LABORATORIO DE SISTEMAS OPERATIVOS

PERÍODO ACADÉMICO: 2024 – A **EQUIPO:**

PROFESOR: Marco Sánchez PhD.

TIPO DE INSTRUMENTO: Guía de Laboratorio

TEMA: CREACIÓN DE PROCESOS

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

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. MARCO TEÓRICO

Todo programa en ejecución es un proceso. Por ejemplo:

```
#include <stdio.h>
int main(void)
{
   printf("Hola soy un proceso que imprime un mensaje\n");
}
```

- Una vez que se compile y ejecute dicho programa se volverá un proceso. Dicho proceso será creado en la memoria principal con los datos necesarios para su ejecución. Usando esos datos en memoria, el proceso finalmente será capaz de ejecutar las instruccionPS -Aes que componen el programa.
- Todos los procesos son manejados por el sistema operativo.
- Los procesos se identifican entre sí con un identificador único, o process ID (pid).
- El PID se puede consultar mediante la función getpid() de la librería unistd.

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

int main(void)
{
   printf("Hola soy un proceso y mi identificador es: %d\n", getpid());
}
```

Un proceso puede crear otro proceso independiente.

Fork
La función fork() crea un clon idéntico del proceso que la ejecutó.

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

int main(void)
{
   fork(); // Crea un clon de si mismo

   printf("Hola mi identificador de proceso es: %d\n", getpid());
}
```

La salida sería:

```
Hola mi identificador de proceso es: 4608
Hola mi identificador de proceso es: 4609
```

Generalmente no tiene sentido crear un proceso que haga lo mismo que el padre sino que realice una tarea distinta. Por medio de if ambos procesos comprobarán si son el proceso hijo o son el proceso padre.

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

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

   // instrucciones que tanto el padre como el hijo harán

   if (pid != 0)
   {
      // instrucciones que solo el proceso padre hará
   }
   else
   {
      // instrucciones que solo el proceso hijo hará
```

```
    }
}
```

Función `getppid()`

La función `getppid()` en un sistema operativo basado en Unix/Linux es una llamada al sistema que devuelve el PID (Process ID) del proceso padre del proceso que la llama.

¿Qué es un PID?

Un PID es un número único que identifica a cada proceso en el sistema operativo.

¿Qué es un proceso padre?

Un proceso padre es el que crea (o forkea) otro proceso. En Unix/Linux, cada proceso tiene un proceso padre, excepto el primer proceso que se ejecuta al iniciar el sistema, que generalmente es llamado `init` o `systemd`.

Usos de `getppid()`

La función `getppid()` puede ser utilizada por cualquier proceso en el sistema, no necesariamente tiene que haber sido creado mediante una llamada a `fork()`. La razón es que todos los procesos (excepto el primer proceso) tienen un proceso padre.

Función `fork()`

La llamada al sistema `fork()` se utiliza para crear un nuevo proceso en Unix/Linux. Cuando `fork()` se llama, el proceso actual (proceso padre) se duplica en un nuevo proceso (proceso hijo).

Resultados de `fork()`

- `-1`: Indica que hubo un error al intentar crear el proceso hijo.
- `0`: Indica que estás en el proceso hijo recién creado.
- `>0`: Indica que estás en el proceso padre y el valor es el PID del proceso hijo.

Ejemplo de Uso

Supongamos que tienes un programa en C que llama a `fork()` y luego a `getppid()`.

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

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

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

    if (pid == -1) {
        // Error al crear el proceso hijo
        perror("fork");
        return 1;
    } else if (pid == 0) {
        // Esto se ejecuta en el proceso hijo
        printf("Proceso hijo, PID del padre: %d\n", getppid());
    } else {
        // Esto se ejecuta en el proceso padre
        printf("Proceso padre, PID del hijo: %d\n", pid);
    }

    return 0;
}
```

Exec

La llamada `exec` en Unix/Linux se utiliza para reemplazar el proceso actual con un nuevo programa. Esto significa que el proceso que llama a `exec` dejará de ejecutar su código original y comenzará a ejecutar el nuevo programa especificado en la llamada `exec`.

Diferencia entre `fork` y `exec`

- **`fork()`**: Crea un nuevo proceso (hijo) que es una copia del proceso llamador (padre). El nuevo proceso tiene su propio espacio de memoria y una copia exacta del contexto de ejecución del proceso padre en el momento de la llamada.
- **`exec()`**: No crea un nuevo proceso. En lugar de eso, reemplaza la imagen de memoria del proceso actual con una nueva imagen de un programa diferente. El PID del proceso no cambia, pero el contenido del proceso (código, datos, pila) se reemplaza completamente.

Comportamiento de `exec`

- Cuando se llama a `exec`, el proceso actual comienza a ejecutar el nuevo programa desde el inicio.
- Toda la memoria ocupada por el programa original se pierde y es reemplazada por la del nuevo programa.

- Cualquier código que se haya escrito después de una llamada exitosa a `exec` no se ejecutará, ya que la imagen de memoria del proceso se ha reemplazado por completo.

Combinación de `fork` y `exec`

- **`fork()`**: Primero se llama a `fork()` para crear un nuevo proceso (hijo).
- **`exec()`**: Luego, el proceso hijo llama a `exec()` para reemplazar su imagen de memoria con la de un nuevo programa.

Esta combinación es la forma en que Unix/Linux crea un nuevo proceso que ejecuta un programa diferente.

Variantes de `exec`

Existen varias funciones de la familia `exec` que permiten diferentes formas de especificar el nuevo programa y sus argumentos:

- **`execl(const char *path, const char *arg, ...)`**:
  - Toma el camino al nuevo programa y una lista de argumentos.
  - Los argumentos se pasan como una lista de argumentos separados por comas.
- **`execv(const char* path, char* const argv[])`**:
  - Similar a `execl`, pero los argumentos se pasan como un array de cadenas.
- **`execve(const char* path, char* const argv[], char* const envp[])`**:
  - Similar a `execv`, pero también permite especificar un conjunto de variables de entorno.
- **`execvp(const char *file, char *const argv[])`**:
  - Similar a `execv`, pero busca el archivo ejecutable en los directorios especificados en la variable de entorno `PATH`.

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

a) **Identifique los valores PID y PPID de cada proceso.**
b) **¿Qué realiza la función sleep? ¿Qué proceso concluye antes de su ejecución?**
c) **¿Qué ocurre cuando la llamada al sistema fork devuelve un valor negativo?**
d) **¿Cuál es la primera instrucción que ejecuta el proceso hijo?**
e) **Modifique el código para que el proceso padre imprima su mensaje antes que el hijo.**

f) 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?

```c
#include <stdio.h>
#include <unistd.h>
int main ()
{
        printf ("Ejemplo de fork.\n");
        printf ("Inicio del proceso padre. PID=%d\n", getpid ());
        if (fork() == 0)
        {
                /* Proceso hijo */
                printf ("Inicio proceso hijo. PID=%d, PPID=%d\n", getpid (), getppid ());
                sleep (1);
        }
        else
        {
                /* Proceso padre */
                printf ("Continuación del padre. PID=%d\n", getpid ());
                sleep (1);
        }
        printf ("Fin del proceso %d\n", getpid ());
        return 0;
}
```

Figura 1. Script de ejemplo para el uso de Fork

3.1.2 Creación de procesos hijos

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

b) ¿Qué realiza la función wait?

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>

int main()
{
        int pid;
        int i;
        int cont1=0;
        int estado;

        pid = fork();

        switch(pid)
        {
                case -1: // Si pid es -1 quiere decir que ha habido un error
                        perror("No se ha podido crear el proceso hijo\n");
                        break;
                case 0: // Cuando pid es cero quiere decir que es el proceso hijo
                        for(i=1; i<=10; i++)
                                printf("Soy el hijo %d\n",i);
                        break;
                default: // Cuando es distinto de cero es el padre
                        for(i=1; i<=10; i++)
                                printf("Soy el padre %d\n",i);
                        // La función wait detiene el proceso padre y se queda esperando hasta
                        // que termine el hijo
                        wait(0);
                        printf("Mi proceso hijo ya ha terminado. \n");
                        break;
        }
}
```

Figura 2. Script de ejemplo que explica los valores que puede tomar la función fork( )

3.1.3 Creación de procesos hijos

a) **¿Las variables enteras i y j del proceso padre son las mismas que las del proceso hijo?**
b) **Modificar el código para que ambos procesos inicien con igual valor de la variable i, y además el proceso padre realice su cuenta de uno en uno y el proceso hijo de dos en dos?**

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
 int i;
 int j;
 pid_t rf;
 rf = fork( );
 switch (rf)
        {
        case -1:
                printf ("\nNo he podido crear el proceso hijo");
                break;
        case 0:
                i = 0;
                printf ("\nSoy el hijo, mi PID es %d y mi variable i  (inicialmente a %d) es par", getpid( ), i);
                for ( j = 0; j < 5; j ++ )
                {
                        i ++;
                        i ++;
                        printf ("\nSoy el hijo, mi variable i es %d", i);
                }
                break;
        default:
                i = 1;
                printf ("\nSoy el padre, mi PID es %d y mi variable i  (inicialmente a %d) es impar", getpid( ), i);
                for ( j = 0; j < 5; j ++ )
                {
                        i ++;
                        i ++;
                         printf ("\nSoy el padre, mi variable i es %d", i);
                }
        }
 printf ("\nFinal de ejecucion de %d \n", getpid());
 exit (0);
}
```

Figura 3. Script de ejemplo para la creación de procesos hijos

3.1.4 Creación de un proceso hijo sin espera.

a) **Ejecutar en segundo plano.**
b) **¿Cuál es el PPID del proceso hijo?**
c) **¿Cómo se denomina al tipo de proceso hijo?**

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int pid;
int main ( ) {
 pid = fork( );
 switch (pid)
        {
        case -1:
                printf ("\nNo he podido crear el proceso hijo");
                break;
        case 0:
                 printf ("\nSoy el hijo, mi PID es %d y mi PPID es %d", getpid( ), getppid( ));
                sleep (10);
                 break;
        default:
                 printf ("\nSoy el padre, mi PID es %d y el PID de mi  hijo es %d", getpid( ), pid);
}
 printf ("\nFinal de ejecucion de %d \n", getpid( ));
 exit (0);
}
```

Figura 4. Script de ejemplo para la creación de un proceso hijo sin espera (A)

3.1.5 Creación de un proceso hijo con espera.

a) **Ejecutar en segundo plano.**
b) **¿Cuál es el PPID del proceso hijo?**
c) **¿En qué se diferencia el resultado con el código 3.1.4?**
d) **Modifique el código para que el proceso padre imprima el mensaje de finalización de su ejecución 10 segundos más tarde que el proceso hijo.**

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>

int pid;
int main ( ) {
 pid = fork( );
 switch (pid)
        {
        case -1:
                printf ("\nNo he podido crear el proceso hijo");
                break;
        case 0:
                printf ("\nSoy el hijo, mi PID es %d y mi PPID es %d", getpid( ), getppid( ));
                sleep (10);
                break;
        default:
                printf ("\nSoy el padre, mi PID es %d y el PID de mi hijo es %d", getpid( ), pid);
                wait(0);
        }
 printf ("\nFinal de ejecución de %d \n", getpid( ));
 exit (0);
}
```

Figura 5. Script de ejemplo para la creación de un proceso hijo con espera (B)

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

a) **¿Qué proceso terminará primero? ¿Por qué?**
b) **Modificar el código para que aparezca al final en pantalla el número de procesos hijos que se ejecutaron.**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 10
#define NUM_PROC 3

void hijoHasAlgo(int numero);
int main ()
{
        int numero, i, pid, estado;
        for (numero=1; numero <=NUM_PROC; numero++)
        {
                pid = fork();
                switch(pid)
                {
                        case -1:
                                fprintf(stderr, "Error al crear el proceso\n");
                                break;
                        case 0:
                                hijoHasAlgo(numero);
                                break;
                        default:
                                printf("Ejecutando el padre\n");
                                wait(0);
                                printf("Mi hijo %d ha terminado. \n", numero);
                                exit(0);
                }
        }
}

void hijoHasAlgo (int numero)
{
        int i;
        printf ("Ejecutando el hijo %d: \n", numero);
        for (i=1; i<=MAX; i++)
                printf("%d\t", i);
        printf("\n");
}
```

Figura 6. Script de ejemplo para la creación de un proceso padre y tres procesos hijos

3.1.7 Ejecución concurrente de procesos

a) **¿Qué salida produce?**
b) **¿Cuál es la diferencia con el código 3.1.6?**
c) **Modificar el código para que aparezca al final en pantalla el número de procesos hijos que se ejecutaron.**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

#define MAX 10
#define NUM_PROC 3
void hijoHasAlgo(int numero);
int main ()
{
        int numero, pid, estado;
        for (numero=1; numero <=NUM_PROC; numero++)
        {
                pid = fork();
                switch(pid)
                {
                        case -1:
                                fprintf(stderr, "Error al crear el proceso\n");
                                break;
                        case 0:
                                hijoHasAlgo(numero);
                                break;
                }
                if(wait(0)>0)
                {
                printf("Ejecutando el padre\n");
                printf("Se ha ejecutado el %d proceso hijo\n", numero);
                }
                else
                exit(0);
        }
}

void hijoHasAlgo (int numero)
{
        int i;
        printf ("Ejecutando el hijo %d: \n", numero);
        for (i=1; i<=MAX; i++)
                printf("%d\t", i);
        printf("\n");
}
```

Figura 7. Script de ejemplo para observar la ejecución concurrente de procesos

3.2 USO DE EXEC

a) **¿Qué información se despliega al ejecutar el código?**
b) **¿Qué sucede si se cambia la función wait por sleep?**

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int ejecutarNuevoProc(char *programa, char*arg_list[]);
int main()
        {
        char *programa="ls";
        char *arg_list[]={"ls", "-l",NULL};
        //sleep(10);
        int pid = ejecutarNuevoProc(programa, arg_list);
        wait(0);
        fprintf(stdout, "el proceso hijo con PID %d se ha ejecutado\n", pid);
        return 0;
        }

int ejecutarNuevoProc(char *programa, char *arg_list[])
        {
        int pid;
        pid=fork();
        switch(pid)
                {
                case -1:
                        fprintf(stderr, "No se pudo crear el proceso");
                        exit(0);
                case 0:
                        execvp(programa, arg_list);
                //No se ejecuta, si el execvp se ejecuta correctamente
                fprintf(stderr, "error al ejecutar exec");
                        exit(0);
                default:
                        return pid;
                }
        }
```

Figura 8. Script de ejemplo del uso de la función exec( )

3.2.1 Creación de un proceso zombie

a) **Ejecutar el código en segundo plano**
b) **Ejecutar el comando top y verificar que existe un proceso zombie**
c) **¿Cómo solucionaría el problema?**

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
  int pid;
  pid = fork ();
  if (pid > 0)
        {
        printf("Soy el proceso padre y espero 60 segundos antes de terminar, de mi hijo no se nada.\n");
        sleep (60);
        }
  else {
    exit (0);
  }
  return 0;
}
```

Figura 9. Script de ejemplo para la creación de un proceso zombie

3.2.2 Uso de exec

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

3.2.3 Crear un proceso hijo y colocarlo dentro de un bucle infinito.

a) **El proceso padre se congelará durante 1 minuto.**
b) **Una vez que se descongele el proceso padre se deberá matar al proceso hijo.**

4 INFORME

Evidenciar la ejecución de todos los ejercicios y sus resultados.
Responder las preguntas de la sección 3.

5 CONCLUSIONES Y RECOMENDACIONES