Escuela Politécnica Superior de Elche

Grado en Ingeniería Informática en Tecnologías de la Información

Sistemas Operativos
Cuaderno de practicas C - Procesos

Procesos
Llamadas al Sistema con ejemplos prácticos.

Objetivos 2
Requisitos 2
El entorno de desarrollo 2
Llamadas al sistema 3
Resultado de las llamadas al sistema 4
La variable global errno 4
Variables de entorno 8
Procesos 11
La orden ps 12
La orden top 12
Llamadas al sistema relacionadas con procesos 13
POSIX 13
Credenciales de usuario 13
fork 15
wait y waitpid 17
exit 20
exec 22
Ejemplos 26

Objetivos

El objetivo de este cuaderno es presentar de una manera resumida al alumno los contenidos teóricos y prácticos necesarios para la parte práctica de C de la asignatura.

Requisitos

Conocer la programación en C básica.

El entorno de desarrollo

Utilizaremos el compilador GNU C Compiler. http://gcc.gnu.org/

Un tutorial en castellano muy sencillo se puede encontrar en http://iie.fing.edu.uy/~vagonbar/gcc-make/gcc.htm

Brevemente describiremos los pasos básicos para poder compilar los ejercicios y ejemplos.

El comando gcc es el interfaz de usuario (front-end) del compilador de C GNU.

El siguiente comando compila un programa llamado "**uno.c**". El fichero ejecutable será "**uno**"

```
$ gcc uno.c -o uno
```

La opción -o especifica el nombre del fichero ejecutable. Si no se especifica, el nombre del fichero ejecutable generado es "**a.out**".

Si tenemos varios ficheros fuente que forman parte de un único programa ejecutable podemos compilarlos de uno en uno de forma separada (opción -c del compilador) y linkarlos todos al final con el comando "**ld**" (linkador).

Otra opción es hacerlo todo de golpe especificando la lista de ficheros en el compilador. Por ejemplo, si tenemos tres ficheros llamados: **uno.c**, **dos.c** y **tres.c**, y queremos obtener un ejecutable denominado "**prog**", ejecutaríamos el siguiente comando:

```
$ gcc uno.c dos.c tres.c -o prog
```

Ejemplo c001.c:
Programa Hola Mundo

```
// ==========================
// Programa #1: Hola Mundo !!
// Archivo: holamundo.c
// ==========================
#include <stdio.h>
int main(void) {
printf("Hola mundo\n");
return 0
}
```

Lo compilaríamos con:

```
$ gcc holamundo.c -o holamundo
```

Ejemplo c002.c:
Muestra cómo se gestionan los parámetros pasados al programa desde la línea de comandos.

```
// ====================================================
// Programa #2: Paso de parámetros desde la línea de órdenes
// Archivo: ejemplo2.c
// ====================================================
#include <stdio.h>
int main (int argn,char *argv[])
{
int i;
printf("Número de argumentos: %d\n",argn);
printf("Nombre del programa : %s\n",argv[0]);
// Mostramos el resto de parámetros
for(i=1;i<arg;++i) {
   printf("Argumento %d = %s\n",i,argv[i] );
   }
return 0;
}
```

Llamadas al sistema

Un programa de usuario nunca podrá tomar el control del sistema en modo supervisor; de no ser así, la ejecución de un programa podría afectar a otros. ¿Cómo puede un proceso realizar entonces una operación E/S?. Con las "llamadas al sistema".

Una "llamada al sistema" es el método que utiliza un proceso para solicitar al sistema operativo que realice una acción. Los procesos realizan llamadas a las funciones de la librería, las cuales generan una interrupción software.

Una "llamada al sistema" es el equivalente software de las interrupciones hardware. Lo que sucede al realizar una llamada al sistema (petición de servicio) es:

1. Se genera una interrupción, para avisar al sistema operativo
2. El control pasa entonces a una rutina asociada del sistema operativo y se pasa a modo supervisor (se cambia el bit de modo).
3. El núcleo del sistema operativo comprueba que los parámetros son correctos.
4. Si es así, ejecuta la rutina y cuando termine se pasa de nuevo a modo usuario y se retorna el control a la siguiente instrucción del programa de usuario detrás de la llamada al sistema.
5. Si son incorrectos, el sistema toma las acciones oportunas.

Existe una librería de funciones (API) para poder realizar las llamadas desde C.

Estas forman el interfaz **POSIX**. Tendremos que incluir el fichero "**unistd.h**", donde están definidos sus prototipos. Puesto que es una librería del sistema, utilizaremos los símbolos "<", ">".

```
#include <unistd.h>
```

Normalmente, cuando se produce algún error, las rutinas del API retornan el valor -1.

En total disponemos de unas 41 llamadas al sistema, que describimos a continuación agrupadas por sus funciones.

Las funciones del sistema trabajan sobre las abstracciones de proceso, fichero y tiempo.

Proceso:

1. Gestión de procesos.
2. Señales.

Fichero:

3. Gestión de ficheros
4. Gestión de directorio y el sistema de ficheros

Tiempo:

5. Gestión de tiempo

Resultado de las llamadas al sistema

Las llamadas al sistema, normalmente retornan -1 en caso de fallo.

Es muy importante realizar siempre esa comprobación para conseguir programas robustos.

Ejemplo c003.c:
de tratamiento del posible error en la ejecución de la llamada a fork().

¿Por qué sale dos veces el mensaje?

```
// ==========================================
// Programa #3: Llamada al sistema con error
// Archivo: ejemplo3.
// ==========================================
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argn, char *argv[]) {
if ( fork() == -1 ) {
   printf ("La llamada a fork ha fallado ...\n");
   exit (1); // o return 1;
   }
else {
   printf ("La llamada a fork se ha realizado con exito...\n");
   exit (0); // o return 0;
   }
}
```

En caso de error en una llamada al sistema, se actualiza automáticamente una variable global llamada "**errno**" que nos indica un código con el motivo de dicho error. Para más información sobre esta variable consultar el siguiente punto: "La variable global: errno".

La variable global errno

Se define en el fichero **<sys/errno.h>**, el cual deberemos incluir si queremos acceder a ella.

```
#include <errno.h>
```

Aunque no es necesario conocer como se define, el sistema lo hace así:

```
extern int * __error();

#define errno (* __error())
```

La función `__error()` retorna un puntero a un campo de la estructura que define los hilos (threads) hijos del hilo inicial. Para el hilo inicial y procesos que no tienen hilos (procesos pesados), retorna un puntero a la variable **errno** global. Observando el #define anterior llegamos a la conclusión de que cuando nosotros escribimos "**errno**" en nuestro código, realmente estamos obteniendo un puntero a un entero, pero esto será transparente para nosotros.

Cuando una llamada al sistema genera un error, retornará un valor negativo (normalmente -1) y establece la variable errno con un valor numérico que indica el motivo.

Este valor permanecerá hasta que otra llamada al sistema provoque otro error, es decir, las llamadas al sistema que no provocan error nunca establecerán el valor de la variable **errno**. Por esto hay que llevar cuidado y consultar el valor de **errno** **<u>justo después de haberse producido el error</u>**.

La siguiente tabla nos muestra una lista con los códigos de error, los cuales están definidos en el fichero <.../errno.h>. Los puntos suspensivos indican que este fichero estará ubicado en un directorio diferente en cada instalación.

La función perror()

Aparte, existe la función **perror()**, que da un mensaje (en inglés) sobre el error producido.

La numeración de la tabla siguiente, puede variar de sistema en sistema, por lo que se recomienda se utilicen exclusivamente los nombres de las variables simbólicas y no la numeración de dicha tabla.

```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

#define EPERM           1      /* Operation not permitted */
#define ENOENT          2      /* No such file or directory */
#define ESRCH           3      /* No such process */
#define EINTR           4      /* Interrupted system call */
#define EIO             5      /* I/O error */
#define ENXIO           6      /* No such device or address */
```

```
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
#define ENOTTY          25      /* Not a typewriter */
#define ETXTBSY         26      /* Text file busy */
#define EFBIG           27      /* File too large */
#define ENOSPC          28      /* No space left on device */
#define ESPIPE          29      /* Illegal seek */
#define EROFS           30      /* Read-only file system */
#define EMLINK          31      /* Too many links */
#define EPIPE           32      /* Broken pipe */
#define EDOM            33      /* Math argument out of domain of func */
#define ERANGE          34      /* Math result not representable */
#define EDEADLK         35      /* Resource deadlock would occur */
#define ENAMETOOLONG    36      /* File name too long */
#define ENOLCK          37      /* No record locks available */
#define ENOSYS          38      /* Function not implemented */
#define ENOTEMPTY       39      /* Directory not empty */
#define ELOOP           40      /* Too many symbolic links encountered */
#define EWOULDBLOCK     EAGAIN  /* Operation would block */
#define ENOMSG          42      /* No message of desired type */
#define EIDRM           43      /* Identifier removed */
#define ECHRNG          44      /* Channel number out of range */
#define EL2NSYNC        45      /* Level 2 not synchronized */
#define EL3HLT          46      /* Level 3 halted */
#define EL3RST          47      /* Level 3 reset */
#define ELNRNG          48      /* Link number out of range */
#define EUNATCH         49      /* Protocol driver not attached */
#define ENOCSI          50      /* No CSI structure available */
#define EL2HLT          51      /* Level 2 halted */
#define EBADE           52      /* Invalid exchange */
#define EBADR           53      /* Invalid request descriptor */
#define EXFULL          54      /* Exchange full */
#define ENOANO          55      /* No anode */
#define EBADRQC         56      /* Invalid request code */
#define EBADSLT         57      /* Invalid slot */

#define EDEADLOCK       EDEADLK

#define EBFONT          59      /* Bad font file format */
#define ENOSTR          60      /* Device not a stream */
#define ENODATA         61      /* No data available */
#define ETIME           62      /* Timer expired */
#define ENOSR           63      /* Out of streams resources */
#define ENONET          64      /* Machine is not on the network */
#define ENOPKG          65      /* Package not installed */
#define EREMOTE         66      /* Object is remote */
#define ENOLINK         67      /* Link has been severed */
#define EADV            68      /* Advertise error */
#define ESRMNT          69      /* Srmount error */
#define ECOMM           70      /* Communication error on send */
#define EPROTO          71      /* Protocol error */
#define EMULTIHOP       72      /* Multihop attempted */
#define EDOTDOT         73      /* RFS specific error */
#define EBADMSG         74      /* Not a data message */
#define EOVERFLOW       75      /* Value too large for defined data type */
#define ENOTUNIQ        76      /* Name not unique on network */
#define EBADFD          77      /* File descriptor in bad state */
#define EREMCHG         78      /* Remote address changed */
#define ELIBACC         79      /* Can not access a needed shared library */
#define ELIBBAD         80      /* Accessing a corrupted shared library */
#define ELIBSCN         81      /* .lib section in a.out corrupted */
#define ELIBMAX         82      /* Attempting to link in too many shared libraries */
#define ELIBEXEC        83      /* Cannot exec a shared library directly */
#define EILSEQ          84      /* Illegal byte sequence */
#define ERESTART        85      /* Interrupted system call should be restarted */
#define ESTRPIPE        86      /* Streams pipe error */
#define EUSERS          87      /* Too many users */
#define ENOTSOCK        88      /* Socket operation on non-socket */
#define EDESTADDRREQ    89      /* Destination address required */
#define EMSGSIZE        90      /* Message too long */
#define EPROTOTYPE      91      /* Protocol wrong type for socket */
#define ENOPROTOOPT     92      /* Protocol not available */
#define EPROTONOSUPPORT 93      /* Protocol not supported */
#define ESOCKTNOSUPPORT 94      /* Socket type not supported */
#define EOPNOTSUPP      95      /* Operation not supported on transport endpoint */
#define EPFNOSUPPORT    96      /* Protocol family not supported */
#define EAFNOSUPPORT    97      /* Address family not supported by protocol */
#define EADDRINUSE      98      /* Address already in use */
#define EADDRNOTAVAIL   99      /* Cannot assign requested address */
#define ENETDOWN        100     /* Network is down */
#define ENETUNREACH     101     /* Network is unreachable */
#define ENETRESET       102     /* Network dropped connection because of reset */
#define ECONNABORTED    103     /* Software caused connection abort */
#define ECONNRESET      104     /* Connection reset by peer */
#define ENOBUFS         105     /* No buffer space available */
#define EISCONN         106     /* Transport endpoint is already connected */
#define ENOTCONN        107     /* Transport endpoint is not connected */
#define ESHUTDOWN       108     /* Cannot send after transport endpoint shutdown */
#define ETOOMANYREFS    109     /* Too many references: cannot splice */
```

```
#define ETIMEDOUT       110     /* Connection timed out */
#define ECONNREFUSED    111     /* Connection refused */
#define EHOSTDOWN       112     /* Host is down */
#define EHOSTUNREACH    113     /* No route to host */
#define EALREADY        114     /* Operation already in progress */
#define EINPROGRESS     115     /* Operation now in progress */
#define ESTALE          116     /* Stale NFS file handle */
#define EUCLEAN         117     /* Structure needs cleaning */
#define ENOTNAM         118     /* Not a XENIX named type file */
#define ENAVAIL         119     /* No XENIX semaphores available */
#define EISNAM          120     /* Is a named type file */
#define EREMOTEIO       121     /* Remote I/O error */
#define EDQUOT          122     /* Quota exceeded */

#define ENOMEDIUM       123     /* No medium found */
#define EMEDIUMTYPE     124     /* Wrong medium type */

#endif
```

Ejemplo **c004.c**:

de cómo se muestra el código de error generado en errno al fallar una llamada al sistema, en este caso el intento de cerrar un fichero cuyo identificador es 23 (fid = file id)

```
// ==================================================================
// Programa #4: Consultar el valor numérico de la variable global "errno"
// Archivo: ejemplo4.c
// ==================================================================
#include <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(23) == -1) { // Lógicamente, se produce un error. ¿Cual?
     printf ("Motivo del error de la última llamada al sistema: %d\n", errno );
     return 1;
     }
return 0;
}
```

Ejemplo c005.c:

de cómo se muestra el texto de error asociado al error ocurrido, utilizando la función **strerror()** que se limita a extraer el texto asociado al código de error pasado.

```
// ==================================================================
// Programa #5: Consultar el mensaje asociado al valor numérico de "errno"
// Archivo: ejemplo5.
// ==================================================================
#include <stdio.h> // La función "strerror" se define en <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(23) == -1){ // Lógicamente, se produce un error. ¿Cual? "Bad file number"
      printf ("Motivo del error de la última llamada al sistema: %d\n", strerror(errno) );
      return 1;
      }
return 0;
}
```

Ejemplo c006.c:

de cómo se muestra el texto de error utilizando la función **perror()** que adicionalmente recibe un parámetro para poder añadir un texto al error mostrado. En el ejemplo se pasa **NULL**, pero se podría haber pasado cualquier texto que se concatenaría al mensaje mostrado.

```
// ===================================================================
// Programa #6: Imprimir los errores en "stderr" en lugar de en "stdio"
// Archivo: ejemplo6.c
// ===================================================================
#include <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(23) == -1) { // Lógicamente, se produce un error. ¿Cual?
     // Imprime en stderr el motivo igual que strerror(errno): "Bad file number"
     perror ( NULL );
     return 1;
     }
return 0;
}
```

Ejemplo c007.c:
de cómo se muestra el mensaje concatenado que pasamos a **perror()**

```
// ===================================================================
// Programa #7: Imprimir los errores en "stderr" en lugar de en "stdio"
// Archivo: ejemplo7.
// ===================================================================
#include <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(23) == -1) {// Lógicamente, se produce un error. ¿Cual?
     // Imprime la cadena especificada y el motivo: "Ha cascado: Bad file number"
     perror ("Ha cascado: ");
     return 1;
     }
return 0;
}
```

Ejemplo c008.c:
de cómo podemos tratar de manera diferente en función del código (constante genérica) del código de error producido.

```
// ===================================================================
// Programa #8: Uso de las constantes
// Archivo: ejemplo8.c
// ===================================================================
#include <stdio.h>
#include <unistd.h>
#include <errno.h> // La variable "errno" se declara en este fichero
int main (int argn, char *argv[]) {
  // Llamada al sistema para cerrar un fichero que no está abierto
  if ( close(-23) == -1){ // Lógicamente, se produce un error. ¿Cual?
      if (errno==EBADF) {
          // Bad file descriptor
          printf ( "Descriptor de fichero incorrecto\n" );
          }
      else if (errno==EIO) {
              // Input/Output Error
              printf ( "Error físico de E/S\n" );
              }
      else {
         printf ( "Error: %s\n", strerror(errno) );
         }
   }
return 0;
}
```

Variables de entorno

Ejemplo **c009.c**:
de cómo un tercer parámetro con las variables de entorno es procesado en main. Recibe array de cadenas de variables de entorno definidas en el Shell que lanza al proceso.

```
// ===========================================
// Programa #9: Tabla de variables de ambiente
// Archivo: ejemplo9.c
// ===========================================
#include <stdio.h>
int main (int argn, char *argv[], char *env[]) {
  int i;
  for (i=0; env[i]!=NULL; i++) {
       printf ("%s\n",env[i]);
       }
return 0;
}
```

Para más detalles sobre la variable de **environ** ver

http://man7.org/linux/man-pages/man7/environ.7.html

Ejemplo c010.c:
Vemos cómo se puede crear una tabla de variables de entorno

```
// ===========================================
// Programa #10: Tabla de variables de ambiente
// Archivo: ejemplo10.
// ===========================================
#include <stdio.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ
int main (void) {
  int i;
  for (i=0; environ[i]!=NULL; i++)
      printf ("%s\n",environ[i]);
return 0;
}
```

Ejemplo c011.c:
de cómo podemos consultar el valor de una determinada variable de entorno, en este caso PATH

```
// ======================================================
// Programa #11: Consulta de la variable de ambiente PATH
// Archivo: ejemplo11.c
// ======================================================
#include <stdio.h>
#include <string.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ
int main (void) {
  int i;
  for (i=0; environ[i]!=NULL; i++) {
      if (memcmp(environ[i],"PATH=",5) == 0) {
          printf ("La variable PATH vale: %s\n",environ[i]+5);
          break;
          }
      }
return 0;
}
```

Ejemplo c012.c:
De cómo podemos separar las distintas rutas de la variable PATH en distintas cadenas.

```
// =========================================================
// Programa #12: Separación de las rutas de la variable PATH
// Archivo: ejemplo12.
// =========================================================
#include <stdio.h>
#include <string.h>
#include <unistd.h>
extern char **environ; // Declaración externa de environ

// Prototipos de las funciones utilizadas
// Consultar la variable de ambiente PATH
static char *GetEnvPath(void);

// Extraer las distintas rutas desde la variable de ambiente PATH
static int GetPathDirs(char *path,char *patharray[],int arraysize);

// Mostrar las rutas ya separadas
static void PrintPaths(char *paths[],int arraysize);

//-----------------------------------------
int main (void){
char *pathexp;
char pathbuf[1000];
char *paths[100];
int count;

pathexp = GetEnvPath();
strcpy (pathbuf,pathexp);
count = GetPathDirs(pathbuf,paths,100);
PrintPaths (paths,count);
return 0;
}

//-----------------------------------------
static char *GetEnvPath (void) {
int i;
for ( i=0; environ[i]!=NULL; i++) {
    if (memcmp(environ[i],"PATH=",strlen("PATH=")) == 0) {
        return environ[i];
        }
    }
return "";
}

//-----------------------------------------
static int GetPathDirs (char *path, char *array[], int size){
int i;
int pos;
path+=strlen("PATH=");
array[0]=path;
for ( i=0,pos=1; path[i]!=0; i++) {
    if ( path[i]==':' ) {
        path[i]=0;
        array[pos++] = path+i+1;
        if (pos == size)
            return size;
        }
    }
return pos;
}

//-----------------------------------------
static void PrintPaths (char *paths[], int size) {
int i;
printf ("Total de rutas encontradas: %d\n", size);
for (i=0; i<size; i++)
     printf ("%d. : %s\n", i+1, paths[i] );
}
```

Una forma más cómoda de obtener el valor de una variable a partir de su nombre es con la función "**getenv(nombre)**". Nos evita que realizar un bucle en busca de la entrada en la tabla de variables de ambiente. En el siguiente ejemplo pasamos desde la línea de órdenes como parámetro el nombre de la variable de ambiente que queremos visualizar. Algo parecido al comando echo.

Ejemplo c013.c:

Por ejemplo, para visualizar el valor de la variable PATH podríamos hacer:

$ echo $PATH

$ ejemplo13 PATH

En ambos casos obtendremos los mismos resultados.

```
// =================================================================
// Programa #13: Obtención del valor de una variable de ambiente con getenv()
// Archivo: ejemplo13.c
// =================================================================
#include <stdio.h>
#include <stdlib.h>

int main (int argn, char *argv[]) {
int i, char *valor;
// Si sólo tenemos un argumento (nombre del programa) salimos.
if (argn < 2) exit(1);

// Mostramos el valor de todas las variables especificadas (si existen)
for ( i=1; i<argn; i++) {
    valor = getenv ( argv[i] );
    if ( valor == NULL ) {
        printf ("Variable de ambiente '%s' no definida\n", argv[i]);
        }
    else {
        printf ("Valor de '%s' = %s\n",argv[i], valor);
        }
    }
return 0;
}
```

Procesos

Como ya se ha estudiado, un proceso es un programa que está siendo ejecutado por un procesador. Un programa queda completamente definido por el código que resulta de la compilación y del enlazado del código fuente del programa. Este código contiene una serie de instrucciones que debe ejecutar un procesador para realizar una determinada tarea y es una estructura pasiva. Sin embargo el proceso es una estructura activa que incluye el código del programa y sus datos, también el estado de aquellos recursos que necesita el programa para ejecutarse.

El manejo de concurrencia exige a su vez la necesidad de mecanismos que hagan posible la solución de los problemas que surgen de la propia definición de concurrencia, problemas tales como el acceso a una región crítica (exclusión mutua), el interbloqueo, etc.

El S.O. UNIX al ser un Sistema Operativo Multiusuario y Multitarea, nos permite la posibilidad de programar procesos concurrentes, además, posee un conjunto de recursos para comunicación y sincronización entre procesos, que pueden utilizarse mediante llamadas a funciones del sistema formando parte de un programa en lenguaje C.

Este conjunto de recursos (semáforos, paso de mensajes, etc.) se encuentran en la **librería IPC (InterProcess Communication)**.

Este documento tiene como objetivo principal el familiarizarnos con los mecanismos de comunicación entre procesos existentes en UNIX, sobre todo con la utilización de la función **fork()**.

Algunas de las funciones relacionadas con el manejo de procesos son las siguientes:

1. **fork** crea un nuevo proceso.
2. **wait** espera a que el proceso hijo pare o termine.
3. **exit** termina el proceso en curso.
4. **_exit** termina el proceso en curso sin limpieza de I/O.
5. **execl** ejecuta un programa con una lista de argumentos.
6. **execv** ejecuta un programa con un vector de argumentos.
7. **execle** ejecuta un programa con una lista de argumentos y un vector de entorno.
8. **execlp** ejecuta un programa con una lista de argumentos y un PATH de búsqueda.
9. **execvp** ejecuta un programa con un vector de argumentos y un PATH de búsqueda.
10. **getuid** obtiene el user-ID.
11. **getpid** obtiene el process-ID.
12. **getppid** obtiene el parent-process-ID.
13. **getpgr** obtiene el process-group-ID.
14. **getgid** obtiene el group-ID.

Estas rutinas de la librería, se usan para crear y manipular procesos y programas. La diferencia entre programa y proceso es bastante sutil. Un programa será un archivo que reside en el disco de sistema y que se crea con otros programas, como por ejemplo el compilador de C o el editor nano, mientras que un proceso es una copia del programa en la memoria que está haciendo algo.

Un proceso comprende todo el entorno de ejecución del programa, es decir, desde las variables, los ficheros abiertos, el directorio en el que reside, información acerca de usuario que ejecuta el proceso y el terminal donde lo ejecuta así como el código del programa.

La orden ps

El sistema operativo nos permite ver el estado de los procesos en un determinado momento mediante la orden **ps**. Por ejemplo este puede ser el listado que nos presenta la orden ps en un determinado momento:

```
PID TTY STAT TIME COMMAND
1615 p2 S 0:00 -bash
2420 p2 S 0:00 man ps
2422 p2 S 0:00 sh -c /usr/bin/gunzip -c /usr/man/cat1/ps.1.gz | /usr/bin/l
2424 p2 S 0:00 /usr/bin/less -is
2427 p5 S 0:00 -bash
2649 p5 R 0:00 ps
```

donde:

- PID es el identificador de proceso.
- TTY terminal asociada a ese proceso.
- STAT estado en el que están los procesos, S dormido, R ejecutándose, ... (ver man)
- TIME es el tiempo que lleva el proceso ejecutándose.
- COMMAND es la orden correspondiente al proceso.

Mediante la orden **man** consultaremos las distintas opciones de la orden ps. Algunas interesantes:

-l: listado largo
-u: nombre del usuario
-a: procesos de otros usuarios
-f: representa en una especie de árbol el parentesco entre los procesos

Existen otras opciones que se pueden mirar en **man ps**, bien directamente en la sesión del alumno o bien en http://man7.org/linux/man-pages/man1/ps.1.html

La orden top

Nos permite ver de forma dinámica el estado del sistema en general y el estado de los procesos, el tiempo de CPU consumido, la memoria consumida, la prioridad asignada a los procesos, etc.

Para mayor detalle de la orden top ver man top o http://man7.org/linux/man-pages/man1/top.1.html

Llamadas al sistema relacionadas con procesos

A continuación se hace una revisión de las principales llamadas al sistema para la gestión de procesos. Pero antes comentar brevemente el estándar que vamos a seguir para la especificación de las sintaxis y semántica de las llamadas, el estándar POSIX.

POSIX

Las diferencias a nivel de sistema entre los diferentes sistemas Unix de diferentes vendedores, provocó la aparición de diferentes estándares en la especificación de la interfaz de los sistemas con los programas de aplicación. Nosotros vamos a seguir los siguientes estándares: ANSI C, POSIX y Spec 1170.

La IEEE desarrolló una serie de estándares denominados POSIX (Portable Operating System Interface) que especifican la interfaz entre el sistema operativo y el usuario de modo que los programas sean transportables a través de diferentes plataformas.

De entre los diferentes miembros del grupo de estándares nosotros nos centraremos en el POSIX.1 que especifica la API (Lenguaje C).

Credenciales de usuario

Cada usuario del sistema se identifica por un número único denominado ID de usuario o UID, y pertenece a un grupo de usuarios, que posee su ID de grupo o GID. Estos identificadores afectan a la propiedad de los archivos, a los permisos de acceso, y a la capacidad de enviar señales a otros procesos. Estos atributos se denominan globalmente credenciales.

El sistema reconoce a un usuario privilegiado denominado superusuario (normalmente este usuario accede al sistema como root). El superusuario tiene UID = 0 y GID = 1. El superusuario tiene muchos privilegios como el acceso a archivos de otros usuarios, a pesar de la protección de estos, y puede ejecutar un cierto número de llamadas al sistema privilegiadas, como por ejemplo, mknod para crear un nodo del sistema de archivos. Muchos sistemas UNIX actuales tales como SVR4.1/ES soportan mecanismos de seguridad incrementada. Estos sistemas sustituyen la abstracción de superusuario privilegiado único con privilegios para operaciones diferentes.

Cada proceso tiene dos pares de IDs – el real y el efectivo. Cuando un usuario entra al sistema, el pograma de entrada (login) establece ambos pares a los UID y GID especificados en la base de datos de password (el archivo /etc/passwd, o algún mecanismo distribuido como el Network Information Service (NIS) de Sun Microsystems). Cuando un proceso invoca a **fork**, el hijo hereda las credenciales de su padre.

El UID efectivo y el GID efectivo afectan a la creación y acceso de archivos.

Durante la creación de un archivo, el kernel establece los atributos de propiedad del archivo a los UID y GID efectivos del proceso creador. Durante el acceso al archivo, el kernel utiliza los UID y GID efectivos del proceso para determinar si puede acceder al archivo. Los UID y GID reales identifican al propietario real del proceso y afectan a los permisos para enviar señales. Un proceso sin privilegios de superusuario puede enviar señales a otro proceso sólo si el UID real o efectivo del emisor iguala al UID real del receptor.

Existen tres llamadas al sistema que pueden cambiar las credenciales. Si un proceso llama a exec para ejecutar un programa instalado en modo suid, el kernel cambia el UID efectivo del proceso al del propietario del archivo. De la misma forma, si el programa esta instalado en modo sgid, el kernel cambia el GID efectivo del proceso llamador.

UNIX suministra este mecanismo para otorgar privilegios especiales a usuarios para realizar tareas concretas. El ejemplo clásico es el programa passwd, que permite a un usuario cambiar su propio password. Este programa debe escribir en la base de datos de password, en la cual no esta permitida la escritura directa por parte de los usuarios (para evitar la modificación de password de otros usuarios). Así, el programa passwd es propiedad del superusuario y tiene activo el bit SUID. Esto permite al usuario obtener los privilegios de superusuario mientras ejecuta el programa passwd.

Un usuario puede cambiar sus credenciales llamando a setuid o setgid. El superusuario puede invocar estas llamadas al sistema para cambiar tanto los UID y GID efectivos como los reales. Los usuarios ordinarios pueden utilizarlas para cambiar sus UID y GID efectivos y devolverlos a los reales.

A continuación se recogen las funciones con las cuales podemos consultar los identificadores de proceso, del padre del proceso, el identificador de usuario real y efectivo, y el identificador de grupo real y efectivo, del proceso llamador, respectivamente.

```
# include <sys/types.h>
# include <unistd.h>
pid_t getpid(void) Retorna: ID del proceso que la invoca
pid_t getppid(void) Retorna: ID del padre
pid_t getuid(void) Retorna: ID real de usuario
pid_t geteuid(void) Retorna: ID efectivo de usuario
pid_t getgid(void) Retorna: ID real del grupo
pid_t getegid(void) Retorna: ID efectivo del grupo
```

Observar que ninguna de estas funciones devuelve código de error.

Llamada	Descripción
Includes: #include <unistd.h> #include <sys/types.h> **Declaración:** pid_t getpid(void) pid_t getppid(void) **Uso:** pid = getpid() ppid = getppid()	**getpid**: Obtener el identificador único (PID) del proceso que haga la llamada. Se suele utilizar para construir nombres de ficheros temporales. **getppid**: Obtener el identificador único del proceso padre (PPID) del proceso llamador. ERRORES: Estas dos llamadas al sistema siempre tienen éxito, y no retornan ningún valor para indicar si se ha producido un error.
Includes: #include <unistd.h> #include <sys/types.h> **Declaración:** uid_t getuid(void) uid_t geteuid(void) gid_t getgid(void) **Uso:** uid = getuid() euid = geteuid() gid = getgid()	**getuid()**: Obtener el ID del propietario real del proceso llamador. **geteuid()**: Obtener el ID del propietario efectivo del proceso llamador. **getgid()**: Obtener el ID de grupo del propietario El propietario real es el usuario que invocó el programa. Puesto que el usuario real puede dar permisos adicionales a otros usuarios durante la ejecución de procesos "set-user-ID", getuid() se utiliza para determinar el ID del usuario real. ERRORES: Estas dos llamadas al sistema siempre tienen éxito, y no retornan ningún valor para indicar si se ha producido un error.

Llamada	Descripción
Includes: #include <unistd.h> #include <sys/types.h> **Declaración:** pid_t getpgrp(void) pid_t getpgid(pid_t pid) **Uso:** pgrp = getpgrp() pgid = getpgid(pid)	**getpgrp()**: Obtener el identificador de grupo PID al que pertenece el proceso actual. **getpgid(pid)**: Obtener el identificador de grupo PID al que pertenece el proceso especificado. Si pid es 0, funciona exactamente igual que getpgrp(). Los grupos de procesos se utilizan para distribución de señales, y por los terminales para regular peticiones de su entrada: los procesos que están en el mismo grupo en el que el terminal es un proceso de primer plano (foreground) podrán leer, mientras que el resto se pueden bloquear mediante una señal si intentasen leer. Estas llamadas se utilizan, por tanto, por programa como el shell (sh) para crear grupos de procesos para la implementación de trabajos de control. Tenemos además dos llamadas adicionales: **tcgetpgrp()** y **tcsetpgrp()**, que se utilizan para obtener y establecer el grupo de procesos del terminal de control.

fork

El único modo de que el núcleo de Unix cree un nuevo proceso es mediante la ejecución de la llamada **fork()** realizada por un proceso existente.

El nuevo proceso que se crea se llama “proceso hijo”. El proceso al que llama fork lo llamaremos padre y al nuevo proceso (la copia) lo llamará hijo.

Esta función devuelve un valor numérico que será distinto para el padre y para el hijo. El núcleo devuelve un valor cero para el proceso hijo, mientras que al proceso padre le devuelve el identificador del proceso hijo.

El motivo de esto es que un proceso padre puede tener varios hijos mientras que los hijos tienen un solo padre (existe una llamada al sistema que puede permitir al hijo obtener el identificador de su proceso padre: **getppid()**).

El proceso hijo es una copia exacta del proceso padre, de hecho el proceso hijo tiene una copia del espacio de datos, heap (memoria dinámica) y de la pila. Sin embargo, el espacio de direcciones de datos del proceso padre será distinto del espacio de direcciones del proceso hijo, lo cual implica que **ambos procesos no comparten los datos, sólo se hace un copia para el hijo**. Habitualmente el padre y el hijo compartirán el segmento de código ya que este es de sólo lectura.

Normalmente, el segmento de código es compartido -si es de sólo lectura. Algunas implementaciones no realizan una copia completa del padre dado que normalmente la operación fork es seguida de exec. En su lugar se emplea la técnica copia-sobre-escritura.

Ambos procesos siguen su ejecución en la instrucción siguiente a la llamada fork(), es decir, una vez creado el proceso ambos siguen ejecutándose por la instrucción que sigue a fork. En general, **no conoceremos nunca si el hijo se ejecuta antes que el padre o la inversa**. Esto dependerá del algoritmo de planificación.

fork retorna diferentes valores para cada proceso. Para el padre fork retorna el PID del hijo, y para el hijo fork retorna cero.

Como ejemplo de uso véase el siguiente programa.

Ejemplo c014.c:

De cómo se crea un proceso hijo y ambos (padre e hijo) se identifican.

```
#incluye <stdio.h>
#incluye <stdlib.h>
main (){
  if ( fork() == 0 )
      printf ("Este es el hijo\n");
  else
      printf ("Este es el padre\n");
exit(EXIT_SUCCESS);
}
```

Cuando ejecutemos el programa lo que obtendremos será :

```
Este es el padre

Este es el hijo
```

Las únicas diferencias entre el proceso padre y el hijo son los valores que retornan, el process-ID y el parent-process-ID.

<table>
<tr><th>Llamada</th><th>Descripción</th></tr>
<tr><td>Includes:
#include <unistd.h>
#include <sys/types.h>
Declaración:
pid_t fork(void)
Uso:
pid = fork()</td><td>Crea un proceso hijo con la misma imagen de proceso (core image) que el padre, funcionando ambos en paralelo. El hijo es una copia exacta del proceso padre, aunque con alguna diferencia que comentaremos a continuación.

No toma argumentos y retorna:
El valor -1 en caso de no haberse podido crear el proceso hijo (error), y se establece la variable global errno. Más adelante enumeramos los posibles errores que pueden producirse.
En el proceso padre, retorna el PID que se le ha asignado al proceso hijo
En el proceso hijo retorna 0
Esta es la forma de distinguir el proceso padre y el hijo. Un proceso puede hacer uso de una función getpid() para conocer su propio PID, así como de la función getppid() para conocer el PID del proceso padre. En el caso del proceso padre, su padre será el proceso 1, denominado proceso INIT, que representa al sistema operativo.

Ambos procesos tendrán iguales:
UID y GID
Código
Descriptores de ficheros. Todos los ficheros que tiene abiertos el padre también los tendrá el hijo, compartiendo los punteros de fichero. Dicho de otro modo, si el proceso hijo realiza un desplazamiento del puntero (lseek) el puntero del padre también se desplaza
Gestión de señales: idem.
Directorio de trabajo
Máscara de protección
Contador de programa: ambos procesos ejecutarán la siguiente instrucción a la llamada fork(), pero cada uno en su su copia del código.
Pero tendrán diferencias:
PID: puesto que son procesos diferentes tendrán un PID diferente y único.
PPID: del mismo modo, el PID del proceso padre de ambos es diferente.
Se anulan las posibles alarmas pendientes.
Cada proceso tendrá su propio espacio de proceso.
¡Las variables son copias (no se comparten)!
ERRORES:
[EAGAIN] The system-imposed limit on the total number of processes under execution would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROC. (The limit is actually one less than this except for the super user).
[EAGAIN] The user is not the super user, and the systemimposed limit on the total number of processes under execution by a single user would be exceeded. The limit is given by the sysctl(3) MIB variable KERN_MAXPROCPERUID.
[EAGAIN] The user is not the super user, and the soft resource limit corresponding to the resource parameter RLIMIT_NPROC would be exceeded (see getrlimit(2)).</td></tr>
</table>

wait y waitpid

Cuando un proceso hijo acaba se debe notificar a su padre el estado de terminación de éste.

De algún modo **el proceso padre deberá estar esperando la respuesta del hijo**. Esto se realiza mediante la llamada al sistema **wait()**.

El proceso que ejecuta la llamada al sistema **wait() quedará en espera** (se bloquea) hasta que se le notifique que cualquiera de sus hijos ya ha acabado y entonces seguirá ejecutando la línea de código siguiente a **wait()**.

Esta llamada tiene un **parámetro donde recogerá el estado de terminación del primer hijo que haya acabado**, además de que **devuelve el número de proceso del hijo que ha acabado**. Si no nos interesa el estado de terminación del proceso hijo se le puede pasar NULL.

El estado que recoge la función **wait()** se interpreta mediante una serie de macros ya definidas (ver hojas del manual de UNIX).

Existe una variante de esta llamada, es **waitpid()**. En este caso el proceso que ejecuta **waitpid()** esperará a un hijo concreto.

Podemos controlar la ejecución del proceso hijo llamando a **wait** en el padre.

wait fuerza al padre a detener la ejecución hasta que el proceso hijo haya terminado.

wait retorna el processID del proceso hijo que termina y almacena su status en el entero al que apunta el puntero que debe llevar como argumento (statusp) a menos que el argumento sea NULL en cuyo caso no almacena el status.

Ejemplo c014c:

El siguiente programa garantiza que el proceso hijo termina antes que el padre.

```
#incluye <stdio.h>
#incluye <stdlib.h>
main (){
if ( fork() == 0 )
  printf ("Este es el hijo\n");
else {
  wait (NULL)
  printf ("Este es el padre\n");
  }
exit (EXIT_SUCCESS);
}
```

La salida que obtendremos para el anterior programa será :

```
Este es el hijo

Este es el padre
```

wait retorna -1 en caso de fallo.

La finalización de un proceso se notifica a su proceso padre a través de una señal **SIGCHLD**.

Dado que es un evento asíncrono, el padre puede elegir entre ignorar la señal o puede suministrar una función que se ejecute cuando se reciba la señal, un manejador de la señal.

Un proceso que llama a **wait** o puede:

- bloquearse (si todos sus hijos se están ejecutando), o
- retornar inmediatamente con el estado de finalización de un hijo que ha finalizado), o
- retornar inmediatamente con código de error (si no tiene ningún hijo).

La sintaxis de las funciones es:

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

#include <sys/wait.h>

pid_t wait(int *stacloc);

pid_t waitpid (pid_t pid, int *statloc, int opcion);
```

Retornan: PID si OK, 0 (ver comentarios), ó -1 si error.

La diferencia entre las dos funciones es:

- **wait** puede bloquear al llamador hasta que un hijo termine, mientras que **waitpid** tiene una opción (parámetro) que evita el bloqueo.
- **waitpid** no espera la terminación del primer hijo, tiene una opción que permite controlar a que proceso esperar.

Para ambas funciones, el argumento **statloc** es un puntero a un entero. Si este argumento no es un puntero nulo, el estado de terminación del hijo se almacena en la posición apuntada por el argumento.

Si no estamos interesados por el valor, podemos pasar un puntero nulo. Tradicionalmente, el valor de estado devuelto por estas dos funciones ha sido definido por la implementación, con determinados bits indicando el estado de finalización.

POSIX.1 especifica para la visualización del estado de finalización varias macros mutuamente exclusivas definidas en <sys/wait.h>:

- **WIFEXITED(status)** - Devuelve cierto si el hijo ha terminado normalmente, en cuyo caso puede ejecutarse WEXITSTATUS(status) para obtener los 8 bits de orden inferior del argumento pasado por el hijo con exit o _exit.
- **WIFSIGNALED(status)** - Devuelve cierto si el hijo terminó anormalmente (recibió una señal que no pudo manejar). Ejecutaremos **WTERMSIG(status)** para obtener el número de la señal. Además, SVR4 y 4.3+BSD definen la macro **WCOREDUMP(status)** que devuelve cierto si se generó un archivo 'core'.
- **WIFSTOPPED(status)** - Devuelve cierto si el valor de status fue devuelto por un hijo que está actualmente detenido. Podemos ejecutar **WSTOPSIG(status)** para obtener el número de la señal que paró al hijo.

Respecto a los valores que puede tomar la opcion en waitpid:

- **WNHANG** - En este caso waitpid no será bloqueante si el hijo especificado por pid no esta disponible inmediatamente. En este caso devuelve 0.
- **WUNTRACED** - Si la implementación soporta control de trabajos, se devuelve el estado del hijo especificado por pid que ha sido detenido y cuyo estado no ha sido devuelto desde su detención. La macro **WIFSTOPPED(status)** determina si el valor de retorno corresponde a un hijo parado.

La interpretación del argumento **pid** para **waitpid** depende de su valor:

- pid == -1 espera por cualquier hijo (equivalente a wait).
- pid > 0 espera por el hijo cuyo PID es igual a pid.
- pid == 0 espera por cualquier hijo cuyo ID grupo es igual al del proceso llamador.
- pid < -1 espera por cualquier hijo cuyo ID grupo es igual al valor absoluto de pid.

wait da error sólo si el proceso llamador no tiene hijos.

waitpid dará error si el proceso o grupo especificado no existe o no es hijo del llamador. Otro posible error de retorno es la interrupción de la llamada por una señal.

Ejemplo c015c: Un programa sencillo ilustrando el uso de wait.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
void main (void){
pid_t childpid;
int status;

if ((childpid = fork()) == -1) {
   perror ("Fork ha fallado");
   exit(1);
   }
else if (childpid == 0)
        fprintf(stderr, "Soy el hijo con pid = %ld\n", (long)getpid());
else if (wait(&status) != childpid)
        fprintf (stderr, "Wait interrumpido por una señal\n");
else fprintf (stderr, "Soy el padre con pid = %ld\n",(long)getpid());
exit(0);
}
```

Llamada	Descripción
includes: #include <sys/types.h> #include <sys/wait.h> #include <sys/time.h> #include <sys/resource.h> NOTA: los 2 últimos #includes no son necesarios para la llamada wait. **Declaración:** pid_t wait(int *status) pid_t waitpid(pid_t wpid, int *status, int options) **Uso:** pid = wait(&estado)	Suspende la ejecución del proceso llamador, y espera que termine alguno de sus hijos o bien reciba una señal, momento en el cual se pondrá en marcha de nuevo el padre. VERSIONES: Tenemos varias versiones de la misma llamada. Todas retornan el PID del proceso que ha terminado y el estado de terminación del hijo. • **wait** es la más sencilla. Funciona tal y como acabamos de explicar. • **waitpid** nos permite especificar además: ▪ **wpid**: el conjunto de procesos a los que la llamada esperará. Si **wpid es -1**, la llamada esperará cualquier proceso; si **wpid es 0**, la llamada esperará procesos del mismo grupo (GID) que el proceso llamador; si **wpid es > 0**, la llamada esperará la terminación del proceso cuyo PID se ha especificado; si **wpid es < -1**, la llamada esperará la terminación de cualquier proceso cuyo GID (identificador de grupo) sea el valor absoluto del valor especificado en wpid. ▪ **options**: nos permite especificar distintas opciones unidas por el operador OR de bits "|". Las opciones pueden ser: • **WNOHANG**: utilizado para indicar que la llamada no debería bloquear la ejecución si no hay procesos que deseen informar de su estado. • **WUNTRACED**: si se establece, los hijos del proceso actual que se detengan **debido** a las señales **SIGTTIN**, **SIGTTOU**, **SIGTSTP** o **SIGSTOP** también informan de su estado. VALORES RETORNADOS: La llamada a **wait**, retorna el PID del proceso que ha terminado, así como el estado, que es un valor hexadecimal **0x0000**, donde los dos primeros (los de mayor peso) indican el estado del proceso hijo a la salida (lo que ha especificado en exit(n)) y los dos últimos (los de menor peso) el estado de terminación (si es 00 es que ha terminado bien). Si **wait** termina por otros motivos distintos de que algún proceso hijo haya terminado (por ejemplo, que haya llegado una señal), se nos retorna **-1**, y se establece la variable global **errno**. Las llamadas **waitpid()**, también funcionan del mismo modo. Además, si se ha especificado la opción WNOHANG y no hay hijos terminados, se retorna 0. El estado será: • -1 si el padre no tiene ningún hijo. En este caso la llamada a wait no tiene efecto. • 0 si el hijo ha terminado normalmente; • otro valor en caso de error. NOTAS: • Si un proceso padre termina sin esperar a que todos los procesos hijos terminen, a los procesos hijos huérfanos se les asigna el proceso 1 (INIT) como proceso padre. • Si una señal es capturada mientras se está bloqueado en un **wait()**, la llamada podrá ser interrumpida o reiniciada cuando la rutina manejadora de la señal retorne. Esto depende de las opciones establecidas para la señal. MACROS: Se dispone de una serie de macros: • **WIFSTOPPED(status)** True if the process has not terminated, but has stopped and can be restarted. This macro can be true only if the wait call specified the WUNTRACED option or if the child process is being traced (see ptrace(2)). Depending on the values of those macros, the following macros produce the remaining status information about the child process: • **WEXITSTATUS(status)** If **WIFEXITED(status)** is true, evaluates to the loworder 8 bits of the argument passed to _exit(2) or exit(3) by the child. • **WTERMSIG(status)** If **WIFSIGNALED(status)** is true, evaluates to the number of the signal that caused the termination of the process. • **WCOREDUMP(status)** If **WIFSIGNALED(status)** is true, evaluates as true if the termination of the process was accompanied by the creation of a core file containing an image of the process when the signal was received. • **WSTOPSIG(status)** If **WIFSTOPPED(status)** is true, evaluates to the number of the signal that caused the process to stop.

exit

Esta función pone fin a la ejecución de un proceso. Su sintaxis es:

```
#include <stdlib.h>
void exit (int estado);

#include <unistd.h>
void _exit(int estado)
```

Ambas funciones devuelven un valor del estado de finalización al proceso padre. **El estado de finalización es indefinido si**:

1. la función exit se invoca sin valor de estado
2. main realiza un return sin valor de retorno
3. finaliza main sin un return explícito.

Esta función termina el proceso que la invoca, con un código de status igual al byte más a la derecha del status. Cierra todos los descriptores del proceso.

Es conveniente devolver un valor en exit() ya que de lo contrario se devuelve basura en el código del estado del proceso. **Por convención se devuelve cero si no hay error y un código distinto de cero si hay error**.

Ejemplo c016.c:

En el siguiente programa el padre espera a que el hijo salga y luego imprime el valor de status del hijo. El hijo lee el exit status que va a ser introducido por el terminal y entonces devuelve este valor al padre a través del exit . Entonces el padre imprime el exit status del hijo.

```
#incluye <stdio.h>
#incluye <stdlib.h>
main (){
unsigned int status;
if ( fork() == 0 ){ /* ==0 en el hijo */
  scanf("%d",&status);
  exit (status);
  }
else { /* !=0 en el padre */
  wait (status);
  printf ("hijo exit status = %d\n",status >> 8);
  }
exit (0);
}
```

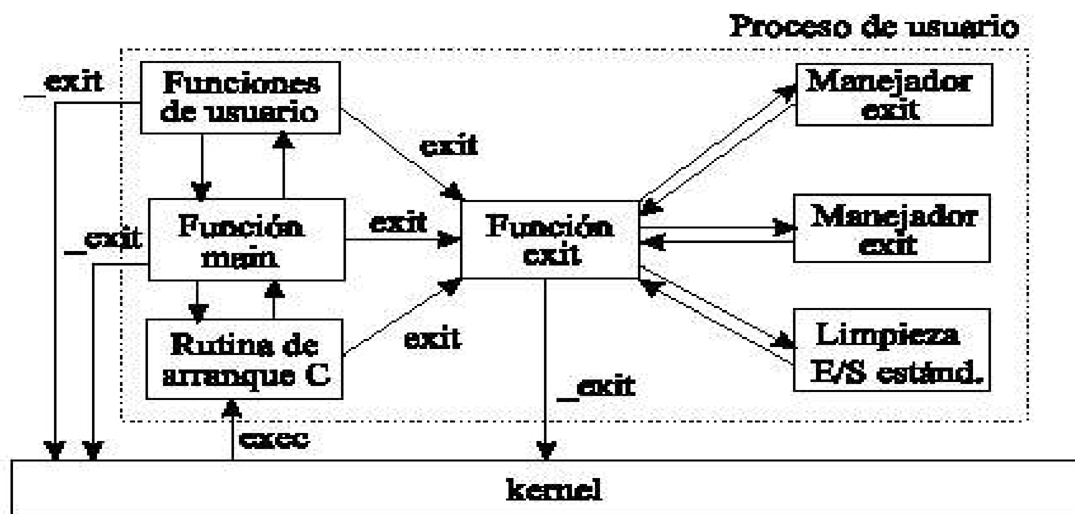

Figura 1.1.- Cómo se inicia y cómo finaliza un programa C.

La Figura 1.1 resume como se inicia un programa C y las varias formas en que puede terminar. Observar que la única forma en que se ejecuta un programa por el kernel es cuando llamamos algunas de las funciones exec. La única forma de finalizar voluntariamente un programa es cuando llamamos a la función _exit, ya sea explícitamente o implícitamente (a través de exit). Un proceso también puede terminar involuntariamente por una señal (esto no se recoge en la figura).

Existen varias formas terminación de un proceso:

Terminaciones normales:

1. Ejecutar un return desde la función main. Lo que es esquivalente a invocar a exit.
2. Llamar a la función exit. Esta función se define en ANSI C. Dado que ANSI C no trata con descriptores de archivos, procesos múltiples, ni control de trabajos, la definición de esta función es incompleta en los sistemas UNIX.
3. Llamar a la función _exit. Esta función se invoca por exit y maneja los detalles específicos de UNIX. La función se especifica en POSIX.1.

Terminaciones anormales:

1. Llamando a la función abort. Esto es un caso especial del siguiente caso, dado que genera la señal SIGABORT.
2. Cuando el proceso recibe determinadas señales. Dichas señales pueden ser generadas por el propio proceso, por otros procesos o por el kernel.

Llamada	Descripción
Includes: #include <unistd.h> **Declaración:** void _exit(int status) **Uso:** exit(estado)	Termina un proceso, retornando los dos dígitos hexadecimales que obtiene el proceso padre en "wait" que indican el estado en que termina un proceso hijo. Por convenio, retornamos un valor 0 cuando no hay error y un valor distinto de 0 cuando se produzca un error. VALORES RETORNADOS: La llamada exit nunca retorna un valor. Esta llamada provoca las siguientes acciones. Cuando un programa en C ejecuta la llamada al sistema exit, se realizan las siguientes, acciones: • se confirman los cambios de los buffers de los ficheros • se cierran todos los ficheros (descriptores de fichero) • se eliminan ficheros temporales • se notifica al proceso padre la terminación mediante el envío de la señal SIGCHLD • se establece como PPID de todos los procesos que hayan quedado huérfanos el valor 1, que equivale al proceso INIT. • si el proceso que ha terminado el es padre de todos los procesos de un grupo, se envían las señales SIGHUP y SIGCONT a todos los procesos huérfanos del grupo. • si el proceso que ha terminado es un proceso controlador (intro), se envía la señal SIGHUP al proceso de primer plano (foreground) que controla el terminal y se prohibe el acceso al terminal controlador.

exec

Uno de los usos de la orden **fork()** es la de crear otro proceso que ejecutará un programa diferente al del padre mediante la orden **exec()**. Cuando un proceso hace una llamada **exec()**, su código se reemplaza por el correspondiente al del nuevo programa que se quiere ejecutar.

Concretamente, **exec()** reemplaza el segmento de código, datos, heap y pila del proceso que hace la llamada por el correspondiente al programa que se quiere ejecutar.

El identificador del proceso no cambia ya que no se ha creado ningún nuevo proceso (además hay otras características del proceso invocante que se mantienen como: identificador real de usuario y de grupo, el identificador de sesión, el identificador de grupo de procesos, directorio actual de trabajo, directorio root, máscara de creación de ficheros, señales pendientes, etc. El identificador efectivo de usuario y grupo dependerá de si está activo o no el bit de usuario o grupo para el fichero que se vaya a ejecutar).

Cuando un proceso invoca a la función **exec**, el programa que ejecuta es totalmente sustituido por uno nuevo (nuevos segmentos de texto, datos, pila y heap), y el nuevo programa comienza ejecutando su función main. Existen **seis funciones exec** que únicamente se diferencian en la forma de pasar los argumentos. Sus sintaxis también se muestran en la tabla.

```
int execl (const char *path, char *const arg0, arg1, ..., NULL)
int execv (const char *path, char *const argv[])
int execle(const char *path, char *const arg0, arg1, ..., NULL, char *const envp[])
int execve(const char *path, char *const argv[],char *const envp[])
int execlp(const char *file, char *const arg0, arg1, ..., NULL)
int execvp(const char *file, char *const argv[])
```

Valor de Retorno:

No retornan nada si tienen éxito, -1 si error.

La primera diferencia entre estas funciones es que las cuatro primeras aceptan un pathname de archivo como argumento y las dos últimas un nombre de archivo. Cuando especificamos un nombre de archivo, si este contiene una barra se interpreta como un camino, en cualquier otro caso se busca un ejecutable en los directorios especificados en la variable del entorno PATH.

Si el archivo especificado no es ejecutable, se asume que es un programa shell e intenta invocar a /bin/sh como el nombre de archivo como argumento.

Otra diferencia afecta al paso de la lista de argumentos (**l** de lista y **v** de vector). Las funciones **execl**, **execlp** y **execle**, requieren que los argumentos del nuevo programa se especifiquen por separado y que se marque el fin de los argumentos con un puntero nulo.

Para las otras funciones, construiremos una matriz de punteros a los argumentos, y se pasa como argumento la dirección de la matriz.

La diferencia final esta en el paso de la lista de variables de entorno al nuevo programa. Las funciones que acaban con **e** permiten pasar un puntero a una matriz de punteros a la cadena de entorno. Las otras funciones usan la variable **environ** (variable global que contiene la dirección de la matriz de punteros, extern char **environ) en el proceso llamador para copiar la lista de entorno. Las variables de entorno se definen de la forma "nombre=valor". La Figura 1.2 ilustra un entorno que consta de cinco cadenas.

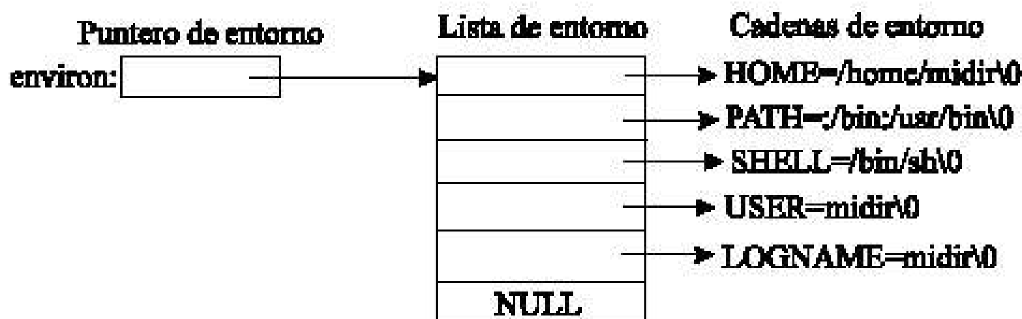

Figura 1.2.- Entorno C con cinco cadenas de caracteres.

En la mayoría de las implementaciones sólo una de estas funciones (execve) es una llamada al sistema. Las otras son funciones de biblioteca que invocan a ésta.

El nuevo programa hereda del proceso invocador las siguientes propiedades:

- Los identificadores del proceso, de grupo y de sesión,
- El terminal de control,
- Tiempo restante del reloj de alarma,
- Directorios actual y root,
- Mascara de creación de archivos,
- Bloqueo de archivos,
- Mascara de señales y señales pendientes,
- Límites de recursos.

El manejo de los archivos abiertos depende del valor del indicador close-on-exec para cada descriptor (ver función **fcntl**).

Ejemplo c017c:

Programa que crea un proceso para ejecutar una orden pasada como argumento.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
void main(int argc, char *argv[]) {
pid_t childpid;
int status;
if ((childpid = fork() == -1) {
   perror("Fork ha fallado");
   exit(1);
   }
else if (childpid == 0) { /*codigo del hijo*/
        if (execvp(argv[1], &argv[1]) < 0) {
            perror("exec ha fallado");
            exit(1);
            }
     }
else /*codigo del padre*/
   while (childpid != wait(&status))
          if ((childpid == -1) && (errno != EINTR)
             break;
exit(0);
}
```

Las rutinas **exec** son llamadas para ejecutar un programa . Esto lo hacen sustituyendo el programa que se está ejecutando por el nuevo programa especificado en **exec**. Las rutinas **exec** no retornan valor cuando terminan adecuadamente y -1 cuando lo hacen mal.

Hay seis rutinas que pueden ser llamadas:

execl() Toma el Path del programa ejecutable como primer argumento. El resto de los argumentos son una lista de argumentos de la línea de comandos para el nuevo programa (argv[]).

```
#incluye <unistd.h>
...
execl ("/bin/cat", "cat", "f1", "f2", NULL);
execl ("a.out", "a.out", NULL);
```

execle() Igual que execl , excepto que el final de la lista de argumentos esta seguido por un puntero a una lista de caracteres terminada por un NULL que se pasa como el entorno del nuevo programa .

```
#incluye <unistd.h>
...
static char *env[] = { "TERM=ansi", "PATH=/bin:/usr/bin", NULL };
...
execle ("/bin/cat","cat","f1","f2",NULL,env);
```

execv() Toma el nombre del Path del programa ejecutable como su primer argumento. El segundo argumento es un puntero a una lista de punteros a caracteres que se pasa como argumentos en la linea de comandos para el nuevo programa.

```
#incluye <unistd.h>
...
static char *args[] = {"cat","f1","f2",NULL};
...
execv ("/bin/cat",args);
```

execve() Igual que execv excepto que el tercer argumento el un puntero a una lista de caracteres que pasa el entorno para el nuevo programa.

```
#incluye <unistd.h>
...
static char *env[] = { "TERM=ansi", "PATH=/bin:/usr/bin", NULL };
static char *args[] = { "cat", "f1", "f2", NULL };
...
execle ("/bin/cat",args,env);
```

execlp() Igual que execl con la diferencia de que el nombre del programa no tiene que ser el nombre del Path y puede ser un programa shell en lugar de un módulo ejecutable.

```
#incluye <unistd.h>
...
execlp ("ls", "ls", "-1", "/usr", NULL);
```

execvp() Igual que execv, solo que el nombre del programa no tiene que ser el nombre de Path, y puede ser un programa Shell en lugar de un módulo ejecutable.

```
#incluye <unistd.h>
...
static char *args[] = { "cat", "f1", "f2", NULL };
...
execv ("cat", args);
```

Llamada	Descripción
Includes: #include <unistd.h> **Declaración:** int execv(const char *path, char *const argv[]) int execvp(const char *path, char *const argv[]) int execve(const char *path, char *const argv[], char *const envp[]) int execl(const char *path, char *const arg0, arg1, ..., NULL) int execlp(const char *path, char *const arg0, arg1, ..., NULL) int execle(const char *path, char *const arg0, arg1, ..., NULL, char *const envp[]) **Uso:** res = execv(...) res = execvp(...) res = execve(...) res = execl(...) res = execlp(...) res = execle(...)	En las llamadas al sistema **exec..()**, al contrario que **wait()**, el proceso hijo no ejecutará el mismo programa que el proceso padre, sino que ejecuta código almacenado en un fichero. Este fichero podrá ser un fichero binario ejecutable o un script. Para poder ejecutar un script debemos especificar como nombre de fichero lo siguiente: "# ! sh [arg]", donde sh es el intérprete de comandos. VERSIONES: Tenemos 6 posibilidades ya comentadas anteriormente. execv execvp execve execl execlp execle VALORES RETORNADOS: Estas 6 funciones **exec..()** retornan 0 si se ejecutan correctamente y -1 si fracasan (por ejemplo, si el fichero no existe), estableciendo la variable global **errno**. Cuando se ejecuta **exec..()**, el fichero de programa del primer argumento se carga en memoria, en el espacio de direcciones del proceso llamador y sobreescribe el programa que hay. Después, el programa recibe los argumentos y comienza su ejecución. Es decir, se cambia la imagen del proceso. Cuando se carga un programa y se ejecuta, dicho programa recibe los argumentos del modo habitual: main(int argn, char **argv, char **envp) donde argn es el número de elementos del array argv y argv apunta al array de cadenas de caracteres que representan los argumentos. Opcionalmente, si hemos pasado el array de variables de ambiente, se reciben exáctamente igual que los que se reciben desde la línea de órdenes, salvo que en vez de contener la tabla de variables del sistema, contendrá la tabla de variables especificadas.

<table>
<tr>
<td></td>
<td>
En el nuevo proceso hay ciertas características que se conservan o que cambian:

Los descriptores de ficheros abiertos se mantienen.
Las señales establecidas para que se ignoren (SIG_IGN) en el proceso llamador, se establecerán también que se ignoran en el nuevo proceso.
Las señales en las que se ha definido una función manejadora, se establecen a la acción por defecto (SIG_DLF) en el nuevo proceso, es decir, exit(0).
El nuevo proceso hereda los siguientes atributos desde el proceso llamador: PID (getpid), PPID (getppid), GID (getpgrp), grupos de acceso (getgroups), directorio de trabajo (chdir), directorio raiz (chroot), recursos utilizados (getrusage), intervalos de timers (getitimer), límites de recursos (getrlimit), máscara de modo de fichero (umask) y máscara de señal (sigvec y sigsetmask).

ERRORES:

[ENOTDIR] A component of the path prefix is not a directory.
[ENAMETOOLONG] A component of a pathname exceeded 255 characters, or an entire path name exceeded 1023 characters.
[ENOEXEC] The new process file has the appropriate access permission, but has an invalid magic number in its header.
[ETXTBSY] The new process file is a pure procedure (shared text) file that is currently open for writing or reading by some process.
[ENOMEM] The new process requires more virtual memory than is allowed by the imposed maximum (getrlimit(2)).
[E2BIG] The number of bytes in the new process' argument list is larger than the system-imposed limit. This limit is specified by the sysctl(3) MIB variable KERN_ARGMAX.
[EFAULT] The new process file is not as long as indicated by the size values in its header.
[EFAULT] Path, argv, or envp point to an illegal address.
[EIO] An I/O error occurred while reading from the file system.

</td>
</tr>
</table>

Ejemplos

Ejemplo c018.c (son dos ficheros):

Con el siguiente programa no sólo creamos un nuevo proceso sino que vamos a ver como ejecutarlo.

Aunque fork() es una herramienta muy poderosa, su acción suele combinarse en la mayoría de las ocasiones con otra llamada al sistema: execl(). Si fork() era la única manera de crear procesos en UNIX, execl() es la única manera de ejecutarlos.

La llamada execl() deber llevar como argumentos mínimos el PATH (camino absoluto o relativo) del proceso hijo y el nombre del proceso hijo. A partir de ahí se le pueden dar todos los argumentos que se deseen, siempre que la lista termine en un NULL.

De esta forma, con esta llamada se sustituye en la copia del proceso primitivo su segmento de datos y de instrucciones con lo que los procesos dejan de ser iguales y se puede empezar a hablar de creación de un verdadero proceso hijo diferente del padre.

El programa padre creará y ejecutará un proceso hijo, cogiendo su código desde otro fichero.

/* Programa que crea un proceso hijo y lo ejecuta con exec1()*/

Fichero c018_padre.c

```
#incluye<stdio.h>
main() {
int pid; /*En pid almacenaremos el nº deidentificación de proceso devuelto por fork()*/
pid=fork(); /*fork() crea un clon del proceso primitivo*/
switch(pid) {
  case -1: /*fork devuelve -1 en caso de error*/
      printf("\nNo se puede crear proceso hijo");
      exit(0); /* salimos al S.O.*/
  case 0: /*el valor 0 es asignado al hijo, mientras que el */
      /* proceso padre recibe el número de identificación del hijo*/
      printf("\n número de identificación del proceso HIJO%d",pid);
      execl("./hijol","hijol",NULL); /* ejecutamos el hijo */
  default:
      printf("\n número de identificación del PADRE %d",pid);
      sleep(1); /*El programa espera un segundo y termina */
  }
}
```

Fichero c018_hijo.c

Es ejecutado desde PADRE1. Lo único que hace es generar una salida por pantalla

/* Programa que es ejecutado desde el padre para mostrar la cooperación entre fork() execl()*/

```
#incluye<stdio.h>
main() {
   printf ("\n\n ------HIJO-----\n\n");
}
```

Ejemplo c019.c:

Es recomendable comprobar siempre que se haya podido crear el proceso hijo después de realizar la llamada al fork(). Esto ocurre cuando el valor devuelto por fork() es -1 (< 0 en general).

Por ello, la estructura típica cada vez que utilicemos la llamada fork() será la siguiente:

```
// ===========================================
// Programa: Ejemplo de fork
// Archivo : fork0.c  o c019.c
// ===========================================
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

// Función para imprimir información sobre un proceso
void PrintInfo (pid_t pid) {
  printf ("PID=%d PPID=%d\r\n",getpid(),getppid() );
  printf ("UID=%d GID=%d\r\n",getuid(),getgid() );
  printf ("EUID=%d EGID=%d\r\n",geteuid(), getegid() );
  printf ("PGRP=%d\r\n",getpgrp() );
  }

int main() {
pid_t pid;

pid = fork(); // Creamos proceso hijo
printf ("Valor retornado por la llamada fork: %d\n",pid);
switch (pid) {
  case -1: // Se ha producido error
       printf ("No se ha podido crear el proceso hijo\n");
       exit(1); // Salimos indicando la situación
  case 0: // Estamos en el proceso hijo
       printf ("\n** PROCESO HIJO **\n");
       // Utilizamos getppid() para obtener el PID del padre
       //printf ("\nEl PID del proceso padre es: %d", getppid());
       // Utilizamos getpid para obtener el PID del hijo (actual)
       //printf ("\nEl PID del proceso hijo es: %d",getpid());
       PrintInfo (pid);
       /* ... */
       break;
   default: /* Estamos en el proceso padre: pid>0 */
       printf ("\r\n** PROCESO PADRE **\r\n");
       // Utilizamos getpid() para obtener el PID del padre
       // printf ("\r\nEl PID del proceso padre es: %d",getpid());
       // El PID del hijo será lo que devolvió fork()
       // printf ("\r\nEl PID del proceso hijo es: %d",pid);
       PrintInfo (pid);
       /* ... */
   }
}
```

Ejemplo c020.c:

```
// ===========================================
// Programa: Ejemplo de fork
// Archivo : fork1.c o c020.c
// ===========================================
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define MAX_COUNT 200
#define BUF_SIZE 100

int main(void) {
pid_t pid;
int i;
char buf[BUF_SIZE];

fork();
pid = getpid();

for (i = 1; i <= MAX_COUNT; i++) {
    sprintf(buf, "Esta línea la ha imprimido PID: %d, Valor=%d\n", pid, i);
    write(1, buf, strlen(buf));
    }
}
```

La razón para utilizar write en vez de printf es porque printf utiliza un buffer, es decir, printf agrupa todas sus salidas para mostrarlas de golpe.

Puesto que la información no se envía a la pantalla inmediatamente, podriamos obtener los resultados en un orden diferente o mezcladas.

Para evitar este problema utilizaremos la función write que no utiliza buffer.

La salida de este programa será como la siguiente:

```
................
Esta línea la ha imprimido PID: 3456, Valor=13
Esta línea la ha imprimido PID: 3456, Valor=14
................
Esta línea la ha imprimido PID: 3456, Valor=20
Esta línea la ha imprimido PID: 4617, Valor=100
Esta línea la ha imprimido PID: 4617, Valor=101
................
Esta línea la ha imprimido PID: 3456, Valor=21
Esta línea la ha imprimido PID: 3456, Valor=22
................
```

Debido a que estos procesos se ejecutan de forma concurrente, sus líneas de salida se entremezclan de forma aleatoria, sin orden predecible. El orden lo determina el planificador de CPU. Cada vez que se ejecuta este programa nos dará unos resultados diferentes.

Ejemplo c021.c:

Consideremos ahora un ejemplo más simple, el cual distingue entre el padre y el hijo. En este programa, ambos procesos imprimen líneas que indican:

- Si la línea la ha imprimido el proceso hijo o el padre.
- El valor de la variable i.

Cuando el programa principal ejecuta el fork(), se crea una copia idéntica del espacio de direcciones, incluyendo el programa y todos sus datos.

La llamada al sistema "fork()" retorna el PID del proceso hijo al padre y 0 al proceso hijo.

En este ejemplo utilizamos printf por simplicidad.

```
// ===========================================
// Programa: Ejemplo de fork
// Archivo : fork2.c o c021.c
// ===========================================
#include <stdio.h>
#include <sys/types.h>
#define MAX_COUNT 200

// Prototipos de los procesos padre e hijo
void ProcesoHijo (void);
void ProcesoPadre (void);

int main (void) {
pid_t pid;
pid = fork();
if (pid == 0)
    ProcesoHijo();
else
    ProcesoPadre();
}

void ProcesoHijo (void) {
int i;
for (i=1; i<=MAX_COUNT; i++)
    printf("Esta linea la ha imprimido el proceso HIJO, Value=%d\n", i);
printf(" *** El proceso hijo ha terminado ***\n");
}

void ProcesoPadre (void) {
int i;
for (i=1; i<=MAX_COUNT; i++)
    printf("Esta linea la ha imprimido el proceso PADRE, Value=%d\n", i);
printf("*** El proceso padre ha terminado ***\n");
}
```

En el proceso padre, como pid!=0, se ejecutará la función "ProcesoPadre()". Por otro lado, en el proceso hijo pid=0 y se ejecutará "ProcesoHijo()".

Debido al hecho de que el planificador de la CPU asigna un quantum de tiempo a cada proceso, ambos procesos se ejecutarán durante algún tiempo antes de que el control cambie al otro proceso, y el proceso en ejecución imprimirá algunas líneas antes de que se puedan ver las líneas imprimidas por el otro proceso.

Por tanto, el valor de MAX_COUNT debería ser suficientemente grande como para los procesos se ejecuten durante al menos 2 o más quantums. Si el valor de MAX_COUNT es demasiado pequeño y termina en menos de un quantum, veremos dos grupos de líneas, cada uno de los cuales contendrá todas las líneas imprimidas por el mismo proceso.

Ejemplo c022.c:

En el ejemplo anterior podría darse el caso de que el proceso padre termine antes que el hijo.

Para evitar esta situación, podemos poner un "wait()" justo después de llamar a la función ProcesoPadre().

En el caso de que el proceso hijo termine (exit) antes de que el padre haya ejecutado el wait, el proceso hijo se queda en un estado denominado "zombie", hasta que el padre ejecute wait, momento en que se destruye el hijo.

Otra posible situación sería un proceso Abuelo que crea otro proceso Hijo, el cual crea otro proceso Nieto. Supongamos que el proceso Abuelo ejecuta wait(). Si el proceso Hijo termina (exit), el proceso Nieto se queda huérfano, con lo que su padre pasaría a ser el proceso INIT (el sistema operativo).

Para que el sistema operativo pueda destruir a todos los procesos huérfanos, el proceso INIT ejecuta un bucle infinito con la sentencia wait(), para que esperar a todos los procesos huérfanos que vaya adoptando y destruirlos cuando terminen su ejecución.

```
void main(void) {
pid_t pid;
pid = fork();
if (pid == 0)
   ProcesoHijo();
else {
   ProcesoPadre();
   // Si no nos interesa el valor del estado debemos poner el puntero nulo NULL
   wait(NULL);
   }
}
```

Ejemplo c023.c:

Este programa muestra algunas técnicas típicas de programación de procesos.

El programa principal crea dos procesos hijos para ejecutar el mismo bucle de impresión y mostrar un mensaje antes de salir. Para el proceso padre (p.e. el programa principal), después de crear dos procesos hijos entra en un proceso de espera ejecutando la llamada al sistema wait(). En el momento en que alguno de los dos procesos hijos termina, el padre comienza la ejecución y el PID del proceso que ha terminado se retorna al padre, el cual se puede imprimir.

Puesto que hay dos procesos hijos, habrán dos wait()s, uno para cada proceso hijo.

En este ejemplo no utilizamos el valor de "estado" retornado.

```
// ===========================================
// Programa: Ejemplo de fork
// Archivo : fork4.c o c023.c
// ===========================================
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define MAX_COUNT 200
#define BUF_SIZE 100
// Prototipo del proceso hijo
void ProcesoHijo(char *, char *);

int main (void) {
pid_t pid1, pid2, pid;
int status;
int i;
char buf[BUF_SIZE];

printf("*** El proceso padre realiza el fork() - 1 ***\n");
pid1 = fork();
if (pid1 < 0) {
   printf("Fallo al crear proceso 1\n");
   exit(1);
   }
if (pid1 == 0) {
   ProcesoHijo("Primero", " ");
   }

printf("*** El proceso padre realiza el fork() - 2 ***\n");
pid2 = fork();
if (pid2 < 0) {
   printf("Fallo al crear proceso 2\n");
   exit(1);
   }
if (pid2 == 0) {
   ProcesoHijo("Segundo", " ");
   }
sprintf(buf, "*** El padre entra en estado de espera .....\n");
write(1, buf, strlen(buf));
pid = wait(&status);
sprintf(buf, "*** El padre ha detectado que el proceso hijo %d ha terminado ***\n", pid);
write(1, buf, strlen(buf));
pid = wait(&status);
printf("*** El padre ha detectado que el proceso hijo %d ha terminado ***\n", pid);
printf("*** El proceso padre termina ***\n");
exit(0);
}

void ProcesoHijo(char *number, char *space) {
pid_t pid;
int i;
char buf[BUF_SIZE];
pid = getpid();
sprintf(buf, "%sEl proceso hijo %s comienza (pid = %d)\n", space, number, pid);
write(1, buf, strlen(buf));
for (i=1; i<=MAX_COUNT; i++) {
    sprintf(buf, "%sSalida del proceso hijo %s, Valor=%d\n", space, number, i);
    write(1, buf, strlen(buf));
    }
sprintf(buf, "%sEl proceso hijo %s (pid = %d) va a terminar\n", space, number, pid);
write(1, buf, strlen(buf));
exit(0);
}
```

Ejemplo c024.c:

En este ejemplo, veremos que el padre no tiene que esperar inmediatamente después de crear los dos hijos.

Puede realizar otras tareas.

```
// ===========================================
// Programa: Ejemplo de fork
// Archivo : fork5.c o c024.c
// ===========================================
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define MAX_COUNT 200
#define BUF_SIZE 100
#define QUAD(x) (x*x*x*x)

// Prototipos de los procesos padre e hijo
void ProcesoHijo (char *, char *);
void ProcesoPadre (void);

int main (void) {
pid_t pid1, pid2, pid;
int status;
int i;
char buf[BUF_SIZE];

printf("*** El proceso padre realiza el fork() - 1 ***\n");
pid1 = fork();
if (pid1 < 0) {
   printf("Fallo al crear proceso 1\n");
   exit(1);
   }
if (pid1 == 0) {
   ProcesoHijo("Primero", " ");
   }

printf("*** El proceso padre realiza el fork() - 2 ***\n");
pid2 = fork();
if (pid2 < 0) {
   printf("Fallo al crear proceso 2\n");
   exit(1);
   }
if (pid2 == 0) {
   ProcesoHijo("Segundo", " ");
   }

// El proceso padre se pone a trabajar
ProcesoPadre();
sprintf(buf, "*** El padre entra en estado de espera .....\n");
write(1, buf, strlen(buf));

pid = wait(&status);
sprintf(buf, "*** El padre ha detectado que el proceso hijo %d ha terminado ***\n", pid);
write(1, buf, strlen(buf));

pid = wait(&status);
printf("*** El padre ha detectado que el proceso hijo %d ha terminado ***\n", pid);
printf("*** El proceso padre termina ***\n");
exit(0);
}

void ProcesoPadre (void) {
int a, b, c, d;
int abcd, a4b4c4d4;
int count = 0;
char buf[BUF_SIZE];
sprintf(buf, "El padre calcula números de Armstrong\n");
write(1, buf, strlen(buf));
for (a = 0; a <= 9; a++)
 for (b = 0; b <= 9; b++)
   for (c = 0; c <= 9; c++)
    for (d = 0; d <= 9; d++) {
      abcd = a*1000 + b*100 + c*10 + d;
      a4b4c4d4 = QUAD(a) + QUAD(b) + QUAD(c) + QUAD(d);
      if (abcd == a4b4c4d4) {
       sprintf(buf, "Desde el padre: El %d-esimo número de Armstrong es %d\n", ++count, abcd);
       write(1, buf, strlen(buf));
       }
```

```
    }
sprintf(buf, "Desde el padre: hay %d números de Armstrong\n", count);
write(1, buf, strlen(buf));
}

void ProcesoHijo(char *number, char *space) {
pid_t pid;
int i;
char buf[BUF_SIZE];
pid = getpid();

sprintf(buf, "%sEl proceso hijo %s comienza (pid = %d)\n",space, number, pid);
write(1, buf, strlen(buf));
for (i=1; i<=MAX_COUNT; i++) {
  sprintf(buf, "%sSalida del proceso hijo %s, Valor=%d\n", space, number, i);
  write(1, buf, strlen(buf));
  }
sprintf(buf, "%sEl proceso hijo %s (pid = %d) va a terminar\n", space, number, pid);
write(1, buf, strlen(buf));
exit(0);
}
```

El programa principal crea dos procesos hijos. Ambos procesos llaman a ProcesoHijo(). El proceso padre llama a la función ProcesoHijo(). Esta función calcula todos los números de Armstrong desde el 0 hasta el 9999. Un número de Armstrong es un entero cuyo valor es igual que la suma de sus dígitos elevado a 4. Después de esto el proceso padre entra en un estado de espera, hasta que termine alguno de sus hijos. Puesto que los dos hijos se ejecutan concurrentemente, no tenemos forma de predecir cual de ellos terminará primero.

ATENCION: aunque teóricamente podemos crear tantos procesos como queramos, los sistemas siempre tienen límites. Por tanto, **siempre debemos comprobar si el valor retornado por fork() es negativo, para notificar el error al programador.** Si esto ocurriera, deberemos intentar reducir el número de procesos hijo o reorganizar el programa. Si el valor retornado PID no es importante, entonces podemos tratar la función wait() como un procedimiento. El siguiente código es una modificación al anterior (algunas líneas de la función main()).

```
sprintf(buf, "*** El padre entra en estado de espera .....\n");
write(1, buf, strlen(buf));

wait(&status);
sprintf(buf, "*** El padre ha detectado que un proceso hijo ha terminado ***\n");
write(1, buf, strlen(buf));

wait(&status);
printf("*** El padre ha detectado que un proceso hijo ha terminado ***\n");
printf("*** El proceso padre termina ***\n");
exit(0);
```

Ejemplo c025.c :

```
#include <stdio.h>
#include <stdlib.h>
int main() {
unsigned int status;
if (fork()==0) { /* ==0 en el hijo */
   // Pedimos por teclado
   printf("Proceso Hijo: Introduce el estado de terminación : ");
   scanf("%d",&status);
   // El proceso hijo termina
   exit(status);
   }
else { /* !=0 en el padre */
   // El padre se queda esperando a que el proceso hijo termine
   wait(status);
   // Cuando el hijo acaba, imprimimos el valor de estado retornado por éste
   printf("Estado de salida del hijo: %d\r\n", status >> 8);
   }
}
```

Ejemplo c026.c:

Observar el funcionamiento de sleep.

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("Hola mundo!!\r\n");
  sleep(10);
  printf("He dormido 10 segundos!\n");
  return 0;
}
```

Ejemplo c027.c:

¿Cual es la salida en este ejemplo?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("Parent PID=%d\n",getpid() );
  if (fork()==0) {
     printf ("PPID=%d\n",getppid() );
     return 0;
     }
  return 0;
}
```

Ejemplo c028.c:

Y en este, ¿cual es la salida?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("Parent PID=%d\n",getpid() );
  if (fork()==0) {
     sleep (2);
     printf ("PPID=%d\n",getppid() );
     return 0;
     }
  return 0;
}
```

Ejemplo c029.c:

Y en este, ¿cual es la salida?

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
  printf("Parent PID=%d\n",getpid() );
  if (fork()==0) {
     printf ("PPID=%d\n",getppid() );
     return 0;
     }
  sleep (2);
  return 0;
}
```

¿Que conclusión se obtiene de estas ejecuciones?

Ejemplo c030.c:

¿Cual es la salida del siguiente ejemplo?

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
void ChildFun(int *ptr) {
  *ptr=100;
  exit (0); // stdlib.h
}

int main(void) {
  int ret;
  int p=10;
  if ((ret=fork()) == -1) {
     perror("fork failed");
     return 1;
     }

  if (ret==0) {
     ChildFun(&p);
     }

  // Codigo del padre
  wait (&ret); // Espera la finalización del hijo
  printf ("p=%d\n", p);

  return 0;
}
```

Ejemplo c031.c:

Analizar con el siguiente ejemplo el significado de status.

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

int main(void) {
int ret,pid,status;

if ((ret=fork())==-1) {
   perror(0);
   return 1;
   }
if (ret==0) {
   printf("CHild Living...");
   exit(10);
   }

pid=wait(&status);
if (pid==-1) {
   printf("No Child\n");
   exit(0);
   }
printf ("Child PID(%d) Terminated\n",pid);
if ((status & 0xFF)==0) { // Got a legal status
   printf("Child Returned (%d)\n",status>>8);
   exit(0);
   }
if ((status & 0xFF00)==0) { // Process Terminated Signal !!
   printf("Signal terminated child - Signal %d\n",status & 0x7F);
   if((status & 0x0080)>0) {
      printf("Child Performed CORE Dump\n");
      }
   exit(0);
   }
if ( (status & 0xFF) == 0x7F) {
   printf("Signal Stopped Child for Debugging \n");
   printf("Signal Id (%d)",status>>8);
   exit(0);
   }
printf("**** This Line Should Never be executed ****");
return 0;
}
```

Ejemplo c032.c:

Uso de execve

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

int main(void) {
int ret;

char *args[] = {"ls",NULL};
printf("ls Is About to start....\n");
ret=execve("/bin/ls",args,NULL);

perror("exec failed");
return 0;
}
```

Ejemplo c033.c:

Con paso de argumentos.

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

int main(void) {
int ret;

char *args[] = {"ls","-l","/usr/bin",NULL}; // Long Listing
printf("ls Is About to start....\n");
ret=execve("/bin/ls",args,NULL);

perror("exec failed");
return 0;
}
```

Ejemplo c034.c:

¿Cual es la salida del siguiente programa? (El shell funciona de la misma manera.)

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

int main(void) {
int ret;

char *args[] = {"ls","-l","a*",NULL};
printf("ls Is About to start....\n");
ret=execve("/bin/ls",args,NULL);

perror("exec failed");
return 0;
}
```

Ejemplo c035.c:

Con paso de variables de ambiente:

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

int main(void) {
int ret;

char *args[]={"which","ls",NULL};
char *env[]={"PATH=/etc:/tmp:/usr/bin:/bin:.","MAIL=/home/user1",NULL};
ret = execve("/usr/bin/which",args,env);
perror("exec failed");
return 0;
}
```

Ejemplo c036.c:

Pasar el mismo bloque de ambiente.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
extern char **environ;

int main(void) {
  char *args[]={"which","ls",NULL};
  execve("/usr/bin/which",args,environ);
  perror("exec failed"); return 0;
}
```

Ejemplo c037.c:

Fork y execve

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

int main(void) {
int ret;
if ((ret=fork())==-1) {
   perror("Fork Failed");
   return 1;
   }
if (ret==0) { // child
   char *args[]={"/bin/dir","-l",NULL};
   execve("/bin/ls",args,NULL);
   perror("Exec Failed");
   return 1;
   }
wait(&ret); // wait for child
return 0;
}
```

Ejemplo c038.c:

```
#include <stdio.h>
#include <sys/types.h>
void parse(char *line, char **argv) {
  while (*line != '\0') { /* if not the end of line ....... */
    while (*line == ' ' || *line == '\t' || *line == '\n')
      *line++ = '\0'; /* replace white spaces with 0 */
    *argv++ = line; /* save the argument position */
    while (*line != '\0' && *line != ' ' && *line != '\t' && *line != '\n')
      line++; /* skip the argument until ... */
    }
  *argv = '\0'; /* mark the end of argument list */
}

void execute(char **argv) {
pid_t pid;
int status;
if ((pid = fork()) < 0) { /* fork a child process */
  printf("*** ERROR: forking child process failed\n");
  exit(1);
  }
else
  if (pid == 0) { /* for the child process: */
      if (execvp(*argv, argv) < 0) { /* execute the command */
         printf("*** ERROR: exec failed\n");
         exit(1);
         }
      }
else { /* for the parent: */
  while (wait(&status) != pid) /* wait for completion */
     ;
  }
}

void main(void) {
char line[1024]; /* the input line */
char *argv[64]; /* the command line argument */
while (1) { /* repeat until done .... */
  printf("Shell -> "); /* display a prompt */
  gets(line); /* read in the command line */
  printf("\n");
  parse(line, argv); /* parse the line */
  if (strcmp(argv[0], "exit") == 0) /* is it an "exit"? */
     exit(0); /* exit if it is */
  execute(argv); /* otherwise, execute the command */
  }
}
```

La función parse() toma como entrada una línea y retorna un array de cadenas de caracteres (punteros a char). Esta función salta hasta encontrar el carácter fin de cadena \0. Si el carácter actual no es \0, la función se salta todos los espacios en blanco y los sustituye por \0, para que la cadena esté bien terminada. En el momento en que se encuentre cualquier cosa distinta del espacio en blanco, almacenamos dicha posición en la posición actual de argv y su índice avanza. El proceso se repite hasta que se alcanza el carácter \0, colocando 0 en argv[i].

Por ejemplo, si la línea de entrada es la siguiente cadena:

```
"cp abc.CC xyz.TT"
```

La función parse() retornará un array argv[] con el siguiente contenido:

```
argv[0]="cp"
argv[1]="abc.CC"
argv[2]="xyz.TT"
```

La función execute() toma un array argv[], tratado como argumentos desde la línea de comandos, con el nombre del programa en argv[0], realiza un fork para crear un proceso hijo y ejecuta el programa indicado en el proceso hijo. Mientras que el proceso hijo está ejecutando el comando, el proceso padre ejecuta un wait(), esperando que termine el hijo. En este caso, el proceso padre conoce el PID del proceso hijo y espera hasta que termine dicho proceso.

El programa principal es muy simple. Muestra un prompt de comando, lee una línea de comando, la analiza (parse) y determina si el nombre es "exit". Si es "exit", ejecuta la llamada "exit()" para terminar la ejecución del programa; en otro caso se ejecuta el comando (execute).

Esto mismo podemos utilizar para ejecutar un programa nuestro, el cual se ejecutará como proceso hijo.

Por ejemplo, podemos escribir un programa "hijo.c" y compilarlo como "hijo". Este fichero lo ejecutaremos en la parte correspondiente al código del padre de la siguiente forma:

execl("./hijo", "hijo", NULL);

```
#include <stdio.h>
int main() {
int i;
printf ("** Proceso HIJO **\r\n");
for (i=0; i<200; i++) {
  printf ("Valor i=%d\r\n", i);
  }
}
```

Ejemplo c039.c:

Un mini shell.

Primero veamos como sería un shell muy simple.

```
#include <sys/types.h>
while (TRUE) { /* Bucle infinito */
  type_prompt(); /* Mostrar símbolo sma. */
  read_command (command, parameters) /* leer entrada terminal */

  if (fork() != 0) { /* crea un proceso hijo */
     /* Código que sigue ejecutando el padre */
     waitpid(-1, &status, 0); /* espera fin del hijo */
     }
  else {
    /* Código que sigue ejecutando el hijo */
    execve(command,parameters,0); /* ejecuta comando */
    }
}
```

Ahora veamos un ejemplo del minishell. ¿Porqué no funcionan los backspaces?

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

int RunProgram(char *pathname); // =0 (success) ,=-1 fail

int main(void) {
char buf[300];
int ret; int i;

for(;;) {
  write(1,"msh $ ",6);
  ret = read(0,buf,sizeof(buf)-1);
  if (ret<=0) {
     write(2,"Error\n",6);
     continue;
     }

 // strip \r\n
  buf[ret]=0;
  for (i=0; buf[i]!=0;++i) {
      if (buf[i]=='\r' || buf[i]=='\n') {
         buf[i]=0; break;
         }
      }

  // Run Program
  if (strcmp(buf,"exit")==0) {
     break;
     }
  if (RunProgram(buf)==-1) {
     write(2,"Error\n",6);
     continue;
     }
  }
  return 0;
}

int RunProgram(char *filename) {
int ret;
char *args[2] = {filename,NULL}; // What does this statement do?
extern char **environ; // declaration of external
if ((ret=fork())==-1) {
   write(2,"fork fail\n",10);
   return -1;
   }
if (ret==0) {
   execve(filename,args,NULL); // exec program
   _exit(1); // An error
   }
if (wait(&ret)==-1) {
   return -1;
   } // If is to remind you Important code
if ((ret & 0xFF)==0 && (ret>>8)==0) {
   return 0;
   }
return -1;
}
```