**System Programming Project 2**

담당 교수 : 김영재

이름 : 김현우

학번 : 20171628

1. **개발 목표**

Thread based와 event based 방식으로 client 의 요청을 처리해주는 서버를 만든다. 각각 서버를 구현함으로써 여러 client로부터오는 동시다발적인 입력을 concurrent하게 처리하는 것이 목적이다. 서버의 모든 구조는 동일하나 client 로부터 들어오는 신호를 받는 과정에서 thread를 사용해서 descriptor를 처리하는 부분에서 하나의 process안에서 루프를 돌면서 descriptor를 처리할 하는 부분에서 각각 차이가 난다.

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

Event-based 방식의 서버가 어떤 식으로 동작하고, 요청을 처리하는지 알 수 있게 된다.

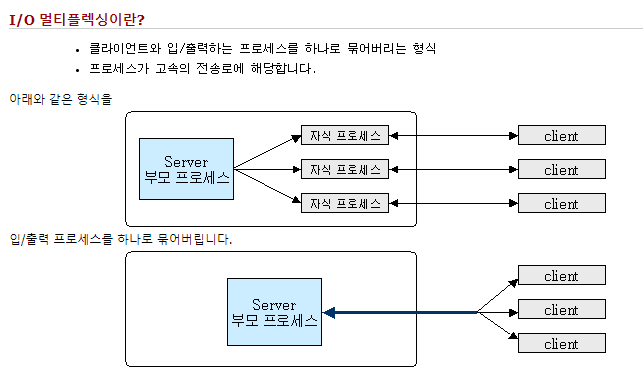
1. Task 2: Thread-based Approach

Thread-based 방식의 서버가 어떤 방식으로 동작하고, 요청을 처리하는지 알 수 있게 된다.

1. Task 3: Performance Evaluation

성능 측정을 실시하면서 event-based 방식과 thread-based 방식의 성능차이를 확인하고, 많은 수의 입력이 들어올 때 성능의 변화를 측정 한다.

* 1. **개발 내용**
* **Task1 (Event-driven Approach with select())**
  + Multi-client 요청에 따른 I/O Multiplexing 설명



여러 client 에서 온 입력을 자식 프로세스나, 스레드로 입출력을 받지 않고, 단일 프로세스 안에서 입력을 받는 방식이다. 이러한 방식을 구현하기 위해서 단일 프로세스에서 여러 개의 파일 디스크립터를 열어놓고 해당 디스크립터가 변했는지 check를 해서 변했다면 입력이 들어온 것으로 가정하는 방식이다. 이 때 반복문을 써서 일일히 디스크립터의 변화를 확인한다.

* + epoll과의 차이점 서술

epoll은 위의 multiplexing 과 달리 for문을 전부 돌지 않고도, 변화가 발생한 파일 디스크립터를 받는게 가능하다.

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

Master thread에서는 NTHREAD 만큼의 thread를 만든 이후에 연결을 할 client의 파일 디스크립터를 받아주고 저장해준다. 이후 각각의 자식 스레드에서는 main thread에서 만든 디스크립터를 받아서 client와 통신을 한다.

* + Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread pool은 master thread에서 가져온 descriptor를 가지고 작업을 하다가 연결이 종료되면 해당 descriptor를 종료한다.

* **Task3 (Performance Evaluation)**
  + 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

이번 실험에서 성능의 지표는 일정 시간 동안 얼마나 서버가 클라이언트의 입력을 동시에 처리할 수 있는 지 여부를 가린다. 그러므로 1초당 서버가 몇개의 클라이언트의 명령을 처리할 수 있는지로 metric을 정했다. 측정 방식은 multiclient.c를 작동시켜서 명령어를 stockserver로 보내서 응답을 모두 받으면 측정을 중단한다. 실험의 변수가 있을 수도 있기 때문에 여러 번 측정을 한 다음에 평균을 내는 방식으로 진행한다.

* + Configuration 변화에 따른 예상 결과 서술

1. 처음에는 연결된 client 와 client 당 명령어의 개수가 아주 처음에는 증가함에 따라 처리량이 증가하나, 일정 부하가 넘어서면 급격하게 줄어 들 것이라고 예상함
2. Thread 같은 경우 당연한 얘기일지도 모르겠지만, thread가 기본적으로 많을수록 작업 효율이 늘어난다. 하지만 스레드가 지나치게 많이 생성되면 이로 인한 overhead가 더 커져서 효율이 떨어지는 순간이 온다.
3. 명령어에 따른 client의 처리는 show 명령어를 사용했을 때 더 처리량이 낮을 것이라고 예상했다. 왜냐하면 show명령여를 사용하면 트리를 전부 순회를 해야 하므로 처리율에서 좋지 않다고 생각했다. 또한 가장 빠른 경우는 sell 과 buy만을 실행하는 경우라고 생각했다. 이유는 show명령어가 없기 때문에 순회를 하지 않아도 되는 장점이 있다고 생각했기 때문이다.
4. Taskset을 통한 cpu 지정

Event 의 경우는 크게 상관 없지만 thread 같은 경우는 multithreading 의 장점을 살리려면 multicore가 반드시 필요하다. 이 때 taskset을 통해서 thread가 작동하는 cpu를 제한한다면 thread based server의 성능이 저하 될 것이다.

1. Event 와 thread based 의 성능 차이

만약에 cpu에 제한을 두지 않는다면 thread가 멀티 코어의 환경상 초당 처리율이 높을 것이고. Cpu를 1개에서만 돌리게 되면 thread based server 의 thread를 생성할 때 overhead로 인해 event based server 의 성능이 더 높을 것이라고 예상한다.

* 1. **개발 방법**

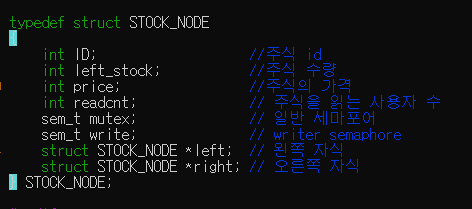
thread base 와 event based의 공통적인 부분을 설명하고 마지막으로 thread와 event 의 부분을 각각 설명한다.

1. 파일 입력 받기

main함수에서 read\_file 함수를 소환해서 파일을 입력 받는다. Read\_file 함수 내부에서는 일단 한번 파일을 전체적으로 읽어서 주식의 개수를 파악한 다음에 파일을 닫고, 해당 lines 변수 만큼 STOCK\_NODE\* temp\_array라는 배열을 동적할당한 이후에 read\_file 함수를 종료한다.(이 함수는 file\_manage.c 에 있다)

1. 이진 트리 만들기

주식을 처리하기 위해 이진트리를 만들어야 하는데 구조체는 아래와 같다.



(server 가 event based 일 때는 sem\_t 타입의 변수는 사용되지 않는다)

main 함수에서

void binary\_tree\_init() 함수(binary\_tree.c 에 존재)를 호출한다. 이후에 binary\_tree\_init함수 내부에서 앞서 만든 temp\_array를 qsort를 사용해서 정렬을 한 이후에 STOCK\_NODE \* binary\_tree\_build 함수(binary\_tree.c)를 호출한다. 여기서 tree\_head = binary\_tree\_build를 함으로써 트리의 root노드를 기록한다.

Binary\_tree\_build

텍스트이(가) 표시된 사진

자동 생성된 설명

정렬된 배열을 활용해서 맨 중간에 있는 값을 ROOT 노드로 활용해서 이진 트리의 균형을 맞췄다. 그러므로 왼쪽 자식과 오른쪽 자식은 자신과 각각 오른쪽 끝과 왼쪽 끝의 중간에 있는 원소가 왼쪽 자식이 된다.

Binary tree 내부에서는 새로운 한 개의 node를 동적할당을 한다. 동적 할당 이후에 동적 할당한 메모리 공간에 temp\_array의 index가 mid인 주소에 있는 STOCK\_NODE의 정보를 새로운 노드에 값을 넣어준다. 이후에 thread\_based server 같은 경우는 semaphore를 init해준다.(event는 그럴 필요 x)

값을 저장하는 과정이 끝났으면 자신의 오른쪽과 왼쪽 자식이 될 주소값을 binary\_tree\_build를 재귀적으로 실행하면서 받는다. 이 때 왼쪽 자식은 자신을 기준으로 왼쪽 끝과 자신의 중간에 있는 원소가 된다. 오른쪽도 자신을 기준으로 오른쪽 끝과 자신의 중간에 있는 원소가 오른쪽 자식이 된다. 재귀적으로 다음 함수로 넘어간 이후에 start가 end보다 커진 경우는 NULL을 리턴해준다.(해당하는 자식이 없다는 의미).

1. 명령어 처리

뒤에 후술할 메시지를 server가 client로부터 받고 난 이후에 받은 command를 처리하는 과정이다. 명령어를 성공적으로 수신하고 나면 command 함수를 호출한다. Command 함수를 호출한 이후에 clientBuf(client로 보내줄 신호)를 0으로 초기화한다. buf(client가 server로 보내준 입력) 을 BUF2에 복사를 한다. 이후에 argvFeed함수(argv.c)를 호출해서 char\* argv[]에 단어의 시작 포인터를 저장한다. 이후에 argv배열의 개수를 세면서 argc를 늘려주어 인자의 개수를 세준다.

텍스트, 화면이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

* + 1. Show

입력으로 show 명령어가 들어오면 인자의 개수가 1개인지 확인하고, 맞다면 show\_binary\_tree(binary\_tree.c)함수를 실행한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Event based 서버에 있는 show\_binary\_tree

텍스트이(가) 표시된 사진

자동 생성된 설명

Thread based server 에 있는 show\_binary\_tree

Show\_binary\_tree는 트리를 재귀적으로 순회하면서 자기에 해당하는 노드를 parameter로 받은 clientBuf에 넣어준다. 만약에 현재의 node 가 null이라면 순회를 중단한다. (event\_based같은 경우는 현재 순회하는 노드에 lock 을 걸지 않아도 상관 없지만, thread\_based server 는 현재 순회하는 node 에 lock을 걸어준다)

* + 1. Buy

Buy 명령어 에서는 처음으로 들어온 명령어가 buy인지 확인하고 만약에 buy 명령어가 들어온 것이 맞고, 명령어 인자의 개수도 일치한다면 binary\_tree\_search(binary\_tree.c)를 통해서 인자로 들어온 id를 가진 주식을 찾는다.

Binary\_tree\_search함수는 이진 트리를 탐색한다. 만약에 현재 찾는 값이

현재 노드의 값보다 크다면 오른쪽 자식으로, 작다면 왼쪽 자식으로, 같다면 해당 노드를 반환한다.

Binary\_tree\_search함수를 통해 해당 노드를 받았으면 명령어를 통해서 받은 주식 거래량인 amount와 비교를 해서 해당 주식에 존재하는 재고가 amount보다 작으면 거래를 하지 않는다. 만약에 amount가 더 작다면 현재 재고에서 amount를 뺀다. 그리고 각각의 상황에 해당하는 메시지를 clientBuf에 넣어준다.

* + 1. Sell

Sell 명령어도 buy 명령어와 동일하게 binary\_search\_tree에서 노드 정보를 가져오고 현재 존재하는 주식 수량 + amount를 해준다.

4. 종료 및 이진 트리 파일에 기록하기

Thread\_based와 event\_based 서버 둘 다 연결을 종료하면서 몇 개의 client가 연결 되어있는지 clientNumber를 통해 확인하고 만약에 clientNumber가 0 이 되면 save\_binary\_tree(file\_manage.c)함수를 실행한다. Save\_binary\_tree는 노드를 받아서 현재 노드를 기준으로 왼쪽 자식과 오른쪽 자식을 재귀적으로 순회하면서 file에 write(recursive\_write)를 해준다.

5. 각각의 서버에서 client 와 server 의 연결 성립 및 종료 과정

(1) event\_based server

Event\_based 서버는 main함수에서 while 루프를 돌면서

텍스트이(가) 표시된 사진

자동 생성된 설명

Pool 구조체에 ready\_set과 read\_set을 넣어주고, select 함수를 통해 신호를 받아준다. Select 함수를 통해 신호를 다 받고 나면 fd+\_issset으로 ready\_set에서 이미 연결이 되어있는 드스크립터는 없에주고, Accept함수를 호출해서 연결을 기다린다. Accept 함수에서 연결이 도착하면 add\_client 함수로 가서 파일 디스크립터를 순회하면서 새로 추가된 디스크립터가 있는지 체크한다. 만약에 새로운 client가 연결이 되어있다면, clientNumber를 늘려서 연결된 클라이언트의 개수를 늘려주고, 연결된 client에 새로운 file descriptor를 할당한다.

다음으로는 check\_client를 실행해서 check\_clients 를 실행해서 모든 연결된 디스크립터를 돌면서 rio\_readlinb를 실행해서 입력을 읽어준다. 만약에 들어온 입력이 존재하고 eof가 아니라면 command 함수를 실행하여 명령어를 수행하고, 명령의 결과를 rio\_wrtiten을 통해서 clientbuf의 내용을 client에 넘겨준다.

만약에 입력이 eof라면 해당 연결을 닫는다. 그리고 해당 연결을 닫을 때 연결된 client의 개수가 0이 되면 save\_binary\_tree함수를 실행해서 파일에 기록한다.

(2) Thread\_based server

Thread based 서버는 sub\_init을 수행해서 실행할 buffer를 활성 화하고, buffer를 만든 이후에는 pthread\_create를 해서 Nthread만큼 서버를 만들고, 개수 만큼 해당 기능을 수행하는 thread함수를 호출한다. 그리고 다음의 while문에서 accept함수를 수행하면서 연결된 client가 있으면 새로운 descriptor 를 할당해준다. 그리고 해당 descriptor 를 sub\_insert 함수를 실행시켜서 buffer에 넣어준다. 이후에 thread에서 subf에서 descriptor를 제거하고 해당 descriptor를 받아온다. 그 다음에 연결된 clientNumber를 늘려준다. 이후에는 echo\_cnt함수를 실행해서 echo\_cnt 내부에서

텍스트이(가) 표시된 사진

자동 생성된 설명

이렇게 입력을 받고 출력을 client에 전달해 주는 무한 루프에 들어간다. 만약에 eof를 받으면 해당 루프에서 나와서 함수를 종류하고 thread 함수로 돌아가서 clientNumber를 줄여주고 만약에 연결된 client가 -0개라면 save\_binary\_tree를 호출해서 파일에 현재 주식 정보를 저장한다, 그리고 디스크립터를 닫아준다.

1. **구현 결과**

Event\_basesd\_server 실행화면

텍스트이(가) 표시된 사진

자동 생성된 설명



Thread\_based\_server 실행화면

텍스트이(가) 표시된 사진

자동 생성된 설명



1. **성능 평가 결과 (Task 3)**
2. Client 당 명령어의 개수에 따른 처리율의 차이

주식 개수 40개, client의 개수 100개로 고정을 하고 실험을 진행했다.(코어를 1개로 제한하고 진행, show buy sell 모두 실행 스레드는 1024개 고정.)

Order per client 가 2개 까지는 둘 다 상승하지만, event based 같은 경우 client 당 주문의 개수가 20개를 넘어가면 부하가 걸려서 초당 처리율이 감소하고, thread based server 같은 경우 client 당 주문의 개수가 60개까지는 초당 처리율이 상승하고 이후로는 부하가 걸려서 초당 처리율이 감소하는 모습을 보여준다.

이것으로 봤을 때 client 당 주문의 개수가 올라가면 초반에는 초당 처리율이 증가하나 부하를 넘어가면 초당 처리율이 감소하는 것을 확인 가능하다.

1. Client 개수에 따른 처리율의 차이

주식 개수 40개, order per client 100개로 고정을 하고 실험을 진행했다.(코어를 1개로 제한 하고 진행, show buy sell 모두 실행. 스레드는 1024개 고정)

주식 개수 10개, order per client 100개로 고정을 하고 실험을 진행했다.(코어를 1개로 제한하고 진행, show buy sell 모두 실행 스레드는 1024개 고정.)

첫 번째 데이터에서는 결과가 제대로 나오지 않지만, 두 번째 데이터를 보면 thread based는 client 개수가 80전까지는 20만 이상이다가 80부터는 급격하게 감소한다. Event based는 60이전에는 횡보하나 60 이후로는 1만으로 감소하는 모습을 보여준다. 이런 것을 미뤄 봤을 때 client number에 의해서 event based server 와 thread based server 둘다 영향을 받는다고 볼 수 있다. 또한 주식의 개수에도 영향을 받는 것을 알 수 있다.

Show 명령어에서 binary\_tree를 모두 순회 하기 때문으로 추정된다.

1. 명령어의 종류에 따른 처리 속도의 차이

명령어에 따라서 처리 속도는 다음과 같이 차이가 난다.

Client 개수를 다르게 하였을 떄(event based )

주식 개수 40개, order per client 100개로 고정을 하고 실험을 진행했다.(코어는 1개로 제한을 하고 진행 스레드는 1024개 고정)

Order per client를 다르게 하였을 때 (event based )

주식 개수 40개, client를 100개로 고정을 하고 실험을 진행했다.(코어는 1개로 제한을 하고 진행)

Client 개수를 다르게 하였을 떄(thread based )

주식 개수 40개, order per client 100개로 고정을 하고 실험을 진행했다.(코어는 1개로 제한을 하고 진행 스레드는 1024개 고정)

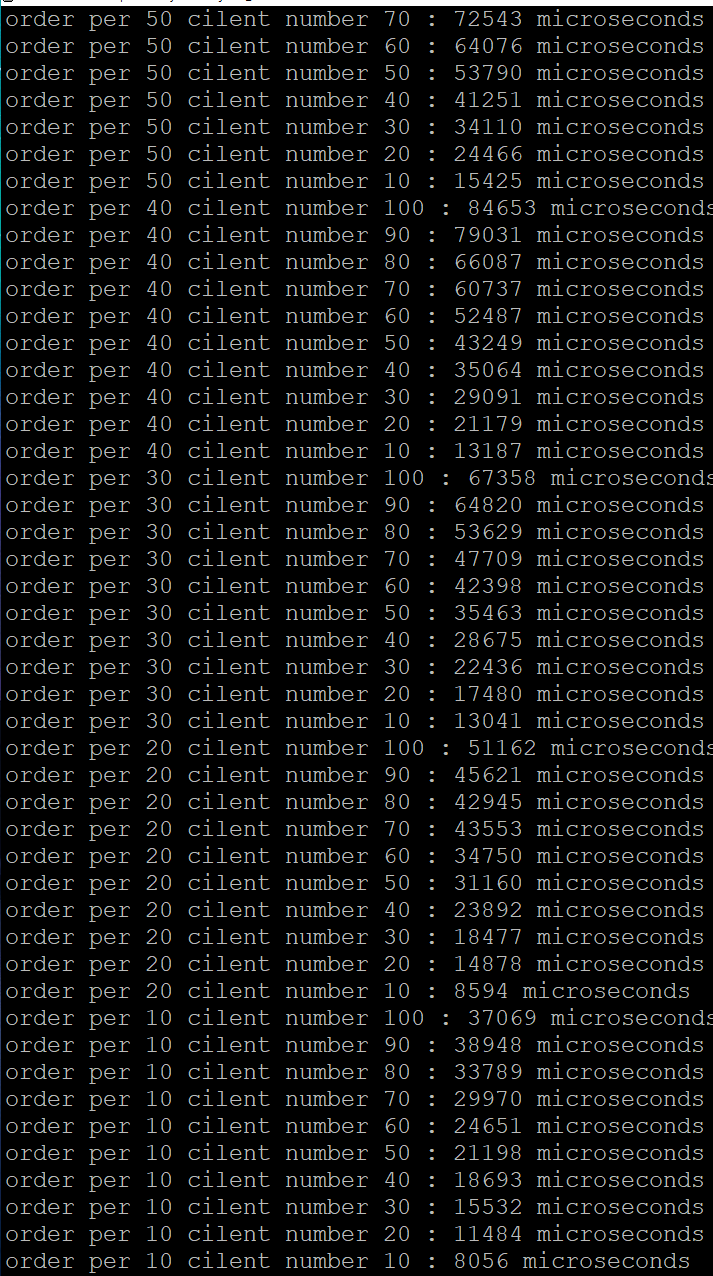
Order per client를 다르게 하였을 때 (thread based )

주식 개수 40개, client를 100개로 고정을 하고 실험을 진행했다.(코어는 1개로 제한을 하고 진행)

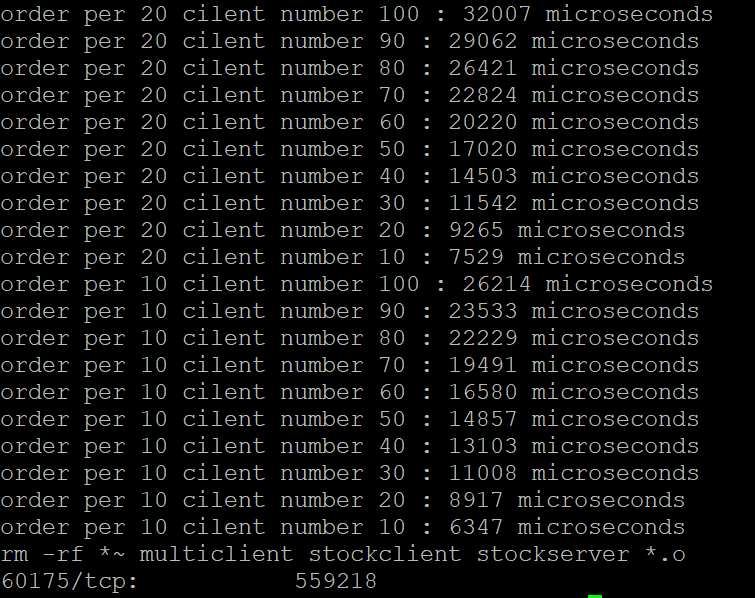
위의 4개 자료에서 공통적으로 나타나는 값은 client 개수, client당 주문의 수에 상관 없이 sell buy명령어만 실행하는 경우가 가장 속도가 높았고, 그 다음으로는 명령어를 섞은 경우가 그 다음이고, show만 명령어로 나오는 것은 가장 속도가 느렸다. 이유는 sell 이나 buy는 해당 주식을 찾고 그에 대한 처리를 해주면 되는데, show 같은 경우는 모든 노드를 순회하면서 출력을 해야하기 때문으로 추청된다.

1. Thread의 개수가 따른 작업 효율 차이(thread server 의 경우)

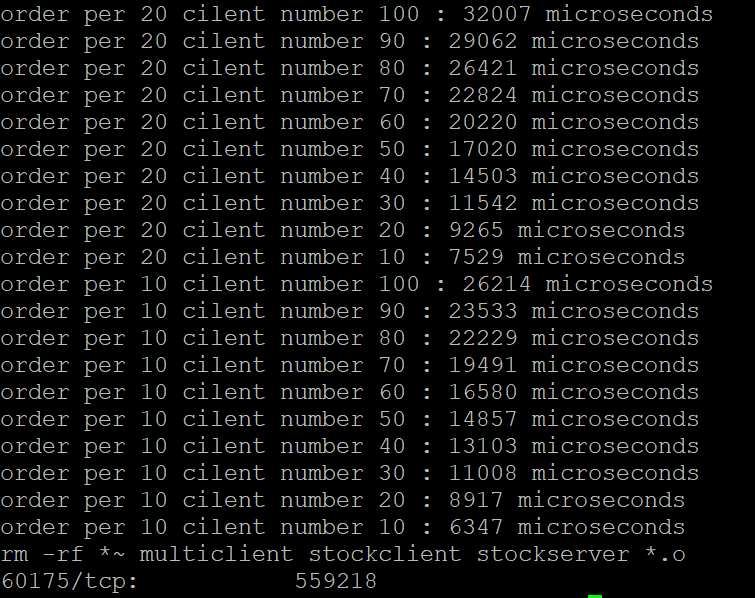
Thread의 개수가 2인 경우



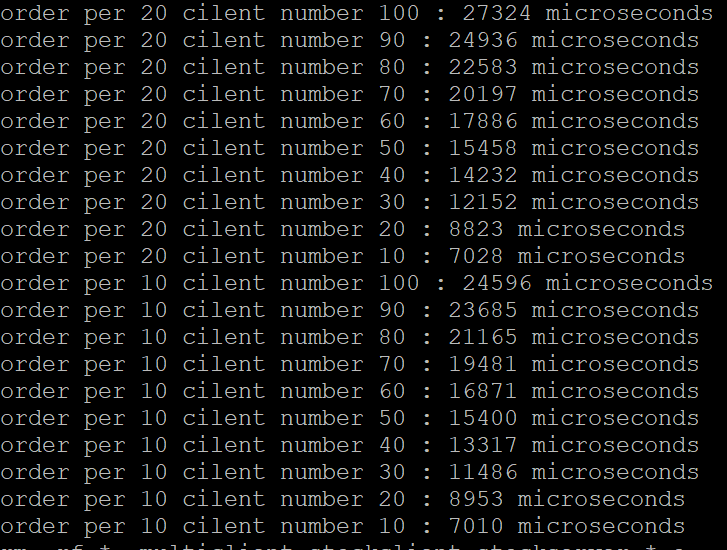
Thread의 개수가 4인 경우



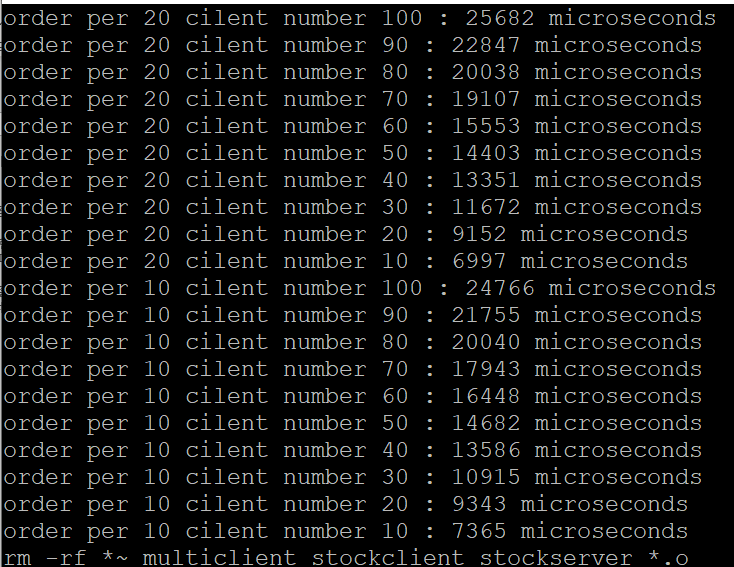
Thread의 개수가 8인 경우



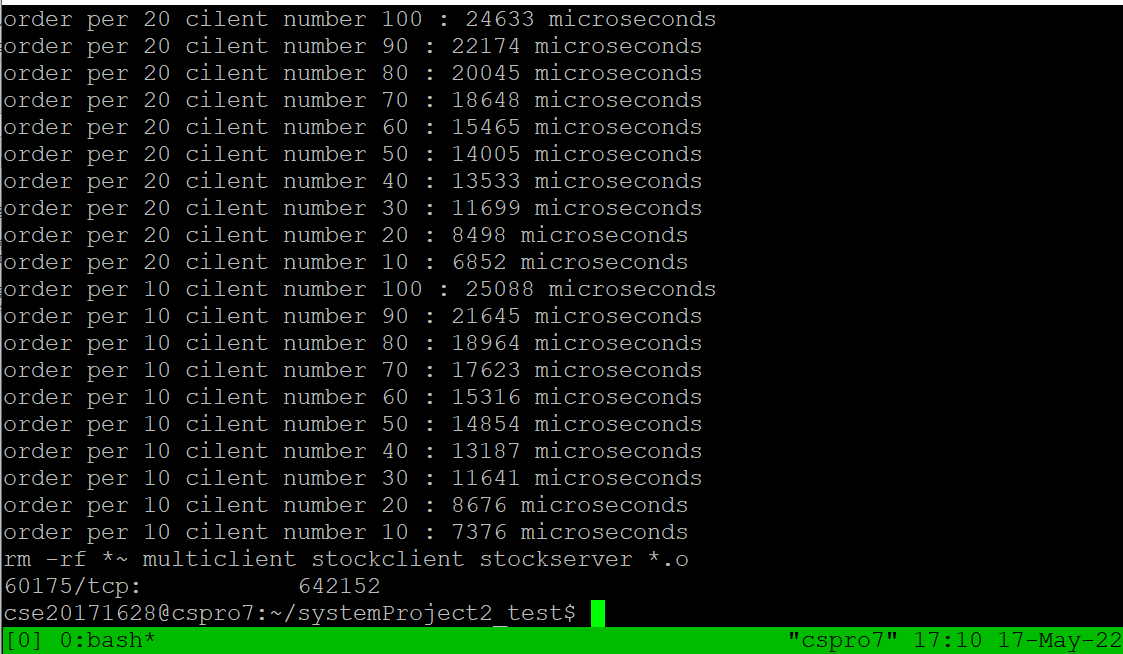
Thread의 개수가 16인 경우



Thread읙 개수가 32인 경우



Thread의 개수가 64인 경우

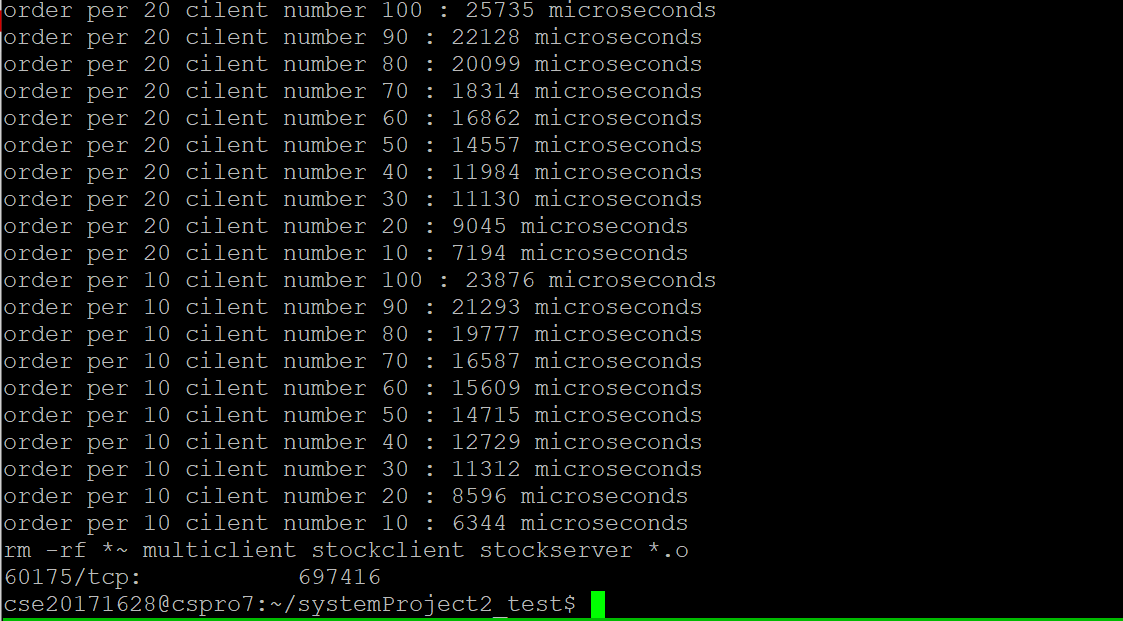


Thread의개수가128개인경우

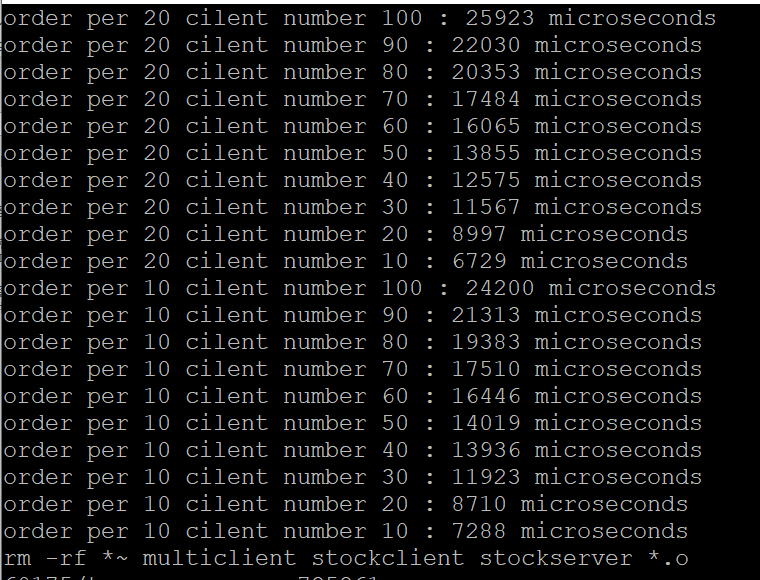
텍스트이(가) 표시된 사진

자동 생성된 설명

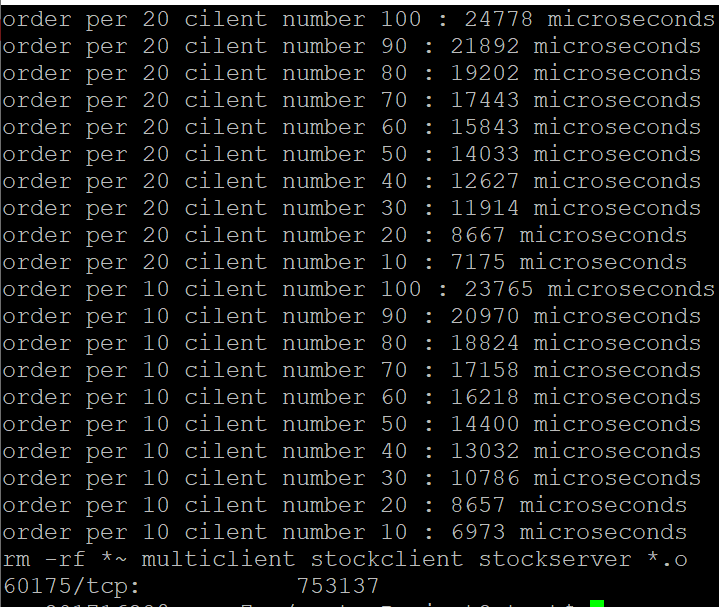
Thread의 개수가 256개 인 경우



Thread의 개수가 512개 인 경우



Thread의 개수가 1024개 인 경우



주식의 개수가 40개이고 order per client는 고정인 차트를 생성했다.(코어 1개에서만 돌아가게 설정)

주식의 개수가 40개이고 client number는 고정인 차트를 생성했다.(코어 1개에서만 돌아가게 설정)

위의 두 자료를 미루어 보았을 때 order per client 가 다르고, client number가 달라도 thread의 개수에 따라서 초당 처리율이 달라지는 것을 확인 가능하다.

스레드가 늘어나면 일정수준 까지는 초당 처리율이 늘어나나 스레드가 일정 수준 이상으로 가면 더 이상 초당 처리율이 늘어나지 않는다. 이유는 스레드를 생성할 때 드는 overhead가 커지기 때문이다. 또한 스레드로 병렬적으로 처리를 하더라도 동시에 처리 할 수 있는 작업의 양은 제한 되어 있기 때문에 크게 차이가 나지 않게 된다.

1. Cpu코어의 개수 제한에 따른 차이

아무런 처리 없이 ./stockserver (포트 번호)의 명령어를 돌리게 되면 해당 프로세스가 여러 코어를 옮겨 다니면서 실행되어서 실행 시간에 영향을 주게 된다. Event based server 의 경우 크게 상관이 없으나 thread based server 는 thread에 따라서 성능차이가 존재 할 수 있다고 생각해서 실험을 진행하였다.

측정한 결과는 다음과 같다.

주식의 개수가 40개이고 명령어는 제한이 없고 order per client는 고정(100개)인 차트를 생성했다.(event based)

주식의 개수가 40개이고 명령어는 제한이 없고 client number는 고정(100개)인 차트를 생성했다.(event based)

Taskset 명령어가 event based server 에는 영향이 미미하다는 것을 알 수 있다.

주식의 개수가 40개이고 명령어는 제한이 없고 order per client는 고정(100개)인 차트를 생성했다.(thread based) 스레드는 1024개를 생성하였다.

주식의 개수가 40개이고 명령어는 제한이 client는 고정(100개)인 차트를 생성했다.(thread based) 스레드는 1024개를 생성하였다.

주식의 개수가 40개이고 명령어는 제한이 없고 order per client는 고정(100개)인 차트를 생성했다.(thread based) 스레드는 100개를 생성하였다.

주식의 개수가 40개이고 명령어는 제한이 client는 고정(100개)인 차트를 생성했다.(thread based) 스레드는 100개를 생성하였다.

결과는 스레드가 1024개 일 때는 돌아가는 cpu에 제한을 두지 않아도 크게 차이가 나지 않지만, 스레드가 100개 일 때는 돌아가는 cpu에 제한을 두게 되면 실행시간이 크게 차이가 나게 된다. 이런 결과를 보면 multicore로 thread server를 돌리게 되면 초당 처리율이 높아진다는 결론이 나온다. 단, 스레드가 너무 많아지면 thread server 에 부하가 걸려서 속도가 비슷해 진다는 분석이 나온다.

1. Event-based vs thread-based

위의 차트에서(1번 실험 cpu를 고정해놓고 실험을 진행함) 다른 변수들이 전부 동일할 경우 thread based server(스레드가 1024개인 경우)가 event based server 보다 전반적으로 초당 처리율이 높다. 이로 미루어 봤을 때 thread 로 인한 overhead가 존재하더라도 단일 cpu에서도 전반적으로 thread\_based server 가 초당 처리율이 높다. 생각하는 이유 2가지는 다음과 같다, 첫번째 이유로는 thread가 여러 개 생성되어서 latency hiding으로 인해 다른 thread가 fetch를 하는 동안 다른 thread는 연산을 진행해서 event가 루프를 1번 돌 동안 thread는 여러 번 연산이 가능하다. 두번째 이유는 서버에서 cat /proc/cpuinfo를 실행하면 cpu의 정보가 뜨는데



이 모델은 10코어 20 스레드 모델이다. 그러므로 cpu를 1개만 사용해도 thread를 2개 쓰는 효과가 나타나서 event based server 보다 thread based server 가 빠르다고 나왔다고 생각한다.

이 실험을 돌릴 때 사용된 코드는 여기서 확인이 가능하다.

https://github.com/mirageoasis/systemProject2\_test