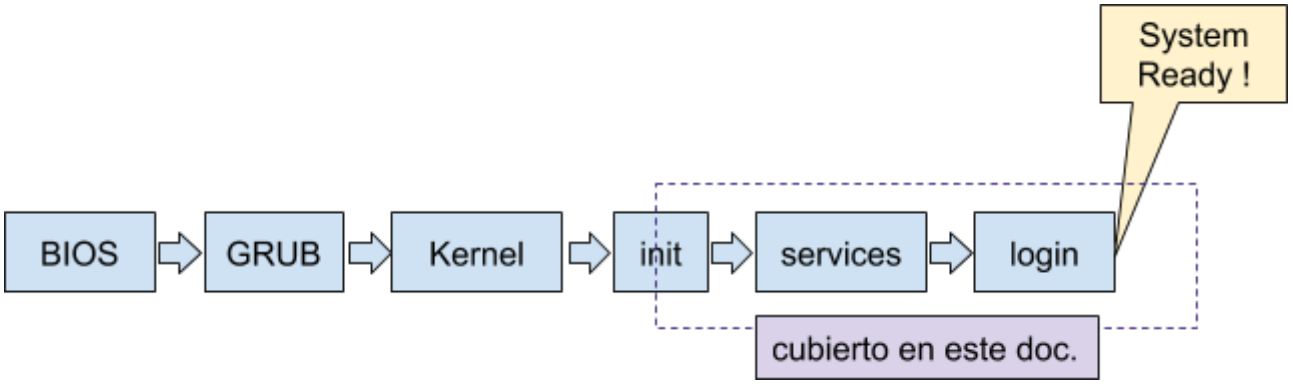
# Procesos servidores y arranque del sistema con **SystemD**

El objetivo inicial del documento es conocer el sistema de arranque **systemd** y hacerlo mediante una actividad práctica. Para ampliar el nivel de conocimientos del público de este documento, se retrocede atrás en muchos de los pilares necesarios para entender el objetivo inicial y se explican o revisan conceptos más básicos y preliminares. Se pretende conformar una "lección" amplia, impartible en más niveles de ciclos e incluso en diferentes módulos.

En este documento se describe el concepto de programa servidor y se realiza estudio de algunas características de dichos procesos. A continuación, se trata el arranque de un sistema desde que el núcleo del sistema operativo está totalmente listo y en ejecución, hasta que han sido inicializados todos los servicios (procesos servidores) requeridos para atender tanto al usuario como a peticiones de servicio realizadas por equipos remotos o dispositivos. El documento se centra en el arranque systemd. Se explican las "unit" como concepto primordial de este sistema de arranque y se continúa con los comandos capaces de iniciar, parar y diagnosticar servicios. Finalmente el alumno creará su propio servicio compilándolo desde cero y lo instalara bajo systemd de tres formas distintas: como servicio normal iniciable en el arranque, como servicio activado por socket y como servicio activado por ruta ("path"). Esta última actividad es la que más valor tiene y resulta entretenida.

El siguiente diagrama complementa la descripción del objetivo de este documento.




Conviene tener un sistema ubuntu/arch en funcionamiento para probar y experimentar todo lo que se realizará en este documento.

El tema está totalmente cubierto en un vídeo (¡ De 2 horas de duración !) en

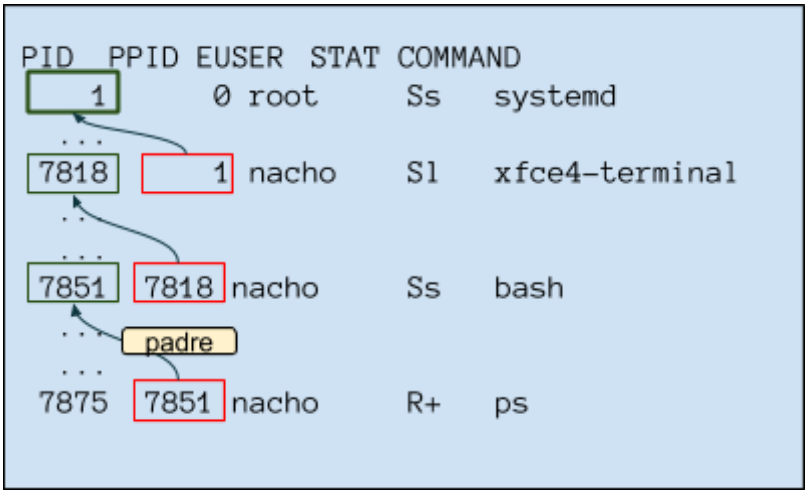
https://youtu.be/_HANHkHOXeM

# 1 Introducción

El siguiente comando "`ps -eo pid,ppid,euser,stat,comm`" muestra los procesos en ejecución y la información sobre:

- PID del proceso,
- PID del padre del proceso,
- usuario
- Estado del proceso
- línea de comandos que originó el proceso

Casi todos los atributos anteriores contribuyen a identificar y conocer a cada proceso en ejecución (es lo habitual al ejecutar el comando ps), pero el atributo del pid del padre del proceso es especial en esta ocasión. Gracias a él puedes identificar la secuencia de ancestros (padre, padre del padre, etc) de cada proceso (Como debes saber, cada proceso es originado a partir de otro proceso que conocemos como "padre"). A continuación puedes ver la salida del comando (con mucha información eliminada para identificar mejor el ejemplo).




El proceso ps lanzado (pid 7875) tiene como padre al proceso 7851 (que es bash), y éste tiene como padre a un terminal xfce (pid 7818), que por circunstancias especiales tiene como padre al proceso número 1… que es el primero y no tiene padre

Ejercicio:

Abre un terminal, y ejecuta el comando

```
ps -eo pid,ppid,euser,stat,comm
```

Identifica al proceso propio proceso ps que lanzas en la salida entregada por dicho comando. Sigue la pista de todos los procesos ancestros (observa cómo el padre de ps es otro proceso llamado bash) Anota los comandos de la secuencia de ancestros e identifica en qué momento aparece un padre cuyo usuario no es el usuario con el que lanzas el comando.

Cabe hacerse varias preguntas:

- ¿Qué  todos esos procesos? ¿Por qué están ahí?
- ¿Quién es el proceso inicial, el padre de todos? (ya está contestado)
- ¿Por qué al arrancar el computador se inicia o son creados?  ¿Dónde se configura eso?

Responderemos primero a la primera pregunta, pero si ya sabes cómo funciona linux y tan sólo quieres conocer systemd, sáltate la siguiente sección

Ejercicio:

De la lista de procesos obtenidos con el comando "ps aux"  descarta aquellos cuyo nombre esté encerrado entre corchetes ("[ ]").  Localiza algunos de los que incluyen las siguientes palabras en su nombre y obtén una descripción de su función.

NetworkManager, dbus (dbus-daemon), dnsmasq, dhcp, Xorg, rtkit, pulseAudio, rtkit, syslog,

# 2 Servicios y procesos demonio

Una manera de entender los procesos servidores de un ordenador, es observar algunas de las funciones que desempeñan ciertos computadores llamados servidores

- Servidor web, ftp, bbdd
- Servidor de impresión.
- Firewalls y routers.
- Servidores de autenticación, LDAP, Active Directory, etc.

Estos ordenadores están configurados para funcionar sin intervención humana, de forma que al ser iniciados se ejecuta una secuencia automática de acciones que activa el servicio que ofrecen. Estos servicios son simplemente procesos en ejecución.

Adicionalmente, en cualquier computador personal (sin necesidad de pensar en "superservidores") existen una serie de servicios y funciones que están disponibles también de forma automática.

- Pantalla de login o acreditación para que un usuario acceda a una sesión.
- Gestión de las Actualizaciones automáticas
- Servicio para reconocer y usar un pen USB
- Entorno gráfico
- Udev (atiende dispositivos)
- journald / syslog (registra eventos)

Todas estas funciones citadas son ofrecidas por procesos que no son activados o iniciados por ningún usuario, muchos no interactúan con él, e incluso vienen preinstalados en cualquier computador. Se trata de programas que son lanzados a ejecución por iniciativa del sistema siguiendo una configuración y un método de arranque que aquí estudiaremos. Visto de otra forma: cuando el usuario introduce su nombre y contraseña, ya existen varios procesos en marcha y de hecho, la pantalla de login misma es un proceso.

Nuestro objetivo en este tema es doble:

1. Entender Cómo funciona un servicio y las características que tienen
2. Cómo se inician y mantienen activos los servicios

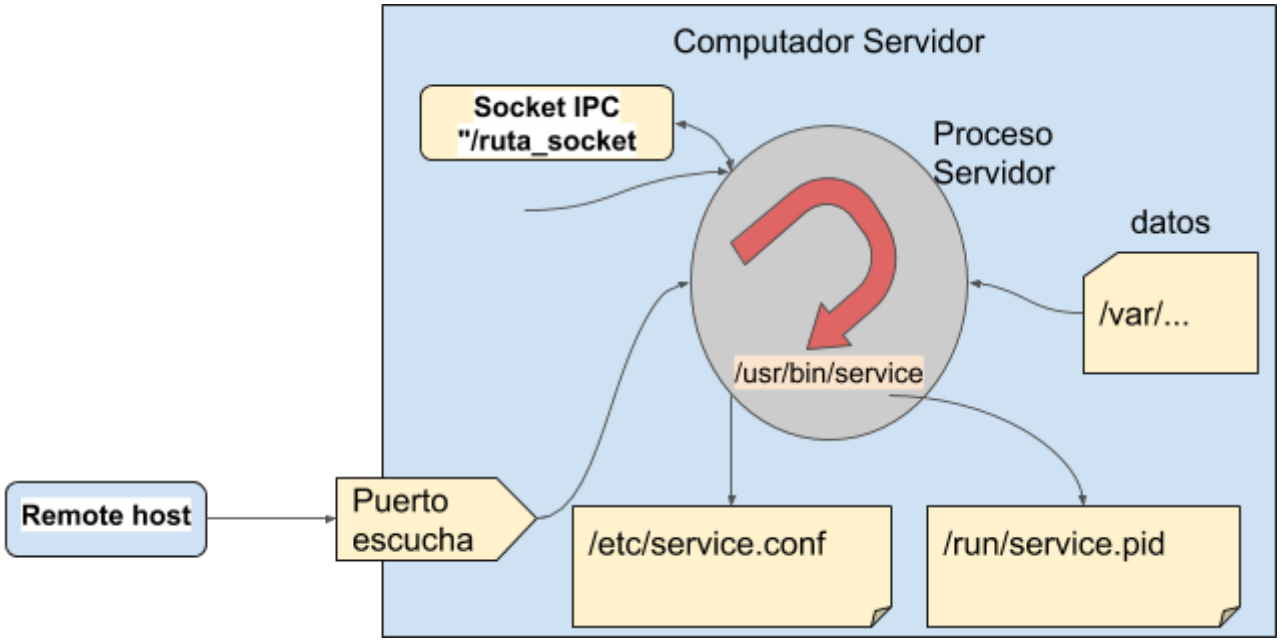
## 2.1 Elementos habituales de un servicio

Al observar el funcionamiento y elementos de los distintos servidores, descubrimos unas tendencias o prácticas comunes, similares en muchos de ellos (ojo! no estamos mirando *dentro* de los procesos servidores, cómo están programados o hechos, sino lo que observa el administrador del computador).

Estos elementos que prácticamente todos los servicios tienen son los siguientes:

| **Elemento** | **Ubicación** | **Función** |
|---|---|---|
| Ejecutables, bibliotecas y otros ficheros | /usr/bin<br>/usr/lib<br>/share | Son los **ficheros ejecutables y bibliotecas** que se depositan en esos directorios cuando se instala el servicio. |
| Proceso o demonio | memoria | Es el servicio en sí. Se trata de uno o varios **procesos en ejecución** que sostienen, ofrecen o implementan algún servicio. Si el servicio está activo es porque el proceso correspondiente está en ejecución |
| Fichero de Configuración | /etc/ | Generalmente todos los servicios tienen un fichero de configuración. La configuración se deposita en **/etc** siempre. En ocasiones existe también un directorio dentro de /etc. |
| Socket | Socket TCP<br>Socket AF_UNIX<br>DBus | Los servicios se comunican con procesos, dispositivos o computadores remotos. Se establecen canales de comunicación según el tipo de servicio. Estos canales de comunicación se conocen como sockets. |
| pid files | /run | Ficheros simples, cuyo contenido es el PID del proceso del servicio. Esto reporta algunos beneficios:<br>● Se puede identificar el servicio<br>● El mismo servicio puede comprobar si una instancia anterior terminó mal<br>● El administrador puede matar fácilmente el servicio con `kill -s SIGKILL` |

| | | |
|---|---|---|
| Ficheros de datos | /var<br>/srv | Algunos servicios están basados en gestionar datos. Por ejemplo, un servidor de BBDD o servidor WEB. Estos ficheros deben residir en algún directorio bien establecido por la configuración del servicio |
| .lock files | /run | En algunos casos, para evitar que por error se lancen dos procesos del mismo servicio, el primero en ser lanzado crea un fichero de cierre (lock) y el segundo, al intentar hacer lo mismo, se encuentra con el fichero ya creado. Esto le hace ver que ya está el servicio en marcha y por tanto el segundo cesa su ejecución |




Ejercicio (opcional y con probabilidad de fracaso):

1. Ve al directorio `/run` y lista los ficheros existentes fijándote en aquellos que tienen una extensión .pid o .lock. Asegúrate de que son ficheros regulares.
2. Elige uno de los ficheros `.pid` y observa el contenido. Su contenido debería ser un número correspondiente a un `PID`.
3. Usa el comando `ps + grep` para obtener una mejor descripción del proceso responsable de ese fichero.
4. Identifica el servicio que tiene ese proceso en el sistema e intenta encontrar su fichero de configuración en `/etc`

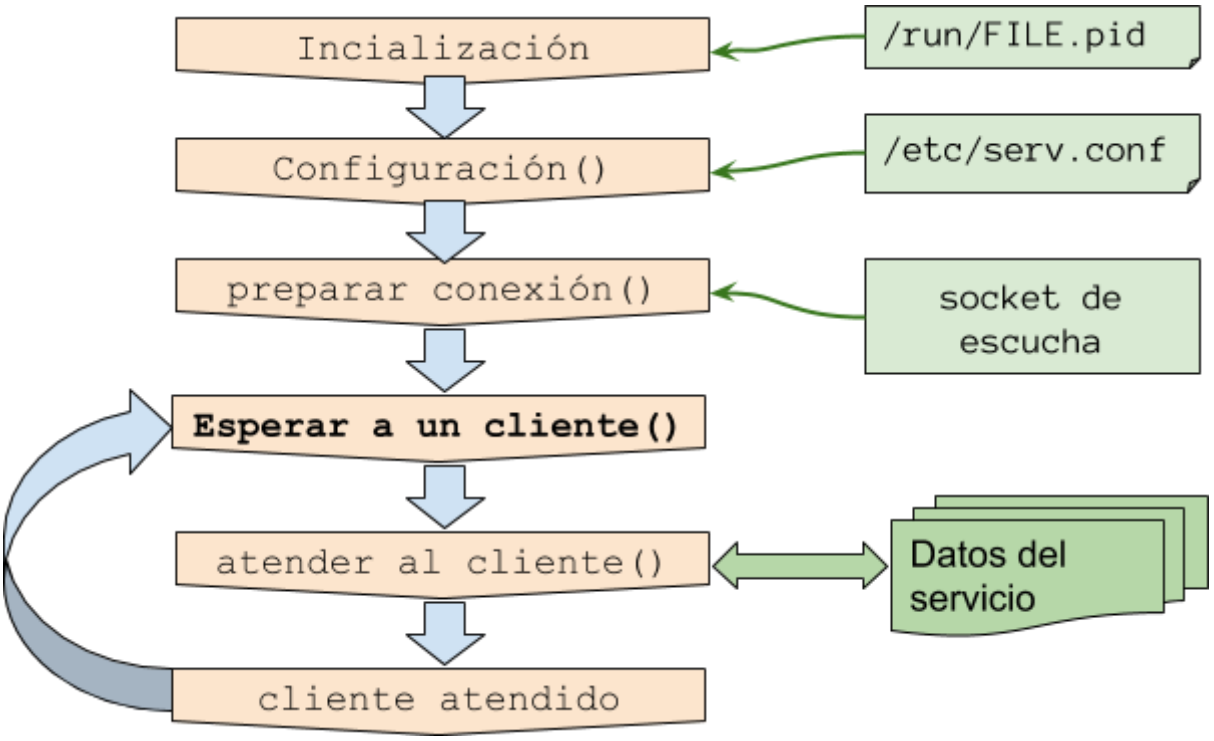
5. Localiza en `/usr/bin` el fichero ejecutable a partir del cual se ha creado el proceso.
6. Si el servicio tiene algún tipo de socket o puerto de escucha, intenta identificarlo (a partir de la documentación o de la configuración ; a veces en la config aparece el nombre del puerto o número

## 2.2 Funcionamiento interno de un proceso servidor.

Es imposible dar una explicación aproximada del funcionamiento interno de todos los procesos servidores, puesto que la naturaleza de los problemas que resuelven y las soluciones dadas son particulares de cada uno. Sin embargo, podemos elegir los procesos servidores que atienden peticiones en red (como un servidor web), y observar que la mayoría de ellos da los mismos pasos para establecer su funcionamiento y atender peticiones de los clientes. Esto nos bastará de ejemplo para el resto del tema.

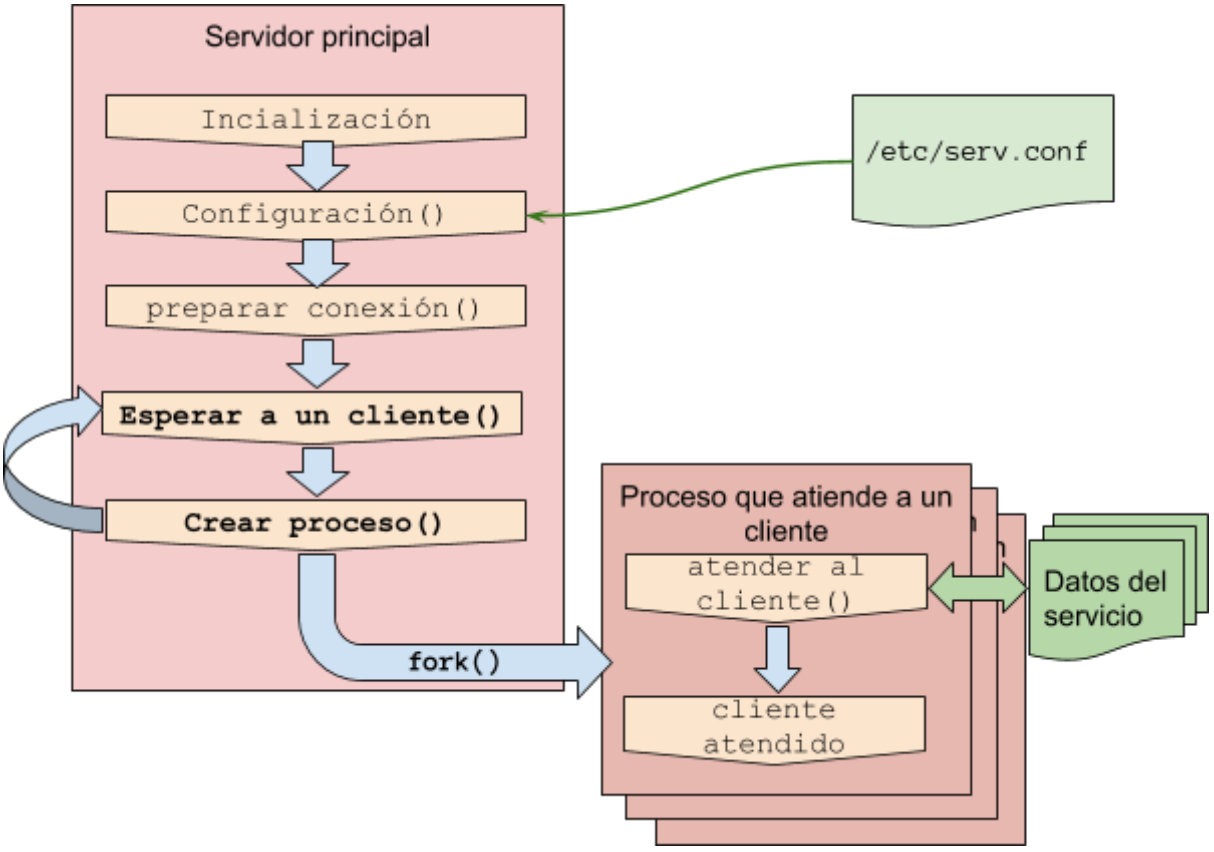
Inicialmente un servidor en red da los siguientes pasos:

1. Se inicia y lee la configuración establecida en algún fichero.
2. Prepara las conexiones en red que serán usadas para recibir solicitudes por parte del cliente
3. Espera a que llegue alguna petición de algún cliente
4. Atiende la petición llegada
5. Al finalizar, el punto anterior, se retorna al punto 3, de forma que esto forma un bucle infinito



Algunos servicios, son muy solicitados, y muchos clientes realizan muchas peticiones frecuentes. En estos casos, un esquema servidor como el anterior produce esperas intolerables (un cliente debe esperar a que terminen los servicios para peticiones previas). Para solventar esta situación, se desarrollan servidores "multiproceso", en los que el servidor principal crea procesos adicionales para atender a clientes.

De esta forma, el servicio principal, sólo se encarga de recibir peticiones, pero cada cliente es atendido por un proceso expresamente creado para él tan pronto como se recibe la petición.




**Práctica con procesos servidores.**

En este ejercicio vamos a crear nuestro propio servidor, ponerlo en marcha, hacerle peticiones y observar su respuesta. Este servidor será utilizado a lo largo del tema.

Necesitarás:

- Compilar programas en C (instala `build-essentials` si hace falta)
- Utilizar el programa `netcat` (versión "netcat" o "nc"). Instálalo si no dispones de él. Opcionalmente ( y sólo si no tienes netcat) podrías utilizar un cliente telnet o ssh, o el comando:

```
printf "" | ( exec 3<> /dev/tcp/localhost/5000; cat >&3; cat <&3;
exec 3<&- )
```

- Utilizar dos terminales, uno para lanzar el servidor y observar algún mensaje de depuración y el otro terminal donde lanzarás el comando **`netcat`** haciendo de cliente
    - Evidentemente, podrías ejecutar el cliente en otro computador, máquina virtual, contenedor, etc, pero no es plan de ir complicándonos la vida demasiado. Podemos trabajar en el mismo computador sin perder el sentido de lo que estamos haciendo.

Pasos:

1. Ve al anexo y copia el código de programa servHora.cc a un fichero llamado servHora.cc. Guarda el fichero
2. Compila el fichero fichero servHora.cc con la siguiente línea de comandos

    ```
    gcc -o servHora servHora.cc
    ```

3. El paso anterior obtiene un ejecutable llamado "**`servHora`**". Ejecuta este programa con la siguiente línea de comandos:

    ```
    ./servHora
    ```

    Este comando habrá dejado al terminal a la espera de finalizar el comando
4. Verifica en otro terminal dentro de la misma máquina que ese proceso está atendiendo en el puerto elegido (5000 en el ejemplo entregad)

    ```
    sudo ss -tulpn  | grep servHora
    ```

5. En el paso anterior has lanzado un ejecutable que hace las funciones de servidor. Abre ahora otro terminal y ejecuta un cliente

    ```
    netcat localhost 5000
    ```

    Donde "**`localhost`**" es el computador actual y **5000** es el puerto que está en el programa que has compilado.  Si deseas, haz esto desde otro ordenador y sustituye "localhost" por la ip del ordenador donde has lanzado el servicio
6. Si todo va bien, observa que el servidor envía la fecha al cliente que la muestra en el terminal (el servidor también muestra la fecha en el terminal, pero esto es un mero mensaje de depuración)

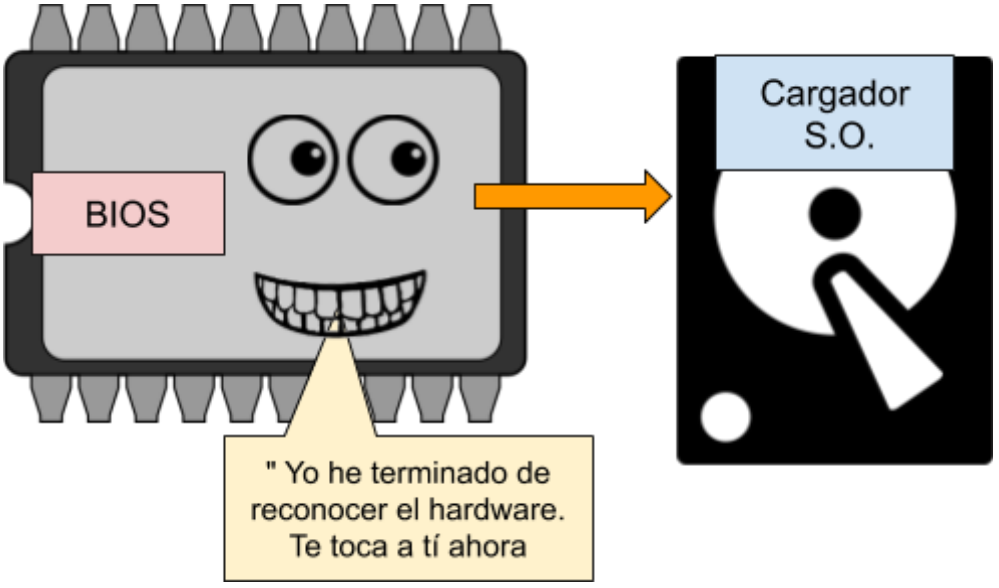
## 2.3 Arranque en sistemas Linux modernos

El sistema de arranque moderno de Linux quiere conocer y controlar todos esos elementos del servicio para cumplir varios objetivos:

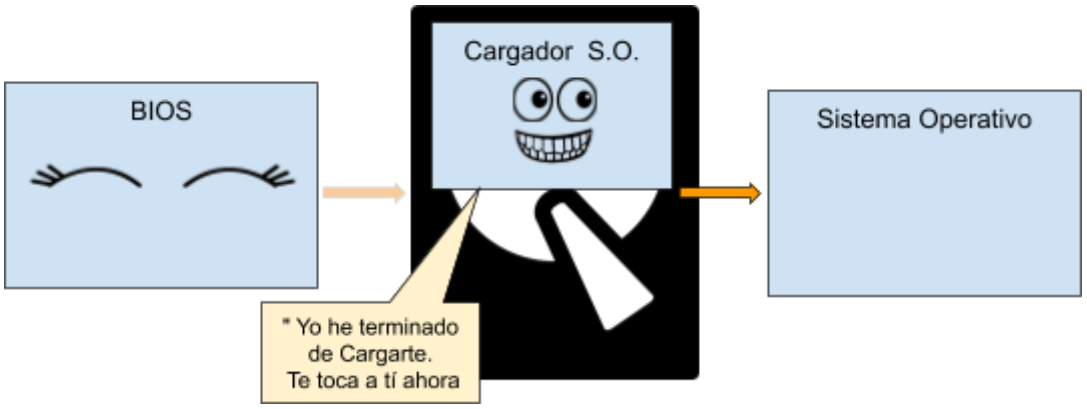
- Facilitar el **correcto inicio del servicio**, asegurándose que todo lo que requiere cierto servicio está previamente disponible y en correcto estado
    - Por ejemplo, si un servicio depende de conexiones de red, entonces el sistema de arranque inicia la red y valida la configuración antes de iniciar el servicio
    - Se establece una **secuencia de dependencias** entre servicios que debe ser conocida.
    - Evitar que algún servicio, por un funcionamiento anómalo, consuma **excesivos recursos** y degrade el rendimiento general del computador.
- Controlar el **estado del servicio** y en caso de error, tomar alguna medida (típicamente relanzar el servicio)
- Controlar el destino de **mensajes de error, depuración y funcionamiento** para que el administrador descubra el origen de errores y problemas.

# 3 Arranque del sistema

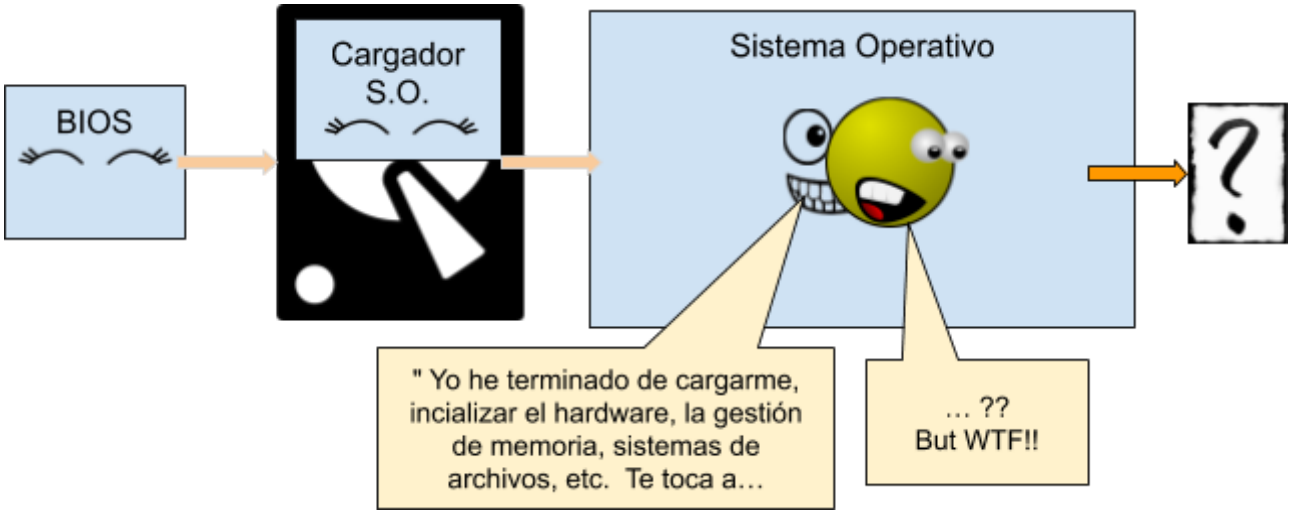
Es sabido que el arranque de una máquina es una secuencia de etapas en las que se ejecutan diferentes partes. Así, al encender un computador, se ejecuta el programa establecido en su "BIOS", y a partir de ahí, ocurren una serie de fases hasta que el usuario finalmente utiliza el programa que desea.

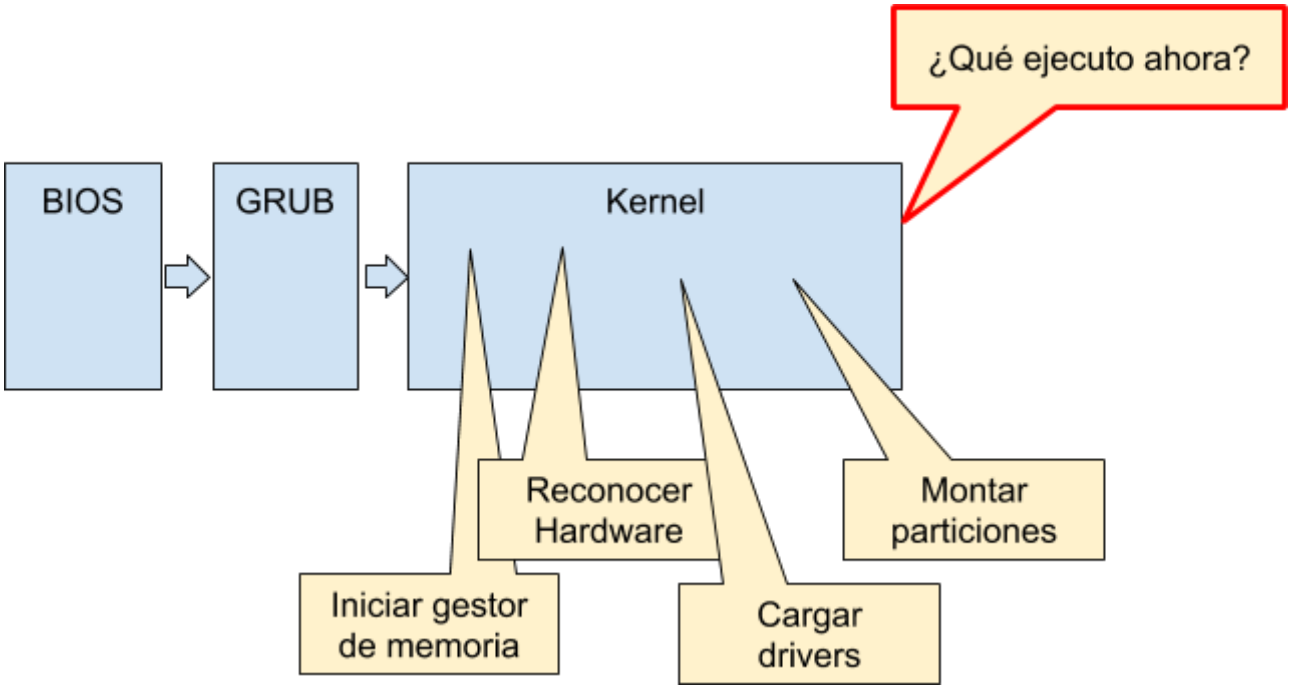



Después de que el cargador se ejecute, el sistema operativo estará cargado y listo para ejecutarse y tomar el control de la máquina.



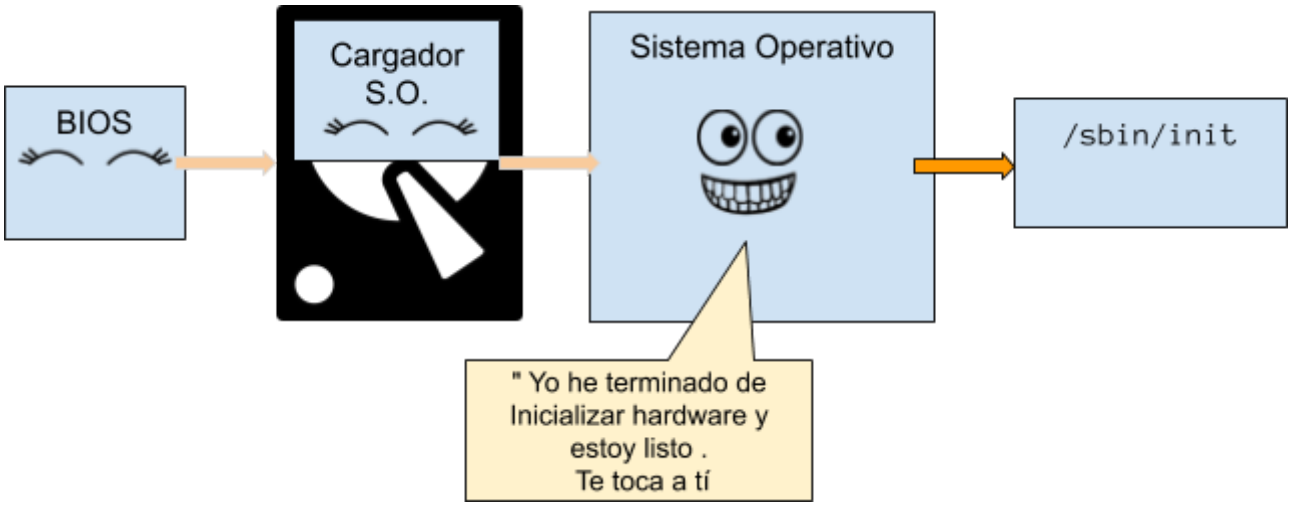

Nuestro interés ahora se centra en el momento en el que el núcleo del sistema operativo termina su completa inicialización, pero todavía no se está ejecutando ningún proceso



Piensa en que Linux y sus distribuciones es un sistema modular, hecho por diferentes equipos de programación que proporcionan distintas partes del sistema. El equipo de desarrollo del núcleo no tiene responsabilidad alguna en todo lo que no sea estrictamente el funcionamiento del núcleo y el resto de equipos no participan activamente del desarrollo del núcleo. Por ello, en este punto se debe llegar a un acuerdo entre todos, fácil, elegante y que permita la máxima flexibilidad y . El convenio es el siguiente:

> Sea cual sea la distribución de linux, el núcleo de linux,
> al terminar su carga e inicialización ejecutará el fichero
> **`/sbin/init`**




Este fichero siempre está presente y siempre se trata de un ejecutable… (aunque puede ser un enlace simbólico como suele ser en las distribuciones de tipo arch o ubuntu). Fíjate en la siguiente captura de pantalla, estando en la carpeta /sbin observa a dónde apunta el fichero "init" (en lubuntu)

```
nacho@base-lubuntu:/sbin$ ls -l init
lrwxrwxrwx 1 root root 20 sep  5 13:01 init -> /lib/systemd/systemd
nacho@base-lubuntu:/sbin$
```

Lo que hace el ejecutable es arrancar al resto de servicios necesarios para poner en pie totalmente el equipo

Ejercicio:

Intenta razonar qué de pasos y acciones que deberá dar el proceso init u otros procesos en el arranque para dejar el computador listo para ser usado por un usuario.

Existen varios sistemas de inicialización, siendo tradicional durante mucho tiempo SysVInit. Pero debido a sus limitaciones han aparecido sistemas alternativos para inicializar los servicios:

- SysVInit: El original de muchos linux. Basado en ejecutar scripts de una carpeta concreta.
- Systemd: Un sistema moderno, muy potente y versátil que se ha implantado recientemente con éxito.
- Upstart: Propio de Ubuntu hasta que se reemplazó por Systemd
- OpenRC: Un arranque muy compatible con SysVInit
- runit: Un sistema alternativo bastante simple. Es parte de la mejor distro del mundo mundial: "void linux"

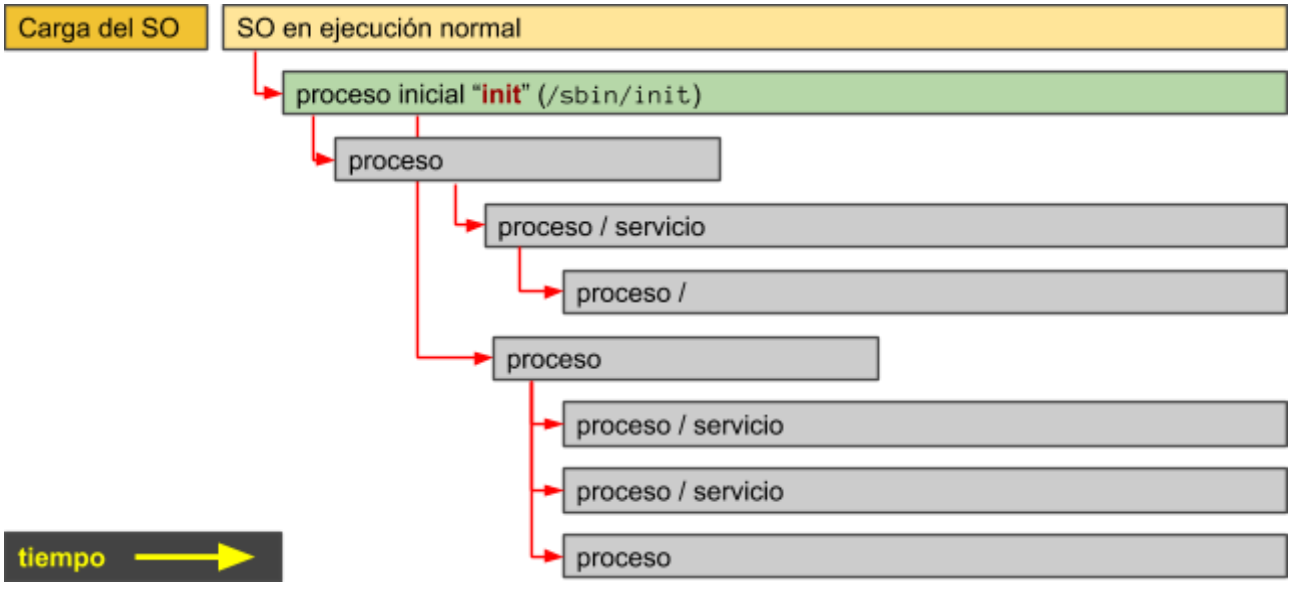
De todos ellos, **Systemd** es el que se está imponiendo en la mayoría de distribuciones y reemplazando a SysVInit. Nuestro objetivo para este tema será conocer el funcionamiento de systemd y realizar una actividad práctica con él.

# 4 Systemd

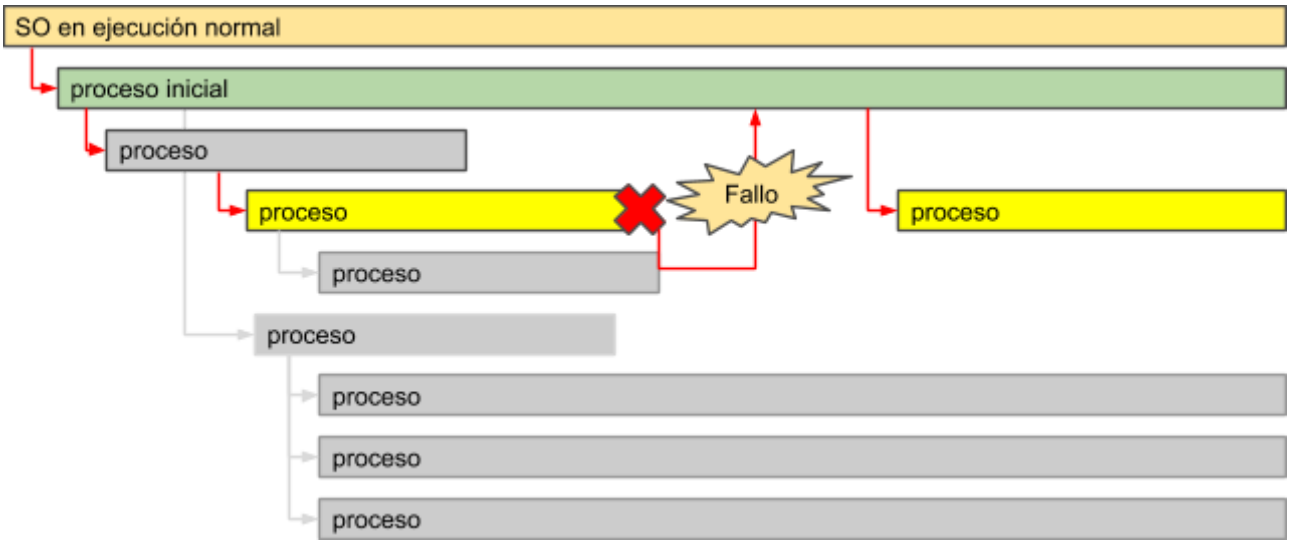
## 4.1 Objetivos de systemd:

- Arrancar los servicios
- Vigilar el estado de los servicios y reiniciarlos si caen
- Responder ante ciertos eventos lanzando servicios cuando ocurren.

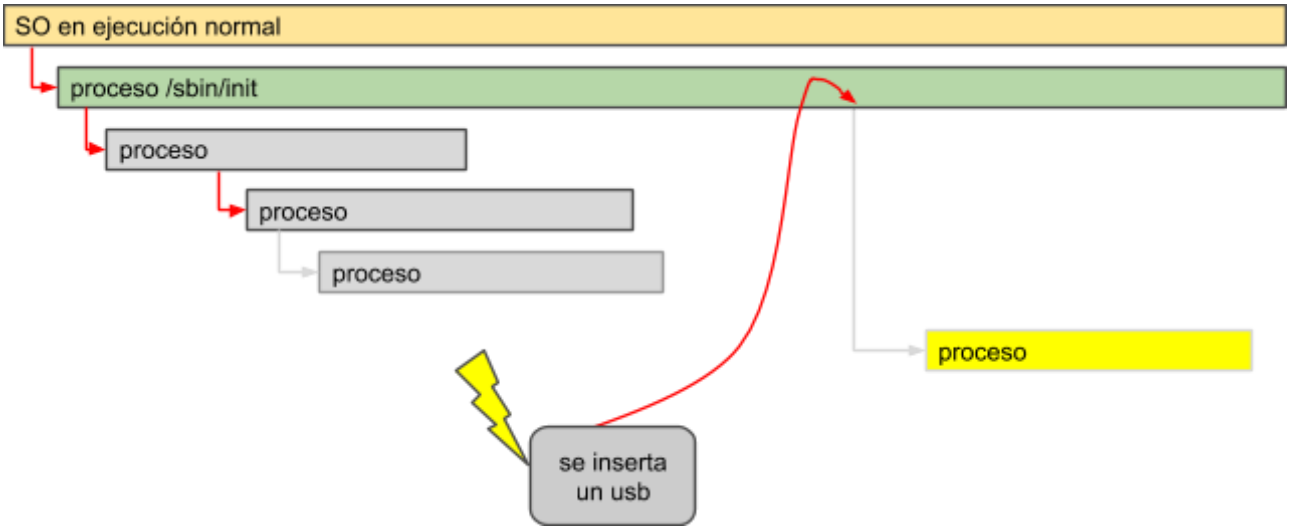
Arrancar los servicios ocurre al  principio siempre durante un arranque normal




Es posible que algún servicio esencial o permanente falle. Systemd lo reiniciará




Systemd también puede intervenir cuando ocurren algún evento que requiere la puesta en marcha de algún proceso.




## 4.2 Units

4.3 Los elementos más importantes para conocer systemd son las "unit". vamos a estudiarlas desde distintos puntos de vista.

Una **unit es un fichero de configuración** que controla algún aspecto relevante para el funcionamiento y arranque de los servicios. Entenderemos mejor esta idea viendo los tipos de unit existentes

### Tipos de units

1. **Service** units, which start and control daemons and the processes they consist of.

2. **Socket** units, which encapsulate local IPC or network sockets in the system, useful for socket-based activation.
3. **Target** units are useful to group units, or provide well-known synchronization points during boot-up.
4. **Device** units expose kernel devices in systemd and may be used to implement device-based activation.
5. **Mount** units control mount points in the file system.
6. **Automount** units provide automount capabilities, for on-demand mounting of file systems as well as parallelized boot-up.
7. **Snapshot** units can be used to temporarily save the state of the set of systemd units, which later may be restored by activating the saved snapshot unit.
8. **Timer** units are useful for triggering activation of other units based on timers.
9. **Swap** units are very similar to mount units and encapsulate memory swap partitions or files of the operating system.
10. **Path** units may be used to activate other services when file system objects change or are modified.
11. **Slice** units may be used to group units which manage system processes (such as service and scope units) in a hierarchical tree for resource management purposes..
12. **Scope** units are similar to service units, but manage foreign processes instead of starting them as well.

En resumen:

*"Algunos servicios (unit.service) se ejecutan durante el inicio automáticamente (unit.target), o bien cuando se establece una conexión mediante socket (unit.socket). Es posible que el servicio dependa de unos datos montados en alguna carpeta (unit.mount) o que dependa de algún dispositivo (unit.device). En ocasiones un servicio se activa cuando aparece un fichero en alguna ruta especial (unit.path), o lo hace periódicamente (unit.timer). Algunos servicios se agrupan (unit.slice) formando un grupo de procesos especial".*

## 4.4 Units de servicio

Son las unit que describen aspectos del **funcionamiento de un servicio**. En concreto se pueden especificar las siguientes informaciones:

- Comando a ejecutar para arrancar el servicio
- Comando para detener el servicio
- Dependencias previas
- Servicios que dependen de esta unit (creo)
- Destino de la salida de error o estándar.

## 4.5 Anatomía de un fichero unit.service

Las  unit de tipo service son las más importantes y habituales, es conveniente conocer sus partes. Veámos un ejemplo y después lo comentamos

```
[Unit]
Description=Foo

[Service]
ExecStart=/usr/sbin/foo-daemon

[Install]
WantedBy=multi-user.target
```

Secciones

1. `[Unit]` Aquí se describen las características generales de esta unit y también las dependencias de esta unit con otras. Observa por ejemplo la sección [Unit] de Networkmanger.service (lubuntu)

```
[Unit]
Description=Network Manager
Documentation=man:NetworkManager(8)
Wants=network.target
After=network-pre.target dbus.service
Before=network.target
```

2. `[Service]` Esta sección es exclusiva de las unit de tipo "service". Aquí se indica como iniciar y parar el servicio, qué hacer si falla y el tipo de ejecución que lo caracteriza

```
[Service]
Type=oneshot
Environment=LVM_SUPPRESS_LOCKING_FAILURE_MESSAGES=1
ExecStart=/sbin/lvm vgchange --monitor y --ignoreskippedcluster
ExecStop=/sbin/lvm vgchange --monitor n --ignoreskippedcluster
RemainAfterExit=yes
```

3. `[Install]` En esta sección se indica bajo qué target debe estar instalada esta unit. (a continuación se trata este punto)

```
[Install]
WantedBy=multi-user.target
```

## 4.6 Targets

"Target" En inglés significa 'objetivo', 'meta'. Aquí es sencillamente un **estado a alcanzar**; una situación en la que el sistema se considera en correcto funcionamiento. Y se traduce en que unos servicios (y otras units) estén activos y cada uno en estado correcto. Básicamente….

Target = conjunto de units activas

Normalmente en un sistema existe un target principal a alcanzar llamado "`multi-user.target`". Éste requiere que otros targets secundarios se cumplan y además se activen ciertos servicios.

Cada target se manifiesta como una subcarpeta dentro de /etc/systemd/system, cuyo nombre es "nombreTarget.target.wants". En dicha carpeta aparecen las otras unit necesarias para ésta

Ejercicio:

Vamos a tener una primera toma de contacto con el target "multi-user.target". Para ello:

1. Ve al fichero "/usr/lib/systemd/system/mult-user.target e identifica en la sección "After" las otras target que se requieren para que ésta se cumplan.
2. Ve a la carpeta "/etc/systemd/system/multi-user.target.wants e identifica todas las otras unit que deben activarse para que este target esté activo

Vamos a tener una primera toma de contacto con units de tipo service

3. En la carpeta anterior, localiza algunos ficheros .service, examina el contenido y comprueba que aparecen las secciones descritas anteriormente de "unit", "service" e "install".
4. Verifica que la sección install indica que "multi-user.target" es la target deseada para la instalación.

A partir de aquí deducimos que para que un servicio esté activo, la unit del mismo debe aparecer dentro del directorio de dependencias de una unit.

# 5 Directorios relevantes

Todas las units se almacenan en un único repositorio del sistema que está en la carpet.

`/lib/systemd/system`

Pero esta carpeta es tan sólo un repositorio que se crea al instalar linux y deja units que se usan y otras que no se usan (pero podrían llegar a usarse si cambia la configuración). Aquí no se

especifica qué servicios se inician automáticamente, qué relación existe entre ellos, etc. Toda la configuración de arranque, targets, etc que se aplica reside en el directorio

```
/etc/systemd/system
```

En el directorio anterior, existen diferentes subdirectorios correspondientes a las targets. En todos ellos,  se colocan enlaces a los ficheros reales que están en el primer directorio.

## 5.1 Operando systemd

Consultar units en marcha

```
$ systemctl
```

Consultar el estado del sistema

```
$ systemctl status
```

Consultar units de cierto tipo

```
$ systemctl -t service    (service, device, mount….)
```

Consultar estado de una unit

```
$ systemctl status unit.tipo
$ systemctl status avahi.service      o   systemctl status avahi
```

Descubrir el fichero unit y su contenido de una unit activa

```
$ systemctl cat unit.tipo
```

(las unit de tipo device n tienen fichero asociado se crean automáticamente por systemd

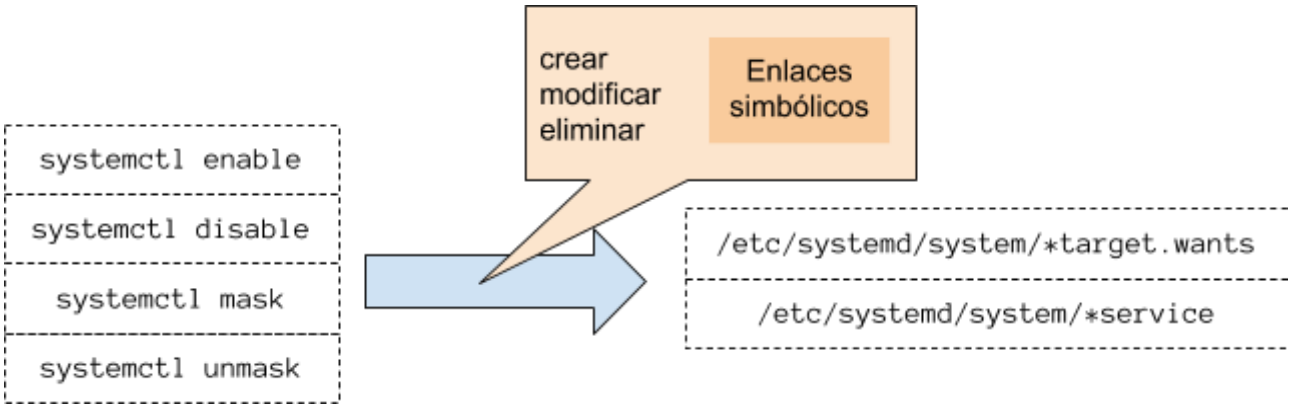
Ejercicio:
Practica los anteriores comandos en el siguiente orden:

1. Examina el estado del sistema
2. Examina la lista de las units activas.
    a. Elige una unit de cada tipo y verifica su estado
    b. Elige una unit y mediante systemctl cat muestra el fichero donde se configura.,

## 5.2 Instalación y activación  de units

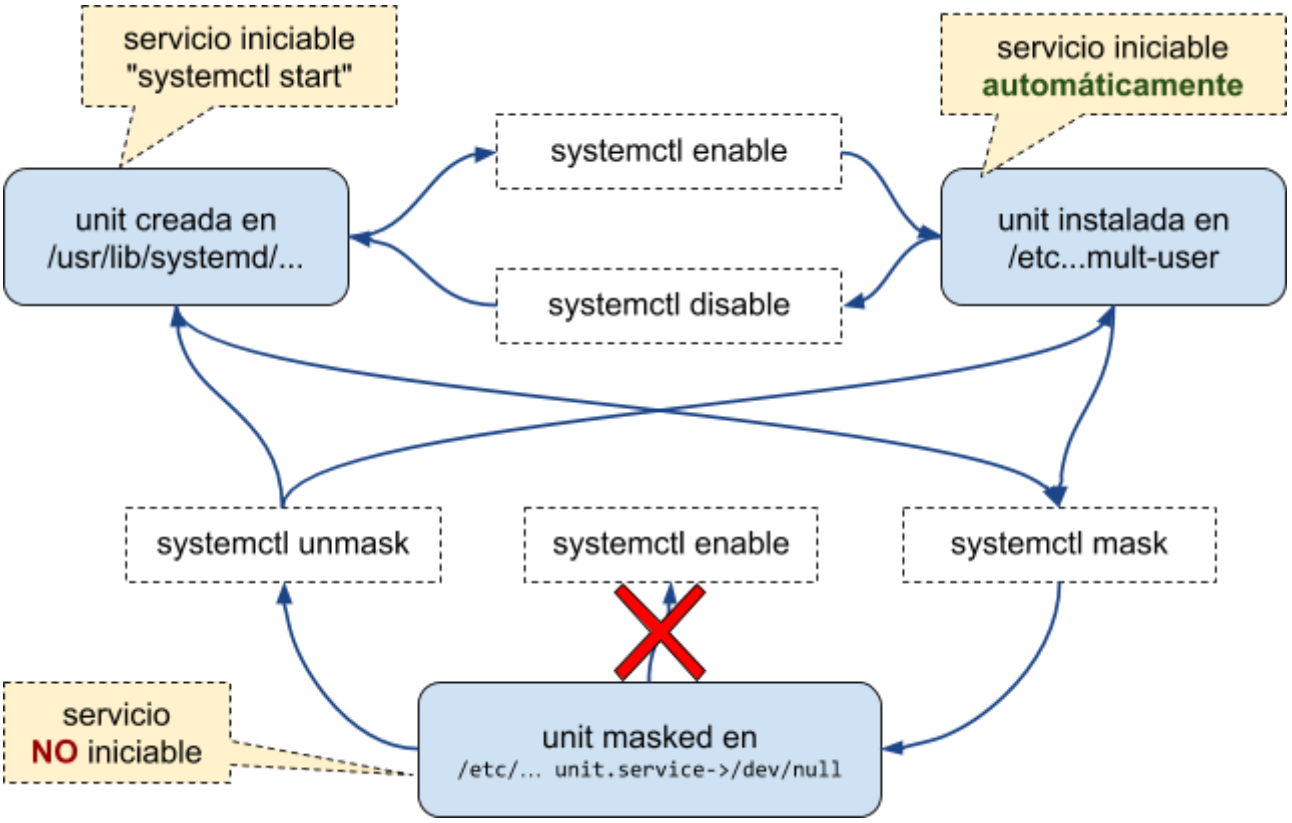
Instalar una unit se reduce a simplemente dejar caer el fichero de la unit en el repostorio /usr/lib/systemd/system

La activación/desactivación de units para que se configure su inicio automático se realiza mediante comandos cuyos efectos son crear, modificar o eliminar ciertos enlaces simbólicos en las carpetas vistas hasta ahora.




Las unit estarán configuradas como habilitadas o deshabilitadas e instaladas o no instaladas. Cada estado tiene un comando para que una unit sea establecida así y un significado:

- **No instalada** (not enabled). comando: "**systemctl disable**" . Es el estado inicial de una unit al ser instalado un servicio y su unit se preconfigura en el repositorio /usr/lib/systemd/system. Una unit así simplemente no se activará al arrancar el ordenador y no está ligada a ningún target.
- **Instalada** (enabled), comando "**systemctl enable**". La unit se configura para ser lanzada junto con un target (o el inicio , si el target es "multi-user"). Lo que esto causa es que se cree un enlace en la carpeta de la target que indica la sección install de la unit del servicio (`/etc/systemd/system/xxxx.target.wants`).
- **masked** / **unmasked**. Una unit no instalada puede ser lanzada por varios motivos. A veces se desea que en ningún caso se pueda lanzar un servicio. Para ello una unit se enmascara "**systemctl mask**". Se crea un enlace simbólico en /etc/systemd/system con el nombre de la unit apuntando a /dev/null y así se "fastidia" cualquier posible arranque. Cuando se quiere levantar tal prohibición sobre la unit, se desenmascara con "**systemctl unmask**"



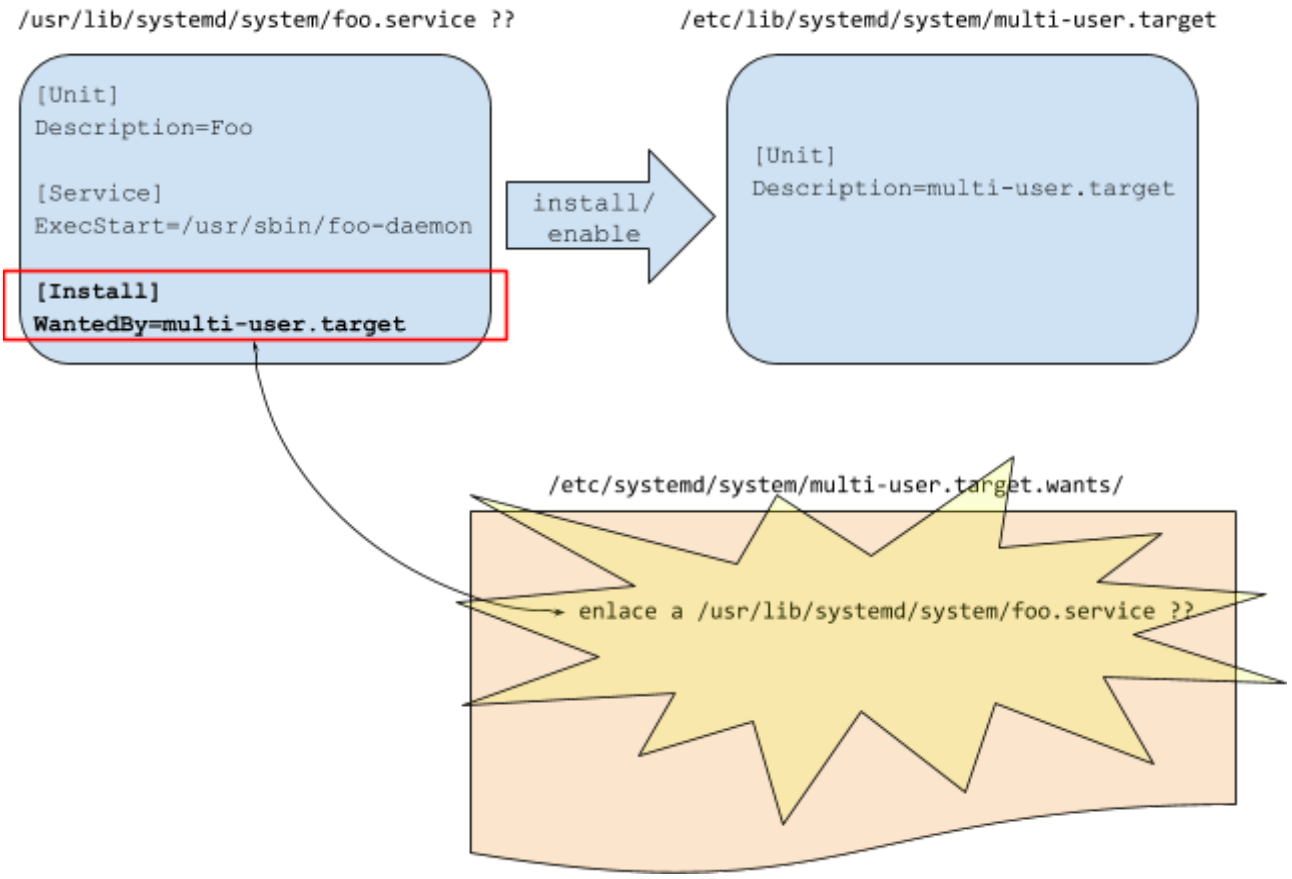

Independientemente de si una unit está enabled o no, se puede activar manualmente… salvo si está masked, en cuyo caso no la arranca ni chuk norris (pero sí charles bronson)

más sobre habilitar una unit. Ejemplo

```
$systemctl enable unit.type
```

Se crearán enlaces simbólicos en directorios específicos, siguiendo la información de la sección [install] (es decir, que la unit anterior tendría un enlace en /etc/systemd/system/multi-user.target.wants)

La habilitación no arranca el servicio de una unit.service




Enmascarar

```
$systemctl mask unit.type
```

Se crean enlaces simbólicos en /etc/systemd/system que apuntan a `/dev/null`. . el efecto es e que pasa a ser imposible activar estas units, ni siquiera manualmente. Es decir: si una unit aparece en `/etc/systemd/system` como enlace simbólico apuntando a `/dev/null`, resulta **imposible arrancarla**.

La operación

```
$systemctl unmask unit.type
```

Elimina el enlace simbólico que apunta a **`/dev/null`** y el recurso asociado a la unit puede ser usado con normalidad.

## Units sin instalación (ejercicio)

Considera una unit que no necesita asociarse a ningún target en particular, no parece que tenga sección Install. Por ejemplo la siguiente unit

```
[Unit]
Description=Unit para reaccionar a la insercion de un usb

[Service]
ExecStart=/bin/bash /usr/bin/usb-script.sh

[Install]
```

Se copiará a /lib/systemd/system (como usb.service) y ya tendremos la unit lista. Se podrá activar y parar manualmente con systemctl start|stop usb.service.

# 6 Ejercicio práctico de uso de systemd

El ejercicio práctico va a consistir en en instalar un servicio partiendo de su código fuente y realizar todos los pasos para que arranque automáticamente usando las funciones de systemd.

En primer lugar, habrá que compilar el código fuente (si no lo has hecho ya) y obtener un ejecutable que usaremos como parte importante del servicio. Ve al anexo y sigue las instrucciones en el punto "servidor hora multiproceso"

## 6.1 Instalación de una unit

Recuerda que la carpeta /usr/lib/systemd/system contiene un repositorio de units. El primer paso es crear la unit ahí.

## 6.2 Instalación a partir de una unit ".service" sólo

fichero **servHoraMult.service .** Se copia a la carpeta /usr/lib/systemd/system

```
[Unit]
      Description=Servidor De la hora

[Service]
      Type=exec
      ExecStart=/usr/bin/servHoraMult

[Install]
      WantedBy=multi-user.target
```

Ahora, Ya podemos iniciar el servicio con "systemctl start". Pero la parte interesante es instalarlo con "systemctl enable". Si lo haces, fíjate cómo avisa de que se crea un enlace simbólico en la carpeta /etc/systemd/systemd

```
nacho@base-lubuntu:/etc/systemd/system/multi-user.target.wants$ systemctl enable servHoraMult.service
Created symlink /etc/systemd/system/multi-user.target.wants/servHoraMult.service → /lib/systemd/system/servH
oraMult.service.
nacho@base-lubuntu:/etc/systemd/system/multi-user.target.wants$
```

Cuando inicies el equipo debería arrancar ese servicio. Podrás verficarlo con

```
systemctl status servHoraMult.service
```

La prueba de que funciona es hacer -como ya hemos hecho- un

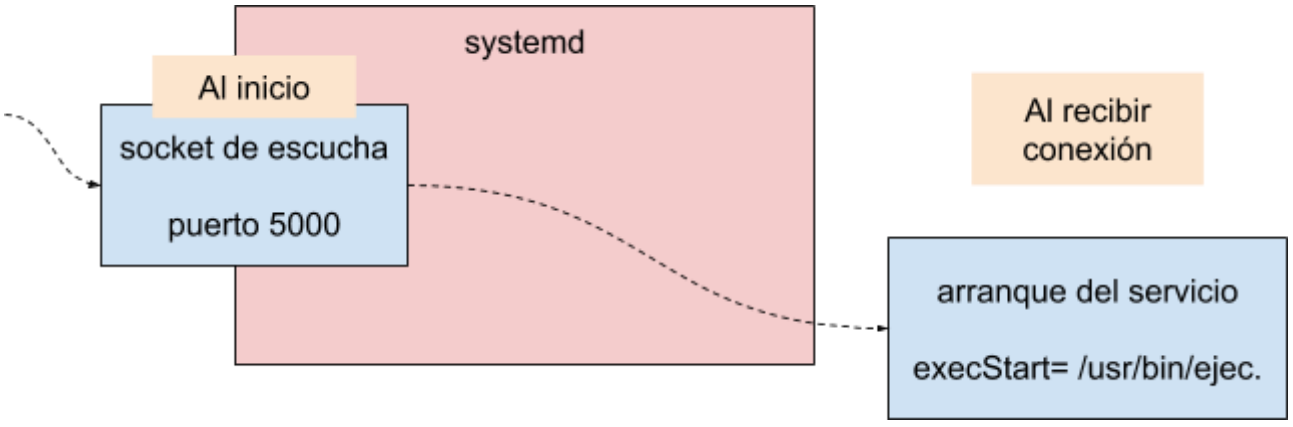
```
netcat localhost 5000
```

Esta instalación de un servicio con systemd es la más normal y típica, pero no es la única manera de automatizar el inicio de un servicio con systemd.

## 6.3 Instalar el servicio a partir de un socket

Alternativamente a la instalación anterior, Un servicio lanzado a partir de conexiones de socket puede ser establecido mediante una combinación de unit socket + service

La idea es que systemd se queda escuchando en un puerto y cuando recibe una conexión, arranca el servicio para atenderla, pero evita arrancar el servicio inútilmente desde el inicio. Con ello se acelera el arranque del sistema.




Systemd, al establecer la conexión, arrancará el ejecutable y le pasa el descriptor del socket ya abierto y listo para intercambiar información. Esto marca una diferencia en cómo está el servicio programado. Con este método no sirve una implementación tradicional como la hecha para el caso anterior (más sobre esto, después)

Este método está basado en dos units que trabajan coordinadamente. El nombre de ambas unit debe ser acorde para que todo funcione.


primero creamos la unit tipo socket. Por ejemplo `servHoraMult.socket`

```
[Unit]
Description=Socket para el servicio del a hora
[Socket]
ListenStream=127.0.0.1:5000
Accept=yes
[Install]
WantedBy=sockets.target
```

Esta unit se instala con

```
systemctl enable servHoraMult.socket
```

Al instalar la unit tipol socket, se establece que el servicio dependerá inicialmente de que sytemd escuche por el socket (puerto) y como consecuencia de una conexión lance el servicio buscando la plantilla de servicio llamada casi igual : **`servHoraMult@.service`**

nótese el @ en el nombre.

```
[Unit]
      Description=Servidor De la hora
      #Requires=servHora.socket

[Service]
      Type=simple
      ExecStart=/home/nacho/ejecutable
      #StandardInput=socket
      #TimeoutStopSec=5

[Install]
      WantedBy=multi-user.target
```

No hay entonces que proporcionar una unit .service normal, pues el servicio se activa a partir del socket. De ahí el símbolo @ en el nombre de la unit

tan sólo hay que enable ( + start) la unit tipo socket. Y ésta, cuando llegue una conexión activará dinámicamente una unit a partir del fichero servHoraMult@.service

Cuando se lance una petición se debe crear una unit transient esto se puede corroborar haciendo "systemctl status servHoraMu… " y solicitar la ayuda del autocompletado, que mostraraá una unit transient, creada para la petición. En ella ser verá el puerto creado por cliente y servidor para la ocasión.

### Implementación de un servicio activable por socket

El ejecutable de antes no nos sirve, puesto que realizaba él todas las etapas de una conexión TCP. Ahora es systemd el que establece la conexión y nuestro nuevo ejecutable debe recibir la conexión y utilizarla. Estos son los pasos a dar :

1. Tener instalado libsystemd-dev
2. #incluir : #include <systemd/sd-daemon.h>
3. Para el caso habitual de tener un sólo socket de escucha, el descriptor de fichero se obtiene así:

```
fd = SD_LISTEN_FDS_START + 0;
```

4. Este descriptor es directamente utilizable para leer o escribir:.en el caso del servidor de la hora:

```
ticks = time(NULL);
snprintf(sendBuff, sizeof(sendBuff),...
write(fd, sendBuff, strlen(sendBuff));
```

5. Cerrar el descriptor al finalizar

```
close(fd);
```

En el anexo se tiene el código completo, comentado y evolucionado a partir del servidor de hora simple:

Para observar bien la práctica, se puede modificar el código añadiendo un sleep() gordo, de forma que el proceso resultante de una activación se queda un tiempo activo y da tiempo a ver su existencia con ps, y con systemctl status. Con este último comando, se puede ver que se crean units transient (transitorias) para cada petición lanzada, que desaparecen cuando finaliza su ejecución.

## 6.4 Instalación de un servicio a partir de una unit ".path"

En este experimento abandonamos el ejecutable ServHoraMult y recurrimos a utilizar scripts sencillos para demostrar otra funcionalidad de systemd.

Necesitaremos dos units; una ".path" y otra llamada igual pero ".service"

`experimento.path`

```
[Unit]
Description=Experimento
Before=experimento.service

[Path]
PathExists=/root/test.trigger

[Install]
WantedBy=multi-user.target
```

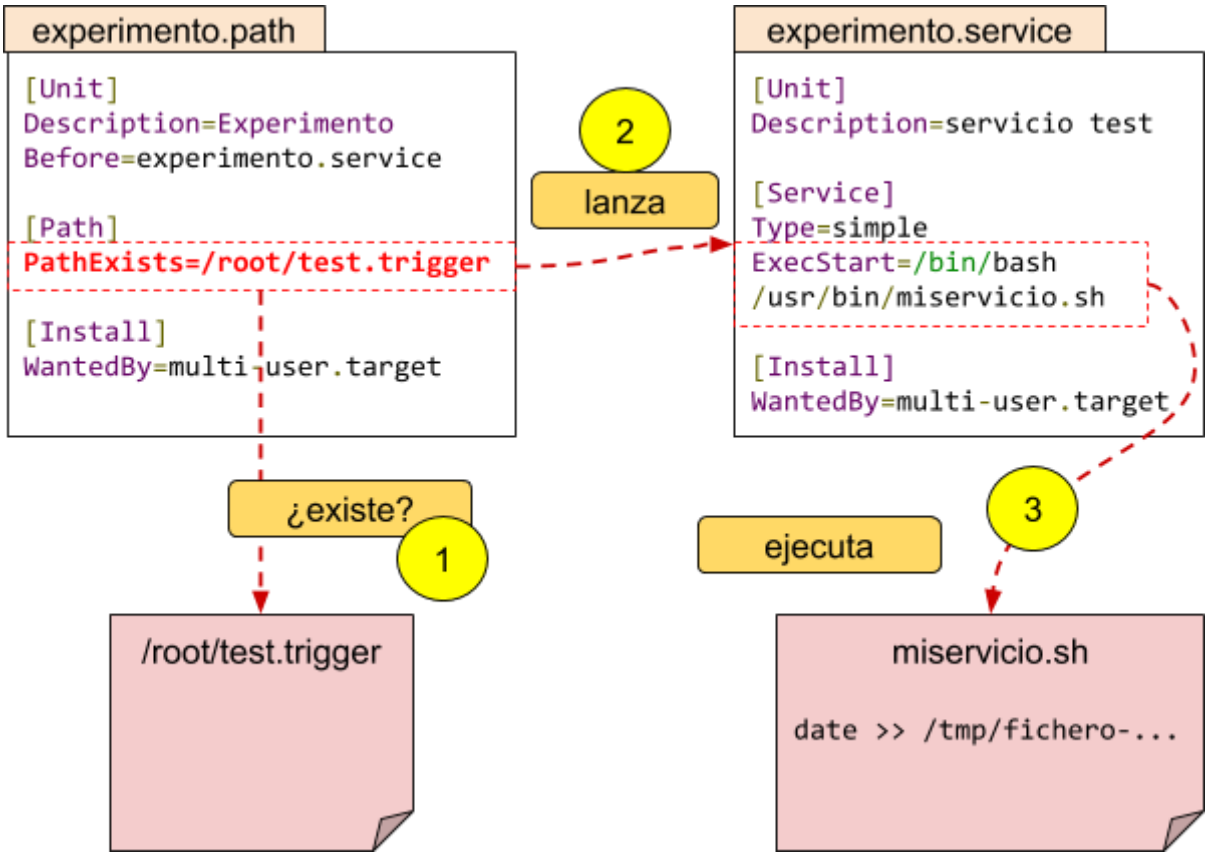
experimento .service

```
[Unit]
Description=servicio test

[Service]
Type=simple
ExecStart=/bin/bash /usr/bin/miservicio.sh

[Install]
WantedBy=multi-user.target
```

Se necesita hacer

```
systemctl enable experimento.path
systemctl start experimento.path (para probar inmediatamente)
```

Ahora tan pronto como se cree el fichero **/root/test.trigger**, se activará el servicio basado en el script /usr/bin/miservicio.sh




El script **`miservicio.sh`** tiene como misión simplemente demostrar que se ha ejecutado a una hora concreta, el script puede contener sencillamente la siguiente línea:

```
date >> /tmp/fichero-test-montaje-systemd.tmp
```

De esa forma, en el fichero anterior veremos una línea por cada ejecución del script y podremos constatar que todo funciona bien.

## 6.5 Resumen de pasos para instalar un servicio

1. Crear el ejecutable, compilarlo, etc.
2. copiar el ejecutable a su ubicación "estable", "final", adecuada
3. Crear las units necesarias en "**`/usr/lib/systemd/system`**"
    a. (si hace faltA) units de tipo socket, path, mount, etc. lo que requiera el servicio y el ejecutable
    b. (seguro) crear la **`unit .service`**. Importante la sección **`[Install]`**

    ```
    [Install]
    WantedBy=multi-user.target
    ```

4. Instalar el servicio (una vez, al principio de los tiempoS). Hay que ser conscientes que hacer el fichero anterior, por sí, no cambia nada. El fichero .service está simplemente en el repositorio y el "mult-user.target" no se ha enterado.

   ```
   systemctl enable servHoraMult.service
   ```

   Systemd observa la sección [Install] de la unit y sabe que este servicio debe ser incluido en multi-user.target… en la configuración qeu se aplica
   **`/etc/systemd/system/multi-user.target.wants`**

   Adicionalmente se ha instalado el socket // OJO!!

   ```
   systemctl enable servHoraMult.socket
   ```

5. A partir de aquí el servicio arrancará solito pero podemos manipularlo manualmente con
   a. `systemctl stop servHoraMult.service`
   b. `systemctl start servHoraMult.service`
   c. `systemctl restart servHoraMult.service`

# 10 Anexos

## 6.6 ServHora simple

Usado en el ejercicio para trastear con el concepto de servidor, Se copia el código a un fichero llamado "servHora.c" y se compila con la línea siguiente:

```
g++ –o servHora servHora.c
```

Con ello se obtendrá un ejecutable llamado servHora que será usado en la actividad descrita

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

static const int PORT = 5000;

/* probar este programa con netcat localhost 5000*/

int main(int argc, char *argv[])
{
       int listenfd = 0, connfd = 0;
```

```
        struct sockaddr_in serv_addr;

        char sendBuff[1025];
        time_t ticks;

        listenfd = socket(AF_INET, SOCK_STREAM, 0);
        memset(&serv_addr, '0', sizeof(serv_addr));
        memset(sendBuff, '0', sizeof(sendBuff));

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port = htons(PORT);

        bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

        listen(listenfd, 10);

        while(1)
        {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

        ticks = time(NULL);
        snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));
        printf("Sending info: %s",sendBuff);
        write(connfd, sendBuff, strlen(sendBuff));

        close(connfd);
        sleep(1);
        }
}
```

## 6.7 Servidor Hora activable por socket

A partir del anterior servidor, hemos implementado otro que es activable con un socket gestionado por systemd.

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

 #include <systemd/sd-daemon.h>
```

```
static const int PORT = 5000;

/* probar este programa con netcat localhost 5000*/

int main(int argc, char *argv[])
{
      int  connfd = 0; // no need of listenfd
//    struct sockaddr_in serv_addr;

      char sendBuff[1025];
      time_t ticks;

  /* Todo lo siguiente está hecho por systemd socket activation
       All the following is done with socket activation by systemd

      listenfd = socket(AF_INET, SOCK_STREAM, 0);
      memset(&serv_addr, '0', sizeof(serv_addr));
      memset(sendBuff, '0', sizeof(sendBuff));

      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      serv_addr.sin_port = htons(PORT);

      bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));

      listen(listenfd, 10);


      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
*/
// la siguiente línea establece el descriptor pasado por systemd
    connfd = SD_LISTEN_FDS_START + 0;

      ticks = time(NULL);
      snprintf(sendBuff, sizeof(sendBuff), "%.24s\n", ctime(&ticks));
      printf("Sending info: %s",sendBuff);
      write(connfd, sendBuff, strlen(sendBuff));

      close(connfd);
}
```

## 6.8 ServHoraMult multiproceso

Este es el código que hay que copiar para realizar el ejercicio. (versión con fichero /run/servHoraMul.pid). Se copia a un fichero llamado "servHoraMult.c++" y se compila con la línea siguiente:

```
g++ –o servHoraMult servHoraMult.c++
```

Con ello se obtendrá un ejecutable llamado servHoraMult que será usado en siguientes pasos de la actividad,

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>

int main(int argc, char *argv[])
{
    bool pidFileOpen = true;
    FILE * fptr;
    fptr = fopen("/run/servhora.pid","w");

    if (fptr == NULL) {
       pidFileOpen = false;
       fprintf(stderr, "Could not open run.pid file");

    }
      pid_t myPid = getpid();

      if (pidFileOpen) fprintf(fptr,"%d",myPid);


      int listenfd = 0, connfd = 0;
      struct sockaddr_in serv_addr;

      char sendBuff[1025];
      time_t ticks;

      listenfd = socket(AF_INET, SOCK_STREAM, 0);
      memset(&serv_addr, '0', sizeof(serv_addr));
      memset(sendBuff, '0', sizeof(sendBuff));

      serv_addr.sin_family = AF_INET;
      serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
      serv_addr.sin_port = htons(5000);

      bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
```

```
      listen(listenfd, 10);

      while(1)
      {
      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
       pid_t pid;

       pid = fork();


       if (pid != 0 ) close(connfd);
       else {
                  for (int i=0; i< 10;i++) {
                        ticks = time(NULL);
                        snprintf(sendBuff,
                        sizeof(sendBuff),
                        "%.24s\r\n",
                        ctime(&ticks));
                        printf("Sending info: %s",sendBuff);
                        write(connfd, sendBuff, strlen(sendBuff));
                        sleep(1);
                  }
                   close(connfd);
                   exit(0);
       }// if hijo
      } // while

      if (pidFileOpen) fclose(fptr);
}
```

Ejercicio:

Ejercicio:

# ---------- Ha partir de haki ----------

# ---------- Todo hes vasura ----------

Units transient:

```
systemd will dynamically create device units for all kernel devices that are
marked with the "systemd" udev tag (by default all block and network devices,
and a few others). This may be used to define dependencies between devices
and other units. To tag a

udev device, use "TAG+="systemd"" in the udev rules file, see udev(7) for
details.

Device units are named after the /sys and /dev paths they control. Example:
the device /dev/sda5 is exposed in systemd as
dev-sda5.device. For details about the escaping logic used to convert a file
system path to a unit name see systemd.unit(5).

systemctl -t device
```

## 7 /run/systemd/transient

cuando se ejecuta systemd, en ocasiones se crean units temporales (transient), los ficheros correspondientes están en /run/systemd/transient

## 8 jerarquía de grupos de control en systemd.

Un sistema linux en funcionamiento es un conjunto de procesos ejecutándose en background. EStos procesos han sido iniciados por systemd. Pero existe una organización y control de todos esos procesos. Están agrupados en "control groups" que es una función proporcionada por el núcleo de linux para limitar los permisos y recursos de un grupo de procesos.

Al hacer

systemctl status se puede observar la jerarquía indicada

Los grupo de control se pueden observar/manipular en

`/sys/fs/cgroups`

Donde cada directorio es un grupo de control al que pueden adscribirse procesos.

systemd tiene su propio directorio que reproduce la salida del comando `systemclt` status.

en

https://www.freedesktop.org/software/systemd/man/systemd.slice.html

se explica cómo se determina que exista esta jerarquía en funcionamiento.

## 8.1 Revisar este ejercicio:

https://blog.tjll.net/systemd-for-device-activation-and-media-archiving/

## 8.2 Revisar este ejercicio

1

Although I still don't know how ENV{SYSTEMD_USER_WANTS} works, I managed to get my specific problem solved after reading this blog.

It turns out that I can install targets as a dependency on devices. I changed my unit file ~/.config/systemd/user/docked.target to:

```
[Unit]
Description=Docked to ThinkPad Mini Dock
BindsTo=dev-tp_mini_dock.device
After=dev-tp_mini_dock.device

[Install]
WantedBy=dev-tp_mini_dock.device
```

and my udev rule to:

```
SUBSYSTEM=="usb", ACTION=="add", ENV{ID_VENDOR}=="17ef", ENV{ID_MODEL}=="100a",
```

SYMLINK+="tp_mini_dock", TAG+="systemd"

and then enable it with systemctl --user enable docked.target.

Now, when I dock it, the udev rule creates the systemd device, which in turn starts up the target. Then the BindsTo option makes sure that when the device disappears (gets unplugged) the target gets stopped.

I had to do some nonsensical magic to get this to work when I login with the dock already plugged in. One would imagine that simply adding default.target to WantedBy and After would be enough... I'll add a link to a blog after I write it.

## 9 slice units

https://www.freedesktop.org/software/systemd/man/systemd.scope.html

básicamente las unit slice no vienen configuradas por ningún fichero si no que son creadas programáticamente y se utilizan para manipular y gestionar conjuntos de procesos servidores.

```
int conexion, conexion_cliente;11
int resultado, tam_dir_cliente, pid;
struct sockaddr_in dir_propia, dir_cliente;
/* CREACION de un socket TCP */
conexion = socket(PF_INET, SOCK_STREAM, 0);
...
/* BIND local (establecer PUERTO) */
memset(&dir_propia, 0, sizeof(struct sockaddr_in));
dir_propia.sin_family = AF_INET;
dir_propia.sin_addr.s_addr = htonl(INADDR_ANY);
dir_propia.sin_port = htons(...); /* establecer PUERTO de escucha*/
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
/* poner en MODO ESCUCHA (tamano cola = 20) */
resultado = listen(conexion, 20);
...
while(1) { /* Bucle principal aceptando conexiones*/
       /* ACEPTAR conexion del cliente */
       tam_dir_cliente = sizeof(dir_cliente);
       conexion_cliente = accept(conexion,
                           (struct sockaddr *) &dir_cliente,&tam_dir_cliente);
```

```
        /* Crear un proceso hijo para procesar la conexion de cada cliente */
        pid = fork();
        if (pid == -1) { /* error */ ... }

        else if (pid == 0) { /* PROCESO HIJO */
           close(conexion); /* liberar conexion */
           while (! fin_dialogo) { /* Bucle de DIALOGO con cliente */
                recibidos = recv(conexion_cliente,peticion,tam_peticion,0); /*RECEPCION
        */
                ...
                respuesta = procesar_peticion(peticion);
                ...
                enviados = send(conexion_cliente,respuesta,tam_respuesta,0); /* ENVIO */
                ...
                }
           close(conexion_cliente); /* CERRAR conexion con cliente */
          exit(0); /* finaliza proceso hijo */
          }
        else { /* Proceso PADRE */
               close(conexion_cliente); /* liberar conexion con cliente */
        }
}
close(conexion); /* CERAR conexion */
...
```

## 9.1 Simplificación del esquema de un servidor multiproceso:

¡ Sin errores que controlar ! :

```
...
/* Creamos un socket que usaremos en el resto de proceso */
conexion = socket(PF_INET, SOCK_STREAM, 0);
...
/* Asociamos el socket a un puerto  */
resultado = bind(conexion,(struct sockaddr *) &dir_propia,sizeof(dir_propia));
...
/* poner en MODO ESCUCHA */
resultado = listen(conexion, ...);
...
while(1) { /* Bucle principal aceptando conexiones*/
       ...
       /* Esperar y aceptar la conexión del cliente */
       conexion_cliente = accept(conexion,
                           (struct sockaddr *) &dir_cliente,&tam_dir_cliente);

       /* Crear un proceso hijo para procesar la conexion de cada cliente */
       pid = fork();
       if (pid == 0) { /* PROCESO HIJO */
          close(conexion); /* liberar socket del padre nos quedamos con la del
```

```
cliente */
          while (! finm ) {
              intercambiar_datos_con_cliente();
              }
           close(conexion_cliente); /* CERRAR conexion con cliente */
          exit(0); /* finaliza proceso hijo */
          }
       else { /* Proceso PADRE */
           close(conexion_cliente); /* liberar conexión con cliente (el hijo se
       encarga) */
       }
}
close(conexion); /* CERRAR conexión. En teoría nunca se llega aquí*/
```

Modificación : introducir una espera artificial para que tarde 10 segundos en responder al cliente. Esto deja "epserando " al cliente durante 10 segundos y después el servidor envía la cadena, el cliente la muestra y el close(connfd) del servidor cierra la comunicación normal

```
#include <sys/socket.h>
#i...
      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

      ticks = time(NULL);
      snprintf(sendBuff, sizeof(sendBuff), "%.24s\r\n", ctime(&ticks));
       printf("Sending info: %s",sendBuff);
       sleep (10 );
      write(connfd, sendBuff, strlen(sendBuff));

      close(connfd);
      sleep(1);
      }
}
```

Ahora, mientras el servidor está tardando en responder, debemos lanzar uno o más clientes intentando obtener respuesta… veremos que parece que hasta que no termine con un cliente, el servidor no empieza con el otro…. retrasando mucho la respuesta. Esto es una simulación de lo que ocurre cuando varios clientes "atacan" al servidor a la vez con peticiones,

Modifico el bucle para que tarde…. enviando 20 fechas.

```
 while(1)
     {
     connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);

    for (int i=0; i<20; i++) {
      ticks = time(NULL);
      snprintf(sendBuff,
             sizeof(sendBuff),
             "%.24s\r\n",
             ctime(&ticks));
      printf("Sending info: %s",sendBuff);
      write(connfd, sendBuff, strlen(sendBuff));
      sleep(1);
    }
     close(connfd);
     }
```

## 9.2 Ampliación log (esto está hecho !!!! y bien!!! en otro documento y lección)

para generar log con escritura en salida de error, podemos escribir las direcciones ip desde las que se hace cada llamada:

¿Cómo averiguar la dirección IP? <adaptar del siguiente ejmplo>

```
int sockfd;

void main(void) {
    //[...]
    struct sockaddr_in clientaddr;
    socklen_t clientaddr_size = sizeof(clientaddr);
    int newfd = accept(sockfd, (struct sockaddr *)&clientaddr,
&clientaddr_size);
    //fork() and other code
    foo(newfd);
    //[...]
}
```

```
void foo(int newfd) {
    //[...]
    struct sockaddr_in addr;
    socklen_t addr_size = sizeof(struct sockaddr_in);
    int res = getpeername(newfd, (struct sockaddr *)&addr, &addr_size);
    char *clientip = new char[20];
    strcpy(clientip, inet_ntoa(addr.sin_addr));
    //[...]
}
```

## 9.3 servHora Multiproceso

```
while(1)
      {
      connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
       pid_t pid;

       pid = fork();

       if (pid != 0 ) close(connfd);
       else {
                 ticks = time(NULL);
                 snprintf(sendBuff,
                       sizeof(sendBuff),
                       "%.24s\r\n",
                       ctime(&ticks));
                 printf("Sending info: %s",sendBuff);
                 write(connfd, sendBuff, strlen(sendBuff));
                 sleep(1);
            close(connfd);
            exit(0);
       }// if hijo
 } // while
```

Aunque se puede probar, no lleva espera como el ejemplo anterior, si introducimos el sleep(10) antes de write() y lanzamos varios clientes, observaremos que la espera es independiente unos de otros . Es decir, cada cliente es atendido inmediatamente y su espera es de 10 segundos, sin más, sin tener que esperar a otros clientes.

```
...
                snprintf(sendBuff,
                      sizeof(sendBuff),
                      "%.24s\r\n",
                      ctime(&ticks));
                printf("Sending info: %s",sendBuff);
                sleep(10);
                write(connfd, sendBuff, strlen(sendBuff));
                sleep(1);
            close(connfd);
      ...
```

Se puede modificar la espera de 10 por una sucesión de mensajes y esperas de sleep (1) para ver que simultáneamente el servidor es capaz de atender a varios clientes.