Week 2: OS Structure

Layered systems, open systems, microkernels, exokernels, virtual machines & modules

CSC 469 / CSC 2208

Fall 2018

Angela Demke Brown

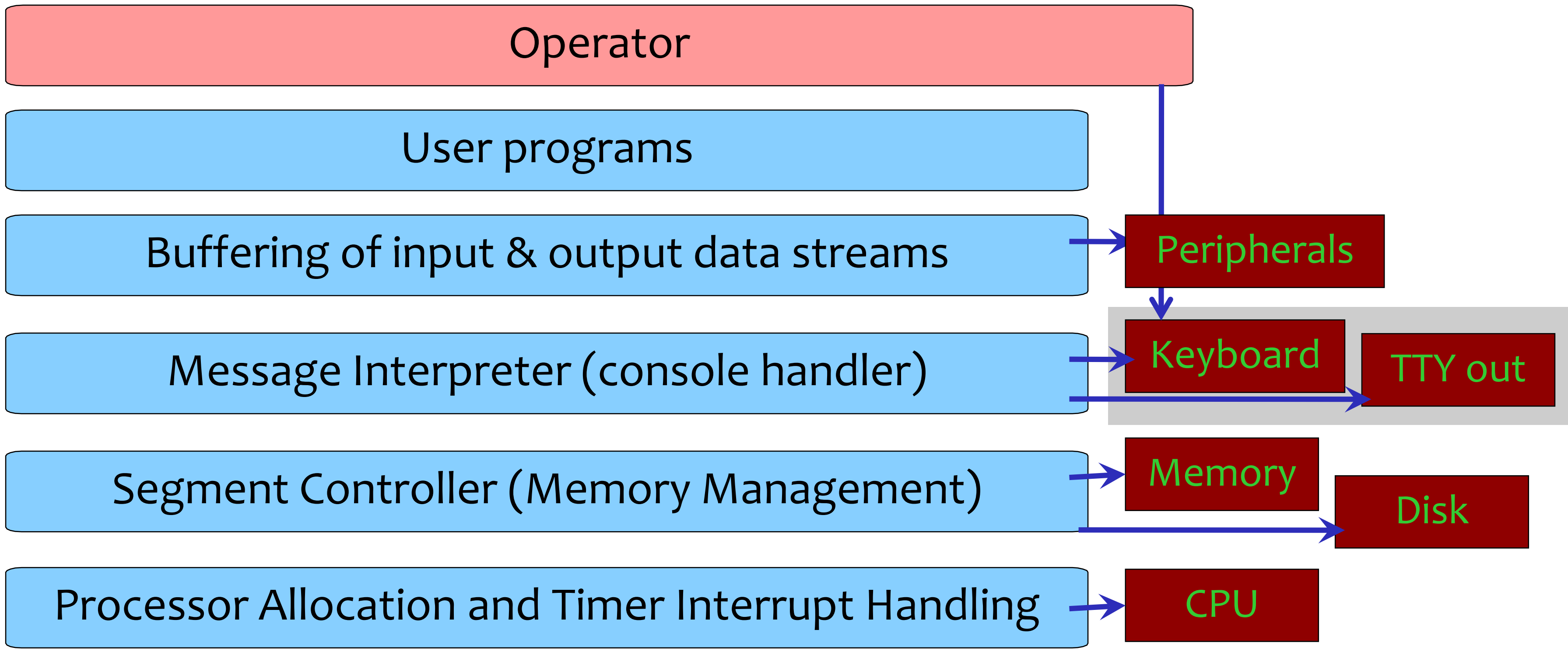
University of Toronto, Department of Computer Science

Overview

- Kernel structures
  - Layered systems
  - Open systems
  - Monolithic kernels
  - Microkernels
  - Kernel Extensions
  - Virtual Machines

Early Layered System: THE

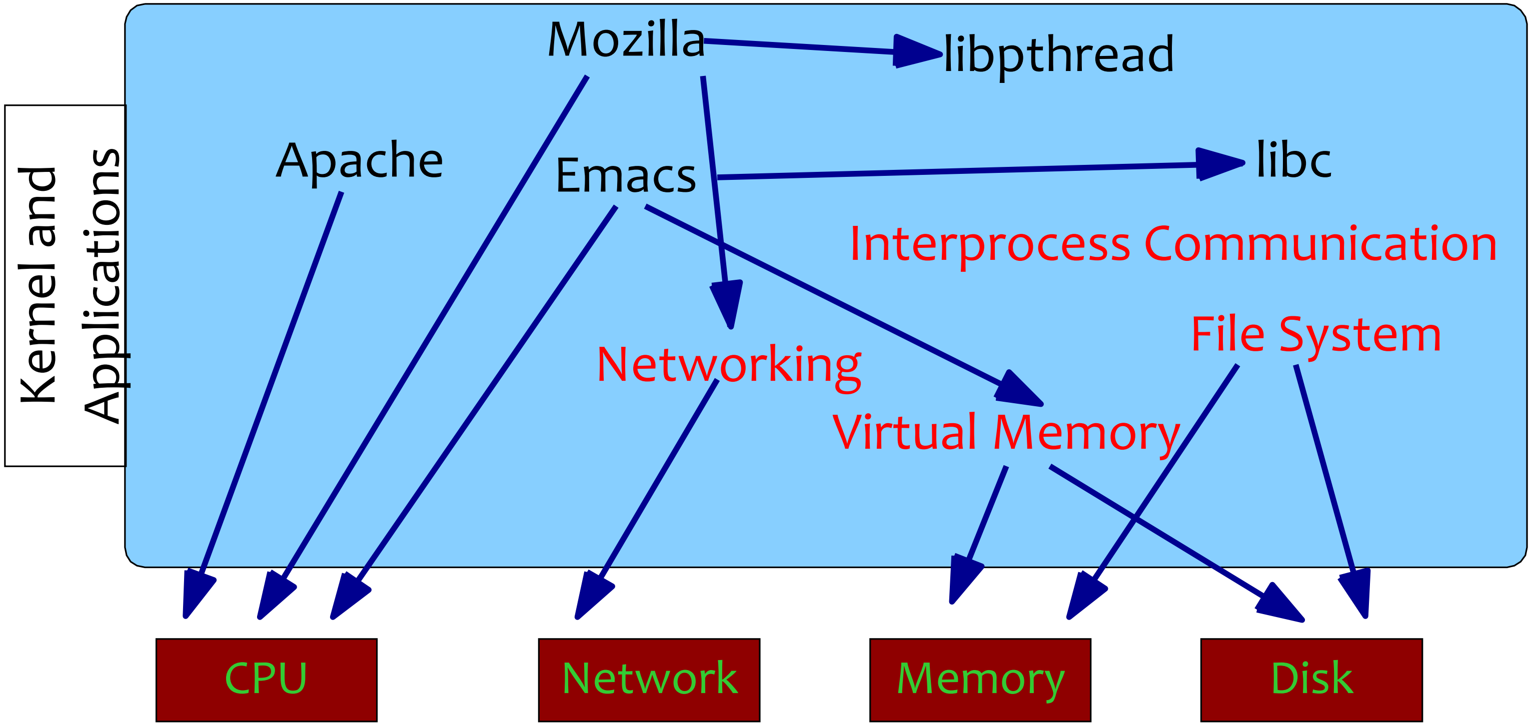
- Djikstra, 1st SOSP, 1967



Properties of Layered Systems

- Each layer has well-defined function and interface to layer above/below
  - Provides easier-to-use abstraction for higher layers
- Other examples: MULTICS (rings)
- Advantages?
  - Each layer can be designed, implemented and tested independently
  - Processes at any level can only invoke services of level below → no circular wait → no deadlock
- Disadvantages?
  - Hard to partition functions into this strict hierarchy (why is console below other peripherals?)

Open Systems



Properties of Open Systems

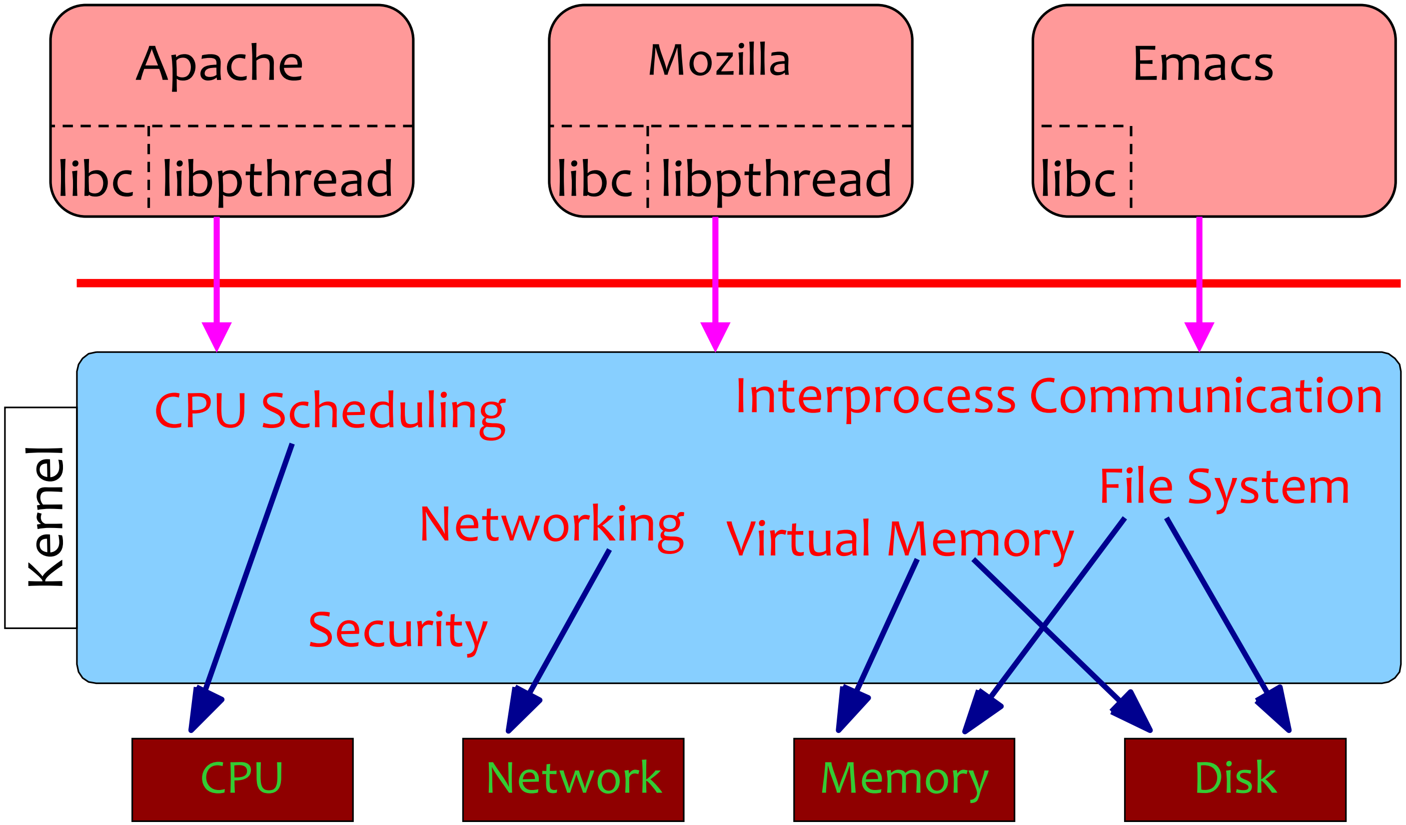
- Applications, libraries, kernel all in the same address space
- Crazy?
  - Idea first described by Lampson & Sproull, 7^{th} SOSP, 1979 “An open operating system for a single-user machine”
  - MS-DOS; Mac OS 9 and earlier; Windows ME, 98, 95, 3.1
  - Palm OS and some embedded systems
- Used to be *very* common
- Advantages?
  - *Very* good performance, very extensible, works well for single-user
- Disadvantages?
  - No protection between kernel and/or apps, not very stable, composing extensions can lead to unpredictable behavior

Properties of Open Systems

- Why is Windows 95/98/ME classified as “open”?
  - 32-bit applications have own address space BUT
  - 16-bit Win, DOS apps, and dll’s share 1GB space
    - Including key system dll’s (kernel32.dll)

- Next up: monolithic OSs and microkernels…

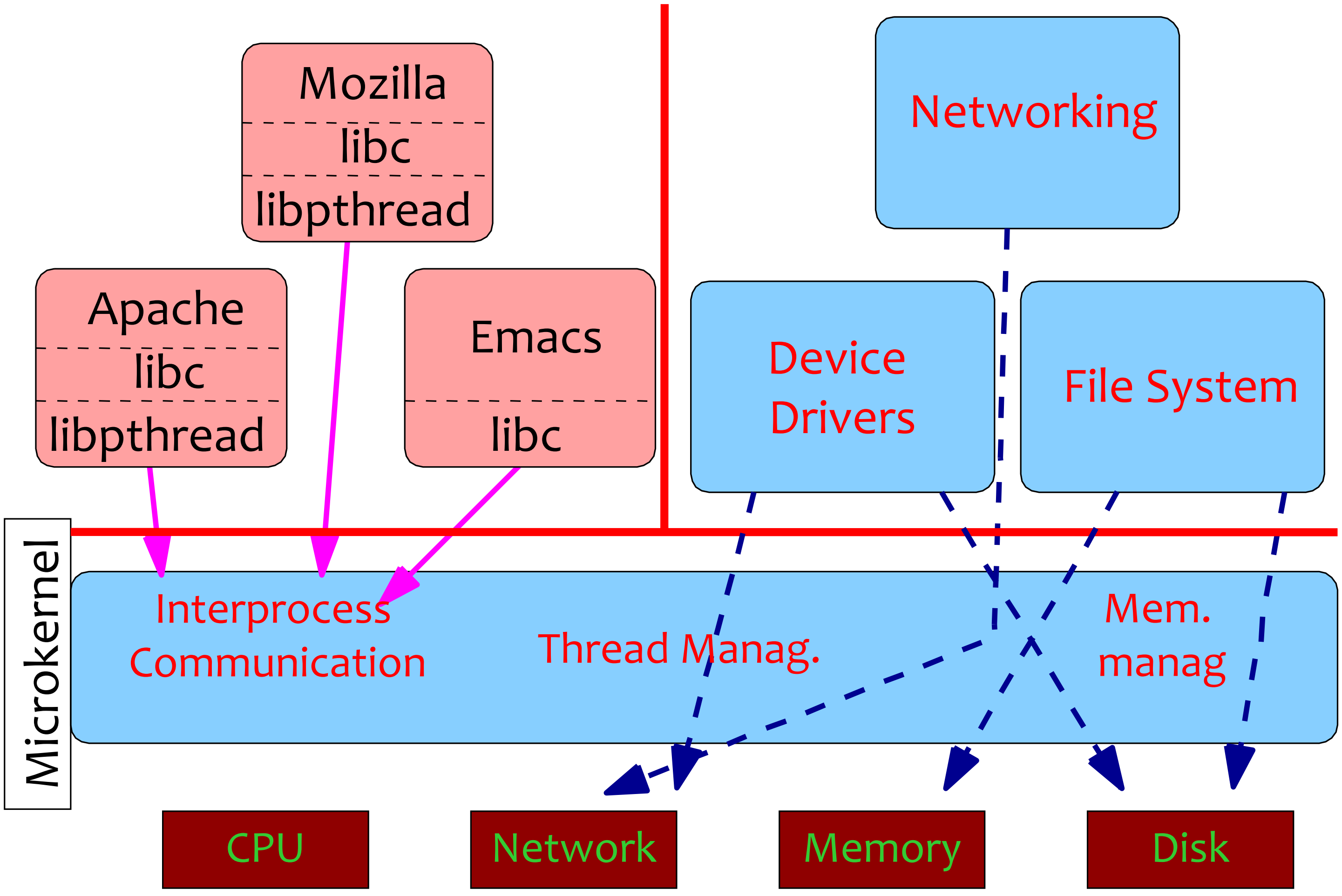
Monolithic OS



Properties of Monolithic Kernels

- OS is all in one place, below the “red line”
- Applications use a well-defined system call interface to interact with kernel
- Examples: Unix, Windows NT/XP, Linux, BSD, OS/161
  - Common in commercial systems
- Advantages?
  - Good performance, well-understood, easy for kernel developers, high level of protection between applications
- Disadvantages?
  - No protection between kernel components, not (safely, easily) extensible, overall structure becomes complicated (no clear boundaries between modules)

Microkernel OS



Properties of Microkernels

- Design Philosophy: protected kernel code provides minimal "small, clean, logical" set of abstractions
  - Tasks and threads
  - Virtual memory
  - Interprocess communication
- Everything else is a *server process* running at user-level

- Early examples: Nucleus (1970),
- Later examples: Mach, Chorus, QNX, L4, GNU Hurd
- Mixed results …

Microkernel Advantages

- Extensible: add a new server to add new OS functionality
- Kernel does not determine operating system environment
  - Allows support for multiple OS *personalities*
  - Need an emulation server for each system (e.g. Mac, Windows, Unix)
  - All applications run on same microkernel
  - Applications can use customized OS (e.g. for databases)

More Advantages

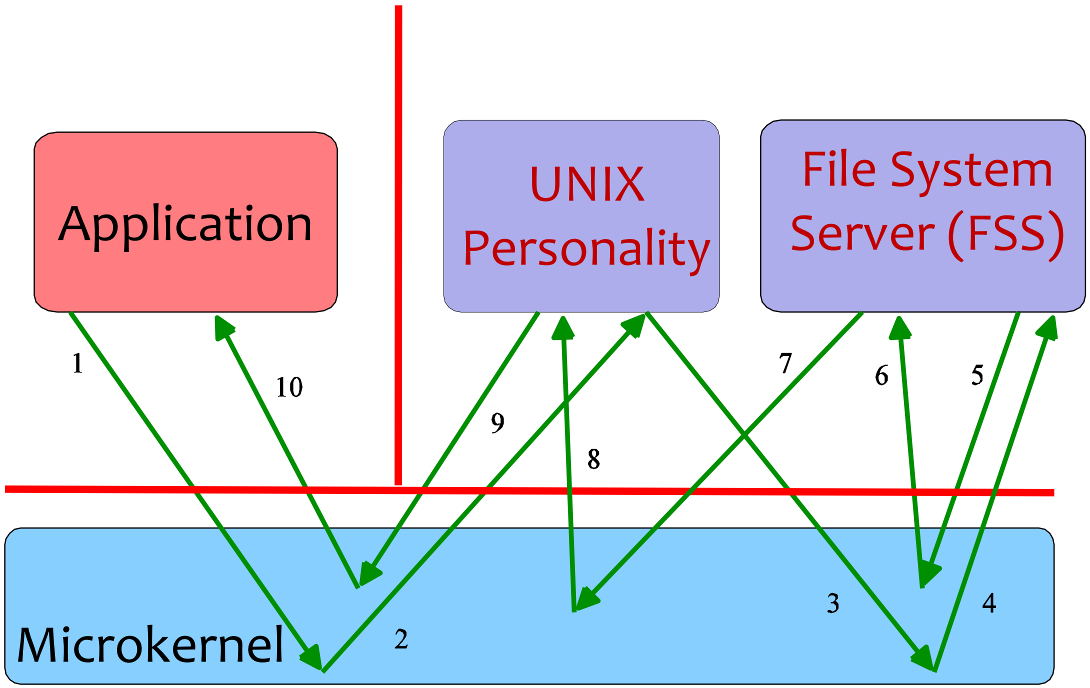
- Mostly hardware agnostic
  - Threads, IPC, some user-level servers don’t need to worry about underlying hardware
- Strong protection
  - Even of the OS against itself (i.e., the parts of the OS that are implemented as servers)
- Easy extension to multiprocessor and distributed systems

Microkernel Disadvantages

- Performance
  - System calls can require a lot of protection mode changes (next slide)
- Expensive to reimplement everything with a new model
  - OS personalities are easier to port to new hardware *after* porting to microkernel, but porting to microkernel may be *harder* than porting to new hardware
- Bad past history
  - See IBM Workplace OS story

Microkernel System Call Example

1. Application calls read(), traps to microkernel
2. microkernel sends message to Unix Personality requesting read
3. Unix personality sends message to File System Server (FSS) asking for data
4. FSS receives message and begins processing
5. FSS sends message to microkernel asking for disk blocks
6. Microkernel sends data back to FSS
7. FSS sends message to UNIX Personality with results
8. Unix Personality receives message with data
9. Unix Personality sends data to Application
10. Application receives data



As we will see, IPC performance is critical for microkernels.

The Mach Microkernel

- CMU Research Project
- IPC abstraction was seen as a general means for passing info between programs (as opposed to pipes, widely used at the time)
- The Plan:
  - Step 1: Proof of Concept
    - Take BSD 4.3 and “fix” VM, threads, IPC
  - Step 2: Microkernel and “single-server” Unix emulation
    - Take Unix kernel and “saw it in half”
  - Step 3: Microkernel and multiple servers (for FS, paging, network, etc.)
    - Servers glued together by modules that catch system calls

Mach

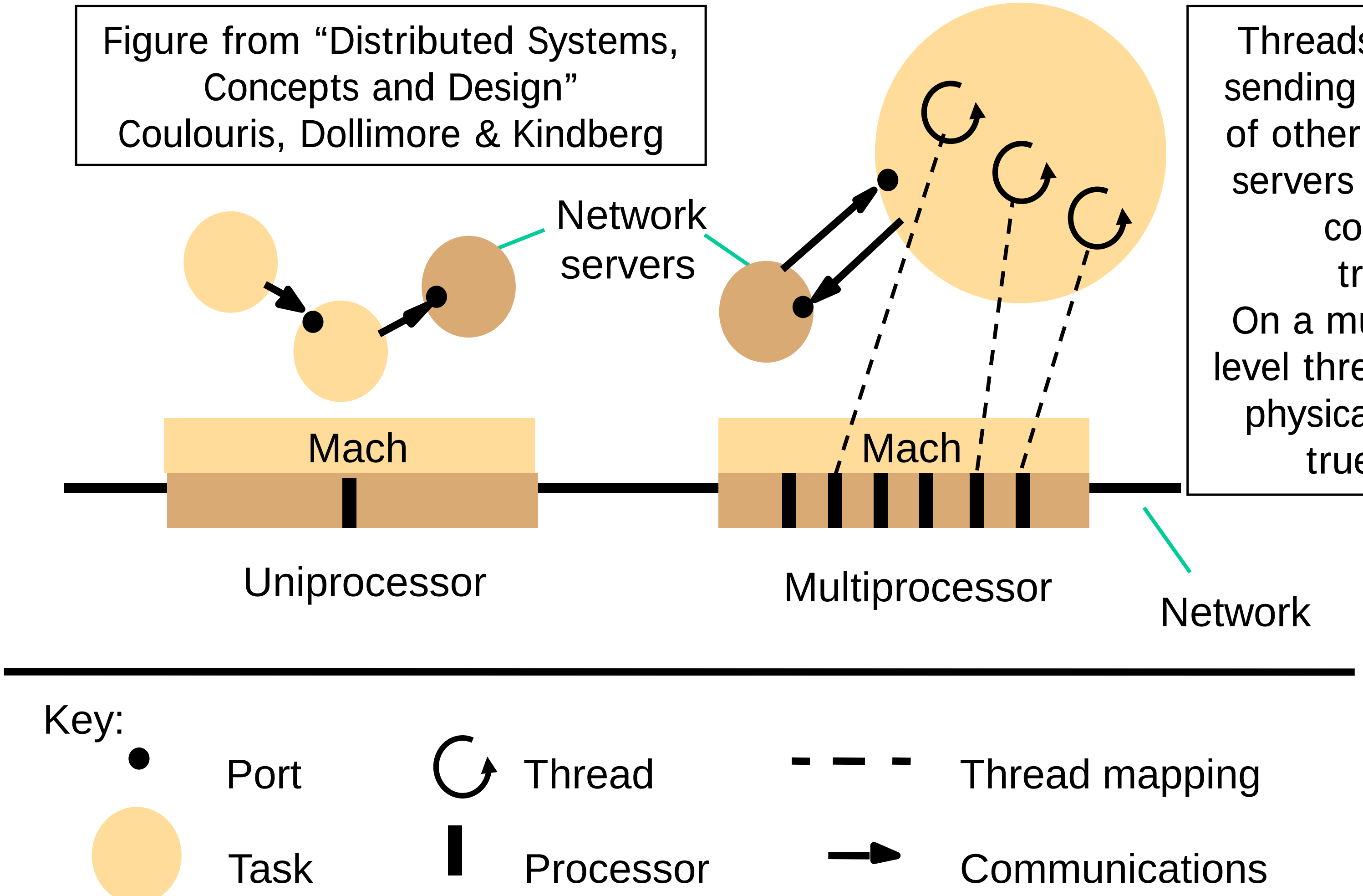
- Reality:
  - Proof of concept completed in 1989
    - Unix server, SMP support, process abstraction into tasks + threads
    - VM: large sparse address spaces, sharing, COW, mmapped files
    - Capability-based IPC, language support for RPC between tasks, etc.
    - Commercial deployment: Encore Multimax, Convex Exemplar, OSF/1, NeXT/NeXTSTEP (and eventually to OS X, iOS)
  - Microkernel and single-server completed and deployed to 10's of machines
  - Multi-server never fully completed
  - Hugely influential

Key Mach Abstractions

- 1+2. Tasks + threads
  - Tasks are passive (address space + resources)
  - Threads are active, perform computation
- 3. Ports
  - Communication channel, message queue
  - Queue has access rights (embodied as capabilities)
  - Send/Receive operations
  - Essentially an object reference mechanism
- 4. Messages
  - Collections of typed data objects
  - Basis of all communication in Mach

Tasks, threads and communication

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

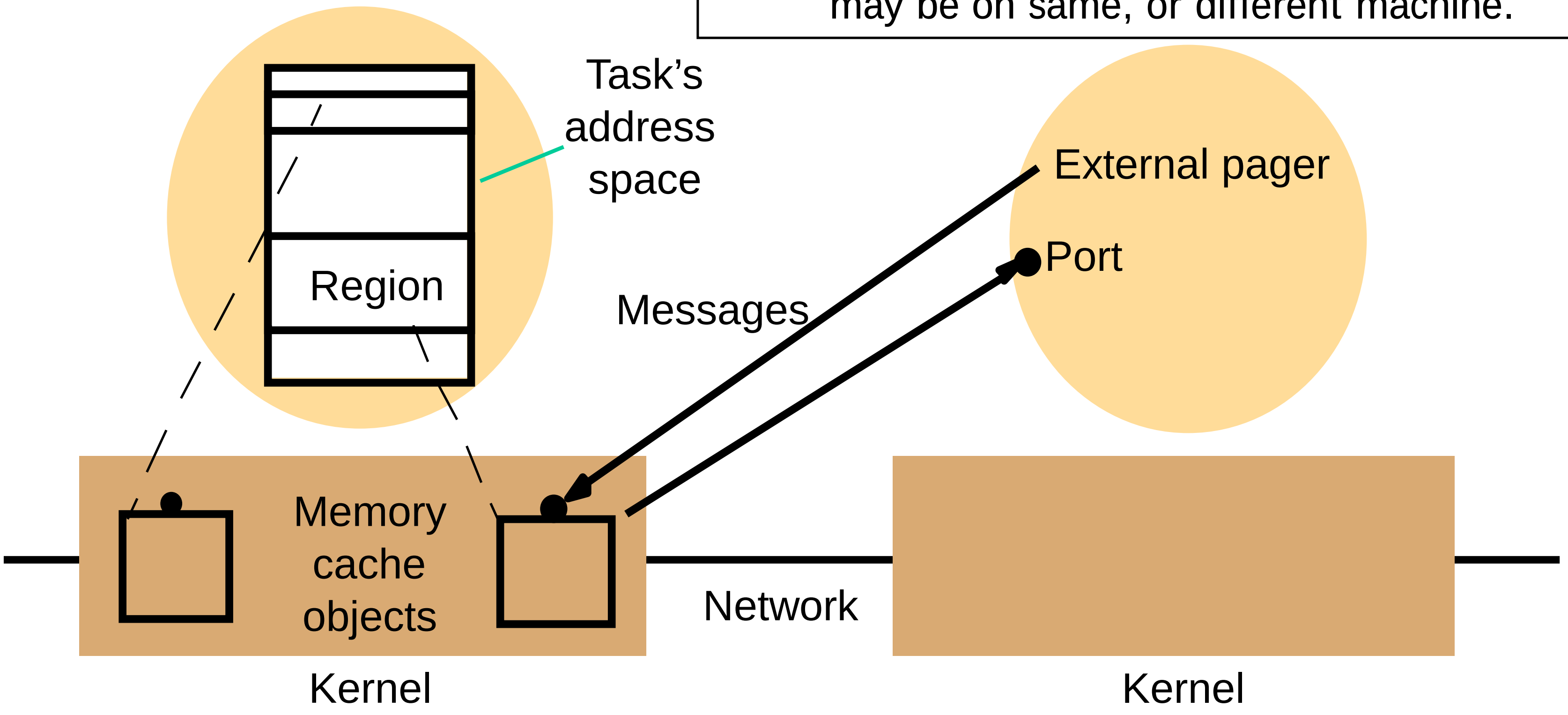


Threads communicate by sending messages to ports of other threads. Network servers handle distributed communication transparently.
On a multiprocessor user-level threads are mapped to physical CPUs, providing true concurrency.

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 problem

- Mach's IPC is too complex, especially buffering IPC
- Re-assessing the whole microkernel concept
  - Some developers proceeded to move critical components (file systems, drivers) back into the kernel
  - Somewhat helps performance but violates idea of microkernel
- Bottleneck analysis indicated too large working set
  - Too many cache misses => idea that efficient microkernel should be small enough to fit performance-critical code in L1 cache
- Idea of thinner IPC layer, and performance-oriented design
  - L3 microkernel: no buffering, very light-weight

IPC problem in a nutshell

- Microkernels are desirable: modularity, security, etc.
- However, drawback: slow IPC
  - Mach: 100-500 usec, depending on message size
  - Ideally, 5-7 usec for small message
- L3 microkernel: "Improving IPC by Kernel Design" by J. Liedtke, Proceedings of the 14th SOSP, December 1993.
  - Predecessor of L4

Objective: Improve IPC performance

- Simple scenario: thread A sends a null message to thread B (both user-level, different addr spaces)
- Minimal sequence of actions:

thread A (user mode):	load id of B set msg length to 0 call kernel
kernel:	access thread B switch stack pointer switch address space load id of A return to user
thread B (user mode):	inspect received msg

- Minimum:
  - 20 instructions, 127 cycles (107 for the 2 instructions to enter/leave kernel)
  - Plus at least 5 TLB misses (9 cycles each) when changing addr space
  - => minimum 172 cycles (3.5 usec) lower bound for IPC
  - They aim at 7 usec, but end up with 5 usec

L3 microkernel – some details

- Tasks, threads, mem obj. /dataspaces, addr. spaces where dataspaces can be mapped
- IPC model: threads communicate via direct messages (strings, memory objects)
  - No communication channels, just global thread IDs and task IDs (unique)
  - A server decides whether action is permitted based on uids
- Hardware interrupts delivered as messages to the right threads
- Clans and chiefs model (Lie ’92)
  - Security: messages stay within a “clan”. Otherwise, redirected to the “chief”

Example of IPC Performance: L3 microkernel

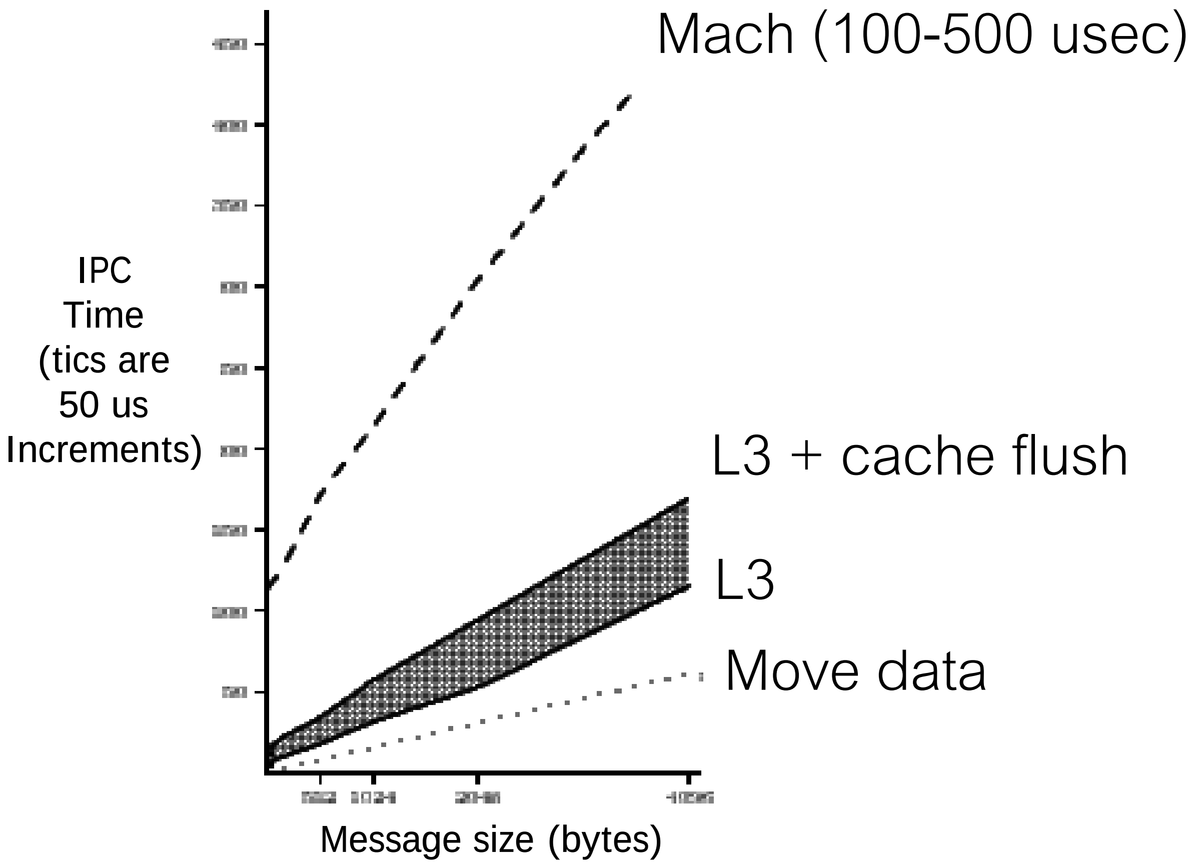


Figure 8: *486-DX50, L3 versus Mach Ipc Times*

More details: Mach-style ports

- L3 operates directly thread-to-thread
- How expensive would it be to introduce buffered Mach-style ports?
  - Use one port link table per address space (kernel-accessible only)
  - User-level: indices identifying the accessed port
  - Access: R/W
  - Illegal accesses are marked in port link table to point to a non-mapped page
- Estimated to 0.6 usec overhead => port-based IPC can also be implemented efficiently

Next step ...

- First generation microkernels (Mach) were not ideal
  - Complex API, Too many features
  - Liedtke concluded that several other Mach concepts can be simplified => L3
- Second generation: L4 microkernel
  - Move more stuff into user-space, rewrite everything in assembly language => 20x performance improvement. Problems?
- Minimality principle: a concept is tolerated inside the microkernel only if moving it outside the kernel would prevent the implementation of the system's required functionality.
  - UNIX: everything is a file
  - Mach: IPC generalizes files
  - L4: Can it be put outside the kernel?

L4 Abstractions & Mechanisms

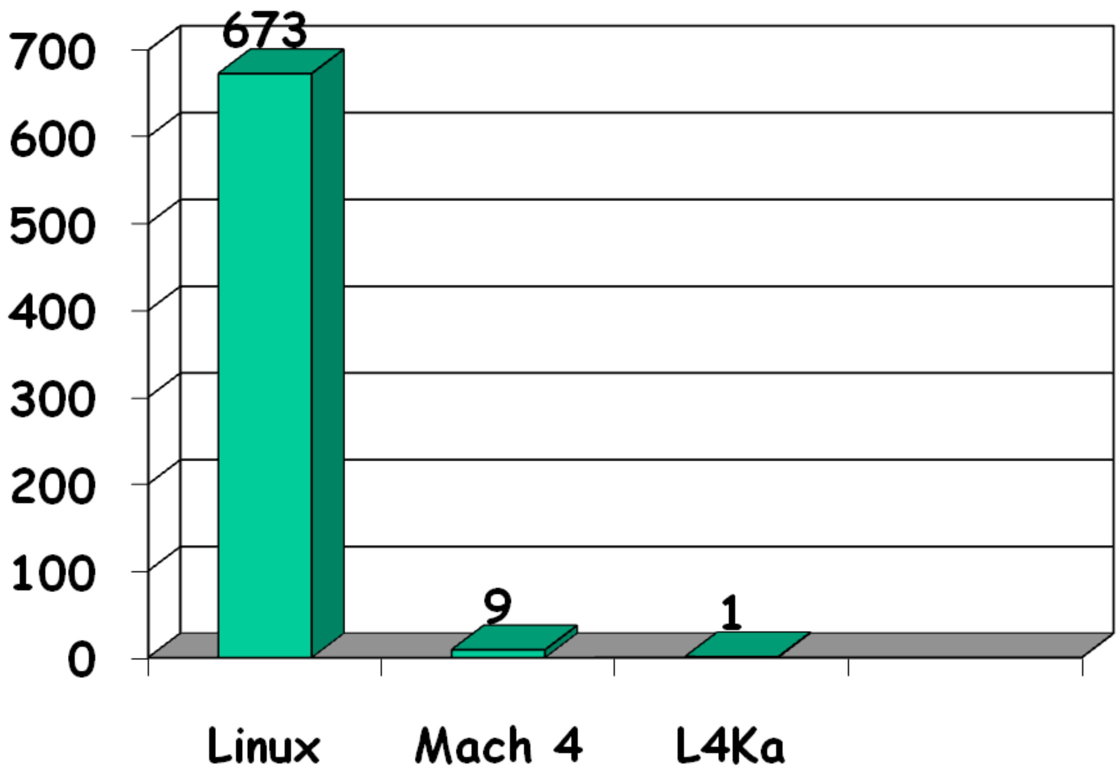
- Two basic abstractions (in latest version)
  - Address spaces – unit of protection and isolation (code, data, resources)
    - Initially empty
    - Populated by privileged mapping operations (map, grant, unmap)
    - Capabilities + memory pages
  - Threads – unit of execution
    - User-code (appl.), or kernel code (syscalls, page faults, interrupts, etc.)
    - Kernel-scheduled, user-level managed
    - Associated to an address space
      - Executes code in 1 task, but can be migrated; can have several threads per task
- Two basic mechanisms
  - IPC – synchronous message passing
  - Mapping – all access to memory, devices

Summary

- First generation microkernels (Mach, Chorus, Amoeba) were slow
  - Poorly designed
  - Complex API, Too many features
  - Large cache footprint → memory B/W limited
- L4 is fast due to small cache footprint
  - API size: 7 functions
  - 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

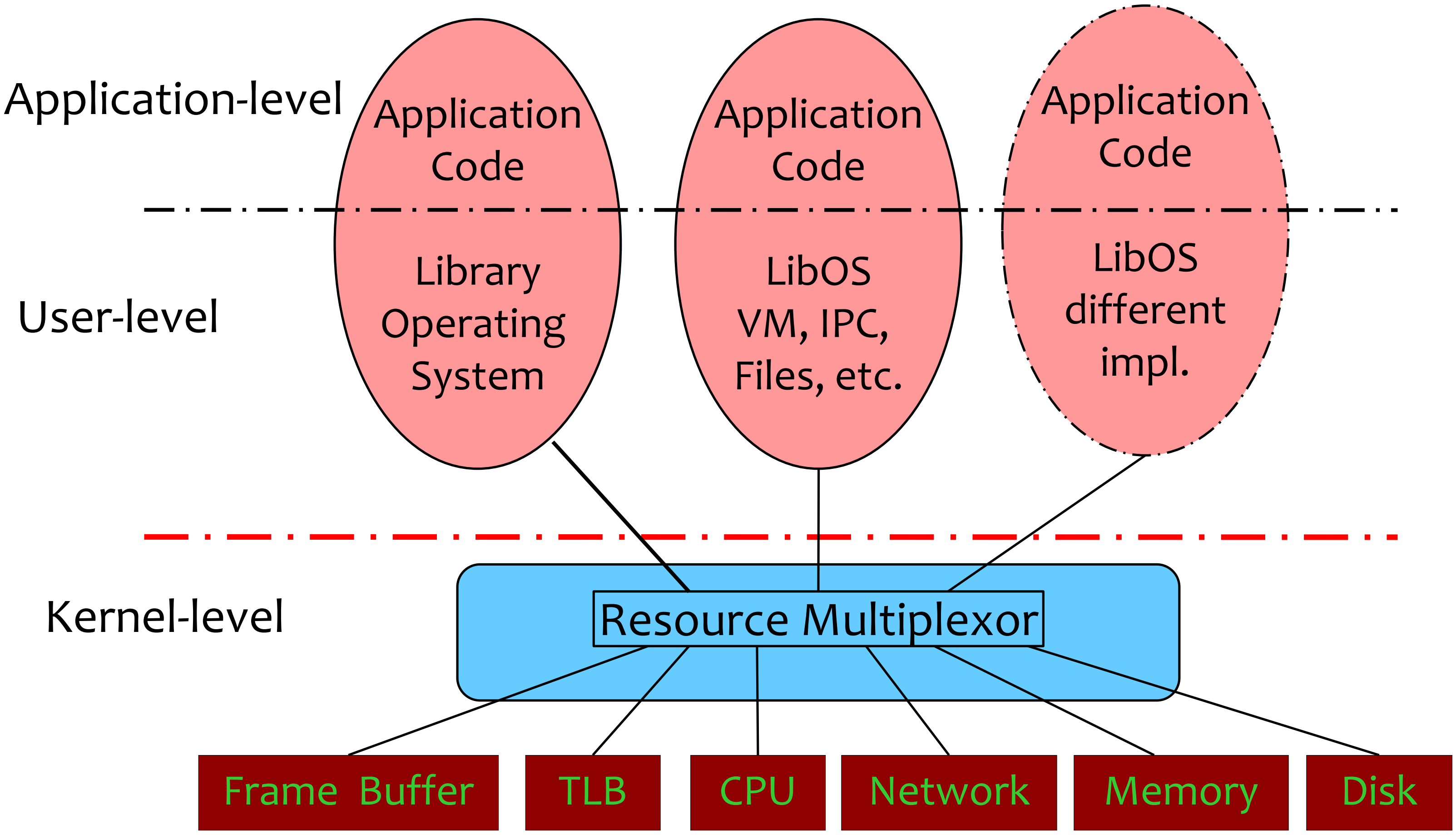
- Lines of code (x 10,000)



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
  - Follows end-to-end principle: minimal, fewest H/W abstractions possible
  - Exokernel is basically a secure *resource multiplexor*
  - *Drawbacks?*

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
- Expose both allocation and revocation
  - Library OSes can request specific physical resources
  - Revocation of resources is visible to user-level libOS
- Code can be downloaded to exokernel by application
  - Application safe handlers (ASH)

Hardware resource management

- Processor resource
  - Represented as a timeline, programs can allocate intervals of time and yield timeslice
  - Kernel notifies programs of events (interrupts/exceptions, begin/end of timeslice)
- Memory
  - Allocates physical memory pages to programs and controls the TLB
  - Sharing pages via sending capabilities, kernel checks page accesses for capabilities
- Disk
  - Kernel refers to blocks by physical address, application can optimize data placement
  - Security applies to disk sectors instead of files
  - Secure bindings: download code into kernel to determine ownership
- Network - Challenging! Why?
  - Kernel implements a programmable packet filter (secure bindings)
  - May need dedicated trusted authority (applications could lie)

Kernel comparison

- Monolithic
  - \+ performance
  - \- difficult to debug and maintain
- Microkernel
  - \+ more reliable and secure
  - \- performance overhead
- Hybrid kernels
  - \+ benefits of both monolithic and microkernels
  - \- same as monolithic kernels, just a marketing gimmick (Linus Torvalds)
- Exokernels
  - \+ minimal and simple
  - \- more work for application developers

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)
  - VirtualBox (Innotek GmBH, 2007 → Sun → Oracle)
  - Hyper-V (2008, Microsoft)
- 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:
    - Transparency: Essentially identical execution environment (as real machine)
    - Efficiency: Minor performance penalty for programs in VM
    - Resource Control: 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"

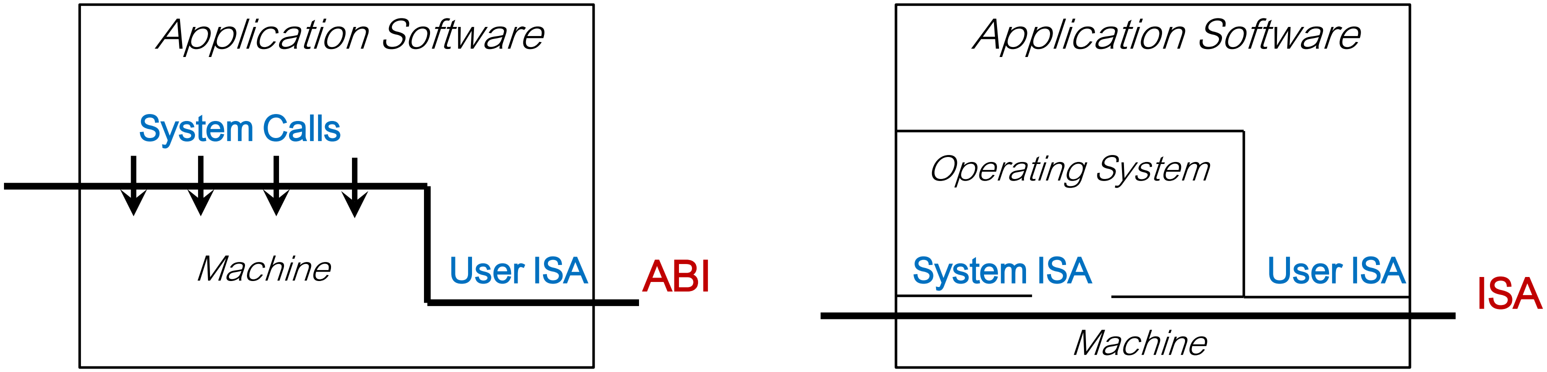
Why virtual machines?

- 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 - similar as original motivation
  - Security
  - Reliability/fault tolerance
  - Portability/compatibility
  - Avoid dealing with multiprocessor issues in OS
  - Migration
  - Performance
  - Innovation

Types of virtual machines

- Many uses of the term "virtual machine"
  - A matter of perspective (process, OS)
- Conventional software is developed/compiled for a specific OS and ISA
  - *Application binary interface (ABI)*: interface between a process and the machine
  - *Instruction set architecture (ISA)*: real machine vs. virtual machine

Machine interfaces

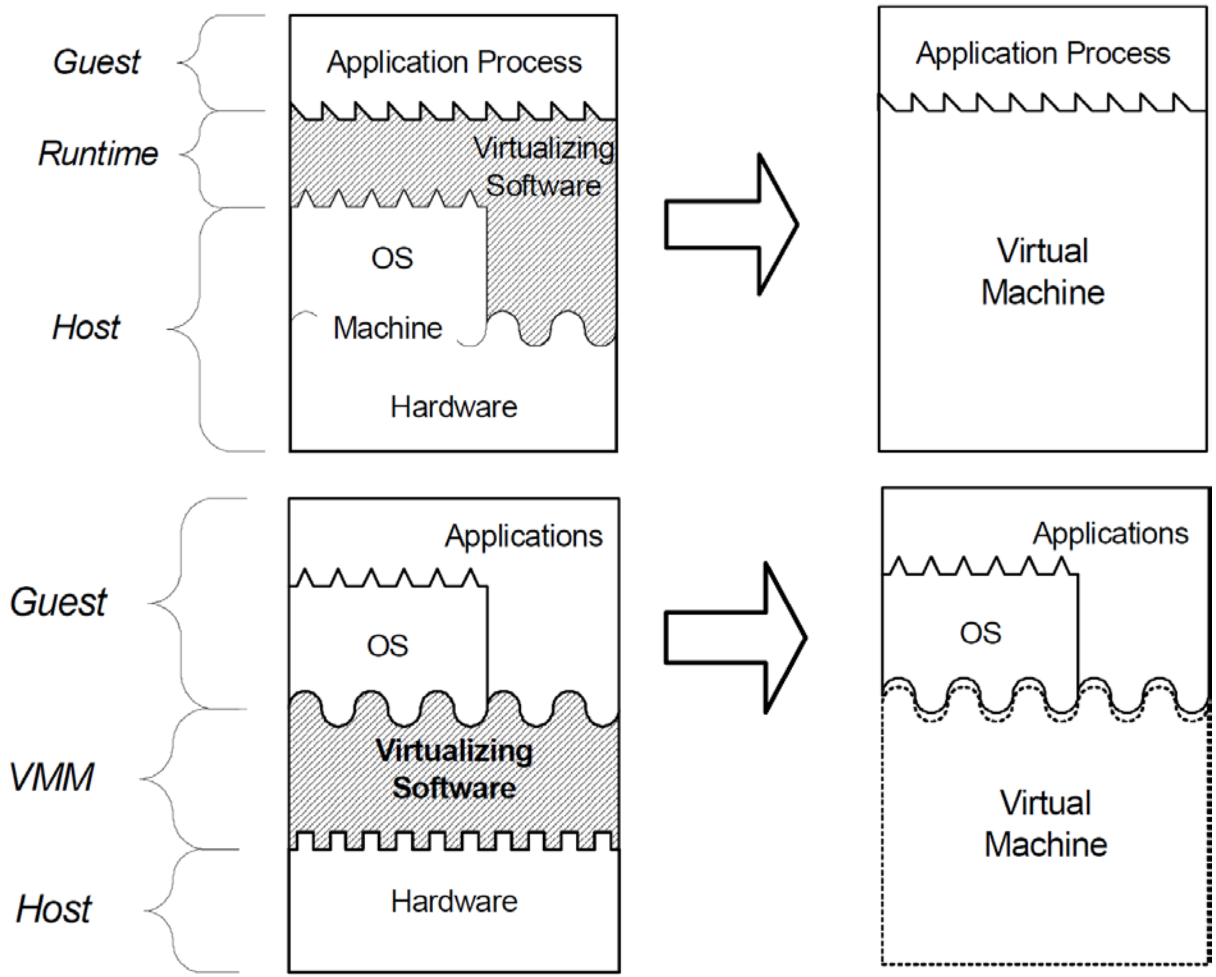


Source: Smith & Nair – Virtual Machines: Versatile Platforms for Systems and Processes

Types of virtual machines

- Virtualization – 2 parts:
  - Map virtual resources (registers, mem, files, etc.) to real resources
  - Use real machine instructions or syscalls on the host OS, to carry out instructions or syscalls specified by the VM
  - => Virtualization S/W must emulate the virtual machine ABI or ISA
- Distinguish virtual machines based 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.

Virtualization software



Images: Smith & Nair – Virtual Machines: Versatile Platforms for Systems and Processes

Process Virtual Machine

Virtualization S/W = ***Runtime***

System Virtual Machine

Virtualization S/W = ***VMM***

Smith & Nair's Taxonomy

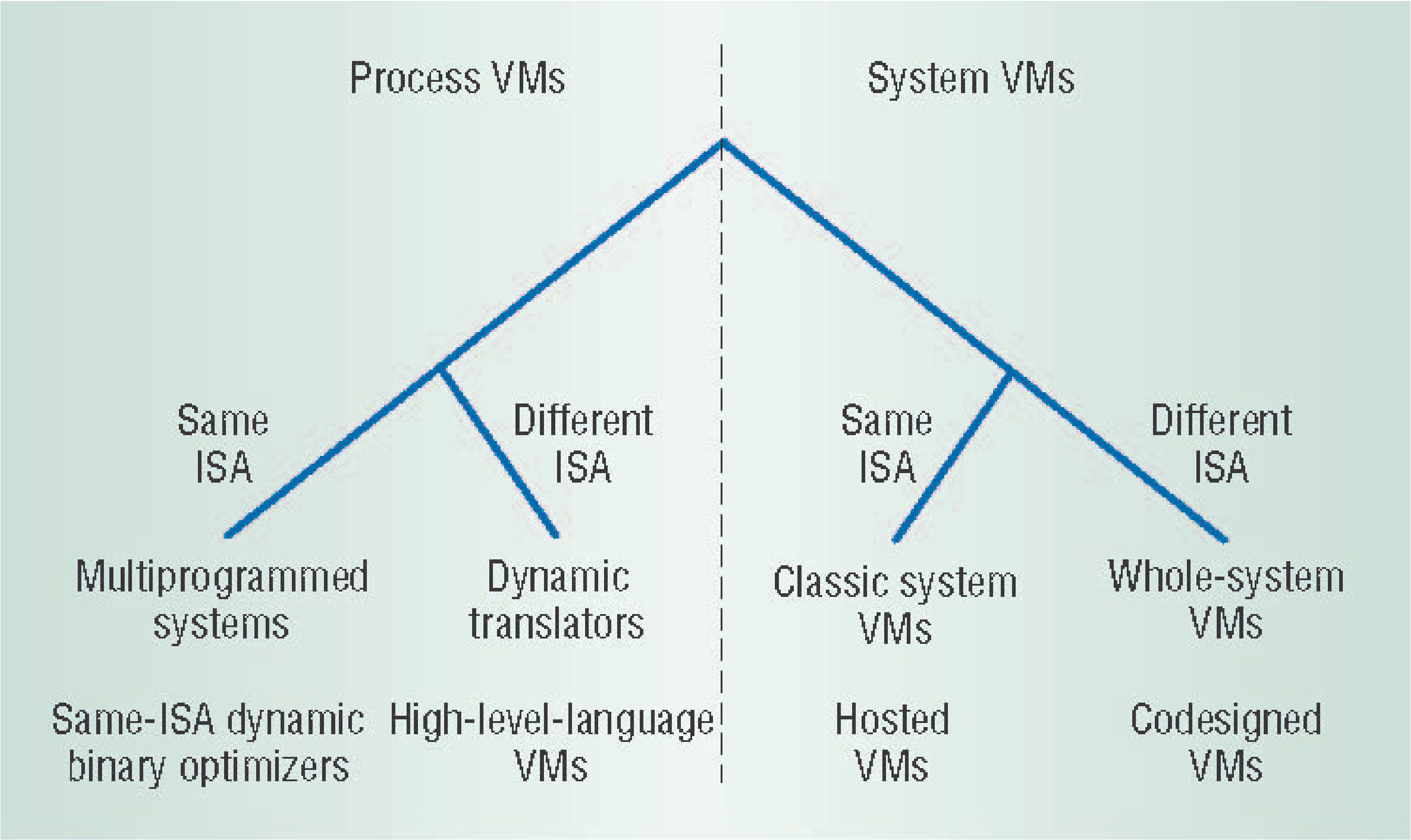


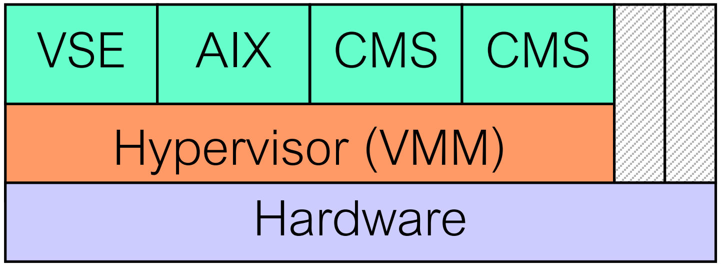
Image from: The architecture of virtual machines, J.E. Smith and Ravi Nair; IEEE Computer, Volume 38, Issue 5, May 2005 Page(s):32 - 38

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 (IA-32) Windows binaries on Alpha Windows platform
- Dynamic optimizers
  - Same guest/host ISA, only purpose is optimization. E,g., Dynamo
- High-level language VMs
  - Designed together with language
  - Mainly for portability & to support language features
    - E.g. Pascal P-code, Java bytecode, MS Common Language Infrastructure (CLI)

System VMs

- "classic" VMM (type I)
  - VMM runs on bare hardware, everything else runs on top
  - VMM is most privileged software, everything else less
- "hosted" VMM (type II)
  - Virtualizing software installed on top of existing OS
    - E.g. VMWare Workstation



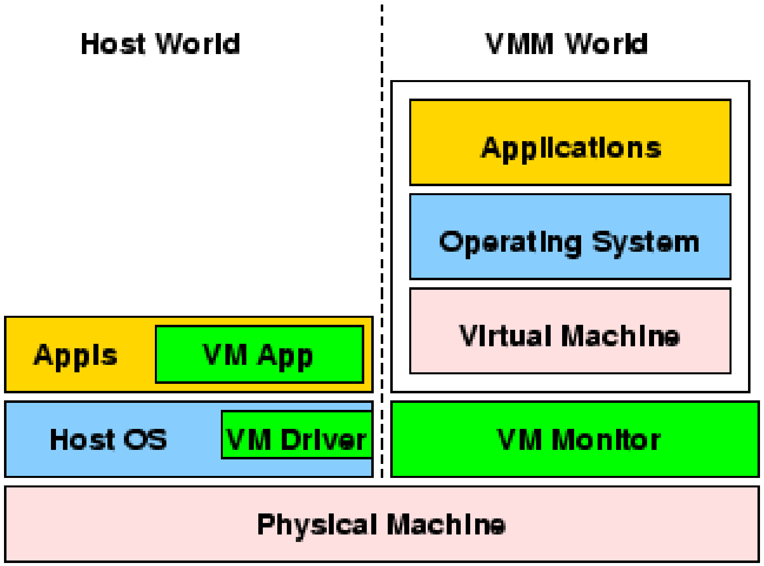


Image from: "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", J. Sugerman et al., Usenix 2001.

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
- VMM must provide 3 primary functions
  - Interpreter (“Virtualizing the computer”)
  - Dispatcher component
  - Allocator

Instruction Requirements

- Virtualizing is easy if all instructions are virtualizable.
- *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 used binary rewriting to deal with this
  - Xen required changes to the OS → *paravirtualization*
  - Intel VT, AMD-V (Pacifica) fix this

Disco

- Goals
    - Extend modern OS to run efficiently on shared memory multiprocessors without large changes to the OS
    - VMM can run multiple copies of Silicon Graphics IRIX operating system on a Stanford Flash shared memory multiprocessor

Problem

- Commodity OS's not well-suited for ccNUMA (1997)
  - Do not scale: Lock contention, memory architecture
  - Do not isolate/contain faults: more processors => more failures
- Customized operating systems
  - Take time to build, lag hardware
  - Cost a lot of money
- => Reduce the gap between H/W innovation and release of adapted system S/W

Solution

- Add a virtual machine monitor (VMM)
  - Commodity OSes run in their own virtual machines (VMs)
  - Communicate through distributed protocols
- VMM uses global policies to manage resources
  - Moves memory between VMs to avoid paging
  - Schedules virtual processors to balance load

Advantages

- Scalability
- Flexibility
- Hide NUMA effect
- Fault Containment
- Compatibility with legacy applications

VM challenges

- Overheads
  - Instruction execution, exception processing, I/O
  - Memory
    - Code and data of hosted operating systems
    - Replicated buffer caches
- Resource management
  - Lack of information
    - Idle loop, lock busy-waiting
    - Page usage
- Communication and sharing
  - Not a problem -> distributed protocols

Disco interface

- VCPUs provide abstraction of a MIPS R10000 processor
  - Emulates all instructions, the MMU, trap architecture
  - Enabling/disabling interrupts, accessing privileged registers -> Memory-based interface to VMM
- Physical memory
  - Contiguous address space, starting at address 0
  - Physical-to-machine address translation, second (software) TLB

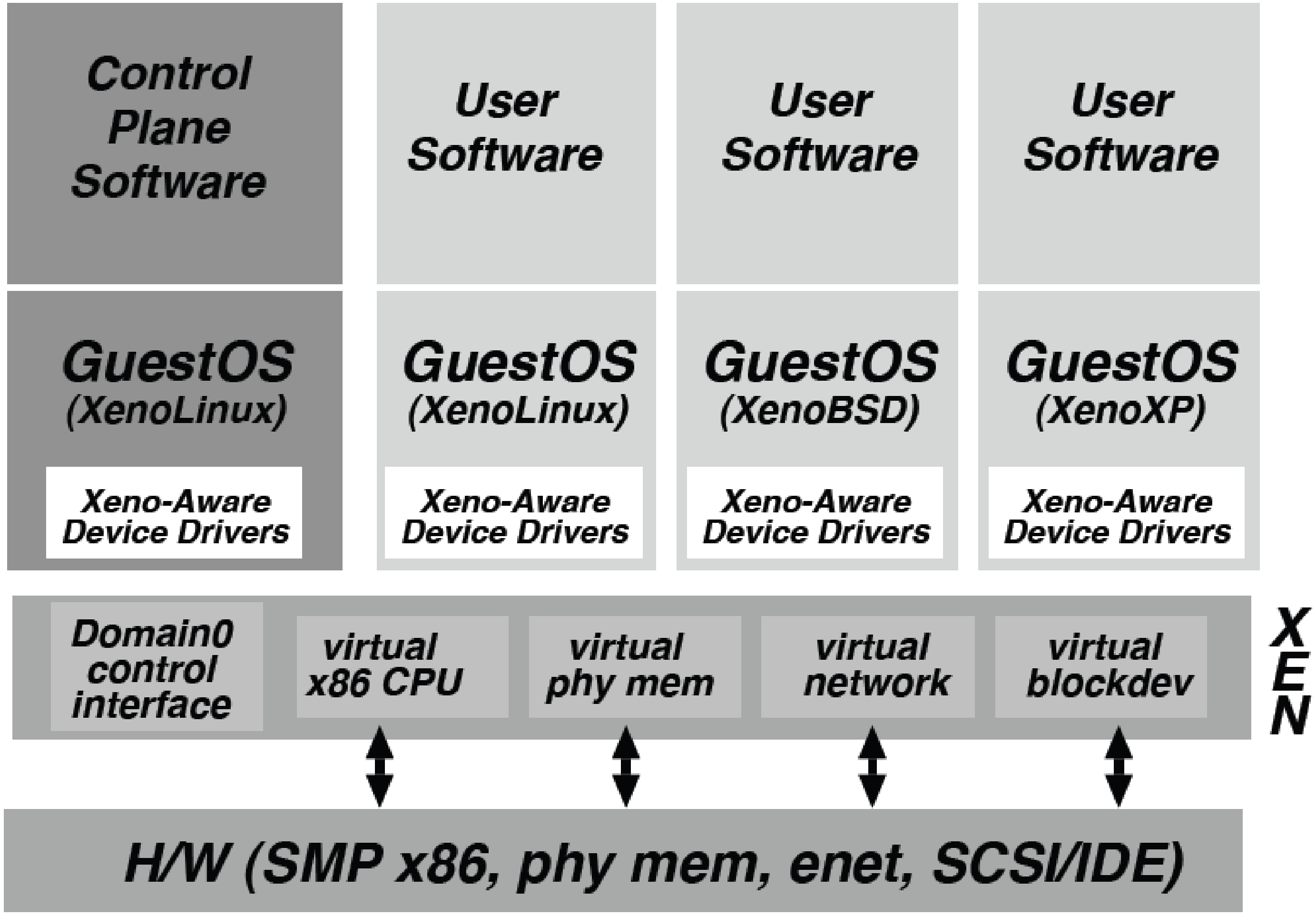
Disco interface (cont'd)

- I/O devices
  - Each VM assumes exclusive access to I/O devices
  - Virtual devices exclusive to VM
  - Physical devices multiplexed between virtual ones
  - Special interface to SCSI disks and network devices
  - Interpose on DMA calls
  - *Disk:*
    - Set of virtualized disks to be mounted by VMs
    - Copy-on-write disks; for persistent disks, uses NFS
  - *Network:*
    - Virtual subnet across all virtual machines
    - Uses copy-on-write mappings => reduces copying, allows sharing

Xen virtualization

- Technically, two kinds
- Paravirtualization
  - Guests run a modified OS
  - High performance on x86
- Hardware-assisted virtualization
  - CPUs that support virtualization
  - Unmodified guest OSes

Xen infrastructure



VM Performance

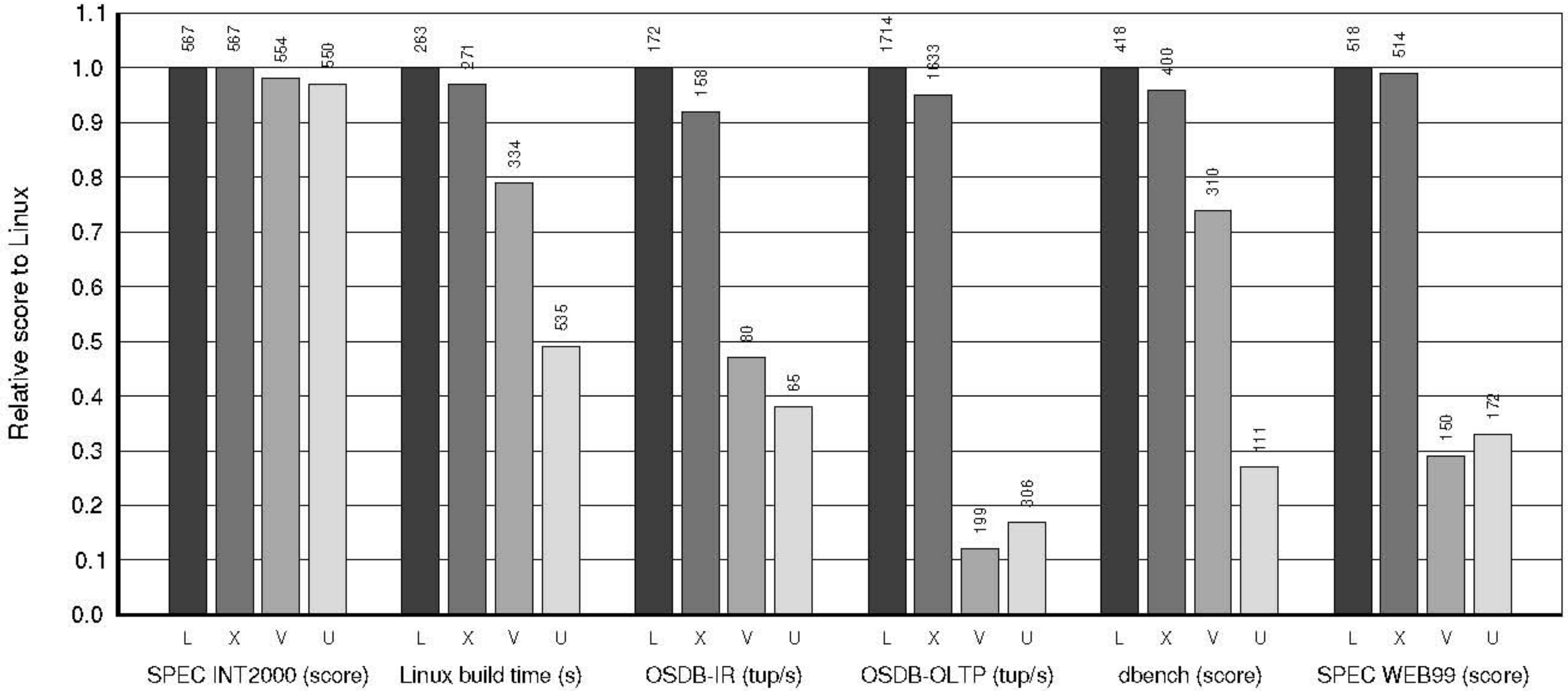


Figure 3: Relative performance of native Linux (L), XenoLinux (X), VMware workstation 3.2 (V) and User-Mode Linux (U).

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

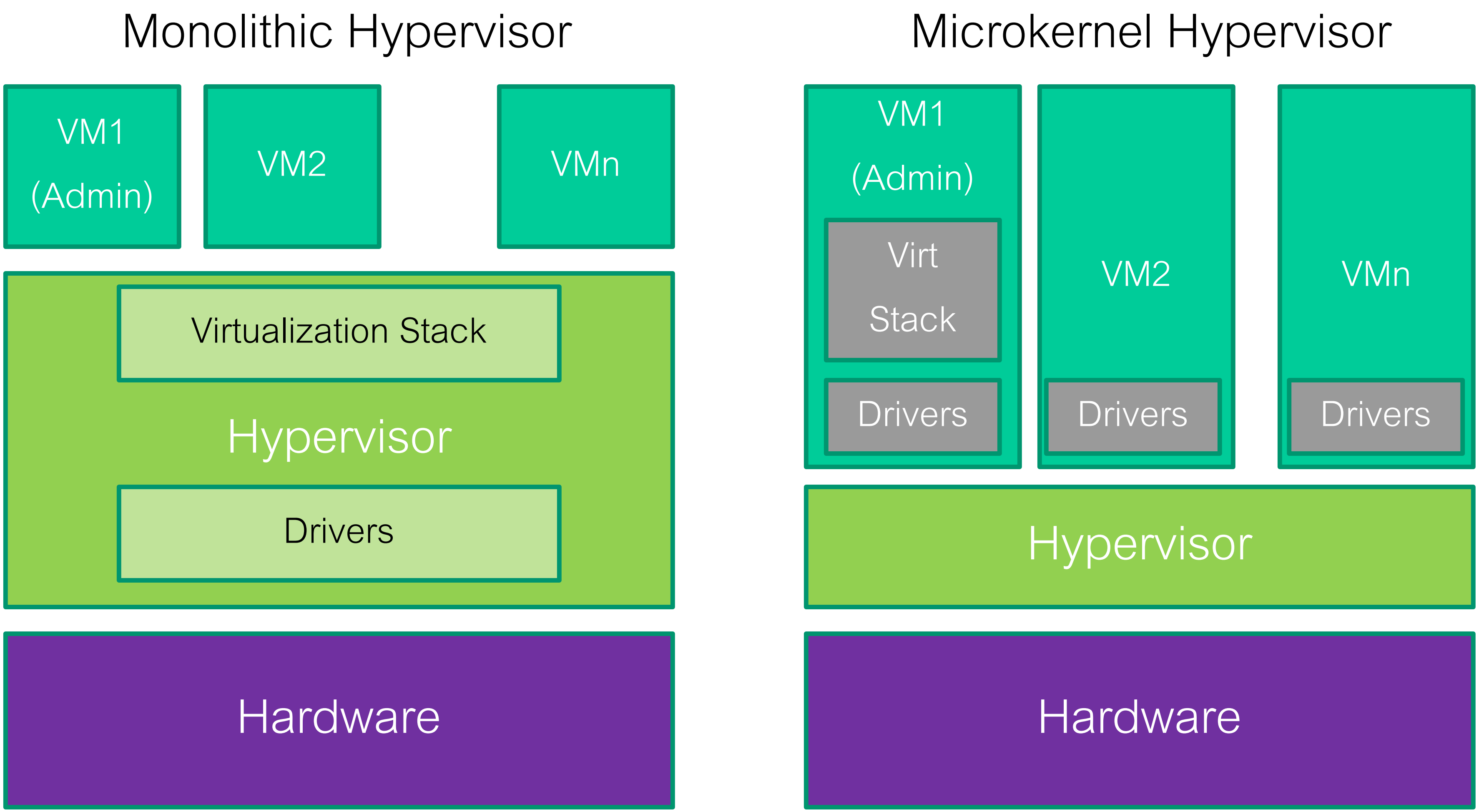
How does it compare to Disco?

- Three main differences
  - Less complete virtualization
  - *Domain0* to initialize/manage VMs, incl. to set policies
  - Strong performance isolation
- Other
  - Interface is pretty close to hardware and enables low-overhead high-performance virtualization
  - Need to change more OS code than in Disco
- All the cool details: readings and tutorial discussion
  - "Xen and the art of virtualization" – SOSP'03

Hypervisors for servers

- Type 1 or Type 2?
- Hyper-V: “MicroKernelized” Hypervisor Design
- VMWare ESX Server: “Monolithic” hypervisor architecture

Hypervisor design



- Both true Type 1 hypervisors – no host OS
- The hardware is the physical machine; OSs are all virtual

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

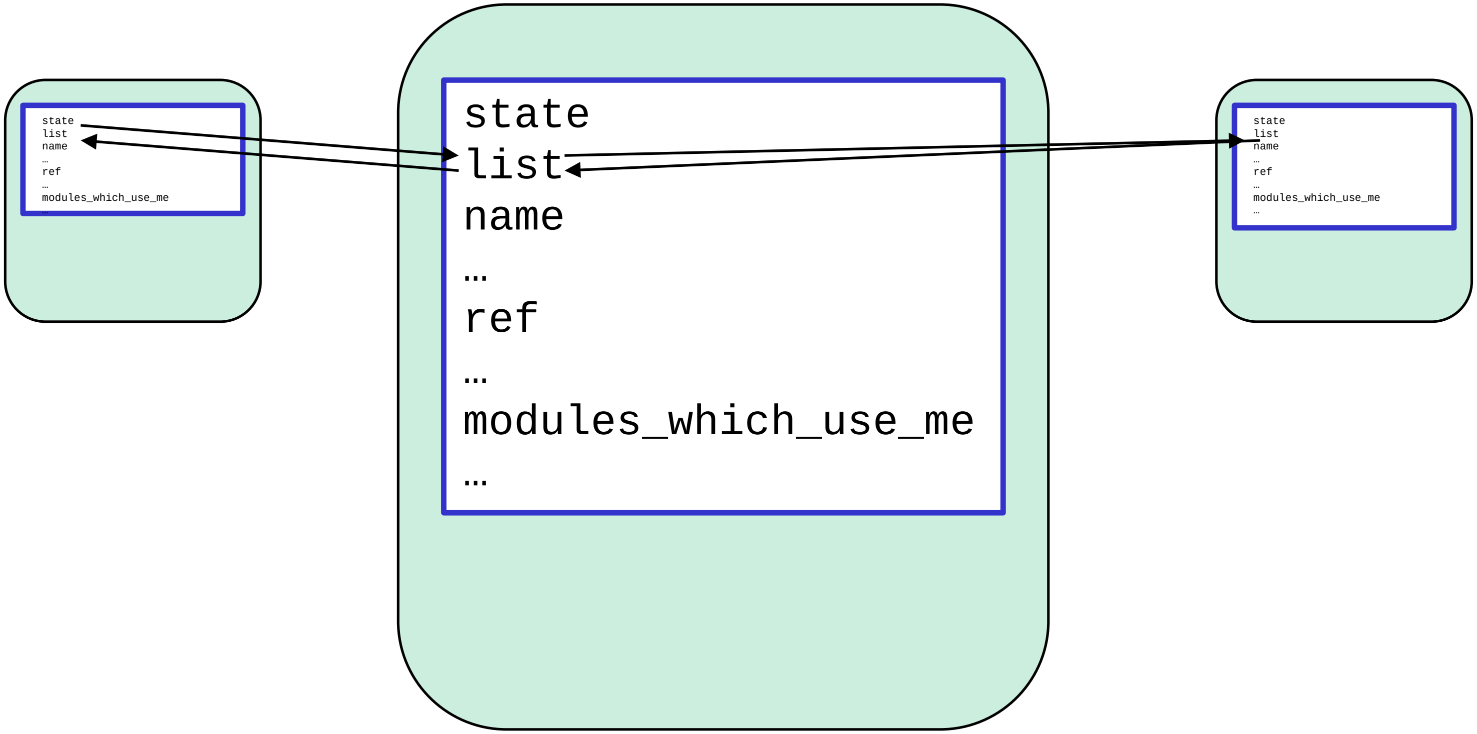
- Giving everyone a virtual machine doesn't entirely solve the extension problem
  - You can run what you want on your VM, but do you really want to write a custom OS?
- Often just want to modify/replace small part
- Solution: Allow parts of the kernel to be dynamically loaded / unloaded
  - Requires 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 .ko file (since 2.6)
- *insmod* command loads module into running kernel
  - 2.4 – insmod resolves references to kernel symbols
  - 2.6 – invokes init_module() syscall, kernel does the linking
- *rmmod* command removes module from kernel
- *lsmod* command lists currently-installed modules

Tracking Modules

- Kernel has a linked list of module objects
  - struct contained in the module memory itself



sys_init_module()

- Kernel handler for init_module() system call
- Checks permission and copies arguments to kernel
- Checks that module is not loaded already
- Allocates memory for module and fills it in from ELF sections
- Locates `module` object structure in this memory, initializes fields
- Relocates all external or global symbols with correct addresses
- Links new module into list
- Sets state to MODULE_STATE_COMING
- Calls init function of module
- Sets state to MODULE_STATE_LIVE
- Module states: UNFORMED, COMING, LIVE, GOING

rmmod

- Unlinks module from kernel
- Needs to ensure no one is using module first!
  - Reference count incremented whenever module is used
  - modules_that_use_me list identifies other modules that that depend on this one
- Frees memory

Problems with module approach

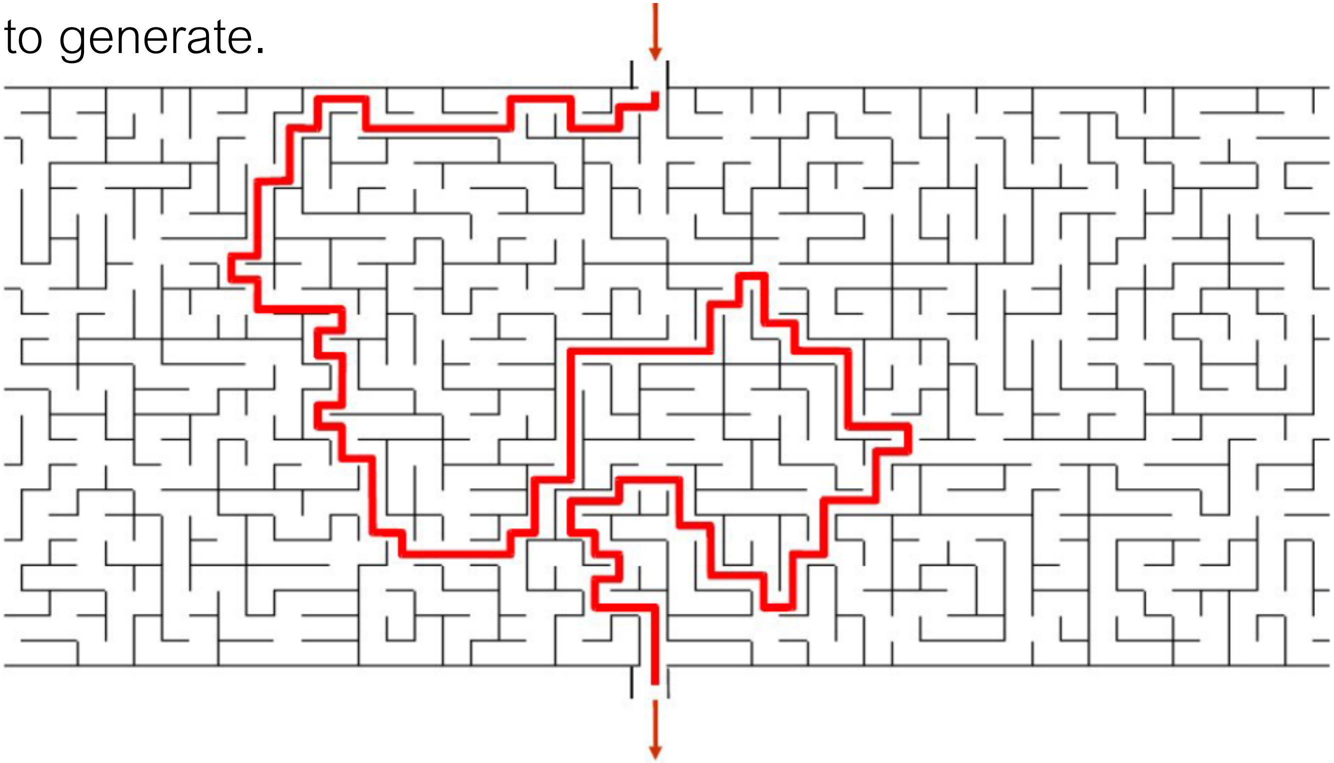
- Requires stable interfaces
  - Linux uses version numbers to check if module is compiled for correct version of kernel, but it is easy to get this wrong
- Unsafe
  - Module code can do *anything* because it runs privileged
    - E.g. VMWare Workstation driver
      - "hijacks" machine by changing *interrupt 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
  - Code Consumer (OS) supplies a specification for what extensions are allowed to do
  - Code Producer (the 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)

Checking a proof vs generating one

- G. Necula - Safe Kernel Extensions Without Run-Time Checking, OSDI'96
- A maze is "safe" if there's a path through it. => Easy to check a path, but hard to generate.



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”
  - Byte-Granularity Isolation (Microsoft) - BGI