Llamadas al sistema para la gestión de Señales

C Prácticas

Tabla de contenido

Ejemplo c041.c: Un ejemplo complejo
Señales de tiempo real

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

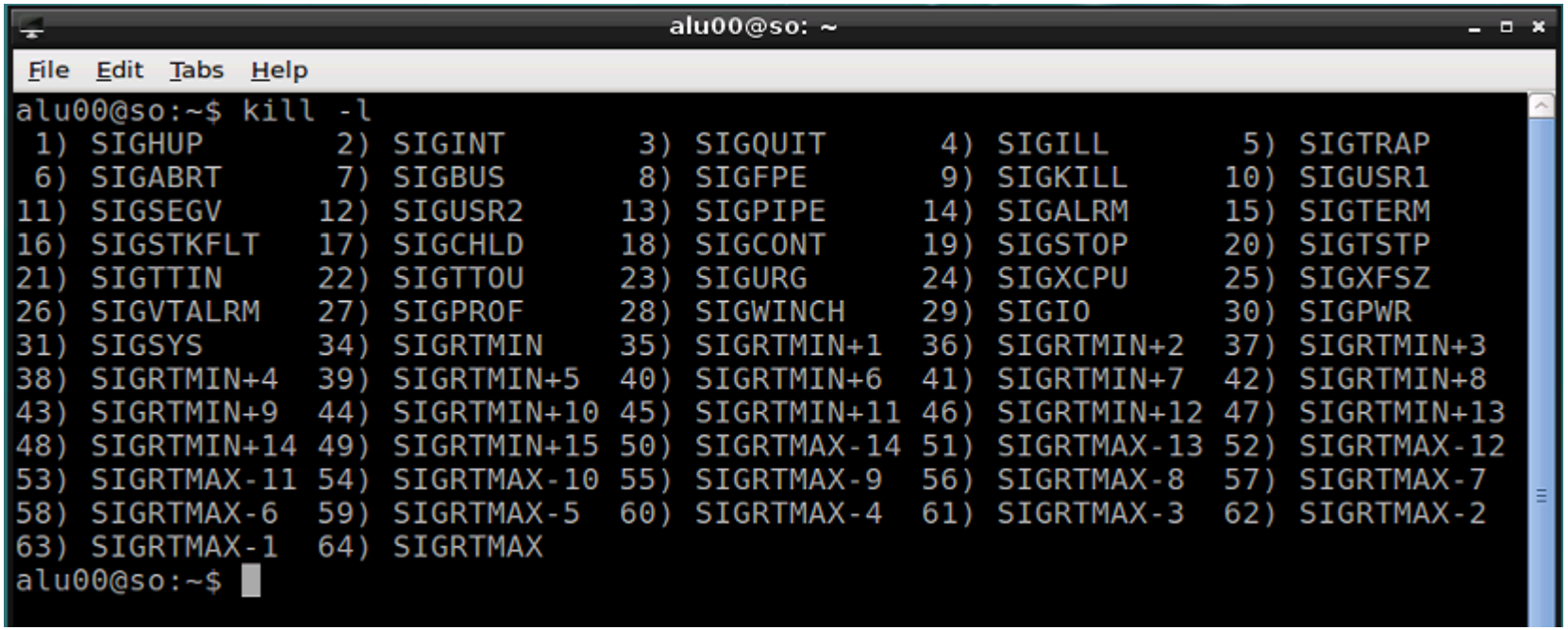
La orden Kill

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.



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 es un puntero a una estructura donde se define la nueva función a la que se debe transferir el control al recibir la señal
3. El tercero es un puntero a una estructura con el antiguo manejador establecido.

La estructura struct sigaction tiene dos miembros muy importantes:

```
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()**, para añadir un conjunto de señales se utiliza **sigfillset()**.
- **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**.

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**.

```
int main(int argc, char *argv[])
{
   signal(SIGALRM, ALARMhandler);
   ...

}

void  ALARMhandler(int sig)
{
  signal(SIGALRM, SIG_IGN);          /* ignore this signal       */
  ...
  signal(SIGALRM, ALARMhandler);     /* reinstall the handler    */
}
```

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

signal

LLamada	Descripción
Includes:#include <signal.h>**Declaración:**void (* signal(int **sig**, void(***func**) (int)))(int)**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 pendientesEjecutar una rutina o función de usuarioEn 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.ATENCION: Además la utilización de sigaction en vez de signal deja también activada la función manejadora.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**, puedan 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

Hay algunas diferencias entre usar signal o sigaction. Aquí comentan el tema. Lo resumimos también a continuación.

Usa siempre sigaction() en vez de signal() a no ser que tengas razones para ello.

La interfaz de signal() tiene a su favor que está definida en el estandar, pero tiene algunas características que hacen que intentenmos evitarla.

1. signal() no necesariamente bloquea la llegada de otras señales mientras se está en el manejador. sigaction() puede bloquearlas hasta que termine el manejador actual.
2. siganl() habitualmente resetea a la acción por defecto (SIG_DFL) para casi todas las señales. Esto significa que hay que reinstalar el manejador en el propio manejador.
3. El comportamiento exacto de signal() varia entre sistemas, lo que es permitido por el estandar.
4. Según la man page de signal() su comportamiento es indefinido en procesos multi-hilo.

Estas razones hacen que sea mejor usar sigaction(), aunque su interfaz sea más complejo.

El siguiente ejemplo muestra como usar sigaction() para establecer el manejador de la señal SIGINT.

Ejemplo: c040.c

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

static void manejadoraA (int sig){
    printf("Recibida señal %d\n",sig);
}

static void manejadoraB (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 ("Recibida señal %d y de PID: %ld, UID: %ld\n", sig, (long)siginfo->si_pid, (long)siginfo->si_uid);
}

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

    printf("Mi PID = %d\n",getpid());

    memset (&act, '\0', sizeof(act)); //Garantizamos que la estructura está a cero. Se puede obviar, el compilador la inicializará.

    //OPCION A
        // Use the sa_handler field if  the handles does not have two additional parameters
        act.sa_handler = &manejadoraA;
    //--------------

    //OPCION B
        // Use the sa_sigaction field because the handles has two additional parameters.
        // The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler.
           act.sa_sigaction = &manejadoraB;
           act.sa_flags = SA_SIGINFO;
    //--------------

    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror ("Error en sigaction al establecer la función manejadora.");
        return 1;
    }

    while (1)
```

```
        sleep (10);

    return 0;
}
```

En este ejemplo tenemos dos opciones para instalar el manejador. La opción A, que es la que está sin comentar utiliza la función manejadoraA que simplemente muestra un texto con la señal que llega. Si queremos obtener más información como por ejemplo el PID y el UID del proceso que envía debemos usar la opción B. Entonces la función manejadora recibirá una estructura siginfo_t con la información adicional, que es la que muestra en su printf.

En el ejemplo utilizamos un bucle sin fin donde ponemos a dormir al proceso durante 10 segundos y la señal que se captura es SIGINT. Como hemos dicho sigaction reestablece el manejador automáticamente. La señal SIGINT es la que se envía cuando se pulsa CTRL-C para matar al proceso. Como la capturamos y por tanto no morimos, sino que sacamos un printf por pantalla, entonces para poder matar este ejemplo debemos usar el PID que nos muestra para enviar una señal SIGKILL desde otro terminal a dicho proceso.

Optimización del compilador y datos en el manejador

Supongamos el siguiente código

```
#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;
}
```

Si este código lo compilamos usando la opción de optimización -O3 del compilador gcc, podemos tener problemas.

Si nos fijamos la condición del bucle está en una variable que se modifica en el manejador, pero esto no lo sabe el compilador. Por tanto si le decimos que optimice lo que hará sería coger esta variable y colocarla en un registro del micro y comprobará en el bucle contra el registro y no contra el valor en memoria modificado en el manejador. En estos casos es necesario decirle al compilador que la variable es volatil, con lo que no la coloca en el registro.

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

Ejemplo c041.c: Un ejemplo complejo

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // alarm()
#include <string.h>
#include <signal.h> // sigaction(), sigsuspend(), sig*()
//#include <sys/siginfo.h>

//Declaracion de funciones
void handle_signal(int signal);  //Función manejadora de señales
void handle_sigalrm(int signal); //Función manejadora de las alarmas
void do_sleep(int seconds);      //Función para enviar a dormir al proceso

int ContHUP;
int ContUSR1;
int ContRTMIN;
const int SLEEP_TIME=10;

//Esta función nos permitirá ver la máscara de señales.
void print_sigset_t(sigset_t *set)
{
 int i;

 i = SIGRTMAX;
 do {
 int x = 0;
 i -= 4;
```

```
 if (sigismember(set, i+1)) x |= 1;
 if (sigismember(set, i+2)) x |= 2;
 if (sigismember(set, i+3)) x |= 4;
 if (sigismember(set, i+4)) x |= 8;
 printf("%x", x);
 } while (i >= 4);
 printf("\n");
}

/* Compilarlo y ejecutarlo
 *
 * Mostrará el PID, que puede ser usado desde otro terminal para enviarle señales
 *     $ kill -HUP <pid>
 *     $ kill -USR1 <pid>
 *     $ kill -ALRM <pid>
 *
 * Termina el programa con CTRL-C ( = SIGINT) o enviando SIGKILL, SIGTERM
 */
int main() {
    //Primero vemos un ejemplo de cómo se ve un conjunto de señales en un sigset_t
 sigset_t set;
 sigemptyset(&set);
       print_sigset_t(&set);
     /* add signals to set */
     sigaddset(&set, SIGINT);  printf("+ SIGNINT\n"); print_sigset_t(&set);
     sigaddset(&set, SIGHUP);  printf("+ SIGHUP\n");  print_sigset_t(&set);
     sigaddset(&set, SIGALRM); printf("+ SIGALRM\n"); print_sigset_t(&set);
 sigemptyset(&set);
    print_sigset_t(&set);


    ContHUP=0;
    ContUSR1=0;
    ContRTMIN=0;

     // Declaramos la esttructura sigaction
    struct sigaction sa;

    print_sigset_t(&sa.sa_mask);

    // Mostramos el pid para poder enviar señales desde otro shell
    printf("Mi pid es: %d\n", getpid());

    // Establecemos el manegador de las señales.
    sa.sa_handler = &handle_signal;

    // Indicamos el flag de restart, que reinicia el setup de la señal si es posible
    sa.sa_flags = SA_RESTART;
```

```c
    // Bloquea cualquier señal durante el manejador
    sigfillset(&sa.sa_mask);

    // Intercepts SIGHUP, SIGURSR1 and SIGINT
    if (sigaction(SIGRTMIN, &sa, NULL) == -1) {
        perror("Error: NO se pudo establecer el manejador para SIGRTMIN"); // No debería ocurrir
    }
    else{
        printf("SIGHUP - Manejador establecido\n");
    }


    if (sigaction(SIGHUP, &sa, NULL) == -1) {
        perror("Error: NO se pudo establecer el manejador para SIGHUP"); // No debería ocurrir
    }
    else{
        printf("SIGHUP - Manejador establecido\n");
    }

    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("Error: No se puedo establecer el manejador para SIGUSR1"); // No debería ocurrir
    }
    else{
        printf("SIGUSR1 - Manejador establecido\n");
    }


    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Error: No se pudo establecer el manejador para SIGINT"); // No debería ocurrir
    }
    else{
        printf("SIGINT - Manejador establecido\n");
    }

    // Intentamos manejar SIGKILL que es la señal que forzará la muerte del proceso
    if (sigaction(SIGKILL, &sa, NULL) == -1) {
        perror("No se pudo establecer el manejador para SIGKILL"); // Siempre ocurrirá
        printf("Nunca se te permitirá manejar SIGKILL ...\n");
    }

    //Entramos en bucle a dormir
    for (;;) {
        printf("\n(Main) Durmiendo durante ~%d seconds\n",SLEEP_TIME);
        do_sleep(SLEEP_TIME); // Later to be replaced with a SIGALRM
        printf("\n(Main) Me he despertado.\n");
    }
}

//Definición del manejador de las señales excepto SIGARLM (que tiene el suyo propio)
```

```c
void handle_signal(int signal) {
    const char *signal_name;
    char buffer[200];

    sigset_t pending; //Definimos un sigset para las que tenemos pendientes.

    // Deteriminamos que señal estamos manejando
    switch (signal) {
        case SIGHUP:
            ContHUP++;
            signal_name = "SIGHUP";
            sprintf(buffer,"    Capturada SIGHUP (#%d) (signal %d - %s)\n", ContHUP, signal,signal_name);
            write(STDERR_FILENO, buffer,  strlen(buffer));
            break;
        case 34:
            ContRTMIN++;
            signal_name = "SIGRTMIN";
            sprintf(buffer,"    Capturada SIGRTMIN (#%d) (signal %d - %s)\n", ContRTMIN, signal,signal_name);
            write(STDERR_FILENO, buffer,  strlen(buffer));
            break;
        case SIGUSR1:
            ContUSR1++;
            signal_name = "SIGUSR1";
            sprintf(buffer,"    Capturada SIGUSR1 (#%d) (signal %d)\n", ContUSR1,  signal);
            write(STDERR_FILENO, buffer,  strlen(buffer));
            break;
        case SIGINT:
            sprintf(buffer,"    Capturada SIGINT, saliendo ahora (signal %d)\n", signal);
            write(STDERR_FILENO, buffer,  strlen(buffer));
            exit(0);
        default:
            sprintf(buffer,"    Capturada señal desconocida: %d\n", signal);
            write(STDERR_FILENO, buffer, strlen(buffer));
            //fprintf(stderr, "Caught wrong signal: %d\n", signal);
            return;
    }

    sprintf(buffer, "    Capturada %s, a dormir durante ~%d seconds\n    Intenta enviar otra señal SIGHUP / SIGINT / SIGALRM (o más)
entre tanto\n", signal_name,SLEEP_TIME);
    write(STDOUT_FILENO, buffer,  strlen(buffer));
    /*
        Todas las señales están bloqueadas durante este manejador
        Cuando durante el manejador llegan nuevas señales, sólo se marca
        como pendiente la primera de cada tipo de señal, el resto del mismo
        tipo se pierde, no se encolan.

        Si enviamos al proceso HUP, INT, HUP en ese orden, se trata el primer HUP,
        se encola INT y se encola HUP. Al terminar el procesado del primer HUP,
        se lanza el procesado del siguiente HUP y finalmente el INT que causa la terminación.
```

```
        El segundo HUP es procesado antes del INT porque es del mismo tipo de que la señal
        que se estaba tratando en el manejador.

        Si enviamos al proceso HUP, USR1, INT en ese orden, se trata el primer HUP
        se detecta pendiente el USR1 pero se trata INT, que termina el proceso.

        Para que no se pierdan las distintas señales del mismo tipo que llegan
        mientras estamos en el manejador tendremos que utilizar señales en tiempo real
        que son las señales SIGRTMIN (34) y siguientes, hasta SIGRTMAX.

        Mientras que sólo puede haber encolada una de las señales normales para cada tipo,
        para las de tiempo real se encolan todas las del mimso tipo.

    */
    do_sleep(SLEEP_TIME);
    sprintf(buffer, "    (Manejador) Me despierto despues de %d segundos\n", SLEEP_TIME);
    write(STDOUT_FILENO,  buffer, strlen(buffer));

    // Bien, que es lo que me has enviado mientras dormía?
    sigpending(&pending); //Capturamos el conjunto de señales pendientes

    if (sigismember(&pending, SIGHUP)) { //Miramos si una SIGHUP está pendiente
        sprintf(buffer,"    Al menos una SIGHUP está pendiente\n");
        write(STDERR_FILENO, buffer, strlen(buffer));
    }
    if (sigismember(&pending, SIGUSR1)) { //Miramos si una SIGUSR1 está pendiente
        sprintf(buffer,"    Al menos una SIGUSR1 está pendiente\n");
        write(STDERR_FILENO, buffer, strlen(buffer));
    }
    if (sigismember(&pending, SIGRTMIN)) { //Miramos si una SIGUSR1 está pendiente
        sprintf(buffer,"    Al menos una SIGRTMIN está pendiente\n");
        write(STDERR_FILENO, buffer, strlen(buffer));
    }

    sprintf(buffer, "    (Manejador) Termino de manejar la señal %s\n\n", signal_name);
    write(STDERR_FILENO, buffer, strlen(buffer));
}

//Definición del manejador de SIGALRM
void handle_sigalrm(int signal) {
    char buffer[100];

    if (signal != SIGALRM) {
        sprintf(buffer, "    Capturada señal erronea %d\n", signal);
        write(STDERR_FILENO, buffer, strlen(buffer));
    }

    sprintf(buffer,"    Capturada SIGALRM, do_sleep() termina.\n");
    write(STDERR_FILENO, buffer, strlen(buffer));
```

```
}

void do_sleep(int seconds) {
    struct sigaction sa;
    sigset_t mask;
    char buffer[100];

    sa.sa_handler = &handle_sigalrm; // Intercept and ignore SIGALRM
    //Quitamos el handler tras la primera señal. Esto lo hace por defecto signal() pero sigaction() no.
    sa.sa_flags = SA_RESETHAND; // En este caso nos interesa, pues así en el manejador no se atiende una segunda SIGALRM
    sigfillset(&sa.sa_mask);
    sigaction(SIGALRM, &sa, NULL);

    // Obtenemos el conunto (mascara) de señales actual (las que se bloquean)
    sigprocmask(0, NULL, &mask);

    // Quitamos del conjunto (de la mascara) de señales a bloquear la SIGALRM para que llegue.
    sigdelset(&mask, SIGALRM);
    //sigdelset(&mask, SIGINT); //Con esta linea comentada, SIGINT estará bloqueada durante el manejador

    // Establecemos una alarma de los segundos indicados. Vencidos los mismos recibieremos una señal SIGALRM.
    // (Infor adicional: Las alarmas no se heredan con fork)
    alarm(seconds);

    // Esperamos los segundos indicados especificando la máscara de bloqueos (que ya no tiene a SIGALRM)
    // Suspendemos el proceso con la máscara especificada
    sigsuspend(&mask);

    sprintf(buffer, "Termina sigsuspend()\n");
    write(STDERR_FILENO, buffer, strlen(buffer));
}
```

Señales de tiempo real

Supongamos un proceso padre que genera un número determinado de procesos hijos, y éstos comunican con el padre enviándole una señal (todos la misma señal) para indicar una situación determinada en su evolución. Por tanto tenemos un proceso que puede recibir varias veces la misma señal, con lo que puede darse el caso de que estando en el manejador lleguen otras señales de los distintos hijos.

Como hemos visto, la misma señal si llega varias veces al proceso estando éste en el manejador, se tratan como si fuera sólo una. Si se utiliza el manejador para contabilizar el número de procesos que han contestado al padre, entonces podemos contabilizar menos señales de las que realmente se hayan enviado.

Usaremos las señales de tiempo real para asegurar que todas las señales de los hijos que llegan al padre son tratadas en el manejador.

Podemos utilizar el manejador del padre para contar el número de señales recibidas de los hijos y hasta que no hayan llegado todas no continua el padre. Esto lo podemos hacer colgando al padre en un bucle con un sleep() (para que no sea espera activa) saliendo del bucle cuando el contador de señales, que se incrementa en el manejador, llega al número de señales esperadas.

En este caso, si estando en el manejador llegan varias señales de los hijos, todas ellas serán tratadas como si hubiera llegado sólo úna (pues son la misma señal, mismo tipo), y por tanto el manejador incrementará sólo una vez el contador, no llegando este núnca al número de señales esperadas, por lo que el padre no saldrá del bucle y se cuelga.

Las señales de tiempo-real no se pierden, se llama siempre al manejador por cada una de ellas, el sistema operativo las encola siempre, y si no puede encolarlas da un error, cosa que no hace con las señales normales.

Para enviar una señal normal utilizaremos kill(...) para una señal en tiempo real sigqueue(...).

De igual forma, para definir los handlers de las señales utilizaremos sigaction(..) en vez de signal(...).

Definiremos una estructura para establecer la acción (manejador) y sus propiedades para la señal:

```
struct sigaction sa;
```

Cuando definamos el manejador de la señal que recibe el padre (y que ha sido enviada por el hijo) haremos

```
sa.sa_sigaction = handler;
sa.sa_flags = SA_SIGINFO;
sigaction(SIGRTMIN,&sa,NULL);
```

... donde handler es el nombre de nuestra función manejadora. Debemos definir el flag SA_SIGINFO si queremos enviar información adicional junto con nuestra señal en tiempo real.

Si os fijáis la señal que recibirá el padre será la señal SIGRTMIN que es una macro con el valor de la señal de tiempo real con valor mínimo. Existe también la SIGRTMAX con el valor máximo.

Una señal es de tiempo real si su valor está entre SIGRTMIN y SIGRTMAX inclusive. Definidos en signal.h

Por tanto nuestro envío desde el código del hijo con sigqueue(...) será con una señal entre SIGRTMIN y SIGRTMAX.

Definiremos una unión de tipo sigval para almacenar la información adicional que el hijo le envía al padre junto con la señal SIGRTMIN

```
union sigval v;
```

... siendo esta la unión :

```
union sigval {
int sival_int;
void *sival_ptr;
}
```

Cuando desde el código del hijo queramos enviar la señal haremos lo siguiente:

```
v.sival_int = getpid(); //asignamos valor al dato
sigqueue(getppid(),SIGRTMIN,v); //enviamos la señal de tiempo real
```

Os coloco el código de un manejador de ejemplo para la señal SIGRTMIN con datos adjuntos :

```
void handler (int signo, siginfo_t *info, void *other) {
  char *src;
  switch(info->si_code) {
    case SI_QUEUE: src = "sigqueue"; break;
    case SI_TIMER: src = "timer"; break;
    case SI_ASYNCIO: src = "asyncio"; break;
    case SI_MESGQ: src = "msg queue"; break;
    case SI_USER: src = "kill/abort/raise"; break;
    default: src = "unknown!"; break;
    }
  printf("Signal %d: source = %s, data = %d\n",signo, src, info->si_value.sival_int);
}
```

Si compiláis y todo va bien.... ya no se pierden señales de los hijos, con lo que solucionamos el problema.

← Entrada anterior

Entrada siguiente →

Escuela Politécnica Superior de Elche

Universidad Miguel Hernández

Miguel Onofre Martínez Rach