System Programming Project 2

담당 교수 : 김 영 재

이름 : 이 호 성

학번 : 20171680

1. 개발 목표

단일 프로세스/쓰레드 기반 stockserver는 여러 client와 connection이 불가능하다. 따라서, 각각 client의 connection부터 요청된 request가 모두 직렬화 되어 순서대로 처리된다. 이번 프로젝트는 concurrent 프로그래밍을 통해 동시 주식 서버를 두 가지 방식을 이용하여 설계 및 구현하는 것이 목표이다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   여러 개의 connection fd를 생성하여 client의 요청이 들어오는 fd가 있다면 처리를 해주는 방식이다.

2. Task 2: Thread-based Approach

   여러 개의 thread를 생성하여 Master Thread는 client와의 연결을 담당하고 Worker Thread가 요청의 처리를 담당하게 된다.

3. Task 3: Performance Evaluation

   Event-driven approach는 하나의 logical control flow를 이용하면서도 concurrent 한 프로그래밍을 구현한다. 하나의 logical control flow 를 이용하기 때문에 process/thread control overhead가 매우 적게 나올 것으로 기대된다. 하지만 결국 하나의 logical control flow를 이용하기 때문에 fine grained concurrency를 구현하기는 쉽지 않을 것이며 multi-core의 이점을 살리지 못하게 된다. 이에 반해 Thread-based approach는 여러 개의 logical control flow (thread)를 사용하기 때문에 event-driven approach 보다 finely-grained concurrency를 제공하며 multi-core의 장점도 활용할 수 있을 것으로 기대된다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**

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

  ✓ Event based server 에서 I/O Multiplexing은 하나의 listenfd 와 여러 개의 clientfd을 통해 이루어지며 multiclient 의 연결 요청은 하나의 listenfd를 통해, 작업 처리 요청은 각각의 clientfd를 통해 처리하게 된다. 여러 개의 작업 요청은 select 혹은 epoll 함수로 처리할 수 있는데 이번 프로젝트에서는 select 함수를 사용한다

  ✓ epoll과의 차이점 서술

  select의 경우 pool의 모든 fd를 순회하며 FD_ISSET으로 체크하게 되는데 이는 실제로 pending input 이 있는 fd 외에 다른 fd들도 확인하게 되어 비효율적이다. epoll의 경우 select의 단점을 보완하여 만든 I/O 통지 기법으로 모든 fd를 반복문으로 확인하지 않고 운영체제가 fd를 직접 관리해주어 요청에 따른 epoll_fd 를 반환하여 이를 통해 요청을 처리하게 된다. 하지만 해당 프로젝트에서는 concurrent server 구현이 주된 목적이기 때문에 조금 더 간단한 select 함수를 사용한다.

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

  ✓ Master Thread의 Connection 관리 , Worker Thread Pool

  thread based server 의 master thread의 경우 listenfd 로 들어오는 연결 요청을 받고 승인을 한다. 연결이 완료 된다면 pthread_create() 함수를 통해 peer thread 를 생성하고 해당 thread 가 client 가 종료 할 때까지 요청을 처리하게 된다. detached mode 로 peer thread 를 설정하여 master thread 가 peer thread 의 종료에 대한 처리를 별도로 해줄 필요가 없게 만든다. stock.txt 의 원활한 업데이트를 위해 worker thread pool의 개수를 저장해 두어 client 가 연결 되어있지 않다면 stock table을 stock.txt 에 최신화 한다.

- **Task3 (Performance Evaluation)**

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

  이번 프로젝트에서 두 서버간 성능 평가 및 분석은 동시 처리율을 기준으로 평가한다. 동시 처리율은 기준 시간당 client 처리 요청의 개수로 (client * client 당 요청) / (요청을 처리하는데 걸린 시간)의 식을 통해 구할 예정이다.

  show 와 buy/sell 요청간의 성능 차이는 client * client 당 요청을 동일하게

설정 한 뒤 요청을 처리하는데 걸린 시간으로 성능평가를 할 예정이다.

✓ Configuration 변화에 따른 예상 결과 서술

event based server 의 경우 thread-based server 에 비해 overhead가 적어 client의 수가 적을 경우 thread-based server 에 비해 좋은 성능을 가질 것으로 예상되지만 client 의 개수가 늘어남에 따라 fine-grained concurrency를 제공하는 thread based server 에 비해 효율의 감소가 더 클 것으로 예상된다.

또한 show 의 경우 모든 노드를 한번씩 방문해야 하므로 buy/sell 요청보다 시간이 더 걸릴 것으로 예상된다.

C. 개발 방법

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

두가지 서버 모두 show buy sell 의 요청이 들어왔을 때 자료구조를 처리하는 코드는 동일하게 작성하였다.

stockserver 프로그램이 실행 되면 stock.txt 의 정보를 각각의 stock의 정보를 읽어 구조체 배열 struct item {...}items[] 을 통해 tree의 형태로 stock의 정보를 저장하였다.

Event-based server

event-based server는 pool을 통해 clientfd 의 요청을 처리한다. select 함수를 통해 pending input이 있는 fd들을 선정 한 다음 listenfd의 경우 client와 연결을 하고 add_client를 통해 pool에 해당 fd를 추가한다. connectfd 의 경우 check_clients() 를 통해 pending input들을 처리하게 된다. check_clients()함수는 show buy sell 의 요청을 처리하게 된다. show 의 경우 DFS 를 통해 tree 자료구조를 순회하며 buffer에 stock 정보들을 추가한다. buy와 sell의 경우 find_tree_idx 를 통해 요청이 들어온 stock의 index를 찾아 개수를 조정한 후 성공했다는 메세지를 buffer 에 저장한다. 이때 buy의 경우 stock의 수량이 부족하다면 성공했다는 메시지 대신 부족하다는 메시지를 남긴다. persistency를 위해 stock.txt 를 update 해야 하는데 현재 연결되어 있는 connection의 개수를 저장 해 이 값이 0이 된다면 stock.txt 를 최신화 하도록 하였다.

Thread-based server

Event based server의 경우 concurrent하게 처리가 되지만 logical flow 가 하나기 때문에 shared variable과 같은 문제를 고려 할 필요가 없다. 하지만 여러 logical flow가 존재하는 thread-based server의 경우 race condition이 발생 할 수 있기 때문에 stock을 관리하는 자료구조에 추가로 semaphore을 추가해야 한다. thread-based server는 client 연결 요청이 들어온다면 pthread_create()을 통해 peer thread를 생성한다. peer thread가 할 일이 정의 된 thread() 함수는 receive_request() 함수를 실행하는데 receive_request()가 하는 일은 위의 check_clients() 와 같은 방식으로 show, buy, sell 요청을 처리하는데 주식 정보가 shared variable이기 때문에 race condition을 방지하기 위해 semaphore를 추가해야 한다. 이때 buy 와 sell 의 경우 자료의 내용을 바꾸지만 show의 경우 내용을 바꾸지 않고 확인만 하기 때문에 두가지 reading/writing semaphore(mutex, r_lock)을 두어 fine-grained locking을 구현한다. persistency를 위해 stock.txt 를 update 하는 부분의 경우 thread의 개수를 저장하고 있으려면 전역 변수로 가지고 있어야 하나, 전역 변수의 경우 shared variable로 race condition이 일어날 수 있어 thread_cnt 를 관리하는 thread_lock을 두어 thread 생성 시 thread_cnt를 증가 시키고 종료 시 thread_cnt를 감소시키며 thread 의 개수를 관리하며 이 값이 0이 되었을 때 lock을 잡고 stock.txt 를 최신화 하여 최신화 도중 새로운 요청이 들어와 thread를 생성하는 경우 thread_cnt를 증가시키기 전 P(&thread_lock)에 의해 잠시 대기하도록 한다.

3. 구현 결과

프로젝트 구현 결과, 프로젝트의 모든 요구사항을 구현하였다. Event-based server와 Thread-based server 모두 concurrent하게 여러 client의 요청을 처리할 수 있음을 실행을 통하여 확인할 수 있다.

4. 성능 평가 결과 (Task 3)

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

기본 값 : ORDER_PER_CLIENT : 10 , STOCK_NUM : 10 , BUY_SELL_MAX 10

측정 시작 : multiclient.c중 fork for each client process while 문 이전

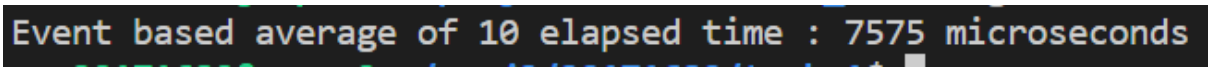
측정 종료 : 모든 client 가 종료되어 waitpid() 반복문이 종료된 후

측정 1 : Client 가 show, buy, sell 을 섞어서 요청하는 경우 Client 수 변화에 따른 동시 처리율의 변화

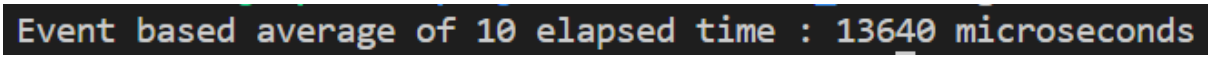
측정 및 평가 방식 : 10번의 반복을 통해 얻은 소요 시간의 평균을 (ORDER_PER_CLIENT*CLIENT_NUM / elapsed_time(ms)) 으로 동시처리율을 계산하였다.

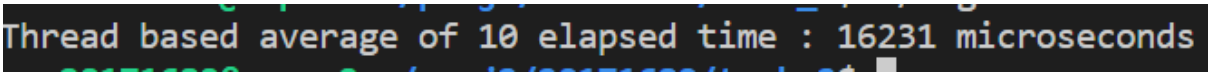
CLIENT_NUM : 4

CLIENT_NUM : 8

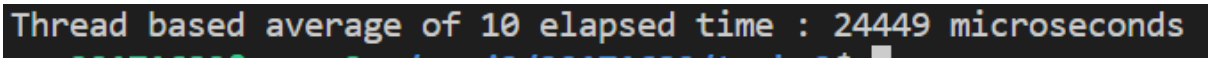


CLIENT_NUM : 16

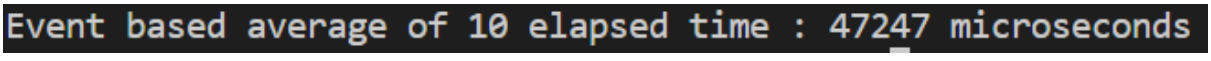


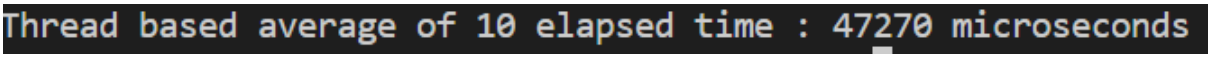


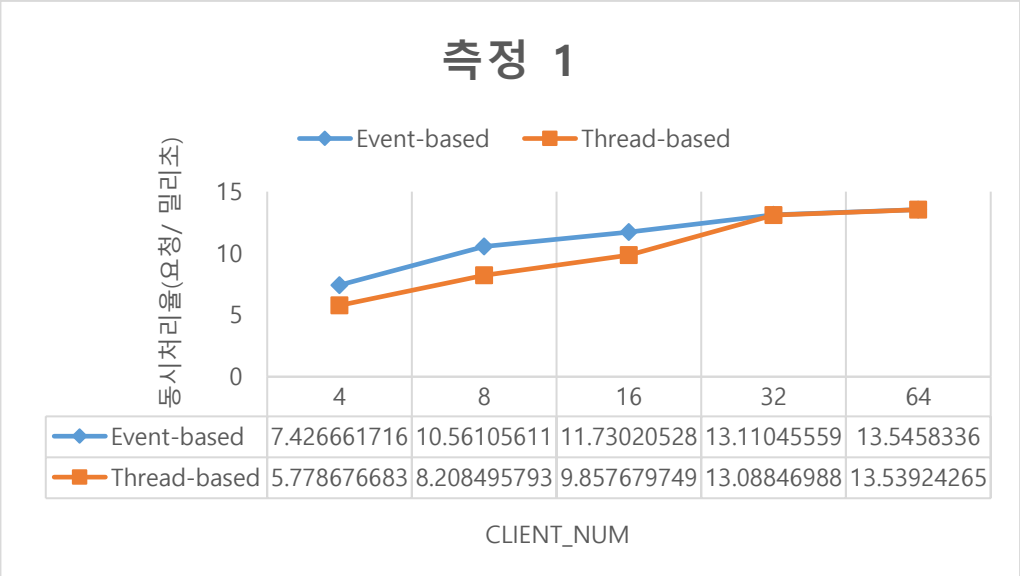
CLIENT_NUM : 32



CLIENT_NUM : 64







분석 1

Event based server 의 경우 process/thread 의 overhead나 lock으로 인한 overhead 가 없기 때문에 모든 경우 Thread based server 보다 동시 처리율이 좋다는 것을 확인할 수 있었다. 하지만 client 의 개수가 증가할수록 event based server의 경우 상대적으로 fine-grained concurrency를 제공하지 못하기 때문에 thread based server와의 차이가 감소함을 확인 할 수 있었다.

측정 2 : Client 가 show 만을 섞어서 요청하는 경우와 buy sell만을 요청하는 경우와의 비교

측정 및 평가 방식 : 10번의 반복을 통해 얻은 소요 시간의 평균을 기준으로 평가하였다. (ORDER_PER_CLIENT*CLIENT_NUM 이 10*16 으로 동일한 상황에서 요청의 종류에 따른 변화를 측정하기 위해 소요시간만을 확인하였다)

Event based : CLIENT_NUM : 16

show , buy , sell 모든 요청이 들어 온 경우

```
Event based average of 10 elapsed time : 13640 microseconds
```

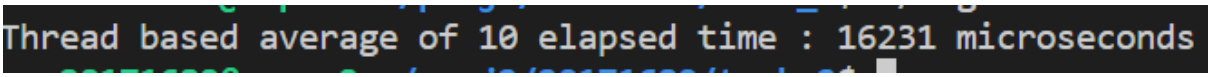
show 만 요청한 경우

```
Event based average of 10 elapsed time : 12978 microseconds
```

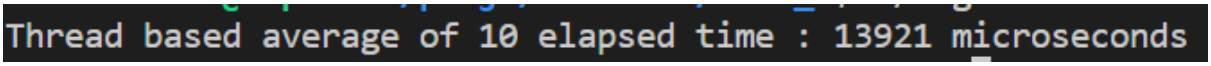
buy sell 만 요청한 경우

Thread based : CLIENT_NUM : 16

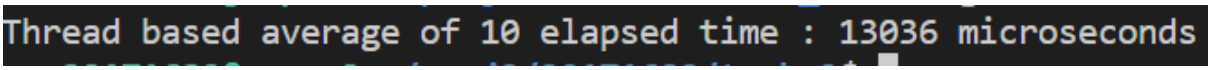
show , buy , sell 모든 요청이 들어 온 경우



show 만 요청한 경우



buy sell 만 요청한 경우



분석 2 :

Event based server 의 경우 3가지 상황 모두 비슷한 결과를 보였는데 실험 전 show 요청은 모든 노드를 순회하고 buy 와 sell 은 한 노드의 값 만을 바꾸기 때문에 시간차이가 날 것이라고 생각했으나 STOCK_NUM이 10개로 노드의 개수가 많지 않고 buy 와 sell 요청 역시 알맞은 노드의 index를 찾는 과정이 필요하기 때문에 비슷한 결과가 나왔다.

Thread based server 의 경우 semaphore 를 이용하여 Reader 와 Writer 가 나뉘어 있다. show buy sell 을 모두 요청하는 경우 reader 와 writer 가 모두 존재하여 reader 가 있을 때 writer 가 대기 하여야 하는 delay 가 생기지만 reader만 존재하는 show의 요청만 있는 경우 reader lock 은 한번에 여러 reader 가 값을 참조 할 수 있기 때문에 소요시간이 적게 나왔음을 확인할 수 있다. buy 와 sell 을 요청하는 것 역시 reader를 고려하지 않고 한번에 같은 노드만 write 하지 않으면 되므로 소요시간이 세가지 요청이 다양하게 있을 때 보다 적게 걸린 것을 확인 할 수 있다.

분석 3 : 이론과의 비교

수업 중 Event based server는 구현하기 어려운 대신 process/thread 의 overhead가 없기 때문에 좋은 성능을 낼 수 있지만 Thread based server에 비해

fine-grained concurrency를 제공하기는 어렵다고 배웠다. 이는 해당 프로젝트로도 확인 할 수 있었다. Event based server의 경우 pool 과 select 등을 활용하여 한 logical flow 로 concurrent programming 을 코드로 구현하기 힘들었지만 Thread based server 의 경우 client 마다 thread 가 요청을 처리하여 peer thread 가 어떠한 작업을 처리할지 별개로 생각 할 수 있어 코드 작성이 수월하였다. 또한, 측정 1 을 통해 Event based server 의 경우 client의 수가 적을 때는 thread based server 보다 좋은 동시 처리율을 보였지만 client의 수가 증가할 수록 그 차이는 작아지는 것을 확인 하였고 측정 2 를 통해 lock 을 통한 overhead가 유의미 하다는 것을 확인 할 수 있었다. 이렇게 이론과 일치하는 프로젝트의 결과를 통해 해당 프로젝트가 성공적으로 진행되었음을 확인할 수 있다.