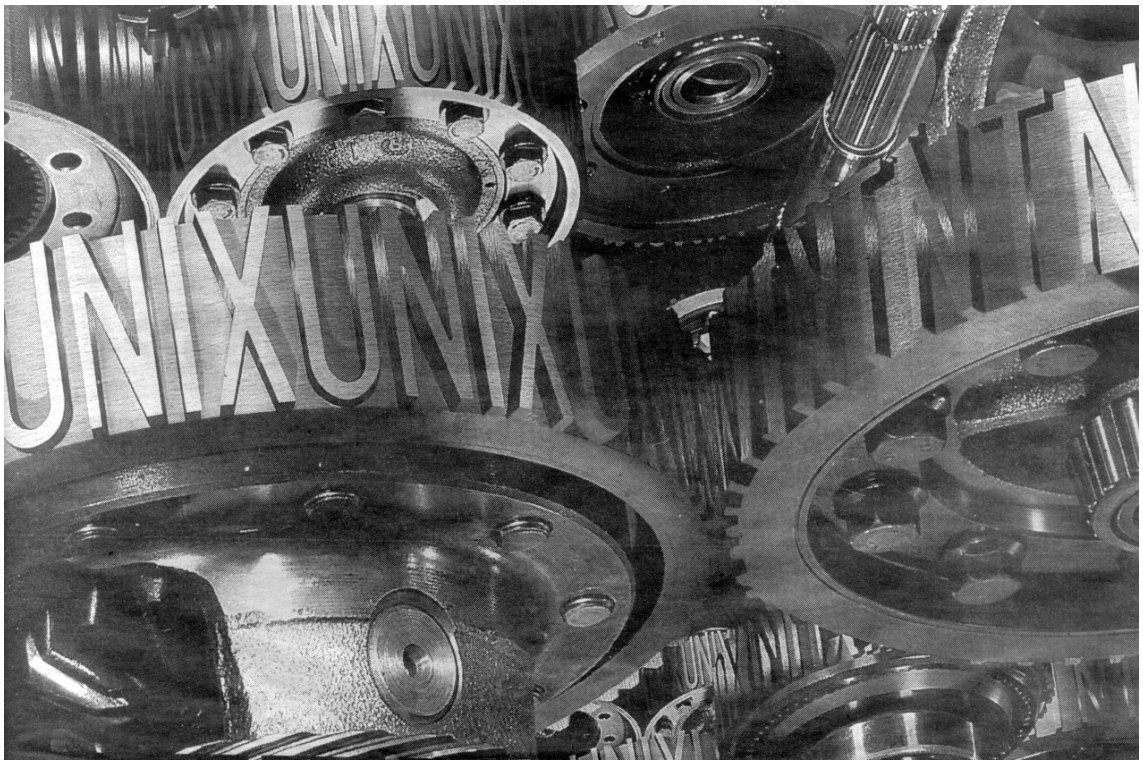
**Universidad de Cantabria**
**Grado de Ingeniería Informática**

# Sistemas Operativos

## Práctica 3 - Parte 1




Rafael Menéndez de Llano Rozas

Guion

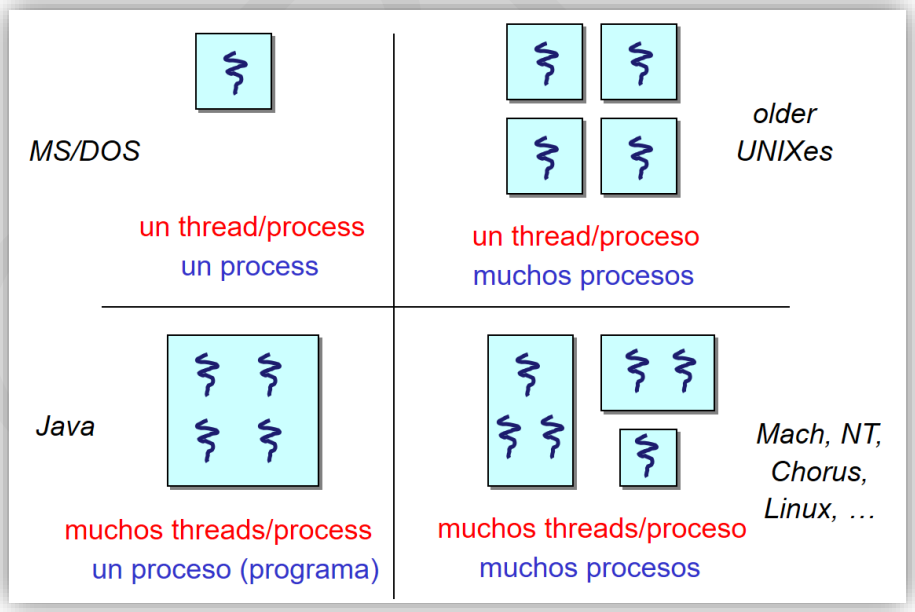
3

# Creación y sincronización de Hilos

En esta práctica verás qué son los hilos, cómo se pueden crear y sincronizar en la creación. Y que tipos de hilos existen.

## Introducción

Como has visto en teoría, la mayoría de los Sistemas Operativos modernos manejan dos entidades o conceptos: El proceso (*process*, *task*) y el hilo (*thread*), hebra o proceso ligero. El primero define un espacio de dirección global y una serie de recursos pedidos. El segundo define un flujo de control (programa secuencial) propio. En este tipo de sistemas los procesos siempre tienen al menos un flujo de control (en los sistemas que no tenían hilos, se considera que el proceso tiene uno).

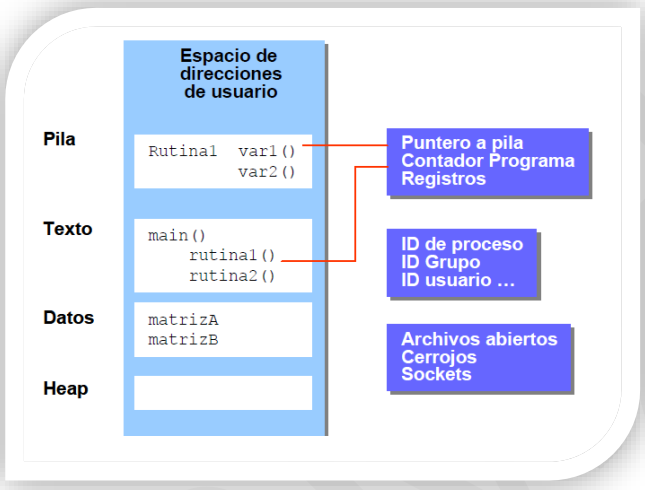



Cuando se trata de hacer un programa concurrente, podemos optar por hacerlo con procesos o hilos (existen otras formas como la programación asíncrona o la gestión de interrupciones mucho más incómodas). La ventaja de hacerlo con hilos es que son más rápidos en creación y planificación que los procesos y a diferencia de estos comparten un espacio de direcciones (el del propio proceso), con lo cual la comunicación es más sencilla (a través de memoria compartida).

Como hemos dicho, un hilo define un flujo de control propio, por lo tanto, necesita tener sus propios registros de CPU, en particular el PC, su propio *stack* (ya que está definido por una función que

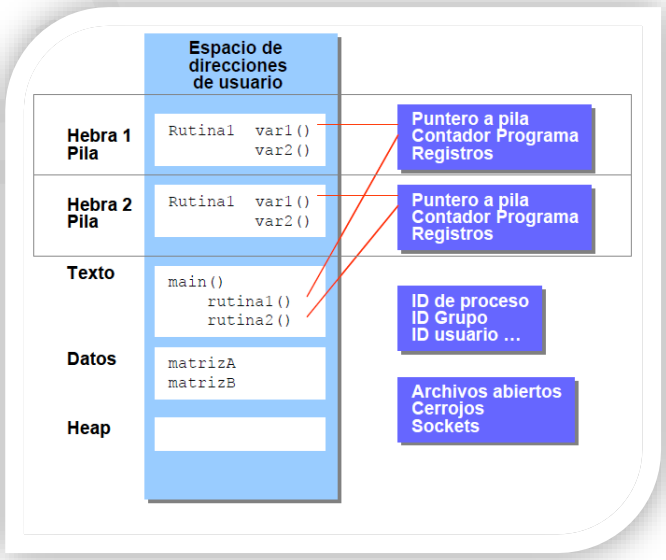
establece ese código secuencial que va a ejecutar y que además puede necesitar datos locales y por tanto su SP), y por último información sobre su planificación o datos propios relativos al hilo (como su identificación). Todos los demás recursos los comparte con el proceso al que pertenece, típicamente los ficheros que tiene abiertos, los segmentos de memoria, etc.

Un esquema comparativo de la memoria de un proceso clásico y de uno que soporte multihilo se puede ver en las siguientes dos figuras. El proceso tiene un segmento de texto (el de código) de sólo lectura, un *stack* (pila) donde se guardan las variables locales, un segmento de datos donde están las variables globales y un segmento de *heap* (montón) donde están las variables dinámicas, además tendremos los oportunos registros de la CPU y la información de contexto del proceso.



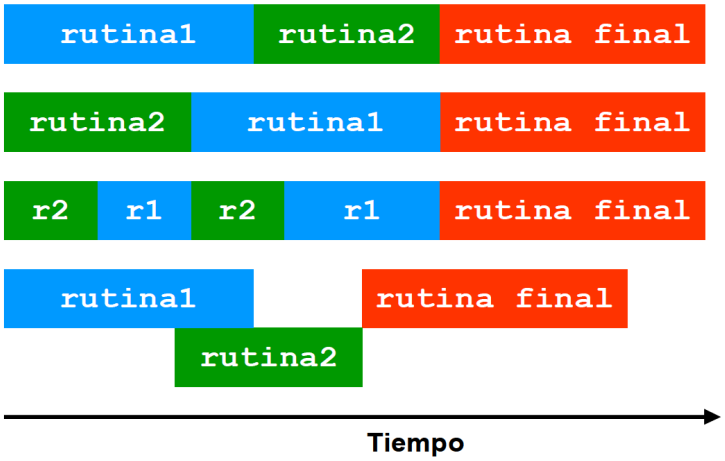

El proceso anterior también lo podemos ver como un proceso con un *thread*, donde el *thread* tiene los registros de la CPU y el resto de la información se deja al proceso pesado.

Si el proceso tuviera más de un *thread*, cada uno de ellos tendría su propia pila (para guardar los argumentos pasados a las funciones que ejecutan y variables locales que crean) y un sitio para guardar una copia de los registros que usan, pero podrían acceder a las mismas variables globales y recursos del proceso, como por ejemplo ficheros abiertos. De hecho, acceden al mismo segmento de texto donde está el código de sus funciones.

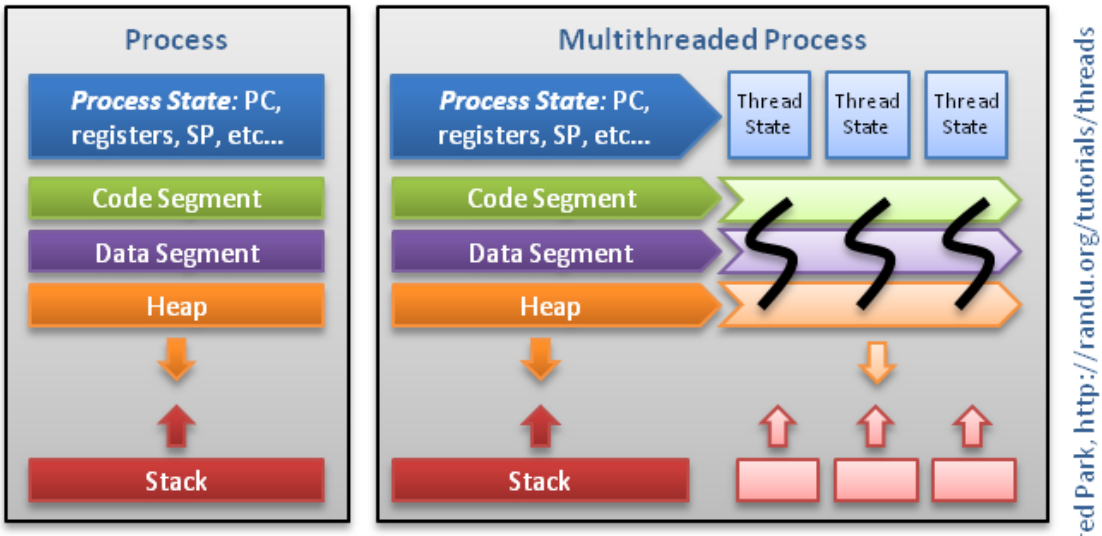


Una de las grandes ventajas de utilizar *threads* es que podemos ejecutar ese programa en un sistema multiprocesador, ubicando la ejecución de cada *thread* en un procesador (paralelismo real), con lo cual el tiempo de ejecución del programa será mucho menor[1]. De hecho, haciendo una analogía, se dice que los procesadores son los agujeros de las agujas por donde pasa el hilo y que una costura se puede hacer más rápido si dispones de dos agujas que trabajen en paralelo.

En la siguiente figura se ven las cuatro posibilidades que hay de ejecutar dos rutinas que no tienen dependencias (una final si la tiene) y que por tanto se podrían ejecutar a la vez. En las dos primeras no habría posibilidad de intercalarse, en la tercera tendríamos esa posibilidad, pero con un único procesador (paralelismo potencial) y en la cuarta con al menos dos procesadores ubicando la ejecución de cada *thread* en un procesador (paralelismo real).




Debes tener en cuenta que normalmente el *stack* y el *heap* están enfrentados y crecen en sentidos contrarios. Si llegan a cruzarse a mitad de camino se producirá un error (stack/ heap overflow).

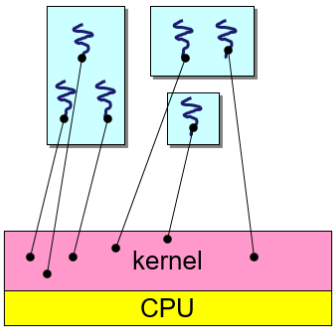



[1] Tanto la concurrencia como el paralelismo siempre les tendremos que aprovechar porque los procesos realizan operaciones de entrada/salida y dejan inactiva la CPU, porque muchos programas se ejecutan en respuesta a eventos que pueden ocurrir de forma simultánea (podemos dedicar un *thread* a cada evento) o porque en sistemas de tiempo real unas tareas son más importantes que otras (tareas o *threads* de distinta prioridad).

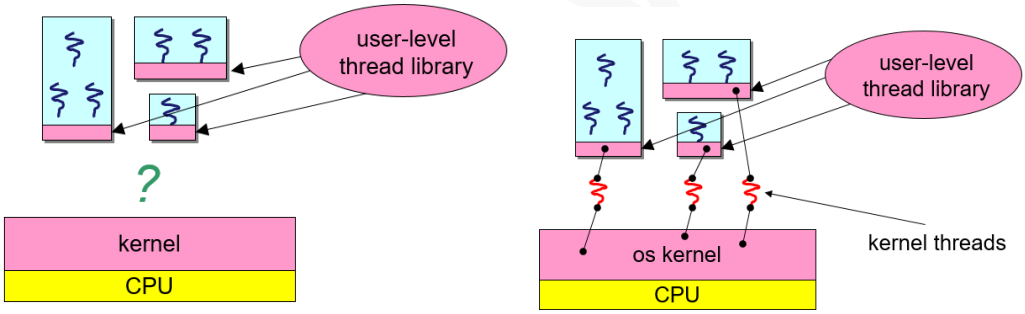
# POSIX 1003.1c: Utilización de pthreads

Ha habido muchas implementaciones de hilos que pueden clasificarse en tres grandes familias como se ha visto en teoría:

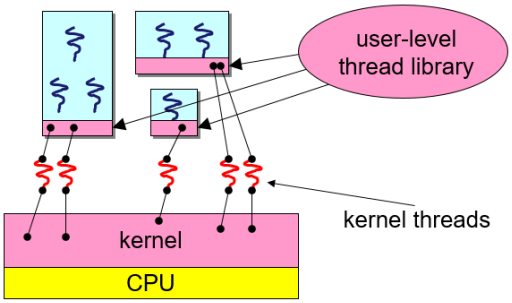
1. Hilos de tipo *kernel*. Se realizan con llamadas al *kernel* y por tanto éste tiene conocimiento de su existencia. Su creación y planificación es más rápida que la de procesos (un orden de magnitud). El mapeo entre el kernel y los hilos es 1:1.




2. Hilos de tipo librería / usuario. Se realizan con llamadas a procedimientos de una librería en el espacio de usuario y sin conocimiento por parte del *kernel*. Su creación y planificación es más rápida (otro orden de magnitud) que la de hilos *kernel* pero por el contrario, una llamada al sistema bloquea a todos los hilos del proceso (realmente, como se ve en la segunda figura, hay una conexión entre el *kernel* y los hilos de usuario que es el propio hilo que existen en cada proceso, por tanto la conexión es muchos:1).




3. Hilos mixtos. Trata de tomar ventaja de las dos anteriores. A los hilos *kernel,* en este caso, se les llama procesos ligeros (*Lightweight Process* - LWP) y pueden ser vistos como si se trataran de procesadores virtuales. Es el caso de Solaris y W7.




Independientemente de la implementación que exista y que proporcione el vendedor de hardware/SO, el programador necesita de una API (*Application Programming Interface*) que ofrezca las llamadas y tipos de datos necesarios para crear y trabajar con hilos. Y esta interfaz debe ser lo más estándar posible. Por ello IEEE creo en 1995 la norma POSIX 1003.1c, todos los hilos que la respeten se llaman

hilos POSIX o *pthreads*. Esta API contiene alrededor de una centena de funciones que pueden clasificarse en cuatro tipos de llamadas:

1. Manejo de hilos.
2. Mutex.
3. Variables Condicionales.
4. Sincronización (barreras y cerraduras –locks-).

Todas las funciones comienzan con la palabra "`pthread_`", necesitan un fichero de cabecera que las define <`pthread.h`> y pueden necesitar o no una librería específica, si es así se enlazará `pthread` (o `rt`) y si no estarán incluidas en nuestra vieja amiga `libc`. En el compilador gcc de GNU también existe la posibilidad de compilar con la opción `-pthread`.

Al compartir memoria, las llamadas devuelven el código del error (no hay una variable `errno` por *thread* sino por proceso), por lo tanto, para saber si una llamada ha funcionado bien, se ha de comparar con cero. Existen unas macros (`assert`) que nos permiten abortar un programa si el valor de la expresión no es cero (aborta si la expresión escalar es falsa).

Para su uso hay que incluir la cabecera `assert.h` y el prototipo y uso es:

```
void assert(scalar expression);
void assert_perror(int errnum);
rc = pthread_create( …              //una llamada de hilos
assert(rc == 0);                    //haciendo que cero sea verdad
```

También podemos usar las llamadas embebidas en una sentencia `if` y con un `exit` como cuerpo.

Hay ciertas cosas que no están definidas en el estándar y que limitan la portabilidad de las aplicaciones, entre ellas están por ejemplo el número máximo de *pthreads* que se pueden crear (se puede saber y cambiar con la orden `ulimit`) o el tamaño por defecto del *stack*. Aspectos como estos deben ser tenidos en cuenta para asegurar que nuestros programas sean universales.

## Creación y terminación de pthreads

Como hemos dicho, todos los programas en GNU/Linux tienen al menos un hilo por defecto que es el que ejecuta la función principal (`main`). A parte de él, cualquier otro hilo debe ser creado específicamente por el programador. Para ello se utiliza la llamada:

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

que básicamente nos devuelve un identificador de hilo (un *pthread* se identifica con un TID, que sólo es válido para *threads* del mismo proceso, internamente pueden tener un PID si se hacen con la llamada al sistema *clone)* y asocia el hilo a una función que tiene que ejecutar, esta función, como el caso de las señales, debemos asegurarnos de que tenga código reentrante. El tipo de dato para el identificador es `pthread_t` y por universalidad la función es de un tipo fijado general (`void *`) y toma un único argumento también general que es el mismo `void *`.

Existen dos tipos de *pthreads*:

- Los independientes (*detached*). Devuelven sus recursos cuando terminan.
- Los sincronizados ("*joinables"*). Mantienen sus recursos hasta que el hilo padre advierte su terminación con una llamada de sincronización `join`.

A la hora de crear un hilo se puede definir su tipo en el atributo `pthread_attr_t`. Por ahora lo dejaremos como NULL y se crearan por defecto de tipo sincronizado.

Los hilos también se pueden terminar, la llamada para hacerlo es `pthread_exit` (otra forma de que un pthread acabe es utilizar un `return` en la función que ejecuta que ya sabemos que es de tipo `void*`). El prototipo es:

```
void pthread_exit(void *retval);
```

De forma análoga al `exit` de procesos, se le da un argumento que es el código de salida que puede ser recogido por su hilo creador, en este caso puede ser de cualquier tipo siempre que se hagan las conversiones adecuadas, ya que es de tipo `void *`. En cuanto a esta terminación hay que tener en cuenta que:

- Una vez usada ya no se puede acceder a sus variables locales (eliminación de su *stack*).
- Si se utiliza en el hilo padre, el proceso no terminará hasta que no terminen todos los hilos hijo, pero si no se usa y el proceso termina, terminarán forzadamente todos sus hilos.

Usaremos estas llamadas en el ejemplo más típico que se realiza al empezar, que no es otro que crear un programa “hola mundo”:

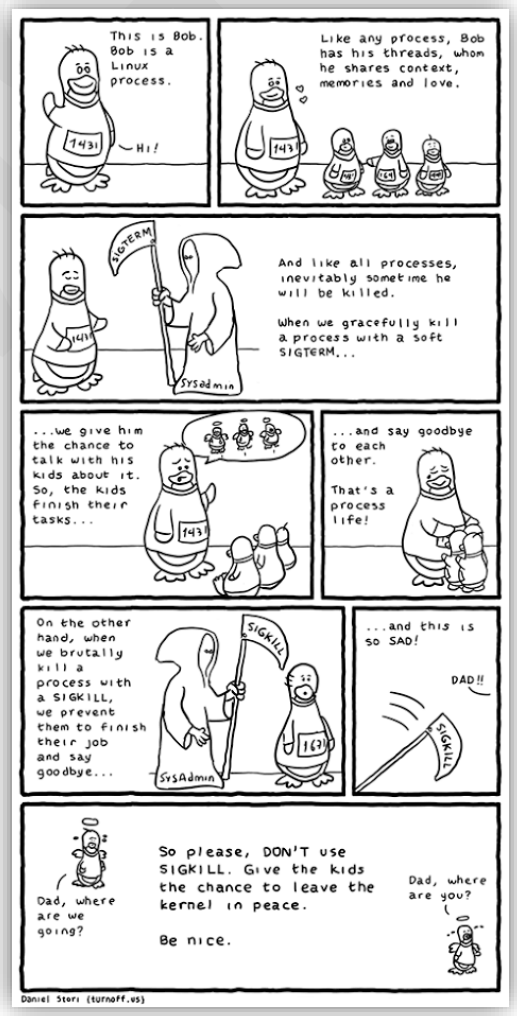
```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#define NUM_THREADS     5

void * PintaHolaMundo(void *identificador)
{
      long tid;
      tid = (long)identificador;
      printf("Hola Mundo! Soy yo #%ld!\n",
tid);
      pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
      pthread_t threads[NUM_THREADS];
      int rc;
      long t;
      for(t=0; t<NUM_THREADS; t++)
      {
            printf("En main: creo thread
%ld\n", t);
            // se puede hacer con
      assert(rc==0);
            rc = pthread_create(&threads[t],
NULL,

PintaHolaMundo, (void *)t);
            if (rc)
            {
                  printf("ERROR is %d\n",
rc);
                  exit(1);
            }
      }

      pthread_exit(NULL);
}
```



Para realizar el programa hemos incluido las cabeceras `pthread.h / assert.h` y hemos utilizado la llamada al sistema `pthread_create` de tipo entero para realizar la creación, la cual toma 4 argumentos:

1. Un puntero a la variable de tipo `pthread_t` que servirá de llave (identificador o TID) para utilizar a partir de ese momento el *thread* (análogo al PID de procesos).
2. Un puntero a *struct* conocido como objeto de atributos de *thread* (existen una serie de llamadas para definirlo), que indicará varias características del thread: tamaño y dirección de inicio del *stack* y control de recursos devueltos. Si lo ponemos a NULL, aceptaremos las opciones por defecto, que son que el hilo sea síncrono ("*joinable*") y de planificación normal (no de tiempo real).
3. Un puntero donde se empezará a ejecutar el *thread* (hace las funciones de PC) que en C será normalmente una dirección a una función. Para ser lo más genérica posible, devuelve un puntero a *void* y toma como argumento un puntero a *void*. Si las funciones no son de tipo `(void *)` habrá que usar los convertidores de tipo (`type cast`) adecuados.
4. Un puntero al parámetro que será pasado a la rutina del *thread*. Si se necesita pasar varios argumentos siempre se puede pasar una estructura (se puede convertir a `void*`).

Un hilo también puede acceder a las variables globales definidas en el programa (antes del `main`). Siempre se debe tener cuidado con este tipo de estrategia y asegurarse de acceder de forma segura a estas variables. Es buena costumbre re-declararlas con `extern`.

1. Bájate el programa `hola.c` y entiende lo que está desarrollado en él. Compila el programa como se te ha dicho (incluyendo la librería). Comenta la línea del `pthread_exit` del padre e indica qué ocurre y por qué. ¿Por qué crees que se utilizan variables de tipo `long` en el argumento en máquinas de 64 bits? Copia y cambia el programa para pasar el parámetro `t` por referencia (puntero). ¿Funciona el programa? ¿Por qué funcionaba antes? Cambia de nuevo el programa para pasar a cada hilo un parámetro distinto por referencia.

2. Copia y cambia de nuevo el programa para que se utilice la macro `assert` en vez de la sentencia `if`, además en vez de pasar el identificador pasa un *string* con lo que tiene que pintar el hilo.

Hay varias maneras de definir la función y su argumento, debes tener en cuenta que con el tipo `void*` no se puede hacer nada (salvo la asignación), por eso es necesario hacer las conversiones pertinentes con el tipo de dato que realmente queremos pasar. Esto se puede hacer en dos sitios o de dos maneras diferentes:

- Dejando la función tal cual (como el ejemplo) y haciendo internamente el "*cast*" y en el cuarto argumento de `pthread_create`.
- Definir la función como queremos realmente y en la llamada a `pthread_create` hacer el *cast* en el tercer (el de la función) y cuarto argumento. Desgraciadamente con este formato aparecerán al compilar múltiples *warnings*, aunque generará el ejecutable. Se pueden evitar forzando con `void *` aunque la función ya sea de ese tipo.

3. Copia y cambia de nuevo el programa para hacer una función de `pthread` que tome un argumento de tipo `long`. Evita los warnings. Introduce una llamada bloqueante en el hilo como `sleep` y según el comportamiento indica qué implementación de hilos tienes.

Como hemos visto, hay dos tipos de *threads* a los que se puede esperar a que terminen y recupera su código de salida (sincronizados) o no (independientes). Además, tienen dos estados posibles para controlar la devolución de recursos al sistema:

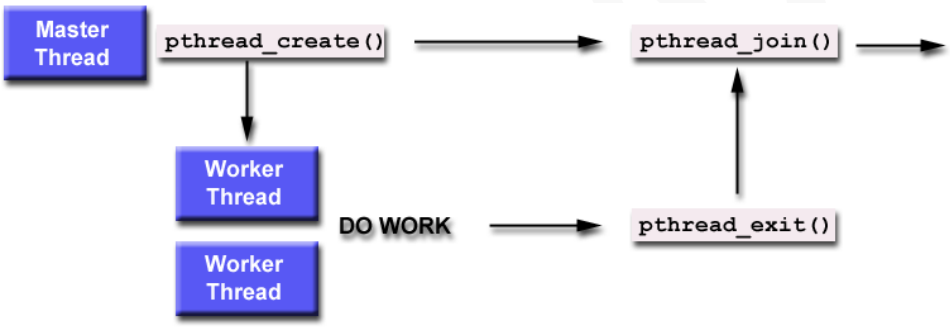
- "*detached*" (independiente): Cuando el p*thread* termina, devuelve al sistema los recursos utilizados (Bloque de control, stack, etc.), no se puede esperar la terminación de este *pthread* con ninguna llamada.
- "*joinable*" (sincronizado): Cuando el p*thread* termina, mantiene sus recursos, los recursos se liberan cuando su padre llama a `pthread_join()` para sincronizarse (análogo a `wait`).

El carácter del p*thread* se define en el objeto de atributos que hemos utilizado en la llamada para crearlos (se verá con más detalle en el siguiente apartado). Una vez creado y definido el mismo, el padre puede esperar a su terminación con la `pthread_join()`.

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

Toma dos argumentos, el primero será el identificador de p*thread* al que queremos esperar y el segundo pasará por referencia un puntero a `void *`, por lo que se convierte en un doble puntero donde obtendremos el valor de `exit` del p*thread*. Es una situación análoga a la pareja exit-wait de procesos, pero en este caso con el tipo de dato `void`*.

Sólo cuando hacemos `join` en el hilo padre se liberarán los recursos obtenidos por el hilo hijo, hay que tener en cuenta que sólo un hilo "*joinable*" puede ser sincronizado con `join`.




En el siguiente ejemplo se utilizan dos hilos que devuelven en su terminación códigos al padre para que sean tratados. El padre les espera y se sincroniza con su terminación. Además, en cada hilo se utiliza una forma de función de las vistas anteriormente:

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

void *haz_una_cosa(void *); /* cumple el prototipo mirad los type cast */
void haz_otra_cosa(int *);  /* no lo cumple */
void junta (int, int );

int ret1, ret2;  /* ojo no se pueden poner en el stack de las funciones */

int main()
{
    pthread_t  thread1, thread2;
    int  *st1, *st2;        /* recoger el status de exit */
    int r1=0, r2=0;

       /* los creo se podían poner en un if o assert*/
    pthread_create(&thread1, NULL, haz_una_cosa, (void *) &r1);
    pthread_create(&thread2, NULL, (void *)haz_otra_cosa, (void *) &r2);

       /* los sincronizo en su terminacion */
    if (pthread_join(thread1, (void **)&st1)!= 0)
       perror("join error");
```

```
    if (pthread_join(thread2, (void **)&st2) != 0)
       perror("join error");

    junta (r1, r2);   /* sumo lo que cambian los hilos */

    /* ver las direcciones y los codigos de salida */
    printf("dir1=%p, dir2=%p, status1=%d, status2=%d\n",
                                                st1, st2, *st1, *st2);

    pthread_exit(NULL);
}

void *haz_una_cosa(void* puntero)
{
    int i; /* en el stack del hilo */
    extern int ret1; /* debe ser global para que permanezca */

    puts("haciendo una cosa");
    for (i = 0; i < 4; i++)
        (*(int*)puntero)++; /* fijarse en el type cast */
    ret1=23;    /* se devuelve algo, normalmente el codigo de error */
    return(&ret1);   /* una forma de terminar el hilo */
}

void haz_otra_cosa(int *puntero)
{
    int i;
    extern int ret2;

    puts("haciendo otra cosa");
    for (i = 0; i < 6; i++)
         (*puntero)++;
    ret2=25;
    pthread_exit(&ret2);  /* otra forma de terminar */
}

void junta(int unas, int otras)
{
    int total;

    total = unas + otras;
    printf("junta: unas %d, otras %d, total %d \n", unas, otras, total);
}
```

Date cuenta que lo que se puede devolver en la llamada `pthread_exit` tiene que permanecer después de la terminación, por lo que devolver cualquier dato almacenado en el stack del hilo es erróneo (debería ser una variable global). Lo que resta bastante utilidad a devolver algo de esta manera.

4. Bájate el programa `junta.c` y entiéndelo. Observa dónde se sitúan las variables para recoger los códigos de salida. Indica si el código de error de un hilo que se pasa por el *exit* puede ser una variable local o no. Cambia las llamadas a `pthread_create` para meterlas en un `if` o un `assert` de comprobación. Entiende qué significa la palabra reservada `extern` del ANSI C. Entiende las dos formas de definir la función.

Hemos dicho que a un hilo se le pueden dar varios argumentos, pero que estos deben ser pasados en una `struct` convertida a puntero a `void`. En el siguiente ejemplo, vemos como dos p*thread*s pueden cooperar para trabajar con un `array` y terminar conjuntamente. En la estructura se mete un *subarray* que es una dirección y una longitud. Además, para que cada hilo tenga su código de salida, también se incluye en la misma.

```
/* prueba para ver como se pasan varios argumentos a un hilo */
#include <stdio.h>
#include <pthread.h>

typedef struct {
        int *ar;
        long n;
        int exito;
} subarray;

void * incrementer (void *arg)
{
        long i,j;
        for (j=0; j< 10000; j++)
         for (i=0; i< ((subarray *)arg)->n; i++)
            ((subarray *)arg)->ar[i]++;   /* ojo a las conversiones */
        ((subarray *)arg)->exito=1;
        pthread_exit (&((subarray *)arg)->exito);
}

int main()
{
        int ar [1000000];
        pthread_t th1,th2;
        subarray sb1,sb2;
        int  *st1, *st2;
        /* testigo de cambio */
        printf("Antes de llamar el valor del 3500 es %d \n",ar[3500]);

        /* hago primera rodaja */
        sb1.ar = &ar[0];
        sb1.n = 500000;
        if (pthread_create(&th1, NULL, incrementer, &sb1) != 0)
          perror("creacion del thread uno");

        /* segunda rodaja */
        sb2.ar = &ar[500000];
        sb2.n = 500000;

        if (pthread_create(&th2, NULL, incrementer, &sb2) != 0)
            perror("creacion del thread dos");

        /* el valor de exit tambien lo puedo obtener en el subarray */
        if (pthread_join (th1, (void **)&st1) != 0)
             perror("join error");

        if (pthread_join (th2, (void **)&st2) != 0)
            perror("join error");

        /* testigo de final comprobacion */
        printf("Despues de llamar el valor del 3500 es %d \n",ar[3500]);

        printf("Hijos terminados con status1=%d; status2=%d\n", *st1,*st2);
        printf("Debe salir lo mismo con sub.exitos:  %d  %d \n",
                              sb1.exito,sb2.exito);
}
```

5. Bájate el programa `subar.c` y describe para qué se utiliza este programa. Entiende qué es y cómo se construye el tipo de dato subarray. Entiende cómo hemos pasado el tipo de dato como un puntero a `void` y esto nos obliga a hacer type *cast* dentro de la función del hilo. Cambia el código para que sólo haya una variable `exito`. ¿Es esto correcto?

## Manejo de threads y cambio de comportamiento

A la hora de crear los *pthreads* con los atributos o después de su creación, el comportamiento de los mismos se puede cambiar en los siguientes aspectos:

- Carácter de la sincronización en la terminación: independiente "detachable" o síncrono "joinable".
- Planificación. Se puede seleccionar la política (algoritmo) de planificación de *pthreads* y su configuración, así como si esta planificación es heredable o no. Además de poner el ámbito (alcance) de la misma. Esto será necesario en casos de tiempo real.
- Stack. Se puede configurar la dirección del stack y su tamaño; y definir en tamaño una cantidad suficiente para evitar *overflows*. Si se quieren hacer programas seguros y portables, no se puede depender del tamaño del *stack* por defecto y es mejor asignarlo explícitamente si la implementación nos lo permite.

Aquí sólo nos referiremos al primer punto, ya que es necesario para hacer la creación de *threads* (para el resto de opciones se puede mirar el manual).

Para crear un *pthread* es preciso definir sus atributos en un objeto especial, el objeto de atributos, que debe crearse antes de usarlo con la llamada a `pthread_attr_init()` y puede borrarse con `pthread_attr_destroy()`:

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
```

Una vez creado, se pueden modificar o consultar los atributos concretos del objeto (depende de la implementación) que son:

- El tamaño de *stack* mínimo (opcional).
- La dirección del *stack* (opcional).
- El control de devolución de recursos ("*detach state*").
- La política (scheduling) de threads.

Para hacerlo existen las llamadas que aparecen a continuación (los valores de los atributos se rellenan con constantes definidas en la librería `pthread.h`).

```
int pthread_attr_setstacksize (pthread_attr_t *attr, size_t stacksize);
int pthread_attr_getstacksize (const pthread_attr_t *attr, size_t *stacksize);

int pthread_attr_setstackaddr (pthread_attr_t *attr, void *stackaddr);
int pthread_attr_getstackaddr (const pthread_attr_t *attr, void **stackaddr);

int pthread_attr_setdetachstate (pthread_attr_t *attr, int detachstate);
int pthread_attr_getdetachstate (const pthread_attr_t *attr, int *detachstate);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const    pthread_attr_t *attr, int *policy);

int pthread_attr_setschedparam(pthread_attr_t *attr,
                                          const struct sched_param *param);
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                                          struct sched_param *param);

int pthread_attr_setinheritsched(pthread_attr_t *attr, int inherit);
int pthread_attr_getinheritsched(const    pthread_attr_t  *attr, int *inherit);

int pthread_attr_setscope(pthread_attr_t   *attr,   int  scope);
int pthread_attr_getscope(const pthread_attr_t *attr,  int *scope);
```

Como ejemplo de utilización del atributo para la creación de p*threads* de tipo independiente se puede examinar el siguiente ejemplo:

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

/* Thread que pone periódicamente un mensaje en pantalla, el periodo se le
pasa como parámetro */

void * periodic (void *arg)
{
        int period;

        period = *((int *)arg);
        while (1)   /* hasta que termine el proceso */
        {
                printf("En el thread con periodo %d\n",period);
                sleep (period);
        }
}

/* Programa principal que crea dos threads period */
int main ()
{
       pthread_t th1,th2;
       pthread_attr_t attr;  /* para los atributos */
       int period1 =2, period2=3;

        /* Crea el objeto de atributos */
       if (pthread_attr_init(&attr) != 0)
              perror("error de creación de atributos");
       if (pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED)!= 0)
              perror("error de creación de atributos");

        /* Crea los threads detachados*/
       if (pthread_create(&th1,&attr,periodic,&period1) != 0)
              perror("error de creación del thread uno");
       if (pthread_create(&th2,&attr,periodic,&period2) != 0)
              perror("error de creación del thread dos");

        /* Les deja ejecutar 30 segundos y termina */
       sleep(30);
        /* si se pone esto reconoce a los hijos y no termina nunca */
       pthread_exit(NULL);
       puts("thread main terminando");
       exit (0);
}
```

El carácter del *thread* se puede cambiar más fácilmente con la llamada `pthread_detach()` que toma como argumento el identificador del *thread* una vez que ha sido creado.

```
int pthread_detach(pthread_t thread);
```

6. Bájate el programa `atrib.c` y describe para qué se utiliza este programa. Prueba a comentar el `sleep` y el `pthread_exit`. ¿Qué ocurre? Cambia el programa para en vez de utilizar el atributo, crearlos por defecto y modificarlos con *detach*.

7. Opcional. Copia y modifica el programa en que un proceso padre creaba a varios procesos hijo y cambia el valor del número de procesos a 100. Los procesos hijo en este caso no hacen nada. Anota con el comando `time` su tiempo de ejecución. Haz lo mismo con hilos. Ahora ponlos con atributos detach y mide. Cambia los hilos a tipo independiente con detach y mide. Compara las cuatro cosas.

8. Opcional. Copia y modifica el programa `atrib.c` para que después de crear los hilos se haga un `fork`, indica si en el proceso hijo se han heredado los hilos del padre o no.

9. Opcional. Crea un programa con hilos a los que se da el *stack* para trabajar. En sus funciones ellos pintan el *stack* que tienen asignado.

## Otras llamadas

Otra información básica que se puede obtener de un p*thread* es su identificación con la llamada `pthread_self(void)` que devuelve un dato de tipo `pthread_t`.

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

También un p*thread* puede ser cancelado por otro con `pthread_cancel()` (se obtendrá en el status de *pthread_join* `PTHREAD_CANCELED`).

```
int pthread_cancel(pthread_t thread);
```

Y al igual que con los procesos podemos hacer que cedan la CPU.

```
int sched_yield (void);
```

En programación (*threads*) de **tiempo real** se puede cambiar la política de planificación. Hay dos llamadas que nos indicarán dónde se realiza la planificación, si en el proceso o en el sistema (equivalente a threads de tipo kernel / librería).

```
pthread_attr_getscope()
pthread_attr_setscope()
```

A las que podremos dar dos constantes:

```
PTHREAD_SCOPE_PROCESS o PTHREAD_SCOPE_SYSTEM
```

También podemos indicar si la planificación la heredamos del padre o ponemos la nuestra:

```
pthread_attr_getinheritsched()
pthread_attr_setinheritsched()
```

con dos constantes:

```
PTHREAD_INHERIT_SCHED     /      PTHREAD_EXPLICIT_SCHED
```

Y por último podemos obtener o poner el algoritmo de planificación y la prioridad:

```
pthread_attr_getschedpolicy     // que algoritmo
pthread_attr_setschedpolicy()   // poner algoritmo
pthread_attr_getschedparam()    // que prioridad básicamente
```