System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 백인찬

학번 : 20150195

1. 개발 목표

Client의 주식 거래 요청을 concurrent하게 처리하는 server를 구현한다. 이를 구현하기 위한 방법으로 select를 이용한 Event-based server와 pthread를 이용한 Thread-based server를 구현한다. 구현 후 여러 상황에 대한 성능 분석을 통하여 두 방법의 차이를 비교한다.

2. 개발 범위 및 내용

A. 개발 범위

1. select

   select()를 이용하여 event-based server를 구현한다. Event-based server는 socket fd의 배열을 지켜보고 있다가 I/O request가 발생했을 시 해당 fd의 요청을 처리한다. 기존 echo server는 다른 client가 앞선 client의 connection이 종료되는 것을 대기했지만 event-base server에서는 request를 지켜보는 것으로 client가 무한정 대기하지 않아도 된다.

2. pthread

   pthread를 이용해 thread-based server를 구현한다. Thread-based server는 main thread가 client의 connection request를 받아 해당 client의 요청을 맡아서 처리할 thread를 생성한다. 해당 thread는 그 client의 connection이 종료될 때까지 생존한다. Concurrent한 thread의 작업으로 인한 readers-writers problem을 semaphore로 해결한다.

B. 개발 내용

- **select**

  ✓ select 함수로 구현한 부분에 대해서 간략히 설명

  1) select 함수를 사용하기 위해서는 FD의 배열이 필요하다. 따라서 fd_set 타입의 변수를 사용한다.

  2) 1에서 선언한 fd_set을 select 함수를 통해 지켜본다. client로부터 요청이 발생하면 fd_set에서 해당 비트(fd)를 set된다.

  3) fd_set에서 set된 비트(fd)가 server에서 connection을 위해 사용하는 fd라면 connection request라고 판단하여 accept한다.

4) fd_set에서 set된 비트(fd)가 이전에 연결된(accept된) fd라면, 주식 거래에 대한 요청이라고 판단한다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

1) log(n)의 시간복잡도로 탐색을 하기 위해 ASL tree 자료구조를 사용한다.

2) 자료구조 사용을 위해 stock info를 담을 구조체를 정의한다.

3) tree에 삽입 시에 stock info의 ID를 기준으로 정렬한다.

4) show 요청으로 순회를 할 땐 inorder 방식으로 순회하여 ID 기준 오름차순으로 정렬된 값으로 나오게 한다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

1) thread가 수행할 함수를 정의한다.

2) server는 accept 함수를 통해 connection request를 받아들인다. 연결된 fd를 인자로 하는 함수를 수행하도록 thread를 생성한다.

3) thread는 생성되면 detach하여 server가 일일이 join하지 않도록 한다.

4) concurrent한 수행으로 read를 수행하는 show 요청과 write을 수행하는 buy/sell 요청에 대해 데이터의 consistency를 유지해야하는 readers-writers problem이 발생하므로 binary semaphore를 통해 해결한다.

C. 개발 방법

- Event-based server

struct item : tree의 node에 해당하며 stock info를 저장하는 구조체이다.

ASL tree 관련 함수 : tree를 rotate하여 balance를 유지하는 함수들을 구현했다.

construct_stock_tree() : stock.txt 파일로부터 stock info를 읽어들여 in-memory 자료구조로 올린다.

create_item() : 읽어들인 stock info를 저장하는 node를 생성하여 반환한다.

insert_item() : node를 tree에 삽입한다. 삽입 시에 ID를 비교하여 적절한 위치에 삽입시키고 tree를 균형있게 재배열한다.

search_item() : ID를 비교하여 left와 right 중 한 쪽의 path만 탐색하도록 하여 log(n)의

탐색이 가능하게 했다.

change_stock() : target이 되는 주식 node의 잔여주를 변경한다.

update_file() : stock.txt 파일을 update한다.

in_traverse() : inorder 순회를 하며, show 또는 update_file() 함수에서 호출되는데, show 요청에서 호출될 경우 문자열을 인자로 받아 해당 문자열에 주식 정보 전체를 담는다. update_file()에서 호출될 경우 주식 node의 정보를 파일에 출력한다.

action() : client와 연결된 fd를 통해 요청을 읽어들이고 해당 요청에 대한 결과를 다시 fd에 출력한다. exit의 경우 -1을 반환하며 main 함수에서 해당 fd를 close한다.

parseline() : action()에서 호출되며 입력으로 들어온 요청 문자열을 공백(' ')을 기준으로 parsing한다.

exec_cmd() : action()에서 호출되며 parseline()에서 parsing한 문자열에 대해 그에 맞는 결과 문자열을 생성한다. 예를 들어 show의 경우 주식 정보를 in-memory에서 읽어와 문자열에 담는다. exit의 경우 -1을 반환하도록 한다.

- Thread-based server

change_stock() : write에 해당하는 함수이므로 P(), V(), write semaphore를 사용해 변경에 대한 부분을 감싸도록 변경했다.

in_traverse() : read에 해당하는 함수이므로 node의 readcnt를 조절하고, read하는 thread가 있을 시 write semaphore를 lock하고, 해당 node를 read하는 것이 종료되면 lock을 해제하여 write을 가능하게 한다.

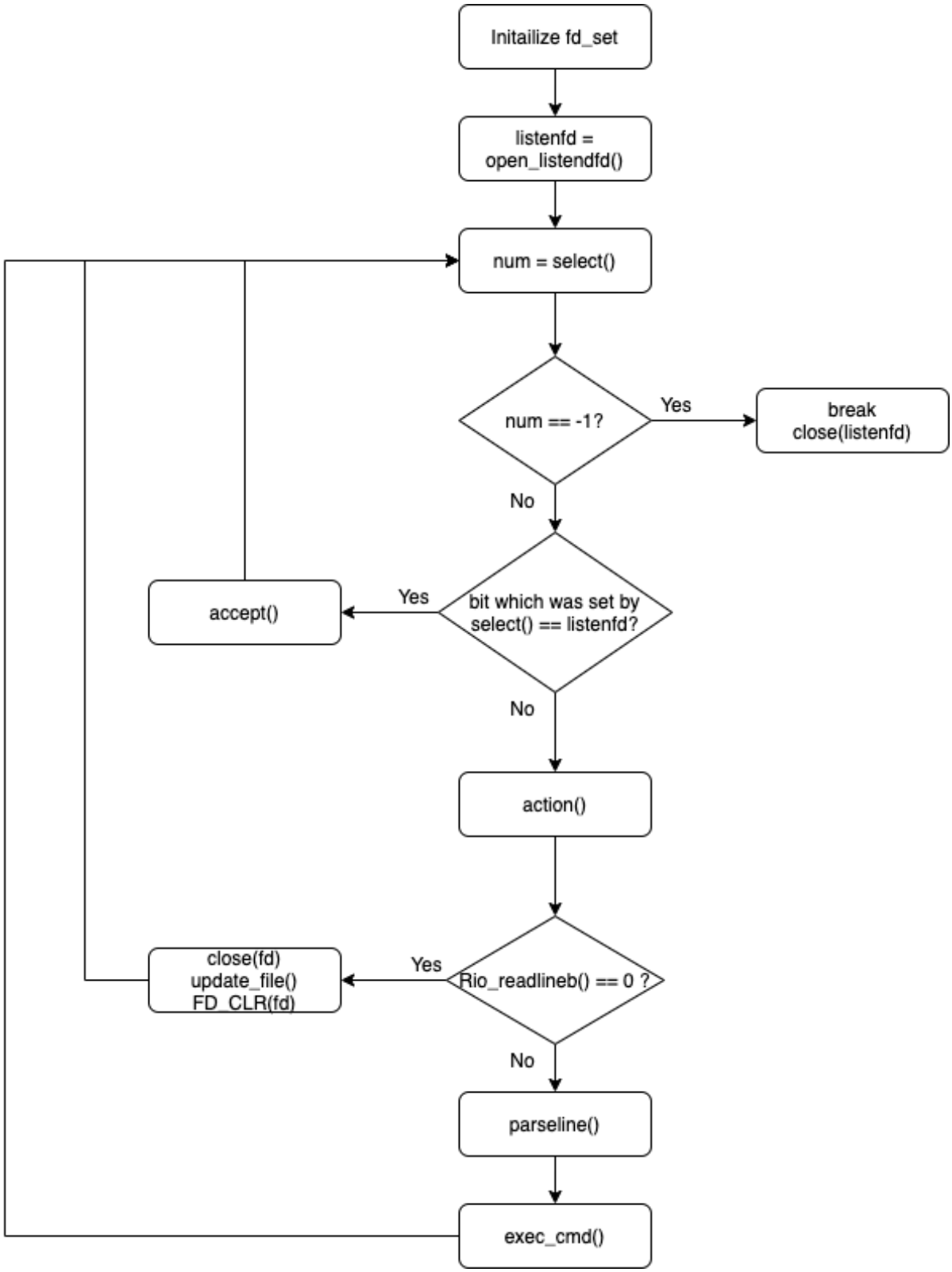
action() : event-based server에서 하나의 요청만 처리하게 하였던 것을 exit 또는 connection이 종료될 때까지 반복하도록 변경했다.

thread_func() : thread를 detach하고, action을 호출하며 종료 시에 file을 update한다.

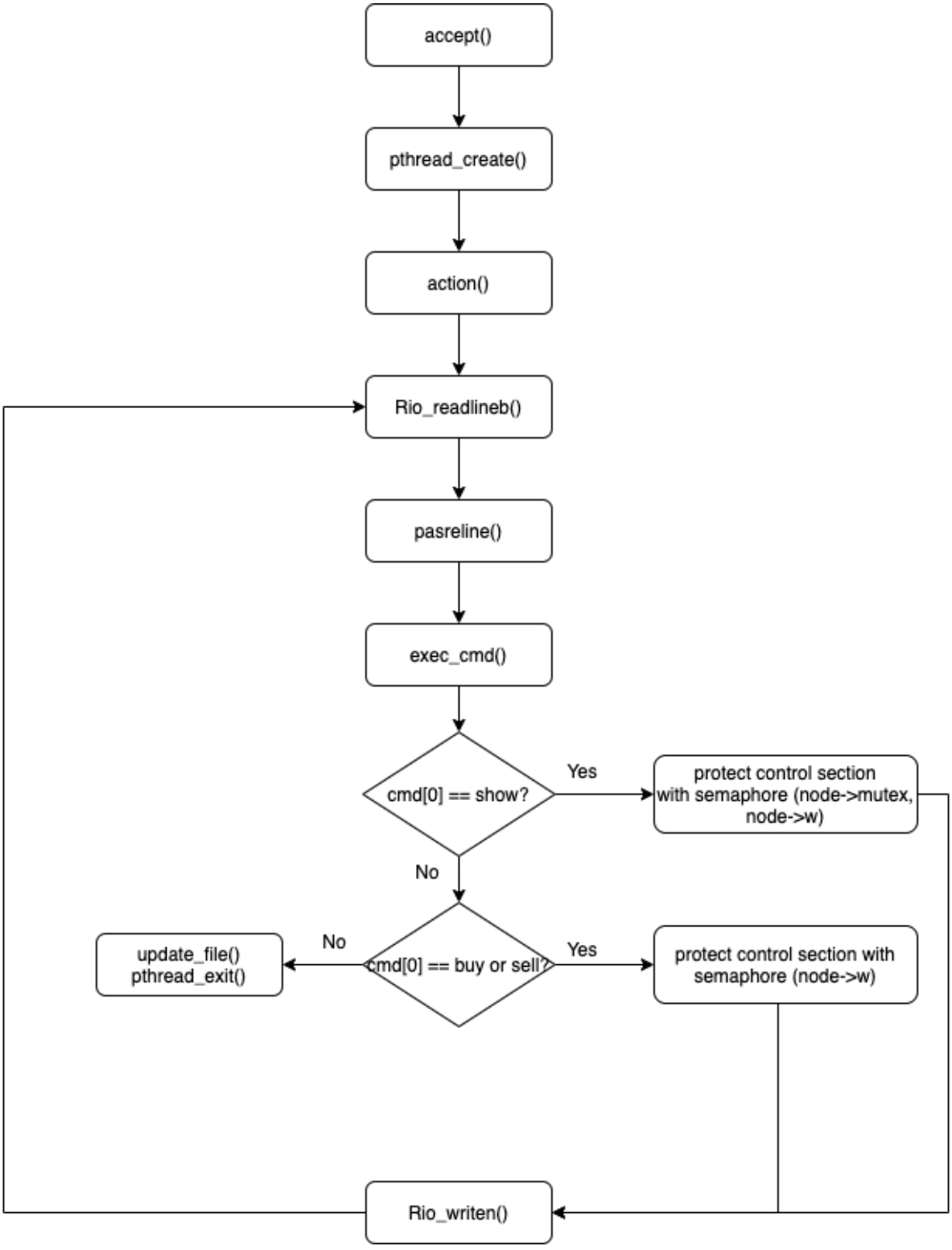
3. 구현 결과

A. Flow Chart

1. select



2. pthread


accept()
pthread_create()
action()
Rio_readlineb()
pasreline()
exec_cmd()
cmd[0] == show?
Yes
protect control section with semaphore (node->mutex, node->w)
No
cmd[0] == buy or sell?
No
update_file() pthread_exit()
Yes
protect control section with semaphore (node->w)
Rio_writen()

B. 제작 내용

1. select

select() : select 함수의 인자로 input request를 지켜보는 fd_set을 넘겨주는데, 이를 위해 fd_set temp를 별도로 선언한다. 기존의 fd_set in은 in의 fd가 set되어 있는지를 검사해 set되어 있는 fd에 한해 temp의 fd를 검사한다. 즉, in은 server가 관심 있게 지켜보는 (connect된) client의 fd 집합를 뜻하고, temp는 select()를 통해 input request가 들어온 client의 fd 집합을 뜻한다. listenfd는 사전에 in과 temp 모두에 set 해놓는다.

action() : Rio_readlineb()를 통해 입력을 받고 parseline()을 통해 parsing하고 exec_cmd()를 통해 적합한 결과 문자열을 생성하여 Rio_writen()으로 client에게 출력한다. exec_cmd()의 리턴값이 -1이거나 Rio_readlineb()의 리턴값이 0이라면 -1을 반환한다.

parseline() : 입력 받은 문자열이 줄바꿈문자라면 바로 리턴한다. 공백(' ')을 기준으로 잘라 저장한다.

exec_cmd() : parseline()에서 parsing한 문자열의 0번 index를 기준으로 구분한다. 만약 exit이라면 -1을 리턴한다. show라면 in_traverse()를 호출한다. buy나 sell이라면 1번 index(ID)의 left_stock을 2번 index(amount)만큼 감소 또는 증가시키도록 계산하여 change_stock()을 호출한다. 그 전에 buy에서 계산한 결과가 0보다 작다면 호출하지 않는다.

✓ 개발상 발생한 이슈

client에서도 Rio_readlineb()를 통해 server의 출력을 읽어들이는데, show의 경우 한 줄이 하나의 주식 node의 정보를 담고 있다. 따라서 '₩n'을 통해 구분이 되는데 Rio_readlinb()를 한번만 호출하면 맨 처음의 주식 node의 정보만이 출력될 것임이 예상되었다. 이 문제를 아래의 시도들을 통해 해결하였다.

1) Rio_readnb()로 변경 : Rio_readnb()를 호출하도록 변경하였고 그에 따라 문자열의 마지막에 '₩0'을 삽입하도록 했다. 하지만 Rio_readnb() 내부에서 호출하는 rio_read() 함수에서, 이미 EOF까지 읽어들여 rp->rio_cnt가 EOF임에도 0이 아니게 되어 read()를 다시 호출하고, 이미 입력 받은 모든 값을 읽어 들였기 때문에 client는 server의 입력을 기다리고, server도 client의 입력을 기다리는 deadlock 상황이 발생했다.

2) show의 줄 바꿈 문자('₩n')를 탭 문자('₩t')로 변경 : 줄 바꿈 문자로 인해 Rio_readlineb()의 문제가 생기는 것이므로 줄 바꿈 문자를 탭 문자로 변경하고,

client측에서 탭 문자가 있다면 그것을 다시 줄 바꿈 문자로 변경하게 해 문제를 해결하였다.

2. **pthread**

action() : event-based server에서 하나의 요청 만을 처리하는 함수에서 thread가 client의 connection 종료까지 처리하는 함수로 변경하기 위해 Rio_readlineb()에 해당하는 조건문을 반복문으로 변경하였다.

in_traverse() : 해당 node의 값을 읽는 코드를 control section(CS)으로 보호하였는데, node의 멤버 변수인 w(writer를 위한 semaphore), mutex(모두에게 해당되는 semaphore), readcnt를 사용하였다. node를 읽기 시작할 때 readcnt를 변경하기 위해 P(node->mutex)를 호출하고 CS에 진입을 성공했다면 readcnt를 1 증가시킨다. readcnt == 1이라면, 즉 해당 node를 읽는 첫 thread는 P(node->w)를 호출하여 node의 값 변경을 막는다. V(node->mutex)를 통해 다른 thread들도 read할 수 있도록 한다. 이후 CS를 빠져나오며 위의 과정을 반복하되 readcnt는 감소를, readcnt가 0이라면 V(node->w)를 통해 write이 가능하도록 한다.

change_stock() : 해당 주식 node의 잔여량을 변경하는 write에 해당하는 함수로, 변경하는 코드를 CS로 보호한다. P(node->w)를 통해 읽고 있는 thread나 이미 write 중인 thread를 기다리고 변경 후 V(node->w)를 호출한다.

✓ 개발상 발생한 이슈

Pthread_create()에 thread가 수행할 함수와 그 인자를 넘겨주는데, 인자를 넘길 때 connfd의 pointer를 넘기면서 race 상황이 발생했다. 그러면서 server가 A client로부터 요청을 받았지만 B client에게 해당 결과를 출력하는 상황이 벌어졌고, A client는 무한정 server를 기다리는 starvation 상황을 초래했다. 생성된 thread에서의 fd 값이 넘겨주었던 fd 값과 다르다는 것을 파악하고 별도의 지역 변수에 fd 값을 복사해 그 지역 변수를 넘겨주었다.

C. 시험 및 평가 내용

- **구현에 따른 성능 상의 차이 예측**

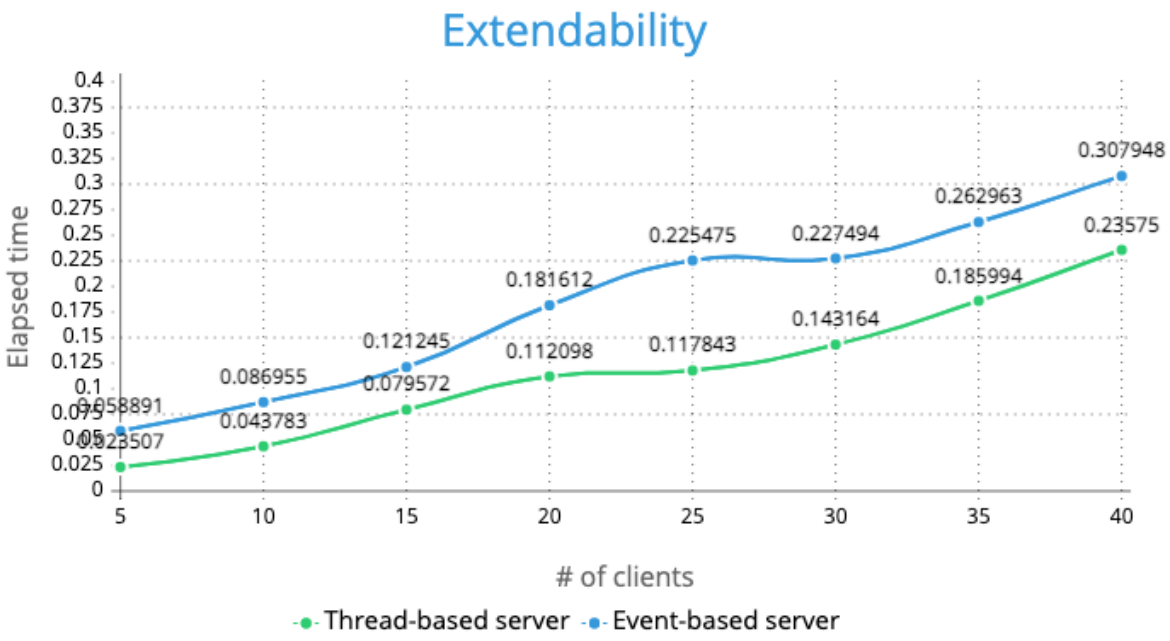
  select를 통해 구현한 event-based server는 client가 무한정 대기하지 않아도 된다는 점에서 concurrent하지만 하나의 control flow(process 또는 thread)가 모든 것을 담당하므로 한 번에 하나의 요청만을 처리할 수 있다는 점에서 parallel하지는 않다. 반면 thread-based server는 각 요청을 별도의 thread가 관리하므로 요청을 parallel하게 처리할 수 있다.

  이런 관점에서 전체적인 성능, 즉 일반적인 상황에서 client가 show, buy, sell 등 모든 종류의 요청을 할 때에는 thread-based server가 뛰어난 성능을 보일 것이다.

  show의 요청만을 한다고 가정했을 때, show는 2개의 semaphore를 사용하기 때문에 그에 대한 오버헤드로 event-based server보다는 뛰어나지만 성능 차이가 줄어들 것이다.

  buy, sell의 요청만을 한다고 가정했을 때, semaphore의 lock과 unlock에 대한 오버헤드가 semaphore를 2개 사용하는 show보다는 적기 때문에 thread-based server의 parallelism이 뚜렷한 효과를 보여 event-based server와의 성능 차이가 더 벌어질 것이다.

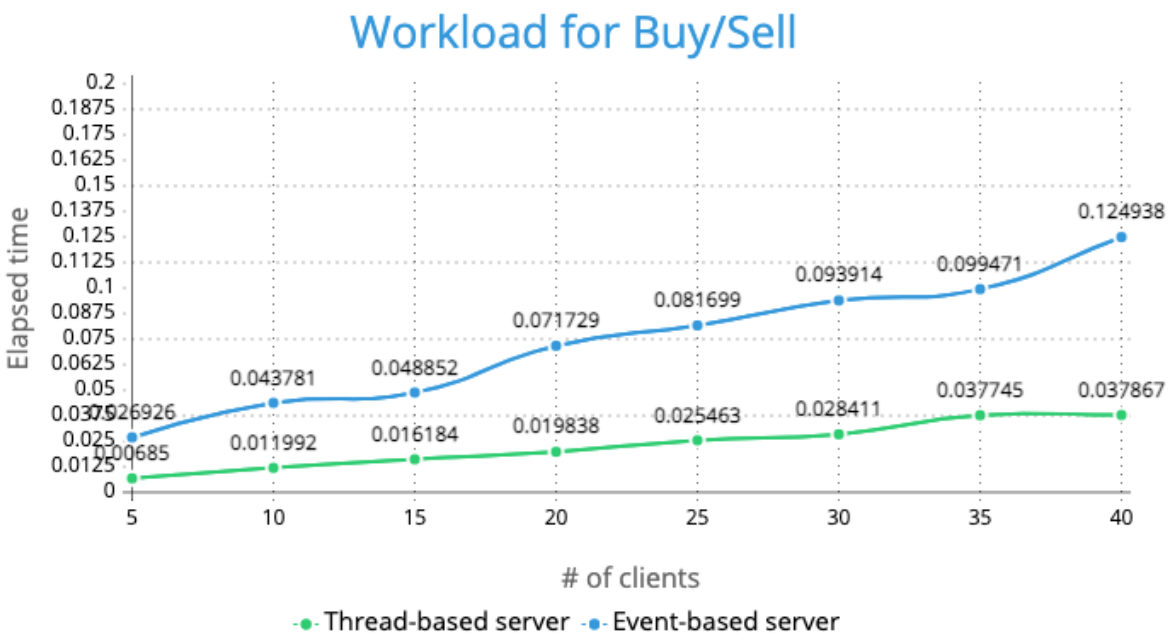
- **확장성 실험 결과**



  모든 요청을 한다고 가정했을 때, 즉 일반적인 상황에서 client의 수의 증가에 따른 성능 비교이다. thread-based server가 약 1.3배 ~ 2배 정도 뛰어난 성능을 보인다.

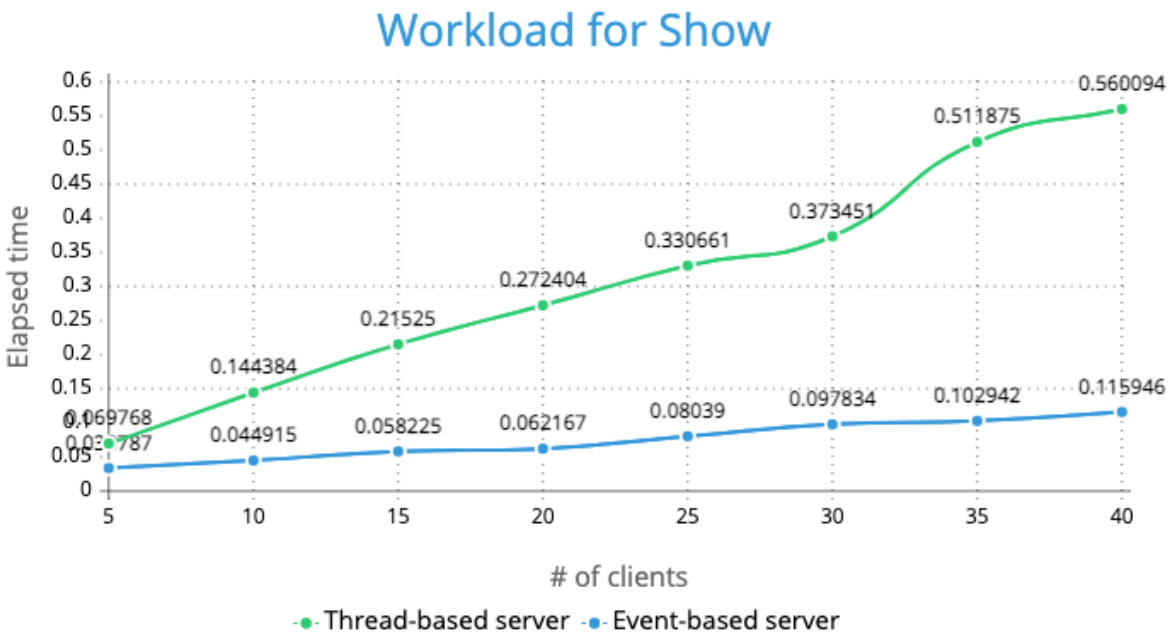
- **워크로드 실험 결과**

  1) 모든 client가 buy 또는 sell을 요청 (write)



가장 적은 오버헤드를 갖는 buy/sell의 요청만을 수행하기 때문에 Event-based server와의 성능 차이가 client가 증가함에 따라 더욱 심해지는 것을 관찰할 수 있다. thread-based server는 parallel한 구현이므로 client수의 증가에도 성능이 크게 저하되지 않는다. 그에 반해 event-based server는 client의 요청은 동시에 들어오지만 server가 그 처리를 병렬적으로 할 수 없어 client 수의 증가에 따라 점점 저하가 심해지는 것을 알 수 있다. 이는 parallelism의 성능을 가장 잘 확인할 수 있는 결과이다.

2) 모든 client가 show를 요청 (read)



show 요청에서 사용하는 2개의 semaphore에 대한 오버헤드로 인해 오히려 event-based server가 훨씬 뛰어난 성능을 보이는 것을 알 수 있다. 이 워크로드에 대한 thread-based server의 성능은 client가 많아질수록 크게 저하된다. 그만큼 semaphore의 lock에 걸려 control section에 진입하지 못하고 대기하는 thread가 많아지기 때문일 것이다.

이는 수업 시간 중 배운 mutex의 오버헤드로 인해 오히려 느려진다는 내용과도 일치하는 결과이다. show 즉, read의 워크로드가 많은 server-client 구조라면 event-based server로 구현하는 것이 유리할 것이다.

- **Shark machine with 8 cores, n=2^{31}**

Threads (Cores)	1 (1)	2 (2)	4 (4)	8 (8)	16 (8)
psum-mutex (secs)	51	456	790	536	681

- **Nasty surprise:**
  - **Single thread is very slow**
  - **Gets slower as we use more cores**