System Programming Project 2

담당 교수 : 김영재 교수님

이름 : 안도현

학번 : 20181650

1. 개발 목표

- 주식 서버를 두 가지 방법으로 만들고, 이에 대한 성능 평가와 분석으로 이루어진 프로젝트이다. Event driven 방식의 주식 서버는 Select문을 사용해 file discriptor set을 관리해 event가 발생한 클라이언트에 대해서 순차적으로 처리한 뒤 다시 Select를 수행하는 것을 반복하는 구조로 작성할 것이다. Thread-based 방식의 서버는 말 그대로 각 클라이언트에 대한 연결을 별개의 쓰레드로 만들어 concurrency를 확보하는 방법으로, 수업 시간에 배운 Readers-Writers problem과 producer consumer problem을 혼합한 방식으로 프로그래밍 하는 것이 목표이다. 성능 평가는 다양한 환경에 대해 프로세스를 실행시킬 때 gettimeofday call을 이용하여 elapsed time을 측정해 서로 비교하는 방식으로 진행할 것이다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   Event-driven 방식은 쓰레드나 멀티 프로세스 기반 프로그래밍이랑은 달리 control flow가 top to down의 일렬로 이루어진다. 따라서 fd set을 처음 구축하는 과정을 제외하면 방식 또한 비교적 간단하며, safety 문제에서도 매우 자유롭다. 또한 주식 클라이언트로부터 요청된 처리가 동시에 되는 게 아니라 여러 클라이언트의 요청을 순서대로 처리하고 다시 trigger된 fd를 찾는 것을 반복하는 식이기 때문에 한 작업이 다른 작업을 침범하지도 않을 것이다.

2. Task 2: Thread-based Approach

   Thread-based 방식은 각 클라이언트와의 연결로 만들어진 connfd를 생성해둔 쓰레드와 대응시켜 처리하는 방식이다. 단순히 클라이언트로부터 connection 요청이 들어올 때마다 새로운 쓰레드를 만드는 방식은 쓰레드 생성에 오버헤드가 들어갈 뿐더러 프로그램이 커지면 unsafe control 문제가 발생하기 쉽기에, 수업 시간에 배운 Prethreaded server 모델을 채용해 코드를 작성할 것이다. 그렇게 함으로써 주식 종목 테이블의 각 노드에 대해 여러 쓰레드들이 R/W를 concurrent하게 수행하는 모델을 완성한다. 물론 각 노드에 대한 Readers Writers problem 또한 고려가 필요하다.

3. Task 3: Performance Evaluation

   기준일로부터의 elapse time을 얻을 수 있는 gettimeofday call을 사용해 서버가 클라이언트 요청 처리에 사용하는 수행 시간을 얻고, 다양한 환경을 입력으로 줘 가면서

그 시간을 비교해 짠 프로그램의 성능을 평가한다. 클라이언트의 개수와 워크로드 종류, 쓰레드 기반 코드의 경우 미리 선언한 사용 쓰레드 개수에 따른 시간 변화 등을 분석해 결과를 기록한다.

B. 개발 내용

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

  ✓ Event-driven 방법은 하나의 통신 채널을 다수의 주식 클라이언트와 connection하고 통신하기 위해서 I/O multiplexing을 사용한다. listenfd를 통해 새로운 주식 클라이언트와 연결이 되면서 생성되는 connfd들이 어떤 자료구조(배열)에 몰아넣어 저장이 되고, 그들로부터 어떤 요청이 발생했는지를 지속적으로 감시(select 함수)하여 해당 클라이언트의 요청들에 대응하는 동작을 주식 서버에서 수행하도록 한다. 따라서 연결된 connfd들을 저장할 자료구조, 매번 select의 인자로 넘길 set들과 검사할 인덱스를 위한 변수들이 필요하고 클라이언트 추가, 각 클라이언트 요청을 처리 등을 담당할 함수 등이 필요하다. 실제 구현에 대해 간단히 설명하면 메인에서 루프를 돌며 클라이언트의 메시지를 포착할 Select 함수를 적절한 인자와 함께 호출하고, 그로 얻은 결과에서 listenfd bit가 set되었는지를 검사해 그에 따라 새로운 client와 연결과 동시에 그들의 정보를 저장하며, 각 클라이언트의 요청을 처리하는 함수 호출을 반복한다.

  ✓ epoll은 select의 단점을 보완한 함수이다. select를 사용해서 i/o multiplexing을 할 때 가장 핵심이 되는 동시에 가장 번거로운 부분은, 매번 select로 set들을 넘겨 새로운 ready set을 받아오는 것과 그들에 대한 요청을 처리할 때 자료구조를 순회하며 FD_ISSET을 호출해 그 각각의 SET 여부를 검사해야 되는 부분이었다. 하지만 epoll을 사용하면 이런 단점을 해결할 수 있다. 즉 ready set과 같은 것을 굳이 프로그래머가 명시적으로 선언해 매번 관리하는 것이 아니라 epoll_create()을 사용해 커널이 직접 관리하도록 하는 것이다. 우리는 감시 대상(클라이언트)가 추가되거나 disconnect될 경우 그 사실만 epoll_ctl을 통해 전달하면 된다. 그럼 epoll이 이벤트가 발생한 대상만 뽑아주거나 할 수 있기 때문에 결국 코딩을 하는 입장에서는 매우 효율적이라고 생각될 것이다.

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

  ✓ 이 방식에서 Master Thread는 직접 connected client에 대한 처리를 담당하지 않는다. task1과는 달라진 부분이다. master thread에서 하는 것은 listenfd로부터 accept한 connfd를 어떤 버퍼에 전달하는 것뿐이다. 그리고 그들에 대한 처리를 담당할 Worker threads가 버퍼에서 connfd를 꺼내 그들 자신이 그 connfd를 사용해 클라이

언트와 통신한다. 이는 producer-consumer problem이 적용된 부분이다.

- ✓ Worker threads는 그 쓰레드를 언제 생성하는지에 따라 두 가지 케이스로 나뉘는데, 상술했듯 이번 과제에선 쓰레드를 미리 생성해두고 그들이 종료되지 않으면서 계속 새로운 connfd를 받아 다수 개의 클라이언트와 통신하도록 하는 방식을 채택했다. 이들은 쓰레드라는 개념 하에 concurrent한 방식으로 각 주식 종목에 접근해 buy나 sell로써 write하거나 show로써 read하게 되는데, 어떤 종목의 update 중에는 read가 불가능하도록 설계했고, 그렇지 않다면 reader는 얼마든지 접근이 가능한 first readers writers problem을 적용해 코드를 작성했다. Worker threads 관리를 위해 통신중인 쓰레드의 개수를 전역 변수로 관리하고 그들을 컨트롤할 때 쓸 바이너리 세마포를 하나 추가로 선언한다. 그렇게 함으로써 모두 sbuf_remove 이후로 진행하지 못하는 상태, 즉 connection 상태인 worker thread가 없는 상태에 들어가면 stock.txt를 현재 트리를 순회하며 갱신한다.

- **Task3 (Performance Evaluation)**

  - ✓ 우선 동시 처리율을 정의한다. 이는 concurrent server가 다수의 client로부터 받은 요청을 처리한다고 할 때 시간당 처리하는 개수를 나타낸다. 당연히 구하는 방법은 multiclient에서 요청하는 개수에서 그 요청을 처리하는 코드의 소요 시간을 나누면 구할 수 있을 것이다. (이 시간은 gettimeofday() call을 이용해 두 포인트의 시간을 구해 차를 계산하는 것으로 구할 것이다.) 다수의 클라이언트 요청을 동시에 처리하는 concurrent 서버는 iterative server에서 각 클라이언트가 연결을 위해 한참을 기다려야 되는 문제를 해결한 것이기 때문에, 단위 시간당 얼마만큼의 처리를 동시에 처리할 수 있느냐는 여러 종류의 concurrent server의 성능을 비교할 때 필수적일 것이다.

  - ✓ 우선 분석할 것들은 client 개수에 따른 동시 처리율, 요청 종류에 따른 동시 처리율, event driven과 thread based의 동시 처리율 비교, thread based 내에서 NTHREAD 변화에 따른 동시 처리율 등을 살펴볼 것이다. 예상되는 결과는, client가 많아질수록 처리율이 감소하고 클라이언트가 많아질수록 이 감소폭이 커질 것이다. 요청 종류는 트리의 모든 부분을 순회하며 버퍼를 채워줘야 하므로 show가 가장 처리율이 낮을 것이다. NTHREAD는 커질수록 쓰레드를 너무 많이 사용하게 되므로 처리율이 낮아질 것이다. event based와 thread based는 일반적으로 쓰레드 생성에 걸리는 오버헤드가 존재하지 않는 event based쪽이 더 효율적일 것 같다고 예상한다.

C. 개발 방법

두 방식에서 공통적으로 관리하는 자료구조로서, 각 주식 종목을 파일로부터 입력받아 메모리에 적재할 때 Binary Search Tree를 사용한다. 그렇게 함으로써 랜덤하게 섞인 주식들을 ID기준으로 탐색할 때 일반적으로 선형 탐색을 하는 것에 비해 빠르게 주식에 접근할 수 있도록 한다. 노드 구조체의 구성은 주식 정보와 child에 접근할 포인터 뿐만 아니라 쓰레드 기반 코드에서 사용할 semaphore도 선언했다. 트리에 관한 함수들은 아래와 같다 서버 실행 시 여러 번 호출되며 트리를 완성할 insert_node()와 SIGINT 발생 시 트리를 할당해제할 free_tree(), 특정 ID를 가진 주식을 트리에서 탐색하는 search_tree(), show 입력이 들어올 때나 stock.txt을 갱신할 때 호출되어 트리를 순회하며 caller에 전달할 버퍼에 정보를 채워주는 show_tree()가 있고, buy나 sell 요청 수행 시에는 search_tree()를 통해 노드를 탐색해 update하는 buy()와 sell()함수가 있다.

Event-driven 방법은 우선 read set과 ready set을 포함해 각종 변수를 선언한 _pool 구조체를 정의한다. 이후 init_pool()에서 max값들과 clientfd 배열을 초기화하고 read set에 listenfd만 set하여 connection 요청을 받도록 한다. 메인함수에서는 무한 루프를 돌며 select->connection if listenfd has been triggered->check_clients() 호출을 반복한다. add_client()는 말 그대로 pool에 새로운 connfd를 추가하는 과정으로, FD SET을 set하고 max값들을 적절히 증가시켜준다. 그리고 check_clients()에서 select에서 선택된 connfd들에 대해 nready를 하나씩 깎아가며 입력 라인을 받아 그에 맞는 동작을 수행해 다시 클라이언트로 전송한다. 매번 요청을 처리한 뒤에 clientfd 배열을 검사해 그 연결이 모두 종료되었으면 stock.txt를 파일 출력을 통해 갱신한다.

Thread-based 방법은 producer-consumer problem을 위해 우선 sbuf_t 구조체를 정의한다. 이는 각 connfd를 저장할 buf와 관련 변수들, 그리고 semaphore들로 이루어져 있다. 메인 함수(master thread)는 sbuf_init()함수로 구조체를 초기화해준 뒤 매크로로 정의된 NTHREAD 값에 따라 그 개수만큼 쓰레드를 생성한다. 물론 이들은 편리함을 위해 detached mode로 변경해준다. 그리고 listenfd로부터 accept가 수행되면 그 생성된 connfd를 sbuf에 넣고, 각 쓰레드는 mutex를 통해 mutual exclusion이 보장된 상태로서 sbuf_remove()를 통해 sbuf 구조체로부터 connfd라는 item을 뽑아낸다. 성공적으로 뽑아낸 쓰레드는 그 connfd를 사용해 주식 클라이언트들과 통신하고, 그 connection이 종료되어도 다시 sbuf_remove()를 호출하는 매커니즘을 반복한다.

위 두가지 서버는 공통적으로 Signal 함수를 통해 SIGINT에 대한 handler를 정의했다. 서버 프로세스의 종료가 ctrl+c로 수행되고 그 핸들러가 다른 시그널에 의해 방해받을 일 또한 프로그램 명세상 일어날 수 없다고 생각하여 그 핸들러에서 트리를 할당해제하고 sbuf를 free하는 루틴을 진행하고 exit 하도록 했다.

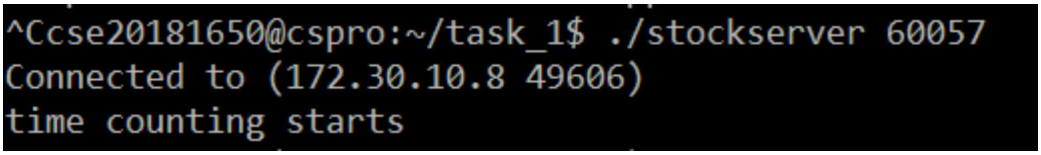
3. 구현 결과

결과적으로 결과는 상술한 내용을 전부 의도한 대로 작동시켰다. Event driven approach는 여러 클라이언트의 요청을 select문을 통해 제대로 잡아내 concurrent하게 처리했으며, Thread based approach는 쓰레드가 의도한 바대로 제대로 동작했다. 전체적으로 쓰레드의 개수 카운트를 위한 쓰레드 뮤텍스, sbuf 작업을 위한 세마포들, 각 노드의 뮤텍스의 세가지 semaphore를 사용했는데, 그들로 작성한 코드도 문제없이 잘 작동했다. 또 NTHREAD의 값에 변화를 줌으로써 NTHREAD가 클라이언트 개수보다 적으면 여러 번에 걸쳐서 코드가 수행되는 것 또한 확인할 수 있었다.

쓰레드 기반 주식서버를 구축하면서 들었던 의문은 event driven 방식에서는 show 요청이 server가 receive 한 시점의 트리 정보를 되돌려주는데 반해, 쓰레드 방식에서는 그 주식들의 시점이 정확히 인지가 안된다는 것이었다. 왜냐하면 각 주식 노드에 대해서만 lock을 관리했기 때문이다. 단순히 일부 주식만 조금 오래된 정도가 아니라, 극단적으로는 비교적 뒤늦게 업데이트 한 것이 반영되지 않고 일찍 수정한 것은 반영되는 경우 또한 발생할 수가 있었다. 이를 일반화하면 클라이언트가 서버로부터 수신한 최신 정보가 사실은 서버가 가진 최신 정보와 꽤 차이가 있을 수도 있는 것이었다. 물론 트리 전체에 대해 lock을 관리한다면 writer가 starving에 빠질 가능성이 매우 높아지거나 정보의 정확도가 떨어지는 등 비효율적인 모델일 것이기에, 사이버 캠퍼스에 질문을 올렸다. 결과적으로 클라이언트는 그 정보의 시점이 정확히 언제일지 아는 것이 어려울 뿐더러 가치도 없다는 답변을 받았다. 어떤 시점의 정보를 얻고 클라이언트에 저장함과 동시에 그 정보들이 다시 바뀌면 클라이언트의 정보는 최신 정보가 아니기 때문이다. 그래서 나는 클라이언트 입장에서 그 정보들이 정확히 같은 시점의 정보임이 보장되어야 하는 제약이 있다면 thread based approach는 적절한 방식이 아니라고 이해했다.

4. 성능 평가 결과 (Task 3)

모든 테스트는 gettimeofday() call을 사용하여 마이크로초 단위로 시간을 측정했는데, 두 경우 모두 처음 listenfd로부터 accept하여 첫 클라이언트와의 connection이 생성되었을 때 start, 어떤 client와도 연결되지 않은 상태에 돌입하면 end를 얻어 두 timeval 변수의 차를 비교했다. 아래 캡쳐와 같이 첫 연결을 수행하는 메시지를 출력한 뒤 바로 counting이 시작된다(모든 실험에서 이 시점은 동일하며 주석 처리된 코드를 확인)



또한 각 서버를 실행해 값을 비교할 때, BST의 특성상 item을 삽입하는 순서에 따라 탐색 시간 차이가 많이 나므로, 다음과 같은 stock.txt를 가지고 일정하게 실험했다.(아래와 같은 데이터를 저장한 tmp.txt를 계속 cp tmp.txt stock.txt를 실행함으로써 사용)

```
4 1187 2323
2 21433 23232
3 1058 1000
5 114 2000
1 2242 232323
8 128 2000
6 319 22232
10 186 800
9 1 4000
7 78 3000
14 323 2000
12 333 3333
13 2354 2222
15 225 2222
11 333 5555
18 323 2323
16 7677 8878
20 2323 2323
19 777 7777
17 2323 232323
```

마지막 조건으로 기존에 클라이언트에서 요청 간에 존재하던 1초의 텀을(usleep 라인) 지우고 각 요청이 연속적으로 실행될 수 있도록 했고, 짧은 시간으로 측정되는 결과에 서버 환경에 따라 오차가 클 수 있기에 3회 시행 후 중간값을 취해 시간을 측정했다.

- Client 개수에 따른 동시 처리율 비교, Event based와 Thread based 방식의 동시 처리율 비교

각 클라이언트는 100개의 요청을 보낸다고 미리 설정해둔다. Event based의 경우 클라이언트가 20,40,60,80,100개일 때 각각 아래와 같은 결과를 얻는다.

```
time counting ends
elapsed time for event driven approach : 162214 usec
```

```
time counting ends
elapsed time for event driven approach : 314321 usec
```

```
time counting ends
elapsed time for event driven approach : 517630 usec
```

```
time counting ends
elapsed time for event driven approach : 788678 usec
```

```
time counting ends
elapsed time for event driven approach : 1017212 usec
```

N_Client	동시처리율(처리/초)
20	12329
40	12725
60	11591
80	10143
100	9830

요청 개수에서 초 단위의 시간을 나누면 표와 같은 값으로 동시 처리율이 감소하는 추세를 보인다. 40개로 늘어날 때는 concurrent한 design이 긍정적이 효과를 내서인지 소폭 처리율이 증가하지만, 그 이후 큰 폭으로 감소한다. 즉 연결된 클라이언트 개수가 많아질수록 maxi나 maxfd같은 값들이 증가하고, 따라서 select()를 진행하거나 FD_SET 순회와 pool 관리에 드는 오버헤드는 클라이언트 개수가 커질수록 커진다는 것을 의미할 것이다.

그렇다면 같은 과정을 Thread based에서 관찰하면 어떻게 될까?

```
time counting ends
elapsed time for thread based approach : 145311 usec
```

```
time counting ends
elapsed time for thread based approach : 282523 usec
```

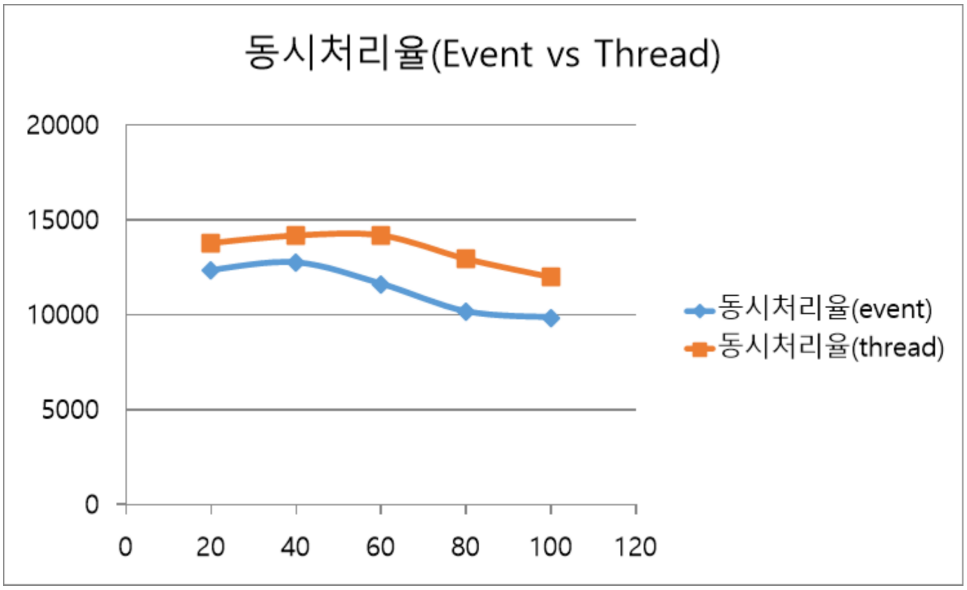
```
time counting ends
elapsed time for thread based approach : 423746 usec
```

```
time counting ends
elapsed time for thread based approach : 619234 usec
```

```
time counting ends
elapsed time for thread based approach : 836283 usec
```

N_Client	동시처리율(처리/초)
20	13763
40	14158

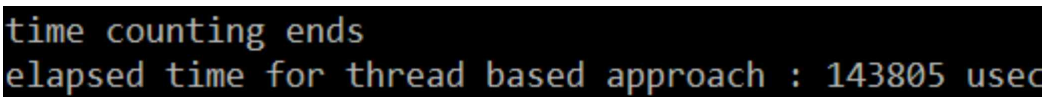
60	14159
80	12919
100	11957

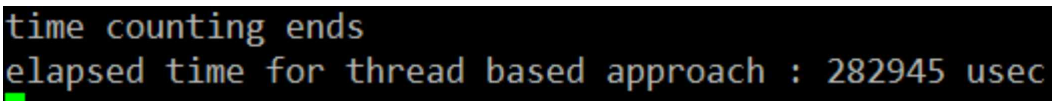


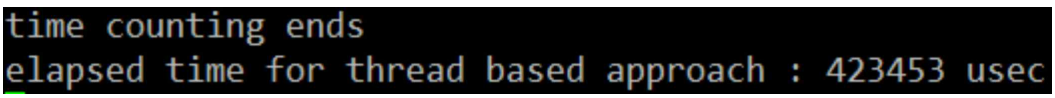
Event 때와 마찬가지의 경향성을 띤다. client 개수가 많아질수록 동시처리율이 감소하는 추세를 보이는데, 주목해야 할 점은 큰 정도로 차이나는 건 아니지만, 두 방식의 시간 소요에는 차이가 있다는 것이다. 여기서 의아한 것은 실험 전 예상했던 것과 다르게 thread based approach에서 전반적으로 동시처리율이 높게 나왔다는 점이다. 심지어 이는 클라이언트의 개수에 상관없이 쓰레드를 100개 생성할 때의 결과이다. 이 의미는 쓰레드 생성의 오버헤드와 context switching하며 다수의 클라이언트를 처리하는 것보다 Event based에서 명시적으로 Select를 반복하고 target set들을 프로그래머가 직접 작성한 코드로 다루는 오버헤드가 더 컸다고 예측할 수 있다. 이는 쓰레드 생성에 의한 오버헤드가 없어 high performance를 위해 Event based를 사용한다는 수업시간의 내용과는 차이가 있는 부분이었다. 정확하게 말하면, epoll이 아니라 select 기반으로 상당한 양의 클라이언트 요청을 수용할 때 시간 소모가 꽤 있다는 것은 확실해 보였다. 이 이유는 무엇일까? 먼저 아래 그림과 같이 cspro 상에서 cat /proc/cpuinfo를 입력해 보자. 엄청난 라인이 서버 cpu 정보를 설명하기 위해 출력되는데, 이에 따르면 쓰레드를 동시에 돌릴 수 있는 자원이 환경 내에 상당히 많다. 따라서 multicore의 장점을 제대로 살릴 수 없는 event based와 그 강점을 발휘할 수 있는 thread based는 client 개수가 많아질수록 차이가 날 수밖에 없게 되는 것이다.

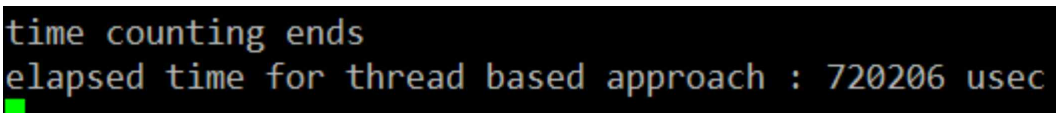
```
x2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveo
bugs            : cpu_meltdown spectre_v1 spectre_v2 spe
bogomips        : 4389.49
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 19
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20
stepping        : 1
microcode       : 0xb000038
cpu MHz         : 2399.976
cache size      : 25600 KB
physical id     : 0
siblings        : 20
core id         : 12
cpu cores       : 10
apicid          : 25
initial apicid  : 25
fpu             : yes
fpu_exception   : yes
cpuid level     : 20
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 api
on pebs bts rep_good nopl xtopology nonstop_tsc aperfmpe
adline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnow
x2 smep bmi2 erms invpcid rtm cqm rdseed adx smap xsaveo
bugs            : cpu_meltdown spectre_v1 spectre_v2 spe
bogomips        : 4389.49
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

위와 같은 결과는 NTHREAD를 100개로 맞추고 그 이하의 클라이언트의 요청만 받았을 때의 결과이고, 만약 NTHREAD를 10으로 고정한 채(각 클라이언트는 100개의 요청 전송) 클라이언트의 개수만 10개에서 20개,40,60,80,100개로 늘리면 어떻게 될까? 아래 그림은 그 실험 결과인데, 뭔가 익숙한 수의 배치가 보인다. 바로 클라이언트 수가 늘어난 만큼에 거의 비례한 만큼 시간 또한 소요되었다는 것이다. 다시 말해, 쓰레드를 100개로 고정해 클라이언트의 요청을 처리하도록 했던 위의 경우와 비교할 때 처리율은 거의 감소하지 않으면서 더 높은 처리율을 얻을 수 있었다. 이를 생각해 볼 때, 특수한 경우가 아니라면 NTHREAD값은 서버가 감당할 수 있는 하드웨어적 성능을 고려하여 결정하되, 너무 큰 값을 잡아서 괜히 쓰레드 생성의 오버헤드를 감수하지 않도록 해야 할 것이다.

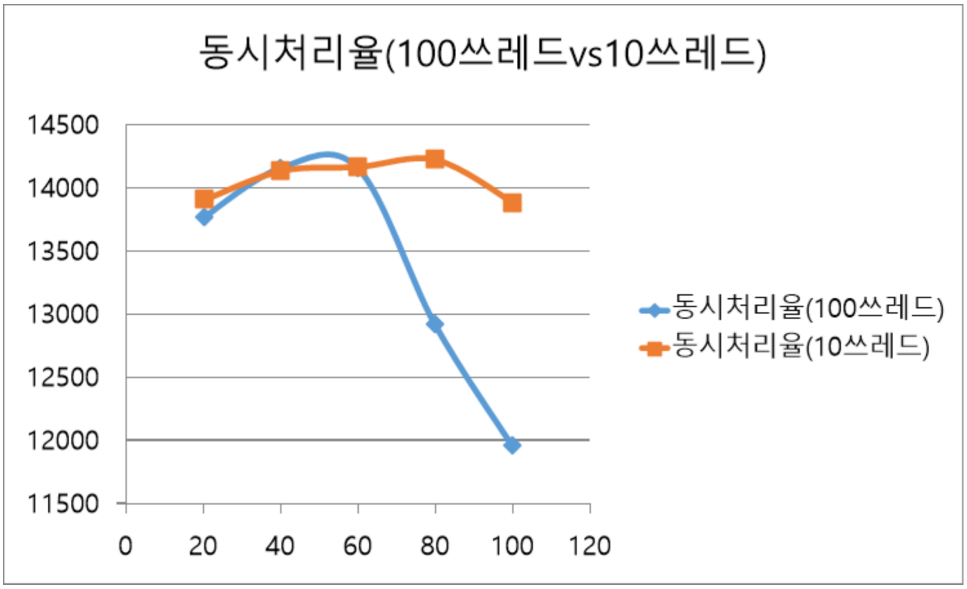








N_Client	동시처리율(처리/초)
20	13907
40	14137
60	14169
80	14225
100	13884



- workload에 따른 동시 처리율 분석(thread based)

Event based approach는 요청을 순서대로 처리하기 때문에 당연히 모든 트리를 순회하는 show가 특정 노드에 대해서만 처리하는 buy나 sell보다 큰 시간이 걸릴 것으로 예측할 수 있다. 따라서 여기서는 먼저 Readers-Writers problem을 적용해 쓰레드 기반이지만 이미 업데이트 중인 노드에 대해서는 read를 할 수 없도록 한 thread based approach에서의 분석을 할 것이다. NTHREAD는 10개, 클라이언트의 개수는 50개로 고정한 채로 시간을 비교한다. 우선 트리의 특정 노드를 찾아 그를 업데이트하는 buy 또는 sell 명령만으로 채울 때(multiclient 코드에서 option을 rand()%3를 rand()%2 + 1로 변경) 값을 의 시간을 살펴보면, 아래와 같다.

```
time counting ends
elapsed time for thread based approach : 352126 usec
```

그리고 모든 트리를 재귀적으로 순회하는 show_tree를 사용하는 show만으로 채운 것의 결과는 아래와 같다.

```
time counting ends
elapsed time for thread based approach : 1166256 usec
```

수행 시간을 볼 때, 20개의 트리 노드와 50개의 클라이언트를 적용한 위 상황에서는 show가 훨씬 많은 시간을 소모한다. 원래 Readers-Writers problem을 고려했을 때 트리 순회와 동기화 오버헤드가 각각 소모되는 양쪽의 케이스는 얼추 비슷한 시간이 나오지 않을까 했던 예상이 정확히 빗나간 것이다. 즉 이 상황에서는 트리를 전부 돌며 정보를 받아오는 show 명령어의 시간 소모가 훨씬 큰 것이다. 그럼 만약 노드 수를 극단적으로

줄이고 클라이언트 수를 늘리면 어떻게 될까? 클라이언트를 100개 노드를 단 2개만 가진 트리를 생성해 비교해보자. 아래는 트리 정보와 buy와 sell만 진행할 경우의 시간이다.

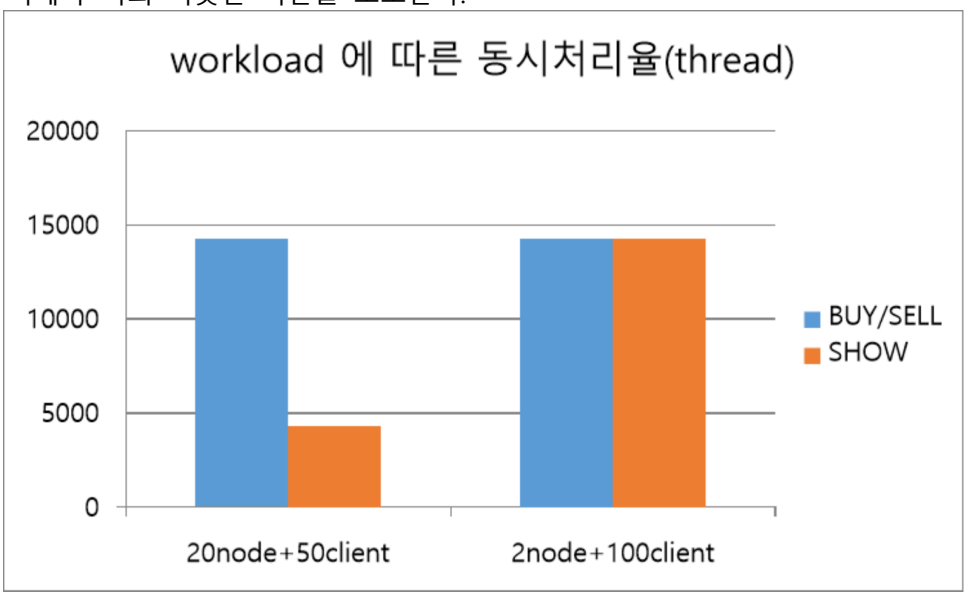
```
1 1 333 5555
2 2 555 5555
```

```
time counting ends
elapsed time for thread based approach : 701786 usec
```

또 아래는 show만 할 경우의 시간이다.

```
time counting ends
elapsed time for thread based approach : 702997 usec
```

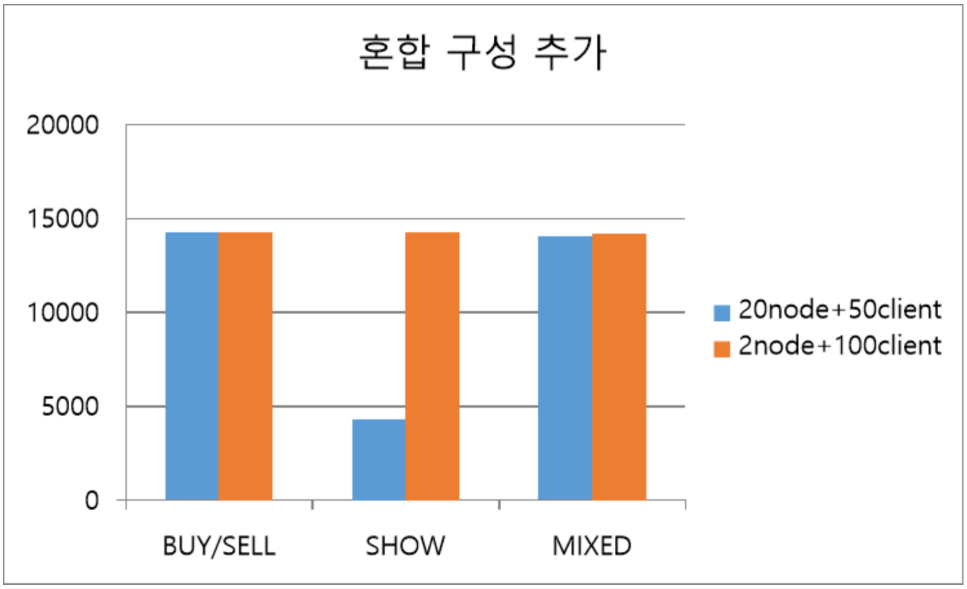
이제야 거의 비슷한 시간을 소모한다.



확실히 이렇게 write 중인 노드에 read하기 위해 접근하는 케이스를 흔한 케이스로 만들어버리면 그렇지 않은 경우에 비해 buy와 sell만 있는 요청 처리가 확 느려진다(노드가 2개밖에 없는 트리에 대해서 연산하는데도 동시처리율이 그대로이다). 즉, 결론을 내자면 서버에서 write 작업이 매우 오래 걸리는 작업이거나 write 때문에 read가 lock이 걸리는 경우가 매우 잦다면, show같은 read 작업만 채운 경우가 buy같은 write 작업으로 채운 경우보다 월등히 빨라질 수 있다. 그로부터 도출할 수 있는 결과로, 두 작업을 섞은 결과를 추가하면 아래와 같이 나타난다.

```
time counting ends
elapsed time for thread based approach : 357266 usec
```

```
time counting ends
elapsed time for thread based approach : 706338 usec
```



- workload에 따른 동시 처리율 분석(event based)

아래는 100개의 클라이언트와 buy,sell만 요청으로 들어올 때의 결과이다.

```
time counting ends
elapsed time for event driven approach : 776504 usec
```

그리고 아래는 show만 요청으로 들어올 때의 결과이다.

```
time counting ends
elapsed time for event driven approach : 2377958 usec
```

사실 여기서 바로 결론을 얻을 수 있다. 예상한 바와 같이 모든 요청을 순회하며 하나씩 순차적으로 처리하는 event based approach에서는 show만으로 이루어져 있는 요청이 훨씬 오래 걸릴 것이라고 예측했고, 결과도 그대로였다. 여기서는 이 결과값을 좀더 명확하게 관찰하기 위해 방법을 구상했고, 트리의 크기가 더 커지면 트리를 전부 순회해야 하는 show가 더더욱 오랜 시간이 걸릴 것이라고 예상해, 노드가 200개인 트리를 만들어 실험해 보기로 했다.(트리 순회에 시간이 더 걸리도록 하기 위해 정렬된 데이터를 그대로 BST에 넣어 치우친 트리를 사용했다.) 아래는 그 stock.txt 데이터의 끝 부분이다.

```
168 168 1008 10000
169 169 994 10000
170 170 1011 10000
171 171 1017 10000
172 172 981 10000
173 173 1009 10000
174 174 953 10000
175 175 1043 10000
176 176 1008 10000
177 177 977 10000
178 178 983 10000
179 179 1013 10000
180 180 984 10000
181 181 989 10000
182 182 980 10000
183 183 1021 10000
184 184 1003 10000
185 185 986 10000
186 186 953 10000
187 187 1023 10000
188 188 1032 10000
189 189 1003 10000
190 190 992 10000
191 191 1017 10000
192 192 1003 10000
193 193 1013 10000
194 194 988 10000
195 195 1015 10000
196 196 1025 10000
197 197 1002 10000
198 198 983 10000
199 199 1018 10000
200 200 1018 10000
```

아래는 sell과 buy만 사용한 결과이다.

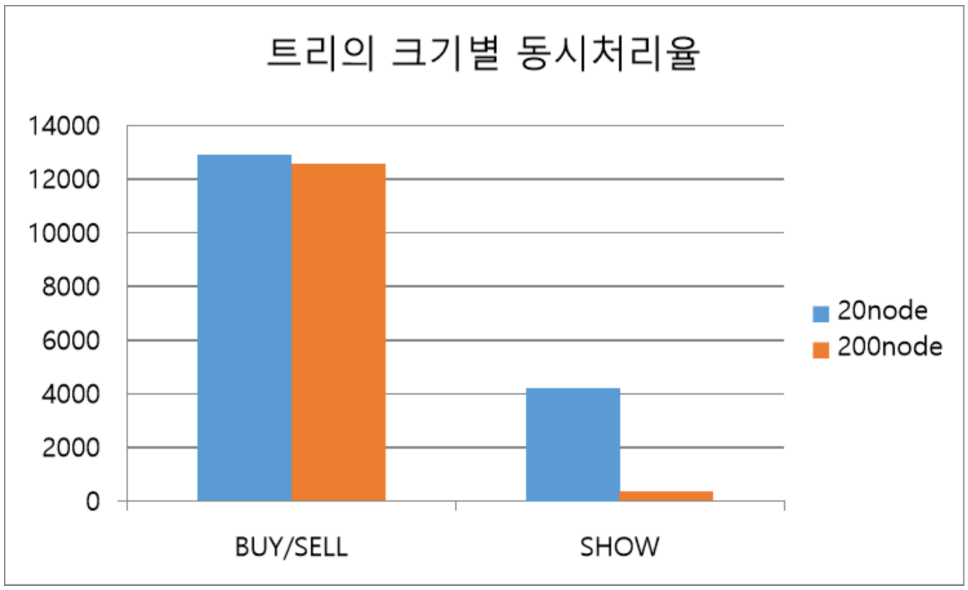
```
time counting ends
elapsed time for event driven approach : 797448 usec
```

또 아래는 show만 사용한 결과이다.

```
time counting ends
elapsed time for event driven approach : 30709863 usec
```

정말 어마어마한 시간의 차이가 발생했다. 이런 결과값들을 볼 때, 확실히 쓰레드를 사용해 멀티 코어로 이루어진 서버 자원의 강점을 전혀 살리지 못하고 순차적으로 처리해야 하는 event driven approach의 단점이 너무나 명확하게 나타난다. 그냥 동시에 읽지를 못

하므로 200*10000개의 노드를 전부 탐색하며 30초 가까이의 시간을 사용한 결과값이었다.



정리하면, Event driven approach의 경우 FD_SET을 돌며 처리해야 하는 것들을 순차적으로 처리하는 방식이라 클라이언트의 요청별로 동기화 문제가 없다. 그래서 워크로드에 따른 분석에서 중요한 포인트는 트리를 순회하며 모든 정보를 얻어야 하는지, 또는 특정 노드를 탐색해 그 부분만 수정하는지로 나뉘는 요청의 종류 구성이 핵심이고 그에 따라 시간 차이가 위와 같이 난다는 것을 알 수 있다.