



**2º de Grado en Ingeniería Informática**  
**Escuela Politécnica Superior**  
**Universidad de Córdoba**  
**Departamento de Informática y Análisis Numérico**



**Sistemas operativos, apuntes de la asignatura.**

Profesor: Juan Carlos Fernández Caballero

email: [jfcaballero@uco.es](mailto:jfcaballero@uco.es)

## **TEMA 2 – PROCESOS e HILOS**

# 1. Índice de contenidos

1	Procesos y bloque de control de procesos (BCP).....	3
2	Dispatcher.....	5
2.1	Ejemplo de ejecución del dispatcher.....	5
3	Control de llamadas a funciones mediante pila.....	8
4	Arranque del sistema.....	12
5	Creación y terminación de procesos.....	13
6	Un modelo de estados para procesos.....	16
6.1	Transición entre estados.....	17
6.2	Un posible esquema de colas en relación a los estados.....	19
6.3	Procesos suspendidos.....	20
7	Hilos (hebras).....	21
7.1	Estados de los hilos.....	23
7.2	Motivación y ventajas de las hebras.....	24
8	Servicios en GNU/Linux.....	25
8.1	Registro de la actividad de los demonios.....	26
8.2	Arranque de servicios con Upstart (basado en SysVinit).....	27
8.2.1	Gestión de un servicio, comando <i>service</i> :.....	31
8.3	Arranque de servicios con el sistema systemd.....	32
8.3.1	Gestión de un servicio, comando <i>systemctl</i> (sustituto de <i>service</i> ).....	33

## 1 Procesos y bloque de control de procesos (BCP)

Un **programa o tarea** es una unidad inactiva, como un archivo almacenado en un disco. Si hablásemos en términos de objetos, un programa sería una clase y un proceso sería una instancia de esa clase (objeto). Por tanto, **un programa no es un proceso**.

Se han dado muchas **definiciones del término proceso**, incluyendo:

- Una unidad de actividad **cargada en memoria principal** y caracterizada por un solo **hilo secuencial de ejecución**, un **estado actual**, y un **conjunto de datos y recursos del sistema asociados** al proceso que lo caracterizan (**contexto de un proceso**).

El **contexto de un proceso** es por tanto un **conjunto de datos** por el cual el sistema operativo es capaz de **supervisar y controlarlo**, y se almacena de manera diferente dependiendo del sistema operativo, pero de forma general se hace en una estructura de datos que se llama **Bloque de Control de Proceso** o **BCP** (*process control block* - *PCB* en Inglés) (Figura 3.1). Cada proceso tiene asociado un BCP que está almacenado en una **lista simplemente enlazada**, llamada **tabla de procesos**.

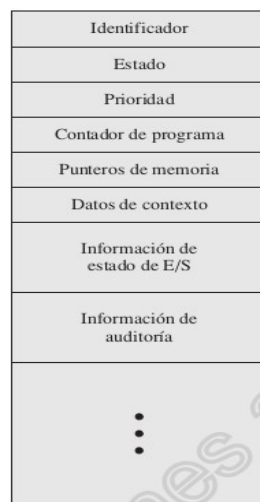
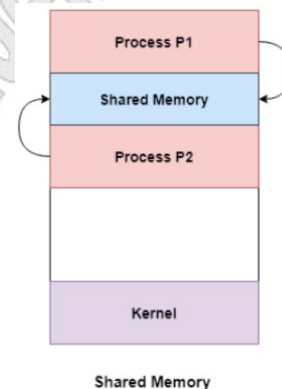


Figura 3.1. Bloque de control de programa (BCP) simplificado.

Entre ese conjunto de datos que se almacenan en el BCP, algunos destacables son:

- **Identificación.** Un identificador único asociado al proceso para distinguirlo del resto de procesos. Puede también almacenar el identificador del proceso padre creador de este proceso y el identificador del usuario del proceso, dependiendo del sistema.
- **Estado.** Si el proceso está actualmente corriendo, está en el estado Ejecución. Hay más tipos de estado, dependiendo del sistema operativo, como por ejemplo estado Listo, Bloqueado, Suspendido, Terminado o Zombie (se estudiarán más adelante).
- **Información de planificación:** Nivel de prioridad relativo al resto de procesos. Esta información se usa por parte del planificador para decidir qué trabajo ejecutar a continuación.

- **Descripción de los segmentos de memoria asignados al proceso:** Espacio de direcciones o límites de memoria asignado al proceso en el espacio de usuario en RAM.
- **Punteros a memoria.** Incluye los punteros al código de programa y los datos asociados a dicho proceso, e incluso si el proceso utiliza memoria virtual. Incluye los límites del último registro de activación de la pila.
- **Datos de contexto en relación a los registros de la CPU.** Estos son datos que están presentes en los registros del procesador cuando el proceso está corriendo (PC, IR, RDIM, etc), banderas de estado, señales, datos temporales en registros, es decir, todo lo necesario para poder continuar la ejecución del proceso cuando el planificador del sistema operativo lo decida.
- **Información de estado de E/S y recursos asignados:** Incluye las peticiones de E/S pendientes, los dispositivos de E/S asignados a dicho proceso (por ejemplo, un disco duro), una lista de los ficheros usados, etc.
- **Comunicación entre procesos.** Se guarda cualquier dato que tenga que ver con la comunicación entre procesos independientes, es decir, información sobre IPC (*Inter-Process Communication*):
  - Memoria compartida.
  - Señales.
  - Semáforos.
  - Paso de mensajes.
    - Tuberías (*pipelines*).
    - Colas (*queues*).
  - Paso de mensajes en red.
    - *Sockets*.
    - Estándar MPI (*Message Passing Interface*). MPI es también una implementación del estándar que proporciona un conjunto de librerías (*Middleware*) para comunicar procesos entre máquinas en red.
- **Información de auditoría.** Puede incluir la cantidad de tiempo de procesador o ciclos utilizados por el proceso, cuánto le queda por ejecutar.



El punto más significativo en relación al BCP, es que contiene suficiente información de forma que **es posible interrumpir un proceso cuando está corriendo y posteriormente restaurar su estado de ejecución** como si no hubiera tenido interrupción alguna. Por tanto, el BCP es la herramienta clave que permite al sistema operativo dar soporte a múltiples procesos y **proporcionar multiprogramación**.

Un **BCP** se crea cuando se acepta un proceso en el sistema por parte de **planificador a largo plazo**, y se actualiza a medida que la ejecución del proceso avanza desde el inicio hasta el final.

Se llama **imagen del proceso** al conjunto del código, datos, pila, montículo (se estudia a continuación) y BCP de un proceso. Para ejecutar un proceso es necesario que su imagen esté cargada en memoria principal.

## 2 Dispatcher

Ampliando con la sección anterior, para un computador monoprocesador (y mononúcleo), como máximo un único proceso puede estar ejecutándose en un instante de tiempo, y dicho proceso estará en el estado “Ejecución”. La conmutación entre unos procesos y otros es lo que da lugar a la multiprogramación, y el proceso que se encarga de conmutar según la política de planificación del sistema, se llama **activador** o **dispatcher**, el cual reside en el núcleo del sistema.

El *dispatcher* pasa a ejecución cuando se producen:

- **Interrupciones:**
  - Por temporización (**rodaja o quantum de tiempo**).
  - De cualquier otro tipo, por ejemplo ante la llegada de un proceso prioritario o interrupciones software.
- Solicitud de **operaciones bloqueantes**. Por ejemplo, un proceso en estado ejecutando invoca una operación *wait()*.
- Solicitud de **operaciones de E/S**. Dado que un proceso no puede continuar hasta que termine dicha operación, es un buen momento para ejecutar otro proceso.

Es el planificador a corto plazo el que dice al *dispatcher* qué proceso es el que debe introducir en la CPU. Dependiendo de la literatura que consulte encontrará que se habla indistintamente de **planificador a corto plazo** o **dispatcher**, o que se habla de planificador a corto plazo (**low level scheduler**) y por otro lado el **dispatcher**. Aquí se tratará como la segunda opción, como un proceso aparte utilizado por el planificador a corto plazo. Es decir, el planificador a corto plazo se encargará de reordenar la lista de procesos listos para su ejecución y el dispatcher recogerá al primero según ese ordenamiento que ha hecho el planificador a corto plazo.

Gracias al **BCP**, el *dispatcher* puede introducir un proceso nuevo o hacer que se continúe con otro por la instrucción donde se quedó (salvado y restauración de contexto).

### 2.1 Ejemplo de ejecución del dispatcher

Suponga 3 procesos, los 3 residen en memoria principal y representan a 3 programas. De manera adicional, existe un pequeño programa activador que forma parte del núcleo del sistema y que intercambia procesos en el procesador.

La siguiente figura muestra las trazas de cada uno de los 3 procesos en los primeros instantes de ejecución. Tenga en cuenta los siguientes supuestos:

- Se muestran las 12 primeras instrucciones ejecutadas por los procesos A y C, y las 4 instrucciones del proceso B.
- Se supone que el activador es un pequeño programa con 6 instrucciones.

- En este ejemplo, se asume que el sistema operativo sólo deja que un proceso continúe durante **6 ciclos de instrucción (rodaja de tiempo)**, después de los cuales se interrumpe por alarma de reloj (temporizador). Esto previene que un solo proceso monopolice el uso del tiempo del procesador y provoque inanición en los demás.
- Por simplicidad y a modo didáctico **se omite**:
  - **1) la ejecución de la rutina ISR** que trata la alarma de reloj y
  - **2) la ejecución del proceso del planificador a corto plazo.**
- Además se supone que el planificador decide comenzar a ejecutar el proceso **A**.

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011
(a) Trazas del Proceso A	(b) Trazas del Proceso B	(c) Trazas del Proceso C

5000 = Dirección de comienzo del programa del Proceso A.

8000 = Dirección de comienzo del programa del Proceso B.

12000 = Dirección de comienzo del programa del Proceso C.

Figura 3.3. Trazas de los procesos de la Figura 3.2.

Dicho esto, ahora vea estas **trazas desde el punto de vista del procesador**. La Figura 3.4 muestra las trazas entrelazadas resultante de los 52 primeros ciclos de ejecución (por conveniencia los ciclos de instrucciones han sido numerados).

1	5000		27	12004	
2	5001		28	12005	
3	5002				
4	5003		29	100	Temporización
5	5004		30	101	
6	5005		31	102	
			32	103	
7	100	Temporización	33	104	
8	101		34	105	
9	102		35	5006	
10	103		36	5007	
11	104		37	5008	
12	105		38	5009	
13	8000		39	5010	
14	8001		40	5011	
15	8002				
16	8003				
			41	100	Temporización
17	100	Petición de E/S	42	101	
18	101		43	102	
19	102		44	103	
20	103		45	104	
21	104		46	105	
22	105		47	12006	
23	12000		48	12007	
24	12001		49	12008	
25	12002		50	12009	
26	12003		51	12010	
			52	12011	Temporización

100 = Dirección de comienzo del programa activador.

Las zonas sombreadas indican la ejecución del proceso de activación;

la primera y la tercera columna cuentan ciclos de instrucciones;

la segunda y la cuarta columna las direcciones de las instrucciones que se ejecutan

Figura 3.4. Trazas combinadas de los procesos de la Figura 3.2.

A → Dispatcher → B → Dispatcher → C → Dispatcher → A → Dispatcher → C

La siguiente figura muestra un ejemplo del despliegue en memoria de los 3 procesos anteriores en el ciclo de instrucción 13 (en ese ciclo está ejecutando el proceso B).

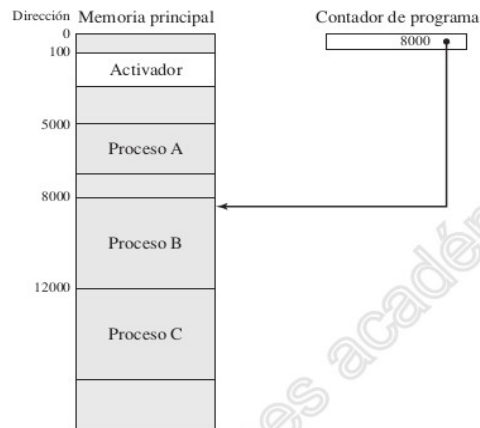


Figura 3.2. Instantánea de un ejemplo de ejecución (Figura 3.4) en el ciclo de instrucción 13.

Se resume la ejecución de los tres procesos junto con el activador de la siguiente manera:

- 1) Las primeras 6 instrucciones del proceso A se ejecutan seguidas de una alarma de temporización y de la ejecución del activador, que ejecuta sus 6 instrucciones antes de devolver el control al proceso B<sup>1</sup>.
- 2) Después de que se ejecuten 4 instrucciones de B, el procesador deja de ejecutar el proceso B y se ejecuta el activador.
- 3) Se pasa a ejecutar el proceso C, y posteriormente tras temporización se vuelve a ejecutar el activador.
- 4) El procesador vuelve al proceso A y cuando llega a su temporización se vuelve a ejecutar el activador.
- 5) Se pasa de nuevo al proceso C.

<sup>1</sup> El reducido número de instrucciones ejecutadas por los procesos y por el activador es irreal; se ha utilizado para simplificar el ejemplo y clarificar las explicaciones.

### 3 Control de llamadas a funciones mediante pila

Una técnica habitual para controlar la ejecución de llamadas a funciones (procedimientos, subrutinas) y los retornos de las mismas en un proceso es utilizar su pila.

Una pila (*stack*) es un **conjunto ordenado de elementos** llamados **registros de activación** (a este tipo de registros también se les llaman marcos de pila), que almacenan determinada **información sobre las rutinas que se invocan**, y que en cada momento solamente se puede acceder a uno de ellos, el más recientemente añadido. El número de elementos de la pila, o longitud de la pila, es variable (investigue en la web cuál es el tamaño máximo de pila en un sistema GNU/Linux) . Por esta razón, se conoce también a una **pila** como una **lista donde el último que entra es el primero que sale** (*Last-In First-Out*, LIFO).

Por cada llamada que se realice desde el proceso en ejecución a una función o subrutina, **en la pila se almacenará lo siguiente**. Tenga en cuenta que esto se hace **a nivel de instrucciones en ensamblador** al compilar, de forma que es transparente al programador:

- **La dirección de retorno**, reservada (colocada) y asignada por la función llamante, siendo la dirección de memoria de la siguiente instrucción de código donde retornar después de ejecutarse la función llamada.
- Los datos pasados como **parámetros** a la función o subrutina, colocados y asignados por la función llamante.
- El **valor de retorno (devuelto)** de la función llamada. No confundir con la dirección de retorno, es el valor devuelto por la función o subrutina (entero, flotante, etc). La función llamante reserva el espacio para ese valor, y lo recogerá cuando termine la función llamada, que es quien le asigna el valor correspondiente.

Los tres datos comentados hasta ahora se colocan antes de invocar a la función llamada, y son instrucciones a nivel de ensamblador que el programador a nivel de API no ve.

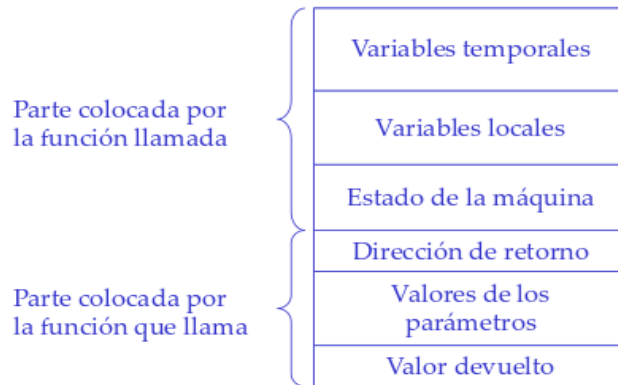
- El **estado de la máquina** (registros SP y FP del procesador), colocados y asignados por la función llamada y de los cuales se habla después (delimitan el último registro de activación).

En ensamblador son una serie de instrucciones que se ejecutan al inicio de la función llamada, y que se suele llamar **“prologo”**.

- Las **variables locales**, colocadas y asignadas por la función llamada. Por ejemplo si se declara un entero, un array de enteros, un flotante, etc.
- Las **variables temporales** colocadas y asignadas por la función llamada. Son variables que almacenan información pero que no son necesariamente variables locales de la función llamada, lo contrario a lo que ocurre con una variable local.

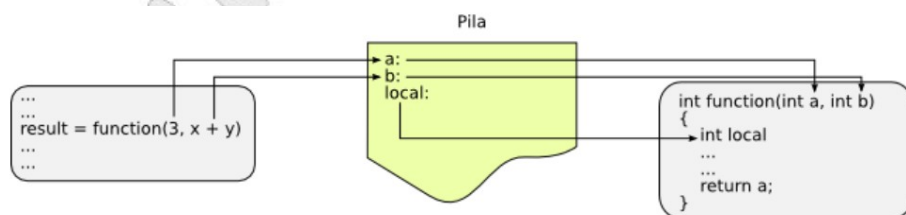


En la siguiente imagen se muestra un **registro de activación**, que contiene todo lo anteriormente expuesto. Para cada función llamada **se apila un registro de activación**.

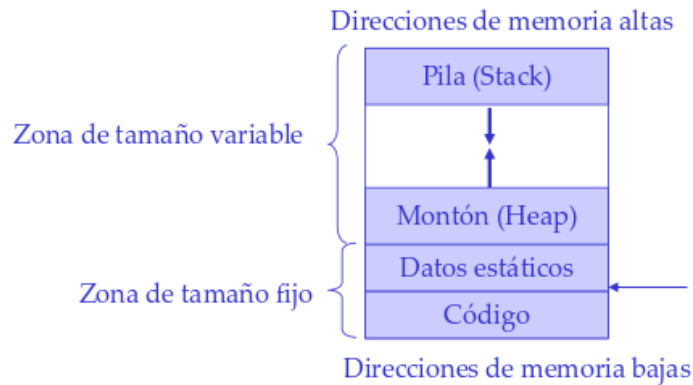


Ejemplo de variables temporales:

<pre>main() {   ...   //No habrá 2 variables locales en la función llamada   a = funcion (b, x+y);   ... }</pre>	<pre>int funcion (int a, int b) {   int local = a+b;   ...   //No habrá 4 variables locales en la función llamada   printf(“%d\n”, local + 20 + b + 50);   return local; }</pre>
--	--



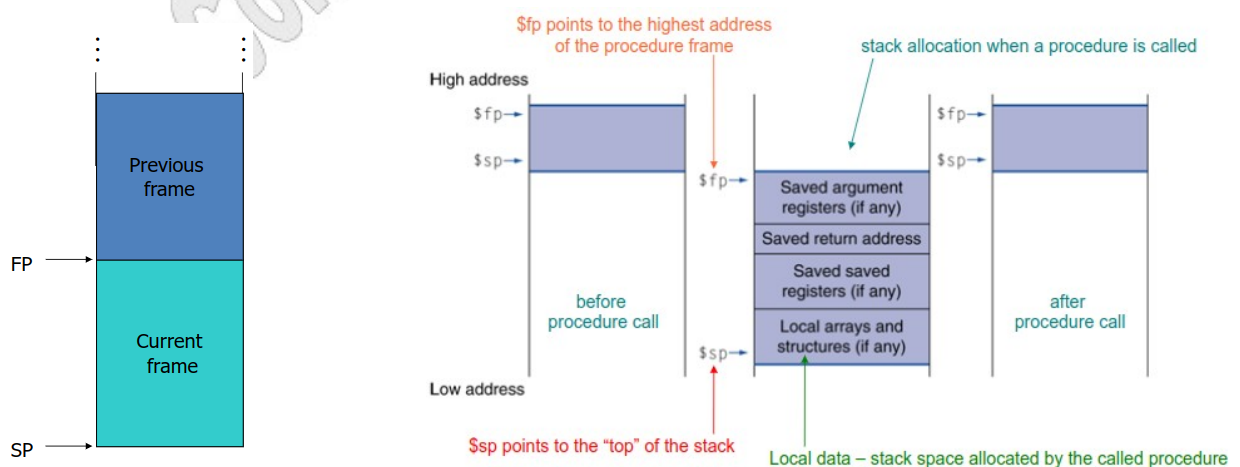
Dicho esto, un proceso se compone de varias zonas de memoria principal diferenciadas (no tienen porque tener ese orden). Una vez que el sistema operativo ha reservado las zonas de memoria en el espacio de usuario, este está listo para ser ejecutado (se dispone de su imagen):



- **La pila de llamadas (*stack*).** Zona de memoria variable para las llamadas a subrutinas, que almacenará los registros de activación de llamadas a funciones o subrutinas.

El procesador maneja dos registros especiales referentes a la pila (*stack*), que sirven para que en una llamada a función se pueda acceder a los parámetros y demás datos que necesiten la función llamante y la función llamada, de forma que se tenga bien localizado su registro de activación. Por tanto, estos dos registros delimitan el espacio de memoria del último registro de activación apilado:

- **Puntero de pila o *Stack Pointer* (SP):** Dirección de la cima de la pila, es decir, de la parte superior del último registro de activación apilado. SP se va actualizando a medida que se van poniendo datos en el registro de activación de la pila. Según el sistema, la pila puede crecer hacia direcciones bajas de memoria o hacia direcciones altas.
- **Puntero de base o *Frame Pointer* (FP):** Dirección base del último registro de activación apilado.



- **El área de datos dinámicos o montón (*heap*).** Una zona para la gestión dinámica de la memoria que se solicita durante la ejecución, por tanto es una zona de memoria variable. Por ejemplo en C con la orden “*malloc()*”, “*calloc()*”, “*free()*”, “*realloc()*”, o en los lenguajes orientados a objetos con el operador “*new()*”.
- **El área de datos estáticos:** Se usa para almacenar las variables globales, las constantes (*#define* y *const int* en C) y variables estáticas (*static* en C), reservando el espacio justo ya que se conoce en tiempo de compilación. Como ejercicio de investigación, busque en la web la diferencia y uso de variables *const* y *static*.
- **El área del código.** Se usa para almacenar el código fuente en instrucciones máquina, por consiguiente se reserva solamente el espacio justo.

Véase a continuación el conjunto de pasos en una llamada a una rutina y su retorno al ámbito de la rutina que llama.

**Pasos en la llamada a una rutina:**

- Se reserva espacio para el valor devuelto, pero no se asigna valor alguno todavía.
- Se reserva espacio y se almacena el valor de los parámetros que se pasan a la función llamada.
- Se reserva espacio y se almacena el valor de la dirección de retorno donde comenzar a ejecutar cuando finalice la función llamada.
- Se reserva espacio y se almacena el estado de la máquina, SP y FP (delimitan el último registro de activación).
- Se reserva espacio y se almacenan las variables locales y temporales de la función llamada. Respecto a las variables locales, solo se almacenan las que recogen los parámetros de entrada, el resto se irán reservando y almacenando cuando se vaya ejecutando la función llamada.
- Ejecución del resto del código de la función llamada (recuerde el “**prologo**”).

**Pasos al final de la ejecución de la rutina llamada, retorno:**

- Se copia el valor devuelto en el espacio que había para ello en la pila. Lo podrá usar posteriormente la función llamante para asignarlo a alguna variable local o temporal suya.
- Se recoge el valor de la dirección de retorno de la función que llama.
- Se restaura el contenido del estado de la máquina. Esto modifica el valor de FP y SP a los valores de la rutina llamante, y provoca la liberación de la memoria ocupada en la pila por el registro de activación creado al hacer la llamada a la rutina.
- Se devuelve el control a la función que llama (instrucciones de tipo *jump* o *ret* a nivel de ensamblador).

Estos dos últimos pasos constituyen una serie de instrucciones a nivel de ensamblador que se suelen denominar “**epílogo**”.

## 4 Arranque del sistema

Para que una computadora comience a funcionar, por ejemplo cuando se enciende o se reinicia, es necesario que produzcan una serie de etapas:

- 1) Se ejecuta un programa de inicio que se almacena en una memoria ROM (*read only memory*, memoria de sólo lectura) o en una memoria EEPROM (*electrically erasable programmable read only memory*, memoria de sólo lectura programable y eléctricamente borrrable), y se conoce con el término general **firmware** o **Bios (Basic Input and Output System)**. Si se usa una ROM, para cambiar el sistema de arranque hay que cambiar el *chip* en si, si embargo si la computadora posee una EEPROM puede actualizarse el *firmware*.

El propósito del *Bios* es identificar y diagnosticar los dispositivos del sistema, como la CPU, la RAM, las controladoras, la tarjeta de vídeo, el teclado, la unidad de disco duro, la unidad de disco óptico y otro hardware básico, además de detectar el disco desde el cual se tiene que leer el cargador de segunda etapa o cargador de arranque.

El *Bios* lo ejecuta la CPU, que sabe su dirección de inicio almacenada en la ROM.

Actualmente también se usa Bios con el termino **UEFI (Unified Extensible Firmware Interface)**. De manera genérica, la UEFI es una Bios más rápida y con un sistema de arranque más seguro que evita que se inicien sistemas operativos no autorizados y software no deseado.

- 2) Si se pasan las pruebas de diagnóstico satisfactoriamente, el programa puede continuar con la secuencia de arranque. En esta segunda etapa el *Bios* hace que se comience a ejecutar un programa llamado **gestor o cargador de arranque de segunda etapa**.

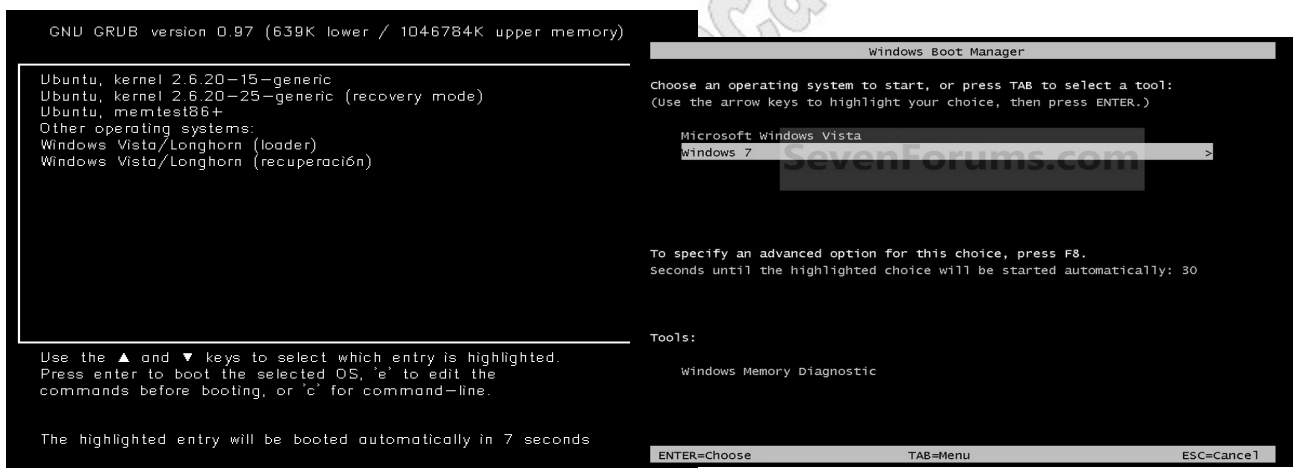
Los **cargadores de arranque suelen encontrarse** en el **primer sector (bloque 0)** del dispositivo de arranque (disco duro – puede ser en una partición -, CD-ROM, Pendrive). A este sector se le suele llama *Master Boot Record* (MBR). Un disco que tiene una partición marcada como de arranque en su sector cero, se denomina **disco de arranque o disco del sistema**.

Para que comience a ejecutar el gestor de arranque, la CPU busca en ese sector y lo carga en memoria, de forma que se comienza a ejecutar su código.

- 3) Cuando el gestor de arranque empieza a ejecutarse comienza a **explorar el sistema de archivos para localizar el kernel del sistema operativo y cargarlo en memoria principal**. Por tanto comienza a ejecutarse el núcleo del sistema. En esa ejecución se pueden realizar tareas como pueden ser:
  - Configuración de la memoria principal y paginación, además de la memoria virtual o Swapper.
  - Configuración de los sistemas de E/S.
  - Establecimiento del manejo de interrupciones.
  - Montar el sistema de archivos del root en /
  - etc.

- 4) Una vez ejecutado el *Kernel*, el sistema operativo comienza a establecer el **espacio de usuario**. En el caso de GNU/LINUX se ejecuta el proceso *init()*. Este a su vez ejecuta una serie de servicios y scripts, como por ejemplo:
- Montaje de otras particiones o discos.
  - Inicialización del servicio de red.
  - Inicio del entorno gráfico.
  - Inicio de sesiones de otros usuarios y control de logeado.
  - Inicialización de bluetooth.
  - Inicialización del servicio de impresión.
  - Etc.

Como ejemplos de **cargadores de arranque** tenemos *Lilo* y *Grub* en sistemas GNU/LINUX, o *NT Loader* y *Windows Boot Manager* en sistemas Windows. En las siguientes figuras se muestran arranques duales gestionados por *Grub* y *Windows Boot Manager*. Si su sistema no posee más de un sistema operativo, lo normal es que estos cargadores ni siquiera aparezcan de manera visual en el arranque, aunque si que se están ejecutando de manera transparente para usted.



## 5 Creación y terminación de procesos

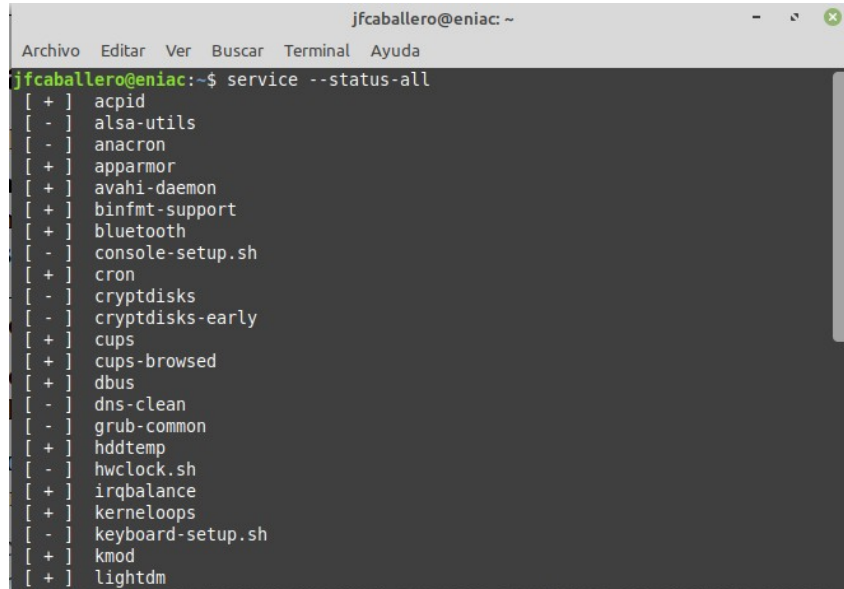
La creación de procesos se puede dar de varias maneras:

- 1) **El arranque del sistema:** Demonios y servicios como correo electrónico, servidor de páginas web, gestor de impresión, gestor de bluetooth, gestor del entorno gráfico, gestor de red, gestor de sonido, etc.

Los demonios se ejecutan en el espacio del usuario, pero eso no significa que no tengan que acceder al núcleo para realizar las tareas que necesite su servicio.

Cuando se carga el kernel en el arranque, también se crean los procesos necesarios en el espacio del kernel para gestionar el sistema.

En Gnu/Linux podemos utilizar el comando `service --status-all` para visualizar los **servicios que se están ejecutando en el sistema**.



```
jfcaballero@eniac: ~
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
jfcaballero@eniac:~$ service --status-all
[ + ] acpid
[ - ] alsa-utils
[ - ] anacron
[ + ] apparmor
[ + ] avahi-daemon
[ + ] binfmt-support
[ + ] bluetooth
[ - ] console-setup.sh
[ + ] cron
[ - ] cryptdisks
[ - ] cryptdisks-early
[ + ] cups
[ + ] cups-browsed
[ + ] dbus
[ - ] dns-clean
[ - ] grub-common
[ + ] hddtemp
[ - ] hwclock.sh
[ + ] irqbalance
[ + ] kerneloops
[ - ] keyboard-setup.sh
[ + ] kmod
[ + ] lightdm
```

- 2) La **invocación interna desde un proceso de algún ejecutable**.
- 3) La **invocación de una llamada al sistema para creación de procesos**, por ejemplo *fork()* mediante llamadas Posix.
- 4) La **creación de procesos** también se puede producir:
  - En sistemas interactivos mediante el uso del ratón haciendo por ejemplo un **doble clic para abrir un programa**,
  - mediante la invocación de algún **ejecutable desde terminal** (se busca el ejecutable y se crea un procedimiento `fork-exec()`), o
  - incluso a partir de un *script*.

Una vez que se crea un proceso, empieza a ejecutarse y realiza el trabajo al que está destinado. Sin embargo, nada dura para siempre, ni siquiera los procesos. Tarde o temprano **un proceso terminará**, por lo general debido a una de las siguientes causas:

#### Salidas voluntarias:

1. **Salida normal.** La mayoría de los procesos terminan debido a que han concluido su trabajo. Por ejemplo, cuando un compilador compila un programa añade implícitamente una llamada al sistema para indicar al sistema operativo su terminación normal. Esta llamada es *exit()* en



sistemas POSIX.

En sistemas con **entorno gráfico**, los procesadores de palabras, navegadores de Internet y programas similares siempre tienen un icono o elemento de menú en el que el usuario puede hacer *clic* para indicar al proceso que elimine todos los archivos temporales que tenga abiertos y después termine. Esa acción en realidad lleva a invocar a *exit()* tras ejecutarse un escuchador o **listener de eventos** (hilo del proceso), en este caso el **recoger la acción de clic del ratón**.

2. **Salida controlada por error.** La segunda razón de terminación es que el proceso descubra un error. Por ejemplo, si un usuario escribe el comando “*cc foo.c*” para compilar el programa “foo.c” y no existe dicho archivo, el compilador simplemente termina. Note que este **error es predecible y tenido en cuenta por el proceso**, ya que es el programador el que en su código contempló la opción de terminar el programa en el caso de que no se encontrase el fichero.

#### Salidas involuntarias:

3. **Error fatal.** La tercera razón de terminación es un error fatal producido por el proceso, a menudo debido a un **error en el programa**. Algunos ejemplos incluyen:
  - Ejecutar una instrucción ilegal (operando no permitido para ella, código de llamada al sistema no implementado).
  - Hacer referencia a una parte de memoria no existente o no reservada (*segmentation fault*).
  - División entre cero.

En algunos sistemas, un proceso puede manejar ciertos errores por sí mismo, o incluso recibir una señal que pueda tratar en vez de terminar el proceso. Esto forma parte del **tratamiento de errores y captura de excepciones**.

4. **Eliminado por otro proceso.** La cuarta razón por la que un proceso podría terminar es que algún otro proceso ejecute una llamada al sistema que indique que lo elimine. En sistemas POSIX esta llamada es *kill()*.

También puede ocurrir la finalización con **procesos lanzados desde un terminal**. Si se **cierra el terminal**, éste se encargaría de eliminar a sus procesos hijos (que son los procesos lanzados desde el).

## 6 Un modelo de estados para procesos

La responsabilidad principal del sistema operativo es controlar la ejecución de los procesos y asignar recursos a los mismos. El primer paso en el diseño de un sistema operativo para el control de procesos es **construir un modelo de comportamiento para los procesos**.

Un **esquema general de 5 estados** se muestra la Figura 3.6. Dependiendo del sistema puede haber más o menos estados.



Figura 3.6. Modelo de proceso de cinco estados.

Los cinco estados de este modelo de procesos son los siguientes:

1. **Nuevo.** Un proceso que se acaba de crear y que aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. Típicamente, se trata de un nuevo proceso que **no ha sido cargado en memoria principal**, aunque su **BCP** si ha sido creado por el **planificador a largo plazo**.

Cuando se solicita un nuevo trabajo a un sistema (mediante alguna de las formas de creación de procesos vistas en las secciones anteriores), el sistema operativo realiza todas las tareas internas que correspondan: se asocia un identificador a dicho proceso y se construyen todas aquellas tablas que se necesiten para gestionar al proceso. En este punto, el proceso se encuentra el estado Nuevo, esto significa que el sistema operativo ha realizado todas las tareas necesarias para crear el proceso pero el proceso en sí aún no **está cargado en RAM**.

Un sistema operativo puede **limitar el número de procesos que puede haber por razones de rendimiento** (capacidad de la CPU) o limitaciones de memoria principal (capacidad de la RAM). Mientras un proceso está en el estado Nuevo, la información relativa al proceso que se necesite por parte del sistema operativo se mantiene en la tabla de procesos. Sin embargo, el proceso en sí mismo no se encuentra en memoria principal, es decir, la zona de código, datos, pila y montículo no se encuentra en memoria principal, sino que permanece en almacenamiento secundario codificado en código máquina, normalmente en disco duro.



2. **Ejecutando.** El proceso está actualmente en **ejecución**. Para una mejor comprensión, asumimos que el computador tiene un único procesador con un solo núcleo, de forma que sólo un proceso puede estar en este estado en un instante determinado.
3. **Listo.** Un proceso que se prepara para **ejecutar cuando tenga oportunidad** y lo decida el planificador a corto plazo.
4. **Bloqueado.** Un proceso que no puede ejecutar **hasta que se cumpla un evento determinado**, por ejemplo que se complete una operación E/S que tenía pendiente (esperar a que el usuario introduzca por teclado un número mediante `scanf()`), o que reciba la comunicación de otro proceso al que espera mediante una llamada bloqueante para poder continuar (por ejemplo `wait()`).
5. **Saliente.** Un proceso que ha sido liberado del grupo de procesos ejecutables por el sistema operativo, debido a que **ha terminado por alguna razón** (ya sea **voluntaria** o **involuntaria**).

La terminación de un proceso lo mueve a estado Saliente, de forma que **el proceso no es elegible de nuevo para su ejecución**, no está cargado en RAM.

Las tablas y otra información asociada con el proceso saliente se encuentran **temporalmente preservadas** por el sistema operativo, el cual proporciona **tiempo para que programas auxiliares o de soporte extraigan la información necesaria**. Tome como ejemplo un proceso que termina y que va a ser recogido por su padre (el cual está todavía en ejecución) mediante una primitiva `wait()`. Aunque el proceso hijo haya terminado, su BCP sigue en el sistema hasta que se recoja.

Otros ejemplos podrían ser un **programa de auditoría** que puede requerir **registrar el tiempo de procesamiento y otros recursos utilizados por un proceso saliente**, con objeto de realizar una contabilidad de los recursos del sistema; o un programa de utilidad que requiere extraer información sobre el histórico de los procesos del sistema por temas relativos con el rendimiento o análisis de la utilización. Una vez que estos programas han extraído la información necesaria, el sistema operativo no necesita mantener ningún dato relativo al proceso saliente y se borra del sistema, junto con sus estructuras utilizadas.

## 6.1 Transición entre estados

La Figura 3.6 indica qué tipos de eventos llevan a cada transición de estado (en un modelo de 5 estados) para cada proceso. Las posibles transiciones son las siguientes:

- **Null → Nuevo.** Se crea un nuevo proceso para ejecutar un programa. Este evento ocurre por cualquiera de las relaciones de creación de procesos indicadas en secciones anteriores.
- **Nuevo → Listo.** El sistema operativo (planificador a largo plazo) mueve a un proceso de estado nuevo a listo cuando éste se encuentre preparado para añadir un nuevo proceso en cuestión de recursos. La mayoría de sistemas fijan un límite basado en el número de procesos existentes o la cantidad de memoria primaria y virtual que se podrá utilizar por parte de los procesos existentes. Este límite asegura que no haya demasiados procesos activos y que se degrade el rendimiento del sistema por falta de memoria RAM, memoria

virtual o capacidad de computo de la CPU.

- **Listo → Ejecutando.** Cuando llega el momento de seleccionar un nuevo proceso para ejecutar, el sistema operativo selecciona uno de los procesos que se encuentre en el estado Listo. Esta es una tarea que lleva acabo el planificador a corto plazo según su política de planificación.
- **Ejecutando → Saliente.** El proceso actual en ejecución se finaliza por parte del sistema operativo por cualquiera de las causas de salida o terminación indicadas en las secciones anteriores.
- **Ejecutando → Listo.** La razón más habitual para esta transición son las políticas de planificación, por ejemplo que el proceso en ejecución haya alcanzado el máximo tiempo posible de ejecución de forma ininterrumpida (interrupción por *alarma de reloj*).

Existen otras posibles **causas alternativas para esta transición**, pongamos algunas de ellas:

- Una razón para pasar de estado Ejecutando a estado Listo podría ser porque el sistema operativo tuviera que **atender a una interrupción de Entrada/Salida**, de manera que se expulse el proceso que actualmente se esté ejecutando.
- Otro ejemplo relacionado con el caso anterior y con una **política basada en prioridades** podría ser que un proceso A está ejecutando a un determinado nivel de prioridad, y un proceso B que se encuentra en estado Bloqueado tiene un nivel de prioridad mayor. Si se produce el evento por el cual el proceso B estaba esperando (buffer de E/S, llamada bloqueante, etc) el sistema operativo pondrá el proceso B en estado Listo. Esto podría interrumpir al proceso A y poner en ejecución al proceso B, ya que éste tiene mayor prioridad.
- **Ejecutando → Bloqueado.** Un proceso se pone en el estado Bloqueado si solicita algo por lo cual debe esperar o no está disponible, por ejemplo:
  - Un proceso ha solicitado una **operación de E/S**.
  - Similar al anterior, un proceso realiza una **llamada bloqueante** (wait(), waitpid()).
  - Cuando un **proceso se comunica con otro**, un proceso puede bloquearse mientras está **esperando a que otro proceso le proporcione datos** o esperando un mensaje de ese otro proceso, por ejemplo usando **colas de mensajes** como técnica IPC (*Inter-process Communication*).
- **Bloqueado → Listo.** Un proceso en estado Bloqueado se mueve al estado Listo cuando sucede el evento por el cual estaba esperando.
- **Listo → Saliente.** Por claridad, esta transición no se muestra en el diagrama de estados. En algunos sistemas, un padre puede terminar la ejecución de un proceso hijo en cualquier momento.

Esta transición incluso se puede producir si un proceso es terminado mediante una señal enviada por parte de otro proceso (por ejemplo la señal *kill*).
- **Bloqueado → Saliente.** Se aplican los comentarios indicados en el caso anterior.

## 6.2 Un posible esquema de colas en relación a los estados

La figura 3.8 (a) sugiere la forma de aplicar un esquema de dos colas, la **colas de Listos** y la **cola de Bloqueados**.

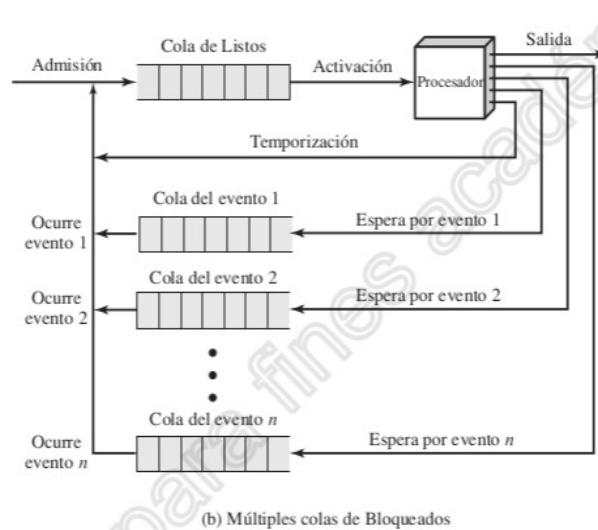
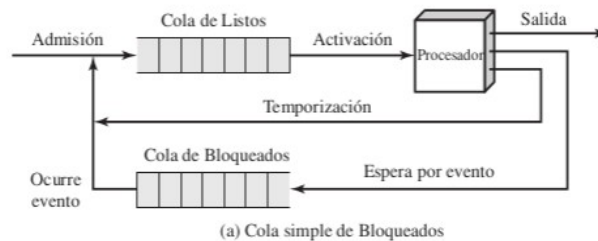


Figura 3.8. Modelo de colas de la Figura 3.6.

Cada proceso admitido por el sistema, se coloca en la cola de Listos.

Cuando llega el momento de que el sistema operativo seleccione otro proceso a ejecutar, selecciona uno de la cola de Listos. En ausencia de un esquema de prioridad, esta cola puede ser una lista de tipo FIFO (*first-in-first-out*).

Cuando el proceso en ejecución termina de utilizar el procesador: o bien finaliza, o bien se coloca en la cola de Listos o de Bloqueados, dependiendo de las circunstancias.

Por último, cuando sucede un evento, cualquier proceso en la cola de Bloqueados que únicamente esté esperando a dicho evento, se mueve a la cola de Listos. Esta última transición significa que, cuando sucede un evento, el sistema operativo debe recorrer la cola entera de Bloqueados, buscando aquellos procesos que estén esperando por dicho evento. En los sistemas operativos con muchos procesos, esto puede significar cientos o incluso miles de procesos en esta lista, por lo que sería mucho más eficiente tener una cola por cada evento. De esta forma, cuando sucede un evento, la lista entera de procesos de la cola correspondiente se movería al estado de Listo, Figura 3.8(b).

**Un refinamiento final sería:** si la activación de procesos está dictada por un esquema de prioridades, sería conveniente tener **varias colas de procesos Listos, una por cada nivel de prioridad**. El sistema operativo podría determinar cual es el proceso listo de mayor prioridad simplemente seleccionando éstas en orden.

### 6.3 Procesos suspendidos

Existe una buena justificación para añadir otros dos estados al modelo de 5 estados anterior, concretamente el estado **Listo/Suspendido** y **Bloqueado/Suspendido** (hay más pero estos son los más interesantes).

Para ver este beneficio de los nuevos estados, suponga un sistema que puede utilizar memoria virtual o *swapping*. Como siempre, cada proceso que se ejecuta en un sistema debe cargarse completamente en memoria principal (imagen), pero puede haber situaciones en que eso no sea posible, por ejemplo:

- Es necesario dejar libre parte de la memoria RAM para otro proceso que es más prioritario con una imagen muy grande, y no hay suficiente RAM en ese momento concreto.
- La RAM está tan llena que no cogen más imágenes de procesos.

En estos casos lo que se hace es pasar procesos (excepto su BCP) que estén en estado Listo o en estado Bloqueado a *swap* (memoria secundaria, normalmente el disco duro), teniéndose estos dos nuevos estados: **Bloqueado/Suspendido** y **Listo/Suspendido**.

Cuando un proceso está **en estado Suspendido no esta listo para su ejecución**, ya que solo queda en **memoria principal su BCP**, y al encontrarse la zona de código, datos, montículo y pila en memoria virtual (en un dispositivo de E/S) el sistema se volverá más lento e ineficiente.

Las características de los dos nuevos estados son las siguientes:

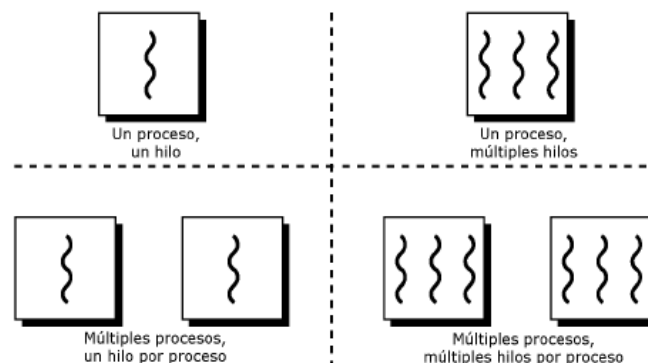
- **Listo/Suspendido.** El proceso está en almacenamiento secundario pero está disponible para su ejecución tan pronto como sea cargado en memoria principal.
- **Bloqueado/Suspendido.** El proceso está en almacenamiento secundario y esperando un evento.

Cuando el **sistema operativo** ha realizado una operación de *swap*, tiene **dos opciones para seleccionar un nuevo proceso para traerlo a estado Listo**:

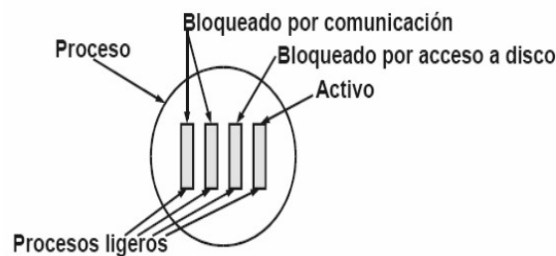
1. Puede **admitir un Nuevo proceso** que se haya creado por parte del planificador a largo plazo.
2. Puede **traer un proceso que anteriormente se encontrase en estado de Suspendido**. Parece que sería preferible traer un proceso que anteriormente estuviese suspendido, para proporcionar dicho servicio en lugar de incrementar la carga total del sistema, pero eso dependerá de la política de planificación y de los recursos existentes.

## 7 Hilos (hebras)

La mayoría de los sistemas operativos modernos proporcionan características que permiten que un proceso tenga múltiples hilos. Un **hilo**, también llamado **proceso ligero o hebra**, es una unidad básica de utilización de la CPU. Si un **proceso** tradicional, también llamado **proceso pesado**, tiene múltiples hilos (además del suyo propio – *main()* –), entonces podría realizar más de una tarea a la vez si el sistema dispone de varios núcleos.



Cada **hilo** perteneciente a un proceso **posee lo siguiente de manera individual**:



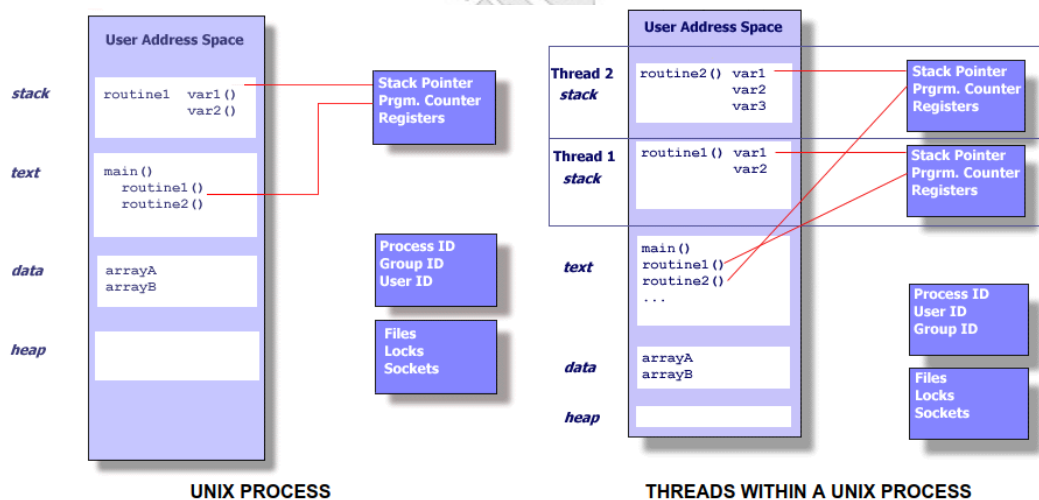
- Un **estado de ejecución** por hilo: Ejecutando, Listo o Bloqueado.
- Un **contexto o BCP** de hilo propio. Vea un hilo como un contador de programa independiente dentro de un proceso.
- Una **pila de ejecución** por hilo, con sus correspondientes registros de activación (dirección de retorno, parámetros, valor devuelto, variables locales, etc).

Por otro lado, cada **hilo** posee **información que es compartida con el proceso** al que pertenece y con los restantes hilos de dicho proceso:

- **Zona de código**, ya que los hilos se crean desde el mismo código del proceso padre.
- **Zona de datos estáticos** (globales, *static* en C, *const* en C). Las variables globales hay que protegerlas adecuadamente para no corromperlas, por ejemplo con semáforos (se estudiarán en otro tema).
- **Montículo**. La reserva de memoria que haga un proceso padre o un hilo es visible para el resto de hilos, basta con que sepan la dirección de memoria de inicio del bloque reservado para poder trabajar con el.
- **Otros recursos asociados al proceso**: Señales y manejadores de señales, directorio de trabajo actual, variables de entorno del proceso.

Todos los hilos de un proceso comparten el estado general del proceso más el suyo propio y los recursos de ese proceso, residen en el **mismo espacio de direcciones en RAM** y **tienen acceso a los mismos datos reservados en el montículo**. Tendrá que tener especial cuidado cuando desde una hebra quiera **devolver determinada información**, ya que si no la ha **reservado** previamente en el **montículo** ésta se perderá al terminar el hilo.

De esta forma, cuando un hilo cambia determinados datos en memoria de montículo, otros hilos ven los resultados cuando acceden a estos datos (no en las variables locales propias del hijo).



Los cuadrados azul oscuro representan a información del BCP, de manera que se vea que tanto el proceso como los hilos tienen su propio BCP.



Muchos paquetes de software que se ejecutan en los PC modernos de escritorio son multihebra. Normalmente, una aplicación se implementa como un proceso propio con varias hebras de control.

Veamos algunos ejemplos del uso de hebras:

- Un **servidor web** acepta solicitudes de los clientes que piden páginas web, imágenes, sonido, etc. Un servidor web sometido a una gran carga puede tener varios cientos de clientes accediendo de forma concurrente a él. Si el servidor web funcionara como un proceso tradicional de una sola hebra, sólo podría dar servicio a un cliente cada vez y la cantidad de tiempo que un cliente podría tener que esperar para que su solicitud fuera servida podría ser enorme.

Una solución es que el servidor funcione como un solo proceso de aceptación de solicitudes. Cuando el servidor recibe una solicitud, crea otro hilo para dar servicio a dicha solicitud.

- Los hilos también se pueden utilizar para **trabajos en primer plano y en segundo plano**. Por ejemplo, un **procesador de textos** puede tener una hebra para mostrar gráficos, otra hebra para responder a las pulsaciones de teclado del usuario y una tercera hebra para el corrector ortográfico y gramatical que se ejecuta en segundo plano. También se usan en procesamiento asíncrono, por ejemplo, puede haber un hilo de protección contra un fallo de corriente o cierre del procesador de textos, de manera que salve automáticamente un documento a disco una vez cada cierto tiempo.

## 7.1 Estados de los hilos

En un sistema operativo que soporte hilos, la planificación y la activación se realizan a nivel de hilo, por tanto, los principales estados de los hilos son: **Ejecutando**, **Listo** y **Bloqueado**.

Existen diversas acciones que afectan a todos los hilos de un proceso y que el sistema operativo debe gestionar a nivel de proceso. **Suspender un proceso implica expulsar el espacio de direcciones de un proceso** de memoria principal para dejar hueco a otro espacio de direcciones de otro proceso. Ya que **todos los hilos** de un proceso comparten el mismo espacio de direcciones, todos los hilos **se suspenderían al mismo tiempo**.

De forma similar, la **finalización de un proceso** (de manera voluntaria o involuntaria) **termina todos los hilos de ese proceso**, ya que todos están dentro del mismo espacio de direcciones de memoria liberado a la finalización.

Como ejercicio, consulte e investigue en la Web cómo consultar los hilos tiene un proceso y sus estados en un sistema GNU/Linux actual.

## 7.2 Motivación y ventajas de las hebras

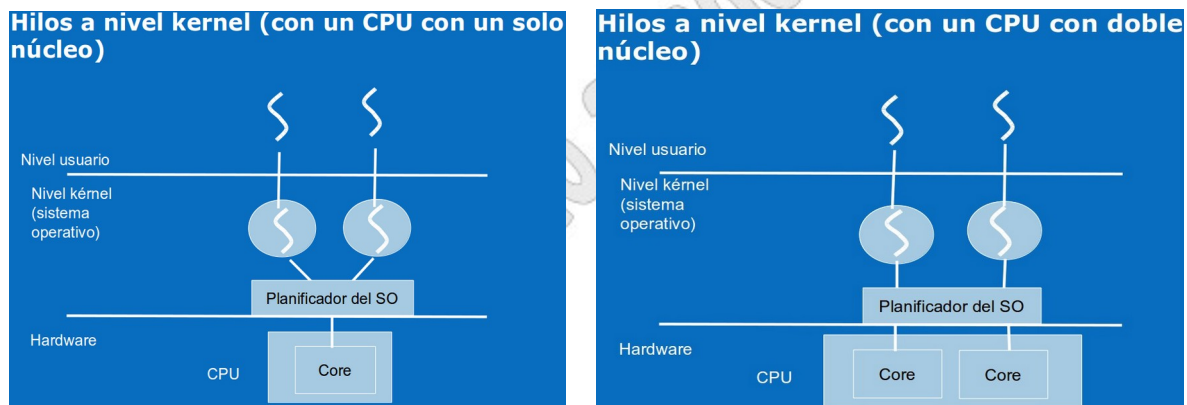
Las **ventajas de la programación multihebra** pueden dividirse en 4 categorías principales:

1. **Utilización sobre arquitecturas multiprocesador (al igual que los procesos).** Las ventajas de usar configuraciones multihebra también se obtienen en una arquitectura multiprocesador, donde las hebras pueden ejecutarse en paralelo en los diferentes procesadores o núcleos.

El kernel mantiene información de contexto del proceso como una entidad y de los hilos individuales del proceso. **La planificación realizada por el núcleo se realiza a nivel de hilo.**

El uso de múltiples hebras en una aplicación permite que un programa continúe ejecutándose incluso aunque parte de él esté bloqueado o realizando una operación muy larga. Por ejemplo, un explorador web permite la interacción del usuario a través de una hebra mientras que en otra hebra se está cargando una imagen.

La siguiente Figura muestra una representación de hilos a nivel de núcleo gestionados por el sistema operativo:



2. **Compartición de recursos propios del proceso.** Las hebras comparten parte de la memoria y los recursos del proceso al que pertenecen, **no siendo necesario su duplicidad.**
3. **Tiempo de ejecución.** En general, se consume mucho más tiempo en crear y gestionar los procesos que las hebras. Por ejemplo, crear un proceso es treinta veces lento que crear una hebra,
  - **Crear o terminar una hebra de un mismo proceso** es mucho **más rápido** que crear o terminar un proceso, ya que las hebras comparten determinados recursos del padre que no hay que crear de nuevo. Crear un proceso es treinta veces lento que crear una hebra. Cuando se termina un proceso se debe eliminar el BCP del mismo y todas las estructuras de datos y espacio de memoria de usuario ocupado, mientras que si se termina un hilo se elimina solo su BCP y pila.
  - El **cambio de contexto entre hebras de un mismo proceso** es realizado mucho **más**



**rápidamente que el cambio de contexto entre procesos**, mejorando el rendimiento del sistema. El cambio de contexto es aproximadamente cinco veces más lento a nivel de proceso que a nivel de hebra. En los hilos, como pertenecen a un mismo proceso, al realizar un cambio de hilo el tiempo perdido es casi despreciable, ya que hay estructuras de datos que no son necesarias de salvar/restaurar.

- Si se necesitase **comunicación entre hebras** también sería mucho **más rápido que intercomunicar procesos**, ya que los datos están inmediatamente habilitados y disponibles entre hebras. En la mayoría de los sistemas, en la comunicación entre procesos debe intervenir el núcleo en la invocación de determinadas llamadas al sistema para realizar la comunicación (por ejemplo memoria compartida - IPC). En cambio, los **hilos de un mismo proceso pueden comunicarse entre sí sin la invocación al núcleo** (comparten memoria en el **montículo, variables globales**).

De esta forma, si se desea implementar una aplicación como un conjunto de unidades de ejecución relacionadas, es mucho más eficiente hacerlo con un conjunto de hilos que con un conjunto de procesos independientes.

## 8 Servicios en GNU/Linux

Un **demonio o servicio** es un **proceso** que se **ejecuta en segundo plano**, fuera del control interactivo directo de los usuarios del sistema, ya que carecen de interfaz con estos a excepción de poder pararlos o reanudarlos (se auto-gestionan). Generalmente están esperando a que ocurran eventos y ofreciendo servicios.

El término demonio se usa más frecuentemente en sistemas GNU/Linux o Mac OS, aunque también se usa el termino servicio.

**Tradicionalmente** en sistemas GNU/Linux y derivados, los nombres de los **demonios terminaban con la letra “d”**, por lo que podemos encontrar nombres de demonios como por ejemplo “*sshd*”, que es el demonio que sirve a las conexiones SSH entrantes en un servidor (acceso a una máquina mediante conexión remota).

El **sistema inicia los demonios durante el arranque**, una vez que se carga el kernel, aunque no tiene porque ser así, **se pueden lanzan nuevos demonios en cualquier momento**.

Algunos de los demonios más usuales lanzados en el arranque del sistema son los siguientes:

- **acpid**: Es un demonio que maneja el apagado del sistema cuando el usuario pulsa el botón de encendido del equipo, y controla la energía y acciones a llevar a cabo al pulsar determinados botones en nuestro ordenador (por ejemplo hibernar).
- **cron**: Es un demonio que se ejecuta para lanzar tareas planificadas.
- **networking**: Demonio necesario para que funcione la interfaz de red, independientemente del protocolo utilizado.
- **apache2**: Demonio del servidor web Apache que permite servir páginas Web. Este demonio está continuamente esperando a una solicitud para entregar una página web.
- **smbd**: Demonio para poder acceder a carpetas, archivos e impresoras en red,

independientemente de que sean de Windows o GNU/Linux. Se denomina **Samba** a este servicio.

- **cups:** Demonio del servicio de impresión, necesario para poder enviar y recibir información de las impresoras.
- **udev:** Demonio necesario para la detección de hardware.
- **alsa-utils:** Demonio de sonido, necesario para poder activar y controlar el sistema de sonido.
- **udisksd:** Demonios para montar en el arranque determinados sistemas de ficheros (ntfs, fat, ext).
- **lm-sensors:** Demonio para hacer auditorias y sensores, como por ejemplo la frecuencia de trabajo de la CPU.
- **bluetooth:** Demonio encargado de las conexiones por Bluetooth e infrarrojos (irDA).
- **Servidor de correo:** Demonio para poder aceptar conexiones de correo electrónico en una máquina que actúe como servidor.

## 8.1 Registro de la actividad de los demonios

Puesto que un demonio se ejecuta en segundo plano, **no debe estar conectado a ninguna terminal o shell**, ya que cualquier proceso que se ejecute desde el *shell* o interprete de comandos es hijo de este (la *shell* lanza un *fork()* y posteriormente un *exec(comando)*), y si se cierra la terminal se cerraría el demonio.

Por lo tanto los procesos que quieren convertirse en un demonio deben asegurarse que siempre se ejecutan en segundo plano, desvinculándose del proceso de la *shell* que los invocó. Para ello hay que crear y **configurar una serie de ficheros** que hay que localizar en un parte determinada del sistema (puede consultar en la Web como crear demonios en sistemas basados en *Upstart* o *Systemd*).

Esto plantea la cuestión de **¿cómo indicar condiciones de error, advertencias u otro tipo de sucesos del servicio?**.

1. Algunos demonios **almacenan los mensajes en archivos específicos** o en su propia base de datos de sucesos.
2. Sin embargo en muchos sistemas existe un servicio específico, es decir, **un demonio al que otros demonios pueden enviar mensajes para registrar sus eventos**. Por ejemplo, el demonio *rsyslog* recibe y registra eventos de otros demonios a través de la función *syslog()*.

```
// Abrir una conexión al demonio syslog. Puede consultar Posix para una completa
descripción
openlog(argv[0], LOG_NOWAIT | LOG_PID, LOG_USER);
...

// Enviar un mensaje al demonio syslog
syslog(LOG_NOTICE, "Demonio iniciado con éxito\n");
```

```
...  
  
// Cuando el demonio termine, cerrar la conexión con el servicio syslog  
closelog();
```

### **¿Donde están localizados los ficheros de log?**

Todas las distribuciones de Linux utilizan archivos de registro para registrar los eventos del sistema, incluyendo conectar dispositivos, sesiones nuevas y otros mensajes. La mayoría de las distribuciones guardan estos archivos en el directorio **/var/log**. Algunos de los archivos están protegidos, hace falta iniciar una sesión como usuario **root** o usar **sudo** para leerlos.

Los archivos de registro varían por distribución. No obstante, la mayoría de las distribuciones GNU/Linux suelen tener siguientes archivos similares a los siguientes:

- **syslog**: Registro del sistema de registro (`cat /var/log/syslog | more`)
- **auth.log**: Contiene información de autorización del sistema, incluidos los inicios de sesión de los usuarios y los mecanismos de autenticación que se utilizaron.
- **cron.log**: Registra tareas de *cron*. Puede estar deshabilitado, quedar en blanco y escribir en *syslog*.
- **daemon.log**: Registra alertas de servicios como *ntfs-3g* (permite soporte completo de lectura y escritura en sistemas de archivos *ntfs*). Puede estar deshabilitado, quedar en blanco y escribir en *syslog*.
- **dmesg**: En este archivo se almacena la información que genera el kernel durante el arranque del sistema, por ejemplo información sobre los dispositivos de hardware que el kernel detectó durante el proceso de arranque. Podemos ver su contenido con el comando **dmesg**.

## **8.2 Arranque de servicios con Upstart (basado en SysVinit)**

Recordemos la última etapa de arranque del sistema vista en secciones anteriores. Los demonios se han lanzado clásicamente a través del proceso **init**, es lo que se conoce como arranque al estilo tradicional de **Unix System-V** o **SysVinit**.

**Upstart** es un sistema de arranque del espacio de usuario más moderno y **basado en SysVinit**, y es el sistema de arranque más utilizado hasta hace poco tiempo en los sistemas GNU/Linux. Por último, **Systemd** está sustituyendo a **Upstart** y es el sistema de arranque a nivel de usuario que se utiliza actualmente y que en algunos sistemas GNU/Linux convive con **Upstart** (**SysVinit** → **Upstart** → **Systemd**).

El proceso **init** es llamado por el núcleo de GNU/Linux para **iniciar el espacio de usuario durante el proceso de arranque y gestionar (arrancar, recargar, terminar...) posteriormente todos los demás servicios y procesos a nivel de usuario**. Por tanto **init** es el primer proceso que se inicia a nivel de usuario (se ejecuta en el espacio de usuario), y una de sus funciones es **lanzar los demonios necesarios a partir de scripts** escritos en *bash* (lenguaje de *scripting*<sup>2</sup>) bajo un

---

2 <https://es.wikipedia.org/wiki/Script>

determinado formato y directrices que el administrador del sistema debe seguir.

### ¿Qué demonios se arrancan con init?

Para ello, **init opera indicando un nivel de ejecución** que define un estado de la máquina después del arranque, **estableciendo, entre otras cosas, qué demonios deben ser iniciados** a través del lanzamiento de los citados *scripts*.

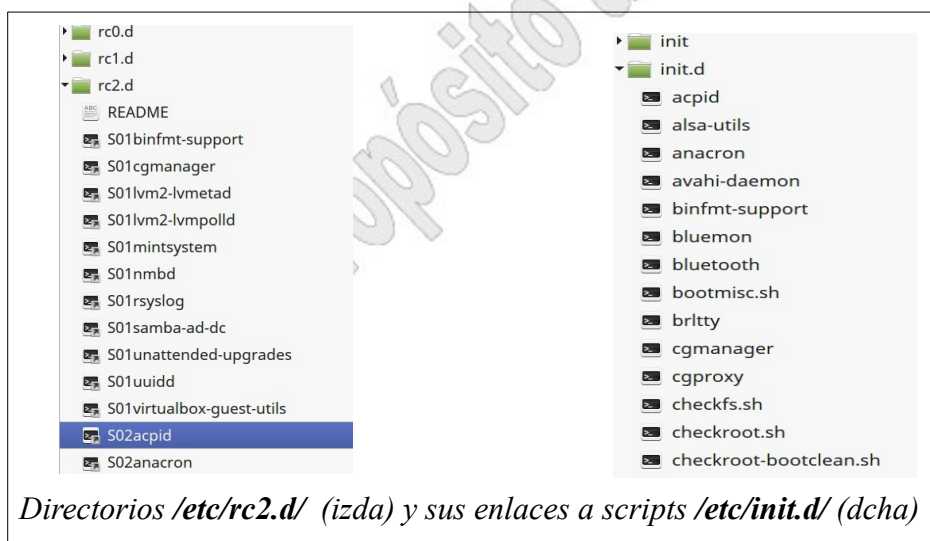
Por lo general existen **7 niveles, del 0 al 6** (depende del sistema operativo):

- En términos prácticos cuando el computador entra al **runlevel 0** está **apagado**,.
- Cuando entra al **runlevel 6** se **reinicia**.
- Los **runlevels** intermedios (**1 a 5**) difieren en relación a qué servicios son iniciados.

Los *scripts* de *bash* que puede lanzar *init*, que a su vez ejecutarán ficheros binarios, por ejemplo de */sbin*, */bin*, */usr/sbin*, */usr/bin*, etc, se localizan en el directorio **/etc/init.d/**.

En **/etc/init.d/** se encuentran los *scripts* que se encargan de lanzar servicios. Estos *scripts* están enlazados mediante enlaces simbólicos (*symlink*) desde directorios con nombres como **/etc/rc0.d/** a **/etc/rc6.d/**, uno para cada nivel. Por ejemplo, en **/etc/rc3.d** se encuentran los enlaces simbólicos a los *scripts* del directorio **/etc/init.d/** que se lanzarán en el nivel 3.

La siguiente figura muestra parte del contenido del directorio **/etc/rc2.d/** que contiene enlaces simbólicos a los *scripts* que se encuentran en el directorio **/etc/init.d/**. Los scripts de los directorios **/etc/rc2.d/** a **/etc/rc5.d/** suelen tener como prefijo la letra “S” de “Start”.



Durante el arranque de sistema, antes de iniciar los *scripts* que haya en alguno de los directorios **etc/rc?.d/**, se produce un arranque de los *scripts* situados en el directorio **/etc/rcS.d/**, que sirven para completar la inicialización de determinado hardware básico (teclado, sensores de la cpu, sonido). Después se procede a ejecutar todos los enlaces simbólicos del directorio **/etc/rc?.d/** relativos al nivel de ejecución especificado.

Los detalles particulares de configuración del *runlevel* varía entre sistemas operativos, aunque hay proyectos de estandarización (LSB – *Linux System Standard*).

La mayoría de las distribuciones Linux, definen los niveles de ejecución mostrados en la siguiente tabla. Los **niveles más bajos** se utilizan para el **mantenimiento o la recuperación de emergencia**, ya que por lo general no ofrecen ningún servicio de red:

Nivel de ejecución	Nombre o denominación	Descripción
0	Alto	Alto o cierre del sistema (apagado). Se ejecutan <i>scripts</i> para finalizar demonios. No se debe poner este nivel como predeterminado.
1	Modo monousuario	No configura la interfaz de red o los demonios de inicio, ni permite que ingresen otros usuarios que no sean el usuario <i>root</i> . Este nivel de ejecución permite reparar problemas, o hacer pruebas en el sistema.
2	Multiusuario	Multiusuario sin soporte de red (sin NFS o xinetd - demonio de servicios extendidos de Internet- )
3	Multiusuario con soporte de red	Inicia el sistema normalmente. GNU/Linux completamente funcional con soporte multiusuario y acceso a la red. La interfaz de usuario es en modo texto.
4	Multiusuario con soporte de red.	Similar al 3 o sin uso, reservado para administradores de sistemas.
5	Multiusuario gráfico (X11)	Nivel de ejecución 3 + display manager (entorno gráfico usando gestores de pantalla como <i>gdm</i> - <i>GNOME Display Manager</i> - ). Normalmente siempre se arranca en este nivel.
6	Reinicio	Se reinicia el sistema. Se ejecutan <i>scripts</i> para finalizar demonios. No se debe poner este nivel como predeterminado.

**Después** de que se han **lanzado todos los *scripts***, el proceso *init* se “**aletarga**” y espera a que uno de estos eventos sucedan:

- Que determinados procesos del sistema terminen y tenga que recogerlos *init* como proceso padre.
- Una petición a través del proceso */sbin/telinit* para cambiar el nivel de ejecución.

**En el apagado o en el reinicio de la computadora**, *init* es llamado a **cerrar** todas las funcionalidades del espacio de usuario de una manera controlada, de nuevo a través de secuencias de comandos o *scripts* dentro de los directorios */etc/rc0.d* o */etc/rc6.d* respectivamente, tras lo cual *init* termina y el núcleo ejecuta el apagado. Estos *scripts* se suelen nombrar poniendo como prefijo la letra “**K**” de “**Kill**”.

El administrador puede cambiar en cualquier momento el valor del nivel de ejecución con el comando *telinit*. Una instrucción como la siguiente:

```
# telinit 3
```

cambia a nivel de ejecución 3. Si se hace esto se dejará de tener una pantalla gráfica y aparecerá una terminal, y es probable que se pierda la información que no se tenga guardada. Para regresar al modo gráfico es necesario establecer de nuevo a 5 (depende de la distribución de



GNU/Linux). El comando **telinit** no altera el contenido de los archivos de configuración de más alto nivel de **init**.

¿Cómo podemos saber cuál es el nivel de ejecución actual del sistema?

- Con el comando **runlevel**.
- Con el comando **who** y la opción **-r**.

```
jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ runlevel
N 5
jfcaballero@eniac:~$ who -r
`run-level' 5 2021-11-10 08:54
jfcaballero@eniac:~$
```

Puede consultar los servicios activos en su máquina con el siguiente comando, mostrándose la salida que aparece en la siguiente figura. Puede comprobar que los servicios que aparecen están en los directorios **/etc/rcS.d** y **/etc/rc5.d** (si hemos arrancado en nivel 5).

**# service --status-all**

No debe confundir este comando y su salida con el comando **pstree -p** (visualizar el árbol de procesos, mostrando la relación padre hijo y sus PID), el cual **muestra todos los procesos a nivel de usuario**, los cuales son todos hijos del proceso **init**. No todos los procesos de usuario son servicios o demonios, por lo que habrá procesos al ejecutar **#pstree -p** que no verá reflejados al ejecutar **# service --status-all**.

```
jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ service --status-all
[ + ] acpid
[ - ] alsa-utils
[ - ] anacron
[ + ] apparmor
[ + ] avahi-daemon
[ + ] binfmt-support
[ + ] bluetooth
[ - ] console-setup.sh
[ + ] cron
[ - ] cryptdisks
[ - ] cryptdisks-early
[ + ] cups
[ + ] cups-browsed
[ + ] dbus
[ - ] dns-clean
[ - ] grub-common
[ + ] hddtemp
[ - ] hwclock.sh
[ + ] irqbalance
[ + ] kerneloops
[ - ] keyboard-setup.sh
[ + ] kmod
[ + ] lightdm

jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ pstree -p
systemd(1)─ModemManager(792)─{ModemManager}(811)
               │               │{ModemManager}(815)
               └─NetworkManager(694)─{NetworkManager}(772)
                                   │{NetworkManager}(784)
               └─accounts-daemon(684)─{accounts-daemon}(691)
                                   │{accounts-daemon}(779)
               └─acpid(685)
               └─agetty(864)
               └─avahi-daemon(687)─avahi-daemon(735)
               └─blueberry-tray(1600)─safechild(1612)─rfkill(1615)
                                   │{blueberry-tray}(1603)
                                   │{blueberry-tray}(1604)
                                   │{blueberry-tray}(1608)
                                   │{blueberry-tray}(1609)
               └─bluetoothd(688)
               └─boltd(793)─{boltd}(810)
                           │{boltd}(823)
               └─colord(785)─{colord}(803)
                           │{colord}(807)
               └─cron(689)
               └─csd-printer(1269)─{csd-printer}(1272)
                                   │{csd-printer}(1273)
               └─cups-browsed(781)─{cups-browsed}(850)
```

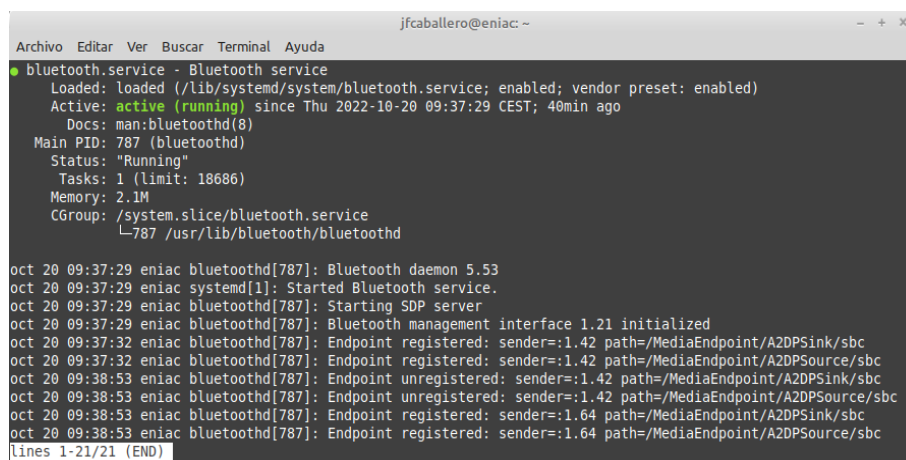
### 8.2.1 Gestión de un servicio, comando **service**:

Los parámetros que admite un determinado servicio se pasan mediante el comando **service** y el nombre del servicio. Por ejemplo para **bluetooth**:

**# service bluetooth**

*Usage: /etc/init.d/bluetooth {start|stop|restart|force-reload|status}*

- **# service bluetooth status:** Muestra información más detallada sobre el estado de un servicio, en este caso **bluetooth**.



```
jfcaballero@eniac: ~  
Archivo Editar Ver Buscar Terminal Ayuda  
● bluetooth.service - Bluetooth service  
   Loaded: loaded (/lib/systemd/system/bluetooth.service; enabled; vendor preset: enabled)  
   Active: active (running) since Thu 2022-10-20 09:37:29 CEST; 40min ago  
     Docs: man:bluetoothd(8)  
  Main PID: 787 (bluetoothd)  
   Status: "Running"  
    Tasks: 1 (limit: 18686)  
   Memory: 2.1M  
   CGroup: /system.slice/bluetooth.service  
           └─787 /usr/lib/bluetooth/bluetoothd  
  
oct 20 09:37:29 eniac bluetoothd[787]: Bluetooth daemon 5.53  
oct 20 09:37:29 eniac systemd[1]: Started Bluetooth service.  
oct 20 09:37:29 eniac bluetoothd[787]: Starting SDP server  
oct 20 09:37:29 eniac bluetoothd[787]: Bluetooth management interface 1.21 initialized  
oct 20 09:37:32 eniac bluetoothd[787]: Endpoint registered: sender=:1.42 path=/MediaEndpoint/A2DPSink/sbc  
oct 20 09:37:32 eniac bluetoothd[787]: Endpoint registered: sender=:1.42 path=/MediaEndpoint/A2DPSink/sbc  
oct 20 09:38:53 eniac bluetoothd[787]: Endpoint unregistered: sender=:1.42 path=/MediaEndpoint/A2DPSink/sbc  
oct 20 09:38:53 eniac bluetoothd[787]: Endpoint unregistered: sender=:1.42 path=/MediaEndpoint/A2DPSink/sbc  
oct 20 09:38:53 eniac bluetoothd[787]: Endpoint registered: sender=:1.64 path=/MediaEndpoint/A2DPSink/sbc  
oct 20 09:38:53 eniac bluetoothd[787]: Endpoint registered: sender=:1.64 path=/MediaEndpoint/A2DPSink/sbc  
lines 1-21/21 (END)
```

Otros comandos interesantes de **service** son:

- **# service bluetooth start:** Inicia el servicio indicado en *service*.
- **# service bluetooth stop:** Para el servicio indicado en *service*.
- **# service bluetooth restart:** Reinicia el servicio indicado en *service*.
- **# service bluetooth force-reload:** Recarga el servicio indicado en *service* sin llegar a pararlo. Si el servicio no permite la recarga de la configuración se reinicia. Si se permite la recarga, al ejecutar el comando se aplicarán los cambios realizados en los archivos de configuración del servicio sin tan siquiera llegar a parar el servicio.

Actualmente el proceso de arranque **Upstar con init** se está **sustituyendo** por el nuevo sistema de inicio **systemd**, pero este último se ha hecho compatible hacia atrás con **init**, con lo que aún se puede gestionar el arranque del espacio de usuario y sus servicios de la manera tradicional.

### 8.3 Arranque de servicios con el sistema *systemd*

*Systemd* fue diseñado para **reemplazar** al mencionado *Upstart* basado en *SysVinit* y su proceso *init*. El principal motivo de la aparición de *systemd* es **unificar para todas las distribuciones de GNU/Linux** las configuraciones básicas de la **administración de servicios** y su **conexión con el núcleo**. Desde 2015 la mayoría de las distribuciones ya usan *Systemd*.

- *Systemd* es un “**super servicio**” (en enero de 2013, Poettering, su autor, describió a *systemd* como una *suite* o conjunto de aplicaciones que incluía 69 binarios individuales).
- Permite en el **arranque la ejecución paralela de procesos** (el demonio *init* tradicional es estrictamente síncrono, bloqueando futuras tareas hasta que la actual se haya completado),
- **Reemplaza varios servicios del sistema**, como por ejemplo *journal*, que es el sistema de registros o *log* de *systemd*. En este caso **reemplaza o convive con *syslog***.

Esto ha generado gran controversia dentro de la comunidad del software libre. Los críticos se argumentan en al menos dos niveles:

1. **Arquitectura interna:** Critican que *systemd* es **demasiado complejo** y que sufre de una continua invasión de nuevas características, absorbiendo cada vez más funciones, procesos y demonios, además de ser **más complicado de administrar que *Upstart***.
2. **Implementación futura:** También existe la preocupación de que se torne en un sistema de dependencias entrelazadas, **obligando** así a los **mantenedores de distribuciones** a no tener más remedio que **dependen de *systemd*** a medida que más **piezas de software del espacio de usuario** sigan **dependiendo de sus componentes**.

En *systemd* los **demonios** se definen y configuran a partir de los llamados **archivos de unidad** o **unit files**, que son ficheros con un **tipo de extensión definida** (hasta 12 extensiones distintas, los ficheros con extensión *.service* y los *.target* son los más interesantes), y que se alojan en varios directorios.

En *systemd* los **diferentes niveles de arranque en los subdirectorios que usaba *init***, se denominan *target*, que son **ficheros de unidad con extensión *.target*** de los cuales **dependen los ficheros de unidad *.service*** e incluso otros ficheros *.target*.

En el directorio de configuración */etc/systemd/system/* (similar al directorio */etc/init.d/* en el sistema *Upstart*) se encuentran enlaces simbólicos hacia el directorio */lib/systemd/system/* donde están los servicios (similar a los directorios */etc/rc?.d* en el sistema *Upstart*).

En la siguiente tabla se muestran algunas **equivalencias** entre los niveles de *Upstart* y *Systemd*. Hay dos *.target* por cada nivel, ya que en algunos sistemas toman un nombre u otro:

Niveles de Upstart	Unidades <i>.target</i> de Systemd	Descripción
0	runlevel0.target, poweroff.target	Alto o cierre del sistema (apagado), mediante el comando <i>halt</i> . No se debe poner este nivel como predeterminado.
1	runlevel1.target, rescue.target	No configura la interfaz de red o los demonios de inicio, ni permite que ingresen otros usuarios que no sean el usuario <i>root</i> . Este nivel de ejecución permite reparar problemas, o hacer pruebas en el sistema.



2	runlevel2.target, multi-user.target	Multiusuario sin soporte de red (sin NFS o xinetd - demonio de servicios extendidos de Internet- )
3	runlevel3.target, multi-user.target	Inicia el sistema normalmente. GNU/Linux completamente funcional con soporte multiusuario y acceso a la red. La interfaz de usuario es en modo texto.
4	runlevel4.target, multi-user.target	Similar al 3 o sin uso, reservado para personalización para administradores de sistemas.
5	runlevel5.target, graphical.target	Nivel de ejecución 3 + display manager (entorno gráfico usando gestores de pantalla como <i>gdm</i> - <i>GNOME Display Manager</i> - ). Normalmente siempre se arranca en este nivel.
6	runlevel6.target, reboot.target	Se reinicia el sistema. No se debe poner este nivel como predeterminado.

De los 12 tipos de unidades, las más interesantes son *.service* y *.target*.

- **.service**: Una unidad *.service* controla demonios y los procesos relativos a ellos. Es el tipo de unidad que interesa para crear un nuevo servicio.
- **.target**: Las unidades *.target* codifican información respecto a agrupar unidades *.service* y lograr una buena sincronización entre ellas. Se encargan de las dependencias entre unidades y sus relaciones entre si.

Se deja al lector interesado la consulta de la creación y gestión de demonios usando *systemd*, ya que está más ligado a la administración de sistemas, indicándose aquí conceptos generales.

### 8.3.1 Gestión de un servicio, comando *systemctl* (sustituto de *service*)

El comando *service* de *init* se **sustituye** o **convive** con el comando *systemctl* de *systemd*.

- **# *systemctl list-units --type service***: Permite obtener un listado de la totalidad de servicios que actualmente están activos:
  - *UNIT*: El nombre de la unidad.
  - *LOAD*: Si la información de la configuración de la unida está cargada en memoria.
  - *ACTIVE*: Informa de si la unidad está o no activa.
  - *SUB*: Muestra más información sobre la unidad.
  - *DESCRIPTION*: Una descripción breve de lo que hace la unidad.

```
jfcaballero@eniatic ~
Archivo Editar Ver Buscar Terminal Ayuda
UNIT LOAD ACTIVE SUB DESCRIPTION
accounts-daemon.service loaded active running Accounts Service
acpid.service loaded active running ACPI event daemon
avahi-daemon.service loaded active running Avahi mDNS/DNS-SD Stack
binfmt-support.service loaded active exited Enable support for additional
bluemon.service loaded active exited LSB: Bluetooth monitoring daem
bluetooth.service loaded active running Bluetooth service
cgmanager.service loaded active running Cgroup management daemon
colord.service loaded active running Manage, Install and Generate C
console-kit-daemon.service loaded active running Console Manager
console-kit-log-system-start.service loaded active exited Console System Startup L
console-setup.service loaded active exited Set console font and keymap
cpufrequtils.service loaded active exited LSB: set CPUFreq kernel param
cron.service loaded active running Regular background program pro
cups-browsed.service loaded active running Make remote CUPS printers avai
cups.service loaded active running CUPS Scheduler
dbus.service loaded active running D-Bus System Message Bus
getty@tty1.service loaded active running Getty on tty1
grub-common.service loaded active exited LSB: Record successful boot fo
hddtemp.service loaded active exited LSB: disk temperature monitori
irqbalance.service loaded active running LSB: daemon to balance interr
keyboard-setup.service loaded active exited Set console keymap
kmod-static-nodes.service loaded active exited Create list of required static
lm-sensors.service loaded active exited Initialize hardware monitoring
loadcpufreq.service loaded active exited LSB: Load kernel modules neede
lvm2-lvmetad.service loaded active running LVM2 metadata daemon
lvm2-monitor.service loaded active exited Monitoring of LVM2 mirrors, sn
mdm.service loaded active running MDM Display Manager
lines 1-28
```

- ***systemctl list-units --type service --all***: Permite obtener un listado de la totalidad de servicios, independientemente de si están activos o inactivos.
- ***systemctl status unit\_name.service***: Permite ver la información de estado sobre el servicio en cuestión (por ejemplo para bluetooth sería *systemctl status bluetooth.service*):
  - *Loaded* - Información sobre si la unidad servicio ha sido cargada y la ruta absoluta.
  - *Active* - Información sobre si la unidad de servicio está ejecutándose, con *timestamp* incluido.
  - *Main PID* - El PID del servicio, seguido de su nombre.
  - *Process* - Información adicional sobre procesos relacionados.
  - *CGroup* - Información adicional sobre si ese servicio pertenece a un determinado grupo de servicios.

```
jfcaballero@eniatic:~$ systemctl status bluetooth.service
● bluetooth.service - Bluetooth service
   Loaded: loaded (/lib/systemd/system/bluetooth.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2021-09-23 09:04:59 CEST; 4h 14min ago
     Docs: man:bluetoothd(8)
    Main PID: 733 (bluetoothd)
     Status: "Running"
      Tasks: 1 (limit: 18736)
     Memory: 1.7M
    CGroup: /system.slice/bluetooth.service
            └─733 /usr/lib/bluetooth/bluetoothd
```

Para **iniciar, reiniciar, recargar o parar un servicio** se dispone de los siguientes comandos (esto no hace que se inicie automáticamente en el arranque):

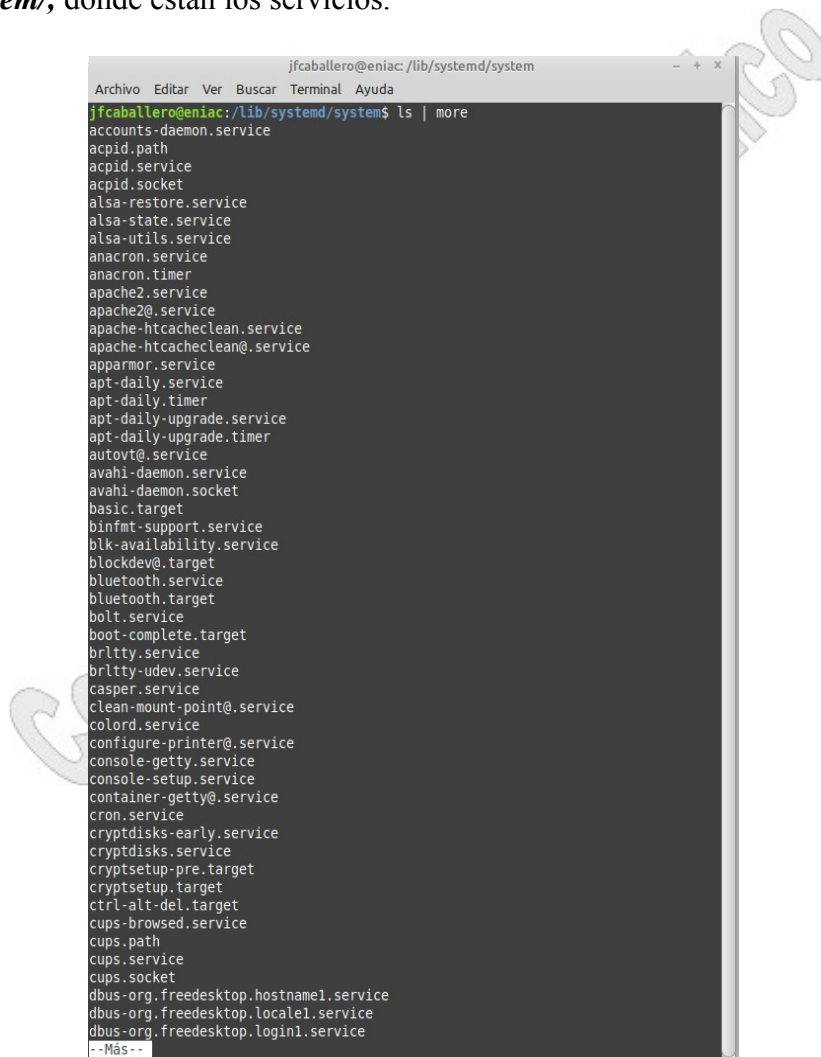
- ***# sudo systemctl start unit\_name.service***: Permite iniciar un servicio.
- ***# sudo systemctl restart unit\_name.service***: Permite reiniciar el servicio.

- **# `sudo systemctl reload unit_name.service`:** Permite recargar un servicio sin llegar a pararlo o reiniciarlo. Al ejecutar el comando se aplicarán los cambios realizados en los archivos de configuración del servicio sin tan siquiera llegar a parar el servicio.
- **# `sudo systemctl stop unit_name.service`:** Permite detener el servicio.

Para **activar demonios permanentemente en la secuencia de arranque** es necesario ejecutar el comando en modo *root*:

- **# `systemctl enable unit_name.service`,**

siendo *unit\_name* el nombre del servicio o unidad. En el momento de ejecutar este comando se creará un enlace simbólico en el directorio de configuración `/etc/systemd/system/` hacia el directorio `/lib/systemd/system/`, donde están los servicios.



```
jfcaballero@eniac: /lib/systemd/system
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:/lib/systemd/system$ ls | more
accounts-daemon.service
acpid.path
acpid.service
acpid.socket
alsa-restored.service
alsa-state.service
alsa-utils.service
anacron.service
anacron.timer
apache2.service
apache2@.service
apache-htcacheclean.service
apache-htcacheclean@.service
apparmor.service
apt-daily.service
apt-daily.timer
apt-daily-upgrade.service
apt-daily-upgrade.timer
autovt@.service
avahi-daemon.service
avahi-daemon.socket
basic.target
binfmt-support.service
blk-availability.service
blockdev@.target
bluetooth.service
bluetooth.target
bolt.service
boot-complete.target
brltty.service
brltty-udev.service
casper.service
clean-mount-point@.service
colord.service
configure-printer@.service
console-getty.service
console-setup.service
container-getty@.service
cron.service
cryptdisks-early.service
cryptdisks.service
cryptsetup-pre.target
cryptsetup.target
ctrl-alt-del.target
cups-browsed.service
cups.path
cups.service
cups.socket
dbus-org.freedesktop.hostname1.service
dbus-org.freedesktop.locale1.service
dbus-org.freedesktop.login1.service
--Más--
```

*Figura 1: Similar al directorio `/etc/init.d/` de Upstart*

```
jfcaballero@eniac: /etc/systemd/system
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:/etc/systemd/system$ ls -l
total 72
drwxr-xr-x 2 root root 4096 jul 3 2021 bluetooth.target.wants
lrwxrwxrwx 1 root root 42 ago 27 2021 dbus-fi.wl.wpa_supplicant1.service -> /lib/systemd/system/wpa_supplicant.service
lrwxrwxrwx 1 root root 37 ago 27 2021 dbus-org.bluez.service -> /lib/systemd/system/bluetooth.service
lrwxrwxrwx 1 root root 40 ago 27 2021 dbus-org.freedesktop.Avahi.service -> /lib/systemd/system/avahi-daemon.service
lrwxrwxrwx 1 root root 40 ago 27 2021 dbus-org.freedesktop.ModemManager1.service -> /lib/systemd/system/ModemManager.service
lrwxrwxrwx 1 root root 53 ago 27 2021 dbus-org.freedesktop.nm-dispatcher.service -> /lib/systemd/system/NetworkManager-dispatcher.service
lrwxrwxrwx 1 root root 44 ago 27 2021 dbus-org.freedesktop.resolve1.service -> /lib/systemd/system/systemd-resolved.service
lrwxrwxrwx 1 root root 36 ago 27 2021 dbus-org.freedesktop.thermald.service -> /lib/systemd/system/thermald.service
lrwxrwxrwx 1 root root 45 ago 27 2021 dbus-org.freedesktop.timesync1.service -> /lib/systemd/system/systemd-timesyncd.service
drwxr-xr-x 2 root root 4096 jul 3 2021 default.target.wants
lrwxrwxrwx 1 root root 35 ene 12 2022 display-manager.service -> /lib/systemd/system/lightdm.service
drwxr-xr-x 2 root root 4096 jul 3 2021 display-manager.service.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 emergency.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 final.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 getty.target.wants
drwxr-xr-x 2 root root 4096 ago 27 2021 graphical.target.wants
-rw-r--r-- 1 root root 649 jul 2 2021 hddtemp.service
drwxr-xr-x 2 root root 4096 jun 28 08:50 multi-user.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 network-online.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 oem-config.service.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 paths.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 printer.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 rescue.target.wants
drwxr-xr-x 2 root root 4096 ago 27 2021 sleep.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 sockets.target.wants
drwxr-xr-x 2 root root 4096 jul 3 2021 sysinit.target.wants
lrwxrwxrwx 1 root root 35 ago 27 2021 syslog.service -> /lib/systemd/system/rsyslog.service
drwxr-xr-x 2 root root 4096 ene 13 2022 timers.target.wants
jfcaballero@eniac:/etc/systemd/system$
```

Figura 2: Similar a los directorios `/etc/rc?.d/` de Upstart

También se dispone del comando para **eliminar el servicio de la secuencia de arranque**, que hará lo propio en el directorio `/etc/systemd/system/`:

- **# `systemctl disable unit_name.service`**

**Otros comandos** que pueden ser útiles para trabajar con servicios `systemd`:

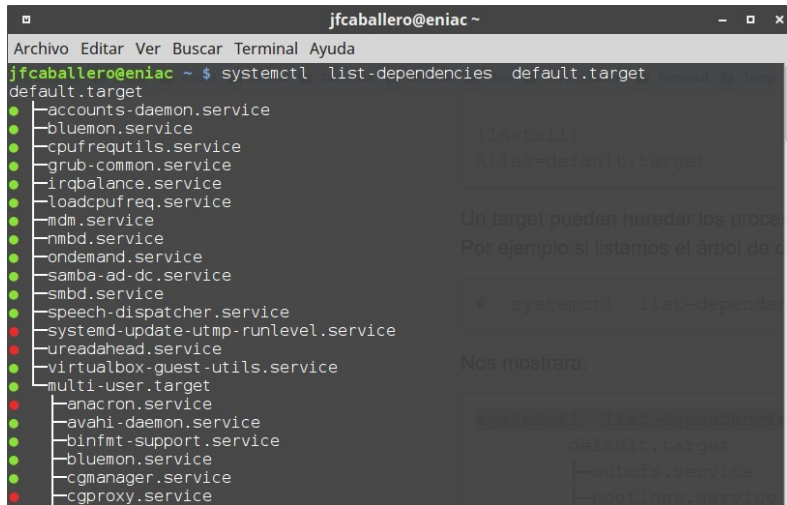
- **# `systemctl get-default`**: Indica la unidad `.target` o nivel actual del sistema.

```
jfcaballero@eniac: ~
Archivo Editar Ver Buscar Terminal Ayuda
jfcaballero@eniac:~$ systemctl get-default
graphical.target
jfcaballero@eniac:~$
```

- **# `sudo systemctl set-default unit_name.target`**: Establece a `unit_name.target` el nivel `.target` por defecto en el arranque del sistema.



- # **systemctl list-dependencies default.target** se pueden consultar las dependencias del arranque por defecto.



- **journalctl**: Permite ver el registro o log con la información que genera el núcleo en el arranque del sistema. Es similar a **dmesg** de **Upstart**.
- **journalctl -u unit\_name.service**: Permite ver el registro o *log*, generado por el servicio.

## Bibliografía

El contenido de este documento se ha elaborado principalmente a partir de las siguientes referencias bibliográficas, y con propósito meramente académico y no lucrativo:

- W. Stallings. *Sistemas operativos*, 5ª edición. Prentice Hall, Madrid, 2005.
- A. S. Tanenbaum. *Sistemas operativos modernos*, 3a edición. Prentice Hall, Madrid, 2009.
- A. Silberschatz, G. Gagne, P. B. Galvin. *Fundamentos de sistemas operativos*, séptima edición. McGraw-Hill, 2005.
- A. McIver, I. M. Flynn. *Sistemas operativos*, 6ª edición. Cengage Learning, 2011.
- J. A. Alamansa, M. A. Canto Diaz, J. M. de la Cruz García, S. Dormido Bencomo, C. Mañoso Hierro. *Sistemas operativos, teoría y problemas*. Editorial Sanz y Torres, S.L, 2002.
- F. Pérez, J. Carretero, F. García. *Problemas de sistemas operativos: de la base al diseño*, 2ª edición. McGraw-Hill. 2003.
- S. Candela, C. Rubén, A. Quesada, F. J. Santana, J. M. Santos. *Fundamentos de Sistemas Operativos, teoría y ejercicios resueltos*. Paraninfo, 2005.
- J. Aranda, M. A. Canto, J. M. de la Cruz, S. Dormido, C. Mañoso. *Sistemas Operativos: Teoría y problemas*. Sanz y Torres S.L, 2002.
- J. Carretero, F. García, P. de Miguel, F. Pérez, *Sistemas Operativos: Una visión aplicada*. Mc Graw Hill, 2001.