Multicore Programming Project 2

담당 교수 : 박성용 교수님

이름 : 이수빈

학번 : 20200422

1. 개발 목표

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

여러 client와 통신하며 주식 정보 list를 저장하고 판매 및 구매 서비스를 제공하는 concurrent server를 구현한다. client는 서버에 기본적으로 show, buy, sell이라는 명령어를 통해 주식 정보 list를 확인하고 아이템을 사고 파는 동작을 할 수 있으며, 추가적으로 exit이라는 명령어로 주식 시장에서 퇴장할 수 있다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   server는 listenfd를 포함한 array를 가지고 있으며, 어떤 file descriptor에 pending된 input이 있는지 판단한다. 그 pending input을 event라 하며, 즉 event가 발생할 때마다 순차적으로 client의 명령어를 처리해주는 방법이 event-driven approach이다. 만약 listenfd에 event가 발생하면 connection을 accept 후 배열에 새로운 connfd를 추가해준다.

2. Task 2: Thread-based Approach

   server는 thread pool을 미리 형성해놓고, client로부터 connection 요청이 들어오면 이를 accept한 후 thread pool의 thread가 해당 client의 요청을 서비스한다. event-driven approach와 다르게 multi thread가 동작하기 때문에 동시적으로 명령어 처리가 가능하며, synchronization issue를 고려해야 한다.

3. Task 3: Performance Evaluation

   buy, sell과 같이 write 행위만 있는 경우, show와 같이 read 행위만 있는 경우, 모든 명령어가 random하게 섞여 있는 경우의 3가지로 나누어서 event-driven approach와 thread-based approach의 performance를 비교 분석하였다. 각 client 당 10개의 요청을 서비스하였으며, 위 3가지의 경우에서 client의 수를 1, 5, 10, 50, 100, 500, 1000, 2000명으로 증가시켜가며 evaluate해보

았다.

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

    단일 process, 단일 thread를 이용하는 event-driven approach에서 다수의 client I/O 요청을 동시에 효율적으로 처리하기 위한 기술을 I/O Multiplexing 이라고 부른다. select 함수를 사용하면 blocking과 동시에 다수의 socket 상태를 monitoring할 수 있다. monitoring 중인 socket 중 상태 변화가 일어난 socket, 즉 I/O 작업이 준비된 socket의 목록을 반환한다. 이렇게 반환된 socket들에 대해, 하나의 I/O 작업을 수행하는 동안 다른 socket의 작업을 순차적으로 처리한다. 이 과정을 계속 반복하여 다중 client의 요청에 대해 서비스할 수 있는 concurrent server가 이루어진다.

  - ✓ epoll과의 차이점 서술

    I/O Multiplexing을 구현할 때 보통 epoll 또는 select 함수를 사용한다. epoll은 select의 단점을 보완하여 linux 환경에서 사용할 수 있도록 만든 것으로, select 함수보다 대규모 네트워크 어플리케이션에서 많은 클라이언트 요청을 효율적으로 처리할 수 있다. 첫째로, select 함수는 처리할 수 있는 file descriptor의 개수가 최대 1024개로 제한되어 있어, 대체로 수백 개의 처리가 가능한 데 반해 epoll은 수천 개의 file desciptor를 처리할 수 있다. 둘째로, select는 level-triggered 모드로 동작하여 관찰 영역에 포함되는 모든 fd를 순회하며 확인하는데, epoll은 관찰 대상인 fd를 전체 순회하면서 일일이 확인하지 않고, 상태 변화가 발생한 시점에만 일림을 받기 때문에 불필요한 알림을 최소화하고 select보다 개선된 성능으로 동작한다.

- **Task2 (Thread-based Approach with pthread)**
  - ✓ Master Thread의 Connection 관리

    Master thread는 새로운 client와의 연결을 수락하여 worker thread에게 할당하는 역할을 한다. 구체적으로는, server socket을 생성해 새로운 client의

connection을 accept하며, 해당 client에 대한 socket을 생성해 worker thread에게 할당한다. 그 이후에는 다음 connection을 위해 다시 대기 상태로 돌아간다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

앞서 서술했듯, server는 정의된 수의 worker thread pool을 미리 생성하며, master thread는 새로운 connection이 있을 때 이를 worker thread에 분배 및 할당한다. 이후 client에 매칭된 worker thread는 client의 요청을 서비스하고 I/O 작업을 수행한다. 작업이 완료되면 worker thread는 다음 요청을 처리하기 위해 대기 상태로 돌아가며, 다음 요청이 들어오면 재사용된다. 멀티코어 환경에서 이러한 방식은 client의 요청에 대한 동시다발적 처리가 가능하게 해준다.

- **Task3 (Performance Evaluation)**

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

1. 경과 시간 (Elapsed Time)

client의 connection이 이루어지는 시점부터 다수 client의 모든 요청에 대한 서비스가 완료되기까지 걸리는 시간이다. 연결 및 응답 시간은 서버의 성능을 나타내는 가장 대표적인 지표라고 생각했다. 측정은 clock 함수를 이용하였다. server와 client의 connection이 이루어지기 이전부터 측정하기 위해 fork로 client가 생성되기 이전을 start 지점으로 설정하였으며, 모든 요청이 완료되었을 때를 end 지점으로 설정하였다. Configuration 변화에 따른 비교 분석을 위해 (double)(end - start)/CLOCKS_PER_SEC)라는 수식으로 경과 시간을 출력해 확인하였다.

2. 처리량 (Throughput)

요청의 증가에 따른 처리 시간을 측정해 event-driven approach와 thread-based approach의 처리량을 알아보고자 하였다. 서버의 성능과 장단점을 구체적으로 파악하기 위해서는 요청이 적은 경우와 많은 경우를 모두 알아보아야 한다고 판단했다. 이를 측정하기 위해 client의 수를 1명부터 5명, 10명, 50명, 100명, 500명, 1000명, 2000명까지 차례로 늘려가며 테스트하였다.

3. 명령어 종류 (Reader, Writer)

명령어 중 buy와 sell의 경우에는 stock.txt의 내용을 update하는 write 작업이 주가 되고, show의 경우에는 update 작업 필요 없이 read만 하면 되기 때문에 서로 다른 작업에 대하여 각각의 server의 성능이 어떻게 변화하는지 알아볼 필요성이 있다. 따라서 client가 buy, sell만 요청하는 경우, show만 요청하는 경우, 세 명령어를 모두 섞어서 요청하는 경우에 대해 모두 성능을 테스트해보았다.

✓ Configuration 변화에 따른 예상 결과 서술

Thread-based approach가 일반적으로 event-driven approach보다 좋은 성능을 보일 것이다. Event-driven approach의 경우 단일 thread를 이용해 deadlock 또는 synchronization issue 없이 안정적으로 동작하지만 multicore의 이점을 활용할 수 없기 때문이다.

metric에 기반해 예상을 서술해보자면, 위에서 언급한 바와 같이 thread case가 처리량(throughput) 면에서 뛰어나기 때문에 client 수가 늘어남에 따라 event case보다 눈에 띄게 효율이 좋아질 것이라고 예측한다. 또한 명령어 종류에 따라 성능에 차이가 생길 수 있다는 점을 감안해야 한다. Write 작업을 주로 수행하면 event case가, read 작업을 주로 수행하면 thread case가 더 효율적으로 작동할 것이다. 전자의 경우 semaphore를 사용하는 thread case에서는 synchronization overhead issue가 있어 한 번에 하나의 client에만 접근할 수 있기 때문이며, 후자의 경우 반대로 thread case에서 한 번에 여러 client에 접근할 수 있기 때문이다.

C. 개발 방법

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

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

  주식 정보를 담는 item_t 구조체와 **client의 정보를 담는 pool_t 구조체**를 이용해 데이터를 관리할 것이다.

  첫째, item_t 구조체를 관리하는 함수들을 모아놓은 **server_bst.c** 파일을 생성할 것이다. 이 파일에서는 **init_items()** 함수를 통해 item_t 구조체를 initialize할 것이며, item을 binary tree 자료구조에 저장하는 **add_item()** 함수를 통해 stock.txt의 내용을 binary tree로 저장할 것이다. 추가적으로, Buy와 sell 명령어 구현을 위해 id를 가지고 node를 찾는 **find()** 함수와 해당 내용을 stock.txt에 update하기 위한 **update_textfile(), print_fp()** 함수, 프로세스 종료 시 필요한 **free_tree()** 함수도 구현되어야 할 것이다.

  둘째, pool_t 구조체를 관리하는 함수들을 모아놓은 **server_utils.c** 파일을 생성할 것이다. 이 파일에서는 pool_t 구조체를 initialize하는 **init_pool()** 함수, listenfd에 event가 발생했을 때 fd를 추가해주는 **add_client()** 함수, client의 요청을 read하고 서비스하는 **check_clients()** 함수가 기본적으로 있어야 할 것이다. 또한, 그 명령어를 실행해주는 **execute_command(), show(), buy(), sell()** 함수까지 구현되어야 할 것이다.

  셋째, main문에서는 while loop를 통해 client의 connection을 기다리면서 event가 발생 시 listenfd에 pending되어 있으면 add_client()를, 그 외 fd에 pending되어 있으면 check_clients()를 실행하도록 한다. sigint_handler도 구현하여 ctrl-c를 통해 서버를 정상적으로 종료할 수 있도록 할 것이다.

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

  주식 정보를 담는 item_t 구조체와 **client의 정보를 담는 sbuf_t 구조체**를 이용해 데이터를 관리할 것이다.

  첫째, item_t 구조체를 관리하는 함수들을 모아놓은 **server_bst.c** 파일을 생성할 것이며, 해당 파일의 내용은 Task1의 내용과 동일할 것이다. 다만, multi-thread를 운영해야 하기 때문에 update_textfile()과 print_fp()에서 semaphore를 이용하여

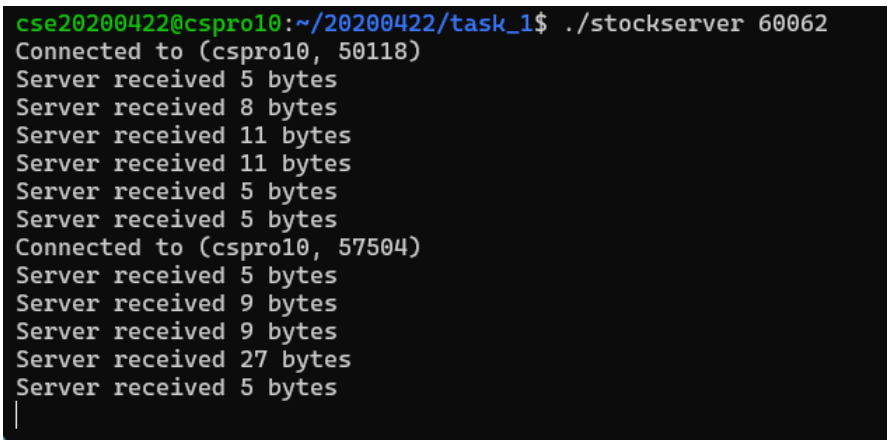
여러 thread가 하나의 file에 write 작업을 수행해야 할 때 하나씩 순차적으로 접근할 수 있도록 해야 할 것이다. print_fp에서는 readers-writers problem과 관련하여 readers를 favor하는 방식으로 구현할 예정이다.

둘째, sbuf_t 구조체를 관리하는 함수들을 모아놓은 **server_utils.c** 파일을 생성할 것이다. 해당 파일에서는 thread가 shared FIFO buffer를 관리할 수 있도록 **sbuf_init(), sbuf_deinit(), sbuf_insert(), sbuf_remove()** 함수들을 구현할 것이며, pthread_create의 세번째 인자로 들어가는 **thread()** 함수에서는 sbuf에 들어온 connfd를 확인하여, 해당 client의 요청을 서비스할 것이다. 명령어를 직접적으로 실행하는 execute_command와 각 명령어에 대한 함수들은 동일하게 구현될 것이며, 이 또한 semaphore에 관한 작업이 추가적으로 이루어져야 할 것이다. buy와 sell에서는 writer's lock과 unlock이, show에서는 reader's lock과 unlock이 수행될 것이다.

셋째, main문에서는 Task1과 동일하게 sigint_handler를 구현할 것이며, 지정 개수만큼 worker thread를 충분히 만들어 둔 후 while문을 통해 exit 명령어 또는 ctrl-c로 종료되기 전까지 client의 connection을 계속하여 수락하고 처리할 것이다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**



```
cse20200422@cspro10:~/20200422/task_1$ ./stockclient 172.30.10.10 60062
show
1 408 1000
2 2300 20000
3 0 1200
4 4790 5000
5 3000 3700
buy 1 8
[buy] success
buy 2 3000
Not enough left stock
sell 3 100
[sell] success
show
1 400 1000
2 2300 20000
3 100 1200
4 4790 5000
5 3000 3700
exit
cse20200422@cspro10:~/20200422/task_1$ ./stockclient 172.30.10.10 60062
show
1 400 1000
2 2300 20000
3 100 1200
4 4790 5000
5 3000 3700
buy 4 90
[buy] success
buy 6 90
Invalid ID
command that doesn't exist
Invalid command
exit
```

실행 결과는 위와 같으며, task_1과 task_2의 결과는 동일하다.

show 명령어는 stock.txt의 리스트를 client에게 동일하게 보여준다. buy 명령어는 client가 해당 리스트의 아이템을 살 수 있도록 하며 성공 시 '[buy] success', 없는 아이템 구매 시도 시 'Invalid ID', 잔여 수량보다 많이 구매 시도 시 'Not enough left stock'이라는 출력문을 띄운다. sell 명령어는 client가 해당 리스트의 아이템을 팔 수 있도록 하며 성공 시 '[sell] success' 문구의 출력문을 띄운다. 존재하지 않는 명령어 입력 시 'Invalid command'라는 출력문을 띄우고, exit 명령어는 client가 주식 장에서 퇴장하도록 한다. 현재 sell 명령어는 성공만을 가정하여 구현했기 때문에 리스트에 없는 아이템 판매 시도 시 'Invalid ID'라는 문구를 출력하는데, 새로운 아이템도 list에 추가해주는 기능을 추가하면 좋을 듯하다. 또한, server와 client의 출력문에 색상 또는 띄어쓰기를 추가하면 가독성을 높일 수 있을 것 같다.

4. 성능 평가 결과 (Task 3)

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

측정 시점은 아래 pseudo code와 같으며, multiclient.c에 clock 함수를 추가하여 측정하였다.

```
uint64_t start = clock();

/* fork for each client process */
while (runprocess < num_client) {
    pids[runprocess] = fork();
    if (pids[runprocess] < 0) return -1;
    else if (pids[runprocess] == 0) {
        // open clientfd
        for (i = 0; i < ORDER_PER_CLIENT; i++) {
            // show
            // buy
            // sell
        }
        // close clientfd
    }
    runprocess++;
}
// waitpid

uint64_t end = clock();
printf("elapsed time: %8.6f seconds\n", (double)(end - start)/CLOCKS_PER_SEC);

```

x축은 client의 수이며, y축은 위 수식의 결과값으로 숫자가 작을수록 처리 속도가 빠름을 의미한다. 한 client 당 10개의 요청을 하도록 설정하여 성능을 테스트했다.

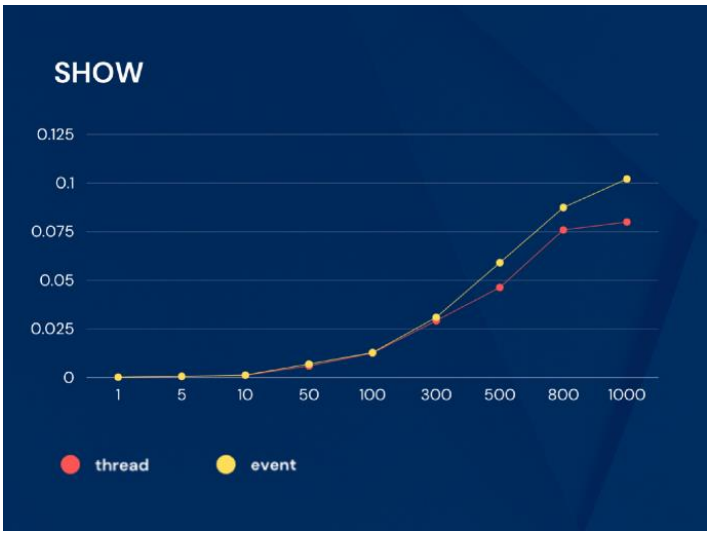
- **Write 작업만 있는 경우**

	thread	event
1	0.000165	0.000168
5	0.000527	0.000658
10	0.001336	0.001166
50	0.006651	0.006964
100	0.013466	0.013071
300	0.033228	0.030823
500	0.049395	0.046285
800	0.078568	0.069574
1000	0.107371	0.089201

Write 작업을 수행하는 buy, sell 요청만 있는 경우에는 thread-based보다 event-driven approach의 성능이 더 뛰어났다. Client 수가 적을 때는 유의미한 차이가

없었으나, client가 많아질수록 그 차이가 도드라졌다. 그 이유는 thread-based approach의 경우 synchronization overhead로 인해, semaphore에 의해 한 번에 하나의 client에만 access할 수 있기 때문이다.

- **Read 작업만 있는 경우**



	thread	event
1	0.000157	0.000183
5	0.000545	0.000642
10	0.001134	0.001247
50	0.006017	0.006976
100	0.012608	0.012835
300	0.029221	0.030927
500	0.046356	0.059034
800	0.075922	0.087436
1000	0.080025	0.102081

Read 작업을 수행하는 show 요청만 있는 경우에는 event-driven보다 thread-based approach의 성능이 더 뛰어났다. Client 수가 적을 때는 유의미한 차이가 없었으나, client가 많아질수록 그 차이가 도드라졌고 1000명일 때 특히 눈에 띄게 드러났다. 이러한 현상은 multicore 활용도가 반영된 결과이다. Event-driven approach의 경우 단일 thread를 이용하기 때문에 multicore 환경에서도 물리적으로 동시에 작업을 수행하지 못하고 순차적으로 진행하는 데 반해, thread-based approach는 다중 thread를 이용하기 때문에 multicore의 이점을 활용할 수 있어 show의 작업이 동시에 이루어질 수 있다.

- **Write와 read 작업이 모두 있는 경우**

	event	thread
1	0.000124	0.000146
5	0.000547	0.000533
10	0.001315	0.001283
50	0.006429	0.006185
100	0.012040	0.011703
500	0.044736	0.049257
1000	0.094493	0.114475
2000	0.173487	0.205187

Client가 1명일 때, 5-100명일 때, 500-2000명일 때로 나눌 수 있다.

먼저, client가 1명인 경우에는 event-driven approach가 효율적이다. 이는 thread-based approach의 경우 synchronization에 의한 overhead가 있기 때문인 것으로 파악된다.

다음으로, 5-100명일 때에는 thread case가, 500-2000명일 때에는 event case가 더 효율적인 것으로 나타났다. 여기에는 몇 가지 변수가 있을 것으로 예상된다. Buy, sell, show 명령어의 개수를 random하게 제시하였기 때문에 결과를 완전히 신뢰할 수는 없을 것이며, 특히 500명 이상의 매우 많은 client가 접속하였을 경우에는 CPU 등 컴퓨터의 성능에 따라 다른 결과가 나타날 수 없을 것이라 예상한다. 단일 thread가 아닌 다중 thread에 의해 작동하는 process이기 때문이다. 따라서, 이러한 변수를 최대한 제하기 위해 client의 수가 5-100명일 때의 결과만을 보았을 때, client 수가 많을수록 thread-based approach가 event-driven approach보다 효율이 더 좋다고 보는 것이 맞다고 판단했으며, 예측과 동일한 결과였다.