

6. CREACIÓN DE PROCESOS

Una vez visto el concepto de un proceso y su estructura, abordaremos el estudio de las facilidades que nos brinda UNIX para manipular y gestionar los procesos.

Lo primero que vamos a ver es la forma de invocar un programa desde otro. Esto lo vamos a conseguir con la llamada `exec`.

6.1 EJECUCIÓN DE PROGRAMAS MEDIANTE EXEC

Existe toda una familia de funciones `exec` que podemos usar para ejecutar programas. Dentro de esta familia, cada función tiene su interfaz propia, pero todas tienen aspectos comunes y obedecen al mismo tipo de funcionamiento.

Básicamente, el resultado que se consigue con estas funciones es cargar un programa en la zona de memoria del proceso que ejecuta la llamada, sobrescribiendo los segmentos del programa antiguo con los del nuevo.

El contenido del contexto del nivel de usuario del proceso que llama a `exec` deja de ser accesible y es reemplazado de acuerdo con el nuevo programa. Es decir, el programa viejo es sustituido por el nuevo y nunca retornaremos a él para proseguir con su ejecución, ya que es el programa nuevo el que se ejecutará.

La declaración de la familia de funciones `exec` es la siguiente:

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ... , (char *) 0);
int execv(const char *path, char *const argv[]);
int execlp(const char *path, const char *arg0, ..., (char *) 0,
           char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., (char *) 0);
int execvp(const char *file, char *const argv[]);
```

En todas estas funciones, `path` apunta a la ruta (absoluta o relativa) de un archivo ejecutable.

`file` apunta al nombre de un archivo ejecutable. La ruta del archivo se construye buscando el archivo en los directorios que se indican en la variable de entorno `PATH`. Tanto `path` como `file` se refieren a archivos ejecutables o a archivos de datos (scripts de shells) para un intérprete de órdenes. Si el archivo no tiene un número mágico que lo identifique como directamente ejecutable, se le pasa a `/bin/sh` como un archivo de órdenes para que lo interprete.

`arg0`, `arg1`, ..., `argn` son apuntadores a cadenas de caracteres y constituyen la lista de argumentos que se le pasa al nuevo programa. Por convenio, al menos `arg0` está presente siempre y apuntando a una cadena idéntica a `path` o al último componente de `path`. Hay que resaltar que a continuación de `argn` pasamos un apuntador a `NULL` (`(char *) 0`) para indicar el final de los argumentos.

`argv` es un arreglo de cadenas de caracteres que constituyen la lista de argumentos que va a recibir nuestro programa. Por convenio, `argv` debe tener al menos un elemento, que debe apuntar a una cadena

idéntica a `path` o al último componente de `path`. El final de `argv` se indica haciendo que a continuación de su último elemento significativo haya un apuntador a `NULL`.

`envp` es un arreglo de apuntadores a cadenas de caracteres que constituyen el entorno en el que se va a ejecutar el nuevo programa. `envp` también termina con un apuntador a `NULL`.

Los argumentos para estas 6 funciones `exec` son difíciles de recordar. Las letras en el nombre de la función nos pueden ayudar en algo. La letra "p" significa que la función toma el argumento `file` y usa la variable de entorno `PATH` para encontrar el archivo ejecutable. La letra "l" indica que la función toma una lista de argumentos y es mutuamente exclusiva con la letra "v" que toma `argv`. Finalmente, la letra "e" significa que la función toma el arreglo `envp` en vez de usar las variables de entorno.

Si el nuevo programa que ejecutará se encuentra escrito en C, entonces recibe los parámetros `arg0`, `argv1`, ..., `argn` o `argv` y `envp` a través de la función principal `main`. Esta función se puede declarar de la forma siguiente:

```
main(int argc, char *argv[], char *envp[]);
```

Donde `argc` es el total de argumentos que recibe el programa (total de elementos de `argv`). `argv` es un arreglo de apuntadores a cada uno de los argumentos y `envp` tiene el mismo significado que hemos descrito antes. Si, por ejemplo, invocamos a un programa C mediante la siguiente línea.

```
$ copiar archivo1 archivo2
```

`argc` va a tomar el valor 3 y los contenidos de `argv` serán `argv[0] = "copiar"`, `argv[1] = "archivo1"`, `argv[2] = "archivo2"`. Como vemos, `argv[0]` contiene el nombre del programa. Esto es así porque el shell se encarga de llamar a `exec` con `arg0 = "copiar"` o `argv[0] = "copiar"`.

Si `exec` devuelve el control al programa que la invoca, es porque no se ha ejecutado correctamente. En este caso devuelve el valor -1 y en `errno` estará el código del tipo de error producido.

Básicamente, todo programa consta de una cabecera principal y una serie de secciones, cada una de las cuales se compone de una cabecera y una zona de datos. En la siguiente figura podemos ver la estructura lógica de todo programa. Los programas son generados por el enlazador (programa `ld`) y sus partes son:

- La cabecera principal, que contiene el total de secciones del programa, la dirección de inicio de ejecución y el número mágico del programa que identifica qué tipo de archivo es.
- Cabeceras de sección que describen cada una de las secciones del archivo. Contiene el tamaño de la sección, el total de direcciones virtuales que va a ocupar durante su ejecución y otra información.
- Secciones que van a contener el código del programa y variables globales.
- Secciones que van a contener tablas de símbolos, información para el depurador, etc.

Cabecera principal

Cabecera de la sección 1

Cabecera de la sección 2

Sección 1

Sección n

Número mágico Número de secciones Valores iniciales de los registros
Tipo de sección Tamaño de la sección Espacio de direcciones virtuales
Tipo de sección Tamaño de la sección Espacio de direcciones virtuales
...
Datos de la sección
...
Datos de la sección
Otra información

Como ejemplo de aplicación de dos de las funciones de la familia `exec`, vamos a escribir un programa que consulte el estado de un archivo y que ejecute una acción determinada cuando el archivo sufra su última modificación. La forma de invocar este programa será:

```
$ esperar nombre_archivo [-t tiempo] [orden]
```

El programa lee el estado del archivo `nombre_archivo` y espera el tiempo indicado por `timeout`; si el archivo no ha sufrido cambios, entonces ejecuta la secuencia de instrucciones indicada en `orden`; en caso contrario, vuelve a esperar otro `timeout` segundos. Si no especificamos ningún valor para `timeout`, se deben tomar 60 segundos por defecto.

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>

int main(int argc, char *argv[]) {
    int fd, timeout;
    struct stat buf;
    time_t ultima_fecha = 0;
    char **orden;

    if (argc < 2) {
        fprintf(stderr, "forma de uso: %s nombre_archivo [-t timeout]
[orden]\n", argv[0]);
        return -1;
    }
}
```

```

if ((fd = open(argv[1], O_RDONLY)) == -1) {
    perror(argv[1]);
    return -1;
}

if (strcmp(argv[2], "-t") == 0) {
    timeout = atoi(argv[3]);
} else {
    timeout = 60;
}

if (argc == 3) {
    orden = &argv[2];
} else if (argc >= 5) {
    orden = &argv[4];
}

fstat(fd, &buf);
while (buf.st_mtime != ultima_fecha) {
    ultima_fecha = buf.st_mtime;
    fprintf(stdout, "durmiendo\n");
    sleep(timeout);
    fprintf(stdout, "despertando\n");
    fstat(fd, &buf);
}
execvp(*orden, orden);
perror(argv[0]);
}

```

6.2 CREACIÓN DE PROCESOS – FORK

La única forma de crear un proceso en el sistema UNIX es mediante la llamada `fork`. El proceso que invoca a `fork` se llama proceso padre y el proceso creado es el proceso hijo. La declaración de `fork` es la siguiente:

```

#include <sys/types.h>
pid_t fork();

```

Y la forma de invocarla es `pid = fork()`.

La llamada a `fork` hace que el proceso actual se duplique. A la salida de `fork`, los dos procesos tienen una copia idéntica del contexto del nivel de usuario excepto el valor de `pid`, que para el proceso padre toma el valor del PID del proceso hijo y para el hijo toma el valor de 0.

La razón de por qué el PID del hijo es regresado a su padre es debido a que un proceso padre puede tener más de un hijo y no existe ninguna llamada que permita a un proceso padre obtener, posteriormente, el PID de sus hijos. Por otro lado, `fork` regresa 0 al proceso hijo debido a que un proceso sólo puede tener

un padre, y cualquier proceso hijo siempre puede obtener el PID del padre a través de la llamada `getppid`. (El PID 0 es reservado para uso exclusivo del kernel).

Si la llamada a `fork` falla, devolverá el valor de -1 y en `errno` estará el código del error producido.

Una secuencia de código típica para manejar la llamada a `fork` es la siguiente:

```
int pid;
...
if ((pid = fork()) == -1) {
    perror("fork");
} else if (pid == 0) {
    /* código que va a ejecutar el proceso hijo. */
} else {
    /* código que va a ejecutar el proceso padre. */
}
```

Como primer ejemplo de uso de `fork`, vamos a ver un programa que crea un proceso hijo y donde tanto el proceso como el hijo van a estar escribiendo por pantalla:

Soy el proceso PADRE
Soy el proceso HIJO

El código sería:

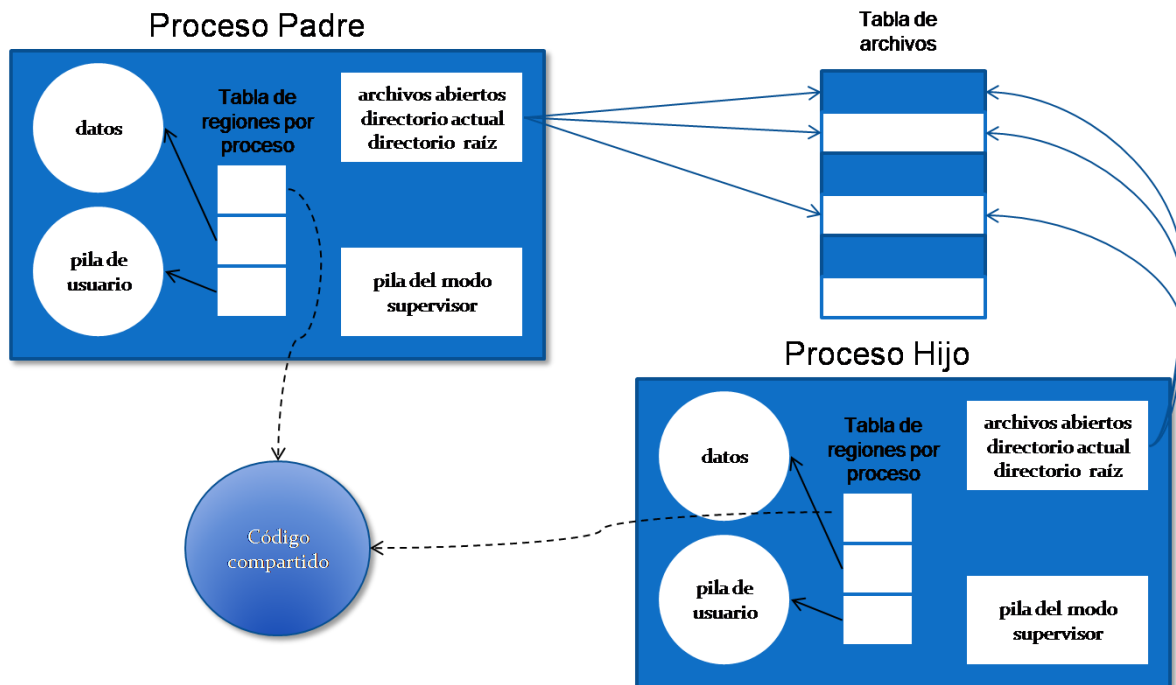
```
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int i = 0;
    switch( fork() ) {
        case -1:
            perror("fork");
            return -1;
        case 0:
            while (i < 100) {
                fprintf(stdout, "\t\tSoy el proceso hijo: %d\n",
i++);
            }
            break;
        default:
            while (i < 100) {
                fprintf(stdout, "\t\tSoy el proceso padre: %d\n",
i++);
            }
            break;
    }
    return EXIT_SUCCESS;
}
```

Cuando llamamos a `fork`, el kernel realiza las siguientes operaciones:

1. Busca una entrada libre en la tabla de procesos y la reserva para el proceso hijo.
2. Asigna un identificador de proceso para el proceso hijo. Este número es único e invariable durante toda la vida del proceso y es la clave para poder controlarlo desde otros procesos.
3. Realiza una copia del contexto del nivel de usuario del proceso padre para el proceso hijo. Las secciones que deban ser compartidas, como el código o las zonas de memoria compartida, no se copian, sino que se incrementan los contadores que indican cuántos procesos comparten esas zonas.
4. Las tablas de control de archivos locales al proceso, como puede ser la tabla de descriptores de archivo, también se copian del proceso padre al proceso hijo, ya que forman parte del contexto del nivel de usuario. En las tablas globales del kernel, tabla de archivos y tabla de nodos-i, se incrementan los contadores que indican cuántos procesos tienen abiertos esos archivos.
5. Retorna al proceso padre el `PID` del proceso hijo, y al proceso hijo le devuelve el valor de 0.

En la figura anterior, podemos ver el resultado de la ejecución de `fork`.



Es importante aclarar que ambos procesos (padre e hijo) comparten los mismos descriptores de archivos (con los mismos `offsets`). Sin ningún tipo de sincronización (una llamada `wait`) es posible que la salida de ambos procesos se mezcle. Por esto es necesario que los procesos se encarguen de manejar los descriptores. Existen dos formas de hacerlo:

- El padre espera hasta que el hijo haya terminado. En este caso, el padre no necesita hacer nada con los descriptores. Cuando el hijo termina, cualquiera de los descriptores compartidos que hayan sido usados por el hijo tendrán sus `offsets` actualizados de acuerdo a las operaciones realizadas.
- Tanto el proceso padre como el hijo siguen su propio camino. Después de la llamada `fork`, el proceso padre cierra aquellos descriptores que no necesita, al igual que el proceso hijo. De esta

manera, ninguno interfiere con los descriptores del otro. Este escenario es utilizado regularmente en servidores de red.

Es difícil que una llamada a `fork` falle, pero si lo hace puede ser porque:

- Ya existen demasiados procesos en el sistema, lo que usualmente significa que algo está mal.
- El número total de procesos para el `UID` real excede los límites del sistema (la constante `CHILD_MAX` especifica el número máximo simultáneo de procesos para un `ID` real de usuario).

Hay dos razones para usar la función `fork`:

- Cuando el proceso quiere duplicarse a sí mismo y así los procesos padre e hijo puedan cada uno ejecutar diferentes secciones del código al mismo tiempo. Esto es común para servidores de red en donde el proceso padre espera por una petición de servicio de un cliente. Cuando la petición llega, el padre ejecuta un `fork` y permite que el proceso hijo maneje la petición. El proceso padre vuelve a quedar esperando por otra petición.
- Cuando un proceso quiere ejecutar un programa diferente. Esto es común en intérpretes de shells. En este caso, el proceso hijo realiza un `exec` justo después de regresar del `fork`.

Algunos sistemas operativos combinan la operación de un `fork` seguido de un `exec` en una operación atómica llamada `spawn`.

6.2.1 LA FUNCIÓN `VFORK`

La función `vfork` tiene un funcionamiento similar y regresa los mismos valores que `fork`. Pero la semántica de las dos funciones difiere.

`vfork` intenta crear un nuevo proceso cuyo propósito es ejecutar un nuevo programa (`exec`). Un ejemplo del uso de `vfork` es el siguiente código:

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

int glob = 6;

int main(int argc, char *argv[]) {
    int var;
    pid_t pid;

    var = 88;
    fprintf(stdout, "before vfork\n");
    if ((pid = vfork()) < 0) {
        perror("vfork");
        return -1;
    } else if (pid == 0) {
        glob++;
        var++;
    }
```

```

        fprintf(stdout, "CHILD pid = %d, glob = %d, var = %d\n",
getpid(), glob, var);
        _exit(0);
    }
    fprintf(stdout, "pid = %d, glob = %d, var = %d\n", getpid(), glob,
var);
    exit(0);
}

```

La función `vfork` crea un nuevo proceso, justo como `fork`, pero sin copiar el espacio de direccionamiento (contexto) del proceso padre. En su lugar mientras el proceso hijo esté corriendo y hasta que ejecuta una llamada `exec` o `exit` el proceso corre en el espacio de direccionamiento del padre. Esta optimización provee eficiencia en las implementaciones de memoria virtual de algunos sistemas UNIX.

Otra diferencia entre las dos funciones es que `vfork` garantiza que el proceso hijo se ejecute primero, hasta que se haga una llamada `exec` o `exit`. Cuando el proceso hijo ejecuta cualquiera de estas funciones, el proceso padre retoma el control. (Esto pueda llevar a un `deadlock` si el proceso hijo depende de que el proceso padre realice algunas acciones antes de que se ejecuten las llamadas).

6.3 TERMINACIÓN DE PROCESOS – EXIT Y WAIT

Una situación muy típica en programación en UNIX es que cuando un proceso crea a otro, el proceso padre se quede esperando a que termine el hijo antes de continuar su ejecución. Un ejemplo de esta situación es la forma de operar de los intérpretes de órdenes. Cuando escribimos una orden, el intérprete arranca un proceso para ejecutar la orden y no devuelve el control hasta que no se ha ejecutado completamente. Naturalmente, esto no es aplicable cuando la orden se ejecuta en segundo plano.

Para poder sincronizar los procesos padre e hijo, se emplean las llamadas `exit` y `wait`. 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 estatus al sistema. Este valor lo podemos consultar a través de la variable de entorno `“?”`.

Un retorno efectuado desde la función principal (`main`) de un programa C tiene el mismo efecto que la llamada a `exit`. Si efectuamos el retorno sin devolver ningún valor en concreto, el resultado devuelto al sistema estará indefinido.

La llamada a `exit` tiene además, las siguientes, consecuencias:

- Las funciones registradas por `atexit` son invocadas en orden inverso a como fueron registradas. `atexit` permite indicarle al sistema qué acciones queremos que se ejecuten al producirse la terminación de un proceso.
- El contexto del proceso es descargado de memoria, lo que implica que la tabla de descriptores de archivos es cerrada y sus archivos asociados cerrados, si no quedan más procesos que los tengan abiertos.

- Si el proceso padre del que ejecuta la llamada a `exit` está ejecutando una llamada a `wait`, se le notifica de 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. Un proceso zombi sólo ocupa una entrada en la tabla de proceso del sistema y su contexto es descargado de memoria.

`exit` es uno de los pocos ejemplos de llamada que no devuelve ningún valor. Es lógico, ya que el proceso que la ejecuta deja de existir después de haberla ejecutado.

La declaración de `wait` es la siguiente:

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

`wait` suspende la ejecución del proceso que la invoca hasta que alguno de sus procesos hijos termina. La forma de invocar a `wait` es:

```
int pid, estado;
...
pid = wait(&estado);
/* otra cosa */
pid = wait(NULL);
```

PID es el identificador de alguno de los procesos hijos zombis. `estado` es la variable donde vamos a almacenar el valor que el proceso hijo le envía al proceso padre mediante la llamada a `exit` y que da idea de la condición de finalización del proceso hijo. Si queremos ignorar este valor, podemos pasarle a `wait` un apuntador `NULL`.

Puede ocurrir que el proceso hijo termine de forma anormal. Para estos campos hay definidas, en el archivo `<wait.h>`, unas macros que analizan el valor de estado para determinar la causa de terminación del proceso. Estas macros son:

Macros	Descripción
WIFEXITED (estado)	Devuelve verdadero (cualquier valor distinto de 0) cuando el proceso termina con una llamada a <code>exit</code> o a <code>_exit</code> .
WEXITSTATUS (estado)	Si <code>WIFEXITED</code> devuelve verdadero, esta macro devuelve el valor de 8 bits menos significativos que <code>exit</code> le pasa al proceso padre.
WIFSIGNALED (estado)	Devuelve verdadero cuando el proceso termina debido a la acción por defecto de alguna señal.
WTERMSIG (estado)	Si <code>WIFSIGNALED</code> devuelve verdadero, esta macro devuelve el número de la señal que ha causado la terminación del proceso.
WCOREDUMP (estado)	Devuelve verdadero si se ha generado un archivo con un volcado de la memoria (core image) del proceso.
WIFSTOPPED (estado)	Devuelve verdadero si el proceso está parado.
WSTOPSIG (estado)	Si <code>WIFSTOPPED</code> devuelve verdadero, esta macro devuelve el número de la señal que ha causado la parada del proceso.

Si durante la llamada a `wait` se produce algún error, la función devuelve -1 y en `errno` estará el código del tipo de error producido.

`exit` y `wait` se suelen usar para conseguir que el proceso padre espere a la terminación de su proceso hijo. Una secuencia de código para realizar esta operación puede ser:

```
int pid, estado;
...
if ((pid = fork()) == -1) {
    /* error en la creación del proceso hijo. */
} else if (pid == 0) {
    /* código del proceso hijo. */
    exit(10);
} else {
    /* código del proceso padre. */
    pid = wait(&estado);
    /* cuando el proceso hijo llame a "exit", le pasará al padre el
    valor 10, que éste puede recibir a través de "estado"
    (estado = 10). */
}
```

Como ejemplo de aplicación, vamos a codificar la función `system`, que es estándar en la biblioteca de funciones de C. Esta función se declara de la forma:

```
int system(const char *cadena);
```

Donde `cadena` es un apuntador a una cadena de caracteres que debe contener una orden para el intérprete de órdenes del sistema. Así, la llamada,

```
system("ls -al");
```

Despliega por pantalla el contenido del directorio actual. El código para esta función puede ser el siguiente:

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

int sistema(const char *orden) {
    int pid, tty, estado, w;

    fflush(stdout);
    if ((tty = open("/dev/tty", O_RDWR)) == -1) {
        perror("sistema");
        return -1;
    }
    /* código para el proceso hijo */
    if ((pid = fork()) == 0) {
        close(0); dup(tty);
        close(1); dup(tty);
        close(2); dup(tty);
```

```

        close(tty);
        execlp("sh", "sh", "-c", orden, NULL);
        exit(127);
    }
    /* código para el proceso padre */
    close (tty);
    while ((w = wait(&estado)) != pid && w != -1);
    if (w == -1) {
        estado = -1;
    }
    return estado;
}

int main(int argc, char *argv[]) {
    sistema("clear");
    sistema("ps -ef");
    sistema("ls -la");
}

```

6.3.1 FUNCIÓN WAITPID

Esta forma en que un proceso puede ser detenido es usando la llamada `waitpid`:

```

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

```

La función `waitpid` tiene, esencialmente, la misma funcionalidad que la llamada `wait`. Sus principales diferencias son:

- La función `wait` puede bloquear el proceso hasta que un proceso hijo termina, mientras que `waitpid` tiene una opción que lo previene de este bloqueo.
- La función `waitpid` no espera a que el proceso hijo termine; tiene un número de opciones que controla qué es lo que espera el proceso.

Como mencionamos antes, si nosotros tenemos más de un proceso hijo, `wait` regresa en la terminación de cualquiera de ellos. ¿Qué sucede si queremos esperar a que un determinado proceso termine (asumiendo que conocemos el ID de ese proceso)? En versiones anteriores de UNIX, nosotros podemos tener el resultado de la llamada `wait` y compararlo con el ID del proceso al cual estamos esperando. Si el proceso terminado no es el que estamos esperando, guardamos esa información (ID y estado de terminación) y volvemos a ejecutar `wait`. Continuaremos haciendo esto hasta que el proceso deseado termine. Esta funcionalidad está implementada en la función `waitpid`.

La interpretación del argumento `pid` que recibe `waitpid` depende de su valor:

PID	Descripción
<code>pid == 1</code>	Espera hasta que termine cualquiera de sus procesos hijos. Con este valor <code>waitpid</code> es igual a <code>wait</code> .
<code>pid > 0</code>	Espera hasta que termine el proceso hijo cuyo ID sea igual a <code>pid</code> .
<code>pid == 0</code>	Espera hasta que termine cualquier proceso hijo cuyo ID de grupo sea

	igual que el ID del proceso que invoca a <code>waitpid</code> .
<code>pid < 1</code>	Espera hasta que termine el proceso hijo cuyo ID de grupo sea, en valor absoluto, igual a <code>pid</code> .

La llamada `waitpid` regresa el ID del proceso hijo que haya terminado y guarda el estado de terminación del mismo en la localidad de memoria apuntado por `statloc`.

6.4 INFORMACIÓN SOBRE PROCESOS

En este párrafo vamos a estudiar las llamadas necesarias para consultar y fijar algunos de los parámetros más importantes de un proceso, los que describen cómo se relaciona el proceso con el resto del sistema. Nos vamos a centrar en los siguientes aspectos: identificadores asociados a un proceso, usuarios asociados al proceso, variables de entorno y parámetros por defecto.

6.4.1 IDENTIFICADORES DE PROCESO

Todo proceso tiene asociados dos números desde el momento de su creación: el identificador de proceso y el identificador del proceso padre.

El identificador de proceso (`PID`) es un número entero positivo que actúa a modo de nombre del proceso. El identificador del proceso padre (`PPID`) es el `PID` del proceso que ha creado al actual. El `PID` de un proceso no cambia durante el tiempo de vida de éste; sin embargo, su `PPID` sí puede variar. Esta situación se da cuando el proceso padre muere, pasando el `PPID` del proceso hijo a tomar el valor 1 (`PPID` del proceso `init`).

Para leer los valores de `PID` y `PPID` utilizaremos las llamadas `getpid` y `getppid`, respectivamente. Las declaraciones de estas funciones son:

```
#include <types.h>
pid_t getpid();
pid_t getppid();
```

Las llamadas a estas funciones no van a fallar nunca.

En UNIX, los procesos van a estar agrupados en conjuntos de procesos que tienen alguna característica común (por ejemplo, tener un mismo proceso padre). A estos conjuntos se les conoce como grupos de procesos y desde el sistema son controlados a través de un identificador de grupo de procesos. Para determinar a qué grupo pertenece un proceso, utilizaremos la llamada `getpgrp`, cuya declaración es:

```
#include <sys/types.h>
pid_t getpgrp();
```

El identificador de grupo de procesos es heredado por los procesos hijos después de una llamada a `fork`, pero también puede cambiarse creando un nuevo grupo de procesos. Esto lo conseguimos con la llamada `setpgrp` cuya declaración es la siguiente:

```
#include <sys/types.h>
pid_t setpgrp();
```

`setpgrp` hace que el proceso actual se convierta en el líder de un grupo de procesos. El identificador de este grupo va a coincidir con el `PID` del proceso y es el valor devuelto por `setpgrp`. Si la llamada falla, devolverá el valor -1. Insistiremos más sobre los grupos de procesos cuando estudiemos las señales.

El ejemplo siguiente muestra un programa que crea un proceso hijo. Padre e hijo van a mostrar su `PID`, `PPID` y su identificador de grupo de procesos. Pasados 5 segundos, el padre va a morir, y el hijo va a mostrar sus nuevos `PID`, `PPID` e identificador de grupo de procesos. Pasados 10 segundos, el hijo se va a convertir en el líder de un nuevo grupo de procesos y va a mostrar sus identificadores.

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

int main(int argc, char *argv[]) {
    if (fork() == 0) {
        fprintf(stdout, "PID = %d, PPID = %d, ID de grupo = %d\n",
            getpid(), getppid(), getpgrp());
        sleep(10);
        fprintf(stdout, "PID = %d, PPID = %d, ID de grupo = %d\n",
            getpid(), getppid(), getpgrp());
        setpgrp();
        fprintf(stdout, "PID = %d, PPID = %d, ID de grupo = %d\n",
            getpid(), getppid(), getpgrp());
        exit(0);
    }
    sleep(5);
    fprintf(stdout, "\n\nPID = %d, PPID = %d, ID de grupo = %d\n",
        getpid(), getppid(), getpgrp());
    exit (0);
}
```

6.4.2 IDENTIFICADORES DE USUARIO Y DE GRUPO

El kernel asocia a cada proceso dos identificadores de usuario y dos identificadores de grupo. Los identificadores de usuario son el identificador del usuario real (`UID`) y el identificador del usuario efectivo (`EUID`). Para el grupo, están el identificador del grupo real (`GID`) y el identificador del grupo efectivo (`EGID`).

El `UID` identifica al usuario que es responsable de la ejecución del proceso y el `GID` al grupo al cual pertenece el usuario.

El `EUID` se usa para determinar el propietario de los archivos recién creados, comprobar la máscara de permiso de acceso a archivos y los permisos para enviar señales a otros procesos. El identificador de usuario efectivo va a permitir acceder a archivos de otros usuarios. Normalmente, el `UID` y `EUID` coinciden, pero si un proceso ejecuta un programa que pertenece a otro usuario y que tiene activo el bit `S_ISUID` (cambiar el identificador del usuario al ejecutar), el proceso va a cambiar su `EUID` y va a tomar el valor de `UID` del

nuevo usuario. Es decir, a efectos de comprobación de permisos de usuario, va a tener los mismos permisos que tiene el usuario cuyo `UID` coincide con el `EUID` del proceso.

Con respecto al identificador de grupo efectivo, se aplica la misma norma, y el `EUID` es el `GID` del grupo al cual pertenece el usuario indicado en el `EUID`.

Para determinar qué valores toman estos identificadores, nos valemos de las llamadas: `getuid` (devuelve el identificador de usuario real), `geteuid` (devuelve el identificador del usuario efectivo), `getgid` (devuelve el identificador del grupo real) y el `getegid` (devuelve el identificador del grupo efectivo). La declaración de cada una de estas funciones es:

```
#include <sys/types.h>
uid_t getuid();
uid_t geteuid();
uid_t getgid();
gid_t getgid();
gid_t getegid();
```

Para cambiar los valores que toman estos identificadores, podemos usar las llamadas `setuid` y `setgid`, cuyas declaraciones son:

```
#include <sys/types.h>
int setuid(uid_t uid);
int setgid(uid_t gid);
```

Tanto `setuid` como `setgid` van a tener un comportamiento u otro dependiendo del valor que tenga el `EUID` del proceso. En el caso de `setuid`, si el identificador de usuario efectivo es el del superusuario, el kernel va a cambiar el `UID` y el `EUID` del proceso para que tomen el valor del parámetro `uid`. Si el identificador del usuario efectivo no es el del superusuario, pero el valor del parámetro `uid` coincide con el del identificador del usuario real, `EUID` va a tomar el valor `uid`. Esto se hace para restaurar el valor de `EUID` después de que el proceso ejecuta algún programa que tiene activo el bit `S_ISUID`. En estos casos, `setuid` devuelve 0 para indicar que no se ha producido ningún error. En cualquier otro caso devolverá el valor -1 y en `errno` estará el código del error producido.

Para `setgid` se aplica lo dicho en `setuid`, pero referido al `GID` y `EGID`. En concreto, si el `EUID` del proceso es el del superusuario, entonces `GID` y el `EGID` cambian para tomar el valor del parámetro `gid`, y si no lo es, pero `gid` tiene el valor `GID`, entonces el `EGID` toma el valor `gid`. En cualquier otro caso se producirá un error.

Un ejemplo típico de programa que usa estas llamadas es el programa `login`. Este programa se ejecuta con el `EUID` del usuario `root` (superusuario). Después de preguntarnos nuestro nombre de usuario y nuestra contraseña, consulta en el archivo `/etc/passwd` cuáles son nuestros `UID` y `GID`, para hacer sendas llamadas a `setuid` y `setgid` y que los identificadores `UID`, `EUID`, `GID` y `EGID` pasen a ser los del usuario que quiere iniciar la sesión de trabajo. Luego llama a `exec` para ejecutar un intérprete de órdenes que nos dé servicio. Este intérprete se va a ejecutar con los identificadores de usuario y grupo, tanto reales como efectivos, de acuerdo con el usuario que ha entrado en sesión.

Los procesos hijo van a heredar el `UID`, `EUID`, `GID` y `EGID` del padre.

6.4.3 VARIABLES DE ENTORNO

Cuando un proceso empieza a través de una llamada a `exec`, el sistema pone a su disposición un arreglo de cadenas de caracteres conocido como entorno.

Para los programas que se ejecutan mediante una llamada a `execl`, `execv`, `execlp`, `execvp`, el entorno es accesible únicamente a través de la variable global `environ`, que se declara

```
extern char **environ;
```

Para las llamadas `execle` y `execve`, el entorno es accesible también a través del tercer parámetro de la función `main`, el parámetro `envp`:

```
main(int argc, char **argv, char **envp);
```

Tanto `environ` como `envp` tienen estructura de arreglo de cadena de caracteres terminado con un apuntador a `NULL`. Por convenio, cada una de estas cadenas tiene la forma:

```
VARIABLE_ENTORNO=VALOR_VARIABLE
```

Algunas de las variables usadas por programas estándar son:

Variables	Descripción
HOME	Directorio de inicio de sesión de un usuario.
LANG	Identifica el idioma del Soporte en Lengua Nativa.
PATH	Indica la secuencia de directorios en lo que programa como <code>sh</code> , <code>time</code> , <code>nice</code> , <code>nohup</code> , etc. van a buscar cuando se especifica un archivo mediante su nombre y no mediante su ruta.
TERM	Identifica el tipo de terminal hacia el que se va a dirigir la salida. Es usado por programas como <code>vi</code> , <code>ed</code> , <code>mm</code> , etc.

Podemos ver el valor de todas las variables de entorno asociadas a nuestra sesión de trabajo mediante la orden `env`. Un ejemplo de entorno puede ser:

```
$ env
ORBIT_SOCKETDIR=/tmp/orbit-manchas
GPG_AGENT_INFO=/tmp/seahorse-6P3tHM/S.gpg-agent:4871:1
SHELL=/bin/bash
TERM=xterm
USER=manchas
SSH_AUTH_SOCK=/tmp/keyring-nPbS33/ssh
GNOME_KEYRING_SOCKET=/tmp/keyring-nPbS33/socket
SESSION_MANAGER=local/manchas-laptop:/tmp/.ICE-unix/4795
USERNAME=manchas
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
...
COLORTERM=gnome-terminal
XAUTHORITY=/home/manchas/.Xauthority
_=/usr/bin/env
```

El entorno de un proceso es heredado por todos sus procesos hijo, después de la llamada `fork`.

El ejemplo siguiente muestra un programa que presenta por pantalla todas las variables de entorno asociadas a un proceso.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    extern char **environ;

    while (*environ) {
        puts(*environ++);
    }
}
```

Este programa puede servirnos de ejemplo para ver cómo está codificada la orden `env`.

Para preguntar por el valor de una variable de entorno determinada o para declarar nuevas variables, podemos usar las funciones que ofrece la biblioteca estándar de C. Estas funciones son `getenv` y `putenv` y están declaradas como sigue:

```
#include <stdlib.h>
char* getenv(char *name);
int putenv(char *cadena);
```

`getenv` busca en la zona de variables de entorno una cadena de caracteres que tenga la forma de `name=value` y devuelve un apuntador a `value` en esta zona. Si la cadena no existe, devuelve un apuntador a `NULL`. Ejemplos de llamadas a `getenv` son:

```
char *valor;
...
valor = getenv("TERM"); /* 1 */
valor = getenv("TERM="); /* 2 */
valor = getenv("TERM=vt100"); /* 3 */
```

En estos ejemplos, si la variable `TERM` vale `300h`, las líneas 1 y 2 devuelve `valor="300h"`, pero la 3 devuelve `NULL`. Sólo en el caso de que `TERM` valga `vt100`, la sentencia devolverá `valor="vt100"`.

`putenv` permite declarar una nueva variable de entorno o modificar el valor de una ya existente. `cadena` debe tener la forma de `"name=value"`. Hay que tener presente que la zona de memoria a la que apunta `cadena` es añadida al entorno y que por tanto se producirá un error si `cadena` es una variable automática (local a una función), ya que al abandonar la función dejará de existir.

Hay que advertir también que `putenv` manipula el entorno referenciado por `environ`, pero no el referenciado en `envp` (tercer argumento de `main`).

Ejemplos de llamada a `putenv` son:

```
char *var1 = "TERM=vt100";
char *var2 = "NOMBRE=francisco";
...
```



```
putenv(var1);  
putenv(var2);
```

Por último, hay que indicar que las variables de entorno son locales a cada proceso, de tal forma que las modificaciones que se realicen sobre ellas no se conservan cuando muere el proceso. Esto no es aplicable a las variables que han sido declaradas como globales mediante la orden `export`.

6.4.4. PARÁMETROS RELATIVOS A ARCHIVOS

Todo proceso tiene asociado un directorio de trabajo y un directorio raíz. El directorio de trabajo indica dónde van a estar referidos los accesos a archivos que se realicen mediante un nombre de archivo y no mediante una ruta. Así, por ejemplo, si un proceso tiene como directorio de trabajo actual el directorio `/usr/local/bin` y abre el archivo `datos` mediante la llamada:

```
fd = open("datos", O_RDONLY);
```

El archivo que se está abriendo tiene `/usr/local/bin/datos` por ruta. Como vimos en el tema dedicado a sistema de archivos, la llamada `chdir` puede cambiar el directorio de trabajo asociado a un proceso.

El directorio raíz indica cuál es, dentro del sistema de archivos, el directorio que va a considerar el proceso como directorio raíz. Por lo general, el directorio raíz coincide con `/`, pero lo podemos cambiar con la llamada `chroot`. Así, la siguiente secuencia de código permite acceder al archivo `/usr/local/bin/prueba` indicando `/bin/prueba` como ruta:

```
chroot("/usr/local");  
fd = open("/bin/prueba", O_WRONLY); /*abriendo /usr/local/bin/prueba*/
```

Relacionado con la creación de archivos, cada proceso tiene una máscara que indica qué bits, dentro de los bits de permiso en el modo de archivo, deben estar inactivos. La función `umask` es la que permite establecer esta máscara. Así, por ejemplo, en la siguiente secuencia de código vamos a intentar crear un archivo con los permisos `rw-rw-rw-` (lectura, escritura y ejecución para el propietario, el grupo y otros), pero el archivo se va a crear realmente con los permisos `rw-r--r--`, ya que la máscara de creación de archivos se va a fijar previamente al valor `0033`.

```
umask(0033);  
fd = open("datos", O_WRONLY | O_CREAT, 0777);
```

Los procesos tienen limitado el tamaño máximo de los archivos que pueden crear y el tamaño máximo de memoria que pueden tener asignado. Para consultar estos parámetros se emplea la función `ulimit`. La declaración de esta llamada es la siguiente:

```
#include <ulimit.h>  
long ulimit(int cmd, ...);
```

`cmd` le indica a `ulimit` el tipo de operación que queremos realizar. Sus valores posibles son:

Valores	Descripción
UL_GETFSIZE	<code>ulimit</code> devolverá el tamaño máximo (expresado en bloques de 512

	bytes) de los archivos que puede escribir el proceso. Para la lectura no existe ningún límite.
UL_SETFSIZE	<code>ulimit</code> establece el tamaño máximo de los archivos que el proceso puede escribir. El tamaño se indica a través del segundo parámetro de la función (este parámetro es de tipo <code>long</code>). El tamaño se expresa en unidades de bloques de 512 bytes.
UL_GETMAXBRK	<code>ulimit</code> devuelve el tamaño máximo de memoria que puede ocupar cualquier proceso.

6.4.5 IDENTIFICACIÓN DE USUARIO

Cualquier proceso puede encontrar información sobre su `ID` y `EID` de usuario y de grupo. Algunas veces, sin embargo, queremos encontrar el nombre de usuario del usuario que está ejecutando un programa. Podríamos ejecutar la instrucción `getpwuid(getuid())` pero, ¿qué sucede si un usuario tiene múltiples nombres de usuario pero solo tiene un `ID` de usuario? El sistema normalmente mantiene información sobre el nombre bajo el cual entró un usuario. La función es:

```
#include <unistd.h>
char* getlogin();
```

La función falla si el proceso no se encuentra relacionado a una terminal con un usuario conectado. Un proceso de este tipo es llamado `process daemons`.

Dado el nombre de usuario, podemos usarlos para buscar el usuario dentro del archivo de `passwords` usando, por ejemplo, `getpwnam`.

6.5 CONTROL DE LA MEMORIA ASIGNADA

Para conocer las direcciones dinámicas donde se encuentran los segmentos de código y de datos de un proceso, están disponibles 6 símbolos que se definen de la siguiente forma:

```
extern _end;
extern end;
extern _etext;
extern etext;
extern _edata;
extern edata;
```

La dirección de símbolos `_etext` y `etext` es la primera dirección que hay después del segmento de código (también conocido como segmento de texto). La dirección `_edata` y `edata` es la primera que hay después de la región de datos que se inicializan al arrancar el programa. La dirección `_end` y `end` es la primera que hay después de la región de datos que no necesitan inicialización.

Para poder acceder desde un programa en C/C++ al valor de estos símbolos, hemos de utilizar el operador `&`. Así, `&end` será una forma correcta de acceder a la dirección donde está `end`.

La dirección asociada a `end` se conoce también como `program break`, y puede ser modificada con llamada `brk` y `sbrk`. La declaración de estas llamadas es:

```
int brk(char *endds);
char *sbrk(int incr);
```

Ambas funciones se emplean para cambiar dinámicamente la cantidad de memoria reservada para el segmento de datos del proceso que las invoca.

`brk` hace que `end` tome el valor indicado en `endds`. `sbrk` añade `incr` bytes al segmento de datos a continuación del `program break`. `incr` puede ser un valor negativo, con lo que conseguiremos decrementar el tamaño dedicado al segmento de datos. Las dos funciones inicializan a 0 la zona de datos añadida.

Si la llamada a `brk` no da ningún mensaje de error, la función devuelve 0. `sbrk` devuelve el antiguo valor del `program break` cuando se ejecuta satisfactoriamente. Ambas funciones devuelven -1 cuando se produce algún error.

6.5.1 STIKY BIT

Del Stiky bit ya hemos hablado en temas anteriores, pero quizá no quedó totalmente claro su significado, ya que aún no habíamos hablado de los conceptos relativos a procesos.

El stiky bit es uno de los bits de la palabra de modo de un archivo. Sólo es aplicable a los archivos ejecutables (programas) e indica que el segmento de código de este programa puede ser compartido por varios procesos. Así, cuando el kernel tiene que liberar memoria asignada al proceso (bien porque ha terminado su ejecución, bien porque debe pasar a la zona de intercambio), si hay otros procesos que estén compartiendo el segmento de código, este segmento no es descargado de memoria.

Solo el superusuario puede modificar el stiky bit de un programa.

El empleo de este bit supone una optimización del intercambio de información entre el disco y memoria, en el caso de programas muy utilizados como pueden ser el editor `vi`, el compilador de `C/C++`, etc.

6.5.2 BLOQUEO DE MEMORIA

Además del stiky bit, hay otros mecanismos para dejar residentes otros segmentos de un programa. La función `plock` se utiliza con esa idea. Su declaración es la siguiente:

```
#include <sys/lock.h>
int plock(int top);
```

Con `plock` podemos dejar residentes en memoria el segmento de código de un programa, su segmento de datos o ambos segmentos. Dependiendo del valor que tome `op`, indicaremos una acción u otra. En el archivo cabecera `<sys/lock.h>` están definidos los siguientes valores para `op`:

Valores	Descripción
PRLOCK	Dejar en memoria los segmentos de código y de datos. <code>Procces lock</code> .
TXTLOCK	Dejar en memoria el segmento de código. <code>Text lock</code> .
DATLOCK	Dejar en memoria el segmento de datos. <code>Data lock</code> .

UNLOCK	Desbloquear y liberar memoria ocupada.
---------------	--

Si la llamada a `plock` se realiza satisfactoriamente, la función devuelve 0; en caso contrario, devuelve -1 y en `errno` estará el código del error producido.

Sólo el superusuario tiene privilegios para emplear la llamada `plock`.

Los segmentos que queden residentes en memoria van a ser inmunes al intercambio con la memoria secundaria y van a restar capacidad del total de memoria libre del sistema. Como la memoria es un recurso escaso, caro y codiciado por todos los procesos, conviene no abusar del bloqueo de memoria, ya que puede bajar el rendimiento del sistema.

La técnica de dejar residente, total o parcialmente, un programa, mejora mucho la velocidad de ejecución en el caso de que un programa sea invocado muchas veces. Imaginemos, por ejemplo, una secuencia de órdenes que hace repetidas llamadas a un programa. Si ese programa se deja residente en memoria la primera vez que se ejecuta y se extrae de ella la última vez que se le invoca, habremos suprimido, del tiempo total de ejecución de la secuencia, el tiempo que se emplea en llevar el programa desde el disco a la memoria. Esto puede suponer una mejora sustancial en cuanto a rendimiento.

Cuando más adelante estudiemos memoria compartida, veremos que también hay posibilidad de bloquear una zona de memoria compartida para que no esté intercambiándose entre la zona de intercambio y la memoria principal.