HW2

1분반

2024년 09월 24일

32211792

박재홍

SeekStrategy 인터페이스)

```java
package template;

public interface SeekStrategy {
    void seek(int[] queue, int start);

}

```

DiskShedular 클래스)

```java
public class DiskScheduler {

    private SeekStrategy strategy; // Seek 전략을 저장하는 인터페이스. FCFS, SSTF, SCAN 등 여러 알고리즘이 이 인터페이스를 구현

    public DiskScheduler() {     // 기본 생성자
    }

    public void setSeekStrategy(SeekStrategy strategy){ // 사용자가 원하는 디스크 탐색을 설정하는 메소드
        this.strategy = strategy; // SCAN, FCFS, SSTF, CSCAN 등으로 설정
    }

    public void executeSeek(int[] queue, int start){ // 탐색을 시작하는 메소드

        strategy.seek(queue, start); // queue 는 탐색해야할 트랙의 순서가 들어있는 배열, start 는 탐색을 시작해야할 헤드의 초기위치

    }
}
```

seek 전략을 저장하는 인터페이스, FCFS, SSTF, SCAN, SCAN 등 여러 알고리즘이 이 인터페이스를 구현하도록 선언해주었다.
setSeekStrategy 메소드를 통해 사용자가 원하는 디스크 탐색을 선언해주었고 SCAN, FCFS, SSTF, CSCAN 등으로 설정해주었다. 다음 excuteSeek 메소드를 통해 탐색을 시작하게 했다.

SeekAlgorithm 클래스)

```
package template;


public class SeekAlgorithm {

    // SCAN 알고리즘에서는 배열을 복사해서 정렬한 후에 sta
    public void seekBySCAN(int[] queue, int start) {
    }

    // CFS(First Come First Served 알고리즘은 **순차적으로
    // 화면에 출력. 마지막에 줄바꿈 문자 출력.
    public void seekByFCFS(int[] queue, int start) {
    }

    // 현재 위치에서 가장 가까운 요소를 큐에서 찾아 처리.
    public void seekBySSTF(int[] queue, int start) {
    }

}
```

seekalgorithm 클래스를 통해 각 알고리즘들을 찾을 수 있게 해주었다.

FCFSSeekStrategy 클래스)

```
1   package template;
2
3   public class FCFSSeekStrategy implements SeekStrategy {
4       @Override
5       public void seek(int[] queue, int start){
6           System.out.print(s:"FCFS ");
7           for( int q : queue){ // for-each 문을 사용해 queue에 있는 배열들을 q에 넣음
8               System.out.printf(format:"%d ", q ); // q에 있는 배열를 출력
9           }
10          System.out.printf(format:"\n");
11      }
12  }
13
```

FCFS 알고리즘은 queue에 있는 입력값 그대로 출력하는 알고리즘인데 우선 어떤 알고리즘인지 알 수 있도록 출력 앞에 FCFS가 나오도록 해주었고 for-each문을 통해 q에 있는 배열들을 출력할 수 있도록 설정해주었다.

SSTFSeekStrategy 클래스)

```java
package template;


public class SSTFSeekStrategy implements SeekStrategy {
    @Override
    public void seek(int[] queue, int start) {
        boolean[] visited = new boolean[queue.length]; // 트랙 방문 여부 확인
        int currentPosition = start;  // 현재 헤드 위치


        System.out.print(s:"SSTF " );

        for (int i = 0; i < queue.length; i++) {
            int closestTrackIndex = -1;
            int minSeekTime = Integer.MAX_VALUE;


            for (int j = 0; j < queue.length; j++) { // 아직 방문하지 않은 트랙 중에서 가장 가까운 트랙을 찾음
                if (!visited[j]) {
                    int seekTime = Math.abs(currentPosition - queue[j]);
                    if (seekTime < minSeekTime) {
                        minSeekTime = seekTime;
                        closestTrackIndex = j;
                    }
                }
            }


            visited[closestTrackIndex] = true; // 가장 가까운 트랙 방문
            currentPosition = queue[closestTrackIndex];


            System.out.print(currentPosition + " " );
        }

        System.out.println();
    }
```

SSTF 알고리즘은 헤드위치에 대해서 가장 가까운 곳을 가는것인데 우선적으로 한번 방문한 위치에 대해서는 재방문을 하면 안되기 때문에 boolean 배열을 통해 위치에 대해 방문을 했는지 안했는지를 알 수 있게 해주었고 currentposition에 현재 시작위치를 넣어주었다. 그 후 for문을 통해 아직 방문하지 않은 트랙중에서 가장 가까운 트랙을 찾을 수 있게 했다.

SCANSeekStrategy 클래스)

```java
package template;
import java.util.*;
public class SCANSeekStrategy implements SeekStrategy {
    @Override
    public void seek(int[] queue, int start){
        Arrays.sort(queue); // 큐를 정렬하여 방향성을 유지

    int splitIndex = 0; // 첫 번째로 방문해야 할 인덱스 (현재 위치보다 작은 값 중 최대 값)
    while (splitIndex < queue.length && queue[splitIndex] < start) {
        splitIndex++;
    }

    System.out.print(s:"SCAN ");

        for (int i = splitIndex - 1; i >= 0; i--) { // 작은 방향인 왼쪽으로 처리
        System.out.print(queue[i] + " ");
    }


    if (splitIndex > 0) { // 왼쪽 가장 끝인 0으로 이동
        System.out.print(s:"0 ");
      }


      for (int i = splitIndex; i < queue.length; i++) { // 큰 방향인 오른쪽으로 처리
          System.out.print(queue[i] + " ");
      }

      System.out.println();
  }
  }
```

SCAN 알고리즘은 이동하는 방향에 대해 끝까지 이동 후 역방향에 대해 끝까지 다시 이동하는 알고리즘이다. 그렇기에 처음에 큐를 정렬하여 방향성을 유지해주었고 현재 위치보다 작은값 중 최대 값인 곳을 첫 번째로 방문해야 할 인덱스로 정해 이동하게 했다. for문을 통해 작은 방향인 왼쪽으로 이동하게 했고 그 후 왼쪽의 끝인 0으로 이동해서 0을 출력하게 했고 그 다음 큰 방향인 오른쪽으로 이동할 수 있도록 해주었다.

Your Code)

CSCANSeekStrategy 클래스)

```java
package template; // Your Code
import java.util.*;
public class CSCANSeekStrategy implements SeekStrategy {

    @Override
    public void seek(int[] queue, int start){
        Arrays.sort(queue); // 큐를 정렬

        int splitIndex = 0;  // 현재 위치보다 큰 값 중 최소 값이며 첫 번째로 방문해야 할 인덱스
        while (splitIndex < queue.length && queue[splitIndex] < start) {
            splitIndex++;
        }

        System.out.print(s:"C-SCAN ");


        for (int i = splitIndex; i < queue.length; i++) { // 큰방향인 오른쪽 부터 처리
            System.out.print(queue[i] + " ");
        }


        if (splitIndex < queue.length) { // 오른쪽 가장 끝인 199로 이동 후, 처음인 0으로 이동
            System.out.print(s:"199 0 ");
        }


        for (int i = 0; i < splitIndex; i++) { // 처음부터 다시 오른쪽으로 처리
            System.out.print(queue[i] + " ");
        }

        System.out.println();
    }

    }
```

CSCAN 알고리즘은 한 방향에 대해 끝으로 간 다음 그 후 아예 반대편에서부터 다시 출력하는 알고리즘이다. 예를 들어 0 과 199가 서로 끝이라면 199에 도착 후 0으로 간 후 다시 오른쪽으로 가는 식이다. 우선 큐를 정렬 한 후에 현재 위치보다 큰 값중 최소값이며 첫 번째로 방문해야 할 인덱스를 정해주었고 for문을 통해 큰 방향인 오른쪽부터 갈 수 있도록 처리해주었다. 그 후 오른쪽의 끝인 199에 도착 후 199를 출력하게 해주었고 0에서부터 다시 오른쪽으로 가기에 0도 출력되게 해주었다.

MainTest 클래스)

```java
package template;

public class MainTest {
    public MainTest() {
        int[] queue = { 70,  153,  24,  57,  140,  15,  115,  80, 85 };
        int start = 43;

        // seek
        DiskScheduler scheduler = new DiskScheduler(); // DiskScheduler 객체 생성
        SeekStrategy[] algorithms = { new SCANSeekStrategy() , new FCFSSeekStrategy()  , new SSTFSeekStrategy(), new CSCANSeekStrategy()  };
        for ( SeekStrategy algorithm : algorithms) {
            scheduler.setSeekStrategy(algorithm);
            scheduler.executeSeek(queue.clone(), start); // 원본 queue 를 건드리지 않기위해 clone() 을 사용

        }

    }
}

```

우선 큐배열에 정수들을 넣어주었고 시작지점을 43으로 설정해주었다. 그 후 DiskSchedular 객체를 생성해주었고 그 후 SeekStrategy배열에 각 알고리즘의 대한 객체를 생성해 넣어주었다. 그 후 for-each 문을 통해 알고리즘을 찾고 알고리즘을 실행할 수 있도록 해주었고 queue가 계속 입력된 값 그대로가 아니라 정렬된 값으로 나오기에 queue.clone()을 사용해 원본 queue를 사용할 수 있게 해주었다.

실행결과)

```
SCAN 24 15 0 57 70 80 85 115 140 153
FCFS 70 153 24 57 140 15 115 80 85
SSTF 57 70 80 85 115 140 153 24 15
C-SCAN 57 70 80 85 115 140 153 199 0 15 24
PS C:\Users\박재홍\source\java24\HW2>
```