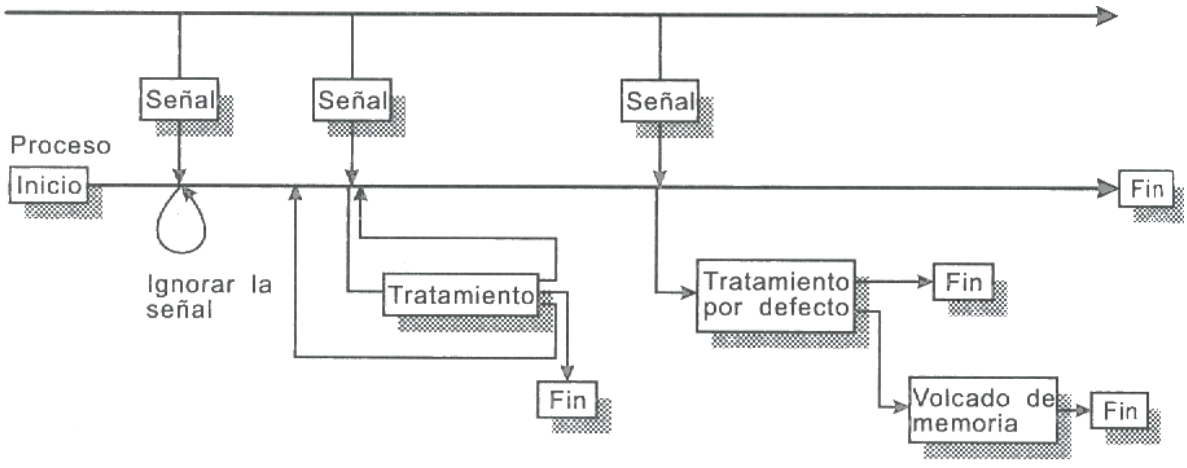
7. Señales y Funciones de Tiempo

7.1 Concepto de Señal

Las señales son interrupciones software que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial. El término señal se emplea también para referirse a un evento.

Los procesos pueden enviarse señales unos a otros a través de la llamada `kill` y es bastante frecuente que, durante la ejecución, un proceso reciba señales procedentes del kernel. Cuando un proceso recibe una señal, pueden tratarla de tres formas diferentes:

1. Ignorar la señal, con lo cual es inmune a la misma.
2. Invocar a la rutina de tratamiento por defecto. Esta rutina no la codifica el programa, sino que la aporta el kernel. Según el tipo de señal, la rutina de tratamiento por defecto va a realizar una acción u otra. Por lo general suele provocar la terminación del proceso mediante una llamada a `exit`. Algunas señales no sólo provocan la terminación del proceso, sino que además hacen que el kernel genere, en el directorio de trabajo actual del proceso, un archivo llamado `core` que contiene un volcado de memoria del contexto del proceso. Este archivo podrá ser examinado con ayuda de un programa depurador (`adbm`, `sdb`) para determinar qué señal provocó la terminación del proceso y en qué punto exacto de su ejecución se produjo. Este mecanismo es útil a la hora de deputar programas que contienen errores de manejo de los números de punto flotante, instrucciones ilegales, acceso a direcciones fuera de rango, etc.
3. Invocar a una rutina propia que se encarga de tratar la señal. Esta rutina es invocada por el kernel en el supuesto de que esté montada y será responsabilidad del programa codificarla para que tome las acciones pertinentes como tratamiento de la señal. En estos casos, el programa no va a terminar a menos que la rutina de tratamiento indique lo contrario.



En la figura anterior vemos esquematizada la evolución temporal de un proceso y cómo, a lo largo de su ejecución, recibe las señales procedentes del kernel. La primera señal que recibe no provoca

que el proceso cambie el curso de su ejecución, esto es debido a que la acción que está activada es que el proceso ignore la señal. El proceso prosigue su ejecución y recibe una segunda señal que lo fuerza a entrar en una rutina de tratamiento. Esta rutina, después de tratar la señal, puede optar por tres acciones: restaurar la ejecución del proceso al punto donde se produjo la interrupción, finalizar el proceso o restaurar alguno de los estados pasados del proceso y continuar la ejecución desde ese punto (más adelante veremos que esto se consigue con las funciones estándar `sigsetjmp` y `siglongjmp`). El proceso puede también recibir una señal que lo fuerce a entrar en la rutina de tratamiento por defecto.

En las siguientes secciones volveremos a hablar sobre estos conceptos y veremos cómo se especifica cuál de las tres formas es la elegida para tratar una señal.

7.2 Tipos de Señales

Cada señal tiene asociado un número entero positivo, que es intercambiado cuando algún proceso envía una señal a otro. En UNIX System V hay definidas 19 señales, 4.3BSD, Mac y Linux tienen 31, mientras que Solaris soporta 38. Las señales están prácticamente en todas las versiones de UNIX, y a éstas cada fabricante le añade las que considera necesarias. Podemos clasificar las señales en los siguientes grupos:

- Señales relacionadas con la terminación de procesos.
- Señales relacionadas con las excepciones inducidas por los procesos. Por ejemplo, el intento de acceder fuera del espacio de direcciones virtuales, los errores producidos al manejar número de punto flotante, etc.
- Señales relacionadas con los errores irrecuperables originados en el transcurso de una llamada al sistema.
- Señales originadas desde un proceso que se está ejecutando en modo usuario. Por ejemplo, cuando un proceso envía una señal a otro, vía `kill`, cuando un proceso activa un temporizador y se queda en espera de la señal de alarma, etc.
- Señales relacionadas con la interacción con la terminal. Por ejemplo, pulsar la tecla break.
- Señales para ejecutar un proceso paso a paso. Son usadas por los depuradores.

En el archivo de cabecera `<signal.h>` están definidas las señales que pueden ser manejar por el sistema y sus nombres.

Señal	Significado
SIGHUP	Desconexión. Es enviada cuando una terminal se desconecta de todo proceso del que es terminal de control. También se envía a todos los procesos de un grupo cuando el líder del grupo termina su ejecución. La acción por defecto de esta señal es terminar la ejecución del proceso que la recibe.
SIGINT	Interrupción. Se envía a todo proceso asociado con una terminal de control cuando se pulsa la tecla de interrupción (`CTRL + C`). Su acción por defecto es terminar la ejecución del proceso que la recibe.
SIGQUIT	Salir. Similar a `SIGINT`, pero es generada al pulsar la tecla de salida (`CTRL + \`). Su acción por defecto es genera un archivo `core` y terminar el

	proceso.
SIGILL	Instrucción ilegal. Es enviada cuando el hardware detecta una instrucción ilegal. En los programas escritos en C/C++ suele producirse este tipo de errores cuando manejamos apuntadores a funciones que no han sido correctamente inicializados. Su acción por defecto es general un archivo `core` y terminar el proceso.
SIGTRAP	`Trace trap`. Es enviada después de ejecutar cada instrucción, cuando el proceso se está ejecutando paso a paso. Su acción por defecto es genera un archivo `core` y terminar el proceso. Es utilizado cuando se utiliza un depurador.
SIGIOT	`I/O trap instruction`. Se envía cuando se da un fallo de hardware. La naturaleza de este fallo depende de la computadora. Es enviada cuando llamamos a la función `abort`, que provoca el suicidio del proceso generando un archivo `core`.
SIGEMT	`Emulator trap instruction`. También indica un fallo de hardware. Raras veces se utiliza. Su acción por defecto es generar un archivo `core` y terminar el proceso.
SIGFPE	Error de punto flotante. Es enviada cuando el hardware detecta un error de punto flotante, como el uso de un número de punto flotante con una forma desconocido, errores de desbordamiento, etc. Su acción por defecto es generar un archivo `core` y terminar el proceso.
SIGKILL	Terminación abrupta. Esta señal provoca irremediablemente la terminación del proceso. No puede ser ignorada y siempre que se recibe se ejecuta su acción por defecto, que consiste en generar un archivo `core` y terminar el proceso.
SIGBUS	Error de bus. Se produce cuando se da un error de acceso a memoria. Las dos situaciones típicas que la provocan suelen ser acceder a una dirección que físicamente no existe o intentar acceder a una dirección impar, violando así las reglas de alineación que impone el hardware. Su acción por defecto es generar un archivo `core` y terminar el proceso.
SIGSEGV	Violación de segmento. Es enviada a un proceso cuando intenta acceder a datos que se encuentran fuera de su segmento de datos. Su acción por defecto es generar un archivo `core` y terminar el proceso.
SIGSYS	Argumento erróneo en una llamada al sistema. No se usa.
SIGPIPE	Intento de escritura en un pipe del que no hay nadie leyendo. Esto suele ocurrir cuando el proceso de lectura termina de una forma anormal. Su acción por defecto es generar un archivo `core` y terminar el proceso.
SIGALRM	Despertador. Es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso. Se basa en el tiempo real.
SIGTERM	Finalización controlada. Es la señal utilizada para indicarle a un proceso que debe terminar su ejecución. Esta señal no es tajante como `SIGKILL` y puede ser ignorada. Lo correcto es que la rutina de tratamiento de esta señal se encargue de tomar las acciones necesarias antes de terminar un proceso (como, por ejemplo, borrar los archivos temporales) y llame a la

	rutina `exit`. Esta señal es enviada a todos los procesos cuando se emite una orden `shutdown`. Su acción por defecto es terminar el proceso.
SIGUSR1	Señal número 1 de usuario. Esta señal está reservada para uso del programador. Ninguna aplicación estándar va a utilizarla y su significado es el que le quiera dar el programador en su aplicación. Su acción por defecto es terminar el proceso.
SIGUSR2	Señal número 2 de usuario. Su significado es idéntico al de `SIGUSR1`.
SIGCLD/SIGCHLD	Terminación del proceso hijo. Es enviada al proceso padre cuando alguno de sus procesos hijos termina. Esta señal es ignorada por defecto.
SIGPWR	Fallo de alimentación. Esta señal tiene diferentes interpretaciones. En algunos sistemas es enviada cuando se detecta un fallo de alimentación y le indica al proceso que dispone tan sólo de unos instantes de tiempo antes de que se produzca una caída del sistema. En otros sistemas, esta señal es enviada, después de recuperarse de un fallo de alimentación, a todos aquellos procesos que estaban en ejecución y que se han podido rearrancar. En estos casos, los procesos deben disponer de mecanismos para restaurar las posibles pérdidas producidas durante la caída de la alimentación.
SIGVTALRM	Despertador. Es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso. Se basa en el tiempo que el proceso está en modo usuario. Se puede usar junto con `SIGPROF` para medir el tiempo usado por el proceso en modo usuario y kernel.
SIGPROF	Despertador. Es enviada a un proceso cuando alguno de sus temporizadores descendentes llega a cero. Su acción por defecto es terminar el proceso. Se basa en el tiempo que el proceso está en modo kernel. Se puede usar junto con `SIGVTALRM` para medir el tiempo usado por el proceso en modo usuario y kernel.
SIGIO/SIGPOLL	Señal de entrada/salida asíncrona. Cuando se realiza la operación `I_SETSIG` sobre un descriptor de archivos a través de la llamada `ioctl`, el kernel envía esta señal cuando un evento ocurre sobre el descriptor, por ejemplo, cuando existen datos que leer/escribir. Indica que un dispositivo o archivo está listo para una operación de entrada/salida. Su acción por defecto es ignorar la señal.
SIGWINCH	Cambio del tamaño de una ventana. Se usa en interfaces gráficas orientadas a ventanas como X-WINDOWS. Su acción por defecto es ignorar la señal.
SIGSTOP	Señal de pausa de un proceso. Esta señal no proviene de una terminal de control. La señal no puede ser ignorada, ni capturada y su acción por defecto es pausar el proceso. El proceso solo podrá continuar después de haber recibido la señal de `SIGCONT`.
SIGTSTP	Señal de pausa procedente de una terminal. Es generado por el teclado (tecla `SUSP` o `CTRL – Z`). El proceso solo podrá continuar después de haber recibido la señal de `SIGCONT`. A diferencia de `SIGSTOP` esta señal puede ser ignorada o manejada. Su acción por defecto es ignorar la señal.

SIGCONT	Continuar. Señal para reanudar la ejecución de un proceso. Su acción por defecto es ignorar la señal.
SIGTTIN	La reciben los procesos que se ejecutan en segundo plano y que intentan leer datos de una terminal de control. Su acción por defecto es parar el proceso.
SIGTTOU	La reciben los procesos que se ejecutan en segundo plano y que intentan escribir en una terminal de proceso. Su acción por defecto es parar el proceso.
SIGURG	Indica que ha llegado un dato urgente a través de un canal de entrada/salida asíncrona. Un ejemplo sería cuando hay datos que leer de un descriptor de archivo que está conectado a un socket. Su acción por defecto es ignorar la señal.
SIGXCPU	Le indica al proceso que la recibe que ha superado su tiempo de CPU asignado. Su acción por defecto es ignorar la señal.
SIGXFSZ	La indica al proceso que la recibe que superado el tamaño máximo de archivo que puede manejar. Su acción por defecto es terminar el proceso.

7.3 Manejo de Señales

Ahora estudiaremos el manejo de señales que brinda UNIX.

7.3.1 Envío de señales – kill y raise

Para enviar una señal desde un proceso a otro o a un grupo de procesos, emplearemos la llamada `kill`. Su declaración es:

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

`pid` identifica el conjunto de procesos al que queremos enviarle la señal. `pid` es un número entero y los distintos valores que puede tomar tienen los siguientes valores.

PID	Significado
pid > 0	Es el `PID` del proceso al que le enviamos la señal.
pid = 0	La señal es enviada a todos los procesos que pertenecen al mismo grupo que el proceso que la envía.
pid = -1	La señal es enviada a todos aquellos procesos cuyo identificador real es igual al identificador efectivo del proceso que la envía. Si el proceso que la envía tiene identificador efectivo de superusuario, la señal es enviada a todos los procesos, excepto al proceso 0 (`swapper`) y al proceso 1 (`init`).
pid < -1	La señal es enviada a todos los procesos cuyo identificador de grupo coincide con el valor absoluto de `pid`.

En todos los casos, si el identificador efectivo del proceso no es el del superusuario o si el proceso que envía la señal no tiene privilegios sobre el proceso que la va a recibir, la llamada a `kill` falla.

`sig` es el número de la señal que queremos enviar. Si `sig` vale 0 (señal nula), se efectúa una comprobación de errores, pero no se envía ninguna señal. Esta opción se puede utilizar para verificar la validez del identificador `pid`.

Si el envío se realiza satisfactoriamente, `kill` devuelve 0; en caso contrario, devuelve -1 y en `errno` estará el código del proceso producido.

En el siguiente ejemplo vemos cómo un proceso envía una señal a su proceso hijo para forzar su terminación.

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

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

	if ((pid = fork()) == 0) {
		while (1) {
			fprintf(stdout, "HIJO. PID = %d\n", getpid());
			sleep(1);
		}
	}
	sleep(10);
	fprintf(stdout, "PADRE. Terminacion del proceso %d\n", pid);
	kill(pid, SIGTERM);
	exit(0);
}
```

Si queremos que un proceso se envíe señales a sí mismo, podemos usar la llamada `raise`. Su declaración es:

```
#include <signal.h>
int raise(int sig);
```

`sig` es el número de la señal que queremos enviar. `raise` se puede codificar a partir de `kill` de la siguiente manera:

```
int raise(int sig) {
	return kill(getpid(), sig);
}
```

7.3.2 Tratamiento de Señales

Para especificar qué tratamiento debe realizar un proceso al recibir una señal, se emplea la función `signal`. Su declaración es la siguiente:

```
#include <signal.h>
void (*signal (int sig, void (*action) (int))) (int);
```

`signal` tiene una declaración que puede resultarnos extraña, pero es frecuente en los programas C/C++. Como podemos ver, `signal` es del tipo "*función que devuelve un apuntador a una función* `void` *que recibe parámetros*".

`sig` es el número de la señal sobre la que queremos especificar la forma de tratamiento.

`action` es la acción que queremos que se inicie cuando se reciba la señal. `action` puede tomar tres tipos de valores:

Valores	Significado
SIG_DFL	Indica que la acción a realizar cuando se recibe la señal es la acción por defecto asociada a la señal (manejador por defecto). Por lo general, esta acción es terminar el proceso y en algunos casos también incluye generar el archivo `core`.
SIG_IGN	Indica que la señal se debe ignorar.

dirección	Es la dirección de la rutina de tratamiento de la señal (manejador suministrado por el usuario). La declaración de esta función debe ajustarse el siguiente modelo.
	`#include <signal.h>`
	`void handler(sig [, code, scp])`
	`int sig, code;`
	`struct sigcontext *scp;`

Cuando se recibe la señal `sig`, el kernel es quien se encarga de llamar a la rutina `handler` pasándole los parámetros `sig`, `code` y `scp`. `sig` es el número de la señal, `code` es una palabra que contiene la información sobre el estado del hardware en el momento de invocar a `handler` y `scp` contiene información de contexto definida en `<signal.h>`. Tanto `code` como `scp` son parámetros opcionales y dependientes de la arquitectura de la computadora

La llamada a la rutina `handler` es asíncrona, lo cual quiere decir que puede darse en cualquier instante de la ejecución del programa. Esta rutina debe estar codificada para tratar las situaciones especiales que ocasionan que se produzca el envío de las señales.

La llamada a `signal` devuelve el valor que tenía `action`, que puede servirnos para restaurarlo en cualquier instante posterior. Si se produce algún error, `signal` devuelve SIG_ERR y en `errno` estará el código del error producido.

Los valores del `SIG_DFL`, `SIG_IGN` y `SIG_ERR` son direcciones de funciones ya que los debe poder devolver `signal`. Sin embargo, deben ser direcciones que sepamos que nunca van a estar ocupadas por otras funciones. Para darle solución a este problema, las constantes anteriores se definen de la siguiente forma:

```
#define SIG_DFL ((void (*) ()) 0)
#define SIG_IGN ((void (*) ()) 1)
#define SIG_ERR ((void (*) ()) -1)
```

La conversión explícita de tipo que aparece delante de las constantes -1, 1 y 0 fuerza a que estas constantes sean tratadas como direcciones de inicio de funciones. Estas direcciones no van a contener ninguna función, ya que en todas las arquitecturas UNIX son zona reservada para el kernel. Además, la dirección -1 ni siquiera tiene existencia física.

Como primer ejemplo de tratamiento de señales, vamos a codificar un programa que trata la señal de `SIGINT`, generada al pulsar las teclas `CTRL+C` (interrupción).

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

void sigint_handler(int sig) {
	fprintf(stdout, "Señal número %d.\n", sig);
}

int main(int argc, char *argv[]) {
	if (signal(SIGINT, sigint_handler) == SIG_ERR) {
		perror("signal");
		exit(-1);
	}
	while (1) {
		fprintf(stdout, "En espera de CTRL+C\n");
		sleep(30);
	}
	exit(0);
}
```

7.3.3 Señales Inestables

En las primeras versiones del UNIX, las señales no eran confiables. Esto quiere decir, que la señales podían perderse: sí una señal ocurría, el proceso quizás no podría enterarse nunca. También, un proceso tenía poco control sobre una señal: solo podía cacharla o ignorarla, no es posible bloquearla.

Un problema que ocurría en las primeras versiones es que la acción para una señal era restaurada a la acción por defecto que tenía asignada la señal. El siguiente es un ejemplo clásico de un código utiliza para manejar una señal.

```
void sigint_handler(int sig) {
	…
	if (signal(SIGINT, sigint_handler) == SIG_ERR) {
		…
	}
}

int main(int argc, char *argv[]) {
	if (signal(SIGINT, sigint_handler) == SIG_ERR) {
		…
	}
}
```

El problema con este fragmento de código es que existe una ventana de tiempo después de que la señal ha ocurrido y se realiza la siguiente llamada a `signal`. Durante este tiempo puede ocurrir una segunda señal. Esta segunda señal en vez de utilizar el manejador asignado, realizaría la acción por defecto.

Otro problema con las versiones anteriores es que el proceso no es capaz de apagar una señal cuando no queremos que esa señal no suceda. Lo que se podía hacer era ignorar la señal. Pero existen ocasiones en que nosotros le queremos decir al sistema: "previene que las siguientes señales ocurran, pero recuérdame si suceden". El clásico ejemplo que demuestra esta falla está indicado en el segmento de código que captura una señal y establece una bandera que indica que la señal ha ocurrido:

```
int sig_int_flag = 0;

void sigint_handler(int sig) {
	…
	if (signal(SIGINT, sigint_handler) == SIG_ERR) {
		…
	}
	sig_int_flag ) 1;
}

int main(int argc, char *argv[]) {
	if (signal(SIGINT, sigint_handler) == SIG_ERR) {
		…
	}
	while (sig_int_flag == 0) {
		pause();
	}
}
```

Aquí, el proceso utiliza la llamada `pause` para dormirse hasta que una señal es capturada. Cuando la señal es capturada, el manejador establece la bandera a un valor diferente de cero. El proceso es automáticamente despertado por el kernel después de que el manejador termina, nota que la bandera es diferente de cero y hacer todo lo que tiene que hacer. Pero, existe una ventana de tiempo donde todo puede ir mal. Si la señal ocurre después de verificar el valor de la bandera, pero antes de la llamada a `pause`, el proceso podría dormir por siempre (asimiento que la señal nueva es generara nuevamente), ya que esta ocurrencia de la señal es ignorada.

7.3.4 En Espera de Señales – Pause

En ocasiones puede interesar que un proceso suspenda su ejecución en espera de que ocurra algún evento exterior a él. Por ejemplo, al ejecutar una entrada/salida. Para estas situaciones nos valemos de la llamada a pause, cuya declaración es la siguiente:

```
#include <unistd.h>
pause();
```

`pause` hace que el proceso quede en espera de la llegada de alguna señal. Cuando esto ocurre, y después de ejecutarse la rutina de tratamiento de la señal, pause devuelve el valor -1 y en errno sitúa el valor `EINTR` que significa que se ha producido una interrupción de la llamada. En otras llamadas al sistema, ésta es una condición de error, pero en el caso de pause es su forma correcta de operar. El programa continúa con la sentencia que sigue a pause.

El programa siguiente es un ejemplo en el que un proceso está esperando una señal y cuando recibe la señal `SIGUSR1` presenta en pantalla un número aleatorio.

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

void sigterm_handler(int sig) {
	fprintf(stdout, "Terminación del proceso %d a petición del usuario.\n",
getpid());
	exit(-1);
}

void sigusr1_handler(int sig) {
	signal(sig, SIG_IGN);
	fprintf(stdout, "%d\n", rand());
	signal(sig, sigusr1_handler);
}

int main(int argc, char * argv[]) {
	fprintf(stdout, "PID = %d\n", getpid());
	signal(SIGTERM, sigterm_handler);
	signal(SIGUSR1, sigusr1_handler);
	while (1) {
		pause();
	}
	exit(0);
}
```

Para ejecutar este programa y que muestre los números aleatorios, debemos enviarle la señal `USR1` con la orden `kill`,

```
$ kill –s USR1 pid
```

Podemos terminar la ejecución del proceso enviándole la señal `SIGTERM`.

```
$ kill –s TERM pid
```

7.3.5 Saltos Globales – SIGsetjmp y SIGlongjmp

En párrafos anteriores hemos indicado que la rutina de tratamiento de una señal puede hacer que el proceso vuelve a alguno de los estados por los que ha pasado con anterioridad. Esto no sólo es aplicable a las rutinas de tratamiento de señales, sino que se puede extender a cualquier función. Para realizar esto nos valemos de las funciones estándar `sigsetjmp` y `siglongjmp`. Sus declaraciones las podemos ver a continuación:

```
#include <setjmp.h>
int sigsetjmp (jmp_buf env, int savemask);
void siglongjmp (jmp_buf env, int val);
```

Si savemask es 0, sigsetjmp guarda el entorno de pila en env para un uso posterior de siglongjmp. sigsetjmp devuelve el valor 0 en su primera llamada. El tipo de env, jmp_buf, está definido en el archivo cabecera <setjmp.h>.

siglongjmp restaura el entorno guardado en env por una llamada previa a sigsetjmp. Después de haberse ejecuta la llamada a siglongjmp, el flujo de la ejecución del programa vuelve al punto donde se hizo la llamada a sigsetjmp. Pero en este caso sigsetjmp devuelve el valor val que hemos pasado mediante siglongjmp. Esta es la forma de averiguar si sigsetjmp está saliendo de una llamada para guardar el entorno o de una llamada siglongjmp. siglongjmp no puede hacer que sigsetjmp devuelva 0, ya que en el caso de que val sea igual a 0, sigsetjmp va a regresa 1.

En la siguiente figura podemos ver representa la forma de trabajar de sigsetjmp y siglongjmp.

Las funciones sigsetjmp y siglongjmp se puede ver como una forma elaborada de implementa una sentencia goto capaz de saltar desde una función a etiquetas que están en la misma o en otra función. Las etiquetas serían los entornos guardados por sigsetjmp en la variable env.

Como ejemplo, vamos a ver un programa que cuenta desde 1 hasta 10 incrementando su valor cada 10 segundos. Además, cada 10 segundos se va a establecer un punto de retorno de tal forma que, si se recibe la señal SIGUSR1 en algún momento, el programa va a reiniciar su ejecución en ese punto.

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

jmp_buf env;

void sigusr1_handler(int sig) {
	//signal(SIGUSR1, sigusr1_handler);
	siglongjmp(env, 1);
}

int main(int argc, char * argv[]) {
	int i;

	fprintf(stdout, "PID = %d\n", getpid());
	signal(SIGUSR1, sigusr1_handler);
	for (i = 0; i < 10; i++) {
		if (sigsetjmp(env, 1) == 0) {
			fprintf(stdout, "Punto de retorno en el estado %d.\n", i);
		} else {
			fprintf(stdout, "Regresando al punto de retorno en el estado
%d.\n", i);
		}
		sleep(10);
	}
```

```
}
```

7.3.6 Función sigaction

La función `sigaction` nos permite examinar o modificar (o ambas) la acción asociada con una señal en particular. Esta función reemplaza a la llamada `signal` de las primeras versiones de UNIX. De hecho, es posible implementar `signal` usando `sigaction`.

Su declaración es la siguiente:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict act,
                  struct signaction *restrict oact);
```

`signo` es el número de señal cuya acción que queremos analizar o modificar. `act` es un apuntador a una estructura del tipo `sigaction`. Si el valor de `act` es diferente de nulo significa que queremos modificar la acción y el kernel nos regresa la acción previa a través del apuntador `oact`. La definición de la estructura `sigaction` es la siguiente:

```
struct sigaction {
      void (*sa_handler) (int); /* Dirección del manejador */
                                      /* o SIG_IGN, o SIG_DFL */
      sigset_t sa_mask;          /* Señales adicionales a bloquear. */
      int sa_flags;       /* Opciones adicionales. */
      void (*sa_sigaction) (int, siginfo_t *, void *);
                                 /* manejador alternativo. */
};
```

El campo `sa_handler` es un apuntador a una función que devuelve tipo `void` y tiene el mismo significado que el parámetro `action` de la llamada `signal`. Este campo se utiliza para indicar cuál va a ser la rutina de tratamiento de la señal. Al igual que en el caso de `signal`, puede tomar tres valores con diferente significado:

Valores	Significado
SIG_DFL	La rutina de tratamiento va a ser la rutina de tratamiento por defecto.
SIG_IGN	Indica que la señal se debe ignorar.
dirección	La rutina de tratamiento empieza en la dirección indicada y ha sido codificada por el usuario.

El campo `sa_mask` codifica, en cada uno de sus bits, las señales que no deseamos que han tratadas si sin recibidas mientras se está ejecutando la rutina de tratamiento actual. Normalmente, este campo está en 0, lo que indica que mientras está tratando una señal, cualquier otra señal puede interrumpir. Si alguno de los bits de `sa_mask` está en 1, vamos a impedir el anidamiento cuando se recibe esa señal. Cuando el manejador termina, la máscara de señales del proceso es regresado a su estado previo.

El campo `sa_flags` codifica cuál va a ser la semántica (significado) que se emplee en la recepción de la señal. Los siguientes bits están definidos para este campo:

Valores	Significado
SA_INTERRUPT	Las llamadas a sistema que son interrumpidos por esta señal no son automáticamente reiniciadas.
SA_NOCLDSTOP	Si signo es `SIGCHLD`, no genera esta señal cuando un proceso hijo se detiene por un control de trabajo (`SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`). Esta señal, sin embargo, es generada cuando el hijo termina.
SA_NOCLDWAIT	Si signo es `SIGCHLD`, esta opción previene al sistema de crear procesos zombis. El proceso que ejecuta la llamada se bloquea hasta que todos sus hijos hayan terminado y entonces regresa 1 y `errno` es igual a `ECHILD`.
SA_NODEFER	Cuando esta señal es capturada, la señal no es automáticamente bloqueada por el sistema mientras el manejador se ejecuta (a menos que la señal esté incluida en `sa_mask`). Este comportamiento es el equivalente al de las primeras versiones de UNIX.
SA_ONSTACK	Si una pila alterna ha sido declarada con `sigaltstack`, la señal es entregada al proceso en esa pila alterna.
SA_RESETHAND	Esto hace que el manejador de la señal sea nuevamente el manejador por defecto (`SIG_DFL`).
SA_RESTART	Las llamadas a sistema que son interrumpidas por esta señal son automáticamente reiniciadas.
SA_SIGINFO	Esta opción provee información adicional al manejador: aun apuntador a una estructura `siginfo_t` y un apuntador a un identificador del contexto del proceso.

El campo `sa_sigaction` es un manejador alterno de la señal usado cuando `SA_SIGINFO` es utilizado con la llamada a `sigaction`. Normalmente, el manejador de la señal es:

```
void handler(int signo);
```

Pero si `SA_SIGINFO` es usado, la definición del navegador debe ser:

```
void handler(int signo, siginfo_t *info, void *context);
```

La estructura `siginfo_t` contiene información acerca del porque la señal fue generada.

```
struct siginfo_t {
    int      si_signo;  /* Número de la señal */
    int      si_errno;  /* Número de error */
    int      si_code;   /* Código de la señal */
    pid_t    si_pid;    /* ID del proceso que envía la señal */
    uid_t    si_uid;    /* ID real del proceso que envía la señal */
    int      si_status; /* valor de salida o señal */
    clock_t  si_utime;  /* Tiempo de usuario consumido */
    clock_t  si_stime;  /* Tiempo de sistema consumido */
    sigval_t si_value;  /* Valor de la señal */
    int      si_int;    /* Señal POSIX.1b */
    void *   si_ptr;    /* Señal POSIX.1b */
    void *   si_addr;   /* Localidad de memoria que ha causado
                           la señal */
```

```
    int      si_band;   /* Número de banda del evento */
    int      si_fd;     /* Descriptor del archivo */
};
```

En el siguiente programa veremos las tres formas de tratamiento de una señal: con un manejador, ignorando la señal o con la rutina por defecto. Este programa se envía a sí mismo en repetidas ocasiones la señal `SIGUSR1`; según la rutina de tratamiento que esté instalada, la respuesta será una u otra.

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

void sigusr1_handler(int sig) {
        fprintf(stdout, "sigusr1_handler: señal recibida: %d.\n", sig);
}

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

        action.sa_handler = sigusr1_handler;
        sigemptyset(&action.sa_mask);
        action.sa_flags = 0;
        sigaction(SIGUSR1, &action, 0);
        fprintf(stdout, "Envio de señal de SIGUSR1. Manejador activo.\n");
        kill(getpid(), SIGUSR1);
        fprintf(stdout, "Envio de señal de SIGUSR1. Manejador activo.\n");
        kill(getpid(), SIGUSR1);

        action.sa_handler = SIG_IGN;
        sigaction(SIGUSR1, &action, 0);
        fprintf(stdout, "Ignorar activo.\n");
        kill(getpid(), SIGUSR1);
        fprintf(stdout, "Ignorar activo.\n");
        kill(getpid(), SIGUSR1);

        action.sa_handler = SIG_DFL;
        sigaction(SIGUSR1, &action, 0);
        fprintf(stdout, "Rutina de tratamiento por defecto activo.\n");
        kill(getpid(), SIGUSR1);

        fprintf(stdout, "Este mensaje no aparece en pantalla.\n");
        exit(0);
}
```

Otro ejemplo que hacer con `sigaction` es la implementación de la función `signal`:

```
typedef void function(int);

function* signalX(int sig, function *fun) {
        struct sigaction act, oact;

        act.sa_handler = fun;
        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;
        if (sig == SIGALRM) {
                #ifndef SA_INTERRUPT
                act.sa_flags |= SA_INTERRUPT;
                #endif
```

```
        } else {
                #ifndef SA_RESTART
                act.sa_flags |= SA_RESTART;
                #endif
        }
        if (sigaction(sig, &act, &oact) < 0) {
                return (SIG_ERR);
        }
        return oact.sa_handler;
}
```

7.3.7 FuncionEs para manejar Conjuntos de señales

Existe un tipo de dato (`sigset_t`) que nos permite manejar un conjunto de señales. Con este tipo de dato usaremos algunas funcionas como `sigprocmask` para decirle al kernel que no permita que ninguna de las señales incluidas en este conjunto ocurra. Para manipular estos conjuntos de señales tenemos 5 funciones:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
int sigismember(const sigset_t *set, int sig);
```

La función `sigemptyset` inicializa a `set` con un valor que indica que todas las señales son excluidas. La función `sigfillset` inicializa a `set` con un valor que indica que todas las señales van a ser incluidas.

Una vez que hemos inicializado el conjunto de señales, podemos agregar o eliminar señales de ese grupo. La función `sigaddset` nos permite agregar una señal al conjunto, mientras que `sigdelset` nos permite eliminar una señal del conjunto.

Por último, `sigismember` nos permite determinar si una señal específica se encuentra defina dentro del conjunto.

7.3.8 Función sigprocmask

Cada proceso tiene una máscara de señales que define el conjunto de señales que actualmente no se están entregando al proceso (bloqueadas). Un proceso puede examinar y/o cambiar esta máscara a través de la llamada a `sigprocmask`. Su definición es la siguiente:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *restrict set,
                    sigset_t *restrict oset);
```

Primero, si `oset` es un apuntador no nulo, la función regresa la máscara actual del proceso a través de `oset`.

Segundo, si `set` es un apuntador no nulo, el argumento `how` indica cómo es que la máscara actual será modificada. Los posibles valores que puede tomar `how` son:

Valores	Significado
SIG_BLOCK	La nueva máscara del proceso es la unión entre la máscara actual y el conjunto indicado por `set`. Es decir que set tiene señales adicionar que queremos bloquear.
SIG_UNBLOCK	La nueva máscara para el proceso es la intersección de la máscara actual y el conjunto indicado por `set`. Es decir que set tiene señales que queremos desbloquear.
SIG_SETMASK	La nueva señal es la que está siendo indicar por `set`.

Si set es un apuntador nulo, la máscara de señales no es cambiada y `how` es ignorado.

Si cuando se ejecuta la llamada a `sigprocmask` existe una señal pendiente de ser atendida, se entregará al proceso. Y éste la deberá atender antes de que la función termine.

7.3.9 Función sigpending

La función `sigpending` regresa el conjunto de señales que están siendo bloqueadas y que, por lo tanto, no han sido entregadas al proceso. Este conjunto de señales es devuelto a través del argumento set.

```
#include <signal.h>
int sigpending(sigset_t * set);
```

El siguiente programa empieza bloquea la señal `SIGQUIT`, guardando antes la máscara de señales (para poder restaurarla después) y luego se va a dormir por 5 segundos. Cualquier ocurrencia de la señal `quit` durante este periodo de tiempo es bloqueado y no se entregará al proceso hasta que la desbloquee. Al final de los 5 segundos, reinstalamos la máscara original y, al mismo tiempo, checamos que existe una señal que esté pendiente de ser entregada.

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

void sigquit_handler(int sig) {
        fprintf(stdout, "Capturando SIGQUIT.\n");
        if (signal(SIGQUIT, SIG_DFL) == SIG_ERR) {
                perror("No se puede restablecer SIGQUIT");
                exit(1);
        }
}

int main(int argc, char *argv[]) {
        sigset_t newmask, oldmask, pendmask;

        if(signal(SIGQUIT, sigquit_handler) == SIG_ERR) {
                perror("no puede capturarse SIGQUIT");
                exit(1);
        }

        sigemptyset(&newmask);
        sigaddset(&newmask, SIGQUIT);
```

```
        if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
                perror("SIG_BLOCK");
                exit(1);
        }

        sleep(5);
        if (sigpending(&pendmask) < 0) {
                perror("sigpending");
                exit(1);
        }
        if (sigismember(&pendmask, SIGQUIT)) {
                fprintf(stdout, "\nSIGQUIT pendiente.\n");
        }

        if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
                perror("SIG_SETMASK");
                exit(1);
        }
        fprintf(stdout, "\nSIGQUIT desbloqueado.\n");

        sleep(5);
        return 0;
}
```

7.3.10 Función sigsuspend

Anteriormente visto que es posible cambiar la máscara de señales para bloquear o desbloquear alguna en particular. Podemos usar esta técnica para proteger regiones críticas de código que no queremos que sean interrumpidas por una señal. El siguiente código intenta hacer esto, para ellos que la señal a ocurrir es `SIGINT`.

```
sigset_t newmask, oldmask;

sigemptyset(&newmask);
sigaddset(&newmask, SIGINT);

if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
        perror("SIG_BLOCK");
        exit(1);
}

/* región crítica de código */

if (sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
        perror("SIG_SETMASK");
        exit(1);
}

/* ventana abierta */
pause();

/* continua el proceso */
```

Si la señal es enviada al proceso mientras está bloqueada, la señal será entregada hasta que sea desbloqueada. Para la aplicación, esto puede parecer como si la señal ocurriera entre el desbloqueo y pause (dependiendo de cómo el kernel implementa las señales). Si esto sucede, o si la señal sucede realmente en ese punto, estamos en un problema. Cualquier ocurrencia de la señal en esta ventana de tiempo se pierde en el sentido de que quizás no se capture esa señal otra vez,

haciendo que pause bloquee el proceso de manera indefinida. Este es otro problema que se tenía con las primeras implementaciones de señales.

Para corregir este problema, necesitamos de una función que nos permita reinicializar la máscara de señales y, al mismo tiempo, poner el proceso a dormir en una operación atómica. Esta característica nos la provee la función `sigsuspend`.

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```

La máscara de señal del proceso es establecida con el valor al que hace referencia el apuntado `sigmask`. Entonces el proceso es suspendido hasta que una señal es capturada o hasta que ocurra una señal que termine el proceso. Si la señal es capturada, el control es pasado al manejador. Cuando el manejador termina, entonces `sigsuspend` termina y la máscara de señal del proceso es restablecida al valor que tenía hasta antes del `sigsuspend`.

Esta función no regresa un valor exitoso de terminación. Si la función regresa el control al proceso, siempre devuelve 1 y la variable `errno` es igualada a `EINTR` (indicando una llamada a sistema interrumpida).

El siguiente programa es un ejemplo de su uso:

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

void sigint_handler(int sig) {
        fprintf(stdout, "\nsig_int: señal recibida %d\n", sig);
}

int main(int argc, char *argv[]) {
        sigset_t newmask, oldmask, waitmask;

        fprintf(stdout, "inicia programa.\n");

        if (signal(SIGINT, sigint_handler) == SIG_ERR) {
                perror("SIGINT");
                exit(1);
        }
        sigemptyset(&waitmask);
        sigaddset(&waitmask, SIGUSR1);
        sigemptyset(&newmask);
        sigaddset(&newmask, SIGINT);

        if(sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0) {
                perror("SIG_BLOCK");
                exit(1);
        }

        fprintf(stdout, "región crítica.\n");
        if (sigsuspend(&waitmask) != -1) {
                perror("WAIT");
                exit(1);
        }
        sleep(30);
```

```
        fprintf(stdout, "terminando región crítica.\n");

        if(sigprocmask(SIG_SETMASK, &oldmask, NULL) < 0) {
                perror("SIG_SETMASK");
                exit(1);
        }

        fprintf(stdout, "termina programa.\n");
        exit(0);
}
```

Al ejecutar este programa, tendremos siguiente comportamiento:

```
$ ./sigsuspend
inicia programa - PID 5249.
región crítica.
^C
sig_int: señal recibida 2
^C
^C
^C
terminando región crítica.

sig_int: señal recibida 2
termina programa.
```

Como pueden ver, mientras el proceso está en región crítica la señal es bloqueada. Es hasta después de que termina la región, cuando se vuelve a aceptar la señal de SIGINT.

7.3.11 Función abort

La llamada `abort` causa una terminación anormal del proceso. Su definición es la siguiente:

```
#include <stdlib.h>
void abort(void);
```

Esta función manda una señal de `SIGABRT` al proceso que la invoca que no puede ser ignorada o bloqueada.

Sin embargo, se permite que la señal `SIGABRT` sea capturada con el fin de realizar un proceso de limpieza previo. Pero la única manera en que puede terminar el manejador es con una llamada a una función de la familia `exit`.

En caso de que el manejador no cumpla esta condición, `abort` se encarga de terminar el proceso.

7.3.12 Funciones adicionales

Algunos sistemas proveen el siguiente arreglo:

```
extern char *sys_siglist[];
```

El arreglo almacena el nombre de todas las señales que maneja el sistema. El índice del arreglo es el número de la señal.

Otra función que podemos encontrar es:

```
#include <signal.h>
void psignal(int sig, const char *msg);
```

El apuntador msg, que normalmente corresponde al nombre del programa, es colocado en la salida estándar de error seguido de una coma, un espacio, la descripción de la señal y un salto de línea. Esta función es similar a perror.

Por último, tenemos la función:

```
#include <string.h>
char* strsignal(int sig);
```

La cual nos regresa un apuntador a la descripción de la señal.

7.4 Funciones de Tiempo

En las siguientes secciones vamos a estudiar cómo consultar y controlar los tiempos asociados a un proceso. Si bien UNIX no es un sistema para aplicaciones en tiempo real, en condiciones de baja carga donde no está congestionado podremos controlar a nuestros programas para que se ejecuten con los tiempos de respuesta adecuados. Hay que tener presente que una respuesta en tiempo real no significa solamente que nuestro programa se ejecute muy rápido (para similares procesos físicos, por ejemplo), sino también que seamos capaces de controlar el tiempo para dar la respuesta en el momento exacto y no antes. Imaginemos, por ejemplo, un programa para simular un reloj. Si nuestra respuesta se tiene que dar cada segundo, de nada nos vale darla más deprisa.

La parte central de este tema va a estar dedicada a estudiar los mecanismos para poder dotar a nuestros programa de una temporización que va a estar impuesta por la naturaleza física del proceso que intentamos simular.

7.4.1 Establecimiento de la fecha del sistema

Si necesitamos fijar la fecha y hora locales de nuestro sistema, debemos usar la llamada `stime`. Su declaración es:

```
int stime(long *tp);
```

`tp` es un apuntador a un número que contiene la hora actual medida en segundos con respecto a las 00:00.00 GMT del día 1 de enero de 1970.

La fecha del sistema sólo la puede fijar un usuario con los privilegios del superusuario; por lo tanto, `stime` sólo se podrá ejecutar satisfactoriamente en aquellos procesos cuyo EID sea el de superusuario.

`stime` devuelve el valor 0 si se ejecuta satisfactoriamente y -1 en caso de error.

7.4.2 Lectura de la fecha del sistema – time y gettimeofday

Si queremos leer la fecha actual que almacena el sistema, podemos usar la llamada time. Su declaración es:

```
#include <time.h>
time_t time (time_t *tloc);
```

time devuelve el valor de la hora en segundos con respecto a las 00:00.00 GMT del día 1 de enero de 1970. Si tloc no es un apuntador nulo, el valor que devuelve time, también se copia en la dirección apuntada por tloc.

Si la llamada a time falla, la función devolverá el valor (time_t) (-1) y en errno estará el código del tipo de error producido.

Los saltos de tiempo que se producen a intervalos irregulares por necesidades de ajuste del calendario no que quedan reflejados en la hora del sistema. En el intervalo que va desde 1970 a 1988 se ha producido una variación de 14 segundos.

Si la resolución que ofrece time (segundos) no es suficiente y necesitamos una medida más exacta, podemos usar la llamada gettimeofday. Su declaración es la siguiente:

```
#include <time.h>
int gettimeofday(strutct timeval *tp, struct timezone *tzp);
```

gettimeofday devuelve, en la estructura apuntada por tp, la hora del sistema medida con respecto al meridiano de Greenwich, y en la estructura apuntada por tzp, las correcciones con respecto a la hora GMT para poder calcular la hora local. Esta hora estará en función de la zona de la Tierra donde nos encontremos.

timeval es una estructura definida en <time.h> y se declara de la siguiente forma:

```
struct timeval {
        unsigned long tv_sec; /* Segundos transcurridos. */
        long tv_usec; /* Microsegundos. Su rango está comprendido entre
                         0 y 999,999. */
};
```

timezone también está definida en <time.h> y sus campos son:

```
struct timezone {
        int tz_minuteswest; /* Variación de la hora local, en minutos,
                               con respecto a la hora GMT. */
        int tz_dsttime; /* Tipo de corrección a aplicar según las
                           estaciones (horario de invierno o de
                           verano. */
};
```

En la mayor parte de los programas no se utiliza el valor devuelto por tzp. Si está definida la variable de entorno TZ, no se debe emplear la corrección de hora indicada en tzp.

También se puede cambiar la hora del sistema mediante una llamada que maneja estructuras del tipo timeval y timezone. Esta llamada es settimeofday y su declaración es la siguiente:

```
#include <time.h>
int settimeofday(strutct timeval *tp, struct timezone *tzp);
```

Para poder usar `settimeofday` hay que tener privilegios de superusuario. Tanto `gettimeofday` como `setimeofday` devuelven el valor 0 si se han ejecutado satisfactoriamente y -1 en caso de error.

Hay que decir que la resolución del campo que mide los microsegundos (`tv_usec`) depende de la frecuencia del reloj del sistema, que suele ser bastante pequeña en comparación con la del reloj de la computadora.

Como ejemplo de uso de la estructura `timeval`, vamos a ver un programa para calcular el tiempo empleado en calcular 1, 000,000 de raíces cuadradas de los números positivos aleatorios.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>
#include <math.h>

int main(int argc, char *argv[]) {
        struct timeval t_inicio, t_fin, t_dif;
        struct timezone tz;
        unsigned long i;
        double x;

        gettimeofday(&t_inicio, &tz);

        for (i = 0; i < 1000000; i++) {
                x = sqrt((double) rand());
        }

        gettimeofday(&t_fin, &tz);

        if (t_inicio.tv_sec > t_fin.tv_sec) {
                t_fin.tv_sec += 1000000;
                --t_fin.tv_sec;
        }
        t_dif.tv_sec = t_fin.tv_sec - t_inicio.tv_sec;
        t_dif.tv_usec = t_fin.tv_usec - t_inicio.tv_usec;

        fprintf(stdout, "Tiempo usado = %lds, %ldus.\n",
                t_dif.tv_sec, t_dif.tv_usec);
        return 0;
}
```

El tiempo que presenta este programa depende de la carga del sistema. Si el programa no tiene otros procesos compitiendo por el CPU, tardará menos en ejecutarse que si tiene otros competidores. En el próximo punto vamos a ver cómo podemos leer el tiempo real de CPU que el proceso emplea en su ejecución.

7.4.3 Tiempos de ejecución asociados a un proceso – times

Para conocer el tiempo real empleado por un proceso en su ejecución, podemos usar la llamada `times`. Su declaración es:

```
#include <sys/times.h>
clock times(struct tms *buffer);
```

`times` rellena la estructura apuntada por buffer con la información estadísticas relativa a los tiempos de ejecución empleados por el proceso, desde su inicio hasta el momento de invocar times. La estructura `tms` se define de la siguiente forma:

```
struct tms {
        clock_t tms_utime; /* Tiempo de CPU en modo usuario. */
        clock_t tms_stime; /* Tiempo de CPU en modo supervisor. */
        clock_t tms_cutime; /* Tiempo de CPU en modo usuario de los
                               procesos hijo. */
        clock_t tms_cstime; /* Tiempo de CPU en modo supervisor de los
                               procesos hijo. */
};
```

El tipo `clock_t` se define para contabilizar los pulsos de reloj. Cada segundo se compone de un total de `CLOCKS_PER_SEC` pulsos. El valor de `CLOCKS_PER_SEC` depende de la implementación del sistema y se puede leer mediante la llamada `sysconf`. Para calcular el tiempo en segundos que almacena una variable de tipo `clock_t`, tenemos que dividirla entre `CLOCKS_PER_SEC`.

Los valores que aparecen en los campos de buffer se refieren al proceso que ejecuta la llamada a `times` y a todos aquellos procesos hijo para los cuales el proceso padre ejecuta una llama a `wait`. El significado de los campos buffer es el siguiente:

Valores	Significado
tms_utime	Tiempo empleado por el CPU ejecutando el procesos en modo usuario.
tms_stime	Tiempo empleado por el CPU
SIG_SETMASK	La nueva señal es la que está siendo indicar por `set`.

En el cómputo anterior de todos los tiempos no se tiene en cuenta el tiempo dedicado por los procesos del sistema a los procesos del usuario (por ejemplo, tiempo que emplea el sistema en hacer que un proceso usuario cambie de contexto). Los tiempos anteriores son tiempos reales de CPU, por lo que no se contabilizan los períodos en los que el proceso duerme.

Si `times` se ejecuta satisfactoriamente, devuelve el tiempo real transcurrido, en pulsos de reloj, contado a partir de un instante pasado arbitrario. Este instante puede ser el momento de arranque del sistema y no cambia de una llamada a otra. Si `times` falla, devolverá -1 y en `errno` estará el código del error producido.

Como ejemplo de aplicamos, vamos a implementar la orden estándar de UNIX `time`. `time` nos permite ver el tiempo empleado por un proceso en su ejecución. La forma de invocarla es:

```
$time línea_de_órdenes
```

`línea_de_órdenes` es cualquier otra orden o programa ejecutable. Un ejemplo de ejecución puede ser:

```
$time ls -laR /home
Tiempos de ejecución:
```

```
        Real: 0.000116 seg.
        En modo usuario: 1e-06 seg.
        En modo supervisor: 2.1e-05 seg.
```

En nuestra implementación, a la orden `time` la hemos llamado `tiempo` y su código es el siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <time.h>
#include <sys/times.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
        struct tms bt1, bt2;
        clock_t t1, t2;
        int pid, w, estado, tty;

        if (argc < 2) {
                fprintf(stderr, "Forma de uso: %s línea_de_órdenes\n", argv[0]);
                exit(-1);
        }

        if ((tty = open("/dev/tty", O_RDWR)) == 1) {
                perror(argv[0]);
                exit(-1);
        }

        t1 = times(&bt1);

        if ((pid = fork()) == 1) {
                perror(argv[0]);
                exit(-1);
        } else if (pid == 0) {
                /* codigo del proceso hijo */
                /* independizamos los descriptors básicos del hijo */
                close(0); dup(tty);
                close(1); dup(tty);
                close(2); dup(tty);
                /* ejecuta la instrucción */
                execvp(argv[1], &argv[1]);
                /* regresamos 127 en caso de que falle*/
                exit(127);
        } else {
                close(tty);
                /* deshabilitamos las señales para evitar que cualquier
                   señal que vaya al hijo afecte al padre */
                signal(SIGINT, SIG_IGN);
                signal(SIGQUIT, SIG_IGN);
                /* esperamos hasta que el hijo termine */
                while ((w = wait(&estado)) != pid && w != -1);
                t2 = times(&bt2);
                /* habilitamos nuevamente las señales */
                signal(SIGINT, SIG_DFL);
                signal(SIGQUIT, SIG_DFL);
                /* presentamos resultados */
                fprintf(stdout, "Tiempos de ejecución: \n");
```

```
                fprintf(stdout, " \tReal: %g seg.\n", (float) (t2- t1) /
CLOCKS_PER_SEC);
                fprintf(stdout, " \tEn modo usuario: %g seg.\n", (float)
(bt2.tms_cutime - bt1.tms_cutime) / CLOCKS_PER_SEC);
                fprintf(stdout, " \tEn modo supervisor: %g seg.\n", (float)
(bt2.tms_cstime - bt1.tms_cstime) / CLOCKS_PER_SEC);
        }
        return 0;
}
```

7.4.4 Temporizadores

En determinadas aplicaciones puede interesarnos que el código se ejecute de acuerdo con una temporización determinada. Esto puede venir impuesto por las especificaciones del programa, donde una respuesta antes de tiempo puede ser tan perjudicial como una respuesta con retraso.

La primera llamada al sistema que nos ayuda a resolver las necesidades de temporizadores es `alarm`. Su declaración es la siguiente:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

`alarm` activa un temporizador que inicialmente toma el valor `sec` segundos y que se va a decrementar en tiempo real. Cuando haya transcurrido los `sec` segundos, el proceso va a recibir la señal `SIGALRM`.

El valor de `sec` está limitado por la constante `MAX_ALARM`, definida en `<sys/param.h>`. En todas las implementaciones se garantiza que `MAX_ALARM` debe permitir una temporización de al menos 31 días (en segundos). Si `sec` toma un valor superior al de `MAX_ALARM`, se trunca al de esta constante.

Llamadas sucesivas a `alarm` restauran el valor del temporizador al de sec. Así, para cancelar un temporizador previamente declarado, haremos la llamada `alarm(0)`.

Es importante tener presente que una alarma no es heredada por un proceso hijo después de una llamada a `fork`, pero sí es heredada por los programas cargados con `exec`.

`alarm` devuelve el tiempo que le quedaba al temporizador que había sido definido con anterioridad.

Como ejemplo de aplicación, vamos a implementar la función estándar `sleep`. Esta función permite parar la ejecución de un proceso durante una cantidad de segundos determinada. Nuestra versión se llamará `dormir` y su declaración va a ser:

```
void dormir(unsigned int seg);
```

Donde `seg` es la cantidad de segundos que el proceso va a estar parado.

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

```
static jmp_buf entorno;

static void sigalarm_handler(int sig) {
        siglongjmp(entorno, 1);
}

void dormir(unsigned int seg) {
        unsigned int alarma_ant;
        void (*sigalarm_handler_ant) (int);

        if (seg == 0) {
                return;
        }
        sigalarm_handler_ant = signal(SIGALRM, sigalarm_handler);
        if (sigsetjmp(entorno,1) == 0) {
                /* activamos temporizador */
                alarma_ant = alarm(seg);
                /* esperamos una señal */
                pause();
        } else {
                /* restauramos cualquier alarma anterior */
                signal(SIGALRM, sigalarm_handler_ant);
                alarm(alarma_ant);
        }
}

int main (int argc, char *argv[]) {
        int i ;

        for (i = 0; i < 10; i++) {
                fprintf(stdout, "Durmiendo %d segundos.\n", i);
                dormir(i);
        }
        return 0;
}
```

La llamada `alarm` maneja la resolución de segundos, pero podemos necesitar temporizadores que controlen tiempos más pequeños. Para dar respuesta a estas necesidades, el UNIX de Berkeley define dos llamadas (una para declarar temporizadores y otra para leer su estado) que manejan resoluciones de tiempo mayores. Estos temporizadores responden al siguiente esquema de funcionamiento:

1. Se debe definir el temporizador. Dentro de la definición, entra declarar cuál va a ser su valor inicial (en segundos y microsegundos) y cuál será el valor con el que se recargue automáticamente cuando llegue a 0.
2. Cuando el temporizador llega a cero, el proceso que lo declaró recibe algunas de las señales `SIGALRM`, `SIGVTALARM` o `SIGPROF`.
3. El contador se reinicializa con un valor predefinido y continúa con su cuenta regresiva, repitiéndose así todo el proceso.

Las funciones de 4.3BSD son `getitimer` y `setitimer`, y tienen las siguientes declaraciones:

```
#include <time.h>
getitimer(int wich, struct itimerval *value);
```

```
setitimer(int wich, struct itimerval *value, struct itimerval
*ovalue);
```

El sistema permite que cada proceso tenga tres temporizadores, que se definen en <sys/time.h>. La llamada getitimer devuelve el valor actual del temporizador especificado en wich. setitimer declarar el nuevo valor de un temporizador, el especificado en wich, y opcionalmente devuelve el valor anterior. El valor del temporizador se especifica mediante una estructura del tipo itimerval, que se declara de la siguiente forma:

```
struct itimerval {
      struct timeval it_interval; /* interval del temporizador */
      struct timeval it_value; /* valor actual del temporizador */
};
```

Si it_value no vale cero, indica el tiempo que falta para que el temporizador llegue a cero. Si it_interval no vale cero, indica el valor con que se recargará el temporizador cuando llegue a cero. Poniendo it_value a cero, desactivamos el temporizador. Poniendo it_interval a cero, hacemos que el temporizador quede desactivado cuando llegue a cero.

wich especifica qué temporizadores vamos a usar. Sus posibles valores son:

Valores	Significado
ITIMER_REAL	El temporizador se decrementa en tiempo real. Cuando llega a cero, el proceso recibe la señal de SIGALRM. Este temporizador también es modificado por alarm, por lo que el uso de esta llamada al sistema después de setitimer va a cambiar el valor del temporizador.
ITIMER_VIRTUAL	El temporizador se decrementa sólo cando el proceso se está ejecutando en modo usuario (tiempo virtual). Mientras el proceso duerme, el temporizador está parado. Cuando el temporizador llega a cero, el proceso recibe la señal de SIGVTALRM.
ITIMER_PROF	El temporizador se decrementa cuando el proceso se está ejecutando en modo usuario y cuando está en modo supervisor. Cuando el temporizador llega a cero, el proceso recibe la señal SIGPROF. Puesto que esta señal puede llegar mientras se está ejecutando una llamada al sistema (recordemos que las llamadas al sistema se ejecutan en modo supervisor), los procesos que la reciban deben estar programas para reiniciar la ejecución de las llamadas interrumpidas.

La resolución con la que trabajan estos temporizadores depende de nuestra computadora. La constante HZ, definida en <sys/param.h>, indica la frecuencia, en hercios, más alta que puede manejar un temporizador. Así, su resolución es de 1/HZ segundos. Los valores máximos de intervalo que pueden manejar los tres temporizadores son MAX_ALARM, MAX_VTALARM y MAX_PROF, respectivamente. En todas las implementaciones se garantiza un mínimo de 31 días para estos valores.

Tanto `getitimer` como `setitimer` devuelven el valor de 0 si se ejecutan satisfactoriamente y -1 en caso contrario.

La siguiente secuencia de código declara un temporizador en tiempo real que va a tardar 10.5 segundos en terminar la primera vez, y después va a expirar periódicamente cada 1.54 segundos.

```
struct itimerval temporizador, temporizador_ant;
…
temporizador.it_value.tv_sec = 10;
temporizador.it_value.tv_usec = 500000;
temporizador.it_interval.tv_sec = 1;
temporizador.it_interval.tv_usec = 540000;
setitimer(ITIMER_REAL, &temporizador, &temporizador_ant);
```

Como ejemplo de aplicación, vamos a codificar una versión reducida de la orden `at(1)` que permite lanzar trabajos en segundo plano que se deben ejecutar a una hora terminada. Nuestro programa se va a llamar `a_las`, y la forma de invocar será:

```
$ a_las hh:mm:ss línea_de_órdenes
```

Por ejemplo:

```
$ a_las 13:55:00 echo HORA DE COMER!!!!
```

Esta línea va a hacer que a las 13:55:00 horas aparezca en pantalla el mensaje.

El programa `a_las`, tras crear un proceso hijo, le devuelve el control al sistema. El proceso hijo es el encargado de activar un temporizador en tiempo real que llegará a cero cuando sea la hora deseada.

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

#define SEGUNDOS_POR_DIA 86400

#define hhmmss_a_ss(_hh, _mm, _ss) ((long) (_hh)*3600 + (_mm)*60 + (_ss))

#define ss_a_hhmmss(_seg, _hh, _mm, _ss) \
{\
       _hh = (int)((_seg)/3600);\
       _mm = (int)(((_seg) - (long)_hh*3600) / 60);\
       _ss = (int)(((_seg) - (long)_hh*3600) - _mm*60);\
}

void sigalarm_handler(int sig) {}

int main(int argc, char *argv[]) {
       int hh, mm, ss, pid;

       if (argc < 3) {
```

```
            fprintf(stderr, "forma de uso: %s hh:mm:ss línea_de_órdenes\n",
argv[0]);
            return -1;
      }

      if (sscanf(argv[1], "%d:%d:%d", &hh, &mm, &ss) != 3) {
            fprintf(stderr, "formato de hora erróneo: %s\n", argv[1]);
            return -1;
      } else if (hh < 0 || hh > 23 || mm < 0 || mm > 59 || ss < 0 || ss > 59) {
            fprintf(stderr, "hora fuera de rango: %s\n", argv[1]);
            return -1;
      }

      if ((pid = fork()) < 0) {
            perror(argv[0]);
            return -1;
      } else if (pid == 0) {
            long total_segundos;
            time_t t;
            struct tm *tmm;
            struct itimerval temporizador, temporizador_ant;

            temporizador.it_value.tv_usec = 0;
            temporizador.it_interval.tv_sec = 0;
            temporizador.it_interval.tv_usec = 0;
            signal(SIGALRM, sigalarm_handler);

            /* lectura de la fecha */
            time(&t);
            tmm = (struct tm*) localtime(&t);

            /* cálculo del valor del temporizador */
            total_segundos = hhmmss_a_ss(hh, mm, ss) - hhmmss_a_ss(tmm->tm_hour,
tmm->tm_min, tmm->tm_sec);
            if (total_segundos < 0) {
                  total_segundos += SEGUNDOS_POR_DIA;
            }
            temporizador.it_value.tv_sec = total_segundos;
            setitimer(ITIMER_REAL, &temporizador, &temporizador_ant);
            pause();
            execvp(argv[2], &argv[2]);
      } else {
            return 0;
      }
}
```