System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 전찬

학번 : 20201635

1. 개발 목표

network programming과 concurrent programming을 바탕으로 두 형태의 주식 서버를 구현한다. network programming을 바탕으로 client / server 가 통신할 수 있는 형태를 구현하며, concurrent programming을 바탕으로 실제 HTS(home trading system)와 같이 여러 client가 동시에 한 서버에 접속할 수 있는 형태를 구현한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Event-driven Approach를 바탕으로 한 server process에서 여러 client를 concurrent 하게 처리할 수 있도록 구현한다. 이를 위해 event-driven approach가 어떻게 구현되는지 파악하며, server / client가 read/write 를 반복하는 형태를 구현한다. 또한 server의 4가지 요청(show, exit, buy, sell)을 처리할 수 있도록 server을 구현한다.

2. Task 2: Thread-based Approach

   Thread-based Approach를 바탕으로 Task 1과 동일한 작동을 하는 server을 구현한다. 각 Accept에서 새로운 thread를 생성해내며, Thread routine을 통해 위에서 설명한 4가지 요청을 수행할 수 있도록 구현한다. 또한 thread-based 형태에서 발생할 수 있는 문제점인 sharing problem을 해결하기 위해 Semaphore 개념을 바탕으로 각 server thread 간의 mutual exclusive access 를 보장해주도록 프로그램을 작성한다.

3. Task 3: Performance Evaluation

   event-based, thread-based 각각의 고유한 특성, 혹은 두 concurrent server에서 발생할 수 있는 여러 차이점을 측정을 통해 파악해 본다. time.h module의 gettimeofday function을 통해 이를 측정해낼 수 있다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Event-driven concurrent 방식은 한 개의 process만을 가지고 concurrent를 구현하는 형태이다. 이때 사용되는 기술이 Multiplexing 인데, multiplexing은 원래 회로에서 사용하는 용어로, 여러 개의 input 중 하나의 input을 사용하는 것을 의미한다. event-driven 형식에서는 이와 비슷하게, Multi-client의 여러 connection 요청 중, 하나의 connection을 선택해서 처리하는 것을 의미한다. 이와 같은 multiplexing은 select, 혹은 epoll function을 사용해서 구현할 수 있다. 이번 project의 구현에서는 select function을 사용했으며, 이 흐름은 아래와 같다.

1. select function을 통해 준비된 file descriptor(fd_set ready_set)가 몇 개인지 파악한다.

2. ready_set 중 listenfd에 해당하는 pending bit가 set 되어 있는 경우, Accept function을 통해 새로운 connection을 만들어주며, pool structure에 추가해준다.

3. 각 connection에 대해 ready_set에 pending bit가 존재하는 경우, server에서 연결된 client와 상호작용한다.

4. 1~3 과정을 while(1) 형태로 계속해서 반복한다.

따라서 위 과정을 통해 하나의 process에서 concurrent 하게 multi-client의 접속을 허용할 수 있는 server을 구현해낼 수 있다.

✓ epoll과의 차이점 서술

select() function의 경우 여러 한계점이 존재하는데, 우선 첫 번째로 fd_set의 허용 가능한 최대 길이가 1024라는 점이 있다. 또한 fd_set의 모든 element를 검색한다는 비효율적인 측면도 존재한다. 이와 다르게 epoll() function은 커널 레벨에서 multiplexing을 지원하며, select() function 보다 빠른 처리를 수행할 수 있다. 또한 실행할 때마다 모든 fd_set을 확인하는 것이 아닌, 어떠한 event가 발생한 element만 관리하기 때문에, 이에 대해서도 장점이 존재한다. 마지막으로 select()

function에서의 fd 제한이 1024인 것과 달리, epoll() function은 최대 제한 개수가 훨씬 크기 때문에, 동시에 많은 client를 제어할 수 있는 장점 또한 존재한다.

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

  ✓ Master Thread의 Connection 관리

  Thread-based approach에서 Master Thread는 listenfd를 관리하는 역할만을 수행한다. 새로운 connection이 존재하면, Accept function을 통해 새로운 connection을 할당하며, 이를 Pthread_create function을 통해 새로운 peer thread를 만들어내며, 만들어낸 thread에게 connect된 client와의 상호작용을 수행하도록 하는 형태이다. 결과적으로 master thread는 listenfd만을 관리하며, connection과 상호작용은 peer thread에게 완전히 일임하는 형태이다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  worker thread pool은 위에서 설명한 peer thread로, 실질적으로 connection을 통해 개별적인 client와 상호작용하는 thread를 의미한다. 이와 같은 thread는 child process에서 fork() 이후 wait()을 통해 reaping하는 것과 동일하게 reaping하는 작업이 필요한데, 이를 Pthread_join(), Pthread_detach() 등의 함수를 통해 만들어낼 수 있다. Accept 이후 Pthread_create function에 의해 만들어진 worker thread는 Pthread_detach(pthread_self()); 와 같은 형태로 reaping을 수행할 수 있도록 구현했는데, Pthread_detach() function은 input thread id에 해당하는 thread가 종료되는 경우, 자동적으로 reaping을 수행할 수 있도록 만들어주는 함수이다. 따라서 위와 같은 형태로 reaping()을 thread가 종료되는 시점에서 자동으로 종료될 수 있도록 구현했다.

- **Task3 (Performance Evaluation)**

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

  기본적으로 구현한 2가지 방법의 server에서 각 명령이 실행되기 위해 걸리는 시간(show와 buy, sell command 간의 요청 타입에 따른 속도 차이 비교)을 측정해 볼 수 있다. 이 경우에는 1개의 client에서 많은 show, 혹은 buy, sell 명령어를 처리하는 데에 걸리는 시간을 측정해서 파악할 수 있다. 이를 측정하는 이유는, event-based server와 달리 thread-based server 에서는 여러 control flow가 존재하기 때문에 semaphore 방법을 사용해서 정보를 보호해야 한다. 하지만 위 방법을 사용하는 경우 각 buy, sell command를 수행하는 데에 걸리는 시간이 더 클 것임을 예상할 수 있는데, 이를 파악하기 위해서이다. 또한 위 방법으로 워크로드에 따른 분석을 수행할 수 있다.

  또한 각 server에서 client가 늘어남에 따라서 속도가 어떻게 변화하는지 또한 측정해볼 수 있다. 이 경우에는, 동일한 명령 개수(명령의 개수가 영향을 미칠 수 없도록 10개로 고정한다.)에서 client의 수가 증가함에 따라 속도가 어떻게 변화하는지를 통해 각 server에서 client 연결의 처리 속도가 어떻게 변화하는지를 파악할 수 있다.

  마지막으로 client 수 * command 수의 개수를 고정시키고 비교해볼 수 있다. 총 command 수를 동일하게 유지시키는 경우, client 수가 많아짐에 따라 시간이 얼마나 걸리는지 양상을 파악할 수 있는데, 이는 각 server가 multi-core의 이점을 살릴 수 있는지, 또한 확장성에서 장점이 존재하는지를 파악하는 한 척도가 될 수 있다.

  추가로 thread-based server에서 semaphore을 적용하는 server와 적용하지 않는 server 사이의 실행 시간을 비교해볼 수 있다. semaphore을 적용하지 않는 경우, 이 server은 정상적으로 작동하는(unsafe region에 접근하지 않는) server은 아니지만, 위 비교를 통해 mutex, P, V에 접근하는 데에 얼마나 시간이 걸리는지 파악할 수 있을 것이다.

✓ Configuration 변화에 따른 예상 결과 서술

위 4가지 형태로 분석하는 경우, 수업 시간에 배운 내용을 토대로 결과를 예측할 수 있다.

우선 첫 번째 각 server에서 show / buy & sell 의 속도 비교에서 show, buy & sell command 모두 단순하게 server와 client가 read / write만을 수행하면 되기 때문에 속도 차이가 크지 않을 것임을 예측할 수 있다. 하지만 thread-based server의 경우, show command의 수행에서는 단순한 read / write 이지만, buy & sell command 의 경우에는 semaphore을 바탕으로 P(&mutex), V(&mutex)를 수행해야 하기 때문에 show 보다는 buy & sell command가 더 오래 걸릴 것임을 예측할 수 있다.

두 번째 동일한 command 수(10개의 명령)에 대해 client을 증가시키는 경우, thread-based에서 Pthread_create routine을 수행하기 위해 걸리는 시간이 존재하므로, client가 증가함에 따라 걸리는 시간 또한 점점 더 커질 것임을 파악할 수 있다.

세 번째 client * command number 을 고정시켰을 때, event-based server의 경우 하나의 process가 모든 command를 수행해야 하기 때문에, 속도의 변화가 거의 없지만, thread-based server의 경우, multi-core의 이점을 살리게 되며 많은 command가 동시에 수행될 수 있다. 따라서 client가 core 수에 가까워질수록 점점점 더 빠른 속도를 보이며, 이후에는 다시 느려지는 양상을 보일 것임을 예상할 수 있다.

마지막으로 semaphore을 적용하는 server와 적용하지 않는 server 사이에 비교를 통해서 semaphore routine(P & V)을 수행하는 데에 걸리는 시간과 함께, semaphore을 적용하는 경우 생길 수 있는 parallelism 문제 또한 관측할 수 있을 것 같다. 여기에서 parallelism 문제란 각 P & V 가 존재하기 때문에, thread-based라도 온전하게 multi-core의 이점을 살릴 수 없이, suspended 될 수 있는 control flow가 존재할 수 있다는 것이다.

C. 개발 방법

기본적으로 stock.txt file에 저장된 data을 tree 형태로 구현하기 위해, left, right child node를 point 할 수 있는 node 구조체가 필요하다. 또한 thread-based의 경우에, buy, sell을 진행할 때 unsafe region의 접근 가능성이 존재하기 때문에, 위 node에 추가 원소로 mutex 또한 필요하다. 또한 tree 구조체에 관련되어 일정 node를 찾는 find() function, tree data를 저장하는 save_data() function 또한 구현해주어야 한다.

event-based concurrent server는 강의 시간에 교수님께서 설명해 주신 event-based concurrent echo server을 변형해서 구현해낼 수 있다. 기존의 echo server은 단순하게 server에서 check_clients() function을 통해 Rio_readlineb(), Rio_writen() 을 반복해서 수행하는 형태였지만, stock server은 이 형태를 변형하여 Rio_readlineb() function을 통해 client로부터 전송된 요청 command를 받으며, 받은 command를 바탕으로 show / exit / buy / sell 에 따라 각 case에 관련된 프로세스를 진행하며, 결과를 다시 client에게 Rio_writen()으로 보내주는 형태로 구현할 수 있다.

thread-based concurrent server 또한 강의 자료를 바탕으로 구현해낼 수 있다. listenfd에 요청이 존재하면, Pthread_create() function 을 통해 새로운 thread를 만들어내는 것이다. 이 경우에는, thread routine을 echo에서 stock server로 변경해주어야 하는데, 이 routine은 connection이 종료될 때까지 while(1) 을 통해서 event-based server의 check_clients() 부분에서 구현한 것과 동일하게 구현하면 된다.

3. 구현 결과

2번을 바탕으로 두 개의 server을 구현했으며, event-based는 while loop를 통해 concurrent 하게, thread-based는 thread routine을 통해 client와 connect할 수 있는 형태로 두 가지 server을 구현했다. 또한 ./multiclient ~~ 을 통한 테스트, 혹은 ./stockclient ~~ 을 통해 직접 입력을 통해 server – client 간의 상호작용이 정상적으로 작동함을 파악할 수 있었다. 하지만 이번 과제에서 thread-based concurrent server을 구현할 때, pool 형식으로도 구현할 수 있는데, 이는 time, space의 trade-off

를 바탕으로 server와 client의 connect 시간을 단축시킬 수 있는 형태이다. 하지만 이러한 방식으로 thread-based concurrent server을 구현하지는 못했다. 만약 위 형태로 구현한다면, server의 실행과 함께 Pthread_create() 로 생성한 thread들을 array로 저장하고, Accept가 실행되는 경우, 해당 thread를 connection에 사용하는 형식으로 구현할 수 있을 것 같다.

4. **성능 평가 결과 (Task 3)**

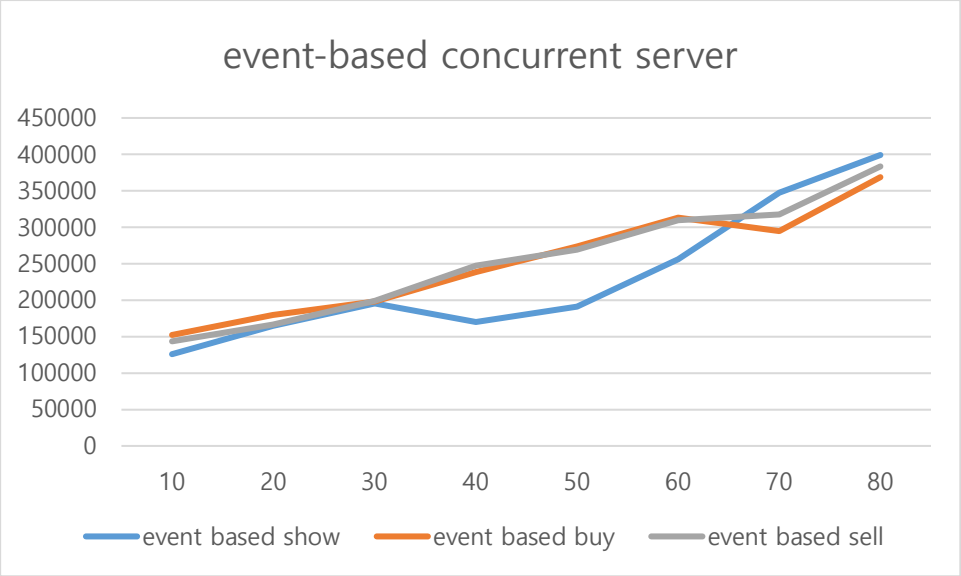
성능 평가에서 가장 중요한 것은, 어느 부분을 측정할 것인지 정하는 것이다. 측정 부분의 차이가 측정에 큰 영향을 미칠 수 있기 때문이다. 예를 들어 fork() 와 같은 system call은 한 줄의 코드지만 걸리는 시간이 다른 코드 부분보다 클 수 있기 때문에, 이러한 형태를 고려해주어야 한다. 또한 측정 방법으로는 프로젝트 코드로 제시된 multiclient.c 의 내용에 약간의 변형을 가하여 시간을 측정을 수행했다.

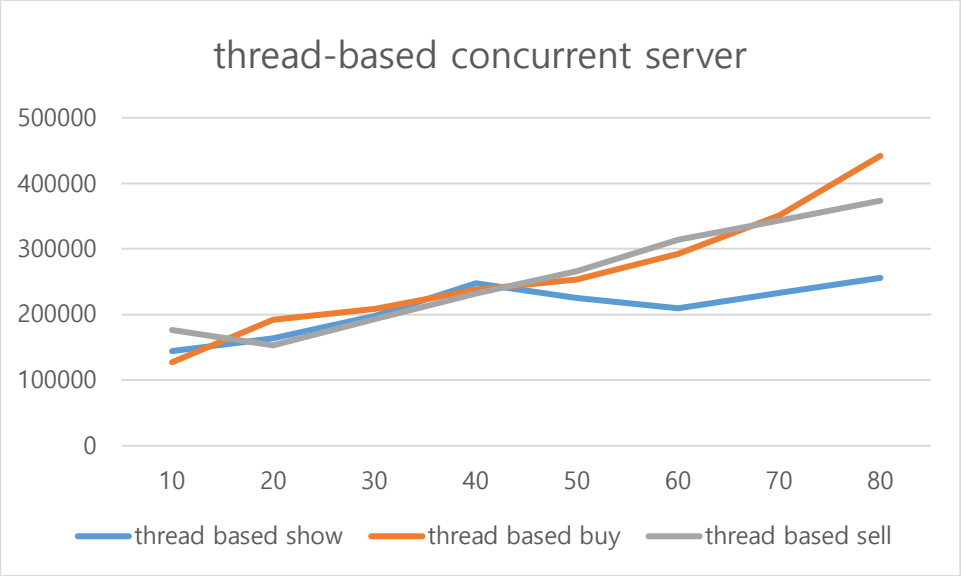
위에서 설명한 것처럼, 어느 부분을 측정할 것인지 정하는 것이 중요하다. client의 실행 ~ 종료를 측정한다면 코드 내부의 fork()와 같은 system call의 영향을 많이 받을 것이고, Read / write 의 통신 부분만 측정한다면, Open_clientfd, Accept를 통해 server에서 새로운 thread, 혹은 새로운 connfd를 만드는 데에 측정하는 시간을 모두 무시하게 된다. 따라서 개인적으로 선택한 방법은, 실행 ~ 종료 시간을 측정하되, fork() 수행에 걸리는 시간만을 따로 측정해 총 측정 시간에서 빼주는 것으로 측정을 수행했다. 위와 같이 측정하는 경우, fork()의 system call 측정 시간을 줄일 수 있으며, Open_clientfd 등의 event, thread based 각 경우의 특징을 살릴 수 있기 때문이다. 추가로 wait()의 측정 시간을 제거하지 않는 이유는, wait() 자체가 child process의 통신 시간에 연관될 수 있기 때문에 wait()에 걸리는 시간은 제거해주지 않았다. 예를 들어, 종료되지 않은 child process의 wait()에 걸리는 시간을 제거해준다면, child process가 server와 통신하는 시간도 일부분 제거되기 때문이다.

또한 다른 cspro – cspro7 과 같이 다른 서버끼리 연결을 수행하게 되면, 네트워크 오버헤드가 존재하게 된다. 네트워크 오버헤드 또한 각 서버의 특성을 파악하기 위한 측정에서는 고려해야 할 사항이 아니기 때문에, 측정은 cspro – cspro 형태로 동일한 네트워크 내에서 수행했다.

따라서 위 측정 시간을 기준으로 측정한 결과는 아래와 같다.

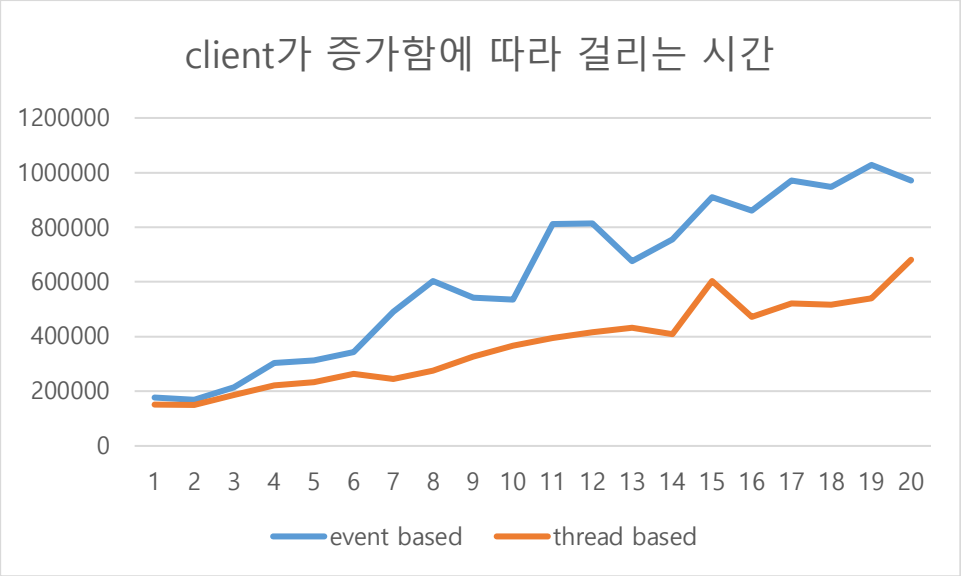
1. 한 client가 event / thread-based에서 각 command을 수행하는 데에 걸리는 시간





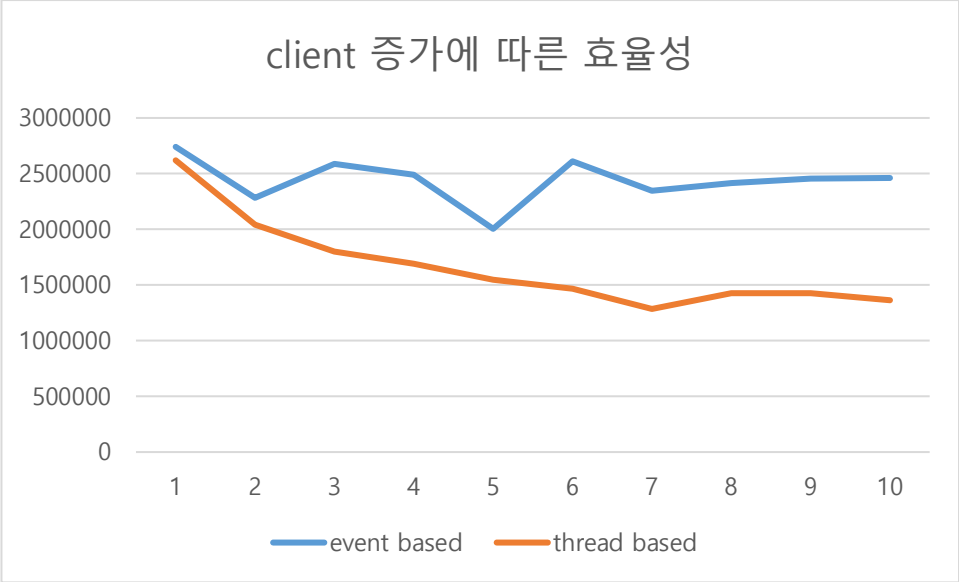
1개의 client가 x축 command의 개수를 처리하는 데에 걸리는 시간은 위 그래프와 같다. 위 경우 event-based에서는 command가 70-80 개일 때, show, buy, sell command 사이에 큰 차이가 존재하지 않지만, thread-based에서는 buy, sell은 400000, show 는 250000 정도가 걸리는 것을 파악할 수 있다, 이는 thread-based에서 semaphore 개념을 바탕으로, buy, sell을 수행하는 데에 걸리는 시간이 더 크다고 해석할 수 있다.

2. 1~20개의 client가 100개의 명령을 수행하는 데에 걸리는 시간



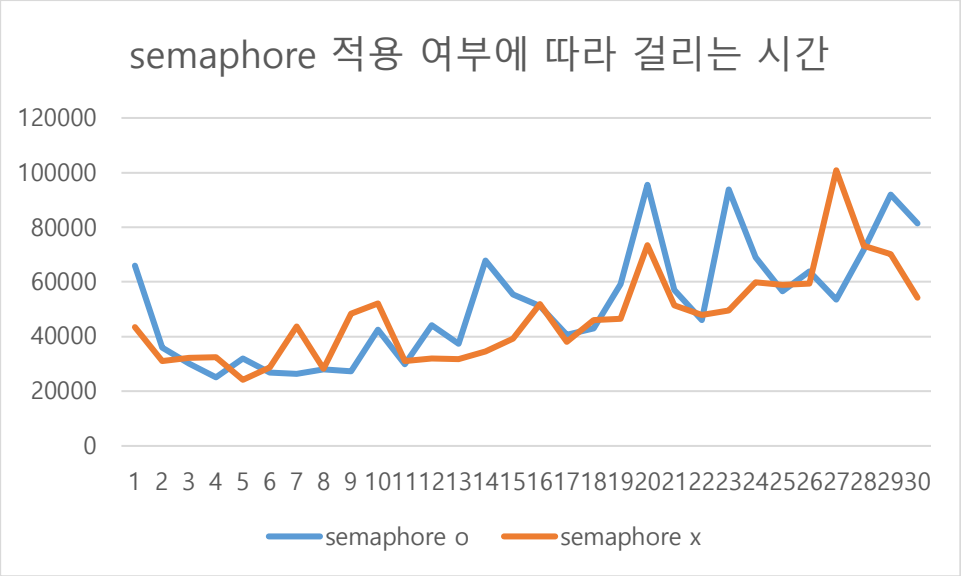
위 형태를 통해 각 형태에서 client가 많아짐에 따라 걸리는 시간 또한 증가함을 파악할 수 있다. 더 나아가 측정 이전까지는 event-based server이 process-based, thread-based와 달리 system call이 없기 때문에 더 적은 시간이 걸릴 것이라고 예상했는데, 생각과 달리 thread-based보다 오래 걸리는 형태임을 파악할 수 있다. 그 이유로 가장 큰 것은, /proc/cpuinfo 로 확인할 수 있는 정보를 기반으로 server가 단일 cpu가 아닌, 8 core, 16thread cpu이기 때문에 multi-core의 이점을 살리며, 위와 같은 결과가 도출되었다고 해석할 수 있었다.(/proc/cpuinfo 명령의 접근 권한이 부족해서, 위 정보는 강의 자료를 토대로 8 core, 16 thread라고 해석했다.) 또한 더 나아가, server을 1 core에서만 작동하도록 제한해보고 싶었는데, 아직은 이와 같은 방법을 몰라서 cpu 제한 방법을 나중에 배운다면, 이를 적용해볼 것이다.

3. client * command = 1000 인 경우 client 가 증가함에 따라 걸리는 시간



총 명령 수가 동일한 경우에, client가 증가함에 따라, event-based는 속도 차이가 거의 없는 형태인 것에 반해 thread-based는 총 걸리는 시간이 감소하는 형태임을 파악할 수 있다. 그 이유로는 위에서 말한 것과 같이 multi-core의 이점을 살릴 수 있기 때문이며, 1~10개의 client로 증가함에 따라, 16개의 thread를 이용할 수 있는 thread-based가 훨씬 빠른 속도로 동작할 수 있게 된다. 이는 수업 시간에 배운 내용인 P & V가 존재하는 경우에 true parallelism 한 형태가 아닌 것에도 적용할 수 있는데, true parallelism 인 경우에, client가 증가함에 따라 걸리는 시간이 1/2, 1/3 형태로 크게 변화해야 하는데, P & V가 존재하기 때문에 multi-core의 이점은 살릴 수 있으나, client 수에 반비례하게 걸리는 시간이 감소하지 않는다는 것 또한 파악할 수 있다.

4. thread-based server에서 semaphore의 적용 여부에 따라 걸리는 시간



semaphore을 사용하는 thread-based, 사용하지 않는 thread-based에서 동일한 총 command(3번과 동일)에 대해 client 가 증가함에 따라 걸리는 시간은 위와 같다. 강의를 바탕으로, 사용하는 것과 사용하지 않는 것 사이에서 큰 차이가 날 것이라 예상했는데, 별다른 패턴, 혹은 차이를 파악하지는 못했다. 그 이유로는 option = rand() % 3 + 1 으로 되어있어서 확률적으로 semaphore을 기반으로 P & V을 적용하는 경우가 전체의 절반 정도이기 때문에 정확한 차이를 볼 수 없을 수 있으며, 1~3번의 분석에서는 loop을 100~1000번씩 돌리는 형태로 분석해서 더 정확한 결과가 나왔다면, 위 경우에서는 client 수가 1~30으로 많아서 loop을 많이 돌리다 보면 Open_clientfd 에러가 잘 발생하기 때문에, 정확한 분석을 수행하지는 못했다.