Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 박정주

학번 : 20200185

1. 개발 목표

클라이언트로부터 show, sell, buy, exit 요청을 받을 수 있는 Concurrent stock server를 구현한다. 주식 서버는 stock.txt를 읽어 이진 트리 형태로 저장한 뒤, 여러 개의 client를 동시에 서비스한다. 서버가 종료될 때 주식의 상태를 stock.txt에 저장하여, 변경사항이 다음 실행 때에도 반영될 수 있도록 한다.

이러한 서버를 event-based와 thread-based 두 가지 방법으로 구현한다. 각 방법에 대해, 클라이언트 개수와 클라이언트의 요청 유형에 따른 동시처리율을 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

단일 프로세스, 단일 스레드만을 사용하여 concurrency를 구현한다. select함수를 통해 connfd들이 저장된 pool을 감시하고 pending input이 발생했을 경우 알맞은 처리를 해준다. 이를 통해 여러 클라이언트가 동시에 주식 서버에 접속해 요청을 보낼 수 있게 된다.

2. Task 2: Thread-based Approach

여러 개의 thread를 사용하여 concurrency를 구현한다. 여러 개의 스레드를 미리 준비해두고 connfd를 sbuf에 넣는다. 각 스레드가 sbuf에서 connfd를 꺼내 클라이언트를 서비스한다. . 이를 통해 여러 클라이언트가 동시에 주식 서버에 접속해 요청을 보낼 수 있게 된다.

3. Task 3: Performance Evaluation

Multiclient의 실행 시작부터 끝까지 걸리는 시간을 microsecond 단위로 출력한다. 파일을 수정하여 클라이언트 수와 클라이언트의 요청 유형에 따른 동시처리율 변화를 측정한다.

B. 개발 내용

- **Task1 (Event-driven Approach with select())**
  - ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

서버는 select 함수를 호출하는 것을 통해, 어떤 파일 디스크립터에 I/O 이벤트가 발생하여 처리할 pending input이 생겼는지 감시한다. Listening descriptor에 pending input이 발생한 경우, accept 함수가 리턴한 connfd를 add_client를 호출하여 pool에 추가한다. pool에 기록된 각 connected descriptor에 대해서는, check_clients를 호출하여 서비스한다.

✓ epoll과의 차이점 서술

select 함수는 모든 파일 디스크립터를 순회하며 I/O가 발생하였는지 검사하기 때문에, 파일 디스크립터의 수가 많아지면 성능이 저하된다는 단점이 있다. 반면 epoll 함수는 I/O 이벤트가 발생한 파일 디스크립터만 검사하기 때문에, 파일 디스크립터의 수와 관계없이 항상 일정한 성능을 가진다. 따라서 대규모의 연결을 처리하기 위해서는 epoll 함수를 사용하는 것이 유리하다.

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

✓ Master Thread의 Connection 관리

Master thread는 서버와 클라이언트들을 연결하는 각 connfd들을 shared buffer에 추가한다. 각 Worker thread들은 이 shared buffer에서 connfd를 제거한 뒤에, 해당 클라이언트를 서비스한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

각 worker thread는 Pthread_detach(pthread_self())를 통해 메인 스레드에서 스스로를 분리시킨다. Sbuf_remove를 통해 connfd를 shared buffer에서 제거한 뒤, echo_cnt(connfd)를 호출하여 클라이언트의 명령어에 따라 답변을 전송한 후 connfd를 닫는다. Detach된 스레드는 자기 자신이 종료되는 즉시 모든 자원을 반납한다.

- **Task3 (Performance Evaluation)**

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

동시처리율 = (클라이언트의 개수 * ORDER_PER_CLIENT) / elapsed time(s)

클라이언트의 개수와 클라이언트 당 명령의 개수를 곱한 값을 Multiclient의 실행

시간(초)으로 나눈 값을 동시처리율로 정하였다. Multiclient의 실행 시간은 getimeofday 함수를 사용하여 구하였다. 클라이언트 당 명령의 개수는 multiclient.c에 정의된 ORDER_PER_CLIENT의 값을 변경하여 조절하였다. 클라이언트의 개수는 multiclient 실행 파일을 실행시킬 때 인자로 전달한다. 이를 통해, 서버가 1초 당 몇 개의 클라이언트 요청을 처리할 수 있는지 구할 수 있다.

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트의 수가 증가하면, thread based server 는 여러 개의 스레드를 만들고 관리하기 위한 컨트롤 오버헤드가 발생하기 때문에, 미리 준비해 둔 스레드보다 많은 양의 스레드가 필요한 경우 event based server 에 비해 elapsed time 이 더 크게 증가할 것으로 예상된다.

클라이언트 요청 타입에 따른 동시처리율 변화는 크지 않을 것으로 예상된다.

C. 개발 방법

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

Event-based 서버를 구현하기 위해 stocksever.c 소스코드를 수정하였다. connected descriptor들을 저장할 구조체 pool을 선언하였다. 주식 정보를 저장할 구조체 item, 이진 트리의 노드가 될 구조체 node를 선언하였다.

이진트리의 root인 변수 node_pointer root를 전역변수로 선언하였다. 명령 수행 뒤 클라이언트로 보낼 서버의 답변을 저장하기 위한 배열 char printbuf[MAXLINE]을 전역변수로 선언하였다.

Concurrent server 구현을 위해 다음과 같은 함수들을 정의하였다.

void init_pool(int listenfd, pool* p) : pool 구조체를 초기화한다.
void add_client(int connfd, pool* p) : connfd를 pool에 추가한다.
void check_clients(pool* p): 각 ready connfd로부터 명령어를 읽어 처리한다.

main 함수에서 init_pool을 호출하여 pool을 초기화한 뒤, while문 내부에서 select 함수를 호출하여 listening/connected descriptor가 준비될 때까지 기다린다. Listening descriptor가 준비되면, add_client를 호출하여 connfd를 pool에 추가한다. 그 후 check_client를 호출하여 pool에 기록된 각 ready connected descriptor로부터 명령을 받아 처리하고 답변을 전송하도록 하였다.

stock.txt 파일로부터 주식 정보를 읽어 이진 트리로 저장하기 위해 create_node,

insert_node, read_stock 함수를 정의하였다. 변화한 주식 정보를 텍스트 파일에 쓰기 위해 write_inorder, write_stock 함수를 정의하였다. Buy와 sell 명령어를 받았을 때, 이진 트리에서 입력 받은 id에 해당하는 노드를 찾기 위해 tree_search 함수를 정의하였다. 각 명령어를 받았을 때 알맞은 답변을 printbuf에 저장해주는 함수인 show_stock, buy_stock, sell_stock 함수를 정의하였다.

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

Thread-based server를 구현하기 위해 stockserver.c 소스코드를 수정하였다. Shared buffer를 저장할 구조체 sbuf_t를 선언하였다. Connected descriptor의 shared buffer를 의미하는 전역변수 sbuf_t sbuf를 선언하였다. 초기화된 shared buffer를 생성하는 sbuf_init, sbuf에 할당된 메모리를 해제하는 sbuf_deinit, sbuf의 맨 뒤에 원소를 삽입하는 sbuf_insert, sbuf의 맨 앞의 원소를 제거하는 sbuf_remove 함수를 정의하였다.

모든 스레드로부터 받은 바이트 수의 합을 나타내는 전역변수 int byte_cnt와, byte_cnt에 대한 접근을 보호하는 전역변수 sem_t mutex를 모두 static으로 선언하였다.

Reader-writers problem을 해결하기 위해, task 1에서 사용된 구조체와 함수를 일부 수정하여 사용하였다. 먼저 주식 아이템을 나타내는 구조체 item에 int readcnt, sem_t mutex, sem_t w를 추가하였다. 또한, show_stock, buy_stock, sell_stock 함수를 일부 수정하였다. Show를 원하는 reader thread는 노드를 읽기만 하므로, stock item의 세마포어 mutex, w, readcnt를 사용하여 여러 스레드가 동시에 주식 상태를 읽는 것이 가능하지만 읽는 동안 노드를 업데이트 할 수는 없도록 하였다. Buy나 sell을 원하는 writer thread는 노드를 수정하므로, buy_stock과 sell_stock의 함수는 stock item의 세마포어 w를 사용하여 노드를 업데이트하는 동안 다른 스레드의 접근을 막았다. 이를 통해 노드 단위의 fine-grained locking을 구현하였다.

Task 1과 동일하게, stock.txt 파일로부터 주식 정보를 읽어 이진 트리로 저장하기 위해 create_node, insert_node, read_stock 함수를 정의하였다. 변화한 주식 정보를 텍스트 파일에 쓰기 위해 write_inorder, write_stock 함수를 정의하였다. Buy와 sell 명령어를 받았을 때, 이진 트리에서 입력 받은 주식id에 해당하는 노드를 찾기 위해 tree_search 함수를 정의하였다.

main 함수에서 sbuf_init을 호출하여 sbuf를 초기화하고, NTHREADS개의 thread를 생성해 둔다. While loop 안에서 connfd를 sbuf에 삽입하고, 각 worker thread가 이 sbuf에서

connfd를 제거하고 클라이언트를 서비한다.

각 worker thread의 루틴을 함수 thread로 정의하였다. Thread 함수 내에서, sbuf에서 connfd를 제거하고 echo_cnt(connfd)를 호출하여 클라이언트를 서비스 한 뒤 connfd를 닫는다. echo_cnt 함수는 명령어를 읽고 답변을 준비하여 클라이언트로 보내는 기능을 한다.

- **Task3 (Performance Evaluation)**

multiclient.c 소스 코드의 main 함수 시작 부분에 struct timeval start, end와 unsigned long e_usec을 선언한다. While문 시작 전에 gettimeofday(&start, 0)을 호출하여 시작 시간을 저장한다. 메인함수가 종료되기 전에 gettimeofday(&end, 0)을 호출하여 끝나는 시간을 저장한다. e_usec을 출력하여 elapsed time을 구한다.

3. 구현 결과

Event-based와 thread-based 서버가 공통적으로 show, sell, buy, exit의 기능을 가지게 된다. 서버에 접속한 각 클라이언트는 show 명령어를 입력하여 서버로부터 주식 상태 정보를 답변 받아 볼 수 있다. sell [id] [개수] 명령어를 입력하여 주식을 팔고 서버로부터 [sell] success라는 메시지를 받는다. 판 개수만큼 주식 서버에서 해당 주식의 개수가 증가하게 된다. buy [id] [개수] 명령어를 통해 주식을 살 수 있다. 입력한 주식의 개수가 남아있는 주식의 수보다 많으면 서버로부터 Not enough left stock이라는 메시지를 받고, 주식 구매에 성공하면 [buy] success라는 메시지를 받는다. exit을 입력할 경우 서버와의 연결을 종료한다.

4. 성능 평가 결과 (Task 3)

- **확장성: client 개수 변화에 따른 동시처리율 변화**

CLIENT 당 요청 개수(ORDER_PER_CLIENT)는 10으로 설정하였다.

2) Event-based server

Client 개수	Elapsed time	동시처리율(소수점제거)
1	elapsed time: 12759 microseconds	795
10	elapsed time: 106201 microseconds	941
20	elapsed time: 163035 microseconds	1226
50	elapsed time: 582924 microseconds	857
100	elapsed time: 997232 microseconds	1002
200	elapsed time: 1690167 microseconds	1183
300	elapsed time: 3268125 microseconds	917
400	elapsed time: 4391298 microseconds	910
500	elapsed time: 6388621 microseconds	782
1000	elapsed time: 12023338 microseconds	831

Multiclient의 매 실행마다, 클라이언트의 개수가 같더라도 elapsed time에 어느 정도의 오차가 나타났다는 점을 고려할 때, Event-based server는 클라이언트 개수에 따라 유의미한 동시처리율의 차이를 보이지는 않는다.

2) Thread-based server

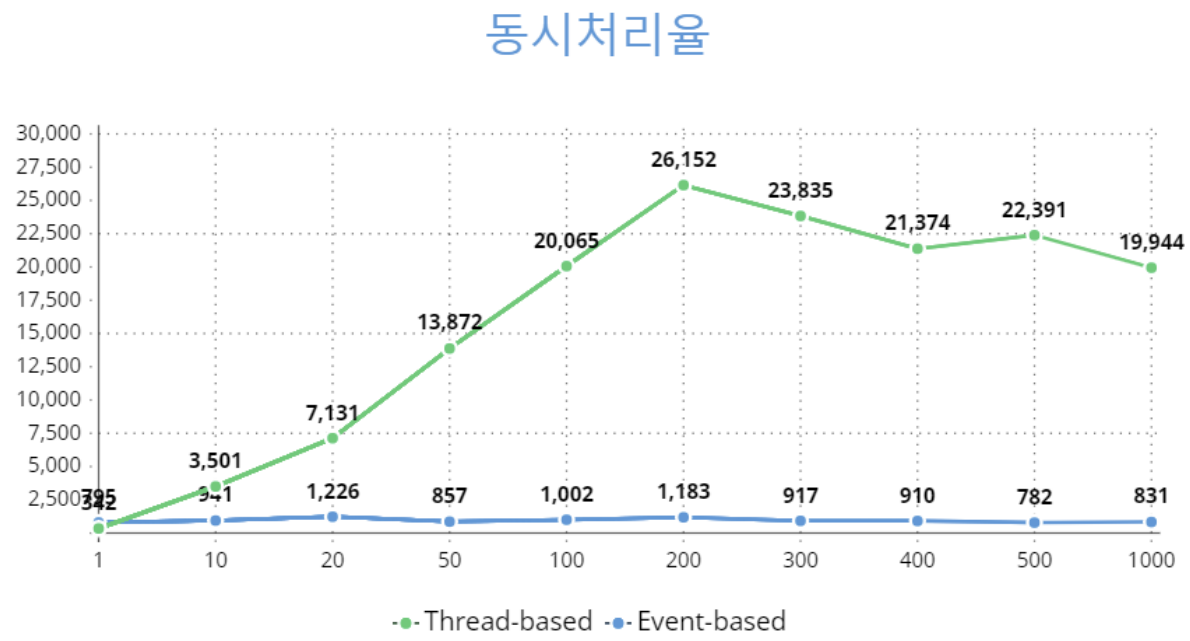
NTHREADS는 10으로 설정하였다.

Client 개수	Elapsed time	동시처리율(소수점제거)
1	elapsed time: 29235 microseconds	342
10	elapsed time: 28557 microseconds	3501
20	elapsed time: 28043 microseconds	7131
50	elapsed time: 36041 microseconds	13873
100	elapsed time: 49837 microseconds	20065
200	elapsed time: 76475 microseconds	26152
300	elapsed time: 125863 microseconds	23835

400	elapsed time: 187139 microseconds	21374
500	elapsed time: 223297 microseconds	22391
1000	elapsed time: 501400 microseconds	19944

Thread-based server는 event-based에 비해 훨씬 높은 동시 처리율을 보였다. 특히, 클라이언트의 개수가 많아질수록 동시처리율도 급격하게 증가하다가, 클라이언트 개수가 200을 넘어가면서부터 약간의 감소를 보였다. 이는 많은 개수의 스레드를 생성하고 처리하는 과정에서 발생하는 thread control overhead 때문으로 추정된다.

Event-based server와 thread-based server의 클라이언트 개수에 따른 동시처리율을 그래프로 나타내면 다음과 같다.



가로축은 클라이언트의 개수, 세로축은 동시처리율을 의미한다. Event-based server의 동시처리율은 거의 일정하게 유지되었다. 반면, thread-based server의 경우 클라이언트 개수가 증가함에 따라 동시처리율도 급격히 증가하다가, 클라이언트의 개수가 매우 많을 때 약간의 감소를 보였다.

또한, 전체적으로 event-based에 비해 thread-based server가 좋은 성능을 가진 것을 확인할 수 있었다. 특히 클라이언트의 개수가 많을 경우 thread-based server가 월등한 성능을 보였다. 이는 thread-based server는 여러 개의 스레드가 각 클라이언트를 서비스하는 반면, event-based server의 경우 하나의 컨트롤 플로우만을 가지고 모든 클라이언트를 서비스하기 때문으로 보인다.

- **워크로드: client의 요청 타입에 따른 동시처리율 변화**

  클라이언트의 개수는 500, 클라이언트 당 요청 수는 10으로 설정하였고, 각 경우에 대해 5번씩 multiclient를 실행하였다.

  1) Event-based server

- Show, buy, sell을 모두 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 4510179 microseconds	1108
elapsed time: 4633315 microseconds	1079
elapsed time: 4762203 microseconds	1049
elapsed time: 5693998 microseconds	878
elapsed time: 5001015 microseconds	999

- Buy, sell만 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 5643119 microseconds	886
elapsed time: 4987391 microseconds	1002
elapsed time: 4913167 microseconds	1017
elapsed time: 4342296 microseconds	1151
elapsed time: 4560052 microseconds	1096

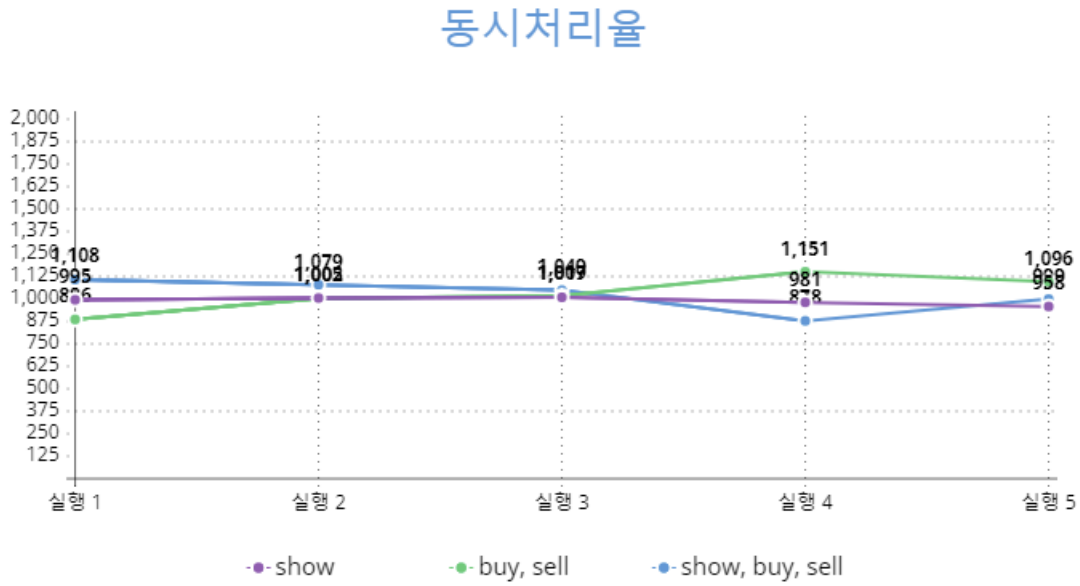
- Show만 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 5021450 microseconds	995
elapsed time: 4968388 microseconds	1006
elapsed time: 4953169 microseconds	1009
elapsed time: 5094032 microseconds	981
elapsed time: 5217314 microseconds	958

Event-based server는, 클라이언트가 show, buy, sell 세 가지 명령어 중 하나를 랜덤하게 실행하는 경우, buy와 sell만 랜덤하게 실행하는 경우, show만 실행하는 경우 모두 유사

한 동시처리율을 가진다. 클라이언트 요청 타입에 따른 유의미한 차이는 나타나지 않았다.

이를 그래프로 나타내면 다음과 같다. Event-based server의 동시처리율은 클라이언트의 요청 타입에 거의 영향을 받지 않고 일정함을 알 수 있다.



2) Thread-based server

클라이언트의 개수는 500, 클라이언트 당 요청 수는 10, NTHREADS는 10으로 설정하였고, 각 경우에 대해 5번씩 multiclient를 실행하였다.

- Show, buy, sell을 모두 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 268006 microseconds	18656
elapsed time: 260337 microseconds	19205
elapsed time: 248821 microseconds	20094
elapsed time: 195240 microseconds	25609
elapsed time: 202920 microseconds	24640

- Buy, sell만 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 143638 microseconds	34809

elapsed time: 168887 microseconds	29605
elapsed time: 193807 microseconds	25798
elapsed time: 172975 microseconds	28905
elapsed time: 196188 microseconds	25485

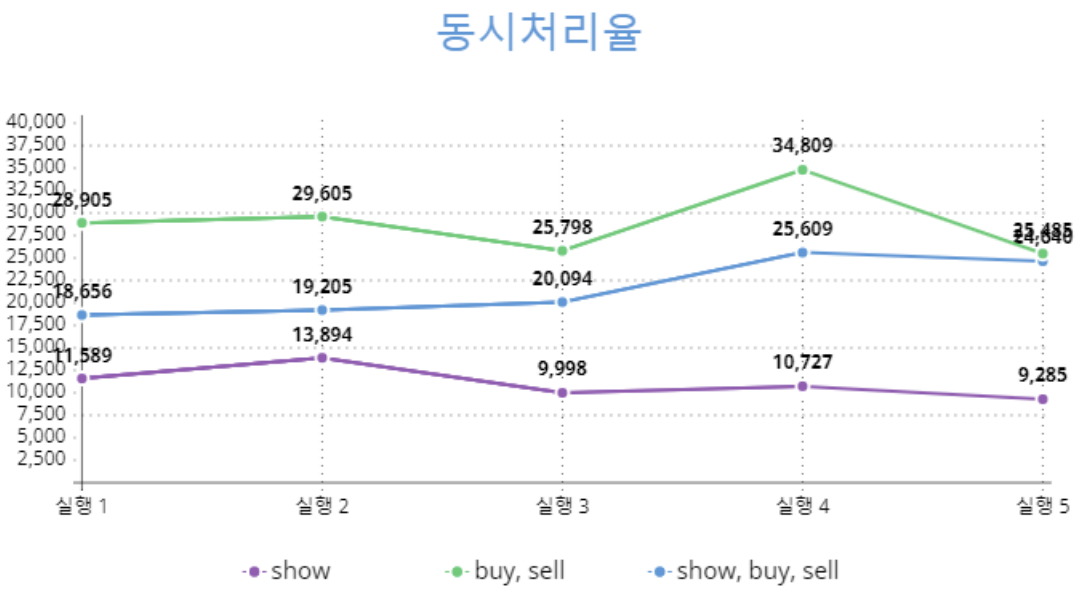
클라이언트가 buy와 sell만 요청으로 보낼 경우, show, buy, sell을 모두 보낼 때보다 실행 속도가 빠르고 동시처리율이 높아졌다.

- Show만 하는 경우

Elapsed time	동시처리율 (소수점 삭제)
elapsed time: 431443 microseconds	11589
elapsed time: 359866 microseconds	13894
elapsed time: 500092 microseconds	9998
elapsed time: 466098 microseconds	10727
elapsed time: 538482 microseconds	9285

클라이언트가 show만 요청으로 보낼 경우, show, buy, sell을 모두 보낼 때보다 실행 속도가 느리고 동시처리율이 낮아졌다..

이를 그래프로 나타내면 다음과 같다.



그래프를 보면 Thread-based server는 클라이언트의 요청 타입에 따라 유의미한 차이를 보임을 알 수 있다. Buy와 sell만 하는 경우에는 세 가지 명령을 모두 섞어서 하는 경우에 비해 동시처리율이 높았고, show만 하는 경우 동시처리율이 낮았다. 이러한 현상은 show가 재귀하며 이진트리의 모든 노드를 방문하는 데 소요되는 시간, 2개의 세마포어 mutex, w를 이용해 locking 처리를 하는 데 소요되는 시간 등으로 인한 것으로 보인다.