**4.- Control y estructura de los procesos**

Controlar y hacer operativo todo este sistema de gestión de los procesos requiere la definición de una serie de estructuras de datos que contengan toda la información asociada a cada proceso, así como las listas y tablas para controlar las diferentes colas implicadas en la planificación. Dos de estas estructuras básicas de información son el bloque de control de proceso (**PCB**) y el bloque de control de sistema (**SCB)**.

Por otra parte, es necesario comprender el mecanismo que permite al procesador cambiar el proceso que está ejecutando por otro. Este mecanismo se denomina **cambio de contexto**. La problemática de la penalización en tiempo de CPU que supone el cambio de contexto nos llevará a la aparición de técnicas más óptimas, como los sistemas multihilo basados en **hilos de ejecución o threads.**

**Bloque de control de proceso (PCB)**

Cada proceso se representa en el sistema operativo como una estructura de datos llamada bloque de control de procesos (en inglés ***process control block***o ***PCB***) que describe el proceso y contiene una serie de campos de información:

Identificador del proceso.

**Estado**: estado actual del proceso (preparado, bloqueado, en ejecución, etc.)

**Contador del programa**: indica la dirección de la siguiente instrucción de este proceso que se ejecutará.

**Registros de la CPU**: acumuladores, punteros de pila (stack) y registros generales.

**Información de planificación**: como la prioridad del proceso y el valor del quantum.

**Información de la gestión de la memoria**: definición de la ventana de memoria con el registro base y límite, tabla de páginas o segmentos.

**Información de contabilidad**: tiempo de CPU, tiempo consumido, etc.

**Información del estado de las entradas/salidas**: dispositivos de entrada/salida asignados al proceso, lista de archivos abiertos, etc.

**Puntero a la memoria**: apunta al nodo siguiente de la lista enlazada de PCB.

Para gestionar los procesos el sistema operativo debe leer, tratar y manipular estas estructuras de datos.

**Bloque de control del sistema (SCB)**

Los sistemas operativos disponen también de una estructura llamada bloque de control del sistema (del inglés ***system control block***o ***SCB***) para controlar y reunir información de todas las estructuras de datos de los procesos, las interrupciones y las excepciones. El bloque de control de sistema contiene habitualmente la siguiente información:

* Un puntero hacia el descriptor (PCB) del proceso que está haciendo uso del procesador (proceso activo)
* Un puntero a una cola de descriptores de los *procesos preparados* (ready)
* Un puntero a una cola de descriptores de los *procesos bloqueados* (waiting)
* Un puntero a una cola de descriptores de los *procesos suspendidos preparados*
* Un puntero a una cola de descriptores de los *procesos suspendidos bloqueados*
* Punteros a los **vectores de interrupciones y excepciones** que son referencias a las rutinas necesarias para atender estos eventos.

Una **interrupción**, sea de software (**llamadas al sistema**, por ejemplo *write, read, open,* …, es decir, operaciones sobre ficheros) o de hardware (reloj, dispositivos de entrada/salida –*pulso una tecla en el teclado o hago clic en el ratón, la impresora “me dice” que no tengo papel o que está ocupada, …* - , reinicialización, etc.), es la presencia de un evento que obliga el sistema operativo a tomar el control del procesador para analizar y tratar la situación mediante rutinas específicas.

Una **excepción** es un tipo de interrupción generada al sistema por un error (división por cero, desbordamiento de memoria ...).

Una interrupción se produce cuando, por ejemplo, desde mi aplicación voy a escribir en el disco duro, en un fichero. Es una llamada al sistema, una situación perfectamente contemplada. Imaginemos que le hemos indicado un nombre de archivo que no existe o que no hay espacio en el disco para escribir. Eso provocaría una **excepción**. No es algo esperado, pero hay que controlarlo.

Cundo programamos, las excepciones, de aparecer, es en tiempo de ejecución del programa.

**Vector de interrupciones / excepciones**

Es una tabla de las direcciones de memoria donde están situadas las rutinas a ejecutar por el tratamiento de la interrupción / excepción generada.

**Cambio de contexto**

Cuando un proceso en ejecución tiene que abandonar la CPU por algún evento (interrupción, agotamiento del quantum, operación de entrada/salida, etc.) y se debe comenzar o reanudar otro proceso es necesario hacer un cambio de contexto. Para hacer esto hay que guardar el contexto del proceso que sale, es decir, su estado, registros, contador de programa, etc., que está representado por su PCB (bloque de control de proceso) y recuperar el contexto del proceso que entra en ejecución.

Los cambios de contexto permiten la conmutación de la CPU entre diferentes procesos, pero también suponen una penalización por el tiempo necesario para efectuarlos. Para intentar reducir esta penalización (**overhead**) se utilizan nuevas estructuras como los **hilos de ejecución (threads).**

**Prioridades de los procesos**

Uno de los factores fundamentales que intervienen en la planificación a corto plazo de los procesos es la prioridad, que determina la cola donde un proceso tendrá que esperar su turno. En cada proceso se le asigna una prioridad en función de si es más crítico o más urgente y de los recursos que necesita, lo que determinará finalmente la frecuencia de acceso al procesador.

**Tipo de prioridades**

Podemos establecer una clasificación de las prioridades en función de su posibilidad de variación a lo largo de la ejecución del proceso. Así, tendremos:

Prioridad estática: Aquellas que no pueden ser modificadas mientras se ejecuta el proceso.

Prioridad dinámica: Cuando un proceso puede modificar su prioridad en función de determinados eventos.   
 Cada proceso tiene asignada una prioridad que puede ser estática, dinámica o una combinación de ambas. El problema de la prioridad estática es la **inanición**, es decir, que un proceso nunca se ejecute por no tener nunca la prioridad suficiente. Para solucionar esto se utiliza la prioridad dinámica por envejecimiento (aging), en la que los procesos van aumentando su prioridad en función del tiempo que hace que están en espera.

Por otra parte, la prioridad de un proceso está compuesta por dos factores:

**Prioridad asignada por el sistema operativo** en función del tipo de proceso, algoritmos de planificación, políticas para prever la inanición, etc.

**Prioridad asignada por el propio usuario**. Naturalmente sólo afecta a los procesos de usuario y permite tener en cuenta sus intereses y necesidades. Esta libertad conlleva un cierto riesgo, como, por ejemplo, **dejar bloqueada el resto del sistema si un programa con prioridad alta asignada por el usuario entra en un bucle infinito.**

En los sistemas Unix/Linux, la prioridad de usuario se llama prioridad **nice** y se puede asignar con las órdenes ***nice*** y ***renice.***

**Órdenes para manejar la prioridad**

*“Más es menos”:* nice significa 'amable' en inglés. Así se entiende mejor por qué cuando mayor es el valor de nice menos prioridad tiene el proceso y por tanto es *más "amable" con el resto de procesos que compiten por los recursos.*

Todos los procesos tienen una prioridad llamada **prioridad nice**. Los procesos con más prioridad tienen valores negativos de nice y utilizan más recursos que los demás.

**El total de prioridades (PR) de Linux es de 140**, desde la prioridad 0 hasta la prioridad 139. Las 40 que vemos en las herramientas como renice, nice o top/htop, son en realidad las prioridades que van desde 100 a 139, las prioridades de procesos de usuario.

Las prioridades reales desde la 0 a la 99 son prioridades de tiempo real, no podemos modificarlas con herramientas de usuario, y, por ejemplo, en top/htop las veremos como **rt.**

El valor de nice (NI) que establece la prioridad máxima a un proceso es **-20** y el valor que establece la prioridad mínima es **+19**. Si la prioridad no está definida, cada proceso se ejecutará con una prioridad con valor nice de 0.

**Orden ps para ver el nice y otra información importante**

$ps –eo pid, ppid, ni, comm

**Fórmula importante: PR = 100 + 20 + NI (ver en top/htop)**

(prioridades entre 0 y 99 no las podemos modificar)

La columna etiquetada como **PR(iority)**hace referencia a la prioridad, por defecto, 20.

Ese 20 hace referencia a una prioridad media por defecto. Las prioridades de procesos de usuario se numeran desde 0 a 39 (**realmente es desde 100 a 139**). Mientras menor es el número, más alta será la prioridad, por lo que una prioridad cercana a 0 indicará mayor prioridad, y el proceso que la posea tendrá más chances de ser planificado en el procesador, que uno que tenga una prioridad más baja, cercana al 39.

La columna **NI(ce)** hace referencia a una prioridad relativa que por lo general mapea con la prioridad absoluta PRI, y que depende del estado y carga de procesamiento, pero que va desde -20 a +19, donde -20 es la mayor prioridad de proceso de usuario (equivalente a un PRI de 0) y +19 es la menor prioridad de usuario (equivalente a un PRI de 39), siendo el NI de 0 la prioridad media, equivalente a un PRI de 20.

**Orden *nice***

La orden nice permite arrancar un proceso asignándole de entrada una determinada prioridad y tiene como argumento el nombre de la orden o el PID del proceso del proceso sobre el que queremos actuar. Si no especifican el **nivel de prioridad** (**PR)** queda definida por defecto en 10. Un proceso lanzado sin nice tiene una prioridad de 0.

**$nice -n 19 COMANDO**

Esta orden deja con **prioridad mínima** al comando especificado.

**Observación:** cuidado porque estoy asignando un valor de +19.

**Orden *renice***

Para cambiar la prioridad nice de un proceso en ejecución se utiliza la orden renice. Por ejemplo:

**$renice -n -5 987 -u root**

Esta orden dará prioridad -5 al proceso de PID = 987 y a todos los procesos del usuario root.

También se puede cambiar la prioridad nice con la orden interactiva top.

Podemos ver un esquema de la prioridad nice y la sintaxis de las órdenes nice y renice a la tabla

-20 0 +19

|  |  |  |
| --- | --- | --- |
| **Prioridad alta** |  | **Prioridad baja** |
| nice - - 20 orden | nice -19 orden |
| **nice -n -20 orden** | **nice -n 19 orden** |
|  | | |
| renice -20 pid |  | renice 19 pid |
| **renice –n -20 pid** | **renice –n -19 pid** |

**Herencia de la prioridad**

Los procesos hijo heredan la prioridad del padre, pero si se cambia la prioridad del proceso padre, los procesos hijos ya nacidos no cambian de prioridad.

**5.- Comunicación con los procesos**

Hay diferentes mecanismos que permiten la comunicación entre los procesos, algunos de los cuales veremos en este apartado. Por un lado, todo proceso tiene asociado unos archivos (stdin, stdout, stderr) que reciben la comunicación de entrada y salida de datos. Estos flujos de datos de los procesos pueden ser intercomunicados mediante tuberías.

Por otra parte**, todo proceso que finaliza devuelve información en una variable de retorno**. Esta variable puede ser leída por otro proceso y condicionar su funcionalidad.

Finalmente, un proceso puede recibir información y órdenes mientras se está ejecutando mediante el envío de determinadas señales.

Entre los recursos asociados a un proceso está **la interfaz estándar de comunicación con el exterior**. Esta interfaz consta de tres archivos conocidos como archivos de entrada, de salida y de errores estándar. A Unix/Linux, cuando un proceso abre un archivo, el núcleo del sistema le entrega un número entero que el proceso utilizará para realizar operaciones de entrada y salida. Los descriptores asociados a los archivos estándar son los siguientes:

0, entrada estándar, asociada por defecto al terminal.

1, stdout, salida estándar, asociada por defecto a la pantalla del terminal.

2, stderr, salida estándar de errores, asociada por defecto a la pantalla del terminal

**Valores de retorno**

Todo proceso que finaliza vuelve un número comprendido entre 0 o 255 que representa la causa por la que finalizó. A continuación, podemos ver algunos valores de retorno.

La *variable de entorno* **?** contiene el valor de retorno de la última orden ejecutada y se puede visualizar con **echo $?**.

**Valores de retorno de un proceso**

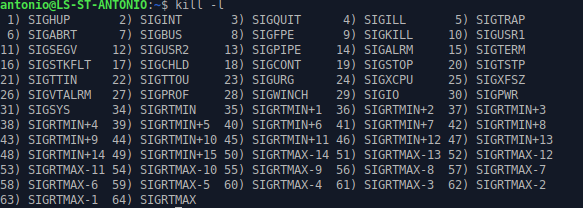
* **0**, finalización correcta del proceso. Si este valor se utiliza en una comparación o iteración, se considera como un valor lógico (en este caso **true**). Cualquier valor distinto de 0 se considera falso (false) ($ls)
* **1**, indica error o no ha habido éxito. Finalización anormal del proceso. *($grep pepepepepe /etc/passwd*)
* **2**, hay un error en la sintaxis, quizás faltan argumentos *($ls –pepe*)
* **126**, intentas ejecutar y no es ejecutable o no tienes permisos (tengo un fichero de texto, alumnos, y lo intento ejecutar: *$./alumnos*).
* **127**, no se encuentra el comando *($pepepe*)
* **129…160**, quiere decir que el proceso ha finalizado como consecuencia de una señal. Restando 128 al valor de retorno se obtiene el código de esta señal. Por ejemplo, si terminas un proceso con Control + C, te devuelve el valor 130. En este caso, 130 -128 = 2 que es el valor de la señal SIGINT (ver señales a continuación).

**Señales**

Las señales son **interrupciones que recibe el proceso mientras está en ejecución** para indicar que se ha producido algún evento significativo ante el que debe responder.

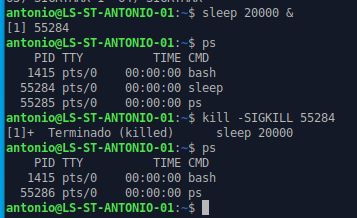
Las señales pueden ser enviados por el núcleo del sistema, por otros procesos o por el mismo usuario. Algunas de estas señales pueden ser capturados (*trap*), es decir, pueden ser ignorados o recibir un tratamiento específico para el proceso si así está programado. En cambio, las señales **SIGKILL (9)** y **SIGSTOP** (**19**) se deben atender obligatoriamente y no se pueden ignorar.

Hay un buen número de señales que podemos visualizar con la orden kill (opción -l)



Podemos destacar las señales más importantes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Núm.** | **señal** | **Trap** | **Descripción** |
| **1** | **SIGHUP** | **SÍ** | Finalización por cierre del terminal o del proceso padre (como un exit) |
| **2** | **SIGINT** | **SÍ** | Finalización por teclado (Ctrl + C) |
| **9** | **SIGKILL** | **NO** | **Finalización forzada (**cuando no hay más remedio que terminar con un proceso. El sistema puede quedar inconsistente**)** |
| **15** | **SIGTERM** | **SÍ** | Finalización ordenada por defecto (es una buena forma de terminar, si se puede, con un proceso). |
| **18** | **SIGCONT** | **SÍ** | Sigue un proceso parado |
| **19** | **SIGSTOP** | **NO** | **Detener el proceso** |
| **20** | **SIGSTP** | **SÍ** | Detener el proceso por teclado (Ctrl + Z) |



Hay un grupo de señales de finalización que tienen como objetivo la finalización del proceso y la liberación de los recursos que utilizaba (señales 1, 2, 9 y 15). Otros tienen como objetivo detener el proceso, pero sin sacarlo de memoria para poder ponerlo en marcha de nuevo en el mismo punto en que se dejó (señales 19 y 20).

Para obtener información completa de las señales:

**$man 7 signal**

**Órdenes relacionadas con señales**

A continuación, se detallan una serie de combinaciones de teclas y órdenes de consola relacionadas con el envío de señales a los procesos.

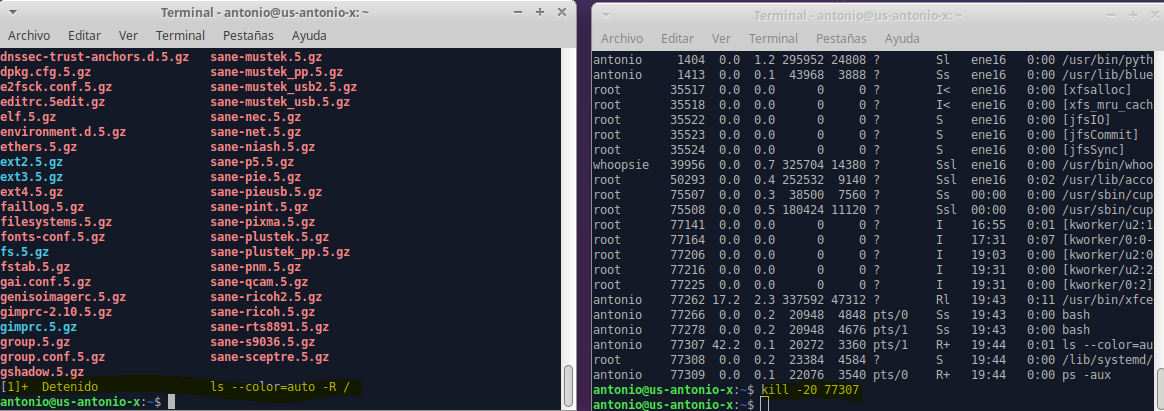
**Señales de teclado**

Hay una serie de combinaciones de teclas que envían señales directamente al proceso que se está ejecutando en primer plano:

**Ctrl + Z**: Envía la señal 20 (SIGTSTP). Esta señal detiene el proceso, pero lo deja en la memoria y se puede reanudar en el punto en el que quedó mediante una señal 18 (SIGCONT).

**Práctica**: abro dos consolas, en una ejecuto **ls –R /** y en la otra veo el PID que tiene y lo detengo: **kill -20 77307**

Después, hago que siga ejecutándose: **kill -18 77307**



También puedo ejecutar el ls –R / y pulsar Control + Z, veo el PID (ps) y lo reanudo con -18 PID.

**Observación**: algunos procesos detenidos se deben reanudar con el comando fg.

**Ctrl + C**: Envía la señal 2 (SIGINT). La mayoría de aplicaciones están programadas para finalizar cuando reciben esta señal, pero hay que tener en cuenta **que se puede programar una aplicación para que capture esta señal y la ignore.**

**Orden kill**

Podemos utilizar la orden interna **kill** (type kill para comprobar que es interna) para enviar señales a cualquier proceso que hayamos creado tanto en primer como en segundo plano. Además, si somos administradores podemos enviar señales a procesos de otros usuarios.

La orden kill necesita identificar el proceso mediante su PID (identificador de proceso), aunque si se trata de un proceso en segundo plano de nuestro terminal, también puede utilizar su número de trabajo (job) precedido del símbolo %.

Por defecto, tanto el orden kill como killall envían la señal 15 (SIGTERM) de finalización ordenada del programa.

Veamos algún ejemplo:

**#kill 2343**

En este caso se envía la señal de finalización ordenada SIGTERM al proceso de PID = 2343. En algunos casos, los procesos no responden a la señal de finalización por defecto y hay que enviar la señal de finalización forzada, que no se puede capturar y debe obedecer. Cualquiera de las sintaxis siguientes es válida:

***#kill -9 2343***

***#kill -SIGKILL 2343***

**Orden killall**

Con la orden kill podíamos enviar señales a un proceso determinado definido por su PID. **Si queremos enviar señales a un proceso a partir de su nombre debemos usar killall.**

***#killall -9 firefox***

Esta instrucción cerrará todos los procesos “firefox” abiertos por el usuario que ejecuta la orden.

**Orden trap**

Es una orden interna que nos permite capturar una señal y especificar lo que queremos hacer cuando nos llegue. Su sintaxis es **trap [órdenes] [señales**]:

**Ejemplo: trap "" SIGTERM**

Esta orden, dentro de un script, capturará la señal de finalización SIGTERM y lo ignorará, ya que como órdenes a seguir le hemos puesto una cadena vacía.

**Observación:** Veremos un ejemplo en la hoja de actividades.

**6.- Secuencia de arranque del sistema**

En el momento que encendemos el ordenador, empieza una larga secuencia de acciones y ejecuciones de procesos que finalmente nos lleva hasta que en la pantalla nos aparece el entorno gráfico del sistema operativo o una consola de texto nos invita con el **prompt** (indicador de órdenes) a introducir una orden.

Hacemos ahora un repaso resumido de toda esta secuencia y, a lo largo de los diferentes apartados, pondremos especial atención en los últimos eventos que culminan en la creación del árbol de jerarquía de procesos y en la carga del intérprete de comandos o shell.

**GRUB**

Acrónimo de (***GRand Unified Bootloader***). Cargador de arranque múltiple del proyecto GNU que se utiliza para poner en marcha uno de los sistemas operativos instalados en un mismo ordenador.

1.- Después de reiniciar (reset) o de poner en marcha el interruptor del equipo, los componentes electrónicos reciben alimentación. El microprocesador comienza a ejecutar instrucciones desde una posición del mapa de memoria determinada, llamada vector de reset o de reinicio y comienza a ejecutar un programa especial llamado **BIOS** (Basic Input / Output System) que está alojado en la memoria del sistema de tipo ROM (memoria de sólo lectura).

2.- Este programa que incorpora cada placa base constituye el llamado firmware y contiene rutinas de comprobación de memoria, detección de dispositivos de hardware, rutinas POST (**Power-On Self Test**) para la verificación de los componentes, …

Puedes ser que también ofrezca al usuario la interacción opcional con el sistema de memoria CMOS, que guarda la configuración de diferentes características del sistema tales como el reloj de tiempo real, la memoria secundaria, la secuencia de arranque, etc.

3.- Una vez ejecutadas las instrucciones del BIOS/UEFI, la última acción que realiza es la búsqueda del sistema operativo en la memoria secundaria, habitualmente el disco duro, para cargarlo en la memoria. Para hacer esto se dirige al primer sector del disco duro llamado **MBR (Master Boot Record)**.

4.- Tradicionalmente, el MBR **contiene un gestor de arranque** simple y la tabla de particiones del disco duro, que indica cuál es la partición activa que contiene el sistema operativo a cargar. Sin embargo, casi todas las distribuciones Linux usan programas gestores de arranque múltiple (bootloader) más complejos, como GRUB.

5.- Así pues, por ejemplo, en un sistema *Debian*, se inicia en el MBR la **fase 1**de GRUB llamada **primer gestor de arranque** (**Initial Program Loader** o **ILP**).

6.- Como casi no hay espacio en el MBR para un programa complejo, es necesaria la ejecución de una fase llamada **fase 1.5**, que contiene código adicional y está situada *en los primeros sectores tras el MBR*, siempre en la primera pista del disco duro por motivos de compatibilidad.

7.- El resto del código de GRUB se ejecuta en la **fase 2** y normalmente se instala en el sector de arranque de la partición donde está instalado el sistema Linux. Este código interpreta la configuración del archivo ***/boot/grub/grub.cfg*** (soporta diferentes sistemas de archivos y permite el paso de parámetros en el núcleo del sistema, además de interactuar con el usuario mostrando el menú de selección en pantalla).

8.- Una vez elegida la partición de arranque a partir del menú de GRUB se comienza la carga del núcleo del sistema operativo propiamente dicho. En el caso de Linux, una vez cargado el núcleo del sistema operativo comienza la creación de procesos que se ejecutan en un orden determinado y culmina en la creación de toda una estructura jerárquica de procesos y servicios**.**

**Jerarquía de procesos (PID, PPID)**

En Unix/Linux todos los procesos se identifican con un número entero de 16 bits que se asigna secuencialmente en cada nuevo proceso que se crea. Este número es único para cada proceso y se llama identificador de proceso o **PID** (del inglés *Process IDentifier*). Además, todo proceso, a excepción del proceso raíz **/sbin/init  con PID = 1 (dependiendo de la distribución, este proceso puede cambiar. En Ubuntu /sbin/init es un enlace simbólico a /lib/systemd/systemd, es decir, el proceso padre realmente es systemd)**, ha sido creado por un proceso padre. Así, pues, un proceso padre puede tener muchos procesos hijos, pero cualquier proceso sólo tiene un proceso padre, identificado por su **PPID** (Parent Processs IDentifier ). Esto crea una estructura jerárquica de procesos en forma de árbol con el proceso init como raíz.

Proceso padre y proceso hijo pueden o no compartir recursos y espacio de memoria, y estar o no sincronizados, es decir, se pueden ejecutar concurrentemente o bien el proceso padre puede permanecer en espera hasta la finalización del proceso hijo.

**Trabajos en segundo plano de ejecución**

Cuando un proceso padre crea un proceso hijo hay dos posibilidades en términos de ejecución:

1.- El proceso padre espera la finalización del proceso hijo.

2.- Padre e hijo se ejecutan concurrentemente.

De esta manera en el entorno del proceso de shell, cuando llamamos a una orden de sistema esta puede generar un proceso o procesos hijos del shell que se pueden ejecutar en:

**Primer plano (foreground):** cuando el shell queda a la espera de la finalización de la orden ejecutada y por tanto el terminal no acepta nuevas órdenes hasta la finalización del proceso hijo.

**Segundo plano (background):** él proceso o procesos hijos se ejecutan concurrentemente al proceso shell padre, que continúa aceptando nuevas órdenes.

**Directiva &**

Para poder enviar la ejecución de una orden a segundo plano sólo hay que añadir el símbolo & al final de la línea de comandos. Por ejemplo:

**#find / -name "\*.txt" 🡪 primer plano**

**#find / -name "\*.txt" & 🡪 segundo plano**

En este último ejemplo, mientras el sistema busca en todo el árbol de directorios los archivos con extensión txt, deberíamos volver enseguida a tener el indicador de sistema (prompt), que nos aceptará nuevas órdenes. El problema, en este ejemplo, es que los mensajes de la orden find en segundo plano aparecen en la consola y dificultan el trabajo, pero siempre podemos redirigir la salida principal y la de errores hacia un archivo:

**#find / -name "\* .txt" > listado 2> errores &**

**[1] 2456**

#

Se nos muestra el número de trabajo [1] y el PID asociado al proceso, 2456. Después aparece de nuevo el indicador que nos invita a introducir nuevas órdenes, como, por ejemplo:

**#yes**

La orden yes genera continuamente el carácter "y" y bloquea el terminal. Podemos utilizar la combinación de teclas Ctrl + Z, que enviará este proceso a segundo plano, en estado **detenido**.

**#^Z**

[2]+ Detenido yes

Ahora volvemos a ejecutar la misma orden, pero con más cuidado, enviando la orden directamente a segundo plano y su salida al dispositivo null (sumidero) para que no nos moleste:

**#yes > /dev/null &**

**[3] 2484**

Ya tenemos un tercer trabajo en segundo plano con PID = 2484.

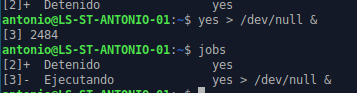
**Orden jobs**

La orden jobs nos permite visualizar los trabajos que tenemos en segundo plano y ver el estado en el que se encuentran. Si se han ejecutado las anteriores órdenes veremos algo parecido a esto:

**# jobs**

**[2]+ Detenido yes**

**[3]- Ejecutando yes> /dev/null &**

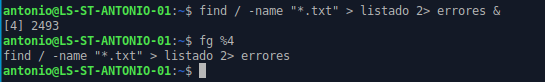


Como puedes ver, la primera orden find ya no aparece, pues ha finalizado. Tenemos el segundo trabajo yes que hemos detenido y pasado a segundo plano con la combinación Ctrl + Z y un tercer trabajo ejecutándose en segundo plano, pero con su salida redirigida al sumidero.

En segundo plano podemos tener procesos detenidos y procesos ejecutándose.

**Órdenes fg y bg**

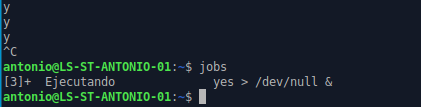
Estas órdenes permiten retomar una tarea parada en segundo plano en el punto donde se detuvo. Esta tarea se puede reanudar en primer plano con fg o en segundo plano con bg. Para indicar cuál de las tareas paradas en segundo plano queremos retomar debe indicar el número de trabajo precedido del símbolo %.



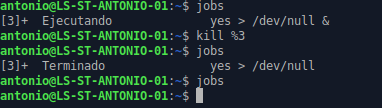
En nuestro ejemplo anterior:

**#fg %2**

Con esta orden retomaremos en primer plano el trabajo [2] que estaba parado. Comenzará a salir letras "y" en la pantalla. Podemos volver a detener el proceso y enviarlo a segundo plano con la combinación Ctrl + Z o parar el proceso definitivamente con Ctrl + C y podremos volver a llamar jobs:



Ahora sólo nos queda un trabajo en ejecución en segundo plano. Para finalizar definitivamente un proceso que se está ejecutando en el background podemos utilizar la orden kill indicando el número de PID o bien el número de trabajo precedido del símbolo %: **#kill %3 (es un kill -15 %3**)



**Demonios**

Un demonio (del inglés ***daemon***) es un *proceso no interactivo en segundo plano que generalmente tenemos cargado en la memoria en espera de alguna señal proveniente de un dispositivo o del mismo núcleo del sistema para despertarse y realizar las acciones y funciones necesarias y ofrecer un determinado servicio***.**

**daemon**es el acrónimo de *disk and execution monitor*.

Más información en **man 7 daemon**

Los demonios no disponen de interfaz con el usuario (aparece una **?** cuando ejecutamos el comando ps -aux o -ef por ejemplo), por tanto, no utilizan las entradas y salidas estándar para comunicar errores o registrar su funcionamiento, **sino archivos de registro (log ), situados habitualmente en el directorio /var/log.**

**SysV (System V)**

Aunque un demonio sea un proceso como cualquier otro, que se ejecuta en segundo plano (background), el modo de gestionarlo e invocarlo es diferente al resto de órdenes y programas del sistema. **Generalmente, los demonios tienen un shell script situado en el directorio /etc/init.d/ que permite iniciarlos, pararlos o ver su estado de ejecución según la sintaxis:**

***#/etc/init.d/script\_de\_demonio acción***

Los parámetros de acción básicos (hay más) que puede aceptar un shell de demonio son:

*start*: para iniciar el demonio; *stop*: para parar el demonio; *restart*: reinicia el demonio y sirve para que se vuelvan a leer sus archivos de configuración y *reload*: aunque no todos los demonios lo permiten, esta acción permite recargar los archivos de configuración sin tener que parar el demonio.

Por ejemplo:

**#/etc/init.d/cupsd start** (en mi instalación se llama **cups**)

**#/etc/init.d/networking restart** (en mi instalación es **network-manager**)

La primera línea pone en marcha el servicio de administración de impresoras CUPS y la segunda reinicia la red leyendo sus archivos de configuración.

Algunas distribuciones disponen de la orden **service** que permite hacer lo mismo sin especificar la ruta completa:

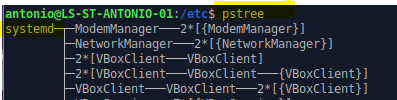
**#service cupsd stop**

En las distribuciones actuales, vemos que todo pasa por **systemd/ systemctl**.

**Systemd**

Desde 2.015, la mayoría de las principales distribuciones Linux adoptaron **systemd** como el sistema de inicio predeterminado, frente al tradicional **SysV.**

**systemd** es un gestor del sistema y de los servicios para Linux. Es el proceso padre (PID igual a 1, antes /usr/sbin/init).



Vamos a ver brevemente **systemd** y **systemctl**, que es la herramienta central para controlar, entre otras cosas, los servicios.

Con el comando systemctl podemos ver todo lo que se puede gestionar con esta herramienta (se llaman **unidades**). Prestaremos especial atención a las que terminan en ***.service***, es decir, a las **unidades de servicio**.

**$systemctl**

Todos los servicios están en **/usr/lib/systemd/system** y los habilitados (es decir, lo que se cargan durante el arranque) en **/etc/systemd/system**

Si queremos ver las unidades que han tenido algún tipo de problema, ejecutamos systemctl con la opción --failed.



Si detectamos alguna unidad que falla, podemos revisar el sistema de registro propio de Systemd que se llama **journal (/var/log/journal)**. Para ello ejecutaremos la siguiente orden:

**$journalctl**

Por supuesto, podemos utilizar journalctl en combinación con grep para localizar aquello que buscamos, pero además nos proporciona algunas herramientas que nos ayudaran en esa tarea.

* journalctl -b nos muestra los mensajes del arranque
* journalctl -u <unidad> nos muestra los mensajes de una determinada unidad

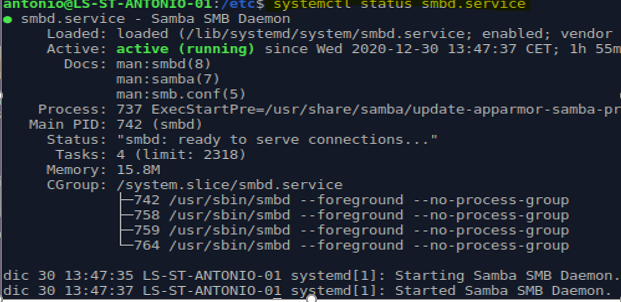
**$journalctl –u smbd.service | tail**

* journalctl \_PID=<proceso> muestra los mensajes de un proceso específico

**Comprobar el estado de los servicios**

$systemctl status unidad

**$systemctl status smbd.service**



Si la unidad es un servicio, no es obligatorio especificarlo con .service ya que es la opción por defecto.

Podemos saber si una determinada unidad está activa ejecutando la orden:

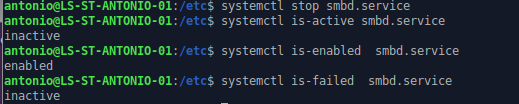
**$systemctl is-active unidad** que nos devolverá active o inactive.

Para saber si la unidad está habilitada (arranca al iniciar sesión), ejecutaremos la orden:

**$systemctl is-enabled** **unidad** que nos devolverá el valor enabled o disabled.

Hay un tercer estado en el que pueden estar las unidades, que es el estado fallo. Para averiguar si una unidad está en fallo, ejecutaremos la orden:

**$systemctl is-failed unidad** que nos devolverá active ó failed; o en el caso de que se halla detenido de forma intencionada nos devolverá el valor unknown o **inactive**.



Gestionando unidades

Con lo que hemos visto hasta el momento, ya sabemos el estado de una unidad, pero, ¿cómo podemos cambiar el estado de una unidad?

Para habilitar una unidad ejecutaremos la orden sudo **systemctl enable unidad.** Una vez habilitada la unidad la podemos iniciar, para lo que ejecutaremos la orden **sudo systemctl start unidad**. Si queremos detener una unidad ejecutaremos la orden **sudo systemctl stop unidad** mientras que si lo que queremos hacer es reiniciar la unidad, ejecutaremos la orden **sudo systemctl restart unidad**. También es posible marcar una determinada unidad como imposible de iniciar tanto de forma manual como de forma automática. Al marcar la unidad de esta manera se apunta a /dev/null. Esta operación se conoce como **enmascarar la unidad** y se hace ejecutando la siguiente orden **sudo systemctl mask unidad.** Si una vez enmascarada una unidad, intentas iniciarla ejecutando la orden **sudo systemctl start** unidad verás un mensaje de error indicándote que la unidad está enmascarada. Para desenmascarar la unidad ejecuta la orden, **sudo systemctl unmask unidad** que devolverá la unidad a su estado anterior.

Para gestionar servicios en **Systemd**, hay que tener en claro los tres estados en los que se puede encontrar un servicio. Estos estados, son los siguientes,

1. ***Activo e inactivo***
2. ***Habilitado e inhabilitado***
3. ***Enmascarado o desenmascarado***

**ACTIVO E INACTIVO**

**Un servicio está activo cuando está en funcionamiento**. Así utilizaremos las siguientes herramientas para conocer su estado y gestionar servicios:

**systemctl is-active servicio**. Nos permite saber si el servicio está activo. Nos devolverá el valor active en caso de estar activo. En otro caso nos devolverá inactive.

Para activar el servicio utilizaremos **sudo systemctl start servicio**. Mientras que para detenerlo utilizaremos **sudo systemctl stop servicio**.

Para reiniciar un servicio utilizaremos **sudo restart servicio**. Esta orden detiene el servicio y lo inicia de nuevo. Si el servicio no está iniciado cuando ejecutamos esta orden, igualmente, lo iniciará.

**sudo systemctl try-restart** servicio tiene el mismo comportamiento que el anterior, pero si el servicio no está iniciado al ejecutar esta orden, no se iniciará el servicio. Es decir, que para que esta orden funcione correctamente, el servicio debe estar iniciado.

Para recargar la configuración de un servicio sin interrumpir su funcionamiento podemos utilizar **sudo systemctl reload servicio**. Sin embargo, esto no está soportado por todos los servicios. Si el servicio contra el que estamos ejecutando esta orden, no soporta lo soporta, simplemente se ignorará.

En ocasiones, interesará recargar la configuración, aunque sea a costa de parar el servicio. Para esto, se han creado dos opciones interesantes como son **sudo systemctl reload-or-restart service** y **sudo systemctl reload-or-try-restart service**, que se comportan como es de esperar.

Un servicio que soporta la recarga de la configuración es Apache. Esto implica que cada vez que cambiemos la configuración, no necesitamos reiniciar Apache, simplemente tenemos que ejecutar **sudo systemctl reload apache2**

**HABILITADO O INHABILITADO**

El siguiente estado en el que podemos encontrar los servicios en systemd es el de habilitado o inhabilitado. **Que un servicio esté habilitado nos indica que se iniciará de forma automática cuando iniciemos el equipo.** Por contra estará inhabilitado, cuando no se inicie de forma automática. Este estado, no es incompatible con el anterior, es decir, **un servicio puede estar activo y deshabilitado (no se inicia al arrancar el sistema, pero lo puedo activar, una vez arrancado el sistema, si me interesa) o inactivo y habilitado (al iniciar el sistema ha arrancado pero después lo he parado)**. Igual que en el caso anterior, para conocer si un servicio está habilitado o no, y para habilitarlo o inhabilitarlo, utilizaremos las siguientes órdenes:

**systemctl is-enabled servicio**, nos permite saber si un servicio está habilitado, en cuyo caso nos devolverá el valor enabled, en caso contrario, nos devolverá el valor disabled.

Para que un servicio se inicie de forma automática con nuestro equipo ejecutaremos la siguiente orden **sudo systemctl enable servicio**.

Esta instrucción crea los enlaces simbólicos de los servicios que queramos habilitar, es decir, de  **/usr/lib/systemd/system/servicio.service**a**/etc/systemd/system** solo si estos enlaces no existen. En el caso de que existan no los reescribe. Si queremos por contra, asegurarnos de que los enlaces simbólicos se reescriben, utilizaremos la instrucción, **sudo systemctl reenable service**. Esta instrucción inhabilita y vuelve a habilitar el servicio, reescribiendo los enlaces simbólicos.

Si no queremos que nuestro servicio arranque con nuestro equipo, ejecutaremos la siguiente orden sudo **systemctl disable service.**

**ENMASCARADO Y DESENMASCARADO**

El último estado que nos queda por definir para gestionar servicios en Systemd es el enmascaramiento. Un servicio está enmascarado cuando no se puede iniciar de forma manual o por medio de otro servicio. Por supuesto, cuando un servicio está enmascarado **no puede ser iniciado en el arranque de la máquina**. Es decir, un servicio enmascarado puede estar activo, pero no puede estar habilitado. Ahora bien, si el servicio no está activo y está enmascarado, no se podrá iniciar

**Vamos a ver un ejemplo de aplicación como proceso y como servicio**

**Aplicación como proceso**

Hemos instalado el ERP Odoo. Para ejecutarlo, de forma manual, hacemos lo siguiente:

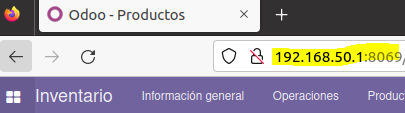
**$/home/odoo/odoo/odoo-bin –c /home/odoo/conf/odoo.conf**

En este caso, ejecutamos **odoo-bin** con un determinado parámetro:

**–c odoo.conf** (muy importante poner las rutas absolutas para garantizar las ejecución correcta del comando y, en este caso, del parámetro que se refiere a un fichero de configuración).

Pero ¿qué sucede cuando apagamos (o reiniciamos) el equipo? Simplemente la aplicación deja de existir.

En este caso, la aplicación la estamos ejecutando como proceso. Puesto que es una aplicación Web (como cups), la podemos ver en funcionamiento ejecutando el navegador y teclear IP:8069.



**Aplicación como servicio**

La ventaja de convertir una aplicación en un servicio es que no tenemos que preocuparnos de iniciarlo de forma manual cada vez que queramos ejecutarlo: el sistema, si hacemos algunas cosas, lo hará por nosotros.

**Proceso**

1.- Creamos el fichero:  **/lib/systemd/system/odoo15.service**

2.- Le añadimos el siguiente contenido.

**[Unit]**

**Description=Odoo Open Source ERP and CRM**

**After=network.target**

**[Service]**

**Type=simple**

**User=odoo**

**Group=odoo**

**ExecStart=/home/odoo/odoo/odoo-bin -c /home/odoo/conf/odoo.**

**[Install]**

**WantedBy=multi-user.target**

Ahora, reiniciamos el servicio

**$systemctl restart odoo15.service**

Le estamos indicando al sistema, entre otras cosas, el comando para arrancar Odoo.

3.- Habilitamos el servicio para que se arranque al inicio:

**$systemctl enable odoo15.service**

**Observación:** podemos deshabilitar el servicio tecleando disable en lugar de enable.

Este servicio pasa a alojarse, por haberse habilitado, a **/etc/systemd/system/multi-user.target.wants**

4.- Ya sabemos cómo parar y arrancar el servicio manualmente:

**$systemctl start odoo15.service   
$systemctl stop odoo15.service**

A continuación, vemos alguna de las claúsulas más importantes del fichero de configuración anterior:

**Description**: podemos introducir aquí el nombre y una descripción de lo que va a hacer nuestro daemon, en este caso podemos poner lo que nosotros queramos, es una descripción de cara al administrador de sistemas, para conocer qué hace el daemon sin necesidad de mirar más código.

**After**: indicaremos si queremos que se cargue después de otros servicios o componentes del sistema. Esto es algo muy importante, porque si nuestro servicio requiere que otro servicio esté levantado, entonces tendremos que seguir una secuencia en concreto. También existe otra directiva llamada «Before» que hace lo contrario, indicando que nuestro servicio se ejecute antes que ese otro servicio que nosotros especificamos.

**User**: el usuario del sistema operativo que ejecutará el daemon. También puede funcionar con Group. Esto es muy importante, porque dependiendo de los permisos que necesite el servicio, tendremos que ejecutarlo con root, nuestro usuario no privilegiado o cualquier otro usuario del sistema.

**ExecStart**: debemos indicar la ruta absoluta (no funcionan relativas) al script o binario que queremos ejecutar para arrancar el servicio en cuestión.

**WantedBy**: indica, entre otras cosas, el objetivo o target (equivalente a runlevel).

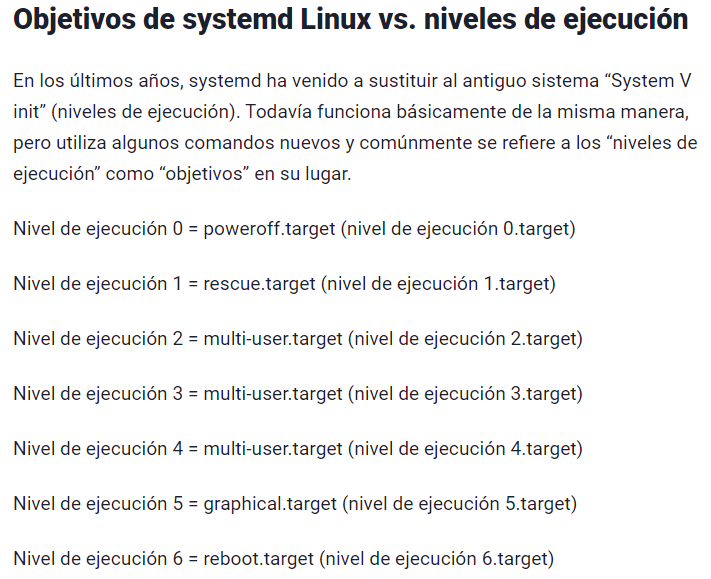
**Niveles de ejecución**

Los demonios que están en ejecución en un momento determinado nos marcan los servicios que un sistema operativo ofrece como servidor y que recibe como cliente. Para organizar adecuadamente la cantidad e interacción de los servicios que necesitamos en un entorno o circunstancia determinados disponemos de niveles de ejecución (**runlevels** en inglés).

Así, pues, *un nivel de ejecución no es más que la agrupación de una serie de demonios en ejecución que tiene como finalidad crear un entorno de servicios ajustado a unas necesidades concretas*.

Generalmente, los sistemas Unix/Linux nos proporcionan diferentes niveles de ejecución, con la funcionalidad indicada en la siguiente tabla

|  |  |
| --- | --- |
| **Nivel** | **Funcionalidad** |
| **0** | El nivel de ejecución 0 está configurado para detener el sistema. |
| **1** | Nivel monousuario. Se ponen en marcha los demonios mínimos para realizar **labores de mantenimiento** y sólo permite la entrada a root. |
| **2 a 5** | **Niveles destinados a ser configurados según las necesidades de cada instalación, aunque habitualmente todos son multiusuario**. En algunas distribuciones el nivel 2 está configurado por defecto como multiusuario sin servicio de red, el nivel 3 como multiusuario con servicio de red, y el nivel 5 para poner en marcha el sistema con el entorno gráfico. |
| **6** | Este nivel está preparado para reiniciar el sistema. |



El nivel de ejecución se puede consultar desde consola con el orden **runlevel**.



N es ningún nivel anterior y 5 es el nivel en el que estoy.

**Obtener el nivel de ejecución de dos formas**



**SysV**

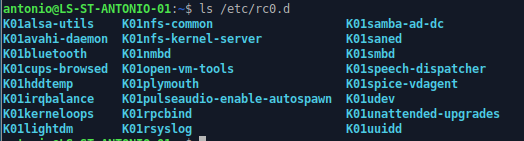
Cada nivel de ejecución tiene un directorio asociado a **/etc/rcX.d** (donde X es el número del nivel). En estos directorios encontramos **enlaces simbólicos** a los guiones de shell que controlan a los demonios y que ya hemos visto que están situados en el directorio **/etc/init.d.**



Además de los subdirectorios de los diferentes niveles, hay otro, **/etc/rcS.d** que **contiene las referencias a los servicios básicos que se ejecutan previamente a cualquier otro nivel.**

Veamos por ejemplo el contenido del directorio asociado al nivel de ejecución 0:

$ls /etc/rc0.d



  La primera letra del nombre de estos enlaces simbólicos lleva información sobre la acción que deben hacer:

**K**: si el enlace empieza por K (kill) indicamos que queremos detener el demonio.

**S**: si el enlace empieza por S (start) indicamos que queremos activar el demonio.

Después de esta letra se pone un número de dos cifras, entre 00 y 99, que indica el orden en la secuencia de inicio y parada de servicios. Este orden es importante, ya que algunos demonios tienen dependencias, es decir, necesitan que otros estén en ejecución antes de ser iniciados. Finalmente encontramos el nombre del demonio en cuestión que nos interesa parar o activar.

La orden para cambiar de nivel de ejecución es init y le pasamos como parámetro el nivel de ejecución que queremos poner en marcha. Así, por reiniciar el sistema:

**#init 6**

El proceso de cambio de nivel es el siguiente: el sistema inspecciona el directorio correspondiente al nivel que queremos habilitar y empezará primero a detener a los demonios correspondientes a las entradas que empiezan por K el parámetro stop y después iniciará los correspondientes a las entradas que empiezan por S mediante el parámetro start.

La modificación de la configuración de un nivel de ejecución se puede hacer manualmente:

1.- Incluir el script de tratamiento del servicio en el directorio /etc/init.d.

2.- Crear los enlaces simbólicos en cada directorio de nivel de ejecución (/etc/rcX.d) dando en el mismo nombre la información de orden secuencial de ejecución y comenzando por K o S si el proceso debe detenerse o marcha respectivamente.

**Sistema de arranque**

La jerarquía del árbol de procesos, su mecanismo de creación y finalización, el concepto de demonio y servicio y la agrupación de servicios en diferentes niveles de ejecución configurables, son parte fundamental de la secuencia final de arranque del sistema operativo Linux y la creación de su entorno de procesos y servicios.

El núcleo del sistema operativo finalmente está cargado ya en la memoria y activa dos procesos previos a la generación de todo el árbol jerárquico de procesos:

El **planificador (scheduler),** proceso con **PID = 0** que gestiona la asignación de procesos a la CPU para su ejecución.

El **proceso de inicio init,** con **PID = 1**. El padre de todos los procesos. Todo el árbol de procesos de Linux es hijo de este proceso.

Para generar el árbol de procesos tenemos dos enfoques: uno **secuencial**, en el que los diferentes servicios se ponen en marcha siguiendo un orden prefijado y configurado previamente en los niveles de ejecución, y un enfoque **basado en eventos**, en el que los servicios inician dependiendo de la ocurrencia de determinados sucesos.

**Sistema secuencial: proceso init**

Tradicionalmente, los sistemas Linux han utilizado un sistema de arranque heredado de Unix System V y que está basado en el proceso init. Este proceso lo inicia el núcleo del sistema y es el encargado de poner en marcha todos los servicios necesarios definidos en el nivel de ejecución configurado por defecto.

Para realizar su tarea, el proceso init utiliza la información contenida en el archivo de configuración /etc/inittab, que contiene, habitualmente:

* El nivel de ejecución por defecto al arrancar.
* Los scripts ejecutar previos al nivel de ejecución por defecto (contenidos en /etc/rcS.d).
* La configuración de los diferentes niveles de ejecución disponibles en el sistema.
* La definición de consolas abiertas en cada nivel de ejecución.
* …

Para iniciar una sesión, el usuario debe identificarse con un nombre y una contraseña mediante el proceso de login. Estas credenciales son verificadas en /etc/ passwd y /etc/shadow o bien utilizando módulos PAM.

**PAM**

Acrónimo de **Pluggable Authentication Modules**. Consiste en un mecanismo de autenticación flexible que permite abstraer las aplicaciones del proceso de identificación.

Una vez identificado el usuario, se leen diferentes archivos de configuración de perfil tanto general como particulares del usuario concreto, entre ellos:

**/etc/profile**

**/etc/bash.bashrc**

**~/.bashrc**

**~/.profile**

Finalmente se carga el shell o shell que se haya configurado en el archivo /etc/ passwd para aquel usuario concreto. Hay toda una serie de alternativas de shells que pueden verse listadas en /etc/shells. Uno de los shells más empleados es el Bash, evolución del original intérprete sh.

La carga del shell Bash da por finalizado el sistema de arranque de Linux y la puesta en marcha del entorno de servicios, y el sistema operativo queda en espera de la introducción de órdenes por parte del usuario.

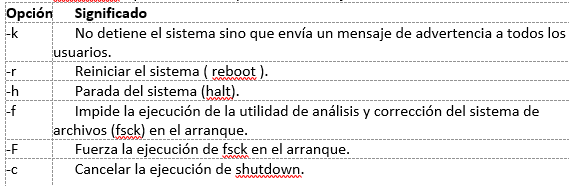
**Parada del sistema**

Ya hemos visto que init gestiona las paradas del sistema con los niveles 0 y 6:

**Nivel de ejecución 0**: Para el sistema deteniendo los diferentes procesos configurados.

**Nivel de ejecución 6**: Detener el sistema y lo reinicia. Sin embargo, disponemos de una orden específica, shutdown, que nos proporciona funcionalidades adicionales.

**shutdown**



Se puede especificar el plazo de cierre del sistema de diferentes maneras:

1. Una hora y un minuto concreto con hh:mm
2. Pasado un número de minutos concreto con +m
3. De manera inmediata con **now** (alias de +0)

En el ejemplo se programa un reinicio del sistema para dentro de 10 minutos con un mensaje de aviso:

#shutdown -r +10 "Reinicio del sistema en 10 minutos"

Esta reanudación se puede cancelar desde otra consola de raíz con la opción -c.

#shutdown -c "Reinicio cancelado"

  Una forma de ver los mensajes es con el comando **dmesg**

Otras órdenes de cierre del sistema que aún se conservan para garantizar la compatibilidad hacia atrás:

halt es equivalente a #shutdown -h now.

reboot es equivalente a #shutdown -r now.

**7.- Monitorización de procesos Unix/Linux**

Para administrar y gestionar procesos en Linux se pueden utilizar directamente órdenes de consola, aunque también se puede trabajar desde el entorno gráfico con la utilidad de monitorización del sistema.

**7.1.- El directorio virtual /proc**

Al arrancar, el núcleo del sistema (kernel) pone en marcha un sistema de archivos virtual /proc que almacena información que recoge del sistema. El directorio

/proc está implementado en la memoria y no se guarda en el disco duro. Los datos que contiene son tanto de naturaleza estática como dinámica, es decir, que varían a lo largo de la ejecución.

**Directorio virtual**

Definimos el directorio /proc como virtual porque no está realmente en el disco duro, sino que el núcleo del sistema operativo lo crea en la memoria RAM.

Una de las características interesantes del directorio /proc es que podemos encontrar las imágenes de los procesos en ejecución con toda la información gestionada por el núcleo del sistema. Cada proceso se puede encontrar en un directorio etiquetado con su identificador de proceso **/proc/PID\_proceso**. Esta información es útil para los programas de depuración o para las mismas órdenes del sistema como ps o top, que la utilizan para ver el estado en el que se encuentran los procesos.

Por otra parte, en /proc podemos encontrar también archivos con información sobre el estado global del sistema como:

/proc/cpuinfo: Información sobre el procesador. Por ejemplo, el tipo, fabricación, modelo y rendimiento.

/proc/devices: Lista de controladores de dispositivo configurados en el núcleo que estaba disponible.

/proc/diskstats: Contiene información estadística de las operaciones de entrada /salida para cada dispositivo de disco.

/proc/dma: Muestra qué canales de DMA (acceso directo a memoria) se están utilizando.

/proc/filesystems: Lista de los sistemas de archivos configurados dentro del núcleo.

/proc/ide: Directorio de información del bus IDE, características de los discos.

/proc/interrupts: Muestra qué líneas de interrupción se están utilizando y, para cada una, un contador de cuantas ha habido.

/proc/ioports: Muestra qué puertos de entrada/salida están utilizando.

/proc/loadavg: La media de la carga del sistema expresada en tres indicadores que representan la media de procesos en marcha en el último minuto, cinco minutos y quince minutos.

/proc/meminfo: Información sobre la utilización de la memoria, tanto la física como la de intercambio.

/proc/net: Directorio con archivos de información de estado sobre protocolos de red. Los utiliza la orden netstat.

/proc/partitions: Contiene los nombres de las particiones del sistema y su tamaño en bloques.

/proc/pci: Dispositivos PCI del sistema.

/proc/stat: Diversas estadísticas sobre el núcleo del sistema, como el tiempo que la CPU ha dedicado a diferentes tareas, el número de interrupciones atendidas o el tiempo transcurrido desde el inicio del sistema.

/proc/version: La versión de núcleo del sistema operativo.

Aunque los archivos que se pueden consultar en /proc suelen ser archivos de texto de lectura fácil con la orden cat o con cualquier procesador de textos, a veces el formato dificulta su interpretación. Es por ello que hay otras órdenes del mismo sistema que cogen esta información y la presentan al usuario de una manera más elaborada que la entienda mejor. Por ejemplo, **free** lee el archivo /proc/meminfo, y **uptime** accede y presenta la información del archivo /proc/loadavg.

**7.2.- Línea de comandos**

El sistema operativo Linux dispone de diferentes órdenes en el entorno de consola de texto para la gestión y monitorización de procesos. Las podríamos clasificar en diferentes apartados funcionales:

Monitorización de procesos: **ps, pstree, time, uptime, vmstat y top**.

Prioridad de los procesos: **nice y renice**.

Órdenes relacionadas con señales: **kill, killall, nohup, disown, trap**.

Ejecución diferida: **at, crontab** .

Tareas en segundo plano: **directiva & y jobs, fg y bg**.

Otras órdenes: **wait, sleep**.

A continuación, detallaremos las órdenes de monitorización de procesos.

**Orden ps**

En la mayoría de sistemas operativos de la familia Unix, la orden **ps** (*process status*) permite visualizar los procesos en ejecución. Esta orden está basada en la información contenida en el directorio virtual /proc y admite un gran número de opciones que incluso varían en función de los diferentes formatos de sintaxis:

**Estilo Unix**: las opciones son letras mayúsculas o minúsculas precedidas de un guión. Por ejemplo: ps -A.

**Estilo BSD**: las opciones no llevan guión. Por ejemplo: ps r.

**Estilo GNU**: las opciones utilizan nombres largos y llevan doble guion. Por ejemplo: ps --user.

Las opciones completas de ps se encuentran en las páginas del manual.

#man ps

Descripción de los campos más habituales del orden ps

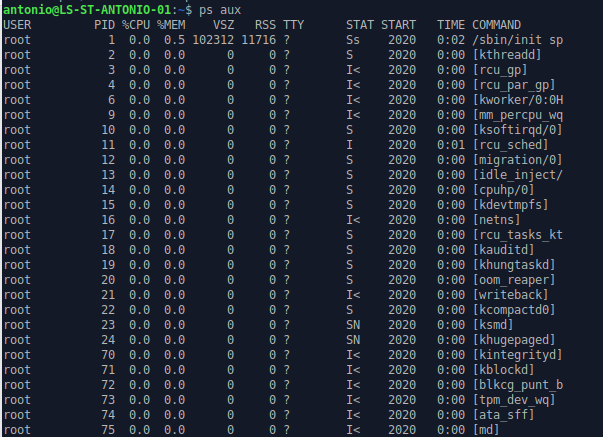
|  |  |
| --- | --- |
| **Campo** | **Significado** |
| **PID** | Identificador del proceso |
| **PPID** | Identificador del padre del proceso |
| **UID** | Identificador del usuario propietario del proceso |
| **USER** | Nombre del usuario propietario del proceso |
| **TTY** | Terminal asociada al proceso. Sin terminal asociado aparece un ? |
| **TIME** | Tiempo de CPU acumulado por el proceso |
| **CMD** | El nombre de la orden que inició el proceso |
| **SIZE** | Tamaño virtual de la imagen del proceso |
| **NI** | Prioridad nice asignada por el usuario |
| **% CPU** | Porcentaje de CPU usado por el proceso |
| **% MEM** | Porcentaje de memoria usado por el proceso en relación a la memoria física |
| **START** | Hora de inicio del proceso |
| **VSZ** | Memoria virtual en kilobytes |
| **RSS** | Memoria residente en kilobytes que utiliza el proceso |
| **STAT** | Estado del proceso como por ejemplo:  ***\* R (running): en ejecución  \* S (sleeping): proceso bloqueado en espera de un suceso  \* T (stopped): proceso detenido pero que se puede reiniciar  \* Z (zombie): proceso finalizado pero que permanece en la memoria*** |

Si no ponemos ningún argumento, sólo se mostrarán aquellos procesos iniciados por el usuario actual y que hacen referencia al terminal que éste está utilizando.

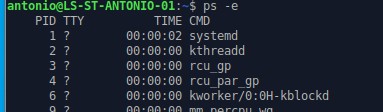


La primera columna muestra el identificador del proceso, la terminal asociada al proceso que en este caso se trata del primer ***pseudotermina***l (pts/0), dado que hemos utilizado un terminal del entorno gráfico, el tiempo acumulado de uso de CPU y finalmente la orden que ha generado el proceso.

Añadiendo algunas opciones (**aux**) en formato BSD podemos obtener más información.

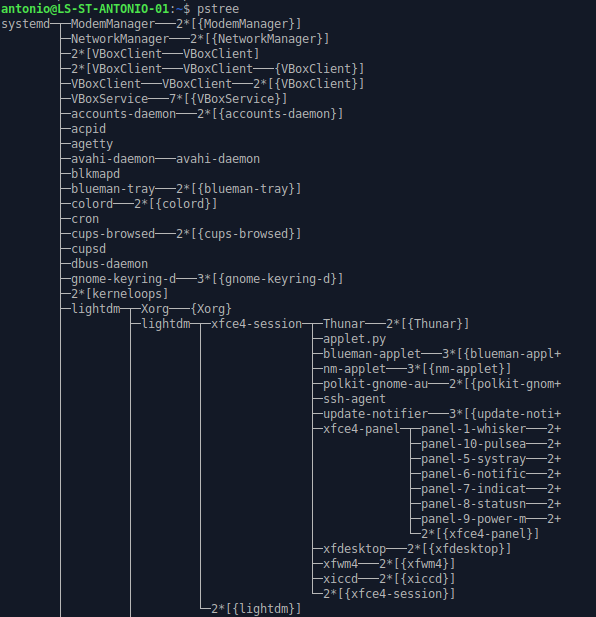


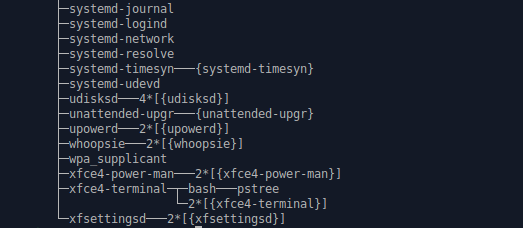
Con ps –e obtenemos PID y proceso asociado (que es en muchas ocasiones lo que nos hace falta).



**Orden pstree**

La orden pstree muestra una lista de procesos en forma de árbol que sigue la jerarquía de procesos Unix y que permite identificar fácilmente cuál es el proceso padre de otro.





Podemos ver, por ejemplo, que systemd (init) es el padre de todos los procesos, que hay algunos demonios conocidos en marcha (cron, cupsd, etc.) y que en el terminal gráfico (xfce4-terminal) encontramos el proceso bash / pstree que genera el árbol.

**Orden time**

Esta orden permite ejecutar cualquier otra aplicación y registrar los recursos que ha utilizado. Por defecto sólo presenta el tiempo real (estimado con el reloj del sistema) y el tiempo de CPU empleado tanto en modo usuario como en modo sistema. Sin embargo, esta misma orden ejecutada con privilegios de super administrador y utilizando diferentes modificadores puede suministrar más parámetros de información.

**$time ls –R /**



**Orden sleep**

Suspende la ejecución durante los segundos especificados como parámetro.

#sleep 20; echo "Han pasado 10 segundos"

Han pasado 10 segundos

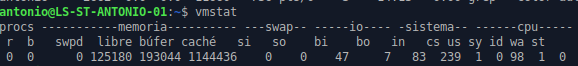
**Orden uptime**

La orden uptime da la hora del sistema, el tiempo que lleva encendido, la cantidad de usuarios conectados y la carga media del sistema durante el último minuto, los últimos cinco minutos y los últimos quince minutos.



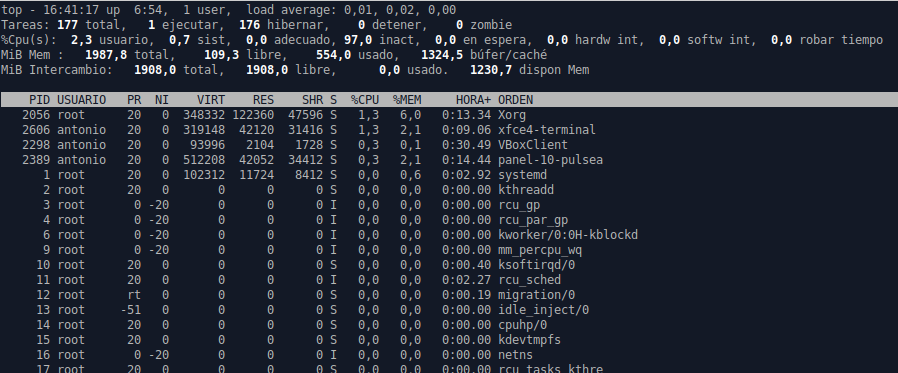
**Orden vmstat**

La orden *vmstat* da información del estado de la memoria física, de la memoria virtual, del intercambio entre memoria interna y disco (swapping), de las transferencias de disco, las interrupciones y uso del procesador. Admite el uso de modificadores y permite monitorización continuado.



**Orden top**

La orden top presenta la información de los procesos de manera dinámica e interactiva, en tiempo real, con una actualización por defecto cada tres segundos, y ordenados por el porcentaje de uso de CPU.

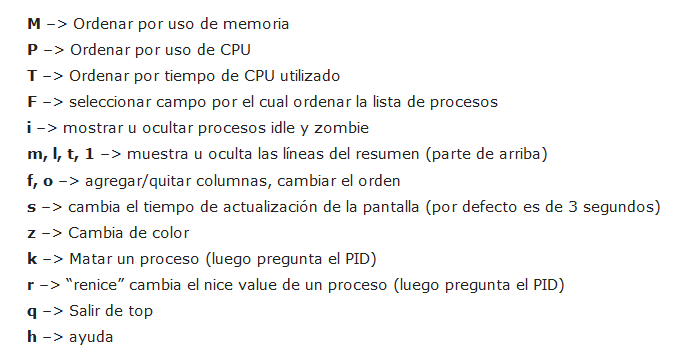


Como se ve en el listado, la primera línea de información es la misma que da la orden **uptime**. A continuación, muestra información de los procesos en ejecución con una serie de campos de información, similares a los obtenidos con la orden ps pero con pequeñas variaciones.

|  |  |
| --- | --- |
| **Campo** | **Significado** |
| PID | Identificador del proceso |
| USER | Nombre del usuario propietario del proceso |
| **PR** | **Prioridad del proceso** |
| **NI** | **Prioridad nice asignada por el usuario** |
| VIRT | Memoria virtual en kilobytes |
| RES | Memoria residente en kilobytes que utiliza el proceso |
| SHR | Memoria compartida |
| S | Estado del proceso |
| %CPU | Porcentaje de CPU usado por el proceso |
| %MEM | Porcentaje de memoria usado en relación a la memoria física |
| HORA | Hora de inicio del proceso |
| ORDEN | El nombre de la orden que inició el proceso |

La orden top no sólo muestra la información actualizando los datos dinámicamente, también puede interactuar con el usuario, que puede actualizar la información, ordenar los procesos por cualquier campo de información o, lo que es más importante, finalizar el proceso enviando una señal kill o incluso cambiarle la prioridad.

La lista de las principales órdenes de top se muestra a continuación. Debemos ejecutar top y entonces, interactuar con estas opciones.



**Herramientas sysstat**

El paquete sysstat es una colección de herramientas de monitorización de rendimiento para sistemas Linux. Proporciona datos instantáneos de rendimiento, que se pueden almacenar en archivos históricos. En entornos de servidor, estos datos proporcionan información valiosa para detectar carencias y cuellos de botella del sistema.

La orden de instalación es la siguiente:

**#apt-get install sysstat**

Algunas de las herramientas que incluye este paquete son:

**mpstat:** recoge información del rendimiento de cada uno de los procesadores del sistema. Permite hacer un monitoreo continuo y admite modificadores.

**iostat:** es una herramienta más completa que la anterior, ya que además de presentar estadísticas de la CPU, da información de los dispositivos de entrada y salida, las particiones y los sistemas de archivos.

**pidstat**: información estadística de los procesos de Linux.

**Entorno gráfico (gestor de tareas)**

Hay muchas herramientas gráficas para monitorizar procesos. En esta distribución encontramos Gestor de tareas y como la mayoría, permiten la visualización en tiempo real de los recursos, de los procesos que se están ejecutando y el consumo de recursos del sistema que hacen.

