```cpp
/*
Nhat-Huy Tran
10-20-2023
COP4106
CPU Scheduling Programming Assignment - FCFS Algorithm
*/

#include <iostream>
#include <algorithm>
#include <iomanip>

using namespace std;

class Process                                                   //Each object under Process will have their own data stored.  I.e. CPU Burst, Arrival time, etc.
{
    public:
        void setID(Process P[], int count);                     //Sets ID for each Process in the class array
        void MLFQ (Process P[], int count);                     //Main function of the algorithm MLFQ
        void check (Process P[], int ID);                       //Checks the CPU and I/O Burst inputs to see if a process is either complete, expires on time quantum or has remaining time.
        void sortProcess (Process P[], int count);              //Sorts Class array elements in order of updated arrival times and Queues.
        void setArrival(Process P[], int ID);                   //Sets and Updates the arrival time.
        void initiate(Process P[],int i);                       //Asks for user input of processes' CPU and I/O Bursts.
        void useRemainTime(Process P[],int i);                  //If a process fails to finish before time quantum, use time quantum and have remaining time.
        void finishRemain(Process P[],int i);                   //If the process has remaining time burst finishes it, use the remaining burst.
        void waitCheck(Process P[], int count);                 //Checks if the process is in the waiting queue during I/O Burst or if the execution time has arrived after the burst.
        void idleCheck(Process P[], int count);                 //Checks if there are no available processes in the ready queue.
        void setResponse(Process P[], int count);               //Sets Response time for each process.
        void IsQueue1Available(Process P[], int count);         //Checks if there is a process available in previous queue.
        void IsQueue2Available(Process P[], int count);         //Checks if there is a process available in previous queue.
        void sortID(Process P[], int count);                    //Sorts Class array elements in order of Process ID for results output.
        void CompComplete(Process P[], int ID);                 //Second part of the check function.  Checks if the process is complete.
        void calculateTurnaround(Process P[], int count);       //Calculates turnaround time for each process and average.
        void calculateWaiting(Process P[], int count);          //Calculates Waiting time for each process and average.
        void calculateResponse(Process P[], int count);         //Gets response time for each process and average.
        void endProcess(Process P[], int count);                //Ends the loop and displays the results after all processes in the array are completed.
    private:
        int CPUBurst;
        int IOBurst;
        int arrival=0;
        int num;
        int firstArrive=0;
        int totalBurst=0;
        int complete=0;
        int waitQueue=0;
        int Queue=1;
        int remain=0;
```

```
        int waiting;
        int turnaround;
        int response;
};

int minBurst=0;                                             //Smallest burst in the class array.
int Queue1Ava=0;                                            //Checks if there is a process available in Queue 1 that are ready to be processed.
int Queue2Ava=0;                                            //Checks if there is a process available in Queue 2 that are ready to be processed.
int completeCount=0;                                        //Keeps track of how many process are completed.
int Queue1Count=8, Queue2Count=0, Queue3Count=0;            //Checks and moves each process to a certain queue based on conditions
int exeTime=0, idleTime=0;                                  //Execution time and Idle time for algorithm.
int responseCheck=0;                                        //Checks if the response times are fulfilled.
double AVGresponse, AVGturnaround, AVGwaiting;              //Averages of Turnaround, Waiting, and Response time.

int main()
{
    cout << "Welcome to the Multilevel Feedback Queue (MLFQ) Simulation!\n\n";
    Process P[8];                            //Initializes class array with 8 processes.
    P[8].setID(P, 8);                        //Sets ID for Processes in the class array.
    P[8].MLFQ(P, 8);                          //Initiates the algorithm.
}

void Process::endProcess(Process P[], int count)            //Ends the loop if all processes are completed.
{
    if (completeCount == 8)                                 //If all processes are completed, Prints out results for turnaround, waiting, response, and CPU Utilization
    {
        cout << "RESULTS:\n";
        cout << "______________________________________________________\n";
        cout << "\nTotal Execution time: " << exeTime << " units.\n";                  //Prints Total execution time to complete the algorithm.
        cout << "Total Idle time: " << idleTime << " units.\n";                      //Prints Total idle time between processes.

        double AVGcpu = exeTime-idleTime;                                        //Calculates CPU Utilization in the program.
        AVGcpu = AVGcpu/double(exeTime);
        AVGcpu = AVGcpu*100;

        sortID(P, 8);                                                              //Sorts Processes in ascending order based on Process ID.

        cout << "CPU Utilization: " << fixed << setprecision(2) << AVGcpu << "%\n";

        cout << "\nTurnaround time results:\n" << "__________________________________\n";
        calculateTurnaround(P,8);
//Prints turnaround time for each process and average turnaround time.
        cout << "\nAverage turnaround time: " << fixed << setprecision(2) << AVGturnaround << "\n";

        cout << "\nWaiting time results:\n" << "__________________________________\n";
        calculateWaiting(P,8);
//Prints waiting time for each process and average waiting time.
        cout << "\nAverage waiting time: " << fixed << setprecision(2) << AVGwaiting << "\n";

        cout << "\nResponse time results:\n" << "__________________________________\n";
        calculateResponse(P,8);
//Prints response time for each process and average response time.
        cout << "\nAverage response time: " << fixed << setprecision(2) << AVGresponse << "\n";
```

```
        exit(1);                                                    //Ends program.
    }
    else
    {
        cout << "Not complete yet.\n\n";
    }
}

//Sets ID for Processes in the class array
void Process::setID(Process P[], int count)
{
    for(int i=0; i<count; i++)
    {
        P[i].num = i+1;
    }
}

//Main function of the algorithm MLFQ
void Process::MLFQ (Process P[], int count)
{
    for(int i=0; i<count; i++)
    {
        if(P[i].complete != 1)
        {
            idleCheck(P,8);
            waitCheck(P,8);
            if(P[i].waitQueue==0)
            {
                if((P[i].arrival <= minBurst)||(P[i].arrival <= exeTime))
                {

                    if (responseCheck==1)
                        {
                            sortProcess(P, i);                  //Sorts list based on arrival time and queues.
                        }

                    if(P[i].Queue==1)
                    {
                        initiate(P, i);
                    }
                    else if (P[i].Queue == 2)
                    {
                        IsQueue1Available(P,8);
                        if(Queue1Ava==0)
                        {
                            if(P[i].remain == 0)
                            {
                                initiate(P, i);
                            }
                            else
                            {
                                useRemainTime(P,i);
                            }
                        }
                        else
                        {
                            Queue1Ava=0;
                            continue;
                        }
                    }
                    else
                    {
                        IsQueue1Available(P,8);
```

```
                        if(Queue1Ava==0)
                        {
                            IsQueue2Available(P,8);
                            if(Queue2Ava==0)
                            {
                                if(P[i].remain == 0)
                                {
                                    initiate(P, i);
                                }
                                else
                                {
                                    useRemainTime(P,i);
                                }
                            }
                            else
                            {
                                Queue2Ava=0;
                                continue;
                            }
                        }
                        else
                        {
                            Queue1Ava=0;
                            continue;
                        }
                    }
                }
                else
                {
                    continue;
                }
            }
            else
            {
                continue;
            }
        }
        else
        {
            continue;
        }
    }

    if (responseCheck==0)
    {
        setResponse(P, 8);                                  //Sets Response time for each process.
        sortProcess(P, 8);                                  //Sorts the Process class array based on arrival time and Queue.
    }

    P[8].MLFQ(P,8);
}

//Sets Response time for each process.
void Process::setResponse(Process P[], int count)
{
    int exe=0;
    for (int i=0; i<count; i++)
    {
        if((P[i].Queue == 1)&&(P[i].remain ==0))
        {
            P[i].response = exe;
            exe = exe + P[i].CPUBurst;
        }
        else
        {
```

```
            P[i].response = exe;
            exe = exe + 5;
        }
    }
    responseCheck=1;                                        //Checks that response time has been recorded.
}

//Asks for user input of processes' CPU and I/O Bursts.
void Process::initiate (Process P[],int i)
{
    cout << "Process #" << P[i].num << " is set to arrive at " << P[i].arrival << " in Queue: " << P[i].Queue << endl;          //States the Process and their arrival time.
    cout << "Enter the CPU Burst time for Process #" << P[i].num << ": ";           //Reads input CPU Burst.
    cin >> P[i].CPUBurst;
    cout << "Enter the IO Burst time for Process #" << P[i].num << ": ";            //Reads input IO Burst.
    cin >> P[i].IOBurst;
    check(P,i);
    cout << "\n";
    endProcess(P, 8);
}

//If a process fails to finish before time quantum, use time quantum and have remaining time.
void Process::useRemainTime(Process P[],int i)
{
    cout << "Process #" << P[i].num << " is set to arrive at " << P[i].arrival << " in Queue: " << P[i].Queue << endl;
    cout << "Process #" << P[i].num << " has a remaining time of " << P[i].remain << endl;
    check(P,i);
}

//If the process has remaining time burst finishes it, use the remaining burst.
void Process::finishRemain(Process P[],int ID)
{
    P[ID].arrival = P[ID].IOBurst + P[ID].remain + exeTime;
    P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst + P[ID].IOBurst;
    exeTime = exeTime + P[ID].remain;
    P[ID].remain = 0;
}

//Sets and Updates the arrival time.
void Process::setArrival(Process P[], int ID)
{
    P[ID].arrival = P[ID].IOBurst + P[ID].CPUBurst + exeTime;
    exeTime = exeTime + P[ID].CPUBurst;
}

//Second part of the check function.  Checks if the process is complete.
void Process::CompComplete(Process P[], int ID)
{
    P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst + P[ID].IOBurst;
    if (P[ID].IOBurst==0)                                                       //0 units for IO bursts completes the process as it recognizes the last CPU burst.
    {
        P[ID].complete=1;
        exeTime = exeTime + P[ID].CPUBurst;
        P[ID].turnaround = exeTime - P[ID].firstArrive;                                          //Calculates turnaround time based on execution time - arrival time.
        cout << "Process #" << P[ID].num  << " is completed at " << exeTime << " units\n";    //Prints the completion time for each process.
        completeCount++;                                                                        //Increments to how many processes completed.
        cout << "Number of Processes complete: " << completeCount << endl;
        P[ID].arrival = P[ID].IOBurst + P[ID].remain + exeTime + 700;
    }
```

```
    else
    {
    setArrival(P, ID);
    }
}

//Checks the CPU and I/O Burst inputs to see if a process is either complete, expires on time quantum or has remaining time.
void Process::check(Process P[], int ID)
{
        if (P[ID].Queue == 1)
        {
            if(P[ID].CPUBurst > 5)
            {
                exeTime = exeTime + 5;
                P[ID].remain = P[ID].CPUBurst-5;
                P[ID].Queue = 2;
                cout << "Process #" << P[ID].num << " has moved down to Queue 2.\n";
                P[ID].arrival = exeTime + 5;
                cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
                Queue1Count--;
                Queue2Count++;
            }
            else
            {
                CompComplete(P, ID);
            }
        }
        else if (P[ID].Queue == 2)
        {
            if (P[ID].remain!=0)
            {
                if(P[ID].remain > 10)
                {
                    exeTime = exeTime + 10;
                    P[ID].remain = P[ID].remain-10;
                    P[ID].Queue = 3;
                    cout << "Process #" << P[ID].num << " has moved down to Queue 3.\n";
                    P[ID].arrival = exeTime + 10;
                    cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
                    Queue2Count--;
                    Queue3Count++;
                }
                else
                {
                    finishRemain(P, ID);
                }
            }
            else if(P[ID].CPUBurst > 10)
            {
                exeTime = exeTime + 10;
                P[ID].remain = P[ID].CPUBurst-10;
                P[ID].Queue = 3;
                cout << "Process #" << P[ID].num << " has moved down to Queue 3.\n";
                P[ID].arrival = exeTime + 10;
                cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
                Queue2Count--;
                Queue3Count++;
            }
            else
            {
                CompComplete(P, ID);
            }
        }
        else
        {
```

```cpp
            if(P[ID].remain!=0)
            {
                finishRemain(P, ID);
            }
            else
            {
                CompComplete(P, ID);
            }
        }
    cout << "\nCurrent execution time: " << exeTime << endl;
}

//Sorts Class array elements in order of updated arrival times and Queues.
void Process::sortProcess(Process P[], int count)
{
    for(int i=0; i<count; i++)                                      //Bubble sort for the class array.
    {
        for(int j=0; j<count-i-1; j++)
        {
            if((P[j].arrival > P[j+1].arrival)&&(P[j].Queue > P[j+1].Queue))                        //Sorts processes in ascending order based on arrival time and priority queue.
            {
                std::swap(P[j], P[j+1]);
            }
            else if ((P[j].arrival > P[j+1].arrival)&&(P[j].Queue == P[j+1].Queue))
            {
                std::swap(P[j], P[j+1]);
            }
            else if ((P[j].arrival == P[j+1].arrival)&&(P[j].Queue == P[j+1].Queue))                //In a situation where two processes have the same arrival time and queue, the class array is sorted based on process ID.
            {
                if(P[j].num > P[j+1].num)
                {
                    std::swap(P[j], P[j+1]);
                }
            }
        }
    }
}

//Check if the process is in the wait queue or waiting for execution time.
void Process::waitCheck(Process P[], int count)
{
    for(int i=0; i<count; i++)                              //Scans the list to check which processes are in ready queue or not.
    {
        if(exeTime < P[i].arrival)
        {
            P[i].waitQueue=1;                               //Goes into Waiting queue.
        }
        else
        {
            P[i].waitQueue=0;                               //Returns to Ready queue.
        }
    }
}

//Checks if there is a process available in previous queue.
void Process::IsQueue1Available(Process P[], int count)
{
    for(int i=0; i<count; i++)
    {
        if((P[i].Queue==1)&&(P[i].arrival <= exeTime))
        {
            Queue1Ava=1;
```

```
        }
    }
}

//Checks if there is a process available in previous queue.
void Process::IsQueue2Available(Process P[], int count)
{
    for(int i=0; i<count; i++)
    {
        if((P[i].Queue==2)&&(P[i].arrival <= exeTime))
        {
            Queue2Ava=1;
        }
    }
}

//Checks if there are no processes in the ready queue.
void Process::idleCheck(Process P[], int count)
{
    int waitCount=0, cCount=0;
    for(int i=0; i<count; i++)
    {
        if(P[i].complete != 1)
        {
            if(P[i].waitQueue==1)
            {
                waitCount++;                        //Counts which processes are in waiting queue.
            }
            else
            {
                continue;
            }
        }
        else
        {
            cCount++;                               //Counts which processes are completed.
        }
    }

    if(waitCount == (8-cCount))                        //Remaining processes that are not in ready queue, and the algorithm goes into idle.
    {
        exeTime++;
        idleTime++;
        cout << "The Algorithm is Idle at execution time: " << exeTime << " units.\n";
    }
}

//Sorts Class array elements in order of Process ID for results output.
void Process::sortID(Process P[], int count)
{
    for(int i=0; i<count; i++)                              //Bubble sort for sorting class array.
    {
        for(int j=0; j<count-i-1; j++)
        {
            if(P[j].num > P[j+1].num)
            {
                std::swap(P[j], P[j+1]);
            }
        }
    }
}

//Prints response time for each process and calculates average response time.
void Process::calculateResponse(Process P[], int count)
{
```

```
    double TotalResponse=0;
    cout << "\n";
    for(int i=0; i<count; i++)
    {
        cout << "Response time for Process #" << P[i].num << ": " << P[i].response << "\n";
        TotalResponse = TotalResponse + P[i].response;
    }
    AVGresponse = TotalResponse/8;
}

//Prints turnaround time for each process and calculates average turnaround time.
void Process::calculateTurnaround(Process P[], int count)
{
    double TotalTurnaround=0;
    cout << "\n";
    for(int i=0; i<count; i++)
    {
        cout << "Turnaround time for Process #" << P[i].num << ": " << P[i].turnaround << "\n";
        TotalTurnaround = TotalTurnaround + P[i].turnaround;
    }
    AVGturnaround = TotalTurnaround/8;
}

//Prints waiting time for each process and calculates average waiting time.
void Process::calculateWaiting(Process P[], int count)
{
    double TotalWaiting=0;
    cout << "\n";
    for(int i=0; i<count; i++)
    {
        P[i].waiting = P[i].turnaround-P[i].totalBurst;
        cout << "Waiting time for Process #" << P[i].num << ": " << P[i].waiting << "\n";
        TotalWaiting = TotalWaiting + P[i].waiting;
    }
    AVGwaiting = TotalWaiting/8;
}
```