Escuela Politécnica Superior de Elche
Grado en Ingeniería Informática en Tecnologías de la Información

Sistemas Operativos
Cuaderno de practicas C - Señales

Señales
Llamadas al Sistema con ejemplos prácticos.

Objetivos

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

Requisitos

Conocer la programación en C básica.

El entorno de desarrollo

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

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

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

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

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

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

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

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

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

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

Ejemplo c001.c:
Programa Hola Mundo

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

Lo compilaríamos con:

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

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

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

Llamadas al sistema

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

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

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

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

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

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

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

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

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

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

Proceso:

1. Gestión de procesos.
2. Señales.

Fichero:

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

Tiempo:

5. Gestión de tiempo

Resultado de las llamadas al sistema

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

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

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

¿Por qué sale dos veces el mensaje?

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

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

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

La variable global errno

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

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

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

```
extern int * __error();

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

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

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

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

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

La función perror()

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

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

```
#ifndef _I386_ERRNO_H
#define _I386_ERRNO_H

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

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

#define EDEADLOCK       EDEADLK

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

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

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

#endif
```

Ejemplo **c004.c**:

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

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

Ejemplo c005.c:
de cómo se muestra el texto de error asociado al error ocurrido, utilizando la función **strerror()** que se limita a extraer el texto asociado al código de error pasado.

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

Ejemplo c006.c:

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

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

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

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

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

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

Variables de entorno

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Ejemplo c013.c:

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

$ echo $PATH

$ ejemplo13 PATH

En ambos casos obtendremos los mismos resultados.

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

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

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

Señales

Las señales son interrupciones software que nos permiten manejar eventos asíncronos, son un ejemplo clásico de eventos asíncronos.

En unix las señales tienen nombres. Sus nombres empiezan siempre con las letras **SIG**. Se pueden consultar en /usr/include/asm/**signal.h**. Por ejemplo:

SIGTERM: señal para terminar un proceso

SIGABRT: señal de abortar un proceso. Se genera con la función abort()

SIGALRM: señal de alarma que se genera cuando vence el tiempo que se ha establecido mediante alarm()

SIG_CHILD: es enviada por todo proceso hijo a su padre en el mismo instante que realiza exit. De esta manera, el padre sabe que su hijo ha pedido terminar.

SIGUSR1 y **SIGUSR2**: para enviar señales de usuario entre procesos, conociendo el PID del proceso al que se envía la señal. Atención para estas señales el kernel no mantiene una cola, es decir si un proceso está atendiendo una de estas y llega otra, se pierde.

Se pueden generar señales de muchos modos:

- **Mediante una combinación de teclas:** Por ejemplo Ctrl+C genera una señal (**SIGINT**, interrupción de teclado) que provoca la terminación del proceso que se está ejecutando en primer plano, o por ejemplo Ctrl+Z (**SIGTSTP**) que provoca la parada de todos los procesos en primer plano.
- **Por excepciones hardware:** por ejemplo una división por 0, hacer referencia a una dirección de memoria inválida. Normalmente estas condiciones las detecta el hardware y lo notifica al núcleo. Entonces el núcleo genera la señal apropiada al proceso que se estaba ejecutando cuando se produjo la excepción. Por ejemplo, un proceso que hace referencia a una dirección de memoria inválida hará que el núcleo mande la señal **SIGSEGV** a ese proceso.
- **Mediante la función kill():** esta función nos permite enviar una señal a otro proceso, siempre que el que genere la señal sea el propietario del proceso o sea el superusuario.
- **Mediante el comando kill:** Esta orden es en realidad una interfaz de la función kill
- **Por condiciones software:** Por ejemplo, se genera la señal **SIGALRM** cuando vence el tiempo establecido mediante alarm() o **SIGPIPE** que se genera cuando un proceso escribe en una tubería después de que el lector de la tubería haya acabado.

Cuando se genera una señal, al núcleo se le puede indicar que haga diferentes cosas como:

1. **Que ignore la señal:** Esto es así para muchas señales pero hay dos que no se pueden ignorar nunca: **SIGSTOP** y **SIGKILL**.
2. **Que capture la señal para que se ejecute un trozo de código definido por el usuario:** Por ejemplo un proceso ha creado ficheros temporales, al capturar la señal **SIGTERM** se asocia una función que borre estos ficheros antes de terminar el proceso.
3. **Permitir que se ejecute la acción por defecto asociada a la señal:** Para muchas señales la acción por defecto de las señales es la finalización del proceso.

Las principales señales que se encuentran en el fichero signal.h son:

Name	Default Action	Description
SIGHUP	terminate process	terminal line hangup
SIGINT	terminate process	interrupt program
SIGQUIT	create core image	quit program
SIGILL	create core image	illegal instruction
SIGTRAP	create core image	trace trap
SIGABRT	create core image	abort() call (formerly SIGIOT)
SIGEMT	create core image	emulate instruction executed
SIGFPE	create core image	floating-point exception
SIGKILL	terminate process	kill program
SIGBUS	create core image	bus error
SIGSEGV	create core image	segmentation violation
SIGSYS	create core image	non-existent system call invoked
SIGPIPE	terminate process	write on a pipe with no reader
SIGALRM	terminate process	real-time timer expired
SIGTERM	terminate process	software termination signal
SIGURG	discard signal	urgent condition present on socket
SIGSTOP	stop process	stop (cannot be caught or SIGXFSZ terminate process file size limit exceeded (setrlimit())
SIGVTALRM	terminate process	virtual time alarm (setitimer())
SIGPROF	terminate process	profiling timer alarm (setitimer())
SIGWINCH	discard signal	Window size change
SIGINFO	discard signal	status request from keyboard
SIGUSR1	terminate process	User defined signal 1
SIGUSR2	terminate process	User defined signal 2

La orden kill

Envía la señal especificada a un determinado proceso. Si no se especifica la señal, por defecto se envía la señal SIGTERM (terminación de in proceso).

Para enviar una señal haríamos:

```
kill –SIGTERM <pid del proceso>
```

La opción –l nos permite obtener un listado de todas las señales.

```
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      32) SIGRTMIN    33) SIGRTMIN+1
34) SIGRTMIN+2  35) SIGRTMIN+3  36) SIGRTMIN+4  37) SIGRTMIN+5
38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8  41) SIGRTMIN+9
42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMAX-15 49) SIGRTMAX-14
50) SIGRTMAX-13 51) SIGRTMAX-12 52) SIGRTMAX-11 53) SIGRTMAX-10
54) SIGRTMAX-9  55) SIGRTMAX-8  56) SIGRTMAX-7  57) SIGRTMAX-6
58) SIGRTMAX-5  59) SIGRTMAX-4  60) SIGRTMAX-3  61) SIGRTMAX-2
62) SIGRTMAX-1  63) SIGRTMAX
```

Llamadas al sistema relacionadas con señales

sigaction

Para cambiar la acción por defecto de una señal utilizaremos la función sigaction.

El prototipo es el siguiente:

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Esta función espera tres parámetros:

1. el primero es la señal,
2. el segundo define la nueva función a la que se debe transferir el control al recibir la señal
3. el tercero devuelve el antiguo manejador establecido.

La estructura **struct sigaction** tiene dos miembros muy importantes:

```
The sigaction structure is defined as something like

       struct sigaction {
         void (*sa_handler)(int);
         void (*sa_sigaction)(int, siginfo_t *, void *);
         sigset_t sa_mask;
         int sa_flags;
         void (*sa_restorer)(void);
         }
```

- **void (*sa_handler)(int)**: Puntero a la función que recibe la señal o bien, **SIG_DFL** para volver al tratamiento normal o **SIG_IGN** para deshabilitar su tratamiento.
- **sigset_t sa_mask**: Mascara con las señales adicionales a bloquear cuando se ejecuta la función que trata la señal. Para inicializar un conjunto de señales se utiliza **sigemptyset()** y para añadir una nueva señal a ese conjunto se utiliza **sigaddset()**.

Se obtiene más información en las páginas correspondientes del manual en http://man7.org/linux/man-pages/man2/rt_sigaction.2.html

alarm

```
unsigned int alarm(unsigned int seconds);
```

Mediante la función **alarm()** se establece una alarma mediante la cual, cuando ha vencido el tiempo establecido en la llamada, se genera la señal **SIGALRM**.

Man page en http://man7.org/linux/man-pages/man2/alarm.2.html

Llamada	Descripción
Includes: #include <signal.h> **Declaración:** void (* signal(int sig, void(*func)(int)))(int) En FreeBSD existe un equivalente más fácil de leer: typedef void (*sig_t) (int) sig_t signal(int sig, sig_t func) **Uso:** **signal** (señal, función)	Habilita al proceso para el tratamiento de señales que le llegan, para que pueda "defenderse" de las agresiones de "kill". **Si un proceso recibe una señal que no estaba esperando, se mata al proceso; cuando la señal es esperada, se trata en un segundo nivel de interrupción**. Recibe el nombre de la señal a capturar, y el nombre de la rutina (puntero) que se ejecuta en caso de recibir una interrupción del manejador de interrupciones. El parámetro "sig" especifica la señal que se va a recibir. El parámetro "func" es el nombre de la función que se ejecutará cuando se reciba la señal especificada. En este último parámetro podemos poner además dos constante predefinidas por el sistema: • **SIG_DFL** (valor 0, rutina por defecto, que consiste en realizar un exit() con el número de la señal que le llega) • **SIG_IGN** (valor 1, ignorar señal). Por tanto, las 3 posibilidades que tiene un proceso cuando recibe una señal son: • Terminar su ejecución (**SIG_DFL**) • Ignorar la señal (**SIG_IGN**), y todas las que queden pendientes • Ejecutar una rutina o función de usuario En la imagen de cada proceso tenemos una tabla que almacena la dirección de la rutina que maneja cada una de las posibles señales. La llamada **signal**, lo que hace es cambiar el valor de una de dichas señales. Cuando se produce una señal que el proceso quiere gestionar se realiza un cambio de contexto y se ejecuta la rutina asociada con la señal, inicializando al valor por defecto la entrada correspondiente a la señal en el array de la imagen del proceso antes de llevar a cabo esa ejecución. Cuando se vuelva de la rutina manejadora se continuará ejecutando el proceso desde donde estaba, imitando el comportamiento de una llamada a una función de usuario. Cuando se recibe una señal, si no se utiliza **SIG_IGN**, se bloquea la recepción de señales y se ejecuta la función manejadora. En cuanto ésta termine su ejecución, se retorna al punto donde se había detenido la ejecución y se habilita de nuevo la recepción de señales. Si había alguna señal en la cola esperando ser atendida se atenderá inmediatamente. IMPORTANTE: Después de recibir una señal hay que reactivar la captura para que siga siendo efectiva. De no hacerlo, al recibir la siguiente señal se mataría el proceso, ya que la rutina asociada después de haber recibido la primera señal será la rutina por defecto (**SIG_DFL**). Esto puede hacerse incluyendo en el propio manejador la llamada a signal. ATENCION: en Linux no es necesario reactivar la captura, ya que la función manejadora permanece instalada después de haber atendido a una señal. Cuando el proceso ejecuta la rutina por defecto al recibir una señal, si es debido a algún error, el kernel escribe una imagen del proceso ("core") que contiene valores acerca de la ejecución del programa en el momento en que ocurrió el error. En general, hay dos tipos de señales: • Señales que causan terminación de un proceso: pueden ser el resultado de un error irrecuperable o el haber pulsado CTRL+C desde el teclado. La mayoría de las señales provocan la terminación del proceso que las recibe si no se ejecuta alguna acción. • Señales que no causan terminación de un proceso: algunas señales provocan que el proceso que las recibe se detenga (CTRL+Z) o las ignore. La función **signal()** permite que todas las señales, excepto la **SIGKILL** y **SIGSTOP**, pueden capturarse, ser ignoradas o generen una interrupción. Para algunas llamadas al sistema, si se recibe una señal capturada mientras dichas llamadas se están ejecutando, la llamada se reiniciará automáticamente. Estas llamadas son: read(), write(), sendto(), recvfrom(), sendmsg() y recvmsg() en un canal de comunicaciones o dispositivo lento y durante un ioctl() o wait(). Sin embargo, si las llamadas ya han terminado y grabados los cambios no se reinician, pero retornan un éxito parcial (por ejemplo, un contador de bytes leidos). Cuando un proceso que tiene instalados manejadores de señales crea procesos hijos con fork(), éstos heredan las señales. Todas las señales capturadas pueden resetearse a su acción por defecto mediante una llamada a la función **execve()**; las señales ignoradas permanecen ignoradas en el proceso hijo. VALORES RETORNADOS: En caso de éxito, se retorna la acción anterior. En caso contrario, se retorna **SIG_ERR** y se establece la variable global **errno**. ERRORES:La función **signal()** fallará en las siguientes situaciones: • [EINVAL] El número de señal sig no es válido. • [EINVAL] Se intenta ignorar o capturar las señales **SIGKILL** o **SIGSTOP**.

signal vs sigaction

What is the difference between sigaction and signal?

The signal(2) function is the oldest and simplest way to install a signal handler but it's deprecated. There are few reasons and most important is that the original Unix implementation would reset the signal handler to it's default value after signal is received. If you need to handle every signal delivered to your program separately like handling SIGCHLD to catch a dying process there is a race here. To do so you would need to set to signal handler again in the signal handler itself and another signal may arrive before you cal the signal(2) function.

Establishing a Signal Handler

The following example demonstrates the use of **sigaction()** to establish a handler for the **SIGINT** signal.

```
#include <signal.h>

static void handler(int signum){
    /* Take appropriate actions for signal delivery */
}

int main(){
    struct sigaction sa;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART; /* Restart functions if
                                 interrupted by handler */
    if (sigaction(SIGINT, &sa, NULL) == -1)
        /* Handle error */;

    /* Further code */
}
```

The recommended way of setting signal actions: sigaction

(Fuente: All about linux signals – Linux Programmig Blog)

The sigaction() function is a better way to set the signal action. It has the prototype:

```
int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact);
```

As you can see you don't pass the pointer to the signal handler directly, but instead a struct sigaction object. It's defined as:

```
struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

For a detailed description of this structure's fields see the sigaction() manual page. Most important fields are:

sa_handler - This is the pointer to your handler function that has the same prototype as a handler for signal(2).

sa_sigaction - This is an alternative way to run the signal handler. It has two additional arguments beside the signal number where the siginfo_t * is the more interesting. It provides more information about the received signal, I will describe it later.

sa_mask - allows you to explicitly set signals that are blocked during the execution of the handler. In addition if you don't use the SA_NODEFER flag the signal which triggered will be also blocked.

sa_flags - allow to modify the behavior of the signal handling process. For the detailed description of this field, see the manual page. To use the sa_sigaction handler you must use SA_SIGINFO flag here.

What is the difference between signal(2) and sigaction(2) if you don't use any additional feature the later one provides?
The answer is: portability and no race conditions.

The issue with resetting the signal handler after it's called doesn't affect sigaction(2), because the default behavior is not to reset the handler and blocking the signal during it's execution. So there is no race and this behavior is documented in the POSIX specification.

Another difference is that with signal(2) some system calls are automatically restarted and with sigaction(2) they're not by default.

Example use of sigaction()

See example of using sigaction() to set a signal handler with additional parameters.
You can try to run it and do kill PID to see what happens.

```
/* Example of using sigaction() to setup a signal handler with 3 arguments
 * including siginfo_t.
 */
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
   // Atención: Evitar el uso de printf en la función manejadora pues no se garantiza su
   // correcto funcionamiento dentro de ellas. Usar write en su lugar. En este caso
   // funciona porque es lo único que hace la función manejadora. En la man page de
   // signal(2) hay una relación de funciones y llamadas que pueden ser llamadas desde la
   // función manejadora sin problemas, printf no es una de ellas.

   printf ("Sending PID: %ld, UID: %ld\n", (long)siginfo->si_pid, (long)siginfo->si_uid);
}

int main (int argc, char *argv[])
{
   struct sigaction act;

   memset (&act, '\0', sizeof(act));

   // Use the sa_sigaction field because the handles has two additional parameters

   act.sa_sigaction = &hdl;

   //The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler.

   act.sa_flags = SA_SIGINFO;

   if (sigaction(SIGTERM, &act, NULL) < 0) {
      perror ("sigaction");
      return 1;
   }

   while (1)
      sleep (10);

   return 0;
}
```

In this example we use the three arguments version of signal handler for SIGTERM.

Without setting the **SA_SIGINFO** flag we would use a traditional one argument version of the handler and pass the pointer to it by the sa_handler field. **It would be a replacement for signal(2)**.

In the signal handler we read two fields from the **siginfo_t *siginfo** parameter to read the sender's PID and UID. This structure has more fields, I'll describe them later.

The sleep(3) function is used in a loop because it's interrupted when the signal arrives and must be called again.

SA_SIGINFO handler

In the previous example **SA_SIGINFO** is used to pass more information to the signal handler as arguments.

We've seen that the siginfo_t structure contains **si_pid** and **si_uid** fields (PID and UID of the process that sends the signal), but there are many more. They are all described in sigaction(2) manual page.

On Linux only **si_signo** (signal number) and **si_code** (signal code) are available for all signals.

Presence of other fields depends on the signal type.

Some other fields are:

- **si_code -** Reason why the signal was sent. It may be SI_USER if it was delivered due to kill(2) or raise(3), SI_KERNEL if kernel sent it and few more. For some signals there are special values like ILL_ILLADR telling you that SIGILL was sent due to illegal addressing mode.
- For SIGCHLD fields **si_status**, **si_utime**, **si_stime** are filled and contain information about the exit status or the signal of the dying process, user and system time consumed.
- In case of SIGILL, SIGFPE, SIGSEGV, SIGBUS **si_addr** contains the memory address that caused the fault.

Compiler optimization and data in signal handler

Let's see the following example:

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

static int exit_flag = 0;

static void hdl (int sig){
   exit_flag = 1;
}

int main (int argc, char *argv[]){
   struct sigaction act;

   memset (&act, '\0', sizeof(act));
   act.sa_handler = &hdl;
   if (sigaction(SIGTERM, &act, NULL) < 0) {
      perror ("sigaction");
      return 1;
      }

   while (!exit_flag)
      ;

   return 0;
}
```

What it does? It depends on compiler optimization settings.

Without optimization it executes a loop that ends when the process receives SIGTERM or other sgnal that terminates the process and was not handler.

When you compile it with the -O3 gcc flag it will not exit after receiving SIGTERM.

Why? because whe while loop is optimized in such way that **the exit_flag variable is loaded into a processor register once and not read from the memory in the loop**.

The compiler isn't aware that the loop is not the only place where the program accesses this variable while running the loop.

In such cases - modifying a variable in a signal handler that is also accessed in some other parts of the program you must remember to instruct the compiler to always access this variable in memory when reading or writing them.

You should use the volatile keyword in the variable declaration:

```
static volatile int exit_flag = 0;
```

After this change everything works as expected.

Llamada	Descripción
Includes: #include <signal.h> #include <sys/types.h> **Declaración:** int kill(pid_t pid, int sig) **Uso:** kill (pid,señal)	Envía una señal al proceso o grupo de procesos especificado. Recibe el pid del proceso destinatario y la señal enviada. Si el pid es > 0, la señal se envía al proceso cuyo PID es el especificado. Si el pid = 0, la señal se envía a todos los procesos cuyo ID de grupo (GID) es el mismo que el GID del proceso que envía, para lo que dicho proceso tiene permiso. Esto es una variante de killpg(). Si el pid = -1, si el usuario tiene permisos de superusuario, la señal se envía a todos los procesos, exceptuando los procesos del sistema (los que tienen establecido el flag P_SYSTEM), el proceso cuyo PID es 1 (proceso INIT) y al proceso que envía la señal. Si el usuario no es el superusuario, la señal se envía a todos los procesos con el mismo UID, exceptuando como siempre, al proceso que envío la señal. No se genera error si se consigue señalar algún proceso. Por compatibilidad con UNIX System V, si pid < -1 (negativo pero distinto de -1), la señal se enviará a todos los procesos cuyo ID de grupo (GID) es igual al valor absoluto del pid especificado. Esto es una variante de killpg(). La señal sig es cualquiera de las que hemos descrito antes. Si especificamos 0 como señal se realiza una comprobación de errores, pero no se envía ninguna señal. Esto puede ser de utilidad para comprobar la validez del pid especificado. Normalmente, la señal que se envía es SIG_KILL, para "matar" procesos, ya que esta señal no puede capturarse ni ignorarse. Si un proceso recibe una señal que no está esperando se mata al proceso. Sólo podemos enviar señales a procesos del mismo usuario (mismo "uid"), excepto el superusuario. Se puede matar un proceso en "foreground" o en "background". Hay una excepción con la señal SIGCONT, la cual puede siempre enviarse a cualquier proceso hijo (descendiente en general). VALORES RETORNADOS: En caso de éxito, se retorna 0. En caso contrario, se retorna -1 y se establece la variable global errno. ERRORES: La función kill() fallará y no se enviará ninguna señal si: [EINVAL] El número de señal sig no es válido. [ESRCH] No existe el proceso que hemos especificado.
Includes: #include <signal.h> **Declaración:** int alarm(int seconds) **Uso:** residual = alarm (seg)	Cuando un proceso hace esta llamada lo que está pidiendo al sistema es que le envíe una señal SIG_ALARM transcurridos los "n" segundos que acepta esta función como argumento. Por ello, antes de realizar esta llamada, el proceso debe estar preparado para recibir señales, especificando un "signal" delante. Si ejecutamos un nuevo "alarm", antes de que se haya finalizado el anterior, éste se cancela. Retorna la cantidad de tiempo que falta desde el momento actual hasta el momento en que se disparará la alarma.
Includes: #include <signal.h> **Declaración:** void pause(void) **Uso:** pause()	Suspende al proceso hasta la siguiente señal. Hace lo mismo que "wait", con la diferencia de que no espera que llegue un proceso sino una interrupción (señal que hemos capturado).

Ejemplos con signal (deprecated - Modificarlos para usar sigaction)

Estos ejemplos usan signal() en vez de sigaction(), pero aunque están deprecated, deben funcionar por compatibilidad y sobre todo sirven para aprender el funcionamiento de las señales. Se recomienda cambiar signal() por sigaction() cuando sea posible.

Ejemplo:

Envío de señales: SIGKILL() - El padre deja vivir al hijo 5 segundos y después lo mata

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

int main(void) {
int ret;
ret = fork();
if (ret>0) { // parent
   int tmp; // ¿Que ventajas tiene declarar la variable aqui?
   sleep(5);
   tmp = kill(ret,SIGKILL); // send suspend
   printf ("Parent Send signal %d to child Return:%d\n",SIGKILL,tmp);
   exit(0);
   }
else
   if (ret==0) { // child
      int i; // ¿Que ventajas tiene declarar la variable aqui?
      for (i=0; ;++i) {
           printf ("Child Ticking %d\n",i);
           }
      }
return 0;
}
```

Ejemplo:

¿Suicidio? ¿Cual es la salida?

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

int main(void) {
  printf ("Antes de enviar la señal ...");
  kill (getpid(),SIGKILL);
  printf ("Después de recibir la señal ...");
  return 0;
}
```

Y ahora...cual será la salida. ¿Como se evita este resultado?.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
int main(void) {
  printf ("Antes de enviar la señal ...");
  kill (getpid(),SIGUSR1);
  printf ("Después de recibir la señal ...");
  return 0;
}
```

Ejemplo:

Vamos a manejar la señal SIGUSR1.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret;
  signal(SIGUSR1,MiManejador); // Establece el manejador
  printf ("Sending Signal: %d\n",SIGUSR1);
  kill (getpid(),SIGUSR1);
  return 0;
}

void MiManejador (int signal) {
  printf("Llega una señal llamada: %d\n",signal);
}
```

Ejemplo:

Señales capturadas y no capturadas

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret;
  signal (SIGUSR1,MiManejador); // Set a signal Handler
  printf ("Sending Signal: %d\n",SIGUSR1);
  kill (getpid(),SIGUSR1);
  kill (getpid(),SIGUSR1); // Some OLD UNIX OS's will reset handler!
  printf ("Bye..Bye...\n");
  return 0;
}

void MiManejador(int signal) {
  printf("Llega una señal called: %d\n",signal);
}
```

Ejemplo:

Señales anidadas

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador (int signal);
int GCount=0;

int main(void) {
  int ret;
  signal (SIGUSR1,MiManejador); // Set a signal Handler
  printf ("Sending Signal: %d\n",SIGUSR1);
  // Enviamos una señal: se deshabilita el manejador de interrupciones
  // y se ejecuta la función manejadora
  kill (getpid(),SIGUSR1);
  printf("Bye..Bye...\n");
  return 0;
}

void MiManejador(int signal) {
  ++GCount;
  printf("PID:%d GCount=%d\n",getpid(),GCount); // Siempre imprime 1 !!
  // Enviamos otra señal, pero el gestor de interrupciones está deshabilitado
  kill(getpid(),SIGUSR1);
  printf("Descontando ..."); // Esto se ejecuta antes de gestionar la nueva señal
  --GCount;
  }

¿Cual será la salida? Siempre 1
```

Ejemplo:

Reseteando el manejador de señales al valor por defecto

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret;
  signal (SIGUSR1,MiManejador);
  kill (getpid(),SIGUSR1);
  signal (SIGUSR1,SIG_DFL); // set signal handler to default
  kill (getpid(),SIGUSR1);
  printf("Bye..Bye...\n"); return 0;
}

void MiManejador(int signal) {
printf ("Llega una señal....\n");
}
```

Ejemplo:

Ignorando la señal

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

int main(void) {
  int ret;
  signal(SIGUSR1,SIG_IGN); // Ignoramos la señal
  kill(getpid(),SIGUSR1);
  printf("Bye..Bye...\n");
  return 0;
}
```

Ejemplo:

Signal_Fork. Los manejadores de señales permanecen a través de un fork (se heredan).

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret;
  signal(SIGUSR1,MiManejador); // Parent set signal handler
  if ((ret=fork())==0) { // Child
     int i;
     for (i=0;i<10;++i) {
          sleep(1);
          printf("Child %d\n",i);
          }
     exit(0);
     }
  // Parent Sends Signal
  sleep(3);
  kill (ret,SIGUSR1); // Send to Child
  wait (NULL);
  printf ("Bye..Bye...\n"); return 0;
}

void MiManejador(int signal) {
  printf("Llega una señal....\n");
}
```

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret; char buf[100];
  signal(SIGUSR1,MiManejador); // Parent set signal handler
  if (fork()==0) { // Child
     sleep(3);
     kill(getppid(),SIGUSR1);
     exit(0);
     }
  printf ("Enter a Line:");
  fflush(stdout);
  ret = read(0,buf,100);
  buf[ret]=0;
  printf ("Line: %s\n",buf); return 0;
}

void MiManejador(int signal) {
  printf("Llega una señal....\n");
}
```

¿Cual es la salida ahora?

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret; char buf[100];
  signal (SIGUSR1,MiManejador); // Parent set signal handler
  if (fork()==0) { // Child
     sleep(3);
     kill(getppid(),SIGUSR1);
     exit(0);
     }
  sleep(10);
  printf("Bye..Bye...");
  return 0;
}

void MiManejador(int signal) {
  printf("Llega una señal....\n");
}
```

Asegúrate de entender la salida de este programa.

Ejemplo:

```
#include <signal.h>

int main() {
  printf ("Programamos la captura de señales SIGALARM\r\n");
  signal(SIGALARM,gestionAlarma);
  printf ("Programamos una auto-alarma para que nos avise transcurridos 3 segundos\r\n");
  alarm(3);
  pause();
  printf ("Después de haber recibido alarma, continuamos la ejecución normal ...\r\n");
}

void gestionAlarma() {
  printf("Señal SIGALARM recibida!\r\n");
}
```

Si en este ejemplo comentamos la línea de "signal" comprobaremos que no se ejecuta la última línea. Esto es así porque por defecto, el sistema utiliza el manejador SIG_DFL que hace un exit() con el número de la señal recibida.

```
#include <signal.h>

int main() {
  printf ("Programamos la captura de señales SIGALARM\r\n");
  /* signal(SIGALARM,alarma); */
  printf ("Programamos una auto-alarma para que nos avise transcurridos 3 segundos\r\n");
  alarm(3);
  pause();
  printf ("Después de haber recibido alarma, continuamos la ejecución normal ...\r\n");
}

void gestionAlarma() {
  printf("Señal SIGALARM recibida!\r\n");
}
```

Por otro lado, como ya hemos comentado, debemos volver a activar la recepción de señales; de lo contrario, si llega otra señal mataría el proceso. Sin embargo, en en algunas implementaciones de Linux no ocurre esto. Podemos probarlo programando de nuevo otra alarma sin captural la señal SIGALARM.

```
#include <signal.h>

int main() {
  printf ("Programamos la captura de señales SIGALARM\r\n");
  signal(SIGALARM,alarma);
  printf ("Programamos una auto-alarma para que nos avise transcurridos 3 segundos\r\n");
  alarm(3);
  pause();
  printf ("Después de haber recibido alarma, programamos otra alarma de 3 seg.\r\n");
  alarm(3);
  pause();
  printf ("En POSIX esta línea nunca se ejecutará porque el  proceso se habrá matado al llegar
           la seguna señal\r\n");
}

void gestionAlarma() {
  printf("Señal SIGALARM recibida!\r\n");
}
```

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret; int i;
  signal(SIGALRM,MiManejador); // Parent set signal handler
  alarm(3); // Set up an alarm
  for(i=0;;++i) {
    printf("Ejecutando....%d\n",i);
    }
}

void MiManejador(int signal) {
  printf ("Llega una señal....\n");
  sleep(2);
```

```
  exit (0);
}
```

¿Que modificación le haríais para que generase la alarma cada 3 segundos?

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
void MiManejador(int signal);

int main(void) {
  int ret; int i;
  signal(SIGALRM,MiManejador); // Parent set signal handler
  alarm(3);
  for (i=0;;++i) {
      printf ("Ejecutando....%d\n",i);
      }
}

void MiManejador(int signal) {
  printf ("Llega una señal....\n");
  sleep(2);
  alarm (3);
}
```

Ejemplo:

Signal_SIGCHILD. Una señal muy utilizada es SIG_CHILD, que es enviada por todo proceso hijo a su padre en el mismo instante que realiza exit. De esta manera, el padre sabe que su hijo termina.

En este ejemplo, creamos un proceso hijo, el cual terminará a los 3 segundos retornando el valor 5 (exit(5)). Mientras tanto, el padre espera a que termine el hijo, pero en este caso sin utilizar wait, sino la señal SIGCHILD.

```
#include <signal.h>
int status;

int main() {
  int pid;
  signal(SIGCHILD,finhijo);
  pid = fork();
  if (pid==0) { // Código del proceso Hijo
     sleep(3);
     exit(5);
     }
  pause();
  printf ("El hijo %d ha finalizado con el valor: %d\r\n", status >> 8);
}
```

Ejemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
static void ChildExitHandler(int signal);

int main(void) {
  signal(SIGCHLD,ChildExitHandler);
  if (fork()==0) {
     sleep(3);
     printf("Child Dying...\n");
     exit(0);
     }
  for(;;) {
    // parent is doing something
    }
}

static void ChildExitHandler(int signal) {
  wait (NULL); // Call Wait
  printf ("Child Exited....\n");
}
```

¿Que ocurre si no se ejecuta la llamada a wait()?