

## PRÁCTICA 3 DE SISTEMAS OPERATIVOS

### TEMA: CREACIÓN DE PROCESOS

**Nombre:** Fernando Eliceo Huilca Villagómez

**Carrera:** Software

**Grupo:** GR1SW

**Fecha:** 09/07/2024

### Índice de Contenidos:

1. OBJETIVOS .....	2
1.1. Familiarizar al estudiante con el uso de las funciones fork, system, exec. ....	2
1.2. Realizar varias actividades de creación de procesos padres e hijos. ....	2
2. INFORME.....	3
3.1 USO DE FORK .....	3
3.1.1 Ejecute el programa en segundo plano.....	4
3.1.2 Creación de procesos hijos.....	7
3.1.4 Creación de un proceso hijo sin espera. ....	8
3.1.5 Creación de un proceso hijo con espera. ....	8
3.1.7 Ejecución concurrente de procesos .....	11
3.2 USO DE EXEC .....	13
3.2.1 Creación de un proceso zombie.....	15
3.2.2 Uso de exec.....	17
3.2.3 Crear un proceso hijo y colocarlo dentro de un bucle infinito.....	18
3. CONCLUSIONES Y RECOMENDACIONES.....	19
4. BIBLIOGRAFÍA .....	19

## Índice de Figuras

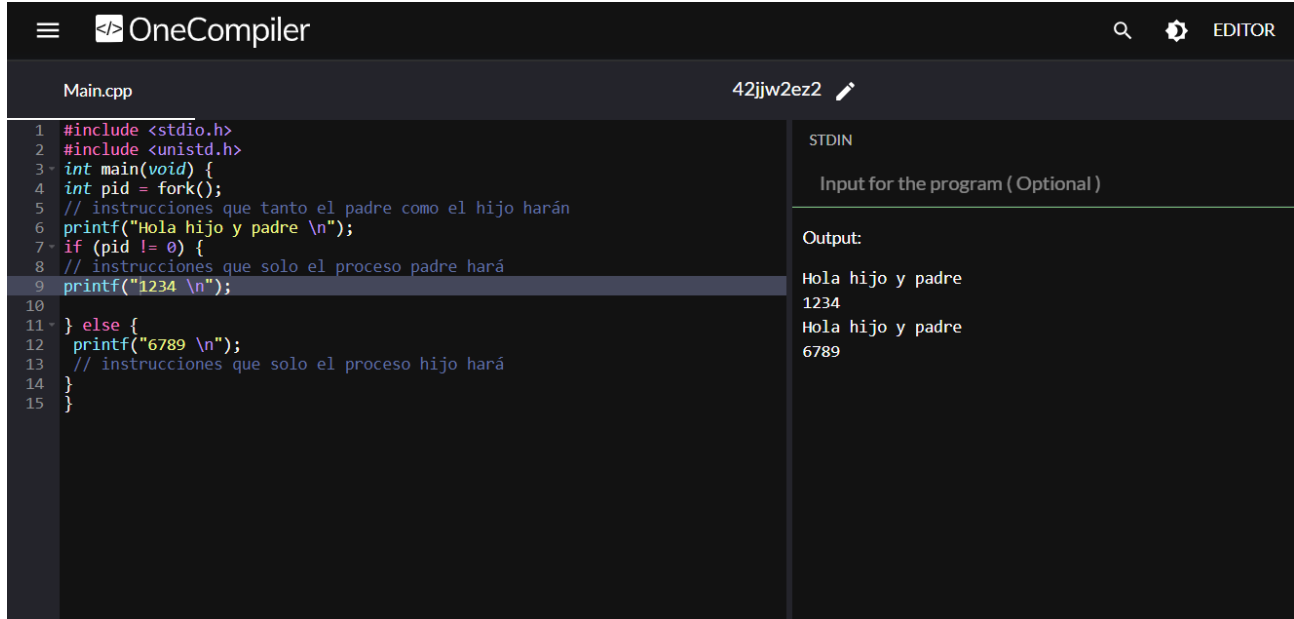
Ilustración 1 Uso de fork para familiarizarse con el concepto .....	3
Ilustración 2 Uso de getppid y pid en procesos.....	3
Ilustración 3 Ejecución de un proceso padre y su hijo .....	4
Ilustración 4 Código modificado para que imprima el mensaje de padre primero .....	5
Ilustración 5 Código que imprime sumatorio propio del padre e hijo .....	6
Ilustración 6 Imprimir 10 veces soy el hijo y soy el padre .....	7
Ilustración 7 Proceso hijo sin espera. ....	8
Ilustración 8 Proceso hijo con espera. ....	9
Ilustración 9 Mensaje de finalización 10 segundos más tarde que el proceso hijo.....	10
Ilustración 10 Proceso padre y de tres procesos hijos .....	10
Ilustración 11 Código que imprime número el de procesos hijos que se ejecutaron. ....	11
Ilustración 12 Ejecución concurrente de procesos.....	12
Ilustración 13 Código para que aparezca al final en pantalla el número de procesos hijos.....	13
Ilustración 14 Uso de exec .....	14
Ilustración 15 Se ha cambiado la función wait por sleep.....	15
Ilustración 16 Creación de un proceso zombie .....	15
Ilustración 17 Código que soluciona el problema planteado .....	16
Ilustración 18 Proceso hijo que liste los procesos del sistema usando execl .....	17
Ilustración 19 Creación de un proceso hijo colocado dentro de un bucle infinito.....	18

## 1. OBJETIVOS

**1.1. Familiarizar al estudiante con el uso de las funciones fork, system, exec.**

**1.2. Realizar varias actividades de creación de procesos padres e hijos.**

## 2. INFORME



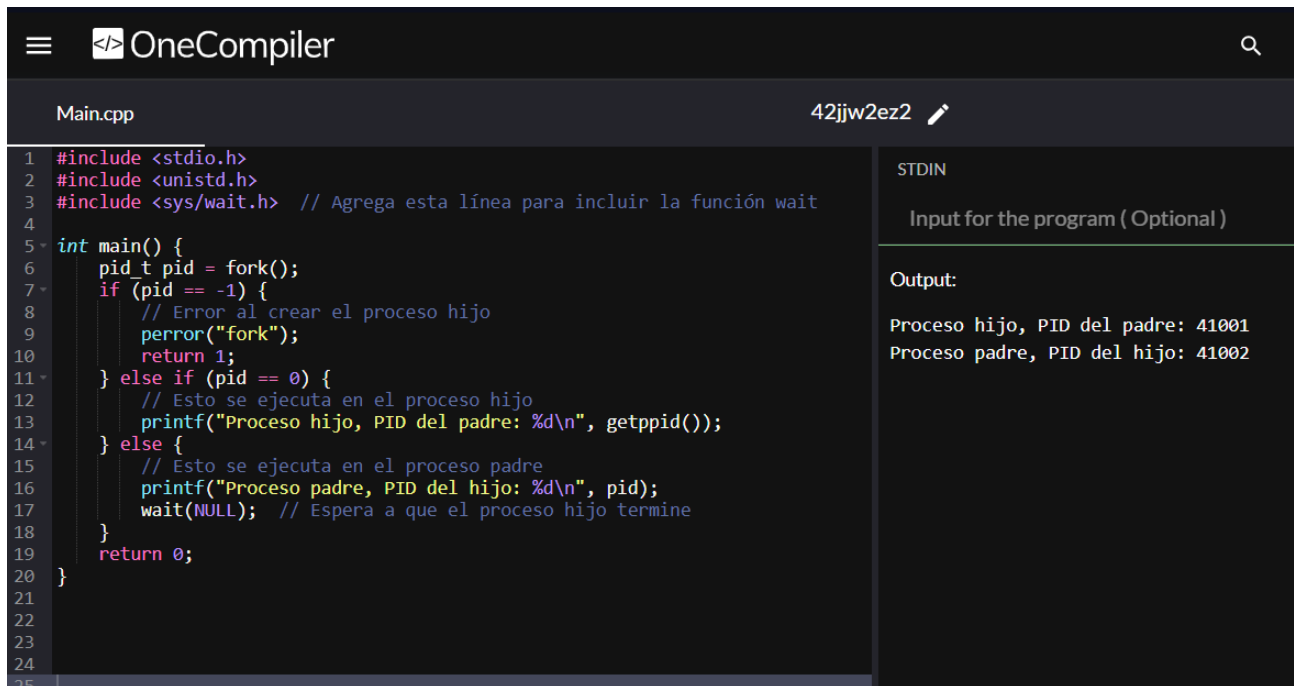
The screenshot shows the OneCompiler IDE interface. The editor displays a C++ program named Main.cpp. The program uses `fork()` to create a child process. Both parent and child processes execute `printf("Hola hijo y padre \n");`. The parent process also prints the PID of the child process. The output window shows the results of the program execution.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 int main(void) {
4     int pid = fork();
5     // instrucciones que tanto el padre como el hijo harán
6     printf("Hola hijo y padre \n");
7     if (pid != 0) {
8         // instrucciones que solo el proceso padre hará
9         printf("1234 \n");
10    }
11    else {
12        printf("6789 \n");
13        // instrucciones que solo el proceso hijo hará
14    }
15 }
```

Output:

```
Hola hijo y padre
1234
Hola hijo y padre
6789
```

Ilustración 1 Uso de fork para familiarizarse con el concepto



The screenshot shows the OneCompiler IDE interface. The editor displays a C++ program named Main.cpp. The program uses `fork()` to create a child process. The parent process prints the PID of the child process and uses `wait(NULL)` to wait for the child process to terminate. The child process prints the PID of the parent process. The output window shows the results of the program execution.

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h> // Agrega esta línea para incluir la función wait
4
5 int main() {
6     pid_t pid = fork();
7     if (pid == -1) {
8         // Error al crear el proceso hijo
9         perror("fork");
10        return 1;
11    } else if (pid == 0) {
12        // Esto se ejecuta en el proceso hijo
13        printf("Proceso hijo, PID del padre: %d\n", getpid());
14    } else {
15        // Esto se ejecuta en el proceso padre
16        printf("Proceso padre, PID del hijo: %d\n", pid);
17        wait(NULL); // Espera a que el proceso hijo termine
18    }
19    return 0;
20 }
```

Output:

```
Proceso hijo, PID del padre: 41001
Proceso padre, PID del hijo: 41002
```

Ilustración 2 Uso de getpid y pid en procesos

## 3. PROCEDIMIENTO

### 3.1 USO DE FORK

Ejecutar cada uno de los códigos propuestos y contestar las preguntas:

### 3.1.1 Ejecute el programa en segundo plano

#### Identifique los valores PID y PPID de cada proceso.

El PID (Process ID) es un identificador único para cada proceso en ejecución. El PPID (Parent Process ID) es el identificador del proceso padre.

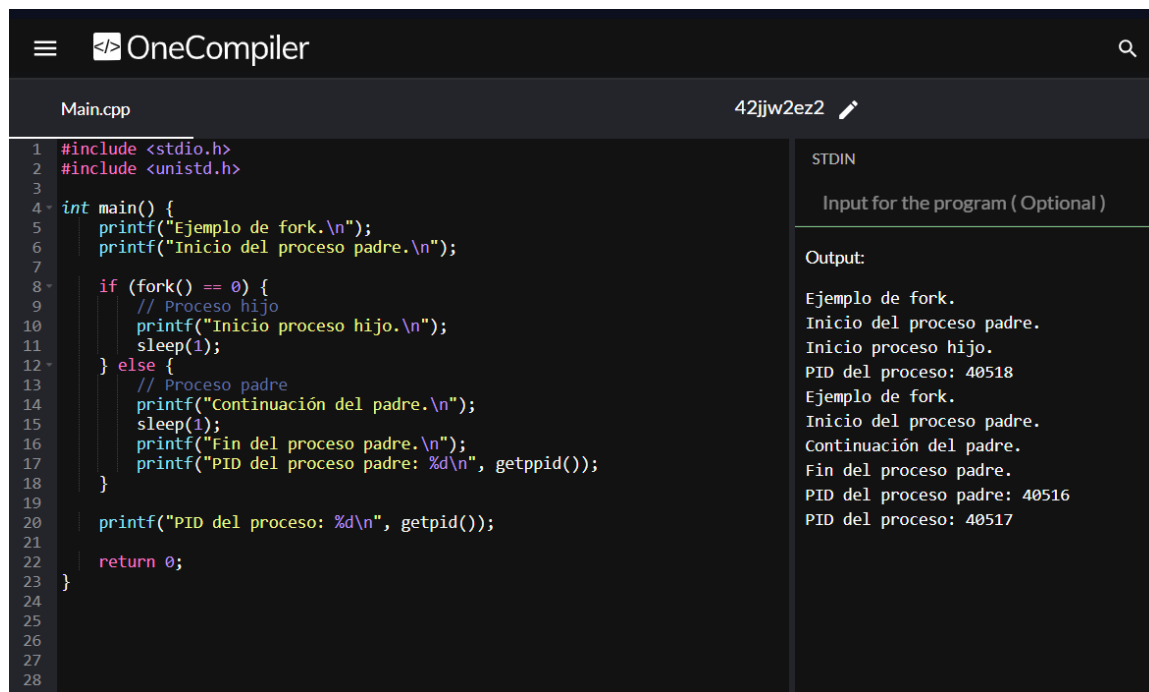
En el código:

- El PID del proceso padre se obtiene con `getpid()`.
- El PID del proceso hijo se obtiene con `getpid()`.
- El PPID del proceso hijo se obtiene con `getppid()`.

Cuando se ejecutó el programa, se observó esto:

- PID del proceso padre: 40516
- PID del proceso hijo: 40517
- PPID del proceso hijo: 40518

El proceso padre tiene un PID de 40516 y un PPID de 40516, mientras que el proceso hijo tiene un PID de 40517 y un PPID de 40516.



```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main() {
5     printf("Ejemplo de fork.\n");
6     printf("Inicio del proceso padre.\n");
7
8     if (fork() == 0) {
9         // Proceso hijo
10        printf("Inicio proceso hijo.\n");
11        sleep(1);
12    } else {
13        // Proceso padre
14        printf("Continuación del padre.\n");
15        sleep(1);
16        printf("Fin del proceso padre.\n");
17        printf("PID del proceso padre: %d\n", getpid());
18    }
19
20    printf("PID del proceso: %d\n", getpid());
21
22    return 0;
23 }
```

Output:

```
Ejemplo de fork.
Inicio del proceso padre.
Inicio proceso hijo.
PID del proceso: 40518
Ejemplo de fork.
Inicio del proceso padre.
Continuación del padre.
Fin del proceso padre.
PID del proceso padre: 40516
PID del proceso: 40517
```

Ilustración 3 Ejecución de un proceso padre y su hijo

### ¿Qué realiza la función `sleep`? ¿Qué proceso concluye antes de su ejecución?

La función `sleep(1)` detiene la ejecución del proceso que la llama durante 1 segundo. En el caso del código, tanto el proceso hijo como el proceso padre llaman a `sleep(1)`. No se puede determinar de antemano cuál de los procesos concluye primero, ya que ambos se ejecutan en paralelo y su finalización depende del sistema operativo y de la carga de trabajo en el momento.

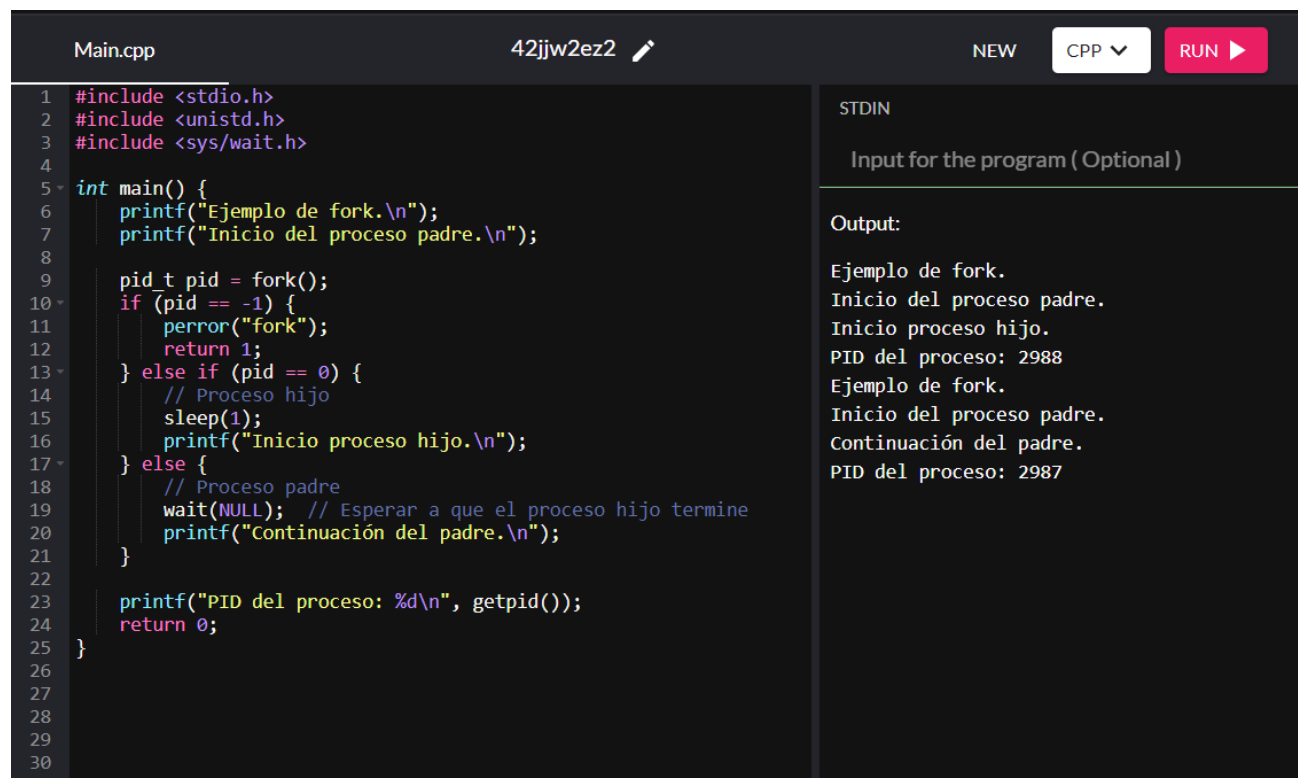
### ¿Qué ocurre cuando la llamada al sistema `fork` devuelve un valor negativo?

Cuando `fork()` devuelve un valor negativo, indica que la creación del proceso hijo ha fallado. Esto puede ocurrir debido a la falta de recursos del sistema, como memoria insuficiente o límites de procesos alcanzados. En tal caso, `fork()` retorna `-1` y típicamente se utiliza `perror("fork")` para imprimir un mensaje de error.

### ¿Cuál es la primera instrucción que ejecuta el proceso hijo?

La primera instrucción que ejecuta el proceso hijo en el código proporcionado es:  
`printf("Inicio proceso hijo.\n");`

**Modifique el código para que el proceso padre imprima su mensaje antes que el hijo.**



```
Main.cpp 42jjw2ez2 NEW CPP RUN
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 int main() {
6     printf("Ejemplo de fork.\n");
7     printf("Inicio del proceso padre.\n");
8
9     pid_t pid = fork();
10    if (pid == -1) {
11        perror("fork");
12        return 1;
13    } else if (pid == 0) {
14        // Proceso hijo
15        sleep(1);
16        printf("Inicio proceso hijo.\n");
17    } else {
18        // Proceso padre
19        wait(NULL); // Esperar a que el proceso hijo termine
20        printf("Continuación del padre.\n");
21    }
22
23    printf("PID del proceso: %d\n", getpid());
24    return 0;
25 }
```

STDIN

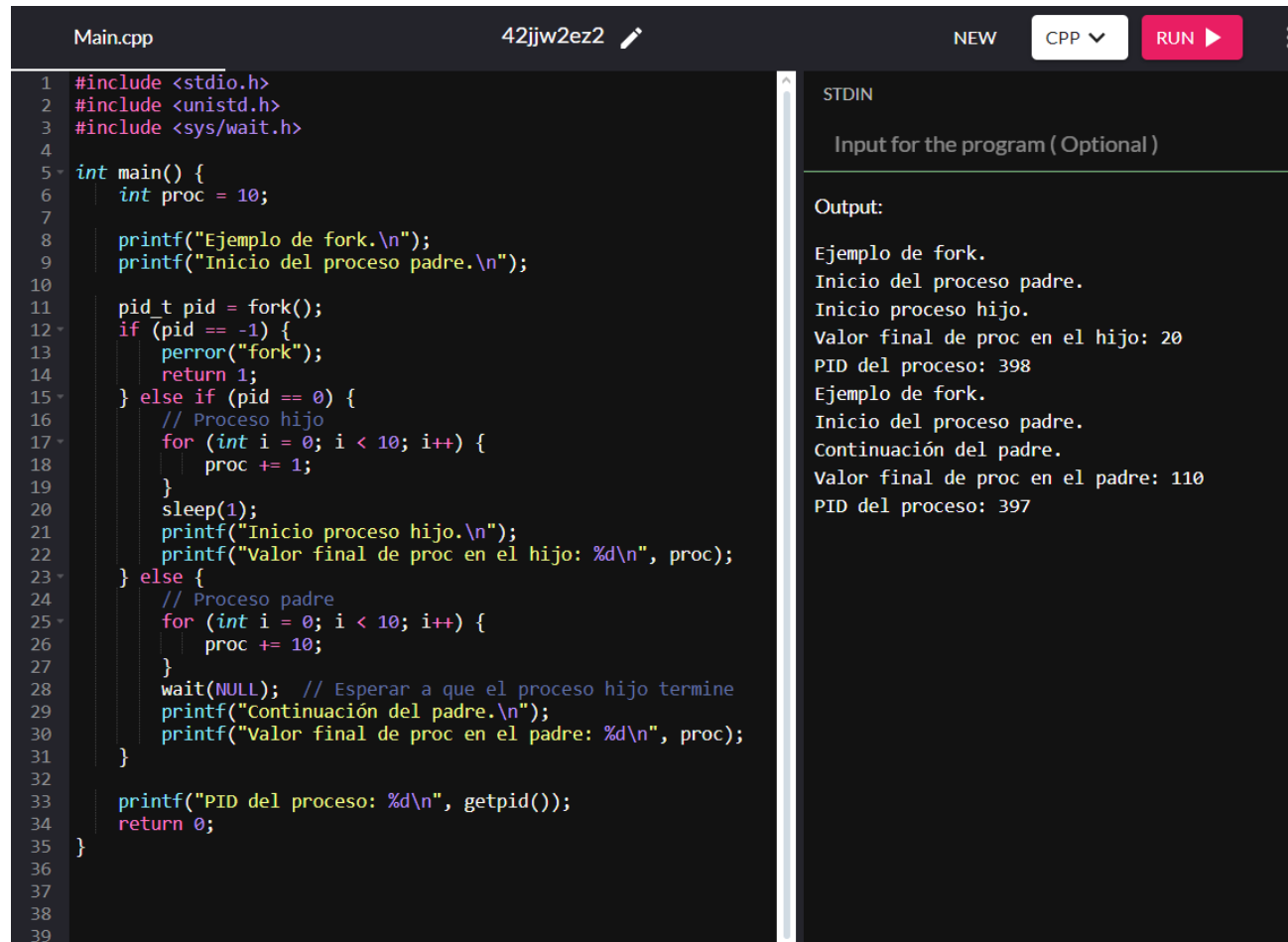
Input for the program (Optional)

Output:

Ejemplo de fork.  
Inicio del proceso padre.  
Inicio proceso hijo.  
PID del proceso: 2988  
Ejemplo de fork.  
Inicio del proceso padre.  
Continuación del padre.  
PID del proceso: 2987

Ilustración 4 Código modificado para que imprima el mensaje de padre primero

Modifique el código fuente del programa declarando una variable entera llamada `proc` e inicializándola a 10. Dicha variable deberá incrementarse 10 veces en el padre y de 10 en 10. Mientras que el hijo la incrementará 10 veces de 1 en 1. ¿Cuál será el valor final de la variable para el padre y para el hijo?



```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 int main() {
6     int proc = 10;
7
8     printf("Ejemplo de fork.\n");
9     printf("Inicio del proceso padre.\n");
10
11     pid_t pid = fork();
12     if (pid == -1) {
13         perror("fork");
14         return 1;
15     } else if (pid == 0) {
16         // Proceso hijo
17         for (int i = 0; i < 10; i++) {
18             proc += 1;
19         }
20         sleep(1);
21         printf("Inicio proceso hijo.\n");
22         printf("Valor final de proc en el hijo: %d\n", proc);
23     } else {
24         // Proceso padre
25         for (int i = 0; i < 10; i++) {
26             proc += 10;
27         }
28         wait(NULL); // Esperar a que el proceso hijo termine
29         printf("Continuación del padre.\n");
30         printf("Valor final de proc en el padre: %d\n", proc);
31     }
32
33     printf("PID del proceso: %d\n", getpid());
34     return 0;
35 }
```

STDIN

Input for the program ( Optional )

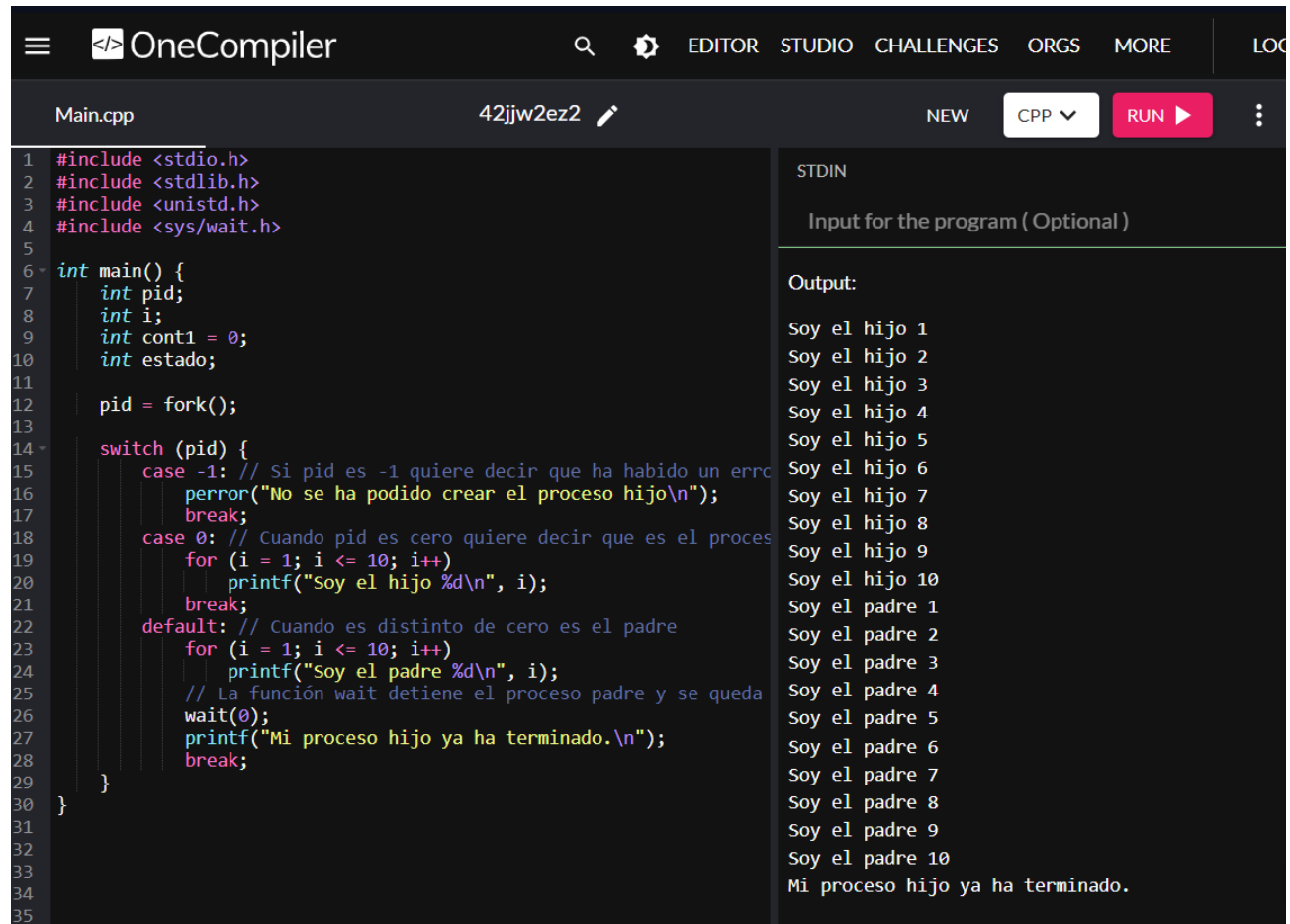
Output:

Ejemplo de fork.  
Inicio del proceso padre.  
Inicio proceso hijo.  
Valor final de proc en el hijo: 20  
PID del proceso: 398  
Ejemplo de fork.  
Inicio del proceso padre.  
Continuación del padre.  
Valor final de proc en el padre: 110  
PID del proceso: 397

Ilustración 5 Código que imprime sumatorio propio del padre e hijo

### 3.1.2 Creación de procesos hijos

Creación de un proceso padre que escribe 10 veces “Soy el padre” y de un proceso hijo que escribe 10 veces “Soy el hijo”.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main() {
7     int pid;
8     int i;
9     int cont1 = 0;
10    int estado;
11
12    pid = fork();
13
14    switch (pid) {
15        case -1: // Si pid es -1 quiere decir que ha habido un error
16            perror("No se ha podido crear el proceso hijo\n");
17            break;
18        case 0: // Cuando pid es cero quiere decir que es el proceso hijo
19            for (i = 1; i <= 10; i++)
20                printf("Soy el hijo %d\n", i);
21            break;
22        default: // Cuando es distinto de cero es el padre
23            for (i = 1; i <= 10; i++)
24                printf("Soy el padre %d\n", i);
25            // La función wait detiene el proceso padre y se queda esperando
26            wait(0);
27            printf("Mi proceso hijo ya ha terminado.\n");
28            break;
29    }
30 }
31
32
33
34
35
```

Output:

Soy el hijo 1  
Soy el hijo 2  
Soy el hijo 3  
Soy el hijo 4  
Soy el hijo 5  
Soy el hijo 6  
Soy el hijo 7  
Soy el hijo 8  
Soy el hijo 9  
Soy el hijo 10  
Soy el padre 1  
Soy el padre 2  
Soy el padre 3  
Soy el padre 4  
Soy el padre 5  
Soy el padre 6  
Soy el padre 7  
Soy el padre 8  
Soy el padre 9  
Soy el padre 10  
Mi proceso hijo ya ha terminado.

Ilustración 6 Imprimir 10 veces soy el hijo y soy el padre

#### ¿Qué realiza la función wait?

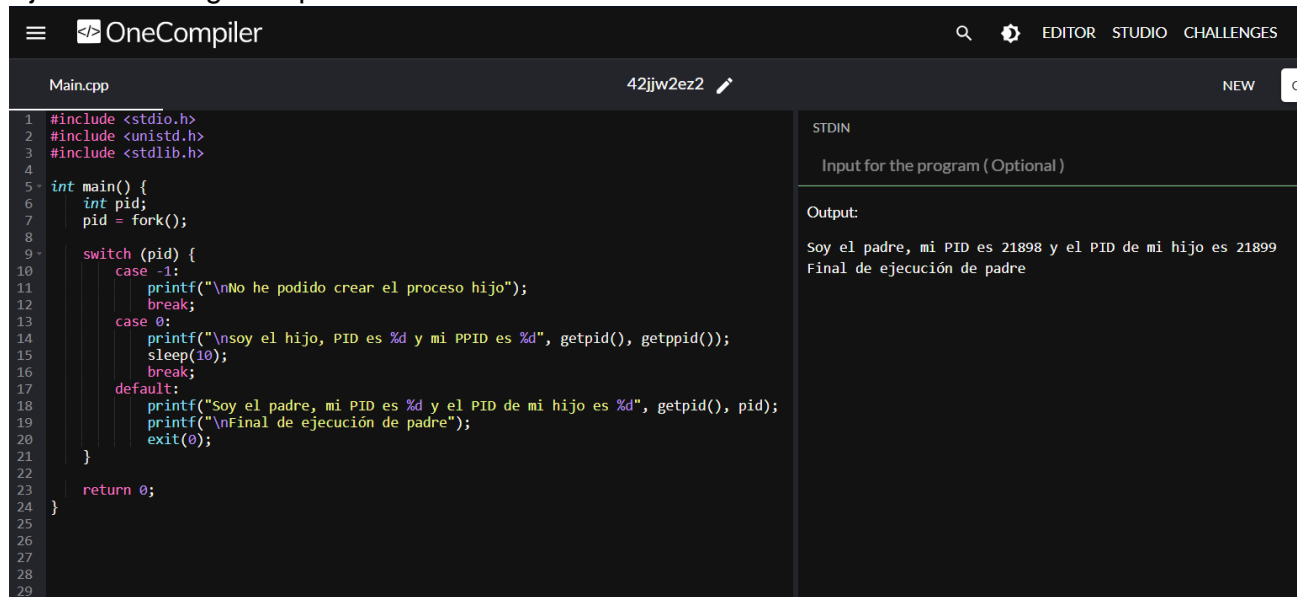
La función `wait` en C es utilizada por un proceso para esperar a que uno de sus procesos hijos termine su ejecución.

1. **Esperar la finalización del proceso hijo:** La función `wait` hace que el proceso padre se detenga y espere hasta que uno de sus procesos hijos termine su ejecución. Esto evita que el proceso padre continúe ejecutándose hasta que el proceso hijo haya terminado.
2. **Recibir el estado de salida del proceso hijo:** `wait` también permite que el proceso padre reciba el estado de salida del proceso hijo, lo cual puede incluir información sobre si el proceso hijo terminó normalmente o si fue interrumpido por una señal.

3. **Liberar recursos:** Cuando un proceso hijo termina, su entrada en la tabla de procesos aún consume recursos del sistema hasta que el proceso padre llama a `wait`. Al llamar a `wait`, el proceso padre libera estos recursos, lo cual es importante para evitar la acumulación de procesos "zombies".

### 3.1.4 Creación de un proceso hijo sin espera.

Ejecutar en segundo plano.



```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 int main() {
6     int pid;
7     pid = fork();
8
9     switch (pid) {
10         case -1:
11             printf("\nNo he podido crear el proceso hijo");
12             break;
13         case 0:
14             printf("\nsoy el hijo, PID es %d y mi PPID es %d", getpid(), getppid());
15             sleep(10);
16             break;
17         default:
18             printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d", getpid(), pid);
19             printf("\nFinal de ejecución de padre");
20             exit(0);
21     }
22
23     return 0;
24 }
```

STDIN

Input for the program ( Optional )

Output:

Soy el padre, mi PID es 21898 y el PID de mi hijo es 21899  
Final de ejecución de padre

Ilustración 7 Proceso hijo sin espera.

### ¿Cuál es el PPID del proceso hijo?

El PPID (Parent Process ID) del proceso hijo es 21898. Esto se puede ver en la salida del programa, donde el proceso hijo imprime "Soy el hijo, mi PID es 21899 y el PID de mi padre es 21898".

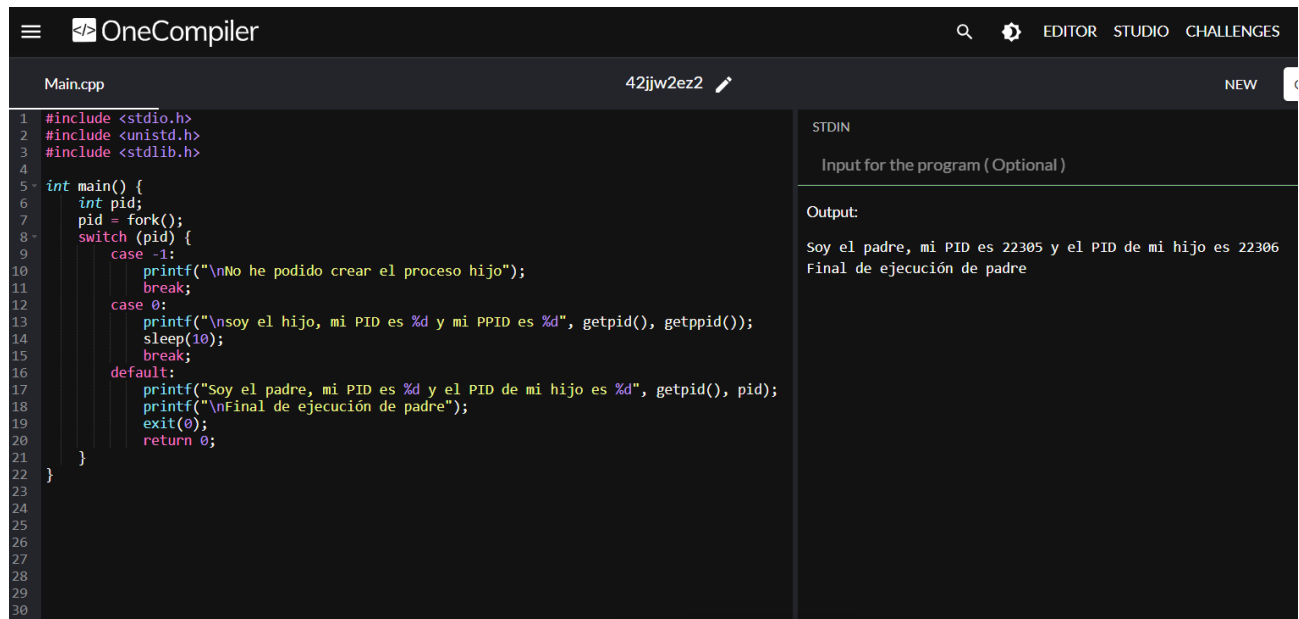
### ¿Cómo se denomina al tipo de proceso hijo?

El tipo de proceso hijo se denomina "proceso fork". Esto se debe a que el programa utiliza la función `fork()` para crear un nuevo proceso hijo a partir del proceso padre.

### 3.1.5 Creación de un proceso hijo con espera.

Ejecutar en segundo plano.





```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>
4
5 int main() {
6     int pid;
7     pid = fork();
8     switch (pid) {
9         case -1:
10             printf("\nNo he podido crear el proceso hijo");
11             break;
12         case 0:
13             printf("\nsoy el hijo, mi PID es %d y mi PPID es %d", getpid(), getppid());
14             sleep(10);
15             break;
16         default:
17             printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d", getpid(), pid);
18             printf("\nFinal de ejecución de padre");
19             exit(0);
20             return 0;
21     }
22 }
```

STDIN

Input for the program (Optional)

Output:

Soy el padre, mi PID es 22305 y el PID de mi hijo es 22306  
Final de ejecución de padre

Ilustración 8 Proceso hijo con espera.

### ¿Cuál es el PPID del proceso hijo?

El PPID (Parent Process ID) del proceso hijo es el PID del proceso padre. En la ejecución, el PID del proceso padre es 22305. Por lo tanto, el PPID del proceso hijo será 22305.

### ¿En qué se diferencia el resultado con el código 3.1.4?

En el código, el proceso padre imprime su mensaje de finalización y termina inmediatamente después de crear el proceso hijo. No espera a que el hijo termine su ejecución. Esto se observa en la llamada a `exit(0)` justo después del mensaje del padre.

El proceso hijo imprime su mensaje y luego duerme por 10 segundos (`sleep(10)`). Como el padre no espera, su mensaje de finalización se imprime antes de que el hijo termine de dormir.

**Modifique el código para que el proceso padre imprima el mensaje de finalización de su ejecución 10 segundos más tarde que el proceso hijo.**



```
1 #include <unistd.h>
2 #include <stdlib.h>
3 #include <sys/wait.h> // Incluir la biblioteca para la función wait
4
5
6 int main() {
7     int pid;
8     pid = fork();
9     switch (pid) {
10         case -1:
11             printf("\nNo he podido crear el proceso hijo");
12             break;
13         case 0:
14             printf("\nSoy el hijo, mi PID es %d y mi PPID es %d", getpid(),
15                 getpid());
16             sleep(10);
17             break;
18         default:
19             // Esperar a que el proceso hijo termine
20             wait(0);
21             sleep(10); // Dormir 10 segundos más después de que el hijo
22             termine
23             printf("Soy el padre, mi PID es %d y el PID de mi hijo es %d",
24                 getpid(), pid);
25             printf("\nFinal de ejecución de padre");
26             exit(0);
27             return 0;
28     }
29 }
```

Soy el hijo, mi PID es 186 y mi PPID es 185 Soy el padre, mi PID es 185 y el PID de mi hijo es 186  
Final de ejecución de padre

Ilustración 9 Mensaje de finalización 10 segundos más tarde que el proceso hijo.

### 3.1.6 Creación de un proceso padre y de tres procesos hijos. Cada uno imprime en pantalla del 1 al 10.

```
1 void hijoHasAlgo(int numero);
2
3 int main() {
4     int numero, i, pid, estado;
5     for (numero = 1; numero <= NUM_PROC; numero++) {
6         pid = fork();
7         switch (pid) {
8             case -1:
9                 fprintf(stderr, "Error al crear el proceso\n");
10                break;
11            case 0:
12                hijoHasAlgo(numero);
13                break;
14            default:
15                printf("Ejecutando el padre\n");
16                printf("Mi hijo %d ha terminado.\n", numero);
17                wait(0);
18                exit(0);
19            }
20        }
21        return 0;
22    }
23
24 void hijoHasAlgo(int numero) {
25     printf("Ejecutando el hijo %d:\n", numero);
26     for (int i = 1; i <= MAX; i++) {
27         printf("%d\t", i);
28     }
29     printf("\n");
30 }
```

Ejecutando el padre  
Mi hijo 1 ha terminado.  
Ejecutando el hijo 1:  
1 2 3 4 5 6 7 8 9 10  
Ejecutando el padre  
Mi hijo 2 ha terminado.  
Ejecutando el hijo 2:  
1 2 3 4 5 6 7 8 9 10  
Ejecutando el padre  
Mi hijo 3 ha terminado.  
Ejecutando el hijo 3:  
1 2 3 4 5 6 7 8 9 10

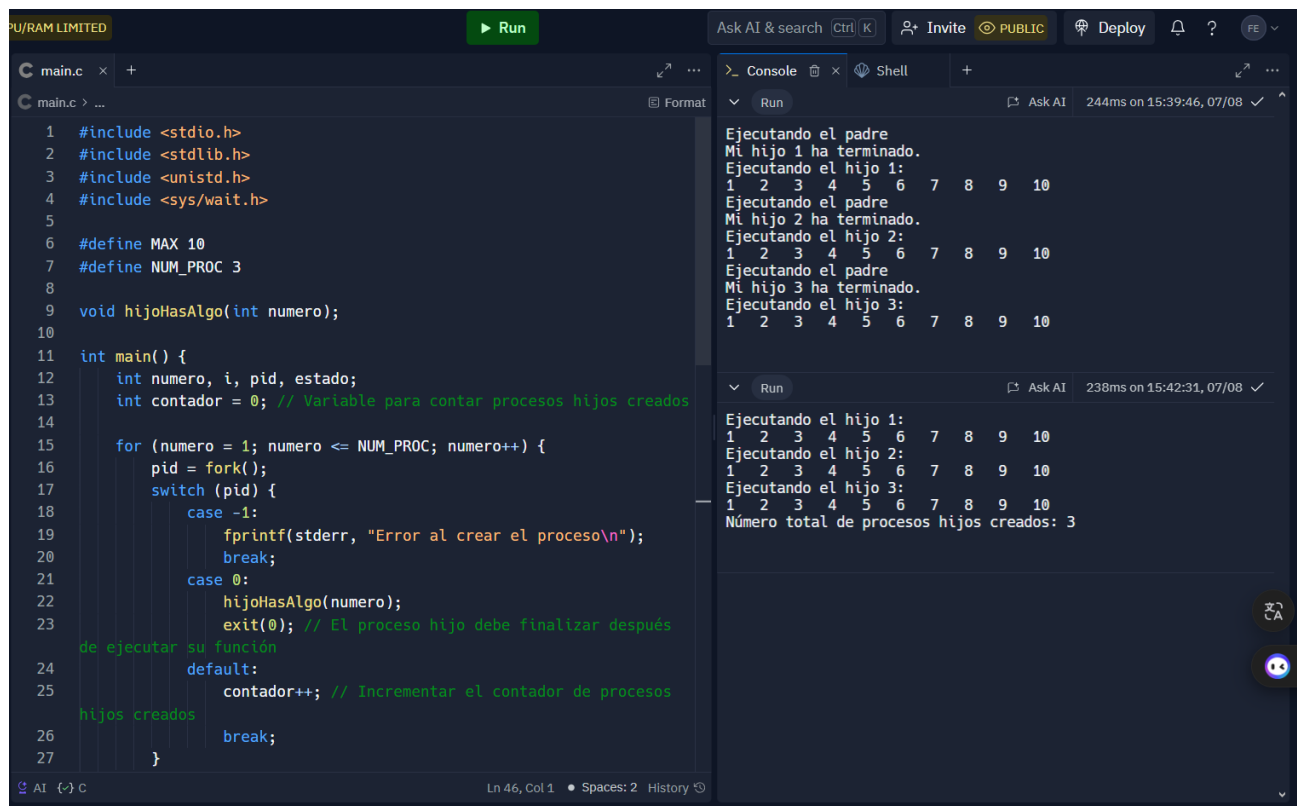
Ilustración 10 Proceso padre y de tres procesos hijos

¿Qué proceso **terminará** primero? ¿Por qué?

Para determinar cuál proceso terminará primero, debemos considerar que los procesos se ejecutan en paralelo y compiten por recursos de CPU. Sin embargo, no podemos predecir con certeza cuál será el orden exacto de finalización debido a la naturaleza concurrente de los procesos.

El orden de finalización puede variar según el sistema operativo y la planificación de procesos. Factores como la asignación de recursos, la carga del sistema y la sincronización afectarán el resultado.

**Modificar el código para que aparezca al final en pantalla el número de procesos hijos que se ejecutaron.**



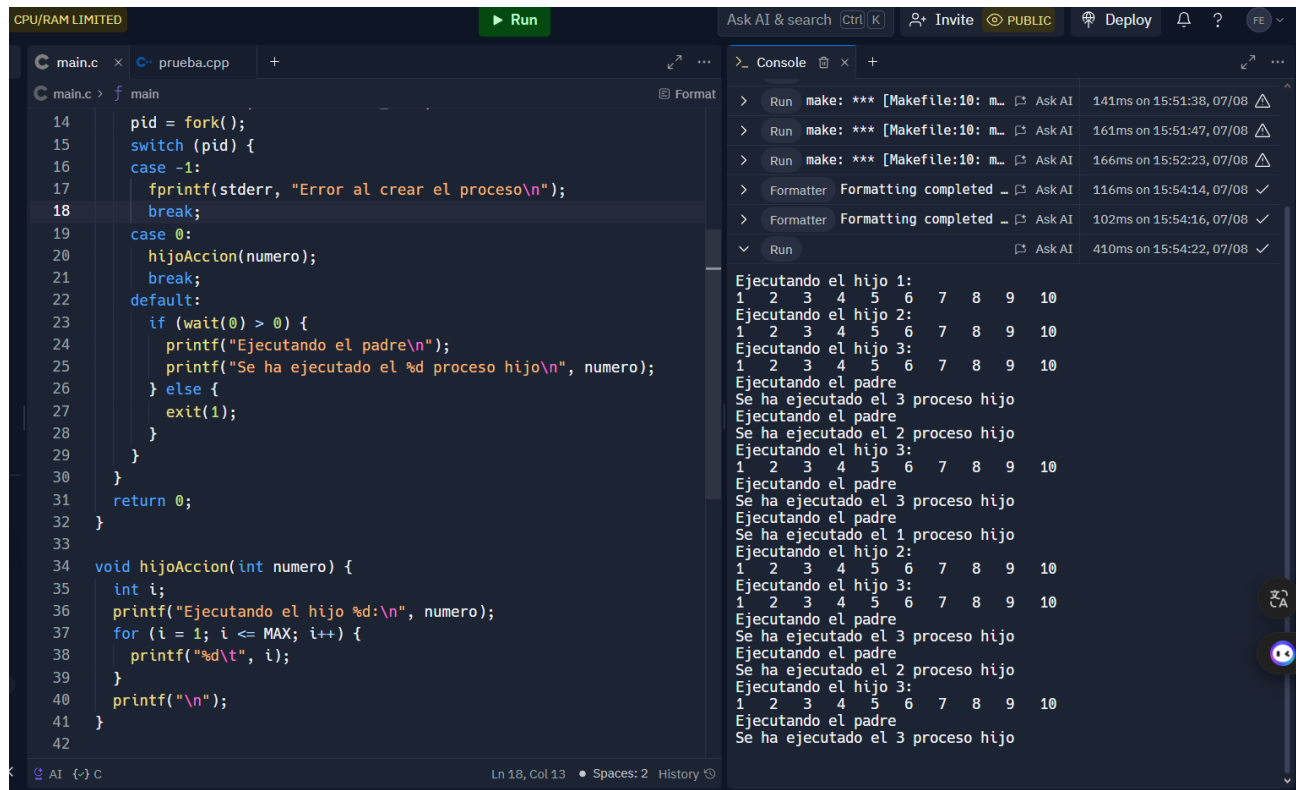
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 #define MAX 10
7 #define NUM_PROC 3
8
9 void hijoHasAlgo(int numero);
10
11 int main() {
12     int numero, i, pid, estado;
13     int contador = 0; // Variable para contar procesos hijos creados
14
15     for (numero = 1; numero <= NUM_PROC; numero++) {
16         pid = fork();
17         switch (pid) {
18             case -1:
19                 fprintf(stderr, "Error al crear el proceso\n");
20                 break;
21             case 0:
22                 hijoHasAlgo(numero);
23                 exit(0); // El proceso hijo debe finalizar después
24             // de ejecutar su función
25             default:
26                 contador++; // Incrementar el contador de procesos
27                 // hijos creados
28         }
29     }
30
31     // Ejecutando el padre
32     for (i = 1; i <= MAX; i++) {
33         printf("%d ", i);
34     }
35     printf("\n");
36
37     // Esperar a que los hijos terminen
38     wait(NULL);
39
40     // Imprimir el número total de procesos hijos creados
41     printf("Número total de procesos hijos creados: %d\n", contador);
42 }
```

Ejecutando el padre  
Mi hijo 1 ha terminado.  
Ejecutando el hijo 1:  
1 2 3 4 5 6 7 8 9 10  
Ejecutando el padre  
Mi hijo 2 ha terminado.  
Ejecutando el hijo 2:  
1 2 3 4 5 6 7 8 9 10  
Ejecutando el padre  
Mi hijo 3 ha terminado.  
Ejecutando el hijo 3:  
1 2 3 4 5 6 7 8 9 10  
Número total de procesos hijos creados: 3

Ilustración 11 Código que imprime número el de procesos hijos que se ejecutaron.

### 3.1.7 Ejecución concurrente de procesos

¿Qué salida produce?



```

14 pid = fork();
15 switch (pid) {
16 case -1:
17     fprintf(stderr, "Error al crear el proceso\n");
18     break;
19 case 0:
20     hijoAccion(numero);
21     break;
22 default:
23     if (wait(0) > 0) {
24         printf("Ejecutando el padre\n");
25         printf("Se ha ejecutado el %d proceso hijo\n", numero);
26     } else {
27         exit(1);
28     }
29 }
30 }
31 return 0;
32 }
33
34 void hijoAccion(int numero) {
35     int i;
36     printf("Ejecutando el hijo %d:\n", numero);
37     for (i = 1; i <= MAX; i++) {
38         printf("%d\t", i);
39     }
40     printf("\n");
41 }
42

```

Console Output:

```

Run make: *** [Makefile:10: m... Ask AI 141ms on 15:51:38, 07/08
Run make: *** [Makefile:10: m... Ask AI 161ms on 15:51:47, 07/08
Run make: *** [Makefile:10: m... Ask AI 166ms on 15:52:23, 07/08
Formatter Formatting completed ... Ask AI 116ms on 15:54:14, 07/08
Formatter Formatting completed ... Ask AI 102ms on 15:54:16, 07/08
Run Ask AI 410ms on 15:54:22, 07/08

Ejecutando el hijo 1:
1 2 3 4 5 6 7 8 9 10
Ejecutando el hijo 2:
1 2 3 4 5 6 7 8 9 10
Ejecutando el hijo 3:
1 2 3 4 5 6 7 8 9 10
Ejecutando el padre
Se ha ejecutado el 3 proceso hijo
Ejecutando el padre
Se ha ejecutado el 2 proceso hijo
Ejecutando el hijo 3:
1 2 3 4 5 6 7 8 9 10
Ejecutando el padre
Se ha ejecutado el 3 proceso hijo
Ejecutando el padre
Se ha ejecutado el 1 proceso hijo
Ejecutando el hijo 2:
1 2 3 4 5 6 7 8 9 10
Ejecutando el hijo 3:
1 2 3 4 5 6 7 8 9 10
Ejecutando el padre
Se ha ejecutado el 3 proceso hijo
Ejecutando el padre
Se ha ejecutado el 2 proceso hijo
Ejecutando el hijo 3:
1 2 3 4 5 6 7 8 9 10
Ejecutando el padre
Se ha ejecutado el 3 proceso hijo

```

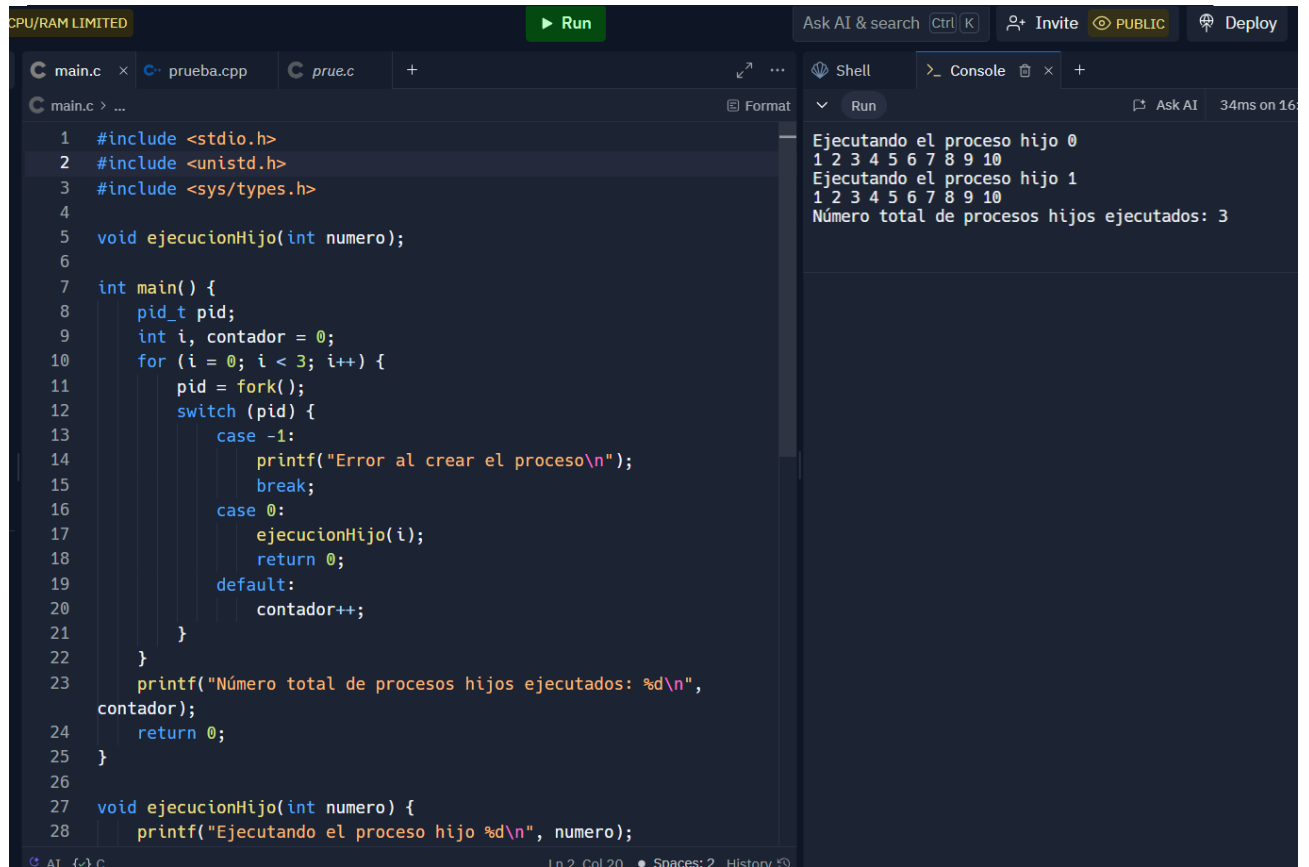
Ilustración 12 Ejecución concurrente de procesos

### ¿Cuál es la diferencia con el código 3.1.6?

En este código, se crean tres procesos hijos utilizando `fork()`, y cada uno de ellos imprime los números del 1 al 10. Sin embargo, no se muestra el número total de procesos hijos ejecutados al final.

Para modificar el código y mostrar el número de procesos hijos ejecutados, podemos agregar una variable contador y aumentarla cada vez que se crea un proceso hijo. Luego, al final, imprimimos el valor de ese contador.

**Modificar el código para que aparezca al final en pantalla el número de procesos hijos que se ejecutaron.**



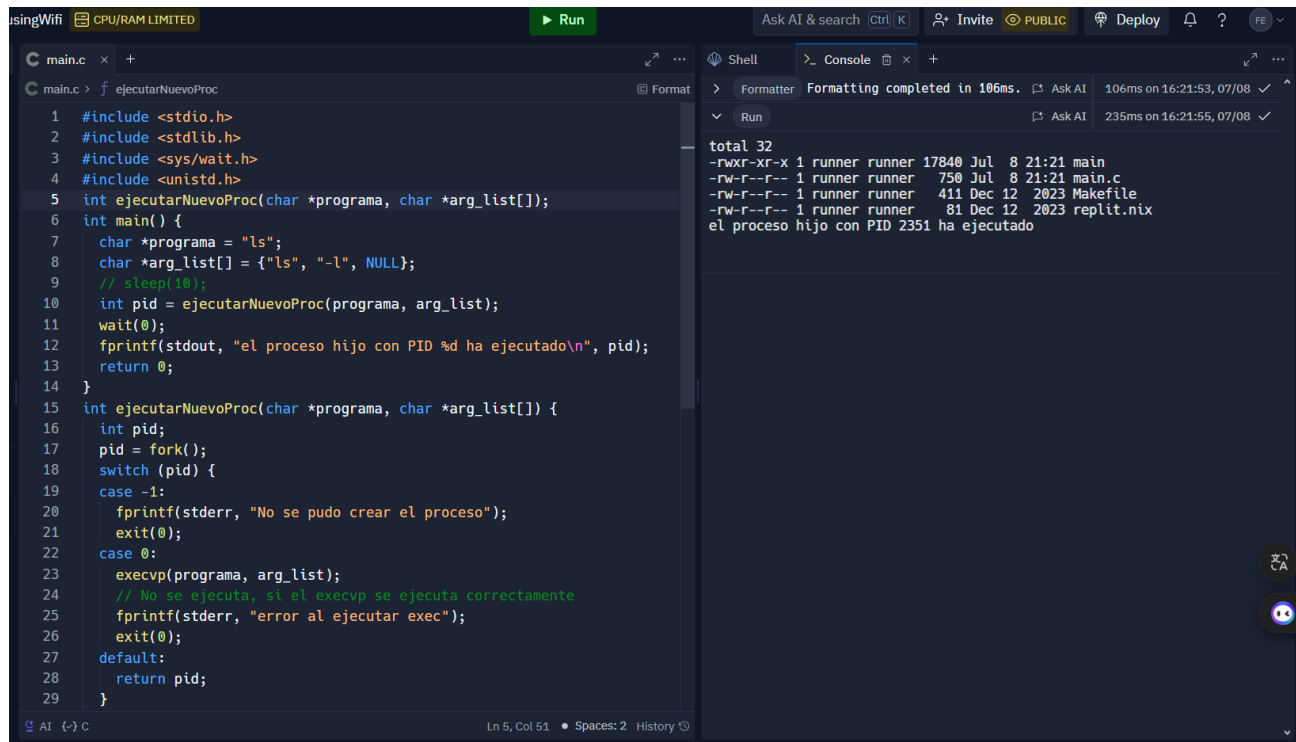
```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/types.h>
4
5 void ejecucionHijo(int numero);
6
7 int main() {
8     pid_t pid;
9     int i, contador = 0;
10    for (i = 0; i < 3; i++) {
11        pid = fork();
12        switch (pid) {
13            case -1:
14                printf("Error al crear el proceso\n");
15                break;
16            case 0:
17                ejecucionHijo(i);
18                return 0;
19            default:
20                contador++;
21        }
22    }
23    printf("Número total de procesos hijos ejecutados: %d\n",
24          contador);
25    return 0;
26
27 void ejecucionHijo(int numero) {
28     printf("Ejecutando el proceso hijo %d\n", numero);
29     for (int j = 1; j <= 10; j++) {
30         printf("%d ", j);
31     }
32     printf("\n");
33 }
```

Ejecutando el proceso hijo 0  
1 2 3 4 5 6 7 8 9 10  
Ejecutando el proceso hijo 1  
1 2 3 4 5 6 7 8 9 10  
Número total de procesos hijos ejecutados: 3

Ilustración 13 Código para que aparezca al final en pantalla el número de procesos hijos

### 3.2 USO DE EXEC

¿Qué información se despliega al ejecutar el código?



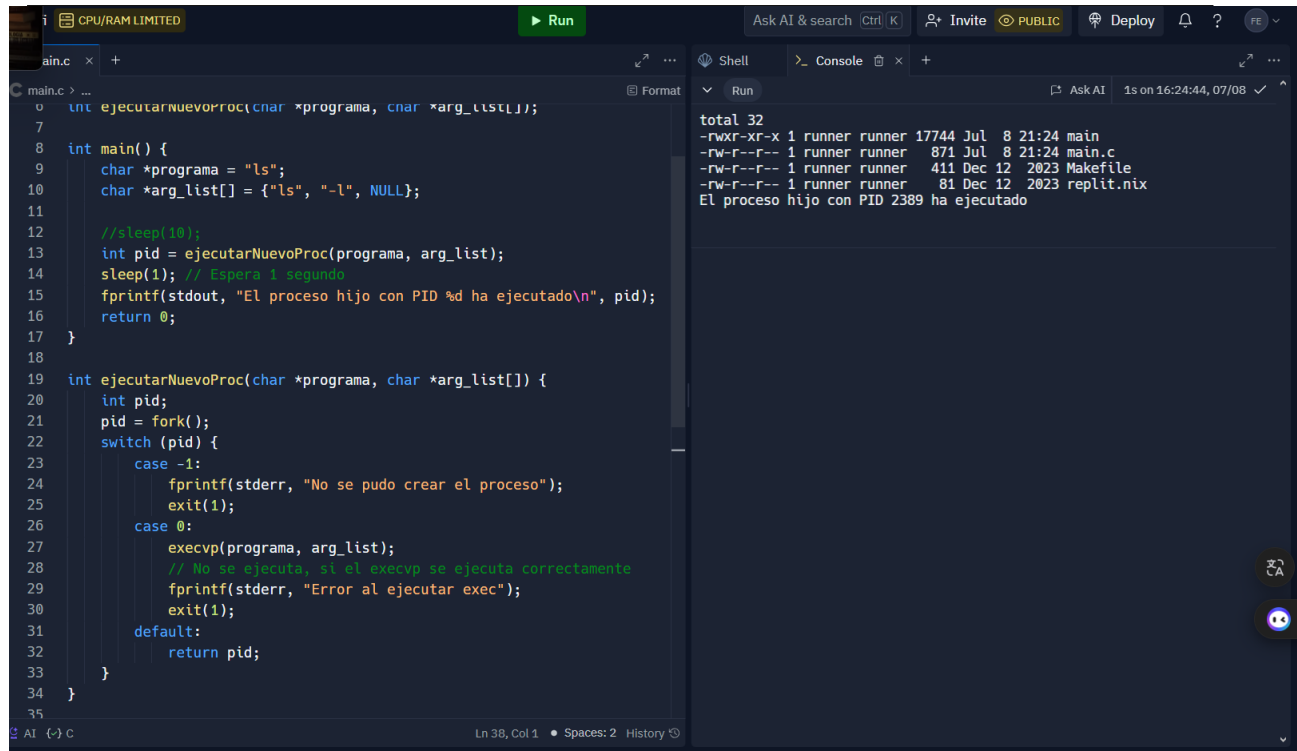
```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5 int ejecutarNuevoProc(char *programa, char *arg_list[]);
6 int main() {
7     char *programa = "ls";
8     char *arg_list[] = {"ls", "-l", NULL};
9     // sleep(10);
10    int pid = ejecutarNuevoProc(programa, arg_list);
11    wait(0);
12    fprintf(stdout, "el proceso hijo con PID %d ha ejecutado\n", pid);
13    return 0;
14 }
15 int ejecutarNuevoProc(char *programa, char *arg_list[]) {
16     int pid;
17     pid = fork();
18     switch (pid) {
19     case -1:
20         fprintf(stderr, "No se pudo crear el proceso");
21         exit(0);
22     case 0:
23         execvp(programa, arg_list);
24         // No se ejecuta, si el execvp se ejecuta correctamente
25         fprintf(stderr, "error al ejecutar exec");
26         exit(0);
27     default:
28         return pid;
29     }
```

total 32  
-rwxr-xr-x 1 runner runner 17840 Jul 8 21:21 main  
-rw-r--r-- 1 runner runner 750 Jul 8 21:21 main.c  
-rw-r--r-- 1 runner runner 411 Dec 12 2023 Makefile  
-rw-r--r-- 1 runner runner 81 Dec 12 2023 replit.nix  
el proceso hijo con PID 2351 ha ejecutado

Ilustración 14 Uso de exec

### ¿Qué sucede si se cambia la función wait por sleep?

- Si reemplazas `wait(NULL)` por `sleep`, el proceso padre simplemente se detendrá durante el número de segundos especificado en `sleep` y luego continuará su ejecución sin esperar realmente a que el proceso hijo termine.
- Si el proceso hijo termina antes de que el tiempo de `sleep` haya pasado, el proceso padre no lo sabrá y continuará durmiendo.
- Si el proceso hijo no ha terminado cuando el proceso padre se despierta, el proceso padre continuará su ejecución de todas formas, lo que puede llevar a comportamientos indeterminados si el programa depende del resultado del proceso hijo.



```

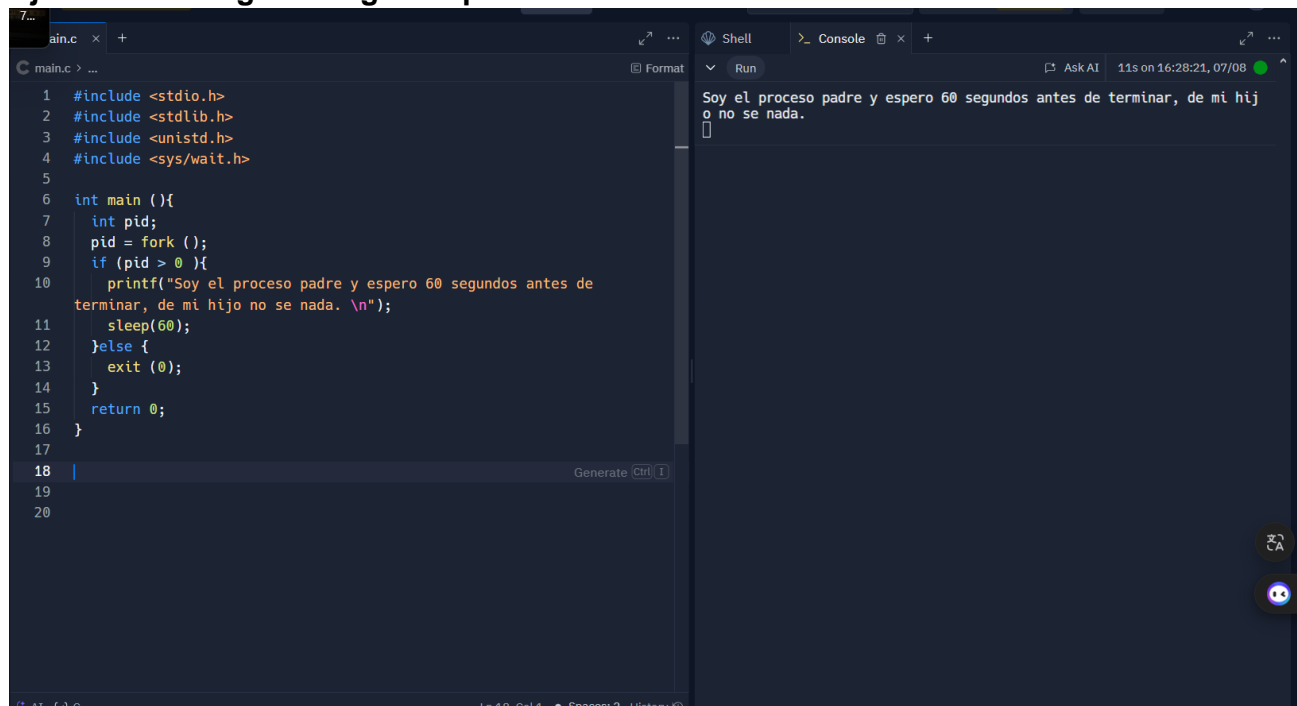
1 int ejecutarNuevoProc(char *programa, char *arg_list[]);
2
3 int main() {
4     char *programa = "ls";
5     char *arg_list[] = {"ls", "-l", NULL};
6
7     //sleep(10);
8     int pid = ejecutarNuevoProc(programa, arg_list);
9     sleep(1); // Espera 1 segundo
10    fprintf(stdout, "El proceso hijo con PID %d ha ejecutado\n", pid);
11    return 0;
12 }
13
14 int ejecutarNuevoProc(char *programa, char *arg_list[]) {
15     int pid;
16     pid = fork();
17     switch (pid) {
18         case -1:
19             fprintf(stderr, "No se pudo crear el proceso");
20             exit(1);
21         case 0:
22             execvp(programa, arg_list);
23             // No se ejecuta, si el execvp se ejecuta correctamente
24             fprintf(stderr, "Error al ejecutar exec");
25             exit(1);
26         default:
27             return pid;
28     }
29 }
30
31
32
33
34
35

```

total 32  
-rwxr-xr-x 1 runner runner 17744 Jul 8 21:24 main  
-rw-r--r-- 1 runner runner 871 Jul 8 21:24 main.c  
-rw-r--r-- 1 runner runner 411 Dec 12 2023 Makefile  
-rw-r--r-- 1 runner runner 81 Dec 12 2023 replit.nix  
El proceso hijo con PID 2389 ha ejecutado

Ilustración 15 Se ha cambiado la función wait por sleep

### 3.2.1 Creación de un proceso zombie Ejecutar el código en segundo plano



```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 int main (){
7     int pid;
8     pid = fork ();
9     if (pid > 0){
10        printf("Soy el proceso padre y espero 60 segundos antes de
11terminar, de mi hijo no se nada. \n");
12        sleep(60);
13    }else {
14        exit (0);
15    }
16    return 0;
17 }
18
19
20

```

Soy el proceso padre y espero 60 segundos antes de terminar, de mi hijo no se nada.

Ilustración 16 Creación de un proceso zombie

Ejecutar el comando top y verificar que existe un proceso zombie

### ¿Cómo solucionaría el problema?

Cuando el proceso hijo termina después de llamar a `exit(0)`, se convierte en un "proceso zombie". Un proceso zombie es aquel cuyo proceso padre aún no ha llamado a `wait()` o `waitpid()` para obtener su estado de salida. Aunque el proceso hijo ha terminado de ejecutarse, su entrada en la tabla de procesos aún existe hasta que el proceso padre obtenga su estado de salida.

Para evitar que los procesos hijos se conviertan en zombies, el proceso padre debe manejar adecuadamente la finalización de los procesos hijos. Esto se logra utilizando la función `wait()` o `waitpid()` después de que el proceso hijo haya terminado. Estas funciones permiten al proceso padre esperar y obtener el estado de salida del proceso hijo, lo que elimina la entrada del proceso zombie de la tabla de procesos del sistema.

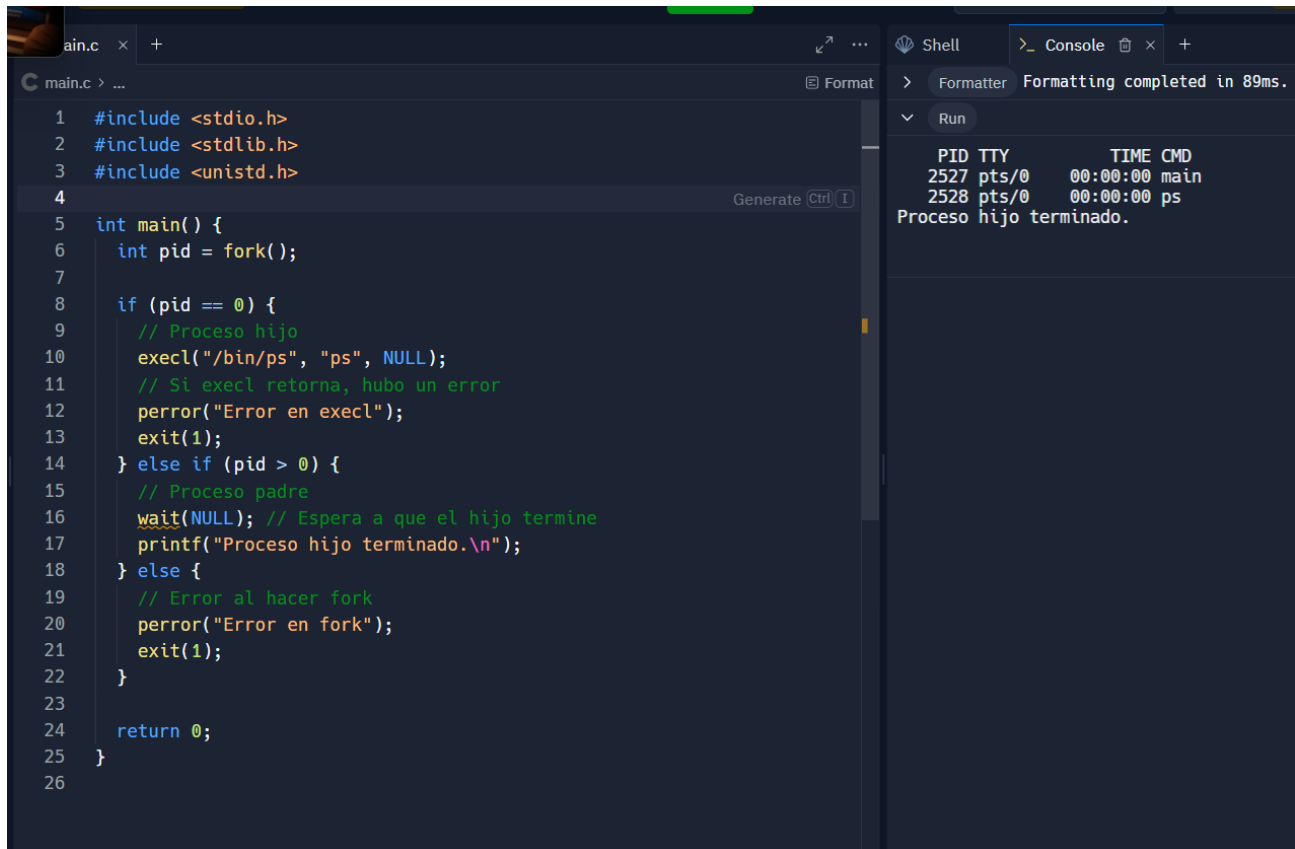
```
C main.c > f main
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main() {
7      int pid;
8      pid = fork();
9      if (pid > 0) {
10         printf("Soy el proceso padre y espero 60 segundos antes de
terminar, de mi hijo no se nada.\n");
11         sleep(60);
12         wait(NULL); // Espera a que el hijo termine y limpia su estado
13     } else {
14         exit(0);
15     }
16     return 0;
17 }
18
```

Ilustración 17 Código que soluciona el problema planteado



### 3.2.2 Uso de exec

Crear un **proceso** hijo que liste los procesos del sistema usando `execl`



The screenshot shows a C program in a code editor. The program uses `fork()` to create a child process. The child process calls `execl("/bin/ps", "ps", NULL);` to execute the `ps` command. The parent process then calls `wait(NULL)` to wait for the child to finish and prints a message. The console output shows the output of the `ps` command.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 int main() {
6     int pid = fork();
7
8     if (pid == 0) {
9         // Proceso hijo
10        execl("/bin/ps", "ps", NULL);
11        // Si execl retorna, hubo un error
12        perror("Error en execl");
13        exit(1);
14    } else if (pid > 0) {
15        // Proceso padre
16        wait(NULL); // Espera a que el hijo termine
17        printf("Proceso hijo terminado.\n");
18    } else {
19        // Error al hacer fork
20        perror("Error en fork");
21        exit(1);
22    }
23
24    return 0;
25 }
26
```

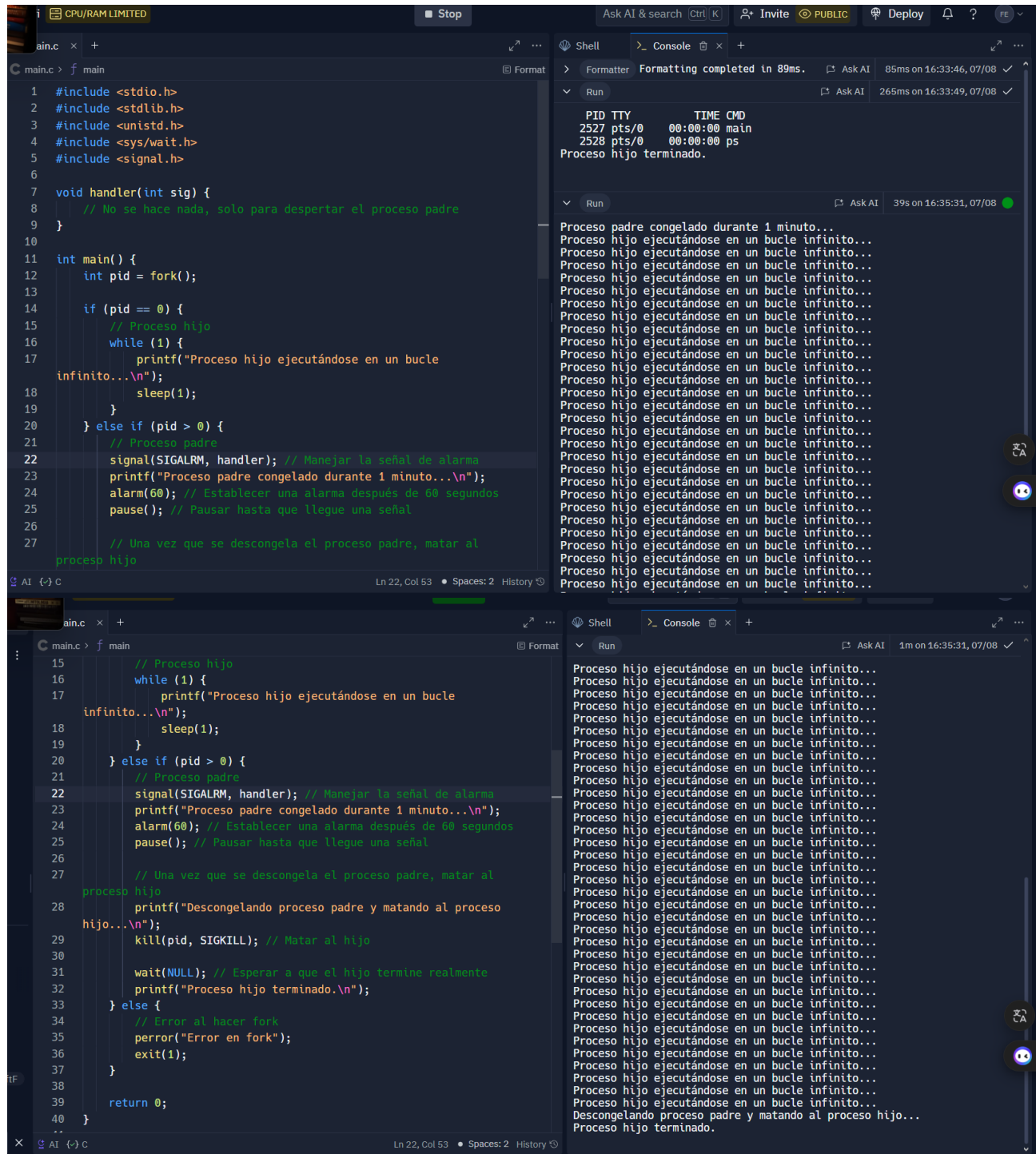
Console Output:

PID	TTY	TIME	CMD
2527	pts/0	00:00:00	main
2528	pts/0	00:00:00	ps

Proceso hijo terminado.

Ilustración 18 Proceso hijo que liste los procesos del sistema usando `execl`

**3.2.3 Crear un proceso hijo y colocarlo dentro de un bucle infinito.**  
**El proceso padre se congelará durante 1 minuto.**  
**Una vez que se descongele el proceso padre se deberá matar al proceso hijo.**



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <signal.h>
6
7 void handler(int sig) {
8     // No se hace nada, solo para despertar el proceso padre
9 }
10
11 int main() {
12     int pid = fork();
13
14     if (pid == 0) {
15         // Proceso hijo
16         while (1) {
17             printf("Proceso hijo ejecutándose en un bucle infinito...\n");
18             sleep(1);
19         }
20     } else if (pid > 0) {
21         // Proceso padre
22         signal(SIGALRM, handler); // Manejar la señal de alarma
23         printf("Proceso padre congelado durante 1 minuto...\n");
24         alarm(60); // Establecer una alarma después de 60 segundos
25         pause(); // Pausar hasta que llegue una señal
26
27         // Una vez que se descongela el proceso padre, matar al
28         // proceso hijo
29         printf("Descongelando proceso padre y matando al proceso
30         hijo...\n");
31         kill(pid, SIGKILL); // Matar al hijo
32
33         wait(NULL); // Esperar a que el hijo termine realmente
34         printf("Proceso hijo terminado.\n");
35     } else {
36         // Error al hacer fork
37         perror("Error en fork");
38         exit(1);
39     }
40
41     return 0;
42 }
```

Ilustración 19 Creación de un proceso hijo colocado dentro de un bucle infinito.

### 3. CONCLUSIONES Y RECOMENDACIONES

- Se pudo comprender la sincronización entre procesos, como esperar a que un proceso hijo termine antes de que el padre continúe, es crucial para evitar problemas como los procesos zombies (procesos hijos que han terminado pero cuyo estado no ha sido recogido por el padre).
- Aprendimos a manejar adecuadamente la creación y finalización de procesos utilizando funciones como `wait()`, `waitpid()` y señales como `SIGKILL` para asegurar que los recursos del sistema se utilicen de manera eficiente y segura.
- Cuando se trabaja con múltiples procesos, es crucial diseñar y desarrollar aplicaciones que sean eficientes y seguras. Esto implica pensar en la sincronización, la comunicación entre procesos y la gestión adecuada de recursos compartidos.
- Es esencial manejar adecuadamente la creación y finalización de procesos utilizando funciones como `wait()`, `waitpid()` y señales como `SIGKILL` para asegurar que los recursos del sistema se utilicen de manera eficiente y segura.práctica.

### 4. BIBLIOGRAFÍA

- [1] A. S. Tanenbaum and H. Bos, Modern Operating Systems, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.
- [2] W. Stallings, Operating Systems: Internals and Design Principles, 9th ed. Boston, MA, USA: Pearson, 2017.
- [3] J. López y A. García, "Gestión de Procesos en Sistemas Operativos Unix," Revista Iberoamericana de Informática, vol. 21, no. 3, pp. 45-58, Sep. 2010.