System Programming Project 3

담당 교수: 이영민

이름: 방지혁

학번: 20190808

1. 개발 목표

프로젝트3에서는 여러 사용자들의 처리를 동시에 처리할 수 있는 주식 서버를 구현하고자 한다. Concurrent programming 방식으로는 pool을 사용하는 even-driven 방식과 thread를 사용하는 thread-based 방식이 있는데 이를 각각 task1, task2에 사용한다. Client는 server에게 show(현재 주식 상태를 보는 것), buy(해당하는 id의 주식을 사는 것), sell(해당하는 id의 주식을 파는 것), exit(connection을 나가는 것) 4종류의 요청을 할 수 있으며, server는 stock.txt에서 주식 정보를 읽어 binary tree에 저장했다가 이를 바탕으로 서비스를 제공한다. client 개수가 0이 되거나 server가 종료되는 경우 이를 다시 stock.txt에 저장한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

Task 1에서는 select()함수를 활용한 I/O multiplexing을 통해 여러 client들이 concurrency를 지원하는 주식 서버를 구현한다.

Stock의 ID를 key로 하는 binary search tree를 통해 해당 정보들을 관리한다. stock.txt로부터 데이터를 얻어와 위의 자료구조에 저장하고, program 종료 시 변경사항을 다시 text 파일에 저장한다.

2. Task 2: Thread-based Approach

Master-Worker thread를 기반으로 concurrency를 지원하는 주식 서버를 구현한다. Thread Pool과 producer-consumer pattern을 응용한다.

Master Thread의 경우 producer로서 client의 connection request를 수락하고 해당 descriptor를 shared buffer에 삽입한다. Worker Thread Pool의 경우 consumer로서 buffer에서 descriptor를 빼서 client의 request를 처리한다. 또한, semaphore 및 mutex를 활용하여 Reader-Writer 문제를 해결해 race condition을 방지한다.

3. Task 3: Performance Evaluation

Task1과 task2 각각의 서버 구현 방법에 대해 동시 처리율을 여러 방법 및 기준

으로 테스트해보며 차이점을 비교한다. Client 개수를 늘려보고, client 요청 타입 중 어떤 것이 제일 많이 시간이 걸리는지 코드를 바꿔보면서 구현한 주식 서버에서 무엇이 더 적합한지 확인해본다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  Task1에서 구현했던 해당 select() 기반 I/O 멀티플렉싱은 여러 프로세스를 만들지 않고도 단일 프로세스에서 여러 클라이언트의 요청을 concurrent하게 처리하는 방법이다. server는 모든 active한 descriptor를 fd_set 구조체에 등록하고, select() 시스템 콜을 통해 읽기 준비가 완료된 소켓들만 선택한다. 이 방식의 핵심은 기존과 다르게 blocking 없이 여러 I/O 채널을 모니터링할 수 있다는 점이다.

  서버는 계속 루프를 돌면서 새로운 연결 요청이 들어오면 listenfd에서 accept()를 수행하여 새 클라이언트를 read_set에 추가한다. select()를 호출하여 준비된 소켓이 있을 때까지 대기하고, 반환되면 FD_ISSET 매크로로 각 소켓의 상태를 확인하여 기존 클라이언트로부터 데이터가 도착하면 읽기 작업을 수행한다.

  ✓ epoll과의 차이점 서술

  select()의 경우 등록된 file descriptor를 모두 선형적으로 스캔하기에 O(n)의 시간 복잡도를 가며 FD_SETSIZE가 1024로 제한되어 있기 때문에 그 이상의 client와 연결할 수 없다. 또한, active 상태인 file descriptor의 관리 및 새로운 호출이 있을 시 read_set을 복사하여 ready_set으로 설정하는데 이 모든 과정에 있어 kernel-level이 아니라 user-level에서 이루어져 효율성이 떨어진다.

  반면 epoll의 경우 kernel 내부에서 알아서 관리하기에 O(1)의 시간 복잡도를 가지며 변경이 된 파일 디스크립터만 반환한다. 그렇기에 메모리 사용 측면에 있어 select()와 달리 모든 비트맵을 복사할 필요가 없기에 메모리 효율적이다. 성능의 측면에서는 연결된 client 수가 적을 때는 select()가 효율적이지만, 대규모 연결에서는 epoll이 압도적으로 우수하다. 단 epoll은 linux전용이라는 단점도 존재한다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 client의 connection request 수락부터 Worker Thread에게 이후의 작업을 넘겨주는 것까지 한다. 서버 시작 시 listen socket을 생성하고, 무한 루프를 통해 accept() 시스템 콜로 클라이언트 연결을 지속적으로 대기한다.

  Client의 connection request가 있을 시 accept()함수가 반환하는 connfd를 공유 버퍼인 sbuf에 삽입한다. 또한 활성 클라이언트 수 카운터인 active_clients를 세마포어로 보호하면서 관리하여, 클라이언트 수를 추적하여 모든 클라이언트가 종료되었을 때 stock.txt에 저장이 될 수 있도록 한다.

  즉 Master Thread는 client의 연결 및 수락을 담당하고 Worker Thread들은 순수하게 client service를 담당하는 구조이다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Worker Thread Pool은 서버 초기화 시점에 미리 생성되는 고정 크기의 스레드 집합이다. 해당 Pool의 각 Worker Thread는 pthread_create()로 생성되고 pthread_detach()를 호출하여 독립적으로 동작하며, shared buffer에서 작업을 가져온다. sbuf_remove() 함수를 통해 블로킹 방식으로 작업을 대기하며, 작업이 없을 때는 자동으로 대기 상태에 들어갔다가 작업이 할당되면 깨어나서 client의 request를 처리한다. 이와 같은 방식을 쓰레드를 동적으로 할당하지 않기에 효율적이다. 해당 pool의 크기는 시스템 자원과 예상 작업량을 고려하여 결정되며, 너무 적으면 전체 처리량이 제한되지만 너무 많으면 context switching 측면에서 비효율적이다.

- **Task3 (Performance Evaluation)**

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

  Task3에서는 동시처리율을 핵심 성능 지표로 선정했다. 동시처리율은 단위 시간당 완료된 클라이언트 요청 수를 의미하며, client 개수 * client 당 요청 개수 / 시간으로 계산할 수 있다.

  이로 선정한 이유는 주식 거래 시스템에서 서버를 늘리지 않고 동시에 많은 거래 요청을 처리할 수 있는 능력이 시스템의 안정성 나아가서 사업의 경쟁

력을 결정하기 때문이다.

측정 방법은 gettimeofday() 함수를 사용하여 시작과 종료 시점을 기록하고, 모든 클라이언트가 완료한 총 요청 개수를 시간으로 나누어 계산한다.

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트 수 증가에 따른 성능 변화는 task마다 서로 다른 패턴을 보일 것으로 예상된다. Event-driven 방식은 client 개수가 적을 때에는 우수한 성능을 보일 것 같지만, select()함수는 O(n)의 시간 복잡도를 가지기에 성능 저하가 늘어날 것이다. 한편 Thread-based 방식은 Worker Thread 수 내외의 클라이언트 개수까지는 괜찮은 성능을 보이다가, 해당 크기를 초과하면 컨텍스트 스위칭 비용으로 인해 성능이 늘어나지 못할 것이다.

워크로드 패턴별로는 더욱 뚜렷한 차이가 예상된다. Read 중심의 요청(show 명령)에서는 Thread-based 방식이 여러 스레드의 병렬 읽기가 가능하기에 처리량이 크게 향상될 것이다.

그러나 Write 중심의 요청(buy/sell 명령)에서는 두 방식 모두 성능이 저하되지만, Event-driven이 상대적으로 나을 것이다. 세마포어 계산이 없고 단순하게 직렬적으로 처리하기 때문이다.

C. 개발 방법

구조체

stockNode 구조체

주식 정보를 저장하는 bst node이다. id는 주식의 primary key로서 역할을 하며, left_stock은 현재 주식 수량, price는 단가를 나타낸다. file_order는 stock.txt 파일에서의 원래 순서를 기억하기 위한 필드로, show 명령 및 다시 stock.txt에 저장할 때 원래 순서대로 출력 및 저장하기 위해 사용된다. left와 right 포인터는 이진 탐색 트리의 좌우 자식 노드를 가리킨다.

pool 구조체

select() 기반 I/O multiplexing에서 핵심 데이터 구조이다. maxfd는 현재 등록된 파일 디스크립터 중 가장 큰 값으로 select() 호출 시 첫 번째 인자로 사용된다. read_set은 모든 활성 소켓 디스크립터를 담고 있는 fd_set이고 ready_set은 select() 반환 후 실제로 읽기 준비가 완료된 소켓들을 나타낸다.

nready는 select()가 반환한 준비된 디스크립터의 개수이며, maxi는 clientfd 배열에서 현재 사용 중인 가장 높은 인덱스를 나타낸다. clientfd 배열은 각 클라이언트의 연결 socket descriptor **pool 구조체**는 select() 기반 I/O 멀티플렉싱에서 다중 클라이언트 연결을 관리하는 핵심 데이터 구조이다. maxfd는 현재 등록된 파일 디스크립터 중 가장 큰 값으로 select() 호출 시 첫 번째 인자로 사용되며, read_set은 모든 활성 소켓 디스크립터를 담고 있는 fd_set이고 ready_set은 select() 반환 후 실제로 읽기 준비가 완료된 소켓들을 나타낸다. nready는 select()가 반환한 준비된 디스크립터의 개수이며, maxi는 clientfd 배열에서 현재 사용 중인 가장 높은 인덱스를 추적한다. clientfd 배열은 각 클라이언트의 연결 소켓 디스크립터를 저장하고, clientrio 배열은 각 클라이언트별로 독립적인 rio_t 버퍼를 유지하여 부분 읽기 상황을 처리한다. 를 저장하고, clientrio 배열은 각 클라이언트별로 독립적인 rio_t 버퍼를 유지하여 부분 읽기 상황을 처리한다.

전역 변수

byte_cnt 변수

서버가 클라이언트로부터 수신한 총 바이트 수를 누적하여 저장하는 정수형 전역 변수.

head 변수

주식 정보를 저장하는 BST의 루트 노드를 가리키는 stockNode 포인터 전역 변수.

original_order 변수

stock.txt 파일에서 주식들의 원래 순서를 보존하기 위한 정수 배열 포인터. 메모리는 주식 개수만큼 동적 할당되며 서버 종료 시 해제된다.

global_pool 변수

현재 활성화된 pool 구조체를 가리키는 전역 포인터. Signal handler에서 SIGINT 시그널 발생 시 이 포인터를 통해 모든 활성화된 연결을 정리하고 끊을 수 있도록 한다.

global_listenfd 변수

서버의 리스닝 소켓 디스크립터를 저장하는 전역 변수로, global_pool 변수와

마찬가지로 SIGINT 발생 시 시그널 핸들러에서 서버 소켓을 닫기 위해 사용된다.

함수

sigintHandler 함수

SIGINT 시그널(Ctrl+C) 발생 시 호출되는 시그널 핸들러. 우선 writeFile 함수를 호출하여 주식 데이터 변경사항을 stock.txt 파일에 저장하고, global_pool을 통해 모든 활성화된 소켓을 닫는다. 또한, global_listenfd를 닫아 리스닝 소켓을 닫고, free_all_stocks함수를 호출하여 BST 메모리를 해제한 후 original_order 배열의 메모리도 해제하여 서버를 종료시킨다.

init_pool 함수

초기에 client connection를 위한 pool 구조체를 설정한다. maxi를 -1로 초기화하여 현재 사용 중인 클라이언트가 없음을 표시하고, FD_SETSIZE 크기의 clientfd 배열을 모두 -1로 설정하여 모든 슬롯이 비어 있음을 나타낸다. maxfd를 listenfd로 설정하고, FD_ZERO로 read_set을 초기화한 후 FD_SET으로 listenfd를 추가하여 새로운 클라이언트 연결 요청을 받을 수 있도록 한다.

add_client 함수

새로운 client connection을 pool에 추가한다. 먼저 nready를 감소시켜 해당 select 이벤트를 처리했음을 표시하고, clientfd 배열에서 -1인 빈 슬롯을 찾아 새로운 connfd를 저장한다. 해당 인덱스의 clientrio를 Rio_readinitb로 초기화하여 독립적인 읽기 버퍼를 설정하고, FD_SET으로 read_set에 새 소켓을 추가한다. connfd가 현재 maxfd보다 크면 maxfd를 업데이트하고, 사용된 인덱스가 maxi보다 크면 maxi도 업데이트한다. 모든 슬롯이 사용 중이면 에러를 발생시킨다.

check_clients 함수

select로 준비된 클라이언트들의 요청을 순차적으로 처리하는 함수이다. 0부터 maxi까지 반복하면서 각 clientfd가 유효하고 FD_ISSET으로 읽기 준비 상태인지 확인한다. 준비된 클라이언트가 있으면 nready를 감소시키고 Rio_readlineb로 한 줄씩 데이터를 읽는다. analyze_commands를 호출하여 명령을 파싱하고 처리한다. exit 명령이거나 클라이언트가 연결을 끊은 경우

(n=0) Close로 socket을 닫고, FD_CLR로 read_set에서 제거한 후 clientfd를 -1로 설정하여 slot을 해제하고 check_and_update_file을 호출한다.

scanFile 함수

stock.txt 파일에서 주식 데이터를 읽어 메모리상의 이진 탐색 트리로 구성하는 함수이다. 먼저 파일을 열어 전체 주식 개수를 세어 original_order 배열을 동적 할당하고, rewind로 파일 포인터를 처음으로 되돌린 후 다시 읽으면서 각 주식의 ID, 재고량, 가격을 파싱한다. insertStockNode를 호출하여 각 주식을 BST에 삽입한다. 파일 읽기 완료 후 fclose로 파일을 닫는다.

analyze_commands 함수

클라이언트로부터 받은 문자열을 파싱하여 명령에 맞는 함수를 호출한다. strtok_r 인자들을 분리하고, strncmp로 명령어 종류를 판별한다.

buy_stock 함수

클라이언트의 주식 구매 요청을 처리한다. 먼저 응답 버퍼를 0으로 초기화하고 search_stock으로 해당 ID의 주식을 찾는다. 주식이 존재하지 않으면 바로 리턴하고, 존재하면 현재 재고량과 구매 요청량을 비교하여 재고가 충분하면 left_stock에서 요청량을 빼고 "[buy] success" 메시지를 출력하고 재고가 부족하면 "Not enough left stock" 메시지를 출력한다.

show_stocks 함수

주식 정보를 원래 저장 순서대로 클라이언트에게 전송하는 함수이다. MAXLINE 크기의 결과 버퍼를 0으로 초기화하고, original_order 배열을 순회하며 각 주식 ID에 대해 search_stock으로 목표하는 노드를 찾는다. 그렇게 주식 정보 포맷팅하여 Rio_writen으로 클라이언트에게 전송한다.

sell_stock 함수

클라이언트의 주식 판매 요청을 처리하는 함수로, 주식이 존재하지 않으면 바로 리턴하고, 존재하면 left_stock에 판매량을 추가하여 재고를 증가시킨다. "[sell] success" 메시지를 버퍼에 복사하고 응답을 전송한다.

writeFile 함수

메모리상의 주식 데이터 변경사항을 stock.txt 파일에 영구 저장하는 함수이

다. fopen으로 stock.txt를 열고, original_order 배열을 순회하며 각 주식 ID에 대해 search_stock으로 해당 노드를 찾는다. 찾은 주식의 ID, 현재 재고량, 가격을 fprintf로 파일에 기록하고, fclose로 파일을 닫는다.

check_and_update_file

클라이언트 수를 확인하여 클라이언트 수가 0이 되었을 때 주식시장 변경 상황 저장을 수행하는 함수이다. pool 구조체의 clientfd 배열을 순회하면서 0보다 큰 유효한 디스크립터 개수를 세어 alive_count를 계산한다. 만약 활성 클라이언트가 0명이면 writeFile을 호출하여 현재 주식 상태를 파일에 저장한다.

free_all_stocks 함수

BST의 모든 노드를 재귀적으로 해제하는 메모리 정리 함수이다.

insertStockNode 함수

새로운 주식 노드를 이진 탐색 트리에 삽입하는 재귀 함수이다. 현재 노드가 NULL이면 malloc으로 메모리를 할당하고 반환한다. 그렇지 않을 경우 삽입할 주식 ID와 현재 노드 ID를 비교하여, 작으면 왼쪽에, 크면 오른쪽에 재귀적으로 삽입한다.

search_stock 함수

주식 ID 값으로 BST에서 해당 노드를 찾는 재귀 함수이다. 현재 노드가 NULL이면 찾는 주식이 없다는 의미로 NULL을 반환하고, 찾는 ID와 같으면 해당 노드를 반환한다. 찾는 ID가 현재 노드보다 작으면 왼쪽에서, 크면 오른쪽에서 재귀적으로 탐색을 계속한다.

Task2에서 추가 사항

sbuf_t 구조체

Producer-Consumer 버퍼이다. buf는 connfd들을 저장하는 정수 배열 포인터이고, n은 버퍼의 최대 용량, front와 rear는 큐의 앞, 뒤 위치를 나타낸다. mutex는 버퍼 전체에 대한 이진 semaphore이고, slots는 빈 슬롯 수를 나타내는 counting semaphore이며, items는 처리해야 할 작업 수를 나타내는 counting semaphore이다.

stockNode 구조체 변경사항

readcnt는 Reader Thread의 수를 추적하고 mutex는 앞서 언급한 readcnt 변수에 대한 binary semaphore이고, write_lock은 Writer Thread에 대한 binary semaphore이다.

sbuf_init 함수는 공유 버퍼를 초기화하는 함수로, Producer-Consumer 패턴의 동기화 구조를 설정한다. 먼저 Calloc으로 n개 크기의 정수 배열을 할당하여 buf에 저장하고, n을 버퍼 크기로 설정한다. front와 rear를 모두 0으로 초기화하여 빈 원형 큐 상태를 만들고, Sem_init으로 mutex를 1로 초기화하여 상호 배타적 접근을 가능하게 한다. slots 세마포어를 n으로 초기화하여 모든 슬롯이 비어있음을 나타내고, items 세마포어를 0으로 초기화하여 처리할 작업이 없음을 표시한다.

sbuf_deinit 함수는 공유 버퍼의 정리 작업을 수행하는 함수로, 동적 할당된 buf 배열을 Free로 해제하여 메모리 누수를 방지한다. 세마포어들은 자동으로 정리되므로 별도의 해제 작업은 수행하지 않는다.

sbuf_insert 함수는 Master Thread가 새로운 클라이언트 연결을 공유 버퍼에 삽입하는 Producer 함수이다. 먼저 P(&slots)로 빈 슬롯이 있을 때까지 블로킹 대기하고, P(&mutex)로 버퍼에 대한 배타적 접근을 획득한다. rear를 전위 증가시키고 모듈로 연산으로 원형 큐의 다음 위치를 계산하여 해당 위치에 connfd를 저장한다. V(&mutex)로 버퍼 접근 권한을 해제하고, V(&items)로 새로운 작업이 추가되었음을 대기 중인 Worker Thread들에게 신호를 보낸다.

sbuf_remove 함수는 Worker Thread가 공유 버퍼에서 작업을 가져오는 Consumer 함수이다. P(&items)로 처리할 작업이 있을 때까지 블로킹 대기하고, P(&mutex)로 버퍼 접근 권한을 획득한다. front를 전위 증가시키고 모듈로 연산으로 다음 제거 위치를 계산하여 해당 위치의 connfd를 지역 변수에 복사한다. V(&mutex)로 버퍼 접근 권한을 해제하고, V(&slots)로 빈 슬롯이 하나 생겼음을 Master Thread에게 알린 후 읽어온 connfd를 반환한다.

thread 함수는 각 Worker Thread의 메인 실행 함수로, 스레드 풀의 개별 워커 역할을 수행한다. pthread_detach(pthread_self())로 현재 스레드를 분리 상태로 만들어 종료 시 자동으로 자원이 해제되도록 설정한다. 무한 루프에서 sbuf_remove를 호출하여 Master Thread로부터 할당된 작업을 기다리고, 작업이 도착하면 stock_service 함수를 호출하여 해당 클라이언트의 요청을 처리한다. 클라이언트 서비스 완료 후 Close로 연결을 종료하고 다시 대기 상태

로 돌아가 다음 작업을 기다린다.

추가 전역변수

mutex 변수

byte_cnt와 관련 출력에 대한 binary semaphore이다.

active_clients 변수

서버에 연결된 클라이언트 수를 위한 static 정수형 전역 변수이다.

client_mutex 변수

active_clients 변수에 대한 접근을 보호하는 binary semaphore이다.

함수

sbuf_init 함수

공유 버퍼를 초기화하는 함수. Calloc으로 n개 크기의 정수 배열을 할당하여 buf에 저장하고, n을 버퍼 크기로 설정한다. front와 rear를 모두 0으로 초기화하여 큐 상태를 만들고, Sem_init으로 mutex를 1로, slots 세마포어를 n으로, items 세마포어를 0으로 초기화한다.

sbuf_deinit 함수

종료시에 동적 할당된 buf 배열을 Free로 해제하여 메모리 누수를 방지한다.

sbuf_insert 함수

Master Thread가 새로운 connection을 sbuf에 삽입하는 함수이다. 먼저 P(&slots)로 빈 슬롯이 있을 때까지 블로킹 대기하고, P(&mutex)로 배타적 접근을 획득한다. rear를 1 증가시키고 해당 위치에 connfd를 저장한다. 이후 V(&mutex)로 배타적 접근을 해제하고, V(&items)로 새로운 작업이 있음을 다른 Thread들에게 알린다.

sbuf_remove 함수

Worker Thread가 sbuf에서 작업을 가져오는 함수이다. P(&items)로 처리할 작업이 있을 때까지 대기하다가 P(&mutex)로 배타적 접근 권한을 획득한다. front를 증가시키고 해당 위치의 connfd를 지역 변수에 복사한다. V(&mutex)로 배타적 접근 권한을 해제하고, V(&slots)로 빈 슬롯이 있음을 Master

Thread에게 알린다.

thread 함수

각 Worker Thread의 메인 함수로, 스레드 풀의 개별 워커 역할을 수행한다. pthread_detach(pthread_self())로 분리 상태로 스레드를 만들어 종료 시 알아서 해제되도록 설정한다. sbuf_remove를 호출해서 작업이 도착하면 stock_service 함수를 호출하여 해당 요청을 처리한다. 서비스 완료 후 Close로 종료하고 다시 다음 작업을 기다린다.

stock_service 함수

analyze_commands를 호출하여 명령을 파싱하고 처리하며, exit 명령을 받으면 루프를 종료한다. 연결이 종료되면 client_mutex로 보호된 영역에서 active_clients를 감소시킨다. 이는 모든 연결이 종료되면 writeFile을 호출하여 stock.txt에 저장하기 위해서이다.

init_echo_cnt 함수

Worker Thread에서 사용하는 공유 변수들을 초기화하는 함수이다.

buy_stock, sell_stock, show_stocks 함수 추가 내용

buy_stock과 sell_stock은 Writer 작업이기에 배타적 접근을 보장한 후 작업을 수행하고 해제한다. show_stocks는 Reader 작업으로, 각 노드에 대해 mutex로 readcnt를 보호하면서 첫 번째 Reader일 때 write_lock을 획득하고, 마지막 Reader일 때 write_lock을 해제한다.

insertStockNode 함수 추가내용

새 노드 생성 시 readcnt를 0으로 초기화하고, Sem_init으로 mutex와 write_lock을 각각 1로 초기화한다.

3. 구현 결과

Task 1: Event-driven Approach 구현 결과

I/O multiplexing 방식으로 주식 서버를 성공적으로 구현했다. 클라이언트가 service된 data를 읽어오는데 어려움이 있어 기존 stockclient.c의 Rio_readlineb를 Rio_readnb로 변경하였다.

BST를 사용하여 O(log n) 시간복잡도를 이룰 수 있었, stock.txt 파일의 원래 순서를 유지하기 위해 original_order 배열을 생성하였다 모든 클라이언트 명령(buy, sell, show, exit)에 대해 정상적인 응답을 확인했으며, 예상대로 동작함을 검증했다. 서버 종료 시 변경사항이 stock.txt에 정확히 반영되는 점도 확인했다.

Task 2: Thread-based Approach 구현 결과

멀티스레드 서버를 구현했다. Master Thread는 연결 수락에만 집중하고, 실제 요청 처리 및 서비스는 Worker Thread들이 담당한다. sbuf를 구현하여 안전한 큐를 구현했으며, 여러 세마포어의 조합으로 데드락이 없도록 했다.

성능 검증 결과, 모든 기능이 정상 동작함을 확인했으며, 특히 race condition 없이 안정적인 서비스를 제공한다. 종료 시 업데이트된 stock.txt 파일의 내용이 정확히 반영하고 있음을 확인했다.

4. 성능 평가 결과 (Task 3)

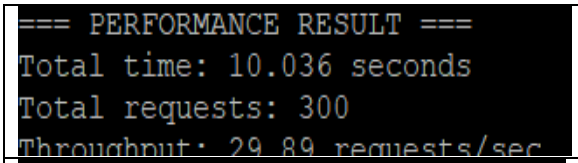
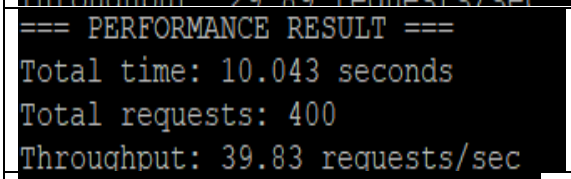
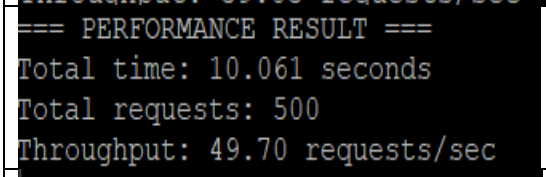
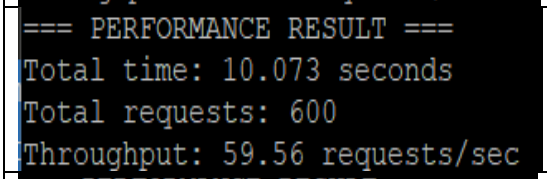
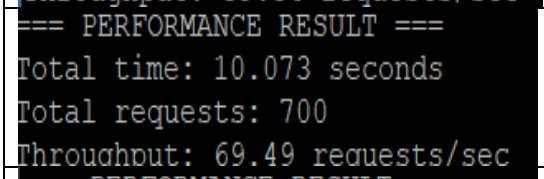
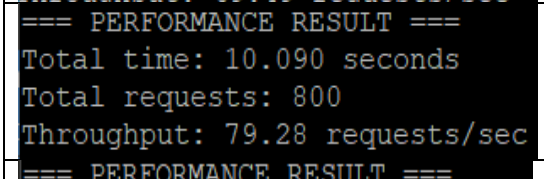
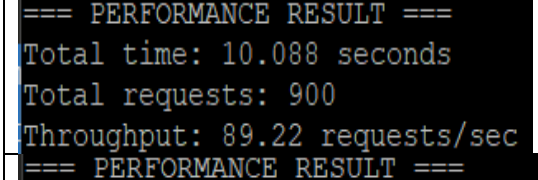
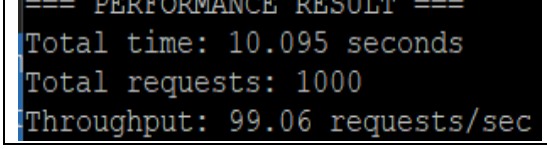
- **강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡쳐 포함)**
- **측정방법**

  측정 시작 시간은 첫 번째 클라이언트 프로세스 생성 직전이고 종료 시간은 마지막 클라이언트 프로세스 종료 직후이다. 이는 모든 클라이언트 프로세스 생성시간, 서버 연결 시간, 요청 및 응답 처리 시간, 프로세스 종료시간이 포함되어 있다. 서버 시작 및 종료 시간은 고려되지 않으며 실질적으로 사용자가 체감하는 성능이라고 볼 수 있다.

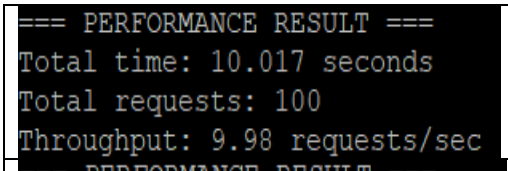
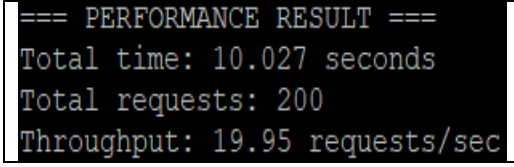
- **Task_1(even-driven approach)**
- 10에서 100까지 늘려가며 실험했다.

`=== PERFORMANCE RESULT ===`
`Total time: 10.020 seconds`
`Total requests: 100`
`Throughput: 9.98 requests/sec`
	Total Time: 10.020 sec
	Total Requests: 100
	Throughput: 9.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.026 seconds`
`Total requests: 200`
`Throughput: 19.95 requests/sec`
	Total Time: 10.026 sec
	Total Requests: 200
	Throughput: 19.95 req/sec

	Total Time: 10.036 sec Total Requests: 300 Throughput: 29.89 req/sec
	Total Time: 10.043 sec Total Requests: 400 Throughput: 39.83 req/sec
	Total Time: 10.061 sec Total Requests: 500 Throughput: 49.70 req/sec
	Total Time: 10.073 sec Total Requests: 600 Throughput: 59.56 req/sec
	Total Time: 10.073 sec Total Requests: 700 Throughput: 69.49 req/sec
	Total Time: 10.090 sec Total Requests: 800 Throughput: 79.28 req/sec
	Total Time: 10.088 sec Total Requests: 900 Throughput: 89.22 req/sec
	Total Time: 10.095 sec Total Requests: 1000 Throughput: 99.06 req/sec

- **Task_2(thread-based approach)**

	Total Time: 10.017 sec Total Requests: 100 Throughput: 9.98 req/sec
	Total Time: 10.027 sec Total Requests: 200 Throughput: 19.95 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 20.021 seconds`
`Total requests: 300`
`Throughput: 14.98 requests/sec`
	Total Time: 20.021 sec
	Total Requests: 300
	Throughput: 14.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 20.032 seconds`
`Total requests: 400`
`Throughput: 19.97 requests/sec`
	Total Time: 20.032 sec
	Total Requests: 400
	Throughput: 19.97 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 30.025 seconds`
`Total requests: 500`
`Throughput: 16.65 requests/sec`
	Total Time: 30.025 sec
	Total Requests: 500
	Throughput: 16.65 req/sec

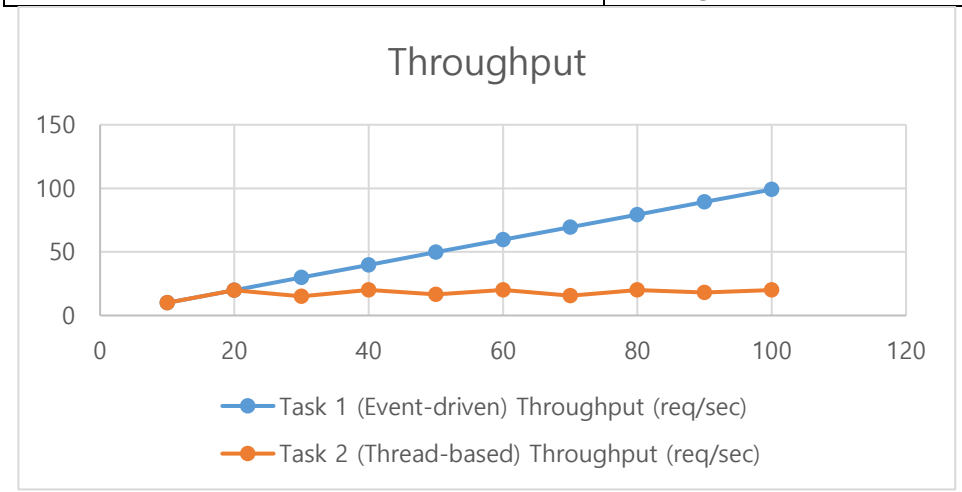
`=== PERFORMANCE RESULT ===`
`Total time: 30.039 seconds`
`Total requests: 600`
`Throughput: 19.97 requests/sec`
	Total Time: 30.039 sec
	Total Requests: 600
	Throughput: 19.97 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 45.034 seconds`
`Total requests: 700`
`Throughput: 15.54 requests/sec`
	Total Time: 45.034 sec
	Total Requests: 700
	Throughput: 15.54 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 40.046 seconds`
`Total requests: 800`
`Throughput: 19.98 requests/sec`
	Total Time: 40.046 sec
	Total Requests: 900
	Throughput: 19.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 50.035 seconds`
`Total requests: 900`
`Throughput: 17.99 requests/sec`
	Total Time: 50.035 sec
	Total Requests: 900
	Throughput: 17.99 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 50.049 seconds`
`Total requests: 1000`
`Throughput: 19.98 requests/sec`
	Total Time: 50.049 sec
	Total Requests: 1000
	Throughput: 19.98 req/sec



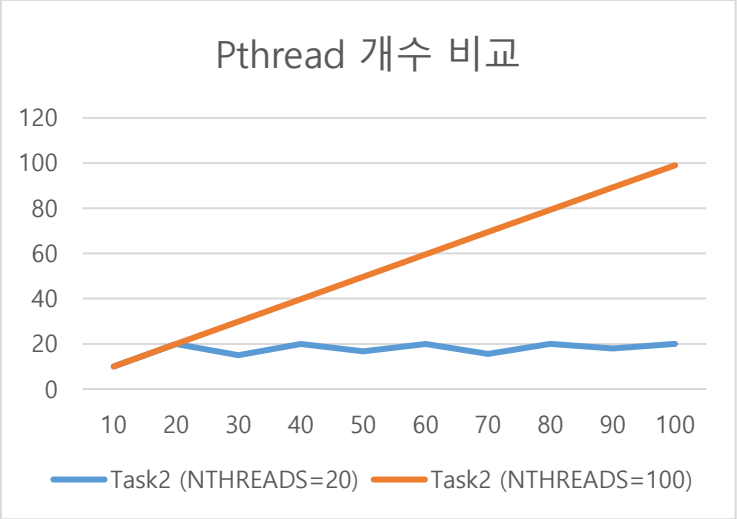
분석

Task1의 경우 선형 증가의 모습을 보이는데 I/O 대기 시간을 활용했기 때문에 클라이언트 수에 비례하여 처리량이 증가하며 client 개수가 증가해도 10초 이내에 완료되는 모습을 보인다.

그러나 task2의 경우 처리량이 제한되는 모습을 보이는데 이는 thread의 개수를 20개로 제한했기에 그런 것으로 보인다. NTHREADS=20으로 설정했기에 클라이언트 수가 Thread Pool 크기를 초과하면서 sbuf에서의 대기 시간이 급격히 증가하고 세마포어끼리의 경합 그리고 컨텍스트 스위칭이 일어나면서 병목현상이 일어나는 것으로 판단된다.

추가분석: Nthreads를 100개로 설정할 시

Clients	Task2 (NTHREADS=20)	Task2 (NTHREADS=100)
10	9.98	**9.98**
20	19.95	**19.94**
30	14.98	**29.89**
40	19.97	**39.84**
50	16.65	**49.73**
60	19.97	**59.62**
70	15.54	**69.5**
80	19.98	**79.35**
90	17.99	**89.18**
100	19.98	**99.01**



Thread-based 방식도 적절한 Thread Pool 크기를 설정하면 Event-driven과 동등한 성능을 보인다는 것을 알 수 있었다. 이후 비슷한 성능으로 측정하기 위해 thread based 방식의 thread pool 크기를 100으로 설정하고 계속 실험해보았다.

이어서

요청 타입에 따른 변화

Show-only의 경우(read-heavy)로 client의 개수를 10에서 50개씩 늘려가며 비교해보았다.

Task1

`=== PERFORMANCE RESULT ===`
`Total time: 10.019 seconds`
`Total requests: 100`
`Throughput: 9.98 requests/sec`
	Total Time: 10.019 sec
	Total Requests: 100
	Throughput: 9.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.030 seconds`
`Total requests: 200`
`Throughput: 19.94 requests/sec`
	Total Time: 10.030 sec
	Total Requests: 200
	Throughput: 19.94 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.041 seconds`
`Total requests: 300`
`Throughput: 29.88 requests/sec`
	Total Time: 10.041 sec
	Total Requests: 300
	Throughput: 29.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.048 seconds`
`Total requests: 400`
`Throughput: 39.81 requests/sec`
	Total Time: 10.048 sec
	Total Requests: 400
	Throughput: 39.81 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.066 seconds`
`Total requests: 500`
`Throughput: 49.67 requests/sec`
	Total Time: 10.066 sec
	Total Requests: 500
	Throughput: 49.67 req/sec

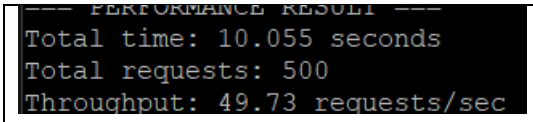
Task2

`=== PERFORMANCE RESULT ===`
`Total time: 10.019 seconds`
`Total requests: 100`
`Throughput: 9.98 requests/sec`
	Total Time: 10.019 sec
	Total Requests: 100
	Throughput: 9.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.029 seconds`
`Total requests: 200`
`Throughput: 19.94 requests/sec`
	Total Time: 10.029 sec
	Total Requests: 200
	Throughput: 19.94 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.038 seconds`
`Total requests: 300`
`Throughput: 29.89 requests/sec`
	Total Time: 10.038 sec
	Total Requests: 300
	Throughput: 29.89 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.049 seconds`
`Total requests: 400`
`Throughput: 39.80 requests/sec`
	Total Time: 10.049 sec
	Total Requests: 400
	Throughput: 39.80 req/sec

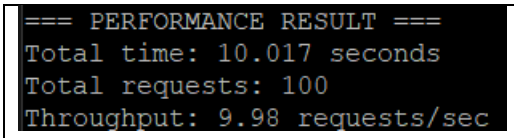
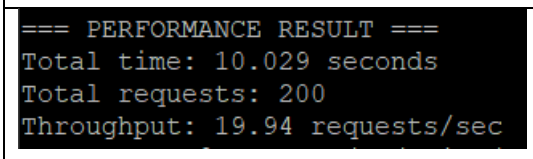
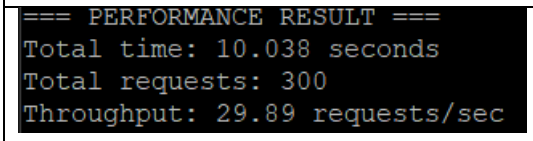
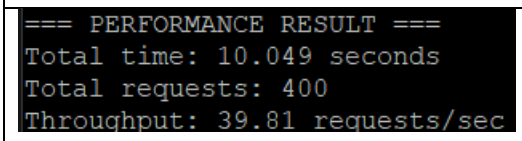
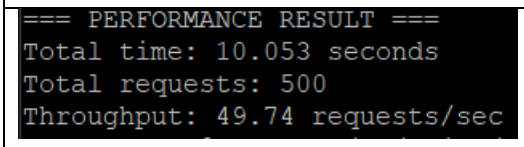
	Total Time: 10.055 sec Total Requests: 500 Throughput: 49.73 req/sec

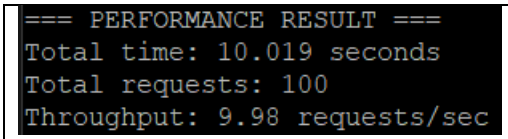
이론적으로는 task2가 병렬적으로 읽기 때문에 더 빨라야 하지만 실제로는 차이가 미미하다는 것을 발견할 수 있었다.

Clients	Task1 (Event)	Task2 (Thread)
10	9.98	9.98
20	19.94	19.94
30	29.98	29.89
40	39.81	39.80
50	49.67	49.73

Buy/sell-only의 경우(write-heavy)로 client의 개수를 10에서 50개씩 늘려가며 비교해보았다.

Task1

	Total Time: 10.017 sec Total Requests: 100 Throughput: 9.98 req/sec
	Total Time: 10.029 sec Total Requests: 200 Throughput: 19.94 req/sec
	Total Time: 10.038 sec Total Requests: 300 Throughput: 29.89 req/sec
	Total Time: 10.049 sec Total Requests: 400 Throughput: 39.81 req/sec
	Total Time: 10.053 sec Total Requests: 500 Throughput: 49.74 req/sec

	Total Time: 10.019 sec Total Requests: 100 Throughput: 9.98 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.026 seconds`
`Total requests: 200`
`Throughput: 19.95 requests/sec`
	Total Time: 10.026 sec
	Total Requests: 200
	Throughput: 19.95 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.034 seconds`
`Total requests: 300`
`Throughput: 29.90 requests/sec`
	Total Time: 10.034 sec
	Total Requests: 300
	Throughput: 29.90 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.042 seconds`
`Total requests: 400`
`Throughput: 39.83 requests/sec`
	Total Time: 10.042 sec
	Total Requests: 400
	Throughput: 39.83 req/sec

`=== PERFORMANCE RESULT ===`
`Total time: 10.054 seconds`
`Total requests: 500`
`Throughput: 49.73 requests/sec`
	Total Time: 10.054 sec
	Total Requests: 500
	Throughput: 49.73 req/sec

Task2에서 write_lock 경합으로 인해 task1에 비해 성능이 떨어질 것이라고 예상했지만, 실험자의 예상으로는 연산이 단순해서, client 개수가 너무 적어서 혹은 여러 이유로 거의 동일한 성능을 보이는 양상이다.

Clients	Task1 (Event)	Task2 (Thread)
10	9.98	9.98
20	19.94	19.95
30	29.89	29.90
40	39.81	39.83
50	49.74	49.73

Read와 write 둘 다 사용하는 기존 결과는 이미 앞에 했으니 생략하도록 하겠다.

핵심 결론

본 실험을 통해 얻은 가장 중요한 발견은 적절한 Thread Pool 크기를 설정해야 좋은 성능을 낼 수 있다는 점이다. NTHREADS를 20에서 100으로 증가시킨 결과, Event-driven 방식과 비슷한 성능을 낼 수 있었다. 이는 자원 할당 및 시스템 설계의 중요성을 보여준다고 생각한다.