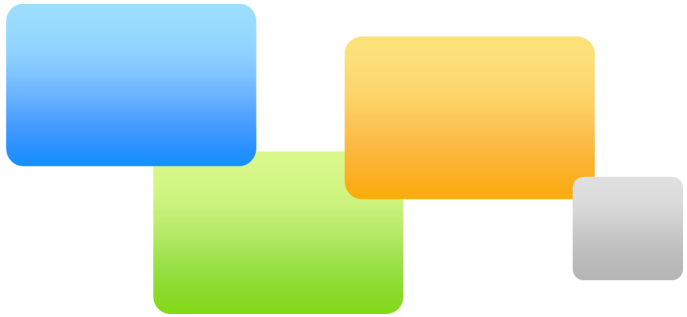
Administración de Sistemas Operativos Linux

Certificación LPIC-1



Rafael Rodríguez Gaioso
CTI Tegnix S.L. Vigo (Spain)

Usted es libre de:

 copiar, distribuir y comunicar públicamente la obra

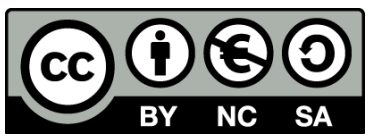
 hacer obras derivadas

Bajo las condiciones siguientes:

 **Reconocimiento** - Debe reconocer los créditos de la obra de la manera especificada por el autor o licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).

 **No comercial** - No puede utilizar esta obra para fines comerciales.

 **Compartir bajo la misma licencia** - Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.



Tema 3 – ¿Quién le teme a la shell?

Contenidos:

- ¿Que es la shell?
- Comandos.
  - Tipos y estructura
- Variables de shell y el entorno
- Comandos simples: sleep, echo y date
- La shell como herramienta
- Más sobre la línea de comandos

Objetivos:

- Apreciar las ventajas de la línea de comandos
- Trabajar con los comandos de Bash
- Comprender la estructura de los comandos Linux
- Manejar procesos en primer y segundo plano

¿Qué es la shell?

A diferencia de otros sistemas, Linux se basa en la idea de introducir comandos a través del teclado. Puede suponer un 'shock' para los usuarios de otros sistemas, y que han aprendido a usar un equipo a través de la interfaz gráfica.

Es como situar a una persona del siglo XXI en la corte del Rey Arturo !

Sin embargo, también existen buenas interfaces gráficas en Linux, algo que tampoco excluye el uso de la línea de comandos.

No se trata de evitar el uso de la interfaz gráfica, sino de aprovechar todas las '*marchas*' del sistema.

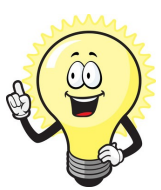
¿Qué es la shell?

Los usuarios no se pueden comunicar directamente con el kernel del sistema operativo. Se realiza a través de programas que realizan “llamadas al sistema”.

La función de la shell es iniciar dichos programas.

Lee comandos desde el teclado y los interpreta como comandos a ser ejecutados.

De este modo, la shell actúa como una interfaz del sistema, y lo protege como una “*shellfish*”.



Incluso los 'escritorios' gráficos actuales, como KDE pueden ser considerados 'shells'. Leen comandos gráficos a través del ratón. Seleccionamos objetos haciendo clic y luego realizamos una operación.

Diferentes shell's

El primer sistema Unix ya tenía una shell (años 70). La más antigua es la desarrollada para “Unix version 7” por Stephen L. Bourne, llamada “Bourne shell”.

Otra shell clásica es la C shell, basada en el lenguaje C y desarrollada en Berkeley.

La shell estándar en los sistemas Linux es la Bourne-Again Shell, o ***bash***. Fué desarrollada dentro del proyecto GNU por Brian Fox y Chet Ramey.

Las shells se usan a través de 'terminales' donde poder introducir comandos. También es posible leer comandos desde un fichero (*shell scripts*).

¿Qué es la shell?

Una shell ejecuta los siguientes pasos:

1. Lee un comando desde la terminal (o fichero)
2. Valida el comando
3. Ejecuta el comando o inicia el programa correspondiente
4. Muestra el resultado por pantalla (u otro lugar)
5. Continúa en el punto 1

Además, incluye características de un lenguaje de programación, como bucles, condiciones y variables.

Las sesiones de shell suelen finalizarse a través del comando `exit`.

¿Qué es la shell?

Si hay varias shells disponibles en el sistema, podemos usar los siguientes comandos para cambiar entre ellas:

sh para la shell clásica de Bourne -suele referirse a **bash**.
bash para la shell Bourne-again.
ksh para la shell de Korn.
csh para la shell de C.
tcsh para la shell "Tenex C"- **csh** suele referirse a **tcsh**.

En caso de no recordar la shell que se está ejecutando, el comando "`echo $0`" nos mostrará el nombre de la shell actual.

Ejercicios

1. ¿Cuántas shells diferentes hay instaladas en tu sistema?

2. Comprueba la salida del comando "`echo $0`". Inicia una nueva shell usando el comando "`bash`" y vuelve a introducir "`echo $0`". Compara la salida de los dos comandos.

Comandos

Un ordenador, sin importar el sistema operativo que utilice, funciona del siguiente modo:

1. Espera por una entrada del usuario
2. El usuario selecciona un comando y lo introduce a través del teclado o ratón
3. El ordenador ejecuta el comando

En Linux, la shell muestra un 'prompt', indicando que se pueden introducir comandos. Suele estar formado por los nombres de usuario y host, el directorio actual y un carácter final:

```
user@host:/home> _
```

Estructura de Comandos

Un comando es una secuencia de caracteres que finalizan pulsando la tecla INTRO, y es evaluada por la shell. Suelen estar inspirados en el idioma inglés y deben seguir una sintaxis.

Para interpretar una línea de comando, la shell primero intenta dividir la línea en palabras (separadas por espacios). La primera palabra suele ser el comando actual. El resto de palabras son parámetros para indicar en detalle la acción a realizar.

A diferencia de DOS y Windows, la shell distingue entre minúsculas y mayúsculas. Los comandos en Linux suelen escribirse en minúsculas.

Estructura de Comandos

A la hora de dividir un comando en palabras, un sólo espacio equivale a muchos. También se admite el uso de tabuladores, siempre que lo hagamos dentro de un fichero script.

También podemos usar el fin de línea para distribuir un comando largo a través de varias líneas de entrada, usando el carácter "\" justo antes para evitar que la shell lo interprete como salto de línea.

Estructura de Comandos

Los parámetros de los comandos pueden ser divididos en dos tipos:

- **Opciones**. Comienzan con ' - '. Son *opcionales* y modifican el comportamiento del comando. Se pueden agrupar, por ejemplo, la secuencia "`-a -l -F`", equivale a "`-alF`". Existen programas que utilizan "opciones largas", que suelen comenzar con ' -- ', y no se pueden agrupar.
- Los parámetros sin guión son denominados **argumentos**. Suelen ser nombres de fichero que el comando debe procesar.

Estructura de Comandos

La estructura general de un comando se puede resumir del siguiente modo:

- **Comando** - ¿Qué quiero hacer?
- **Opciones** - ¿Cómo lo hago?
- **Argumentos** - ¿Con quién lo hago?

Normalmente las opciones van después del comando y antes de los argumentos. Pero no todos los comandos siguen esta regla, y permiten la mezcla arbitraria.

Tipos de Comandos

Básicamente hay dos tipos de comandos:

- **Comandos internos**. Están disponibles en la propia shell. La shell Bash contiene unos 30 comandos internos, que se ejecutarán muy rápido. Algunos comandos, como `exit` o `cd`, que alteran el estado de la shell, no pueden implementarse en el exterior.
- **Comandos externos**. La shell no ejecuta directamente estos comandos, sino que lanza los ficheros ejecutables, que suelen situarse en los directorios `/bin` y `/usr/bin`. Podríamos crear nuestros propios programas, que la shell ejecutará al igual que el resto de comandos externos.

Tipos de Comandos

Podemos utilizar el comando type para saber si un comando es externo o interno. Le debemos pasar el nombre de un comando como argumento:

```
$ type echo
echo is a shell builtin
$ type date
date is /bin/date
```

También podemos obtener información de un comando usando el comando help:

```
$ help type
```

Ejercicios

1. En bash, ¿cuáles de los siguientes programas son proporcionados de forma interna en la shell, o de forma externa?:

```
alias, echo, rm, test, cd, mv
```

Comandos

La shell distingue entre minúsculas y mayúsculas cuando se introducen comandos. Pero no sólo se aplica a los comandos, sino también a las opciones y a los parámetros.

Debemos tener cuidado con algunos caracteres que la shell trata de forma especial. El espacio es usado para separar las palabras en la línea de comando. Existen otros caracteres con un significado especial:

```
$&;(){}[]*?!<>”'
```

Si queremos usar alguno de estos caracteres, sin que la shell los interprete de modo especial, debemos “escaparlos”. Usar el carácter “\” para usar un sólo caracter especial, o comillas simples o dobles para varios caracteres.

Comandos

Por ejemplo:

```
$ touch 'Nuevo fichero'
```

gracias a las comillas el comando se aplica a un fichero llamado "`Nuevo fichero`". Sin las comillas, haríamos referencia a dos ficheros, "`Nuevo`" y "`fichero`".

Variables de la shell y el Entorno

Al igual que el resto de shells, bash incorpora características típicas de un lenguaje de programación. Por ejemplo, es posible almacenar datos en variables y recuperarlas más tarde. Las variables también controlan diferentes aspectos de la shell.

Una variable se declara a través de un comando como "`foo=bar`". ¡Sin espacios!. Podemos recuperar el valor de la variable anteponiendo el símbolo `$`.

```
$ foo=bar
$ echo foo
foo
$ echo $foo
bar
```

Variables de la shell y el Entorno

Debemos distinguir entre **variables de shell** y **variables de entorno**. Las variables de shell sólo son visibles en la shell en la cual han sido definidas. Sin embargo, las variables de entorno son pasadas al proceso hijo cuando se ejecuta un comando externo, y pueden accederse desde ese proceso.

Todas las variables de entorno de una shell también son variables de shell, pero no al revés.

A través del comando export, podemos convertir una variable existente en variable de entorno:

```
$ foo=bar

$ export foo
```

Variables de la shell y el Entorno

O bien, se puede definir una nueva variable como variable de shell y de entorno al mismo tiempo:

```
$ export foo=bar
```

Del mismo modo para diferentes variables:

```
$ export foo baz

$ export foo=bar baz=quux
```

Podemos ver todas las variables de entorno usando el comando `export` sin parámetros. El comando `env`, también muestra el entorno actual. Todas las variables de shell pueden mostrarse usando el comando `set`.

Variables de la shell y el Entorno

El comando env, también se puede utilizar para modificar el entorno de un proceso. Por ejemplo:

```
$ env foo=bar bash
$ echo $foo
bar
$ exit
$ echo $foo

_
```

Al menos en bash, no es necesario utilizar env para ejecutar comandos con un entorno modificado:

```
$ foo=bar bash
```

Variables de la shell y el Entorno

Para eliminar una variable, usaremos el comando `unset`. También la elimina del entorno. Si lo que queremos es eliminarla del entorno pero que permanezca en la shell, usaremos el comando "`export -n`":

```
$ export foo=bar
$ export -n foo
$ unset foo
```

Ejercicios

1. Comprueba que pasando (o no pasando) variables de shell y de entorno a los procesos hijo, se accede o no a ellas.

```
$ foo=bar
$ bash
$ echo $foo
$ exit
$ export foo
$ bash
$ echo $foo
$ exit
```

Ejercicios

2. ¿Qué ocurre si modificamos una variable de entorno en un proceso hijo?:

```
$ foo=bar
$ bash
$ echo $foo
$ foo=baz
$ exit
$ echo $foo
```

¿Qué valor obtenemos?

Comandos simples: sleep, echo y date

Para empezar a practicar, usaremos algunos comandos muy simples:

sleep Este comando no hace nada durante los segundos que se le pasen como argumento. Podemos usarlo para tomarnos un descanso.

```
$ sleep 10
```

echo Este comando muestra sus argumentos (y nada más), separados por espacios. Es interesante y muy útil, ya que la shell reemplaza las variables por su contenido en primer lugar:

Comandos simples: sleep, echo y date

```
$ p=Planeta
$ echo Hola $p
Hola Planeta
$ echo Hola $pTierra
Hola PlanetaTierra
```

Si usamos la opción -n con el comando echo, NO inserta un salto de línea al final de la línea de salida.

```
$ echo -h Hola
```

Comandos simples: sleep, echo y date

date Muestra la hora y fecha actual. Podemos modifcar el formato de salida a nuestro gusto (`date --help`).

El comando date nos puede servir como un reloj mundial, si asignamos un valor a la variable de entorno TZ con el nombre de una ciudad importante, o una capital.

```
$ date
----
$ export TZ=Asia/Tokyo
$ date
----
$ unset TZ
```

Podemos encontrar los nombres de las zonas válidas observando el directorio `/usr/share/zoneinfo`

Ejercicios

1. Asumiendo que ahora son las 12:34:56 del 22 de Octubre de 2003, examinar la documentación de date para conseguir las siguientes salidas:
    1. 22-10-2003
    2. 03-294 (año con dos dígitos, número de días transcurridos)
    3. 12h34m56s
2. ¿Que hora es ahora mismo en Los Angeles?

Ejercicios

3. Muestra la fecha del sistema.
4. Muestra el calendario completo del año 2008
5. Muestra el mes de Enero de 1999 y 99. ¿Son iguales?
6. Visualiza el texto "**Salgo a comer**" en la pantalla
7. Genera una lista de todos los usuarios presentes en tu sistema.
8. Muestra tu nombre de login.

La shell como herramienta

Debido a que la shell es la herramienta más utilizada por muchos usuarios de Linux, los desarrolladores se han preocupado de facilitar su uso.

Editor de comandos Podemos editar las lineas de comandos como si fuera un editor de textos simple. Mover el cursor y borrar o añadir caracteres antes de pulsar INTRO.

Abortar comandos Con tantos comandos, es fácil confundir un nombre o pasar un parámetro equivocado. Podemos abortar la ejecución del comando, pulsando CTRL+C.

La shell como herramienta

El historial. La shell recuerda los últimos comandos introducidos como parte del historial, y nos podemos desplazar usando las teclas ↑ y ↓ del cursor. Podemos de este modo volver a ejecutar un comando anterior, o incluso modificarlo.

Con la combinación CTRL+R realizamos una búsqueda incremental. Tecleando algunas letras, se mostrarán los comandos más recientemente utilizados que contienen esas letras.

La shell como herramienta

Cuando salimos del sistema (logout), la shell almacena el historial en el fichero oculto ~/.bash_history y lo habilita de nuevo al iniciar la siguiente sesión. El fichero es gestionado por la variable HISTFILE.

El historial se almacena en texto plano, y por lo tanto, se puede modificar con un editor de texto. Por lo que si introducimos de forma accidental una contraseña en la línea de comandos, podemos (y debemos) eliminarla del historial de comandos manualmente – *sobre todo si nuestro directorio home es visible por otros usuarios*

Por defecto, la shell recuerda los últimos 500 comandos. Este valor está almacenado en la variable HISTSIZE. Y la variable HISTFILESIZE especifica cuantos comandos se escribirán en el fichero HISTFILE – por defecto, también 500.

La shell como herramienta

Además de las teclas del cursor, podemos acceder al historial a través de las secuencias de caracteres 'mágicos'. La shell reemplaza esta secuencia de caracteres en primer lugar, y luego lee la línea de comandos. Existen dos etapas:

• La shell averigua qué comando del historial hay que reemplazar. La secuencia `!!` se cambia por el último comando, `!-n` se refiere al enésimo comando antes del actual (`!-2`, es el penúltimo), y `!n` se reemplaza por el comando con el número n en el historial. `!abc` selecciona el comando más reciente que comienza por abc, y `!?abc` el más reciente que contiene abc.

• Después, la shell decide qué parte del comando seleccionado será 'reciclada' y cómo. Si no se indica nada, se insertará el comando completo.

La shell como herramienta

Una llamada al comando como ésta:

```
$ history 33
```

sólo muestra las últimas 33 lineas del historial. “history -c” limpia el historial completamente. Hay más opciones que se pueden consultar a través de “help history”.

Autocompletado Una gran utilidad de bash es la capacidad de autocompletado para nombres de comandos y ficheros. Si pulsamos la tecla TAB, la shell tratará de completar una entrada incompleta. Para la primera palabra de la línea buscará entre los comandos y para las siguientes entre los nombres de fichero del directorio indicado o actual. Si existen varias coincidencias, completa hasta donde sea posible, y una segunda pulsación mostrará todas las posibilidades.

Ejercicios

1. Usa el comando history para ver los últimos 20 comandos utilizados.

2. Ejecuta de nuevo el comando echo, pero cambiando la palabra comer por cenar.

3. ¿Qué ocurre al ejecutar el comando `echo "Hola!"`?

Más sobre la línea de comandos

Habíamos visto que existían dos tipos de comandos, los internos y los externos.
¿Cómo localiza la shell los programas correspondientes a los comandos externos? En primer lugar, los programas se almacenan en ficheros, y los ficheros en directorios. El problema queda reducido a localizar un fichero con el mismo nombre que el comando.
Pero, ¿dónde buscar?, pues la shell mantiene una lista de directorios en la variable de entorno PATH:

```
$ echo $PATH
```

Los directorios están separados por dos puntos. Si escribimos el siguiente comando:

```
$ ls
```

Más sobre la línea de comandos

La shell sabe que el comando `ls` no es un comando interno, y comienza la búsqueda en los directorios listados en la variable PATH, empezando por el situado más a la izquierda. Por ejemplo:

```
/usr/local/bin/ls
/usr/bin/ls
/bin/ls
```

Esto implica que el fichero `/bin/ls` será el utilizado para ejecutar el comando ls.

La tarea de búsqueda del fichero es costosa, por lo que la shell debe estar preparada para el futuro. Una vez que ha relacionado el fichero `/bin/ls` con el comando `ls`, recuerda esta relación para usos posteriores. Este proceso se llama "hashing", y puede comprobarse usando el comando `type` con el comando `ls`.

Más sobre la línea de comandos

El comando hash nos muestra los comandos que ha almacenado en su tabla hash, y cuantas veces han sido utilizados. Con "`hash -r`" podemos eliminar por completo la tabla almacenada. Hay otras opciones para buscar elementos en la tabla, podemos usar "`help hash`".

Varios comandos

Es posible introducir varios comandos en la misma línea, separandolos mediante punto y coma (;)

```
$ echo Hoy es; date
```

En este ejemplo, el segundo comando se ejecuta una vez finalizado el primero.

Más sobre la línea de comandos

Hay ocasiones en las que puede resultar útil que la ejecución del segundo comando dependa del resultado de la ejecución del primero. Todo proceso Unix devuelve un **valor de retorno** que indica si ha finalizado con éxito su ejecución o ha ocurrido algún problema. En el primer caso, el valor devuelto es 0, y cualquier otro si ha ocurrido algún error.

Podemos conocer el valor de retorno de un proceso hijo de la shell examinando la variable $?:

```
$ bash
$ exit 33
$ echo $?
```

Más sobre la línea de comandos

Si en lugar de utilizar el punto y coma, usamos como separador los caracteres `&&`, entonces el segundo comando sólo se ejecutará cuando el primer comando haya finalizado **con** éxito. En el ejemplo usamos la opción `-c` de la shell:

```
$ bash -c "exit 0" && echo "Con éxito"
$ bash -c "exit 33" && echo "Con éxito"
```

Y con el separador `||`, el segundo comando sólo se ejecutará cuando el primer comando haya finalizado **sin** éxito.

```
$ bash -c "exit 0" || echo "Sin éxito"
$ bash -c "exit 33" || echo "Sin éxito"
```

Comandos en un fichero

También es posible almacenar comandos de shell en un fichero y ejecutarlos en bloque (*veremos como crearlos en lecciones posteriores*). Tan sólo necesitamos ejecutar la shell y pasarle el fichero como parámetro.

```
$ bash mis-comandos
```

Dicho fichero se llama “shell script”, y contamos con características de programación muy potentes.

Si ejecutamos el script del modo anterior, es ejecutado en una subshell, que es un proceso hijo de la shell actual. Esto significa que los cambios en el entorno no afectan a la shell actual.

Comandos en un fichero

Por ejemplo, el fichero `asignacion` contiene la siguiente línea:

```
foo=bar
```

```
$ foo=tux
$ bash asignacion
$ echo $foo
??
```

Pero, también existe la posibildad de ejecutar un script de forma que los cambios sí afecten a la shell actual. El comando `source` lee las líneas del fichero igual que si las escribieramos directamente en la propia shell:

Comandos en un fichero

```
$ foo=tux
$ source asignacion
$ echo $foo
bar
```

Un nombre equivalente para source es “.” (si, punto), por lo tanto:

```
$ source asignacion
```

es equivalente a:

```
$ . asignacion
```

Viaje sin retorno

La shell, normalmente, espera que el comando externo finalice la ejecución, y luego lee el siguiente comando. Usando el comando exec, podemos ejecutar un comando externo, que reemplaza la propia shell. Por ejemplo, si deseamos usar la shell C en lugar de bash:

```
$ exec /bin/csh
% _
```

El comando `exec` se suele utilizar en los scripts, y no con demasiada frecuencia.

Ejercicios

1. Supongamos que el fichero `test1` contiene las siguientes líneas:

```
echo Hola
exec test2
echo Adios
```

y el fichero `test2` la línea:

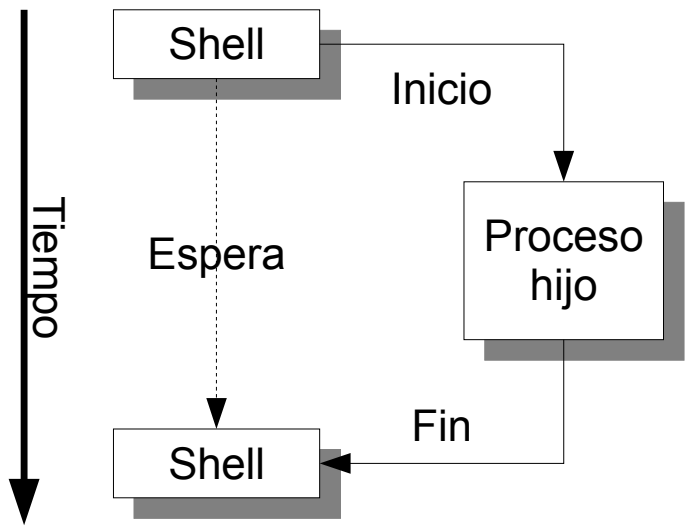
```
echo Tux
```

¿Qué salida produce el comando "`bash test1`"?

Procesos en primer y segundo plano

Después de escribir un comando, es procesado por la shell. Los comandos internos, son ejecutados directamente, y los comandos externos generan un **proceso hijo**, que ejecuta el comando.

La shell espera que el proceso hijo finalice. Observamos que durante esta ejecución no se muestra el prompt de la shell. Entonces, la ejecución de la shell y del proceso hijo, es **síncrona**.



Procesos en primer y segundo plano

Desde el punto de vista del usuario:

```
$ sleep 10
```

No ocurre nada durante 10 segundos

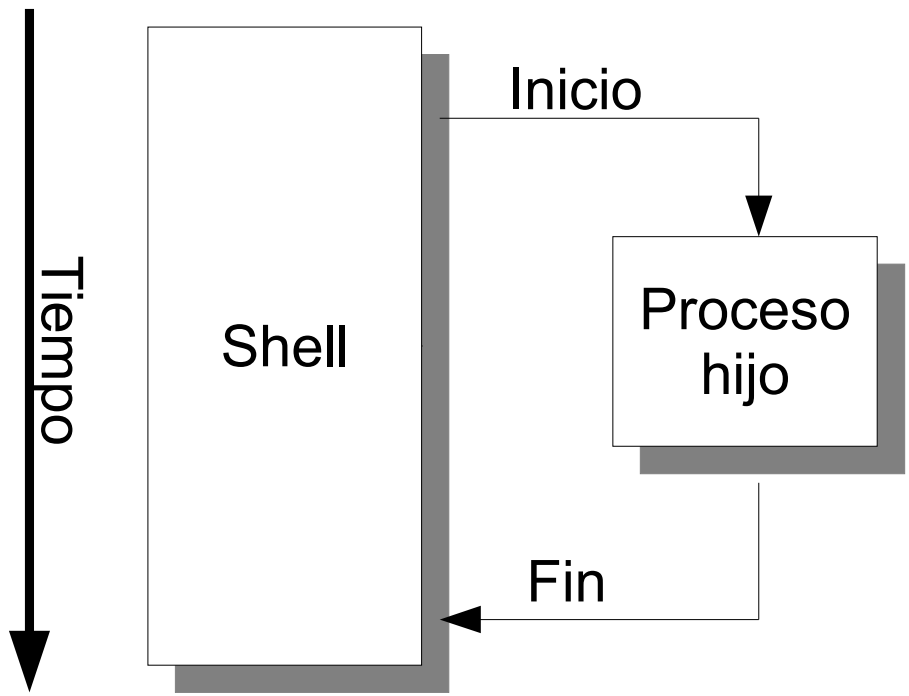
```
$ _
```

Si queremos que la shell no espere a que el proceso hijo finalice, debemos añadir un ampersand (&) al final de la línea. De este modo, el proceso hijo es ejecutado en segundo plano, mostrando un mensaje en el terminal:

```
$ sleep 10 &
[2]   6210
$ _
```

Procesos en primer y segundo plano

Este modo de operación se denomina “**asíncrono**”, debido a que la shell no espera por el final del proceso hijo.



Procesos en primer y segundo plano

Si tenemos varios procesos ejecutandose en segundo plano, podemos confundirnos, es por ello, que la shell proporciona el comando `jobs` para conocer el estado de los procesos en segundo plano. Se usa sin opciones, y muestra una lista de números de trabajo, estado de proceso y líneas de comandos.

```
$ jobs
[1]   Done                    sleep
$ _
```

En el ejemplo, el trabajo 1 ha finalizado, de no ser así se mostraría “Running”.

Es posible parar un proceso del primer plano, usando la combinación CTRL+Z.

Procesos en primer y segundo plano

Ese proceso 'parado' se muestra como “`Stopped`”. Y puede continuar como un proceso en segundo plano mediante el comando `bg`. Por ejemplo, el comando “`bg %5`” envía el trabajo 5 al segundo plano, donde se continuará ejecutando.

Del mismo modo, podemos traer trabajos del segundo plano usando el comando `fg`.

Para finalizar un proceso en primer plano usamos la combinación CTRL-C. Y un proceso en segundo plano podemos finalizarlo mediante el comando `kill`, con la sintaxis similar a `bg`.

Ejercicios

1. Usando un programa como "`xclock -update 1`", realizar experimentos con el control de trabajos y el segundo plano. Inicar procesos en segundo plano, parar procesos con CTRL-Z, enviarlos al segundo plano con `bg` y listar los trabajos con el comando `jobs`.

2. Explicar las diferencias entre las siguientes líneas de comandos:

```
$ sleep 5 ; sleep 5
$ sleep 5 ; sleep 5 &
$ sleep 5 & sleep 5 &
```

Comandos vistos en el tema

.	Lee un fichero con comandos y los interpreta.
`bash`	“Bourne-Again Shell”
`bg`	Continúa un proceso (parado) en el segundo plano
`csh`	La “C-shell”
`date`	Muestra la hora y la fecha
`env`	Muestra el entorno de proceso, o inicia un programa con un determinado entorno
`exec`	Inicia un nuevo programa en el proceso de la shell actual
`export`	Declara y gestiona variables de entorno
`fg`	Recupera a primer plano un proceso en segundo plano
`hash`	Muestra y gestiona los comandos 'localizados'
`help`	Muestra ayuda en línea para los comandos bash
`history`	Muestra los comandos recientemente utilizados
`jobs`	Lista los trabajos en segundo plano

Comandos vistos en el tema

`kill`	Finaliza un proceso en segundo plano
`ksh`	La “Korn Shell”
`set`	Gestiona las variables de la shell y las opciones
`sh`	La “Bourne Shell”
`source`	Lee un fichero con comandos y los interpreta.
`tcsh`	La “Tenex C shell”
`type`	Averigua el tipo de comando (interno, externo o alias)
`unset`	Elimina variables de shell o de entorno
`xclock`	Muetra un reloj gráfico

Resumen

• La shell lee comandos de usuario y los ejecuta. La mayoría de las shells incorporan características de lenguaje de programación y soportan shell scripts.
• Los comandos pueden tener opciones y argumentos. Las opciones modifican el comportamiento, y los argumentos sobre qué elementos actuar.
• Las shells diferencian entre comandos internos, que son implementados en la propia shell, y comandos externos, que corresponden a ficheros ejecutables que se inician en procesos separados.
• El comando `sleep` espera un cierto número de segundos, indicados como argumento.
• El comando `echo` envia a pantalla sus argumentos.
• La fecha y la hora se muestran a través del comando `date`.
• Bash incorpora funciones para uso interactivo, como autocompletado de comandos y ficheros, edición de la línea de comandos, alias y variables.
• Los programas externos pueden ser iniciados de forma síncrona, o asíncrona en segundo plano.