Área de Arquitectura y Tecnología de Computadores

Universidad Carlos III de Madrid

SISTEMAS OPERATIVOS

Laboratorio 2. Programación de un intérprete de Shell scripts

Grado de Ingeniería en Informática

Grado en Matemáticas Aplicadas y Computación

Doble Grado de Ingeniería en Informática y Administración de Empresas

Curso 2024/2025

1. Enunciado del laboratorio

Este laboratorio permite al alumno familiarizarse con los servicios para la gestión de procesos que proporciona POSIX. Asimismo, se pretende que conozca cómo es el funcionamiento interno de un intérprete de comandos (shell) en UNIX/Linux. En resumen, una shell permite al usuario comunicarse con el kernel del sistema operativo mediante la ejecución de comandos o comandos, ya sean simples o encadenados.

Para la gestión de procesos pesados, se utilizarán las llamadas al sistema de POSIX relacionadas como fork, wait, exit. Para la comunicación entre procesos, igualmente se utilizarán las llamadas al sistema pipe, dup, close, signal.

El alumno debe diseñar y codificar, en lenguaje C y sobre el sistema operativo Linux, un programa que lea un archivo que incluye comandos de Linux y los ejecute. El archivo tendrá un comando por línea y se debe ejecutar todo el comando de la línea antes de pasar a la línea siguiente. El programa debe seguir estrictamente las especificaciones y requisitos contenidos en este documento.

1.1. Descripción del laboratorio

El programa a desarrollar lleva por nombre "**Scripter**", el cual debe leer línea a línea un archivo de comandos recibido como argumento de entrada, y mediante un intérprete de comandos deberá ejecutar los comandos como si de un terminal nativo se tratara.

A continuación, se muestra un ejemplo del contenido de un archivo de entrada:

```
## Script de SSOO
ls -l
cat fich | grep palabra
cat fich > fich2 &
ls -l | sort < fichero
```

La primera línea es obligatoria, por lo que el programa debe comprobar que exista para saber si el fichero es un script. Si no existe, se debe devolver un error mediante la función **perror** y retornar -1.

Posteriormente, el programa desarrollado debe ser capaz de ejecutar, uno por uno, los comandos listados en el fichero. No pueden incluirse líneas vacías, por lo que debe controlarse que este formato sea correcto. En caso de encontrar una línea vacía, el programa deberá terminar y mostrar un mensaje de error mediante la función **perror** y retornar -1.

El programa debe ejecutar todos los comandos, aunque estos fallen (el fallo de un comando no implica un fallo en el intérprete). El intérprete debe finalizar:

1. Cuando se han completado todos los comandos del fichero.
2. Cuando se ha producido un error en las llamadas al sistema utilizadas en el código (fork, exec, pipe, dup, etc.).
3. Cuando la primera línea del fichero de entrada sea diferente a "**## Script de SSOO**".
4. Se ha encontrado una línea vacía.

1.2. Procesamiento de comandos

Para el desarrollo de este laboratorio, es necesario que los alumnos procesen un fichero de comandos pasado como argumento del programa. **Deben leer carácter a carácter hasta encontrar un salto de línea e insertar un '\0' al final del string.** Una vez implementado este paso, cada una de las líneas puede ser pasada como parámetro a la función ***procesar_linea (****incluida en el código inicial; scripter.c****)***, la cual se encarga de proporcionar todo lo necesario para que el alumno pueda proceder con la gestión de los procesos.

Variables que serán utilizadas por los estudiantes incluidas en el código inicial:

- **char * argvv[]:** estructura equivalente al "argv" que almacena la línea de comandos a la hora de ejecutar un programa. En este caso, esta estructura almacenará la línea correspondiente a un comando con sus argumentos.
- **char * filev[3]:** arreglo que almacena si hay una redirección desde/hacia un fichero. Cada posición se corresponde con:
    - filev[0]: redirección de entrada (descriptor de fichero 0).
    - filev[1]: redirección de salida estándar (descriptor de fichero 1).
    - filev[2]: redirección de salida de error (descriptor de fichero 2).

  En caso de haber redirecciones, el contenido de la posición será el puntero al nombre del fichero. En caso contrario, el valor de la posición será *NULL*.
- **int background:** si hay background, el valor es 1. En otro caso, 0.

El código inicial (*scripter.c*) incluye, dentro de la función *procesar_linea*, un código que muestra por pantalla el contenido de argvv, de background y de redirecciones. Se recomienda evaluar distintos casos una vez que se ha realizado la lectura y entendimiendo del fichero de comandos. Posteriormente, se recomienda modificar ese código para agregar la funcionalidad de ejecución de procesos.

Consideraciones:

1. Una línea puede ser un comando con o sin argumentos (p.e. "ls" vs "ls -l" vs "ls -l directorio").

2. Una línea puede ser una serie (pipe) de comandos encadenados, cada uno con sus respectivos flags y argumentos (ls -l | wc).
3. Además, cada línea de comandos puede tener una serie de caracteres estandarizados al final que indican redirección y background. La sintaxis que se utiliza es la siguiente:
    a. " **>** ": redirección de salida.
    b. " **<** ": redirección de entrada.
    c. " **!>** ": redirección de error.
    d. " **&** ": indica que el comando se ejecutará en background.

Por lo tanto, **se entiende como un comando** a una secuencia de textos separados por espacios. El primer texto especifica el nombre del comando a ejecutar. Las restantes son los argumentos del comando invocado.

El nombre del comando se pasa como argumento 0 (*man execvp*). Cada comando se ejecuta como un proceso hijo directo del intérprete (*man 2 fork*). El valor de un comando es su estado de terminación (*man 2 wait*). Si la ejecución falla se notifica el error por la salida estándar error mediante la función **perror**.

Una secuencia de dos o más comandos es una serie de comandos separados por el carácter " | " (*man 2 pipe*). La salida del primer comando se pasa automáticamente como entrada al siguiente. El programa principal espera a que el último comando termine antes de aceptar una nueva instrucción. El resultado de toda la secuencia será el del último comando ejecutado.

Una **redirección** indica que la entrada o salida de datos de un comando se debe tomar desde/hacia un fichero en lugar de la terminal. La entrada o la salida de un comando o secuencia de comandos puede ser redirigida añadiendo tras él la siguiente notación:

< fichero Usa un **fichero** como entrada estándar abriéndolo para lectura (*man 2 open*).

> fichero Usa un **fichero** como salida estándar. Si el fichero no existe se crea, si existe se trunca (*man 2 open / man creat*).

!> fichero Usa un **fichero** como salida estándar de errores. Si el fichero no existe se crea, si existe se trunca (*man 2 open / man creat*).

En caso de cualquier error durante las redirecciones, se notifica por la salida estándar de error (**perror**), se suspende la ejecución de esa línea y se continúa con la siguiente línea.

Un **comando o secuencia terminado en** " **&** " supone la ejecución asíncrona del mismo, es decir, se ejecuta en segundo plano, por lo cual el programa principal continúa la ejecución del resto de los comandos. Se debe considerar:

1. El proceso padre debe imprimir mediante la salida estándar el *pid* del proceso hijo (Ver Anexos).
2. Se debe gestionar correctamente la finalización de los procesos hijos, ya que de no hacerlo supondrá una penalización en la nota final.

Nota: Para familiarizarse con las estructuras en las que se devuelven los comandos, redirecciones y background, se recomienda al alumno compilar y ejecutar el programa proporcionado para probar diferentes comandos y secuencias de comandos en el archivo de entrada. El objetivo es comprender claramente cómo acceder a cada uno de los comandos y asegurarse de que se decodifican bien. No obstante, para ello, el alumno debe implementar la lectura línea por línea del fichero de entrada, y hacer uso de la función facilitada: **int procesar_linea(char *linea);**

1.3. Desarrollo del intérprete de comandos

Para desarrollar el intérprete de comandos se recomienda al alumno seguir una serie de pasos, de tal forma que se construya el programa de forma incremental. En cada paso se añadirá una nueva funcionalidad sobre el anterior.

1. Lectura del fichero de entrada carácter por carácter, realizando el reconocimiento de las líneas de comandos y añadiendo '\0' como delimitador.
2. Ejecución de comandos simples del tipo ls, who, etc.
3. Ejecución de comandos simples en "background" (&).
4. Ejecución de secuencias de comandos conectados por pipes (|). El número de comandos en una secuencia se limitará a 3, por ejemplo: ls –l | sort | wc. Si los alumnos deciden implementar una secuencia ilimitada de pipes **se considerará para nota complementaria** (nota máxima 10).
5. Ejecución de comandos simples, secuencias de comandos con redirecciones de entrada, salida y de error (<, >, !>) y de background (&).
6. Ejecución de un comando externo (*mygrep*).

1.4. Desarrollo de un comando externo: mygrep

Además del código para el programa *Scripter*, el alumno debe implementar un comando nuevo llamado *mygrep*. Este comando se encargará de buscar una cadena de texto en un fichero de texto y mostrar por la salida estándar aquellas líneas en las que aparece. En caso de que el fichero no contenga dicha cadena de caracteres, se mostrará el mensaje: " %s not found.\n ", indicando la cadena objetivo entre comillas. Finalmente, si se produce cualquier

tipo de error, se debe mostrar un mensaje indicativo por la salida estándar de error utilizando la función **perror** y retornar -1.

El programa debe tener la siguiente interfaz:

mygrep *<ruta_fichero> <cadena_a_buscar>*

La ejecución de este programa se realizará desde el archivo de comandos de entrada del Scripter. Por lo que al detectar el comando *mygrep*, se debe ejecutar en un proceso hijo (como cualquier otro comando) pero indicando la ruta donde se encuentra el binario. Para más información consultar *man 3 exec*.

Para la ejecución de este programa externo se pueden utilizar las funciones que se consideren necesarias, siempre que estén incluidas en el estándar de C. No se permite enlazar bibliotecas estáticas o dinámicas nuevas.

Una vez implementado el programa, se debe agregar una línea de compilación al Makefile para que al ejecutar "make", "make mygrep" y "make clean", el programa se compile automáticamente o se borren los ficheros generados.

Nota: Se debe tener en cuenta que se debe utilizar la salida estándar *(descriptor de fichero = 1)* para presentar el resultado por pantalla de los comandos internos, mientras que **todos los mensajes de error** de los códigos generados se mostrarán por la salida estándar de error (descriptor de fichero = 2) mediante la función **perror (**ver documentación en man 3 perror**)**. Además, **para la realización de la entrega no se permite la aparición de mensajes de depuración en la salida estándar** (no debe aparecer en la pantalla nada más que las salidas de los comandos en ejecución).

2. Código Fuente inicial

Para facilitar la realización de este laboratorio se dispone del fichero *p2_scripter.zip* que contiene código fuente inicial. Para extraer su contenido se deberá ejecutar lo siguiente:

unzip p2_scripter.zip

Al extraer su contenido, se crea el directorio *p2_scripter/*, donde se debe desarrollar el laboratorio. Dentro de este directorio se habrán incluido los siguientes ficheros:

- **Makefile**
  Fichero fuente para la herramienta make, con el que se consigue la recompilación automática solo de los ficheros fuente que se modifiquen. **Deberá ser modificado para agregar la compilación del comando externo solicitado**.
- **scripter.c**

Fichero fuente de C. **Este fichero es el que se debe modificar para el desarrollo del procesador de comandos**.

- **mygrep.c:**

  Fichero fuente de C para el desarrollo del comando externo.

3. Compilación y ejecución

Para poder compilar y ejecutar el laboratorio es necesario utilizar el comando "make". Las reglas iniciales se encuentran ya en el fichero Makefile proporcionado. Sin embargo, es necesario que el alumno **modifique** el archivo **"Makefile"** para agregar la regla de compilación del programa "mygrep".

Ejemplo de compilación:

```
$ make
$ make scripter
$ make mygrep → debe ser agregado en el Makefile por el alumno
$ make clean → debe ser modificado en el Makefile por el alumno
```

Ejemplo de ejecución:

```
$ ./scripter <script_fichero_con_comandos>
```

4. Entrega

4.1. Plazo de entrega

La fecha límite de entrega del laboratorio en AULA GLOBAL será el **03 de abril de 2025 (hasta las 23:55h).**

4.2. Procedimiento de entrega

En AULA GLOBAL se habilitará **un entregador para el código** y otro de tipo **TURNITIN para la memoria**. Deberá ser entregado por **un único** integrante del grupo.

4.3. Documentación a Entregar

Se debe entregar un archivo comprimido en formato ZIP con el nombre:

ssoo_p2_NIA1_NIA2_NIA3.zip

Con los NIAS de los integrantes del grupo. En caso de realizar el laboratorio en solitario el formato será: **ssoo_p2_NIA1.zip.** El archivo debe contener:

- **scripter.c**: Fichero principal del intérprete de comandos.
- **autores.txt**: Fichero de texto en formato csv con un autor por línea. El formato es: NIA, Apellidos, Nombre
- **mygrep.c**: Fichero principal del comando externo solicitado.
- **Makefile**: Fichero Makefile modificado por el alumno para compilar el programa externo *mygrep*.

Importante: no se permite la eliminación/modificación de las variables "*CC*", "*FLAGS*", "*CFLAGS*" en el archivo Makefile.

En el entregador TURNITIN deberá entregarse el fichero con el nombre:

ssoo_p2_NIA1_NIA2_NIA3.pdf

La memoria tendrá que contener al menos los siguientes apartados:

- **Portada:** con los nombres completos de los autores, NIAs y direcciones de correo electrónico.
- **Índice**
- **Descripción del código** detallando las principales funciones implementadas. NO incluir código fuente del laboratorio en este apartado. Cualquier código será ignorado.
- **Batería de pruebas** utilizadas y resultados obtenidos para ambos programas (Scripter y mygrep). Se dará mayor puntuación a pruebas avanzadas, casos extremos, y en general a aquellas pruebas que garanticen el correcto funcionamiento del laboratorio en todos los casos. Hay que tener en cuenta:
    - Que un programa compile correctamente y sin advertencias (*warnings*) no es garantía de que funcione correctamente.
    - Evite pruebas duplicadas que evalúan los mismos flujos de programa. La puntuación de este apartado no se mide en función del número de pruebas, sino del grado de cobertura de las mismas. Es mejor pocas pruebas que evalúan diferentes casos a muchas pruebas que evalúan siempre el mismo caso.
- **Conclusiones**, describe los problemas encontrados, cómo se han solucionado, y opiniones personales.

Se puntuará también los siguientes aspectos relativos a la **presentación** del laboratorio:

- La memoria debe tener números de página en todas las páginas (menos la portada).
- El texto de la memoria debe estar justificado.

La longitud de la memoria no deberá superar las 15 páginas (portada e índice incluidos). Es imprescindible aprobar la memoria para aprobar el laboratorio, por lo que no debe descuidar la calidad de la misma.

NOTA: Se evaluará sólo la última versión entregada del código.

LA MEMORIA ÚNICAMENTE SE PODRÁ ENTREGAR UNA ÚNICA VEZ A TRAVÉS DE TURNITIN.

Normas

1) **Los programas que no compilen o que no se ajusten a la funcionalidad y requisitos planteados, obtendrán una calificación de 0.**
2) Los programas que utilicen funciones de biblioteca **(fopen, fread, fwrite, etc.)** o similares, en vez de llamadas al sistema, obtendrán una calificación de 0.
3) Se prestará especial atención a detectar funcionalidades copiadas entre dos laboratorios. En caso de encontrar implementaciones comunes los alumnos involucrados (copiados y copiadores) perderán las calificaciones obtenidas por evaluación continua.
4) Los programas deben compilar sin warnings.
5) Los programas deberán funcionar bajo un sistema Linux, no se permite la realización del laboratorio para sistemas Windows. Además, para asegurarse del correcto funcionamiento del laboratorio, deberá comprobar su compilación y ejecución en los laboratorios de informática de la universidad (en el servidor **guernika.lab.inf.uc3m.es**). Si el código presentado no compila o no funciona sobre estas plataformas la implementación no se considerará correcta.
6) Un programa no comentado, obtendrá una calificación mínima.
7) La entrega del laboratorio se realizará a través de aula global, tal y como se detalla en el apartado "**Entrega**" de este documento. No se permite la entrega a través de correo electrónico sin autorización previa.
8) Se debe respetar en todo momento el formato de la entrada y salida que se indica en cada programa a implementar.
9) Se debe realizar un control de errores en cada uno de los programas.

Los programas entregados que no sigan estas normas no se considerarán aprobados.

Anexos

1.2. *man function*

man es el paginador del manual del sistema, es decir, permite buscar información sobre un programa, una utilidad o una función. Véase el siguiente ejemplo:

man 2 fork

Las páginas usadas como argumentos al ejecutar *man* suelen ser normalmente nombres de programas, utilidades o funciones. Normalmente, la búsqueda se lleva a cabo en todas las secciones de manual disponibles según un orden predeterminado, y solo se presenta la primera página encontrada, incluso si esa página se encuentra en varias secciones.

Para salir de la página mostrada, basta con pulsar la tecla 'q'.

Una página de manual tiene varias partes. Estas están etiquetadas como NOMBRE, SINOPSIS, DESCRIPCIÓN, OPCIONES, FICHEROS, VÉASE TAMBIÉN, BUGS, y AUTOR. En la etiqueta de SINOPSIS se recogen las librerías (identificadas por la directiva *#include*) que se deben incluir en el programa en C del usuario para poder hacer uso de las funciones correspondientes.

Las formas más comunes de usar *man* son las siguientes:

- **man sección elemento:** Presenta la página de elemento disponible en la sección del manual.
- **man -a elemento:** Presenta, secuencialmente, todas las páginas de elementos disponibles en el manual. Entre página y página se puede decidir saltar a la siguiente o salir del paginador completamente.
- **man -k palabra-clave:** Busca la palabra-clave entre las descripciones breves y las páginas de manual y presenta todas las que casen.

1.3. *Modo background y foreground*

Cuando una secuencia de mandatos se ejecuta en background, el *pid* que se imprime es el del proceso que ejecuta el último mandato de la secuencia.

Cuando un mandato simple se ejecuta en background, el *pid* que se imprime es el del proceso que ejecuta este mandato.

Con la operación de background, es posible que el proceso principal muestre el prompt entremezclado con la salida del proceso hijo. Esto es correcto.

Después de ejecutar un mandato en foreground, el intérprete no puede tener procesos zombies de mandatos anteriores ejecutados en background.

1.4. Mandatos externos

Los mandatos externos (mygrep) se ejecutan como un programa cualquiera, ejecutado sobre un proceso. Por lo tanto:

- Forman parte de secuencias de mandatos
- Tienen redirecciones de ficheros
- Se ejecutan en background

2. Bibliografía

- El lenguaje de programación C: diseño e implementación de programas Félix García, Jesús Carretero, Javier Fernández y Alejandro Calderón. Prentice-Hall, 2002.
- The UNIX System S.R. Bourne Addison-Wesley, 1983.
- Advanced UNIX Programming M.J. Rochkind Prentice-Hall, 1985.
- Sistemas Operativos: Una visión aplicada. Jesús Carretero, Félix García, Pedro de Miguel y Fernando Pérez. McGraw-Hill, 2001.
- Programming Utilities and Libraries SUN Microsystems, 1990.
- Unix man pages (man function)