**Técnicas Digitales III**

**Trabajo práctico: Procesos**

**Creación de procesos**

**1. Compile y ejecute prc01.c**

El PID(process ID) es un número que identifica a un proceso.

Mi PID es 15730 el de mi papa es 9249.

**2. Compile y ejecute prc02.c. Luego, desde otra consola, ejecute el comando "pstree -p" e identifique los procesos en ejecución.**

Mi pid es 2337 y el pid de papa es 1866. fork() devolvio 2338

Mi pid es 2338 y el pid de papa es 2337. fork() devolvio 0

─gnome-terminal-(1856)─┬─bash(1866)───prc02(2337)───prc02(2338)

│ │ ├─bash(2321)───pstree(2339)

│ │ ├─{gnome-terminal-}(1857)

│ │ ├─{gnome-terminal-}(1858)

│ │ └─{gnome-terminal-}(1859)

**3. El programa prc03.c ejecuta 2 veces la función fork() y hace una espera activa de 30 segundos con la función sleep().Compile y ejecute prc03.c. Luego, desde otra consola ejecute el comando pstree -p e identifique los procesos en ejecución.**

Mi pid es 2354 y el pid de papa es 1866

Mi pid es 2356 y el pid de papa es 2354

Mi pid es 2357 y el pid de papa es 2355

Mi pid es 2355 y el pid de papa es 2354

gnome-terminal-(1856)─┬─bash(1866)───prc03(2354)─┬─prc03(2355)───prc03(2357)

│ │ │ └─prc03(2356)

│ │ ├─bash(2321)───pstree(2358)

**4. La función fork() devuelve el pid del proceso hijo cuando lo ejecuta el padre; y devuelve 0 (cero) cuando lo ejecuta el proceso hijo. ¿Qué estructura de bifurcación de C le parece más conveniente para implementar que padre e hijo ejecuten diferente código (if, while, for, case)? Modifique prc02.c con la estructura de bifurcación seleccionada. Compile y ejecute el programa.**

**int pid;**

**printf("pid: %d, bash pid: %d\n", getpid(), getppid());**

**pid=fork();**

**if(pid==0)**

**{printf("Hijo, pid: %d, ppid: %d\n", getpid(), getppid());**

**return 0; }**

**printf("Padre, pid: %d, ppid: %d\n", getpid(), getppid());**

**wait(NULL);**

**exit (0);**

**pid: 1829, bash pid: 1779**

**Padre, pid: 1829, ppid: 1779**

**Hijo, pid: 1830, ppid: 1829**

**5. Compile y ejecute prc05.c. La variable x es inicializada en 100. Luego es decrementada por el proceso hijo e incrementada por el proceso padre. ¿Por qué x nunca retorna a su valor original?**

Cuando el proceso padre crea al proceso hijo, se copia todo el contenido del proceso padre en el hijo, incluido el espacio de direccionamiento. Este espacio de direcciones es relativo a cada proceso y cada proceso es independiente entre si. Por lo tanto, al modificar la variable x en el proceso padre no se modifica la variable x del proceso hijo, aunque ambas variables se llamen igual.

**6. Modifique el programa prc05.c de tal manera de que además de imprimir el valor de la variable x, imprima la dirección en memoria de esta variable (&x). Ejecute el programa y explique la salida del mismo.**

Soy el proceso: 2147

Soy el proceso padre, pid: 2147 , x = 101, la dirección de x es: 140734056285940

Soy el proceso hijo, pid: 2148 , x = 99, la dirección de x es: 140734056285940

Como podemos observar, cada variable x tiene un valor distinto pero la dirección de memoria es la misma en ambos casos. Esto sucede porque la dirección de la variable x mostrada es una dirección relativa, no física. Por lo tanto, respecto a cada proceso, la variable x se encuentra en la misma dirección, pero en la memoria física de la computadora ambas variables se encuentran en direcciones distintas.

**7. Tome proc02.c y ponga las funciones de las líneas 14 y 15 dentro de un bucle que se repita 3 veces. Imprima también el valor de la variable de control del bucle (variable i). Analice y deduzca cuántos hijos son creados. Justifique su respuesta. ¿Qué sucede con el valor de i?.**

Mi pid es 1874 y el pid de papa es 1766. fork() devolvio 1875, i es:1

Mi pid es 1875 y el pid de papa es 1874. fork() devolvio 0, i es:1

Mi pid es 1874 y el pid de papa es 1766. fork() devolvio 1876, i es:2

Mi pid es 1875 y el pid de papa es 1874. fork() devolvio 1877, i es:2

Mi pid es 1874 y el pid de papa es 1766. fork() devolvio 1878, i es:3

Mi pid es 1875 y el pid de papa es 1874. fork() devolvio 1879, i es:3

Mi pid es 1877 y el pid de papa es 1875. fork() devolvio 0, i es:2

Mi pid es 1879 y el pid de papa es 1875. fork() devolvio 0, i es:3

Mi pid es 1877 y el pid de papa es 1875. fork() devolvio 1880, i es:3

Mi pid es 1880 y el pid de papa es 1877. fork() devolvio 0, i es:3

Mi pid es 1876 y el pid de papa es 1874. fork() devolvio 0, i es:2

Mi pid es 1876 y el pid de papa es 1874. fork() devolvio 1881, i es:3

Mi pid es 1878 y el pid de papa es 1874. fork() devolvio 0, i es:3

Mi pid es 1881 y el pid de papa es 1876. fork() devolvio 0, i es:3

**8. Tome el programa del Ej. 4 y fuerce a que el proceso hijo haga una espera activa de 30 segundos con la función sleep(). El proceso padre debe terminar antes que el proceso hijo. ¿Qué sucede con el proceso hijo?. Ejecute "pstree -p" e identifique si persisten los procesos en cuestión. Observe los números de pid.**

Unico proceso con pid: 2052

Soy el proceso padre, pid: 2052 ,el pid de mi papa: 1766, fork() devolvio 2053

Mi pid es 2052

Soy el proceso hijo, pid: 2053 ,el pid de mi papa es: 2052, frok() devolvio 0

Mi pid es 2053

─systemd(951)─┬

├─prc08(2053)

Vemos que el proceso hijo quedó huérfano y lo adoptó otro proceso.

**9. Tome el programa del Ej. 8 y agregue al final del código del proceso padre la función wait(NULL). Ejecute "pstree -p" e identifique si persisten los procesos en cuestión. Observe los números de pid.**

Unico proceso con pid: 2102

Soy el proceso padre, pid: 2102 ,el pid de mi papa: 1766, fork() devolvio 2103

Soy el proceso hijo, pid: 2103 ,el pid de mi papa es: 2102, frok() devolvio 0

Mi pid es 2103

Mi pid es 2102

─gnome-terminal-(1756)─┬─bash(1766)───prc09(2102)───prc09(2103)

│ │ ├─bash(2036)───pstree(2104)

**10. Tome el programa del Ej. 5 y fuerce a que el proceso hijo entre en un bucle infinito con la función while(1). ¿Qué sucede con el proceso hijo? ¿Con qué combinación de teclas puede terminar el proceso hijo?**

Al colocar while(1), el proceso queda bloqueado pues entra en un bucle infinito. La forma de terminar este proceso es con la combinacion de teclas ctrl+C, que envia al sistema operativo una señal SIGINT que fuerza al proceso a terminar.

**11. Similar a lo hecho en el Ej. 10, ahora fuerce a que el proceso padre entre en un bucle infinito con la función while(1). Visualice luego los procesos en ejecución con "pstree -p" e identifique el proceso “zombie”.**

gnome-terminal-(1982)─┬─bash(2118)───pstree(2251)

│ │ ├─bash(2213)───prc11(2249)───prc11(2250)

En este caso el proceso padre(2249) continua ejecutándose indefinidamente, por lo que el proceso hijo(2250) terminará antes y quedará en estado zombie.

**12. Escriba y compile un programa que imprima un texto en el flujo stdout y otro texto en el flujo stderr mediante la función fprintf().**

fprintf(stdout, "Texto stdout\n");

fprintf(stderr, "Texto stderr\n");

**Ejecute el programa desde la consola. ¿A dónde está direccionado cada flujo?.**

En este caso, cada flujo está direccionando a los descriptores de archivo de salida(stdout=standard output) y el de error(stderr=standard error).

**13. Desde la consola o a través de un script ejecute los siguientes comandos:**

**Ejecute**

**ls -al > ./stdout**

**Ejecute**

**cat stdout**

**¿Qué operaciones se han realizado?. Luego de ejecutar el primer comando,**

**¿observa algo por consola?, ¿por qué?.**

El primer comando manda la salida del comando ls -al que lista todos los archivos con sus permisos, fecha de creación, usuario, etc; y la salida de este comando pasa al descriptor de archivo stdout. Cuando se ejecuta este primer comando no se ve nada por pantalla, pero luego con el segundo comando, cat muestra el contenido del archivo stdout el cual es la lista que le paso el comando ls -al y se muestra dicha lista por pantalla.

**14. Ejecute el programa creado en el Ej. 12 de la siguiente manera:**

**Ejecute**

**./prc12 2> err.txt**

**Compare el resultado con la salida del Ej. 12. ¿Qué observa por consola?.**

Se observa solo “Texto stdout” y no “Texto stderr”.

**¿Cuál es el contenido del archivo err.txt? ¿Cuál es la función del operador “2>” ?**

En el archivo err.txt creado se guarda el contenido del descriptor de archivo de error. Por lo tanto, el operador 2 que identifica al stderror indica a que descriptor hace referencia cuando la salida fprintf del programa pasa al archivo de texto.

**Función execl()**

**15. Compile y ejecute prc15.c. ¿Qué sucede al ejecutar la función execl()?**

Al ejecutar la función execl() se listaron todos los archivos del directorio con sus permisos, fecha de creación, usuario, etc; es decir ejectó el comando ls -l y a su vez retorna un entero a la variable entera err.

**16. En el archivo prc15.c, comente la línea 13 y descomente la línea 14. Compile y ejecute. ¿Qué observa por consola? ¿Por qué ha cambiado la salida respecto al Ej. 15?.**

En este caso la función execl() devuelve un -1, indicando un error. Ésto se debe a que, en el ejercicio 15, el primer argumento de execl() indica la ruta donde se encuentra el comando ls. Al comentar ésta linea y descomentar la siguiente, el primer argumento de la nueva funcion execl() es un directorio que no existe y por lo tanto no encuentra al comando ls. Por esto devuelve un -1 y entra al if mostrando por pantalla "Este printf se ejecuta en caso de error. Por que?\n".