```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <sys/wait.h>

#include "queue.h"



struct Jobs {
   Job *job;
   struct Jobs *next;
};

struct Jobs *registeredJobs = 0,    //linked-list to keep track of all jobs
          *listend;

char *sMap[26];
int jid = 0;
queue *jobs_queue;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

char* ptoa(Job *process) {
   char *str = (char*)malloc(sizeof(char) * 200);
   str[0] = '\0';
```

```
    strcat(str, process->cmd);
    strcat(str, " ");

    int i=0;
    while (process->args[i]) {
        strcat(str, process->args[i++]);
        strcat(str, " ");
    }

    return str;
}

int driver() {
    while(1) {
        Job *process = malloc(sizeof (Job));
        char buffer[200];
        char *token;

        printf("Enter command> ");
        fgets(buffer, sizeof(buffer), stdin);

        //remove trailing newline character
        buffer[strcspn(buffer, "\n")] = 0;

        token = strtok(buffer, " ");

        if (strcmp(token, "submit") == 0) {
            //initialize process with program name and arguments
```

```
    process->cmd = strdup(strtok(NULL, " "));
    int arg_num = 0;
    token = strtok(NULL, " ");
    while (token != NULL) {
        process->args[arg_num] = strdup(token);
        arg_num++;
        token = strtok(NULL, " ");
    }
    process->args[arg_num] = NULL;
    time(&process->start);
    process->status = 'w';
    process->id = ++jid;

    struct Jobs *h = malloc(sizeof (struct Jobs));
    h->job = process;
    h->next = 0;
    if(registeredJobs) {
        listend->next = h;
        listend = h;
    } else {
        registeredJobs = h;
        listend = h;
    }
    queue_insert(jobs_queue, *process);        //add job to the queue
    printf("job %d added to the queue\n", jid);
    pthread_cond_signal(&cond);     //signal all the worker: work is available
}
else if (strcmp(token, "showjobs") == 0) {
    struct Jobs *crr = registeredJobs;
```

```
    printf("%-5s %-37s %s\n", "jobid", "cmd", "status");
    while (crr) {
        if(crr->job->status == 's' || crr->job->status == 'f') {
            crr = crr->next;
            continue;
        }
        printf("%-5d %-37s %s\n", crr->job->id, ptoa(crr->job), sMap[crr->job->status-'a']);
        crr = crr->next;
    }
    free(process);
}
else if (strcmp(token, "submithistory") == 0) {
    struct Jobs *crr = registeredJobs;
    printf("%-5s %-37s %-27s %-27s %s\n", "jobid", "cmd", "starttime", "endtime", "status");
    while (crr) {
        if(crr->job->status == 'w' || crr->job->status == 'r') {
            crr = crr->next;
            continue;
        }
        printf("%-5d", crr->job->id);
        printf(" %-37s ", ptoa(crr->job));

        char *start = strdup(ctime(&crr->job->start)),
             *end   = strdup(ctime(&crr->job->end));

        start[strcspn(start, "\n")] = '\0';
        end[strcspn(end, "\n")] = '\0';
        printf("%-27s %-27s", start, end);
        printf(" %s\n", sMap[crr->job->status-'a']);
```

```
            crr = crr->next;
        }
        free(process);
    }
    else {
        printf("invalid cmd!\n");
        free(process);
    }
  }
}

void *work(void *arg) {
  Job *job = malloc(sizeof (Job));

  while (1) {
    pthread_mutex_lock(&mutex);
    while (jobs_queue->count < 1) {
      pthread_cond_wait(&cond, &mutex);
    }
    *job = queue_delete(jobs_queue);
    struct Jobs *crr = registeredJobs;
    while(crr->next) {
      if(crr->job->id == job->id) break;
      crr = crr->next;
    }
    job = crr->job;

    pthread_mutex_unlock(&mutex);
```

```
//execute the job
job->status = 'r';

FILE *oFile, *eFile;
char ofName[20], efName[20];

snprintf(ofName, sizeof(ofName), "%d.out", job->id);
snprintf(efName, sizeof(efName), "%d.err", job->id);

//open the output & the error file
oFile = fopen(ofName, "w");
eFile = fopen(efName, "w");

if (oFile == NULL || eFile == NULL) {
   perror("error: ");
   exit(EXIT_FAILURE);
}

pid_t pid = fork();
if (pid < 0) {
   perror("fork: ");
   exit(EXIT_FAILURE);
}
else if (pid == 0) {
   //child process
   dup2(fileno(oFile), STDOUT_FILENO);
   dup2(fileno(eFile), STDERR_FILENO);

   char *argv[16];
```

```
        argv[0]  = strdup(job->cmd);
        int i=1;
        for(; job->args[i-1]; ++i)
            argv[i] = strdup(job->args[i-1]);
        argv[i] = NULL;
        if (execvp(argv[0], argv) < 0) {
            exit(EXIT_FAILURE);
        }
    }
    else {
        //parent process
        int status;
        waitpid(pid, &status, 0);

        //close files
        fclose(oFile);
        fclose(eFile);

        //check if child process exited successfully
        if (WIFEXITED(status) && WEXITSTATUS(status) == EXIT_SUCCESS) {
            job->status = 's';
        }
        else {
            job->status = 'f';
        }

        time(&job->end);
    }
}
```

```
}

int main(int argc, char ** argv) {
   if(argc < 2) {
      printf("Error: missing argument(s)\n");
      printf("usage: ./scheduler <p>  #p: no of cores\n");
      exit(1);
   }
   int p = atoi(argv[1]);
   jobs_queue = queue_init(1000);
   sMap['w'-'a'] = "Waiting";
   sMap['r'-'a'] = "Running";
   sMap['s'-'a'] = "Success";
   sMap['f'-'a'] = "Failure";

   pthread_t cores[p];
   for(int i=0; i<p; ++i) {
      if (pthread_create(&cores[i], NULL, work, NULL) != 0) {
         perror("pthread_create");
         exit(1);
      }
   }

   driver();

   for(int i=0; i<p; ++i) {
      if (pthread_join(cores[i], 0) != 0) {
         perror("pthread_join");
         exit(1);
```

```
        }
    }


    return 0;
}
```