Sistemas operativos avanzados

Tema 1

Introducción

Contenido

- *Definición de SO*
- *Componentes del SO (y temario)*
- *Arquitectura interna del SO*
- *Virtualización*

- *Trabajo sustitutivo*
  - *Controversias en el campo de los sistemas operativos*

Definición de Sistema Operativo (repaso)

- "Ubicuos pero no hacen un trabajo útil concreto" (¿gobierno?)
  - Crean un entorno para la ejecución de los programas
- Doble rol:
  - Gestión segura y eficiente de los recursos del computador
    - Reparto **temporal y espacial** de recursos entre programas
      - Multiplexación entre los procesos de procesadores, memoria…
  - Creando una "máquina **extendida**"
    - Abstracciones de más alto nivel de los recursos hardware
      - fichero sobre disco, socket sobre interfaz de red…
- ¿Algo más?: solo con eso, tan inútil como máquina desnuda
  - SO de propósito general debe ofrecer **interfaz a usuarios**
    - ¿Debe considerarse esa interfaz como parte del SO?
    - Suele estar implementada como aplicación externa
  - Definición precisa de SO: Asunto polémico
    - Linux vs. GNU/Linux (**primera controversia**)

Distintas interpretaciones del término SO

- ¿A qué nos referimos cuando hablamos de SO?
- Definición más estricta (la que usaremos en la asignatura):
  - Gestor de recursos que ofrece servicios a aplicaciones
    - Interfaz de usuario es otra aplicación más fuera del SO
  - SO = Núcleo (*kernel*): ejecuta en **modo sistema** (*/boot/vmlinuz*)
    - Equivalencia no aplicable a sistemas con arquitectura micronúcleo
      - Parte del SO ejecuta en modo usuario
- Definición más amplia (concepto de distribución en Linux):
  - Todo el software que **hace operativo** al sistema
    - Software de interfaz de usuario (p.e. GUI, *bash*)
    - Herramientas del sistema (p.e. montador *ld*)
    - "Demonios del sistema" (p.e. *spooler* de impresora)
    - Bibliotecas del sistema (p.e. *libc*)
    - **Más controversias**: ¿incluye el navegador web?

Componentes del sistema operativo (y temario)

- Gestión de procesos
- Gestión de memoria
- Sistema de almacenamiento
- Sistema de seguridad y protección

- Se asume conocimiento de conceptos básicos de SS.OO.
  - Procesos, *threads*, ficheros, directorios, comunicación y sincronización de procesos, *file mapping*…

- Seguridad y virtualización son transversales en la asignatura

Gestión de procesos

- Abstracción principal creada por el SO
  - Proceso → programa en ejecución
- Cada proceso tiene un conjunto de recursos asociados:
  - *threads*, mapa de memoria, ficheros abiertos, puertos usados...
- SO debe controlar:
  - Creación y destrucción de procesos
  - Asignación y liberación de recursos al proceso
- **Tema se centra en el modo de operación interno del SO:**
  - ¿Cómo SO gestiona llamadas, excepciones e interrupciones para crear la abstracción de "proceso" y "*thread*"?
- Fundamento para afrontar el proyecto práctico de la asignatura
  - *minikernel*

Otros temas vinculados con la gestión de procesos

- **Sincronización**
  - SO provee mecanismos comunicación/sincronización procesos
  - **Tema se centra en mecanismos de sincronización interna**
- **Planificación del procesador** (capítulo 4; volumen I del libro)
  - Algoritmos de planificación
  - Planificación en multiprocesadores
  - **Planificación de máquinas virtuales y contenedores**
    - ¿De qué están hechos los contenedores?
  - Planificación en Linux
- **Interbloqueos** (capítulo 7; volumen II del libro)
  - Algoritmos teóricos de tratamiento de interbloqueos
  - Tratamiento de los interbloqueos por parte del SO
  - Pautas para desarrollar programas libres de interbloqueos

Gestión de memoria (capítulo 5; volumen I del libro)

- SO crea espacio lógico propio (mapa) a cada proceso
  - Creado a partir del ejecutable
    - **Estudiaremos en detalle todo el ciclo de vida de un programa**
      - Compilación, montaje, bibliotecas dinámicas, PIC, PIE…
  - Mapa del proceso incluye todas las regiones requeridas
    - Código, datos, pilas, DLL, ficheros proyectados, etc.
- SO gestiona la memoria del sistema:
  - Adaptándose a esquema de la MMU: segmentación, paginación
  - Implementando la técnica de memoria virtual
- Seguridad en la gestión de memoria. **Estudiaremos**:
  - Distintos tipos de ataques: *buffer overflow*, ROP, Meltdown…
  - Diversas soluciones del SO: ASLR, KASLR, KPTI…
- Virtualización de la memoria:
  - ¿Cómo hacer creer a SO anfitrión que tiene su propia memoria?

Sistema de almacenamiento

- Fichero: abstracción de espacio de almacenamiento
- Espacio jerárquico de nombres usando directorios
- Tendencia: dar soporte a distintos tipos de sistemas de ficheros
  - Concepto de sistema de ficheros virtual: interfaz común
- Cada tipo de sistema de ficheros específico usa estrategias para:
  - Organización del espacio ocupado por los ficheros
  - Gestión del espacio libre
  - Técnicas de prevención ante caídas (p.e. *journaling*)

Sistema de seguridad y protección

- Seguridad es transversal a todos los componentes del SO:
  - Aparece en todos los temas anteriores
- **Este tema se centra en la protección**:
  - ¿Qué operaciones puede hacer un usuario con cada recurso?
  - ¿Cómo almacenar información sobre permisos?
    - Asociada al usuario: Capacidades
    - Asociada al recurso: Listas de control de acceso (ACL)
  - **Estudiaremos distintos esquemas de control de acceso**:
    - DAC, MAC, RBAC
    - Y su implementación en Linux (SE-Linux, AppArmor…)

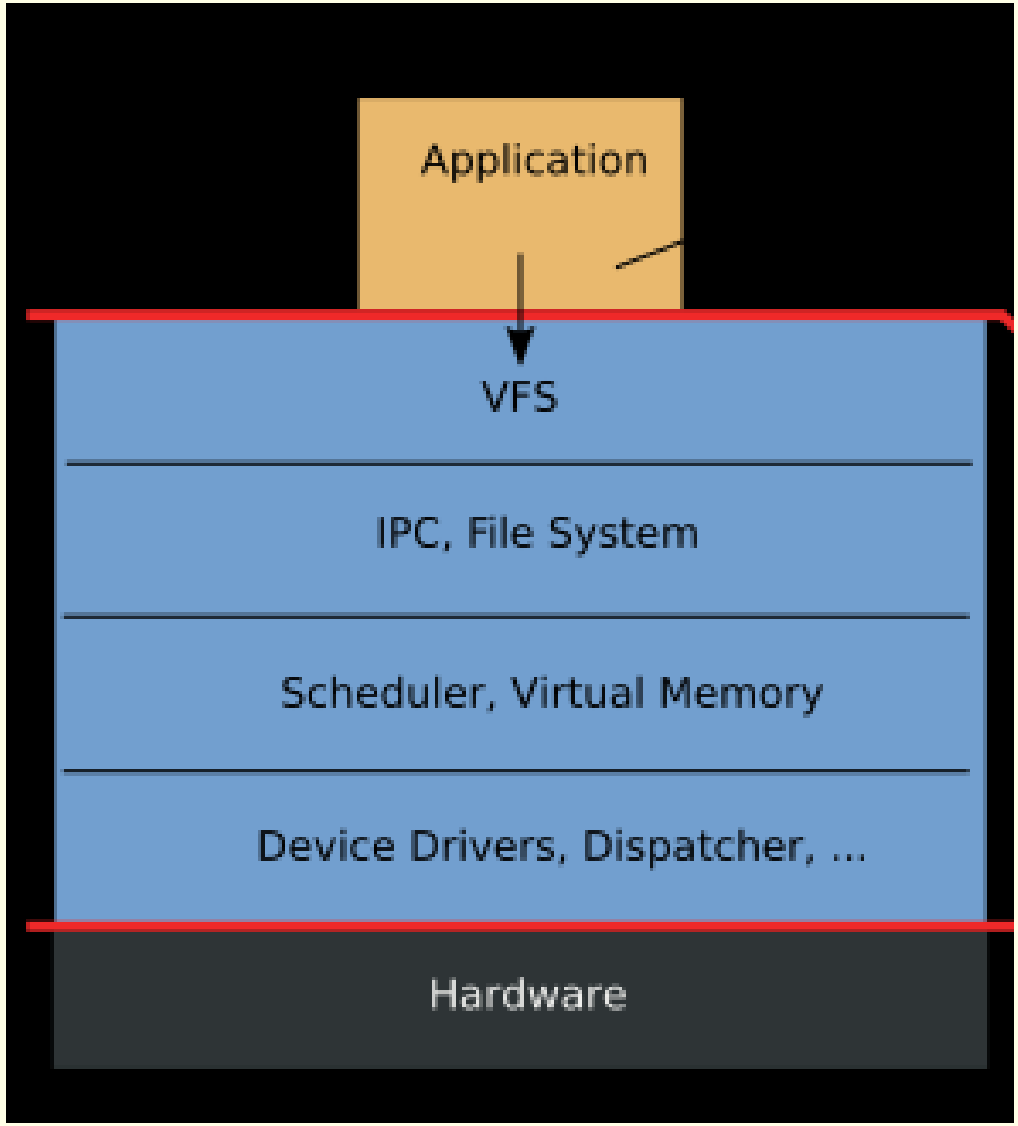
Arquitectura interna del sistema operativo

- Existen distintas alternativas:
  - Sistemas monolíticos
  - Sistemas basados en micronúcleos
  - Sistemas híbridos
  - Exonúcleos

Sistemas operativos monolíticos

- SO = núcleo (*kernel*) → programa que ejecuta en modo sistema
  - Todo el código del SO enlazado en un único programa que
  - Ejecuta en un mismo espacio de direcciones
- Aplicaciones y programas de sistema ejecutan en modo usuario
- **Ventaja**: Eficiencia
  - Ejecución de servicio de SO:
    - solo requiere cambio de usuario a sistema y viceversa
- **Desventaja**: Difícil depuración y extensibilidad
  - Error en cualquier parte del SO afecta al resto
- Es la arquitectura más habitual
  - Característica de la familia UNIX (Linux)

Sistema monolítico (wikipedia)



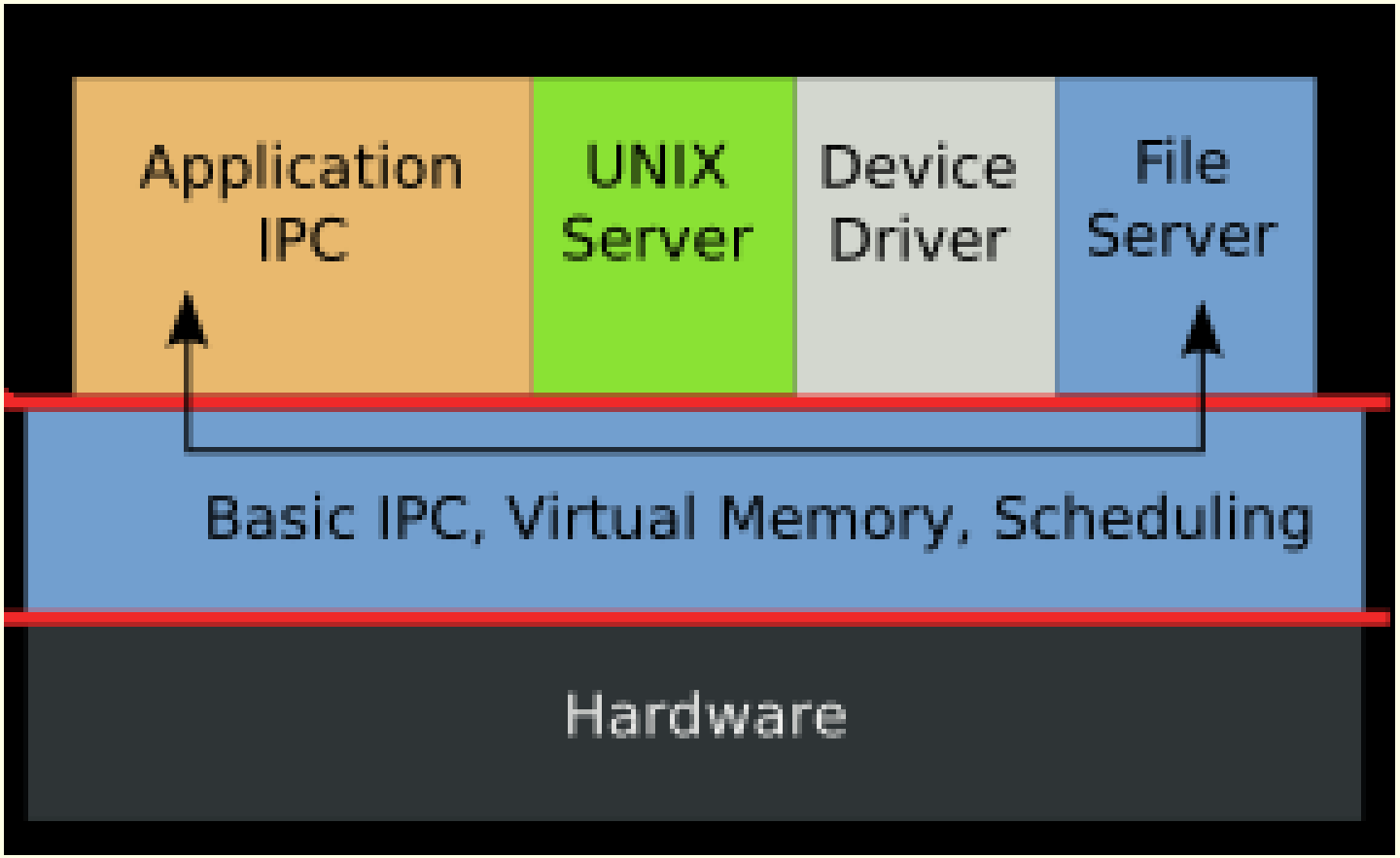
Sistemas con módulos cargables

- Mayoría de sistemas monolíticos actuales no están “cerrados”
  - Permiten cargar módulos en tiempo de ejecución
- Ejecutable del SO contiene funcionalidad básica
- Restante en módulos (manejadores, s. ficheros, protocolos, etc.)
- Módulo similar a biblioteca dinámica pero para el núcleo
  - Se incorpora a espacio de SO y comparte símbolos
  - Se mantienen los mismos problemas de fiabilidad
- **Ventajas**:
  - Facilita extensibilidad del SO
  - Permite adaptar núcleo a características de la plataforma
    - P.ej. Crear núcleo mínimo para sistema empotrado
  - Posibilita técnicas como *hot-plugging*

Micronúcleos

- “Lo perfecto no es que no falte nada sino que no sobre nada”
- Núcleo queda reducido a funcionalidad mínima
  - Gestión de procesos y de memoria de bajo nivel + IPC
  - Micronúcleo proporciona nº muy reducido de llamadas
  - Gestión básica de hardware
    - P.e. interrupción → mensaje a proceso que la gestiona
- Funcionalidad del SO en servidores en modo usuario
  - Sistema de ficheros, gestor de memoria, manejadores...
  - “Llamada al sistema” de aplicación: mensaje a servidor
    - Y entre servidores si es necesario
- Aplicación no sabe si ejecuta sobre micronúcleo o monolítico

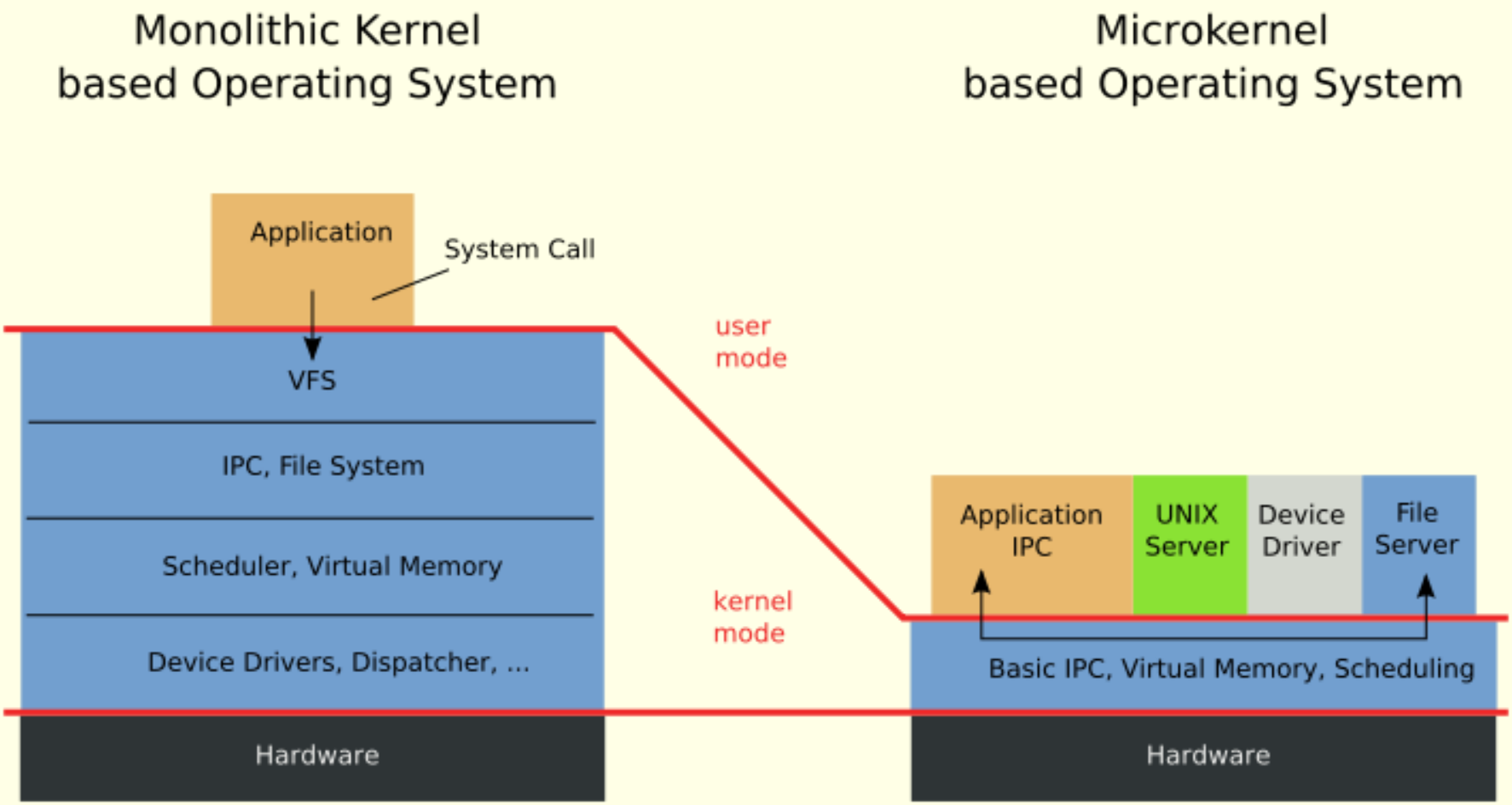
Sistema basado en micronúcleo (wikipedia)



Micronúcleos vs. monolíticos

- Ventajas:
  - Extensibilidad, fiabilidad y facilidad de depuración
  - Error en parte del SO solo afecta al servidor involucrado
  - Posible convivencia de varios SS.OO.
  - Facilita programación funcionalidad del S.O.
    - Programación convencional vs. Programación módulos Linux
- Desventajas:
  - Eficiencia.
  - Coste de llamada:
    - Sobrecarga mensajes y cambios de proceso
- Asunto polémico (**segunda controversia**):
  - Monolíticos son los eternos aspirantes

Monolítico versus micronúcleo (wikipedia)



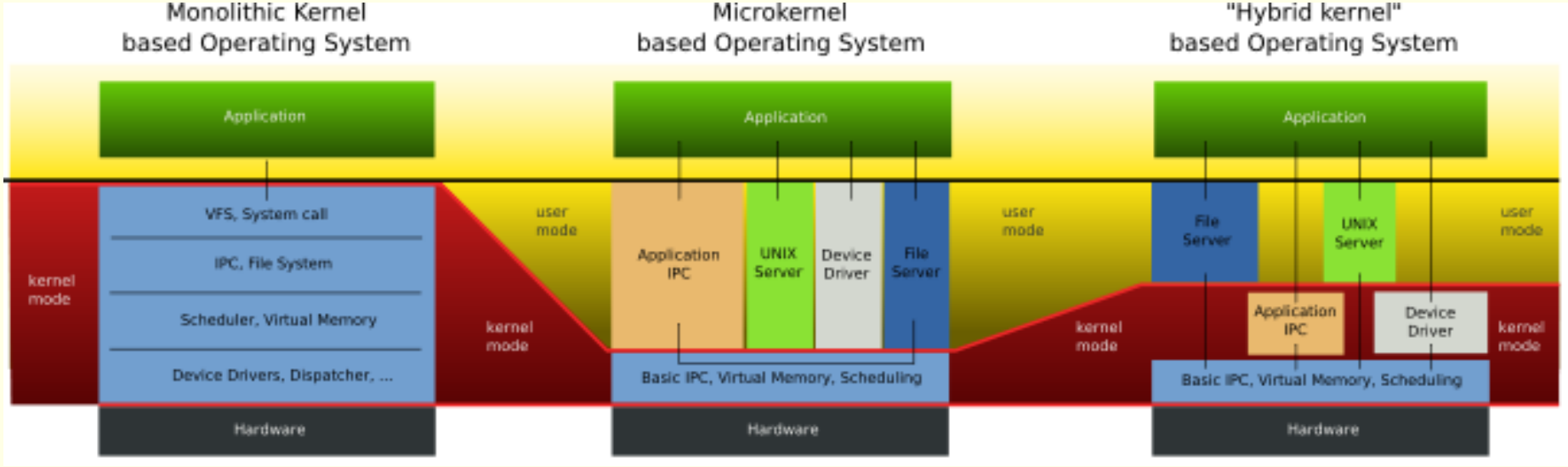
Evolución de los micronúcleos

- Primeros sistemas: mono-servidor
  - Único servidor en modo usuario proporciona todos servicios
  - Menos sobrecarga por mensajes y cambios de contexto
  - Pero pierde muchas de las ventajas de micro-núcleos
    - ¿Es realmente un micronúcleo? ¿Flexibilidad? ¿Fiabilidad?
- Eficiencia mejora en 2ª generación (L4) frente a 1ª (Mach):
  - Mach (1987): IPC ≈ 100 μs; L4 (1995): IPC ≈ 1 μs
- Tamaño del núcleo ha ido disminuyendo:
  - Mach ≈ 100 llamadas; L4 ≈ 10 llamadas
  - Mach ≈ 100.000 líneas de código; L4 ≈ 10.000
- 3ª generación: núcleos verificados formalmente (SeL4)
- Fracasos: GNU Hurd; ¿Renacimiento?: Google Fuchsia
  - Proyecto secreto de Google descubierto en GitHub (8/2016)

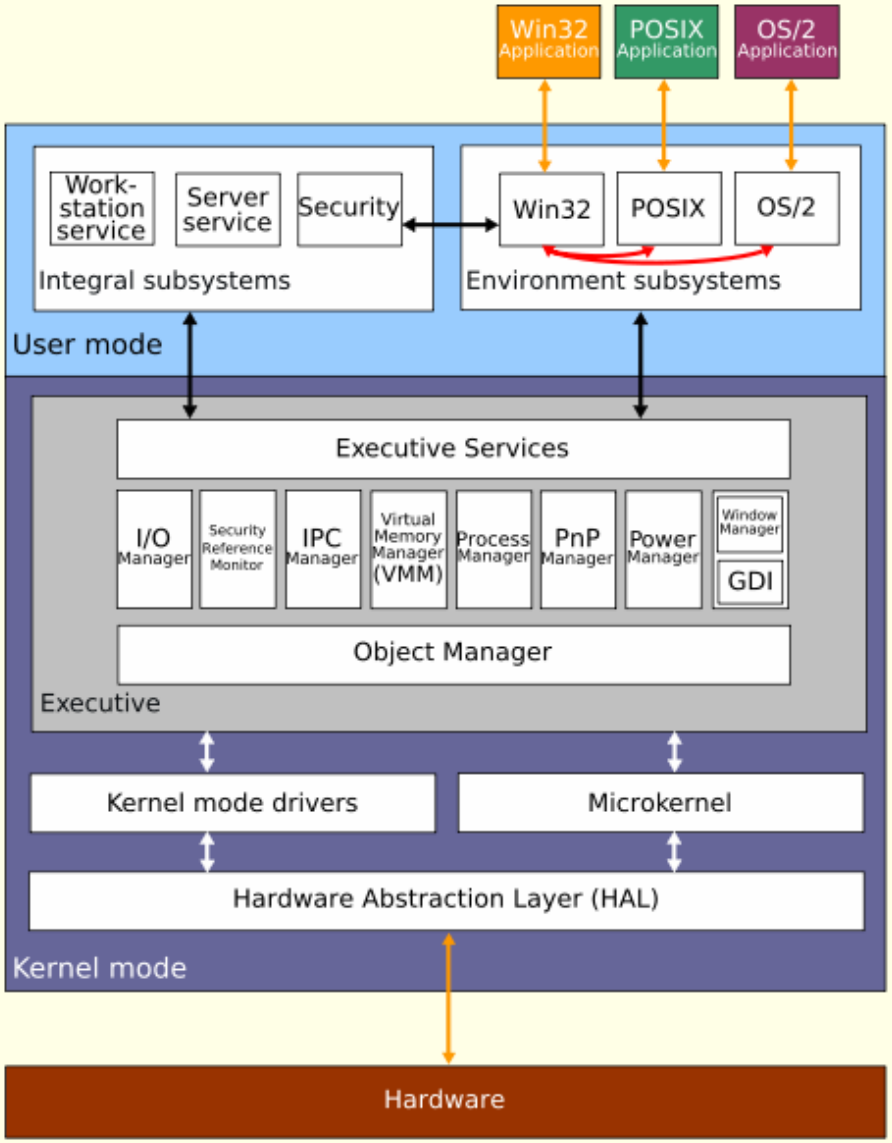
Sistemas híbridos

- Algunos s. micronúcleo permiten servidores en modo sistema
  - Más eficiente pero rompe la filosofía micronúcleo
  - Servidores son programas independientes pero
    - Ejecutan en mismo espacio de direcciones del micronúcleo
    - Y no usan IPCs para comunicarse
- Categoría discutida
  - Puristas consideran que son monolíticos
- Ejemplo: Mac OS X
  - Interfaz UNIX
  - Micronúcleo Mach con servidores en modo sistema
- ¿Y Windows?
  - Difícil clasificación

Monolítico | Micronúcleo | Híbrido (wikipedia)



Estructura de Windows 2000 (wikipedia)



Exonúcleos

- Motivación:
  - No todas las aplicaciones necesitan ver mismas abstracciones
    - Gestor base de datos mejor maneja bloques de disco que ficheros
      - Para poder controlar ubicación de datos en disco o el uso de caché
  - Uso de abstracciones inadecuadas es ineficiente
- Propuesta: *Exokernel* (MIT, 1995)
  - Núcleo provee abstracciones básicas (página, bloque...)
  - Funcionalidades de tipo SO en bibliotecas en modo usuario
  - Cada aplicación se enlaza con las bibliotecas que requiera
    - Gestor de base de datos no requiere de sistema de ficheros
  - **Aplicación sabe que está ejecutando sobre un exonúcleo**
    - A diferencia de lo que ocurre con micronúcleos
- Aplicación del principio "*end-to-end*"
- Objeto de investigación: no usada en sistemas relevantes

Virtualización (sección 4.16 de volumen I del libro)

- Una de las tecnologías más importantes de este siglo
  - Motor de *cloud*, nuevos modelos de despliegue de aplicaciones…
- Pero SO ya crea una máquina virtual…
  - Proceso *virtualiza* procesador
  - Mapa de memoria de proceso *virtualiza* memoria del sistema
  - Fichero *virtualiza* disco
  - Socket *virtualiza* interfaz de red
- … sin embargo, es una máquina virtual *extendida*
  - Con un nivel de abstracción mayor que la máquina real
- ¿Virtualización ajena al SO? ¿debemos estudiarla en este curso?
  - Aunque trabaja en otro nivel de abstracción
  - **Hipervisor** usa técnicas similares a un SO
    - KVM convierte a Linux en un hipervisor
  - Virtualización soportada por SO (contenedor) frente a tradicional
  - **Se trata de forma transversal durante el curso**

Virtualización “tradicional”: hitos

- Pionero CP-40 (IBM, 1967): ¿t. compartido y SO monousuario?
  - CP crea 1 MV/usuario: SO monousuario CMS sobre cada MV
- Explosión a finales del siglo XX (VMware, 1998)
- Popek & Goldberg (P&G 1974): 3 condiciones
  1. **Equivalence / Fidelity**:
  2. **Resource control / Safety**:
  3. **Efficiency / Performance**:
- Interpretación de las tres condiciones:
  1. SSOO y aplicaciones se comportan como en una máquina real
     - No tienen que modificarse para ejecutar sobre el hardware virtual
  2. Sistema de virtualización tiene control completo de recursos
     - Ni SO ni aplicaciones pueden saltárselo
  3. Gran mayoría de instrucciones ejecuta directamente UCP real
     - Asegura eficiencia y descarta solución basada en emulación
  - MV: duplicado **eficiente** y aislado de una máquina real

Ventajas de la virtualización

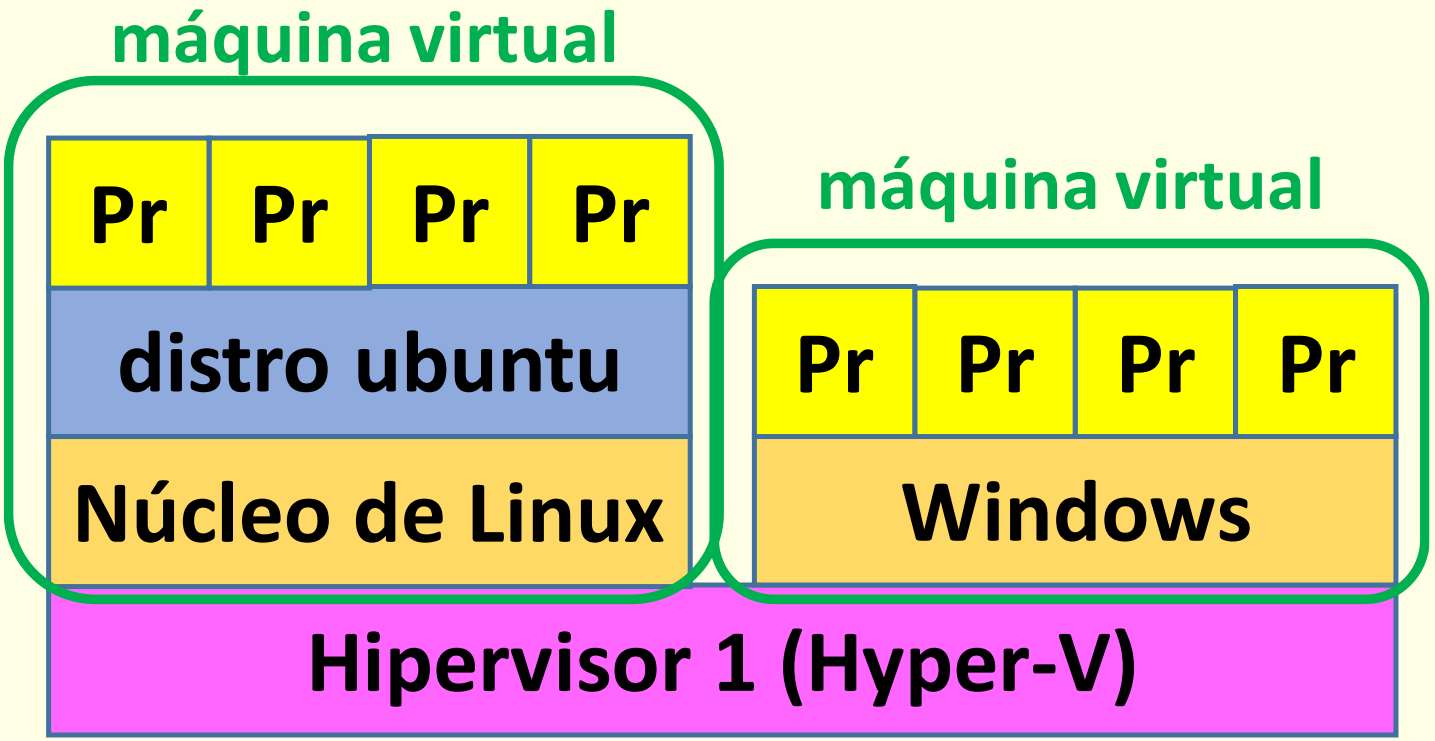
- Coexistencia de distintos SSOO
- Aislamiento de aplicaciones (*sandboxing*)
- Ejecución de *legacy systems*
- Creación de entornos de desarrollo y pruebas
- Mejora en disponibilidad
- Mejor aprovechamiento del hardware
- Ahorro energético
- Consolidación de sistemas
- Migración en vivo
- Facilita administración de sistemas
- Reducción de costes de inversión y mantenimiento
- Desventajas: rendimiento, curva de aprendizaje…

Tipos de virtualización

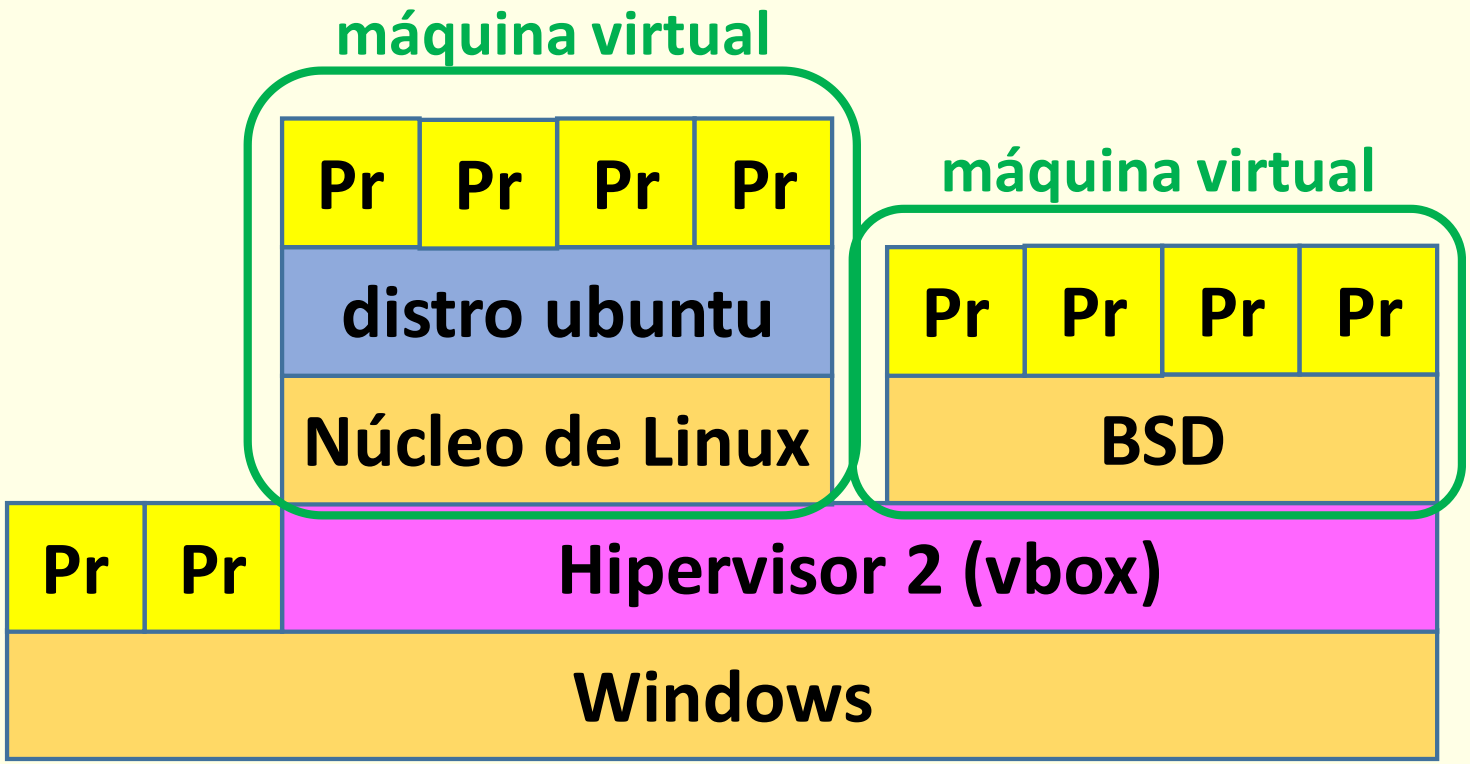
- Tipo 1 (nativa o *bare-metal*):
  - Hipervisor ejecuta sobre máquina desnuda
  - Sobre cada MV ejecuta **SO alojado**
  - VMware ESX/ESXi, Citrix XenServer, MS-Hyper-V...

- Tipo 2 (alojada):
  - Hipervisor ejecuta sobre **SO anfitrión**
  - Sobre cada MV ejecuta **SO alojado** (posiblemente distinto)
  - VMware Workstation, VMware Player, VirtualBox…
  - Hipervisor invierte la abstracción de SO anfitrión
    - P.ej. Interrupción real → Señal UNIX → Interrupción MV

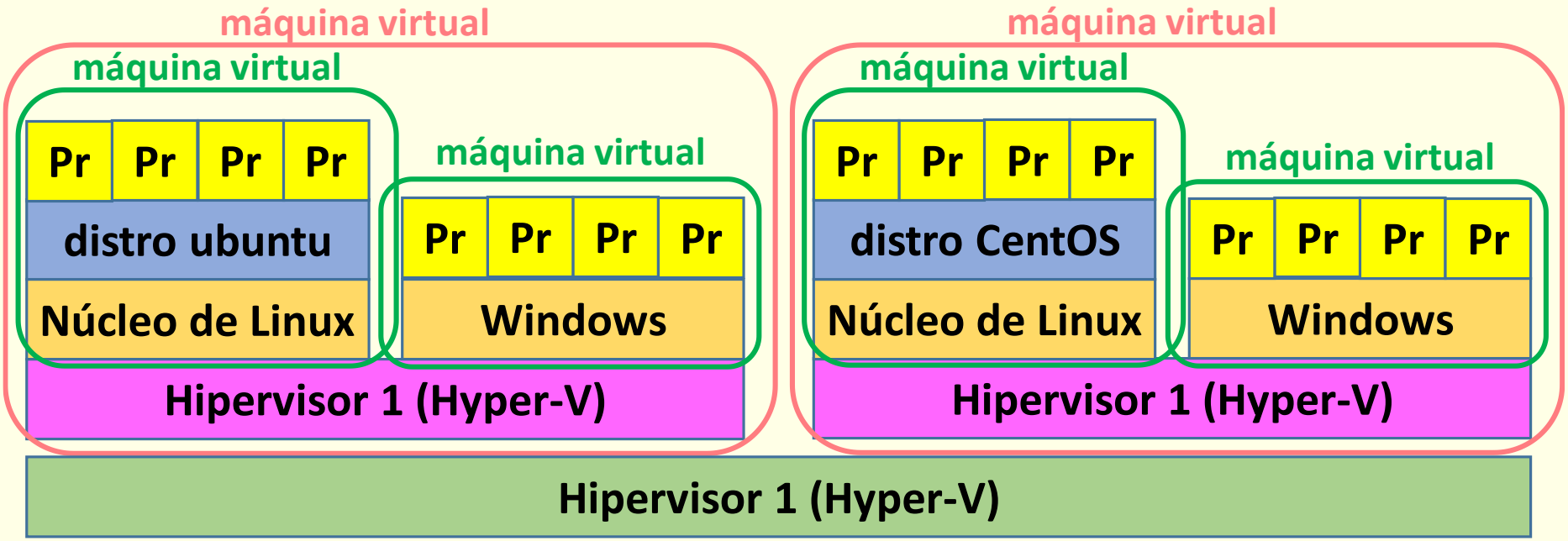
Hipervisor de tipo I (nativa)



Hipervisor de tipo II (alojada)



Anidamiento de virtualización



Implementación de MV: *Trap&emulate*

- Para mantener control y aislamiento entre MVs, SO alojado:
  - No puede ejecutar instr. privilegiadas (p.e. inhibir interrup.)
  - No debe acceder directamente a ciertas direcciones
    - P.e. modificar una entrada de una tabla de páginas
- 3 modos ejecución: usuario, privilegiado virtual, privilegiado real
- Solución: SO alojado ejecuta con UCP en modo no privilegiado
  - Si inst. privilegiada o acceso a dirección no permitida → ***trap***
    - VMM toma control y **emula** comportamiento de instrucción
  - Si procesador con solo dos modos:
    - SO alojado convive en modo usuario con aplicaciones
    - pero aislado mediante protección de memoria
  - Si varios modos, SO alojado usa modo intermedio
    - Nivel(Hipervisor) > Nivel(SO alojado) > Nivel(Aplicaciones)
    - Pentium: VMM anillo 0, SO alojado anillo 1, Apps anillo 3
- Sobrecarga por procesamiento de *traps* (excepciones)

Virtualización del x86: ¿misión imposible?

- Popek & Goldberg (1974): distinguen entre tipos de instrucciones
  - **Privileged**: causan *trap* si ejecutadas modo no privilegiado
  - **Control sensitive**: cambian configuración de recursos sistema
  - **Behavior sensitive**: dependen de conf. de recursos del sistema
- Popek & Goldberg: **Teorema 1**
  - Se puede construir una VMM *eficaz* si: *sensitive* ⊆ *privileged*
- Pentium no lo cumple
  - IA-32 ISA tiene 18 instrucciones *sensitive* pero *unprivileged*
  - P.e. POPF se comporta diferente en modo usuario y privilegiado
    - No *trap*: solo diferente comportamiento
  - ¿Es imposible virtualizar Pentium?
- Hacemos trampas: **Paravirtualización** (Xen): ~~1ª condición P&G~~
  - Se modifica SO alojado (necesitamos fuente; ¿Windows?)
  - Cambia i. *sensitive* pero *unprivileged* por llamada a hipervisor
  - También puede agilizar operaciones reduciendo *traps*

Traducción binaria

- *trap&translate&emulate* (VMware)
- Traductor dinámico sustituye en tiempo de ejecución
  - instrucciones *sensitive* pero *unprivileged* por *traps*
- ¿Cumple 3ª condición P&G (**eficiencia**)?
  - Traducción muy sofisticada y eficiente
  - Similar a compiladores JIT
  - Solo se traduce el código del SO alojado
- Sobrecarga por procesamiento de *traps*
  - Traducción inteligente puede evitar algunos
- Resuelve el problema de virtualizar x86
  - Sin requerir modificar el SO alojado
- UCPs actuales soporte para virtualización (AMD-V, Intel-VT)
  - Instrucciones *sensitive* son *privileged*→ ***trap***, nuevos modos de ejecución del procesador, redirección de interrupciones, tablas de páginas anidadas, IOMMU…

Almacenamiento de la MV (tipo 2)

¿cómo SO alojado lo interpreta?
¿cómo conciliar usuarios anfitrión y SSOO alojados?

anfitrión

directorio compartido con MVs

MV1

disco MV1

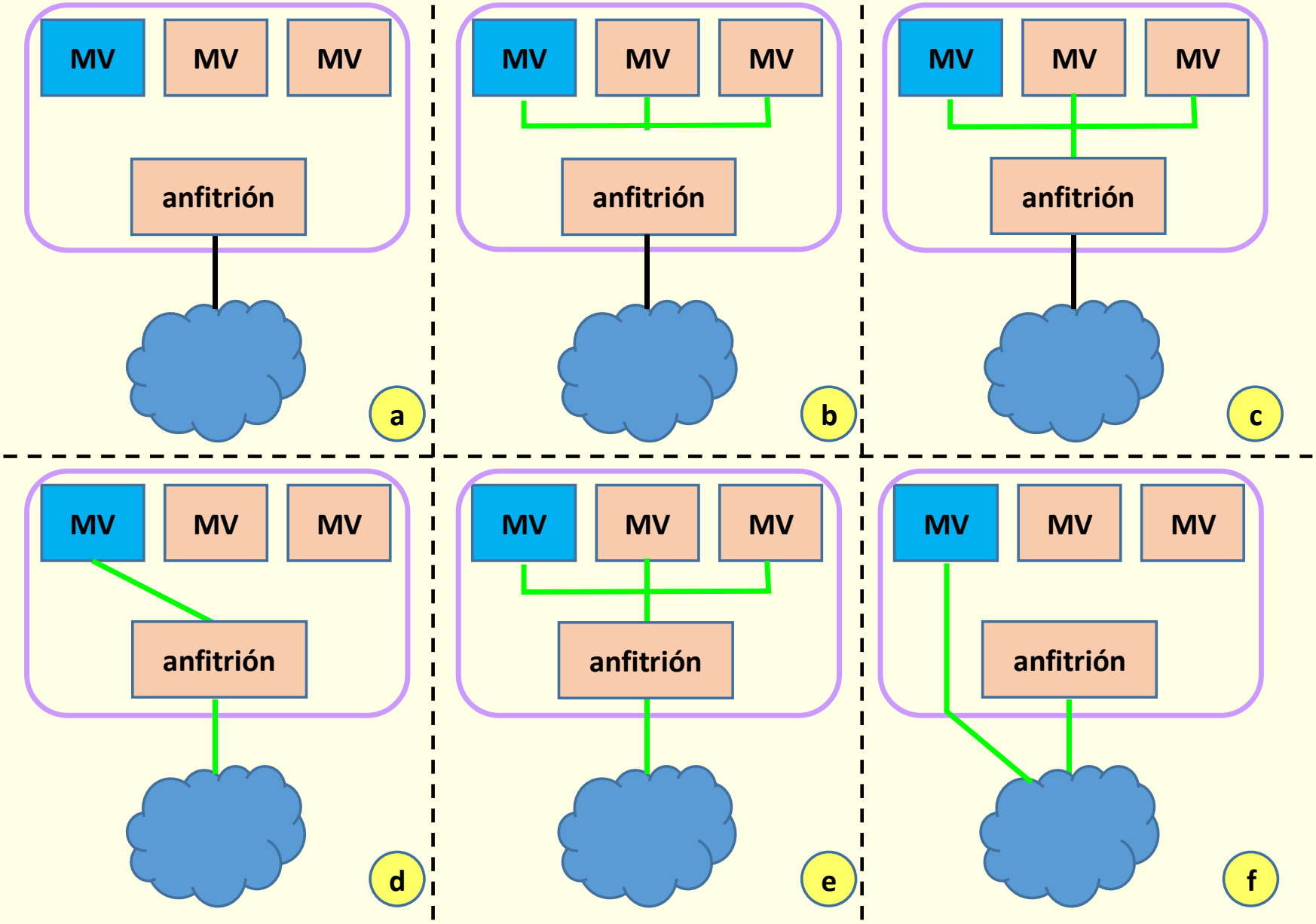
MV2

disco MV2

disco MV

- fichero (*loop device*)
- partición física
- volumen lógico

Conectividad de máquinas virtuales (tipo 2)



Limitaciones de virtualización tradicional

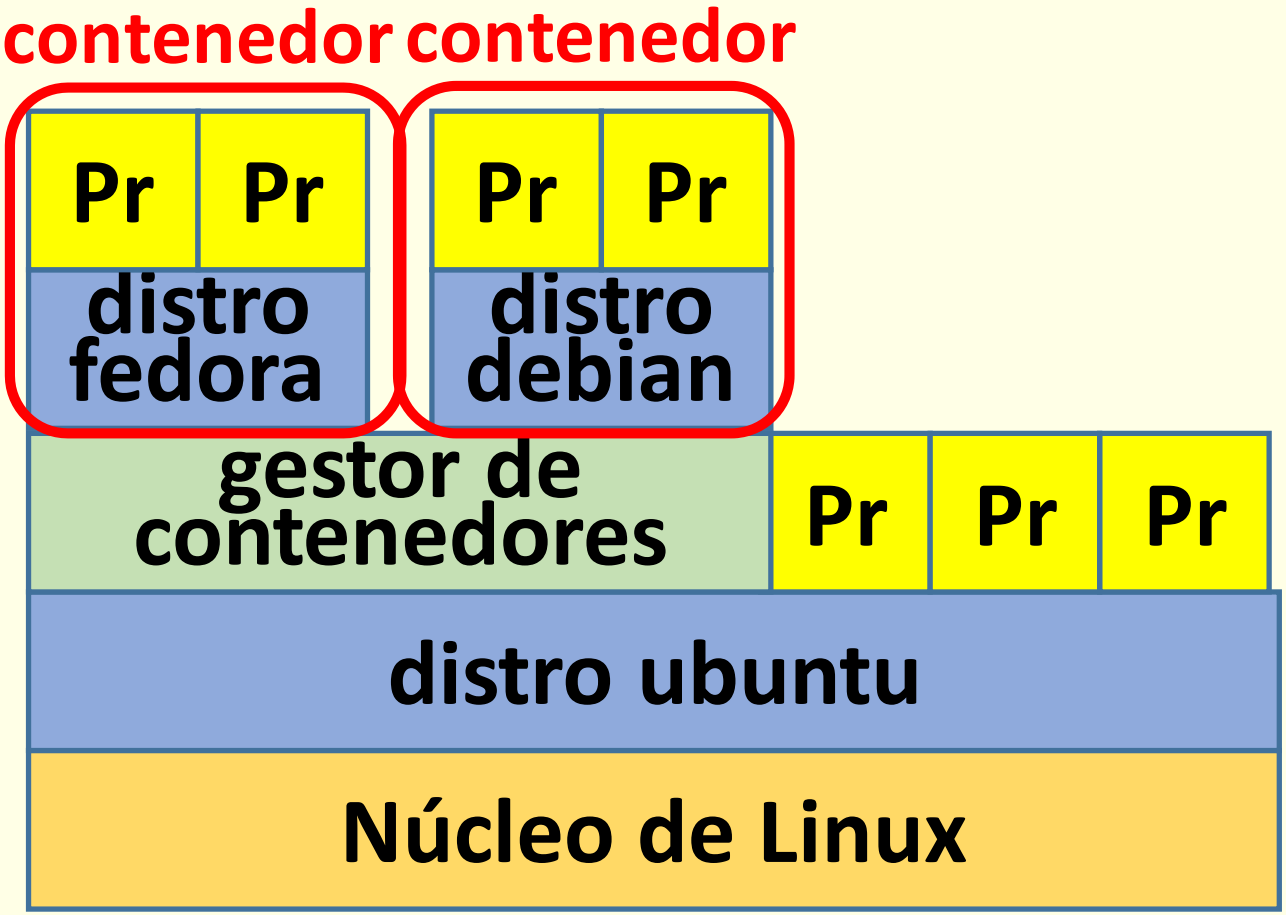
- Máquinas virtuales son “pesadas”:
  - Requieren múltiples recursos por cada instancia
    - Una copia de SO alojado/instancia aunque sea el mismo
  - Despliegue lento: requiere arranque de SO
  - Sobrecarga requerida por virtualización
    - Fallos de página y cambios de contexto adicionales…

- Solución no escalable
  - No adecuada si se requiere nº elevado de MVs
  - que se puedan activar y desactivar con agilidad

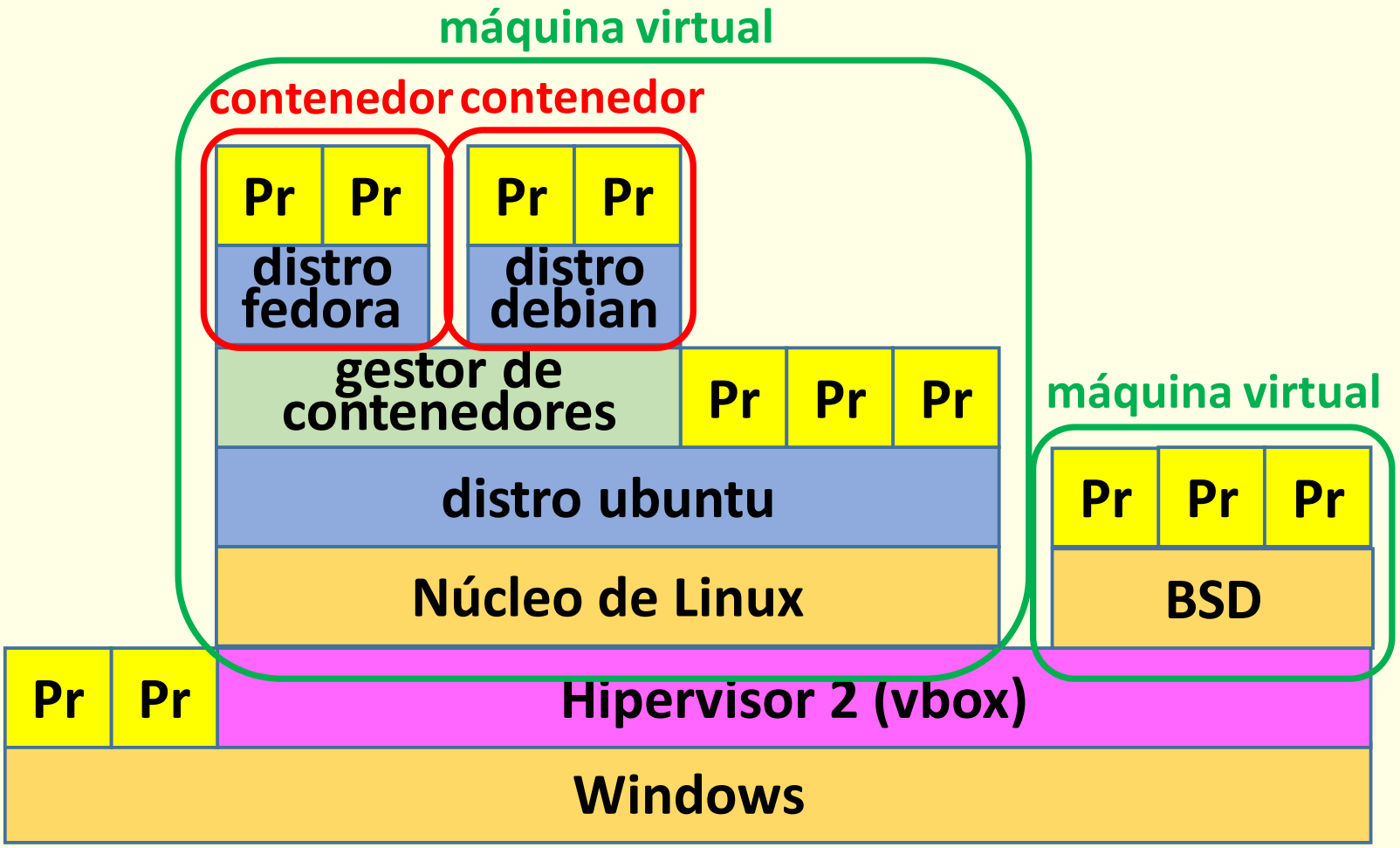
Virtualización soportada por el SO: contenedores

- El propio SO da soporte a la virtualización: **contenedor**
  - Contexto de ejecución independiente y estanco donde ejecuta un **grupo de procesos** que creen disponer de su propio equipo
  - Cada contenedor recibe conjunto de recursos independiente
  - Cada contenedor ve un espacio de recursos independiente
- Gestión de contenedores más eficiente que la de MV:
  - Pueden crearse muchas más instancias de forma mucho más ágil
    - Crear MV → arranque SO; crear contenedor → crear proceso
  - Solo 1 copia SO: menor sobrecarga
    - pero no convivencia de SSOO
  - Contenedor independiente de distribución pero no del núcleo
- Tres hitos en la historia de los contenedores (criterio subjetivo)
  - FreeBSD Jails (2000), Zonas de Solaris (2005), Docker (2013)

Uso de contenedores



Uso combinado de hipervisor y contenedores



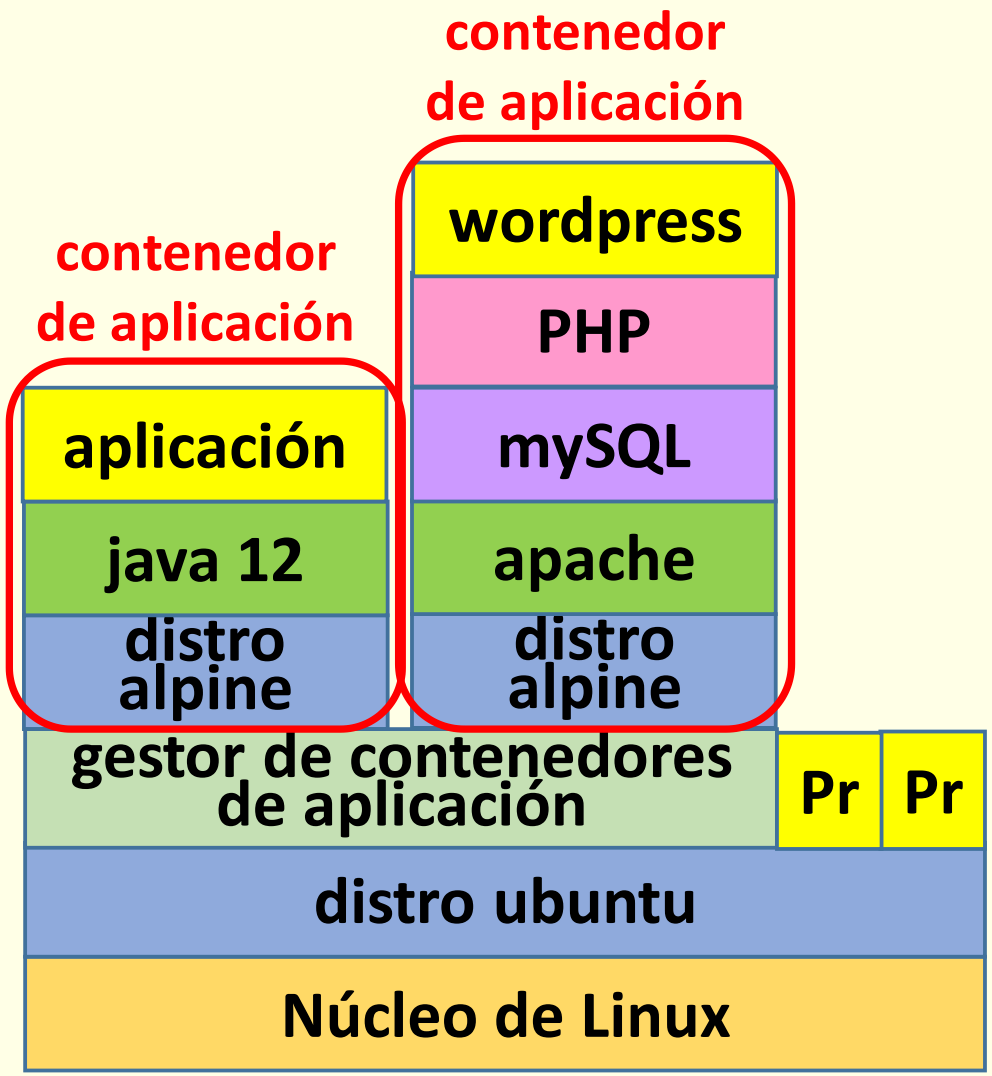
Virtualización basada en contenedores: requisitos

- SO asigna y controla uso de recursos por un contenedor
  - procesador, memoria, discos, red…
  - Cuotas de reparto y límites de uso máximo
- SO debe llevar contabilidad de uso de recursos por contenedor
- SO debe permitir realizar operaciones sobre un contenedor
  - P.e. detener temporalmente ejecución de todos sus procesos
- SO debe ofrecer a un contenedor su propia visión de:
  - Nombre de la máquina, PIDs, usuarios existentes, montajes…
  - La red: nombre de interfaces de red, puertos...
    - Contenedor: Funcionalidad de un nodo de red
  - El sistema de almacenamiento
    - Inicialmente contiene la **imagen** asociada al contenedor
    - Además, compartir información entre contenedores y con anfitrión
- Virtualización basada en contenedores: soporte Linux
  - *control groups* (*cgroups*) y *namespaces*

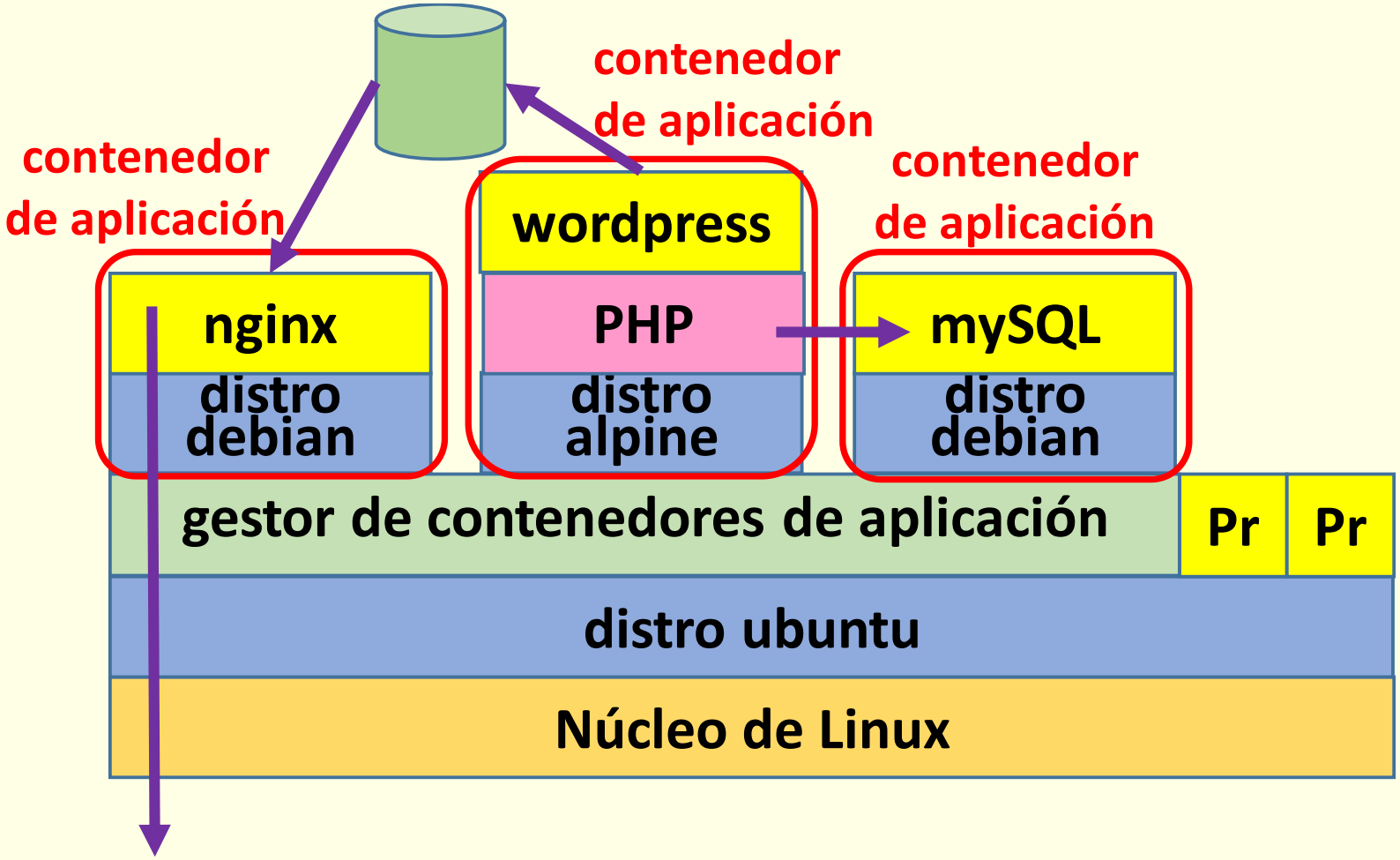
Contenedores de aplicación (p.ej. *docker*)

- Mecanismos presentes en SO hace tiempo pero apenas usados
- ¿Por qué Docker revolución en desarrollo/despliegue apps/serv.?
- ¿Qué tipos de problemas se dan en este campo y cómo los resuelve?
- Instalación de software compleja y peligrosa (*dependency hell*)
  - Gestores de paquetes resuelven el problema pero solo en parte
- Modo de desarrollo y despliegue tradicional: tensión *Dev* vs *Ops*
  - Desarrolladores programan app en equipos de desarrollo
  - Equipo operaciones despliega app en plataforma en producción
- Docker facilita *CI/CD* y confluencia de roles: *DevOps*
  - *Dev* crean imagen con app y sus dependencias (autocontenida)
  - *Ops* descargan imagen de repositorio
    - Y activa su ejecución en producción en contexto de un contenedor
- Imágenes organizadas en capas: facilita reutilización
- Buenas prácticas: 1 proc/contenedor y composición de contenedores
- S. distribuido: Kubernetes y arquitectura basada en microservicios

Contenedores de aplicación: capas



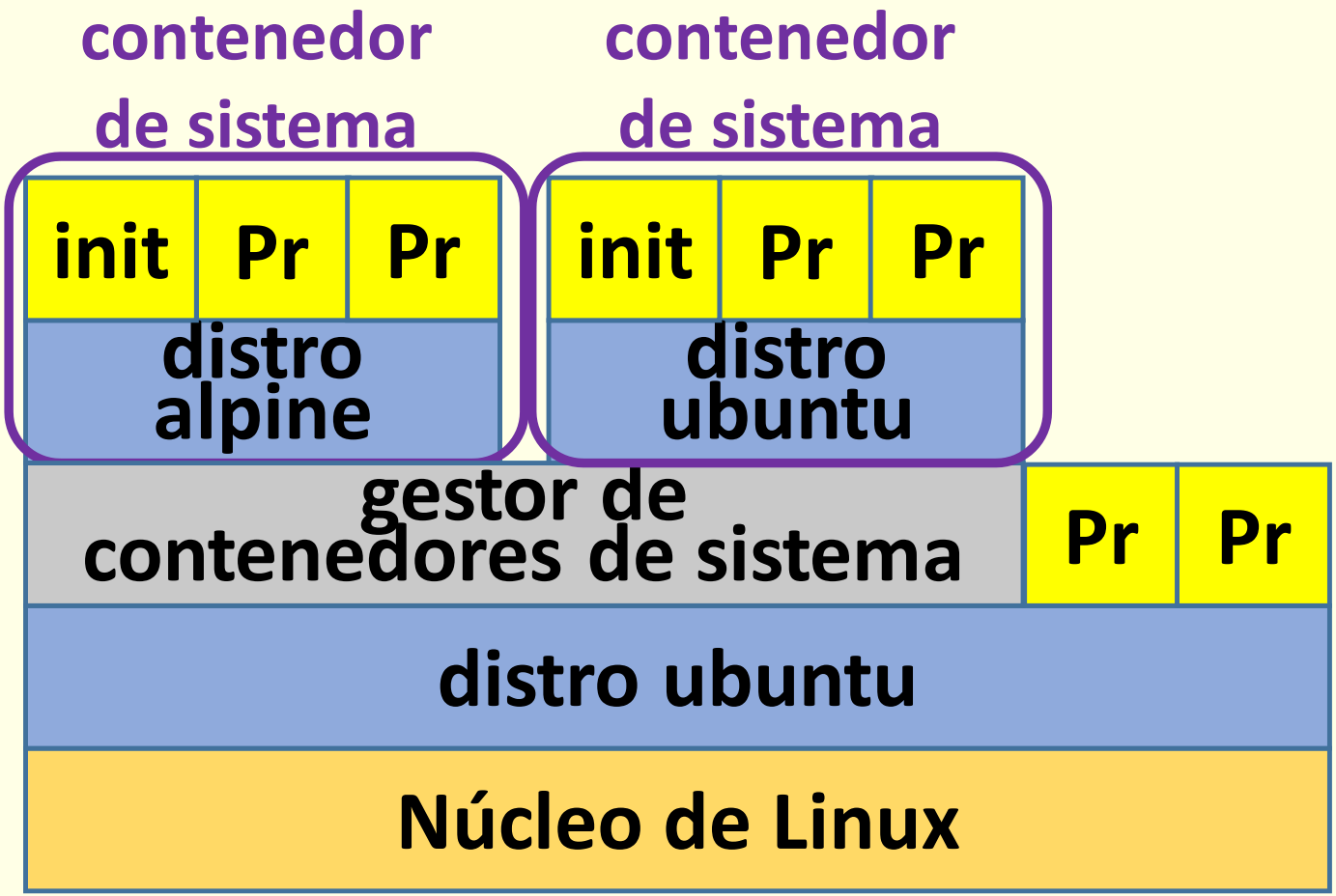
Contenedores de aplicación: composición



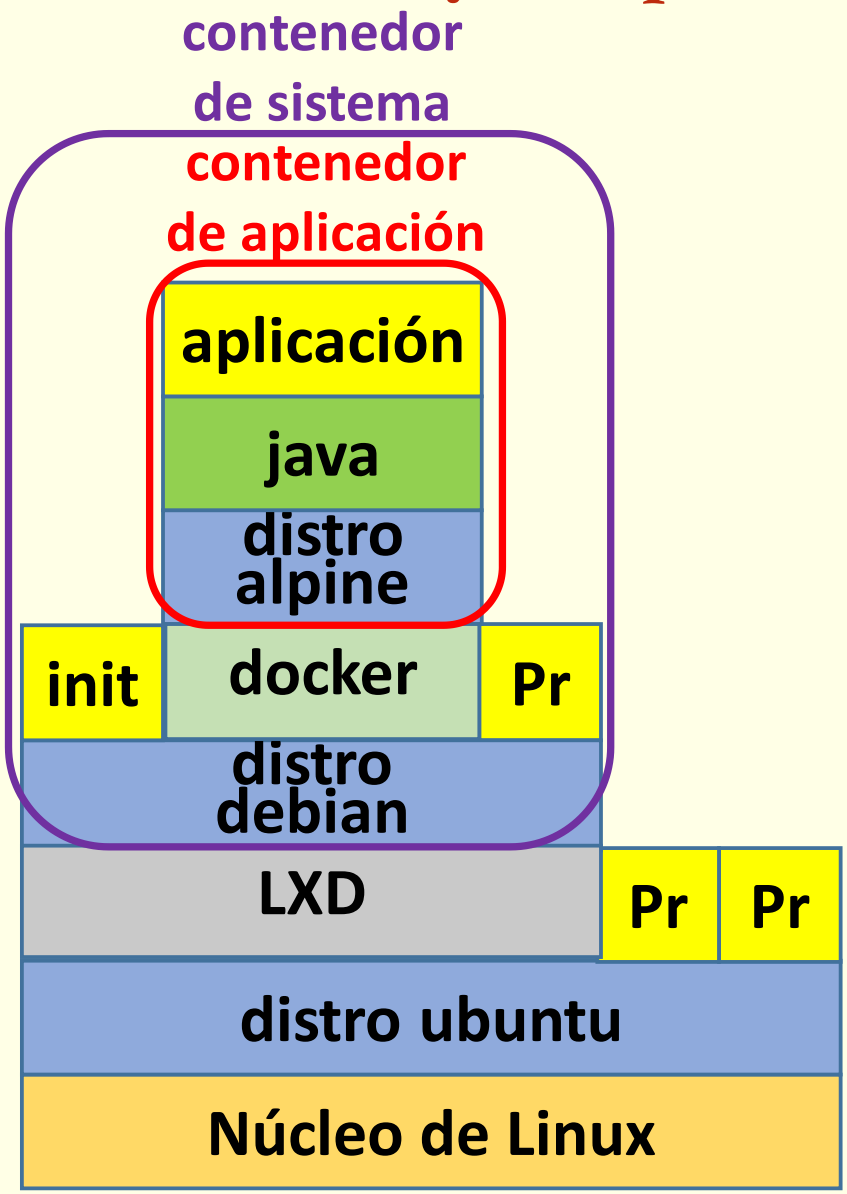
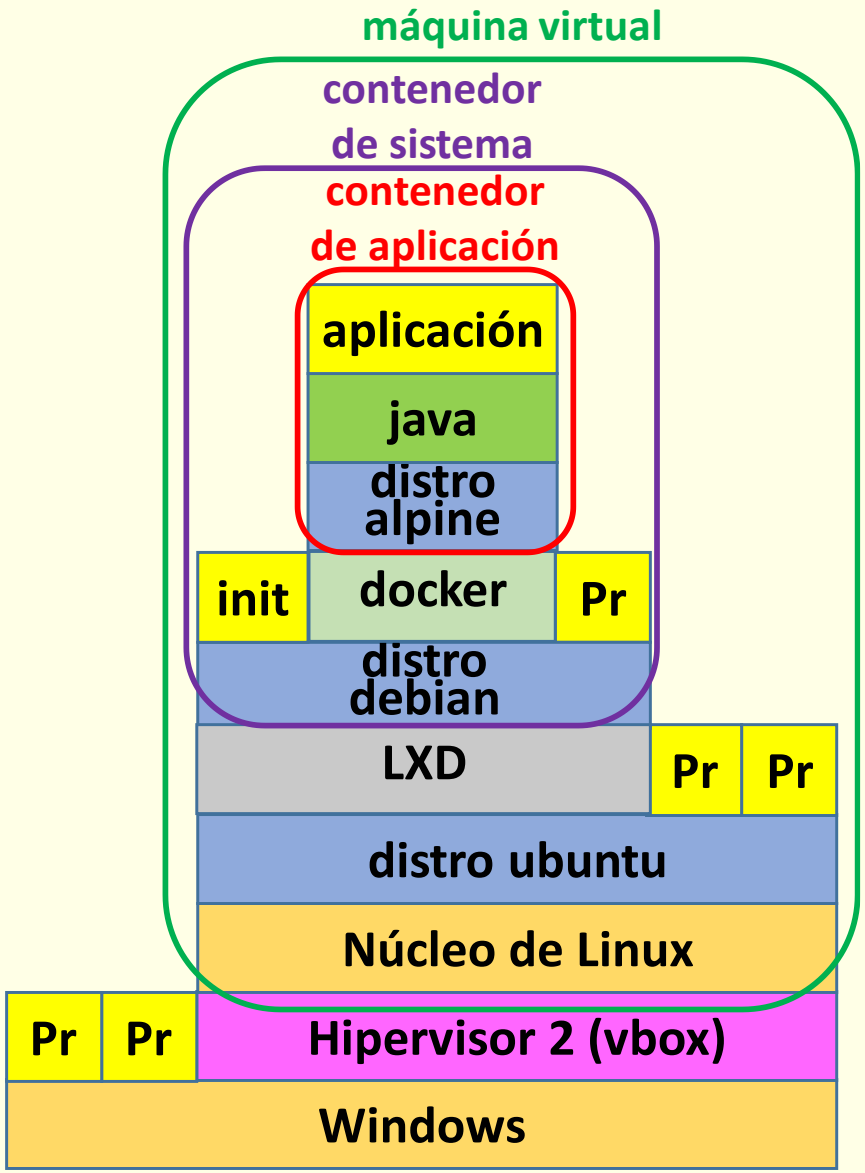
Contenedores de sistema (p.ej. *LXD*)

- Contenedores como sustitutos de virtualización de tipo 2
- Imagen asociada a contenedor solo SO: no aplicación autocontenida
- Activación del contenedor: arranca proceso *init*
- Usuario puede trabajar interactivamente en ese entorno:
  - Instalar paquetes, crear usuarios, ejecutar aplicaciones...
- Puede compartir volúmenes con anfitrión y otros contenedores
- Conectividad con otros contenedores, anfitrión y exterior
- Parece confusa diferencia entre contenedor de aplicación y sistema
  - Se basan en los mismos mecanismos
  - Contenedor aplicación suele incluir SO: facilita depuración
  - Contenedor app multiproceso suele incluir proceso supervisor
    - Una especie de *init*
- Pero están diseñados para escenarios distintos
  - Y de hecho se pueden anidar

Contenedores de sistema



Contenedores de sistema y de aplicación



Virtualización anidada