```
// Authors : Sai Pranathi Veldanda & Sai Theja Kumar Bogineni

#include <unistd.h>
#include <string.h>
#include <wait.h>
#include "tpool.h"
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>


struct tpool_work {
   thread_func_t      func;
   void              *arg;
   struct tpool_work *next;
};

typedef struct tpool_work tpool_work_t;

struct tpool {
   tpool_work_t    *work_first;
   tpool_work_t    *work_last;
   pthread_mutex_t  work_mutex;
pthread_cond_t   work_cond;
   pthread_cond_t   working_cond;
   size_t           working_cnt;
   size_t           thread_cnt;
   bool             stop;
};

static tpool_work_t *tpool_work_create(thread_func_t func, void *arg)
{
   tpool_work_t *work;
 if (func == NULL)
       return NULL;

   work       = malloc(sizeof(*work));
   work->func = func;
   work->arg  = arg;
   work->next = NULL;
   return work;
}

static void tpool_work_destroy(tpool_work_t *work)
{
   if (work == NULL)
       return;
   free(work);
}

static tpool_work_t *tpool_work_get(tpool_t *tm)
{
   tpool_work_t *work;

   if (tm == NULL)
```

```
        return NULL;

    work = tm->work_first;
    if (work == NULL)
        return NULL;

    if (work->next == NULL) {
        tm->work_first = NULL;
        tm->work_last  = NULL;
    } else {
        tm->work_first = work->next;
    }

    return work;
}

static void *tpool_worker(void *arg)
{
    tpool_t      *tm = arg;
    tpool_work_t *work;
  while (1) {
        pthread_mutex_lock(&(tm->work_mutex));

        while (tm->work_first == NULL && !tm->stop)
            pthread_cond_wait(&(tm->work_cond), &(tm->work_mutex));

        if (tm->stop)
            break;

        work = tpool_work_get(tm);
        tm->working_cnt++;
        pthread_mutex_unlock(&(tm->work_mutex));

        if (work != NULL) {
            work->func(work->arg);
            tpool_work_destroy(work);
        }
        pthread_mutex_lock(&(tm->work_mutex));
        tm->working_cnt--;
        if (!tm->stop && tm->working_cnt == 0 && tm->work_first == NULL)
            pthread_cond_signal(&(tm->working_cond));
        pthread_mutex_unlock(&(tm->work_mutex));
    }

    tm->thread_cnt--;
    pthread_cond_signal(&(tm->working_cond));
    pthread_mutex_unlock(&(tm->work_mutex));
    return NULL;
}
tpool_t *tpool_create(size_t num)
{
    tpool_t   *tm;
    pthread_t  thread;
    size_t     i;
    if (num == 0)
```

```
        num = 2;

    tm             = calloc(1, sizeof(*tm));
    tm->thread_cnt = num;

    pthread_mutex_init(&(tm->work_mutex), NULL);
    pthread_cond_init(&(tm->work_cond), NULL);
    pthread_cond_init(&(tm->working_cond), NULL);

    tm->work_first = NULL;
    tm->work_last  = NULL;

    for (i=0; i<num; i++) {
        pthread_create(&thread, NULL, tpool_worker, tm);
        pthread_detach(thread);
    }
    return tm;
}

void tpool_destroy(tpool_t *tm)
{
    tpool_work_t *work;
    tpool_work_t *work2;

    if (tm == NULL)
        return;

    pthread_mutex_lock(&(tm->work_mutex));
    work = tm->work_first;
    while (work != NULL) {
    work2 = work->next;
        tpool_work_destroy(work);
        work = work2;
    }
    tm->stop = true;
    pthread_cond_broadcast(&(tm->work_cond));
    pthread_mutex_unlock(&(tm->work_mutex));

    tpool_wait(tm);

    pthread_mutex_destroy(&(tm->work_mutex));
    pthread_cond_destroy(&(tm->work_cond));
    pthread_cond_destroy(&(tm->working_cond));

    free(tm);
}

bool tpool_add_work(tpool_t *tm, thread_func_t func, void *arg)
{
    tpool_work_t *work;

    if (tm == NULL)
        return false;

    work = tpool_work_create(func, arg);
    if (work == NULL)
        return false;

    pthread_mutex_lock(&(tm->work_mutex));
    if (tm->work_first == NULL) {
        tm->work_first = work;
        tm->work_last  = tm->work_first;
    } else {
        tm->work_last->next = work;
        tm->work_last       = work;
    }
  pthread_cond_broadcast(&(tm->work_cond));
    pthread_mutex_unlock(&(tm->work_mutex));
    return true;
}

void tpool_wait(tpool_t *tm)
{
    if (tm == NULL)
        return;

    pthread_mutex_lock(&(tm->work_mutex));
    while (1) {
        if ((!tm->stop && tm->working_cnt != 0) || (tm->stop && tm->thread_cnt != 0)) {
            pthread_cond_wait(&(tm->working_cond), &(tm->work_mutex));
        } else {
            break;
        }
    }
    pthread_mutex_unlock(&(tm->work_mutex));
}



static const size_t num_threads = 4;
static const size_t num_items   = 100;
int jobs = 1;

struct work_args {

    size_t job_id;

    char *args[3];
};


void worker(void *arg)

{

    struct work_args *job_args = arg;

    if (fork() == 0){

        printf("%s %s \n",job_args->args[0],job_args->args[1]);
        if (freopen(strcat((char *)job_args->job_id,".out"), "w+", stdout) == NULL)

            {

                perror("freopen() failed");

            }

        execvp(job_args->args[0], job_args->args);

        fclose(stdout);

    }else{

        wait(NULL);

    }
}



void print_menu (tpool_t *tm, int *vals) {

    bool stop = false;

    while (stop ==  false) {

//        printf("%d\n",stop);

        printf("Enter command>");

        char *line = NULL;
   size_t len = 0;
        ssize_t lineSize = 0;



        lineSize = getline(&line, &len, stdin);
        line[lineSize-1]='\0';

        char *command[3];

        int i =0;
   char *token;

        token = strtok(line, " ");

        while(token != NULL)

        {

            command[i++]=token;
  token = strtok(NULL," ");
```

```
    }

    if (strcmp(command[0], "submit")==0){

        struct work_args *args = calloc(1,sizeof(*args));

        args->job_id = jobs++;

        args->args[0] = command[1];

        args->args[1] = command[2];

        args->args[2] = NULL;

        tpool_add_work(tm, worker, args);

        printf("job submitted\n");

        jobs++;

    }
    if (strcmp(command[0], "showjobs")==0){

         printf("jobid\tcommand\t\status\n");

    }

    if (strcmp(command[0], "exit")==0){

        printf("e\n");

        stop = true;

    }}}


int main(int argc, char **argv)

{

    tpool_t *tm;
    int     *vals;

    tm   = tpool_create(num_threads);
   vals = calloc(num_items, sizeof(*vals));
   print_menu(tm,vals);
    tpool_wait(tm);
    free(vals);
   tpool_destroy(tm);
   return 0;

}
```