System Programming Project 5

담당 교수 : 김영재

이름 : 원성현

학번 : 20161614

1. 개발 목표

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

여러 client들의 동시 접속 및 서비스를 위한 Concurrent stock server 을 구축해보는 것이 개발 목표이다. 여기서 주식 서버는 주식 정보를 저장하고 있고 여러 client들과 소통하며, 주식 클라이언트는 server에 주식 사기, 팔기 등의 요청을 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Select
   Select 를 이용한 구현, 즉 event-driven approach를 구현했을 때에는 select를 이용해서 multiclient들을 concurrent하게 다룬다.
2. Pthread
   Pthread를 이용한 구현, 즉 thread approach를 구현했을 때에는 쓰레드를 계속 생성하고 이것을 client마다 할당하여 concurrency를 구현한다.

B. 개발 내용

- 아래 항목의 내용만 서술
- (기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)
- **select**
  - ✓ select 함수로 구현한 부분에 대해서 간략히 설명
    Select는 input된 events(여기서는 각종 연결)들을 감지하고, add_client를 호출해서 새로운 논리흐름을 만든다. (thread나 process를 fork하는 것과 동일).
    이렇게 add_client를 실행하고, 그 다음에 check_clients함수를 호출한다.
    이 함수는 clients들의 pool에서 client들을 검색하며 현재 file descriptor가 SET이라면(즉, 연결이 active)하다면 Rio함수를 통해 통신하고, 읽기쓰기를 한다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명
명세서에서 요구한대로, stock.txt를 받아와서 구조체의 array에 저장했다. 그리고 그 구조체를 Node-화 시켜서 노드들로 이진탐색트리를 구성했다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명
멀티쓰레드를 이용한 구현은 다음과 같다. Read와 Write가 필요하므로 sbuf를 이용한다. Sbuf_init으로 초기화하고, Pthread_create로 스레드를 생성한다. 그리고 스레드에서는 buffer에서 connfd를 제거, 그리고 echo_cnt를 실행하되, echo가 아니라 통신으로 전해받은 command string에 대한 처리를 진행한다.

C. 개발 방법

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

이상의 내용을 구현하기 위해서 수업시간에 강의하였고, 책에서 제공하는 reference코드인 sbuf.c와 sbuf.h를 이용했다.
select와 pthread 모두 이진트리를 이용한 자료구조가 필수적으로 요구되기 때문에, stock.txt의 데이터를 가진 구조체를 만들고, 이것으로 이진탐색트리를 구성하였다.
또한, 두 방식 모두 제공해준 코드의 echo기능을 제거하였다. 대신 command string을 받고, server에서 case-by-case로 나누어 원하는 기능을 실행후, 그 결과를 다시 client에게 보내주는 방식을 선택했다.

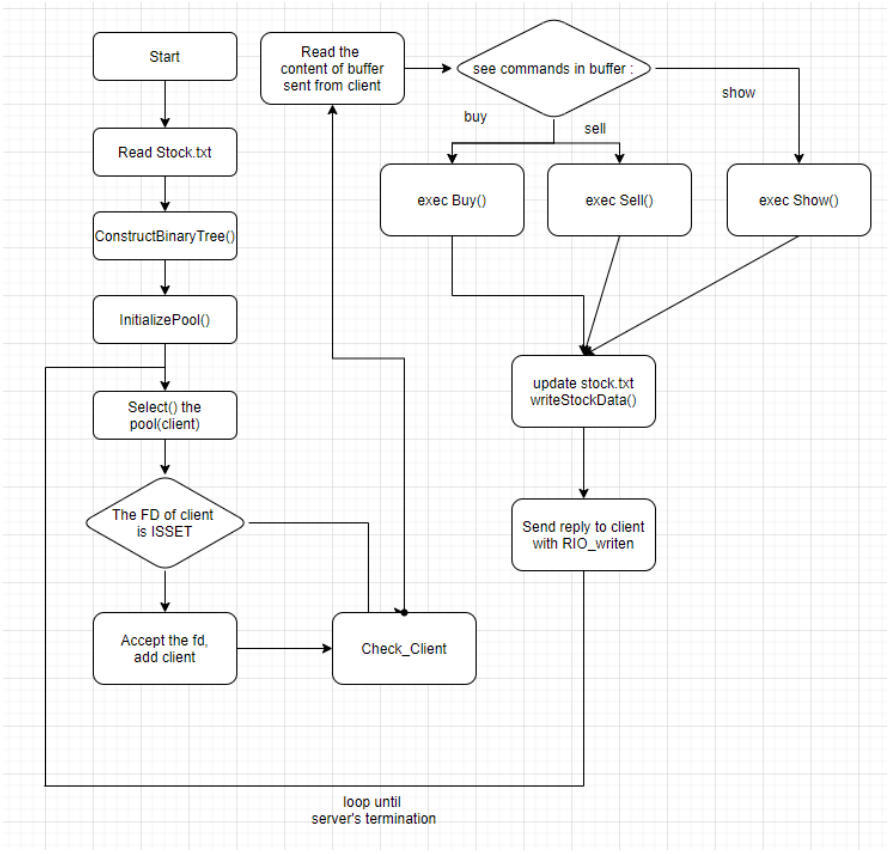
3. 구현 결과

A. Flow Chart

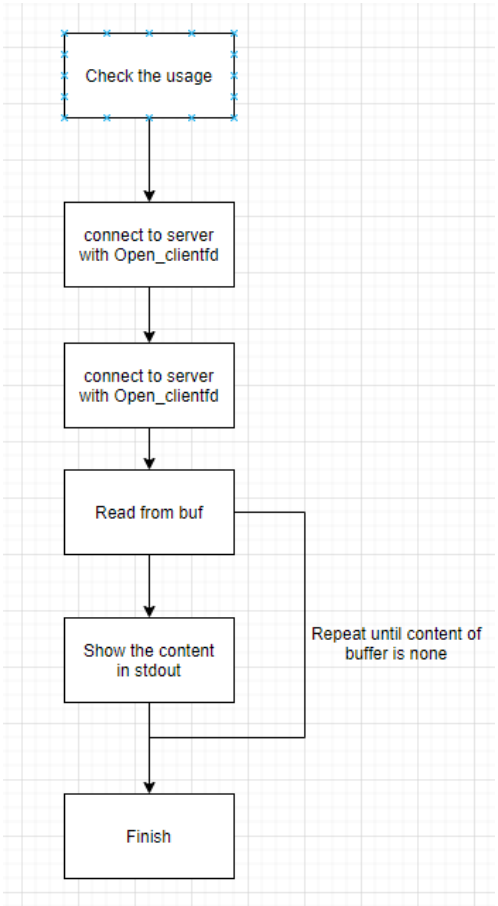
- **2.B.개발 내용에 대한 Flow Chart를 작성.**
- **(각각의 방법들(select, phtread)의 특성이 잘 드러나게 그리면 됨.)**

1. Select

stockserver.c

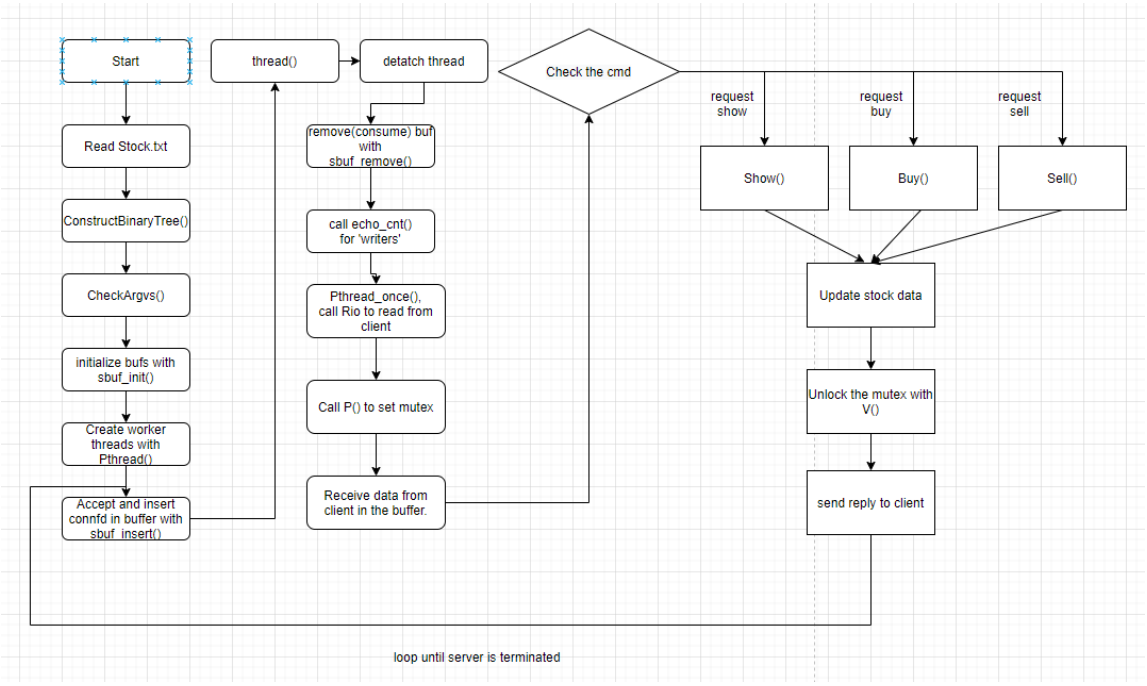


stockclient.c



2. Pthread

stockserver.c



stockclient.c -> 1번과 동일함.

B. 제작 내용

- **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
- **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

1. Select

첫째로 data structure을 구성하기 위해서 stockNode를 구성한다.

StockNode는 id,quantity,price의 3가지 정보를 담고 있으며 이것으로 tree를 구성하는데 treenode는 left,right의 child와 stockNode data를 담고 있다.

그리고 A에서 설명한 함수들의 호출을 통해 concurrency를 구현한다.

최초로 init_pool함수를 호출한다. 여기서 pool은 당연히 client들의 pool을 의미한다. connection이 생기면 add_client로 fd를 add하고, check_clients를 호출한다. 현재 FD가 Set된 (connection active)한 client들에 대해서 통신을 Rio함수를 이용해 주고 받는다. Server는 받은 신호가 정상 input이라는 가정하에, buy/sell/show등의 명령을 처리한다.

Buy에서는 tree의 id를 가지고 있는 node를 search하여 그 개수를 서버에서

decrement한다. Sell은 반대로 increment한다. Show는 현재 트리의 데이터를 inorder()로 출력한 뒤에, 그것을 저장하고 결과를 client에게 보내준다.

2. **Pthread**

   Pthread의 경우 역시 Select와 큰 차이 없다.
   하지만 pool을 관리하는 과정 없이, create_thread를 통해 주어진 개수만큼 worker thread를 생성한다. 그리고 각 thread에 대해 echo_cnt함수를 실행한다. Select와 마찬가지로 buy/sell/show의 명령을 처리한다.
   함수는 그대로 재활용하였다.

C. 시험 및 평가 내용

- **select, pthread에 대해서 각각 구현상 차이점과 성능상에 예측되는 부분에 대해서 작성. (ex. select는 ~~한 점에 있어서 pthread보다 좋을 것이다.)**
- **실제 실험을 통한 결과 분석 (그래프 삽입)**

1. 방식에 따른 예측

Select의 경우 Event-driven, 즉 iterative approach라고 할 수 있다.
우리는 수업시간에서 thread를 생성하는 것은 fork(), 즉 process를 생성하는 것보다는 못하지만 일정의 오버헤드를 발생시킨다는 것을 배웠다. 즉 메모리와 시간을 잡아먹는다는 것이다. 하지만 event-driven은 이러한 overhead가 없기 때문에 처리해야 할 client의 수가 적을 때에는 thread-based보다 좋을 것이다. 하지만 big scale의 경우 전체적인 효율성은 thread-based보다 떨어질 것이므로, 아마 client의 개수의 임계점을 돌파하면 그때부터 thread-based가 시간이 더 빠르지 않을까 생각한다.
아마도, event based는 선형적으로 증가하되, 어떤 구간에서는 그 증가수치가 팍 올라가지 않을까 싶다. Thread-based는 초기에는 event랑 비슷하지만, 특정 클라이언트의 수가 넘어가면 event-based보다 빨라질 것으로 예측된다.
만약 cspro 서버가 훌륭한 multicore성능을 자랑한다면, 아마 thread 방식이 event랑 비슷하거나 더 빠를 것이다. 성능이 좋을수록 thread의 오버헤드는 줄어들 것으로 예상되기 때문이다.

따라서 최종 예측은 다음과 같다:
초기에는 event와 thread가 비슷할 것이다. (cspro의 멀티코어 성능이 준수한 경우)

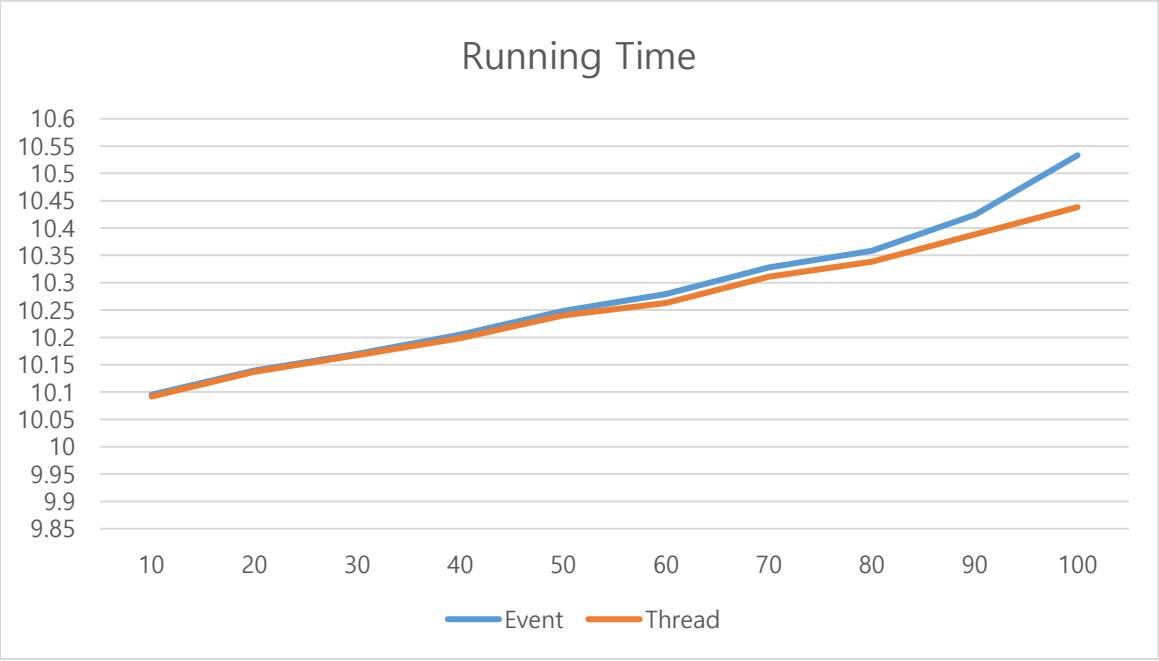
혹은 thread가 event보다 더 빠를 수도 있다(cspro의 멀티코어 성능이 훌륭한 경우)
중요한 것은 결국 어느 순간부터는 thread가 event보다 확연히 빨라지는 순간이 올것이다.

2. 결과 데이터

최종 측정 데이터는 다음과 같다.

	10	20	30	40	50	60	70	80	90	100
Event	10.0952	10.13917	10.16987	10.20493	10.24812	10.27945	10.32775	10.3583	10.42425	10.53316
Thread	10.09181	10.13705	10.16777	10.19832	10.2404	10.26323	10.31093	10.33841	10.3883	10.43826

표 안의 숫자는 실행에 걸린 시간을 의미한다.
이것을 그래프로 나타내면 다음과 같다.



그래프를 분석해보자면, 예상내의 결과였지만 눈여겨 볼만한 부분이 있었다.
두 방식 모두 선형적으로 증가하는 추세를 보이지만, thread의 경우 초반에 event-driven 방식과 크게 차이가 나지 않았다.
처음에는 thread방식의 경우 쓰레드 생성 오버헤드 때문에 event보다 월등히 느릴 것이라고 생각했다. 하지만 cspro 서버컴퓨터의 병렬컴퓨팅 능력은 내 상상이상인 모양이다.
초반-중반까지 thread방식은 event방식과 항상 비슷하거나 더 빠르다.

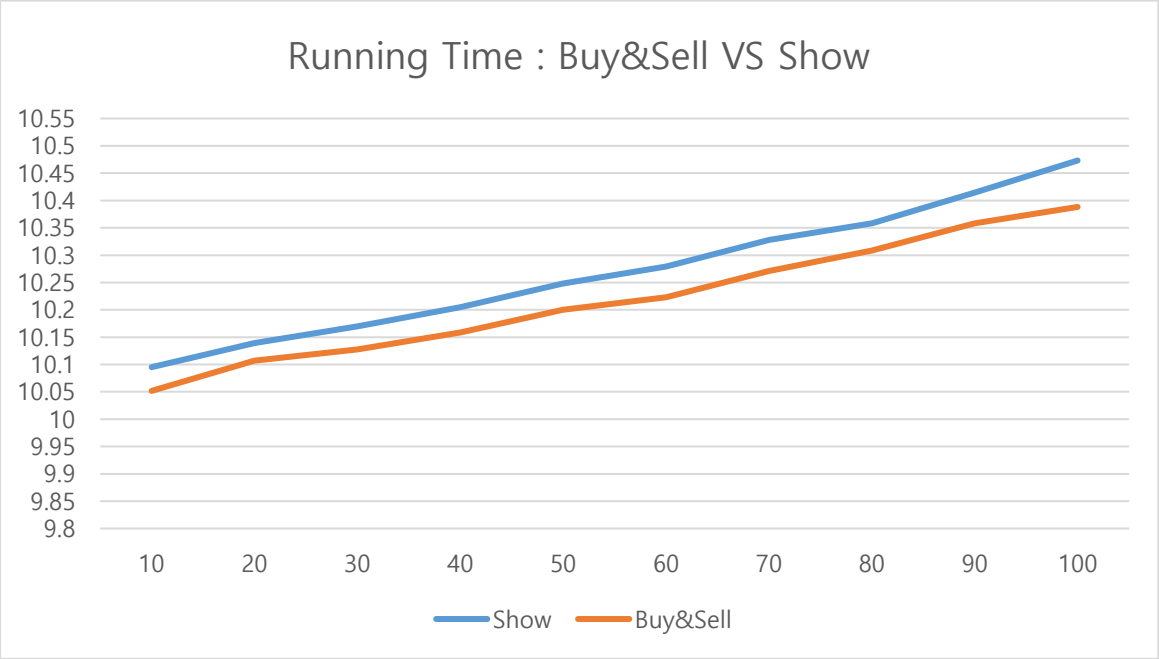
결과적으로 thread방식이 초반단계에서 event방식과 같거나 나은 결과를 보여줬다.

추가로, 임계점에 관한 예측을 했었는데 여기서는 client수가 60정도일 때가 임계점으로 보인다. 이때를 기점으로 event방식이 thread방식보다 시간이 훨씬 오래걸린다.
임계점에 대한 예측은 옳게 된 것 같다.

결과를 종합하자면, 초반에는 event, 후반에는 thread가 우세할 것이라고 점쳤으나, 실제로는 thread가 event의 상위호환이라고 볼 수 있겠다.

3. 추가적인 분석

추가로 명령의 종류에 따라 그 수행속도를 분석해보았다.
명령에서 참조하는 변수가 많을수록, memory access가 잦을수록 속도가 느릴것이라고 초기에 예상했다. 그리고 이것은 예측과 들어맞았다.



실제로 구현된 함수 Buy, Sell, Show에서 Buy는 id를 통해 노드를 검색하고, 이 노드의 값을 감소시킨다. Sell은 노드의 quantity값을 증가시킨다. 따라서 Buy과 Sell은 오버헤드가 동일하므로 수행속도가 비슷하거나 동일할 것이라 예상했다. 그리고 실제 두 수행속도는 비슷했다.
Show의 경우가 조금 달랐는데, Show에서는 루트를 기반으로 모든 트리를 탐색한다.
즉 buy와 sell과는 탐색하는 트리의 범위가 다르다. Buy와 sell은 최악의 경우, 즉 leaf

node중에서도 일부의 case에만 모든 노드를 탐색하지만 Show는 상황을 불문하고 tree의 모든 data를 출력하기 위해 Inorder traversal을 진행한다.
따라서 Buy와 Sell처럼 노드 1개 탐색은 '대다수의 경우 운이 나쁘지 않다면(not the worst case)' 모든 Inorder traversal을 수행하는 Show보다 느릴 것으로 예상했다.
그리고 실제로 수행속도는 Show()가 훨씬 더 빠르게 나타났다.