# System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 김성희

학번 : 20141196

## 1. 개발 목표

미니 주식 서버 개발.

여러 명의 고객이 서버에 접속할 수 있도록 서버를 구현한다.

고객들은 주식의 잔량과 가격을 볼 수 있고 주식을 사고 팔 수 있다.

## 2. 개발 범위 및 내용

### A. 개발 범위

1. select

   확인할 소켓을 선택하여 소켓에 따라(listen소켓, client소켓) 처리한다.

2. pthread

   고객과 소통하기 위한 스레드를 생성하여 해당 스레드에서 고객의 요청을 처리한다.

### B. 개발 내용

- **select**
  - ✓ selet 할 때마다 감시대상으로 등록한 소켓 중 하나를 선택한다.
  - ✓ 선택된 소켓이 서버 소켓이면 연결 요청이 들어왔는지 판단하여 연결을 승인하고 그 외의 고객 소켓이라면 고객의 요청에 대한 처리를 진행한다.
- **pthread**
  - ✓ 서버 소켓을 통해 고객의 연결 요청을 들은 뒤 스레드를 생성한다.
  - ✓ 스레드를 생성할 때 진행해야 할 프로세스를 함수로 처리하여 해당 함수를 넘긴다.
  - ✓ 추가로 스레드를 생성할 때 고객의 정보와 소켓 정보, 스레드 번호(혹은 인덱스)를 넘긴다.
- **stock info (file -> memory)**

- ✓ 이진 검색 트리를 사용하여 메모리에 올린다.
- ✓ 우선 배열에 저장하여 정렬을 한다.
- ✓ 정렬한 뒤 가운데 값을 루트로 잡은 뒤 이진 분할을 하여 첫번째 부분배열에서 다시 가운데 값을 왼쪽 자식에, 두번째 부분배열에서 가운데 값을 오른쪽 자식에 할당하는 방식을 반복한다. (마지막 잎 노드는 -1 id를 통해 nill 노드를 구현한다.)

### C. 개발 방법

공용

1. stock info에 대한 구조체: struct item

주식의 ID, 잔량, 가격, readcnt, readcnt용 mutex, write용 mutex

2. stock info를 트리로 만들기 위한 node: stock_node
    A. struct item
    B. stock_node *left_child
    C. stock_node *right_child
3. stock info 트리를 만들기 전 동적 배열을 만들기 위해서 list 사용: stock_list
    A. stock_node *head, *tail
    B. int length
4. struct item dealing functions (구체적인 것은 실제 stock.c, sotck.h 확인)
5. list dealing functions (구체적인 것은 실제 stock.c, sotck.h 확인)
6. Binary search tree dealing functions (구체적인 것은 실제 stock.c, sotck.h 확인)
7. sorting functions (구체적인 것은 실제 stock.c, sotck.h 확인)

merge sort

8. stock tree 만들기 함수 (구체적인 것은 실제 stock.c, sotck.h 확인)
    A. stoi(): string to integer

B. parse_stock(): stock info인 id, 잔량, 가격을 읽어서 integer 배열에 저장

C. read_stock_list(): stock info가 있는 파일을 읽어서 메모리에 올린다.

9. 고객 요청에 대응되는 함수 (구체적인 것은 실제 stock.c, sotck.h 확인)
   A. 고객이 입력한 정보를 해석하여 show, buy, sell에 맞는 처리를 위한 함수들.
   B. 대표적으로 stock(), show(), buy(), sell()

project1

1. stockserver.c 함수: select 함수를 이용한 event-based 서버용 코드로 수정
2. char stock_filepath[]
3. int global_listenfd: ctrl-c 로 강제 종료 시 listenfd 닫기위해 사용.

project2

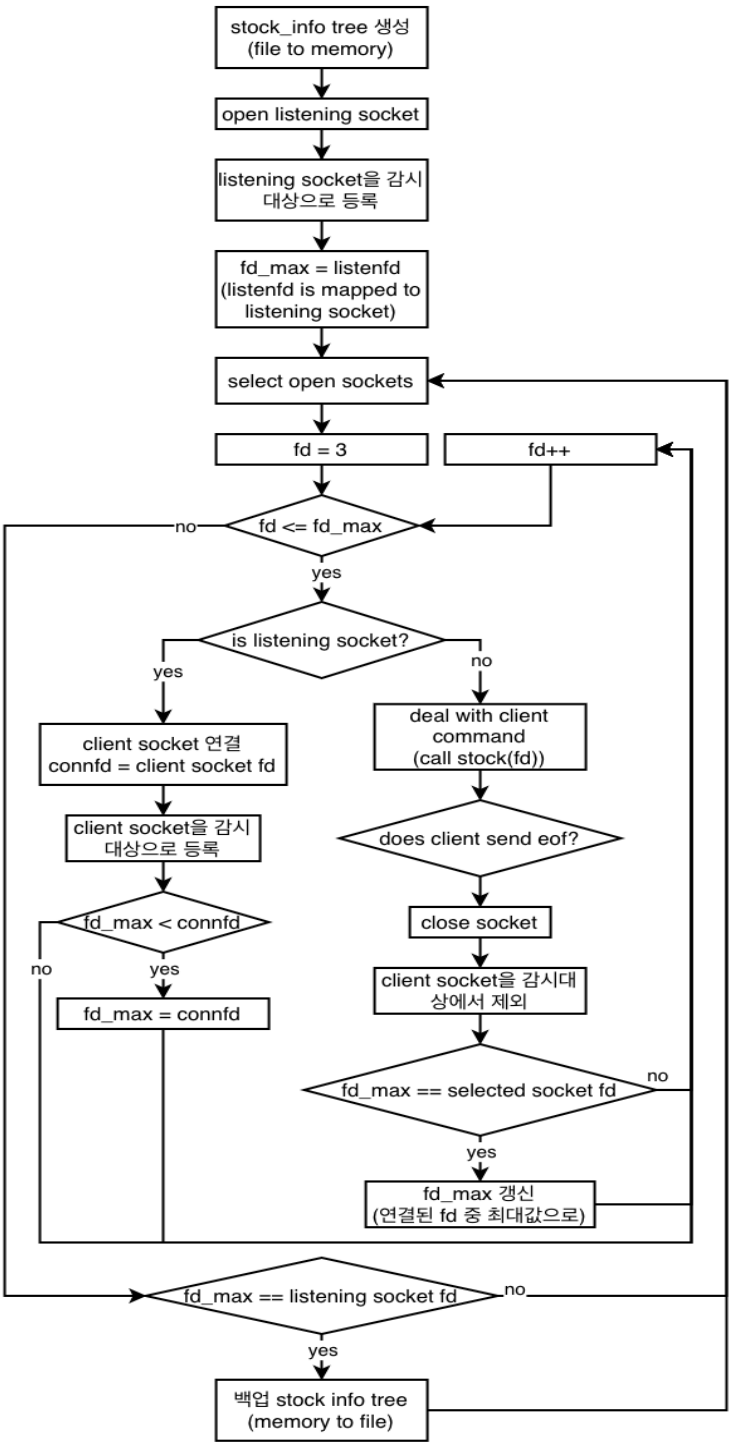
1. stockserver.c 함수: thread 함수(pthread_create, pthread_self 등)을 이용한 multi-threads 서버용 코드로 수정
2. struct thread_info:
   A. tid
   B. fd
3. char stock_filepath[]
4. int check_thread[]
   A. 스레드 array에서 사용 가능한 index 파악용
5. int thread_cnt: 사용 중인 thread 개수
6. sem_t mutex_cnt_t: thread_cnt 접근 및 변경 시 필요한 lock

## 3. 구현 결과

### A. Flow Chart

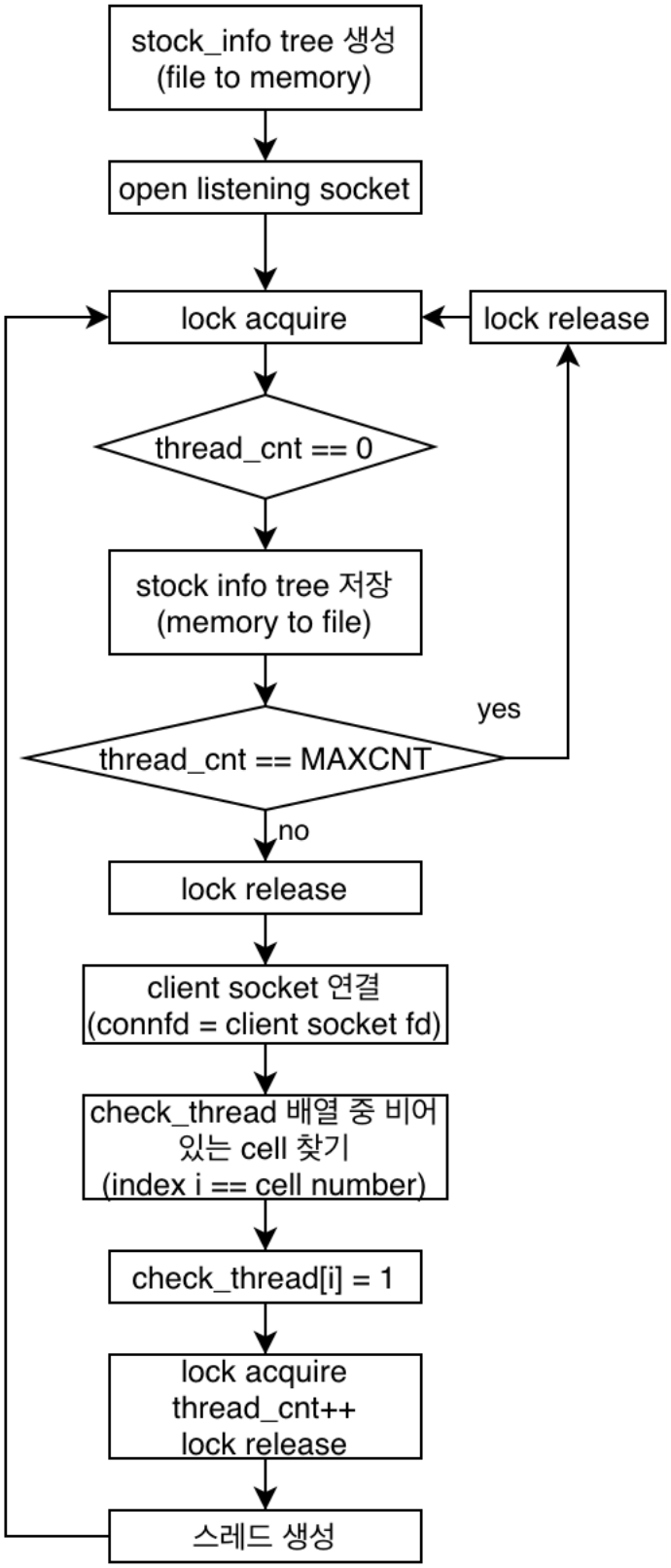
#### 1. select

**main()**

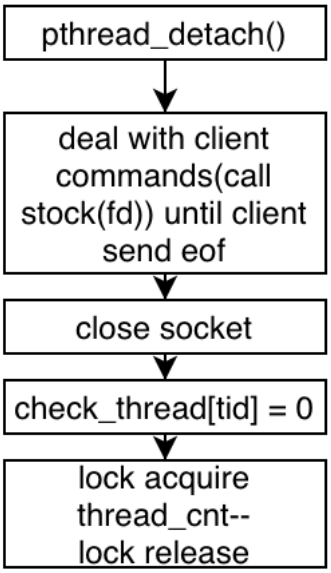


## 2. pthread

**main()**



thread()




**B. 제작 내용**

1. **select**

   **A.** fd_set 타입 변수에 연결된 소켓 번호가 저장된다.

   **B.** select 함수는 fd_set 타입 변수에 등록된 번호 중 하나만 남기고 없앤다. 따라서 select 함수를 쓸 때는 항상 임시 변수에 저장해서 임시 변수를 사용한다.

   **C.** 연결할 때는 FD_SET(fd, &fd_set 변수) 함수를 통해서 소켓 번호를 저장하고, 연결을 해지 할 때는 FD_CLR(fd, &fd_set 변수) 함수를 통해서 소켓 번호를 지운다.

   **D.** 항상 연결된 소켓 번호 중 max 값인 fd_max를 기억하고 3부터 fd_max까지 반복문을 통해서 FD_ISSET(fd, &fd_set 변수)로 select 한 socket 번호를 알아낸다.

   **E.** 선택된(알아낸) 소켓 번호가 listen 용이면 client socket 연결을 기다리고 client socket이면 stock(fd)를 호출하여 client와 소통한다. 소통이 끝나면 소켓을 닫고 연결 해지를 fd_set 변수에 알린 뒤 fd_max == fd이면 fd_max를 갱신한다.

   **F.** fd_max를 갱신할 때는 fd_max 부터 listenfd까지 순서대로 찾아보며 처

음으로 open socket fd가 나오면 갱신하고 반복문을 종료한다.

2. **pthread**

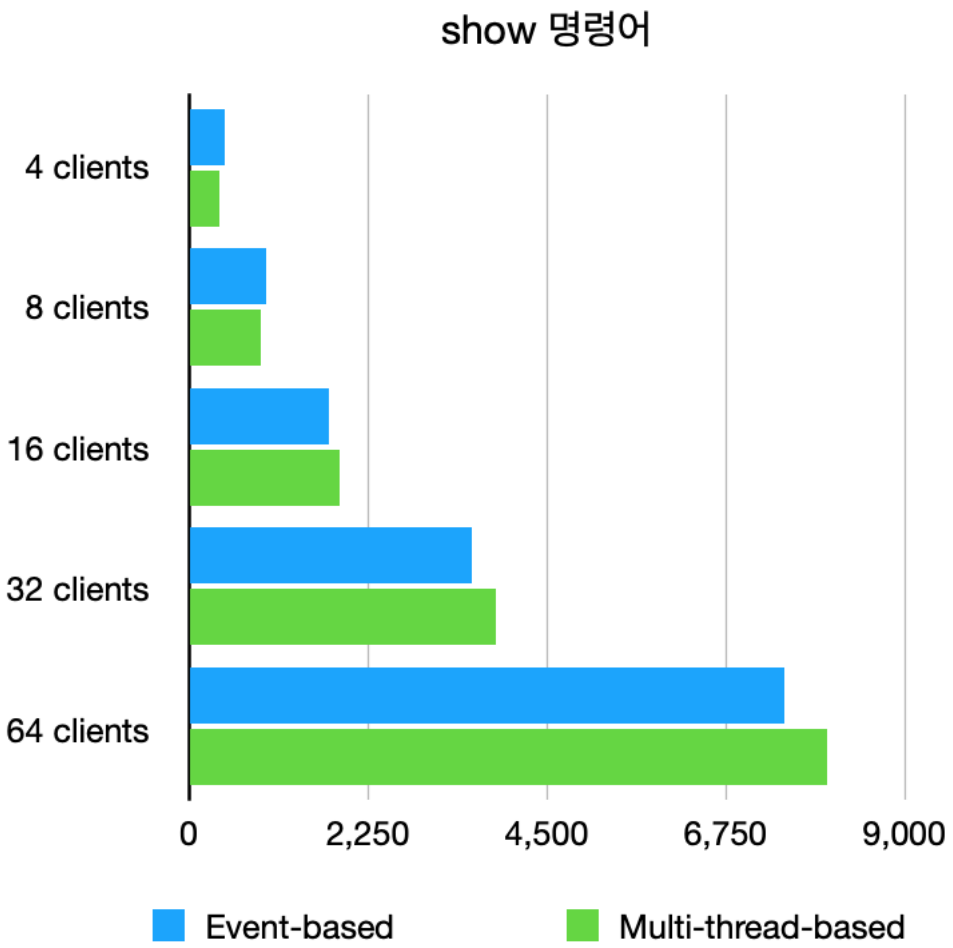
   **A.** client 소켓 연결 요청이 들어오면 check_thread 배열 중 비어 있는 index i를 찾아서 thread_arr[i]에 thread를 정보를 담고 스레드를 생성한다. 이 때 thread process 함수에 index i와 연결된 소켓 번호를 넘긴다.

### C. 시험 및 평가 내용

- **select, pthread에 대해서 각각 구현상 차이점과 성능상에 예측되는 부분**
- select는 control flow가 한 개여서 순차적인 진행을 해야한다.
- pthread는 control flow가 client 수만큼 존재한다.
- 만약 1개의 client의 요청당 서버에서 작업해야하는 시간이 길어진다면 pthread가 유리할 것이다.
- 만약 1개의 client의 요청당 서버에서 작업해야하는 시간이 짧다면 thread 생성, 및 수확 그리고 동기화 작업을 위한 mutex lock 때문에 select기반의 서버가 더 유리할 것이다.
- 이번 show/buy/sell 요청은 작은 단위이다. 그러므로 pthread 서버가 더 불리할 것이다.

- **그래프**

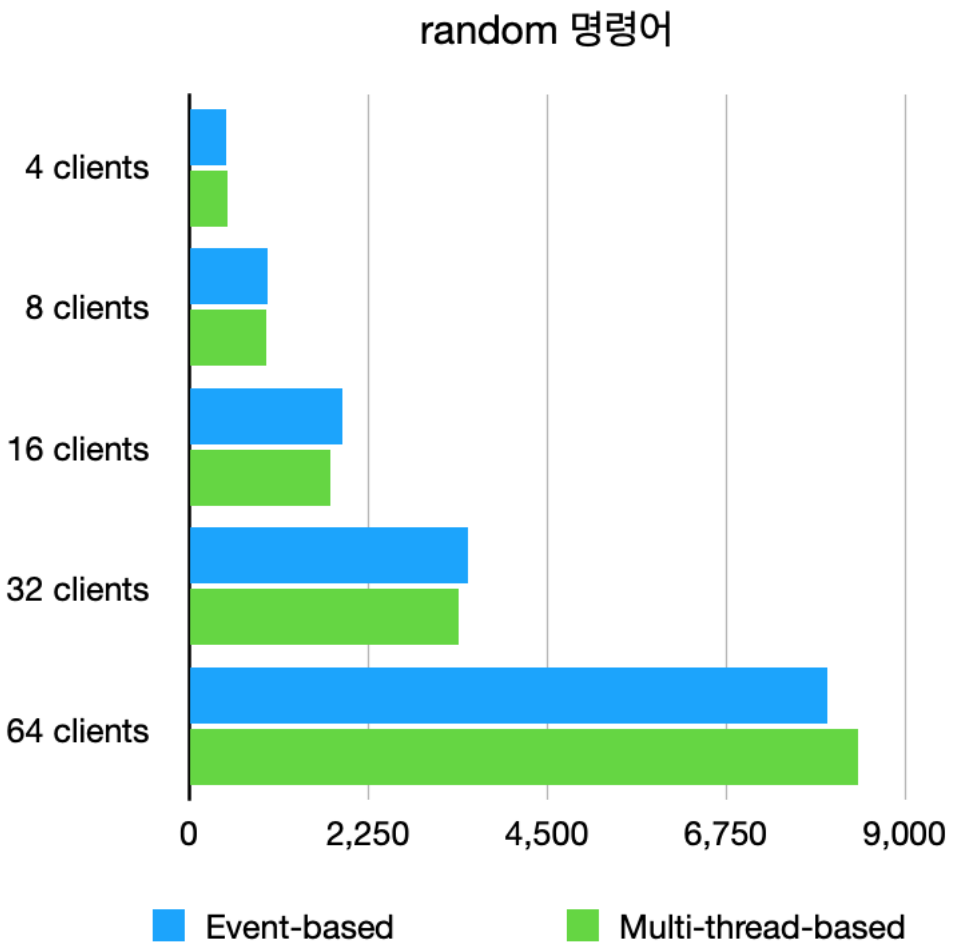
- 각 실험은 5번의 multiclient.c 를 실행하여 구한 평균값이다.
- 각 client마다 100번의 명령어를 수행한다.
- 가로축의 단위는 ms (= 0.001 sec)다.
- 주식의 번호는 1~20이다.




대체적으로 event-base server가 더 빠르고 클라이언트의 수가 증가할 수록 그 차이가 더 커지는 것을 확인할 수 있다. 이는 thread 생성 및 수확의 오버헤드가 누적되기 때문인 것으로 보인다.


show 명령어에 대한 내용과 같다.

이를 통해 show 명령어와 buy/sell 명령어의 작업 단위가 그리 차이가 나지 않는 것을 알 수 있다.




이 그래프를 보면 4 clients와 64 clients의 경우를 제외하면 multi thread server가 더 빠른 것으로 보인다. 그 차이가 작은 것으로 보아 client의 수가 적고 평균값을 내기 전 수행한 횟수가 충분치 못해서 좋은 데이터 set을 확보하지 못한 탓으로 보인다. 실제로 64 clients의 경우의 차이가 가장 큰 것을 확인할 수 있다.

결론: 대체적으로 예상과 맞아 떨어졌다. 즉 event based server가 평균적으로 multi threads based server보다 성능이 좋을 것이다.