```c
/**
 * Return dynamically allocated stats for the scheduler simulation, see
 * scheduler.h for details. Memory allocated by your code will be free'd
 * by the similulator. Do NOT return a pointer to a stack variable.
 */
stats_t *stats() {
    stats_t *stats = malloc(sizeof(stats_t));
    stats->thread_count=queue_size(thrds_queue);
    stats->tstats = malloc(sizeof(stats_t)*stats->thread_count);
    //Thrd *current = malloc(sizeof(Thrd));
    //Thrd *current;
    int total_turn=0;
    int total_waiting =0;
    int loopLimit= queue_size(thrds_queue);
    for(int  i=0;i<loopLimit;i++){
        Thrd *current;
        // get the first thread info
        current =  queue_dequeue(thrds_queue);
        // update the tstats for each therad (for loop)
        stats->tstats[i].tid=  current->th->tid;
        stats->tstats[i].turnaround_time =  current->turn_time;
        stats->tstats[i].waiting_time =  current-> wait_time;
        // add to the total turnaround time for the general stats
        total_turn+=stats->tstats[i].turnaround_time;
        total_waiting += stats->tstats[i].waiting_time;
        free(current);
    }
    // calculate final waiting_time
    stats->waiting_time=total_waiting/stats->thread_count;
    // calculate final turnaround_time
    stats->turnaround_time = total_turn/stats->thread_count;
    queue_destroy(ready);
    queue_destroy(thrds_queue);
    return stats;

}
```

```c
/*
Abdel Rahman Alnajjar
*/
#include <stdlib.h>

#include "simulator.h"
#include "scheduler.h"
#include "queue.h"

static thread_t *running = NULL; // current running thread
static void *ready; // ready queue
static void *thrds_queue; //  queue of all threads

static enum algorithm algo;


typedef struct _Thrd {
  thread_t *th;
  //int arrive_time;
    int turn_time;
    int wait_time;
    int started_waiting;

}Thrd;


static bool inner_comparator(void *a, void *b) {
        return ( void * )((Thrd*)a)->th == ( void * )((Thrd*)b);

    }

static int inner_comparator_for_priority(void *a, void *b) {
        return ((thread_t*)a)->priority - ((thread_t*)b)->priority;
      }
```

```c
void update_waiting_time(thread_t *t){
    // for tick
    if(t== NULL){
        Thrd *current = queue_find(thrds_queue, inner_comparator, running);
        if(current !=NULL){
            current->wait_time += (sim_time()- current->started_waiting);
        }
    }
    // for io_starting()
    else{
        Thrd *current = queue_find(thrds_queue, inner_comparator, t);
        if(current !=NULL){
        current->wait_time += (sim_time()- current->started_waiting);
        }
    }
}


/**
 * Initalise a scheduler implemeting the requested ALGORITHM. QUANTUM is only
 * meaningful for ROUND_ROBIN.
 */

void scheduler(enum algorithm algorithm, unsigned int quantum) {
    ready = queue_create(); //initialize the readt queue
    thrds_queue = queue_create(); //initialize the thrds_queue
    algo = algorithm;
}
```

```c
/**
 * Thread T is ready to be scheduled for the first time.
 */
void sys_exec(thread_t *t) {
        queue_enqueue(ready, t);
        Thrd *thrd = malloc(sizeof(Thrd));
        thrd->th = t;
        thrd->turn_time=sim_time(); // the initial time;
        thrd-> wait_time= 0; // initially wait time is 0;
        thrd -> started_waiting=sim_time(); // set started_waiting
        queue_enqueue(thrds_queue, thrd);
}

/**
 * Programmable clock interrupt handler. Call sim_time() to find out
 * what tick this is. Called after all calls to sys_exec() for this
 * tick have been made.
 */
void tick() {


    if(!running  && queue_size(ready) >0){

        // sort the ready queue for the Non-preemptive priority
        if(algo ==2 || algo==3){
            queue_sort(ready, inner_comparator_for_priority);
        }

        running = queue_dequeue(ready);
        sim_dispatch(running);
        // update waiting time FCFS
        update_waiting_time(NULL);
    }
}
```

```c
/**
 * Thread T has completed execution and should never again be scheduled.
 */
void sys_exit(thread_t *t) {

    Thrd *current = queue_find(thrds_queue, inner_comparator, t);
    current->turn_time =sim_time()-current->turn_time +1;
    //current->turn_time =sim_time()-current->turn_time;

    //update_turnaround_time();

    running = NULL;

}

/**
 * Thread T has requested a read operation and is now in an I/O wait queue.
 * When the read operation starts, io_starting(T) will be called, when the
 * read operation completes, io_complete(T) will be called.
 */
void sys_read(thread_t *t) {

    Thrd *current = queue_find(thrds_queue, inner_comparator, t);

    if(current !=NULL){
        current->started_waiting =sim_time()+1;
    }
    running = NULL;
}

/**
 * Thread T has requested a write operation and is now in an I/O wait queue.
 * When the write operation starts, io_starting(T) will be called, when the
 * write operation completes, io_complete(T) will be called.
 */
void sys_write(thread_t *t) {
    sys_read(t);
}
```

```
147  /**
148   * An I/O operation requested by T has completed; T is now ready to be
149   * scheduled again.
150   */
151  void io_complete(thread_t *t) {
152
153      // start waiting
154      Thrd *current = queue_find(thrds_queue, inner_comparator, t);
155
156      if(current !=NULL){
157          current->started_waiting =sim_time()+1;
158      }
159
160      queue_enqueue(ready, t);
161
162  }
163
164
165  /**
166   * An I/O operation requested by T is starting; T will not be ready for
167   * scheduling until the operation completes.
168   */
169  void io_starting(thread_t *t) {
170
171      update_waiting_time(t);
172
173
174  }
```