```c
/**
 * Family Name:
 * Given Name:
 * Section:
 * Student Number:
 * CSE Login:
 *
 * Description: First-come-first-serve (quad-core) CPU scheduling simulator.
 */

/* WARNING: This code is used only for you to debug your own FCFS scheduler. You should NOT use this as a template for the remaining two schedulers. Otherwise, your codes may not pass plagiarism tests. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#include "sch-helpers.h"    /* include header file of all helper functions */

/* Declare some global variables and structs to be used by FCFS scheduler */

process processes[MAX_PROCESSES+1];   /* a large array to hold all processes read from data file */
                                      /* these processes are pre-sorted and ordered by arrival time */
int numberOfProcesses;                /* total number of processes */
int nextProcess;                      /* index of next process to arrive */
process_queue readyQueue;             /* ready queue to hold all ready processes */
process_queue waitingQueue;           /* waiting queue to hold all processes in I/O waiting */
process *cpus[NUMBER_OF_PROCESSORS];  /* processes running on each cpu */
int totalWaitingTime;                 /* total time processes spent waiting */
int totalContextSwitches;             /* total number of preemptions */
int simulationTime;                   /* time steps simulated */
int cpuTimeUtilized;                  /* time steps each cpu was executing */

/* holds processes moving to the ready queue this timestep (for sorting) */
process *preReadyQueue[MAX_PROCESSES];
int preReadyQueueSize;

/** Initialization functions **/

/* performs basic initialization on all global variables */
void initializeGlobals(void) {
    int i = 0;
    for (;i<NUMBER_OF_PROCESSORS;i++) {
        cpus[i] = NULL;
    }

    simulationTime = 0;
    cpuTimeUtilized = 0;
    totalWaitingTime = 0;
    totalContextSwitches = 0;
    numberOfProcesses = 0;
    nextProcess = 0;

```

```
55      preReadyQueueSize = 0;
56
57      initializeProcessQueue(&readyQueue);
58      initializeProcessQueue(&waitingQueue);
59  }
60
61  /** FCFS scheduler simulation functions **/
62
63  /* compares processes pointed to by *aa and *bb by process id,
64     returning -1 if aa < bb, 1 if aa > bb and 0 otherwise. */
65  int compareProcessPointers(const void *aa, const void *bb) {
66      process *a = *((process**) aa);
67      process *b = *((process**) bb);
68      if (a->pid < b->pid) return -1;
69      if (a->pid > b->pid) return 1;
70      assert(0); /* case should never happen, o/w ambiguity exists */
71      return 0;
72  }
73
74  /* returns the number of processes currently executing on processors */
75  int runningProcesses(void) {
76      int result = 0;
77      int i;
78      for (i=0;i<NUMBER_OF_PROCESSORS;i++) {
79          if (cpus[i] != NULL) result++;
80      }
81      return result;
82  }
83
84  /* returns the number of processes that have yet to arrive in the system */
85  int incomingProcesses(void) {
86      return numberOfProcesses - nextProcess;
87  }
88
89  /* simulates the CPU scheduler, fetching and dequeuing the next scheduled
90     process from the ready queue.  it then returns a pointer to this process,
91     or NULL if no suitable next process exists. */
92  process *nextScheduledProcess(void) {
93      if (readyQueue.size == 0) return NULL;
94      process *result = readyQueue.front->data;
95      dequeueProcess(&readyQueue);
96      return result;
97  }
98
99  /* enqueue newly arriving processes in the ready queue */
100 void moveIncomingProcesses(void) {
101
102     /* place newly arriving processes into an intermediate array
103        so that they will be sorted by priority and added to the ready queue */
104     while (nextProcess < numberOfProcesses &&
105            processes[nextProcess].arrivalTime <= simulationTime) {
106         preReadyQueue[preReadyQueueSize++] = &processes[nextProcess++];
107     }
108 }
109
110 /* move any waiting processes that are finished their I/O bursts to ready */
111 void moveWaitingProcesses(void) {
112     int i;
113     int size = waitingQueue.size;
```

```
114
115     /* place processes finished their I/O bursts into an intermediate array
116        so that they will be sorted by priority and added to the ready queue */
117     for (i=0;i<size;i++) {
118         process *front = waitingQueue.front->data; /* get process at front */
119         dequeueProcess(&waitingQueue);             /* dequeue it */
120
121         assert(front->bursts[front->currentBurst].step <=
122                front->bursts[front->currentBurst].length);
123
124         /* if process' current (I/O) burst is finished,
125            move it to the ready queue, else return it to the waiting queue */
126         if (front->bursts[front->currentBurst].step ==
127             front->bursts[front->currentBurst].length) {
128
129             /* switch to next (CPU) burst and place in ready queue */
130             front->currentBurst++;
131             preReadyQueue[preReadyQueueSize++] = front;
132         } else {
133             enqueueProcess(&waitingQueue, front);
134         }
135     }
136 }
137
138 /* move ready processes into free cpus according to scheduling algorithm */
139 void moveReadyProcesses(void) {
140     int i;
141
142     /* sort processes in the intermediate preReadyQueue array by priority,
143        and add them to the ready queue prior to moving ready procs. into CPUs */
144     qsort(preReadyQueue, preReadyQueueSize, sizeof(process*),
145           compareProcessPointers);
146     for (i=0;i<preReadyQueueSize;i++) {
147         enqueueProcess(&readyQueue, preReadyQueue[i]);
148     }
149     preReadyQueueSize = 0;
150
151     /* for each idle cpu, load and begin executing
152        the next scheduled process from the ready queue. */
153     for (i=0;i<NUMBER_OF_PROCESSORS;i++) {
154         if (cpus[i] == NULL) {
155             cpus[i] = nextScheduledProcess();
156         }
157     }
158 }
159
160 /* move any running processes that have finished their CPU burst to waiting,
161    and terminate those that have finished their last CPU burst. */
162 void moveRunningProcesses(void) {
163     int i;
164     for (i=0;i<NUMBER_OF_PROCESSORS;i++) {
165         if (cpus[i] != NULL) {
166             /* if process' current (CPU) burst is finished */
167             if (cpus[i]->bursts[cpus[i]->currentBurst].step ==
168                 cpus[i]->bursts[cpus[i]->currentBurst].length) {
169
170                 /* start process' next (I/O) burst */
171                 cpus[i]->currentBurst++;
172
```

```
                /* move process to waiting queue if it is not finished */
                if (cpus[i]->currentBurst < cpus[i]->numberOfBursts) {
                    enqueueProcess(&waitingQueue, cpus[i]);

                /* otherwise, terminate it (don't put it back in the queue) */
                } else {
                    cpus[i]->endTime = simulationTime;
                }

                /* stop executing the process
                   -since this will remove the process from the cpu immediately,
                    but the process is supposed to stop running at the END of
                    the current time step, we need to add 1 to the runtime */
                cpus[i] = NULL;
            }
        }
    }
}

/* increment each waiting process' current I/O burst's progress */
void updateWaitingProcesses(void) {
    int i;
    int size = waitingQueue.size;
    for (i=0;i<size;i++) {
        process *front = waitingQueue.front->data; /* get process at front */
        dequeueProcess(&waitingQueue);             /* dequeue it */

        /* increment the current (I/O) burst's step (progress) */
        front->bursts[front->currentBurst].step++;
        enqueueProcess(&waitingQueue, front);      /* enqueue it again */
    }
}

/* increment waiting time for each process in the ready queue */
void updateReadyProcesses(void) {
    int i;
    for (i=0;i<readyQueue.size;i++) {
        process *front = readyQueue.front->data; /* get process at front */
        dequeueProcess(&readyQueue);             /* dequeue it */
        front->waitingTime++;                    /* increment waiting time */
        enqueueProcess(&readyQueue, front);      /* enqueue it again */
    }
}

/* update the progress for all currently executing processes */
void updateRunningProcesses(void) {
    int i;
    for (i=0;i<NUMBER_OF_PROCESSORS;i++) {
        if (cpus[i] != NULL) {
            /* increment the current (CPU) burst's step (progress) */
            cpus[i]->bursts[cpus[i]->currentBurst].step++;
        }
    }
}

int main(void) {
    int sumOfTurnaroundTimes = 0;
    int doneReading = 0;
    int i;
```

```
232
233       /* read in all process data and populate processes array with the results */
234       initializeGlobals();
235       while (doneReading=readProcess(&processes[numberOfProcesses]))  {
236          if(doneReading==1)  numberOfProcesses ++;
237          if(numberOfProcesses > MAX_PROCESSES) break;
238       }
239
240       /* handle invalid number of processes in input */
241       if (numberOfProcesses == 0) {
242           fprintf(stderr, "Error: no processes specified in input.\n");
243           return -1;
244       } else if (numberOfProcesses > MAX_PROCESSES) {
245           fprintf(stderr, "Error: too many processes specified in input; "
246                           "they cannot number more than %d.\n", MAX_PROCESSES);
247           return -1;
248       }
249
250       /* sort the processes array ascending by arrival time */
251       qsort(processes, numberOfProcesses, sizeof(process), compareByArrival);
252
253       /* run the simulation */
254       while (1) {
255           moveIncomingProcesses();  /* admit any newly arriving processes */
256           moveRunningProcesses();   /* move procs that shouldn't be running */
257           moveWaitingProcesses();   /* move procs finished waiting to ready-Q */
258           moveReadyProcesses();     /* move ready procs into any free cpu slots */
259
260           updateWaitingProcesses(); /* update burst progress for waiting procs */
261           updateReadyProcesses();   /* update waiting time for ready procs */
262           updateRunningProcesses(); /* update burst progress for running procs */
263
264           cpuTimeUtilized += runningProcesses();
265
266           /* terminate simulation when:
267               - no processes are running
268               - no more processes await entry into the system
269               - there are no waiting processes
270           */
271           if (runningProcesses() == 0 &&
272               incomingProcesses() == 0 &&
273               waitingQueue.size == 0) break;
274
275           simulationTime++;
276       }
277
278       /* compute and output performance metrics */
279       for (i=0;i<numberOfProcesses;i++) {
280           sumOfTurnaroundTimes += processes[i].endTime - processes[i].arrivalTime;
281           totalWaitingTime += processes[i].waitingTime;
282       }
283
284       printf("Average waiting time                : %.2f units\n"
285              "Average turnaround time             : %.2f units\n"
286              "Time all processes finished         : %d\n"
287              "Average CPU utilization             : %.1f%%\n"
288              "Number of context switches          : %d\n",
289              totalWaitingTime / (double) numberOfProcesses,
290              sumOfTurnaroundTimes / (double) numberOfProcesses,
```

```
            simulationTime,
            100.0 * cpuTimeUtilized / simulationTime,
            totalContextSwitches);

    printf("PID(s) of last process(es) to finish :");
    for (i=0;i<numberOfProcesses;i++) {
        if (processes[i].endTime == simulationTime) {
            printf(" %d", processes[i].pid);
        }
    }
    printf("\n");
    return 0;
}

```