**Multicore Programming Project 2**

담당 교수 : 최재승 교수님

이름 : 유종선

학번 : 20191611

1. **개발 목표**

: 주식 정보를 저장하고 있는 server를 구현하고, 주식 정보에 접근하는 복수의 client들이 주식을 concurrent 하게 사고 팔 수 있는 stock server를 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Task 1: Event-driven Approach

: 각각의 client는 fd를 trigger한다. 복수개의 connfd가 생기는 것이다. 이때 select 함수를 사용하여 pending된 input을 실행할 descriptor를 결정한다. 이렇게 multiclient의 명령들을 수행한다.

1. Task 2: Thread-based Approach

: master thread가 먼저 복수의 thread를 생성시킨 후에, 그 중에서 worker threads들을 지정하여 각각의 worker thread들은 multiclient의 명령을 수행한다.

1. Task 3: Performance Evaluation

: gettimeofday 함수를 사용하여 Elapse time을 측정하였다. Multiclient에서 생성된 명령어를 모두 실행하고 나면 stockserver 쪽에 elapse time이 출력된다. 다양한 상황을 설정해 두고, 시간을 측정하였다.

* 1. **개발 내용**
* **Task1 (Event-driven Approach with select())**
  + Multi-client 요청에 따른 I/O Multiplexing 설명
  + epoll과의 차이점 서술
* single process로 server는 connection request와 사용자가 타이핑하는 명령어 2가지를 모두 수행해야한다. 서버는 client로부터 connection request가 발생하면 fd를 형성하여 connfd의 array에 삽입한다. Select 함수를 사용하여 array에 저장된 connfd를 돌면서 pending input이 있는지 찾아다닌다. Pending input을 찾게 되면 새로운 connfd를 삽입해준다.
* Epoll 함수는 select와 같은 역할을 수행하는 함수인데, epoll은 select에 비해 더 많은 기능을 가지고 있다. Select가 최대 file descriptor에 제한이 있는 반면, 더 많은 수의 file descriptor를 처리할 수 있다. 또한, 성능적인 면에서도 뛰어나다. Select와 비교했을 때, 더 효율적인 event handling 구조를 지니고 있어 overhead가 비교적 덜 발생하게 된다. Select는 epoll과 비교했을때, 더 portable 하다는 장점이 있다. Select는 epoll과 달리, linux에 국한되지 않는다.
* **Task2 (Thread-based Approach with pthread)**
  + Master Thread의 Connection 관리
  + Worker Thread Pool 관리하는 부분에 대해 서술
* Master thread(= main thread)는 먼저 worker thread들을 생성해놓고, client로부터 받은 요청을 수행하기 위해 그 worker thread들 중에서 일부를 사용하여 명령어를 수행한다. Client로부터 connection request를 받으면, connfd를 생성해서 buffer에 넣어준다. 이 buffer는 shared buffer로, 이를 사용하여 connecton을 관리한다.
* Main thread가 connfd를 생성받게 되면, sbuf\_insert 함수를 사용하여 Connfd를 item으로 하여 buffer에 넣어준다. 앞서서 connfd를 생성받기도 전에 생성된 buffer에 item에 들어오면, V(&sp->items)를 실행하여 available 한 Item이 있음을 알린다. 이를 worker thread들이 기다리고 있다가 해당 item(connfd)을 사용하여 client의 명령어를 수행한다. 명령어를 수행하고 나면, close(connfd)를 해준 뒤에 다시 다음 실행을 기다리게 된다.
* **Task3 (Performance Evaluation)**
  + 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술
  + Configuration 변화에 따른 예상 결과 서술
* 몇가지 기준에서 Performace를 평가하고자한다.
  + 단일 client에서 명령어의 숫자를 기준으로 했을 경우
    - Ex. command 갯수 : 5, 10, 50, 100…
  + 명령어의 숫자는 고정되어 있고, client의 갯수를 늘렸을 경우
    - Ex. command 갯수 100, client : 1, 5, 10, 100
  + Reader 명령어(show)인 경우와 writer 명령어(buy, sell)의 실행 시간 차
    - Reader 명령어만 실행하는 경우, writer 명령어만 실행하는 경우, mix
* 같은 조건에서 thread server와 event-based server의 시간 측정
* Configuration을 수정하여 다음과 같은 실험 수행. 실험에는 gettimeofday 함수를 사용한다. Service 부분의 시작과 끝에 gettimeofday를 선언하여 실행시간을 측정한다. 데이터의 단위가 작아서 초단위에서는 변화를 잡기 어려울 것으로 예상되므로 밀리세컨드 단위로 실험을 수행하였음.
* 정렬 알고리즘이 그렇듯이 데이터의 크기가 작을 경우에는 큰 차이가 나지 않을 것으로 예상됨. Thread의 갯수도 마찬가지임. 하지만, 데이터의 크기가 커졌을 경우에 얼마나 변화가 생기는 지 확인할 필요가 있음. Thread는 병렬 연산이 가능하기 때문에, 작은 데이터보다 큰 데이터에서 더 큰 영향이 있을 것으로 추정됨.
* Reader와 writer의 경우에는 reader는 데이터의 변화가 없기 때문에, writer에 비해 더 빠른 수행시간을 가질 것으로 예측됨.
  1. **개발 방법**
* **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

**: 기본적으로 event-based와 thread 모두, 효율성을 위해서 data는 이진트리를 사용하여 구현하였다. 이는 data의 크기가 커지는 경우, 더 큰 효과를 낼 수 있다. 이 이진트리를 탐색, 수정, 삽입 하기 위해서, 노트 포인터를 선언하였다. 이를 위해 선언된 함수는 Search, new, insert\_node이다.**

* Event-based

: main에서 connfd가 accept되면, 먼저 add\_client함수에 들어가서, 이용가능한 slot을 찾아서 connfd를 pool에 추가해준다. 이 pool은 connected descriptor들을 표현하기 위해서 선언한 구조체이다. 그러고 나서, check\_clients(&pool)을 실행하여 client의 명령어를 실행한다. Check\_clients에서는 어떠한 명령어가 입력되었는지 flag를 선언하고, 그에 맞는 명령어를 실행한다. Exit이 들어오는 경우에는 아무것도 하지않고 Break하여 반복문 밖으로 나간다.

* Thread

: main에서 처음 실행할 때, 미리 Pthread\_create 함수를 사용하여 thread를 생성해 놓는다. Client로부터 request를 받아서 생성된 connfd를 삽입하는 함수 sbuf\_insert에서 P(&sp->mutex) 와 V(&sp->mutex) 사이에 삽입하면서 synchronizing 시켜준다. 명령어 실행 함수인 echo\_cnt도 마찬가지로, P(&sp->mutex)와 V(&sp->mutex)사이에 실행하면서 error의 발생 위험도를 줄인다.

* Performace evaluation

: gettimeofday를 사용하여 start, end 포인트를 잡고, 명령어 실행이 전부 끝나면 차이를 프린트 해준다. 단위는 밀리세컨드이다.

1. **구현 결과**

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

: select함수가 계속해서 array를 돌면서 pending input을 찾아다니고, 찾으면 해당 connfd를 사용하여 명령어를 수행하는 것을 계속해서 반복한다. Flag를 통해서 마지막 명령어를 수행했음을 인지하면, 변경된 내용을 다시 파일에 써주는 작업을 하고 connfd를 close해준다.

* Thread

: semaphore인 Mutex를 사용하면, reader와 writer가 data를 접근하는 과정에서 충돌이 일어나지 않게된다. 실행시간은 더 걸리게 되지만, exclusive한 접근 덕분에 사용자가 원하는 결과 값을 얻을 수 있게 된다.

Item을 버퍼로부터 제거할 때는 sbuf\_remove함수를 사용하는데, 이는 sbuf\_insert함수와 매우 유사하지만 P와 V의 순서만이 약간의 차이가 있다.

Thread들은 입력을 기다리고 있다가 available한 item이 있다는 것을 인식하면 connfd를 통해 명령어를 수행하고 connfd를 닫고 다시 기다리고 다시 수행하고를 반복하면서 client의 명령어를 수행하게 된다.

1. **성능 평가 결과 (Task 3)**

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

1. Client = 1, order\_per\_count = 5, 10 , 50, 100

이 경우에는 단일 client라서 그런지 몰라도, 두 방법의 차이가 크지 않았다. 숫자는 크지만, 밀리세컨즈이므로 실제 체감 속도는 거의 차이가 없다고 봐야할 것이다.

1. Order\_per\_count = 5, client = 5, 10, 50, 100

Order\_per\_count는 고정시키고, client의 값을 변경시켜보았는데, 확실히 client가 늘어나는 비율보다 시간이 늘어나는 비율이 현저히 작은 것을 확인할 수 있다. Client는 5에서 100까지 약 20배 가량 늘어난 것에 반해, 시간은 단 1.5배도 늘어나지 않았다.

1. Order\_per\_count = 5, client = 5, 10, 50, 100
   1. Event-based

위의 그래프를 보면, buy-sell 보다 show에서 조금 더 많은 시간이 걸리는 것을 확인할 수 있다. Show는 reader이고, buy,sell 은 writer로 정보를 변경하는 것 까지 작업을 수행하기 때문에 더 많은 시간이 걸리는 것이 아닐까 추측해볼 수 있겠다. 하지만, 만약 트리의 구조가 커지고, print함수를 더 많이 사용하게 되면, 이것 또한 시스템 시간에 영향을 줄 수 있을 것 같기도 하다.

* 1. Thread