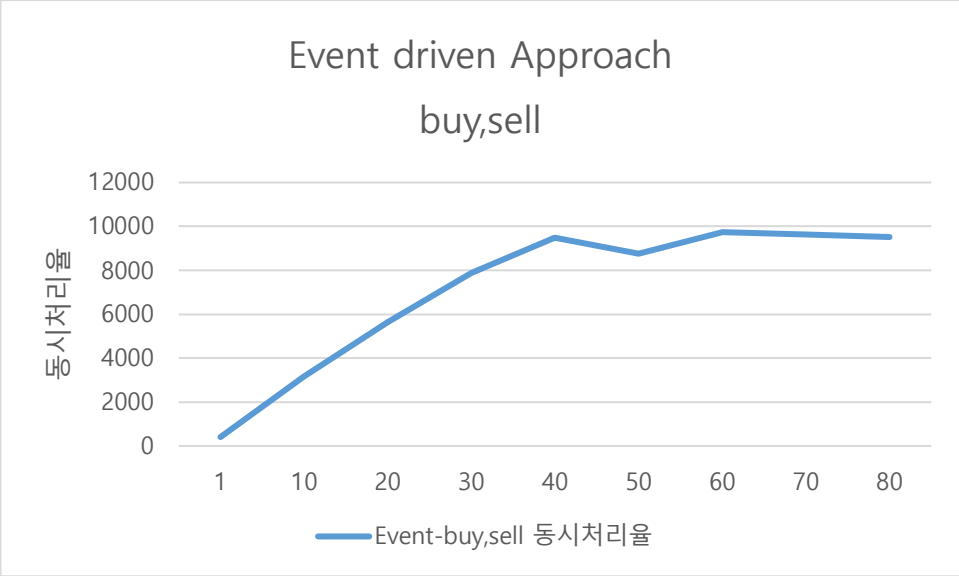

Figure 7 Event driven Approach(buy, sell)

Thread-buyse	1	10	20	30	40	50	60	70	80
측정1	26216	35500	27755	32472	43895	35114	42647	44395	44367
측정2	20234	32071	36326	32460	36180	39660	43714	43812	39022
측정3	28140	31829	29523	42550	41149	39807	43791	38624	36197
측정4	25547	32743	37881	41803	34807	37066	38363	44031	52976
측정5	16193	32674	32313	34219	35895	44855	37743	36791	43826
측정6	24674	28364	35820	39747	31064	42935	46133	36220	47775
측정7	28950	27886	48046	35537	37858	37401	36511	43324	46656
측정8	29036	36167	29201	36579	39518	35797	43899	52605	35494
측정9	24407	31805	33898	34709	39439	39806	38383	41305	55318
측정10	16732	28801	33346	31849	35359	40348	36780	44231	38918
측정11	15782	28208	29766	36124	24991	42216	47067	43897	47049
중앙값	24674	31829	33346	35537	36180	39806	42647	43812	44367
동시 처리율	405.284915	3141.78893	5997.72087	8441.90562	11055.832	12560.9205	14068.9849	15977.3578	18031.4197

Table 6 Thread based Approach(buy, sell)

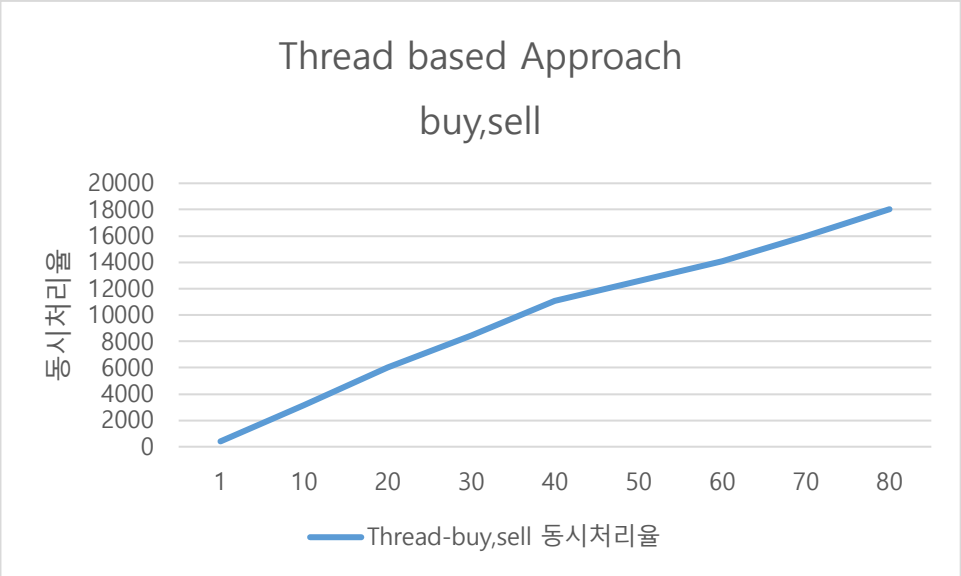

Figure 8 Thread based Approach(buy, sell)

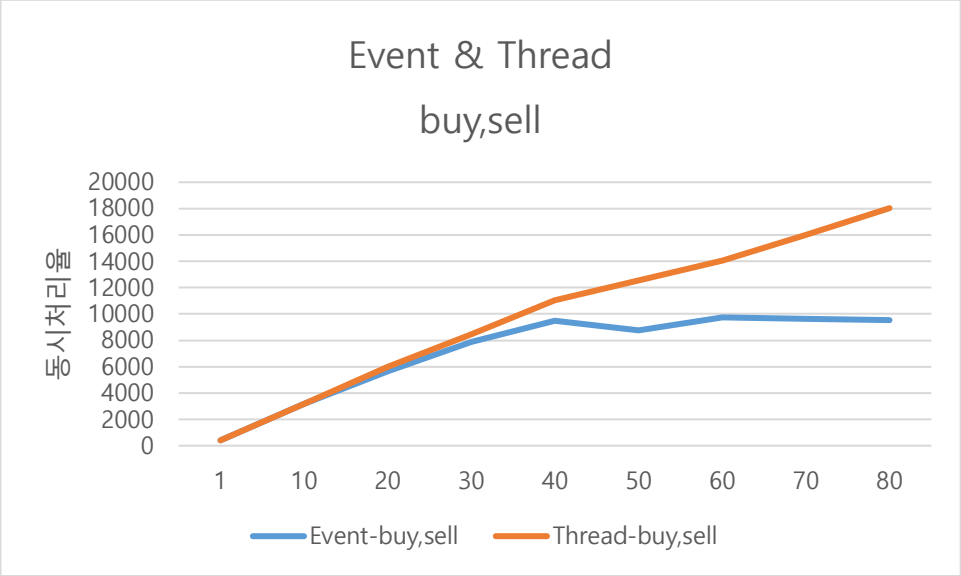

Figure 9 Event & Thread(buy, sell)

4) show와 buy,sell 명령어 처리에 따른 동시처리율 비교

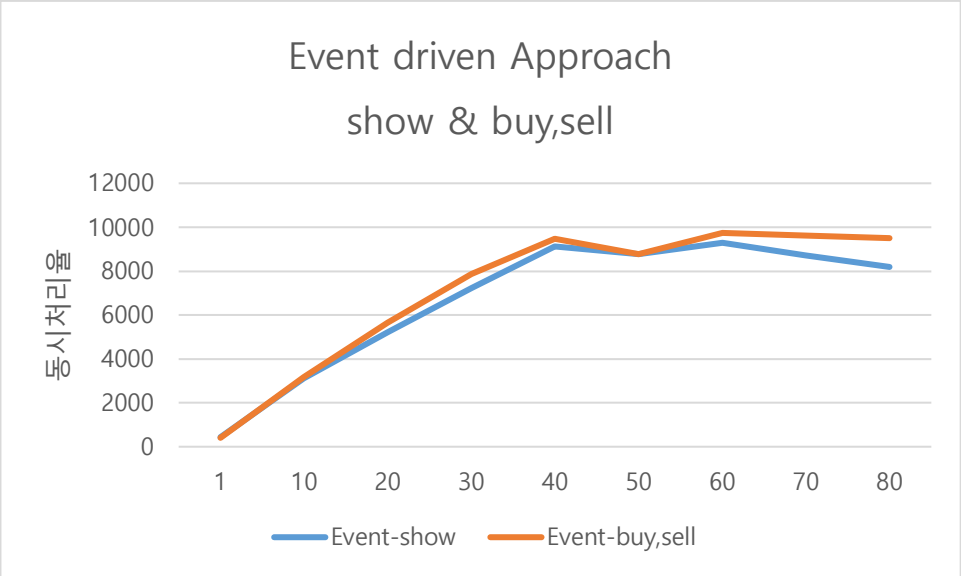

Figure 10 Event(show & buy, sell)

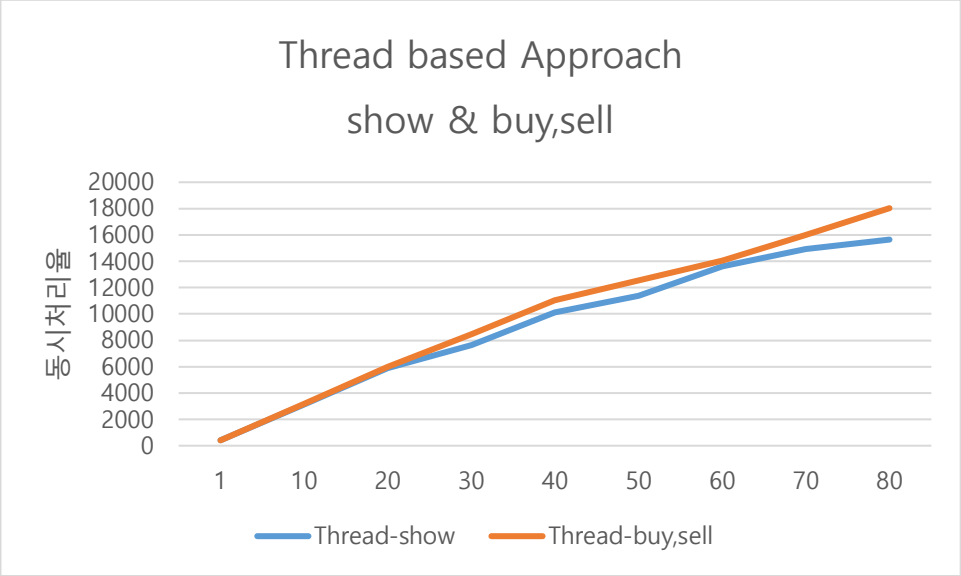

Figure 11 Thread(show & buy,sell)

먼저 1)번은 Event driven 방식과 Thread based 방식의 동시처리율 차이를 확인하였다. Show, buy, sell 모든 명령어가 입력되고, client 수를 1부터 80까지 증가시키며 동시처리율을 비교해 보았다. 그 결과 table1과 figure 3에서 볼 수 있듯이 client수가 1일 때는 Event 방식이 좀 더 동시처리율이 높지만, client 수가 증가함에 따라 이 상황은 역전되어 thread 방식이 동시처리율이 더 높은 걸 확인할 수 있었다. B의 task3에서 예상하였듯이 Event 방식은 overhead가 적어 client가 적을 때는 thread 방식보다 우세한 경향을 보이지만, client 수가 증가함에 따라 여

러 thread에서 동시에 처리하는 thread 방식이 더 우세하다는 것을 알 수 있었다. 또한 event 방식의 결과 그래프들을 살펴보면 client 수가 증가함에 따라 동시처리율이 증가하다가 일정 수준이 넘어서면 거의 일정해지는 것을 확인할 수 있다. 이는 단일 프로세스를 가진 event 기반 서버에서는 단위시간 당 처리할 수 있는 요청의 수에 한계가 있다는 것을 알 수 있다. Thread 기반 서버에서는 100개의 thread를 설정하였으므로 동시처리율이 1차함수 형태로 계속 증가할 수 밖에 없다.

2),3),4)번을 확인하면 show만 실행했을 때와 buy와 sell만 실행했을 때의 차이를 확인할 수 있다. Figure 10과 11을 살펴보면 두 서버 모두 Show의 명령이 동시처리율이 더 낮은 것을 알 수 있다. 이런 결과가 나온 이유는 B의 task3에서 예상하였듯이 buy와 sell 명령어는 해당 주식 id 하나에 접근하여 결과를 확인하고 보내는 반면에, show 명령어는 모든 노드를 확인해야 하므로 동시처리율이 더 느릴 수밖에 없다.

성능 평과 결과 B 개발 내용의 task3에서 예측했던 상황과 모두 맞아 떨어진다. 이를 통해 수업시간에 배운 이론을 실제 실험을 통하여 눈으로 확인할 수 있었고, 이론이 정확하다는 것을 증명할 수 있었다. 이번 과제에서 제시된 확장성과 워크로드에 따른 분석을 성공적으로 수행하였다. Client 개수 변화에 따른 동시처리율의 경향을 확인하였고, 요청 타입에 따른 동시처리율의 변화도 확인하였다.

성능 평가가 성공적으로 나오고, 서버의 실행과 client와의 연결, 요청, 응답 또한 잘 이루어졌다. 서버가 종료되었을 때 stock.txt파일의 업데이트 또한 잘 이루어졌다. 이 모든 것을 통해 이번 프로젝트에서 작성된 Event driven Approach 서버와 Thread based Approach 서버 모두 잘 작성되었음을 알 수 있다.