Laboratorio de Sistemas Operativos

Juan Antonio Álvarez, Toñi Reina, David Ruiz,
Antonio Tallón, Javier Gutiérrez, Pablo Neira,
Paco Silveira, Jose Angel Bernal y Sergio Segura

Boletín ♯4: Procesos

Curso 2008/09

Índice

1. Introducción **2**

1.1. El comando ps 2

2. Identificadores de usuarios y procesos **2**

3. La llamada fork **3**

3.1. Herencia de descriptores 4

4. Las llamadas wait y exit **5**

5. La llamada exec **6**

6. La terminación de procesos **8**

7. Comunicación entre procesos con tuberías **8**

8. Ejercicios **9**

A. Ejercicios de Examen **11**

---

1. Introducción

En Unix los procesos se identifican con un número entero denominado *ID de proceso* o *PID*. Al proceso que ejecuta la solicitud para la creación de un procesos se le suele llamar procesos *padre*, y al proceso creado se le suele llamar proceso *hijo*.

1.1. El comando ps

El comando ps sirve para mostrar información sobre procesos, sus sintaxis (POSIX) es la siguiente:

```
ps [-aA] [-G grouplist] [-o format] ... [-p proclist]
   [-t termlist] [-U userlist]
```

Muchas de las implementaciones que hacen los distintos vendedores del comando ps no cumplen con el estándar POSIX, un ejemplo claro es Sun Solaris (SO del servidor de prácticas de la escuela) que emplea la opción -e en lugar de -A para mostrar la información de todos los procesos. La versión larga del comando ps de Solaris muestra mucha información interesante sobre los procesos asociados a un terminal (la figura 1 muestra algunos).

Campo	Significado
UID	ID del usuario propietario del proceso
PID	ID del proceso
PPID	ID del padre del proceso
C	Utilización del preprocesador de C para la administración de procesos
STIME	Hora de comienzo
TTY	terminal de control
TIME	Tiempo acumulado de CPU
CMD	Nombre del comando

Figura 1: Algunos campos del comando ps

Ejemplo 1 *El siguiente comando muestra, en formato largo, todos los procesos cuyo usuario propietario es i5599:*

```
murillo:/export/home/cursos/so> ps -fu i5590
   UID   PID  PPID  C    STIME TTY      TIME CMD
 i5590 12246 12195  0 20:53:16 pts/15   0:02 clips
 i5590 12164 12150  0 20:30:56 pts/15   0:00 -ksh
 i5590 12194 12164  0 20:35:23 pts/15   0:00 -ksh
 i5590 12175 12152  0 20:31:12 pts/14   0:00 -ksh
 i5590 12195 12194  0 20:35:23 pts/15   0:00 /export/home/cursos/CCIA/bin/tcsh
 i5590 12176 12175  0 20:31:12 pts/14   0:00 /export/home/cursos/CCIA/bin/tcsh
 i5590 12205 12150  0 20:37:29 pts/16   0:00 -ksh
 i5590 12152 12150  0 20:30:39 pts/14   0:00 -ksh
 i5590 12192 12176  0 20:34:47 pts/14   0:01 /opt/sfw/bin/emacs oo.clp
```

Este comando es prácticamente equivalente a ps –ef | grep i5590.

2. Identificadores de usuarios y procesos

Las funciones C que se utilizan para obtener el identificador de proceso (PID) o el identificador de usuario (UID) son getpid, getppid y getuid:

```
#include <sys/types.h>
#include <unistd.h>

 pid_t getpid(void);
 pid_t getppid(void);
 uid_t getuid(void);
```

- pid_t es un entero largo con el ID del proceso llamante (getpid) o del padre del proceso llamante (getppid)
- uid_t es un entero con el ID del usuario propietario del proceso llamante.
- En caso se error se devuelve -1.

Ejemplo 2 *El siguiente programa imprime el identificadores del proceso llamante, del proceso padre y del propietario:*

```
 #include <stdio.h>
 #include <sys/types.h>
 #include <unistd.h>

 int main(void)
 {
    printf("ID de proceso: %ld\n", (long)getpid());
    printf("ID de proceso padre: %ld\n", (long)getppid());
    printf("ID de usuario propietario: %ld\n", (long)getuid());

 return 0;
 }
```

3. La llamada fork

En Unix los procesos se crean a través haciendo una llamada fork al sistema. El nuevo proceso que se crea recibe una copia del espacio de direcciones del padre. Los dos procesos continúan su ejecución en la instrucción siguiente al fork:

```
#include <sys/types.h>
#include <unistd.h>

 pid_t fork(void);
```

- La creación de dos procesos totalmente idénticos no suele ser útil. Así, el valor devuelto por fork permite diferenciar el proceso padre del hijo, ya que *fork devuelve 0 al hijo y el ID del hijo al padre*.
- En caso de error devuelve -1.

La llamada fork crea procesos nuevos haciendo una copia de la imagen en la memoria del padre. El hijo *hereda*[1] la mayor parte de los atributos del padre, incluyendo el entorno y los privilegios. El hijo también hereda alguno de los recursos del padre, tales como los archivos y dispositivos abiertos. Las implicaciones de la herencia son mucho más complicadas de lo que en principio pueda parecer.

Ejemplo 3 *El siguiente código produce un árbol de procesos como el de la figura 2:*

[1] No confundir este concepto con el de herencia en el contexto de la orientación a objetos.

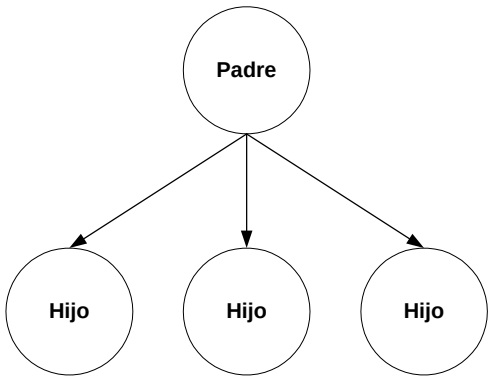

Figura 2: Árbol de procesos

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main (void)
{
  int i;
  int padre;
  padre = 1;
  for (i=0; i < 3; i++)
  {
    if (padre == 1)
    {
      if (fork()  == 0) /* Proceso hijo */
      {
         fprintf(stdout, "Éste es el proceso hijo con padre %ld\n",
                        (long)getppid());
         padre = 0;
      }
      else  /* Proceso padre */
      {
        fprintf(stdout, "Éste es el proceso padre con ID %ld\n",
                        (long)getpid());
        padre = 1;
    }
  }

return 0;
}
```

3.1. Herencia de descriptores

Cuando *fork* crea un proceso hijo, éste hereda la mayor parte del entorno y contexto del padre, que incluye el estado de las señales, los parámetros de la planificación de procesos y la tabla de descriptores de archivo.

Hay que tener cuidado ya que las implicaciones de la herencia de los descriptores de archivos no siempre resultan obvias, ya que el proceso padre e hijo comparten el mismo desplazamiento de archivo para los archivos que fueron abiertos por el padre antes del *fork*.

4. Las llamadas wait y exit

Si queremos que el proceso padre espere hasta que la ejecución del hijo finalice tenemos que utilizar la llamada a wait o a waitpid junto con la llamada exit. La declaración de exit es la siguiente:

```
#include <stdlib.h>

void exit (int status);
```

- exit termina la ejecución de un proceso y le devuelve el valor de status al sistema. Este valor se puede ver mediante la variable de entorno ?. El return efectuado desde la función principal (main) de un programa C tiene el mismo efecto que la llamada a exit.

  Además de esto, la llamada a exit tiene también las siguientes consecuencias:

  - Si el proceso padre del que ejecuta la llamada a exit está ejecutando una llamada a wait, se le notifica la terminación de su proceso hijo y se le envían los 8 bits menos significativos de status. Con esta información el proceso padre puede saber en qué condiciones ha terminado el proceso hijo.
  - Si el proceso padre no está ejecutando una llamada a wait, el proceso hijo se transforma en un proceso zombi.

Las declaraciones de wait y waitpid son las siguientes:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- wait detiene al proceso llamante hasta que un hijo de éste termine o se detenga, o hasta que el proceso llamante reciba una señal.
- En el caso de que cuando se solicita la espera el hijo ya ha terminado (o simplemente no existe ningún proceso hijo) la llamada wait devuelve el control de inmediato.
- Cuando un proceso hijo termina, la llamada wait devuelve el ID de dicho hijo al padre. En caso contrario devuelve -1.
- stat_loc es un puntero a entero que en caso de valer distinto de NULL la llamada wait devuelve el estado devuelto por el hijo (exit o return). En el estándar POSIX se definen las siguiente macros para analizar el estado devuelto por el proceso hijo:
  - WIFEXITED, terminación normal.
  - WIFSIGNALED, terminación por señal no atrapada.
  - WTERMSIG, terminación por señal SIGTERM.
  - WIFSTOPPED, el proceso está parado.
  - WSTOPSIG, terminación por señal SIGSTOP (que no se puede atrapar ni ignorar).

Ejemplo 4 *En el siguiente programa el proceso padre espera a que termine la ejecución del hijo imprimiendo el código de terminación del mismo. El proceso hijo estará inactivo durante dos segundos y devolverá 0 si el ID del padre es mayor que su ID, en caso contrario, devuelve distinto de 0:*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

int main (void) {
```

```
  pid_t childpid;
  int   status=0;
  int   result;

  if ((childpid = fork()) == -1)
    {
      perror("Error en llamada a fork\n");
      exit(1);
    }
  else if (childpid == 0)
    {
      result = getpid() < getppid() ;
      fprintf(stdout, "Soy el proceso hijo (%ld) y
                       voy a devolver a mi padre (%ld)
                       el valor %d después
                       de esperar 2 segundos\n",
       (long)getpid(),
       (long)getppid(),
       result);
      sleep(2);
      exit (result);
    }
 else
   {
     while( childpid != wait(&status))
       ;
     fprintf(stdout, "Soy el proceso padre (%ld)
                      y mi hijo (%ld) me ha devuelto %d\n",
      (long)getpid(),
      (long)childpid,
      status);
   }

  return (0);
}
```

5. La llamada exec

La llamada fork crea una copia del proceso llamante. Muchas veces es necesario que el proceso hijo ejecute un código totalmente distinto al del padre. En este caso, la familia de llamadas exec nos permitirá sustituir el proceso llamante por uno nuevo[2].

```
 #include <unistd.h>
  int execl  (const char *path, const  char  *arg0,  ...,
              const char *argn, char * /*NULL*/);
  int execv  (const char *path, char *const argv[]);
  int execle (const char *path, const char  *arg0,  ...,
              const char *argn, char * /*NULL*/, char *const envp[]);
  int execve (const char *path, char *const argv[], char *const envp[]);
  int execlp (const char *file, const char  *arg0,  ...,
              const char *argn, char * /*NULL*/);
  int execvp (const char *file, char *const argv[]);
```

- Las seis variaciones de la llamada exec se distinguen por la forma en que son pasados los argumentos de la

[2]Normalmente se suele utilizar la combinación fork–exec para dejar que sea el hijo el que sea sustituido mientras que el padre ejecuta el código original.

línea de comando y el entorno, y por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable.

- Las llamadas execl (execl, execle y execlp) pasan la lista de argumentos de la línea de comando como una lista y son útiles sólo si se conoce a priori el número de éstos.
- Las llamadas execv (execvl, execvp y execve) pasan la lista de argumentos en una cadena (un *array* de punteros a *char*) y son útiles cuando no se sabe el número de argumentos en tiempo de compilación.
- path es la ruta (completa o relativa al directorio de trabajo) de acceso al archivo ejecutable.
- arg_i es el argumento i–ésimo (sólo en llamadas execl).
- argv es una cadena con todos los argumentos (sólo en llamadas execv).
- envp es una cadena con el entorno que se le quiere pasar al nuevo proceso.

Ejemplo 5 *El siguiente programa utiliza la función* execl *para listar los procesos activos en el momento de la ejecución. Ejecútelo y vea los errores del mismo.*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main() {
   int status;
   printf ("Lista de procesos\n");
   if (execl ("ps", "ps", "-f", 0) < 0)
        {
          fprintf(stderr, "Error en exec %d\n", errno);
          exit(1);
        }
   printf ("Fin de la lista de procesos\n");
   exit(0);
}
```

Ejemplo 6 *El siguiente programa utiliza un esquema* fork–exec *para listar los procesos del usuario propietario del proceso.*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

int main()
{
   pid_t childpid, waitreturn;
   int status;

   if ((childpid = fork()) == -1)
     {
       fprintf(stderr, "Error en fork %d\n", errno);
       exit(1);
     }
   else if (childpid == 0)
     {          /* código del proceso hijo */
```

```
      if ( execl ("/bin/ps", "ps", "-fu", getenv ("USER"), 0) < 0)
       {
         fprintf(stderr, "Error en exec %d\n", errno);
         exit(1);
       }
    }
  else                              /* código del proceso padre */
    while(childpid != (waitreturn = wait(&status)))
      if ((waitreturn == -1) && (errno != EINTR))
            break;

  exit(0);
}
```

6. La terminación de procesos

Cuando termina un proceso (normal o anormalmente), el SO recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la terminación.

Las actividades realizadas durante la terminación de un proceso incluyen la cancelación de temporizadores y señales pendientes, la liberación de los recursos de memoria virtual así como la de otros recursos del sistema ocupados por el proceso.

Cuando un proceso termina, sus hijos *huérfanos* son adoptados por el proceso init, cuyo ID el 1. Si un proceso padre no espera a que sus hijos terminen la ejecución, entonces éstos se convierten en procesos *zombies* y tiene que ser el proceso init el que los libere (lo hace de manera periódica).

Las llamadas a exit y _exit[3] se utilizan para terminar de forma normal un proceso. La función exit lo que hace es llamar a los manejadores de terminación del usuario (si existen) y después llamar a _exit.

7. Comunicación entre procesos con tuberías

Para comunicar distintos procesos haciendo uso de tuberías en C se utiliza la función *pipe*:

```
#include <unistd.h> int pipe (int fildes[2]);
```

Devuelve 0 en caso de éxito y -1 en caso de error, almacenando en errno el código del mismo.

fildes es una tabla con dos descriptores de ficheros: fildes[0] (descriptor de lectura) y fildes[1] (descriptor de escritura).

Ejemplo 7 *El siguiente programa ejecuta el comando ls -l — sort -n +4 comunicando a los procesos* ls *y* sort *con tuberías:*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

int main(void)
{
   int fd[2];
   pid_t childpid;

   pipe(fd);
```

[3] Fíjese en que return no es una llamada al sistema sino que se trata de una instrucción del lenguaje C.

```
    if ((childpid = fork()) == 0) {  /* ls es el hijo */
       dup2(fd[1], STDOUT_FILENO);
       close(fd[0]);
       close(fd[1]);
       execl("/usr/bin/ls", "ls", "-l", NULL);
       perror("Exec falló en ls");
    } else {                        /* sort es el padre */
       dup2(fd[0], STDIN_FILENO);
       close(fd[0]);
       close(fd[1]);
       execl("/usr/bin/sort", "sort", "-n", "+4", NULL);
       perror("Exec falló en sort");
    }
    exit(0);
 }
```

8. Ejercicios

1. Realice un programa que cree cuatro procesos, A, B, C Y D, de forma que A sea padre de B, B sea padre de C, y C sea padre de D.

2. Realice un programa *copiaConc* al que se le pase una lista de nombres de archivos y para cada archivo *f* cree un nuevo proceso que se encargue de copiar dicho archivo a *f.bak*. Reutilice el código del ejercicio 1 del boletín 3.

3. Realice un programa *ejecuta* que lea de la entrada estándar el nombre de un programa y cree un proceso hijo para ejecutar dicho programa.

4. Realice un programa *aviso* que reciba como argumentos un entero *n* y el nombre de un archivo *f*, de forma que cada *n* segundo compruebe si *f* ha sido modificado[4].

5. Modifique el programa anterior (llámelo *ejeTemp*) para que ejecute cada *n* segundos el programa apuntado por *f*.

6. Realice un programa que cree dos procesos, de forma que en el proceso padre se le pida al usuario un mensaje por la entrada estándar. Cuando el usuario escriba dicho mensaje, se enviará mediante una tubería al proceso hijo, el cual se encargará de imprimirlo por la salida estándar.

   El proceso terminará cuando el usuario introduzca un terminador de mensaje, como, por ejemplo, la cadena ”FIN”.

7. Escriba un programa *tuberia* que reciba como argumento una lista de programas (de sistema o aplicación) y los ejecute comunicándolos a través de tuberías.

```
  murillo:/so> tubería "ls -l /tmp" "grep alumnos"  "more"
 drwx------   2 i6458    alumnos      117 Sep 23 20:06 3ca4ew
 drwx------   2 s0624    alumnos      117 Sep 25 12:40 HSaaHo
 -rwxr--r--   1 rodrigud alumnos       67 Sep 13 11:21 compila
 -r-xr-xr-x   1 temp2    alumnos   226520 Sep 30 12:19 f9
 -rw-r--r--   1 temp3    alumnos       45 Sep 23 21:07 fich_temp-15934
 drwx------   2 i4746    alumnos      117 Sep 19 10:06 g9aiEF
 ...
 -rw-------   1 s2879    alumnos        0 Sep 23 17:22 mp6gaGvj
 --More--
```

   NOTA:

   - Para ejecutar los comandos utilice la llamada del sistema `system()`.

[4]Este programa puede ser lanzado en segundo plano para saber cuándo un usuario Unix recibe correo (fichero */var/mail/usuario*)

8. Modifique el programa anterior para que en caso de recibir un único argumento por la línea de comandos, éste se interprete como el nombre de un archivo que contiene en cada línea un comando.

A. Ejercicios de Examen

1. ***(Ejercicio de Examen 1ª CONV ITI 2002-03)***

   Escriba un programa en C de nombre `redireccion` que implemente la redirección de la entrada estándar o de la salida de un comando a/o desde un fichero. Para almacenar la salida del comando en un archivo se utilizará la opción `-s`, mientras que para utilizar el contenido del archivo como entrada al comando se utilizará la opción `-e`. Estas dos opciones nunca se utilizarán a la vez. La forma de invocar al programa será la siguiente:

   ```
   redireccion {-e | -s} <fichero> <comando>
   ```

   Por lo tanto, la invocación:

   ```
   murillo:/tmp> redireccion -s fichero.txt  ls -la
   ```

   será equivalente a ejecutar la siguiente orden del intérprete de comandos:

   ```
   murillo:/tmp> ls -la > fichero.txt
   ```

   Y de la misma forma, la invocación:

   ```
   murillo:/tmp> redireccion -e fichero.txt cat -n
   ```

   es equivalente a ejecutar la siguiente orden del intérprete de comandos:

   ```
   murillo:/tmp> cat -n < fichero.txt
   ```

   NOTAS:

   - No se permite la utilización de la llamada al sistema `system()` para resolver el ejercicio.
   - No se pueden dejar procesos zombies.
   - En caso de que se vaya a almacenar la salida del comando en un fichero que ya exista, se sobrescribirá el contenido del mismo con la salida del comando.

2. ***(Ejercicio de Examen 2ª CONV ITI 2002-03)***

   Escriba un programa en C que cree tres nuevos procesos y los comunique por tuberías para ejecutar la siguiente línea de comandos:

   ```
   $ who -u | grep old | sort -r
   ```

   NOTAS:

   - No se permite la utilización de la llamada al sistema `system()` para resolver el ejercicio.
   - No se pueden dejar procesos huérfanos.

3. ***(Ejercicio de Examen 3ª CONV II 2002-03)***

   Escriba un programa anillo que cree un anillo de tres procesos conectados por tuberías. El primer proceso deberá pedirle al usuario un mensaje por la entrada estándar y enviárselo al segundo proceso. El segundo proceso deberá invertir la cadena y enviarla al tercer proceso. El tercer proceso deberá convertir la cadena a mayúsculas y enviársela de vuelta al primer proceso. Cuando el primer proceso obtenga la cadena procesada, deberá mostrarla por la salida estándar. Cuando esta cadena se termine, los tres procesos deberán terminar. La Figura 3 muestra un esquema del anillo de procesos y un ejemplo de una posible ejecución del programa.

   NOTAS:

   - No se pueden dejar procesos huérfanos.
   - Como ayuda se suponen definidas las dos siguientes funciones:

```
$ anillo
Introduzca una cadena: hola Mariola
La cadena procesada es: ALOIRAM ALOH
$ _
```

Figura 1. Ejemplo de ejecución del programa `anillo`.

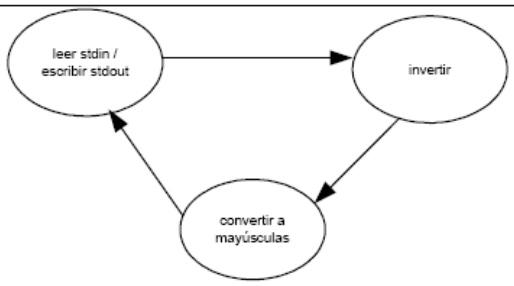

Figura 2. Esquema de procesos del programa `anillo`.

Figura 3: Árbol de procesos

- `void invertir (char cadena[]);`
  La función `invertir` tiene un parámetro (cadena) de entrada/salida que será una cadena bien formada. Este argumento deberá contener la cadena a invertir, y una vez que invertir haya realizado su tarea, contendrá la cadena invertida.
- `void amayuscula (char cadena[]);`
  La función `amayuscula` tiene un argumento de entrada/salida que también deberá ser una cadena bien formada. Este argumento deberá contener la cadena que se quiere convertir a mayúsculas, y una vez que `amayuscula` haya realizado su tarea, contendrá la cadena en mayúsculas.

4. ***(Ejercicio de Examen 1ª CONV ITI 2003-04)***

   Implemente en C un programa llamado `ejecutadir` que reciba como argumento un directorio y ejecute sin pasarles ningún argumento todos los archivos regulares y con permiso de ejecución que se encuentren en el mismo. Antes de comenzar a ejecutar un archivo aparecerá un mensaje indicativo de qué programa se está ejecutando. Además, hasta que no se termine de ejecutar un archivo, no se debe comenzar a ejecutar el siguiente. Por último, el resultado de la ejecución de cada archivo deberá aparecer en un archivo con el mismo nombre del programa que se está ejecutando, y con extensión ".salida".

   Un ejemplo de ejecución del programa sería:

```
$ejecutadir $HOME/practicasSO
Ejecutando. . . . . /home/practica/practicasSO/practITIS-01
Ejecutando. . . . . /home/practica/practicasSO/practITIG-01
Ejecutando. . . . . /home/practica/practicasSO/practITIS-02
Ejecutando. . . . . /home/practica/practicasSO/practITIG-02
```

   Además de la información que se muestra por la salida estándar, una vez que `ejecutadir` termine, en el directorio `$HOME/practicasSO`, habrá como resultado cuatro nuevos archivos, llamados:

```
practITIS- 01.salida, practITIG-01.salida, practITIS-02.salida y
practITIG-02.salida.
```

   NOTAS:

   - No se puede usar la llamada al sistema system ( ).
   - No se pueden quedar procesos huérfanos.
   - La máscara que comprueba si el propietario de un fichero tiene permiso de ejecución sobre el mismo es: `S_IEXEC`

5. ***(Ejercicio de Examen 4ª CONV ITI 2002-03)***

   Realizar un programa el C que cree tres nuevos procesos y los comunique mediante tuberías para ejecutar la siguiente línea de comandos:

```
$ ls -al | grep  ^d  | sort -r > salida.txt
```

Nótese que la salida no será mostrada por pantalla, sino almacenada en un fichero llamado salida.txt .

NOTAS:

- No se permite la utilización de la llamada al sistema system() para resolver el ejercicio.
- No se pueden dejar procesos huérfanos.

6. ***(Ejercicio de Examen 4ª CONV II 2003-04)***

Queremos implementar en C un programa llamado patronDirectorio. La forma de invocar dicho programa será:

```
$ patronDirectorio <patron> <directorio>
```

Dónde `<patron>` es una cadena que representa una expresión regular con el mismo formato que las utilizadas en el comando grep, y `<directorio>` es un directorio. El programa mostrará por la salida estándar el número de líneas que hay en los ficheros regulares que contienen el patrón especificado por `<patron>` situados en `<directorio>`.

Una posible ejecución de dicho programa sería este:

```
$ patronDirectorio '^#' .

Fichero: ./prueba.o, numero de lineas con el patron '^#':        0
Fichero: ./prueba.c~, numero de lineas con el patron '^#':       6
Fichero: ./prueba.c, numero de lineas con el patron '^#':        6
Fichero: ./prueba, numero de lineas con el patron '^#':          0
Fichero: ./prueba2.c, numero de lineas con el patron '^#':       7
Fichero: ./#prueba#, numero de lineas con el patron '^#':        0
Fichero: ./prueba2, numero de lineas con el patron '^#':         0
Fichero: ./prueba2.c~, numero de lineas con el patron '^#':      6
```

NOTAS:

- No se permite la utilización de la llamada al sistema system().
- No se pueden dejar procesos huérfanos.
- Para contar las líneas de un fichero recuerde el comando wc.