System Programming Project 2

담당 교수 : 김영재

이름 : 최원빈

학번 : 20181693

1. 개발 목표

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

I/O 다중화에 기초한 이벤트 기반 방법, 그리고 쓰레드를 이용한 방법, 이 두 동시성 프로그래밍 방식을 이용하여 주식 서버를 개발한다. 개발한 주식 서버는 다수의 client로부터 동시적으로 request를 받아도 문제 없이 알맞은 응답을 동시적으로 할 수 있어야 한다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach
    - 동작하고 있는 클라이언트의 집합을 pool 구조체에서 관리한다. 이것을 초기화한 후 서버는 무한 루프 상태에 진입한다. 이때 서버는 select 함수를 통해 서로 다른 두 입력 이벤트를 감지한다. 새로운 클라이언트에서 연결 요청이 오면 서버는 새 연결 식별자를 열고, 클라이언트를 pool에 추가한다. 이후 서버는 읽을 준비가 된 연결 식별자들로부터 텍스트 라인(request)을 받는다. 이후 그 request를 분석해서 알맞은 response를 보낸다.
2. Task 2: Thread-based Approach
    - 메인 쓰레드는 client에서 온 연결 요청을 처리한다. 이후 생성한 피어 쓰레드에게 연결식별자를 전달받고 작업 요청을 처리한다. 각 쓰레드들은 분리되어 종료시 메모리 자원을 자동으로 반환한다.
3. Task 3: Performance Evaluation
    - 성능 측정을 위해 프린트문이나 usleep 등을 뺀 만큼 원래의 프로그램보다 빠르다. Multiclient 실행 후 처음 요청부터 마지막 요청까지 걸린 응답 시간을 특정 텍스트 파일에 저장한다. 여러 번 반복 실행 후 평균을 구해 확인한다.

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

여기서 서버는 일종의 상태 머신으로 볼 수 있는데, 자체 루프를 돌면서 입력과 출력 상태의 전환을 반복하고 있다는 뜻이다. 이때 기본적으로 서버는 연결 요청을 기다리고 있고, 연결해준 각각의 새로운 클라이언트들에 대해서는 작업 요청을 기다리는 상태라 볼 수 있다. 이때 결국 입력 이벤트의 발생 감지가 필요한데 이를 도와주는 것이 여기서는 select 함수이다. Select를 통해 서버에 온 연결 요청을 처리하고, 각 연결 식별자가 읽을 준비가 되면 서버는 대응하는 상태 머신을 전환시킨다.

✓ epoll과의 차이점 서술

select 함수는 동작 환경에 따라 다르지만 보통 fd 개수가 최대 1024개로 제한된다. 또 관찰 영역에 포함되는 모든 파일 디스크립터에 대해 순회하는 불필요한 체크를 하고, FD_SET을 계속해서 커널에 넘기는 것도 오버헤드가 상당히 크다. epoll은 이러한 select의 단점을 보완한 방법이다. 계속해서 정보를 넘기지 않기 위해 fd의 정보를 os가 직접 가지고 있다. 또 전체를 순회하지 않고 변경사항을 체크한다. 이렇듯 많은 부분이 개선되었지만 기본적으로는 프로세스가 커널에 지속적으로 I/O 상황을 체크하는 개념은 유효하다. 따라서 이번 프로젝트에서는 concurrent server 구현이 주목적이기 때문에 크게 신경쓰지 않고 select 함수를 사용하였다.

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

✓ Master Thread의 Connection 관리

마스터 쓰레드(메인 쓰레드)에서 client로부터 연결 요청을 받는다. 이후 피어 쓰레드를 생성해 연결 식별자를 전달한다. 생성된 쓰레드는 client로부터 오는 작업 요청에 따라 알맞은 응답을 한다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

생성된 쓰레드들은 요청을 처리하기 전에 pthread_detach 함수로 분리해 놓았다. 때문에 각 쓰레드들은 종료 후 자동으로 메모리 자원을 반환한다. 전역변수로 client_num을 선언해 놓았다. 이 변수는 연결된 client의 수의 정보를 갖고 있는데, 이는 결국 생성된 피어 쓰레드의 개수와 동일하다. 쓰레드가 생성될 때와 종료될 때 락을 잡고 업데이트를 해준다.

- **Task3 (Performance Evaluation)**

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

먼저 두 가지 동작 방식의 동시처리율을 측정하여 분석한다. 동시처리율을 'client 수 * client 당 요청 수 / 걸린 시간'으로 생각하면 두 방식을 같은 조건에서 걸린 시간을 측정하는 것은 그리 어려운 일이 아니지만 어느 쪽의 성능이 뛰어난지 명확히 알 수 있다. 이후 각 방식에 대해 작업 요청을 종류별로 나눠서 어떤 요청을 더 빠르게 처리하는지 확인한다.

✓ Configuration 변화에 따른 예상 결과 서술

쓰레드 베이스 서버의 동기화 연산은 굉장히 오버헤드가 크기 때문에 이벤트 베이스 서버가 더 성능이 좋을 것이다. 그러나 client의 수를 늘리면 쓰레드 베이스 서버는 fine-grained coucurrency를 제공하므로 그 차이가 줄어들 것이다.

Show는 트리의 모든 노드를 방문해야 하므로, 특정 노드를 찾으면 되는 buy나 sell보다 평균적인 작업 시간이 더 길 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)**

기본적으로 task1과 task2 모두 stock 정보를 저장하기 위한 동일한 자료구조(item)를 사용하였다. 차이점은 쓰레드 기반의 경우 readcnt, mutex와 w의 추가이다. 그 외에는 둘 모두 동일한 링크드리스트로 구현한 binary tree를 사용한다.

또 서버가 각 클라이언트의 요청(show, buy, sell, exit)에 대응하는 방식 또한 같은 코드와 함수를 사용한다. New_item, insert, search는 새 item 생성 및 삽입, 그리고 특정한 item을 찾는 함수다. Inorder, inorder2 함수는 show와 exit일 때 트리를 순회하며 정보를 출력 및 저장하는 함수이다.

처음 서버를 실행하면 서버는 stock.txt에서 정보를 불러와 자료구조에 저장해 관리한다. 또 client_num이라는 전역변수로 연결되어 있는 client 수를 확인해 0이 될 경우(처음 제외) stock.txt에 업데이트 한다.

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

  이벤트 베이스 서버는 pool 구조체가 추가되었다. 여기서 clientfd 배열은 연결된 식별자들의 집합을 나타낸다. 이 구조체를 초기화 하기 위해 init_pool 함수를 사용한다. 이후 서버는 무한 루프를 돌면서 select 함수를 통해 이벤트를 기다린다. 연결 요청이 온 경우 add_client 함수로 pool의 배열에 새 connfd를 추가한다. 이후 check_client 함수로 작업 처리 요청(client에서 보낸 텍스트 라인)이 있는 경우 해당 요청에 맞는 처리를 한다. Show는 현재 inorder 함수로 binary tree를 순회해 한 char 포인터에 알맞은 양식으로 저장한뒤 이를 보내준다. Buy는 남아있는 stock의 개수가 요청한 개수보다 많거나 같으면 실행되고 그렇지 않을 경우 실패한다. 각각의 경우 성공과 실패 메시지를 클라이언트에게 보낸다. Sell은 절대로 실패하지 않으며 무조건 업데이트된다.

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

  메인(쓰레드)에서 무한루프를 도는 것은 동일하지만 연결 요청을 받은 후 쓰레드를 생성한다. 이후 연결 식별자를 전달하고 그 쓰레드가 해당 client의 작업 요청을 처리한다. 이를 위해 item 자료구조에는 mutex와 w가 추가되었고 new_item에서 이들을 초기화 해준다. 이후 작업 요청을 처리할 때 동기화 연산(P와 V)을 사용하는데 show는 읽는 연산이므로 mutex와 w의 락을 잡고, buy나 sell연산은 w만 잡는다. 또 전역변수로 선언한 client_num으로 연결되어 있는 client 수를 확인하는데 이 변수도 각 쓰레드에서 업데이트 할 때 락을 잡고 업데이트한다.

- **Task3 (Performance Evaluation)**

Stockserver와 multiclient에서 오버헤드가 큰 프린트문을 제거한다. Multiclient 에서는 gettimeofday로 걸린 시간을 측정한 뒤 gettime.txt에 저장한다. 반복 실행 후 텍스트 파일에 어느 정도 케이스가 쌓이면 평균을 구해주는 프로그램으로 결과를 확인한다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**
- Task 1: Event-driven Approach

  여러 클라이언트의 여러 연결 및 작업 요청을 동시적으로 모든 경우에 대해 잘 동작한다. 서버를 실행하면 무한 루프에 진입하고 이후 클라이언트에서 연결 요청을 하면 연결 완료 메시지가 출력된다. 다음으로 show 요청 시 현재 stock의 정보를 받고 출력하고, buy 입력 시 잔여 주식 개수이 충분하면 성공 메시지를 부족하면 실패 메시지를 띄운다. Sell 명령은 항상 성공한다. 연결된 클라이언트가 0이 되면 stock.txt를 업데이트 한다.

- Task 2: Thread-based Approach

  Task 1과 동일한 방식으로 잘 작동한다.

- Task 3: Performance Evaluation

  성능 측정용 클라이언트 프로그램을 실행하면 응답시간이 텍스트 파일에 업데이트 된다. 이후 그 텍스트 파일의 응답시간들의 평균을 구해주는 프로그램 실행을 통해 평균값을 얻을 수 있다.

4. 성능 평가 결과 (Task 3)

- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)**
- 1. 확장성: 각 방법에 대한 Client 개수 변화에 따른 동시 처리율 변화 분석

  -ORDER_PER_CLIENT = 100, STOCK_NUM = 10, BUY_SELL_MAX = 10

  -Stockserver.c와 multiclient.c의 프린트문 및 usleep함수 제거 후 측정

-측정 시작 시점 : multiclient에서 while문 진입 직전

-측정 종료 시점 : multiclient에서 while문 종료 직후

-각 10번씩 실행 후 평균 출력.

Client num = 1

Event

```
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./multiclient 172.30.10.11 60029 1
cse20181693@cspro8:~/proj2/task3_1$ ./a.out
avg of elapsed  time : 0.028697s
```

Thread

```
avg of elapsed  time : 0.029694s
```

Client num = 4

Event

```
avg of elapsed  time : 0.041044s
```

Thread

```
avg of elapsed  time : 0.043913s
```

Client num = 10

Event

```
avg of elapsed  time : 0.083479s
```

Thread

```
avg of elapsed  time : 0.077720s
```

Client num = 20

Event

```
avg of elapsed  time : 0.171214s
```

Thread

```
avg of elapsed  time : 0.146092s
```

Client num = 50

Event

```
avg of elapsed  time : 0.400382s
```

Thread

```
avg of elapsed  time : 0.357460s
```

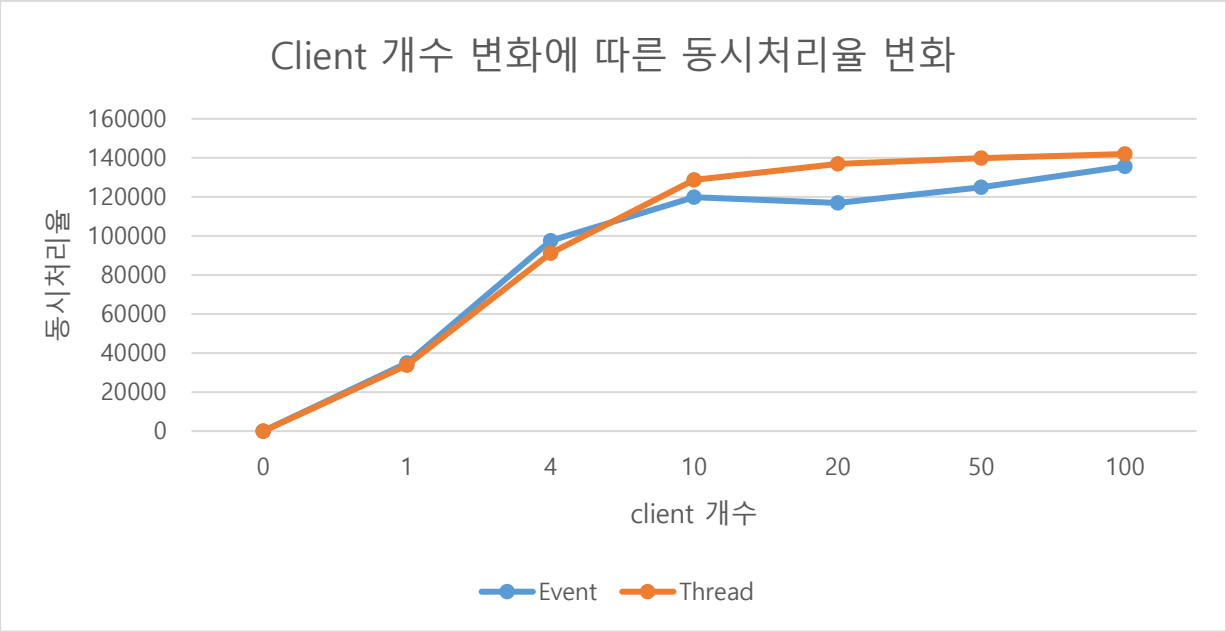
Client num = 100

Event

```
avg of elapsed  time : 0.737212s
```

Thread

```
avg of elapsed  time : 0.704422s
```



- 분석 : Event-driven 서버는 쓰레드 생성, 동기화연산 등 큰 비용이 필요한 경우가 Thread-based에 비해 적다. 때문에 클라이언트 수가 적으면 상대적으로 수행시간이 적게 걸리고 더 나은 확장성을 제공하는듯 보인다. 그러나 클라이언트 수가 늘어나면 multi-core 환경을 잘 활용할 수 있는 쓰레드 기반 서버가 더 수행시간이 적게 걸리고 동시처리율이 높아진다.

- 2. 워크로드에 따른 분석: Client 요청 타입 (buy, show, sell 등)에 따른 동시 처리율 변화 분석

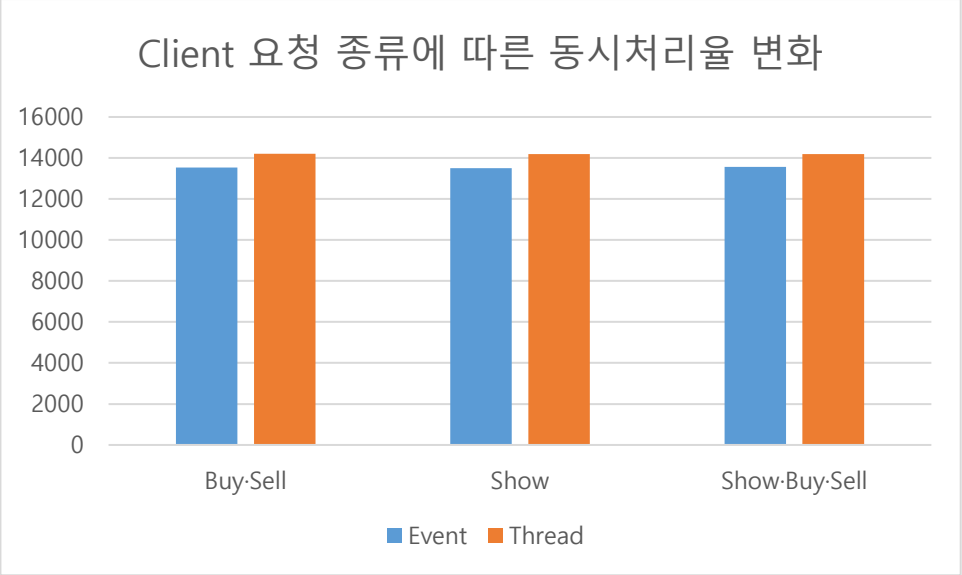
- 클라이언트 수 = 100 그 외 조건 1번과 동일(작업 요청 종류 제외).
- Buy sell만 요청
- Event
- `avg of elapsed  time : 0.739052s`
- Thread

  `avg of elapsed  time : 0.703816s`

- Show만 요청

- Event
- `avg of elapsed time : 0.740561s`
- Thread
- `avg of elapsed time : 0.704527s`

- Show Buy sell 모두 요청
- Event
- `avg of elapsed time : 0.737212s`
- Thread
- `avg of elapsed time : 0.704422s`



분석 : 이벤트의 경우 show만 요청하는 경우 가장 오래 걸렸다. 모든 노드를 순회해야 하기때문에 오래 걸렸을 것이다. 그러다 보니 동시처리율이 가장 낮다. 그러나 STOCK_NUM이 10개로 그렇게 많지 않다 보니 그 차이는 실제로 그렇게 크지는 않다. 쓰레드의 경우도 show만 요청하는 경우가 가장 오래 걸렸다. 마찬가지로 가장 낮은 동시처리율을 가진다.

이론 및 예상과 차이: 두 경우 모두 실험 전 예측과 비슷한 양상을 띄었다. 클라이언트 수가 적을 때는 이벤트 기반 서버가 더 좋은 결과를 냈지만 일정 수준 위로 올라가면 쓰

레드 기반 서버가 더 빨랐다. 이는 상대적으로 오버헤드가 작은 이벤트 기반이 더 빨랐지만 결국 멀티코어 환경을 더 잘 이용할 수 있는 쓰레드 기반 서버가 빨라질 수 있음을 확인할 수 있다. 두 번째 측정 방식의 결과 역시 이론과 상통한다. 모든 노드를 순회해야만 하는 상대적으로 오버헤드가 큰 show가 많을수록 더 오랜 시간이 걸린다. 응답 시간에 기초한 동시처리율을 측정한 실험인 만큼 다른 많은 요소들(ex. 메모리 사용량 등)까지 확인은 하지 못 했지만 충분히 좋은 결과를 얻은 것 같다.