Introducción a Sistemas Operativos: Comunicación entre Procesos

Clips xxx
Francisco J Ballesteros

1. Señales

Otro mecamismo de intercomunicación de procesos es la posibilidad de enviar mensajes a un proceso dado y la posibilidad de actuar cuando recibimos un mensaje en un proceso. Si estás pensando en la red o en TCP/IP, no nos referimos a eso.

UNIX permite enviar mensajes consistentes en números enteros concretos a un proceso dado. A estos mensajes se los denomina **señales** (*signals* en inglés). Cada mensaje tiene un significado concreto y tiene asociada una acción por defecto.

Cuando un proceso entra al kernel (o sale) UNIX comprueba si tiene señales pendientes y el código del kernel hace que cada proceso procese sus propias señales. Así es como funciona, aunque puedes ignorar este detalle. La recepción de una señal

- puede ignorarse,
- puede hacer que ejecute un manejador para la misma, o
- puede matar al proceso.

En breve vamos a ver qué supone esto en realidad.

El comando de shell capaz de enviar una señal a otro proceso es *kill(1)* y la llamada al sistema para hacerlo desde C es *kill(2)*. Estos son algunos de los valores según indica *kill(1)*:

```
unix$ man kill
...
             Some of the more commonly used signals:
     1       HUP (hang up)
     2       INT (interrupt)
     3       QUIT (quit)
     6       ABRT (abort)
     9       KILL (non-catchable, non-ignorable kill)
     14      ALRM (alarm clock)
     15      TERM (software termination signal)
...
```

Por ejemplo, tras ejecutar

```
unix$ sleep 3600 &
[1] 15984
unix$
```

podemos ejecutar `kill` para enviarle la señal `TERM` (simplemente "`15`") al proceso con pid `15984`:

```
unix$ kill 15984
[1]+  Terminated: 15          sleep 3600
unix$
```

Como puedes ver, por omisión, la señal `TERM` hace que UNIX mate el proceso que la recibe (salvo que dicho proceso cambie la acción por defecto para esta señal).

Incluso si un comando pide a UNIX que ignore la señal `TERM`, es imposible ignorar la señal `KILL`. Así pues,

```
unix$ kill -9 15984
```

es más expeditivo que el comando anterior, y pide a UNIX que mate el proceso con pid `15984` en cualquier caso (supuesto que tengamos permisos para hacer tal cosa, claro está).

Cuando UNIX inicia un *shutdown* (el administrador decide apagarlo, o pulsas el botón de encendido/apagado) primero envía la señal `TERM` a todos los procesos. Esto permite a dichos procesos enterarse de que el sistema está terminando de operar y podrían salvar ficheros que necesiten salvar o terminar ordenadamente si es preciso. Aquellos procesos a los que no importa esto simplemente habrán dejado que `TERM` los mate.

Pasados unos segundos, UNIX envía la señal `KILL` a todos los los procesos vivos, lo que efectivamente los mata. Puedes pensar que `TERM` significa "*por favor, muérete*" y que `KILL` es una puñalada trapera por la espalda.

Otra señal que seguramente has utilizado es `INT`. Se utiliza para interrumpir la ejecución de un proceso. Es de hecho la señal que se utiliza cuando pulsas *Control* y la tecla `c` antes de soltar *Control. Normalmente decimos "*control-c" o escribimos `^C` para representar esto.

Veámoslo. Pero primero hagamos un programa que podamos modificar luego para ver qué hacen las señales.

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

int
main(int argc, char* argv[])
{
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Este programa duerme una hora tras imprimir un mensaje informando de su *pid*. Además, hemos comprobado si `sleep` ha fallado e imprimimos un mensaje tras dormir. Si lo ejecutamos

```
unix$ signal1
pid 16056 sleeping...
```

vemos que no obtenemos el prompt del shell. El proceso está en `sleep`. Si pulsamos ahora *control-c* vemos que la ejecución del programa se interrumpe.

```
unix$ signal1
pid 16056 sleeping...
^C
unix$
```

Obtenemos igual resultado si desde otro shell ejecutamos

```
unix$ kill -INT 16056
```

El programa termina igualmente. Si en cambio le envíamos una señal `TERM`, se nos informa de que el proceso ha terminado, aunque el proceso muere igualmente.

```
unix$ signal1
pid 16102 sleeping...
```

Y ejecutando en otro shell

```
unix$ kill -TERM 16102
```

vemos que en shell original el proceso muere:

```
unix$ signal1
pid 16102 sleeping...
Terminated: 15
unix$
```

Cambiemos nuestro programa para pedirle a UNIX que la acción al recibir la señal `INT` no sea morir, sino ignorar la señal.

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

int
main(int argc, char* argv[])
{
    signal(SIGINT, SIG_IGN);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done");
    exit(0);
}
```

Esto lo hacemos utilizando *signal(3)* para pedir que acción por defecto sea ignorar la señal que indicamos (`SIGINT` simplemente es un `2`). La constante `SIG_IGN` indica que queremos ignorar dicha señal.

Ahora volvemos a ejecutar el programa en un shell y pulsar *control-c* varias veces...

```
unix$ signal2
pid 16058 sleeping...
^C^C
```

Esta vez el programa sigue ejecutando como si tal cosa. Podemos utilizar otro shell para ejecutar

```
unix$ kill  16058
```

y matar el proceso (por omisión, `kill` envía la señal `TERM`).

No es muy amable por parte del programa ignorar la señal de interrupción. Seguramente el usuario acabe frustrado sin poder interrumpir el programa y lo mate de todos modos.

Algo más habitual es utilizar *signal* para pedir que cuando se nos envíe la señal `INT` el programa ejecute un manejador para atender dicha señal. Esto es prácticamente lo que sucede con una interrupción software, pero las señales son una abstracción de UNIX para hablar con los procesos, no son interrupciones

realmente.

Esta nueva versión del programa pide a UNIX que cuando el proceso reciba la señal `INT` se ejecute la función `handleint`.

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

static void
handleint(int no)
{
    fprintf(stderr, "hndlr got %d\n", no);
}

int
main(int argc, char* argv[])
{
    signal(SIGINT, handleint);
    fprintf(stderr, "pid %d sleeping...\n", getpid());
    if(sleep(3600) != 0) {
        warn("sleep");
    }
    fprintf(stderr, "done\n");
    exit(0);
}
```

Si lo ejecutamos y pulsamos *control-c* o utilizamos `kill` desde otro shell para enviar la señal `INT`, esto es lo que sucede:

```
unix$ signal3
pid 16150 sleeping...
^C
hndlr got 2
signal3: sleep: Interrupted system call
done
unix$
```

Lo primero que vemos es que *mientras* el programa estaba dentro de `sleep`, el proceso ha recibido la señal. Considerando la llamada a `signal` que hemos hecho, UNIX ajusta la pila (de usuario) del proceso para que cuando continúe ejecutando ejecute en realidad `handleint`. Por eso vemos que lo siguiente que sucede es que `handleint` imprime su mensaje. Una vez `handleint` (el manejador de la señal) termina, su retorno hace que volvamos a llamar a UNIX. Esto lo ha conseguido UNIX ajustando la pila de usuario, nosotros no hemos hecho nada especial en el código como puedes ver. El kernel hace ahora que el programa continúe por donde estaba cuando recibió la señal. En nuestro caso estábamos dentro de `sleep` y, dado que lo hemos interrumpido, `sleep` falla y retorna `-1`. Puedes ver el mensaje de error que imprime `warn` en nuestro código justo detrás del mensaje que ha impreso el manejador. Por último, el programa escribe `done` y termina.

Es preciso tener cuidado con el código que programamos en los manejadores de señal. Dado que el programa podría estar ejecutando cualquier cosa cuando UNIX hace que se interrumpa y ejecute el manejador de la señal, sólo deberíamos utilizar funciones **reentrantes** dentro de los manejadores de señal. Decimos que una función es *reentrante* si es posible llamarla antes de que una llamada en curso termine. Por ejemplo, las funciones que utilizan variables globales (o almacenamiento *estático* en el sentido de `static` en

C) no son reentrantes. En pocas palabras, cuanto menos haga un manejador de señal mejor. La página de manual *sigaction(2)* y la de *signal(3)* suelen tener mas detalles respecto a qué funciones puede utilizarse dentro de un manejador y cuales no.

Cada vez que se llama a `signal` para una señal se cambia el efecto de dicha señal en nuestro proceso. Para hacer que `INT` recupere su comportamiento por defecto tras haber instalado un manejador, podríamos ejecutar la llamada

```
signal(SIGINT, SIG_DFL);
```

y UNIX olvidaría que antes teníamos un manejador instalado. La constante `SIG_DFL` indica que queremos que la señal en cuestión tenga la acción por defecto.

Los manejadores de señal que tenemos instalados (incluyendo si consisten en ignorar la señal) son atributos del proceso. Cada proceso tiene los suyos. Un cambio en un proceso a este respecto no afecta a otros procesos.

La página de manual *signal(3)* detalla la lista completa de señales, qué acción por defecto tienen y si se pueden ignorar o no.

Las llamadas al sistema interrumpidas por una señal suelen devolver una indicación de error y dejan `errno` al valor `EINTR` (normalmente), que significa "*interrupted*".

Pero `read` y `write` tienen un comportamiento especial. Cuando se interrumpen por el envío de una señal, UNIX re-inicia las llamadas al sistema tras la interrución (y tras la ejecución del manejador si lo hay). Naturalmente, salvo que la acción de la señal en cuestión sea matar al proceso. Eso quiere decir que nuestro programa continuaría ejecutando `read` si es que `read` se ha interrumpido, tras ejecutar el manejador que hemos instalado antes para `INT`.

Para cambiar este comportamiento, podemos utilizar *siginterrupt(3)* y pedir que una señal interrumpa las llamadas al sistema que pueden re-iniciarse tras una señal, esto es, `read` y `write` principalmente. Por ejemplo, ejecutando

```
siginterrupt(SIGINT, 1);
```

incluso `read` se interrumpirá y dejará en `errno` el valor `EINTR`. En cambio, tras

```
siginterrupt(SIGINT, 0);
```

las llamadas a `read` interrumpidas volverán a reiniciarse automáticamente.

Dos señales útiles son `USR1` y `USR2`. Suelen estar disponibles para lo que nos plazca. Un uso habitual es hacer que el programa vuelque información de depuración, por ejemplo, cuando se le envíe `USR1`, o que relea la configuración. También suele utilizarse la señal `HUP` para que el programa relea la configuración, aunque esta señal no es precisamente para eso.