1. 개발 목표

이진 트리를 구축하여 만들어진 메모리를 이용해 각각 event based approach 와 thread based approach를 이용하여 명령어를 요청한 클라이언트에게 응답을 주는 주식 서버를 구축하는 것이 목표이다.

2. 개발 범위 및 내용

A. 개발 범위

1. select

   이미 준비된 각각의 descriptor로부터, 클라이언트의 요청이 들어오면 pool로부터 새로운 client connection을 맺어준다. check_client 함수를 통해 요청된 클라이언트 명령에 따라(sell, buy, show), 이진 트리로 구축된 주식 메모리를 변경시키고, 클라이언트에게 응답을 보내준다.

2. pthread

   한 명의 producer 와 여러 명의 consumer 로 구성된 producer, consumer 모델을 사용한다. thread의 creation, destroy의 과정에서 오버헤드를 막기 위해 thread pool을 통해 thread들을 미리 만들어 놓는다. connected descriptor array를 버퍼로 잡아 놓고 기다리고 있음. 만약 connected descriptor가 있으면 클라이언트 메시지 보내고, 메시지에 따라 서버가 이진 트리로 구축된 주식 메모리를 변경시키고 클라이언트에게 응답을 해준다. 다 끝났으면 descriptor를 close한다.

B. 개발 내용

- **select**

  ✓ select 함수로 구현한 부분에 대해서 간략히 설명

  ✓ 개발 범위에서 pool에 대한 설명과, 전반적인 흐름에 대해 설명하였다. 따라서 여기서는 descriptor를 select 하고 connection 맺는 방식을 설명할 것이다. 처음 thread pool이 만들어졌을 때 listenfd만 read set에 존재한다. 클라이언트를 add 할 때 slot 찾는 과정을 거쳐야 하는데, 클라이언트와 연결된 descriptor를 set에 넣어주고, 최대 descriptor의 개수를 바꿔 줌으로써 구현이 할 수 있다. 그 후 과정은 클라이언트를 처리하는 과정인데, 클라이언트와

연결된 descriptor가 set에 존재한다면, ready된 descriptor 개수를 하나 줄인다. 이 후 요청을 읽고 이진 트리 탐색을 통해 해당 요청을 처리한 후, 클라이언트에게 응답하는 방식이다. 만약 SERVER가 EOF를 받는 다면, 이는 연결이 종료되었다는 뜻이므로, readset에서 해당 descriptor를 제거하고 클라이언트 연결을 종료시킨다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

효율적인 탐색을 위해 이진 탐색 트리를 구축하여, 프로그램이 실행하는 시작 구간에 파일을 읽으면서, 이진 탐색트리에 stockinfo를 insert 하는 과정을 넣었다. 한 클라이언트가 처리될 때마다, 디스크에 다시 메모리를 옮겨주는(write) 방식으로 stockinfo를 업데이트 하였다. 이는 pthread에서 구현한 방식과 같다.

- **pthread**

✓ pthread로 구현한 부분에 대해서 간략히 설명

기존에 connfd 가 왔을 때 thread 생성시켜주는 방법 대신 이미 가용된 thread들을 만들어 놓고 클라이언트가 오면 처리해주는 thread pool 방식을 앞서 개발 범위에서 설명하였다. 여기서는 pthread를 이용해 어떻게 concurrency 하게 작동했는지 설명한다. 여러 클라이언트가 들어올 때 읽기 읽기는 상관없지만, 읽기쓰기, 쓰기쓰기는 동시에 실행 되서는 안된다. 따라서 이는 Semaphores의 reader - writers problem을 이용하여 해결한다. lock을 현재 읽고 쓰는 노드에 한정해 lock과 release를 하기 때문에 critical section을 줄일 수 있었다. buy, sell 하는 노드에는 쓰기 mutex 를, show 는 읽기 mutex를 걸고, 읽고 있는 클라이언트가 한명일 때 쓰기 mutex 에 lock 을 걸어 주고, 클라이언트가 0명일 때 읽고 mutex를 release 하는 방식으로 구현하였다.

C. 개발 방법

1) 이진 트리 구조체

바이너리 트리 노드의 멤버 변수에는 mutex(reader 를 구현할 때 쓰이는 mutex) ,w (쓰기를 구현할 때 쓰이는 mutex), left (왼쪽 노드를 가리키는 포인터) , right (오른쪽 노드를 가리키는 포인터) , 그리고 STOCK 의 정보를 갖고 있는 data 로 구성 되어있다

data는 또 하나의 구조체로 멤버 변수로는 readcnt (읽는 cnt의 개수), price, left_stock을 갖고 있다.

2) thread : semaphore – reader writer problem

buy, sell 하는 노드에는 w 세마포를 설정하고 그 안의 critical section에서만 write를 할 수 있도록 설정한다, show는 접근을 제어하는 mutex를 설정한다. show 함수에다가는 읽고 있는 클라이언트가 한명일 때 w에 lock을 걸어 주고, 클라이언트가 0명일 때 w를 release 하는 방식으로 구현하였다.

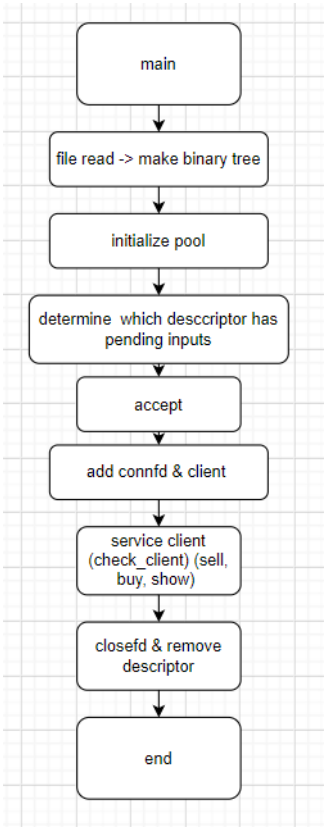
3) event 기반

이벤트는 thread 와 달리, 논리적인 control flow가 단 하나여서 thread 와 같이 의도치 않은 data sharing을 걱정할 필요가 없다. 즉 접속한 client를 핸들링 하다가 다른 pending input을 처리할 수 없다. 따라서 B에서 설명한 대로, select의 방식으로 현재 들어온 파일 디스크립터를 확인하고, 왔을 시 클라이언트의 요청을 처리해주면 된다.

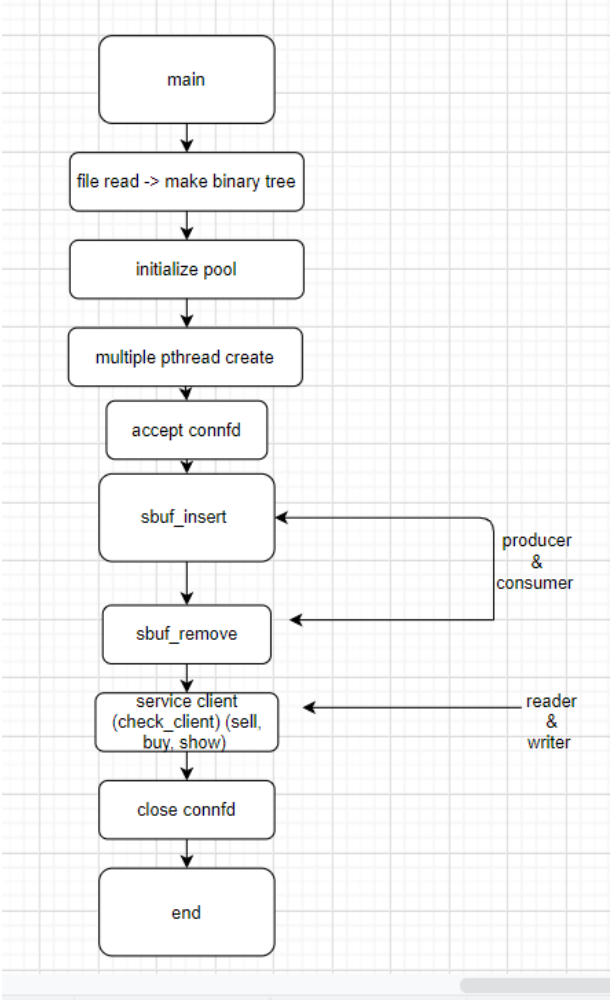
3. 구현 결과

A. Flow Chart

1. select



2. pthread



B. 제작 내용.

1. select

listenfd 가 FD_ISSET 함수를 통해 connfd 확인. 만약 accept 함수를 통 connfd가 accept 될 시, add_client 함수 호출. check_clients 함수를 통해 연결된 connfd 메시지 처리. 메시지 처리할 때 sell이면 Sellfunc 함수 호출, buy 일 때 Buyfunc 호출, Show 일 때 Showfunc 호출한다.

2. pthread

처음에 critical section을 어느 정도로 잡아야 할지 몰라, 노드 하나가 아닌 함수 전체를 건적이 있었다. 이는 매우 느렸고 몇번의 시행 착오 끝에 값을 수정하는 구간(buy 를 해서 주식의 개수 바뀌는 경우, sell을 해서 주식의 개수가 바는 경우) 을 writer 로, show 읽는 구간을 reader 로 구현할 수 있었다. node를 구하는

것은 해당 클라이언트가 요청한 stock ID를 기준으로 BSTSEARCH 함수에서 해당하는 노드 포인터를 구한다음 해당 mutex를 걸어주는 방식으로 진행하였다.

1,2 번 둘 다 공통된 문제점으로 서버가 클라이언트에게 메시지를 보낼 때 Rio_readlineb 에서는 ₩n을 기준으로 한 줄을 처리하기 때문에, 여러 줄을 보낼 때 문제점이 있었다. 따라서, show를 했을 때 문제점에 봉착했는데 서버가 보낸 줄 수를 클라이언트가 for문으로 반복하여 Rio_readlineb 하면서 해결하였다.

C. 시험 및 평가 내용

- 구현하는 것은 select는 coarse grained 한 형식이어서 동시성을 처리하는 데 문제점이 없어서 thread에 비해서 처리하는 부분에 있어서 좋았다. 반면에 pthread는 fine grained 한 concurrency 이기 때문에 노드 단위로 mutex를 걸어줘야 하기 때문에 동시성을 처리하는 데에서 어려움이 있었다.
- 성능상으로는 pthread는 thread pool을 사용하지 않는다면, event based 보다 creation, destroy 하는 데에 있어서 overhead가 더 있을 것이다. 하지만 thread pool을 사용하여서 이에 대한 overhead를 줄일 수 있어서 측정 값의 성능이 비교적 좋을 것이다.

- **실제 실험을 통한 결과 분석 (그래프 삽입)**

- 정확한 실험을 위해 다양한 방법을 통해 처리 시간을 측정하였다. 각각의 THREAD, EVENT based 방법에 대해 케이스를 나누어 (1. buy 또는 sell 일 경우, 2. show만 나타날 경우, 3. 모든 경우의 수가 섞어서 나올 경우) 5번 실험하고 이를 client의 개수를 다르게 하여 5개, 10개, 20개를 측정해 도출한 평균값을 바탕으로 다음과 같은 그래프를 제작하였다.

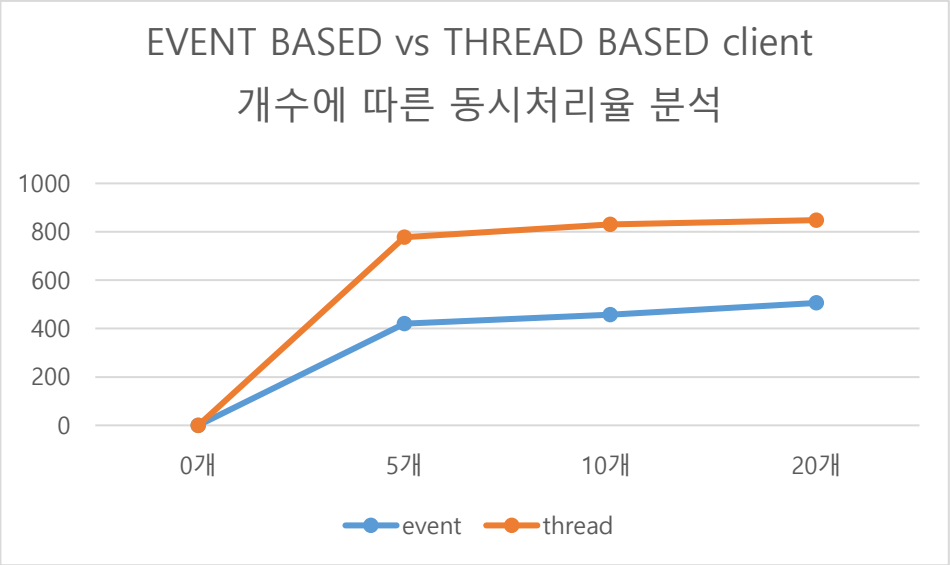
1	event	5개	1	2	3	4	5	평균
2	1.buy or sell		0.0117	0.0119	0.0119	0.0116	0.0119	0.0118
3	2. show 만		0.0133	0.0134	0.012	0.0125	0.012	0.01264
4	3. buy, show 섞어서		0.012	0.0092	0.0109	0.0122	0.0121	0.01128	0.01191
5		10개						
6	1.buy or sell		0.0224	0.0219	0.0191	0.0194	0.0229	0.02114
7	2. show 만		0.023	0.0232	0.0236	0.0271	0.0234	0.02406
8	3. buy, show 섞어서		0.0192	0.02	0.021	0.0205	0.0209	0.02032	0.02184
9		20개						
10	1.buy or sell		0.0399	0.0305	0.0411	0.0389s	0.0410s	0.03717
11	2. show 만		0.037	0.0446	0.0371	0.0447	0.0464	0.04196
12	3. buy, show 섞어서		0.0372	0.0383	0.042	0.0425	0.0374	0.03948	0.03954
13	pthread	5개	1	2	3	4	5	
14	1.buy or sell		0.0062	0.0059	0.006	0.0055	0.006	0.00592
15	2. show 만		0.0066	0.0072	0.007	0.0071	0.0069	0.00696
16	3. buy, show 섞어서		0.0068	0.0066	0.0055	0.0067	0.0064	0.0064	0.00643
17		10						
18	1.buy or sell		0.0116	0.0113	0.0112	0.0114	0.0112	0.01134
19	2. show 만		0.0125	0.0126	0.0135	0.013	0.0129	0.0129
20	3. buy, show 섞어서		0.012	0.0127	0.0117	0.0114	0.0116	0.01188	0.01204
21		20						
22	1.buy or sell		0.0211	0.0233	0.0221	0.0223	0.0225	0.02226
23	2. show 만		0.0255	0.0255	0.0254	0.0255	0.0255	0.02548
24	3. buy, show 섞어서		0.0229	0.0236	0.0221	0.0232	0.0232	0.023	0.02358
25								

이를 통해 client 개수 각각 5,10,20개에 따라 동시 처리율을 구하였다.

위의 표를 보면 event, thread 방식 모두 inorder 방식을 통하여 모든 노드를 도는 show 명령어가 시간이 가장 오래 걸림을 확인할 수 있다. case 1(buy or sell) 와 case 3 (buy, show 섞어서는) 는 우세정도를 측정하기가 어려웠다.

	0개	5개	10개	20개
event	0	419.933	457.875	505.874
thread	0	778.008	830.565	848.176

각각의 케이스 (case1 buy or show, case 2: show만 , case 3 : buy 만) 에서의 평균 값을 통해 시간 당 클라이언트 개수를 구하였다.



이를 그래프로 표시하였다. 결과는 thread 가 더 동시처리율 면에서 높은 결과를 얻음을 알 수 있다. 수업시간에서는 생성 /소멸에서 event가 overhead 가 가장 낮고, thread 가 medium 한 결과값을 나타낸다고 나와있지만 여러 thread를 미리 생성해 놓았음으로 overhead가 크지 않을 것이라 생각하였다. 또한 이러한 결과가 나온 가장 큰 이유는 event는 fine-grained concurrency를 제공하기 어렵다는 점이다. 반면에 thread 에서는 노드 단위로 세마포를 설정함으로써 critical section을 작게 만들어 동시 처리율을 높였다. 이로 인해 thread 와 event 사이에 다음과 같은 결과가 나옴을 예측할 수 있다.