UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería Informática Grado en Ingeniería de Computadores

Practice 4: Thread programming in Linux

Operating Systems

Contents

1 Learning outcomes (competences) 3

2 Introduction 3

3 Thread programming 3

4 Basic POSIX services for thread management 4
4.1 POSIX service `pthread_create()` 4
4.2 POSIX service `pthread_exit()` 5
4.3 POSIX service `pthread_join()` 5
4.4 POSIX service `pthread_self()` 5

5 Thread synchronization 6
5.1 POSIX threads synchronization. 6
5.2 POSIX service `pthread_mutex_init()` 6
5.3 POSIX service `pthread_mutex_destroy()` 7
5.4 POSIX service `pthread_mutex_lock()` 7
5.5 POSIX service `pthread_mutex_unlock()` 7

6 Compiling a POSIX thread program 7

1 Learning outcomes (competences)

1. Make the difference between (heavy) processes and light processes or *threads*.
2. Understand how do POSIX services works to manage threads (creation, synchronization and termination).
3. Apply different services to solve a given problem.
4. Develop thread applications in the C programming language,
5. Understand where and when race conditions might arise between cooperating threads when accessing shared resources.
6. Understand the concept of mutual exclusion when accessing shared resources.
7. Apply the appropriated POSIX services to provide safe access to shared resources.

2 Introduction

The processes introduced in practice 3 were independent from each other; they have their own memory map. This guarantee that a process can't not alter another process in any way, but as a drawback it makes difficult to interchange information between processes. Inter-process communication mechanisms must be provided by the kernel so processes can cooperate towards a goal. This communication need happens quite often, and use use pipes, clipboard, client-server, sockets, etc. Threads, or light-weight processes were introduced to overcome (among others) these problems.

A thread can be defined as a basic execution entity, that owns only a program counter, CPU register and its own stack. To be able to run, a thread must belong to a higher-level entity called *task*. Task can be thought as containers of threads. The task itself does not have execution capability, its threads have. Each thread will execute a given function, while the task contains all the resources, like memory, files, and of course the code to be executed by the threads. Therefore, all these resources are shared by all the threads in the task: code, data, *heap* [1], making communication straightforward.

The goal of this practice is to introduce the student in the thread programming in Linux, to understand and apply the POSIX API interface for thread management (creation, wait and exit). Another goal is that the student need to realize that concurrent access to a shared resource must be carefully controlled, providing for this the appropriated synchronization mechanisms. In this practice we will focus on the mechanisms provided by the operating systems, applying the functions provided by the POSIX standard that we already studied in the previous practice.

3 Thread programming

To program threads we have two possibilities:

1. Use a conventional programming language, for example C, using system calls (kernel-level threads) or library calls (user-level threads).
2. Use a programming language that provides concurrent execution support, such as Ada or Java. In this case, is the own language the one that will provide threads and synchronization services, possibly making system or library calls to the underlying operating system.

[1] *heap*Temporally storage area that can be requested and released dynamically with the call to the functions `malloc` and `free`

According to the Operating System teaching guide, this practice will be focused on thread programming using the C programming language, using the POSIX library IEEE Std 1003.1c-1995, also known as POSIX threads or pThreads. Therefore, since this library is platform-agnostic, any program that adheres to it will be easily portable to any other POSIX-compliant system (like Windows or Mac OS)

To make the compiler aware of the use of POSIX calls to program threads, we need to include the header file `pthread.h` with the following preprocessor directive:

```
#include <pthread.h>
```

The header file `pthread.h` includes the declaration of several functions that will allow us to create and manage threads. The next table summarizes some of these functions:

Function	Description
`pthread_create()`	Creates a new thread
`pthread_exit()`	Finish the execution of the calling thread
`pthread_join()`	Waits for a thread to finish
`pthread_t pthread_self()`	Returns thread information with a `pthread_t structure`
`pthread_mutex_init()`	Creates a mutex lock semaphore and initializes it
`pthread_mutex_destroy()`	Destroys a mutex lock semaphore
`pthread_mutex_lock()`	Tries to lock a mutex lock semaphore
`pthread_mutex_unlock()`	Unlocks a mutex lock semaphore

4 Basic POSIX services for thread management

In these section the following POSIX services are described so they can be used in this practice:

1. Thread creation.
2. Thread ending.
3. Wait for thread ending.
4. Thread identification.

4.1 POSIX service `pthread_create()`

The POSIX service `pthread_create()` creates a thread that is ready for execution, therefor, the newly created threads competes for the CPU with its creator threads from that precise moment. It is up to the CPU scheduler to determine which thread will execute in a given moment. Unlike processes, there are no parenting relationship between threads, and they can be created by any thread in the system. There is not such a thing as a *main thread*. If the function completed successfully, it return 0, and a different value otherwise.

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

Parameters

thread If the call is successful, this variable holds the thread identifier. This identifier is necessary to perform further operations on the thread.

`[attr]` It contains the parameters of the thread. It can be set with the corresponding POSIX function, for example to define its priority, the scheduling policy, etc. If it is set to `NULL`, then the thread is created with default parameter values.

start_routine Is a pointer to the function the thread will run. This function must comply with two rules: it must return `void*` and may have only one argument and this argument have to be also a `void*`. If special typing for the arguments is needed, we can use the *casting* mechanism provided by the C language. For example, we can declare a structure `s_data`, pass it as a *null pointer* with `(void*)s_data` and then cast it back inside the function with `(s_data*)`.

arg Is a pointer to the parameter to the function that the thread will execute. It can be `NULL`.

4.2 POSIX service `pthread_exit()`

The POSIX service `pthread_exit()` terminates the execution of the running thread. The execution also finishes when the thread reaches the last instruction of its function. This service does not return any value.

```
void pthread_exit(void *status);
```

Parameters

status Is a `(void*)` to the results that the thread would like to communicate to another thread that is waiting for it to finish. In other words, to the thread that executes the function `pthread_join()` with the ID of the finishing thread.

4.3 POSIX service `pthread_join()`

This service suspends the execution of the calling thread until the thread with the ID passed as argument finish its execution. This service returns the value given by the thread when it finished if any. For this function to work, the thread must be in *joinable* state. [2] This function return zero if it executes correctly and a different value otherwise.

```
int pthread_join(pthread_t thread, void **status);
```

Parameters

thread ID of the thread to wait for.

status It is a pointer to a valid memory zone where the finishing thread will store its return value (the one that it specified when it executed the function `pthread_exit()`.). It `status` is NULL, then the return value is discarded.

4.4 POSIX service `pthread_self()`

It returns the ID of the calling thread. Its syntax is as follows:

```
pthread_t pthread_self(void);
```

[2] A thread is *joinable* by default. This implies that upon termination, its resources are not freed (thread ID and stack) until another thread executes `pthread_join` with its ID.

5 Thread synchronization

As already seen in the introduction, cooperant threads share their task's addressing space and, therefore, they can freely access global variables and heap. However, concurrent access to shared variables must be carefully synchronized to avoid race conditions.[3].

5.1 POSIX threads synchronization.

The *pthread* library provides, among others, *mutex semaphores* to deal with multi-thread synchronization problems.

The *mutex*, *mutex lock* or just *lock* can be used to define a mutual exclusion code segment, that is, a part of the code that can be executed by only one thread at the same time. It a thread tries to execute a mutual exclusion code segment while another thread is already executing it, then it must wait until the second exits the mutual exclusion segment.

The *mutex* variable are of type `pthread_mutex_t`. Before using the variable it must be initialized to either open or close state. The *mutexes* are generally global or static variables that are widely accessible by all the threads. They work as a regular lock, with two operations: lock or unlock. We say that a thread owns a *mutex* when it could lock it up successfully. The behaviour of the *mutex lock* to provide mutual exclusion is as follows:

- A *mutex* created and initialized as open (and therefore unowned)
- A thread that wants to access the mutual exclusion code segment tries to lock the mutex up.
  - If the *mutex* was opened (unowned), then it it locked and becomes owned by the locking thread.
  - It the *mutex* was closed (owned by another thread), then the execution of the calling thread is suspended until the *mutex* is unlocked by its owner.
- A thread that leaves the mutual exclusion code unlocks the *mutext*. If there was suspended threads waiting for that mutex, then one of them is waken up, completes the pending locking operation and becomes the new owner of the lock, proceeding to the mutual exclusion code segment alone.

5.2 POSIX service `pthread_mutex_init()`

The POSIX service `pthread_mutex_init()` initializes a *mutex* prior to its usage. If the function is successful it returns zero, and another value if otherwise. Its syntax is as follows:

```
int pthread_create(pthread_mutex_t *mutex, const pthread_mutexattr_t
*attr);
```

Parameters

mutex It is a pointer to a `pthread_mutex_t` variable to be initialized.

attr It is a pointer to a `pthread_mutex_attr_t` structure used to define the kind of mutex semaphore to initialize. If this argument is `NULL` the POSIX library will apply default values.

Before initializing the *mutex* variable, it has first to be created. Usually this is done as a global variable, with default attributes using the macro `PTHREAD_MUTEX_INITIALIZER` as follows:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

[3]Race conditions will be explained in detail in theory or lab classes

5.3 POSIX service `pthread_mutex_destroy()`

This service destroys a *mutex*. This means that this element is no longer going to be use and therefore all its resources can be freed. It returns zero if it executed correctly and a value different than zero if otherwise. It syntax is as follows:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Parameters

mutex It is a pointer to a `pthread_mutex_t` type variable that represents the *mutex* to be destroyed.

5.4 POSIX service `pthread_mutex_lock()`

This service tries to lock the *mutex* given as a parameter. It returns zero if it executed correctly and a value different than zero if otherwise. Its syntax is as follows:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Parameters

mutex It is a pointer to a `pthread_mutex_t` variable that represents the *mutex* to be locked.

If the *mutex* was open or unlocked, then it is locked and the calling thread becomes its owner. Conversely, if the *mutex* was closed then the calling thread execution is suspended.

5.5 POSIX service `pthread_mutex_unlock()`

The POSIX service `pthread_mutex_unlock()` unlocks the *mutex* given as an argument. It returns zero if it executed correctly and a value different than zero if otherwise. Its syntax is as follows:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Parameters

mutex It is a pointer to a `pthread_mutex_t` variable that represents the *mutex* to be locked.

If when unlocking a *mutex* there are suspended threads waiting to own the lock, the one with the highest priority becomes the new owner and its execution is resumed.

6 Compiling a POSIX thread program

In Linux the POSIX Threads library is called `libpthread` and it is present in most of the Linux distributions. Normally it is installed along with the development tools.

To successfully compile a program that makes use of the library's services it is necessary to specify that this library has to be linked. This is done using the `-lpthread` as follows:

```
gcc programa.c -o programa -lpthread
```

Exercise

To exercise the contents learned about POSIX threads, you must complete an application that will simulate a car race, using C language and the Linux operating system. In this simulation each car will be modeled with a thread that delays its execution with a random value.

The thread function receives an structure as argument. This structure includes an arbitrary string that will be use during the simulation to identify the car, and a numerical `id` that will range from 0 to `n_coches-1`.

The thread that creates all the car-threads will wait for all of them to finish. When this happens they will return their `id` so they can be identified in the race ranking.

The next incomplete code is given as a starting point for this exercise. You must complete it and answer to the questions that you will find at the end.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #define n_coches 8
6
7  // Array para los identificadores de los hilos
8  pthread_t tabla_hilos[n_coches];
9
10 // Tipo de datos hilos_param
11 typedef struct {
12     int id;
13     char *cadena;
14 } hilos_param;
15
16 // Array de datos de tipo hilos_param
17 hilos_param parametros[n_coches];
18
19 // Funcion ejecutada por los hilos
20 void *funcion_coches(hilos_param *p)
21 {
22     int aleatorio;
23     printf("Salida␣%s␣%d\n", p->cadena, p->id);
24
25     fflush ( stdout );
26     aleatorio = rand();
27     sleep(1 + (aleatorio%4));
28
29     printf("Llegada␣%s␣%d\n", p->cadena, p->id);
30
31
32     /* CODIGO 2 */
33
34
35 }
36
37 int main(void)
38 {
39     int i, *res;
40
41     printf("Inicio␣proceso␣de␣creacion␣de␣los␣hilos...\n");
42
43     // Se procede a crear los hilos
44
45     for (i=0; i<n_coches; i++)
46     {
47
48         /* CODIGO 1 */
49
50     }
51
52     printf("Proceso␣de␣creacion␣de␣hilos␣terminado\n");
53     printf("SALIDA␣DE␣COCHES\n");
54
55
56
57     for (i=0; i<n_coches; i++)
58     {
59
60         /* CODIGO 3 */
61
62     }
63
64     printf("Todos␣los␣coches␣han␣LLEGADO␣A␣LA␣META␣\n");
```

```
65      return 0;
66  }
```

1. Complete the code in line 52 with the appropriated POSIX call.
2. Complete the code in line 36 with the appropriated POSIX call.
3. Complete the code in line 64 with the appropriated POSIX call.
4. What is the call to the function `rand()` in `function_coches()` for?
5. What happens if the thread that creates the other threads finishes its execution without waiting for the others?
6. Declare a global variable `suma` shared by all the thread cars. Every time that a car computes its random value, this value has to added to the `suma` variable. At the end, this value has to be printed by the main thread. May something go wrong with this calculation? Explain your answer.
7. *Optional exercise*: Introduce the appropriated changes in the code to address the problem found in the previous question.