


	<p>UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p>Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 1</p>

INFORME DE LABORATORIO

(formato estudiante)

INFORMACIÓN BÁSICA					
ASIGNATURA:	Sistemas Operativos				
TÍTULO DE LA PRÁCTICA:	<i>Programacion de procesos en C para Linux.</i>				
NÚMERO DE PRÁCTICA:	03	AÑO LECTIVO:	2023	NRO. SEMESTRE:	05
FECHA DE PRESENTACIÓN	02/06/2023	HORA DE PRESENTACIÓN	—		
INTEGRANTE (s): Yoset Cozco Mauri				NOTA:	
DOCENTE(s): <i>NIETO VALENCIA, RENE ALONSO</i>					

SOLUCIÓN Y RESULTADOS
<p>I. SOLUCIÓN DE EJERCICIOS/PROBLEMAS</p> <p>1.El siguiente código crea un proceso hijo, realice un seguimiento de la variable value y describa el por que tiene ese comportamiento.</p> <pre> #include <sys/types.h> #include <stdio.h> #include <unistd.h> int value = 5; int main() { pid_t pid; pid = fork(); if (pid == 0) { /* child process */ value += 15; return 0; } else if (pid > 0) { /* parent process */ wait(NULL); printf("PARENT: value = %d", value); /*LINE A*/ return 0; } }</pre> <p>1.- El proceso padre se ejecuta y tiene value = 5.</p>

2.- El proceso padre crea un proceso hijo utilizando la función `fork()`.

En el proceso hijo, se incrementa el valor de `value` en 15 unidades, lo que resulta en
3.- `value = 20`.

4.- El proceso hijo retorna (finaliza) con un valor de 0.

5.- El proceso padre espera a que el proceso hijo termine utilizando la función `wait(NULL)`.

6.- Después de que el proceso hijo finaliza, el proceso padre imprime el valor actual de `value`, que en este caso es 5 (ya que cada proceso tiene su propia copia de las variables)

Se incluyo al código `#include <sys/wait.h>`

2. En el siguiente código, detalle que parte del código es ejecutada por el proceso padre y que porción del código es ejecutada por el proceso hijo. Describa la actividad de cada uno.

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

int main(void)
{
    pid_t pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0)
    { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0)
    { /* child process */
        execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete \n");
    }
    return 0;
}
```

Analisis:

Proceso padre:

Ejecuta la línea `pid = fork();` para crear un proceso hijo.

- Verifica si ocurrió un error al crear el proceso hijo. En caso de error, imprime "Fork Failed" y retorna 1.
- Si no hay errores, continúa con la siguiente instrucción.
- Si el valor de `pid` es mayor que 0, significa que se está ejecutando en el proceso padre.
- Ejecuta la línea `wait(NULL);` para esperar a que el proceso hijo complete su ejecución.
- Después de que el proceso hijo ha terminado, imprime "Child Complete" utilizando `printf("Child Complete \n");`.
- Retorna 0 para indicar que el proceso padre ha finalizado.

Proceso hijo:

- Si el valor de `pid` es igual a 0, significa que se está ejecutando en el proceso hijo.
- En el proceso hijo, se ejecuta la línea `execlp("/bin/ls", "ls", NULL);` para reemplazar la imagen del proceso hijo con el comando "ls".
- Después de ejecutar el comando "ls", el proceso hijo finaliza.

3. Analice el siguiente código, explique cómo se da el flujo del código desde el proceso padre y como procede con los procesos hijos.

```
#include <stdio.h>
int main(void)
{
    int pid;
    printf("Hasta aqui hay un unico proceso...\n");
    printf("Primera llamada a fork...\n");
    /* Creamos un nuevo proceso. */
    pid = fork();
    if (pid == 0) {
        printf("HIJO1: Hola, yo soy el primer hijo...\n");
        printf("HIJO1: Voy a pararme durante 20 seg. y luego terminare\n");
        sleep(20);
    }
    else if (pid > 0) {
        printf("PADRE: Hola, soy el padre. El pid de mi hijo es: %d\n", pid);
        /* Creamos un nuevo proceso. */
        pid = fork();
        if (pid == 0) {
            printf("HIJO2: Hola, soy el segundo hijo...\n");
            printf("HIJO2: El segundo hijo va a ejecutar la orden 'ls'...\n");
            execlp("ls", "ls", NULL);
            printf("HIJO2: Si ve este mensaje, el execlp no funciono...\n");
        }
        else if (pid > 0) {
            printf("PADRE: Hola otra vez. Pid de mi segundo hijo: %d\n", pid);
            printf("PADRE: Voy a esperar a que terminen mis hijos...\n");
            printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));
        }
    }
}
```

```
        printf("PADRE: Ha terminado mi hijo %d\n", wait(NULL));  
    }  
    else  
        printf("Ha habido algun error al llamar por 2a vez al fork\n"); }  
else  
    printf("Ha habido algun error al llamar a fork\n");  
}
```

Flujo de codigo proceso padre-hijo:

Flujo del proceso padre:

- Imprime el mensaje "Hasta aquí hay un único proceso...".
- Imprime el mensaje "Primera llamada a fork...".
- Crea un nuevo proceso hijo utilizando fork().
- Si la llamada a fork() es exitosa, se ejecuta el código dentro del if.
- Imprime el mensaje de saludo como "PADRE: Hola, soy el padre. El pid de mi hijo es: <pid>".
- Realiza una segunda llamada a fork() para crear un segundo proceso hijo.
- Si la llamada a fork() en la segunda vez es exitosa, se ejecuta el código dentro del segundo if.
- Imprime el mensaje de saludo como "PADRE: Hola otra vez. Pid de mi segundo hijo: <pid>".
- Espera a que ambos hijos terminen utilizando wait(NULL).
- Imprime el mensaje "PADRE: Ha terminado mi hijo <pid>" para cada uno de los hijos.

Flujo de los procesos hijos:

- Si el valor de retorno de fork() es 0, significa que se está ejecutando en un proceso hijo.
- En el primer hijo:
 - Imprime el mensaje de saludo como "HIJO1: Hola, yo soy el primer hijo...".
 - Imprime el mensaje "HIJO1: Voy a pararme durante 20 seg. y luego terminar...".
 - Espera durante 20 segundos utilizando sleep(20).
 - Termina la ejecución.
- En el segundo hijo:
 - Imprime el mensaje de saludo como "HIJO2: Hola, soy el segundo hijo...".
 - Imprime el mensaje "HIJO2: El segundo hijo va a ejecutar la orden 'ls'...".
 - Ejecuta el comando "ls" utilizando execlp("ls", "ls", NULL).
 - Si el comando execlp() no tiene éxito, imprime el mensaje "HIJO2: Si ve este mensaje, el execlp no funcionó...".
 - Termina la ejecución.

4. Analice el siguiente código. ¿Cuántos procesos se generan?

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    /* fork a child process */
    fork();
    /* fork another child process */
    fork();
    /* and fork another */
    fork(); return 0;
}
```

12 procesos generados

5. Los siguientes códigos muestran un ejemplo del problema del productor-consumidor, donde un proceso se encarga de generar información, mientras que el segundo lo lee de la memoria. Ambos códigos utilizan funciones del API POSIX con memoria compartida.

Analice el código y detalle las funciones que permiten compartir memoria entre procesos. (Para poder compilar el código, agregue -lrt a gcc, y ejecute los códigos: "./productor && ./consumidor")

```
//Producer
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* strings written to shared memory */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char* ptr;
    /* create the shared memory object */
    fd = shm_open(name, O_CREAT | O_RDWR, 0766);
```

```
/* configure the size of the shared memory object */
ftruncate(fd, SIZE);
/* memory map the shared memory object */
ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, fd, 0);
/* write to the shared memory object */

sprintf(ptr, "%s", message_0);
ptr += strlen(message_0);
sprintf(ptr, "%s", message_1);
ptr += strlen(message_1);
return 0;
}

//Consumer
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* shared memory file descriptor */
    int fd;
    /* pointer to shared memory object */
    char* ptr;
    /* open the shared memory object */
    fd = shm_open(name, O_RDONLY, 0766);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, fd, 0);
    /* read from the shared memory object */
    printf("%s", (char*)ptr);
    /* remove the shared memory object */
    shm_unlink(name); return 0;
}
```

```
• tty3@tty3:~/Desktop/ns23/nsa_2023/so_lab/lab03$ gcc -o productor productor.c -lrt
• tty3@tty3:~/Desktop/ns23/nsa_2023/so_lab/lab03$ gcc -o consumidor consumidor.c -lrt
• tty3@tty3:~/Desktop/ns23/nsa_2023/so_lab/lab03$ ./productor && ./consumidor
o HelloWorld! tty3@tty3:~/Desktop/ns23/nsa_2023/so_lab/lab03$
```

Las funciones clave que permiten compartir memoria entre procesos son:

-
- `shm_open()`: Se utiliza para crear o abrir un objeto de memoria compartida.
- `ftruncate()`: Establece el tamaño del objeto de memoria compartida.
- `mmap()`: Mapea el objeto de memoria compartida en la memoria del proceso.
- `sprintf()`: Escribe datos en el objeto de memoria compartida.
- `shm_unlink()`: Elimina el objeto de memoria compartida cuando ya no es necesario.

Estas funciones se utilizan para compartir y comunicar datos entre los procesos productor y consumidor a través de la memoria compartida.

II. SOLUCIÓN DEL CUESTIONARIO

- ¿Cuál es la principal característica de crear un proceso utilizando la función **FORK**?
La principal característica de crear un proceso utilizando la función `fork()` es que se crea una copia exacta del proceso padre, incluyendo el código, los datos y el estado del programa en ese momento. A partir de ese punto, los procesos padre e hijo pueden ejecutar diferentes secciones de código de forma independiente.
- ¿Cuántos procesos **FORK** se pueden crear de forma secuencial? ¿Existe algún límite establecido por el sistema operativo?
En teoría, se pueden crear un número ilimitado de procesos `fork()` de forma secuencial. Sin embargo, en la práctica, el número de procesos que se pueden crear puede estar limitado por recursos como la memoria y la capacidad del sistema operativo para manejar tantos procesos activos.
- ¿Qué trabajo realiza la función **EXECLP**? Explique utilizando un ejemplo de utilización del comando
La función `execlp()` se utiliza para reemplazar el programa actual de un proceso con un nuevo programa. Esta función carga y ejecuta un nuevo programa, y termina la ejecución del proceso actual. El nuevo

programa se ejecuta en el mismo espacio de memoria del proceso original. Un ejemplo de uso de `execlp()` sería ejecutar el comando "ls" para listar el contenido de un directorio:

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main() {
5      // Se ejecuta el comando "ls" con sus argumentos
6      execlp("ls", "ls", "-l", NULL);
7
8      // Esta línea no se ejecutará, ya que el proceso se reemplaza por "ls"
9      printf("Esta línea no se imprimirá\n");
10
11     return 0;
12 }
```

III. CONCLUSIONES

En resumen, al aprender sobre la programación de procesos en sistemas operativos, he adquirido los siguientes conocimientos:

La función `fork()` es utilizada para crear nuevos procesos, generando copias exactas del proceso padre. Los procesos padre e hijo tienen flujos de ejecución independientes y pueden comunicarse a través de técnicas como la memoria compartida.

La memoria compartida permite a los procesos compartir datos de manera eficiente y sincronizada.

La función `execlp()` se utiliza para ejecutar programas externos y reemplazar el programa actual del proceso.

Es fundamental comprender los conceptos básicos de la programación de procesos y las funciones relacionadas para desarrollar aplicaciones eficientes y efectivas en sistemas operativos.

Estos conocimientos me permiten entender cómo se crean y gestionan los procesos, cómo se comunican entre sí y cómo se ejecutan programas externos. Con esta comprensión, puedo desarrollar aplicaciones más complejas y aprovechar las capacidades del sistema operativo de manera más efectiva.

RETROALIMENTACIÓN GENERAL



UNIVERSIDAD NACIONAL DE SAN AGUSTIN
FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA



Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación

Aprobación: 2022/03/01

Código: GUIA-PRLE-001

Página: 9

REFERENCIAS Y BIBLIOGRAFÍA