Multicore Programming Project 2

담당 교수 : 최재승

이름 : 남현준

학번 : 20181625

1. 개발 목표

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

Multiclient의 connection request를 concurrent하게 대응할 수 있는 서버를 event-based approach와 thread-based approach 방식을 통해 구현하고, 해당 구현의 결과를 비교하여 어느 방식이 효율적으로 접근할 수 있는지 알아볼 것이다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   프로젝트에서 global data 영역에서 선언한 pool 구조체를 활용하여 concurrent server의 구현을 진행하게 된다.

   하나의 반복문에서, 선언된 pool을 순회하여 client가 서버에 요청한 connection request가 있는지 없는지 체크하고, connection request가 있었던 client들에 대해 client들의 요구에 맞춰 해당 요청에 대응되는 서비스를 concurrent하게 처리할 수 있게 된다.

2. Task 2: Thread-based Approach

   Event-driven server와 달리, thread 기반으로 생성되는 pool을 기준으로 하게 된다.

   마찬가지로, 반복문에서 개별 thread를 핸들러를 통해 관리하게 되며, client의 connection request가 있었을 경우, client의 요구사항에 맞춰 서버가 그에 맞는 서비스를 진행하여 client에게 return 하게 된다.

3. Task 3: Performance Evaluation

   Task1과 Task2에서 구현한 event-driven server, thread-based server에서의 평가를 수행해, 어떤 기반의 서버가 조금 더 효율적으로 client request를 처리할

수 있는지 알아볼 수 있을 것이다.

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

Task1에서 select를 통해 구현한 서버에서는 file descriptor의 검사를 통해 I/O를 수행하게 된다. 이를 위해 file descriptor의 설정이 필요하며 프로젝트 구현 코드 상에 적힌 FD_ZERO, FD_SET 함수를 통해 file descriptor의 설정을 진행하게 된다.

Select 함수는 FD_ISSET()을 통해 특정 file descriptor가 어떤 상태인지 확인한 다음, 해당 file descriptor의 현재 상태에 따라 request를 처리하게 된다. 읽기 가능한 descriptor라면 데이터의 수신을 진행하고, 쓰기 가능한 descriptor라면 데이터를 전송하게 된다.

처리가 끝난다면 다시 select를 통해 file descriptor의 상태를 확인한 다음 위의 과정을 반복한다. 이와 같은 방법을 통해 multiclient에서의 connection request를 처리할 수 있게 된다.

✓ epoll과의 차이점 서술

Select는 반복문을 통해 file descriptor set에 접근하게 된다. 이 때, file descriptor set은 비트 배열로 나타낼 수 있고, 해당 배열에서의 값이 변했는지 select를 통해 확인하는 것으로 작동하게 된다. Select의 return 값은 변경된 비트의 개수, 즉 I/O에 대해 준비된 fd의 개수를 반환하게 되고 에러가 발생할 시 -1을 return하게 된다.

하지만 select는 file descriptor set에 변형을 가하기 때문에, 매번 file descriptor의 check를 시작하며 비트 배열을 새로 설정해주어야 한다. 또한, file descriptor를 하나씩 체크하기 때문에 수행시간면에서 단점이 존

재하고, fd의 길이 또한 제한되어있다.

Epoll은 select의 단점을 보완하기 위해 kernel-level에서의 multiplexing이 가능하다. 따라서, 모든 fd를 탐색할 필요가 없고 특정 범위에 대해서만 OS에 요청하는 것을 통해 fd값을 반환받게 된다. Epoll_create를 통해 epoll 구조체를 생성하고, epoll_wait를 통해 file descriptor에 생기는 변화를 확인하게 된다.

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

  ✓ Master Thread의 Connection 관리

  Client의 connection request가 추가될 때마다 서버는 pthread_create를 이용하여 해당 request를 담당하는 thread를 생성하게 될 것이다. Thread를 정의하며 해당 thread를 handle할 때 사용할 함수를 정하게 되는데, client의 request에 맞게 작동할 수 있도록 한다.

  Thread handler에서 pthread_detach(pthread_self())를 통해 thread를 free 시킴을 통해 race가 일어나는 것을 방지하고 정의된 로직에 따라 file descriptor에 변화가 생기고 request에 반응할 것이다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Main()함수에서 thread를 pthread_create()를 통해 생성한다. 생성된 thread는 pthread_detach()를 이용해서 개별적으로 작동하는 것이 가능하게 함과 동시에 해당 thread가 종료되면 kernel이 thread에 사용한 리소스를 reap하는 것이 가능하도록 알린다.

  Connection이 종료되었다면 return NULL (혹은 pthread_exit())을 통해 해당 thread가 종료됐다는 것을 알려 reaping이 진행되도록 한다.

- **Task3 (Performance Evaluation)**

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

  각 approach에서의 수행 완료까지 걸리는 시간을 측정하여, task1과 task2의 구현 간의 차이를 관찰하게 될 것이다.

  client에서 요청하는 connection request의 수가 많아질수록 처리해야

하는 file descriptor의 수도 늘어나게 될 것이고, 이에 따라 구현방식에 따라 처리 속도에도 변화가 생기게 될 것이다. 따라서, 해당 변화를 찾아낼 수 있다면 두 approach 간의 차이를 관찰하여 유의미한 추론을 할 수 있게 될 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

Task1 의 event-based approach 에서는 매 반복마다 file descriptor set 의 비트 배열을 재정의 해야하고, 읽어들일 수 있는 fd 의 값에 제한이 있고 pool 구조체로 정의된 데이터를 모두 탐색하여 결과를 얻게 된다. 따라서, 각 thread 에서 처리하게 되는 thread-based approach 에서의 수행시간 보다 event-based approach 에서의 수행시간이, client 의 수에 따라 직접적으로 영향을 받게 될 것 같다.

C. 개발 방법

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

해당 프로젝트에서는 강의자료 및 프로젝트 제공 자료에서 제공된 semaphor와 노드에 들어갈 구조 샘플을 활용하여 구현을 진행했다.

주식의 정보를 저장할 수 있는 item 구조체를 선언하여 주식과 관련된 내용, 그리고 mutually exclusive하게 처리될 수 있도록 sem_t mutex를 item 구조체 안에 선언해주었다.

Event-based approach와 thread-based approach 모두 공통적으로 프로젝트 요구사항에 binary search tree를 이용해서 stock 정보를 저장한다고 명시되었다. 따라서, binary search tree의 구현을 위해 구조체를 정의할 필요가 있었다.

해당 구현을 위해 tree의 node 구현이 필요했고 앞서 정의한 item과 tree의 노드에 필요한 left, right child의 저장을 위해 left와 right를 node 포인터형으로 선언하여 노드의 추가가 가능하도록 하였다.

아래의 내용부터는 각 approach에 따라 추가된 요소들에 대해 서술한다.

◆ Event-based approach

강의자료에서 제공된 정보를 활용하여 pool 구조체를 선언하였다. 반복문에서 해당 pool을 탐색하여 어떤 clientfd가 준비될 수 있는지 확인한 후, 해당 부분을 반영하게 된다.

이후 check_client() 함수에서 준비된 file descriptor들에 대해 connection request를 수행하고 요구사항에 맞게 작동되어 나온 결과물을 return하게 된다.

◆ Thread-based approach

Thread를 handle하는 logic을 추가하여 thread에서 request를 처리할 수 있게 구현한다. 새 client의 연결이 들어오게 될 때마다 pthread_create()를 통해 새로운 thread를 생성하게 된다.

Thread handler에 로직에 detach하는 기능을 넣어서 thread가 개별 client의 request 대응 할 수 있도록 만들 것이다.

3. 구현 결과

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

Event based approach에서는 강의자료에서 제공된 pool 구조체를 활용해서 file descriptor를 관리하게 되었다. Pool을 순회하며 select()와 FD_ISSET()을 통해 준비된 file descriptor를 발견하고 concurrent하게 해당 connection request를 처리할 수 있게 되었다.

Thread-based approach에서는 각 client의 요청을 처리할 수 있도록 개별 thread를 pthread_create()를 통해 생성해주고 pthread_detach()를 통해 개별 thread를 통해 각 client들의 connection request를 concurrent하게 처리할 수 있게 되었다.

4. 성능 평가 결과 (Task 3)

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

Task 1 – event-based approach

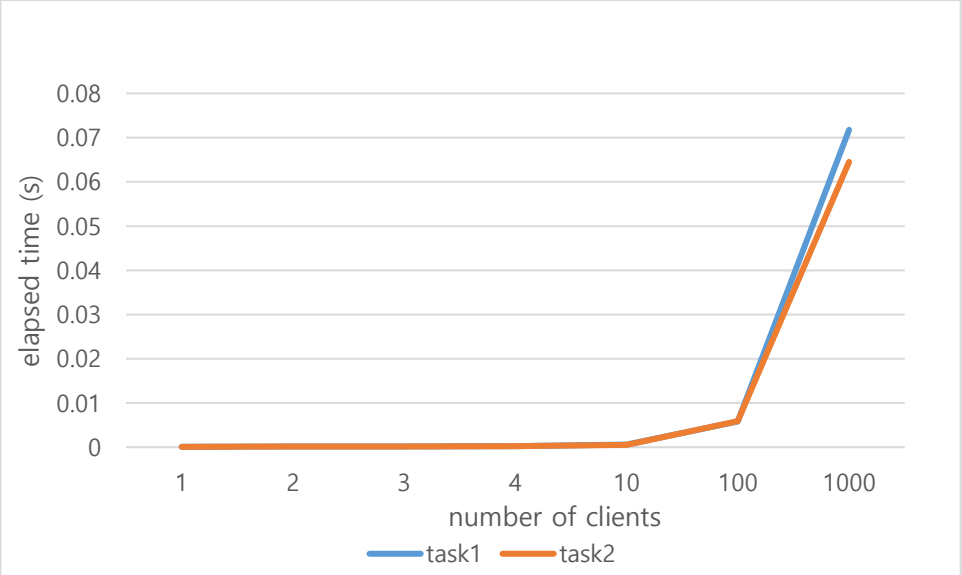
```
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 1
Elapsed time : 0.000097s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 2
Elapsed time : 0.000147s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 3
Elapsed time : 0.000203s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 4
Elapsed time : 0.000277s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 10
Elapsed time : 0.000588s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 100
Elapsed time : 0.005801s
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 1000
Elapsed time : 0.071758s
```

Task 2

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 1
Elapsed time : 0.000088s
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 2
Elapsed time : 0.000154s
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 3
Elapsed time : 0.000197s
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 4
Elapsed time : 0.000265s
```

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 10

Elapsed time : 0.000558s

cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 100

Elapsed time : 0.005860s

cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 1000

Elapsed time : 0.064515s
```

	1	2	3	4	10	100	1000
task1	0.000097	0.000147	0.000203	0.000277	0.000588	0.005801	0.071758
task2	0.000088	0.000154	0.000197	0.000265	0.000558	0.00586	0.064515



해당 표는 task1과 task2의 구현을 multiclient로 실행했을 때, elapsed time을 그래프로 나타낸 것이다

해당 결과는 multiclient.c 파일의 macro 부분에서 MAX_CLIENT = 100, ORDER_PER_CLIENT = 10, STOCK_NUM = 10, BUY_SELL_MAX 10인 상태에서의 elapsed time이다.

일반적으로, task2(thread-based approach)가 task1(event-based approach)보다 빠르게 client들의 connection request를 처리함을 알 수 있다. Connection

request의 수가 늘어남에 따라, event-based approach에서 pool의 순회에서 fd의 재설정 횟수가 늘어남에 따라 multiclient의 실행 시간이 늘어나는 것으로 보인다.

- 워크로드에 따른 분석

i) 모든 client가 show를 요청할 때

A. Task 1

```
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 100
Elapsed time : 0.006642s
Elapsed time : 0.006079s
Elapsed time : 0.006241s
```

B. Task 2

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 100
Elapsed time : 0.005947s
Elapsed time : 0.006177s
Elapsed time : 0.006080s
```

ii) 모든 client가 buy를 요청할 때

A. Task 1

```
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 100
Elapsed time : 0.006023s
Elapsed time : 0.005886s
Elapsed time : 0.005953s
```

B. Task 2

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 100
Elapsed time : 0.005860s
```

```
Elapsed time : 0.006703s
```

```
Elapsed time : 0.005974s
```

iii) 모든 client가 sell을 요청할 때

A. Task 1

```
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 100
```

```
Elapsed time : 0.006088s
```

```
Elapsed time : 0.005958s
```

```
Elapsed time : 0.005773s
```

B. Task 2

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 100
```

```
Elapsed time : 0.005875s
```

```
Elapsed time : 0.005839s
```

```
Elapsed time : 0.005829s
```

iv) 모든 client가 buy or sell만 요청할 때

A. Task 1

```
cse20181625@cspro:~/proj2/task1$ ./multiclient 172.30.10.9 60018 100
```

```
Elapsed time : 0.006190s
```

```
Elapsed time : 0.006202s
```

```
Elapsed time : 0.006464s
```

B. Task 2

```
cse20181625@cspro:~/proj2/task2$ ./multiclient 172.30.10.9 60018 100
```

```
Elapsed time : 0.006220s
```

```
Elapsed time : 0.005853s
```

```
Elapsed time : 0.005962s
```

다른 설정들은 건드리지 않고, option에서 특정 option만 선택되도록 하고 난 다음 multiclient를 통해 100개의 client에 대해 명령어를 처리했을 때의 결과이다.

일반적으로, event-based approach보다 thread-based approach의 표본의 개수가 적어서 100% 확실하다고는 할 수 없지만 성능이 더 좋은 것을 확인할 수 있다. 이는 앞서 예측한 thread-based approach가 event based-approach보다 실행시간이 빠를 것이라는 예측에 부합하는 결과였다.

종종 multiclient의 실행시간이 길게 잡히는 경우가 종종 보이는데 이는 프로그램의 구현 상, mutex의 구조적 요인으로 생긴 결과라고 판단된다. 프로그램의 구현 상 mutex에 sem_wait()를 통해 lock이 가해지면 sem_post()를 통해 unlock이 진행될 때까지 request가 pending되고 이에 따라 수행하는 과정에서 시간 지연이 생길 수 있기 때문이다.

특이사항으로는 multiclient 파일을 실행할 때, client의 수를 늘릴수록 event-based approach에서의 execution time이 늘어나는 것을 확인할 수 있었지만, 특정 command만 단일로 실행되는 경우에는, client의 수에 따른 event-based approach와 thread-based approach에서의 execution time이 큰 차이를 보이지 않는다는 점이 있다. 해당 부분은, 실험의 시행횟수가 적어서 생긴 실험적 오차일 수도 있고, multiclient의 execution time을 측정할 때 사용한 시간 측정 함수의 구현에 따라서 생기는 오차일 수도 있다는 추측을 할 수 있을 것이다.

Buy or sell의 execution time을 보여주는 iv 결과의 경우, 표본의 수가 많지는 않지만 thread에 overhead가 생겨서 execution time이 늦어지는 경우가 아니라면, thread-based approach가 유의미하게 event-based approach보다 빠르게 수행될 수 있는 경향을 띄운다는 것을 알 수 있다.

따라서, 종합적으로 수행시간을 기반으로 결론을 내린다면 mutex의 lock으로 인해 생기는 시간 지연과 같은 문제가 없다면 thread 기반의 서버가 concurrent한 connection request를 처리하는 데에 이점을 가지게 될 것이다.