System Programming Project 5

담당 교수 : 김영재

이름 : 이예진

학번 : 20181668

1. 개발 목표

프로젝트 5에서는 여러 클라이언트들의 동시 접속 및 서비스를 위한 Concurrent stock server를 구축한다. 주식 서버는 주식 정보를 저장하고 있는 여러 클라이언트들과 소통한다. 주식 클라이언트는 서버에 주식 show, buy, sell, exit 요청을 할 수 있다. concurrent 프로그래밍을 통해 동시 주식 서버를 Event-driven Approach(using select)과 Thread-based Approach(using pthread)를 사용하여 설계하고 구현한다.

-buy : 주식을 사는 명령어이다. 잔여 주식이 부족한 경우 에러 메시지를 출력한다.

-sell : 주식을 파는 명령어이다.

-show : 현재 주식 테이블을 보여주는 명령어이다.

주식은 stock.txt 파일에 테이블 형태로 관리된다. 주식 단가는 변동이 없고 잔여 주식만 변동된다. 남은 주식보다 많은 주식을 요구하면 잔여 주식이 부족하다는 메시지만 출력하고 요청은 처리되지 않는다. 이진 트리 자료구조를 사용한다. Readers-writers problem solution을 고려한 노드 단위의 관리(fine-grained locking)가 필요하다. 프로그램이 종료되면 테이블의 업데이트된 내용이 stock.txt에 반영된다.

2. 개발 범위 및 내용

A. 개발 범위

1. select

각 클라이언트는 fd를 trigger하고 이 fd들을 monitoring하던 select에 의해 동작을 시작한다. 하나의 프로세스가 multiple 클라이언트를 handling하는 방식이기 때문에 서버 시작과 함께 파일의 내용을 메모리로 올리고 In-memory 프로세싱을 진행한다.

select 함수를 사용하여 concurrent event-driven 프로그램을 작성하는데 프로그램의 실행 과정을 설명하면 우선 active한 클라이언트의 집합은 pool에 저장되어 관리된다. init_pool 함수로 초기화하고 무한 루프를 사용하여 프로그램을 진행한다. 서버는 loop마다 select를 호출하여 입력 이벤트를 탐지한다. 새로운 연결 리퀘스트나 준비된 클라이언트에 대한 입력 이벤트가 있다. 서버는 연결을 열고 add_client를 호출하여 pool에 클라이언트를 추가한다. 다음으로 check_clients 함수로 주식 관련하여 명령어를 받고 해당하

는 결과를 보내주는 상호작용을 한다. 1에서 설명한 명령어를 모두 사용할 수 있고 여러 명의 클라이언트에게 서비스를 제공할 수 있다.

2. pthread

select와 마찬가지로 서버 시작과 함께 파일의 내용을 메모리로 올리고 In-memory 프로세싱을 진행한다.

pthread를 사용하여 concurrent server based on prethreading 프로그램을 작성한다. 프로그램 실행과정을 설명하면 우선 sbuf를 초기화하고 메인 쓰레드는 worker 쓰레드들을 생성한다. 다음으로 무한루프에 들어가서 연결 리퀘스트를 승인하고 sbuf에 연결된 디스크립터를 삽입한다. 각 worker 쓰레드는 버퍼에서 연결된 디스크립터를 제거하고 stock_func 함수로 주식 관련하여 명령어를 받고 해당하는 결과를 보내주는 상호작용을 한다. 1에서 설명한 명령어를 모두 사용할 수 있고 여러 명의 클라이언트에게 서비스를 제공할 수 있다.

B. 개발 내용

- **select**

✓ select 함수로 구현한 부분에 대해서 간략히 설명

select 함수를 사용하여 사용자 여러 명 접속을 받아서 그 사용자로부터 입력, 서버에서 사용자에게로 출력, 그리고 예외 처리를 해준다. 무한 루프 안에 select 함수가 pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL); 로 사용된다. select 함수의 반환값은 준비된 디스크립터의 개수이므로 pool.nready에 저장한다. pool.maxfd+1은 검사 대상이 되는 파일 디스크립터의 수이다. &pool.ready_set에는 수신된 데이터의 존재여부에 관심 있는 파일 디스크립터 정보가 등록되어 있다. 이 fd_set형 변수의 주소값을 select 함수에 전달한다. 나머지 인자는 NULL로 전달한다.(블로킹, 예외상황, 타임 아웃에 관련된 인자들이다.) select 함수의 호출 순서를 보면 우선 pool.ready_set에 pool.read_set으로 파일 디스크립터를 설정해준다. 이 변수와 검사 범위 관련 인자들을 select 함수에 전달하고 호출 결과를 확인한다. 다음으로 나오는 FD_ISSET는 &pool.ready_set으로 전달된 주소의 변수에 listenfd로 전달된 파일 디스크립터 정보가 있으면 양수를 반환한다. 양수인 경우에 Accept를 해주고 클라이언트 정보를 받아서 add_client를 호출해준다. 양수인 경우와 아닌 경우에 모두 check_clients를 호출한다. check_clients에서는 준비된 디스크립터들이 주식 관련 명령어를 주고 받는다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

stock.txt 파일을 fopen으로 열어서 만약 파일이 존재한다면 while 루프로 처리한다. 파일의 끝 EOF까지 while문을 반복한다. 파일에 있는 텍스트를 1줄씩 읽어서 id, left stock, price 순서대로 변수에 저장한다. 이 변수를 insert 함수로 넘겨줘서 이진 트리를 작성한다. insert 함수에서는 현재 이진 트리에서 노드가 삽입될 위치를 찾아서 malloc으로 공간을 할당하고 입력 받은 파라미터를 노드에 저장한다. 노드의 mutex와 w 변수에 대해 1로 초기화해준다.(pthread에서만 초기화한다. select에서는 mutex와 w가 사용되지 않기 때문이다.) 노드의 순서는 id에 따라 부모의 id보다 작으면 왼쪽, 크면 오른쪽에 삽입되도록 하였다. 이진 트리 생성이 끝나면 파일을 닫는다. 프로그램이 종료되면 업데이트된 이진 트리의 내용을 stock.txt에 업데이트 해준다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

sbuf_init으로 sbuf를 초기화하고 반복문으로 Pthread_create(&tid, NULL, thread, NULL)를 호출하여 worker thread를 생성한다. 그리고 무한 루프에 들어간다. 연결 요청이 오면 Accept로 승인하고 sbuf_insert(&sbuf, connfd)로 sbuf에 클라이언트 정보인 connfd를 삽입한다. 생성된 worker 쓰레드는 thread 함수에서 Pthread_detach로 쓰레드를 메인 쓰레드에서 분리시킨다. 다음으로 무한 루프로 들어가서 sbuf_remove(&sbuf)와 stock_func(connfd)를 호출하여 sbuf에서 클라이언트 연결을 빼와서 주식 관련 서비스를 제공한다. stock_func에서는 클라이언트의 요청에 맞게 알맞은 주식 관련 명령어를 주고받는다. worker 쓰레드의 개수는 NTHREADS 변수에 값을 임의로 할당하여 사용할 수 있고 SBUFSIZE도 변수에 임의로 값을 할당하여 사용할 수 있다. 새로운 클라이언트에 새로운 쓰레드를 생성하는 방법보다 오버헤드를 줄일 수 있다. 메인 쓰레드는 반복적으로 클라이언트로부터 연결 리퀘스트를 승인하고 버퍼에 있는 디스크립터에 연결한다. 각 worker 쓰레드는 반복적으로 버퍼로부터 반복적으로 디스크립터를 제거하고 새로운 디스크립터를 기다린다.

C. 개발 방법

- select

우선 item이라는 구조체에 id, left stock, price 기본 정보에 변수를 추가하였다. left, right는 이진 트리의 왼쪽, 오른쪽에 연결되어 있는 노드를 가리키는 포인터이다. item *root는 이진 트리의 루트 노드이다.

pool이라는 구조체는 연결된 디스크립터들의 pool을 나타내는 구조체이다. maxfd는 최대 디스크립터 수를 나타내고 read_set, ready_set은 active 디스크립터, 읽기위한 준비된 디스크립터의 부분집합을 나타낸다. nready는 준비된 디스크립터 수를 나타내고 maxi는 클라이언트 집합에 들어갈 인덱스이고 client_fd는 active 디스크립터 집합, client_rio는 active read 버퍼의 집합이다.

insert, fMin, freenode, print, preorderPrint, strsave, preorderStr, find_id는 이진 트리 관련 함수이고 코드에 설명을 적어두었다. init_pool, add_client, check_clients는 select 관련 함수이고 코드에 설명을 적어두었다.

sigint_handler는 서버가 종료될 때 업데이트된 이진 트리의 내용을 stock.txt에 업데이트해준다.

check_clients는 위에서 설명한 것과 같은 역할을 한다.

- pthread

item 구조체는 select의 코드에다가 변수를 더 추가하였다. readcnt 변수는 현재 critical section에 있는 reader의 숫자이다. mutex는 readcnt 공유 변수에 대한 접근을 보호해주고 w는 critical section에 있는 이진트리 대한 접근을 보호해주는 역할을 한다. 이진트리 관련 함수들도 위에 작성한 select의 코드에서와 같다.

sbuf_t라는 구조체는 n개의 아이템들을 동적으로 할당하는 정수 배열인 buf와 배열의 앞과 뒤를 가리키는 front, rear 변수를 가지고 있다. mutex는 버퍼에 접근하는 권한을 제공하고 slots와 items는 빈 슬롯과 가능한 아이템을 세는 세마포어이다.

sbuf 관련 함수는 sbuf_init으로 버퍼에 힙 메모리를 할당할 수 있고 sub_deinit으로 버퍼 저장공간을 비울 수 있다. sbuf_insert로 가능한 슬롯을 기다릴 수 있고 뮤텍스를 잠그거나 풀 수 있고, 아이템을 추가하거나 새로운 아이템의 가능성을 알린다. sbuf_remove는 insert와 대칭적이다.

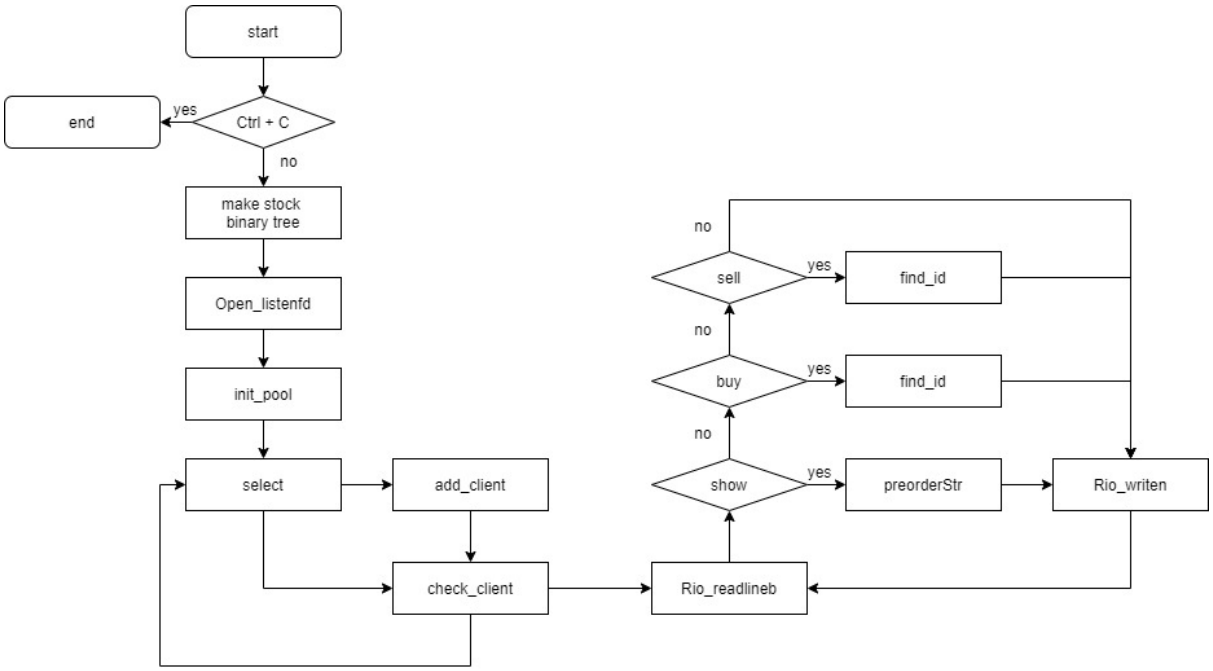
stock_func은 주식 관련 서비스를 제공하는 함수이다. Rio_readlineb로 입력을 받아서 show, buy, sell 등의 명령어를 처리한다.

sigint_handler는 서버가 종료될 때 업데이트된 이진 트리의 내용을 stock.txt에 업데이트해준다.

3. 구현 결과

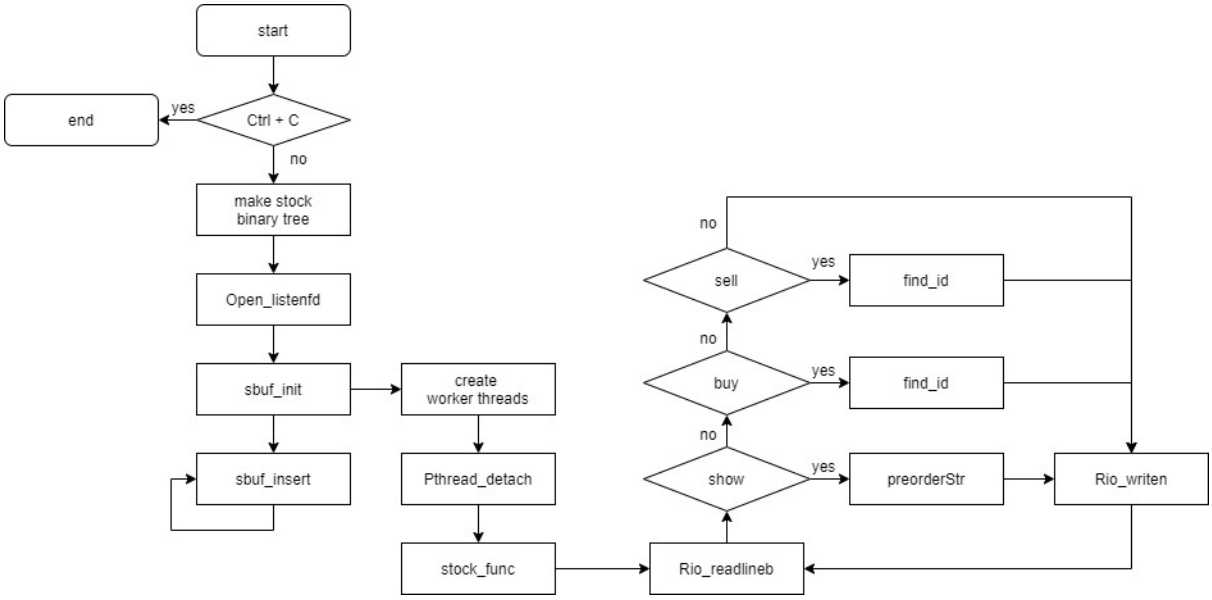
A. Flow Chart

1. select



프로그램 실행 중 Ctrl + C를 누르면 프로그램이 종료된다. 서버가 시작되면 stock.txt 파일을 열어 내용을 한줄씩 읽어서 프로그램 내 메모리에 이진 트리 자료구조로 저장한다. 다음으로 listenfd를 열고 pool을 초기화한다. 무한 루프에서 select로 이벤트를 탐지하고 add_client나 check_client를 호출한다.

2. pthread



프로그램 실행 중 Ctrl + C를 누르면 프로그램이 종료된다. 서버가 시작되면 stock.txt 파일을 열어 내용을 한줄씩 읽어서 프로그램 내 메모리에 이진 트리 자료구조로 저장한다. 다음으로 listenfd를 열고 sbuf를 초기화한다. 무한 루프에서 클라이언트의 연결을 받아 sbuf_insert를 호출한다. workder thread들은 sbuf에 있는 연결을 빼서 stock_func으로 서비스를 제공한다.

B. 제작 내용

1. select

서버에서 Rio_writen을 했을 때 ₩n때문에 show가 1줄씩 끊겨서 보내지는 문제가 있었다. 이 문제를 해결하기 위해 show에 대해 출력할 내용을 1줄로 concatenate하여 클라이언트로 보내주었다. 클라이언트에서는 서버로부터 받은 1줄의 텍스트를 공백 단위로 끊고 ID, left_stock, price 3개의 단어를 출력하면 ₩n를 출력하도록 했다. 이 방법으로 pdf에 나와있는 출력 예시와 같은 출력이 가능해졌다.

2. pthread

select에서 발생한 Rio_writen 문제에 대한 해결책은 pthread에서도 동일하게 처리를 하였다.

readers-witers 문제가 발생할 수 있으므로 이진 트리의 노드 단위(fine_grained locking)로 문제를 관리하도록 코드를 작성했다. 각 노드가 mutex와 w 세마포어를 갖고 처음 노드 생성시 초기화된다. 이진 트리는 전역 변수로 접근할 수 있으므로 critical section의 보호되어야 할 변수이다. show에서는 reader 처리를 해주었는데 각 노드가 출력되기 전에 mutex로 readcnt를 안전하게 증가시키고 readcnt가 1이라면 이진 트리에 다른 reader가 접근하지 못하도록 했다. 노드의 내용을 str 변수에 다 적고 난 이후에 mutex로 readcnt를 안전하게 감소하고 readcnt가 0이라면 이진 트리에 다른 reader가 접근할 수 있도록 했다. 이진 트리 전체에 lock을 걸지는 않았고 각 노드에 lock을 걸었다. buy에서 find_id 함수로 입력 받은 id를 가지는 노드를 반환하는데 이때 reader 관련 처리를 해줬다. mutex로 readcnt를 안전하게 증가시키고 readcnt가 1이라면 이진 트리에 다른 reader가 접근하지 못하도록 했다. 노드가 반환되고 check_clients 함수로 돌아왔을 때 이 노드에 남은 주식이 사려고 했던 주식보다 작을 때, 크거나 같을 때 모두 mutex로 readcnt를 안전하게 감소하고 readcnt가 0이라면 이진 트리에 다른 reader가 접근할 수 있도록 했다. 남은 주식이 사려고 했던 주식보다 크거나 같을 때에는 w를 사용하여 안전하게 writer가 주식의 left_stock을 감소할 수 있도록 했다. sell에서도 buy와 동일한 과정으로 구현했는데 다른 점은 find_id에서 반환된 노드가 NULL이 아닌 경우 readcnt를 buy와 동일하게 처리해주고 w를 사용하여 안전하게 writer가 주식의 left_stock을 증가할 수 있도록 했다.

pdf에 나와있는대로 클라이언트의 연결 요청을 master thread에서 처리하고 디스크립터를 buffer에 삽입하면 worker thread가 버퍼에 있는 디스크립터를 삭제하면서 클라이언트에 서비스를 제공해주는 pool of worker thread 방식을 사용했다. worker 쓰레드의 개수와 sbuf slot의 개수가 너무 적으면 성능이 좋지 않아서 적당한 개수를 설정하는 것이 중요한 문제였다. 실제 테스트를 위해 worker 쓰레드는 10개, sbuf는 50개로 진행하였다. client 수를 증가시키면 Rio_readlineb error: Connection reset by peer가 출력되고 프로그램이 종료되는 경우가 발생했다. 클라이언트가 데이터를 보냈는데 서버쪽 입력 스트림이 닫혀있을 경우에 발생하는 오류라고 한다. NTHREADS와 SBUFSIZE를 증가시켜주면 이 문제가 해결되었다.

C. 시험 및 평가 내용

(실험 환경 : 서버-cspro4, 클라이언트-cspro9, 포트-1668)

1. 확장성 : Client 개수 변화에 따른 동시 처리율 계산을 위한 elapse time

(1) client = 1, order per client = 10

select	pthread
elapsed time : 0.000145 ms	elapse time : 0.000168 ms

(2) client = 4, order per client = 10

select	pthread
elapsed time : 0.000464 ms	elapse time : 0.000492 ms

(3) client = 10, order per client = 10

select	pthread
elapsed time : 0.001175 ms	elapse time : 0.001302 ms

(4) client = 20, order per client = 10

select	pthread
elapsed time : 0.002564 ms	elapse time : 0.002753 ms

(5) client = 50, order per client = 10

select	pthread
elapsed time : 0.006859 ms	elapse time : 0.007609 ms

2. 워크로드에 따른 분석

(1) 모든 client가 buy 또는 sell을 요청하는 경우 (client = 20, order per client = 10)

select	pthread
elapsed time : 0.002609 ms	elapse time : 0.002929 ms

(2) 모든 client가 show만 요청하는 경우 (client = 20, order per client = 10)

select	pthread
elapsed time : 0.002524 ms	elapse time : 0.002963 ms

(3) client가 buy, show를 섞어서 요청하는 경우 (client = 20, order per client = 10)

select	pthread
elapsed time : 0.002727 ms	elapse time : 0.003224 ms

(4) client가 sell, show를 섞어서 요청하는 경우 (client = 20, order per client = 10)

select	pthread
elapsed time : 0.002727 ms	elapse time : 0.002696 ms

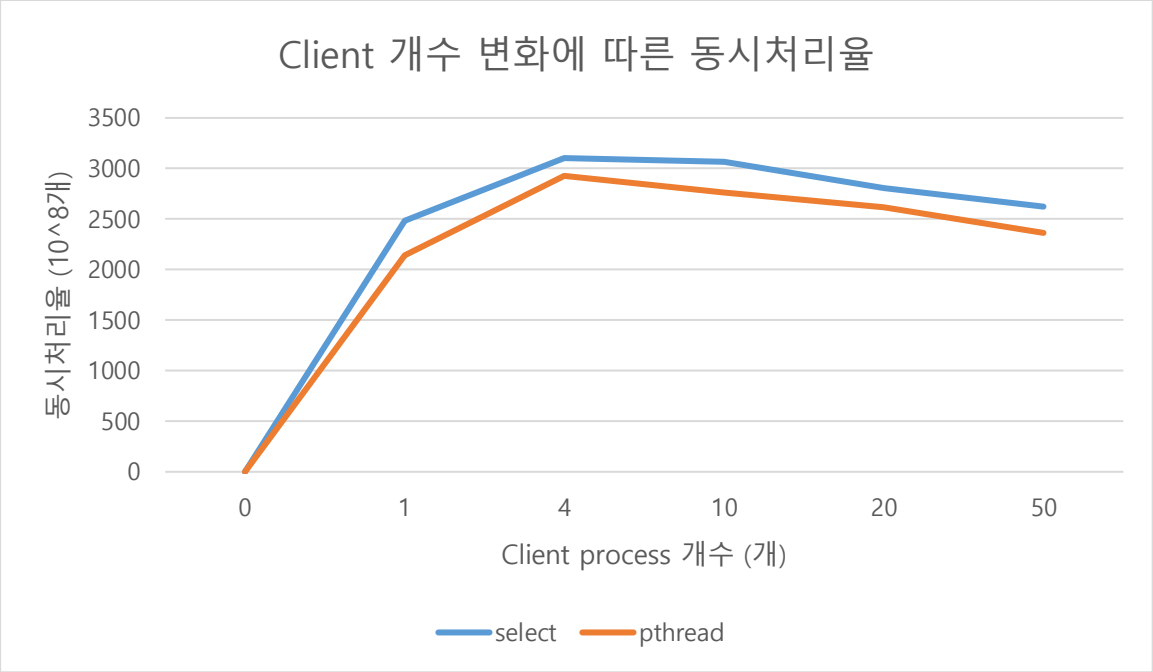
3. 예측과 결과 분석

- 예측

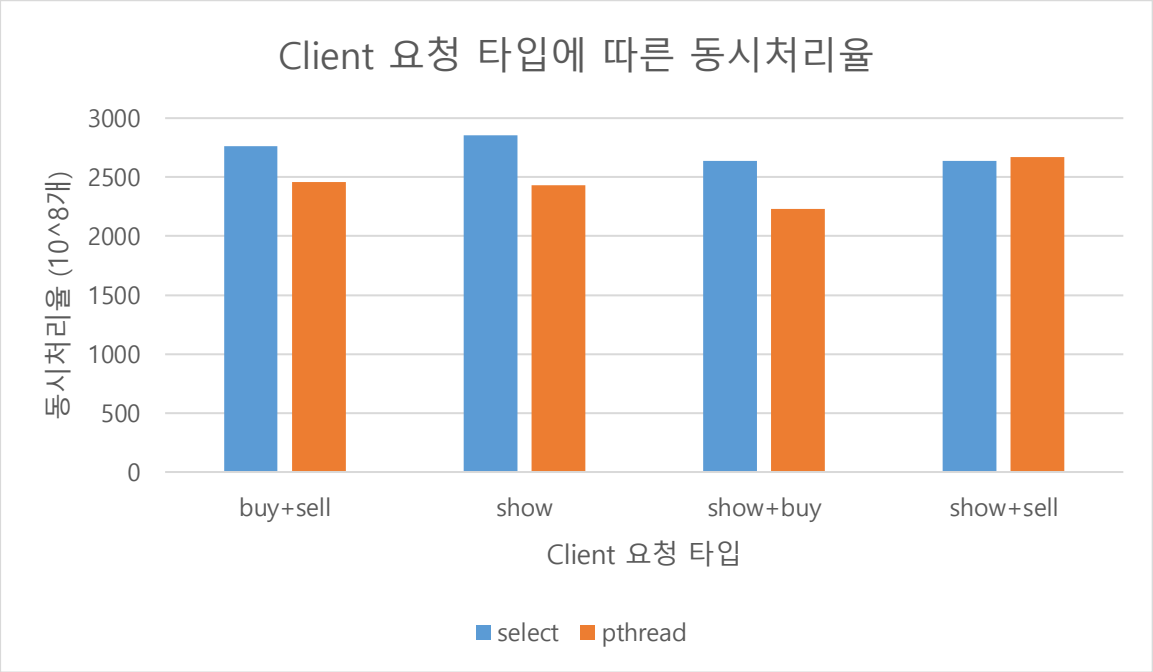
Event-driven 서버는 여러 개를 동시 연결 처리하더라도 더 적은 메모리를 요구할 것이고 동시성에서 더 나은 확장성을 제공할 것 같다. 또한 쓰레드 생성 및 소멸로 발생하는 비용이 없고 쓰레드간 context switching 비용이 없어서 수행 시간이 적게 걸릴 것 같다. 그리고 단일 쓰레드 구조이기 때문에 쓰레드 병행 수행 문제나 상호 배제 문제가 발생하지 않는다. multi thread 모델에서 lock이나 mutex의 사용은 코드 속도를 저하시키거나 오버 헤드를 증가시킬 가능성이 있다. 따라서 select 방식이 show나 buy, sell과 같이 공유 변수에 대해 세마포어를 사용하는 명령에서는 pthread보다 빠를 것으로 예측된다. 왜냐하면 lock이나 mutex 연산이 없기 때문에 지연되는 시간이 적을 것이기 때문이다.

하지만 Event-driven 서버는 single thread의 context에서 실행이 된다. 따라서 multi-core 환경을 충분히 활용하지 못할 수 있다. 이를 구현하기 위해 multi thread의 사용이 필요할 것이다. 이번 프로젝트에서 Multi-thread 서버는 thread pool 방식을 사용하였다. 연결 요청별로 쓰레드를 생성하는 모델은 쓰레드 생성/삭제 및 context switching에 의한 성능 저하가 심하기 때문이다. thread pool 방식은 쓰레드 관리 비용이 발생하지 않아 연결 요청별로 쓰레드 생성하는 모델에 비해 수행 시간이 줄어들 것이다.

(동시처리율 계산시 반올림하여 계산하였고 단위는 10^8개이다.)



	0	1	4	10	20	50
select	0	2483	3103	3064	2808	2624
pthread	0	2143	2927	2765	2615	2366



	buy+sell	show	show+buy	show+sell
select	2760	2853	2640	2640
pthread	2458	2430	2233	2670

- 결과 분석

우선 확장성에 대한 실험 결과를 보면 전체적으로 select 방식이 pthread 방식보다 높은 동시처리율을 보였다. 둘 다 client의 숫자가 증가할수록 동시처리율이 높아졌다가 완만하게 줄어드는 모양을 보였다. event-driven 서버가 더 나은 확장성을 제공할 것 같다는 예측이 맞았음을 알 수 있었다.

다음으로 워크로드에 따른 실험 결과를 보면 sell과 show만을 요청하는 경우를 제외한 3가지 경우에서 pthread 방식이 select 방식보다 elapsed time이 오래 걸렸다. elapsed time은 동시처리율에 반비례하므로 3가지 경우에서 select가 pthread보다 높은 동시처리율을 보인다. show+sell의 경우에 동시처리율이 select보다 pthread가 30*10^8개 정도 더 많다. 결과를 통해 pthread는 세마포어로 공유 변수에 대한 접근을 처리하기 때문에 시간이 더 걸릴 것이라는 예측이 맞았음을 알 수 있었다.

event-driven 방식은 프로세스나 쓰레드 컨트롤 오버헤드가 없다는 것을 수업 시간에 배웠었는데 실험 결과가 일치하게 나왔다. select 방식 서버의 elapsed time이 더 적은 것을 통해 오버헤드가 적다는 것을 알 수 있다. 수업 시간에 event-driven 방식이 작업 중간에 다른 디스크립터에 와있는 pending을 처리하지 못한다는 내용도 배웠었는데 이 내용은 실험으로 확인하지는 못했지만 만약 맞다면 이 부분이 pthread가 select보다 더 나은 점이 될 수 있을 것 같다.

메모리 관점으로 보았을 때, pthread 방식은 정해진 worker thread만큼 메모리가 사용되기 때문에 대부분의 경우에 select보다 메모리 사용이 많을 것 같다. 새로운 클라이언트에게 추가로 리소스가 필요할 것이기 때문이다.

pthread 방식도 장점이 있으므로 select 방식과 pthread 방식을 적절하게 섞어서 사용하면 좋은 성능의 프로그램을 작성할 수 있을 것 같다고 생각한다. pthread 방식은 리소스 공유가 쉽고 세분화된 동시성을 제공하기 쉽다.