System Programming Project 2

담당 교수 : 김영재

이름 : 김도현

학번 : 20181600

1. 개발 목표

- 해당 프로젝트에서는 여러 client들이 동시 접속하여 주식을 거래할 수 있는 Concurrent stock server를 구축한다.
- 각 client에서는 주식 사기, 팔기, 가격과 재고 조회 등의 요청을 하며
- server에서는 가지고 있는 주식 정보를 바탕으로 여러 client들과 통신하여 요청을 수행한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach
   - Event-based server를 구축하여, 동시에 다수의 클라이언트에서 들어오는 주식의 매매 요청과 주식 현재 상태 확인 요청을 I/O Multiplexing 기법을 사용하여 처리할 수 있다.
2. Task 2: Thread-based Approach
   - Thread-based server를 구축하여, 동시에 다수의 클라이언트에서 들어오는 주식의 매매 요청과 주식 현재 상태 확인 요청을 멀티쓰레딩 방식으로 처리할 수 있다.
3. Task 3: Performance Evaluation
   - Client의 configuration들을 바꿔 가면서 앞의 두 방식에 대한 elapse time을 측정하여 비교한다.
   - 각 방법의 client 개수 변화에 따른 동시 처리율 변화, client 요청 타입에 따른 동시 처리율 변화, 성능 등의 관점에서 비교 분석하여 확인할 수 있다.

B. 개발 내용

- **Task1 (Event-driven Approach with select())**
  - ✓ 각각의 client에서 connection request를 요청하면 server는 select() 혹은 epoll() 함수를 사용하여 어떤 descriptor와 연결할지 결정하고, 해당

descriptor와 연결되면 들어온 입력에 대한 함수를 수행한다. 이 때 선택된 descriptor의 요청을 제외한 다른 요청들은 pending되며, 각각의 입력을 event라고 할 수 있다. 하나의 event에 기반한 처리를 하고 있을 때에는 다른 신호를 받지 않는데, select()나 epoll()함수를 통해 한 번에는 반드시 하나의 event를 처리하고 다음으로 넘어가는 것이 I/O Multiplexing이다.

- ✓ Select() 함수는 read/write/error 세 가지 I/O에 대한 정보를 받는데, I/O 신호가 들어올 수 있는 관찰 영역에 포함되는 모든 file descriptor에 대해 FD_ISSET으로 체크를 한다. 또한 FD_SET을 계속해서 select문을 통해 OS에 전달한다. 반면 epoll() 함수는 전체 file descriptor에 대한 정보를 매번 넘기지 않으며, FD_SET에 해당하는 fd 저장소를 OS에서 직접 저장하며 이에 대한 접근을 요청하면 그 저장소에 해당하는 file descriptor를 리턴하여 변경을 요청할 수 있게 한다.

- **Task2 (Thread-based Approach with pthread)**
  - ✓ Master Thread는 클라이언트와 접속하여 얻은 connfd를 저장하기 위한 buffer를 필요로 하며, 이러한 공유 버퍼(shared buffer)를 선언하여 해당 버퍼에서 fd를 가지고 connection을 관리한다.
  - ✓ Worker Thread Pool은 Worker Thread들이 요청의 끝, 즉 EOF를 받을 때마 Pthread_detach() 함수를 통해 커널에서 해당 스레드를 종료시키는 식으로 관리한다.

- **Task3 (Performance Evaluation)**
  - ✓ 동시 처리율을 C라고 하고, client의 개수를 n, client당 요청 개수를 r, 전체 elapsed time을 T라고 했을 때 다음과 같은 식을 도출해낼 수 있다. 이 때 T의 단위는 초(sec)이다.

$$C = \frac{n \times r}{T}$$

  - ✓ Configuration 변화에 따른 예상 결과 서술
    - ◆ Client 개수에 따른 동시 처리율
      - ● Client 개수가 늘어날수록 동시에 처리해야 하는 명령어의 개수가 많아지므로 동시 처리율은 낮아질 것으로 예상된다.
    - ◆ 명령어에 따른 동시 처리율

- Buy, Sell은 특정 ID를 가진 stock item에 대해서만 연산을 수행하므로 Show 연산만을 사용했을 때보다 동시 처리율이 오를 것으로 예상된다.

◆ Thread-based server와 Event-based server의 차이

- Thread-based server의 overhead가 Event-based server보다 크므로 request의 수가 많아질수록 event-based server의 동시처리율이 우수할 것으로 예상된다.

C. 개발 방법

i. 공통

- Stock item : csapp.h에 구현되어 있으며, ID, left_stock, price를 가진다. Task2에 한해서 semaphore 구현을 위한 mutex와 w, readcnt 값까지 가진다.
- 주식을 저장할 자료구조는 이진 트리이다.
- 기존에 존재하던 echo.c 파일에 파일 이름과 함수 이름을 trade로 바꾸고, show, buy, sell, exit 명령을 처리하는 함수를 구현한다.

ii. Event-driven Approach

- Clientfd Pool을 관리하기 위한 struct pool을 구현하고, client를 pool에 집어넣고 삭제하는 함수를 구현한다.

iii. Thread-based Approach

- Master thread가 제공하는 shared buffer를 구현하기 위해 sbuf를 구현하며, 미리 define한 값만큼 work thread를 생성하여 해당 thread마다 명령어를 받아들여 구동하는 함수를 구현한다.
- 이 때 전역변수 tree[]를 사용하기 때문에 reader-writer problem을 방지하기 위해 mutex를 설정한다.

3. 구현 결과

✓ 전역변수 tree 배열을 선언하여 인덱스 1이 tree의 root가 되고 2*n이 left child, 2*n + 1이 right child인 이진 트리가 되도록 구현했다.

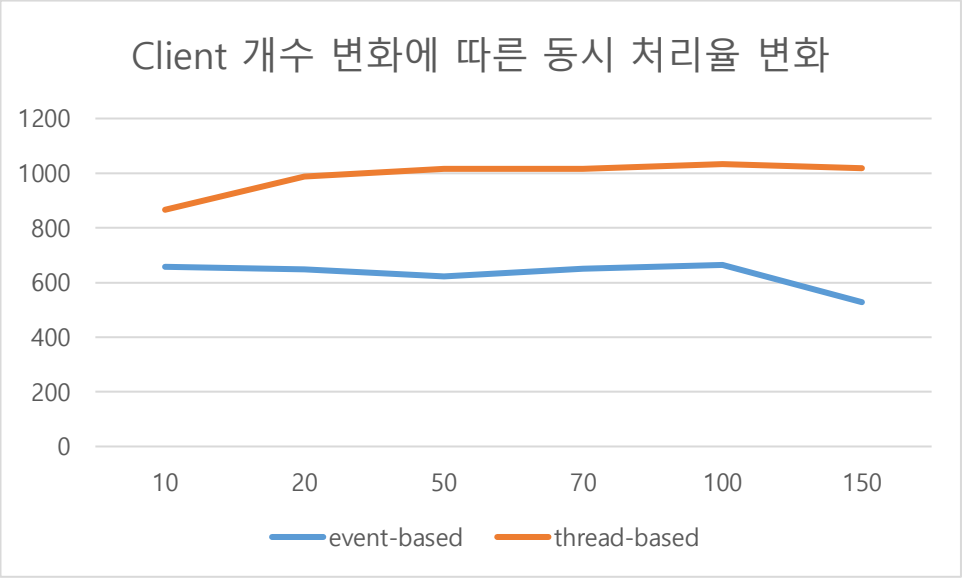
✓ Task1과 task2 공통적으로 trade.c에 있는 trade 함수에서 명령어를 받으며, buf 전체를 받아 이를 strncmp로 show, buy, sell, exit 함수 각각을 구분하여 해당 명령어를 수행한다.

✓ Task1에서는 fd들을 관리하기 위한 pool struct를 선언하였고 FD_ISSET 함수에서 이러한 pool에 있는 conncetion들을 Accept하면서 client를 추가하고, 추가된 client의 connfd에서 들어온 요청을 trade.c에 있는 trade 함수에서 처리하도록 하였다.

- ✓ Task2에서는 강의 시간에 배운 바 있는 sbuf 라이브러리를 사용하여 공유 buffer로 각 thread를 관리하였으며, tree[]가 전역변수이므로 thread-safe하지 않아 주식 종목을 추상화한 item struct에 이를 lock할 수 있는 mutex 변수 두 개를 추가하였다. Show와 파일 저장에 있어서는 client가 reader이므로 mutex를 하나만 lock했지만, buy와 sell에서는 writer이므로 mutex와 w를 모두 lock하여 안정성을 확보하였다.
- ✓ 저장은 본래 서버가 종료되는 Ctrl + C 입력에 대해 signal handler로 저장하는 함수를 구현하는 방법도 있으나, 서버가 예상치 않게 종료되었을 때도 파일의 안정성을 확보하기 위해 연산이 진행될 때마다 파일에 write를 해주는 방법으로 구현하였다.
- ✓ Thread-based server에서 Segmentation Fault가 뜨는 issue가 있었는데, 주로 Client가 매우 많아지거나 했을 때 buffer가 덧씌워지면서 제대로 명령을 읽지 못하고 파일이 꼬여 stock.txt의 내용이 삭제되거나 0으로 초기화되는 이슈가 있었다.

4. 성능 평가 결과 (Task 3)

- Client 개수 변화에 따른 동시 처리율 변화



x축은 client 개수, y축은 전술한 식으로부터 도출한 동시처리율이다. 전체적으로 event-based server의 동시처리율이 thread-based server의 동시처

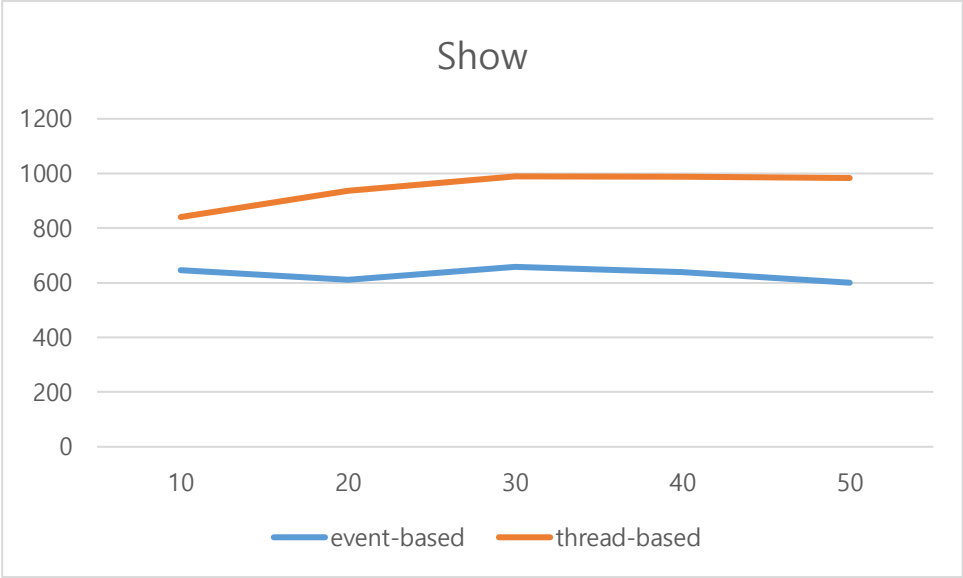
리율보다 낮았으며, client의 개수가 늘어날수록 thread-based server의 동시처리율은 조금 증가하다가 감소하는 추세를 보였고, event-based server의 동시처리율은 전체적으로 감소하는 추세를 보였다.

- 워크로드에 따른 분석 : random하게 요청하는 경우는 위에서 설명한 것과 동일하다.

  ✓ Buy, sell만 요청하는 경우

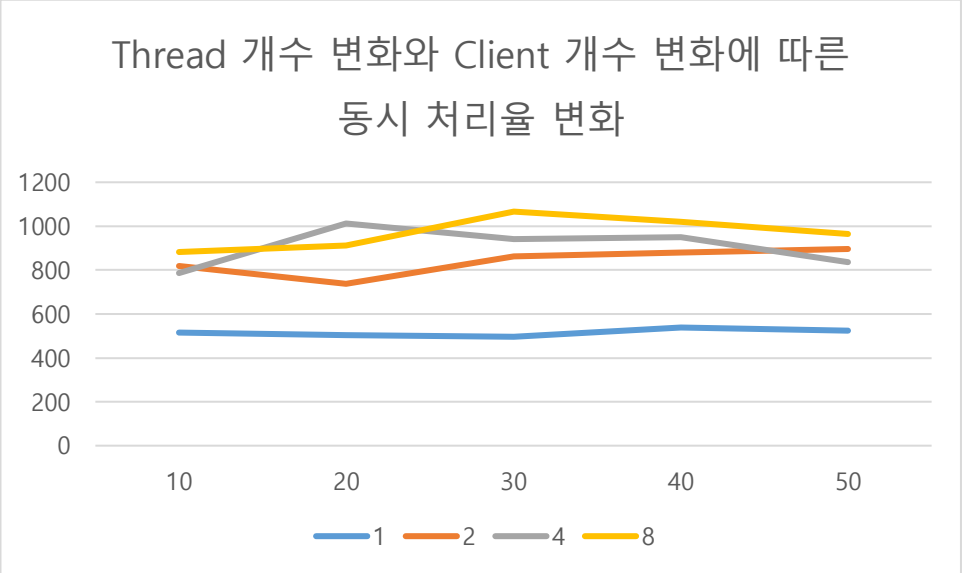
  Thread-based server의 경우 클라이언트의 개수가 많아질수록 동시처리율이 감소세를 띄었지만 event-based server의 경우 클라이언트의 개수가 많아질수록 약간의 상승세를 보였다.

  ✓ Show만 요청하는 경우



Event-based에 비해 thread-based의 그래프가 전체적으로 위로 올라와 있는 것을 확인할 수 있다.

- Thread-based server에서 Thread 개수에 따른 차이



전체적으로 명확한 추세가 드러나지는 못했으나 예상했던 것처럼 Thread의 개수가 많을수록 동시 처리율이 늘어나는 것을 볼 수 있다. 그러나 Thread의 개수가 많아지는 것에 비례하지는 않았으며, 오히려 Thread의 개수가 늘어날수록 동시처리율의 차이는 미미해졌다고 볼 수 있다.