Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 김동현

학번 : 20190345

1. 개발 목표

여러 클라이언트의 접속을 받아 클라이언트 요청을 concurrent하게 처리하여 응답하는 주식 서버 구축을 목표로 한다. 이때, 주식 정보는 이진 트리 자료구조를 사용하고, concurrent한 처리는 event-based approach와 thread-based approach 두 가지 접근법을 사용한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

I/O multiplexing을 사용하여 event-driven approach를 통한 주식 서버를 구축하고자 한다. 하나의 프로세스 및 하나의 메인 스레드에서 클라이언트를 accept하여 connection을 만들고, 이러한 클라이언트들의 요청을 처리하고 다시 전달한다. 클라이언트들은 add_client 함수를 이용해 pool 구조체에 저장하여 관리하며, 주식 정보는 "stock.txt"에서 주식 정보를 읽어와 이진 트리에 저장한 뒤, 서버가 종료될 때에 다시 "stock.txt"에 저장하여 정보의 휘발을 방지한다. 클라이언트의 요청은 총 3가지로, "show"는 현재 주식 정보를 보여주며, "buy"는 클라이언트가 어느 주식을 얼마나 살지에 대한 요청, "sell"은 클라이언트가 어느 주식을 얼마나 판매할지에 대한 요청이다.

2. Task 2: Thread-based Approach

Pthread interface를 사용하여 thread-based approach를 통한 주식 서버를 구축하고자 한다. 하나의 프로세스에서 여러 개의 스레드를 사용한다. 스레드는 초기 주식 정보를 생성하고, 클라이언트의 connection 요청을 받아들인 메인 스레드와, 실제 connection된 클라이언트의 요청을 처리할 worker 스레드로 나뉜다. 클라이언트와의 연결 fd는 shared buffer를 사용하여 스레드간 정보를 전달하며, 주식 정보는 semaphore를 사용하여 충돌 없이 공유된 이진 트리에 접근하고자 한다. 이외에 "stock.txt"와 클라이언트 명령어 처리는 event-driven approach와 같은 방법으로 접근한다.

3. Task 3: Performance Evaluation

"time.h" 헤더 파일을 include하여 clock()함수를 사용한다. Multiclient.c 코드의 시

작부분과 끝부분에서 시간을 체크하고 두 시간의 차이를 이용해 수행 시간을 얻고자 한다. 수행시간은 multiclient 실행이 완료되고 난 뒤, 표준 입출력을 통해 수행 시간을 출력한다. 클라이언트가 전달하는 명령어는 Rand() 함수가 포함된 statement를 조작하여 특정 요청만 전달되록 한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  Select()함수를 사용하여 multi 클라이언트 요청을 받아들이는 I/O multiplexing을 구현하고자 한다. Select() 함수는 커널에게 프로세스를 중지시키는 요청을 보낸다. 프로세스는 하나 이상의 입출력이 발생했을 경우, 즉 클라이언트의 연결 요청이 있거나, 연결된 클라이언트에서 show, buy, sell, exit과 같은 명령어를 보냈을 경우 프로세스를 재개시킨다. 이때 어떤 file descriptor에서 입력이 들어왔는지는 fd_set 자료구조에 저장되어 있다. 이를 통해 어떤 파일 디스크립터에서 입력을 받아와, 요청을 처리하고, 이에 알맞은 출력을 보내줄 수 있다. 혹은 연결 요청이 있을 경우 accept() 함수를 통해 connection을 진행하고, pool 구조체에 추가해준다. 이후 pool구조체에 저장된 연결된 디스크립터를 순회하며 입력이 있는지 확인하고, 입력이 있다면 이에 대한 요청을 수행하며 I/O multiplexing을 진행한다.

  ✓ epoll과의 차이점 서술

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

  ✓ Master Thread의 Connection 관리

  Master thread에서의 Connection관리는 sbuf 구조체를 통해 이루어진다. Sbuf는 여러개의 스레드가 공유하는 버퍼 영역으로 메인 스레드는 listenfd 소켓을 통해 클라이언트로부터 연결 요청을 받아 공유 버퍼에 해당 클라이언트 정보를 입력한다. 이때 사용하는 함수가 sbuf_insert함수이며, 이 함수 내에서는 두 개의 semaphore를 통해 race현상을 막고자 한다. 그 결과 버퍼에 더 이상 넣을 곳이 없다면 빈 공간이 생길 때까지 기다려 clientfd의 누수를 방지하며, semaphore를 통해 데이터를 저장하는 동안 다른 스레드가 접근하지 못하도록 하여 안전하게 clientfd 정보를 저장한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Woker thread는 실질적으로 클라이언트의 요청을 수행하는 스레드이다. Sbuf라는 공유 버퍼에서 현재 연결이 완료된 클라이언트의 디스크립트를 받아온다. 이후 클라이언트와 소통하며 클라리언트가 종료될 때 까지 주식 서비스를 제공한다. 스레드의 스택 영역에 선언된 클라이언트 디스크립터와 rio_t 타입의 버퍼를 사용하여 고유한 영역을 다른 스레드에 의해 침범받지 아니하고 서비스를 제공하고자 한다.

- **Task3 (Performance Evaluation)**

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

얻고자 하는 metric은 event-based approach와 thread-based approach 간의 client process 개수에 따른 시간 당 request 처리 개수인 동시 처리율을 비교하고자 한다. 또한 명령어 구성에 따른 두 접근법의 속도 차이를 살펴보고자 한다. 이러한 분석 방법을 설정한 이유는 서버의 특성상 다양한 범위의 클라이언트가 접속 할 수 있고, 클라이언트 역시 다양한 명령어 구성으로 요청을 보내온다. 이러한 차이에 따른 request 처리 결과가 곧 서버의 성능이므로 이를 파악하고자 한다. 측정 방법은 “time.h” 헤더 파일의 clock() 함수를 통해 접근법에 따라 정해진 숫자의 클라이언트가 정해진 개수의 명령어를 요청하여 대답을 받는 시간을 측정하는 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트 개수를 증가함에 따라 event-based와 thread-based 접근법 모두 동시 처리율이 향상될 것으로 보인다. Event-based는 fork로 인한 프로세스 생성 과정의 Overhead가 줄어들기 때문이다. 즉, fork로 인해 새로운 프로세스(클라이언트)가 만들어지는 것과 Concurrent하게 기존에 만들어진 프로세스(클라이언트)는 이미 서버와 통신을 진행하고 있기 때문이며 이는 프로세스의 개수가 증가할 수록 fork()에 사용되는 시간 비율이 적어진다. 반면, thread-based는 worker thread의 개수를 100으로 설정하여 진행할 예정이기 때문에 쉬고 있는 스레드 없이 다 같이 클라이언트와의 통신을 진행하기 때문에 동시 처리율이 향상 될 것으로 보인다.

또한 명령어의 종류를 다르게 했을 경우, 이진 트리에 접근하는 횟수에 따라 동시 처리율에서 차이를 보일 것으로 예상된다. show명령어는 모든 노드를 순회하

며, 버퍼에 노드를 저장해야하기 때문에 O(n)의 시간복잡도로 가장 시간이 오래 걸리는 명령어로 예상된다. 반면 sell과 buy는 모든 노드를 순회하는 것이 아닌 특정 노드를 찾아가는 것이기 때문에 O(logn)의 시간복잡도가 소요될 것으로 예상된다. 따라서 sell 또는 buy 명령어만 사용되는 케이스가 show 명령어만 사용되는 케이스보다 동시 처리율이 높으며, 모든 명령어를 사용하는 경우가 그 중간의 동시 처리율을 보일 것으로 예상된다.

task 간의 동시 처리율은 thread-based 접근법이 event-based 접근법보다 높을 것으로 예상된다. 이는 하나의 프로세스로 실행된다는 점은 같지만, thread-based 접근법은 여러개의 worker 스레드를 사용하는 반면, event-based 접근법은 하나의 단일 스레드만을 사용하여 모든 클라이언트의 요청을 처리하는 데에 성능 차이가 발생할 것이다. 하지만 스레드의 공유 변수 접근을 막기 위해 사용하는 semaphore의 overhead로 인해 그 차이는 크지 않고, 명령어의 구성에 따라 성능 차이가 발생하지 않는 경우도 있을 것이라 예상한다.

C. 개발 방법

주식 정보를 이진 트리로 구성하기 위해 또한 이진 트리를 파일에 출력하기 위해 다음과 같은 자료구조 및 함수를 구현하였다.

1) Struct item

Item구조체는 다음과 같은 정보를 담고 있다. 주식의 ID(ID), 남은 개수(left_stock), 가격(Price), 현재 해당 아이템을 참조하고 있는 스레드 개수(readcnt), semaphore for mutual exclusive(mutex), 부모 item 포인터(parent), 본인보다 ID가 작은 자식 아이템을 가리키는 포인터(Left_child), 본인보다 ID가 큰 자식 아이템을 가리키는 포인터(right_child)가 element로 선언되어 있다. 또한 이러한 item 구조체를 원소로 하는 이진 트리의 head 노드를 가리키기 위한 전역 변수 item* head가 선언되었다.

2) Init_stock(), writeStockToFile(), file_print_stock(), deallocate_stock(), sigint_hander()

서버가 시작되고 나면 "stock.txt" 파일로부터 주식 정보를 읽어와 이진 트리를 구성하고자 한다. 이를 위해 init_stock() 함수를 구현하였으며, 이 함수에서는 "stock.txt" 파일을 열어 파일에 저장된 주식의 id, left_stock, price를 읽어온다. 저

장된 정보는 동적 할당을 통해 생성된 item노드에 초기화된다. 이후 새로 만들어진 노드를 인자로 하는 insertItemToBinTree를 호출하여 이진 트리에 주식 정보를 삽입한다. 서버가 종료될 때, 변경된 주식 정보를 다시 "stock.txt"에 출력하기 위해, writeStockToFile(), file_print_stock() 함수를 구현하였다. 위 함수를 통해 파일을 열고, 이진 트리를 재귀로 순회하며 모든 노드를 방문해 파일에 주식 정보를 출력한다. 또한 deallocate_stock() 함수를 통해 프로그램 종료 전 힙 영역에 할당되었던 모든 Item노드를 재귀로 순회하며 Free 시켜준다. 이러한 파일 출력 및 메모리 반환 과정은 서버가 종료될 때 수행되며, 서버를 종료시키는 방법은 ctrl+c를 통한 SIGINT 시그널의 전달이다. 이를 위해 SIGINT signal에 대한 signal_hander를 singint_handler를 구현하였다.

3) insertItemToBinTree(), find_stock_item(), print_stock()

이진 트리에 item 노드를 삽입하기 위해 insertItemToBinTree() 함수를 구현하였다. 인자로 들어온 노드 포인터를 알맞는 이진 트리 위치에 구현하기 위해, curr 포인터가 가리키는 노드와의 비교를 통해 삽입에 적절한 위치로 이동한다. 이때 기준은 주식의 ID로 현재 노드보다 id가 작을 경우 왼쪽 자식으로, id가 클 경우 오른쪽에 삽입되며, 이는 자식 노드가 null일 때까지 반복된다.

이진 트리를 순회하는 목적은 2가지이다. 원하는 노드를 찾거나 모든 노드를 순회하는 것이다. 순회는 재귀를 통해 이루어진다. 원하는 노드를 찾기 위해서 find_stock_item() 함수를 구현하였으며 ID를 비교해가며 알맞은 자식 노드로 내려간다. 이때 원하는 노드를 찾으면 해당 포인터를 반환한다. 모든 노드를 순회하기 위해서 print_stock() 함수를 구현하였으며, show 요청을 위한 함수이다. 이를 통해 모든 이진 트리를 순회하며 인자로 들어온 Result 버퍼에 해당 노드 정보를 덧붙여준다.

클라이언트의 요청을 처리하기 위해 다음과 같은 함수를 구현하였다.

1) handle_client_request()

Strncmp 내장 함수를 통해 클라이언트의 요청이 저장된 버퍼의 첫 부분을 확인한다. 클라이언트로 들어올 수 있는 명령어는 총 4개로 show, buy, sell이다. 각각 명령어가 확인되면, 해당 명령어를 수행하는 함수를 호출한다. 이때 추가적인 인자가 필요하다면 parsing을 통해 전달한다. Exit 요청에 대해서는 check_client() 함수에서 미리 처리하였다.

2) command_show()

Result 버퍼에 모든 주식의 id, left_stock, price의 정보를 담아 출력해주는 함수이다. 모든 이진 트리를 순회하여 result 버퍼에 정보를 담기위해 print_stock() 함수를 호출한다.

4) command_sell()

인자로 들어온 주식 id를 cnt개수만큼 파는 함수이다. Find_stock_item() 함수를 사용하여 해당 id 주식을 가리키는 노드 포인터를 반환받는다. 이 포인터가 NULL이라면 해당 id의 주식이 없으므로 요청을 수행하지 못했음을 알려준다. 그렇지 않다면, 해당 노드의 남은 주식의 개수에 인자로 들어온 팔 주식의 개수를 추가해준다.

5) command_buy()

인자로 들어온 주식 id를 cnt개수만큼 사는 함수이다. Find_stock_item() 함수를 사용하여 해당 id 주식을 가리키는 노드 포인터를 반환받는다. 이 포인터가 NULL이라면 해당 id의 주식이 없으므로 요청을 수행하지 못했음을 알려준다. 또한 해당 노드의 남은 주식 개수가 사려고 하는 개수보다 적다면 남은 주식이 부족함을 알려준다. 그렇지 않다면, 해당 노드의 남은 주식의 개수에 인자로 들어온 살 주식의 개수를 빼준다.

I/O multiplexing을 통한 event-based approach를 위해 다음과 같은 자료구조 및 함수를 구현하였다.

1) Struct pool, init_pool

서버와 connection을 이루고 있는 클라이언트의 디스크립터와 버퍼를 담기 위한 구조체이다. FD_SETSIZE의 개수만큼 선언된 배열 clientfd와 clientrio는 각각 descriptor 번호와 버퍼를 나타낸다. Read_set은 연결되어 있는 디스크립터를 저장하고 있는 fd_set 자료형이며, ready_set은 입력이 들어온 디스크립터를 저장하고 있는 fd_set 자료형이다. Nready는 입력이 들어온 디스크립터의 개수이다. Maxfd는 현재 read_set에 존재하는 디스크립터 번호중 가장 큰 디스크립터 번호를 나타내며, maxi는 그에 대한 인덱스를 나타낸다. 이를 통해 연결된 디스크립터를 순회할 때, 마지막 디스크립터 번호를 저장하며 불필요한 탐색을 방지한다. Init_pool()은 Pool 구조체를 초기화하는 함수로 서버가 시작되면 호출하여 클라이언트 정보를 담을 수 있도록 초기화한다. Clientfd 배열의 원소들을 -1로 초기

화하여 모두 비어 있는 상태로 만들며, 클라이언트의 입력을 받아들일 서버의 listenfd를 pool 구조체에 추가한다.

2) Main 함수의 수정

Open_listenfd() 함수를 호출하여 listenfd 네트워크 소켓을 반환받는다. 이는 서버가 클라이언트로부터 connection을 요청받는 디스크립터이다. Signal() 함수를 통해 SIGINT 시그널에 대한 핸들러를 초기화한다. 이 함수는 서버 터미널에서 SIGINT(ctrl+c)을 받는다면, 서버 프로세스가 종료되는데, 종료하기 전 그 동안 클라이언트와의 통신으로 업데이트 된 주식 정보를 보전하기 위해 "stock.txt"에 저장하고 할당된 동적 할당 영역(이진 트리)을 반환한다. Init_pool(), init_stock() 함수를 호출하여 Pool구조체와 이진 트리를 활용한 주식 정보를 초기화 한다. 이후 반복문을 돌며, select 함수를 통해 ready_set에 있는 클라이언트의 입력이 들어올 때까지 기다린다. 이후 새롭게 서버에 접속하고자 하는 클라이언트의 요청이 있다면, accept() 함수를 통해 서버와 클라이언트를 연결하고 add_client() 함수를 호출해 pool 구조체에 새롭게 연결된 클라이언트를 추가해준다. 그렇지 않고 기존 연결되어 있는 클라이언트에서 입력이 들어왔을 경우 Check_clients() 함수를 호출하여 주식 서비스를 제공한다.

3) Add_client()

새롭게 연결된 클라이언트의 디스크립터를 인자로 받아 pool 구조체에 저장하는 함수이다. Pool함수 내의 clientfd 배열을 순회하며 빈 곳에 해당 클라이언트 디스크립터를 저장하고, 버퍼를 초기화한다. 이후 Maxfd, maxi의 변경이 있으면 값을 변경한다.

4) Check_client()

기존에 연결된 클라이언트의 디스크립터 중 입력이 있는 경우 해당 디스크립터의 버퍼를 읽어 이에 맞는 서비스를 제공하는 함수를 호출하는 함수이다. Pool 구조체의 clientfd 배열을 순회하며, ready_set에 있는 인자, 즉 버퍼로부터 입력이 들어온 디스크립터를 확인한다. 입력이 있는 경우 버퍼로 부터 요청을 읽어들여 handle_client_request() 함수를 호출하여 주식 서비스를 제공한다. 만약 exit 요청 혹은 EOF가 들어온 경우 서버와의 연결을 종료하고자 하므로, close() 함수를 호출하여 클라이언트와의 연결을 종료하고, clientfd 배열에서도 해당 디스크립터를 제거한다.

Pthread를 통한 thread-based approach를 위해 다음과 같은 자료구조 및 함수를 구현하였다.

1) struct sbuf_t, subf_init(), sbuf_deinit(), sbuf_insert(), sbuf_remove()

Sbuf_t 구조체는 클라이언트의 연결 요청을 받는 메인 스레드와 어떤 클라이언트에게 주식 서버에 관한 서비스를 제공하는 워커 스레드간의 데이터 공유를 위한 자료구조이다. 클라이언트의 디스크립터 번호를 저장하는 버퍼로 buf 포인터를 동적할당하여 값을 저장한다. N은 저장할 수 있는 원소의 최대 개수이며, front와 rear는 각각 버퍼에 담겨있는 원소의 처음과 마지막 원소의 인덱스를 가리키는 정수값으로 front는 버퍼에서 가장 먼저 빼와야할 값을, rear는 버퍼에서 가장 먼저 넣어야할 곳을 가리키고 있다. Slots와 items는 sem_t 자료형으로서 현재 버퍼에서 비어있는 원소의 개소와 데이터가 들어있는 원소의 개수를 나타낸다. 이때 sem_t의 자료형으로 선언한 이유는 원소의 추가나 삭제 요청이 있지만 버퍼가 꽉차 있거나 비어있을 경우 해당 추가 삭제를 무시하는 것이 아니라 대기하여 가능한 상태가 되었을 때, 해당 명령을 수행하기 위해서이다. Mutex는 버퍼에 추가나 삭제가 이러날 경우 atomic하게 버퍼를 업데이트 하기 위한 semaphore이다. 이러한 공유 버퍼 sbuf의 초기화는 sbuf_init(), 반환은 subf_deinit()을 통해 이루어지며, sbuf_insert()는 공유 버퍼에 클라이언트 디스크립터를 삽입할 때 호출하는 함수로 마스터 클라이언트가 클라이언트 연결 요청을 받은 뒤 공유 버퍼에 데이터를 삽입할 때, 호출한다. Sbuf_remove()는 공유 버퍼에서 클라이언트 디스크립터를 꺼내올 때 호출하는 함수로 워커 클라이언트가 아무런 클라이언트에게도 서비스를 제공하지 않을 때, 서비스를 제공한 클라이언트를 고르는 과정에서 호출한다.

2) struct item의 수정

Item구조체에 rw_mutex의 sem_t 자료형을 추가하였다. 이는 스레드 간의 readers writers problem을 해결하기 위해 추가로 선언하였다. 클라이언트의 요청은 반드시 주식 정보가 담겨있는 노드에 접근한다. Show 명령어는 reader로서, buy, sell 명령어는 writer로서 노드에 접근한다. 여러 스레드가 클라이언트의 요청을 처리하기 위해 노드를 순회하는데, 이때 readers writers problem이 발생할 수 있다. 이를 해결하기 위해 rw_mutex를 선언하였다. 현재 노드에 접근하여 reading하고 있는 스레드가 자신이 유일하다면 해당 스레드는 P함수를 호출하여 writer 스레드의 접근을 대기 시킨다. 이후 마지막 reading 스레드가 작업을 끝내면, V함수를 호출한다. 이때 대기하고 있는 writer 스레드가 있다면 writer 스레드는 P 함수를 호출하여 다른 모든 스레드가 접근하고자 할 때 스레드를 대기시키고, 작업이 끝나면 V함수를 호출하여 대기하고 있는 스레드가 접근할 수 있도록 한다.

3) thread()

Worker 스레드가 생성되었을 때, 실행되는 context로 pthread_create()함수의 인자로 전달된다. 스레드가 생성되고 난 뒤, pthread_detach를 통해 스레드가 종료되고 난 뒤, reaping 과정을 커널이 해주도록 한다. 이후 무한 루프를 돌며 클라이언트에게 주식 서비스를 제공한다. 공유 버퍼에 클라이언트 디스크립터가 있을 경우 sbuf_remove를 통해 해당 클라이언트 디스크립터를 가지고 온다. 이후 echo_service() 함수를 통해 클라이언트에게 주식 서비스를 제공하도록 한다.

4) echo_service

Worker thread에서 실행되는 thread() 함수에서 호출되는 함수로 실질적으로 클라이언트에게 주식 서비스를 제공하는 함수이다. 기존 task_1의 check_client() 함수를 응용한 것으로 버퍼와 rio_t 데이터를 지역 변수로 선언하여 스레드 각각의 고유한 영역으로 설정한다. 이후 반복해서 rio_readlineb() 함수를 호출하여 클라이언트의 요청을 받는다. 이때 byte_cnt는 현재까지 모든 스레드에서 입력받은 바이트 수로 전역 변수로 선언되어 있다. 이 전역 변수는 모든 스레드에서 접근할 수 있기 때문에 race 발생 위험이 있다. 이를 위해 semaphore를 사용하여 btye_cnt를 읽어오고, 더하고, 저장하는 과정을 atomic하게 수행하도로 한다. 이후 클라이언트의 입력을 확인하여 exit이라면 클라리언트와의 연결을 종료하고, 그렇지 않은 경우에는 handle_client_request함수를 호출하여 기존 task_1의 함수를 재사용하였다.

5) main 함수의 수정

스레드를 통한 주식 서비스 제공을 위해 만든 자료구조인 sbuf와 semaphore를 초기화하는 함수를 추가하였다. 이후 worker 스레드를 NTHREAD 만큼 반복하여 생성한다. 각각의 스레드는 thread() 함수로 진입하여 클라이언트의 연결을 대기하게 된다. 이후 무한 루프를 통해 클라이언트의 연결을 기다리며, 클라이언트의 연결이 성공할 경우 클라이언트의 디스크립터 번호를 sbuf_insert() 함수를 통해 공유 버퍼에 삽입하여 스레드에게 전달한다.

6) print_stock(), command_sell(), command_buy() 함수의 수정

위 3개의 함수는 힙 영역에 저장되어 있는 주식 정보, 즉 모든 스레드가 접근할 수 잇는 이진 트리에 접근하는 함수이다. 이러한 특징으로 인해 스레드 간의 race problem, readers writers problem이 발생할 수 있다. 이를 위해 semaphore를 사용하는 구문을 추가하였다. Print_stock() 함수는 show 명령어를 통해 호출되는 함수로 reader 역할을 수행

한다. 이는 reader 스레드가 해당 데이터에 접근하고 있을 경우 writer 스레드에 의해 데이터 값이 변경되는 것을 방지해야 함을 의미한다. 이를 위해 접근하고자 하는 해당 item 구조체의 인자인 semaphore를 P함수와 V함수를 통해 atomic한 접근을 보장하여 안전하게 값을 읽고 업데이트 할 수 있도록 한다. Command_sell() 함수와 command_buy()함수는 sell, buy 명령어를 통해 호출되는 함수로 writer 역할을 수행한다. Writer 스레드의 안전한 데이터 업데이트를 위해 해당 노드의 semaphore를 사용하여 데이터의 atomic한 업데이트 과정을 보장한다. 나머지 부분은 기존 task_1의 실행 흐름과 동일하다.

<time.h>를 통한 performance evaluation을 위해 다음과 같은 코드를 추가하였다.

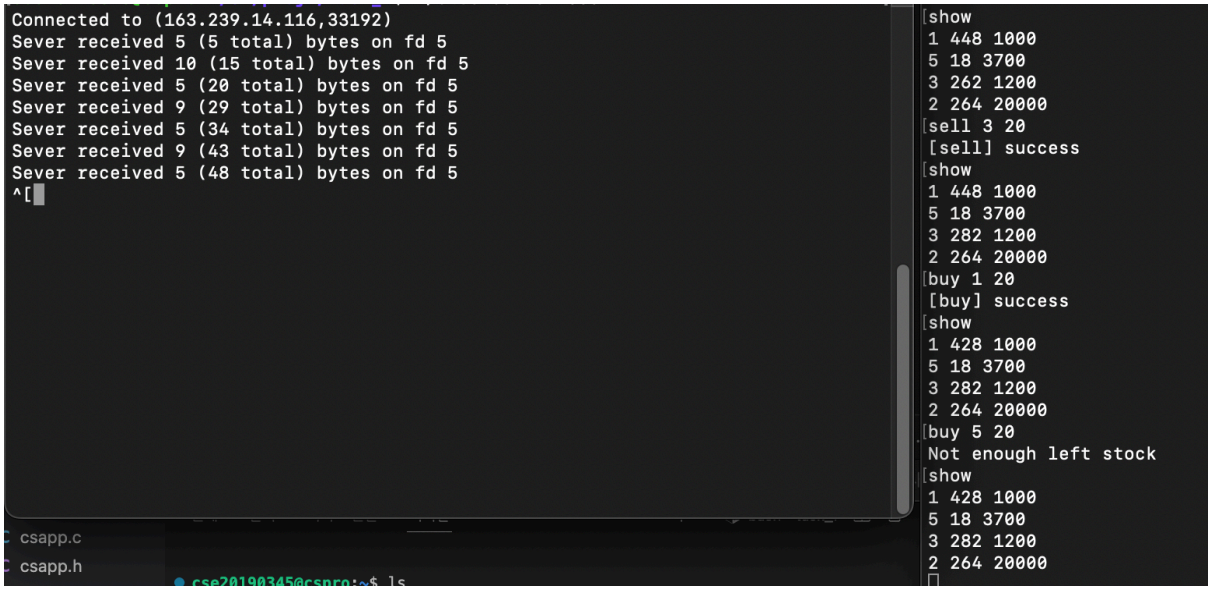
우선 multiclient.c 파일을 통해 수행 시간을 측정하였다. Multiclient에서 모든 초기화 과정을 마지고, fork() 함수를 통해 프로세스(클라이언트)를 복제하기 전, clock() 함수를 통해 현재 시각을 측정한다. 이후 모든 프로세스가 reaping되고 난 이후 다시 clock() 함수를 통해 현재 시각을 측정한다. 이 두 값의 차이를 CLOCKS_PER_SEC으로 나누어 터미널에 프린트 하여 총 수행시간을 구한다. 이때 task_2에서 worker thread의 개수는 4로 고정하였으며, 클라이언트의 개수는 5개, 20개, 50개, 100개 순으로 생성하였다. 또한 클라이언트가 요청하는 경우는 1) show만 요청, 2)sell, buy만 요청, 3) 모든 명령어 요청이며, 해당 요청 구성에 따라 시간을 측정하였다. 또한 하나의 케이스 당 3번씩 실행하여 평균을 구하였다.

3. 구현 결과

1) 서버의 실행 및 클라이언트 접속

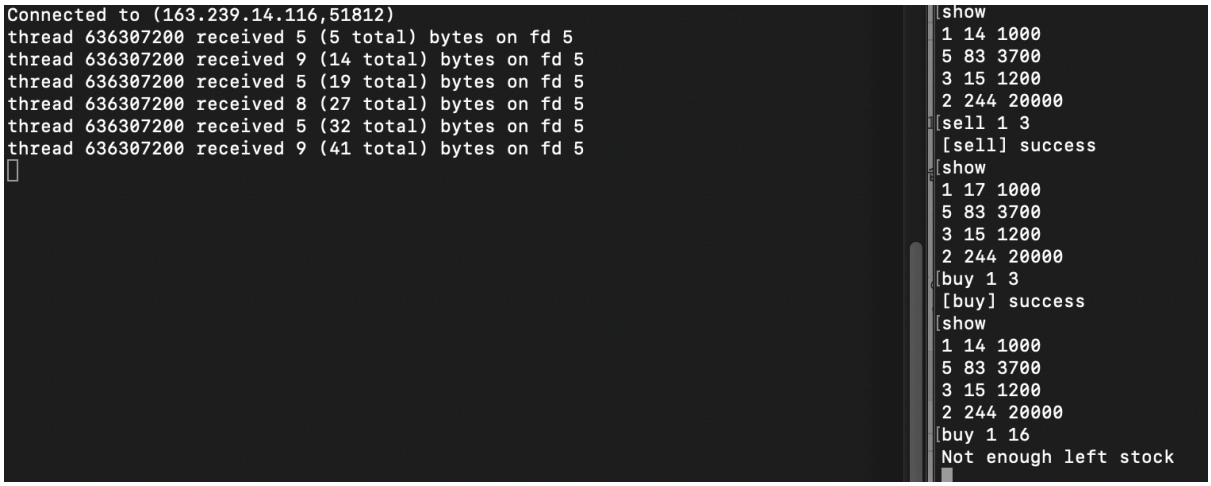
좌측이 서버를 실행할 터미널이며, 우측이 클라이언트를 실행할 터미널이다. 각각 cspro.sogang.ac.kr과 cspro9.sogang.ac.kr로 실행한 터미널이다. 좌측 터미널에서 ./stockserver 60026 명령어를 통해 60026번 포트로 서버를 오픈하였다. 이후 ./stockclient cspro.sogang.ac.kr를 통해 클라이언트 프로세스를 생성하여 서버에 연결한다.

2) Event-based approach를 통한 주식 서비스 제공



Task_1으로 서버와 클라이언트(stockclient)를 실행하여 show, sell, buy 명령어를 수행한 모습니다. 좌측 서버에서는 클라이언트의 연결 여부와 받은 명령어의 바이트 수를 출력하고 있다. 우측 클라이언트에서는 클라이언트가 요청을 서버에 보내면 서버에서 이에 대한 결과를 보여주고 클라이언트의 터미널에 출력된다. Show를 통해 모든 주식 정보를 확인 할 수 있으며, sell, buy를 통해 주식을 사고 팔 수 있다. 이때, 기존에 있는 주식보다 많은 양의 주식을 사고자한다면, 주식이 부족하다는 문자와 함께 주식 구매가 이루어지지 않는다.

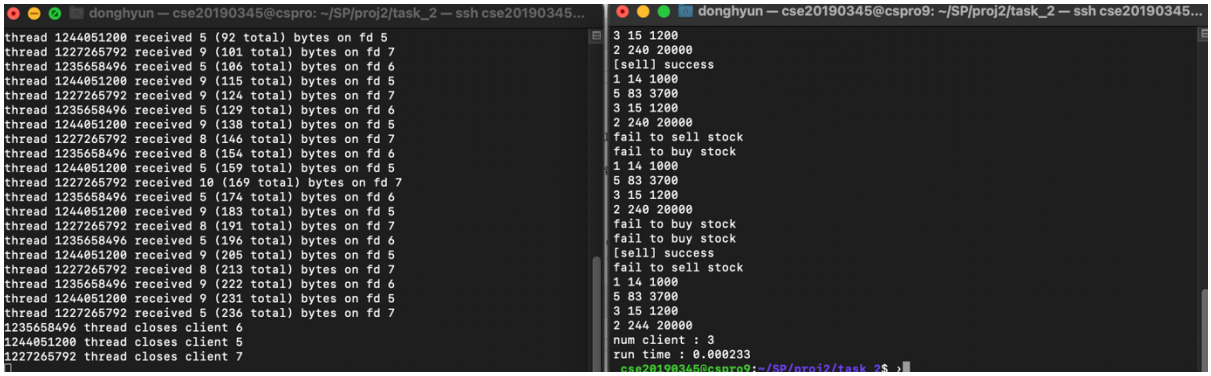
3) Thread-based approach를 통한 주식 서비스 제공



Task_2로 주식 서버와 클라이언트(stockclient)를 실행하여 show, sell, buy 명령어를 수행한 모습니다. Task_1과 마찬가지로 좌측 서버에서는 클라이언트의 연결 여부와 받은 명령어의 바이트 수를 출력하고 있다. 추가적으로 요청을 받은 스레드의 번호 역시 출력하고 있어 여러 클라이언트가 접속한다면, 여러 스레드가 처리하는 것을 확인할 수 있다. 우측

클라이언트에서는 클라이언트가 요청을 서버에 보내면 서버에서 이에 대한 결과를 보여주고 클라이언트의 터미널에 출력된다. Show를 통해 모든 주식 정보를 확인 할 수 있으며, sell, buy를 통해 주식을 사고 팔 수 있다. 이때, 기존에 있는 주식보다 많은 양의 주식을 사고자한다면, 주식이 부족하다는 문자와 함께 주식 구매가 이루어지지 않는다.

4) Performance evaluation을 위한 시간 측정



Multiclient.c를 실행한다면, 클라이언트의 모든 요청에 대한 응답을 출력하고 난 뒤, 현재 실행된 클라이언트의 개수와 걸린 시간이 출력되는 것을 확인할 수 있다.

4. 성능 평가 결과 (Task 3)

측정 시작 시점은 start=clock()으로 모든 초기화를 끝내고 fork()를 통해 클라이언트가 될 자식 프로세스를 생성하기 직전이다. 측정 종료 시점은 end=clock()으로 모든 프로세스가 terminate되어 부모 프로세스에 의해 reaping되고 난 직후이다. 이 사이 프로세스들은 하나의 클라이언트가 되어 서버와 Connection을 형성하고 랜덤으로 만들어진 요청을 서버에 보내고 수신해 터미널에 출력한다. 이후 두 시간 측정 값의 차이를 통해 수행 시간을 측정한다. 다음은 multiclient 실행 파일을 실행하기 위해 터미널에 입력한 명령어이다. 클라이언트 개수를 다양하게 설정하여 클라이언트 개수에 따른 실행시간을 분석하였다.

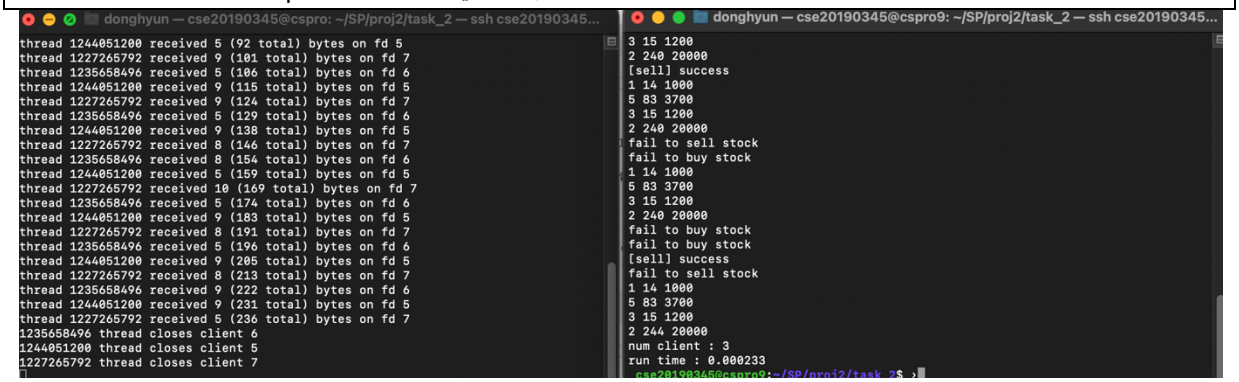
```
./multiclient cspro.sogang.ac.kr 60026 4
./multiclient cspro.sogang.ac.kr 60026 20
./multiclient cspro.sogang.ac.kr 60026 50
./multiclient cspro.sogang.ac.kr 60026 100
```

또한 client의 요청 타입을 1) show만 요청 2) buy & sell만 요청 3) 모든 명령어 요청으로 나누어서 보내기 위해 다음과 같은 코드를 사용하였다.

```
Only show : int option = 0;
```

Sell & buy : int option = rand() % 2 + 1;

All command : int option = rand() % 3;



실제 서버와 클라이언트를 실행하면 다음과 같이 클라이언트의 개수와 실행 시간을 터미널에 출력하여 결과를 확인할 수 있다. 이와 같은 과정을 통해 event-based와 thread-based 접근법의 측정 시간을 표로 정리하면 다음과 같다. 시간의 단위는 초 단위이다.

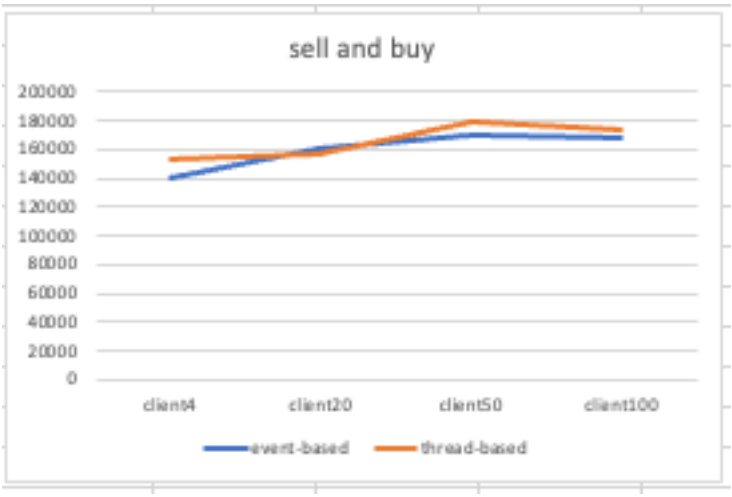
event-based				
buy or sell	clinet 4	client 20	client 50	client 100
test1	0.000311	0.001252	0.002971	0.005983
test2	0.000283	0.001245	0.002994	0.005879
test3	0.000259	0.001217	0.002865	0.006053
average	0.0002843	0.001238	0.0029433	0.0059717
only show	clinet 4	client 20	client 50	client 100
test1	0.000267	0.001292	0.003037	0.006306
test2	0.000251	0.001202	0.003095	0.006023
test3	0.000275	0.001228	0.002955	0.006324
average	0.0002643	0.0012407	0.003029	0.0062177
all command	clinet 4	client 20	client 50	client 100
test1	0.000267	0.001237	0.003025	0.006173
test2	0.000263	0.00126	0.003172	0.005968
test3	0.000261	0.001227	0.003245	0.005856
average	0.0002637	0.0012413	0.0031473	0.0060705

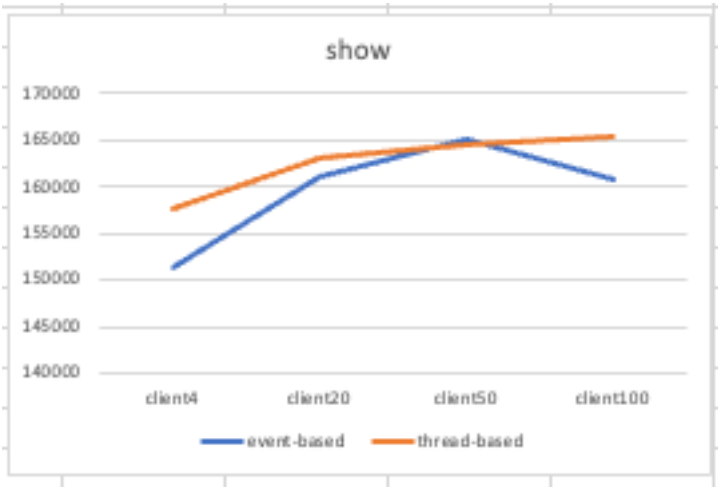
thread-based				
buy or sell	client 4	client 20	client 50	client 100
test1	0.000253	0.001254	0.002956	0.005745
test2	0.000263	0.001237	0.002545	0.005711
test3	0.000266	0.001332	0.002856	0.005845
average	0.0002607	0.0012743	0.0027857	0.005767
only show	clinet 4	client 20	client 50	client 100
test1	0.000248	0.001275	0.003022	0.006117
test2	0.000252	0.001199	0.003098	0.006009
test3	0.000261	0.001205	0.002991	0.006003
average	0.0002537	0.0012263	0.003037	0.006043
all command	clinet 4	client 20	client 50	client 100
test1	0.00026	0.001212	0.002966	0.005959
test2	0.00027	0.001237	0.002993	0.006012
test3	0.000248	0.001183	0.002926	0.005931
average	0.0002593	0.0012107	0.0029617	0.0059673

이를 통해 동시 처리율(총 요청 개수 / 시간)을 명령어 구성 단위로 정리하면 다음과 같이 나온다.

buy or sell	client4	client20	client50	client100
event-based	140679.95	161550.89	169875.42	167457.44
thread-based	153452.69	156944.81	179490.25	173400.38
only show	client4	client20	client50	client100
event-based	151324.09	161203.65	165070.98	160832.04
thread-based	157687.25	163087.8	164636.15	165480.72
all command	client4	client20	client50	client100
event-based	151706.7	161117.08	158864.65	164731.08
thread-based	154241.65	165198.24	168823.86	167579.04

위 표를 그래프로 표현하였다.





위에서 예상한 바와 같이 전체적으로 스레드를 사용한 주식 서버가 단위 시간당 더 많은 클라이언트의 명령어를 처리하는 양상을 보이고 있다. 이는 단일 스레드를 사용하여 하나의 스레드가 모든 명령을 처리하는 event-based 방식과 달리 thread-based는 다중 스레드를 사용해 동시에 명령을 처리하기 때문이다. 또한 클라이언트가 요청하는 명령어

구성에 따라서, show만 요청하는 경우, 모든 명령어를 요청하는 경우, sell과 buy만 요청하는 경우 순으로 낮은 동시 처리율을 보이고 있다. 이는 모든 노드를 순회하는 show의 시간복잡도 O(n)과 특정 노드에만 접근하는 sell, buy의 시간복잡도 O(logn)의 차이때문으로 확인되며 semaphore의 영향이 이진 트리 자료구조의 시간복잡도에 큰 영향을 끼치지 않는 것으로 보인다. 특정 경우에서 event-based 접근법이 thread-based 접근법보다 높은 동시 처리율을 보이는 경우도 있는데, 이는 스레드 간의 접근 순서가 겹치는 경우가 많아 semaphore로 인해 스레드가 대기하는 시간이 길어졌기 때문으로 분석할 수 있다.