Señales y Alarmas en C – Linux

[Rodrigo Paszniuk](https://www.programacion.com.py/author/raps) | 2013-09-5 | [0 Comments](https://www.programacion.com.py/escritorio/c/senales-y-alarmas-en-c-linux#disqus_thread)

Una señal es un «aviso» que puede enviar un proceso a otro proceso. El sistema operativo unix se encarga de que el proceso que recibe la señal la trate inmediatamente. De hecho, termina la línea de código que esté ejecutando y salta a la función de tratamiento de señales adecuada. Cuando termina de ejecutar esa función de tratamiento de señales, continua con la ejecución en la línea de código donde lo había dibujado.

El sistema operativo envía señales a los procesos en determinadas circunstancias. Por ejemplo, si en el programa que se está ejecutando en una shell nosotros apretamos **Ctrl-C**, se está enviando una señal de terminación al proceso. Este la trata inmediatamente y sale. Si nuestro programa intenta acceder a una memoria no válida (por ejemplo, accediendo al contenido de un puntero a **NULL**), el sistema operativo detecta esta circunstancia y le envía una señal de terminación inmediata, con lo que el programa «se cae».

**Las señales van identificadas por un número entero:**

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

**No voy a detallar todas, pero sí alguna de las más interesantes para el programador:**

|  |  |  |
| --- | --- | --- |
| Nombre de señal | Nº | Propósito. Se envia al proceso cuando… |
| **SIGINT** | 2 | … se pulsa Ctrl-C |
| **SIGFPE** | 8 | … hay un error en coma flotante (ejemplo, división por cero) |
| **SIGPIPE** | 13 | … se intenta escribir en una conexión (socket, tubería, …) rota (no hay proceso leyendo al otro lado). |
| **SIGALRM** | 14 | … cuando termina un temporizador. |
| **SIGUSR1** | 16 | … el programador lo decide. Esta señal es para uso del programador. No la utiliza el sistema operativo. |
| **SIGUSR2** | 17 | … el programador lo decide. Idem a la anterior |

**Todas las señales pueden ser ignoradas o bloqueadas, a excepción de SIGSTOP y SIGKILL, que son imposibles de ignorar.**

Una limitación importante de las señales es que no tienen prioridades relativas, es decir, si dos señales llegan al mismo tiempo a un proceso puede que sean tratadas en cualquier orden, no podemos asegurar la prioridad de una en concreto. Otra limitación es la imposibilidad de tratar múltiples señales iguales: si nos llegan 14 señales SIGCONT a la vez, por ejemplo, el proceso funcionará como si hubiera recibido sólo una.

Cuando queremos que un proceso espere a que le llegue una señal, usaremos la función pause(). Esta función provoca que el proceso (o thread) en cuestión “duerma” hasta que le llegue una señal. Para capturar esa señal, el proceso deberá haber establecido un tratamiento de la misma con la función signal(). Aquí tenemos los prototipos de ambas funciones:

int pause(void);

typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler);

La función pause() no parece tener demasiada complicación: no recibe ningún parámetro y retorna –1 cuando la llamada a la función que captura la señal ha terminado. La función signal() tiene un poco más de miga: recibe dos parámetros, el número de señal que queremos capturar (los números en el sistema en concreto en el que nos encontremos los podemos obtener ejecutando “kill –l”, como ya hemos visto), y un puntero a una función que se encargará de tratar la señal especificada. Esto puede parecer confuso, así que aclaremos esto con un ejemplo:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | #include <signal.h>  #include <unistd.h>    void trapper(int);    int main(int argc, char \*argv[])  {      int i;        for(i=1;i<=64;i++)          signal(i, trapper);        printf("Identificativo de proceso: %d\n", getpid() );      pause();      printf("Continuando...\n");        return 0;  }    void trapper(int sig)  {      signal(sig, trapper);      printf("Recibida la señal: %d\n", sig);  } |

La explicación de este pequeño programa es bastante simple. Inicialmente declaramos una función que va a recibir un entero como parámetro y se encargará de capturar una señal ( trapper() ). Seguidamente capturamos todas las señales de 1 a 64 haciendo 64 llamadas a signal(), pasando como primer parámetro el número de la señal (i) y como segundo parámetro la función que se hará cargo de dicha señal (trapper). Seguidamente el programa indica su PID llamando a getpid() y espera a que le llegue una señal con la función pause(). El programa esperará indefinidamente la llegada de esa señal, y cuando le enviemos una (por ejemplo, pulsando Control+C), la función encargada de gestionarla ( trapper() ) será invocada. Lo primero que hace trapper() es volver a enlazar la señal en cuestión a la función encargada de gestionarla, es decir, ella misma, y luego saca por la salida estándar la señal recibida. Al terminal la ejecución de trapper(), se vuelve al punto donde estábamos ( pause() ) y se continua:

txipi@neon:~$ gcc trapper.c -o trapper

txipi@neon:~$ ./trapper

Identificativo de proceso: 15702

Recibida la señal: 2

Continuando...

txipi@neon:~$

**Ejemplo 2 de signal:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include<signal.h>  #include<stdio.h>  #include<stdlib.h>    //Declaramos los prototipos de funciones  void manejador(int signum);    //Variables globales  int con = 0;    //Función principal  main(int argc, char \*\*argv)  {    //Capturamos la señal SIGINT  signal(SIGINT,manejador);    printf("Ejemplo de signal\n");    //Hacemos un bucle para permitir que se hagan hasta 5 SIGINTs  while(con<=5);    }    //Funcion manejador  //Cuando se presiona un sigint se llama a esta función  void manejador (int signum){  printf("\nRecibi la señal sigint\n");  con++;  signal(SIGINT,manejador);  } |

Como podemos observar, capturar una señal es bastante sencillo. Intentemos ahora ser nosotros los emisores de señales a otros procesos. Si queremos enviar una señal desde la línea de comandos, utilizaremos el comando “kill”. La función de C que hace la misma labor se llama, originalmente, kill(). Esta función puede enviar cualquier señal a cualquier proceso, siempre y cuando tengamos los permisos adecuados (las credenciales de cada proceso, explicadas anteriormente, entran ahora en juego (uid, euid, etc.) ). Su prototipo es el siguiente:

int kill(pid\_t pid, int sig);

**Ejemplo de kill:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49 | #include <stdio.h>  #include <signal.h>  #include <sys/types.h>    //Declaramos los prototipos de funciones  void manejador(int signum);    //Variable global  int bandera = 1;    //Función principal  main(int argc, char \*\*argv)  {   //Declaramos variables   int status,pid;     //Si fork() es igual a 0 entonces es hijo   if((pid=fork())==0)   {   //Proceso hijo   printf("Soy hijo y estoy esperando una señal de mi padre, mi pid es: %d\n",getpid());   //Capturamos la señal SIGUSR1   signal(SIGUSR1,manejador);   //Utilizamos la variable bandera para que el proceso no termine   while(bandera);   //Usamos el comando kill para enviar una señal   //Se necesita el pid del padre y la señal   kill(getppid(),SIGUSR2);   }   else   {   //Proceso padre   //Capturamos la señal SIGUSR2   signal(SIGUSR2,manejador);   printf("Soy Padre, mi pid es: %d\n",getpid());   //Esperamos 3 segundos   sleep(3);   //Usamos el comando kill para enviar una señal   //Se necesita el pid y la señal   kill(pid, SIGUSR1);   //Esperamos que termine el hijo   wait(&status);   printf("Mi hijo termino con un estado: %d\n",status);   }  }    //Funcion manejador  void manejador (int signum){   //Si la señal es SIGUSR1 entonces   if(signum==SIGUSR1){   printf("Recibi una senal de mi padre %d\n",signum);   }   //Si es SIGUSR2 entonces   else   {   printf("Recibi una senal de mi hijo %d\n",signum);   }   bandera=0;  } |

**Ejemplo 2 de kill:**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | //El padre debe mandar una señal SIGUSR1 O SIGUSR2 al hijo  //La señal debe pasarse a través de la consola  #include <stdio.h>  #include <signal.h>  #include <sys/types.h>    //Prototipos  void manejador(int signum);  void manejador2(int signum);    //Variables globales  int bandera =1;  int pid,status;  char \*parametro;    main(int argc, char \*\*argv){   //Guardamos el argumento 1 en la variable parametro   parametro = argv[1];   //Capturamos las señales   signal(SIGUSR1,manejador2);   signal(SIGUSR2,manejador2);   if((pid=fork())==0){   // proceso hijo   printf("soy el hijo\n");   //Utilizamos la variable bandera para que el proceso no termine   while(bandera);   }else{   //proceso padre   //Capturamos la señal SIGINT para luego enviar la señal al hijo   signal(SIGINT,manejador);   //Esperamos que termine el hijo   wait(&status);   }  }    void manejador(int signum){   printf("%d",pid);   //Comparamos si el parametro es SIGUSR1   if(strcmp(parametro,"SIGUSR1")==0){   printf("\nmandando la seÃ±al SIGUSR1 a mi hijo\n");   //Enviamos la señal al hijo   kill(pid,10);   }else{   printf("\nmandando la seÃ±al SIGUSR2 a mi hijo\n");   //Enviamos la señal al hijo   kill(pid,12);   }    }    void manejador2(int signum){   //Mostramos la señal que fue enviada por el padre   printf("Recibi la seÃ±al %d de mi padre\n",signum);   bandera= 0;  } |

Un pequeño detalle. En algunos sistemas unix, cuando se recibe una señal y se ejecuta nuestra función, se restaura automáticamente la función por defecto. Por ello, a veces es necesario que nuestra función termine volviendo a ponerse ella misma como tratamiento de la esa señal.

**También podemos hacer que una señal se ignore, es decir, que no la trate nadie, ni nuestra función ni la de por defecto. Para ello hay que poner**

signal (numeroSenhal, SIG\_IGN);

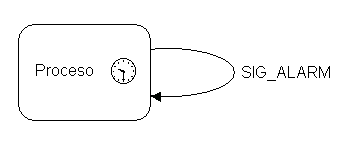
**Para restaurar el tratamiento por defecto de la señal**

signal (numeroSenhal, SIG\_DFL);

**Una utilización bastante potente de las señales es el uso de SIGALARM para crear temporizadores en nuestros programas. Con la función alarm() lo que conseguimos es que nuestro proceso se envíe a sí mismo una señal SIGALARM en el número de segundos que especifiquemos. El prototipo de alarm() es el siguiente:**

unsigned int alarm(unsigned int seconds);

En su único parámetro indicamos el número de segundos que queremos esperar desde la llamada a alarm() para recibir la señal SIGALARM.

[](http://www.programacion.com.py/wp-content/uploads/2013/09/cursoc05.gif)

**La llamada a la función alarm() generará una señal SIG\_ALARM hacia el mismo proceso que la invoca.**

El valor devuelto es el número de segundos que quedaban en la anterior alarma antes de fijar esta nueva alarma. Esto es importante: sólo disponemos de un temporizador para usar con alarm(), por lo que si llamamos seguidamente otra vez a alarm(), la alarma inicial será sobrescrita por la nueva. Veamos un ejemplo de su utilización:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include<signal.h>  #include<stdio.h>  #include<stdlib.h>    //Declaramos los prototipos de funciones  void manejador(int signum);  //Variable global  int bandera = 1;    //FunciÃ³n principal  main(int argc, char \*\*argv)  {    //Capturamos la señal SIGALRM  signal(SIGALRM,manejador);  printf("EN 10 segundos se creara una alarma\n");  //Crear alarma en segundos  alarm(10);    //Mientras bandera sea 1, no finalizar el programa  while(bandera);    }    //Funcion manejador  void manejador (int signum){  printf("\nRecibi una alarma\n");  //signal(SIGINT,SIG\_DFL);  bandera=0;  } |