SEÑALES: GENERANDO Y RECIBIENDO AVISOS ENTRE PROCESOS

¡Bienvenidos a la central de avisos de vuestro sistema operativo favorito! En estas sesiones aprenderemos a utilizar este mecanismo de Unix/Linux, por el cual el sistema puede enviar notificaciones (avisos, o señales si utilizamos la terminología inglesa) a cualquier proceso. De hecho, es bastante habitual que lo haga. El primer ejemplo que nos viene a la cabeza es la situación en la que queremos interrumpir la ejecución de un proceso que hemos lanzado por consola. Todo el mundo piensa, de forma instintiva, en pulsar simultáneamente las teclas CONTROL y C (a partir de ahora, nos referimos a esta combinación como Ctrl-C). Esto hace que el intérprete de comandos mande un aviso al proceso, provocando su cierre.

Otro ejemplo clásico es observar los mensajes del sistema operativo que aparecen en pantalla cuando se va a apagar la máquina. Lo normal es ver un mensaje del tipo (lo digo traducido): "Enviando el aviso de terminar a todos los procesos del sistema". Efectivamente, en Unix es importante dar información a todos los programas en ejecución de que la máquina está a punto de cerrar. Como veremos más adelante, los procesos pueden ser conscientes de ese importante aviso, y les puede dar tiempo a cerrar ficheros abiertos, o notificar por un canal de comunicaciones al otro extremo que están a punto de terminar y que se va a cerrar dicho canal (lo que puede ser vital para que los servicios en red funcionen correctamente). También, en algunos casos, puede darse el caso de procesos "rebeldes". Son aquellos que, tras recibir la notificación del sistema, se hacen los locos y se niegan a terminar. En ese caso, el sistema los asesina sin piedad, y además se jacta de ello a través de una nueva notificación: "Enviando el aviso de matar a los procesos restantes".

De todo esto podéis deducir un par de detalles importantes: los procesos pueden estar, de alguna forma, pendientes de estos avisos. Y también pueden reaccionar de diversas formas, hasta ignorándolos, si no quieren hacer lo que les ordena el sistema. Eso sí, debe haber algunos de estos avisos que no se puedan ignorar (¡ese de "te voy a asesinar" suena a definitivo!). La forma en que escuchamos y reaccionamos ante avisos en nuestros programas lo veremos más adelante.

LOS PROCESOS DENTRO DEL SISTEMA OPERATIVO

El sistema gestiona y atiende a todos los procesos que se ejecutan en él, de una forma jerárquica. Cuando ejecutamos un programa en la consola, ésta ejerce como proceso "padre" de este nuevo proceso que ha nacido. A su vez, la consola también es hija del proceso que ejecuta todo el entorno gráfico de ventanas. Si seguimos subiendo en la jerarquía, llegaremos al proceso "adán", el primero de todos, que se denomina init. El init es el primer proceso del sistema, que ejecuta todos los demás al arrancar la máquina.

Un proceso Unix/Linux está identificado de forma única dentro del sistema mediante un número positivo mayor que cero. El init tiene asignado el valor 1, y todos los demás tendrán un valor mayor. A este identificador numérico asociado a un proceso se le conoce como PID (process identifier). El comando de consola:

ps aux

muestra una lista de los procesos que hay ejecutándose en ese mismo momento en el sistema (el argumento aux permite ver todos los procesos). Como Unix/Linux fue concebido para ser multitarea, veréis al ejecutar el comando un listado enorme, donde cada línea da información en columnas sobre un proceso, y LA SEGUNDA FILA es el PID. Podéis utilizar algunos comandos adicionales para recorrer de forma algo más controlada ese larguísimo listado:

ps aux | more

la barra de OR en Unix hace que la salida de un comando (en este caso ps aux) se redireccione como entrada a otro comando. more se utiliza para mostrar solo las líneas que caben en la pantalla del terminal, sin scroll. Si hiciera falta, pulsando la barra espaciadora, se muestra el contenido de la siguiente página del listado.

También podéis hacer que se muestren solo las líneas que contienen un texto concreto, así:

ps aux | grep nombre_ejecutable

usando el comando grep, que filtra las líneas de entrada y solo muestra aquellas que contienen el texto que le pasamos como argumento, podemos obtener la información concreta de un proceso que se llama nombre_ejecutable. Os recomiendo que utilicéis esta forma para obtener el PID de los procesos que estéis ejecutando (si el programa en ejecución o permite que introduzcáis nuevos comandos al terminal, obviamente tendréis que abrir otro terminal para poder utilizar dichos comandos).

Procesos en primer plano y segundo plano

Si, en consola, ejecutamos un programa y este no termina, no es posible introducir nuevos comandos, como hemos comentado hace un momento. A estos procesos que se ejecutan de esta forma se les llama procesos en primer plano (foreground), y solo admiten un Ctrl-C para finalizar su ejecución. Es por esto que, cuando queramos mirar su PID (¡sin interrumpirlos!), tengamos que recurrir a abrir un terminal adicional. Sin embargo, en Unix/Linux existe una forma de ejecutar un proceso y que, independientemente de que termine inmediatamente o no su ejecución, se nos devuelva el control del intérprete de comandos para introducir más órdenes. De esta forma, cuando ejecutamos:

./nombre_ejecutable &

ese ampersand al final, le indica a la shell que el proceso debe ser ejecutado en segundo plano (background). Automáticamente, recibimos un aviso por pantalla con un número que corresponde al PID del proceso en background, y ya podemos seguir introduciendo comandos en el mismo terminal. Tenéis que notar que, en esta nueva situación, un Ctrl-C ya no termina con la ejecución de dicho proceso, puesto que se ha desligado de la consola y ahora la shell ya no sabe cómo dirigirle el aviso (el mecanismo detrás de este curioso fenómeno es también de vital importancia para el software de comunicaciones, y le sacaremos partido en la siguiente sesión).

AVISOS (SEÑALES) HABITUALES

El sistema de avisos del sistema operativo fue diseñado con la finalidad principal de informar a los procesos de situaciones que podían ocurrir en algún momento de su ejecución, y que se consideran críticas para seguir o no con su funcionamiento normal. De hecho, la mayoría de

señales provenientes del sistema notifican errores, y solo una pequeña parte son meramente informativas.

El diseño del sistema operativo determinó que todo aviso recibido en forma de señal tendría también un comportamiento POR DEFECTO del proceso. En la mayoría de los casos, el comportamiento suele ser que el proceso termine su ejecución inmediatamente si la recibe.

En la sección siete del manual, donde se documentan algunas de los comportamientos variopintos del sistema y sus bibliotecas, se puede consultar

man 7 signal

y allí se nos muestra la cantidad de señales definidas en el sistema, cada una con su significado, y el comportamiento por defecto ante su recepción. Empezamos a identificar señales que ya hemos introducido: SIGINT es la señal que se recibe indicando "interrumpir el proceso" y se envía pulsando el Ctrl-C. SIGTERM es la señal que se manda a los procesos para que terminen, en ese cierre ordenado antes de apagar la máquina. SIGKILL es la señal asesina. Aunque vemos que todas las señales tienen su significado, hay dos que se definieron para que los usuarios le den significado propio: SIGUSR1 y SIGUSR2. Eso significa que, si una de estas señales llega a nuestro programa en ejecución, se ha tenido que definir previamente qué sentido darle y que hacer a partir de ahí (en el enunciado del problema, los requisitos de la aplicación que se han pactado con un cliente de la empresa, etc.).

Y eso nos lleva directamente a utilizar de forma proactiva el servicio de avisos. No solo no vamos a quedarnos sentarnos esperando a que se produzca una situación que requiera que el sistema envíe una señal a un proceso, sino que somos nosotros los que vamos a forzar ese envío utilizando el comando de consola kill (man 1 kill):

kill -SIGTERM pid_proceso

donde podéis cambiar SIGTERM por otra señal, y pid_proceso lo tenéis que reemplazar por el PID concreto del proceso al que el sistema operativo tiene que enviarle vuestra señal. ¡Practicad con distintos procesos, consolas, incluso, intenta desde una consola terminar con el Visual Studio Code! Puedes buscarlo (si está ejecutándose, claro) como: /usr/share/code/code

NOTA: El hecho de que el comando de consola se llame kill quizá es algo desafortunado, ya que en realidad en un lanzador de avisos genérico. Tú como usuario configuras qué señal es la que manda, y no se presupone que va a mandar SIGKILL por defecto (de hecho, por defecto manda SIGTERM).

HACIENDO USO DE SEÑALES EN NUESTRO PROGRAMAS

La llamada al sistema kill (man 2 kill) permite, en C, decirle al S.O. que mande una señal a otro proceso, si previamente conocemos su PID.

Hagamos un experimento, con este sencillo programa (faltan los includes por legibilidad (*)):

```
int main(int argc, char *argv[]){
	if (argc < 2){
		printf("Introduzca un PID para enviarle una senal\n");
		return 1;
	}
	pid_t pid = atoi(argv[1]);
```

```
        int retorno = kill(pid, SIGTERM);
        if(retorno < 0){
                perror("Algo malo paso con kill");
                return 1;
        }
        return 0;
}
```

Hay algún detalle interesante aquí. El tipo de dato pid_t es el que utilizamos para almecenar un número de pid. En este caso lo obtenemos al pasar a entero la cadena con el primer argumento introducido tras el nombre del programa, al ejecutarlo por consola.

¿Por qué introduzco el PID requerido como argumento al programa? Porque el PID de que adquiere un programa al ejecutarse solo dura hasta que termina. La próxima vez que se ejecute obtendrá un nuevo valor. Es inútil codificar como constante ese valor del pid que quieres usar en tu código, al compilarlo.

La llamada kill() tiene dos argumentos, el pid y la señal a enviar. Si miráis su prototipo, el segundo argumento (la señal a enviar) es un entero, lo que significa que SIGTERM, SIGINT y las demás, son constantes enteras (en realidad, macros de preprocesado definidas con #define al importar la biblioteca <signal.h>). En caso de fallo, devuelve -1.

(*) NOTA: En gcc, a pesar de incluir la biblioteca correspondiente para kill(), nos va a dar un warning cuando compilamos con -std=c99 o -std=c11, quejándose de kill con un "warning: implicit declaration of function '*kill*'". No pasa nada en este caso. Si os molesta, se puede quitar si ponemos en la primera línea de nuestro programa esta definición:

#define _DEFAULT_SOURCE

o, de forma equivalente, si incluimos -D_DEFAULT_SOURCE como opción extra al compilar. Por supuesto que no hace falta aprender esta macro, ni muchísimo menos. **Podéis ignorar este warning de forma segura para vuestros programas**.

Vamos a probar el programa anterior. Tenemos dos formas de hacerlo:

1) Abre un terminal y ejecuta un programa en primer plano, de forma que deje bloqueada la shell. Por ejemplo, teclea esta orden y dale al Intro para que se ejecute un editor de texto.:

mousepad

Nota que no vuelves a tener control sobre el terminal hasta que no cierres la ventana de mousepad. Ahora abre otro terminal, y busca el pid del proceso mousepad (ps aux | grep mousepad). Cuando lo tengas, compila el programa propuesto indicando, por ejemplo, que el ejecutable se va a llamar "prueba" (-o prueba), y ejecútalo así:

./prueba PID

donde, en este caso, PID representa el número que representa al identificador de proceso obtenido tras la llamada a ps aux. Si todo va bien, la ventana del editor mousepad se cerrará. Prueba con otras señales en el código ¿qué pasa si mandas la señal SIGCHLD? En man 7 signal, tienes la solución.

2) Abre un terminal y ejecuta un programa en segundo plano. Quédate con el PID que se muestra por pantalla. Cuando lo tengas, compila el programa propuesto indicando, por ejemplo, que el ejecutable se va a llamar "prueba" (-o prueba), y ejecútalo así:

./prueba PID

¿Puedo hacer que el sistema operativo me mande un aviso a mi mismo? Pues sí, en algunos casos podría ser útil hacerlo. ¿Y cómo sé mi propio PID una vez me estoy ejecutando como proceso? Para eso está la función getpid(), que te devuelve ese valor:

```
pid_t mipid = getpid();
kill(mipid, SIGINT);
```

Por comodidad, también está disponible una función de biblioteca denominada raise, equivalente a usar kill para decirle al sistema que me mande un aviso a mí mismo:

raise(SIGUSR1); //equivale a kill(getpid(),SIGUSR1);

Y ya que estamos, y para recordar que todos los procesos se organizan en torno a una jerarquía de padres a hijos, existe otra función denominada getppid(), que permite saber el pid de nuestro proceso creador. En esta asignatura tendremos que comunicarnos con nuestro proceso padre por diversas circunstancias, y esta es una de ellas (haciendo que el sistema le mande un aviso).

IGNORAR SEÑALES

Ya sabemos, y habéis practicado, que el comportamiento por defecto del proceso al recibir una señal es terminar su ejecución (otras lo paran -stop-, y otras lo termian también generando un archivo .core con información a veces útil para depuración). Sin embargo, existe la opción de ignorarlas. La llamada al sistema signal, informa al sistema operativo de qué tiene que hacer cuando el proceso reciba una señal. Usándola de esta manera:

signal(SIGINT, SIG_IGN); //no hace falta chequear su valor de retorno

estamos diciendo al sistema que ignore las señales SIGINT. Si eso pasa, el programa pasa a ser un rebelde, y no hace caso a terminar cuando pulsamos Ctrl-C (si está ejecutándose en primer plano, ni a kill -SIGINT su_pid si está ejecutándose en segundo plano).

Vamos a probarlo con un programa (de nuevo se omiten los includes):

```
int main(){
        signal(SIGINT, SIG_IGN); //no hace falta chequear su valor de retorno

        while(1){
        }
}
```

Compila y ejecuta en un terminal. Luego intenta un Ctrl-C y verás que no funciona (¡tendrás que matarlo usando kill con otra señal, y desde otro terminal!).

Habrás reparado en ese while(1){ } que expresa un bucle infinito, puesto que su condición siempre es verdad. Lo he puesto después de la llamada a signal() por una razón muy simple ¡si

termina de ejecutarse el programa antes de que te de tiempo a mandarle una señal, no estamos consiguiendo nuestro objetivo!. Otras alternativas para expresar un bucle infinito en C/C++:

while(1); //ni siquiera necesito usar las llaves si no voy a hacer nada en cada iteración del bucle

for(;;); //otra forma que se suele ver en códigos antiguos

Si alguien se lo está preguntando, estos bucles infinitos, que no tienen instrucciones dentro de sus llaves, SÍ que están utilizando la CPU, y de manera poco solidaria para el resto de procesos que aguardan su turno para ejecutarse. Por supuesto que es insolidario consumir ciclos de CPU del ordenador para no estar haciendo nada útil. En nuestro caso nos viene bien para darnos tiempo a enviarle señales al proceso, así que vamos a buscar una alternativa, más justa para el resto de procesos, que no consuma ciclos de CPU y también, por tanto, que sea más eficiente de cara al consumo de energía (¡salvemos el planeta, cualquier gesto cuenta!). De esta forma, os presento a pause():

```
int main(){
	signal(SIGINT, SIG_IGN); //no hace falta chequear su valor de retorno

	while(1){
		pause(); //no hace falta chequear su valor de retorno
	}
}
```

pause() es la llamada que hace que el sistema operativo haga al programa esperar (sin ocupar CPU) hasta que no recibe una señal. No hace falta ponerla dentro de un bucle infinito, aquí la hemos puesto para que el programa se reanude al recibir una señal y, si la hemos ignorado, se ponga a esperar a una nueva. Y así de forma indefinida.

Si en algún momento queremos que el sistema operativo restaure el comportamiento por defecto que tendrá el proceso al recibir una señal, tenemos que acudir de nuevo a signal(), esta vez indicando como segundo argumento el valor SIG_DFL. ¿Qué haría este cambio en el código si le mandas al proceso dos Ctrl-C?:

```
while(1){
	pause(); //no hace falta chequear su valor de retorno
	signal(SIGINT, SIG_DFL);
}
```

No obstante, el sistema operativo siempre está al mando. Hay dos señales que no se pueden ignorar, son las señales SIGKILL y SIGSTOP

ATENDER LOS AVISOS

A pesar de que, por ahora, hemos visto signal() como una llamada al sistema que permite decirle al sistema operativo que ignore determinadas señales o restaure el proceso a su comportamiento original al recibirlas, signal() es capaz de mucho más.

De hecho, lo normal es utilizar signal para indicarle al sistema operativo que, ante la llegada de una señal, suspenda la ejecución normal del proceso (vaya por donde vaya en ese momento) y ejecute una función especial. Cuando termine de ejecutar esta función especial, inmediatamente se retomará la ejecución normal por donde iba. Aquí entra en juego el concepto de puntero a función. Efectivamente, signal() recibe como segundo argumento la dirección de memoria de aquella función que ha de ejecutar el sistema operativo ¡nunca nosotros directamente en nuestro código! en caso de que se recibiese la señal indicada en su primer argumento. Vamos con un ejemplo:

```
void ejecutame_si_llega_SIGINT(int num_sig){
	write(1,"no me da la gana terminar!",26);
}

int main(){
	signal(SIGINT, ejecutame_si_llega_SIGINT); /*ojo, no estas ejecutando la funcion! Estas utilizando su nombre como puntero */

	while(1){
		pause(); //no hace falta chequear su valor de retorno
	}
}
```

NOTA IMPORTANTE: para compilar este programa no hagas uso de la opción -std=c99 o c11. Luego verás porqué.

Algunas personas de dudosa solvencia moral se confunden y creen que signal() se utilizar para enviar una señal (no es vuestro caso, por supuesto). Nada más lejos de la realidad: signal() solo sirve para que le quede constancia al sistema operativo de lo que tiene que hacer con el proceso en caso de que llegue un aviso. Y, claro está, podría incluso no llegar antes de que acabe el programa.

Como buen puntero a función, el segundo argumento a signal() requiere proporcionar la dirección de memoria a un tipo de función concreta, que devuelve void y acepta como argumento un entero. A este tipo de funciones se las llama manejadoras (una horrible traducción de handler, en inglés). El argumento sirve para que el sistema operativo introduzca por valor el número de señal que ha provocado la ejecución de la función. Gracias a esto, podemos registrar la misma función manejadora para que se atienda a varias señales distintas, si es que llegaran a recibirse. Como ejemplo, establezcamos que una programa se haga inmune a SIGINT pero no a SIGTERM, mientras imprime por pantalla un mensaje por pantalla informando de cada recepción:

```
void ejecutame_si_llega_SIGINT_SIGTERM(int num_sig){
	if(num_sig == SIGINT){
		write(1,"no me da la gana terminar!",26);
	}else{
		write(1,"vale, termino",13);
		exit(0);
	}
}

int main(){
```

```
	signal(SIGINT, ejecutame_si_llega_SIGINT_SIGTERM); /*ojo, no estas ejecutando la funcion! Estas utilizando su nombre como puntero */
	signal(SIGTERM, ejecutame_si_llega_SIGINT_SIGTERM); /*ojo, no estas ejecutando la funcion! Estas utilizando su nombre como puntero */

	while(1){
		pause(); //no hace falta chequear su valor de retorno
	}
}
```

NOTA IMPORTANTE: vuelve a compilar este programa sin las opciones -std=c99 o -std=c11. Estás más cerca de saber el porqué (¡emoción!).

Aunque hemos visto que es posible registrar la misma función manejadora para atender la recepción de varias señales, es una buena guía de estilo utilizar una función distinta para manejar cada señal recibida.

Por último, es necesario indicar que ni SIGKILL ni SIGSTOP pueden registrar ninguna función manejadora. Con el sistema operativo no se juega.

¿EL REGISTRO DEL MANEJADOR SE HACE UNA SOLA VEZ EN EL PROGRAMA?

Históricamente, ha habido dos formas de reaccionar ante la recepción de una señal. El modo BSD, que es el que utilizó la implementación de Unix de referencia de la Universidad de Berkeley, asume que la función manejadora de la señal sigue registrada hasta que el programador la cambie explícitamente con una nueva llamada a signal() y registre otra. Sin embargo, en el modo System V (otra versión de Unix), se asumía que el registro del manejador era para un solo uso. Es decir, si volvía a aparecer de nuevo la señal en cuestión, el comportamiento se había reseteado a SIG_DFL (y ese reseteo se hace justo cuando se entra a ejecutar la función manejadora).

¿Cómo practicar el comportamiento System V? Utilizando las opciones -std=c99, o -std=c11, o -ansi a la hora de compilar vuestro código C

¿Cómo practicar el comportamiento BSD? Sin utilizar ninguno de las anteriores opciones de compilación en C. En C++ por defecto, también.

Así que, si queremos generar un código portable, e independiente a las opciones de compilación, se requiere volver a registrar la funcion manejadora dentro del mismo:

```
void manejadora(int num_senal){
	//...
	signal(num_senal, manejadora); /*en caso BSD no tiene efecto, en caso de System V, lo restaura antes de salir*/
}
```

Es absolutamente recomendable que siempre volváis a hacer este registro, por seguridad, en vuestro código. Así, siempre se comportará del mismo modo.

UNA ÚLTIMA CUESTIÓN: ¿POR QUÉ UTILIZAS write Y NO printf dentro de las funciones manejadoras?

Creía que no os habíais dado cuenta. Hay reglas de lo que se puede y no se puede hacer en un manejador. Si utilizáis un printf en estos ejemplos no pasa nada, porque son códigos de juguete, pero jamás podréis utilizarlo en código de verdad en la empresa. Las razones quedan fuera del alcance de esta asignatura. Solo os diré que, si una función no es REENTRANTE, no se debe utilizar dentro de una función manejadora (todas las funciones, en su página del manual indican si son reentrantes o no). El concepto de función reentrante cobrará sentido en asignaturas de programación concurrente, en cursos superiores de la carrera.

En realidad, en las funciones manejadoras no se debería realizar casi ninguna operación, deben ser cortas y rápidas de ejecutar puesto que, recuerda, hemos interrumpido la ejecución del proceso hasta que salgamos de ellas. Sí que es muy conveniente utilizar una variable enterna que sirva como condición a chequear en el programa para en caso de aparición de una señal. Por ejemplo, supongamos que un programa tiene que esperar a hacer algo hasta que no se reciba la señal SIGUSR1. Lo podemos hacer así:

```
int ha_llegado_sigusr1 = 0;

void ejecutame_si_llega_SIGIUSR1(int num_sig){
	ha_llegado_sigusr1 = 1;
	signal(num_sig, ejecutame_si_llega_SIGUSR1);
}

int main(){
	signal(SIGUSR1, ejecutame_si_llega_SIGUSR1); /*ojo, no estas ejecutando la funcion! Estas utilizando su nombre como puntero */

	while(!ha_llegado_sigusr1){
		pause(); //no hace falta chequear su valor de retorno
	}
	//ha llegado SIGUSR1, puede continuar el programa
}
```

USO DEL TIEMPO EN SOFTWARE DE COMUNICACIONES. LOS TEMPORIZADORES

La gestión del tiempo es importantísima en nuestras aplicaciones. Por ejemplo, puede que cada determinado tiempo tengamos que mandar un mensaje por un canal para demostrar que seguimos vivos, o quizá tengamos que poner un temporizador (una cuenta atrás) para retransmitir un paquete de datos si no hemos recibido la confirmación del otro extremo de que ha llegado sin errores. El sistema operativo tiene más de un mecanismo para permitirnos tener control sobre el tiempo transcurrido en nuestros programas. Aquí se va a estudiar el que tiene relación directa con los avisos.

De manera directa, podemos poner decirle al sistema operativo que active una cuenta atrás y nos envíe la señal de alarma (SIGALRM) cuando dicha cuenta llegue a cero. Para ello utilizamos la llamada al sistema alarm(), que admite un parámetro entero, con los segundos que han de transcurrir hasta que el sistema genere la señal SIGALRM. Dado este código sencillo:

```
int main(){
	alarm(5);
```

```
        while(1){
                pause(); //no hace falta chequear su valor de retorno
        }
}
```

El programa espera 5 segundos y, después, termina (el comportamiento asociado a la recepción de la señal SIGALRM es terminar el proceso). Si quiero hacer un temporizador periódico, podría pensar en reactivar la alarma con este código:

```
int main(){
        alarm(5);

        while(1){
                pause(); //no hace falta chequear su valor de retorno
                alarm(5);
        }
}
```

Lamentablemente, no va a funcionar. Pruébalo. Recuerda que tras la recepción de la primera señal SIGALRM, el proceso termina sin dar lugar a que se ejecute la nueva llamada a alarm();

Lo que vamos a hacer es registrar una función manejadora para SIGALRM. ¡No se te ocurra hacerlo después de llamar a alarm()! Siempre antes.

```
void manejadora(int num_senal){
        signal(num_senal, manejadora);
}

int main(){
        signal(SIGALRM, manejadora);
        alarm(5);

        while(1){
                pause(); //no hace falta chequear su valor de retorno
                alarm(5);
        }
}
```

El uso de alarmas tiene dos inconvenientes fundamentales:

1) No se puede precisar valores más pequeños al segundo. Esto NO es bueno para el software de comunicaciones. En muchos casos, los temporizadores de envío de mensajes necesitan reaccionar cuando pasen milisegundos, no segundos.

2) En caso de que queramos hacer un temporizador periódico, hay un tiempo que transcurre desde que se sale del pause y se vuelve a reactivar la alarma. Ese pequeño tiempo (su nombre técnico es deriva temporal) hace que haya un desajuste en el periodo. Imagina (exagerando muchísimo) que la deriva en la activación cada alarma es de 0.5 segundos. Eso quiere decir que la primera vez llegará la señal SIGALRM a los cinco segundos, pero la siguiente vez llegará a los

10.5 segundos (contando desde el principo del temporizador, la primera alarma). la siguiente SIGALRM saltará a los 16 segundos (en vez de a los 15, si no hubiera derivas), y así seguiríamos hasta que en la décima iteración saltase a los 55 segundos. Un momento, ¿parece que se ha vuelto a sincronizar? No, ha acumulado tal deriva que, se ha saltado un ciclo entero. Esto es muy peligroso para nosotros.

La solución a estos dos cuestiones la tenemos utilizando la llamada al sistema setitimer() (poner un temporizador de intervalos/periodos, de ahí la "i").

Si miráis la página del manual para setitimer() (man 2 setitimer), veréis que tiene tres argumentos, pero nosotros solo vamos a usar los dos primeros.

Primer argumento: dice qué tipo de cuenta atrás tiene que hacer el sistema operativo. Lo habitual es que cuente todo el tiempo transcurrido (tiempo real), así que ponemos ITIMER_REAL. Otras veces, solo queremos que se cuente el tiempo en el que el proceso está ocupando cpu, y entonces se pone ITIMER_VIRTUAL. Además de que no es lo mismo uno que otro, las señales que manda el S.O (y que nosotros tenemos que registrar con su correspondiente manejador), son distintas. En el primer caso se lanza la ya conocida SIGALRM, pero en el segundo se lanza SIGVTALRM.

Segundo argumento: se pide la dirección a una estructura de tipo struct itimerval, donde nosotros habremos rellenado previamente los valores de la cuenta atrás (incluyendo tiempo en segundos y microsegundos en el que nos llegará la primera señal de alarma, y luego el periodo también en segundos y microsegundos, para que siga llegando).

Si ponemos los campos del periodo (interval) a cero, el temporizador NO lanzará señales periodicamente después de la primera.

Entonces, antes de llamar a setitimer(), hay que crear una variable en la pila, de tipo struct itimerval, con los valores adecuados. Pero ¿de qué tipo son y como se llaman los campos de la estructura struct itimerval? Por supuesto que no hay que aprendérselos de memoria, puesto que están explicados en la página del manual de setitimer.

Tercer argumento: no lo utilizamos, y lo ponemos a cero.

UNA ACLARACIÓN NECESARIA LLEGADOS A ESTE PUNTO

Si os dais cuenta, tanto alarm() como setitimer() utilizan unos argumentos de cuenta atrás que siempre son relativos al momento en el que se ejecutan. Es decir, una alarma de 5 segundos indica que hay que empezar a contarlos desde que se efectúa la llamada al sistema. A este tipo de temporizadores se les conoce como temporizadores de **TIEMPO RELATIVO**. Pero, ojo, en otras bibliotecas de funciones se especifica en sus manuales que sus argumentos se tienen que pasar en **TIEMPO ABSOLUTO**. ¿Qué significa esto? Que hay que dar un valor de tiempo especificado en año-mes-día-hora-minuto-segundo-microsegundo. ¿Qué diferencia hay entre usar temporizadores de un tipo u otro? El fundamental es que, los temporizadores de tiempo absoluto se pueden parar y luego reanudar sin hacer ningún cambio en su argumento de tiempo: cuando tengan que llegar a cero, lo harán cuando se esperaba. Esto no es así con los temporizadores de tiempo relativo: si paras un temporizador, al reiniciarlo tienes que restar a su argumento el tiempo que estuvo inactivo, porque si no, estás cambiando su comportamiento original.

Saber describir qué son las derivas de tiempo y como se corrigen, así como la diferencia entre los temporizadores de tiempo relativo y absoluto entra siempre en evaluación. En el laboratorio, no obstante, manejaremos solo por simplicidad los temporizadores de tiempo relativo alarm() y setitimer().

Y AHORA OS PROPONGO MINI RETOS:

Practicaremos alarm() y setitimer(), y espero que entendamos su uso y estemos cómodos con la forma en que se trabaja con ellas. De hecho, setitimer() casi siempre se necesita en ejercicios de examen. Vamos a ver si somos capaces de hacernos más preguntas de ingeniero (curiosidad innata, es la profesión que habéis elegido). Asumo que leéis bibliografía y tenéis acceso a buscar por Internet:

1) ¿Qué pasa si ejecuto este código?: alarm(5); alarm(3);

¿Saltan dos señales, una a los cinco segundos y otra a los tres?

2) ¿Qué pasa si ejecuto este código?: alarm(5); setitimer(.../*pongo otro valor de temporizador aqui dentro*/...);

¿Saltan dos señales, una a los cinco segundos y otra al tiempo que haya en el setitimer()?

3) ¿Cómo se cancela un temporizador? Imagina que en tu protocolo de comunicaciones, activas una cuenta de 30 milisegundos, para que, si salta la alarma y no te ha llegado un paquete, entonces retransmites tú el último que enviaste, pero si te llega un paquete antes de que pasen los 30 milisegundos, debes cancelar el temporizador.

La respuesta a estas tres preguntas está en la misma página del manual de alarm (man 2 alarm). No hay que irse muy lejos.

4) Si quisiéramos pausar un temporizador y luego continuarlo por donde se quedó, conociendo solo alarm() y setitimer() ¿podríamos hacerlo? ¿tiene sentido? Esta es de nivel. Si queréis, sacadme el tema y lo comentamos en clase.