Lab 3: POSIX Threads

Master M1 MOSIG – Université Grenoble Alpes (Ensimag & UFR IM2AG)

2022

In this lab, we are going to manipulate our first multi-threaded programs. In a first step, we are going to observe the behavior of some provided pieces of codes. In a second step, we are going to write our own programs. To this end, we will use the POSIX thread (pthread) interface.

1 Instructions

This lab is not graded. Try to answer the questions and do not hesitate to ask for explanations. Working on the exercises proposed in this lab should help you better understand the lectures on thread synchronization.

A brief description of the POSIX thread (pthread) interface is provided at the end of this document. For more details, look at the corresponding man pages.

2 Before jumping into the code ...

Threads have been introduced during Lecture 4. Try to answer the following questions related to this lecture. (Take a few minutes to review the corresponding slides if need be.)

Question 2.1: *How many processes were created by each of the programs we have considered in the previous labs (on memory allocation and virtual memory)?*

Question 2.2: *How many threads were created by the programs we have considered in the previous labs (on memory allocation and virtual memory)?*

Question 2.3: *Are the following statements true of false?*

a) *The concept of thread refers to a flow of instructions that can be scheduled independently.*

b) *A process can contain multiple threads.*

c) *Threads are a mechanism that permits a program to perform multiple tasks concurrently.*

d) *Threads exist to make life of students harder.*

Question 2.4: *One of the purposes of threads is performance. Explain in a few words why performance can be improved when running a multi-threaded program on a multiprocessor machine.*

Question 2.5: *I have designed a program that helps users downloading files from remote servers on the Internet. My program comprises a single thread that reads the user command from* `stdin`*, and then, executes the command (that is downloading the requested file from the server). Users complain that they cannot enter a new command as long as the previous file has not been fully downloaded. Can threads help me solve the problem? If yes, how?*

Question 2.6: *Are the following statements true or false?*

a) *All the threads of one process share the same virtual address space.*

b) *All the threads of one process share the same stack.*

3 A first multi-threaded program

We consider the program defined in the provided file `match.c`. The following line should be used for compilation:

```
$ gcc -o match match.c -lpthread
```

This program simulates the behavior of rugby supporters attending a match and singing. Each supporter is modeled by a thread. The program takes two input parameters: the number of supporters for team 1 and team 2.

Try to run the program and observe what happens. Then, answer the following questions:

Question 3.7: *What happens on a call to* `pthread_create()`*? Give a detailed answer.*

Question 3.8: *What can you say about the order of the messages printed by the supporter threads?*

Question 3.9: *If I run the program with the following parameters "*`./match 2 2`*", how many threads run in the context of the created process in total?*

Question 3.10: *What is the purpose of the calls to* `pthread_join()` *at the end of the* `main`*? What would happen without these calls?*

Question 3.11: *What is the purpose of the* `tids` *variable in the* `main` *function?*

4 Parameter Passing

The `pthread_create` function takes a single `void *` pointer argument. In the previous example, the arguments were the lyrics sung by the supporters.

The goal here is to write a new version of the program that takes as an input the lyrics of the song and the number of times a supporter is going to repeat the song (we presume that an English supporter is eagerer than a French one ...).

To do so, you need to pass two parameters to the threads but the `pthread_create` specification allows you to pass only one. In order to overcome this limitation, you need to create a structure containing two fields and pass a pointer to the structure.

Implement the new version of program `match` that we will call `matchp`. The program `matchp` takes 4 parameters as input:

1. The number of French supporters
2. The number of times they repeat their song
3. The number of English supporters
4. The number of times they repeat their song

5 A disappointing multi-threaded program

We are now considering the program defined by `counting.c`. This *dummy* program has multiple threads sharing an integer variable `result` initialized to 0. Each thread increases the value of `result` by 1 during `N_ITER` iterations. By default `N_ITER` is set to 1000000. Running with N threads, the final value of `result` should be $N \times N\_ITER$. The program takes one input parameter: the number of *counting* threads to run.

Run the program with different numbers of threads (going from 1 to a few) and observe the obtained result.

Question 5.12: *Do we get the expected results?*

Question 5.13: *Try to explain what happens.*

6 Dependencies between threads

Write a program that executes threads T_i ($i \in [0..9]$) according to the dependencies described in Figure 1. In this graph, an arrow from thread T_a to thread T_b means that thread T_b starts only when T_a is terminated. If a thread depends on several threads, it can only start when all threads it depends on have terminated. Also, all threads appearing on the same horizontal dashed line should be executed concurrently.

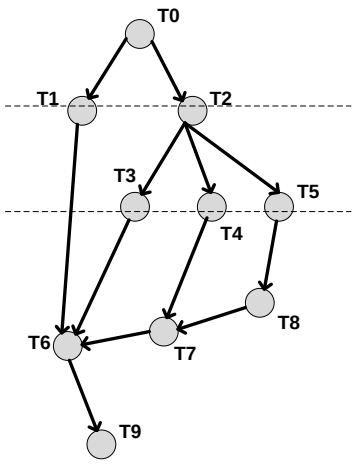

Figure 1: Dependencies between threads

Note: To check that your implementation has the expected behavior, each thread should display a message including its identifier i when it starts executing and another message just before terminating. In between, make it sleep for a small random number of seconds.

Question 6.14: *Implement the multi-threaded program that will run the threads according to the dependencies defined in Figure 1.*

7 Parallel sum on a vector

In this part, we are going to study how to compute the sum of the elements of a vector using multiple threads.

Before going to a multi-threaded version, it is good to start by implementing a sequential version of the code (that is, a single-threaded version of the code).

To do so, implement a function `int sum(int *vect, int size)` such that:

- `vect` is a pointer to the vector to sum.
- `size` is the size of the vector to sum.
- returns the result of the summation.

To test your program, you are provided with a tool to generate a file containing a random sequence of integers ranging from 0 to 9. The tool takes a single parameter `X` which is the size of the generated file in MB. The name of the generated file is `/tmp/random_int_XMB.data`. Your program should `mmap` the content of the generated file in memory and compute the sum of the included values.

Question 7.15: *Implement the sequential version of the program.*

Comments about mmap()

The `mmap` system call (see "`man 2 mmap`") can be used to map files in the virtual address space of processes. Among the capabilities offered by this system call, it allows a process to read and write into a file through virtual memory accesses.

Tips for this exercise:

- We can find an example of usage of `mmap` in `generate_random_file.c`.
- Warning: Take some time to understand the meaning of the parameters of `mmap()` to select the appropriate ones for your program.

7.1 Using return values

We would like to implement a multi-threaded version of this program. The idea is to have each thread computing the sum for a sub-part of the vectors and then to sum the result of each sub-part. To make things simpler, we will only consider cases where $vector\_size \bmod nb\_threads = 0$.

The `pthread_join` function allows you to get the return value of the function executed by a thread.

Question 7.16: *Write a multi-threaded version of the program using return values to get the result of the partial summations. You should re-use the `sum` function defined in the sequential version of the code. Compare the result against the sequential version to check for correctness.*

7.2 Using a shared variable

Another way to get the result of the computation used by each thread is to have an array shared between all threads, where each thread is going to write the result of its partial sum.

Question 7.17: *What should you check before computing the final sum?*

Question 7.18: *Write a new version of the multi-threaded program using a globally shared array.*

7.3 Performance evaluation

Finally, we would like to see whether the parallel versions of the program perform better than the sequential version. To do so, we suggest you to generate a rather large file (in the order of a few hundreds of megabytes).

Using the tool `time`[1], compare the execution time using different numbers of threads.

Question 7.19: *What do you observe? Explain.*

[1] `time ./my_exec`

8 Collaborating to count (Bonus)

Implement a program that will create 3 threads T_i ($i \in [0..2]$) that should collaborate to generate a sequence of numbers from 0 to 100. By *"collaborate"*, we mean that each thread will display one number in the sequence alternatively. To check that your implementation is correct, include the identifier of the corresponding thread in each message that is generated.

Note: In this exercise, you are only allowed to execute read and write operations on the shared memory to synchronize threads.

Question 8.20: *Implement the multi-threaded program that will generate the sequence of numbers according to the specification.*

9 The POSIX Threads Library (Pthread)

POSIX (Portable Operating System Interface) defines a standard thread interface.The primitives can be consulted using man (`man pthread`). Here are some basic primitives for thread manipulation.

- **Thread creation**: `pthread_create`

```
int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attr,
                    void*(*start_routine)(void *),
                    void *arg)
```

  - `pthread_t` : thread type
  - `pthread_t *thread` : after a successful creation, the first argument contains the thread identifier.
  - `const pthread_attr_t *attr` : We will ignore these attributes that may be used to configure the scheduling strategy and the thread priorities.
  - `void*(*start_routine)(void *)` : the third argument gives the function that the thread should execute.
  - `void *arg` : the fourth argument is the argument to pass to the function to be executed by the thread.

- **Thread termination**: `void pthread_exit (void *status);`

  Terminates the thread and gives a return value in `status`.

- **Wait for a thread to terminate**: `int pthread_join(pthread_t th, void ** status);`

  Wait of the termination of the thread `th` and store the return value in `status`. The thread should not be detached (see `pthread_detach`).

- **Releasing the CPU**: `int pthread_yield(void)`

- **Thread identification**: `pthread_t pthread_self (void);`

- **Sending a signal to a thread**: `int pthread_kill(pthread_t th, int sig);`

  Sends the signal sig to thread `th`.

9.1 Example

```
#include <pthread.h>
#include <stdio.h>
void *routine (void *arg)
{
    int *status = malloc (sizeof(int)); /* To receive the return status */
    printf ("Arg = %d\n", *(int *)arg); /* Necessary cast to (int *) */
    *status = *(int *)arg * 2;
    pthread_exit (status);
}

int main()
{
    pthread_t pth;
    int err, arg = 3;
    int *res;
    err = pthread_create (&pth, NULL, routine, &arg);
    if (err != 0) fprintf(stderr,"Failed to create a thread: %d\n",err);
    pthread_join (pth, (void **)&res);
    printf ("Result: %d\n", *res);
    free (res);
    exit(0);
}
```