

INTRODUCCION A LOS PROCESOS

Que es un proceso

A estas alturas más de uno empezará a preguntarse si realmente es necesario conocer todos estos detalles para un simple uso de un SO. En Linux desde la consola y resulta necesario conocer estas cosas incluso para un usuario normal. No vamos a describir detalladamente como están implementados los procesos en Linux.

Quizás para un curso de programación avanzada si fuera necesario, pero lo que nosotros vamos a describir es únicamente los detalles más importantes desde un punto de vista didáctico y práctico para un usuario normal.

En un sistema multitarea todo funciona con procesos así que conocer unos cuantos principios básicos sobre procesos le resultará de la máxima utilidad. En un sistema monotarea se usa frecuentemente el término programa indistintamente para hablar de un programa en papel, en cdrom, en disco duro o en funcionamiento.

En un sistema multitarea no se debe usar el término programa para hablar de la ejecución del mismo. En su lugar hablaremos de proceso indicando con ello que esta arrancado y funcionando. Un programa puede dar lugar a varios procesos. Por ejemplo en un mismo instante varios usuarios pueden estar usando un mismo editor. Un proceso puede estar detenido pero a diferencia de un programa existe una información de estado asociada al proceso. Un programa es algo totalmente muerto. Un proceso detenido es más bien como si estuviera dormido.

Los procesos tienen organizada la memoria de una forma especial muy eficiente. Por ejemplo la parte de código del proceso es una parte de solo lectura y puede ser compartida por varios procesos a la vez. Imaginemos que hay varios usuarios en el sistema usando un mismo editor. En ese caso sería un desperdicio tener la misma información de código de ese programa repetida varias veces en memoria y ocupando un recurso tan valioso como es la memoria RAM.

En el caso de tener varios programas distintos en ejecución también se suele dar el caso de que contengan partes comunes de código pertenecientes a librerías que contienen gran cantidad de funciones de propósito general. Para hacer un uso eficiente de estas librerías existen librerías dinámicas de uso compartido. En Linux el uso de estas librerías no está organizado de una forma unificada para todas las distribuciones por lo cual los ejecutables binarios pueden ser incompatibles entre distintas distribuciones. Este problema se puede solucionar partiendo de los fuentes y recompilando la aplicación en nuestro sistema. Por esta razón un binario de RedHat, o Suse puede no funcionar en Slackware o en Debian.

PID y PPID

A cada proceso le corresponderá un número PID que le identifica totalmente. Es decir en un mismo momento es imposible que existan dos procesos con el mismo PID. Lo mismo que todos los procesos tienen un atributo PID que es el número de proceso que lo identifica en el sistema también existe un atributo llamado PPID. Este número se corresponde con el número PID del proceso padre. Todos los procesos deben de tener un proceso que figure como padre pero entonces que ocurre si un padre muere antes que alguno de sus hijos ? En estos casos el proceso 'init' del cual hablaremos en seguida adoptará a estos procesos para que no queden huérfanos.

El proceso init

Cuando arranca el sistema se desencadena una secuencia de procesos que a grandes rasgos es la siguiente. Primero se carga el núcleo de Linux (Kernel) de una forma totalmente especial y distinta a otros procesos. Dependiendo de los sistemas puede existir un proceso con PID=0 planificador, o swapper. En Linux y en casi todos los sistemas tipo Unix seguirá un proceso llamado 'init'. El proceso init tiene PID = 1. Lee un fichero llamado inittab donde se relacionan una serie de procesos que deben arrancarse para permanecer activos todo el rato (demonios). Algunos de ellos están definidos para que en caso de morir sean arrancados de nuevo inmediatamente garantizando la existencia de ese servicio de forma permanente.

Es decir 'init' es un proceso que va a generar nuevos procesos pero esta no es una cualidad especial. Es muy frecuente que un proceso cualquiera genere nuevos procesos y cuando esto ocurre se dice que genera procesos hijos.

Este no es un curso de administración pero diremos que a init se le puede indicar que arranque el sistema de diferentes formas, por ejemplo en modo monousuario para mantenimiento. Este es un capítulo en el cual pueden surgir muchas preguntas retorcidas tales como, que pasa si matamos a init, o quien es el padre de init, pero no nos interesa responder a esto ya que init es un proceso muy especial y su padre aún más. En cada sistema de tipo Unix las respuestas a cosas como estas pueden variar mucho porque a ese nivel la implementaciones varían mucho. Ningún programa normal necesitará usar ese tipo de información. Quedan

muchos detalles interesantes relativos a temas de administración. Los curiosos siempre tienen el recurso de mirar la página man de `init(8)` y de `inittab(5)` pero nosotros no insistiremos más en este tema.

UID y EUID

Los procesos tienen un EUID (Efectiv User Identif), y un UID normalmente ambos coinciden. El UID es el identificador de usuario real que coincide con el identificador del usuario que arrancó el proceso. El EUID es el identificador de usuario efectivo y se llama así porque es el identificador que se tiene en cuenta a la hora de considerar los permisos que luego explicaremos.

El UID es uno de los atributos de un proceso que indica por decirlo de alguna manera quien es el propietario actual de ese proceso y en función de ello podrá hacer ciertas cosas. Por ejemplo si un usuario normal intentara eliminar un proceso del cual no es propietario el sistema no lo permitirá mostrando un mensaje de error en el que advierte que usted no es el propietario de ese proceso y por tanto no está autorizado a hacer esa operación. Por el contrario el usuario root puede hacer lo que quiera con cualquier proceso ya que el sistema no comprueba jamás si root tiene permisos o no para hacer algo. root siempre tiene permisos para todo. Esto es cierto a nivel de llamadas del sistema pero nada impide implementar un comando que haga comprobaciones de permisos incluso con root. Algunos comandos en Linux tienen opciones que permiten hacer estas cosas y solicitar confirmación en caso de detectar una operación peligrosa o infrecuente.

El UID también es un atributo presente en otros elementos del sistema. Por ejemplo los ficheros y directorios del sistema tienen este atributo. De esta forma cuando un proceso intenta efectuar una operación sobre un fichero. El kernel comprobará si el EUID del proceso coincide con el UID del fichero. Por ejemplo si se establece que determinado fichero solo puede ser leído por su propietario el kernel denegará todo intento de lectura a un proceso que no tenga un EUID igual al UID del fichero salvo que se trate del todo poderoso root.

Aunque estamos adelantando cosas sobre el sistema de ficheros vamos a continuar con un ejemplo. Comprobaremos cual es el UID de su directorio home.

Cambiamos el directorio actual a home

\$ cd

Comprobamos a quien pertenece 'uid' y 'gid'

\$ ls -ld .

Ahora obtenemos el 'uid' y el 'gid' con sus valores numéricos.

\$ ls -ln .

Si su directorio home está configurado de forma lógica deberá pertenecerle a usted. Si esto no es así reclame enérgicamente a su administrador, pero si el administrador resulta ser usted sea más indulgente y límitese a corregirlo y no confíese su error a nadie. En realidad casi todo lo que se encuentre dentro de su directorio home debería pertenecerle a usted. Usted debe ser el propietario de su directorio home porque de otra forma y dependiendo de los permisos asociados a este directorio los procesos que usted arranque se verían o bien incapaces de trabajar con él, o lo que es peor cualquiera podría hacer cualquier cosa con él. Como es lógico hemos mencionado de pasada el tema de permisos de directorios para ilustrar un poco la utilidad del uid pero esto se verá en detalle en el capítulo dedicado al sistema de ficheros de Linux.

Algunos ficheros ejecutables poseen un bit de permisos que hace que cambie el EUID del proceso que lo ejecute convirtiéndose en el UID del propietario del fichero ejecutado. Suena complicado pero no lo es. Es decir imaginemos que usted ejecuta un comando propiedad de root que tiene este bit. Pues bien en ese momento el EUID de su proceso pasaría a ser el de root. Gracias a esto un proceso puede tomar temporalmente la identidad de otro usuario. Por ejemplo puede tomar temporalmente la identidad de root para adquirir privilegios de superusuario y así acceder por ejemplo a ficheros del sistema propiedad de root. El sistema recibe continuamente peticiones de los procesos y el EUID del proceso determinará que el kernel le conceda permiso para efectuar la operación deseada o no.

Muchas veces sorprende que en Linux apenas se conozcan unos pocos casos de virus, mientras que en otros sistemas parecen estar a la orden del día. Es perfectamente posible realizar un virus que infecte un sistema Linux pero de una forma o de otra el administrador tendría que darle los privilegios que le convierten en peligroso. Por eso no es fácil que estos virus lleguen muy lejos.

Como se crea un nuevo proceso

El núcleo del sistema llamado también kernel es el encargado de realizar la mayoría de funciones básicas que gestiona entre otras cosas los procesos. La gestión de estas cosas se hacen por medio de un limitado número de funciones que se denominan llamadas al sistema. Estas llamadas al sistema están implementadas en lenguaje C y hablaremos ahora un poco sobre un par de ellas llamadas `fork()` y `exec()`. Si logramos que

tenga una vaga idea de como funcionan estas dos importantísimas funciones facilitaremos la comprensión de muchas otras cosas más adelante.

La llamada al sistema exec()

Cuando hablamos de proceso debe usted pensar solamente en algo que se está ejecutando y que está vivo. Un proceso puede evolucionar y cambiar totalmente desde que arranca hasta que muere. Lo único que no cambia en un proceso desde que nace hasta que se muere es su identificador de proceso PID. Una de las cosas que puede hacer un proceso es cambiar por completo su código. Por ejemplo un proceso encargado de procesar la entrada y salida de un terminal (getty) puede transformarse en un proceso de autenticación de usuario y password (login) y este a su vez puede transformarse en un interprete de comandos (bash). Si la llamada exec() falla retornará un -1. Esto no es curso de programación así que nos da igual el valor retornado pero lo que si nos interesa es saber que cuando esta llamada tiene éxito no se produce jamás un retorno. En realidad no tendría sentido retornar a ningún lado. Cuando un proceso termina simplemente deja de existir. En una palabra muere y ya está. La llamada exec() mantiene el mismo PID y PPID pero transforma totalmente el código de un proceso en otro que cargará desde un archivo ejecutable.

La llamada al sistema fork()

La forma en que un proceso arranca a otro es mediante una llamada al sistema con la función fork(). Lo normal es que el proceso hijo ejecute luego una llamada al sistema exec(). fork() duplica un proceso generando dos procesos casi idénticos. En realidad solo se diferenciaran en los valores PID y PPID. Un proceso puede pasar al proceso hijo una serie de variables pero un hijo no puede pasar nada a su padre a través de variables. Además fork() retorna un valor numérico que será -1 en caso de fallo, pero si tiene éxito se habrá producido la duplicación de procesos y retornará un valor distinto para el proceso hijo que para el proceso padre. Al proceso hijo le retornará el valor 0 y al proceso padre le retornará el PID del proceso hijo. Después de hacer fork() se pueden hacer varias cosas pero lo primero que se utiliza después del fork es una pregunta sobre el valor retornado por fork() para así saber si ese proceso es el padre o el hijo ya que cada uno de ellos normalmente deberá hacer cosas distintas. Es decir la pregunta sería del tipo si soy el padre hago esto y si soy el hijo hago esto otro. Con frecuencia aunque no siempre el hijo hace un exec() para transformarse completamente y con frecuencia aunque no siempre el padre decide esperar a que termine el hijo.

También normalmente aunque no siempre esta parte de la lección es necesario releerla más de una vez. Estamos dando pequeños detalles de programación porque en estas dos llamadas del sistema son muy significativas. Su funcionamiento resulta chocante y su comprensión permite explicar un montón de cosas.

MAS SOBRE PROCESOS Y SEÑALES <http://doblev.wordpress.com>

Las formas de comunicación entre procesos

Las formas de comunicación entre procesos

Los procesos no tiene una facilidad de acceso indiscriminada a otros procesos. El hecho de que un proceso pueda influir de alguna manera en otro es algo que tiene que estar perfectamente controlado por motivos de seguridad. Comentaremos solo muy por encima las diferentes formas de comunicación entre procesos.

1. A través de variables de entorno: Solo es posible de padres a hijos.
2. Mediante una señal: Solo indica que algo ha ocurrido y solo lleva como información de un número de señal.
3. Mediante entrada salida: Es la forma más corriente a nivel de shell. Ya hemos comentado el operador pipe '|' que conecta dos procesos.
4. Mediante técnicas IPC u otras: Semáforos, Memoria compartida, Colas de mensajes.
5. Mediante sockets: Este sistema tiene la peculiaridad de que permite comunicar procesos que estén funcionando en máquinas distintas.

No profundizamos sobre esto porque ahora no estamos interesados en la programación. Más adelante si comentaremos bastante sobre variables y entrada salida porque daremos nociones de programación en shell-script. También daremos a continuación unos pequeños detalles que tienen que ver con el arranque del sistema porque también nos resultará útil para comprender como funcionan las cosas.

Secuencia de arranque en una sesión de consola

Para consultar la dependencia jerárquica entre unos procesos y otros existe en Linux el utilísimo comando pstree. No es esencial y quizás no lo tenga usted instalado pero resulta muy práctico. Si dispone de él pruebe los comandos 'pstree', y 'pstree -p'. Nosotros vamos a mostrar el resultado de ambos comandos en

el sistema que estamos usando en este momento para que en cualquier caso pueda apreciar el resultado pero le recomendamos que lo instale si no dispone de él ya que resulta muy práctico. También puede usar como sustituto de 'pstree' el comando 'ps axf'.

```
$ pstree
```

```
init -- apache --- 7*[apache]
  |
  | -atd
  | -bash --- pstree
  | -2*[bash]
  | -bash --- vi
  | -bash --- xinit -- XF86 S3V
  |           \- mwm +- .xinitrc --- xterm --- bash
  |           \- .xinitrc --- xclock
  |
  | -cron
  | -getty
  | -gpm
  | -inetd
  | -kflushd
  | -klogd
  | -kpiod
  | -kswapd
  | -kupdate
  | -lpd
  | -portmap
  | -postmaster
  | -sendmail
  | -sshd
  | -syslogd
  \- xfs
```

Este formato nos da un árbol de procesos abreviado en el que procesos con el mismo nombre y que tengan un mismo padre aparecerán de forma abreviada. En el ejemplo anterior aparece '2*[bash]' indicando que hay dos procesos bash como hijos de init, y también hay algunos proceso apache arrancados. El que existan muchos procesos arrancados no indica necesariamente un alto consumo de CPU.

```
$ pstree -p
```

```
init(1) -- apache(204) -- apache(216)
  |
  | | -apache(217)
  | | -apache(218)
  | | -apache(219)
  | | -apache(220)
  | | -apache(1680)
  | | \- apache(1682)
  | -atd(196)
  | -bash(210) --- pstree(1779)
  | -bash(211)
  | -bash(212) --- vi(1695)
  | -bash(215) --- xinit(1639) -- XF86 S3V(1644)
  |           \- mwm(1647) +- .xinitrc(1652) --- xterm(1660) --- bash(1661)
  |           \- .xinitrc(1655) --- xclock(1673)
  |
  | -bash(214)
  | -cron(199)
  | -getty(213)
  | -gpm(143)
  | -inetd(138)
  | -kflushd(2)
  | -klogd(131)
  | -kpiod(4)
```

```
|-kswapd(5)
|-kupdate(3)
|-lpd(153)
|-portmap(136)
|-postmaster(168)
|-sendmail(179)
|-sshd(183)
|-syslogd(129)
`-xfs(186)
```

En este otro formato. Aparece cada proceso con su PID. Podemos ver que el Proceso 'init' tiene pid = 1 y ha realizado varios forks() generando procesos con pid > 1. En algunos sistemas la generación de números de PID para procesos nuevos se realiza en secuencia. En otros resulta un número impredecible.

Entre los procesos generados por 'init' están los procesos 'getty'. Se arrancará un 'getty' por cada terminal. Este proceso configura la velocidad y otras cosas del terminal, manda un saludo y luego se transforma con exec el proceso 'login'. Todos estos procesos se ejecutan con EUID y UID = 0, es decir como procesos del superusuario root. Cuando el proceso 'login' conoce nuestra identidad después de validar usuario password se transformará con exec en la shell especificada para nuestro usuario el fichero /etc/passwd

Para ver la línea que contiene sus datos pruebe a hacer lo siguiente:

```
$ grep `whoami` /etc/passwd
```

La línea tiene el siguiente formato. login:contraseña:UID:GID:nombre:dir:intérprete

Vamos a suponer que su shell por defecto sea la bash. Si esta shell arrancara con el EUID = 0 tendríamos todos los privilegios del super usuario pero esto no ocurre así. Esta shell ya tendrá nuestro UID y nuestro EUID. Vamos a representar todo esto marcando los puntos en los que ocurre algún fork() con un signo '+'.
[init]+fork()->[getty]

```
|
|+fork()->[getty]-exec()->[login]-exec()->[bash]+fork()-exec()->[comando]
|
|+fork()->[getty]
|
```

La shell puede arrancar un comando mediante un fork() y luego un exec() y esperar a que este muera. Recuerde que la función exec() no tiene retorno posible ya que finaliza con la muerte del proceso. En ese momento la shell detecta la muerte de su hijo y continua su ejecución solicitando la entrada de un nuevo comando. Cuando introducimos el comando 'exit' estamos indicando a la shell que finalice y su padre 'init' se encargará de lanzar nuevo proceso 'getty'. Lógicamente 'exit' es un comando interno de la shell. Quizás le llame la atención que la muerte de 'bash' termine provocando un nuevo 'getty' cuando 'getty' pasó a 'login' y este a 'bash' pero en esta secuencia getty-login-bash no hay ningún fork() por eso getty, login, y bash son en realidad el mismo proceso en distintos momentos con el mismo PID obtenido en el fork() realizado por 'init' solo que ha ido cambiando su personalidad manteniendo la misma identidad (mismo PID). Para 'init' siempre se trató del mismo hijo y la muerte de cualquiera de ellos (getty, login o bash) provoca que se arranque un nuevo 'getty' sobre ese mismo terminal con el fin de que ese terminal no quede sin servicio.

La presentación del mensaje de Login es mostrada por 'getty'. Una vez introducido el identificador de usuario será 'login' quien muestre la solicitud de introducción de la password, y una vez introducido el password será la shell quien muestre el introductor de comandos pero estamos hablando siempre del mismo proceso.

A modo de ejercicio compruebe estas cosas por usted mismo usando algunos de los comandos que ya conoce. Para hacer esta práctica no basta con usar un terminal remoto sino que necesitará un PC completo para ir haciendo cosas desde distintas sesiones. le proponemos hacerlo más o menos de la siguiente manera:

1. Entre en cada uno de los terminales disponibles de forma que todos los terminales estén ocupados por un interprete de comandos. Bastará con hacer login en todos ellos.
2. Luego compruebe que no hay ningún proceso 'getty'.
3. Haga un exit desde uno de estos terminales.
4. Compruebe desde otro termina que ahora si existe un proceso 'getty' y anote su pid.
5. Introduzca el nombre de usuario en ese terminal que quedó libre.
6. Compruebe ahora desde otra sesión que existe un proceso login con el PID que usted anotó.
7. Termine de indentificarse tecleando la password
8. Compruebe desde otra sesión que ahora existe una shell con el PID que anotamos.

Si no tiene el comando 'pstree' tendrá que usar 'ps' pero con pstree puede ver más fácilmente lo que ocurrirá ahora.

9. Ahora teclee el comando 'sleep 222' desde la sesión que tiene el PID anotado por usted.

10. Compruebe desde otra sesión que el interprete de comandos ha realizado un fork() generando un comando que está ejecutando el comando indicado.

Si no ha podido realizar el ejercicio anterior tendrá que confiar en que las cosas son como decimos y ya está.

Comando ps

Muestra los procesos activos. Este comando es muy útil para saber que comandos están funcionando en un determinado momento.

Siempre que se mencione un comando puede consultar la página man del mismo. En el caso de 'ps' se lo recomendamos ya que es un comando muy útil con una gran cantidad de opciones. Nosotros mencionaremos algunos ejemplos pero se aconseja probar 'ps' probando las distintas opciones que se mencionan en la página del manual.

Ejemplos:

Para ver todos sus procesos que están

asociados a algún terminal.

\$ ps

PID TTY STAT TIME COMMAND

.....

Para ver todos sus procesos y los de otros

usuarios siempre asociados a algún terminal.

\$ ps a

PID TTY STAT TIME COMMAND

.....

Para ver todos sus procesos estén asociados o

no a algún terminal.

\$ ps x

PID TTY STAT TIME COMMAND

.....

Para ver todos los proceso asociados al

terminal 1

\$ ps t1

PID TTY STAT TIME COMMAND

.....

Para ver todos los procesos del sistema.

\$ ps ax

PID TTY STAT TIME COMMAND

.....

Estos ejemplos que acabamos de ver obtienen un mismo formato de datos. Explicaremos el significado de estos atributos

PID	Es el valor numérico que identifica al proceso.
TTY	Es el terminal asociado a ese proceso. Los demonios del sistema no tienen ningún terminal asociado y en este campo figurará un ?
STAT	Tiene tres campos que indican el estado del proceso (R,S,D,T,Z) (W) (N) La S indica que el proceso está suspendido esperando la liberación de un recurso (CPU. Entrada Salida. etc)

	necesario para continuar. Explicaremos solo algunos de estos estados en su momento.
TIME	Indica el tiempo de CPU que lleva consumido ese proceso desde que fue arrancado.
COMMAND	Muestra el comando y los argumentos que le fueron comunicados.

Existen muchas opciones para el comando ps que ofrecen un formato distinto. Le recomendamos especialmente que pruebe 'ps u', 'ps l', y 'ps f'

En Unix los comandos suelen servir para una sola cosa, aunque suelen tener muchas opciones. La entrada de los comandos suele tener una estructura simple y la salida de los comandos también. Si un comando no encuentra nada que hacer existe la costumbre de que termine de modo silencioso. Todo esto permite que los comandos puedan combinarse enganchado la salida de uno con la entrada de otro. Algunos comandos están especialmente diseñados para ser usados de esta forma y se les suele denominar filtros.

La salida del comando 'ps' se puede filtrar con 'grep' para que muestre solo las líneas que nos interesan.

Configuración del terminal

Conviene que comprobemos si su terminal está correctamente configurado para poder interrumpir un proceso. Normalmente se usa <Ctrl-C> pero esto depende de la configuración de su terminal. Si en algún momento su terminal queda desconfigurado haciendo cosas raras como por ejemplo mostrar caracteres extraños intente recuperar la situación tecleando el comando 'reset'. Esto solo es válido para Linux. Para otros sistemas puede ser útil 'stty sane' que también funciona en Linux pero no es tan eficaz como el comando 'reset'. Para comprobar la configuración de su terminal puede hacer 'stty -a' aunque obtendrá demasiada información que no es capaz de interpretar, podemos indicarle que se fije en el valor de 'intr'. Debería venir como 'intr = ^C'. Si no lo localiza haga 'stty -a | grep intr'. De esta forma solo obtendrá una línea. Para configurar el terminal de forma que pueda interrumpir procesos con <Ctrl-C> puede intentar configurar su terminal haciendo 'stty ^V^C'. El carácter <Ctrl-V> no se mostrará en el terminal ya que actúa evitando que el siguiente carácter (<Ctrl-C> en nuestro caso) no sea interpretado como carácter de control. No pretendemos ahora explicar los terminales de Linux pero si queremos que compruebe su capacidad para interrumpir procesos con <Ctrl-C> ya que usaremos esto en las prácticas que siguen. Una prueba inofensiva para comprobar la interrupción de un proceso es el siguiente comando que provoca una espera de un minuto. Deberá introducir el comando e interrumpirlo antes de que el tiempo establecido (60 segundos se agote).

```
$ sleep 60
```

```
<Ctrl-C>
```

Si no ha conseguido interrumpir el proceso no siga adelante para evitar que alguna de las prácticas deje un proceso demasiado tiempo consumiendo recursos de su máquina. Si esta usted solo en la máquina eso tampoco tendría mucha importancia pero es mejor que averigüe la forma de interrumpir el proceso del ejemplo anterior.

Comando time

Da los tiempos de ejecución. Este comando nos da tres valores cuya interpretación es:

```
real Tiempo real gastado (Duración real)
user Tiempo CPU de usuario.
sys. Tiempo CPU consumido como proceso de kernel.
(Es decir dentro de las llamadas al kernel)
```

La mayoría de los comandos están gran parte del tiempo sin consumir CPU porque necesitan esperar para hacer entrada salida sobre dispositivos lentos que además pueden estar en uso compartidos por otros procesos. Existe un comando capaz de esperar tiempo sin gastar tiempo de CPU. Se trata del comando 'sleep'. Para usarlo le pasaremos un argumento que indique el número de segundos de dicha espera.

Por ejemplo vamos a comprobar cuanta CPU consume una espera de 6 segundos usando sleep

```
$ time sleep 6
```

```
real 0m6.021s
```

```
user 0m0.020s
```

```
sys 0m0.000s
```

El resultado obtenido puede variar ligeramente en cada sistema pero básicamente obtendrá un tiempo 'real' de unos 6 segundos y un tiempo de CPU ('user' + 'sys') muy pequeño.

Vamos a medir tiempos en un comando que realice operaciones de entrada salida así como proceso de datos.

```
$ time ls /* > /dev/null
```

```
real 0m0.099s
user 0m0.080s
sys 0m0.010s
```

En este comando verá que el consumo total de CPU es superior al del comando sleep. En cualquier caso el tiempo real tardado en la ejecución del comando es siempre muy superior al consumo de CPU.

Vamos a probar un comando que apenas realice otra cosa que entrada salida. Vamos a enviar 10Mbytes al dispositivo /dev/null. Existe un comando 'yes' que provoca la salida continua de un caracter 'y' seguido de un caracter retorno de carro. Esta pensado para sustituir la entrada de un comando interactivo en el cual queremos contestar afirmativamente a todo lo que pregunte. Nosotros fitraremos la salida de 'yes' con el comando 'head' para obtener solo los 10Mbytes primeros producidos por 'yes' y los enviaremos al dispositivo nulo '/dev/null' que viene a ser un pozo sin fondo en el cual podemos introducir cualquier cosa sin que se llene, pero no podremos sacar absolutamente nada. En una palabra vamos a provocar proceso de entrada salida perfectamente inutil.

```
time yes | head --bytes=1000000 > /dev/null
```

Tubería rota

```
real 0m6.478s
user 0m5.440s
sys 0m0.900s
```

Podemos hacer un consumo fuerte de CPU si forzamos a cálculos masivos que no tengan apenas entrada salida. Por ejemplo podemos poner a calcular el número PI con 300 cifras decimales. 'bc' es un comando que consiste en una calculadora. Admite uso interactivo pero tambien acepta que le pasemos las operaciones desde otro proceso combinando entrada salida.

```
time ( echo "scale=300; 4*a(1)" | bc -l )
```

```
3.141592653589793238462643383279502884197169399375105820974944592307\
81640628620899862803482534211706798214808651328230664709384460955058\
22317253594081284811174502841027019385211055596446229489549303819644\
28810975665933446128475648233786783165271201909145648566923460348610\
454326648213393607260249141272
```

```
real 0m2.528s
user 0m2.520s
sys 0m0.010s
```

En un Pentium 200Mhz este comando tardó 3 segundos para 300 cifras y 20 segundos usando 600 cifras. Decimos esto para que vea que el tiempo que se tarda aumenta exponencialmente y que dependiendo de la potencia de su ordenador puede suponer bastante tiempo de proceso. Quizás tenga que variar el número de cifras significativas para poder medir tiempos con comodidad.

Este comando 'time' es un comando interno pero en Linux también hay un comando externo llamado de la misma forma. Para poder ejecutarlo con esta shell debería incluir el camino completo. No deseamos abusar de su escaso 'time' así que no lo comentaremos. Para buscar en el man el comando 'time' que hemos explicado o de cualquier otro comando interno tendría que mirar en la página del manual de bash.

Comando kill

Este comando se utiliza para matar procesos. En realidad envía señales a otros procesos, pero la acción por defecto asociada a la mayoría de las señales de unix es la de finalizar el proceso. La finalización de un proceso puede venir acompañada del volcado de la información del proceso en disco. Se genera un fichero 'core' en el directorio actual que solo sirve para que los programadores localicen el fallo que provocó esta prusca finalización del proceso. Por ejemplo si el proceso intenta acceder fuera del espacio de memoria concedido por el kernel, recibirá una señal que lo matará. Lo mismo ocurrirá si se produce una división por cero o algún otro tipo de error irrecoverable.

Un proceso unix puede capturar cualquier señal excepto la señal 9. Una vez capturada la señal se puede activar una rutina que puede programarse con toda libertad para realizar cualquier cosa.

'kill' por defecto es 'kill -15' envía un SIGTERM y generalmente provoca cierre ordenado de los recursos en uso. Esta señal puede ser ignorada, o puede ser utilizada como un aviso para terminar ordenadamente. Para matar un proceso resulta recomendable enviar primero un kill -15 y si no se consigue nada repetir con kill -

9. Este último -9 envía SIGKILL que no puede ser ignorada y termina inmediatamente. Solo fallará si no tenemos permisos para matar ese proceso, pero si es un proceso nuestro, kill -9 resulta una opción segura para finalizar un proceso.

Las señales actúan frecuentemente como avisos de que ha ocurrido algo. Existen muchos tipos de señales para poder distinguir entre distintas categorías de incidencias posibles.

Comando nice

El multiproceso está implementado concediendo cíclicamente la CPU en rodajas de tiempo a cada proceso que está en ejecución.

Existen dos números de prioridad. La prioridad NICE y la prioridad concedida por el Kernel mediante un algoritmo. Esta última no tiene porque coincidir con nice y puede valer más de 39. En cambio el comando nice solo acepta valores comprendidos entre 0 y 39, siendo 20 el valor por defecto. Cuando nice sube el valor significa que el proceso tiene baja prioridad. El comando 'nice -10' incrementará el valor nice en 10 (es decir baja la prioridad). Para bajar el valor de nice (Es decir para subir la prioridad) hace falta permisos de superusuario.

En un sistema con poca carga de trabajo no se notará apenas diferencia al ejecutar un comando con baja prioridad o con alta prioridad. Pero en un sistema con la CPU sobrecargada los comandos ejecutados con prioridad más baja se verán retrasados, ya que el kernel concederá más tiempo de CPU a los procesos con prioridad más alta.

Hay otros comandos de interés. Por ejemplo 'top' muestra los procesos que más CPU estén consumiendo. 'vmstat' saca información del consumo de memoria virtual.

Hay que tener en cuenta que el sistema gasta recursos en la gestión de los procesos. Por ejemplo si estamos compartiendo una máquina con otros usuarios y tenemos que realizar 15 compilaciones importantes terminaremos antes haciéndolas en secuencia una detrás de otra que lanzándolas todas a la vez. La avaricia de querer usar toda la CPU posible para nosotros puede conducir a una situación en la cual ni nosotros ni nadie sacará gran provecho de la CPU. El sistema realizará una cantidad enorme de trabajo improductivo destinado a mantener simultáneamente funcionando una gran cantidad de procesos que gastan mucha CPU y mucha memoria. La máquina comienza a usar el disco duro para suplir la falta de RAM y comienza a gastar casi todo el tiempo en el intercambio de la RAM con el disco duro. A esta situación se la denomina swapping.

Comando renice

Sirve para cambiar la prioridad de un proceso. Sigue la misma filosofía que el comando nice pero hay que identificar el o los procesos que deseamos cambiar su prioridad. Se puede cambiar la prioridad de un proceso concreto dado su PID o los procesos de un usuario dando su UID o todos los procesos pertenecientes a un determinado grupo, dando su GID. Tanto el comando nice como el comando 'renice' tienen mayor interés para un administrador de sistemas que para un usuario normal. Consulte las páginas del manual para más detalles.

Test

"... cuando se inicia un Sistema UNIX, el núcleo del Sistema UNIX(/unix) se carga en memoria y se ejecuta. El núcleo inicializa las interfases hardware y las estructuras de datos internas y crea un proceso del sistema, el proceso 0, conocido como el intercambiador (swapper). El proceso 0 se bifurca (fork) y crea el primer proceso de nivel usuario, el proceso 1.

El proceso 1 es conocido como proceso init ya que es responsable de la preparación, o inicialización, de todos los procesos subsiguientes en el sistema. Es responsable de disponer el sistema en modo usuario o multiusuario...." Esto es absolutamente válido en un sistema Linux.

Init arranca y se convierte en padre o superior de todos los procesos que componen el sistema Linux.

PASOS DEL PROCESO INIT

Primero, ejecuta /etc/rc.d/rc.sysinit, que configura el path, configura la red si es necesario, arranca el swapping y comprueba los sistemas de archivos; rc.sysinit se encarga de la inicialización del sistema. Por ejemplo, en un sistema en red rc.sysinit utiliza la información de /etc/sysconfig/network, también puede ejecutar rc.serial, si hay procesos de puerto serie que necesiten inicialización.

Init lee e implementa el fichero /etc/inittab. El fichero /etc/inittab describe como debe configurarse el sistema para cada nivel de ejecución y da un nivel por defecto. Este fichero declara que /etc.rc.d/rc debe ser ejecutados siempre que se ejecuta un nuevo nivel.

Cada vez que se cambia el nivel de ejecución, /etc/rc.d/rc arranca y para servicios. Primero, rc pone la función fuente de librería para el sistema (comúnmente /etc/rc.d/init.d/functions), que dice como

arrancar/parar un programa y como encontrar el pid de un proceso. El fichero rc averigua el actual nivel de ejecución y el anterior y notifica a linuxconfig el nivel apropiado.

El fichero rc arranca todos los procesos necesarios para que el sistema pueda funcionar, y busca un directorio rc para el nivel de ejecución: /etc/rc.d/rc<x>.d, donde <x> está numerado de 0 a 6. Entonces inicializa todos los scripts de arranque en el directorio del nivel de ejecución apropiado, para que todos los servicios sean arrancados correctamente.

El fichero /etc/inittab se duplica para ejecutar un proceso getty para cada consola virtual para cada nivel (niveles 2-5 tienen las 6; nivel 1, que es monousuario, solo tiene una consola, los niveles 0 y 6 no tienen consolas virtuales).

También, /etc/inittab describe como el sistema debe gestionar la traducción de Ctrl+Alt+Delete en algo como el comando /sbin/shutdown -t3-r now. /etc/inittab indica finalmente que debe hacer el sistema si falla el fluido eléctrico.

En este momento el proceso de arranque ha concluido y puede realizarse una conexión.

RUN LEVELS

Inittab distingue múltiples run levels, los cuales tienen cada uno sus procesos a inicializar. Hay run levels válidos desde el 0 hasta el 6.

El run level "1" inicia el sistema en modo "monousuario". Cuando iniciamos en modo monousuario no se cargan muchos de los demonios y programas que se cargarían cuando entramos normalmente en modo multiusuario.

El run level "2" inicia el sistema en modo multiusuario sin soporte para NFS (network file system).

El run level "3" inicia el sistema en modo multiusuario. Este es el run level que generalmente se encuentra por defecto en el sistema.

El run level "4" inicia el sistema directamente en el administrador de ventanas Xwindows. Este run level es muy práctico para los que acostumbra a trabajar todo el tiempo en Xwindows y no utilizan la consola casi nunca.

Dependiendo de la distribución de Linux el número de este run level puede variar entre "4" y "5".

Los run levels "0" y "6", a diferencia de los otros run levels son utilizados por el sistema en el momento de apagar el sistema. El run level "0" se utiliza cuando se desea apagar el sistema definitivamente, sin intenciones de reiniciar, a este proceso se le conoce como "halt". El run level "6" se utiliza, sin embargo, al momento de reiniciar el sistema.

Estos dos run levels dan instrucciones al sistema de que hacer antes de apagarlo o reiniciarlo.

El archivo "inittab" es la parte donde se define cual será el run level que se cargará cada vez que se inicie la máquina y se verá así de la siguiente forma:

```
localhost/# more /etc/inittab
```

```
#default runlevel.
```

```
id:3:inittab:
```

```
localhost#
```

Para cambiar el run level por defecto solo se debe cambiar el número que aparece entre "id:" e ":inittab" por el nivel que se desea.

Si se tiene instalado el administrador de booteo Lilo en el sistema se puede iniciar en el modo que se quiera al momento de elegir que Sistema Operativo se iniciará agregando el número del run level que se quiera precedido por el nombre con que identifica el Lilo a Linux.

Boot

El proceso de boot o inicio de la máquina se realiza mediante los siguientes pasos:

1. BIOS: el Basic Input/Output System es el nivel más bajo de interfaz entre el computador y los periféricos. El BIOS realiza chequeos de integridad de la memoria y busca instrucciones en el Master Boot Record (MRB) en el floppy o en el disco duro.
2. El MRB apunta al cargador de boot, en este caso el LILO (LILO: GNU/Linux boot Loader).
3. El LILO preguntará por la etiqueta del sistema operativo que identifica cual kernel debe correr. Después cargará GNU/Linux.
4. La primera cosa que el kernel hace es ejecutar el programa init. Init es el root/parent de todos los procesos que se ejecutan en GNU/Linux.
5. Basado en un run-level apropiado, los scripts son ejecutados para iniciar varios procesos y hacer el sistema funcional.

Proceso init de GNU/Linux

El proceso init es el último paso en el booteo de una máquina, y está identificado con el ID de proceso 1 (ver figura 2.1).

ID de Proceso	Descripción
0	The Scheduler
1	The init Process
2	kflushd
3	kupdate
4	kpiod
5	kswapd
6	mdrecoveryd

Figura 3.1: Procesos del sistema

Init es el responsable de iniciar los procesos del sistema definidos en el archivo `/etc/inittab` ver (figura 2.2). Hasta en el apagado del sistema, init controla la secuencia y los procesos por terminar, pero init nunca se 'apaga'.

```
# /etc/inittab
#
# The default runlevel is defined here
id:5:initdefault:
# First script to be executed, if not booting in emergency (-b)
# mode
si:l:bootwait:/etc/init.d/boot
# /etc/init.d/rc takes care of runlevel handling
#
# runlevel 0 is System halt (Do not use this for initdefault!)
# runlevel 1 is Single user mode
# runlevel 2 is Local multiuser without remote network (e.g. NFS)
# runlevel 3 is Full multiuser with network
/etc/inittab"100L, 3049C
.
```

Figura 3.2: Así luce el archivo `/etc/inittab`

Booteo

GNU/Linux tiene seis estados de operación, donde '0' es el estado 'apagado' y '3' y siguiente son estados completamente operacionales con todos los procesos esenciales corriendo para su interacción. Para el booteo del sistema Linux hará:

- Ejecutar el `/sbin/init` que iniciará todos los otros procesos. Este 'subirá' la máquina iniciando los procesos definidos en el archivo `/etc/inittab`.
- La máquina será booteada hasta el nivel de ejecución (run-level) definido en el `initdefault` del archivo `/etc/inittab`.
id:5:initdefault: En este ejemplo el nivel de ejecución '5' es escogido.
- Uno de los procesos iniciados por init es `/sbin/rc`. Este script corre una serie de scripts en los directorios `/etc/rc.d/rc0.d/`, `/etc/rc.d/rc1.d/`, `/etc/rc.d/rc2.d/`, etc.
- Los scripts en estos directorios son ejecutados para cada estado del booteo hasta que este se vuelva operacional. Los scripts que comienzan con 'S' son scripts de inicio, mientras los que comienzan con 'K' son scripts de terminación. Los números que siguen estas líneas denotan el orden de ejecución.

Si se instalaron todos los daemons (procesos en background), GNU/Linux los iniciará todos, haciendo lenta la máquina. Se pueden iniciar/parar los daemons individualmente cambiando al directorio `/etc/rc.d/init.d` y dando el comando `start` (iniciar), `stop` (parar), `status` (ver el estado), `restart` (reiniciar) o `reload` (volver a cargar) el daemon. P.e. para detener el servidor web se debe hacer en una consola:

- `cd /etc/rc.d/init.d`
- `httpd stop`

Se puede usar el comando `ps -aux` para ver todos los procesos corriendo en la máquina

Niveles de ejecución en GNU/Linux

En el nivel de ejecución 3 se bootea para texto o modo consola, y en el nivel de ejecución 5 ya se ha booteado al modo login del entorno gráfico (4 para slackware y BSD)

NdE	Estado
0	Apagado
1	Modo de Usuario sencillo
2	Multiusuario sin servicios de red
3	Inicio de texto por defecto. Modo multiusuario completo.
4	Resevado para uso local.
5	X-windows(entorno grafico)
6	Reboot
s o S	Modo de mantenimiento (Slackware)
M	Modo de Multiusuario (Slackware)

Se puede usar el comando 'init #' donde # es 0,1,3,5,S dependiendo del nivel en el que se desee estar. También sirve 'telinit #'.

Los sript para un determinado nivel de ejecución corren durante el booteo y el apagado. Los scripts ubicados en /etc/rc.d/rc0.d/ inician los daemons requeridos por el sistema. Este sistema provee una manera ordenada de llevar la máquina a diferentes estados de producción y mantenimiento.

TIP: Un listado de los estados y niveles de ejecución de todos los servicios que pueden ser iniciados por init: chkconfig -list

Estados del Sistema

El funcionamiento del Sistema Operativo Unix, está ligado al estado o nivel de ejecución que se haya establecido para operar. Los niveles o estados de ejecución Unix se identifican con número que va del 0 al 6 y un estado adicional que se conoce como **s o S** siempre que se arranca el sistema, éste entra al nivel por default. Sin embargo es posible cambiar a otros niveles de ejecución en función de las tareas que desee realizar.

0	Nivel Prom Monitor
1	Administrativo
2	Nivel multiusuario
3	Nivel Multiusuario, con acceso a recursos en red
4	Actualmente sin uso
5	Usado por Xwindows
6	Rearranque a NivelPor Default

Para apagar el sistema, el S.O. realiza un cierre adecuado de todos los procesos que están corriendo en el sistema, cambia el nivel monousuario y deshabilita los recursos del sistema por consiguiente, siempre que haya necesidad de apagar el equipo, debe ejecutar la orden **shutdown**, cuya función principal es efectuar un proceso normal de apagado, no hacerlo así implicará daños a la información que en ocasiones pueden ser irrecuperables.

En los sistemas multiusuario y estaciones de trabajo, al usar **shutdown**, el sistema queda en estado **0** conocido también como PROM monitor. Si este fuera el caso, entonces

\$shutdown -h 0

Apaga el sistema inmediatamente es equivalente a

\$shutdown -h now

\$shutdown -k "En unos minutos se apagará el sistema"

Solo envía una aviso a todos los usuarios activos

\$shutdown -h 21:30 -k "El Sistema se apagará a las 21:30 hrs"

El sistema iniciará el proceso de cierre a las 21:30 hrs

\$shutdown -h 23:45 "El sistema se apagará a las 23:\$5"
Se programa el proceso de apagado para las 23:45

shutdown se utiliza también para rearrancar el sistema en nivel 6, que equivale a reiniciar en el nivel por omisión.

\$shutdown -r now

CAMBIOS EN NIVELES DE EJECUCION

Unix proporciona un conjunto de herramientas diseñadas especialmente para trabajar con los diferentes estados posibles del Sistema. A medida que se familiarice con éstos, podrá identificar cual es el apropiado para cada caso en particular.

- **shutdown**

Es principalmente para apagar lógicamente el equipo.

- **init**

Se utiliza para cambiar de un nivel a otro. Cuando se cambia el sistema de un nivel superior a un nivel inferior, **init** efectúa cierres de ciertos procesos de sistema. Cuando el sistema pasa a un nivel superior, **init** crea los procesos necesarios para entrar al nivel especificado.

\$init 1

Cambia a nivel administrativo, por tanto se cerrarán los procesos que hacen posible la capacidad de **multiusuario**.

Si se especifica cambio a nivel 1 pedirá el password de root, y si este se proporciona correctamente hace el cambio, de otra manera se coloca en nivel por omisión.

\$init s

Tiene el mismo significado que **init 1**.

Cambia a modo multiusuario. Por lo tanto crea los procesos que habilitan las capacidades de multiusuario.

- **reboot**

Se utiliza sin opciones, y siempre rearrancará el sistema al nivel establecido como el default.

- **halt**

No se recomienda usarlo, puesto que detiene inmediatamente el sistema, sin cerrar los procesos en ejecución.

- **stop A**

Funciona normalmente en estaciones de trabajo y equipos multiusuarios. Va a modo de PROM Monitor, pero todavía siguen abiertos los sistemas de archivos.

Para saber en que nivel de ejecución está trabajando, simplemente ejecute.

\$runlevel

Archivos de inicialización

Al arrancar el sistema, el proceso **init** lee una serie de órdenes de inicialización que se encuentran en ciertos archivos de este tipo, localizados en el directorio `/etc/rc.d`.

Los archivos de inicialización que juegan un papel importante son: `rc.0`, `rc.s`, `rc.M` y `rc.K`

El script `rc.0` tiene la tarea básica de bajar a nivel de ejecución 0 (apagado) o nivel 6 (rearranque), y por tanto se encarga de terminar los procesos en ejecución usando las señales **TERM** y **KILL**, y luego desmonta los sistemas de archivos para posteriormente ir al nivel 0 (**halt**) o a nivel 6 (**reboot**),

El script **rc.s** tiene la función de colocar el sistema en el nivel administrativo o monousuario al arrancar el sistema, y por consiguiente, se encarga de habilitar la partición de intercambio y los sistemas de archivos.

El script `rc.M` se encarga de arrancar los procesos necesarios para que el sistema pueda quedar disponible en modo multiusuario incluyendo además los servicios de red.

El script `rc.K` es activado cuando el sistema baja de un nivel superior al estado 1, por tanto se terminan los procesos con las señales **TERM** y **KILL** para los procesos de usuario y procesos de sistema, dejando solo los necesarios para que el sistema quede en modo administrativo.