SISTEMAS OPERATIVOS

TALLER No. 4: Operaciones sobre procesos

OBJETIVOS

1. Comprender cómo se crean y cómo se terminan procesos en el sistema operativo mediante funciones de la capa de servicios del sistema operativo Linux y Windows.
2. Comprender la jerarquía de procesos, las diferencias y similitudes con sus imágenes de memoria.
3. Enviar señales de gestión a los procesos ene Linux mediante el uso del comando **`kill`**.
4. Crear proceso hijo como parámetro que se pasa a un proceso padre.

CONTENIDO DEL TALLER

1. Creación de procesos con **`fork()`**.
2. Creación de procesos con **`exec()`**.
3. Señalización básica de procesos en Linux.
4. Escritura de programa en C para jerarquía de procesos secuencial.
5. Escritura de programa en C para jerarquía de procesos de un solo proceso padre.
6. Creación de proceso hijo pasado como parámetro a proceso padre.
7. Creación de procesos en Windows con **`CreateProcess()`**.

DESARROLLO DEL TALLER

1. Creación de procesos con **`fork()`**.

   Escriba, compile y ejecute el siguiente programa en Linux.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    printf("Proceso padre (pid:%d)\n", (int) getpid());
    // Creación proceso hijo
    int rc = fork();
    if (rc < 0) {
        // Falla creación proceso hijo
        printf("Falló fork()\n");
        exit(1);
    } else if (rc == 0) {
        // Proceso hijo: nuevo proceso
        printf("Proceso hijo (pid:%d)\n", (int) getpid());
    } else {
        // Proceso padre sigue por aquí
        printf("Proceso padre de (pid:%d)\n", rc);
    }
    return 0;
}
```

   1.1 ¿Cuántos procesos se crean el ejecutar este programa?
   1.2 ¿Quién termina primero la ejecución? Ejecútelo varias veces.
   1.3 ¿Qué modificaciones haría sobre el programa anterior para verificar que los procesos tienen la misma imagen de memoria, pero son espacios de memoria separados? Hágalas y compruébelo.
   1.4 ¿Qué modificaciones haría usted al programa anterior para verificar la jerarquía de estos procesos en ejecución mediante las herramientas **`top`**, **`htop`** o **`ps`**? Busque visualizar el PID y PPID.
   1.5 Agregue al programa anterior una espera en el proceso mediante el llamado al *system call* **`wait()`**: https://man7.org/linux/man-pages/man2/wait.2.html
   1.6 Después de esta última modificación, ¿cuál proceso termina primero?

2. Creación de procesos con **exec()**.

Escriba, compile y ejecute el siguiente programa en Linux.

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid;
    /* Se crea nuevo proceso */
    pid = fork();
    if (pid < 0) { /* error en fork() */
        printf("Falló fork()");
        return 1;
    }
    else if (pid == 0) { /* Proceso hijo */
        execlp("/bin/ls","ls","-l",NULL);
    }
    else { /* Proceso padre */
        /* Se espera a que proceso hijo termine */
        wait(NULL);
        printf("Hijo termina\n");
    }
    return 0;
}
```

2.1 ¿Cuántos procesos se están creando en este caso?

2.2 ¿Cómo podría modificar el programa anterior para que el proceso hijo mantenga su ejecución?

2.3 Con la modificación anterior (realizada por usted), intente visualizar e identificar la jerarquía de los procesos que se crean a partir del programa anterior.

2.4 ¿Cómo podría verificar que se trata de dos imágenes de memoria completamente diferentes?

2.5 Modifique el programa anterior para que: i) el proceso hijo mantenga su ejecución, ii) el proceso padre no espere a que el hijo termine. Con esta modificación verifique de nuevo la jerarquía de procesos. ¿Qué cambió en la jerarquía de procesos?

3. Señalización básica de procesos en Linux.

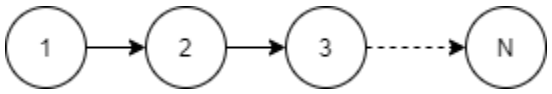
   En Linux se puede finalizar un proceso a la fuerza usando el comando **`kill -SIGKILL <pid>`**. Si se desea que un proceso termine normalmente, se debe enviar la señal **`SIGTERM`** y no la señal **`SIGKILL.`** También se pueden enviar otras señales como **`SIGSTOP`** que sirve para poner en pausa un proceso y **`SIGCONT`** para continuar la ejecución de un proceso previamente detenido. **`kill`** no es solo un comando, también es una *system call* que se usa para enviar diferentes señales a los procesos.

   3.1 Abra dos sesiones mediante PuTTy al sistema operativo e inicie sesión en ambas,
   3.2 Desde la segunda sesión de PuTTY, ubique el PID del proceso en ejecución asociado a la primera sesión: ¿cuál es el nombre del proceso de usuario a la *shell* que le permite ejecutar ordenes en la primera sesión?
   3.3 Desde la segunda sesión envíe una señal **`SIGKILL`** al **PID** del proceso hijo que identificó en el numeral anterior. ¿Qué sucedió?
   3.4 Inicie sesión de nuevo en la primera sesión.
   3.5 Desde la segunda sesión envíe ahora una señal **`SIGKILL`** al **PPID** del proceso asociado a la *shell* de la primera sesión. ¿Qué sucedió?
   3.6 Escriba un programa en C que ejecute un ciclo infinito. Ubique el PID de este proceso y señalícelo para que se detenga (**`SIGSTOP`**). Verifique su estado con **`top`** y una vez verificado señalícelo para que continue su ejecución. Use dos sesiones de PuTTy: en una ejecuta el proceso que se queda en un ciclo, en la segunda ejecute las señales al proceso.

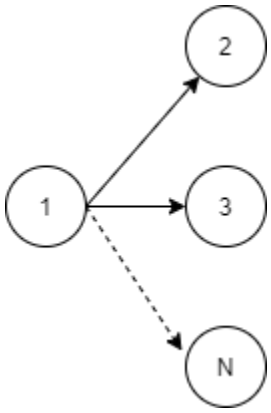
   La documentación de **`kill`** como *system call* para que pueda ser usada desde el código de sus propios programas, la puede consultar en el siguiente enlace.

   - https://man7.org/linux/man-pages/man2/kill.2.html

4. Escriba un programa en C que cree la siguiente jerarquía de procesos padre – hijo en Linux con N ($2 \leq N \leq 10$) determinado por el usuario en tiempo de ejecución. Imprima el PPID y el PID de cada proceso para verificar que la jerarquía de procesos corresponde a la de la figura. Recuerde que puede usar las *system call* `getpid()` y `getppid()`.



5. Repita el programa del numeral anterior, pero para la siguiente jerarquía de procesos de la figura indicada a continuación.



6. Modifique el programa del numeral 2 para que reciba como parámetro el nombre de otro programa. El programa recibido como parámetro debe crearse como proceso hijo y puede ser un ejecutable del sistema operativo (p. ej.: **`ls`**) o puede ser un programa escrito por usted. Tenga en cuenta que el programa pasado como parámetro podría recibir parámetros también en su función **`main()`**. Para este punto apóyese en la *system call* **`execvp()`** cuya documentación puede consultar en el siguiente enlace: https://linux.die.net/man/3/execvp

   Recuerde que los programas en C inician su ejecución en la función **`main()`** que tiene el siguiente prototipo: `int main(int argc, char* argv[])`. Así, por ejemplo, si se ejecuta el programa **`ls`** con la opción **`-l`** y **`-h`**, la orden en la *shell* sería la siguiente.

```
# ls -l -h
```

   En este caso, los argumentos de la función **`main()`** del programa **`ls`** tienen los siguientes valores al momento de su ejecución.

   `argc = 3` → Total de elementos en el arreglo **`argv[]`**.
   `argv[0] = "ls"` → Nombre del programa.
   `argv[1] = "-l"` → Primera opción recibida por el programa desde la *shell*.
   `argv[2] = "-h"` → Segunda opción recibida por el programa desde la *shell*.

7. Creación de procesos en Windows con **`CreateProcess()`**.

   Escriba, compile y ejecute el siguiente programa en Windows.

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

void main(int argc, char* argv[]) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    if (!CreateProcess(NULL, // Usar la línea de comandos
        "C:\\Windows\\system32\\calc.exe", // Ejecutable
        NULL,           // Manejador del proceso: no heredable
        NULL,           // Manejador del hilo: no heredable
        FALSE,          // No herencia
        0,              // Sin flags
        NULL,           // Usar bloque del entorno del padre
        NULL,           // Use el directorio de inicio del padre
        &si,            // Apuntador a estructura STARTUPINFO
        &pi))           // Apuntador a estructura PROCESS_INFORMATION
    {
        printf("Falló CreateProcess() (%d).\n", GetLastError());
        return;
    }
    // Esperar hasta que proceso hijo termine.
    WaitForSingleObject(pi.hProcess, INFINITE);
    // Terminar proceso padre y cerrar manejadores.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

7.1 ¿Cuántos procesos crea el programa anterior?

7.2 ¿Los procesos del programa anterior tienen el mismo mapa de memoria?