Multicore Programming Project 2

담당 교수 : 박성용

이름 : 신서영

학번 : 20200562

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

concurrent한 주식 서버를 만드는 것이 이 과제의 목표이다. 주식 서버는 event-based server와 thread-based server 두 가지 방식으로 구현한다. 해당 주식 서버에서 user는 show, buy, sell, exit의 네 가지 명령어를 사용할 수 있다. show 명령어는 시장의 주식 정보를 보여주고, buy명령어는 user로 하여금 주식을 구매할 수 있도록 한다. sell명령어로 user는 자신이 보유한 주식을 판매할 수 있고, exit을 통해 서버와의 접속을 끊을 수 있다. 이 프로젝트에서는 사용자가 보유한 주식 또는 계좌 정보를 고려하지 않으며, concurrent한 서버 구현에 중점을 두고 개발하였다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   서버는 계속해서 loop를 돌며 Select 함수를 호출하여 새로운 이벤트를 탐지한다. FD_ISSET 함수를 통해 이벤트의 종류를 체크하여 클라이언트의 연결 요청이 있을 경우 add_client함수를 이용하여 클라이언트를 풀에 추가하고, 입력 요청의 경우 check_client함수를 통해 처리한다.

2. Task 2: Thread-based Approach

   thread-based approach는 다중 스레드를 이용한다는 점에서 event-driven approach와 큰 차이가 있다. event-based server는 서버가 concurrent하게 돌아가는 것처럼 보이지만 실제로는 하나의 프로세스가 커널의 도움을 받아 이벤트를 비동기적으로 처리한다. 하지만 thread-base server에서는 여러 개의 스레드가 각각의 요청을 처리한다. 본 프로그램에서는 main thread가 master thread의 역할을 하여 worker 스레드를 create하고 connfd를 sbuf에 할당하는 방식으로 진행된다.

3. Task 3: Performance Evaluation

gettimeofday 함수를 활용하여 event-based server와 thread-based server의 동시 처리율을 측정한다. 단일 프로세스에서 실행되는 event-based server보다는 다중 스레드를 이용하는 thread-based server의 동시처리율이 높을 것으로 예상된다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

I/O Multiplexing은 단일 스레드 또는 프로세스가 여러 개의 I/O작업을 동시에 처리하는 매커니즘을 의미한다. 서버에 여러 명의 클라이언트가 접속해 input을 입력하면 서버는 그에 따른 output을 동시에 출력할 수 있다. 한 번에 한 명의 클라이언트의 요청을 파악하기 위해 다른 클라이언트들의 요청을 blocking하게 되는 문제점을 극복할 수 있다.

✓ epoll과의 차이점 서술

epoll은 select의 단점을 보완한 함수이다. select의 경우 이벤트를 탐지하기 위해 ready_set을 loop를 돌며 확인한다. 따라서 descriptor가 많아질수록 확인해야 하는 descriptor 수가 많아지기 때문에 오버헤드가 증가하는 단점이 있다. epoll은 운영체제의 시스템 콜을 사용하기 때문에, 루프를 돌며 이벤트를 탐지할 필요 없이 효율적으로 이벤트를 감지할 수 있다는 장점이 있다.

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

✓ Master Thread의 Connection 관리

먼저 스레드 전체가 공유하는 변수로 sbuf를 선언한다. 마스터 스레드는 worker 스레드를 생성하고, client와의 connection이 연결될 때마다 sbuf에 connfd를 추가한다. worker 스레드는 sbuf에서 connfd를 가져와 해당 client의 입력 요청을 처리한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

worker 스레드는 sbuf에서 connfd를 가져와 클라이언트의 입력 요청을 처리한다. 클라이언트가 연결을 해제하면 worker 스레드는 sbuf에서 다시 새로운 connfd를 가져와 새로운 클라이언트의 request를 처리한다.

- **Task3 (Performance Evaluation)**

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

event-based server와 thread-based server의 동시처리율을 측정하고자 한다. 동시처리율은 완료된 작업 수 / 소요된 시간으로, 클라이언트 수 * 클라이언트당 요청 개수 / 소요된 시간으로 계산한다.

확장성에 중점을 둔 분석으로는 각 서버에 대해 client개수에 변화에 따른 변화를 분석할 예정이다.

워크로드에 따른 분석으로는 client가 show를 요청하는 경우와 buy, sell을 요청하는 경우의 동시 처리율 또한 분석하고자 한다.

✓ Configuration 변화에 따른 예상 결과 서술

event-based server 보다 thread-based server 에서 동시처리율이 높을 것으로 예상된다. 클라이언트가 많아질수록 thread-based server 의 성능이 좋아질 것이다.

client 가 show 를 요청하는 경우와 buy, sell 을 요청하는 경우의 차이 또한 유의미할 것이라 예상된다. show 명령의 경우 buy 와 sell 에서 write 를 수행하는 것과는 달리 read 만을 수행한다. 따라서 나머지 두 명령어보다 수행속도 면에서 우위에 있을 것으로 짐작된다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

먼저, event-based server와 thread-based server에서 공통적으로 구현한 내용부터 서술하겠다. 사용되는 AVL 트리를 구현하였다. 효율적인 탐색을 위해 이진 트리 중에서도 AVL트리를 구현하였으며, 트리의 각 노드에 주식 정보를 담았다. event-

based server에서는 오버헤드를 최소화하기 위해 is+all_disconnected 함수로 client와의 모든 커넥션이 끊겼는지 확인한 후 stock.txt를 업데이트하였고, thread-based server에서는 한 client와의 커넥션이 끊길 때마다 stock.txt를 업데이트하였다. 또한, 서버에서 Ctrl+C가 감지되었을 때 signal handler를 작동시켜서 stock.txt가 업데이트되도록 하였다.

event-based server의 경우 check_client 함수를, thread-based server의 경우 echo_cnt함수를 수정하여 show, buy, sell, exit 명령어에 대해 처리 부분을 추가하였다. show함수의 경우 client의 요청에 대해 한 번의 응답을 전송해야 했기 때문에 showbuf 변수에 여러 줄의 주식 정보를 이어붙여 담았다. buy함수에서는 left_stock 개수를 확인한 후 left_stock 개수가 요청받은 주식 개수보다 크거나 같은 경우에만 buy가 success되도록 처리했다. sell함수는 별다른 조건 없이 success되도록 처리했다.

thread_based server의 경우 공유 변수에 대한 처리를 각별히 신경써야 했다. 따라서 node 구조체에 readcnt, mutex_readcnt, mutex_write 변수를 추가하였다. 또한 echo_cnt함수에서 show, buy, sell, exit 명령어를 구현할 때 뮤텍스를 추가적으로 걸어주었다. read-writer 문제의 해결방안을 코드에 적용하여 reader가 read하고 있는 도중에는 다른 reader들도 read할 수 있도록 했고, writer의 접근을 막았다. 이로써 클라이언트들이 동시에 show 명령을 내렸을 때의 오버헤드가 조금 줄었을 것이라 예상된다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

본래의 개발 목표를 성공적으로 달성하였다.

show, buy, sell, exit 명령어가 잘 작동하였으며, thread-based server의 경우 뮤텍스가 변수의 값을 잘 protect하고 있음을 확인하였다.

thread-based server에서 클라이언트와의 커넥션이 끊길 때마다 stock.txt를 업데이트해주었는데, 시간상의 이유로 모든 클라이언트와의 커넥션이 끊길 때 stock.txt를 업데이트하는 방식으로 코드를 수정하지 못했다. 클라이언트와의 커넥

션을 추적하는 공유 변수를 둔다면 모든 클라이언트와의 커넥션이 끊길 때 한꺼번에 stock.txt를 업데이트하도록 구현할 수 있을 것 같다.

또한 echo_cnt의 함수 이름 또한 해당 함수의 기능에 어울리게 수정해주면 좋을 것 같다.

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**

event			
Client 수	**Random**	**Show**	**Buy/sell**
1	10003032	10026806	10019871
20	10020364	10026111	10021080
50	10023024	10034182	10026704
100	10069832	10070362	10057256

thread			
Client 수	**Random**	**Show**	**Buy/sell**
1	10017197	10018821	10018057
20	10034762	10033185	10019226
50	20516511	20669039	20284244
100	40651102	41827233	41416318

위의 표는 각각 event-based server와 thread-based server의 elapsed time을 ms 단위로 측정한 결과를 담은 표이다. 이를 바탕으로 계산한 동시처리율은 다음과 같다.

event			
Client 수	**Random**	**Show**	**Buy/sell**
1	1	1	1
20	**20**	20	20
50	50	50	50
100	100	100	100

thread			
Client 수	**Random**	**Show**	**Buy/sell**
1	1	1	1
20	20	20	**20**
50	25	25	25
100	24.6	23.9	24.1

마지막 줄은 소수점 첫째 자리까지 연산한 결과이다.

event-based server의 동시처리율이 thread-based server의 동시처리율보다 높게 나타났다. 이는 예상과 다른 결과였다. 아마도 thread-based server에서 thread를 create하고 함수를 호출하는 과정에서 오버헤드가 큰 것으로 추측된다. 또한 이는 NTHREAD를 30으로 고정하고 측정한 결과이며, thread 개수를 줄이거나 늘리면 다른 결과가 나타날 것이라 추측된다.

client 수를 늘린 것에 비하여 걸린 시간은 크게 차이나지 않았다는 점이 주목할 만하다. 이는 특히 event-based server에서 시간차가 적게 나타났다. thread-based server에서는 event에 비해 시간차가 크게 나타났다.

show와 buy/sell(random) 간 동시처리율 차이는 크지 않았다. 앞선 elapsed time 측정표를 보면 show 명령을 수행하는 데 걸리는 시간이 buy/sell 명령을 수행하는 데 걸리는 시간보다 조금 더 큰 것을 확인할 수 있다. 하지만 이는 생각보다 크지 않은 차이였다. 동시처리율 또한 거의 비슷했다.