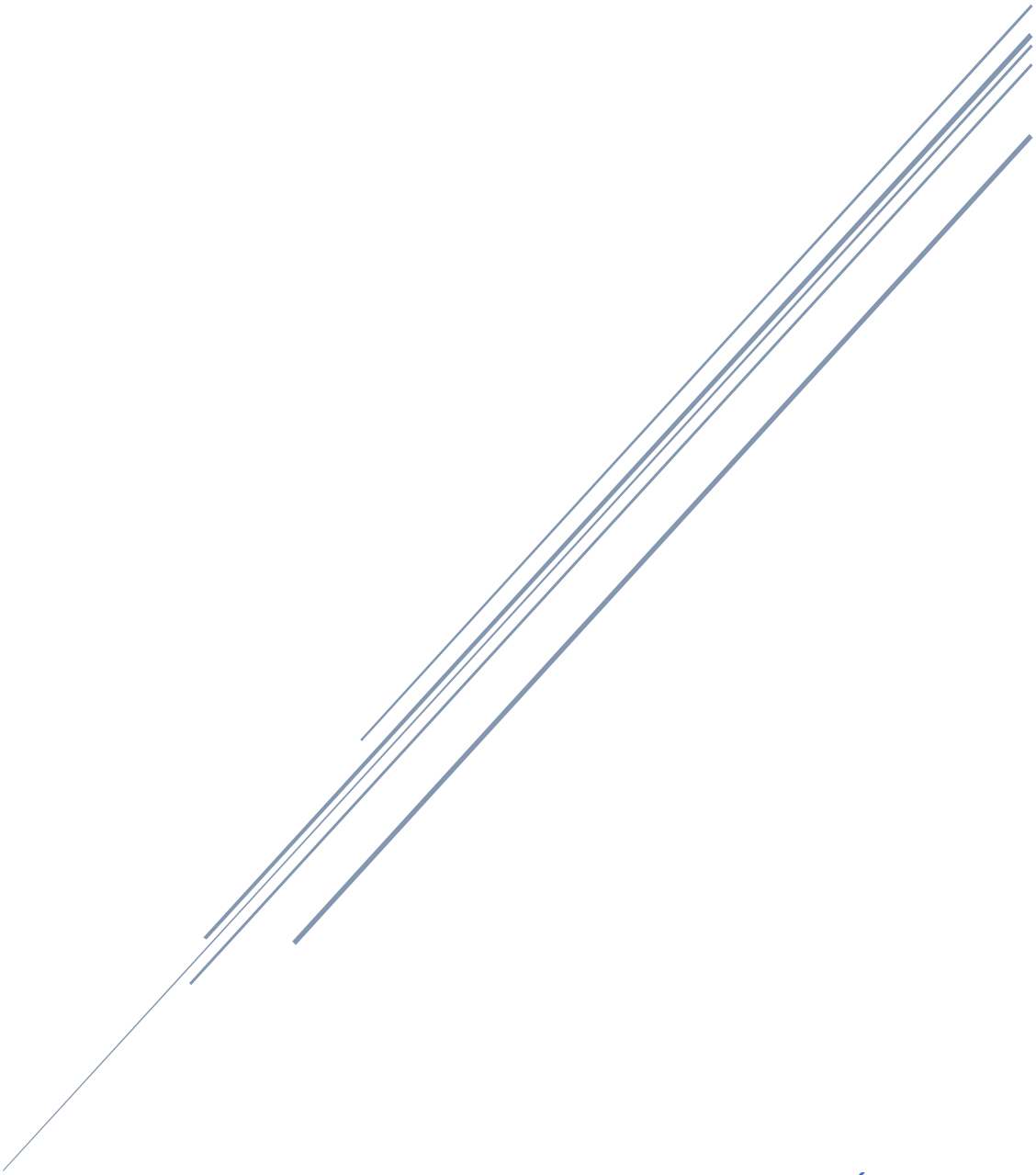
PRÁCTICA 1: GESTIÓN DE PROCESOS

INGENIERÍA INFORMÁTICA



JAIME HERNÁNDEZ

2023/24

INTRODUCTION

La creación de procesos con fork() es un concepto fundamental en los sistemas operativos tipo Unix y Linux. Es una técnica que permite a un programa dividirse en dos procesos independientes, conocidos como el proceso padre y el proceso hijo. Este mecanismo es ampliamente utilizado para lograr la concurrencia y la multitarea en los sistemas informáticos modernos.

Cuando un programa llama a la función fork(), el sistema operativo crea una copia exacta del proceso original, incluyendo el código, los datos y el estado actual del proceso. Después de la llamada a fork(), el proceso padre y el proceso hijo se ejecutan de forma independiente, cada uno con su propio espacio de memoria y su propio identificador de proceso (PID). Ambos procesos continúan ejecutando el mismo programa, pero a menudo toman caminos diferentes en función de la lógica del programa y las decisiones tomadas durante la ejecución.

La función fork() devuelve el valor 0 al proceso hijo y el PID del proceso hijo al proceso padre. De esta manera, el programa original puede distinguir entre el proceso padre y el proceso hijo y tomar acciones específicas en función del valor devuelto.

El uso de fork() es esencial para la creación de procesos en sistemas Unix y Linux. Permite la ejecución paralela de tareas, la implementación de servidores concurrentes, y muchas otras aplicaciones donde la multitarea es necesaria. Además, la combinación de fork() con otras llamadas al sistema, como exec(), wait(), y exit(), permite una gestión avanzada de procesos, incluyendo la comunicación y sincronización entre procesos.

Ejercicio 1:

Hacemos uso de la función waitproc () al recibir la señal SIGALRM y utiliza wait() para esperar a que un proceso hijo termine.

```c
void comand_pstree() {
    int pid, status;

    pid = fork(); // creación de proceso

    if(pid == 0) {
        execlp("/bin/pstree", "pstree", "-p", NULL);
    } else {
        wait(&status);
    }
}
```

Creamos un nuevo proceso hijo utilizando fork(). El hijo ejecuta el comando pstree -p utilizando execlp() para mostrar la estructura de procesos. El proceso padre espera a que el hijo termine usando wait(), hacemos de manera similar la función comand_ls().

La función principal del programa comienza obteniendo el PID del proceso principal y luego se ramifica para crear un nuevo proceso hijo. Dependiendo de si es el proceso padre o el hijo, se ejecutan diferentes partes del código.

Proceso A:

- Creamos dos procesos hijos (procesos B y C) y definimos señales para ejecutar comand_ps() y waitproc().
- Al recibir SIGUSR1, ejecuta comand_pstree().
- Al recibir SIGUSR2, espera a que sus hijos (procesos B y C) terminen antes de imprimir un mensaje y salir.

Proceso B:

- Creamos tres procesos hijos (procesos X, Y y Z) y definimos las señales para ejecutar diferentes acciones basadas en el argumento de línea de comandos argv[1].
- Imprimimos su PID, el PID de su padre (proceso A) y el PID de su abuelo (proceso principal).
- Cada hijo (X, Y, Z) tiene su propia lógica para enviar señales y esperar antes de imprimir un mensaje y salir.

Podemos observarlo en la siguiente captura donde aparece parte del código:

```
if (pid != 0) {

    wait(&status);    // espera de ejeccuón hasta que el hijo muera
    printf("Soy ejec (%d) y muero\n", ejec);

} else {
    PID_A = getpid();
    printf("Soy A: mi pid es %d. Mi padre es %d\n", PID_A, ejec);

    pid = fork(); //creación proceso B

    if(pid != 0) {
        signal(SIGUSR1, comand_pstree);  // caso de que reciva una señal (SIGUSR1) se ejecuta comand_pstree
        signal(SIGUSR2, waitproc);  // caso de que reciva una señal (SIGUSR2) se ejecuta waitproc

        wait(&status);
        printf("Soy A (%d) y muero\n", PID_A);
    } else {
        signal(SIGUSR1, comand_pstree);
        signal(SIGUSR2, waitproc);

        PID_B = getpid();
        printf("Soy B: mi pid es %d. Mi padre es %d. Mi Abuelo es %d\n", PID_B, PID_A, ejec);

        //Creación del árbol
        for(int i = 0; i < 3; i++) {
            pid = fork(); // hijos x, y ,z por cada i

            if(pid == 0) { // Caso en el que se este ejecutando el hijo hace el siguiente árbol (horizontal)
                switch(i) { // No creamos mas hijos a partir de x, y,z terminarmos sus procesos
```

El resultado que nos sale cuando ejecutamos el código en la terminal es el siguiente:

```
nick@ubuntu:~/Desktop/SO/p1$ ./ejec A 15
Soy el proceso ejec: mi pid es 6233
Soy A: mi pid es 6234. Mi padre es 6233
Soy B: mi pid es 6235. Mi padre es 6234. Mi Abuelo es 6233
Soy X. mi pid es 6236. Mi padre es 6235. Mi abuelo es 6234. Mi bisabuelo es 6233
Soy Z. mi pid es 6238. Mi padre es 6235. Mi abuelo es 6234. Mi bisabuelo es 6233
Soy Y. mi pid es 6237. Mi padre es 6235. Mi abuelo es 6234. Mi bisabuelo es 6233
```

Formación del árbol: podemos observar que la formación del árbol se ha hecho de manera correcta con la implementación de los for mediante el siguiente esquema:

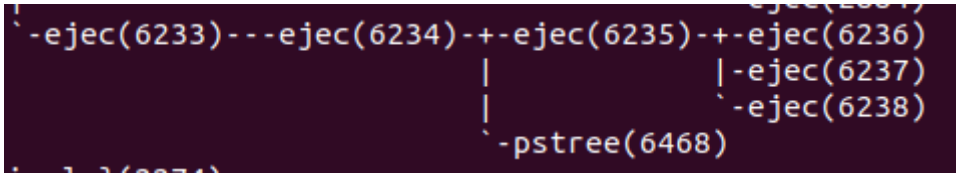
for(…){

    if(fork() == 0) break; -------> Estructura horizontal

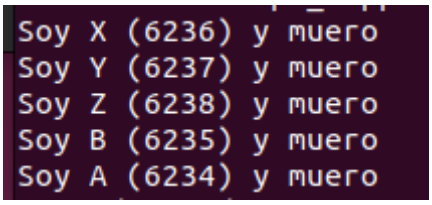
}

La Segunda parte del árbol lo hacemos implementado switch(…) para poder formar los procesos X, Y y Z.

En la parte siguiente podemos observar el árbol creado de los procesos:



Y por último cuando muestra la muerte de cada proceso mediante la función kill() → int kill(int PID, int numero) La llamada kill permite a un proceso enviar una señal a otro proceso o así mismo a través del PID. PID es el PID del proceso señalado y numero es el tipo de señal enviada. Devuelve –1 en caso de error y 0 en caso contrario:




```c
if(strcmp(argv[1], "A") == 0 || strcmp(argv[1], "B") == 0){
    kill(PID_A, SIGUSR1);

    signal(SIGALRM, end_Sleep);
    alarm(1);
    pause();

    kill(PID_X, SIGUSR2);
    kill(PID_Y, SIGUSR2);
    kill(PID_A, SIGUSR2);
    kill(PID_B, SIGUSR2);


    printf("Soy Z (%d) y muero\n", getpid());
    exit(0);
}
```

Ejercicio 2: Realizar un programa llamado copiar.c que permita copiar archivos. El programa recibirá dos argumentos: archivo_origen y archivo_destino, el archivo origen debe existir y el archivo destino se creará. El proceso copiar (proceso padre) generará un proceso hijo y, a partir de ese momento, el proceso padre será el encargado de leer el archivo origen y enviarle la información al proceso hijo que la almacenará en el archivo de destino. La comunicación entre los dos procesos se realizará mediante tuberías (pipe).

El siguiente ejercicio consiste en hacer una copia de archivos utilizando un tubo para la comunicación entre procesos.

Primero, creamos un tubo con un extremo para lectura y otro para escritura. Luego, creamos un proceso hijo utilizando la función fork().

El proceso padre abre un archivo de entrada especificado por el usuario (argv[1]), lee su contenido carácter por carácter y escribe cada carácter en el extremo de escritura del tubo. Una vez que hemos leído todo el archivo, enviamos un carácter de fin de archivo (end) al tubo y cerramos el archivo de entrada.

```
if(pid != 0) {

    close(fd[0]); // Cerramos lectura

    file_read = open(argv[1], O_RDONLY);

    do {
        byte_read = read(file_read, &character, sizeof(char)
        if (byte_read > 0) {
            write(fd[1], &character, sizeof(char));
        }
    } while(byte_read > 0);

    write(fd[1], &end, sizeof(char));

    close(fd[1]);  // Cerramos escritura
```

En el proceso hijo, esperamos 5 segundos usando sleep(5). Luego, cerramos el extremo de escritura del tubo y creamos un nuevo archivo de salida (argv[2]). A continuación, leemos los caracteres del extremo de lectura del tubo, los escribimos en el archivo de salida y continuamos hasta encontrar el carácter de fin de archivo. Finalmente, cerramos el archivo de salida y termina nuestra ejecución como el proceso hijo.

```
} else {

    close(fd[1]); // Cerramos escritura

    file_write = creat(argv[2], 0777);

    while(read(fd[0], &character, sizeof(char)) > 0){
        if (character == end) {
            break;
        }

        write(file_write, &character, sizeof(char));
    }

    close(fd[0]); // Cerramos lectura
}
```

Ejercicio 3: Diseña un programa llamado hijos.c que cree un árbol de procesos según la siguiente estructura a partir de dos parámetros

Creamos dos segmentos de memoria compartida, shm1 y shm2, utilizando shmget(). Estos segmentos son cruciales para compartir información entre procesos el sistema. shm1 se utiliza para el proceso padre y sus hijos directos (x hijos), mientras que shm2 se utiliza para los nietos (y nietos).

```
// Convertimos los argumentos de cadena a enteros.
x = atoi(argv[1]);
y = atoi(argv[2]);

// Creamos dos segmentos de memoria compartida.
shm1 = shmget(IPC_PRIVATE, sizeof(int) * (x + 1), IPC_CREAT
shm2 = shmget(IPC_PRIVATE, sizeof(int) * y, IPC_CREAT | 0666
```

Una vez creados los segmentos de memoria compartida, los vinculamos a punteros, pX y pY, usando shmat(). Esto nos permite acceder y modificar datos dentro de estos segmentos compartidos de memoria.

Ahora entramos en un bucle donde comenzamos a crear procesos. En el caso del proceso padre y sus hijos directos, el bucle se ejecuta x veces. En cada iteración, creamos un nuevo hijo utilizando fork(). Cada hijo muestra su propio PID y los PID de sus padres (anteriores) utilizando la memoria compartida pX. Cada hijo también almacena su propio PID en pX.

```
// Crear procesos hijos directos (x hijos).
for (i = 0; i < x; i++) {
    pid = fork();
    if (pid != 0) {
        // Código del padre
        break;
    } else {
        // Código de los hijos
        printf("Soy el subhijo %d, mis padres son: ", getpid
        for (children = 0; children <= i; children++) {
            printf("%d", pX[children]);
            if (children != i) {
                printf(", ");
            }
        }
        printf("\n");

        // Almacenar el PID del hijo en el segmento pX.
        pX[i + 1] = getpid();
    }
}
```

Cuando hemos creado todos los hijos directos (x), esperamos a que todos ellos terminen utilizando la función wait(). Luego, imprimimos los PID de todos nuestros hijos y también iniciamos y procesos adicionales para los nietos. Estos nietos también muestran su propio PID y almacenan su PID en el segmento de memoria compartida pY.

Finalmente, esperamos a que todos los nietos terminen (y veces) usando wait(). Una vez que todos los procesos han terminado, finalizamos nuestra ejecución. En resumen, creamos una jerarquía de procesos con padres, hijos y nietos, y utilizamos memoria compartida para comunicarnos y compartir información entre estos procesos de manera efectiva.

La salida en la terminal es la siguiente: