CS342 Operating System Lab – Endsem

Name: M Maheeth Reddy	Roll No.: 1801CS31	Date: 20-Apr-2021

Answer 1:

Suppose there are n processes P1, P2, P3 … Pn in the system such that the processes require x1, x2, x3 ….. xn copies of resource R respectively.

To find the maximum number of copies of resource R to ensure deadlock, we can do the following:

In worst case, the number of units that each process holds is one less than its maximum demand. In other words, process Pi holds (xi – 1) copies of resource R (i varies from 1 to n).

So, maximum number of copies of resource R to ensure deadlock is (sum of max needs of all n processes) – n.

If there is one more unit of resource R in the system, then the system would be ensured deadlock free.

So, minimum number of copies of resource R to ensure **no** deadlock is (sum of max needs of all n processes) – n + 1.

For Part (i):

Total copies of resource = r

No. of processes = n

No. of copies required by each resource = c

To ensure deadlock free operation, r = (c + c + c + … + c (n times)) – n + 1

- ⇨ r = n*c – n + 1
- ⇨ r – 1 = n*(c – 1)
- ⇨ n = floor((r-1)/(c-1)) [floor(x) = integer part of x, where x is real number]

Details of submitted file:

Filename	Q1_i.c
Compile	gcc Q1_i.c -o Q1_i
Execute	./Q1_i

Testcase 1:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:26:29  IST]
└─>$ ./Q1_i
No. of copies of resource (r): 6
No. of copies of resource required by each process (c): 2
Maximum of 5 process(es) are required to ensure a deadlock free operation
```

Testcase 2:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:27:48  IST]
└─>$ ./Q1_i
No. of copies of resource (r): 100
No. of copies of resource required by each process (c): 4
Maximum of 33 process(es) are required to ensure a deadlock free operation
```

Testcase 3:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:27:54  IST]
└─>$ ./Q1_i
No. of copies of resource (r): 10
No. of copies of resource required by each process (c): 4
Maximum of 3 process(es) are required to ensure a deadlock free operation
```

For Part (ii): The two yellow-highlighted formulae above would be sufficient.

Details of submitted file:

Filename	Q1_ii.c
Compile	gcc Q1_ii.c -o Q1_ii
Execute	./Q1_ii

Testcase 1:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:31:44  IST]
└─>$ gcc Q1_ii.c -o Q1_ii
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:33:29  IST]
└─>$ ./Q1_ii
Number of processes (n): 3
Number of copies for process 1, (c1): 2
Number of copies for process 2, (c2): 3
Number of copies for process 3, (c3): 4

Maximum of 6 copies of resource are required to
not guarantee deadlock free operation for the system

Minimum of 7 copies of resource are required to
guarantee deadlock free operation for the system
```

Testcase 2:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:34:05  IST]
└─>$ ./Q1_ii
Number of processes (n): 4
Number of copies for process 1, (c1): 35
Number of copies for process 2, (c2): 45
Number of copies for process 3, (c3): 54
Number of copies for process 4, (c4): 20

Maximum of 150 copies of resource are required to
not guarantee deadlock free operation for the system

Minimum of 151 copies of resource are required to
guarantee deadlock free operation for the system
```

Testcase 3:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[05:35:47  IST]
└─>$ ./Q1_ii
Number of processes (n): 3
Number of copies for process 1, (c1): 13
Number of copies for process 2, (c2): 18
Number of copies for process 3, (c3): 10

Maximum of 38 copies of resource are required to
not guarantee deadlock free operation for the system

Minimum of 39 copies of resource are required to
guarantee deadlock free operation for the system
```

Answer 2:

My Page Replacement Algorithm

In the existing page replacement algorithms, pages are read one by one. I have designed an algorithm based on block reading of pages. Such algorithm would be helpful when there is frequent disc access. In my algorithm, whenever there is a page fault, I retrieve adjacent distinct pages instead of just retrieving the missing page. Number of pages retrieved is equal to number of frames allocated for the process.

Advantage of my algorithm: Since multiple distinct pages are retrieved, the number of page faults is greatly reduced when compared to existing algorithms. This is because; the existing algorithms retrieve only the missing page, which cause page faults later, but this doesn't happen in my algorithm.

Testcase:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[06:40:19  IST]
└─>$ g++ Q2.cpp -o Q2
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[06:40:22  IST]
└─>$ ./Q2
No. of Trials: 5

Inputs for Trial 1:
-------------------------
No. of Frames: 3
Length of page sequence: 10
Enter Page Sequence: 4 7 6 1 7 6 1 2 7 2

Inputs for Trial 2:
-------------------------
No. of Frames: 4
Length of page sequence: 16
Enter Page Sequence: 1 2 3 4 2 7 5 1 1 6 4 7 2 1 2 5

Inputs for Trial 3:
-------------------------
No. of Frames: 3
Length of page sequence: 15
Enter Page Sequence: 8 1 2 3 1 4 1 5 3 4 1 4 2 3 1

Inputs for Trial 4:
-------------------------
No. of Frames: 3
Length of page sequence: 10
Enter Page Sequence: 1 2 1 3 7 4 5 6 3 1

Inputs for Trial 5:
-------------------------
No. of Frames: 4
Length of page sequence: 10
Enter Page Sequence: 2 2 4 3 1 6 7 3 5 6
```

Output:

```
Output:
-------

Page faults for various trials:
A:      Trial 1: 3              Trial 2: 4              Trial 3: 5              Trial 4: 3              Trial 5: 2
LRU:    Trial 1: 6              Trial 2: 13             Trial 3: 12             Trial 4: 9              Trial 5: 7
FCFS:   Trial 1: 6              Trial 2: 13             Trial 3: 12             Trial 4: 9              Trial 5: 7

Time taken for various trials:
Trial 1: own algorithm: 5 us            LRU: 10 us              FIFO: 7 us
Trial 2: own algorithm: 5 us            LRU: 12 us              FIFO: 3 us
Trial 3: own algorithm: 6 us            LRU: 22 us              FIFO: 6 us
Trial 4: own algorithm: 5 us            LRU: 10 us              FIFO: 4 us
Trial 5: own algorithm: 3 us            LRU: 7 us               FIFO: 3 us

Total time for all trials by my own algorithm = 24 us
Total time for all trials by LRU algorithm = 61 us
Total time for all trials by FIFO algorithm = 23 us
```

Observation:

In the test cases above, there are some pages that are retrieved repeatedly. As per the advantages I mentioned above, the number of page faults for my algorithm is always lesser than LRU and FCFS/FIFO.

If there are more page faults, then the algorithm has to look for a proper location in the frame to place the retrieved page. So, running time increases. This explains the lesser running time of my algorithm when compared to LRU and FIFO/FCFS.

Answer 3:

Note: This program takes input from user via terminal and not from file.

Simulation of given process scheduling algorithm

Filename	Q3.cpp
Compile	g++ Q3.cpp -o Q3
Execute	./Q3

Test Case 1:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[07:07:44  IST]
└─>$ g++ Q3.cpp -o Q3
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[07:08:08  IST]
└─>$ ./Q3
Time slice for processor 1 (d1): 5
Time slice for processor 2 (d2): 10
Number of Jobs: 7
Enter job id, arrival time and execution time for each job:
A 0 18
B 0 12
C 0 7
D 0 11
E 0 28
F 7 18
G 16 12
Time instant: 53
Output:
job_id  arrival_time    Execution time  start_time      end_time        current_status
A       0               18              0               43              C
B       0               12              0               24              C
C       0               7               5               22              C
D       0               11              10              30              C
E       0               28              10              -1              E
F       7               18              20              51              C
G       16              12              25              52              C
Note: if end_time = -1, then process is not yet completed
```

Test Case 2:

```
┌─[maheeth@maheeth-PC:~/D/W/C/endsem]─[07:08:31  IST]
└─>$ ./Q3
Time slice for processor 1 (d1): 10
Time slice for processor 2 (d2): 3
Number of Jobs: 6
Enter job id, arrival time and execution time for each job:
A 2 3
B 7 12
C 4 13
E 1 8
D 0 7
G 8 14
Time instant: 25
Output:
job_id  arrival_time    Execution time  start_time      end_time        current_status
D       0               7               0               7               C
E       1               8               1               18              C
A       2               3               3               6               C
C       4               13              7               24              C
B       7               12              9               -1              E
G       8               14              12              -1              E
Note: if end_time = -1, then process is not yet completed
```

Answer 4:

Simulation of S-LOOK disk scheduling algorithm

Filename	Q4.cpp
Compile	g++ Q4.cpp -o Q4
Execute	./Q4

Test Case 1:

```
┌──[maheeth@maheeth-PC:~/D/W/C/endsem]-[04:38:07  IST]
└─>$ g++ Q4.cpp -o Q4
┌──[maheeth@maheeth-PC:~/D/W/C/endsem]-[04:38:16  IST]
└─>$ ./Q4
Number of Cylinders: 200
Head position: 98
Disk request: 98 183 37 122 14 124 65 67

S-LOOK Algorithm
Disk head movement:     98 -> 67 -> 65 -> 37 -> 14 -> 122 -> 124 -> 183
Total Head movement (THM): 253
Seek Time is 1265 ms for seek rate of 5 ms
```

Test Case 2:

```
┌──[maheeth@maheeth-PC:~/D/W/C/endsem]-[04:38:58  IST]
└─>$ ./Q4
Number of Cylinders: 200
Head position: 100
Disk request: 23 89 132 42 187

S-LOOK Algorithm
Disk head movement:     89 -> 42 -> 23 -> 132 -> 187
Total Head movement (THM): 241
Seek Time is 1205 ms for seek rate of 5 ms
```

Test Case 3:

```
┌──[maheeth@maheeth-PC:~/D/W/C/endsem]-[04:39:40  IST]
└─>$ ./Q4
Number of Cylinders: 1001
Head position: 460
Disk request: 660 740 350 700 800 443 880 761

S-LOOK Algorithm
Disk head movement:     443 -> 350 -> 660 -> 700 -> 740 -> 761 -> 800 -> 880
Total Head movement (THM): 640
Seek Time is 3200 ms for seek rate of 5 ms
```

----------x----------x----------