Prashant Yadav

CS 4348.002 – Program 1

```
/*****************************************************************************
* File: pthread_create.c
* Author: Linux Author
* Procedures:
* main – main method from where program starts its execution
* pthread_create-This method initializes a new thread which start execution from start routine.
* thread_start – every thread starts its execution from this method.
*****************************************************************************/

#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>

//it is macro to handle error (with error number)during execution  This loop will execute only once because of
//while(0) condition.
#define handle_error_en(en, msg)
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)

//It is macro to handle error (without error number) during execution.
#define handle_error(msg) \
        do { perror(msg); exit(EXIT_FAILURE); } while (0)

//This structure is used to store thread info consists of thread_id, thread number and arguments passed from
//command line.
struct thread_info {            // it is used as argument to thread_start()
    pthread_t thread_id;        // ID returned by pthread_create()
    int       thread_num;       // Number defined by application
    char      *argv_string;     // Arguments passed from command line
};

// Thread start function: display address near top of our stack,
//  and return upper-cased copy of argv_string

/*****************************************************************************
* void * thread_start(void *arg)
* Author: Linux author
* Date: 8 september 2019
* Description: It is the entry point for the threads created by pthread_create routine.
* Parameters:
* arg I/P It is the only argument to this method passed from command line.
* uargv O/P it returns command line arg to pthread in upper case.
*****************************************************************************/

static void * thread_start(void *arg)
{
   struct thread_info *tinfo = arg; //tinfo stores thread info
   char *uargv, *p;
   //This statement prints thread number and command line argument to that thread.
   printf("Thread %d: top of stack near %p; argv_string=%s\n",
```

```
tinfo->thread_num, &p, tinfo->argv_string);
uargv = strdup(tinfo->argv_string); //It creates a duplicate of  argv_string and sets its //address
//to pointer uargv.
     if (uargv == NULL)
         handle_error("strdup");

//It is a loop over uargv to print character in upper case.
    for (p = uargv; *p != '\0'; p++)
         *p = toupper(*p);
    return uargv;
}

/*****************************************************************************
* int main(int argc, char *argv[])
* Author: Linux author
* Date: 8 september 2019
* Description: This is the main method to test pthread_creation.
* Parameters:
* argc I/P int Number of arguments on the command line
* argv I/P char *[] The arguments on the command line
* main O/P int Status code (not currently used)
*****************************************************************************/

int
main(int argc, char *argv[])
{
     int s, tnum, opt, num_threads;
     struct thread_info *tinfo;
     pthread_attr_t attr;
     int stack_size;
     void *res;

     // The "-s" option specifies a stack size for our threads.
     stack_size = -1;
     // It is a while loop to iterate over all command line arguments.
     // getopt function parses the command line arguments.
     while ((opt = getopt(argc, argv, "s:")) != -1) {
         switch (opt) {
         case 's':                    // When getopt return option ‘s’
            stack_size = strtoul(optarg, NULL, 0)//Stroul changes all arguments to base 0
            break;

         default:                      //when there are no more args or something not ‘s’
             fprintf(stderr, "Usage: %s [-s stack-size] arg...\n",argv[0]);
             exit(EXIT_FAILURE);
         }
     }
    num_threads = argc - optind;   //Computer num_threads based on arg count
    s = pthread_attr_init(&attr); //Initializing pthread attributes.
    if (s != 0)
       handle_error_en(s, "pthread_attr_init");
    if (stack_size > 0) { //Set stack size attr if stack_size>0
       s = pthread_attr_setstacksize(&attr, stack_size);
       if (s != 0)
          handle_error_en(s, "pthread_attr_setstacksize");
     }

    //Allocate memory for pthread_create() arguments
    tinfo = calloc(num_threads, sizeof(struct thread_info));
     if (tinfo == NULL)
```

```
    handle_error("calloc");

    // Create one thread per command-line argument
    for (tnum = 0; tnum < num_threads; tnum++) {
         tinfo[tnum].thread_num = tnum + 1;
         tinfo[tnum].argv_string = argv[optind + tnum];

         // The pthread_create() call stores the thread ID into
         // corresponding element of tinfo[]

         s = pthread_create(&tinfo[tnum].thread_id, &attr,
                            &thread_start, &tinfo[tnum]);
         if (s != 0)
             handle_error_en(s, "pthread_create");
     }

    // Delete thread attributes object, as it is no  longer needed
    s = pthread_attr_destroy(&attr);
     if (s != 0)
         handle_error_en(s, "pthread_attr_destroy");

    // Now join each spawned thread and print its argument value.
    for (tnum = 0; tnum < num_threads; tnum++) {
         s = pthread_join(tinfo[tnum].thread_id, &res);
         if (s != 0)
             handle_error_en(s, "pthread_join");
         printf("Joined with thread %d; returned value was %s\n",
                 tinfo[tnum].thread_num, (char *) res);
         free(res);       // Free memory allocated by thread.
     }

    free(tinfo);
    exit(EXIT_SUCCESS);
}
```