```
/*
Nhat-Huy Tran
10-19-2023
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 set(Process P[], int count);                               //Main function of FCFS.  Sets the CPU and I/O bursts, arrival times, other data for each class array element.
        void setArrival(Process P[], int ID);                           //Sets and Updates the arrival time.
        void sortArrival(Process P[]);
        void setResponse(Process P[], int count);                       //Sets Response time for each process.
        void check(Process P[],int ID);                                 //Checks the CPU and I/O Burst inputs to see if a process is complete.
        void sortProcess(Process P[], int count);                       //Sorts Class array elements in order of updated arrival times.
        void sortID(Process P[], int count);                            //Sorts Class array elements in order of Process ID for results output.
        void getNextBurst(Process P[], int count);                      //Scans the entire class array and gets the smallest arrival time for a process to go first.
        void idleCheck(Process P[], int count);                         //Checks if there are no available processes in the ready queue.
        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 endProcess(Process P[], int count);                        //Ends the loop and displays the results after all processes in the array are completed.
        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.
    private:
        int CPUBurst;
        int IOBurst;
        int arrival=0;
        int num;
        int firstArrive=0;
        int totalBurst=0;
        int complete=0;
        int waitQueue=0;
        int waiting;
        int turnaround;
        int response;
};

int minBurst=0;                                               //Smallest burst in the class array.
int completeCount=0;                                          //Keeps track of how many process are completed.
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 First-Come-First-Serve (FCFS) 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].set(P, 8);                              //Initiates the algorithm.

//    P1({5,3,5,4,6,4,3,4},{27,31,43,18,22,26,24});
//    P2({4,5,7,12,9,4,9,7,8},{48,44,42,37,76,41,31,43});
//    P3({8,12,18,14,4,15,14,5,6},{33,41,65,21,61,18,26,31});
//    P4({3,4,5,3,4,5,6,5,3},{35,41,45,51,61,54,82,77});
//    P5({16,17,5,16,7,13,11,6,3,4},{24,21,36,26,31,28,21,13,11});
//    P6({11,4,5,6,7,9,12,15,8},{22,8,10,12,14,18,24,30});
//    P7({14,17,11,15,4,7,16,10},{46,41,42,21,32,19,33});
//    P8({4,5,6,14,16,6},{14,33,51,73,87});

}

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

void Process::set(Process P[], int count)                   //Main function of FCFS.
{
        for(int i=0; i<count; i++)                          //Loops through the 8 elements or processes.
        {
            if (P[i].complete != 1)                         //If a certain process is complete, it skips the element.
            {
                idleCheck(P,8);
                waitCheck(P,8);
                if(P[i].waitQueue == 0)                     //Checks if the process is in the waiting queue.
                {
                    getNextBurst(P, 8);                     //Gets the updated smallest arrival time.
                    if(P[i].arrival <= minBurst)
                    {
                        if (responseCheck==1)
                            {
                                sortProcess(P, i);                  //Sorts list based on arrival time.
                            }
                        cout << "Process #" << P[i].num << " is set to arrive at " << P[i].arrival << 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);
//Checks the CPU and I/O Burst inputs to see if a process is complete.
                        cout << "\n";
                        endProcess(P, 8);
//Checks if all processes are complete before ending the loop.
                    }
                    else
                    {
                        continue;                                                                  //Skips
```

```
the element.
                    }
                }
                else
                {
                    continue;                                                                   //Skips the element.
                }
            }
            else
            {
                continue;                                                                       //Skips the element.
            }
        }

        if (responseCheck!=1)
        {
            setResponse(P, 8);                                  //Sets Response time for each process.
            sortProcess(P, 8);                                  //Sorts the Process class array based on arrival time.
        }

        P[8].set(P, 8);                               //Loops the list back until all processes are complete.
}

void Process::setResponse(Process P[], int count)                       //Sets Response time for each process.
{
    int exe=0, total;
    for (int i=0; i<count; i++)
    {
            P[i].response = exe;
            total = total + exe;
            exe = exe + P[i].CPUBurst;
    }
    responseCheck=1;                                          //Checks that response time has been recorded.
}

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";
```

```cpp

        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.
    }
}

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

void Process::check (Process P[], int ID)                                                      //Checks the CPU and I/O Burst inputs to see if a process is complete.
{
    P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst + P[ID].IOBurst;          //Updates the combined number of used CPU and IO bursts for each process.  Important for waiting time calculation.
    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";    //Prints the completion time for each process.
        completeCount++;                                                                       //Increments to how many processes completed.
    }
    else
    {
        setArrival(P, ID);                                                                     //If the process has not ended, update arrival time.
    }
    cout << "\nCurrent execution time: " << exeTime << endl;                           //Displays the current execution time for the algorithm.
}

void Process::sortProcess(Process P[], int count)                        //Sorts Class array elements in order of updated arrival times.
{
    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)                            //Sorts processes in ascending order based on arrival time.
            {
                std::swap(P[j], P[j+1]);
            }
            else if (P[j].arrival == P[j+1].arrival)                     //In a situation where two processes have the same arrival time, the class array is sorted based on process ID.
            {
                if(P[j].num > P[j+1].num)
                {
```

```
                    std::swap(P[j], P[j+1]);
                }
            }
        }
    }
}

void Process::sortID(Process P[], int count)                    //Sorts Class array elements in order of Process ID for results output.
{
    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]);
            }
        }
    }
}

void Process::getNextBurst(Process P[], int count)              //Assigns the next minimum arrival time of the updated class array.
{
    for(int i=0; i<count; i++)
    {
        if(P[i].complete != 1)                                  //If a process is not complete, proceed.
        {
            minBurst = P[i].arrival;
        }
        else                                                    //If completed then skip element.
        {
            continue;
        }
    }

    for(int i=0; i<count; i++)          //If the arrival time of the process is less than the minimum arrival time and is not completed, Minimum arrival time is assigned for the next process CPU burst to happen.
    {
        if(P[i].arrival < minBurst)
        {
            if(P[i].complete != 1)
            {
                minBurst = P[i].arrival;
            }
            else
            {
                continue;               //if process is completed, skip the element.
            }
        }
    }
}

void Process::waitCheck(Process P[], int count)                 //Check if the process is in the wait queue or waiting for execution time.
{
    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.
        }
    }
}

void Process::idleCheck(Process P[], int count)         //Checks if there are no processes in the ready queue.
{
    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";
    }
}

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

void Process::calculateTurnaround(Process P[], int count)                                       //Prints turnaround time for each process and calculates average turnaround time.
{
    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;
}

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

```
    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;
}
```