

---

# Lecture 3: OS Structure II

microkernels, exokernels,  
virtual machines & modules

CSC 469H1F

Fall 2007

Angela Demke Brown



# Recap: Microkernels

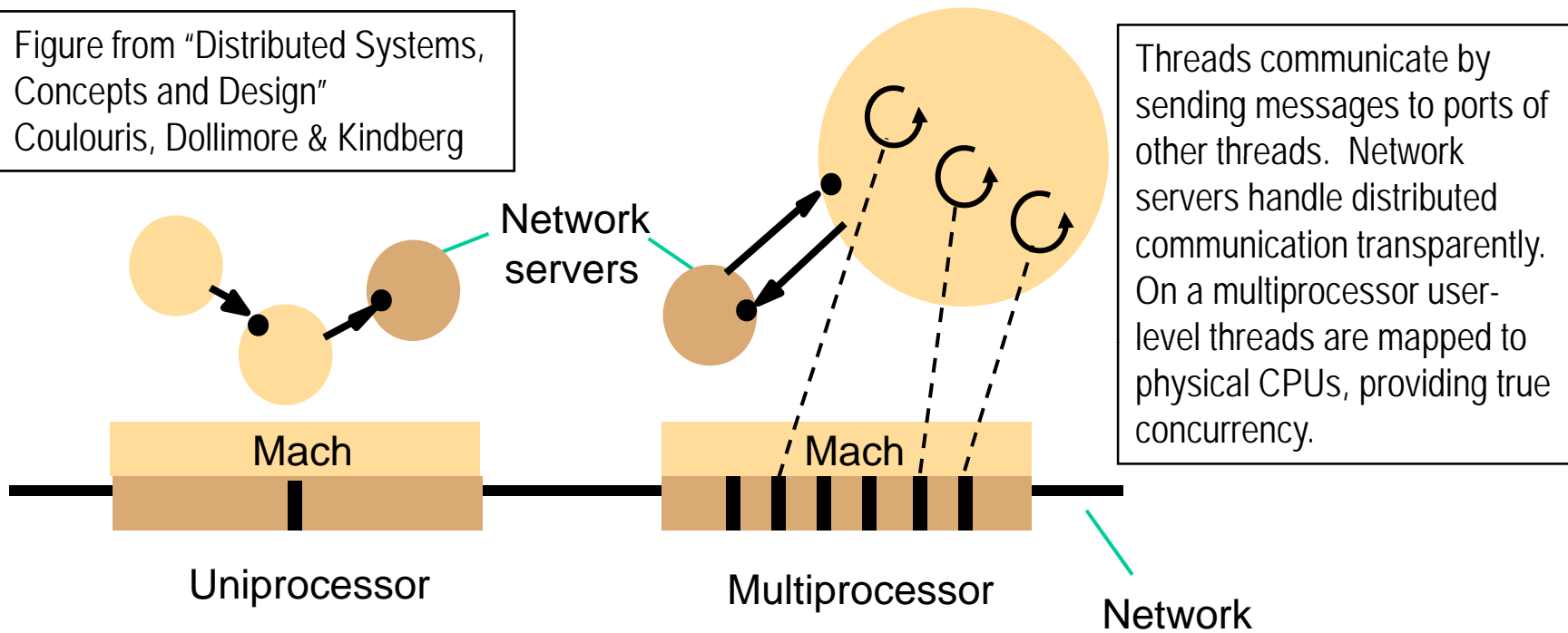
- Design philosophy
  - Small privileged kernel provides core function
  - Most OS services provided by user-level servers
- Promise
  - Less complex kernel → more robust, maintainable
    - Dramatically less privileged code
    - Hw-enforced interfaces between modules
  - Flexibility, customizability, extensibility
  - Natural base for distributed systems
- Mach was a typical example

# Key Mach Abstractions

- **Tasks/threads**
  - Tasks are passive (address space + resources)
  - Threads are active, perform computation
- **Ports**
  - Message origin / destination
  - Have access rights (embodied as capabilities)
  - Essentially an object reference mechanism
- **Messages**
  - Basis of all communication in Mach
- **Devices**
- **Memory objects and memory cache objects**

# Tasks, threads and communication

Figure from "Distributed Systems, Concepts and Design"  
Coulouris, Dollimore & Kindberg



Threads communicate by sending messages to ports of other threads. Network servers handle distributed communication transparently. On a multiprocessor user-level threads are mapped to physical CPUs, providing true concurrency.

Key:



Port

Task



Thread



Processor



Thread mapping

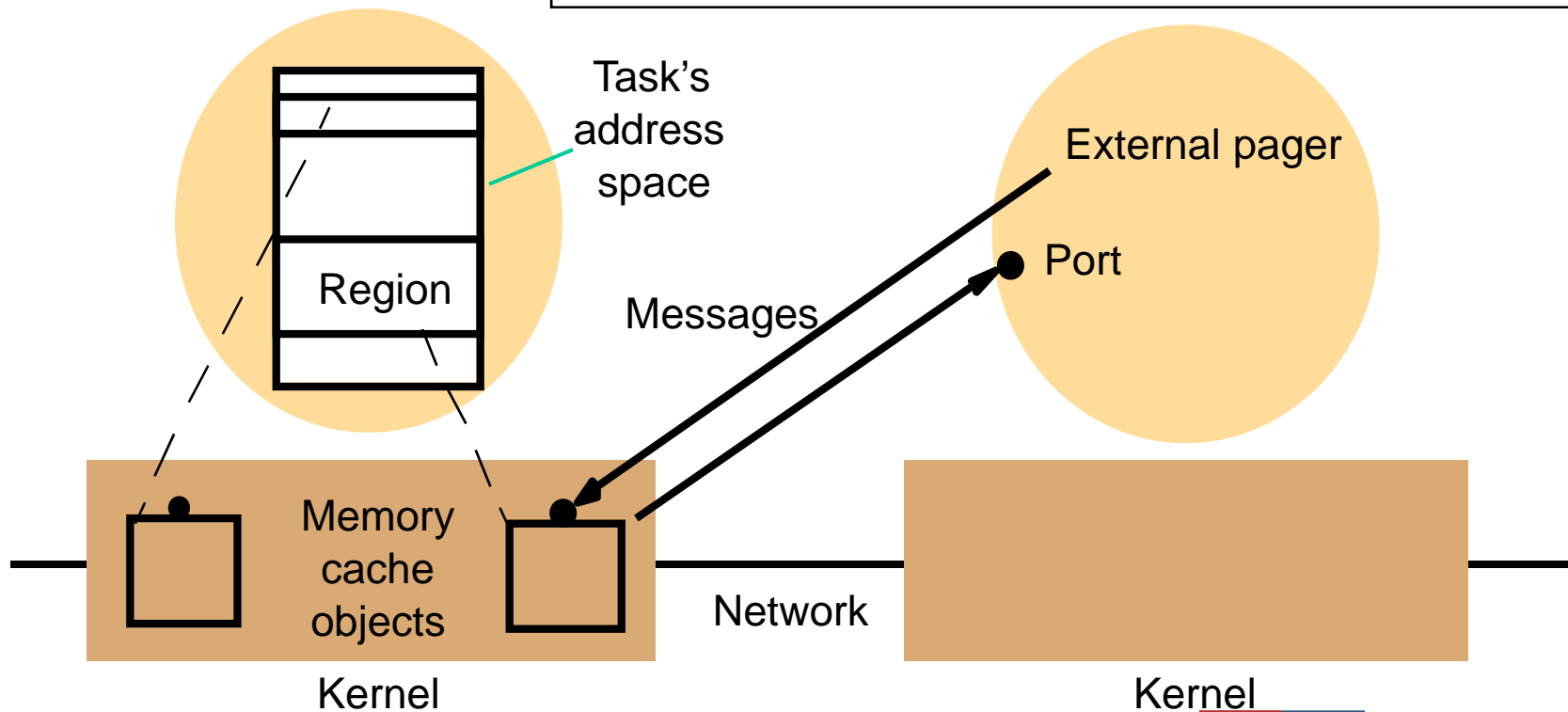


Communications

# Mach External pager

Figure from "Distributed Systems, Concepts and Design"  
Coulouris, Dollimore & Kindberg

Address space maps memory objects; microkernel maintains cache of memory object contents in physical memory while a user-level pager manages the backing store for each object. External pager may be on same, or different machine.



# IPC Costs

- First generation microkernels were slow
  - Mach, Chorus, Amoeba
  - 100 microsecs IPC (almost independent of CPU clock speed!)
  - Many concluded this was inherent limitation of microkernel approach
- Second generation microkernels tackled IPC performance head on
  - L4 (Jochen Liedtke @ Karlsruhe, Gernot Heiser @ UNSW)
  - 20 times faster than Mach on same hardware

# Example of IPC Performance

- "Improving IPC by Kernel Design" by J. Liedtke, Proceedings of the 14th SOSP, December 1993.
- L3 is a micro-kernel, the predecessor to L4

IPC  
Time  
(tics are  
50 us  
Increments)

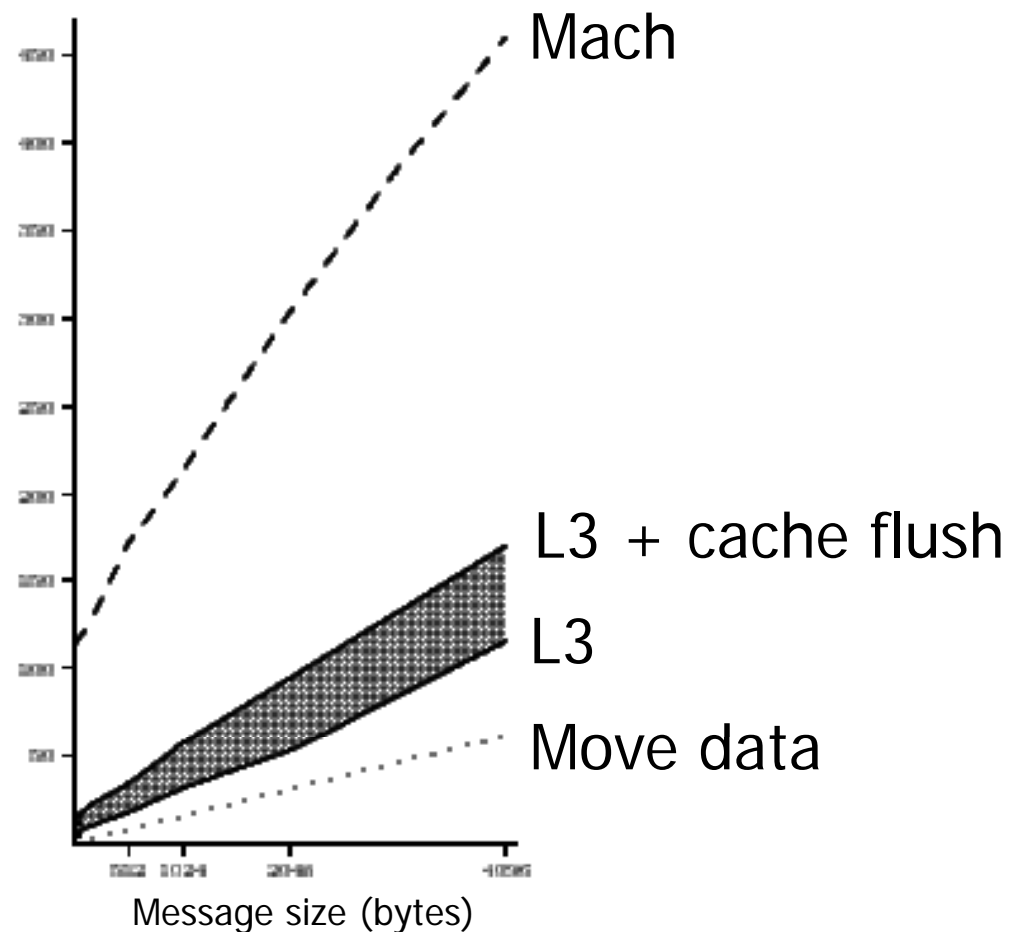


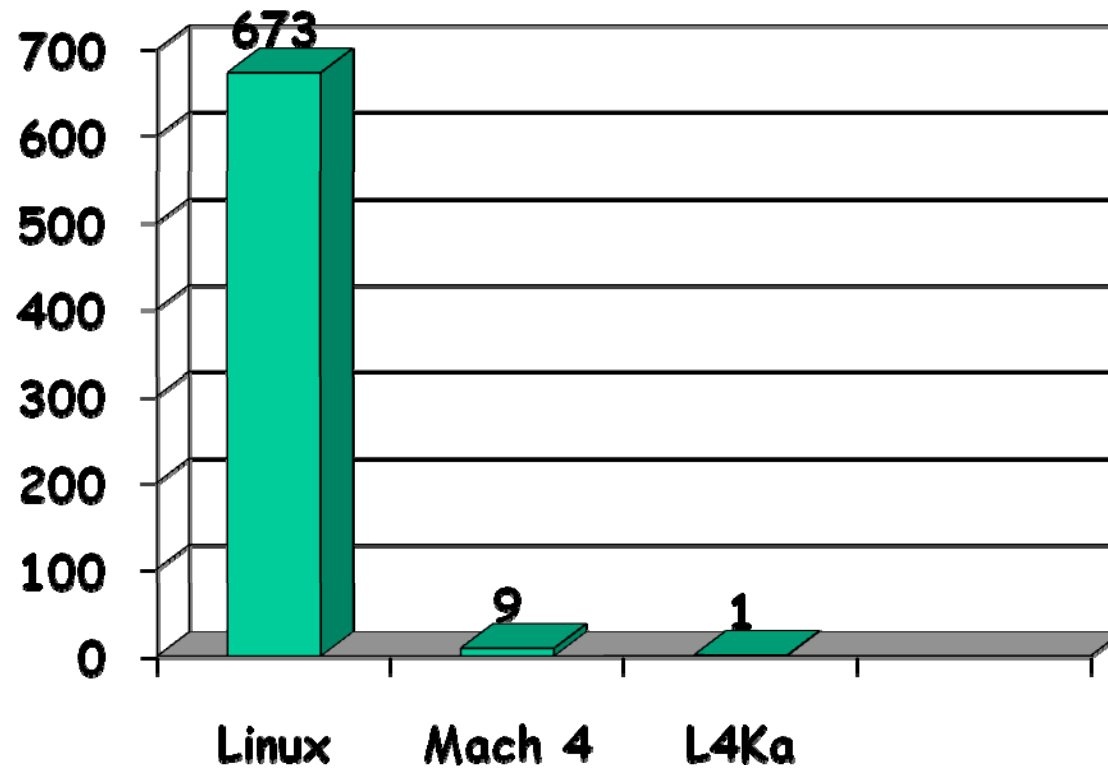
Figure 8: 486-DX50, L3 versus Mach Ipc Times

# Why the difference?

- First generation poorly designed (Liedtke)
  - Complex API
  - Too many features
  - Large cache footprint → memory bw limited
- L4 is fast due to small cache footprint
  - 10-14 I-cache lines
  - 8 D-cache lines
  - Small cache footprint → CPU limited
  - L4 + user-level Linux server 5-7% slower than native Linux

# Size Comparison

- Lines of code (x 10,000)



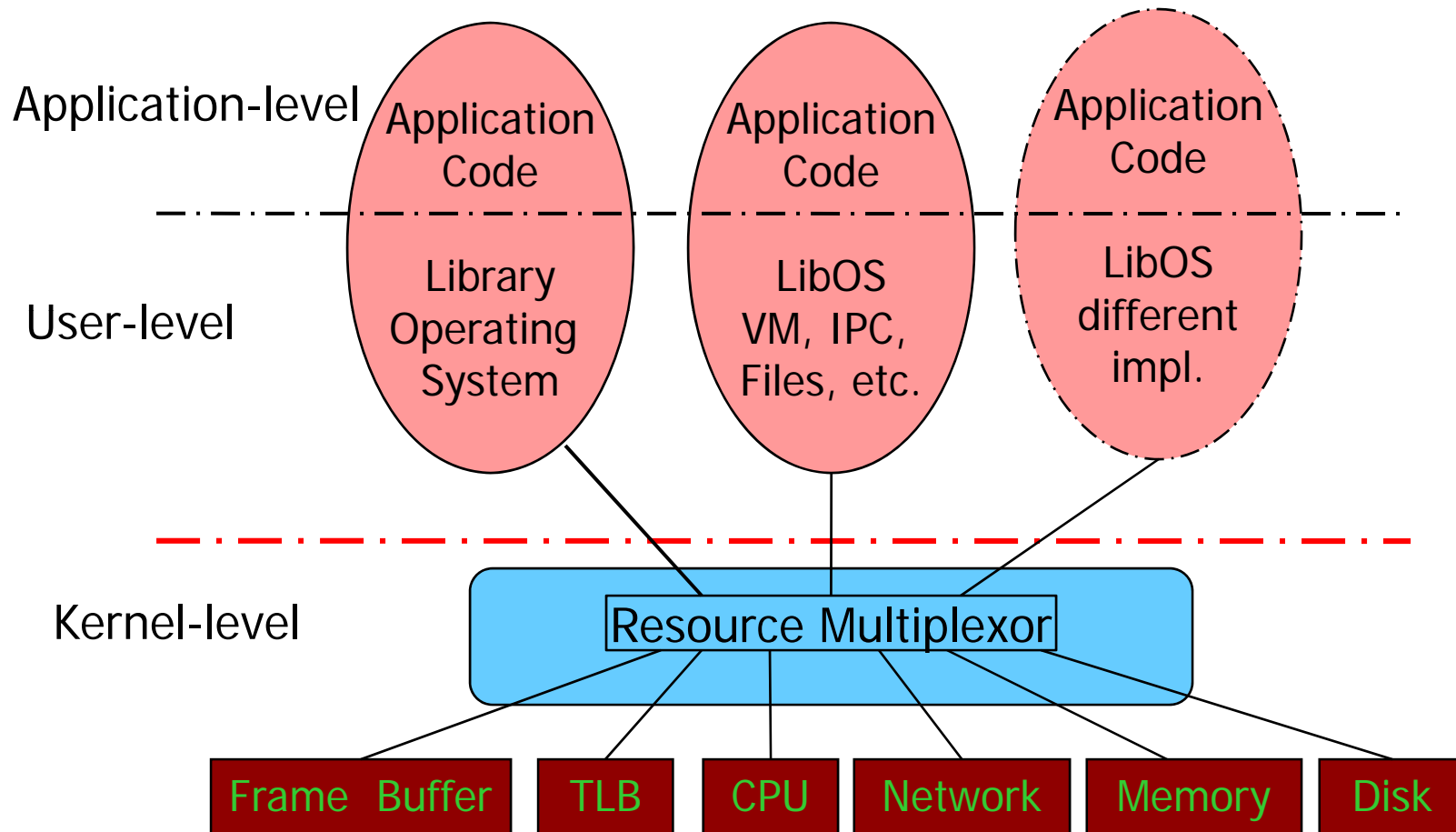
# L4 Abstractions & Mechanisms

- Two basic abstractions (in latest version)
  - Address spaces - unit of protection
    - initially empty
    - Populated by privileged mapping operating
  - Threads - unit of execution
    - Kernel-scheduled, user-level managed
- Two basic mechanisms
  - IPC - synchronous message passing
  - Mapping - all access to memory, devices

# How far can we take this?

- Microkernels: minimal set of abstractions and mechanisms
- Exokernel: MIT Research project
  - Claim: OS abstractions are bad
    - Deny application-specific optimizations
    - Discourage innovation
    - Impose "mandatory costs"
  - Soln: Separate concept of protection from abstraction and management
- Exokernel is a *resource multiplexor*

# Exokernel Architecture



---

# Exokernel basics

---

- Interface is low-level (expose HW, kernel data structures)
- Fine-grained resource multiplexing (i.e., individual disk blocks, not disk partitions)
- Management is limited to protection
- Revocation of resources is visible to user-level libOS
- Code can be downloaded to exokernel by application

---

## Going farther...

---

- Exokernel drops OS abstractions, multiplexes hardware
- Much like an older strategy... Virtual Machines
  - Place thin layer of software "above" hardware
    - *virtual machine monitor (VMM, hypervisor)*
  - Exports raw hardware interface
  - OS/application above sees "virtual" machine identical to underlying physical machine
  - VMM multiplexes virtual machines

---

# VM Examples

---

- Original - IBM's VM/CMS (1970's)
- Now hot again:
  - Disco (Stanford research, 1997) → VMWare
  - Denali (U. of Washington, 2002)
  - Xen (Cambridge, 2003)
  - Linux KVM (kernel virtual machine, as of 2.6.20, 2007)
- What's the big deal about virtual machines?

# What is a virtual machine?

- An efficient, isolated duplicate of the real machine
  - Popek & Goldberg, 1974 "Formal Requirements for Virtualizable Third Generation Architectures"
  - Provide by "virtual machine monitor" with three essential characteristics:
    - Essentially identical execution environment (as real machine)
    - Minor performance penalty for programs in VM
    - VMM has complete control over system resources
- Software added to the execution platform to give the appearance of a *different* platform or *multiple* platforms
  - Smith & Nair, 2004 "Virtual Machines"

# Why virtual machines?

- Original motivation in 1960's
  - Large, expensive computers shared by many users
  - Different groups wanted or needed different operating systems
  - Convenient timesharing mechanism (each user gets own virtual machine)
- Today's motivation?
  - Large scale servers have similar issues as original motivation
  - Portability/compatibility
    - Avoid dealing with multiprocessor issues in OS
  - Security
  - Reliability/fault tolerance
  - Migration
  - Performance
  - Innovation

# Types of virtual machines

- Many uses of the term “virtual machine”
- Conventional software is developed/compiled for a specific OS and instruction set architecture (ISA)
  - Together, these are the *application binary interface* (ABI)
  - Can distinguish virtual machines depending on whether they virtualize the ABI or the ISA.
- *Process virtual machines* provide virtual ABI
  - Created and destroyed along with the process they run
- *System virtual machines* provide a complete system environment
  - Multiple user processes, file system, I/O, GUI, etc.

# Smith & Nair's Taxonomy

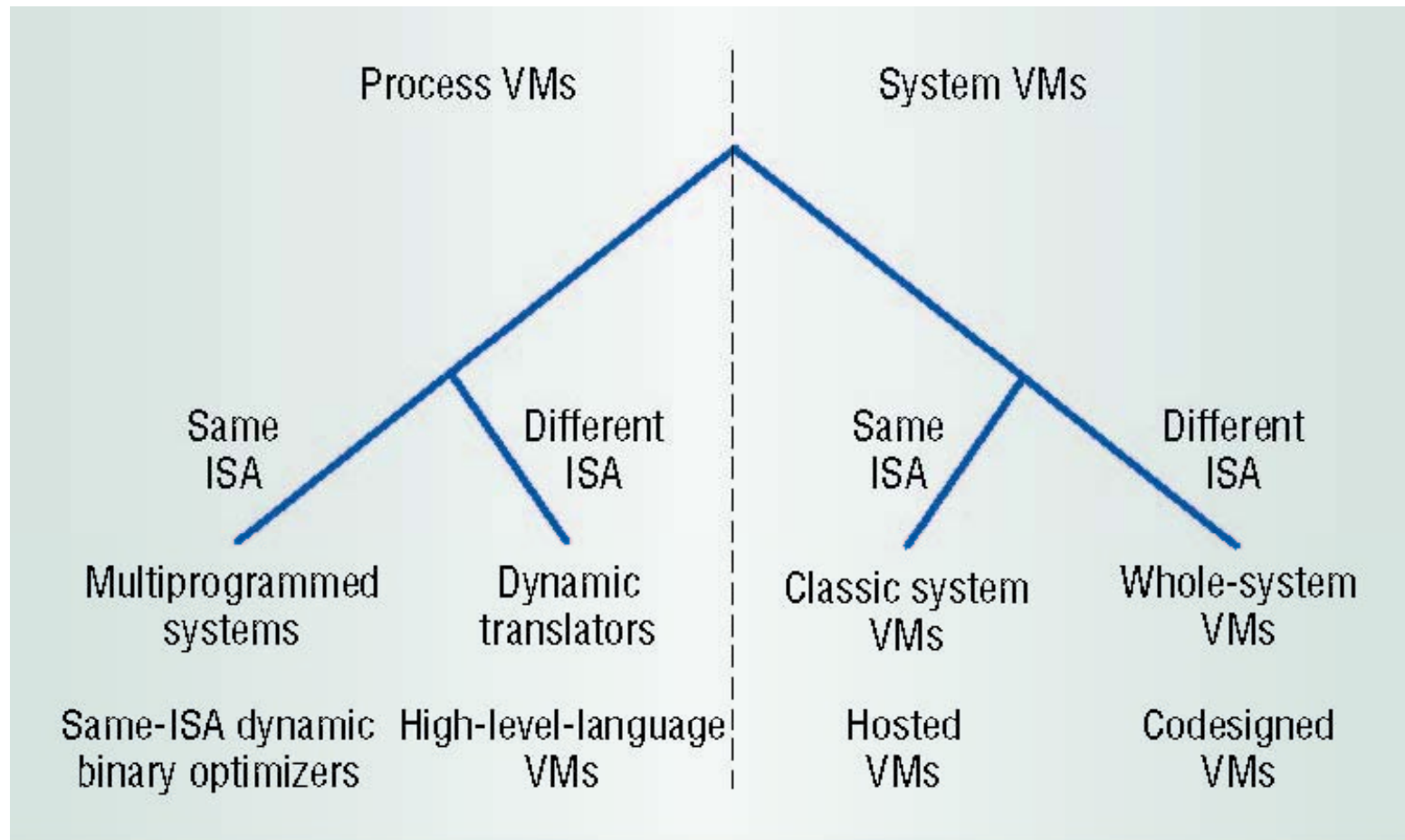


Image from: **The architecture of virtual machines**, J.E. Smith and Ravi Nair;  
IEEE Computer, Volume 38, Issue 5, May 2005 Page(s):32 - 38

# Process Virtual Machines

- Multiprogramming
  - Each conventional process has illusion of own machine
    - Address space, CPU, file table, etc
- Emulation / dynamic binary translators
  - Code compiled for one ISA translated on-the-fly to host ISA
    - E.g. Digital FX!32 runs x86 Windows binaries on Alpha
- Dynamic optimizers
  - Same guest/host ISA, only purpose is optimization
- High-level language VMs
  - Designed together with language
  - Mainly for portability & to support language features
    - E.g. Pascal P-code, Java bytecode

# System VMs

- “classic” VMM
  - VMM runs on bare hardware, everything else runs on top
  - VMM is most privileged software, everything else less
- “hosted” VM
  - Virtualizing software installed on top of existing OS
    - E.g. VMWare Workstation

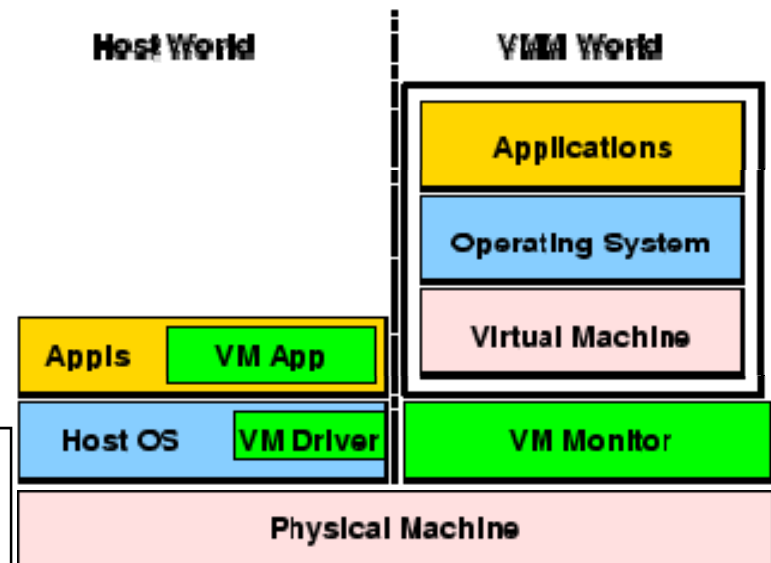
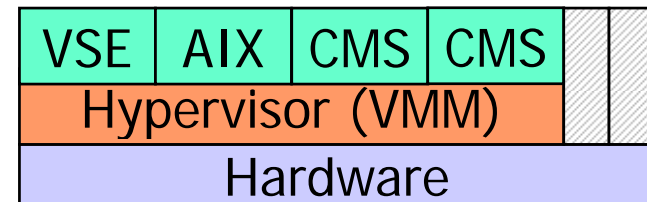


Image from: “Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor”, J. Sugerman et al., Usenix 2001.

# Requirements for Virtualizability

- Architecture requirements
  - Dual mode operation
  - A way to call privileged operations from non-privileged mode
  - Memory relocation / protection hardware
  - Asynchronous interrupts for I/O to communicate with CPU
    - Goldberg, 1972
- Generic VM operation / implementation
  - Dispatcher component
  - Allocator
  - Interpreter

# Instruction Requirements

- *Privileged instructions*: required to trap if not executed in supervisor mode
- *Sensitive instructions*: affect the operation of the system in some way
- THEOREM: An efficient VMM may be constructed if the set of sensitive instructions is a subset of the set of privileged instructions
- Intel Pentium: 17 instructions are sensitive but not privileged (Robin & Irvine, USENIX Security 2000)
  - VMware does binary rewriting to deal with this
  - Xen requires changes to the OS → *paravirtualization*
  - Intel VT, AMD-V (Pacifica) fix this

# VM Performance

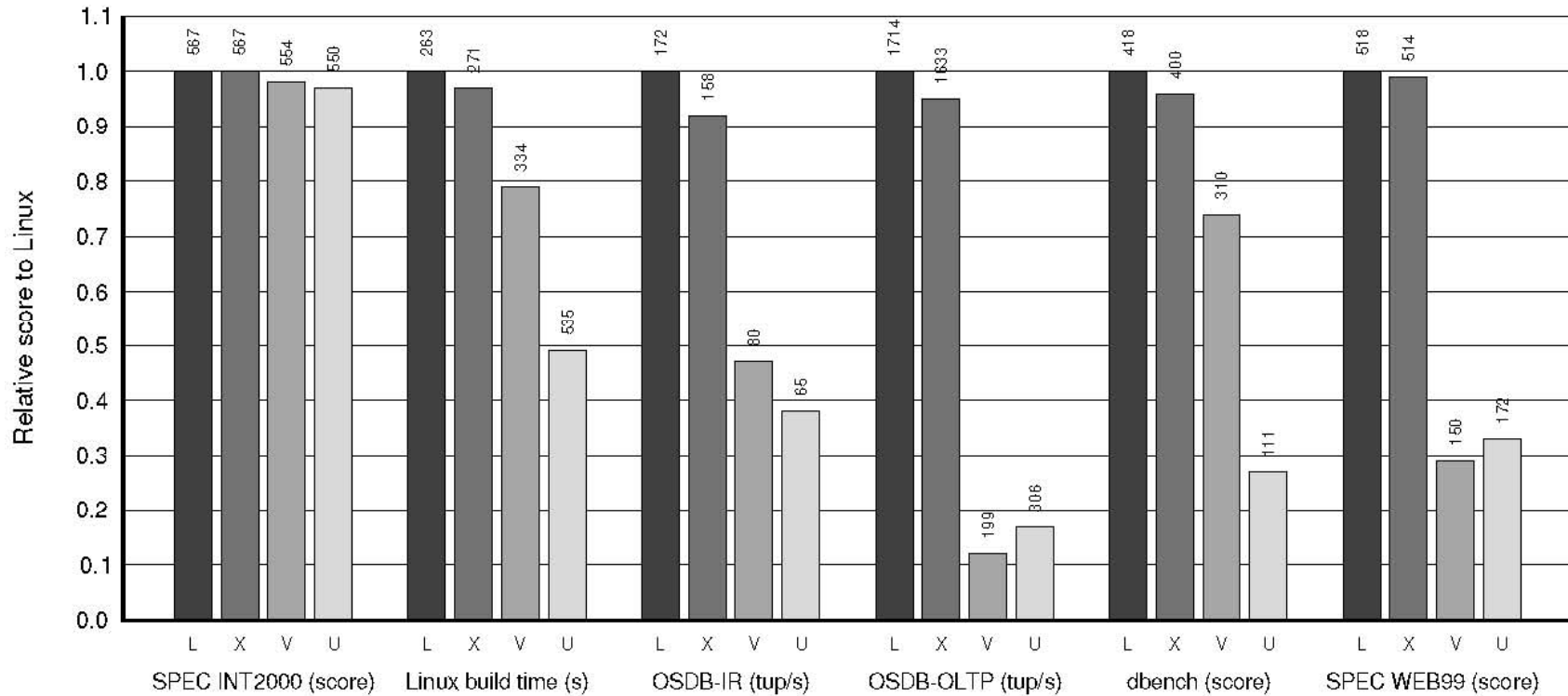


Figure 3: Relative performance of native Linux (L), XenLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

From: "Xen and the art of virtualization" Barham et al

---

# OS Extensions

---

- Adding new function to OS "on the fly"
- Why?
  - Fixing mistakes
  - Supporting new features or hardware
  - Efficiency / Custom implementations
- How?
  - Give everyone their own machine (VMs)
  - Allow some OS function to run outside (ukernel)
  - Allow users to modify the OS (modules)

# Loadable Kernel Modules

- Giving everyone a virtual machine doesn't entirely solve the extension problem
  - You can run what you want on your VM, but do you really want to write a custom OS?
- Often just want to modify/replace small part
- Solution: Allow parts of the kernel to be dynamically loaded / unloaded
  - Requires dynamic relocation and linking
- Common strategy in monolithic kernels for device drivers (FreeBSD, Windows NT/2K/XP, Linux)

# Linux Loadable Kernel Modules

- Module writer must define (at least) two functions
  - *init\_module* - code executed when module loads
  - *cleanup\_module* - code executed when module unloads
  - Module functions can refer to any exported kernel symbols
- Module is compiled into relocatable `.o` file (2.4) or `.ko` file (2.6)
- *insmod* command loads module into running kernel
  - 2.4 - *insmod* resolves references to kernel symbols
  - 2.6 - kernel does the linking
- *rmmmod* command removes module from kernel
- *lsmod* command lists currently-installed modules

# insmod - 2.4 kernel

- User-level command (program) restricted to superuser
- Gets help from some special system calls
  - *sys\_create\_module* - allocate kernel memory to hold module
  - *get\_kernel\_syms* - get kernel symbol table to link module (patch symbolic references in .o file to actual kernel addresses)
  - *sys\_init\_module* - copy relocatable .o file into kernel space
- Then calls *init\_module* function
- insmod is trivial for 2.6 kernel

---

# rmmod

---

- Unlinks module from kernel
- Needs to ensure no one is using module first!
  - Reference count incremented whenever module is used, or a module that depends on this one is loaded
- Removes module symbols from symbol table
- Frees memory
- Getting module unloading right is tricky

# Problems with module approach

- Requires stable interfaces
  - Linux uses version numbers to check if module is compiled for correct version of kernel, but it is easy to get this wrong
- Unsafe
  - Module code can do *anything* because it runs privileged
    - E.g. recall VMWare Workstation driver?
      - "hijacks" machine by changing *interrupt descriptor table (IDT)* base register and then jumps to code in the VM application!

# Alternate kernel-level schemes

- Trusted compiler (or certification authority) + digital signatures
  - Allows verification of source of code added to kernel
  - You still have to decide if you trust that source
  - Code can still do anything
- Proof-carrying code
  - Consumer (OS) supplies a specification for what extensions are allowed to do
  - Extension must supply a proof that it is safe to execute according to specification
  - OS validates proof
  - Proof should be easy to check, but may be hard to generate (e.g. maze example)

# Alternates (2)

- Sandboxing (software fault isolation)
  - Limit memory references to per-module segments
  - Check for certain unsafe instructions
- Examples:
  - SPIN (U. of Washington)
    - Modula-3 + trusted compiler
    - Safety properties provided by language
    - Problems with dynamic behavior (e.g. "while(1)")
  - Vino (Harvard)
    - Sandboxed C/C++ code called "grafts"
    - Timeouts to guard against misbehaved grafts
    - Resource limits + transactional "undo"