Guía de Programación en C y Unix

José Miguel Santos Espino - jomis@dis.ulpgc.es

1. El compilador de C

Este apartado es suficiente para dominar el uso básico del compilador de lenguaje C, a través del programa `cc`. Para comprender este texto no tienen por qué estar familiarizados con el C, aunque se precisan unas mínimas nociones de este lenguaje. Además, deberían conocer conceptos básicos relacionados con el desarrollo de programas, como programa fuente, objeto, enlace (*linking*), etc.

1.1. El programa cc. Pasos de compilación

Para compilar programas escritos en C, disponen de un compilador de nombre `cc`. Este compilador toma como parámetros los archivos fuentes de que consta el programa final y, tras una serie de pasos, produce un archivo ejecutable. Si durante la compilación se produce un error, no se genera el ejecutable.

Los pasos de compilación en UNIX son al menos estos tres:

- Preproceso (macros, inclusión de archivos...)
- Compilación a objeto
- Enlace (*linking*) de objetos y bibliotecas

El preproceso interpreta las macros creadas con `#define` y expande los archivos para incluir con `#include`. No es de mucho interés para ustedes.

Un archivo una vez preprocesado se compila a código máquina, pero no se genera un ejecutable, sino un **archivo objeto**. Este estadio intermedio es necesario por muchos motivos, entre ellos que las rutinas de biblioteca, como `printf`, tienen que ser "empotradas" posteriormente para generar un ejecutable, y en general debido a que un programa en C puede constar de varios archivos compilados por separado.

Por eso existe un último paso, denominado enlace (*linking* en inglés), en el cual se recogen todos los archivos objetos más las bibliotecas (que también residen en archivos) necesarios para producir el archivo ejecutable.

1.2. Convenciones en los nombres de archivos

Como se ha visto, en el transcurso de la ejecución del cc aparecen en escena varias clases de archivos: fuentes, objetos, bibliotecas, ejecutables... El compilador de C es capaz de distinguir la clase de un archivo en base a sus últimos caracteres. La siguiente tabla muestra las convenciones más habituales.

`.c`	archivo fuente en C
`.h`	archivo cabecera fuente en C (sólo útil para los #include)
`.s`	archivo fuente en ensamblador (también reconocido)
`.i`	archivo fuente tras ser preprocesado (raramente empleado por el usuario)
`.o`	archivo objeto
`.a`	archivo de biblioteca

El compilador de C sólo genera un ejecutable, de nombre `a.out`, aunque se le puede indicar que tenga otro nombre con la opción `-o`**`nombre_de_ejecutable`**.

1.3. Uso del compilador

El programa `cc` se invoca desde el `shell`, admitiendo como argumentos los archivos empleados para construir el ejecutable más una serie de opciones de compilación. Las opciones y modalidades de uso del cc son amplísimas, por lo que en esta guía nos limitaremos a exponer las más comunes y útiles.

Por omisión, el `cc` genera un ejecutable llamado `a.out`. En la línea de órdenes pueden incluir tanto archivos fuentes en C como archivos objeto, incluso fuentes en ensamblador[1]. Los nombres de los archivos pueden aparecer en cualquier orden.

En el siguiente apartado aparece un resumen de opciones; antes de ello daremos algunos ejemplos.

Ejemplo 1:

```
cc pepe.c
```

Si `pepe.c` es un archivo fuente en C, se compila y se enlaza con las bibliotecas del sistema. Si no había errores sintácticos ni referencias a funciones o variables inexistentes, se genera el ejecutable `a.out`.

Ejemplo 2:

```
cc -o pepe main.c utilidades.c pepe.o -lm
```

Éste es un ejemplo más complejo donde anticipamos el uso de un par de opciones de compilación. La opción `-o pepe` sirve para que se genere el archivo ejecutable `pepe`, en lugar de `a.out`. Para construir el ejecutable se hace uso explícito de cuatro elementos:

- Un fuente `main.c` (quizás con el programa principal)
- Otro fuente, `utilidades.c` (quizás con funciones adicionales)
- Un objeto `pepe.o` (tal vez con otras utilidades ya compiladas)
- La biblioteca matemática estándar (con la opción `-lm`)

Con este ejemplo hacemos ver que un programa en C puede estar compuesto de varios módulos en forma de archivos fuentes. Y que si un módulo ya está compilado, podemos pasar como argumento al compilador el correspondiente `.o` para ahorrar tiempo.

1.4. Resumen de opciones para el compilador

Como ya se dijo, existe una infinidad de opciones para controlar la ejecución del cc. Algunas de las opciones más necesarias o comprensibles para ustedes son las siguientes:

[1] De hecho el paso de compilación a objeto suele atravesar una fase intermedia en que se genera un archivo en lenguaje ensamblador y se invoca al programa ensamblador del sistema.

`-Aa`	compila en C ANSI
`-Ac`	compila en C de K&R
`-c`	sólo compila, no hace el montaje
`-I`**`directorio`**	define directorios para buscar los `#includes`
`-l`**`lib`**	monta la biblioteca lib
`-L`**`directorio`**	define directorios para buscar las bibliotecas
`-o` **`archivo`**	establece el nombre del ejecutable
`-O`	optimiza el código
`-O`**`n`**	optimiza el código hasta el nivel n
`-S`	genera un fuente .s en ensamblador; no compila
`-v`	modo locuaz: imprime todos los pasos

Las opciones del compilador de C pueden variar según la versión de UNIX que se utilice.

Laboratorio de operativos: El compilador de C está preparado para compilar en modo ANSI, con lo que no hace falta que escriban la opción -Aa.

Algunas bibliotecas útiles, para la opción -llib, son:

`c`	biblioteca estándar del C. No hace falta incluirla explícitamente
`m`	biblioteca estándar matemática
`termcap`	contiene rutinas de manejo de terminales
`curses`	contiene rutinas de visualización con ventanas, etc.
`l`	biblioteca para uso del lex y el yacc

1.5. El compilador de C++

Es muy probable que en la máquina en la que trabajen exista un compilador de C++. Describiremos brevemente las características de uno de los compiladores de C++ más difundidos, el de la casa GNU (software de distribución gratuita).

El compilador de la GNU se llama **gcc**, aunque se le puede invocar como `gcc` o `c++`. Es capaz de reconocer tanto fuentes en C como en C++. El **gcc** reconoce como fuentes en C++ a los archivos con extensión `.cxx` o `.C` (letra "C" mayúscula).

Las opciones admitidas por el **gcc** son más o menos las mismas que acepta el compilador convencional de UNIX; aunque siempre es recomendable consultar el manual en línea.

2. Llamadas al sistema

El sistema operativo UNIX ofrece un conjunto de llamadas al sistema, en forma de funciones en C, que sirven de interfaz con los servicios provistos por el núcleo. En los siguientes apartados describiremos algunas de estas funciones, en concreto aquéllas que consideramos de uso más frecuente en esta asignatura.

Este texto no ha de considerarse un manual de programación, sino una simple guía. Lo que viene a continuación es poco más que un catálogo de funciones con algunos ejemplos de uso, pero no es sustituto de un buen manual de programación en C bajo UNIX, cuya lectura les recomendamos encarecidamente.

2.1. Interfaz de uso

Las llamadas al sistema en UNIX suelen acogerse a este prototipo:

```
int función ( arg1, arg2, ... );
```

Todas las llamadas al sistema son funciones C que retornan un entero. Habitualmente un valor negativo (especialmente un -1) indica algún tipo de error. Estas funciones siempre van en minúsculas y hay una tendencia a la parquedad en los nombres (suelen ser cortos). El número de argumentos aceptado en una llamada al sistema UNIX no suele ir más allá de tres.

2.2. Códigos de error

En general, las llamadas al sistema del UNIX devuelven un valor convencional en caso de error (casi siempre un -1). Esta información no basta casi nunca para explicar el cariz de la anomalía producida. Por suerte, todo programa puede acceder a una variable entera llamada `errno`, donde se deposita un código de error cada vez que sucede algo incorrecto.

Para acceder a `errno` han de incluir el archivo `<errno.h>`, que también define constantes simbólicas para los códigos de error. Por ejemplo, `ENOMEM` significa memoria insuficiente, `EPERM` es para "tipo de acceso denegado" (como cuando queremos abrir en modo escritura un archivo para el que no tenemos permiso de escritura), etc., etc.

En los manuales encontrarán el listado completo de los códigos de error, muchos de los cuales casi nunca se les manifestarán, por fortuna para ustedes.

3. Llamadas al sistema para manejo de archivos

El UNIX proporciona un conjunto de llamadas al sistema para la manipulación de archivos. Todas las aplicaciones o utilidades que en UNIX trabajan con archivos están fundamentadas en estos servicios básicos. La biblioteca estándar de C dispone de un conjunto de funciones para utilizar directamente estas llamadas al sistema, proporcionando al programador la misma visión que sobre los recursos tiene el sistema operativo UNIX.

Estas funciones se suelen denominar "de bajo nivel". La biblioteca estándar también ofrece otras rutinas más cómodas, construidas a partir de las llamadas al sistema. (La interfaz con estos servicios se encuentra en `<stdio.h>`).

Este apartado explica algunas de las llamadas al sistema del UNIX que trabajan con archivos, que les permitirán:

- Abrir y cerrar un archivo
- Crear y borrar un archivo
- Leer en un archivo
- Escribir en un archivo
- Desplazarse por un archivo

Laboratorio de Operativos: Las llamadas al sistema del HP-UX se encuentran en los manuales "C Programming Routines (sección 2)".

3.1. Manejo de archivos en UNIX

El manejo de archivos en UNIX sigue el modelo de la sesión. Para trabajar con un archivo hay primero que abrirlo con una invocación a la función `open`. Ésta devuelve un **descriptor de archivo** (*file descriptor* en inglés), un número entero que servirá de identificador de archivo en futuras operaciones. Finalmente hay que cerrar el archivo, con la función `close`, para liberar los recursos que tengamos asignados.

Existen al menos tres descriptores ya establecidos en la ejecución de un programa (ya los han abierto por nosotros). El descriptor 0 es la entrada estándar (normalmente el teclado), el descriptor 1 es la salida estándar (normalmente la pantalla) y el descriptor 2 el archivo estándar de visualización de errores (también la pantalla, normalmente). Los pueden considerar como simples archivos que ya han sido abiertos, y pueden trabajar con ellos con cierta normalidad. Incluso los pueden cerrar.

Los archivos en UNIX permiten tanto el acceso directo como el secuencial. Cada archivo abierto dispone de un puntero que se mueve con cada lectura o escritura. Hay una función especial llamada `lseek` para posicionar ese puntero donde se quiera dentro del archivo.

3.2. Especificación de los permisos: forma octal

Las llamadas al sistema `creat` y `open` admiten un parámetro entero en el que se especifican los permisos con los que se crea un archivo. Una de las maneras más cómodas de declararlos es mediante la representación octal.

Los permisos se forman como un número de 9 bits, en el que cada bit representa un permiso, tal y como se muestra en el cuadro (es el mismo orden con el que aparecen cuando hacemos un `ls`).

RWX	RWX	RWX
usuario	grupo	otros

Se toman los nueve permisos como tres números consecutivos de 3 bits cada uno. Un bit vale 1 si su permiso correspondiente está activado y 0 en caso contrario. Cada número se expresa en decimal, del 0 al 7, y los permisos quedan definidos como un número octal de tres dígitos. Para poner un número en octal en el lenguaje C, se escribe con un cero a la izquierda.

Por ejemplo, los permisos `rw-r-r-x` son el número octal 0645.

```
/* Crea un archivo con permisos RW-R--R-- */

int fd = creat ( "mi_archivo", 0644);
```

3.3. Apertura, creación y cierre de archivos

3.3.1. Función open

La función `open` abre un archivo ya existente, retornando un descriptor de archivo. La función tiene este prototipo:

```
int open ( char* nombre, int modo, int permisos );
```

El parámetro **nombre** es la cadena conteniendo el nombre del archivo que se quiere abrir.

El parámetro **modo** establece la forma en que se va a trabajar con el archivo. Algunas constantes que definen los modos básicos son:

`O_RDONLY`	Abre en modo lectura
`O_WRONLY`	Abre en modo escritura
`O_RDWR`	Abre en modo lectura-escritura
`O_APPEND`	Abre en modo apéndice (escritura desde el final)
`O_CREAT`	Crea el archivo y lo abre (si existía, se lo machaca)
`O_EXCL`	Usado con `O_CREAT`. Si el archivo existe, se retorna un error
`O_TRUNC`	Abre el archivo y trunca su longitud a 0

Para usar estas constantes, han de incluir la cabecera `<fcntl.h>`. Los modos pueden combinarse, simplemente sumando las constantes, o haciendo un "or" lógico, como en este ejemplo:

```
O_CREAT | O_WRONLY
```

El parámetro acceso sólo se ha de emplear cuando se incluya la opción `O_CREAT`, y es un entero que define los permisos de acceso al archivo creado. Consulten en la bibliografía cómo se codifican los permisos.

La función `open` retorna un descriptor válido si el archivo se ha podido abrir, y el valor -1 en caso de error.

3.3.2. Función creat

Si desean expresarmente crear un archivo, disponen de la llamada `creat`. Su prototipo es:

```
int creat ( char* nombre, int acceso );
```

Equivale (más o menos) a llamar a `open (nombre, O_RDWR|O_CREAT, acceso)`. Es decir, devuelve un descriptor si el archivo se ha creado y abierto, y -1 en caso contrario.

3.3.3. Función close

Para cerrar un archivo ya abierto está la función `close`

```
int close ( int archivo );
```

donde **archivo** es el descriptor de un archivo ya abierto. Retorna un 0 si todo ha ido bien y -1 si hubo problemas.

3.4. Borrado de archivos

La función

```
int unlink ( char* nombre );
```

borra el archivo de ruta **nombre** (absoluta o relativa). Devuelve -1 en caso de error.

3.5. Lectura y escritura

Para leer y escribir información en archivos, han de abrirlos primero con `open` o `creat`. Las funciones `read` y `write` se encargan de leer y de escribir, respectivamente:

```
int read ( int archivo, void* buffer, unsigned bytes );
int write( int archivo, void* buffer, unsigned bytes );
```

Ambas funciones toman un primer parámetro, **archivo**, que es el descriptor del archivo sobre el que se pretende actuar.

El parámetro **buffer** es un apuntador al área de memoria donde se va a efectuar la transferencia. O sea, de donde se van a leer los datos en la función `read`, o donde se van a depositar en el caso de `write`.

El parámetro **bytes** especifica el número de bytes que se van a transferir.

Las dos funciones devuelven el número de bytes que realmente se han transferido. Este dato es particularmente útil en la función `read`, pues es una pista para saber si se ha llegado al final del archivo. En caso de error, retornan un -1.

Hay que tener especial cautela con estas funciones, pues el programa no se va a detener en caso de error, ni hay control sobre si el puntero buffer apunta a un área con capacidad suficiente (en el caso de la escritura), etc., etc.

La primera vez que se lee o escribe en un archivo recién abierto, se hace desde el principio del archivo (desde el final si se incluyó la opción `O_APPEND`). El puntero del archivo se mueve al byte siguiente al último byte leído o escrito en el archivo. Es decir, UNIX trabaja con archivos secuenciales.

3.6. Movimiento del puntero del archivo

El C y UNIX manejan archivos secuenciales. Es decir, conforme se va leyendo o escribiendo, se va avanzando en la posición relativa dentro del archivo. El acceso directo a cualquier posición dentro de un archivo puede lograrse con la función `lseek`.

```
long lseek ( int archivo, long desp, int origen );
```

Como siempre, **archivo** es el descriptor de un archivo y abierto.

El parámetro **desp** junto con **origen** sirven para determinar el punto del archivo donde va a acabar el puntero. **desp** es un entero largo que expresa cuántos bytes hay que moverse a partir del punto indicado en **origen**, parámetro que podrá adoptar estos valores:

0	`SEEK_SET`	Inicio del archivo
1	`SEEK_CUR`	Relativo a la posición actual
2	`SEEK_END`	Relativo al final del archivo

Las constantes simbólicas se encuentran en `<stdlib.h>` y `<unistd.h>`.

El parámetro `desp` puede adoptar valores negativos, siempre que tengan sentido. Si el resultado final da una posición mayor que el tamaño del archivo, éste crece automáticamente hasta esa posición.

La función `lseek` devuelve un entero largo que es la posición absoluta donde se ha posicionado el puntero; o un -1 si hubo error. Obsérvese que la función `lseek` puede utilizarse también para leer la posición actual del puntero.

3.7. Ejemplos

En las siguientes líneas se muestran dos programas que emplean las llamadas al sistema.

3.7.1. Primer programa: creación y escritura

Este ejemplo crea un archivo y escribe en él unos caracteres.

```
#include <string.h>     /* Función strlen() */
#include <fcntl.h>      /* Modos de apertura y función open()*/
#include <stdlib.h>     /* Funciones write() y close() */

main ( int argc, char* argv[] )
{
     /* Cadena que se va a escribir */
     const char* cadena = "Hola, mundo";

     /* Creación y apertura del archivo */
     int archivo = open ("mi_archivo", O_CREAT|O_WRONLY,0644);

     /* Comprobación de errores */
     if (archivo==-1)
     {
          perror("Error al abrir archivo:");
          exit(1);
     }

     /* Escritura de la cadena */
     write(archivo, cadena, strlen(cadena));
     close(archivo);

     return 0;

}
```

3.7.2. Segundo programa: lectura

Este programa lee diez caracteres, a partir de la posición 400, de un archivo ya existente.

```
#include <fcntl.h>          /* Modos de apertura */
#include <stdlib.h>         /* Funciones de archivos */

main ( int argc, char* argv[] )
{
     char cadena[11];     /* Depósito de los caracteres */
     int leidos;

     /* Apertura del archivo */
```

```
    int archivo = open ("mi_archivo", O_RDONLY);

    /* Comprobación */
    if (archivo==-1)
    {
         perror("Error al abrir archivo:");
         exit(1);
    }

    /* Coloca el puntero en la posición 400 */
    lseek(archivo,400,SEEK_SET);

    /* Lee diez bytes */
    leidos = read(archivo, cadena, 10);
    close(archivo);
    cadena[10]=0;

    /* Mensaje para ver qué se leyó */
    printf ( "Se leyeron %d bytes. );
    printf ( "La cadena leída es %s\n", leidos,cadena );

    return 0;
}
```

4. Flujos estándares y redirección

Un proceso toma y escribe datos desde y hacia el exterior. El shell, por ejemplo, lee caracteres del teclado e imprime caracteres en la pantalla. En UNIX, los procesos se comunican con el exterior a través de flujos (`streams`).

Conceptualmente, un flujo es una **sucesión de bytes** que se puede ir leyendo o sobre la que se puede escribir caracteres. Un flujo puede ser un archivo ordinario, o estar asociado a un dispositivo. Cuando se lee del teclado es porque previamente se ha abierto como flujo de caracteres del que leer. Un proceso, cuando muestra algo por pantalla, está escribiendo caracteres a un flujo de salida.

4.1. Entrada, salida y error estándares

Los procesos lanzados por un procesador de órdenes (un shell) utilizan dos flujos de particular interés: la entrada estándar y la salida estándar. La entrada estándar es utilizada para recoger información, y la salida estándar para enviar información al exterior.

La entrada estándar suele ser el teclado. La salida estándar suele ser la pantalla. Por ejemplo, cuando se ejecuta `ls`, este programa nos muestra los archivos del directorio actual escribiendo los caracteres necesarios sobre la salida estándar (que suele ser la pantalla). Los flujos estándares pueden ser redirigidos a otros archivos.

Existe también el llamado **error estándar**, flujo donde se vierten los mensajes de error. Habitualmente coincide con la salida estándar, pero se considera un flujo diferente.

4.2. Utilización en un programa en C

La entrada y la salida estándares en UNIX son utilizables en forma de sendos archivos que están abiertos desde el inicio de la ejecución del programa. Como ya sabrán, cuando se abre un archivo con open se devuelve un **descriptor de archivo** (`file descriptor`), que es un número entero que se empleará en posteriores llamadas a las funciones de manejo de archivos como `read` o `write`.

En UNIX siempre se cumple que la entrada estándar tiene el descriptor de valor 0, y la salida estándar el de valor 1.

4.3. Redirección

Para redirigir la entrada o la salida hay que conseguir abrir el archivo de redirección de forma que el sistema le asigne el handle 0 ó 1. ¿Cómo lograr esto? Pues teniendo en cuenta que el algoritmo de la llamada `open` reserva el primer descriptor disponible, empezando a explorar desde el número cero.

Esto significa que si cerramos la salida estándar (descriptor número 1), el siguiente `open` que se realice asignará al archivo abierto el descriptor 1, quedando automáticamente reasignada la salida estándar. Sirva de ejemplo este código:

```
close (1);
open ("pepe.txt",O\_CREAT,0777);
printf("Estoy escribiendo en pepe.txt\n");
```

Cualquier nueva operación sobre la salida estándar se dirigirá al archivo "`pepe.txt`". (Suponiendo que el descriptor 0 no esté libre).

4.4. Preservar el antiguo flujo estándar: función `dup`

A veces conviene preservar el archivo de entrada o salida estándar actual, para restituirlo en el futuro. Para ello se emplea la llamada `dup`, que duplica un archivo ya abierto, esto es, lo vuelve a abrir y le reserva otro descriptor disponible.

Sintaxis: `int dup (int` **`handle`**`);`

Ejemplo:

```
int guardado = dup(1);       // preserva la salida estándar

/* Realiza la redirección */
close (1);
open ("pepe.txt",O\_CREAT,0777);

... trabajamos con la salida estándar redirigida ...

/* Restituye la antigua salida estándar */

close(1);                    // Cierra la actual
dup(guardado);               // Duplica el guardado
                             // (le asigna el descriptor 1)
close(guardado);             // Cierra el guardado (no hace más falta)
```

5. Ejecución de procesos

5.1. La función main y los argumentos

Habitualmente, cuando se lanza un programa a ejecución desde el shell, se le añaden parámetros o argumentos para definir exactamente qué queremos de él. Por ejemplo:

```
vi pepe.txt
ls -l /usr/include
cp pepe.c ../pipo.c
```

En el primer caso se invoca al editor `vi` especificándose que se desea trabajar con el archivo "`pepe.txt`". El archivo es un parámetro pasado al shell. El segundo caso es una llamada al programa `ls` que incorpora dos parámetros, como ocurre en el último ejemplo.

¿Qué son los parámetros o argumentos?

En principio, puede afirmarse que son conjuntos de caracteres separados por blancos (espacios o tabuladores). Ahora bien, no se consideran parámetros los redireccionamientos, y caracteres como el ampersand "`&`" o el punto y coma "`;`" actúan de separadores (en general, eso ocurre con todos los caracteres que tengan un significado para el intérprete de órdenes).

Ejemplo:

En la orden

```
ls -l /usr/include & >pepe.txt
```

los parámetros son "`ls`", "`-l`" y "`/usr/include`" ('`&`' y la redirección no cuentan).

Cada programa recibe los parámetros a través de su punto de entrada, que en el caso del lenguaje C es la función `main`. El formato mínimo que acepta esta función en UNIX es

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

donde `argc` expresa cuántos parámetros se han reconocido, y `argv` es un vector de cadenas de caracteres que precisamente contienen los parámetros, siendo `argv[i]` el parámetro `i`.

Los parámetros se empiezan a numerar en 0. El parámetro 0 es el nombre del programa invocado tal y como se pasó en la línea de órdenes. En el ejemplo de `vi pepe.txt`, los valores de `argc` y `argv` serían

```
argc = 2
argv[0] = "vi"
argv[1] = "pepe.txt"
```

5.2. Funciones para ejecución de programas

5.2.1. La función system()

La forma más sencilla de invocar una orden UNIX desde un programa en C es mediante la función `system`, que toma como único parámetro la orden que quieren ejecutar. Reconoce redirecciones, expresiones regulares, conductos (pipes), etc. Por ejemplo, la línea

```
system("ls -l /usr/include/*.h >pepe.txt")
```

ejecuta la cadena pasada como parámetro tal y como si la hubiéramos tecleado desde la consola. La función `system` se limita a lanzar un shell hijo pasándole como parámetro de entrada la cadena suministrada en la función.

La forma de más bajo nivel para ejecutar una orden consiste en lanzar a ejecución el programa deseado mediante alguna de las llamadas al sistema que empiezan por `exec`. Existen varias modalidades que difieren en la forma de pasar los parámetros al programa (aunque realmente se trata de una sola llamada al sistema UNIX).

5.2.2. Las llamadas exec...

El sistema operativo UNIX ofrece una llamada al sistema llamada '`exec`' para lanzar a ejecución un programa, almacenado en forma de archivo. Aunque en el fondo sólo existe una llamada, las bibliotecas estándares del C disponen de varias funciones, todas comenzando por '`exec`' que se diferencian en la manera en que se pasan parámetros al programa.

La versión típica cuando se conoce a priori el número de argumentos que se van a entregar al programa se denomina `execl`. Su sintaxis es

```
int execl ( char* archivo, char* arg0, char* arg1, ... , 0 );
```

Es decir, el nombre del archivo y luego todos los argumentos consecutivamente, terminando con un puntero nulo (vale con un cero). Sirva este ejemplo:

Para ejecutar

```
/bin/ls -l /usr/include
```

escribiríamos

```
execl ( "/bin/ls", "ls", "-l", "/usr/include", 0 );
```

Obsérvese que el primer argumento coincide con el nombre del programa.

En caso de desconocer con anticipación el número de argumentos, habrá que emplear la función `execv`, que tiene este prototipo:

```
execv ( char* archivo, char* argv [] );
```

El parámetro `argv` es una tira de cadenas que representan los argumentos del programa lanzado, siendo la última cadena un nulo (un cero). El ejemplo anterior se resolvería así:

```
char* tira [] =  "ls", "-l", "/usr/include", 0 ;
...
execv ( "/bin/ls", tira );
```

En los anteriores ejemplos se ha escrito el nombre completo del archivo para ejecutar ("`/bin/ls`" en vez de "`ls`" a secas). Esto es porque tanto `execl` como `execv` ignoran la variable `PATH`, que contiene las rutas de búsqueda. Para tener en cuenta esta variable pueden usarse las versiones execlp o execvp. Por ejemplo:

```
execvp ( "ls", tira );
```

ejecutaría el programa "`/bin/ls`", si es que la ruta "`/bin`" está definida en la variable `PATH`.

Todas las llamadas `exec...` retornan un valor no nulo si el programa no se ha podido ejecutar. En caso de que sí se pueda ejecutar el programa, se transfiere el control a éste y la llamada `exec...` nunca retorna. En cierta forma el programa que invoca un `exec` desaparece del mapa.

5.3. Procesos concurrentes: llamadas `fork` y `wait`

Para crear nuevos procesos, el UNIX dispone únicamente de una llamada al sistema, `fork`, sin ningún tipo de parámetros. Su prototipo es

```
int fork();
```

Al llamar a esta función se crea un nuevo proceso (proceso hijo), idéntico en código y datos al proceso que ha realizado la llamada (proceso padre). Los espacios de memoria del padre y el hijo son disjuntos, por lo que el proceso hijo es una copia idéntica del padre que a partir de ese momento sigue su vida separada, sin afectar a la memoria del padre; y viceversa.

Siendo más concretos, las variables del proceso padre son inicialmente las mismas que las del hijo. Pero si cualquiera de los dos procesos altera una variable, el cambio sólo repercute en su copia local. Padre e hijo no comparten memoria.

El punto del programa donde el proceso hijo comienza su ejecución es justo en el retorno de la función `fork`, al igual que ocurre con el padre.

Si el proceso hijo fuera un mero clon del padre, ambos ejecutarían las mismas instrucciones, lo que en la mayoría de los casos no tiene mucha utilidad. El UNIX permite distinguir si se es el proceso padre o el hijo por medio del valor de retorno de `fork`. Esta función devuelve un cero al proceso hijo, y el identificador de proceso (PID) del hijo al proceso padre. Como se garantiza que el PID siempre es no nulo, basta aplicar un `if` para determinar quién es el padre y quién el hijo para así ejecutar distinto código.

Con un pequeño ejemplo:

```
main()
{
     int x=1;
     if ( fork()==0 )
     {
          printf ("Soy el hijo, x=%d\n",x);
     } else {
          x=33;
```

```
            printf ("Soy el padre, x=%d\n",x);
      }
}
```

Este programa mostrará por la salida estándar las cadenas “Soy el hijo, x=1” y “Soy el padre, x=33”, en un orden que dependerá del compilador y del planificador de procesos de la máquina donde se ejecute.

Como aplicación de `fork` a la ejecución de programas, véase este otro pequeño ejemplo, que además nos introducirá en nuevas herramientas:

```
1     if ( fork()==0 )
2     {
3           execlp ("ls","ls","-l","/usr/include",0);
4           printf ("Si ves esto, no se pudo ejecutar el programa\n");
5           exit(1);
6           }
7     else
8     {
9           int tonta;
10          wait(&tonta);
11    }
```

Este fragmento de código lanza a ejecución la orden `ls -l /usr/include`, y espera por su terminación.

En la línea 1 se verifica si se es padre o hijo. Generalmente es el hijo el que toma la iniciativa de lanzar un archivo a ejecución, y en las líneas 2 a la 6 se invoca a un programa con `execlp`. La línea 4 sólo se ejecutará si la llamada `execlp` no se ha podido cumplir. La llamada a `exit` garantiza que el hijo no se dedicará a hacer más tareas.

Las líneas de la 8 a la 11 forman el código que ejecutará el proceso padre, mientras el hijo anda a ejecutar sus cosas. Aparece una nueva llamada el sistema, la función `wait`. Esta función bloquea al proceso llamador hasta que alguno de sus hijos termina. Para nuestro ejemplo, dejará al padre bloqueado hasta que se ejecute el programa lanzado o se ejecute la línea 5, terminando en ambos casos el discurrir de su único hijo.

Es decir, la función `wait` es un mecanismo de sincronización entre un proceso padre y sus hijos.

La llamada `wait` recibe como parámetro un puntero a entero donde se deposita el valor devuelto por el proceso hijo al terminar; y retorna el PID del hijo. El PID del hijo es una información que puede ser útil cuando se han lanzado varios procesos hijos y se desea discriminar quién exactamente ha terminado. En nuestro ejemplo no hace falta este valor, porque sólo hay un hijo por el que esperar.

Como ejemplo final, esta función en C es una implementación sencilla de la función `system`, haciendo uso de `fork`, `wait` y una función `exec`.

```
int system ( const char* str )
{
      int ret;
      if (!fork())
      {
            execlp ( "sh", "sh", "-c", str, 0 );
            return -1;
      }
      wait (&ret);
      return ret;
}
```

6. Comunicación entre procesos (IPC)

Los procesos en UNIX no comparten memoria, ni siquiera los padres con sus hijos. Por tanto, hay que establecer algún mecanismo en caso de que se quiera comunicar información entre procesos concurrentes. El sistema

operativo UNIX define tres clases de herramientas de comunicación entre procesos (IPC): los semáforos, la memoria compartida y los mensajes.

El tipo de llamadas al sistema para estos IPCs es análogo al de los semáforos: existen sendas funciones **shmget** y **msgget** para crear o enlazarse a un segmento de memoria compartida o a una cola de mensajes, respectivamente. Para alterar propiedades de estos IPCs, incluyendo su borrado, están las funciones **shmctl** y **msgctl**.

Para enviar o recibir mensajes, se utilizan las funciones **msgsnd** y **msgrcv**.

En este apartado se describirán brevemente algunas llamadas al sistema disponibles para el uso de las IPCs dentro de la programación en C.

6.1. Semáforos

¿Qué es un semáforo para el UNIX? Formalmente es muy similar a la definición clásica de Dijkstra, en el sentido de que es una variable entera con operaciones atómicas de inicialización, incremento y decremento con bloqueo.

El UNIX define tres operaciones fundamentales sobre semáforos:

- **semget**: Crea o toma el control de un semáforo
- **semctl**: Operaciones de lectura y escritura del estado del semáforo. Destrucción del semáforo
- **semop**: Operaciones de incremento o decremento con bloqueo

Como el lenguaje C no tiene un tipo "semáforo" predefinido, si queremos usar semáforos tenemos que crearlos mediante una llamada al sistema (**semget**). Esta llamada permite crear un conjunto de semáforos, en lugar de uno solo. Las operaciones se realizan atómicamente sobre todo el conjunto; esto evita interbloqueos y oscuras programaciones en muchos casos, pero para esta práctica es más bien un engorro.

Al crear un semáforo se nos devuelve un número identificador, que va a funcionar casi igual que los identificadores de fichero de las llamadas **open**, **creat**, etc. La función semget nos permite además "abrir" un semáforo que ya esté creado. Así, por ejemplo, si un proceso crea un semáforo, otros procesos pueden sincronizarse con aquél (con ciertas restricciones) disponiendo del semáforo con **semget**.

Para darle un valor inicial a un semáforo, se utiliza la función **semctl**.

El UNIX no ofrece las funciones clásicas P y V o equivalentes, sino que dispone de una función general llamada **semop** que permite realizar una gama de operaciones que incluyen las P y V.

semctl también se emplea para destruir un semáforo.

6.2. Llamadas al sistema para semáforos

Esta es una descripción resumida de las tres llamadas al sistema para operar con semáforos (**semget**, **semctl** y **semop**). Para una información más completa y fidedigna, diríjanse al manual de llamadas al sistema (sección 2).

Para el correcto uso de todas estas funciones, han de incluir el fichero cabecera **<sys/sem.h>**

Las tres funciones devuelven -1 si algo ha ido mal y en tal caso la variable **errno** informa del tipo de error.

6.2.1. Apertura o creación de un semáforo

Sintaxis:

int semget (`key_t` **key**, `int` **nsems**, `int` **semflg**);

semget devuelve el identificador del semáforo correspondiente a la clave `key`. Puede ser un semáforo ya existente, o bien **semget** crea uno nuevo si se da alguno de estos casos:

1. `key` vale **IPC_PRIVATE**. Este valor especial obliga a **semget** a crear un nuevo y único identificador, nunca devuelto por ulteriores llamadas a **semget** hasta que sea liberado con **semctl**.

2. `key` no está asociada a ningún semáforo existente, y se cumple que (**semflg** & **IPC_CREAT**) es cierto.

A un semáforo puede accederse siempre que se tengan los permisos adecuados.

Si se crea un nuevo semáforo, el parámetro nsems indica cuántos semáforos contiene el conjunto creado; los 9 bits inferiores de semflg contienen los permisos estilo UNIX de acceso al semáforo (usuario, grupo, otros).

semflg es una máscara que puede contener **IPC_CREAT**, que ya hemos visto, o **IPC_EXCL**, que hace crear el semáforo, pero fracasando si ya existía.

Ejemplo:

`int` **semid = semget** (**IPC_PRIVATE**, **1**, **IPC_CREAT | 0744**);

6.2.2. Operaciones de control sobre semáforos

Sintaxis:

`int` **semctl** (`int` **semid**, `int` **semnum**, `int` **cmd**...);

Esta es una función compleja (y de interfaz poco elegante) para realizar ciertas operaciones con semáforos. **semid** es un identificador de semáforo (devuelto previamente por **semget**) y **semnum**, el semáforo del conjunto sobre el que quieren trabajar. **cmd** es la operación aplicada; a continuación puede aparecer un parámetro opcional según la operación definida por **cmd**.

Las operaciones que les interesan a ustedes son:

- **GETVAL semctl** retorna el valor actual del semáforo
- **SETVAL** se modifica el valor del semáforo (un cuarto parámetro entero da el nuevo valor)
- **IPC_RMID** destruye el semáforo

Ejemplos:

valor = semctl (**semid**, **semnum**, **GETVAL**);

semctl (**semid**, **semnum**, **SETVAL**, **nuevo_valor**);

6.2.3. Operaciones sobre semáforos

Sintaxis:

`int` **semop** (`int` semid, `struct sembuf*` **sops**, `unsigned` **nsops**);

Esta función realiza atómicamente un conjunto de operaciones sobre semáforos, pudiendo bloquear al proceso llamador. **semid** es el identificador del semáforo y **sops** es un apuntador a un vector de operaciones. **nsops** indica el número de operaciones solicitadas.

La estructura **sembuf** tiene estos campos:

```
struct sembuf {
     unsigned short  sem\_num;  // número del semáforo dentro del conjunto
     short           sem\_op;   // clase de operación
                               // según sea >0, <0 o ==0
     short           sem\_flg;  // modificadores de operación
};
```

Cada elemento de **sops** es una operación sobre algún semáforo del conjunto de **semid**. El algoritmo simplificado de la operación realizada es éste (**semval** es el valor entero contenido en el semáforo donde se aplica la operación).

```
si semop<0
     si semval >= |semop|
        semval -= |semop|
     si semval < |semop|
        si (semflag \& IPC\_NOWAIT)!=0
           la función semop() retorna
        si no
           bloquearse hasta que semval >= |semop|
           semval -= |semop|
     si semop>0
        semval += semop
     si semop==0
        si semval = 0
           SKIP
        si semval != 0
           si (semflag \& IPC\_NOWAIT)!=0
              la función semop() retorna
           si no
              bloquearse hasta que semval == 0
```

Resumiendo un poco, si el campo **semop** de una operación es positivo, se incrementa el valor del semáforo. Asimismo, si **semop** es negativo, se decrementa el valor del semáforo si el resultado no es negativo. En caso contrario el proceso espera a que se dé esa circunstancia. Es decir, **semop==1** produce una operación V y **semop==-1**, una operación P.

6.3. Ejemplos de uso

Para ilustrar de forma concreta el empleo de semáforos bajo UNIX, les mostramos unos ejemplos de subrutinas en C que les pueden servir como modelos para elaborar sus rutinas de sincronización en las prácticas de la asignatura.

En concreto, son unas funciones que implementan las operaciones P y V de un semáforo clásico (inicialización, incremento y decremento con posible bloqueo del proceso llamador). Así definidas, o con pocas modificaciones, les pueden servir como la interfaz para uso de semáforos en sus aplicaciones.

```
\#include <sys/types.h>/* para key\_t */

/* Crea un semáforo con un valor inicial, dada una clave */
/* Devuelve el identificador (válido o no) del semáforo */

int crea\_sem ( key\_t clave, int valor\_inicial );

/* Operaciones P y V sobre un semáforo */

void sem\_P ( int semid );
void sem\_V ( int semid );

/**********************************/
/********   IMPLEMENTACIÓN   *******/
/**********************************/

\#include <sys/ipc.h>
\#include <sys/sem.h>
\#define PERMISOS 0644

/* crea\_sem: abre o crea un semáforo */

int crea\_sem ( key\_t clave, int valor\_inicial )
```

```
{
int semid = semget(              /* Abre o crea un semáforo... */
            clave,               /* con una cierta clave */
            1,                   /* con un solo elemento */
            IPC\_CREAT|PERMISOS  /* lo crea (IPC\_CREAT) con  */
                                 /* unos PERMISOS */
            );

if ( semid==-1 ) return -1;

/* Da el valor inicial al semáforo */
semctl ( semid, 0, SETVAL, valor\_inicial );
return semid;
}

/* abre\_sem: Abrir un semáforo que otro proceso ya creó */
int abre\_sem (key\_t clave)
{
     return semget(clave,1,0);
}

/* Operaciones P y V */
void sem\_P ( int semid )        /* Operación P */
{
     struct sembuf op\_P [] =
     {
          0, -1, 0               /* Decrementa semval o bloquea si cero */
     };

     semop ( semid, op\_P, 1 );
}

void sem\_V ( int semid )        /* Operación V */
{
     struct sembuf op\_V [] =
     {
          0, 1, 0                /* Incrementa en 1 el semáforo */
     };
     semop ( semid, op\_V, 1 );
}
```

7. Memoria compartida

Las utilidades de memoria compartida permiten crear segmentos de memoria a los que pueden acceder múltiples procesos, pudiendo definirse restricciones de acceso (sólo lectura).

Para trabajar con un segmento de memoria compartida, es necesario crear un vínculo (attachment) entre la memoria local del proceso interesado y el segmento compartido. Esto se realiza con la función shmat. El proceso que vincula un segmento de memoria compartida cree estar trabajando con ella como si fuera cierta área de memoria local. Para deshacer el vínculo está la función shmdt.

7.1. Obtener un segmento de memoria

`shmget` - obtiene un segmento de memoria compartida.

7.1.1. Sintaxis

```
#include <sys/shm.h>
int shmget( key_t key, size_t size, int shmflg);
```

7.1.2. Descripción

La función `shmget` retorna el identificador de memoria compartida asociada a `key`. Un identificador de memoria compartida y la estructura de datos asociada se crearán para `key` si una de las siguientes condiciones se cumple:

- Si `key` es igual a `IPC_PRIVATE`. Cuando esto ocurre se creará un nuevo identificador, si existen disponibilidades, este identificador no será devuelto por posteriores invocaciones a shmget mientras no se libere mediante la función `shmctl`. El identificador creado podrá ser utilizado por el proceso invocador y sus descendientes; sin embargo esto no es un requerimiento. El segmento podrá ser accedido por cualquier proceso que posea los permisos adecuados.
- Si `key` aún no tiene asociado un identificador de memoria compartida y además `shmflg & IPC_CREAT` es verdadero.

Como consecuencia de la creación, la estructura de datos asociada al nuevo identificador se inicializa de la siguientes manera: `shm_perm.cuid` y `shm_perm.uid` al identificador de usuario efectivo del proceso invocador, `shm_perm.gcuid` y `shm_perm.guid` al identificador de grupo efectivo del proceso invocador, los 9 bits menos significativos de `shm_perm.mode` se inicializan a los 9 bits menos significativos del parámetro `shmflg`, `shm_segsz` al valor especificado por `size`, `shm_ctime` a la fecha y hora que el sistema poseía en el momento de la invocación y por último se ponen a cero `shm_lpid`, `shm_nattach`, `shm_atime` y `shm_dtime`.

Si la ejecución se realiza con éxito, entonces retornará un valor no negativo denominado identificador de segmento compartido. En caso contrario, retornará -1 y la variable global `errno` tomará en código del error producido.

7.2. Vincular un área de memoria compartida

`shmat`, `shmdt` - funciones de operación sobre memoria compartida.

7.2.1. Sintaxis

```
#include <sys/shm.h>
char *shmat( int shmid, void *shmaddr, int shmflg );
int shmdt( void *shmaddr )
```

7.2.2. Descripción

`shmat` asocia el segmento de memoria compartida especificado por `shmid` al segmento de datos del proceso invocador. Si el segmento de memoria compartida aún no había sido asociado al proceso invocador, entonces `shmaddr` debe tener el valor de cero y el segmento se asocia a una posición en memoria seleccionado por el sistema operativo. Dicha localización será la misma en todos los procesos que acceden al objeto de memoria compartida. Si el segmento de memoria compartida ya había sido asociado por el proceso invocador, `shmaddr` podrá tener un valor distinto de cero, en ese caso deberá tomar la dirección asociada actual del segmento referenciado por `shmid`. Un segmento se asocia en modo sólo lectura si `shmflg & SHM_RDONLY` es verdadero; si no entonces se podrá acceder en modo lectura y escritura. No es posible la asociación en modo sólo escritura. Si la función se ejecuta con éxito, entonces retornará la dirección de comienzo del segmento compartido, si ocurre un error devolverá -1 y la variable global `errno` tomará el código del error producido.

shmdt desasocia del segmento de datos del proceso invocador el segmento de memoria compartida ubicado en la localización de memoria especificada por shmaddr. Si la función se ejecuta sin error, entonces devolverá 0, en caso contrario retornará -1 y errno tomará el código del error producido.

7.3. Operaciones de control

shmctl - Realiza operaciones de control en una región de memoria compartida.

7.3.1. Sintaxis

```
#include <sys/shm.h>
int smctl(int shmid, int cmd, struct shmid_ds *buff);
```

7.3.2. Descripción

La función shmctl permite realizar un conjunto de operaciones de control sobre una zona de memoria compartida identificada por shmid. El argumento cmd se usa para codificar la operación solicitada. Los valores admisibles para este parámetro son:

- IPC_STAT lee la estructura de control asociada a shmid y la deposita en la estructura apuntada por buff.
- IPC_SET actualiza los campos shm_perm.uid, shm_perm.gid y shm_perm.mode de la estructura de control asociada a shmid tomando los valores de la estructura apuntada por buff.
- IPC_RMID elimina el identificador de memoria compartida especificado por shmid del sistema, destruyendo el segmento de memoria compartida y las estructuras de control asociadas. Si el segmento está siendo utilizado por más de un proceso, entonces la clave asociada toma el valor IPC_PRIVATE y el segmento de memoria es eliminado, el segmento desaparecerá cuando el último proceso que lo utiliza notifique su desconexión del segmento. Esta operación sólo la podrán utilizar aquellos procesos que posean privilegios de acceso a recurso apropiados, para llevar a cabo esta comprobación el sistema considerará el identificador de usuario efectivo del proceso y lo comparará con los campos shm_perm_id y shm_perm_cuid de la estructura de control asociada a la región de memoria compartida.
- SHM_LOCK bloquea la zona de memoria compartida especificada por shmid. Este comando sólo puede ejecutado por procesos con privilegios de acceso apropiados.
- SHM_UNLOCK desbloquea la región de memoria compartida especificada por shmid. Esta operación, al igual que la anterior, sólo la podrán ejecutar aquellos procesos con privilegios de acceso apropiados.

La función shmctl retornará el valor 0 si se ejecuta con éxito, o -1 si se produce un error, tomando además la variable global errno el valor del código del error producido.

8. Colas de mensajes

8.1. Creación y obtención de una cola

msgget : Obtiene una cola de mensajes.

Sintaxis

#include sys/msg.h

int msgget (key_t **key**, int **msgflg**);

Descripción

La función **msgget** se utiliza para solicitar al sistema una cola de mensajes referenciada por el parámetro key. Una cola de mensajes se creará mediante esta función si se cumple alguna de las siguientes condiciones:

1. El argumento key toma el valor **IPC_PRIVATE**.
2. No existe cola de mensajes en el sistema con clave identificadora igual al valor de **key** y además se ha especificado el flag **IPC_CREAT** en el parámetro **msgflg**.

La creación de una cola de mensajes en el sistema implicará que la estructura de control asociada a la cola se inicialice de la siguiente forma:

1. **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid** y **msg_perm.gid** se ponen al valor del identificador de usuario y de grupo efectivo respectivamente del proceso invocador.
2. Los 9 bits menos significativos de **msg_perm.mode** se igualan a los 9 bits menos significativos del parámetro **msgflg**.
3. **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime** y **msg_rtime** se ponen a 0.
4. **msg_ctime** toma el valor de la hora/fecha actual.
5. **msg_qbytes** se pone al límite del sistema

El retorno de la función **msgget** será un entero no negativo si ésta se ejecuta con éxito. Dicho valor será el identificador del sistema asociado a la cola referenciada por **key**. En caso de error, la función retornará -1 y la variable **errno** tomará el valor del código del error producido.

8.2. Envío y recepción de mensajes

msgsnd, **msgrcv** : Operaciones de envio y recepción de mensajes.

8.2.1. Envío: sintaxis

#include <sys/msg.h>

int msgsnd(int *msqid*, const void ***msgp**, void ***msgp**, `size_t` **msgsz**, int **msgflg**);

8.2.2. Envío: descripción

La función **msgsnd** envía un mensaje a la cola asociada al identificador del sistema referenciado por el parámetro **msqid**. El argumento **msgp** es un puntero al comienzo de la zona de memoria de usuario donde está almacenado el mensaje a enviar, el mensaje debe poseer un campo de comienzo del tipo `long` en el que se indica el tipo del mensaje, el siguiente campo deberá contener el mensaje própiamente dicho, por ejemplo:

long mi_tipo;

char texto[ ];

El campo tipo de mensaje, **mi_tipo**, se podrá utilizar un proceso receptor para seleccionar el tipo de mensajes que desea recibir. El campo del mensaje en sí, **texto**, deberá poseer una longitud igual al valor especificado en el parámetro **msgsz**. El último parámetro de **msgsnd**, **msgflg**, especifica la acción a realizar si al menos una de las siguientes condiciones se cumple:

1. En la cola de mensajes ya existe uno del tipo especificado, esto se controla mediante el campo **msg_qbytes** de la estructura de control asociada a la cola de mensajes referenciada.
2. Se ha alcanzado el número máximo de mensajes en las colas del sistema.

Si (**msgflg & IPC_NOWAIT**) es verdadero, entonces el mensaje no se envia y se produce un retorno inmediato. En caso contrario, es decir, si (**msgflg & IPC_NOWAIT**) es falso, entonces el proceso invocador suspende la ejecución hasta que alguna de las siguientes condiciones ocurra:

1. La condición responsable para la suspensión ya no se da.
2. La cola referenciada por msqid es eliminada del sistema. Cuando esto ocurre **msgsnd** retorna con -1 y la variable global **errno** toma el valor **EIDRM**.
3. El proceso invocador recibe una señal del sistema. Cuando esto ocurre, el proceso continua su ejecución tal y como establece el tratamiento de señales (ver función signal).

Una vez finalizada con éxito la ejecución de la función msgsnd, se actualizan los siguientes campos de control:

- **msg_qnum** se incrementa en 1.
- **msg_lspid** se iguala al identificador del proceso invocador.
- **msg_stime** toma el valor de la fecha/hora actual.

8.2.3. Recepción: sintaxis

#include <sys/msg.h>

int msgrcv((int *msqid*, void **msgp*, `size_t` *msgsz*, `long` *msgtyp*, `int` *msgflg*);

8.2.4. Recepción: descripción

La función **msgrcv** toma un mensaje de la cola especificada por el parámetro **msqid** y lo copia en la zona de memoria apuntada por **msgp**. El mensaje recibido tiene una estrura análoga a la del mensaje enviado por emisor. El argumento **msgtyp** se utiliza para especificar el tipo de mensaje a recibir, concretamente:

- Si **msgtyp** es igual a 0, entonces se tomará el primer mensaje recibido en la cola.
- Si **msgtyp** es mayor o igual que 1, entonces el primer mensaje del tipo igual al valor especificado se tomará de la cola.
- Si **msgtyp** es negativo, entonces el primer mensaje cuyo tipo sea menor o igual que el valor absoluto de **msgtyp** se tomará de la cola.

El parámetro **msgflg** especifica la acción que se realizará en el caso que no exista en la cola mensaje alguno del tipo requerido, específicamente:

- Si (**msgflg & IPC_NOWAIT**) es verdadero, entonces el proceso invocador retornará inmediatamente con un valor de -1 y la variable **errno** tomará el valor **ENOMSG**.
- Si (**msgflg & IPC_NOWAIT**) es falso, entonces el proceso invocador suspenderá la ejecución hasta que se cumpla alguna de las siguientes condiciones:
- Se recibe un mensaje del tipo deseado.
- La cola de mensajes es eliminada del sistema, en este caso se retornará-1 y la variable errno tomará el valor **EIDRM**.
- Se produce una señal a tratar por el proceso invocador, en este caso se continuará la ejecución de la forma definida en el tratamiento de señales.

Una vez ejecutada con éxito la función msgrcv, alguno campos de control asociados a la cola se actualizan de la siguiente manera:

- **msg_qnum** se decrementa en 1.
- **msg_lrpid** se pone al valor del identificador del proceso invocador.
- **msg_setime** se actualiza actualiza a la fecha/hora actual.

8.3. Operaciones de control (ej. borrar cola)

msgctl - Operaciones de control en colas de mensajes.

Sintaxis

#define <sys/msg.h>

int msgctl(int *msqid*, int *cmd*, struct `msqid_ds` **buf*);

Descripción

La función **msgctl** permite realizar operaciones de control en la cola de mensajes especificada por el parámetro **msqid**. La operación se especifica mediante el argumento **cmd**, cuyos valores posibles son:

- IPC_STAT: lee la estructura de control asociada a la cola y los deposita en la estructura apuntada por **buf**.
- IPC_SET: actualiza los campos de control siguientes: **msg_perm.uid**, **msg_perm.gid**, **msg_perm.mode** (sólo los 9 bits menos significativos) y **msg_qbytes** a los valores que poseen los campos homónimos en la estructura apuntada por **buf**. Para poder realizar esta operación el proceso invocador debe poseer un identificador igual al usuario `root` o bien coincidir con el valor almacenado en el campo **msg_perm.uid** o en el campo **msg_perm.cuid** de la estructura asociada a la cola de mensajes.
- IPC_RMID: elimina la cola de mensajes identificada por **msqid**. Esta orden sólo puede ejecutarla un proceso con identificador de usuario igual a `root`, o igual al campo **msg_perm.cuid** o al campo **msg_perm.uid**.

Si la función **msgctl** se ejecuta con éxito, entonces retornará 0. En caso contrario retornará-1 y la variable global **errno** tomará un valor igual al código del error producido.