```c
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include "queue.h"

#define CMD_BUF_SIZE 0x800
#define QUEUE_SIZE 50

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int alloc_threads = 0;
queue *q;
cmd_info *running_cmds;

void *threadProc(void *arg);
void runCmd();
void initRunningCmdArr();
void resetThread(int thread);
void checkThread(pthread_t *tid);

void runCmd()
{
}

void *threadProc(void *arg)
{
    int bisQ = 1;
    int thread_num = (int)(long)arg;
    do {
        // Lock mutex and update global vars
        pthread_mutex_lock(&mutex);
        cmd_info *command;
        command = queue_delete(q);

        strcpy(running_cmds[thread_num].cmd, command->cmd);
        running_cmds[thread_num].jobid = command->jobid;
        pthread_mutex_unlock(&mutex);

        // Run the command given
        char line[CMD_BUF_SIZE], *token, *jobidout, *jobiderr;
        char *total_args[20];
        int count, pid, status;
        int fdout, fderr;

        jobidout = malloc(sizeof(char) * 20);
        jobiderr = malloc(sizeof(char) * 20);
        if (mkdir("jobs_out", 0775) && errno != EEXIST) {
            printf("Error creating output directory\n");
            exit(-1);
        }

        strcpy(line, command->cmd);

        count = 0;
        token = strtok(line, " ");
        while (token != NULL) {
            while (strstr(token, "\n") != NULL)
                token[strcspn(token, "\n")] = 0;
            total_args[count++] = token;
            token = strtok(NULL, " ");
        }
        total_args[count] = 0;

        pid = fork();
        if (pid == 0) {  // Child process exec
            sprintf(jobidout, "jobs_out/%d.out", command->jobid);
            if ((fdout = open(jobidout,
                              O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
                printf("\nError opening <jobid>.out\n");
                return (NULL);
            }
            sprintf(jobiderr, "jobs_out/%d.err", command->jobid);
            if ((fderr = open(jobiderr,
                              O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1) {
                printf("\nError opening <jobid>.err\n");
                return (NULL);
            }
            dup2(fdout, 1);
            dup2(fderr, 2);
            execvp(total_args[0], total_args);
            perror("Exec failed");
            return (NULL);
        }
        else if (pid > 0) {  // Parent process
            wait(&status);
        }
        else {  // ICE
            perror("Fork failed");
            exit(EXIT_FAILURE);
        }

        // Lock mutex and update global vars
        pthread_mutex_lock(&mutex);
        resetThread(thread_num);
        bisQ = q->count;
        pthread_mutex_unlock(&mutex);

    } while (bisQ);

    return (NULL);
}

void initRunningCmdArr()
{
    int i;
    for (i = 0; i < alloc_threads; i++) {
        resetThread(i);
    }
}

void resetThread(int thread)
{
    strcpy(running_cmds[thread].cmd, "");
    running_cmds[thread].jobid = -1;
}

int runningProcCount() {
    int i, count = 0;
    for (i = 0; i < alloc_threads; i++) {
        if (running_cmds[i].jobid != -1)
            count++;
    }
    return count;
}

void checkThread(pthread_t *tid)
{
    int i;
    for (i = 0; i < alloc_threads; i++) {
        if (running_cmds[i].jobid == -1) {
            pthread_create(&tid[i], NULL, threadProc, (void *)(long)i);
            break;
        }
    }
}

int main(int argc, char const *argv[])
{
    if (argc < 2) {
        printf("Command must be in the format: ./myscheduler <# of cores>\n");
        exit(-1);
    }
    else if (atoi(argv[1]) < 1) {
        printf("At least 1 core must be allocated\n");
        exit(-1);
    }

    int thread_count = atoi(argv[1]);
    char cmd[CMD_BUF_SIZE], cmd_dup[CMD_BUF_SIZE];
    int jobcounter = 0;

    alloc_threads = thread_count;
    q = queue_init(QUEUE_SIZE);

    pthread_t *tid;
    tid = (pthread_t *)malloc(sizeof(pthread_t) * thread_count);
    running_cmds = (cmd_info *)malloc(sizeof(cmd_info) * thread_count);
    initRunningCmdArr(thread_count);

    while (1) {
        // Prompt for command to be entered
        printf("Enter command> ");
        fgets(cmd, CMD_BUF_SIZE, stdin);
        cmd[strcspn(cmd, "\n")] = 0;

        strcpy(cmd_dup, cmd);
        strtok(cmd_dup, " ");
        if (!strcmp(cmd_dup, "submit")) {
            // Check that submit also contains command
            if (strlen(cmd) > strlen(cmd_dup) + 2)
                memmove(cmd, cmd + strlen(cmd_dup) + 1, strlen(cmd));
            else {
                printf("Please include a command with \"submit\"\n");
                continue;
            }

            cmd_info *new_cmd = malloc(sizeof *new_cmd);
            strcpy(new_cmd->cmd, strdup(cmd));
            new_cmd->jobid = jobcounter++;
            queue_insert(q, new_cmd);
            printf("Job %d added to the queue\n", new_cmd->jobid);
            checkThread(tid);
        }
        else if (!strcmp(cmd_dup, "showjobs")) {
            int j, count = q->count;
            printf("jobid\tcommand\t\tstatus\n");
            for (j = 0; j < alloc_threads; j++) {
                if (running_cmds[j].jobid >= 0) {
                    printf("%d\t%s\tRunning\n", running_cmds[j].jobid, running_cmds[j].cmd)
;
                }
            }
            for (j = 0; j < count; j++) {
                printf("%d\t%s\tQueued\n", q->buffer[(q->start + j) % q->size].jobid, q->bu
ffer[(q->start + j) % q->size].cmd);
            }
        }
        else if (!strcmp(cmd_dup, "exit")) {
            if (q->count > 0) {
                printf("%d process(es) are running/queued; still exit? (y/n): ", (q->count)
+runningProcCount());
                fgets(cmd, CMD_BUF_SIZE, stdin);
                cmd[strcspn(cmd, "\n")] = 0;
                strcpy(cmd_dup, cmd);

                if (strcasecmp(cmd_dup, "y")) {
                    continue;
                }
            }
            printf("Goodbye\n");
            queue_destroy(q);
            exit(0);
        }
        else if (!strcmp(cmd_dup, "")) {
        }
        else {
            printf("\"%s\" not OK\t", cmd_dup);
            printf("Available commands: \"submit\", \"showjobs\"\n");
        }

        fflush(stdin);
        fflush(stdout);
    }

    return 0;
}
```