

## Laboratorio 5: Process API

**Ejercicio 1:** Escriba un **script de shell** que muestre un menú de opciones y que realice las operaciones indicadas en el menú: [C]opiar un archivo, [E]liminar un archivo y [S]alir del script. El script no debe de terminar si no se elige la opción respectiva. Para elegir la opción debe ingresarse letras mayúsculas o minúsculas. Un archivo solo puede copiarse o eliminarse si este existe, de lo contrario mostrar un mensaje adecuado. La invocación del shell script debe ser: script01.

- **Propósito del ejercicio:** Crear un script de shell, en este script se debe mostrar un menú para copiar, pegar o eliminar algún archivo que se encuentre en el mismo directorio, nos pide mostrar un error si el usuario coloca parámetros incorrectos y que el script siga ejecutando hasta que el usuario decida salir del programa.
- **Lógica de solución:**

```
#!/bin/bash
while true #Iniciar un bucle infinito
do
    echo "[C]opiar un archivo" #Mostrar la opción para copiar un archivo.
    echo "[E]liminar un archivo" #Mostrar la opción para eliminar un archivo.
    echo "[S]alir del script" #Mostrar la opción para salir del script.
    read -p "Elige una opción: " opcion #Pedir al usuario que ingrese una opción y se guardará en la variable
    'opcion'.

    case $opcion in #Usar una estructura case para evaluar la opción ingresada
        [cC]) #Lógica para la opción de copiar.
            read -p "Ingresa el nombre del archivo a copiar: " origen
            read -p "Ingresa el nombre de la copia: " destino
            if [ -f "$origen" ]; then #verifica si el archivo de origen existe.
                cp "$origen" "$destino" #se utiliza el comando 'cp' para copiarlo.
                echo "Archivo '$origen' copiado a '$destino' exitosamente."
            else
                echo "Error: El archivo '$origen' no existe."
            fi
            ;; #El doble punto y coma finaliza este bloque de 'case'.
        [eE]) #Lógica para la opción de eliminar.
            read -p "Ingresa el nombre del archivo a eliminar: " a_eliminar
            if [ -f "$a_eliminar" ]; then
                rm "$a_eliminar"
                echo "Archivo '$a_eliminar' eliminado exitosamente."
            else
                echo "Error: El archivo '$a_eliminar' no existe."
            fi
            ;;
        [sS]) #Lógica para la opción de salir.
            echo "Saliendo del script. ¡Adiós!"
            exit 0 #finaliza la ejecución del script. 0 es el código de éxito.
            ;;
        *) #Lógica para cualquier otra opción que no sea válida
            echo "Opción no válida. Por favor, elige una de las opciones del menú."
            ;;
    esac
done
```

```

Laboratorio5 [master] $ ./script01
[C]opiar un archivo
[E]liminar un archivo
[S]alir del script
Elige una opción: c
Ingresa el nombre del archivo a copiar: text.txt
Ingresa el nombre de la copia: noexiste.txt
Error: El archivo 'text.txt' no existe.

[C]opiar un archivo
[E]liminar un archivo
[S]alir del script
Elige una opción: s
Saliendo del script. ¡Adiós!

```

- **Anomalías:** Ninguna.

**Ejercicio 2:** Modifique el ejemplo `fork.c` para que el hijo a su vez cree otro proceso hijo (nieto), y que el inicio ejecute el script desarrollado en el ejercicio anterior que mostraba un menú de opciones para copiar y borrar un archivo determinado.

- **Propósito del ejercicio:** Crear una jerarquía de procesos, en este caso, hacer una generación de jerarquía de 3 procesos y en el tercer proceso tiene como tarea ejecutar el script que hicimos anteriormente.
- **Lógica de solución:**

```

#include <stdio.h> //Biblioteca estándar de entrada/salida
#include <sys/types.h> //Sirve para el pid_t.
#include <unistd.h> //Biblioteca para llamadas al sistema POSIX, como fork() y execvp().
#include <sys/wait.h> //Para la función wait().
#include <stdlib.h> //Para la función exit().

int main() {
    pid_t pid_hijo; //Declara una variable para almacenar el id del proceso hijo.
    printf("El pid del programa principal es %d\n", (int) getpid());
    pid_hijo = fork(); //Crea un nuevo proceso donde el valor de retorno es 0 para el hijo y el pid del
    hijo en el padre.

    if(pid_hijo != 0) {
        printf("Este es el proceso padre con ID %d\n", (int) getpid());
        printf("El ID del hijo es %d\n", (int) pid_hijo);
        wait(NULL); //El proceso padre espera a que su hijo termine para evitar un proceso zombi.
    } else {
        pid_t pid_nieto; //Declara una variable para el id del proceso nieto.
        printf("Este es el proceso hijo con ID %d\n", (int) getpid());
        pid_nieto = fork();

        if(pid_nieto != 0) {
            wait(NULL); // El proceso hijo espera a que el nieto termine para evitar un zombi.
        }
    }
}

```

```

} else {
    printf("Este es el proceso nieto, con ID %d\n", (int) getpid());
    printf("El nieto va a ejecutar el script del Ejercicio 1...\n");
    execlp("./script01", "script01", NULL);
    //execlp() reemplaza el código del nieto con el del script.
    //El primer argumento es el nombre del programa.
    //Los siguientes argumentos son los argumentos de línea de comandos para el nuevo
programa, que se pasan como cadenas separadas y el último argumento debe ser siempre NULL.
    perror("execlp falló"); //Es por si falla, se dirige a esta línea, sino, nunca se va a esta línea
    exit(1); //Sale con un código de error.
}
}
return 0;
}

```

```

└─ Laboratorio5 [master] $ ./fork
El pid del programa principal es 20261
Este es el proceso padre con ID 20261
El ID del hijo es 20262
Este es el proceso hijo con ID 20262
Este es el proceso nieto, con ID 20263
El nieto va a ejecutar el script del Ejercicio 1...
[C]opiar un archivo
[E]liminar un archivo
[S]alir del script
Elige una opción: e
Ingresa el nombre del archivo a eliminar: noexiste.txt
Error: El archivo 'noexiste.txt' no existe.

[C]opiar un archivo
[E]liminar un archivo
[S]alir del script
Elige una opción: s
Saliedo del script. ¡Adiós!

```

- **Anomalías:** Ninguna.

**Ejercicio 3:** Implementar el grafo de procedencia de la figura 1 utilizando las llamadas al sistema **fork** y **wait** (no utilizar **waitpid**). El grupo de sentencias a ejecutar en cada nodo del grafo se simularán mediante la sentencia **printf("cadena")**, donde **"cadena"** es la cadena de caracteres que contiene cada nodo de la figura. La frase deberá aparecer en una única línea.

- **Propósito de ejercicio:** Tenemos que simular un grafo de procedencia de procesos usando **fork()** para crear esos procesos y el **wait()** para controlar el orden en que termina cada uno, cada nodo tendrá una palabra del gráfico 1 y el objetivo es hacer que el proceso hijo termine antes que el padre continúe.
- **Lógica de solución:**

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

```

#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
// Declaración de funciones
void hola();
void buenos();
void dias();
void tenga();
void y();
void usted();
void hasta();
void malos();
void luego();
void lucas();

//Manejar la lógica de fork y wait donde recibe la palabra que se va a imprimir y una lista de
funciones de los nodos hijos.
void nodo(const char* palabra, void (*hijos[])()) {
    printf("%s\n", palabra); //Imprime la palabra del nodo actual.
    pid_t pid; //Declara una variable para el id del proceso hijo.
    int c = 0; //Contador para el número de hijos hay en la lista.

    //Cuenta cuántos nodo hijos hay
    while (hijos[c] != NULL) {
        c++;
    }

    //Crea un proceso hijo para cada nodo en la lista en cada bucle.
    for (int i = 0; i < c; i++) {
        pid = fork(); //Aquí crea un nuevo proceso hijo.
        if (pid == 0) {
            hijos[i](); //El hijo llama a la función del siguiente nodo en el grafo.
            exit(0); //El hijo termina su ejecución después de llamar a la función.
        }
    }

    // El padre espera a todos sus hijos.
    for (int i = 0; i < c; i++) {
        wait(NULL);
    }
}

// Implementación de cada nodo.
void hola() {
    void (*hijos[])() = {buenos, dias, tenga, NULL}; //Declaramos un array de punteros a funciones para
los nodos hijos de "Hola".
    nodo("Hola", hijos); //Llama a la función auxiliar para ejecutar la lógica del nodo "Hola" y sus
hijos.
}

void buenos() {
    void (*hijos[])() = {y, usted, NULL};
    nodo("Buenos", hijos);
}

void dias() {
    void (*hijos[])() = {usted, NULL};
    nodo("dias", hijos);
}

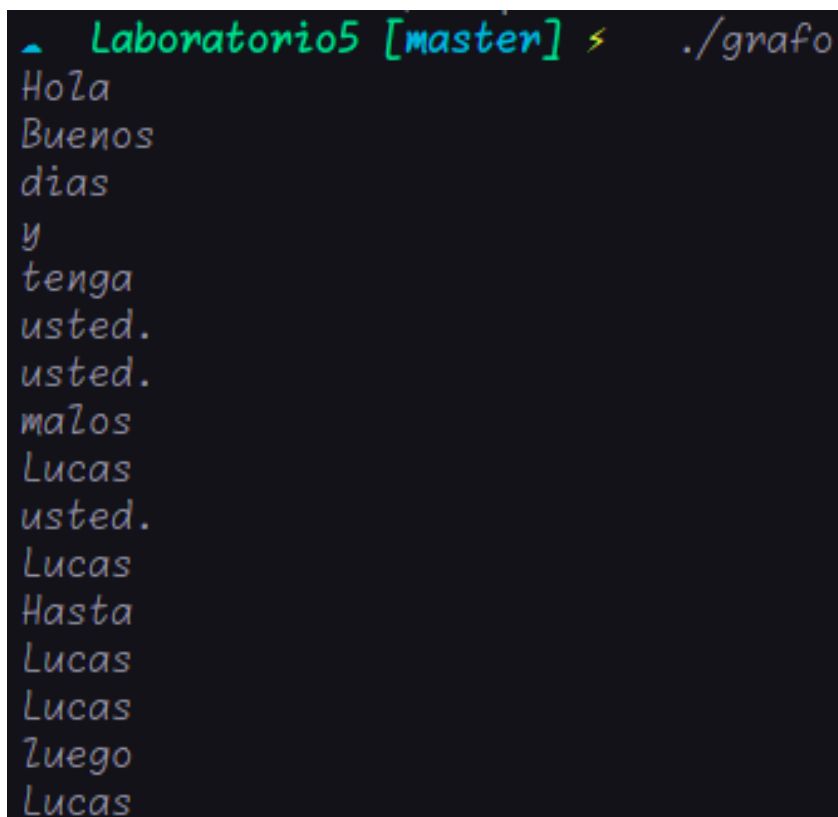
void tenga() {
    void (*hijos[])() = {usted, hasta, NULL};
    nodo("tenga", hijos);
}

```

```

}
void y() {
    void (*hijos[])() = {malos, NULL};
    nodo("y", hijos);
}
void usted() {
    void (*hijos[])() = {lucas, NULL};
    nodo("usted.", hijos);
}
void hasta() {
    void (*hijos[])() = {luego, NULL};
    nodo("Hasta", hijos);
}
void malos() {
    void (*hijos[])() = {lucas, NULL};
    nodo("malos", hijos);
}
void luego() {
    void (*hijos[])() = {lucas, NULL};
    nodo("luego", hijos);
}
void lucas() {
    void (*hijos[])() = {NULL}; // "Lucas" es una hoja del grafo, no tiene hijos.
    nodo("Lucas", hijos);
}
int main() {
    hola();
    return 0;
}

```



```

Laboratorio5 [master] ./grafo
Hola
Buenos
dias
y
tenga
usted.
usted.
maños
Lucas
usted.
Lucas
Hasta
Lucas
Lucas
luego
Lucas

```

- Anomalías:** Aquí si tiene anomalías, como lo es la repetición de palabras y se ve en las palabras “usted.” y “Lucas”, una posible causa creo que se debe a cómo funciona la función fork(), porque cuando un proceso llama a fork(), el SO crea una copia exacta del proceso original y cuando el proceso “Buenos”, “días” y “tenga” están a punto de crear un hijo para el proceso “usted.”, los tres procesos creo que imprimen el mismo procesos “usted.” y lo mismo ocurre con el proceso “Lucas”.

**Ejercicio 4:** Repetir el ejercicio anterior utilizando un proceso auxiliar **imprimir** cuyo código será ejecutado por los procesos hijo mediante la llamada al sistema `execl`. Dicho proceso deberá imprimir en pantalla, mediante la sentencia `printf("cadena")`, la(s) cadena(s) de caracteres que reciba como argumento(s).

- **Propósito del ejercicio:** Nos piden implementar el mismo grafo del ejercicio 3 solo que tenemos que utilizar una diferente estrategia que muestre la distinción entre `fork()` y `exec()`, en lugar de que cada proceso hijo tenga su propia lógica para crear y esperar a otros hijos, ahora cada hijo usará la función `exec()` para reemplazar el código con un programa auxiliar llamado `imprimir`.
- **Lógica de solución:**

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

```
int main(int argc, char *argv[]) {
    //La función main recibe dos argumentos: argc: el número de argumentos de línea de comandos y
    argv: un array de cadenas (char*) que contiene los argumentos.

    // El bucle for comienza en 1 porque argv[0] siempre es el nombre del programa
    for (int i = 1; i < argc; i++) {
        printf("%s ", argv[i]); //Imprime cada argumento que se le pasa, seguido de un espacio.
    }
    printf("\n");
    return 0;
}
```

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

```
int main() {
    pid_t pid; //Variable para almacenar el id del proceso hijo.
    printf("Hola\n");

    pid = fork(); //Proceso del nodo "Buenos"
    if (pid == 0) {
        pid = fork(); //En el hijo, se forkea para crear la rama de "y" y "usted.".
        if (pid == 0) {
            pid = fork(); //En el hijo de "Buenos", se forkea para crear "y" y "usted.".
            if (pid == 0) {
                pid = fork(); //En el hijo de "y", se forkea para crear "malos".
                if (pid == 0) {
                    pid = fork(); //En el hijo de "malos", se forkea para crear "Lucas".
                    if (pid == 0) {
                        execl("./imprimir", "imprimir", "Lucas", NULL); //El nieto se convierte en el programa
                        'imprimir' para imprimir "Lucas".
                    } else {
                        wait(NULL); //El padre de "Lucas" espera a que este termine y luego sale.
                        exit(0);
                    }
                } else {
                    wait(NULL); //El padre de "malos" espera a su hijo y luego se convierte en 'imprimir'.
                    execl("./imprimir", "imprimir", "y", "malos", NULL);
                }
            } else {
                wait(NULL); //El padre de "y" espera a su hijo y luego se convierte en 'imprimir'.
                execl("./imprimir", "imprimir", "usted.", "Lucas", NULL);
            }
        } else {
            wait(NULL); //El padre de "usted." espera a su hijo y luego se convierte en 'imprimir'.
        }
    }
}
```

```

    execl("./imprimir", "imprimir", "Buenos", "y", "usted.", NULL);
}
} else {
    pid = fork(); //En el padre, se forkea para crear la rama de "dias".
    if (pid == 0) {
        pid = fork(); //En el hijo de "dias", se forkea para crear "usted.".
        if (pid == 0) {
            execl("./imprimir", "imprimir", "usted.", "Lucas", NULL); //El hijo se convierte en el
programa 'imprimir' para imprimir "usted.".
        } else {
            wait(NULL); //El padre de "usted." espera a su hijo y luego se convierte en 'imprimir'.
            execl("./imprimir", "imprimir", "dias", "usted.", NULL);
        }
    } else {
        pid = fork(); //En el padre, se forkea para crear la rama de "tenga".
        if (pid == 0) {
            pid = fork(); //En el hijo de "tenga", se forkea para crear "usted." y "Hasta".
            if (pid == 0) {
                execl("./imprimir", "imprimir", "usted.", "Lucas", NULL); //El hijo se convierte en
'imprimir' para imprimir "usted.".
            } else {
                wait(NULL); //El padre de "usted." espera a su hijo y luego se convierte en 'imprimir'.
                execl("./imprimir", "imprimir", "Hasta", "luego", NULL);
            }
        } else {
            wait(NULL); //El padre de "Hasta" espera a su hijo y luego se convierte en 'imprimir'.
            execl("./imprimir", "imprimir", "tenga", "usted.", "Hasta", NULL);
        }
    }
}
}

// El proceso principal espera a que sus tres hijos ("Buenos", "dias", "tenga") terminen.
wait(NULL);
wait(NULL);
wait(NULL);

return 0;
}

```

```

Laboratorio5 [master] ⚡ ./grafo_exec
Hola
usted. Lucas
usted. Lucas
Lucas
dias usted.
Hasta luego
y malos
tenga usted. Hasta
usted. Lucas

```

- **Anomalías:** Aquí, a diferencia del ejercicio 3, no duplica el código sino que lo reemplaza y es por eso que no hay palabras repetidas lo que evita que varios procesos repitan la misma ejecución de impresión.