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

ASIGNATURA ***SISTEMAS OPERATIVOS***

2º DE GRADO EN INGENIERÍA INFORMÁTICA

PRÁCTICA 2

Hilos

Juan Carlos Fernández Caballero
jfcaballero@uco.es

Índice de contenido

1 Objetivo de la práctica

La presente práctica persigue instruir al alumnado con la creación y gestión de hilos en sistemas que siguen el estándar POSIX[1]. Los hilos se conocen también 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 con *glibc*, se practicarán los conceptos aprendidos, utilizando las rutinas de interfaz del sistema que proporciona a los programadores la biblioteca *pthread*, basada 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 uno de los objetivos de las prácticas es potenciar la capacidad autodidacta y de análisis de un problema.

Es recomendable también que, a parte de los ejercicios prácticos que se proponen, pruebe y modifique otros que encuentre 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.

Al igual que se le instruyó en las asignaturas de Metodología de la Programación, es recomendable que siga unas normas y estilo de programación claro y consistente. No olvide tampoco comentar los aspectos más importantes de sus programas, así como añadir información de cabecera a sus funciones (nombre, parámetros de entrada, parámetros de salida, objetivo, etc).

No olvide que debe consultar siempre que lo necesite el estándar *POSIX* en:

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

y la librería GNU C (*glibc*) en:

http://www.gnu.org/software/libc/libc.html

3 Conceptos teóricos

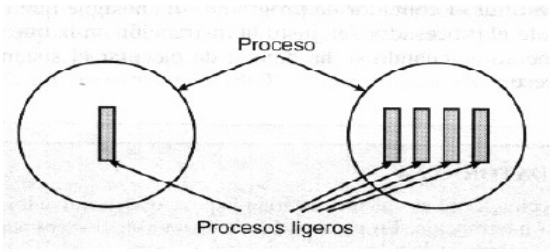
3.1 Diferencia entre procesos y hebras

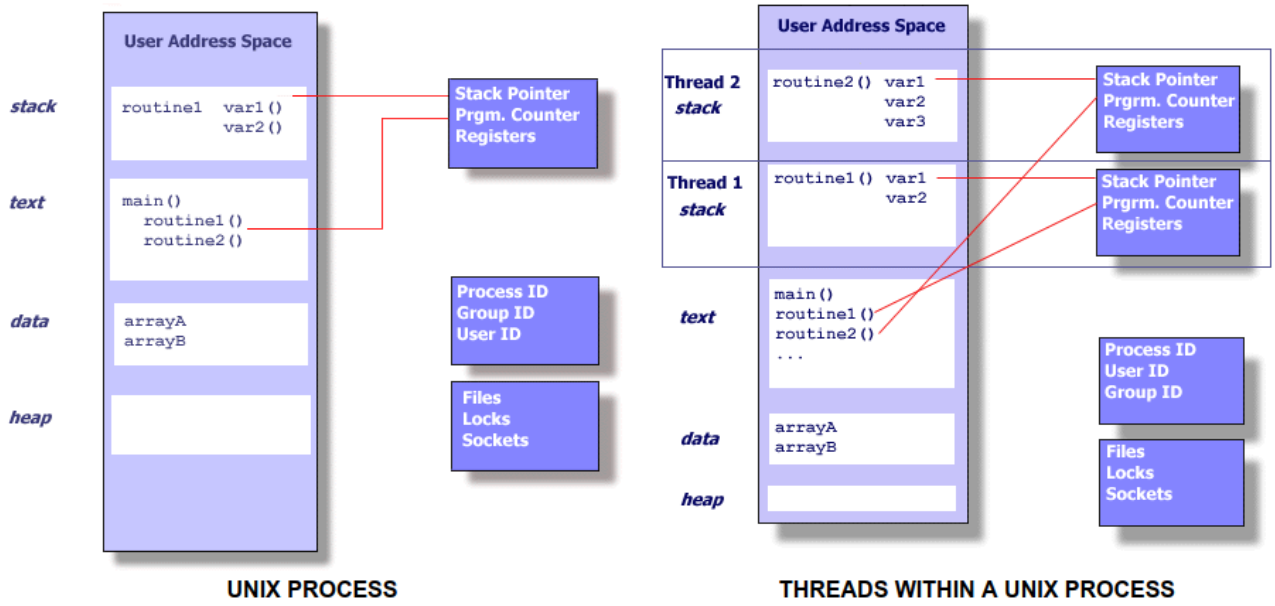
Un **proceso** es un programa en ejecución que se ejecuta en secuencia, no más de una instrucción a la vez. Como se vio en la Práctica 1, se pueden crear procesos nuevos mediante la llamada a *fork()*. Estos nuevos procesos son idénticos al proceso padre que hizo la llamada (ya que son una copia del 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

1 http://es.wikipedia.org/w/index.php?title=POSIX&oldid=53746603

u otros procesos (con algunas excepciones, como por ejemplo los descriptores de fichero). 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*),** como pueden ser el paso de mensajes con tuberías y colas, la memoria compartida, señales, o *sockets*.

Una **hebra**, **hilo** o **proceso ligero** (***thread*** en inglés) es un flujo de control perteneciente a un proceso. A diferencia de un proceso, **un *thread* puede compartir memoria con otros *threads* que pertenezcan al mismo proceso**. 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()*. Un proceso puede tener por tanto, una o varias hebras, tal y como se refleja en las siguientes figuras.





Cada hebra o hilo dentro de un proceso tiene determinada i**nformación propia que NO comparte** con el resto de hebras que nacen del mismo proceso, como por ejemplo:

- Un estado de ejecución por hilo (Ejecutando, Listo, Bloqueado).
- Un contexto o BCP propio del hilo, al igual que los procesos.
- Un contador de programa independiente, PC.
- Una pila de ejecución (*stack*).
- Un espacio de almacenamiento para variables locales.

Con respecto a la i**nformación que SI comparte** con otras hebras procedentes del mismo proceso, es la siguiente:

- **Mismo espacio de memoria reservado para el proceso (montículo)**. El espacio de memoria del montículo corresponde al proceso y a todas las hebras que engloba ese proceso. Esto hace imprescindible el uso de mecanismos como los semáforos o *mutex* (EXclusión MUTua), que evitan que dos *threads* accedan a la vez a la misma estructura de datos (se estudiarán en las siguientes prácticas). Así mismo, un fallo de terminación 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 mediante variables globales, pero hay que tener extremado cuidado con su programación.
- **Archivos abiertos (descriptores).** Las hebras comparten los descriptores de ficheros.
- **Señales**.
- **Variables de entorno del proceso.**

Las **ventajas** principales 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 y estructuras del padre que no hay que volver a crear o eliminar.
- 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 usado por el procesador para realizar un cambio de contexto. Un cambio de contexto es 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, el tiempo en ese cambio de contexto es menor.
- 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. 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 hilos pueden seguir ejecutando**. Estos son los llamados **hilos a nivel de núcleo**.

Resumiendo, en el caso de sistemas basados en POSIX, 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. 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 uso 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, por ejemplo:
  - Interacción con el usuario.
  - Corrector ortográfico/gramatical.
  - Guardar automáticamente y/o en segundo plano.
  - Mostrar resultado texto en pantalla.
- **Servidor web**:
  - Interacción con multiples clientes que piden un recurso.
- **Navegador.** Varias hebras simultaneas, como por ejemplo:
  - Una por cada pestaña e interacción con el usuario local.
  - Otras hebras para interacción con los servidores (web, ftp, ...).

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

Al igual que con los procesos, las hebras también tienen una especificación en el estándar *POSIX*, 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 que 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 con hebras, deberemos compilarlo y enlazarlo con la misma. El fichero de cabecera <*pthread.h*> debe estar incluido en nuestras implementaciones si usamos hebras. La forma más usual de compilar si estamos usando *gcc* es la siguiente (adición de *-lpthread*):

```
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.*

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

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

El prototipo de la función es el siguiente[2]. Consulte en la Web los posibles valores que puede devolver esta llamada para el tratamiento de errores (EAGAIN, EPERM):

#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 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, establecerles prioridad y un algoritmo de planificación concreto, 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*. 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 **).**

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

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.

Tenga cuidado al reutilizar variables que se pasan por referencia a los hilos cuando estos se crean, podría suceder que el hilo creado no se ejecutase a tiempo a fin de utilizar los valores antes de que se sobreescriban. Tenga en cuenta que está utilizando punteros y puede que al hilo no le de tiempo a copiar el parámetro en su memoria local. **Consulte el ejemplo "hello_arg_bad_parameter.c" disponible en Moodle**, donde se crean 4 hebras en paralelo a partir del proceso o hilo principal y a las cuales se les pasa como parámetro la dirección de un número entero, cuyo contenido respecto al que tenía al pasarse a cada hebra es probable que cambie antes de que cada hebra lo reciba y lo utilice**.**

Una vez hemos llamado a la función *pthread_create()* ya tenemos a nuestro(s) thread(s) funcionando, pero ahora tenemos **dos opciones** (se estudiarán en las siguientes secciones):

1. Esperar a que terminen los threads usando ***pthread_join()***, en el caso de que nos interese recoger algún resultado.
2. Decirle a la biblioteca *pthread* que cuando termine la ejecución de la función del thread, elimine todos los datos de sus tablas internas, usando ***pthread_detach().***

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

Esta función hace que el hilo invocador espere a que termine el hilo especificado. Supongamos que varios hilos están realizando un cálculo y es necesario el resultado de todos ellos para obtener el resultado total por parte del *main()* o hilo principal. Para ello el hilo encargado del resultado total debe esperar a que todos los demás hilos terminen.

La función *pthread_join()* tiene el siguiente prototipo[3]. Consulte en la Web el tratamiento de errores de esta función (EDEADLK):

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

Esta función bloquea el *thread* llamante hasta que termine la ejecución del *thread* indicado por *th*. Además, una vez éste último termina, pone en *thread_return* el resultado devuelto que estamos esperando.

- ***th:*** Es el identificador del thread que queremos esperar.
- ***thread_return***: Es un puntero a puntero (puntero doble) que apunta 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 si todo está correcto, o valor diferente de 0 si hubo algún error.

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

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 se llame a un *pthread_join()* o termine el hilo invocador. Cuando se ha recibido una hebra con *pthread_join()*, sus recursos que queden en el sistema, como el estado y valor devuelto por la misma, se liberan.

Si se hace una creación de hilos desde el *main()* es necesario poner en dicho *main()* un *pthread_join()* para que no se termine nuestro programa y se elimine de memoria antes de que terminen los hilos creados.

Por defecto un *thread* es ***joinable***, es decir, requiere que el hilo creador utilice *pthread_join()* para recoger su finalización, si no queda en estado *zombie*. **En caso contrario no tendremos control sobre nuestro programa** y depende del planificador el que nuestras hebras tengan "la suerte" o no de ejecutarse antes de que termine el hilo creador. Si una hebra *joinable* no se recibe con un *pthread_join() y* termina cuando el proceso principal aún no ha finalizado, su estado y determinados recursos del sistema asociados a la hebra seguirán estando ocupados hasta que el hilo o proceso principal termine.

Por último, si hace un *pthread_join()* de una hebra que ya está siendo esperada por otro *ptread_join()* obtendrá un error. Es responsabilidad del programador el hacer un solo *pthread_join()* por hebra.

El programa **"demo1.c"** crea dos hebras a las que se le pasa una estructura con un mensaje. Estúdielo y pruébelo. Después elimine los *pthread_join()* y observe sus resultados.

En **"demo2.c"** dispone de otro ejemplo del que sacar conclusiones, estúdielo y pruébelo.

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

La terminación de un hilo se produce cuando la función asignada al mismo termina. Eso ocurre cuando el hilo ejecuta un *return()* o cuando se llama a *pthread_exit()*.

Si la función asignada al hilo no ejecuta ni *return()* ni *pthread_exit()* a su finalización, automáticamente y de manera transparente para el programador, se ejecuta un *pthread_exit(NULL)*. **Por tanto, *pthread_exit()* y *return()* hace que el hilo invocador termine de manera normal sin causar que todo el proceso llegue a su fin.**

La función *pthread_exit()* invoca controladores de terminación de hilos, cosa que *return()* no realiza. Otra diferencia de *pthread_exit()* con respecto a *return()*, es que la primera se puede llamar desde cualquier subrutina que invoque la función de la hebra, haciendo que ésta termine, con *return()* se retornaría de la subrutina. Para estudiar con más profundidad las diferencias entre ambas llamadas consulte la Web.

Hay que tener en cuenta que *pthread_exit()* libera los recursos asociados a una hebra, pero no elimina por completo el estado en que terminó está y su valor devuelto, el cual se puede recoger posteriormente con *pthread_join()*.

El prototipo de *pthread_exit()* es el siguiente[4].

#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 (desaparece al terminar la hebra). Si necesitamos que cada hebra tenga datos sobre los que operar y devolver necesitaremos utilizar la función *malloc()* de C. Eso es porque el valor devuelto es un puntero, por lo que el dato debe estar en una zona de memoria localizable después de que termine la hebra, no puede ser una variable local a la hebra, ya que la variable se pierde al terminar su ejecución.

  Estudie los programas **"sample_OK.c"**, **"sample_FAIL.c"**, **"sample_OK_inapropiado2.c"** y **"sample_OK_inapropiado3.c"** dispuestos en Moodle, observe sus resultados y obtenga conclusiones.

  Los ficheros "**sample_OK_alternativa1.c**", "**sample_OK_alternativa2.c**" y "**sample_OK_alternativa3.c**" son diferentes formas de utilizar los tipos de datos de las variables que intervienen en las funciones de las hebras y lo que devuelven. Puede utilizar la manera que prefiera, siempre que funcione correctamente y razone su utilización.

Es importante que decir que si en una hebra usásemos *exit()*, **se terminaría el proceso completo con todas sus hebras**. Y por consiguiente, también podemos afirmar que si en un determinado momento **terminamos un proceso, automáticamente también se terminarán todas las hebras asociadas al mismo.**

Como nota final, decir que por razones de portabilidad y para que un valor entero no coincida con la macro PTHREAD_CANCELED (definida como un valor entero con *casting* a void) que devuelven las hebras que se cancelan, las hebras no deberían devolver número enteros, por lo que es mejor utilizar en estos casos un "*long*". En la documentación de Moodle y en la Web puede encontrar más información sobre este tema.

En "**demo3.c**" puede consultar un ejemplo en el que se espera a un hilo al que se le pasa como parámetro un vector de enteros y no devuelve nada. Estúdielo y ejecútelo. Modifique "**demo3.c**" cambiando el *pthread_exit(NULL)* por un *exit(0)*, observe sus resultados y saque conclusiones.

En "**demo4.c**", tiene otro ejemplo donde trabajar los conceptos asociados a *pthread_exit() y return()* en las hebras.

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

3.3.4 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 a *(unsigned long) pthread_t* para imprimir el ID.

En "**demo5.c**" dispone de un ejemplo.

3.3.5 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. **Cuando no nos interese el resultado de una hebra y queremos que automáticamente el sistema limpie su estado y tablas internas a su finalización**, sin tener que invocar a *pthread_join()*, tenemos que indicarlo con la función *pthread_detach()*. Así, una vez que el *thread* haya terminado, se eliminarán sus datos y tablas internas, y además su estado de terminación y devolución, liberándose espacio y recursos en tiempo de ejecución.

Cuando invocamos a *pthread_detach()* por parte de una hebra, no debemos invocar un *pthread_join()* de la misma, ya que si lo hacemos obtendremos un error y un comportamiento inesperado de nuestro programa. ¿Pero que pasa si el proceso principal termina su ejecución antes que la hebra que ha invocado a *pthread_detach()*?, pues que ésta continuará sin problema hasta que termine, siempre y cuando **finalicemos el proceso principal con la llamada *pthread_exit(NULL).*** En caso contrario es probable que la hebra no llegue a ejecutarse al terminar el proceso principal. Por tanto, *pthread_exit(NULL)* invocado en el *main()*, permite que las creadas que han invocado a *pthread_detach()*, no terminen de manera abrupta y no sean eliminadas del sistema hasta su finalización.

Dicho esto, puede pensar que ¿para qué poner entonces un *pthread_join(th_i,NULL)* en un *main()* en el caso de que no queramos recoger nada de una o varias hebras? La diferencia está en que si usa *pthread_exit(NULL)* en el *main()* no puede seguir realizando tareas en cuando se hayan finalizado las hebras, mientras que con *pthread_join(th_i,NULL)* si. Otro ejemplo está en que quiera realizar determinadas acciones solamente cuando hayan terminado de ejecutar todas las hebras, no antes, ya que necesita que se hayan hecho determinados trabajos previos para poder continuar.

5 http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_self.html

El prototipo de *pthread_detach()* es el siguiente[6]. Consulte en la Web los posibles valores que puede devolver esta llamada para el tratamiento de errores (EINTR):

```
#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, en "**demo6.c**" y "**demo7.c**", tiene un ejemplo del uso de *pthread_detach()*. Compílelo, ejecútelo, complete el tratamiento de errores consultando la Web y Moodle y observe sus resultados.

En "**demo8.c**" hay una pequeña modificación de "**demo7.c**" para que estudie y observe el paso de parámetros.

En "**demo9.c**" dispone de un ejemplo en el que puede observar que el uso de *pthread_join()* una vez invocado a *pthread_detach()*. POSIX no define su comportamiento, todo depende de la secuencia en que se produzca la ejecución de la hebra y el *main()*, **por lo tanto no debemos hacer uso de *pthread_join()* cuando invocamos a *pthread_detach()***. Es responsabilidad del programador el no realizar este tipo de implementaciones. Ejecute varias veces el programa y observe resultados.

3.3.6 Señal a una hebra desde el proceso llamador (*pthread_kill()*)

Para enviar a una hebra una señal desde el proceso que la crea podemos utilizar la llamada *pthread_kill()*[7]. Consulte en la Web los posibles valores que puede devolver esta llamada para el tratamiento de errores (EINVAL):

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

- **thread**: identifica el thread al cual le queremos enviar la señal.
- **signo**: MACRO o número de la señal que queremos enviar al thread.
  Podemos usar las constantes definidas en <signal.h>[8]. 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 algún ejemplo de *pthread_kill()*. Aunque pueda parecer útil a primera vista, la única utilidad que tiene SIGKILL 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. Se estudiarán algunos de estos mecanismo en sucesivas prácticas.

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

4 Ejercicios prácticos

4.1 Ejercicio1

Implemente un programa que cree un número *N* de hebras. Cada hebra creará 2 números aleatorios flotantes (consulte Moodle para la generación de aleatorios) y guardará su suma en una variable para ello, que será devuelta a la hebra principal o 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 2 números que va creando cada hebra y su suma, de forma que pueda comparar esas sumas parciales con la suma final que realizará el *main()* o hebra principal. Utilice macros definidas y comprobación de errores en sus programas (*errno* y comprobación de valores devueltos en cada llamada, con sus posibles alternativas).

4.2 Ejercicio2

Implemente un programa que cuente las líneas de los ficheros de texto que se le pasen como parámetros y al final muestre 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 cada fichero indicado por linea de argumentos, de forma que todos los ficheros se procesen de manera paralela, uno por cada hilo.

4.3 Ejercicio3

Implemente un programa que cree un vector de 10 elementos relleno con números aleatorios entre 1 y 9. Reparta ese vector entre 2 hebras o 5 hebras a partes iguales, según se indique por linea de argumentos un 2 o un 5, de forma que cada hebra sume la parte del vector que le corresponda y se lo devuelva al hilo principal, el cual mostrará la suma de los resultados devuelto por las hebras creadas.

Ejemplo de invocación del programa para crear 4 hebras que se repartan el vector: ./a.out 5

4.4 Ejercicio4

Implemente un programa que cree dos hebras y cada una incremente 10000 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 20000 – problema de concurrencia –). Intente razonar el resultado, el cual le servirá como concepto introductorio de la siguiente práctica de la asignatura.