```c
/* Implements a first-come, first-served scheduler.
 *
 * Created by Henry Walker, 27 September 2004
 * Last modified by Janet Davis, 25 September 2010
 * Revised by Jerod Weinman, 10 August 2012
 */

#include <stdlib.h>
#include <stdio.h>
#include "scheduler.h"

/* The ready queue */
job_queue_t ready;

/* Initializes the ready queue.  Call before any other functions. */
void ready_queue_init(void) {
  ready.first = NULL;
  ready.last  = NULL;
}

/* Returns true or false, according to whether any jobs are waiting
 * in the ready queue.
 */
int ready_queue_empty(void) {
  return (ready.first == NULL);
}

/* Adds the specified job to the ready queue.
 *
 * Preconditions:
 *   job != NULL
 * Postconditions:
 *   Creates a new node for the job
 *   job is inserted at the end of the queue
 */
void ready_queue_insert(job_t* job) {
  job_queue_node_t* node
      = (job_queue_node_t *)malloc(sizeof(job_queue_node_t));

  if (!node) {
    perror("Unable to allocate job node");
    exit(EXIT_FAILURE);
  }

  /* copy event data to new node */
  node->job = job;

  /* insert node into ready queue*/
  node->next = NULL;
  if (ready_queue_empty()) {
    ready.first = node;
    ready.last  = node;
  } else {
    ready.last->next  = node; /* add after current last */
    ready.last  = node;       /* make new node last */
  }
}

/* Removes and returns the job at the head of the ready queue.
 *
 * Postconditions:
 *   If ready_queue_empty(), returns NULL
 *   Otherwise, returns head job and frees the associated node
 */
job_t* ready_queue_select(void) {
  job_t* job;
  job_queue_node_t* old_node;

  /* if no jobs are ready, return NULL */
  if (ready_queue_empty())
    return NULL;

  /* next job is at front of queue */
  job = ready.first->job;

  /* record node at front of queue */
  old_node = ready.first;
  ready.first = ready.first->next;

  /* check if queue is -now- empty */
  if (ready_queue_empty()) {
    ready.last = NULL; /* make last pointer consistent */
  }

  /* return old front of queue to memory pool */
  free(old_node);

  return job;
}
```

```c
/****************************************************************************
Authors : Manuella,Evan      , Box 4065, <manuella@grinnell.edu>
        : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>

Date    : Fri Sep 19 22:02:11 CDT 2014

This contains the scheduler simulation for sjn.

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/scheduling.html

*****************************************************************************/


/* Citations:
 * Code cited here is part of scheduler_fcfs.c, which was written by Jerod
 * Weinman. We copied this file, and modified it. Jeord Weinman's sources
 * are listed below.
 *
 * Our additions to this code are marked.
 */




/* Implements a fastest job next scheduler.
 *
 * Created by Henry Walker, 27 September 2004
 * Last modified by Janet Davis, 25 September 2010
 * Revised by Jerod Weinman, 10 August 2012
 */

#include <stdlib.h>
#include <stdio.h>
#include "scheduler.h"

/* The ready queue */
job_queue_t ready;

/* Initializes the ready queue.  Call before any other functions. */
void ready_queue_init(void) {
  ready.first = NULL;
  ready.last  = NULL;
}

/* Returns true or false, according to whether any jobs are waiting
 * in the ready queue.
 */
int ready_queue_empty(void) {
  return (ready.first == NULL);
}

/* Adds the specified job to the ready queue.
 *
 * Preconditions:
 *   job != NULL
 * Postconditions:
 *   Creates a new node for the job
 *   job is inserted at the end of the queue
 */
void ready_queue_insert(job_t* job) {

  job_queue_node_t* node
    = (job_queue_node_t *)malloc(sizeof(job_queue_node_t));

  /* check if memory allocated */
  if (!node) {
    perror("Unable to allocate job node");
    exit(EXIT_FAILURE);
  }

  /* Addition: pointers to keep track of previous and current nodes*/
  job_queue_node_t* prev
    = (job_queue_node_t *)malloc(sizeof(job_queue_node_t));

  /* check if memory allocated */
  if (!prev) {
    perror("Unable to allocate job node");
    exit(EXIT_FAILURE);
  }

  job_queue_node_t* current
    = (job_queue_node_t *)malloc(sizeof(job_queue_node_t));

  /* check if memory allocated */
  if (!current) {
    perror("Unable to allocate job node");
    exit(EXIT_FAILURE);
  }

  /* copy event data to new node */
  node->job = job;
  node->next = NULL;
  /* Implementation of queue
   * Addition: sort queue by the amount of cpu time for each job
   */
  //if queue is empty
  if (ready_queue_empty()) {
    ready.first = node;
    ready.last  = node;
  } else //queue is not empty
    {
      //if cpu_time for new node is shortest on queue
      if(((ready.first)->job->cpu_time_left) >= node->job->cpu_time_left){

        node->next = ready.first;
        ready.first = node;
      } else
        // cpu_time for new node not shortest
        {
          prev = ready.first;
          current  = ready.first;
          // find position of insert
          while(current->next != NULL) {

            if((node->job)->cpu_time > current->job->cpu_time) {
            prev = current;
            current = current->next;
            }
          }
          /* insert node in designated position*/
          //if designated position at end of the queue
          if( prev->next == NULL){
            prev->next = node;
            node->next = NULL;
            ready.last = node; // modify end of ready queue
          } else
            //if designated position not at the end of the queue
            {
              prev->next = node;
              node->next = current;
            }
        } // else cpu time not the shortest

    } //else  queue not empty

} // end of modifications
```

```c
/* Removes and returns the job at the head of the ready queue.
 *
 * Postconditions:
 *   If ready_queue_empty(), returns NULL
 *   Otherwise, returns head job and frees the associated node
 */
job_t* ready_queue_select(void) {
  job_t* job;
  job_queue_node_t* old_node;

  /* if no jobs are ready, return NULL */
  if (ready_queue_empty())
    return NULL;

  /* next job is at front of queue */
  job = ready.first->job;

  /* record node at front of queue */
  old_node = ready.first;
  ready.first = ready.first->next;

  /* check if queue is -now- empty */
  if (ready_queue_empty()) {
    ready.last = NULL; /* make last pointer consistent */
  }

  /* return old front of queue to memory pool */
  free(old_node);

  return job;
}
```

```c
/* This documents uses code from scheduler_sfcfs
 * Parts code that was added to the original will be marked
 * citation for the original can be found below
 */
/* Implements a first-come, first-served scheduler.
 *
 * Created by Henry Walker, 27 September 2004
 * Last modified by Janet Davis, 25 September 2010
 * Revised by Jerod Weinman, 10 August 2012
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include "scheduler.h"

/* The ready queue */
job_queue_t ready;
job_queue_t priority_1;
job_queue_t priority_2;
job_queue_t priority_3;
job_queue_t priority_4;
job_queue_t priority_5;

/* Modification to original: new queues for different priorities */


/* Modification to original : array of priority queues */



/*
job_queue_t priorities [5]={ priority_1, priority_2, priority_3, priority_4, priori
ty_5 } ;

int i = 0 ;
for (i = 0 ; i < 5 ; i++)
  {
     priorities[i] = (job_queue_t *)malloc(sizeof(job_queue_t));
    priorities[i].first = NULL;
    priorities[i].last  = NULL;
  }
*/



/* Modification to original : Initialise ready and other queues  for all our other
queues */

/* Initializes the ready queue.  Call before any other functions. */

void ready_queue_init(void) {
  // initialise the ready queue
  ready.first = NULL;
  ready.last  = NULL;


  // initialise other priority queues
  priority_1.first = NULL;
  priority_1.last  = NULL;

  priority_2.first = NULL;
  priority_2.last  = NULL;

  priority_3.first = NULL;
  priority_3.last  = NULL;

  priority_4.first = NULL;
  priority_4.last  = NULL;


  priority_5.first = NULL;
  priority_5.last  = NULL;


}

/* Returns true or false, according to whether any jobs are waiting
 * in the ready queue.
 */
int ready_queue_empty(void) {
  return (ready.first == NULL);
}


/*
 * Modification to original : new method queue_priority
 *
 */

void queue_priority(job_queue_t *  priority,job_t* job) {


  job_queue_node_t* node
      = (job_queue_node_t *)malloc(sizeof(job_queue_node_t));

  if (!node) {
    perror("Unable to allocate job node");
    exit(EXIT_FAILURE);
  }

  /* copy event data to new node */
  node->job = job;

  /* insert node into ready queue*/
  node->next = NULL;
  if ((*priority).first == NULL) {
    (*priority).first = node;
    (*priority).last  = node;
  } else {
    (* priority).last->next  = node; /* add after current last */
    (* priority).last  = node;       /* make new node last */
  }
}

/* Adds the specified job to the ready queue.
 *
 * Preconditions:
 *   job != NULL
 * Postconditions:
 *   Creates a new node for the job
 *   job is inserted at the end of the queue
 */
void ready_queue_insert(job_t* job) {

  job_queue_t* priority = ( job_queue_t*) malloc(sizeof( job_queue_t));

 //get the priority of the incoming job
  int priority_val = job->priority;

  //determine which queue to add it to and then add it  to the queue

  switch (priority_val ){
  case 1:
    {
      priority = &priority_1;
      queue_priority( priority,job);
```

```c
      break;

    }
  case 2:
    {
      priority = &priority_2;
      queue_priority( priority,job);
      break;
    }
  case 3:
    {
      priority = &priority_3;
      queue_priority( priority,job);
      break;
    }

  case 4:
    {
      priority = &priority_4;
      queue_priority( priority,job);
      break;
    }

  case 5:
    {
      priority = &priority_5;
      queue_priority( priority,job);
      break;
    }
  }
}

/* Removes and returns the job at the head of the ready queue.
 *
 * Postconditions: *   If ready_queue_empty(), returns NULL
 *   Otherwise, returns head job and frees the associated node
 */
job_t* ready_queue_select(void) {



  if( priority_5.first != NULL)
    {
      ready.first = priority_1.first;
      ready.last = priority_1.last;
    }
  else if( priority_4.first != NULL)
      {
        ready.first = priority_2.first;
        ready.last = priority_2.last;
      }
  else if( priority_3.first != NULL)
        {
          ready.first = priority_3.first;
          ready.last = priority_3.last;
        }
  else if( priority_2.first != NULL)
          {
            ready.first = priority_4.first;
            ready.last = priority_4.last;
          }
  else if( priority_1.first != NULL)
      {
        ready.first = priority_5.first;
        ready.last = priority_5.last;
      }

  job_t* job;
  job_queue_node_t* old_node;
  /* if no jobs are ready, return NULL */
  if (ready_queue_empty())
    return NULL;

  /* next job is at front of queue */
  job = ready.first->job;

  /* record node at front of queue */
  old_node = ready.first;
  ready.first = ready.first->next;

  /* check if queue is -now- empty */
  if (ready_queue_empty()) {
    ready.last = NULL; /* make last pointer consistent */
  }

  /* return old front of queue to memory pool */
  free(old_node);

  return job;
}
```

```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);
```

```
  }
}
```

Algorithm : Fist Come Fisrt Served

OVERHEAD	responsetime	turnaroundtime	throughput
0.0	28.78	61.21	0.0277
0.01	28.87	61.30	0.0276
0.02	28.96	61.38	0.0276
0.03	29.04	61.47	0.0276
0.04	29.13	61.56	0.0276
0.05	29.22	61.65	0.0276

Algorithm : Shortest job next

OVERHEAD	responsetime	turnaroundtime	throughput
0.0	25.09	57.52	0.0277
0.01	25.18	57.61	0.0276
0.02	25.27	57.70	0.0276
0.03	25.36	57.79	0.0276
0.04	25.45	57.87	0.0276
0.05	25.53	57.96	0.0276

Algorithm : simple round-robin(RR), fixed QUANTUM
default QUANTUM value : 0.1

OVERHEAD	responsetime	turnaroundtime	throughput
0.0	0.24	80.69	0.0277
0.01	0.37	132.74	0.0255
0.02	0.53	186.75	0.0236
0.03	0.66	239.36	0.0220
0.04	0.84	293.20	0.0205
0.05	0.98	347.52	0.0193

Algorithm : simple round-robin(RR), fixed OVERHEAD
default OVERHEAD value : 0.03

QUANTUM	responsetime	turnaroundtime	throughput
0.05	0.54	402.20	0.0182
0.10	0.66	239.36	0.0220
0.15	0.77	186.72	0.0236
0.20	0.92	160.42	0.0245
0.25	1.00	143.94	0.0251

/***
Authors : Manuella,Evan   , Box   , <manuella@grinnell.edu>
*   : Albert Owusu-Asare , Box 4497, <owusuasa@grinnell.edu>*

Date   : Fri Sep 19 22:02:11 CDT 2014

This document contains answers to questions 1 - 9 of Part A of
the lab on Scheduling Simulation :

http://www.cs.grinnell.edu/~weinman/courses/CSC213/2014F/labs/scheduling.html

***/

PartA: Understanding the code
----

4. The macros __EVENTQ_H and __SCHEDULER_H__ are used to make sure that the contents of **"eventq.h"** and **"scheduler.h"** are not redifined when they are encountered numerous times during the compilation and linking process. This prevents us from redefining anything that we've defined before.

5
a) the ready queue

Code in**"scheduler_simulation.c"** can change the ready queue structure
because it can:
i) Initialise the queue
ii) add jobs to the queue,
iii)remove jobs from the queue

b) the event queue

Code in**"scheduler_simulation.c"** can change the event queue structure
because it can:
i) it can initialise the queue
ii) delete an event from the queue
iii) add an event to the queue

c) the performance statistic structure

Code in **"scheduler_simulation.c"** can only initialise the structure
it calls **stats_print**(**double** sim_time) however the call does not change the underlying structure.

6. Say flag -1 initially corresponds to the scheduler handler and flag -2 initially corresponds to the register_job handler,**if for** some reason later we decide to replace say register_job with some other handler, all we need is to change remap flag -2 to this new handler and map register_job with something **else**. Our code will still work as expected because its implementation is with the flag and not what lies at the other end of the flag

However flags are extra work and perhaps there is some overhead in the work done because, the code has to first try to preprocess what each flag means and then perhaps copy the underlying handler implementation whereever the flag was called. This is extra work and perhaps innefficient management of resources as we could have simply pointed to where in memory our particular handler lies instead of making copies everytime.

7. eventq_next_event_time returns the time field of the next event in our event queue.

When called in **"scheduler_simulation.c"** :
It is used to set the 'sim_time' (time **for** the simulation) by adding the **return** of 'event_next_event_time' to sim_time to have an idea of how **long** the simulation might run.

8. This simulation assumes a constant overhead per sheduler run(0.35). This might not neccessarily reflect the actual overhead and so might not accurately represent scheduler overhead. However we believe that using an approximation might give us an insight into how fast each scheduler is in relation to other schedulers- even **if** the outcome does not reflect reality.

9. Machine/system failures can have an effect on job completion time. System failures can be caused by many factores: **for** instance external such as power-shortages, overheating, and other factors such as hardware/software failures. We are not in direct control over some of these factors hence it is reasonable they were left out of our model. Even though we could try to fit such models in our simulation it might make our overall simulation model extra complex.

PART B: Non-preemptive scheduling

4.
a) Overall the average waiting times(response times) and turnaround times **for** the SJN algorithm is lower per each overhead time, than the average waiting time and turnaround times **for** the FCFS algorithm. The throughput of both the FCFS and SJN algorithm stayed relatively the same.

Conclusion:
When response time is critical to us in our design,(high I/O bound number of jobs), SJN might be a better choice. Withing a set time more jobs will be completed with the SJN.Hence overall it seems the SJN algorithm is the more effective among the two

b) Examining the data, we found no relative differences in the derivatives of response time versus overhead or turn around time versus overhead between sjn and fcf. We found no significant differences in the derivate of throughputs with respect to time (both almost 0).

PART C: Simple preemptive scheduling

4.

a)
average Wait time:

Overall the average wait time **for** the RR algorithm was lesser in magnitude per overhead than the average wait time **for** the SJN.

Based solely on response time, preemptive scheduling is more effective than non preemptive scheduling.

average Turnaround time:

Overall the average turnaround time **for** the RR algorithm was higher in magnitude per overhead than the average turnaround time **for** the SJN algorithm

Based solely on turnaround time, non preemptive scheduling is more effective than preemptive scheduling.

throughputs:
Throughputs **for** the RR algorithm were realatively lower than throughputs of the SJN algorithm.

Based solely on throughputs, non preemptive scheduling is more effective than preemptive scheduling.

b) The derivative of response time in respects to overhead stat increases linearly in round robin. This is because as we have more jobs, the time between the same job being scheduled twice equals the number of jobs * (overhead + quantom). In the SJN scheduling algorithm, overhead does not have the same amount of impact, because overhead only happens when we **switch** jobs, and SJN minimizes occurences of job switches.

There is a similiar effect on turn-around times.

Over head time takes a significanlty larger toll on round robin compared to SJN, because round robin switches jobs more often.

c. Quantom has a significant effect on round robin. As expected, it increased the response time, because it takes longer to finish a quantom, increasing the time between responses.

Increasing quantom also decreases the average turn-around time. This is because there is reduced **overhead** (from reduced context switches).

PART D

We could not work out the particulars of the implentation, but we wanted to explain our idea. Our idea was to have the code in insert_queue insert a job into a corresponding priority queue. There would be x, x being the number of priorities, number of priority queues that correspond to priorities. When queue_select is called, it would **return** the first job from the highest priority queue. Quantum would be set, and the rest of the jobs in this priority queue would run in round robin. Our implementation failed because we needed to move to queues in arrays, instead of a **switch** statement of priority queues.

```
Script started on Mon 22 Sep 2014 10:59:55 PM CDT
driscoll$ make clean all
rm -rf *.o *~ sim_fcfs sim_sjn sim_rr sim_mlq
gcc -c -Wall -o scheduler_fcfs.o scheduler_fcfs.c
gcc -c -Wall  -DOVERHEAD=0.03 -DQUANTUM=0.0 -o nonpreemptive_scheduler_simulation.o
 scheduler_simulation.c
gcc -c -Wall -o eventq.o eventq.c
gcc -c -Wall -o stats.o stats.c
gcc -lm -o sim_fcfs scheduler_fcfs.o nonpreemptive_scheduler_simulation.o eventq.o
stats.o
gcc -c -Wall -o scheduler_sjn.o scheduler_sjn.c
gcc -lm -o sim_sjn scheduler_sjn.o nonpreemptive_scheduler_simulation.o eventq.o st
ats.o
gcc -c -Wall  -DOVERHEAD=0.03 -DQUANTUM=0.1 -o rr_scheduler_simulation.o scheduler_
simulation.c
gcc -lm -o sim_rr scheduler_fcfs.o rr_scheduler_simulation.o eventq.o stats.o
gcc -c -Wall -DNUM_PRIORITY_LEVELS=5 -o scheduler_mlq.o scheduler_mlq.c
gcc -c -Wall  \
        -DOVERHEAD=0.03 -DQUANTUM=0.1 \
        -DNUM_PRIORITY_LEVELS=5 -o mlq_scheduler_simulation.o scheduler_simulation.
c
gcc -lm -DNUM_PRIORITY_LEVELS=5 -o sim_mlq scheduler_mlq.o mlq_scheduler_simulation
.o eventq.o stats.o
driscoll$ ./sim_fcfs
Beginning simulation
Scheduler overhead: 0.03
Time quantum:       0.00
reading job list from file:  "scheduler_job_data.txt"
Simulation completed
Summary statistics:
        Current time:             796.15
        Number of jobs started:       22
        Number of jobs completed:     22
        System throughput:        0.0276 jobs per unit time
        Total proc time:          713.42
        Total turnaround time:   1352.37
        Total wait time:          638.95
        Average turnaround time:           61.47
        Average wait time:         29.04
driscoll$ ./sim_sjn
Beginning simulation
Scheduler overhead: 0.03
Time quantum:       0.00
reading job list from file:  "scheduler_job_data.txt"
Simulation completed
Summary statistics:
        Current time:             796.15
        Number of jobs started:       22
        Number of jobs completed:     22
        System throughput:        0.0276 jobs per unit time
        Total proc time:          713.42
        Total turnaround time:   1271.28
        Total wait time:          557.86
        Average turnaround time:           57.79
        Average wait time:         25.36
driscoll$ ./sim_rr
Beginning simulation
Scheduler overhead: 0.03
Time quantum:       0.10
reading job list from file:  "scheduler_job_data.txt"
Simulation completed
Summary statistics:
        Current time:            1001.25
        Number of jobs started:       22
        Number of jobs completed:     22
        System throughput:        0.0220 jobs per unit time
        Total proc time:            1.32
        Total turnaround time:   5265.81
        Total wait time:           14.56
        Average turnaround time:          239.36
        Average wait time:          0.66
driscoll$ ./sim_mlq
Beginning simulation
Scheduler overhead: 0.03
Time quantum:       0.10
reading job list from file:  "scheduler_job_data.txt"
Simulation completed
Summary statistics:
        Current time:             729.03
        Number of jobs started:        1
        Number of jobs completed:      0
        System throughput:        0.0014 jobs per unit time
        Total proc time:            0.00
        Total turnaround time:      0.00
        Total wait time:            0.03
        Average turnaround time:            0.00
        Average wait time:          0.03
driscoll$ exit

Script done on Mon 22 Sep 2014 11:00:30 PM CDT
```