5️⃣ Llamadas al sistema para gestión y control de señales

Concepto de señal

Una **señal** es una notificación a un proceso de que ha ocurrido un evento, a veces son descritas como *interrupciones software*.

El kernel es el responsable de enviar señales, aunque un proceso puede, si tiene permisos suficientes, enviar una señal a otro proceso. En este caso, podemos usar las señales como una técnica de sincronización, una forma primitiva de IPC (*interprocess communication*). Sin embargo, reiteramos que el kernel es el que envía las señales en mayor parte. Los tipos de eventos que pueden hacer que el kernel genere una señal para un proceso son, entre otros:

- La ocurrencia de una excepción hardware, es decir, el hardware detectó una condición de fallo que fue notificada al kernel, que envió la correspondiente señal al proceso en cuestión. Ejemplos de esto son instrucciones mal formadas en lenguaje máquina, dividor por 0, o referirse a una parte de memoria inaccesible.
- El usuario tecleó uno de los caracteres especiales que generan señales. Estos incluyen el caracter de *interrupción* (normalmente *Ctrl-C*), y el caracter de *suspensión* (normalmente *Ctrl-Z*).
- Un evento de software ocurrió. Por ejemplo, la entrada es finalmente disponible en un descriptor de archivo, una ventana de terminal fue cambiada de tamaño, el límite de tiempo de CPU fue excedido, o un hijo del proceso terminó.

Cada señal se identifica por un entero único, comenzando secuencialmente por 1. Sin embargo, es aconsejable referenciarlos mediante nombres simbólicos en la forma de `SIGxxxx`, ya que los enteros pueden variar dependiendo de la arquitectura. Estas están definidas en `<signal.h>`, y podemos verlas desde `man 7 signal`.

Una señal se dice que es *generada* por algún evento. Una vez generada, una señal puede ser *enviada* a otro proceso, que realiza alguna acción en respuesta. Entre el tiempo en el que ha sido generado y el tiempo que ha sido enviado, una señal se dice que está *pendiente*.

Normalmente, una señal pendiente se envía a un proceso tan pronto como esté planificado para ejecutarse, o inmediatamente si el proceso ya se está ejecutando (por ejemplo, un proceso se manda una señal a sí mismo). A veces, de hecho, necesitamos asegurar que un segmento de código no ha sido interrumpido por el envío de una señal. Para hacer esto, podemos añadir una señal a la *máscara de señal*, un conjunto de señales cuyo envío está *bloqueado*. Si una señal es generada mientras que está bloqueada, permanece pendiente hasta que es desbloqueada (es

eliminada de la máscara). Existen varias llamadas al sistema para añadir y eliminar señales de la máscara de señal.

Al mandar una señal, el proceso realiza alguna de las sigueintes acciones por defecto:

- La señal es *ignorada*, es decir, es descartada por el kernel y no tiene efecto en el proceso. De hecho, el proceso ni si quiera sabe que ocurrió.
- El proceso es *terminado* (matado). Esto se denomina a veces *terminación anormal del proceso*, en lugar de la terminación normal que ocurre al hacer `exit()`.
- Un archivo de *volcado de memoria* es generado, y el proceso es terminado. Un archivo de volcado de memoria contiene una imagen de la memoria virtual de un proceso, que puede ser cargado en un depurador para inspeccionar el estado del proceso en el momento en el que finalizó.
- El proceso es *parado*, es decir, la ejecución del proceso es suspendida.
- La ejecución del proceso *continúa* tras haber sido previamente parado.

En lugar de aceptar la acción predeterminada para una señal particular, un programa puede cambiar la acción que ocurre cuando una señal es enviada. Esto se denomina *disposición* de la señal. Un programa puede configurar una de las siguientes disposiciones:

- Que ocurra la *acción predeterminada*.
- Que la señal sea *ignorada*.
- Un *manejador de señal* es ejecutado. Es una función, escrita por el programador, que ejecuta ciertas acciones en respuesta al envío de una señal.

Es importante notar que no se puede configurar la disposición de una señal para *eliminar* o *volcar memoria* (a menos que sea la disposición predeterminada). Lo más cercano que podemos hacer es llamar a `exit()` o a `abort()`. Este último genera la señal `SIGABRT` para el proceso, que causa un volcado de memoria y la terminación.

Glosario práctico

Envío de señales: `kill()`

> Envío de señales entre procesos, es el análogo al comando de terminal `kill`.

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

- **Valor de retorno:** 0 en éxito, -1 en error.
- `pid` modifica el comportamiento de `kill()`:
  - `pid > 0`: la señal se envía al proceso con el PID especificado por `pid`.
  - `pid == 0`: la señal se envía a todos los procesos en el mismo grupo de procesos que el proceso invocador, incluyendo el proceso en sí mismo.
  - `pid < -1`: la señal se envía a todos los procesos en el grupo de procesos cuyo ID (GID) es igual al valor absoluto de `pid` (es decir, `-pid`).

    - `pid == -1`: la señal se envía a todos los procesos para los que el proceso invocador tenga permiso para enviar una señal excepto a *init* (PID 1) y el proceso invocador. Si un proceso privilegiado hace esta llamada, todos los procesos del sistema recibirán la señal, excepto por los dos últimos.
    - **Posibles errores:** si ningún proceso coincide con el PID especificado, `kill()` falla y modifica `errno` a `ESRCH` ("no such process").
- `sig` es la señal a enviar. Caso especial: **comprobación de la existencia de procesos**.
    - `sig == 0`: no se envía ninguna señal, por lo que `kill()` simplemente hace la comprobación de error, para ver si el ID especificado existe.

Cambio de acción ante señal: `sigaction()`

> Cambia la acción de un proceso al recibir una determinada señal

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, struct sigaction *oldact);
```

- **Valor de retorno:** 0 en éxito, -1 en error.
- `sig`: señal cuya disposición queremos cambiar. No puede ser ni `SIGKILL` ni `SIGSTOP`.
- `act`: puntero a estructura especificando nueva dispsición para señal. Si queremos únicamente conocer la dispsición existente, la hacemos `NULL`.
- `oldact`: aquí se guarda el puntero a la estructura `sigaction` existente, si `act` es `NULL`.

```
struct sigaction {
    void (*sa_handler)(int);  /* Dirección de manejador */
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;     /* Señales bloqueadas durante la invocación del
manejador */
    int sa_flags;       /* Flags controlando la invación del manejador */
    void (*sa_restorer)(void);  /* No para el uso en aplicaciones */
}
```

- `sa_handler`: dirección a función manejadora, o las constantes `SIG_IGN` o `SIG_DFL`. Los argumentos siguientes se interpretan si `sa_handler` no es ninguna de estas constantes.
- `sa_mask`: conjunto de señales que se bloquearán al invocar el manejador definido por `sa_handler`. Nos permite especificar un conjunto de señales que no permiten interrumpir la ejecución del manejador.
    - Nota sobre comportamiento: cuando se invoca el manejador, si aparecen señales en el `sigset_t` que no sean parte de la máscara de señales, serán añadidas antes de invocar al manejador, y eliminadas una vez que actúe.
- `sa_flags`: conjunto de flags, hechas OR (`|`).

Flag	Descripción
`SA_NOCLDSTOP`	Si `sig` es `SIGCHLD`, indica al núcleo que el proceso no desea recibir notificación cuando los procesos hijos se paren (cuando reciban `SIGSTP`, `SIGTTIN` o `SIGTTOU`)
`SA_NOCLDWAIT`	Si `sig` es `SIGCHLD`, no transformar los hijos en zombies si terminan.
`SA_NODEFER` o `SA_NOMASK`	Cuando se capta la señal, no añadirla de forma automática a la máscara de señales del proceso mientras se ejecute el manejador. Es decir, se pide al núcleo que no impida la recepción de la señal desde el propio manejador de la señal.
`SA_ONSTACK`	Invocar el manejador utilizando un *stack* alternativo instalado por `sigaltstack()`.
`SA_RESETHAND`	Cuando se capta esta señal, se resetea su disposición a la predeterminada antes de invocar el manejador.
`SA_RESTART`	Automáticamente reinicia las llamdas al sistema interrumpidas por este manejador de señal. Proporciona un comportamiento compatible con la semántica de señales de BSD haciendo que ciertas llamadas al sistema reinicien su ejecución cuando son interrumpidas por la recepción de una señal.
`SA_SIGINFO`	Invoca el manejador de señal con tres argumentos en lugar de uno. En este caso, se debe configurar `sa_sigaction` en lugar de `sa_handler`.

Asignación de valores a `sa_mask`

Se utilizan las funciones que especificamos a continuación.

```
#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);
```

- **Valores de retorno:** ambos 0 en éxito, -1 en error.
- `sigemptyset()` inicializa el set de señales para no contener miembros.
- `sigfillset()` inicializa el set de señales para contenerlas todas.
- `sigaddset()` añade la señal `sig` al conjunto de señales.

- `sigdelset()` elimina la señal `sig` del conjunto de señales.

```
#include <signal.h>

int sigismember(const sigset_t *set, int sig);
```

- **Valor de retorno:** 1 si `sig` es miembro de `set`, 0 en otro caso.

Cambio de máscara de señales: `sigprocmask()`

> Examina y cambia la máscara de señales.

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

- **Valor de retorno:** 0 en éxito, -1 en error.
- `how` especifica los cambios que se realizan a la máscara:
  - `SIG_BLOCK`: las señales especificadas en el set de señales al que apunta `set` son añadidas a la máscara de señal. En otras palabras, la máscara de señal cambia a la unión de su valor actual y `set`.
  - `SIG_UNBLOCK`: las señales en el set de señales al que apunta `set` son eliminadas de la máscara de señal. Desbloquear una señal que no es bloqueada no causa error.
  - `SIG_SETMASK`: el set de señales al que apunta `set` es asignado a la máscara de señal.
- `oldset`: si no es `NULL`, apunta a un búfer `sigset_t` que es usado para devolver la máscara de señal anterior. Si queremos obtener la máscara de señal, nos basta establecer a `NULL` el argumento `set`, en cuyo caso se ignorará a `how`.

Señales pendientes: `sigpending()`

> Permite examinar el conjunto de señales bloqueadas y/o pendientes de entrega.

Cuando un proceso recibe una señal que está bloqueada, es añadida al set de señales pendientes. Cuando (y si) la señal es desbloquada más tarde, es enviada al proceso. Para determinar estas señales, usamos `sigpending()`.

```
#include <signal.h>

int sigpending(sigset_t *set);
```

- **Valor de retorno:** 0 en éxito, -1 en error.

- `set`: aquí es donde se devuelven el set de señales pendientes para el proceso invocador. Podemos examinarlo con las funciones anteriores.

Esperar a una señal usando una máscara: `sigsuspend()`

> Reemplaza temporalmente la máscara de señal para el proceso con la dada por el argumento `mask` y luego suspende el proceso hasta que se recibe una señal.

Antes de entender qué hace, describamos una situación donde necesitemos usarla. Consideremos el siguiente caso, encontrado frecuentemente al programar con señales:

1. Bloqueamos temporalmente una señal para que el manejador de la señal no interrumpa la ejecución de una sección de código crítico.
2. Desbloqueamos la señal, y luego suspendemos la ejecución hasta que la señal es enviada.

La forma de desbloquear una señal *de forma atómica* y suspender el proceso es mediante `sigsuspend()`.

```
#include <signal.h>

int sigsuspend(const sigset_t *mask);
```

- **Valor de retorno:** (normalmente) devuelve -1 con `errno` en `EINTR`, si es interrumpida por una señal capturada (no está definida la terminación correcta).
- **Comportamiento:** reemplaza la máscara de señal por el set de señales al que apunta `mask`, y luego suspende la ejecución del proceso hasta que se capta una señal y su manejador devuelve. Una vez que su manejador devuelve (*return*), `sigsuspend()` restablece la máscara de señal al valr que tenía antes de la llamada.

  Es equivalente a hacer las siguientes operaciones:

  ```
  sigprocmask(SIG_SETMASK, &mask, &prevMask);  // asignar nueva máscara
  pause();
  sigprocmask(SIG_SETMASK, &prevMask, NULL);  // restaurar máscara anterior
  ```