

```

1  /*
2  Nhat-Huy Tran
3  10-20-2023
4  COP4106
5  CPU Scheduling Programming Assignment - FCFS Algorithm
6  */
7
8  #include <iostream>
9  #include <algorithm>
10 #include <iomanip>
11
12 using namespace std;
13
14 class Process //Each object under Process will have their
own data stored. I.e. CPU Burst, Arrival time, etc.
15 {
16     public:
17         void setID(Process P[], int count); //Sets ID for each Process in the
class array
18         void MLFQ (Process P[], int count); //Main function of the algorithm
MLFQ
19         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.
20         void sortProcess (Process P[], int count); //Sorts Class array elements in
order of updated arrival times and Queues.
21         void setArrival(Process P[], int ID); //Sets and Updates the arrival
time.
22         void initiate(Process P[],int i); //Asks for user input of
processes' CPU and I/O Bursts.
23         void useRemainTime(Process P[],int i); //If a process fails to finish
before time quantum, use time quantum and have remaining time.
24         void finishRemain(Process P[],int i); //If the process has remaining
time burst finishes it, use the remaining burst.
25         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.
26         void idleCheck(Process P[], int count); //Checks if there are no
available processes in the ready queue.
27         void setResponse(Process P[], int count); //Sets Response time for each
process.
28         void IsQueue1Available(Process P[], int count); //Checks if there is a process
available in previous queue.
29         void IsQueue2Available(Process P[], int count); //Checks if there is a process
available in previous queue.
30         void sortID(Process P[], int count); //Sorts Class array elements in
order of Process ID for results output.
31         void CompComplete(Process P[], int ID); //Second part of the check
function. Checks if the process is complete.
32         void calculateTurnaround(Process P[], int count); //Calculates turnaround time for
each process and average.
33         void calculateWaiting(Process P[], int count); //Calculates Waiting time for
each process and average.
34         void calculateResponse(Process P[], int count); //Gets response time for each
process and average.
35         void endProcess(Process P[], int count); //Ends the loop and displays the
results after all processes in the array are completed.
36     private:
37         int CPUBurst;
38         int IOBurst;
39         int arrival=0;
40         int num;
41         int firstArrive=0;
42         int totalBurst=0;
43         int complete=0;
44         int waitQueue=0;
45         int Queue=1;
46         int remain=0;

```

```

47     int waiting;
48     int turnaround;
49     int response;
50 };
51
52 int minBurst=0; //Smallest burst in the class array.
53 int Queue1Ava=0; //Checks if there is a process available in
Queue 1 that are ready to be processed.
54 int Queue2Ava=0; //Checks if there is a process available in
Queue 2 that are ready to be processed.
55 int completeCount=0; //Keeps track of how many process are
completed.
56 int Queue1Count=8, Queue2Count=0, Queue3Count=0; //Checks and moves each process to a
certain queue based on conditions
57 int exeTime=0, idleTime=0; //Execution time and Idle time for
algorithm.
58 int responseCheck=0; //Checks if the response times are
fulfilled.
59 double AVGresponse, AVGTurnaround, AVGwaiting; //Averages of Turnaround, Waiting, and
Response time.
60
61 int main()
62 {
63     cout << "Welcome to the Multilevel Feedback Queue (MLFQ) Simulation!\n\n";
64     Process P[8]; //Initializes class array with 8 processes.
65     P[8].setID(P, 8); //Sets ID for Processes in the class array.
66     P[8].MLFQ(P, 8); //Initiates the algorithm.
67 }
68
69 void Process::endProcess(Process P[], int count) //Ends the loop if all processes are
completed.
70 {
71     if (completeCount == 8) //If all processes are completed, Prints
out results for turnaround, waiting, response, and CPU Utilization
72     {
73         cout << "RESULTS:\n";
74         cout << "_____ \n";
75         cout << "\nTotal Execution time: " << exeTime << " units.\n"; //Prints Total
execution time to complete the algorithm.
76         cout << "Total Idle time: " << idleTime << " units.\n"; //Prints Total idle
time between processes.
77
78         double AVGcpu = exeTime-idleTime; //Calculates CPU
Utilization in the program.
79         AVGcpu = AVGcpu/double(exeTime);
80         AVGcpu = AVGcpu*100;
81
82         sortID(P, 8); //Sorts Processes
in ascending order based on Process ID.
83
84         cout << "CPU Utilization: " << fixed << setprecision(2) << AVGcpu << "%\n";
85
86         cout << "\nTurnaround time results:\n" << "_____ \n";
87         calculateTurnaround(P,8);
//Prints turnaround time for each process and average turnaround time.
88         cout << "\nAverage turnaround time: " << fixed << setprecision(2) << AVGTurnaround << "\n";
89
90         cout << "\nWaiting time results:\n" << "_____ \n";
91         calculateWaiting(P,8);
//Prints waiting time for each process and average waiting time.
92         cout << "\nAverage waiting time: " << fixed << setprecision(2) << AVGwaiting << "\n";
93
94         cout << "\nResponse time results:\n" << "_____ \n";
95         calculateResponse(P,8);
//Prints response time for each process and average response time.
96         cout << "\nAverage response time: " << fixed << setprecision(2) << AVGresponse << "\n";

```

```

97
98         exit(1); //Ends
program.
99     }
100     else
101     {
102         cout << "Not complete yet.\n\n";
103     }
104 }
105
106 //Sets ID for Processes in the class array
107 void Process::setID(Process P[], int count)
108 {
109     for(int i=0; i<count; i++)
110     {
111         P[i].num = i+1;
112     }
113 }
114
115 //Main function of the algorithm MLFQ
116 void Process::MLFQ (Process P[], int count)
117 {
118     for(int i=0; i<count; i++)
119     {
120         if(P[i].complete != 1)
121         {
122             idleCheck(P,8);
123             waitCheck(P,8);
124             if(P[i].waitQueue==0)
125             {
126                 if((P[i].arrival <= minBurst) || (P[i].arrival <= exeTime))
127                 {
128
129                     if (responseCheck==1)
130                     {
131                         sortProcess(P, i); //Sorts list based on arrival time and
132                                         queues.
133                     }
134
135                     if(P[i].Queue==1)
136                     {
137                         initiate(P, i);
138                     }
139                     else if (P[i].Queue == 2)
140                     {
141                         IsQueue1Available(P,8);
142                         if(Queue1Ava==0)
143                         {
144                             if(P[i].remain == 0)
145                             {
146                                 initiate(P, i);
147                             }
148                             else
149                             {
150                                 useRemainTime(P,i);
151                             }
152                         }
153                         else
154                         {
155                             Queue1Ava=0;
156                             continue;
157                         }
158                     }
159                     else
160                     {
161                         IsQueue1Available(P,8);

```

```

161         if(Queue1Ava==0)
162         {
163             IsQueue2Available(P,8);
164             if(Queue2Ava==0)
165             {
166                 if(P[i].remain == 0)
167                 {
168                     initiate(P, i);
169                 }
170                 else
171                 {
172                     useRemainTime(P,i);
173                 }
174             }
175             else
176             {
177                 Queue2Ava=0;
178                 continue;
179             }
180         }
181         else
182         {
183             Queue1Ava=0;
184             continue;
185         }
186     }
187 }
188 else
189 {
190     continue;
191 }
192 }
193 else
194 {
195     continue;
196 }
197 }
198 else
199 {
200     continue;
201 }
202 }
203
204 if (responseCheck==0)
205 {
206     setResponse(P, 8);           //Sets Response time for each process.
207     sortProcess(P, 8);           //Sorts the Process class array based on
208 }                                arrival time and Queue.
209
210 P[8].MLFQ(P,8);
211 }
212
213 //Sets Response time for each process.
214 void Process::setResponse(Process P[], int count)
215 {
216     int exe=0;
217     for (int i=0; i<count; i++)
218     {
219         if((P[i].Queue == 1)&&(P[i].remain ==0))
220         {
221             P[i].response = exe;
222             exe = exe + P[i].CPUBurst;
223         }
224         else
225         {

```

```

226         P[i].response = exe;
227         exe = exe + 5;
228     }
229 }
230 responseCheck=1; //Checks that response time has been recorded.
231 }
232
233 //Asks for user input of processes' CPU and I/O Bursts.
234 void Process::initiate (Process P[],int i)
235 {
236     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.
237     cout << "Enter the CPU Burst time for Process #" << P[i].num << ": ";
//Reads input CPU Burst.
238     cin >> P[i].CPUBurst;
239     cout << "Enter the IO Burst time for Process #" << P[i].num << ": ";
//Reads input IO Burst.
240     cin >> P[i].IOBurst;
241     check(P,i);
242     cout << "\n";
243     endProcess(P, 8);
244 }
245
246 //If a process fails to finish before time quantum, use time quantum and have remaining time.
247 void Process::useRemainTime(Process P[],int i)
248 {
249     cout << "Process #" << P[i].num << " is set to arrive at " << P[i].arrival << " in Queue: " << P[i].
Queue << endl;
250     cout << "Process #" << P[i].num << " has a remaining time of " << P[i].remain << endl;
251     check(P,i);
252 }
253
254 //If the process has remaining time burst finishes it, use the remaining burst.
255 void Process::finishRemain(Process P[],int ID)
256 {
257     P[ID].arrival = P[ID].IOBurst + P[ID].remain + exeTime;
258     P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst + P[ID].IOBurst;
259     exeTime = exeTime + P[ID].remain;
260     P[ID].remain = 0;
261 }
262
263 //Sets and Updates the arrival time.
264 void Process::setArrival(Process P[], int ID)
265 {
266     P[ID].arrival = P[ID].IOBurst + P[ID].CPUBurst + exeTime;
267     exeTime = exeTime + P[ID].CPUBurst;
268 }
269
270 //Second part of the check function. Checks if the process is complete.
271 void Process::CompComplete(Process P[], int ID)
272 {
273     P[ID].totalBurst = P[ID].totalBurst + P[ID].CPUBurst + P[ID].IOBurst;
274     if (P[ID].IOBurst==0) //0 units for IO bursts
completes the process as it recognizes the last CPU burst.
275     {
276         P[ID].complete=1;
277         exeTime = exeTime + P[ID].CPUBurst;
278         P[ID].turnaround = exeTime - P[ID].firstArrive; //Calculates
turnaround time based on execution time - arrival time.
279         cout << "Process #" << P[ID].num << " is completed at " << exeTime << " units\n"; //Prints the
completion time for each process.
280         completeCount++; //Increments to
how many processes completed.
281         cout << "Number of Processes complete: " << completeCount << endl;
282         P[ID].arrival = P[ID].IOBurst + P[ID].remain + exeTime + 700;
283     }

```

```

284     else
285     {
286         setArrival(P, ID);
287     }
288 }
289
290 //Checks the CPU and I/O Burst inputs to see if a process is either complete, expires on time quantum or
has remaining time.
291 void Process::check(Process P[], int ID)
292 {
293     if (P[ID].Queue == 1)
294     {
295         if(P[ID].CPUBurst > 5)
296         {
297             exeTime = exeTime + 5;
298             P[ID].remain = P[ID].CPUBurst-5;
299             P[ID].Queue = 2;
300             cout << "Process #" << P[ID].num << " has moved down to Queue 2.\n";
301             P[ID].arrival = exeTime + 5;
302             cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
303             Queue1Count--;
304             Queue2Count++;
305         }
306         else
307         {
308             CompComplete(P, ID);
309         }
310     }
311     else if (P[ID].Queue == 2)
312     {
313         if (P[ID].remain!=0)
314         {
315             if(P[ID].remain > 10)
316             {
317                 exeTime = exeTime + 10;
318                 P[ID].remain = P[ID].remain-10;
319                 P[ID].Queue = 3;
320                 cout << "Process #" << P[ID].num << " has moved down to Queue 3.\n";
321                 P[ID].arrival = exeTime + 10;
322                 cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
323                 Queue2Count--;
324                 Queue3Count++;
325             }
326             else
327             {
328                 finishRemain(P, ID);
329             }
330         }
331         else if(P[ID].CPUBurst > 10)
332         {
333             exeTime = exeTime + 10;
334             P[ID].remain = P[ID].CPUBurst-10;
335             P[ID].Queue = 3;
336             cout << "Process #" << P[ID].num << " has moved down to Queue 3.\n";
337             P[ID].arrival = exeTime + 10;
338             cout << "Remaining time for Process #" << P[ID].num << ": " << P[ID].remain << "\n";
339             Queue2Count--;
340             Queue3Count++;
341         }
342         else
343         {
344             CompComplete(P, ID);
345         }
346     }
347     else
348     {

```

```

349         if(P[ID].remain!=0)
350         {
351             finishRemain(P, ID);
352         }
353         else
354         {
355             CompComplete(P, ID);
356         }
357     }
358     cout << "\nCurrent execution time: " << exeTime << endl;
359 }
360
361 //Sorts Class array elements in order of updated arrival times and Queues.
362 void Process::sortProcess(Process P[], int count)
363 {
364     for(int i=0; i<count; i++) //Bubble sort for the class array.
365     {
366         for(int j=0; j<count-i-1; j++)
367         {
368             if((P[j].arrival > P[j+1].arrival)&&(P[j].Queue > P[j+1].Queue)) //Sorts
369             { //processes in ascending order based on arrival time and priority queue.
370                 std::swap(P[j], P[j+1]);
371             }
372             else if ((P[j].arrival > P[j+1].arrival)&&(P[j].Queue == P[j+1].Queue))
373             {
374                 std::swap(P[j], P[j+1]);
375             }
376             else if ((P[j].arrival == P[j+1].arrival)&&(P[j].Queue == P[j+1].Queue)) //In a
377             { //situation where two processes have the same arrival time and queue, the class array is sorted based on process
378                 if(P[j].num > P[j+1].num)
379                 {
380                     std::swap(P[j], P[j+1]);
381                 }
382             }
383         }
384     }
385 }
386
387 //Check if the process is in the wait queue or waiting for execution time.
388 void Process::waitCheck(Process P[], int count)
389 {
390     for(int i=0; i<count; i++) //Scans the list to check which processes are
391     { //in ready queue or not.
392         if(exeTime < P[i].arrival)
393         {
394             P[i].waitQueue=1; //Goes into Waiting queue.
395         }
396         else
397         {
398             P[i].waitQueue=0; //Returns to Ready queue.
399         }
400     }
401 }
402
403 //Checks if there is a process available in previous queue.
404 void Process::IsQueue1Available(Process P[], int count)
405 {
406     for(int i=0; i<count; i++)
407     {
408         if((P[i].Queue==1)&&(P[i].arrival <= exeTime))
409         {
410             Queue1Ava=1;

```

```

411     }
412 }
413 }
414
415 //Checks if there is a process available in previous queue.
416 void Process::IsQueue2Available(Process P[], int count)
417 {
418     for(int i=0; i<count; i++)
419     {
420         if((P[i].Queue==2)&&(P[i].arrival <= exeTime))
421         {
422             Queue2Ava=1;
423         }
424     }
425 }
426
427 //Checks if there are no processes in the ready queue.
428 void Process::idleCheck(Process P[], int count)
429 {
430     int waitCount=0, cCount=0;
431     for(int i=0; i<count; i++)
432     {
433         if(P[i].complete != 1)
434         {
435             if(P[i].waitQueue==1)
436             {
437                 waitCount++; //Counts which processes are in waiting queue.
438             }
439             else
440             {
441                 continue;
442             }
443         }
444         else
445         {
446             cCount++; //Counts which processes are completed.
447         }
448     }
449
450     if(waitCount == (8-cCount)) //Remaining processes that are not in ready queue,
and the algorithm goes into idle.
451     {
452         exeTime++;
453         idleTime++;
454         cout << "The Algorithm is Idle at execution time: " << exeTime << " units.\n";
455     }
456 }
457
458 //Sorts Class array elements in order of Process ID for results output.
459 void Process::sortID(Process P[], int count)
460 {
461     for(int i=0; i<count; i++) //Bubble sort for sorting class array.
462     {
463         for(int j=0; j<count-i-1; j++)
464         {
465             if(P[j].num > P[j+1].num)
466             {
467                 std::swap(P[j], P[j+1]);
468             }
469         }
470     }
471 }
472
473 //Prints response time for each process and calculates average response time.
474 void Process::calculateResponse(Process P[], int count)
475 {

```



```

476     double TotalResponse=0;
477     cout << "\n";
478     for(int i=0; i<count; i++)
479     {
480         cout << "Response time for Process #" << P[i].num << ": " << P[i].response << "\n";
481         TotalResponse = TotalResponse + P[i].response;
482     }
483     AVGresponse = TotalResponse/8;
484 }
485
486 //Prints turnaround time for each process and calculates average turnaround time.
487 void Process::calculateTurnaround(Process P[], int count)
488 {
489     double TotalTurnaround=0;
490     cout << "\n";
491     for(int i=0; i<count; i++)
492     {
493         cout << "Turnaround time for Process #" << P[i].num << ": " << P[i].turnaround << "\n";
494         TotalTurnaround = TotalTurnaround + P[i].turnaround;
495     }
496     AVGTturnaround = TotalTurnaround/8;
497 }
498
499 //Prints waiting time for each process and calculates average waiting time.
500 void Process::calculateWaiting(Process P[], int count)
501 {
502     double TotalWaiting=0;
503     cout << "\n";
504     for(int i=0; i<count; i++)
505     {
506         P[i].waiting = P[i].turnaround-P[i].totalBurst;
507         cout << "Waiting time for Process #" << P[i].num << ": " << P[i].waiting << "\n";
508         TotalWaiting = TotalWaiting + P[i].waiting;
509     }
510     AVGwaiting = TotalWaiting/8;
511 }

```