OS-LAB PROJECT

Disk-Scheduling-Algorithms

Project By:

Raj Ailani (18BIT088)
Smit Shah (18BIT110)
Shrutik Shah (18BIT106)
Harshil Jain (18BIT025)
Vishwas Tomar(18BIT118)

Disk-Scheduling Algorithms

1. **FCFS:** FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue. Let us understand this with the help of an example.

```
53  def FCFS(hp, requests):
54      time = 0
55      pos = hp
56      distance = []
57      distance.append(hp)
58      for request in requests:
59          time += abs(request-pos)
60          pos = request
61          distance.append(pos)
62
63      # calculate average seek time
64      save('total', time, 'distance', distance)
65
```

2. **SSTF:** In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system. Let us understand this with the help of an example.

```
def SSTF(hp, reqs):
    requests = reqs.copy()
    time = 0
    position = hp
    heap = []
    distance = []
    distance.append(hp)
    while len(requests) > 0:
        for r in requests:
            heappush(heap, (abs(position-r), r))
        x = heappop(heap)[1]
        time += abs(position-x)
        position = x
        distance.append(x)
        requests.remove(x)
        heap = []

    # calculate average seek time
    save('total', time, 'distance', distance)

```

3. **SCAN:** In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

```
26  def SCAN(hp, reqs):
27      requests = reqs.copy()
28      pos = hp
29      time = 0
30      end = 200
31      start = 0
32      distance = []
33      distance.append(hp)
34      for i in range(pos, end+1):
35          if i in requests:
36              time += abs(pos-i)
37              pos = i
38              distance.append(i)
39              requests.remove(i)
40
41      time += abs(pos-end)
42      pos = end
43      for i in range(end, start-1, -1):
44          if i in requests:
45              time += abs(pos-i)
46              pos = i
47              distance.append(i)
48              requests.remove(i)
49
50      save('total', time, 'distance', distance)
```

4. **<u>CSCAN</u>**: In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.

These situations are avoided in *CSCAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

```
67  def C_SCAN(hp, reqs):
68      requests = reqs.copy()
69      pos = hp
70      time = 0
71      end = 200
72      start = 0
73      distance = []
74      distance.append(hp)
75      # seek from curr_pos to end which is 200
76      for i in range(pos, end+1):
77          if i in requests:
78              time += abs(pos-i)
79              pos = i
80              distance.append(i)
81              requests.remove(i)
82      time += abs(pos-end)
83      pos = end
84      # seek to hp from start
85      for i in range(start, hp+1):
86          if i in requests:
87              time += abs(pos-i)
88              pos = i
89              distance.append(i)
90              requests.remove(i)
91
92      # calculate average seek time
93      save('total', time, 'distance', distance)
94
```

5. **LOOK:** It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

```
96  def LOOK(hp, reqs):
97      requests = reqs.copy()
98      pos = hp
99      time = 0
100     end = max(requests)
101     start = min(requests)
102     distance = []
103     distance.append(hp)
104     # seek from curr_pos to end which is 200
105     for i in range(pos, end+1):
106         if i in requests:
107             time += abs(pos-i)
108             pos = i
109             distance.append(i)
110             requests.remove(i)
111
112     # seek back to start
113     for i in range(end, start-1, -1):
114         if i in requests:
115             time += abs(pos-i)
116             pos = i
117             distance.append(i)
118             requests.remove(i)
119     # calculate average seek time
120     save('total', time, 'distance', distance)
```

6. **<u>CLOOK:</u>** As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

```
123  def C_LOOK(hp, reqs):
124      requests = reqs.copy()
125      pos = hp
126      time = 0
127      end = max(requests)
128      start = min(requests)
129      distance = []
130      distance.append(hp)
131      # seek from curr_pos to max of list
132      for i in range(pos, end+1):
133          if i in requests:
134              time += abs(pos-i)
135              pos = i
136              distance.append(i)
137              requests.remove(i)
138
139      time += abs(pos-start)
140      pos = start
141      # seek to hp from start
142      for i in range(start, hp+1):
143          if i in requests:
144              time += abs(pos-i)
145              pos = i
146              distance.append(i)
147              requests.remove(i)
148
149      # calculate average seek time
150      save('total', time, 'distance', distance)
151
```