UNIVERSIDAD DE ALCALÁ

Departamento de Automática

Grado en Ingeniería Informática

Práctica 2: La consola de Linux (II)
Gestión de procesos

Sistemas Operativos

Índice

1. Competencias asociadas a la práctica

1. Comprender la diferencia entre programa y proceso.
2. Comprender los estados principales por los que transita un proceso y la existencia de diferentes estados en función de cada SO: Windows, Unix, etc.
3. Conocer y comprender los posibles estados de un proceso en Linux.
4. Comprender la existencia para todo proceso de una interfaz de comunicación con el exterior: archivos de entrada, salida, y salida estándar de errores, respectivamente.
5. Saber qué significa redireccionar la entrada, salida, y salida estándar de errores.
6. Ser capaz de manejar redirecciones.
7. Comprender la diferencia entre las variables de entorno y variables shell.
8. Ser capaz de manejar las principales variables de entorno y las variables shell desde la consola.
9. Comprender la funcionalidad de las tuberías y su interrelación con el mecanismo de las redirecciones.
10. Ser capaz de manejar tuberías.
11. Comprender qué son las señales.
12. Ser capaz de manejar señales con las órdenes principales (`kill` y `trap`).
13. Ser capaz de realizar las tareas comunes para crear procesos, gestionarlos y terminarlos desde la consola.
14. Comprender la funcionalidad de un filtro y su interrelación con las tuberías.
15. Ser capaz de utilizar los filtros básicos de Unix.

2. Introducción

Desde una perspectiva práctica, Unix se construye sobre dos grandes abstracciones: archivos y procesos. Todos los elementos “estéticos” del sistema se abstraen en forma de archivos, desde los archivos regulares que todos conocemos, hasta los directorios e incluso dispositivos físicos. Si bien los archivos abstraen de una manera muy conveniente los distintos elementos que componen el computador, no tienen algo fundamental para que los computadores sean útiles, capacidad de ejecución, que es aportada por los procesos.

Un error muy extendido es confundir el concepto de proceso con el de programa. Un programa es una secuencia de órdenes ejecutables guardadas en un archivo. Si ese programa es leído por el Sistema Operativo (SO), guardado en memoria principal y ejecutado, se transforma en proceso. Por lo tanto, otra forma de definir proceso es diciendo

que un proceso es un programa en ejecución. Conviene fijarse en que, mientras un programa es algo estático, que no cambia con el tiempo, un proceso es algo dinámico; se van ejecutando sus instrucciones, el contenido de las variables cambia con el tiempo, se le asignan nuevas zonas de memoria, etc. Por otra parte, de un único programa pueden surgir distintos procesos. Por ejemplo, generalmente vamos a tener un único ejecutable de *Firefox* y, sin embargo, podemos tener varias instancias de *Firefox* funcionando, es decir, de un solo programa pueden generarse varios procesos.

En esta práctica los contenidos aportados sobre el manejo de los archivos se complementa con los correspondientes al manejo de procesos. Se estudia cómo crear procesos, cómo gestionarlos, cómo se pueden comunicar procesos y, finalmente, cómo terminarlos. De este modo, se cierran los conocimientos mínimos necesarios para poder desenvolverse en el terminal, aportando los conocimientos imprescindibles para permitir introducir la programación de *shell scripts*, una herramienta de vital importancia para la administración de sistemas.

3. Procesos

Un proceso es un programa en ejecución, cargado en memoria, sea en memoria principal o bien en el área de *swap*. Un programa almacenado en memoria, por ejemplo una orden o comando, no es un proceso hasta que no se ejecute. En un sistema multiproceso, como Unix, pueden existir varios procesos simultáneamente. La mayoría de las máquinas son monoprocesador por lo que, en cada instante, sólo un proceso se encuentra en estado de ejecución. Normalmente, los programadores no preparan sus programas para que cedan voluntariamente la CPU a otros procesos, por lo que el SO debe considerar el procesador como un recurso más del sistema, gestionando su reparto entre los distintos procesos. La parte del núcleo encargada de esta tarea es el gestor, planificador de procesos o *scheduler*, que asigna la CPU a cada proceso durante pequeños instantes de tiempo (quanta o rodajas de tiempo).

3.1. Estados de un proceso

Los procesos, en función de diversas variables, pasan por varios estados a lo largo de su ciclo de ejecución. El concepto de estado proviene de la *Teoría de Autómatas*, y hace referencia a la configuración de una entidad, en este caso, de un proceso. El concepto informático de estado es análogo al concepto popular de estado. Por ejemplo, una persona puede estar en estado *enfermo*, y eso implica que no puede realizar determinadas acciones, o que debe hacer otra serie de acciones, como guardar reposo, o tomar un medicamento. Además, para todos los estados posibles se definen una serie de transiciones posibles en forma de grafo.

Cada SO define un conjunto de estados en los que puede estar un proceso. Por lo tanto, los estados en los que puede estar un proceso en Unix son distintos de, por ejemplo, los estados en Windows. Ahora bien, existen una serie de estados que habitualmente consideran todos los SO, tal y como se refleja en la figura 1.

Hablando en términos generales, se puede decir que existen tres estados básicos que todo sistema operativo contempla de una u otra manera: *Ejecución*, *Bloqueado* y

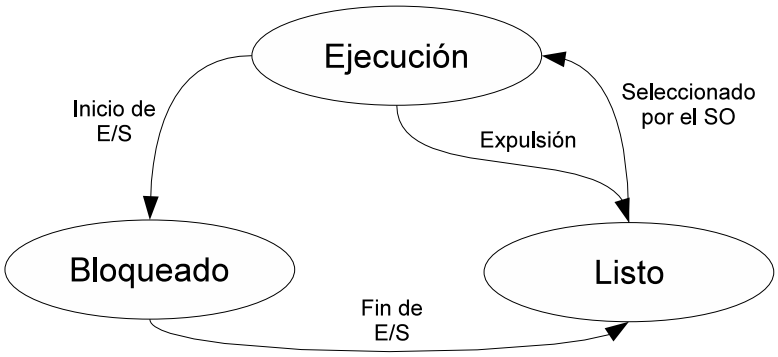

Figura 1: Estados elementales de un proceso.

Listo. Un proceso está en *Ejecución* cuando está ocupando la CPU, y permanece en este estado hasta iniciar una operación de E/S, en cuyo caso pasa a estar *Bloqueado*, o bien hasta que el SO determina que otro proceso diferente debe ocupar la CPU, en cuyo caso es expulsado y pasa a estar *Listo*. Por otra parte, un proceso *Bloqueado* es aquel proceso que está esperando a que finalice una operación de E/S, por ejemplo, porque está leyendo un dato del disco duro. Hasta que no finaliza la operación, el proceso no puede continuar su ejecución. Por ejemplo, si necesita leer del disco un número para calcular su raíz cuadrada, no puede hacer el cálculo hasta que no tenga el dato y, por lo tanto, no podrá utilizar la CPU hasta que deje de estar *Bloqueado*. Una vez que la operación de E/S termina, pasa a un estado de *Listo*, es decir, ya cumple todas las condiciones para poder utilizar la CPU, y sólo está a la espera de que el SO le permita pasar a ejecución.

En el caso concreto de Linux, existe un número mucho más alto de estados, y no se corresponden exactamente con el modelo simplificado anteriormente expuesto. De hecho, el número de estados existente es bastante elevado. Una simplificación de dichos estados puede verse en la figura 2. A continuación, se hace una descripción de cada uno de los posibles estados.

Activo. En UNIX se dice que un proceso se encuentra en estado *Activo* cuando se está ejecutando o cuando está listo para ejecutarse, es decir, abarca los estados de *Listo* y *Ejecución* que se vio en la figura 1.

Listo. Espera interrumpible; el proceso está esperando por la llegada de una señal. Cuando la señal llegue, el proceso pasará al estado *Activo*.

Esperando. Espera ininterrumpible; el proceso está esperando por algún recurso hardware o alguna operación de E/S.

Detenido. El proceso ha sido detenido momentáneamente, generalmente por una señal. Esto es útil, por ejemplo, en depuración de programas.

Zombie. El proceso ha finalizado, pero no se ha eliminado por parte del SO (eliminando su entrada de la tabla de control de procesos). El proceso ya no existe, pero deja

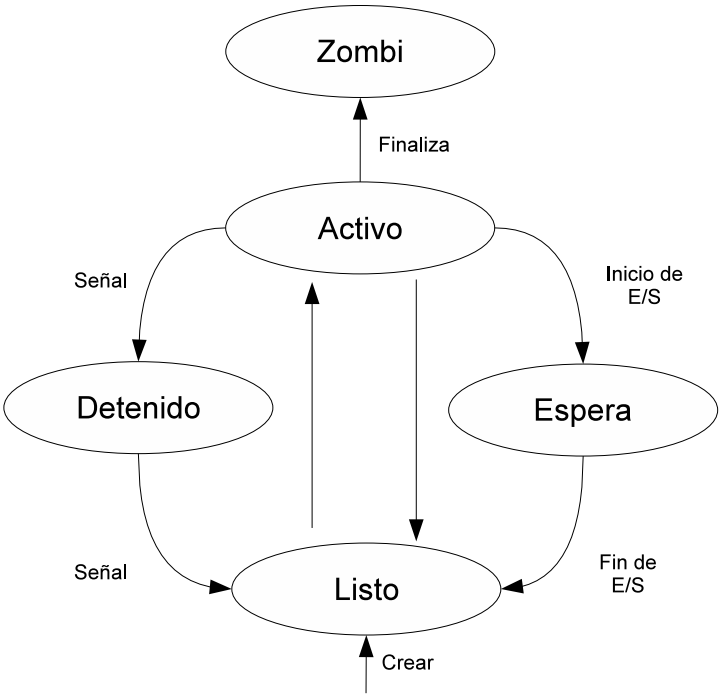

Figura 2: Resumen de los estados de un proceso bajo Linux.

para su proceso padre un registro que contiene el código de salida (estado de finalización) y algunos datos estadísticos tales como los tiempos de ejecución.

4. Inicio de la consola

Existen dos maneras de acceder la consola. La forma tradicional consistía en entrar en la consola por medio del proceso de *login*, es decir, al entrar en el sistema. El proceso *login* es el encargado de tomar los datos del usuario (nombre y contraseña) y validarlo con el sistema de autenticación. Normalmente, los datos de autenticación están guardados en el archivo */etc/passwd*, en donde se guardan los nombres de los usuarios del sistema, directorio de inicio, intérprete de órdenes y otra información. Una vez que *login* ha validado la información del usuario, arranca el intérprete de órdenes que está configurado. El proceso descrito era el habitual antes de la utilización de interfaces gráficas, y se sigue utilizando actualmente con asiduidad en servidores. Hoy en día, a nivel de usuario y no de administrador de sistemas, es más habitual acceder a la consola invocándola explícitamente desde la interfaz gráfica, en cuyo caso el intérprete de órdenes se ejecuta como cualquier otro programa.

La consola resultante de realizar un *login* se distingue de las consolas ordinarias porque comienza ejecutando las ordenes existentes en el archivo */etc/profile* y en *~/.profile*.

El archivo *profile* es un tipo de archivo muy utilizado en Unix llamado *shell script*[1].

[1] Los shell scripts contienen una secuencia de órdenes que son ejecutadas por el intérprete como si

En el archivo *profile*, el administrador del sistema sitúa una serie de órdenes de configuración comunes a todos los usuarios, mientras que el archivo *~/.profile* es empleado por cada usuario para personalizar su sesión según sus preferencias[2]. Además, si el intérprete es *bash*, se ejecutan una serie de archivos que contienen una secuencia de órdenes ubicados en *~/.bashrc* y */etc/bash.bashrc*. Al igual que en el caso anterior, el primero es específico de cada usuario mientras que el segundo se aplica a todos los usuarios del sistema.

4.1. Modos de ejecución

En muchas ocasiones es necesario ejecutar órdenes que necesitan mucho tiempo para finalizar. O bien, por ejemplo, son aplicaciones gráficas y, sencillamente, no tienen fin hasta que el usuario sale de la aplicación. En estos casos, lanzar una orden va a bloquear la utilización del terminal, impidiendo su utilización. Para evitar estas situaciones existe la posibilidad de ejecutar órdenes en segundo plano (o en *background*), es decir, se ejecutan de una manera no bloqueante. La mejor manera de entenderlo es por medio de un ejemplo. Desde un terminal, se puede ejecutar la orden *xeyes*. Observe qué le pasa al terminal, luego salga de la aplicación, y vea nuevamente el terminal. Ahora ejecute *xeyes* en segundo plano, poniendo al final el carácter "&", esto es, con la orden *xeyes &* y vea qué sucede.

Al ejecutar una orden en segundo plano, la *shell* muestra un número entre corchetes, conocido como identificador del trabajo, y el PID del proceso de más alta jerarquía de los que se ponen en marcha. La ejecución en segundo plano es muy interesante cuando se quiere realizar una tarea que requiere mucho tiempo y no se quiere permanecer a la espera como, por ejemplo, buscar un archivo en todo el sistema de archivos. Se utiliza mucho en administración de sistemas.

4.2. Entrada y salida estándar. Redirecciones

Las primeras aproximaciones a Unix pueden ser fustrantes. Hay muchas órdenes que es necesario conocer, y mayoritariamente son órdenes que realizan acciones muy sencillas, como visualizar un archivo o contar el número de líneas, que en un principio no parecen ser de gran utilidad. Es una situación similar a cuando en la película *Karate Kid* el aprendiz se dedica a "dar cera, pulir cera"; es frustrante porque no se aprecia todo el potencial que encierra una enseñanza, en principio, sin aplicación práctica. Sólo cuando se ve todo desde una determinada perspectiva empieza a cobrar algo de sentido, y uno se da cuanta de que, efectivamente, contar líneas en un archivo puede servir para realizar tareas de gran potencia y complejidad. El redireccionamiento tiene mucho que ver con esta potencia.

Uno de los principios que guiaron el diseño de Unix fue crear muchas herramientas de pequeño tamaño, muy especializadas en realizar una única tarea y -aquí está el secreto-, capaces de comunicarse entre ellas, es decir, de combinar sus funciones de manera efectiva para realizar tareas arbitrariamente complejas. Y éste es uno de los motivos

estuviera tecleado por el usuario.

[2]Conviene recordar que el punto inicial significa que el archivo es oculto y, por lo tanto, no se visualiza mediante la mera invocación de la orden *ls*; hace falta utilizar el modificador -a.

del éxito de Unix, y de su potencia, su capacidad de combinar distintas herramientas simples para realizar tareas complejas. Realizar esto en el terminal es sencillo, y está conseguido desde hace décadas, pero hacerlo por medio de interfaces gráficas es algo muy complejo, hasta el punto de que, a día de hoy, no hay soluciones satisfactorias. Gracias a la capacidad de redireccionar procesos y a las tuberías, que se estudiarán posteriormente, combinar procesos es una tarea sencilla.

Para comprender las redirecciones hace falta conocer algunos conceptos teóricos. Todos los procesos tienen asociados una serie de recursos, y entre ellos se encuentra el interfaz estándar de comunicación con el exterior. Este interfaz consta de tres archivos conocidos como archivos de entrada, salida y salida de errores estándar, de tal manera que la comunicación con el usuario se realiza por medio de dichos archivos. Hay que recordar que en Unix casi todo se representa mediante un archivo, y esto incluye a los dispositivos. Por lo tanto, desde la perspectiva de un proceso, escribir datos en un archivo regular es idéntico a mostrar algo por pantalla; en ambos casos se envían datos a un archivo.

En Unix, cuando un proceso abre un archivo, el núcleo (*kernel*) le devuelve un número entero positivo conocido como *descriptor de archivo*, que el proceso utilizará para identificar a dicho archivo. Todo proceso, cuando es arrancado, tiene asignado tres archivos por parte del SO, y siempre tienen los mismos descriptores (0, 1, y 2), tal y como se resume en la siguiente tabla.

Descriptor	Archivo	Dispositivo	Descripción
0	**stdin** (entrada estándar)	Teclado del terminal por defecto	Entrada de datos estándar que utiliza el proceso
1	**stdout** (salida estándar)	Pantalla del terminal por defecto	Salida de datos estándar que utiliza el proceso
2	**stderr** (salida de errores estándar)	Pantalla del terminal por defecto	Salida de datos que utiliza el proceso en caso de situaciones anómalas

Por lo tanto, cuando un proceso necesita leer datos, por lo general lo hará por la *entrada estándar*, que tiene asociado el descriptor 0 y por defecto corresponde al teclado. De la misma manera, cuando un proceso necesita mostrar información, generalmente lo hará por medio de la *salida estándar*, con descriptor 1, y asociado al monitor. Por último, la *salida de errores* se realiza mediante el descriptor 2 y está asociado también al monitor.

Parte de la belleza de Unix radica en que estos descriptores tienen asociados unos archivos por defecto, pero se pueden cambiar con extrema sencillez por medio de las redirecciones. Existen tres tipos de redirecciones: redirección de entrada (<), redirección de salida (>, ») y redirección de errores (2>).

- **Redirección de la entrada estándar (<)**

  Este operador permite redirigir la entrada de datos de un proceso desde un archivo, sustituyendo así al teclado.

Ejemplo: La siguiente orden mostrará el contenido del archivo */etc/passwd* ordenado alfabéticamente:

```
$ sort < /etc/passwd
```

- **Redirección de la salida estándar (>, >>)**

  Estos operadores permiten redirigir la salida de un proceso a un archivo en lugar de la pantalla. El primero de ellos , >, redirige **eliminando** el contenido anterior del archivo, y el segundo, >>, redirige **añadiendo** la salida del proceso al final del archivo. En ambos casos, el archivo se crea si no existía.

  Ejemplo: La salida de la orden `cat` no es enviada al terminal sino que será almacenada en el archivo *saludo.txt*.

```
$ cat hola > saludo.txt
```

- **Redirección de la salida de errores estándar (2>)**

  A través de este operador se puede redirigir la salida de errores a un archivo. Conviene fijarse que esta redirección está compuesta por el signo de la redirección de salida, >, junto con el número 2, que corresponde con el descriptor del archivo de errores estándar.

  Ejemplo: Si se intenta mover un archivo que no existe, se generará un error. El mensaje de error se envía a la salida estándar de errores, que puede ser redireccionado mediante la siguiente orden.

```
$ mv noexisto.txt fichero.txt 2> salida.txt
```

Los operadores de redirección son evaluados de izquierda a derecha, siendo el orden de las redirecciones importante. Las redirecciones cobran especial importancia en la realización de *shell scripts*, es decir, programas que contienen órdenes del intérprete de órdenes, y que son muy útiles en la administración de sistemas. Los archivos *profile* y *bashrc*, vistos anteriormente, son ejemplos de *shell scripts*.

Aparte de las redirecciones existe otra serie de delimitadores para modificar diversos aspectos de la ejecución de una instrucción. Uno de los operadores más sencillos es el punto y coma (;), que permite ejecutar secuencias de órdenes una detrás de otra. Ejemplo:

```
$ sleep 20; echo Han pasado 20 segundos
(20 segundos después aparecerá el siguiente mensaje:)
Han pasado 20 segundos
$
```

Ejercicios

1. Guardar un listado del contenido del directorio de inicio en el archivo *listado.txt*.
2. Visualizar el contenido del archivo recién creado.

3. Añadir al archivo anterior el contenido del directorio */etc*.

4. Visualizar el contenido de *listado.txt* alfabéticamente. Para ello, utilizar la orden *sort*, con y sin redirecciones.

4.3. Variables de entorno y variables shell

El terminal incorpora muchos elementos propios de un lenguaje de programación[3], entre otros elementos define lo que se conoce como *variables de entorno* y *variables shell*. Su importancia radica en que diversos elementos importantes de la configuración del intérprete se realizan por medio de estas variables. Por ejemplo, la ubicación de los directorios en donde el intérprete buscará los ejecutables se configura por medio de una variable, PATH. Se puede visualizar el contenido de una variable por medio de la orden *echo*, por ejemplo, para ver el contenido de PATH habría que ejecutar `echo $PATH`.

Una cuestión importante a tener en cuenta es que el intérprete diferencia el nombre de la variable del contenido de la variable. El contenido de la variable se especifica por medio del signo de dólar ($), por esta razón, en el ejemplo anterior se pone `echo $PATH`. Si no se pusiera, el intérprete consideraría que PATH es una cadena de texto, y asi la mostraría por pantalla.

4.3.1. Variables shell

Caracterizan al intérprete y, a diferencia de las variables de entorno, no son heredadas por sus subprocesos. Funcionan como variables dentro de una instancia del intérprete. Para definir una variable shell basta con teclear:

```
identificador=valor
```

Ejemplo:

```
$ DIA=Viernes
```

No deben teclearse espacios en blanco en torno al operador de asignación (=). Se puede obtener un listado de todas las variables (shell y entorno) tecleando *set* sin argumentos. Una variable shell puede transformarse en una variable de entorno mediante la orden interna *export*.

4.3.2. Variables de entorno

Caracterizan el entorno de trabajo del intérprete, y los subprocesos que genere (puesto que se heredan de padres a hijos). Sus valores se fijan mediante los siguientes mecanismos:

/etc/profile Archivo en el que se definen las variables de entorno comunes a todos los usuarios.

[3] Estrictamente hablando, el intérprete define un lenguaje de programación interpretado, con todas las construcciones clásicas de un lenguaje de programación: variables, bucles, funciones, etc. Este hecho es lo que permite la realización de *shell scripts*, y proporciona a la *shell* una gran potencia.

Variable	Descripción
HOME	Contiene el camino absoluto del directorio de conexión del usuario
PWD	Contiene el camino absoluto del directorio de trabajo
PATH	Contiene el conjunto de caminos empleados para buscar las órdenes a ejecutar. Se pueden especificar múltiples caminos separándolos mediante :
HOSTNAME	Nombre de la máquina en donde se está ejecutando el intérprete
USER	Nombre del usuario
UID	Número identificador del usuario
PS1	El valor de esta variable se utiliza como inductor de órdenes primario (*prompt*) del intérprete de órdenes
PS2	El valor de esta variable se utiliza como inductor de ordenes secundario (por ejemplo, cuando se introducen bucles) del intérprete de órdenes
SHELL	Contiene el camino absoluto de la *shell*
?	Valor decimal retornado por la última orden ejecutada; si es 0 indica que la orden se ejecutó correctamente, mientras que si es 1 indica que hubo algún tipo de error. Si es mayor que 1 significa que el proceso terminó como consecuencia de haber recibido una señal (se explicará más adelante).

Cuadro 1: Variables de entorno.

.profile Si en el directorio de conexión del usuario existe este archivo, el intérprete lo ejecuta. Mediante este mecanismo cada usuario puede personalizar su entorno.

ENV variable de entorno que contiene el nombre del último archivo de configuración que se ejecutará. Suministra un mecanismo adicional para que los usuarios personalicen su entorno. Habitualmente no se suele fijar o, si se fija, se hace al archivo `.bashrc` del directorio de conexión del usuario.

Variables de entorno hay muchas, pero las principales vienen descritas en la tabla 1.

Ejercicios

1. Modificar el *prompt* de la *shell* para que muestre la cadena *éste es mi prompt*.
2. Crear tres variables, VAR1, VAR2 y VAR3, inicializar su valor con “Buenos dias”, “14” y “Linux mola”.
3. Visualizar el contenido de las variables creadas en el ejercicio anterior.
4. Abrir un nuevo terminal y visualizar el contenido de VAR1, VAR2 y VAR3. ¿Qué sucede?

5. Modificar el archivo *.bashrc* para que muestre un mensaje de bienvenida[4].

6. Modificar el archivo *.bashrc* para que muestre un mensaje de bienvenida y además se informe al usuario del directorio en que se encuentra su *home*. Para ello, hay que visualizar el contenido de la variable de entorno adecuada.

4.4. Comunicación entre procesos mediante tuberías

Como se comentaba antes, uno de los elementos que caracterizan la filosofía de Unix es tener un gran número de órdenes que implementan funcionalidades muy específicas, y por lo general, poco útiles por sí mismas. Ahora bien, dichas órdenes hacen una magnífica implementación de sus funciones y, lo más importante, Unix facilita una serie de herramientas sencillas y extremadamente potentes para que dichas órdenes se comuniquen entre sí. De este modo, se pueden realizar funciones muy complejas con una relativa sencillez, como si se construyera un puzle. Ya se ha visto una manera de realizar esto, las redirecciones. No obstante, existen otros mecanismos que en algunas circunstancias suelen ser más apropiados, especialmente las *tuberías*.

Las tuberías, o *pipelines*, son una herramienta de comunicación entre procesos que permiten conectar la salida estándar de un proceso con la entrada estándar de otro. Así pues, todo lo que el primer proceso productor envíe a su salida estándar será tomado como entrada por el segundo proceso consumidor y en el mismo orden en que se generó la información.

La forma de indicar que se quiere emplear una tubería para comunicar dos o más procesos es la siguiente:

```
$ orden_1 | orden_2 |orden_3 | ...
```

Ejemplo:

```
$ ls | wc -w
```

La salida de *ls* es introducida a *wc*, que cuenta el número de palabras que genera *ls*. Es decir, el resultado de ejecutar la orden del ejemplo es el número de archivos existentes en el directorio de trabajo.

Las tuberías y las redirecciones pueden ser utilizadas para comunicar dos procesos, si bien la utilización de tuberías hace que dicha comunicación sea más sencilla. El ejemplo anterior se podría realizar por medio de tuberías de la siguiente manera:

```
$ ls > temp
$ wc -w < temp
$ rm temp
```

que es algo más larga que utilizando tuberías, además de incómodo y poco eficiente. Esto es especialmente importante cuando queremos comunicar más de dos procesos.

[4] El archivo *.bashrc* es en realidad un *shell script* que todavía no ha sido explicado. Sin embargo, debería ser capaz de hacer la tarea descrita de manera intuitiva examinando el contenido del archivo.

Tuberías y redirecciones se pueden, y de hecho se suelen, juntar, aportando una extraordinaria potencia a la línea de órdenes. Por ejemplo:

```
$ cat /etc/passwd | cut -f 1 -d ‘‘:’’ | grep ‘‘^l’’ | sort | uniq > usuarios.txt
```

La orden *cat* pasa el contenido del archivo */etc/passwd* a la orden *cut*, que divide cada línea utilizando el carácter ":", y saca el segundo campo. La salida de `cut` es procesada por *grep*, que filtra aquellas líneas que empiezan por `l`, y posteriormente *sort* ordena alfabéticamente el resultado de `grep`. Finalmente, la orden *uniq* elimina las líneas duplicadas de su entrada estándar, y el resultado lo vuelca en el archivo *usuarios.txt*. En definitiva, lo que hace esa orden es obtener un listado ordenado alfabéticamente de los usuarios del sistema, y lo guarda en el fichero `usuarios.txt`. La orden anterior sería equivalente a la siguiente:

```
$ cut -f 1 -d ‘‘:’’ < /etc/passwd | grep ‘‘^l’’ | sort > usuarios.txt
```

Y esto nos lleva a otra característica importante de Unix (y de la informática en general): no existe una única manera de hacer las cosas, de hecho, suele haber muchas, y todas igualmente válidas.

Ejercicios

1. Ejecutar la siguiente orden e indicar por qué el resultado de ejecutarla es `10`:

   ```
   head /etc/passwd | wc -l
   ```

2. Indique qué hace la siguiente orden:

   ```
   users | wc -w
   ```

3. Contar el número de archivos que hay en el directorio de inicio.
4. La orden `find /` permite visualizar todos los archivos que hay en el sistema, salvo aquellos contenidos en directorios para los que no hay permiso de lectura. Por otra parte, la orden `grep jpg` filtra todas las líneas de la entrada estándar que contienen la cadena "jpg". Utilizando dichas órdenes, visualizar todos los archivos del sistema que contengan el texto `jpg`.
5. Modificar la orden anterior para que sólo busque archivos dentro del directorio de trabajo. Hágalo de, al menos, dos maneras diferentes.
6. Modificar la orden del ejercicio tres para que no se muestren los mensajes de error. Sugerencia: Utilizar el archivo */dev/null*, que sirve como "papelera", es decir, cualquier dato que se escriba en él es desechado.
7. Crear un archivo que contenga el listado recursivo (en formato largo) de todos los archivos y directorios que hay en su directorio de conexión.
8. Existe un paquete software muy útil llamado *imagemagick*, que permite la edición de imágenes, no interactiva, dentro de la consola. Justificar qué utilidad puede tener el hecho de que este paquete no sea interactivo. Sugerencia: buscar en internet más información relativa a este software.
9. Utilizar el programa *convert*, contenido en el paquete *imagemagick*, para reducir a la mitad de tamaño todas las imágenes que empiecen por 'a' de un determinado directorio.
10. La orden `grep` se utiliza muy a menudo junto a una expresión regular. Utilizar Internet y la bibliografía recomendada para averiguar qué es exactamente una expresión regular. Intentar crear una expresión regular capaz de extraer una fecha con el formato `dd/mm/aaaa`.

5. Monitorización de procesos

Las órdenes más comúnmente utilizadas relacionadas con la gestión de procesos son aquéllas que nos permiten conocer qué procesos se están ejecutando en el sistema y los datos fundamentales sobre los mismos, como la cantidad de memoria que están consumiendo. Las dos órdenes clásicas utilizadas para estas cuestiones son `ps` (no interactivo) y `top` (interactivo). A continuación, se muestran los principales argumentos que admite `ps`.

ps [opciones] Lista los procesos en curso y muestra información sobre ellos. Sin opciones, muestra los procesos activos del usuario, con su PID, terminal asociado (TTY), tiempo de ejecución acumulativo del proceso (TIME) y nombre de la orden que genera el proceso (CMD). Algunas opciones interesantes son:

[-f] Muestra información más completa sobre los procesos, añadiendo a la información anterior el identificador del usuario (UID), PPID, procesador utilizado para la programación de tareas (C) y hora de inicio de la orden (STIME).

[-e] Muestra información sobre todos los procesos activos en el sistema, y no sólo los del usuario.

[-S] Muestra el estado en que se encuentran los procesos (STAT). Puede ser uno de los siguientes: R (*activo*), S (*listo*), T (*detenido*), D (*esperando*) o Z (*zombi*).

[-a] Restringe la información a los procesos asociados al terminal.

Ejemplo:

```
$ ps -S
PID      TTY      STAT    TIME    COMMAND
1201   pts/0   S          0:00     bash
1484   pts/0   R          0:00     ps -S
```

La orden `top`, por el contrario, al ser una orden interactiva, no suele utilizarse con argumentos. Se puede salir de `top` utilizando la tecla *q*. Otra orden que puede ser útil es `time`, que permite medir el tiempo que tarda en ejecutarse un proceso.

time [opciones] orden [argumentos] Ejecuta la orden con los argumentos especificados y muestra por la salida de errores estándar: el tiempo transcurrido en el sistema mientras se ejecutaba la orden (`real`), el tiempo consumido en la ejecución de la orden (`user`) y el tiempo empleado por el sistema para ejecutar la orden (`sys`). Ejemplo:

```
$ time sleep 5
real 0m5.003s user 0m0.000s sys 0m0.002s
```

Ejercicios

1. Obtener todos los procesos que se están ejecutando en el sistema. Observar qué usuario lanzó el proceso y el PID asociado al proceso.
2. ¿Cuál es el proceso cuyo PPID es 0? Indique con qué línea de órdenes consigue visualizar dicho proceso y su PPID. Indique la función de dicho proceso. Sugerencia: Utilizar las páginas de `man` e Internet.
3. Ejecutar *sleep 3000 &* y observar los procesos que se están ejecutando mediante la orden `ps`. Volver a ejecutar la misma orden y visualizar los procesos en ejecución con `ps`. ¿Qué diferencias observa entre el resultado del primer y segundo `ps`?

4. Supongamos que no se sabe cuál es el PID del proceso lanzado en *background* en el ejercicio anterior. Con esta suposición, y con la restricción de no poder utilizar la orden `ps` para averiguar su PID, indique cómo se podría eliminar el proceso.
5. La orden `time` da tres tiempos, a los que llama *real*, *user*, y *sys*. Averiguar qué significa cada uno de estos tiempos.
6. Mediante el uso de la orden `time`, averiguar el tiempo que tarda en ejecutarse la orden `sleep 5`.
7. ¿Con qué opción de qué orden visualizaría el estado de sus procesos?

6. Señales

A veces es necesario enviar información asíncrona a los procesos que se están ejecutando. Por ejemplo, podemos lanzar una orden que tarda mucho en ejecutarse y, mientras se ejecuta, surge la necesidad de forzar la terminación de dicha orden. Para este tipo de comunicación, que no se inscribe dentro de la ejecución secuencial y determinista de un programa, están las señales. Visto de otra manera, las señales son eventos gestionados no a nivel de hardware, sino a nivel del SO, de hecho, eventos hardware pueden convertirse en señales.

Una señal es una herramienta software que sirve para indicar al proceso que la recibe que se ha producido algún evento significativo ante el cual debe responder. Un proceso puede recibir señales generadas por el propio núcleo del SO, por otros procesos, o por los propios usuarios. Existen dos órdenes básicas a nivel del intérprete de órdenes para el manejo de señales: `kill`, que permite enviar señales, y `trap`, que "las recibe"[5].

kill [-num_de_señal] PID o kill -l [señal] Permite enviar señales a procesos identificados por su PID. Se utiliza mucho para matar procesos que no responden[6].

Si no se indica el `num_de_señal` se envía la señal número `15`.

Si `PID` = 0, significa que todos los procesos del grupo de procesos actual quedan señalados; un ejemplo de grupo de procesos son todos los procesos ejecutados con una misma orden.

Con la opción [-l], muestra información sobre el nombre de las posibles señales a enviar y su número correspondiente.

Algunas señales habituales son:

2: SIGINT Interrupción. Se genera al pulsar Ctrl+C.

9: SIGKILL Muerte segura. No puede ser ignorada; se utiliza para matar procesos que se niegan a "suicidarse".

15: SIGTERM Finalización de proceso (software). Es similar a la 9, pero puede ser ignorada.

Ejemplo:

```
$ sleep 2000 &
[1] 1986
$ ps -S
PID      TTY      STAT  TIME   COMMAND
```

[5] Hablando con más propiedad habría que decir que la orden `trap` indica cómo actuar ante una señal.

[6] Existe una versión muy útil de `kill` para el entorno gráfico denominada `xkill`. Se ejecuta o desde un terminal o ejecutando manualmente una orden (Alt+F2).

```
1201   pts/0   S        0:00   bash
1986   pts/0   S        0:00   sleep 2000
1987   pts/0   R        0:00   ps -S
$ kill -9 1986
$ ps -S
PID       TTY    STAT   TIME  COMMAND
1201    pts/0  S         0:00   bash
1988    pts/0  R         0:00   ps -S
[1]+ Terminado (killed) sleep 2000
```

trap [orden o secuencia_de_órdenes] [señales] Ejecuta la `orden` o `secuencia_de_órdenes` cuando recibe las señales desde un procedimiento o programa. Por ejemplo, una línea con `trap` en un *shell script* haría que el resto de las líneas del programa se ejecutaran de forma normal hasta que se reciba la señal indicada en dicha línea, momento en el que se ejecutarán la orden u órdenes indicadas en `trap`.

Ejemplo:

```
echo Esto es un programa shell script
# En un programa shell script las líneas de comentario van
# precedidas por el carácter #
líneas de programa ...
# Las líneas anteriores se ejecutarán siempre, puesto que van
# antes del trap.
trap 'orden' 15
Más líneas de programa ...
```

En el ejemplo anterior, las primeras líneas de programa se ejecutarán siempre. A continuación, empezará a ejecutarse el segundo bloque de líneas, pero se interrumpirá si se recibe la señal 15, en cuyo caso se ejecutarán las órdenes incluidas en la línea de la orden `trap`.

Ejercicios

1. Ejecutar una orden larga en segundo plano (por ejemplo, `sleep 3000 &` o bien, `xeyes &`). Después, obtener el PID de dicho proceso y enviarle una señal SIGKILL. Por último, verificar que el proceso ha finalizado.
2. Utilizar las órdenes necesarias para que cuando se pulse `Ctrl+C` en el terminal, se visualize por pantalla el mensaje "Has pulsado Ctrl+C"[7].
3. Modificar el apartado anterior para que el texto que se visualice sea la ruta del directorio de trabajo actual.
4. Explicar qué ocurre en los siguiente casos:
   a) Crear una *subshell* (tecleando `bash`).
   b) Enviar la señal 15 a esa *subshell*.
   c) Repetir el proceso pero enviando la señal 9.
   d) Enviar una señal de terminación a un proceso de otro usuario.
   e) Enviar la señal 15 al proceso padre de la *shell* que le atiende.

[7] Para que la *shell* interprete varias palabras separadas como un único texto se utilizan las dobles comillas.

5. ¿Cómo protegería un *shell script* para que ignorase la señal 2 (SIGINT)? Indicar qué línea debe sustituir a la línea de puntos suspensivos del programa que se muestra a continuación para que, en caso de que el proceso *unixados* reciba la señal 2 (SIGINT), muestre el mensaje "Adiós" y finalice. No es preciso conocer *shell scripts* para realizar esta pregunta (utilizar la imaginación). Para poder ejecutar el *script* mostrado a continuación es necesario proporcionarle permiso de ejecución.

```
[lsoi13t3@Mordor lsoi13t3]$ vi unixados
# El script unixados crea una copia de todos los archivos del
# directorio actual en un disquete con formato D.O.S. (si
# estás interesado, consulta mtools en man).
.....................................
for ARCHIVO in `ls`
do
if
     test -r $ARCHIVO
     then
         echo `pwd`/$ARCHIVO
         mcopy $ARCHIVO a:$ARCHIVO
     fi
done
mdir a:
exit 0
```

7. Filtros

Cualquier proceso que lea de su entrada estándar y escriba en su salida estándar se denomina *filtro*. Un ejemplo de lo anterior es `cat`. Esta orden sin argumentos toma la información del teclado y cuando finaliza su entrada escribe su salida en pantalla. En Unix la marca de fin de archivo se representa con Ctrl+D (^d). El nombre de filtro proviene del hecho de que, esencialmente, funcionan como filtros: obtienen los datos de entrada de su entrada estándar, de alguna manera los procesan, y generan una salida a través de la salida estándar. Los filtros normalmente se utilizan junto con tuberías para realizar tareas complejas.

7.1. wc

Sintaxis: `wc [-lwc] [archivo1 ...]`

`wc` cuenta el número de caracteres, palabras y líneas por cada archivo que recibe como parámetro y el total de cada una de las categorías para todos los archivos. Si no se especifica ningún parámetro, `wc` realiza estas operaciones sobre el archivo de entrada estándar. Las opciones posibles limitan el tipo de información generada:

-c cuenta únicamente el número de caracteres.

-l cuenta sólo el número de líneas.

-w cuenta sólo el número de palabras.

Ejemplo de uso de `wc`:

```
$ echo Tres tristes tigres ... > xxx
$ echo comian trigo en un trigal > yyy
```

```
$ wc -w xxx yyy
4 xxx
5 yyy
9 total
$
```

7.2. tee

Sintaxis: `tee [-a] [archivo1 ...]`

`tee` transcribe su entrada estándar a su salida estándar copiándola además en los archivos especificados. La opción `-a` permite añadir la entrada al archivo (o archivos) en lugar de sobreescribirlos.

Ejemplo de uso de `tee`:

```
$ who | tee usuarios
luis pty/ttys0 Nov 10 09:08
$ cat usuarios
luis pty/ttys0 Nov 10 09:08
$
```

7.3. sort

Sintaxis: `sort [archivo1 ...]`

`sort` ordena las líneas de los archivos `archivo1 ...` y las envía a la salida estándar. Si no se especifica ningún archivo o uno de los archivos es `-`, entonces toma las líneas de la entrada estándar. Para la ordenación se emplean uno o varios campos claves extraídos de cada línea. Por defecto, el campo empleado como clave es la línea completa.

7.4. cut

Sintaxis: `cut -f lista [-d carácter] [-s] [archivo1 ...]`
Sintaxis: `cut -c lista [archivo1 ...]`

`cut` extrae uno o varios campos para cada línea de un archivo. Los campos a extraer se especifican en lista y pueden ser de longitud variable o fija.

En el caso de la primera sintaxis, es necesario especificar un carácter delimitador de campo. Si no se especifica `archivo1 ...`, se utiliza la entrada estándar. El significado de las opciones es el siguiente:

lista Lista de números enteros que se emplean para especificar los campos a extraer. Los elementos de la lista deben separarse por comas y ordenarse en orden creciente. Se pueden especificar rangos empleando el símbolo `-`. Ejemplo:

1, 3, 7 selecciona los campos 1, 3 y 7.

1-5, 8 selecciona los cinco primeros campos y el 8. Este rango se puede especificar también como -5,8.

2-5, 8 selecciona los campos 2 a 5 (ambos incluidos) y todos los campos desde el 8 al último.

-c Especificaciones de campos que aparecen en la lista. Se refieren a las posiciones que ocupan los caracteres dentro de la línea. Cada carácter es un campo.

-f Los campos se encuentran separados por delimitadores por lo que un campo puede contener varios caracteres. El carácter delimitador puede ser especificado por la opción `-d`. Las líneas sin caracteres delimitadores se transcriben intactas a la salida estándar salvo que se especifique la opción `-s`.

-d carácter Permite especificar el carácter delimitador de campo. Si no se especifica esta opción, se toma como delimitador por defecto el carácter de tabulación. Para especificar el espacio en blanco o caracteres especiales de la *shell*, es necesario encerrarlos entre comillas. Dos caracteres delimitadores consecutivos delimitan un campo vacío.

-s Con esta opción se eliminan de la salida estándar las líneas que no contengan caracteres delimitadores.

Ejemplos de uso de `cut`:

```
$ who | cut -c 1-9
luis
tracy
root
$ cat agenda
Dick Tracy:tracy@pareatis.lightwind.es
Sharleen Spiteri:sharleen@texas.com
archie:archie@quiche.cs.mcgill.ca
$ cut -f 2 -d':' agenda
tracy@pareatis.lightwind.es
sharleen@texas.com
archie@quiche.cs.mcgill.ca
$
```

7.5. grep

Sintaxis:

```
grep [-ilnsv] patron1 [archivo1 ...]
grep [-ilnsv] [-f archivo_patrones] [archivo1...]
grep [-ilnsv] -e patron1 [-e patron2 ...] [archivo1 ...]
```

Busca los patrones de texto especificados por `patron1`, `patron2`, ... en los archivos `archivo1`, ... Normalmente, cada línea emparejada se envía a la salida estándar, precedida por el nombre del archivo en que se encontró, si se especificaron varios. Si no se especifica ningún archivo, busca los patrones en la entrada estándar.

`grep` dispone de una amplia gama de opciones. A continuación, se detallan las más habituales:

-f archivo_patrones Esta opción permite especificar un nombre de archivo del que se tomarán los patrones a emparejar. Los patrones que se deseen buscar han de escribirse cada uno en una línea dentro de `archivo_patrones`. Una línea de un archivo se envía a la salida estándar si se empareja con cualquiera de los patrones.

-e patrón Especifica múltiples patrones en la línea de órdenes.

-i Trata mayúsculas y minúsculas sin distinción durante la comparación; así, el patrón `la` se emparejará con las cadenas “la”, “La”, “lA” o “LA”.

-l Limita la salida de la orden a los nombres de los archivos que contienen líneas emparejadas con algún patrón.

-n Cada línea emparejada aparece precedida por su posición dentro del archivo. Esta opción se ignora si está activa la anterior.

-s No se muestra ningún tipo de error que pudiera producirse.

-v Se muestran sólo las líneas que no se emparejan.

Los patrones pueden contener caracteres especiales por lo que, para evitar que sean interpretados por la *shell*, conviene introducir los patrones entre comillas.

Ejemplo de uso de la orden `grep`:

```
$ cat agenda
Dick Tracy:tracy@pareatis.lightwind.es
Sharleen Spiteri:sharleen@texas.com
archie:archie@quiche.cs.mcgill.ca
$ grep -ni spiteri agenda
2:Sharleen Spiteri:sharleen@texas.com
$ echo archivo con cuatro palabras > f1.txt
$ echo un archivo con cinco palabras > f2.txt
$ echo otro archivo con cinco palabras > f3.txt
$ grep -l cinco *.txt
f2.txt
f3.txt
$
```

En el mundo UNIX, un patrón no es únicamente una cadena de caracteres con la que emparejarse, sino que puede contener una expresión regular.

Ejercicios

1. Mostrar por orden alfabético todos los usuarios que hay en el sistema.
2. Mostrar por orden alfabético el contenido del directorio `/etc`.
3. Copiar el archivo `/etc/services` en el directorio de trabajo. Dicho archivo contiene un listado de los distintos servicios existentes y los puertos[8] asociados. Sin visualizar la totalidad del archivo, es decir, filtrando su contenido, realice las siguientes tareas:

   a) Mostrar todos los servicios que se basan en TCP[9].

   b) Mostrar todos los servicios que se basan en UDP.

   c) Averiguar qué servicio se aloja en el puerto 80.

   d) Averiguar en qué puerto se accede al protocolo SMTP (correo electrónico saliente).

   e) Contar el número de servicios que hay registrados en el archivo `services`. Ejercicio extra: contar los servicios SIN tener en cuenta las líneas que empiezan por almohadilla (#).

[8] Un puerto es un número que identifica un servicio. Por ejemplo, si queremos acceder a una página web, necesitamos saber la dirección donde está esa página, que viene dada por la dirección IP o la URL, pero la máquina que identifica la IP contiene varios servicios, por ejemplo, además de web, puede tener correo electrónico. Mediante el puerto se indica a qué servicio específico queremos acceder.

[9] TCP es uno de los protocolos fundamentales que se utilizan en Internet. Los servicios se pueden prestar bien por TCP o bien por UDP.

4. Mostrar en pantalla una lista ordenada de los archivos del directorio actual y, al mismo tiempo, haga que esta información se almacene en un archivo llamado `lista`.
5. Utilizando el archivo `/etc/passwd`, obtener el número de usuarios que tienen como intérprete de órdenes `/bin/bash`.