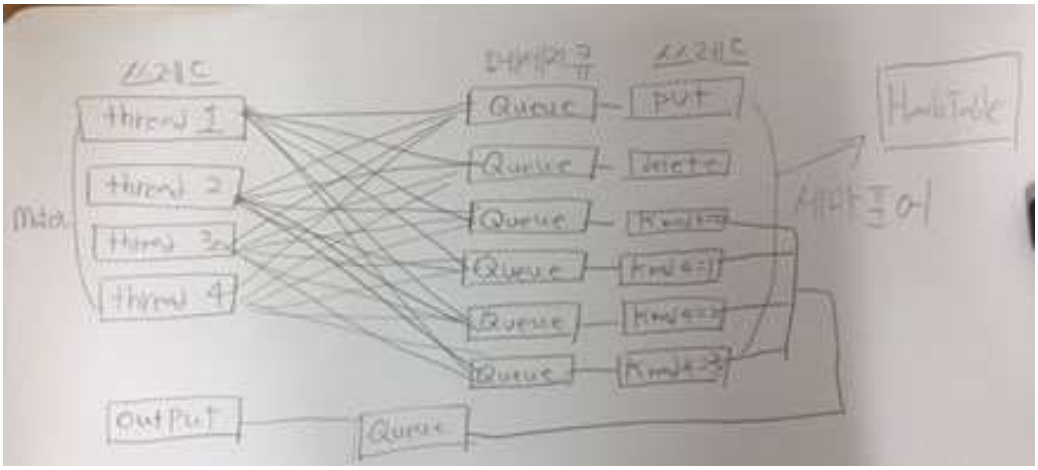
운영체제 PBL 개인보고서

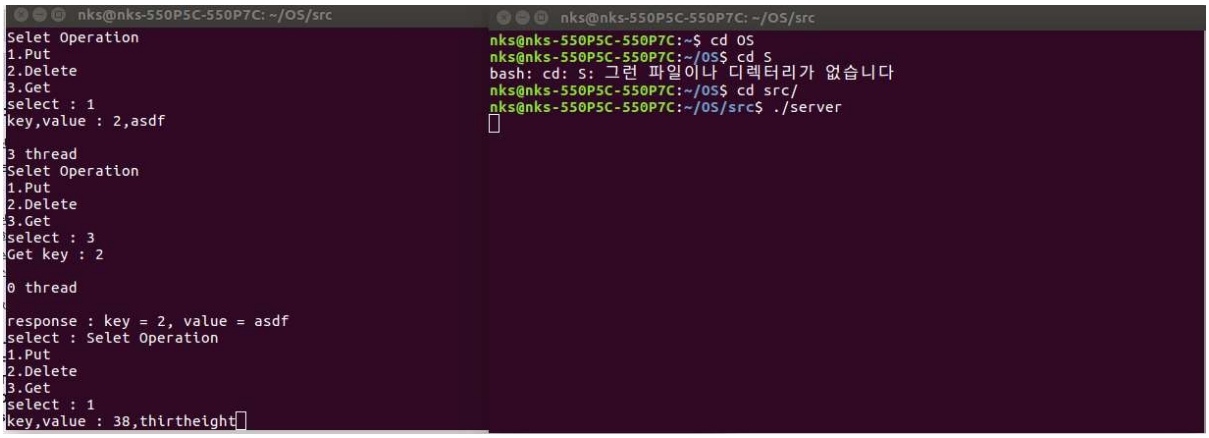
2013042776

컴퓨터 공학과

남궁 선

1. 동작구조 및 구현, 테스트





1) 스레드

클라이언트에서 서버로의 데이터 전송에는 4개의 스레드를 사용하였으며 각각의 스레드가 서버의 스레드들과 전부 연동되 있는 형태이다. 각각의 스레드들이 콘솔에서 명령을 받아 해당명령의 서버의 스레드들과 연결된 메시지 큐로 데이터를 입력한다. 4개를 사용한 이유는 현재 사용하는 컴퓨터의 코어 수와 동일하게 맞추기 위해 4개로 지정하였다. 이 스레드 들은 입력 받은 key값을 1차적으로 hash를 판단해 알맞은 서버의 스레드로 전송하여 준다.

또한 4개의 전송 스레드와 별개로 하나의 서버->클라이언트 전용 OUTPUT 스레드를 구축하였다. 왜냐하면 클라이언트의 입력은 프론트에서 직접적으로 유저와 이루어 지기 때문에 빠른 전환을 통해 많은 클라이언트를 수용해야 한다고 판단하였다. 그래서 전송 스레드들은 입력만 받고 바로바로 전환이 이루어지게 구조를 설계하였다. OUTPUT스레드는 서버로부터 전송 받는 데이터를 출력해주는 역할을 한다. 스레드를 하나만 둔 이유는 클라이언트의 개수가 4개밖에 되지 않으며, OUTPUT스레드가 많을수록 클라이언트의 전환속도가 느려진다. 또한 OUTPUT스레드의 작업은 단지 메시지큐로부터 데이터를 읽어와 출력해주는 역할만 있기 때문에 스레드 1개로도 충분하다 판단되었다.

서버의 스레드는 각자의 업무 분담량에 기반하여 설계하였다. PUT스레드 같은 경우는 put명령을 처리한다. 시간이 지나고 put은 5%밖에 발생하지 않기 때문에 많은 스레드를 할당할 필요가 없다 판단하였다. Delete 스레드는 다른 작업에 비해 현저히 적은 발생률을 보이며 또한 delete는 싱크문제, 에러문제 등 많은 버그가 발생할 수 있는 치명적인 작업이기 때문에 다중 스레드보다는 싱글스레드로 관리하는 것이 수월하다 판단하였다.

Get을 담당하는 스레드는 해시테이블 구조와 업무 분담량을 고려하여 설계하였다. Hash Function이 기본적으로 K mod 4에 기반을 두었고, 클라이언트의 스레드 수가 4개이기 때문에 4개로 설정하였다. 각각의 스레드들은 전송 받은 데이터를 Hash Function을 사용해 hash테이블에서 데이터를 가져온다. 그리고 이들은 일괄적으로 클라이언트의 OUTPUT 메시지큐에 입력을 하여 output스레드에게 데이터를 전송한다.

2)Hash Table

Char * hashTable[4][11] 로 설게 하였다. Key & 4 -> Row, Key /4 -> Column 이다. 해당 인덱스에 데이터를 저장한다.

3)CPU Pinning

스레드들의 CPU pinning은 사용하지 않았다. 시간간격을 측정하여 성능을 유추해 보았을 때 CPU pinning을 사용하는 것 보다 OS에게 스케쥴링을 맡기는 것이 성능이 더 좋게 나왔기 때문이다. 왜냐하면 현실적으로 백그라운드에서 실행되는 프로그램이 다양하기 때문에 코어에 고정시켜서 사용하는 것보다 OS가 스케쥴링을 통해 배치해주는 것이 더 효율적이기 때문이라 생각한다. 당연하지만 OUTPUT스레드의 경우는 독점코어를 사용하고 다른 스레드들이 그 외의 코어들을 사용하게 했을 때 훨씬 빠른 속도를 보여주었다.

4)Synchronization

클라이언트 전송 스레드들의 Synch 문제는 mutex로 해결하였다. 사실 다중 프로세스에서 입력을 받아서 스레드가 수신해서 전송만 하는 상황이라면 상관이 없겠지만 단일 프로세스에서 멀티스레드를 사용해 콘솔 입출력을 진행하다보니 스레드의 스케쥴링 문제가 생겨서 콘솔창 출력 및 입력이 뒤죽박죽이 되었기 때문에 Mutex로 한 스레드가 동작 중일 때 다른 스레드가 콘솔에서 입출력을 하지 못하도록 하였다. 세마포어 또한 Mutex로 돌리지 않는 이상 뒤섞여서 입출력이 되는 것을 확인하였다. OUTPUT스레드의 경우는 SYSCH를 하지 않았다. 서버로부터 데이터가 오는 대로 바로바로 출력이 되도록 하였기 때문이다.

서버의 경우는 PUT 혹은 DELETE가 진행될 때 데이터의 변동이 생기므로 다른 스레드들이 동작하지 못하도록 세마포어를 사용해 LOCK을 걸어주었다.

5) 인기 컨텐츠

서버측에 LRU캐시 배열을 설정해 두었다. 크기는 40개의 20%인 8개 이며, 사용되면 큐의 맨 뒤로 보내며, 내부 데이터의 수정이 생길 시 동기화를 진행하여 주었다. Get이 발생하기전 사전에 캐시에서 key값 비교를 통해 데이터가 있는지 체크를 한다.

6) 성능분석

별개의 성능분석 툴을 사용하지 않고 예제코드를 참고해 시간차이를 계산해 성능을 판단해 보았다. 그리고 동시에 콘솔에서 CPU사용 현황 및 메모리 사용현황을 참고해 성능을 판단하였다.

루프문을 사용해 GET을 반복적으로 발생시켜서 테스팅 하였다.

2. 시행착오

초기에는 하나의 메시지큐를 두고 서버의 스레드들이 TYPE을 사용해 자신의 데이터를 찾아가는 방식을 사용하였다. 그러나 루프를 통해 GET을 돌려 본 결과 입력에 비해 출력이 현저히 느렸다. 때문에 다중 메시지 큐를 구현해야 할 필요성을 느껴 다중 메세시큐를 이용한 구조설계를 하였다.

스레드간의 동기화에 많은 시간을 소모하였다. 서버스레드간의 동기화는 크게 문제가 되지 않았으나 클라이언트에서 입력을 받는 스레드들의 동기화에 많은 노력을 투자하였다. 입력 스레드간의 동기화는 콘솔입출력 때문에 많이 시간을 소모하였고, INPUT을 기다리는 동안 해당 스레드가 wait으로 가버려서 다른 스레드가 실행되었기 때문에 이를 해결하기 위한 방책을 강구하였다. 그러나 현재의 단일 프로세스에 콘솔입출력 구조에서는 mutex제어를 하나하나 실행하는 방법밖에 찾지 못하였다. OUTPUT스레드와 입력 스레드들 간의 실행순서는 일부러 맞추지 않았다.

왜냐하면 입력을 기다리는 동안 다른 유저들이 데이터를 못 받는 것은 현실적으로 말이 되지 않기 때문에 입력스레드들이 동작하는 사이에 출력 스레드가 동작해도 상관이 없도록 설계하였다.

캐싱을 사용한 결과 실행시간이 다소 빨라진 것을 확인할 수 있었다. 데이터의 수가 워낙 적기 때문에 눈에 띄게 빨라지지는 않았지만 근소하게 빨라졌다.

3. 문제점

완성을 하고 테스팅용 코드를 전부 지워버렸습니다. 콘솔 입출력 기반이다 보니 루프함수를

통해 동작을 시키는 것과 구조가 달라 별개의 스레드를 통해 루프함수 테스트를 진행하였었습니다.

헤더파일을 이용했다면 더 가독성이 좋은 코드를 짤 수 있었는데 시간관계상 헤더파일로 리팩토링을 진행하지 못하였습니다.