UNIVERSIDAD DE ALCALÁ

Departamento de Automática

*Grado en Ingeniería Informática*

# Ejercicio opcional: Servicios POSIX para la gestión de hilos en Linux

Sistemas Operativos

# Índice

# 1. Competencias asociadas al ejercicio opcional

1. Ser capaz de comprender la diferencia entre procesos pesados y procesos ligeros (hilos o *threads*).
2. Ser capaz de comprender el funcionamiento de los servicios POSIX para la gestión básica de hilos (creación, espera y finalización).
3. Ser capaz de aplicar los diferentes servicios POSIX para la gestión básica de hilos (creación, espera y finalización).
4. Entender cuándo, por qué surgen y cómo se pueden resolver los problemas de condiciones de carrera (*race conditions*) entre procesos/hilos *cooperantes* en el acceso no controlado a recursos compartidos y su relación con el concepto de exclusión mutua.
5. Ser capaz de aplicar los servicios POSIX adecuados para que hilos cooperantes accedan a variables compartidas evitando condiciones de carrera y garantizando la exclusión mutua en el acceso.
6. Ser capaz de expresar con claridad razonando sobre aspectos relativos a la ejecución de la simulación con hilos planteada en el ejercicio y sobre el funcionamiento de los servicios POSIX básicos de hilos.
7. Ser capaz de desarrollar aplicaciones simples en lenguaje de programación C que permitan manejar hilos de ejecución.

# 2. Introducción

Los procesos pesados, analizados en la práctica 3, se caracterizan por ser independientes; poseen sus propios mapa de memoria. Esta característica aporta como ventaja principal la garantía de inexistencia de intromisiones de un proceso en el espacio de memoria de otro, pero, como desventaja, dificulta el intercambio de información entre procesos forzando a utilizar alguno de los mecanismos de comunicación entre procesos proporcionados por el sistema operativo para lograr la cooperación entre procesos. Sin embargo, la necesidad de colaboración entre procesos y, por tanto, de comunicación entre ellos, surge en muchas ocasiones cuando se ejecutan aplicaciones (ejecución de tuberías, intercambio de información entre aplicaciones a través del portapapeles, intercambio de información en aplicaciones de tipo Cliente/Servidor, etc.). Los hilos o procesos ligeros (*threads*) surgieron como solución a esta problemática.

Un hilo se puede definir como una entidad básica de ejecución que simplemente posee su contador de programa, los registros del procesador y la pila. Para poder ejecutarse, un hilo debe pertenecer a una entidad de mayor nivel denominada tarea (*task*) sin capacidad de ejecución (se ejecutan sus hilos, denominados *hilos cooperantes*, no la tarea). En definitiva, existen varias “entidades de ejecución activas” (los diferentes hilos que ejecutarán una función asociada a ellos) en una misma aplicación. La tarea sólo posee los recursos (memoria, archivos, etc.) compartidos por sus hilos. La compartición

entre los hilos cooperantes del espacio de direccionamiento perteneciente a su tarea (código, datos globales y área de *heap*[1]) facilita el acceso a la información común y, por tanto, la cooperación entre ellos.

El objetivo de este ejercicio es introducir al alumno en la aplicación de servicios POSIX de gestión de hilos; comprender y ser capaces de aplicar las funciones API de POSIX básicas utilizadas para el manejo de hilos (creación, espera y finalización). Otro objetivo primordial es que el alumno sea capaz de entender la necesidad de controlar el intento de acceso simultáneo a recursos compartidos por varios procesos (denominados *procesos cooperantes*) en un escenario concurrente y, en consecuencia, la necesidad de proporcionar mecanismos que permitan el acceso sincronizado de esos procesos a dichos recursos. En este sentido, nos centraremos en la aplicacióon de algunas de las funciones principales de sincronización de hilos del estándar POSIX.

# 3. Programación con hilos

Para programar con hilos existen básicamente dos alternativas:

1. Utilizar un lenguaje de programación convencional, por ejemplo, lenguaje C, utilizando llamadas al sistema (hilos a nivel de núcleo) o funciones de biblioteca (hilos a nivel de usuario).

2. Utilizar construcciones (o clases) pertenecientes a alguno de los lenguajes con soporte nativo de hilos como Ada o Java. En este caso, es el compilador el que traduce esas construcciones a las correspondientes llamadas al sistema o funciones de biblioteca para el soporte de hilos. Normalmente estos lenguajes ya contienen construcciones integradas dentro de los mismos para dar soporte a hilos, como son los métodos *synchronized* de Java.

De acuerdo al contexto de esta asignatura, Sistemas Operativos, el ejercicio se enfocará a la programación con hilos utilizando lenguaje C y el API para hilos POSIX Threads, aprobado en 1995 y especificado por el estándar formal internacional POSIX 1003.1c-1995.

Para crear y gestionar hilos de ejecución, el estándar POSIX define una biblioteca que proporciona las funciones necesarias para crear y gestionar hilos adecuadamente. Esta biblioteca se denomina *POSIX Threads* o *pthreads*. Al ser un estándar, los programas desarrollados con ella se podrán portar a cualquier sistema operativo POSIX que soporte hilos.

Para gestionar hilos con POSIX es necesario incluir en los programas en lenguaje C el archivo de cabecera `<pthread.h>` mediante la siguiente directiva:

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

El archivo de cabecera `pthread.h` incluye la declaración de determinadas funciones que permiten la creación y gestión de hilos. En la Tabla 1 se presentan algunas de estas funciones y en posteriores secciones se proporciona su descripción.

[1] *Heap*: área para asignación de memoria dinámica; su tamaño crece y decrece con la invocación, por ejemplo, en lenguaje C a funciones del tipo `malloc()` y `free()` respectivamente.

| Función | Descripción |
|---|---|
| `pthread_create()` | Crea un nuevo hilo |
| `pthread_exit()` | Termina la ejecución del hilo que la invoca |
| `pthread_join()` | Espera a la terminación de un hilo |
| `pthread_t pthread_self()` | Devuelve al hilo que lo invoca su identificador |
| `pthread_mutex_init()` | Inicializa un semáforo mutex |
| `pthread_mutex_destroy()` | Destruye un semáforo mutex |
| `pthread_mutex_lock()` | Intenta bloquear un semáforo mutex |
| `pthread_mutex_unlock()` | Desbloquea un semáforo mutex |

Tabla 1: Servicios POSIX de gestión de hilos.

# 4. Servicios POSIX para la gestión básica de hilos

En esta sección se describen los siguientes servicios proporcionados por POSIX, entre otros, para el manejo de hilos de este ejercicio:

1. Creación de hilos.
2. Terminación de un hilo.
3. Espera a la terminación de un hilo.
4. Identificación de hilos.

## 4.1. Servicio POSIX `pthread_create()`

El servicio POSIX `pthread_create()` permite crear inmediatamente un hilo en estado preparado, por lo que el hilo creado y su hilo creador compiten por la CPU según la política de planificación del sistema. Puede ser invocada por cualquier hilo del proceso, no sólo por el "hilo inicial" -hilo que ejecuta la función `main()`-. Si la función se ejecuta correctamente, retorna el valor 0. Si, por el contrario, la función falla, retorna un valor distinto de 0. Su sintaxis es la siguiente:

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

**Parámetros**

**thread** si la llamada tiene éxito, esta variable contiene el identificador del hilo necesario para realizar posteriormente acciones sobre él.

**attr** es el atributo que contiene las características del hilo creado y que se puede inicializar previamente con el correspondiente servicio POSIX para asignar al hilo los atributos que queramos (por ejemplo: prioridad, algoritmo de planificación asociado, etc.). Si se especifica `NULL`, la biblioteca le asignará al hilo unos atributos por defecto.

**start_routine** función que ejecutará el hilo. La función devuelve un puntero genérico (`void *`) como resultado, y tiene como único parámetro otro puntero genérico. Los punteros genéricos tienen como ventaja que podemos devolver o pasar, respectivamente, un dato de cualquier tipo realizando los *castings* necesarios en lenguaje C (por ejemplo, se puede crear una estructura para incluir un conjunto de diferentes datos y devolver/pasar la dirección de esa estructura como único parámetro). Desde una óptica práctica, la creación de un hilo nuevo implica la ejecución concurrente de la función indicada en este parámetro.

**arg** puntero al parámetro que se le pasa a la función que ejecutará el hilo. Puede ser `NULL` si no se le quiere pasar nada a la función.

## 4.2. Servicio POSIX `pthread_exit()`

El servicio POSIX `pthread_exit()` finaliza la ejecución del hilo que invoca esta función. La terminación de un hilo también tiene lugar cuando finaliza la ejecución de las instrucciones de su función. La función no devuelve ningún valor. Su sintaxis es la siguiente:

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

**Parámetros**

**status** puntero genérico a los datos que se desean devolver como resultado. Estos datos pueden ser posteriormente examinados por otro hilo que realice una invocación con éxito a la función `pthread_join()` con el identificador de hilo que ha finalizado.

## 4.3. Servicio POSIX `pthread_join()`

El servicio POSIX `pthread_join()` suspende la ejecución del hilo que la invoca hasta que el hilo cuyo identificador es pasado como parámetro a esta función no termine su ejecución. El valor devuelto por este hilo puede ser recogido por `pthread_join()`. Este hilo debe estar en estado sincronizable (*joinable state*)[2]. La espera por la terminación de un hilo para el cual ya hay otro hilo esperando por él genera un error. Si la función se ejecuta correctamente, retorna el valor 0. Si, por el contrario, la función falla, retorna un valor distinto de 0. Su sintaxis es la siguiente:

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

**Parámetros**

**thread** identificador del hilo por cuya terminación se está esperando al invocar a esta función.

**status** si `status` no es `NULL`, el valor o resultado devuelto por el hilo con identificador `thread` cuando finaliza (valor del argumento en la función `pthread_exit()`) se almacena en la dirección indicada por `status`. Si se especifica `NULL`, indica que no interesa ese valor.

[2]Un hilo está en estado *joinable* por defecto. Este estado implica que, cuando termina el hilo, no se liberan sus recursos (descriptor de hilo y pila) hasta que otro hilo espere por él.

### 4.4. Servicio POSIX `pthread_self()`

El servicio POSIX `pthread_self()` devuelve el identificador del hilo que lo invoca . La función siempre se ejecuta correctamente. Su sintaxis es la siguiente:

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

## 5. Sincronización de hilos

Como se ha visto previamente en la introducción, los hilos cooperantes comparten el espacio de direccionamento de la tarea, por lo tanto, el acceso al área de variables globales y al área de heap es compartido por todos ellos. Sin embargo, el acceso a esta información común debe hacerse de forma controlada para evitar resultados inconsistentes provocados por la aparición de condiciones de carrera (*race conditions*)[3].

### 5.1. Sincronización de hilos con POSIX

La biblioteca *Pthreads* proporciona, entre otros, los *mutex* como mecanismo para resolver los problemas de sincronización en programas multihilos.

El *mutex* o *mutex lock* se emplea para acceder de forma exclusiva a los recursos (garantizar la *exclusión mutua*), es decir, con ellos se indica que una región específica del código sólo puede ser ejecutada por un determinado hilo al mismo tiempo.

El tipo de una variable *mutex* es `pthread_mutex_t` y el programador debe inicializarla antes de utilizarla para sincronizar hilos (generalmente, son variables estáticas accesibles por todos los hilos de una tarea -o proceso-).

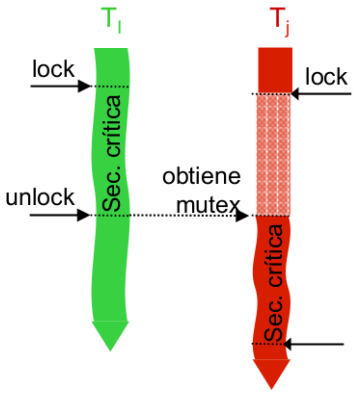
Los *mutex* funcionan como un cerrojo con dos operaciones básicas de acceso: apertura y cierre. De este modo, se puede decir que posee dos estados posibles internos: abierto (desbloqueado) y cerrado (bloqueado). Se dice que un hilo es el propietario de un cerrojo de exclusión mutua cuando ha ejecutado sobre él una operación de cierre con éxito. El funcionamiento del *mutex* puede sintetizarse de la siguiente forma:

- Un *mutex* se crea inicialmente abierto y sin propietario.
- Cuando un hilo invoca una operación de cierre:
  - Si el *mutex* estaba ya abierto (por tanto, sin propietario), lo cierra y pasa a ser propietario.
  - Si el *mutex* estaba ya cerrado, el hilo que invoca la operación de cierre se suspende (es decir, espera por el cerrojo o *mutex*).
- Cuando el propietario del *mutex* invoca la operación de apertura del *mutex*:
  - Se abre el *mutex*.
  - Si existían hilos suspendidos (es decir, esperando por el cerrojo), se selecciona uno y se despierta (con lo que este hilo puede volver a cerrar el *mutex* y pasar a ser el nuevo propietario).

[3] Este concepto será explicado en clase o en la sesión de laboratorio

A continuación se muestra cómo se soluciona el acceso al código de una sección crítica utilizando un objeto *mutex* y sus dos operaciones atómicas (o indivisibles) *lock* y *unlock*, respectivamente:

```
lock(mutex);
/* código sección crítica */
unlock(mutex);
```




## 5.2. Servicio POSIX `pthread_mutex_init()`

El servicio POSIX `pthread_mutex_init()` inicializa un *mutex*. Hay que invocarla antes de realizar cualquier operación con el *mutex*. Si la función pudo crear el *mutex*, retorna el valor 0. En caso contrario, retorna -1. Su sintaxis es la siguiente:

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

**Parámetros**

**mutex** es un puntero a un parámetro de tipo `pthread_mutex_t` que representa el *mutex* que se va a inicializar.

**attr** es un puntero a una estructura del tipo `pthread_mutex_attr_t` que sirve para definir el tipo de mutex deseado. Si el valor es `NULL` (recomendado), la biblioteca le asignará un valor por defecto.

Antes de la inicialización de la variable *mutex*, ésta se debe crear. Frecuentemente, se crea como una variable global con unos atributos por defecto utilizando la macro `PTHREAD_MUTEX_INITIALIZER` de la siguiente forma:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

### 5.3. Servicio POSIX `pthread_mutex_destroy()`

El servicio POSIX `pthread_mutex_destroy()` destruye un *mutex*. Invocar este servicio significa que el *mutex* no se va a utilizar más y se puede liberar la memoria ocupada por el cerrojo. Si la función se ejecuta correctamente, retorna el valor 0. Si, por el contrario, la función falla, retorna un valor distinto de 0. Su sintaxis es la siguiente:

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

**Parámetros**

**mutex** es un puntero a un parámetro de tipo `pthread_mutex_t` que representa el *mutex* que se va a destruir.

### 5.4. Servicio POSIX `pthread_mutex_lock()`

El servicio POSIX `pthread_mutex_lock()` bloquea o cierra el *mutex* y continúa con la ejecución. Si la función se ejecuta correctamente, retorna el valor 0. Si, por el contrario, la función falla, retorna un valor distinto de 0. Su sintaxis es la siguiente:

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

**Parámetros**

**mutex** es un puntero a un parámetro de tipo `pthread_mutex_t` que representa el *mutex* que se va a bloquear o cerrar.

Cuando se ejecuta esta función, si el cerrojo está abierto o desbloqueado, se cierra (bloquea) y el hilo invocador de esta función pasa a ser el propietario del *mutex*. Si, por el contrario, cuando se ejecuta `pthread_mutex_lock()` el *mutex* está cerrado, esta operación *lock* suspende al hilo invocador de la función que esperará hasta que otro hilo desbloquee el cerrojo.

### 5.5. Servicio POSIX `pthread_mutex_unlock()`

El servicio POSIX `pthread_mutex_unlock()` libera el bloqueo que se tuviera sobre el *mutex* (lo abre). Si la función se ejecuta correctamente, retorna el valor 0. Si, por el contrario, la función falla, retorna un valor distinto de 0. Su sintaxis es la siguiente:

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

**Parámetros**

**mutex** es un puntero a un parámetro de tipo `pthread_mutex_t` que representa el *mutex* que se va a desbloquear o abrir.

Si cuando se realiza esta operación hay ya hilos suspendidos en espera de poseer este cerrojo o *mutex*, se selecciona el más prioritario y se permite que cierre el *mutex* de nuevo.

# 6. Compilación de un programa con POSIX Threads

La librería de hilos de POSIX Pthreads se llama `libpthread` y viene en la mayoría de las distribuciones de Linux. Normalmente, esta biblioteca se instala con los paquetes incluidos para el desarrollo de aplicaciones (`libc`).

Para compilar un programa en C, `programa.c`, que utilice la biblioteca de hilos con el compilador GNU `gcc`, es necesario enlazar con la biblioteca `libpthread` como se muestra en la siguiente orden:

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

## Ejercicio

Como aplicación de los servicios POSIX de gestión de hilos descritos, debe completar el desarrollo, en lenguaje C y en el sistema operativo Linux, de una aplicación denominada `simula_car`. Esta aplicación simula una carrera de coches mediante la creación de un hilo por coche.

Cada hilo (coche) ejecuta una funcion genérica, `funcion_coche()`, responsable de modelar la salida de un coche de la parrilla de salida y su llegada a la meta mediante sendos mensajes que se mostrarán en pantalla. Además, la función simula la diferente evolución de cada coche en la carrera con la introducción de un retardo aleatorio durante su ejecución. `funcion_coche()` recibe una estructura como argumento formada por una cadena arbitraria y un entero (comprendido entre 0 y `n_coches-1`), datos utilizados conjuntamente durante la simulación para identificar cada coche.

El hilo padre (`main()`) debe esperar a que los coches finalicen la carrera. Cuando llegen todos a la meta, se encargará de mostrar en pantalla la clasificación final.

A continuación se proporciona, como apoyo inicial para el desarrollo de este ejercicio, el código incompleto de `simula_car.c`, que simula el escenario descrito. Complételo y conteste RAZONADAMENTE y CON CLARIDAD a las preguntas que se plantean.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  #define N_COCHES 8
7
8  // Tipo de datos que representa un coche
9  typedef struct {
10     int id;
11     char *cadena;
12 } coche_t;
13
14 // Array de datos de tipo coche_t
15 coche_t Coches[N_COCHES];
16
17
18 // Funcion ejecutada por los hilos
19 void *funcion_coche(coche_t *pcoche)
20 {
21     int aleatorio;
22     unsigned int semilla = (pcoche->id) + getpid(); // semilla generacion num. aleatorios
23
24     printf("Salida de %s %d\n", pcoche->cadena, pcoche->id);
25
```

```
26      fflush (stdout);
27
28      // generar numero aleatorios con funcion re-entrante rand_r()
29      aleatorio = rand_r(&semilla) % 10;
30
31      sleep(aleatorio);
32
33      printf("Llegada de %s %d\n", pcoche->cadena, pcoche->id);
34
35      /* CODIGO 4 */
36
37
38      /* CODIGO 2 */
39  }
40
41
42  int main(void)
43  {
44      pthread_t hilosCoches[N_COCHES]; // tabla con los identificadores de los hilos
45      int i;
46
47      printf("Se inicia proceso de creacion de hilos...\n\n");
48      printf("SALIDA DE COCHES\n");
49
50      for (i=0; i<N_COCHES; i++)
51      {
52
53          /* CODIGO 1 */
54
55      }
56
57      printf("Proceso de creacion de hilos terminado\n\n");
58
59
60      for (i=0; i<N_COCHES; i++)
61      {
62
63          /* CODIGO 3 */
64
65      }
66
67      printf("Todos los coches han LLEGADO A LA META \n");
68
69      /* CODIGO 5 */
70
71      return 0;
72  }
```

1. Como aplicación de los servicios POSIX básicos de gestión de hilos, complete en el programa las líneas que a continuación se indican:

   1.1 Línea 53 indicada mediante el comentario `CODIGO 1`. Inicialice la estructura de cada coche e invoque el servicio POSIX que considere adecuado.

   1.2 Línea 38 indicada mediante el comentario `CODIGO 2`. Invoque el servicio POSIX que considere adecuado.

   1.3 Línea 63 indicada mediante el comentario `CODIGO 3`. Invoque el servicio POSIX que considere adecuado.

2. ¿Qué efecto produce la función `rand_r()` en la función `funcion_coche()` que ejecuta cada hilo?

3. ¿Qué ocurre si el hilo inicial (que ejecuta la función `main()`) no espera la finalización del resto de hilos?

4. El enunciado especifica que, cuando todos los coches hayan llegado a la meta, el hilo padre se encargará de mostrar en pantalla la clasificación final. Respecto a esta funcionalidad, responda a las siguientes cuestiones:

   4.1 ¿Es correcto pensar en obtener la clasificación mediante una solución en la que el padre, a medida que vaya finalizando cada hilo (coche) por el que espera (es decir, tras el código incluido en las líneas 60 a 65), vaya imprimiendo el identificador del coche que ha finalizado? Razone la respuesta.

   4.2 Resuelva el problema planteado en este apartado ejecutando los pasos que a continuación se describen:

   *a*) Declare las siguientes variables globales:

   ```
   volatile int clasificacionFinal[N_COCHES]
   volatile int finalCarrera=0
   ```

   *b*) Inserte las líneas de código que considere necesarias en el espacio indicado mediante el comentario `CODIGO 4` para que, al finalizar cada hilo coche la carrera, almacene su identificador (campo `id`) en la siguiente posición vacía de `clasificacionFinal[]`.

   *c*) ¿Se podrían producir errores al ejecutar las operaciones anteriores sin ningún tipo de control? ¿Y con la función `rand_r()`? Razone claramente la respuesta. En caso afirmativo, indique cómo solucionaría el problema mencionando alguna de las funciones incluidas en la Tabla 1 (se pide sólo indicar las funciones sin codificar en este apartado).

   *d*) Si la respuesta a la pregunta anterior ha sido afirmativa, codifique correctamente el programa incluyendo la solución al problema indicado.

   *e*) Inserte las líneas de código que considere necesarias en el espacio indicado mediante el comentario `CODIGO 5` para que el hilo padre, una vez que todos los hilos hayan llegado a la meta, muestre en pantalla la clasificación final de la carrera accediendo a cada una de las posiciones del array `clasificacionFinal[]`.

5. Realice un *makefile* que permita generar correctamente la aplicación y que incluya un objetivo ficticio `clean`, tal y como ya se ha explicado en el laboratorio. Se valorará además la estructuración del código de la aplicación e incluir un archivo de cabecera `simula_car.h` con los contenidos adecuados.

NOTA: Para usar funciones *re-entrantes* como `rand_r()`, necesita utilizar la opción `-D_REENTRANT` al compilar con `gcc`.