System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 윤영인

학번 : 20192135

1. 개발 목표

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

이진 트리를 이용한 event-driven approach와 thread-based approach 방식의 여러 client의 동시 접속 및 서비스를 위한 주식 서버를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

각 client는 fd를 trigger하여 각 클라이언트가 서버로 요청을 보내면 pool을 통해 client connection을 해준다. stock.txt 안의 주식 정보를 이진트리를 생성하여 넣고 check_clients 함수를 통해 클라이언트가 보낸 명령인 sell, buy, show에 따라 알맞은 동작을 수행한다.

2. Task 2: Thread-based Approach

thread를 미리 생성하고 connfd에 클라이언트로부터 받아온 명령을 통해 주식 정보를 넣은 이진트리를 생성하여 클라이언트에게 알맞은 응답을 한다.

3. Task 3: Performance Evaluation

buy/sell, show, random 3가지의 경우로 나눠서 클라이언트 10, 20, 100개마다 5번씩 실험하면서 event-driven과 thread-based의 동시처리율을 계산하였다.

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

✓ epoll과의 차이점 서술

event-based approach는 프로세스 1개와 logical flow 1개를 가진다. I/O multiplexing은 concurrent하지 않고 1개의 프로세스가 여러 이벤트를 빠르게 처리한다. Select를 사용하여 descriptor가 pending input을 가지는지 확인하고, 커널에게 프로세스를 suspend하도록 요청한다. 하나 이상의 I/O 이벤트가 발생하면 제어권을 리턴하도록 한다. 이후 listenfd가 connect 요청을 accept하여 connfd를 생성하고, connfd는 클라이언트와 connect하여 채널을 생성하여 클라이언트가 보낸 메시지를 읽어서 echo한다. listenfd가 pending input을 가지면 connection을 accept하고 새로운 connfd를 connfd 배열에 추가한다. 이후 pending input이 있는 모든 connfd를 처리하고 이 과정을 계속 반복하도록 한다. check_clients 함수를 통해 클라이언트가 보낸 명령에 대해 적절한 기능을 수행하도록 한다.

epoll은 커널에 관찰 대상의 정보를 한 번만 전달하고 변경이 있을 시에만 변경사항을 알려준다. 하지만 select 함수는 FD_SET을 사용하기 때문에 많은 fd를 한 번에 관찰이 가능하며, fd의 상태를 하나의 비트로 표현한다. 따라서 fd의 번호를 인덱스로 하여 준비 상황을 파악할 수 있다. 하지만 epoll과는 다르게 모든 fd를 순회하면서 FD_ISSET으로 fd를 확인해야 하는 비효율이 생긴다.

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

✓ Master Thread의 Connection 관리

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Thread-based approach는 프로세스 하나에 여러 execution flow를 가져 concurrent하게 프로그램을 수행할 수 있다. 메인함수에서 먼저 sem_init, sbuf_init 함수로 mutex와 sbuf를 초기화해준다. 이후 지정한 스레드의 개수만큼 스레드를 생성하고, while문 안에서 client와 연결이 될 때마다 sbuf_insert 함수로 client의 connfd를 sbuf에 넣어준다. Pthread_create 함수에서 인자로 받는 thread 함수는 echo 함수를 호출하여 클라이언트가 보내는 명령에 따라 적절한 수행을 하도록 한다. 클라이언트와의 연결이 종료되면 stock.txt의 주식의 정보가 업데이트 된다.

- **Task3 (Performance Evaluation)**

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

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트가 보내는 명령에 따른 동시 처리율(시간 당 클라이언트 처리 요청 개수)의 변화를 분석하기 위해 명령어가 buy 또는 sell일 경우, show만 있을 경우, 모든 명령어가 랜덤으로 처리될 경우로 3가지로 나눈다. 이때 클라이언트를 10, 20, 100개로 증가시키면서 각각 5번씩 시간을 측정한다. 우선 gettimeofday 함수를 사용하여 3가지 경우에 대해 5번씩 elapse time을 측정한 후 평균을 구한다. 이후 시간 당 클라이언트 개수를 계산하여 동시 처리율을 구한다. 동시 처리율은 1초 동안 얼마나 많은 요청을 처리하는지에 대한 계산으로 task1은 하나의 프로세스가 여러 이벤트를 처리하는 것이기 때문에 task2에 비해 동시 처리율이 낮을 것이다.

명령어가 buy/sell일 경우, 이진트리의 해당 노드를 찾아 수행하는 것이므로 수행 시간이 가장 적고 show는 트리를 모두 방문해야 하므로 수행시간이 보다 클 것이다. 명령어가 랜덤인 경우는 이 모든 명령어를 처리하므로 당연히 수행 시간이 가장 클 것이다. event-based approach의 경우는 concurrent하게 처리하지 않기 때문에 thread-based approach에 비해 클라이언트의 개수가 늘어날수록 동시 처리율은 낮을 것이다.

C. 개발 방법

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

1. 이진 트리 구조체

주식 정보에서 id를 기준으로 이진 트리를 생성한다. 이진 트리 안의 각 노드들은 주식의 id, 가격, 남은 주식의 개수의 정보가 들어있으며 *leftChild (왼쪽 자식을 가리키는 포인터), *rightChild (오른쪽 자식을 가리키는 포인터)를 가진다. task2의 Stock 구조체에는 읽기를 구현할 때 쓰이는 mutex와 쓰기를 구현할 때 쓰이는 mutex인 w를 추가적으로 넣어준다.

2. Show

readnb로 읽기를 수행하기 위해 이진 트리를 전위 순회하면서 buf에 각 주식의

정보를 넣어준다. 방문이 끝나면 주식의 정보가 모두 담긴 buf를 Rio_writen 함수로 클라이언트에게 전송한다.

3. Semophore의 Readers-Writers problem

buy, sell 함수를 수행할 때, P함수를 이용하여 쓰기 w 뮤텍스의 lock을 걸어주고, buy 일 경우 산 주식의 개수 만큼 빼고, sell일 경우 산 주식의 개수만큼 더해준다. 수행이 끝나면 V 함수를 이용하여 lock을 해제한다.

show 함수를 수행할 때는 이진트리를 순회하는 visitTree 함수에서 buf에 주식의 정보를 담아준 후 읽기 mutex를 lock을 걸어주고, 현재 노드의 read_cnt를 하나 증가시킨다. 만약 올바르게 증가하여 read_cnt가 1일 경우, 다시 현재 노드의 쓰기 w 뮤텍스를 lock해준다. 다시 읽기 mutex의 lock을 풀어주고 전위순회이므로 왼쪽 자식 노드를 탐색하도록 한다. 탐색이 끝나면 현재 노드의 읽기 mutex의 lock을 다시 걸어주고 현재 노드의 read_cnt를 하나 감소시킨다. 만약 올바르게 감소시켜 read_cnt가 0일 경우, 아까 lock을 걸어주었던 쓰기 w 뮤텍스의 lock을 해제하고, 읽기 mutex도 lock을 해제한다. 그리고 오른쪽 자식노드를 탐색한다.

3. 구현 결과

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

(1) Event-based approach

클라이언트와 connect가 되면 add_client에 현재 connfd를 추가한다. check_clients 함수를 통해 클라이언트가 보내는 명령어를 readlineb로 읽어와서 각 명령에 알맞은 함수를 수행하도록 한다. buy, sell이 올바르게 수행되었을 경우 buf에 success 문자열을 넣고 클라이언트로 전송한다. 클라이언트는 readnb로 서버의 메시지를 읽어서 출력한다. buy와 sell이 올바르게 수행되어 success를 출력할 경우, 화면에 초록색으로 표시되도록 구현하였다.

(2) Thread-based approach

클라이언트와 connect가 되면 sbuf에 connfd를 넣고, Pthread_create함수에서 스레드를 생성하면서 인자로 넣은 thread 함수가 수행된다. thread 함수는 echo함수를 통해 클라이언트가 보낸 명령어에 맞는 함수를 수행한다. 하나의 프로세스가 여러 이벤트를 수

행하는 event-based approach와는 다르게 show, sell, buy 뮤텍스를 P와 V 함수를 이용하여 concurrent한 작업을 수행하도록 구현하였다. 이때 show와 updateTxt는 읽기, buy와 sell 함수는 쓰기에 해당한다.

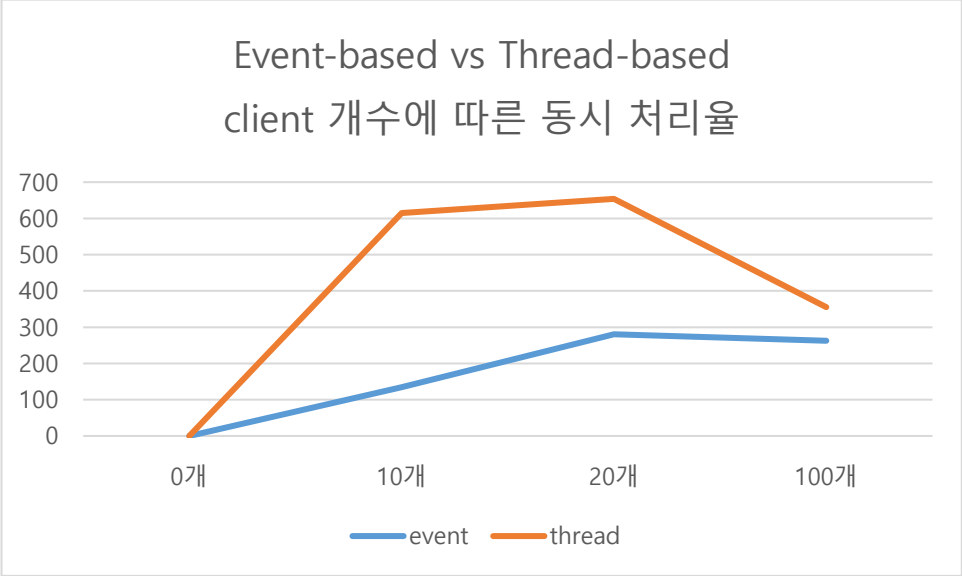
4. 성능 평가 결과 (Task 3)

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

event	10개	1	2	3	4	5	평균	
1. buy/sell		0.040181	0.034448	0.030949	0.032555	0.49975	0.127577	
2. show		0.077794	0.073297	0.044042	0.046968	0.05465	0.05935	
3. buy, show, sell		0.032087	0.047368	0.032899	0.033439	0.034005	0.03596	0.074295
event	20개	1	2	3	4	5	평균	
1. buy/sell		0.06875	0.077367	0.081513	0.071603	0.070207	0.073888	
2. show		0.067945	0.066651	0.061649	0.085328	0.063521	0.069019	
3. buy, show, sell		0.064268	0.057373	0.085764	0.068138	0.079516	0.071012	0.071306
event	100개	1	2	3	4	5	평균	
1. buy/sell		0.313926	0.378606	0.356856	0.32912	0.383019	0.352305	
2. show		0.356497	0.387395	0.356252	0.326828	0.332469	0.351888	
3. buy, show, sell		0.349209	0.320356	0.38007	0.464391	0.392368	0.381279	0.381279
thread	10개	1	2	3	4	5	평균	
1. buy/sell		0.01566	0.018169	0.019944	0.016407	0.015951	0.017226	
2. show		0.017779	0.016719	0.015144	0.013847	0.015713	0.01584	
3. buy, show, sell		0.016769	0.016834	0.015392	0.015388	0.014072	0.015691	0.016253
thread	20개	1	2	3	4	5	평균	
1. buy/sell		0.026185	0.028825	0.02659	0.025738	0.025595	0.026587	
2. show		0.034699	0.039586	0.030455	0.031711	0.026445	0.032579	
3. buy, show, sell		0.032039	0.021717	0.028252	0.028616	0.052281	0.032581	0.030582
thread	100개	1	2	3	4	5	평균	
1. buy/sell		0.112426	0.159244	0.136854	0.123448	0.156759	0.137746	
2. show		0.472548	0.46017	0.425528	0.632651	0.596158	0.517411	
3. buy, show, sell		0.164505	0.221984	0.231378	0.12769	0.203353	0.189782	0.281646

위의 사진은 3가지 경우 (1. buy/sell, 2. show, 3. buy, show, sell) 로 나눠 클라이언트의 개수에 따라 5번씩 수행시간을 측정한 결과이다. 클라이언트의 개수를 각 평균값으로 나눠 동시 처리율을 계산하였다.

(1) 확장성

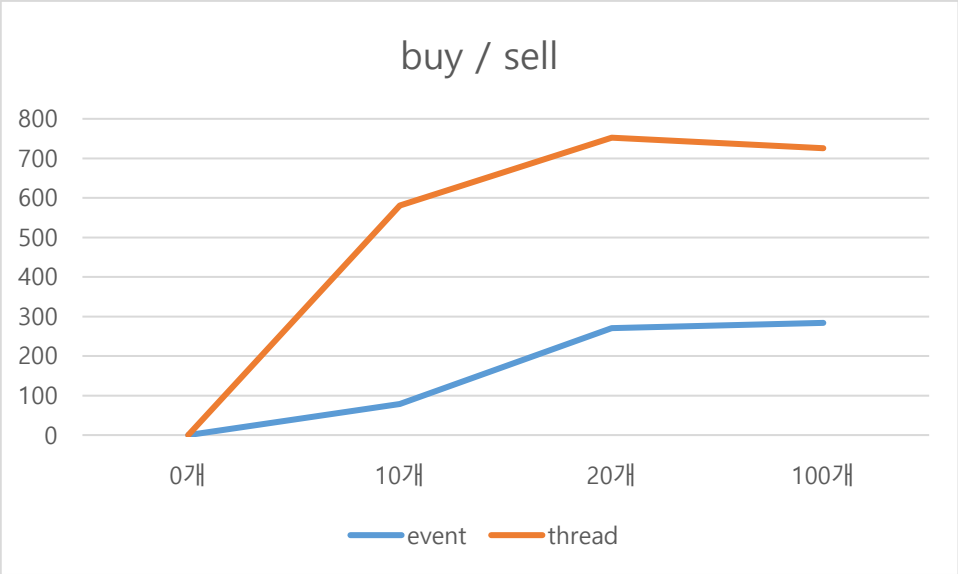


클라이언트 개수에 따른 3가지 경우의 수의 수행시간의 평균을 구하여 동시 처리율을 계산하였다. 위에서 예상한 것과 마찬가지고 thread가 동시 처리율이 더 높았다.

event-based의 경우 overhead가 매우 적지만, thread-based의 경우 상대적으로 overhead를 많이 가져 데이터 침범을 막기 위해 semaphore를 사용한다는 점에서 전자에 비해 비용이 많이 든다. 또한 thread-based approach에서 각 노드마다 semaphore를 사용하여 critical section을 작게 만들어 동시처리율을 높게 만들었다.

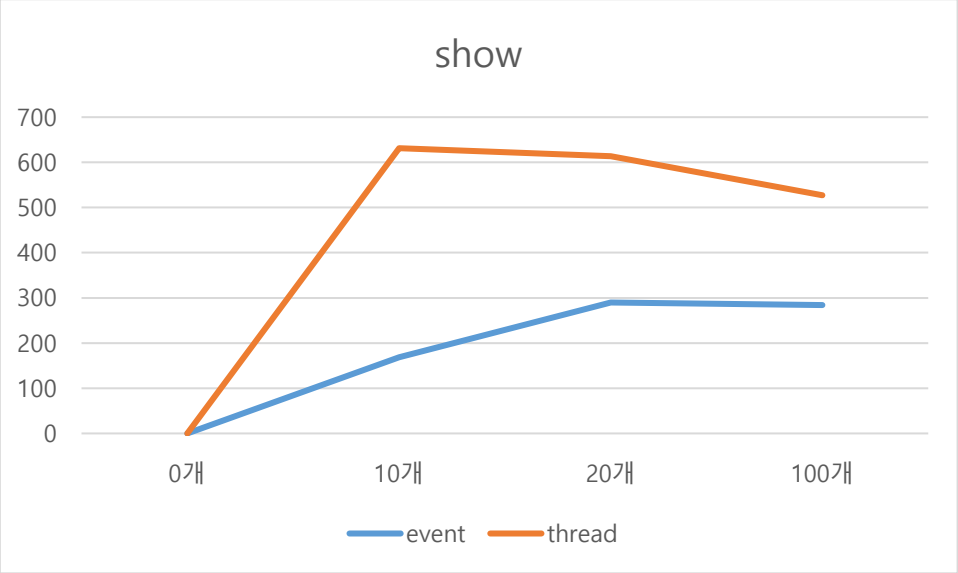
(2) 워크로드에 따른 분석

① buy/sell



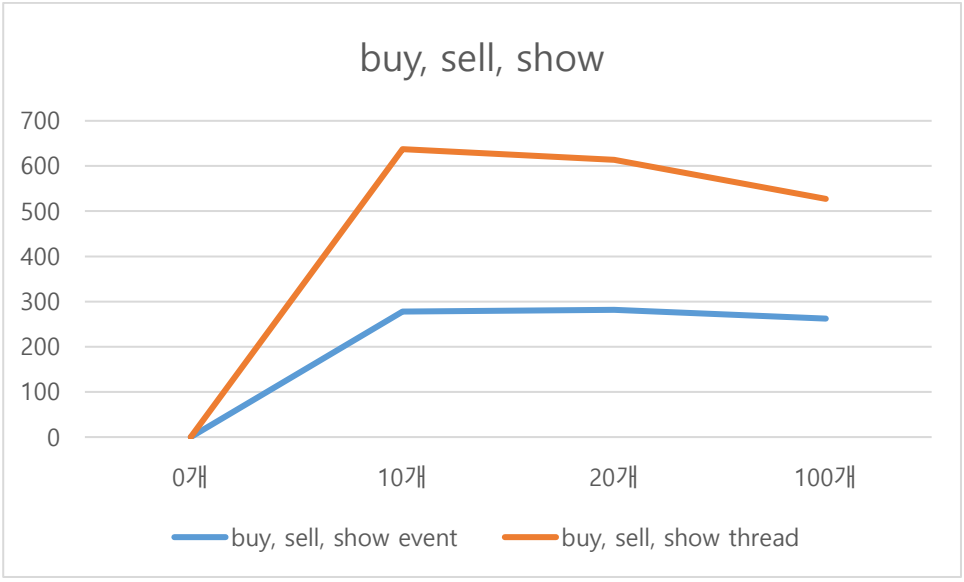
buy와 sell은 thread일 때 뮤텍스를 사용하여 concurrent하게 작업을 수행하기 때문에 event에 비해 동시 처리율이 높음을 확인할 수 있었다. 특정한 주식 id에 대해 작업을 수행하므로 show에 비해 동시 처리율이 낮다.

② show



show에서 thread는 트리를 순회하면서 왼쪽 자식으로 이동하거나 오른쪽 자식으로 이동할 때마다 뮤텍스 lock을 걸어줘야 했다. 위의 그래프를 보면 event의 경우, 클라이언트 개수에 따라 동시 처리율이 점점 증가하지만 thread는 클라이언트의 개수가 증가하면서 동시 처리율이 살짝 감소함을 보인다. show는 모든 노드를 순회하기 때문에 동시 처리율이 다른 명령에 비해 높다.

③ show, buy, sell



명령어를 random하게 수행했을 경우, 이 역시 thread가 event에 비해 전체적으로 동시 처리율이 높음을 알 수 있다. 그래프의 모양이 show와 비슷한 것으로 보아 show가 전체 작업에 영향을 주었다는 것을 알 수 있다.