OPERATING SYSTEM PROJECT

Phase I

BY :

Sara El besomy 201601849

Project Description:

The project is to design a multi-level feedback queue schedular working as follow:

- The scheduler consists of two queues.
- Processes in the first queue are scheduled using a round-robin scheduling algorithm.
- Processes in the second queue are scheduled using a First-Come-First-Serve scheduling algorithm.
- New Processes enter the scheduler at the first queue. If the process is not complete after 4 units, the process is demoted to the second queue.
- No rules for promoting a process from second to first queue.
- The execution of the second queue only begins when the first queue finishes its execution.

User Input:

User needs to enter the no. of processes that will be executed. It should be entered ascendingly with the arrival time. For each process, the user enters the arrival time and the burst time (CPU execution time). This could be written as follow:

```
26  int main()
27  {
28      int N = 0;   // total number of processes
29      cout << "\nEnter the number of processes :";
30      cin >> N;
31      process P[10] = { 0 };
32      for (int i = 0; i < N; i++)
33      {
34          cout << "\nImportant Note : The processes should be ascendingly sorted with the arrival time.";
35          cout << "\nEnter the data of process (" << i+1 << ") :";
36          cout << "\nEnter the arrival time :";
37          cin >> P[i].arrival_time;
38          cout << "\nEnter the burst time :";
39          cin >> P[i].burst_time_original;
40
41          P[i].q_no = 1;
42          P[i].burst_time = P[i].burst_time_original;   // to easily update
43
44      }
```

Program body:

In any schedular, the process is the main part because it the object that is executed. So that it is needed to create a struct that contains all the properties of the process. Such that the following code.

```
17    struct process
18    {
19        int arrival_time = 0;
20        int burst_time_original = 0;
21        int burst_time = 0;
22        int q_no = 0;
23    };
```

Variables illustration:

arrival_time	Indicates the actual arrival time of the user and it is a user input.
burst_time_original	Indicates the CPU execution time and it is a user input.
burst_time	Indicates the remaining CPU execution time after every execution for the process.
q_no	Indicates the queue number of the process. Its value is equal 1 once the process enters the schedular and it remains 1 till 4 times of execution, and then the process demoted to the second queue. So, the value will equal 2.

First queue: Round Robin algorithm

As required, in the first queue we implement round robin algorithm so, define a time quantum equals 2. Briefly in this algorithm, the schedular choose the first process to execute for time quantum an this is a one unit, then the schedular check if there is another process wants to execute by comparing the current time index with the arrival time of the next process. If there is a process, it will execute also for a time quantum. If there is no process, the current process will execute for another time quantum until it complete four units of execution, then if it is not finish yet, it will demote to the second queue.

To implement the algorithm, we need to define the following variables.

```
52        // Handling queue1 ( round-robin scheduling algorithm )
53        int n = N;  // number of process ......> to easily update it
54        int total_time = 0; // total time of execution
55        int q = 2;  // time quantam
56        int c = 0;  // flage to know that the process is executed
57        int i = 0;   // interations index
```

In each process enters the first queue, we should handle two cases:

- The burst time of the process is less than the quantum

  In this case, we move the total_time which is the time index by the burst time of the process. We also need to set the process burst time to zero and change the value of the flag **c** from zero to one to indicate that this process is fully executed.

- The burst time of the process is greater than or equal the quantum

  In this case, we move the time index by the quantum for every execution. We also subtract the quantum from the process burst time to get the remaining burst time.

In both cases, we calculate variable called completion_time which indicates the time of every execution of every process.

```
if (P[i].burst_time <= q && P[i].burst_time > 0)
{
    total_time = total_time + P[i].burst_time;
    P[i].burst_time = 0;
    c = 1;
    completion[i] = completion[i] + total_time;
    cout << " \nProcess (" << i + 1 << ")" << "--------> "<< completion[i];
}
else if (P[i].burst_time > 0)
{
    P[i].burst_time = P[i].burst_time - q;
    total_time = total_time + q;
    completion[i] = completion[i] + total_time;
    cout << " \nProcess ("<< i + 1 << ")"<< "--------> "<< completion[i]
}
```

After the changes happened after the process execution, we want to check if the process is still in the first queue and not totally finish its execution so we check if the burst time of the process is equals zero and if the flag equals one. If so, we need to remove the process from the first queue and set the flag to zero. The program code of that could be as following:

```
if (P[i].burst_time == 0 && c == 1)  // Here the process is fully executed
{
    n--;
    int wait_time = total_time - P[i].arrival_time - P[i].burst_time;
    c = 0;


}
```

After the process was executed, we need to move to the next process and do the same. To do that we have 3 cases:

- We are in the last process so that we return to the first one.
  And this could be detected by comparing the process number with the number of total processes N.
- The arrival time of the next process is less than or equal the current time index.
  In that case, we move to the next process by increasing the counter of the processes by one (i++).
- We are in any process but not the last one, and the next process does not come yet.
  In this case we return to the first process so we set i = 0.

```
if (i == N - 1)  // the case of last process and return to the first one
{

    i = 0;


}
else if (P[i+1].arrival_time <= total_time)    // the case in which it will forward to the next process
{
    i++;
}
else                    // return to the first process
{
    i = 0;
}
```

As required after executes 4 units of quantum, the process leaves the first queue and demotes to the second one to complete its remaining burst time if any. So, before the execution of any process we need to check if it exceeds the 4 units of quantum. By the coming condition we can do that and this is placed in the first place in the while-loop:

```
while (  n != 0 )
{

    int completion[10] = { 0 };    // completion time for every unit

    if (P[i].burst_time <= ((P[i].burst_time_original) - (4 * q)))
    {
        P[i].q_no = 2;
        n--;
    }
    else
    {
```

The second queue: First Come First Served algorithm

After the processes exceed the threshold of 4 units of quantum, they demote to the second queue to complete the execution. The algorithm here is simpler than RR algorithm. Once the process enters the queue, it starts execution. The algorithm is non-preemptive so, the next process should wait the first one to full finish its execution.

```
if (n == 0)
{
    for (int j = 0; j < N; j++)
    {
        int completion[10] = { 0 };    // completion time for every unit

        if (P[j].q_no == 2)
        {
                total_time = total_time + P[j].burst_time;
                completion[j] = completion[j] + total_time;
                cout << " \nProcess (" << j + 1 << ")" << "--------> " << completion[j];
        }


    }
}
```

And here also we calculate the completion_time variable.

Output:

The output of that program is the Gantt chart.

Example:

process	Arrival time	Burst time
P_1	0	5
P_2	2	5
P_3	3	4
P_4	5	7
P_5	6	2
P_6	8	9

The solution of that example is such that:

P_1	P_2	P_3	P_4	P_5	P_6	P_1	P_2	P_3	P_4	P_6	P_1	P_2	P_4	P_6	P_4	P_6	P_6
0	2	4	6	8	10	12	14	16	18	20	22	23	24	26	28	29	31	32

We use this example to test the program:

We run the program and enter the inputs which are the number of process and the data of each process.

```
Enter the number of processes :6

Important Note : The processes should be ascendingly sorted with the arrival time.
Enter the data of process (1) :
Enter the arrival time :0

Enter the burst time :5

Important Note : The processes should be ascendingly sorted with the arrival time.
Enter the data of process (2) :
Enter the arrival time :2

Enter the burst time :5

Important Note : The processes should be ascendingly sorted with the arrival time.
Enter the data of process (3) :
Enter the arrival time :3

Enter the burst time :4
```

The output is such that:

```
 Gantt Chart :
The time index starts from 0
Process (1)--------> 2
Process (2)--------> 4
Process (3)--------> 6
Process (4)--------> 8
Process (5)--------> 10
Process (6)--------> 12
Process (1)--------> 14
Process (2)--------> 16
Process (3)--------> 18
Process (4)--------> 20
Process (6)--------> 22
Process (1)--------> 23
Process (2)--------> 24
Process (4)--------> 26
Process (6)--------> 28
Process (4)--------> 29
Process (6)--------> 31
Process (6)--------> 32
```