```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <fcntl.h>
#include "helpers.h"

/*
Create a job with the given values and fill
all other values with safe defaults.
*/
job create_job(char *cmd, int jid)
{
    job j;
    j.jid = jid;
    j.cmd = get_copy(cmd);
    j.stat = "waiting";
    j.estat = -1;
    j.start = j.stop = NULL;
    sprintf(j.fnout, "%d.out", j.jid);
    sprintf(j.fnerr, "%d.err", j.jid);
    return j;
}

/*
If mode is "showjobs", list all non-completed jobs from jobs,
displaying each job's ID, command, and status.

If mode is "submithistory", list all completed jobs from jobs,
displaying each job's job ID, thread ID, command,
exit status, start time, and end time.
*/
void list_jobs(job *jobs, int n, char *mode)
{
    int i;
    if (jobs != NULL && n != 0)
    {
        if (strcmp(mode, "showjobs") == 0)
        {
            for (i = 0; i < n; ++i)
            {
                if (strcmp(jobs[i].stat, "complete") != 0)
                    printf("Job ID: %d\n"
                           "Command: %s\n"
                           "Status: %s\n\n",
                           jobs[i].jid,
                           jobs[i].cmd,
                           jobs[i].stat);
            }
        }
        else if (strcmp(mode, "submithistory") == 0)
        {
            for (i = 0; i < n; ++i)
            {
                if (strcmp(jobs[i].stat, "complete") == 0)
                    printf("Job ID: %d\n"
                           "Thread ID: %ld\n"
                           "Command: %s\n"
                           "Exit Status: %d\n"
                           "Start Time: %s\n"
                           "Stop Time: %s\n\n",
                           jobs[i].jid,
                           jobs[i].tid,
                           jobs[i].cmd,
                           jobs[i].estat,
                           jobs[i].start,
                           jobs[i].stop);
            }
        }
    }
```

```
}

/*
Create the queue data structure and initialize it.
Adapted from Professor Bangalore's queue example.
*/
queue *queue_init(int n)
{
    queue *q = malloc(sizeof(queue));
    q->size = n;
    q->buffer = malloc(sizeof(job *) * n);
    q->start = 0;
    q->end = 0;
    q->count = 0;

    return q;
}

/*
Insert the job pointer jp into the queue q, update the
pointers and count, and return the number of items
in the queue (-1 if q is null or full).
Adapted from Professor Bangalore's queue example.
*/
int queue_insert(queue *q, job *jp)
{
    if ((q == NULL) || (q->count == q->size))
        return -1;

    q->buffer[q->end % q->size] = jp;
    q->end = (q->end + 1) % q->size;
    ++q->count;

    return q->count;
}

/*
Delete a job pointer from the queue, update the
pointers and count, and return the job pointer
deleted (-1 if q is null or empty).
Adapted from Professor Bangalore's queue example.
*/
job *queue_delete(queue *q)
{
    if ((q == NULL) || (q->count == 0))
        return (job *)-1;

    job *j = q->buffer[q->start];
    q->start = (q->start + 1) % q->size;
    --q->count;

    return j;
}

/*
Delete the queue data structure.
Adapted from Professor Bangalore's queue example.
*/
void queue_destroy(queue *q)
{
    free(q->buffer);
    free(q);
}

/*
Record input character-by-character from stdin in s
until n-1 characters are read or a newline is reached.
If EOF is reached, return -1. Otherwise, explicitly end s
with a null terminator and return the number of characters read.
*/
```

```
int get_line(char *s, int n)
{
    int i, c;
    for (i = 0; i < n - 1 && (c = getchar()) != '\n'; ++i)
    {
        if (c == EOF)
            return -1;
        s[i] = c;
    }
    s[i] = '\0';
    return i;
}

/*
Return 1 if c is a whitespace character and 0 otherwise.
*/
int is_space(char c)
{
    return (c == ' ' ||
            c == '\t' ||
            c == '\n' ||
            c == '\r' ||
            c == '\x0b' ||
            c == '\x0c');
}

/*
Return a pointer to the first non-whitespace character in s.
*/
char *left_strip(char *s)
{
    int i;

    i = 0;
    while (is_space(s[i]))
        ++i;

    return s + i;
}

/*
Return a null-terminated copy of
the null-terminated string s.
*/
char *get_copy(char *s)
{
    int i, c;
    char *copy;

    i = -1;
    copy = malloc(sizeof(char) * strlen(s));
    while ((c = s[++i]) != '\0')
        copy[i] = c;
    copy[i] = '\0';

    return copy;
}

/*
Return a null-terminated copy of the null-terminated
string s up until the first newline or the end of s.
*/
char *get_copy_until_newline(char *s)
{
    int i, c;
    char *copy;

    i = -1;
    copy = malloc(sizeof(char) * strlen(s));
    while ((c = s[++i]) != '\0' && c != '\n')
```

```
        copy[i] = c;
    copy[i] = '\0';

    return copy;
}

/*
Return a copy of the result of ctime() with the current
timestamp from time(). Returning a copy is important
because subseqent calls to ctime() mutate strings
previously made by ctime().
*/
char *current_datetime_str()
{
    time_t tim = time(NULL);
    return get_copy_until_newline(ctime(&tim));
}

/*
Parse space/tab separated tokens in order from the given string
into a null-terminated array of strings and return it.
I used my submission for Project #1 as a guide for this.
*/
char **get_args(char *line)
{
    char *copy = malloc(sizeof(char) * (strlen(line) + 1));
    strcpy(copy, line);

    char *arg;
    char **args = malloc(sizeof(char *));
    int i = 0;
    while ((arg = strtok(copy, " \t")) != NULL)
    {
        args[i] = malloc(sizeof(char) * (strlen(arg) + 1));
        strcpy(args[i], arg);
        args = realloc(args, sizeof(char *) * (++i + 1));
        copy = NULL;
    }
    args[i] = NULL;
    return args;
}

/*
Open a log file with the given filename and
return its file descriptor.
*/
int open_log(char *fn)
{
    int fd;
    if ((fd = open(fn, O_CREAT | O_APPEND | O_WRONLY, 0755)) == -1)
    {
        fprintf(stderr, "Error: failed to open \"%s\"\n", fn);
        perror("open");
        exit(EXIT_FAILURE);
    }
    return fd;
}
```