UNIVERSIDAD DE CÓRDOBA
ESCUELA POLITÉCNICA SUPERIOR
DEPARTAMENTO DE INFORMÁTICA Y ANÁLISIS NUMÉRICO

ASIGNATURA *SISTEMAS OPERATIVOS*

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

Hilos

Profesorado: Juan Carlos Fernández Caballero
Alberto Cano Rojas

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue familiarizar al alumnado con la creación y gestión de hilos en UNIX, también conocidos como procesos ligeros o hebras. En una primera parte se dará una introducción teórica sobre hilos, siendo en la segunda parte de la misma cuando, mediante programación en C, se practicarán los conceptos aprendidos, utilizando las rutinas de interfaz del sistema que proporcionan a los programadores el conjunto de librerías estándar de *glibc*, basándose en el estándar POSIX.

2 Recomendaciones

El lector debe completar las nociones dadas en las siguientes secciones con consultas bibliográficas, tanto en la Web como en la biblioteca de la Universidad, ya que unos de los objetivos de las prácticas es potenciar su capacidad autodidacta y su capacidad de análisis de un problema. Es recomendable que, aparte de los ejercicios prácticos que se proponen, pruebe y modifique otros que se encuentren en la Web (se dispone de una gran cantidad de problemas resueltos en C sobre esta temática), ya que al final de curso deberá acometer un examen práctico en ordenador como parte de la evaluación de la asignatura.

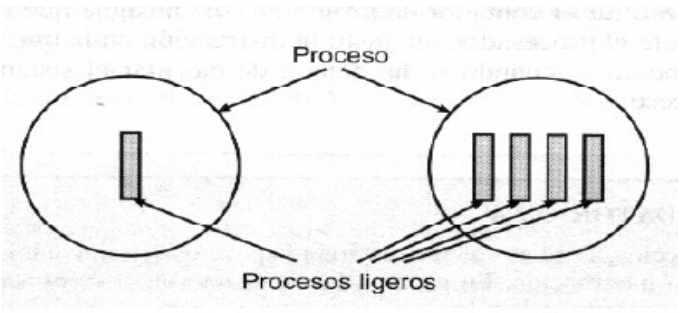
No olvide que debe consultar el estándar POSIX http://pubs.opengroup.org/onlinepubs/9699919799/ y la librería GNU C (*glibc*) en http://www.gnu.org/software/libc/libc.html.

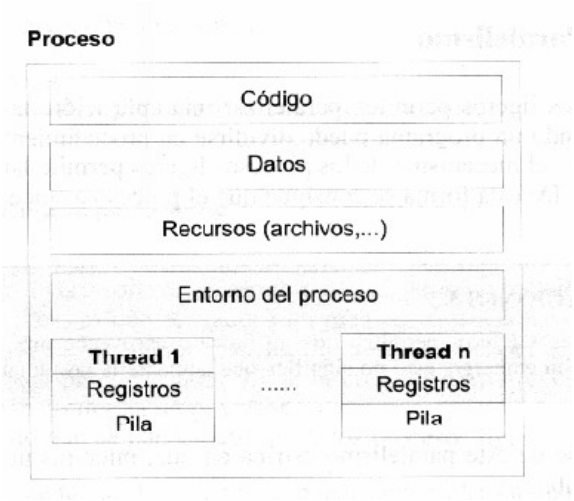
3 Conceptos teóricos

3.1 Diferencia entre procesos y hebras (threads, hilos o procesos ligeros)

Un **proceso** es un programa en ejecución que se ejecuta en secuencia, no más de una instrucción a la vez. Como se vio en la Práctica 1, se pueden crear procesos nuevos mediante la llamada a *fork()*. Estos nuevos procesos son idénticos al proceso padre que hizo la llamada (ya que son una copia del mismo), excepto en que se alojan en zonas o espacios de memoria distintos. Al ser copias tienen las mismas variables con los mismos nombres, pero distintas instancias, por lo que si un proceso hijo realiza una modificación en una variable, ésta no se ve afectada en el proceso padre u otros procesos. Por tanto, los procesos no comparten memoria ni se comunican entre si a no ser que utilicemos mecanismos de intercomunicación de procesos (IPC – *InterProcess Communication*) específicos como las colas de mensajes, la memoria compartida, pipelines, señales, o sockets.

Una **hebra**, **hilo** o **proceso ligero** (**thread** en inglés) es un flujo de control perteneciente a un proceso, que tiene su propia pila. A diferencia de un proceso, un *thread* comparte memoria con otros *threads*. Un proceso puede tener una o varias hebras, tal y como se refleja en la siguiente figura. Desde el punto de vista de la programación, una hebra se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario o proceso ligero primario se correspondería con el *main()*.



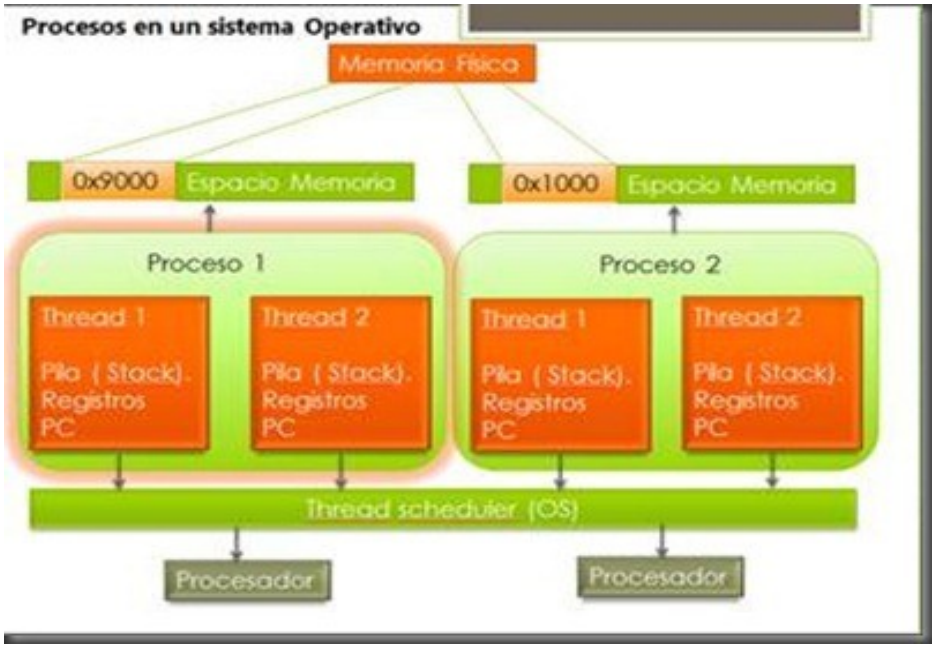


Cada hebra dentro de un proceso tiene determinada información propia que **no comparte** con el resto de hebras que nacen del mismo proceso:

- Tienen su propio estado (ejecutando, listo, bloqueado).
- Tienen su propia pila o stack.
- Tienen sus propios valores de los registros que usa del procesador (contexto).
- Propio contador de programa.
- *errno*.

Con respecto a la información que se **comparte** con otras hebras procedentes del mismo proceso son:

- **Mismo espacio de memoria.** El espacio de memoria incluye: Código, datos y pilas del conjunto de hebras (ver figuras anteriores). El espacio de memoria corresponde al proceso y a todas las hebras que engloba ese proceso, por lo que no hay una protección de memoria como ocurre con los procesos. Esto hace imprescindible el uso de semáforos o mutex (EXclusión MUTua) para evitar que dos *threads* accedan a la vez a la misma estructura de datos (se estudiará en las siguientes prácticas). También hace que si un hilo "se equivoca" y corrompe una zona de memoria, todos los demás hilos del mismo proceso vean la memoria corrompida. Un fallo en un hilo puede hacer fallar a todos los demás hilos del mismo proceso.
- **Variables globales.** Las hebras de un mismo proceso se pueden comunicar (entre una de las maneras posibles) mediante variables globales, pero hay que tener extremado cuidado con su programación.
- **Archivos abiertos.**
- **Temporizadores.**
- **Señales y Semáforos** (se estudiará más adelante).
- **Entorno de trabajo.**



La ventajas principales (debido a que comparten el mismo espacio de memoria) de usar un grupo de *threads* en vez de un grupo de procesos son:

- Crear o terminar una hebra es mucho más rápido que crear o terminar un proceso, ya que las hebras comparten determinados recursos del padre. Cuando se termina un proceso se debe eliminar el BCP del mismo, mientras que en un hilo se elimina solo su contexto y pila.
- El cambio de contexto entre *threads* es realizado mucho más rápidamente que el cambio de contexto entre procesos, mejorando el rendimiento del sistema. Al cambiar de un proceso a otro el sistema operativo (mediante el *dispatcher*) genera lo que se conoce como *overhead*, que es tiempo desperdiciado por el procesador para realizar un cambio de contexto, por ejemplo pasar del estado de ejecución del proceso actual al estado de espera o bloqueo y colocar un nuevo proceso en ejecución. En los hilos, como pertenecen a un mismo proceso, al realizar un cambio de hilo el tiempo perdido es casi despreciable.
- Si se necesitase comunicación entre hebras también sería mucho más rápido que intercomunicar procesos, ya que los datos están inmediatamente habilitados y disponibles entre hebras. En la mayoría de los sistemas, en la comunicación entre procesos, debe intervenir el núcleo para ofrecer protección de los recursos y realizar la comunicación misma. En cambio, entre hilos pueden comunicarse entre sí sin la invocación al núcleo. Por lo tanto, si hay una aplicación que debe implementarse como un conjunto de unidades de ejecución relacionadas, es más eficiente hacerlo con una colección de hilos que con una colección de procesos separados.

Así como podemos tener múltiples procesos ejecutando en un PC, también podemos tener múltiples *threads*. Como dentro de un proceso puede haber varios hilos de ejecución (varios threads), en el caso de que tuviéramos más de un procesador o un procesador con varios núcleos, un proceso podría estar haciendo varias cosas "a la vez", pero de manera más rápida que si lo hiciéramos con procesos puros. Así, cada hebra podría asignarse a un núcleo o a un procesador, y si un hilo se interrumpe o bloquea los demás pueden seguir ejecutando. Estos son los procesos a nivel de núcleo o KLT que estudiará en clases teóricas.

Resumiendo, en el caso de UNIX, mediante *fork()* creamos procesos independientes entre si, de forma que sea imposible que un proceso se entremezcle por equivocación en la zona de memoria de otro proceso, haciendo que el sistema sea fiable. Por otro lado, las hebras pueden ser útiles para programar aplicaciones que deben hacer tareas simultáneamente y/o queremos que haya bastante intercomunicación entre ellas. Dependiendo de la aplicación optaremos por una solución u otra, aunque eso es también un aspecto que debe elegir el analista-programador en base a su experiencia.

Algunos ejemplos de usos de hebras, cuyo uso permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente, pueden ser:

- Procesador de textos: utilizar a una hebra por cada tarea que realiza.
  - ✔ Interacción con el usuario.
  - ✔ Corrector ortográfico/gramatical.
  - ✔ Guardar automáticamente y/o en segundo plano.
  - ✔ Mostrar resultado final en pantalla.
- Hoja de cálculo:
  - ✔ Interacción con el usuario.
  - ✔ Actualización en segundo plano.
- Servidor web: dos tipos de hebras.
  - ✔ Interacción con los clientes (navegadores).
  - ✔ Gestión del caché de páginas.
- Navegador: varios tipos de hebras.
  - ✔ Interacción con el usuario.
  - ✔ Interacción con los servidores (web, ftp, ...).
  - ✔ Dibujo de la página (1 hebra por pestaña).

3.2 Biblioteca de C para el uso de hebras y normas de compilación

Al igual que con los procesos en sistemas UNIX, las hebras también tienen una especificación en el estándar POSIX IEEE Std 1003.1-2008, concretamente en la **biblioteca *pthread***. Para crear programas que hagan uso de la biblioteca *pthread* necesitamos, en primer lugar, la biblioteca en sí. Ésta viene en la mayoría de distribuciones Linux, y seguramente se instale al mismo tiempo que los paquetes incluidos para el desarrollo de aplicaciones. Si no es así, o usa un sistema que no sea Linux pero se base en POSIX, la biblioteca no debería ser difícil de encontrar en la red, porque es bastante conocida y usada. Una vez tenemos la biblioteca instalada, y hemos creado nuestro programa, deberemos compilarlo y "linkarlo" con la misma. El fichero de cabecera <*pthread.h*>*w* debe estar incluido en nuestras implementaciones. La forma más usual de compilar si estamos usando *gcc* es:

```
gcc programa_con_pthreads.c -o programa_con_pthreads -lpthread
```

3.3 Servicios POSIX para la gestión de hebras

A continuación se expondrán las funciones o llamadas al sistema para la gestión de hebras que implementa la librería *pthread* al seguir el estándar POSIX especificado en la IEEE (Institute of Electrical and Electronics Engineers, Instituto de Ingenieros Eléctricos y Electrónicos).

3.3.1 Creación y ejecución de una hebra (pthread_create())

Para crear un *thread* nos valdremos de la función *pthread_create()* de la biblioteca *pthread*, y de la estructura *pthread_t,* la cual identifica cada *thread* diferenciándola de las demás y conteniendo todos sus datos.

El prototipo de la función es el siguiente[1]:

```
#include <pthread.h>

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

- *thread*: Es una variable del tipo *pthread_t* que contendrá los datos del *thread* y que nos servirá para identificar un *thread* en concreto (ID).
- *attr*: Es un parámetro del tipo *pthread_attr_t* y que se debe inicializar previamente con los atributos que queramos que tenga el *thread*. Si pasamos como parámetro NULL la biblioteca le asignará al *thread* los atributos por defecto. La biblioteca *pthread* admite implementar hilos a nivel de usuario y a nivel de núcleo. Por defecto se hace a nivel de núcleo, de tal manera que cada hebra se pueda tratar como un proceso independiente. Si queremos manejarlas a nivel de usuario y establecerles prioridad y algoritmo de planificación habría que indicarlo en el momento de crearla, cambiando alguno de los atributos por defecto. También se pueden indicar cosas como la cantidad de reserva de memoria utilizada en la pila de la hebra, si una hebra debe o no esperar a la finalización de otra y algunas más que puede consultar en la Web. Los atributos de una hebra no se pueden modificar durante su ejecución.
- *start_routine*: Aquí pondremos la dirección de memoria de la función que queremos que ejecute el *thread*. La función debe devolver un puntero genérico (*void **) como resultado, y debe tener como único parámetro otro puntero genérico. La ventaja de que estos dos punteros sean genéricos es que podremos devolver cualquier cosa que se nos ocurra mediante los *castings* de tipos necesarios. Si necesitamos pasar o devolver más de un parámetro a la vez, se puede crear una estructura y meter allí dentro todo lo que necesitemos. Luego pasaremos o devolveremos la dirección de esta estructura como único parámetro.
- *arg*: Es un puntero al parámetro que se le pasará a la función *start_routine* (parámetro *start_routine*). Puede ser NULL si no queremos pasar nada a la función. **Cualquier argumento pasado a la hebra se debe pasar por referencia y hacerle un *casting* a *void ***.

En caso de que todo haya ido bien, la función devuelve un 0, o un valor distinto de 0 en caso de que hubiera algún error.

Una vez hemos llamado a esta función, ya tenemos a nuestro(s) *thread*(s) funcionando, pero ahora tenemos dos opciones: esperar a que terminen los *threads*, en el caso de que nos interese recoger algún resultado, o simplemente decirle a la biblioteca *pthread* que cuando termine la ejecución de la función del *thread* elimine todos los datos de sus tablas internas. Para ello, disponemos de dos funciones: *pthread_join()* y *pthread_detach()*.

1 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_create.html

3.3.2 Espera a la finalización de una hebra (pthread_join())

A veces es necesario hacer que un hilo espere a otro/s hilo/s. Por ejemplo, supongamos que varios hilos están realizando un cálculo y es necesario el resultado de todos ellos para obtener el resultado total. El hilo encargado de este resultado total debe esperar a que todos los demás hilos terminen.

Si nos interesa esperar a que finalice una hebra o hilo y recoger resultados utilizamos la función *pthread_join()*, cuyo prototipo es el siguiente[2]:

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return)
```

Esta función suspende el *thread* llamante (el que invoca a esta función) hasta que termine la ejecución del *thread* indicado por *th*. Además, una vez éste último termina, pone en *thread_return* el resultado devuelto y que estamos esperando.

- *th:* Es el identificador del thread que queremos esperar, y es el mismo que usamos al invocar a *pthread_create()*.
- *thread_return*: Es un puntero a puntero que apunta (valga la redundancia) al resultado devuelto por el *thread* que estamos esperando cuando terminó su ejecución. Ese *thread* al que esperamos devolverá un valor usando *return()* o *pthread_exit()*. Si el parámetro parámetro *thread_return* es NULL, le estamos indicando a la biblioteca que no nos importa el resultado de la hebra a la que estamos esperando.

Esta función devuelve 0 en caso de que todo esté correcto, o valor diferente de 0 si hubo algún error.

Si cuando se invoca a *pthread_join()* la hebra a la que esperamos ya ha terminado, se devuelve el control a la hebra llamante en ese mismo momento. Es importante que tenga en cuenta esta distinción, puede utilizar la función *pthread_join()* en cualquier momento, ya que aunque una hebra haya terminado, el estado y los resultados de la misma se guardan hasta que en un determinada zona del código se llame a *pthread_join()*. Es decir, no tiene porque hacerse la llamada justo después de que la hebra se cree y comience a ejecutar mediante la invocación de *pthread_create()*.

Si se hace una creación de hilos desde el *main()* y estos hilos creados, a su vez, no crean ningún otro hilo al cual tengan que esperar, es necesario poner en dicho *main()* un *pthread_join()* para que no se termine nuestro programa y se desapile de memoria antes de que terminen los hilos creados. Por defecto un *thread* es *joinable, es decir,* requiere que el proceso creador utilice *pthread_join()*, si no queda en estado zombie.

Copie el siguiente ejemplo, compílelo, ejecútelo y observe sus resultados. Después elimine los *pthread_join()* del ejercicio anterior y observe sus resultados.

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

2 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_join.html

```
//Declaración de una estructura
struct param
{
  char * frase;
  int numero;
};

/* Función que se asignará a los hilos que se creen. Recibe un puntero a estructura */
void hiloMensaje (struct param * mensa)
{
   //Casting siempre necesario, ya que realmente lo que recibe es un puntero a void
   struct param * aux = (struct param *) mensa;
   printf("%s %d\n", mensa->frase, mensa->numero);
}

int main()
{
  pthread_t thd1, thd2; //Declaración de dos hebras, hilos o procesos ligeros. NO CREACION
  //Inicializacion de 2 estructuras de tipo “struct param”
  struct param param1 = {"Soy el hilo: ", 1};
  struct param param2 = {"Digo otra cosa ", 2};

/* Creamos dos hilos. La función la pasaremos como (void *) nombreFuncion. Es decir, hacemos un casting a (void *), aunque por defecto no es necesario, ya que el nombre de una función es su dirección de memoria. También es importante realizar esto con la dirección de memoria de la variable que contiene los parámetros que se le pasan a la función */
  pthread_create (&thd1, NULL, (void *) hiloMensaje, (void *) &param1);
  pthread_create (&thd2, NULL, (void *) hiloMensaje, (void *) &param2);

/* Esperamos la finalización de los hilos. Si la función devolviera algo habría que recogerlo con el segundo argumento, que en este caso esta a NULL. Cuando el segundo argumento no es NULL, se recogen los resultados que vienen de pthread_exit(), que se explicará a continuación.*/
/*Si no se ponen estos join() en el programa principal y simplemente lanzamos los dos hilos y finalizamos, lo más probable es que los hilos no lleguen a ejecutarse completamente o incluso que no lleguen ni a terminar de arrancar antes de que el programa principal termine.*/
  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);

  printf("Han finalizado los thread.\n");
}
```

Aquí dispone de otro ejemplo de hebras del que debe sacar conclusiones importantes, cópielo, ejecútelo y observe sus resultados.

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

/* this function is run by the second thread */
void *inc_x(void *x_void_ptr)
{
/* increment x to 100 */
int *x_ptr = (int *)x_void_ptr;
while(++(*x_ptr) < 100);
printf("x increment finished\n");
/* the function must return something - NULL will do */  /*Or pthread_exit(NULL)*/
```

```
return NULL;
}

int main()
{
int x = 0, y = 0;
/* this variable is our reference to the second thread */
pthread_t inc_x_thread;
/* show the initial values of x and y */
printf("x: %d, y: %d\n", x, y);
/* create a second thread which executes inc_x(&x) */
if(pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
        fprintf(stderr, "Error creating thread\n");
        return 1;
}
/* increment “y” to 100 in the first thread */
while(++y < 100);
printf("y increment finished\n");
/* wait for the second thread to finish */
if(pthread_join(inc_x_thread, NULL)) {
        fprintf(stderr, "Error joining thread\n");
        return 2;
}
/* show the results. “x” is now 100 thanks to the second thread */
printf("x: %d, y: %d\n", x, y);
return 0;
}
```

3.3.3 Finalizar una hebra y devolver resultados (pthread_exit())

Los recursos asignados por el sistema operativo a un hilo son liberados cuando el hilo termina. La terminación del hilo se produce cuando la función asignada al hilo que está ejecutando termina, cuando ejecuta un *return()* o cuando se llama a *pthread_exit()*. Si la función asignada a la hebra no ejecuta ni *return()* ni *pthread_exit(),* se ejecuta automáticamente y de manera transparente para el programador un *pthread_exit(NULL)*.

El prototipo de la función es el siguiente[3].

```
#include <pthread.h>
void pthread_exit (void *retval)
```

- *retval*: Es un puntero genérico a los datos que queremos devolver como resultado. Estos datos serán recogidos cuando alguien haga un *pthread_join()* con el identificador de *thread*. El parámetro es el valor que se devolverá al hilo que espera. Como es un *void* *, puede ser un puntero a cualquier cosa. Según están implementadas las hebras, el valor devuelto no debe estar localizado en la pila de la hebra. Si necesitamos que cada hebra tenga su propio grupo de datos sobre los que operar y devolver, necesitaremos utilizar la función *malloc()* de C.

Note que si en una hebra usásemos *exit()*, se terminaría el proceso completo (con todas sus hebras).

3 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_exit.html

Copie el siguiente ejemplo, compílelo, ejecútelo y observe sus resultados.

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

//Prototipo
void *mifuncion (void *arg);

main ()
{
  pthread_t tid;

  //Vector de enteros que vamos a pasar como parámetro a una hebra haciendo casting a void *
  int misargs[2];
  misargs[0] = -5;
  misargs[1] = -6;

  printf("Se va a crear un hilo...\n");
  pthread_create(&tid, NULL, mifuncion, (void *) misargs);

  printf("Hilo creado. Esperando su finalizacion con pthread_join()...\n");
  pthread_join(tid, NULL);

  printf("Hilo finalizado...\n");
}

void *mifuncion(void *arg)
{
  int *argu;

  printf("Hilo hijo ejecutando...\n");
  argu = (int *) arg; //Casting a entero del parámetro de entrada.
  printf("Hilo hijo: arg1= %d arg2= %d\n", argu[0], argu[1]);

  printf("Hilo hijo finalizando....\n");
  /* Esta función no devuelve nada, por tanto no se podrá recoger nada con un join(). Por defecto,
  si no se incluye se hace implícitamente un pthread_exit(NULL); */
  pthread_exit(NULL);
}
```

Modifique el ejemplo anterior, cambiando el *pthread_exit(NULL)* por un *exit(0)* y observe sus resultados.

3.3.4 Desconectar una hebra creada al terminar su ejecución (pthread_detach())

Por defecto, el resultado y estado de ejecución de todos los *threads* se guardan hasta que hacemos un *pthread_join()* para recogerlos, e incluso después, de modo que cualquier otra hebra puede obtener el resultado de otra. Cuando no nos interese el resultado de una hebra y queremos que automáticamente el sistema limpie sus datos y tablas internas tenemos que indicarlo con la función *pthread_detach()*. Así una vez que el *thread* haya terminado, se eliminarán sus datos y tablas internas y se liberará espacio y recursos. Recuerde, esto no lo hace *pthread_join()*. **Una vez que se ha invocado a *pthread_detach()* y se ha hecho un *pthread_join()* ya no se puede hacer otro *pthread_join()* de la hebra indicada. Por otro lado, cuando usamos *pthread_detach()* no es necesario que usemos *pthread_join()* para esperar o hacer posible que la hebra acabe. Si el proceso principal termina su ejecución antes que la hebra, ésta continua sin problema hasta que termina, siempre que finalicemos el proceso principal con *pthread_exit(NULL);***

El prototipo de *pthread_detach()* es el siguiente[4]:

```
#include <pthread.h>
int pthread_detach (pthread_t th)
```

- th: Es el identificador del thread.

Devuelve 0 en caso de que todo haya ido bien o diferente de 0 si hubo error (EINVAL - El valor especificado para el argumento no es correcto, ESRCH - No se pudo encontrar elemento que coincide con el valor especificado). Busque en la Web estas macros.

A continuación tiene un ejemplo del uso de *pthread_detach()*. Copie el código, compílelo, ejecútelo, complete el tratamiento de errores consultando la Web y Moodle y observe sus resultados.

```
void * message_print (void * ptr)
{
  int error = 0;
  char *msg;

  /* Desconexión del hilo cuando finalice. pthread_self() devuelve el ID de la hebra que invoca
  esta función. Se estudiará a continuación. */
  error = pthread_detach(pthread_self());
 /* Manejar el error */
 //...

  msg = (char *) ptr;
  sleep(5);
  printf("THREAD: This is the Message %s\n", msg);
  pthread_exit(NULL);
}

int main(void)
{
  int error = 0;
  size_t i = 0;
  char mess[] = "This is a test";

  /* Creación de un conjunto de hebras */
  pthread_t thr [5]; //Array de hebras
  for(i = 0; i < thread_no; i++)
  {
    error = pthread_create( &(thr[i]), NULL, message_print, (void  *) mess);
    /* Manejar el error */
    //...
  }
  printf("MAIN: Thread Message: %s\n", mess);
  pthread_exit(NULL);
}
```

4 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_detach.html

3.3.5 Obtener la información de una hebra (pthread_self())

Para obtener la información de un hebra (entre otras su ID) utilizaremos la función pthread_self()[5]:

```
#include <pthread.h>
pthread_t pthread_self(void)
```

Esta función devuelve al *thread* que la llama su identificación, en forma de variable del tipo *pthread_t*. Se puede hacer un *casting (unsigned int) pthread_t* para imprimir el ID, busque algún ejemplo en la Web.

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

void *thread(void *vargp)
{
  /*En cuanto se haga el pthread_detach() se eliminará el estado de esta hebra y no se podrán hacer  más  joins con ella.*/
  sleep(3);
  pthread_detach(pthread_self());
  pthread_exit((void*)42);
}

int main()
{
  int i = 0;
  pthread_t tid;

  pthread_create(&tid, NULL, thread, NULL);
  //Recogemos el resultado que devuelve la hebra. Ninguna otra hebra o proceso principal podra hacerle otro join().
  pthread_join(tid, (void**)&i);
  printf("%d\n",i);
}
```

3.3.6 Matar una hebra desde el proceso llamador (pthread_kill())

Para "matar" a una hebra desde el proceso que la crea podemos utilizar la llamada *pthread_kill()*[6].

```
#include <pthread.h>
int pthread_kill(pthread_t thread, int signo)
```

- thread: identifica el thread al cual le queremos enviar la señal.
- signo: número de la señal que queremos enviar al thread. Podemos usar las constantes definidas en <signal.h>[7]. Para matar la hebra se utiliza la macro SIGKILL.

Devuelve 0 si no hubo error, o diferente de 0 si lo hubo. Busque información en la Web para ver

5 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_self.html
6 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_kill.html
7 http://es.wikipedia.org/wiki/Se%C3%B1al_%28inform%C3%A1tica%29

algún ejemplo de *pthread_kill()*. Aunque pueda parecer útil a primera vista, la única utilidad que tiene esta función es matar un *thread* desde el proceso que la crea. Si se quiere usar con fines de sincronización hay formas mejores de hacerlo tratándose de *threads*: mediante semáforos, paso de mensajes y variables de condición.

3.3.7 Atributos de un thread

Cada hilo o hebra posee una serie de atributos o propiedades asociados. Un objeto o entidad atributo puede ser asignado o asociado a varios hilos, de manera que si cambia la entidad atributo cambian los hilos asociados a la misma. Los objetos atributo son del tipo *pthread_attr_t.* La siguiente tabla muestra alguna de las funciones para establecer los atributos que se pueden asociar a un hilo.

Propiedad	Función
Inicialización	pthread_attr_init
	pthread_attr_destroy
Tamaño de pila	pthread_attr_setstacksize
	pthread_attr_getstacksize
Dirección de pila	pthread_attr_setstackaddr
	pthread_attr_getstackaddr
Estado de desconexión	pthread_attr_setdetachstate
	pthread_attr_getdetachstate
Alcance	pthread_attr_setscope
	pthread_attr_getscope
Herencia	pthread_setinheritsched
	pthread_getinheritsched
Política de programación	pthread_attr_setschedpolicy
	pthread_attr_getschedpolicy
Parámetros de programación	pthread_attr_setschedparam
	pthread_attr_getschedparam

El prototipo de las funciones para inicializar un objeto atributo y destruirlo son[8]:

```
#include <pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

- *pthread_attr_init:* Inicializa el objeto de atributos de un hilo *attr* y establece los valores por defecto. Posteriormente, este objeto, con los atributos por defecto establecidos, se puede utilizar para crear múltiples hilos.
- *pthread_attr_destroy:* Destruye el objeto de atributos de un hilo *attr* y éste no puede volver a utilizarse hasta que no se vuelva a inicializar.

Los atributos (más relevantes) de un hilo POSIX son (busque y amplíe en la Web):

- *detachstate:* controla si otro hilo hilo podrá esperar por la terminación de este hilo (mediante la invocación a *pthread_join()*).
    - PTHREAD_CREATE_JOINABLE (valor por defecto).
    - PTHREAD_CREATE_DETACHED (desconectado).

8 http://pubs.opengroup.org/onlinepubs/9699919799/

- *schedpolicy*: controla cómo se planificará el hilo.
    - SCHED_OTHER (valor por defecto, planificación normal).
    - SCHED_RR (Round Robin+ tiempo real + privilegios root).
    - SCHER_FIFO (First In First Out+ tiempo real + privilegios root).
- *scope*: controla a qué nivel es reconocido el hilo, si compite por recursos dentro del proceso o bien a nivel del sistema.
    - PTHREAD_SCOPE_SYSTEM (valor por defecto, el hilo es reconocido por el núcleo).
    - PTHREAD_SCOPE_PROCESS (no soportado en la implementación LinuxThreads de hilos POSIX).

Para establecer o consultar los atributos anteriores asociados a un hilo podemos utilizar las siguientes funciones:

```
#include <pthread.h>

//Para establecer y consultar el estado de terminación detachstate de un hilo
int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);

//Para establecer y consultar el estado de terminación schedpolicy de un hilo
int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy (const pthread_attr_t *attr, int *policy);

//Para establecer y consultar el estado de terminación scope de un hilo
int pthread_attr_setscope (pthread_attr_t *attr, int contentionscope);
int pthread_attr_getscope (const pthread_attr_t *attr, int *contentionscope);
...
```

Para ajustar los atributos de un thread hay proceder de la siguiente manera:

1. Crear un objeto de tipo *pthread_attr_t.*
2. Utilizar la llamada *pthrad_attr_init* para iniciar el objeto creado en el punto 1.
3. Modificar los atributos a sus necesidades.
4. Pasar un puntero al objeto cuando se invoca la llamada *pthread_create.*
5. Utilizar la llamada *pthread_attr_destroy* para liberar la variable y que pueda ser reutilizada.

Un thread puede ser creado como PTHREAD_CREATE_JOINABLE o PTHREAD_CREATE_DETACHED. Por defecto, si no se especifica, un thread será *joinable*. Un thread *joinable* requiere que el proceso padre utilice *pthread_join()*, si no queda en estado zombie. Un *thread detach* es liberado automáticamente por el sistema después de que la hebra finalice.

El siguiente fragmento de programa muestra como crear un thread de tipo *detach*. Cópielo, ejecútelo y piense en los resultados que está observando por pantalla:

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

void* start_function(void* value)
{
   printf("%s is now entering the thread function.\n", (char*)value);
   printf("%s is now leaving the thread function.\n", (char*)value);
   pthread_exit((void*)99);
}

main()
{
   int res,err;
   pthread_attr_t attr;
   pthread_t thread1;
   int i = 0;

   res = pthread_attr_init(&attr);
   if (res != 0) {
      perror("Attribute init failed");
      exit(EXIT_FAILURE);
   }
   res = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   if (res != 0) {
      perror("Setting detached state failed");
      exit(EXIT_FAILURE);
   }

   res = pthread_create(&thread1, &attr, start_function, (void*)"Thread1");
   if (res != 0) {
      perror("Creation of thread failed");
      exit(EXIT_FAILURE);
   }

   sleep(4);
   pthread_attr_destroy(&attr);

   pthread_join(thread1, (void**)&i);
   printf("Valor devuelto:%d\n",i);
}
```

Aquí tiene otro ejemplo similar al anterior. Pruébelo, después comente las lineas que están en rojo y vuelva a probarlo.

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

int thread_finished = 0;

void* start_function(void* value)
{
   printf("%s is now entering the thread function.\n", (char*)value);
```

```
    sleep(4);
    thread_finished =1;
    printf("%s is now leaving the thread function.\n", (char*)value);
    pthread_exit(value);
}

main()
{
    int res,err;
    pthread_attr_t attr;
    pthread_t thread1;
    int i=0;

    res = pthread_attr_init(&attr);
    if (res != 0) {
        perror("Attribute init failed");
        exit(EXIT_FAILURE);
    }
    res = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    if (res != 0) {
        perror("Setting detached state failed");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&thread1, &attr, start_function, (void*)"Thread1");
    if (res != 0) {
        perror("Creation of thread failed");
        exit(EXIT_FAILURE);
    }

    while(!thread_finished) {
        printf("Waiting for thread1 to finish.\n");
        sleep(1);
    }

    pthread_join(thread1, (void**)&i);
    printf("Valor devuelto:%d\n",i);

    printf("Child thread finished.\n");
    pthread_attr_destroy(&attr);
}
```

4 Ejercicios prácticos

4.1 Ejercicio1

Implemente un programa que cree dos hebras. Cada hebra ejecutará una función a la que se le pasará como parámetro una cadena, concretamente a la primera hebra se le pasará la cadena “hola” y a la segunda “mundo”. La función que deben ejecutar ambas debe imprimir carácter a carácter la cadena recibida, haciendo un *sleep(1)* entre cada impresión de carácter. Observe los resultados obtenidos.

Repita lo mismo pero recogiendo las dos cadenas por la linea de argumentos.

4.2 Ejercicio2

Implemente un programa que cree un número *N* de hebras. Cada hebra creará 2 números aleatorios (consulte la web para la generación de aleatorios) y guardará su suma en una variable para ello, que será devuelta a la hebra llamadora (la que invocó *pthread_create()*). La hebra principal ira recogiendo los valores devueltos por las *N* hebras y los ira sumando en una variable no global cuyo resultado mostrará al final por pantalla. Para ver que los resultados finales son los que usted espera, muestre los números que va creando cada hebra y su suma, de forma que pueda comparar esas sumas parciales con la suma final de todos los números creados por todas las hebras. Utilice macros definidas y comprobación de errores en sus programas (*errno* y comprobación de valores devueltos en cada llamada, con sus posibles alternativas), será valorado en el examen final de la asignatura.

4.3 Ejercicio3

Implementar un programa para realizar la suma en forma paralela de los valores de un vector de 10 números enteros que van de 0 a 9 (puede probar con aleatorios). Utilice una cantidad de hilos indicada como parámetro de entrada por la linea de argumentos y reparta la suma del vector entre ellos (como considere oportuno). La suma debe ser el subtotal devuelto por cada hilo. Haga comprobación de errores en su programa.

4.4 Ejercicio4

Obtenga los dos ficheros .mp4 que se facilitan en la plataforma Moodle u obtenga usted dos vídeos cualesquiera que no sean demasiado largos de la Web. Cree un programa que de forma paralela convierta dichos ficheros a .mp3 (extracción de audio) con el programa "ffmpeg" (disponible en Linux). Pida el nombre de los dos ficheros por la linea de argumentos. Use la llamada al sistema *system()* para invocar a "ffmpeg".

Para extraer el audio en formato mp3 de un fichero de video mp4, puedo utilizar la siguiente linea de comandos (si tienen curiosidad por el significado de los argumentos busque en la Web):

ffmpeg -i ficheroOriginal.mp4 -f mp3 -ab 192000 -ar 48000 -vn ficheroNuevoMP3.mp3

Pruebe a realizar el ejercicio usando alguna de las funciones *exec()* para procesos que se explicaron en la Práctica 1, en vez de usar la función *system()*.

4.5 Ejercicio5

Implemente un programa que cuente las líneas de los ficheros de texto que se le pasen como parámetros y al final muestre también el número de líneas totales (contando las de todos los ficheros juntos). Ejemplo de llamada: ./a.out fichero1 fichero2 fichero3

Debe crear un hilo por fichero obtenido por linea de argumentos, de forma que todos los ficheros se cuenten de manera paralela.

4.6 Ejercicio6

Realice la multiplicación de una matriz por un escalar usando para ello una hebra. Tanto la matriz como el escalar formarán parte de una estructura. Cree un programa cuyo proceso general cree dos hebras que se encargan de manera paralela de multiplicar dos matrices por sus correspondientes escalares, es decir, una hebra multiplicará una matriz por un escalar concreto y la otra hará lo mismo pero con otra matriz y otro escalar concreto. Por tanto, deberá crear funciones que sirvan para las dos hebras. Finalmente, el proceso general, una vez que hayan concluido su trabajo las

hebras, deberá mostrar las matrices por pantalla para comprobar el resultado de las operaciones. Aquí tiene un ejemplo del tipo de estructura a manejar:

```
struct parametros {
float escalar ;
float matriz [3][3];
};
```

4.7 Ejercicio7

Cree una estructura que contenga dentro un mensaje y un entero. Cree N hebras a las que se le pase a cada una una estructura (puede crearse un array de estructuras), de forma que la hebra incremente en 1 el entero y sustituya el primer carácter del mensaje por el número 9. Desde el proceso principal imprima el array de estructuras para comprobar los cambios.

4.8 Ejercicio8

Implemente un programa que cree dos hebras y cada una incremente 50 veces en un bucle una variable global (recuerde que la variable global, al estar en el mismo espacio de memoria para las dos hebras, es compartida, y que su uso es "peligroso"). Imprima al final del programa principal el valor de la variable (en cada ejecución posiblemente obtenga un valor diferente a 100 – problema de concurrencia –). Intente razonar el resultado, el cual le servirá como concepto introductorio de la siguiente práctica de la asignatura.