System Programming Project 3

담당 교수 : 김영재

이름 : 차수환

학번 : 20222089

1. 개발 목표

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

본 프로젝트는 주식 거래 서버를 구현하여 여러 client들의 동시 접속 및 서비스를 효율적으로 처리하는 것을 목표로 한다.

서버는 주식 데이터를 이진 트리로 관리하며, 클리이언트의 show, buy, sell, exit 명령어를 처리한다.

이를 위한 Concurrentstock server를 select 기반의 Event-based 접근법과, pthread 기반의 Thread-based 접근법으로 구현하고 이 두 서버의 동시 처리율을 비교하여 처리 성능을 평가한다.

이를 통해 I/O 다중화와 병렬 처리의 특성을 이해하고, 시스템 부하에 따른 확장성을 분석한다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach
    - select()를 이용하여 단일 Thread 서버를 구현하여 다중 클라이언트의 요청을 비동기적으로 처리한다
    - I/O 다중화를 통해 여러 클라이언트 요청을 순차 처리한다
    - 주식 데이터는 이진 트리로 관리한다
2. Task 2: Thread-based Approach
    - 클라이언트마다 개별 Thread를 생성하여 병렬 처리한다
    - 세마포어로 공유 데이터 동기화를 보장하고, 같은 명령어를 안정적으로 처리한다

3. Task 3: Performance Evaluation

   - ■ 클라이언트 수와 명령어의 조합을 조정하며 처리율을 비교한다
   - ■ multiclient.c를 사용하여 다양한 시나리오에 대해 실험해본다

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

  select()는 단일 스레드로 여러 클라이언트 소켓을 모니터링한다. pool 구조체를 사용하여 각 클라이언트 소켓과 상태(read_set, ready_set)를 관리하며, 준비된 소켓으로부터 요청을 읽고 parse_request 함수로 처리한다.

  parse_request 함수는 전달받은 요청을 해석하고, 그에 맞게 주식 데이터를 처리한 뒤, 결과를 클라이언트에 다시 보낸다.

  ✓ epoll과의 차이점 서술

  epoll은 select와 유사한 기능을 하지만 더 큰 규모의 클라이언트 처리에 적합한 고성능 I/O 이벤터 처리 방식이다.

  select()는 매번 전체 FD 집합을 순회해야 하고, FD 수가 제한되어 확장성에 한계가 있다. 반면에, epoll은 이벤트가 발생한 소켓만 처리하며, 성능 및 확장성 측면에서 우수하다. 감시하고자 하는 FD를 커널 내부에 등록하고, 변화가 있는 Descriptor만 처리를 하는 식으로 작동한다.

  select은 모든 소켓을 순회하며 상태를 확인하므로 시간복잡도는 O(n) 이고, epoll은 준비된 소켓만 확인하므로 시간복잡도는 O(1)이다.

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

  ✓ Master Thread의 Connection 관리

  Master Thread는 클라이언트의 연결 요청을 감지하고, Open_listenfd 를 사용하여 서버 소켓을 열고, Accept로 새로운 클라이언트의 연결을 수락한다. 이후에 연결된 소켓은 동적으로 할당되고, Pthread_create로 Worker Thread에 전달된다. 각 Worker Thread는 클라이언트와 독립적으로 통신하며, Pthread_detach로 분리된 상태로 실행된다. client_thread는 요청을 읽고, parse_request로 그 요청을 처리하며, 세마포어(stock_sem)로 Stock *root 접근을 동기화하여 공유 주식 트리를 보호한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Worker Thread Pool 은 클라이언트의 요청을 처리하는 Worker Thread의 집합을 말하는데, 이번 프로젝트의 Task2에서 Thread 풀 방식은 구현하지 않았으며, Thread가 각 클라이언트 연결마다 Pthread_create로 새로 생성되도록 구현하였다.

  만약 Worker Thread Pool을 사용했다면, 고정된 Thread의 수를 사전에 정의하고, 그만큼의 Thread를 서버 시작 시 생성하고, 클라이언트를 공유 큐에 저장하여 Thread가 처리했을 것이다. 미리 생성된 Worker Thread들이 공유 큐에 클라이언트 요청이 들어올 때까지 대기하다가, 요청이 들어오면 하나의 Thread가 해당 작업을 꺼내 처리했을 것이다. 이후에 작업이 끝나면 다시 공유 큐에 명령이 들어올 때까지 대기 상태로 돌아갔을 것이다.

  하지만 클아이언트의 요청이 고정된 Thread의 수를 초과했을 경우에, 큐에서 대기해야 하며 지연이 일어날 수 있다. 따라서 유연성이 높은 동적 Thread 생성 방식을 선택하였다.

- **Task3 (Performance Evaluation)**

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

  각 서버 환경마다 총 요청 수를 경과 시간으로 나눈 동시 처리율을 측정하여 비교한다.

  concurrent한 서버의 성능을 비교하는 것이므로 초당 몇 개의 명령을 처리할 수 있을지를 평가하기 위해 동시 처리율을 측정하기로 하였다.

각 실험 환경에서 gettimeofday()를 사용하여 테스트 시간과 요청 수를 기록하여 계산하도록 하였다.

✓ Configuration 변화에 따른 예상 결과 서술

클라이언트 수와 명령어의 조합을 조정하며 처리율을 비교한다

multiclient.c를 사용하여 실험 시나리오를 모든 명령이 가능한 경우, show 명령만 가능한 경우, buy/sell 명령만 가능한 경우, 주식이 1개만 있고 모든 명령이 가능한 경우로 나눴으며 각 시나리오에 대해 클라이언트 수가 10, 20, 30, 50, 100, 300개 일 경우로 조정하여 실험하였다.

클라이언트 수가 적을 때는 Task1과 Task2의 성능 차이가 작지만, 클라이언트 수가 많아질수록 Task2의 병렬 처리 효과가 커지며 더 높은 처리율을 기록할 것으로 예상된다.

C. 개발 방법

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

주식 데이터는 ID를 기준으로 이진 트리에 저장하며, Stock 구조체를 사용하여 ID, 수량, 가격, 좌우 자식 노드를 관리한다. 이를 위해 make_stock, find_stock, insert_stock, free_stock, load_stocks 함수를 구현했다.

Task 1~2에서 공통으로 선언한 함수는 다음과 같다.

- 공통 함수

✓ make_stock: id, quantity, price를 인자로 받아 새로운 Stock 구조체를 만들고, Malloc으로 메모리를 할당한다. load_stocks에서 호출한다.

✓ find_stock: id를 인자로 받아 이진 트리에서 해당 ID의 Stock을 검색한다. buy_stock과 sell_stock에서 사용한다.

✓ insert_stock: root와 새 Stock 노드를 인자로 받아서 이진 트리에 삽입한다. 재귀적으로 노드를 순회하며 id가 작으면 왼쪽, 크면 오른쪽으로 이동하고 빈

자리에 새 노드를 저장한다. 만약 ID가 같다면, quantity와 price를 업데이트 한다. load_stocks에서 사용된다.

- ✓ free_stock: 이진 트리 노드를 재귀적으로 free한다. 좌우 자식을 먼저 해제하고 현재 노드를 Free로 해제한다.
- ✓ load_stocks: "stock.txt" 파일을 열고 줄 단위로 읽으며, id, quantity, price 순으로 파싱한다. 이후에 make_stock으로 노드를 생성하고 insert_stock으로 이진 트리를 구성하낟. 파일을 다 읽고 나서는 Fclose로 닫고 root를 리턴한다.
- ✓ print_stocks: 이진 트리를 순회하며 Stock의 정보를 문자열 buf에 추가한다.
- ✓ buy_stock: id와 구매 수량을 입력으로 받아, find_stock으로 Stock 정보를 검색한다. 재고가 충분하다면 구매 수량만큼 감소시키고 1을 반환하고, 충분하지 않다면 0을 반환하낟. parse_request의 buy에서 호출한다.
- ✓ save_stocks: "stock.txt"를 쓰기 모드로 열고 print_stocks로 주식의 모든 정보들을 문자열로 생성하고, Fputs로 파일에 저장하고, Fclose로 닫는다.

- Task1: pool구조체 및 select 기반 루프로 서버를 구현하였고, 다중 클라이언트 요청을 단일 Thread에서 순차적으로 처리한다. 하지만 동기화가 없어 데이터 경합이 발생할 수 있다. 주요 함수와 구조체는 아래와 같다.
  - ✓ pool: 클라이언트의 소켓과 I/O상태를 관리하는 구조체이다. maxfd, read_set, ready_set, clientfd[], clientrio[] 등을 포함한다.
  - ✓ init_pool: pool 구조체를 초기화한다.
  - ✓ add_client: 새 클라이언트를 pool에 추가하고, clinetfd[]에서 -1로 설정된 부분(-1인 init_pool로 초기화된 부분으로, 빈 슬롯을 나타냄)을 찾아 새 클라이언트를 저장한다. 이후에 Rio_readinitb로 I/O를 초기화한다.
  - ✓ check_clients: select()로 감지된 소켓에 대해 클라이언트의 요청을 읽고, parse_request로 명령을 처리한다. 연결이 끝난 뒤에 소켓을 닫고 슬롯을 비운다.
  - ✓ parse_request: 명령어를 파싱하고, 클라이언트의 요청을 처리한다.
  - ✓ sigint_handler: SIGINT 신호가 발생하면 주식 정보를 저장하고, 메모리를 모두 해제하고 서버를 종료한다.

- Task2: 동적 Thread 기반 서버로, 클라이언트별 병렬 처리를 지원하며, 주요 함수는 아래와 같다.
  - ✓ client_thread: 클라이언트 연결마다 생성된 Thread가 요청을 읽고, parse_request로 처리한 뒤, 종료 시 연결을 닫고 Pthread_detach로 결과를 회수한다.
  - ✓ parse_request: Task1과 동일하며, 공유 자원의 보호를 위해 세마포어를 사용해, 이진 트리의 접근을 동기화했다.
  - ✓ sigint_handler: 서버 종료 시 주식들의 정보를 저장하고 이진 트리를 모두 해제한다.

- Task3: multiclient.c를 수정하여 성능 평가 코드를 구현하고, 처리율을 측정하였다. 다양한 워크로드와 클라이언트 수로 실험하며, 추가한 부분은 다음과 같다.
  - ✓ gettimeofday로 시간 측정 후 처리율 계산
  - ✓ 워크로드는 option = rand() % 3 에서 "rand() % 3"부분을 수정하면서 조정
  - ✓ 클라이언트 수는 ./multiclinet 시에 조정

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**
- Task1은 select 기반 Event-driven 서버를 구현하여 다중 클라이언트 요청을 단일 스레드에서 비동기적으로 처리했다. pool 구조체로 클라이언트 소켓을 관리하며, stock.txt에서 읽은 BST를 기반으로 show, buy, sell, exit 명령을 처리한다. 하지만 공유 자원 보호 미구현으로 race condition 가능성 존재한다.
- Task2는 각 클라이언트에 대해 Thread를 생성하는 방식으로, 각 클라이언트 연결에 Pthread_create로 스레드를 할당하여 병렬 처리했다. 세마포어로 동기화를 구현하고, 요청 효율적으로 처리했다.
- 서버는 클라이언트 요청에 따라 BST를 갱신하고, 결과를 반환한 뒤 save_stocks로 stock.txt를 업데이트한다. stockclient 테스트에서 show, buy, sell 명령이 안정적으로 작동했으며, parse_request의 strtok으로 명령어 파싱을 최적화했다. multiclient 테스트에서 Task1은 소규모 클라이언트에서 효율적이었고, Task2는 병렬 처리로 쓰기 요청에서 우수한 성능을 보였다.
- 미구현 부분: Task2에서 Worker Thread Pool을 구현하지 않았는데, 만약 구현했다면, 미리 Thread를 생성하고 클라이언트들의 요청을 저장하는 Queue를 선언하여, Thread가 Queue안의 요청들을 처리하도록 구현하였을 것이다.

4. 성능 평가 결과 (Task 3)

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

성능 평가를 위한 실험은 multiclinet.c에서 랜덤하게 생성되는 명령어의 경우를 다음 네 가지 Case로 나눠서 진행하였다.

모든 Case에서 Client의 수는 10, 20, 30, 50, 100, 300개로 수행하였다.

1. 모든 명령(show, buy, sell)이 가능할 때

(1)Task1:

```
Elapsed time: 10.02 seconds
Total requests: 100
Throughput: 9.98 requests/second
```

```
Elapsed time: 10.03 seconds
Total requests: 200
Throughput: 19.95 requests/second
```

```
Elapsed time: 10.04 seconds
Total requests: 300
Throughput: 29.88 requests/second
```

```
Elapsed time: 10.05 seconds
Total requests: 500
Throughput: 49.74 requests/second
```

```
Elapsed time: 10.12 seconds
Total requests: 1000
Throughput: 98.86 requests/second
```

```
Elapsed time: 11.17 seconds
Total requests: 3000
Throughput: 268.53 requests/second
```

(2)Task2:

```
Elapsed time: 10.02 seconds
Total requests: 100
Throughput: 9.98 requests/second
```

```
Elapsed time: 10.03 seconds
Total requests: 200
Throughput: 19.94 requests/second
```

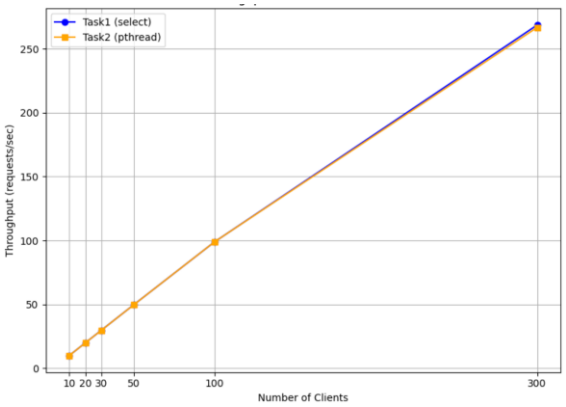
```
Elapsed time: 10.04 seconds
Total requests: 300
Throughput: 29.87 requests/second
```

```
Elapsed time: 10.06 seconds
Total requests: 500
Throughput: 49.70 requests/second
```

```
Elapsed time: 10.11 seconds
Total requests: 1000
Throughput: 98.89 requests/second
```

```
Elapsed time: 11.25 seconds
Total requests: 3000
Throughput: 266.64 requests/second
```

(3)결과:



2. show 명령만 가능할 때

(1)Task1:

```
Elapsed time: 10.02 seconds
Total requests: 100
Throughput: 9.98 requests/second
```

```
Elapsed time: 10.03 seconds
Total requests: 200
Throughput: 19.95 requests/second
```

```
Elapsed time: 10.04 seconds
Total requests: 300
Throughput: 29.88 requests/second
```

```
Elapsed time: 10.06 seconds
Total requests: 500
Throughput: 49.69 requests/second
```

```
Elapsed time: 10.11 seconds
Total requests: 1000
Throughput: 98.89 requests/second
```

```
Elapsed time: 11.18 seconds
Total requests: 3000
Throughput: 268.40 requests/second
```

(2)Task2:

```
Elapsed time: 10.02 seconds
Total requests: 100
Throughput: 9.98 requests/second
```

```
Elapsed time: 10.03 seconds
Total requests: 200
Throughput: 19.95 requests/second
```

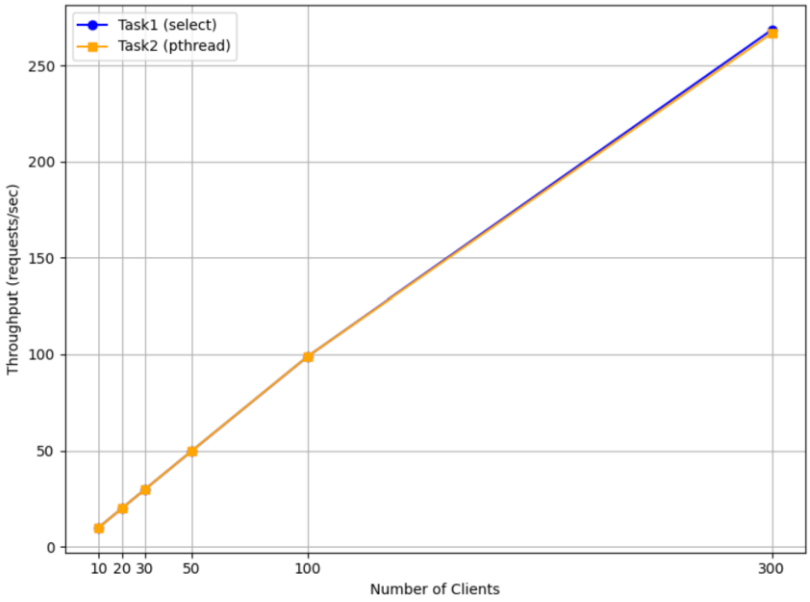
```
Elapsed time: 10.04 seconds
Total requests: 300
Throughput: 29.88 requests/second
```

```
Elapsed time: 10.06 seconds
Total requests: 500
Throughput: 49.70 requests/second
```

```
Elapsed time: 10.11 seconds
Total requests: 1000
Throughput: 98.92 requests/second
```

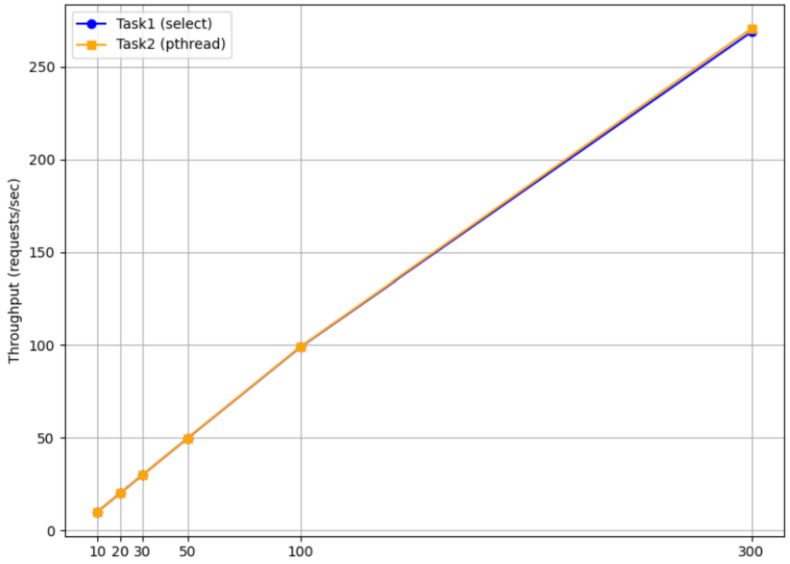
```
Elapsed time: 11.26 seconds
Total requests: 3000
Throughput: 266.55 requests/second
```

(3)결과:



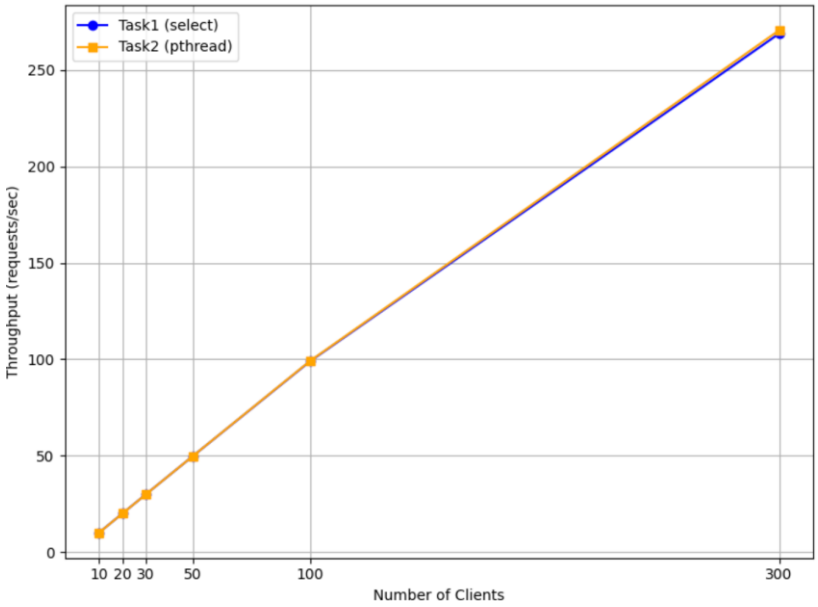
3. buy/ sell 명령만 가능할 때

결과 그래프:



4. 주식이 1개만 존재하고, 모든 명령이 가능할 때

결과 그래프:



5. 결론

모든 Case에서 처리율은 클라이언트 수에 비례하여 증가했으며, Case 간 유의미한 차이는 없었다. 이는 이진 트리 기반의 Stock 구조와 명령어 처리 논리가 효율적으로 구성되어 있어, 다양한 워크로드에서도 일관된 성능을 유지했기 때문이다. Task1은 select 방식으로 읽기 요청에 안정적이었고, Task2는 동적 스레드 병렬 처리로 큰 차이는 아니지만 쓰기 요청에서 소폭 우세했다.

하지만 Client의 개수가 비정상적으로 많아질 때에는(500개 이상) Server가 클라이언트의 요청을 제대로 받지 못하거나, 처리율이 오히려 떨어지는 것도 관찰할 수 있었다.

이는 공유 자원에 대한 경쟁이 심화되며 처리율 증가폭이 둔화되는 것으로 보인다. 특히 Task2의 경우, 세마포어를 통한 동기화로, 병렬 처리의 이점이 상쇄되기도 한다.

또한, Task1은 select()의 구조적 한계로 인해 FD 수의 제한에 가까워질수록 성능 저하 가능성이 존재하며, 대규모 서버 환경에서는 epoll과 같은 확장성이 높은 방식으로의 전환이 필요할 수 있을 것이다.