# Programación de Sistemas (PrSist)

Departamento de Informática de Sistemas y Computadoras (DISCA)
*Universitat Politècnica de València*

# Práctica 3
# Creación de procesos y threads

**Contenido**



## 1 Objetivos

La creación de procesos y threads es uno de los servicios más importantes que podemos encontrar en un sistema operativo multiprogramado. Es precisamente la capacidad de ejecutar varios procesos de forma concurrente la que se busca cara a mejorar el rendimiento del sistema. Desde el punto de vista del usuario es necesario que los servicios relacionados con los procesos e hilos ofrezcan la interfaz correspondiente de programación para su uso en las aplicaciones. Esta interfaz de aplicación es la que vamos a estudiar en la presenta practica. En concreto los objetivos de la misma son:

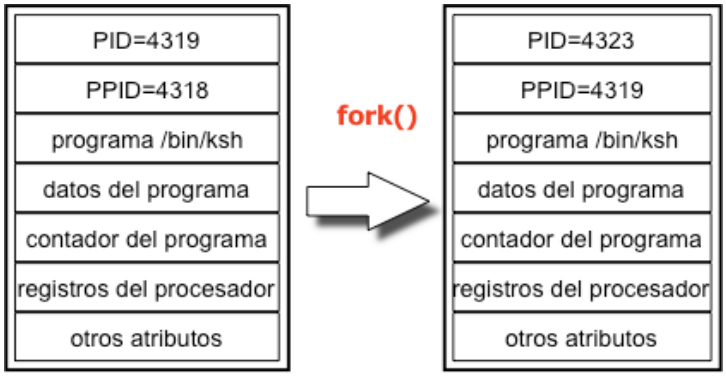
- Adquirir destreza en la utilización en código de las llamadas fork(), exit(), wait(), pthread_create, pthread_exit y pthread_join.
- Analizar mediante programas de ejemplo el comportamiento de las llamadas en términos de valores de retorno
- Adquirir experiencia en el manejo de las funciones del estándar POSIX para creación y espera de hilos, trabajando con un escenario donde se produzcan operaciones concurrentes.

## 2 Introducción

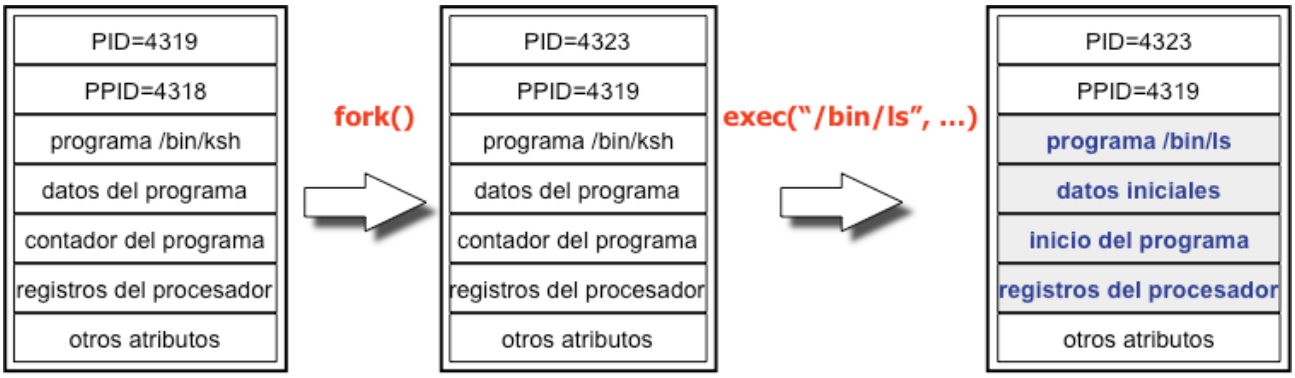
En UNIX el mecanismo para la creación de procesos está basado en la clonación del proceso que desea crear en otro. Bajo este concepto de clonación se esconde una implementación que crea una nueva estructura de soporte a la tarea (PCB) en el sistema a partir de la copia de los datos del proceso que invoca a llamada (llamada a sistema fork() en el proceso padre). De esta forma ambos son iguales excepto en los identificadores de proceso (PID) y

de padre de proceso (PPID). Como consecuencia de esto al proceso hijo se le facilita una copia de las variables del proceso padre y de los descriptores de fichero. Es importante destacar que las variables del proceso hijo son una copia de las del padre pero en localizaciones de memoria diferentes por lo que modificar una variable en uno de los procesos no se refleja en el otro. Como resumen podemos decir que En el momento de la llamada a fork() el proceso hijo:

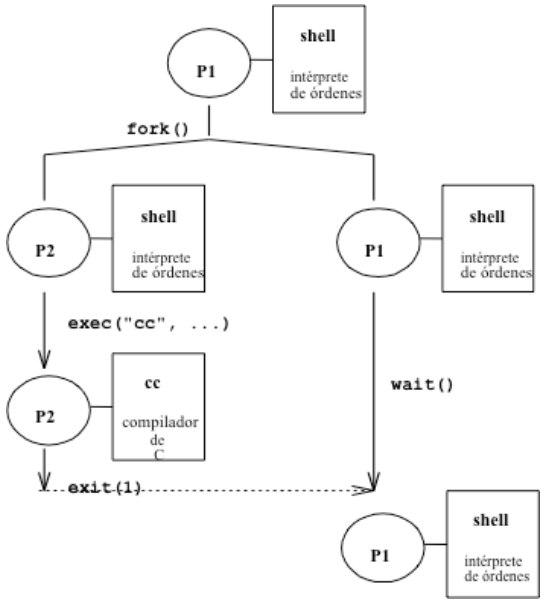
- es una copia exacta del padre excepto el PID y PPID.
- tiene las mismas variables y ficheros abiertos
- las variables son independientes
- los ficheros son compartidos (heredan el descriptor)




Hasta este punto la funcionalidad del sistema operativo es insuficiente ya que tan solo nos permite crear procesos iguales unos de otros por lo que todos ellos ejecutarían el mismo código. Es necesario por tanto una llamada a sistema que nos permita cambiar lo que el proceso va a ejecutar. De forma más correcta definiremos este servicio como aquel que permite cambiar el mapa de memoria de un proceso. Este servicio es proporcionado por la llamada exec() en alguna de sus variantes. Hay que destacar que la llamada a exec() no crea ningún proceso nuevo, si no que modifica la memoria del que la invoca cargando las regiones de datos, código etc de un fichero de disco que debe contener un programa ejecutable válido. Mediante la interpretación de dicho programa ejecutable (en formato ELF por ejemplo) el sistema operativo crea las regiones correspondientes y cargar su valores con los datos del fichero.




La creación de procesos genera dentro del sistema un árbol de parentesco entre los mismos y que vincula a los procesos en términos de sincronización. Un caso de aplicación típico de la sincronización entre padre e hijo es el caso del Shell.




El Shell de Unix es un proceso el cual crea un proceso hijo por cada comando invocado desde la línea, debiendo quedar a la espera de que el proceso que ejecuta el comando invocado termine para volver a imprimir el prompt por pantalla (excepto en la ejecución en background) y quedar a la espera de un nuevo comando. Esta sincronización entre Shell y proceso hijo en ejecución se realiza mediante las llamadas wait (o su variante waitpid) y exit.

La llamada al sistema exit finaliza la ejecución de un proceso y devuelve el valor de estado al proceso padre. Si el proceso padre del que ejecuta la llamada a exit esta ejecutando una llamada a wait, se le notifica la terminación de su proceso hijo y se le envían los 8 bits menos significativos de la variable usada en el exit.

Exit es una de las pocas llamadas que no devuelve ningún valor al proceso que la invoca. Es lógico, ya que el proceso que la llama deja de existir después de haberla ejecutado.

La llamada al sistema wait suspende la ejecución del proceso que la invoca hasta que alguno de los procesos hijo haya terminado.

Un proceso puede terminar en un momento en el que su padre no le esté esperando. Como el núcleo debe asegurar que el padre pueda esperar por cada proceso, los procesos hijos por los que el padre no espera se convierten en procesos zombis. Cuando el padre realiza una llamada wait, el proceso hijo es eliminado de la tabla de procesos. De forma similar si un proceso termina dejando procesos en ejecución, estos son adoptados por el proceso INIT (de PID 1) con el objeto de que sea este proceso el que recoja los valores devueltos en las llamadas a exit.

# 3 Creación de procesos en UNIX mediante la llamada fork().

**Ejemplo 1: en este ejercicio se introduce la llamada fork()**

Escribe en el fichero pr4-1.c el siguiente código que crea un proceso mediante la llamada `fork()`.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
     printf("Soy el proceso %ld antes de crear otro proceso\n", (long)getpid());
     fork();
     printf("Soy el proceso %ld y mi padre es %ld\n",(long)getpid(),
(long)getppid());
     sleep(15);
     return 0;
}
```

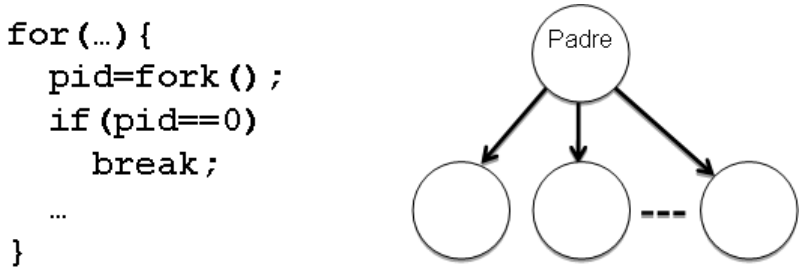
Compílalo mediante la orden:
gcc -o pr4-1 pr4-1.c
Ejecútalo en background:
./pr4-1 &

Ejercicio 1: Ejecuta la orden ps antes de que se consuma el sleep de 15 segundos y comprueba los PID y PPID's de los dos procesos tanto con el comando "ps l" como con la salida generada del programa.
Anotar en la tabla los valores de PID, PPID y COMMAND devueltos por la orden ps asociados a los procesos del ejercicio

| | PID | PPID | COMMAND |
|---|---|---|---|
| Proceso Padre | | | |
| Proceso hijo | | | |

**Ejemplo 2: Valor de retorno del fork**
En este ejercicio se va analizar el valor de retorno de la llamada `fork()` para controlar en código si la ejecución es del padre o del hijo. El ejemplo se basará en la creación de procesos en abanico. La siguiente figura ilustra el concepto.


De forma general para un valor dado de procesos el esquema del código sería el siguiente:




Así mismo se va a comprobar el efecto que tiene en el estado de los procesos la terminación de los mismos cuando el padre no ha realizado la llamada a `wait()`.
Escribe el programa pr4-2.c que realice lo siguiente:
Crea 5 procesos en abanico (consulta las transparencias del seminario 3).

- Cada hijo debe escribir la frase "soy el hijo número id" siendo id un identificador único asociado a cada proceso hijo, y después deben terminar su ejecución con una llamada a `exit(0)`.
- El proceso padre debe realizar un `sleep(25)` antes de terminar su ejecución.

Compila y ejecuta el programa pr4-2 en background e inmediatamente ejecuta la orden "`ps l`". Anota en la tabla los resultados. ¿Cuál es el estado de los procesos hijo generados por el programa? ¿Por qué?

| | PID | PPID | COMMAND | ESTADO |
|---|---|---|---|---|
| Proceso Padre | | | | |
| Proceso hijo 1 | | | | |

| | | | | |
|---|---|---|---|---|
| Proceso hijo 2 | | | | |
| Proceso hijo 3 | | | | |
| Proceso hijo 4 | | | | |
| Proceso hijo 5 | | | | |

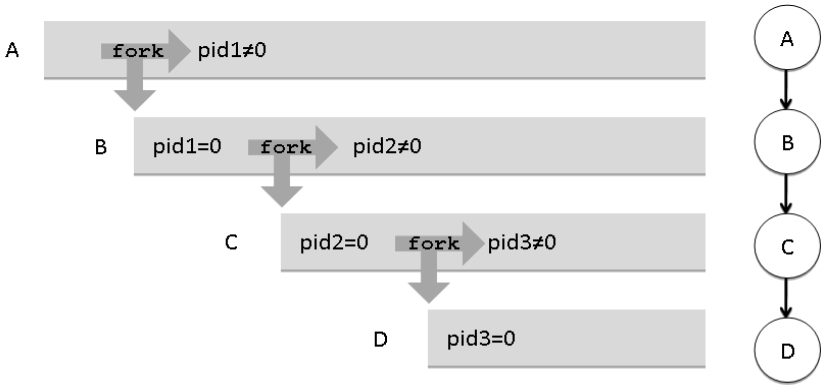
**Ejercicio 3:**
En este caso vamos a analizar sobre el mismo ejemplo anterior el efecto sobre le estado de los procesos cuando el padre finaliza antes de que termine la ejecución de los hijos que ha lanzado. Para ello copia el código anterior en el un fichero pr4-3.c
Ahora añade un `sleep(20)` antes del `exit(0)` que realiza cada proceso hijo. Asegúrate de que el valor del sleep de los hijos es mayor que el del padre. Ejecuta pr4-3 en background y luego ejecuta la orden "`ps l`" (para poder ver tanto el PID como el PPID de cada proceso) varias veces antes y después de que se haya consumido el sleep del proceso padre. Fíjate en el PPID de los procesos hijos (antes y después de finalizar el padre). Anota los resultados en la tabla. ¿Han cambiado? ¿Por qué?

| | PID | PPID | | COMMAND | ESTADO |
|---|---|---|---|---|---|
| Proceso Padre | | | | | |
| Proceso hijo 1 | | | | | |
| Proceso hijo 2 | | | | | |
| Proceso hijo 3 | | | | | |
| Proceso hijo 4 | | | | | |
| Proceso hijo 5 | | | | | |

# 4 Espera de procesos

En el apartado anterior hemos visto cómo se generan procesos huérfanos y zombies. En el caso de los procesos zombies, no es conveniente que se generen ya que pueden acarrear una sobrecarga en el sistema.
La generación de un proceso zombie se da cuando el padre no está esperando en una llamada `wait()` cuando uno de sus hijos realiza la llamada a `exit()`. Para evitarlo, basta que se realice la llamada al sistema `wait()` o `waitpid()` para todos los procesos hijos generados.

**Ejercicio 4:**
Modifica el programa pr4-2 (llámalo pr4-4.c) para que el proceso padre espere a sus hijos antes de que estos terminen y no se generen procesos zombies. Compruébalo ejecutando el programa en background y con la orden ps como en el ejercicio 2.
**Ejemplo 3:**
La llamada exit permite devolver al padre un código de retorno en el cual el hijo puede pasar información sobre su finalización. En este ejercicio vamos a usar dicho código de retorno para retornar un valor de finalización entre procesos creados en profundidad. Estos procesos responden a un esquema de código como el que podemos ver en la figura:




```
for(i=0;i<MAX_PROC;i++){
  pid=fork();
  if(pid!=0)
    break;
}
If (i<MAX_PROC) wait();//esperan todos menos el último
exit(0); //todos hacen exit
```

Ejercicio 5: Basándose en el esquema anterior se deberá implementar un programa pr4-5.c que:

- Genere 4 procesos a partir del proceso padre. Nota: implementarlo mediante un bucle que contenga una variable contador i. En ese momento imprimirán un mensaje diciendo "Soy el padre con valor de i: XXX con PID: XXX y he creado a mi hijo con PID: XXX".
- Todos espera 10 segundos antes de hacer la llamada a exit.
- Todos los hijos invocan el exit (excepto quien no ha creado hijo) con valor de retorno igual al valor de la variable i con la que entraron en el bucle.
- Todos los padres esperan a que terminen su hijos y recogen el valor de retorno en la llamada wait. En ese momento imprimirán un mensaje diciendo "Soy el padre con PID: XXX y el valor de retorno de mi hijo es: XXX". Para obtenerlo aplicar la mascara WEXITSTATUS(retorno) donde retorno es la variable usada en la llamada wait().

Anotar en la tabla los resultados:

| | PID | Valor retorno hijo |
|---|---|---|
| Proceso Padre | | |
| Proceso hijo 1 | | |
| Proceso hijo 2 | | |
| Proceso hijo 3 | | |
| Proceso hijo 4 | | |

Cuestión:

¿Por qué los procesos creados no inician el bucle con el valor inicial 0?

# 5 Creación de hilos

El código siguiente constituye el esqueleto básico de una función que utiliza hilos en su implementación.

```
/**
* Programa de ejemplo "Hola mundo" con pthreads.
* Para compilar teclea:  gcc hola.c -lpthread -o hola
**/
#include <stdio.h>
#include <pthread.h>
#include <string.h>

void *Imprime( void *ptr )
 {
   char *men;
   men=(char*)ptr;

   //EJERCICIO1.b
   write(1,men,strlen(men));
 }
 int main()
 {

  pthread_attr_t atrib;
  pthread_t hilo1, hilo2;

  pthread_attr_init( &atrib );

  pthread_create( &hilo1, &atrib, Imprime, "Hola \n");
  pthread_create( &hilo2, &atrib, Imprime, "mundo \n");

  //EJERCICIO1.a
  pthread_join( hilo1, NULL);
  pthread_join( hilo2, NULL);
 }
```

Cree un archivo "hola.c" que contenga dicho código, compílelo y ejecútelo desde la línea de órdenes.

```
$ gcc hola.c –lpthread –o hola
```

Como se observa en el código de la figura-1, las novedades que introduce el manejo de hilos van de la mano de las funciones necesarias para inicializarlos, de las cuales sólo hemos hecho uso de las más básicas o imprescindibles.

- Tipos **`pthread_t`** y **`pthread_attr_t`** que se acceden desde el archivo de cabecera `pthread.h`.

```
#include <pthread.h >
pthread_t  th;
pthread_attr_t attr;
```

- Función **`pthread_attr_init`** encargada de asignar unos valores por defecto a los elementos de la estructura de atributos de un hilo. ¡AVISO! Si no se inicializan los atributos, el hilo no se puede crear

```
#include <pthread.h >

 int pthread_attr_init(pthread_attr_t *attr)
```

- Función **pthread_create** encargada de crear un hilo.

```
#include <pthread.h >

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

***Parámetros de pthread_create:***

***thread:*** Es el primer parámetro de esta función, *thread,* contendrá el identificador del hilo
***attr:*** El argumento *attr* especifica los atributos del hilo. Puede tomar el valor NULL, en cuyo caso recibe también valores por defecto: "*the created thread is joinable (not detached) and has default (non real-time) scheduling policy*".
***start_routine***: El comportamiento del hilo que se está creando viene definido por la función que se le pasa como tercer parámetro *start_routine* y a la que se le pasará como argumento el puntero *arg*.
**Valor de retorno de la función pthread_create():**
Devuelve 0, si la función se ejecuta con éxito. En caso de error, la función devuelve un valor distinto de cero.

- Función ***pthread_join***. Su efecto es suspender al hilo que la invoca hasta que el hilo que se le especifica como parámetro termine. Este comportamiento es necesario ya que cuando el hilo principal "termina" destruye el proceso y, por lo tanto, obliga a la terminación de todos los hilos que se hayan creado.

**Parámetros de pthread_join:**

**thread**: Parámetro que identifica al hilo a esperar.
**exit_status**: contiene el valor que el hilo terminado comunica al hilo que invoca a pthread_join

```
#include <pthread.h >

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

- Función ***pthread_exit*** Permite a un hilo terminar voluntariamente su ejecución. La finalización del último hilo de un proceso finaliza el proceso. Mediante el parámetro `exit_status` puede comunicar un valor de terminación a otro hilo que estuviera esperando su finalización.

```
#include <pthread.h >

 int pthread_exit(void *exit_status);
```

**Ejercicio 5: trabajando con pthread_join y pthread_exit**
Compruebe el comportamiento de la llamada pthread_join() realizando las siguientes modificaciones en el código del programa "hola.c" mostrado anteriormente.

**Cuestiones Ejercicio 5:**

Elimine (o comente) las llamadas pthread_join del hilo principal.

- ¿Qué ocurre?¿Por qué?

Sustituya las llamadas pthread_join por una llamada pthread_exit(0), cerca del punto del programa marcado como //EJERCICIO1.a)

- ¿Completa ahora correctamente el programa su ejecución?¿Por qué?

Elimine (o comente) cualquier llamada a pthread_join o pthread_exit (cerca del comentario //EJERCICIO1.a) e introduzca en ese mismo punto un retraso de 1 segundo (usando usleep(…) )

```
#include <unistd.h>
  void usleep(unsigned long usec);  // usec en microsegundos
```

- ¿Qué ocurre tras la realización de las modificaciones propuestas?

Introduzca ahora un retraso de 2 segundos cerca del comentario `//EJERCICIO1.b`

- ¿Qué ocurre ahora?¿Por qué?

# 6 Trabajando con hilos periódicos

En muchas ocasiones los hilos se utilizan para llevar a cabo tareas que se han de repetir periódicamente, es decir cada cierto intervalo de tiempo. Una manera sencilla de conseguirlo es que el código del hilo incorpore una llamada de espera `sleep()` dentro de un bucle. Siguiendo esta idea, se propone completar un programa que realice una animación que recuerde al conocido efecto "lluvia digital" donde una serie de caracteres aleatorios van apareciendo periódicamente formando columnas descendentes.

El código de partida "matrix_base.c" (aparece a continuación), se basa en la creación de un hilo "Dibujador" para cada una de las columnas de la pantalla. Cada uno de estos hilos dibujadores va completando la columna que tiene asignada escribiendo sus caracteres en una matriz bidimensional `m` compartida. Al mismo tiempo, otro hilo "RefrescadorDePantalla" (único) va trasladando periódicamente el contenido completo de esta matriz a la pantalla para visualizar la animación.

**Ejercicio 6: Animación mediante hilos.**

Elabore un programa "matrix_dibujar.c" a partir del programa "matrix_base.c", añadiendo a su función main las llamadas necesarias para crear un hilo dibujador por cada una de las columnas. Haga uso de la constante `COLUMNAS` y de las variables globales que tiene ya definidas para albergar los identificadores de los hilos. Emplee la función `Dibujador` como cuerpo de sus hilos. Esta función debe recibir como argumento (por valor) el número de columna asignado al hilo (de 0 a `COLUMNAS`-1). Cree también un hilo refrescador (la función `RefrescadorDePantalla` no utiliza el valor que recibe como argumento). Asegúrese además de esperar la finalización de todos los hilos dibujadores, de forma que el programa termine cuando se haya completado el dibujo de todas las columnas (observará que algunas de ellas no se dibujan, para simular el efecto original).

**Cuestión Ejercicio 6:**

- ¿Debe esperar también la finalización del hilo refrescador de pantalla?

```
/** programa matrix_base.c **/
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

#define COLUMNAS 80
#define FILAS 25

char m[FILAS][COLUMNAS];
long retardo[COLUMNAS];
int fila_b[COLUMNAS];

pthread_attr_t atrib;
pthread_t  hilo_dibujador[COLUMNAS];
pthread_t  hilo_borrador[COLUMNAS];
pthread_t  hilo_refrescador;

void *Borrador(void *ptr)
{
  int fila, col=(int)(long)ptr;
  for(fila=0;fila<FILAS;fila++){
    m[fila][col]= ' '; //Escribir un espacio
    usleep(retardo[col]); //Esperar antes del siguiente borrado
  }
}


void *Dibujador(void *ptr)
{
  int fila, col=(int)(long)ptr;
  retardo[col]= 50000+rand()%450000;//Retardo aleat. de carácter, de 0,05s a 0,5s
  if (rand()%10>4) //A veces, no dibujar la columna
    usleep(retardo[col]*FILAS); //Esperar sin dibujar
  else{
    for(fila=0;fila<FILAS;fila++){
      m[fila_b[col]=fila][col]= 32+rand()%94; //Escribir nuevo carácter aleatorio
      usleep(retardo[col]); //Esperar antes del siguiente carácter
    }
  }
}

void *RefrescadorDePantalla(void *ptr)
{
  int fila, col;
  char orden[20];
  while(1){
    write(1,"\033[1;1f\033[1;40;32m",16); //Volver a esquina superior izquierda,
texto verde
```

```
    for(fila=0;fila<FILAS;fila++){
      write(1,m[fila],COLUMNAS); write(1,"\n",1); //Refrescar fila
    }
    write(1,"\033[1;37m",7); //Texto blanco
    for(col=0;col<COLUMNAS;col++){
      sprintf(orden,"\033[%d;%df%c",fila_b[col]+1,col+1,m[fila_b[col]][col]);
      if(fila_b[col]<FILAS-1) write(1,orden,strlen(orden)); //Reescribir en
blanco el último carácter de la columna
    }
    usleep(100000); //Esperar 0,1s antes de volver a refrescar
  }
}

int main()
{
  int col;
  memset (m,' ',FILAS*COLUMNAS); //Borrar la matriz m
  write(1,"\033[2J\033[?25l",10); //Borrar pantalla, ocultar cursor

  pthread_attr_init(&atrib);

  //Crear un hilo dibujador para cada columna

  //Crear un hilo refrescador de pantalla

  //Esperar la finalización de los hilos dibujadores

  write(1,"\033[0m\033[?25h\r",11); //Restaurar texto normal, restaurar cursor
}
```

# 7 Acceso de hilos a datos comunes

En este apartado vamos a avanzar en el desarrollo de el trabajo de la asignatura que se basa en la implementación de un modelo productor/consumidor. El paradigma de productor/consumidor es un ejemplo general de todas aquellas relaciones entre actividades concurrentes (o incluso paralelas que no son el caso de esta asignatura) en las cuales una de ellas genera datos (productor), mientras que otra los consume. Podemos imaginar múltiples situaciones en las que este modelo se aplica en sistemas informáticos:

- Un tarjeta de adquisición de datos que convierte una señal analógica para que un proceso de control la use en el bucle
- Un bucle de control que genera una acción que debe ejecutar un actuador
- Una interfaz de Ethernet que toma paquetes de la red y los deja en una zona de memoria para ser usados por el protocolo de nivel superior

Todos estos casos (y muchos más) son en esencia un modelo productor/consumidor. Las principales características de los que vamos a implementar en el trabajo son:

- Intercomunican dos o varios threads
- Comparten una zona de memoria (variable de tipo buffer circular) común
- A través de ella algunos Threads generan datos (productores) y otros los recogen (consumidores)

Por tanto para avanzar en el trabajo en esta practica se deberá:

- Desarrollar una aplicación que lance 2 threads, y se quede a la espera de que ambos finalicen su ejecución. Cuando terminen este proceso lanzará el mensaje “FIN APLICACIÓN”
- A ambos threads se les pasa como parámetro en la llamada de la creación el puntero a una estructura de tipo tampón circular con espacio para 10 datos

- Uno de los threads actuará como productor insertando (mediante la llamada correspondiente a la función implementada en la practica anterior) un dato cada 2 segundos, mostrando por consola el dato insertado. Realizará esta acción durante 30 iteraciones, al cabo de las cuales terminará.
- Uno de los threads actuará como consumidor recogiendo (mediante la llamada correspondiente a la función implementada en la practica anterior) un dato cada 4 segundos, mostrando por consola el dato extraido. Realizará esta acción durante 30 iteraciones, al cabo de las cuales terminará.