Fundamentos de los Sistemas Operativos

Práctica 5: Procesos

El objetivo fundamental de este bloque es aprender a usar llamadas al sistema relacionadas con procesos.

Como ya sabes, las llamadas al sistema son la API (*Application Programming Interface*) que usan los programas para solicitar servicios al sistema operativo. Esta práctica servirá para que aprendas a usar llamadas al sistema Linux relacionadas con procesos. Recuerda que en el Campus Virtual tienes el material de apoyo para realizar esta práctica.

1 Introducción

El objetivo principal de esta práctica es que aprendas a usar llamadas al sistema Linux relacionadas con procesos pesados.

En UNIX todo proceso se crea mediante la llamada al sistema *fork*. El proceso que llama a *fork* se conoce como **proceso padre** y el proceso creado se conoce como **proceso hijo**. Todos los procesos tienen un único proceso padre, pero pueden tener varios procesos hijos. Todo proceso tiene un identificador numérico único **(*PID – process identifier)*** que permanece constante durante toda la vida del proceso.

En esta práctica aprenderás a utilizar la llamada al sistema *fork* para la creación de procesos y la familia de llamadas al sistema *exec* que permiten la carga de un programa en memoria. Igualmente conocerás las llamadas al sistema que nos permiten obtener el identificador de un proceso y el de su padre (*getpid*, *getppid*).

En la parte final de esta práctica veremos llamadas al sistema que permiten que un proceso padre espere hasta que uno de sus hijos haya finalizado: *wait* y *waitpid*. Además, la llamada al sistema *exit* que finaliza un proceso sirve para informar a su proceso padre acerca de cómo finalizó (ej. si ocurrió algún error de ejecución).

2 Objetivos de aprendizaje

Los objetivos de aprendizaje de esta práctica son:

- Saber crear procesos y cargar programas en memoria (*fork* y *exec*).
- Saber cómo obtener información de un proceso: identificador de proceso (*getpid*), identificador del proceso padre (*getppid*), tiempos de uso de la CPU (*times* y *getrusage*).
- Saber sincronizar procesos padres e hijos (*wait* y *waitpid*).
- Finalizar un proceso y devolver un código de salida (*exit*).

Esta práctica tiene asociada una entrega y forma parte de la calificación final.

3 Requisitos previos

Para abordar esta práctica debes haber completado la Práctica 3, ya que necesitas saber cómo realizar llamadas al sistema Linux desde C y tratar adecuadamente los errores que se pueden producir en las llamadas al sistema. También necesitas saber procesar los argumentos que se le pasan a un programa desde la línea de órdenes (**argc** y **argv**).

4 Bibliografía

Para realizar esta práctica te aconsejamos consultar UNA de las siguientes referencias bibliográficas:

- [1] *Linux System Programming, 2nd Edition*. Puedes encontrar información relacionada con las llamadas al sistema que se trabajan en esta práctica en el capítulo 5.
- [2] Programación en Linux (2ª edición). Kurt Wall et al. Prentice Hall. Puedes encontrar información relacionada con las llamadas al sistema que se trabajan en esta práctica en el capítulo 13.
- [3] *Advanced Programming in the UNIX Environment (2nd Edition)*. W. Richard Stevens, Stephen A. Rago. Addison Wesley. Puedes encontrar información relacionada con las llamadas al sistema que se trabajan en esta práctica en el capítulo 8.
- [4] UNIX Programación Avanzada (3ª edición). Francisco M. Márquez. Ra-Ma. Puedes encontrar información relacionada con las llamadas al sistema que se trabajan en esta práctica en los capítulos 5 y 6.

5 Plan de actividades

Actividades	*Objetivos / orientaciones*
Leer documentación sobre las llamadas al sistema: *fork, exec, getpid, getppid, times, getrusage, wait, waitpid, exit*	Conocer el funcionamiento básico de las principales llamadas al sistema relacionadas con procesos. Antes de empezar a resolver los problemas propuestos deberías leer al menos una de las fuentes recomendadas [1] [2] [3] [4].
Sesiones prácticas	Habrá dos sesiones prácticas en el laboratorio. El profesor realizará ejemplos de uso de llamadas al sistema relacionadas con procesos.
Ejercicios de entrenamiento	Es muy importante que realices los ejercicios propuestos en esta ficha. Están diseñados para aprender los aspectos esenciales del uso de llamadas al sistema relacionadas con procesos y te facilitarán la realización del ejercicio entregable.
Ejercicio entregable	Debes realizar la tarea propuesta en esta ficha y entregarla en Moodle.

6 Ejercicios de entrenamiento

En este apartado te proponemos una hoja de ruta para que te adiestres de forma autónoma en los objetivos de la práctica. Te recomendamos que hagas todos los ejercicios en el orden en el que están propuestos.

6.1 Obtener atributos básicos de un proceso

Como primer ejercicio te proponemos que crees un programa en C que averigüe qué PID tiene (*getpid*) y el PID de su padre (*getppid*). Si ejecutas repetidamente el programa, ¿qué observas?

6.2 Crear procesos pesados

Escribe un programa que cree un proceso hijo de tal forma que tanto el proceso padre como el proceso hijo impriman un mensaje cada segundo como el que se muestra a continuación, durante diez iteraciones:

```
Padre PID XXXX: Iteración 1
Hijo PID YYYY: Iteración 1
…más iteraciones …
```

Para resolver el ejercicio tendrás que hacer uso de la llamada al sistema *fork()*. Puedes consultar detalles acerca de la llamada al sistema *fork()* en la bibliografía recomendada en la ficha [1] [2] [3] [4] y en el manual en línea del sistema (**man 2 fork**).

A continuación modifica el programa anterior declarando una variable global entera inicializada a 1. Modifica el código para que una vez creado el proceso hijo, ambos procesos muestren el valor de la variable global en cada iteración, de forma que el padre incrementa el valor de la variable en 1 en cada iteración y el hijo en 5:

```
Padre PID AAAA: Iteración 1. Valor de la variable x: XXXX
Hijo PID BBBB: Iteración 1. Valor de la variable x: XXXX
…más iteraciones …
```

Observa atentamente la salida para ver si se corresponde con la que esperabas. Es muy importante que entiendas que los procesos pesados no comparten memoria y, por tanto, las variables globales parten con los valores que contengan antes de la llamada al sistema *fork()* para el proceso hijo, pero a partir de ese momento se trata de variables de dos procesos pesados diferentes y por tanto están almacenadas en posiciones de memoria diferentes.

6.3 Cargar y ejecutar un programa

Escribe un programa que lance la orden de borrar un archivo o directorio en función del parámetro que se le pasa:

borrar *file|dir*

Si el parámetro es un directorio deberá ejecutar la orden *rmdir*. Si el parámetro es un fichero deberá lanzar la orden *rm*.

6.4 Espera y terminación de procesos (1)

Haz un programa que imprima la fecha actual invocando a la orden *date*, haga una pausa de 10 segundos, luego vuelva a llamar a la orden *date* y finalmente imprima un mensaje de despedida.

Cada vez que quieras ejecutar *date*, tendrás que crear un proceso hijo que lance la orden con la llamada al sistema *exec()*. El programa padre tendrá que esperar a que el proceso hijo finalice, para que se respete el orden de ejecución descrito arriba. Esto lo tendrá que hacer con la llamada al sistema *wait()*.

6.5 Espera y terminación de procesos (2)

(En este punto puedes pasar a realizar el ejercicio entregable. Si quieres practicar un poco más, puedes realizar este ejercicio algo más simple).

Escribe un programa que cree dos procesos hijos encargados de contar el número de caracteres de dos cadenas almacenadas en sendos ficheros de texto. La forma de invocar al programa será la siguiente:

./cuentacaracteres *file1 file2*

En primer lugar, deberás controlar que el programa se invoque correctamente. El programa debe crear dos procesos hijos de forma que cada uno de ellos cuente el número de caracteres almacenados en cada fichero. Crearás ambos procesos hijos mediante la llamada al sistema *fork()*. El número máximo de caracteres permitido para cada fichero es de 30. Cada proceso hijo debe retornar al padre el número de caracteres que contiene el fichero analizado. Retornará -1 si la cadena contiene más de 30 caracteres y -2 si el fichero no existe o bien no se tienen los permisos de acceso adecuados. El proceso padre esperará a que ambos procesos hijos terminen y mostrará por pantalla un mensaje indicando el número de caracteres contenidos en cada fichero:

```
Padre PID XXXX: El fichero file1 contiene N caracteres
```

```
Padre PID XXXX: El fichero file2 supera el número máximo de caracteres permitido
```

```
Padre PID XXXX: El fichero file2 no existe o no se tienen los permisos de acceso adecuados
```

Para este ejercicio será necesario emplear las llamadas al sistema *wait()* o *waitpid()* y *exit()*. Mediante la llamada al sistema *wait()* el padre esperará por la terminación de los hijos, además de permitirle recuperar el valor de retorno de cada uno de ellos. Los hijos por su parte emplearán *exit()* para retornar el número de caracteres que contiene el fichero; -1 si supera el máximo de 30 caracteres y -2 si el fichero no existe o bien no se tienen los permisos de acceso adecuados. Consulta el manual en línea del sistema para ver cómo recuperar el valor de retorno de un proceso hijo a partir de la llamada al sistema *wait()* o *waitpid()* (**man 2 wait**; **man 2 waitpid**).

7 Ejercicio entregable: implementar un shell

En esta práctica tendrás que desarrollar un pequeño *shell* que recogerá líneas tecleadas por el usuario y las procesará como hace el *shell* que utilizas en Linux.

Por ejemplo, el programa será capaz de leer esta línea escrita por el usuario: «cat fichero.txt» y ejecutar la orden *cat* con el argumento *fichero.txt*.

7.1 Implementación básica del shell

Para implementar este *shell*, cada vez que el usuario escriba una línea, tendrás que crear un proceso hijo que ejecute la orden deseada y con los argumentos que haya escrito el usuario. Esto lo harás combinando las llamadas al sistema *fork()* y *exec().* A su vez, el propio *shell* tendrá que esperar a la terminación de la orden con la operación *wait*(). Tras ello el *shell* volverá a leer una nueva línea escrita por el usuario. *(Nota: esta es la técnica que realmente utilizan los shells de Unix para ejecutar las órdenes).*

Si el usuario escribe la palabra «exit», el *shell* terminará (reconocerá esa palabra como una orden especial para finalizar).

IMPORTANTE: para lanzar el proceso hijo, **NO** se puede utilizar la función *system*(). Hay que lanzarlo necesariamente con alguna variante de *exec*(). La función *system()* es una operación de alto nivel que justamente realiza lo que estamos pidiendo que tú implementes por ti misma/o.

7.2 Requisitos opcionales

La funcionalidad básica del *shell* te dará derecho a obtener un **máximo de 6 puntos** en la calificación de esta práctica. Si deseas tener opción a más puntos, puedes implementar las funcionalidades adicionales que aquí te especificamos en forma de *retos*.

7.2.1 Mayúsculas y minúsculas [+1 punto]

Permite que las órdenes se puedan escribir en cualquier combinación de mayúsculas y minúsculas. Por ejemplo, para lanzar la orden **date** el usuario podría escribir cualquiera de estas formas: «date», «DATE», «DaTe».

7.2.2 Cambio de directorio con «cd» [+1 punto]

Observa lo que ocurre si con tu *shell* intentas cambiar el directorio de trabajo escribiendo «cd *nuevo_directorio*». Verás que no te hace caso y sigue en el directorio original. Esto es porque la operación de cambio de directorio no puede hacerse desde un proceso separado, sino que la tiene que ejecutar el propio *shell* para mantener de forma permanente el cambio de directorio.

En esta opción te pedimos que reconozcas la orden «cd» como un caso especial en el cual no se lanza un proceso hijo, y en lugar de ello se invoca a una operación *chdir()* para realizar el cambio dentro del propio *shell*.

7.2.3 Ejecución asíncrona con «&» [+2 puntos].

En este reto, te pedimos que añadas esta nueva funcionalidad: si al final de la orden se escribe el carácter **«&»**, el proceso hijo se ejecutará de forma asíncrona

(*background*). El *shell* no esperará a la finalización del hijo y seguirá leyendo nuevas órdenes del usuario.

Ejemplo: **find /etc -name "*.d" &**

Al lanzar una orden con "&", debe mostrarse en pantalla el PID del proceso que se genera, igual que hace el *shell* de Unix.

7.2.4 Redirección de la entrada y salida estándares [+2 puntos]

Los *shells* de Unix permiten redirigir la entrada y la salida de datos de las órdenes que lanzan. Esto se declara usando los signos '>' y '<':

- Redirección de salida: ***orden > fichero_salida***
- Redirección de entrada: ***orden < fichero_entrada***

En ambos casos, el fichero es cualquier ruta válida hacia un fichero regular. El efecto es que la salida (o entrada) de datos, en lugar de ser la terminal, es el fichero especificado.

Por ejemplo, esta orden: **ls /etc >listado.txt**

Genera un fichero «listado.txt» que contiene la lista de ficheros que hay en el directorio «/etc». Es decir, se ha ejecutado «ls /etc» y su resultado, en lugar de mostrarse en pantalla, se vuelca en un fichero.

Para implementar este reto tendrás que cerrar la salida (o la entrada) estándar en el proceso hijo y a continuación abrir el fichero que actúa como redirección. Recuerda que la entrada estándar tiene asociado el descriptor de fichero número 0 y la salida estándar es el descriptor número 1. Además, el núcleo Unix/Linux, cuando abre un nuevo fichero, utiliza el descriptor más bajo disponible. Por tanto, si cierras el fichero 1 y justo a continuación abres un fichero cualquiera, este recibirá el descriptor número 1. Con eso consigues la redirección de la salida.

7.2.5 Tuberías (*pipes*) [+2/3 puntos]

Las tuberías son una de las características más potentes del *shell* de Unix. Con la sintaxis ***orden1* | *orden2*** lanzamos *orden1* y *orden2* en paralelo y conectamos la salida de *orden1* con la entrada de *orden2*. Todo lo que normalmente saldría por pantalla en *orden1* se introduce como entrada de datos de *orden2*.

Veamos un ejemplo. La orden **wc -l** lee líneas del teclado y, cuando finalizamos la entrada de datos, nos muestra el número de líneas que hemos escrito. ¿Qué pasa si conectamos el programa **ls** con **wc -l** mediante una tubería?

Ejemplo: **ls | wc -l**

Lo que ocurrirá es que la salida de **ls** se enviará como datos de entrada al programa **wc** y este nos mostrará cuántas líneas generó **ls**. En este caso, nos mostrará un número que coincidirá con cuántos ficheros hay en el directorio actual.

En este *reto* te pedimos que implementes las tuberías de dos procesos, a cambio de un *botín* de dos puntos. Para conseguirlos tendrás que crear un fichero especial llamado «tubería» o *pipe* en inglés, mediante la llamada al sistema *pipe*(). Además, tendrás que cerrar previamente la entrada y salida estándar de cada proceso y usar la operación *dup()* para reutilizar la tubería de la forma adecuada. ¡Busca en Internet cómo implementar una tubería (*pipe*) en lenguaje C!

Obtendrás otro punto adicional si tu *shell* permite crear tuberías con más de dos procesos, como en este ejemplo: **ls | nl | head**

7.3 Entrega del trabajo

Recuerda que debes entregar todos los ficheros fuentes que implementan la solución a este ejercicio entregable, junto con la ficha correspondiente en formato PDF. Empaqueta todos los ficheros en un archivo **.zip** o **tar.gz** y súbelos al Campus Virtual.

8 Calificación del trabajo

Como en los anteriores trabajos, esta práctica 5 se evaluará mediante un criterio funcional (el *shell* funciona conforme a las especificaciones) y mediante un criterio de calidad del código (el código está construido con arreglo a buenas prácticas de programación).

La puntuación final del trabajo resultará de esta fórmula:

$$\mathbf{MIN\ (\ F\mathit{calidad} \times P\mathit{funcional},\ 10\ )}$$

Pfuncional es la *puntuación funcional*, es decir, la nota base de la práctica según criterios puramente funcionales. *Fcalidad* es un factor de calidad que valdrá entre 0,60 y 1 y que ajustará la puntuación funcional según la calidad del código fuente entregado.

Si la puntuación ajustada resulta superior a 10, se obtendrá un 10 en la calificación definitiva.

8.1 Puntuación funcional

La puntuación funcional consistirá en la suma de los puntos obtenidos según las funcionalidades que se hayan implementado satisfactoriamente. Si el objetivo funcional de cada apartado no se cumple plenamente, se te aplicará una puntuación inferior en ese apartado.

Por ejemplo, supongamos que tu entrega consiste en lo siguiente:

- Funcionalidad básica, que funciona bien al 80%.
- Un reto de dos puntos que funciona perfectamente.
- Un reto de un punto totalmente incorrecto.

En este caso, tendrás una puntuación funcional de 4,8 + 2 + 0 = 6,8 puntos.

Dentro de los criterios funcionales, aparte de cumplir con las especificaciones del *Shell*, tu trabajo deberá cumplir estos requisitos:

- El código está correctamente separado en módulos (.h,.c).
- Se documentan las pruebas realizadas.
- Se ha entregado una ficha en PDF, entendible y conforme con la plantilla.
- La ficha describe cómo debe compilarse y probarse el código.
- La entrega se ha realizado como un archivo *zip*, *tar* o *gzip*.

8.2 Factor de calidad

El factor de calidad tendrá un valor entre el 60% y el 100% y se calculará mediante esta fórmula:

F*calidad* = 60% + mantenibilidad + eficiencia + robustez

Este factor cuenta con un término fijo (60%) y un término variable que valorará tres aspectos de calidad. La contribución máxima de cada aspecto, así como su descripción, se explica a continuación:

- **Mantenibilidad (20%)**: legibilidad del código (identificadores entendibles para variables, funciones, constantes, etc.); organización modular; reutilización de código (no hay bloques duplicados de código); facilidad de parametrización/configuración; rutinas para testeo del código; etc.
- **Eficiencia (10%)**: implementación eficiente en cuanto a tiempo de ejecución y a consumo de recursos (sobre todo la memoria).
- **Robustez (10%)**: comprobaciones de errores y situaciones excepcionales, mensajes de error, comprobación de límites de *arrays*, cuidado con las variables sin inicializar, cuidado con el uso de punteros, etc.