3️⃣ Llamadas al sistema para el control de procesos

Identificador de un proceso

Cada proceso tiene un ID de proceso (PID), un entero positivo que identifica de forma única a cada proceso del sistema. Los PIDs son usados en una gran variedad de llamadas al sistema.

El tipo de dato `pid_t` es un tipo entero especificado por SUSv3 para el propósito de almacenar PIDs.

A excepción del proceso *init*, con PID 0, no hay relación entre el programa y el PID del proceso que es creado para ejecutar dicho programa.

Creación y terminación de procesos

Las llamadas al sistema involucradas en la creación y terminación de procesos son:

- **`fork()`**: permite a un proceso, el *proceso padre*, crear un nuevo proceso, el *proceso hijo*. Esto se hace creando para el proceso hijo una (casi) exacta duplicación del padre: el hijo obtiene copias de los datos, el *stack*, el *heap* y los segmentos de texto del padre.
- **`exit(status)`**: termina un proceso, haciendo que todos los recursos que utiliza (memoria, descriptores abiertos, etc.) puedan ser reutilizados por el kernel. `status` es un entero que determina el estado de terminación del proceso. Usando la llamada `wait()`, el padre puede captar su estado.
- **`wait(&status)`**: tiene dos propósitos. Primero, si un hijo de este proceso todavía no se ha terminado usando `exit()`, `wait()` suspende la ejecución del proceso hasta que uno de sus hijos haya terminado. Segundo, el estado de terminación del hijo es devuelto en el argumento de estado de `wait()`.
- **`execve(pathname, argv, envp)`**: carga un nuevo programa (`pathname`, con lista de argumentos `argv` y lista de entorno `envp`) en la memoria de un proceso. El texto del programa existente es descartado, y el *stack*, los datos y los segmentos del *heap* son creados nuevos para el nuevo programa. Más tarde, veremos que hay más funciones de librería diseñadas sobre `execve()`, cada una proporcionando una variación útil en la interfaz de programación.

Glosario práctico

Llamadas útiles para procesos e identificadores de procesos

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);   // devuelve el PID del proceso que la invoca
pid_t getppid(void);  // devuelve el PID del padre del proceso que la invoca
pid_t getuid(void);   // devuelve el UID real del proceso que la invoca
pid_t geteuid(void);  // devuelve el UID efectivo del proceso que la invoca
pid_t getgid(void);   // devuelve el GID real del proceso que la invoca
pid_t getegid(void);  // devuelve el GID efectivo del proceso que la invoca
```

- **Valor de retorno:** el comentado, siempre satisfactorio.

Creación de un proceso: `fork()`

> Crea un nuevo proceso, el *hijo*, que es un duplicado casi exacto del proceso que lo llama, el *padre*.

```
#include <unistd.h>

pid_t fork(void);
```

- **Valor de retorno:** retorna el ID del proceso hijo si satisfactorio, o -1 en error; en el hijo creado satisfactoriamente siempre retorna 0.
- **Comportamiento:** tras hacer `fork()` existen dos nuevos programas, y *la ejecución continúa desde el punto en el que `fork()` devuelve*. Aunque se duplique el proceso, se crean dos copias, pudiendo modificar cada uno de ellos las variables del *stack*, el *heap*, los dats y los segmentos de texto sin afectar al otro proceso.

Algunos usos útiles de `fork()`

- **Distinguir entre el padre y el hijo:** nos basta guardar el valor de retorno de `fork()` en un tipo `pid_t`, entonces:
  - Si `pid_t == 0`, estamos en el proceso hijo.
  - Si `pid_t > 0`, estamos en el proceso padre, y en la variable `pid_t` tenemos el PID del hijo.
  - Si `pid_t < 0` ha habido algún error haciendo `fork()`.

- **Distinguir si se hacen múltiples `fork()`:** basta hacer un array de `pid_t`.

Terminación de un proceso: `exit()`

> Un proceso puede terminar de forma *anormal*, si ocurre alguna señal cuya acción haga que el proceso termine (con o sin volcado de memoria), o de forma *normal*, llamando a esta llamada al sistema.

```
#include <stdlib.h>

void exit(int status);
```

Realiza la siguientes acciones:

- Se llama a los manejadores de terminación (registrados con `atexit()` y `on_exit()`), en orden inverso a su registro.
- Se arrasa con los *stream buffers* de `stdio`.
- Se invoca a la llamada al sistema `_exit()`, usando el valor especificado en `status`.
    - Por convención, un valor 0 indica que el proceso terminó de forma satisfactoria, y un valor no-cero indica que terminó insatisfactoriamente. SUSv3 especifica dos constantes: `EXIT_SUCCESS` (0) y `EXIT_FAILURE` (1).

Esperando a un hijo: `wait()` y `waitpid()`

En muchas aplicaciones en el que el padre crea un proceso hijo, es útil para el padre monitorizar a los hijos para saber cuándo y cómo terminan. Esto es posible gracias a `wait()` y otras llamadas al sistema relacionadas.

La llamada al sistema `wait()`

> Espera a que termine uno de los hijos del proceso que la invoca, y devuelve el estado de terminación del hijo al búfer al que aupunta `status`.

```
#include <sys/wait.h>

pid_t wait(int *status);
```

- **Valor de retorno:** PID del hijo terminado, -1 en error.

- **Comportamiento:** hace lo siguiente:
    1. Si ningún hijo del proceso invocador ha terminado, la llamada se bloquea hasta que uno de sus hijos termina. Si un hijo ha terminado ya, `wait()` devuelve inmediatamente.
    2. Si `status` no es `NULL`, la información del hijo terminado se devuelve en el lugar a donde apunta `status` (ver más adelante "el valor de estado de `wait`").
    3. El kernel añade los tiempos de computación de CPU y las estadísticas de uso de recursos a los totales de todos los hijos del proceso padre.
    4. Como resultado, `wait()` devuelve el PID del hijo que ha terminado.

La llamada al sistema `waitpid()`

Esta llamada resuelve los siguientes problemas de `wait()`:

- Si el proceso padre ha creado múltiples procesos hijos, no se puede hacer `wait()` para la completación de un hijo específico, únicamente podemos esperar al siguiente hijo que termine.
- Si ningún hijo ha terminado, `wait()` siempre se bloquea.
- Usando `wait()`, sólo podemos conocer qué hijos han terminado. No es posible ser notificados de cuándo un hijo es parado por una señal (como `SIGSTOP` o `SIGTTIN`) o cuándo un hijo parado ha continuado mediante una señal `SIGCONT`.

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- **Valor de retorno:** el PID del hijo, 0 (seguir leyendo) o -1 en error.
- **Comportamiento:** el valor retornado y los argumentos `status` son los mismos que en `wait()`. Además, existen parámetros adicionales:
- `pid` permite la selección del hijo a esperar:
    - `pid > 0`: esperar al hijo cuyo PID sea igual a `pid`.
    - `pid == 0`: esperar a cualquier hijo en el mismo grupo de procesos que el padre.
    - `pid < -1`: esperar a cualquier hijo cuyo grupo de procesos sea igual a `-pid`.
    - `pid == -1`: esperar a cualquier hijo. La llamada `wait(&status)` equivale a la llamada `waitpid(-1, &status, 0)`.
    - `options` es una máscara de bits que puede incluir una o más (haciendo OR (`|`)) de las siguientes flags (SUSv3):

Flag	Descripción
`WUNTRACED`	Además de devolver información sobre hijos terminados, devolver información sobre cuando un hijo sea *parado* por una señal.
`WCONTINUED` (Linux 2.6.10+)	También devolver información de estado sobre hijos parados que hayan sido reanudados por la entrega de una señal `SIGCONT`.
`WNOHANG`	Si ningún hijo especificado por `pid` ha cambiado de estado, devolver inmediatamente en lugar de bloquear. En este caso, el valor de retorno es 0. Si el proceso que llama no tiene hijos que coincidan con la especificación de `pid`, `waitpid()` falla con el error `ECHILD`.

El valor de estado de `wait`

El valor `status` retornado por `wait()` y por `waitpid()` nos permite distinguir los siguientes eventos para el hijo:

- El hijo terminó llamando a `exit()`.
- El hijo fue terminado por la entrega de una señal no manejada.
- El hijo fue parado por una señal, y `waitpid()` fue llamado con la flag `WUNTRACED`.
- El hijo fue restaurado por una señal `SIGCONT`, y `waitpid()` fue llamado con la flag `WCONTINUED`.

El archivo de cabecera `<sys/wait.h>` define un set de macros estándar para interpretar el valor de `status`. Al aplicarlos, sólo uno de ellos devolverá *verdadero*.

Macro	Descripción
`WIFEXITED(status)`	Devuelve verdadero si el proceso hijo hizo exit de forma normal. En este caso, la macro `WEXITSTATUS(status)` devuelve el estado de salida del proceso hijo.
`WIFSIGNALED(status)`	Devuelve verdadero si el proceso hijo fue matado por una señal. En este caso, la macro `WTERMSIG(status)` devuelve el número de la señal que causó la finalización, y la macro `WCOREDUMP()` devuelve verdadero si el proceso produjo un volcado de memoria.
`WIFSTOPPED(status)`	Devuelve verdadero si el proceso hijo fue parado por una señal. En este caso, la macro `WSTOPSIG(status)` devuelve el número de la señal que paró el proceso.
`WIFCONTINUED(status)`	Devuelve verdadero si el hijo reanudó su ejecución mediante la entrega de `SIGCONT`.

Las funciones de librería `exec()`

La función principal: `execve()`

> Carga un nuevo programa en la memoria del proceso.

```
#include <unistd.h>

int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- **Valor de retorno:** nunca devuelve en éxito, -1 en error.
- `pathname`: ruta del nuevo programa. Puede ser absoluta o relativa.
- `argv` especifica los argumentos de línea de comandos que se pasará al nuevo programa.
- `envp` especifica el *environment list* para el nuevo programa.

Funciones derivadas

```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char*) NULL, char *const envp[] */);
int execlp(const char *filename, const char *arg, ...
                /* , (char*) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
               /* , (char*) NULL */);
```

- **Valor de retorno:** ninguno devuelve en éxito, todos devuelven -1 en error.

[Completar]