```cpp
/*
Nhat-Huy Tran
10-21-2023
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 getBursts(Process P[], int count);                 //User input for each Process's CPU and IO Burst arrays.
        void SJF(Process P[], int count);                       //Main function of SJF.
        void sortProcess(Process P[], int count);               //Sorts Class array elements in order of updated CPU Bursts.
        void setArrival(Process P[], int ID);                   //Sets and Updates the arrival time.
        void check(Process P[], int ID);                        //Checks the CPU and I/O Burst inputs to see if a process is complete.
        void firstSort(Process P[], int count);                 //Sorts the process in ascending order by CPU Burst time at the start.
        void sortID(Process P[], int count);                    //Sorts Class array elements in order of Process ID for results output.
        void endProcess(Process P[], int count);                //Ends the loop and displays the results after all processes in the array are completed.
        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 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 getNextBurst(Process P[], int count);              //Gets the next lowest burst for SJF.
        void ZeroSort(Process P[], int count);                  //Sorts out any completed process out of the loop.
    private:
        int CPUBurst[10];
        int IOBurst[10];
        int arrival=0;
        int num;
        int firstArrive=0;
        int totalBurst=0;
        int complete=0;
        int waitQueue=0;
        int waiting;
        int turnaround;
        int response;
        int responseCheck=0;
        int CPUincre=0;
};

int completeCount=0;                                    //Keeps track of how many process are completed.
int minBurst;                                           //Stores the next burst.
```

```
51  int exeTime=0, idleTime=0;                                  //Execution time and Idle time for
algorithm.
52  double AVGresponse, AVGturnaround, AVGwaiting;              //Averages of Turnaround, Waiting, and
Response time.
53
54  int main()
55  {
56      cout << "Welcome to the Shortest-Job-First (SJF) Simulation!\n\n";
57      Process P[8];
58      P[8].setID(P, 8);
59      P[8].getBursts(P,8);
60      P[8].firstSort(P, 8);
61      P[8].SJF(P,8);
62
63  // Data used:
64  //    P1 5 3 5 4 6 4 3 4 0 0
65  //       27 31 43 18 22 26 24 0 0 0
66  //    P2 4 5 7 12 9 4 9 7 8 0
67  //       48 44 42 37 76 41 31 43 0 0
68  //    P3 8 12 18 14 4 15 14 5 6 0
69  //       33 41 65 21 61 18 26 31 0 0
70  //    P4 3 4 5 3 4 5 6 5 3 0
71  //       35 41 45 51 61 54 82 77 0 0
72  //    P5 16 17 5 16 7 13 11 6 3 4
73  //       24 21 36 26 31 28 21 13 11 0
74  //    P6 11 4 5 6 7 9 12 15 8 0
75  //       22 8 10 12 14 18 24 30 0 0
76  //    P7 14 17 11 15 4 7 16 10 0 0
77  //       46 41 42 21 32 19 33 0 0 0
78  //    P8 4 5 6 14 16 6 0 0 0 0
79  //       14 33 51 73 87 0 0 0 0 0
80
81  }
82
83  //Sets ID for Processes in the class array
84  void Process::setID(Process P[], int count)
85  {
86      for(int i=0; i<count; i++)
87      {
88          P[i].num = i+1;
89      }
90  }
91
92  //User input for each Process's CPU and IO Burst arrays.
93  void Process::getBursts(Process P[], int count)
94  {
95      for (int i=0; i<count; i++)
96      {
97          cout << "Enter the CPU Bursts for Process #" << P[i].num << ": ";
98          for (int j=0; j<10; j++)
99          {
100             cin >> P[i].CPUBurst[j];
101
102         }
103         cout << "Enter the IO Bursts for Process #" << P[i].num << ": ";
104         for (int j=0; j<10; j++)
105         {
106             cin >> P[i].IOBurst[j];
107         }
108     }
109     cout << "\nGREAT!\n";
110 }
111
112 //Main function of SJF.
113 void Process::SJF(Process P[], int count)
114 {
```

```
115      for (int i=0; i<count; i++)
116      {
117          if (P[i].complete == 1)
118          {
119              continue;
120          }
121          else
122          {
123              idleCheck(P,8);
124              waitCheck(P,8);
125              getNextBurst(P,8);
126              if(P[i].waitQueue==0)
127              {
128                  if((P[i].arrival <= exeTime)||(P[i].CPUBurst[P[i].CPUincre] <= minBurst))
129                  {
130                      if(P[i].CPUBurst[P[i].CPUincre] <= minBurst)
131                      {
132                          sortProcess(P, i);
133                          cout << "Process #" << P[i].num << " is set to arrive at " << P[i].arrival << endl;
134                          cout << "Current CPU Burst for Process #" << P[i].num << ": " << P[i].CPUBurst[P[i
].CPUincre] << endl;
135                          cout << "Current IO Burst for Process #" << P[i].num << ": " << P[i].IOBurst[P[i].
CPUincre] << endl;
136                          check(P,i);
137                          cout << "\n";
138                          endProcess(P,8);
139                      }
140                      else
141                      {
142                          continue;
143                      }
144                  }
145                  else
146                  {
147                      continue;
148                  }
149              }
150              else
151              {
152                  continue;
153              }
154          }
155      }
156      sortProcess(P, 8);
157
158      P[8].SJF(P,8);
159
160  }
161
162  //Ends the loop and displays the results after all processes in the array are completed.
163  void Process::endProcess(Process P[], int count)
164  {
165      if (completeCount == 8)                                    //If all processes are completed, Prints
out results for turnaround, waiting, response, and CPU Utilization
166      {
167          cout << "RESULTS:\n";
168          cout << "_____________________________________________________\n";
169          cout << "\nTotal Execution time: " << exeTime << " units.\n";                    //Prints Total
execution time to complete the algorithm.
170          cout << "Total Idle time: " << idleTime << " units.\n";                       //Prints Total idle
time between processes.
171
172          double AVGcpu = exeTime-idleTime;                                          //Calculates CPU
Utilization in the program.
173          AVGcpu = AVGcpu/double(exeTime);
174          AVGcpu = AVGcpu*100;
```

```
175
176         sortID(P, 8);                                                   //Sorts Processes
in ascending order based on Process ID.
177
178         cout << "CPU Utilization: " << fixed << setprecision(2) << AVGcpu << "%\n";
179
180         cout << "\nTurnaround time results:\n" << "_____________________________________\n";
181         calculateTurnaround(P,8);
//Prints turnaround time for each process and average turnaround time.
182         cout << "\nAverage turnaround time: " << fixed << setprecision(2) << AVGturnaround << "\n";
183
184         cout << "\nWaiting time results:\n" << "_____________________________________\n";
185         calculateWaiting(P,8);
//Prints waiting time for each process and average waiting time.
186         cout << "\nAverage waiting time: " << fixed << setprecision(2) << AVGwaiting << "\n";
187
188         cout << "\nResponse time results:\n" << "_____________________________________\n";
189         calculateResponse(P,8);
//Prints response time for each process and average response time.
190         cout << "\nAverage response time: " << fixed << setprecision(2) << AVGresponse << "\n";
191
192         exit(1);                                                                    //Ends
program.
193     }
194 }
195
196 //Sets and Updates the arrival time.
197 void Process::setArrival(Process P[], int ID)
198 {
199     P[ID].arrival = P[ID].IOBurst[P[ID].CPUincre] + P[ID].CPUBurst[P[ID].CPUincre] + exeTime;
200     exeTime = exeTime + P[ID].CPUBurst[P[ID].CPUincre];
201     P[ID].CPUincre++;
202 }
203
204 //Checks the CPU and I/O Burst inputs to see if a process is complete.
205 void Process::check(Process P[], int ID)
206 {
207     P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst[P[ID].CPUincre] + P[ID].IOBurst[P[ID].CPUincre];
      //Updates the combined number of used CPU and IO bursts for each process.  Important for waiting time
calculation.
208
209     if(P[ID].responseCheck==0)
210     {
211         P[ID].response = exeTime;
212         P[ID].responseCheck=1;
213     }
214
215     if (P[ID].IOBurst[P[ID].CPUincre]==0)                                       //0
units for IO bursts completes the process as it recognizes the last CPU burst.
216     {
217         P[ID].complete = 1;
218         exeTime = exeTime + P[ID].CPUBurst[P[ID].CPUincre];
219         P[ID].turnaround = exeTime - P[ID].firstArrive;                         //Calculates
turnaround time based on execution time - arrival time.
220         cout << "Process #" << P[ID].num  << " is completed at " << exeTime << " units\n";    //Prints the
completion time for each process.
221         completeCount++;
222         cout << "Number of Processes complete: " << completeCount << endl;
223     }
224     else
225     {
226         setArrival(P, ID);                                                  //If the process has
not ended, update arrival time.
227     }
228     cout << "\nCurrent execution time: " << exeTime << endl;                 //Displays the current
execution time for the algorithm.
```

```
}

//Sorts the process in ascending order by CPU Burst time at the start.
void Process::firstSort(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].CPUBurst[P[j].CPUincre] > P[j+1].CPUBurst[P[j+1].CPUincre])
//Sorts processes in ascending order based on CPU Burst.
            {
                std::swap(P[j], P[j+1]);
            }
            else if (P[j].CPUBurst[P[j].CPUincre] == P[j+1].CPUBurst[P[j+1].CPUincre])                //In
a situation where two processes have the same CPU Burst and 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]);
                }
                else if (P[j].arrival > P[j+1].arrival)
                {
                    std::swap(P[j], P[j+1]);
                }
            }
        }
    }
}

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

//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.
```

```
290     {
291         for(int j=0; j<count-i-1; j++)
292         {
293             if(P[j].num > P[j+1].num)
294             {
295                 std::swap(P[j], P[j+1]);
296             }
297         }
298     }
299 }
300 
301 //Assigns the next minimum CPU Burst time of the updated class array.
302 void Process::getNextBurst(Process P[], int count)
303 {
304     for(int i=0; i<count; i++)
305     {
306         if(P[i].complete != 1)                                  //If a process is not complete, proceed.
307         {
308             minBurst = P[i].CPUBurst[P[i].CPUincre];
309         }
310         else                                                    //If completed then skip element.
311         {
312             continue;
313         }
314     }
315 
316     for(int i=0; i<count; i++)
317     {
318         if(P[i].arrival <= exeTime)
319         {
320             if(P[i].CPUBurst[P[i].CPUincre] < minBurst)
321             {
322                 if(P[i].complete != 1)
323                 {
324                     minBurst = P[i].CPUBurst[P[i].CPUincre];
325                 }
326                 else
327                 {
328                     continue;               //if process is completed, skip the element.
329                 }
330             }
331             else
332             {
333                 continue;
334             }
335         }
336         else
337         {
338             continue;
339         }
340     }
341 }
342 
343 //Check if the process is in the wait queue or waiting for execution time.
344 void Process::waitCheck(Process P[], int count)
345 {
346     for(int i=0; i<count; i++)                                  //Scans the list to check which processes are
in ready queue or not.
347     {
348         if(exeTime < P[i].arrival)
349         {
350             P[i].waitQueue=1;                                   //Goes into Waiting queue.
351         }
352         else
353         {
354             P[i].waitQueue=0;                                   //Returns to Ready queue.
```

```
355         }
356     }
357 }
358 
359 //Checks if there are no processes in the ready queue.
360 void Process::idleCheck(Process P[], int count)
361 {
362     int waitCount=0, cCount=0;
363     for(int i=0; i<count; i++)
364     {
365         if(P[i].complete != 1)
366         {
367             if(P[i].waitQueue==1)
368             {
369                 waitCount++;                        //Counts which processes are in waiting queue.
370             }
371             else
372             {
373                 continue;
374             }
375         }
376         else
377         {
378             cCount++;                           //Counts which processes are completed.
379         }
380     }
381 
382     if(waitCount == (8-cCount))                     //Remaining processes that are not in ready queue,
and the algorithm goes into idle.
383     {
384         exeTime++;
385         idleTime++;
386         cout << "The Algorithm is Idle at execution time: " << exeTime << " units.\n";
387     }
388 }
389 
390 //Prints response time for each process and calculates average response time.
391 void Process::calculateResponse(Process P[], int count)
392 {
393     double TotalResponse=0;
394     cout << "\n";
395     for(int i=0; i<count; i++)
396     {
397         cout << "Response time for Process #" << P[i].num << ": " << P[i].response << "\n";
398         TotalResponse = TotalResponse + P[i].response;
399     }
400     AVGresponse = TotalResponse/8;
401 }
402 
403 //Prints turnaround time for each process and calculates average turnaround time.
404 void Process::calculateTurnaround(Process P[], int count)
405 {
406     double TotalTurnaround=0;
407     cout << "\n";
408     for(int i=0; i<count; i++)
409     {
410         cout << "Turnaround time for Process #" << P[i].num << ": " << P[i].turnaround << "\n";
411         TotalTurnaround = TotalTurnaround + P[i].turnaround;
412     }
413     AVGturnaround = TotalTurnaround/8;
414 }
415 
416 //Prints waiting time for each process and calculates average waiting time.
417 void Process::calculateWaiting(Process P[], int count)
418 {
419     double TotalWaiting=0;
```

```
420     cout << "\n";
421     for(int i=0; i<count; i++)
422     {
423         P[i].waiting = P[i].turnaround-P[i].totalBurst;
424         cout << "Waiting time for Process #" << P[i].num << ": " << P[i].waiting << "\n";
425         TotalWaiting = TotalWaiting + P[i].waiting;
426     }
427     AVGwaiting = TotalWaiting/8;
428 }
```