Multicore Programming Project 2

담당 교수 : 최재승

이름 : 이우진

학번 : 20181669

1. 개발 목표

본 프로젝트에서는 주식을 사고 파는 기능을 가진 stock server을 구현한다. Client는 stock server에 접속하여 주식의 목록을 보거나 사고 팔 수 있다. 이 때, 사고 파는 과정에서 잔액 또는 주식 잔고를 확인하지는 않는 간단한 형태의 프로그램을 작성한다. 주식 서버를 실행하면 파일로부터 주식의 정보인 id, 남은 개수, 가격을 읽어 들여 binary-search tree 형태로 저장하며, client가 요청을 하게 되면 해당하는 주식 데이터를 찾아 반영한다. Stock server가 종료할 때 update된 정보를 다시 파일에 저장한다. 실제 주식 서버와 같이, 본 프로젝트에서의 주식 서버도 여러 client가 동시에 접속하여 서버에 요청을 할 수 있도록 다양한 방식을 사용하여 concurrent processing을 할 수 있도록 개발한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

Task 1에서는 먼저 stock 관리를 위한 자료구조와 관련 함수를 작성하여 single client에 대하여 기본적인 stock server 역할을 수행하는 프로그램을 작성한다. 이후 Stock Server에서의 concurrent한 작업을 event-driven 방식으로 구현한다. Event-driven 방식을 이용해 stock server는 여러 client의 show, buy, sell 요청에 대한 작업을 하나의 process에서 올바르게 수행하여야 한다.

2. Task 2: Thread-based Approach

Task 2에서는 Task 1에서 구현한 stock 관리 자료구조 및 함수를 이용하며, Stock Server에서의 concurrent한 작업을 thread 기반의 방식으로 구현한다. Thread 기반의 방식을 이용해 여러 client의 show, buy, sell 요청에 대한 작업을 여러 thread을 통해 올바르게 수행하여야 한다. 이를 위해 프로그램은 동시에 access될 수 있는 변수 값에 대해 semaphore을 유지해야 한다.

3. Task 3: Performance Evaluation

Task 1과 Task 2에서 구현한 Stock Server의 성능을 측정하여 Event-driven Approach 기반의 concurrent process 처리에서 어떤 방법으로 구현하였을 때 가장 좋은 성능을 가지게 되는지 확인한다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  ✓ epoll과의 차이점 서술

Task 1에서는 select()을 이용한 event-driven approach을 사용한다. Event-driven Approach는 여러 client들을 server에 동시에 연결하여 처리하기 위한 방법 중 select 함수를 이용한 방법이다. 이 방법에서 server는 현재 접속되어 있는 여러 client들의 connection file descriptor을 저장하고 유지하면서 특정 file descriptor가 읽어 들일 준비가 완료되었을 때 해당하는 file descriptor에서 요청을 읽어 처리한다. 이를 가능하게 해주는 함수가 select() 함수인데, server에서 select() 함수를 호출하면 select() 함수에서는 인자로 넘긴 file descriptor의 목록 중에서 읽어 들일 준비가 완료된 file descriptor가 존재할 때까지 process을 중단한다. 이어 server에서는 select()가 반환되었을 때 저장된 file descriptor 중 읽어 들일 준비가 된 file descriptor을 읽어 서비스를 제공한다. 이 때, select 함수에 전달한 read set이 ready set으로 대체되어 버리기 때문에 read set을 유지하기 위한 작업을 해주어야 한다.

Event-driven Approach의 가장 큰 특징은 multi-client을 하나의 process, 하나의 thread에서 수행한다는 점이다. 이를 통해 Multi-client을 다루는 다른 방식인 process 기반 또는 thread 기반의 방식과 다르게 하나의 context switching이 이루어지지 않기 때문에 수행 속도가 빠르고, race나 dead rock 같은 concurrent programming에서 흔히 발생하는 문제를 신경 쓰지 않아도 된다는 장점이 있다. 하지만, 각각의 client의 정보를 저장하고 유지해야 한다는 점, 그로 인한 구현 난이도가 높아진다는 점 등의 단점 또한 존재한다.

Select()의 가장 큰 문제점은 모니터링 하는 file descriptor의 개수가 1024로 정의된 FD_SETSIZE 까지라는 한계가 있다는 점이다. 이러한 한계로 인해 Linux man page 에서는 모니터링 하는 file descriptor의 개수에 제한이 없는 poll() 또는 epoll API을 대신 사용할 것을 권장하고 있다. Poll()은 select()와 유사하게 동작하는 함수이고, epoll API는 poll()에 몇가지 feature을 추가한 버전으로 명시되어 있다. File descriptor의 개수에 제한이 없다는 차이점에 더해, select()의 side effect였던 전달한 read set이 ready set이 되어버리는 점이 해결되어 매번 read set을 유지하기 위한 작업을 해주지 않아도 된다.

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

  ✓ Master Thread의 Connection 관리

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Task 2에서는 thread을 생성한 방법을 이용하여 multi-client에게 서비스를 제공한다. Thread는 하나의 process 내에서 여러 개가 존재할 수 있으면서 각자의 context, 즉 stack, general register, pc 등에서 실행된다. 여러 process가 실행되는 process 기반의 방식과는 다르게, 하나의 process 내에서 실행되기 때문에 global 변수 등 자원의 공유가 쉽게 이루어진다는 장점이 있다.

  Thread 기반의 stock server을 구현하기 위해, 최초에 실행되는 master thread는 listen file descriptor을 사용해 client와 connection을 생성하고, 생성된 connect file descriptor을 thread에 넘겨주어 각자의 thread에서 해당 client의 요청을 처리하는 방법을 사용한다. 이 때, client가 연결 될 때마다 thread을 생성하지 않고, 프로그램을 시작할 때 여러 개의 thread을 생성한다. 각자의 thread는 P()을 이용해 현재 connect된 client들을 저장하는 sbuf에 client 정보가 저장될 때 마다 client 정보를 가져오고, sbuf로부터 제거한 후 client 요청을 처리한다. 이를 위해 master thread에서는 connection이 이루어질 때 sbuf에 공간이 있는지 확인하여 공간이 있다면 client 정보를 저장하고, 공간이 없다면 공간이 생성될 때까지 대기하여야 한다.

- **Task3 (Performance Evaluation)**

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

  ✓ Configuration 변화에 따른 예상 결과 서술

  Task 3에서는 task 1과 task 2에서 개발한 각각의 프로그램의 성능을 테스트한다. 테스트 방식은 다음과 같다.

  1. 동시에 접속하는 client 단위가 10, 100, 300, 500, 1000 일 때를 각각 분석한다. 이 때 event-driven approach에서 최대 1024개의 client fd을 관리할 수 있기 때문에 더이상 client을 늘려서 테스트 할 수는 없다. Thread 기반의 방식이 context switch을 사용하므로, 더 시간이 오래 걸릴 것으로 예상된다.
  2. Thread base방식에서 1000개의 client에 대한 최적의 thread를 확인한다. 각각 50, 100, 250, 500, 1000을 돌아가며 비교한다. Thread가 많아질 수록

context switch가 많아지므로 느려질 수 있으나, 그만큼 동시에 처리되는 child가 많아지므로 오히려 더 빨라질 수 있을 것으로 보인다.

C. 개발 방법

Stock server 구현을 위한 기본적인 stock data structure과 관련 함수는 stock.h 및 stock.c에 구현하여 task 1과 task 2에서 공통적으로 사용한다. Task 1과 task 2에서의 각기 다른 접근방법에 따른 자료구조 및 함수는 process.h 및 process.c에 구현한다. Stock 및 main은 각자의 접근 방법에 따라 조금씩 변경될 수 있다.

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

Event-driven Approach을 위해 client_t structure와 pool structure을 생성한다.

Client_t을 이용하여 생성한 client structure는 현재 연결 되어 있는 client의 정보를 담는 변수이며, 이를 이용해 현재 연결되어 있는 client들의 linked list을 만들어 유지한다.

pool structure는 현재 연결되어 있는 connected fd의 read set과 client 정보를 저장하고 있으며 read set을 select 함수에 넘기기 위한 ready set 변수를 가진다.

Event-driven Approach을 위한 함수는 다음과 같다.

- Process() : client의 요청을 분석하여 처리하기 위한 함수
- Initpool() : Pool이 listen fd을 가지도록 초기화 하는 함수
- AddClient() : Pool에 client 정보를 추가하는 함수
- CheckClient() : Pool에 저장된 client의 요청을 체크하고, process()을 호출하여 처리하는 함수

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

Thread-based Approach을 위해 sbuf_t structure을 생성한다.

Sbuf_t을 이용하여 생성한 sbuf 변수는 현재 연결 되어 있는 client중 아직 처리되지 않은 client의 정보를 저장하기 위한 buffer을 가지며, queue의 형태를 가진다.

Concurrent한 접속으로 인한 race 문제를 해결하기 위해 sem_t 타입의 mutex 변수를 가지면서 서로 다른 thread가 동시에 buffer에 접근하여 수정할 수 없도록 한다. 또한, 각자의 thread들이 buffer에 남은 client 정보가 있는지 확인하기 위한 sem_t 타입의 items 변수, main에서 buff에 남은 공간이 있는지 확인하기 위한 sem_t 타입의 slots 변수를 가진다.

Thread-based Approach을 위한 함수는 다음과 같다.

- Process() : client의 요청을 분석하여 처리하기 위한 함수

- sbuf_init() : sbuf을 초기화하기 위한 함수
- sbuf_deinit() : 프로그램을 종료할 때 sbuf를 할당 해제 하기 위한 함수
- sbuf_insert() : sbuf에 client 정보를 삽입하는 함수
- sbuf_remove() : sbuf로부터 client 정보를 제거하고 반환하는 함수
- thread() : main에서 호출하는 worker thread 함수

추가적으로, stock data structure에 stock에 대한 reader-writer problem을 해결하기 위한 sem_t 타입의 변수 mutex, w을 추가로 선언하고 stock 정보를 읽거나 변경할 때 동시 접근으로 인한 오류가 발생하지 않도록 한다.

Main에서는 처음 실행할 때 여러 개의 thread을 생성하고, sbuf을 선언하고 client와 connection을 형성하며 client 정보를 유지한다.

- **Task3 (Performance Evaluation)**

Task 3를 수행하기 위해, multiclient.c에 수행 시간을 출력하는 코드를 작성하고 stockserver.c의 configuration을 조금씩 바꿔가며 테스트한다.

3. 구현 결과

Stock 데이터를 처리하기 위한 stock.c와 stock.h을 만들어, task 1과 task 2에서 최소한의 변경 만으로 stock 자료구조를 사용할 수 있도록 구현하였다. task 1과 task 2에서 요구하는 방법을 이용하여 multi client 처리를 처리할 수 있도록 구현하였다.

4. 성능 평가 결과 (Task 3)

1) 동시 접속 client 수에 따른 변화

a) Event-driven approach

```
cse20181669@cspro:~/test/SP_HW2/task1$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10017223 microseconds
cse20181669@cspro:~/test/SP_HW2/task1$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10098315 microseconds
cse20181669@cspro:~/test/SP_HW2/task1$ ./multiclient 172.30.10.11 60021 300
elapsed time: 11106587 microseconds
cse20181669@cspro:~/test/SP_HW2/task1$ ./multiclient 172.30.10.11 60021 500
elapsed time: 13171509 microseconds
cse20181669@cspro:~/test/SP_HW2/task1$ ./multiclient 172.30.10.11 60021 1000
elapsed time: 13196085 microseconds
```

b) Thread-based approach

NTHREAD = 100

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10013189 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10121607 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 500
elapsed time: 50056005 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 1000
elapsed time: 100107100 microseconds
```

수행 결과, 적은 수의 client에서는 두 방법이 별 차이가 없었으나 client의 수가 증가함에 따라 event-driven approach는 수행 시간이 크게 증가하지 않았던 반면 thread-based approach에서는 급격하게 증가하여 client의 수가 500인 경우에는 5배, client의 수가 1000인 경우에는 10배 로 증가하는 모습을 볼 수 있다. 다만 이는 thread의 개수가 너무 적어서 발생한 차이일 수 있으므로, 이후 테스트에서 thread의 개수를 변화하면서 다시 한번 확인해볼 필요가 있다.

2) Thread의 개수에 따른 변화

a) NTHREAD = 10

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10010178 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 100065950 microseconds
```

b) NTHREAD = 100

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10025204 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10048701 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 500
elapsed time: 50054820 microseconds
```

c) NTHREAD = 300

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10013770 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10044285 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 500
elapsed time: 20135925 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 1000
elapsed time: 40067645 microseconds
```

d) NTHREAD = 500

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10024568 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10054720 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 500
elapsed time: 11297269 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 1000
elapsed time: 20420152 microseconds
```

e) NTHREAD = 1000

```
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 10
elapsed time: 10014460 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 100
elapsed time: 10043973 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 500
elapsed time: 13093436 microseconds
cse20181669@cspro:~/test/SP_HW2/task2$ ./multiclient 172.30.10.11 60021 1000
elapsed time: 13253236 microseconds
```

Worker thread의 수가 증가할 수록 처리되는 속도가 빠른 것을 볼 수 있다. Context switch가 많아짐으로 인해 속도가 저하될 것을 생각하였으나, 적어도 thread의 개수가 1000개일 경우 까지는 속도가 꾸준히 증가하는 것을 볼 수 있다. 특히, 성능 평가 1의 event-driven approach와 비교하여, NTHREAD의 개수가 1000일 경우에 수행 속도가 비슷한 것을 볼 수 있는데 NTHREAD가 100, 300, 500인 경우와 함께 살펴 보았을 때 현재 client의 수보다 thread의 수가 같거나 클 때 event-driven approach와 비슷한 수행 속도를 보이는 것으로 생각된다.