**System Programming Project 3**

담당 교수 : 김영재 교수님

이름 : 김종원

학번 : 20211523

1. **개발 목표**

이번 프로젝트에서 구현하고자 하는 내용은 여러 client들의 동시 접속과 동시 서비스를 위한 Concurrent한 주식 서버(stock server)를 구축하는 것이다.

먼저, 주식 서버는 주식 정보를 저장하고 있고 여러 client들과 통신하여, 주식 정보인 주식 번호, 주식의 남은 개수와 가격과 같은 정보를 보관하고, 주식 구매(buy), 주식 판매(sell), 주식 정보 보기(show)와 같은 동작을 수행한다.

다음으로, 주식 클라이언트는 동시에 여러 개가 작동하며, 각각 server에 주식 구매(buy), 주식 판매(sell), 주식 재고 조회(show)와 같은 동작 등을 요청을 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Task 1: Event-driven Approach

서버는 select() 기반의 외부에서 발생하는 이벤트를 중심으로 프로그램 흐름을 제어하는 Event-driven Approach방식을 적용하여 여러 클라이언트를 동시 서비스할 수 있도록 구현했다.

시작 시 stock.txt를 BST(Binary Search Tree)에 로드하고, select()로 listen\_fd 및 각 conn\_fd를 모니터링하면서 show, buy, sell 명령을 처리하여 그에 맞는 응답을 반환하였다.

클라이언트가 exit이나 소켓 종료를 발생시키면 해당 연결을 정리하고 Ctrl-C(SIGINT) 시그널이 들어오면 더 이상 새 연결을 받지 않으면서 남은 클라이언트를 처리하였다.

모든 클라이언트가 종료된 후, 즉, 진행 중이던 마지막 클라이언트가 종료되어 남은 클라이언트가 없는 경우와 SIGINT 처리가 이뤄진 경우 후에는 BST에 반영된 재고 정보를 stock.txt에 저장하여, 다음 실행 시에도 상태가 유지될 수 있도록 하였다.

1. Task 2: Thread-based Approach

Task1에서 task2로 넘어오면서, 대부분의 사용 방법은 동일하지만, 구동 방식에서 많이 달라졌다.

Thread-based Approach 방식을 구현하여, master thread가 accept()로 들어오는 연결을 받아 connection queue에 저장하고, N개의 client마다 하나씩 worker thread를 줘서, 총 N개의 worker threads가 queue에서 conn\_fd를 꺼내 service\_client()를 실행함으로써 다수의 클라이언트를 동시에 처리하게 구현했다.

전역 pthread\_rwlock\_t를 사용해 show 명령에는 다중 읽기를, buy/sell 명령에는 배타적 쓰기를 적용하여 BST의 데이터 일관성을 보장하며, 변경된 데이터는 task1과 동일하게 SIGINT로 강제 종료 시 또는 모든 클라이언트가 종료되어 현재 서버에 남은 클라이언트가 없는 경우에 stock.txt에 현재 주식 정보를 저장되게 구현하였다.

1. Task 3: Performance Evaluation

Task 3에서는 task1의 Event-driven 방식과 task2의 Thread-based 방식으로 구현된 두가지 서버를 각각 동일한 조건 아래에서 실행하여 성능을 측정하고 비교한다.

구체적으로, multiclient 프로그램을 수정하여 여러 클라이언트를 동시에 생성하고, 각각이 ORDER\_PER\_CLIENT만큼 요청을 전송할 때 전체 요청을 처리하는 데 걸린 경과 시간인 elapsed time을 측정하여 이를 통해 비교한다.

측정된 경과 시간은 밀리초(ms)로 계산한다. 이를 토대로 처리율(throughput)을 계산하거나, 워크로드 유형별로 그리고 클라이언트 수 별로 클라이언트 수 증가에 따른 처리 시간 변화 즉 확장성을 분석하여 어느 접근 방식이 더 효율적인지를 알아보는 것이 목표이다.

* 1. **개발 내용**
* **Task1 (Event-driven Approach with select())**
  + Multi-client 요청에 따른 I/O Multiplexing 설명

select() 호출을 통해 듣기 소켓인 listenfd와 연결된 클라이언트 소켓들인 connfd들을 하나의 이벤트 루프 안에서 감시하게 구현했다.

이벤트가 발생한 소켓에서만 읽기와 쓰기 작업을 수행함으로써 단일 프로세스만으로도 다수의 클라이언트 요청을 효율적으로 처리할 수 있는 구조를 만들었다.

* + epoll과의 차이점 서술

select()는 감시할 연결된 클라이언트 소켓들인 connfd들의 개수가 늘어날수록 매번 전체 fd 집합을 스캔해야 하므로 오버헤드가 커진다는 단점이 있다.

반면 epoll은 커널 내부에서 이벤트의 리스트를 유지하기 때문에, 대량의 동시다발적인 연결을 처리할 시에 일어나는 오버헤드가 상대적으로 작을 수 있다는 장점이 있다.

* **Task2 (Thread-based Approach with pthread)**
  + Master Thread의 Connection 관리

Master Thread는 Open\_listenfd(port)로 생성된 듣기 소켓인 listenfd에서 accept()를 호출하여 새로운 클라이언트 연결해주는 connfd를 얻는다.

연결을 얻으면, 전역 연결 큐인 connection queue에 connfd를 삽입하는 함수인 enqueue를 통해 큐에 넣는다.

SIGINT인 CTRL+C 시그널을 받으면, shutdown\_requested 플래그를 설정하고 듣기 소켓인 listenfd을 닫아 더 이상 새로운 연결을 받지 않는 구조로 구현을 하였다.

* + Worker Thread Pool 관리하는 부분에 대해 서술

프로그램 시작 시 4개의 Worker Thread를 생성하여 대기 상태에 둔다.

Worker Thread들은 dequeue() 함수를 통해 연결 큐인 connection queue에서 connfd를 꺼내오고, 해당 소켓에 대해 service\_client(connfd)를 호출하여 요청을 처리하게 만들었다.

모든 클라이언트가 연결을 종료하면(active\_clients == 0), save\_stock("stock.txt")를 호출하여 메모리상의 주식 정보를 파일로 덮어쓴다.

* **Task3 (Performance Evaluation)**
  + 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

Metric – 경과 시간(elapsed time, ms)

클라이언트 요청 총량이 동일한 상황에서, 동시성 기법별로 동시 처리에 소요된 전체 시간을 비교한다.

Throughput(req/s) 대신 경과 시간을 직접 측정하는 이유는, 동일한 총 요청 수를 처리하는 데 걸린 절대 지표를 직관적으로 파악할 수 있기 때문이다.

->측정 방법

- 서버: cspro 서버

- 클라이언트: cspro9 서버

- 주식 초기 상태: 10개의 주식/ 시작 개수: 999개 / 가격 랜덤

ORDER\_PER\_CLIENT = 10, STOCK\_NUM = 10, BUY\_SELL\_MAX = 10

쓰레드 개수: 1, 2, 3, 5, 8, 10, 13, 15, 18, 20개

all 버전 – Multiclient가 show/buy/sell 요청을 1/3 비율로 섞어 전송함

show만 버전 – Multiclient가 오직 show 요청만 전송함

buy만 버전 – Multiclient가 오직 buy 요청만 전송함

sell만 버전 – Multiclient가 오직 sell 요청만 전송함

실행 순서:

* task 1일 때는 task1 파일에서, task2일 때는 task2 파일에서 서버를 실행한다.
* Multiclient를 ./multiclient <IP> <port> <client\_num>를 통해 클라이언트 수를

조절해가며 실행한다.

* multiclient 실행 직후에 gettimeofday(&start, NULL)을 호출해서 시간을 얻는다.
* Multiclient가 모든 요청을 종료한 순간에 gettimeofday(&end, NULL) 다시 호출

하여 시간을 얻는다.

* elapsed\_ms = (end.tv\_sec - start.tv\_sec)\*1000 + (end.tv\_usec - start.tv\_usec)/1000

위와 같은 계산 식으로 밀리초 단위(ms) 경과 시간을 계산한다.

각 조합에 대해 5회 이상 반복 수행하여, 서버 이상과 같은 현상으로 인한

이상치 제외 후 최종 평균값 사용한다.

* + Configuration 변화에 따른 예상 결과 서술

Event-driven:

클라이언트 수가 증가할수록 select() 호출 및 모든 fd 스캔 비용이 누적될 것으로 예상된다. 따라서, 클라이언트 수가 증가할 수록 경과 시간이 급격하게 증가할 것으로 예상한다.

Thread-based:

Worker Thread Pool을 통해 병렬 처리 이점을 활용하여, 클라이언트 수가 증가해도 성능이 저하되는 폭이 상대적으로 완만할 것으로 예상된다.

* 1. **개발 방법**
* Task 1

load\_stock(const char \*filename) 함수로 stock.txt를 fopen한 뒤 fscanf()를 반복하여 (ID, left\_stock, price) 데이터를 읽고 item\_t 노드를 동적으로 생성한다

insert\_item(item\_t \*node, item\_t \*new)를 재귀하여 호출하여 BST 즉 Binary Search Tree 구조로 삽입하며, 각 노드는 다음과 같은 형태로 정의했다.

**typedef struct item {**

**int id;**

**int left\_stock;**

**int price;**

**struct item \*left;**

**struct item \*right;**

**} item\_t;**

Open\_listenfd(port)로 듣기 소켓을 생성하고 listen() 호출 후 listenfd를 얻고,

전역 변수로 fd\_set master\_set, read\_set; int maxfd; rio\_t rio\_pool[FD\_SETSIZE]; int shutdown\_requested = 0; int active\_client\_count = 0; 등을 선언한다.

FD\_ZERO(&master\_set); FD\_SET(listenfd, &master\_set); maxfd = listenfd;로 초기화한 뒤에

**Signal(SIGINT, sigint\_handler);**

**while (!shutdown\_requested || active\_client\_count > 0) {**

**read\_set = master\_set;**

**int nready = Select(maxfd + 1, &read\_set, NULL, NULL, NULL);**

**if (FD\_ISSET(listenfd, &read\_set) && !shutdown\_requested) {**

**int connfd = Accept(listenfd, &clientaddr, &clientlen);**

**Rio\_readinitb(&rio\_pool[connfd], connfd);**

**FD\_SET(connfd, &master\_set);**

**if (connfd > maxfd) maxfd = connfd;**

**active\_client\_count++;**

**}**

**for (int fd = 0; fd <= maxfd && nready > 0; fd++) {**

**if (!FD\_ISSET(fd, &read\_set) || fd == listenfd) continue;**

**nready--;**

**int ret = handle\_request(fd);**

**if (ret < 0) {**

**Close(fd);**

**FD\_CLR(fd, &master\_set);**

**active\_client\_count--;**

**if (active\_client\_count == 0 && shutdown\_requested) {**

**save\_stock("stock.txt");**

**}**

**}**

**}**

**}**

sigint\_handler(int sig): shutdown\_requested = 1; Close(listenfd);를 수행하여 select()를 빠져나오게 유도한다.

handle\_request(int connfd) 내부에서 Rio\_readlineb(&rio\_pool[connfd], buf, MAXLINE)로 요청을 한 줄 읽고, sscanf(buf, "%s %d %d", cmd, &id, &num)로 파싱하는데, 각 단어 별로 행동은 다음과 같이 일어난다.

**show**: print\_stock(connfd, root) 호출하고

print\_stock()는 build\_stock\_str(item\_t \*node, char \*out)를 재귀 호출하여 BST 전체를 문자열로 누적하고, Rio\_writen(connfd, out, MAXLINE)로 전송한다.

**buy**: item\_t \*it = find\_item(root, id)로 노드를 탐색하고

if (!it) sprintf(out, "Invalid stock ID: %d\n", id);

else if (it->left\_stock >= num) { it->left\_stock -= num; sprintf(out, "[buy] success\n"); }

else sprintf(out, "Not enough left stocks\n"); Rio\_writen(connfd, out, MAXLINE);

**sell**: item\_t \*it = find\_item(root, id);로 노드를 탐색하고

if (!it) sprintf(out, "Invalid stock ID: %d\n", id);

else { it->left\_stock += num; sprintf(out, "[sell] success\n"); } Rio\_writen(connfd, out, MAXLINE);

**exit**: return -1; 하여 연결 종료를 알린다.

서버 종료 시점과 마지막에 있던 클라이언트 종료시점에 저장하는데 방법은 다음과 같다.

if (active\_client\_count == 0 && shutdown\_requested) 시점에

printf("All clients done, saving stock.txt...\n");

fopen(filename, "w"), inorder\_save(root, fp), fclose(fp) 순으로 수행하는 save\_stock("stock.txt")를 호출한다.

* Task 2

이번에는 task1에서 구현한 내용을 어느정도 이용하면서 task2의 내용을 구현한다.

item\_t에서 pthread\_rwlock\_t lock;를 추가하여 노드 단위의 RW 락을 만들어준다.

또한, pthread\_rwlock\_init(&node->lock, NULL)로 각 노드의 RW 락을 초기화한다.

Signal 핸들러인 Signal 함수에서는 SIGINT가 들어오면 종료하게 shutdown\_requested = 1; Close(listenfd);를 넣어준다.

Master thread가 사용하는 worker thread 4개를 만들기 위해

**pthread\_t tids[NTHREADS];**

**for (int i = 0; i < NTHREADS; i++) {**

**Pthread\_create(&tids[i], NULL, worker\_thread, NULL);**

**}**

위와 같은 코드도 작성한다.

추가적으로 enqueue(int connfd), dequeue(void)를 만들어 연결 큐(Connection Queue)에 넣고 뺼 수있게 구현해주고 전역변수를 다음과 같이 만들어준다.

**conn\_node\_t \*q\_head = NULL, \*q\_tail = NULL;**

**pthread\_mutex\_t queue\_mutex = PTHREAD\_MUTEX\_INITIALIZER;**

**pthread\_cond\_t queue\_cond = PTHREAD\_COND\_INITIALIZER;**

**int active\_clients = 0;**

**int shutdown\_requested = 0;**

이를 통해 Connection Queue를 완성해준다.

worker\_thread(void \*vargp)에서 service\_client(int connfd)를 실행하고 service\_client 내부에서 클라이언트 요청(“show”, “buy”, “sell”, “exit”)을 처리한다.

1. **구현 결과**

1) Task 1 (Event-driven) 구현 결과

stockserver1.c를 make하여 실행하면, ./stockserver1 <port>로 서버가 시작된다.

stockclient.c 또는 multiclient.c를 통해 show, buy, sell, exit 명령을 순차 혹은 동시 다발적으로 전송할 때, 서버는 select() 루프를 통해 각 클라이언트의 I/O 이벤트를 감시하고, 요청에 따라 메모리 상 BST를 읽거나 수정 후 응답을 반환한다.

모든 클라이언트가 종료되면(active\_client\_count == 0), “All clients done, saving stock.txt…” 메시지와 함께 메모리 상 BST 내용을 stock.txt에 덮어쓴다.

2) Task 2 (Thread-based) 구현 결과

stockserver2.c를 make하여 실행하면, ./stockserver2 <port>로 서버가 시작된다.

Master Thread가 연결을 대기하며, Worker Thread Pool(4개)이 미리 생성되어 dequeue()를 통해 연결을 분배받는다.

클라이언트는 stockclient.c 또는 multiclient.c를 사용하여 동시 접속 후 여러 요청을 전송할 수 있으며,

Worker Thread 각각이 service\_client() 내부에서 pthread\_rwlock을 사용하여 BST 노드 단위 동기화를 수행한 뒤 응답을 반환한다.

모든 클라이언트가 종료되면(active\_clients == 0), “All clients disconnected, stock.txt saved.” 메시지와 함께 save\_stock()를 수행한다.

1. **성능 평가 결과 (Task 3)**

방법은 위에 작성한 것과 같아서 간단하게 설명 후 결과를 말하려고 한다.

서버는 cspro에서, 클라이언트는 cspro9에서 실행했고

버전은 4개로 all, show만, buy만, sell만으로 4가지 버전으로 성능을 task1, 2 모두에서 4가지로 측정했다.

각 버전마다 1-2-3-5-8-10-13-15-18-20은 모두 진행했고, all은 50, 100도 진행했다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | | task1 | | | task2 | | |
| 횟수 | random | show | buy | sell | random | show | buy | sell |
| 1 | 5992 | 6586 | 5941 | 6042 | 5919 | 5977 | 5706 | 5912 |
| 2 | 7699 | 6918 | 6580 | 6855 | 6016 | 7031 | 7197 | 6706 |
| 3 | 8361 | 9693 | 8459 | 7751 | 8103 | 7999 | 8008 | 7534 |
| 5 | 10168 | 10389 | 10315 | 10730 | 9274 | 10077 | 9555 | 9246 |
| 8 | 14325 | 16004 | 13863 | 13031 | 12524 | 12993 | 12546 | 12237 |
| 10 | 18773 | 19264 | 18081 | 17314 | 15769 | 15969 | 14893 | 13308 |
| 13 | 22611 | 21860 | 21449 | 21203 | 18614 | 19748 | 17114 | 17913 |
| 15 | 25957 | 26842 | 25418 | 24574 | 20265 | 21856 | 18399 | 18620 |
| 18 | 27885 | 29815 | 26940 | 27637 | 23901 | 23969 | 22063 | 22765 |
| 20 | 30882 | 31513 | 29202 | 29754 | 26067 | 27135 | 24708 | 23481 |
| 50 | 64763 |  |  |  | 53955 |  |  |  |
| 100 | 114056 |  |  |  | 98001 |  |  |  |

각 task의 모든 버전의 ms 시간

**텍스트, 스크린샷, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

예시 출력 화면(task1 - all 버전 - 20개)

텍스트, 스크린샷, 도표, 그래프이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

각 task별 all(=random), show만, buy만, sell만 버전 보여주는 차트

라인, 그래프, 도표, 텍스트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

각 task별 all 버전 비교 버전

1. 확장성(Expandability)

클라이언트 수가 증가함에 따라 두 방식에서 모두 경과 시간이 증가하지만, Event-driven 방식은 select()의 fd 집합 스캔 비용이 비교적 조금 더 기하급수적으로 많이 늘어나서 20명 이상에서는 경과 시간 증가폭이 매우 커진다.

Thread-based 방식은 Worker Thread Pool을 통한 병렬 처리와 pthread\_rwlock 기반 동시성 제어가 잘된 덕분인지 같은 구간에서도 항상 비교적 낮은 경과 시간을 유지하며 확장성을 상대적으로 잘 견디는 모습을 보인다.

2. 워크로드 유형(Workload type)에 따른 차이

* Show(읽기 전용)

Event-driven은 매 요청마다 전체 BST를 순회하며 select()를 다시 호출하므로, 다수 클라이언트가 동시에 읽기만을 수행할 때도 오버헤드가 크다.

Thread-based는 pthread\_rwlock\_rdlock을 통해 다중 읽기를 병렬 처리할 수 있어, 클라이언트 수 증가 시 성능 저하 폭이 작다.

* Buy/Sell(쓰기 전용)

Event-driven은 락 없이 직접 메모리 수정하지만, 다수의 동시 쓰기 요청이 들어오는 환경에서는 경쟁 상태가 발생하면서 내부적으로 순차화되어 지연이 누적된다.

Thread-based는 pthread\_rwlock\_wrlock 기반 쓰기 락으로 순차적으로 처리하면서도 Worker Pool을 통해 병렬화에서 이점을 어느 정도 유지하기 때문에, 성능 저하 폭이 상대적으로 작다.

* All(혼합)

Event-driven은 읽기와 쓰기를 모두 단일 루프에서 처리해야 하므로, 읽기·쓰기 경쟁 상황이 복합적으로 발생하여 가장 큰 경과 시간 증가를 보인다.

Thread-based는 읽기 시 다중 읽기 병렬 처리, 쓰기 시 순차 처리 구조를 모두 활용할 수 있어, Random 워크로드에서도 Event-driven 대비 일관되게 우수한 성능을 기록한다.

결과적으로, show > all > buy = sell 이런 순서이다.

Show가 가장 많은 시간이 들었고, buy와 sell은 비슷한 시간으로 show보다 적은 시간이 들었으며 all은 아무래도 모든 기능을 사용한 버전이라서 show보다는 적지만, buy와 sell보다는 많은 시간이 소요되었다.

또한, 전체적으로 task1이 task2보다 시간이 더 많이 소요되는 양상을 띄었다.