

Resumen Colaborativo SO

Tema 1: Kernel

Sistema operativo

Es un **software que actúa como intermediario** entre el **usuario** de una computadora y su **hardware**.

- **Gestiona el HW**
- **Controla la ejecución de los procesos**
- **Interfaz entre aplicaciones y HW**
- **Actúa como intermediario entre un usuario de una computadora y el HW de la misma**

Perspectiva del usuario:

- Se genera una **abstracción de la arquitectura**, el SO "oculta" el **HW** y le da a los **programas abstracciones más simples** de manejar. Brinda **Comodidad** y "amigabilidad".

Perspectiva desde la administración de recursos:

- El **SO administra los recursos de HW** de uno o más **procesos**, brinda un **conjunto de servicios** al usuario y se encarga del **manejo** de la memoria secundaria, I/O, ejecución simultánea de procesos y multiplexación en tiempo y espacio.

Objetivos:

- **Comodidad:** Hacer más fácil el uso del hardware
- **Eficiencia:** Hacer un uso más eficiente de los recursos del sistema
- **Evolución:** Permitir la introducción de nuevas funciones al sistema sin interferir con funciones anteriores

Kernel

Porción de código que se encuentra en memoria principal y se encarga de la administración de los recursos. Implementa servicios esenciales:

- **Manejo de memoria** → Funciones Principales.
- **Manejo de la CPU** → Funciones Principales.

- Administración de procesos
- Comunicación y Concurrency
- Gestión de la E/S

Caracterizaciones:

- **Monolítico:** Incluye todos los servicios del sistema operativo en un solo bloque de código que se ejecuta en modo supervisor
- **Microkernel:** Minimiza la cantidad de código que se ejecuta en modo supervisor con el fin de hacerlo más liviano respecto a un monolítico
- **Híbrido:** Combina características de los Kernels monolíticos y Microkernel
- **Exokernel**
- **Tiempo real**

Kernel linux

El de Linux es Monolítico Híbrido porque la base es la de un Kernel Monolítico, pero provee capacidades para cargar y descargar módulos dinámicamente.

Es responsable de facilitar a los procesos acceso seguro al hardware, para ello utiliza una interfaz conocida como "llamadas al sistema". Los procesos de usuario son clientes del Sistema Operativo. El Kernel gestiona y atiende los requerimientos de los distintos procesos siguiendo un criterio de "equidad"

- El kernel se ejecuta en modo supervisor
- Los procesos en modo usuario

El Kernel de GNU/Linux está dividido en **subsistemas**, cada subsistema es mantenido por uno o más responsables que aceptan o no parches o pull requests de los desarrolladores.

El kernel de Linux también adopta un **enfoque modular**. Los módulos del kernel son fragmentos de código que pueden ser cargados y descargados dinámicamente en el kernel mientras el sistema está en funcionamiento. Esto permite extender la funcionalidad del kernel sin necesidad de recompilar todo el kernel o reiniciar el sistema. Esta modularidad ofrece **flexibilidad y eficiencia**. En cuanto a la **portabilidad**, el kernel de Linux está diseñado para soportar una amplia variedad de arquitecturas de hardware utilizando una misma estructura de código fuente. Esto se logra mediante la **abstracción del hardware**. El kernel está escrito principalmente en lenguaje C, con algunas partes en Assembler para instrucciones de bajo nivel y específicas de la arquitectura, y experimentalmente en Rust para algunos módulos.

Apoyo del hardware

Interrupción de Clock: Se debe evitar que un proceso se apropie de la CPU

Protección de la Memoria: Se deben definir límites de memoria a los que puede acceder cada proceso (registros base y límite)

Modos de Ejecución: Define limitaciones en el conjunto de instrucciones que se puede ejecutar en cada modo

- El bit en la CPU indica el modo actual
 - Cuando se arranque el sistema, arranca con el bit en modo supervisor.
 - Cada vez que comienza a ejecutarse un proceso de usuario, este bit se DEBE PONER en modo usuario mediante una interrupción
- Las instrucciones privilegiadas deben ejecutarse en modo Supervisor
- En modo Usuario, el proceso puede acceder sólo a su espacio de direcciones

Compilar/recompilar

RAZONES

- Soportar nuevos dispositivos
- Agregar mayor funcionalidad (soporte de nuevos filesystems)
- Optimizar funcionamiento de acuerdo al sistema en el que corre
- Adaptarlo al sistema donde corre (quitar soporte de hardware no utilizado)
- Corrección de bugs (problemas de seguridad o errores de programación)

¿Qué necesitamos?

- **gcc:** Compilador de C
- **make:** Herramienta que genera ejecutables u otros archivos a partir de archivos fuente. Para hacer esto, lee las instrucciones de un archivo llamado Makefile que contiene reglas que definen cómo generar un archivo específico, llamado "target" (objetivo).
- **binutils:** assembler, linker
- **libc6:** Archivos de encabezados y bibliotecas de desarrollo
- **ncurses:** bibliotecas de menú de ventanas (solo si usamos menuconfig)
- **initrd-tools:** Herramientas para crear discos RAM

Proceso

1. Obtener el código fuente.
2. Preparar el árbol de archivos del código fuente.

3. Configurar el kernel
 - `$ make menuconfig`
4. Construir el kernel a partir del código fuente e instalar los módulos.
 - `$ make -jX`
 - `$ make modules_install`
5. Reubicar el kernel.
 - `$ make install`
6. Creación del `initramfs`: Su funcionalidad principal es contener los ejecutables, los drivers y los módulos que son necesarios para lograr iniciar el sistema
7. Configurar y ejecutar el gestor de arranque (GRUB en general): no lo detecta automáticamente al nuevo kernel
8. Reiniciar el sistema y probar el nuevo kernel.
 - `$ reboot`

Módulo

Un módulo del kernel es un **fragmento de código** que puede cargarse o descargarse en el mapa de memoria del Sistema Operativo (Kernel) bajo demanda.

Su **funcionalidad principal** es permitir **extender la funcionalidad del Kernel** en "caliente", es decir, **sin necesidad de reiniciar** el sistema. Todo el código de un módulo se ejecuta en modo Kernel (privilegiado)

Ventajas de compilar una funcionalidad como **módulo** en lugar de como **built-in**:

- **Flexibilidad:** Se puede cargar solo cuando sea necesario, ahorrando memoria cuando no se usa.
- **Actualización sin reinicio:** Permite agregar o actualizar funcionalidades sin recompilar ni reiniciar el kernel.
- **Menor tamaño del kernel base:** El kernel compilado es más pequeño, ya que no incluye todas las funcionalidades de forma estática.
- **Facilidad de depuración y prueba:** Es más fácil probar nuevas versiones de un módulo sin recompilar todo el kernel.
- **Soporte para hardware opcional:** Ideal para drivers de hardware que no siempre están presentes.

Comandos y Funciones principales usados:

- **lsmod:** Muestra una **lista de todos los módulos** que están actualmente cargados en el kernel.

- **modprobe:** Es la herramienta más utilizada para cargar o descargar módulos de manera inteligente, ya que se encarga de gestionar las dependencias entre módulos.
- **dmesg:** Muestra los mensajes del buffer del kernel, donde se registran las actividades de carga y descarga de módulos. Es útil para ver si un módulo se cargó correctamente o si hubo algún error.
- **module_init(init_function):** se ejecuta automáticamente cuando un módulo ha sido cargado para inicializarlo, se asignan recursos, registran dispositivos, etc.
- **module_exit(exit_function):** se ejecuta automáticamente cuando un módulo es descargado, libera recursos, desregistrar dispositivos, etc.

Parche

Un parche del kernel es un mecanismo que permite aplicar actualizaciones sobre una versión base. Se basa en archivos diff (archivos de diferencia), que indican qué agregar y qué quitar.

Make

Menús para realizar la configuración de opciones de compilación de un kernel

1. make config

- Interfaz: Texto, secuencial.
- Requisitos: Ningún software adicional.
- Ventajas:
 - Funciona en cualquier terminal.
 - Muy básico y universal.
- Desventajas:
 - Tedioso y lento.
 - No permite modificar fácilmente respuestas anteriores.
 - Propenso a errores para usuarios inexpertos.

2. make xconfig

- Interfaz: Gráfica (basada en ventanas).
- Requisitos: Entorno gráfico (X11) y bibliotecas gráficas.
- Ventajas:
 - Intuitivo y visual.
 - Permite navegar fácilmente entre opciones y buscarlas.
- Desventajas:

- No funciona en entornos sin interfaz gráfica (como servidores).

3. **make menuconfig**

- Interfaz: Texto interactivo con menús (ncurses).
- Requisitos: Librería ncurses.
- Ventajas:
 - Equilibrio entre usabilidad y disponibilidad.
 - Navegación jerárquica y posibilidad de búsqueda.
 - Apto para terminales sin entorno gráfico.
- Desventajas:
 - Puede requerir un poco de aprendizaje inicial.

Comandos

make menuconfig: Lanza una interfaz de menú en terminal (ncurses) para configurar opciones del kernel.

make clean: Elimina archivos generados por compilaciones anteriores.

make -jN: Compila el kernel y sus módulos usando N hilos de CPU para acelerar el proceso.

make modules: (Antiguo) Compila solo los módulos del kernel.

make modules_install: Instala los módulos compilados en /lib/modules/\$(uname -r)/.

make install: Instala el kernel compilado en el sistema, copiando los archivos a /boot/.

initramfs

El **initramfs** es una imagen comprimida cargada en memoria al arrancar Linux. Contiene los archivos y módulos necesarios para iniciar el sistema antes de montar el sistema de archivos raíz.

¿Cuándo se genera o actualiza?

- Al instalar un nuevo kernel.
- Al actualizar o cambiar módulos del kernel.
- Con actualizaciones del sistema (según la distribución).

¿Qué debe incluir mínimamente?

- Controladores para acceder al disco (SATA, SCSI, NVMe).

- Módulo del sistema de archivos raíz (ej. ext4).
- Módulos adicionales si se usa LVM o cifrado (ej. dm-mod, dm-crypt).
- Herramientas básicas: shell, scripts de arranque, mount, modprobe.

GRUB

Una vez que se compila e instala un **nuevo kernel**, es necesario **reconfigurar** el gestor de arranque (como GRUB) para que pueda **reconocer** y **ofrecer** la opción de **arrancar** con este nuevo kernel.

Tema 2: Llamadas al sistema

Definición y función

Es el mecanismo utilizado por un proceso de usuario para solicitar un servicio al Sistema Operativo.

El responsable de garantizar que los procesos de usuario puedan ejecutarse controladamente y sin interferirse, es el SO. El SO provee una **interfaz de programación (API)** que los procesos pueden utilizar en cualquier momento para solicitar recursos gestionados por él mismo.

El SO actúa como un servidor que recibe continuamente requerimientos de sus clientes (Procesos) y determina cuál es el mejor mecanismo de satisfacerlos. Para poder llevar adelante la tarea, define un mecanismo y los parámetros que debe enviar el programador para solicitar los servicios. Del mismo modo define los valores de retorno

Ejemplo: **count=read(file, buffer, nbytes);**

- file: archivo que se quiere leer
- buffer: área de memoria en la que será leído
- nbytes: cantidad de bytes a leer

Pasos

1. meter en la pila los parámetros necesarios
2. Se invoca a la función read implementada en la librería y se comienza su ejecución

3. Read será una función sencilla que básicamente es la que indicará el número de System Call que se debe ejecutar y permitirá realizar la llamada al sistema correspondiente
4. La función read será la encargada de ejecutar el TRAP (Interrupción por Software) para cambiar de modo Usuario a modo Kernel y pasar el control al SO
 - a. Para todas las System Calls se utiliza la misma interrupción}
 - b. La forma de identificar la Syscall invocada es a través del valor del registro
5. Una vez que el SO tiene el control, verificará cuál es la llamada al sistema que debe atender y ejecutará el código correspondiente
6. Se accede al dispositivo de almacenamiento para obtener el archivo solicitado y leerlo en memoria

Categorías

- Control de Procesos
- Manejo de archivos
- Manejo de dispositivos
- Mantenimiento de información del sistema
- Comunicaciones

GNU/Linux

En GNU/Linux, las System Calls son identificadas por un número

- Pueden tener como máximo 6 (seis) parámetros
- Para x86 en 32 bits está definida en arch/x86/entry/syscalls/syscall_32.tbl
- Para x86 en 64 bits está definida en arch/x86/entry/syscalls/syscall_64.tbl
 - Permite mapear los números de syscall usados por programas a funciones reales en el kernel.
 - Si estás desarrollando o agregando una syscall personalizada, tenés que agregar una entrada acá para que el kernel la reconozca.
 - Se usa durante la compilación del kernel para generar el código de despacho de syscalls.

La primer tarea que realiza el dispatcher cuando se produce una interrupción es verificar el número en la tabla correspondiente y ejecutar las funciones asociadas

Los parámetros de la System Call deben manejarse con cuidado, dado que se configuran en el espacio de usuario:

- No se puede asumir que sean correctos
- En el caso de pasarse punteros, no pueden apuntar al espacio del Kernel por cuestiones de seguridad
- Los punteros deben ser siempre válidos
- El Kernel deberá tener acceso al espacio de usuario con APIs especiales que garanticen que se accede al espacio de direcciones de quien invocó la System Call.
 - `get_user()`, `put_user()`, `copy_from_user()`, `copy_to_user()`.

libc

API/Librería que provee el SO en **GNU/LINUX**.

Se **ejecuta en Modo Usuario** y el **código de la función implementada** en la librería es la que **hace el cambio de modo**.

La **funcionalidad** de la libc y las System Calls están definidas por el **estándar POSIX**:

- Busca proveer una **interfaz común** para lograr **portabilidad**.
- Busca que la **interacción** sea con la **API** y no con el **Kernel**.

Macro syscall

La macro syscall en sistemas Linux se utiliza para **invocar directamente una llamada al sistema** usando su número de syscall. Es una forma de hacer una llamada al sistema sin pasar por funciones wrapper (envoltorios) de la biblioteca estándar como `read()`, `write()`, etc. Es útil cuando:

- Se quiere usar una syscall que no tiene wrapper en la libc.
- Se necesita bajo nivel de control.
- Se está desarrollando o probando funciones del kernel directamente.

Parámetros

- **Number:** Es el número de la syscall que querés invocar.
- **Parámetros específicos de syscall:** Estos son los argumentos que requiere la syscall que estás llamando. Su cantidad y tipo depende de la syscall que estés usando.

Ejemplos de Macros

- **SYSCALL_DEFINE:** se utiliza para definir llamadas al sistema (syscalls) en el kernel de Linux. Cuando una syscall es definida con este macro, se **registra automáticamente** en la tabla de syscalls del kernel.

- **for_each_process(task)**: recorre todos los procesos en ejecución en el sistema.
- **for_each_thread(task, thread)**: se usa para recorrer todos los hilos de un proceso.

Herramientas

Strace: permite trazar (trace) las llamadas al sistema (syscalls) que hace un programa mientras se ejecuta. Te muestra cada vez que el programa le pide algo al sistema operativo

Ausyscall: Para saber el número de syscall correspondiente a un nombre (y viceversa). `ausyscall write - ausyscall 1`

Drivers

Un **driver** (o controlador) es un programa que actúa como intermediario entre el sistema operativo y un dispositivo de hardware. Sirve para que el **sistema operativo pueda entender y comunicarse con el hardware**, sin necesidad de que el programador conozca todos los detalles internos del dispositivo.

Es necesario escribir drivers porque cada dispositivo de hardware es distinto y el sistema operativo no puede saber de antemano cómo hablar con todos ellos. Los drivers traducen las instrucciones genéricas del sistema operativo en comandos específicos que un determinado dispositivo entiende.

En GNU/Linux, **muchos drivers están implementados como módulos del kernel**.

Ejemplo: Cuando conectas una impresora, el kernel detecta el dispositivo y carga el módulo que contiene el driver para esa impresora.

Un **bug** en un **driver o módulo** del kernel puede tener implicaciones **muy serias** como por ejemplo **comprometer la estabilidad, la seguridad, la compatibilidad y el funcionamiento completo del Sistema Operativo**.

Estructuras y Funciones

- **Major Number y Minor Number**: El **Major** identifica qué driver del kernel manejará las llamadas a este archivo. El **Minor** identifica una instancia específica del dispositivo manejado por ese driver.
- **ssize_t**: Es un tipo de dato firmado que se utiliza para representar el tamaño de una transferencia de datos.

- Valor positivo → número de bytes transferidos.
- Valor 0 → fin de archivo o sin datos.
- Valor negativo → error.
- **memory_fops:** Define qué funciones del driver deben llamarse cuando una aplicación de espacio de usuario realiza operaciones sobre el dispositivo. Es el "puente" entre llamadas al sistema y tu código.
- **register_chrdev(int major, const char *name, struct file_operations *):** Registra un dispositivo de carácter con el número mayor especificado. Asocia ese número a un conjunto de funciones (las que están en **memory_fops**). Si el número mayor es 0, el kernel asigna uno dinámicamente.
- **unregister_chrdev(int major, const char *name):** Libera el número mayor y desregistra el dispositivo del sistema.

Directorio /dev

En esencia, **/dev** contiene **representaciones del hardware y dispositivos virtuales** en forma de **archivos especiales**.

Archivos de dispositivos de bloque: Estos dispositivos manejan bloques de datos, lo que significa que leen y escriben datos en bloques de tamaño fijo (por ejemplo, discos duros, SSDs). **Es un tipo de Driver.**

Archivos de dispositivos de caracteres: transmiten datos de manera secuencial, un byte a la vez. Son más adecuados para dispositivos como terminales, puertos serie, o impresoras. **Es un tipo de Driver.**

Dispositivos especiales/Pseudodispositivos: no representan un hardware físico, pero tienen una función especial en el sistema.

Dispositivos de red: Representan interfaces de red o sockets de red, permitiendo la comunicación de datos entre sistemas.

Sockets y pipes: Son archivos utilizados para la comunicación interprocesos (IPC), permitiendo que diferentes procesos en el sistema intercambien datos.

/lib/modules/<version>/modules.dep

El archivo **modules.dep** contiene una lista de dependencias entre los módulos del kernel. Es decir, define qué módulos necesitan ser cargados antes de otros para que el sistema funcione correctamente. Esto es necesario porque algunos módulos dependen de otros para realizar su funcionalidad.

Cuando ejecutas el comando **modprobe** para cargar un módulo, el sistema consulta el archivo **modules.dep** para asegurarse de que todas las dependencias necesarias estén satisfechas. Si un módulo requiere otros módulos para funcionar, **modprobe** los carga automáticamente antes de cargar el módulo principal.

/proc/modules

Es una **vista del sistema de archivos proc** que muestra información sobre los **módulos actualmente cargados en el kernel.**

Tema 3: Threads

Procesos

Unidad básica de utilización de CPU

En los primeros sistemas operativos, cada proceso tenía un espacio de direcciones y un solo hilo de control. Un único flujo secuencial de ejecución. Se ejecuta una instrucción y cuando finaliza se ejecuta la siguiente

Para ejecutar otro proceso, se debe llevar adelante un cambio de contexto

Proceso - Thread

Proceso:

- Espacio de direcciones
- Unidad de propiedad de recursos
- Conjunto de threads (eventualmente uno)

Thread:

- Unidad de trabajo (hilo de ejecución)
- Contexto del procesador
- Stacks de Usuario y Kernel
- Variables propias
- Acceso a la memoria y recursos del PROCESO

Ventajas de thread:

- Sincronización de Procesos
- Mejorar tiempos de Respuesta
- Compartir Recursos

- Economía
- Analicemos uso de RPC, o servidor de archivos

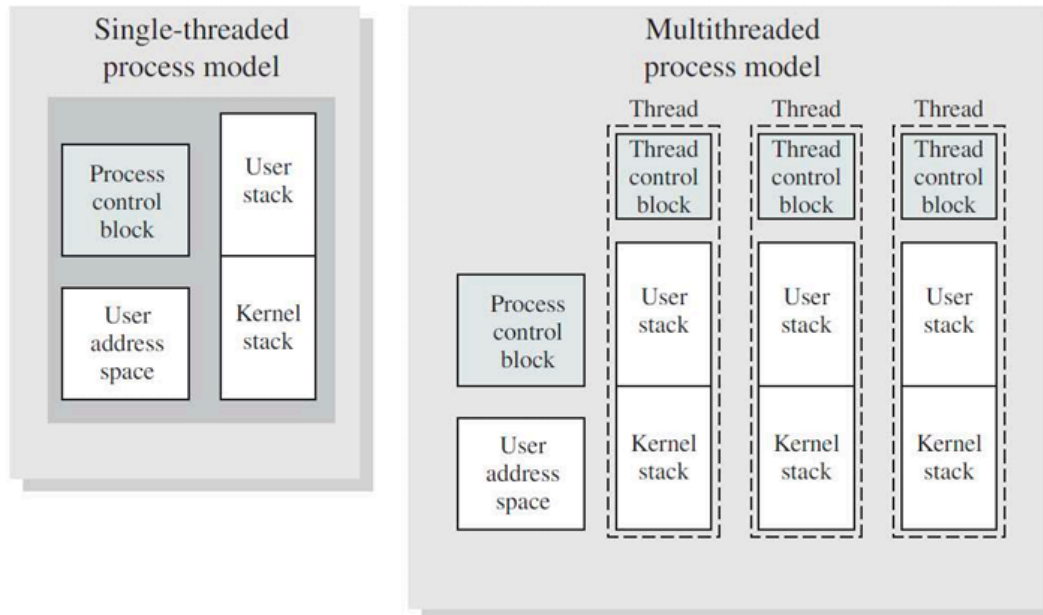
| | Proceso | Hilo |
|----------------|---------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Context-switch | El SO debe intervenir con el fin de salvar el ambiente del proceso saliente y recuperar el ambiente del nuevo | El cambio de contexto solo se realiza a nivel de registros y no espacio de direcciones. Lo lleva a cabo el proceso sin necesidad de intervención del SO |
| Creación | Implica la creación de un nuevo espacio de direcciones, PCB, PC, etc. Lo lleva a cabo el SO | Implica la creación de una TCB, registros, PC y un espacio para el stack. Lo hace el mismo proceso sin intervención del SO |
| Destrucción | El SO debe intervenir con el fin de salvar el ambiente del proceso saliente y eliminar su PCB | La tarea se realiza dentro del proceso sin necesidad de intervención del SO |
| Planificación | Es llevada a cabo por el sistema operativo. El cambio implica cambios de contexto continuos | Es responsabilidad del desarrollador quien debe planificar sus hilos. Es menos costoso, pero puede traer desventajas aparejadas |
| Protección | El SO garantiza la protección a través de distintos mecanismos de seguridad. La comunicación entre ellos implica el uso de técnicas más avanzadas | La protección debe darse desde el lado del desarrollo. Todos los hilos comparten el mismo espacio de direcciones. Un hilo podría bloquear la ejecución de otros |

Estructura de un hilo

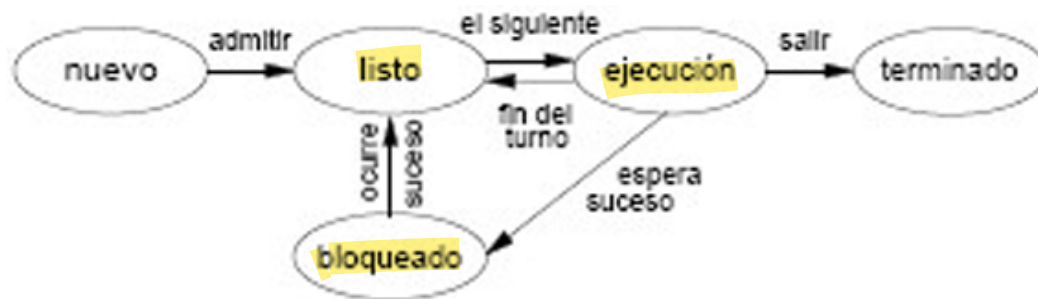
Cada hilo cuenta con:

- Un estado de ejecución
- Un contexto de procesador
- Stacks (uno en modo usuario y otro en modo kernel)
- Acceso a memoria y recursos del proceso:
 - Archivos abiertos
 - Señales
 - Código

- Todos estos datos se compartirán con el resto de los hilos del proceso.
- TCB – Thread Control Block



Estados mínimos: ejecución, listo y bloqueado



User Level Thread (ULT)

La aplicación, en modo usuario, se encarga de la gestión

Por medio de una Biblioteca de Threads

La Biblioteca deberá brindar funciones para:

Crear, destruir, planificar, etc.

El Kernel "no se entera" de la existencia de Threads.

Ventajas:

- **Intercambio entre hilos:** comparten el espacio de direcciones
- **Planificación independiente:** cada proceso los planifica como mas le conviene
- Podrían reemplazarse llamadas al sistema bloqueantes por otras que no bloqueen
- **Portabilidad:** pueden correr en distintas plataformas
- No requiere cambios para su "existencia"
- No es necesario que el SO soporte hilos

Desventajas

- No se puede ejecutar hilos del mismo proceso en distintos procesadores
- Si un hilo produce un Page Fault, todo el proceso se bloquea
- Un hilo podría monopolizar el uso de la CPU por parte del proceso
- Bloqueo del proceso durante una System Call bloqueante

Kernel Level Thread (KLT)

La gestión completa se realiza en modo Kernel

Ventajas

- Se puede multiplexar hilos del mismo proceso en diferentes procesadores
- Independencia de bloqueos entre Threads de un mismo proceso

Desventajas

- Cambios de modo de ejecución para la gestión (Planificación, creación, destrucción, etc.)

Combinaciones

- Es posible combinar ULT y KLT
- En este tipo de sistemas, la creación de hilos se realiza a nivel de usuario y los mismos son mapeados a una cantidad igual o menor de KLT.
- La sincronización de hilos en este modelo, permite que un hilo se bloquee y otros hilos del mismo proceso sigan ejecutándose
- Permite que hilos de usuario mapeados a distintos KLT puedan ejecutarse en distintos procesadores.
- Este enfoque aprovecha las ventajas de ambos tipos

Modelo uno a uno

- Cada ULT mapea con un KLT
- Cuando se necesita un ULT se debe crear un KLT
- Si se bloquea un ULT, otro hilo del mismo proceso puede seguir ejecutándose
- La concurrencia y/o paralelismo es máximo, ya que cada hilo puede correr en un procesador distinto
- Introduce un costo alto, ya que cada vez que se crea un hilo de usuario se debe crear un KLT

Modelo muchos a uno

- Muchos ULT mapean a un único KLT
- Usado en sistemas que no soportan KLT
- Si se bloquea un ULT, se bloquea el proceso

Modelo muchos a muchos

- Muchos ULT mapean a muchos KLT
- Este modelo multiplexa los ULT en KLT, logrando un balance razonable:
 - No tiene el costo del modelo 1:1
 - Minimiza los problemas de bloqueo del modelo M:1

Mecanismos de sincronización

Para KLTs

- Mutexes de Pthreads.
- Variables condición.
- Barreras.
- Semáforos.

Para ULTs:

- Mutexes provistos por la biblioteca de ULT.
- Semáforos provistos por la biblioteca de ULT.
- Otros mecanismos de sincronización que ofrezca la biblioteca de ULT.

Diferencias observadas entre fork, pthread y pth

- **fork():**
 - Crea un nuevo proceso con distinto PID y TID.

- No comparten memoria.
- El valor de una variable modificada por el hijo no afecta al padre (se usa Copy-On-Write).
- **pthread_create()** (KLT - Kernel Level Thread):
 - Comparten el mismo PID pero tienen diferente TID.
 - Comparten memoria (cambios en variables afectan a todos los hilos).
 - Utiliza la syscall clone() con flags CLONE_VM y CLONE_THREAD.
- **pth_create()** (ULT - User Level Thread):
 - El kernel no "ve" los hilos: mismo PID y TID a nivel kernel.
 - Tienen diferente PTH_ID (ID interno de la biblioteca).
 - También comparten memoria.

Memoria compartida y comportamiento

- En procesos (fork), padre e hijo tienen memoria separada.
- En hilos (KLT y ULT), todos acceden a la misma memoria: el cambio en una variable afecta a todos.

Análisis con strace

- fork() genera una syscall clone().
- pthread_create() utiliza clone() o clone3() con:
 - CLONE_VM: comparte espacio de memoria.
 - CLONE_THREAD: mismo grupo de hilos (mismo PID visible con getpid()).
- ULT no usa clone(), el kernel no los ve como hilos.

Comparación en tareas CPU-bound

- **KLT (en C):**
 - Ejecución paralela real en múltiples núcleos.
 - Mejores tiempos de ejecución.
- **ULT (en C):**
 - Solo 1 hilo real ejecutándose.
 - Más lento para tareas de cálculo intensivo.
- **Python con GIL:**
 - Solo un hilo se ejecuta a la vez.
 - No mejora el rendimiento en tareas CPU-bound.
- **Python sin GIL + KLT:**
 - Paralelismo real.

- Mejor rendimiento para tareas de cálculo.

Comparación en tareas IO-bound

- **ULT y KLT** tienen tiempos similares.
- Las operaciones bloqueantes como `sleep()` permiten interleaving incluso en **ULT**.

Funciones y IDs importantes

- `getpid()`: PID del proceso (igual para todos los hilos).
- `gettid()`: ID del hilo a nivel kernel (diferente para cada hilo KLT).
- `pthread_self()`: ID POSIX del hilo.
- `pth_self()`: ID del hilo según la biblioteca ULT (no gestionado por el kernel).

Conclusiones

- **ULT** es adecuado para tareas IO-bound o cuando no se requiere paralelismo real.
- **KLT** es mejor para tareas CPU-bound.
- Python con GIL limita el uso real de múltiples hilos.
- Python sin GIL con KLT permite paralelismo efectivo.

Tema 4: Virtualización

Virtualizar

Es una capa de abstracción sobre el hw para obtener una mejor utilización de los recursos y flexibilidad.

Es una capa abstracta que desacopla el hardware físico del sistema.

Permite que una computadora pueda realizar el trabajo de varias, a través de la **compartición de recursos de un único dispositivo de hardware**

Se aprovecha el hardware corriendo varios SO al mismo tiempo junto con sus aplicaciones

Cada SO desconoce de la existencia de otros SO

Tipos

- **Process Level:** permite lograr portabilidad entre diferentes sistemas. Java Virtual Machine (JVM)

- **Storage Level:** presenta^a una vista lógica del almacenamiento al usuario, quien no sabe donde están los datos guardados (RAID, LVM, etc.)
- **Network Level:** integra recursos de hardware de red con recursos de software
- **Operating System Level:** SO permite la existencia de varias instancias de espacio de usuario aisladas (containers)
- **System Level:** permite la creación de máquinas virtuales

Razones para virtualizar

- Tengo muchas máquinas servidores, poco usados.
- Tengo que correr aplicaciones heredadas (legacy) que no pueden ejecutarse en nuevo hw o SO.
- Tengo que probar aplicaciones no seguras.
- Tengo que crear un SO o entorno de ejecución con recursos limitados.
- Tengo que simular la computadora real, pero con un subconjunto de recursos
- Necesito usar un hw que no tengo (necesito "crear la ilusión" de hw).
- Necesito simular redes de computadoras independientes.
- Tengo que correr varios y distintos SO simultáneamente.
- Necesito hacer testeo y monitoreo de performance
- Necesito que SOs existentes se ejecuten en ambientes multiprocesadores que comparten memoria
- Necesito facilidad de migración.
- Necesito ahorrar energía (tendencias de green IT o tecnología verde)

Ventajas

- Fácil de almacenar, copiar.
- Fácil de hacer backup y restaurar.
- Simple de expandir y agregar recursos.
- Sistemas completos (aplicaciones ya configuradas, SO, hw virtual) pueden moverse de un servidor a otro rápidamente.

Componentes

Existe un software **host** (que simula) y un software **guest** (lo que se quiere simular).

Monitor de máquina virtuales (VMM)

Programa que se corre sobre el hardware para implementar las máquinas virtuales que se encarga de controlar los recursos y de la planificación de los guests

Consideraciones:

- El VMM necesita ejecutarse en modo supervisor
- El software guest se ejecuta en modo usuario
- Las instrucciones privilegiadas en los guests implican traps al VMM
- El VMM interpreta/emula las instrucciones privilegiadas

Características de una MV

Equivalencia / Fidelidad: un programa ejecutándose sobre un VMM (Virtual Machine Monitor), también conocido como Hipervisor, debería tener un comportamiento idéntico al que tendría ejecutándose directamente sobre el hardware subyacente.

Control de recursos / Seguridad: El VMM tiene que controlar completamente y en todo momento el conjunto de recursos virtualizados que proporciona a cada guest.

Eficiencia / Performance: Una fracción estadísticamente dominante de instrucciones tienen que ser ejecutadas sin la intervención del VMM, o en otras palabras, directamente por el hardware.

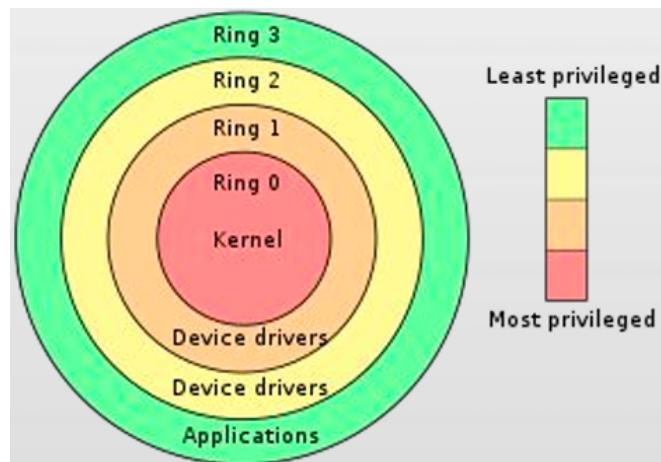
Instrucciones

- **Instrucciones inocuas o no privilegiadas:** se ejecutan nativamente
- **Instrucciones privilegiadas:** provocan una interrupción al ser ejecutadas en modo usuario
- **Instrucciones sensibles:** deben ejecutarse en modo kernel (E/S, configuración del hard (MMU), administración de interrupciones)

Para construir un VMM es suficiente con que todas las instrucciones que podrían afectar al correcto funcionamiento del VMM (instrucciones sensibles) siempre generen una excepción y pasen el control al VMM.

Las instrucciones no privilegiadas deben ejecutarse nativamente en el hardware (es decir, eficientemente).

Para que un sistema soporte virtualización, las instrucciones sensibles deben ser un subconjunto de las privilegiadas



Binary translation

Técnica usada en virtualización donde el hypervisor traduce dinámicamente las instrucciones privilegiadas del guest OS a instrucciones seguras para ejecutar en modo usuario. Permite ejecutar sistemas operativos no modificados, pero con overhead por la traducción. (Ejemplo: VMware Workstation)

Trap and emulate

El hypervisor captura (traps) las instrucciones privilegiadas del guest OS (generando una excepción) y las emula en modo kernel. Requiere soporte hardware (como Intel VT-x/AMD-V) para evitar traducción. Más eficiente que binary translation. (Ejemplo: KVM con CPUs modernas).

Hypervisors

También conocidos como VMM (Virtual Machine Monitor) es una porción de software que separa las "aplicaciones/SO" del hardware subyacente. Proveen una plataforma de virtualización que permite múltiples SO corriendo en un host al mismo tiempo.

| Hypervisor tipo 1 | Hypervisor tipo 2 |
|------------------------------------|--------------------------------|
| Se ejecuta en modo kernel (Ring 0) | Se ejecuta como un programa de |

| | |
|------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|
| | usuario en un SO host |
| Cada VM se ejecuta como un proceso de usuario en modo usuario (Ring 3) | Su función principal de interpretar un subconjunto de las instrucciones de hardware de la máquina sobre la que corre |
| El SO guest no requiere ser modificado | Debe emularse el hardware que se mapea a los SO guest |
| Existen un modo kernel virtual y modo usuario virtual | |
| Debe tener asistencia de Hardware Siempre | |

Beneficios Principales de los Hypervisors

- **Mejor aprovechamiento del hardware:** se pueden ejecutar múltiples sistemas sobre un solo equipo.
- **Aislamiento entre sistemas:** errores o fallos en una VM no afectan a las otras.
- **Portabilidad:** las VMs pueden moverse entre servidores fácilmente.

Técnicas de virtualización

Emulación

Simula hardware completo para ejecutar sistemas con arquitecturas o kernels diferentes. Permite ejecutar sistemas incompatibles (ej: QEMU emulando SPARC en x86)

- Aplicación/SO emulado ejecuta en modo usuario
- Todas las instrucciones son capturadas por el emulador
- Tiende a ser lenta

Full Virtualization

Se trata de particionar un procesador físico en distintos contextos, donde cada uno de ellos corre sobre el mismo procesador.

- Los SO guest deben ejecutar la misma arquitectura de hardware sobre la que corren
- No requiere que los guest se modifiquen

- Traducción binaria

Asistida por hardware

Utiliza extensiones del procesador (como Intel VT-x o AMD-V) para mejorar el rendimiento y permitir que el hypervisor delegue operaciones críticas directamente al hardware, eliminando la sobrecarga de emulación. Ejemplo: KVM con CPUs modernas

Paravirtualización

La paravirtualización es una técnica de virtualización donde el guest OS se modifica para interactuar directamente con el hypervisor mediante APIs optimizadas, evitando la emulación completa de hardware y la traducción binaria. Esto reduce la sobrecarga al eliminar operaciones complejas como binary translation.

El SO guest es como un proceso de usuario que hace llamadas al SO (el hypervisor). Los guests, en vez de invocar instrucciones sensibles, invocan a estas llamadas (hypercalls).

El hypervisor se transforma en un microkernel.

Decimos que un SO está paravirtualizado cuando se han eliminado, intencionalmente, algunas instrucciones sensibles (si se eliminan todas es paravirtualización completa si solo se eliminan algunas, se la llama paravirtualización parcial).

Modelos

Recompilando el Kernel del sistema guest: Los drivers y la forma de invocar a la API residen en el kernel. Es necesario instalar un sistema operativo modificado/específico.

Instalando drivers paravirtualizados: La paravirtualización es parcial (para algunas funciones y dispositivos). Generalmente utilizada para placas de red, o gráficas.

Virtualización vs. Paravirtualización

Virtualización completa o nativa puede generar problemas de performance ya que tiene que emular la totalidad del hardware.

Paravirtualización completa en kernel tiene mejor performance, menor overhead y más escalabilidad. Pero **soporta pocos SO**, pues **necesita modificar el SO original**.

Paravirtualización parcial en drivers es una solución intermedia

Virtualización vs. Emulación

Virtualización:

- **Objetivo:** Ejecutar múltiples sistemas operativos sobre el mismo tipo de hardware físico.
- **Hardware subyacente:** Se usa el mismo tipo de arquitectura que la del host (por ejemplo, x86 sobre x86).

Emulación:

- **Objetivo:** Imitar completamente un sistema diferente (hardware y software), **permitiendo ejecutar software compilado para una arquitectura distinta**.
- **Hardware subyacente:** Puede ser diferente al del sistema emulado (por ejemplo, emular una SPARCstation en una PC x86).

Tema 5: Control Groups, Namespaces y Contenedores

Chroot

Comando chroot: Es una **syscall/comando** que cambia el directorio raíz ("/") aparente para un proceso y sus hijos, restringiendo su acceso al filesystem fuera de ese directorio.

`chroot /nuevo/root /bin/bash # Ejecuta bash con "/nuevo/root" como raíz`

Finalidad:

- **Aislar procesos en un entorno controlado** ("jail").
- **Ejecutar aplicaciones con dependencias** o bibliotecas específicas **sin afectar el sistema principal**.
- Usado como **base temprana para contenedores** (antes de namespaces/cgroups)

Aislamiento limitado:

- **Solo protege a nivel de filesystem**
- **No aísla procesos, usuarios, dispositivos o red**

Puntos débiles:

- Los procesos pueden escapar si obtienen root /proc y /dev mal configurados pueden filtrar información del host
- No hay control de recursos (CPU, memoria)

Usos adecuados:

- Aislamiento básico de aplicaciones
- Entornos de construcción (build environments) Recuperación de sistemas

Alternativas más seguras:

- Containers (Docker, LXC) que combinan chroot con namespaces y cgroups
- Máquinas virtuales para aislamiento completo

Práctica

- Preparación del entorno **chroot**:

```
mkdir -p /root/sobash/{bin,lib,lib64}
cp /bin/bash /root/sobash/bin/
ldd /bin/bash
cp /lib/x86_64-linux-gnu/* /root/sobash/lib/
```

- Con **ldd** vemos si faltan librerías si falla chroot /root/sobash.
- **pwd** muestra / (aunque físicamente está en /root/sobash).
- Escape posible: Si el proceso tiene **permisos root, puede remontar / y escapar**.
- Conclusión del aislamiento
 - Ventaja: Simple y rápido para aislamiento básico de filesystem.
 - Desventaja:
 - No protege contra escapes si el proceso tiene root.
 - No limita recursos (necesita combinarse con cgroups).

Cgroups

Característica del kernel de linux que permite organizar los procesos en **grupos jerárquicos**, donde recursos (**subsistemas**) como CPU, memoria, I/O, entre otros, pueden ser **limitados** en recursos, **monitoreados** en su uso de recursos, **priorizados** en su acceso a recursos y **controlado** en su estado (pausar/reiniciar grupos).

Cgroups v1

- Cada controlador (CPU, memory, I/O, etc.) puede montarse por separado.
- Hay múltiples jerarquías: cada recurso puede tener su propia estructura de directorios.

- Los procesos pueden pertenecer a un solo cgroup por jerarquía.
- El diseño fue incremental y descoordinado, lo que lo volvió complejo con el tiempo.
- Se usan herramientas como cgcreate, o comandos con mkdir y echo para administrar
- Soporta asignación de threads a cgroups distintos (aunque luego se limitó).
- Se pueden usar herramientas como libcgroup.

Reglas de la jerarquía en V1

- Una jerarquía cualquiera puede tener 1 o más subsistemas adosados
- Un subsistema no puede estar en más de una jerarquía si esas jerarquías tienen subsistemas distintos
- Un proceso solo puede estar en un cgroup por jerarquía.
- Los procesos hijos heredan los cgroups de sus padres (se cumple para v2)

Cgroups v2

- Introducido en Linux Kernel 4.5 (2016).
- Tiene una única jerarquía unificada para todos los controladores.
- No se pueden montar controladores individualmente.
- Se usan archivos como:
 - cgroup.controllers: lista de controladores disponibles.
 - cgroup.subtree_control: activa/desactiva controladores para cgroups hijos (+pids, -memory).
 - cgroup.events: muestra si un cgroup está poblado o congelado.
- Solo se permite que los procesos estén en cgroups hoja (sin hijos).
- Mucho más coherente, más simple de usar y mantener.

Subsistemas (Controladores) Comunes

- **cpu**: Limita el uso de CPU (ej. cpu.shares).
- **memory**: Controla el uso de RAM y swap.
- **io/blkio**: Restringe ancho de banda de disco.
- **pids**: Limita el número de procesos.
- **net_prio** (solo v1): Prioridad de tráfico de red.

Comandos claves de cgroup

- Ver controladores en v2:

```
cat /sys/fs/cgroup/unified/cgroup.controllers
```

- Asignar proceso a cgroup (v1):

```
echo $$ > /sys/fs/cgroup/cpu/mi_grupo/cgroup.procs
```

- Limitar CPU al 30% (v1):

```
echo 307 > /sys/fs/cgroup/cpu/mi_grupo/cpu.shares
```

- Forzar Cgroups v1:

```
GRUB_CMDLINE_LINUX_DEFAULT="quiet systemd.unified_cgroup_hierarchy=0"  
sudo update-grub && reboot
```

Casos de uso

1. **Contenedores:** Docker/Kubernetes usan cgroups para aislar recursos.
2. **Servicios críticos:** Asignar más CPU a un servicio web.
3. **Evitar DoS:** Limitar memoria de un proceso malicioso.
4. **Systemd:** Usa cgroups para administrar servicios.

| Característica | V1 | V2 |
|-------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Jerarquía | Múltiples, 1 por subsistema | Única unificada |
| Controladores | Controladores separados, más flexible | Limitados, no soporta net_cls y net_prio, más simple |
| Ubicación | /sys/fs/cgroup/ | /sys/fs/cgroup/unified/ |
| Asignación de procesos | Procesos en múltiples jerarquías pero un solo cgroup dentro de una jerarquía | Solo en las hojas de la jerarquía |
| Archivos clave | <ul style="list-style-type: none">• cpu.shares: Asignar prioridad de CPU (valor relativo, default 1024)• memory.limit_in_bytes: Establece límite máximo de memoria en bytes• cgroup.procs: Lista de procesos asignados al grupo• tasks: Lista de hilos asignados (solo en v1) | <ul style="list-style-type: none">• cgroup.controllers: Muestra los controladores disponibles• cpu.max: Define límite absoluto de CPU (formato "max periodo")• memory.max: Establece límite máximo de memoria• cgroup.subtree_control: Habilita/deshabilita |

| | | |
|---------------------|------------------------------------------------------------------------------------------------------------------------|-----------------------------------|
| | <ul style="list-style-type: none"> • blkio.weight: Controla prioridad de E/S de disco (100-1000) | controladores para subgrupos |
| Herramientas | cgcreate, libcggroup, etc. | Uso directo del pseudo-filesystem |

Namespace Isolation

Permite abstraer un recurso global del sistema para que los procesos dentro de ese "namespace" piensen que tienen su propia instancia aislada de ese recurso global

- Limitan lo que un proceso puede ver y, en consecuencia, lo que puede usar.
- Un proceso solo puede estar en un namespace de un tipo a la vez.
- Un namespace es automáticamente eliminado cuando el último proceso en el finaliza o lo abandona
- Un proceso puede utilizar ninguno/algunos/todos de los namespace de su padre
- Se asignan al crear procesos con flags como CLONE_NEWPID (ej: clone() o unshare())

Tipos principales

- **PID**: Aísla IDs de procesos. Posibilidad de tener múltiples árboles de procesos anidados y aislados
- **Network**: Aísla interfaces de red, puertos, etc.
- **Mount**: Aísla puntos de montaje del filesystem. Cada proceso, o conjunto de procesos, tiene una vista distinta de los puntos de montajes
- **UTS**: Aísla hostname y nombre de dominio.
- **IPC**: Aísla comunicación entre procesos (System V IPC).
- **User**: Aísla IDs de usuarios/grupos.
- **Cgroup**: Aísla jerarquías de cgroups.

Practica

- Aislar procesos:

```
unshare --pid --fork --mount-proc sh # ¡--mount-proc es crucial!
ps aux # Solo verá procesos del namespace (PID 1 = sh)
```

--mount-proc es importante porque si no verá el /proc del padre y vería todos los procesos.

- Ver namespaces activos:

```
ls -l /proc/$$/ns # Muestra namespaces del proceso actual
lsns               # Lista todos los namespaces del sistema
```

- Crear namespace UTS (hostname aislado):

```
unshare --uts sh # Crea nuevo namespace UTS
hostname nuevo_nombre # Cambia hostname solo en este namespace
```

Contenedores

Tecnología liviana de virtualización (lightweight virtualization) a nivel de sistema operativo que permite ejecutar múltiples sistemas aislados (conjuntos de procesos) en un único host.

Instancias ejecutan en el espacio del usuario. Comparten el mismo kernel (el del SO base)

Dentro de cada instancia son como máquinas virtuales. Por fuera, son procesos normales del SO

No es necesario un software de virtualización tipo hypervisor.

No es posible ejecutar instancias de SO con kernel diferente al SO base

- De sistemas operativos: ejecutan un SO completo (menos el kernel). LXC, BSD jails, Solaris Zone, etc.
- De aplicaciones: empaqueta una aplicación o proceso. Docker, Podman, etc

Características:

- **Autocontenidos:** tiene todo lo que necesita para funcionar
- **Aislados:** mínima influencia en el nodo y otros contenedores
- **Independientes:** administración de un contenedor no afecta al resto
- **Portables:** desacoplados del entorno en el que ejecutan. Pueden ejecutar de igual manera en diferentes entornos.

Los procesos en un contenedor tienen 2 IDs: uno en el contenedor y otro en el host (PID Namespace).

Tema 6: Docker

Docker es una plataforma open-source que permite empaquetar y ejecutar una aplicación en containers livianos.

Componentes:

- **Docker Daemon** (dockerd): es el servidor. Responsable por crear, ejecutar y monitorear los contenedores, construcción de imágenes, etc.
- **API**: especifica la interfaz que los programas pueden usar para interactuar con el servidor.
- **CLI**: cliente. Permite a los usuarios interactuar con el servidor mediante comandos.

Utiliza para proveer containers:

- **Namespaces**: Docker lo utiliza para proveer el espacio de trabajo aislado que denominamos container. Por cada container Docker crea un conjunto de espacios de nombres (entre ellos pid, net, ipc y mnt).
- **Control groups**: Para, opcionalmente, limitar los recursos asignados a un contenedor.
- **Unión file systems**: permite apilar capas de archivos en una sola estructura, donde cada capa representa cambios incrementales. Docker lo usa para optimizar el almacenamiento: las imágenes se construyen en capas (ej: una capa base + capas de configuración). Las capas inferiores son de solo lectura, mientras que la última capa, la del contenedor en ejecución, es escribible. Al eliminar un contenedor, se elimina solo su capa escribible; las demás capas no se modifican.

Rango direcciones

Por defecto, Docker asigna direcciones IP a los contenedores dentro de la red 172.17.0.0/16 (rango: 172.17.0.1 a 172.17.255.254)

- **Origen**: Docker crea una red bridge llamada docker0 con esta subred al instalarse.
- **Asignación dinámica**: Cada contenedor recibe una IP única dentro de este rango al iniciarse.
- **Personalización**: Pueden definirse redes custom con otros rangos usando docker network create.

Imagen

template (molde) de solo lectura con todas las instrucciones para construir un contenedor

Cada imagen está compuesta de una serie de **capas** que se montan una sobre otra (stacking) y pueden ser reutilizadas. Cada capa es un conjunto de diferencias con la capa previa. Solo la última es R/W (la capa del container). Las demás, de solo lectura (inmutables).

Contenedor: Instancia en ejecución de una imagen. **Diferencia principal:** Las imágenes son estáticas (como un "molde"), mientras que los contenedores son dinámicos (en ejecución). Desde una imagen es posible generar varios contenedores. Cada contenedor es autónomo y ejecuta en su propio entorno aislado

Registry: Es un almacén de imágenes de Docker. Puede ser público o privado. Por defecto, docker utiliza Docker Hub.

Dockerfile

Las imágenes se construyen siguiendo las instrucciones de un Dockerfile.

```
docker build -t nginx-aplicacion:2025.01 .
```

```
docker run -d -p 80:80 --name mi-aplicacion nginx-aplicacion:2025.01
```

Almacenamiento (Storage)

Docker tiene dos opciones para almacenar datos en el host para que sean persistentes:

- **Volúmenes:** almacenados en una parte del filesystem administrada por Docker (por default: `/var/lib/docker/volumes`). Mejor portabilidad
- **Bind Mounts:** pueden estar en cualquier parte del filesystem. Pueden ser modificados por procesos que no sean de Docker. Son más flexibles, pero dependen del sistema de archivos del host. Son menos portables.

| Característica | Volumes | Bind Mounts |
|-----------------------|----------------------------------------|-------------------------------------------|
| ¿Quién lo administra? | Docker | El usuario (manual) |
| Ubicación | <code>/var/lib/docker/volumes/</code> | Cualquier carpeta del host |
| Portabilidad | Alta (independiente del host) | Baja (depende de ruta local) |
| Permisos | Docker controla el acceso | Más difícil de gestionar |
| Casos de uso | Bases de datos, persistencia confiable | Desarrollo local, debug, compartir código |

Networking

Los contenedores, por defecto, tienen habilitado el **networking**, lo que les permite realizar conexiones salientes sin configuración adicional. Los usuarios pueden definir redes personalizadas y conectar múltiples contenedores a una misma red, permitiendo que se comuniquen entre sí mediante direcciones IP o nombres. Además, un contenedor puede estar conectado simultáneamente a varias redes.

Para que un servicio dentro del contenedor sea accesible desde fuera, es necesario publicar el puerto correspondiente. Es importante tener en cuenta que dos contenedores en el mismo host no pueden publicar el mismo puerto. Por último, los contenedores utilizan los mismos servidores DNS que el host, aunque esta configuración puede ser modificada.

Comandos Básicos de Docker

- **docker pull <imagen>**: descargar imagen de DockerHUB.
- **docker run <imagen>**: ejecutar imagen.
- **docker image build -t NOMBRE.TAG**: crear una imagen a partir de un Dockerfile.
- **docker ps**: containers en ejecución.
- **docker image ls** y **docker container ls -a**: listar imágenes y contenedores

Tema 7: Contenedores - Docker Compose, Podman

Docker Compose

Es una herramienta para correr aplicaciones que requieren múltiples contenedores.

docker compose up -d para iniciar todos los servicios. **-d** para que ejecuten en background (versiones anteriores usaban el comando docker-compose).

Se puede indicar la política de reinicio del contenedor si se detiene por algún motivo.

Permite definir la dependencia de arranque entre contenedores.

Se puede indicar que el contenedor se debe crear a partir de la imagen creada desde un Dockerfile.

Por default, Compose establece una red default a la que todos los contenedores se unen.

Es posible definir redes propias para cada compose

Archivo compose

Archivo de configuración que se utiliza con Docker Compose para definir los servicios que componen una aplicación multi-contenedor.

Su **función principal** es: Describir **qué servicios** (contenedores) necesita nuestra aplicación, **cómo deben ejecutarse**, **qué variables de entorno usar**, **qué redes y volúmenes compartir**, etc.

El **archivo compose** está escrito en **YAML** (Yet Another Markup Language), que es un **lenguaje de marcado legible por humanos** muy utilizado para archivos de configuración.

Un **Dockerfile** es **diferente** a un archivo **Compose** en el sentido de que un Dockerfile define como construir una imagen, mientras que el **Compose** define cómo ejecutar uno o más contenedores basados en imágenes (que pueden haber sido construidas con Dockerfile).

Partes del archivo

- **services:** Define los **contenedores** (servicios) que componen la aplicación.
- **build:** Se usa para **construir una imagen personalizada** a partir de un Dockerfile.

- **image:** Indica la imagen a utilizar para el contenedor.
- **volumes:** Permite montar volúmenes entre el host y el contenedor, o usar volúmenes gestionados por Docker para persistencia de datos.
- **restart:** Define la política de reinicio automático del contenedor en caso de falla o reinicio del sistema.
- **depends_on:** Indica que un servicio depende de otro y debe arrancar después.
 - **depends_on solo garantiza el orden de arranque de los contenedores, no espera a que estén listos.** La solución correcta para estos casos es usar **healthcheck + depends_on + una lógica de espera (wait-for.it.sh o dockerize)**
- **environment:** Define variables de entorno que se pasan al contenedor.
- **ports:** Mapea puertos del host a puertos del contenedor.
- **expose:** Expone puertos solo para otros contenedores en la misma red, no al host.
- **networks:** Permite definir o unir servicios a redes personalizadas, en lugar de usar la red default.

Comandos

- **docker compose create:** Crea los contenedores definidos en el archivo `compose.yml`, pero no los inicia.
- **docker compose up:** Crea e inicia los contenedores automáticamente.
- **docker compose stop:** Detiene los contenedores, pero los mantiene creados.
- **docker compose down:** Detiene y elimina los contenedores, redes, volúmenes anónimos, etc.
- **docker compose down -v:** Se va a hacer lo mismo que **docker compose down** pero adicionalmente, **se eliminan también todos los volúmenes nombrados creados por ese proyecto Compose.**
- **docker compose run:** Crea y ejecuta un contenedor nuevo temporal basado en un servicio.
- **docker compose exec:** Ejecuta un comando dentro de un contenedor ya en ejecución.
- **docker compose ps:** Muestra la lista de contenedores activos definidos en el `compose.yaml`.
- **docker compose logs:** Muestra los logs (salida estándar) de los servicios.

Podman

Podman (abreviación de Pod Manager) es un motor de contenedores daemonless que permite crear, ejecutar y administrar contenedores y pods compatibles con la especificación OCI (Open Container Initiative), sin necesidad de un proceso en segundo plano como `dockerd`. Funciona principalmente en Linux, aunque también es compatible con Windows y Mac.

Puntos clave:

- Usa comandos muy similares a Docker (incluso tiene `podman compose`).
- No necesita permisos de superusuario (soporta rootless containers).
- Permite manejar contenedores en grupos llamados pods, compartiendo red, almacenamiento e IPC.
- Está basado en la biblioteca `libpod`, que maneja el ciclo de vida de contenedores y pods.
- Compatible con runtimes como `runc` y `crun`.

Pod


Un pod es una unidad lógica que agrupa uno o más contenedores que comparten recursos como red, almacenamiento y configuración. Todos los contenedores de un pod se ejecutan en el mismo espacio de red (misma IP y puerto) y pueden comunicarse entre sí mediante `localhost`.

Puntos clave:

- Proviene del modelo de Kubernetes.
- Todos los contenedores del pod **comparten namespaces** como red, IPC y PID.
- Cada pod incluye un contenedor especial llamado **infra** o *pause container*, que mantiene abiertos los namespaces compartidos.
- Se puede crear un pod vacío y agregarle contenedores luego.
- Ideal para contenedores que trabajan en conjunto (por ejemplo, una app y su proxy o base de datos local).
- El aislamiento y configuración del pod se manejan desde el contenedor **infra**.

Contenedores Orquestador

- **Cluster:** grupo de nodos interconectados que trabajan en conjunto.

- 
- Permite aprovisionar, desplegar, escalar y administrar automáticamente contenedores sin preocuparse por la infraestructura subyacente.
 - Creación de servicios de manera declarativa.
 - En general, dos tipos de nodos:
 - **Manager:** encargado de administrar el cluster.
 - **Worker:** encargado de ejecutar las aplicaciones.
 - **Service Discovery:** orquestador brinda información para encontrar otro servicio.
 - **Routing:** paquetes deben llegar entre servicios ejecutando en diferentes nodos.
 - **Load Balancing:** distribuir las cargas de trabajos entre las distintas instancias de un servicio
 - **Scaling:** aumentar/disminuir las instancias de un servicio según la carga de trabajo.

Tema 8: Protección y Seguridad

Los recursos informáticos deben protegerse frente a accesos no autorizados, destrucciones maliciosas o introducción accidental de incoherencias. El responsable de llevar a cabo la tarea es, entre otros, el **Sistema Operativo**, a través de un conjunto de mecanismos.

Protección

Mecanismos específicos del SO para resguardar la información dentro de una computadora, para controlar el acceso de los procesos (o usuarios) a los recursos existentes.

- Acceso al sistema se resuelve con:
 - Autenticación
 - Control sobre bd de usuarios
- Acceso a los recursos del sistema se resuelve con:
 - Permisos
 - Control de acceso

Seguridad

Medida de la confianza en que se puede preservar la integridad de un sistema y sus datos. Utiliza distintos mecanismos con el fin de **proteger** y garantizar ante:

- Amenazas

- Confidencialidad de los datos (Intercepción / Modificación)
- Integridad de los Datos (Modificación)
- Disponibilidad (Interrupción)
- **Intrusos**
 - Acceso indebido al sistema o datos
- **Pérdida accidental de datos**
 - Accidentes naturales
 - Errores de hw o sw
 - Errores humanos

Políticas y mecanismos

Las **políticas** (qué) definen lo que se quiere hacer, en base a los objetivos. Las podemos asociar a los papeles. Rara vez incluyen configuraciones

Los **mecanismos** (cómo) definen cómo se hace. En este punto aparecen las configuraciones e implementaciones reales. Hay diferentes mecanismos para cumplir una política.

Objetos y dominios

Un sistema informático es una colección de procesos y objetos.

Objetos: de HW (CPU, Memoria, etc.) o de SW (archivos, programas, semáforos). Cada objeto debe tener un identificador único que permita referenciar. Los procesos pueden realizar un conjunto finito de operaciones sobre los objetos

Un **dominio** es un conjunto de pares (objeto, derecho). Cada par especifica un objeto y un subconjunto de operaciones que se pueden realizar con él.

Un **derecho** (right) significa autorización para efectuar esas operaciones.

Ejemplo: el dominio D contiene la pareja (archivo A, {read,write}).

Un proceso que se ejecuta dentro del dominio D puede leer y grabar el archivo A.

Dominios y procesos

Principio de **need-to-know o POLA** (Principle of least authority): define que los procesos accedan sólo a los objetos que necesitan (con los derechos que necesiten) para completar su tarea.

La relación entre un proceso y un dominio puede ser:

- **Estática:** si el conjunto de objetos a los que el proceso accede durante su ciclo de vida es fijo.
 - Siempre mismo dominio
 - Puede generar que los procesos tengan más privilegios que los que necesitan en sus fases de ejecución
- **Dinámica:** si el conjunto de objetos puede variar
 - Puede cambiar de dominio. Por ejemplo usando los bits SETUID y SETGID en UNIX sobre los archivos

Ejemplo 1: En Unix el dominio lo definen el UID y el GID. Dos procesos con igual (UID, GID) pueden acceder al mismo conjunto de archivos, es decir, pertenecen al mismo dominio.

Ejemplo 2: Modo usuario y modo supervisor. Dominios de protección organizados jerárquicamente en una estructura de anillos concéntricos. Los privilegios se asignan por anillo y desde ese punto hacia adentro.

Matriz de acceso

- Controla la pertenencia de objetos a dominios y sus derechos.
- Las filas representan dominios
- Las columnas representan objetos.
- Cada elemento $access(i,j)$ representa el conjunto de operaciones (derechos) que un proceso puede invocar en un objeto O_j dentro del dominio D_i .
- Implementa las políticas de protección/seguridad de un sistema

| Objeto Dominio | File1 | File2 | File3 | Printer |
|-------------------|------------|------------|---------|---------|
| D1 | Read | Read | | Print |
| D2 | | Read | execute | print |
| D3 | Read/write | Read/write | | |

Switch

Para poder cambiar de un dominio a otro se debe habilitar la operación switch sobre un objeto (dominio).

La matriz en sí es un objeto que posee atributos. De esta manera se define si sobre un dominio se realiza asignación estática o no:

La conmutación del dominio D_i al dominio D_j estará permitida si se encuentra definida en el switch $\text{access}(i,j)$

La operación ~~control~~ es aplicable **sólo a dominios**.
switch

Copy

Indica que un proceso ejecutándose en ese dominio puede copiar los derechos de acceso de ese objeto dentro de su columna. Se nota con *

Puede modificar derechos dentro de una **columna**

Variantes:

- **Transferencia:** si un derecho se copia desde $\text{matriz}(i,j)$ a $\text{matriz}(k,j)$, el derecho desaparece para $\text{matriz}(i,j)$, o sea, el derecho fue transferido.
- **Propagación o copia limitada:** Se copia el derecho pero no el derecho a copia en el nuevo (R^* es copiado como R , no como R^*)

Owner

Permite agregar nuevos derechos y borrar los ya existentes.

Puede modificar derechos dentro de una **columna**

Si $\text{matriz}(i,j)$ incluye el derecho de owner entonces un proceso ejecutándose en el dominio D_i puede agregar y borrar cualquier entrada en la columna j .

Control

Indica que pueden modificarse y borrarse derechos dentro de una **fila**.

La operación control es aplicable **sólo a dominios**.

Si matriz (i,j) incluye el derecho de control, entonces un proceso ejecutándose en el dominio D_i puede remover cualquier derecho de acceso dentro de la fila j .

Implementación de matriz de acceso

Tabla global

- Consiste en **conjunto de tuplas** $\langle \text{dominio, objeto, derechos-acceso} \rangle$
- Cada vez que se ejecuta una operación M sobre un objeto O_j sobre el dominio D_i , se analiza la tabla y se verifica si se encuentra una terna $\langle D_i, O_j, M \rangle$
 - Si se encuentra se permite la operación
 - Si no se encuentra se deniega
- Su principal **desventaja** es que el tamaño de la tabla hace que no se pueda almacenar toda en memoria
- Fácil de implementar

Lista de control de acceso (ACL)

- Consiste en asociar con cada **objeto una lista** (ordenada) que contenga todos los dominios que pueden acceder al objeto, y la forma de hacerlo
- Cada columna de la matriz se puede ver como una lista de acceso a un objeto, descartándose elementos vacíos
- Para cada objeto, hay una lista de pares ordenados $\langle \text{dominio, derechos} \rangle$
- Cuando se intenta realizar una operación M sobre un objeto O_j (F_1, F_2, F_3) en el dominio D_i (A, B, C), se busca en la lista en el objeto O_j una entrada, donde M pertenece al conjunto R_k
- Cada archivo tiene una ACL asociada. El archivo F_1 tiene dos entradas en su ACL
 - La primera indica que cualquier proceso que sea propiedad del usuario A puede leer y escribir en el archivo.
 - La segunda indica que cualquier proceso que sea propiedad del usuario B puede leer el archivo.
 - Todos los demás accesos están prohibidos.

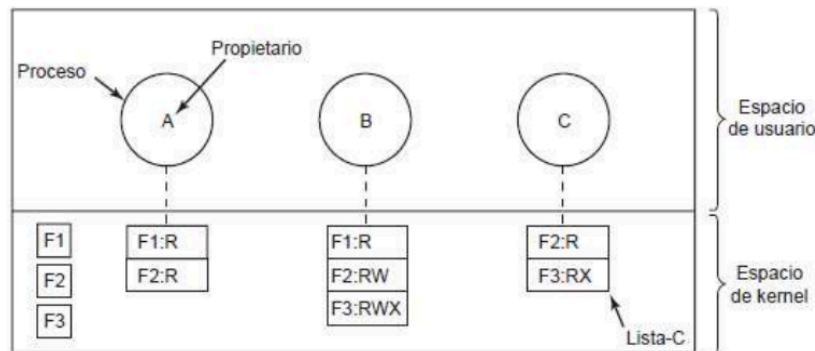


cuando un dominio quiere realizar una operación sobre un obj. se busca en la lista del obj. que aparezca el dominio con los derechos necesarios

* cuando un dominio quiere realizar una operación sobre un obj. se busca en la lista del dominio que aparezca el obj. con los derechos necesarios

Lista de capacidades

- Es una **lista de objetos** del dominio con sus derechos (división x filas)
- A esta lista se le conoce como lista de capacidades y a los elementos individuales que contiene se les conoce como capacidades
- El proceso no la accede directamente
- Esta lista **es un objeto protegido**, a la que sólo accede el SO
- Cada proceso tiene una lista con los objetos que puede utilizar, junto con qué operaciones (el dominio)
- **Presenta dificultades al momento de revocar o modificar un permiso sobre un objeto** – Se deben recorrer todas las listas de capacidades
- Cada capacidad otorga al propietario ciertos derechos sobre un objeto
- Por ejemplo, el proceso que pertenece al usuario A puede leer los archivos F1 y F2



Errores en código

Los procesos, junto con el Kernel son una potencial amenaza a la seguridad de un sistema. Los atacantes aprovechan errores en la codificación del SO, o algún proceso con alto nivel de privilegios con el fin de que los mismos cambien su funcionamiento normal:

- **Desbordamiento de buffer:** Tiene como **objetivo** sobrescribir datos de ciertas zonas de memoria intencionalmente. Si el dato que se sobrescribe no tiene un sentido, entonces probablemente lo que se verá es un error en la ejecución. Si la posición de memoria se sobrescribe con un valor sensible, la ejecución podría encadenar una nueva ejecución de un programa malicioso
- Cadenas de formato
- Retorno a libc
- Desbordamiento de enteros

- Inyección de código

Conceptos de la práctica:

ASLR

ASLR (Address Space Layout Randomization) es una técnica de seguridad que aleatoriza la ubicación de ciertas áreas clave de memoria (stack, heap, data, text y bibliotecas) en cada ejecución de un proceso.

Objetivo principal

Dificultar los ataques de tipo buffer overflow, especialmente aquellos que intentan ejecutar código inyectado. Con ASLR, esas direcciones cambian aleatoriamente cada vez que se ejecuta un proceso, por lo que un atacante no puede predecir con facilidad dónde se encuentra el código o los datos que quiere usar o modificar.

Linux implementa ASLR para los procesos de usuario. Está controlado por el archivo: `/proc/sys/kernel/randomize_va_space`. Este archivo puede tener los siguientes valores:

0 : ASLR deshabilitado.

1 : Randomiza stack, VDSO (Virtual Dynamic Shared Object), memoria compartida. La sección data se ubica al final del text. [Habilitado Parcialmente](#)

2 : Randomiza stack, VDSO, memoria compartida y data. [Completamente Habilitado](#)

Linux también implementa ASLR para el kernel. Esto se conoce como **KASLR** (Kernel Address Space Layout Randomization). KASLR aleatoriza la ubicación de ciertas estructuras internas del kernel durante el arranque del sistema.

Ejercicio Buffer Overflow reemplazando dirección de retorno

El ejercicio muestra cómo un bug de programación simple (como el uso de `gets()`) puede llevar a una escalada de privilegios crítica si se cumplen ciertas condiciones (como SETUID y falta de mitigaciones).

1. El error original: `gets()` y desbordamiento de buffer

Hay una función que pide una contraseña y la guarda en una variable `char password[16]`. El problema es que usa `gets()` para leer la entrada del usuario:

- **gets()** no verifica la cantidad de caracteres que ingresás, entonces si escribís más de 16, el programa sigue escribiendo en la memoria que está al lado.

Esto se llama desbordamiento de buffer (buffer overflow).

2. Objetivo del atacante: cambiar el flujo del programa

Al escribir demasiado en el buffer, podés pisar (sobrescribir) la dirección de retorno de la función actual.

- La **dirección de retorno** le dice al programa a qué parte del código volver cuando termina la función.

Si la reemplazás con una dirección distinta (por ejemplo, de una función secreta como *privileged_fn()*), entonces el programa va a ejecutar esa función en lugar de seguir normalmente.

3. Construcción del ataque (payload)

El atacante hace lo siguiente:

- Calcula cuántos bytes hay entre el inicio del buffer y la dirección de retorno (en este caso, 24).
- Escribe 24 caracteres cualquiera (relleno).
- Luego pone la dirección de la función que quiere ejecutar (por ejemplo *privileged_fn()*).

4. Escalada de privilegios con SETUID

El binario (01-stack-overflow-ret) se configuró para que tenga **SETUID** con propietario root.

- Esto significa que, aunque lo ejecutás como usuario normal, el programa corre con permisos de root.

Entonces, si lográs ejecutar la función *privileged_fn()* (que abre una shell), vas a tener acceso total al sistema (UID=0).

5. Por qué esto es importante en seguridad

Este tipo de vulnerabilidad es muy grave porque:

- Permite que cualquier usuario se convierta en root.
- Es muy común en programas escritos en C sin cuidado.
- Por eso existen defensas modernas como:

| Defensa | Qué hace |
|----------------|----------------------------------------------------------------------------------------------------------------------------------|
| ASLR | Cambia las direcciones de memoria cada vez que se ejecuta el programa. Dificulta saber dónde está <code>privileged_fn()</code> . |
| NX bit | Hace que la memoria del stack no se pueda ejecutar. |
| Stack Canaries | Detectan si se rompió la pila y cortan el programa antes que sea explotado. |
| SMEP/SMAP | Protecciones específicas del kernel contra accesos no permitidos. |

Ejercicio SystemD Usa namespaces, cgroups y capabilities

SystemD es un sistema de inicialización y gestor de servicios para sistemas Linux que se encarga de arrancar, detener y supervisar los servicios y procesos del sistema desde el arranque.

Comandos

- **systemctl enable:** Activar el inicio automático de un servicio al arrancar el sistema
- **systemctl disable:** Desactivar el inicio automático de un servicio.
- **systemctl daemon-reload:** Recargar la configuración de SystemD después de modificar archivos de unidad
- **systemctl start:** Iniciar un servicio inmediatamente
- **systemctl stop:** Detener un servicio en ejecución.
- **systemctl status:** Ver el estado de un servicio (activo, inactivo, fallido).
- **Systemd-cgls:** Mostrar la jerarquía de control de grupos (cgroups) de SystemD
- **journalctl -u [unit]:** Mostrar logs específicos de una unidad (servicio)

Opciones para configurar en una unit service de systemd

IPAddressDeny: Bloquea conexiones desde direcciones IP específicas.

IPAddressAllow: Permite conexiones sólo desde direcciones IP específicas (anula `IPAddressDeny`).

User: Ejecuta el servicio con un usuario específico (no root).

Group: Ejecuta el servicio con un grupo específico.

ProtectHome: Protege directorios

PrivateTmp: Crea un directorio `/tmp` privado para el servicio, aislado del sistema.

ProtectProc: Restringe acceso a `/proc` para evitar fugas de información

MemoryAccounting: Habilita el monitoreo de uso de memoria.

MemoryHigh: Límite "blando" de memoria (el servicio puede superarlo, pero el kernel intentará reducir su consumo).

MemoryMax: Límite "duro" de memoria (el servicio será terminado si lo supera).

AppArmor

Es un sistema de control de acceso obligatorio (MAC) que restringe lo que pueden hacer los programas en Linux.

Funciona cargando perfiles que definen qué archivos, comandos, recursos de red, etc., puede usar cada programa.

Perfil de appArmor

Es un archivo que indica exactamente qué puede hacer un programa (leer carpetas, conectarse por red, ejecutar otros programas, etc.).

Los perfiles pueden estar en distintos modos:

- **enforce:** bloquea accesos no permitidos. Cuando un perfil está en modo enforcing, AppArmor bloquea cualquier operación no permitida, aunque el programa tenga permisos de root.
- **complain:** solo registra violaciones, sin bloquearlas (modo aprendizaje).

Herramientas clave

- **aa-status:** muestra qué perfiles están cargados y qué programas los están usando.
- **aa-enabled:** verifica si AppArmor está activado.
- **aa-genprof:** genera un nuevo perfil observando qué hace un programa.
- **aa-enforce, aa-complain, aa-logprof:** permiten activar, desactivar o modificar perfiles existentes.

AppArmor como defensa en profundidad

No reemplaza otras medidas de seguridad, pero reduce el impacto de vulnerabilidades. Incluso si un atacante toma control de un programa, AppArmor puede evitar que acceda a recursos sensibles.