```
/*
Name: Eric Latham
BlazerID: EOLATHAM
Project #: 4
To compile: gcc -Wall -lpthread -o job_scheduler job_scheduler.c helpers.c
To run: ./job_scheduler CONCURRENCY
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <sys/wait.h>
#include <pthread.h>
#include "helpers.h"

#define LINELEN 1000 /* maximum input line length */
#define JOBSLEN 1000 /* maximum JOBS length */
#define JOBQLEN 100  /* maximum JOBQ length */

int CONCUR;        /* user-defined job concurrency */
int NWORKING;      /* number of currently working jobs */
job JOBS[JOBSLEN]; /* permanent array of submitted jobs */
queue *JOBQ;       /* queue of pointers to waiting jobs */

/*
Complete the job pointed to by arg using fork/exec.
*/
void *complete_job(void *arg)
{
    job *jp;     /* job pointer from arg */
    char **args; /* array of args to be parsed from job command */
    pid_t pid;   /* process ID */

    jp = (job *)arg;

    ++NWORKING;
    jp->stat = "working";
    jp->start = current_datetime_str();

    pid = fork();
    if (pid == 0) /* child process */
    {
        dup2(open_log(jp->fnout), STDOUT_FILENO); /* redirect job stdout */
        dup2(open_log(jp->fnerr), STDERR_FILENO); /* redirect job stderr */
        args = get_args(jp->cmd);
        execvp(args[0], args);
        fprintf(stderr, "Error: command execution failed for \"%s\"\n", args[0]);
        perror("execvp");
        exit(EXIT_FAILURE);
    }
    else if (pid > 0) /* parent process */
    {
        waitpid(pid, &jp->estat, WUNTRACED);
        jp->stat = "complete";
        jp->stop = current_datetime_str();

        if (!WIFEXITED(jp->estat))
            fprintf(stderr, "Child process %d did not terminate normally!\n", pid);
    }
    else
    {
        fprintf(stderr, "Error: process fork failed\n");
        perror("fork");
        exit(EXIT_FAILURE);
    }

    --NWORKING;
    return NULL;
```

```
}

/*
Continuously check for and complete jobs.
Sleep for a second between iterations to avoid the
race condition of a new job starting before the previous
complete_job call was able to increment NWORKING.
*/
void *complete_jobs(void *arg)
{
    job *jp; /* job pointer */

    NWORKING = 0;
    for (;;)
    {
        if (JOBQ->count > 0 && NWORKING < CONCUR)
        {
            /* pull next job from queue */
            jp = queue_delete(JOBQ);

            /* use a new thread to complete job */
            pthread_create(&jp->tid, NULL, complete_job, jp);

            /* mark new thread detached to free its resources when done */
            pthread_detach(jp->tid);
        }
        sleep(1); /* wait a second, then check again */
    }
    return NULL;
}

/*
Handle user input; insert new jobs and respond to commands.
Kill the current process group upon reaching EOF.
*/
void handle_input()
{
    int i;              /* job counter */
    char line[LINELEN]; /* input line buffer */
    char *kw;           /* input keyword */
    char *cmd;          /* input submit job command */

    printf("Enter `submit COMMAND [ARGS]` to create and run a job.\n"
           "Enter `showjobs` to list all JOBS that are "
           "currently waiting or working.\n"
           "Enter `submithistory` to list all JOBS that were "
           "completed during the current session.\n\n");

    i = 0;
    while (printf("> ") && get_line(line, LINELEN) != -1)
    {
        if ((kw = strtok(get_copy(line), " \t\n\r\x0b\x0c")) != NULL)
        {
            if (strcmp(kw, "submit") == 0)
            {
                if (i >= JOBSLEN)
                    printf("Job history full; restart the program to schedule more\n");
                else if (JOBQ->count >= JOBQ->size)
                    printf("Job queue full; try again after more jobs complete\n");
                else
                {
                    cmd = left_strip(strstr(line, "submit") + 6);
                    JOBS[i] = create_job(cmd, i);
                    queue_insert(JOBQ, JOBS + i);
                    printf("Added job %d to the job queue\n", i++);
                }
            }
            else if (strcmp(kw, "showjobs") == 0 ||
                     strcmp(kw, "submithistory") == 0)
                list_jobs(JOBS, i, kw);
```

```
        }
    }
    kill(0, SIGINT); /* kill the current process group upon reaching EOF */
}

int main(int argc, char **argv)
{
    char *fnerr;   /* filename where main stderr is redirected */
    pthread_t tid; /* thread ID */

    if (argc != 2)
    {
        printf("Usage: %s CONCURRENCY\n", argv[0]);
        exit(EXIT_SUCCESS);
    }

    CONCUR = atoi(argv[1]);
    if (CONCUR < 1)
        CONCUR = 1;
    else if (CONCUR > 8)
        CONCUR = 8;

    printf("Concurrency: %d\n\n", CONCUR);

    /* redirect main stderr to file */
    fnerr = malloc(sizeof(char) * (strlen(argv[0]) + 5));
    sprintf(fnerr, "%s.err", argv[0]);
    dup2(open_log(fnerr), STDERR_FILENO);

    JOBQ = queue_init(JOBQLEN);

    /* use a new thread to complete jobs */
    pthread_create(&tid, NULL, complete_jobs, NULL);

    /* use the main thread to handle input */
    handle_input();

    exit(EXIT_SUCCESS);
}
```