```c
/* Program to simulate a scheduler for an operating system.
 * Since details of the scheduler are placed in an include file,
 * this program may be used with different scheduling algorithms.
 *
 * The main framework for this lab follows Lab Exercise 7.1 in Nutt.
 * The generalization to multiple scheduling algorithms, however,
 * requires some adjustments.
 *
 * Framework created by Henry M. Walker on 27 September 2004
 * Revised by Janet Davis, 25 September 2010
 * Revised by Jerod Weinman, 10 August 2012
 * Revised by Jerod Weinman, 7 August 2014
 *
 * Portions of the function simulate_job that differ from the starter code at
 *   http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/code/
 *                                scheduling/scheduler_simulation.c
 * are written by
 *   YOUR NAME(S) HERE
 */

/* debugging flags (uncomment or use with gcc option -Dflag) */
// #define D_INPUT         /* print input as it is read from the file */
//#define D_EVENTLIST     /*  print event list in main simulation loop */
//#define D_PRINTSTATS    /*  print times in main simulation loop */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "scheduler.h"
#include "eventq.h"
#include "stats.h"

/* QUANTUM
 * any positive number represents the time quantum for a preemptive
 * scheduling algorithm
 * 0 indicates a nonpreemptive scheduling algorithm
 */
#ifndef QUANTUM
#define QUANTUM 0.0
#endif

/* OVERHEAD is the simulated time required for each call to the
 * scheduler */
#ifndef OVERHEAD
#define OVERHEAD 0.35
#endif

/* specify file listing the jobs to simulate */
#define JOB_FILE_NAME "scheduler_job_data.txt"

/* helpers for enqueueing events */
void load_jobs(void);
void run_scheduler(void);

/* event handler prototypes */
void register_job(job_t* job);
void simulate_job(job_t* job);
void scheduler(job_t* job);

/* global variables */
double sim_time = 0.0;          /* the clock for this simulation */

/* The main() function initializes the event list, job queue, and
 * statistics.  It then enters the main event handling loop.  When there
 * are no more events, it prints out the final stats.
 */
int main( void ) {
  printf( "Beginning simulation\n" );
  printf( "Scheduler overhead: %3.2f\n", OVERHEAD );
  printf( "Time quantum:       %3.2f\n", QUANTUM );

  ready_queue_init();
  stats_init();
  eventq_init();
  load_jobs();

  /* main event loop */
  while (!eventq_empty()) {
    #ifdef D_EVENTLIST
      eventq_print();
    #endif

    eventq_next();

    #ifdef D_PRINTSTATS
      printf("accumulated times:\n");
      stats_print(sim_time);
    #endif
  }

  /* print summary of performance data */
  printf ("Simulation completed\n");
  printf ("Summary statistics:\n");
  stats_print(sim_time);
  return( 0 );
}

/* read jobs for the simulation from a file.
 * Preconditions:
 *   JOB_FILE_NAME is defined
 *   The file named contains a list of jobs where each line gives:
 *     arrival_time duration priority
 * Postconditions:
 *   All jobs listed in JOB_FILE_NAME are on the event queue
 *   The event queue includes an event to run the scheduler at time 0
 *   The event queue is sorted by event arrival time
 */
void load_jobs() {
  FILE *job_file;
  job_t*job;
  int arrival_time;
  float duration;
  int priority;

  printf ("reading job list from file:  \"%s\"\n", JOB_FILE_NAME);
  job_file = fopen (JOB_FILE_NAME, "r");

  if (!job_file) { /* Check for file open failure */
    perror("Unable to open job file");
    exit(EXIT_FAILURE);
  }

  while( fscanf(job_file, "%d %f %d", &arrival_time, &duration, &priority)
         != EOF ){
    /* first create event for beginning job */
    job = (job_t*) malloc(sizeof(job_t));

    if (!job) { /* verify job creation */
      perror("Unable to allocate job");
      exit(EXIT_FAILURE);
    }

    job->cpu_time = duration;
    job->cpu_time_left = duration;
    job->arrival_time = arrival_time;
    job->priority = priority;
    job->has_started = 0;
```

```c
    eventq_enqueue(arrival_time, "new job", &register_job, job);

#ifdef D_INPUT
    printf ("reading file: \tarrival: %d, \tduration:  %8.2f, \tpriority:  %d\n",
            arrival_time, duration, priority);
#endif
  }

  if (ferror(job_file)) { /* Handle any read problems */
    perror("Error reading job file");
    exit(EXIT_FAILURE);
  }

  /* Start the simulation. */
  run_scheduler();
}

/* Command to run the scheduler by enqueuing the scheduler event at
   the current simulation time. */
void run_scheduler(void) {
  eventq_enqueue(sim_time, "scheduler", &scheduler, NULL);
}

/* Insert the given job into the ready queue
   (i.e., according to the current policy) */
void register_job(job_t* job) {
  ready_queue_insert(job);
}

/* Run the simulation of a given job */
void simulate_job(job_t* job) {
  double quantum;
  int counter = 0;


#ifndef NUM_PRIORITY_LEVELS
  quantum = QUANTUM;
#else
  quantum = (QUANTUM / (pow(2, (job->priority - 1))));
  /* YOUR CODE HERE */
  /* variable quantum used for a MLQ scheduler is assigned here */
  /* this section to be completed in step D1 of the scheduling lab */
#endif

  if (quantum > 0) {
    if (quantum >= job->cpu_time_left) {
      /* job will finish in this time slice */
       /* advance the simulation time */
      /* YOUR CODE HERE */

      /* compilation of statistics goes here */
      //job has not started
      if ( !job->has_started){
        job->has_started = 1;
        stats.jobs_started++;
        stats.total_wait_time += sim_time - job->arrival_time;
        counter++;
      }
       sim_time += job->cpu_time_left;

        stats.jobs_completed++;
        stats.total_proc_time += job->cpu_time_left;
        stats.total_turnaround_time += sim_time - job->arrival_time;
      /* job struct freed from memory */
        free(job);

      /* this section to be completed in step C1 of the scheduling lab */
    }
    else {
      /* job will require an additional time slice */

      /* YOUR CODE HERE */

      /* this section to be completed in steps C1 or D1 of the lab, as indicated */

      /* remaining job time and statistics updated (C1) */

      if(!job->has_started){
        job->has_started = 1;
        stats.jobs_started++;
        stats.total_wait_time += sim_time - job->arrival_time;
        counter++;
      }

      sim_time += quantum;
      job->cpu_time_left -= quantum;
      job->priority --;
      /* job priority updated (D1) */

      /* job returns to ready state (C1) */
      register_job( job);
    }
  } else {
    /* non-preemptive algorithm */
    /* job runs to completion */

    /* update statistics for jobs that have been started */
    stats.jobs_started++;
    stats.total_wait_time += sim_time - job->arrival_time;
    job->has_started = 1;

    /* advance the simulation time */
    sim_time += job->cpu_time_left;

    /* update statistics for jobs that have completed */
    stats.jobs_completed++;
    stats.total_proc_time += job->cpu_time_left;
    stats.total_turnaround_time += sim_time - job->arrival_time;

    /* free the job memory, as it will no longer be referenced */
    free(job);
  }

  /* after simulating the running of this job, run the scheduler again */
  run_scheduler();
}

/* select next job for execution and place it on the eventq */
void scheduler( job_t* job ) {
  /* The job parameter is ignored. */

  job_t* next_job;

  sim_time += OVERHEAD;

  next_job = ready_queue_select();
  if (next_job == NULL) {
    if (eventq_empty()) {
      /* all done! */
      return;
    } else {
      /* increment time to next meaningful event */
      sim_time = eventq_next_event_time();
      /* put the scheduler back in the simulator event queue */
      run_scheduler();
    }
  } else {
    simulate_job(next_job);
```

```
  }
}
```