**Multithreading – Thread Creation assignments**

Mandatory

1. Refer code in “simple\_thread.c”.

a. Modify the thread function to read and return username read from the user. Allocate heap memory for the user name and free in caller after displaying it.

b. Display the thread id’s of parent and child thread

A screen shot of a computer program

Description automatically generated

A computer screen shot of text

Description automatically generated

A computer screen with white text

Description automatically generated

2. Refer the code “pthread\_creation.c”. Modify the existing functions as below.

a. Main()- read a line of text and pass to new threadproc function below

threadproc() – to create 2 child threads to count words and to count characters, display the received values, return the values to the caller

Other thread functions to be used by threadproc() are given below

startroutine1()—to count words and return word count to caller

startroutine2() – to return the character count to the caller

A computer screen with text on it

Description automatically generated

#include <unistd.h>

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <string.h>

// Define a buffer size for input string

#define MAX\_STR\_LEN 1024

// Global variables to hold the results

int word\_count = 0;

int char\_count = 0;

// Function to count the words in the input string

void \*startroutine1(void \*thr\_p)

{

char \*input\_str = (char \*)thr\_p;

int count = 0;

char \*token = strtok(input\_str, " "); // Split the string by spaces

// Count words by splitting the string

while (token != NULL)

{

count++;

token = strtok(NULL, " ");

}

word\_count = count; // Store the word count globally

pthread\_exit((void \*)&word\_count);

}

// Function to count characters in the input string

void \*startroutine2(void \*thr\_p)

{

char \*input\_str = (char \*)thr\_p;

int count = strlen(input\_str); // Use strlen to count the characters, including spaces

char\_count = count; // Store the character count globally

pthread\_exit((void \*)&char\_count);

}

// Function to be executed by the main thread, which creates two child threads

void \*threadproc(void \*thr\_p)

{

pthread\_t tid1, tid2;

int retval;

char \*input\_str = (char \*)thr\_p;

printf("Input string: %s\n", input\_str); // Debugging statement

// Create the first thread to count words

retval = pthread\_create(&tid1, NULL, startroutine1, (void \*)input\_str);

if (retval != 0)

{

perror("Thread creation failed for counting words");

pthread\_exit(NULL);

}

// Create the second thread to count characters

retval = pthread\_create(&tid2, NULL, startroutine2, (void \*)input\_str);

if (retval != 0)

{

perror("Thread creation failed for counting characters");

pthread\_exit(NULL);

}

int \*word\_result, \*char\_result;

// Wait for the threads to complete and retrieve their results

retval = pthread\_join(tid1, (void \*\*)&word\_result);

if (retval != 0)

{

perror("pthread\_join failed for word count thread");

}

retval = pthread\_join(tid2, (void \*\*)&char\_result);

if (retval != 0)

{

perror("pthread\_join failed for character count thread");

}

// Display the results

printf("Word Count: %d\n", \*word\_result);

printf("Character Count: %d\n", \*char\_result);

pthread\_exit(NULL);

}

// Main function to read input and start the threadproc

int main()

{

char input[MAX\_STR\_LEN]; // Buffer to read input

// Read a line of text from the user

printf("Enter a line of text: ");

fgets(input, sizeof(input), stdin);

// Remove the newline character if it exists

input[strcspn(input, "\n")] = 0;

// Call threadproc to handle word and character counting

threadproc((void \*)input);

return 0;

}

3. Refer the program in “thread\_prg.c”. Complete the TBD sections , check the final solution for memory leak if any

A screen shot of a computer

Description automatically generated

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>

#include <pthread.h>

#include <unistd.h>

// Define the structure for thread arguments

typedef struct {

unsigned long long large\_serial\_number;

const char \*my\_name;

pthread\_t caller\_thread;

} fun\_thread\_args\_t;

#define NUM\_LOOPS 5u

// Thread function to perform the desired work

static void\* \_fun\_thread\_run(void \*arg) {

// Get our thread args from the dynamically allocated thread args passed-in

fun\_thread\_args\_t \*thread\_args = (fun\_thread\_args\_t \*)arg;

unsigned int counter = 0;

bool done = false;

while (!done) {

// Use args here in your thread loop

counter++;

printf("My name is: '%s', my number is: %llX, loop count: %u\n",

thread\_args->my\_name,

thread\_args->large\_serial\_number,

counter);

if (counter >= NUM\_LOOPS) {

printf("'%s' is sooooo done.\n", thread\_args->my\_name);

done = true;

} else {

sleep(1u);

}

}

// Free incoming args pointer before exiting the thread loop.

free(arg);

return NULL;

}

// Function to launch a thread with the provided arguments

static pthread\_t \*\_launch\_fun\_thread(pthread\_t \*thread, unsigned long long serial\_number, const char \*name) {

// Allocate memory and prepare args for the thread

fun\_thread\_args\_t \*args = (fun\_thread\_args\_t \*)calloc(1, sizeof(fun\_thread\_args\_t));

// Check for memory allocation failure

if (args == NULL) {

perror("Failed to allocate memory for thread arguments");

return NULL;

}

// Initialize the arguments

args->caller\_thread = pthread\_self();

args->large\_serial\_number = serial\_number;

args->my\_name = name;

// Create the thread

int rc = pthread\_create(thread, NULL, &\_fun\_thread\_run, args);

// If thread creation fails, free the allocated memory and return NULL

if (rc != 0) {

perror("Thread creation failed");

free(args); // Free memory on failure

return NULL;

}

return thread;

}

int main(void) {

puts("The fun we are having!");

pthread\_t fun\_thread1;

pthread\_t fun\_thread2;

// Launch first thread

if (\_launch\_fun\_thread(&fun\_thread1, 0xABCDEF12345678ULL, "super FUN thread 1") == NULL) {

return EXIT\_FAILURE;

}

usleep(500UL \* 1000UL); // Sleep for 500ms

// Launch second thread

if (\_launch\_fun\_thread(&fun\_thread2, 0xDEADBEEF55AA55ULL, "super FUN thread 2") == NULL) {

return EXIT\_FAILURE;

}

// Wait for both threads to finish

pthread\_join(fun\_thread1, NULL);

pthread\_join(fun\_thread2, NULL);

puts("We are done having fun :(");

return EXIT\_SUCCESS;

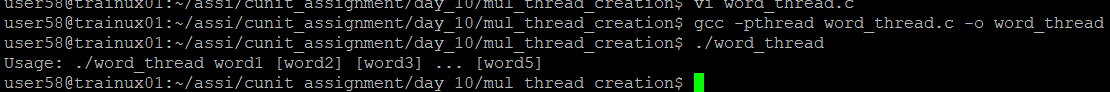
}

4. Write a program

a. to read a set of words as command line arguments and to create an array of threads (Consider a maximum of 5 words )

b. process each word using an separate thread. Let each thread append “\_ed” to the input word and return to main thread

c. main thread to wait for completion of each thread, retrieve returned string display with thread number, free memory



#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <pthread.h>

#define MAX\_WORDS 5

// Structure to pass data to each thread

typedef struct {

char \*word;

int thread\_num;

char \*modified\_word;

} thread\_data\_t;

// Function to append "\_ed" to the word and return it

void \*process\_word(void \*arg) {

thread\_data\_t \*data = (thread\_data\_t \*)arg

// Allocate memory for the modified word

data->modified\_word = (char \*)malloc(strlen(data->word) + 4); // "\_ed" = 3 chars

if (data->modified\_word == NULL) {

perror("Memory allocation failed");

pthread\_exit(NULL);

}

// Append "\_ed" to the word

strcpy(data->modified\_word, data->word);

strcat(data->modified\_word, "\_ed");

pthread\_exit(NULL); // Thread exits

}

int main(int argc, char \*argv[]) {

if (argc < 2 || argc > MAX\_WORDS + 1) {

fprintf(stderr, "Usage: %s word1 [word2] [word3] ... [word5]\n", argv[0]);

return 1;

}

pthread\_t threads[MAX\_WORDS];

thread\_data\_t thread\_data[MAX\_WORDS];

// Create threads for each word

for (int i = 1; i < argc; ++i) {

thread\_data[i - 1].word = argv[i];

thread\_data[i - 1].thread\_num = i;

// Create a new thread for each word

if (pthread\_create(&threads[i - 1], NULL, process\_word, (void \*)&thread\_data[i - 1]) != 0) {

perror("Failed to create thread");

return 1;

}

}

// Wait for each thread to finish and retrieve the result

for (int i = 0; i < argc - 1; ++i) {

pthread\_join(threads[i], NULL);

// Print the result

printf("Thread %d: %s\n", thread\_data[i].thread\_num, thread\_data[i].modified\_word);

// Free the allocated memory

free(thread\_data[i].modified\_word);

}

return 0;

}