Multicore Programming Project 2

담당 교수 : 박성용

이름 : 성진규

학번 : 20191598

1. 개발 목표

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

이번 프로젝트2에서는 Concurrent stock server를 구축하는 것을 그 목표로 하고 있다. 통상적으로 주식 서버 하나에는 한 명 이상의 주식 클라이언트들이 접속한다. 따라서 주식 서버를 Concurrent하게 구축하지 않으면 주식 서버가 여러 클라이언트와 connection이 불가능하므로, 서로 다른 클라이언트들의 요청 사항을 동시에 처리하지 못하는 문제가 발생한다.

Concurrent programming의 방법으로는 Process-based, Event-driven, Thread-based Approach가 있다. 프로젝트2에서는 Event-driven과 Thread-based 방법을 각각 Task1, Task2에서 구현해보고 두 구현 방법의 성능 차이를 Task3에서 확인해본다.

클라이언트가 주식 서버에 요구하는 명령어들은 아래 네 가지이고 이번 프로젝트에서는 이 명령어들을 구현하는 것이 핵심이다. 이를 위해서 이진 트리를 사용하여 주식 정보 데이터를 메모리에 적재하는 과정이 필요하다. 왜냐하면 구현할 명령어들 중에서 buy, sell, show는 해당 데이터에 반복적으로 접근하고 그 값을 갱신하기 때문이다.

- show : 현재 주식의 상태를 클라이언트에게 보여준다
- buy [주식 ID] [살 주식 개수] : 해당 ID의 주식을 주어진 개수만큼 산다. 이때 잔여 수량이 살 주식 개수보다 작다면 "Not enough left stocks"를 클라이언트에 출력한다.
- sell [주식 ID] [팔 주식 개수] : 해당 ID의 주식을 주어진 개수만큼 판다.
- exit : 주식 장을 퇴장한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

Multi-client의 connect와 request를 처리하기 위해 I/O Multiplexing을 활용한 networking programming으로 concurrent stock server를 구현한다.

- BST 자료구조를 이용한 stock data loading : stock에 대한 정보 (ID, left_stock, price) 는 해당 순서대로 "stock.txt" 파일에 저장되어있다. 이 값에 빠르게 접근하기 위해서 주식 ID 값을 기준으로 BST를 생성한다. 이 경우 배열을 사용하였을 때보다. 더 적은 메모리를 사용하면서도 log 시간복잡도에 원하는 ID 값을 가지는 주식의 정보를 찾을 수 있다.
- buy, sell : 사고 팔려는 주식 ID를 기반으로 해당 주식 정보를 BST에서 찾는다. 해당 명령이 성공한 경우 해당 주식 정보를 갱신한다. 성공 여부는 클라이언트에게 전달해준다.
- show : 모든 주식의 갱신된 최신 정보를 순서에 관계없이 클라이언트에게 전달해준다.
- exit : connfd를 닫아버리는 방법으로 구현한다. 따라서 exit를 클라이언트가 호출하면 클라이언트의 서버 접속이 끊어진다.

2. Task 2: Thread-based Approach

Task2에서는 Thread-based Approach를 이용해 Multi-clinet의 connect와 request를 처리한다. request 구현은 Task1과 같으며, Task2 구현 개요는 아래와 같다.

- Master thread 사전에 일정 개수의 Worker thread pool을 생성한다.
- Master thread는 multi-client들의 connect을 accept하며 이를 Worker thread에 전달한다. 이 때 Worker thread 마다 하나의 client의 요청을 처리하도록 구현하여 concurrent 하게 서버가 동작하도록한다.
- thread에서 shared data에 접근하는 경우의 race을 방지하기 위해서 semaphore을 사용하여 이를 처리해준다.
- 해당 client가 연결 해제되면, Worker thread를 비어있는 상태로 업데이트하고 다음 connect이 들어올 때 비어있는 Worker thread를 사용하며 반복한다.

3. Task 3: Performance Evaluation

Task 3에서는 Event-driven과 Thread-based Approach로 구현한 Concurrent Stock Server의 성능을 평가하려고 한다. 성능 평가의 목적성에 부합하려면 평가 대상의 주요한 특징에 입각하여 평가 항목과 그 방법을 설정해야한다.

평가 대상이 Concurrent Stock Server이므로, 우리는 Server에 얼마나 많은 Client가 동시에 접속하였을 때 성능이 어떻게 감소하는지 측정하는 것이 타당하다. 이를 위해 동시처리율이라는 개념을 설정하고 이 상황에서 아래의 변수들을 조정했을 때의 동시처리율의 추이를 살펴보고자 한다.

- Workload의 비율 : Buy, Sell, Show request의 비율에 따라서 다르게 조사한다. Buy, Sell은 Read / Write 동작을 순서대로 실행해야하는 request들이다. 왜냐하면 BST에서 Search한 뒤 해당 노드의 값을 업데이트하기 때문이다. Task2 구현에서 Reader / Writer 방식으로 최적화하였기에 이에 대해 조사한다.
- STOCK_NUM의 값 : STOCK_NUM은 stock.txt에서 인식하려는 주식 정보의 개수이다. 우리는 주식 정보를 BST 자료구조에 Load하여 사용할 것이므로 STOCK_NUM의 값에 따라서 BST 자료구조의 크기가 달라지게 된다. 이에 따라서 자료 접근 시간이 달라지므로 이를 조사한다.
- Worker Thread 생성 개수 : Task2에서 구현한 Pre-thread-based approach의 경우 client connection을 받기 이전에 정해진 개수의 Worker Thread를 생성하여 두고, 이를 이용해 Client request를 처리한다. 따라서, 생성 개수에 따라 Concurrent한 실행 양상이 달라지게 되므로 이에 대해 조사한다.

위 측정을 위해서 multiclient.c 코드를 임시로 적절히 수정하여서 평가한다. 제출 시에는 원래 코드로 복구하여 진행한다.

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

네트워크 프로그래밍에서 EOF는 곧 Close(connfd)가 발생한 것과 동치이다. 따라서 Close 호출이 이루어지지 않으면 서버는 클라이언트의 입력 라인을 클라이언트의 접속이 끊어질 때까지 계속 echo (Rio_writen) 할 것이다. 따라서, 만약 클라이언트의 STDIN이 Socket을 통해 서버로 전달되더라도 이에 대한 응답을 클라이언트가 서버로 부터 받을 수 없게 된다. 이를 해결하여 각 입력 라인마다 클라이언트로 응답해주는 방법이 바로 I/O Multiplexing 이다. I/O Multiplexing의 작동 과정을 개략적으로 아래와 같다.

a. Servers : listenfd 하나를 열고 초기화한다.

b. Servers : listenfd 가 event가 발생할 때까지 기다린다.

c. Client(s) : event를 발생시킨다.

d. Servers : event가 발생하였으므로 이를 발생시킨 client와 연결한다.

e. Client(s) : request를 보낸다.

f. Servers : request를 check하여 이를 탐지하면 그에 맞는 처리를 한 뒤 다시 Client에게 return한다.

✓ epoll과의 차이점 서술

아래 내용은 Linux manpage를 참고하여서 작성하였다.

epoll의 동작의 핵심은 바로 커널 내부의 데이터 구조인 epoll 인스턴스이다. 이는 사용자의 관점에서 다음과 같은 두 개의 목록에 대한 컨테이너로 생각할 수 있다.

- The interest list (a.k.a epoll set) : 해당 프로세스가 '관심있다(interest)'라고 등록한 file descriptors의 집합이다.
- The ready list : I/O를 위해 ready 상태인 file descriptors의 집합이다. ready list의 원소는 항상 프로세스가 interest한 fd이므로, ready list는 항상 interest list의 부분 집합이다. I/O ready 상태에 따라서 커널에 의해 동적으로 집합이 구성된다는 것에 유의한다.

select도 epoll과 동작 자체는 비슷하다.

- `fd_set read_set` : read_set은 fd_set 자료형(비트맵)으로 acitve 상태인 fd들을 관리한다.
- `fd_set ready_set` : ready_set은 I/O를 위해 ready 상태인 fd들을 관리한다.

select를 이용해서 I/O multiplexing을 구현할 때에는 ready_set을 설정하기 위해서 select()의 호출이 반드시 필요하다. select()를 호출하면 항상 read_set을 copy한 다음, 이를 인자로 호출하면 해당 인자가 ready_set으로 바뀌는 동작을 한다. 이 과정은 user(server)-level에서 이루어지고 비트맵 복사가 이루어진다는 점에서 관리와 성능 측면에서 좋지 않다. 또한, ready 상태인 fd들을 순차적으로 연결하고 이를 배열을 선언하여 관리해주어야 한다.

반면 epoll은 ready list의 설정이 커널에 의해 이루어지기 때문에 관리 측면에서도 용이하고 메모리 복사가 이루어지지 않으므로 비교적 성능도 우수하다. 또한 배열을 별도로 선언해줄 필요가 없다.

다만, epoll은 Linux I/O 함수이다. 즉 다시 말해 OS에 종속되어있다는 단점이 있다. 예를 들어 Windows 환경에도 호환되어야하는 경우는 epoll 함수를 사용할 수 없다. 이 경우는 어쩔 수 없이 select 함수를 사용해야 한다.

Task2 (Thread-based Approach with pthread)

✓ **Master Thread의 Connection 관리**

Stock server가 최초로 실행되었을 때, Master Thread는 정해진 개수만큼의 Worker Thread를 생성한다. 이에 대해서는 후술한다.

Client의 Connection들은 buffer를 이용해 관리한다. 버퍼는 순환 연결리스트 구조로 생각할 수 있으며, 이를 배열로 구현되어있다. 어떤 Connection이 새롭게 들어왔을 때, Master Thread는 buffer에 빈 공간이 있는지를 먼저 따져본다. buffer가 꽉차지 않았다면, 비어있는 버퍼에 connfd를 삽입한다. buffer가 꽉 차 있다면, Semaphore를 사용하여 connfd가 Pending 되도록 한다.

이게 버퍼에 저장되어있는 connfd를 삭제하면서 이를 Worker Thread에서 가져가게 한다. 그러면 해당 Worker Thread는 connfd를 받게되고 이를 이용해 Client의 request를 전달받아 이에 response해주는 구조로 Networking 하게 된다.

✓ **Worker Thread Pool 관리하는 부분에 대해 서술**

앞서 언급했듯이 Stock server의 최초 실행에서, Master Thread는 정해진 개수만큼의 Worker Thread를 생성하는데, 이렇게 생성된 Thread들을 **Worker Thread Pool** 이라고 한다.

위에서 설명한 Master Thread의 Connection 관리 부분에서 언급한 Connection 관리를 배경으로 생각해보자.

Worker Thread는 connfd가 buffer에서 삭제되면서 전달될 때, 해당 connfd에 대해 request를 처리한다. 모든 request를 처리하고 client가 접속 종료를 한다고 생각해보자. 그러면 단순히 connfd를 Close()하면 그만이다. 왜냐하면 buffer에는 이미 Thread가 connfd를 받아오면서 지워졌기 때문이다.

이제 Thread는 while (1)을 돌면서 계속해서 buffer에 남아있는 다른 connfd를 받아오고 이에 대한 처리를 해준다. 만일 buffer에 아무것도 없다면, 받아올 것이 없으니 계속 while(1)인 상태일 것이다.

즉, Worker Thread Pool 관리 부분에서, 이전 단원에서 배운 Thread echo server와 다르게, 한 번 생성된 Worker Thread Pool의 Thread들은 절대로 종료되지 않는다는 점을 유의해야 한다. 모든 Thread들이 계속 대기 중이면서, connfd가 buffer에 들어올 때마다 이를 처리해주는 방식으로 동작한다.

- **Task3 (Performance Evaluation)**

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

Step1. Defines `Concurrent Process ratio` of Servers

$$\text{Concurrent Process ratio} \overset{\text{def}}{=} \frac{\text{(Total Processed) Request}}{\text{(Total Processed) Time}},$$

rewrite,

$$\mathbf{CPR} \overset{\text{def}}{=} \frac{\mathbf{R}}{\mathbf{T}} \qquad \because \text{PPT Task3 요구사항에서 동시처리율}$$

such that

$$\mathbf{R} = \sum_{i=1}^{N} (\text{request of client}_i), \quad \mathbf{N} = \text{total clients number} \ \cdots (*)$$

Moveover,

$$\mathbf{OPC} = \text{ORDER PER CLIENT} \overset{\text{def}}{=} \text{Request per client}$$

Since, The term `Request per client` literally means that E[request] for each client.

It follows from $(*)$ that,

$$\text{E[request]} = \sum_{i=1}^{N} \frac{(\text{request of client}_i)}{\text{total clients number}} = \frac{1}{\mathbf{N}} \sum_{i=1}^{N} (\text{request of client}_i) = \frac{\mathbf{R}}{\mathbf{N}} \cdots (1)$$

Step2. Defines some factors which affect to T,

We exactly conclude that

$$\mathbf{T} = \sum (\text{request execute time})$$

Remarks an request execute time depends on spatial complexity of BST data structure, denoted by $O(\alpha)$. $(the\ number\ of\ nodes\ of\ BST \triangleq \alpha)$

And on $request\ types\ ratio_i (= p_i)$ of server.
Spacially, it also depends on $the\ number\ of\ \ threads\ (\triangleq \beta)$ in task2.

Let `the execute time of $request_i$` be a function ψ_i, we denote,

$$\mathbf{T} = \psi(\alpha, \beta) = \sum p_i \psi_i(\alpha, \beta) \ \ (\text{where} \sum p_i = 1) \cdots (2)$$

Then it follows from $(1), (2)$ that,

$$\mathbf{CPR} \overset{\text{def}}{=} \frac{\mathbf{R}}{\mathbf{T}} = \frac{\mathbf{OPC} \times \mathbf{N}}{\psi(\alpha, \beta)}$$

우리는 아래 세 가지 경우에 대해서 CPR을 측정하려고 한다.

I. p_i : request의 비율을 buy, sell만 존재하는 경우와 show만 존재하는 경우 그리고 전부 포함하는 경우로 나누고 N을 키워가면서 측정

II. α를 늘려가면서 측정

III. β를 늘려가면서 측정 (only task2)

각 경우에서 필요한 변수를 제외한 나머지는 적절한 상수로 둔다.

✓ **Configuration 변화에 따른 예상 결과 서술**

I. Task1이 Task2 보다 N이 커질 수록 성능이 안좋을 것이다. 왜냐하면 Task1의 경우는 request 처리가 concurrent하지는 않기 때문이다.

또한, Task2의 성능은 buy, sell은 read / write을 전부해야하지만, show는 read만 하기 때문에 show만 존재하는 경우에 그렇지 않은 경우보다 더 좋을 것이다.

II. α의 크기, 즉, stock.txt의 데이터가 많을 수록 느릴 것이다. 다만, BST의 $\text{depth} = \log_2 \alpha$ 이기 때문에 급격한 성능 저하가 있지는 않을 것이다.

III. β의 크기, 즉, Worker thread pool의 크기가 클 수록 빠를 것이다. 더 많은 Thread가 concurrent하게 동작할 것으로 예상되기 때문이다.

C. 개발 방법

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

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 구현

  a. Servers : listenfd 하나를 열고 초기화한다.

  `listendfd = Open_listenfd(...)`와 `init_pool(listenfd, &pool)`을 순서대로 호출하여 초기화해준다. 여기서 `pool` 구조체를 선언할 필요가 있다. B에서 설명했듯이 `select` 함수는 user-level에서 배열을 선언하여 server와 client가 통신하기 위해 사용할 fd와 buffer를 관리해주어야 한다. 이를 위해 필요한 자료형들을 구조체로 묶어 둔 것이 바로 `pool` 구조체이다. 구조체 멤버들은 아래와 같다.

  - `int maxfd : read_set`의 `fd` 중에서 최대값이다. 즉, 마지막 `fd` 값이면서 동시에 우리가 탐색할 `read_set`의 개수를 나타낸다고 볼 수 있다.
  - `fd_set read_set : 모든 active fd`의 집합 (비트맵)이다.
  - `fd_set ready_set : 모든 I/O ready fd`의 집합 (비트맵)이다.
  - `int nready : select` 호출을 기다리고 있는 (ready 상태) `fd`의 개수이다. `select` 함수의 리턴값이 이 값이므로 항상 `nready = select(...)` 로 호출한다.
  - `int maxi : int clientfd[FD_SETSIZE]` 의 최대 인덱스이다. high water mark라고도 하는데, 즉, `pool`에서 '윗물'이므로 `maxi`보다 작거나 같은 인덱스에만 `clientfd`가 채워졌을 수 있다.
  - `rio_t clientrio[FD_SETSIZE] : active read buffer`를 저장하기 위한 배열이다.

  b. Servers : listenfd 가 event가 발생할 때까지 기다린다.

  `pool.nready = select(pool.maxfd+1, &pool.ready_set, ... )` 으로 호출한다. 즉, event가 발생한 `fd`의 인덱스만 `1`로 바꾼 `ready_set` 비트맵을 만들어준다.

c. Client(s) : event를 발생시킨다.

Client가 Stock server에 connect을 시도하면 된다.

d. Servers : event가 발생하였으므로 이를 발생시킨 client와 연결한다.

`connfd = Accept(listenfd, ...)` 를 호출하고 이를 `add_clients(&pool)` 하여서 `pool` 구조체의 `clientfd[0~maxi]`에 추가한다.

e. Client(s) : request를 보낸다.

buy, sell, show, exit 네 가지 event가 가능하다. 각 구현은 간단하다. 전체 주식데이터를 이진탐색트리로 가져오고 buy, sell의 경우는 주어진 인자에 대해 트리 노드에 접근하여 해당 노드의 정보를 기반으로 적절히 메시지를 생성하고 노드의 정보를 갱신한다. show의 경우는 이진탐색트리를 순회하면서 출력용 메세지를 만들어준다. exit가 호출되면 클라이언트의 연결을 종료 (`clientfd`의 값을 `-1`로 설정) 한 뒤, `fd`를 `close`해준다. 노드 구조체는 아래와 같다.

I. `int id` : 주식의 `ID`

II. `int left_stock` : 해당 주식의 잔여수량 (유일하게 변하는 값)

III. `int price` : 주식 가격

IV. `Node_t* left, right` : 노드의 왼쪽 자식, 오른쪽 자식이다.

f. Servers : request를 check하여 이를 탐지하면 그에 맞는 처리를 한 뒤 다시 Client에게 response한다.

위에서 설명한 구현을 서버에서 실행한 뒤 생성된 메세지들을 Rio_writen을 사용하여 Client에게 response해준다.

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

  ✓ **Master Thread의 Connection 관리**

  Connection을 관리해주는 buffer를 선언해야한다. 단순히 저장 공간을 할당하는 것이 아니라, 이를 배열을 이용한 순환 연결리스트처럼 동작하도록 하고 싶으므로, 관련된 변수들을 포함한 구조체를 sbuf_t 를 선언한다.

```
typedef struct {
        int *buf;  // 배열을 할당한 포인터
        int n;     // 배열 크기
        int front;  // 순환 연결리스트 첫번째 노드
        int read;  // 순환 연결리스트 마지막 노드
        sem_t mutex; // buf 접근 관리 용도의 세마포어
        sem_t slots;  // slots 개수를 세기 위한 세마포어
        sem_t items;  // items 개수를 세기 위한 세마포어
} sbuf_t

sbuf_t sbuf;  // global var.
```

  Master Thread에서 Worker Thread Pool을 생성하기 전에 sbuf을 초기화해주기 위해서 init_sbuf()를 호출한다. 여기서는 sbuf 구조체 멤버 변수를 초기값으로 설정할 뿐만 아니라 int* buf에 대한 동적할당도 이루어지도록 구현한다.

  이후 Accept를 수행하며 connfd를 받아올 때마다 void insert_sbuf()를 호출하여 connfd를 sbuf에 삽입한다. 이 때, 세마포어를 이용해 빈 슬롯이 있는 경우에만 삽입이 이루어지고, 그렇지 않은 경우는 pending 되어 있는 상태가 되게 구현해야 한다.

  connfd를 Thread Pool에 전달해줄 때 사용하는 int sbuf_remove() 함수의 경우, sbuf에 items이 존재하는 경우에만 connfd를 받아올 수 있다. 따라서 세마포어를 이용해서 items가 1이상이 경우에만 실행되도록 구현한다. 이 함수의 경우 반환값으로 sbuf에서 제거한 connfd를 가지도록 구현하여 Thread Pool에 넘겨줄 수 있도록 구현하였다.

  위 두 함수 모두, mutex에 대한 세마포어 기법을 활용하여 sbuf가 lock되지 않은 상태에서만 동작이 이루어져야한다. 왜냐하면 동시에 sbuf에 접근하다보면 Race가 발생할 수 있기 때문이다.

✓ **Worker Thread Pool 관리하는 부분에 대해 서술**

Master Thread에서 init_sbuf() 호출 이후에 곧바로 Worker Thread Pool을 생성한다. 이 때, #define NTHREADS 를 이용하여 해당 개수만큼 생성한다.

```
for (int i=0; i<NTHREADS; i++)
        Pthread_create(&tid, NULL, thread, NULL);
```

이렇게 생성된 Worker Thread들은 절대로 종료되지 않고 계속해서 Pool에서 다음 connfd에 대한 request를 처리해주기 위해 대기하고 있는 상태여야 한다. 그러므로, 어떤 connfd에 대한 request를 모두 처리하고 해당 Client가 접속 종료했다고 해서 Thread를 종료하면 안되고, 단순히 connfd를 해제해주기만 해야 한다. 결과적으로 void* thread(void *vargp)의 구현은 아래와 같이 한다.

```
void* thread(void *vargp) {
        Pthread_detach(pthread_self());
        while (1) {  // Thread 종료하지 않아야 하므로
                int connfd = sbuf_remove(&sbuf);
                echo(connfd);  // request에 대한 처리를 하는 함수
                Close(connfd);
        }
}
```

- **Task3 (Performance Evaluation)**

  multiclient.c 에서 코드를 약간 수정하여서 실행 시간 $\mathbf{T} = \psi(\alpha, \beta)$를 측정하도록 한다. 이 때, CPU clock time이 아니고 실제 실행 시간을 전부 측정해야하므로, clock() 함수를 사용하지 않고 gettimeofday() 함수를 호출한다.

  multiclient의 실행이 최초 시작되는 시점부터 완전히 종료되는 지점까지의 시간을 측정하도록 한다.

  측정할 각 경우에 대해서 변수를 적절히 조정해가면서 시간을 측정하고, 이렇게 측정한 시간은 perform.txt 파일에 기록하도록 구현한다.

  C로 데이터 관측을 하기에는 상당히 불편하기 때문에, Linux에서는 데이터 생성만 하도록하고, MATLAB을 이용해서 데이터를 관측하고 그래프를 살펴본다.

3. 구현 결과

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

  I/O Multiplexing이 Event-driven 방식을 통해 적절히 구현되었다. 프로젝트 예시 파일과 함께 주어진 Clients 프로그램 두 가지를 사용하여 테스트해본 결과 모든 기능을 만족하면서 적절히 동작하는 것을 알 수 있었다. stockclient.c 는 오직 하나의 클라이언트가 우리가 구현한 서버에 접속하는 소스코드이다. 이 클라이언트를 실행하면서 클라이언트 request 네 가지를 적절히 처리해주는 서버의 함수들을 구현하였다. 여러 클라이언트가 적절히 동작하는지는 multiclient.c 를 실행해서 테스트해보았다. 구현 결과는 입력에 대해 적절히 동작하였고, 클라이언트가 접속 종료될 때마다 stock.txt 파일을 갱신하도록 구현하여서 persistency를 높였다.

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

  NTHREADS 만큼의 Thread Pool이 생성되었고, 여기에 Client의 Stock server connection 마다 connfd를 넘겨주면서 동작하도록 구현하였다. 이에 따라서, Stock server의 request 처리는 이번 프로젝트에서 원하는 바와 같이 concurrent 하게 동작하였다.

  이 상황에서 각 Worker Thread에서 호출되는 request를 처리해주는 echo 함수를 생각해보자. echo 내부에서 주식정보 BST에 대해 읽고 쓰는 처리 과정이 존재하고 주식정보 BST는 Thread 모두가 공유할 수 있는 데이터이다. 따라서, Race condition에 따라서 Readers / Writers Problems 문제가 발생할 수 있으므로 이를 해결해주어야만 한다.

  echo() 내부에서 read / write 를 각 한번씩 수행하는 buy, sell의 경우는 writer로 간주하며 세마포어를 이용했으며, read만 수행하는 show의 경우는 reader이므로 이에 맞게 세마포어를 이용해 구현하였다.

  이를 통해 echo() 내부에서 reader / writer problem이 해결되었다.

  모든 client 실행 파일을 이용해 테스트한 결과 잘 동작하는 것을 확인 할 수 있었으며, 각 Worker Thread에 전달된 connfd가 Close()될 때, 다시 말하면 Client의 접속이 종료될 때 stock.txt 파일이 갱신되도록 구현하여서 persistency를 높였다.

- **Task3 (Performance Evaluation)**

각 측정 경우에 대해서 설정한 상수값을 표로 정리한 것이다. 이를 바탕으로 실험을 진행하였다. OCR 값을 늘리는 것은 N을 늘리는 것과 동일한 효과를 낼 것이므로 이 경우는 생각하지 않았다. 왜냐하면 Thread가 서로 물리적으로 다른 코어에서 돌도록 구현한 멀티코어 프로그래밍을 구현한 것이 아니기 때문에 하나의 코어에서 모든 Thread의 request들이 concurrent하게 처리될 것이기 때문이다. BUT_SELL_MAX 값도 성능 평가에 거의 영향을 주지 않을 것이기 때문에 생각하지 않았다. BST 노드의 값에 +1을 더하고 빼는 것과 +1000을 더하고 빼는 것을 측정하는 것은 성능 측정 실험에서 의미가 없다고 생각했다.

case / value	p_i	α	β
N	**var**	20	20
α	10	**var**	1
β	20	20	**var**
p_i	(buy & sell) or Show	all	all

4. 성능 평가 결과 (Task 3)

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

성능 평가에서 측정 시점은 항상 multiclient 시작과 끝이다. 이에 대한 캡쳐는 한 번만 제시한다. 시간의 단위는 sec로 측정하였다.

```c
#include "csapp.h"
#include <time.h>

#define MAX_CLIENT 20          // N
#define ORDER_PER_CLIENT 10    // OPC - Constants
#define STOCK_NUM 1            // Alpha
#define BUY_SELL_MAX 1         // Constants

/* Task3 - Performance test */
struct timeval tv;
double begin, end;
FILE* perform;
/* ------------------------ */

int main(int argc, char **argv)
{
```

```c
    /* Task3 - Performance test */
    gettimeofday(&tv, NULL);
    end = (tv.tv_sec) * 1000 + (tv.tv_usec) / 1000 ;
    fprintf(perform, "%lf\n", (end - begin) / 1000);
    /* ------------------------ */

    return 0;
}
```

모든 측정시간 데이터는 그 안정성을 보장하기 위해 5회의 시행 후의 평균을 낸 것이다.

또한, 충분히 많은 횟수의 request를 rand()으로 생성하였으므로 큰 수의 법칙에 의해 어떤 특정 request가 나타날 횟수는 해당 횟수의 모평균에 수렴할 것이다.

따라서, 모든 request 비율 p_i는 거의 같다고 볼 수 있다.

예를 들어, Buy & Sell만 request하도록 구현하였을 때, 비록 Buy인지 Sell인지를 rand()으로 생성하지만, 큰 수의 법칙에 의해 충분히 큰 시행을 했기 때문에 결과적으로 Buy와 Sell이 1:1 비율로 나타날 것이라는 의미이다.

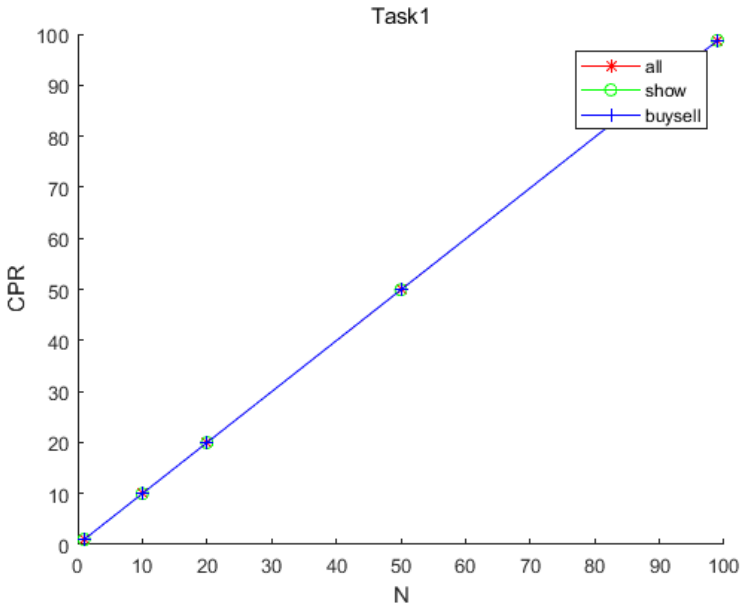
I. p_i의 값을 변경하면서 테스트한 경우

아래는 task1에서 어떤 Workload (request ratio)가 주어졌을 때, N에 따른 측정 시간 결과이다.

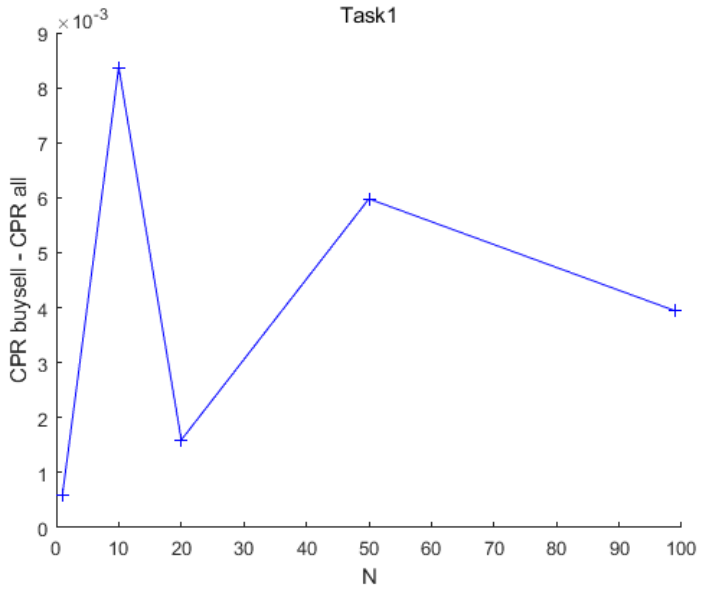
20191598 > task_1 > perform.txt

```
[All, var = All requests by N]
25.028000, 1
25.040000, 10
25.035000, 20
25.052000, 50
25.057000, 99
[All, var = show requests by N]
25.046000, 1
25.057000, 10
25.047000, 20
25.046000, 50
25.061000, 99
[All, var = buy sell requests by N]
25.031000, 1
25.036000, 10
25.045000, 20
25.043000, 50
25.060000, 99
```

실험에서 $\mathbf{OPC} = \mathbf{25}$ 으로 일정하므로 이에 따라서 구한 CPR은 아래 그래프와 같다.



Workload에 따른 차이가 거의 나타나지 않는 것을 알 수 있다. 오차를 알아보기 위해서 offset 그래프를 살펴보자.



buy & sell만 하는 경우가 show 만하는 경우보다 CPR 값이 약간 높은 것을 확인 할 수 있지만 그 차이가 크지는 않다.

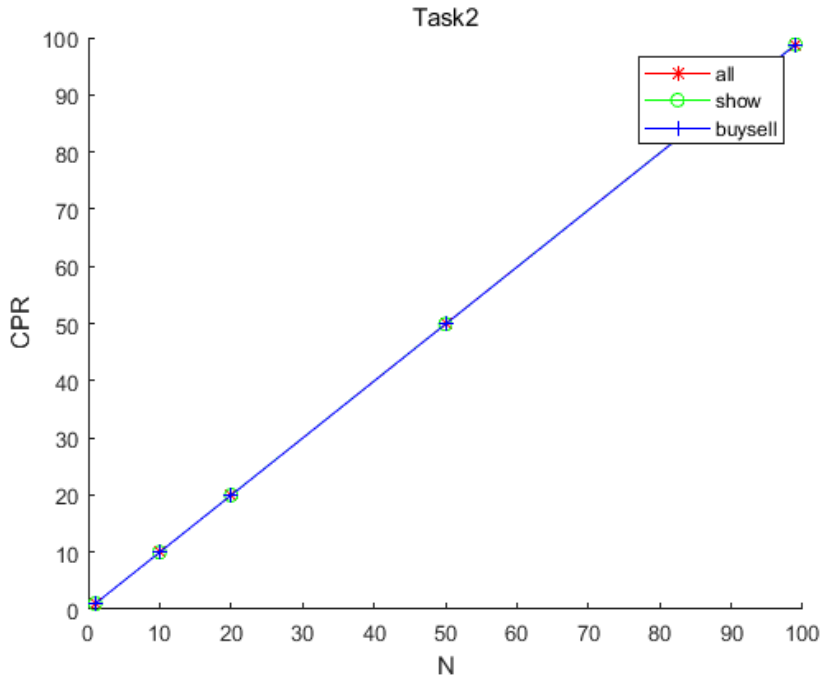
아래는 task2에서 어떤 Workload (request ratio)가 주어졌을 때, N에 따른 측정 시간 결과이다.

20191598 > task_2 > perform.txt

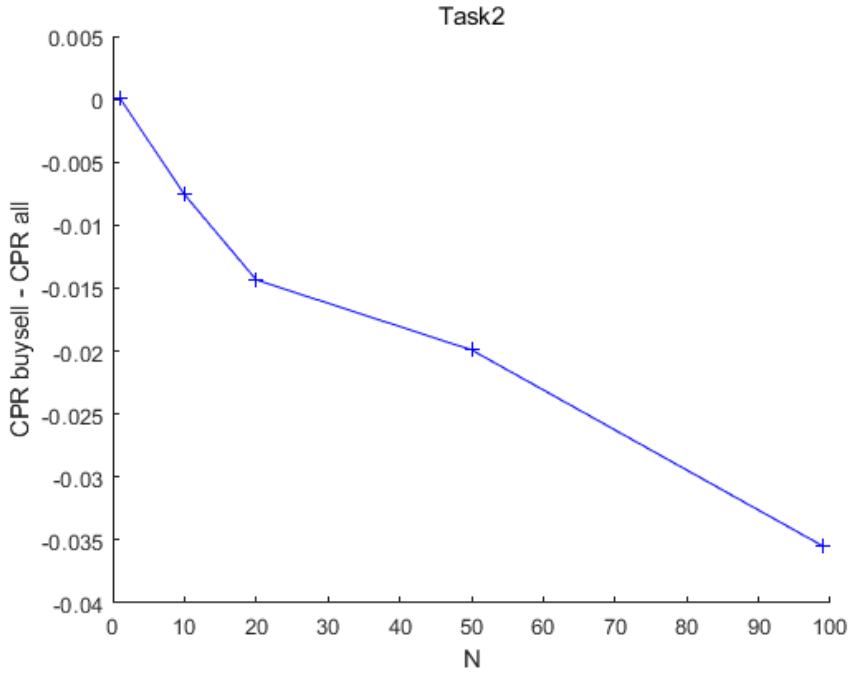
```
[All, var = All requests by N]
25.029000, 1
25.035000, 10
25.036000, 20
25.047000, 50
25.045000, 99

[All, var = show requests by N]
25.033000, 1
25.034000, 10
25.037000, 20
25.053000, 50
25.047000, 99

[All, var = buy sell requests by N]
25.031000, 1
25.053000, 10
25.055000, 20
25.063000, 50
25.056000, 99
```



Workload에 따른 차이가 거의 나타나지 않는 것을 알 수 있다. 오차를 알아보기 위해서 offset 그래프를 살펴보자.



buy & sell만 하는 경우가 show 만하는 경우보다 CPR 값이 약간 낮은 것을 확인 할 수 있지만 그 차이가 크지는 않다.

예상했던 결과 : Task1이 Task2 보다 N이 커질 수록 성능이 안좋을 것이다. 또한 buy, sell은 read / write을 전부해야하지만, show는 read만 하기 때문에 show만 존재하는 경우에는 Task2의 성능이 더 좋을 것이다.

실제 결과 : Task1과 Task2의 성능 차이가 거의 없는 것을 확인 할 수 있었다(시간 결과 값이 거의 유사하다). 아마 그 이유를 추측해보자면, 멀티코어를 사용하지 않기 때문에 Thread의 강점이 드러나지 않았고 반대로 Thread의 overhead가 더 두드러진 것 같다.

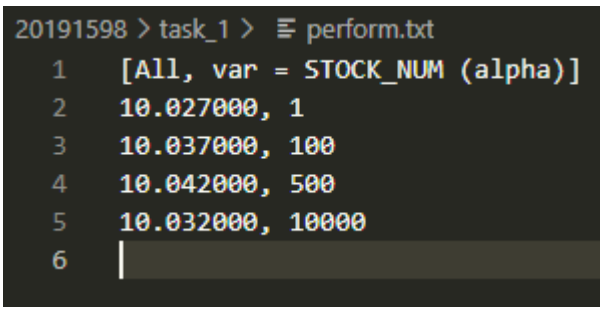
Task1의 경우는 buy / sell 만 있는 것이 show보다 성능이 약간 더 좋았다. 반대로 Task2의 경우는 show만 있는 것이 buy / sell 보다 성능이 약간 더 좋았다.

Task2에서 우리는 reader / writer problem을 해결하면서, 그 효과로 read request를 overlap이 큰 방식으로 처리하도록 구현하였다. 따라서, read만을 수행하는 show request만 처리하는 경우가 더 성능이 좋게 나왔을 것이다.

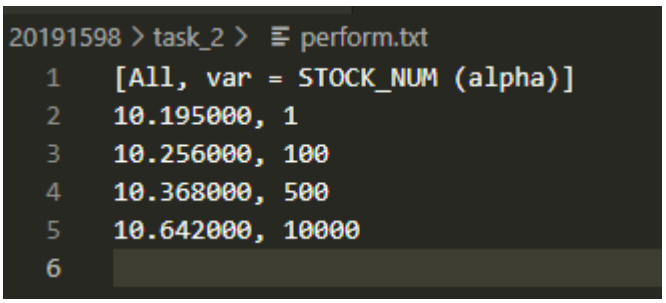
반면 Task1에서는 순서대로 request를 처리하므로, 평균적으로 더 많은 정보를 STDOUT에 출력하는 show보다는, 항상 한 줄만 출력하는 buy / sell 의 경우가 더 성능이 잘 나왔다고 추측할 수 있다.

II. α의 값을 늘려가면서 테스트한 경우

아래는 task1에서 α에 따른 측정 시간 결과이다.

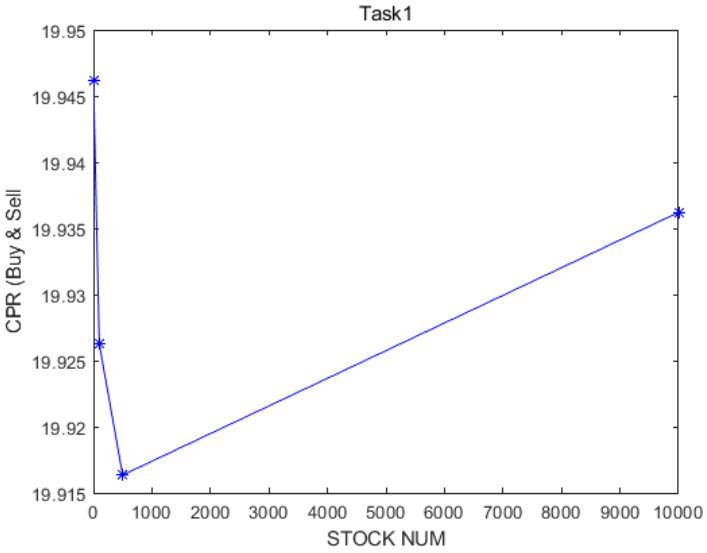


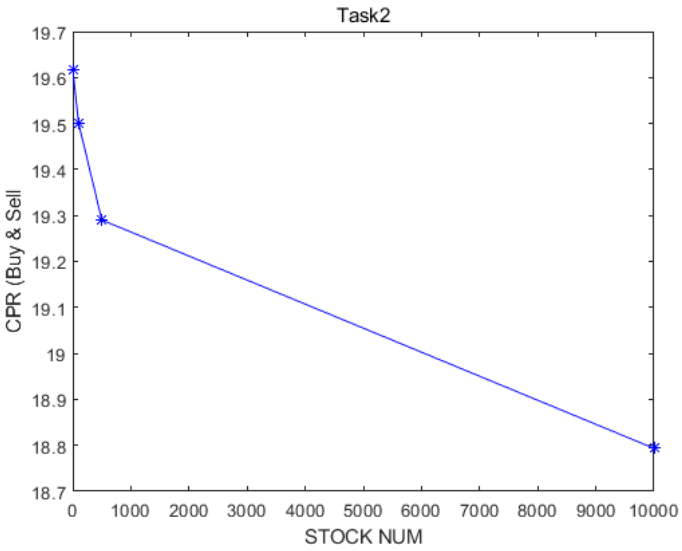
아래는 task2에서 α에 따른 측정 시간 결과이다.



두 경우 모두 실험에서 $\mathbf{R = OPC * N = 10 * 20}$ 으로 일정하므로 이에 따라서 구한 CPR은 아래 그래프와 같다. 이 때 show request의 경우는 모든 BST 노드를 순회하는 $O(\alpha)$ 시간을 가지고 있다. 따라서 show의 비율이 변화함에 따라서 그 결과가 일관적이지 않게 나올 것이므로 이를 제외하고 $O(\log_2 \alpha)$ 시간에 이루어지는 Buy & Sell request만으로 평가하였다.

두 경우의 그래프가 각각 아래와 같다.





예상했던 결과 : $O(\log_2 \alpha)$ 시간이기 때문에 별 차이가 없을 것이다.

실제 결과 : 그래프만 보면 큰 차이가 나는 것처럼 보일 수 있으나, y축을 자세히 살펴보면 그 차이가 매우 미미한 것을 알 수 있다. 심지어 Task1 쪽을 보면 STOCK_NUM이 커졌음에도 불구하고 CPR이 더 높은 그래프를 보여준다.

왜냐하면 buy & sell 의 경우는 데이터를 BST에서 탐색할 때 $O(\log_2 \alpha)$ 시간이 걸리고 데이터를 수정할 때 $O(1)$ 시간이 걸리기 때문에 아무리 데이터가 커지더라도 성능 저하가 상대적으로 미미하기 때문이다.

또한, 데이터 접근이 아닌 다른 여러 요인들도 실행시간에 영향을 주기 때문에 상대적으로 영향이 적은 인자에 대한 그래프는 당연히 노이즈가 있을 수 밖에 없다.

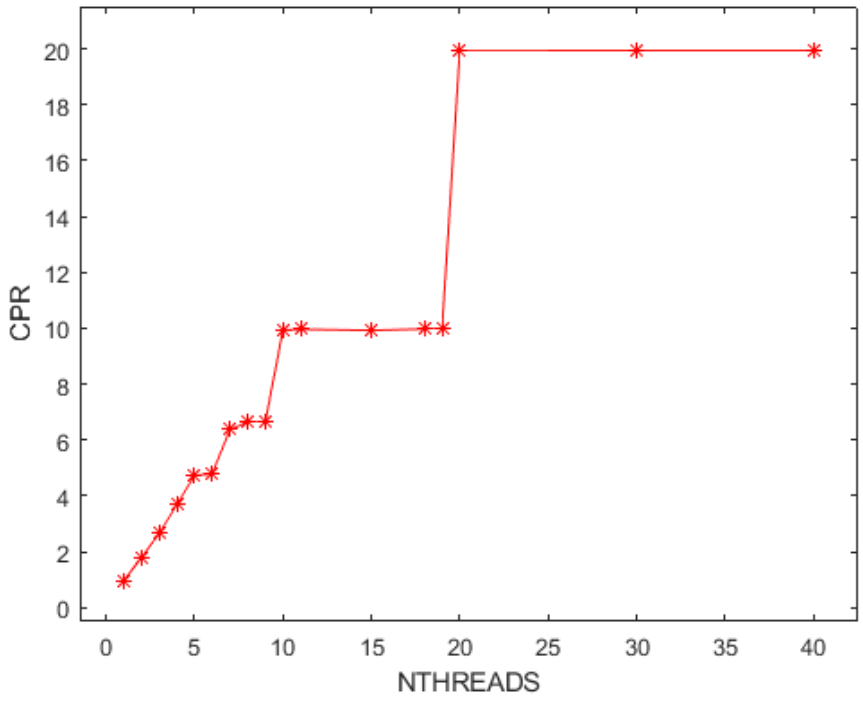
III. β의 값을 늘려가면서 테스트한 경우 (only Task2)

측정 시간 결과

20191598 > task_2 > perform.txt

```
[All, var = NTHREADS (beta)]
206.935000, 1
111.264000, 2
75.307000, 3
54.182000, 4
42.247000, 5
41.707000, 6
31.384000, 7
30.086000, 8
30.149000, 9
20.135000, 10
20.071000, 11
20.154000, 15
20.048000, 18
20.039000, 19
10.039000, 20
10.036000, 30
10.034000, 40
```

실험에서 $\mathbf{R = OPC * N = 10 * 20}$ 으로 일정하므로 이에 따라서 구한 CPR은 아래 그래프와 같다.



예상했던 결과 : β의 크기, 즉, Worker thread pool의 크기가 클 수록 빠를 것이다. 더 많은 Thread가 concurrent하게 동작할 것으로 예상되기 때문이다.

실제 결과 : 같은 R을 가질 때, Worker thread pool의 크기가 클 수록 빠른 경향이 있는 것은 맞다. **하지만 $\beta > N$인 경우부터는, 성능 개선이 이루어지지 않는다.**

왜냐하면 Thread-based stock server의 구현 방법을 생각했을 때, Worker thread 하나 당 Client의 connfd 하나가 들어오면서 동작하게 된다. 그러나 이 상황부터는 thread의 개수가 client의 수보다 많기 때문에 더 이상 추가적인 Thread 간의 overlap이 발생하지 않게 된다. 따라서 더 이상 성능 개선이 이루어지지 않으며, 심지어 thread 개수가 너무 많아지면 thread pool을 생성할 때의 overhead로 인해 오히려 서버의 성능이 저하될 것이다.

또한, 중간 중간 성능 개선이 이루어지지 않는 구간이 있는 것을 관찰할 수 있다. 이러한 현상은 특히 $\beta = 10\sim20$ 인 구간에 두드러진다. 이 현상에 대한 정확한 이유는 찾지 못하였다. 이는 추후 OS 같은 수업에서 thread에 대한 더 많은 공부를 한 뒤에 논의해보아야겠다.

-끝-