**¿Qué es un fork()?**

fork() es una función en sistemas Unix/Linux que permite crear un nuevo proceso duplicando el proceso que la invoca. El proceso original se llama "proceso padre", y el proceso recién creado es el "proceso hijo". Ambos procesos continúan ejecutándose a partir del punto donde se invocó fork(). La única diferencia es que fork() devuelve el pid del proceso hijo al padre y 0 al hijo. Si ocurre un error, fork() devuelve un valor negativo.

**Ejecución paso a paso del código**

c

Copiar código

#include <sys/types.h>

#include <sys/wait.h>

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

int main()

{

pid\_t pid;

pid = fork ();

printf("Chau\n");

if (pid < 0) {

fprintf(stderr, "Fork ha fallado");

return 1;

}

else if (pid == 0) { /\* proceso hijo \*/

printf("Soy el proceso Hijo\n");

execlp("/bin/ls","ls",NULL);

}

else { /\* proceso padre \*/

printf("Esperando al proceso Hijo\n");

wait(NULL);

printf("El proceso Hijo ha terminado\n");

}

}

**Justificación de la salida**

1. **Inicio del programa**: Se ejecuta el programa y llega a la llamada a fork().
2. **Creación del proceso hijo**: fork() crea un proceso hijo. En este punto, hay dos procesos ejecutando el mismo código: el padre y el hijo.
3. **Primera impresión de "Chau"**:
   * Ambos procesos (padre e hijo) continúan la ejecución inmediatamente después de fork().
   * Tanto el proceso padre como el hijo ejecutan la línea printf("Chau\n");. Esto significa que "Chau" se imprime dos veces, una vez por el proceso padre y otra por el proceso hijo.
4. **Verificación del valor de pid**:
   * Si pid < 0, significa que fork() falló. En este caso, se imprime un mensaje de error.
   * Si pid == 0, estamos en el proceso hijo. Se imprime "Soy el proceso Hijo", y luego el proceso hijo reemplaza su espacio de memoria con el comando /bin/ls usando execlp(), mostrando el listado de archivos del directorio actual.
   * Si pid > 0, estamos en el proceso padre. El padre imprime "Esperando al proceso Hijo", espera a que el proceso hijo termine usando wait(NULL), y finalmente imprime "El proceso Hijo ha terminado".

**Resumen**

La línea "Chau" se imprime dos veces porque tanto el proceso padre como el proceso hijo ejecutan el printf inmediatamente después del fork().

Vamos a desglosar lo que hace la instrucción pid\_t pid; pid = fork();:

**1. Declaración de pid\_t pid;**

* pid\_t es un tipo de dato definido en la biblioteca de unistd.h, y se utiliza para almacenar identificadores de procesos (PID).
* pid es la variable que almacenará el valor devuelto por la función fork().

**2. Llamada a fork()**

* fork() es una función que se utiliza para crear un nuevo proceso.
* Cuando fork() es llamado, el sistema operativo crea una copia casi exacta del proceso que la llamó.
* El proceso original (padre) y el proceso recién creado (hijo) continúan ejecutándose desde la instrucción inmediatamente después de fork().

**3. Valor devuelto por fork()**

* **En el proceso padre**: fork() devuelve el PID del proceso hijo. Esto significa que el proceso padre ahora tiene una referencia a su hijo y puede usar este PID para interactuar con él (por ejemplo, para esperar a que termine).
* **En el proceso hijo**: fork() devuelve 0, lo que permite al proceso hijo saber que es el hijo.
* **Si fork() falla**: devuelve un valor negativo (generalmente -1), indicando que no se pudo crear el proceso hijo.

**Ejemplo**

c

Copiar código

pid\_t pid;

pid = fork();

* Si pid es 0, el código que sigue se ejecuta en el proceso hijo.
* Si pid es mayor que 0, el código que sigue se ejecuta en el proceso padre.
* Si pid es negativo, significa que la creación del proceso falló.

**Ejemplo en Acción**

Supongamos que fork() se ejecuta correctamente:

* **Proceso Padre**: pid será el PID del proceso hijo (por ejemplo, 1234).
* **Proceso Hijo**: pid será 0.

Esto permite al programa diferenciar qué código debe ejecutarse en el padre y cuál en el hijo.

**Ejercicio 1: Implementación del Shell (Paso 4 con fork())**

Para implementar un shell que pueda ejecutar comandos, se puede usar fork() para crear un nuevo proceso que ejecute el comando dado por el usuario. Aquí te dejo una estructura básica para el paso 4:

c

Copiar código

while(1) {

// 1. Espera comando del usuario (por ejemplo, "ls -l")

char comando[1024];

printf("shell> ");

fgets(comando, sizeof(comando), stdin);

comando[strcspn(comando, "\n")] = '\0'; // Elimina el salto de línea

// 2. Crea un proceso hijo

pid\_t pid = fork();

if (pid < 0) {

// Si fork falla

fprintf(stderr, "Error al crear el proceso\n");

continue;

}

else if (pid == 0) {

// Proceso hijo: ejecuta el comando

execlp(comando, comando, NULL);

fprintf(stderr, "Error al ejecutar el comando\n");

exit(1);

} else {

// Proceso padre: espera a que termine el hijo

wait(NULL);

}

}

**Explicación**:

* El shell entra en un bucle infinito.
* Lee el comando del usuario.
* Usa fork() para crear un nuevo proceso hijo.
* El proceso hijo ejecuta el comando usando execlp().
* El proceso padre espera a que el hijo termine usando wait(), lo que permite que el shell no acepte nuevos comandos hasta que el actual haya terminado.

**Ejercicio 2: Ejecutar un editor de textos de GUI desde la terminal**

Cuando ejecutas un editor de textos con GUI (como gedit o nano si está configurado con interfaz gráfica), el terminal no regresa inmediatamente al prompt porque el proceso padre (el shell) está esperando que el proceso hijo (el editor de textos) termine. Esto sucede porque el proceso hijo es el que ejecuta la aplicación de GUI, y el padre está bloqueado en wait().

**Ejercicio 3: Ejecutar en segundo plano usando &**

Para resolver el problema de la terminal que no vuelve al prompt, puedes ejecutar el comando en segundo plano agregando un & al final del comando. Esto se puede implementar con un código similar al siguiente:

c

Copiar código

if (strchr(comando, '&') != NULL) {

// Si el comando contiene '&', eliminarlo y ejecutar en segundo plano

comando[strlen(comando) - 1] = '\0'; // Eliminar '&'

pid\_t pid = fork();

if (pid == 0) {

// Proceso hijo: ejecuta el comando

execlp(comando, comando, NULL);

fprintf(stderr, "Error al ejecutar el comando\n");

exit(1);

}

// No llamamos a `wait()` para que el proceso padre (el shell) no espere.

} else {

// Ejecutar en primer plano (esperar a que termine el comando)

pid\_t pid = fork();

if (pid == 0) {

execlp(comando, comando, NULL);

fprintf(stderr, "Error al ejecutar el comando\n");

exit(1);

} else {

wait(NULL);

}

}

**Explicación**:

* Si el comando contiene un &, se elimina ese carácter y se ejecuta el comando en el proceso hijo sin que el proceso padre espere (wait()).
* El shell volverá inmediatamente al prompt mientras el proceso hijo sigue ejecutándose en segundo plano.