Multicore Programming Project 2

담당 교수 : 박성용

이름 : 김도현

학번 : 20181256

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

해당 프로젝트는 network programming 중 concurrent server에 대해 이해하고자 concurrent stock server에 대해 event-driven approach, thread-based approach 이 두가지 방법을 통해 구현해보는 것이다. 여러 client sever 가 동시에 stock server 로 접속을 해서 buy, sell, show, exit 과 같은 적절한 명령어들이 concurrent 하게 처리될 수 있어야 한다. 해당 주식 정보는 stock.txt 파일에 미리 저장되어 있고 이 정보들은 binary tree 로 각 id별 각 node에 저장되어 있다. 해당 stock.txt 파일은 stock server가 종료될 때 update 되어 consistency 를 유지해야 한다. 최종적으로 두 approach에 대해 성능 평가 및 분석이 이루어져야 한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach

   Event-driven approach는 오직 한개의 process를 사용해서 concurreny 한 server를 만든다. Pool 구조체에 clientfd들을 저장하도록 한다. 그리고 select 함수를 통해서 어떤 fd 가 준비가 되었는지 확인을 하고 main() 에서 반복문을 돌며 Accept 함수를 통해 새로 들어오는 client 를 확인하고 확인 시 connfd를 add_client 해서 pool에 넣고 check_clients 함수에서 순차적으로 연결된 connfd들에 대한 명령어들을 수행하면 된다. Check_clients 에서는 pool struct 에서 maxi 및 p->nready>0 인 것들만 반복문을 통해 돌며 세부 명령어들이 수행된다. 이는 reader_writer 문제가 없어 명령어별로 동시처리율이 비슷하고 thread 에 대해 overhead가 발생하지 않지만 결국 한 프로세스의 select 함수를 통해 반복문을 순차적으로 돌아 처리 측면에서 엄청 빠르지는 않았다.

2. Task 2: Thread-based Approach

Thread based approach는 여러 thread들을 생성해서 실제 concurrent 하게 Server를 만드는 것이다. 우선 sbuf_t 구조체를 만들어 미리 동시에 thread들을 생성해 놓고 client 요청이 들어올 때 마다 thread를 만들기 보다는 미리 만들어 놓은 Thread(pre-threaded)를 sbuf_insert, sbuf_remove 함수의 semaphore 기능을 통해 buf에서 하나씩 꺼내 해당 thread를 바로 이용할 수 있도록 해야 한다. 그리고 thread의 경우 전역변수를 동시에 접근 할 수 있기 때문에 Read, write 문제가 생기게 된다. 이를 해결 하기 위해 read 시 다른 Thread들이 write을 못하게 막고 write 시 다른 thread 들이 read를 못하게 semaphore를 활용해야 한다. 필자는 first readers-writers 를 통해 read를 우선적으로 하도록 만들어주었다. 결과적으로 show는 read 만 하는 명령어이기 때문에 속도가 다른 명령어들에 비해 빠르다는것을 알 수 있었다.

3. Task 3: Performance Evaluation

해당 임무는 성능 테스트이다. Multiclient.c 에서 gettimeofday 함수를 이용해 (end-start)시간 계산을 한다. 우선 두 approach를 concurrent client 개수로 비교를 해보았다. Client 개수가 10, 20, 30, 50, 100 이렇게 5부분으로 나누어서 분석해본다. 그리고 워크로드에 대한 분석 역시 필요하다. Read write 문제에 매우 중요한 fine-grained locking이 되었는지, 어느정도 영향을 미치는지 확인 해야 하기 때문이다. 그래서 명령어를 buy, sell 만 했을 때, show 만 했을 때, buy, sell, show를 모두 random 으로 했을 때 등으로 나누어서 분석비교를 한다.

여기서 시간당 client 처리 개수인 동시 처리율을 결정해야 하는데 필자는 client 개수 * ORDER_PER_CLIENT / (실행 시간 us) 으로 결정하고자 한다. 상대적인 분석이기 때문에 추가적으로 상수 값을 곱해 조금 더 그래프가 눈에 들어오기 쉽게 그려본다.

B. 개발 내용

- **아래 항목의 내용만 서술**
- **(기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)**
- **Task1 (Event-driven Approach with select())**

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

해당 event-driven approach 에서는 하나의 프로세스를 사용하기 때문에 pool struct를 사용하지 않으면 새로운 client가 server와 연결될 때 마다 connfd 가 계속해서 변경 되서 실제 요청하는 client 명령어가 수행이 안될 가능성이 있다. 그래서 pool 구조체에 clientfd라는 배열에 저장하고 있어야 한다. Fd_set의 Read_set은 bitmap 형식인데 add_client 마다 계속 update가 되고 select 함수 실행 시 해당 fd에 요청이 일어난 connfd가 1로 바뀌고 나머지는 0으로 return 되기 때문에 미리 read_set을 복사해 select를 이용할 수 있는 fd_set ready_Set 역시 필요하다. 그리고 해당 개수를 p->nready에 넣어 check_client 함수에서 for문의 범위를 구체화할 때 사용할 수 있다. FD_ISSET에서 listenfd가 ready_set 에 있는지 체크를 해보고 있다면 Accept 함수와 add_client 함수를 통해 connfd 값을 받고 해당 Fd 값을 pool 에 넣어 계속해서 concurrent 하게 여러 client server가 들어올 수 있도록 해준다.

✓ epoll과의 차이점 서술

select 함수는 read_set(ready_set)이 계속해서 check_client 함수나 add_client 함수에서 변경되기 때문에 pool 구조체의 모든 fd를 다 돌면서 요청 여부를 확인해야 한다. 때문에 시간적인 면에서 비효율적이라고 할 수 있다. 그래서 이런 단점을 보완하기 위해 Epoll은 운영체제에서 fd를 직접적으로 관리한다. 만약 요청이 들어오게 된다면 epoll_fd 를 return 하여 요청을 할 수 있다. 또한 요청이 일어난 fd 들은 따로 빼서 해당 fd만들을 따로 관리를 해주기도 한다. Kernel 영역에서 일어나기 때문에 select 함수보다 더 빠르지만 상대적으로 더 복잡하게 구현을 해야 한다. 우리의 프로젝트는 concurrent한 server를 구현 하는 것이기 때문에 더 쉽게 구현 가능한 select 함수를 사용하지만 결과적으로 프로세스가 계속해서 kernel 에 요청된 fd 값을 return 받아야 한다는 점에서 큰 관점으로 보면 동기화 개념은 비슷하다 볼 수 있다.

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

✓ Master Thread의 Connection 관리

Master thread는 기준이 되는 thread로 main 함수에서 Open_listenfd 함수를 통해 listenfd를 받아온다. 그리고 원하는 개수만큼 sbuf_init 해주고 peer thread를 생성해야 한다. 이때 sbuf_t 구조체를 이용하기 위해 미리

pthread_create() 함수로 pre_thread를 생성해둔다. 반복문에서 accept 함수를 통해 connfd 를 받아와 해당 정보를 sbuf_insert 에 넣어준다. 그러면 이제 thread 함수에서 sbuf_remove 함수를 통해 sbuf에 들어있는 connfd 값을 하나씩 가져와 action 함수에서 해당 client가 원하는 명령어를 수행할 수 있도록 해주고 Exit과 같은 명령어 등을 통해 connfd를 close 하게 된다. 이때 Pthread_detach 함수를 이용해서 굳이 master thread 가 따로 peer thread를 종료시켜 주지 않아도 thread 함수가 끝나면 자동으로 사라지도록 해준다. 추가적으로 master thread 에서는 sigint_handler 와 같이 해당 thread가 종료될 때 stokc.txt 에 지금까지 일어났던 명령어들에 대한 결과값을 업데이트 해주는 과정이 필요하다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

Worker thread 는 sbuf_t 구조체에 의해서 관리가 된다고 볼 수 있다. Sbuf_t 는 circular queue 로 생각해 볼 수 있다. 기존의 방법 같은 경우 connfd 가 발생할 때 마다 thread를 생성했지만 thread 를 생성할 때 생기는 overhead 를 줄이기 위해서는 미리 pre_thread 를 만들어 놓아야 한다. 이때 sbuf 에 대해 자세히 살펴보면 sem_t mutex, sem_t slots, sem_t items 라는 semaphore 변수가 있다. 초기화 할 때 sbuf_init 함수에서 mutex는 1, 해당 Buf에 공간이 얼마나 남았는지 확인하는 slot은 n, 해당 buf에 현재 item이 몇개나 들어왔는지 확인하는 item은 0으로 해둔다. Mutex는 해당 값을 삽입하거나 지우는 업데이트 과정에서 사용하는 여러 thread가 동시에 write 하지 못하도록 막는 세마포어이다. sbuf_insert 함수, sbuf_remove 함수는 producer, consumer 라고 볼 수 있는데 sbuf_insert에서는 p(&sp->slot)을 통해 사용가능한 slot을 점차 줄여나가 slot이 0이 되면 더이상 접근하지 못하도록 막는다. 그리고 v(&sp->items)을 통해 Item 값은 늘려나간다. thread 함수에서의 sbuf_remove 함수는 처음에 p(&sp->items) 에서 초기의 item이 0이기 때문에 계속 대기가 걸려 있다가 items가 증가하는 순간 buf에서 front, rear를 통해 해당 item을 가져오고 삭제 시킨다. 이때 역시 mutex를 활용해 다른 thread가 동시 접근하지 못하도록 막아야 한다. 그리고 V(&sp->slots)으로 slot 을 1 늘려 다시 sbuf_insert에서 slot 공간이 생겼음을 알릴 수 있다. 최종적으로 sbuf를 clean 하기 위해서 sbuf_deinit 함수를 수행하면 된다.

추가적으로 해당 각 Thread에서는 주식 node들을 동시에 접근할 수 있다. 이 역시 Semaphore가 필요해서 각 노드마다 sem_t mutex, int readcnt, sem_t

w_lock 변수를 활용해서 read, write 할 때 적절하게 구현을 해주어야 한다. 해당 내용은 worker thread pool과는 약간 거리가 있기 때문에 c. 개발방법에서 다시 적도록 하겠다.

- **Task3 (Performance Evaluation)**

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

  우리는 event-driven approach 와 thread-based approach에 대한 성능 분석을 해야 한다. 결국 누가 더 concurrent 하게 돌아가는지 확인하기 위해서는 동시처리율이 중요하다. 이는 client 개수 * ORDER_PER_CLIENT / (실행 시간 us) 으로 계산해서 1초당 얼마나 처리할 수 있는지 상대적인 기준을 정할 수 있다. 즉 전체 요청하는 양을 실행시간으로 나눈 것이다. 추가적으로 상수를 곱해서 조금 더 시각화를 할 수 있다.

  Multiclient.c 에서 gettimeofday 함수를 이용해 (end-start)시간 계산을 한다. Client 개수를 10, 20, 30, 50, 100 이렇게 5개로 나눌 것이다. 그리고 요청 개수 역시 10, 20, 50 이렇게 늘려볼 것 이다. 이 경우 총 5*3 15개를 비교 분석하면 된다.

  그리고 어떤 명령어가 실제 concurrent하게 동작하고 혹은 lock이 되어 있는지도 판단할 수 있다. 이는 Read write문제라고 할 수 있는데 매우 중요한 fine-grained locking이 되었는지, 어느정도 영향을 미치는지 확인해본다. 그래서 명령어를 buy, sell 만 했을 때, show만 했을 때, buy, sell, show를 모두 random 으로 했을 때 등으로 나누어서 분석비교를 한다.

  마지막으로 thread-based approach의 경우 Thread 개수를 얼마만큼 두는지 NTHREAD 를 바꿔가면서 thread 개수에 따른 분석 역시 가능할 것으로 보인다.

  ✓ Configuration 변화에 따른 예상 결과 서술

  Event-driven approach의 경우 실질적으로 프로세스 하나에서 모든 일이 일어나기 때문에 client 개수가 늘어나든 order가 늘어나든 결국 초당 해결해야 하는 동시처리율은 크게 변화가 없을 것으로 예상된다. 하지만 thread-based approach 같은 경우 client 개수가 많을수록 sell, buy 와 같은 명령어를 이용

시 read, write문제를 해결 하는 semaphore에 의해 lock이 걸리는 부분이 많이 생길 수 있다. 이 경우 실제 thread 들이 concurrent 하게 실행되지 못하기 때문에 속도가 느릴 수 있다. 하지만 show 만 실행하는 경우 read만 하기 때문에 실제로 thread들이 concurrent 하게 진행이 된다. 그래서 실행 시간이 상대적으로 더 짧을 것으로 예상된다.

N thread 역시 thread개수가 많아질수록 동시에 돌아가는 thread 개수가 더 많아진다는 것이기 때문에 더 실행 시간이 짧아지지만 이는 client 개수를 초과 시 큰 차이가 없을 것 같다.

C. 개발 방법

- **B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

해당 Approach 에 대한 설명을 하기에 앞서 공통적으로 해당하는 부분에 대해 먼저 서술하겠다. 우선 task1, task2를 위해 최종적으로 우리가 건드려야 할 코드는 stockserver.c 파일이다. 그리고 미리 stock.txt 파일을 요구하는 형식대로 만들어 놓아야 한다. 해당 파일에서는 처음에 주식 구조체인 items 에 주식들을 순차적으로 넣는 과정이 필요하다. 우선 main 함수에서 stock.txt 파일을 읽어들여 putNode 함수를 실행시킨다. 이는 stock ID를 기준으로 root보다 작으면 left child, 크면 right child로 이동하는 binary tree를 만들어주는 함수이다. 추가적으로 exit 명령어는 connection을 끊는 명령어로 인식되도록 해주는데 client는 종료가 되도 stockserver는 계속해서 다른 client를 받아야 된다고 생각을 했다. 그래서 server를 명시적으로 종료시킬 수 이도록 ctrl+c 인 sigint_handler를 등록하고 해당 함수에서 종료 시 새로운 update 된 node들을 하나씩 읽어들어 stock.txt에 적도록 코드를 구성하였다. 마지막으로 읽을때마다 해당 node는 free 시켜주어 메모리 해제도 신경 써야 한다.

Event-based Approach

Event based server의 가장 큰 핵심은 pool 구조체를 구성하는 것이다. 해당 구조체에서는 connfd를 저장하고 있어 select 될 때 해당하는 fd들의 요청 명령어들을 수행시켜 준다. 세부적으로 보면 connfd 를 추가하고 탐색할때 범위를 지정해 주는 maxfd, maxi가 있다. 그리고 비트맵 형식인 fd_Set read_set이 있다. 해당 변수는 지금까지 추가되어 있는 fd들을 0,1 형태로 저장하고 있다. 그리고 fd_set ready_set은 실제 select 함수가 실행되고 나서의 선택된 fd들을 가지고 있는 bitmap이다. Nready는 select 된 fd 개수를 의미하고 clientfd와 clientrio는 각 실제 fd 값과 해당 fd의 buffer를 의미한다고 볼 수 있다.

처음에 Open_listenfd 함수를 실행하여 listenfd를 받아오고 해당 fd를 init_pool 함수를 통해 pool을 초기화 시킨다. 그 다음 while 무한 loop를 들어가는데 다음과 같은 반복문이 시작한다. 우선 read_Set을 ready_set에 복사하고 Select 함수를 통해 요청이 있는 fd들을 찾아보고 이를 ready_set에 저장하고 개수를 nready에 return 한다. 그 다음 FD_ISSET에서 listenfd에서 요청이 왔다면 Accept 함수를 통

해 conntfd를 찾고 실제 connection을 시켜준다. 다음 add_client 함수에 해당 connfd를 pool에 넣어둔다. 다음 check_clients 함수를 실행하는데 여기서는 client들의 명령어들이 수행된다. 이때 for문을 pool을 순차적으로 하나씩 도는데 i<=p->maxi, p->nready>0 처럼 위에서 수정된 조건들을 참고하여 필요 없는 pool을 돌지 않도록 하여 overhead를 줄인다. FD_ISSET에서 이번엔 connfd가 있는지 확인해본다. 있다면 해당 buf를 읽어들인다. 만약 show 라면 binary tree에 있는 node 값들을 buf[maxline]에 하나씩 저장하여 Rio_writen 함수로 client에 결과를 출력해준다. Buy, sell의 경우 들어오는 stock ID 값을 기준으로 해당 NODE를 찾아야 한다. 이때 우리는 기준 node보다 id가 작으면 왼쪽 크면 오른쪽으로 가도록 구성했기 때문에 반복문을 통해 node->id==id 일 때 멈추면 된다. 이때 buy의 경우 사기를 원하는 개수보다 남아있는 개수가 적다면 사지 못하기 때문에 남은 주식이 충분하지 않다고 return 해줘야 하고 node를 업데이트를 시키지는 않는다. 그리고 만약 exit이 들어왔다면 disconnect 명령어를 출력하고 connfd를 Close 하면 된다.

Thread-based approach

여기서는 semaphore와 sbuf_t 구조체가 중요하다. Sbuf_t 구조체는 pre_thread를 구현할 때 사용하는 것인데 circular queue 구조이지만 핵심은 sem_t mutex, sem_t slots, sem_t items라고 볼 수 있다. Sbuf_init 과정에서 mutex는 buf에 item을 넣거나 삭제할 때 다른 thread가 접근하지 못하도록 막는 semaphore로 binary로 구현되어야 해서 sem_init(&sp->mutex,0,1)로 설정한다. 그리고 Slot은 해당 buf에 들어갈 수 있는 공간이 얼마인지 확인하는 Semaphore로 sbuf_insert 시 P() 함수를 통해 하나씩 줄어든다. 그래서 이는 sem_init(&sp->slots,0,n) 으로 최대로 들어올 수 있는 item 개수인 n으로 초기화를 해주어야 한다. Items는 현재 Buf에 들어가 있는 item이 몇개인지 확인하는 변수로 sbuf_insert함수에서 최종적으로 종료시 v() 함수를 통해 items++을 해준다. sbuf_remove에서는 P(&sp->items) 를 통해 items—를 해주는데 이를 통해 thread들이 미리 생성되어서 connfd 발생 시 thread를 만드는 것보다 overhead를 줄여 더 빠르게 구현 할 수 있도록 해준다.

우선 main함수에서 먼저 Pthread_create 함수를 통해 원하는 만큼 thread를 생

성하고 client가 connection을 요청할 때 sbuf_insert 함수를 통해 connfd가 들어왔음을 thread에 알린다. Thread는 sbuf_remove함수에서 P(&sp->items)에서 멈춰있는데 이가 실행되면서 connfd를 하나 가져오고 이를 제거해준다. 그리고 action 함수가 실행된다. 추가 과정은 B.개발 내용과 비슷하기 때문에 해당 내용에서 조금 더 자세히 설명하기로 한 read_write문제에 대해 서술하겠다.

우선 show는 해당 전역변수인 binary tree node들을 읽기만 하면 된다. Node들 역시 readcnt, sem_t mutex, sem_t w_lock 변수를 가지고 있는데 first readers-writers문제를 위한 것이다. 우선 readcnt를 수정하기 위해서 mutex가 필요하다. Readcnt는 현재 node를 읽고 있다는 뜻으로 만약 readcnt가 처음 1로 바뀌면 p() 를 통해 w_lock을 걸어준다. 그러면 그때 모든 sell,buy와 같은 write을 하고자 하는 모든 thread에 lock이 걸린다. 그리고 추가적으로 show명령어로 read를 한다면 계속해서 write을 못하기 때문에 starvation이 발생할 수도 있다. 어쨌든 read가 끝나면 다시 mutex로 해당 thread만 readcnt를 수정할 수 이도록 해주고 readcnt==0이면 다시 write을 할 수 있도록 v() 로 lock을 풀어준다.

Buy와 sell 명령어에서는 write하는 명령어이기 때문에 P(&w)로 w이 1 일 때만 해당 thread가 write을 할 수 있도록 수정해야 한다. 그리고 다 읽고 나서 V(&w)로 다시 풀어주면 된다. 이때 buy 명령어는 우선 남은 재고와 비교를 해야 하기 때문에 read를 먼저 해주는데 show 명령어와 비슷하게 짜서 stock->left_stock을 읽어 들이면 된다. Exit 시 해당 action 함수를 종료 시키고 thread에서 close 함수로 connection을 끝내고 미리 설정해 놓은 Pthread_detach 함수를 통해 자동으로 thread가 종료된다. 세부 명령어 내용들은 위의 event-based approach와 일치한다.

비교 분석

결과적으로 분석을 하기 위해서는 multiclient.c를 수정해야 한다. 우선 project2 명세서에 나온 것처럼 struct timeval start, struct timeval end 를 설정해 gettimeofday(&start,0), gettimeofday(&end,0) 사이에 동시 요청이 일어나는 client 구현에 대한 코드를 넣는다. (While(runprocess<num_client) 와 waitpid 까지) 그리고 eval_usec 을 구해 이를 출력해보는 과정으로 해당 파일을 수정해보았다. 자세한 실험결과는 4번 성능평가 결과에 기술 할 예정이다.

3. 구현 결과

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

  이미 위에서 계속해서 설명했던 대로 구현을 하니 적절하게 event-based approach, thread-based approach 모두 정상적으로 작동함을 확인 할 수 있었다. 이때는 기본 multiclient.c를 통해서 multiclient수를 4개로 두고 확인을 할 수 있었다. 다만 한가지 고려하지 못한 점이 exit에 대한 처리이다. 초기에는 exit 함수에서 freeNode 까지 구현을 해서 한번 multiclient를 돌릴때는 전혀 문제가 없지만 server는 계속 연결을 기다리고 있고 거기서 계속해서 client가 들어올 때 Node가 더이상 존재하지 않아 문제가 생겼다. 이를 위해 freeNode는 최종적으로 server가 ctrl+c를 통해 종료될 때만 free 시켜주는 것으로 코드를 수정하였다.

-

4. 성능 평가 결과 (Task 3)

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

1. 확장성(동시적 client 수 변화)

- 측정 시점: multiclient.c 에서 fork 실행 전 while위에서 gettimeofday(&start,0), waitpid 종료 후 gettimeoday(&end,0)까지

- 실행 환경: visual studio code ssh 환경

- 단위: 실행시간 측정 us

- 동시 처리율: 1000*order_per_client*multiclient/실행 시간(us)

- thread_based approach 의 경우 만들어져 있는 thread는 500개로 통일시켰다.

(1) Order_per_client=10, 일 때 캡쳐 값이다.

Event-driven approach

```
Elapsed time[Event driven]: 28329 us (order_per_client: 10, multiclient: 10)
Elapsed time[Event driven]: 30400 us (order_per_client: 10, multiclient: 20)
Elapsed time[Event driven]: 35226 us (order_per_client: 10, multiclient: 30)
Elapsed time[Event driven]: 54201 us (order_per_client: 10, multiclient: 50)
Elapsed time[Event driven]: 104825 us (order_per_client: 10, multiclient: 100)
```

thread-based approach

```
Elapsed time[Thread based]: 24429 us (order_per_client: 10, multiclient: 10)
Elapsed time[Thread based]: 28776 us (order_per_client: 10, multiclient: 20)
Elapsed time[Thread based]: 38233 us (order_per_client: 10, multiclient: 30)
Elapsed time[Thread based]: 46225 us (order_per_client: 10, multiclient: 50)
Elapsed time[Thread based]: 86510 us (order_per_client: 10, multiclient: 100)
```

(2) Order_per_client=20, 일 때 캡쳐 값이다.

Event-driven approach

```
Elapsed time[Event driven]: 33929 us (order_per_client: 20, multiclient: 10)
Elapsed time[Event driven]: 38353 us (order_per_client: 20, multiclient: 20)
Elapsed time[Event driven]: 61235 us (order_per_client: 20, multiclient: 30)
Elapsed time[Event driven]: 83427 us (order_per_client: 20, multiclient: 50)
Elapsed time[Event driven]: 154962 us (order_per_client: 20, multiclient: 100)
```

thread-based approach

```
Elapsed time[Thread based]: 28394 us (order_per_client: 20, multiclient: 10)
```

```
Elapsed time[Thread based]: 40932 us (order_per_client: 20, multiclient: 20)
```

```
Elapsed time[Thread based]: 57718 us (order_per_client: 20, multiclient: 30)
```

```
Elapsed time[Thread based]: 83216 us (order_per_client: 20, multiclient: 50)
```

```
Elapsed time[Thread based]: 153564 us (order_per_client: 20, multiclient: 100)
```

```
Elapsed time[Event driven]: 87239 us (order_per_client: 50, multiclient: 20)
```

```
Elapsed time[Thread based]: 185870 us (order_per_client: 50, multiclient: 50)
```

```
[buy] success
Elapsed time[Event driven]: 122415 us (order_per_client: 50, multiclient: 30)
```

(3) Order_per_client=50, 일 때 캡쳐 값이다.

thread-based approach

```
Elapsed time[Event driven]: 50610 us (order_per_client: 50, multiclient: 10)
```

```
Elapsed time[Event driven]: 78462 us (order_per_client: 50, multiclient: 20)
```

```
Elapsed time[Event driven]: 114520 us (order_per_client: 50, multiclient: 30)
```

```
Elapsed time[Event driven]: 190569 us (order_per_client: 50, multiclient: 50)
```

```
Elapsed time[Event driven]: 372301 us (order_per_client: 50, multiclient: 100)
```

Event-driven approach

```
Elapsed time[Thread based]: 46813 us (order_per_client: 50, multiclient: 10)
```

```
Elapsed time[Thread based]: 81281 us (order_per_client: 50, multiclient: 20)
```

```
Elapsed time[Thread based]: 113866 us (order_per_client: 50, multiclient: 30)
```

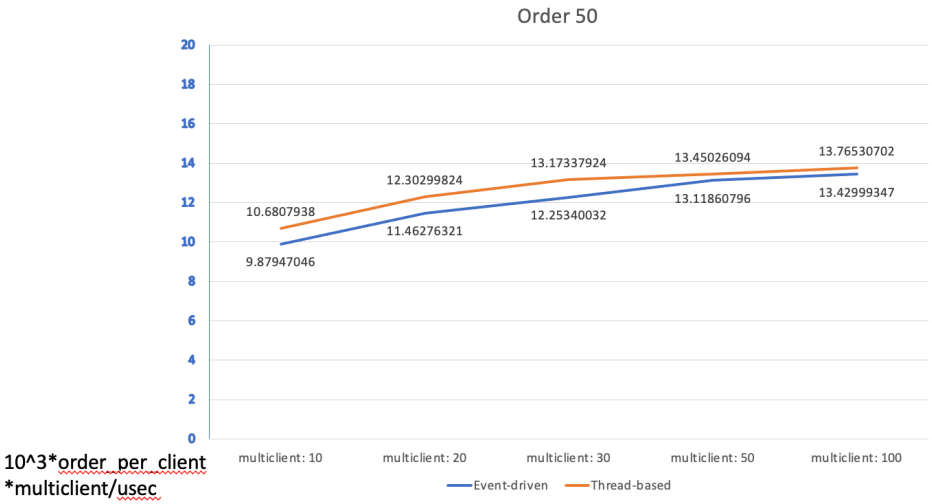
```
Elapsed time[Thread based]: 196546 us (order_per_client: 50, multiclient: 50)
```

```
[Sell] success
Elapsed time[Thread based]: 363232 us (order_per_client: 50, multiclient: 100)
```

다음은 order_per_client 기준으로 그래프를 그려보았다.

10^3*order_per_client *multiclient/usec

10^3*order_per_client *multiclient/usec



분석

우선 order_per_client가 10일 때 그래프부터 보겠다. 이 그래프에서는 동시 처리율이 thread_based, event-driven 둘 다 우상향하는 모습을 볼 수 있다. 이는 우선적으로 동시 처리율 계산이 명령어 개수*client 개수/실행시간(us) 이기 때문에 결국 명령어가 오른쪽으로 갈 수록 많아 지기 때문에 시간 대비 처리하는 속도가 빨라 짐을 알 수 있다.

하지만 thread_based 의 경우 기울기를 보면 multiclient가 증가함에 따라 급속히 증가하는 부분이 보인다. 이는 thread_based 가 실제 동시에 여러 Thread를 돌리고 있어 명령어들에 대한 처리를 한번에 했다고 유추 할 수 있다. 그에 비해 event-driven 같은 경우 결국 select 함수에 의해 명령어가 들어온 fd 들을 찾고 이후 while 반복문에서 하나 씩 배열을 순차적으로 돌면서 이를 수행하기 때문에 큰 변화를 보이지 않은 것을 확인했다.

Order_per_client가 20일 때와 order_per_client가 50 일때 역시 이와 비슷한 양상을 보이는데 multiclient가 50이상이 되면 성능이 thread_based 와 event_driven 이 비슷해지는것을 관찰 할 수 있다. 여러가지 요인들이 있겠지만 명령어 개수가 엄청 많아진다는 의미이고 이는 thread_based의 한계점인 read_write 문제나 sbuf 구조체 사용할 때의 semaphore 문제의 Overhead가 커졌다고 유추할 수 있다. 우리가 sell, buy 했을 때의 경우 다른 thread 들은 모두 멈춰 있어야 하는 상황이 생긴다. 또한 그 시간 뿐만 아니라 semaphore 의 lock을 풀고 잠그는 overhead 역시 누적이 될수록 커졌다는 것을 예측할 수 있다.

전체적으로 보면 일단 thread_based 의 성능이 event_driven 보다 더 좋게 나왔음을 알 수 있다. 이는 thread_based 는 실제 여러 thread 들이 동시에 돌아가게 하기 위해 세세한 조정을 통해 fine_grained concurrency를 만족시키고 있다. 하지만 event_driven은 결국 한 process가 select 함수 및 반복문을 통해 모든 명령어를 수행하는 것이다. 이를 반대로 생각해보면 thread_based 는 명령어 개수 및 multiclient 수에 따라 동시 처리율 변화 폭이 매우 큰 반면 event_Driven 은 상대적으로 더 안정적으로 명령어들을 처리하고 있다고 생각할 수 도 있다. 또한 일부 event-driven이 성능이 더 좋을 경우는 thread_based의 semaphore에 대한 overhead가 커진 상태인 것 같다.

2. 워크로드 분석(client 명령어 별로 분석)

- 측정 시점: multiclient.c 에서 fork 실행 전 while위에서 gettimeofday(&start,0), waitpid 종료 후 gettimeoday(&end,0)까지

- 실행 환경: visual studio code ssh 환경

- 단위: 실행시간 측정 us

- 동시 처리율: 1000*order_per_client*multiclient/실행 시간(us)

- thread_based approach 의 경우 만들어져 있는 thread는 500개로 통일시켰다.

event-driven approach

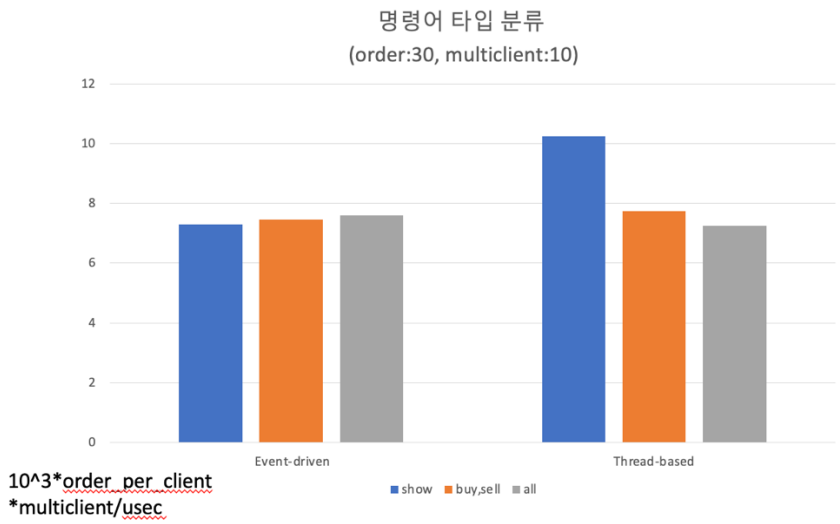
```
Elapsed time[Event driven]: 41125 us (order_per_client: 30, multiclient: 10 [show])
Elapsed time[Event driven]: 40237 us (order_per_client: 30, multiclient: 10 [sell, buy])
Elapsed time[Event driven]: 39426 us (order_per_client: 30, multiclient: 10 [all])
```

thread-based approach

```
Elapsed time[Thread based]: 29281 us (order_per_client: 30, multiclient: 10 [show])
```

```
[buy] success
Elapsed time[Thread based]: 38814 us (order_per_client: 30, multiclient: 10 [sell, buy])
```

```
Elapsed time[Thread based]: 41327 us (order_per_client: 30, multiclient: 10 [all])
```



분석

해당 분석은 명령어 종류에 따라 동시처리율이 어떻게 달라지는지 확인하기 위해서이다. 미리 이론에서 배워 예측한 것과 같이 매우 유사하게 나왔다. Thread_based의 경우 show를 보면 동시 처리율이 다른 buy,sell all 에 비해 매우 크게 나왔음을 알 수 있다. 그 이유는 show는 reader-writer 문제에서 reader 밖에 없기 때문에 모든 thread가 동시에 계속해서 해당 show 명령어를 수행할 수 있다. 반면 buy,sell 은 reader,(buy의 경우 코드 상 현재 남은 주식상태를 읽어야 하기 때문에 reader가 들어가 있다.) writer가 들어가 있기 때문에 해당 정보를 읽고 써야한다. 동시 접근 문제를 해결하기 위해 우리는 first readers-writers problem을 위해 reader 우선적으로 semaphore을 코드에 구성했다. 그래서 writer 시 해당 Node를 접근하고 있는 다른 모든 thread 들의 수행을 멈추게 만들고 이에 대한 Overhead 역시 매우 크다. 또한 reader가 많아질 수록 read 우선적으로 하고 write은 계속 뒤로 밀리기 때문에 속도를 더 늦추는 요인으로 판단 할 수 있다.

하지만 event_driven의 경우 process가 하나에서 명령어를 수행하기 때문에 이런 문제가 생기지 않는다. 그래서 예측한 바와 같이 명령어와 별개로 비슷한 동시처리율이 나왔음을 확인 할 수 있다.

3. thread-based에서 thread 개수에 따른 동시 처리율 변화

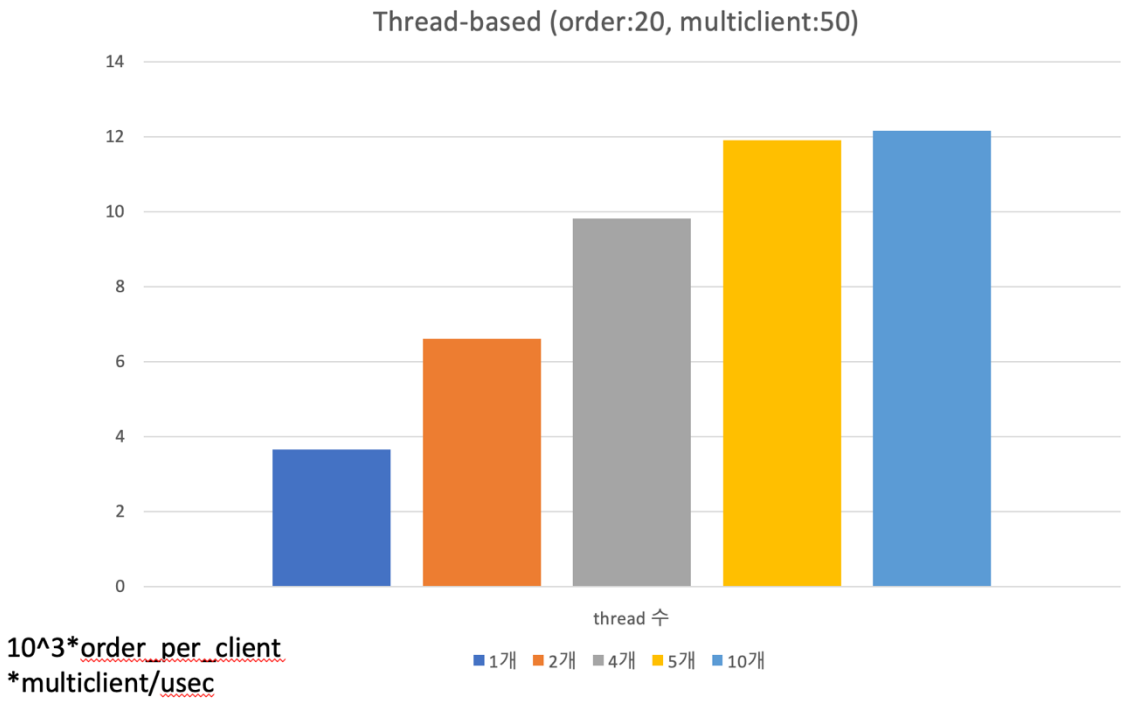
- 측정 시점: multiclient.c 에서 fork 실행 전 while위에서 gettimeofday(&start,0), waitpid 종료 후 gettimeoday(&end,0)까지

- 실행 환경: visual studio code ssh 환경

- 단위: 실행시간 측정 us

- 동시 처리율: 1000*order_per_client*multiclient/실행 시간(us)

```
Elapsed time[Thread based]: 272684 us (order_per_client: 20, multiclient: 50)
[buy] success
Elapsed time[Thread based]: 151191 us (order_per_client: 20, multiclient: 50)
 Elapsed time[Thread based]: 101707 us (order_per_client: 20, multiclient: 50)
Elapsed time[Thread based]: 83948 us (order_per_client: 20, multiclient: 50)
Elapsed time[Thread based]: 82210 us (order_per_client: 20, multiclient: 50)
```



10^3*order_per_client*multiclient/usec

분석

해당 분석은 프로젝트 Task3과 별개로 실제 thread의 개수에 따라 동시처리율이 어떻게 달라지는지 확인하고 싶어서 추가하게 되었다. 왜냐하면 thread_based 특성 상 thread의 개수에 따라 동시에 명령어가 수행되는 것에 대해 개수가 달라지기 때문에 실제 성능 차이가 매우 클 것으로 예측했기 때문이다.

Thread의 수가 1개라는 것은 결국 multithread 를 구현하지 못하고 있다는 뜻이다. 그래서 thread 1개와 thread 2개 일 때를 각각 보면 thread가 2배가 되었으니 처리율도 2배가 되었음을 도표를 통해 알 수 있다. 이는 thread 개수에 대해 동시처리율이 비례함을 알 수 있는데 결국 동시처리율 역시 어느 정도 thread 개수를 초과하면 한 값에 수렴하거나 증가율이 매우 줄어듬 역시 확인 가능하다. 이는 thread가 많아질 수록 reader_writer문제가 심각해진다. 한 thread에서 접근해서 노드를 쓰기 시작하면 다른 모든 thread 에서는 해당 노드에 접근하지 못하고 기다려야 한다. 이런 overhead가 커지기 때문에 thread 개수가 많아진다고 무조건적으로 동시 처리율이 좋아지는 건 아니라고 볼 수 있다.

분석의 한계점

필자가 예상한 것처럼 결과가 나오긴 했으나 사실 데이터 값들에 대한 정확도 및 변수 사항들이 존재한다.

첫번째로 ssh 연결 환경 및 cspro의 성능이다. Visual studio code 의 원격환경에서 돌렸는데 같은 방식으로 돌렸음에도 개발 서버의 동시 접속자 수가 많은지 적은지에 따라 혹은 process 들이 많은지 적은지에 따라 실행 속도 시간이 약간씩 달라지는 것 같았다. 결국 특정 시간대에 여러 번 시행해서 이것의 평균으로 계산했다.

두번째로 multiclient 코드를 보면 명령어들 자체에 대해 수행속도 시간이 다르다는 것을 알 수 있다. 왜냐하면 해당 실행 시간은 명령어를 서버에 보내고 처리한 다음 다시 결과 값을 받아오는 것을 측정하는데 show는 모든 주식 개수를 print 하고 sell, buy 같은 경우 성공 여부 만 출력하기 때문에 우리가 실제 분석하고자 하는 워크로드의 성능 분석에서 show에 대한 추가적인 overhead가 있다는 것을 알 수 있다. 더 정확히 reader_writer 문제를 분석하고자 했으면 출력을 제외하고 측정 하는 것이 더 좋을 것 같았다.

마지막으로 rand 함수를 사용하기 때문에 어떤 명령어가 몇 번 나올지를 정확하게 모른다는 것이다. 실제 rand 함수를 돌렸어도 show, sell, buy 명령어가 계속해서 다른 개수로 나오기 때문에 정확한 분석이 되지는 않았다. 모든 명령어가 나오게 하려면 rand()%3을 했는데 실제 우리가 분석한 바와 같이 워크로드에서도 동시처리율이 차이가 나기 때문에 order_per_client와 client 개수에 따른 첫번째 분석에서 정확한 결과를 얻었다고 결론을 내리긴 힘들 수 있다고 보았다.

하지만 전체적인 흐름 및 도표를 보면 우리가 공부한 event_driven, thread_based 의 성질들을 가지고 있는 분석임을 확인해 볼 수 있었다.

결론적으로 만약 client들이 몇개이고, 명령어가 read를 하는지, write도 같이 하는지, 또 속도를 빨리 해야 하는지 속도를 일정하게 유지하는 게 더 중요한지, 에 대해 알 수 있다면 이를 적절히 분석하여 thread_base approach, event_driven approac를 올바르게 사용해야 함을 알 수 있다.