

# **RESUMEN DE TEORIA DE SISTEMAS OPERATIVOS**

## **Clase 1 | Kernel**

### **SISTEMA OPERATIVO**

Un sistema operativo es un software que actúa como intermediario entre el usuario de una computadora y su hardware. Necesita procesador y memoria para ejecutarse.

El SO “oculta” el HW y presenta a los programas abstracciones más simples de manejar. Los programas de aplicación son los “clientes” del SO.

. *Funciones:*

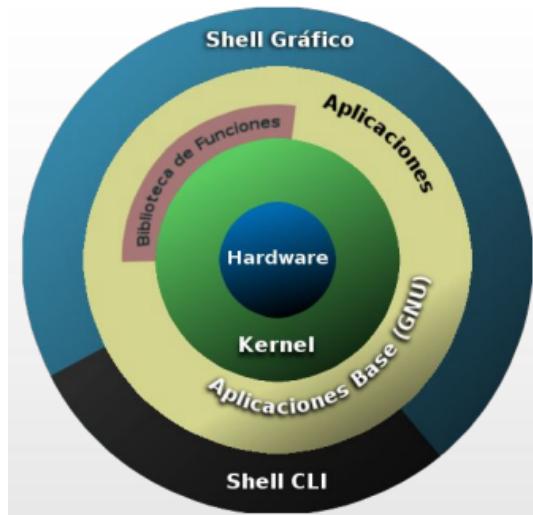
Gestiona el HW.

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.

. *Componentes:*



### **KERNEL**

Es una porción de código que se encuentra en la memoria principal y se encarga de la administración de los recursos.

Es un programa que ejecuta programas y gestiona dispositivos de hardware.

Es el encargado de que el software y el hardware puedan trabajar juntos.

Implementa servicios esenciales:

- ✓ Manejo de memoria
- ✓ Manejo de la CPU
- ✓ Administración de procesos
- ✓ Comunicación y Concurrency
- ✓ Gestión de la E/S

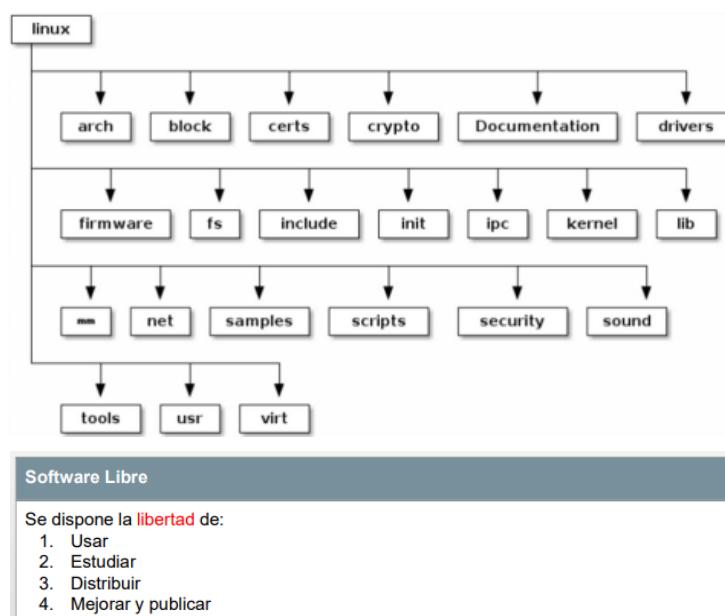
### *. Kernel Linux:*

Es responsable de facilitar a los procesos acceso seguro al hardware. Para ello, utiliza una interfaz conocida como “llamadas al sistema”.

El Kernel se ejecuta en modo supervisor o privilegiado. En este modo se tiene acceso al conjunto completo de instrucción.

Por su parte los procesos se ejecutan en modo usuario.

El Kernel de GNU/Linux está dividido en subsistemas. Cada subsistema es mantenido por uno o más responsables. El responsable de cada subsistema acepta o no parches o pull requests de los desarrolladores. Luego el responsable interactúa con Linus para incluir las modificaciones en una versión candidata “rc”.



### *. Apoyo del Hardware:*

**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. Las instrucciones privilegiadas deben ejecutarse en modo Supervisor o Kernel.

En modo Usuario, el proceso puede acceder sólo a su espacio de direcciones.

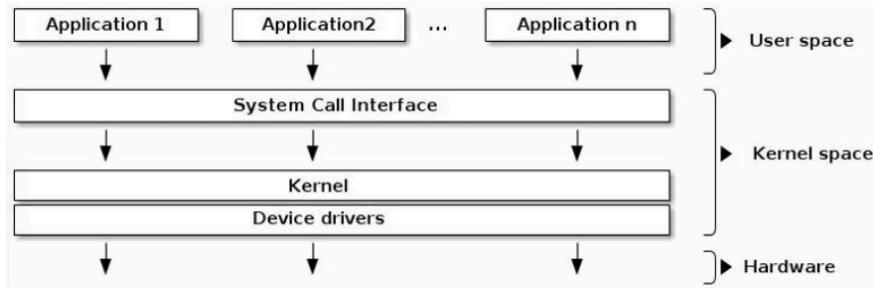
Cuando se arranca el sistema, arranca con el bit en modo supervisor.

Cada vez que comienza a ejecutarse un proceso de usuario, este bit se DEBE PONER en modo usuario.

Cuando hay un trap o una interrupción, el bit de modo se pone en modo Kernel.

**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).



. Caracterizaciones:

### Monolítico:

Incluye todos los servicios del sistema operativo en un solo bloque de código que se ejecuta en modo supervisor (gestión de procesos, memoria, archivos, controladores, etc.).

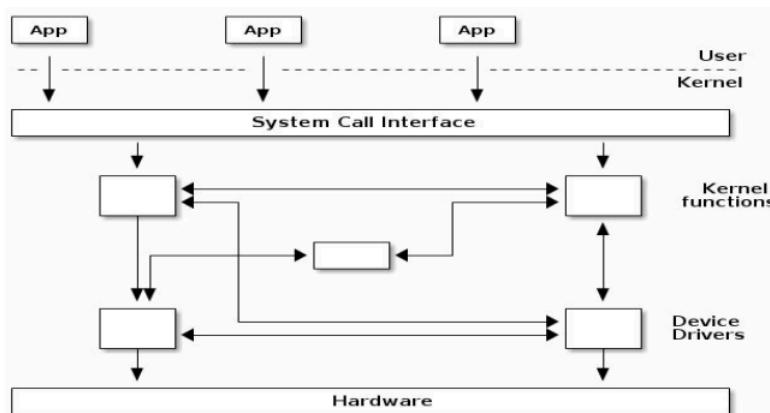
Posee distintos subsistemas y la funcionalidad de cada uno es accedida directamente desde otro a través de funciones públicas.

Gestión completa de recursos (memoria, cpu, E/S).

Posee acceso completo a los recursos de hardware □ Eficiencia al no requerir cambios de modo mientras se ejecuta.

Manejo directo de interrupciones y excepciones.

Permite opcionalmente un enfoque modular, pero los módulos siguen ejecutándose en modo Kernel □ GNU/Linux.



### Microkernel:

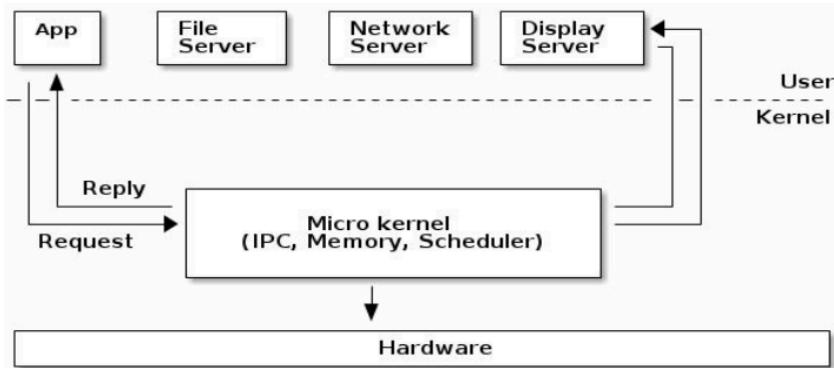
Minimiza la cantidad de código que se ejecuta en modo supervisor con el fin de hacerlo más liviano respecto a un monolítico.

Incluye funciones más esenciales, como la gestión de procesos, comunicación entre procesos y gestión de memoria. Otros servicios (como controladores de hardware, sistemas de archivos y red) se ejecutan en espacio de usuario.

El altamente modular permitiendo su personalización a través de la adición o eliminación de módulos de funcionalidad.

Dado que la mayoría de los servicios se ejecutan en modo usuario, suele ser más estable y seguro que los monolíticos.

Provee un rendimiento inferior por los cambios de modo constantes que requiere para su ejecución.



### Híbrido:

Combina características de los Kernels monolíticos y Microkernels.  
Tiene un núcleo más pequeño que un monolítico. Incluye algunos servicios en modo núcleo (eficiencia) y otros se ejecutan en modo usuario (seguridad).  
Son modulares. Ofrecen un equilibrio entre rendimiento y modularidad lo que permite actualizaciones más sencillas y una mayor flexibilidad y facilidad a su desarrollo.  
Ofrecen un mejor rendimiento que los microkernels y mayor modularidad/seguridad que los monolíticos. Suelen ser una alternativa equilibrada y atractiva.

## Clase 2 | Kernel

El Kernel Linux es un núcleo monolítico híbrido ya que los drivers y el código del Kernel se ejecutan en modo privilegiado. Y lo que lo hace híbrido es la posibilidad de cargar y descargar funcionalidad a través de módulos.

### LINUX, UN POCO DE HISTORIA

En 1991 Linus Torvalds inicia la programación del kernel Linux basado en Minix.  
El 5 de octubre de 1991, se anuncia la primera versión “oficial” de Linux (0.02).  
En 1992, con la release de la versión 0.12, se decide cambiar a una licencia GNU.  
En marzo de 1994 Torvalds considera que todos los componentes del kernel estaban suficientemente maduros y lanza la versión 1.0.  
En julio de 1996 se lanza la versión 2.0 y se define un sistema de nomenclatura. Se desarrolló hasta febrero de 2004 y terminó con la versión 2.0.40. Esta versión comenzó a brindar soporte a sistemas multiprocesadores. La versión 2.4 fue la que catapultó a GNU/Linux como un sistema operativo estable y robusto.  
A fines del año 2003 se lanza la versión 2.6. Esta versión ha tenido muchas mejoras para el kernel dentro de las que se destacan soporte de threads, mejoras en la planificación y soporte de nuevo hardware.  
El 17 Julio de 2011 se lanza la versión 3.0. Termina con la versión 3.8.30. Provee mejoras en Virtualización y FileSystems.  
El 12 de Abril de 2015 se lanza la versión 4.0. Una de sus principales mejoras es la posibilidad de aplicar parches y actualizaciones sin necesidad de reiniciar el SO.  
El 3 de Marzo de 2019 se lanza la versión 5.0. Añade soporte para muchas nuevas cosas, entre ellas para BTRFS.  
El 2 de Octubre de 2022 se lanza la versión 6.0. Agrega soporte para módulos escritos en Rust y mejoras en rendimiento y fragmentación para BTRFS, entre otras cosas.

*Versionado:*

En versiones < 2.6

X.Y.Z

- **X** Indicaba la serie principal. Cambiaba al agregar/quitar una funcionalidad muy importante.
- **Y** Indicaba si era una versión de producción o desarrollo.
- **Z** Bugfixes.

Existían dos versiones del kernel:

- Números **Y** pares indicaban una versión en Producción (estable)
- Números **Y** impares indicaban una versión en Desarrollo

En versiones >= 2.6 y < 3.0

A.B.C.[D]

- **A** Denota Versión. Cambia con menor Frecuencia (cada varios años).
- **B** Denota revisión mayor.
- **C** Denota revisión menor. Solo cambia cuando hay nuevos drivers o características.
- **D** Se utiliza cuando se corrige un grave error sin agregar nueva funcionalidad.

En versiones >= 3.0

A.B.C[-rcX]

- **A** Denota revisión mayor. Cambia con menor Frecuencia (cada varios años).
- **B** Denota revisión menor. Solo cambia cuando hay nuevos drivers o características.
- **C** Número de revisión
- **rcX** Versiones de prueba

## ¿POR QUÉ RECOMPIALAR EL KERNEL?

- Soportar nuevos dispositivos como, por ejemplo, una placa de video.
- 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: ejecuta las directivas definidas en los Makefiles.
- 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.

### . Pasos para recompilarlo:

1. Obtener el código fuente.
2. Preparar el árbol de archivos del código fuente.
3. Configurar el kernel.
4. Construir el kernel a partir del código fuente e instalar los módulos.
5. Reubicar el kernel.

6. Creación del initramfs.
7. Configurar y ejecutar el gestor de arranque (GRUB en general).
8. Reiniciar el sistema y probar el nuevo kernel.

## Clase 3 | System Calls

### **SYSTEM CALL**

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

Como los procesos de usuario se ejecutan en modo usuario y no poseen acceso directo al hardware, solicitan acceso al HW a través de estas system calls.

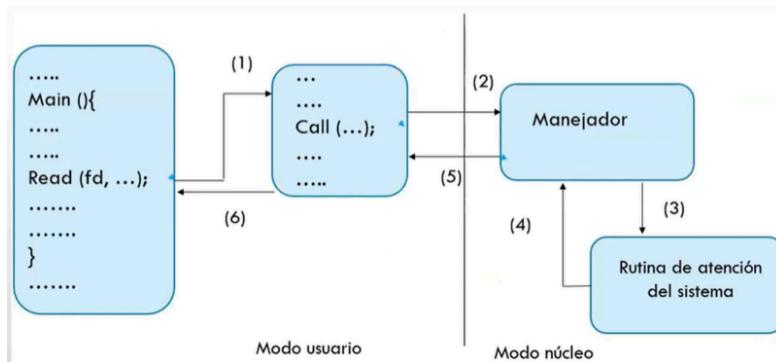
De esta manera un proceso de usuario accede a funciones o servicios que deben ser protegidos por el Sistema Operativo.

El SO provee una API y actúa como un servidor que recibe continuamente requerimientos de sus clientes (procesos de usuario).

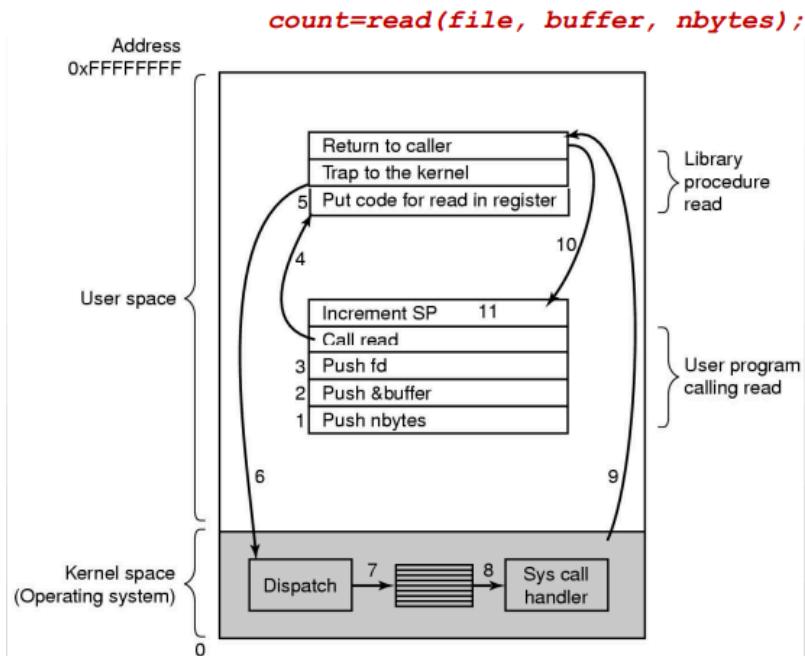
Ejemplo *Read*:

Cuando un proceso necesita leer un archivo, requiere acceder al hardware. Este acceso solo puede realizarlo el SO, con lo cual expone un servicio que permite que sea invocado por el proceso:

```
count=read(file, buffer, nbytes);
```



. Pasos de una system call:



- 1-2-3. Meter en la pila los parámetros necesarios.
4. Se invoca a la función read implementada en la librería y se comienza su ejecución.
5. 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.
6. 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. La forma de identificar la Syscall invocada es a través del valor del registro.
7. Se pasa a modo Kernel. El SO tiene el control, verificará cuál es la llamada al sistema que debe atender y ejecutará el código correspondiente. En este punto se accede al dispositivo de almacenamiento para obtener el archivo solicitado y leerlo en memoria.
8. Se toma el handler correspondiente para atender la system call.
9. Se devuelve el control al modo usuario. Vuelve al punto desde donde se ejecutó la system call.
10. Vuelve al proceso de usuario.
11. Incrementa el SP.

. Para activar la System Call:

Se debe indicar:

- El número de syscall que se quiere ejecutar.
- Los parámetros de esa syscall.

Luego se emite un aviso al SO (Trap) para pasar a modo Kernel y gestionar la system call.

Se evalúa la system call deseada y se ejecuta.

. Registros:

EAX lleva el número de syscall que se desea ejecutar.

EBX lleva el primer parámetro.

ECX lleva el segundo parámetro.

EDX ...

ESI

## EDI

### . Categorías:

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

### . Dispatcher:

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.

### . Parámetros:

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. De no verificarse, en un read por ejemplo el buffer podría tener una dirección del Kernel y sobrescribir datos sensibles.
- Los punteros deben ser siempre válidos. De no verificarse podría producir Kernel Panic.
- 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())

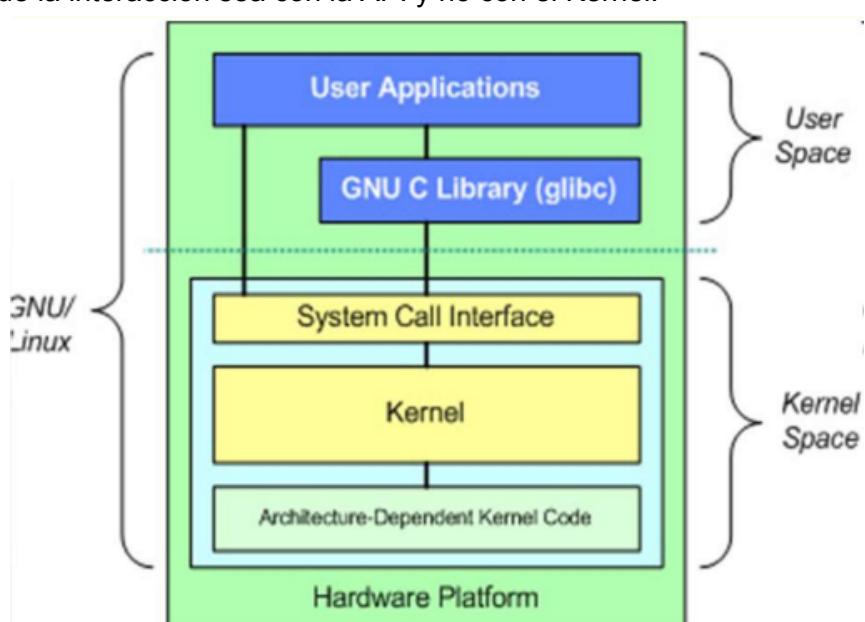
### . Libc:

En UNIX la API principal para la invocación de la System Call es libc:

- Es la API principal del SO.
- Provee las librerías estándar de C.
- Es una interfaz entre aplicaciones de usuario y las System Calls (System Call Wrappers).

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

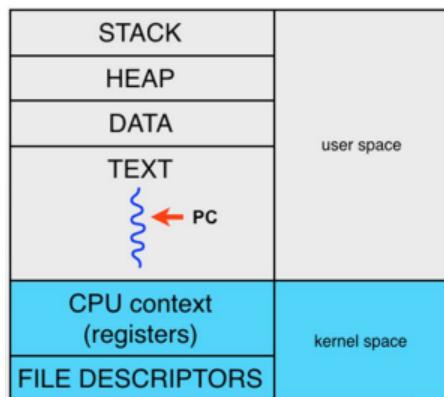
- Busca proveer una interfaz común para lograr portabilidad.
- Busca que la interacción sea con la API y no con el Kernel.



## Clase 4 | Threads

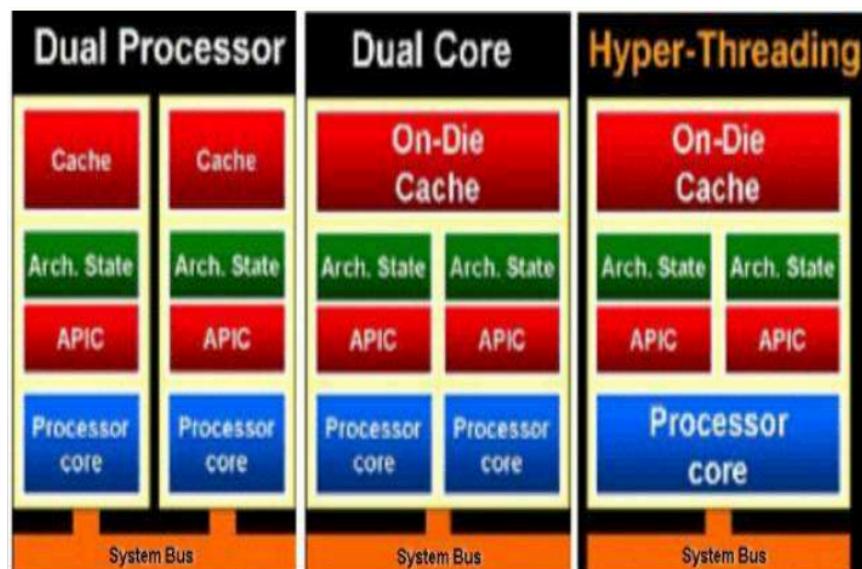
En los primeros sistemas operativos, cada proceso tenía un espacio de direcciones y un solo hilo de control. Un único flujo secuencial de ejecución. Se ejecuta una instrucción y cuando finaliza se ejecuta la siguiente.

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



## EVOLUCIÓN DEL HARDWARE

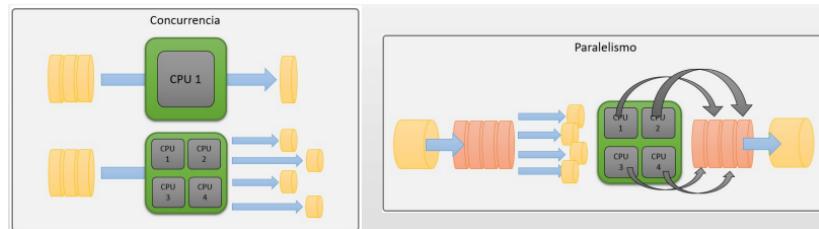
- . **Sistemas Dual-processor (DP):** tiene 2 procesadores físicos en el mismo chasis. Pueden estar en la misma motherboard o no. Cache y controlador independientes.
- . **Sistemas Dual-core:** una CPU con dos cores por procesador físico. Un circuito integrado tiene 2 procesadores completos. Los 2 procesadores combinan cache y controlador.
- . En ambos casos, las APIC (Advanced Programmable Interrupt Controllers) están separadas por procesador. De esta manera proveen administración de interrupciones por procesador.
- . **Multithreading Simultáneo:** generalmente conocido como Hyper Threading, lo cual es propietario de las CPU Intel. Permite que el software programado para ejecutar múltiples hilos (multi-threaded) procese los hilos en paralelo dentro de un único procesador. Esta tecnología simula dos procesadores lógicos dentro de un único procesador físico. Duplica solo algunas “secciones” de un procesador como los Registros de Control (MMU, Interrupciones, Estado, etc) y los Registros de Propósito General (AX, BX, PC, Stack, etc.)



## EVOLUCIÓN DEL SOFTWARE

Se dividen los procesos en diferentes “tareas” que, independientemente o colaborativamente, solucionan el problema.

Por lo general se cuenta con un pool de procesadores para ejecutar nuestros procesos simultáneamente.

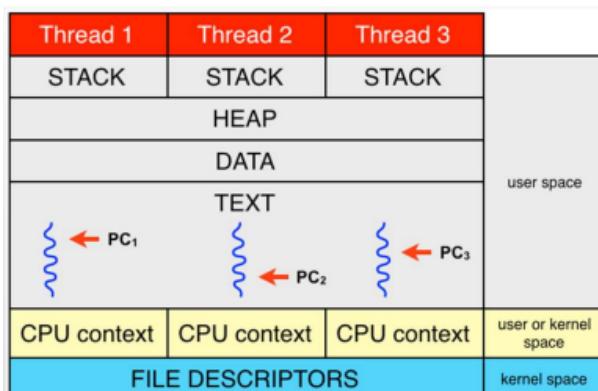


Hoy en día la mayoría de las tecnologías brindan herramientas que nos permiten separar las diferentes “tareas” de los programas en unidades de ejecución diferentes:

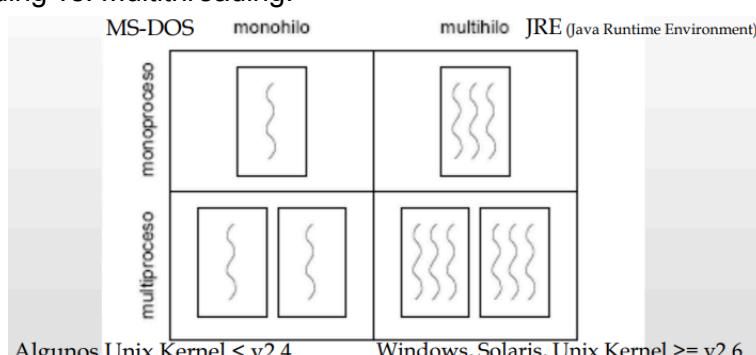
- Java – heredar de “Thread”, implementar la interface “Runnable”
- Delphi – Heredar de “TThread”
- C#, C, etc
- Ruby – Thread.new{CODIGO}
- PHP – Heredar de Thread
- Javascript – HTML5 Web Workers

## EVOLUCIÓN DE LOS SISTEMAS OPERATIVOS

Hay situaciones en las que conviene tener varios hilos de control en el mismo espacio de direcciones y ejecutarlos en quasi-paralelo.



. SO Monothreading vs. Multithreading:



. SO Actuales - Threads:

Proceso:

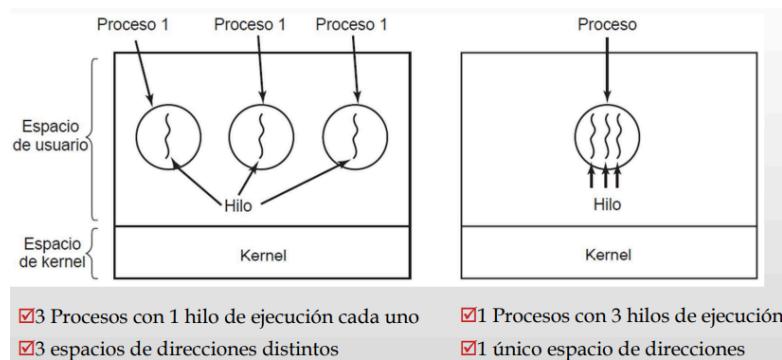
- Espacio de direcciones
- Unidad de propiedad de recursos
- Conjunto de threads (eventualmente uno)

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

## THREADS

Un thread es una unidad de procesamiento dentro de un proceso. Un proceso puede tener uno o varios threads ejecutándose concurrentemente, compartiendo el mismo espacio de memoria.



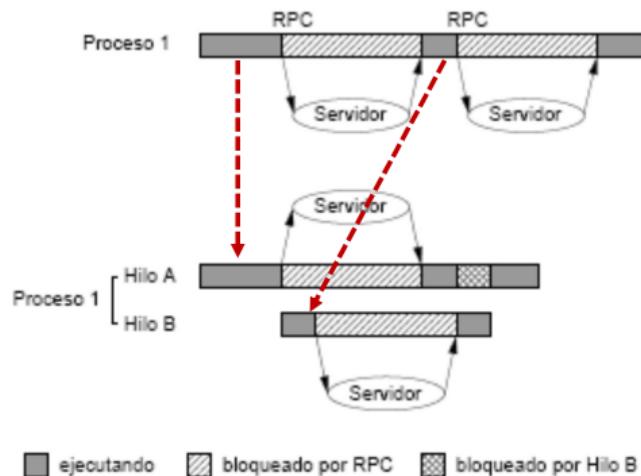
. Ventajas:

- **Sincronización** de Procesos.
- Mejorar **Tiempos de Respuesta**.
- **Compartir Recursos**.
- **Economía**.
  - Con hilos 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. Mientras que en los procesos el SO debe intervenir con el fin de salvar el ambiente del proceso saliente y recuperar el ambiente del nuevo.
- El uso de hilos implica la **creación** de una TCB, registros, PC y un espacio para el stack. Lo hace el mismo proceso sin intervención del SO. Mientras que los procesos implican la creación de un nuevo espacio de direcciones, PCB, PC, etc. Lo lleva a cabo el SO.
- En cuanto a la **destrucción**, con los hilos la tarea se realiza dentro del proceso sin necesidad de intervención del SO. Mientras que en los procesos el SO debe intervenir con el fin de salvar el ambiente del proceso saliente y eliminar su PCB.
- La **planificación** en los hilos es responsabilidad del desarrollador quien debe planificar sus hilos. Es menos costoso, pero puede traer desventajas aparejadas. Mientras que en los procesos es llevada a cabo por el sistema operativo. El cambio implica cambios de contexto continuos.
- En los hilos 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. Mientras que en los procesos el SO garantiza la protección a través de distintos

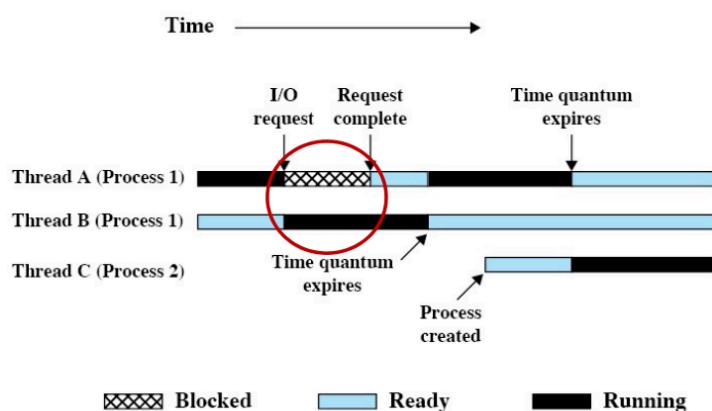
mecanismos de seguridad. La comunicación entre ellos implica el uso de técnicas más avanzadas.

. Ejemplos:

Ej. 1:



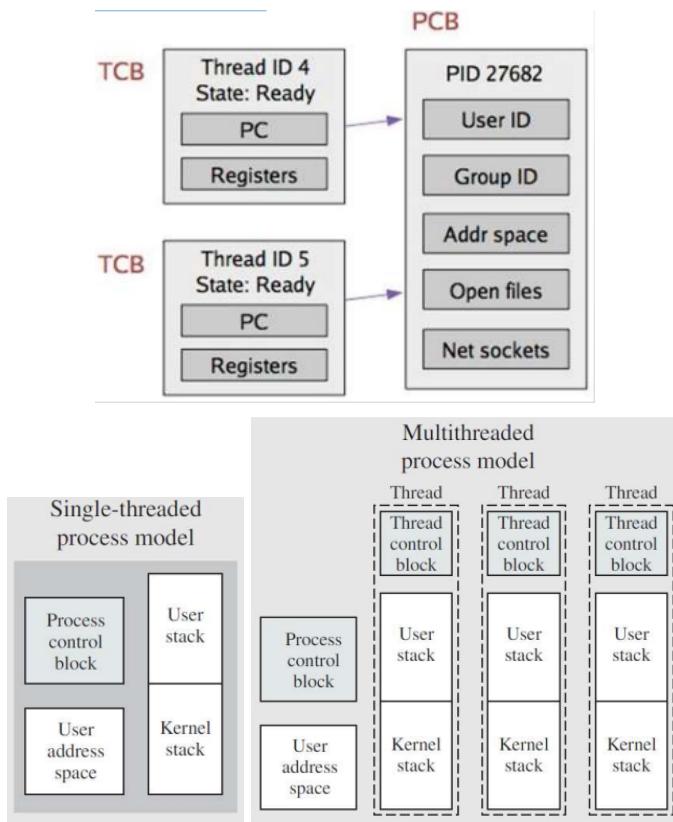
Ej. 2:



. Estructura:

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



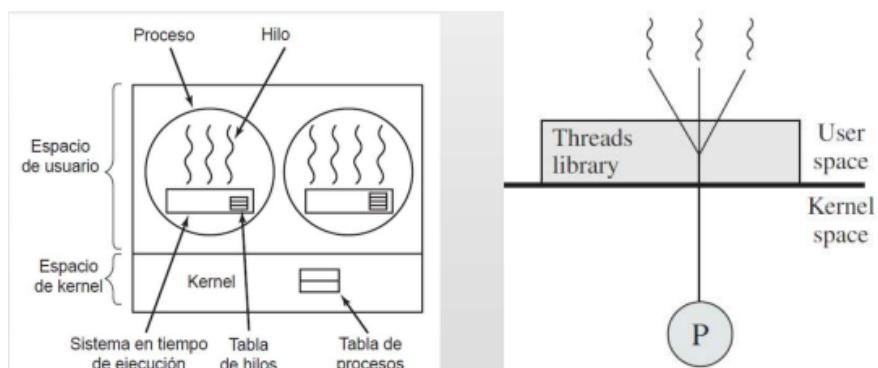
#### . Estados de un Thread:

Los estados mínimos son: **Ejecución, Listo y Bloqueado.**



#### TIPOS - USUARIOS - ULT

- . **ULT: User Level Thread.**
- . 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 más 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.

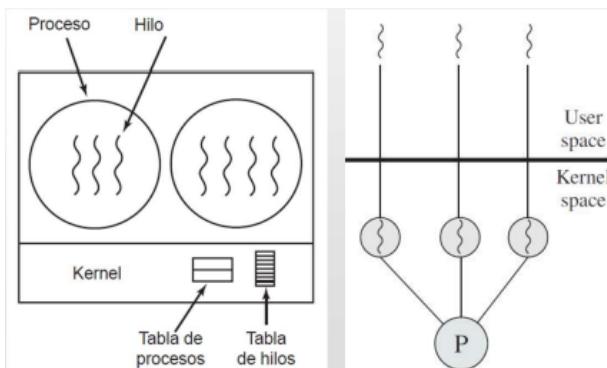
### *Diferencias con KLT:*

Los ULT son threads que actúan en espacio y modo usuario. Se diferencian de los KLT en que:

- Estos últimos realizan la gestión completa en modo Kernel mientras que los ULT se crean, gestionan y destruyen en modo Usuario.
- Los KLT son manejados por el Kernel del sistema operativo mientras que a los ULT los maneja una biblioteca en espacio de usuario.
- Los ULT son invisibles para el Kernel mientras que los KLT los ve.
- En cuanto a la performance, los ULT son más fáciles de manejar, mientras que los KLT son más costosos.
- Con ULT un hilo bloquea a todos, mientras que con los KLT solo el hilo bloqueado se detiene.

## **TIPOS - KERNEL - KLT**

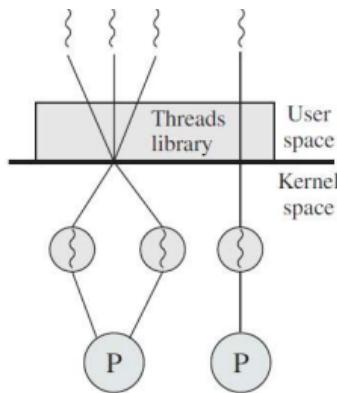
- . KLT: Kernel Level Thread.
- . La gestión completa se realiza en modo Kernel.
- . Ventajas:
  - Se puede multiplexar hilos del mismo proceso en diferentes procesadores.
  - Independencia de bloqueos entre Threads de un mismo proceso.
- . Desventajas:
  - Cambios de modo de ejecución para la gestión: planificación, creación, destrucción, etc.



## **TIPOS DE THREADS - COMBINACIONES**

- . Es posible combinar ULT y KLT.

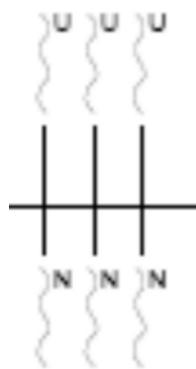
- . En este tipo de sistemas, la creación de hilos se realiza a nivel de usuario y los mismos son mapeados a una cantidad igual o menor de KLT.
- . La sincronización de hilos en este modelo, permite que un hilo se bloquee y otros hilos del mismo proceso sigan ejecutándose.
- . Permite que hilos de usuario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



## **MODELOS DE MULTITHREADING**

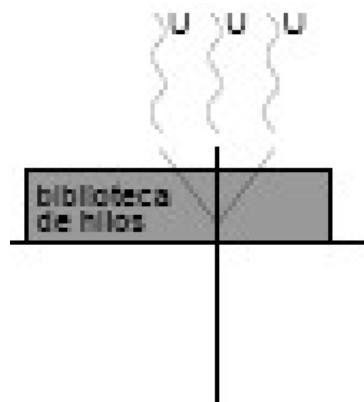
*Uno a Uno:*

- . 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.



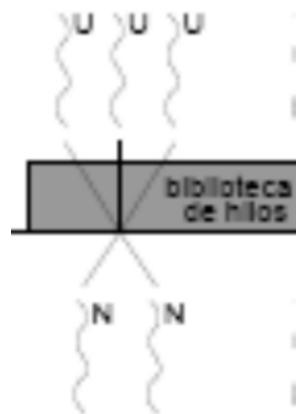
*Muchos a Uno:*

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



#### *Muchos a Muchos:*

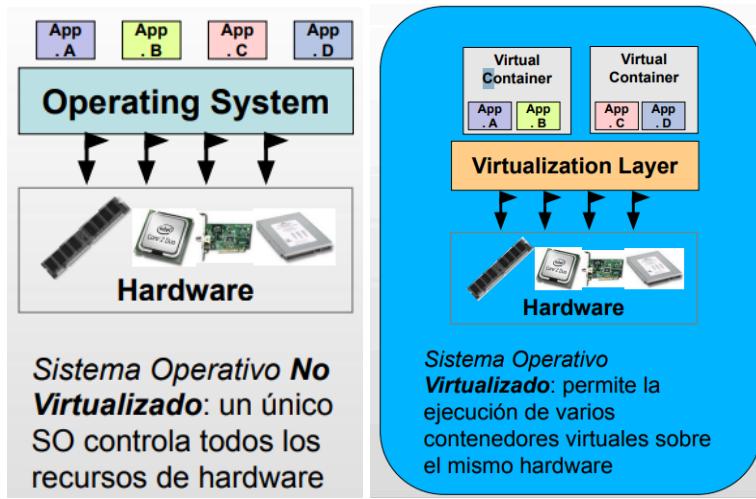
- . Muchos ULT mapean a muchos KLT.
- . Este modelo multiplexa los ULT en KLT, logrando un balanceo razonable:  
No tiene el costo del modelo 1:1.  
Minimiza los problemas de bloqueo del modelo M:1.



## Clase 4 | Virtualización

### **VIRTUALIZAR**

- . Es una técnica que permite realizar “abstracción” de recursos de la computadora. Hoy en día también se virtualizan las redes, storages, data centers, etc...
- . Es una capa abstracta que desacopla el hardware físico del sistema. Permite ocultar detalles técnicos a través de la encapsulación.
- . Permite, también, entre otras cosas, que una computadora pueda realizar el trabajo de varias, a través de la compartición de recursos de un único dispositivo de hardware.



. *Ejemplo: CMS y VM/370:*

- VM/CMS es un sistema operativo orientado a máquinas virtuales que fue lanzado en el año 1972 por IBM.
- El SO corría en entornos mainframe con arquitectura System/370, System/390 y más avanzados en zSeries.
- Utiliza como núcleo a VMCP (Virtual Machine Control Program), lo que permite la ejecución de las VM y el control del hardware.
- En VM/370, hay una máquina CMS (Conversational Monitor System) para cada usuario, con “la ilusión” del hw completo.
- Una aplicación sobre el CMS hace una system call y la “atrapa” (es un trap) el CMS.

## VIRTUALIZACIÓN

- . Es una capa de abstracción sobre el hw para obtener una mejor utilización de los recursos y flexibilidad.
- . Permite que haya múltiples máquinas virtuales (VM), o entornos virtuales (EV), con distintos (o iguales) sistemas operativos corriendo de manera aislada.
- . Cada VM tiene su propio conjunto de hardware virtual (RAM, CPU, NIC, etc.) sobre el cual se ejecuta el SO “guest”.
- . El SO “guest” ve un conjunto consistente de hw, no el hw real (aunque a través de ciertas configuraciones podría ver parte del hardware real).
- . Las VMs están representadas y son encapsuladas en archivos dentro de un File System.
- . Es fácil de almacenar, copiar, de hacer backup y de restaurar.
- . Es simple de expandir y agregar recursos.
- . Son sistemas completos (aplicaciones ya configuradas, SO, hw virtual) pueden moverse de un servidor a otro rápidamente.

. *¿Por qué virtualizar?:*

- Tengo muchas máquinas servidores, poco usadas.
- 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).

*. Esquema Pre-Virtualización:*

En el esquema de los años 2000, 1 máquina corría 1 SO junto con sus aplicaciones.

El aprovechamiento del hardware era bajo.

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

Cada SO desconoce de la existencia de otros SO.

*. Tipos de Virtualización:*

- **Process Level:** permite lograr portabilidad entre diferentes sistemas. Java Virtual Machine (JVM).
- **Storage Level:** presenta una vista lógica del almacenamiento al usuario, quien no sabe dónde 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.

*. Componentes:*

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

Guest puede ser un sistema operativo completo.



*. Monitor de Máquinas Virtuales (VMM):*

Es un software que se va a encargar de llevar a cabo la virtualización. Corre sobre el hardware para implementar las máquinas virtuales.

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 (es él quien recibe y resuelve los traps).
- El VMM interpreta/emula las instrucciones privilegiadas.

. Características:

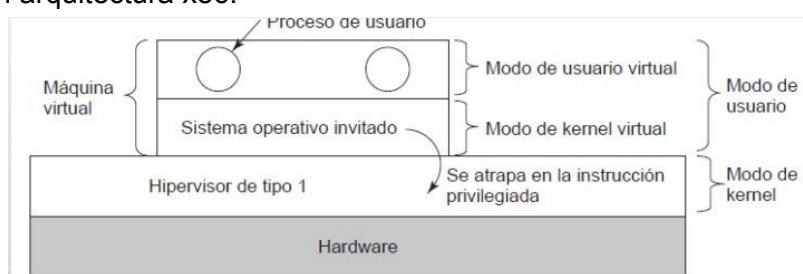
- Equivalencia / Fidelidad: un programa ejecutándose sobre un VMM (Virtual Machine Monitor), también conocido como Hipervisor, debería tener un comportamiento idéntico al que tendría ejecutándose directamente sobre el hardware subyacente. Es decir, que funcione igual si está virtualizada a si no lo está.
- 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. Las instrucciones deberían ser ejecutadas directamente sobre el hardware.

. Modo usuario/supervisor:

Cuando estoy virtualizando, el guest (que siempre corre en modo usuario del so anfitrión) emitirá una instrucción privilegiada que NO debe ser ignorada, sino que debe generar un trap al SO.

Mecanismo conocido como trap-and-emulate.

No aplicable en arquitectura x86.

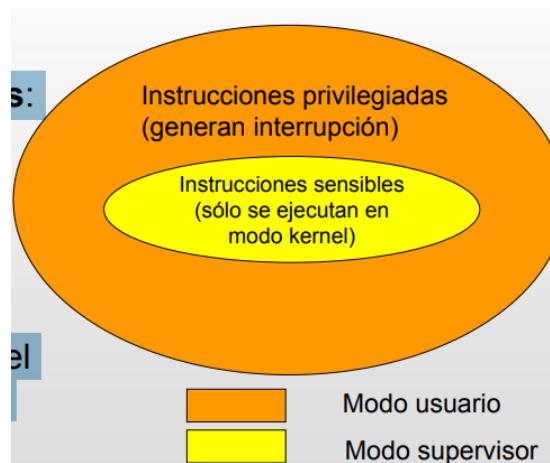


. Instrucciones Privilegiadas e Instrucciones Sensibles:

Pueden ser ejecutadas en modo usuario.

Toda inst. sensible debe ser privilegiada. Las arq x86 no cumplen este principio.

- **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).

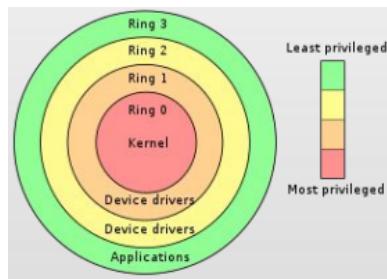


Para construir un VMM es suficiente con que todas las instrucciones que podrían afectar al correcto funcionamiento del VMM (instrucciones sensibles) siempre generen una excepción y pasen el control al VMM.

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.

. *Anillos de Privilegios en x86:*



- Los procesadores x86 proveen protección basada en el concepto de niveles o anillos (rings) de privilegios (0 = más privilegio, 3 menos privilegio). Estos niveles se establecen en 2 bits del registro CS:
  - El software más privilegiado se ejecuta en el Ring 0 (modo Kernel) y el menos privilegiado (modo usuario) en el 1, 2 o 3 (en particular en Intel se ejecuta en el 3).
- El nivel de privilegios determina si las instrucciones privilegiadas, que controlan la funcionalidad básica de la CPU, pueden ejecutarse sin generar una excepción:
  - En Ring 0 se ejecutan directamente.
  - En otros Rings se invocan a través de una excepción.
- Se busca que los Hipervisores se ejecuten en el Ring 0 (o lo más cercano) y los SO guest se ejecuten en el 1 o 3 (x86).

. *Trap and Emulate:*

- Funciona similar a la emulación pero realiza una interpretación selectiva.
- Aplicaciones y sistema operativo ejecutan en modo usuario.
- Aplicaciones ejecutan nativamente en el hardware.
- VMM ejecuta en modo privilegiado.
- Cuando se ejecuta una instrucción privilegiada en el guest SO (en modo usuario) se produce un “trap” al VMM.
- VMM ejecuta las instrucciones necesarias y retorna el control al guest SO.
- No puede ser utilizado en todas las ISAs. Debe cumplir con el teorema de Popek and Goldberg.
- x86 no cumple con el teorema: no todas las instrucciones sensitivas son privilegiadas (por ej. popf).

. *Hypervisors:*

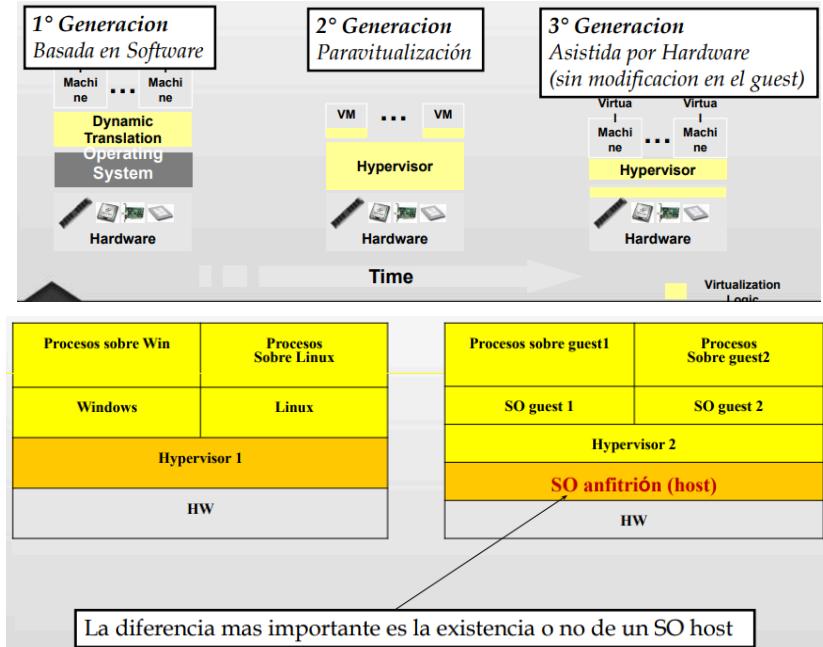
También conocidos como VMM (Virtual Machine Monitor) es una porción de software que separa las “aplicaciones/SO” del hardware subyacente.

Proveen una plataforma de virtualización que permite múltiples SO corriendo en un host al mismo tiempo.

Interactúan con el HW en líneas generales (2° y 3° generación).

Realizan la multiprogramación.

Ofrece varias MV hacia arriba.



### Hipervisor tipo 1:

Se ejecuta en modo kernel (Ring 0).

Cada VM se ejecuta como un proceso de usuario en modo usuario (Ring 3).

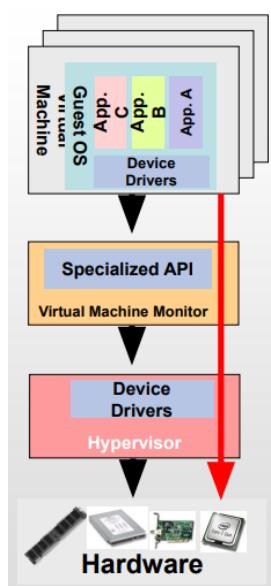
El SO guest no requiere ser modificado.

Existen un modo kernel virtual y modo usuario virtual.

Siempre que la VM ejecuta una instrucción sensible, se produce una trap que procesa el hypervisor. Algunos hipervisores introducen extensiones que le evitan tener que traducir todas las instrucciones.

Debe tener asistencia de Hardware siempre.

Se instala sobre el HW y sobre él se virtualiza.

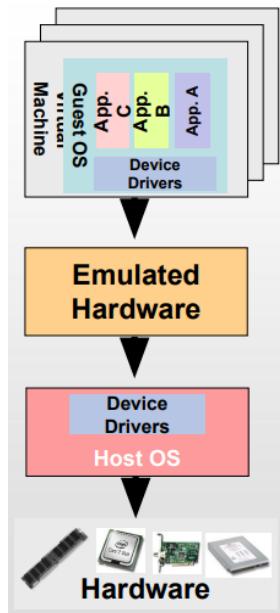


### Hipervisor tipo 2:

Se ejecuta como un programa de usuario en un SO host.

Su función principal es interpretar un subconjunto de las instrucciones de hardware de la máquina sobre la que corre.

Debe emularse el hardware que se mapea a los SO guest.



El tipo 1 se ejecuta sobre el HW, y se ejecuta en modo kernel real. El SO guest se ejecuta en modo kernel “virtual” (pero es modo usuario).

El tipo 2 se ejecuta como un programa de usuario sobre un SO host. Arriba de él, están los SO guests. Interpreta un conjunto de instrucciones de máquina. Y es el SO host quién se ejecuta sobre el HW.

#### *. Técnicas de Virtualización:*

De acuerdo al modo en el que se implemente el VMM podemos nombrar:

- Emulación.
- Asistida por Hardware.
- Paravirtualización.

#### *. Emulación:*

Nos permite correr sobre por ejemplo un x86 cualquier otra arquitectura.

Toda instrucción que ejecute la aplicación va a ser capturada por el emulador y transformada para ser ejecutada en el hardware físico real.

Provee toda la funcionalidad del procesador deseado a través de software:

- Se puede emular un procesador sobre otro tipo de procesador.
- Aplicación/SO emulado ejecuta en modo usuario.
- Se reescribe el conjunto completo de instrucciones.
- Todas las instrucciones son capturadas por el emulador.
- Cada instrucción es interpretada y traducida a una (o varias) equivalente adecuada al hardware subyacente.
- Tiende a ser lenta.

#### *. Full Virtualization:*

Trata de particionar un procesador físico en distintos contextos, donde cada uno de ellos corre sobre el mismo procesador. Hace que el SW virtualizado no sepa que está siendo virtualizado.

Los SO guest deben ejecutar la misma arquitectura de hardware sobre la que corren.

- No requiere que los guest se modifiquen.
- Es en general la técnica más utilizada.
- El VMM analiza el flujo de ejecución:

Los bloques que contienen instrucciones sensibles son modificados.

Los bloques con instrucciones inocuas se ejecutan directamente en el hardware.

### Traducción Binaria:

- Se introdujo en 1998 por VMWare para permitir virtualizar sobre arquitecturas x86.
- Combina traducción binaria con ejecución directa, donde las instrucciones no sensibles se ejecutan directamente sobre el hardware.
- Las instrucciones sensibles deben ser transformadas, pero esa transformación se puede cachear dando una optimización en velocidad. Los bloques traducidos son ejecutados por la CPU directamente.

#### . Asistida por Hardware:

Usa soporte específico del procesador (como **Intel VT-x** o **AMD-V**) para facilitar la virtualización, mejorando el rendimiento y reduciendo la complejidad del hypervisor. Permite ejecutar instrucciones privilegiadas directamente en el hardware.

Intel propone Root-Mode y Non-Root-Mode. AMD Host-Mode y Guest-Mode.

VMM ejecuta en root-mode, VMs en non-root mode. Ambos en ring-0.

Cada modo 4 anillos de seguridad con diferentes niveles de privilegio.

VMs generan traps al hypervisor cuando se ejecutan instrucciones sensibles (non-root mode).

La VM corre en Non-Root. El kernel del SO invitado corre en el ring 0 pero en el modo Non-Root. Si ejecuta una instrucción privilegiada se hace un trap. Sigue sin tener acceso al HW directamente.

Cuando sucede el trap se pasa a modo Root y se resuelve con el hipervisor como siempre.

#### . Paravirtualización:

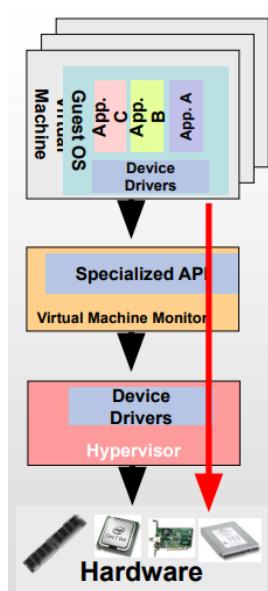
Los hypervisors tipo 1 y 2 ejecutan SO guests no modificados.

Se trata de tener SO guests modificados para mejorar el rendimiento.

Cuando se quiere ejecutar una instrucción sensible, el SO guest la transforma en una llamada al VMM que expone una API específica. Se elimina la traducción binaria, salvo que no esté la traducción.

El VMM no debe emular instrucciones de hardware, lo cual hace que las llamadas se resuelvan de modo más sencillo.

El SO guest es como un proceso de usuario que hace llamadas al SO.



Hay 2 formas de implementarlo:

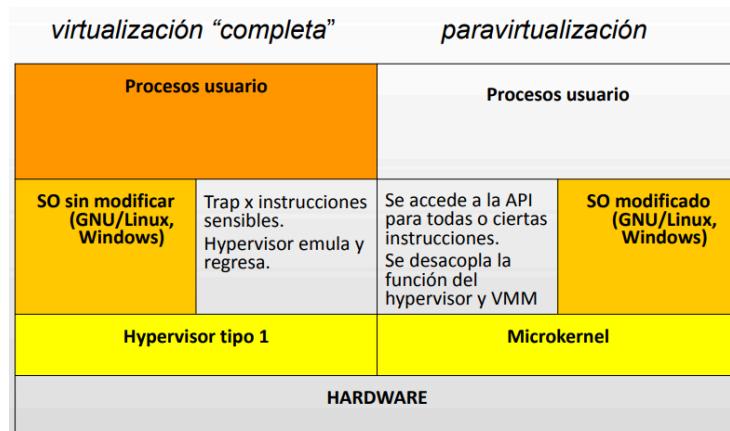
#### Recompilando el Kernel del sistema guest:

- 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.
- Esta es la técnica que se utiliza hoy en día.

#### . 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, pero soporta pocos SO, pues necesita modificar el SO original.

Paravirtualización parcial en drivers es una solución intermedia.

## Clase 5 | cgroups - Namespaces - Docker

### CGROUPS

. En Linux, por defecto, todos los procesos reciben el mismo trato en lo que respecta a tiempo de CPU, memoria RAM, "I/O bandwidth". Y el Kernel no puede determinar cuál proceso es importante y cuál no. Entonces surgen las siguientes preguntas: ¿Qué sucede si se tiene un proceso importante que requiere prioridad? O, ¿Cómo limitar los recursos para un proceso o grupo de procesos?

. Para esto aparecen los **Control Groups, cgroups**, que son una característica del kernel de Linux que permite que los procesos sean organizados en grupos jerárquicos cuyo uso de varios tipos de recursos (CPU, memoria, I/O, etc.) pueda ser limitado y monitoreado.

. La interfaz de cgroups del kernel es provista mediante un pseudo-filesystem llamado cgroups.

. Su objetivo es permitir un control "fine-grained" (permite tomar decisiones detalladas y específicas sobre quién, cómo, cuándo y cuánto accede a los recursos del sistema) en la alocación, priorización, denegación y monitoreo de los recursos del sistema.

. Los procesos desconocen los límites asignados por un cgroup.

### *chroot - Service Isolation*

Chroot es una forma de aislar aplicaciones del resto del sistema.

Cambia el directorio raíz aparente de un proceso. Afecta solo a ese proceso y a sus procesos hijos.

Al entorno virtual creado por chroot a partir de la nueva raíz del sistema se le conoce como “jail chroot”.

No se puede acceder a archivos y comandos fuera de ese directorio.

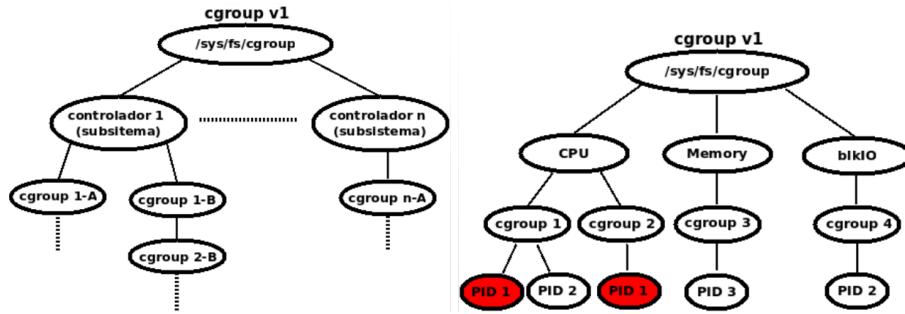
*cgroups* provee

- *Resource Limiting*: los grupos no pueden excederse en la utilización de un recurso (tiempo de CPU, cantidad de CPUs, cantidad de memoria, I/O, etc.).
- *Prioritización*: un grupo puede obtener prioridad en el uso de los recursos (tiempo de CPU, I/O, etc.).
- *Accounting*: permite medir el uso de determinados recursos por parte de un grupo (estadísticas, monitoreo, billing, etc.).
- *Control*: permite freezar y reiniciar un grupo de procesos.

## **CGROUPS v1 - v2**

*cgroup v1*:

- En cgroups v1, el desarrollo de los controladores fue muy descoordinado.
- Distintos controladores se han ido agregando en el tiempo para permitir la administración de distintos tipos de recursos.
- La administración de las distintas jerarquías se hizo cada vez más complejo.
- Diseño posterior a la implementación.
- Características:
  - *cgroup*: asocia un conjunto de procesos con un conjunto de parámetros o límites para uno o más subsistemas.
  - *Subsistema*: componente del kernel que modifica el comportamiento de los procesos en un cgroup. También llamado resource controllers o simplemente controllers. Cada subsistema representa un único recurso: tiempo de CPU, memoria, I/O, etc.
  - *Jerarquía*: es un conjunto de cgroups organizados en una jerarquía. Cada jerarquía es definida mediante la creación, eliminación y renombrado de subdirectorios dentro del pseudo-filesystem. En cada nivel de la jerarquía se pueden definir atributos (por ej. límites). No pueden ser excedidos por los cgroups hijos.
  - Cada proceso del sistema solo puede pertenecer a un cgroup dentro de una jerarquía (pero a varias jerarquías).
  - Los controladores pueden ser montados en pseudo-filesystems individuales o en un mismo pseudo-filesystem. Un controlador puede ser desmontado si no está ocupado: no tiene cgroups hijos.
  - Cada cgroup es representado por un directorio en una relación padre-hijo. Por ejemplo: cpu/procesos/proceso1.
  - Un proceso creado mediante un “fork” pertenece al mismo cgroup que el padre.
  - Es posible asignar threads de un proceso a diferentes cgroups. Deshabilitado en la v2, restaurado luego, pero con limitaciones.
  - Cada cgroup filesystem contiene un único cgroup raíz al cual pertenecen todos los procesos.
  - libcgroup: tools para administrar los cgroups.
- Jerarquía:



#### cgroup v2:

- cgroups v2 pensado como un reemplazo de cgroups v1.
- Controladores también agregados en el tiempo.
- Características:
  - Todos los controladores son montados en una jerarquía unificada. No es posible especificar un controlador en particular para montar.
  - Cada cgroup en la jerarquía v2 contiene, entre otros, los siguientes archivos:
    - `cgroup.controllers`: archivo de solo lectura que indica los controladores disponibles en un cgroup.
    - `cgroup.subtree control`: archivo de lectura/escritura que indica los controladores que se habilitarán en los cgroups hijos. Inicialmente vacío. Determina el conjunto de controladores que pueden ser usados en los cgroups hijos.
  - Un controlador que no está presente en el archivo `cgroup.controllers` no se puede agregar al archivo `cgroup.subtree control`.
  - Los controladores disponibles en el archivo `cgroup.controllers` de un cgroup son idénticos a los existentes en el archivo `cgroup.subtree control` del cgroup padre. Si un controlador se remueve de un cgroup no puede ser rehabilitado en un cgroup hijo.
  - No se permiten procesos internos, a excepción del cgroup raíz. Los procesos deben asignarse a cgroups sin hijos (hojas). Por ej. si `/procesos/proc1` son cgroups entonces un proceso puede residir en `/procesos/proc1`, pero no en `/procesos`.
  - Inicialmente, solo el cgroup raíz existe y todos los procesos pertenecen a él.
  - `mkdir CG NAME` y `rmdir CG NAME` para crear y eliminar cgroups.
  - Cada cgroup tiene un archivo lectura/escritura `cgroup.procs` con los procesos pertenecientes a ese cgroup. Inicialmente vacío.
  - Los archivos correspondientes a un controlador habilitado en el archivo `cgroup.subtree control` de un cgroup son automáticamente generados al crearse un cgroup hijo.
  - En cada cgroup diferente al raíz existe un archivo `cgroup.events` de solo lectura. Contiene pares de clave-valor:

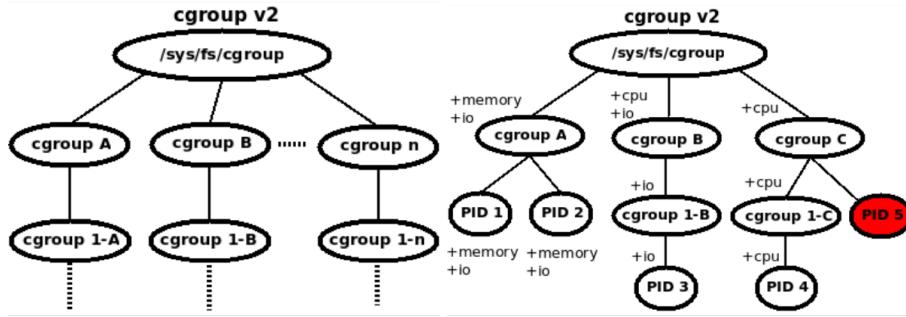
```
$ cat proc1/cgroup.events
populated 1
frozen 0
```

Populated: si es 1, este cgroup o alguno de sus descendientes tiene procesos miembros.

Frozen: si es 1, este cgroup está freezado.

Permite notificar cuando un cgroup está vacío.

- Jerarquía:

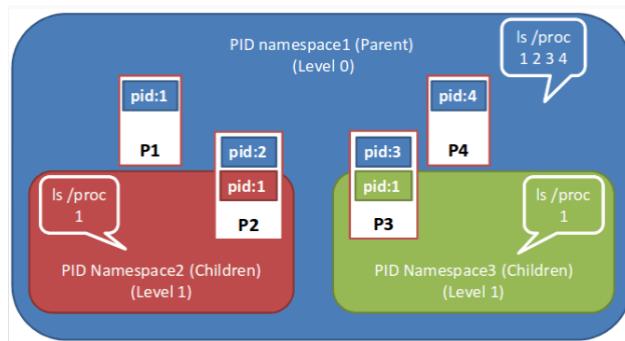


Ambos controladores pueden ser montados en el mismo sistema.

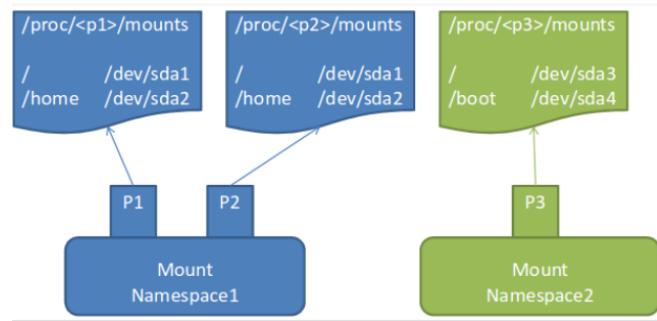
Una jerarquía de un controlador no puede estar en ambos cgroups simultáneamente.

### **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.
- . Las modificaciones a un recurso quedan contenidas dentro del “namespace”.
- . Un proceso solo puede estar en un namespace de un tipo a la vez.
- . Un namespace es automáticamente eliminado cuando el último proceso en él finaliza o lo abandona.
- . Un proceso puede utilizar ninguno/algunos/todos de los namespaces de su padre. Hereda todos los namespaces de su proceso.
- . Hay 3 nuevas system-calls:
  - **clone()**: similar al fork. Crea un nuevo proceso y lo agrega al nuevo namespace especificado. Su funcionalidad puede ser controlada por flags pasados como argumentos.
  - **unshare()**: agrega el actual proceso a un nuevo namespace. Es similar a clone, pero opera en el proceso llamante. Crea el nuevo namespace y hace miembro de él al proceso llamador.
  - **setns()**: agrega el proceso actual a un namespace existente. Desasocia al proceso llamante de una instancia de un tipo de namespace y lo reasocia con otra instancia del mismo tipo de namespace.
- . Cada proceso tiene un subdirectorio que contiene los namespaces a los que está asociado: /proc/[pid]/ns.
- . Nos da la posibilidad de tener múltiples árboles de procesos anidados y aislados:



- . Permite aislar la tabla de montajes (montajes por namespace). Cada proceso, o conjunto de procesos, tiene una vista distinta de los puntos de montajes:



### *User Namespace*

Permite que los usuarios y grupos tengan sus propios IDs de usuario y grupo. Estos usuarios y grupos también existen en el host, pero con diferentes IDs.  
El usuario con ID 0 en el contenedor puede tener una identidad no root, no privilegiada, en el host. Por default, usuario nobody en el contenedor.  
No es necesario ser root para crear un user namespace.

ID-inside-ns ID-outside-ns length:

- ID-inside-ns: indica el UID inicial en el namespace.
- ID-outside-ns: indica el UID fuera del namespace.
- length: indica el número de mapeos de UID subsiguientes.

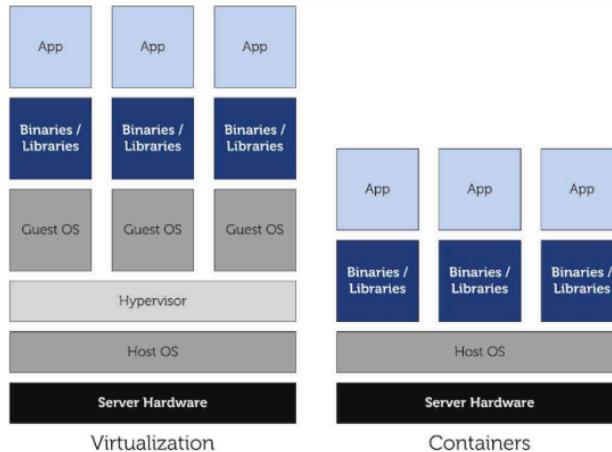
## **CONTENEDORES**

. Es una tecnología liviana de virtualización (lightweight virtualization) a nivel de sistema operativo que permite ejecutar múltiples sistemas aislados (conjuntos de procesos) en un único host.  
. Las instancias se 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.  
. Es el método de virtualización más eficiente: mejor performance, booteo más rápido.  
. No es posible ejecutar instancias de SO con kernel diferente al SO base.

### *Container*

- Una forma de empaquetar aplicaciones.
- Una máquina virtual liviana.
- Un conjunto de procesos aislados del resto.
- Dos tipos de contenedores:
  - 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.
- Principales características:
  - Autocontenidos: tiene todo lo que necesita para funcionar.
  - Aislados: mínima influencia en el nodo y otros contenedores.
  - Independientes: la administración de un contenedor no afecta al resto.
  - Portables: desacoplados del entorno en el que ejecutan. Pueden ejecutar de igual manera en diferentes entornos.

## *Hypervisor vs. Container*



- Containerization es la habilidad de construir y empaquetar aplicaciones como contenedores shippables.
- Un contenedor contiene un código específico y todas las librerías y dependencias necesarias para ejecutarse.
- Ejecutan de manera aislada en modo usuario usando un kernel compartido.
- Procesos en un contenedor tienen 2 IDs: uno en el contenedor y otro en el host (PID Namespace).
- Típicamente, cada contenedor provee un único servicio ("micro-servicio").
- Desde el lado del nodo host, un contenedor es un proceso (o conjunto de procesos) ejecutándose.
- En el nodo host se ven los procesos de todos los contenedores. Esto no es posible a la inversa ni entre distintos contenedores.

## **Clase 6 | Docker**

- . Docker es una plataforma para hacer contenedores, proveyendo todo lo que necesita una aplicación para ser ejecutada. Permite empaquetar y ejecutar una aplicación en containers livianos.
- . Docker Engine está dividido en 3 componentes:
  - Docker Daemon (dockerd): es el servidor. Responsable por crear, ejecutar y monitorear los contenedores, construcción de imágenes, etc.
  - API: especifica la interface que los programas pueden usar para interactuar con el servidor.
  - CLI: cliente. Permite a los usuarios interactuar con el servidor mediante comandos.
- . Docker propone una arquitectura Cliente/Servidor. Ambos pueden ejecutar en el mismo sistema (IPC o socket domain) o en diferentes nodos (socket TCP/IP). Se comunican usando la REST API de Docker. Cliente envía comandos HTTP. Ambos ejecutan en el espacio de usuario.
- . Docker “daemon” escucha por “API requests”.
- . Docker se utiliza para:
  - En desarrollo/testing.
  - Escalado y despliegue (deployment).
  - Más servicios en un equipo sin VMs.

## **. ¿Cómo Funciona?**

Docker es una herramienta que utiliza una serie de características del kernel 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. Fundamental la limitación para que todos puedan convivir.
- Union file systems: Se utilizan como filesystem de los containers. Docker puede utilizar overlay2, AUFS, btrfs, vfs y DeviceMapper(deprecated).

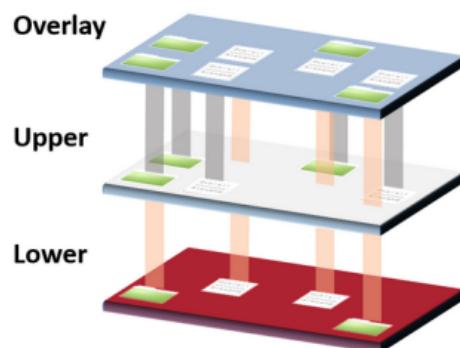
### **Union Mount FileSystem**

Es un mecanismo de montajes, no un nuevo file system.

Permite que varios directorios sean montados en el mismo punto de montaje, apareciendo como un único file system.

Read-only capas inferiores, writable capa superior.

Si hay 2 directorios que se llaman igual, solo muestra el de la capa de más arriba.



## **. Definiciones:**

### **Imagen**

Template (molde) de solo lectura con todas las instrucciones para construir un contenedor. Una imagen puede basarse en otras.

Cada imagen está compuesta de una serie de capas que se montan una sobre otra (stacking). Y cada capa es un conjunto de diferencias con la capa previa. Cada una de estas capas pueden ser reutilizadas entre las imágenes.

La capa modificable permite almacenar datos generados (temporales) durante la ejecución del contenedor. Esto es lo que permite extender las imágenes. Directorio modificable es el que permite correr múltiples contenedores a partir de las mismas capas (uno nuevo por cada contenedor).

Las capas se apilan:

- Cada capa que se baja, se extrae el contenido en un directorio del filesystem del nodo.
- Al ejecutar el contenedor desde una imagen, se genera unión-filesystem donde las capas se apilan una sobre otra.
- Usando chroot, se establece el unión-filesystem creado como directorio raíz del contenedor.
- Por último, se crea un nuevo directorio para el contenedor que permite modificar el filesystem (capa escribible de contenedor).

Docker usa “storage drivers” para almacenar capas de una imagen y para almacenar datos en la capa escribible de un contenedor.

## Container

Es una instancia de una imagen en ejecución.

## Registry

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

## Dockerfile

Archivo de texto que indica los pasos necesarios para construir una imagen.

### **. DockerFile:**

Las Imágenes se construyen siguiendo las instrucciones de un Dockerfile.

Cada capa en la imagen representa una instrucción (build instruction) en el archivo Dockerfile.

Si un comando modifica el filesystem se genera una nueva capa, excepto por aquellos que son solo informativos.

Al remover un archivo (RUN rm -f /home/user1/index.html) se genera una nueva capa.

Archivo sigue existiendo en la capa anterior (sigue ocupando espacio).

```
FROM ubuntu:24.10
MAINTAINER user1 <user1@example.com>
RUN apt-get -y update && apt-get -y install nginx
COPY index.html /usr/share/nginx/html/
ENV APP_USER=nginx-user
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

### **. Containers vs. VMs:**



### **. Contenedores:**

Un contenedor es una instancia de una imagen. La principal diferencia entre un contenedor y una imagen es la capa escribible. Al eliminar un contenedor esa capa también se elimina. Las capas inferiores se mantienen sin modificaciones.

Desde una imagen es posible generar varios contenedores.

Cada contenedor es autónomo y ejecuta en su propio entorno aislado. Todos los contenedores comparten el mismo kernel y ejecutan en sus propios namespaces.

Los contenedores pueden ser iniciados, detenidos, pausados y destruidos usando la CLI de Docker.

### **. Almacenamiento:**

Los archivos creados dentro de un contenedor son almacenados en una capa escribible. Al momento de definir el contenedor se le debe asignar/montar un disco para que el lo pueda ver y trabajar en él. Datos escritos en el contenedor no persisten cuando es destruido.

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

- Volumes: almacenados en una parte del filesystem administrada por Docker (por default: /var/lib/docker/volumes). Se asignan a los contenedores para que cada

contenedor trabaje con uno de ellos (de los volúmenes). Son vistos por el anfitrión que ve todo y por el contenedor. Los maneja Docker.

- Bind Mounts: pueden estar en cualquier parte del filesystem. Pueden ser modificados por procesos que no sean de Docker. No los maneja Docker.

Ambos deben ser montados en el contenedor.

Los volúmenes permiten una mejor portabilidad entre sistemas.

#### **. Networking (red):**

Por default, un contenedor ya tiene la red habilitada, con su IP, DNS, etc. El contenedor tiene salida hacia internet.

Un contenedor puede estar unido a más de una red o puede no estar unido a ninguno. Usuarios pueden definir nuevas redes.

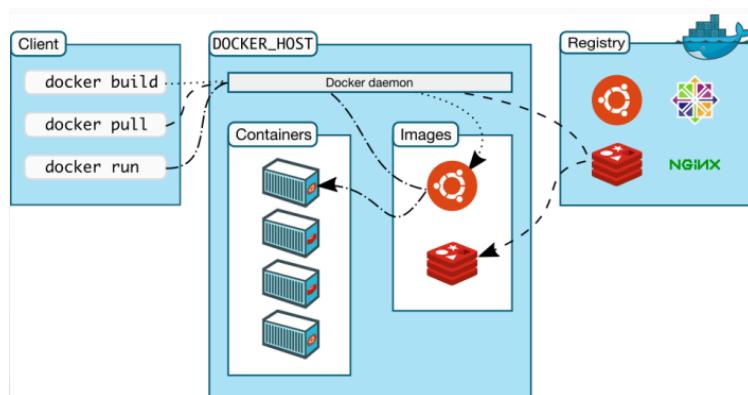
Múltiples contenedores pueden conectarse a la misma red y comunicarse usando direcciones IP y/o nombre.

Para hacer disponible un servicio el contenedor debe publicar el correspondiente puerto.

Dos contenedores en el mismo host no pueden publicar el mismo puerto.

Los contenedores usan los mismos servidores DNS que el nodo host, pero se pueden modificar.

#### **. Arquitectura:**



Al iniciar el sistema existe al menos un namespace de cada tipo.

Todos los procesos pertenecen a un namespace de cada tipo.

## **Clase 7 | Docker Compose, Podman**

### **DOCKER COMPOSE**

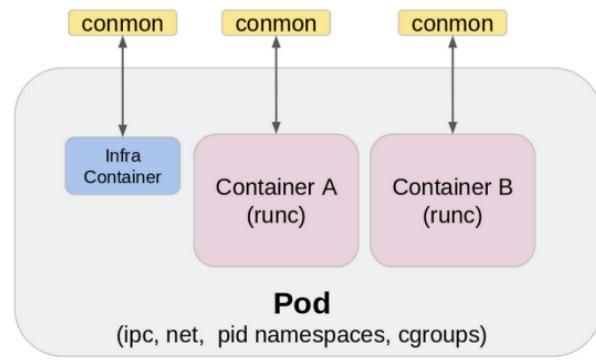
- . Docker Compose es una herramienta para correr aplicaciones que requieren múltiples contenedores.
- . Los contenedores son llamados servicios en Docker Compose.
- . Facilita la creación de servicios, almacenamiento y red mediante un archivo YAML. El archivo YAML se ubica en el directorio de trabajo y recibe el nombre compose.yaml por default.
- . También tiene comandos para iniciar, parar y construir servicios, monitorear los servicios en ejecución, logging, etc.
- . Cada servicio termina siendo un contenedor.
- . 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.

## **PODMAN**

- . Podman es una abreviación de Pod Manager.
- . Es un container engine daemonless para desarrollar, administrar y ejecutar contenedores OCI (Open Container Initiative) en sistemas Linux. Por esto no necesita un proceso demonio central para administrar los contenedores.
- . Un pod comprende uno o más contenedores que comparten los mismos namespaces.
- . Utiliza prácticamente los mismos comandos que Docker (incluso tiene un podman compose).
- . Permite ejecutar imágenes con el formato OCI, tanto como Docker (v1 y v2).
- . Los contenedores inician como procesos estándar del sistema. Es decir que no es necesario ser usuario root para iniciar contenedores.
- . Está basado en la librería libpod que contiene toda la lógica necesaria para instrumentar el ciclo de vida de un contenedor:
  - Formato de las imágenes, tanto Docker como OCI. Autenticación, descarga y almacenamiento de imágenes desde una registry, construcción de nuevas imágenes, etc.
  - Ciclo de vida de los contenedores: crear, ejecutar, eliminar, etc. contenedores.
  - Manejo tanto de simple contenedores como de pods.
  - Aislamiento de los contenedores/pods (mediante cgroups a bajo nivel).
  - CLI para administración de los contenedores/pods.
  - Soporte de contenedores/pods rootless.
- . *Pod:*
  - Concepto que proviene de Kubernetes en el que los contenedores se ejecutan en Pods.
  - Uno o más contenedores trabajando en conjunto con un propósito en común.
  - Representa un conjunto de contenedores que comparten almacenamiento y una única IP. Al estar todos juntos trabajan como localmente.
  - Servicios en los contenedores dentro del pod se pueden comunicar entre sí usando localhost.
  - Pods se pueden crear vacíos y luego agregarle contenedores.
  - Ventajas de agrupar dos o más contenedores:
    - Compartir algunos namespaces y cgroups.
    - Compartir volúmenes para almacenar datos persistentes.
    - Compartir la misma configuración.
    - Compartir el mismo IPC.
  - Cada pod incluye un contenedor llamado *infra*. Este contenedor *infra* se crea al crearse el pod aunque el mismo aun este vacío. También se lo suele llamar el pause container. Su finalidad es mantener abiertos los namespaces asociados con el pod.
  - Al agregarse un contenedor al pod, los procesos comparten varios namespaces del pod. Al compartir el net namespace, los procesos se comunican usando localhost (127.0.0.1). Es posible iniciar/detener un pod y/o un contenedor dentro de un pod.

- La mayoría de los atributos se asignan al contenedor infra: port binding, namespaces, cgroups.
- Si se desea cambiar un atributo se debe regenerar el pod (por ej. agregar un contenedor que escuche en un nuevo puerto).
- Por cada contenedor dentro del pod existe un proceso common. Common es un programa C liviano que monitorea un contenedor hasta que finaliza. Es una herramienta de comunicación entre el container engine (Podman) y el OCI runtime (runc o crun). Ejecuta el runtime, indicandole donde se encuentra el archivo OCI spec y el rootfs (capa que será el punto de montaje en el contenedor). Su principal tarea es monitorear el proceso principal del contenedor. Salva el código de salida si el contenedor muere. Mantiene la tty del contenedor abierta para poder conectarse a él.
- Contenedores comparten los namespaces de tipo PID, networking e IPC:



## CONTENEDORES ORQUESTADOR - CONCEPTOS

- . Son clusters, grupos 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 trabajo entre las distintas instancias de un servicio.
- . *Scaling*: aumentar/disminuir las instancias de un servicio según la carga de trabajo.

## Clase 8 | Protección y Seguridad

- . Los recursos informáticos deben protegerse frente a accesos no autorizados, destrucciones maliciosas o introducción accidental de incoherencias. Se deben proteger Datos/Información, CPU, Memoria, Dispositivos, etc.
- . El responsable de llevar a cabo la tarea es, entre otros, el Sistema Operativo, a través de un conjunto de mecanismos. Ejemplo las interrupciones, que sirven para que los procesos no toquen directamente el HW.
- . En base a la correcta o incorrecta aplicación de los mecanismos de protección, podremos determinar el nivel de Seguridad con el que cuenta el sistema.

## **PROTECCIÓN ≠ SEGURIDAD**

### *. 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 vs. Acceso a recursos en un sistema:

- Acceso al sistema se resuelve a través de:
  - Autenticación.
  - Control sobre una base de datos de usuarios.
- Acceso a recursos en un sistema:
  - Permisos.
  - Control de acceso (obligatorio o no).

### *. Seguridad:*

Medida de la confianza en que se puede preservar la integridad de un sistema y sus datos. Es un concepto más general, que se va dando a medida que vamos aplicando mecanismos de protección.

La seguridad 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 ≠ Mecanismos:*

Las políticas son lo primero que necesitamos definir antes que los mecanismos.

Las políticas (el qué) definen lo que se quiere hacer, en base a los objetivos. Las podemos asociar a los papeles. Rara vez incluyen configuraciones. Una vez que definimos 'que' queremos, podemos comenzar a buscar cuales son las mejores tecnologías para eso.

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

Esta separación entre políticas y mecanismos se realiza para garantizar la flexibilidad de un sistema. Ya que muy probablemente sea siempre lo mismo lo 'que' queremos para la seguridad, pero los mecanismos pueden ser distintos por ejemplo en distintos Sistemas Operativos (Windows/MacOs).

## **OBJETOS Y DOMINIOS**

**. Objeto:** es cualquier recurso que puede ser accedido o manipulado por un proceso.

Ejemplos: archivo, sockets, dispositivos, memoria, CPU, etc.

**. Dominio:** representa el contexto de ejecución de un proceso o conjunto de procesos.

Define qué *permisos* (rights) tiene el proceso para acceder a objetos.

**. Right:** es el permiso que tiene un dominio para realizar una acción sobre un objeto.

Ejemplos: leer, escribir, ejecutar, borrar, etc.

. Un sistema informático es una colección de procesos y objetos. Los objetos pueden ser de HW (CPU, Memoria, etc.) o de SW (archivos, programas, semáforos).

Cada objeto debe tener un identificador único que permita referenciarlo. 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.

¿Quién se puede ejecutar en un dominio?:

Puede ser un usuario y define qué puede hacer ese usuario (por defecto lo que no se permite se deniega).

Puede ser un proceso y el conjunto de objetos a los que podrá acceder dependerá de la identidad del proceso.

Puede ser un procedimiento y definirá el conjunto de variables a las que puede acceder (variables locales, globales, etc...).

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

. Por ejemplo: el dominio D contiene la pareja (archivo A, {read,write}). Entonces, un proceso que se ejecuta dentro del dominio D puede leer y grabar el archivo A.

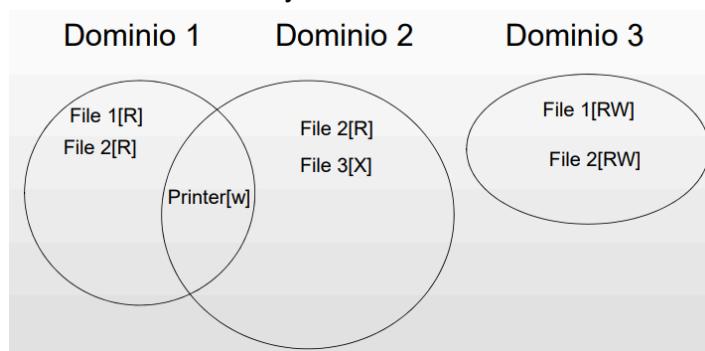
. El Principio de need-to-know o POLA (Principle of least authority): define que los procesos accedan sólo a los objetos que necesitan (con los derechos que necesiten) para completar su tarea.

. La relación entre un proceso y un dominio puede ser estática o dinámica:

- *Relación estática*: si el conjunto de objetos a los que el proceso accede durante su ciclo de vida es fijo. Siempre está en el mismo dominio. Puede generar que los procesos tengan más privilegios que los que necesitan en sus fases de ejecución porque tienes que dar todos los permisos a la vez.

- *Relación dinámica*: si el conjunto de objetos puede variar. Puede cambiar de dominio.

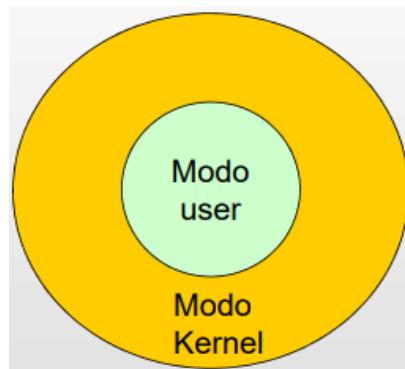
Por ejemplo usando los bits SETUID y SETGID en UNIX sobre los archivos.



#### Ejemplo 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.



. POLA (Principle of Least Authority) establece que un proceso, programa o usuario debe tener sólo los permisos estrictamente necesarios para realizar su tarea y nada más.

Este principio mejora la seguridad del sistema al limitar el daño que puede causar un error o una acción maliciosa.

### **MATRIZ DE ACCESO**

. La matriz de acceso me permite determinar todo lo que se puede hacer y lo que no.

. 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		

#### *. Operación Switch:*

Un proceso puede cambiar de un dominio a otro. 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 commutación del dominio  $D_i$  al dominio  $D_j$  estará permitida si se encuentra definida en el  $switchaccess(i,j)$ .

Dominio y matriz son objetos sobre los cuales se definen accesos.

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

#### . Operación Copy:

Es un derecho que se asocia a un elemento access(i,j) de la matriz.

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

Se denota con \*.

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

#### . Operación Owner:

Permite agregar nuevos derechos y borrar los ya existentes.

Si matriz(i,j) incluye el derecho de owner entonces un proceso ejecutándose en el dominio

Di puede agregar y borrar cualquier entrada en la columna j.

Objeto Dominio	File1	File2	File3	Printer
Dominio				
D1	Read	Read		Print
D2		Read * owner	execute	print
D3	Read/write			
Objeto Dominio	File1	File2	File3	Printer
D1	Read	Read/write		Print
D2		Read * owner	execute	print
D3	Read/write	Read		

#### . Operación 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

Di puede remover cualquier derecho de acceso dentro de la fila j.

Objeto	File1	File2	File3	Printer	D1	D2	D3
Dominio							
D1	Read	Read		Print		switch	control
D2		Read	execute	print			
D3	Read/ Write	Read/ write			switch		switch

Cualquier proceso que se ejecute en el dominio D1, puede controlar los Switch Access de cualquier columna del dominio D3.

#### . Resumen de Operaciones:

Copy, owner y control son operaciones que se utilizan para controlar cambios al contenido de la matriz de acceso.

Switch y Control: son aplicables sólo a dominios.

Copy y owner: puede modificar derechos dentro de una columna.

Control: puede modificar derechos dentro de una fila.

#### . Implementación:

La forma de representar y almacenar la matriz de acceso, generalmente no es bajo el formato de matriz, ya que se desperdiciaría mucho espacio (es ineficiente).

El principal problema es que puede tener muchos elementos vacíos.

Generalmente se almacenan sólo los elementos ocupados utilizando 2 métodos:

- Almacenarla por filas.
- Almacenarla por columnas.

Se debe optimizar el acceso para que sea rápido.

#### Tabla Global

Es la más sencilla de implementar.

Consiste en el conjunto de tuplas <dominio, objeto, derechos-acceso>.

Cada vez que se ejecuta una operación M sobre un objeto O<sub>j</sub> sobre el dominio D<sub>i</sub>, se analiza la tabla y se verifica si se encuentra una terna < D<sub>i</sub> , O<sub>j</sub> , M >:

- 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.

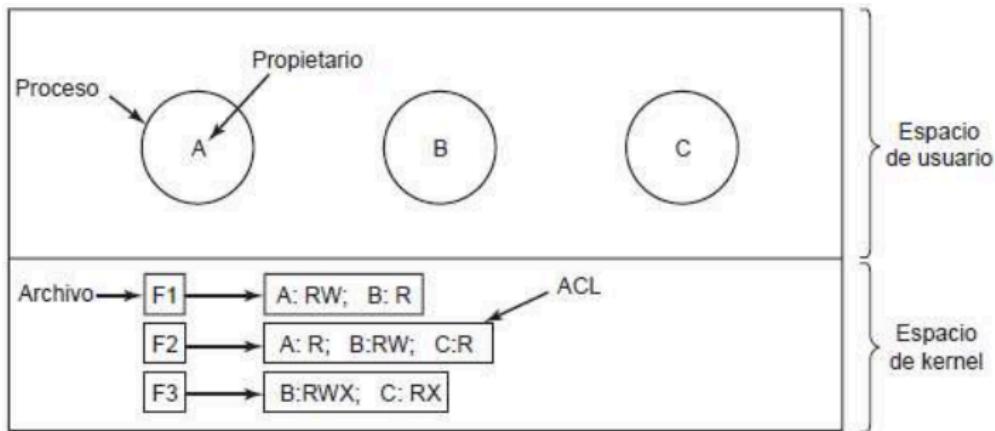
Dominio	Objeto										
	Archivo1	Archivo2	Archivo3	Archivo4	Archivo5	Archivo6	Impresora1	Plotter2	Dominio1	Dominio2	Dominio3
1	Lectura	Lectura Escritura								Enter	
2			Lectura	Lectura Escritura Ejecución	Lectura Escritura		Escritura				
3						Lectura Escritura Ejecución	Escritura	Escritura			

### 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. 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  $R_k$ .



### Lista de Capacidades

Es una lista de objetos del dominio con sus derechos (división x filas).

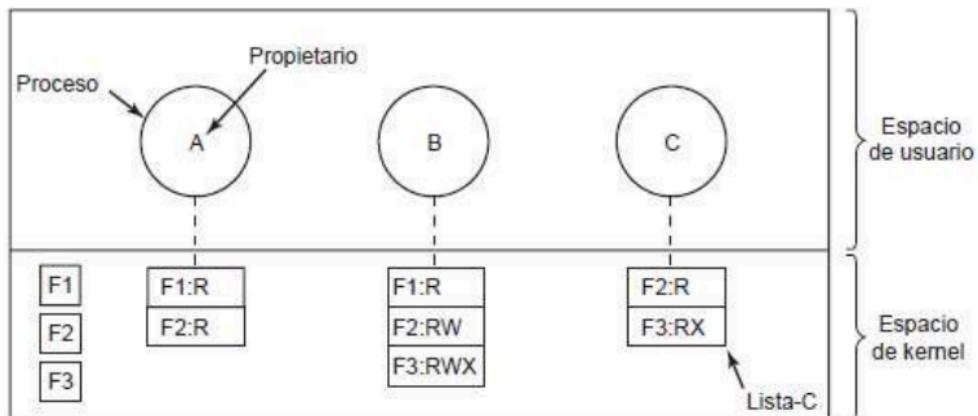
A esta lista se le conoce como lista de capacidades y a los elementos individuales que contiene se les conoce como capacidades. El proceso no la accede directamente.

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 ya que se deben recorrer todas las listas de capacidades.

Cada capacidad otorga al propietario ciertos derechos sobre un objeto. Cada proceso tiene una lista con los objetos que puede utilizar, junto con qué operaciones (el dominio).

Por ejemplo, el proceso que pertenece al usuario A puede leer los archivos F1 y F2:



## **SISTEMAS CONFIABLES**

. Otro factor que afecta el nivel de seguridad de un sistema, es el código:

- Código mal intencionado: Virus, gusanos, troyanos, etc.
- Código con errores de programación: Backdoors.
- . ¿Es posible construir sistemas seguros? La respuesta es sí, pero el gran enemigo de la seguridad es el agregado de funcionalidad.
  - Un sistema minimalista será probablemente muy seguro, pero pobremente usable.
  - Un sistema con muchas características, probablemente tenga más errores y abra más la posibilidad a que sea vulnerado.

. *Exploitación de errores en código:*

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 sobreescribir datos de ciertas zonas de memoria intencionalmente. Se trata de un error del software. Se quiere copiar una cantidad de datos más grande que el área definida.
- Cadenas de formato.
- Retorno a libc.
- Desbordamiento de enteros.
- Inyección de código.

## **Clase 9 | Multiprocesadores**

. Desde su inicio, la industria de las computadoras se ha orientado fundamentalmente a buscar un poder de cómputo cada vez mayor. Las necesidades actuales demandan cada vez mayor poder de cómputo, y los procesadores actuales ya no alcanzan a cubrir todas las necesidades.

. En la actualidad, lograr mayor velocidad es físicamente complejo ya que ninguna señal eléctrica se puede propagar más rápido que la velocidad de la luz. Además hay problemas de disipación de calor y problemas de consumo eléctrico.

. La solución al problema es el cómputo en paralelo y/o distribuido. Contar con varias CPU que operen a velocidad “normal” y que en conjunto provean la potencia de cómputo necesaria.

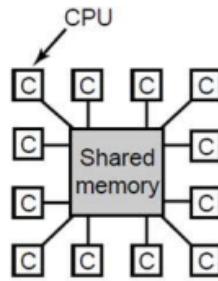
. Si tenemos que resolver un problema en una única CPU, el esquema de trabajo es sencillo, pero si tenemos varios problemas y varias CPU, a priori podríamos asignar estáticamente una tarea a cada una de ellas → no es lo más eficiente, por lo que deberá existir un coordinador que se encargue de repartir las tareas.

Al existir múltiples CPU, la complejidad aumenta en lo que refiere a distribución de tareas, pasaje de mensajes y acceso a memoria.

## **ESQUEMAS**

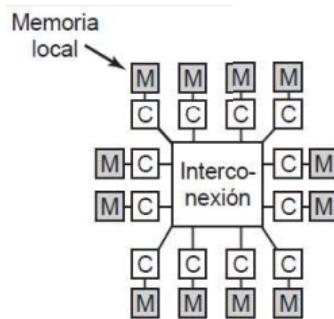
. *Multiprocesador con Memoria Compartida:*

- La comunicación entre las CPU es a través de la memoria compartida.
- Cada CPU tiene el mismo acceso que otras a la memoria física a través de un único BUS físico.
- Para acceder a una palabra de memoria por lo general cada CPU requiere de 2 a 10 nseg.
- Existe un único espacio lógico de direcciones para todos los procesos.



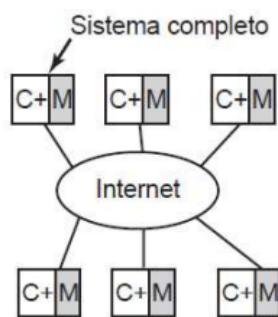
. *Multicomputadora con Memoria Independiente/Pasaje de Mensajes:*

- Varios pares de CPU-memoria se conectan a una interconexión de alta velocidad pasando mensajes.
- Cada memoria es local para una sola CPU y puede ser utilizada sólo por esa CPU.
- El retardo del paso de mensajes es de entre 10 a 50  $\mu$ seg.



. *Sistemas Distribuidos:*

- Conecta sistemas de cómputo completos a través de una red.
- Cada sistema tiene su propia memoria, y se comunican mediante el paso de mensajes.
- El retardo del paso de mensajes es de entre 10 a 100 mseg.
- Cada nodo de cómputo es una computadora completa.
- Es más rápido porque tiene mayor latencia al trabajar por redes.
- Heterogeneidad de sistemas y hardware. Esto permite que no todos los nodos del cluster sean iguales ni tengan la misma arquitectura.



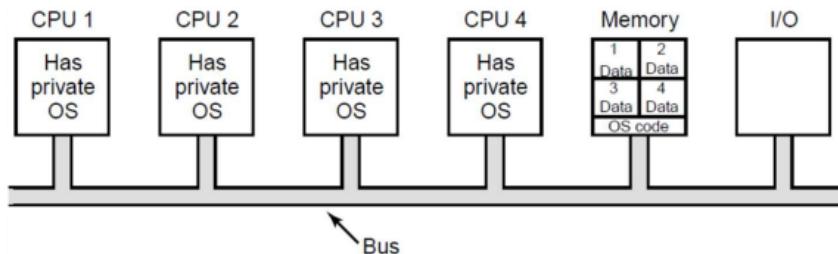
## **CHIPS MULTINÚCLEO**

- . A medida que la tecnología avanza, los transistores se hacen mas pequeños y aparece la posibilidad de agregar más de uno a un chip.
- . Al tener más transistores se puede:
- Agregar más memoria caché → Está demostrado que la tasa de aciertos no se incrementa demasiado.

- Agregar más velocidad de clock a una CPU → Sigue existiendo un único hilo de ejecución.
- Agregar más CPU al mismo chip (núcleos), los que podrían compartir la caché y la memoria principal → Se logra paralelismo.

## **TIPOS DE SO MULTIPROCESADOR**

- . El SO en sistemas multiprocesador tiene varias responsabilidades:
  - Manejo de System Calls.
  - Administración de Memoria.
  - Administración de E/S (sist. de archivos y dispositivos).
  - Sincronización de procesos.
  - Administración de Recursos.
  - Planificación de CPU.
- . Existen diversas metodologías posibles para la administración en esquemas multiprocesadores desde el lado del software:
  - . *Cada CPU con su SO:*  
Se divide estáticamente la memoria para cada CPU con su copia privada (las CPUs operan independientes). Se comparte el código de SO.  
Cada CPU cuenta con su propio conjunto de procesos, lo cual genera un desbalance en la carga de trabajo.  
Hay planificación independiente por CPU.



Cada CPU atrapa y maneja las SysCalls de sus procesos, por lo que los procesos quedan “atados” a una única CPU. No se pueden compartir páginas, solo pasaje de mensajes, lo que genera memoria desperdiciada.

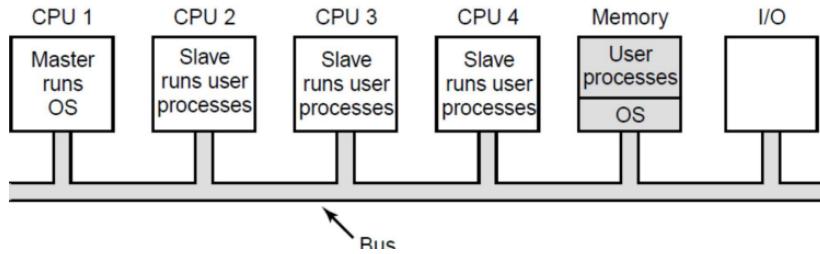
En cuanto a la caché de disco, cada CPU tiene su propia copia → Inconsistencia de la información.

Puede haber problemas con por ejemplo sistemas C/S ya que un servidor produce y deposita en su caché, y el cliente al querer consumir no puede acceder a la caché de la otra CPU. Entonces es necesario mantener una coherencia de caché a través de un mecanismo, y ante cada inconsistencia se debe ejecutar el mecanismo para que no ocurran → Ineficiencia.

. *Maestro - Esclavo:*

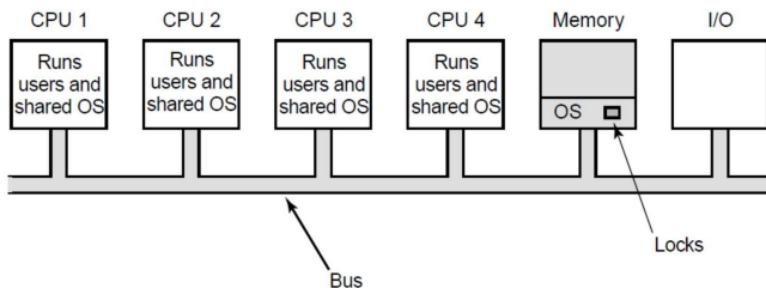
Hay una única copia del SO y de su información.

Todas las SysCalls se redirigen a una CPU. Esta CPU puede ejecutar procesos si “le sobra tiempo” → Se genera un gran cuello de botella.



#### . SMP - Multiprocesadores Simétricos:

Soluciona el inconveniente de saturación de una única CPU: mantiene una única copia del SO en memoria y cualquier CPU puede ejecutarlo. Cuando se invoca una System Call, es ejecutada por la CPU que la invocó.



Provee equilibrio entre procesos y memoria, ya que solo hay un único conjunto de tablas del SO.

No hay cuello de botella, ya que no hay una CPU master.

Problemas:

- Dos o más CPUs ejecutando código del SO en un mismo instante de tiempo.
- Dos CPUs seleccionando el mismo proceso para ejecutar, o seleccionan la misma página de la memoria libre!

Estos problemas, sin una adecuada sincronización genera bloqueos.

Possibles soluciones a los problemas planteados:

1. Utilizar "locks" para las estructuras del SO: Considerar a todo el SO como una gran sección crítica. Cualquier CPU puede ejecutar el SO, pero solo una a la vez. Se comportaría como el modelo maestro-esclavo → Se necesita apoyo del HW.
  2. Lock por estructura(s): Existen varias secciones críticas independientes, cada una protegida por su propio mutex. Esto mejora el rendimiento, pero genera una dificultad para determinar cada sección crítica. Ciertas estructuras pueden pertenecer a más de una sección crítica, lo cual ante bloqueos podría generar Deadlocks.
- Es el esquema que generalmente se utiliza.

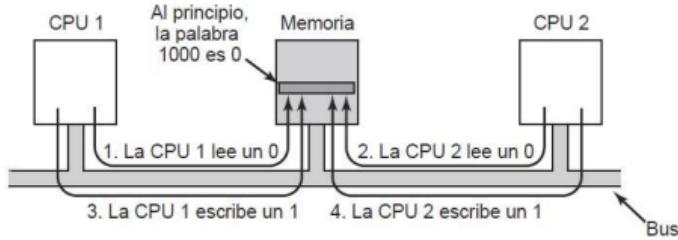
## **SINCRONIZACION DE MULTIPROCESADORES**

. En entornos uniprocesador si un proceso realiza una llamada al sistema que requiera acceder a cierta tabla crítica del kernel, el código del kernel sólo tiene que deshabilitar las interrupciones antes de tocar la tabla. En sistemas multiprocesadores, se deshabilitan las interrupciones de una CPU, pero otra podría generarlas...

. Surge la necesidad de contar con un protocolo de mutex apropiado para garantizar la exclusión mutua.

. Una posibilidad para garantizar la exclusión mutua es el uso de TSL (Probar y establecer bloqueo):

- Lee la palabra de memoria y la almacena en un registro. Al mismo tiempo escribe un 1 en la memoria para hacer el lock (2 accesos al BUS). Cuando termina libera (escribe 0). En uniprocesadores esta implementación es correcta.
- El problema surge en entornos multiprocesadores: (la operación no es indivisible).



- Si ambas CPU obtuvieron un 0 de la instrucción TSL, por lo que ambas tienen acceso a la sección crítica → problema!
- . La solución al problema anterior, es que en multiprocesadores la instrucción TSL bloquee el acceso al BUS:
  - Se necesita soporte de hardware para poder implementarlo.
  - Genera Carga en la memoria y el BUS, ya que la CPU que solicita debe mantener el bloqueo y las otras CPU deben esperar liberación del bloqueo.
  - No es lo más eficiente...

## PLANIFICACION DE MULTIPROCESADORES

. Antes de analizar cómo se va a hacer la planificación, es necesario determinar que se va a planificar:

- Uniprocésador : ¿Qué hilo (o proceso) se seleccionará?
- Multiprocésador: Se suma ¿En qué CPU se va a ejecutar?
- Complejidad con hilos que trabajan “en conjunto”:
  - Si son ULT, el planificador los planifica a nivel de proceso.
  - Si son KLT, es posible tomar decisiones sobre su planificación.

### *. Planificación de Hilos Independientes:*

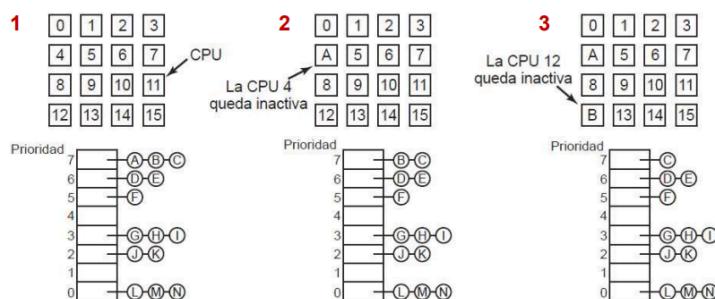
Esta situación se da generalmente en ambientes de tiempo compartido.

Muchos usuarios ejecutando tareas que generalmente no tienen relación entre sí.

El esquema más sencillo para planificarlos es tener una única cola de listos para todos los hilos.

Podríamos tener varias... Una para cada prioridad.

Ejemplo, las 16 CPU están ocupadas y hay 14 hilos en la cola de listos ordenados por prioridad:



Se aprovecha la caché:

- Un hilo que se ejecuta en un CPU donde ya se ha ejecutado, tendrá mayor posibilidad de que sus datos aún sigan en la caché de dicha CPU.

- Se utiliza el algoritmo de Planificación de 2 niveles:

Cuando se crea un hilo, se asigna a una CPU.

Cada CPU tiene su propia colección de hilos y los planifica por separado.

Si queda una CPU ociosa, se reparten los hilos.

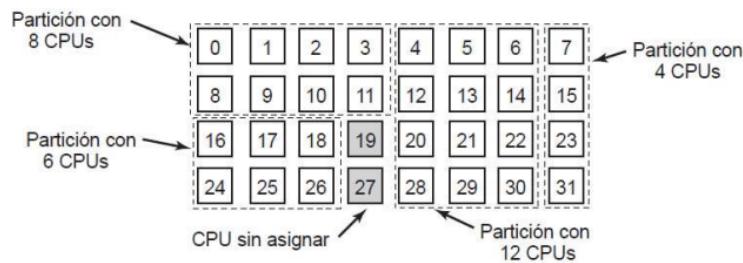
Minimiza la contención de las estructuras de datos asociadas para la planificación, ya que no hay solo una (una o más cola de listos por cpu).

#### *. Planificación de Hilos que trabajan en Conjunto:*

Se planifican en conjunto (en varias CPUs) ya que son tareas que tienen que ver entre sí.

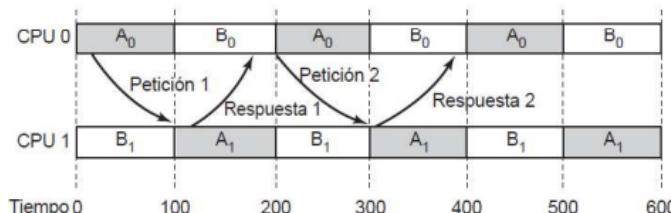
Mejora en trabajos en paralelo.

El grupo se planifica si hay CPUs libres para cada hilo del grupo. Si no las hay, el grupo espera. Por esto no hay multiprogramación por CPU (cada CPU ejecuta solo 1 hilo), baja la productividad.



Se podrían multiprogramar las CPU, pero podría ocurrir que los hilos no se ejecuten sincrónicamente, ya que se planifican independientemente.

Supongamos una situación de 2 hilos A0 y A1 que se ejecutan intercambiando mensajes compartiendo CPU con los hilos de un proceso B:



Como A0 se ejecuta en un intervalo distinto que A1 ocurre que el lapso de ejecución de A es de 200 mseg, cuando podría haber sido de 100 si se hubieran ejecutado en el mismo intervalo.

Por esto el planificador ahora además se debe preocupar por planificar de forma inteligente para optimizar los tiempos → Incrementa muchísimo la complejidad.

Otra opción es la **planificación por pandillas**:

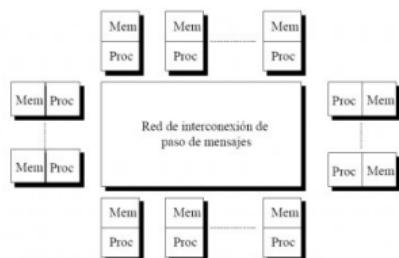
- Los hilos relacionados se toman como una Pandilla.
- Todos los miembros de una pandilla se ejecutan simultáneamente en distintas CPUs multiprogramadas.
- Todos los miembros de la pandilla inician y terminan sus intervalos en conjunto.
- Ejemplo: Supongamos un multiprocesador con 6 CPU utilizadas por 5 procesos (A..E) y un total de 24 hilos. En el instante de tiempo 0, se programan los hilos A0.. A5 . Luego se ejecuta B<sub>0</sub>, B<sub>1</sub>, B<sub>2</sub>, C<sub>0</sub>, C<sub>1</sub>, C<sub>2</sub>, etc... Se continúa la ejecución cíclicamente.

	CPU					
	0	1	2	3	4	5
Time slot	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>
1	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>
2	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	E <sub>0</sub>
3	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>
4	A <sub>0</sub>	A <sub>1</sub>	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>
5	B <sub>0</sub>	B <sub>1</sub>	B <sub>2</sub>	C <sub>0</sub>	C <sub>1</sub>	C <sub>2</sub>
6	D <sub>0</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	E <sub>0</sub>
7	E <sub>1</sub>	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>

- La idea de ejecutar todos los hilos de un proceso juntos, permite que el pasaje de mensajes se realice de manera más rápida y eficiente.

## MULTICOMPUTADORAS

- . También conocidos como cluster de computadoras, son CPUs con acoplamiento fuerte. Cuentan con un bus de interconexión para pasaje de mensajes.
- . No comparten memoria, cada una tiene la suya propia. Poseen además mucha CPU, memoria y placas de interconexión redundantes.
- . Es necesario diseñar correctamente la red.
- . Son PCs con una interfaz de red de alto rendimiento.



### *. Software de Comunicacion a Nivel Usuario:*

Para comunicarse, los procesos en distintas CPUs en una multicamputadora se envían mensajes entre sí.

Se utiliza entonces pasaje de mensajes con Send y Receive con y sin bloqueo.

Otra alternativa es RPC (Remote Procedure Call) para obtener un mayor nivel de abstracción.

### Planificación

Cada nodo tiene su propio conjunto de procesos.

Un nodo no toma procesos de otros para ejecutarlos, ya que sería bastante costoso.

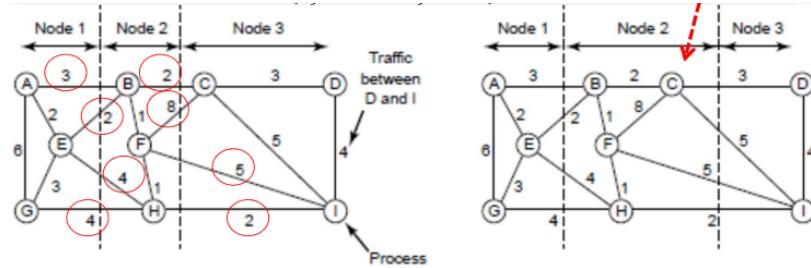
Es importante la asignación de procesos a los nodos → Balanceo de la carga.

Se puede aplicar el concepto de planificación por pandillas → Sincronización entre los nodos en cada inicio de una ranura de tiempo.

El balanceo de la carga se representa al sistema como un grafo donde cada nodo es un proceso, y la comunicación entre ellos se representa a través de una arista, se debe particionar el grafo en tantos subgrafos como nodos se tengan, teniendo en cuenta:

Requerimientos totales de CPU, Requerimientos totales de memoria, Minimizar la cantidad de aristas entre nodos de distintos subgrafos (tráfico de red), y Buscar clusters con acoplamiento fuerte.

Ejemplo: Tenemos 9 procesos, sabiendo el costo de comunicación entre estos y tenemos 3 nodos, tráfico de la red, suma de los pesos entre enlaces de 2 nodos diferentes (ej1: 30 , ej2: 28).



Ahora el planificador además debe ver todo lo anterior, deberá encargarse de optimizar el costo de la comunicación entre nodos.

Otra forma de balancear la carga es que un nodo con poca carga informe sobre la situación:



## SISTEMAS DISTRIBUIDOS

- . Los sistemas distribuidos aparecen cuando un solo data center dejó de alcanzar para sostener todo, entonces se vuelve necesario repartir la carga en varios data centers y comunicarlos entre sí.
- . Los Sistemas Distribuidos son similares a las multicomputadoras ya que cada nodo tiene su propia memoria privada, tienen menor acoplamiento ya que se pueden encontrar esparcidos por todo el mundo.
- . Cada nodo es una PC completa, incluyendo sus dispositivos (MEM, CPU, etc). Además, cada nodo puede ejecutar un SO y Hardware diferente, incluyendo su propio sistemas de archivos → Heterogeneidad.
- . Deben comunicarse entre sí para mostrarse como un único sistema consistente y coherente → Middleware.

### . Definición:

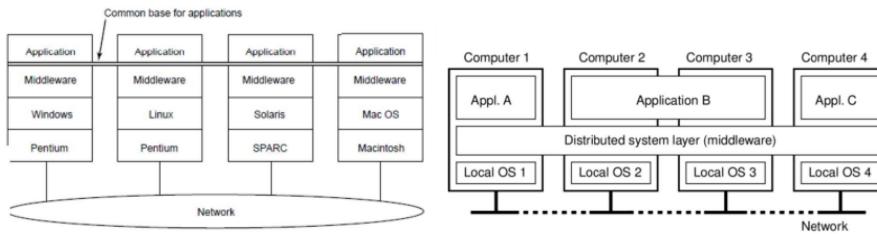
*“Un sistema distribuido es un conjunto de computadoras independientes que a sus usuarios les aparece como un solo sistema coherente.”*

*“Un sistema distribuido es aquel en el que los componentes ubicados en computadoras en red se comunican y coordinan sus acciones solo pasando mensajes.”*

### . Middleware:

Para que los distintos nodos heterogéneos puedan mostrarse como un único sistema aparece el middleware.

Es una capa de software por encima del SO que permite una uniformidad entre los distintos SOs.



El middleware es una capa de Software fundamental en un Sistema distribuido ya que:

- Provee una interfaz común a todos los procesos.
- Soluciona los problemas de heterogeneidad.
- Provee servicios a las capas superiores.
- Provee estructuras de datos y operaciones que permiten a los procesos y usuarios inter-operar, de manera consistente, entre máquinas remotas.

## Clase 10 | Deadlocks

. Un conjunto de procesos están en deadlock cuando cada uno de ellos está esperando por un recurso que está siendo usado por otro proceso del mismo conjunto. Un estado de Deadlock puede involucrar recursos de diferentes tipos.

### **RECURSOS**

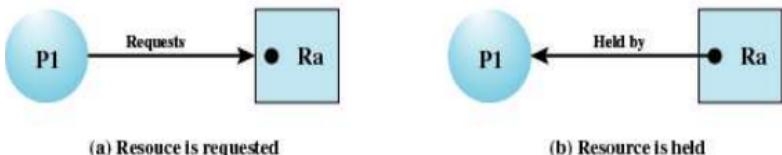
- . **Recursos físicos:** CPU, memoria, dispositivos.
- . **Recursos lógicos:** archivos, registros, semáforos, etc.
- . **Recursos apropiativos:** se le pueden quitar al proceso sin efectos dañinos (ej: memoria, CPU).
- . **Recursos no apropiativos:** si se le saca al proceso, éste falla (interrumpir una escritura a CD, impresora).
- . Cada recurso  $R_j$  puede tener  $W_i$  instancias idénticas (puede haber 2 impresoras del mismo tipo). Si son idénticas, se puede asignar cualquier instancia del recurso.
- . **Clase de Recursos:** Es el conjunto de instancias de un recurso. Está determinada por el Major y el Minor Number.
- . Siguiendo un modo de operación normal, un proceso emplea un recurso siguiendo la siguiente secuencia:
  1. Solicitud: Si no puede ser concedida inmediatamente, el proceso deberá esperar a que el recurso sea liberado.
  2. Uso: El proceso puede operar sobre el recurso.
  3. Liberación: Se libera el recurso para que pueda ser utilizado por otro proceso.
 Si el recurso que se quiere utilizar está ocupado, se sigue un “Ciclo corto de solicitud” →
  1. Solicitud fallida, 2. Espera inactiva, 3. Nuevo intento de solicitud.

#### *. Representación de Procesos y Recursos:*

Para poder representar la asignación de recursos, se utiliza un Grafo de Asignación de recursos. El grafo permite visualizar el estado de los recursos del sistema y procesos en un momento determinado.

Cada Proceso o recurso es representado por un nodo. Un recurso con varias instancias posee más de un “Punto” dentro del nodo.

Una arista representa una relación entre un Proceso y un Recurso. Notar que las aristas son dirigidas y dependiendo de la dirección indican distintos estados.



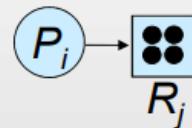
## Proceso



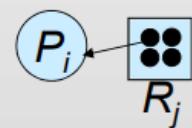
## Tipo de Recurso con 4 instancias



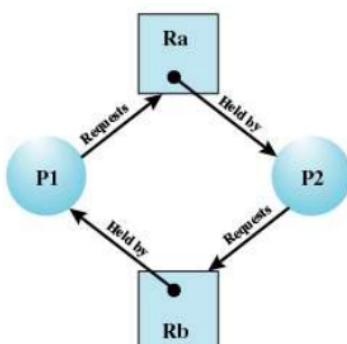
$P_i$  solicita una instancia de  $R_j$



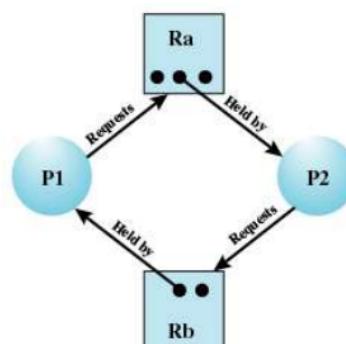
$P_i$  tiene asignada una instancia de  $R_j$



Ejemplo con varias instancias:

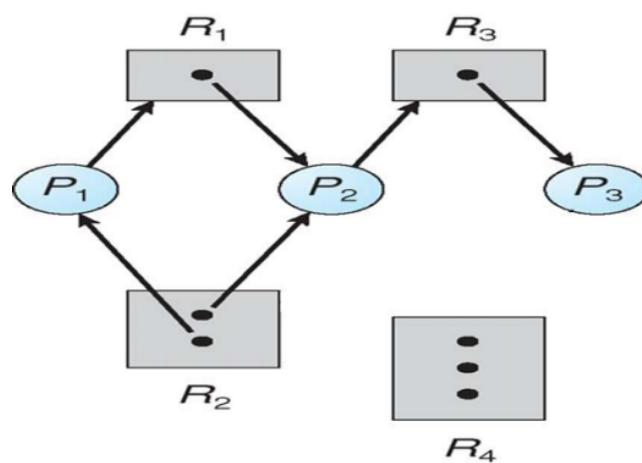


(c) Circular wait



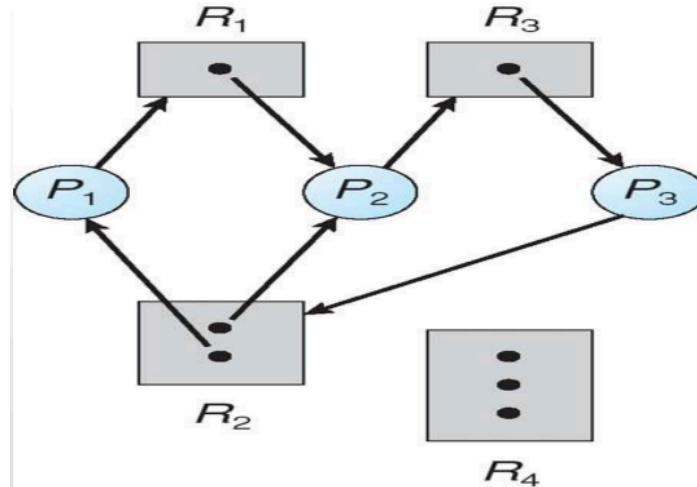
(d) No deadlock

Ejemplo de estado inseguro, donde *puede* haber deadlock, pero no es seguro que lo haya:



Aquí puede no haber deadlock porque no hay espera circular.

En este otro caso si hay deadlock, ya no está en estado inseguro. Acá si hay una espera circular.



. *Condiciones para que se cumpla deadlock:*

#### Hechos

Si el grafo no contiene ciclos → NO hay interbloqueo.

Si el grafo contiene un ciclo:

- Si sólo hay una instancia por tipo de recurso → SI hay interbloqueo.
- Si hay varias instancias por tipo de recurso → hay posibilidad de deadlock.

#### Condiciones para que NO Ocurra Deadlock

1. **Exclusión mutua:** En un instante de tiempo dado, sólo un proceso puede utilizar una instancia de un recurso.
2. **Retención y espera:** Los procesos deben mantener los recursos asignados y esperar por la asignación de los nuevos requeridos.
3. **No apropiación:** Los recursos no pueden ser quitados a un proceso que actualmente los posea.
4. **Espera circular:** El proceso forma parte de una lista circular en la que cada proceso de la lista está esperando por al menos un recurso asignado a otro proceso de la lista.

*Para estar en presencia de un Deadlock se deben cumplir todas!*

## **MÉTODOS PARA EL TRATAMIENTO DEL DEADLOCK**

Existen distintos enfoques con el fin de llevar adelante el manejo de Deadlocks:

1. *Usar un protocolo que asegure que NUNCA se entrará en estado de deadlock:*

#### 1.1 Prevenir

Se intenta que no se cumple alguna de las 4 condiciones:

**Condición de Exclusión Mutua:** Si ningún recurso se asignara de manera exclusiva (no siempre se puede), no habría interbloqueo.

Considerar que hay recursos compatibles (archivos read only, memoria) y no compatibles (impresora). A veces, mantener la exclusión mutua para recursos compatibles, es muy complejo e ineficiente.

**Condición de Retención y Espera:** Se basa en que si un proceso requiere un recurso que no está disponible, debe liberar otros.

Podemos utilizar alguna alternativa:

1. Un proceso debe requerir y reservar todos los recursos a usar antes de comenzar la ejecución (precedencia de las system calls que hacen el requerimiento antes de cualquier otra system call).
2. El proceso puede requerir recursos sólo cuando no tiene ninguno (al comienzo de su ejecución generalmente).

Las desventajas son la baja utilización de recursos y puede ocurrir inanición de algún proceso.

**Condición de No Apropiación:** NO siempre puede atacarse esta condición, ya que no siempre puede expropiarse un recurso a un proceso.

Una posible solución es virtualizar los recursos: El proceso no accede directamente al recurso, sino que accede a un demonio que lo administra. Solo el demonio tiene acceso al recurso, y a medida que los procesos requieren el recurso, interactúan con el demonio quien almacena los trabajos en una cola (spooler). De esta manera se logra que el recurso físico pueda ser “utilizado” por varios procesos al mismo tiempo.

**Condición de Espera Circular:** Se define un ordenamiento de los recursos. Luego, un proceso puede requerir recursos en un orden numérico ascendente. La función F asigna un número único a cada recurso (los números pequeños para recursos muy usados).

Un proceso, que ya tiene  $R_i$  puede requerir  $R_j$  si y sólo si  $F(R_i) < F(R_j)$  (solicita recursos con número mayor a los ya asignados).

Como no se puede dar que  $F(R_i) > F(R_j)$  y  $F(R_j) < F(R_i)$ , podemos garantizar que no habrá un bucle en el grafo de asignación de recursos.

Ejemplo:

$$F(CD)=1; F(disco duro)=4, F(impresora)=7.$$

Un proceso que ya tiene asignado el disco, puede pedir la impresora (pues  $F(impresora) > F(disco duro)$ ). Si ya tiene la impresora, no puede solicitar el CD.

### 1.2 Evitar

Tomar decisiones de asignación en base al estado del sistema.

Requiere que el SO tenga información ANTES sobre el uso de recursos.

El SO cuenta con información sobre el uso de los recursos:

Cómo son requeridos por parte de los procesos.

En qué momento son requeridos.

La demanda máxima, etc.

La técnica se basa en tomar decisiones acerca de la asignación de los recursos, con el fin de que no se llegue a un estado de Deadlock.

En todo momento el SO toma decisiones dinámicas acerca de la asignación. Si en algún momento se evalúa que podría entrarse en un estado de Deadlock, la asignación de recursos es denegada.

Desventajas en implementación: puede producir una baja utilización de los recursos y de la performance del sistema debido a los cálculos que deben realizarse.

### Estado Seguro

Un sistema está en un estado seguro si el SO puede asignar recursos a cada proceso de un conjunto de alguna manera, evitando el deadlock.

Cuando un proceso solicita un recurso disponible, el sistema DEBE DECIDIR si la asignación inmediata deja el sistema en un estado seguro.

Un estado seguro me garantiza que el sistema no va a entrar en deadlock.

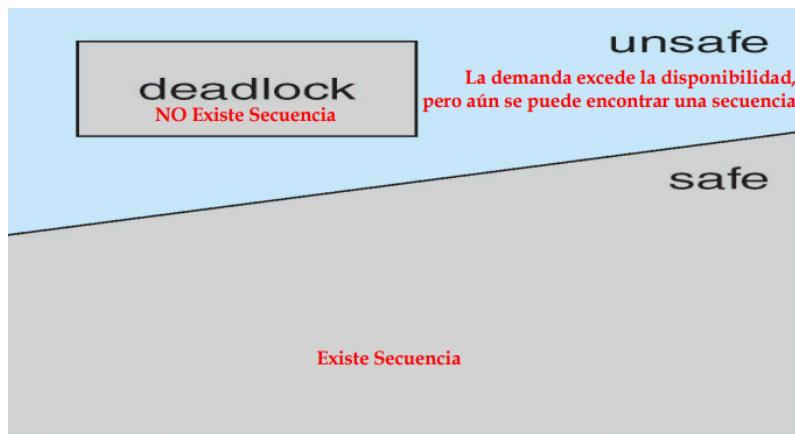
Si no se puede construir la siguiente secuencia, el estado del sistema es inseguro:

- Si los recursos necesarios de  $P_i$  no están disponibles inmediatamente, entonces  $P_i$  puede esperar hasta que todos los  $P_j$  hayan terminado.
- Cuando  $P_j$  está terminado,  $P_i$  puede obtener los recursos necesarios, ejecutar, devolver los recursos asignados y finalizar.
- Cuando  $P_i$  termina,  $P_{i+1}$  puede obtener sus recursos necesarios, y así sucesivamente.

### Estado Inseguro

En estado inseguro hay posibilidad de que haya deadlock, pero no todo estado inseguro es deadlock.

Si hay deadlock, estoy en estado inseguro.



Mientras la demanda sea menor a la cantidad de recursos disponibles, no hay posibilidad de tener deadlock.

### Algoritmos para Evitar el Deadlock

1. Si tengo una instancia única de un tipo de recurso:

Se debe utilizar un algoritmo que determine el estado seguro de un sistema.

Se utiliza un grafo de asignación de recursos.

Se debe encontrar una secuencia segura (evitar los bucles).

2. Si tengo múltiples instancias de un tipo de recurso:

Se utiliza el algoritmo del **banquero**.

Algoritmo teórico.

Busca encontrar una secuencia segura de asignación.

Los procesos declaran el número máximo de instancias de cada recurso que necesitaría. Ese número no puede exceder el total de instancias de recursos de ese tipo en el sistema. El SO decidirá en qué momento asignarlos, garantizando un estado seguro.

Se utilizan matrices y vectores con variables:

n: cantidad de procesos.

m: cantidad de tipos de recursos.

disponible: vector de m componentes, con la cantidad de recursos disponibles para cada tipo, tal que si  $\text{disponible}[j]=k$ , indica que hay k instancias del recurso  $R_j$ .

asignación: matriz de  $n \times m$  que indica cuantos recursos tiene asignados cada proceso. Asignación( $i,j$ )= $k$ , indica que hay  $k$  instancias del recurso  $R_j$  asignadas a  $P_i$ .  
 max: matriz de  $n \times m$  que indica la cantidad máxima de recursos que un proceso necesitará. Max( $i,j$ )= $k$ , indica que  $P_i$  necesitará en total  $k$  instancias del recurso  $R_j$ .  
 need: matriz de  $n \times m$  que indica la cantidad de recursos que le faltan a un proceso para completar su ejecución (need=max-asignacion). Need( $i,j$ )= $k$ , indica que  $P_i$  necesitará  $k$  instancias más de las que ya tiene, del recurso  $R_j$ .

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	1	0	0	P1	3	2	2	P1	2	2	2	R1	0	1	1
P2	6	1	2	P2	6	1	3	P2	0	0	1	R2	1	0	3
P3	2	1	1	P3	3	1	4	P3	1	0	3	R3	0	1	1
P4	0	0	2	P4	4	2	2	P4	4	2	0	Disponible			

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	1	0	0	P1	2	2	2	R1	0	1	1
P2	6	1	3	P2	6	1	2	P2	0	0	1	R2	1	0	3
P3	3	1	4	P3	2	1	1	P3	1	0	3	R3	0	1	1
P4	4	2	2	P4	0	0	2	P4	4	2	0	Vector de disponibles D			

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	1	0	0 <th>P1</th> <td>2</td> <td>2</td> <td>2</td> <th>R1</th> <td>0</td> <td>1</td> <td>1</td>	P1	2	2	2	R1	0	1	1
P2	6	1	3	P2	6	1	2	P2	0	0	1	R2	1	0	3
P3	3	1	4	P3	2	1	1	P3	1	0	3	R3	0	1	1
P4	4	2	2	P4	0	0	2	P4	4	2	0	Vector de disponibles D			

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	1	0	0 <th>P1</th> <td>2</td> <td>2</td> <td>2</td> <th>R1</th> <td>0</td> <td>1</td> <td>0</td>	P1	2	2	2	R1	0	1	0
P2	6	1	3	P2	6	1	2	P2	0	0	0	R2	1	0	3
P3	3	1	4	P3	2	1	1	P3	1	0	3	R3	0	1	0
P4	4	2	2	P4	0	0	2	P4	4	2	0	Vector de disponibles D			

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	1	0	0 <th>P1</th> <td>2</td> <td>2</td> <td>2</td> <th>R1</th> <td>0</td> <td>1</td> <td>0</td>	P1	2	2	2	R1	0	1	0
P2	6	1	3	P2	6	1	3	P2	0	0	0	R2	1	0	3
P3	3	1	4	P3	2	1	1	P3	1	0	3	R3	0	1	0
P4	4	2	2	P4	0	0	2	P4	4	2	0 <th>Vector de disponibles D</th> <td></td> <td></td> <td></td>	Vector de disponibles D			

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	1	0	0 <th>P1</th> <td>2</td> <td>2</td> <td>2</td> <th>R1</th> <td>0</td> <td>1</td> <td>0</td>	P1	2	2	2	R1	0	1	0
P2	6	1	3	P2	6	1	3	P2	0	0	0	R2	1	0	3
P3	3	1	4	P3	2	1	1	P3	1	0	3	R3	0	1	0
P4	4	2	2	P4	0	0	2	P4	4	2	0 <th>Vector de disponibles D</th> <td></td> <td></td> <td></td>	Vector de disponibles D			

- El algoritmo continúa con el fin de evaluar si existe una secuencia segura. En tal caso, podemos decir que el sistema se encuentra en un estado seguro

	R1	R2	R3		R1	R2	R3		R1	R2	R3				
P1	3	2	2	P1	0	0	0 <th>P1</th> <td>0</td> <td>0</td> <td>0</td> <th>R1</th> <td>9</td> <td>3</td> <td>6</td>	P1	0	0	0	R1	9	3	6
P2	6	1	3	P2	0	0	0	P2	0	0	0	R2	1	0	3
P3	3	1	4	P3	0	0	0	P3	0	0	0	R3	0	1	0
P4	4	2	2	P4	0	0	0	P4	0	0	0	Vector de disponibles D			

## 2. Permitir el estado de deadlock y luego recuperar:

Cada cierto tiempo se lanza un algoritmo que recorre los procesos y detecta que haya un deadlock. Si se detecta un deadlock se pasa a un mecanismo de atención al deadlock. Aparece por la ineficiencia del algoritmo del banquero, que es teóricamente el más eficiente para prevenir el deadlock.

Permitir el estado de deadlock y luego recuperar.

La técnica determina si ocurrió un deadlock, y en tal caso busca recuperarse del mismo.

Para ello vamos a contar con:

1. Algoritmo que examine si ocurrió un deadlock.

## 2. Algoritmo para recuperación del deadlock.

### Detección

Con recursos de varias instancias:

Algoritmo del Banquero.

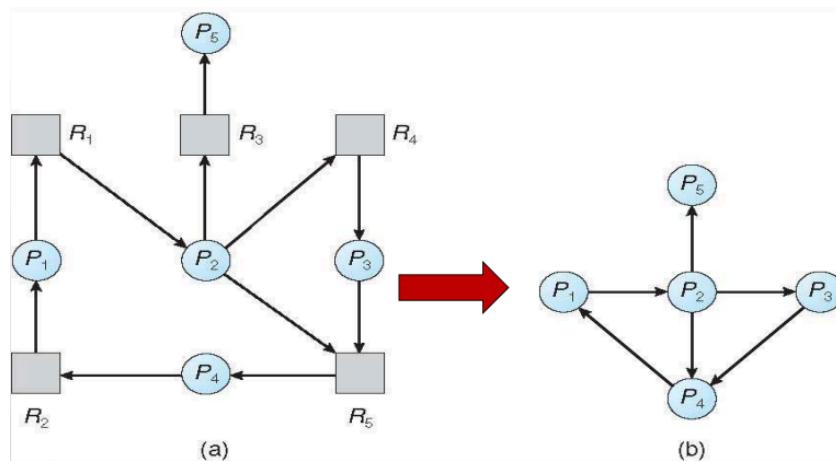
Con recursos con una sola instancia:

Se analiza el grafo de asignación. Se mantiene un grafo 'wait-for' que es un grafo de dependencias.

La cantidad de nodos es igual a la cantidad de procesos.

$P_i \rightarrow P_j$ , si  $P_i$  espera que  $P_j$  libere recurso  $R_q$ .

El algoritmo se basa en buscar periódicamente un ciclo en el grafo. Si hay un ciclo, existe un bloqueo.



### ¿Cuando utilizo el algoritmo de detección?

Cuando, y con qué frecuencia, invocar depende de:

¿Con qué frecuencia es probable que ocurra un deadlock?

¿Cuántos procesos tendrán que ser revertidos (rolledback)?

La frecuencia con la que se debe ejecutar el algoritmo de detección de interbloqueos, suele ser un parámetro del sistema.

Los motores de BD también poseen algoritmos de detección de Deadlocks. Por ejemplo SQL server lo corre cada 5 segundos.

Una posibilidad extrema es comprobar el estado cada vez que se solicita un recurso y estos no pueden ser asignados. Consumir mucha CPU.

Si el algoritmo de detección se invoca arbitrariamente, puede haber muchos ciclos en el grafo de recursos y, por lo tanto, no podríamos decir cuál de los muchos procesos bloqueados "causó" el bloqueo.

Podría ser conveniente meter mano cuando la utilización del Procesador desciende su actividad por debajo de un determinado porcentaje de utilización (recordar que un 'deadlock' eventualmente disminuye el uso de procesos en actividad).

La mayoría de los SO utilizan la técnica de no hacer nada. Lo suben o propagan a la capa superior. Allí por ejemplo la BD será quien se encargue de resolver el problema, por eso por ej SQL Server tiene su mecanismo.

### Recuperación Frente al Deadlock

Cuando un algoritmo de detección determina que existe un interbloqueo, existen varias alternativas para tratar de eliminarlo:

Caso 1: El Operador lo puede resolver manualmente. (Informar al operador del SO).

Caso 2: Esperar que el sistema se recupere automáticamente del deadlock. El Sistema rompe el deadlock.

En cualquiera de los casos, las alternativas son:

- Abortar 1 o más Procesos para romper ciclo.
- Expropiar recursos a 1 o más Procesos del ciclo.

Matando o suspendiendo los procesos se le expropien los recursos que tienen apropiados, y de esta manera se distrae el deadlock y los recursos son reasignados.

Para eliminar el deadlock matando procesos se pueden utilizar 2 métodos:

Matar todos los procesos en estado de deadlock. Simple pero a un muy alto costo.

Matar de a un proceso por vez hasta eliminar el ciclo. Considerable overhead ya que por cada proceso que vamos eliminando se debe reejecutar el Algoritmo de Detección para verificar si el deadlock efectivamente desapareció o no.

#### Terminación de Procesos

Puede no ser fácil terminar un proceso (puede estar actualizando un archivo). Previo a terminar el proceso hay que hacerlo llegar a un estado seguro (check Point) de modo que al reanudarlo quede consistente.

Se presenta un nuevo problema de política o decisión: selección de la víctima. Se debe seleccionar aquel proceso cuya terminación represente el costo mínimo para el sistema.

Hay que asegurarse de no seleccionar siempre al mismo proceso (inanición).

¿En qué orden debemos optar por abortar/terminar?

Prioridad más baja.

Menor cantidad de tiempo de CPU hasta el momento.

Mayor tiempo restante estimado para terminar.

Menor cantidad de recursos asignados hasta ahora.

Tiempo faltante para la liberación de recursos asignados.

¿Cuántos procesos tendrán que ser terminados?

¿El proceso es interactivo o batch? Puede volver atrás?

Ideal: elegir un proceso que se pueda volver a ejecutar sin problemas.

#### Conclusión

La Detección y Recuperación de interbloqueos proporciona un mayor grado potencial de concurrencia que las técnicas de Prevención o de Evitación.

Hay sobrecarga en el tiempo de ejecución de la Detección.

Hay procesos que son reiniciados o rollback lo que genera demoras en la ejecución.

La Detección y Recuperación de interbloqueos puede ser atractiva en sistemas con una baja probabilidad de interbloqueos.

En sistemas con elevada carga, se sabe que la concesión sin restricciones de peticiones de recursos puede conducir a frecuentes interbloqueos.

La solución es comprar más recursos, no queda otra opción. El software tiene un límite.

### *3. Ignorar el problema y esperar que nunca ocurra un deadlock.*

#### *. Estrategia Combinada para el Manejo de Interbloqueos:*

Ninguno de los métodos presentados es 100% adecuado para ser utilizado como estrategia exclusiva de manejo de interbloqueos en un Sistema Complejo.

La Prevención, Evitación y Detección pueden combinarse para obtener una máxima efectividad.

Se dividen los recursos del sistema en “clases de recursos”:

Para cada clase se aplica el método más adecuado de manejo de interbloqueo.

Depende de si soportan apropiaciones.

Depende de si se puede predecir el comportamiento del recurso.