Taller Fork ()

Gabriel Riaño

Dary Palacios

Sistemas Operativos

John Corredor

Pontificia Universidad Javeriana

2025 - 1

Objetivos:

- Comprender y aplicar el funcionamiento de la función fork() como herramienta para la creación de procesos en sistemas operativos tipo Unix/Linux.
- Analizar el comportamiento de los procesos padre, hijo y nieto, incluyendo su identificación con PID y PPID.
- Implementar comunicación entre procesos utilizando pipe() como mecanismo de transferencia de datos.
- Simular una arquitectura jerárquica de procesos (padre → hijo → nieto) para distribuir tareas computacionales.
- Entender y evitar problemas comunes asociados con la creación de procesos, como procesos zombis, procesos huérfanos y errores por exceso de procesos o falta de memoria.

2. Fundamento Teórico

¿Cuál es la función de fork()?

En sistemas operativos tipo Unix/Linux, la función fork() permite que un proceso cree una copia casi idéntica de sí mismo. Este nuevo proceso es el proceso hijo, y ambos (padre e hijo) continúan ejecutando el mismo código desde el punto donde ocurrió el fork().

Principios básicos:

1. Creación de una copia: El proceso hijo es una réplica del padre, con su propio PID y espacio de ejecución.

2. Identificadores de proceso:
    - El padre conserva su PID.
    - El hijo obtiene un nuevo PID.
    - Ambos comparten el mismo código, pero no la misma memoria.
3. Contador de programa: El valor del contador de instrucciones se copia; ambos procesos comienzan desde la siguiente instrucción tras el fork().

Técnica Copy-on-Write (COW):

- El sistema operativo no copia la memoria inmediatamente tras el fork().
- Padre e hijo comparten páginas de memoria hasta que uno de ellos las modifica.
- Esto optimiza el uso de memoria y mejora el rendimiento.

Herencia de contexto:

- Los descriptores de archivo abiertos por el padre también están disponibles para el hijo.
- Las variables de entorno son heredadas, pero independientes.

Manejo de errores:

fork() puede fallar si:

- No hay suficiente memoria (ENOMEM).
- Se supera el número máximo de procesos (EAGAIN).

Zombis y huérfanos:

- Zombi: Proceso hijo finalizado cuyo estado no ha sido recogido por el padre.
- Huérfano: Proceso hijo cuyo padre finaliza antes.
- Se usa wait() para evitar zombis y init adopta huérfanos automáticamente.

3. Descripción de la implementación

Descripción General:

El programa recibe como argumentos los nombres de dos archivos y las cantidades de números en cada uno. Lee los enteros y los almacena en arreglos dinámicos. Luego, crea procesos para calcular sumas específicas y transmite los resultados al proceso padre usando pipe().

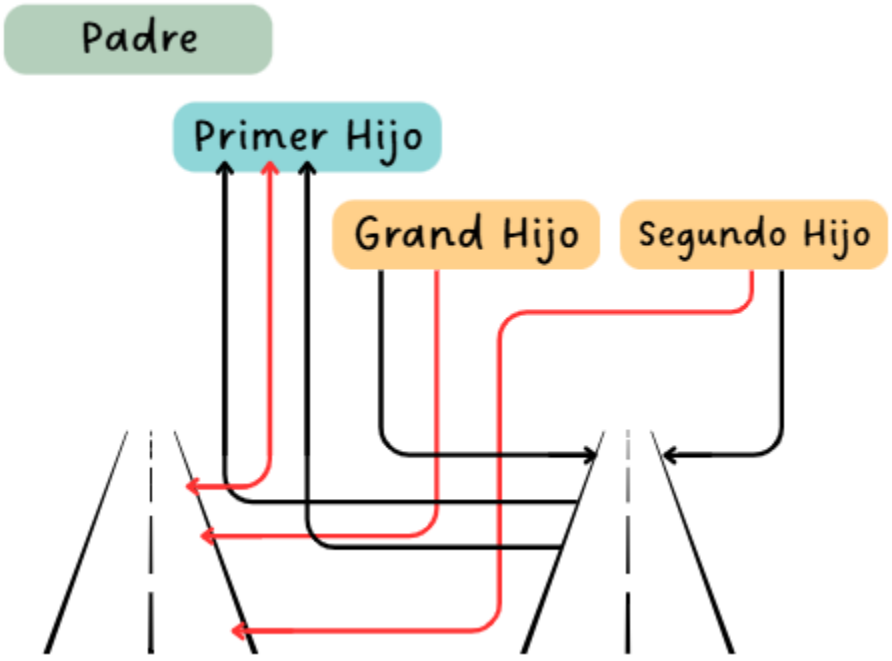

Ilustración 1. Funcionamiento del código

Como se puede observar en la ilustración 1, el proceso **Padre** inicia su ejecución creando un **Primer Hijo** mediante fork(). Este Primer Hijo, a su vez, genera dos procesos adicionales: el **Grand Hijo** y el **Segundo Hijo**.

Tanto el **Grand Hijo** como el **Segundo Hijo** se encargan de realizar la suma de los valores contenidos en sus respectivos arreglos asignados. Una vez se obtiene el resultado de la suma, cada uno lo escribe en un **pipe** dirigido al **Primer Hijo** y en otro pipe que es dirigido al **Padre**.

El **Primer Hijo** lee los dos resultados parciales enviados por sus hijos, los suma y envía este nuevo valor resultante al proceso **Padre** a través del otro pipe.

Finalmente, el **Padre** lee los tres resultados:

- El resultado de la suma del primer arreglo, enviada por el **Grand Hijo**.
- El resultado de la suma del primer arreglo, enviada por el **Segundo Hijo**.
- El valor enviado por el **Primer Hijo** (resultado de sumar los datos del **Grand Hijo** y del **Segundo Hijo**).

Y puede imprimirlos como resultado final del programa.

Descripción Especifica del Código:

1. Carga de Datos

La función cargarArreglo lee los datos de un archivo de texto y los almacena en un arreglo. Esta función recibe el nombre del archivo y un puntero al arreglo donde se guardarán los enteros.

```c
// Función para cargar los enteros desde un archivo en un arreglo
void cargarArreglo(char *nombreArchivo, int *arreglo){

        FILE *file = fopen(nombreArchivo, "r");
        if (file == NULL) {
                perror("Error al abrir el archivo");
                return;
        }

        char linea[256];
        int pos = 0;

        // Imprime los elementos leidos del archivo
        printf("Arreglo de %s: ", nombreArchivo);

        // Lee linea por linea y extrae los números
        while(fgets(linea, sizeof(linea), file)){
                char *token = strtok(linea, " \n");
                while(token != NULL){
                        int numero = (int) atoi(token);
                        printf("%d ", numero);
                        arreglo[pos++] = numero;
                        token = strtok(NULL, " \n");
                }
        }
        printf("\n\n");
        fclose(file);
}
```

Imagen 1. función de carga de datos

2. Cálculo de la Suma

La función sumaArreglo calcula la suma de los elementos de un arreglo. Recibe el arreglo y el número de elementos y devuelve la suma total.

```c
// Función para calcular la suma de los elementos de un arreglo
int sumaArreglo(int *arreglo, int numElementos){
        int suma = 0;
        while(numElementos-- > 0){
                suma += arreglo[numElementos];
        }
        return suma;
}
```

Imagen 2. función de suma de números

3. Creación de Procesos y Comunicación (Bloque Principal)

En el bloque principal se crean los procesos y se gestionan las comunicaciones entre ellos utilizando pipes.

3.1 Proceso Padre

El proceso padre verifica si se han proporcionado los argumentos adecuados y luego crea un pipe principal para la comunicación.

```
int pipefd[2];          // Pipe principal para enviar datos al padre
pipe(pipefd);           // Se crea el pipe
```

Imagen 3. Creación de pipe principal

Luego, el proceso padre crea el primer hijo usando fork().

```
// Proceso padre crea al primer hijo
pid_t primerHijo = fork();
if(primerHijo < 0){
        perror("Error creando primerHijo");
        return 1;
}else if(primerHijo == 0){
        // Proceso primer hijo
```

Imagen 4. Creación de primer hijo

3.2 Primer Hijo y Nieto

El primer hijo crea un pipe auxiliar para la comunicación entre el "nieto" y el primer hijo. Luego, crea al "nieto", que es el encargado de cargar el primer arreglo, calcular su suma, y enviar el resultado al padre y al primer hijo.

```
int pipefd_aux[2];  // Pipe auxiliar para comunicación entre nieto/segundo hijo y primer hijo
pipe(pipefd_aux);   // Se crea el pipe auxiliar

// Primer hijo crea al nieto
pid_t grandHijo = fork();

if(grandHijo < 0){
        perror("Error creando grandHijo");
        exit(1);
}else if(grandHijo == 0){
        // Proceso nieto
```

Imagen 5. Creación de nieto

El código del nieto es el siguiente:

```
int n1 = (int) atoi(argv[1]);
int *vA = (int *) malloc(n1*sizeof(int));     // Reserva espacio para el arreglo
cargarArreglo(argv[2], vA);                   // Carga datos desde el archivo
int suma_vA = sumaArreglo(vA, n1);            // Calcula la suma del arreglo

// Escribe la suma en el pipe principal hacia el padre
close(pipefd[0]);
write(pipefd[1], &suma_vA, sizeof(int));
close(pipefd[1]);

// También escribe la suma al pipe auxiliar para el primer hijo
close(pipefd_aux[0]);
write(pipefd_aux[1], &suma_vA, sizeof(int));
close(pipefd_aux[1]);

free(vA);
exit(0);
```

Imagen 6. Tarea de leer y sumar elementos del arreglo

3.3 Segundo Hijo

Después de que el nieto termine, el primer hijo crea al segundo hijo, que realiza la misma tarea, pero con el segundo archivo.

```
// Proceso primer hijo (después de crear al nieto)
wait(NULL); // Espera a que el nieto termine

// Primer hijo crea al segundo hijo
pid_t segundoHijo = fork();
if(segundoHijo < 0){
        perror("Error creando grandHijo");
        exit(1);
}else if(segundoHijo == 0){
        // Proceso segundo hijo

        int n2 = (int) atoi(argv[3]);
        int *vB = (int *) malloc(n2*sizeof(int));   // Reserva espacio para el arreglo
        cargarArreglo(argv[4], vB);                 // Carga datos desde el archivo
        int suma_vB = sumaArreglo(vB, n2);          // Calcula la suma del arreglo

        // Escribe la suma en el pipe principal hacia el padre
        close(pipefd[0]);
        write(pipefd[1], &suma_vB, sizeof(int));
        close(pipefd[1]);

        // También escribe la suma al pipe auxiliar para el primer hijo
        close(pipefd_aux[0]);
        write(pipefd_aux[1], &suma_vB, sizeof(int));
        close(pipefd_aux[1]);

        free(vB);
        exit(0);
}
```

Imagen 7. Tarea de leer y sumar elementos del arreglo

3.4 Lectura de Resultados y Suma Final

Una vez que los hijos han terminado, el primer hijo lee las sumas de los dos arreglos y calcula la suma total.

```
int suma_vA, suma_vB;
read(pipefd_aux[0], &suma_vA, sizeof(int));
read(pipefd_aux[0], &suma_vB, sizeof(int));
close(pipefd_aux[0]);

// Calcula la suma total
int sumaArreglos = suma_vA + suma_vB;

// Escribe la suma total en el pipe principal hacia el padre
write(pipefd[1], &sumaArreglos, sizeof(int));
close(pipefd[1]);

exit(0);
```

Imagen 8. Suma total de arreglos

4. Proceso Padre (Lectura Final)

El proceso padre espera a que el primer hijo termine, luego lee los resultados de las sumas de los arreglos y la suma total de los pipes.

```
// Proceso padre
wait(NULL); // Espera al primer hijo

int sumaArreglos, suma_vA, suma_vB;

// Lee la suma del nieto
read(pipefd[0], &suma_vA, sizeof(int));
printf("Suma del arregloA: %d\n", suma_vA);

// Lee la suma del segundo hijo
read(pipefd[0], &suma_vB, sizeof(int));
printf("Suma del arregloB: %d\n", suma_vB);

// Lee la suma total calculada por el primer hijo
read(pipefd[0], &sumaArreglos, sizeof(int));
printf("Suma de los arreglos: %d\n", sumaArreglos);

close(pipefd[0]);
close(pipefd[1]);
```

Imagen 9. Impresión de resultados

4. Pruebas de funcionamiento:

Se realizaron dos pruebas para verificar el funcionamiento del programa. La primera prueba utilizó los archivos proporcionados por el usuario, **p1.txt** y **p2.txt**, los cuales contenían los arreglos de números: 1 2 3 4 5 6 7 8 9 10 y 11 12 13 14 15 16 17 18 respectivamente. A continuación, se muestra el resultado de esta prueba:

```
estudiante@NGEN295:~/Downloads/TallerFork$ ./taller_procesos 10 p1.txt 8 p2.txt
Arreglo de p1.txt: 1 2 3 4 5 6 7 8 9 10

Arreglo de p2.txt: 11 12 13 14 15 16 17 18

Suma del arregloA: 55
Suma del arregloB: 116
Suma de los arreglos: 171
```

Imagen 9. Resultado de primera prueba

La segunda prueba utilizó los archivos **p3.txt** y **p4.txt**, con los siguientes arreglos: 5 10 15 20 25 30 y 40 35 30 25 20 15 10 5. El resultado de esta prueba es el siguiente:

```
estudiante@NGEN295:~/Downloads/TallerFork$ ./taller_procesos 6 p3.txt 8 p4.txt
Arreglo de p3.txt: 5 10 15 20 25 30

Arreglo de p4.txt: 40 35 30 25 20 15 10 5

Suma del arregloA: 105
Suma del arregloB: 180
Suma de los arreglos: 285
```

Imagen 11. Resultado de la segunda prueba

5. Conclusiones

- En el desarrollo del programa se logra comprender y aplicar de manera efectiva el funcionamiento de la función fork(), implementando una jerarquía de procesos compuesta por un proceso Padre, un Primer Hijo, un Grand Hijo y un Segundo Hijo.
- El diseño propuesto permite distribuir responsabilidades: el Grand Hijo se encargó de sumar los datos del primer archivo, el Segundo Hijo hizo lo mismo con el segundo archivo, y el Primer Hijo recopiló ambos resultados, los sumó y envió el total al proceso Padre.
- Se implementa la comunicación entre procesos mediante pipe(), garantizando un intercambio de información seguro y eficiente, sin necesidad de compartir memoria entre procesos. Esta comunicación se desarrolló en dos etapas: Entre nietos e hijo, y luego entre el hijo y el padre.
- Se realiza un uso adecuado de wait(), lo que permitió evitar la creación de procesos zombis, asegurando que los procesos hijo y nieto fueran correctamente recogidos por sus respectivos padres.
- El uso de memoria dinámica a través de malloc() fue esencial para adaptar la carga de arreglos de enteros desde archivos, y se garantizó su correcta liberación con free() al finalizar la ejecución de cada proceso.

- Las pruebas realizadas demostraron el correcto funcionamiento del sistema, permitiendo validar los resultados esperados en distintos conjuntos de datos. El programa respondió satisfactoriamente tanto en la lectura y carga de datos, como en la distribución de tareas y recopilación de resultados.

Referencias

CSL MTU. (s.f.). *Process Creation: fork()*. Michigan Technological University. Recuperado de https://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html

Linux Man Pages. (s.f.). *fork(2) – Linux manual page*. Recuperado de https://man7.org/linux/man-pages/man2/fork.2.html

Stack Overflow en Español. (2017). *¿Cómo funciona la función fork()?*. Recuperado de https://es.stackoverflow.com/questions/179414/como-funciona-la-funci%C3%B3n-fork

The Open Group. (s.f.). *fork - Create a new process*. Recuperado de https://pubs.opengroup.org/onlinepubs/009695199/functions/fork.html

IBM Documentation. (s.f.). *fork(): Create a new process*. Recuperado de https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-fork-create-new-process

GNU Project. (s.f.). *Creating a process*. GNU C Library. Recuperado de https://www.gnu.org/savannah-checkouts/gnu/libc/manual/2.34/html_node/Creating-a-Process.html