Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 정서영

학번 : 20211589

1. 개발 목표

이 프로젝트의 목표는 동시 접속 및 서비스를 지원하는 주식 서버를 구현하는 것이다. 이를 위해 Event-driven Approach와 Thread-based Approach의 두 가지 방식으로 주식 서버를 구현하고, 성능을 평가하고자 한다.

이 프로젝트에서 클라이언트는 서버에 show, buy, sell, exit의 4개의 명령어를 요청할 수 있다. 이 4개의 명령어에 대한 구체적인 내용은 다음과 같다.

(1) show : 현재 주식의 상태를 보여준다.

(2) buy [주식 ID] [개수] : [주식 ID]에 해당하는 주식을 [개수]만큼 산다.

(3) sell [주식 ID] [개수] : [주식 ID]에 해당하는 주식을 [개수]만큼 판다.

(4) exit : 서버와의 연결을 끊는다.

이 프로젝트는 주식 서버를 실행하고, 주식 관리 테이블에서 메모리를 받아오고, 클라이언트의 요청을 처리한 후, 주식 관리 테이블이 디스크에 저장되도록 구현한다.

이를 위해 주식들을 stock.txt 파일에 테이블 형태로 관리하며, 테이블에서 각 행은 각 주식의 ID, 잔여 주식, 주식 단가로 이루어져 있다. 주식 단가는 변동이 없으며, 클라이언트가 잔여 주식보다 많은 주식을 요구하면 잔여 주식이 부족하다는 메시지만 출력하도록 설계한다.

또한 효율적인 데이터 관리를 위해 주식 종목(item)을 노드로 가지는 Binary Tree를 사용한다. 이를 통해 Readers-Writers Problem solution을 고려한 노드 단위의 관리(fine-grained locking)를 수행한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   주식 서버를 실행하면 먼저 stock.txt 파일에서 메모리를 받아온다. 구조체 pool과 Select() 함수를 활용하여 Event를 기반으로 여러 클라이언트의 동시

명령을 수행할 수 있다. show, buy, sell, exit의 각 명령에 따라 적절히 작동하고 그 결과가 클라이언트 측에 보내진다. 서버가 ctrl+c로 종료되면 stock.txt 파일에 변경사항이 저장된다.

2. Task 2: Thread-based Approach

   주식 서버를 실행하면 먼저 stock.txt 파일에서 메모리를 받아온다. sbuf와 pthread를 활용하여 Thread를 기반으로 여러 클라이언트의 동시 명령을 수행할 수 있다. show, buy, sell, exit의 각 명령에 따라 적절히 작동하고 그 결과가 클라이언트 측에 보내진다. semaphore를 활용하여 공유 리소스에 대한 액세스를 조정할 수 있다. 서버가 ctrl+c로 종료되면 stock.txt 파일에 변경사항이 저장된다.

3. Task 3: Performance Evaluation

   Task1, Task2에서 구현한 각 주식 서버에 대한 elapse time을 측정하여 성능을 평가한다. 이를 통해 클라이언트의 개수 변화에 따른 동시 처리율 변화, 클라이언트의 요청 타입에 따른 동시 처리율 변화 등을 분석할 수 있다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  I/O Multiplexing은 단일 프로세스에서 여러 클라이언트의 요청을 처리하기 위해 사용하는데, select() 함수를 통해 구현할 수 있다. select() 함수는 여러 file descriptor의 집합을 모니터링하고 그 중 I/O 이벤트가 발생한 file descriptor들을 식별하는 역할을 수행한다.

  I/O Multiplexing을 구현하는 데 사용되는 매크로가 있는데, 먼저 FD_SET은 file descriptor을 file descriptor 집합에 추가하는 역할을 한다. FD_ZERO는 file descriptor 집합을 초기화하는 역할을 한다. FD_CLR은 file descriptor을 file descriptor 집합에서 제거하는 역할을 한다. FD_ISSET은 file descriptor 집합에서 특정 file descriptor의 I/O 이벤트 여부를 확인하는 역할을 한다.

Task1에서 각각의 클라이언트는 fd를 trigger하고, 서버는 기본적으로 연결된 클라이언트의 connfd를 한 배열에 저장한다. 그리고 Select() 함수를 이용하여 file descriptor에 pending inputs이 있는지 알아낸다. 이를 통해 listenfd에 input이 있는 경우 Accept() 함수를 통해 새로운 connfd를 배열에 추가하고, 배열에 있는 모든 connfd에 들어온 input을 처리한다.

✓ epoll과의 차이점 서술

먼저 select의 경우 관리할 수 있는 file descriptor의 수에 제한이 있는 반면, epoll의 경우 관리할 수 있는 file descriptor의 수에 제한이 없다. 또한 select의 경우 file descriptor 집합을 모니터링하기 위해 매번 집합 전체를 전달하는 반면, epoll의 경우 file descriptor 집합을 커널 내에 저장해서 이벤트가 발생한 file descriptor만 체크하기 때문에 효율적인 이벤트 관리가 가능하다.

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

✓ Master Thread의 Connection 관리

처음에 NTHREADS만큼 Pthread를 만들도록 하고, 이후에는 연결 요청을 Accept하고 connfd를 버퍼에 넣어주는 역할을 수행하도록 구성한다. Master Thread가 Producer 역할을 하고 Worker Thread가 Consumer 역할을 하는 형태이다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

worker threads는 suspend 상태로 있다가 connfd가 버퍼에 들어오면 그 중 하나가 깨어나서 해당 클라이언트의 명령을 처리한다. 명령을 처리하기 위해 호출되는 thread 함수에서 thread가 끝나면 커넬에 의해 reaping될 수 있도록 detach 모드로 실행한다. 이때 buy와 sell 명령어의 경우 writer이고 show 명령어의 경우 reader로 P와 V를 이용해 상호 배제적으로 각 item에 접근하도록 한다.

- **Task3 (Performance Evaluation)**

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

동시 처리율(시간당 클라이언트 처리 요청 개수)을 통해 주식 서버의 성능을 평가하고자 했고, 이를 위해 gettimeofday 함수를 사용하여 시간을 측정하였다. multiclient.c 코드 첫 부분에서 start에 시작 시간을 기록하고, 끝 부

분에서 end에 끝난 시간을 기록했고, 이를 활용해서 경과 시간을 얻고 printf를 통해 콘솔창에 출력하도록 했다. 클라이언트 개수와 각 클라이언트가 보낸 명령의 개수로 클라이언트의 처리 요청 개수를 구하고, 앞서 알아낸 시간으로 이를 나누어서 동시 처리율을 구했다. 클라이언트의 개수를 달리 하거나, multiclient.c에서 명령어를 선별적으로 보내는 수정 작업을 하면서 Task3을 수행했다.

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트의 개수가 많아질수록 Task2가 Task1보다 더 높은 동시처리율을 보일 것으로 예상된다. Task2 방식이 Task1보다 멀티코어의 장점을 살릴 수 있기 때문이다. 또한 Task2에서 buy, sell, show를 섞어서 요청하는 경우에 가장 낮은 동시처리율을 보일 것으로 예상된다. 그 이유는 Writer 명령어와 Reader 명령어 간의 동기화 작업으로 인한 수행 시간이 더 발생하기 때문이다.

C. 개발 방법

- **Task1 부분**

[stockserver.c]

기본 구조는 강의 자료 코드를 참고하여 다음 구조체와 함수를 구현했다.

- pool 구조체는 연결된 여러 클라이언트의 파일 디스크립터를 관리하는 데 쓰인다.

- void init_pool(int listenfd, pool *p); pool 구조체를 초기화하는 함수이다.

- void add_client(int connfd, pool *p); 새로 연결된 클라이언트를 pool에 추가하는 함수이다.

- void check_clients(pool *p); 준비된 클라이언트 이벤트를 확인하고 처리하는 함수이다. 이 함수에서 for문 내에서 RIO I/O를 활용해 명령어를 받고 command 함수를 호출해 각 명령어에 해당하는 내용을 수행할 수 있도록 구현했다.

- **Task2 부분**

[stockserver.c]

기본 구조는 강의 자료 코드를 참고하여 다음 구조체와 함수를 구현했다.

- sbuf_t 구조체는 세마포어를 이용해 스레드 간 동기화를 위해 사용하는 버퍼를 관리하는 데 쓰인다.

- void sbuf_init(sbuf_t *sp, int n); sbuf_t 구조체를 초기화하는 함수이다.

- void sbuf_deinit(sbuf_t *sp); sbuf_t 구조체를 해제하는 함수이다.

- void sbuf_insert(sbuf_t *sp, int item); 버퍼에 item을 삽입하는 함수이다. 세마포어를 사용해서 상호 배제적으로 버퍼에 접근할 수 있다.

- int sbuf_remove(sbuf_t *sp); 버퍼에서 item을 제거하는 함수이다. 세마포어를 사용해서 상호 배제적으로 버퍼에 접근할 수 있다.

- void *thread(void *vargp);

이 함수에서 detach 모드를 활용했고, 무한 while 루프 내에서 RIO I/O 를 활용해서 각 thread에 대한 명령어를 받았다. 이후 command 함수를 호출하여 각 명령어에 해당하는 내용을 수행할 수 있도록 구현했다.

- void show(int connfd, node* cur);

- void command(int connfd, char* buf);

Task2에서는 특히 Readers-Writers Problem solution을 고려한 노드 단위의 관리(fine-grained locking)를 수행하기 위해서, 먼저 mutex를 활용해 상호 배제적으로 readcnt를 증가시키거나 감소시키는 작업을 수행하도록 구현했다. readcnt는 item이 접근되어 읽힌 횟수를 나타내는 정수를 의미한다. 그리고 show 함수의 경우 buy나 sell 명령어를 요청하는 다른 thread가 접근할 수 없도록 write 기능을 막은 후 한 item을 읽고 write 기능을 다시 푸는 형태로 구현했다. command 함수에서도 buy나 sell 명령어를 실행할 때 다른 thread가 접근할 수 없도록 구현했다.

- **Task1, Task2 공통 부분**

[stockserver.c]

- void load(FILE* fp);

서버를 시작할 때 stock.txt 파일에서 주식 정보를 받아 오기 위해 만든 함수이다. 파일을 읽어와서 트리에 그 내용을 저장한다.

- void save(FILE* fp, struct node* cur);

서버를 종료할 때 stock.txt 파일에 주식 정보를 저장하기 위해 만든 함수이다. 트리 내용을 파일에 저장한다.

- void sigint_handler(int sig);

서버가 ctrl+c로 종료되었을 경우 실행되는 시그널 핸들러 함수이다. main 함수에서 Signal(SIGINT, sigint_handler) 부분을 추가하여 SIGINT에 대한 시그널 핸들러를 등록했다. stock.txt 파일에 업데이트된 내용을 저장하고, 버퍼나 트리에 쓰인 메모리를 해제하는 역할을 수행한다.

- char sendBuf[MAXLINE];

서버에서 클라이언트로 보내는 버퍼를 저장하는 전역 변수이다. command 함수에서 각 명령어 수행에 대한 결과를 strcpy로 sendBuf에 저장해두고 Rio_writen을 통해 클라이언트로 보내도록 구현했다.

- void show(int connfd, node* cur);

show 명령어를 받은 경우 실행하는 함수로 command 함수에서 처음 호출된다. 주식 종목을 저장해둔 트리를 순회하는 방식으로 현재 주식의 상태를 sendBuf에 strcat으로 저장해 두도록 구현했다. command 함수 끝 부분에서 한 번에 Rio_writen으로 show 명령어의 결과가 클라이언트에 보내지게 된다.

- void command(int connfd, char* buf);

클라이언트로부터 받은 명령어를 처리하기 위해 만든 함수이다. connfd와 명령어를 저장한 buf를 받아와서 show, buy, sell에 대한 각 명령을 수행하도록 구현했다.

[bst.h / bst.c]

주식 종목(item)을 노드로 가지는 Binary Tree에 대해 이진 검색 트리를 사용하기 위해 bst.h와 bst.c 파일을 추가하고 다음과 같은 구조체와 함수를 구현했다.

- item 구조체는 주식 종목을 나타낸다. 먼저 각 주식의 ID, 잔여 주식, 주식 단가를 나타내는 int형 변수 ID, left_stock, price가 있다. readcnt는 item이 접근되어 읽힌 횟수를 나타내는 정수이고, mutex는 item에 접근할 때 상호 배제를 위해 사용하는 세마포어이고, write는 item에 수정이 있을 때 상호 배제를 위해 만든 세마포어이다.

- node 구조체는 BST의 노드를 나타내며, item 구조체와 현재 노드의 왼쪽 자식 노드를 가리키는 포인터, 그리고 오른쪽 자식 노드를 가리키는 포인터를 가지고 있다.

- node* insertNode(node* root, item item);

  BST에 새로운 노드를 삽입하기 위해 만든 함수이다.

- node* searchNode(node* root, int ID);

  해당 ID의 주식 종목을 BST에서 검색하기 위해 만든 함수이다.

- void freeNode(node* root);

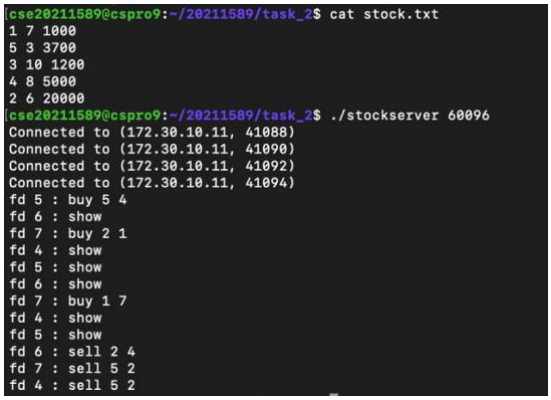
  BST의 모든 노드를 해제하기 위해 만든 함수로, 서버가 종료될 때 호출되도록 구현했다.

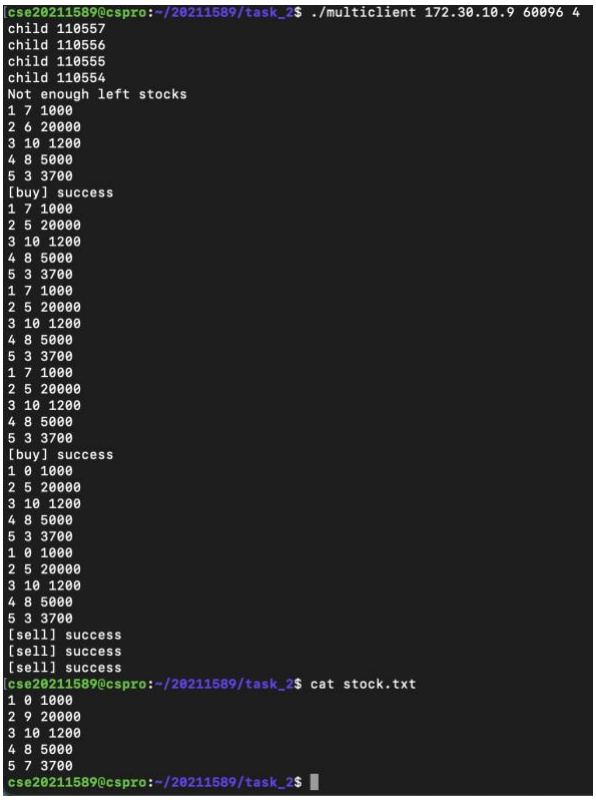
- 이후 [stockserver.c] 에서 node* root를 선언해서 주식 종목을 노드로 가지는 트리를 만들어 사용했다.

3. **구현 결과**

Event-driven Approach와 Thread-based Approach의 두 가지 방식으로 동시 주식 서버를 구현했고, 성능을 평가할 수 있다. 주식 서버를 실행하면 주식 관리 테이블에서 메모리를 받아오고, 클라이언트의 요청을 처리한다. 서버가 ctrl+c로 종료되면 주식 관리 테이블이 디스크에 저장된다. 또한 Thread-based Approach 주식 서버에서 semaphore를 활용하여 공유 리소스에 대한 액세스를 조정할 수 있게 되어 노드 단위의 관리가 가능하다.

다음은 Task2에 관한 실행 예시로, Task1도 실행 예시는 다음과 같은 형태로 나타난다. ORDER_PER_CLIENT는 3개, STOCK_NUM은 5개일 때의 예시이다.





4. 성능 평가 결과 (Task 3)

gettimeofday 함수를 사용하여 시간을 측정하였다. 시작 시간과 끝 시간은 multicline.c 코드의 첫 부분과 마지막 부분에서 측정했고, 다음과 같은 방식으로 구한 시간을 활용해서 분석하였다. multicline.c 코드에서 usleep(1000000) 부분은 주석 처리 후 실험하였다.

```c
struct timeval start;    /* starting time */
struct timeval end;    /* ending time */
unsigned long e_usec;    /* elapsed microseconds */

gettimeofday(&start, 0);    /* mark the start time */
```

```c
gettimeofday(&end, 0);        /* mark the end time */
e_usec = ((end.tv_sec * 1000000) + end.tv_usec) - ((start.tv_sec * 1000000) + start.tv_usec);
printf("elapsed time: %lu microseconds\n", e_usec);
```

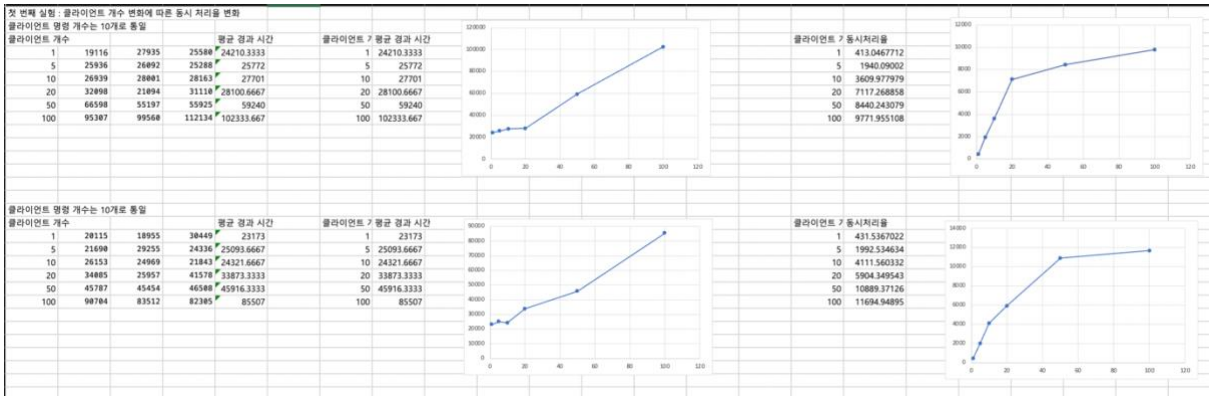
[성능 평가 1] 클라이언트의 개수 변화에 따른 동시 처리율 변화

먼저 Task1, Task2 방법에 대해 각각 클라이언트의 개수 변화에 따른 동시 처리율 변화를 알아보고 확장성을 분석해보고자 했다. 각 클라이언트가 보내는 명령의 개수는 10개로 설정한 후, 클라이언트 개수를 점차 늘려가며 경과 시간을 측정하였다. 각 클라이언트 개수마다 총 3번씩 시행하고 평균 경과 시간을 계산하여 실험의 정확도를 올리고자 했다. 다음은 실험할 때의 출력 결과 중 하나를 캡처한 것이다.

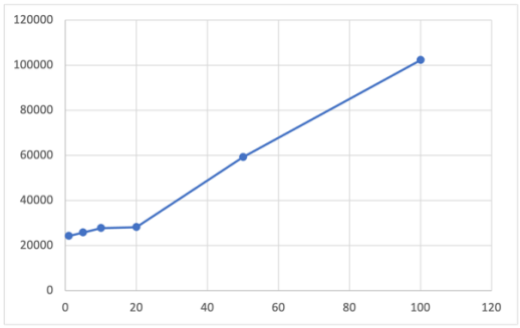
```
cse20211589@cspro9:~/20211589/task_1$ ./stockserver 60096
Connected to (172.30.10.11, 43718)
fd 4 : show
fd 4 : show
fd 4 : buy 4 1
fd 4 : buy 2 2
fd 4 : show
fd 4 : show
fd 4 : sell 5 7
fd 4 : show
fd 4 : buy 3 1
fd 4 : sell 2 6
```

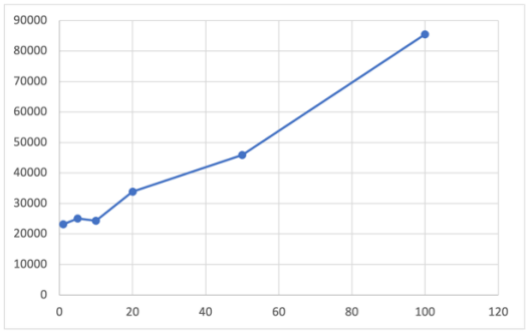
```
cse20211589@cspro:~/20211589/task_1$ ./multiclient 172.30.10.9 60096 1
child 40749
1 31 1000
2 3 20000
3 13 1200
4 16 5000
5 39 3700
1 31 1000
2 3 20000
3 13 1200
4 16 5000
5 39 3700
[buy] success
[buy] success
1 31 1000
2 1 20000
3 13 1200
4 15 5000
5 39 3700
1 31 1000
2 1 20000
3 13 1200
4 15 5000
5 39 3700
[sell] success
1 31 1000
2 1 20000
3 13 1200
4 15 5000
5 46 3700
[buy] success
[sell] success
elapsed time: 19116 microseconds
cse20211589@cspro:~/20211589/task_1$
```

다음은 실험 내용을 정리했던 엑셀 파일을 일부 캡처한 것이다.

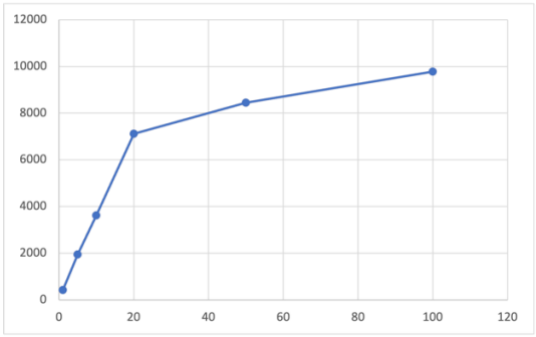


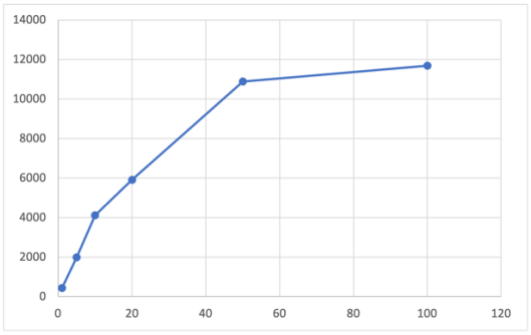
우선 각 Task 결과별로 살펴보면 다음과 같다. 왼쪽이 Task1 그래프, 오른쪽이 Task2 그래프이다.





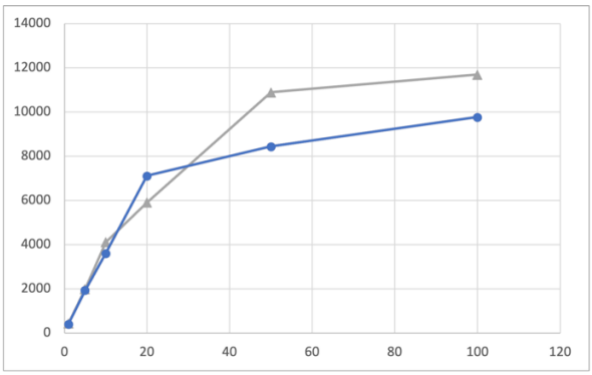
위의 그래프에서 가로축은 클라이언트의 개수, 세로축은 평균 경과 시간을 나타낸다. 시간의 단위는 microsecond이다.





위의 그래프에서 가로축은 클라이언트의 개수, 세로축은 동시처리율을 나타낸다. 동시 처리율은 시간당 클라이언트 처리 요청 개수를 의미하고, 처리한 요청 개수를 경과 시간으로 나누어서 구했다. 동시처리율 계산 시 10^6을 곱해 시간의 단위를 second로 변환해서 계산했다.

다음 그래프에서 가로축은 클라이언트의 개수, 세로축은 동시처리율을 나타낸다. Task1과 Task2를 함께 비교하기 위해 만든 그래프로, 파란색이 Task1, 회색이 Task2를 나타낸다.



그래서 그래프를 살펴본 결과, 클라이언트의 개수가 1개에서 20개로 늘어날 때는 Task1과 Task2 방식이 비슷한 동시처리율을 보이다가 클라이언트의 개수가 더 늘어나게 되면 Task2가 더 높은 동시처리율을 보이는 것을 알 수 있었다. 또한 Task1과 Task2 모두 클라이언트의 개수가 늘어날수록 동시처리율이 높아지는 것을 알 수 있었다. 그리고 이때 클라이언트의 개수에 비례해서 높아지지 않고, 클라이언트 개수가 증가함에 따라 높아지다가 점점 완만해지는 형태로 나타남을 볼 수 있었다.

클라이언트의 개수가 50개 이상일 때는 Task2 방식이 훨씬 높은 동시처리율을 보이는데, 그 이유는 Task1 방식이 멀티코어를 활용하지 못하기 때문으로 추측해볼 수 있었다. 클라이언트의 개수가 20개일 때를 살펴보면 Task1 방식이 Task2 방식보다 높게 나온 것을 볼 수 있는데, 이는 Task1 방식이 Task2 방식과 달리 스레드 제어 오버헤드나 동기화 오버헤드가 없기 때문에 그런 것으로 이해해볼 수 있었다.

[성능 평가 2] 클라이언트의 요청 타입에 따른 동시 처리율 변화

다음으로 Task1, Task2 방법에 대해 각각 클라이언트의 요청 타입에 따른 동시 처리율 변화를 알아보고 워크로드에 따른 분석을 해보고자 했다. 각 클라이언트가 보내는 명령의 개수는 총 10개, 클라이언트의 수는 10개로 설정한 후, 클라이언트가 보내는 요청 타입을 변경해가면서 경과 시간을 측정하였다. 클라이언트가 보내는 요청 타입은 총 3가지 경우로 나누어 생각했는데, 각각 모든 클라이언트가 show만 요청하는 경우, 모든 클라이언트가 buy 또는 sell을 요청하는 경우, 모든 클라이언트가 buy, sell, show를 섞어서 요청하는 경우에 해당한다. 각 경우마다 총 5번씩 시행하고 평균 경과 시간을 계산하여 실험의 정확도를 올리고자 했다.

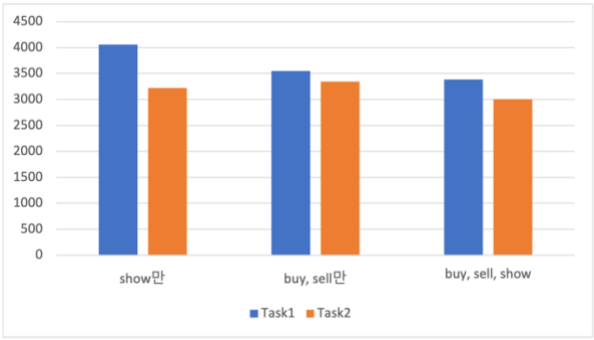
다음은 실험 내용을 정리했던 엑셀 파일을 일부 캡처한 것이다.

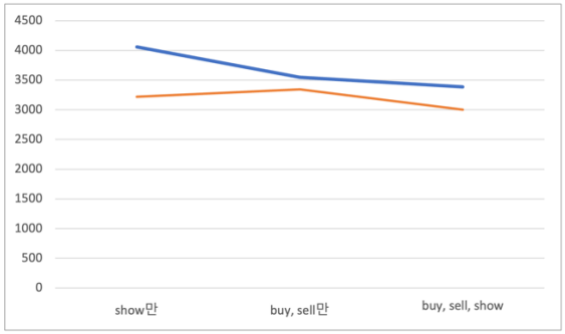
수는 10개										
		show만			buy, sell만			buy, sell, show		
		Task1	Task2		Task1	Task2		Task1	Task2	
길 때		32862	26983		25514	37412		25742	39476	
		25813	41529		29563	33234		35721	29304	
		22142	26083		29731	26576		30072	33562	
		23753	25537		25495	29279		22376	30137	
		18728	35217		30740	23117		33822	34067	
	평균	24659.6	31069.8	평균	28208.6	29923.6	평균	29546.6	33309.2	
	동시처리율	4055.215819	3218.559502	동시처리율	3545.018186	3341.843896	동시처리율	3384.484171	3002.173574	

우선 각 Task 결과별로 살펴보면 다음과 같다. 왼쪽이 Task1 그래프, 오른쪽이 Task2 그래프이다.

위의 그래프에서 가로축은 클라이언트의 개수, 세로축은 동시처리율을 나타낸다. Task1의 경우 모든 클라이언트가 show만 요청하는 경우 가장 높은 동시처리율을 보임을 알 수 있었고, Task2의 경우 모든 클라이언트가 buy 또는 sell을 요청하는 경우 가장 높은 동시처리율을 보임을 알 수 있었다. 그리고 Task1과 Task2 모두 buy, sell, show를 섞어서 요청하는 경우 가장 낮은 동시처리율을 보였다. Task1과 달리 Task2에선 show가 더 낮은 동시처리율을 보인 이유에 대해서는 동기화 작업을 추측해볼 수 있었다. Task2에서 show 명령어를 실행할 때 노드 단위마다 세마포어를 관리하기 때문에 이로 인한 P, V 작업이 수행 시간을 느리게 만들었을 것으로 생각된다.

다음 그래프들도 마찬가지로 가로축은 클라이언트의 개수, 세로축은 동시처리율을 나타낸다. Task1과 Task2를 함께 비교하기 위해 만든 그래프로, 파란색이 Task1, 주황색이 Task2를 나타낸다. 다음 두 그래프는 같은 데이터인데, 그래프 모양만 다르게 한 것이다.





위 그래프를 살펴보면, 먼저 클라이언트의 수를 10개로 설정한 후 실험한 내용이기 때문에 성능 평가 1에서 보인 것처럼 Task1과 Task2의 동시처리율이 큰 차이를 보이지 않고 있다. 성능 평가 1의 내용을 따르면 클라이언트의 수를 50개 이상으로 설정하여 실험하였다면 Task2의 동시처리율이 더 높았을 것으로 예상된다.

추가적으로 buy, sell과 show가 섞여 있는 정도에 따라 결과값이 차이나는 것을 고려하기 위해, buy, sell, show를 섞어서 요청하는 경우를 실험할 때 소스코드를 조금 수정하여 서버를 종료할 때 다음과 같이 명령어 개수를 buy, sell이 나타난 개수와 show가 나타난 개수를 알 수 있도록 구성했다.

```
^Cbuy_sell_cnt: 65
show_cnt: 35
```

다음은 이렇게 실험한 내용에 대한 결과를 정리한 표이다.

buy, sell, show						
Task1				Task2		
경과 시간	buy, sell 개수	show 개수		경과 시간	buy, sell 개수	show 개수
25742	67	33		39476	65	35
35721	75	25		29304	62	38
30072	63	37		33562	73	27
22376	64	36		30137	69	31
33822	60	40		34067	63	37

Task2의 경우, show 개수가 많을수록 더 낮은 동시처리율, 즉 경과 시간이 더 클 것으로 예상하고 실험을 진행하였는데 위의 결과를 살펴보면 그렇지 않은 경우가 있었다. 이를 통해 buy, sell이 나타난 개수와 show가 나타난 개수에 따라 유의미한 경과 시간의 차이를 찾아볼 수는 없었고, 명령어에서 같은 주식 종목이 reader 명령어와 writer 명령어에서 번갈아 가면서 여러 번 나타날 경우 경과 시간이 더 크게 나타날 수 있을 것 같다.