Multicore Programming Project 2

담당 교수 : 박성용

이름 : 남기찬

학번 : 20211180

1. 개발 목표

이번 concurrent stock server 프로젝트에서는 여러 주식 client들이 주식 서버에 접속하고, 각 client가 명령어를 요청하면 server는 이를 수행하는 것을 구현한다. 먼저 'stock.txt'에 있는 주식 ID, 잔여 주식 개수, 단가를 효율적으로 데이터를 관리할 수 있는 이진 트리에 넣는다. 다음, server는 Accept() 함수를 통해서 여러 client들의 서버 연결 요청을 수락하고, 이진 트리에 저장된 데이터에 대해 각 client들이 요청하는 'show', 'buy', 'sell', 'exit' 이 4개의 명령어를 수행한다. 그리고 수행한 결과를 multiclient 터미널에 출력하고, 마지막으로 이를 반영하여 'stock.txt'를 업데이트한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

clientfd 배열, descriptor set 등과 Select() 함수를 이용하여 다수의 client들이 서버에 접속해서 동시에 요청하는 여러 명령어를 수행하는 것을 구현했다. Client에서 'buy a b'를 입력하면 server는 ID가 a인 주식의 잔여 개수를 확인하고 만약 b보다 많다면, 기존 잔여 개수에서 b만큼 빼고 multiclient 터미널에 성공적으로 buy했다고 출력한다. 잔여 개수가 b보다 작다면, 구매 처리를 하지 않고 'Not enough left stock'을 출력한다. 다음, client에서 'sell a b'를 입력하면 client의 보유 주식 정보는 고려하지 않고 ID가 a인 주식의 잔여 개수를 b만큼 늘린다. 그리고 성공적으로 sell했다고 출력한다. Client에서 'show'를 입력하면 현재 트리의 각 노드에 저장되어있는 주식 별 ID, 잔여 주식 개수, 단가를 multiclient 터미널에 출력한다. 마지막으로 client가 'exit'을 입력하면 client의 서버 연결을 끊고 주식 장에서 퇴장한다. 그리고, 모든 client들과 서버 통신이 끝난 뒤 'stock.txt'를 다시 보면 명령어들이 반영되어 업데이트 되어있다.

2. Task 2: Thread-based Approach

Pool of worker threads를 먼저 만들고, 연결된 descriptor들이 공유하는 buffer에 connfd를 넣어서 thread에서 하나씩 꺼내서 쓰면 서버에 연결된 여러 client들의 동시 명령어 요청을 수행할 수 있다. 이렇게 구현했을 때, client가 'show', 'buy', 'sell', 'exit'의 명령어를 입력 시 수행되는 결과는 Task 1의 결과와 동일하다.

3. Task 3: Performance Evaluation

- Client process의 개수를 다르게 하며 Event-based Approach와 Thread-based Approach를 했을 때의 동시 처리율을 비교해보면, client가 적을 때는 거의 차이가 나지 않지만, client가 많아지면 Thread-based Approach의 실행 시간이 빨라져 동시 처리율이 더 높게 나온다.

- client가 요청하는 명령어의 타입에 따라서도 동시 처리율이 다르게 나온다. 기본적으로 client가 'buy', 'sell', 'show'를 모두 요청하는 경우보다는 주식 data를 읽기만 하는 'show' 명령어만 요청할 때 동시 처리율이 더 높다. 그리고 'buy', 'sell'만 요청하는 경우는 동시 처리율의 차이가 거의 없다.

- Thread-based Approach에서 사용하는 Worker Thread Pool의 thread 수를 증가시키면 하면 더욱 병렬적으로 요청을 처리해 실행 시간도 빠르게 나오고, 동시 처리율도 높다. 그러나 이 수를 너무 증가시키면 동시 처리율은 다시 감소하게 된다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

먼저, Open_listenfd()를 통해 생성된 listenfd 소켓이 descriptor set인 rd_set에 추가된다. 그리고 어떤 이벤트가 발생하면 Select() 함수는 이 이벤트를 감지하여 정보를 rdy_set에 저장하는데, 이 이벤트가 listenfd에 대한 I/O 이벤트라면 rdy_set에 listenfd가 있으므로, Accept()를 통해서 client의 연결을 수락하고 통신을 하기 위한 새로운 connfd를 생성한다. 이렇게 각 client마다 생성되는 connfd들을 clientfd 배열에 저장함으로써 여러 client들이 server와 통신할 준비를 한다. 다음, 이 clientfd 배열 중 어떤 connfd에서 I/O 이벤트가 발생했는지 확인하고, 그 해당 connfd에서 'I/O 이벤트를 발생한 client'와 통신한다. 초기화된 rio를 통해 입력된 데이터를 읽고, Command()를 통해 client 요청을 수행한 후, 알맞은 명령어 수행 결과를 multiclient 터미널에 출력한다. 이렇게 각 client마다 서로 다른 connfd를 통해 서버와 독립적으로 통신하면, 다수의 client들의 I/O 작업을 concurrent하게 수행할 수 있다.

✓ epoll과의 차이점 서술

Select() 함수는 반복문을 통해 FD_ISSET()으로 모든 file descriptor들을 한번씩 체크하면서 client의 I/O 이벤트를 확인한다. 그리고 Select() 함수를 호출할 때마다 운영체제 커널에게 관찰할 file descriptor들의 목록을 인자로 전달해야 한다. 그러나 Select()와 다르게 epoll() 함수는 file descriptor들을 직접 관리하지 않고, 이를 처음에 운영체제에게 알려주기만 하면 된다. epoll()은 초기화 단계에서 file descriptor들의 목록과 이벤트를 등록하고, 어떤 이벤트가 발생하면 그 이벤트만 처리한다. kernel에서 file descriptor들의 정보를 유지하고 있으므로 인자 전달이 필요 없고, epoll_wait() 함수를 통해 이벤트가 발생한 file descriptor를 반환하므로 file descriptor에 대한 반복문도 필요하지 않다. 따라서 epoll()은 select()보다 더 효율적으로 file descriptor들을 관리하고, 여러 client들의 요청에 따른 I/O multiplexing도 더 효율적으로 수행할 수 있다. 그러나 epoll()은 select()와 다르게 리눅스에서만 사용하는 I/O multiplexing 기법이므로 다른 운영체제로의 이식성은 떨어진다는 단점이 있다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 Open_listenfd()와 Accept() 함수를 이용해서 여러 client들의 서버 연결 요청을 수락하고, client의 명령어를 수행할 Worker Thread Pool을 생성한다. 다음, Accept()에서 return된 각 connfd(연결 file descriptor)를 초기화된 sharedbuf에 추가하여 Worker Thread Pool이 나중에 사용할 수 있도록 한다. 이러한 구조를 통해 Master Thread는 여러 client들의 서버 연결을 동시에 관리하고, 각각의 통신을 sharedbuf로 구분된 Worker Thread에서 처리하게 하여 병렬적으로 수행할 수 있다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Pthread_create()을 통해 만들어진 Worker Thread들은 독립적으로 peerthread() 함수를 실행한다. 이 함수에서는 thread가 종료되기를 기다릴 필요 없이 thread가 종료되면 자동으로 reaping이 되도록 설정한다. 다음, client들이 연결되어 생성된 connfd들이 저장된 sharedbuf로부터 한 connfd를 가져온다. 그리고 이 connfd를 통해 client의 명령어를 Command()를 통해 수행한다. Client의 명령어를 모두 수행했다면 서버와의 연결이 더 이상 필요하지 않으므로 Close() 함수로 연결을 종료한다. 이를 반복하면 각 Worker Thread는 자기에게 할당된 통신을 개별적으로 수행할 수 있다.

- **Task3 (Performance Evaluation)**

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

  동시 처리율은 시간당 동시에 처리하는 client 요청의 개수이다. 즉 (동시 처리율) = (client 개수) * (client당 request 수) / (총 시간)이다. 동시 처리율을 이렇게 정의하는 이유는, 시스템이 동시에 처리할 수 있는 작업의 수를 나타내는 지표로서 시스템의 성능과 부하를 평가하기 위해서이다. Server는 client들의 요청을 동시에 처리할 수 있는 능력이 있어야 하는데, 이 능력은 동시 처리율을 통해 평가된다. 따라서 동시 처리율은 server 환경에서 매우 중요한 지표이다. Client 개수, client당 request 수는 처음에 직접 설정하는 요인이므로, 동시 처리율을 구하기 위해서는 client가 요청한 명령어들을 처리하는데 걸리는 시간을 측정해야 한다. Client가 명령어를 요청하는 시점을 시작점으로 하고, 명령어 요청이 끝난 시점을 끝 지점으로 한다. 이때 data와 client당 request 수, 주식 매매 시 request당 최대 주식 항목 개수 등을 모두 동일한 상태로 놓고 측정한다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  1) 각 방법에 대한 client 개수 변화에 따른 동시 처리율

  Event-based Apporach는 Thread-based Approach와 다르게 control flow가 단일 thread 내에서 유지되기 때문에, thread 간의 동기화는 필요하지 않고 명령어 수행 시 공유되는 데이터의 동기화만 하면 된다. 따라서 일반적으로는 Event-based Apporach의 동시 처리율이 더 높을 것이다. 그러나 client의 수가 늘어나게 되면 각 client에 대해 별도의 thread를 할당하여 병렬적으로 요청을 처리하는 Thread-based Appoach가 더 효율적일 것이므로, Thread-based Apporach의 동시 처리율이 더 높을 것이다.

  2) Client 요청 타입에 따른 동시 처리율

  만약 client가 'show' 명령어만 요청한다면, 각 thread들은 data를 오로지 읽기만 하기 때문에 명령어들은 동시에 수행될 수 있다. 따라서 실행 시간이 빨라지고 동시 처리율은 증가할 것이다. 그리고 만약 client가 'buy'나 'sell' 명령어만 요청한다면, 언제나 data를 수정하기 때문에 명령어를 동시에 수행할 수 없고, 따라서 'show'를 같이 요청하는 기존의 경우보다는 실행 시간이 조금 더 오래 걸릴 것이다. 동시 처리율은 감소할 것이다.

  3) Worker Thread Pool 안의 thread 수 변화에 따른 동시 처리율

Worker Thread Pool의 thread 수를 늘리면 더 많은 client의 명령어들을 동시에 수행할 수 있으므로, 실행 시간은 더 빨라질 것이고 동시 처리율도 증가할 것이다.

C. 개발 방법

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

먼저 'stock.txt'에 있는 각 주식의 ID, 잔여 주식 개수, 단가를 담기 위해 data 구조체를 만들고, data L 포인터, data R포인터와 DataInsert() 함수를 이용하여 이진 트리를 만든다. 또 Select()를 쓰기 위해 필요한 rd_set이나 active한 연결 descriptor들을 저장하는 clientfd 배열, clientrio 배열 등을 포함하는 pool 구조체를 만들어 client 연결 file descriptor들의 정보를 관리한다. 이제 여러 client들이 연결을 요청하면 Open_listenfd()를 통해 받은 listenfd 소켓을 pool 안의 초기화된 rd_set에 넣는다. 다음, Select() 함수와 FD_ISSET() 조건문을 통해 rd_set에 있는 file descriptor들 중에서 I/O 이벤트가 발생한 것을 감지하고, Accept() 함수를 통해 새로운 연결 connfd를 생성한다. 이때, rdy_set을 새로 만들어 현재 rd_set 정보를 저장한다. 다음, 생성된 connfd는 pool의 rd_set과 clientfd 배열에 넣고, 해당 rio를 초기화하여 각 client와 통신할 준비를 한다. 본격적으로 명령어를 수행하는 check() 함수에서는 clientfd 배열에 저장된 active한 connfd가 기존의 rdy_set에 있는지 FD_ISSET()으로 체크하고, 있다면 이 connfd에서 Command()를 통해 client의 요청을 처리한다. 명령어를 수행할 때는 초기화된 rio 값과 Rio_readline()를 통해 명령어를 읽고, response를 전달하는 건 Rio_writen()을 통해서 multiclient 터미널에 출력한다. 이를 계속 반복하여 여러 client들의 요청에 따른 I/O Multiplexing을 수행한다. 그리고 client의 명령어가 끝나면, Close()를 통해 각 client의 연결을 끊고, rd_set의 connfd는 제거하고 해당 clientfd도 -1로 초기화한다. 모든 client의 연결이 끝나게 되면, show()와 Fputs() 함수를 통해 바뀐 주식 관련 data들을 'stock.txt'에 업데이트한다.

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

Task1에서처럼 먼저 'stock.txt'에 있는 data들을 이진 트리 안에 넣는다. 다음으로, Task2에서 중요한 sharedbuf를 사용하기 위해 초기화를 하는데, 이때 Calloc()을 이용해서 sbuf의 버퍼 부분을 0으로 채우고, front와 rear는 0으로 하고, 버퍼에 새로운 아이템을 추가하는 슬롯 개수를 제어하는 aslot 세마포어는 k로, 버퍼에서 아이템을 제거하는 aitem 세마포어는 0으로 설정한다. 그리고 반복문을 이

용해서 client들의 명령어를 수행할 Pool of Worker Thread를 생성한다. 다음 Open_listenfd()를 거치고 Accept() 함수를 이용해 각 client마다 새로운 connfd를 생성하고, 이 connfd를 sharedbufInsert() 함수를 사용해서 각각 초기화된 sbuf에 넣는다. 이때 P(&aslot)를 이용해서 aslot 세마포어는 감소시키고, V(&aitem)을 이용하여 aitem 세마포어를 증가시켜 다른 Worker Thread에게 이용 가능한 connfd가 있다는 신호를 보낸다. 그리고 버퍼의 끝부분에 connfd을 넣을 때는 P(&mutex), V(&mutex)로 버퍼를 lock하여 다른 thread들이 동시에 접근하지 못하도록 한다. Worker Thread들은 peerthread() 함수를 수행하게 되는데, 각 thread마다 Pthread_detach() 함수를 이용해서 thread가 종료되면 자동으로 reaping이 되도록 설정한다. 그리고 sharedbufRemove()를 이용해서 버퍼에 저장되어 있는 connfd를 return값으로 하나씩 가져오고, 이 connfd를 통해 Command() 함수를 실행하여 client의 명령어를 수행한다. 이때 P(&aitem)을 이용해서 aitem 세마포어는 감소시키고, V(&aslot)을 이용하여 aslot 세마포어를 증가시켜 이용 가능한 slot이 있음을 알린다. connfd를 버퍼의 front에서 가져올 때도 다른 thread들이 접근하지 못하도록 P(&mutex), V(&mutex)로 lock해준다. 명령어가 더 이상 입력되지 않으면 Close() 함수로 client의 서버 연결을 끊는다. 이를 반복하면 client의 명령어가 모두 수행되고, 이후에 서버 접속은 끊어지게 된다. 역시 마지막으로 모든 client 통신이 끝난 경우에는 바뀐 data들을 'stock.txt'에 업데이트한다.

- **Task3 (Performance Evaluation)**

  client들이 요청한 명령어들을 수행하는데 걸리는 시간을 측정하므로, 측정의 시작 시점은 multiclient.c의 맨 윗부분으로 하고, 끝 지점은 multiclient.c의 맨 아랫부분으로 한다. multiclient.c 코드에서 시작 지점에는 gettimeofday(&start, NULL)을 놓고, 끝 지점에는 gettimeofday(&end, NULL)을 놓아 client의 명령어를 수행하는 데 걸린 시간(ms)를 multiclient 터미널에 출력한다. 그리고 이 측정값과 client 개수, client당 request 수를 고려하여 동시 처리율을 구하여 특정 configuration별 변화를 살펴본다.

  1) 각 방법에 대한 client 개수 변화에 따른 동시 처리율

  기존의 multiclient.c는 MAX_CLIENT를 100으로 정의하고 있는데, 이를 4, 10, 50개로 바꿔가면서 실행 시간을 측정한다.

  2) Client 요청 타입에 따른 동시 처리율

  기존의 multiclient.c는 'buy'와 'sell', 'show'이 모두 포함되어 있다. client가

'show'만 요청하게 하려면, option을 random number % 1로 설정하여 option이 0만 나오도록 한다. 그리고 client가 'buy'와 'sell'만 요청하게 하려면, option을 random number % 2로 설정하고 각각을 option 0과 1 조건문에 넣어 buy와 sell 명령어만 통과하도록 한다. 이 3가지 경우의 실행 시간을 측정한다.

3) Worker Thread Pool 안의 thread 수 변화에 따른 동시 처리율

stockserver.c에서 처음 설정했던 Worker Thread Pool의 thread수는 16개이고, 이 thread 수를 4개, 8개, 50개, 1000개로 바꿔가면서 실행 시간을 측정한다.

3. 구현 결과

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

Client가 'buy'나 'sell' 명령어를 요청하면 해당 ID 주식의 잔여 개수를 더하거나 빼고, multiclient 터미널에 성공적으로 매매했다고 출력한다. 이때 만약 사려는 주식의 개수가 잔여 개수보다 많은 경우에는 명령어를 수행하지 않고 'Not enough left stock'을 출력한다. 그리고 client가 'show' 명령어를 요청하면 이진 트리에 저장되어 있는 모든 주식 관련 data들을 출력한다. Server의 터미널에는 어떤 client와 연결 되었는지 'Connected to (,)'가 나오고, 각 명령어들의 글자 수가 출력된다. 가끔씩 처음 3-4개의 client 명령어와 server의 "server received by n bytes" 순서가 조금 바뀌어서 나오기도 했는데, server에 다수의 client가 막 연결돼서 앞다투어 명령어를 요청할 때에도 세마포어 P(&mutex), V(&mutex)를 통해 lock해준다면 조금 느려지지만 해결될 것 같다.

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

./multiclient를 실행하면 Task1에서 구현한 결과와 동일하게 나온다.

- **Task3 (Performance Evaluation)**

1) 각 방법에 대한 client 개수 변화에 따른 동시 처리율

Client process가 4개, 10개인 경우에는 Event-based Approach와 Thread-Based Approach의 동시 처리율이 거의 동일했다. 그러나 Client process의 개수가 50개, 100개로 커지면서 Thread-Based Approach의 동시 처리

율이 Event-based Approach보다 더 커졌다.

2) Client 요청 타입에 따른 동시 처리율

기존의 client가 'buy', 'sell', 'show'를 모두 요청하는 경우보다 client가 'show'만 요청하는 경우에 실행 시간이 더 느리게 나왔고, 동시 처리율은 낮았다. 그리고 client가 'buy'와 'sell'만을 요청하는 경우는 기존과 별 차이가 나지 않았다.

3) Worker Thread Pool 안의 thread 수 변화에 따른 동시 처리율

Pool of Worker Thread안의 thread 수가 많아질수록 실행 시간이 빨라지고, 그에 따라 동시 처리율도 증가했다. 그러나, thread 수가 너무 많은 경우에는 다시 동시 처리율이 감소했다. 그리고, 너무 큰 수를 입력하면 'Pthread_create error: Resource temporarily unavailable'라는 에러가 떴다. 정밀하게 실험을 하면 동시 처리율이 가장 큰 최적의 Worker Thread Pool 크기를 찾을 수 있을 것이다.

4. 성능 평가 결과 (Task 3)

1) 각 방법에 대한 Client 개수 변화에 따른 동시 처리율 변화 분석

※ 동일한 stock.txt(10개 종목)과 client당 request 수(10), 주식 매매 시 request당 최대 주식 항목 개수(10개), client request 타입을 가지고 측정한다.

※ 정확한 측정을 위해 multiclient.c의 usleep()은 제외하고 실행한다.

A) Event-based Approach

Client 수		4	10
실행 시간	Trial 1	Elapsed time: 21227 microseconds	Elapsed time: 30902 microseconds
	Trial 2	Elapsed time: 20797 microseconds	Elapsed time: 26004 microseconds
	Trial 3	Elapsed time: 21970 microseconds	Elapsed time: 26542 microseconds
실행 시간 평균(s)		0.021331	0.027816
동시 처리율(s^{-1})		1875.21	3595.05

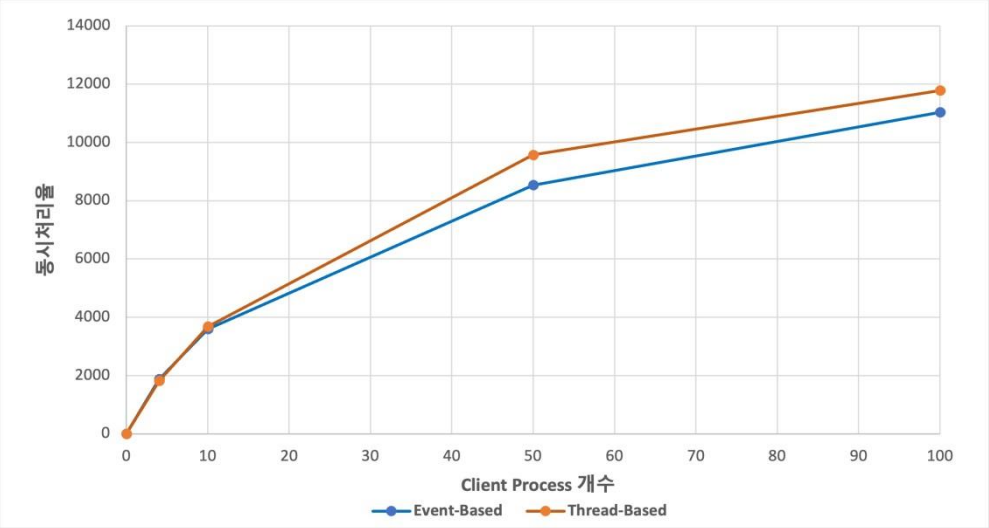
Client 수		50	100
실행	Trial 1	Elapsed time: 61916 microseconds	Elapsed time: 93939 microseconds

시간	Trial 2	Elapsed time: 55448 microseconds	Elapsed time: 89312 microseconds
	Trial 3	Elapsed time: 58357 microseconds	Elapsed time: 88698 microseconds
실행 시간 평균(s)		0.058574	0.09065
동시 처리율(s^{-1})		8536.21	11031.44

B) Thread-based Approach

Client 수		4	10
실행 시간	Trial 1	Elapsed time: 21315 microseconds	Elapsed time: 24962 microseconds
	Trial 2	Elapsed time: 21863 microseconds	Elapsed time: 31453 microseconds
	Trial 3	Elapsed time: 22757 microseconds	Elapsed time: 25083 microseconds
실행 시간 평균(s)		0.021978	0.027166
동시 처리율(s^{-1})		1820.01	3681.07

Client 수		50	100
실행 시간	Trial 1	Elapsed time: 52224 microseconds	Elapsed time: 86595 microseconds
	Trial 2	Elapsed time: 51896 microseconds	Elapsed time: 81995 microseconds
	Trial 3	Elapsed time: 52635 microseconds	Elapsed time: 84866 microseconds
실행 시간 평균(s)		0.052252	0.084485
동시 처리율(s^{-1})		9569.01	11836.42



그래프 1. Client Process 개수에 따른 동시처리율 변화

- 예상했던 것과 다르게, Client Process가 4개, 10개일 때는 Event-based Approach와 Thread-based Approach 간에 실행 시간과 동시 처리율이 별로

차이가 나지 않았다. 그러나 Client process 개수가 많아질수록 동시 처리율 차이는 뚜렷해졌다. Thread-based Approach에서의 실행 시간이 더 빨라져 Event-based Approach보다 더 높은 동시 처리율을 보였다. Thread-based Approach에서는 여러 client들이 동시에 요청하면, 각각의 요청을 별개의 thread에 할당해서 병렬로 처리할 수 있다. 이는 시스템 자원을 최대한 활용하여 전체 처리량을 늘릴 수 있고, 각 client의 상태도 thread에서 독립적으로 관리할 수 있다. 따라서 client process가 많은 경우에는 공유되는 자원들의 동기화 처리를 잘 한다면 Thread-based Approach의 성능이 더 효율적이다.

2) Client 요청 타입에 따른 동시 처리율 변화 분석

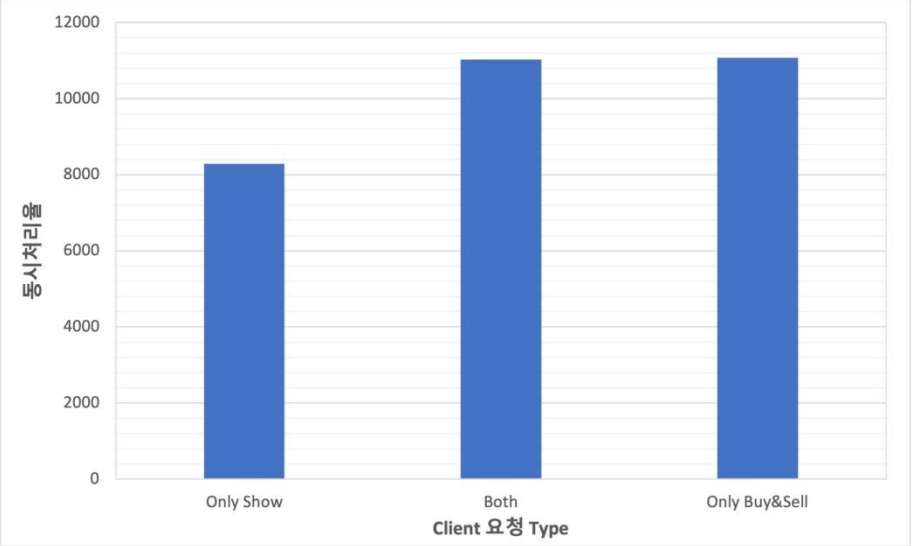
※ 동일한 stock.txt(10개 종목)과 client당 request 수(10), 주식 매매 시 request당 최대 주식 항목 개수(10개), client 개수(100개)를 가지고 측정한다.

※ 모두 Event-based approach로 측정한다.

※ 정확한 측정을 위해 multiclient.c의 usleep()은 제외하고 실행한다.

명령어 타입		Only show	Only buy, sell
실행 시간	Trial 1	Elapsed time: 108210 microseconds	Elapsed time: 92086 microseconds
	Trial 2	Elapsed time: 139292 microseconds	Elapsed time: 86960 microseconds
	Trial 3	Elapsed time: 114586 microseconds	Elapsed time: 91886 microseconds
실행 시간 평균(s)		0.120696	0.090311
동시 처리율(s^{-1})		8285.28	11072.85

명령어 타입		Both
실행 시간	Trial 1	Elapsed time: 93939 microseconds
	Trial 2	Elapsed time: 89312 microseconds
	Trial 3	Elapsed time: 88698 microseconds
실행 시간 평균(s)		0.09065
동시 처리율(s^{-1})		11031.44



그래프 2. Client 요청 Type별 동시처리율

- 예상했던 것과 다르게 client들이 'show'만 요청하는 경우에는 원래 'show'와 'buy', 'sell'이 같이 나오는 경우보다 실행시간이 더 느려 동시 처리율이 더 크게 나왔다. 반대로 client들이 'buy'와 'sell'만 요청하는 경우에는 원래와 비슷하거나 조금 더 빨라, 동시 처리율의 차이는 거의 없었다.
- 이 결과는 수업 시간에 배운 내용과 일치하지 않아 그 이유를 찾아보았고, 'show' 명령어는 이진 트리의 전체 node를 순회하여 data를 출력하지만, 'buy'와 'sell' 명령어는 이진 트리에서 특정 node를 search하여 그 node만 다루기 때문에 이렇게 나온 것이라고 추측했다. 그리고 이진 트리의 node가 더 많아진다면 node를 순회하고 search하는 데 걸리는 시간도 증가하므로 위 그래프 2의 동시 처리율 차이도 달라질 것이라고 생각한다.

3) Worker Thread Pool의 Worker Thread 수 변화에 따른 동시 처리율 변화 분석

※ 동일한 stock.txt(10개 종목)과 client당 request 수(10), 주식 매매 시 request당 최대 주식 항목 개수(10개), client 개수(100개)를 가지고 측정한다.

※ 모두 Thread-based approach로 측정한다.

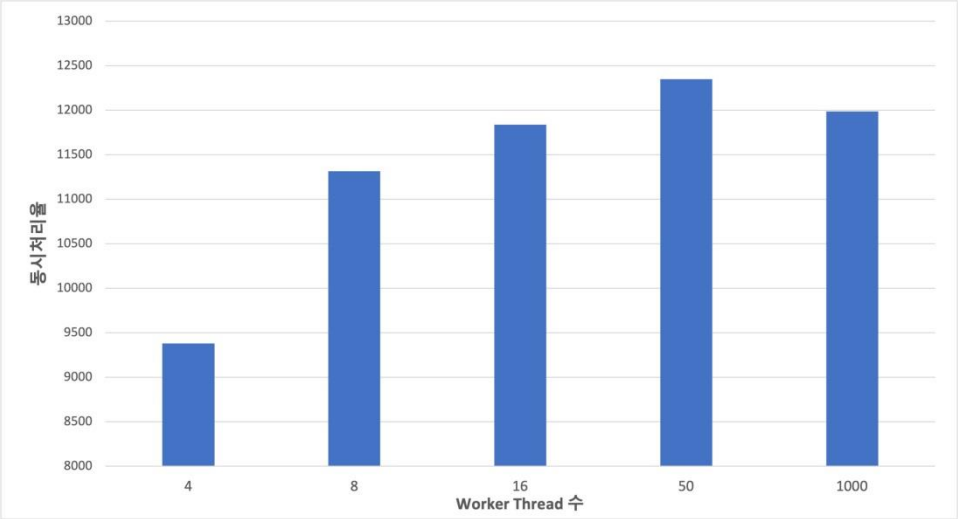
※ 정확한 측정을 위해 multiclient.c의 usleep()은 제외하고 실행한다.

Worker Thread 수		4	8
실행 시간	Trial 1	Elapsed time: 104823 microseconds	Elapsed time: 88641 microseconds
	Trial 2	Elapsed time: 104903 microseconds	Elapsed time: 90727 microseconds
	Trial 3	Elapsed time: 110118 microseconds	Elapsed time: 85767 microseconds
실행 시간 평균(s)		0.106615	0.088378

동시 처리율(s^{-1})	9379.54	11315.03

Worker Thread 수		16	50
실행 시간	Trial 1	Elapsed time: 86595 microseconds	Elapsed time: 80783 microseconds
	Trial 2	Elapsed time: 81995 microseconds	Elapsed time: 80571 microseconds
	Trial 3	Elapsed time: 84866 microseconds	Elapsed time: 81573 microseconds
실행 시간 평균(s)		0.084485	0.080976
동시 처리율(s^{-1})		11836.42	12349.34

Worker Thread 수		1000
실행 시간	Trial 1	Elapsed time: 81562 microseconds
	Trial 2	Elapsed time: 82666 microseconds
	Trial 3	Elapsed time: 86064 microseconds
실행 시간 평균(s)		0.083431
동시 처리율(s^{-1})		11985.95



그래프 3. Worker Thread 수 별 동시처리율(동시처리율 8000부터 그림)

- 예상한대로 Worker Thread Pool의 thread 수를 늘리면 실행 시간이 빨라져 동시처리율이 커졌다. 그러나, 예상과는 다르게 그래프 3을 보면 Worker thread를 50개에서 1000개로 늘렸을 때는 동시처리율이 감소했다. 이를 통해 Worker thread들이 너무 많아지면 thread 간의 경쟁으로 인해 overhead가 발생하고, 또 thread 간에 context switch가 더 많이 발생하여 실행 시간이 느려질 수 있다는 걸 알 수 있다. 따라서, 일반적으로 더 많은 thread들이 병렬로

client들의 요청을 처리할수록 실행 시간은 더 빨라지고 동시처리율이 증가하지만, Thread Pool에 thread가 너무 많은 경우에는 오히려 처리 속도가 느려지고 동시처리율은 감소한다.

- 그리고, 이 결론에 따라 Worker Thread Pool의 크기를 변경하면서 실험을 계속 진행하면 가장 효율적인 최적의 thread 수를 결정할 수 있을 것이다.