11 Señales y temporización

Introducción

Como se mencionó antes, el uso de sockets UDP tiene la ventaja de ser rápido y de generar poco tráfico en la red, siempre y cuando se implemente en forma correcta. Asimismo al ser un protocolo no orientado a conexión, no se garantiza que los mensajes emitidos sean recibidos, ni tampoco hay garantía de que el orden en que se envían los paquetes sea el mismo al recibirse. De modo que el programador de la aplicación debe subsanar estas deficiencias mediante programación.

Podemos darnos cuenta fácilmente de estos problemas, simplemente colocando nuestros programas cliente y servidor lo suficientemente alejados, por ejemplo en distintas subredes. En este caso habrá situaciones en que los datagramas no llegarán a su destino, independientemente si viajan del cliente hacia el servidor o del servidor hacia el cliente.

Supongamos que el mensaje de solicitud que envía el cliente hacia el servidor se pierde, como consecuencia el proceso cliente se quedará bloqueado permanentemente en la instrucción `recvfrom`, debido a que el servidor nunca recibió nada. En casos como éste es indispensable contar con un mecanismo para salir automáticamente del bloqueo, después de un razonable tiempo de espera con la intención de volver a enviar una nueva solicitud hacia el servidor. Los mecanismos empleados en los sistemas UNIX para lograr esto último, son precisamente las señales.

Cabe aclarar que aunque existen *sockets* orientados a conexión que utilizan el protocolo TCP y evitan estos problemas, este mecanismo es recomendable en aplicaciones distribuidas fuertemente acopladas el uso del protocolo UDP.

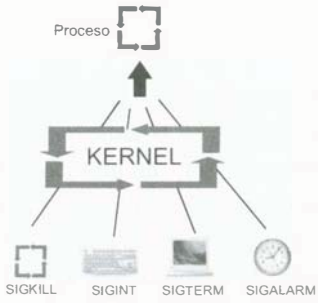
Ejemplo práctico

Una señal es una interrupción que envía el *kernel* hacia un proceso. Puede ser originada de manera asíncrona por distintos métodos, por ejemplo al oprimir una combinación de teclas, al ejecutar un comando específico de señal

en alguna terminal, cuando un proceso, al ejecutar una instrucción, le solicita al *kernel* enviarla hacia otro proceso, o simplemente cuando se ha activado algún cronómetro y se ha terminado el tiempo programado. Estos casos se representan de manera gráfica en la figura 11.1.

Figura 11.1

Las señales son recibidas por el *kernel* quien a su vez actua sobre el proceso



Existen diversas señales definidas en LINUX, las cuales pueden revisarse haciendo uso del manual de `signal` en su sección 7, como sigue:

```
man 7 signal
```

Pregunta 11.1

¿Cuántas señales están definidas en su sistema? ¿Cuáles señales sólo detienen él proceso? ¿Cuáles señales no pueden ser capturadas y provocan irremediablemente que un proceso muera?

Se puede enviar una señal desde la línea de comandos hacia un proceso del usuario mediante el comando `kill` (véase el manual).
En una terminal utilice el comando `man` para visualizar el manual del comando `ls`, posteriormente ubique mediante el comando:

```
pstree -up | less
```

el PID (identificador de proceso) correspondiente al proceso `man`. A continuación ejecute el comando:

```
kill PID
```

con lo cual se envía la señal 15 (valor por defecto) al proceso identificado por el número PID. La señal (15) `SIGTERM` le indica a un proceso que debe terminar su ejecución. Esta señal puede ser ignorada mediante una rutina de tratamiento de señal.

Lo que mencionamos en el párrafo anterior podemos ilustrarlo enviando la señal 15 al *shell (bash)* de la otra terminal. En este caso no ocurre nada debido a que el proceso shell tiene un tratamiento de la señal 15 para ignorarla. Nuestros programas pueden implementar un manejador de señal poder ignorar: las señales. Sin embargo existe una que ningún proceso puede ignorar, la señal (9) `SIGKILL`, que provoca la terminación abrupta e indiscutible del proceso y la generación de un archivo de nombre "`core`" (siempre que se tenga configurado el sistema), con información sobre el error ocurrido. Pruebe esta señal con el shell (bash) mediante el comando:

```
kill -9 PID
```

La señal `SIGINT(2)` se envía a todo proceso asociado a una terminal cuando se presionan las teclas `CTRL-C`. Su acción por defecto es terminar la ejecución del proceso que la recibe. En todos los casos una señal sólo se puede enviar a uno que tiene el mismo `UID` (identificador de usuario) del proceso que envía dicha señal.

Se puede enviar una señal a un proceso dentro de un programa haciendo uso de la función `kill`, la cual tiene la siguiente estructura:

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

En el programa 11-1 se ejecuta la función `tratar_alarma()` cada 3 segundos. Esta se ha programado para ser un manejador de la señal `SIGALRM`.

Ejecute el programa 11-1 y trate de matar el proceso con CTRL-C.

Programa 11-1
Ejecuta una función cuando llega una señal activada por un cronómetro.

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

void tratar_alarma(void)
{
   printf("Alarma activada \n");
}

int main(void)
{
1  struct sigaction act;
   sigset_t mask;

   /* especifica el manejador */
   act.sa_handler = tratar_alarma; /*función a ejecutar */
   act.sa_flags = 0; /* Ninguna acción específica */

   /* Se bloquea la señal 3 SIGQUIT */
2  sigemptyset(&mask);
3  sigaddset(&mask, SIGQUIT);
4  sigprocmask(SIG_SETMASK, &mask, NULL);
```

continúa

```
5  sigaction(SIGALRM, &act, NULL);

   for(;;)
   {
6     alarm(3); /* se arma el temporizador */
7     pause(); /* se suspende el proceso hasta que se reciba la
señal */
   }
}
```

Línea 1 La estructura `sigaction` está formada como sigue:

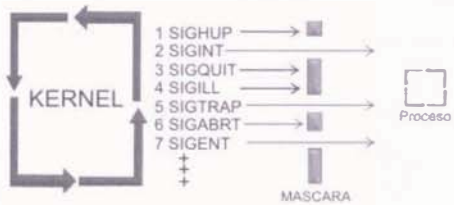
```
struct sigaction {
   void(*sa_handler)(); /* Manejador para la señal */
   sigset_s sa_mask; /* Señales bloqueadas durante la ejecución del
manejador */
   int sa_flags; /* Opciones especiales, véase man sigaction */
};
```

el primer miembro de la estructura indica la acción a ejecutar cuando se reciba la señal. Su valor puede ser una de las siguientes opciones:

- Una función que devuelve un valor tipo `void` y que puede aceptar un entero. Este número entero lo manda el sistema operativo hacia el proceso y es el identificador del número de señal que se envía.
- `SIG_DFL`: indica que se lleve la acción por defecto.
- `SIG_IGN`: especifica que la señal sea ignorada al recibirse.

Línea 2 Existe un conjunto de señales que puede ignorarse en el caso de que las envíe el kernel cuando se está ejecutando el proceso. Esto se logra mediante una máscara que impide el paso del conjunto de señales, mientras que a las señales que no se encuentran en el conjunto se les permite el paso, como se muestra en la figura 11.2.

Figura 11.2
La máscara impide el paso del conjunto de señales



Los conjuntos de señales se definen como tipo `sigset_t`. La función:

```
int sigemptyset(sigset_t *set)
```

inicia un conjunto de señales de modo que no contenga ninguna señal. La función:

```
int sigfillset(sigset_t *set);
```

inicia un conjunto de señales con todas las que están disponibles en el sistema.

Línea 3 La función:

```
int sigaddset(sigset_t *set, int signo);
```

añade una señal con número `signo` al conjunto de señales `set` previamente iniciado. El proceso de eliminar una señal se da con:

```
int sigdelset(sigset_t *set, int signo);
```

para determinar si una señal pertenece a un conjunto de señales se utiliza:

```
int sigismember(sigset_t *set, int signo);
```

la cual devuelve 1 si la señal `signo` pertenece al conjunto y en caso contrario, regresa 0.

Línea 4 La función:

```
int sigprocmask(int how, sigset_t *set, sigset_t *oset);
```

permite activar o desactivar la máscara que impide el paso del conjunto de señales hacia el proceso. El primer parámetro `SIG_SETMASK` activa la máscara, el segundo parámetro es un apuntador a la máscara y el tercer parámetro en `NULL` indica que no se requiere obtener el valor anterior de la máscara.

Línea 5 La función:

```
int sigaction(int sig, struct sigaction *act, struct sigaction
*oact);
```

permite armar una señal, el primer parámetro es el número de la señal para la que se va a elaborar el manejador, el segundo apunta a la estructura que contiene el nuevo manejador y el tercer parámetro apunta a la estructura que almacena información sobre el manejador establecido actualmente.

Línea 6 La función:

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

envía, al proceso que la ejecuta, la señal `SIGALRM` después de transcurrido el número de segundos especificados en `seconds`.

Línea 7 `int pause(void)`

Esta función bloquea el proceso que la invoca hasta que llegue una señal, cualquier señal no ignorada sacará al proceso del estado de bloqueo.

Cabe aclarar que existen otras funciones para el manejo de señales, como las ofrecidas por las versiones System V y 4.3 BSD, sin embargo hemos hecho uso de las que se apegan al estándar POSIX.

Ejercicio 11.1 Modifique el programa para que se envíe la señal `SIGALARM` cada medio segundo en vez de cada tres segundos (véase `ualarm`). Además modifique la función manejadora de la señal para que imprima el número de señal que le envía el *kernel* a nuestro proceso (relea lo que efectúa la línea 1 del código).

Pregunta 11.2 Envíe la señal `SIGALARM` desde otra terminal con el comando kill y explique lo que pasa. ¿Qué sucede si añade `SIGALARM` a la máscara?

Pregunta 11.3 ¿Cuál es la línea de código en el programa 11-1 que se ejecuta inmediatamente después de ejecutarse el manejador de señal? (recomendación: imprima mensajes).

Ejercicio 11.2 Ejecute el programa y presione (`CTRL-\`). Esta combinación envía la señal `SIGQUIT(3)` al proceso, y como podrá observar es una señal ignorada. Modifique el programa para que esta señal provoque su terminación.

Si al programa 11-1 en ejecución le enviamos la señal `SIGTSTP(18, 20 o 24)` desde otra terminal con el comando *kill* podremos observar que el proceso se detiene y aparece un mensaje similar a éste:

```
[1]+  Stopped
```

Para hacerlo andar de nuevo desde donde se detuvo sólo necesitamos teclear el signo `%` seguido del número que aparece entre corchetes, en este caso: `%1`

Este mecanismo es muy útil si se quiere detener por un momento la ejecución de un programa. En LINUX se puede también enviar la señal `SIGSTOP` al oprimir la combinación de teclas `CTRL-Z`.

Ejercicio 11.3 Modifique el programa 11-1 para evitar que se detenga con la señal 18 `SIGTSTP`, y que además imprima el mensaje `"Me han intentado detener"`.

Ejercicio 11.4 Modifique el programa para que ninguna señal lo pueda detener (sugerencia: véase `sigfillset()`).

Ejercicio 11.5 En el programa 11-2 el padre crea un hijo que va a ejecutar un mandato recibido en la línea de comandos y espera a que termine. Si el proceso hijo no termina antes de que pasen 5 segundos, el proceso padre debe matar al proceso hijo con la función `kill()`.Incluya las líneas de código faltantes.

Programa 11-2

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

pid_t pid;

int main(int argc, char **argv)
{
   int pid, status;

   /* Se crea el proceso hijo */
   pid = fork();
   switch(pid)
   {
   case -1:
      exit(-1);
   case 0:
   /* El proceso hijo ejecuta el comando solicitado */

   default:
   /* Establece el manejador */

   /* Espera al proceso hijo */
   wait(&status); /* Véase con man para más detalles sobre wait
*/
   }
   exit(0);
}
```