02 - Arquitecturas de Sistemas Distribuidos

IIC2523 - Sistemas Distribuidos

Cristian Ruz – cruz@ing.puc.cl

Departamento de Ciencia de la Computación
Pontificia Universidad Católica de Chile

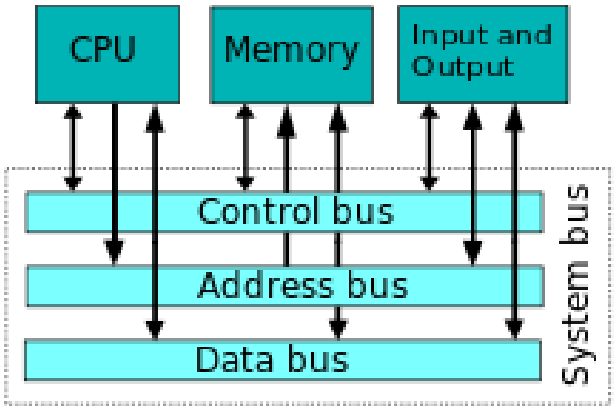
Semestre 2-2017

Contenidos

Contenidos

¿Cómo construir los sistemas?

Recordemos la arquitectura de Von Neumann



Queremos llevar esto a una arquitectura distribuida

- ¿Cómo organizar las CPUs?
- ¿Cómo organizar la memoria?
- ¿Cómo organizar la entrada/salida de datos?

Taxonomía de Flynn

Michael J. Flynn (1972)[1] propuso cuatro modos de procesamiento.
De acuerdo a cantidad de flujos de control y cantidad de flujos de datos:

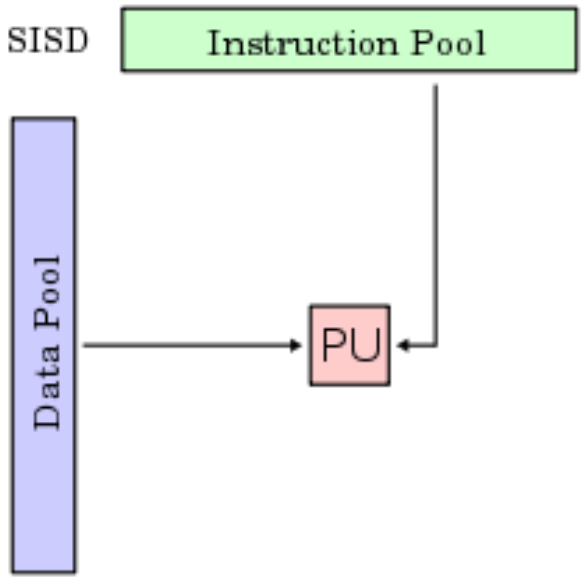
	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

[1] M.Flynn. *Some computer organizations and their efectiveness*, IEEE Transactions on Computers, 1972

SISD: Single Instruction, Single Data

Un flujo instrucciones, operando sobre un flujo de datos.

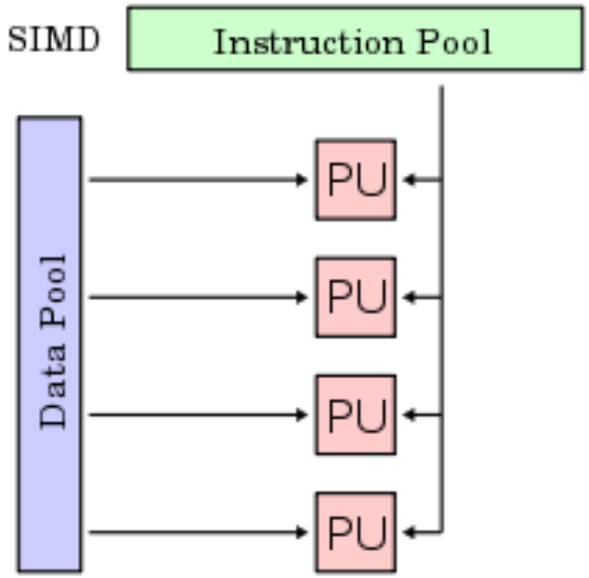
- Corresponde exactamente a una arquitectura de Von Neumann.
- Procesamiento secuencial de instrucciones
  - Incluyendo *pipelining*, *prefetching*, *branch prediction*
- PCs tradicionales, mainframes



SIMD: Single Instruction, Multiple Data

Un flujo de instrucciones, operando sobre distintos datos.

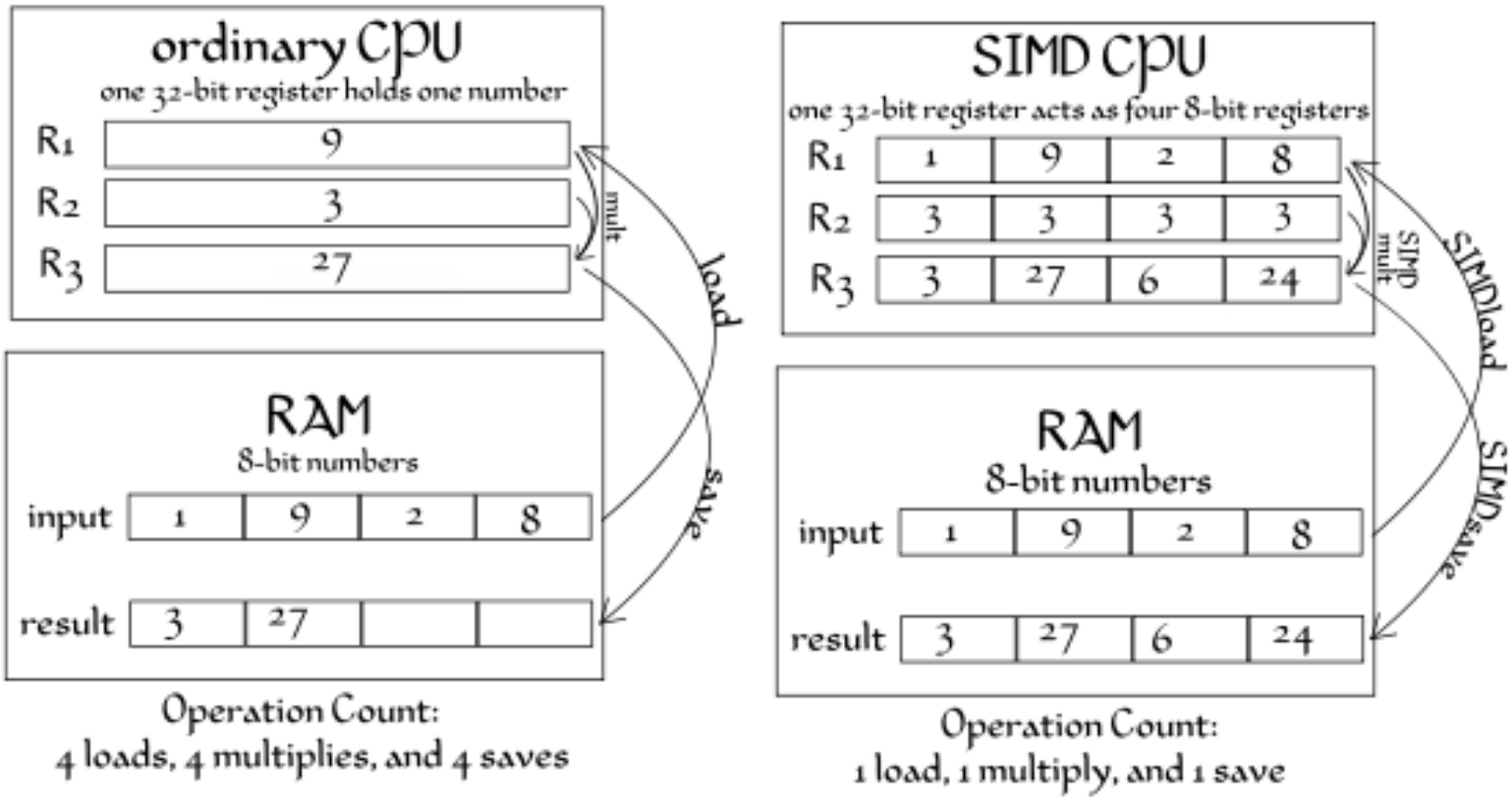
- El flujo de instrucciones se replica en múltiples procesadores.
  - Cada instrucción se ejecuta sobre distintos flujos de datos.
- Aplicaciones sobre vectores o matrices
  - Aplicaciones científicas, multimedia, videojuegos
- Procesadores vectoriales
  - Cray, Intel MMX (Pentium), AltiVec (PowerPC), 3DNow!(AMD)
- Explotan **paralelismo a nivel de datos**



SIMD: Single Instruction, Multiple Data

Ejemplo

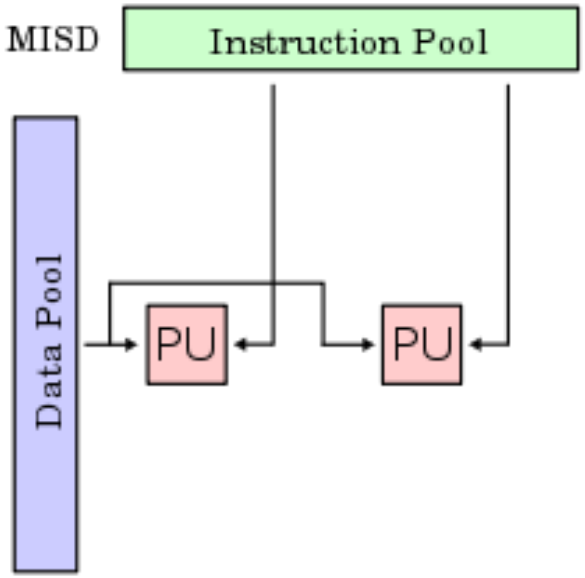
Tarea: multiplicar 4 números almacenados en RAM



MISD: Multiple Instruction, Single Data

Múltiple flujos de instrucciones, sobre un único flujo de datos

- Solo paralelismo de control
- Pipelining de instrucciones
- Systolic Arrays
- No es una arquitectura muy común

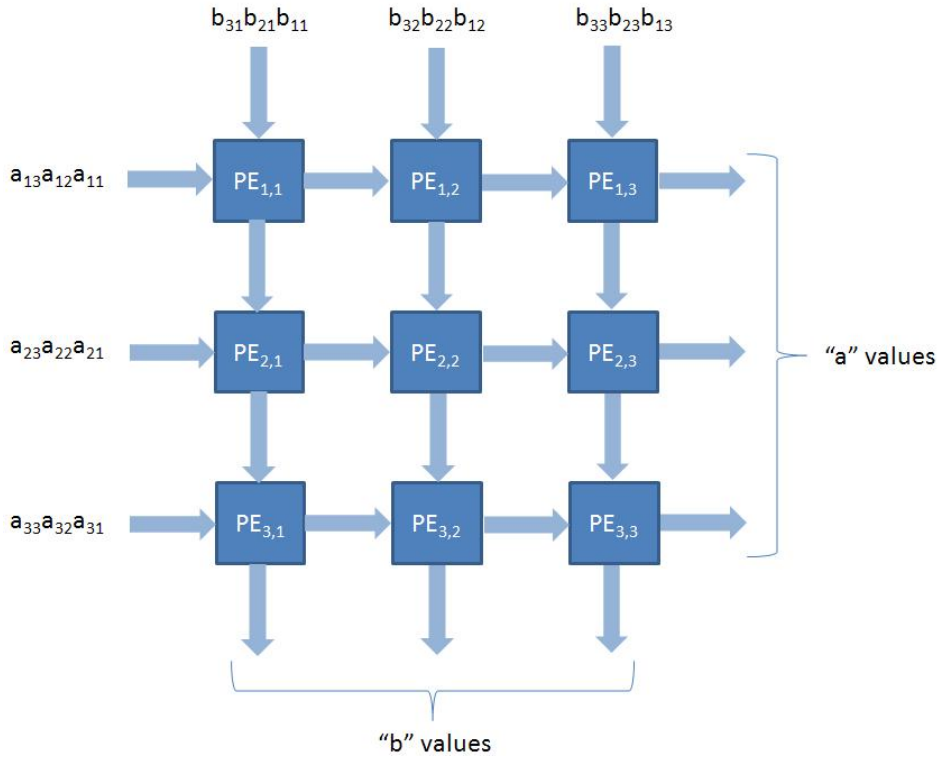


MISD: Multiple Instruction, Single Data

Ejemplo

Arreglos sistólicos

Procesadores independientes, compartiendo su output

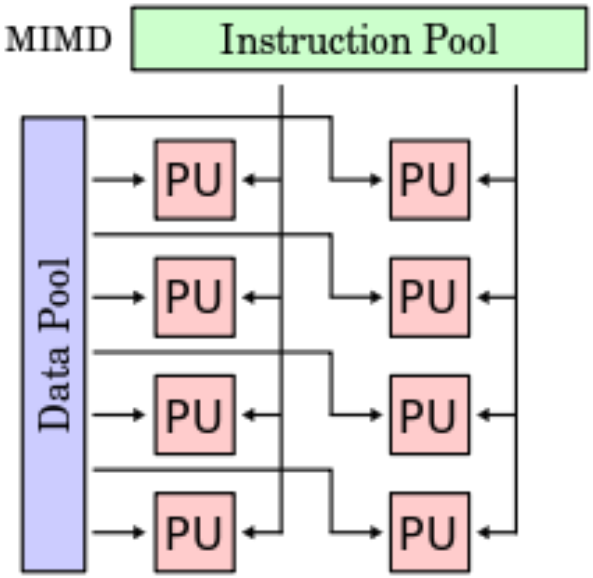


¿Otra aplicación?

MIMD: Multiple Instruction, Multiple Data

Múltiples flujos de instrucciones, sobre múltiples flujos de datos.

- Caso más general de computación paralela
- ¿Memoria compartida o distribuida?
- Multi-cores
- Explotan **paralelismo de control y de datos**



MIMD: Multiple Instruction, Multiple Data

También a través de grupos de instrucciones
Single Programa Multiple Data: SPMD

- Caso particular de MIMD
- Más general que SIMD

Multiple Programa Multiple Data: MPMD

- Programa divididos en cores
- Trabajan de manera colaborativa: Master/worker

Memoria

¿Cómo organizar el acceso a la memoria?
Antes de eso hay que elegir una alternativa:

- Memoria compartida
- Memoria distribuida

Memoria compartida

Todos los procesadores están conectados a una memoria globalmente accesible.

- Todos pueden leer o escribir a la vez
  - Memoria podría ser un cuello de botella
  - . . . ¡pero podemos usar *caches*!
- Se requiere *coherencia* de memoria
  - ¿El dato leído es el mismo que está en memoria?
  - ¿y si alguien lo cambió justo después que lo leí?
- Coherencia puede proveerse por *hardware* o *software*
  - Se pueden usar distintos **modelos de consistencia**

Memoria compartida

pero ... ¿cómo la organizo?

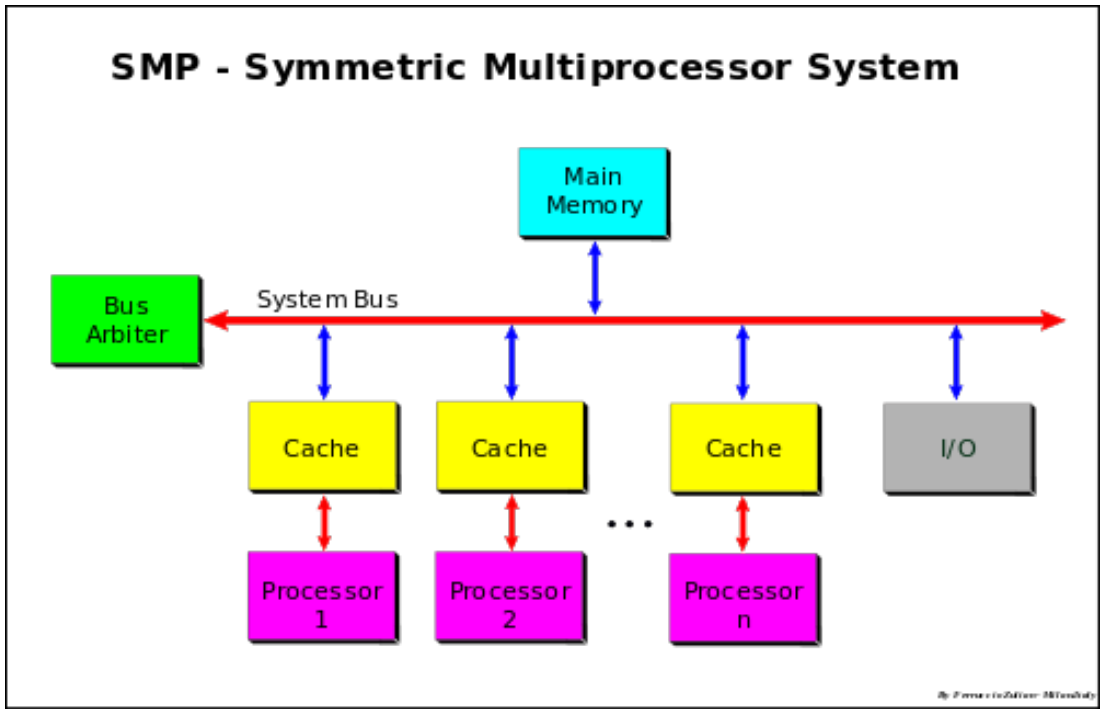
De acuerdo al tipo de acceso:

- UMA, Uniform Memory Access
- NUMA, Non-Uniform Memory Access

Memoria compartida: UMA

Uniform Memory Access

Todos los procesadores acceden de la misma manera a la memoria.
Ejemplo: Symmetric MultiProcessor (SMP)

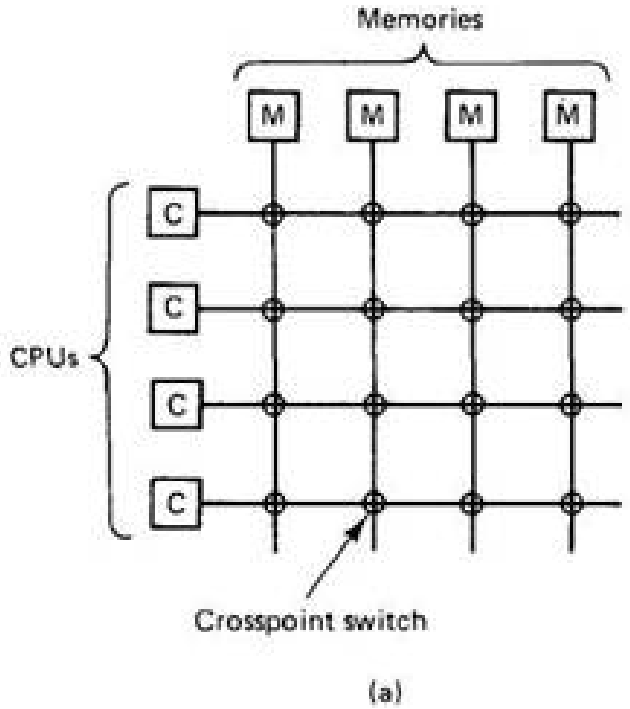


¿Como evitar congestión? $\rightarrow$ ¡Cache!
¿Cómo lo mantengo coherente? ...

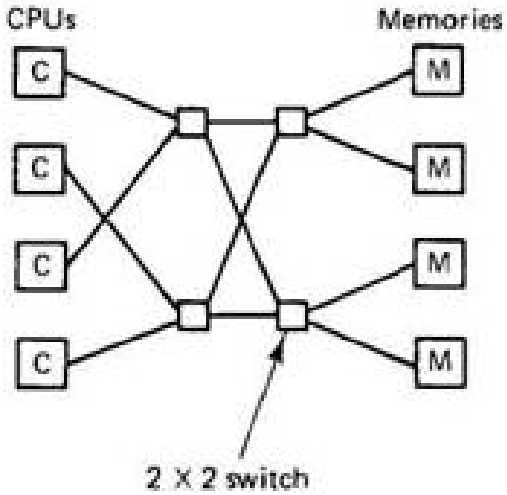
Memoria compartida: UMA

Uniform Memory Access

No solo puede ser un bus. También puede ser un *crossbar switch*:



(a)



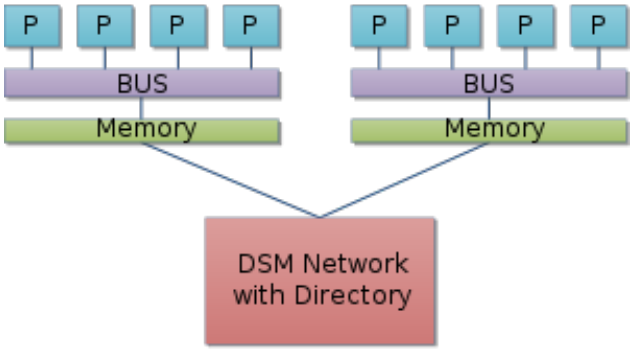
(b)

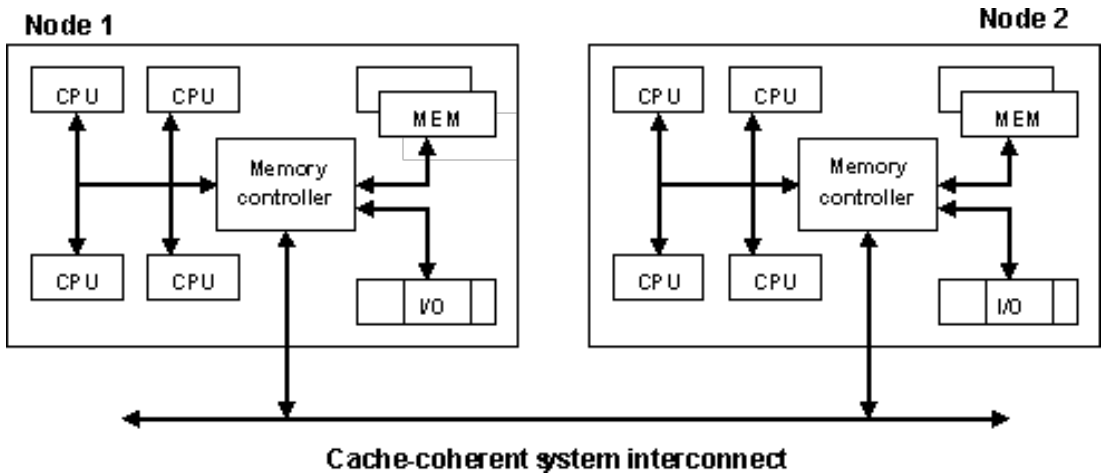
Memoria compartida: NUMA

Non-Uniform Memory Access

Diferentes tipos de acceso. ¿Dependiendo de qué?

- Accesos locales versus accesos remotos
- Diferentes modos de acceder a la memoria
- Local >> Remoto





Clave: aprovechar **localidad de referencia**

Aún se debe mantener **cache coherence**

Memoria compartida: Cache Coherence

¿Cómo mantenar las copias en cache de manera coherente?
Diferentes niveles de coherencia

- Writes aparecen "instantáneamente"
- Cada procesador ve la misma secuencia de cambios
- Cada procesador puede ver distintas secuencias de cambios (i.e. no hay coherencia)

Algunas condiciones:

- Order preservation: $W_P(x)a \Rightarrow R_P(x)a$
- Coherent view: $W_{P2}(x)a \Rightarrow R_{P1}(x)a$
- Secuencial: $W_{P1}(x)a, W_{P2}(x)b \Rightarrow R_{P3}(x)a, R_{P3}(x)b \vee R_{P3}(x)b$

Inmediato si escritura fuera instantánea

Memoria compartida: Cache Coherence

Mecanismos:

- Coherencia basada en directorio.
  - Registro de copias. Invalidación al escribir.
- Snooping
  - Bus de coherencia de caches
  - Cada cache se preocupa de la coherencia con los demás
  - Tres estados: V (valid), D(dirty), S(shared)
  - Read miss $\rightarrow$ broadcast (y $D \rightarrow V$)
  - Local write $\rightarrow$ broadcast (y $V \rightarrow D$)
- Snarfing
  - Controlador de cache monitorea la escritura en memoria de sus datos
  - Updates inmediatos

Protocolos de coherencia:

- Distintos modelos de consistencia de memoria
- Secuencial, causal, weak, release, ...

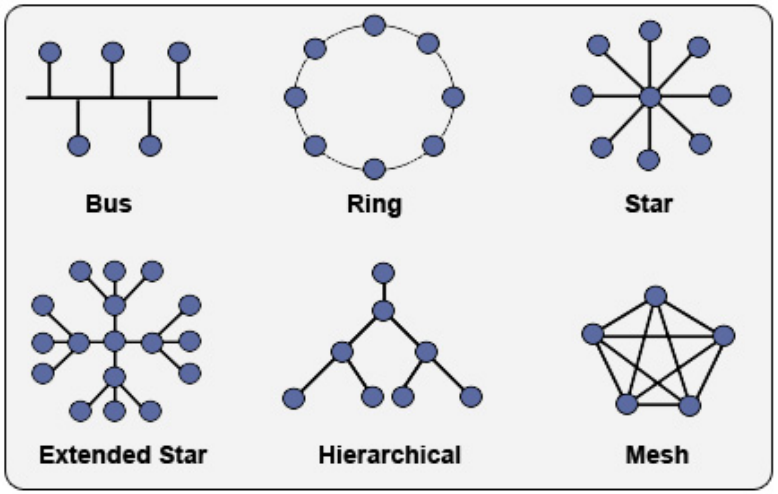
Memoria distribuida

Cada procesador tiene su memoria local.

- Accesos a memorias remotas son explícitos.
- No hay contención

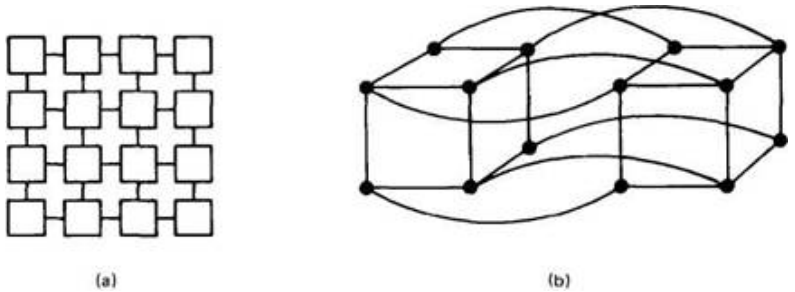
¿Cómo se comparten datos?

- Distintas topologías



Memoria distribuida: Mesh

Procesadores conectados en 2D grid.

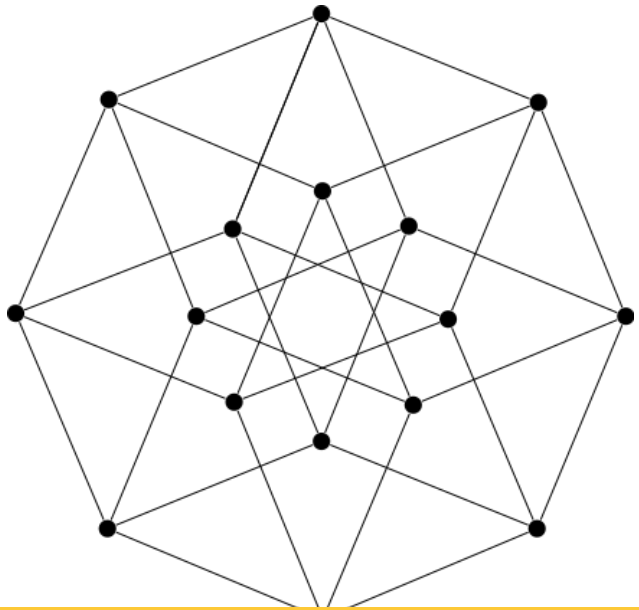


Apropiado para aplicaciones particulares.
Conexiones se incrementan en $O(\sqrt{(n)})$

Memoria distribuida: Hypercube

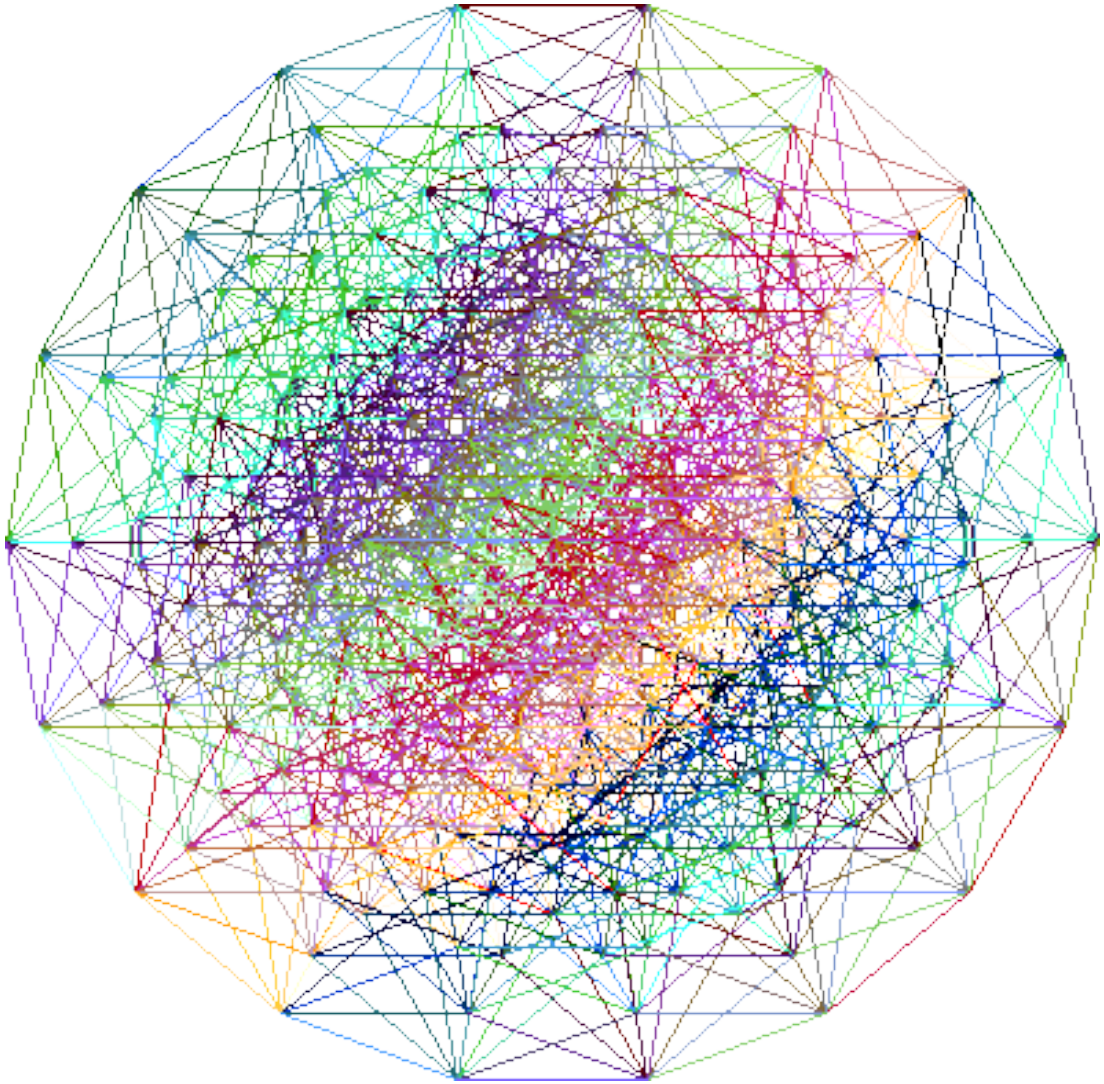
Menos conexiones

- Dimensión d
- Construcción recursiva
- Cada procesador tiene d conexiones a otros
- Complejidad de conexiones $O(\log(d))$
- Propiedades de distancia definidas
  - Distancia máxima: d
  - Dimensión $d \rightarrow 2^d$ nodos



Memoria distribuida: Hypercube

9D Hypercube



Contenidos

Sistemas Operativos

Qué características provee el sistema operativo para ejecutar procesos distribuidos.
¿Qué es lo que hace un Sistema Operativo?

- Abstraer detalles de infrastructura física
- Proveer una interfaz para ejecutar programas sobre el hardware
  - Archivos (en lugar de bloques de disco)
  - Sockets (en lugar de paquetes de red)

Network Operating Systems

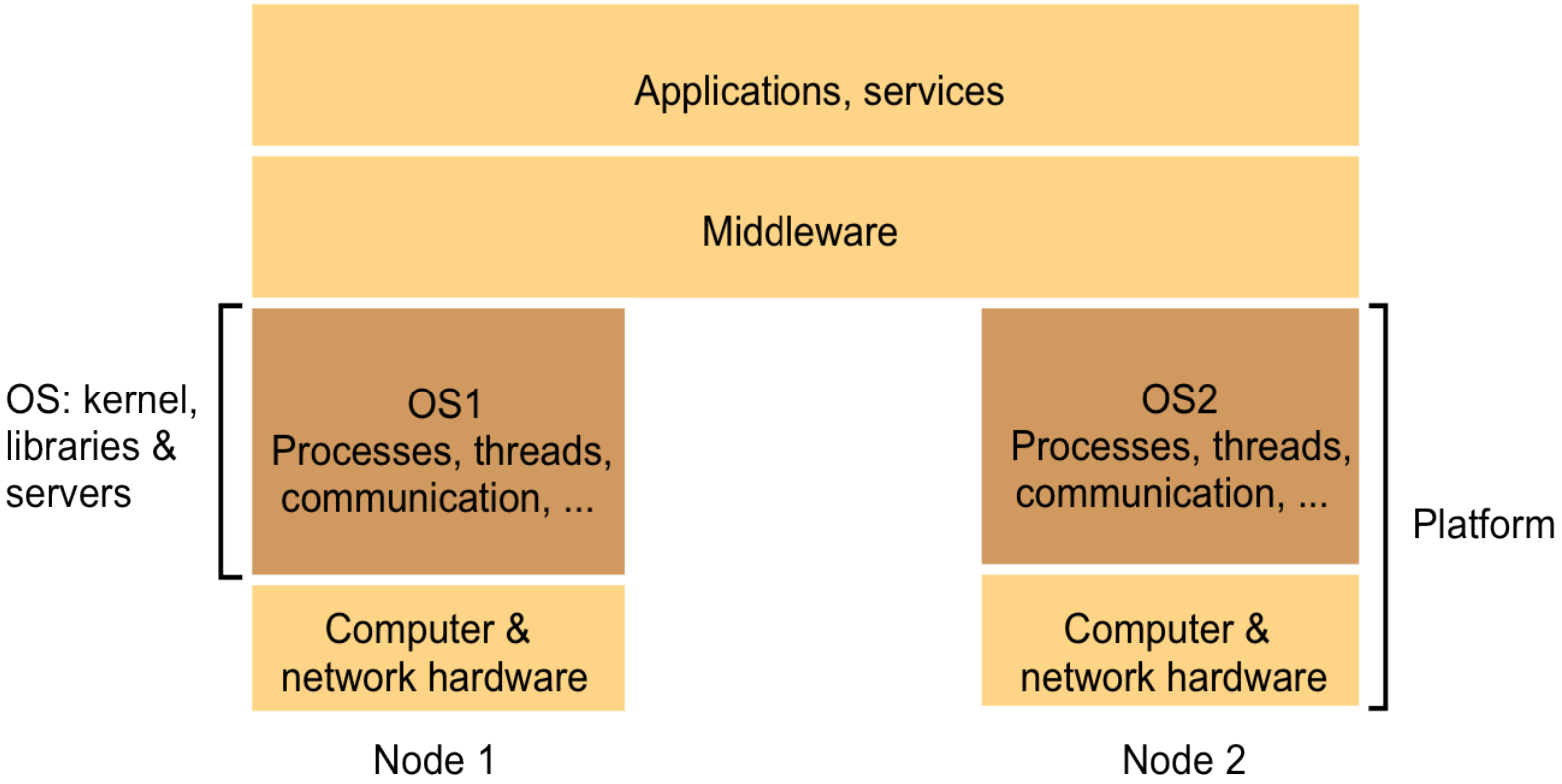
Windows, UNIX, permiten acceso a red: *Sistemas Operativos de Red*

- ¿Proveen una imagen de sistema único?
- Cada sistema administra sus propios recursos
- Nodos mantienen un alto grado de autonomía

¿Habrá algún *Sistema Operativo Distribuido* ?

- Más probable: S.O. + Middleware

Sistema Operativo y Middlewares



- Middleware provee la interfaz única sobre distintos SS.OO.
- Características interesantes:
  - Procesos y *threads*
  - Comunicación e invocación
  - Virtualización

Procesos y *Threads*

Proceso: entorno de ejecución con uno o más *threads*.

- Espacio de direccionamiento
- Sincronización de *threads* y acceso a recursos del S.O. (semáforos, *sockets*, ...)
- Acceso a recursos de alto nivel (archivos abiertos, ventanas, ...)

Thread: actividad dentro de un proceso

- Múltiples *threads* pueden compartir los recursos de un proceso.

Procesos vs. *Threads*

Procesos

- Caros de crear y administrar
- Sirven como un ambiente protegido para *threads*

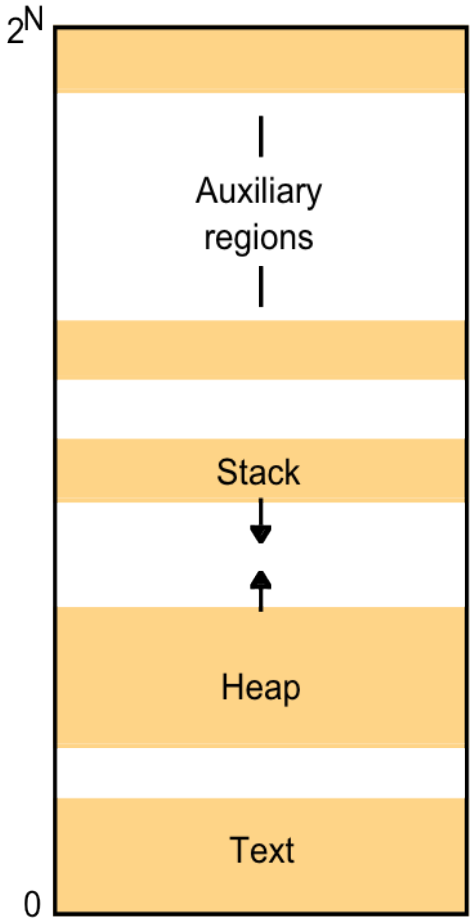
Threads

- Pueden ser creados y destruídos dinámicamente (y fácilmente)
- Permiten aprovechar el grado de ejecución concurrente
  - Separando tareas colaborativas
  - Aprovechando el *overlap* ejecución vs. I/O

Espacios de direcciones

Unidad virtual de memoria de un proceso.

- Regiones: código, *heap*, *stack*
- *Stack*s adicionales por *thread*



Creación de procesos

¿Dónde crearlo?

En UNIX:

- `fork`, nuevo proceso copiado del *caller*
- `exec`, ejecuta el código del programa indicado

En un ambiente distribuido: ¿en qué nodo crearlo?

- Nodo local
- Nodo externo indicado explícitamente
- Nodo externo seleccionado por otra entidad (*scheduler*)

¿Es posible modificar esta asignación?

- Política estática o adaptiva
- Sistemas de balance de carga

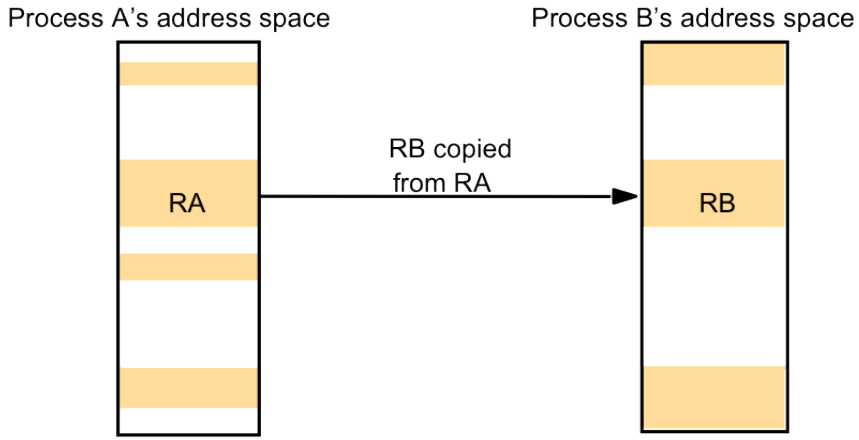
Creación de procesos

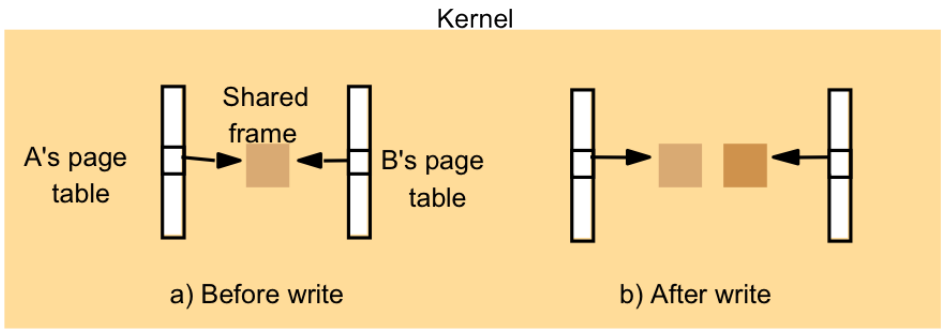
Creando el entorno

`fork`, copia del espacio de direcciones

¿Y si el proceso hijo no necesita todo el espacio?

- Copy-On-Write (COW)
  - Regiones compartidas hasta que se escriben

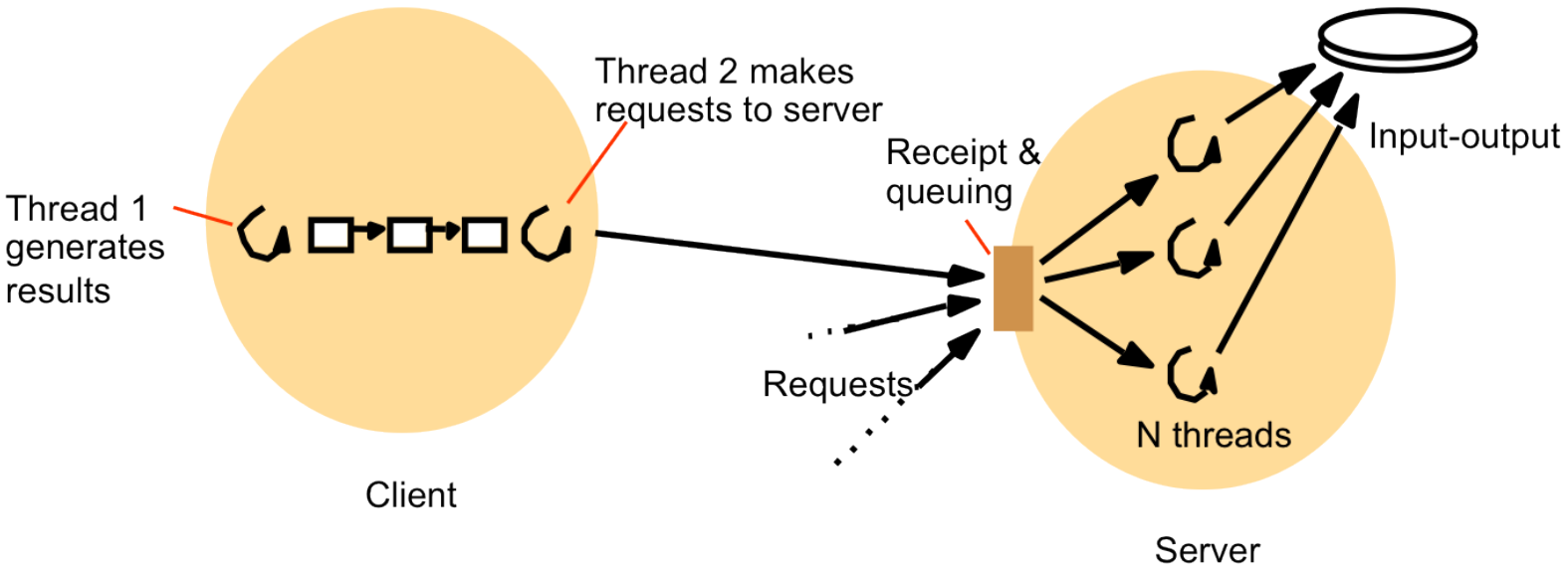




Threads

Procesos *multithreaded*

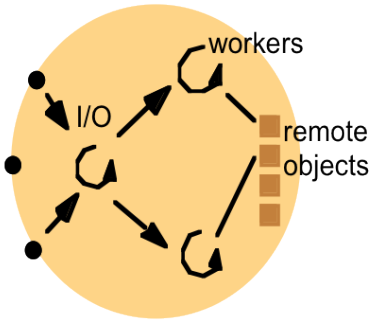
- Capacidad de atender más *request* (¿cuántas?)



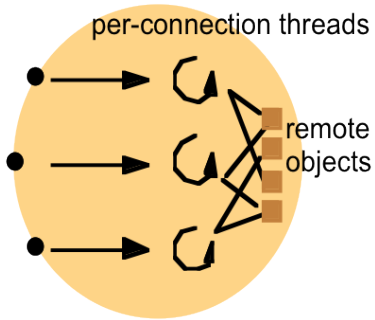
Threads

Distintas maneras de asignar *threads* a tareas.

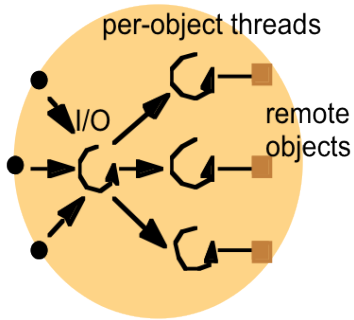
- *Pool* de *workers*
- *Thread-per-request*
- *Thread-per-connection*
- *Thread-per-object*



a. Thread-per-request



b. Thread-per-connection



c. Thread-per-object

Programando *threads*

Kernel *threads*

- Creados y manejados por el S.O.
- Input/Output administrador por S.O.

User *threads*

- Provisto por librerías de usuario
- Más livianos que kernel *threads*
- No pueden aprovechar directamente multiprocesadores
- Administración debe ser proveída por la librería
- Posibilidad de crear *deadlocks* en operaciones de I/O
- *pthreads*, CThreads, Java *Thread* class

Administración de *Threads*

Estados en Java: SUSPENDED, RUNNABLE, READY

- Thread(ThreadGroup group, Runnable target, String name)
- setPriority(int newPriority), getPriority()
- run()
- start(), SUSPENDED $\rightarrow$ RUNNABLE.
- sleep(int millisecs)
- yield()
- destroy()

Sincronización

- thread.join(long millisecs)
- thread.interrupt()
- object.wait(long millisecs, int nanosecs)
- object.notify(), object.notifyAll()

Comunicación e invocación

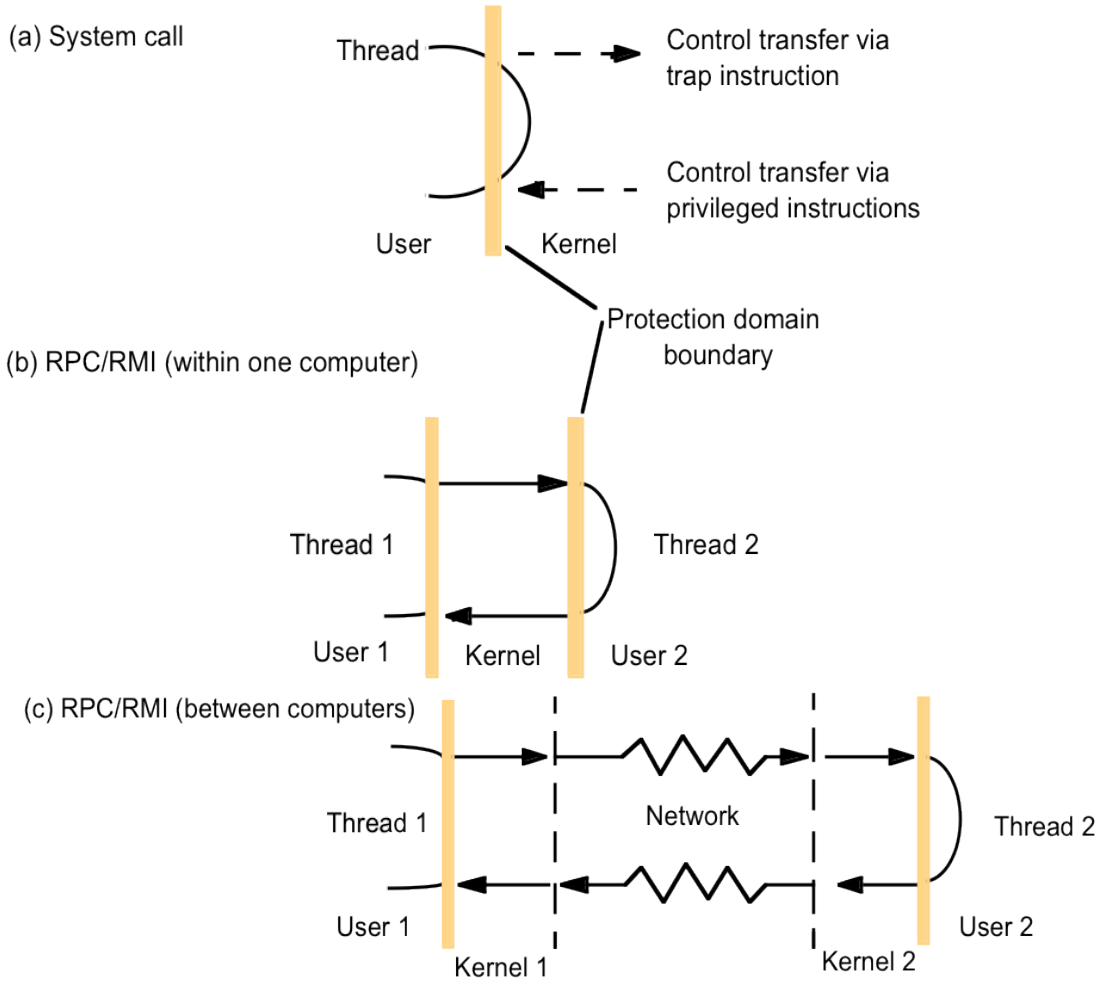
Comunicación: transferencia de datos entre procesos, posiblemente en nodos distintos

Invocación: Solicitud de ejecución de una operación en un espacio de direcciones distinto

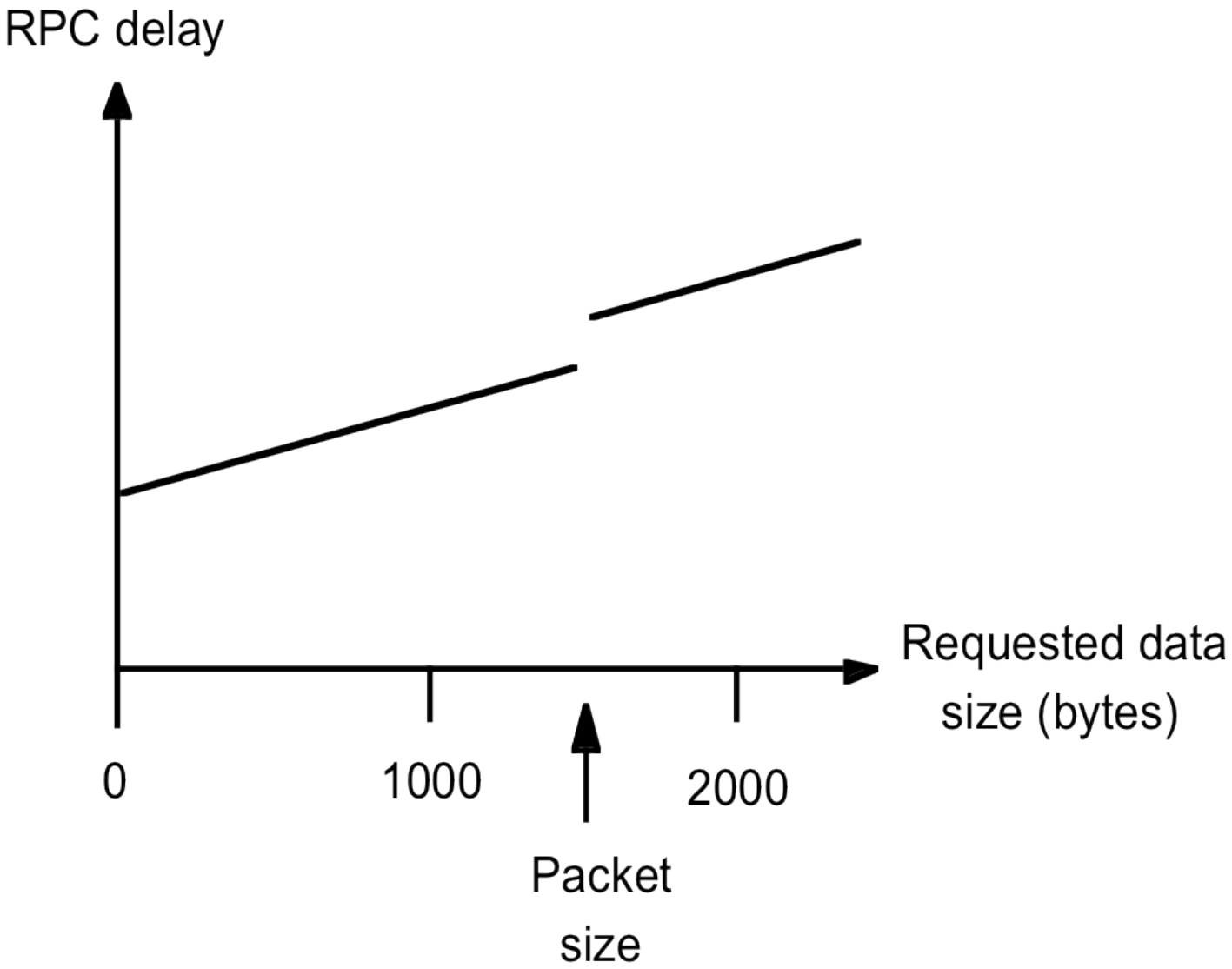
Costo de invocación

Tipos de comunicación

- RPC null $\sim 10^{-3}$ sec
- $\sim$ 100 bytes transmitidos
- Llamado normal $\sim 10^{-6}$ sec



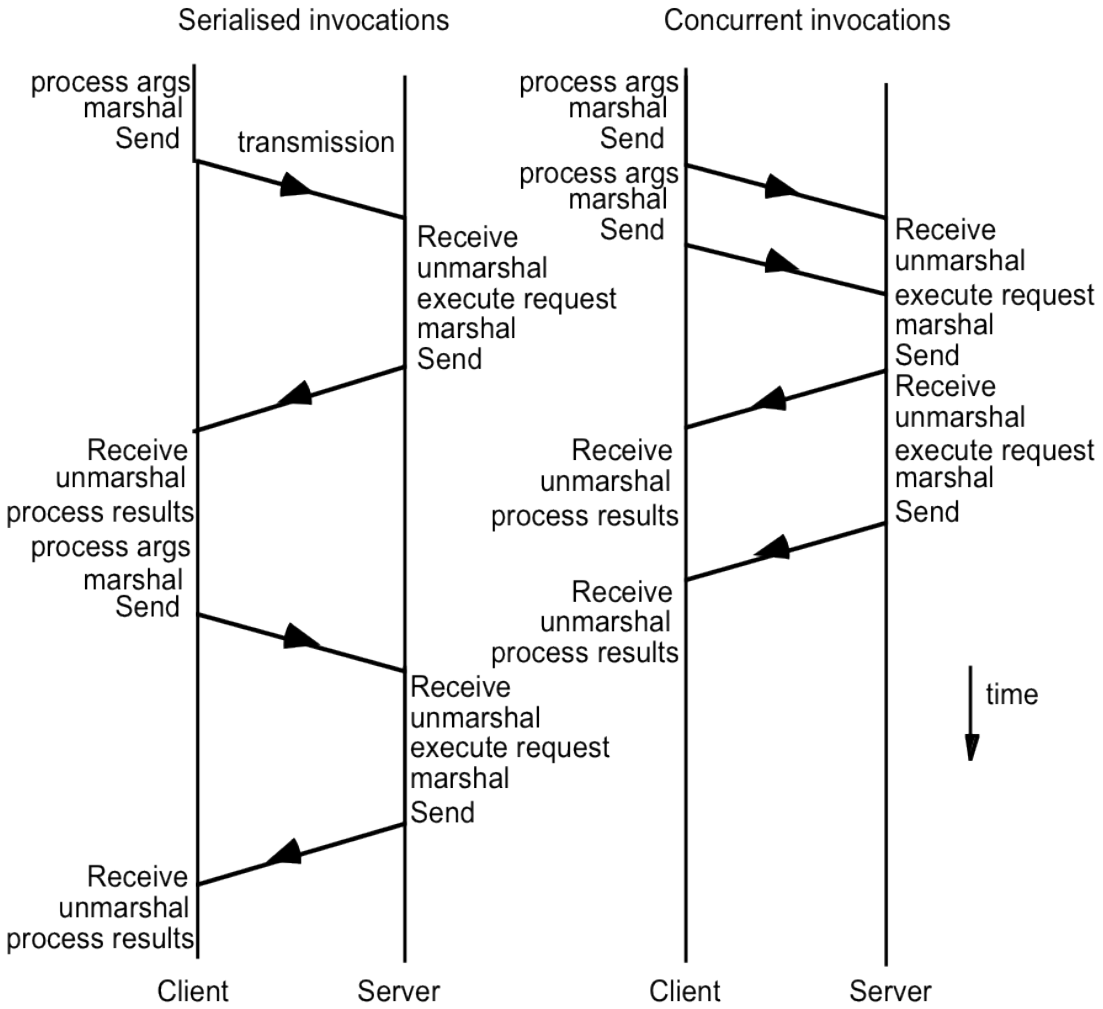
Costo de invocación



¿Podemos mejorar?

Comunicación asíncrona

- Invocaciones concurrentes
  - Aprovechando *multi-threading*
- Invocaciones asíncronas
  - Solo se hace un *rendez-vous*



Contenidos

Virtualización

Objetivo: proveer *máquinas virtuales* sobre una misma máquina física.

- Cada máquina ejecuta su propia versión del S.O.
- Multiplexación de recursos

Virtualización

¿Para qué sirve?

- Máquinas virtuales pueden ser migradas, i.e., mapeadas a distintas máquinas físicas.
- *Cloud Computing*. Permite provee IaaS (Infrastructure as a Service).
- Desafíos en asignación de recursos entre máquinas virtuales y físicas.
- Utilización de otros sistemas operativos en una misma máquina *host*.

Virtualización

¿Quién lo administra?

Virtual Machine Monitor, ó *Hypervisor*

- Provee una interfaz similar a aquella de la máquina física.

Diferentes niveles:

- Full Virtualization
- Paravirtualization

Ejemplos:

- VirtualBox
- VMWare
- Parallels
- Virtual Server
- Xen

Xen[2]

Solución de virtualización. (XenoServer project, 2003)
Objetivo: proveer infrastructura pública para computación distribuida de gran escala.

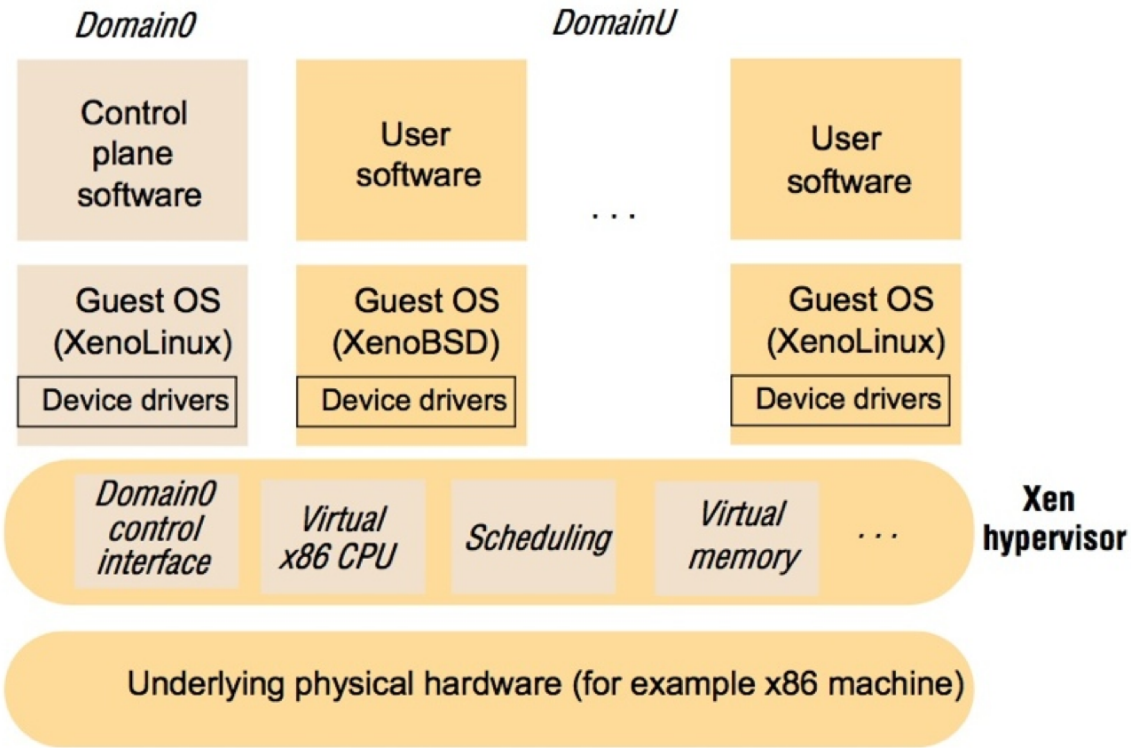
- Conjunto de XenoServers
- Administrados por Xen Virtual Machine Monitor

Xen

- Múltiples SS.OO. ejecutando de manera aislada en hardware tradicional, y con poco *overhead*
- Cada S.O. cree que es el único ejecutando.

[2] http://www.xen.org

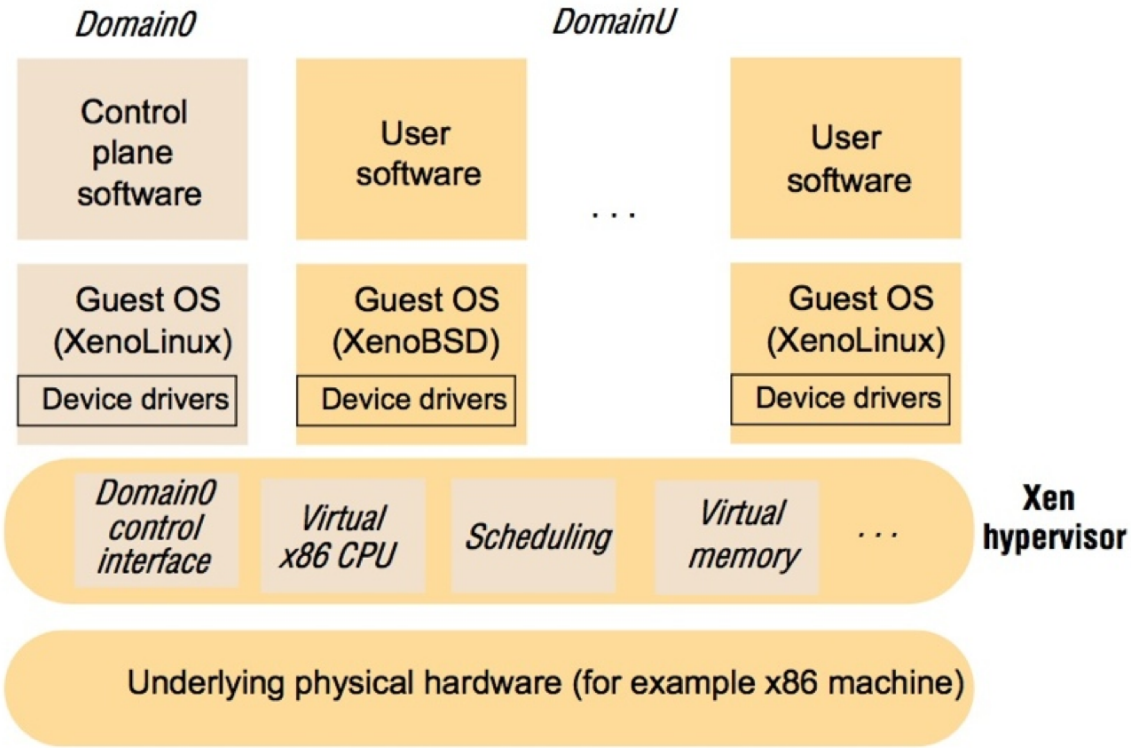
Arquitectura de Xen



Xen Hypervisor

- Provee manejo de recursos y aislamiento
- No conoce los *devices*, pero permite acceder a ellos
- Falla en Hypervisor $\rightarrow$ Falla en todos los S.O.

Arquitectura de Xen



Domains: Instancias de VMs en Xen

- **DomainU**: conjunto de todas las instancias
- **Domain0**: control, y acceso privilegiado a recursos
  - Creación de nuevos dominios

¿Virtualizar qué?

Las mismas funciones de un S.O.:

- CPU
- Scheduling
- Memoria
- Dispositivos

¿Cuándo se puede virtualizar?

Popek y Goldberg (1974) definen las **instrucciones sensibles** como aquellas que modifican el estado de la máquina de manera que puedan afectar a otros procesos.

- Instrucciones sensibles de control: permiten cambiar la configuración de los recursos de un proceso.
- Instrucciones sensibles de comportamiento: permiten leer estados privilegiados obteniendo información de la máquina física

Condición para virtualización

Todas las instrucciones sensibles deben ser privilegiadas.

- De esta manera el *Hypervisor* podría interceptar todas esas instrucciones.
- Sin embargo no es así en las arquitecturas x86.

¿Cuándo se puede virtualizar?

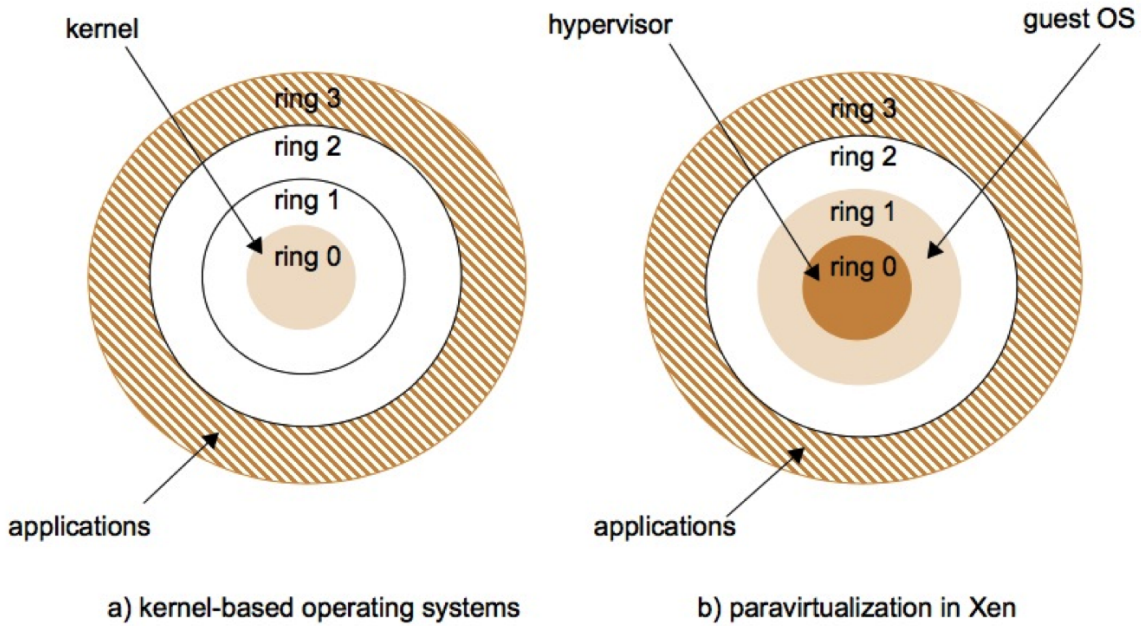
Dos soluciones para virtualizar CPU

1. Full virtualization: proveer la emulación para todas las instrucciones.
   - *Hypervisor* puede interceptar todo.
   - Produce más *overhead*.
2. Paravirtualization: permitir que algunas instrucciones se ejecuten en el hardware.
   - *Hypervisor* solo puede interceptar las privilegiadas.
   - El S.O. *guest* debe manejar las instrucciones sensible.
   - Más eficiente, pero require modificar el S.O. *guest*.

¿Cómo virtualizar?

Ejemplo: 4 niveles de privilegios en x86

- Kernel-based operating systems
- Paravirtualization en Xen



a) kernel-based operating systems

b) paravirtualization in Xen

- En el *guest* modificado, instrucciones privilegiadas se reescriben como *hypercalls*

Scheduling

Tradicionalmente:

- Scheduling de procesos
- Scheduling de *threads* intra-procesos

Xen provee un nivel más: VCPU (Virtual CPU)

- *Hypervisor* hace *scheduling* de VCPUs
- S.O. *Guest* hace *scheduling* de procesos
- S.O. o bibliotecas hacen *scheduling* de *threads*

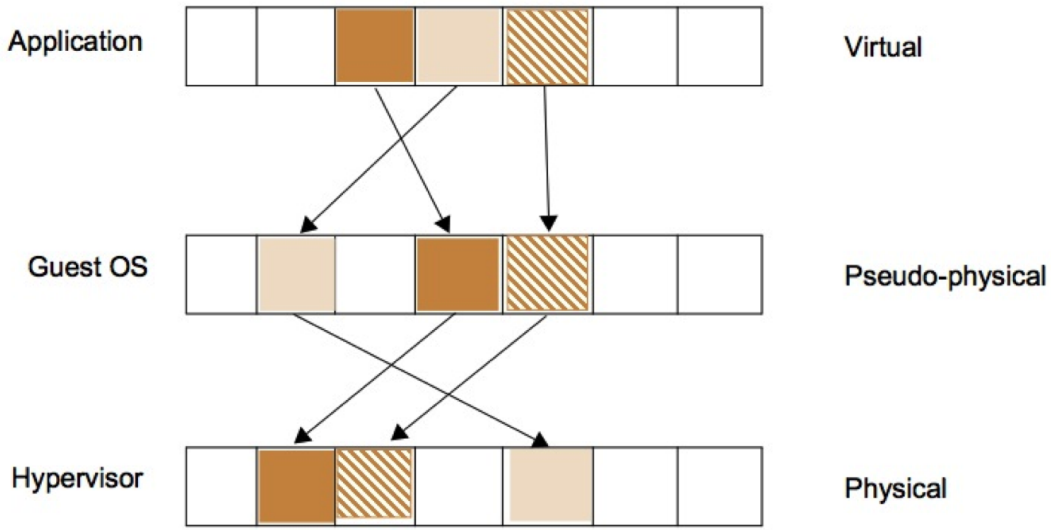
Dos estrategias:

- Simple Earliest Deadline First (SEDF)
  - VCPU con *deadline* más cercano
  - *Deadline* a partir de *slice* y *periodo*
- Credit Scheduler
  - Definido en base a *weight* y *cap* (porcentaje de tiempo que debe correr)
  - VCPU consumen créditos

Virtualización de Memoria

Dos dificultades:

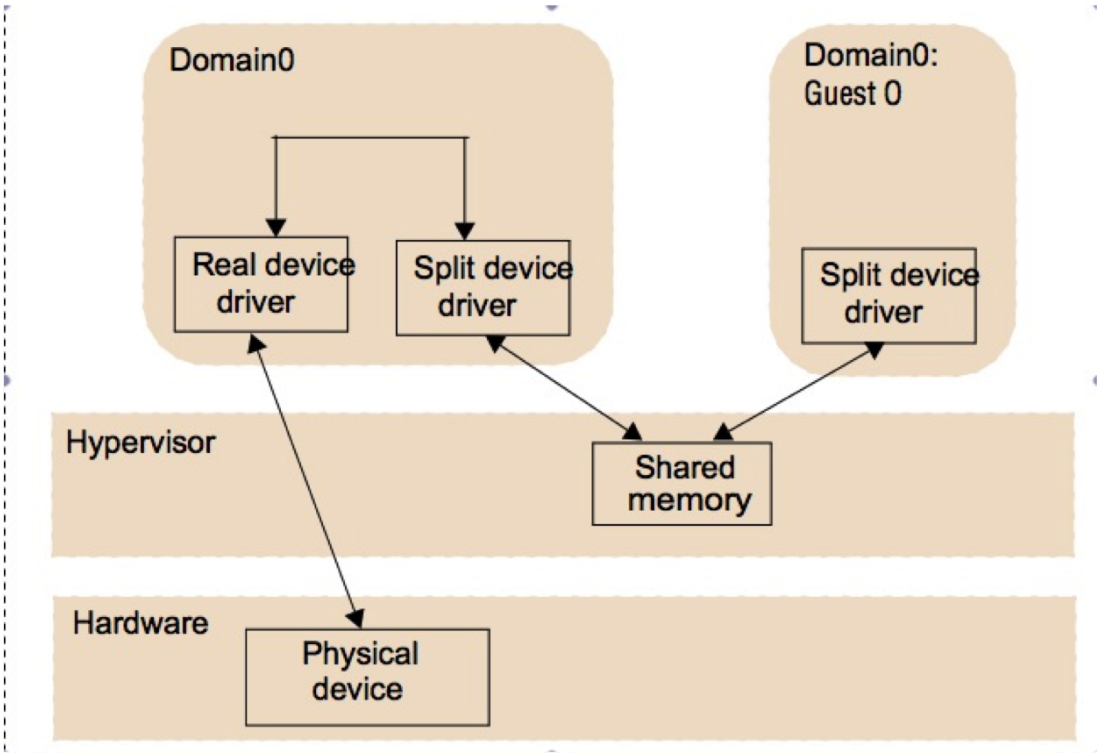
- Asignar nivel intermedio de mapeo
- Mantener protección entre distintos *guest*



- *Hypervisor* ocupa memoria física
- Debe proveer una abstracción de memoria física "limpia" al *guest*
- La memoria *pseudo-física* provee esta abstracción
- *Hypervisor* permite algunas instrucciones de asignación al *guest*

Virtualización de dispositivos

Split Device Drivers: imagen de dispositivo único



- *Back-end*: multiplexación, e interfaz genérica
- *Front-end*: *proxy* para el *guest*

Llevando un sistema *guest* a Xen

Se requiere:

- Reemplazar instrucciones privilegiadas por *hypercalls*
- Reimplementar instrucciones sensibles no-privilegiadas
- Portar el sistema de manejo de memoria
- Proveer *split-device drivers*

Contenidos

Sistemas de Archivos Distribuidos

Uno de los objetivos de un sistema distribuido: compartir recursos

- Sistemas P2P lo hacen para recursos genéricos y a gran escala
  - Réplicas en distintos nodos
  - Garantías de consistencia débiles, en privilegio del rendimiento
- Sistema de archivos distribuidos intentan replicar la experiencia de un sistema de archivos local
  - Adaptado para intranets y múltiples accesos concurrentes
  - Enfocados en transparencia de acceso
  - Sistemas de manejo de consistencia más estrictos

Dos ejemplos clásicos

- NFS
- AFS

Sistemas de Archivos Distribuidos

Sistemas de archivos locales

- Archivos: datos + atributos
- API: open, create, close, read, write, seek, link

Sistemas de archivos distribuidos

- Transparencia: acceso, ubicación, mobilidad, rendimiento, escalabilidad
- Actualizaciones concurrentes
- Replicación
- Tolerancia a fallos
- Consistencia
- Seguridad
- Eficiencia

Sistemas de Archivos Distribuidos

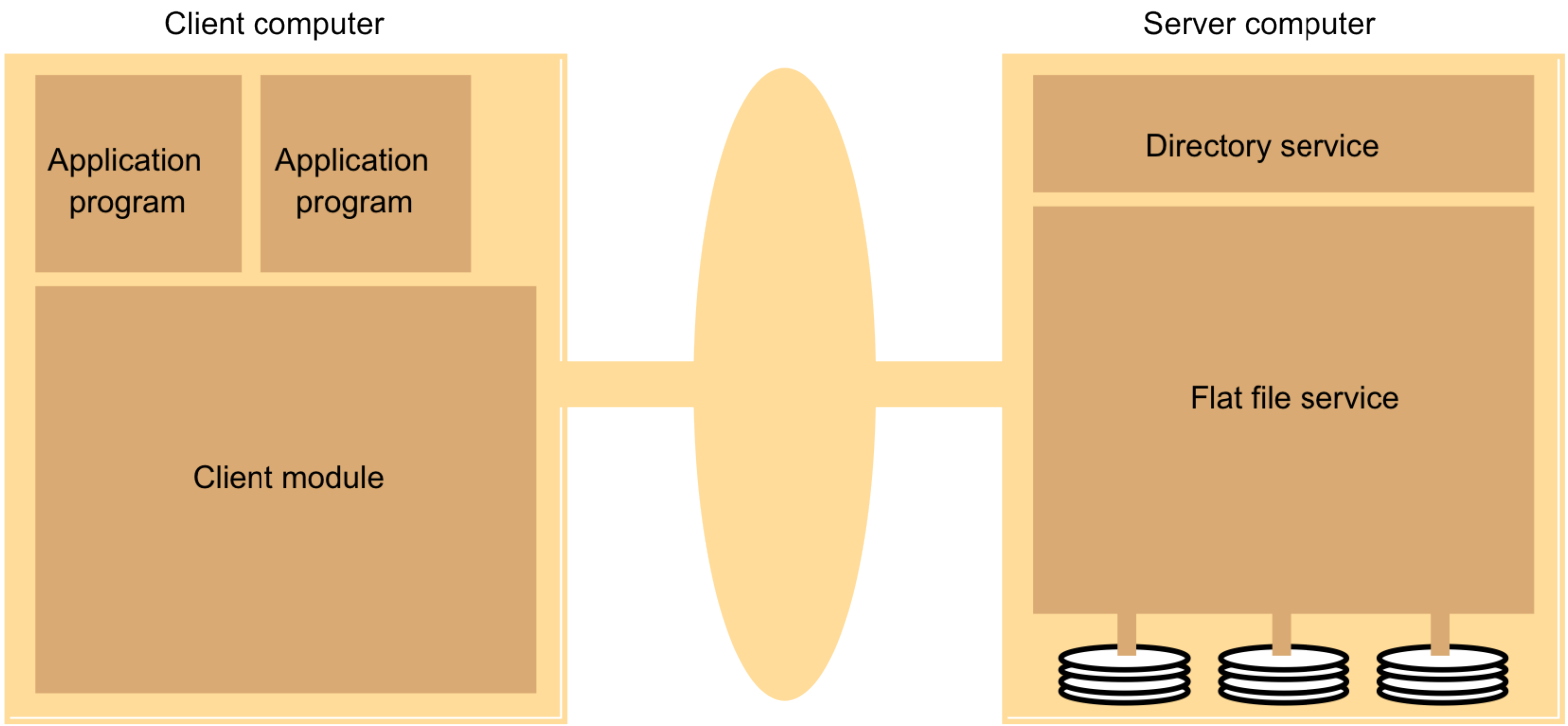
Arquitectura

Tres componentes arquitecturales:

- Servicio plano de archivos
  - Operaciones sobre los contenidos de los archivos
  - Asignación de UFID: Unique File IDentifier
  - Read, Write, Create, Delete, Get/Set Attributes
- Servicio de directorios
  - Mapeo *filename* $\leftrightarrow$ UFID
  - Puede implementar esquema jerárquico de nombres
  - Lookup, AdName, UnName, GetNames
- Módulo cliente
  - Proporciona servicios a aplicaciones en el cliente
  - Proporciona comunicación con el servicio de archivos
  - Puede usar *caching* para mejorar el rendimiento

Sistemas de Archivos Distribuidos

Arquitectura



NFS

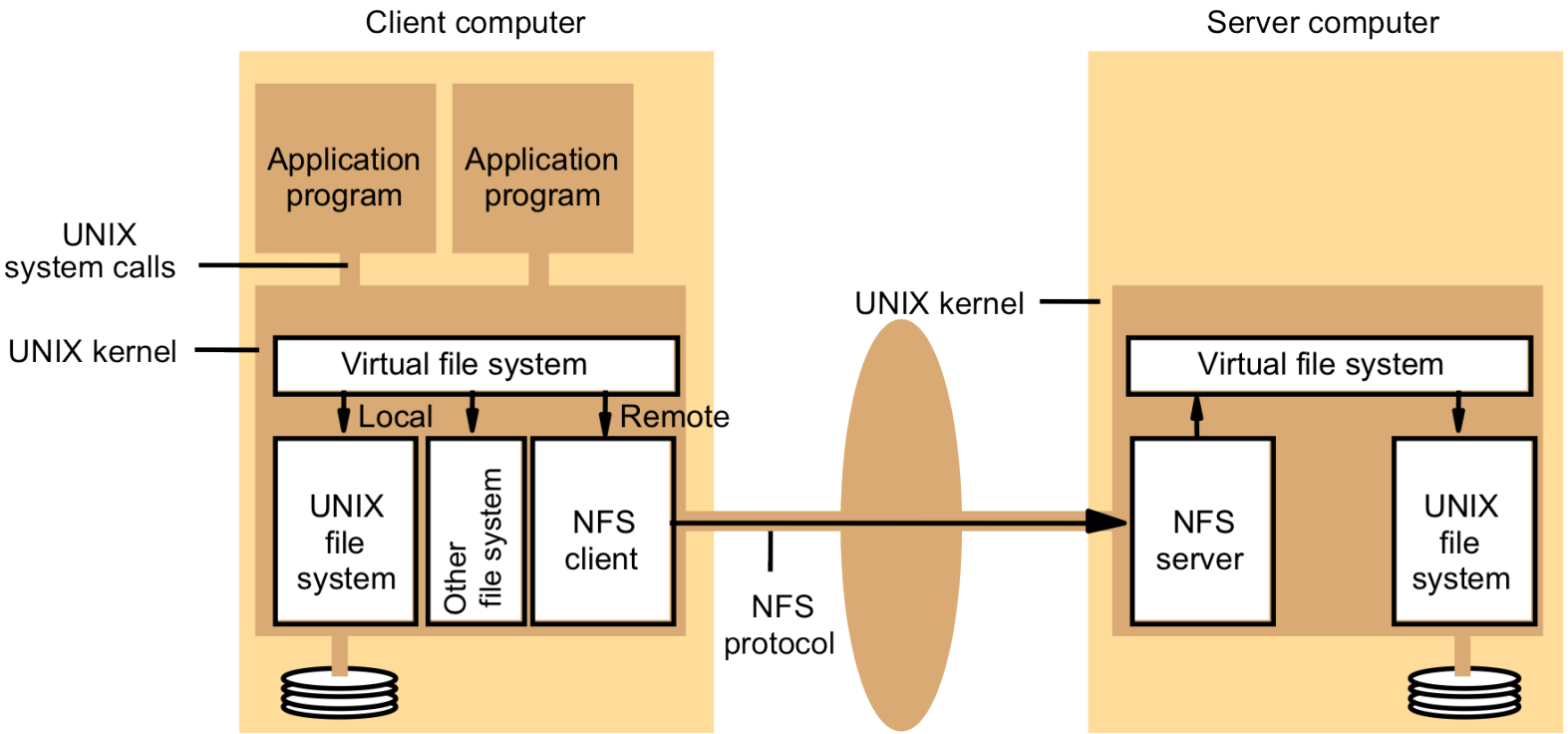
Sun Network File System

Desarrollado en Sun Microsystems (1985)

- Interfaz estándar $\rightarrow$ múltiples implementaciones
  - NFS v3: RFC 1813
- Modelo cliente-servidor
  - Cualquier nodo puede ser servidor de sus archivos
  - Comunicación vía RPC (usando TCP ó UDP)

NFS

Arquitectura



NFS
VFS

¿Cómo provee transparencia de acceso?

- Operaciones son hechas al módulo VFS (Virtual File System)
- VFS traduce llamadas a sistemas locales o remotos

¿Cómo ubica archivos?

- Identificador opacos: *file handle*s
  - Combinación de identificador local e i-nodo
- VFS relaciona directorio remoto con directorio local donde ha sido montado

NFS

API

Interfaz NFSv3, especificada en RFC 1813 (1995)

- {fh, attr} lookup (dirfh, name)
- {newfh, attr} create (dirfh, name, attr)
- remove(dirfh, name)
- {attr} getattr(fh), attr setattr(fh, attr)
- {attr, data} read (fh, offset, count)
- {att} write (fr, offset, count, data)
- rename(dirfh, name, todirfh, toname)
- link(newdirfh, newname, fh)
- symlink(newdirfh, newname, string)
- {string} readlink(fh)
- {newfh, attr} mkdir(dirfh, name, attr)
- rmdir(dirfh, name)
- {entries} readdir(dirfh, cookie, count)
- {fsstats} statfs(fh)

NFS

Mount service

En el servidor:

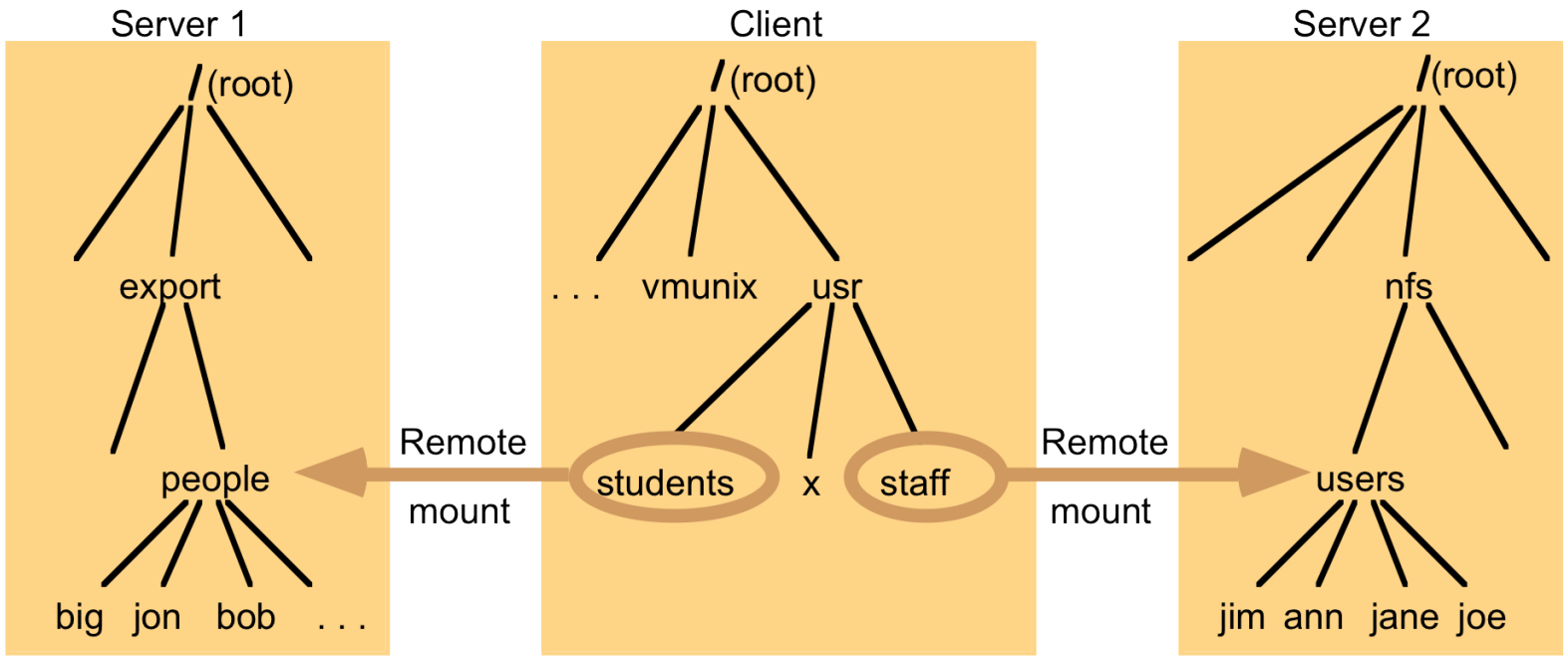
- `/etc/exports`. Sistemas de archivos locales disponible para ser montados remotamente, incluyendo lista de permisos.

En el cliente:

- `mount` usa llamadas RPC para contactar al servidor, obtener los filehandles remotos y asociarlo a un directorio local.
- *Hard mount*: procesos cliente intentan comunicarse con el servidor hasta obtener una respuesta.
  - En caso de caída del servidor, los procesos continúan intentando hasta que el servidor vuelva.
  - Se mantienen bloqueados. Procesos pueden no terminar.
- *Soft mount*: procesos cliente retornan aviso de falla luego de alguna cantidad de reintentos
  - Permite que la aplicación tome medidas al detectar una falla.
  - Evita que los procesos permanezcan bloqueados.
  - Aplicación cliente es responsable de efectuar el chequeo.

NFS

Mount service



NFS

Caching

Server caching

- *Write*s son almacenados en memoria y copiados a disco de acuerdo a una estrategia.
  - *Write-through*. Solicitudes de *write* son escritas al disco antes de enviar un *reply* al cliente.
  - Utilizar operaciones de *commit*. Escribir al disco solamente cuando el archivo ha sido cerrado. En ese momento se envía un *commit* al cliente y éste sabe que su escritura ha sido efectuada.

Client caching

- Cliente acumula solicitudes de *read*, *write*, *get/setattr*, *lookup*, *readdir* de manera de reducir las solicitudes transmitidas.
- Cliente debe chequear que los datos que él posee son válidos
  - Frescura t, timestamp caché T_C, timestamp modificación t_m
  - $(T - T_C < t) \vee (T_{m_{\text{client}}} = T_{m_{\text{server}}})$

NFS

Resumen

- Servicio *stateless*. Simplifica la recuperación de fallas.
- Transparencia de ubicacion y acceso
- Mobilidad de directorios compartidos debe ser manejada manualmente
- Escalabilidad dependiendo del balanceo de archivos de alta demanda
- Heterogeneidad debido a implementaciones para distintas plataformas
- Tolerancia a fallas similar a sistema local en caso de *hard mount*
- Consistencia bastante cercana a semántica de una sola copia
- Transmisión de bloques de archivos

AFS

Andrew File System[3]

Desarollado en Carnegie Mellon University, CMU (1986).

- Altamente enfocado a escalabilidad
- Transmisión y caching a nivel de archivos (NFS usa bloques)
- Intenta minimizar la comunicación cliente-servidor

[3] En honor a Andrew Carnegie y Andrew Mellon

AFS

¿Cómo obtiene escalabilidad? $\rightarrow$ ¡caching de archivos!

- *Whole-file serving*. Archivos son transferidos como unidades completas.
- *Whole-file caching*. Archivos recientemente usados se mantienen en caché local, aún luego de reboots. Copias locales son usadas de preferencia sobre las remotas.

Escenario típico

- *Open*. Si no hay copia local, se pide al servidor.
- Archivo es utilizado de manera local, con identificadores locales.
- *Read, Write*. Son ejecutados sobre la copia local.
- *Close*. Envía actualizaciones al servidor. Copia local se mantiene.

AFS

Algunas observaciones

- Mayoría de los archivos:
  - Actualizados de manera poco frecuente
  - Leídos por un sólo usuario
- Luego, las copias locales permanecen válidas por largo tiempo
- El caché local se utiliza como *working set*. Éste es un parámetro de la configuración.
- Suposiciones (basadas en observaciones estudiadas)
  - Muchos archivos son de tamaño menor a 10 KB.
  - *Reads* son alrededor de 6 veces más comunes que *Writes*.
  - Lectura es mayoritariamente sequencial antes que aleatoria.
  - Muchos archivos son leídos y escritos por un solo usuario.
  - Si un archivo ha sido abierto recientemente, probablemente volverá a ser abierto en el futuro cercano.
- ¿Qué tipos de archivos no son apropiados para AFS?
  - Archivos con múltiples lectura y actualizaciones frecuentes
  - Típicamente bases de datos

AFS

¿Qué falta?

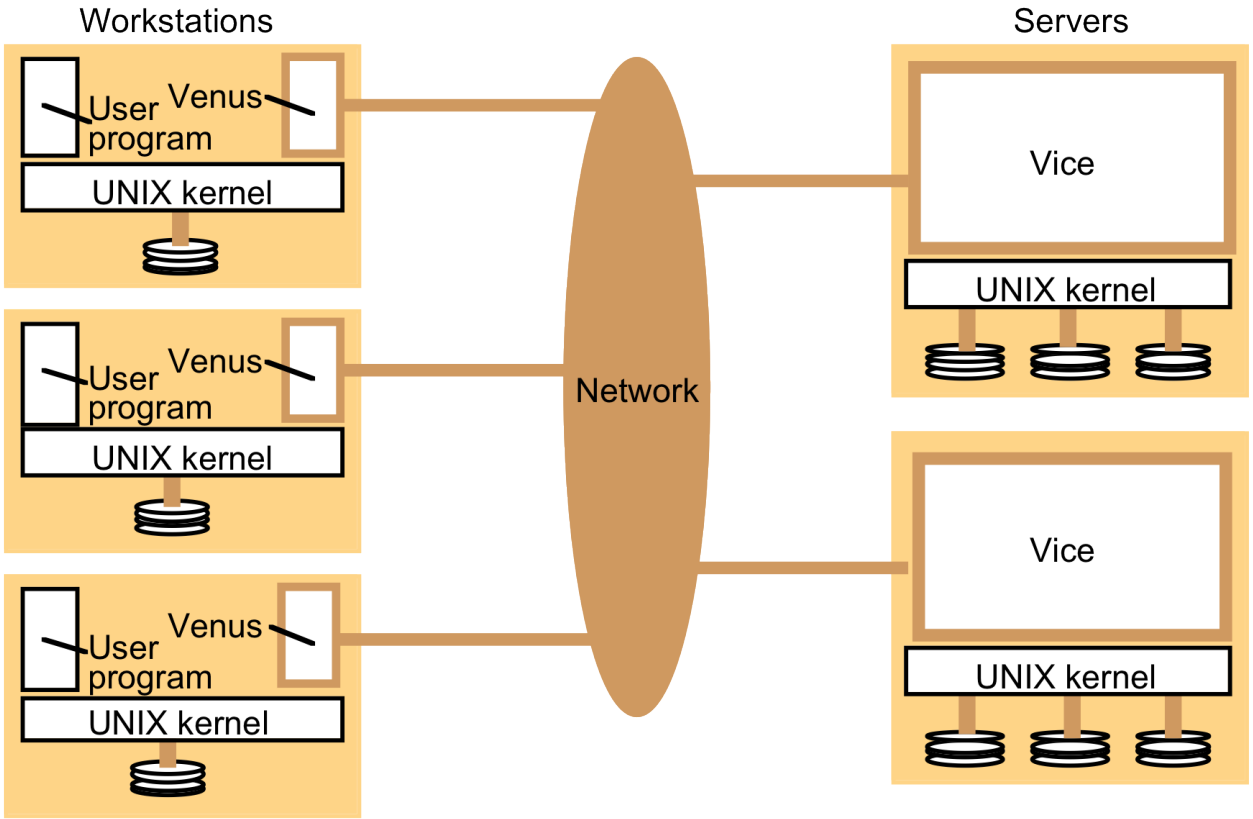
- ¿Quién intercepta llamados *open*, *close*?
- ¿Cómo ubicar el archivo en el espacio compartido?
- ¿De qué tamaño debería ser el caché local?
- Aunque sean pocos, siempre puede haber escrituras concurrentes
  - **¿Cómo se mantiene la consistencia?**

AFS

Arquitectura

Dos componentes

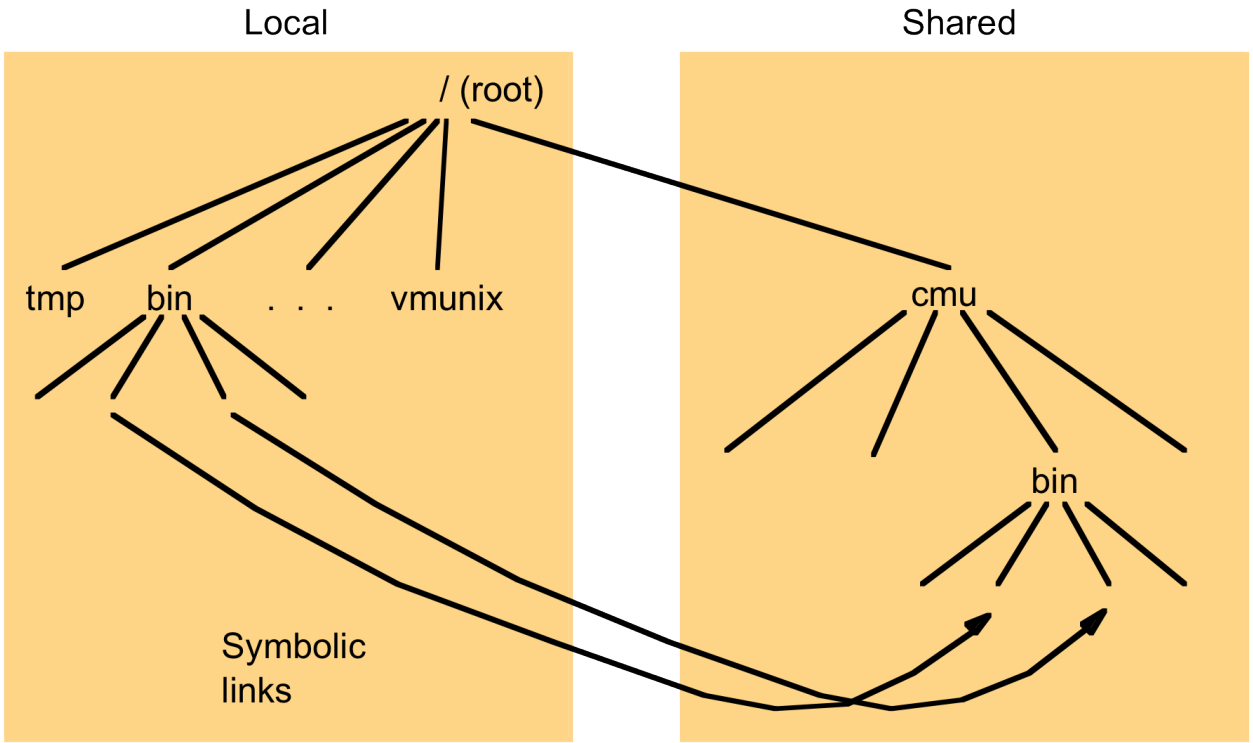
- **Vice**. Proceso servidor.
- **Venus**. Proceso cliente.



AFS

Espacios de nombres

Dos tipos de archivos: *local* y *shared*



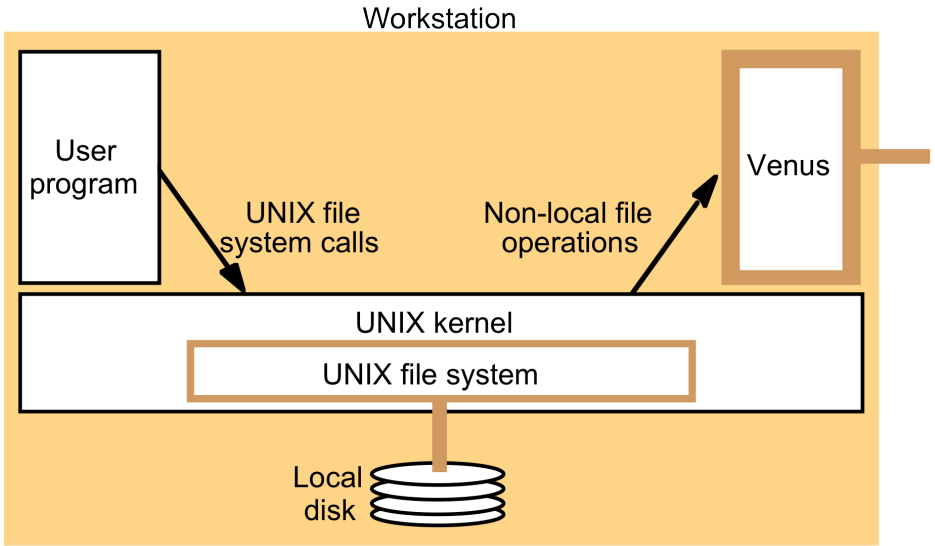
Archivos compartidos en directorio especial *cmu*.

- Espacio local para archivos temporales y archivos de S.O.
- Otros archivos son links simbólicos a espacio compartido.

AFS

Roles

- **Venus**
  - Intercepta llamadas *open* y *close*.
  - Administra el caché local.
- **Vice**
  - Provee *flat file service* y directorio jerárquico.
  - Identificación de archivos via *fid* de 96-bit.



AFS

Consistencia de caché

Consistencia débil usando *callback promises* con estados *valid* y *cancelled*.

open(filename, mode)

- Kernel: intercepción y redirección a *Venus*.
- Venus. Verificación en caché local.
  - Si no existe o *callback promise cancelled*, solicita copia a *Vice*
- Vice. Envía copia + *callback promise*, y registra *callback promise*.
- Venus. Copia del archivo en sistema local, *callback promise valid* y retorno a S.O.

close(FileDescriptor)

- Kernel: cierra archivo y notifica a *Venus*.
- Venus. Si hay modificaciones, envía actualización a *Vice*.
- Vice. Actualiza archivo y envía *callback* a todos los *Venus* registrados para este archivo.
- Venus. Al recibir *callback*, su *callback promise* pasa a *cancelled*.

AFS

Consistencia de caché

¿En caso de pérdida de *callback*s?

- Pasado tiempo T (minutos), o después de una pérdida de comunicación, se solicita *timestamp* de archivos con *callback promise valid*.
  - Si *timestamp*s difieren, *callback promises* pasan a *cancelled*

Consideraciones de rendimiento

- Muy eficiente para lecturas concurrentes.
- ¿Qué pasa con escrituras concurrentes?

AFS

Semántica de actualización

Aproximación a semántica de *una copia*.

- Sin pérdida de *callbacks* (AFSv1)
  - Luego de un *open(F)* exitoso: *latest(F,S)*
  - Luego de un *close(F)* exitoso: *updated(F,S)*
  - Luego de *open* o *close* fallido: *failure(S)*
- Considerando que se pueden perder *callbacks* (AFSv2)
  - Si se pierde un mensaje, cliente puede abrir copia no actualizada
  - Sin embargo esto se mantiene a lo más durante T
  - Luego de *open(F)* exitoso: *latest(F,S,0)* OR (*lostCallback(S,T)* AND *inCache(F)* AND *latest(F,S,T)*)

¿Y si hay **actualizaciones concurrentes**?

AFS

Resumen

- Servidor mantiene *callback*s por archivo
- Directorio (volúmenes) pueden ser compartidos por varios servidores
- Clientes no saben cual servidor es el responsable de un archivo
- Transmisión de archivos completos (en trozos de 64KB)
- Escalabilidad a través de *timestamp*s y *callbacks*
- Privilegia lecturas usando copias locales.
- Minimiza comunicación con servidor