Systems programming

Week 6 – Lab 10

Threads

In this laboratory students will implement concurrent and parallel programs using threads.

1 Threads

man pthreads(7) - Linux manual page – man7.org

man pthread_create(3) - Linux manual page – man7.org

man pthread_join(3) - Linux manual page - man7.org

Threads are a mechanisms offered by modern operating systems to execute multiples functions at the same time and inside the same program.

Programmers need to define special functions following a specif interface (**void * thread_function(void * arg)**) and launch them using the **pthread_create** function. From this point on, the program will continue executing the main and a new thread of execution starts running the **thread_function** code.

In order to use threads in C it is necessary to include the **pthread.h** file.

In order to compile a program that uses threads, it is necessary to use the **-lpthread** gcc option.

Students should read the provided PDF to further understand how threads work and are programmed:

The Linux Programming Interface – Sections 29.1 29.2 29.3 29.4 29.6

1.1 Exercise 1 – launching threads

Observe the **test-threads_1.c** program supplied, compile it and execute it with different input values.

Don't forget to use the **-lpthread** compilation argument.

After running a few times with different input values, answer the following questions:

- How many different values of **r** there are?
- Why does each digit (1, 2, 3, …) is repeated by each thread?

- What happens to the threads if the user presses **enter** and exits main?
- Read the **pthread_create** man page and take note of the various arguments.

2 Waiting for thread termination

pthread_join(3) - Linux manual page - man7.org

Although threads run concurrently at the same time inside the same program, it is possible to wait for the termination of a specific thread.

This wait is accomplished calling the **pthread_join** function. This function receives as first argument the **pthread_t** value assigned by the **pthread_create** and (for now) NULL as second argument. This function will be blocked and wait until the corresponding thread terminates.

2.1 Exercise 2 – waiting for the end of each task

Observe the **test-threads_2.c** program supplied, compile it and execute it with different input values. After running it a few times, compare it with the **test-threads_1.c** program and answer the following questions:

- Whats is difference in the output of this program compared with **test-threads_1.c**?
- What is different in the source code?
- Does **test-threads_2.c** takes advantage of multiple processors?

2.2 Exercise 3 – waiting for tasks termination

Modify the **test-threads_2.c** program so that the various threads run all at the same time, the **fgets** should be removed, but the program should only terminates after the last threads terminates.

Hint to run all threads at the same time:

remove the **pthread_join** from inside the loop

Hint to wait for all the threads before the exit:

- store each **thread_id** in a array of **pthread_t** and initialize this array as **pthread_create** is called.

- create a new loop at the end of the program that performs one **pthread_join** for each thread
- in each **pthread_join** use the **pthread_t** value retrieved when creating the thread

3 Getting thread return value

Threads are defined as functions that return a pointer (void *) value.

It is possible to encapsulate in it pointers to data allocated (**malloc**) inside the thread or even other 32/64 bit values (such as integers, floats, long doubles).

In **test-threads_2.c** the thread returns a random number, but there is way to retrieve it in main.

```
void * thread_function(void * arg){
    int r = random()%100;
     ...
    return (void *)r;
}
```

After the thread terminates, it is possible to get this value with the help of the **pthread_join** function using the second argument.

This second argument should be a pointer to a **void *** variable that will store the value returned by the thread. This value can later be converted to the correct type:

```
void * thread_ret;
pthread_join(thread_id, &thread_ret);
printf("%d", (int) thread_ret);
```

The following examples show how threads can return values of various types: int, string, structures.

Integer	Float

`void * thread_function(void * arg){`
`  int * r_int;`
`  r_int = malloc(sizeof(int));`
`  * r_int = 222;`
`  return (void *) r_int;`
`}`
	`void * thread_function(void * arg){`
	`  float *r_float;`
	`  r_float = malloc(sizeof(float));`
	`  * r_float = 1.5;`
	`  ...`
	`  return (void *)r;`
	`}`

Integer	Float

void * thread_ret;
int * ret_val;
pthread_join(thread_id, &thread_ret);
printf(“%d”, *retval);
free(ret_val);
	void * thread_ret;
	float *ret_val;
	pthread_join(thread_id, &thread_ret);
	printf(“%f”, *retval);
	free(ret_val);

String (array of characters)	Structure

void * thread_function(void * arg){
char * r_str;
r_str = calloc(len, ...)
...
return (void *)r_str;
}
	void * thread_function(void * arg){
	struct_data * r_s
	r_s = malloc(….)
	. . .
	return (void *)r_s;
	}

void * thread_ret;
char *ret_val;
pthread_join(thread_id, &thread_ret);
ret_val = **(char *)** thread_ret;
...
	void * thread_ret;
	struct_data * s_val;
	pthread_join(thread_id, &thread_ret);
	ret_val = **(struct_data *)** thread_ret;
	...

3.1 Exercise 4 – get returned value

Modify exercise 3 so that after each thread terminates the value returned by each thread is printed. This print should be done right after each **pthread_join**.

Hints:

- declare a **void *** variable in the main
- Pass such variable by reference (using the &) as the second argument of each **pthread_join**
- print the value using a **(int)** cast

4 Send values to the threads

The function that defines a thread has one **void *** argument. This argument can point to any data that was allocated before the **pthread_create**. And becomes accessible inside the thread.

Before calling the **pthread_create** the data should be allocated and sent into the thread using the last argument of the **pthread_create**. Inside the thread it then becomes possible to access the allocated data.

The next examples show how to send an integer or a string (array of characters) into a thread.

Integer	string

`int * thr_val = malloc(sizeof(int))`
`*thr_val = 14;`
`pthread_create(&thread_id[i], NULL,`
`thread_function, thr_val);`
`....`
	`char * thr_str = calloc( . . . )`
	`….`
	`pthread_create(&thread_id[i], NULL,`
	`thread_function, thr_str);`
	`....`

`void * thread_function(void * arg){`
`int int_val = *(int *) arg;`
`free(arg);`
`....`
	`void * thread_function(void * arg){`
	`char * str = (char *) arg;`
	`...`

if it is necessary to send more values it is possible to create a structure that will contain all those values.

4.1 Exercise 5 – Send data into threads.

Modify exercise 4 so that the delay each thread waits inside the sleep function is sent by the main. This delay should be between 0 and 4 seconds.

Hints:

- declare a pointer to **int** (**int ***) in main;
- before each **pthread_create** do a **malloc** and assign a random number (between 0 and 5);

- modify the **pthread_create** so that the last argument receives the pointer previously allocated and initialized;
- inside the thread declare a integer and copy there the value pointed by the thread argument;
- modify the value inside sleep.

4.2 Exercise 6 – bad way to send data into threads.

The **test-threads_3.c** is supposed to send to each thread the **i** variable, starting in 0 for the first thread:

```
int i = 0;
 while( i < n_threads) {
     pthread_create(&thread_id, NULL,thread_function, &i);
     i++;
 };
```

Eacjh thread should receive a different value of I, but this does not happen.

Execute the program various times with various numbers of threads and try to explain what is happening.

5 Shared data

Although each thread has his own variables, private to it and not accessible by any other thread, it is possible for multiple threads to access global variables.

Observe the **test-threads_4.c** program supplied, compile it and execute it with different input values. After running it a few times, answer the following questions:

- what happens to the variable **n**?
- Where is variable **n** declared?

5.1 Exercise 7

Modify **test-threads_4.c** so that every thread iterates 10 times as in the other programs and the **n** value finishes with the value of **n_threads*10**.

6 Exercise 8

The provided program (**exercise-8-serial.c**) generates an array with a lot of random number and counts how many multiples of 2, 3, 5, 7 and 11 there are in such array (by calling the **n_mult** function 5 times).

Parallelize the program creating 5 threads. Each thread receives one of the previous numbers (2, 3, 5, 7 or 11), calculates how many multiples of the argument there are in the array, and returns such value.