Programación Paralela y Concurrente (CI-0117) - Uso de Fork

Esteban Rodríguez Betancourt

2024-03

Introducción

En esta práctica vamos a explorar el uso de la llamada al sistema `fork`, para crear procesos hijos en un programa en C. La práctica puede ser realizada en cualquier sistema operativo tipo UNIX, como Linux o macOS.

La llamada `fork` crea un nuevo proceso hijo, que es una copia exacta del proceso padre. La única diferencia entre el proceso padre y el proceso hijo es el valor de retorno de la llamada a `fork`. En el proceso hijo, el valor de retorno es `0`, mientras que en el proceso padre, el valor de retorno es el PID (Process ID) del proceso hijo.

Es importante destacar que aunque el nuevo proceso es una copia exacta del proceso padre, los dos procesos tienen su propio espacio de memoria, y por lo tanto, no comparten variables ni recursos.

En esta práctica se crearán varios programas en C para ilustrar el uso de `fork` y cómo se comportan los procesos padre e hijo, así como posibles errores a la hora de programar usando procesos.

En esta práctica creen un archivo por cada paso que desarrollen. Por ejemplo, si desarrollan un programa que imprime "Hola Mundo" usando `fork`, el archivo se llamaría `practica-01-fork-01.c`, el archivo de la siguiente sección sería `practica-01-fork-02.c`.

> **Cuidado**: En esta práctica NO se añadieron algunas validaciones para simplificar el código. Por ejemplo, no se valida si la llamada a `fork` fue exitosa. En un programa real, SIEMPRE se deben añadir estas validaciones. Esto aplica también para las llamadas a `malloc`, `wait`, `snprintf`, entre otras.
>
> **Sobre la gestión de memoria**: En esta práctica se usa `malloc` para asignar memoria. Recuerden que "siempre" deben liberar la memoria asignada con `free`. En esta práctica no se libera la memoria para enfocarnos en el uso de `fork`. Recordemos que cuando un proceso termina, el sistema operativo libera toda la memoria asignada al proceso.

1. Hola Mundo

Para iniciar vamos a crear un programa muy simple que usa `fork` para crear un proceso hijo. El proceso hijo imprimirá un mensaje en la pantalla, y el proceso padre imprimirá otro mensaje.

```c
// practica-01-fork-01.c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Hello from the other side\n");
    } else {
        printf("Hello, it's me\n");
    }
```

```
    return 0;
}
```

Compilen el programa usando el comando `gcc -o practica-01-fork-01 practica-01-fork-01.c` y ejecútenlo con `./practica-01-fork-01`. Ejecuten el programa varias veces y observen el resultado. Reporten:

1. ¿Se imprimen en el orden esperado?
2. ¿Siempre se imprimen en el mismo orden?
3. ¿A sus compañeros les sucede lo mismo?

2. Introducir trabajo pesado

Usualmente usamos procesos diferentes para distribuir trabajo. De momento vamos a simular una tarea de cómputo pesada usando `sleep(uint seconds)`. Creen una copia del programa anterior y modifíquenla para que el proceso hijo espere 5 segundos antes de imprimir su mensaje. Compilen el programa y ejecútenlo varias veces. Reporten:

1. ¿Se imprimen en el orden esperado?
2. ¿Cómo imprime cada proceso?
3. ¿Se ejecuta el proceso hijo antes o después del proceso padre?

3. Esperar al hijo

En el programa anterior, el proceso padre no espera al proceso hijo antes de terminar. Copien el programa y modifíquenlo para que el proceso padre espere al proceso hijo antes de terminar. Para esto, usen la llamada al sistema `waitpid(pid_t pid, int *status, int options)`. Agreguen un `printf` al inicio del programa y otro al final del programa para indicar cuándo comienza y termina el proceso padre. Compilen el programa y ejecútenlo varias veces. Reporten:

1. ¿Se imprimen en el orden esperado?
2. ¿Cómo imprime cada proceso?
3. ¿Se ejecuta el proceso hijo antes o después del proceso padre?

4. Crear varios procesos

En el programa anterior, el proceso padre crea un solo proceso hijo. Copien el programa y modifíquenlo para que el proceso padre cree varios procesos hijos. Para esto, usen un ciclo `for` para crear varios procesos hijos. Agreguen un `printf` al inicio del programa y otro al final del programa para indicar cuándo comienza y termina el proceso padre. Su programa debe verse algo así:

```c
// practica-01-fork-04.c
#include <stdio.h>
#include <unistd.h>

int main(void) {
    printf("Main process\n")

    int n = 5;
    for (int i = 0; i < n; i++) {
        pid_t pid = fork();

        if (pid == 0) {
            printf("I'm child process number %d\n", i);
            return 0;
        }
    }

    printf("DONE\n");
```

```
    return 0;
}
```

Compilen el programa y ejecútenlo varias veces. Reporten:

1. ¿Cuántos procesos hijos se crean?
2. ¿En qué orden se crean los procesos hijos?
3. ¿En qué orden se ejecutan los procesos hijos? ¿Siempre es el mismo orden?
4. ¿Cómo sabe el proceso hijo cual es su número?

5. Esperar a todos los hijos

En el programa anterior, el proceso padre no espera a que todos los procesos hijos terminen antes de terminar. Copien el programa y modifíquenlo para que el proceso padre espere a que todos los procesos hijos terminen antes de terminar. Para esto, usen un ciclo `for` para esperar a cada proceso hijo. Usen la llamada al sistema `waitpid(pid_t pid, int *status, int options)` para esperar a un proceso hijo. Reporten

1. ¿Ahora se imprimen en el orden esperado?
2. ¿El proceso padre espera a que todos los procesos hijos terminen antes de terminar?

6. Copias de la memoria durante `fork`

En el siguiente ejercicio vamos a explorar cómo se comportan las variables en un proceso padre y un proceso hijo. Copien el siguiente programa y ejecútenlo varias veces. Reporten:

1. ¿Qué valor tiene la variable `msg` al final del programa?
2. ¿Por qué el valor de `msg` cambia o no?
3. ¿Porqué los hijos pueden imprimir el mensaje correcto?
4. Al final, qué valor tiene la variable `msg`? ¿Porqué?
5. ¿Qué pasó con la asignación de memoria de `msg` realizada en los hijos?

```c
// practica-01-fork-06.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
  printf("Process start\n");

  int total_forks = 10;
  char *msg = malloc(100);
  snprintf(msg, 100, "I'm the parent process!!!\n");
  printf("%s", msg);

  for (int i = 0; i < total_forks; i++) {
    pid_t pid = fork();

    if (pid == 0) {
      snprintf(msg, 100, "I'm child process number %d\n", i);
      printf("%s", msg);
      return 0;
    }
  }

  for(int i = 0; i < total_forks; i++) {
    wait(NULL);
  }
```

```
  printf("The program ends\n");
  printf("The final value of msg is: %s\n", msg);

  return 0;
}
```

7. ¿Y los punteros?

En el ejercicio anterior, vimos que cada proceso tiene una copia de su memoria. Esta copia se realiza al momento de llamar a `fork`. ¿Pero las direcciones de memoria son las mismas? Copien el siguiente programa y ejecútenlo varias veces. Reporten:

1. ¿La dirección de memoria de `msg` es la misma en el proceso padre y en los procesos hijos?
2. ¿Porqué?

```
// practica-01-fork-07.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
  printf("Process start\n");

  int total_forks = 10;
  char *msg = malloc(100);
  snprintf(msg, 100, "I'm the parent process!!!\n");
  printf("[Parent] The address of msg is: %p | Value is: %s\n", msg, msg);

  for (int i = 0; i < total_forks; i++) {
    pid_t pid = fork();

    if (pid == 0) {
      snprintf(msg, 100, "hello from child %d", i);
      printf("[Child %d] The address of msg is: %p | Value is: %s\n", i, msg, msg);
      return 0;
    }
  }

  for(int i = 0; i < total_forks; i++) {
    wait(NULL);
  }

  printf("[Parent] The program ends\n");
  printf("[Parent] The final value of msg is: %s\n", msg);

  return 0;
}
```

8. ¿Los procesos gastan mucha memoria?

En teoría, cada proceso tiene su propia memoria. Pero, ¿cuánta memoria se gasta al crear un proceso hijo? Copien el siguiente programa. Abran una instancia de System Monitor o Task Manager y ejecuten el programa. Reporten:

1. ¿Cuánta memoria gasta el programa (incluyendo cada proceso)?
2. ¿Estamos gastando 100GB de memoria o no?

```c
// practica-01-fork-08.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void) {
  printf("Process start\n");

  int total_forks = 100;
  char *msg = malloc(1024 * 1024 * 1024);
  snprintf(msg, 100, "I'm the parent process!!!\n");
  printf("[Parent] The address of msg is: %p | Value is: %s\n", msg, msg);

  for (int i = 0; i < total_forks; i++) {
    pid_t pid = fork();

    if (pid == 0) {
      snprintf(msg, 100, "hello from child %d", i);
      printf("[Child %d] The address of msg is: %p | Value is: %s\n", i, msg, msg);
      sleep(10);
      return 0;
    }
  }

  for(int i = 0; i < total_forks; i++) {
    wait(NULL);
  }

  printf("[Parent] The program ends\n");
  printf("[Parent] The final value of msg is: %s\n", msg);

  return 0;
}
```

9. ¿Y si realmente estamos usando esa memoria?

Los diseñadores de sistemas operativos saben que a los programadores les gusta pedir más memoria de la cuenta. Por eso, los sistemas operativos "hacen trampa" y no asignan memoria hasta que realmente la necesitan. Copien el programa anterior y hagan las siguientes modificaciones:

1. Reduzcan `total_forks` a la cantidad de gigabytes de memoria RAM LIBRES en su computadora.
2. **Asegúrese de haber hecho el paso anterior correctamente**. La computadora podría congelarse si no lo hace.
3. Después de hacer `malloc`, hagan un `memset` para llenar la memoria con un valor específico.
4. Dentro de cada hijo, hagan otro `memset` para cambiar el valor de la memoria. No usen 0.

La sintaxis de `memset` es `memset(void *ptr, int value, size_t num)`. `ptr` es el puntero a la memoria, `value` es el valor que se va a escribir en la memoria, y `num` es la cantidad de bytes que se van a escribir. Por ejemplo, `memset(msg, 0, 1024 * 1024 * 1024)` llena la memoria con ceros. Abra el System Monitor o Task Manager y ejecuten el programa. Reporten:

1. ¿Cuánta memoria gasta el programa (incluyendo cada proceso)?
2. ¿Ahora sí se gasta más memoria?
3. ¿Porqué?
4. ¿Colgó la computadora? ¿Porqué?