```c
// Name: Rodrigo Ignacio Rojas Garcia
// Course Number: ECE 2230
// Section: 001
// Semester: Spring 2017
// Assignment Number: 3
// Â© Rodrigo Rojas. All Rights Reserved.
// Bug: Once the program reaches a request_counter of 27989, the program starts lea
king for unkown reason

// Library Declaration Section
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "disk_queue.h"
#include "event_queue.h"
#include "list.h"
#include "disk.h"
#include "randsim.h"
#include "structures.h"

// Define Declaration section
#define SIZE 1000
#define REQUEST_NUM 1000000
#define MAXCHARACTERS 1994

int main(int argc, char *argvp[])
{
    // Variable Declaration Section
    double global_time = 0.0;
    int request_counter = 1;
    double seek_time_result;
    double next_request_time = 0;
    double queue_time = 0;
    double queue_time_min = 0;
    double queue_time_max = 0;
    double queue_time_average = 0;
    double io_time = 0;
    double io_time_min = 0;
    double io_time_max = 0;
    double io_time_average = 0;
    double total_time = 0;
    double total_time_min = 0;
    double total_time_max = 0;
    double total_time_average = 0;
    int first_time = 0;
    int current_track = 0;
    event_queue_t event_queue;
    disk_queue_t disk_queue;
    request_t request;
    event_t event;
    event_t event1;
    event_t event2;
    event_t event3;
    event_t event4;
    request_t removed_request;

    // Allocates dynamic memory for the event_queue which will also allocate memory
 for the array of pointers
    // depending of the value of variable size
    event_queue = event_queue_init(SIZE);

    // Allocates dynamic memory for a disk_queue linked list
    disk_queue = disk_queque_init();

    // Allocates dynamic memory for a request
    request = (request_t)calloc(1, sizeof(struct request_s));
    request->track = request_track();

    // Allocates dynamic memory for an event_t
    event = (event_t)calloc(1, sizeof(struct event_s));
    // Sets the starting values of an event when nothing has ran and uses event as
a "SHELL" for requests that will be passed
    event->event_time = 0;
    event->event_type = 1;
    event->request = request;


    // Will inser the first event into the array of event pointers
    event_queue_insert(event_queue, event);

    // While the event_queue is not EMPTY this loop will continue running
    while (event_queue_empty(event_queue) == 0)
    {
        // Remove and event from the event queue
        event = event_queue_remove(event_queue);
        // Sets global time to the current event time
        global_time = event->event_time;

        switch (event->event_type)
        {

        // REQUEST_SUBMIT will start if the event_type of an event is equal to 1
            case 1:
                // Sets the arrival_time of the request
                event->request->arrival_time = global_time;
                if (disk_queue_empty(disk_queue) == -1)
                {
                    // Schedules a DISK_READY
                    event1 = (event_t)calloc(1, sizeof(struct event_s));
                    event1->event_time = global_time;
                    event1->event_type = 2;
                    event_queue_insert(event_queue, event1);
                }
                // Inserts the request on the disk queue
                disk_queue_insert(disk_queue, event->request);
                if (request_counter <= REQUEST_NUM)
                {
                    // Computes when the next request should arrive
                    next_request_time = randsim_exp();
                    event2 = (event_t)calloc(1, sizeof(struct event_s));
                    // Creates new request and allocates dynamic memory for it
                    request_t new_request;
                    new_request = (request_t)calloc(1, sizeof(struct request_s));
                    // Uses function reques_track() function to determine the track
 the request is on
                    new_request->track = request_track();
                    event2->event_type = 1;
                    event2->event_time = global_time + next_request_time;
                    event2->request = new_request;
                    event_queue_insert(event_queue, event2);
                    // Increments the request counter
                    request_counter++;
                }
                break;
            // DISK_READY will start if the event_type of an event is equal to 2
            case 2:
                event3 = (event_t)calloc(1, sizeof(struct event_s));
                // Creates reuest_t variable head_request which will be used to sto
re the returned request of function
                // disk_queue_peek().
                request_t head_request;
                head_request = disk_queue_peek(disk_queue);
                // Updates the start_time of the request
                head_request->start_time = global_time;
```

```
                // Determines how long it will take to process IO using function se
ek_time()
                seek_time_result = seek_time(current_track, head_request->track);
                // Update the current track
                current_track = head_request->track;
                event3->event_time = global_time + seek_time_result;
                // Schedules a new REQUEST_DONE event
                event3->event_type = 3;
                event_queue_insert(event_queue, event3);
                break;
                // REQUEST DONE
            case 3:
                // removed_request that will be returned the request removed from t
he disk_queue_remove()
                // function
                removed_request = disk_queue_remove(disk_queue);
                if (disk_queue_empty(disk_queue) == 0)
                {
                    // Schedules another DISK_READY
                    event4 = (event_t)calloc(1, sizeof(struct event_s));
                    event4->event_time = global_time;
                    event4->event_type = 2;
                    event_queue_insert(event_queue, event4);
                }
                // Updates the value of finish_time for the request returned
                removed_request->finish_time = global_time;
                // Computes Queue_time, io_time, and total_time
                queue_time = removed_request->start_time - removed_request->arrival
_time;
                queue_time_average = queue_time_average + removed_request->start_ti
me - removed_request->arrival_time;
                io_time = removed_request->finish_time - removed_request->start_tim
e;
                io_time_average = io_time_average + removed_request->finish_time -
removed_request->start_time;
                total_time = removed_request->finish_time - removed_request->arriva
l_time;
                total_time_average = total_time_average + removed_request->finish_t
ime - removed_request->arrival_time;

                // Sets queue_time_min/max, io_time_min/max, and total_time_minx/ma
x to the first value returend on the removed
                // request
                if (first_time == 0)
                {
                    queue_time_min = queue_time;
                    queue_time_max = queue_time;
                    io_time_min = io_time;
                    io_time_max = io_time;
                    total_time_min = total_time;
                    total_time_max = total_time;
                    first_time++;
                }
                // Updates minimum, maximum, and average of queue_time, io_time, an
d total_time
                else
                {
                    if (queue_time < queue_time_min)
                    {
                        queue_time_min = queue_time;
                    }
                    if (queue_time > queue_time_max)
                    {
                        queue_time_max = queue_time;
                    }
                    if (io_time < io_time_min)
                    {
                        io_time_min = io_time;
                    }
                    if (io_time > io_time_max)
                    {
                        io_time_max = io_time;
                    }
                    if (total_time < total_time_min)
                    {
                        total_time_min = total_time;
                    }
                    if (total_time > total_time_max)
                    {
                        total_time_max = total_time;
                    }
                }
                // Frees the removed_request returned
                free(removed_request);
                break;
        }
        // Frees an event once is no longer needed
        free(event);
    }

    // Prints the final results of time in milliseconds  -
    printf("\nQueue minimum time: %f\n", queue_time_min);
    printf("Queue maximum time: %f\n", queue_time_max);
    printf("Queue average time: %f\n\n", queue_time_average / request_counter);
    printf("IO minimum time: %f\n", io_time_min);
    printf("IO maximum time: %f\n", io_time_max);
    printf("IO average time: %f\n\n", io_time_average / request_counter);
    printf("Total minimum time: %f\n", total_time_min);
    printf("Total maximum time: %f\n", total_time_max);
    printf("Total average time: %f\n\n", total_time_average / request_counter);

    // Frees all allocated dynamic memory for both disk_queue and event_queue
    event_queue_finalize(event_queue);
    disk_queue_finalize(disk_queue);

    return 0;
}
```