

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
Sistemas Operativos 1



“Llamas al sistema fork()”

Pedraza Celón Ian Yael

19/02/2020

CREACIÓN DE PROCESOS MEDIANTE FORK

En Unix, un proceso es creado mediante la llamada del sistema fork. El proceso que realiza la llamada se denomina proceso padre (parent process) y el proceso creado a partir de la llamada se denomina proceso hijo (child process).

La sintaxis de la llamada efectuada desde el proceso padre es:

valor=fork()

La llamada fork pero devuelve un valor distinto a los procesos padre e hijo: al proceso padre se le devuelve el PID del proceso hijo, y al proceso hijo se le devuelve el valor cero. Las acciones implicadas por la petición de un fork son realizadas por el núcleo (kernel) del S.O. Unix. Tales acciones son las siguientes:

1. asignación de un hueco en la tabla de procesos para el nuevo proceso (hijo).
2. asignación de un identificador único (PID) al proceso hijo.
3. copia de la imagen del proceso padre (con excepción de la memoria compartida).
4. asignación al proceso hijo del estado "preparado para ejecución".
5. dos valores de retorno de la función: al proceso padre se le entrega el PID del proceso hijo, y al proceso hijo se le entrega el valor cero.

INICIACIÓN DE PROGRAMAS MEDIANTE EXEC

La llamada exec produce la sustitución del programa invocador por el nuevo programa invocado. Mientras fork crea nuevos procesos, exec sustituye la imagen de memoria del proceso por otra nueva (sustituye todos los elementos del proceso: código del programa, datos, pila, montículo).

El PID del proceso es el mismo que antes de realizar la llamada exec, pero ahora, ejecuta otro programa. El proceso pasa a ejecutar el nuevo programa desde el inicio y la imagen de memoria del antiguo programa se pierde al verse sobrescrita. La imagen de memoria del antiguo programa se pierde para siempre, es decir, todo el código que escribamos posteriormente a la ejecución con éxito de la llamada exec, será inalcanzable.

La combinación de las llamas fork y exec es el mecanismo que ofrece UNIX para crear un nuevo proceso (fork) que ejecute un programa determinado (exec).

De las seis posibles llamadas tipo exec se usará en este apartado de la práctica la llamada execv, cuya sintaxis es:

int execv (const char *filename, char *const argv[]);

El seguimiento de ejecución del proceso, mediante la orden ps del sistema usada antes y después de la llamada execv, permite comprobar cómo se efectúa la sustitución de la imagen de memoria. El nuevo programa activado mantiene el mismo PID, identificador de proceso, así como otras propiedades asociadas al proceso, sin embargo, el tamaño de memoria

DESARROLLO DE LA PRÁCTICA

1. Codifique el siguiente código.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    pid_t rf;
    rf = fork();

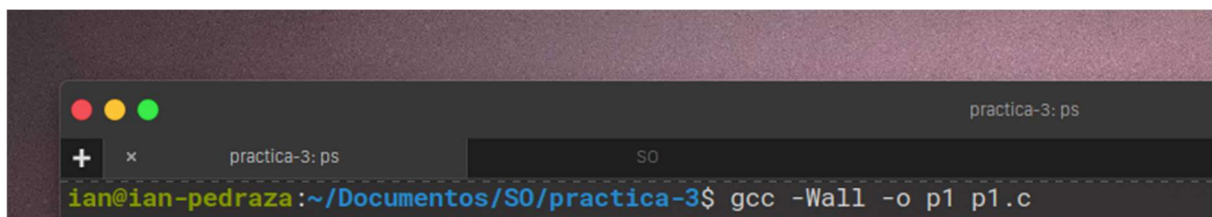
    switch (rf) {
        case -1:
            printf ("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf ("Soy el hijo, mi PID es %d y mi PPID es%d \n", getpid(), getppid());
            sleep (20);
            break;
        default:
            printf ("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n", getpid(), rf);
            sleep (30);
    }

    printf ("Final de ejecución de %d \n", getpid());

    exit (0);
}
```

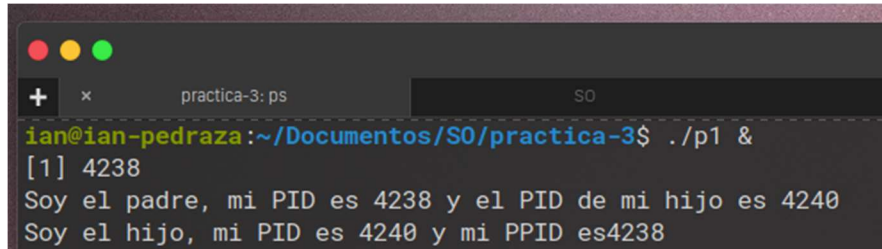
2. Compile el programa anterior mediante el mandato gcc.

\$ gcc -Wall -o p1 p1.c



3. Ejecute el programa forkprog en segundo plano (o background). Para ello, se debe añadir al nombre del programa el carácter & (ampersand).

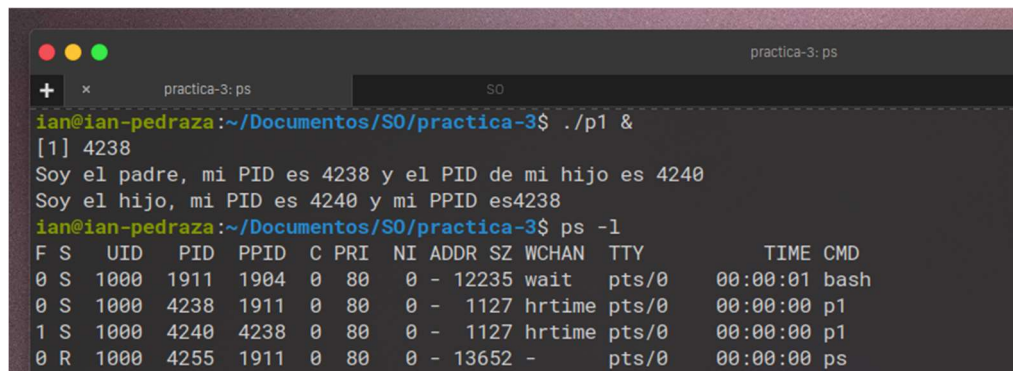
`$./forkprog &`



```
ian@ian-pedraza:~/Documentos/S0/practica-3$ ./p1 &
[1] 4238
Soy el padre, mi PID es 4238 y el PID de mi hijo es 4240
Soy el hijo, mi PID es 4240 y mi PPID es4238
```

4. Verifique su ejecución con la orden ps, que le permitirá comprobar su funcionamiento. Dicha orden ha de ser ejecutada antes de que finalice la ejecución del proceso.

`$ ps -l`



```
ian@ian-pedraza:~/Documentos/S0/practica-3$ ./p1 &
[1] 4238
Soy el padre, mi PID es 4238 y el PID de mi hijo es 4240
Soy el hijo, mi PID es 4240 y mi PPID es4238
ian@ian-pedraza:~/Documentos/S0/practica-3$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1911  1904  0  80   0 - 12235 wait  pts/0    00:00:01 bash
0 S  1000  4238  1911  0  80   0 - 1127 hrtime pts/0    00:00:00 p1
1 S  1000  4240  4238  0  80   0 - 1127 hrtime pts/0    00:00:00 p1
0 R  1000  4255  1911  0  80   0 - 13652 -      pts/0    00:00:00 ps
```

5. Observe los valores PID y PPID de cada proceso e identifique qué atributos son heredados entre padre e hijo y cuáles no.

Los atributos heredados son PID, UID, ADDR

CONTESTE LA SIGUIENTES PREGUNTAS

1. ¿Cuáles son los PID de los procesos padre e hijo?

PID padre: 4238

PID hijos: 4240, 4238

2. ¿Qué tamaño de memoria ocupan los procesos padre e hijo?

1127 b

3. ¿Qué realiza la función sleep? ¿Qué proceso concluye antes su ejecución?

Sleep duerme el proceso por un tiempo determinado, el primero en terminar es el hijo.

4. ¿Qué ocurre cuando la llamada al sistema fork devuelve un valor negativo?

Indica que no se pudo crear el proceso

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

Imprimir su PID

EJERICICIOS DE MODIFICACION:

Modifique el código del programa para asegurar que el proceso padre imprime su mensaje de presentación ("Soy el proceso...") antes que el hijo imprima el suyo.

Modifique el código fuente del programa declarando una variable entera llamada varfork 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. Anote el valor final de la variable varfork para el padre y para el hijo.

Proceso padre varfork=

Proceso hijo varfork=

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){
    pid_t rf;
    rf = fork();
    int i;
    int varforkP = 10;
    int varforkH = 10;

    switch (rf) {
        case -1:
            printf ("No he podido crear el proceso hijo \n");
            break;
        case 0:
            printf ("Soy el hijo, mi PID es %d y mi PPID es%d \n", getpid(), getppid());
            for(i = 0; i < 10; i++) varforkH += 1;
            sleep (20);
            break;
        default:
            printf ("Soy el padre, mi PID es %d y el PID de mi hijo es %d \n", getpid(), rf);
            for(i = 0; i < 10; i++) varforkP += 10;
            sleep (30);
    }

    printf ("Final de ejecución de %d \n", getpid());
    printf("Proceso padre varfork = %d\n", varforkP);
    printf("Proceso hijo varfork = %d\n", varforkH);
    exit (0);
}
```

```

ian@ian-pedraza:~/Documentos/S0/practica-3$ Final de ejecución de 4240
Proceso padre varfork = 10
Proceso hijo varfork = 20
ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1911  1904  0  80   0 - 12235 wait  pts/0    00:00:01 bash
0 S  1000  4238  1911  0  80   0 - 1127 hrtime pts/0    00:00:00 p1
1 Z  1000  4240  4238  0  80   0 -    0 -      pts/0    00:00:00 p1 <defunct>
0 R  1000  4275  1911  0  80   0 - 13652 -      pts/0    00:00:00 ps
ian@ian-pedraza:~/Documentos/S0/practica-3$ Final de ejecución de 4238
Proceso padre varfork = 110
Proceso hijo varfork = 10

```

En este apartado se va a practicar con dos programas (prog1 y prog2), cuyos archivos son: fuentes (prog1.c y prog2.c) se muestran a continuación:

PROG1

```

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int i;

    printf ("\nEjecutando el programa invocador (prog1). Sus argumentos son: \n");

    for ( i = 0; i < argc; i ++ ) printf ("argv[%d] : %s \n", i, argv[i]);

    sleep( 10 );
    strcpy (argv[0], "prog2");

    if (execv ("./prog2", argv) < 0) {
        printf ("Error en la invocacion a prog2 \n");
        exit (1);
    }

    exit(0);
}

```


PROG2

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main (int argc, char *argv[]) {
    int i;
    char a[2000000];

    printf ("Ejecutando el programa invocado (prog2). Sus argumentos son: \n");

    for ( i = 0; i < argc; i ++ ) printf ("argv[%d] : %s \n", i, argv[i]);

    sleep(10);
    exit (0);
}
```

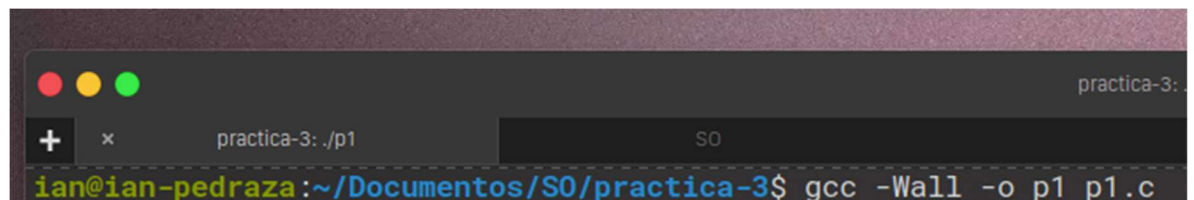
1. Compile los programas prog1.c y prog2.c mediante el mandato gcc.

Nota: En caso de prog2.c, la compilación ocasiona un aviso (warning), que no es significativo y puede ignorarse).

2. Ejecute el programa en background (añadiendo '&') para poder verificar su ejecución con la orden ps. La forma de invocar al programa prog1 es la siguiente:

`$ prog1 arg1 arg2 ... argN &`

`$ ps -l`



```
practica-3: ps
+ x practica-3: ps SO
ian@ian-pedraza:~/Documentos/S0/practica-3$ ./prog1 arg1 arg2 arg3 arg4 &
[1]+  Hecho          ./prog1
[1] 3988

Ejecutando el programa invocador (prog1). Sus argumentos son:
argv[0] : ./prog1
argv[1] : arg1
argv[2] : arg2
argv[3] : arg3
argv[4] : arg4
```

3. Para observar cuánta memoria ocupa cada programa, realice un “ps -l” una vez el proceso ha escrito en pantalla el mensaje correspondiente (hay 10 segundos de plazo antes de realizar el execv en prog1 y antes de terminar en prog2).

```
practica-3: ps
+ x practica-3: ps SO
ian@ian-pedraza:~/Documentos/S0/practica-3$ ./prog1 arg1 arg2 arg3 arg4 &
[1]+  Hecho          ./prog1
[1] 3988

Ejecutando el programa invocador (prog1). Sus argumentos son:
argv[0] : ./prog1
argv[1] : arg1
argv[2] : arg2
argv[3] : arg3
argv[4] : arg4
ian@ian-pedraza:~/Documentos/S0/practica-3$ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1911  1904  0  80   0 - 12235 wait  pts/0    00:00:00 bash
0 S  1000  3988  1911  0  80   0 - 1127 hrtim pts/0    00:00:00 prog1
0 R  1000  4004  1911  0  80   0 - 13652 -      pts/0    00:00:00 ps
ian@ian-pedraza:~/Documentos/S0/practica-3$ Ejecutando el programa invocado (prog2). Sus argumentos son:
argv[0] : prog2
argv[1] : arg1
argv[2] : arg2
argv[3] : arg3
argv[4] : arg4
ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1911  1904  0  80   0 - 12235 wait  pts/0    00:00:00 bash
0 S  1000  3988  1911  0  80   0 - 1585 hrtim pts/0    00:00:00 prog2
0 R  1000  4024  1911  0  80   0 - 13652 -      pts/0    00:00:00 ps
ian@ian-pedraza:~/Documentos/S0/practica-3$
```

CONTESTE LAS PREGUNTAS:

- a) Escriba el contenido de los elementos del vector argv que recibe prog1 y los que recibe prog2.

Prog1: [arg1, arg2, arg3, arg4]

Prog2: [arg1, arg2, arg3, arg4]

- b) ¿Qué PID tiene el proceso que ejecuta prog1?c? ¿Y el de prog2?c?
3988 para ambos programas

- c) ¿Qué tamaño de memoria ocupa el proceso, según ejecute prog1 o prog2?
1127 para prog1, y 21585 para prog2

Conclusión

Un proceso puede generar hijos que heredan una cantidad de atributos de su padre, pero dentro de ese mismo proceso podemos hacer la ejecución a otros procesos que están heredan otros atributos del proceso original, pero cambia completamente el programa, y el resto del programa original no se ejecuta.

Estas características nos pueden ayudar a modularizar nuestros procesos en otro programa, u ocupar otros que ya estén escritos en el sistema, dándonos así la capacidad de crear software más robusto y completo.