

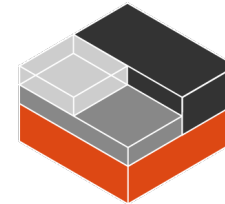
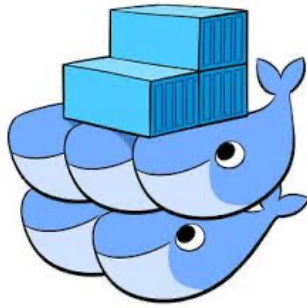
upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

Redes de Nueva Generación  
*Área de Ingeniería Telemática*

# Virtualización a nivel de sistema operativo

# ¿De qué va esto?



podman



docker



Apache  
AURORA™

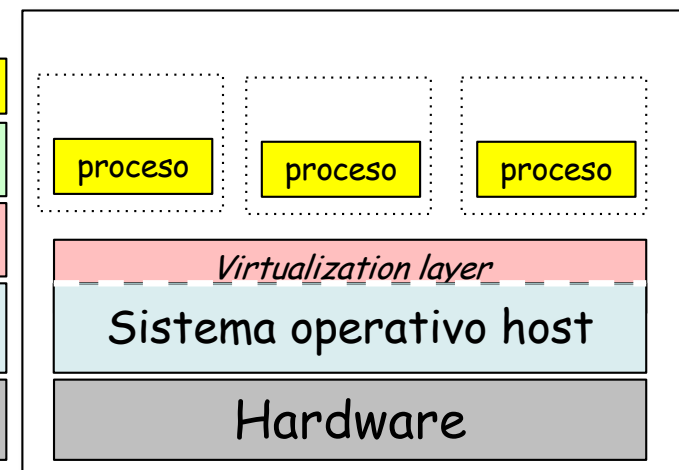
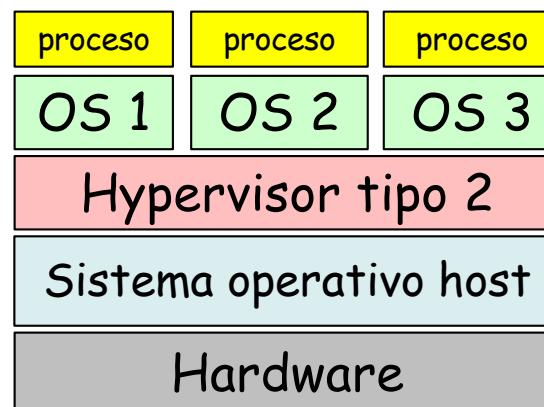
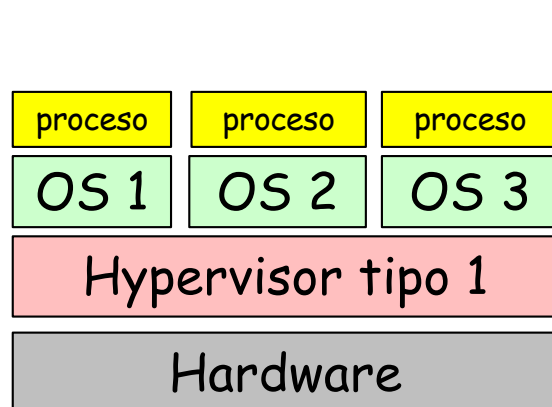


MARATHON



# Operating-system-level virt.

- *Containers, Virtualization Engines, Virtual Private Servers, Jails*
- Ejemplos: Docker, containerd, runc, crun, rkt, LXC, OpenVZ, Solaris Zones, FreeBSD Jails, etc
- En entornos UNIX, ahora mayormente en kernel Linux, también en Windows, también mediante virtualización de OS
- Virtualización de subsistemas del sistema operativo
- No hay múltiples kernels de múltiples OSs sino un solo kernel que virtualiza elementos suyos
- En realidad no son más que un conjunto aislado de procesos
- ¿Por qué? ¿Para qué? ¿Cómo?
- Aislamiento



upna

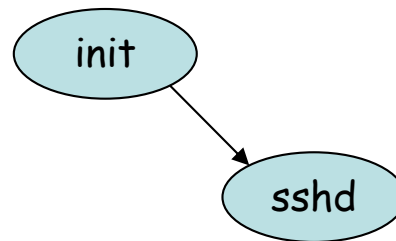
Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*

# Creación de procesos

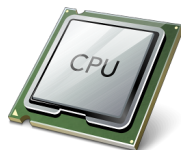
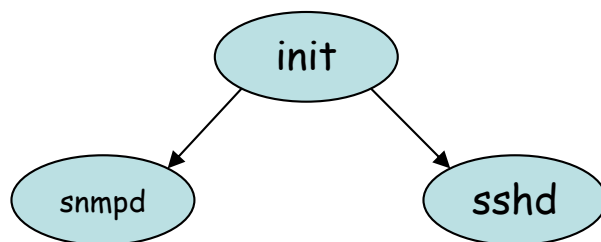
# Origen de los procesos

- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



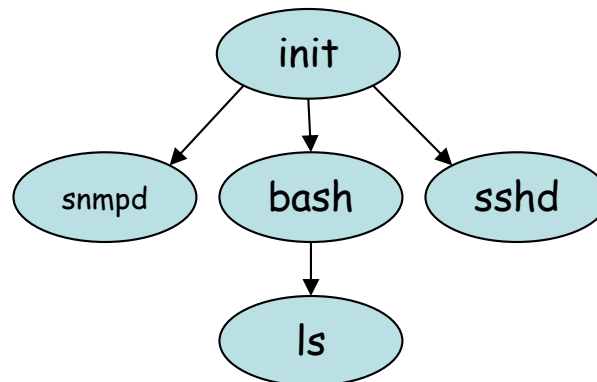
# Origen de los procesos

- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



# Origen de los procesos

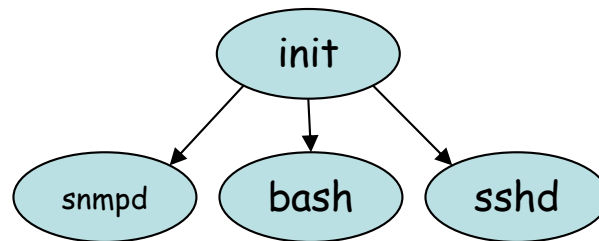
- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



```
$ ls
bin    dev    lib    libx32  mnt    root  snap    sys  var
boot  etc    lib32  lost+found  opt    run    srv      tmp
cdrom  home  lib64  media    proc   sbin  swapfile  usr
```

# Origen de los procesos

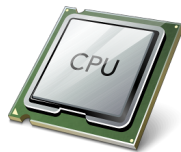
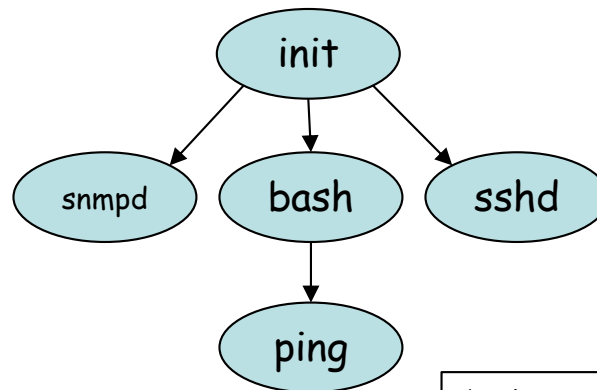
- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...





# Origen de los procesos

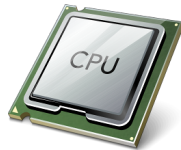
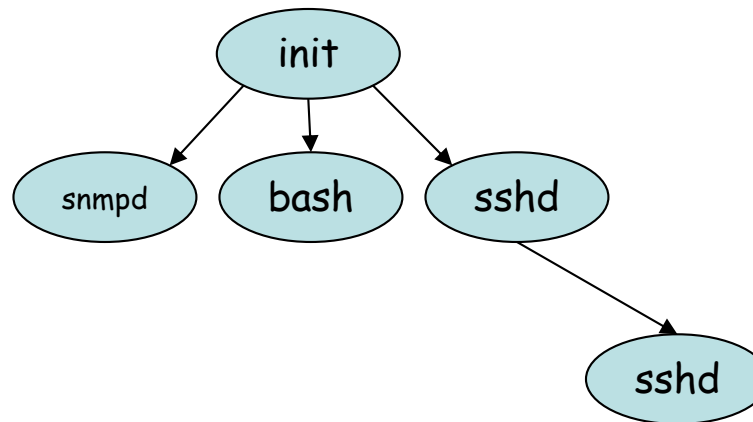
- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



```
$ ping www.google.com
PING www.google.com (172.217.17.4): 56 data bytes
64 bytes from 172.217.17.4: icmp_seq=0 ttl=115 time=10.362 ms
64 bytes from 172.217.17.4: icmp_seq=1 ttl=115 time=10.800 ms
64 bytes from 172.217.17.4: icmp_seq=2 ttl=115 time=9.985 ms
^C
--- www.google.com ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 9.985/10.382/10.800/0.333 ms
```

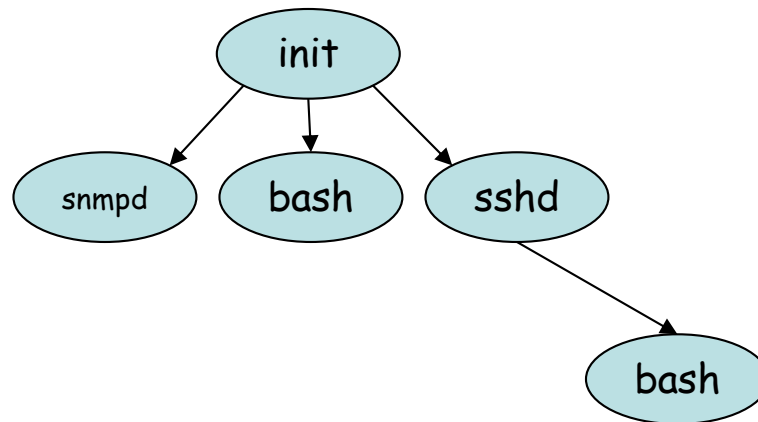
# Origen de los procesos

- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



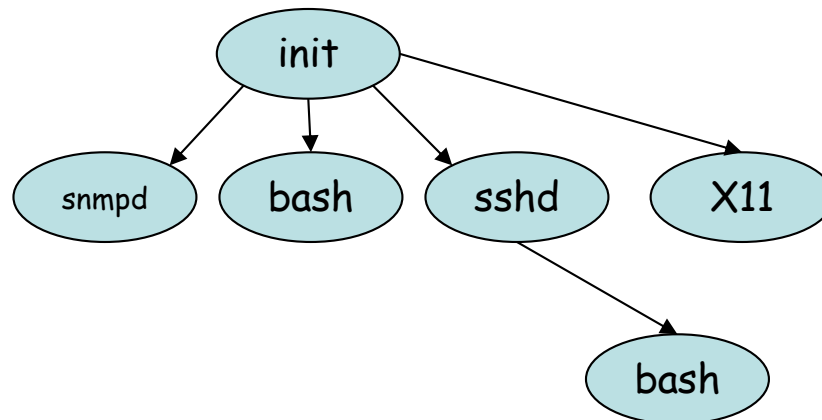
# Origen de los procesos

- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



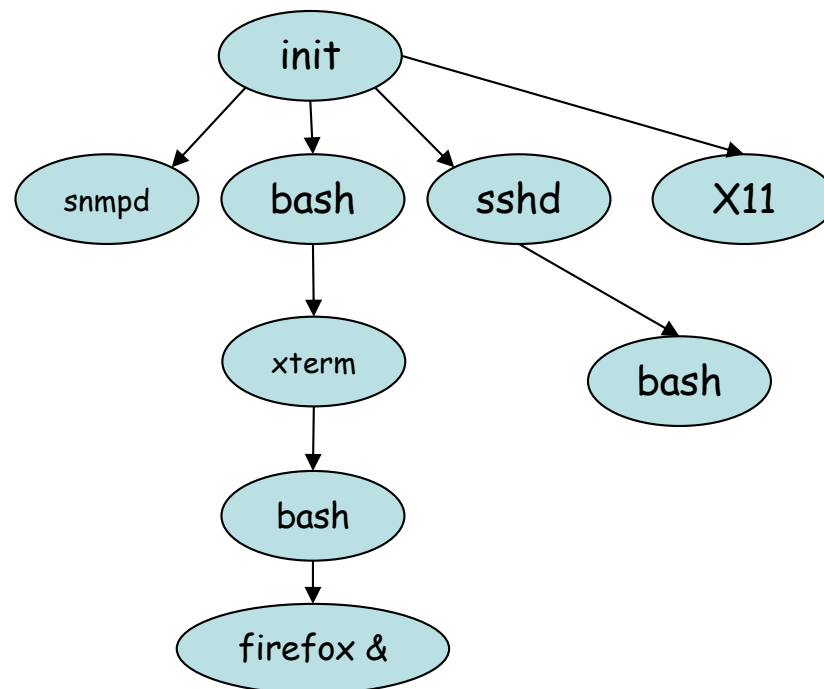
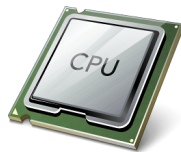
# Origen de los procesos

- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



# Origen de los procesos

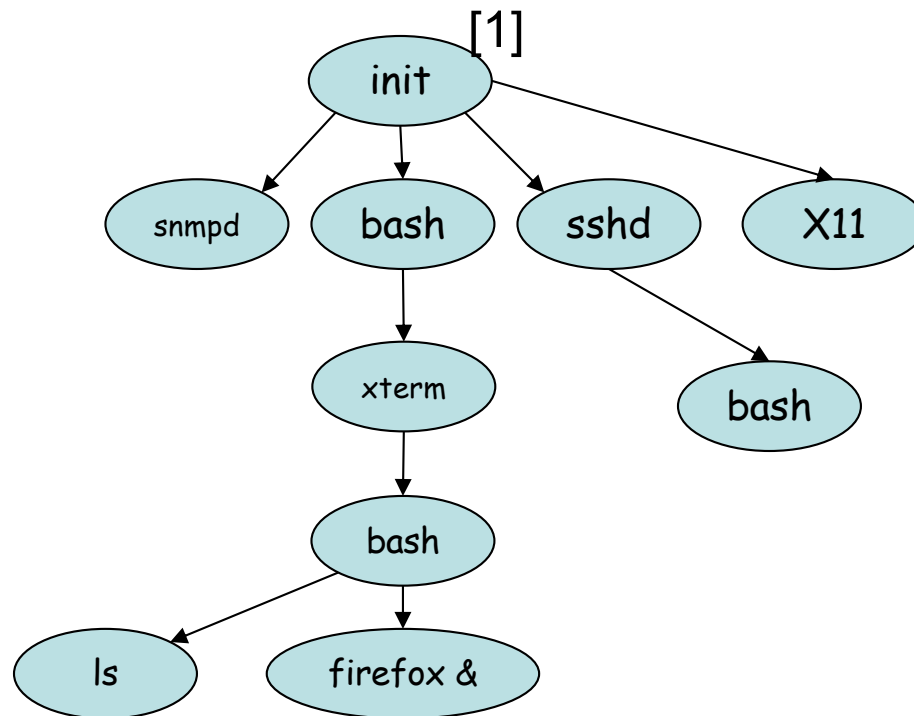
- Suele haber un proceso inicial que lanza el kernel (tradicionalmente init)
- Nuevos procesos son el resultado de otro proceso clonándose
- El proceso original es el padre, el otro el hijo
- Normalmente en el proceso hijo se pasa a ejecutar otro programa
- Cualquiera de esos procesos puede seguir clonándose ...



# Origen de los procesos

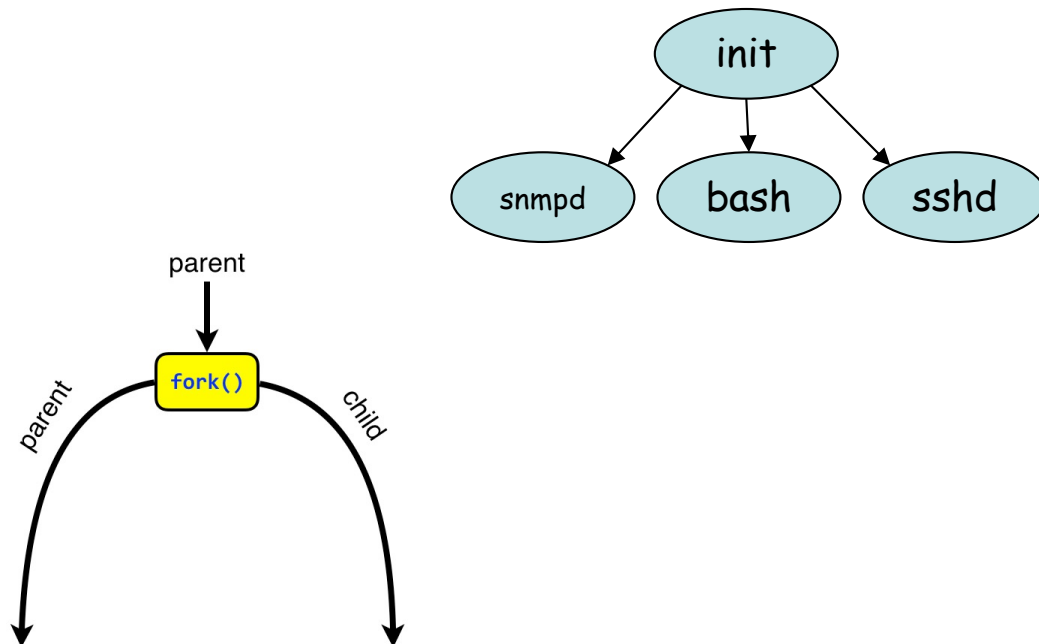
- Tenemos un árbol de procesos
- El proceso inicial tiene el Process ID (PID) 1
- Además de init existen systemd, upstart, launchd...

```
1 /sbin/init maybe-ubiquity
349 /lib/systemd/systemd-journald
381 /lib/systemd/systemd-udev
554 /sbin/multipathd -d -s
596 /lib/systemd/systemd-timesyncd
641 /lib/systemd/systemd-networkd
643 /lib/systemd/systemd-resolved
655 /usr/lib/accountsservice/accounts-daemon
660 /usr/sbin/cron -f
663 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile ...
671 /usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
673 /usr/sbin/rsyslogd -n -iNONE
675 /usr/lib/snapd/snapd
680 /lib/systemd/systemd-logind
686 /usr/sbin/atd -f
```



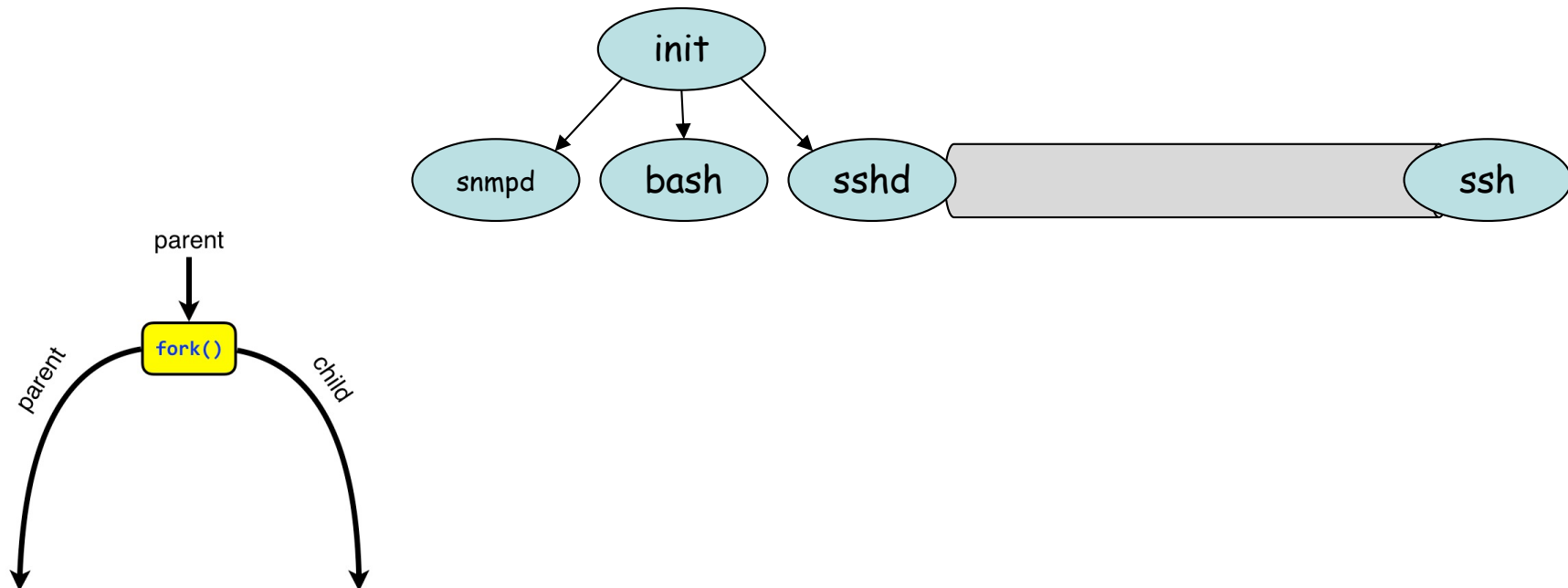
# Herencia

- El proceso hijo tiene una copia de la memoria en uso por el padre
- También una copia de estructuras en el kernel
- Eso incluye los *file descriptors* (descriptores de ficheros abiertos)
- Un socket es un descriptor de ficheros
- Entonces por ejemplo si el padre tenía una conexión TCP (...)



# Herencia

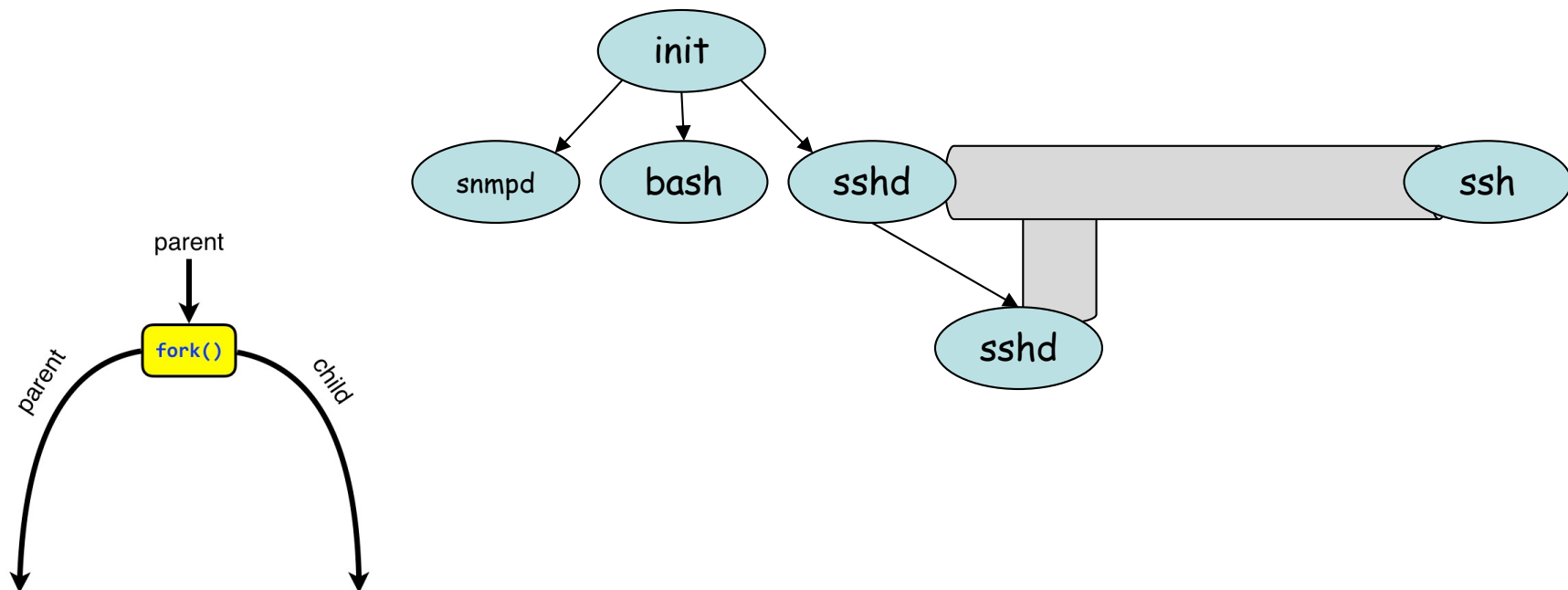
- El proceso hijo tiene una copia de la memoria en uso por el padre
- También una copia de estructuras en el kernel
- Eso incluye los *file descriptors* (descriptores de ficheros abiertos)
- Un socket es un descriptor de ficheros
- Entonces por ejemplo si el padre tenía una conexión TCP y crea ahora un proceso hijo (...)





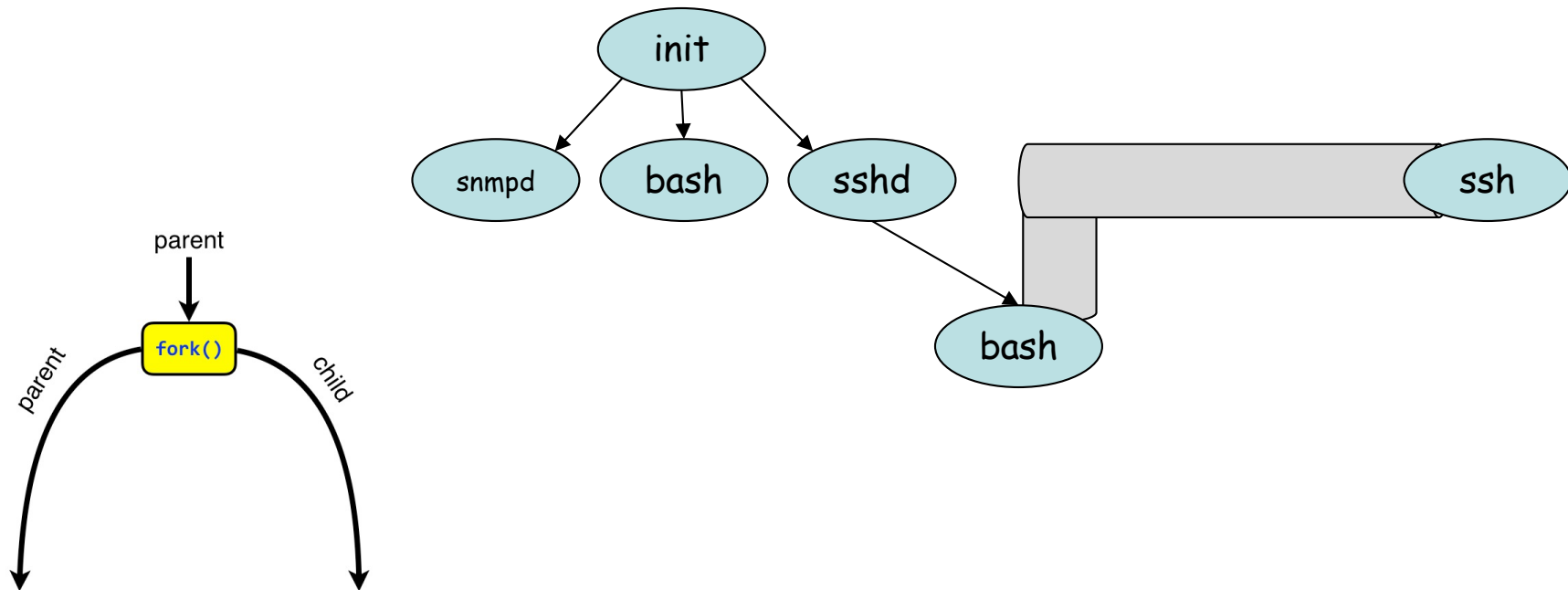
# Herencia

- El proceso hijo tiene una copia de la memoria en uso por el padre
- También una copia de estructuras en el kernel
- Eso incluye los *file descriptors* (descriptores de ficheros abiertos)
- Un socket es un descriptor de ficheros
- Entonces por ejemplo si el padre tenía una conexión TCP y crea ahora un proceso hijo, el hijo tiene una copia de ese fd
- Los dos procesos pueden leer y escribir en la conexión TCP



# Herencia

- Habitualmente el padre cierra su extremo para que se haga cargo el hijo (pero esto ya es decisión del programador)
- En el ejemplo de servidor ssh el hijo podría cambiar ahora a ejecutar otro programa



# Aislamiento

- Hemos visto que el Kernel ejerce bastante control y aislamiento sobre los procesos
  - Aislamiento de memoria (memoria virtual)
  - ... pero un proceso podría solicitar toda la del sistema
  - Preemptive scheduling
  - ... pero un proceso podría requerir mucho tiempo de CPU
  - Permisos de acceso a fichero
  - ... pero podemos ver toda la estructura de ficheros
  - Usuarios
  - ... pero no puedo tener más usuarios root con diferentes privilegios
  - Sockets para la comunicación de red independientes
  - ... pero todos los procesos ven todos los interfaces de red

upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

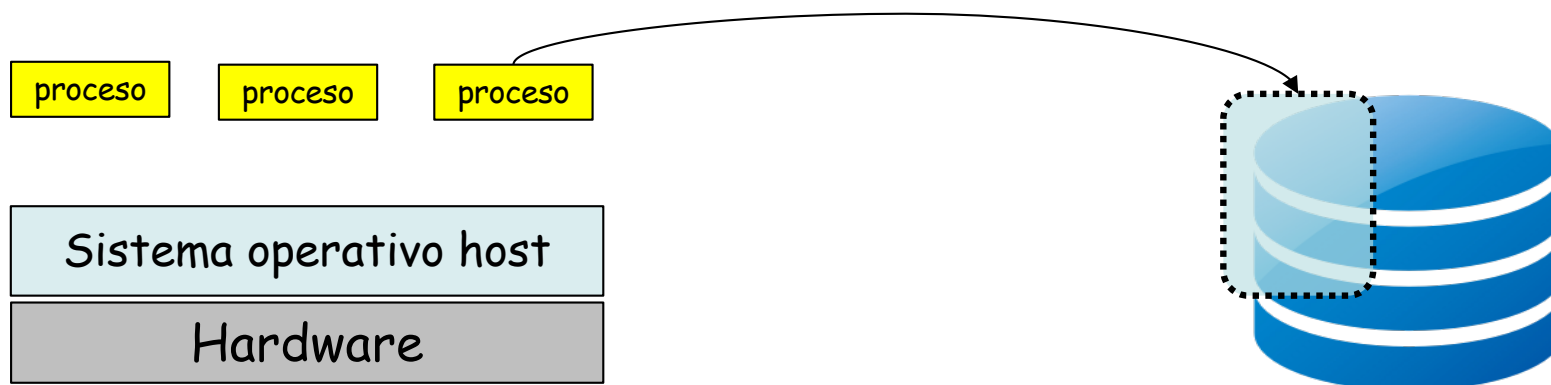
**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*

# Chroot

# Precursores: Chroot jail

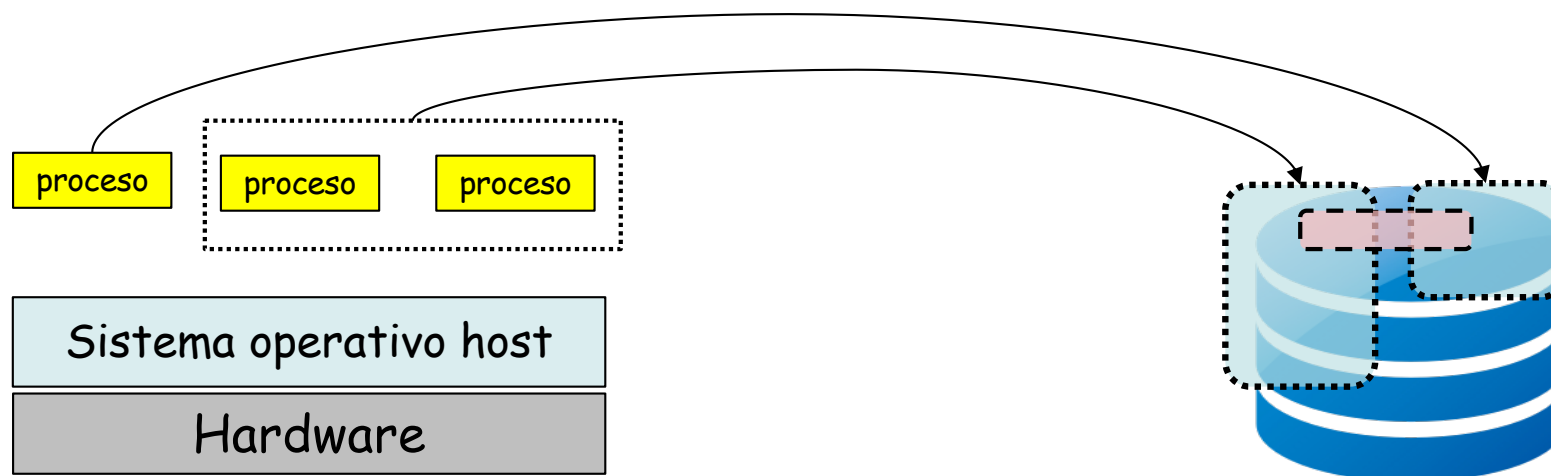
- Normalmente un proceso ve todo el sistema de ficheros (dentro de los permisos otorgados)
- Podemos hacer que vea como / un subdirectorio del sistema

```
$ ls /home/tlm/imagenes/alpine/  
bin  dev  etc  home  lib  media  mm  mnt  opt  proc  root  run  
sbin  srv  sys  tmp  usr  var  
  
$ sudo chroot /home/tlm/imagenes /bin/sh  
  
/ # ls  
bin  etc  lib  mm  opt  root  sbin  sys  usr  
dev  home  media  mnt  proc  run  srv  tmp  var
```



# Precursores: Chroot jail

- Normalmente un proceso ve todo el sistema de ficheros (dentro de los permisos otorgados)
- Podemos hacer que vea como / un subdirectorío del sistema
- Descendientes de ese proceso tienen la misma limitación
- Otros procesos pueden estar en otro *chroot jail*
- El sistema de ficheros que ve cada uno es un subdirectorío de original pero para ellos es todo el sistema de ficheros
- Dentro estarían todas las utilidades y programas
- Pueden compartir disco (por ejemplo mismo inodos)



upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

Redes de Nueva Generación  
*Área de Ingeniería Telemática*

# OS virtualization en Kernel Linux

# Linux actual

- Desarrollo progresivo desde versiones 2.6 del kernel
- Lo que permite hoy en día el concepto de *Contenedores* son:
  - *namespaces*
  - *cgroups*



upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*

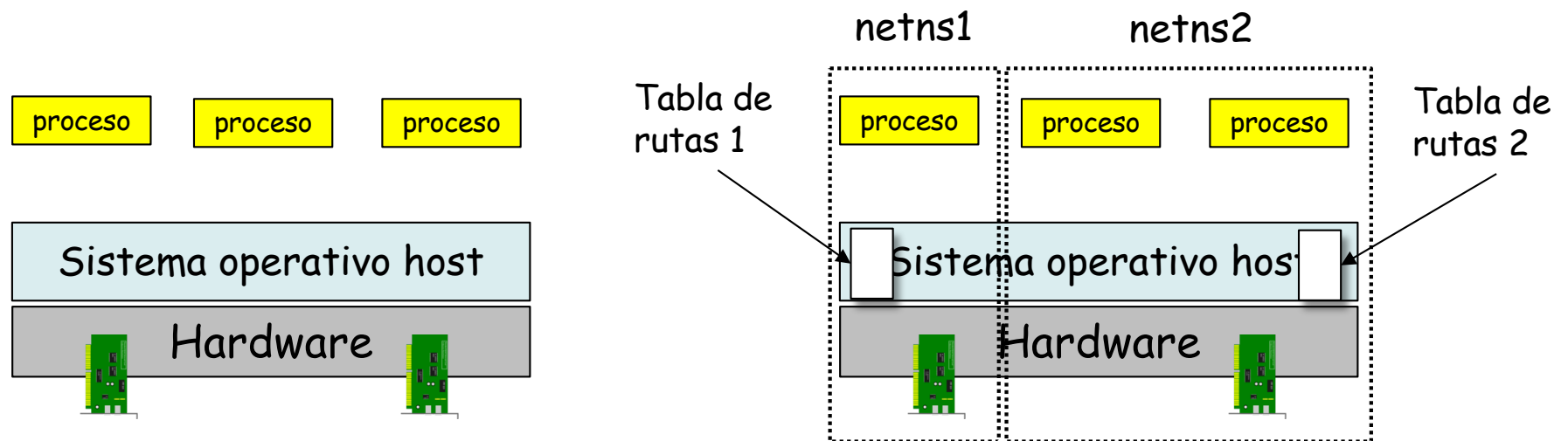
# Namespaces

# Linux namespaces

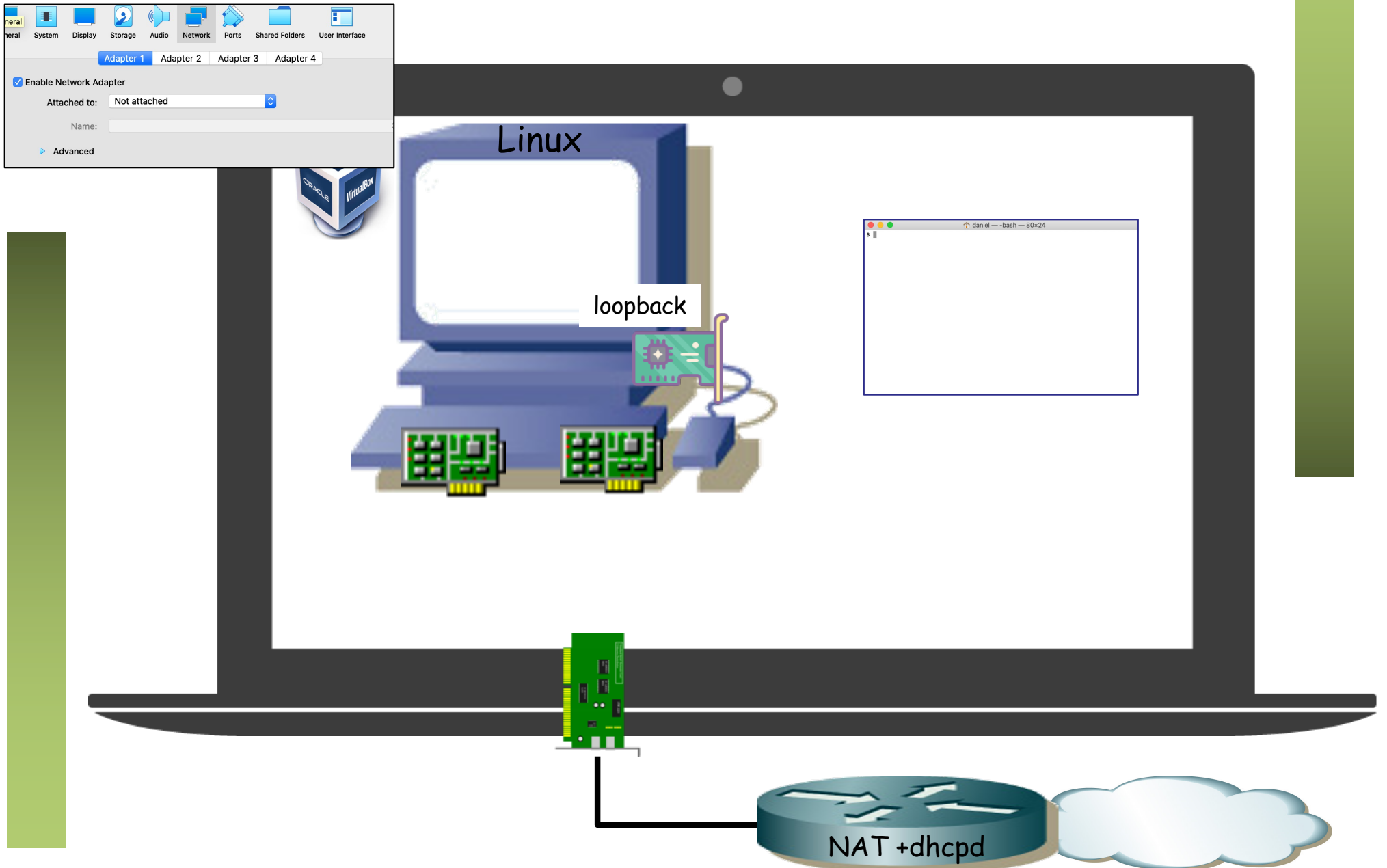
- Permiten que ciertos recursos del Kernel sean vistos por los procesos miembros del *namespace* como solo suyos
- Provee de una virtualización de dichos recursos
- Nos interesan principalmente los *network namespaces*
- Otros: PID, User, cgroup, IPC, Mount, UTS

# Network namespaces

- Permiten virtualizar la parte de red del Kernel
- Por ejemplo:
  - Que un conjunto de procesos vean algunos de los interfaces de la máquina y no el resto
  - Que vean una tabla de rutas que solo afecta a esos interfaces
  - Una tabla propia de reglas de filtrado
  - Lo mismo con la cache de ARP
  - Solo pueden crear sockets sobre direcciones de esos interfaces

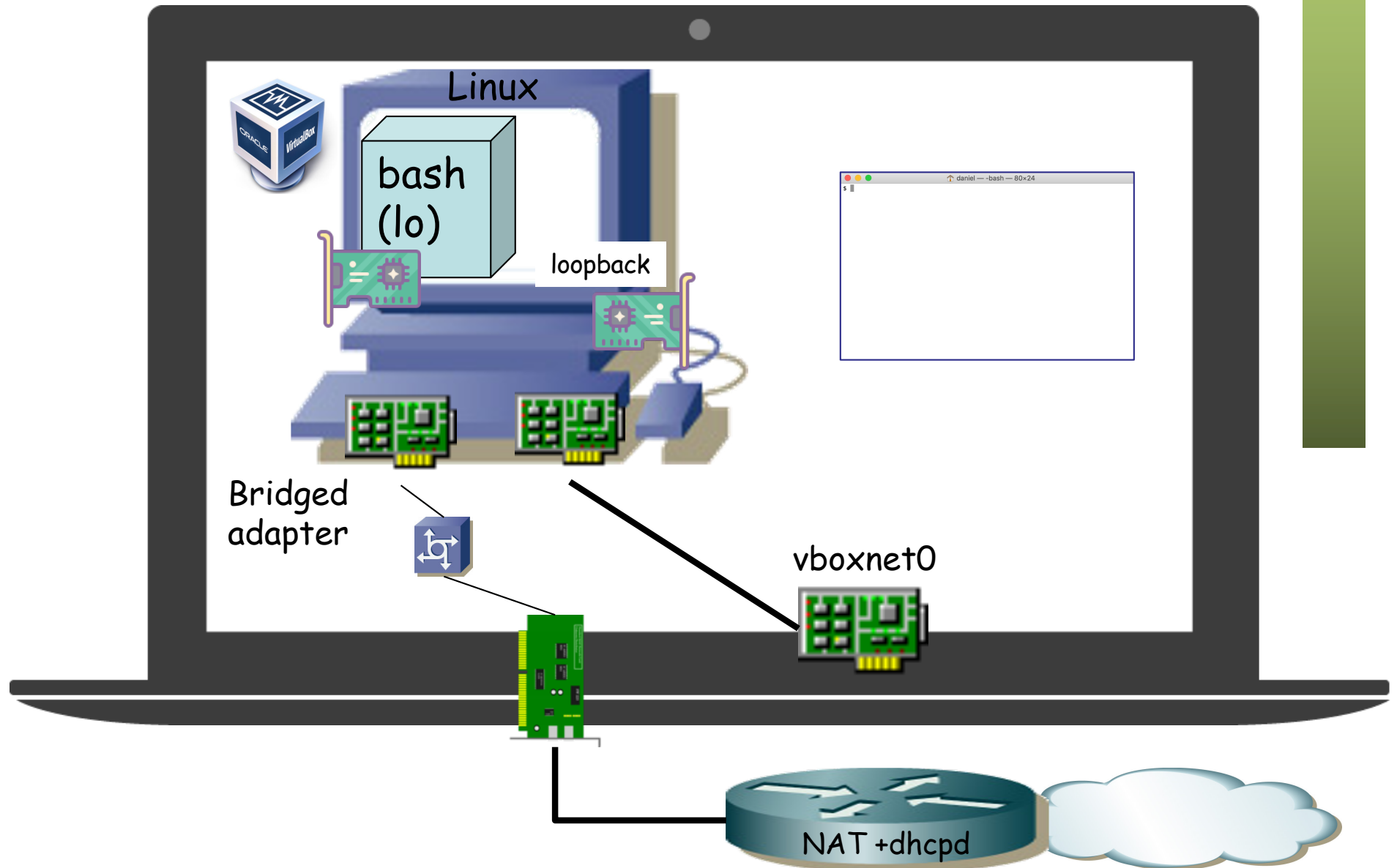


# NetNS: Ejemplo



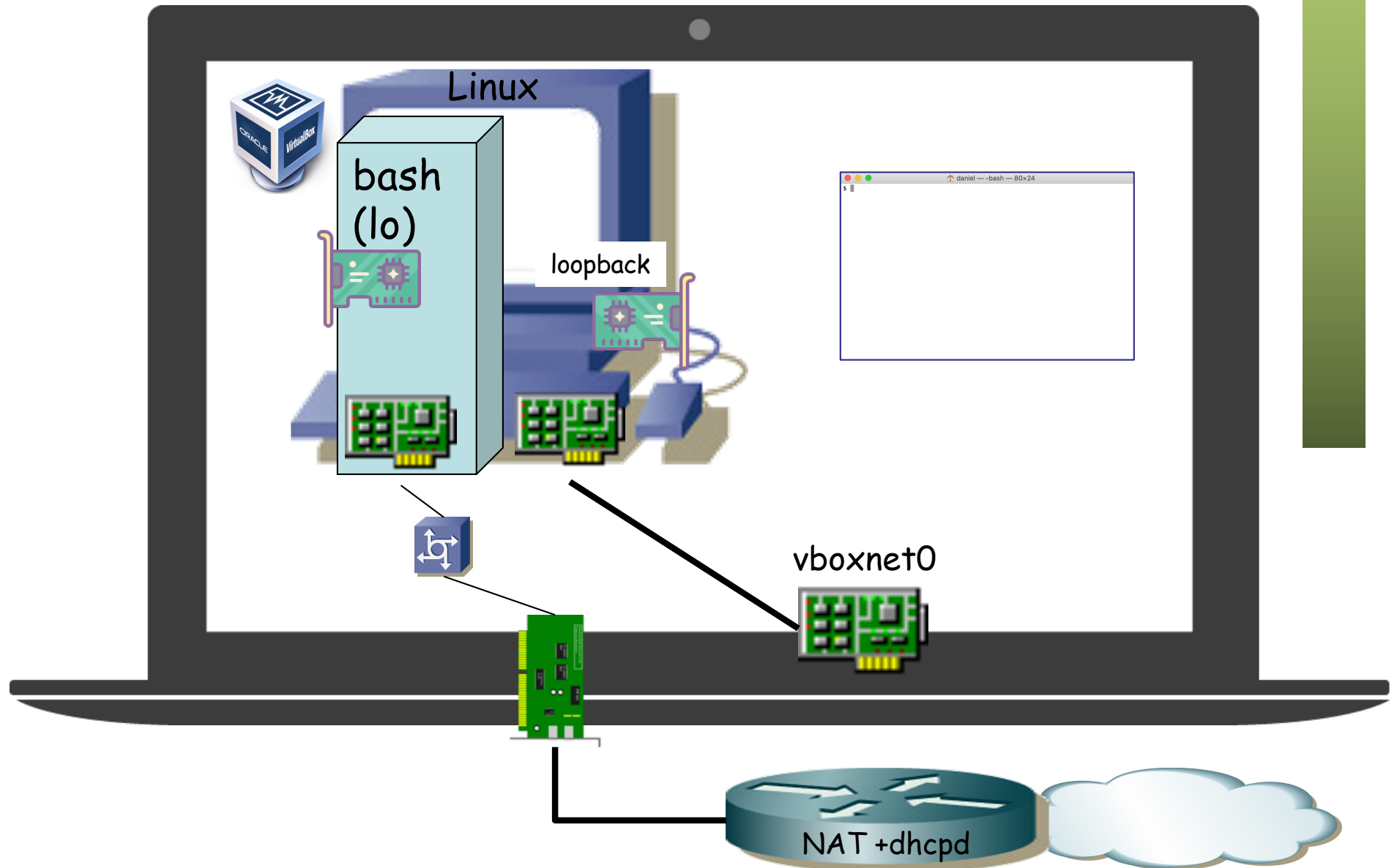
# NetNS: Ejemplo

- Tiene su propio *loopback* (lo)



# NetNS: Ejemplo

- Asignamos uno de los interfaces de la VM al namespace



# NetNS: Ejemplo

- Los procesos hijo heredan los *namespace* del padre
- El *network namespace* desaparece cuando no quede ningún proceso en él
- Los interfaces volverán al *namespace* raíz
- Comandos empleados:
  - `unshare` : Crea un nuevo proceso pudiendo crear nuevos *namespaces* para él
    - `unshare --net`
      - Crea una Shell en un nuevo *network namespace*
    - `unshare --net ifconfig -a`
      - Lanza el proceso en un nuevo *network namespace*; al terminar el proceso se destruye el *namespace*
  - `nsenter` : Crea un nuevo proceso en un *namespace* que ya exista
    - `nsenter --net=/proc/1234/ns/net`
      - Crea una Shell en ese *network namespace*
    - `nsenter --target 1234 bash`
      - Lanza un nuevo proceso (en este caso ejecutando `bash`) en los mismos *namespaces* que el proceso 1234

upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*



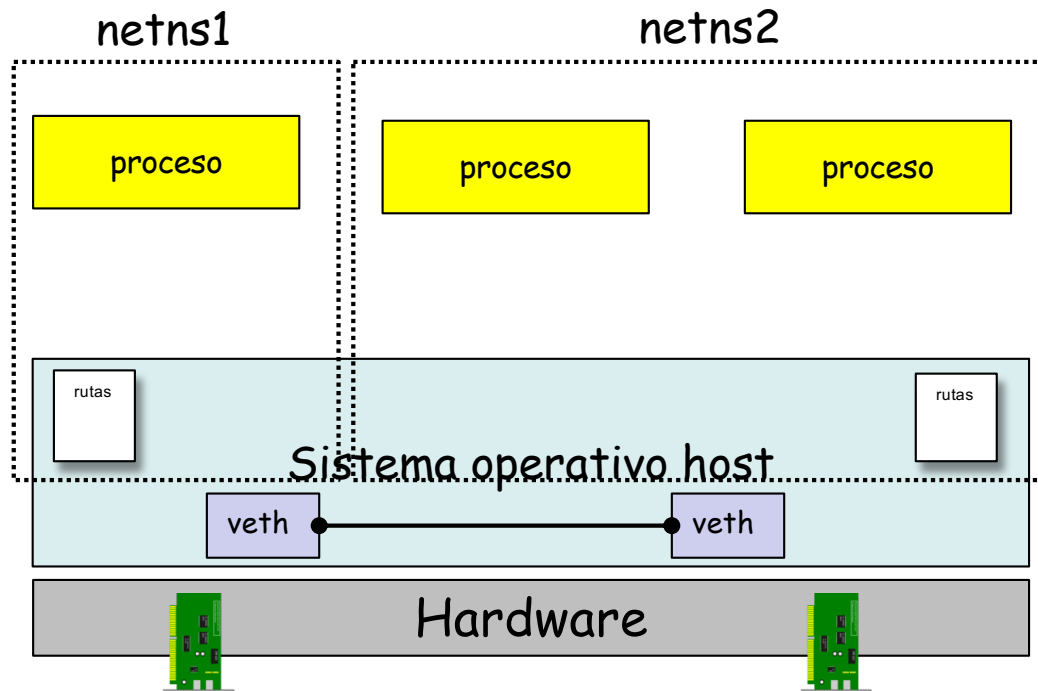
veth





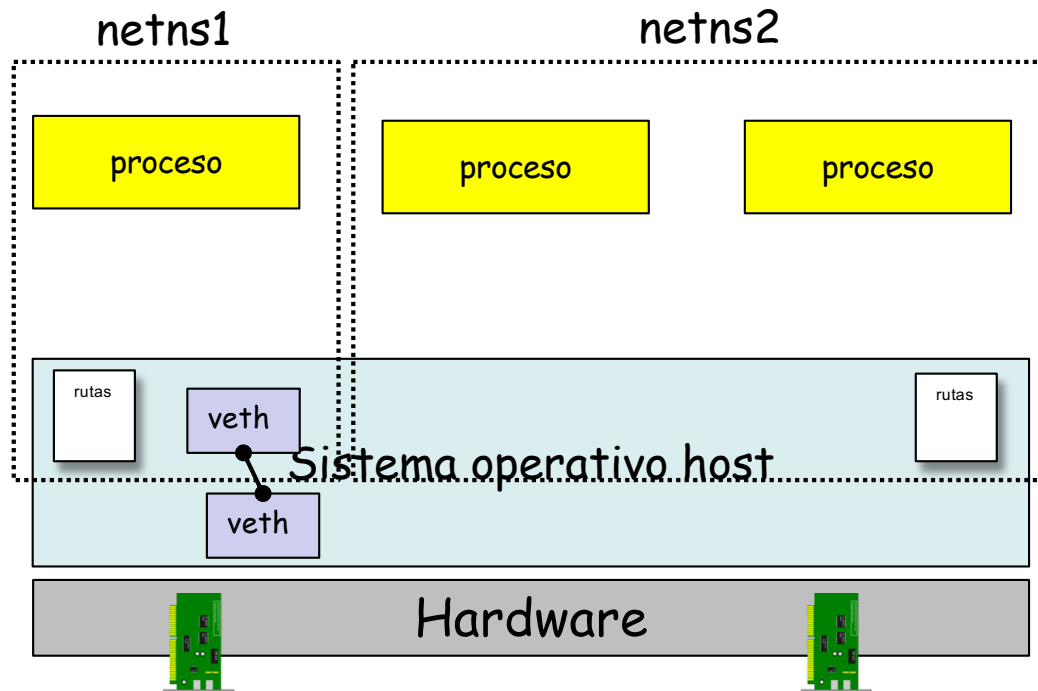
# veth - Virtual Ethernet Device

- Interfaces Ethernet enteramente software
- Se crean en parejas, la trama que se envía por uno se recibe inmediatamente en el otro
- Podemos crearlos y después asignarlos a otro netns
- (...)



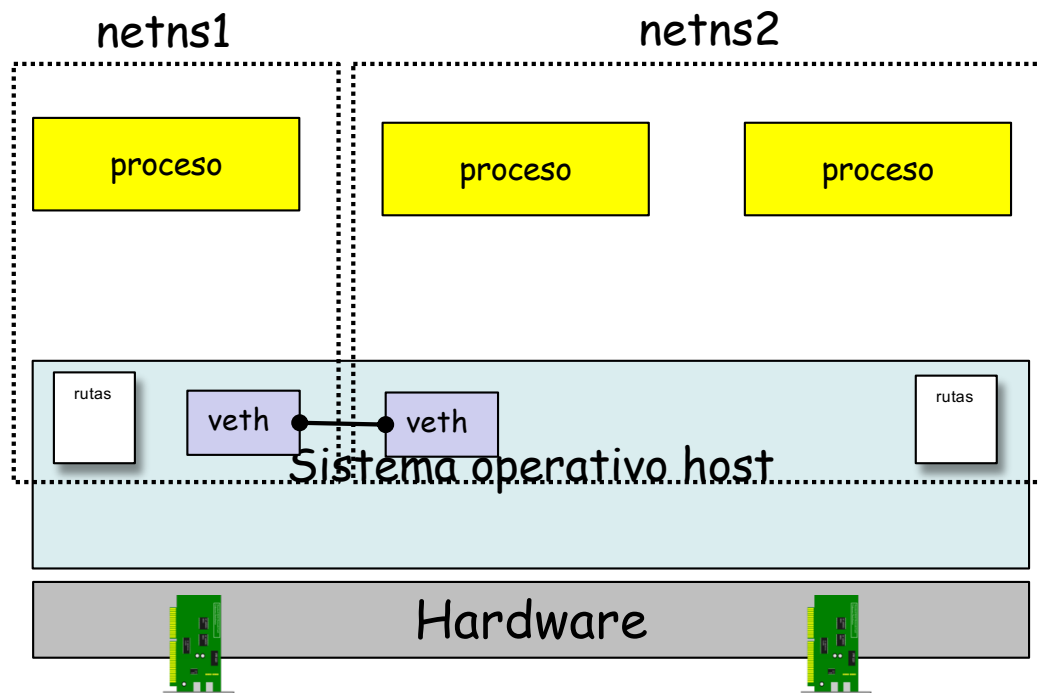
# veth - Virtual Ethernet Device

- Interfaces Ethernet enteramente software
- Se crean en parejas, la trama que se envía por uno se recibe inmediatamente en el otro
- Podemos crearlos y después asignarlos a otro netns
- Podemos usar un par de veth para comunicar el namespace raíz con el host



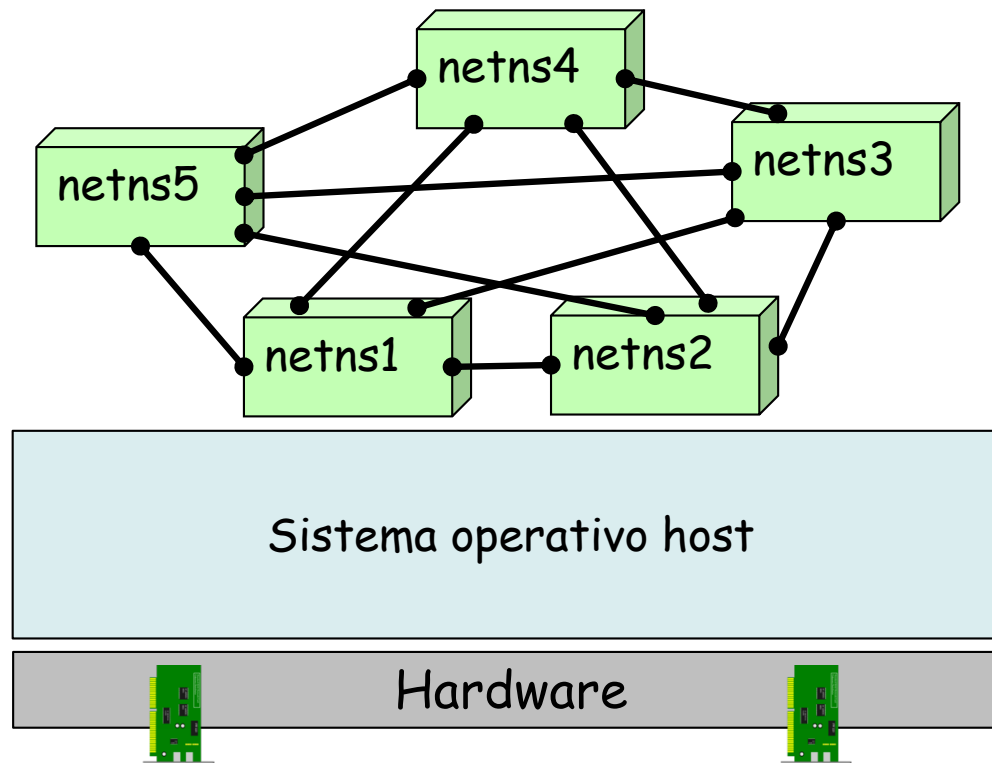
# veth - Virtual Ethernet Device

- Interfaces Ethernet enteramente software
- Se crean en parejas, la trama que se envía por uno se recibe inmediatamente en el otro
- Podemos crearlos y después asignarlos a otro netns
- Podemos usar un par de veth para comunicar dos namespaces entre si



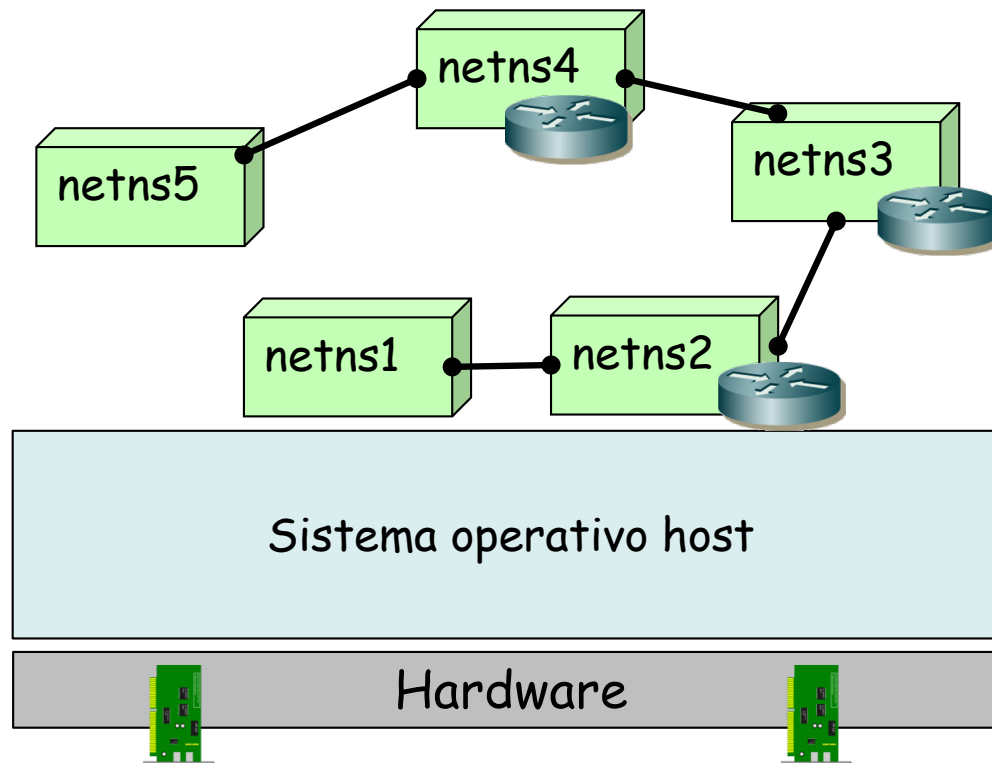
# veth - Virtual Ethernet Device

- Si tenemos muchos netns en el mismo host y queremos que se puedan comunicar todos con todos escala mal
- Necesitaríamos enlaces entre todos ellos
- Posible pero gestión compleja



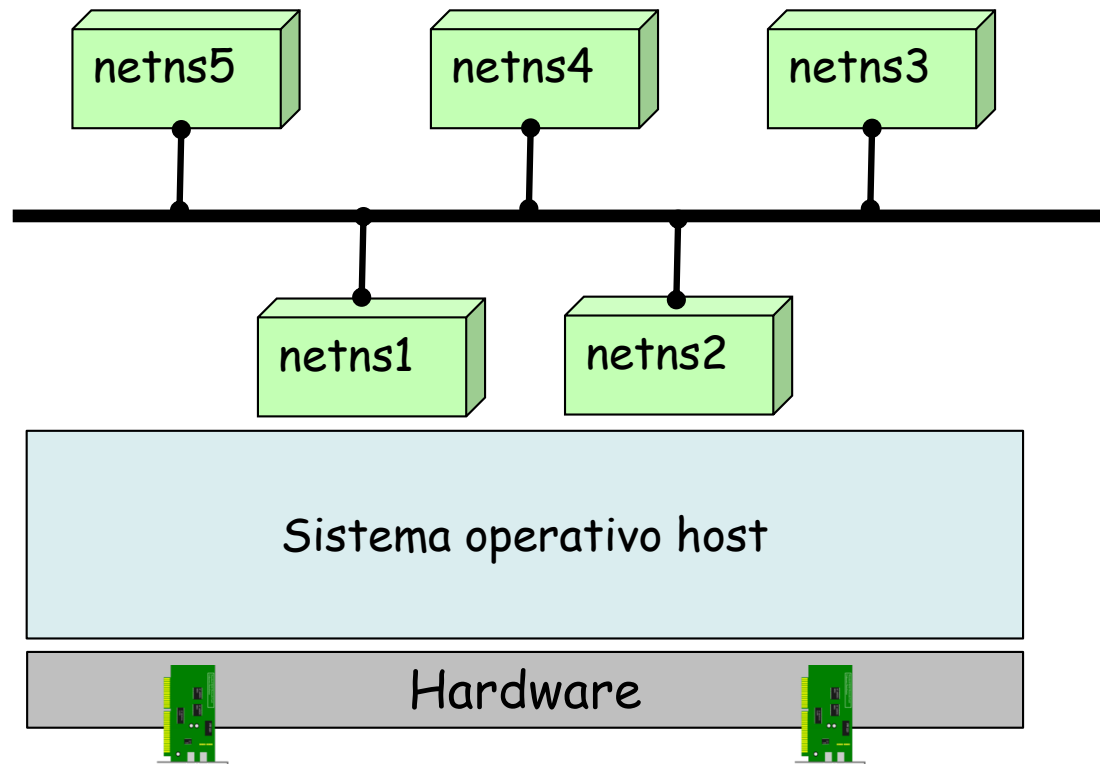
# veth - Virtual Ethernet Device

- Podrían los propios contenedores no solo usarse para aislar aplicaciones sino que además tengan que encaminar tráfico entre otros
- ¿Topología de interconexión? ¿Cálculo de rutas?
- En ocasiones sería más cómodo una LAN (...)



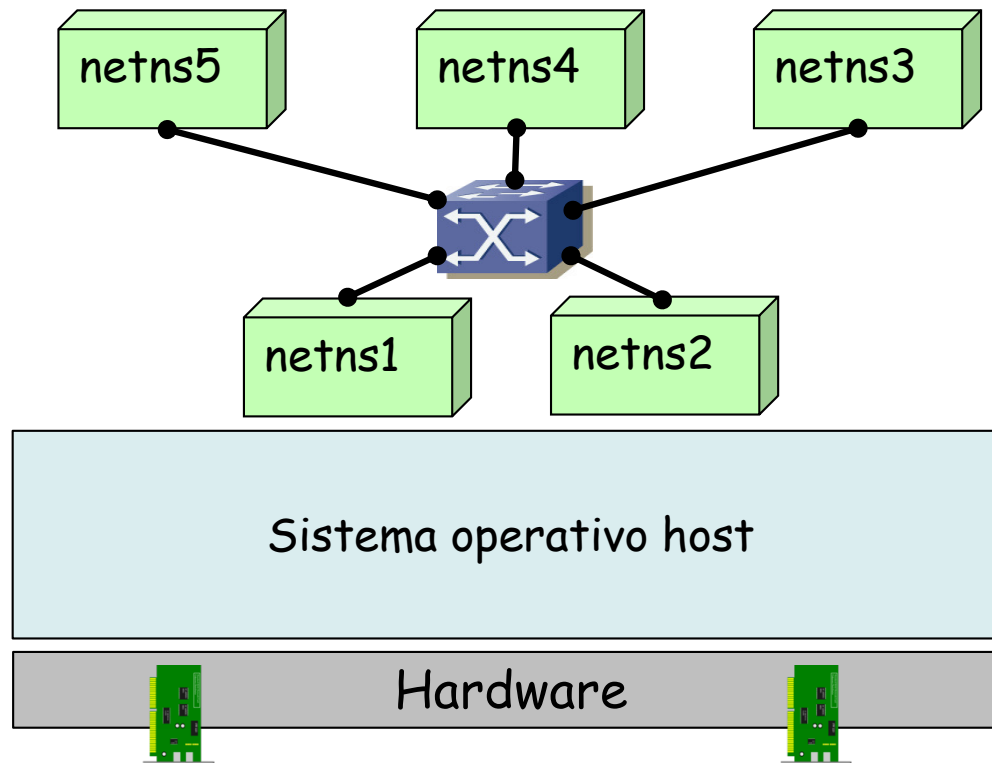
# veth - Virtual Ethernet Device

- Todos en la misma LAN Ethernet y en la misma subred IP
- ¿Cómo? (...)



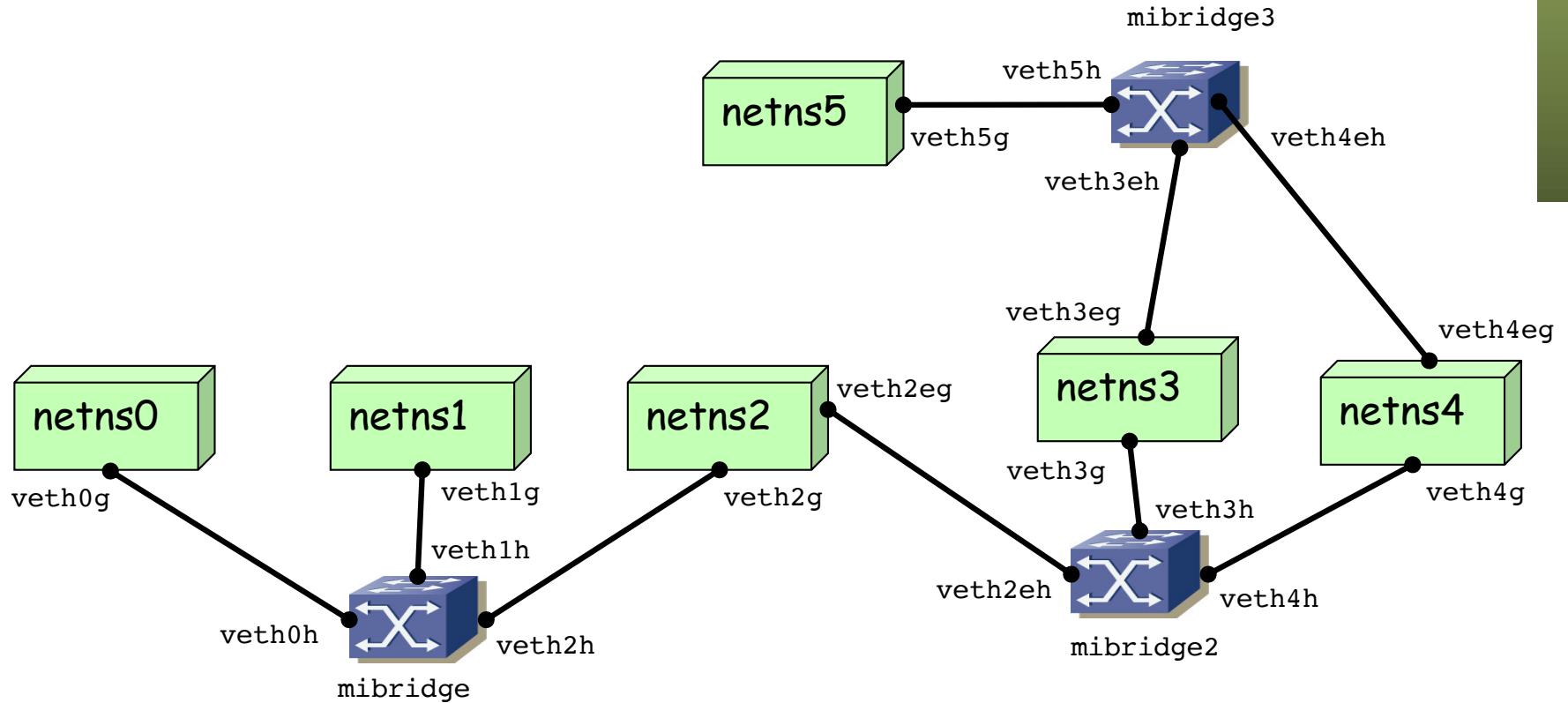
# veth - Virtual Ethernet Device

- Todos en la misma LAN Ethernet y en la misma subred IP
- ¿Cómo? Con un switch virtual



# Combinaciones

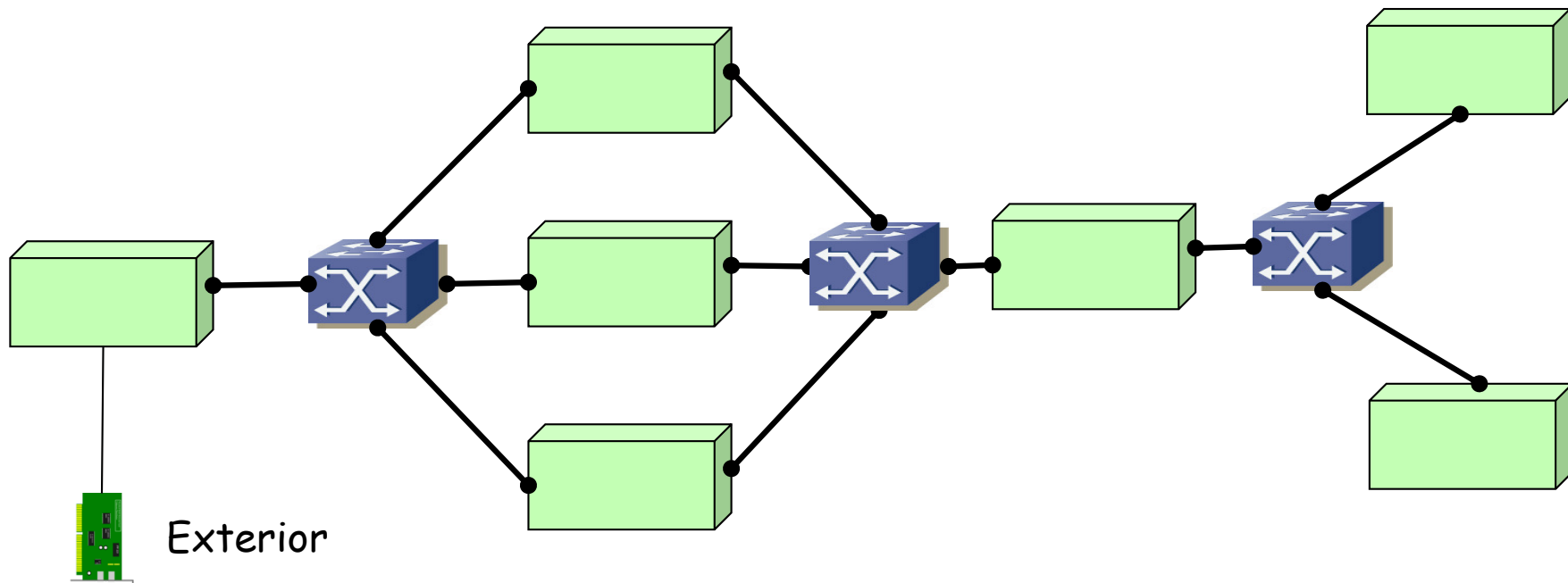
- Podría construir escenarios como éste





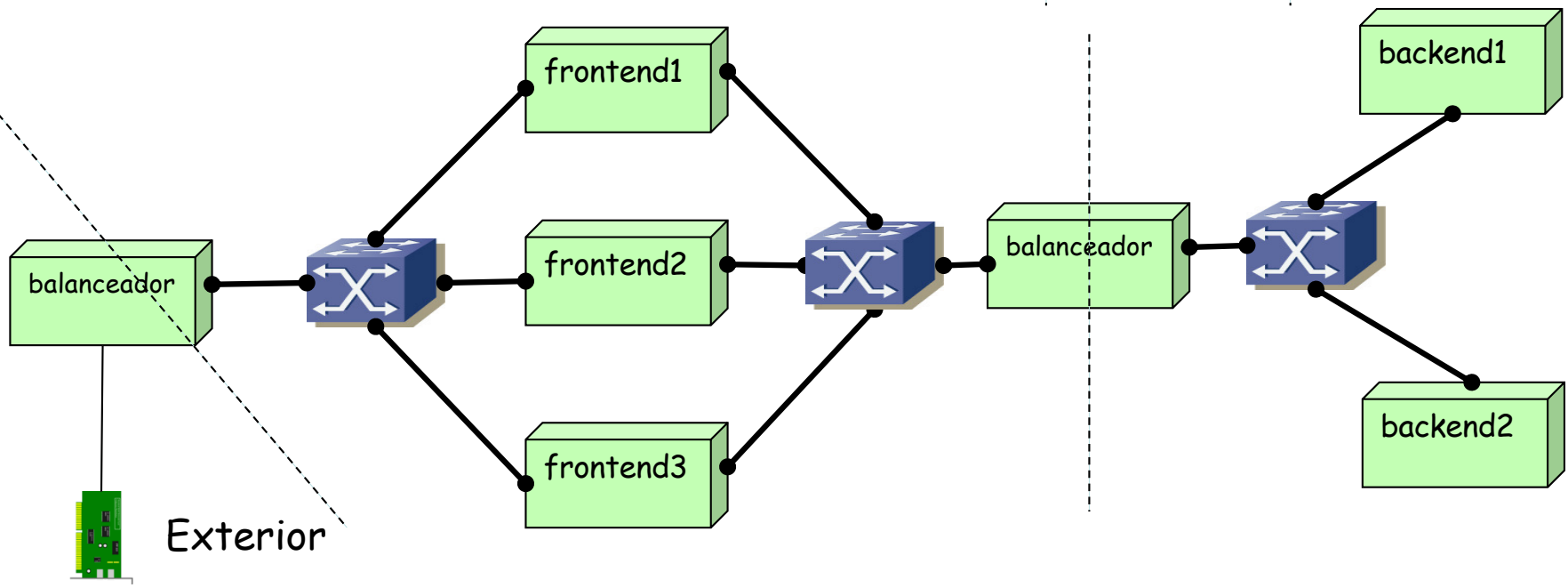
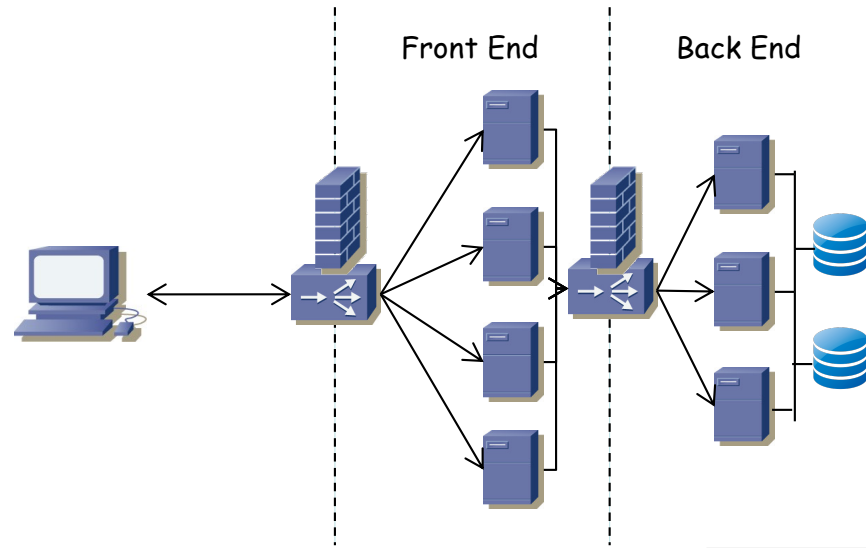
# Combinaciones

- O por ejemplo este otro



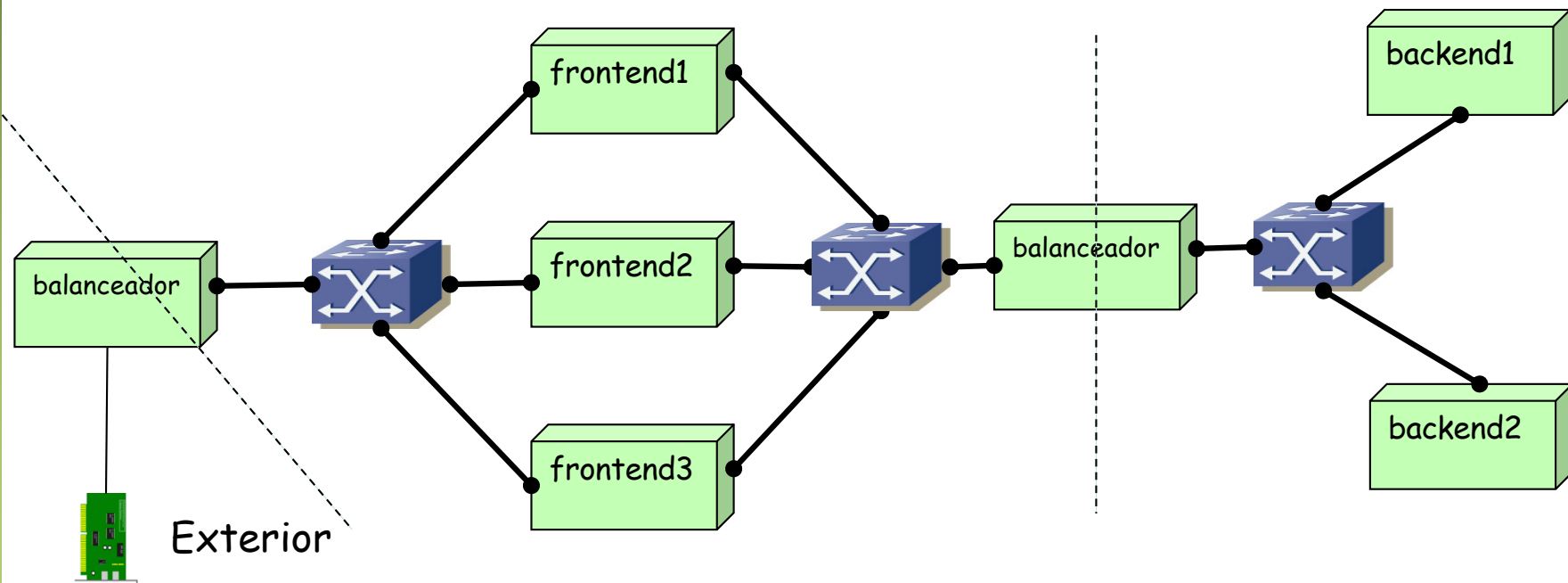
# Combinaciones

- O por ejemplo este otro
- Sospechosamente parecido



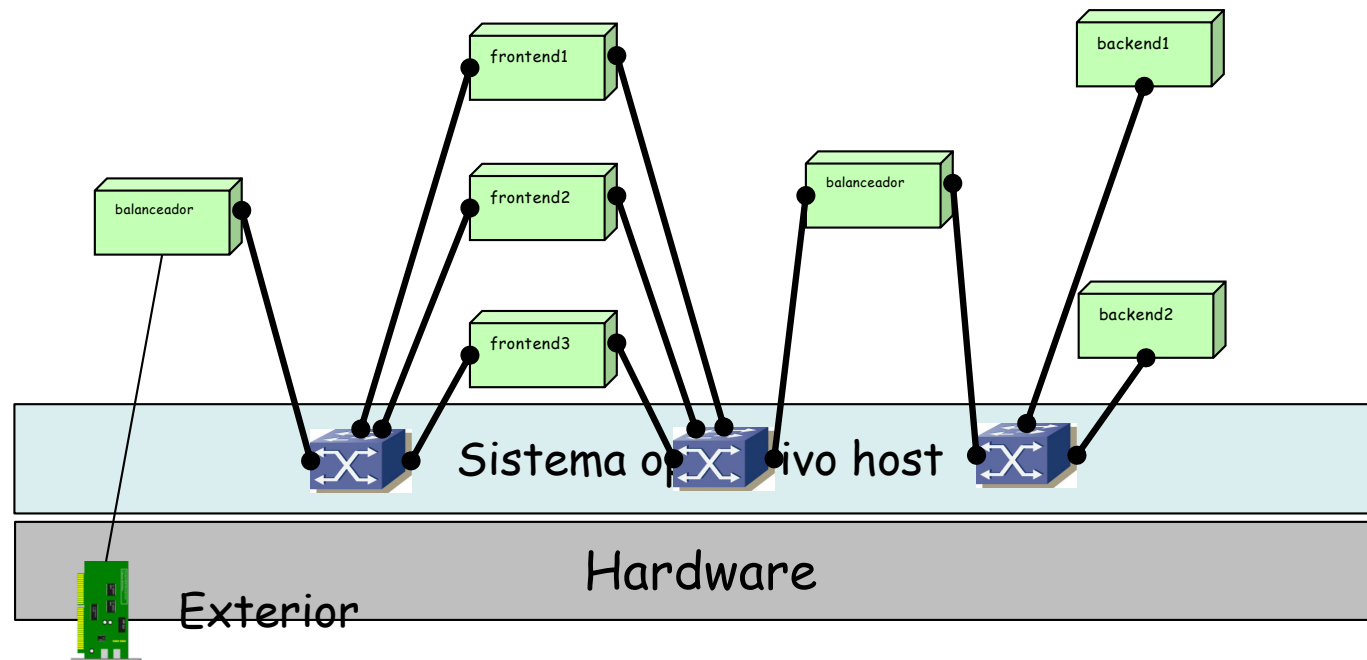
# Combinaciones

- O por ejemplo este otro
- Sospechosamente parecido
- Pero ¿qué hardware tenemos?
- (...)



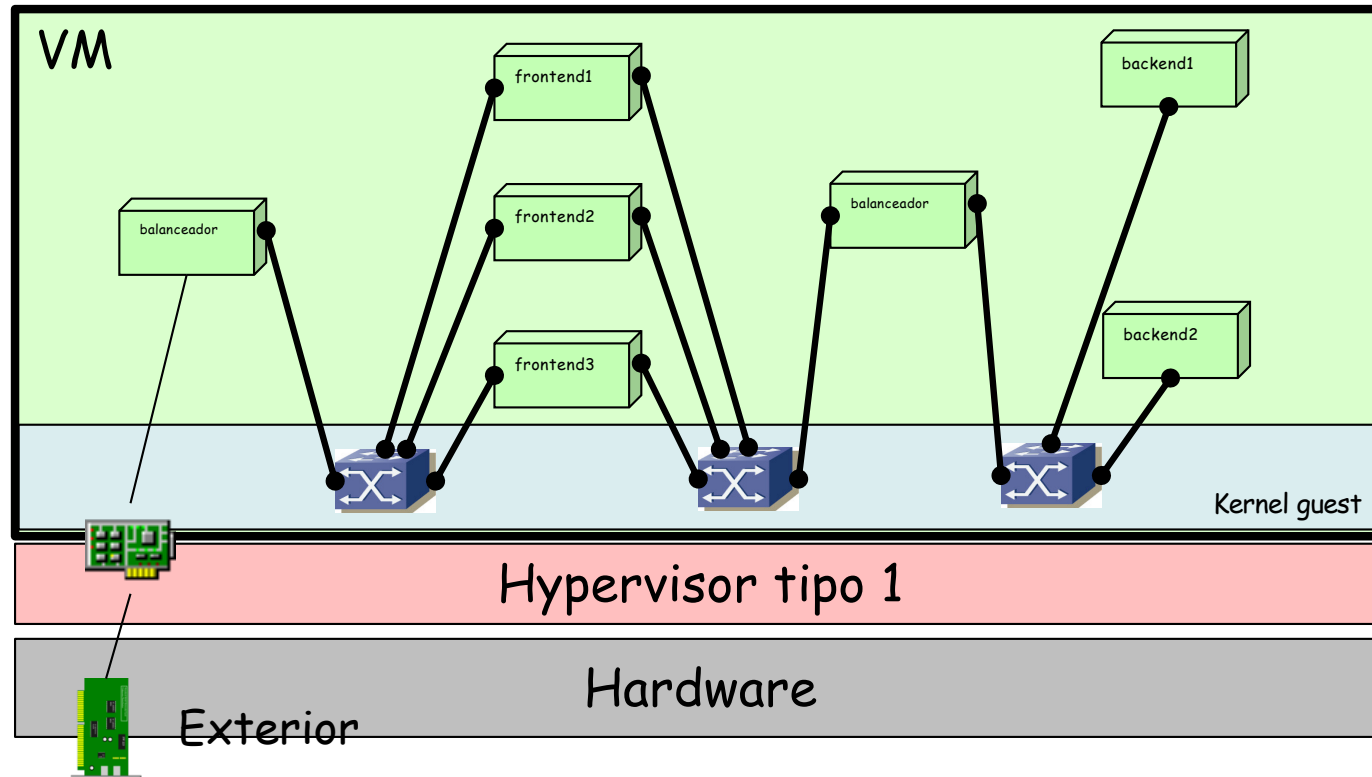
# Combinaciones

- O por ejemplo este otro
- Sospechosamente parecido
- Pero ¿qué hardware tenemos?
- Un PC con un sistema operativo Linux
- Son todo contenedores y vSwitches
- (...)



# Combinaciones

- O por ejemplo este otro
- Sospechosamente parecido
- Pero ¿qué hardware tenemos?
- Un PC con un sistema operativo Linux
- Son todo contenedores y vSwitches
- Podría estar todo dentro de una VM (!!)



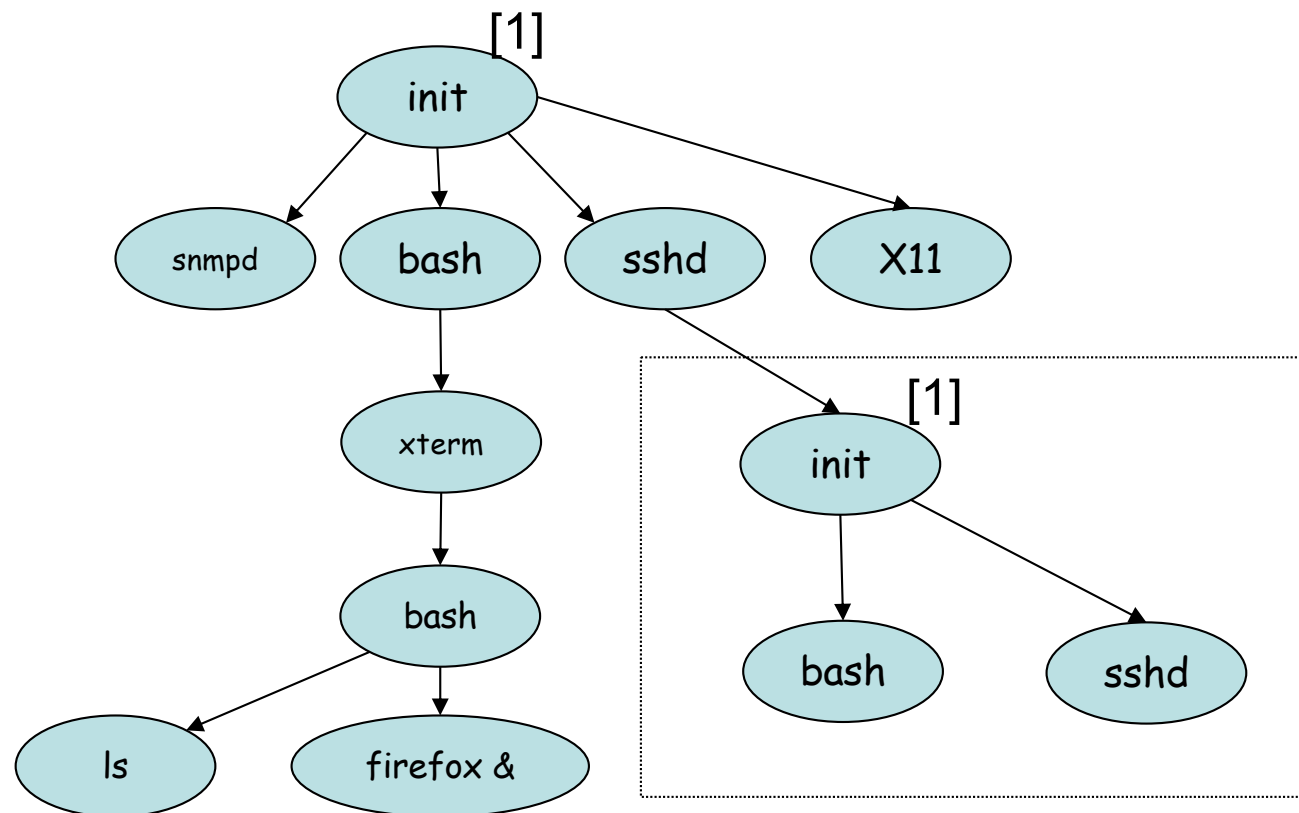
# Otros namespaces

# Linux namespaces

- Permiten que ciertos recursos del Kernel sean vistos por los procesos miembros del *namespace* como solo suyos
- Provee de una virtualización de dichos recursos
- Nos interesan principalmente los *network namespaces*
- **Otros: PID, User, cgroup, IPC, Mount, UTS**

# PID namespaces

- Permite crear jerarquías de procesos independientes
- Un proceso y sus descendientes no ven al resto de procesos
- Tenemos un nuevo proceso “1”, aunque desde la jerarquía global no se ve con ese PID





# User namespaces

- Permite crear sets de usuarios autorizados independientes
- En el nuevo namespace hay un nuevo “root” (uid=0)
- Se mapean los usuarios del namespace en usuarios del padre

upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*

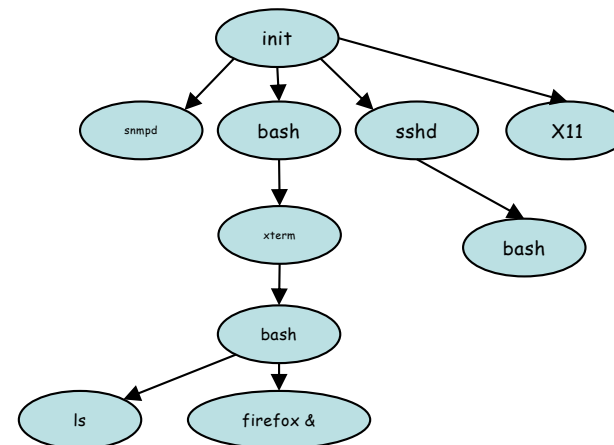


cgroups



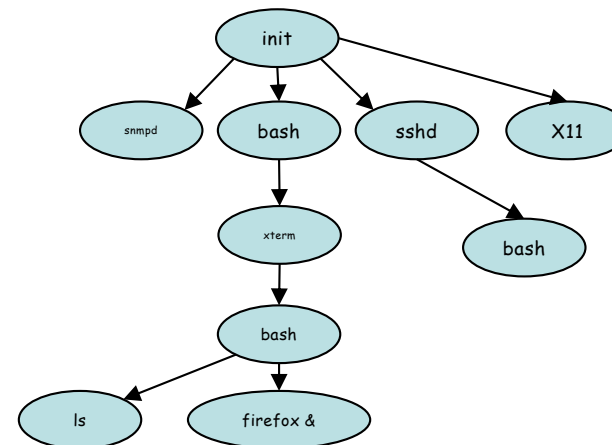
# Precursores: nice

- La utilidad *nice* permite especificar un parámetro de prioridad en el *scheduling* de un proceso
- Es común a todos los sistemas Unix
- En Linux también *ionice* permite algo parecido con el acceso a dispositivos de entrada/salida (discos)
- Varios sistemas Unix implementan más de un *scheduler* (por ejemplo *priorities* o *round-robin*)



# Linux cgroup

- Control groups
- Permiten limitar el uso de recursos hardware que hacen los procesos
- Versión 2 desde Linux 4.5 (diferentes controles en v1 y v2)
- Un cgroup puede aplicar a un proceso o a un grupo de procesos, compartiendo ese límite
- Los descendientes de un proceso heredan su pertenencia a grupos



# Cgroups v1



- **cpu**
  - Permite controlar el uso de tiempo de CPU
- **cpuset**
  - Permite asociar un grupo de procesos a una CPU
- **memory**
  - Limita la cantidad de memoria para procesos, kernel y swap
- **devices**
  - Controla la creación y el acceso a devices (por ejemplo HDs)
- **net\_cls**
  - Permite clasificar los paquetes creados por un cgroup de cara a su planificación (tc)
- **blkio**
  - Limita los accesos a dispositivos de bloques

upna

Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

**Redes de Nueva Generación**  
*Área de Ingeniería Telemática*

# Container runtimes

# Contenedores y runtimes

- Los contenedores se construyen con varios de estos mecanismos
- Podemos no necesitar y no usar todos ellos
- Por ejemplo:
  - Independizar el *network namespace* pero no independizar los PIDs
  - O independizar los *namespaces* pero no limitar recursos
  - O limitar uso de CPU a un grupo de procesos pero no independizar *namespaces*
- Los *container runtimes* son utilidades (programas) para crearlos en base a estas funcionalidades en el Kernel
- Simplifican la gestión de los contenedores
- Cada uno tiene su propia visión de cómo quiere que sean los contenedores

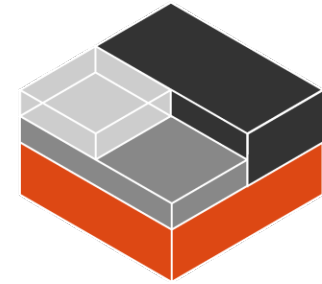


# System vs Application containers

- System containers
  - El volumen (espacio en disco) del contenedor es toda una distribución de Linux sin el kernel (Alpine, CentOS, Ubuntu, Debian, Fedora, OpenWRT,...)
  - Librerías, programas, utilidades, Shell, su propio init
  - Corren un init (o similar) con PID 1 y los demonios típico de esa distribución
  - Parecen VMs
  - Ejemplo: LXC/LXD
- Application containers
  - Corren un solo programa en cada contenedor
  - Ejemplo: Docker



# Ejemplo: LXC



- <https://linuxcontainers.org>
- Un demonio (lxd) crea los contenedores
- Una utilidad (lxc) para comunicarse con el demonio
- Repositorios de imágenes de contenedores
- Por defecto crea un bridge para la comunicación entre contenedores
- Añade reglas de NAT en el kernel

# System vs Application containers

- System containers
  - El volumen (espacio en disco) del contenedor es toda una distribución de Linux sin el kernel (Alpine, CentOS, Ubuntu, Debian, Fedora, OpenWRT,...)
  - Librerías, programas, utilidades, Shell, su propio init
  - Corren un init (o similar) con PID 1 y los demonios típico de esa distribución
  - Parecen VMs
  - Ejemplo: LXC/LXD
- Application containers
  - Corren un solo programa en cada contenedor
  - Ejemplo: Docker



# Runtimes

- runc, crun, containerd, CRI-O, kata-runtime, LXC/LXD, Docker
- OpenVZ, Firecracker, gVisor, runnc, libcontainer, podman
- runV, Clear Containers, rkt, virtcontainers
- Algunos incluyen un demonio que crea y gestiona los contenedores y utilidades para comunicarse con el demonio
- En sistemas no-Linux se puede emplear un hypervisor para virtualizar un Kernel de Linux (WSL2 o HyperKit)
- Muchos cumplen con la runtime-spec de OCI (...)



podman



cri-o



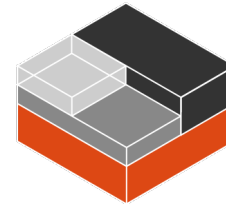
rkt



docker



kata  
containers



Firecracker

# Open Container Initiative



- Proyecto de la Linux Foundation desde 2015
- *The OCI currently contains two specifications: the Runtime Specification (runtime-spec) and the Image Specification (image-spec). The Runtime Specification outlines how to run a “filesystem bundle” that is unpacked on disk. At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime.*

## Our Members



# Imágenes

- OCI Image Specification
- Contiene *layers*
- Cada una es básicamente un *tar* de los ficheros diferentes



upna

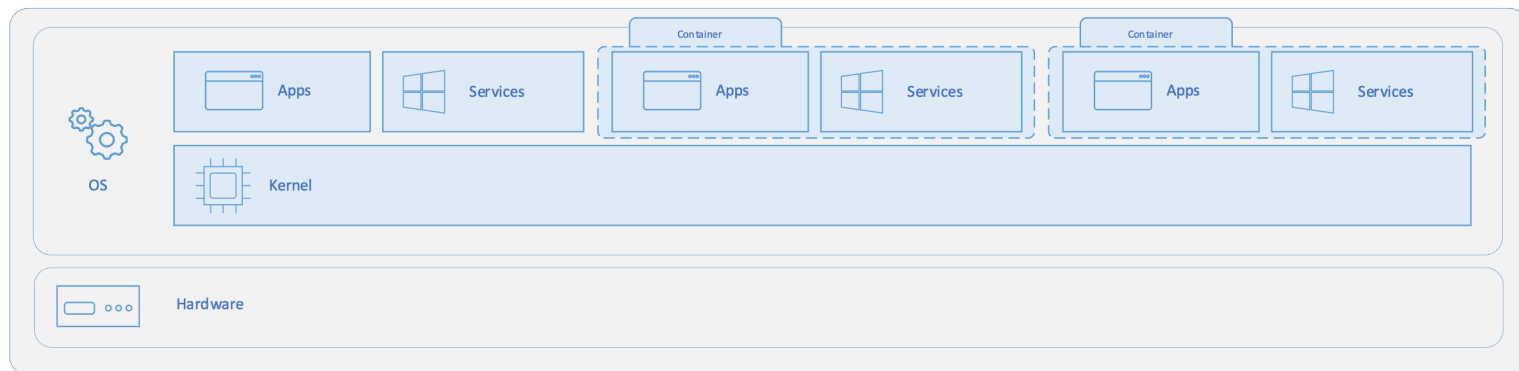
Universidad Pública de Navarra  
Nafarroako Unibertsitate Publikoa

Redes de Nueva Generación  
*Área de Ingeniería Telemática*

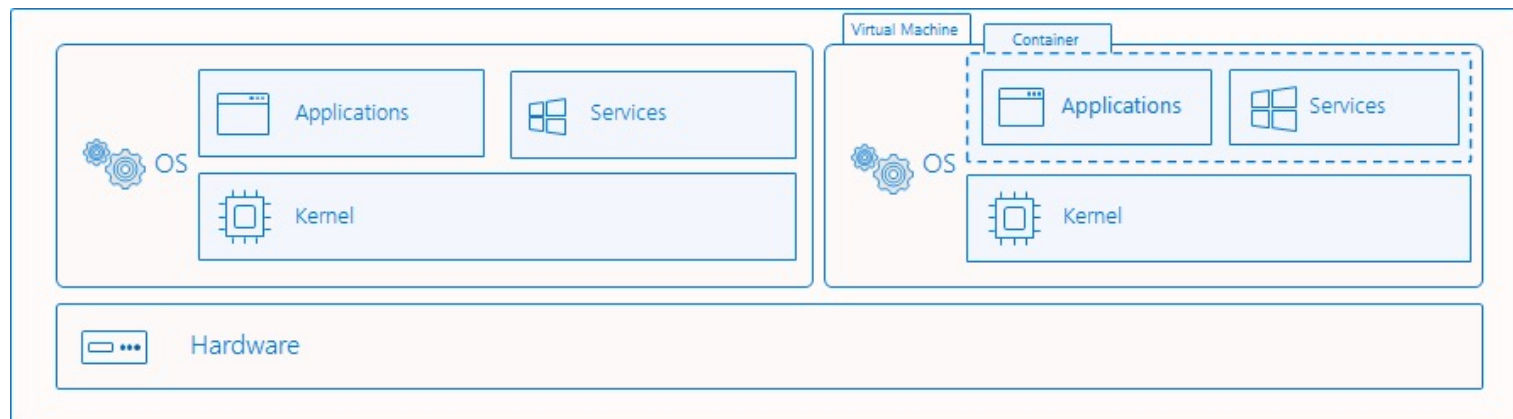
# OS virtualization en Kernel Linux

# Windows containers

- Windows 10 con Hyper-V o Windows Server
- Emplea Docker para su gestión
- Se pueden desplegar en Azure (Azure Kubernetes Service)
- Pueden ser Windows-based o Linux-based (virtuliza un Kernel Linux en una Ubuntu)
- Process isolation:



- Hyper-V isolation:



# Orchestration

- Software para la gestión de contenedores
- Vigila el correcto funcionamiento de los contenedores para reiniciarlos
- Controla el networking (incluyendo balanceadores)
- Ejemplos: Docker Swarm, Kubernetes, Apache Mesos ...

