System Programming Project 3

담당 교수 : 김영재

이름 : 김효림

학번 : 20221549

1. 개발 목표

Cocurrent한 주식 서버를 구현하여 여러 클라이언트가 동시에 접속할 수 있는 주식장을 구현한다. 클라이언트는 서버가 열리면 show 명령어를 통해 stock table을 확인할 수 있고, buy 명령어를 통해 주식을 구매할 수 있으며, sell 명령어를 통해 주식을 팔 수 있다. Exit 명령어를 통해 서버를 닫을 수 있으며, client 접속은 Ctrl + C 버튼을 통해 종료한다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   Select 함수를 활용하여 multi-client의 요청을 수행한다. 각 클라이언트는 fd를 깨우고, select에 의해 동작을 시작한다. 활성화된 클라이언트의 집합은 pool에 저장되어 있다. 메인 함수 내에서 무한 루프를 돌며 select를 반복적으로 호출하고 이벤트를 감지하여 클라이언트의 명령을 수행한다. multiclient의 수행의 경우 Ctrl C를 통해 signal handler를 통하여 stock.txt에 결과를 저장할 수 있다.

2. Task 2: Thread-based Approach

   Accept, pthread를 활용하여 concurrent한 주식 서버가 돌아간다. 메인 스레드로부터 부가적인 스레드가 생성이 되며, 1과 마찬가지로 무한루프 내에서 request를 승인하고 sbuf에 fd를 삽입하게 된다. Process_request함수로부터 명령어에 해당하는 일을 수행한 후 그 결과를 반환하게 된다. multiclient의 수행의 경우 Ctrl C를 통해 signal handler를 통하여 stock.txt에 결과를 저장할 수 있다.

3. Task 3: Performance Evaluation

   Configuration에 따른 elapsed time을 확인하여 task1, task2에 대한 동시 처리율 변화를 분석하며 성능을 평가하고 분석할 수 있다.

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

    이벤트 기반의 concurrent system은 다수의 클라이언트 요청을 비동기적으로 처리한다. 이때 I/O Multiplexing을 사용하는데, select() 함수를 이용해 여러 파일 디스크립터를 모니터링하게 된다. 소캣이 초기화 된 후 파일 디스크립터를 생성하여 클라이언트의 요청을 처리하게 된다.

  - ✓ epoll과의 차이점 서술

    select와 epoll 함수 모두 I/O Multiplexing을 처리하기 위한 함수지만, epoll은 select에 비해 대규모 소켓에 대한 처리를 할 수 있는 고성능 인터페이스이다. Select는 모니터링할 fd set을 매번 재설정하지만, epoll은 변경된 fd만 갱신할 뿐 매번 재설정하지 않는다. 즉, select는 등록할 때마다 전체 fd 목록을 os에 전달하지만, epoll은 변경된 fd만 선별적으로 등록, 삭제 및 수정할 수 있다. 따라서 대규모의 파일 디스크립터를 사용하는 동작에는 epoll이 더 적합하다.

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

    마스터 스레드는 클라이언트의 연결을 수신하고 처리할 worker thread에 할당한다. 소켓을 초기화하고, 클라이언트 연결이 들어오면 파일 디스크립터를 생성하여 작업 큐에 넣어, worker thread가 이를 처리하도록 한다.

  - ✓ Worker Thread Pool 관리하는 부분에 대해 서술

    작업 큐에서 큐잉된 클라이언트의 요청을 꺼내 처리한다. 서버가 시작될 때 생성된 여러 개의 워커 스레드는 작업 큐에서 새로운 작업이 들어오기를 대기하며, 작업이 들어왔을 대 클라이언트의 요청을 처리하여 결과물을 생성한다. 이때 세마포어를 사용하여 스레드를 동기화하여 작업 큐에 대한 접근을 안전하게 처리할 수 있도록 한다.

- **Task3 (Performance Evaluation)**

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

확장성 : 얼마나 많은 유저를 서비스할 수 있는가 판단하기 위해 동시 처리율을 측정한다. 같은 명령 수를 configuration을 통해 제한 한 후 이용자가 늘어남에 따라 처리 시간을 확인하여 판단한다. 동시 처리율은 높을수록 서버가 더 많은 요청을 효율적으로 처리할 수 있음을 나타내 주는 지표이기 때문에 concurrent system이 얼마나 효율적인지를 확인할 수 있어 정하였다.

워크로드에 따른 응답 시간 확인 : 클라이언트의 요청에 대한 서버의 응답시간을 확인한다. 요청에 대한 서버의 응답 속도를 평가하기 위해 정하였다. 각 클라이언트 요청에 따라 시스템이 종료될 때까지 걸리는 시간을 평가한다.

✓ Configuration 변화에 따른 예상 결과 서술

Client 수가 많아질수록, 각 client의 명령어 수가 증가할수록 server의 명령어 처리 시간이 증가할 것이다. 또한 client의 명령어 수가 증가할수록 서버의 부하가 증가하여 응답 시간이 증가될 것이다. 또한 단일 thread를 바탕으로 하는 event 기반의 동시 처리가 더 빠르게 진행될 것이다. Multi thread를 통해 concurrent system을 구축하게 되면 mutex를 통한 semaphore의 사용으로 각 명령어 마다 대기하는 시간이 있기 때문에, event driven의 stock server가 더 좋은 성능을 보일 것 같다.

C. 개발 방법

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

- Task 1

  Item 이름으로 binary tree를 사용하기 위한 구조체를 선언하였다. 구조체 내부에는 stock ID, 남아있는 주식의 개수, 주식의 가격, 연결되어 있는 왼쪽 노드와 오른쪽 노드가 저장되어 있다. 그 후 pool이라는 이름으로 연결된 디스크립터들의 pool을 저장하기 위한 구조체를 선언하였다. 구조체 내부에는 최대 디스크립터의 수, 활성화된 디스크립터와 읽기 위한 디스크립터의 부분집합을 나타내는 변수, 준비된 디스크립터의 수를 나타내는 변수, 클라이언트 집합에 들어갈 인덱스와 active한 디스크립터의 집합인 client_fd, active read buffer의 집합인 client_rio가 있다. Item* insert 함수를 통해 새로운 노드를 삽입하고, save_tree는 파일에 주식에 대한 정보(binary tree)를 저장하는 함수이다. Item* Search 함수를 통해 해당하는 stock ID를 찾을 수 있고, save_stock_table은 save_tree 함수를 통해 최종적으로 stock.txt에 최종 주식 정보를 저장하게 된다. 이때 handle_sigint 및 SIGINT signal을 사용하여 시그널을 받았을 때 save_stock_table이 실행될 수 있도록 하였다. Process_request는 입력받은 명령어(show, buy, sell, exit)에 따라 기능을 수행할 수 있도록 구현한 함수이다. Server는 ctrl+C를 통해 종료할 수 있다.

- Task 2

  sbuf_t라는 이름으로 thread를 관리할 구조체를 선언하였다. Sbuf_init 함수는 sbuf 구조체를 initialzie하며, deinit은 해제, insert는 thread를 추가하고 sbuf_remove는 thread를 삭제한다. 또한 thread는 생성된 pthread가 동작할 함수를 의미하며, 내부적으로 process_request가 함수로 선언되어 client의 각 명령에 대한 기능을 수행한다. Item 변수는 task1 때 가졌던 변수 외에도 semaphore를 위한 mutex 변수, write를 활성화 하기 위한 sem_t w 변수를 추가하여 concurrent system이 구현될 수 있도록 변수를 선언하였다. Item* insert, void save_tree, Item* Search, void Release, void save_stock_table 함수는 task1과 동일한 기능을 수행한다. 외에도 적절히 semaphore가 작용할 수 있도록 Release 함수를 추가하여 관리하였다. Server는 Ctrl+C를 통해 종료할 수 있다.

- Task 3

Elapsed time을 확인하기 위하여 multiclient에서 clock_t start와 end 변수를 설정하였다. 시작할 때 start = clock();을 사용하였고 종료 시에 end = clock() – start; 를 통하여 프로그램이 종료될 때까지 걸리는 시간을 측정하였다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**
- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**
- **Task1**

  Multiclient가 명령어를 전달하면 그에 맞는 기능들이 동작한다. Show는 현재 주식 시장에 대한 상태를 보여주고, buy 명령어를 통해서는 주식을 구매할 수 있다. Sell 명령어를 통해 주식을 팔고, exit 명령어를 통해 포트(주식장)를 종료시킬 수 있다. Server를 종료 시키기 위해서는 Ctrl+C를 입력하면 된다. Stock.txt에는 save_stock_table() 함수를 통해 수행 결과가 저장된다.

- **Task2**

  Task1과 동일하게 작동한다. 명령어를 전달하면 그에 해당하는 기능들이 작동하여 server에서 client에 명령어에 해당하는 응답을 전송한다.

- **Task3**

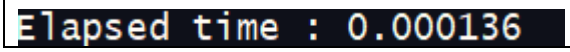
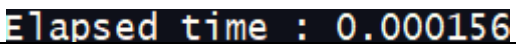
  생성된 모든 child thread의 명령어를 받아 server 내부적으로 stock.txt를 수정한 후(process_request를 실행한 후) elapsed time을 출력한다.

4. 성능 평가 결과 (Task 3)

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

1. 확장성 – 각 방법에 대한 client 개수 변화에 따른 동시 처리율

#child : 1, order per client : 10

Select	Thread
	

#client : 5, order per client : 10

Select	Thread
Elapsed time : 0.000557	Elapsed time : 0.000533

#client : 10, order per client : 10

Select	Thread
Elapsed time : 0.001024	Elapsed time : 0.000949

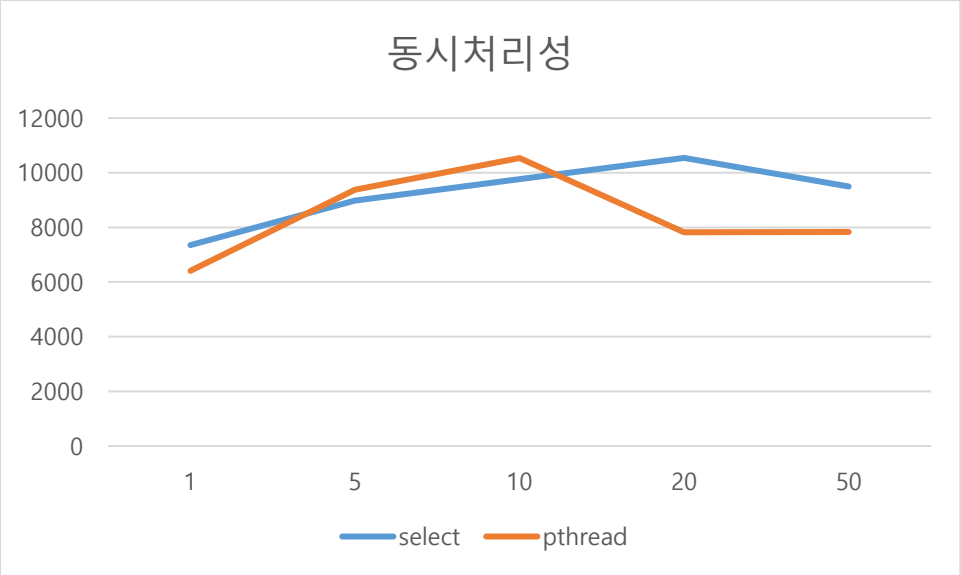
#client : 20, order per client : 10

Select	Thread
Elapsed time : 0.001897	Elapsed time : 0.002559

#client : 50, order per client : 10

Select	Thread
Elapsed time : 0.005265	Elapsed time : 0.006384

이를 통해 동시처리율은 client 수에 order의 수를 곱한 후 시간을 나누어 구하였다. 단위는 10^4로 하였다. 동시처리율은 특정 지점까지는 증가하다, 그 수를 넘어선 client가 존재하는 경우 줄어들었다.



	1	5	10	20	50
select	7352	8976	9765	10542	9496
pthread	6410	9380	10537	7815	7832

client 수가 1명일 때는 event 기반의 server의 효율이 더 좋았는데, 단일 스레드를 사용해 오버헤드가 적기 때문이 이벤트 기반 방식이 효율적으로 이루어진 것 같다. Client 수가 5일 때 pthread가 event driven server의 효율을 앞질렀는데, 병렬 처리가 잘 수행되며 앞지른 것 같다. 마찬가지로 client 10에서도 같은 결과가 나타났다. Client 수가 20명 이상이 되며 상황은 반전됐는데, thread 방식은 client 수에 비례하는 thread를 관리하며 오버헤드가 발생한 거 같다. 이를 통해 thread 방식은 client 수가 증가할수록 많은 오버헤드가 발생한다는 걸 확인할 수 있으며, event driven 방식과 달리 context switch 비용 또한 영향을 미침을 확인할 수 있다.

2. 워크로드에 따른 분석 – client 요청 타입에 따른 동시 처리율 변화 분석(child : 10, order per client :10)

1) 모든 client가 buy 또는 sell을 요청하는 경우

Select	Thread
Elapsed time : 0.001100	Elapsed time : 0.001054

2) 모든 client가 show만 요청하는 경우

Select	Thread
Elapsed time : 0.001224	Elapsed time : 0.001123

3) 모든 client가 buy 또는 show를 요청하는 경우

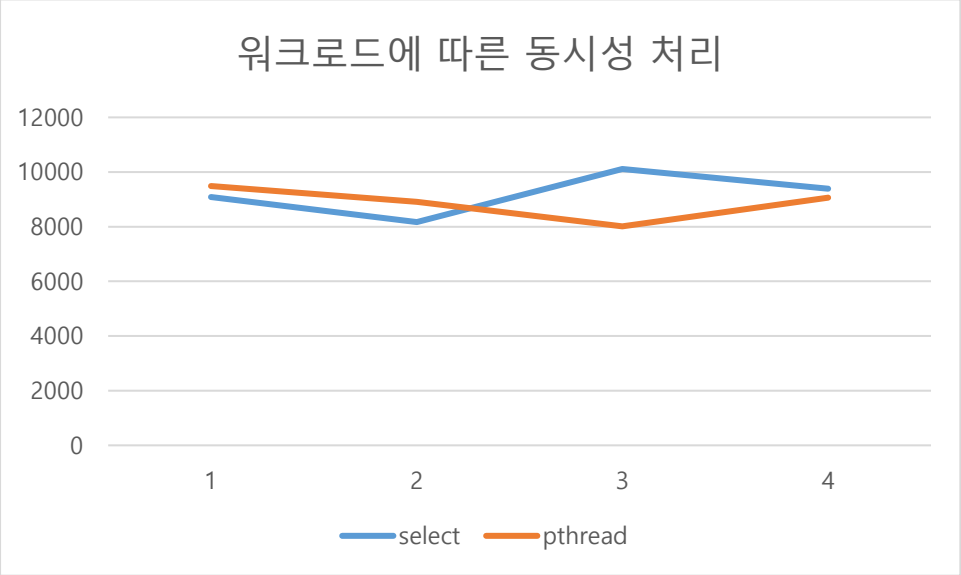
Select	Thread
Elapsed time : 0.000989	Elapsed time : 0.001248

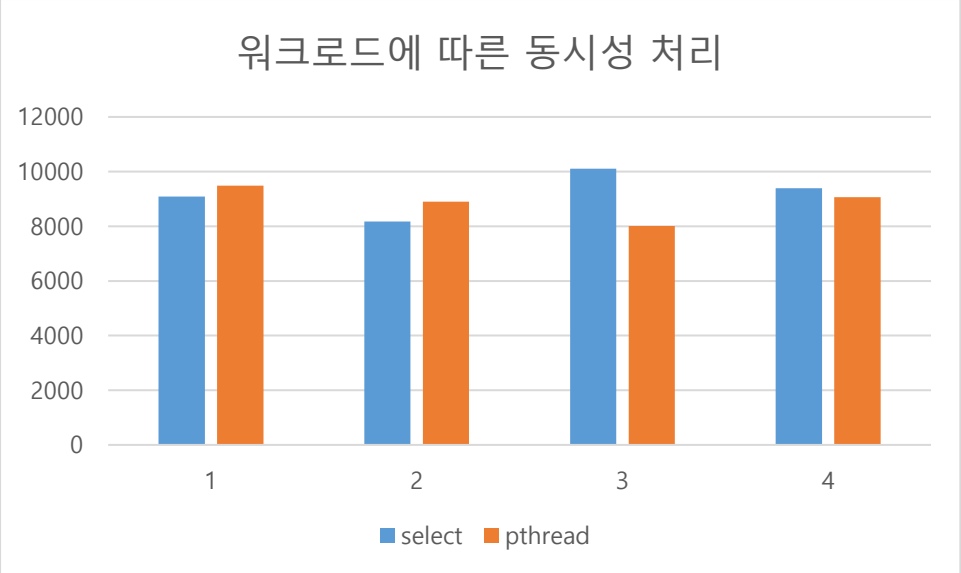
4) 모든 client가 sell 또는 show를 요청하는 경우

Select	Thread
Elapsed time : 0.001065	Elapsed time : 0.001103

Client의 요청에 따른 동시 처리율 역시 같은 방식으로 구하였다. client 수에 order의 수를 곱한 후 시간을 나누었고, 단위는 10^4로 하였다.

	1	2	3	4
select	9090	8169	10111	9389
pthread	9487	8904	8012	9066





그래프를 통해 확인할 수 있듯이, 1번(buy or sell)과 2번(show만) 요청에 대해서는 pthread의 동시 처리율이 좋았으나 3번(buy or show)과 4번(sell or show)에 대해서는 select의 동시처리율이 더 좋았음을 확인할 수 있다.

Buy와 sell 요청은 데이터의 수정이 필요하고, show의 요청은 data의 수정이 필요없이 binary search tree로 구현된 데이터를 순회하며 출력한다. 따라서 앞에 두 명령의 경우 동기화 오버헤드를 증가시키며, show의 경우에는 동기화 비용이 적게 발생한다. Show의

경우 thread에서는 독립적으로 읽기 작업을 병렬적으로 처리하며 더 좋은 성능을 보였다고 생각한다. Show와 buy 처럼 두 가지 경우가 혼합된 경우에서는 select가 더 높은 효율성을 보였는데, 이는 동기화 오버헤드를 증가시키는 명령에 대해 단일 스레드를 사용하는 event-driven server가 더 좋은 효과를 보인 것이라고 생각한다. 워크로드 1번이나 4번에 대해 pthead와 event는 유사한 성능을 보이는데, 이는 동기화 오버헤드와 병렬처리의 이점이 비슷한 수준으로 발생하여 그런 것 같다. Pthread는 독립적으로 요청을 실행하고 수행하며 병렬처리를 진행하는데 반해, select는 단일 스레드에서 모든 요청을 순차적으로 처리한다. 이때 select에서 발생한 병목현상이 pthread의 병렬처리의 이점을 능가하지 못해 저런 현상이 나타났다고 생각한다.

이를 통해 동기화 오버헤드가 발생하더라도, 병렬 처리를 하는 경우에 그 이점이 오버헤드를 상쇄하여 thread 기반의 concurrent system 역시 좋은 성능을 가질 수 있다는 걸 알게 되었다.