

Introduction & 8 Important Problems in Modern Operating Systems

Haibo Chen

Institute of Parallel and Distributed Systems (IPADS)

Shanghai Jiao Tong University

http://ipads.se.sjtu.edu.cn/haibo_chen

Outline

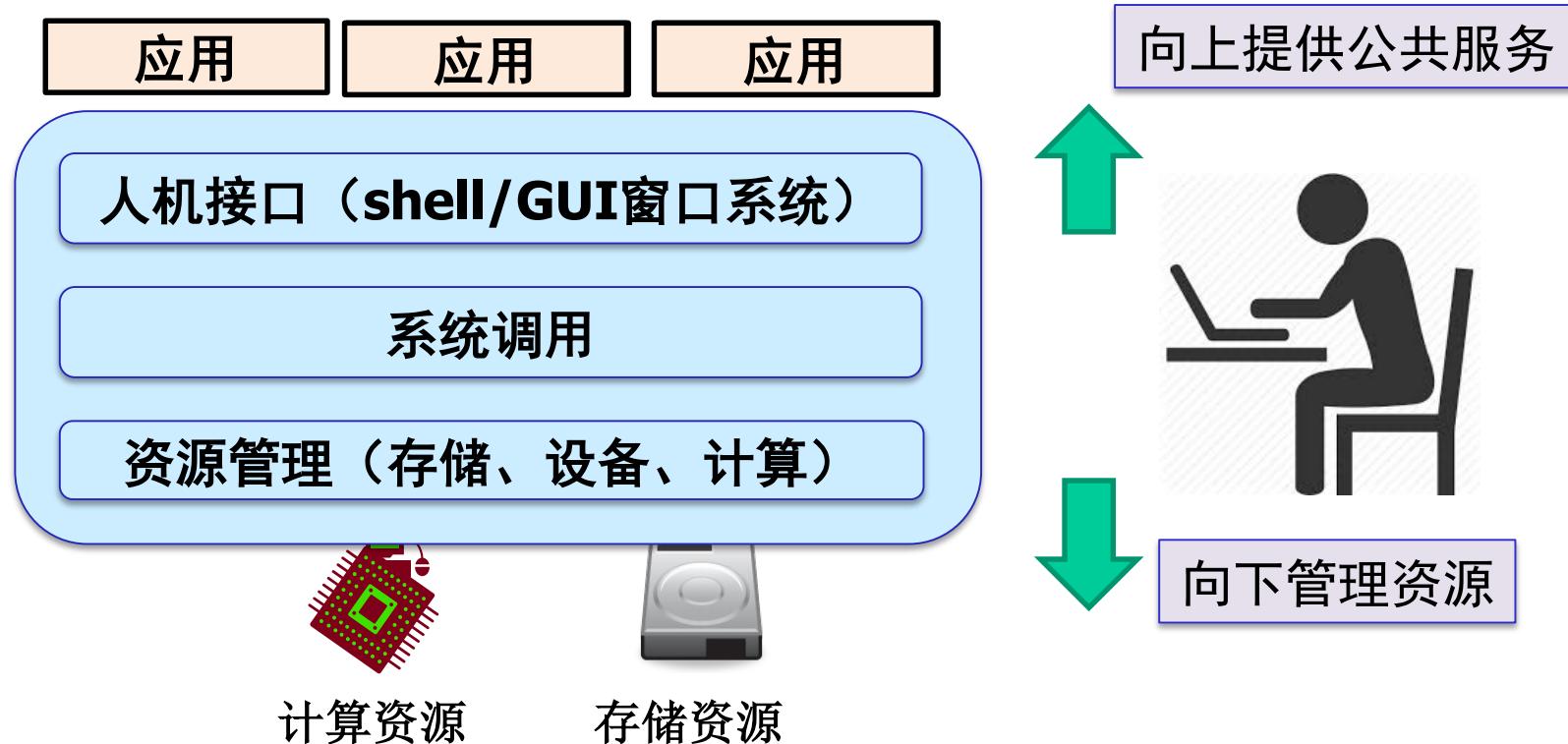
Why Choose This Course?

8 important problems of OS

Course information

What is an Operating System?

- 操作系统是管理硬件资源、控制程序运行、改善人机界面和为应用软件提供支持的一种系统软件。
[计算机百科全书(第2版)]



Why Study OS?

Operating system is a maturing field

Hard to get people to switch operating systems

Then why?

Many things are OS issues

High-performance

Resource consumption

Battery life, radio spectrum

Security needs a solid foundation

New “smart” devices need new OSes

Q: What applications are highly OS-related, which are not?

Why Study OS?

OS is the core of the broad “systems area”

Lots of systems companies

Microsoft, Google, IBM, EMC, CISCO, VMware

What's Google's core?

Google Cluster, GFS, MapReduce, BigTable

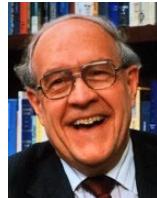
Good OS knowledge is a necessity to get you good offers/promotions

at such companies and top universities

Turing Awards and OS Evolution



Maurice Wilkes
1967年图灵奖



Frederick Brooks
1999年图灵奖



Fernando J. Corbató
1990年图灵奖

EDSAC, 1949
Multi-programming
第一台存储程序式电子计算机

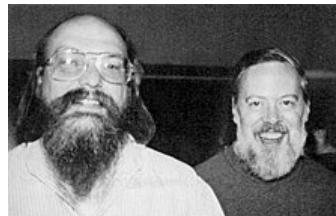
IBM System/360, 1964

CTSS, 1961 & Multics, 1969
分时操作系统

Unix, 1971
多任务多用户操作系统

Venus, 1972
小型低成本交互式分时操作系统

2019
分布式操作系统



Ken Thompson & Dennis Ritchie
1983年图灵奖



Barbara Liskov
2008年图灵奖



缺心(芯)少魂(操作系统)

人民日报:让芯片、操作系统不再有"卡脖子"隐忧

业界：操作系统从“免费”走向收费， 从开放走向“封闭”

- 例子1：Google正式对欧盟区域的Android进行收费，初步高达40美元每设备

Google may charge up to \$40 per Android device for app suite following EU ruling

- 例子2：IBM 340亿美元收购RedHat，构筑其云计算竞争力

IT'S OFFICIAL: IBM is acquiring software company Red Hat for \$34 billion

- 例子3：谷歌投入600+人力数十亿美元研发自研OS Fuchsia

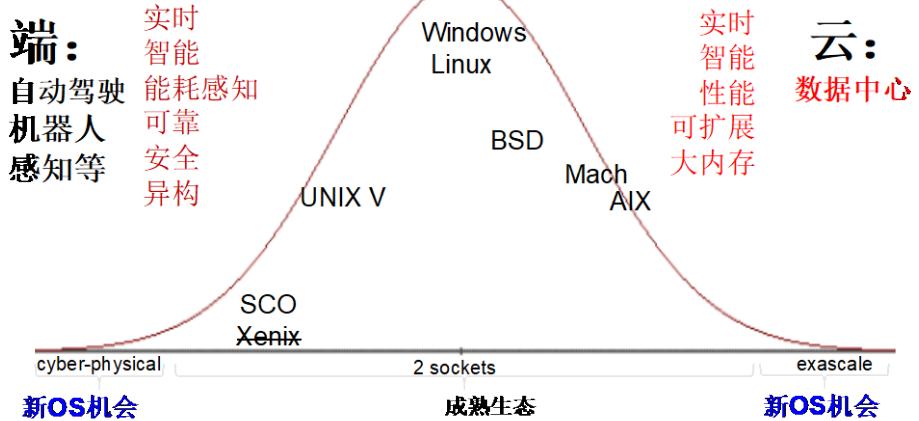
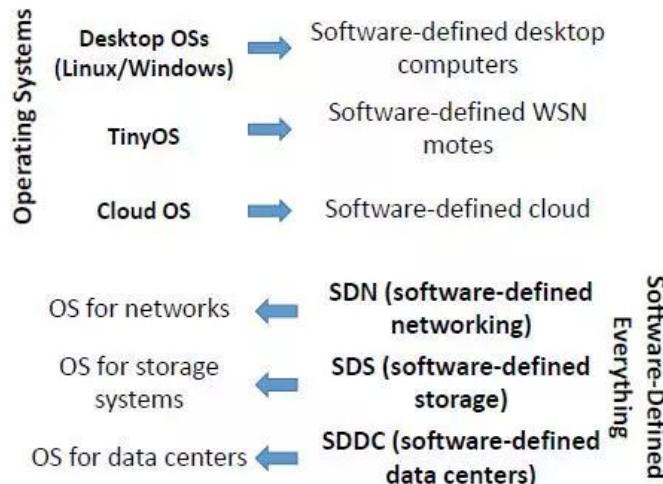


硬件+应用驱动：孕育泛在操作系统机会

操作系统发展：应用驱动+硬件驱动

观点：

1. 硬件与应用蓬勃发展，在端和数据中心侧可能孕育新的操作系统
2. 传统操作系统如Linux进入架构稳定期，难以根本上适应新硬件与应用需求



Real Machine VS. Abstraction Machine

Many courses emphasize abstraction

- Abstract data types

- Asymptotic analysis

These abstractions have limits

- Especially in the presence of bugs

- Need to understand details of underlying implementations

Useful outcomes from taking this course

Become more efficient programmers

Able to find and eliminate bugs efficiently

Able to understand and tune for program performance

Understand design and implementation of real complex systems

Try to build one on your own (like a Turing-award level OS)

Is “simple” really simple?

A simple program?

```
#include <stdio.h>
int main() {
    printf("hello world\n");
    return 0;
}
```

How does it get executed?

How does it get translated
to machine code?

How does the program get
executed in detail?

Link, load, execute,
finish?

Course Perspective

Our course is programmer-centric

Purpose is to show that by knowing more about the underlying system, one can be more effective as a programmer

Enable you to

Write programs that are more reliable and efficient

Incorporate features that require hooks into OS

E.g., concurrency, signal handlers

Build an operating system!

Cover material in this course that you won't see elsewhere

Not just a course for dedicated **hackers**

Outline

Why we study OS?

8 problems of OS

Course information

8 IMPORTANT PROBLEMS IN OPERATING SYSTEMS

8 Important Problems (Not a Rank)

Scale Up

Security & Trustworthy

Energy Efficiency

Mobility

Write Correct (Parallel) Code

Scale Out

Non-Volatile Storage

Virtualization

#1: Scale up (or Performance Scalability)

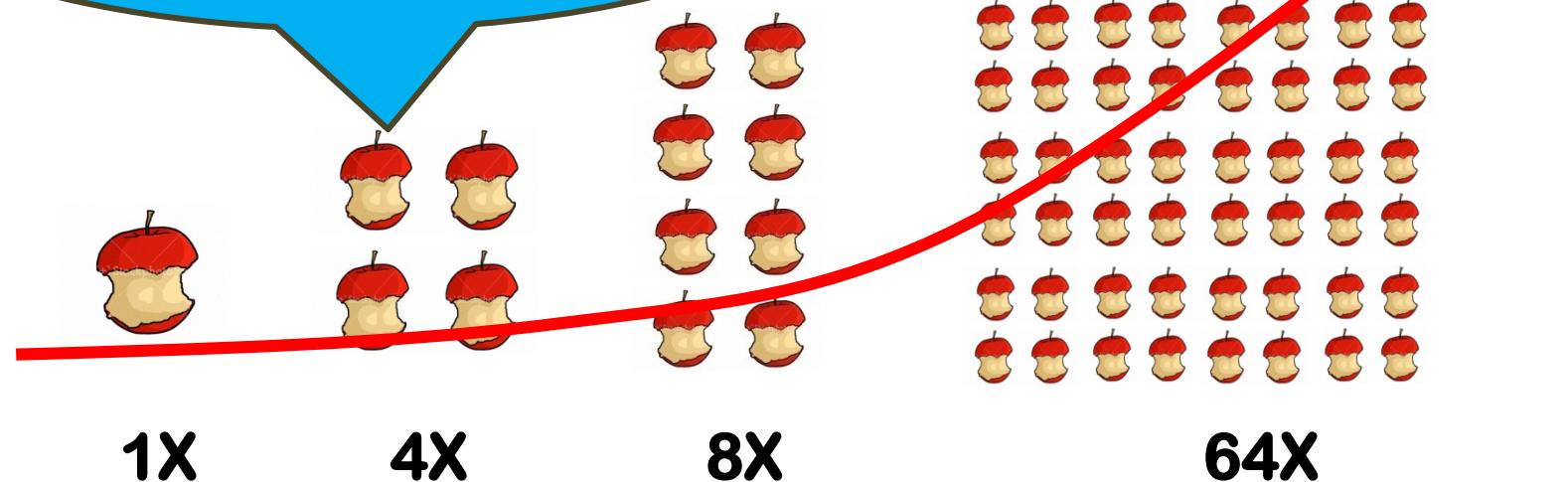
Multicore Evolution

Multicore is commercially prevalent recently

Eight cores and Twelve cores on a chip are common

Hundreds of cores on a single chip will appear in near feature

**Challenge#1: make
software performance
follow Moore's Law**



Why Operating System Matters?

Terms of operating system here

Broadly defined, including hypervisor, operating systems, runtime environments

Operating systems manage and hide resources from applications

- Hide tedious and complex low-level details

- Provide execution environments to apps

Operating systems determines app perf. in many cases

- Many apps spend a large fraction of time in system software

Operating System Meets Multicore

(Operating) systems 20 years ago

Enjoys the free lunch provided by hardware (Moore's Law)

Recall the Andy-Bill's Law

Multicore: ending the free lunch!

Software must evolve with hardware changes

OS: a vital role to bridge the gap between apps and hardware

Need to evolve itself with multicore

Need to evolve for apps with multicore

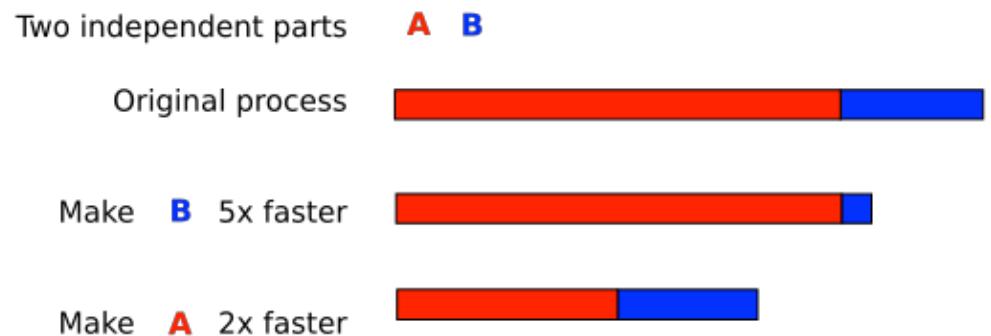
What is scalability?

Application does N times as much work on N cores
as it could on 1 core

Scalability may be limited by Amdahl's Law:

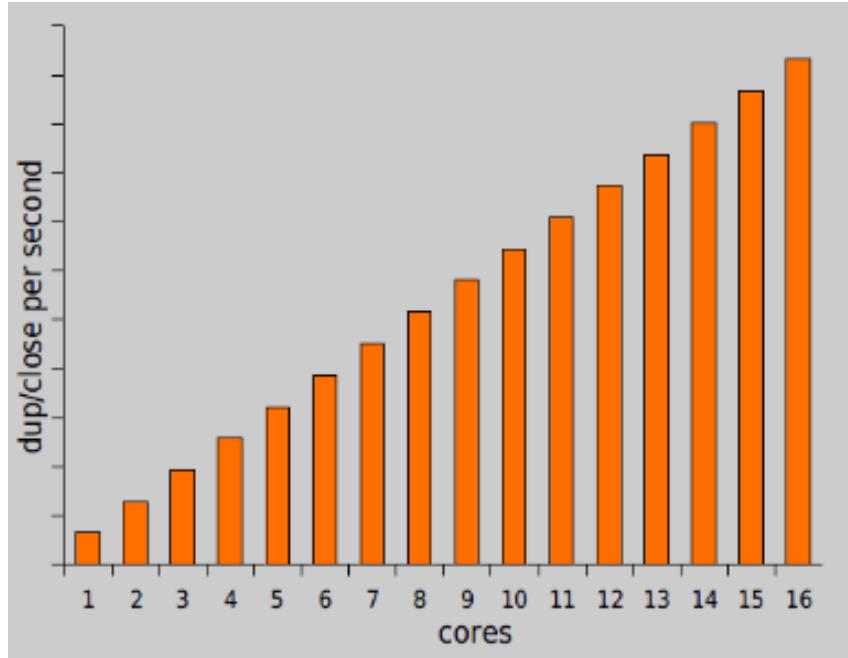
Locks, shared data structures, ... Shared hardware
(DRAM, NIC, ...)

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

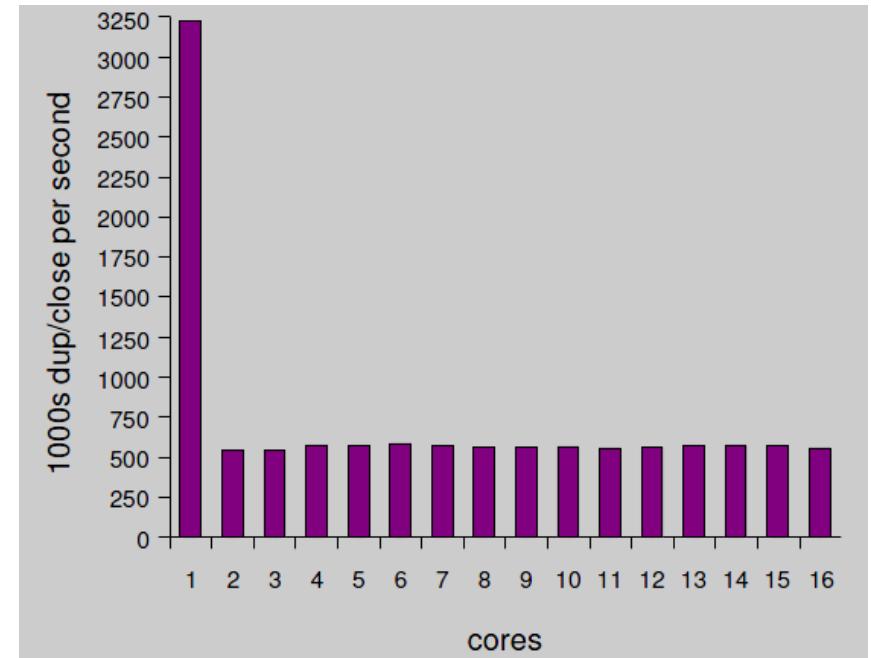


Motivating example: file descriptors

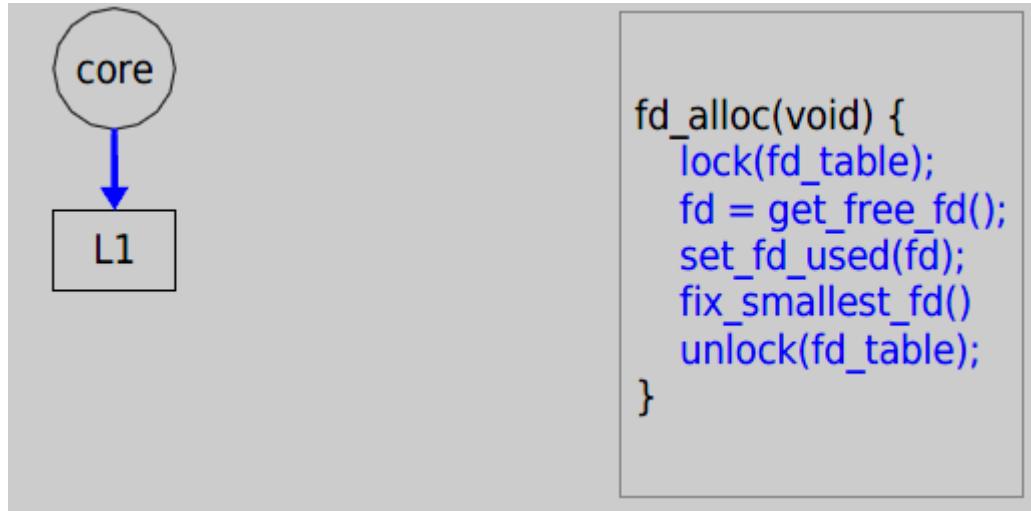
Ideal FD performance graph



Actual FD performance

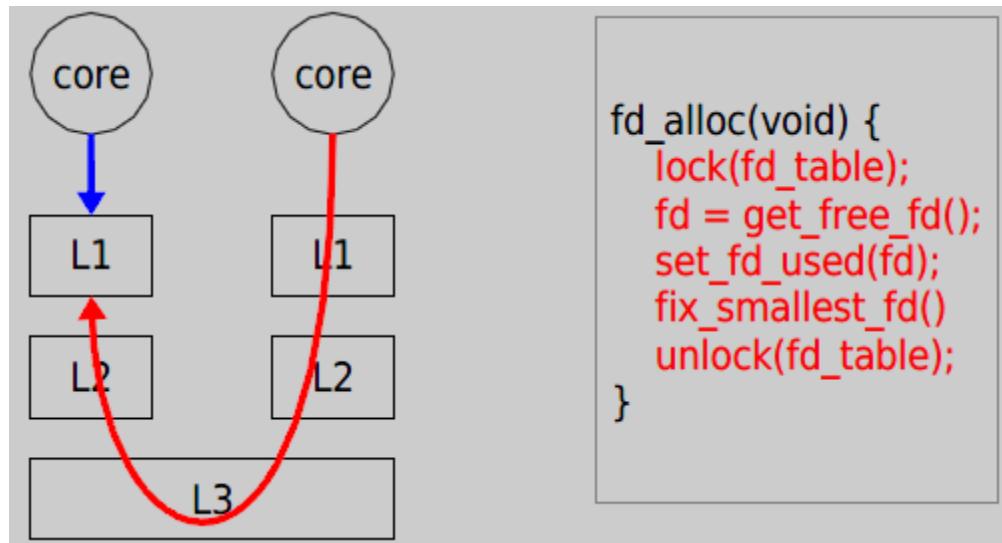


Why throughput drops?



Load `fd_table` data from L1 in 3 cycles.

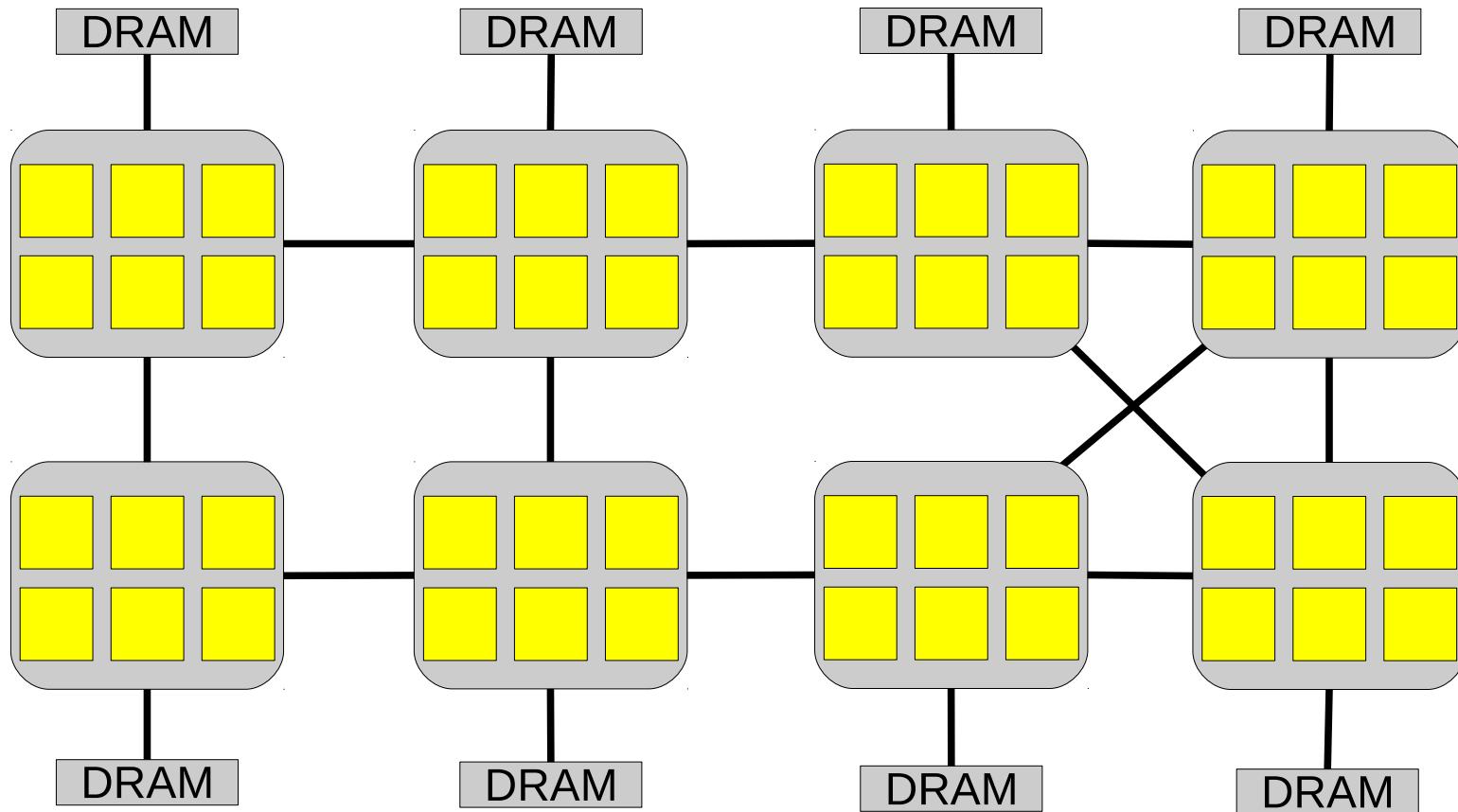
Why throughput drops?



Now it takes 121 cycles!

Off-the-shelf 48-core server

6 core x 8 chip AMD



Scale Up OS

Better abstraction to user applications

Eliminating non-scalable synchronization

Minimize sharing of common data structures

Bridge the memory wall

You will fill this..

Issue#2: Security & Trustworthy

Security



From Wikipedia:

Security is the degree of resistance to, or protection from, harm

Why Operating System matters?

Operating system (including hypervisors) lies in the lowest levels in a computer

How could you protect your application & data without properly protect your OS

CVE-2015-8370



新浪科技 > 业界 > 正文

新闻

佟丽娅



Linux开机漏洞：连接28下Backspace可入侵系统

2015年12月21日 16:50 新浪科技 微博



CVE-2016-4484

Linux神奇漏洞：长按回车键70秒 Root权限到手

2016-11-17 21:32:29 作者：[上方文Q](#) 编辑：[上方文Q](#) 人气：**18907** 次 评论(18)

A- A+

让小伙伴们也看看：



21

收藏文章

一般来说获取系统Root权限是很困难的，尤其是加密系统中，但西班牙安全研究员Hector M arco、Ismael Ripoll发现，Linux系统下只需按住回车键70秒钟，就能轻松绕过系统认证，拿到Root权限，进而远程控制经过加密的Linux系统。

Meltdown caused by the addiction of performance

Break user/kernel isolation

- ▶ Allow attacker to read arbitrary kernel data

Hardware bug in architecture

- ▶ Hard to be fixed by micro-code patch

Exist in almost all Intel CPUs produced **in past 20 years**

Meltdown



key = 0x01

Kernel

Mapped with kernel
privilege in page table

User



Access key

Permission Error!

Meltdown



key = 0x01

Mapped with **kernel privilege** in page table

Kernel

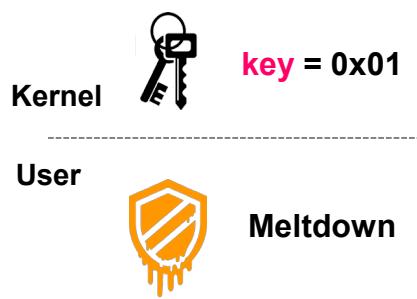
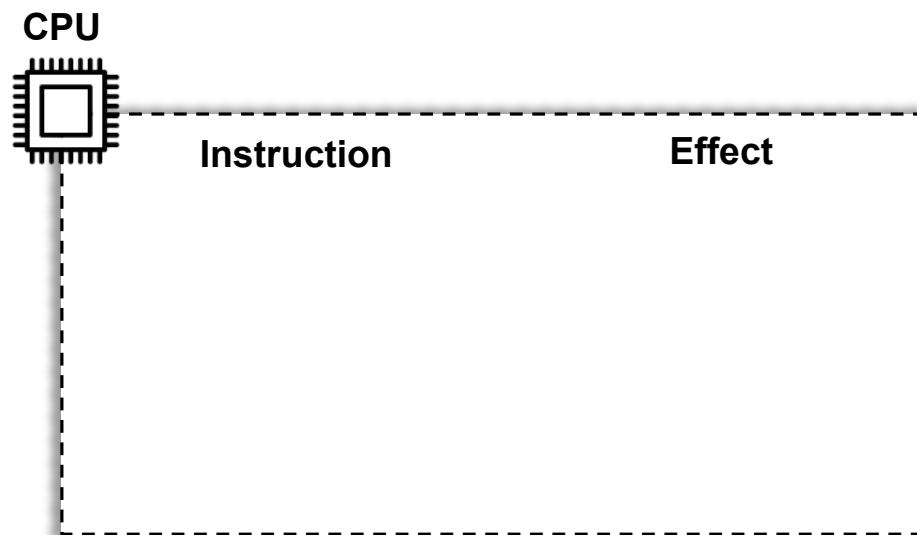
User



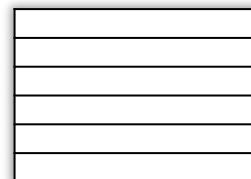
Meltdown

Can access **key!**

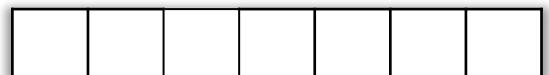
Meltdown



Cache

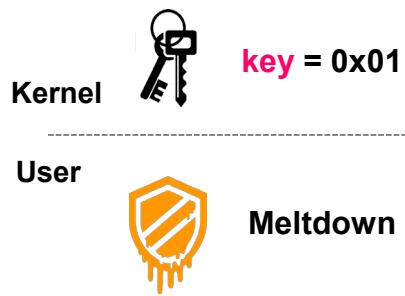
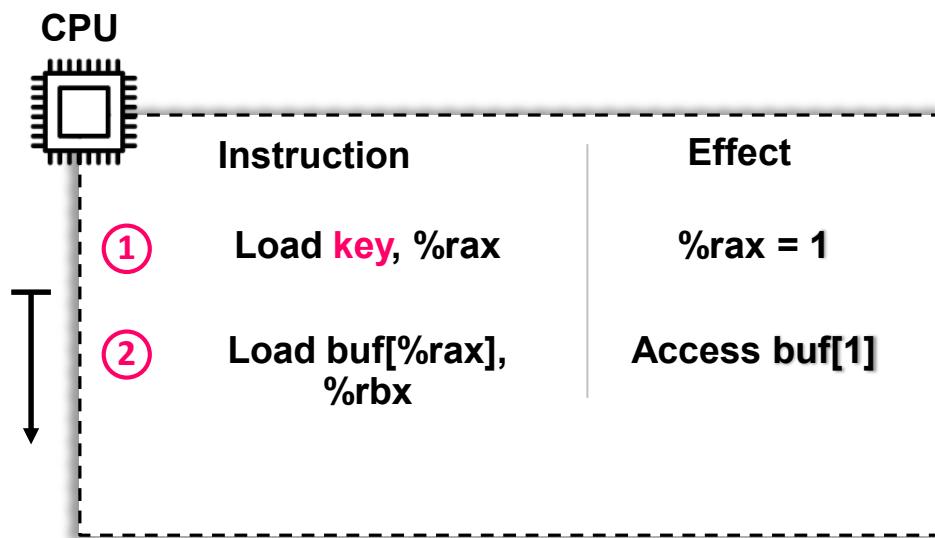


Memory

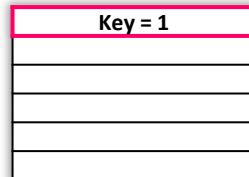


Meltdown

Permission
Error!
Reorder
Execution



Cache

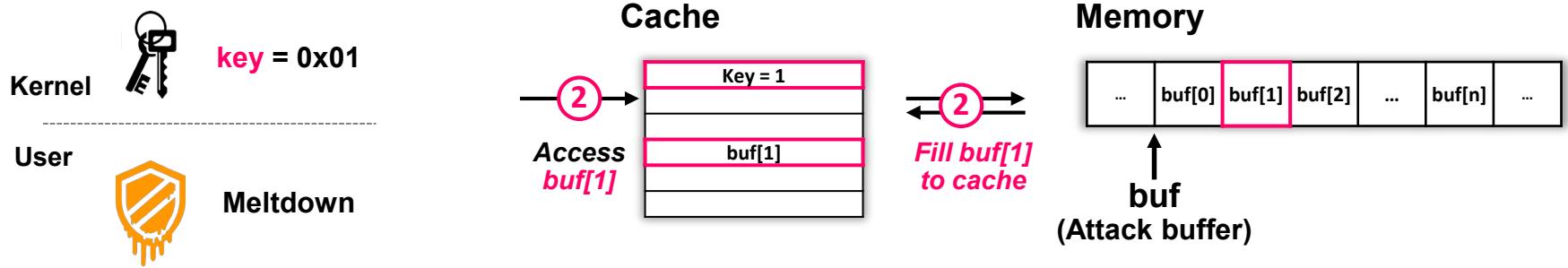
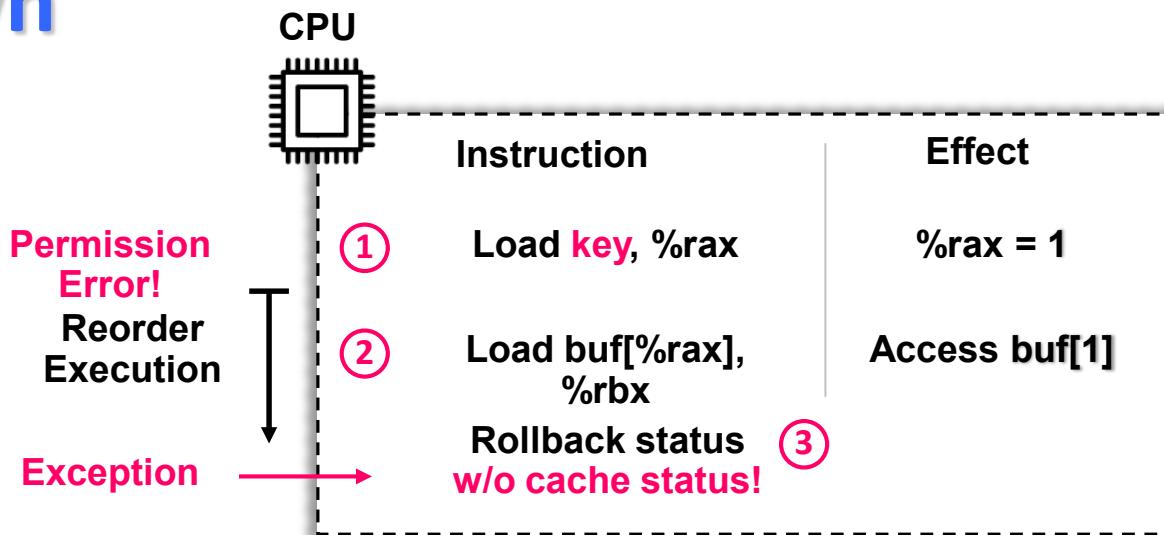


Memory

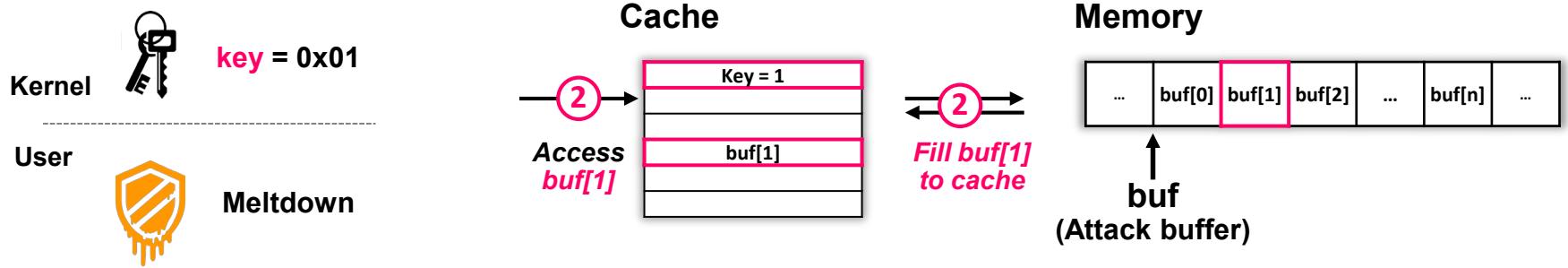
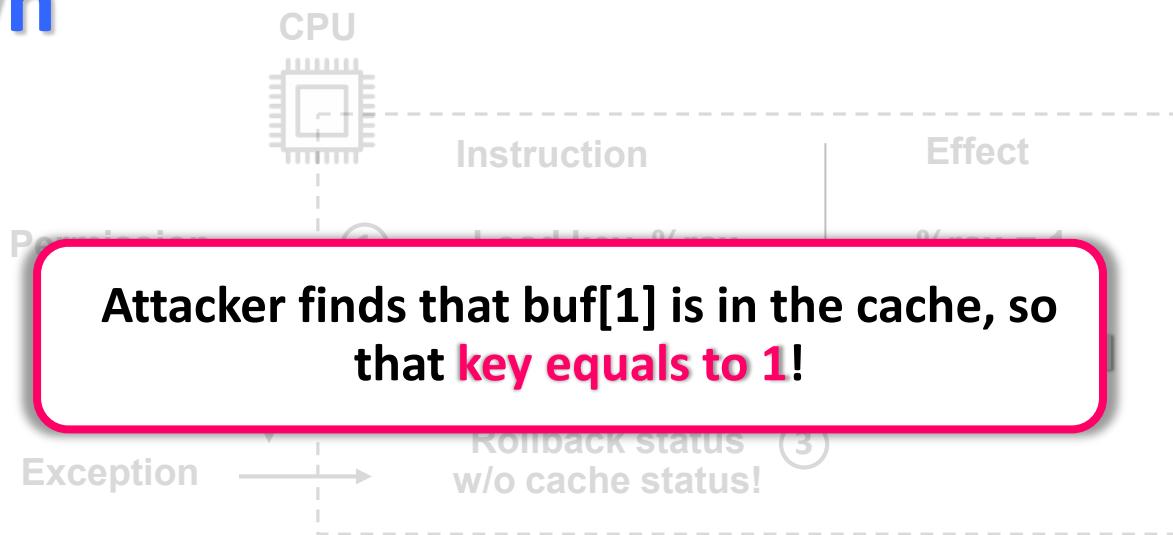


buf
(Attack buffer)

Meltdown



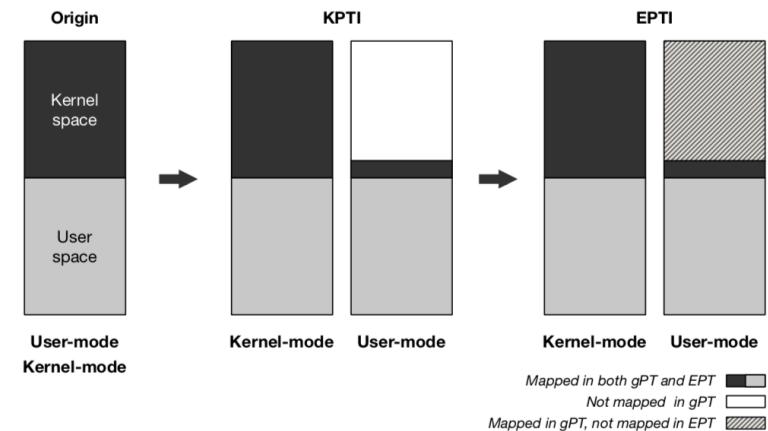
Meltdown



Defense against Meltdown

- Linux standard approach: KPTI

- Need to patch guest VM
 - Large overhead: $\geq 10\%$



- Approach from IPADS: EPTI

- Using EPT and VMFUNC for guest VM protection
 - No hypervisor involvement for Kernel/User crossings
 - No patches required to existing VMs
 - Performance overhead reduced 45% over KPTI

Trustworthy

- “To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.”
 - – Reflection on trusting trust [Thompson84]
- Approach: formal verification
 - Recent research: seL4
 - Industry adoption: HongMeng @ Huawei



Issue#3: Power Efficiency

Power Efficiency

Greener World, Better Life!



Again, why it has to do with OS?

OS control every computer and many devices!

Thus determines the power used by each computer

Google Datacenter: Finland



Power Breakdown in Server

Component	Peak Power	Count	Total
CPU [16]	40 W	2	80 W
Memory [18]	9 W	4	36 W
Disk [24]	12 W	1	12 W
PCI slots [22]	25 W	2	50 W
Motherboard	25W	1	25 W
Fan	10 W	1	10 W
System Total			213 W

Table 1: Component peak power breakdown for a typical server

12 Variable Speed Fans



138 Air Dampers



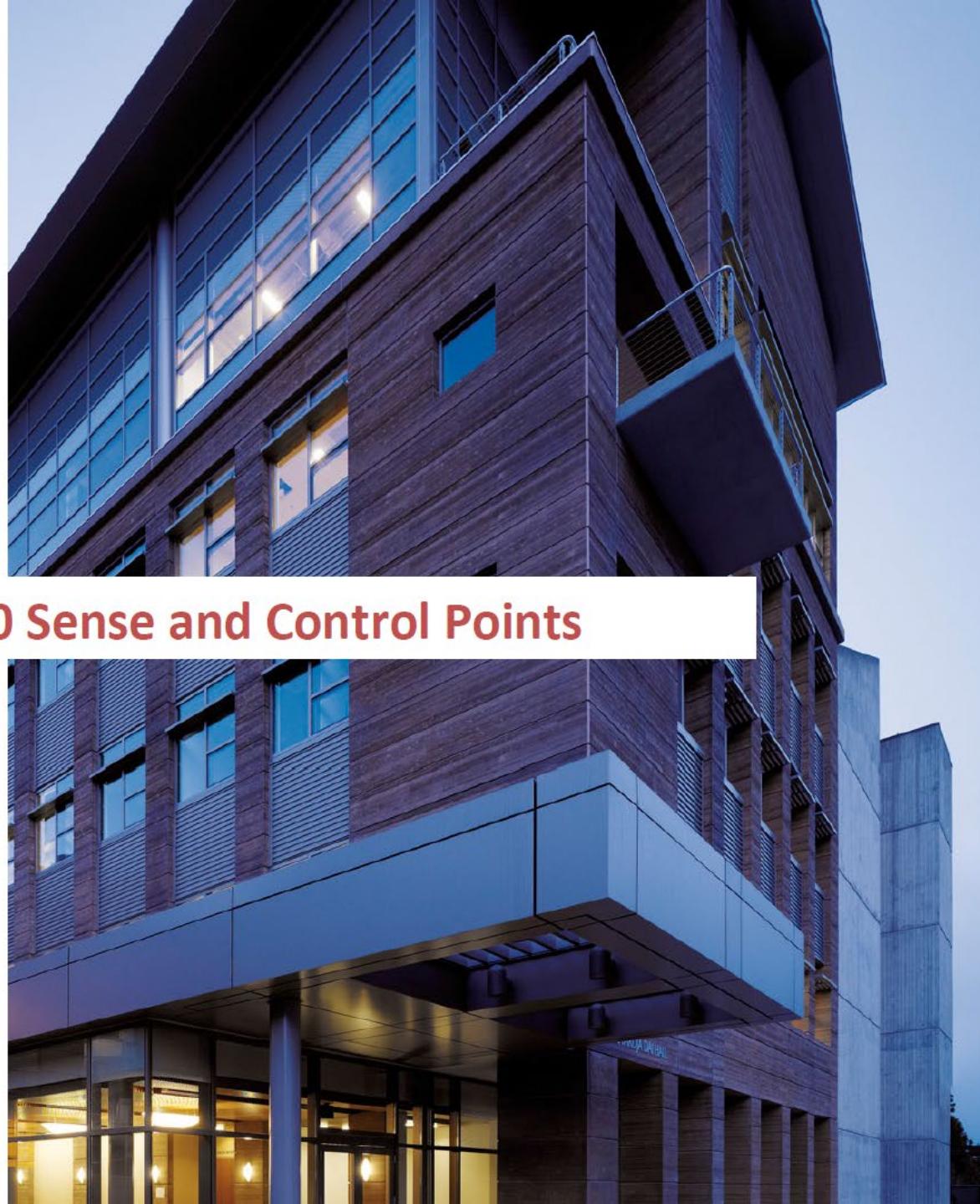
312 Light Relays



50 Electrical Sub-meters



151 Temperature Sensors



> 6,000 Sense and Control Points

Why Operating System Matters?

Dynamic voltage and frequency scaling

Adjust the frequency of CPU according to workload

Power off devices to save power

Like disk, LCD, or even memory

Problem: energy consumption doesn't scale with workloads

Static power, leakage power

New!: Near Threshold Voltage

Issue#4: Mobility

Smartphone OS

Symbian

Windows Mobile

RIM Blackberry OS

Apple iOS

Google Android

Palm WebOS

Windows Phone 7

Android Software Stack

Linux kernel

Libraries

Android run time

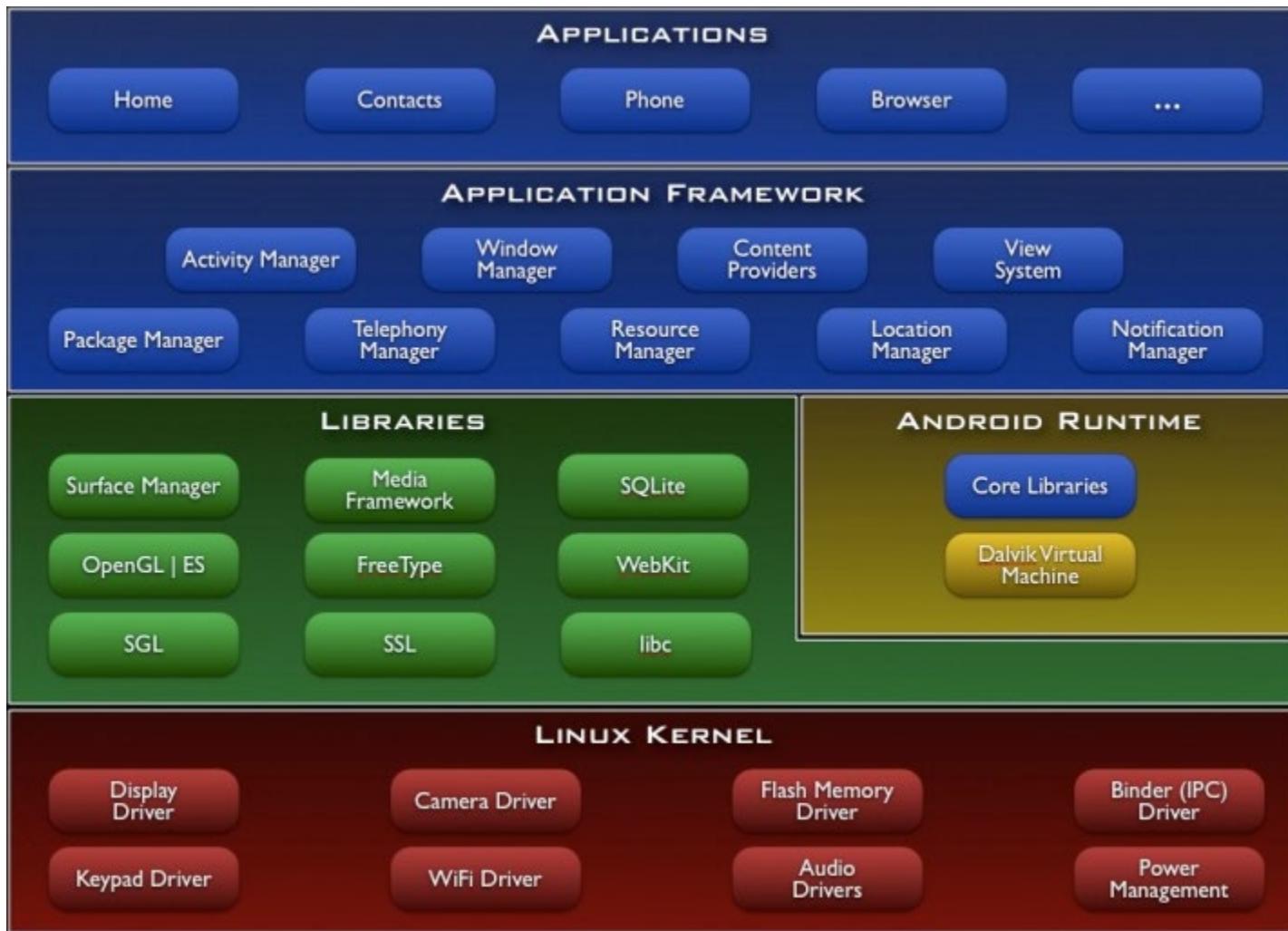
 core libraries

 Dalvik virtual machine

application layer

application protocol

Android Architecture



What's Unique for Mobile OS?

Energy Efficiency

Richer User Experience

But more limited resources

Security

More user data put into mobile OSes

Issue#5: Write Correct Parallel Code

Writing concurrent program is really HARD!

Concurrent programs are prone to concurrency bugs

Concurrency bugs have notorious characteristics

- Non-deterministic

- Hard to test and diagnose

But in multi-core era, we need to write concurrent program to harness the power of multi-core

Quote from Dijkstra

I am convinced more than ever that this type of work is very difficult, and that every effort to do it with other than the best people is doomed to either failure or moderate success at enormous cost.

Edsger Dijkstra
The Structure of the “THE” –Multiprogramming System
1968

Easiest One

Thread 1

Thread 2

```
if (thd->proc_info) {  
    ...  
    thd->proc_info = NULL;  
  
    fputs(thd->proc_info, ...)  
  
    ...  
}
```

MySQL ha_innodb.hpp

Approaches to addressing the problem

Concurrency bug detection

- Race detection

- Atomicity violation detection

- Deadlock bug detection

Concurrent program testing

- Exhaustive testing

- Different coverage criteria proposed

Concurrent programming language/model design

- Transactional memory

What OS can do?

Reduce sources of non-determinism

Tolerate application bugs

Faithfully capture concurrency bugs for reproducing

Provide better programming interfaces to ease
programming

Operating transactions

Issue#6: Scale Out (use distributed systems)

The Datacenter as a Computer

*An Introduction to the Design of
Warehouse-Scale Machines*

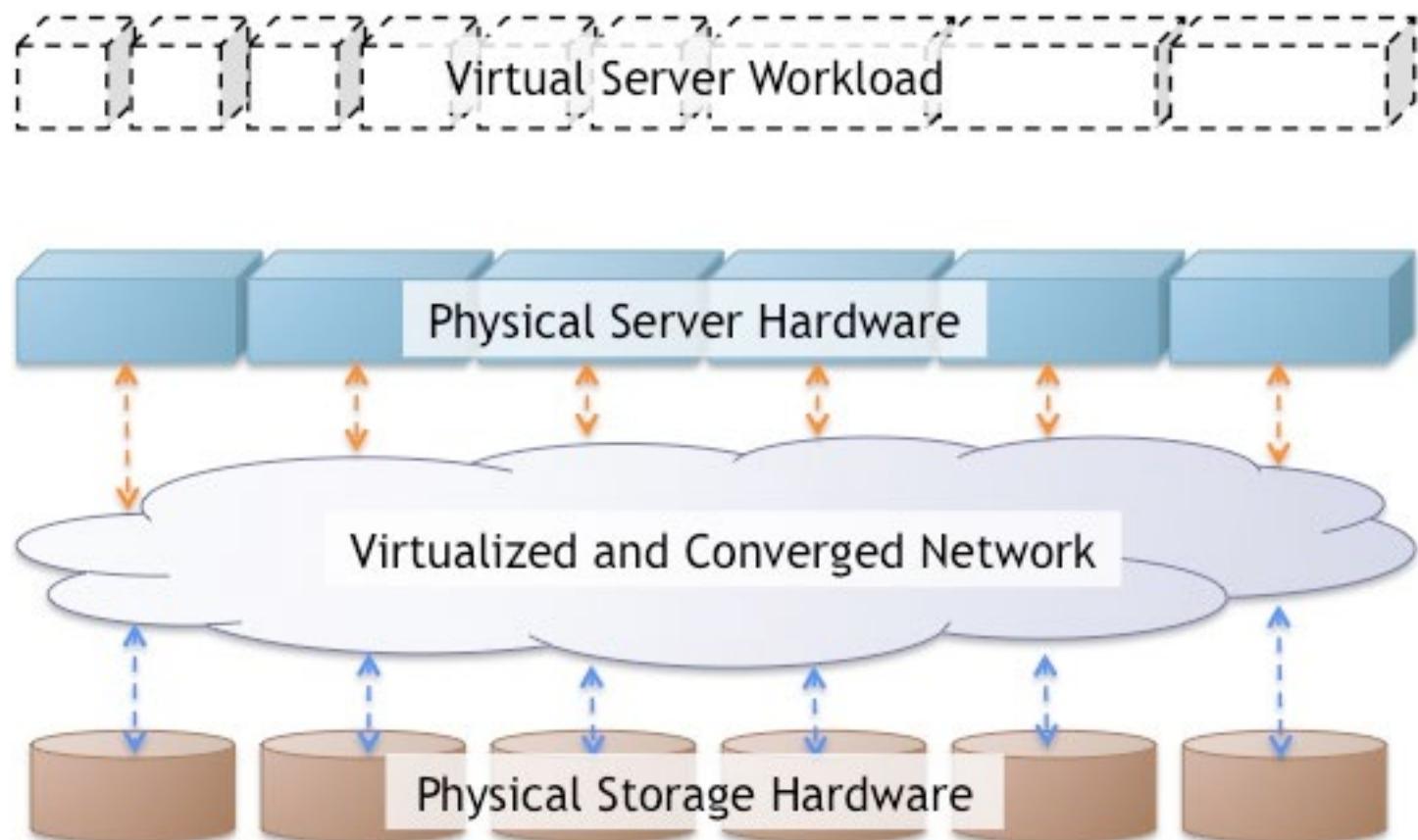
Luiz André Barroso and Urs Hözle
Google Inc.

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE # 6

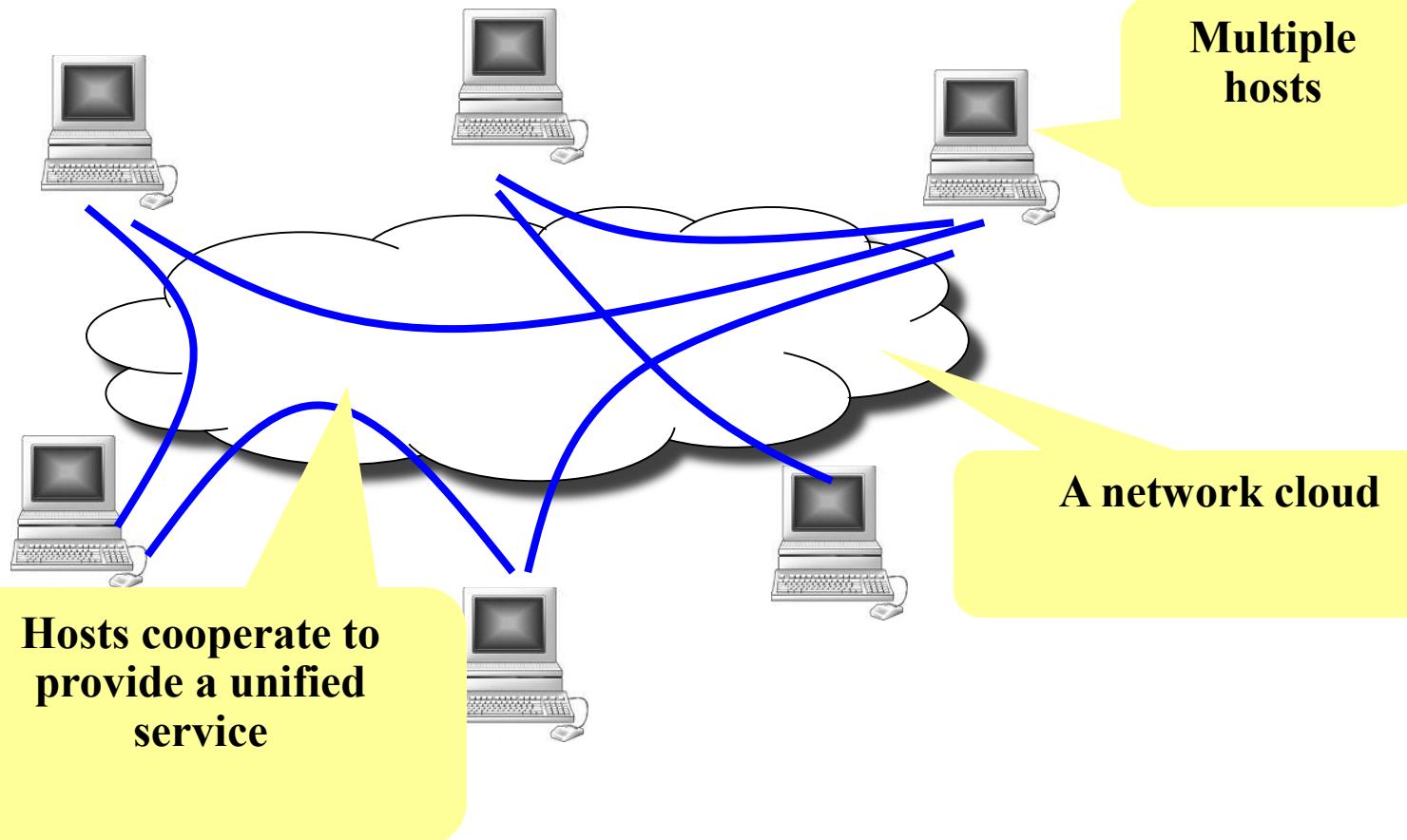


MORGAN & CLAYPOOL PUBLISHERS

The Virtual Datacenter Dream



What are distributed systems?



- Examples?

Why distributed systems? for ease-of-use

Handle geographic separation

Provide users (or applications) with location transparency:

Web: access information with a few “clicks”

Network file system: access files on remote servers as if they are on a local disk, share files among multiple computers

Why distributed systems? for availability

Build a reliable system out of unreliable parts

Hardware can fail: power outage, disk failures, memory corruption, network switch failures...

Software can fail: bugs, mis-configuration, upgrade ...

To achieve 99.999% availability, replicate data/computation on many hosts with automatic failover

Why distributed systems? for scalable capacity

Aggregate resources of many computers

CPU: Dryad, MapReduce, Grid computing

Bandwidth: Akamai CDN, BitTorrent

Disk: Frangipani, Google file system

Why distributed systems? for modular functionality

Only need to build a service to accomplish a single task well.

Authentication server

Backup server

Challenges

System design

What is the right **interface** or abstraction?

How to partition functions for scalability?

Consistency

How to share data consistently among multiple readers/writers?

Fault Tolerance

How to keep system available despite node or network failures?

Challenges (continued)

Different deployment scenarios

- Clusters

- Wide area distribution

- Sensor networks

Security

- How to authenticate clients or servers?

- How to defend against or audit misbehaving servers?

Implementation

- How to maximize concurrency?

- What's the bottleneck?

- How to reduce load on the bottleneck resource?

A word of warning

A distributed system is a system in which I can't do my work because some computer that I've never even heard of has failed.”

-- Leslie Lamport

Issue#7: Non-Volatile Storage

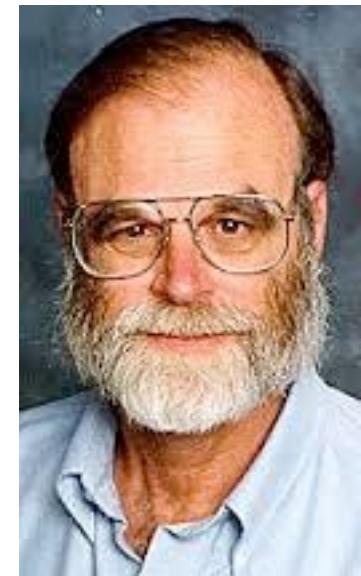
Tape is Dead Disk is Tape Flash is Disk RAM Locality is King

Jim Gray

Microsoft

December 2006

In memory of Jim Gray



Tape Is Dead Disk is Tape

1TB disks are available

10+ TB disks are predicted in 5 years

Unit disk cost: ~\$400 → ~\$80

But: ~ 5..15 hours to read (sequential)

~15..150 days to read (random)

Need to treat **most of disk** as
Cold-storage archive

FLASH Storage?

1995 16 Mb NAND flash chips

2005 16 Gb NAND flash

Doubled each year since 1995

Market driven by Phones, Cameras, iPod,...

Low entry-cost,

~\$30/chip → ~\$3/chip

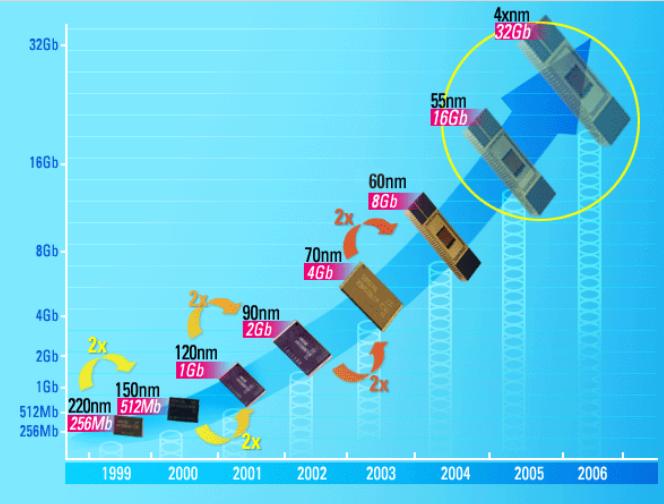
2012 1 Tb NAND flash

== 128 GB chip

== 1TB or 2TB “disk”
for ~\$400

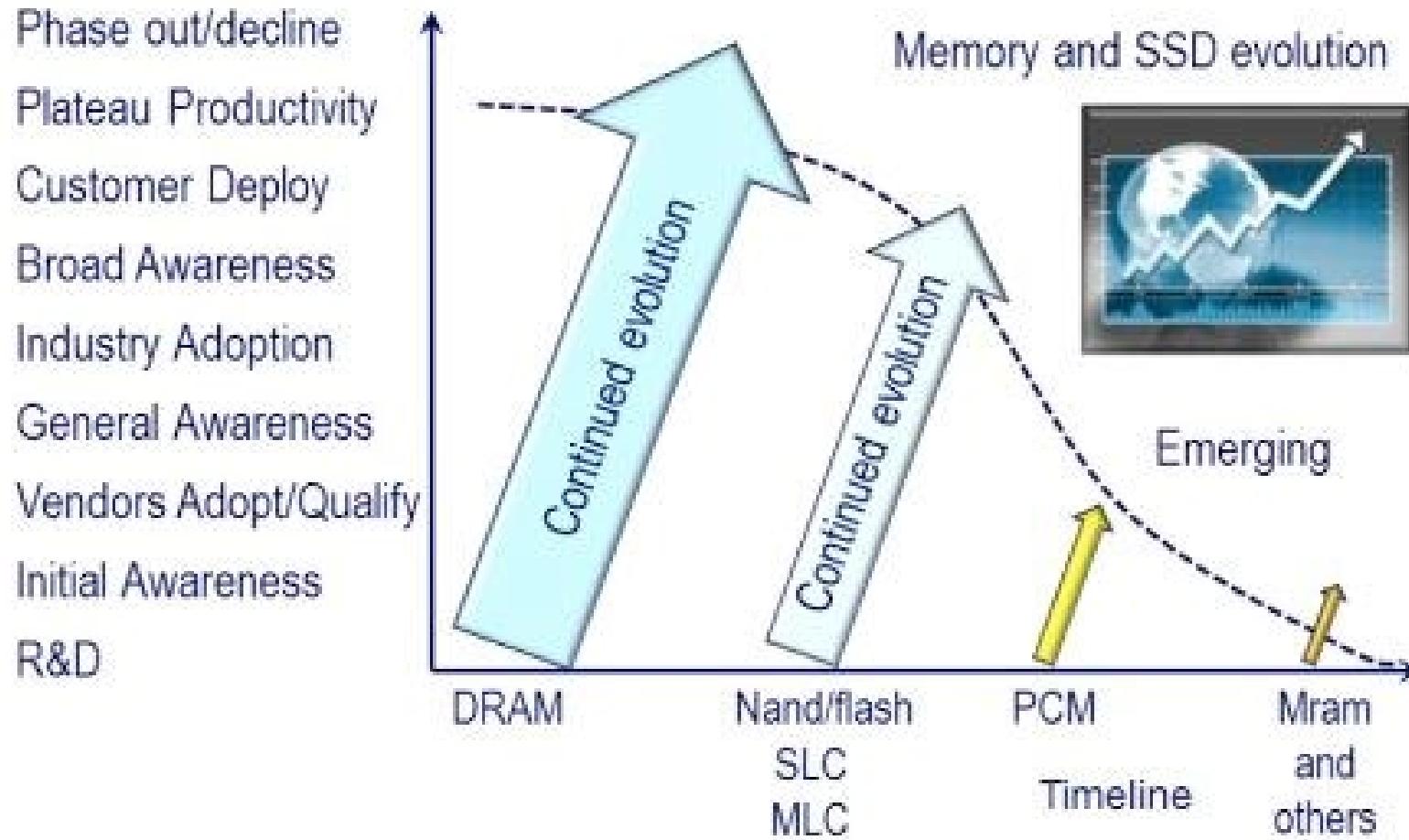
or 128GB disk for \$40

or 32GB disk for \$5



Samsung prediction

Storage Hiereary

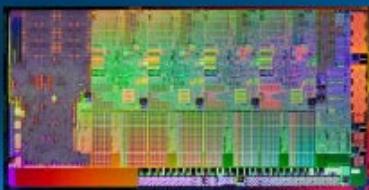


New Storage Keeps Coming

3D XPoint™ TECHNOLOGY

SRAM

Latency: 1X
Size of Data: 1X



DRAM

Latency: ~10X
Size of Data: ~100X

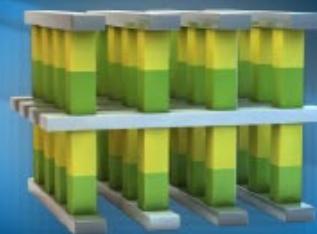


MEMORY

STORAGE

3D XPoint™

Latency: ~100X
Size of Data: ~1,000X



NAND

Latency: ~100,000X
Size of Data: ~1,000X



HDD

Latency: ~10 MillionX
Size of Data: ~10,000 X



What can OS do?

Needs to adapt to the new storage model

Non-volatility changes the way operating system manage resources

- Like crash recovery

- Recall fsck

Better I/O performance

Issue#8: Virtualization

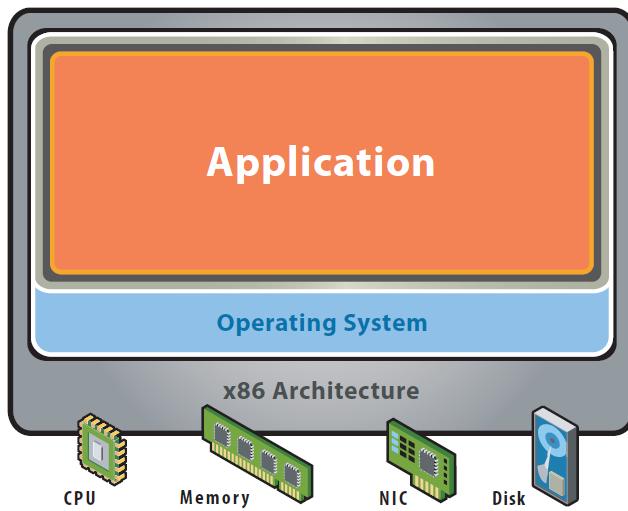
*“Any problem in computer science can
be solved with another level of
indirection.”*

– *David Wheeler in Butler Lampson’s
1992 ACM Turing Award speech.*

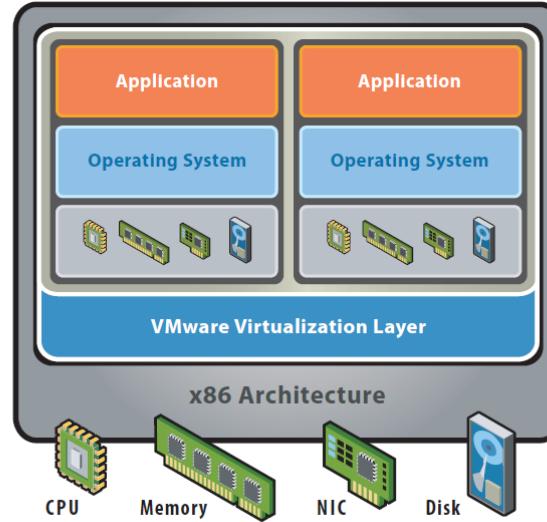
Concept of Virtualization

“Virtualization” refers to the creation of a virtual machine

A virtual machine is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine



Before Virtualization



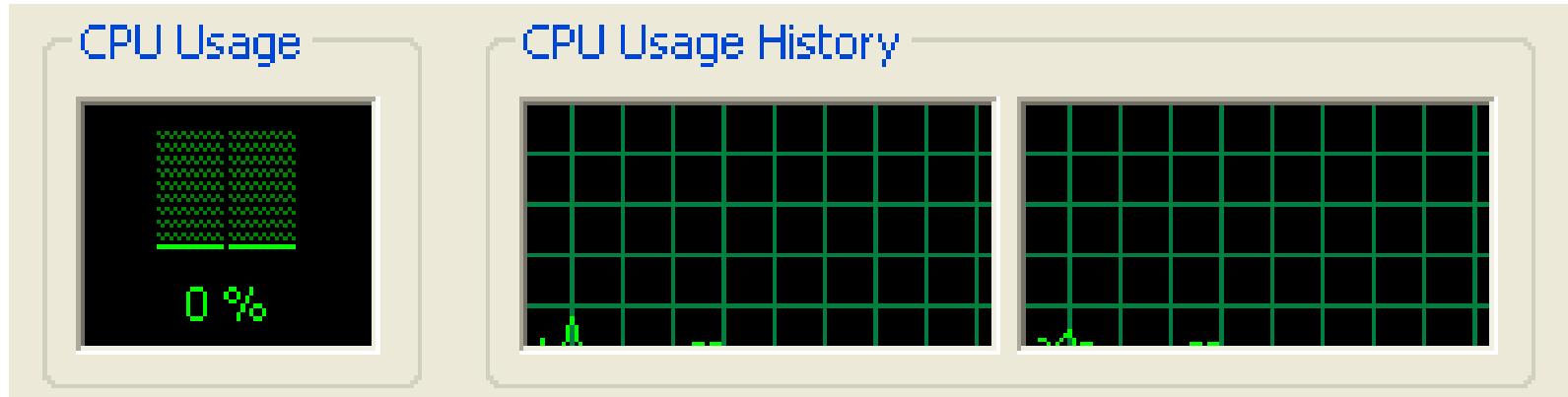
After Virtualization

Why Virtualization ?

- Decoupling the operating systems from physical hardware supports
 - Maximum resource utilization
 - Multiple OS environments can co-exist on the same computer, in strong isolation from each other
 - Server consolidation
 - Fault tolerance
 - Fault and migrate
 - Portability
 - Can use any OS
 - Manageability
 - Easy to maintain

Why Virtualize?

Too many servers for too little work



High costs and infrastructure needs

Maintenance

Networking

Floor space

Cooling

Power

Disaster Recovery



Virtualization is Essentially Another Layer of OS

How does the process and OS use hardware resources?

	process	OS
CPU	Non-privileged registers and instructions	+Privileged registers and instructions
memory	Virtual memory	+Physical memory
exceptions	Signals, errors	+Traps, interrupts
I/O	File system	Programmed I/O, DMA, interrupts

Outline

Why we study OS?

8 problems of OS

Course information

Faculty Information

Instructor

CHEN Haibo 陈海波

XIA Yubin 夏虞斌

WANG Zhaoguo 王肇国

Office: Room 3-401, Software Building

Contact:

haibochen@sjtu.edu.cn xiayubin@sjtu.edu.cn,
wangzhaoguo@sjtu.edu.cn

TA

LI Dingji dj_lee@foxmail.com 18037910004

LI Haoyu haoyu.li@sjtu.edu.cn 17621808964

LI Zinan rododo@sjtu.edu.cn 13162578896

Website



Reference Books

xv6: a simple, Unix-like teaching operating system.

By Russ Cox, Frans Kaashoek, Robert Morris.
Online

Understanding Linux Kernel.

Next Class

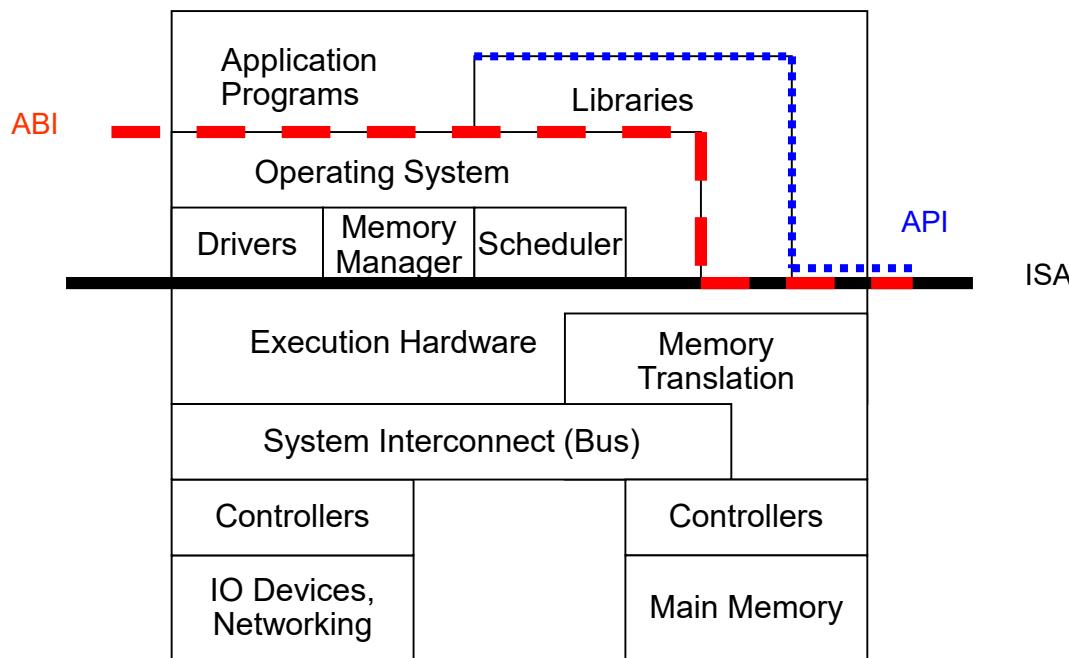
OS architecture

Recall

- What is OS from a programmer's perspective?
 - A set of API, aka, system calls
 - E.g., int 0x80
- What are differences from other APIs?
 - The system calls are usually hardware-related
 - E.g., fork, FILE ops, socket, select/epoll
- Who invokes the system calls?
 - Application itself, libraries (e.g., glibc)
 - Application through the libraries

Software Stack

- Architecture
 - Functionality and Appearance of a computer system but not implementation details
 - Level of Abstraction = Implementation layer
 - ISA, ABI, API

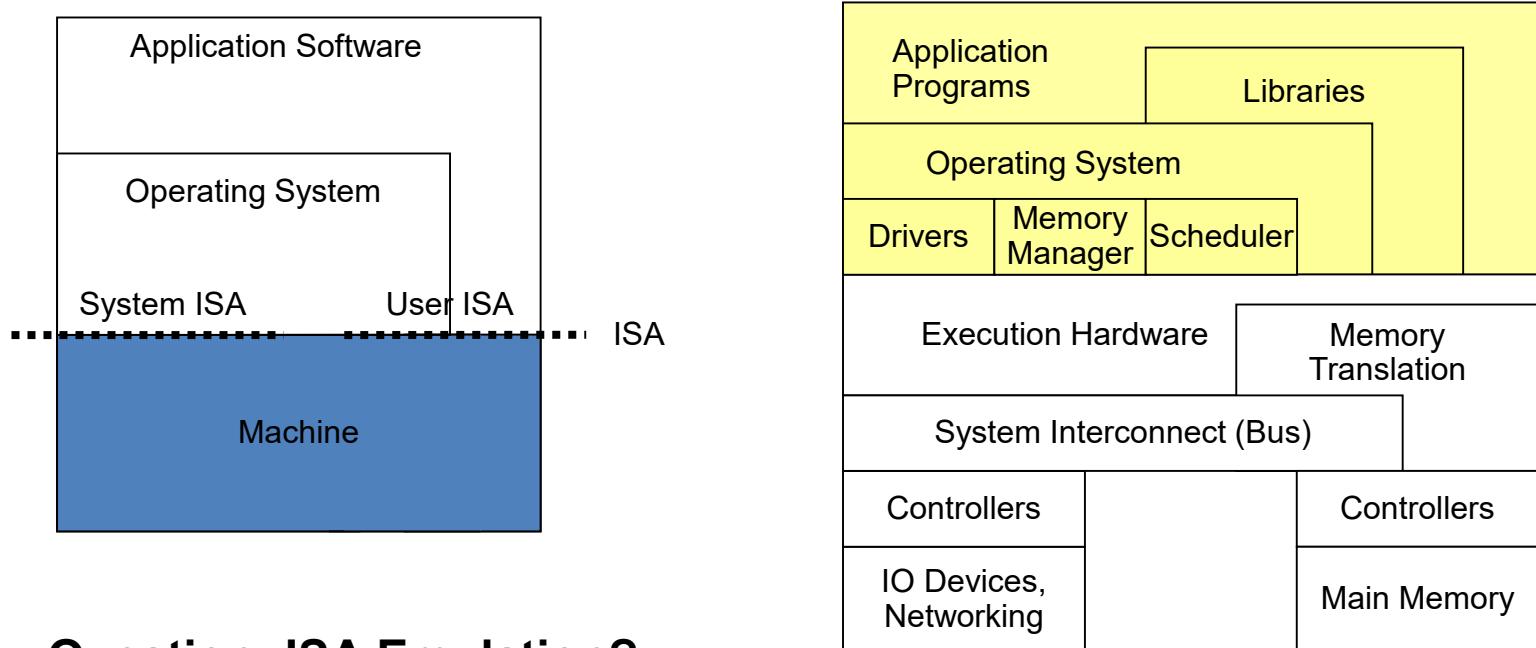


Architecture, Implementation Layers

- Implementation Layer : ISA
 - Instruction Set Architecture
 - Divides hardware and software
 - Concept of ISA originates from **IBM 360**
 - IBM System/360 Model (20, 40, 30, 50, 60, 62, 70, 92, 44, 57, 65, 67, 75, 91, 25, 85, 95, 195, 22) : 1964~1971
 - Various prices, processing power, processing unit, devices
 - But guarantee a *software compatibility*
 - User ISA and System ISA

Implement the ISA

- Machine from the perspective of a system
 - ISA provides interface between system and machine



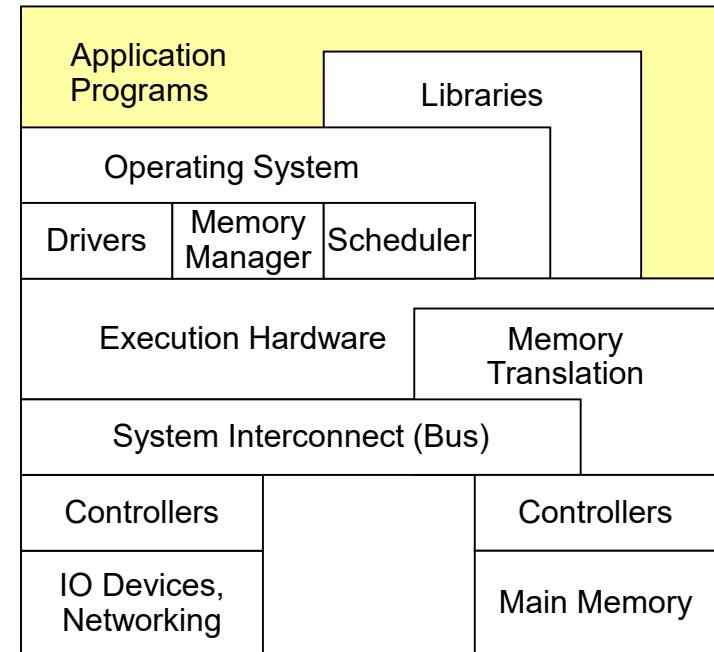
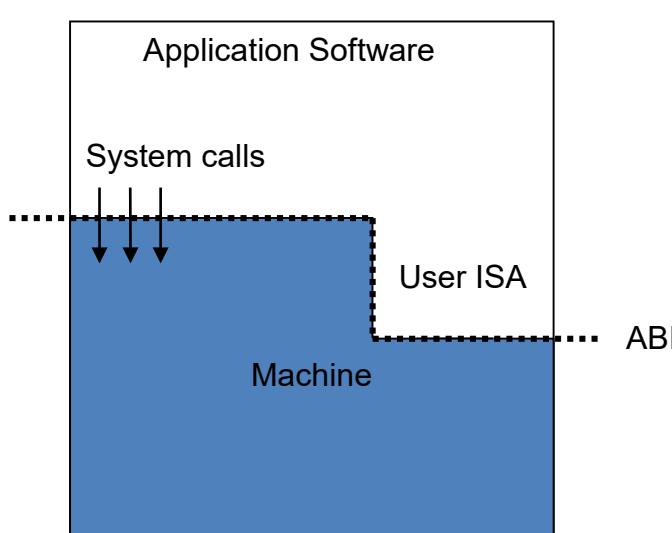
Question: ISA Emulation?

Architecture, Implementation Layers

- Implementation Layer : ABI
 - Application Binary Interface
 - Provides a program with access to the hardware resource and services available in a system
 - Consists of User ISA and System Call Interfaces

Implement the ABI

- Machine from the perspective of a process
 - ABI provides interface between process and machine
 - E.g., ELF in Linux, EXE in Windows



Question: Which software does ABI Emulation?

Architecture, Implementation Layers

- Implementation Layer : API
 - Application Programming Interface
 - Key element is Standard Library (or Libraries)
 - Typically defined at the source code level of High Level Language
 - **Libc** in Unix environment : supports the UNIX/C programming language

OS Structures

Haibo Chen

Review

- Why OS is important?
- 8 important problems in OS

Operating System Design and Implementation

Design and Implementation of OS not “solvable”, but some approaches have proven successful

- Internal structure of different Operating Systems can vary widely

- Start by defining goals and specifications

- Affected by choice of hardware, type of system

Remember “Worse is better design”

Operating System Design and Implementation

User goals and System goals

User goals

operating system should be convenient to use, easy to learn, reliable, safe, and fast

System goals

operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient

Operating System Design and Implementation (Cont.)

Important principle to separate

Policy: What will be done?

Mechanism: How to do it?

Mechanisms determine how to do something

Policies decide what will be done

The separation of policy from mechanism is a very important principle

It allows maximum flexibility if policy decisions are to be changed later

Kernels

Kernel classification (wikipedia)

Kernel (architecture)	Monolithic	Microkernel	Exokernel	hybrid
---------------------------------	------------	-------------	------------------	--------

Monolithic kernel

A **monolithic kernel** is an operating system architecture where the entire operating system is working in kernel space and is alone in supervisor mode.

Linux, BSD...

Kernels

Microkernel

is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system (OS).

low-level address space management, thread management, and inter-process communication (IPC) and so on ...

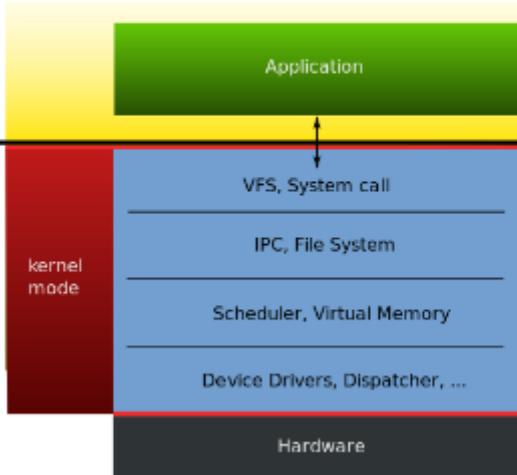
Mach, L4 kernel

Hybrid kernel

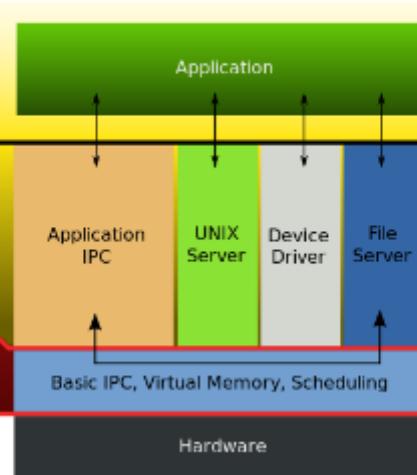
is a kernel architecture based on combining aspects of microkernel and monolithic kernel architectures used in computer operating systems.

Windows NT kernel

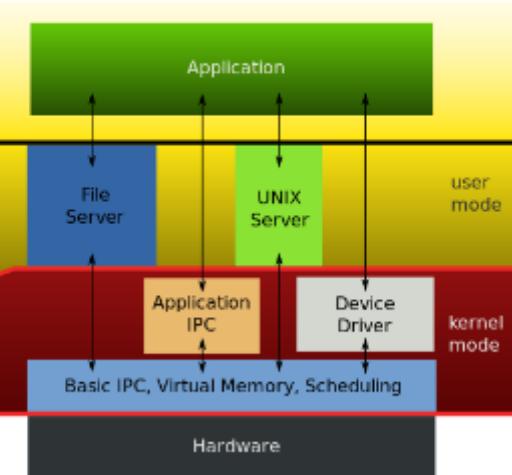
Monolithic Kernel
based Operating System



Microkernel
based Operating System



"Hybrid kernel"
based Operating System



Simple Structure

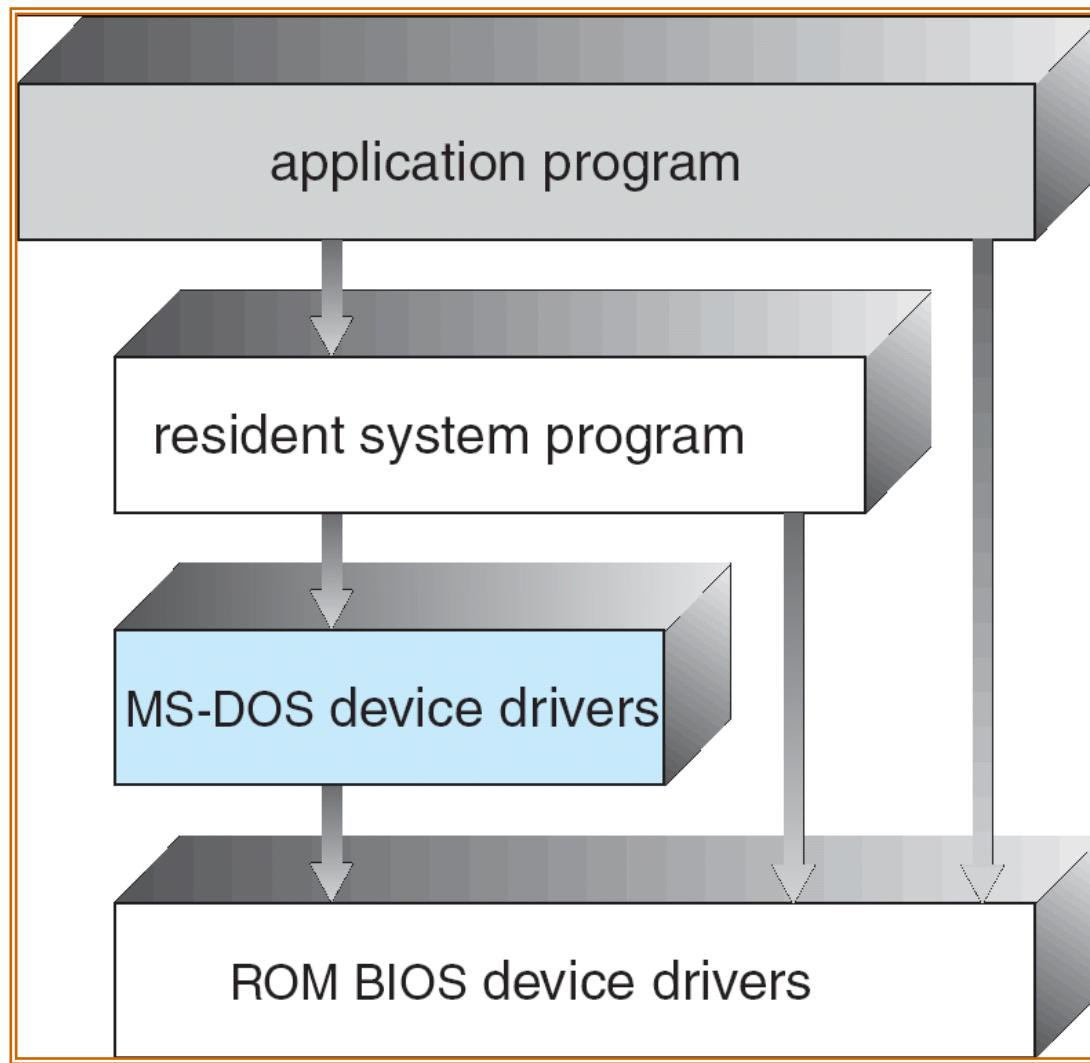
MS-DOS – written to provide the most functionality in the least space (1981~1994)

Not divided into modules

Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



MS-DOS Layer Structure



Layered Approach

The operating system is divided into a number of layers (levels)

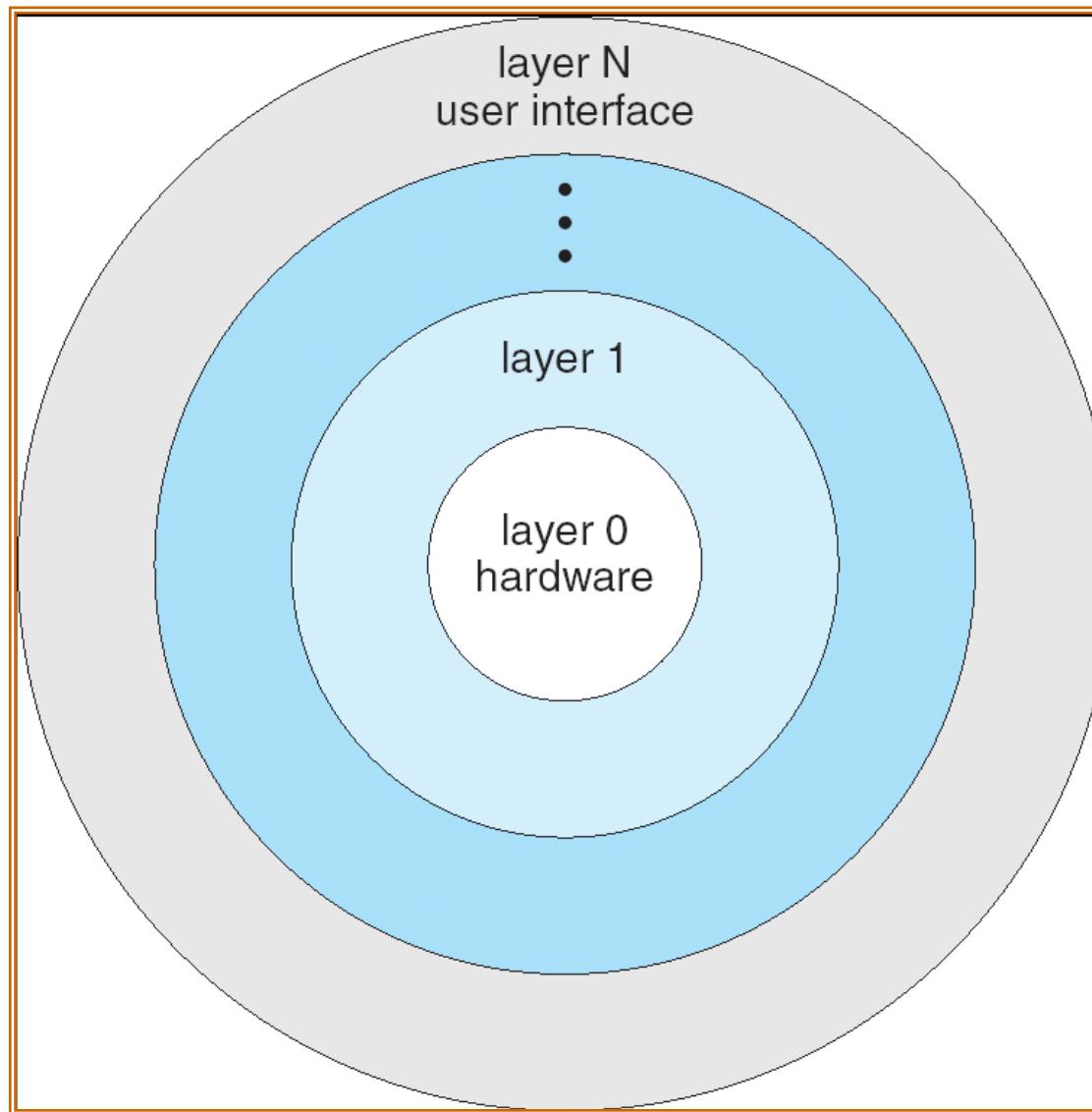
- Each built on top of lower layers

- The bottom layer (layer 0), is the hardware

- The highest (layer N) is the user interface

With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers

Layered Operating System



UNIX

UNIX – limited by hardware functionality

the original UNIX operating system had limited structuring.

The UNIX OS consists of two separable parts

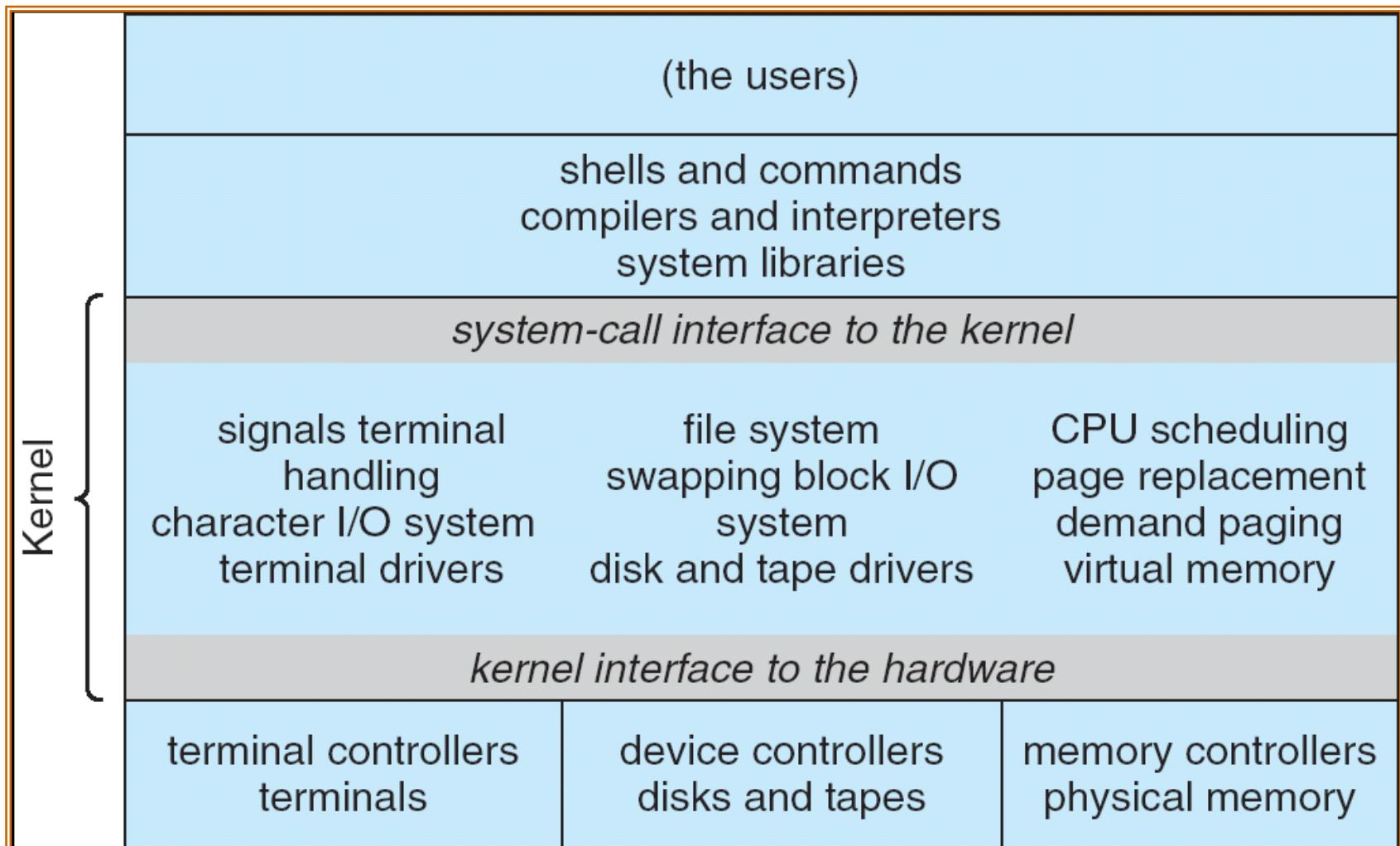
Systems programs

The kernel

Consists of everything below the system-call interface and above the physical hardware

Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

UNIX System Structure



Microkernel System Structure

Moves as much from the kernel into “*user*” space

Communication takes place between user modules using message passing

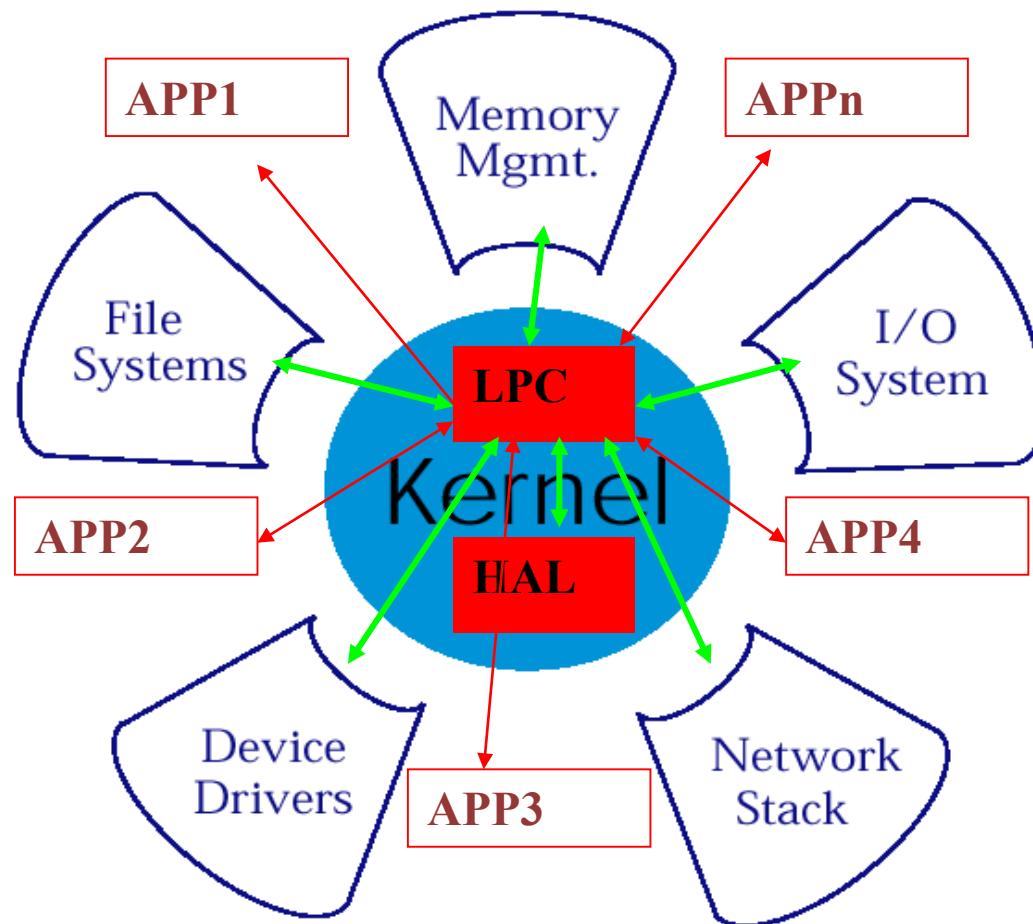
Benefits:

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Detriment:

- Performance overhead of user space to kernel space communication

Microkernel Structure & Instances



Mach

- Developed at CMU
- Led by Rick Rashid
 - Now VP of research at Microsoft
- Initial release: 1985
- Big impact (as we will see)



Rick Rashid

What does a microkernel (Mach) do?

- Task and thread management:
 - Task (process) unit of allocation
 - Thread, unit of execution
 - Implements CPU scheduling: exposed to apps
 - Applications/environments can implement their own scheduling policies
- Interprocess communication (IPC)
 - Between threads via ports
 - Secured by capabilities

What does a microkernel (Mach) do?

- Memory object management:
 - Essentially virtual memory
 - Persistent store accessed via IPC
- System call redirection:
 - Enable to trap system calls and transfer control to user mode
 - Essentially enable applications to modify/extend the behavior and functionality of system calls, e.g.,
 - Enable binary emulation of environments, tracing, debugging

What else does a microkernel (Mach) do?

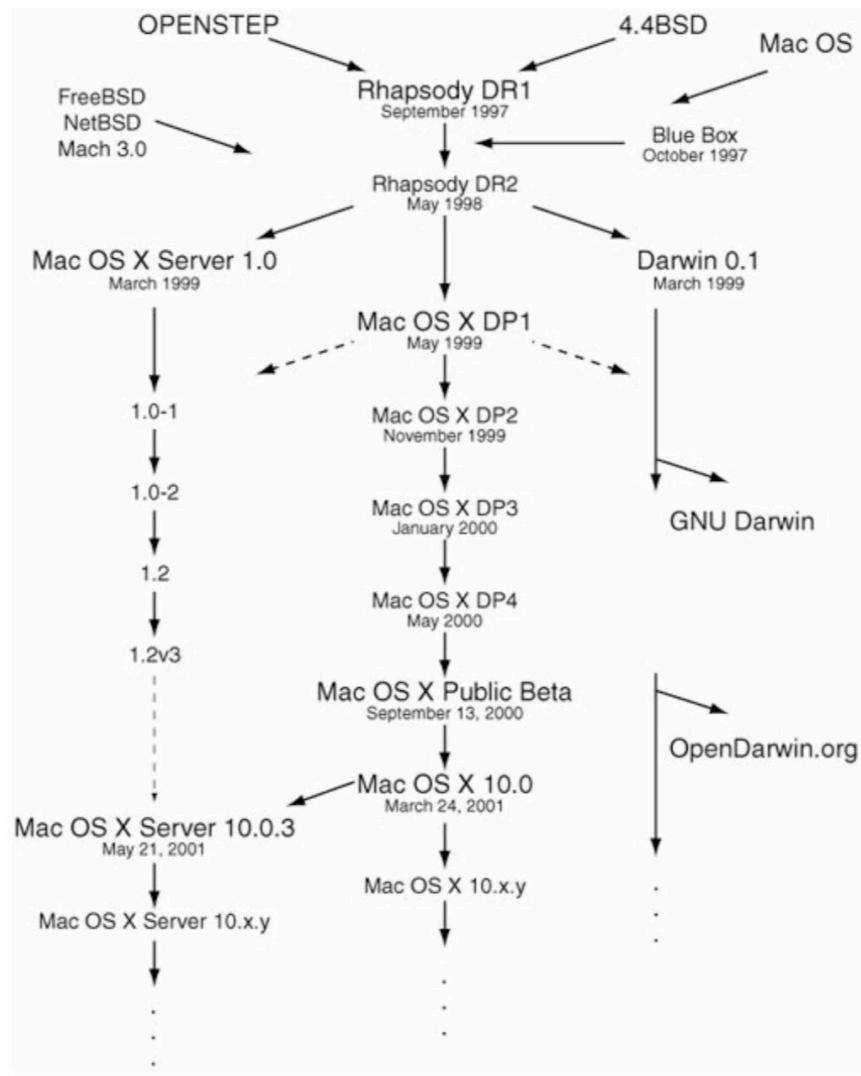
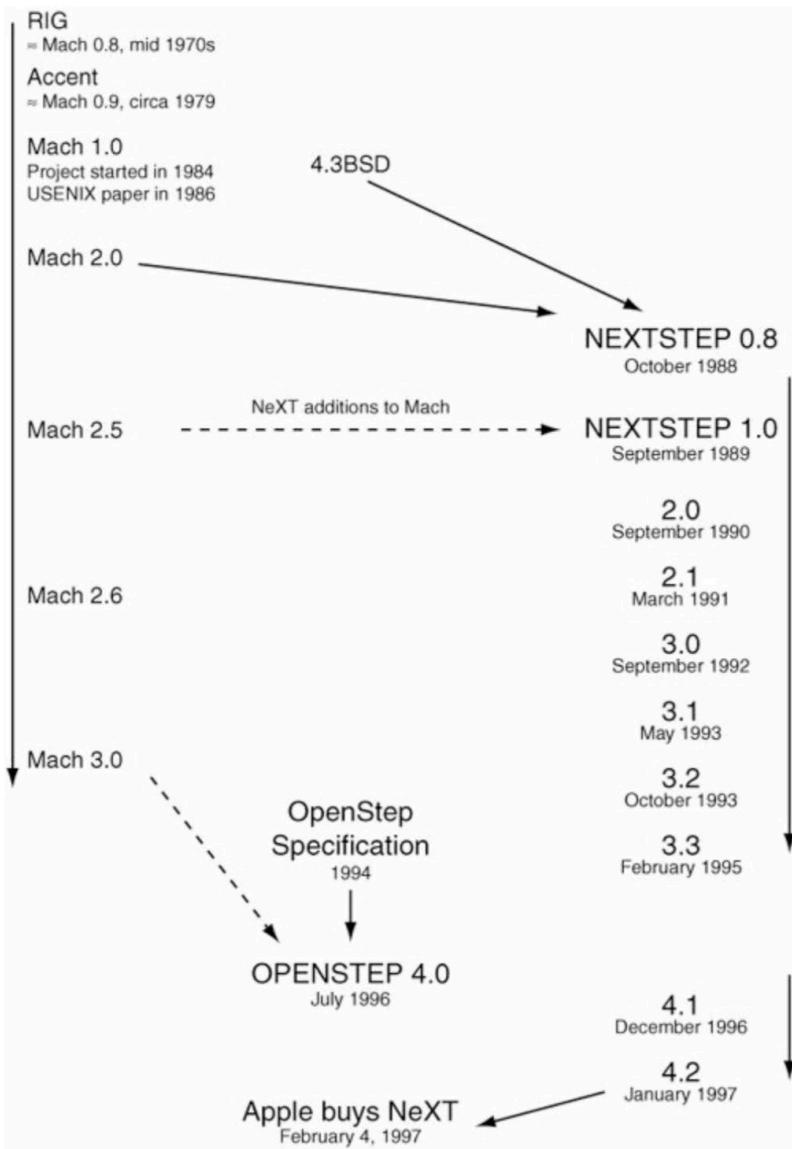
- Device support:
 - Implemented using IPC (devices are contacted via ports)
 - Support both synchronous and asynchronous devices
- User multiprocessing:
 - Essentially a user level thread package, with wait()/signal() primitives
 - One or more user threads can map to same kernel thread
- Multicomputer support:
 - Can map transparently tasks/resources on different nodes in a cluster

Mach 2.5

- Contains BSD code compatibility code, e.g., one-to-one mapping between tasks and processes
- Some commercial success:
 - NeXT
 - Steve Jobs' company after he left Apple
 - Used by Tim Berners-Lee to develop WWW
 - Encore, OSF (Open Software Foundation), ...



The Evolution of MacOS/iOS



Mach 3

- Eliminate BSD code
- Rewrite IPC to improve performance
 - RPC on (then) contemporary workstations: 95 usec
- Expose device interface
- Provide more control to user applications via [continuation](#):
 - Address of an user function to be called when thread is rescheduled plus some data: essentially a [callback](#)
 - Enable application to save restore state, so that the microkernel doesn't need to do it, e.g., saving and restoring register state

OSes and Application Programs

- Mach allows application to implement:
 - Paging
 - Control data cached by virtual memory
 - ...
- Redirection allows call traps to link directly to executable binaries without modifying the kernel!
 - Just need an emulation library

L3 -> L4

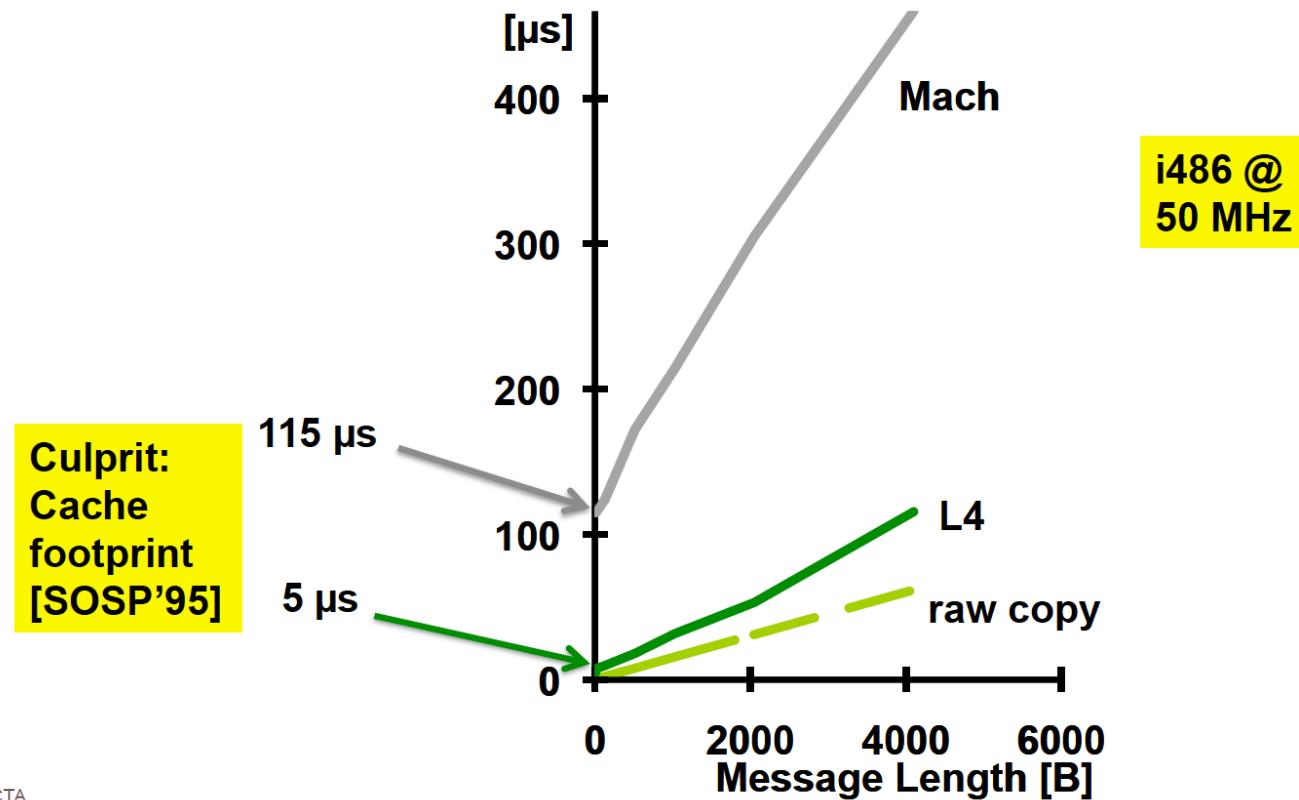
How it started? (1993)

- Microkernels (e.g., Mach) still too slow
 - Mostly because IPCs
- Tide was turning towards monolithic kernels
- **Jochen Liedtke** (GMD – Society for Mathematics and Information technology) aimed to show that IPC can be super-fast



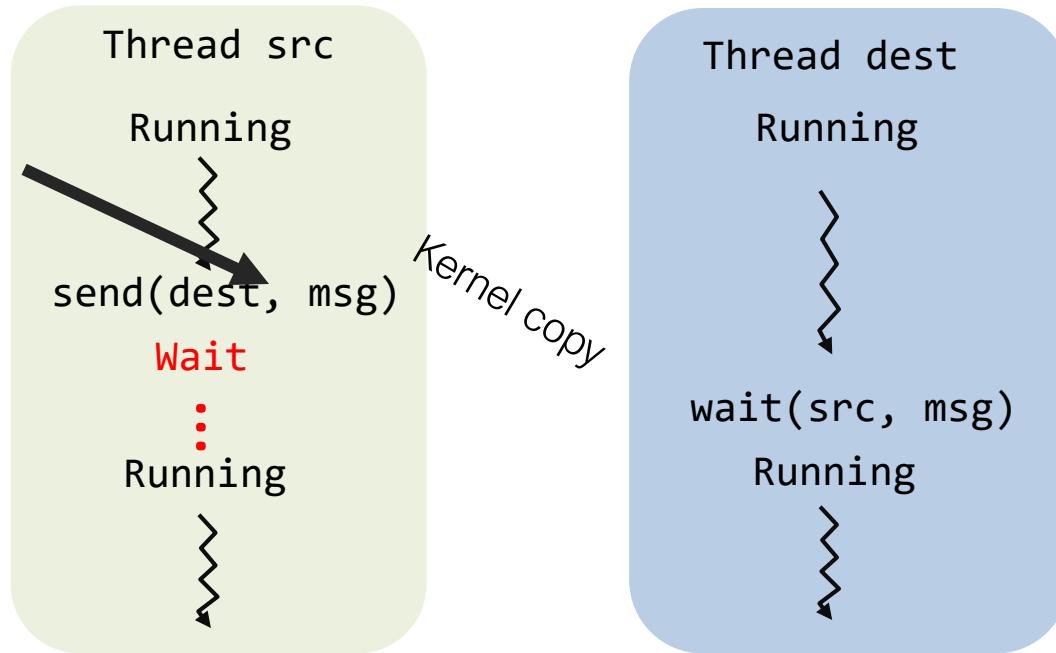
Jochen Liedtke

How fast?



How did he do it?

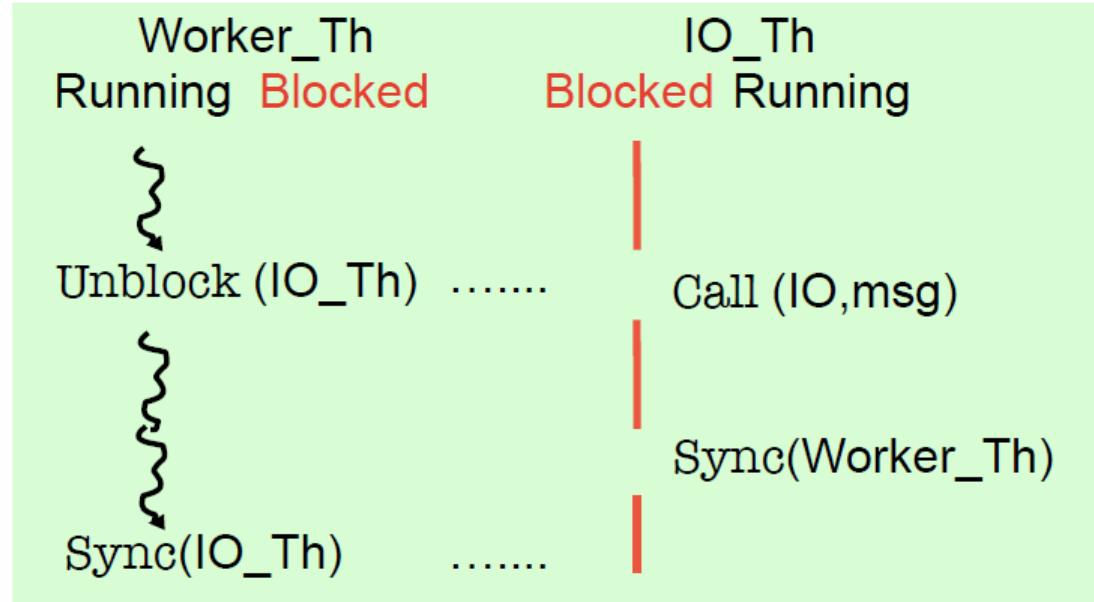
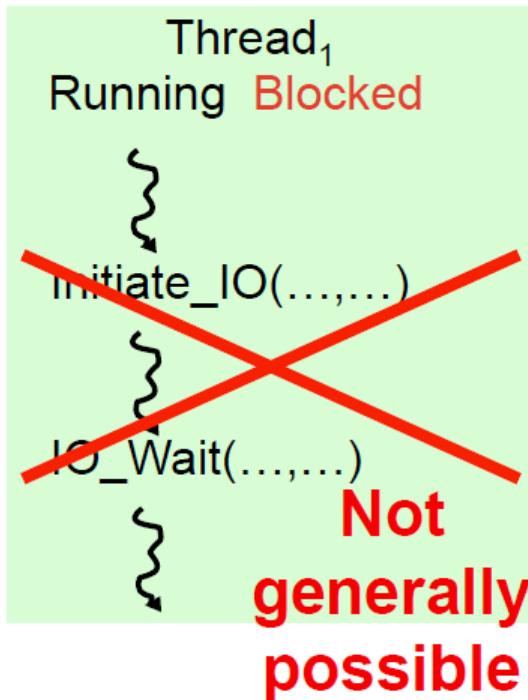
- Synchronous IPC → Rendezvous model



Kernel executes in sender's context

- Copies memory data directly to receiver (**single-copy**)
- Leaves message registers unchanged during context switch (**zero copy**)

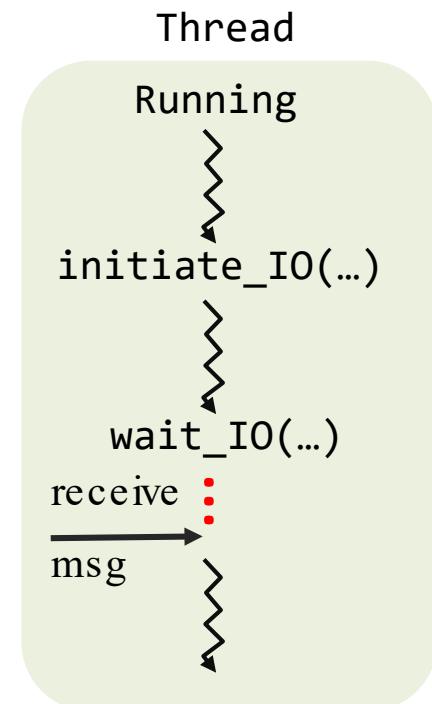
Synchronous IPC Issues



Sync IPC forces multi-threading model

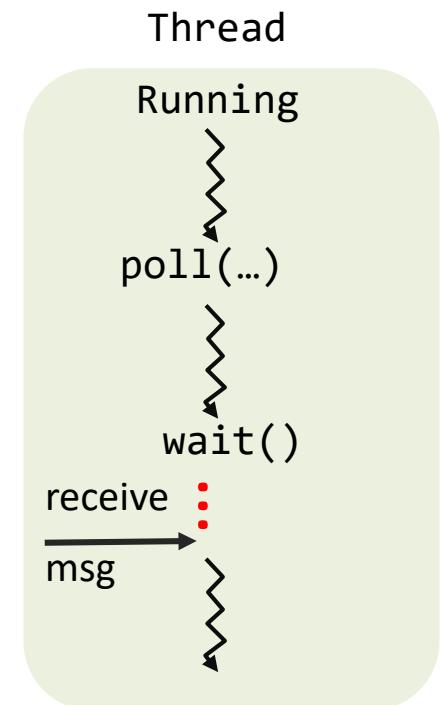
Asynchronous IPCs

- Disadvantages of synchronous IPCs
 - Have to block on I/O operations
 - Forces apps to use multithreading
 - Poor choice for multicores (no need to block if IO executes on another core!)
- Want async IPCs
 - Want something like `select()`/`poll()`/`epoll()` in Unix



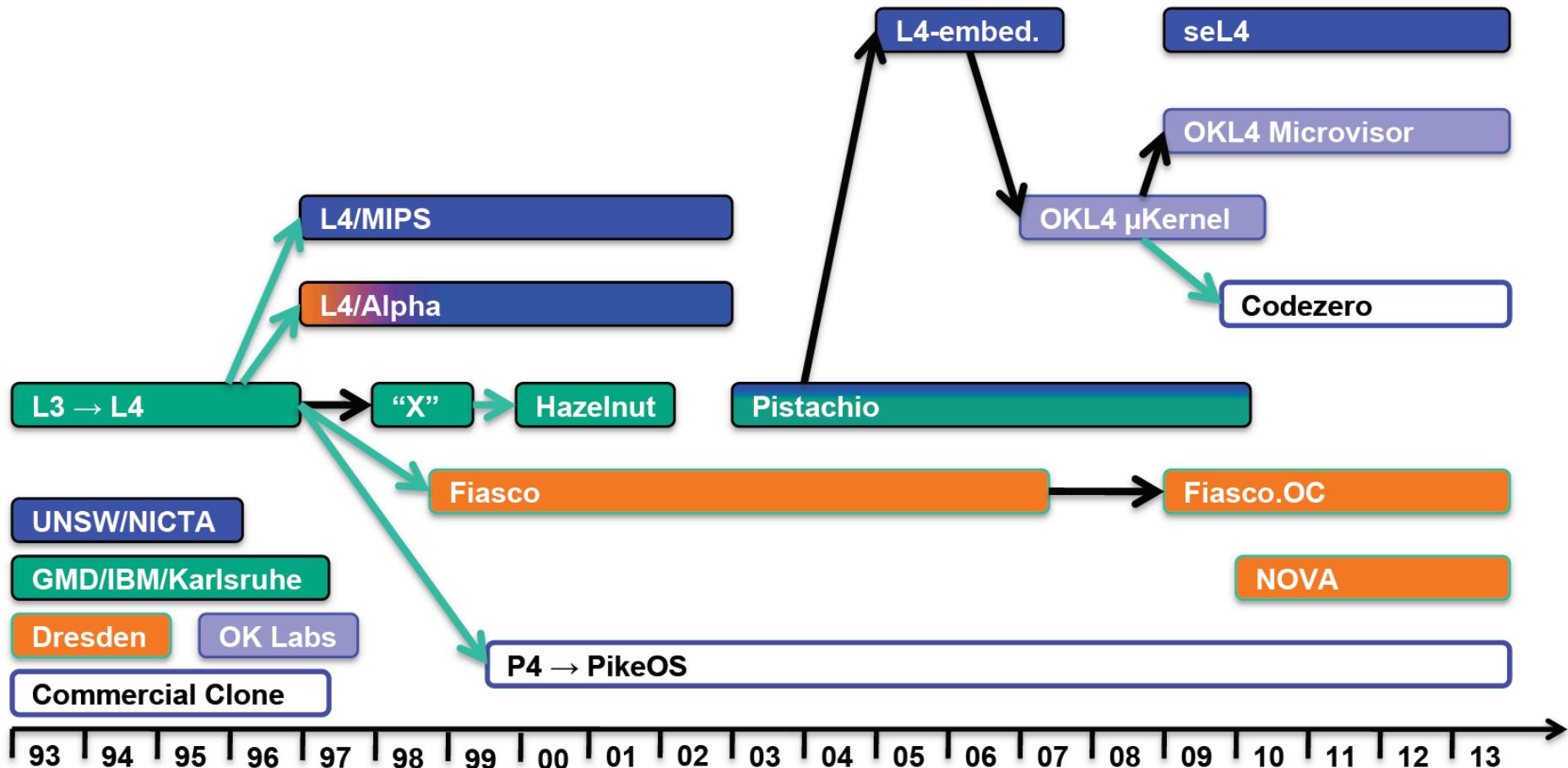
Async notifications

- Sending is non-blocking and asynchronous
- Receiver, who can block or poll for message
- seL4: Asynchronous Endpoints (AEP)
 - Single-word notification field
 - Send sets a bit in notification field
 - Bits in notification field are ORed → notification
 - wait(), effectively select() across notification fields

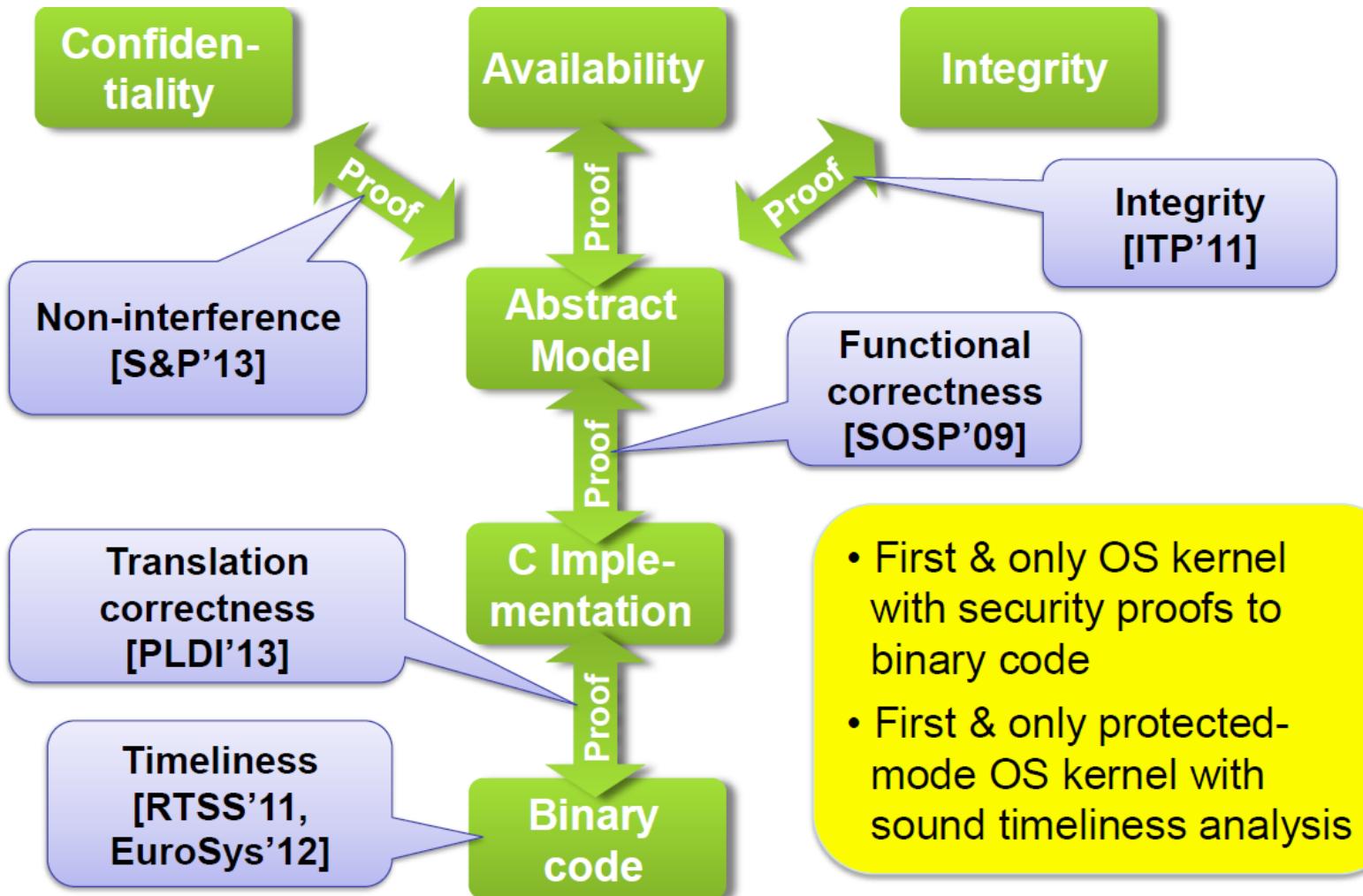


Added async notifications to complement syn IPCs

L4 family tree



seL4: Unprecedented Dependability



So why didn't take over entire world!

- Hardware standardization:
 - Intel and ARM dominating
 - Less need for portability, one of main goals of Mach
- Software standardization:
 - Windows, MacOS/iOS, Linux/Android
 - Less need to factor out common functionality
- Maybe just a fluke?
 - Linux could have been very well adopted the microkernel approach
 - Philosophical debate between Linus and Andy Tanembaum
 - One of Linus main arguments: there is only i386 I need to write code for!

(<http://www.oreilly.com/openbook/opensources/book/appa.html>)

EXOKERNEL

Exokernel Structure

Overview

let the kernel allocate the physical resources of the machine to multiple application programs, and

let each program decide what to do with these resources.

The program can link to an operating system library (libOS) that implements OS abstractions.

Exokernel: An operating system architecture for application-level resource management, SOSP'95

Exokernel & Corey

Exokernel [SOSP'95]

Kernel classification

Motivation

Design and Implementation

Evaluation

Corey [OSDI'08]

Multi-core era and scalability problem

Sharing and Address range

Summary

Motivation

Exokernel's Motivation

In traditional operating systems, only privileged servers and the kernel can manage system resources

Un-trusted applications are required to interact with the hardware via some abstraction model

File systems for disk storage, *virtual address spaces* for memory, etc.

Application demands vary widely!!

An interface designed to accommodate every application must anticipate **all possible needs**

Exokernel's Idea

Give un-trusted applications as **much control** over physical resources as possible

To force as **few abstraction** as possible on developers, enabling them to make as **many decisions** as possible about hardware abstractions.

Let *the kernel* allocate the basic physical resources of the machine

Let *each program* decide what to do with these resources

Exokernel separate *protection* from *management*

They **protect** resources but delegate **management** to application

Library Operating System

Most programs to be linked with *libraries* instead of communicating with the exokernel directly

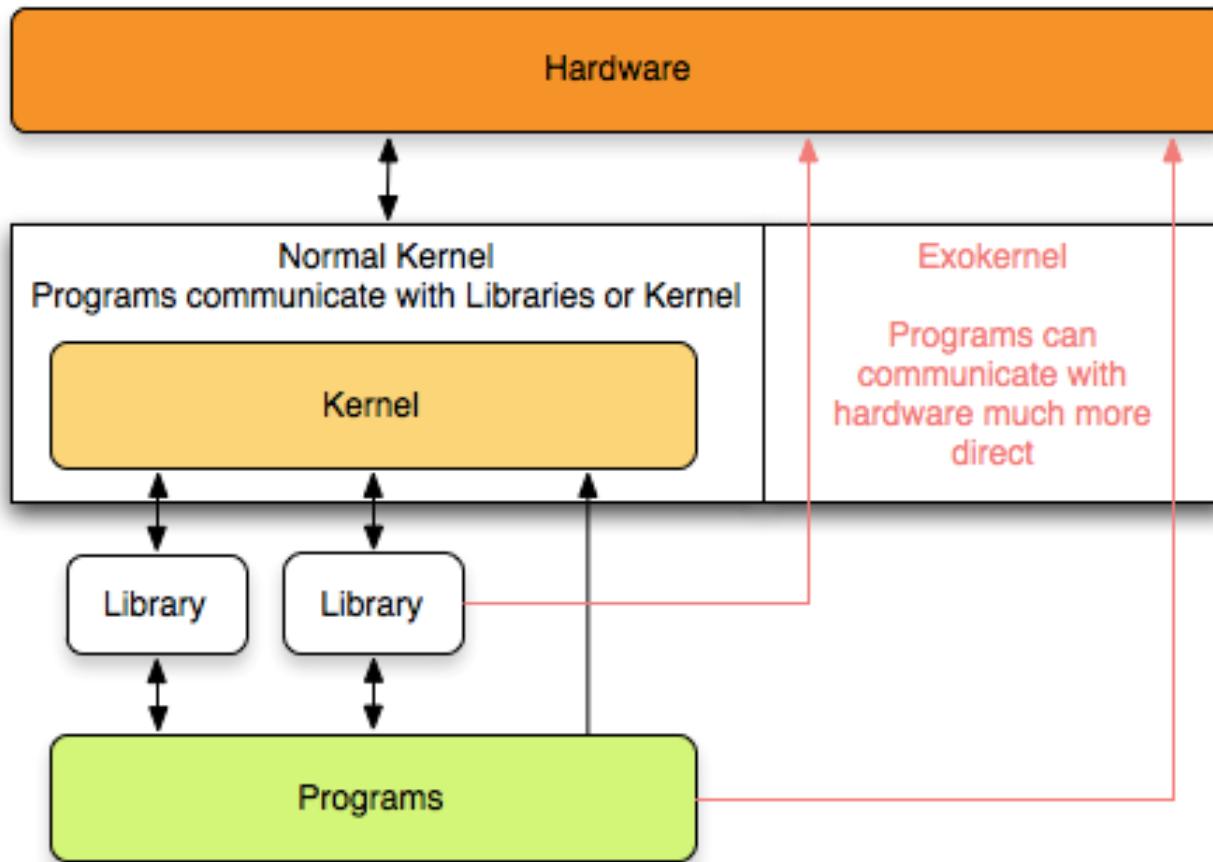
The libraries hide low-level resources

An applications can choose the library which best suits its needs, or even build its own

The kernel only ensures that the requested resource is free, and the application is allowed to access it.

Allow programmer to implement custom abstractions, omit unnecessary ones, most commonly to improve performance

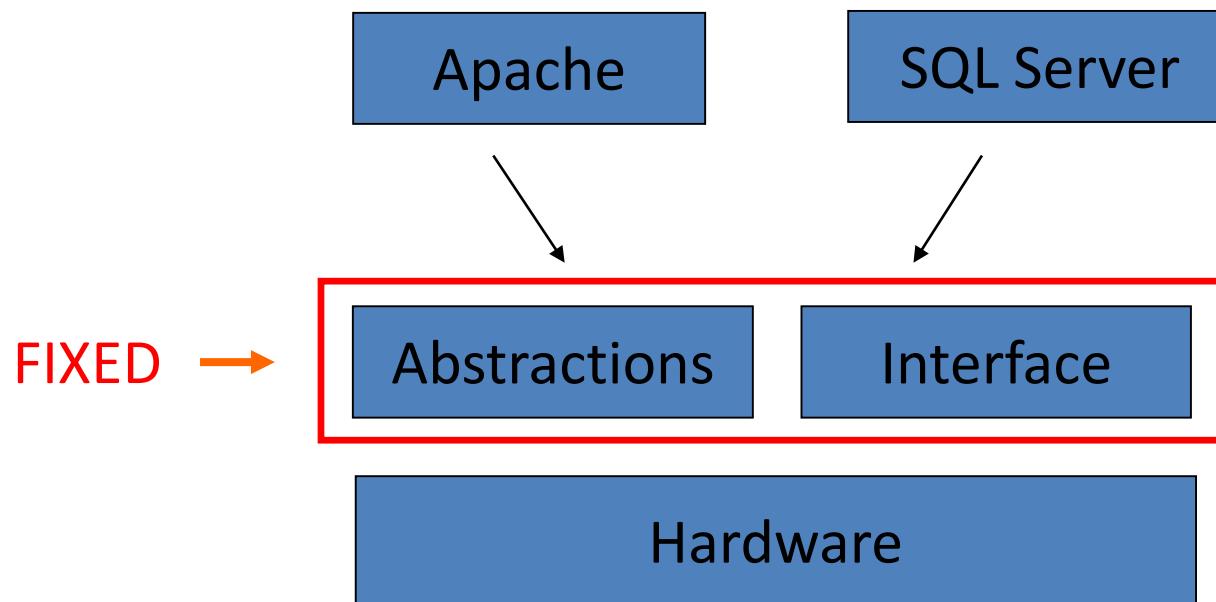
Exokernel Example 1



Exokernel give more direct access to the hardware, thus removing most abstractions

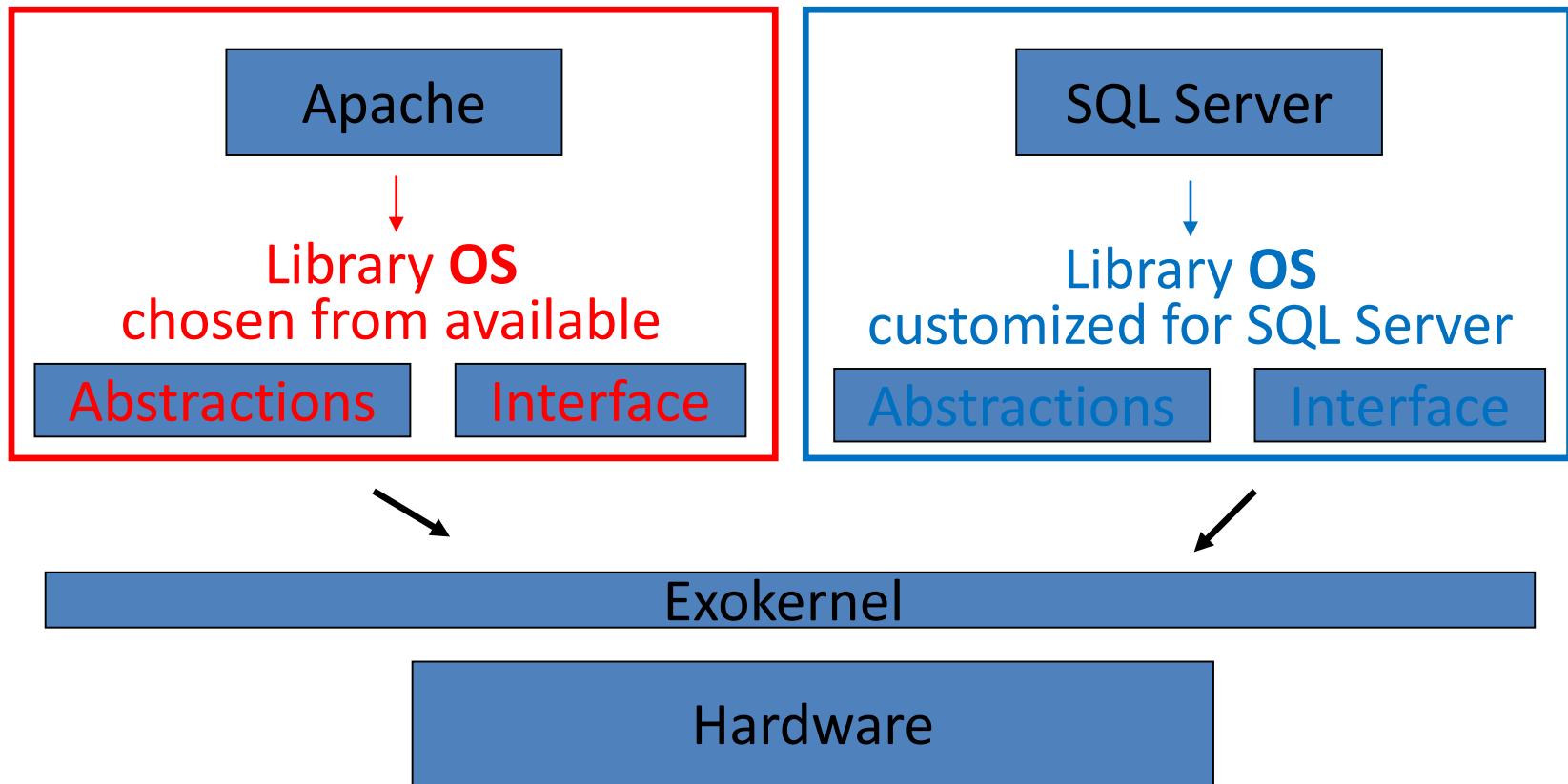
Exokernel Example 2

Traditional OS



Exokernel Example 2

Exokernel + Library OS



Exokernel Design Challenge

Kernel's new role

- Tracking ownership of resources

- Ensuring resource protection

- Revoking resource access

Three techniques

- Secure binding**

- Visible revocation**

- Abort protocol**

Secure binding

It is a protection mechanism that decouples authorization from actual use of a resource

Can improve performance

The protection checks involved in enforcing a secure binding are expressed in terms of simple operations that the kernel can implement quickly

A secure binding performs authorization only at bind time, which allows management to be decoupled from protection

Three techniques

Hardware mechanism, software caching, and downloading application code

Visible resource revocation

A way to reclaim resources and break
their(application & resources) secure binding

An exokernel uses visible revocation for most
resources

Traditionally, OS have performed revocation invisibly,
de-allocating resources without application involvement

Dialogue between an exokernel and a library OS
Library OS should organize resource lists

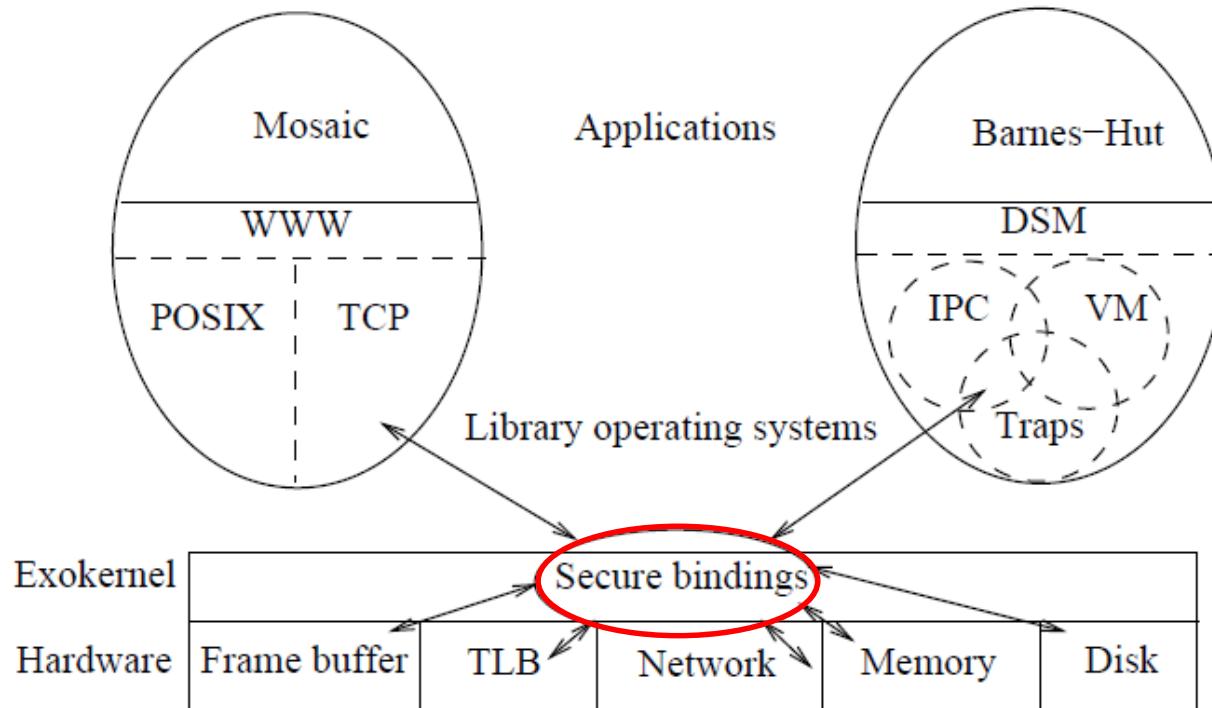
The abort protocol

An exokernel must also be able to take resources from library operating systems that fail to respond satisfactorily to revocation requests

If a library OS fails to respond quickly, the secure bindings need to be broken “**by force**”

An exokernel simply breaks all secure bindings to the resource and informs the library operating system

Exokernel Example 3



An example exokernel-based system consisting of a thin exokernel veneer that exports resources to library operating systems through secure bindings.

Implementation

Prototype (Xok / ExOS)

Xok

Exokernel

Runs on x86 (Aegis runs on DEC)

Safely multiplexes the physical resources

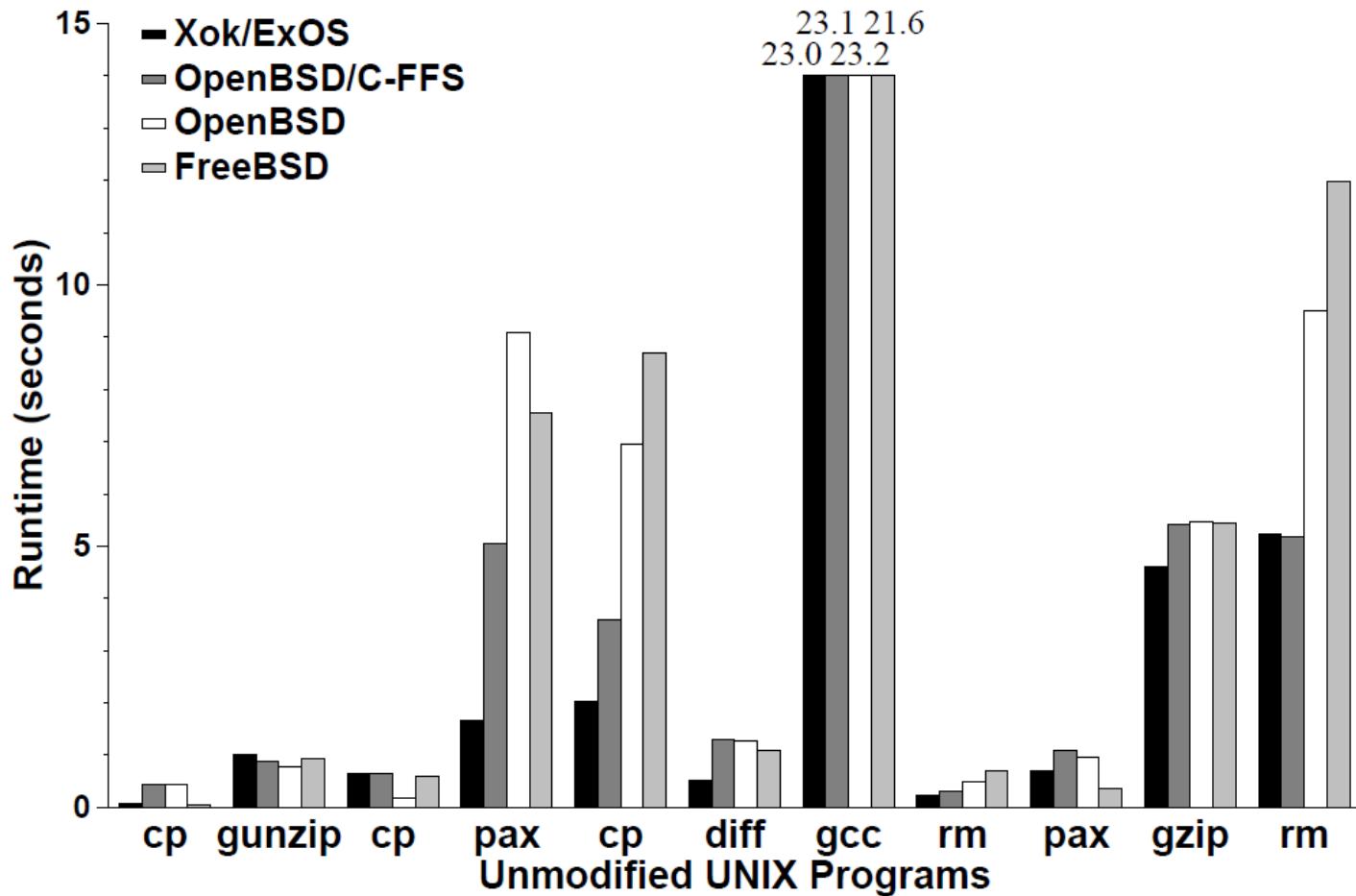
ExOS

Library OS

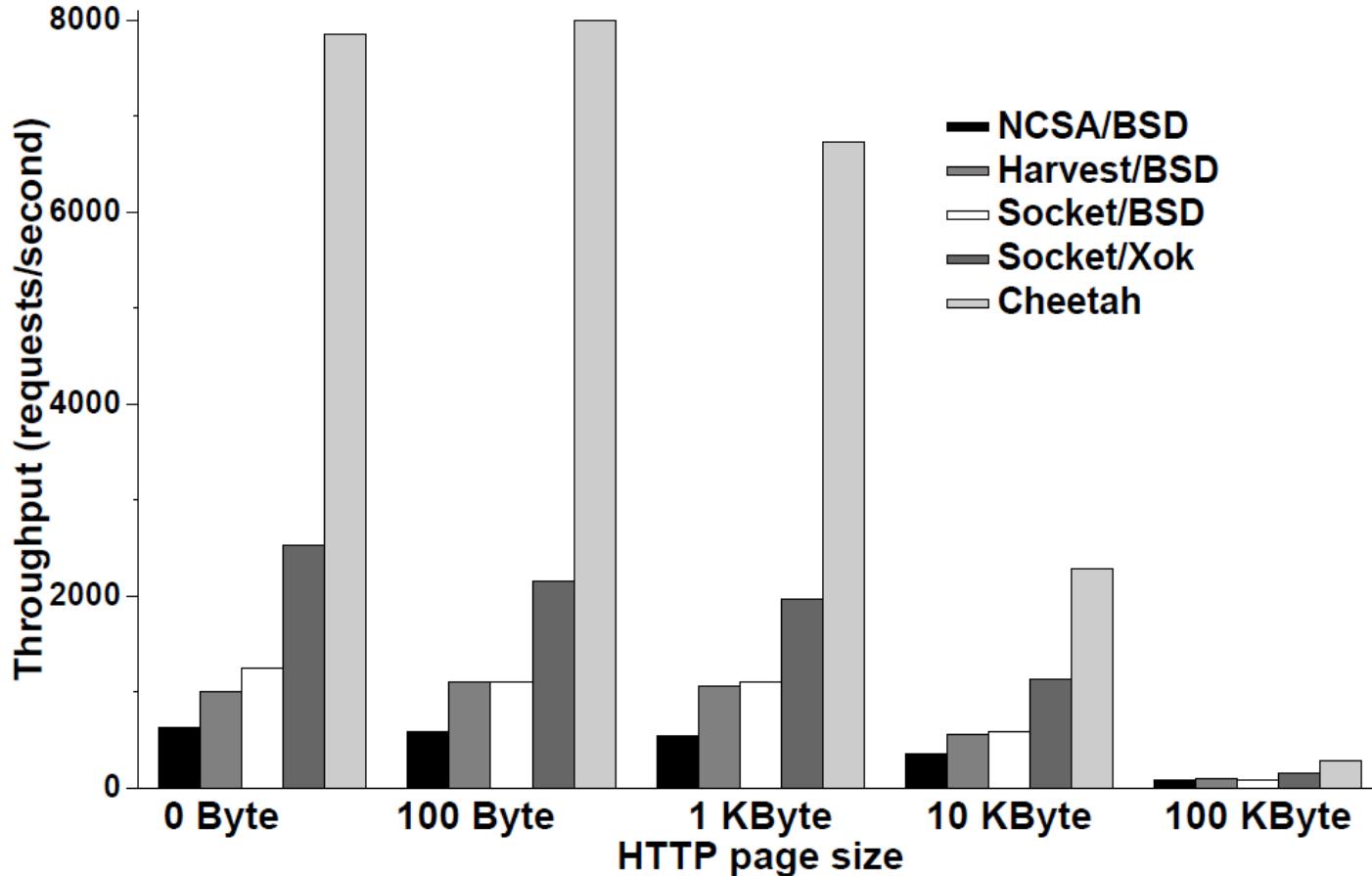
Manages fundamental OS abstractions at application level

completely within the address space of the application
that is using it

Evaluation



Evaluation



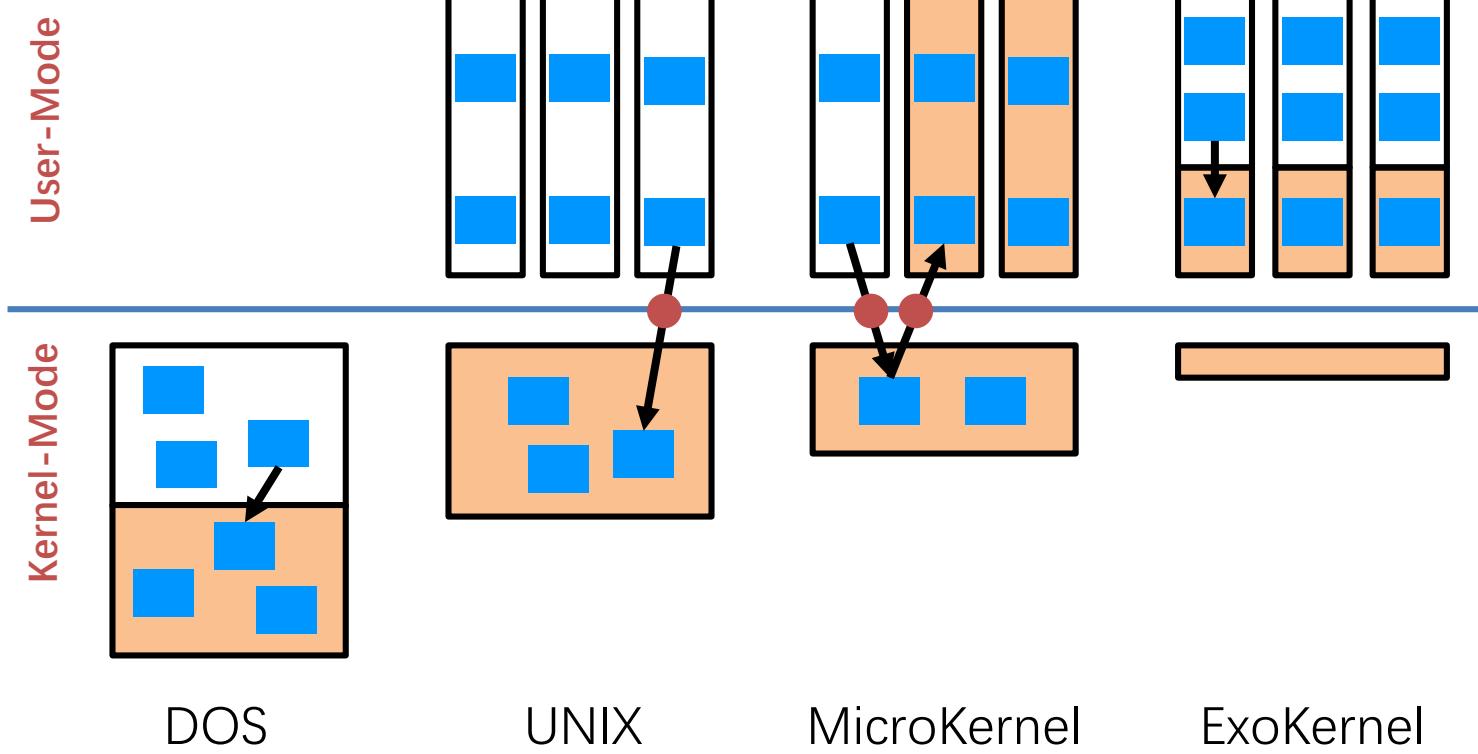
Application-level control can significantly improve the performance of applications

Criticisms of Exokernel

- Customer-Support:
 - *“Extensibility has its problems. For example, it makes the customer-support issues a lot more complicated, because you no longer know which OS each of your customers is running”* (Milojicic, 1999).

Comparison

OS
App
Logic



Homework

Application can gain a better performance of memory and file system in exokernel compared with microkernel or monolithic kernel, why? And what's the price of performance profit?

Exokernel grants the applications more control over hardware resource. How does exokernel protect application against each other? And how to understand this goal of paper: to separate protection from management?

Open question - do you think why has not exokernel been as popular as monolithic kernel(Linux) and Hybrid kernel(Windows NT)?

References

- Exokernel: an operating system architecture for application-specific resource management, SOSP'95
- Application Performance and Flexibility on Exokernel Systems, SOSP'97
- Corey: An Operating System for Many Cores, OSDI'05

Labs

Implement JOS: adopted from MIT 6.828 course

[Lab1](#): Booting a PC

[Lab2](#): Memory Management

[Lab3](#): User Environments

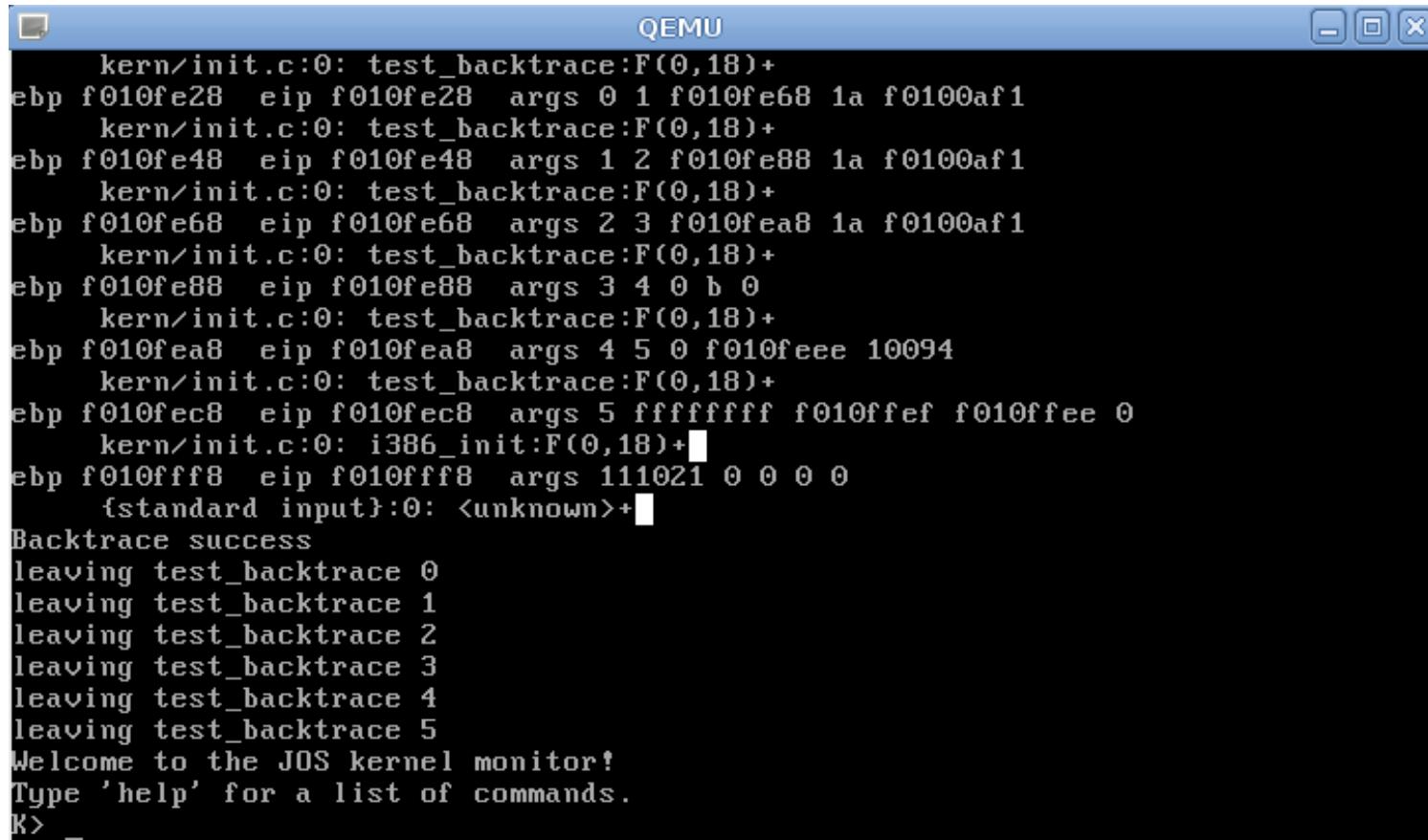
[Lab4](#): Preemptive Multitasking

*Lab 5: XXX

*Lab 6: XXX

Lab 1 Boot a PC

Familiar with PC boot process



The screenshot shows a terminal window titled "QEMU" displaying the boot log of a JOS kernel. The log output is as follows:

```
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fe28 eip f010fe28 args 0 1 f010fe68 1a f0100af1  
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fe48 eip f010fe48 args 1 2 f010fe88 1a f0100af1  
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fe68 eip f010fe68 args 2 3 f010fea8 1a f0100af1  
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fe88 eip f010fe88 args 3 4 0 b 0  
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fea8 eip f010fea8 args 4 5 0 f010feee 10094  
kern/init.c:0: test_backtrace:F(0,18)+  
ebp f010fec8 eip f010fec8 args 5 ffffffff f010ffef f010fee0 0  
kern/init.c:0: i386_init:F(0,18)+  
ebp f010fff8 eip f010fff8 args 111021 0 0 0 0  
{standard input}:0: <unknown>+  
Backtrace success  
leaving test_backtrace 0  
leaving test_backtrace 1  
leaving test_backtrace 2  
leaving test_backtrace 3  
leaving test_backtrace 4  
leaving test_backtrace 5  
Welcome to the JOS kernel monitor!  
Type 'help' for a list of commands.  
K>
```

Lab2 Memory Management

Setup virtual memory map for JOS

The screenshot shows a terminal window titled "QEMU" running SeaBIOS. The output text indicates the boot process:

```
SeaBIOS (version pre-0.6.3-20110315_112143-titi)

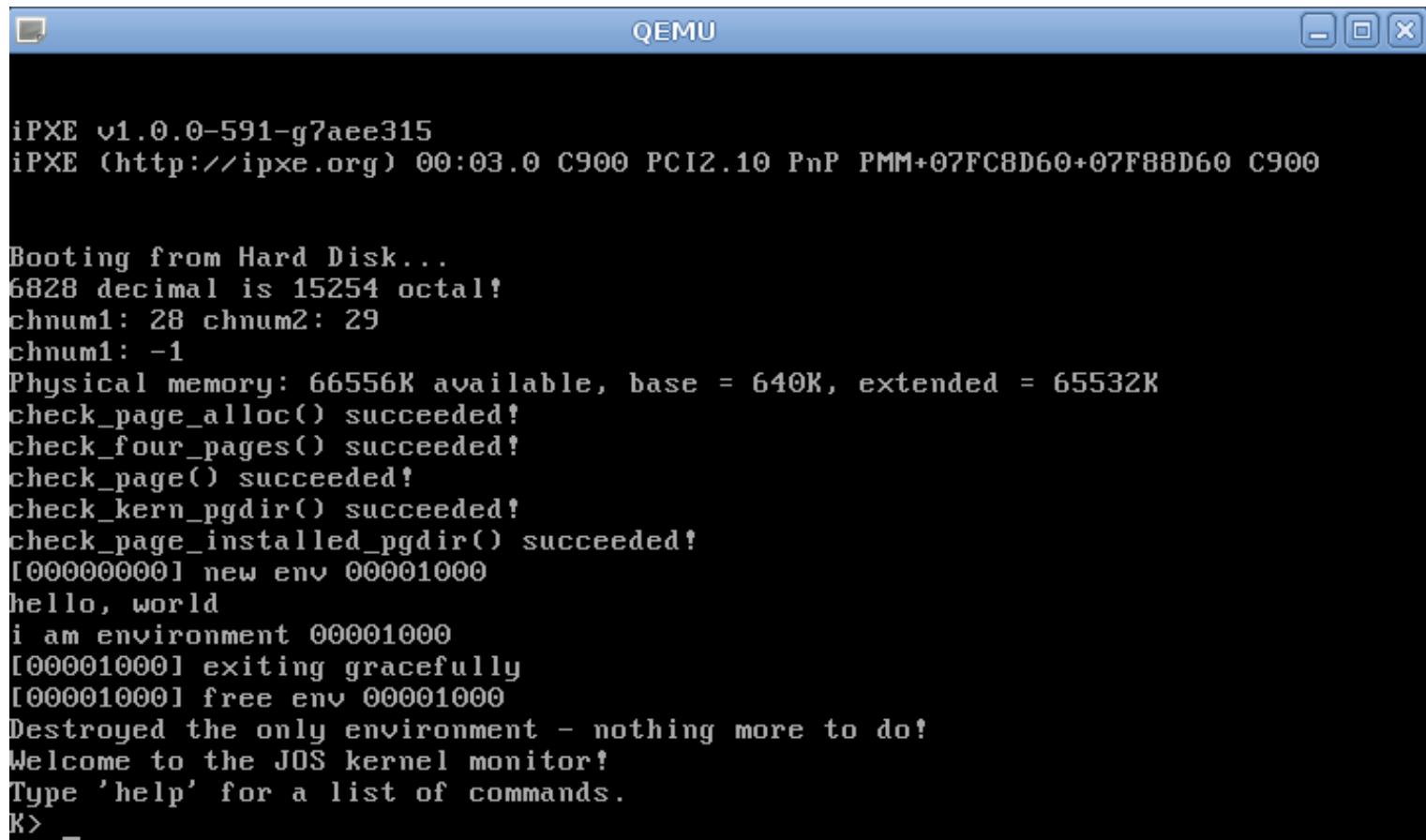
iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC8D60+07F88D60 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
chnum1: 28 chnum2: 29
chnum1: -1
Physical memory: 66556K available, base = 640K, extended = 65532K
    before check page allocate and page free
    ----- page_free_list size: 16482-----
check_page_alloc() succeeded!
check_four_pages() succeeded!
check_page() succeeded!
    after check page allocate and page free
    ----- page_free_list size: 16482-----
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

Lab3: User environment

Environment in JOS = Process in Linux

In this lab, only an environment on a single core



The screenshot shows a terminal window titled "QEMU" running on a Windows operating system. The window contains the following text output from the JOS kernel:

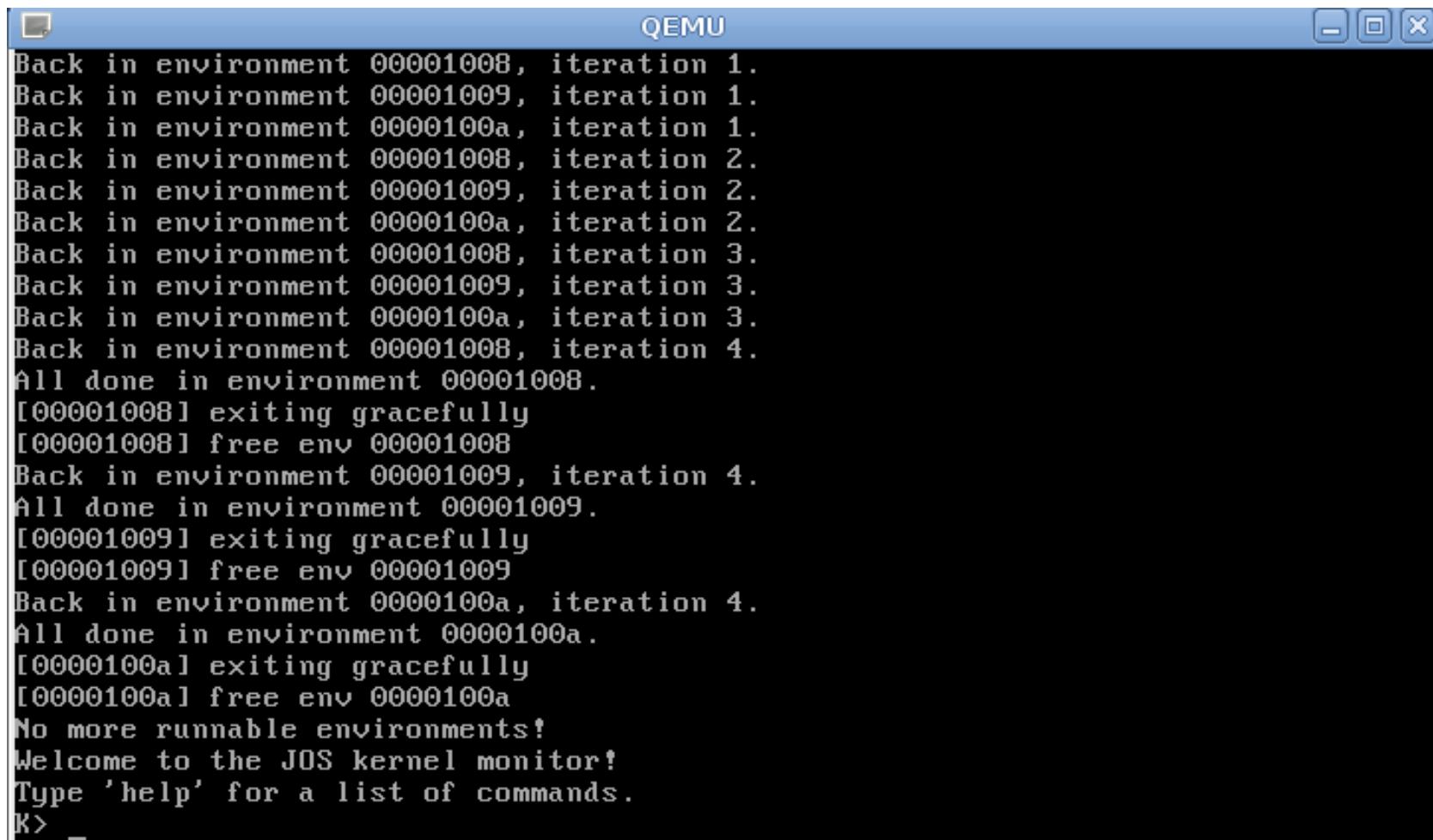
```
iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC8D60+07F88D60 C900

Booting from Hard Disk...
6828 decimal is 15254 octal!
chnum1: 28 chnum2: 29
chnum1: -1
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_four_pages() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
hello, world
i am environment 00001000
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> _
```

Lab4: Preemptive Multitasking

Multicore

Preemptive multitasking



The screenshot shows a terminal window titled "QEMU" running on a Windows operating system. The window contains the following text output:

```
Back in environment 00001008, iteration 1.  
Back in environment 00001009, iteration 1.  
Back in environment 0000100a, iteration 1.  
Back in environment 00001008, iteration 2.  
Back in environment 00001009, iteration 2.  
Back in environment 0000100a, iteration 2.  
Back in environment 00001008, iteration 3.  
Back in environment 00001009, iteration 3.  
Back in environment 0000100a, iteration 3.  
Back in environment 00001008, iteration 4.  
All done in environment 00001008.  
[00001008] exiting gracefully  
[00001008] free env 00001008  
Back in environment 00001009, iteration 4.  
All done in environment 00001009.  
[00001009] exiting gracefully  
[00001009] free env 00001009  
Back in environment 0000100a, iteration 4.  
All done in environment 0000100a.  
[0000100a] exiting gracefully  
[0000100a] free env 0000100a  
No more runnable environments!  
Welcome to the JOS kernel monitor!  
Type 'help' for a list of commands.  
K> -
```

Build your OS

Supporting code will be provided

Still a hard but interesting job

Hints: Start to work early and work hard

Lab Rules

Individual project

Don't copy any code from others

Can read but don't copy other OSes (Linux, Open/FreeBSD, etc.)

Cite any code that inspired your code

Please don't post any “攻略”or code

Grading

General (Tentative)

Lab 1~4: 40%

Final Exam: 40%

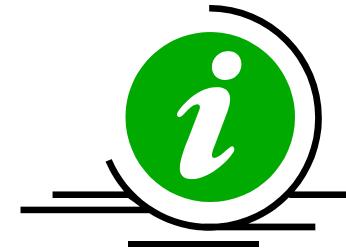
Labs will be reflected in exam

Homework: 10%

Help you to understand OS/Labs

Course performance (5%)

Q&A in classroom, Presentation



Bonus (5%): Lab5 & Lab6

Must finish both in order to get at least 80% scores

PC Programming & Booting

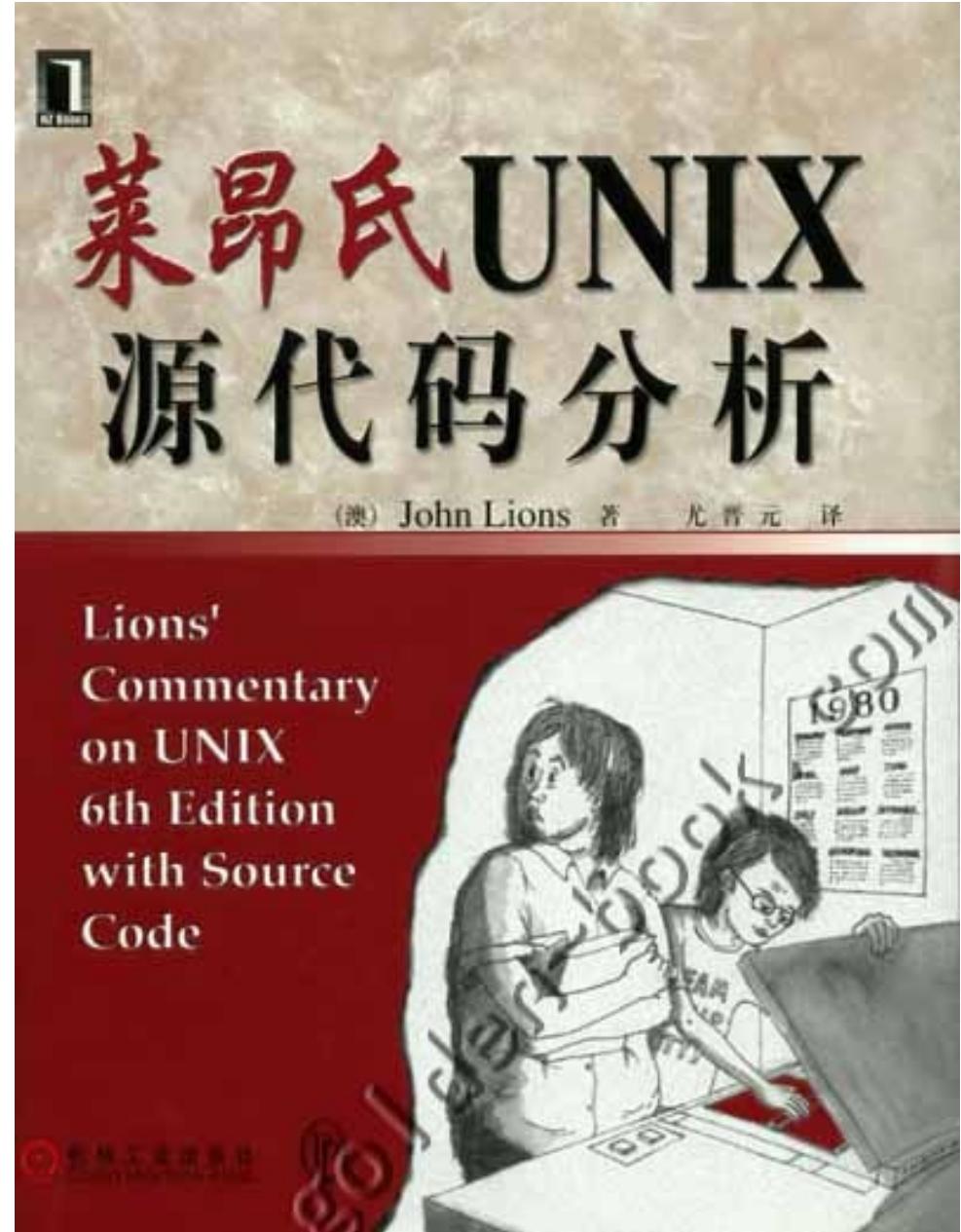
Yubin Xia

Our Tools

- References
 - <http://pdos.csail.mit.edu/6.828/2012/xv6.html>
 - Both the code and book
 - <http://wiki.osdev.org>
 - [IA-32 Intel Architecture Software Developer's Manuals](#)
 - [Volume 3A: System Programming Guide, Part 1](#)
 - [Volume 3B: System Programming Guide, Part 2](#)

Once upon a time ...

- Lion's commentary
- Based on UNIX v6
- Which is not on x86
 - But PDP-11
- Xv6
 - Unix v6
 - For x86!
 - Runnable!



Outline

- PC Architecture
- Memory
- Execution
- PC Emulation

- PC Architecture
- Memory
- Execution
- PC Emulation

A PC



How to make it to do something useful?

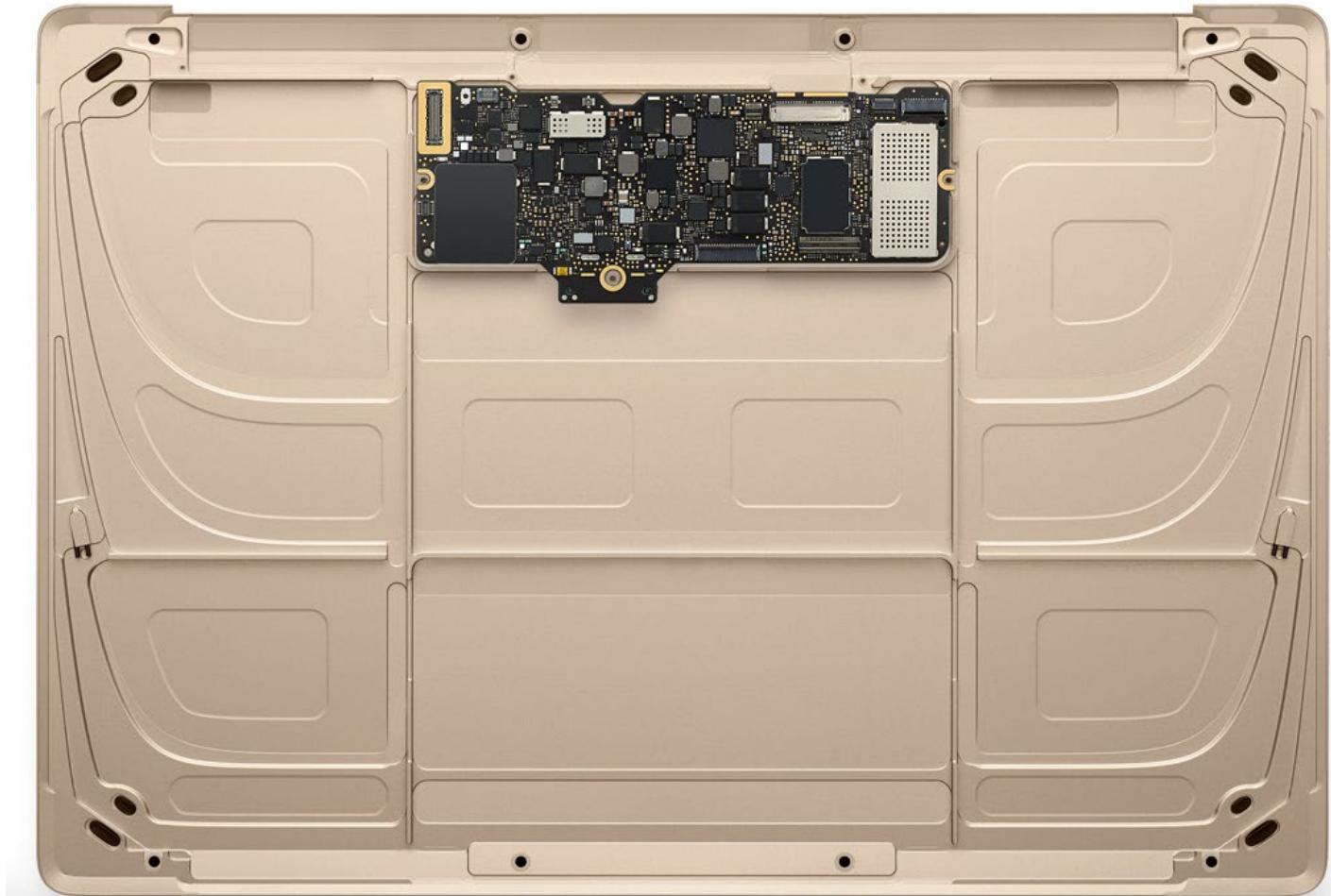
PC board



PC board - Nowadays



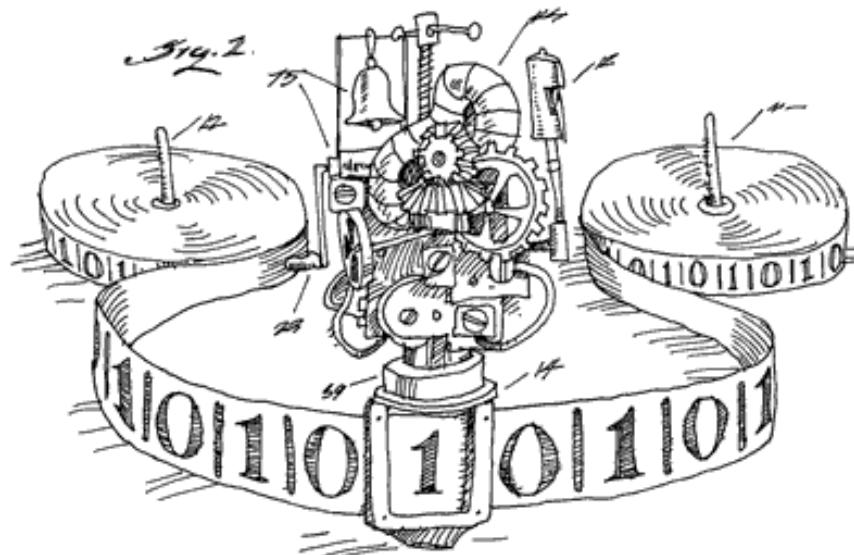
Macbook inside



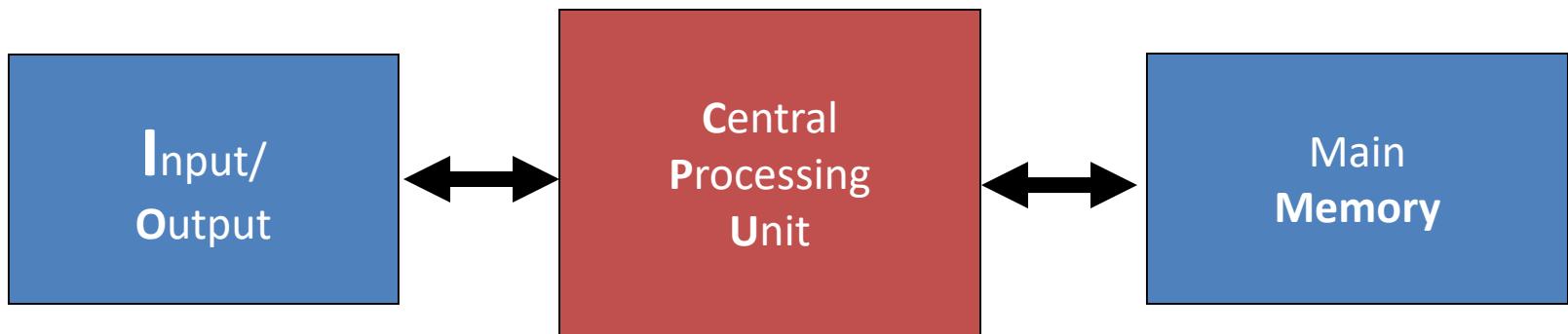
The Turing Machine

Calculate 3+2: 0000001110110000

		- - - - -
1	0	0000001110110000
2	0	0000001110110000
3	0	0000001110110000
4	0	0000001110110000
5	0	0000001110110000
6	0	0000001110110000
7	0	0000001110110000
8	1	0000001110110000
9	1	0000001110110000
10	1	0000001110110000
11	10	0000001111110000
12	10	0000001111110000
13	10	0000001111110000
14	11	0000001111110000
15	0	00000011111100000

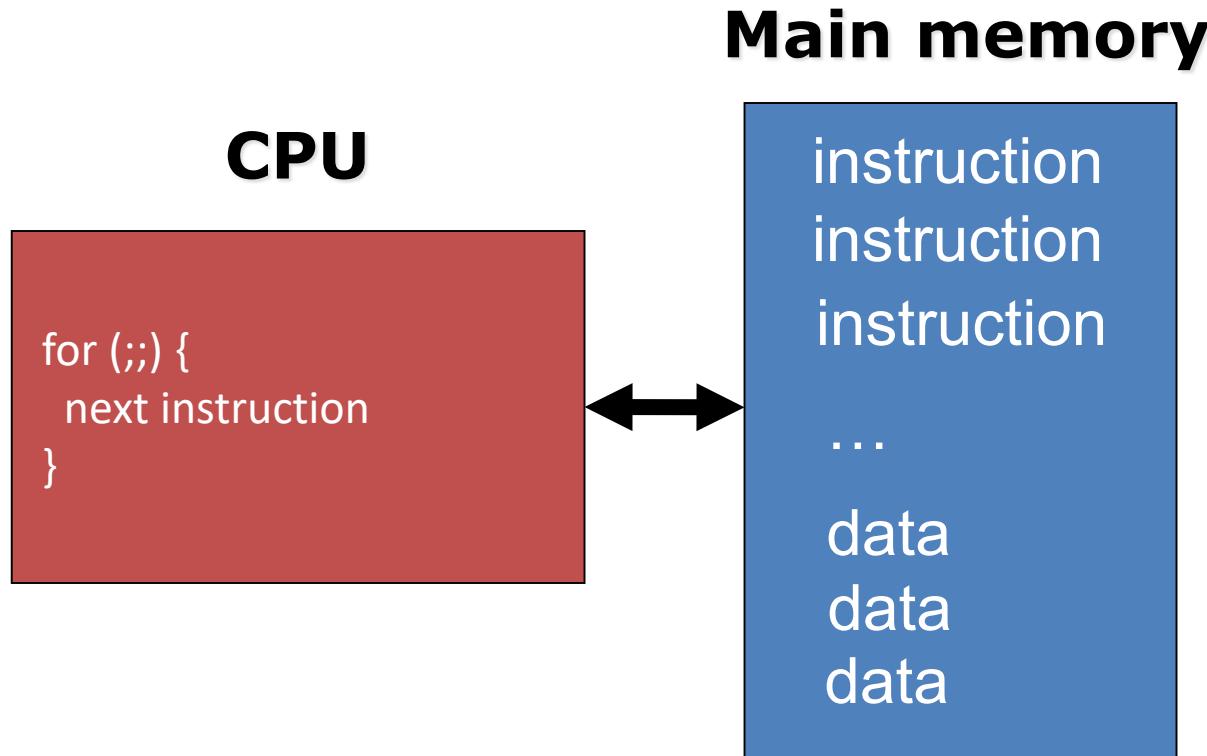


The von Neumann Model



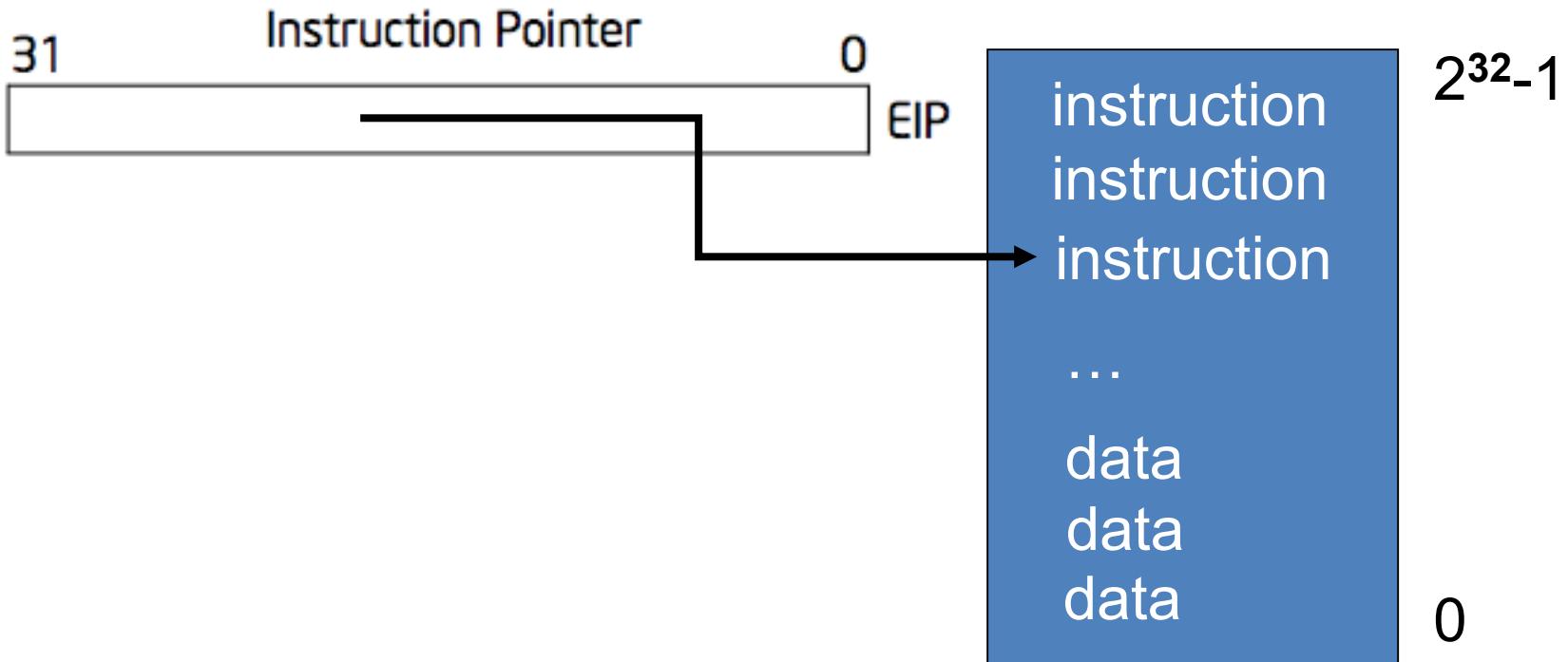
- **I/O:** communicating data to and from devices
- **CPU:** digital logic for performing computation
- **Memory:** **N** words of **B** bits

The Stored Program Computer



- **CPU** interpreter of instructions
- **Memory** holds instructions and data

x86 Implementation (32-bit)



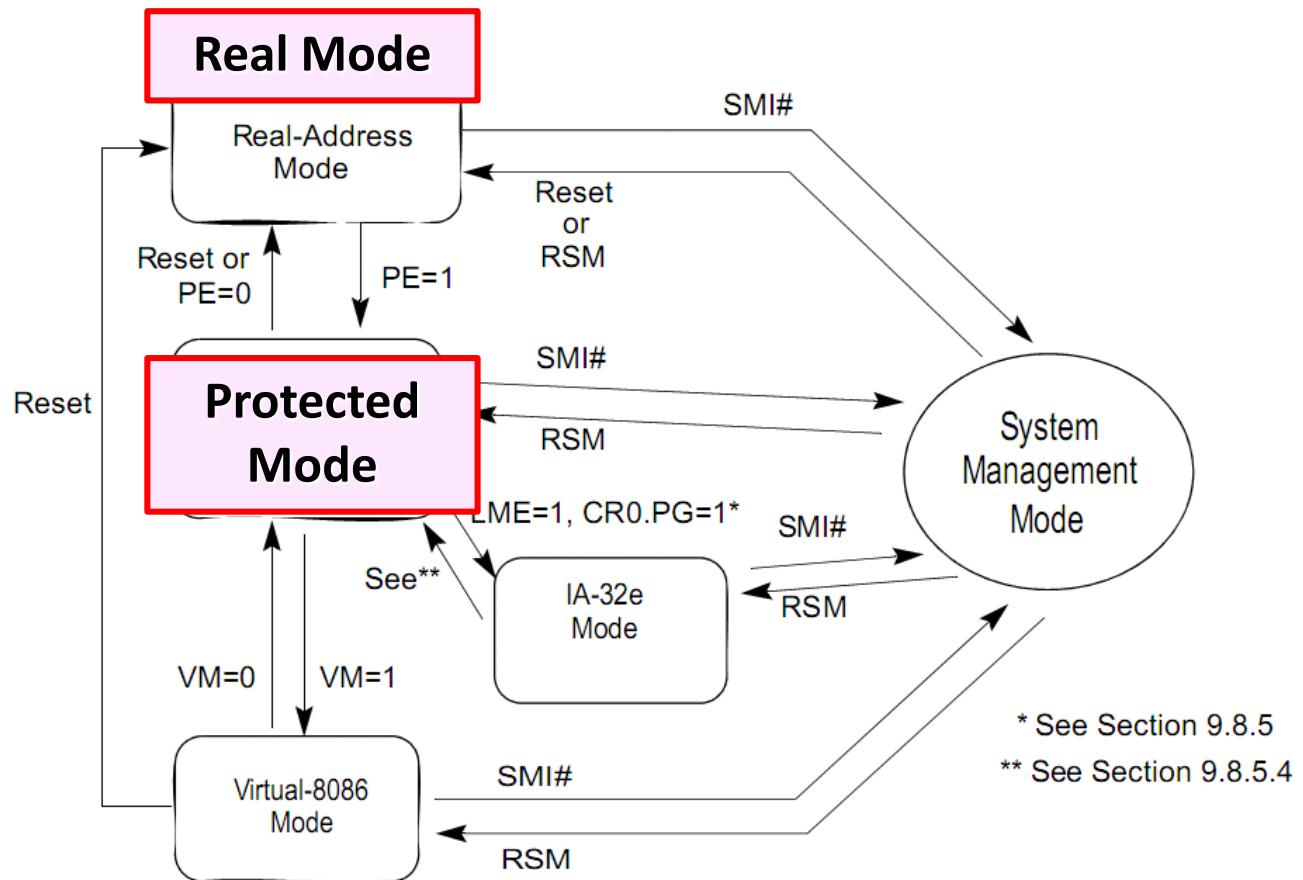
- **EIP** is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and cond. JMP

System Architecture Overview

Volume 3A: System Programming Guide, Part 1

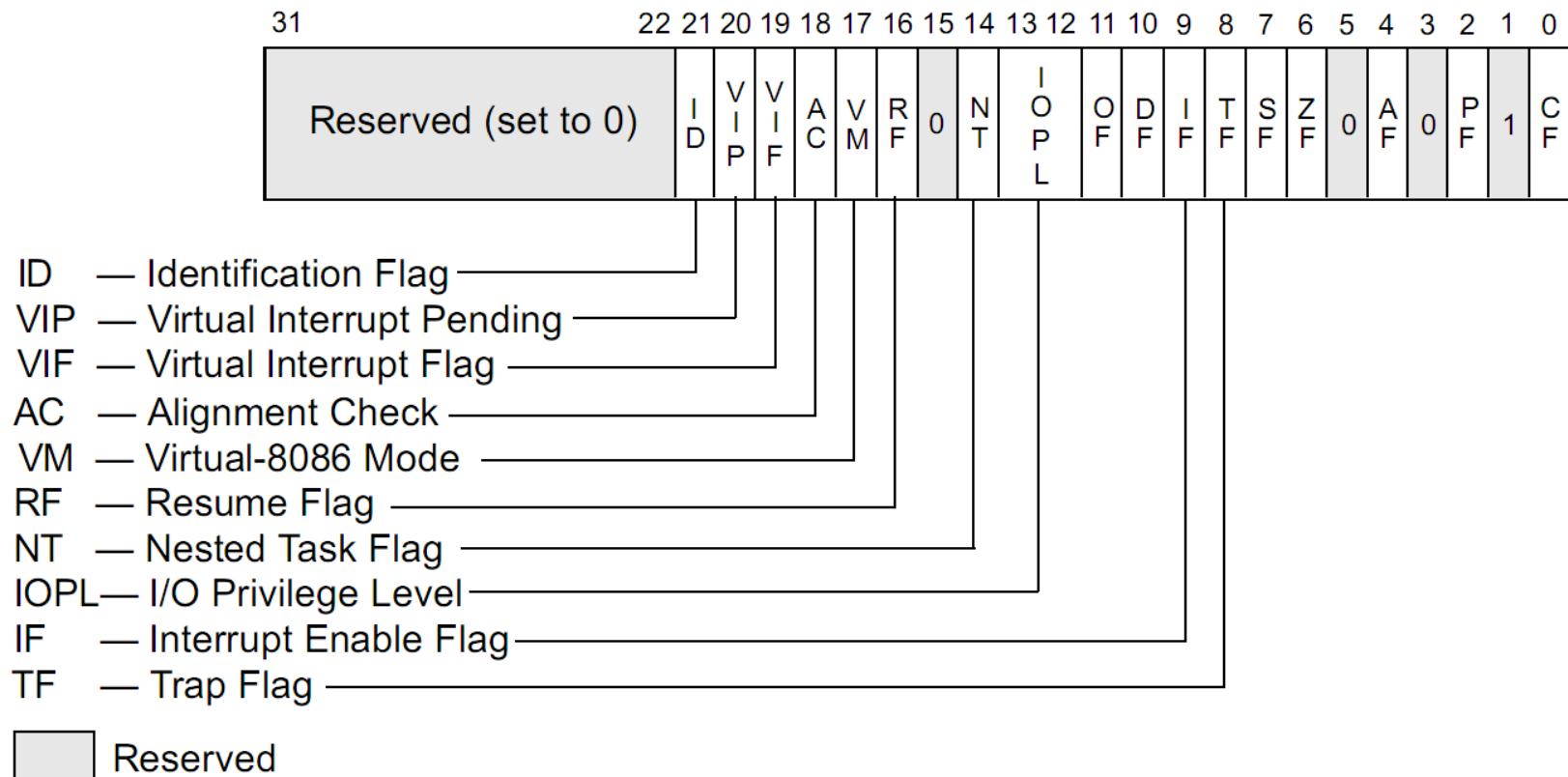
- 2.1~2.5

- Modes



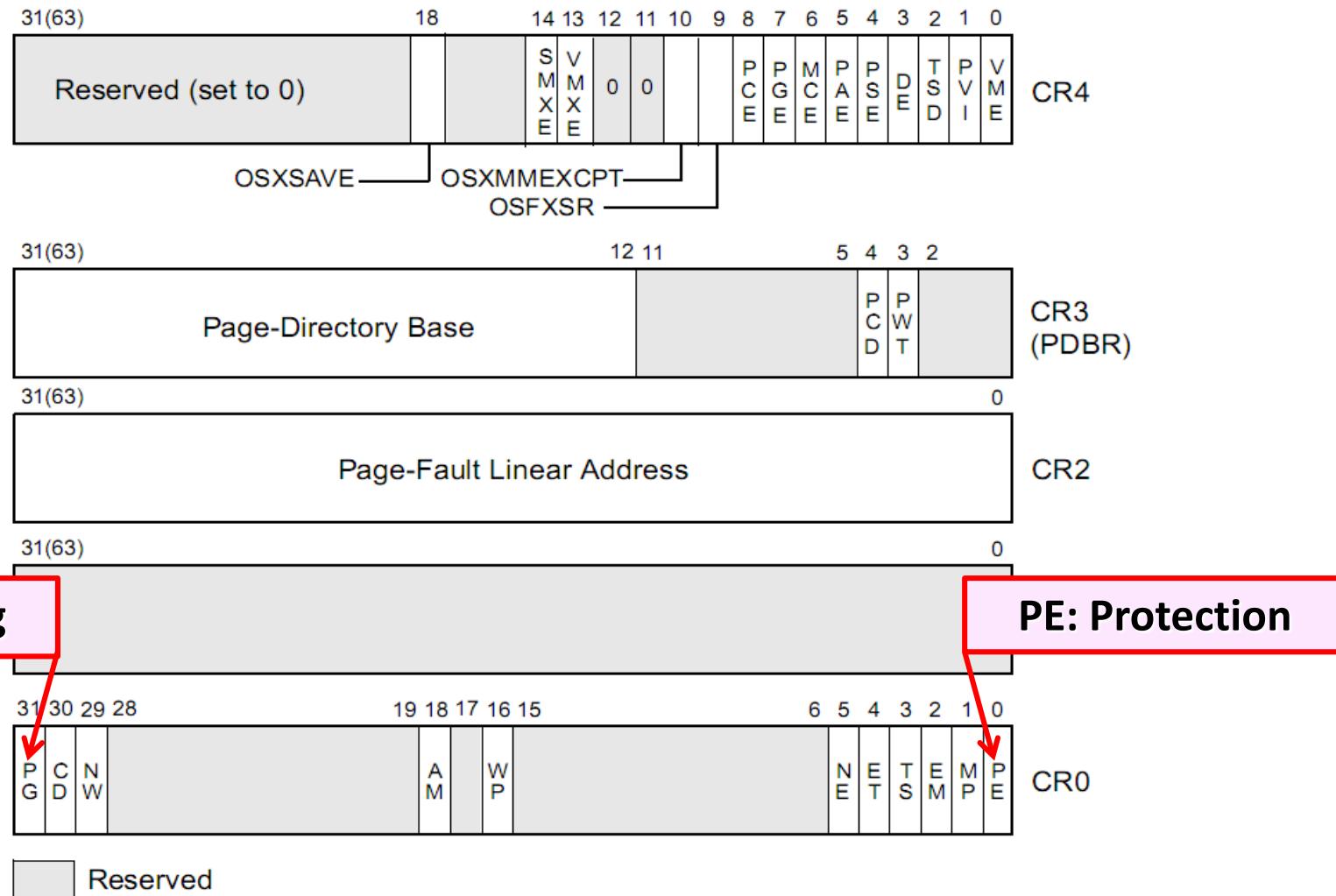
System Architecture Overview

- System Flags in the EFLAGS



System Architecture Overview

- Control Registers



System Architecture Overview

- Memory-Management Registers
 - GDTR (Global Descriptor Table Register)
 - Base Address, Limit ...
 - IDTR (Interrupt Descriptor Table Register)
 - Handler Address, Ring Level ...
 - TR (Task Register)
 - TSS

- PC Architecture
- **Memory**
- Execution
- PC Emulation

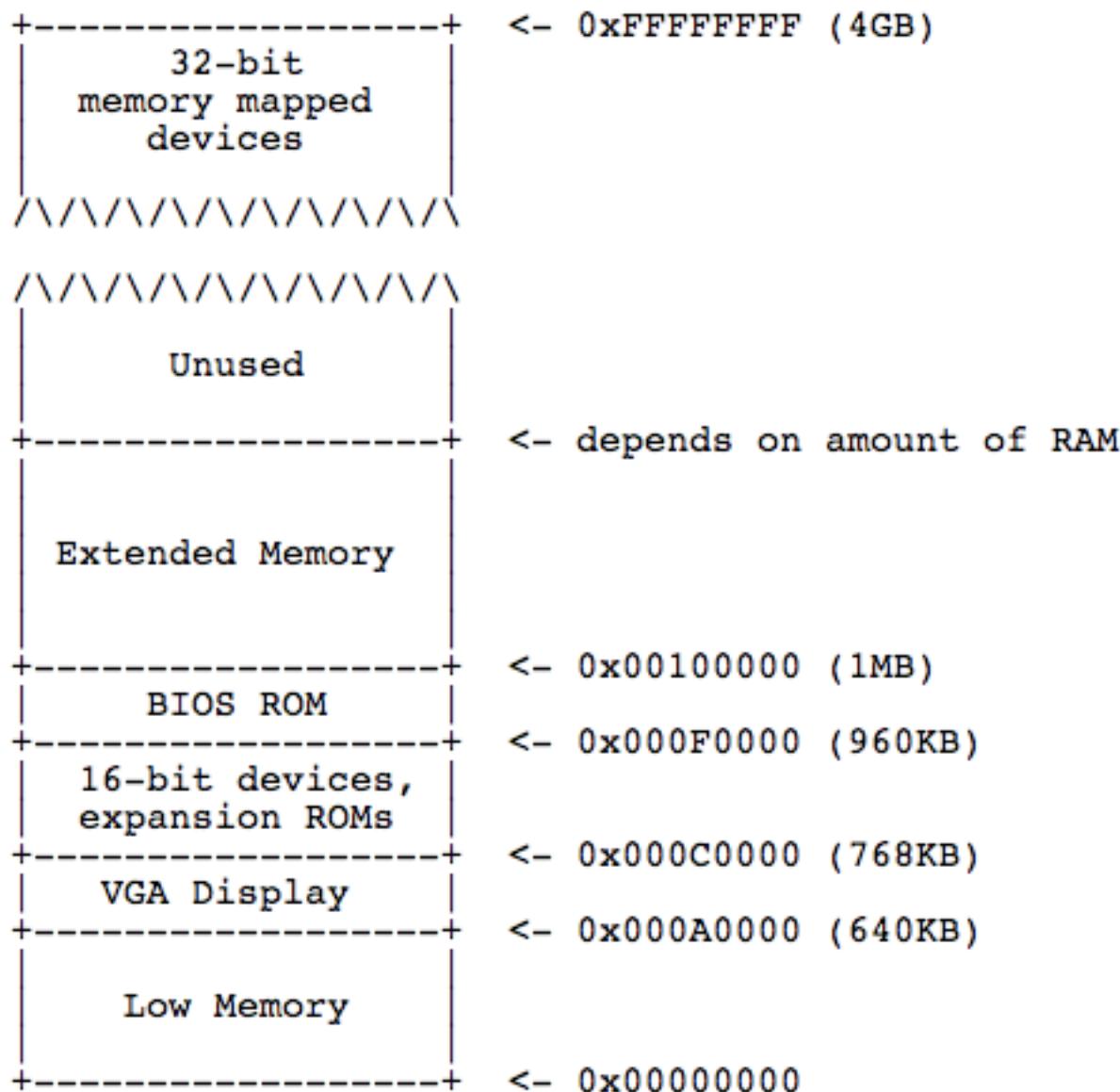
Memory Model

- 8086: **16-bits** microprocessor
 - Real Mode: $physical\ addr = 16 * segment + offset$
 - Space: 64KB
- External address to **20-bits**
 - Space: 1MB
 - The extra 4 bits come from *segment registers*
 - **CS**: code segment, for EIP
 - **SS**: stack segment, for SP and BP
 - **DS**: data segment for load/store via other registers
 - **ES**: another data segment, destination for string ops
 - e.g. $CS=\underline{f}000 \ IP=\underline{fff}0 \Rightarrow ADDR: ffff0$

Memory Model

- 80386: **32-bit** data and bus addresses
 - Protected Mode
 - Now: the transition to **64-bit** addresses
- Backwards compatibility:
 - Boots in 16-bit real mode, and switches to 32-bit protected mode
 - See: “*boot/boot.S*”

Physical Address Space Layout



I/O space

```
#define DATA_PORT      0x378
#define STATUS_PORT     0x379
#define BUSY 0x80
#define CONTROL_PORT   0x37A
#define STROBE 0x01
void
lpt_putc(int c)
{
    /* wait for printer to consume previous byte */
    while((inb(STATUS_PORT) & BUSY) == 0)
        ;

    /* put the byte on the parallel lines */
    outb(DATA_PORT, c);

    /* tell the printer to look at the data */
    outb(CONTROL_PORT, STROBE);
    outb(CONTROL_PORT, 0);
}
```

```
static __inline uint8_t inb(int port)
{
    uint8_t data;
    __asm __volatile("inb %w1,%0"
                    : "=a" (data) : "d" (port));
    return data;
}
```

```
static __inline void
    outb(int port, uint8_t data)
{
    __asm __volatile("outb %0,%w1"
                    : : "a" (data), "d" (port));
}
```

Memory-mapped I/O

- Use normal addresses
 - No need for special instructions
 - No 1024 limit
 - System controller routes to device
- Works like “magic” Memory
 - Addressed and accessed like memory
 - But does not behave like memory
 - Reads and writes have “**side effects**”
 - Read result can change due to external events
 - Recall: the “*volatile*” keyword

- PC Architecture
- Memory
- **Execution**
- PC Emulation

XV6 BOOTING CODE

```
# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.code16                                # Assemble for 16-bit mode
.globl start
start:
    cli                               # BIOS enabled interrupts; disable

    # Zero data segment registers DS, ES, and SS.
    xorw    %ax,%ax                  # Set %ax to zero
    movw    %ax,%ds                  # -> Data Segment
    movw    %ax,%es                  # -> Extra Segment
    movw    %ax,%ss                  # -> Stack Segment

    # Physical address line A20 is tied to zero so that the first PCs
    # with 2 MB would run software that assumed 1 MB. Undo that.
seta20.1:
    inb    $0x64,%al                # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.1

    movb    $0xd1,%al                # 0xd1 -> port 0x64
    outb   %al,$0x64

seta20.2:
    inb    $0x64,%al                # Wait for not busy
    testb   $0x2,%al
    jnz     seta20.2

    movb    $0xdf,%al                # 0xdf -> port 0x60
    outb   %al,$0x60
```

A20 Line

(https://wiki.osdev.org/A20_Line)

The A20 Address Line is the physical representation of the 21st bit (number 20, counting from 0) of any memory access. When the IBM-AT (Intel 286) was introduced, it was able to access up to sixteen megabytes of memory (instead of the 1 MByte of the 8086). But to remain compatible with the 8086, a quirk in the 8086 architecture (memory wraparound) had to be duplicated in the AT. To achieve this, the A20 line on the address bus was disabled by default.

The wraparound was caused by the fact the 8086 could only access 1 megabyte of memory, but because of the segmented memory model it could effectively address up to 1 megabyte and 64 kilobytes (minus 16 bytes). Because there are 20 address lines on the 8086 (A0 through A19), any address above the 1 megabyte mark wraps around to zero. For some reason a few short-sighted programmers decided to write programs that actually used this wraparound (rather than directly addressing the memory at its normal location at the bottom of memory). Therefore in order to support these 8086-era programs on the new processors, this wraparound had to be emulated on the IBM AT and its compatibles; this was originally achieved by way of a latch that by default set the A20 line to zero. Later the 486 added the logic into the processor and introduced the A20M pin to control it.

For an operating system developer this means the A20 line has to be enabled so that all memory can be accessed. This started off as a simple hack but as simpler methods were added to do it, it became harder to program code that would definitely enable it and even harder to program code that would definitely disable it.

The traditional method for A20 line enabling is to directly probe the keyboard controller. The reason for this is that Intel's 8042 keyboard controller had a spare pin which they decided to route the A20 line through. This seems foolish now given their unrelated nature, but at the time computers weren't quite so standardized. Keyboard controllers are usually derivatives of the 8042 chip. By programming that chip accurately, you can either enable or disable bit #20 on the address bus.

When your PC boots, the A20 gate is always disabled, but some BIOSes do enable it for you, as do some high-memory managers (HIMEM.SYS) or bootloaders (GRUB).

```
# Switch from real to protected mode. Use a bootstrap GDT that makes
# virtual addresses map directly to physical addresses so that the
# effective memory map doesn't change during the transition.
lgdt    gdtdesc
movl    %cr0, %eax
orl    $CR0_PE, %eax
movl    %eax, %cr0

//PAGEBREAK!
# Complete transition to 32-bit protected mode by using long jmp
# to reload %cs and %eip. The segment descriptors are set up with no
# translation, so that the mapping is still the identity mapping.
ljmp   $(SEG_KCODE<<3), $start32

.code32 # Tell assembler to generate 32-bit code now.
start32:
# Set up the protected-mode data segment registers
movw   $(SEG_KDATA<<3), %ax      # Our data segment selector
movw   %ax, %ds                   # -> DS: Data Segment
movw   %ax, %es                   # -> ES: Extra Segment
movw   %ax, %ss                   # -> SS: Stack Segment
movw   $0, %ax                   # Zero segments not ready for use
movw   %ax, %fs                   # -> FS
movw   %ax, %gs                   # -> GS

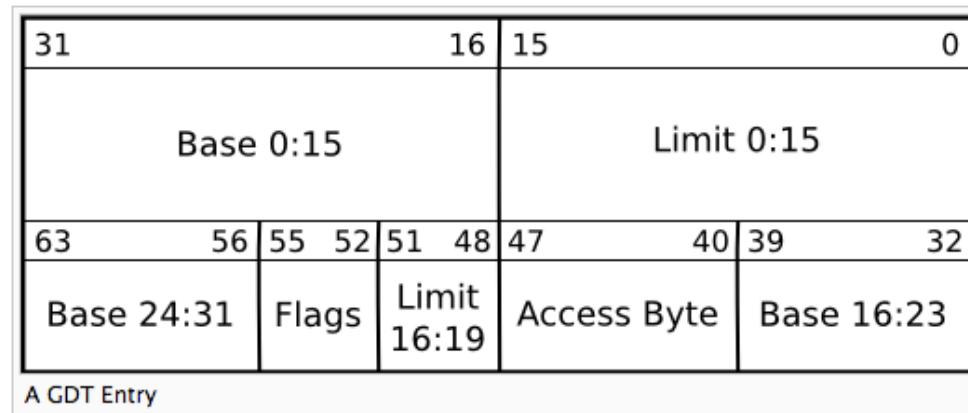
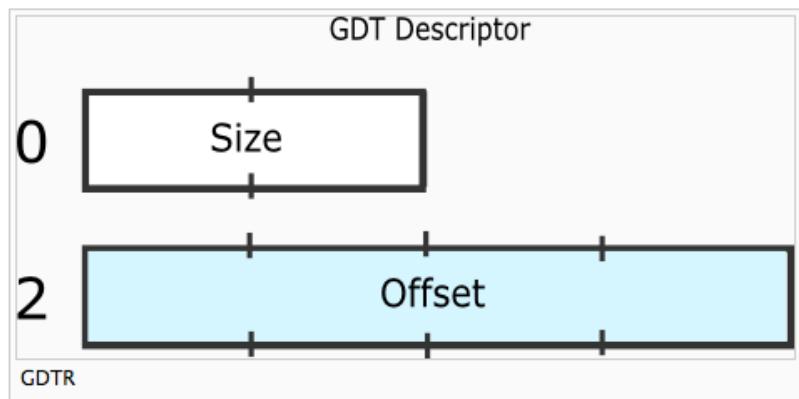
# Set up the stack pointer and call into C.
movl   $start, %esp
call   bootmain
```

```

# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
SEG_NULLASM                                # null seg
SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)      # code seg
SEG_ASM(STA_W, 0x0, 0xffffffff)              # data seg

gdtdesc:
.word  (gdtdesc - gdt - 1)                  # sizeof(gdt) - 1
.long  gdt                                  # address gdt

```



http://wiki.osdev.org/Global_Descriptor_Table

https://wiki.osdev.org/GDT_Tutorial

```
void
bootmain(void)
{
    struct elfhdr *elf;
    struct proghdr *ph, *eph;
    void (*entry)(void);
    uchar* pa;

    elf = (struct elfhdr*)0x10000; // scratch space

    // Read 1st page off disk
    readseg((uchar*)elf, 4096, 0);

    // Is this an ELF executable?
    if(elf->magic != ELF_MAGIC)
        return; // let bootasm.S handle error

    // Load each program segment (ignores ph flags).
    ph = (struct proghdr*)((uchar*)elf + elf->phoff);
    eph = ph + elf->phnum;
    for(; ph < eph; ph++){
        pa = (uchar*)ph->paddr;
        readseg(pa, ph->filesz, ph->off);
        if(ph->memsz > ph->filesz)
            stosb(pa + ph->filesz, 0, ph->memsz - ph->filesz);
    }

    // Call the entry point from the ELF header.
    // Does not return!
    entry = (void(*)(void))(elf->entry);
    entry();
}
```

```
void
readsect(void *dst, uint offset)
{
    // Issue command.
    waitdisk();
    outb(0x1F2, 1);    // count = 1
    outb(0x1F3, offset);
    outb(0x1F4, offset >> 8);
    outb(0x1F5, offset >> 16);
    outb(0x1F6, (offset >> 24) | 0xE0);
    outb(0x1F7, 0x20); // cmd 0x20 - read sectors

    // Read data.
    waitdisk();
    insl(0x1F0, dst, SECTSIZE/4);
}

// Read 'count' bytes at 'offset' from kernel into physical address 'pa'.
// Might copy more than asked.
void
readseg(uchar* pa, uint count, uint offset)
{
    uchar* epa;

    epa = pa + count;

    // Round down to sector boundary.
    pa -= offset % SECTSIZE;

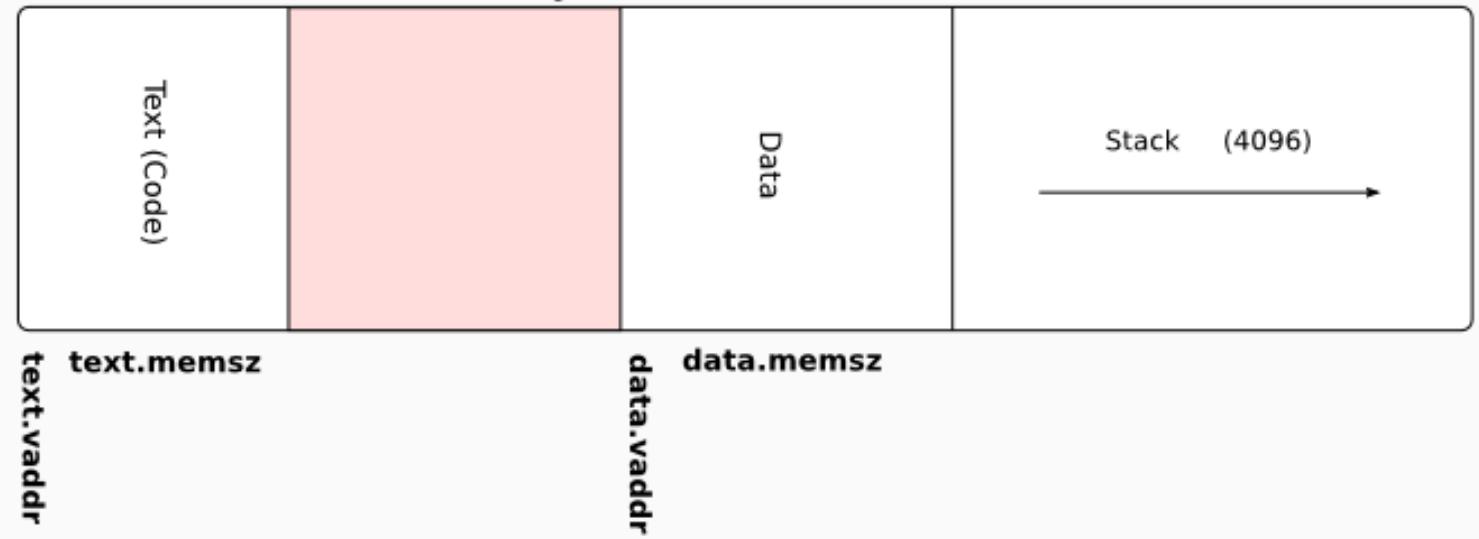
    // Translate from bytes to sectors; kernel starts at sector 1.
    offset = (offset / SECTSIZE) + 1;

    // If this is too slow, we could read lots of sectors at a time.
    // We'd write more to memory than asked, but it doesn't matter --
    // we load in increasing order.
    for(; pa < epa; pa += SECTSIZE, offset++)
        readsect(pa, offset);
}
```

Elf File Image



Executable in Memory



Executable image and elf binary can being mapped onto each other

```
# By convention, the _start symbol specifies the ELF entry point.  
# Since we haven't set up virtual memory yet, our entry point is  
# the physical address of 'entry'.  
.globl _start  
_start = V2P_W0(entry)  
  
# Entering xv6 on boot processor, with paging off.  
.globl entry  
entry:  

```

```
static void startothers(void);
static void mpmain(void) __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file

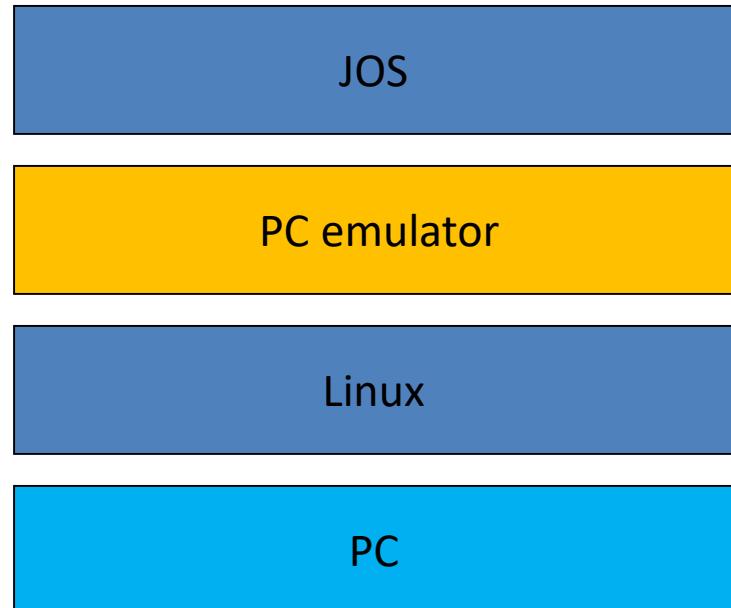
// Bootstrap processor starts running C code here.
// Allocate a real stack and switch to it, first
// doing some setup required for memory allocator to work.
int
main(void)
{
    kinit1(end, P2V(4*1024*1024)); // phys page allocator
    kvmalloc(); // kernel page table
    mpinit(); // collect info about this machine
    lapicinit(mpbcpu());
    seginit(); // set up segments
    cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
    picinit(); // interrupt controller
    ioapicinit(); // another interrupt controller
    consoleinit(); // I/O devices & their interrupts
    uartinit(); // serial port
    pinit(); // process table
    tvinit(); // trap vectors
    binit(); // buffer cache
    fileinit(); // file table
    iinit(); // inode cache
    ideinit(); // disk
    if(!ismp)
        timerinit(); // uniprocessor timer
    startothers(); // start other processors
    kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
    userinit(); // first user process
    // Finish setting up this processor in mpmain.
    mpmain();
}
```

- PC Architecture
- Memory
- Execution
- PC Emulation

Development using PC emulator

- *Bochs* PC emulator
 - **Does** what a real PC **does**
 - Only implemented in software!

Runs like a normal program
on “host” operating system



<http://bochs.sourceforge.net/>

Emulation of CPU

```
for (;;) {
    read_instruction();
    switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
            int src = decode_src_reg();
            int dst = decode_dst_reg();
            regs[dst] = regs[dst] + regs[src];
            break;
        case OPCODE_SUB:
            int src = decode_src_reg();
            int dst = decode_dst_reg();
            regs[dst] = regs[dst] - regs[src];
            break;
        ...
    }
    eip += instruction_length;
}
```

Emulation of Memory

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...
char mem[256*1024*1024];
```

Emulation of x86 Memory

```
uint8_t read_byte(uint32_t phys_addr) {
    if (phys_addr < LOW_MEMORY)
        return low_mem[phys_addr];
    else if (phys_addr >= 960*KB && phys_addr < 1*MB)
        return rom_bios[phys_addr - 960*KB];
    else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
        return ext_mem[phys_addr-1*MB];
    } else ...
}
```

Low Memory

Extended Memory

```
void write_byte(uint32_t phys_addr, uint8_t val) {
    if (phys_addr < LOW_MEMORY)
        low_mem[phys_addr] = val;
    else if (phys_addr >= 960*KB && phys_addr < 1*MB)
        ; /* ignore attempted write to ROM! */
    else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
        ext_mem[phys_addr-1*MB] = val;
    } else ...
}
```

Emulation of Devices

- Hard disk: using a file of the host
- VGA display: draw in a host window
- Keyboard: host's keyboard API
- Clock chip: host's clock
- Etc.

Why Emulator

- OS Test and Debug
- Increase Utilization
- Just as Why IBM's Virtualization
 - IBM's M44/44X, in 1960s

Thanks

Next time:

Intro to PC booting & OS structure

Homework

- What is the difference between user-level ISA and system-level ISA?
- What is memory addressing mode? How many modes are there (please describe them)? Why not just use one?
- Use your own word (and figures, if you want) to describe the process from power-on to BIOS end (just before kernel starts)
 - What is the usage of "ljmp \$(SEG_KCODE<<3), \$start32"?
 - What do you learn from the A20 problem?

XV6 & VM

Yubin Xia

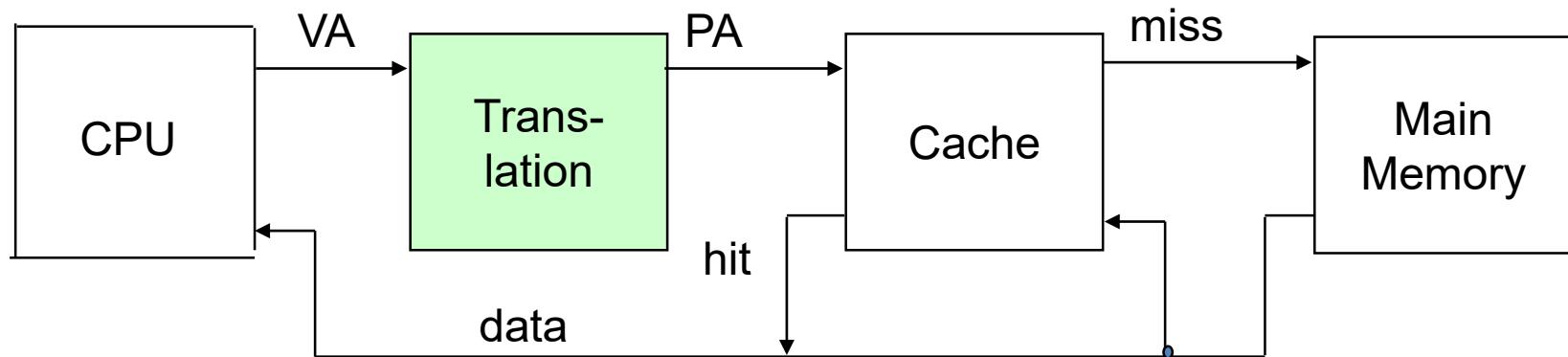
Some slides are based on Frans Kaashoek's slides
kaashoek@mit.edu

Outline

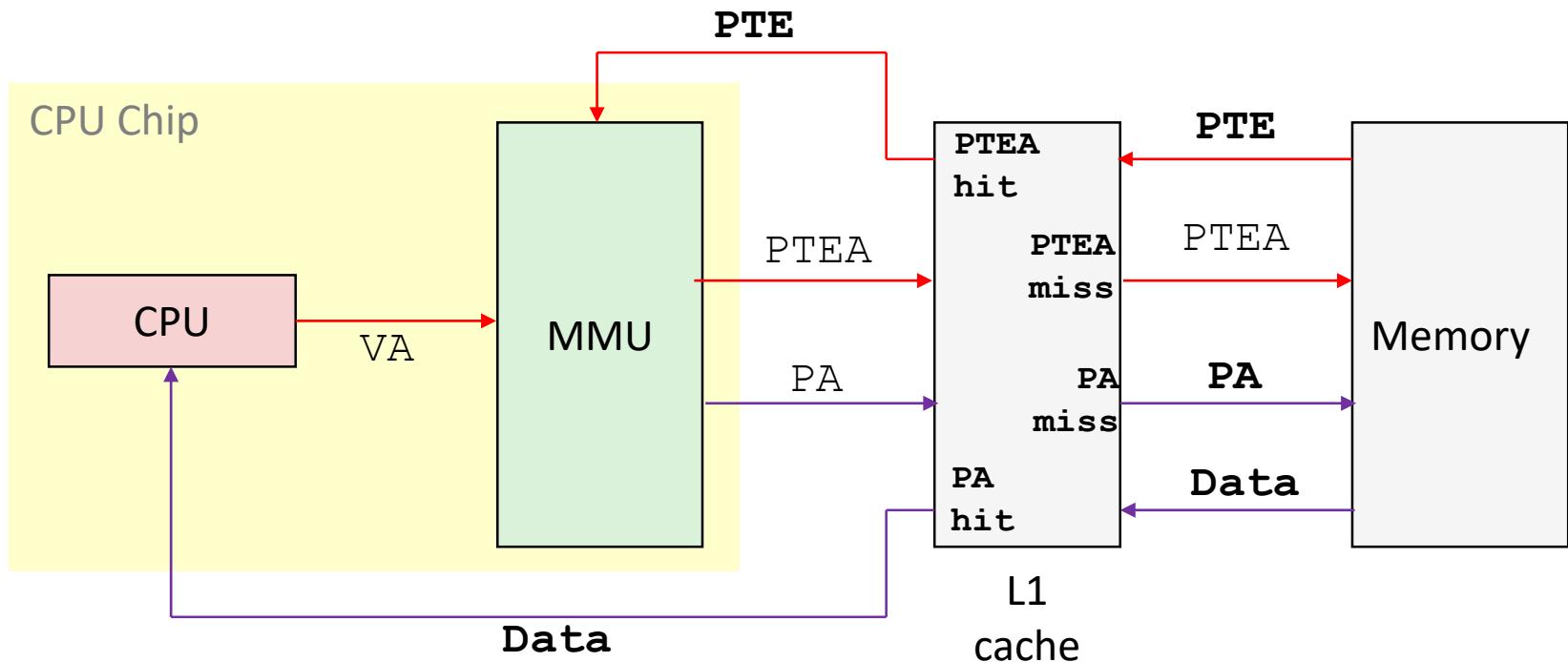
- Segment, Paging
- LA, VA, PA
- VM in xv6
 - Initial page table
 - Kernel page table
 - User page table
- Page Fault

VIRTUAL MEMORY

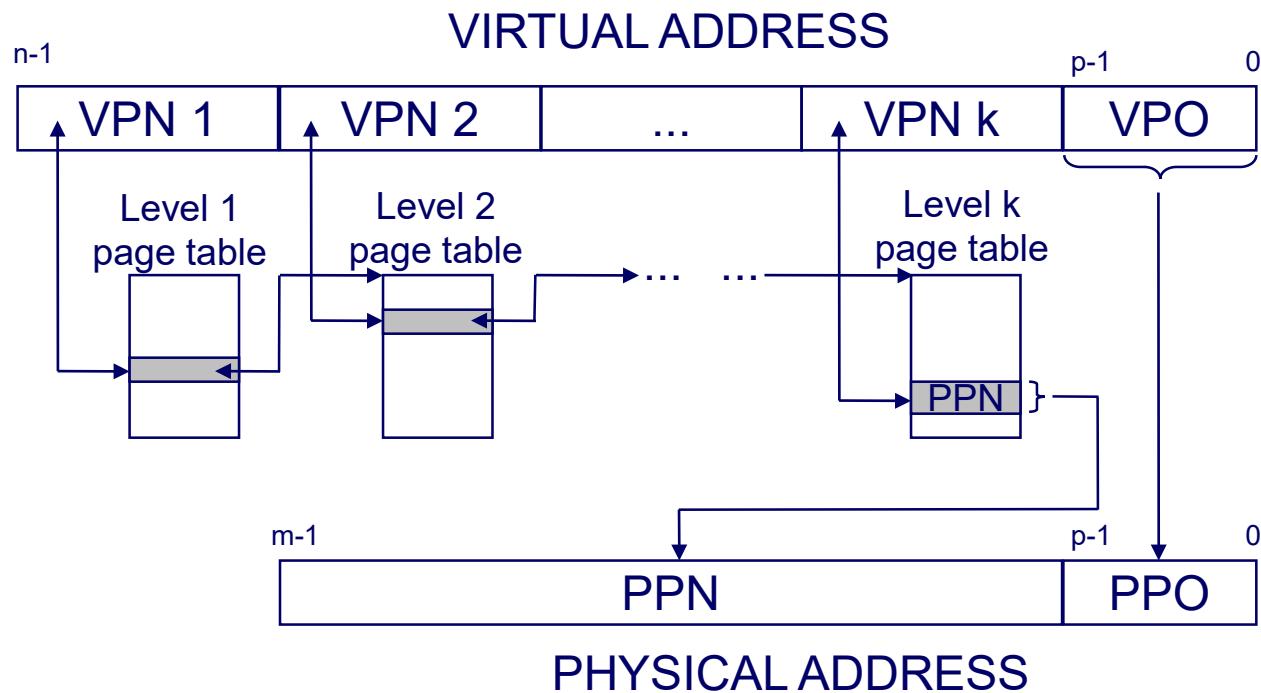
Recap: Integrating Caches and VM



Recap: Integrating Caches and VM



Translating with a k-level Page Table



Page Table Entry (PTE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame												Ignored	G	P A T	D	A	P C D	P W T	U S	R W	1	PTE: 4KB page										
Ignored																													0	PTE: not present		

- P: present
- R/W: readable / writable
- U/S: user / supervisor
- WT: write-through (1) / write-back (0)
- CD: cache disabled
- A: accessed (set by CPU, clear by OS)
- D: dirty
- PAT: page table attribute index
- G: global page (no TLB updating on the address if CR3 is reset)

Page Table Entry (PDE)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Bits 31:22 of address of 2MB page frame	Reserved (must be 0)	Bits 39:32 of address ²	P A T	Ignored	G	1	D	A	P C D	P W T	U /	R /	1	PDE: 4MB page																		
Address of page table					Ignored	0	I g n	A	P C D	P W T	U /	R /	1	PDE: page table																		
Ignored															0	PDE: not present																

- PS: page size (0=4KB, 1=4MB)
 - 4MB: super page

Question

- When the CPU is setting dirty bit of PTE, will it also set the dirty bit of corresponding PDE?
 - Let's check the manual!

Quiz#1

In **IA32** architecture, when the **CR4_PSE** bit in %cr4 register is set, then superpage mode is enabled, and the operating system can use either 4KB sized page or 4MB sized page. There are tradeoffs between small page and large page.

Consider the following scenarios:

- a) Dump the content of a large file of several hundred megabytes mapped to the RAM by mmap() function to another location:

```
memcpy(addr_of_new_location, addr_of_mapped_file, sizeof_file);
```

- b) Do the following calculation, where A, B are int pointers:

```
for(i=0; i<0x40000; ++i){  
    A[i] += B[i] & 0xFFFF;  
}
```

Which sized (4KB or 4MB) page performs better in each scenario?
Give your reasons.

PAE: Physical Address Extension

- Enabled when
 - CR0.PG=1 && CR4.PAE=1
- **32-bit** linear address -> **36-bit** physical address
 - Can be more than 36
- Maintains a set of 4 PDPT registers
 - Page directory pointer table
- 64-bit PTE
 - Instead of 32-bit

Address translation: PAE-4k

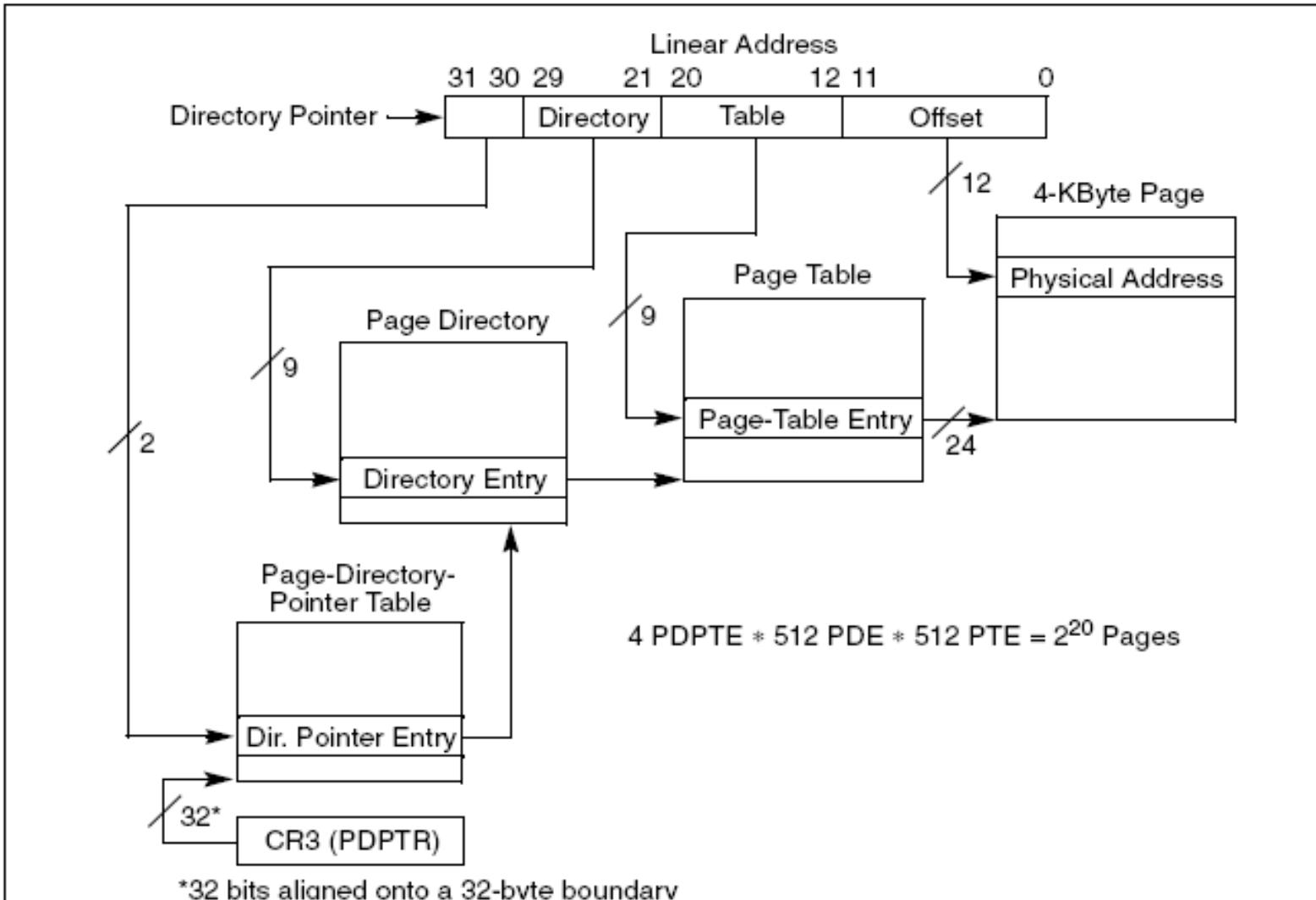


Figure 3-18. Linear Address Translation With PAE Enabled (4-KByte Pages)

Address translation: PAE-2M

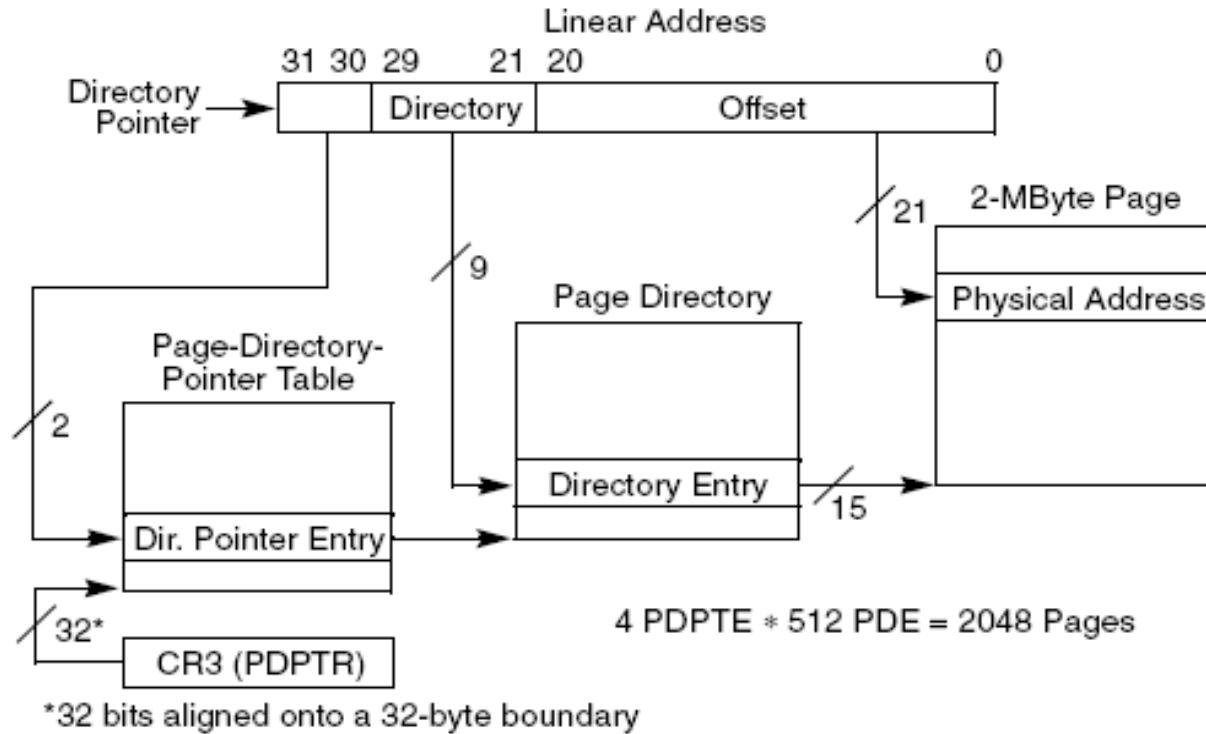
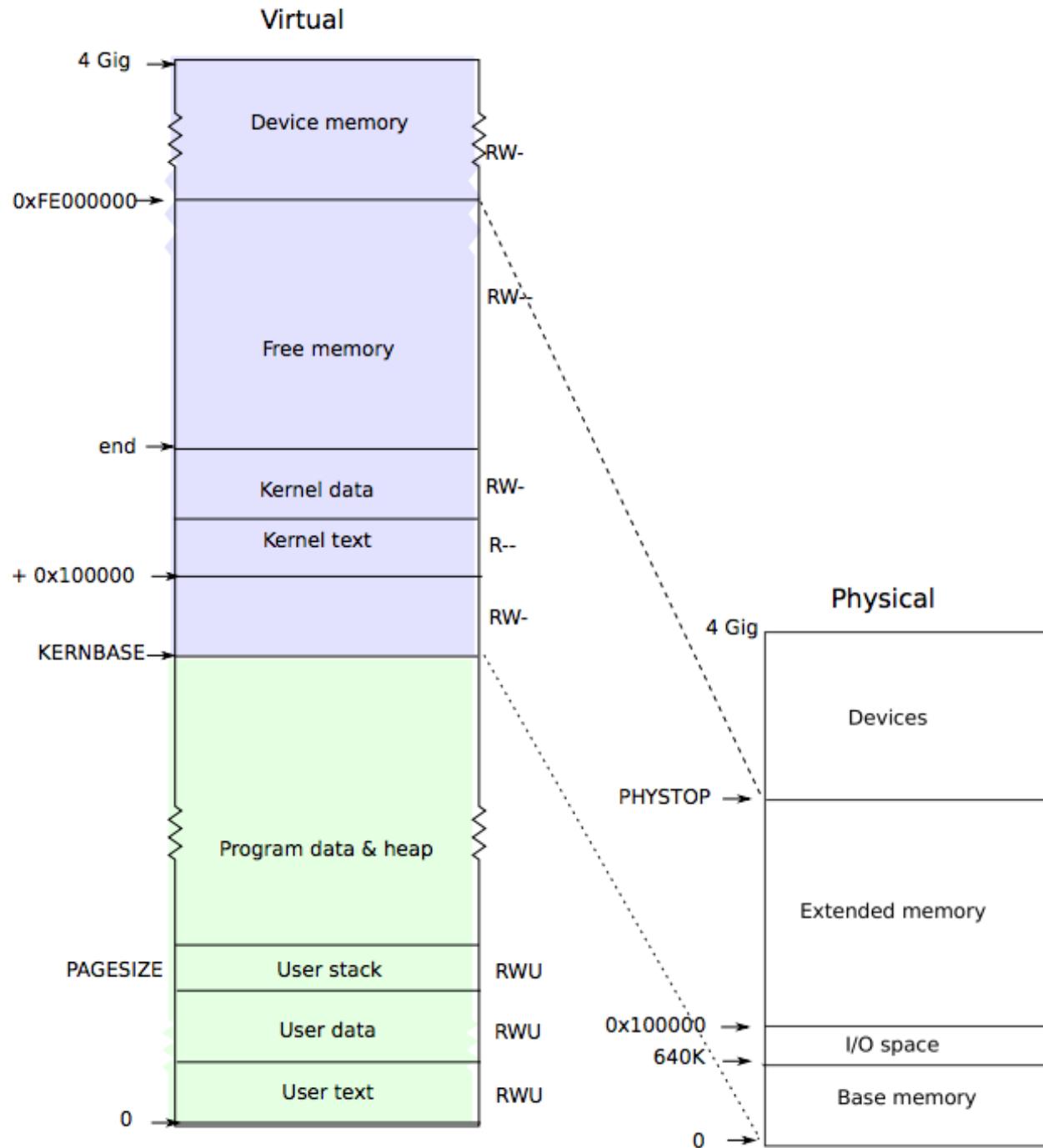


Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

Question: Why 2M instead of 4M?

```
17 int
18 main(void)
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc();      // kernel page table
22     mpinit();        // collect info about this machine
23     lapicinit();
24     seginit();       // set up segments
25     cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
26     picinit();       // interrupt controller
27     ioapicinit();    // another interrupt controller
28     consoleinit();   // I/O devices & their interrupts
29     uartinit();      // serial port
30     pinit();         // process table
31     tvinit();        // trap vectors
32     binit();         // buffer cache
33     fileinit();      // file table
34     iinit();         // inode cache
35     ideinit();       // disk
36     if(!ismp)
37         timerinit(); // uniprocessor timer
38     startothers();  // start other processors
39     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
40     userinit();      // first user process
41     // Finish setting up this processor in mpmain.
42     mpmain();
43 }
```

```
10 extern char data[]; // defined by kernel.ld
11 pde_t *kpgdir; // for use in scheduler()
12 struct segdesc gdt[NSEGS];
13
14 // Set up CPU's kernel segment descriptors.
15 // Run once on entry on each CPU.
16 void
17 seginit(void)
18 {
19     struct cpu *c;
20
21     // Map "logical" addresses to virtual addresses using identity map.
22     // Cannot share a CODE descriptor for both kernel and user
23     // because it would have to have DPL_USR, but the CPU forbids
24     // an interrupt from CPL=0 to DPL=3.
25     c = &cpus[cpunum()];
26     c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
27     c->gdt[SEG_KDATA] = SEG(STA_W, 0, 0xffffffff, 0);
28     c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
29     c->gdt[SEG_UDATA] = SEG(STA_W, 0, 0xffffffff, DPL_USER);
30
31     // Map cpu, and curproc
32     c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);
33
34     lgdt(c->gdt, sizeof(c->gdt));
35     loadgs(SEG_KCPU << 3);
36
37     // Initialize cpu-local storage.
38     cpu = c;
39     proc = 0;
40 }
```



big picture of xv6's virtual addressing scheme

0x00000000:0x80000000

 user addresses below KERNBASE

0x80000000:0x80100000

 map low 1MB devices (for kernel)

0x80100000:?

 kernel instructions/data

? :0x8E000000

 224 MB of DRAM mapped here

0xFE000000:0x00000000

 more memory-mapped devices

4 KB pages

entrypgdir was simple

array of 1024 PDEs, each mapping 4 MB, but 4 MB is too large a page size!

very wasteful if you have small processes xv6 programs are a few dozen kilobytes

4 MB pages require allocating full 4 MB of physical memory

solution: x86 MMU supports 4 KB pages

kvmalloc

```
17 int
18 main(void)
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc(); // kernel page table
22     mpinit(); // collect info about this machine

146 // Allocate one page table for the machine for the kernel address
147 // space for scheduler processes.
148 void
149 kvmalloc(void)
150 {
151     kpgdir = setupkvm();
152     switchkvm();
153 }

155 // Switch h/w page table register to the kernel-only page table,
156 // for when no process is running.
157 void
158 switchkvm(void)
159 {
160     lcr3(v2p(kpgdir)); // switch to the kernel page table
161 }
```

Setup_kvm

Usage

creates a page table

install mappings the kernel will need

will be called once for each new process

Process

must allocate PD, allocate some PTs, put mappings in PTEs

alloc allocates a physical page for the PD

returns that page's virtual address above 0x8000000

so we'll have to call V2P before installing in cr3

memset so that default is no translation (no P bit)

a call to mappages for each entry in kmap[]

```
127 // Set up kernel part of a page table.  
128 pde_t*  
129 setupkvm(void)  
130 {  
131     pde_t *pgdir;  
132     struct kmap *k;  
133  
134     if((pgdir = (pde_t*)kalloc()) == 0)  
135         return 0;  
136     memset(pgdir, 0, PGSIZE);  
137     if (p2v(PHYSTOP) > (void*)DEVSPACE)  
138         panic("PHYSTOP too high");  
139     for(k = kmap; k < &kmap[NELEM(kmap)]; k++)  
140         if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,  
141                     (uint)k->phys_start, k->perm) < 0)  
142             return 0;  
143     return pgdir;  
144 }
```

what is kmap[]?

an entry for each region of kernel virtual address space

same as address space setup from earlier in lecture

0x80000000 -> 0x00000000 (memory-mapped devices)

0x80100000 -> 0x00100000 (kernel instructions and data)

data -> data-0x80000000 (phys mem after where kernel was loaded)

DEVSPACE -> DEVSPACE (more memory mapped devices)

note no mappings below va 0x80000000 -- future user memory there

```
113 // This table defines the kernel's mappings, which are present in
114 // every process's page table.
115 static struct kmap {
116     void *virt;
117     uint phys_start;
118     uint phys_end;
119     int perm;
120 } kmap[] = {
121 { (void*)KERNBASE, 0,           EXTMEM,    PTE_W}, // I/O space
122 { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
123 { (void*)data,      V2P(data),    PHYSTOP,   PTE_W}, // kern data+memory
124 { (void*)DEVSPACE, DEVSPACE,    0,          PTE_W}, // more devices
125 };
```

```
67 // Create PTEs for virtual addresses starting at va that refer to
68 // physical addresses starting at pa. va and size might not
69 // be page-aligned.
70 static int
71 mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
72 {
73     char *a, *last;
74     pte_t *pte;
75
76     a = (char*)PGROUNDDOWN((uint)va);
77     last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
78     for(;;){
79         if((pte = walkpgdir(pgdir, a, 1)) == 0)
80             return -1;
81         if(*pte & PTE_P)
82             panic("remap");
83         *pte = pa | perm | PTE_P;
84         if(a == last)
85             break;
86         a += PGSIZE;
87         pa += PGSIZE;
88     }
89     return 0;
90 }
```

mappages

- arguments are PD, va, size, pa, perm
- adds mappings from a range of va's to corresponding pa's rounds because other uses pass in non-page-aligned addresses
- for each page-aligned address in the range, call walkpgdir to find address of PTE need the PTE's address (not just content) because we want to modify

```
42 // Return the address of the PTE in page table pgdir
43 // that corresponds to virtual address va.  If alloc!=0,
44 // create any required page table pages.
45 static pte_t *
46 walkpgdir(pde_t *pgdir, const void *va, int alloc)
47 {
48     pde_t *pde;
49     pte_t *pgtab;
50
51     pde = &pgdir[PDX(va)];
52     if(*pde & PTE_P){
53         pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
54     } else {
55         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
56             return 0;
57         // Make sure all those PTE_P bits are zero.
58         memset(pgtab, 0, PGSIZE);
59         // The permissions here are overly generous, but they can
60         // be further restricted by the permissions in the page table
61         // entries, if necessary.
62         *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
63     }
64     return &pgtab[PTX(va)];
65 }
```

```
13 static inline uint v2p(void *a) { return ((uint)(a)) - KERNBASE; }
14 static inline void *p2v(uint a) { return (void *)((a) + KERNBASE); }
```

walkpgdir

- mimics how the paging h/w finds the PTE for an address
- a little complex since xv6 allocates page-table pages lazily
 - might not be present, might have to allocate
- PDX extracts top ten bits
- &pgdir[PDX(va)] is the address of the relevant PDE
- now *pde is the PDE
- if PTE_P
 - the relevant page-table page already exists
 - PTE_ADDR extracts the PPN from the PDE
 - p2v() adds 0x80000000, since PTE holds physical address
- if not PTE_P
 - alloc a page-table page
 - fill in PDE with PPN -- thus v2p
- now the PTE we want is in the page-table page
 - at offset PTX(va) , which is 2nd 10 bits of va

```
16 // doing some setup required for memory allocator to work.
17 int
18 main(void)
19 {
20     kinit1(end, P2V(4*1024*1024)); // phys page allocator
21     kvmalloc();      // kernel page table
22     mpinit();        // collect info about this machine
23     lapicinit();
24     seginit();        // set up segments
25     cprintf("\ncpu%d: starting xv6\n\n", cpu->id);
26     picinit();        // interrupt controller
27     ioapicinit();    // another interrupt controller
28     consoleinit();   // I/O devices & their interrupts
29     uartinit();      // serial port
30     pinit();          // process table
31     tvinit();         // trap vectors
32     binit();          // buffer cache
33     fileinit();       // file table
34     iinit();          // inode cache
35     ideinit();        // disk
36     if(!ismp)
37         timerinit();   // uniprocessor timer
38     startothers();    // start other processors
39     kinit2(P2V(4*1024*1024), P2V(PHYSTOP)); // must come after startothers()
40     userinit();       // first user process
41     // Finish setting up this processor in mpmain.
42     mpmain();
43 }
```

```
78 void
79 userinit(void)
80 {
81     struct proc *p;
82     extern char _binary_initcode_start[], _binary_initcode_size[];
83
84     p = allocproc();
85     initproc = p;
86     if((p->pgdir = setupkvm()) == 0)
87         panic("userinit: out of memory?");
88     inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
89     p->sz = PGSIZE;
90     memset(p->tf, 0, sizeof(*p->tf));
91     p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
92     p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
93     p->tf->es = p->tf->ds;
94     p->tf->ss = p->tf->ds;
95     p->tf->eflags = FL_IF;
96     p->tf->esp = PGSIZE;
97     p->tf->eip = 0; // beginning of initcode.S
98
99     safestrcpy(p->name, "initcode", sizeof(p->name));
100    p->cwd = namei("/");
101
102    p->state = RUNNABLE;
103 }
```

User VM mapping

Create a process

Inituvm : initialized user VM

Loaduvm: load program segment

Context switch

switchuvm

Grow/shrink/free VM

allocuvm

deallocuvm

Freeuvm

Fork

copyuvm

```
179 // Load the initcode into address 0 of pgdir.  
180 // sz must be less than a page.  
181 void  
182 inituvm(pde_t *pgdir, char *init, uint sz)  
183 {  
184     char *mem;  
185  
186     if(sz >= PGSIZE)  
187         panic("inituvm: more than a page");  
188     mem = kalloc();  
189     memset(mem, 0, PGSIZE);  
190     mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);  
191     memmove(mem, init, sz);  
192 }
```

```
79 // Allocate one 4096-byte page of physical memory.
80 // Returns a pointer that the kernel can use.
81 // Returns 0 if the memory cannot be allocated.
82 char*
83 kalloc(void)
84 {
85     struct run *r;
86
87     if(kmem.use_lock)
88         acquire(&kmem.lock);
89     r = kmem.freelist;
90     if(r)
91         kmem.freelist = r->next;
92     if(kmem.use_lock)
93         release(&kmem.lock);
94     return (char*)r;
95 }
```

```
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
    pushcli();
    cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
    cpu->gdt[SEG_TSS].s = 0;
    cpu->ts.ss0 = SEG_KDATA << 3;
    cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
    ltr(SEG_TSS << 3);
    if(p->pgdir == 0)
        panic("switchuvm: no pgdir");
    lcr3(v2p(p->pgdir)); // switch to new address space
    popcli();
}
```

```
// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
    uint i, pa, n;
    pte_t *pte;

    if((uint)addr % PGSIZE != 0)
        panic("loaduvm: addr must be page aligned");
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
            panic("loaduvm: address should exist");
        pa = PTE_ADDR(*pte);
        if(sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if(readi(ip, p2v(pa), offset+i, n) != n)
            return -1;
    }
    return 0;
}
```

```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned. Returns new size or 0 on error.
int
allocuvvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    char *mem;
    uint a;

    if(newsz >= KERNBASE)
        return 0;
    if(newsz < oldsz)
        return oldsz;

    a = PGROUNDUP(oldsz);
    for(; a < newsz; a += PGSIZE){
        mem = kalloc();
        if(mem == 0){
            printf("allocuvvm out of memory\n");
            deallocuvvm(pgdir, newsz, oldsz);
            return 0;
        }
        memset(mem, 0, PGSIZE);
        mappages(pgdir, (char*)a, PGSIZE, v2p(mem), PTE_W|PTE_U);
    }
    return newsz;
}
```

```
// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
    pte_t *pte;
    uint a, pa;

    if(newsz >= oldsz)
        return oldsz;

    a = PGROUNDUP(newsz);
    for(; a < oldsz; a += PGSIZE){
        pte = walkpgdir(pgdir, (char*)a, 0);
        if(!pte)
            a += (NPTENTRIES - 1) * PGSIZE;
        else if((*pte & PTE_P) != 0){
            pa = PTE_ADDR(*pte);
            if(pa == 0)
                panic("kfree");
            char *v = p2v(pa);
            kfree(v);
            *pte = 0;
        }
    }
    return newsz;
}
```

```
// Free a page table and all the physical memory pages
// in the user part.
void
freevm(pde_t *pgdir)
{
    uint i;

    if(pgdir == 0)
        panic("freevm: no pgdir");
    deallocuvm(pgdir, KERNBASE, 0);
    for(i = 0; i < NPENTRIES; i++){
        if(pgdir[i] & PTE_P){
            char * v = p2v(PTE_ADDR(pgdir[i]));
            kfree(v);
        }
    }
    kfree((char*)pgdir);
}
```

```
// Given a parent process's page table, create a copy
// of it for a child.
pde_t*
copyuvm(pde_t *pgdir, uint sz)
{
    pde_t *d;
    pte_t *pte;
    uint pa, i;
    char *mem;

    if((d = setupkvm()) == 0)
        return 0;
    for(i = 0; i < sz; i += PGSIZE){
        if((pte = walkpgdir(pgdir, (void *) i, 0)) == 0)
            panic("copyuvm: pte should exist");
        if(!(pte & PTE_P))
            panic("copyuvm: page not present");
        pa = PTE_ADDR(*pte);
        if((mem = kalloc()) == 0)
            goto bad;
        memmove(mem, (char*)p2v(pa), PGSIZE);
        if(mappages(d, (void*)i, PGSIZE, v2p(mem), PTE_W|PTE_U) < 0)
            goto bad;
    }
    return d;
}

bad:
    freevm(d);
    return 0;
}
```

```
// Map user virtual address to kernel address.
char*
uva2ka(pde_t *pgdir, char *uva)
{
    pte_t *pte;

    pte = walkpgdir(pgdir, uva, 0);
    if((*pte & PTE_P) == 0)
        return 0;
    if((*pte & PTE_U) == 0)
        return 0;
    return (char*)p2v(PTE_ADDR(*pte));
}
```

Question

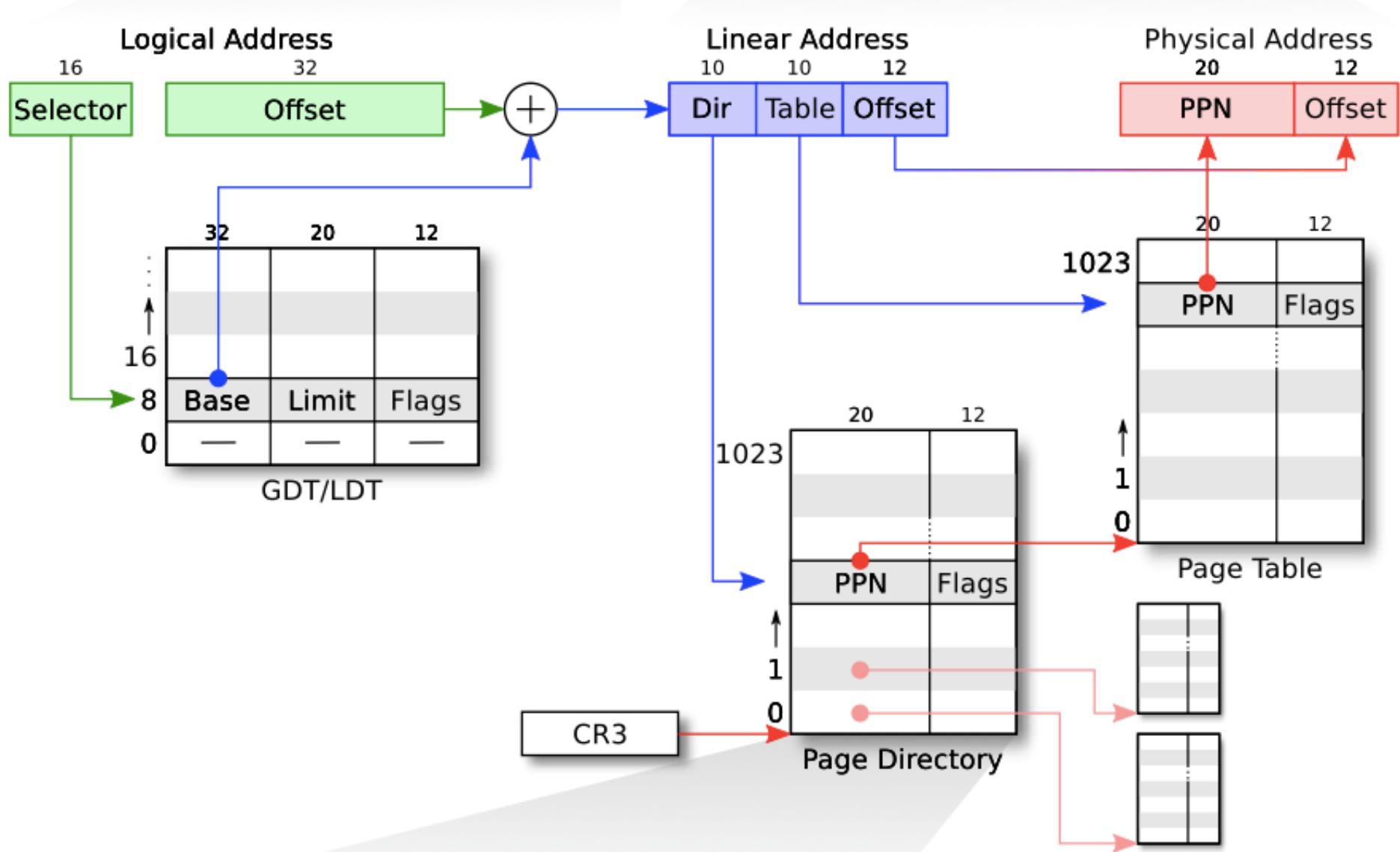
For large page, sometimes we say 2M (PSE), sometimes we say 4M (TLB), what is the difference?

Processes

Yubin Xia
IPADS, SJTU

ACKs: Some slides are adapted from the textbook's original slides
and Frans's OS course notes

Review: LA->LA->PA



Review

- Booting: enabling segment and paging
 - Segment: set GDT & use long jmp (ljmp)
 - Paging: temporary page table (entrypgdir) with kernel mapped at both 0x80100000 and 0x100000
- Different page size: 4KB vs. 4MB(2MB)

PROCESS IN XV6

Processes Outline

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Process: Big Picture

- More programs than processors
 - How to share the limited number of processors among the programs?
- Observation: most programs do not need the processor continuously
 - because they frequently have to wait for input (from user, disk, network, etc.)
- Idea: when one program must wait, it releases the processor, and gives it to another program

Threads

- Mechanism: thread of computation, an active computation.
 - A thread is an abstraction that contains
 - the *minimal state* that is necessary to **stop** an active and **resume** it at some point later
 - The state depends on the processor
 - On x86, it is the processor registers

Threads in xv6

```
34 // Saved registers for kernel context switches.  
35 // Don't need to save all the segment registers (%cs, etc),  
36 // because they are constant across kernel contexts.  
37 // Don't need to save %eax, %ecx, %edx, because the  
38 // x86 convention is that the caller has saved them.  
39 // Contexts are stored at the bottom of the stack they  
40 // describe; the stack pointer is the address of the context.  
41 // The layout of the context matches the layout of the stack in swtch.S  
42 // at the "Switch stacks" comment. Switch doesn't save eip explicitly,  
43 // but it is on the stack and allocproc() manipulates it.  
44 struct context {  
45     uint edi;  
46     uint esi;  
47     uint ebx;  
48     uint ebp;  
49     uint eip;  
50 };
```

Address spaces and threads

- Address spaces and threads are in principle **independent** concepts
 - One can switch from one thread to another thread in the same address space
 - or one can switch from one thread to another thread in another address space

Process Concepts

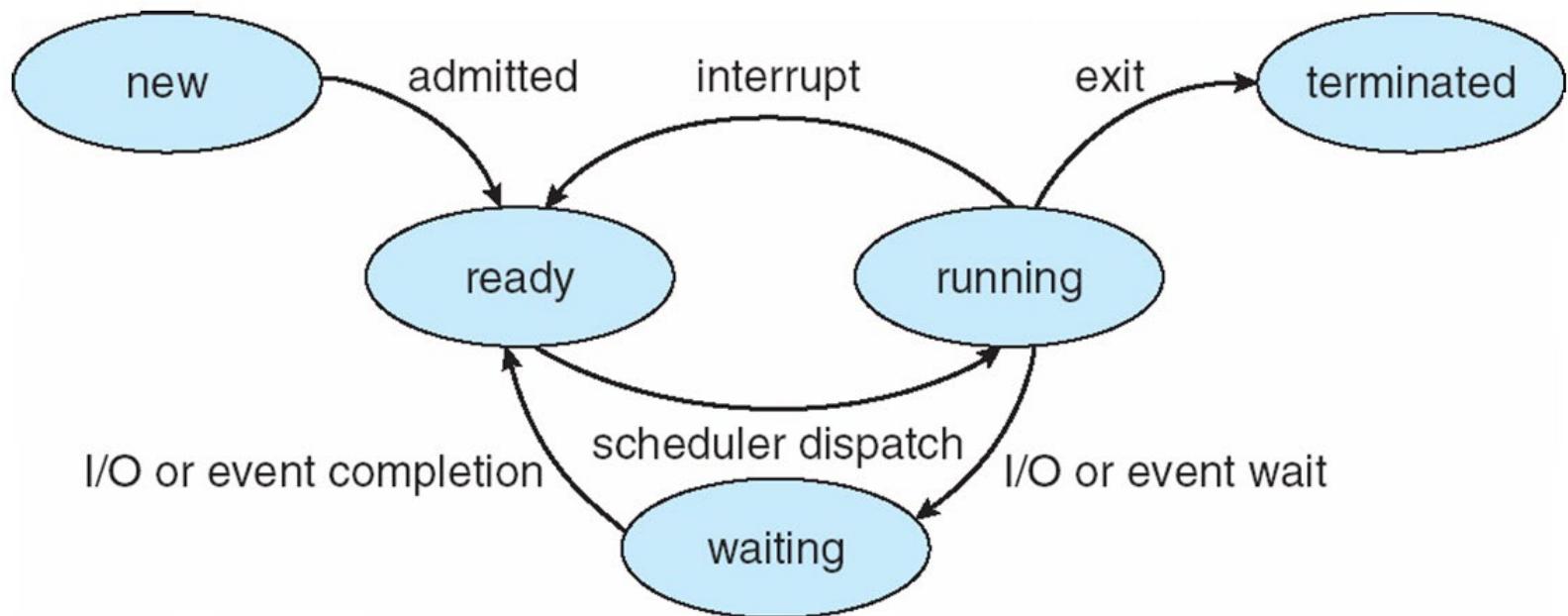
- Process: a program in execution
 - one address space + one or more threads of computation
- A process includes:
 - program counter, stack, data section
- In xv6 all *user* programs contain one thread of computation and one address space
 - Q: *How to support multi-threading in a process?*

Process State

- As a process executes, it changes *state*
 - **new**: The process is being created
 - **running**: Instructions are being executed
 - **waiting**: The process is waiting for some event to occur
 - **ready**: The process is waiting to be assigned to a processor
 - **terminated**: The process has finished execution
- In xv6

```
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Diagram of Process State

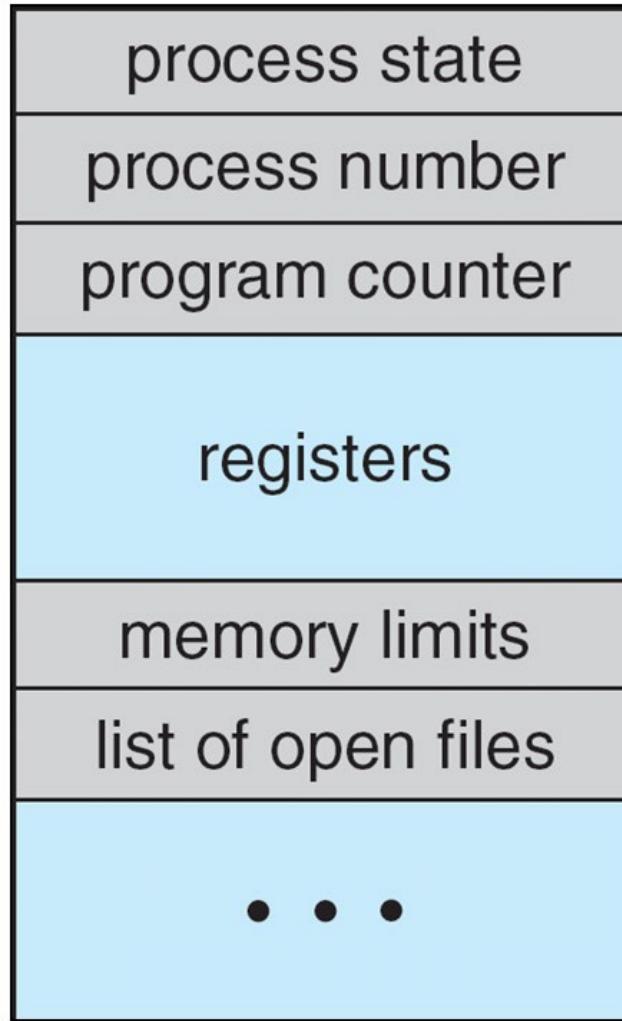


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

Process Control Block (PCB)

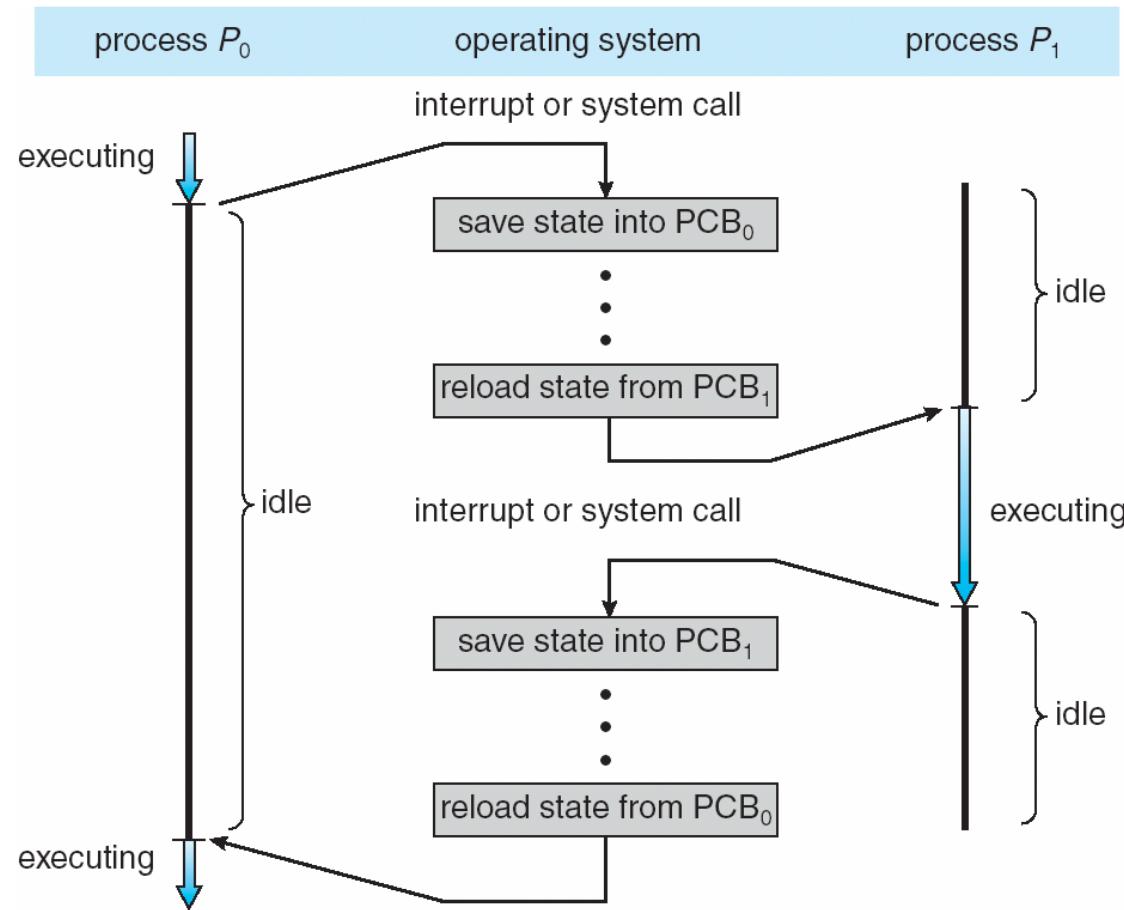


Process in xv6 (proc.c)

Context Switch

- When CPU switches to another process, the system must do it via a **context switch**
 - save the state of the old process
 - load the saved state for the new process **Context** of a process represented in the PCB
- Context-switch time is overhead
 - the system does no useful work while switching
 - Time dependent on hardware support

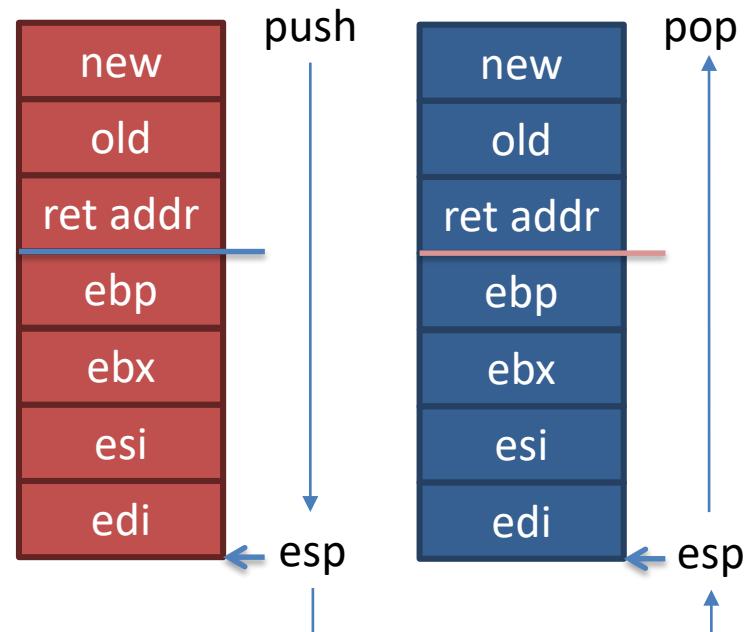
CPU Switch From Process to Process



Context Switch in xv6 (swtch.S)

```
5 # Save current register context in old
6 # and then load register context from new.
7
8 .globl swtch
9 swtch:
10    movl 4(%esp), %eax
11    movl 8(%esp), %edx
12
13    # Save old callee-save registers
14    pushl %ebp
15    pushl %ebx
16    pushl %esi
17    pushl %edi
18
19    # Switch stacks
20    movl %esp, (%eax)
21    movl %edx, %esp
22
23    # Load new callee-save registers
24    popl %edi
25    popl %esi
26    popl %ebx
27    popl %ebp
28    ret
```

```
44 struct context {
45     uint edi;
46     uint esi;
47     uint ebx;
48     uint ebp;
49     uint eip;
50 };
```



```

258 scheduler(void)
259 {
260     struct proc *p;
261
262     for(;;){
263         // Enable interrupts on this processor.
264         sti();
265
266         // Loop over process table looking for process to run.
267         acquire(&ptable.lock);
268         for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
269             if(p->state != RUNNABLE)
270                 continue;
271
272             // Switch to chosen process. It is the process's job
273             // to release ptable.lock and then reacquire it
274             // before jumping back to us.
275             proc = p;
276             switchuvm(p);
277             p->state = RUNNING;
278             swtch(&cpu->scheduler, proc->context); ←
279             switchkvm(); ←
280
281             // Process is done running for now.
282             // It should have changed its p->state before coming back.
283             proc = 0;
284         }
285         release(&ptable.lock);
286
287     }
288 }
```

```

290 // Enter scheduler. Must hold only ptable.lock
291 // and have changed proc->state.
292 void
293 sched(void)
294 {
295     int intena;
296
297     if(!holding(&ptable.lock))
298         panic("sched ptable.lock");
299     if(cpu->ncli != 1)
300         panic("sched locks");
301     if(proc->state == RUNNING)
302         panic("sched running");
303     if(readeflags()&FL_IF)
304         panic("sched interruptible");
305     intena = cpu->intena;
306     swtch(&proc->context, cpu->scheduler);
307     cpu->intena = intena;
308 }
```

User space

User -> kernel -> user

```
310 // Give up the CPU for one scheduling round.  
311 void  
312 yield(void)  
313 {  
314     acquire(&ptable.lock); //DOC: yieldlock  
315     proc->state = RUNNABLE;  
316     sched();  
317     release(&ptable.lock);  
318 }
```

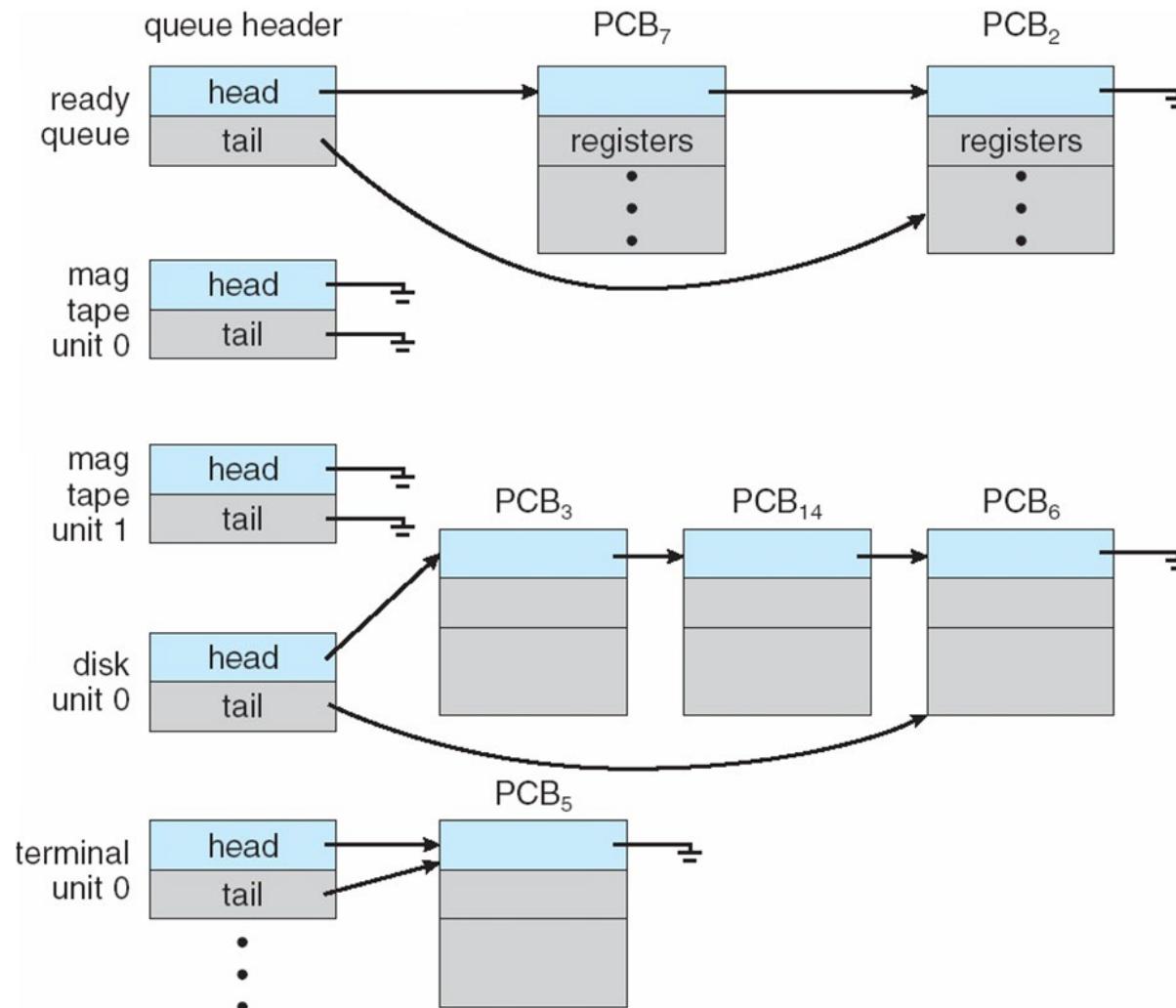
```
342 void  
343 sleep(void *chan, struct spinlock *lk)  
344 {  
345     if(proc == 0)  
346         panic("sleep");  
347  
348     if(lk == 0)  
349         panic("sleep without lk");  
350  
351     // Must acquire ptable.lock in order to  
352     // change p->state and then call sched.  
353     // Once we hold ptable.lock, we can be  
354     // guaranteed that we won't miss any wakeup  
355     // (wakeup runs with ptable.lock locked),  
356     // so it's okay to release lk.  
357     if(lk != &ptable.lock){ //DOC: sleeplock0  
358         acquire(&ptable.lock); //DOC: sleeplock1  
359         release(lk);  
360     }  
361  
362     // Go to sleep.  
363     proc->chan = chan;  
364     proc->state = SLEEPING;  
365     sched();
```

```
166 void  
167 exit(void)  
168 {  
169     struct proc *p;  
170     int fd;  
171  
172     if(proc == initproc)  
173         panic("init exiting");  
174  
175     // Close all open files.  
176     for(fd = 0; fd < NOFILE; fd++){  
177         if(proc->ofile[fd]){  
178             fileclose(proc->ofile[fd]);  
179             proc->ofile[fd] = 0;  
180         }  
181     }  
182  
183     iput(proc->cwd);  
184     proc->cwd = 0;  
185  
186     acquire(&ptable.lock);  
187  
188     // Parent might be sleeping in wait().  
189     wakeup1(proc->parent);  
190  
191     // Pass abandoned children to init.  
192     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){  
193         if(p->parent == proc){  
194             p->parent = initproc;  
195             if(p->state == ZOMBIE)  
196                 wakeup1(initproc);  
197         }  
198     }  
199  
200     // Jump into the scheduler, never to return.  
201     proc->state = ZOMBIE;  
202     sched();  
203     panic("zombie exit");  
204 }
```

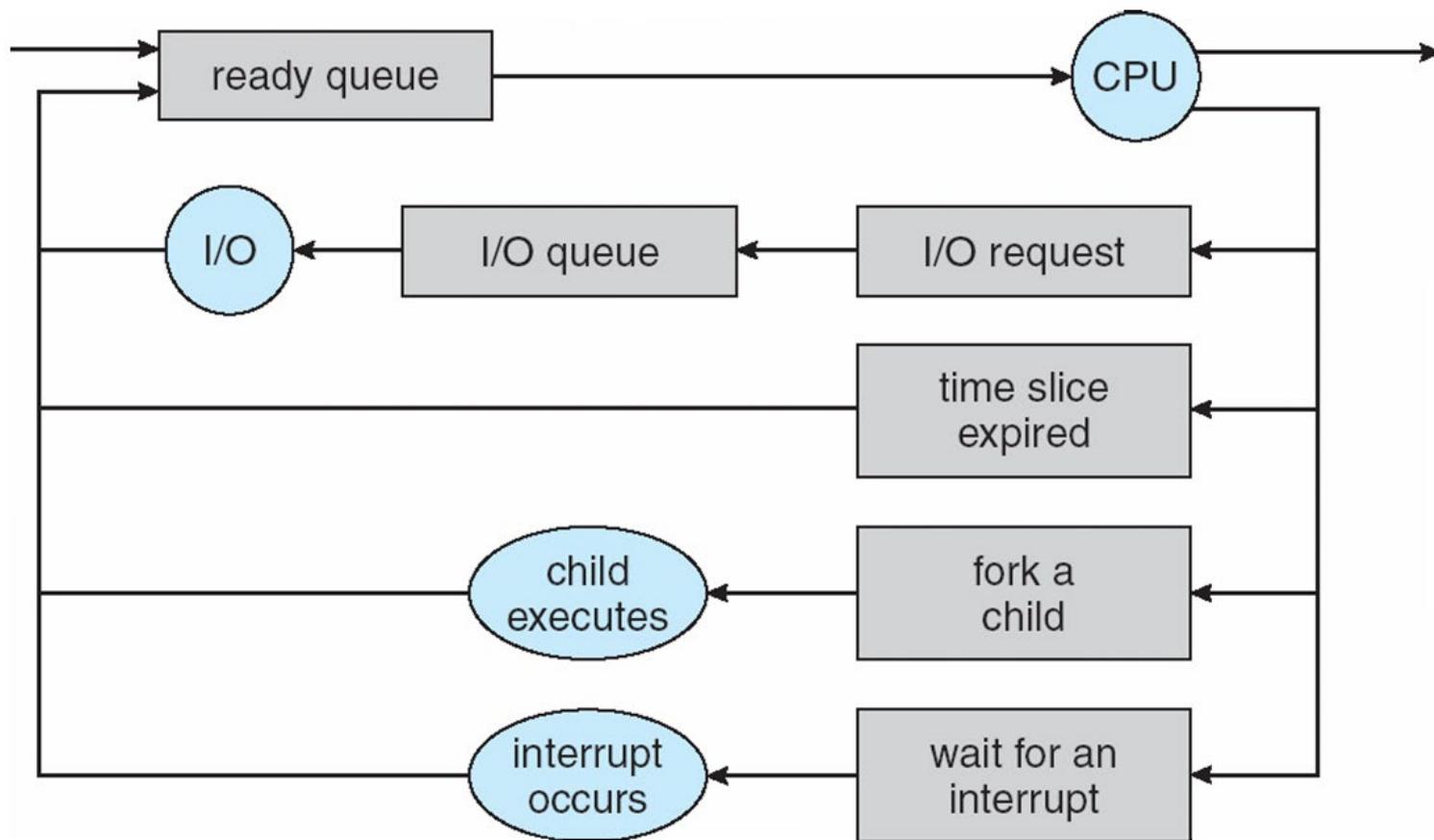
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue & Various I/O Device Queues



Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) \Rightarrow (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) \Rightarrow (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*

Scheduler (cont.)

- Processes can be described as either:
 - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
 - **CPU-bound process** – spends more time doing computations; few very long CPU bursts

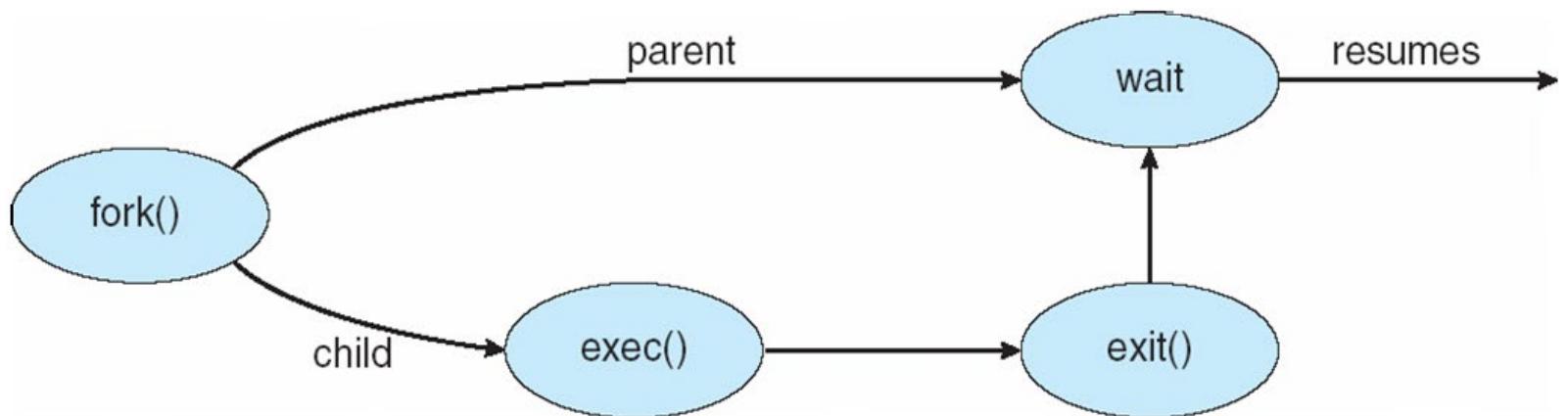
Process Creation

- **Parent** process create **children** processes
 - forming a tree of processes
 - Generally, process identified and managed via **a process identifier (pid)**
- Resource sharing policies
 - share all resources
 - share subset of parent's resources
 - share no resources

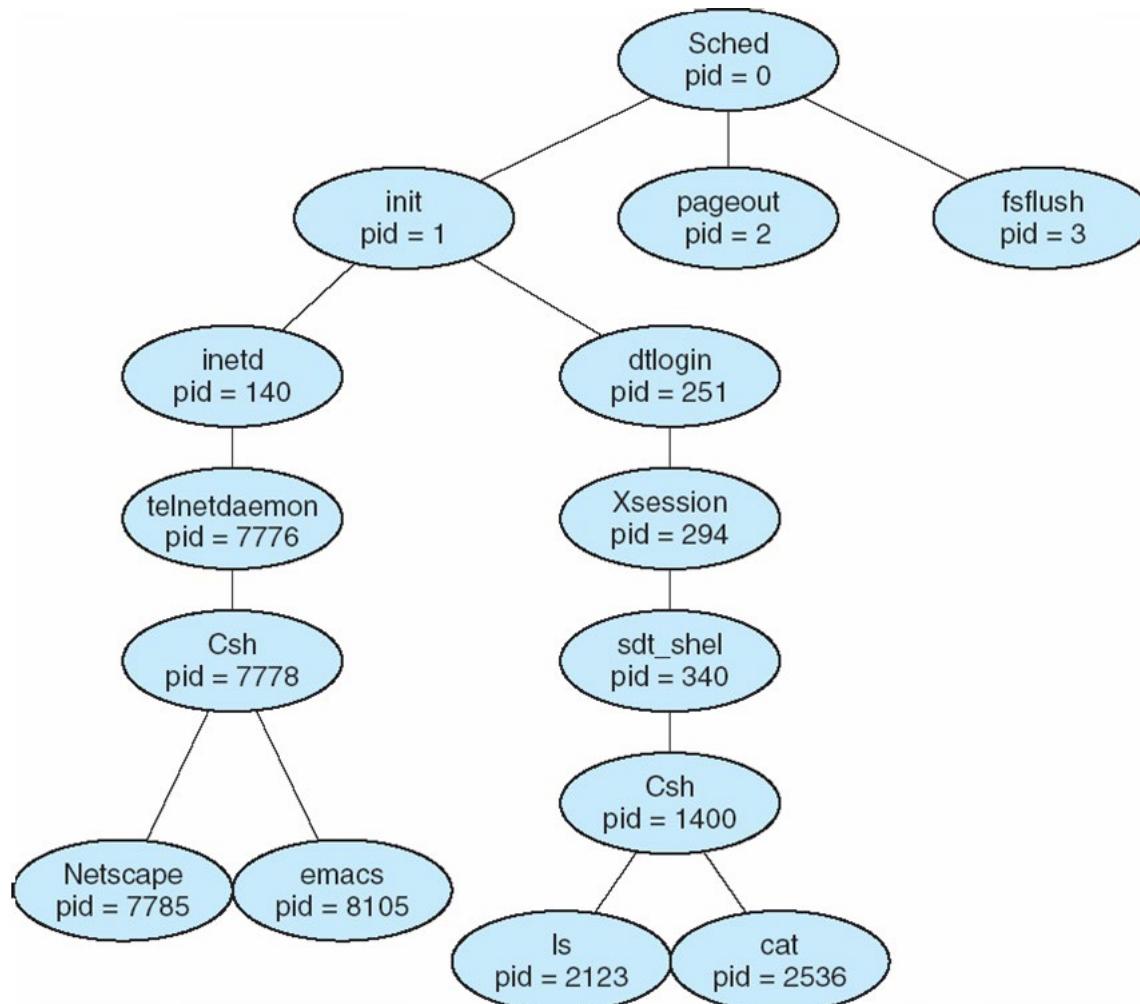
Process Creation (Cont)

- Execution
 - Parent and children execute concurrently
 - Parent waits until children terminate
- Address space
 - Child duplicate of parent's
 - Child has a program loaded into it
- UNIX examples
 - **fork** system call creates new process
 - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation



A tree of processes on a typical Solaris



Process Creation in xv6

```
125 // Create a new process copying p as the parent.
126 // Sets up stack to return as if from system call.
127 // Caller must set state of returned proc to RUNNABLE.
128 int
129 fork(void)
130 {
131     int i, pid;
132     struct proc *np;
133
134     // Allocate process.
135     if((np = allocproc()) == 0)
136         return -1;
137
138     // Copy process state from p.
139     if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
140         kfree(np->kstack);
141         np->kstack = 0;
142         np->state = UNUSED;
143         return -1;
144     }
145     np->sz = proc->sz;
146     np->parent = proc;
147     *np->tf = *proc->tf;
148
149     // Clear %eax so that fork returns 0 in the child.
150     np->tf->eax = 0;
151
152     for(i = 0; i < NOFILE; i++)
153         if(proc->ofile[i])
154             np->ofile[i] = filedup(proc->ofile[i]);
155     np->cwd = idup(proc->cwd);
156
157     pid = np->pid;
158     np->state = RUNNABLE;
159     safestrcpy(np->name, proc->name, sizeof(proc->name));
160     return pid;
161 }
```

Read following code

- allocproc()

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required

Process Termination (Cont.)

- If parent is exiting
 - Some operating system do not allow child to continue if its parent terminates
 - All children terminated - **cascading termination**

```
167 exit(void)
168 {
169     struct proc *p;
170     int fd;
171
172     if(proc == initproc)
173         panic("init exiting");
174
175     // Close all open files.
176     for(fd = 0; fd < NOFILE; fd++){
177         if(proc->ofile[fd]){
178             fileclose(proc->ofile[fd]);
179             proc->ofile[fd] = 0;
180         }
181     }
182
183     iput(proc->cwd);
184     proc->cwd = 0;
185
186     acquire(&ptable.lock);
187
188     // Parent might be sleeping in wait().
189     wakeup1(proc->parent);
190
191     // Pass abandoned children to init.
192     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
193         if(p->parent == proc){
194             p->parent = initproc;
195             if(p->state == ZOMBIE)
196                 wakeup1(initproc);
197         }
198     }
199
200     // Jump into the scheduler, never to return.
201     proc->state = ZOMBIE;
202     sched();
203     panic("zombie exit");
204 }
```

Homework

- In xv6, one process contains only one thread. If we allow one process contains multiple threads, which state of the original PCB (process control block) should be per-thread and which should be per-process?

Inter Process Communication

Yubin Xia
IPADS, SJTU

Credit to Timothy Roscoe, ETH

Review: Thread

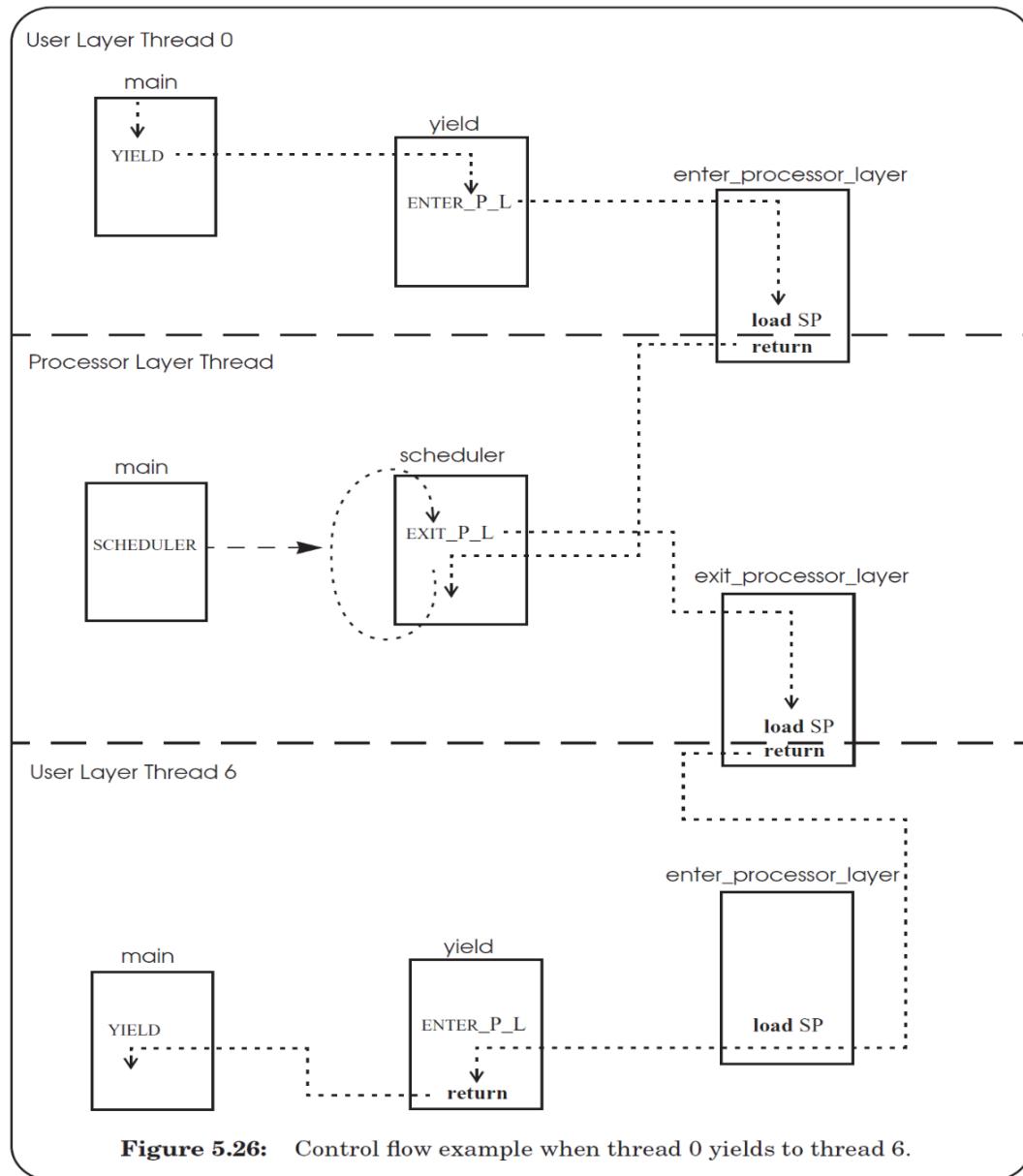
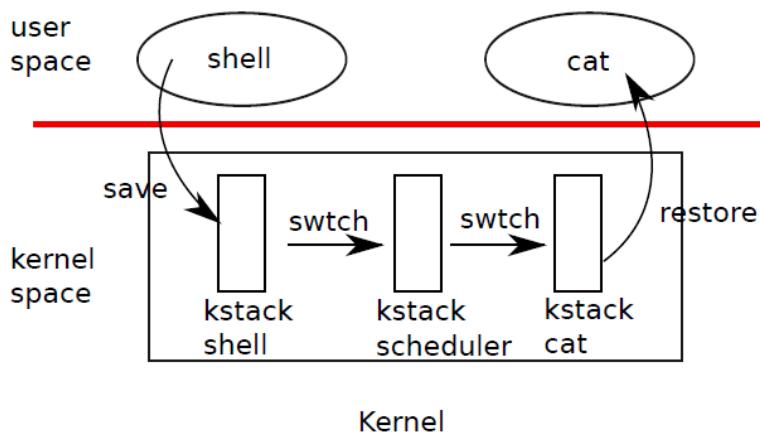


Figure 5.26: Control flow example when thread 0 yields to thread 6.

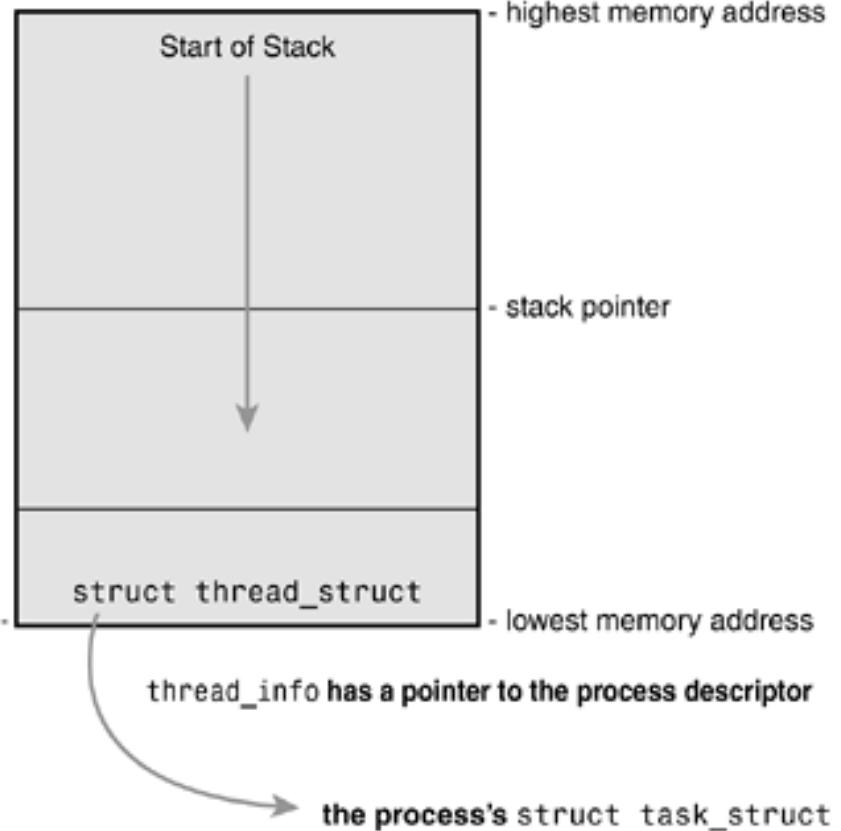
Kernel Stack and thread_struct

- In Linux:

```
struct thread_info {  
    struct task_struct *task;  
    struct exec_domain *exec_domain;  
    unsigned long flags;  
    unsigned long status;  
    __u32 cpu;  
    __s32 preempt_count;  
    mm_segment_t addr_limit;  
    struct restart_block restart_block;  
    unsigned long previous_esp;  
    __u8 supervisor_stack[0];  
};
```

current_thread_info () ->

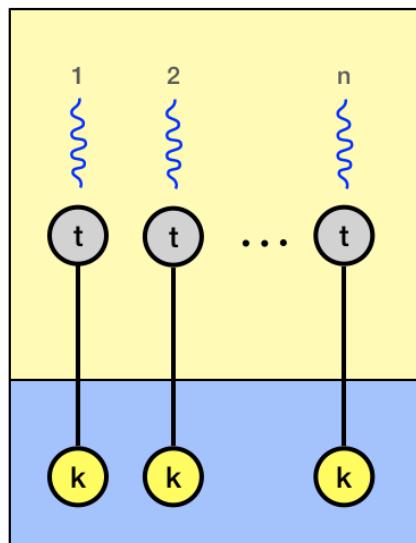
Process Kernel Stack



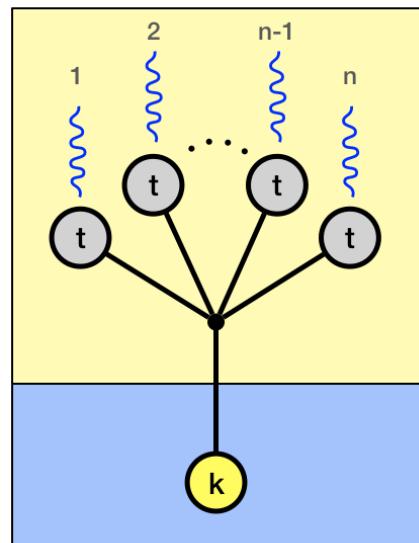
Review Questions

- Q: What is the difference between context and thread?
- Q: What is the difference between a kernel thread and a user thread?
- Q: Does every user-level thread has a kernel-level thread?
- Q: Which one is selected by a scheduler? Context, thread or process?
- Q: The number of kernel stack equals to the number of which: context, thread or process?
- Q: Can a kernel thread has multiple contexts?
- Q: When all CPU cores are running in user-level, all the kernel stacks are empty?
- Context is a concept in kernel, because user cannot switch stack
- Q: To fully utilize a multi-core CPU, which threading model should be used: user-level thread or kernel thread?
- Q: What is the difference between user-level threading and co-routine?

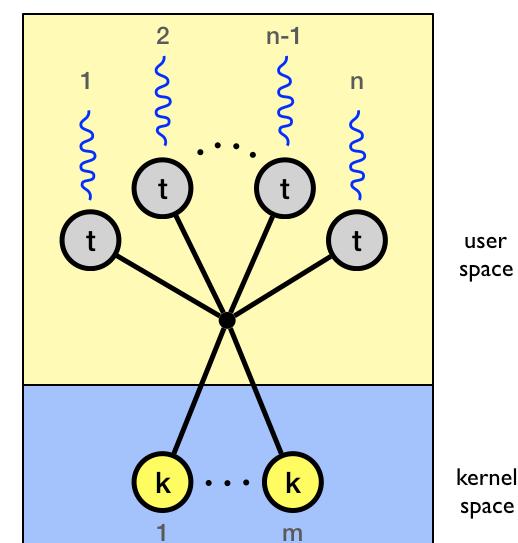
Threading Model



1:1



1:N



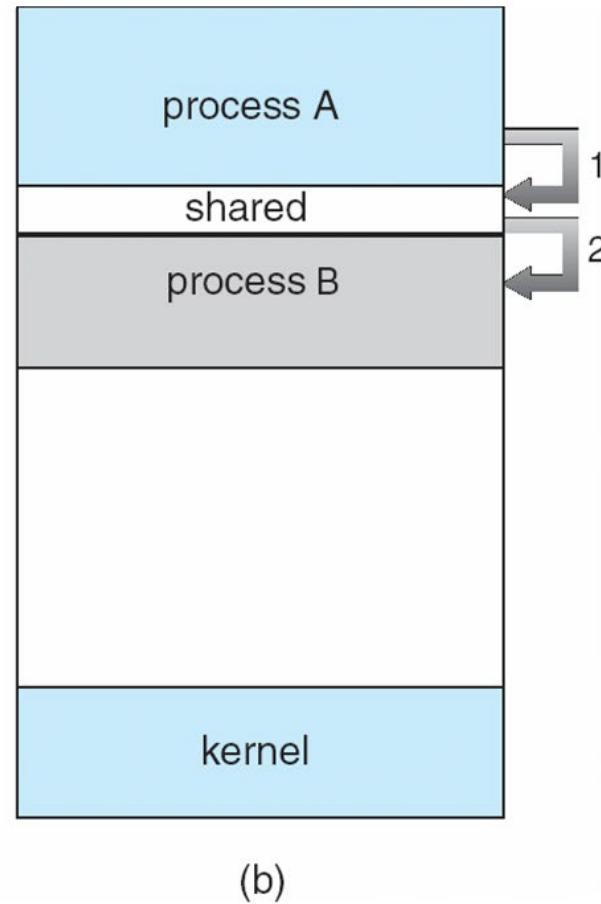
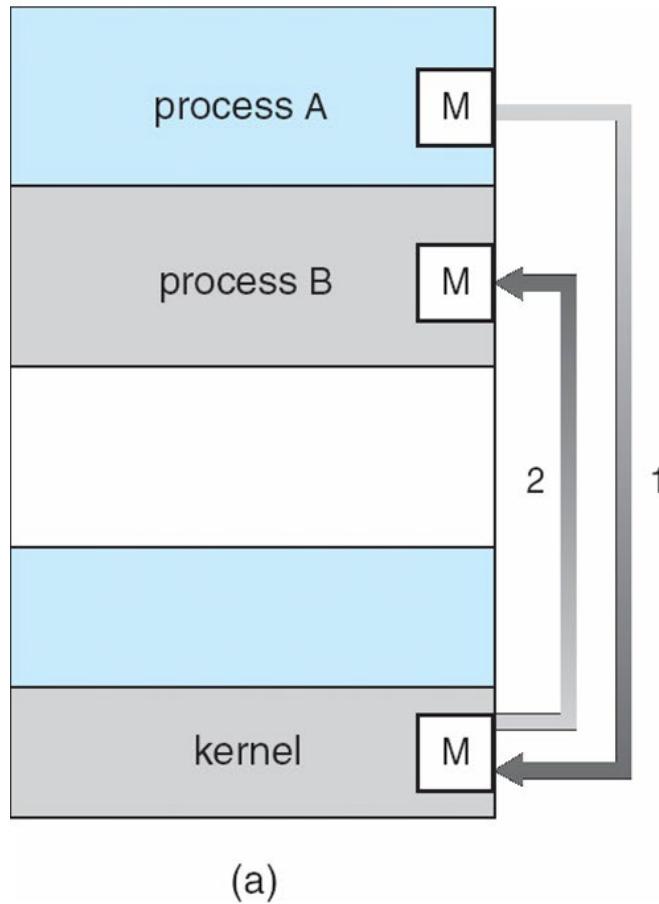
M:N

IPC

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
 - Information sharing
 - Computation speed-up
 - Modularity
 - Convenience

Communications Models



IPC

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
 - Shared memory
 - Message passing

Producer-Consumer Problem

- *Producer* process produces information that is consumed by a *consumer* process
 - *unbounded-buffer* places no practical limit on the size of the buffer
 - *bounded-buffer* assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
```

```
typedef struct {
```

```
    . . .
```

```
} item;
```

```
item buffer[BUFFER_SIZE];
```

```
int in = 0;
```

```
int out = 0;
```

- Solution is correct, but can only use $\text{BUFFER_SIZE}-1$ elements

Bounded-Buffer – Producer

```
while (true) {  
    /* Produce an item */  
    while (((in = (in + 1) % BUFFER SIZE count)  
          == out)  
        ; /* do nothing -- no free buffers */  
    buffer[in] = item;  
    in = (in + 1) % BUFFER SIZE;  
}
```

Bounded Buffer – Consumer

```
while (true) {  
    while (in == out)  
        ; // do nothing -- nothing to consume  
  
    // remove an item from the buffer  
    item = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
    return item;  
}
```

Example of IPC in xv6: Pipe (pipe.c)

```
12 struct pipe {  
13     struct spinlock lock;  
14     char data[PIPESIZE];  
15     uint nread;      // number of bytes read  
16     uint nwrite;    // number of bytes written  
17     int readopen;   // read fd is still open  
18     int writeopen;  // write fd is still open  
19 };
```

```
77 int
78 pipewrite(struct pipe *p, char *addr, int n)
79 {
80     int i;
81
82     acquire(&p->lock);
83     for(i = 0; i < n; i++){
84         while(p->nwrite == p->nread + PIPESIZE){ //DOC: pipewrite-full
85             if(p->readopen == 0 || proc->killed){
86                 release(&p->lock);
87                 return -1;
88             }
89             wakeup(&p->nread);
90             sleep(&p->nwrite, &p->lock); //DOC: pipewrite-sleep
91         }
92         p->data[p->nwrite++ % PIPESIZE] = addr[i];
93     }
94     wakeup(&p->nread); //DOC: pipewrite-wakeup1
95     release(&p->lock);
96     return n;
97 }
```

```
99 int
100 piperead(struct pipe *p, char *addr, int n)
101 {
102     int i;
103
104     acquire(&p->lock);
105     while(p->nread == p->nwrite && p->writeopen){ //DOC: pipe-empty
106         if(proc->killed){
107             release(&p->lock);
108             return -1;
109         }
110         sleep(&p->nread, &p->lock); //DOC: piperead-sleep
111     }
112     for(i = 0; i < n; i++){ //DOC: piperead-copy
113         if(p->nread == p->nwrite)
114             break;
115         addr[i] = p->data[p->nread++ % PIPESIZE];
116     }
117     wakeup(&p->nwrite); //DOC: piperead-wakeup
118     release(&p->lock);
119     return i;
120 }
```

IPC – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system
 - processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(message) – message size fixed or variable
 - **receive**(message)

Message Passing (Cont.)

- If P and Q wish to communicate, they need to:
 - establish a *communication link* between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
 - **send** (P , message) – send a message to process P
 - **receive**(Q , message) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating processes
 - Between each pair there exists exactly one link
 - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox

Indirect Communication

- Properties of communication link
 - Link established only if processes share a common mailbox
 - A link may be associated with many processes
 - Each pair of processes may share several communication links
 - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
 - create a new mailbox
 - send and receive messages through mailbox
 - destroy a mailbox
- Primitives are defined as:
send(A, message) – send a message to mailbox A
receive(A, message) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
 - P_1 , P_2 , and P_3 share mailbox A
 - P_1 , sends; P_2 and P_3 receive
 - Who gets the message?
- Solutions
 - Allow a link to be associated with at most two processes
 - Allow only one process at a time to execute a receive operation
 - Allow the system to select arbitrarily the receiver.
Sender is notified who the receiver was.

Synchronization & Asynchronous

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** has the sender block until the message is received
 - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking** send has the sender send the message and continue
 - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link;
implemented in one of three ways
 1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)
 2. Bounded capacity – finite length of n messages
Sender must wait if link full
 3. Unbounded capacity – infinite length
Sender never waits

Examples of IPC Systems - POSIX

- **POSIX Shared Memory**

- Process first creates shared memory segment

```
segment id = shmget(IPC_PRIVATE, size, S  
IRUSR | S_IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared  
memory");
```

- When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

Communications in Client-Server Systems

- Sockets
- Remote Procedure Calls
- Remote Method Invocation (Java)

Lightweight Remote Procedure Call

BRIAN N. BERSHAD, THOMAS E. ANDERSON, EDWARD D. LAZOWSKA,
and HENRY M. LEVY

University of Washington

LRPC

Lots of Unix IPC mechanisms

- Pipes
- Signals
- Unix-domain sockets
- POSIX semaphores
- FIFOs (named pipes)
- Shared memory segments
- System V semaphore sets
- POSIX message queues
- System V message queues
- etc.

IPC is usually heavyweight

- IPC mechanisms in conventional systems tend to combine:
 - Notification: (telling the destination process that something has happened)
 - Scheduling: (changing the current runnable status of the destination, or source)
 - Data transfer: (actually conveying a message payload)
- Unix doesn't have a *lightweight* IPC mechanism

IPC in Unix is usually polled

- Blocking read()/recv() or select()/poll()
- Signals are the nearest thing to upcalls, but...
 - Dedicated (small) stack
 - Limited number of syscalls available (e.g. semaphores)
 - Calling out with longjmp() problematic, to say the least
- Unix lacks a good upcall / event delivery mechanism

The Problem

- How to perform has cross-domain invocations?
- Does the calling domain/process block?
- Is the scheduler involved?
- Is more than one thread involved?
- What happens across physical processors?

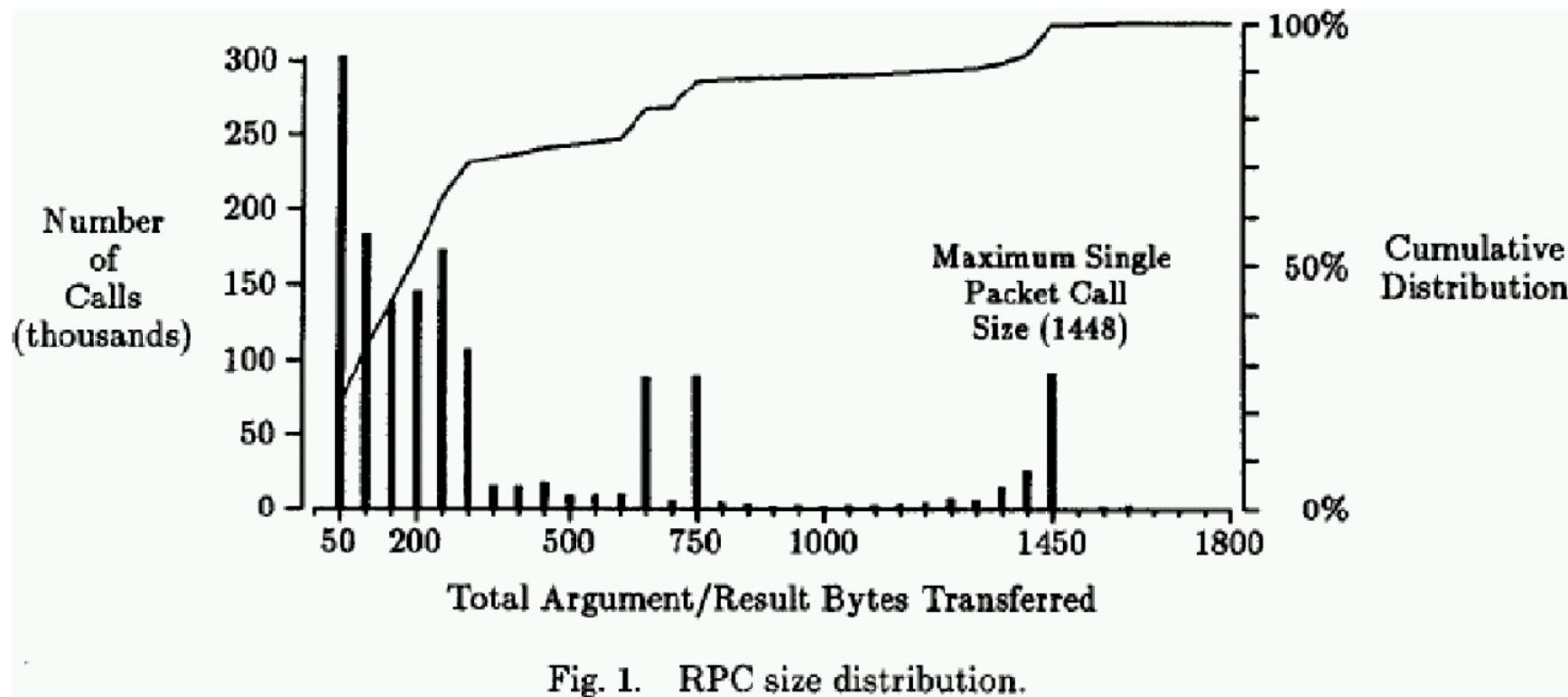
Lightweight RPC (LRPC): Basic concepts

- Simple control transfer: client's thread executes in server's domain
- Simple data transfer: shared argument stack, plus registers
- Simple stubs: i.e. highly optimized marshalling
- Design for concurrency: Avoids shared data structures

High overhead of previous efforts

1. Stubs copy lots of data (not an issue for the network)
2. Message buffers usually copied through the kernel (4 copies!)
3. Access validation
4. Message transfer (queueing/dequeuing of messages)
5. Scheduling: programmer sees thread crossing domains,
system actually rendezvous's two threads in different
domains
6. Context switch (x 2)
7. Dispatch: find a receiver thread to interpret message, and
either dispatch another thread, or leave another one waiting
for more messages

Most messages are short



LRPC Binding: connection setup phase

- Procedure Descriptors (PDs) registered with kernel for each procedure in the called interface
- For each PD, argument stacks (A-stacks) are preallocated and mapped read/write in both domains
- Kernel preallocates linkage records for return from A-stacks
- Returns A-stack list to client as (unforgeable) Binding Object

Calling sequence (all on client thread)

1. Verify Binding Object, find correct PD
2. Verify A-Stack, find corresponding linkage
3. Ensure no other thread using that A-stack/linkage pair
4. Put caller's return addr and stack pointer in linkage
5. Push linkage on to thread control block's stack (for nested calls)
6. Find an execution stack (E-stack) in server's domain
7. Update thread's SP to run off E-stack
8. Perform address space switch to server domain
9. Upcall server's stub at address given in PD

LRPC discussion

- Main kernel housekeeping task is allocating A-stacks and E-stacks
- Shared A-stacks reduce copying of data while still safe
- Stubs incorporated other optimizations (see paper)
- Address space switch is most of the overhead (no TLB tags)
- For multiprocessors:
 - Check for processor idling on server domain
 - If so, swap calling and idling threads
 - (note: thread migration was very cheap on the Firefly!)
 - Same trick applies on return path

Exception

Yubin Xia
IPADS, SJTU

ACKs: Some slides are adapted from the textbook's original slides and
Frans's OS course notes

Review: IPC

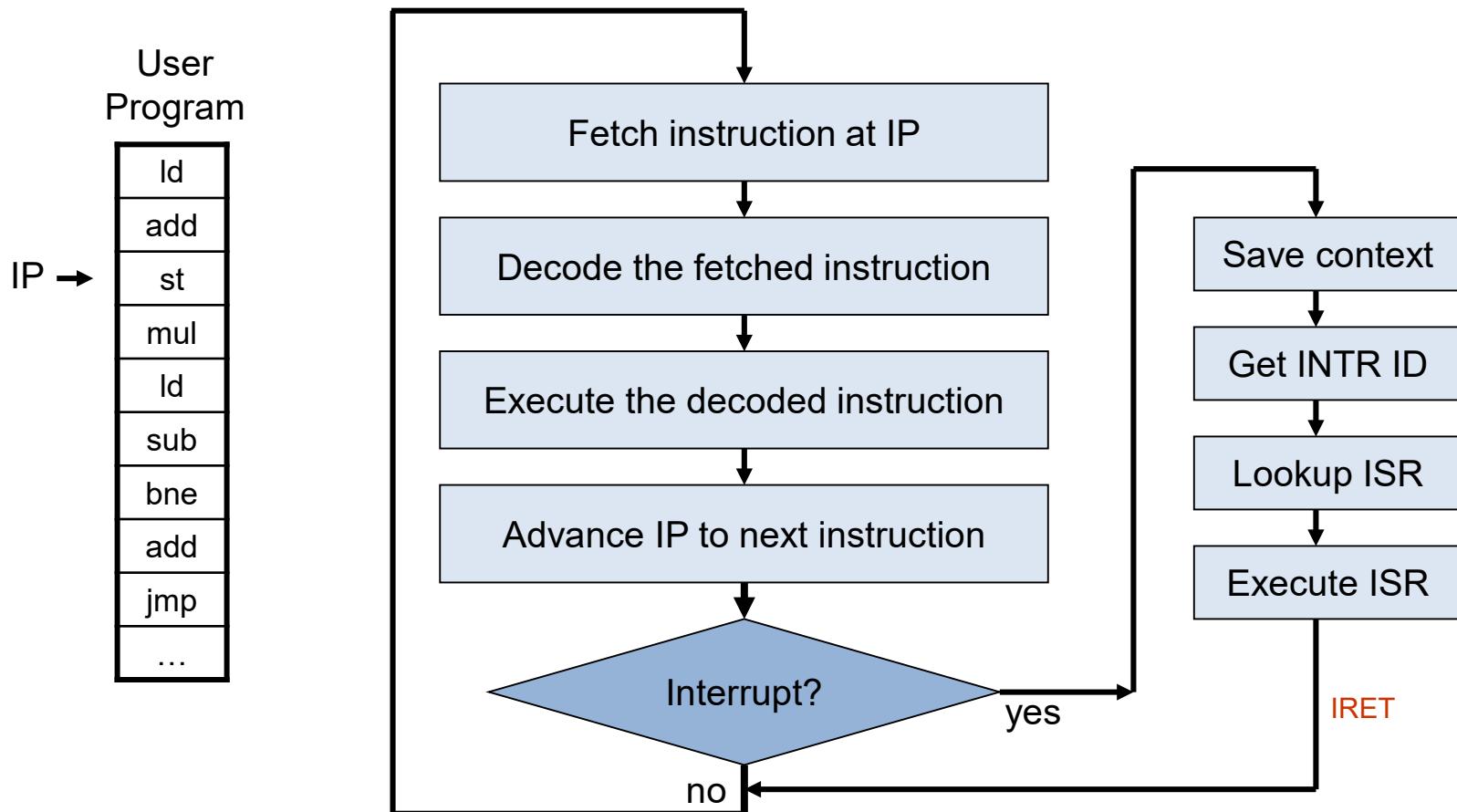
- Cooperating processes need **inter-process communication (IPC)**
- Two models of IPC
 - Shared memory
 - Message passing
- In POSIX
 - Pipe, msg passing, signal, shared memory, semaphore, etc.
- LRPC: Lightweight RPC

Concepts

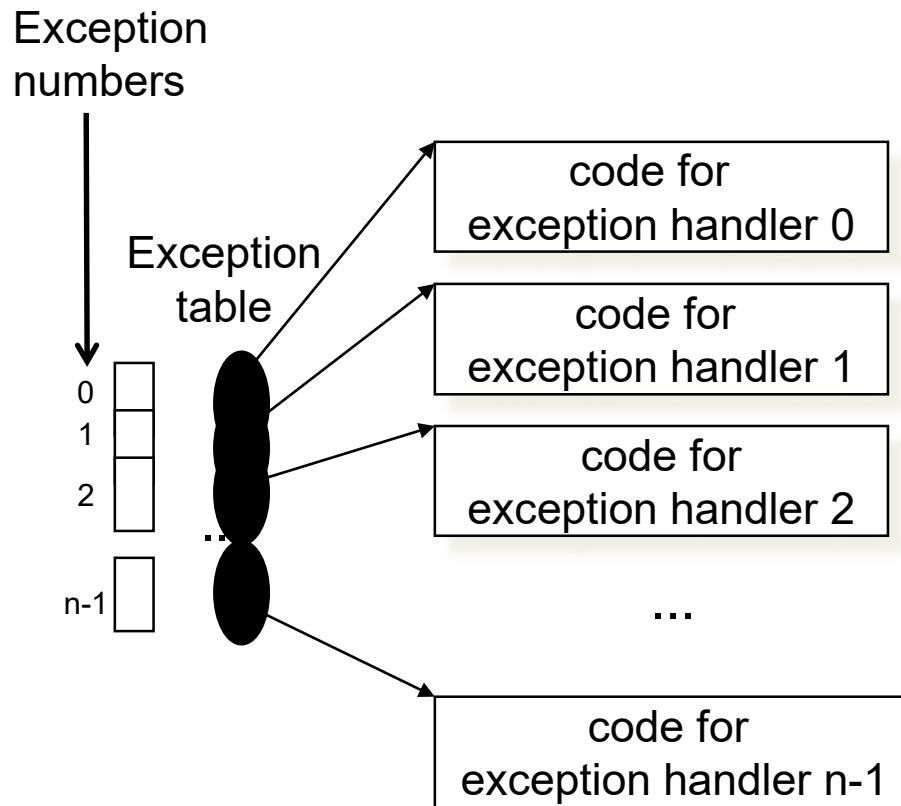
- Exception
 - refers to an illegal program action
- Interrupt
 - refers to a signal generated by a hardware device
- System call
 - a user program can ask for an operating system service

EXCEPTION

CPU's 'fetch-execute' cycle

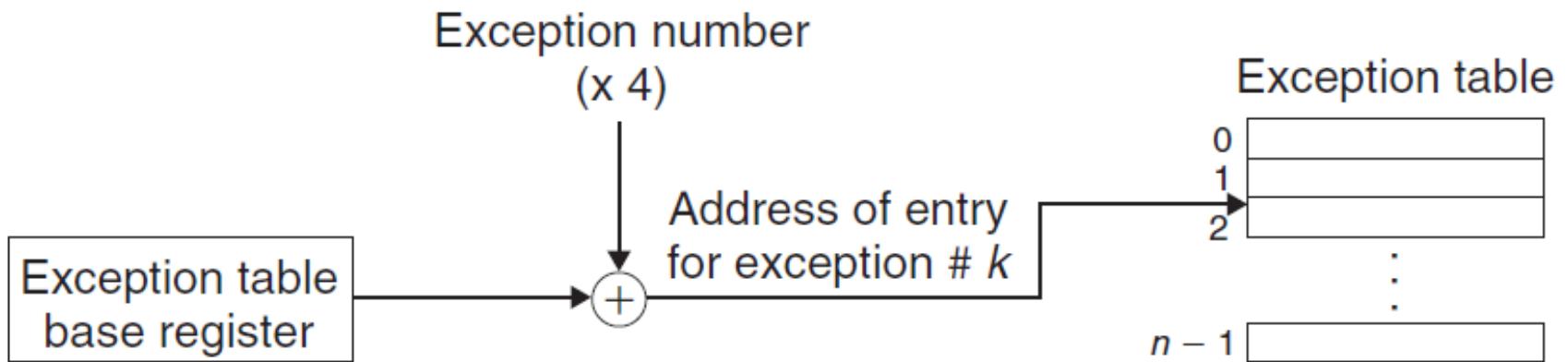


Exception Table



1. Each type of event has a unique exception number k
2. Exception table entry k points to a function (exception handler).
3. Handler k is called each time exception k occurs.

Exception Table



Some Exception on Intel CPU

Exception number	Description	Exception class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32–127	OS-defined exceptions	Interrupt or trap
128 (0x80)	System call	Trap
129–255	OS-defined exceptions	Interrupt or trap

IDT (or Trap Vector in xv6)

- IDT: interrupt descriptor table
 - Another name of exception table
 - Has 256 entries
 - Each giving the `%cs` and `%eip` to be used when handling the corresponding interrupt
- System call
 - A program invokes *int n*
 - *int* is for interrupt (not integer)
 - *n* specifies the index into the IDT

Exception Handler

- The processor pushes a return address on the stack, the return address is:
 - either the current instruction (e.g., page fault)
 - or the next instruction (e.g., hardware interrupt)
- The processor also pushes some additional processor state onto the stack
 - Will be necessary to restart the interrupted program when the handler returns
 - e.g. the current condition codes

Exception Handler

- All items are pushed onto the **kernel stack**
 - Rather than onto the user's stack
 - If control is being transferred from a user program to the kernel
- Exception handlers run in **kernel mode**
 - Means they have complete access to all system resources

Kernel Stack

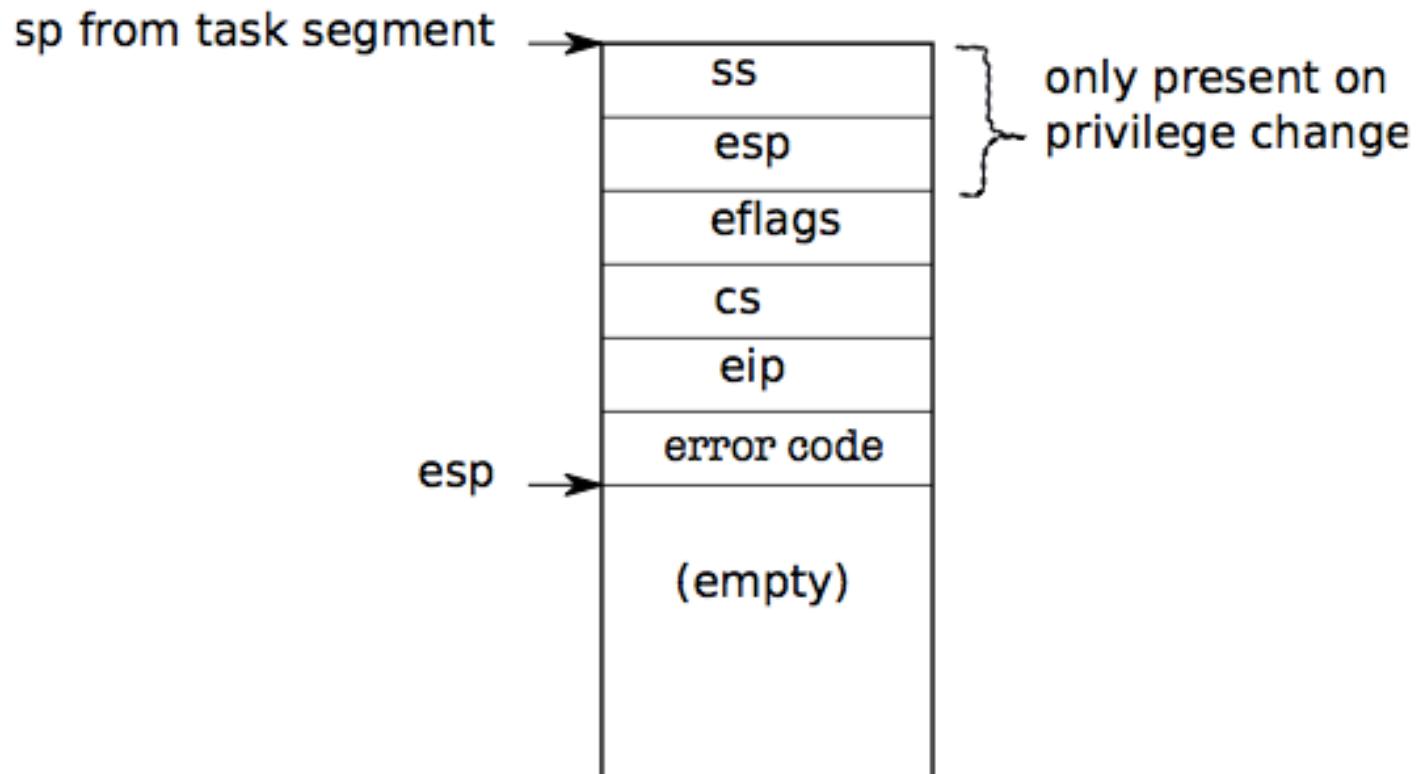
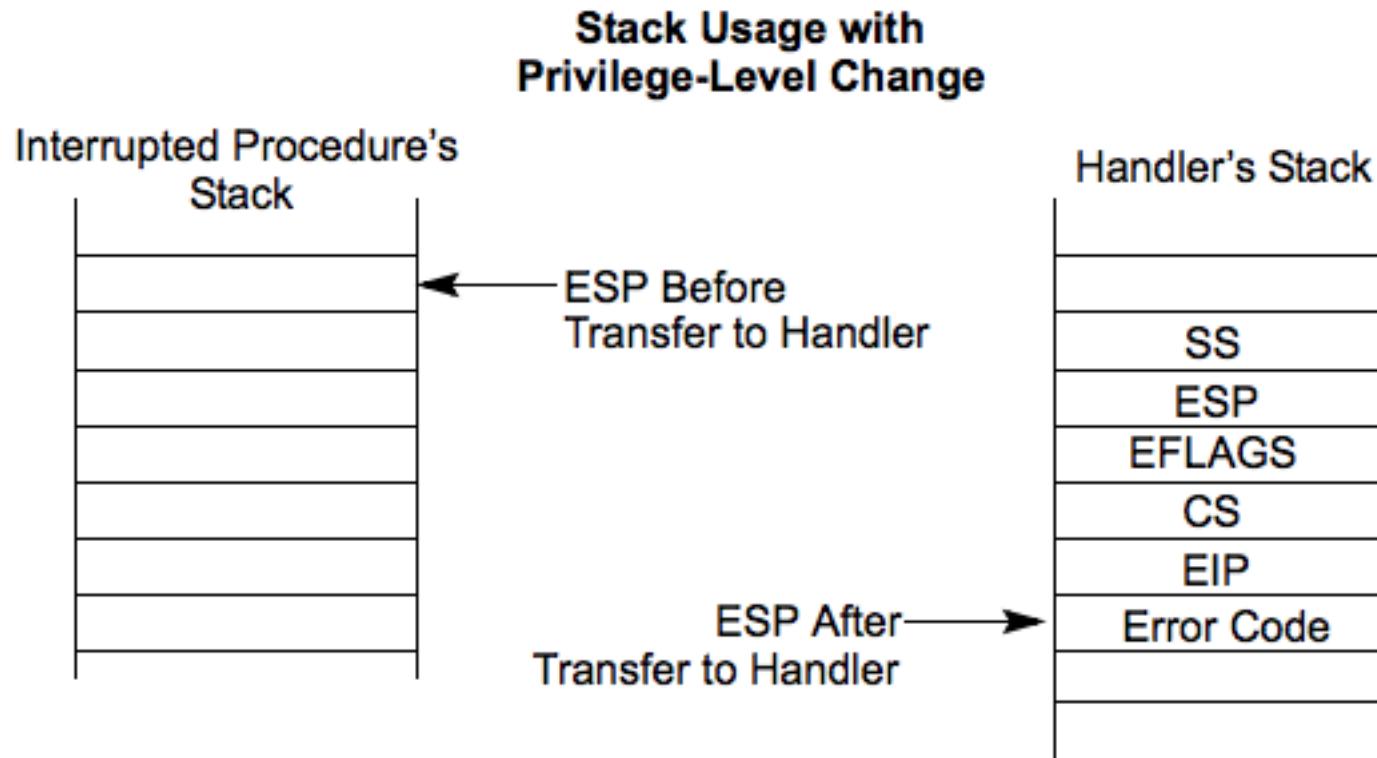


Figure 3-1. Kernel stack after an int instruction.

Stack Change

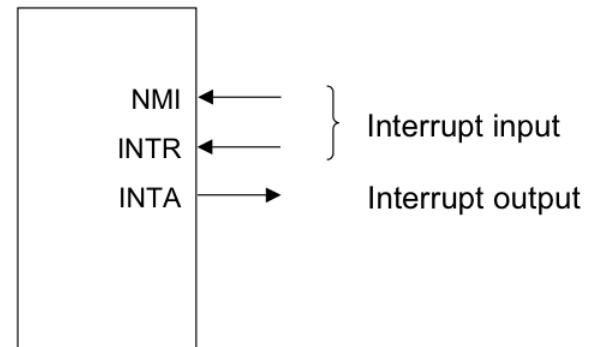


Question: why not use user's stack?

Sources of Events cause User->Kernel

1. Device interrupt: external

- *Nonmaskable interrupt (NMI)* input pin
- *Interrupt (INTR)* input pin



2. Software interrupt: execution of the Interrupt instruction

- E.g., *INT*

3. Program faults: If some error condition occur by the execution of an instruction.

- E.g., *divide-by-zero interrupt*

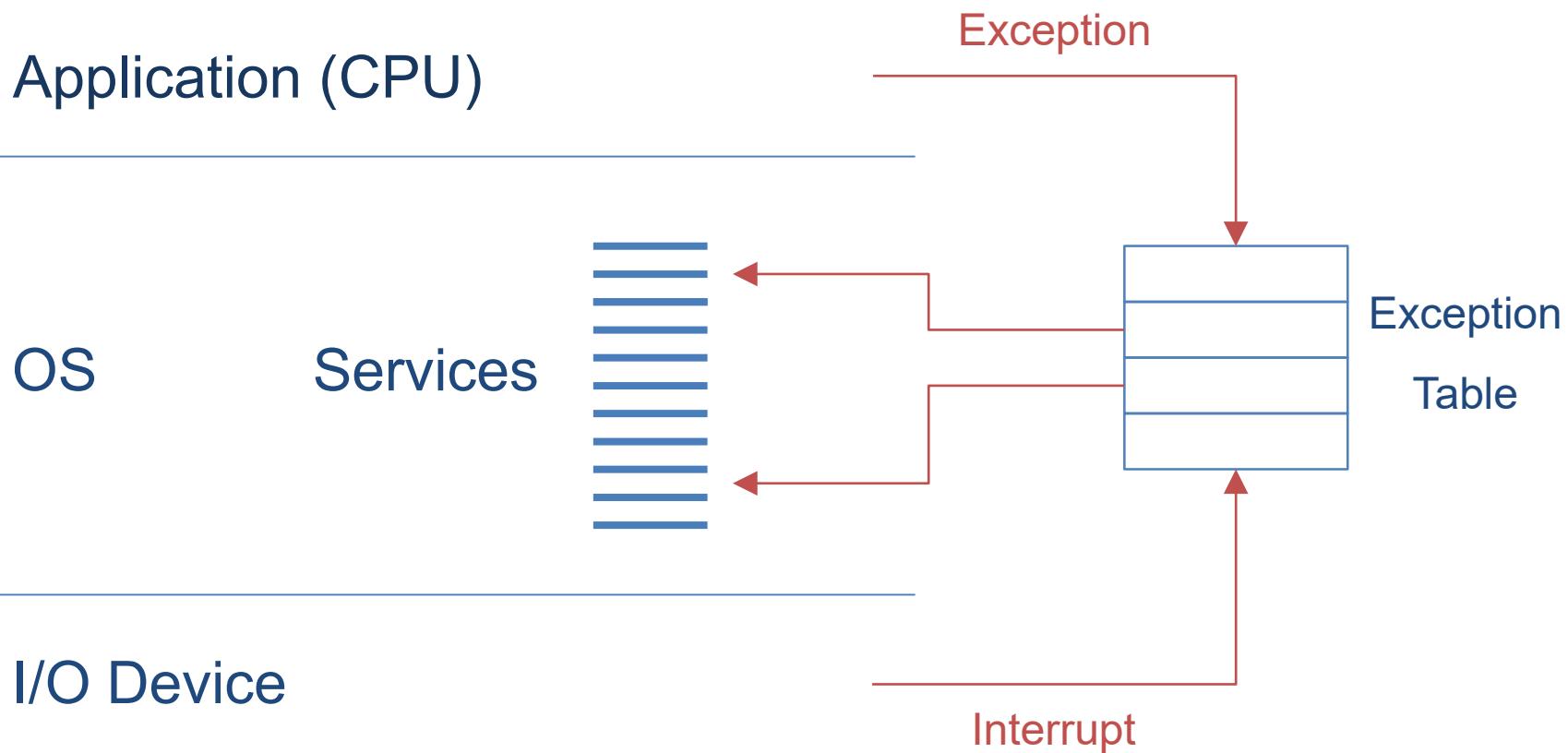
What Has to Happen?

- Must save the processor's registers for future transparent resume
- Must be set up for execution in the kernel
- Must chose a place for the kernel to start executing
- Must be able to retrieve information about the event, e.g., system call arguments
- Must all be done securely
- Must maintain isolation of user processes and the kernel

Return From Interrupt Handler in Kernel

- The '**iret**' instruction
 - Restore the process's context
 - Switch from kernel mode to user mode
 - Continue the execution of user's process

Simplified: Call OS Services from Apps and Devices



xv6: Initialize the Exception Table

```
17 void
18 tvinit(void)
19 {
20     int i;
21
22     for(i = 0; i < 256; i++)
23         SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
24     SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
25
26     initlock(&tickslock, "time");
27 }
```

```
213 #define SETGATE(gate, istrap, sel, off, d) \
214 { \
215     (gate).off_15_0 = (uint)(off) & 0xffff; \
216     (gate).cs = (sel); \
217     (gate).args = 0; \
218     (gate).rsv1 = 0; \
219     (gate).type = (istrap) ? STS_TG32 : STS_IG32; \
220     (gate).s = 0; \
221     (gate).dpl = (d); \
222     (gate).p = 1; \
223     (gate).off_31_16 = (uint)(off) >> 16; \
224 }
```

```
3 .globl alltraps
4 .globl vector0
5 vector0:
6     pushl $0
7     pushl $0
8     jmp alltraps
9 .globl vector1
10 vector1:
11    pushl $0
12    pushl $1
13    jmp alltraps
14 .globl vector2
15 vector2:
16    pushl $0
17    pushl $2
18    jmp alltraps
19 .globl vector3
20 vector3:
```

xv6: Trap

```
4 .globl alltraps
5 alltraps:
6     # Build trap frame.
7     pushl %ds
8     pushl %es
9     pushl %fs
10    pushl %gs
11    pushal
12
13    # Set up data and per-cpu segments.
14    movw $(SEG_KDATA<<3), %ax
15    movw %ax, %ds
16    movw %ax, %es
17    movw $(SEG_KCPU<<3), %ax
18    movw %ax, %fs
19    movw %ax, %gs
20
21    # Call trap(tf), where tf=%esp
22    pushl %esp
23    call trap           ←
24    addl $4, %esp
25
26    # Return falls through to trapret...
27 .globl trapret
28 trapret:
29     popal
30     popl %gs
31     popl %fs
32     popl %es
33     popl %ds
34     addl $0x8, %esp  # trapno and errcode
35     iret
```

```
36 void
37 trap(struct trapframe *tf)
38 {
39     if(tf->trapno == T_SYSCALL){
40         if(proc->killed)
41             exit();
42         proc->tf = tf;
43         syscall();
44         if(proc->killed)
45             exit();
46         return;
47     }
48
49     switch(tf->trapno){
50     case T_IRQ0 + IRQ_TIMER:
51         if(cpu->id == 0){
52             acquire(&tickslock);
53             ticks++;
54             wakeup(&ticks);
55             release(&tickslock);
56         }
57         lapiceoi();
58         break;
```

INTERRUPT

Varying Terminology for Intel

- **Interrupts** (asynchronous, device generated)
 - Maskable: device-generated, associated with IRQs (interrupt request lines); may be temporarily disabled (still pending)
 - Nonmaskable: some critical hardware failures
- **Exceptions** (synchronous, from software)
 - Processor-detected
 - **Faults** – correctable (restartable); e.g. page fault
 - **Traps** – no reexecution needed; e.g. breakpoint
 - **Aborts** – severe error; process usually terminated (by signal)
 - Programmed exceptions (**software interrupts**)
 - int (system call), int3 (breakpoint)
 - into (overflow), bounds (address check)

Terms

- Vector, Interrupt vector, Trap number
- IRQ: Interrupt Request
 - Note: Soft IRQ is a mechanism to implement *bottom half*, it has nothing to do with IRQ!
- Interrupt, trap, fault, exception
- Software interrupt / system call
- IDT: Interrupt Descriptor Table
- ISP: Interrupt Service Procedure

Intel-Reserved ID-Numbers

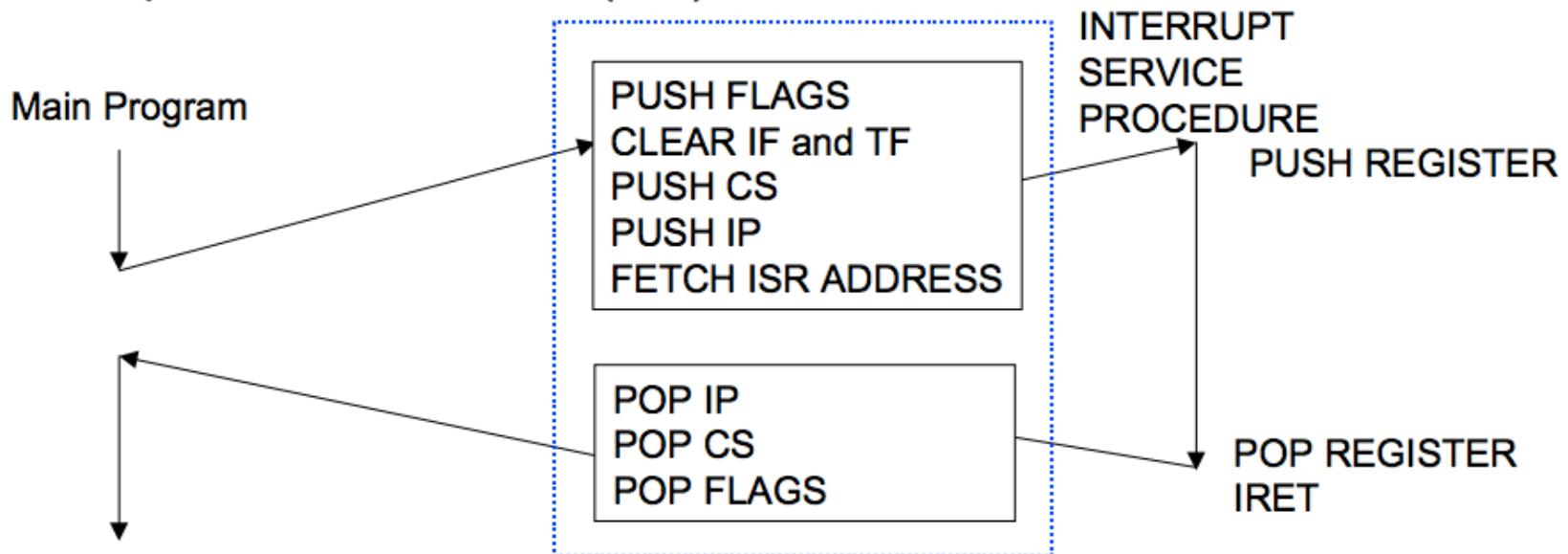
- Of the 256 possible interrupt ID numbers, Intel reserves the first 32 for 'exceptions'
- OS's such as Linux are free to use the remaining 224 available interrupt ID numbers for their own purposes (e.g., for service-requests from external devices, or for other purposes such as system-calls)
- Examples:
 - 0: divide-overflow fault
 - 6: Undefined Opcode
 - 7: Coprocessor Not Available
 - 11: Segment-Not-Present fault
 - 12: Stack fault
 - 13: General Protection Exception
 - 14: Page-Fault Exception

Interrupts

- Interrupts are similar to system calls, except devices generate them at any time
- There is hardware on the motherboard to signal the CPU when a device needs attention (e.g., the user has typed a character on the keyboard)
- We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt

Interrupt Response Sequence

1. PUSH Flag Register
2. Clear IF and TF
3. PUSH CS
4. PUSH IP
5. Fetch Interrupt vector contents and place into both IP and CS to start the Interrupt Service Procedure (ISP)



Early Boards

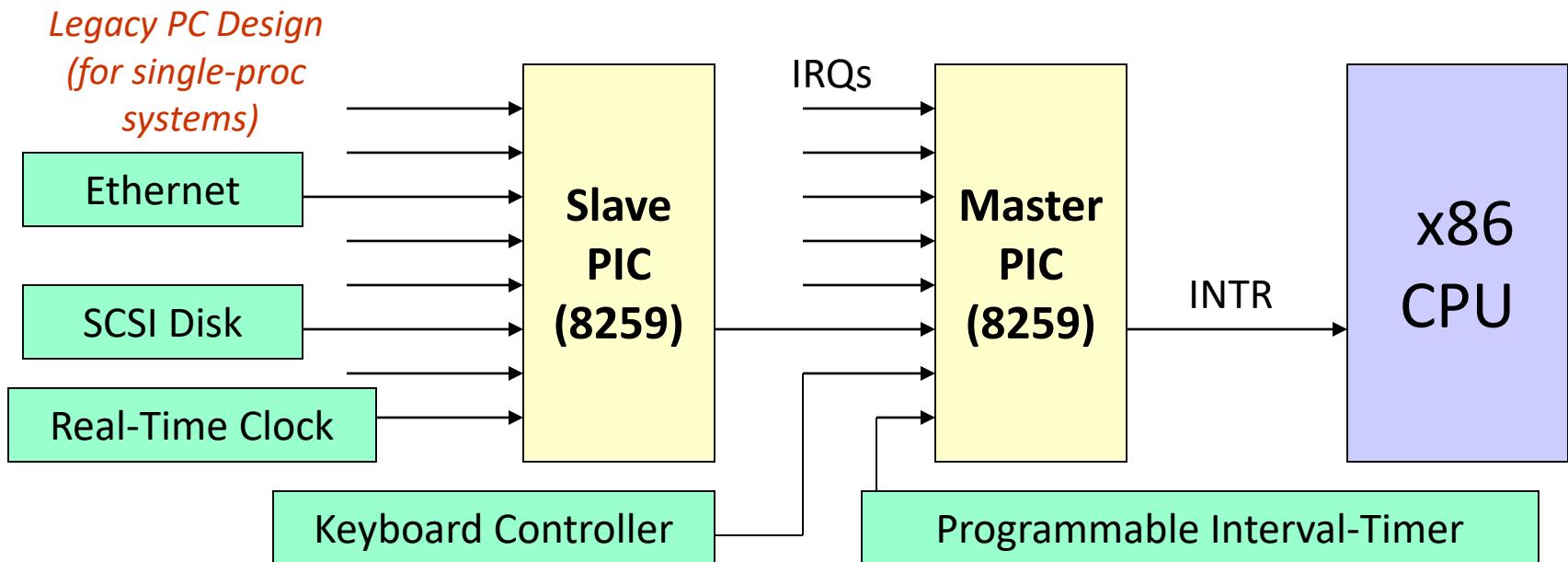
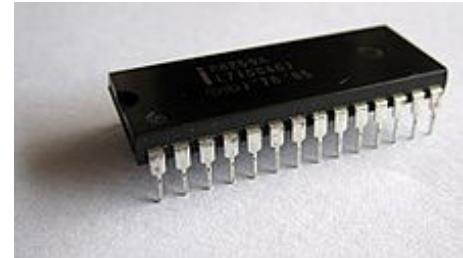
- PIC (Programmable Interrupt Controller)
 - picirq.c
 - 8259A chip
 - Each PIC handles 8 interrupts, and multiplex them on the interrupt pin of the processor
 - PIC can be cascaded
 - Master: 0 to 7
 - Slave: 8 to 15
 - Read timer.c for timer init

8259A Programmable Interrupt Controller (PIC)

- 8259A is a 28-pin integrated circuit which was designed specifically for the 8088/8086 microprocessors

11	D0	18	IR0
10	D1	19	IR1
9	D2	20	IR2
8	D3	21	IR3
7	D4	22	IR4
6	D5	23	IR5
5	D6	24	IR6
4	D7	25	IR7
27	A0		
1	CS		
3	RD		
2	WR		
16	SP/EN	12	CAS0
17	INT	13	CAS1
26	INTA	15	CAS2

Interrupt Hardware

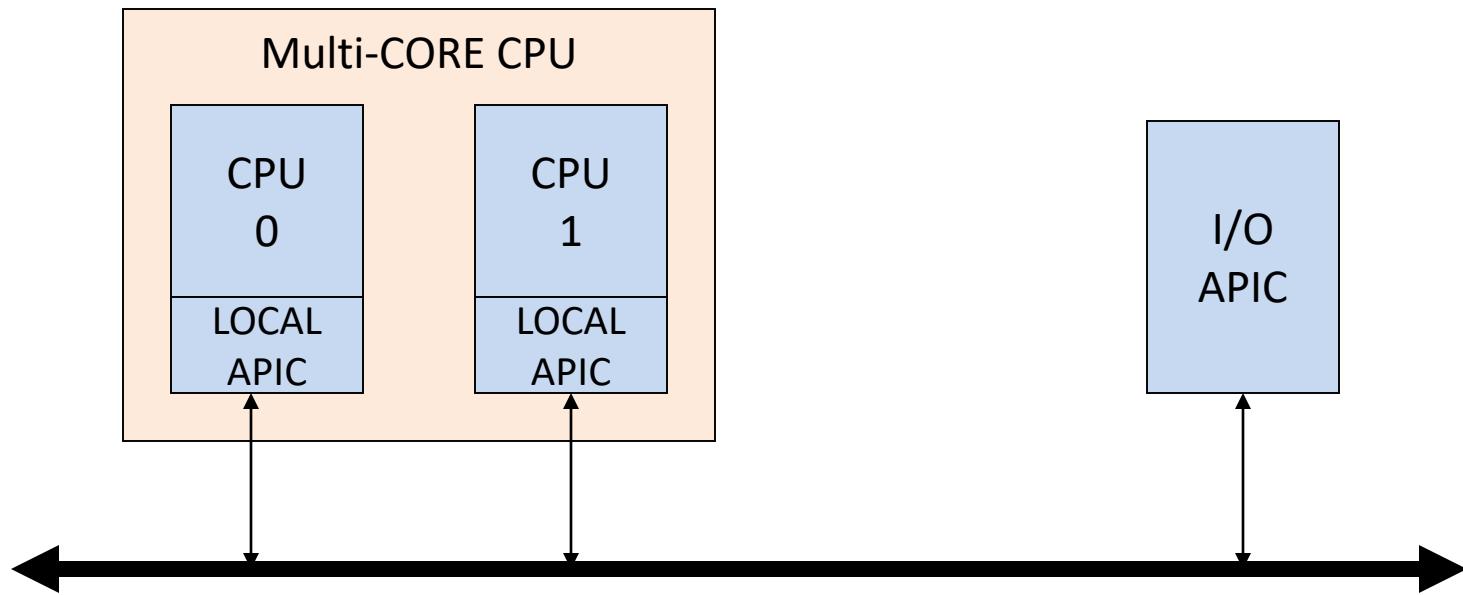


- I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)
- IRQs are mapped by special hardware to *interrupt vectors*, and passed to the CPU
- This hardware is called a *Programmable Interrupt Controller* (PIC)

APIC, IO-APIC, LAPIC

- **Advanced PIC (APIC) for SMP systems**
 - Used in all modern systems
 - Interrupts “routed” to CPU over system bus
 - IPI: inter-processor interrupt
- **Local APIC (LAPIC) versus “frontend” IO-APIC**
 - Devices connect to front-end IO-APIC
 - IO-APIC communicates (over bus) with Local APIC
- **Interrupt routing** (like on a network)
 - Allows broadcast or selective routing of interrupts
 - Ability to distribute interrupt handling load
 - Routes to lowest priority process
 - Special register: Task Priority Register (TPR), i.e., locking
 - Arbitrates (round-robin) if equal priority

Multiple Processors



Advanced Programmable Interrupt Controller is needed to perform 'routing' of I/O requests from peripherals to CPUs

(The legacy PICs are masked when the APICs are enabled)

Multi-processor PC Boards

- Two parts in XV6
 - IO APIC, for the I/O system
 - ioapic.c
 - Local APIC, for each processor
 - lapic.c
- XV6 is designed for multi-processor
 - Each processor must be programmed to receive interrupts

Assigning IRQs to Devices

- **IRQ assignment is hardware-dependent**
 - Sometimes it's hardwired, sometimes it's set physically, sometimes it's programmable
 - PCI bus usually assigns IRQs at boot
- **Some IRQs are fixed by the architecture**
 - IRQ0: Interval timer
 - IRQ2: Cascade pin for 8259A
- **Linux device drivers request IRQs when the device is opened**
 - Note: especially useful for dynamically-loaded drivers, such as for USB or PCMCIA devices
 - Two devices that aren't used at the same time can share an IRQ, even if the hardware doesn't support simultaneous sharing

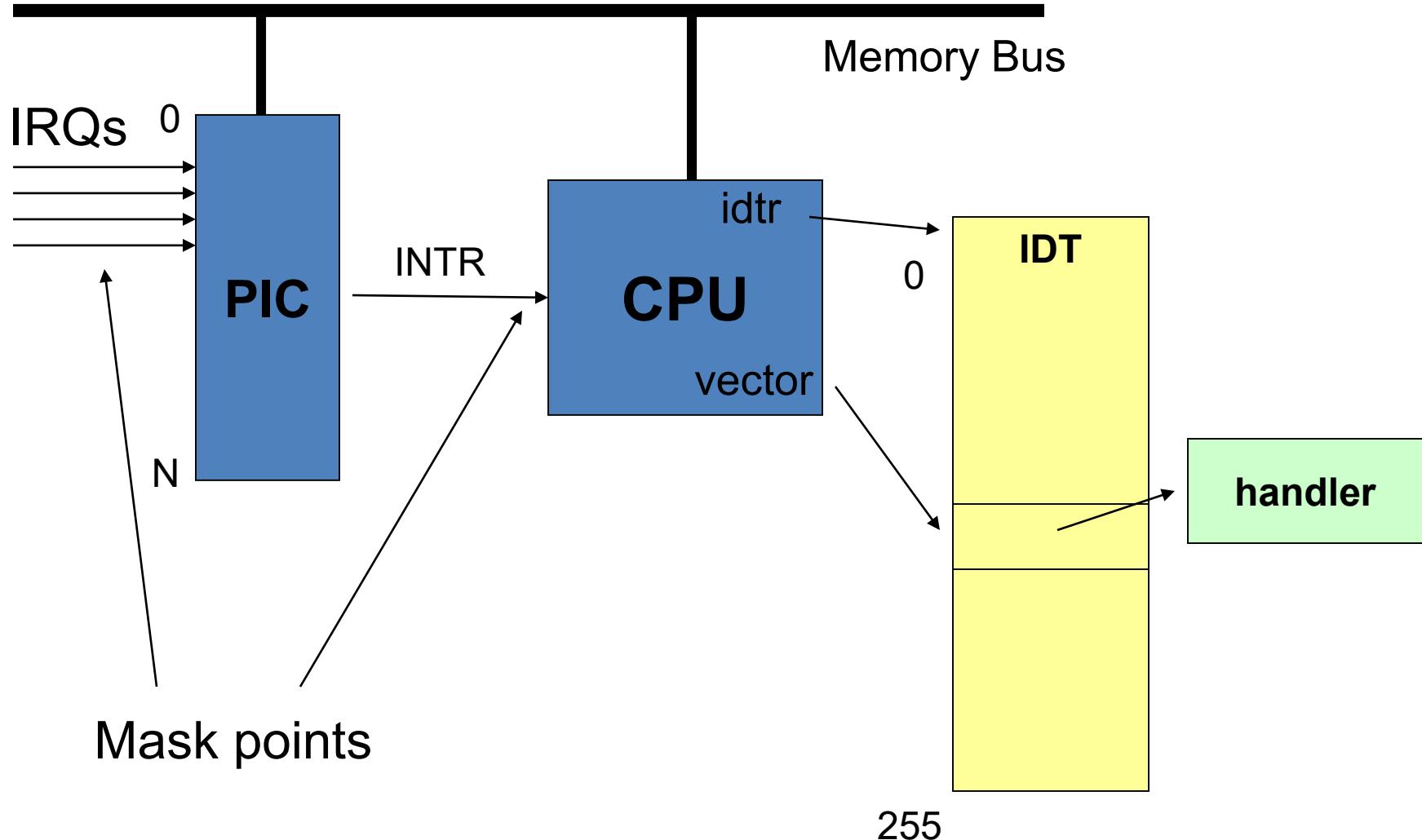
Some Details on IRQ

- **Some hardware IRQs are pre-determined**
 - E.g., the system timer (IRQ0), Keyboard controller (IRQ1), floppy controller (IRQ6), real-time clock (IRQ8) and Math Co-processor (IRQ13)
- **Most others are 'user' determined**
 - via hardware (with jumpers)
 - via software (such as installable drivers) with firmware (PNP)
- **IRQs that are usually available for add-on devices:**
 - Modem (IRQ5), printer (IRQ7), Sound Card (IRQ9/IRQ10)
 - Video Card (IRQ11) and PS/2 mouse (IRQ12)
 - IRQ3 and IRQ4 are usually reserved for serial ports
 - IRQ14 and IRQ15 are used for the IDE (primary and secondary)

Assigning Vectors to IRQs

- Vector: index (0-255) into interrupt descriptor table
- Vectors are usually **IRQ# + 32**
 - Below 32 reserved for non-maskable intr & exceptions
 - Maskable interrupts can be assigned as needed
 - Vector 128 used for syscall
 - Vectors 251-255 used for IPI

Putting It All Together



```
10 #define IO_TIMER1          0x040           // 8253 Timer #1
11
12 // Frequency of all three count-down timers;
13 // (TIMER_FREQ/freq) is the appropriate count
14 // to generate a frequency of freq Hz.
15
16 #define TIMER_FREQ         1193182
17 #define TIMER_DIV(x)       ((TIMER_FREQ+(x)/2)/(x))
18
19 #define TIMER_MODE         (IO_TIMER1 + 3) // timer mode port
20 #define TIMER_SEL0          0x00           // select counter 0
21 #define TIMER_RATEGEN       0x04           // mode 2, rate generator
22 #define TIMER_16BIT         0x30           // r/w counter 16 bits, LSB first
23
24 void
25 timerinit(void)
26 {
27     // Interrupt 100 times/sec.
28     outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
29     outb(IO_TIMER1, TIMER_DIV(100) % 256);
30     outb(IO_TIMER1, TIMER_DIV(100) / 256);
31     picenable(IRQ_TIMER);
32 }
```

Timer Interrupt

```
49 switch(tf->trapno){  
50 case T_IRQ0 + IRQ_TIMER:  
51     if(cpu->id == 0){  
52         acquire(&tickslock);  
53         ticks++;  
54         wakeup(&ticks);  
55         release(&tickslock);  
56     }  
57     lapiceoi();  
58     break;
```

```
58 int  
59 sys_sleep(void)  
60 {  
61     int n;  
62     uint ticks0;  
63  
64     if(argint(0, &n) < 0)  
65         return -1;  
66     acquire(&tickslock);  
67     ticks0 = ticks;  
68     while(ticks - ticks0 < n){  
69         if(proc->killed){  
70             release(&tickslock);  
71             return -1;  
72         }  
73         sleep(&ticks, &tickslock);  
74     }  
75     release(&tickslock);  
76     return 0;  
77 }
```

Sleep & Wakeup

```
378 // Wake up all processes sleeping on chan.  
379 // The ptable lock must be held.  
380 static void  
381 wakeup1(void *chan)  
382 {  
383     struct proc *p;  
384  
385     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
386         if(p->state == SLEEPING && p->chan == chan)  
387             p->state = RUNNABLE;  
388 }  
389  
390 // Wake up all processes sleeping on chan.  
391 void  
392 wakeup(void *chan)  
393 {  
394     acquire(&ptable.lock);  
395     wakeup1(chan);  
396     release(&ptable.lock);  
397 }  
340 // Atomically release lock and sleep on chan.  
341 // Reacquires lock when awakened.  
342 void  
343 sleep(void *chan, struct spinlock *lk)  
344 {  
345     if(proc == 0)  
346         panic("sleep");  
347  
348     if(lk == 0)  
349         panic("sleep without lk");  
350  
351     // Must acquire ptable.lock in order to  
352     // change p->state and then call sched.  
353     // Once we hold ptable.lock, we can be  
354     // guaranteed that we won't miss any wakeup  
355     // (wakeup runs with ptable.lock locked),  
356     // so it's okay to release lk.  
357     if(lk != &ptable.lock){ //DOC: sleeplock0  
358         acquire(&ptable.lock); //DOC: sleeplock1  
359         release(lk);  
360     }  
361  
362     // Go to sleep.  
363     proc->chan = chan;  
364     proc->state = SLEEPING;  
365     sched();  
366  
367     // Tidy up.  
368     proc->chan = 0;  
369  
370     // Reacquire original lock.  
371     if(lk != &ptable.lock){ //DOC: sleeplock2  
372         release(&ptable.lock);  
373         acquire(lk);  
374     }  
375 }
```

Interrupt Priority

- When different types of interrupt (i.e., software, NMI, INTR or exceptions) occur at the same time, the one with the highest priority is handled first
- Intel microprocessors use the following order of priority:

Interrupt Type	Priority
divide error interrupt, INT n, INTO NMI INTR TRAP flag (single step)	highest  lowest

EOI: End of Interrupt

- Issued to the PIC chips at the end of an IRQ-based interrupt routine
 - If the IRQ came from the Master PIC, it is sufficient to issue this command only to the Master PIC;
 - If the IRQ came from the Slave PIC, it is necessary to issue the command to both PIC chips

```
#define PIC_EOI          0x20          /* End-of-interrupt

void PIC_sendEOI(unsigned char irq)
{
    if(irq >= 8)
        outb(PIC2_COMMAND, PIC_EOI);

    outb(PIC1_COMMAND, PIC_EOI);
}
```

Nested Interrupts

- What if a second interrupt occurs while an interrupt routine is executing?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximizing Parallelism

- Keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation
 - Another request should be issued ASAP
- Most devices do not interfere with each others' data structures
 - No reason to block out other devices

Handling Nested Interrupts

- As soon as possible, unmask the global interrupt
- As soon as reasonable, re-enable interrupts from that IRQ
- But that is not always a great idea, since it could cause **re-entry** to the same handler
- IRQ-specific mask is not enabled during interrupt-handling

Nested Execution

- Interrupts can be interrupted
 - By different interrupts; handlers need not be reentrant
 - Small portions execute with interrupts disabled
 - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
 - By interrupts (devices needing service)
- Exceptions can nest two levels deep
 - Exceptions indicate coding error
 - Exception code (kernel code) should not have bugs
 - Page fault is possible (trying to touch user data)

Interrupt Masking

- Two different types: global and selective
 - Global — delays all interrupts
 - Selective — mask individual IRQs selectively
 - This is usually what's needed — interference most common from two interrupts of the same type
- NMI (Non-Maskable Interrupt)

Triple Fault

- Things never to do in an OS #1: Swap out the page swapping code (triple-fault here we come) —Kemp
- When a fault occurs, the CPU invokes an exception handler.
- If a fault occurs while trying to invoke the exception handler, that's called a double fault, which the CPU tries to handle with yet another exception handler.
- If that invocation results in a fault too, the system reboots with a triple fault.

/proc/interrupts

```
$ cat /proc/interrupts
          CPU0
 0:    865119901      IO-APIC-edge    timer
 1:        4      IO-APIC-edge    keyboard
 2:        0        XT-PIC       cascade
 8:        1      IO-APIC-edge     rtc
12:       20      IO-APIC-edge  PS/2 Mouse
14:   6532494      IO-APIC-edge     ide0
15:       34      IO-APIC-edge     ide1
16:       0      IO-APIC-level   usb-uhci
19:       0      IO-APIC-level   usb-uhci
23:       0      IO-APIC-level  ehci-hcd
32:       40      IO-APIC-level     ioc0
33:       40      IO-APIC-level     ioc1
48: 273306628      IO-APIC-level     eth0
NMI:       0
ERR:       0
```

- Columns: IRQ, count, interrupt controller, devices

More in /proc/pci:

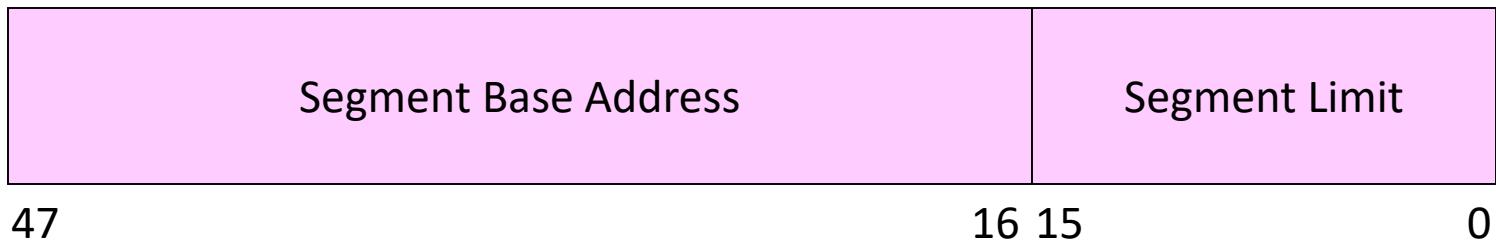
```
$ cat /proc/pci
PCI devices found:
Bus 0, device 0, function 0:
    Host bridge: PCI device 8086:2550 (Intel Corp.) (rev 3).
                Prefetchable 32 bit memory at 0xe8000000 [0xebffff]. .
Bus 0, device 29, function 1:
    USB Controller: Intel Corp. 82801DB USB (Hub #2) (rev 2).
                IRQ 19.
                I/O at 0xd400 [0xd41f].
Bus 0, device 31, function 1:
    IDE interface: Intel Corp. 82801DB ICH4 IDE (rev 2).
                IRQ 16.
                I/O at 0xf000 [0xf00f].
                Non-prefetchable 32 bit memory at 0x80000000 [0x800003ff]. .
Bus 3, device 1, function 0:
    Ethernet controller: Broadcom NetXtreme BCM5703X Gigabit Eth (rev 2).
                IRQ 48.
                Master Capable. Latency=64. Min Gnt=64.
                Non-prefetchable 64 bit memory at 0xf7000000 [0xf700ffff].
```

Three Crucial Data-structures

- The Global Descriptor Table (GDT)
defines the system's memory-segments and their access-privileges, which the CPU has the duty to enforce
- The Interrupt Descriptor Table (IDT)
defines entry-points for the various code-routines that will handle all 'interrupts' and 'exceptions'
- The Task-State Segment (TSS)
holds the values for registers SS and ESP that will get loaded by the CPU upon entering kernel-mode

How does CPU find GDT/IDT?

- Two dedicated registers: GDTR and IDTR
- Both have identical 48-bit formats:



Kernel must setup these registers during system startup (set-and-forget)

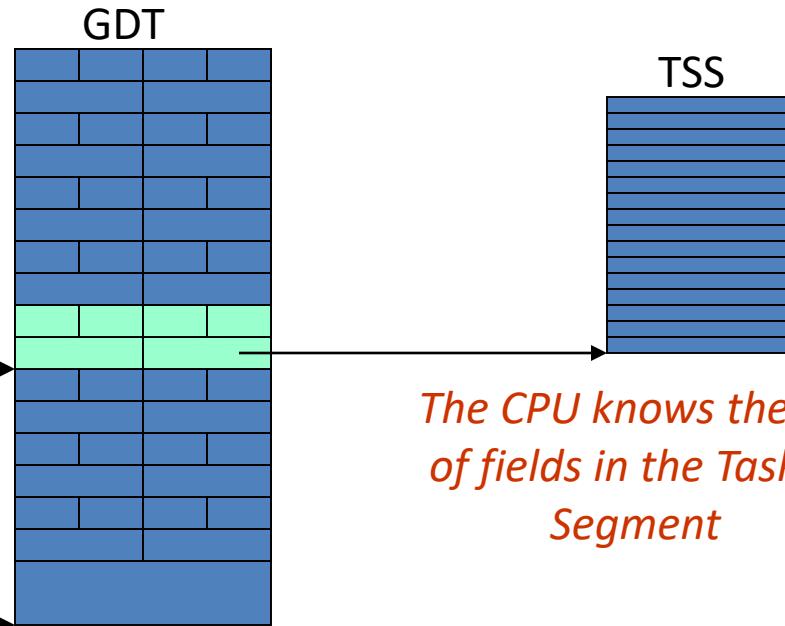
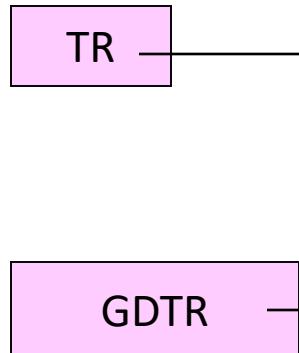
Privileged instructions: LGDT and LIDT used to set these register-values

Unprivileged instructions: SGDT and SIDT used for reading register-values

How does CPU find the TSS?

- Dedicated system segment-register **TR** holds a descriptor's offset into the GDT

The kernel must set up the GDT and TSS structures and must load the GDTR and the TR registers

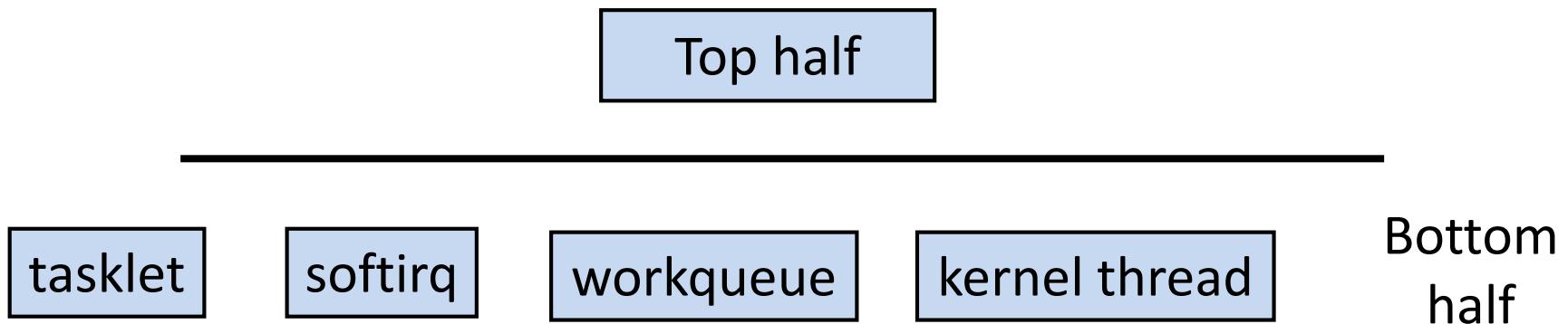


The CPU knows the layout of fields in the Task-State Segment

BOTTOM HALF

Interrupt Handling Philosophy

- Do as little as possible in the interrupt handler
- Defer non-critical actions till later
- Structure: top and bottom halves
 - Top-half: do minimum work and return (ISR)
 - Bottom-half: deferred processing (softirqs, tasklets, workqueues, kernel threads)



Top Half: Do it Now!

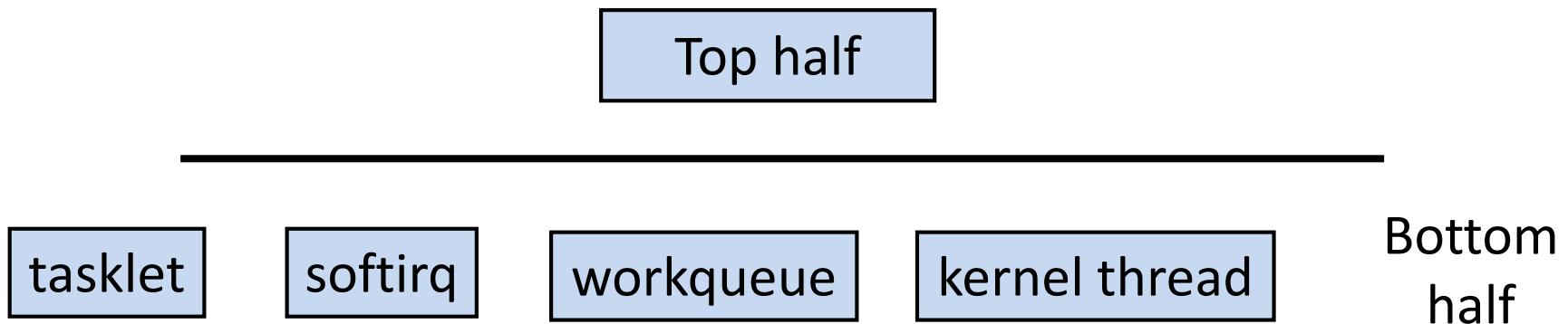
- Technically *is* the interrupt handler
- Perform minimal, common functions: save registers, unmask *other* interrupts. Eventually, undoes that: restores registers, returns to previous context.
 - Often written in assembler
- IRQ is typically masked for duration of top half
- Most important: call proper interrupt handler provided in device drivers (C program)
- Don't want to do too much here
 - IRQs are masked for part of the time
 - Don't want stack to get too big
- Typically queue the request and set a flag for deferred processing in a bottom half

Top Half: Find the Handler

- On modern hardware, multiple I/O devices can share a single IRQ and hence interrupt vector
- First differentiator is the interrupt *vector*
- Multiple *interrupt service routines* (ISR) can be associated with a vector
- Each device's ISR for that IRQ is called
- Device determines whether IRQ is for it

Bottom Half: Do it Later!

- Mechanisms to defer work to later:
 - *softirqs*
 - *tasklets* (built on top of softirqs)
 - *work queues*
 - *kernel threads*
- All can be interrupted



Warning: No Process Context

- Interrupts (as opposed to exceptions) are not associated with particular instructions
- They are also not associated with a given process (user program)
- The currently-running process, at the time of the interrupt, as no relationship whatsoever to that interrupt
- **Interrupt handlers cannot sleep!**

What Can't You Do?

- You cannot sleep
 - or call something that *might* sleep
- You cannot refer to **current**
- You cannot allocate memory with **GPF_KERNEL** (which can sleep), you must use **GPF_ATOMIC** (which can fail)
- You cannot call **schedule()**
- You cannot do a **down()** semaphore call
 - However, you *can* do an **up()**
- You cannot transfer data to/from user space
 - E.g., **copy_to_user()**, **copy_from_user()**

Interrupt Stack

- When an interrupt occurs, what stack is used?
 - Exceptions: The *kernel stack* of the current process, whatever it is, is used (There's always some process running — the “idle” process, if nothing else)
 - Interrupts: hard IRQ stack (1 per processor)
 - SoftIRQs: soft IRQ stack (1 per processor)
- These stacks are configured in the IDT and TSS at boot time by the kernel

Softirqs

- **Statically** allocated: specified at kernel compile time
- Limited number:

<i>Priority</i>	<i>Type</i>
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	Block devices
5	Regular tasklets

When Do Softirqs Run?

- Run at various points by the kernel:
 - After system calls
 - After exceptions
 - After interrupts (top halves/IRQs, including the timer intr)
 - When the scheduler runs ksoftirqd
- Softirq routines can be executed simultaneously on multiple CPUs:
 - Code must be re-entrant
 - Code must do its own locking as needed
- Hardware interrupts always enabled when softirqs are running

Rescheduling Softirqs

- A softirq routine can reschedule itself
- This could starve user-level processes
- Softirq scheduler only runs a limited number of requests at a time
- The rest are executed by a kernel thread, `ksoftirqd`, which competes with user processes for CPU time

Tasklets

- Built on top of softirqs
- Can be created and destroyed dynamically
- Run on the CPU that scheduled it (cache affinity)
- Individual tasklets are locked during execution; no problem about re-entrancy, and no need for locking by the code
- Tasklets can run in parallel on multiple CPUs
 - *Same* tasklet can only run on one CPU
- Were once the preferred mechanism for most deferred activity, now changing

The Trouble with Tasklets

- Hard to get right
- One has to be careful about sleeping
- They run at higher priority than other tasks in the systems
- Can produce uncontrolled latency if coded badly
- Ongoing discussion about eliminating tasklets
- Will likely slowly fade over time

Work Queues

- Always run by kernel threads
 - Are scheduled by the scheduler
- Softirqs and tasklets run in an interrupt context; work queues have a pseudo-process context
 - i.e., have a kernel context but no user context
- Because they have a pseudo-process context, they can sleep
 - Work queues are shared by multiple devices
 - Thus, sleeping will delay other work on the queue
- However, they are kernel-only; there is no user mode associated with it
 - Don't try copying data into/out of user space

Kernel Threads

- Always operate in kernel mode
 - Again, no user context
- 2.6.30 introduced the notion of *threaded interrupt handlers*
 - Imported from the realtime tree
 - `request_threaded_irq()`
 - Now each bottom half has its own context, unlike work queues
 - Idea is to eventually replace tasklets and work queues

Comparing Approaches

	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
Will disable all interrupts?	Briefly	No	No	No	No
Will disable other instances of self?	Yes	Yes	No	No	No
Higher priority than regular scheduled tasks?	Yes	Yes*	Yes*	No	No
Will be run on same processor as ISR?	N/A	Yes	Yes	Yes	Maybe
More than one run can on same CPU?	No	No	No	Yes	Yes
Same one can run on multiple CPUs?	Yes	Yes	No	Yes	Yes
Full context switch?	No	No	No	Yes	Yes
Can sleep? (Has own kernel stack)	No	No	No	Yes	Yes
Can access user space?	No	No	No	No	No

*Within limits, can be run by ksoftirqd

Interrupt

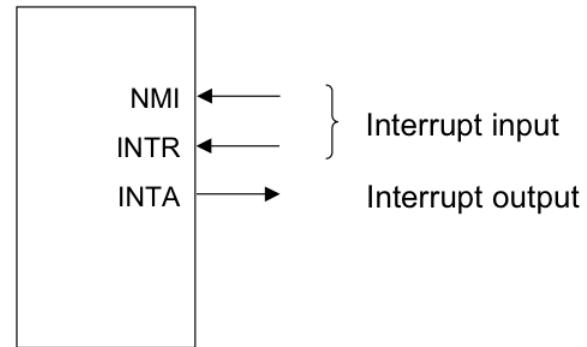
Yubin Xia
IPADS, SJTU

ACKs: Some slides are adapted from the textbook's original slides and
Frans's os course notes

Review: Events causing User->Kernel

1. Device interrupt: external

- *Nonmaskable interrupt (NMI)* input pin
- *Interrupt (INTR)* input pin



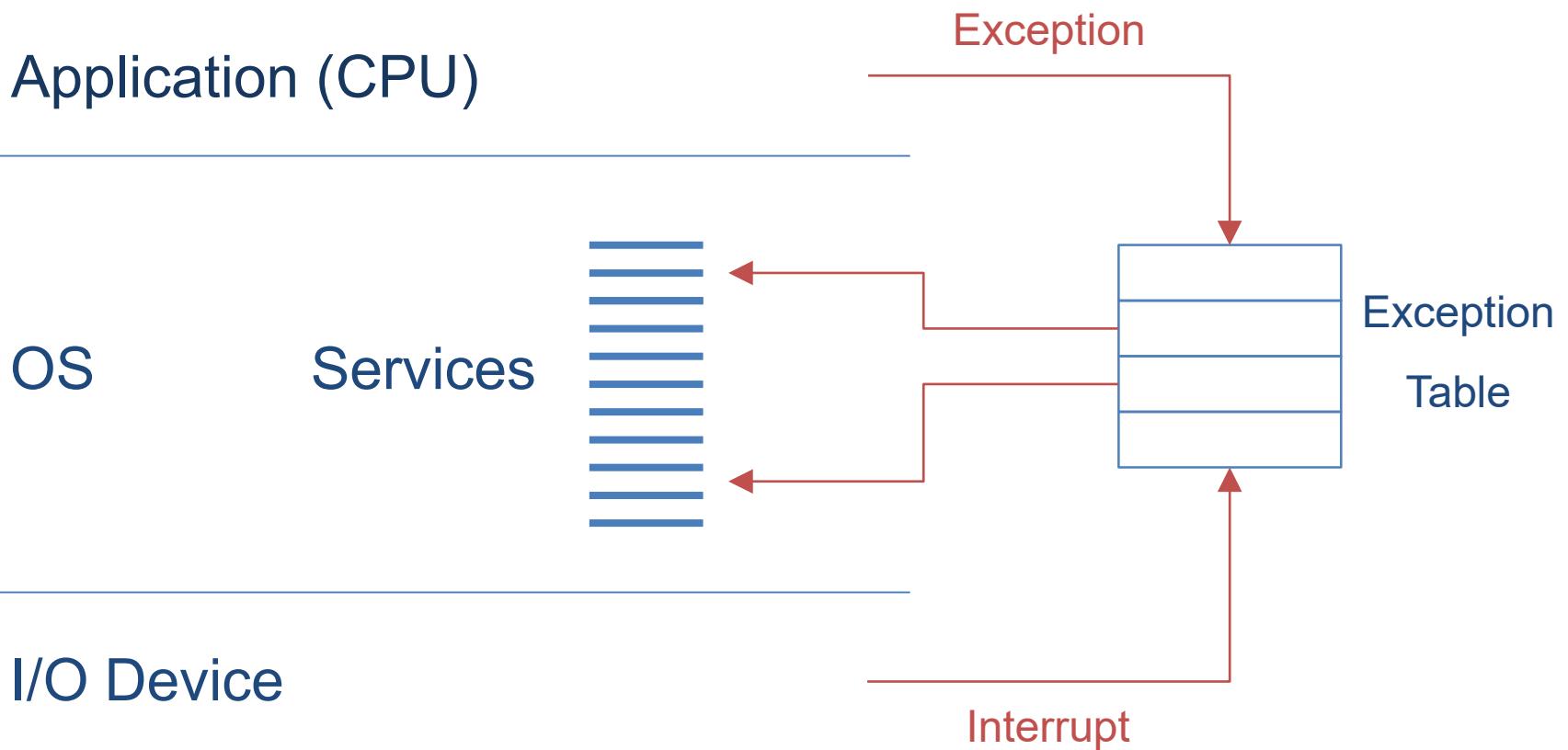
2. Software interrupt: execution of the Interrupt instruction

- e.g., *INT*

3. Program faults: If some error condition occur by the execution of an instruction. E.g.,

- *divide-by-zero interrupt*

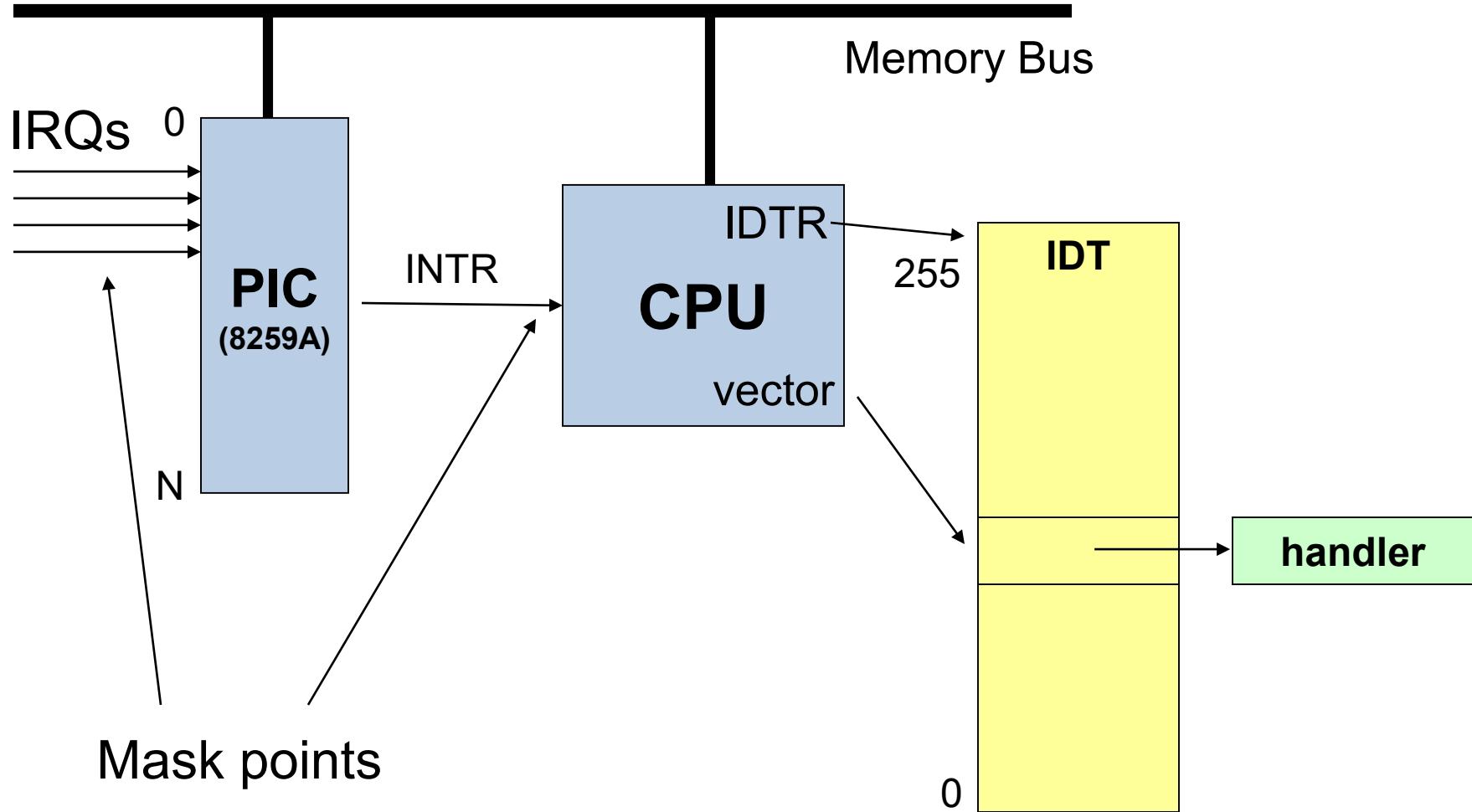
Review: OS as Services



Review: Terminology for Intel

- **Interrupts** (asynchronous, device generated)
 - Maskable: device-generated, associated with IRQs (interrupt request lines); may be temporarily disabled (still pending)
 - Nonmaskable: some critical hardware failures
- **Exceptions** (synchronous, from software)
 - Processor-detected
 - **Faults** – correctable (restartable); e.g. page fault
 - **Traps** – no reexecution needed; e.g. breakpoint
 - **Aborts** – severe error; process usually terminated (by signal)
 - Programmed exceptions (**software interrupts**)
 - int (system call), int3 (breakpoint)
 - into (overflow), bounds (address check)

Putting It All Together



```
10 #define IO_TIMER1          0x040           // 8253 Timer #1
11
12 // Frequency of all three count-down timers;
13 // (TIMER_FREQ/freq) is the appropriate count
14 // to generate a frequency of freq Hz.
15
16 #define TIMER_FREQ         1193182
17 #define TIMER_DIV(x)       ((TIMER_FREQ+(x)/2)/(x))
18
19 #define TIMER_MODE         (IO_TIMER1 + 3) // timer mode port
20 #define TIMER_SEL0          0x00           // select counter 0
21 #define TIMER_RATEGEN        0x04           // mode 2, rate generator
22 #define TIMER_16BIT         0x30           // r/w counter 16 bits, LSB first
23
24 void
25 timerinit(void)
26 {
27     // Interrupt 100 times/sec.
28     outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
29     outb(IO_TIMER1, TIMER_DIV(100) % 256);
30     outb(IO_TIMER1, TIMER_DIV(100) / 256);
31     picenable(IRQ_TIMER);
32 }
```

Timer Interrupt

```
49 switch(tf->trapno){  
50 case T_IRQ0 + IRQ_TIMER:  
51     if(cpu->id == 0){  
52         acquire(&tickslock);  
53         ticks++;  
54         wakeup(&ticks);  
55         release(&tickslock);  
56     }  
57     lapiceoi();  
58     break;
```

```
58 int  
59 sys_sleep(void)  
60 {  
61     int n;  
62     uint ticks0;  
63  
64     if(argint(0, &n) < 0)  
65         return -1;  
66     acquire(&tickslock);  
67     ticks0 = ticks;  
68     while(ticks - ticks0 < n){  
69         if(proc->killed){  
70             release(&tickslock);  
71             return -1;  
72         }  
73         sleep(&ticks, &tickslock);  
74     }  
75     release(&tickslock);  
76     return 0;  
77 }
```

Sleep & Wakeup

```
378 // Wake up all processes sleeping on chan.  
379 // The ptable lock must be held.  
380 static void  
381 wakeup1(void *chan)  
382 {  
383     struct proc *p;  
384  
385     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)  
386         if(p->state == SLEEPING && p->chan == chan)  
387             p->state = RUNNABLE;  
388 }  
389  
390 // Wake up all processes sleeping on chan.  
391 void  
392 wakeup(void *chan)  
393 {  
394     acquire(&ptable.lock);  
395     wakeup1(chan);  
396     release(&ptable.lock);  
397 }  
340 // Atomically release lock and sleep on chan.  
341 // Reacquires lock when awakened.  
342 void  
343 sleep(void *chan, struct spinlock *lk)  
344 {  
345     if(proc == 0)  
346         panic("sleep");  
347  
348     if(lk == 0)  
349         panic("sleep without lk");  
350  
351     // Must acquire ptable.lock in order to  
352     // change p->state and then call sched.  
353     // Once we hold ptable.lock, we can be  
354     // guaranteed that we won't miss any wakeup  
355     // (wakeup runs with ptable.lock locked),  
356     // so it's okay to release lk.  
357     if(lk != &ptable.lock){ //DOC: sleeplock0  
358         acquire(&ptable.lock); //DOC: sleeplock1  
359         release(lk);  
360     }  
361  
362     // Go to sleep.  
363     proc->chan = chan;  
364     proc->state = SLEEPING;  
365     sched();  
366  
367     // Tidy up.  
368     proc->chan = 0;  
369  
370     // Reacquire original lock.  
371     if(lk != &ptable.lock){ //DOC: sleeplock2  
372         release(&ptable.lock);  
373         acquire(lk);  
374     }  
375 }
```

Interrupt Priority

- When different types of interrupt (i.e., software, NMI, INTR or exceptions) occur at the same time, the one with the highest priority is handled first
- Intel microprocessors use the following order of priority:

Interrupt Type	Priority
divide error interrupt, INT n, INTO NMI INTR TRAP flag (single step)	highest  lowest

EOI: End of Interrupt

- Issued to the PIC chips at the end of an IRQ-based interrupt routine
 - If the IRQ came from the Master PIC, it is sufficient to issue this command only to the Master PIC;
 - If the IRQ came from the Slave PIC, it is necessary to issue the command to both PIC chips

```
#define PIC_EOI          0x20          /* End-of-interrupt

void PIC_sendEOI(unsigned char irq)
{
    if(irq >= 8)
        outb(PIC2_COMMAND, PIC_EOI);

    outb(PIC1_COMMAND, PIC_EOI);
}
```

Nested Interrupts

- What if a second interrupt occurs while an interrupt routine is executing?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximizing Parallelism

- Keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation
 - Another request should be issued ASAP
- Most devices do not interfere with each others' data structures
 - No reason to block out other devices

Handling Nested Interrupts

- As soon as possible, unmask the global interrupt
- As soon as reasonable, re-enable interrupts from that IRQ
- But that is not always a great idea, since it could cause **re-entry** to the same handler
- IRQ-specific mask is not enabled during interrupt-handling

Nested Execution

- Interrupts can be interrupted
 - By different interrupts; handlers need not be reentrant
 - Small portions execute with interrupts disabled
 - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
 - By interrupts (devices needing service)
- Exceptions can nest two levels deep
 - Exceptions indicate coding error
 - Exception code (kernel code) should not have bugs
 - Page fault is possible (trying to touch user data)

Interrupt Masking

- Two different types: global and selective
 - Global — delays all interrupts
 - Selective — mask individual IRQs selectively
 - This is usually what's needed — interference most common from two interrupts of the same type
- NMI (Non-Maskable Interrupt)

Triple Fault

- Things never to do in an OS #1: Swap out the page swapping code (triple-fault here we come) —Kemp
- When a fault occurs, the CPU invokes an exception handler.
- If a fault occurs while trying to invoke the exception handler, that's called a double fault, which the CPU tries to handle with yet another exception handler.
- If that invocation results in a fault too, the system reboots with a triple fault.

/proc/interrupts

```
$ cat /proc/interrupts
          CPU0
 0:    865119901      IO-APIC-edge    timer
 1:        4      IO-APIC-edge    keyboard
 2:        0        XT-PIC       cascade
 8:        1      IO-APIC-edge     rtc
12:       20      IO-APIC-edge  PS/2 Mouse
14:   6532494      IO-APIC-edge    ide0
15:       34      IO-APIC-edge    ide1
16:       0      IO-APIC-level  usb-uhci
19:       0      IO-APIC-level  usb-uhci
23:       0      IO-APIC-level ehci-hcd
32:       40      IO-APIC-level    ioc0
33:       40      IO-APIC-level    ioc1
48:  273306628      IO-APIC-level    eth0
NMI:       0
ERR:       0
```

Columns: IRQ, count, interrupt controller, devices

More in /proc/pci:

```
$ cat /proc/pci
PCI devices found:
Bus 0, device 0, function 0:
    Host bridge: PCI device 8086:2550 (Intel Corp.) (rev 3).
        Prefetchable 32 bit memory at 0xe8000000 [0xebffff].

Bus 0, device 29, function 1:
    USB Controller: Intel Corp. 82801DB USB (Hub #2) (rev 2).
        IRQ 19.
        I/O at 0xd400 [0xd41f].
Bus 0, device 31, function 1:
    IDE interface: Intel Corp. 82801DB ICH4 IDE (rev 2).
        IRQ 16.
        I/O at 0xf000 [0xf00f].
        Non-prefetchable 32 bit memory at 0x80000000 [0x800003ff].

Bus 3, device 1, function 0:
    Ethernet controller: Broadcom NetXtreme BCM5703X Gigabit Eth (rev 2).
        IRQ 48.
        Master Capable. Latency=64. Min Gnt=64.
        Non-prefetchable 64 bit memory at 0xf7000000 [0xf700ffff].
```

BOTTOM HALF (IN LINUX)

Interrupt Handling Philosophy

- To preserve IRQ order on the same line, must *disable* incoming interrupts on same line
 - New interrupts can **get lost** if controller buffer overflow
- Interrupt preempts what CPU was doing, which may be important
- Even not important, undesirable to block user program for long
- So, handler must run for a **very short time!**
 - Do as little as possible in the interrupt handler
 - Often just: queue a work item and set a flag
 - Defer non-critical actions till later

Intr Handlers Have No Process Context

- Interrupts (as opposed to exceptions) are not associated with particular instructions
- They are also not associated with a given process (user program)
- The currently-running process, at the time of the interrupt, has no relationship whatsoever to that interrupt
- **Interrupt handlers cannot sleep!**

Why No Sleep?

- The interrupt handler has no way of knowing what the interrupted process was doing. An example:
 1. Process1 enters kernel mode.
 2. Process1 acquires LockA.
 3. Interrupt occurs.
 4. ISR tries to acquire LockA.
 5. ISR calls sleep to wait for LockA to be released.
- At this point, you have a deadlock:
 - Process1 can't resume execution until the ISR is done with its stack, but the ISR is blocked waiting for Process1 to release LockA

What Can't You (ISR) Do?

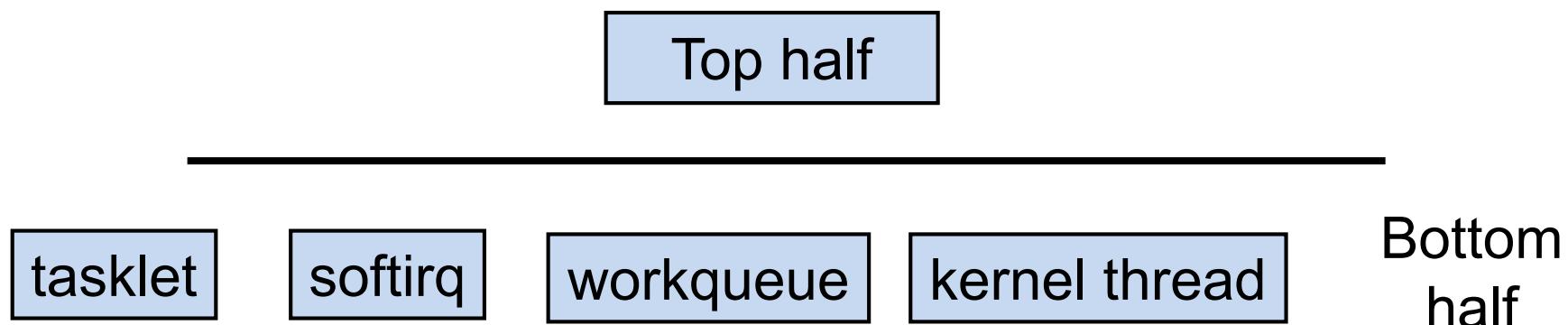
- You cannot sleep
 - or call something that *might* sleep
- You cannot refer to **current**
- You cannot allocate memory with **GPF_KERNEL** (which can sleep), you must use **GPF_ATOMIC** (which can fail)
- You cannot call **schedule()**
- You cannot do a **down()** semaphore call
 - However, you *can* do an **up()**
- You cannot transfer data to/from user space
 - E.g., **copy_to_user()**, **copy_from_user()**

Interrupt Stack

- When an interrupt occurs, what stack is used?
 - **Exceptions**: The *kernel stack* of the current process, whatever it is, is used (There's always some process running — the "idle" process, if nothing else)
 - **Interrupts**: hard IRQ stack (1 per processor)
 - **SoftIRQs**: soft IRQ stack (1 per processor)
- These stacks are configured in the IDT and TSS at boot time by the kernel

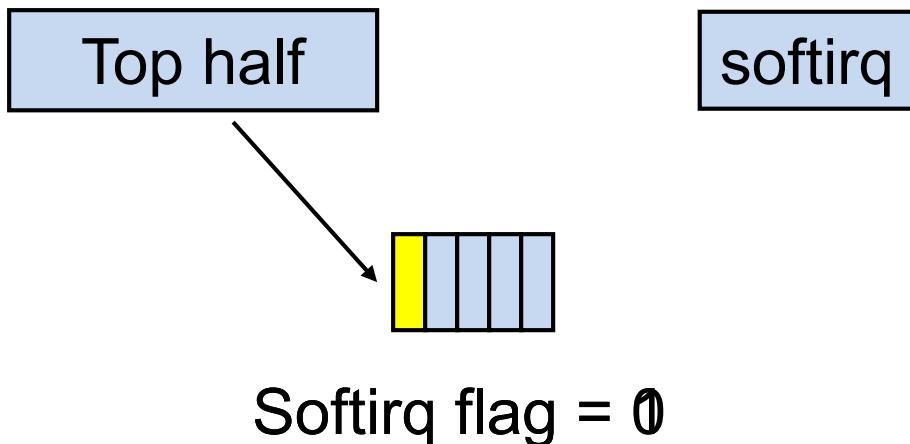
Top and Bottom Halves

- **Top-half:** do minimum work and return
 - ISR (Interrupt Service Routine)
- **Bottom-half:** deferred processing
 - softirqs, tasklets, workqueues, kernel threads



Top Half: Do it Now!

- Perform minimal, common functions: saving registers, unmasking other interrupts. Eventually, undoes that: restores registers, returns to previous context.
- Most important: call proper interrupt handler provided in device drivers (C program)
- Typically queue the request and set a flag for deferred processing



Bottom Half: Do it Later!

- Mechanisms to defer work to later:
 - *softirqs*
 - *tasklets* (built on top of softirqs)
 - *work queues*
 - *kernel threads*
- All can be interrupted

Softirqs

- **Statically** allocated: specified at kernel compile time
- Limited number:

<i>Priority</i>	<i>Type</i>
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	Block devices
5	Regular tasklets

<i>Priority</i>	<i>Type</i>
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	Block devices
5	Regular tasklets

When Do Softirqs Run?

- Run at various points by the kernel:
 - After system calls
 - After exceptions
 - After interrupts (top halves/IRQs, including the timer intr)
 - When the scheduler runs ksoftirqd
- Softirq routines can be executed simultaneously on multiple CPUs:
 - Code must be *re-entrant*
 - Code must do its own locking as needed
- Hardware interrupts are enabled when softirqs run

Rescheduling Softirqs

- A softirq routine can reschedule itself
 - i.e., raise one softirq when handling a softirq
- Problem: while processing one softirq, another is raised. Process it?
 - No -> long delay for new irq ksoftirqd
 - Always -> starve user program when long softirq burst
 - Livelock!
- Solution: Quota + dedicated context ksoftirqd
 - Softirq scheduler only runs a limited number of requests at a time
 - The rest are executed by a kernel thread, ksoftirqd, which competes with user processes for CPU time
 - Ksoftirqd subject to scheduling, as user process

Tasklets

- Problem: softirq is static
 - To add a new type of Softirq, need to convince Linus!
- Solution: tasklets
 - Built on top of softirq
 - New types are created and destroyed dynamically
 - Simplified for multi-core processing: at any time, only **one tasklet** among all of the same type can run
- Problem with softirq and tasklets
 - They have no process contexts, thus cannot sleep

Tasklets

- Can be created and destroyed dynamically
- Run on the CPU that scheduled it (cache affinity)
- Individual tasklets are locked during execution
 - no problem about re-entrancy
 - no need for locking by the code
- Tasklets can run in parallel on multiple CPUs
 - *Same* tasklet can only run on *one* CPU
- Were once the preferred mechanism for most deferred activity, now changing

The Trouble with Tasklets

- Hard to get right
- One has to be careful about sleeping
- Run at higher priority than other tasks
- Can produce uncontrolled latency if coded badly
- Ongoing discussion about eliminating tasklets
 - Will likely slowly fade over time

Work Queues

- Softirqs and tasklets run in an interrupt context; work queues have a process context
- The idea:
 - You throw work (fn, args) to a workqueue
 - Workqueue add to an internal FIFO queue
 - A dedicated *workqueue process* loops forever, dequeuing (fn, args), and running fn(args)
- Since they have a process context, **they can sleep**
 - They are kernel-only; no user mode associated with it
 - Do not try copying data into/out of user space

Kernel Threads

- Always operate in kernel mode
 - Again, no user context
- 2.6.30 introduced the notion of *threaded interrupt handlers*
 - Imported from the realtime tree
 - `request_threaded_irq()`
 - Now each bottom half has its own context, unlike work queues
 - Idea is to eventually replace tasklets and work queues

Comparing Approaches

	ISR	SoftIRQ	Tasklet	WorkQueue	KThread
Will disable all interrupts?	Briefly	No	No	No	No
Will disable other instances of self?	Yes	Yes	No	No	No
Higher priority than regular scheduled tasks?	Yes	Yes*	Yes*	No	No
Will be run on same processor as ISR?	N/A	Yes	Yes	Yes	Maybe
More than one run can on same CPU?	No	No	No	Yes	Yes
Same one can run on multiple CPUs?	Yes	Yes	No	Yes	Yes
Full context switch?	No	No	No	Yes	Yes
Can sleep? (Has own kernel stack)	No	No	No	Yes	Yes
Can access user space?	No	No	No	No	No

*Within limits, can be run by ksoftirqd

SYSTEM CALLS

System Calls in Previous Classes (Partial)

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

Tracing System Calls

- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- The `ptrace()` system call is used
 - Also used by debuggers (breakpoint, singlestep, etc)
- Use the "`strace`" command (man strace for info)
- You can trace library calls using the "`ltrace`" command

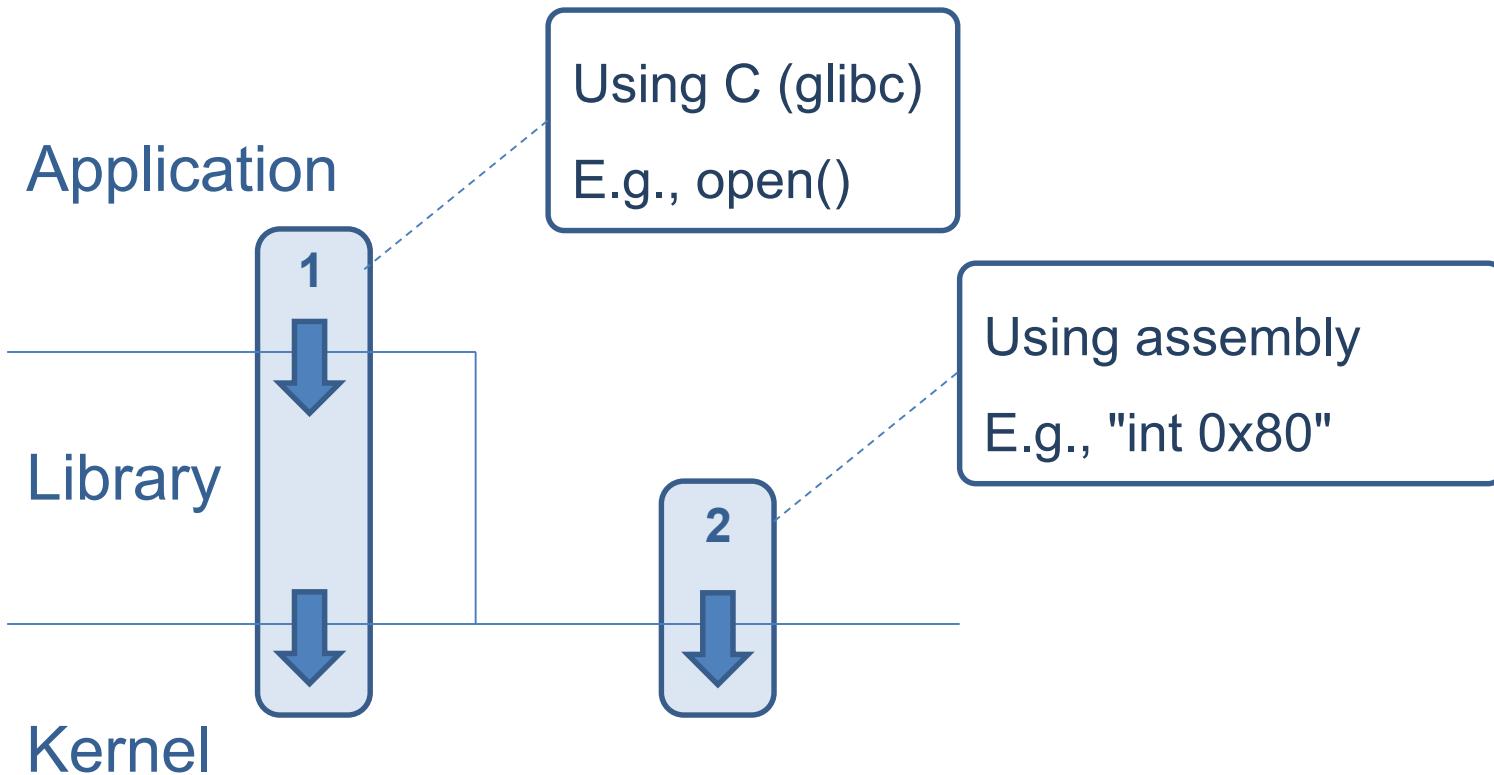
Using strace

```
int main() {
    write(1, "Hello world!\n", 13);
}
```

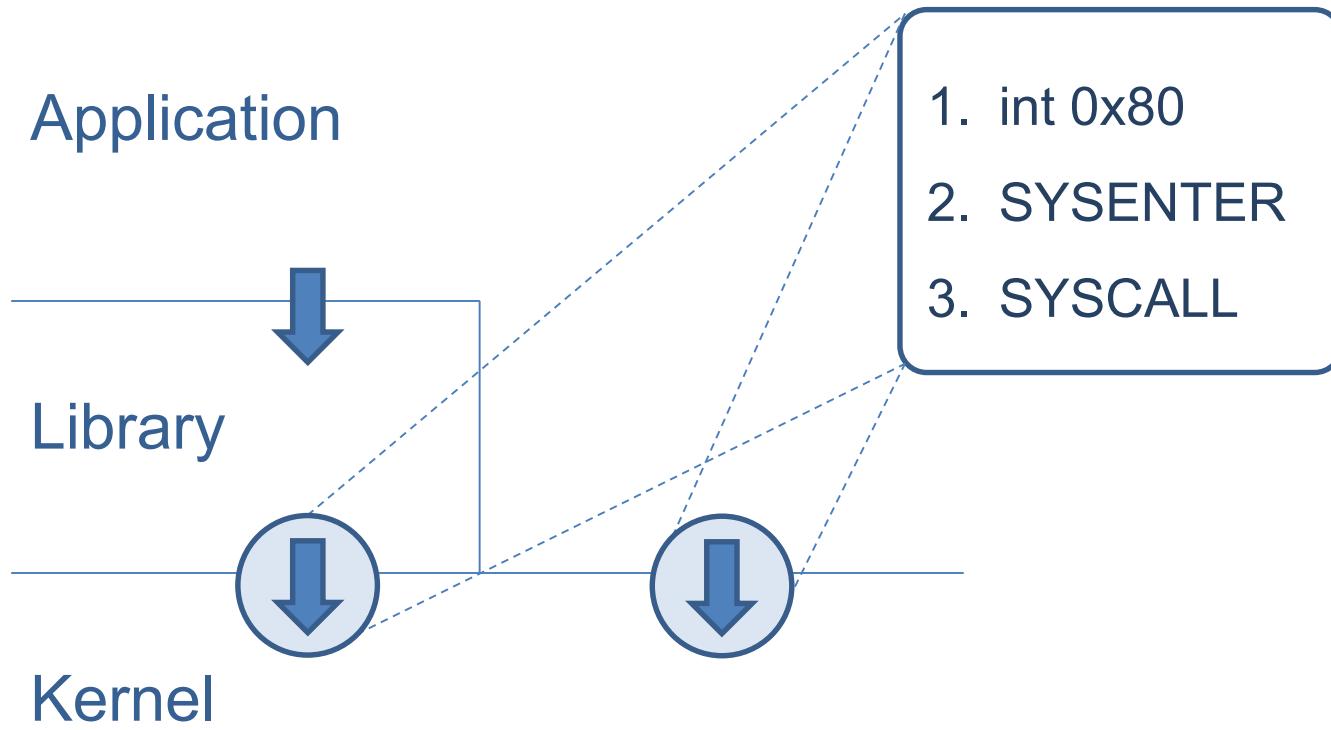
```
$ strace -o hello.out ./hello
```

```
execve("./hello2", ["./hello2"], /* 59 vars */) = 0
uname({sys="Linux", node="kiwi", ...}) = 0
brk(0) = 0xca9000
brk(0caa1c0) = 0caa1c0
arch_prctl(ARCH_SET_FS, 0xa9880) = 0
brk(0xccb1c0) = 0ccb1c0
brk(0xccc000) = 0ccc000
write(1, "Hello world!\n", 13) = 13
exit_group(13) = ?
```

2 Ways to Do System Calls: From Coder's View



3 Ways to Do System Calls: From Machine's View



Example of Calling `exit()`

```
int main(int argc, char *argv[]) {
    unsigned int syscall_nr = 60; // 60 is for exit()
    int exit_status = 42; // 42 is the return value
    asm ("movl %0, %%eax\n"
         "movl %1, %%ebx\n"
         "int $0x80"           ←
         :
         : "m" (syscall_nr), "m" (exit_status)
         : "eax", "ebx");
}
```

Lib call/Syscall Return Codes

- Library calls return -1 on error and place a specific error code in the global variable `errno`
- System calls return **specific negative values** to indicate an error in `%eax`
 - Most system calls return `-errno`
- The library **wrapper code** is responsible for conforming the return values to the `errno` convention

Passing System Call Parameters

- The first parameter is always the **syscall #**
 - eax on Intel
- Linux allows up to **six additional parameters**
 - ebx, ecx, edx, esi, edi, ebp on Intel
- System calls that require more parameters **package the remaining params in a struct** and pass a pointer to that struct as the sixth parameter
- **Problem: must validate pointers**
 - Could be invalid, e.g. NULL → crash OS
 - Or worse, could point to OS, device memory → security hole

How to validate user pointers?

- Too expensive to do a thorough check
 - Need to check that the pointer is within all valid memory regions of the calling process
- Solution: No comprehensive check
 - Linux does a **simple check** for address pointers and only determines if pointer variables are within the **largest possible** range of user memory (more details when talking about process)
 - Even if a **pointer value** passes this check, it is still quite possible that the specific value is invalid
 - Dereferencing an **invalid pointer** in kernel code would normally be interpreted as a **kernel bug** and generate an **Oops** message on the console and kill the offending process
 - Linux does something very sophisticated to avoid this situation

Handling faults due to user-pointers

- Kernel code must access user-pointers using a small set of "paranoid" routines (e.g. `copy_from_user`)
 - Thus, kernel knows what addresses in its code can throw invalid memory access exceptions (page fault)
- When a page fault occurs, the kernel's page fault handler checks the faulting EIP (recall: saved by hw)
- If EIP matches one of the paranoid routines, kernel will not oops; instead, will call "fixup" code
- Many violations of this rule in Linux. Tons of security holes are found

Paranoid functions to access user pointers

Function	Action
get_user(), __get_user()	reads integer (1,2,4 bytes)
put_user(), __put_user()	writes integer (1,2,4 bytes)
copy_from_user(), __copy_from_user	copy a block from user space
copy_to_user(), __copy_to_user()	copy a block to user space
strncpy_from_user(), __strncpy_from_user()	copies null-terminated string from user space
strnlen_user(), __strnlen_user()	returns length of null-terminated string in user space
clear_user(), __clear_user()	fills memory area with zeros

New Instruction: **SYSENTER/SYSEXIT & SYSCALL/SYSRET**

- "int 0x80" is Obsolete
 - Windows XP does not use it any more
 - That's why it requires at least Pentium II
- New Instructions
 - SYSENTER/SYSEXIT: Intel, and then AMD
 - SYSCALL/SYSRET: AMD, and then Intel

Simplify System Call by **SYSCALL/SYSRET**

- SYSCALL/SYSRET has lower latency
 - Less than 25% of the cycles of previous solution
- Why faster?
 - Assume OS implements a flat-memory model
 - Simplifies calls to and returns from the OS
 - Eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers

More details: <http://wiki.osdev.org/SYSENTER>

Compatibility across Intel and AMD

- For a 32-bit Kernel
 - SYSENTER/SYSEXIT are the only compatible pair
- For a 64-bit Kernel
 - SYSCALL/SYSRET are the only compatible pair
 - In Long mode only (not Long Compat mode)

Using SYSCALL to Invoke the `exit()` System Call

```
#include <unistd.h>
int main(int argc, char *argv[]) {
    unsigned long syscall_nr = 60;
    long exit_status = 42;
    syscall(syscall_nr, exit_status);
}
```

```
int main(int argc, char *argv[]) {
    unsigned long syscall_nr = 60;
    long exit_nr = 42;
    asm ("movq %0, %%rax\n"
           "movq %1, %%rdi\n"
           "syscall\n"
           " : : "
           "m" (syscall_nr), "m" (exit_nr) :
           "rax", "rdi"); }
```

System Call

1. Fetch the n 'th descriptor from the IDT, where n is the argument of int.
2. Check that CPL in %cs is \leq DPL, where DPL is the privilege level in the descriptor.
3. Save %esp and %ss in a CPU-internal registers, but only if the target segment selector's PL $<$ CPL.
4. Load %ss and %esp from a task segment descriptor.
5. Push %ss, %esp, %eflags, %cs, %eip,
6. Clear some bits of %eflags
7. Set %cs and %eip to the values in the descriptor

System Call: INT Instruction

1. decide the vector number, in this case it's the 0x80 in ``int 0x80''
2. fetch the interrupt descriptor for vector 0x80 from the IDT.
 - (the CPU finds it by taking the 0x40'th 8-byte entry starting at the physical address that the IDTR CPU register points to)
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL) (*)
5. load SS and ESP from TSS
6. push user SS (*)
7. push user ESP (*)
8. push EFLAGS
9. push CS
10. push EIP
11. clear some EFLAGS bits (on interrupts, see below)
12. set CS and EIP from IDT descriptor's segment selector and offset

4, 6 and 7 are not needed if trap happens in kernel space

Virtual Dynamic Shared Object

VDSO

The Motivation of vDSO

- The latency of syscall is not negligible
 - Especially for those called frequently
 - E.g., gettimeofday()
- How to reduce the latency of syscall?
 - Most of the time is for state saving/restoring
 - If no mode switching, then no state saving/restoring

The Code of *gettimeofday()*

- Defined by the kernel
 - As a part of kernel code during compiling
- Run in user space
 - The code is loaded into a page shared with user
 - The page is known as vDSO
 - Virtual Dynamic Shared Object
 - Time value is also mapped to user space (read-only)
 - Can only be changed in kernel mode

The source can be found in arch/x86/vdso/vclock_gettime.c

Where is the Shared Page for vDSO?

```
$ ldd `which bash`  
linux-vdso.so.1 => (0x00007fff667ff000)  
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f623df7d000)  
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f623dd79000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f623d9ba000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

Flexible System Call Scheduling with Exception-Less System Calls,
OSDI'10

FLEX-SC

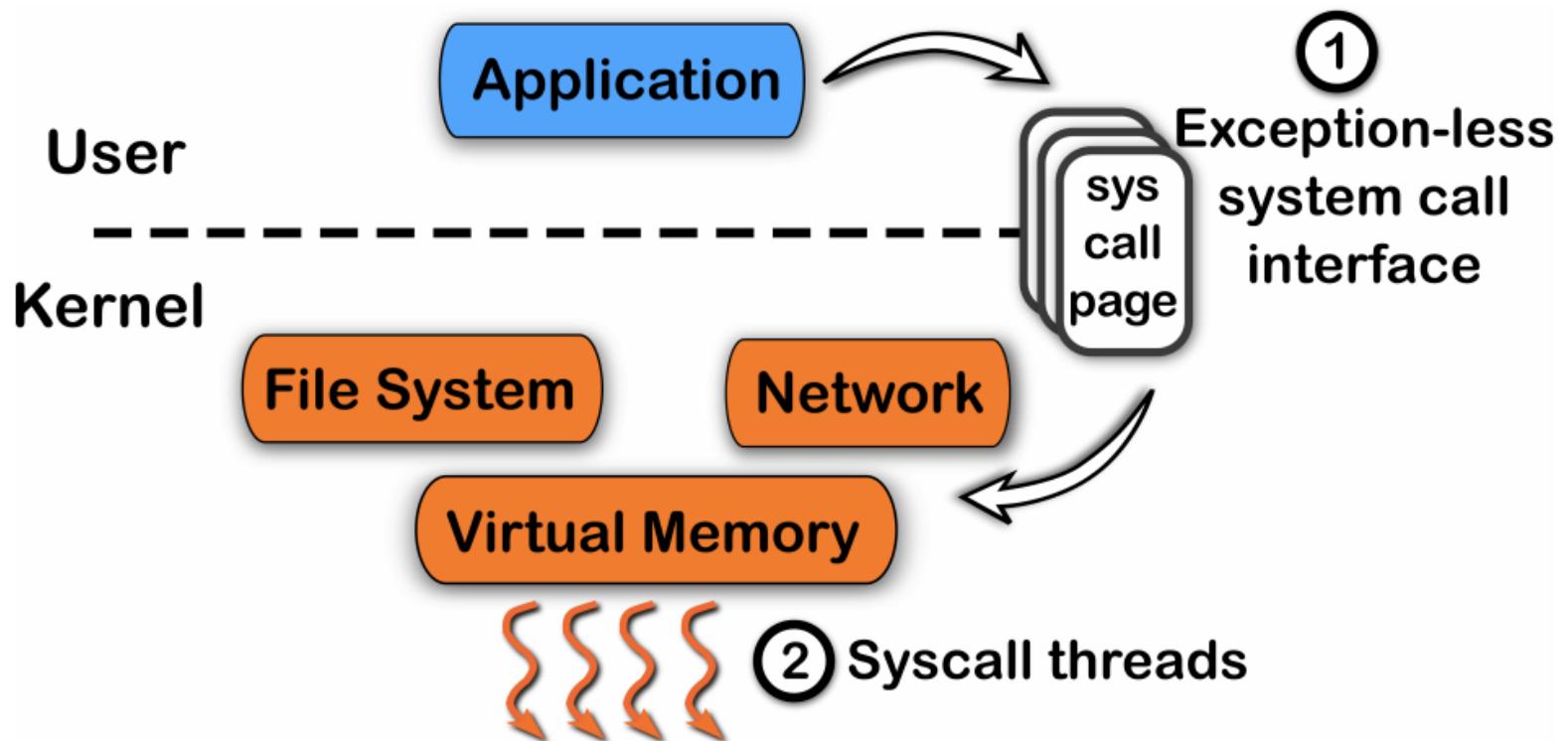
The Motivation

- How to further reduce the latency of syscall?
 - Not only for gettimeofday()
- Where does the latency come from?
 - Mostly for the state switch
 - Save and restore states
 - Privilege checking
 - Cache pollution
- Could we do syscall without state switching?

Flexible System Call

- New syscall mechanism – **Flexible System Call**
 - Introduce **system call page** that is shared by user & kernel
 - User threads can **push** the system call requests into the system call page
 - kernel threads will **poll** the system call requests out the system call page
- Exception-less syscall
 - Remove synchronicity by decoupling invocation from execution

FlexSC: Another Way to System Call



Exception-less System Call

```
write(fd, buf, 4096);
```

```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
    do_something_else();

return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
⋮				

Kernel Fill the Results

```
write(fd, buf, 4096);

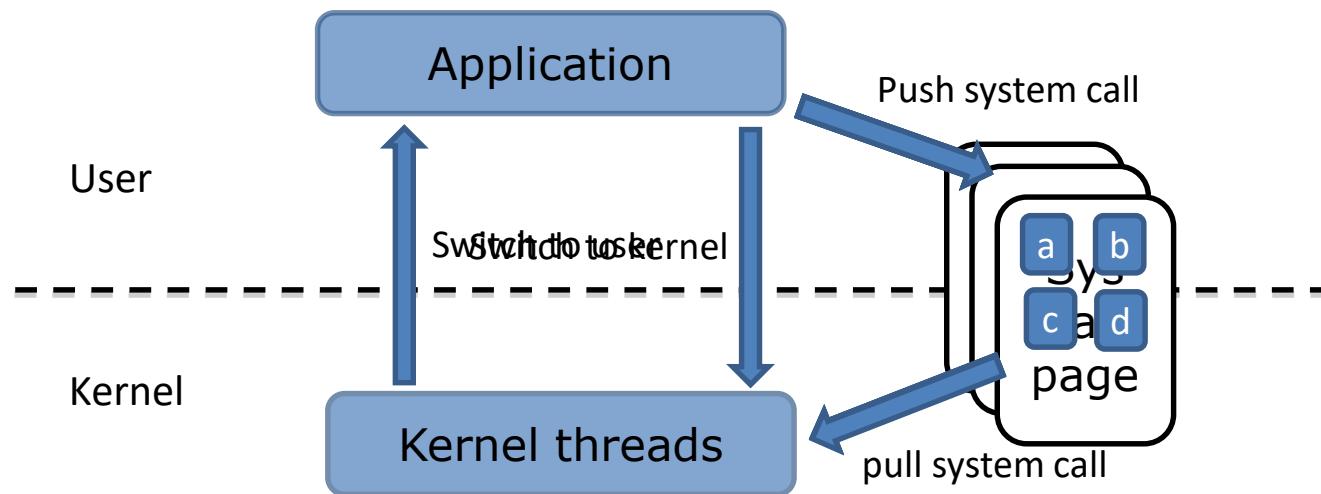
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

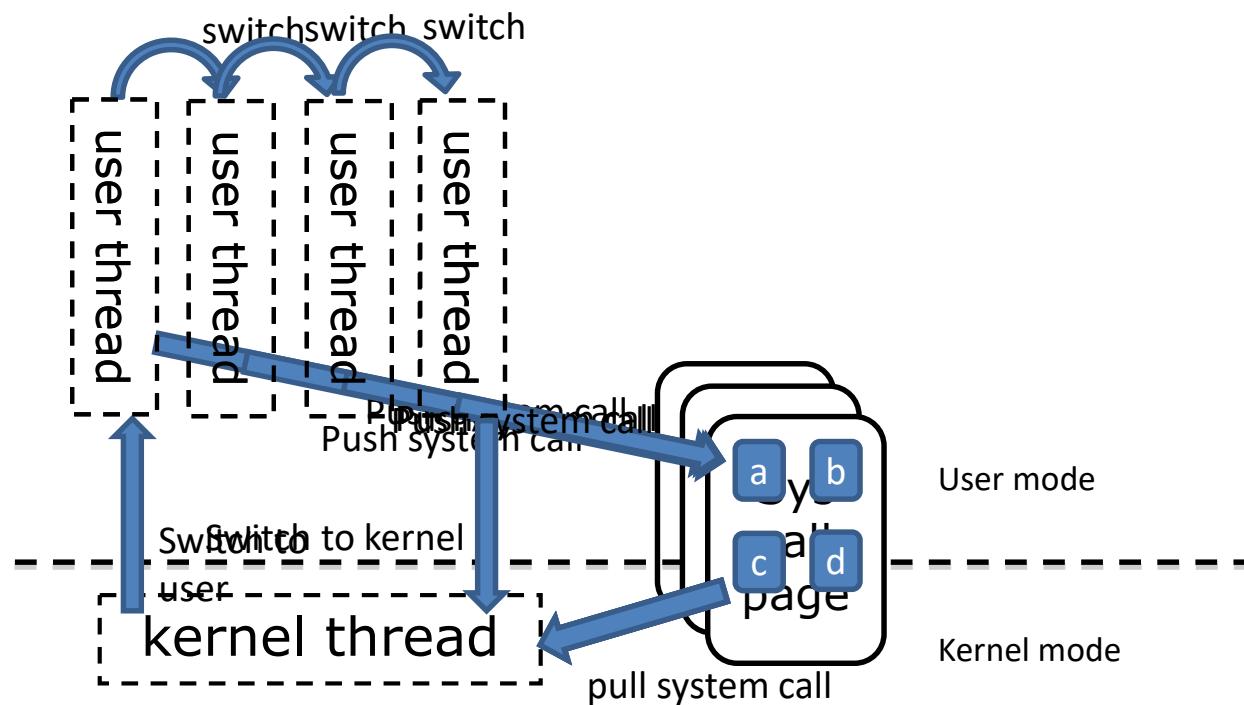
while (entry->status != DONE)
    do_something_else();

return entry->return_code;
```

Running on a Single Core: Single Threads



Running on a Single Core: Multiple Threads



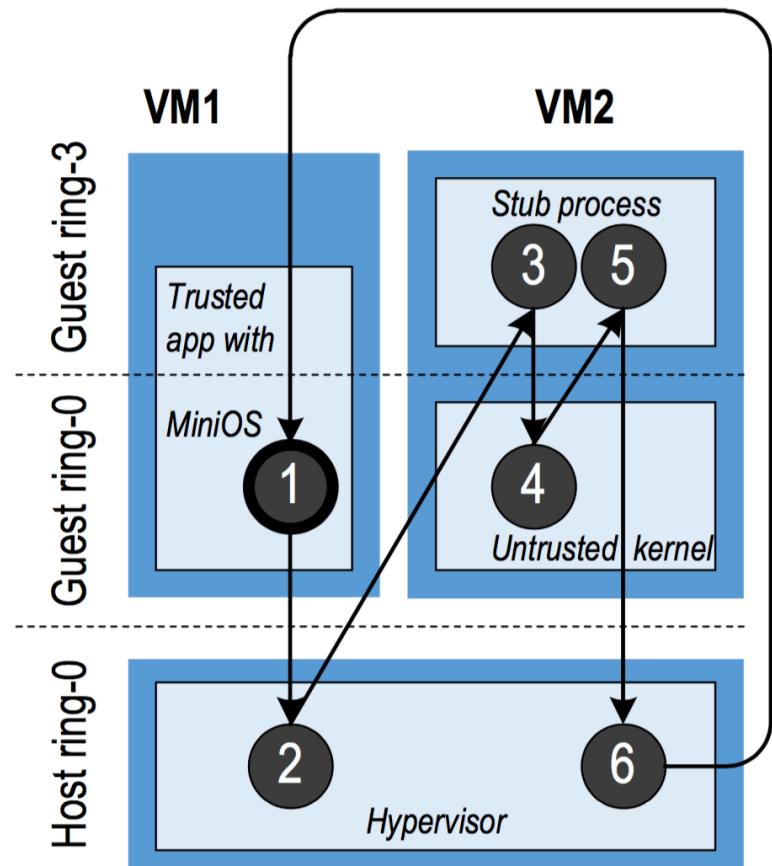
FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

Conclusion

- System call VS. function call
 - Trap! Switching the mode
 - State saving/restoring, changing stack, checking...
- ‘Int 0x80’ is not enough
 - New instructions for lower latency
 - vDSO for reducing trapping
 - FlexSC for further reducing trapping by multi-core

Cross-Mode Calls

- ProxOS
 - Two virtual machines, one is trusted and the other is untrusted
 - Redirect untrusted system calls to the untrusted VMs
 - Overhead of mode switching could be high!
- More
 - Nested Virtualization



Optimization for Cross-Mode Calls

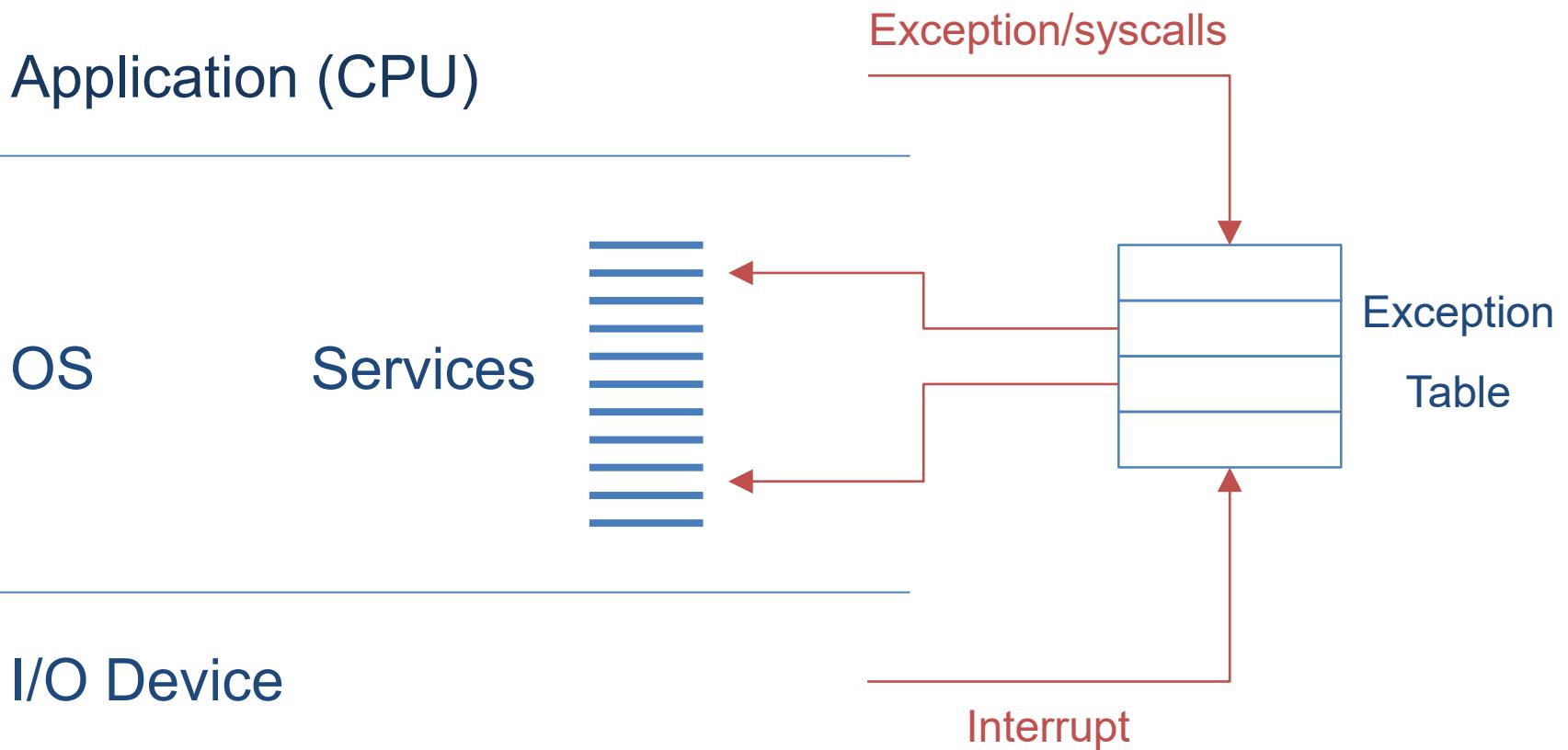
- Using shared memory (like FlexSC)
 - Make synchronous calls to asynchronous ones
 - Require app logic to be changed
- How about a new instruction for world-call?
 - A world is just a mode (including VM, kernel/user, etc.)
 - Separating protection from managing
 - Configure by software (managing)
 - Check by hardware (protection)

System Calls

Yubin Xia
IPADS, SJTU

ACKs: Some slides are adapted from the textbook's original slides and
Frans's os course notes

Review: OS as Services

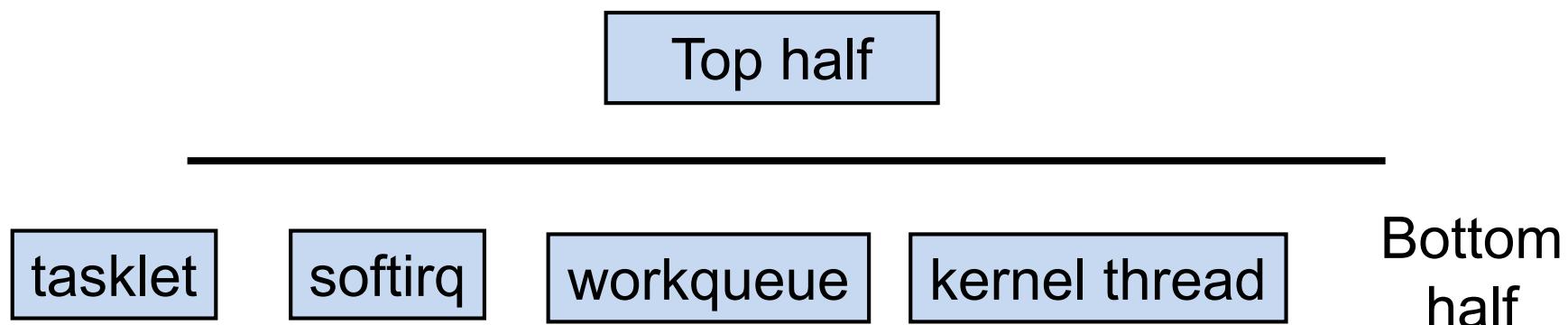


Review: Maximizing Parallelism

- Keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation
 - Another request should be issued ASAP
- Most devices do not interfere with each others' data structures
 - No reason to block out other devices

Review: Top and Bottom Halves

- **Top-half:** do minimum work and return
 - ISR (Interrupt Service Routine)
- **Bottom-half:** deferred processing
 - softirqs, tasklets, workqueues, kernel threads



SYSTEM CALLS

System Calls in Previous Classes (Partial)

Number	Name	Description	Number	Name	Description
1	exit	Terminate process	27	alarm	Set signal delivery alarm clock
2	fork	Create new process	29	pause	Suspend process until signal arrives
3	read	Read file	37	kill	Send signal to another process
4	write	Write file	48	signal	Install signal handler
5	open	Open file	63	dup2	Copy file descriptor
6	close	Close file	64	getppid	Get parent's process ID
7	waitpid	Wait for child to terminate	65	getpgrp	Get process group
11	execve	Load and run program	67	sigaction	Install portable signal handler
19	lseek	Go to file offset	90	mmap	Map memory page to file
20	getpid	Get process ID	106	stat	Get information about file

Review: Open Files

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(char *filename, int flags, mode_t mode);
```

Returns: new file descriptor if OK, -1 on error

Why return fd?
Why not pointer?

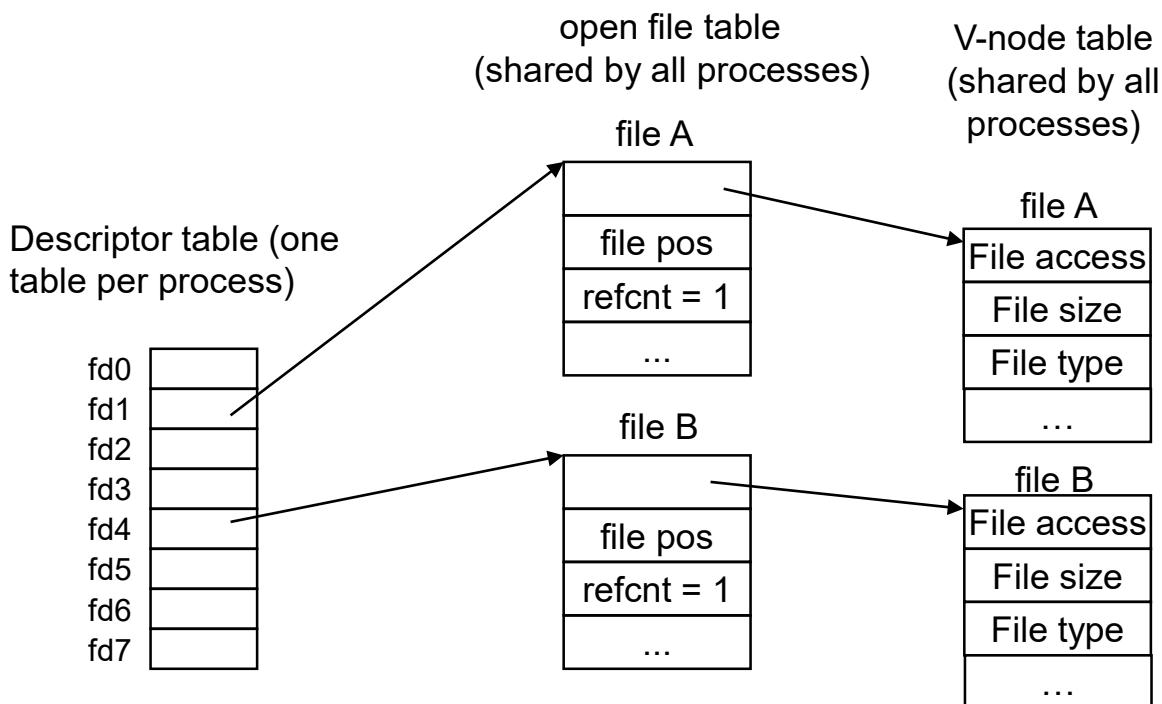
flags

- O_RDONLY, O_WRONLY, O_RDWR (must have one)
- O_CREAT, O_TRUNC, O_APPEND (optional)

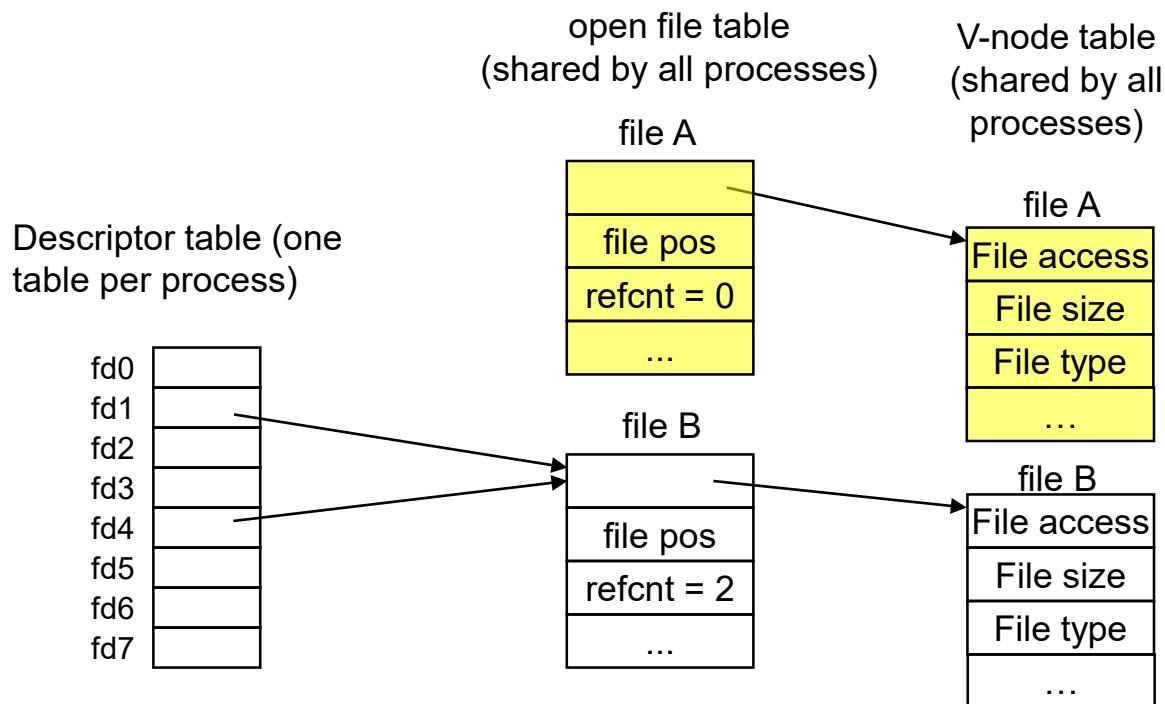
mode

- S_IRUSR, S_IWUSR, S_IXUSR
- S_IRGRP, S_IWGRP, S_IXGRP
- S_IROTH, S_IWOTH, S_IXOTH

FD Table & Opened File Table after fork()



Redirection by dup2(4,1) after fork()



Tracing System Calls

- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- The `ptrace()` system call is used
 - Also used by debuggers (breakpoint, singlestep, etc)
- Use the "`strace`" command (man strace for info)
- You can trace library calls using the "`Itrace`" command

Using strace

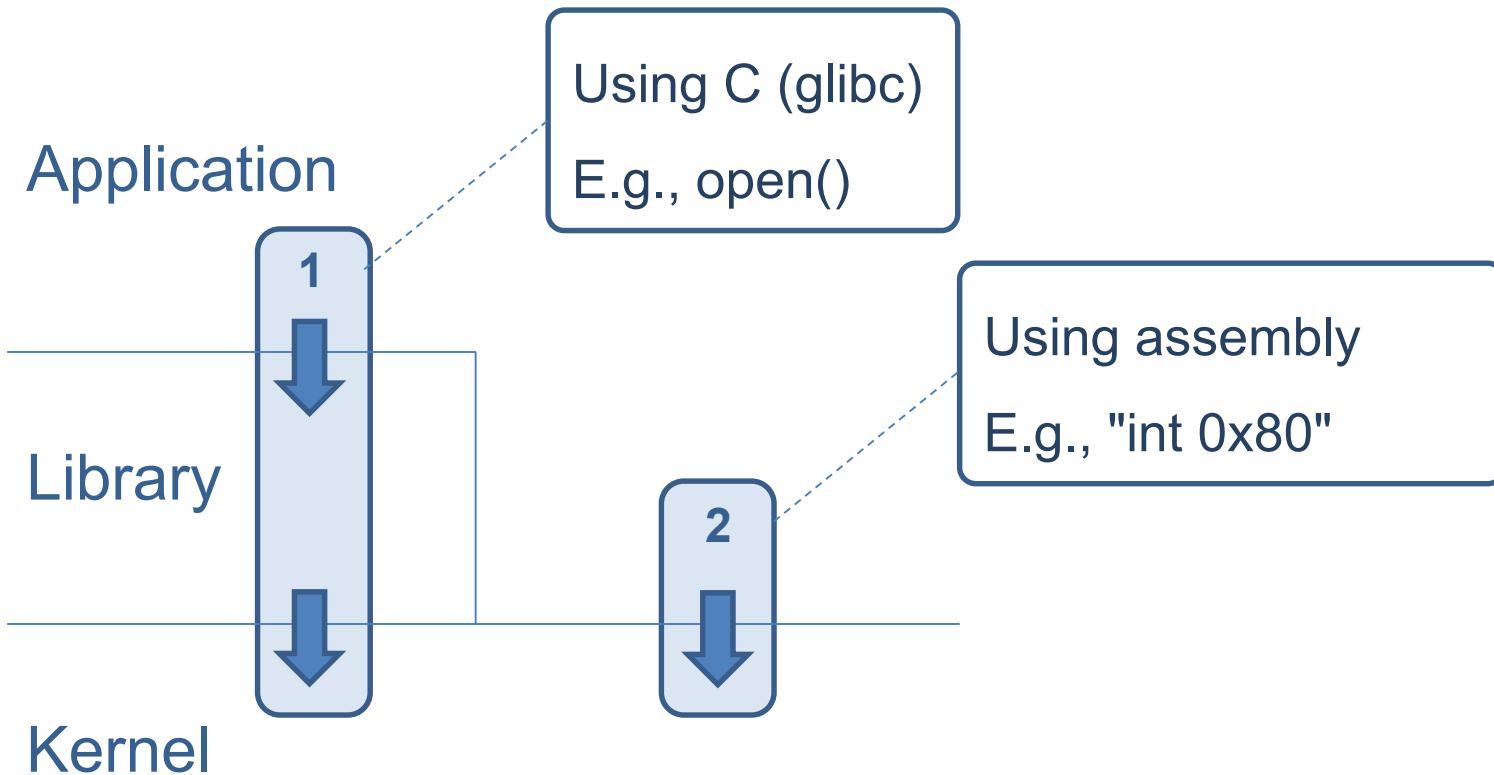
```
int main() {
    write(1, "Hello world!\n", 13);
}
```

```
$ strace -o hello.out ./hello
```

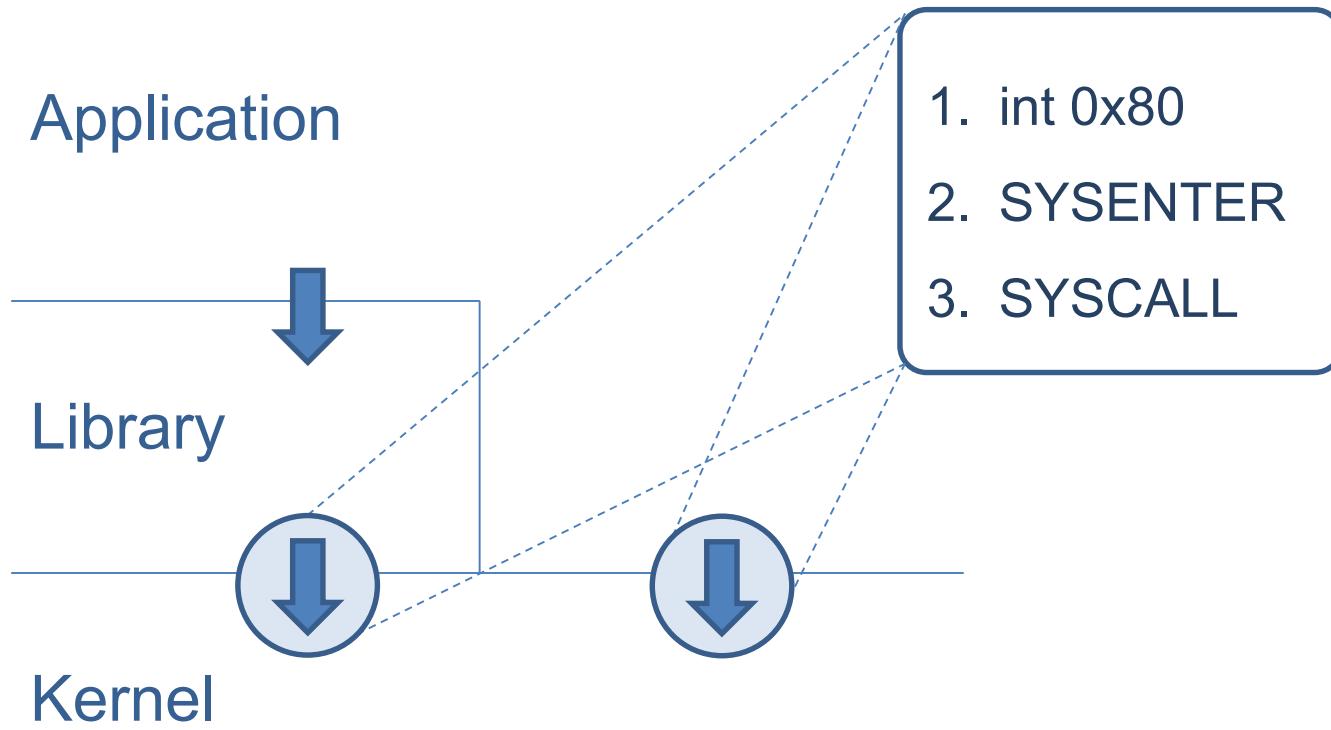
```
execve("./hello2", ["./hello2"], /* 59 vars */) = 0
uname({sys="Linux", node="kiwi", ...}) = 0
brk(0) = 0xca9000
brk(0caa1c0) = 0caa1c0
arch_prctl(ARCH_SET_FS, 0xa9880) = 0
brk(0xccb1c0) = 0ccb1c0
brk(0xccc000) = 0ccc000
write(1, "Hello world!\n", 13) = 13
exit_group(13) = ?
```

MAKE A SYSTEM CALL

2 Ways to Do System Calls: From Coder's View



3 Ways to Do System Calls: From Machine's View



Example of Calling `exit()`

```
int main(int argc, char *argv[]) {
    unsigned int syscall_nr = 60; // 60 is for exit()
    int exit_status = 42; // 42 is the return value
    asm ("movl %0, %%eax\n"
         "movl %1, %%ebx\n"
         "int $0x80"           ←
         :
         : "m" (syscall_nr), "m" (exit_status)
         : "eax", "ebx");
}
```

Lib-call / Syscall Return Codes

- Library calls return -1 on error and place a specific error code in the global variable `errno`
- System calls return **specific negative values** to indicate an error in `%eax`
 - Most system calls return `-errno`
- The library **wrapper code** is responsible for conforming the return values to the `errno` convention

Passing System Call Parameters

- The first parameter is always the **syscall #**
 - eax on Intel
- Linux allows up to **six additional parameters**
 - ebx, ecx, edx, esi, edi, ebp on Intel
- System calls that require more parameters **package the remaining params in a struct** and pass a pointer to that struct as the sixth parameter
- **Problem: must validate pointers**
 - Could be invalid, e.g. NULL → crash OS
 - Or worse, could point to OS, device memory → security hole

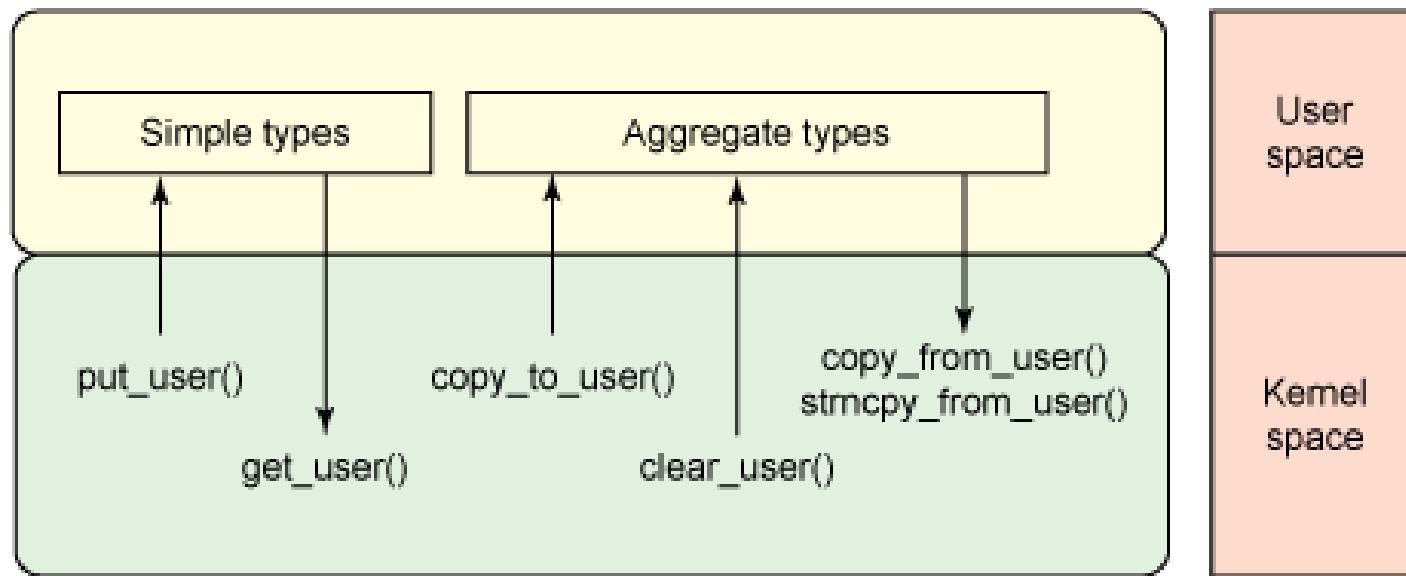
How to validate user pointers?

- Too expensive to do a thorough check
 - Need to check that the pointer is within **all valid memory regions** of the calling process
- Solution: No comprehensive check
 - Linux does a **simple check** for address pointers and only determines if pointer variables are within the **largest possible** range of user memory (more details when talking about process)
 - Even if a pointer value passes this check, it is still quite possible that the specific value is invalid
 - Dereferencing an invalid pointer in kernel code would normally be interpreted as a kernel bug and generate an Oops message on the console and kill the offending process
 - Linux does something very sophisticated to avoid this situation

Handling faults due to user-pointers

- Kernel code must access user-pointers using a small set of "paranoid" routines (e.g. `copy_from_user()`)
 - Thus, kernel knows what addresses in its code can throw invalid memory access exceptions (page fault)
- When a page fault occurs, the kernel's page fault handler checks the faulting EIP (recall: saved by hw)
- If EIP matches one of the paranoid routines, kernel will not oops; instead, will call "fixup" code
- Many violations of this rule in Linux. Tons of security holes are found

Access User's Memory



Paranoid functions to access user pointers

Function	Action
get_user(), __get_user()	reads integer (1,2,4 bytes)
put_user(), __put_user()	writes integer (1,2,4 bytes)
copy_from_user(), __copy_from_user	copy a block from user space
copy_to_user(), __copy_to_user()	copy a block to user space
strncpy_from_user(), __strncpy_from_user()	copies null-terminated string from user space
strnlen_user(), __strnlen_user()	returns length of null-terminated string in user space
clear_user(), __clear_user()	fills memory area with zeros

New Instruction: SYSENTER/SYSEXIT & SYSCALL/SYSRET

- "int 0x80" is Obsolete
 - Windows XP does not use it any more
 - That's why it requires at least Pentium II
- New Instructions
 - SYSENTER/SYSEXIT: Intel, and then AMD
 - SYSCALL/SYSRET: AMD, and then Intel

Simplify System Call by SYSCALL/SYSRET

- SYSCALL/SYSRET has lower latency
 - Less than 25% of the cycles of previous solution
- Why faster?
 - Assume OS implements a flat-memory model
 - Simplifies calls to and returns from the OS
 - Eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers

More details: <http://wiki.osdev.org/SYSENTER>

Compatibility across Intel and AMD

- For a 32-bit Kernel
 - SYSENTER/SYSEXIT are the only compatible pair
- For a 64-bit Kernel
 - SYSCALL/SYSRET are the only compatible pair
 - In Long mode only (not Long Compat mode)

Using SYSCALL to Invoke the `exit()` System Call

```
#include <unistd.h>
int main(int argc, char *argv[]) {
    unsigned long syscall_nr = 60;
    long exit_status = 42;
    syscall(syscall_nr, exit_status);
}
```

```
int main(int argc, char *argv[]) {
    unsigned long syscall_nr = 60;
    long exit_nr = 42;
    asm ("movq %0, %%rax\n"
           "movq %1, %%rdi\n"
           "syscall\n"
           " : : "
           "m" (syscall_nr), "m" (exit_nr) :
           "rax", "rdi"); }
```

System Call

1. Fetch the n 'th descriptor from the IDT, where n is the argument of int.
2. Check that CPL in %cs is \leq DPL, where DPL is the privilege level in the descriptor.
3. Save %esp and %ss in a CPU-internal registers, but only if the target segment selector's PL $<$ CPL.
4. Load %ss and %esp from a task segment descriptor.
5. Push %ss, %esp, %eflags, %cs, %eip,
6. Clear some bits of %eflags
7. Set %cs and %eip to the values in the descriptor

What does CPU do for “INT 0x80”

1. decide the vector number, in this case it's the 0x80 in “int 0x80”
2. fetch the interrupt descriptor for vector 0x80 from the IDT.
 - CPU finds it by taking the 0x40'th 8-byte entry starting at the address that the IDTR CPU register points to
3. check that CPL <= DPL in the descriptor (but only if INT instruction).
4. save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL) (*)
5. load SS and ESP from TSS
6. push user SS (*)
7. push user ESP (*)
8. push EFLAGS
9. push CS
10. push EIP
11. clear some EFLAGS bits (on interrupts, see below)
12. set CS and EIP from IDT descriptor's segment selector and offset

4, 6 and 7 are not needed if trap happens in kernel space

Virtual Dynamic Shared Object

VDSO

The Motivation of vDSO

- The latency of syscall is not negligible
 - Especially for those called frequently
 - E.g., gettimeofday()
- How to reduce the latency of syscall?
 - Most of the time is for state saving/restoring
 - If no mode switching, then no state saving/restoring

The Code of *gettimeofday()*

- Defined by the kernel
 - As a part of kernel code during compiling
- Run in user space
 - The code is loaded into a page shared with user
 - The page is known as vDSO
 - Virtual Dynamic Shared Object
 - Time value is also mapped to user space (read-only)
 - Can only be changed in kernel mode

The source can be found in arch/x86/vdso/vclock_gettime.c

Where is the Shared Page for vDSO?

```
$ ldd `which bash`  
linux-vdso.so.1 => (0x00007fff667ff000)  
libtinfo.so.5 => /lib/x86_64-linux-gnu/libtinfo.so.5 (0x00007f623df7d000)  
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f623dd79000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f623d9ba000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f623e1ae000)
```

Flexible System Call Scheduling with Exception-Less System Calls,
OSDI'10

FLEX-SC

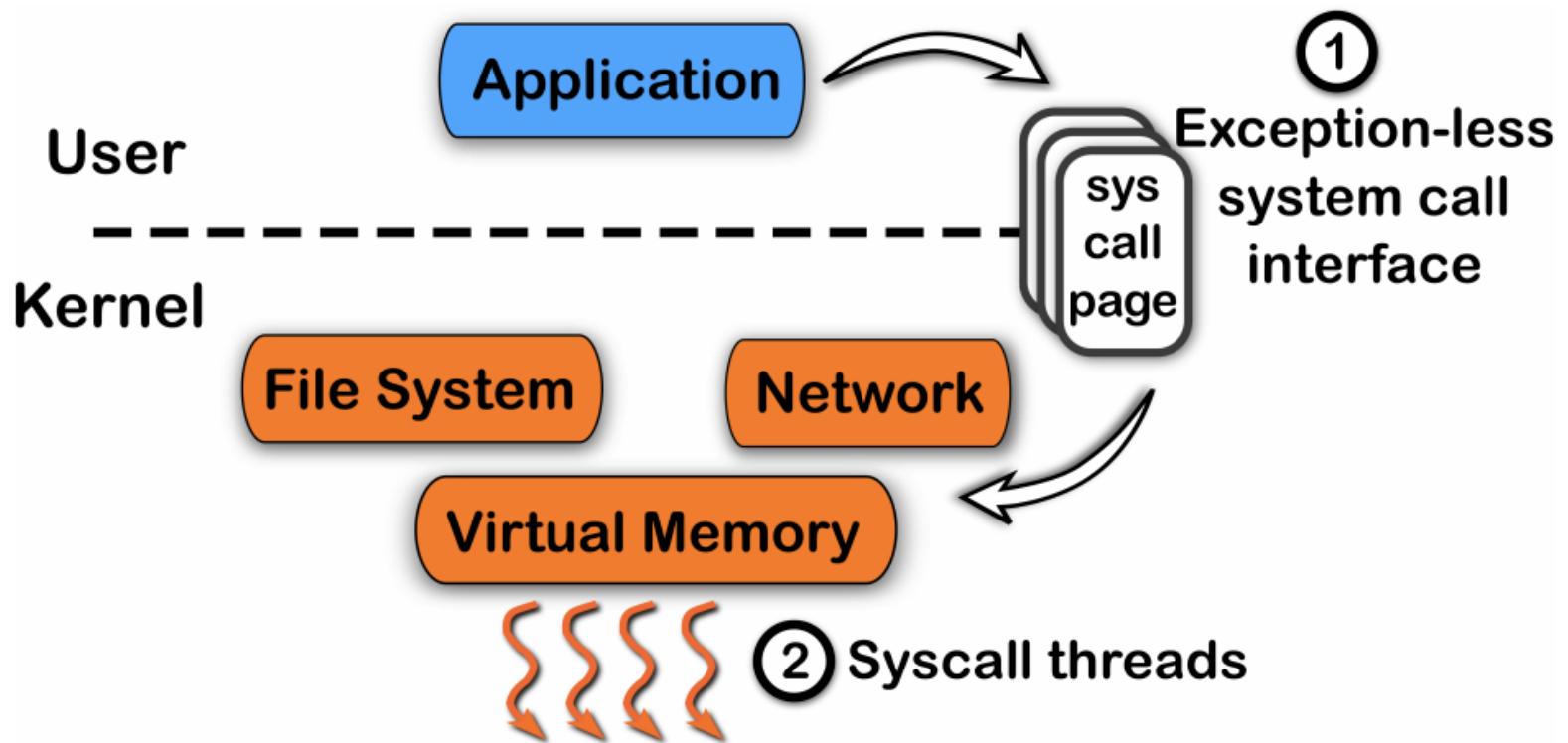
The Motivation

- How to further reduce the latency of syscall?
 - Not only for gettimeofday()
- Where does the latency come from?
 - Mostly for the state switch
 - Save and restore states
 - Privilege checking
 - Cache pollution
- Could we do syscall without state switching?

Flexible System Call

- New syscall mechanism – **Flexible System Call**
 - Introduce **system call page** that is shared by user & kernel
 - User threads can **push** the system call requests into the system call page
 - kernel threads will **poll** the system call requests out the system call page
- Exception-less syscall
 - Remove synchronicity by decoupling invocation from execution

Another Way for System Call



Exception-less System Call

```
write(fd, buf, 4096);
```

```
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

while (entry->status != DONE)
    do_something_else();

return entry->return_code;
```

syscall number	number of args	args 0 ... 6	status	return code
⋮				

Kernel Fill the Results

```
write(fd, buf, 4096);

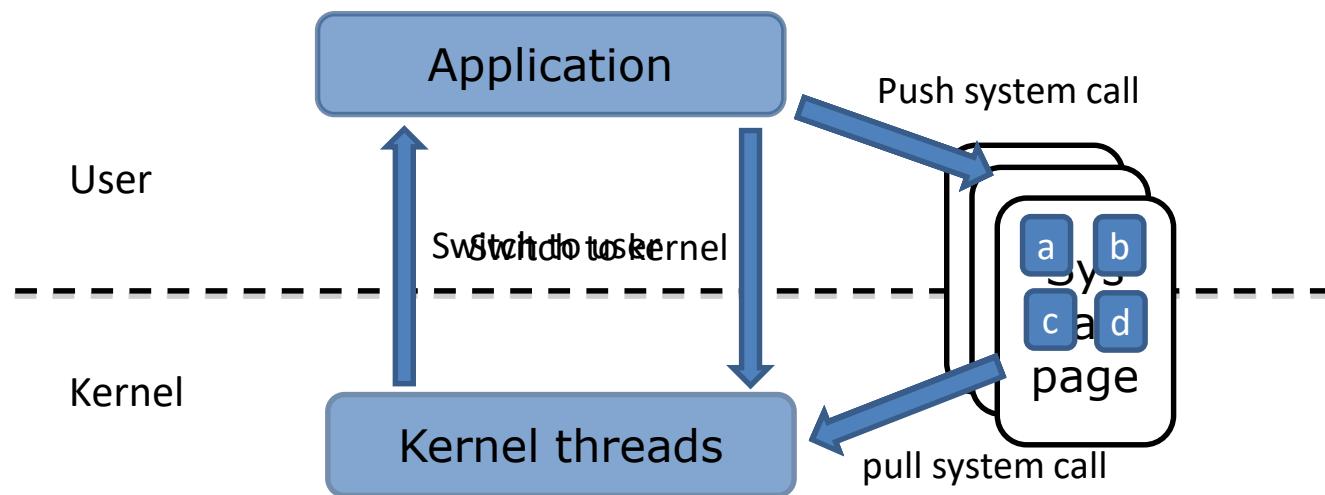
entry = free_syscall_entry();

/* write syscall */
entry->syscall = 1;
entry->num_args = 3;
entry->args[0] = fd;
entry->args[1] = buf;
entry->args[2] = 4096;
entry->status = SUBMIT;

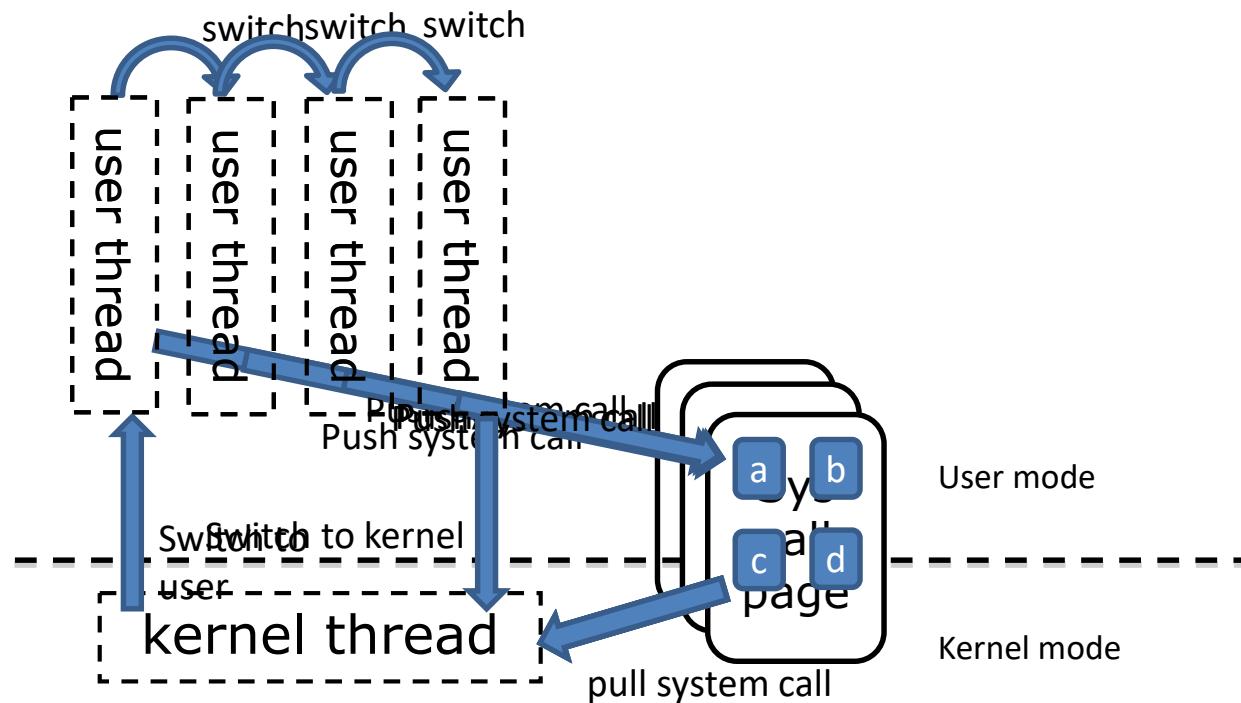
while (entry->status != DONE)
    do_something_else();

return entry->return_code;
```

On a Single Core: Single Threads



On a Single Core: Multiple Threads



FlexSC: Flexible System Call Scheduling with Exception-Less System Calls

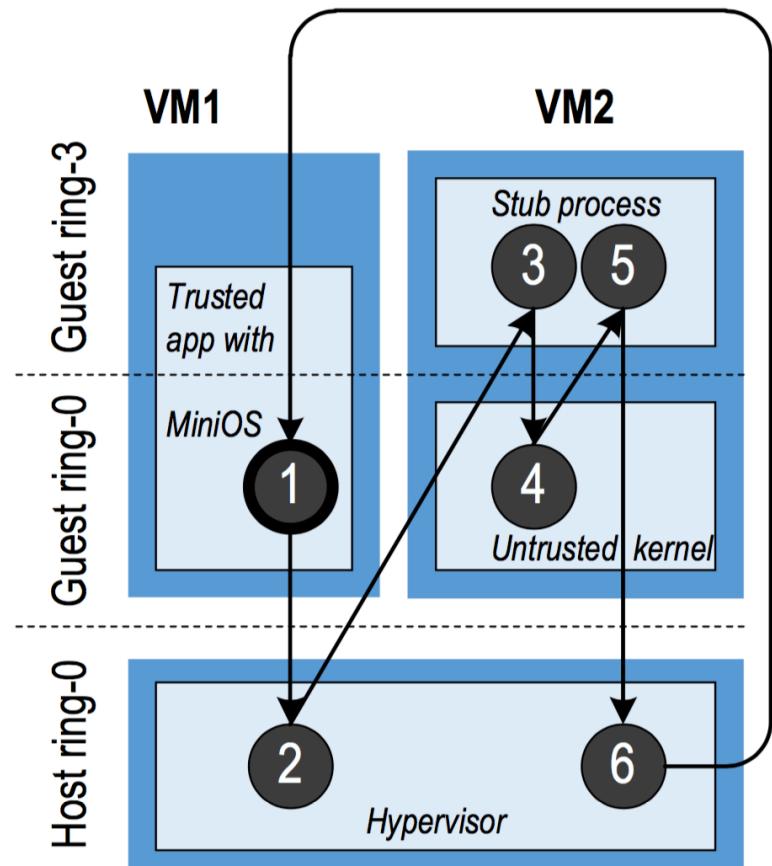
SUMMARY

Summary

- System call VS. function call
 - Trap! Switching the mode
 - State saving/restoring, changing stack, checking...
- ‘Int 0x80’ is not enough
 - New instructions for lower latency
 - vDSO for reducing trapping
 - FlexSC for further reducing trapping by multi-core

Cross-Mode Calls

- ProxOS
 - Two virtual machines, one is trusted and the other is untrusted
 - Redirect untrusted system calls to the untrusted VMs
 - Overhead of mode switching could be high!
- More
 - Nested Virtualization



Optimization for Cross-Mode Calls

- Using shared memory (like FlexSC)
 - Make synchronous calls to asynchronous ones
 - Require app logic to be changed
- How about a new instruction for world-call?
 - A world is just a mode (including VM, kernel/user, etc.)
 - Separating protection from managing
 - Configure by software (managing)
 - Check by hardware (protection)



Zhaoguo Wang
IPADS, SJTU

ACKs: Some slides are adapted from the textbook's original slides
and frans's os course notes

Review: What Can't You Do in an Interrupt Handler?

- You cannot sleep
 - or call something that *might* sleep
- You cannot refer to `current`
- You cannot allocate memory with `GPF_KERNEL` (which can sleep), you must use `GPF_ATOMIC` (which can fail)
- You cannot call `schedule()`
- You cannot do a `down()` semaphore call
 - However, you *can* do an `up()`
- You cannot transfer data to/from user space
 - E.g., `copy_to_user()`, `copy_from_user()`

Review

- FlexSC: Exception-less syscall: remove synchronicity by decoupling invocation from execution
- Why I/O sub-system: Thousands of devices, each slightly different
- Unified interface: file
 - Open, read, write, seek, close, ...
 - Avoid too generalization
- Top-half & bottom-half

- The two main jobs of a computer are I/O and processing. In many cases, the main job is I/O, and the processing is merely incidental.

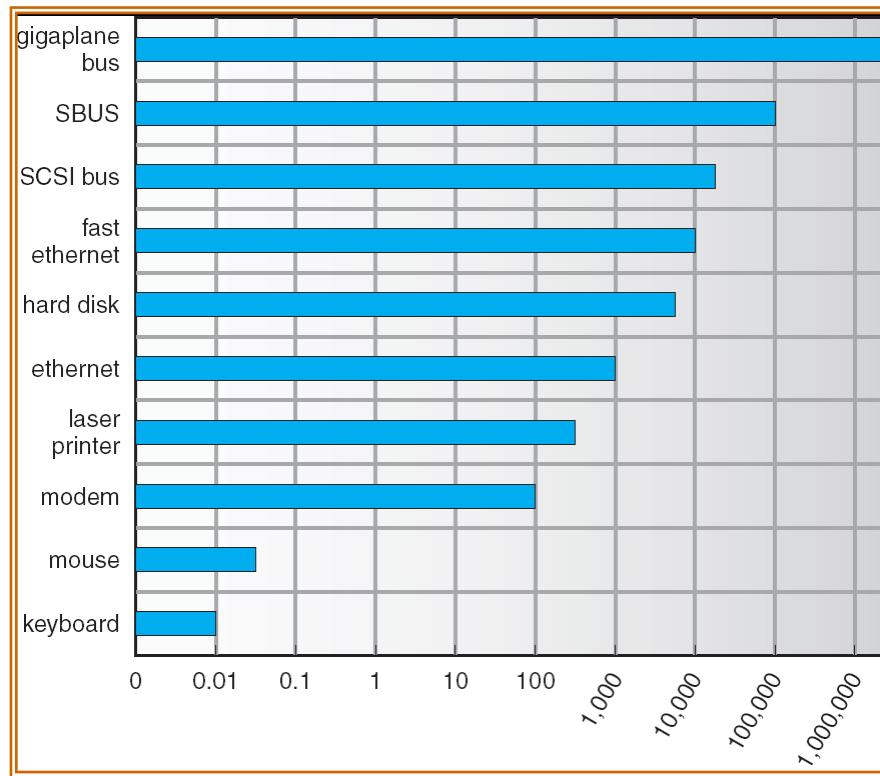
— Operating System Concepts, 8th edition

I/O

Why do we Need an I/O Subsystem

- Thousands of devices, each slightly different
 - How can we standardize the interfaces to these devices?
- Devices are **unreliable**
 - Media failures and transmission errors
 - How can we make them reliable?
- Devices are **unpredictable** and/or **slow**
 - How can we manage them if we don't know what they will do or how they will perform?

Device Rates Vary Many Orders of Magnitude



- I/O subsystem better handle this wide range
 - Better not have high overhead/byte for fast devices!
 - Better not waste time waiting for slow devices

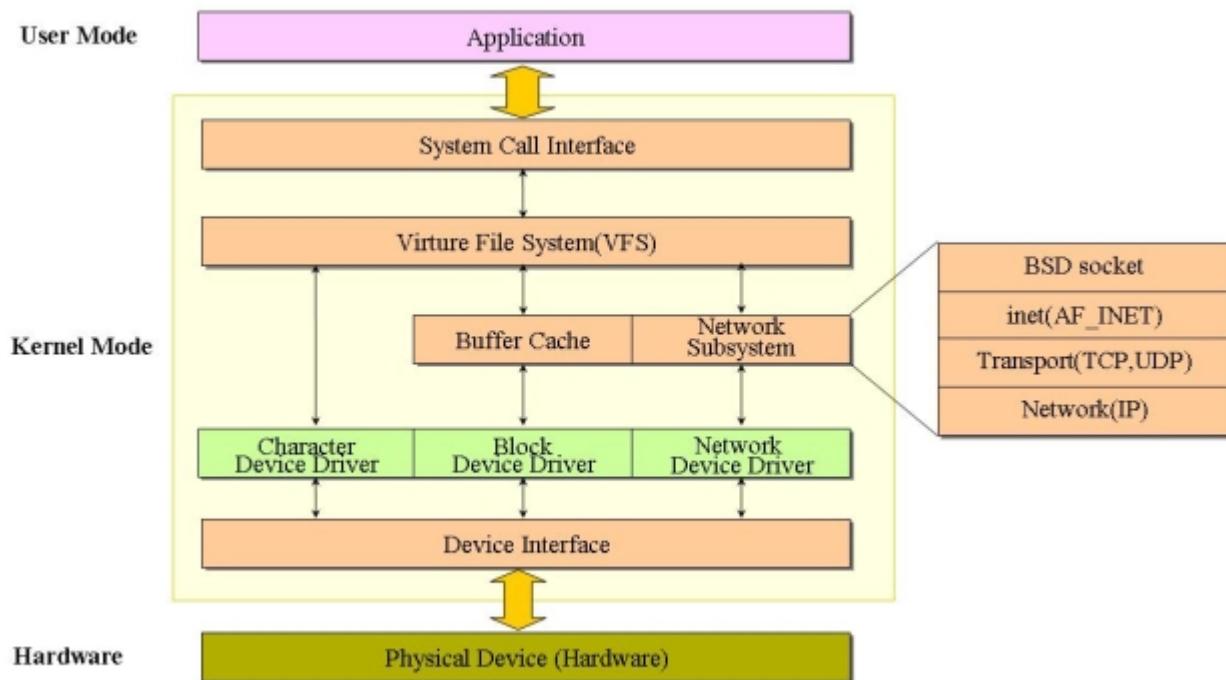
Goal of I/O Subsystem

- Provides uniform interfaces, despite wide range of different devices
 - This code works on many different devices:

```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
    fprintf(fd, "Count %d\n", i);
}
close(fd);
```
 - Why?
 - Because code that controls **devices** (“**device driver**”) implements standard interface.
- Provides a layer of abstraction of I/O hardware
 - Manages and interacts with hardware
 - Hides hardware and operation details

Three Types of Device Interfaces

- Three common types of device interfaces
 - Character devices
 - Block devices
 - Network devices



Character Devices

- Character Devices
 - Example: keyboard/mouse, serial port, some USB devices
 - Sequential access, single characters at a time
 - I/O commands: get(), put(), etc.
 - Often use open file interface and semantics

Press 'a' to display 'a'

```
3 .globl alltraps
4 .globl vector0
5 vector0:
6     pushl $0
7     pushl $0
8     jmp alltraps
9 .globl vector1
10 vector1:
11    pushl $0
12    pushl $1
13    jmp alltraps
14 .globl vector2
15 vector2:
16    pushl $0
17    pushl $2
18    jmp alltraps
19 .globl vector3
20 vector3:
21    pushl $0
22    pushl $3
23    jmp alltraps
```

```
3 # vectors.S sends all traps here.
4 .globl alltraps
5 alltraps:
6     # Build trap frame.
7     pushl %ds
8     pushl %es
9     pushl %fs
10    pushl %gs
11    pushal
12
13    # Set up data and per-cpu segments.
14    movw $(SEG_KDATA<<3), %ax
15    movw %ax, %ds
16    movw %ax, %es
17    movw $(SEG_KCPU<<3), %ax
18    movw %ax, %fs
19    movw %ax, %gs
20
21    # Call trap(tf), where tf=%esp
22    pushl %esp
23    call trap
24    addl $4, %esp
25
26    # Return falls through to trapret...
27 .globl trapret
28 trapret:
29    popal
30    popl %gs
31    popl %fs
32    popl %es
33    popl %ds
34    addl $0x8, %esp # trapno and errcode
35    iret
```

```
36 void
37 trap(struct trapframe *tf)
38 {
39     if(tf->trapno == T_SYSCALL){
40         if(proc->killed)
41             exit();
42         proc->tf = tf;
43         syscall();
44         if(proc->killed)
45             exit();
46         return;
47     }
48
49     switch(tf->trapno){
50     case T_IRQ0 + IRQ_TIMER:
51         if(cpu->id == 0){
52             acquire(&tickslock);
53             ticks++;
54             wakeup(&ticks);
55             release(&tickslock);
56         }
57         lapiceoi();
58         break;
59     case T_IRQ0 + IRQ_IDE:
60         ideintr();
61         lapiceoi();
62         break;
63     case T_IRQ0 + IRQ_IDE+1:
64         // Bochs generates spurious IDE1 interrupts.
65         break;
66     case T_IRQ0 + IRQ_KBD:
67         kbdintr();
68         lapiceoi();
69         break;
70     case T_IRQ0 + IRQ_COM1:
```

```
46 void
47 kbdintr(void)
48 {
49     consoleintr(kbdgetc());
50 }
```

Function pointer

```
187 void
188 consoleintr(int (*getc)(void))
189 {
190     int c;
191
192     acquire(&input.lock);
193     while((c = getc()) >= 0){
194         switch(c){
195             case C('P'): // Process listing.
196                 procdump();
197                 break;
198             case C('U'): // Kill line.
199                 while(input.e != input.w &&
200                     input.buf[(input.e-1) % INPUT_BUF] != '\n'){
201                     input.e--;
202                     consputc(BACKSPACE);
203                 }
204                 break;
205             case C('H'): case '\x7f': // Backspace
206                 if(input.e != input.w){
207                     input.e--;
208                     consputc(BACKSPACE);
209                 }
210                 break;
211             default:
212                 if(c != 0 && input.e-input.r < INPUT_BUF){
213                     c = (c == '\r') ? '\n' : c;
214                     input.buf[input.e++ % INPUT_BUF] = c;
215                     consputc(c);
216                     if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
217                         input.w = input.e;
218                         wakeup(&input.r);
219                     }
220                 }
221                 break;
222             }
223         }
224     release(&input.lock);
225 }
```

kbdgetc()

#define C(x) ((x)-'@') // Control-x

```
6 int
7 kbdgetc(void)
8 {
9     static uint shift;
10    static uchar *charcode[4] = {
11        normalmap, shiftmap, ctlmap, ctlmap
12    };
13    uint st, data, c;
14
15    st = inb(KBSTATP);
16    if((st & KBS_DIB) == 0)
17        return -1;
18    data = inb(KBDATAP);
19
20    if(data == 0xE0){
21        shift |= E0ESC;
22        return 0;
23    } else if(data & 0x80){
24        // Key released
25        data = (shift & E0ESC ? data : data & 0x7F);
26        shift &= ~shiftcode[data] | E0ESC;
27        return 0;
28    } else if(shift & E0ESC){
29        // Last character was an E0 escape; or with 0x80
30        data |= 0x80;
31        shift &= ~E0ESC;
32    }
33
34    shift |= shiftcode[data];
35    shift ^= togglecode[data];
36    c = charcode[shift & (CTL | SHIFT)][data];
37    if(shift & CAPSLOCK){
38        if('a' <= c && c <= 'z')
39            c += 'A' - 'a';
40        else if('A' <= c && c <= 'Z')
41            c += 'a' - 'A';
42    }
43    return c;
44 }
```

```
3 #define KBSTATP          0x64 // kbd controller status port(I)
4 #define KBS_DIB           0x01 // kbd data in buffer
5 #define KBDATAP           0x60 // kbd data port(I)
6
7 #define NO                0
8
9 #define SHIFT             (1<<0)
10 #define CTL               (1<<1)
11 #define ALT               (1<<2)
12
13 #define CAPSLOCK          (1<<3)
14 #define NUMLOCK           (1<<4)
15 #define SCROLLLOCK        (1<<5)
16
17 #define E0ESC              (1<<6)
18
19 // Special keycodes
20 #define KEY_HOME          0xE0
21 #define KEY_END            0xE1
22 #define KEY_UP             0xE2
23 #define KEY_DN             0xE3
24 #define KEY_LF             0xE4
25 #define KEY_RT             0xE5
26 #define KEY_PGUP           0xE6
27 #define KEY_PGDN           0xE7
28 #define KEY_INS            0xE8
29 #define KEY_DEL            0xE9
```

```
160 void  
161 consputc(int c)  
162 {  
163     if(panicked){  
164         cli();  
165         for(;;)  
166             ;  
167     }  
168  
169     if(c == BACKSPACE){  
170         uartputc('\b'); uartputc(' '); uartputc('\b');  
171     } else  
172         uartputc(c);  
173     cgaputc(c);  
174 }
```

```
53 void  
54 cprintf(char *fmt, ...)  
55 {  
56     int i, c, locking;  
57     uint *argp;  
58     char *s;  
59  
60     locking = cons.locking;  
61     if(locking)  
62         acquire(&cons.lock);  
63  
64     if(fmt == 0)  
65         panic("null fmt");  
66  
67     argp = (uint*)(void*)(&fmt + 1);  
68     for(i = 0; (c = fmt[i] & 0xff) != 0; i++){  
69         if(c != '%'){  
70             consputc(c);  
71             continue;  
72         }
```

```
129 static void  
130 cgaputc(int c)  
131 {  
132     int pos;  
133  
134     // Cursor position: col + 80*row.  
135     outb(CRTPORT, 14);  
136     pos = inb(CRTPORT+1) << 8;  
137     outb(CRTPORT, 15);  
138     pos |= inb(CRTPORT+1);  
139  
140     if(c == '\n')  
141         pos += 80 - pos%80;  
142     else if(c == BACKSPACE){  
143         if(pos > 0) --pos;  
144     } else  
145         crt[pos++] = (c&0xff) | 0x0700; // black on white  
146  
147     if((pos/80) >= 24){ // Scroll up.  
148         memmove(crt, crt+80, sizeof(crt[0])*23*80);  
149         pos -= 80;  
150         memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));  
151     }  
152  
153     outb(CRTPORT, 14);  
154     outb(CRTPORT+1, pos>>8);  
155     outb(CRTPORT, 15);  
156     outb(CRTPORT+1, pos);  
157     crt[pos] = ' ' | 0x0700;  
158 }
```

I/O instruction in xv6

```
3 static inline uchar
4 inb(ushort port)
5 {
6     uchar data;
7
8     asm volatile("in %1,%0" : "=a" (data) : "d" (port));
9     return data;
10 }
```

```
21 static inline void
22 outb(ushort port, uchar data)
23 {
24     asm volatile("out %0,%1" : : "a" (data), "d" (port));
25 }
```

```
160 void  
161 consputc(int c)  
162 {  
163     if(panicked){  
164         cli();  
165         for(;;)  
166             ;  
167     }  
168  
169     if(c == BACKSPACE){  
170         uartputc('\b'); uartputc(' '); uartputc('\b');  
171     } else  
172         uartputc(c);  
173     cgaputc(c);  
174 }
```

```
51 void  
52 uartputc(int c)  
53 {  
54     int i;  
55  
56     if(!uart)  
57         return;  
58     for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)  
59         microdelay(10);  
60     outb(COM1+0, c);  
61 }
```

Block Devices

- Block Devices
 - Example: disk drive, tape drive, DVD-ROM
 - Uniform block I/O interface to access blocks of data
 - Raw I/O or file-system access
 - Memory-mapped file access possible

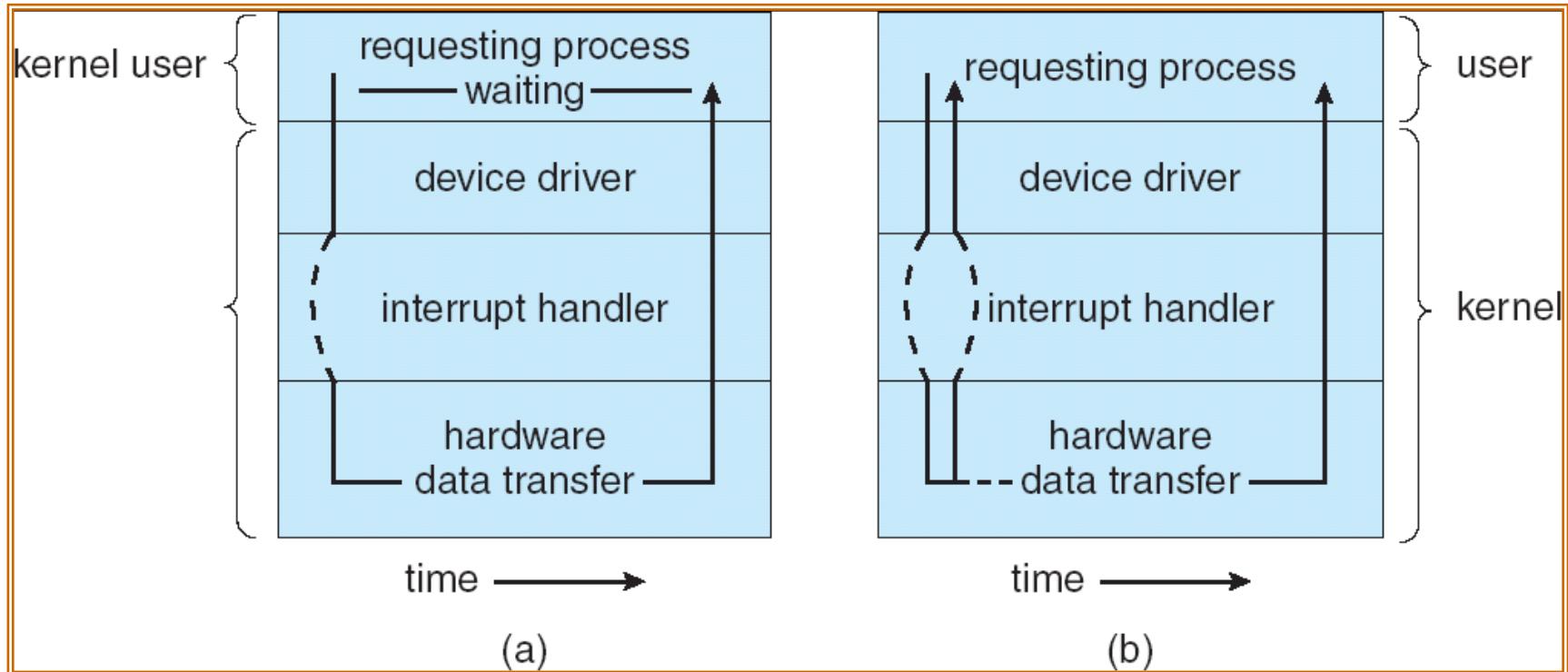
Network Devices

- Network Devices
 - Examples: Ethernet, wireless, bluetooth
 - Different enough from block/character to have own interface
 - Provide special networking interface for supporting various network protocols
 - For example, send/receive network packets

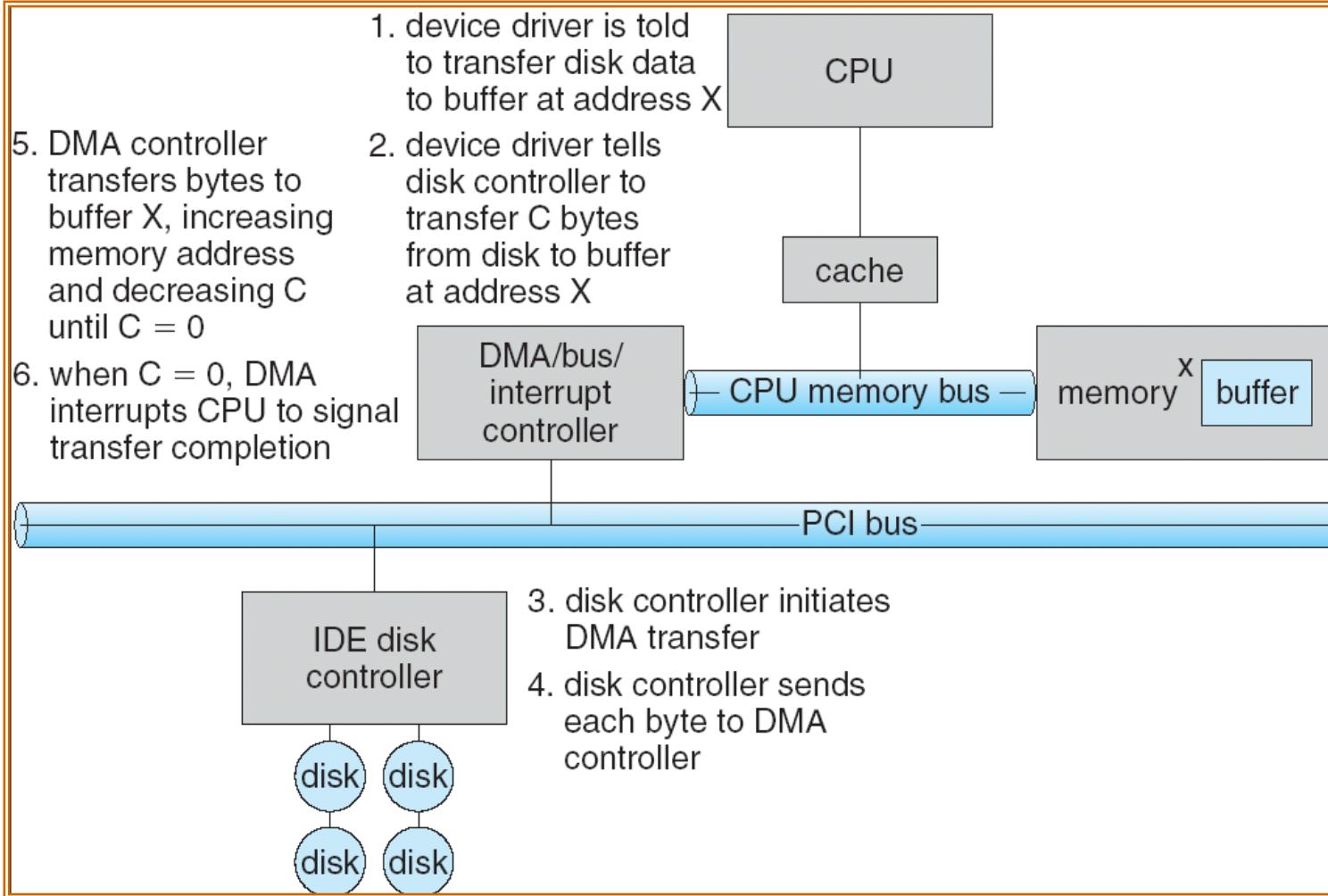
How Does User Deal with Timing?

- **Blocking I/O:** “Wait”
 - When request data (e.g. `read()` system call), put process in waiting state until data is ready
 - When write data (e.g. `write()` system call), put process in waiting state until device is ready for data
- **Non-blocking I/O:** “Don’t Wait”
 - Returns immediately from read or write request with count of bytes successfully transferred
 - Read may return nothing, write may write nothing
- **Asynchronous I/O:** “Tell Me Later”
 - When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
 - When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user

Synchronous and Asynchronous I/O



Steps in a DMA Transfer



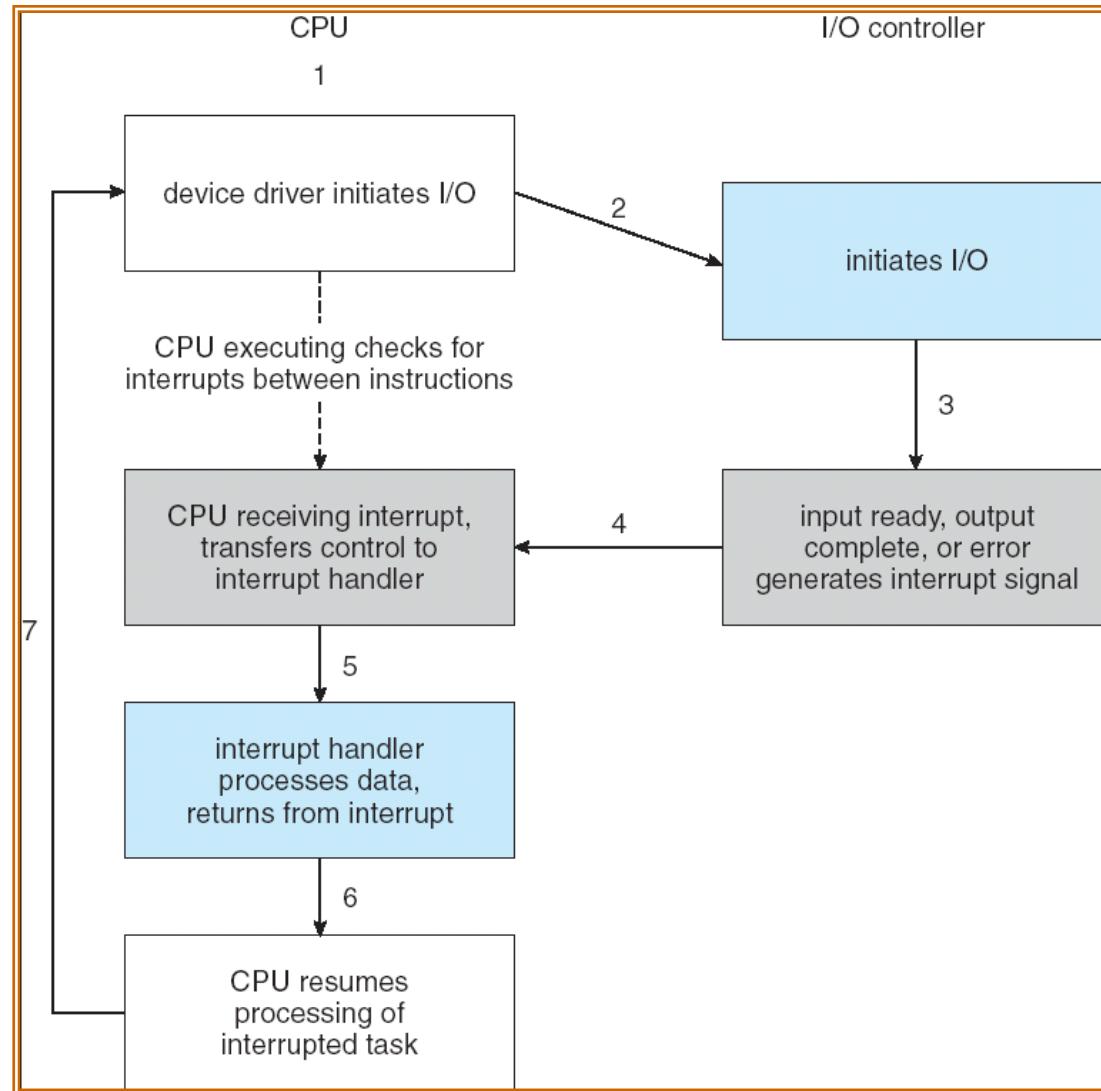
I/O Device Notifying the OS

- The OS needs to know when:
 - The I/O device has completed an operation
 - The I/O operation has encountered an error
- Two methods
 - Polling
 - Interrupt-driven
- Polling:
 - I/O device puts completion/error information in device-specific status register
 - OS periodically checks the status register
 - Pro: low overhead
 - Con: may waste many cycles on polling if infrequent or unpredictable I/O operations

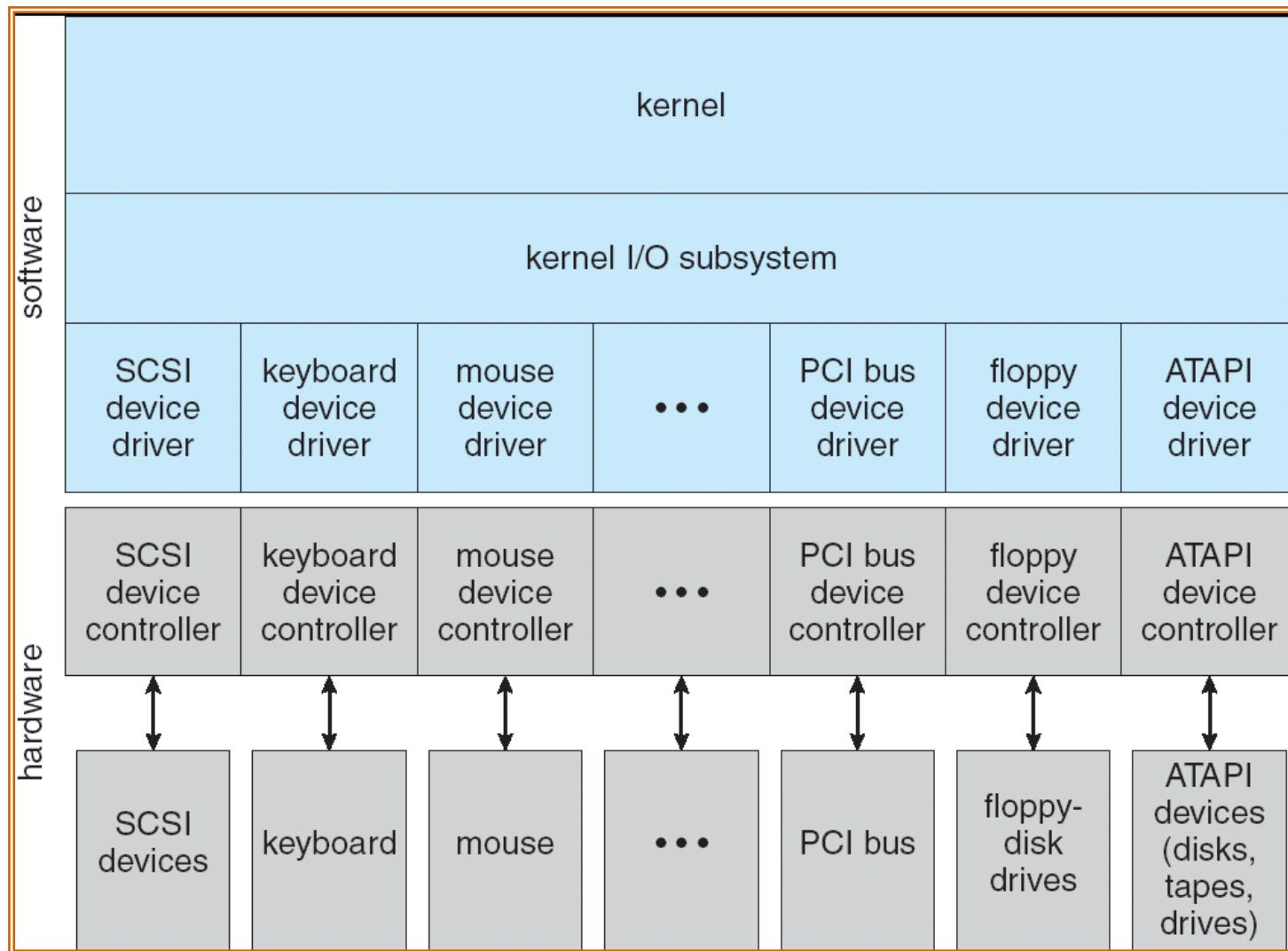
Interrupt Driven

- Pro: handles unpredictable events well
- Con: interrupts relatively high overhead
- Some devices may combine both polling and interrupt-driven
 - High-bandwidth network device example:
 - interrupt for first incoming packet
 - polling for following packets until hardware empty

Interrupt-Driven I/O Cycle



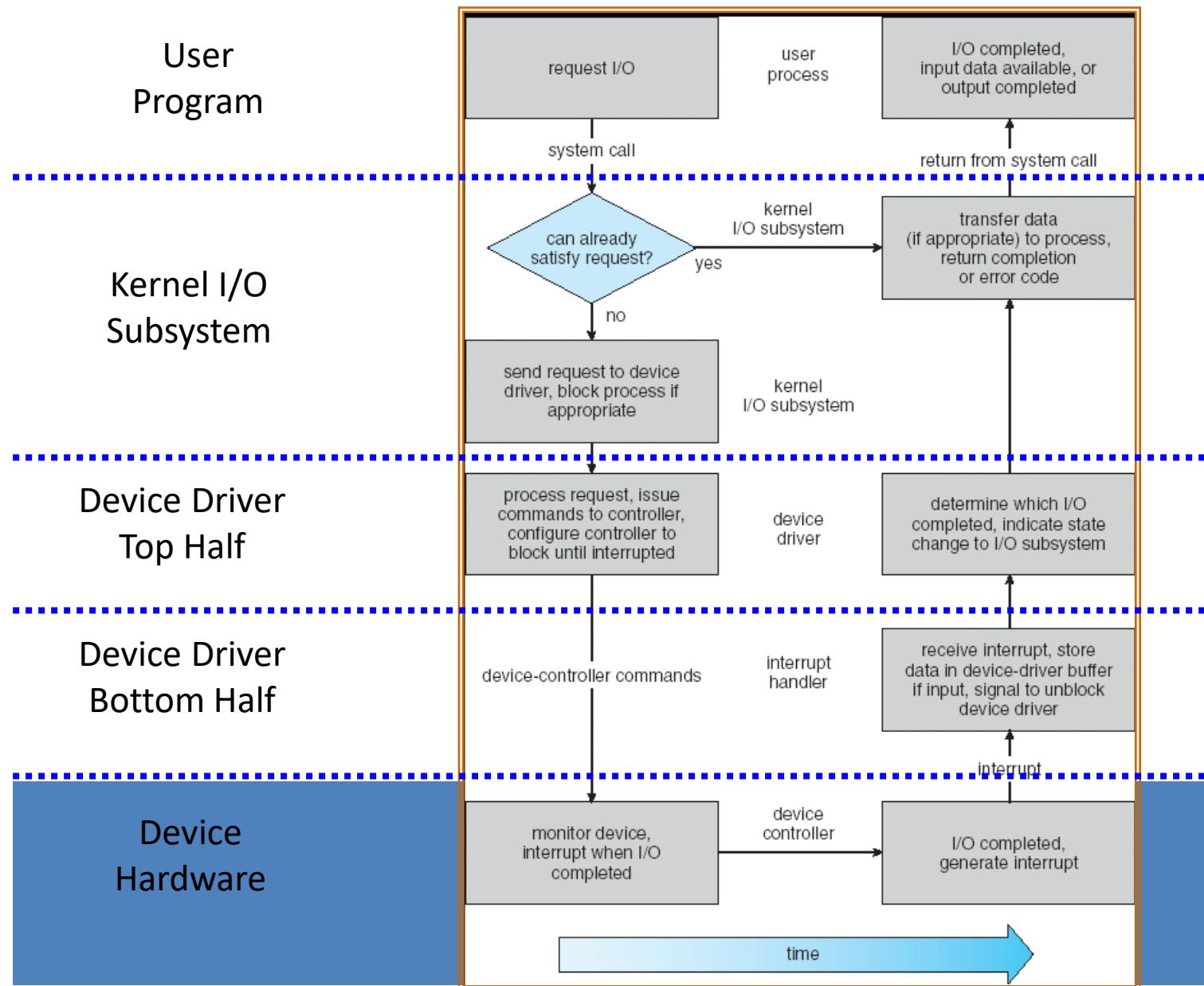
A Kernel I/O Structure



Device Drivers

- Device-specific code in the kernel that interacts directly with the device hardware
 - Supports a standard internal interface
 - Same kernel I/O system can interact easily with different device drivers
 - Special device-specific configuration supported with the ioctl() system call
- Device drivers typically divided into two pieces:
 - Top half
 - Bottom half

Life Cycle of An I/O Request



Recall : Read Disk

- readi()->bread()
 - Call bget() to read from buffer cache first
 - If not found, call iderw() to read from disk, and mark as valid.

```
95 // Return a B_BUSY buf with the contents of the indicated disk sector.
96 struct buf*
97 bread(uint dev, uint sector)
98 {
99     struct buf *b;
100
101     b = bget(dev, sector);
102     if(!(b->flags & B_VALID))
103         iderw(b);
104     return b;
105 }
```

Xv6 disk driver design & Implementation

- Design Overview
 - Support an IDE driver
 - Use I/O Instructions instead of Memory Mapped I/O
 - in, out
 - Asynchronous I/O model
 - Use a simple **queue** of I/O request
 - Use **interrupt** to notify the available of data

IDE Specification

- Xv6 uses a relative old IDE specification
 - Not PCI negotiation
- IDE Ports
 - 0x1F0-0x1F7
 - 0x1f0: write/read port
 - 0x1f2: number of sectors
 - 0x1f3-0x1f5: sector number
 - 0x1f6: diskno and sector number
 - 0x1f7: command registers, status bit
 - 0x3F6 interrupt control line

IDE Initialization

```
45 void
46 ideinit(void)
47 {
48     int i;
49
50     initlock(&idelock, "ide");
51     picenable(IRQ_IDE);
52     ioapicenable(IRQ_IDE, ncpu - 1);
53     idewait(0);
54
55     // Check if disk 1 is present
56     outb(0x1f6, 0xe0 | (1<<4));
57     for(i=0; i<1000; i++){
58         if(inb(0x1f7) != 0){
59             havedisk1 = 1;
60             break;
61         }
62     }
63
64     // Switch back to disk 0.
65     outb(0x1f6, 0xe0 | (0<<4));
66 }
```

Polling Mechanism

```
32 // Wait for IDE disk to become ready.
33 static int
34 idewait(int checkerr)
35 {
36     int r;
37
38     while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
39         ;
40     if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
41         return -1;
42     return 0;
43 }
```

Synchronous I/O (iderw)

```
122 // Sync buf with disk.  
123 // If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.  
124 // Else if B_VALID is not set, read buf from disk, set B_VALID.  
125 void  
126 iderw(struct buf *b)  
127 {  
128     struct buf **pp;  
129  
130     if(!(b->flags & B_BUSY))  
131         panic("iderw: buf not busy");  
132     if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)  
133         panic("iderw: nothing to do");  
134     if(b->dev != 0 && !havedisk1)  
135         panic("iderw: ide disk 1 not present");  
136  
137     acquire(&idelock); //DOC:acquire-lock  
138  
139     // Append b to idequeue.  
140     b->qnext = 0;  
141     for(pp=&idequeue; *pp; pp=&(*pp)->qnext) //DOC:insert-queue  
142         ;  
143     *pp = b;  
144  
145     // Start disk if necessary.  
146     if(idequeue == b)  
147         idestart(b);  
148  
149     // Wait for request to finish.  
150     while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){  
151         sleep(b, &idelock);  
152     }  
153  
154     release(&idelock);  
155 }
```

```
68 // Start the request for b. Caller must hold idelock.
69 static void
70 idestart(struct buf *b)
71 {
72     if(b == 0)
73         panic("idestart");
74
75     idewait(0);
76     outb(0x3f6, 0); // generate interrupt
77     outb(0x1f2, 1); // number of sectors
78     outb(0x1f3, b->sector & 0xff);
79     outb(0x1f4, (b->sector >> 8) & 0xff);
80     outb(0x1f5, (b->sector >> 16) & 0xff);
81     outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
82     if(b->flags & B_DIRTY){
83         outb(0x1f7, IDE_CMD_WRITE);
84         outsl(0x1f0, b->data, 512/4);
85     } else {
86         outb(0x1f7, IDE_CMD_READ);
87     }
88 }
```

```
90 // Interrupt handler.
91 void
92 ideintr(void)
93 {
94     struct buf *b;
95
96     // First queued buffer is the active request.
97     acquire(&idelock);
98     if((b = idequeue) == 0){
99         release(&idelock);
100        // cprintf("spurious IDE interrupt\n");
101        return;
102    }
103    idequeue = b->qnext;
104
105    // Read data if needed.
106    if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
107        insl(0x1f0, b->data, 512/4);
108
109    // Wake process waiting for this buf.
110    b->flags |= B_VALID;
111    b->flags &= ~B_DIRTY;
112    wakeup(b);
113
114    // Start disk on next buf in queue.
115    if(idequeue != 0)
116        idestart(idequeue);
117
118    release(&idelock);
119 }
```

```
12 static inline void
13 insl(int port, void *addr, int cnt)
14 {
15     asm volatile("cld; rep insl" :
16                 "=D" (addr), "=c" (cnt) :
17                 "d" (port), "0" (addr), "1" (cnt) :
18                 "memory", "cc");
19 }
```

```
340 // Atomically release lock and sleep on chan.
341 // Reacquires lock when awakened.
342 void
343 sleep(void *chan, struct spinlock *lk)
344 {
345     if(proc == 0)
346         panic("sleep");
347
348     if(lk == 0)
349         panic("sleep without lk");
350
351     // Must acquire ptable.lock in order to
352     // change p->state and then call sched.
353     // Once we hold ptable.lock, we can be
354     // guaranteed that we won't miss any wakeup
355     // (wakeup runs with ptable.lock locked),
356     // so it's okay to release lk.
357     if(lk != &ptable.lock){ //DOC: sleeplock0
358         acquire(&ptable.lock); //DOC: sleeplock1
359         release(lk);
360     }
361
362     // Go to sleep.
363     proc->chan = chan;
364     proc->state = SLEEPING;
365     sched();
366
367     // Tidy up.
368     proc->chan = 0;
369
370     // Reacquire original lock.
371     if(lk != &ptable.lock){ //DOC: sleeplock2
372         release(&ptable.lock);
373         acquire(lk);
374     }
375 }
```

Sleep and Wakeup Mechanism in xv6

```
378 // Wake up all processes sleeping on chan.
379 // The ptable lock must be held.
380 static void
381 wakeup1(void *chan)
382 {
383     struct proc *p;
384
385     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
386         if(p->state == SLEEPING && p->chan == chan)
387             p->state = RUNNABLE;
388 }
389
390 // Wake up all processes sleeping on chan.
391 void
392 wakeup(void *chan)
393 {
394     acquire(&ptable.lock);
395     wakeup1(chan);
396     release(&ptable.lock);
397 }
```

IDE.c

- The driver can only handle 1 operation at a time
 - Only send the buffer at the front of the queue
 - Others are simply waiting their turn
- ide rw() maintains a queue of requests
 - Adds the buffer b to the end of the queue
 - If b is the first, then call ide start(), if not, just wait
 - Then just sleep, instead of polling
- ide intr() will handle the first request
 - Then start other request if any

Homework

- Please read xv6 code and describe the process of helloworld (from “./hello” to the exit)

File System: XV6 ~ Ext4

Yubin Xia

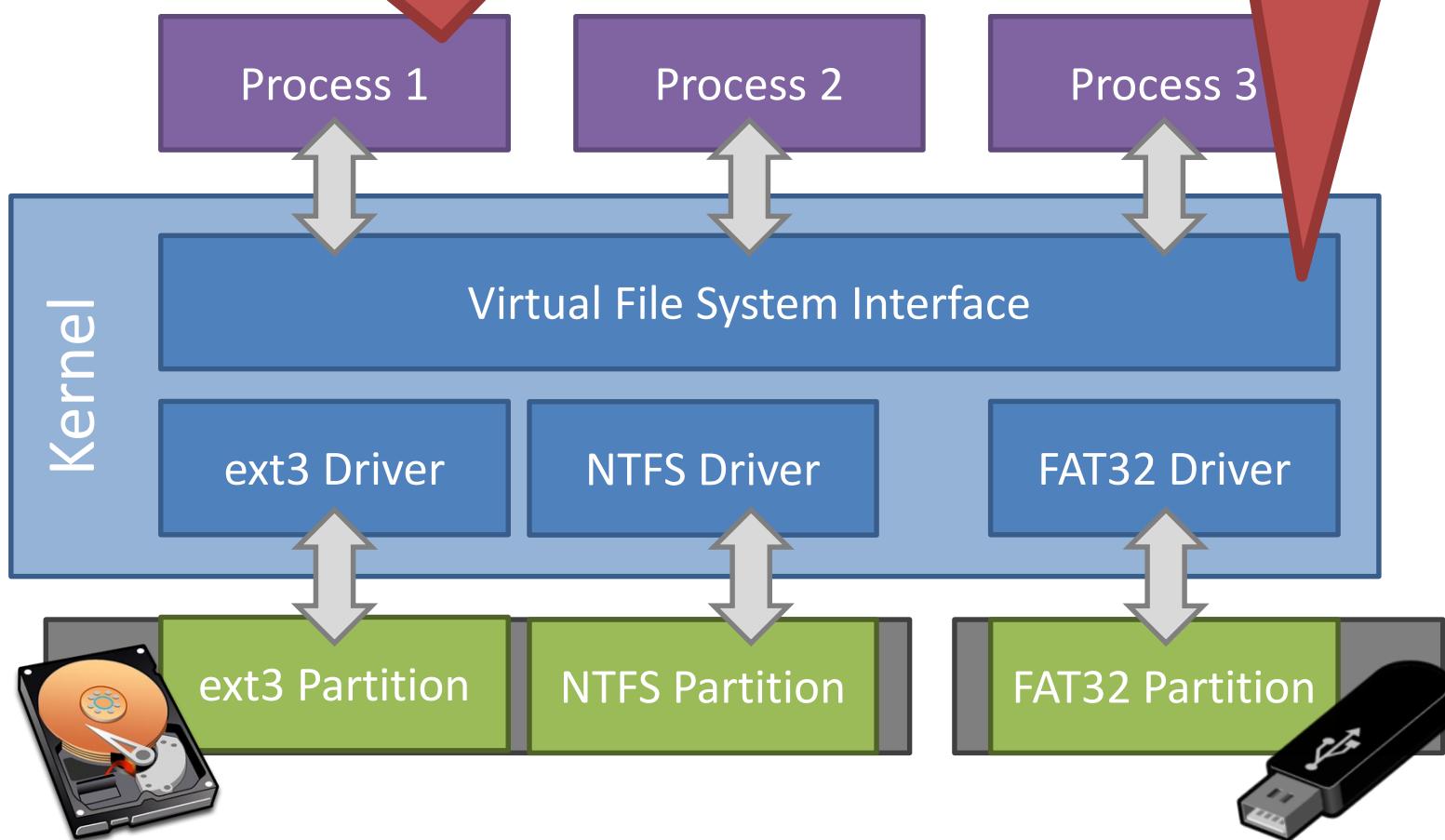
Virtual File System Interface

- Problem: the OS may mount several partitions containing different underlying file systems
 - It would be bad if processes had to use different APIs for different file systems
- Linux uses a Virtual File System interface (VFS)
 - Exposes POSIX APIs to processes
 - Forwards requests to lower-level file system specific drivers
- Windows uses a similar system

VFS Flowchart

Processes (usually) don't need to know about low-level file system details

Relatively simple to add additional file system drivers



Xv6 FS Design

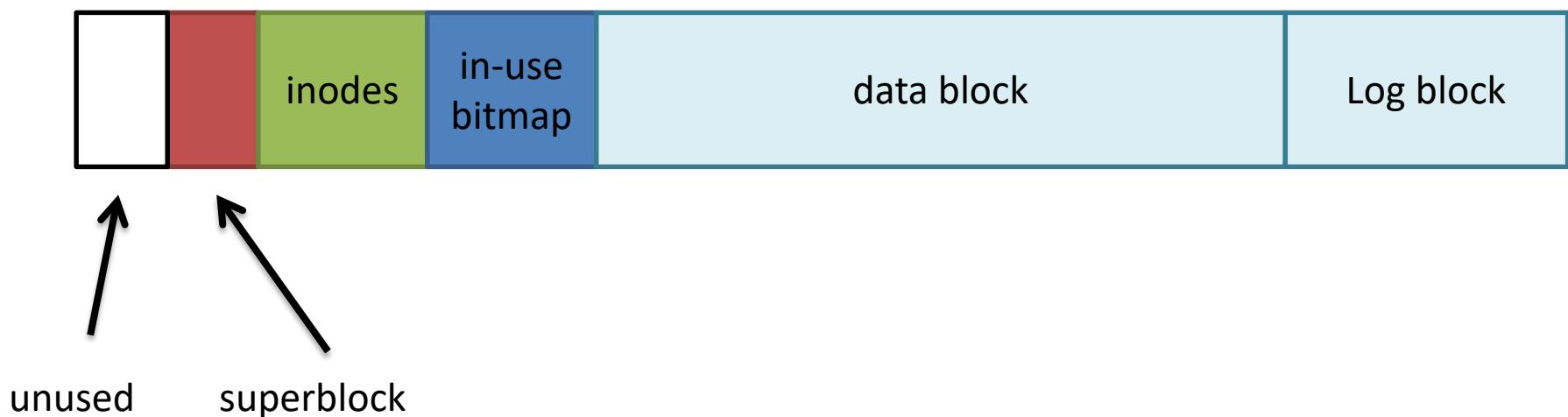
- Implements a minimal Unix file system interface
 - Superblock
 - Inode
 - Dentry
- Limitations
 - Not pay attention to file system layout
 - No disk scheduling
 - Its cache is write-through
 - simplifies keeping disk data structures consistent
 - bad for performance.

xv6 File System

- No disk scheduling
- Write-through
 - Simplifies keeping on disk data structures consistent
 - Bad for performance
- Block size: 512 bytes (*BSIZE*)

Disk Layout

- Block 0: unused
- Block 1: superblock
- Block 2+
 - inode ($\text{ninodes} / \text{inodes_per_block}$)
 - in-use bitmap
 - data block
 - Log block



Recap: Free-Space List Bit Vector

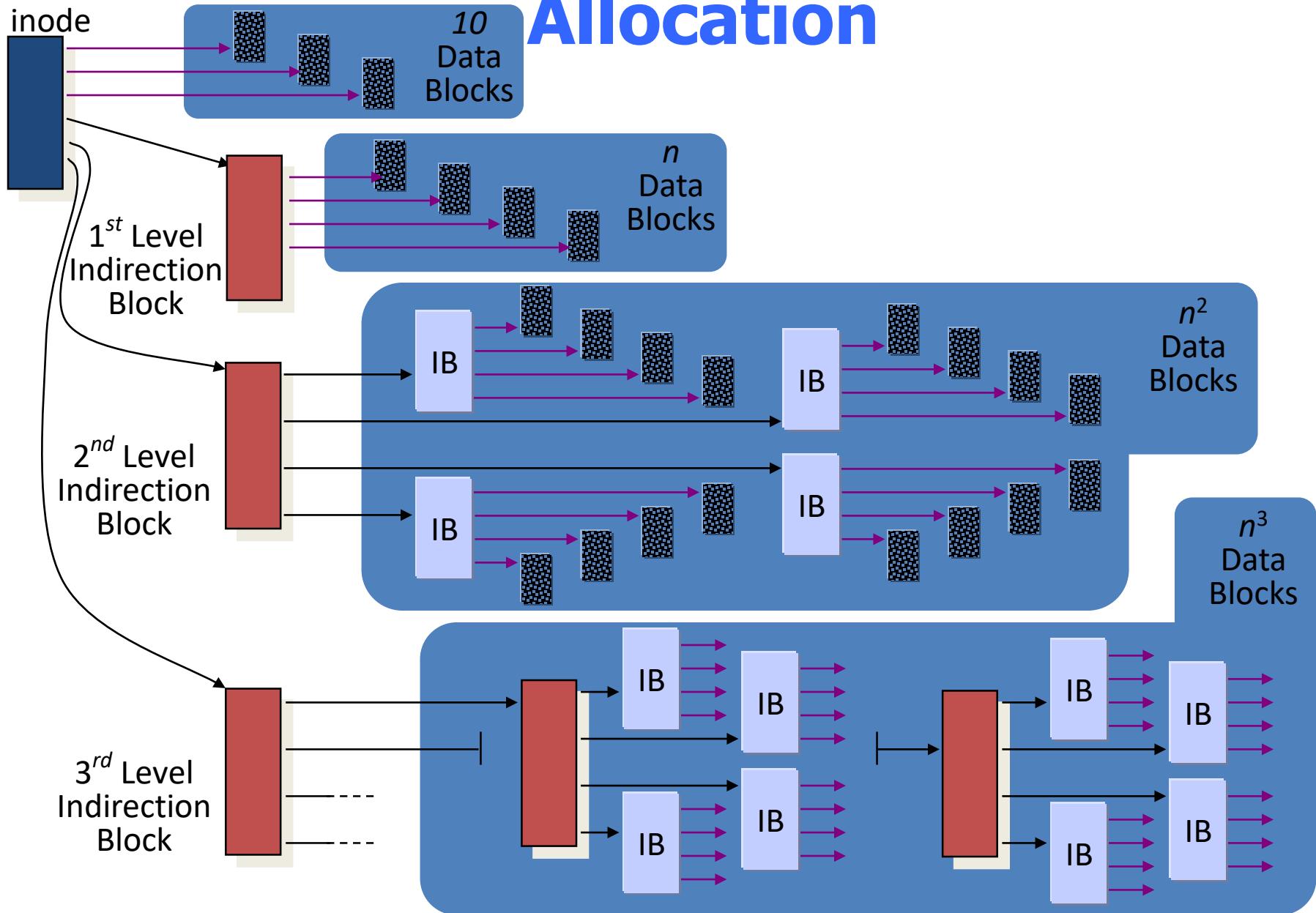
- Represent the list of free blocks as a bit vector:
 - 11111111111111001101010110111...
 - If bit $i = 0$ then block i is free, otherwise it is allocated
- Simple to use but this can be a big vector:
 - 2TB disk \rightarrow 512M blocks \rightarrow 64MB bits
 - If a disk is 90% full, then the average number of bits to be scanned is 10, independent of the size of the disk

Managing Disk Blocks

- Use a bitmap to maintain availability of disk blocks

```
51 // Allocate a zeroed disk block.
52 static uint
53 balloc(uint dev)
54 {
55     int b, bi, m;
56     struct buf *bp;
57     struct superblock sb;
58
59     bp = 0;
60     readsb(dev, &sb);
61     for(b = 0; b < sb.size; b += BPB){
62         bp = bread(dev, BBLOCK(b, sb.ninodes));
63         for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
64             m = 1 << (bi % 8);
65             if((bp->data[bi/8] & m) == 0){ // Is block free?
66                 bp->data[bi/8] |= m; // Mark block in use.
67                 log_write(bp);
68                 brelse(bp);
69                 bzero(dev, b + bi);
70                 return b + bi;
71             }
72         }
73         brelse(bp);
74     }
75     panic("balloc: out of blocks");
76 }
```

Recap: Multi-level Indexed Allocation



Files and inode

- Represented by an **inode**

```
[fs.h]
#define NADDRS      (NDIRECT+1)
#define NDIRECT     12
#define INDIRECT    12
#define NINDIRECT   (BSIZE / sizeof(uint))
#define MAXFILE     (NDIRECT + NINDIRECT)

// On-disk inode structure
struct dinode {
    short type;          // File type
    short major;         // Major device number (T_DEV only)
    short minor;         // Minor device number (T_DEV only)
    short nlink;         // Number of links to inode in file system
    uint size;           // Size of file (bytes)
    uint addrs[NDIRECT+1]; // Data block addresses
};
```

Files (cont.)

- File size
 - Small: up to 12 block addresses ($NADDRS - 1$)
 - Large: Last address in the inode as a disk address for a block with 128 disk addresses ($512 / 4$)
 - At most $12 * 512 + 128 * 512$ bytes

Directories

- Directories are special files
 - Files with a bit of structure to them
 - Contains multiple **dirent** structures

```
// Directory is a file containing a sequence of dirent structures.  
#define DIRSIZ 14  
  
struct dirent {  
    ushort inum;  
    char name[DIRSIZ];  
};
```

Allocating an inode

```
173 // Allocate a new inode with the given type on device dev.  
174 // A free inode has a type of zero.  
175 struct inode*  
176 ialloc(uint dev, short type)  
177 {  
178     int inum;  
179     struct buf *bp;  
180     struct dinode *dip;  
181     struct superblock sb;  
182  
183     readsb(dev, &sb);  
184  
185     for(inum = 1; inum < sb.ninodes; inum++){  
186         bp = bread(dev, IBLLOCK(inum));  
187         dip = (struct dinode*)bp->data + inum%IPB;  
188         if(dip->type == 0){ // a free inode  
189             memset(dip, 0, sizeof(*dip));  
190             dip->type = type;  
191             log_write(bp); // mark it allocated on the disk  
192             brelse(bp);  
193             return igrab(dev, inum);  
194         }  
195         brelse(bp);  
196     }  
197     panic("ialloc: no inodes");  
198 }
```

Case Study: read(fd, buf, 100)

- sys_read()->fileread()->readi()
 - 1. If it's a special device file, use predefined operation
 - E.g., piperead() for pipe file
 - 2. call bread() to get file buffer

```
66 int
67 sys_read(void)
68 {
69     struct file *f;
70     int n;
71     char *p;
72
73     if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
74         return -1;
75     return fileread(f, p, n);
76 }
```

```
94 // Read from file f.
95 int
96 fileread(struct file *f, char *addr, int n)
97 {
98     int r;
99
100    if(f->readable == 0)
101        return -1;
102    if(f->type == FD_PIPE)
103        return piperead(f->pipe, addr, n);
104    if(f->type == FD_INODE){
105        ilock(f->ip);
106        if((r = readi(f->ip, addr, f->off, n)) > 0)
107            f->off += r;
108        iunlock(f->ip);
109        return r;
110    }
111    panic("fileread");
112 }
```

```
433 // Read data from inode.
434 int
435 readi(struct inode *ip, char *dst, uint off, uint n)
436 {
437     uint tot, m;
438     struct buf *bp;
439
440     if(ip->type == T_DEV){
441         if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
442             return -1;
443         return devsw[ip->major].read(ip, dst, n);
444     }
445
446     if(off > ip->size || off + n < off)
447         return -1;
448     if(off + n > ip->size)
449         n = ip->size - off;
450
451     for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
452         bp = bread(ip->dev, bmap(ip, off/BSIZE));
453         m = min(n - tot, BSIZE - off%BSIZE);
454         memmove(dst, bp->data + off%BSIZE, m);
455         brelse(bp);
456     }
457     return n;
458 }
```

Case Study: read (cont.)

- `readi()>bread()`
 - Call bget() to read from buffer cache first
 - If not found, call `ide_rw()` to read from disk, and mark as valid.

```
95 // Return a B_BUSY buf with the contents of the indicated disk sector.
96 struct buf*
97 bread(uint dev, uint sector)
98 {
99     struct buf *b;
100
101     b = bget(dev, sector);
102     if(!(b->flags & B_VALID))
103         iderw(b);
104     return b;
105 }
```

Case Study: read (cont.)

- `readi()->bread()->bget()`
 - Implements a simple buffer cache of recently-read disk blocks
 - 1. Look if the requested block is in the **cache**
 - 2. If some other process has locked the block, **wait** until it releases
 - 3. If it is not in the cache, find a cache entry to hold the block
 - 4. Mark it ours, but not valid

```
639 struct inode*
640 namei(char *path)
641 {
642     char name[DIRSIZ];
643     return namex(path, 0, name);
644 }
645
646 struct inode*
647 nameiparent(char *path, char *name)
648 {
649     return namex(path, 1, name);
650 }
```

```
601 // Look up and return the inode for a path name.  
602 // If parent != 0, return the inode for the parent and copy the final  
603 // path element into name, which must have room for DIRSIZ bytes.  
604 static struct inode*  
605 namex(char *path, int nameiparent, char *name)  
606 {  
607     struct inode *ip, *next;  
608  
609     if(*path == '/')  
610         ip = igit(ROOTDEV, ROOTINO);  
611     else  
612         ip = idup(proc->cwd);  
613  
614     while((path = skipellem(path, name)) != 0){  
615         ilock(ip);  
616         if(ip->type != T_DIR){  
617             iunlockput(ip);  
618             return 0;  
619         }  
620         if(nameiparent && *path == '\0'){  
621             // Stop one level early.  
622             iunlock(ip);  
623             return ip;  
624         }  
625         if((next = dirlookup(ip, name, 0)) == 0){  
626             iunlockput(ip);  
627             return 0;  
628         }  
629         iunlockput(ip);  
630         ip = next;  
631     }  
632     if(nameiparent){  
633         iput(ip);  
634         return 0;  
635     }  
636     return ip;  
637 }
```

```
570 // Examples:  
571 //    skipellem("a/bb/c", name) = "bb/c", setting name = "a"  
572 //    skipellem("//a//bb", name) = "bb", setting name = "a"  
573 //    skipellem("a", name) = "", setting name = "a"  
574 //    skipellem("", name) = skipellem("////", name) = 0
```

```
312 // Drop a reference to an in-memory inode.  
313 // If that was the last reference, the inode cache entry can  
314 // be recycled.  
315 // If that was the last reference and the inode has no links  
316 // to it, free the inode (and its content) on disk.  
317 void  
318 iput(struct inode *ip)  
319 {  
320     acquire(&icache.lock);  
321     if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){  
322         // inode has no links: truncate and free inode.  
323         if(ip->flags & I_BUSY)  
324             panic("iput busy");  
325         ip->flags |= I_BUSY;  
326         release(&icache.lock);  
327         itrunc(ip);  
328         ip->type = 0;  
329         iupdate(ip);  
330         acquire(&icache.lock);  
331         ip->flags = 0;  
332         wakeup(ip);  
333     }  
334     ip->ref--;  
335     release(&icache.lock);  
336 }
```

```
219 // Find the inode with number inum on device dev  
220 // and return the in-memory copy. Does not lock  
221 // the inode and does not read it from disk.  
222 static struct inode*  
223 igeget(uint dev, uint inum)  
224 {  
225     struct inode *ip, *empty;  
226  
227     acquire(&icache.lock);  
228  
229     // Is the inode already cached?  
230     empty = 0;  
231     for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){  
232         if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){  
233             ip->ref++;  
234             release(&icache.lock);  
235             return ip;  
236         }  
237         if(empty == 0 && ip->ref == 0)      // Remember empty slot.  
238             empty = ip;  
239     }  
240  
241     // Recycle an inode cache entry.  
242     if(empty == 0)  
243         panic("igeget: no inodes");  
244  
245     ip = empty;  
246     ip->dev = dev;  
247     ip->inum = inum;  
248     ip->ref = 1;  
249     ip->flags = 0;  
250     release(&icache.lock);  
251  
252     return ip;  
253 }
```

```
503 // Look for a directory entry in a directory.
504 // If found, set *poff to byte offset of entry.
505 struct inode*
506 dirlookup(struct inode *dp, char *name, uint *poff)
507 {
508     uint off, inum;
509     struct dirent de;
510
511     if(dp->type != T_DIR)
512         panic("dirlookup not DIR");
513
514     for(off = 0; off < dp->size; off += sizeof(de)){
515         if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
516             panic("dirlink read");
517         if(de.inum == 0)
518             continue;
519         if(namecmp(name, de.name) == 0){
520             // entry matches path element
521             if(poff)
522                 *poff = off;
523             inum = de.inum;
524             return iget(dp->dev, inum);
525         }
526     }
527
528     return 0;
529 }
```

EXT FILE SYSTEM

Advantages of inodes

- Optimized for file systems with many small files
 - Each inode can directly point to 48KB of data
 - Only one layer of indirection needed for 4MB files
- Faster file access
 - Greater meta-data locality → less random seeking
 - No need to traverse long, chained FAT entries
- Easier free space management
 - Bitmaps can be cached in memory for fast access
 - inode and data space handled independently

Ext File System

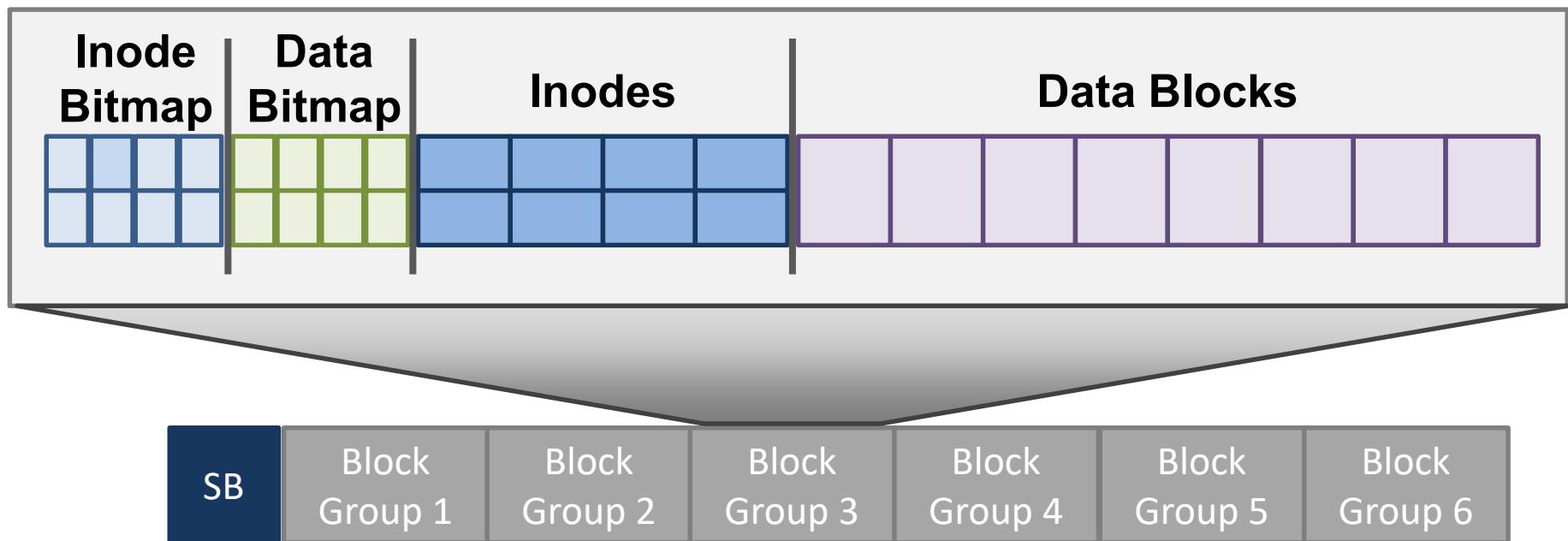
- Ext file system uses inode
 - inodes are imbalanced trees of data blocks
 - Optimized for the common case: small files
- Problem: ext has poor locality
 - inodes are far from their corresponding data
 - This is going to result in long seeks across the disk
- Problem: ext is prone to fragmentation
 - ext chooses the first available blocks for new data
 - No attempt is made to keep the blocks of a file contiguous

Fast File System (FFS) and Ext2

- FFS developed at Berkeley in 1984
 - First attempt at a **disk aware** file system
 - i.e., optimized for performance on spinning disks
- Observation: processes tend to access files that are in the same (or close) directories
 - Spatial locality
- Key idea: place groups of directories and their files into **cylinder groups**
 - Introduced into ext2, called **block groups**

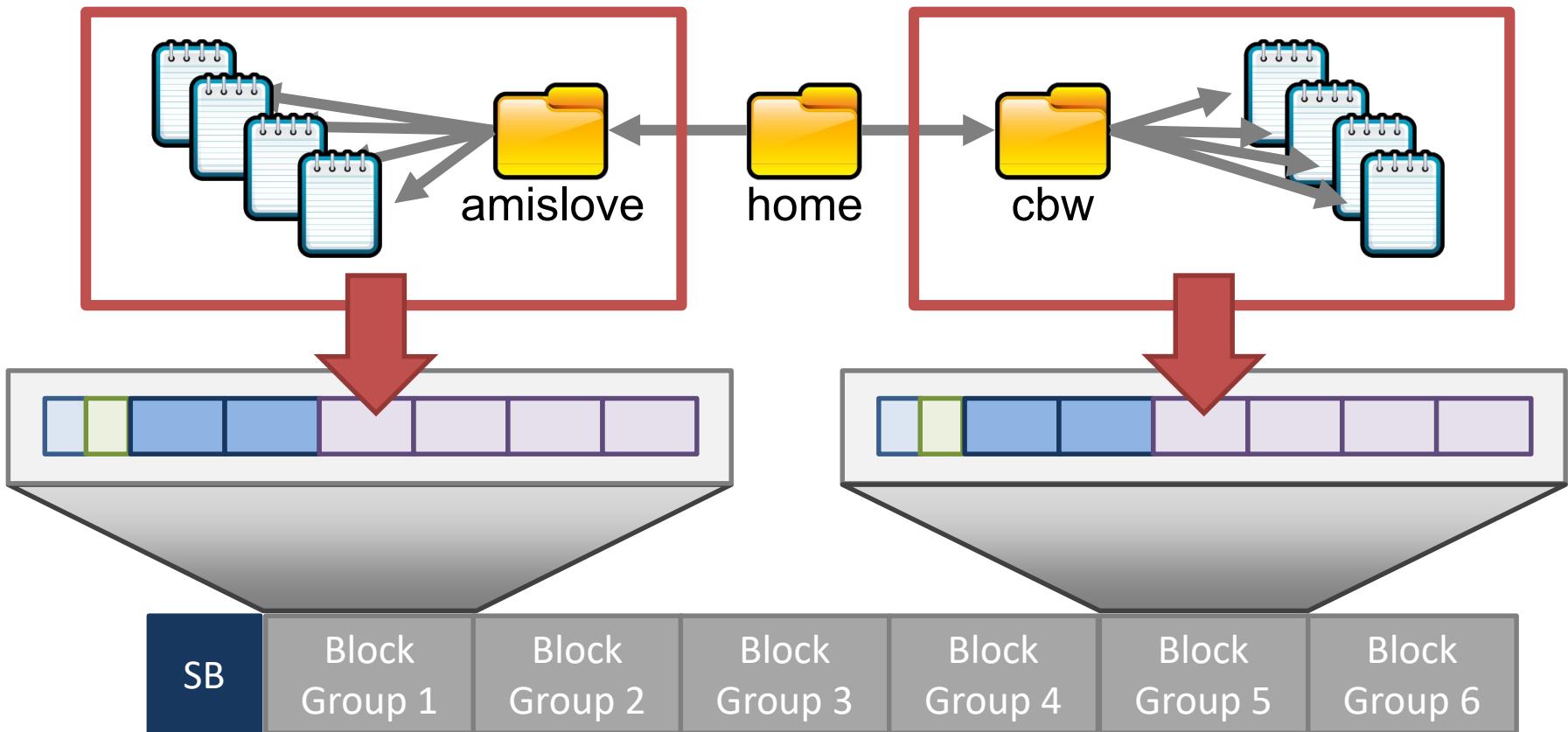
Block Groups

- In ext, there is a single set of key data structures
 - One data bitmap, one inode bitmap
 - One inode table, one array of data blocks
- In ext2, each block group contains its own key data structures



Allocation Policy

- ext2 attempts to keep related files and directories within the same block group



ext2: The Good and the Bad

- The good – ext2 supports:
 - All the features of ext...
 - ... with even better performance (because of increased spatial locality)
- The bad
 - Large files must cross block groups
 - As the file system becomes more complex, the chance of file system **corruption** grows
 - E.g. invalid inodes, incorrect directory entries, etc.

File-size upper limits for data block addressing

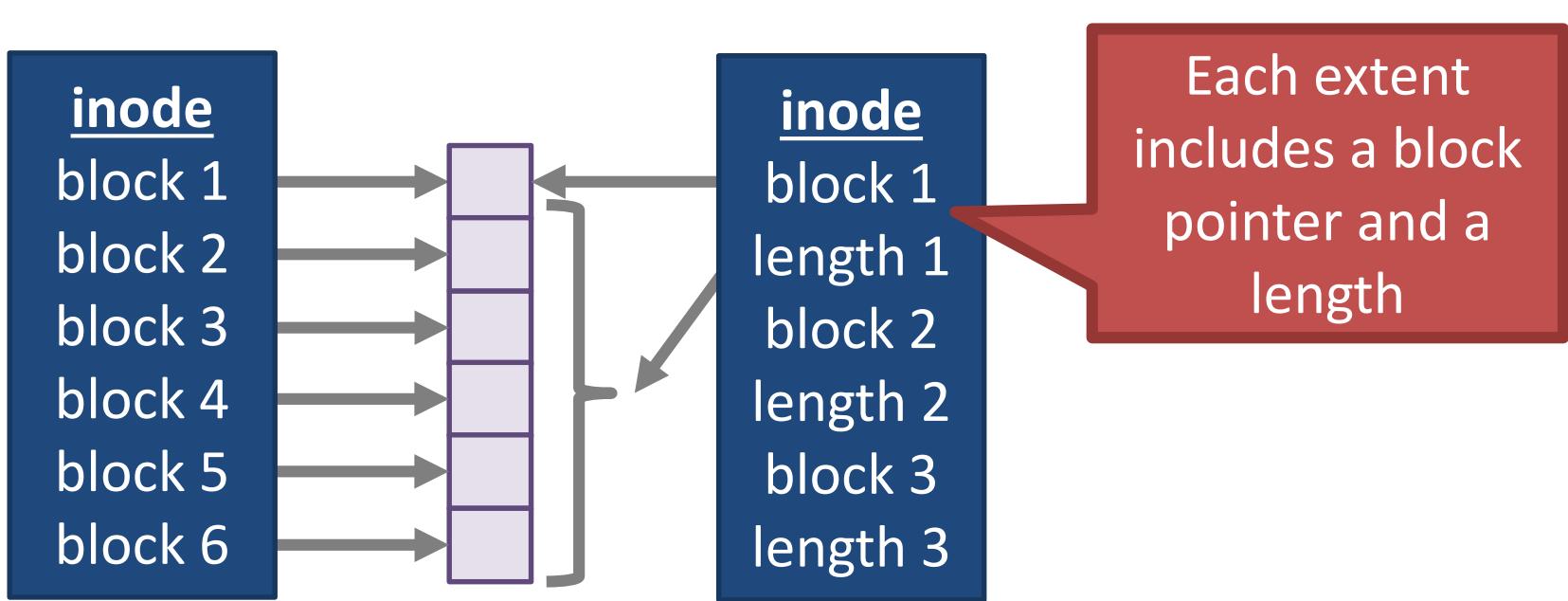
Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~ 4 TB

Extent

- Indirect block maps are incredibly inefficient for large files
 - One extra block read (and seek) every 1024 blocks
 - Really obvious when deleting big CD/DVD image files
- An extent is a single descriptor for a **range** of contiguous blocks
 - An efficient way to represent large file
 - Better CPU utilization, fewer metadata IOs

From Pointers to Extents

- Modern file systems try hard to minimize fragmentation
 - Since it results in many seeks, thus low performance
- **Extents** are better suited for contiguous files



Implementing Extents

- ext4 and NTFS use extents
- ext4 inodes include 4 extents instead of block pointers
 - Each extent can address at most 128MB of contiguous space (assuming 4KB blocks)
 - If more extents are needed, a data block is allocated
 - Similar to a block of indirect pointers

Revisiting Directories

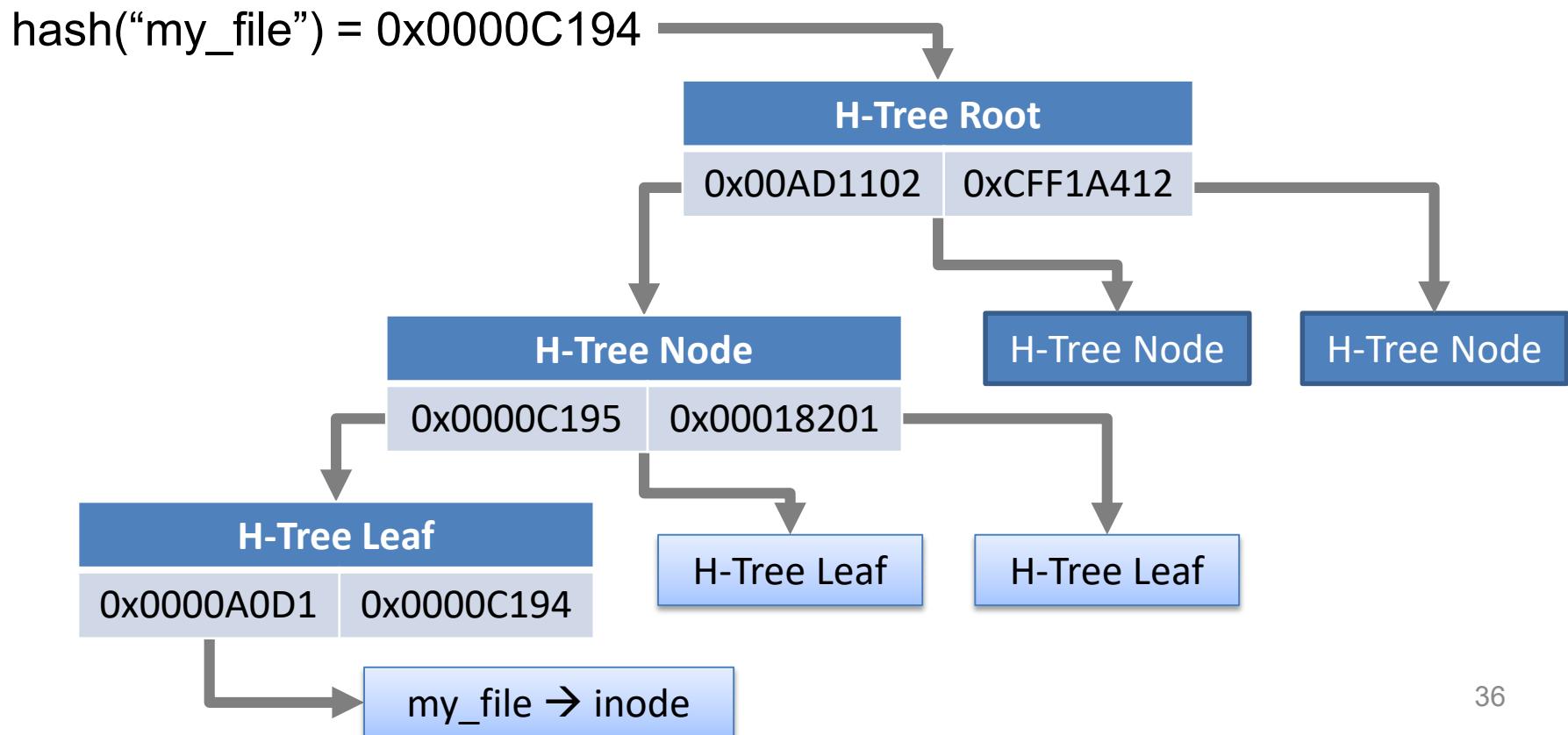
- In ext, ext2, and ext3, each directory is a file with a list of entries
 - Entries are not stored in sorted order
 - Some entries may be blank, if they have been deleted
- Problem: searching for files in large directories takes $O(n)$ time
 - Practically, you can't store $>10K$ files in a directory
 - It takes way too long to locate and open files

From Lists to B-Trees

- ext4 and NTFS encode directories as B-Trees to improve lookup time to $O(\log N)$
- A B-Tree is a type of balanced tree that is optimized for storage on disk
 - Items are stored in sorted order in blocks
- Suppose items i and j are in the root of the tree
 - The root must have 3 children, since it has 2 items
 - The three child groups contain items $a < i$, $i < a < j$, and $a > j$

Example B-Tree

- ext4 uses a B-Tree variant known as a H-Tree
 - The *H* stands for *hash* (sometime called B+Tree)
- Suppose you try to `open("my_file", "r")`



ext4: The Good and the Bad

- The good – ext4 (and NTFS) supports:
 - All of the basic file system functionalities
 - Improved performance from ext3's block groups
 - Additional performance gains from extents and B-Tree directory files
- The bad:
 - Next-gen file systems have even nicer features
 - Copy-on-write semantics (btrfs and ZFS)

File System Durability & Crash Recovery

Haibo Chen

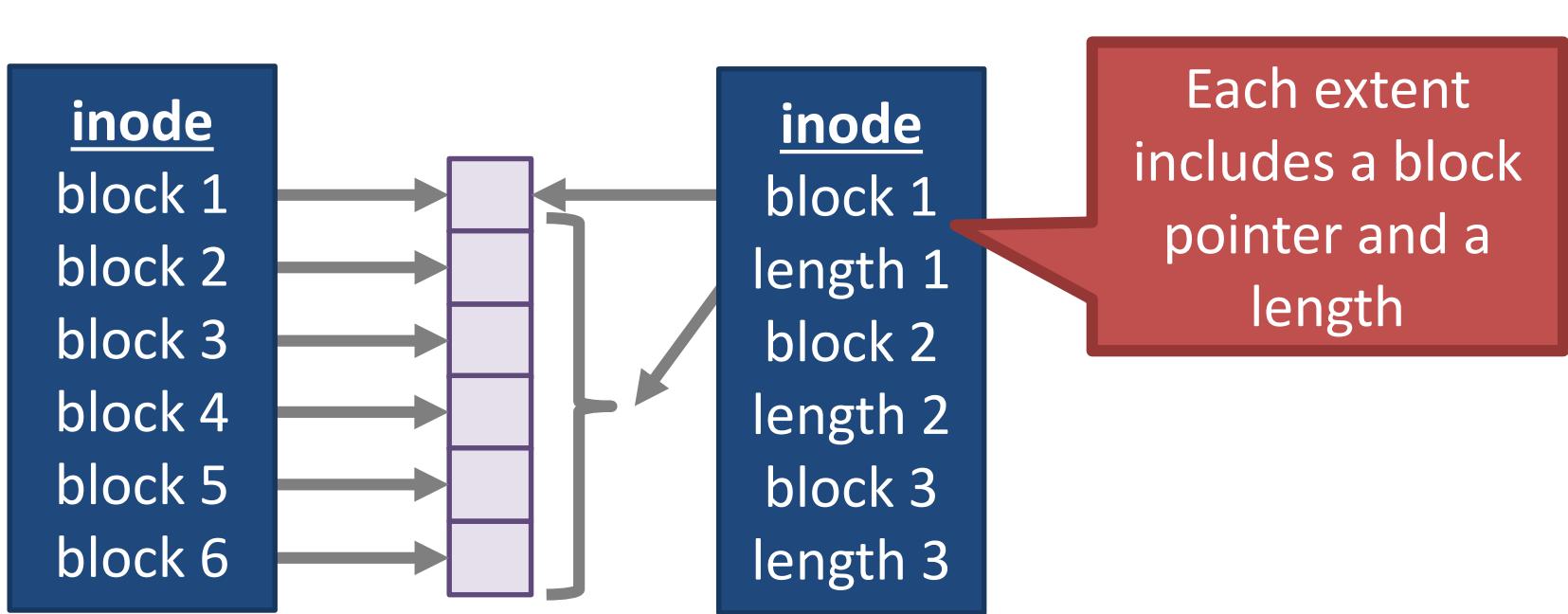
Some of the materials are adopted from Frans'
6.828 course

Review: Ext2

- The good – ext2 supports:
 - All the features of ext...
 - ... with even better performance (because of increased spatial locality)
- The bad
 - Large files must cross block groups
 - As the file system becomes more complex, the chance of file system **corruption** grows
 - E.g. invalid inodes, incorrect directory entries, etc.

Review: From Pointers to Extents

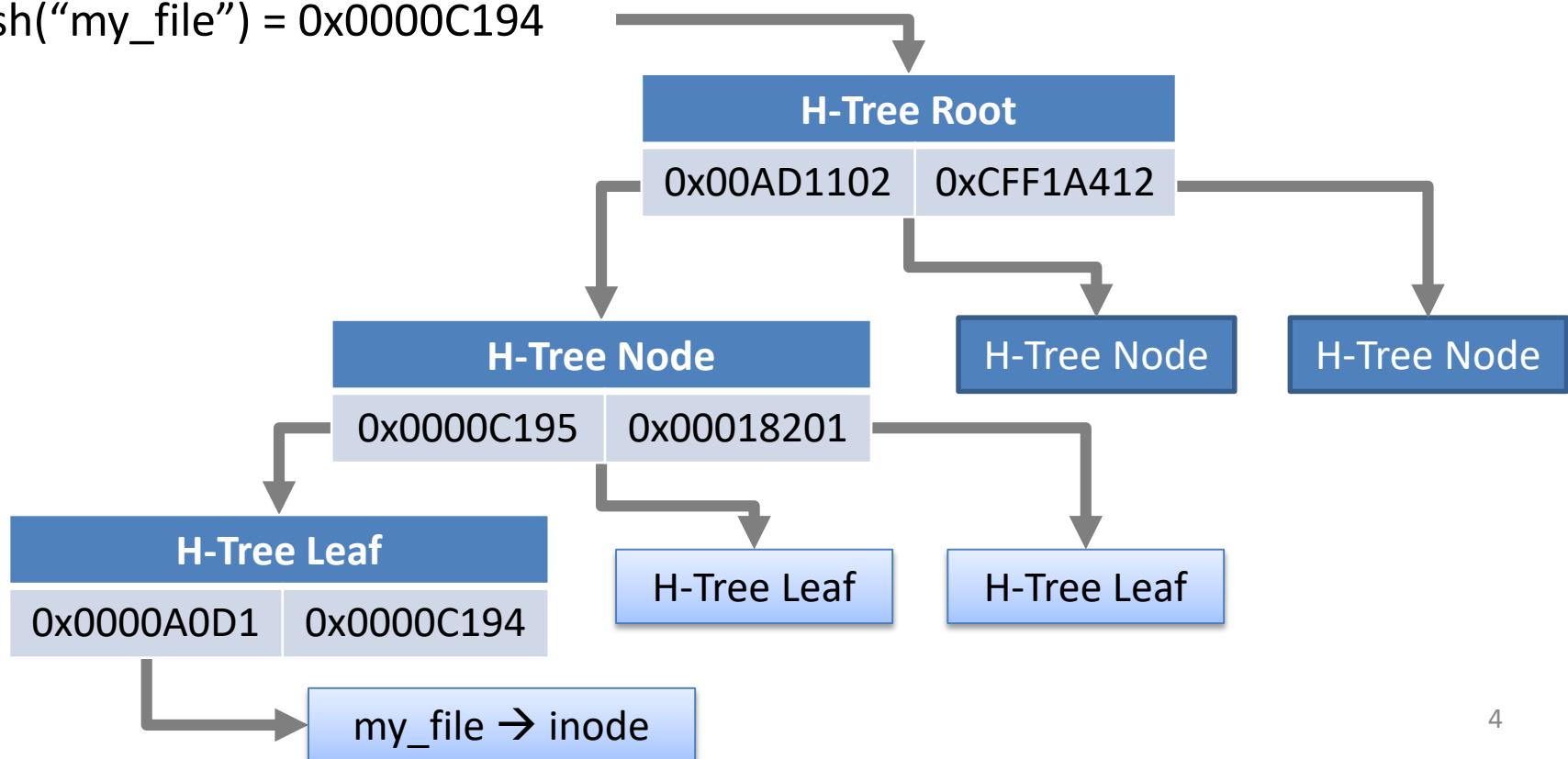
- Modern file systems try hard to minimize fragmentation
 - Since it results in many seeks, thus low performance
- Extents are better suited for contiguous files



Review: Example B-Tree

- ext4 uses a B-Tree variant known as a H-Tree
 - The *H* stands for *hash* (sometime called B+Tree)
- Suppose you try to `open("my_file", "r")`

`hash("my_file") = 0x0000C194`



FS DURABILITY & CRASH CONSISTENCY

File System Durability

Topic: tension between fs perf. and crash recovery

Disk performance is often a #1 bottleneck

"how many seeks will that take?"

Durability != Crash consistency

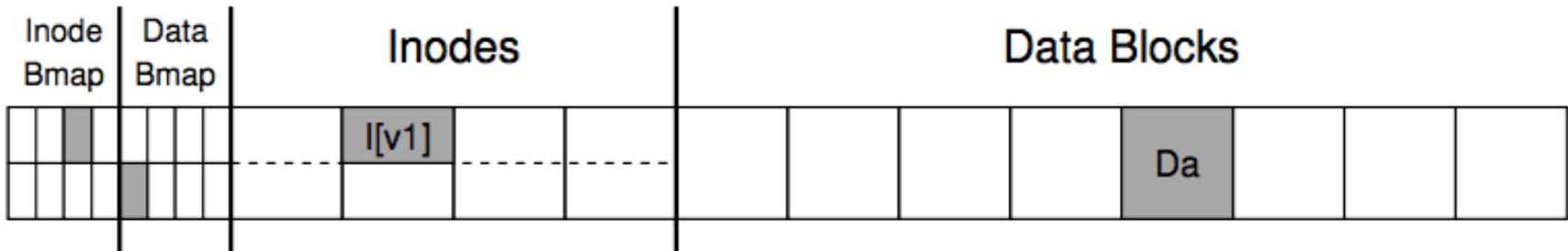
"Here is all of my data. But some of the metadata is wrong."

Crash recovery is much harder than performance

"what if a crash occurred at this point?"

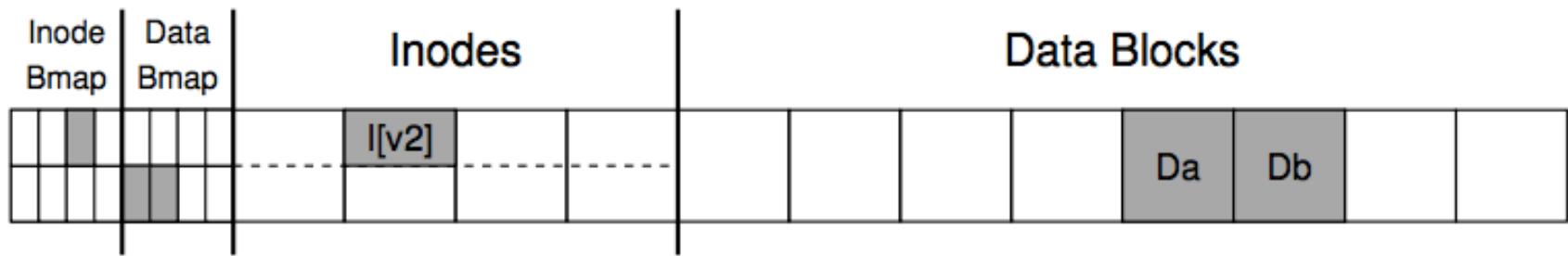
An Example: Append a File

- Inside of $I[v1]$:
 - owner : haibo
 - permissions : read-only
 - size : 1
 - pointer : 4
 - pointer : null
 - pointer : null
 - pointer : null



An Example: Append a File

- Inside of $I[v2]$:
 - owner : haibo
 - permissions : read-only
 - size : 2
 - pointer : 4
 - pointer : 5
 - pointer : null
 - pointer : null



Crash Scenarios: 1 Succeeds

- Imagine only a single write succeeds; there are thus three possible outcomes:
 - 1. Just the data block (D_b) is written to disk
 - What will happen?
 - 2. Just the updated inode ($I[v2]$) is written to disk
 - What will happen?
 - 3. Just the updated bitmap ($B[v2]$) is written to disk
 - What will happen?

Crash Scenarios: 2 Succeed

- Two writes succeed and the last one fails:
- 1. The inode ($I[v2]$) and bitmap ($B[v2]$) are written to disk, but not data (D_b)
- 2. The inode ($I[v2]$) and the data block (D_b) are written, but not the bitmap ($B[v2]$)
- 3. The bitmap ($B[v2]$) and data block (D_b) are written, but not the inode ($I[v2]$)

Our Expectation

After rebooting and running recovery code

1. FS internal invariants maintained
 - E.g., no block is both in free list and in a file
2. All but last few operations preserved on disk
 - E.g., data I wrote yesterday are preserved
 - User might have to check last few operations
3. No order anomalies
 - \$ echo 99 > result ; echo done > status

Our Assumptions

Simplifying assumptions:

- Disk is fail-stop, disk executes the writes FS sends it, and does nothing else
- Perhaps doesn't perform the very last write
- Thus: no wild writes, no decay of sectors

Why is FS crash recovery hard?

Crash = halt/restart CPU

let disk finish current sector write, assume no h/w damage,
no wild write to disk

Goal: automatic recovery

Can fs always make sense of on-disk **metadata** after restart?

Given that the crash could have occurred at **any point**?

Examples

Crash during mkdir, leave directory without . and ..

Crash during free blocks

Offline and Online Recovery

Offline recovery

file system check utility, such as chkdsk in windows and fsck on linux

E.g., ext3

Online recovery

during operation, check some important inconsistency

E.g., ext4 (also has offline fsck, but much simpler)

Terms for properties of fs ops

What effects will app see after restart + recovery

creat("a"); fd = creat("b"); write(fd,...); crash

Durable (Persistence): effects of operation are visible

Both a and b are visible

Atomic: all steps of operation visible or none

Either a and b are visible or none is visible

Ordered: exactly a prefix of operations is visible

If b is visible, then a is visible

Recovery approach

1. **Synchronous meta-data update + fsck**

Used in xv6-rev0

During check, synchronize metadata, such as file size

2. **Soft update** (FreeBSD fs modified on FFS)

not covered in this course

3. **Logging** (ext 3/4), xv6-rev6 and following versions

Before doing actual meta-data update, log the event

After crash, recover from log

SYNC METADATA UPDATE + FSCK

Typical set of tradeoffs

FS ensures it can recover its **meta-data** (minimal requirements for a real FS)

- Internal consistency

- No dangling references

- Inode and block free list contain only used (not using) items

- Unique name in one directory, etc.

Weak semantic FS provided limited guarantees

- Atomicity for creat, rename, delete

- Often no durability for anything

 - (creat("a"), then crash, no a)

- Often no order guarantees

How do applications handle this weak semantics?

Example

Edit your file, then crash, only half of your file is actually updated?

Fsync and rename (shadow copy)

Fsync force durability, only returned if file is actually written on disk

Rename is an atomic operation, only old name or new name, not half old half new

Mac OS intensively uses rename to ensure atomicity

What Does fsck do?

- 1. Check superblock
 - E.g., making sure the file system size is greater than the number of blocks allocated
 - If error, use an alternate copy of the superblock
- 2. Check free blocks
 - Scans the inodes, indirect blocks, double indirect blocks, etc.
 - Uses this knowledge to produce a correct version of the allocation bitmaps
 - Same for the inode bitmap

What Does fsck do?

- 3. Check inode states
 - Check type: regular file, dir, symbolic link, etc.
 - Clear suspect inodes and clear the inode bitmap
- 4. Check inode links
 - Check link count by scanning the entire fs tree
 - If count mismatches, fix the inode
 - If inode is allocated but no dir contains it, lost+found
- 5. Check duplicates
 - Two inodes refer to the same block
 - If one inode is obviously bad, clear it; otherwise, copy the block and give each a copy

What Does fsck do?

- 6. Check bad blocks
 - E.g., point to some out-of-range address
 - What should fsck do? Just remove the pointer
- 7. Check directories
 - The only file that fsck know more semantic
 - Making sure that “.” and “..” are the first entries
 - Ensure no dir is linked more than once
 - No same filename in one dir

Problem of fsck: Too Slow

How long would fsck take?

an example server: fsck takes 10 minutes per 70GB disk w/ 2 million inodes

clearly reading
still a long time

Consider the cost

Scan the disk
Just like find



clearly, not seeking
disk size

writes
entire house

Would an xv6 FS be internally consistent after a crash?

Xv6-rev0 strategy: carefully order disk writes to avoid dangling refs

1. initialize a new inode before creating dirent
2. delete dirent before marking inode free
3. mark block in-use before adding it to inode addrs[]
4. remove block from addrs[] before marking free
5. zero block before marking free

Has some visible bugs:

- . and .. during mkdir(), link counts, sizes

Has some invisible loose ends
may lose freed blocks and inodes

Example

File creation: what's the right order of synchronous writes?

1. mark inode as allocated
2. create directory entry

File deletion

1. erase directory entry
2. erase inode addrs[], mark as free
3. mark blocks free

What about app-visible syscall semantics?

- Durable? Yes
 - Use write-through cache, sync I/O, O_SYNC
- Atomic? Often
 - **Mkdir** is an exception
- Ordered? Yes
 - If all writes are sync

Recall: Sync I/O vs. Async I/O

Asynchronous I/O is a **poor** abstraction for:

- Reliability

- Ordering

- Durability

- Ease of programming

Synchronous I/O is superior but **100x slower**

- Caller blocked until operation is complete

Issues with Synchronous Write

Main issue

- very slow during normal operation

- very slow during recovery

Barrier: Flush the Disk

- Disk's write buffer
 - Disk will inform the OS the write is complete when it simply has been placed in the disk's memory cache
 - But the data is not on disk yet! No durability! No order!
- One solution: disable the buffer
- Another solution: using flush operation
 - Force the disk to write data to disk media
 - Aka., disk write barrier
- However, disks may not do as they claim...
 - Some disks just ignore the flush operation to be faster
 - “the fast almost always beats out the slow, even if it is wrong” --- Kahan”

Ordinary perf. of sync meta-data update?

Creating a file and writing a few bytes

Takes 8 writes, probably 80 ms

(`ialloc`, `init inode`, `write dirent`, `alloc data block`, `add to inode`, `write data`, `set length in inode`, `xxx`)

So can create only about a dozen small files per second! Think about `un-tar` or `rm *`

How to get better performance?

Reality

RAM is cheap

disk sequential throughput is high, 50 MB/sec (maybe someday solid state disks will change the landscape)

Why not use a big write-back disk cache?

no sync meta-data update operations

only modify in-memory disk cache (no disk write)

so creat(), unlink(), write() &c return almost immediately bufs written to disk later

if cache is full, write LRU dirty block

write all dirty blocks every 30 seconds, to limit loss if crash

this is how old Linux EXT2 file system worked

Write-back Cache

Would write-back cache improve performance? why, exactly?

after all, you have to write the disk in the end anyway

What can go wrong with write-back cache?

example: unlink() followed by create() an existing file x with some content, all safely on disk

one user runs unlink(x)

1. delete x's dir entry **
2. put blocks in free bitmap
3. mark x's inode free; another user then runs create(y)
4. allocate a free inode
5. initialize the inode to be in-use and zero-length
6. create y's directory entry **

again, all writes initially just to disk buffer cache

suppose only ** writes forced to disk, then crash

what is the problem?

can fsck detect and fix this?

JOURNALING

Logging (Journaling)

Goal:

atomic system calls with respect to crashes

fast recovery (no hour-long fsck)

speed of write-back cache for normal operations

Approach

will introduce logging in two steps

first xv6's log, which only provides safety

then Linux EXT3, which is also fast

Basic idea behind logging

To ensure atomicity:

all of a system call's writes, or none

let's call an atomic operation a "transaction"

record all writes the sys call *will* do in the log

then record "done"

then do the writes

on crash+recovery:

if "done" in log, replay all writes in log

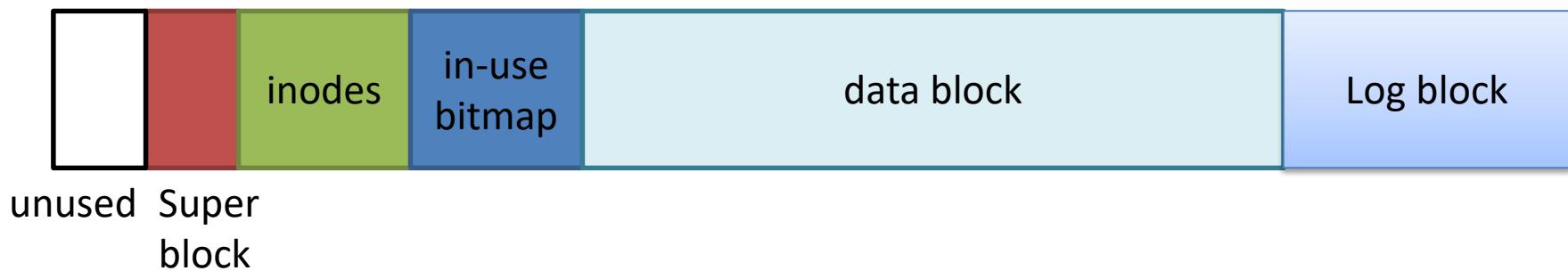
if no "done", ignore log

this is a WRITE-AHEAD LOG

Xv6's simple logging

FS has a log on disk syscall:

```
begin_trans()  
bp = bread()  
bp->data[] = ...  
log_write(bp)  
more writes ...  
commit_trans()
```



Recap: balloc

```
// Allocate a zeroed disk block.
static uint
balloc(uint dev)
{
    int b, bi, m;
    struct buf *bp;
    struct superblock sb;

    bp = 0;
    readsb(dev, &sb);
    for(b = 0; b < sb.size; b += BPB){
        bp = bread(dev, BBL0CK(b, sb.ninodes));
        for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
            m = 1 << (bi % 8);
            if((bp->data[bi/8] & m) == 0){  
                if(b + bi < sb.size) balloc_free?  
                    bp->data[bi/8] |= m; Update bitmap
                    log_write(bp);
                    brelse(bp);
                    bzero(dev, b + bi);
                    return b + bi;
            }
        }
        brelse(bp);
    }
    panic("balloc: out of blocks");
}
```

```
void
log_write(struct buf *b)
{
    int i;

    if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
        panic("too big a transaction");
    if (!log.busy)
        panic("write outside of trans");

    for (i = 0; i < log.lh.n; i++) {
        if (log.lh.sector[i] == b->sector) // log absorbtion?
            break;
    }
    log.lh.sector[i] = b->sector;
    struct buf *lbuf = bread(b->dev, log.start+i+1);
    memmove(lbuf->data, b->data, BSIZE);
    bwrite(lbuf);
    brelse(lbuf);
    if (i == log.lh.n)
        log.lh.n++;
    b->flags |= B_DIRTY; // XXX prevent eviction
}
```

Transaction Semantics in xv6

`begin_trans:`

- need to indicate which group of writes must be atomic!

- lock -- xv6 allows only one transaction at a time

`log_write:`

- record sector #

- append buffer content to log

- leave modified block in buffer cache (but do not write)

`commit_trans():`

- record "done" and sector #s in log

- do the writes

- erase "done" from log

`recovery:`

- if log says "done": copy blocks from log to real locations on disk

```
void  
begin_trans(void)  
{  
    acquire(&log.lock);  
    while (log.busy) {  
        sleep(&log, &log.lock);  
    }  
    log.busy = 1;  
    release(&log.lock);  
}
```

```
void
commit_trans(void)
{
    if (log.lh.n > 0) {
        write_head();      // Write header to disk -- the real commit
        install_trans();  // Now install writes to home locations
        log.lh.n = 0;     // Erase the transaction from the log
    }
}

acquire(&log.lock);
log.busy = 0;
wakeup(&log);
release(&log.lock);
}
```

```
static void
recover_from_log(void)
{
    read_head(); // read log header
    install_trans(); // if committed, copy from log to disk
    log.lh.n = 0;
    write_head(); // clear the log
}
```

```
// Copy committed blocks from log to their home location
static void
install_trans(void)
{
    int tail;

    for (tail = 0; tail < log.lh.n; tail++) {
        struct buf *lbuf = bread(log.dev, log.start+tail+1); // read log block
        struct buf *dbuf = bread(log.dev, log.lh.sector[tail]); // read dst
        memmove(dbuf->data, lbuf->data, BSIZE); // copy block to dst
        bwrite(dbuf); // write dst to disk
        brelse(lbuf); // free log block
        brelse(dbuf);
    }
}
```

Sys_unlink

```
struct inode *dp =
nameiparent(path, name)
```

```
begin_trans();

ilock(dp);

// Cannot unlink .. or ....
if(namecmp(name, "..") == 0 || namecmp(name, "...") == 0)
    goto bad;

if((ip = dirlookup(dp, name, &off)) == 0)
    goto bad;
ilock(ip);

if(ip->nlink < 1)
    panic("unlink: nlink < 1");
if(ip->type == T_DIR && !isdirempty(ip)){
    iunlockput(ip);
    goto bad;
}

memset(&de, 0, sizeof(de));
if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("unlink: writei");
if(ip->type == T_DIR){
    dp->nlink--;
    iupdate(dp);
}
iunlockput(dp);

ip->nlink--;
iupdate(ip);
iunlockput(ip);

commit_trans();
```

Limitation with xv6 logging

Only one transaction at a time

two system calls might be modifying different parts of the FS
synchronous write to on-disk log

each write takes one disk rotation time

commit takes another

a file create/delete involves around 10 writes

thus 100 ms per create/delete -- very slow!

tiny update -> whole block write

creating a file only dirties a few dozen bytes

but produces many kilobytes of log writes

synchronous writes to home locations after commit

i.e. write-through, not write-back

makes poor use of in-memory disk cache

Questions

How can we get both performance and safety?

we'd like system calls to proceed at in-memory speeds
using write-back disk cache

i.e. have typical system call complete w/o actual disk
writes

Journaling and ext3

Yubin Xia

Ack: slides adopted from Frans' 6.828 course

Outline

- Recap
- Journaling
- Ext3 journaling
- Optimization in VM

Recap: File system durability

Topic: tension between FS perf. and crash recovery

Disk performance is often a #1 bottleneck
"how many seeks will that take?"

But many obvious fixes make crash / power failure hard to recover from
"what if a crash occurred at this point?"

Crash recovery is much harder than performance

Recap: Recovery approach

Synchronous meta-data update + fsck

- Used in xv6-rev0

- During check, synchronize meta-data, such as file size

Logging (ext 3/4), xv6-rev6 and following versions

- Before doing actual meta-data update, log the event

- After crash, recover from log

Soft update (FreeBSD fs modified on FFS)

- Soft update, not covered in this course

Recap: synchronous metadata update?

Xv6-rev0 strategy: carefully order disk writes to avoid dangling refs

1. initialize a new inode before creating dirent
2. delete dirent before marking inode free
3. mark block in-use before adding it to inode addrs[]
4. remove block from addrs[] before marking free
5. zero block before marking free

Has some visible bugs:

- . and .. during mkdir(), link counts, sizes

Has some invisible loose ends

may lose freed blocks and inodes

JFS: Journaling FS

- Speed recovery time after a crash
 - fsck on a large disk can be very slow
 - ‘to eliminate enormously long filesystem recovery times after a crash’
- With JFS you just reread the journal after a crash, from the last checkpoint

JFS VS. log-structured file system

- A log structured file system ONLY contains a log, everything is written to the end of this log
- Log-structured FS dictates how the data is stored on disk
- JFS does not dictate how the data is stored on disk

How does it work?

- Each disk update is a Transaction (atomic update)
 - Write new data to the disk (journal)
 - The update is not final until a commit block is written
- Atomicity of the commit block write
 - The commit block is a single block of data on the disk
 - Not necessarily flushed to disk yet!

How does data get out of the journal?

- After a commit the new data is in the journal
 - It needs to be written back to its home location on the disk
 - Cannot reclaim that journal space until we resync the data to disk

To finish a Commit (checkpoint)

- Close the transaction
 - All subsequent filesystem operations will go into another transaction
- Flush transaction to disk (journal), pin the buffers
- After everything is flushed to the journal, update journal header blocks
- Unpin the buffers in the journal **only after** they have been synced to the disk
- Release space in the journal

ext3 and JFS

- Two separate layers
 - /fs/ext3 – just the filesystem with transactions
 - /fs/jdb – just the journaling stuff (JFS)
- ext3 calls JFS as needed
 - Start/stop transaction
 - Ask for a journal recovery after unclean reboot
- Actually do compound transactions
 - Transactions with multiple updates

Linux's ext3's Journaling

Case study of the details required to add logging to a file system

Stephen Tweedie 2000 talk transcript "EXT3, Journaling Filesystem"

<http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>

Ext3 adds a log to ext2, a previous xv6-like log-less file system

Has many modes, start with "journaled data" mode
log contains *both* metadata and file content blocks
Will introduce “ordered mode” later

Ext3 Structures

in memory:

- Write-back block cache

- Per-transaction info:

- set of block numbers to be logged

- set of outstanding "handles" -- one per syscall

on disk:

- FS

- Circular log

What's in the ext3 log?

Log superblock: starting offset and starting seq #
(not the FS superblock; it's a block at start of log file)

Descriptor blocks: magic, seq, block #s

Data blocks (as described by descriptor)

Commit blocks: magic, seq

|super: offset+seq #| ... |Descriptor 4|...blocks...|Commit 4| |Descriptor 5|...

How does ext3 get good perf?

Batching

commits every few seconds, not after every system call
so each transaction includes many syscalls

Why does batching help performance?

1. amortize fixed transaction cost over many transactions
including descriptor and data blocks
2. "write absorption"
many syscalls in the batch may modify the same block (i-node, bitmap, dirent)
thus one disk write for many syscalls' updates
3. better concurrency
less waiting for previous syscall to finish commit

How does ext3 get good perf?

Ext3 allows concurrent transactions and syscalls

There may be multiple transactions:

- some fully committed in the on-disk log

- some doing the log writes as part of commit

- one** "open" transaction accepting new sys-calls

ext3 commits current transaction every few seconds (or fsync())

Syscall process

```
sys_open() {  
    h = start()  
    get(h, block #)  
    modify the block in the cache  
    stop(h)  
}
```

Syscall process

start():

Tells logging system to make set of writes until stop() atomic

Logging system must know the set of outstanding system calls
cannot commit until they are all complete

start() can block this sys call if needed

get():

Tells logging system we will modify cached block
added to list of blocks to be logged

Prevent writing block to disk until after transaction commits

stop():

stop() does **not** cause a commit

Transaction can commit if and only if all included syscalls have called stop()

Committing a transaction to disk

1. block new syscalls
2. wait for in-progress syscalls to stop()
3. open a new transaction, unblock new syscalls
4. write descriptor to log on disk w/ list of block #s
5. write each block from cache to log on disk
6. wait for all log writes to finish
7. write the commit record
8. wait for the commit write to finish
9. now cached blocks allowed to go to homes on disk (but not forced)

Is log correct if concurrent syscalls?

e.g., create of "a" and "b" in same directory

inode lock prevents race when updating directory

other stuff can be truly concurrent (touches different blocks in cache)

transaction combines updates of both system calls

What if syscall B reads uncommitted result of syscall A?

A: echo hi > x

B: ls > y

Could B commit before A, so that crash would reveal anomaly?

case 1: both in same transaction -- ok, both or neither

case 2: A in T1, B in T2 -- ok, A must commit first

case 3: B in T1, A in T2

could B see A's modification?

ext3 must wait for all ops in prev transaction to finish

before letting any in next start

so that ops in old transaction don't read modifications of next transaction

What if syscall B reads uncommitted result of syscall A?

The larger point: the **commit order** must be consistent with the order in which the system calls read/wrote state.

Ext3 sacrifices a bit of performance here to gain correctness.

Is it safe for a syscall in T2 to write a block that was also written in T1?

ext3 allows T2 to start before T1 finishes committing -- can take a while

T1: | -syscalls- | -commitWrites- |

T2: | -syscalls- | -commitWrites- |

the danger:

a T1 syscall writes block 17

T1 closes, starts writing cached blocks to log

T2 starts, a T2 syscall also writes block 17

could T1 write T2's modified block 17 to the T1 transaction in the log?

bad: not atomic, since then a crash would leave some but not all off T2's writes committed

Is it safe for a syscall in T2 to write a block that was also written in T1?

ext3 allows T2 to start before T1 finishes committing -- can take a while

T1: | -syscalls- | -commitWrites- |

T2: | -syscalls- | -commitWrites- |

Solution:

xt3 gives T1 a private copy of the block cache as it existed when T1 closed
T1 commits from this snapshot of the cache

It is efficient using copy-on-write

The copies allow syscalls in T2 to proceed while T1 is committing

The point:

Correctness requires a post-crash+recover state as if syscalls had executed
atomically and sequentially

Ext3 uses various tricks to allow some concurrency

When can ext3 re-use transaction T1's log space?

The log is circular

It can reuse once all transactions prior to T1 have been freed in the log, and T1's cached blocks have all been written to FS on disk

free == advance log superblock's start pointer/seq

What if not enough free space in log for a syscall?

Suppose we start adding syscall's blocks to T2:
Half way through, realize T2 won't fit on disk

We cannot commit T2, since syscall not done
We cannot back out of this syscall, either
 there's no way to undo a syscall
 other syscalls in T2 may have read its modifications

Solution: Reservations

syscall pre-declares how many block of log space it might need

block the syscall from starting until enough free space

may need to commit open transaction, then free older transaction

OK since reservations mean all started syscalls can complete + commit

Performance

create 100 small files in a directory

would take xv6 over 10 seconds (many disk writes per syscall)

repeated modifications to same direntry, inode, bitmap
blocks in cache

write absorption...

then one commit of a few metadata blocks plus 100 file
blocks

how long to do a commit?

seq write of $100 * 4096$ at 50 MB/sec: 10 ms

wait for disk to say “writes are on disk”

then write the commit record

that wastes one rotation, another 10 ms

What if a crash?

crash may interrupt writing last transaction to log on disk

so disk may have a bunch of full transactions, then maybe one partial

may have written some block cache to disk (home) but only for fully committed transactions, not partial last one

How does recovery work?

1. find the start of the log -- the first non-freed descriptor
log "superblock" contains offset and seq# of first transaction
(advanced when log space is freed)
2. find the end of the log
scan until bad magic or not the expected seq number
go back to last commit record
crash during commit -> no commit record, recovery ignores
3. replay all blocks through last complete transaction
in log order

How does recovery work?

what if block after last valid log block looks like a log descriptor?

perhaps left over from previous use of log? (**seq...**)

perhaps some file data happens to look like a descriptor? (**magic #...**)

when can ext3 free a transaction's log space?

after cached blocks have been written to FS on disk
free == advance log superblock's start pointer/seq

Durability of ext3

ext3 not as immediately durable as xv6

creat() returns -> maybe data is not on disk! crash will undo it.

need fsync(fd) to force commit of current transaction, and wait

would ext3 have good performance if commit after every syscall?

would log many more blocks, no absorption

10 ms per syscall, rather than 0 ms

("Rethink the Sync" [OSDI'06] addresses this problem)

Ordered Mode vs Journaled Mode

journaling file content is slow, every data block written **twice**

can we just lazily write file content blocks?

no:

if metadata updated first, crash may leave file pointing to blocks with someone else's data

ext3 ordered mode:

don't write file content to the log

write content blocks to disk *before* committing inode w/ new size and block #

most people use ext3 ordered mode

Correctness w/ ordered mode

- A. rmdir, re-use block for write() to some file,
crash before rmdir or write committed
-
- after recovery, as if rmdir never happened,
but directory block has been overwritten!
-
- fix: no re-use of freed block until freeing syscall
committed

Correctness w/ ordered mode

B. rmdir, commit, re-use block in file, ordered file write, commit,

crash, replay rmdir

file is left w/ directory content e.g, . . and ..

since file content write is not replayed

fix: “revoke” records, prevent log replay of a given block

note: both problems due to changing the type of a block (content vs meta-data) so another solution might be to never do that

Another Corner Case

- open a file, then unlink it
- unlink commits
- file is open, so unlink removes dirent but doesn't free blocks
- Crash
- nothing interesting in log to replay
- inode and blocks not on free list, also not reachable by any name
 - will never be freed! oops
- Solution: add inode to linked list starting from FS superblock
 - commit that along with remove of dirent
 - recovery looks at that list, completes deletions

Checksums

- Recall: transaction's log blocks must be on disk before writing commit block ext3 waits for disk to say "done" before starting commit block write
- Risk: disks usually have write caches and re-order writes, for performance
 - sometimes hard to turn off (the disk lies)
 - people often leave re-ordering enabled for speed, out of ignorance
 - bad news if disk writes commit block before the rest of the transaction
- Solution: commit block contains checksum of all data blocks
 - on recovery: compute checksum of datablocks
 - if matches checksum in commit block: install transaction
 - if no match: don't install transaction
- ext4 has log checksumming

Xv6 vs. ext3

does ext3 fix the xv6 log performance problems?

only one transaction at a time -- yes

synchronous write to on-disk log -- yes, but 5-second window

tiny update -> whole block write -- yes (indirectly)

synchronous writes to home locations after commit – yes

ext3 is very successful

but: no checksum -- ext4

but: not efficient for applications that use fsync()

VIRTUAL DISK & VMRAM

Wide Use of Virtual Disks

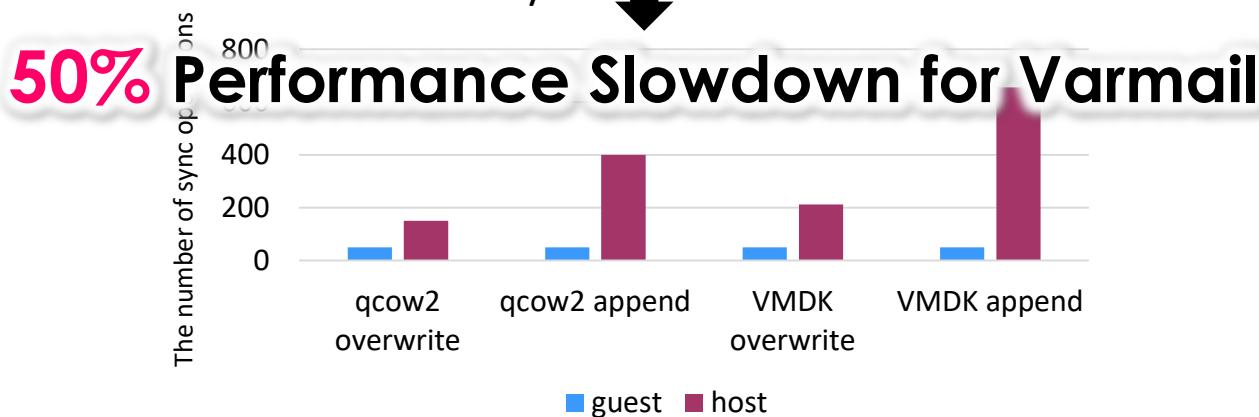
- Virtual disks provide many useful features
 - E.g., dynamic growing of image size, snapshot, deduplication
 - Such features can significantly ease tasks such as VM deployment, backup
- Virtual disks have been widely used in major cloud infrastructures
 - E.g., OpenStack

Sync Amplification for Virtual Disks

Environment setup:

- Platform: QEMU/KVM
- Virtual disk type: **More than 3X for qcow2**
- Workload: overwrite & append

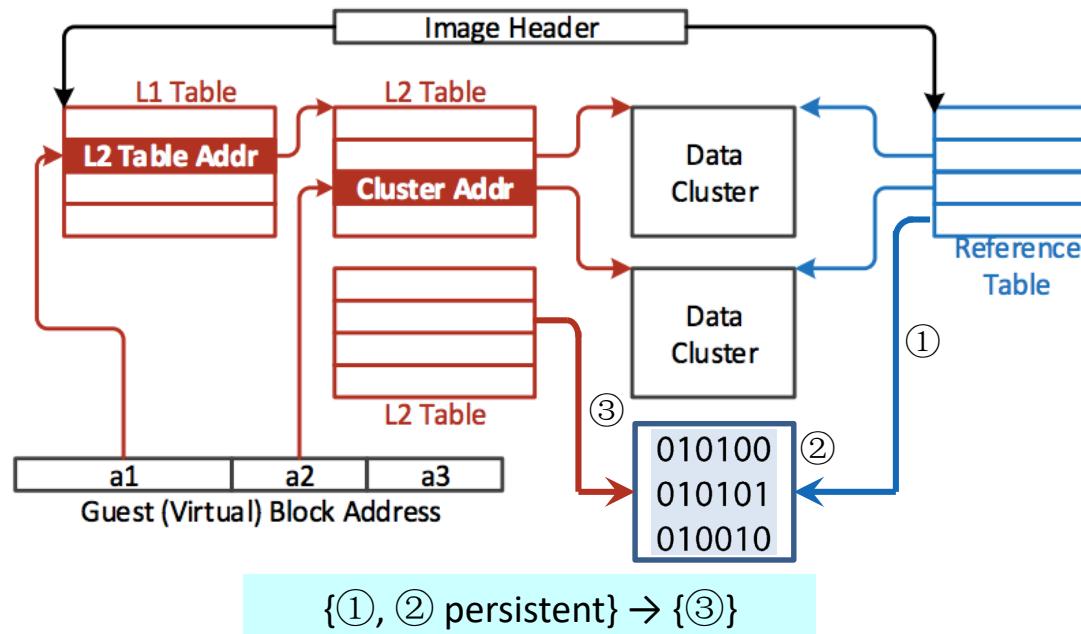
More than 4X for VMDK
#sync operations observed from
inside/outside of the VM



Why Sync Amplification Happens?

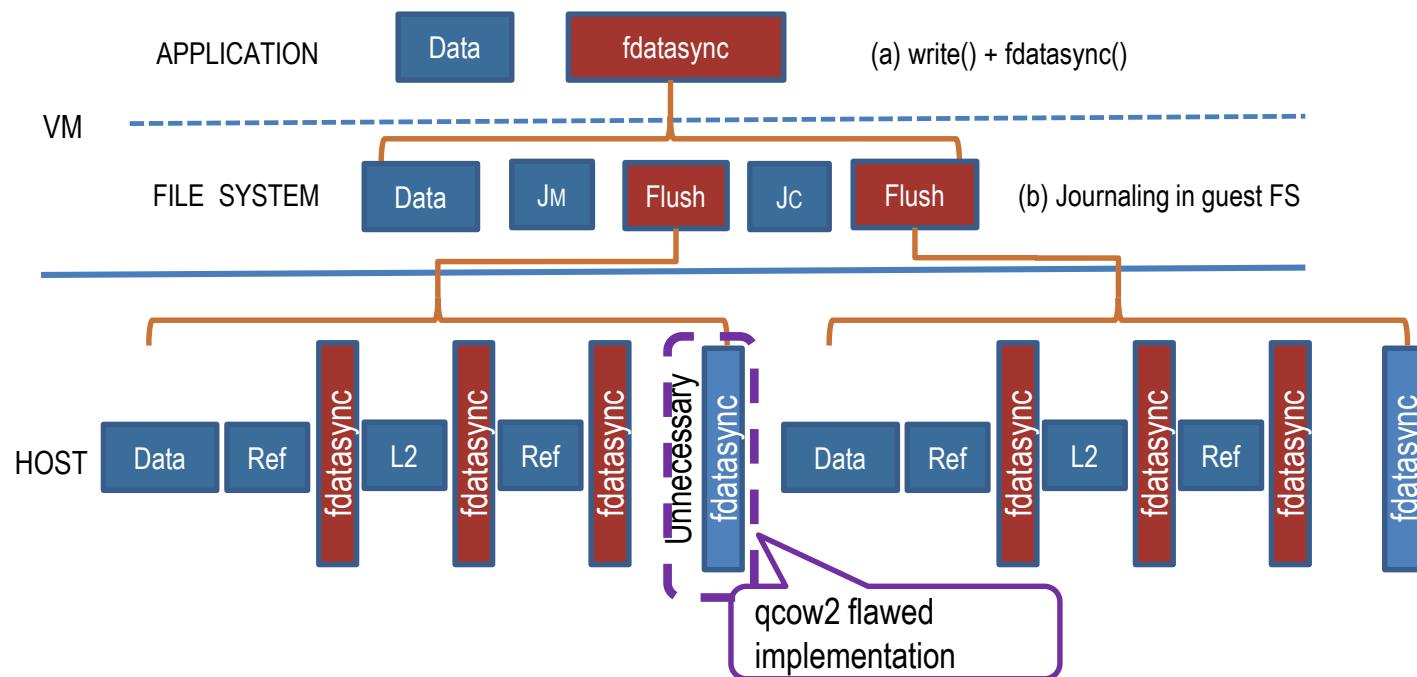
Virtual disks has to impose strong ordering while being updated to retain crash consistency

- E.g., qcow2



How Sync Amplification Happens?

The process of appending a block of data to an empty file and make the data durable:



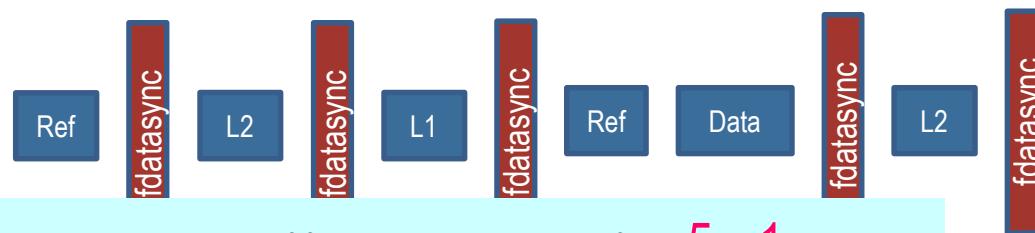
No-data Mode Journaling

Only log data checksum and metadata in journal

Data sequence:

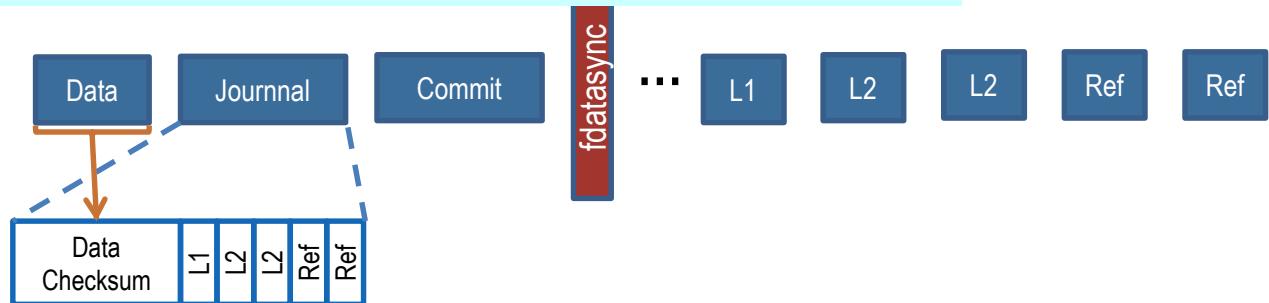


Original system:

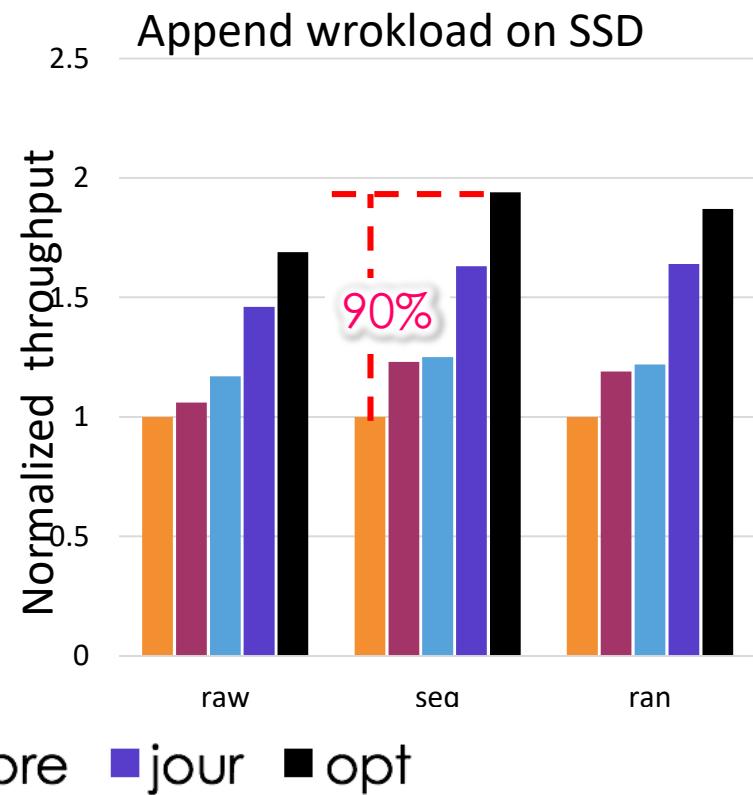
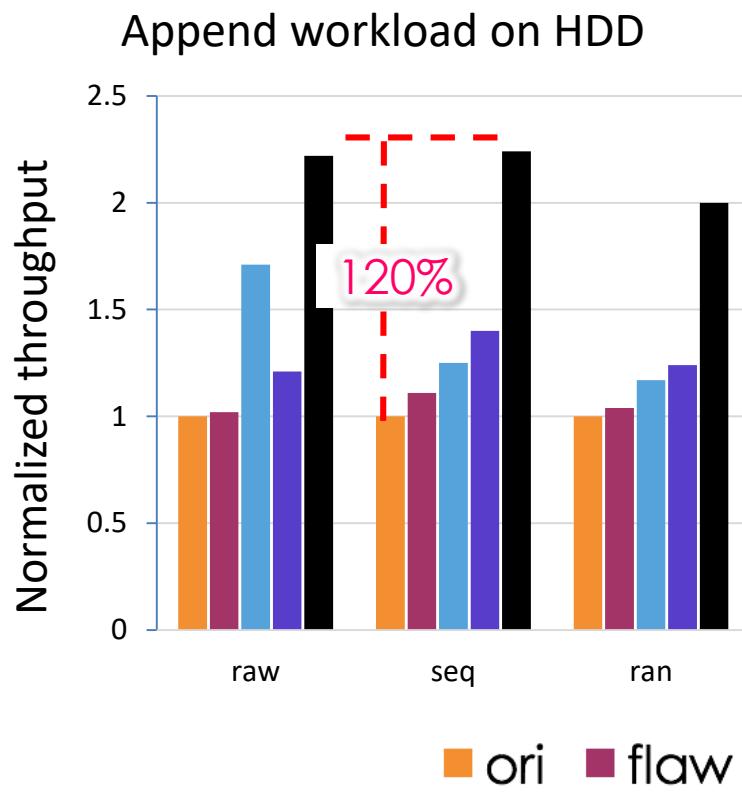


The number of fdatasync decreases from 5 to 1

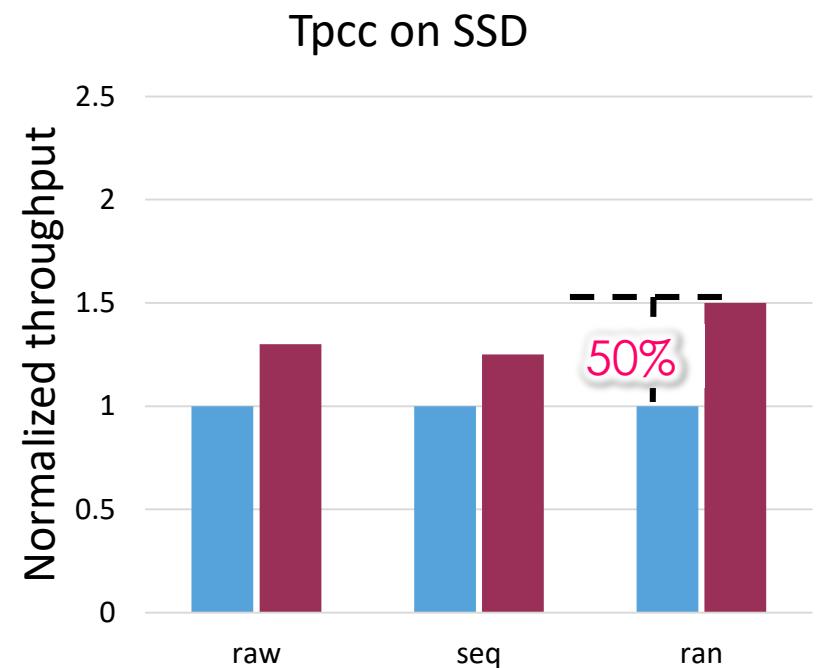
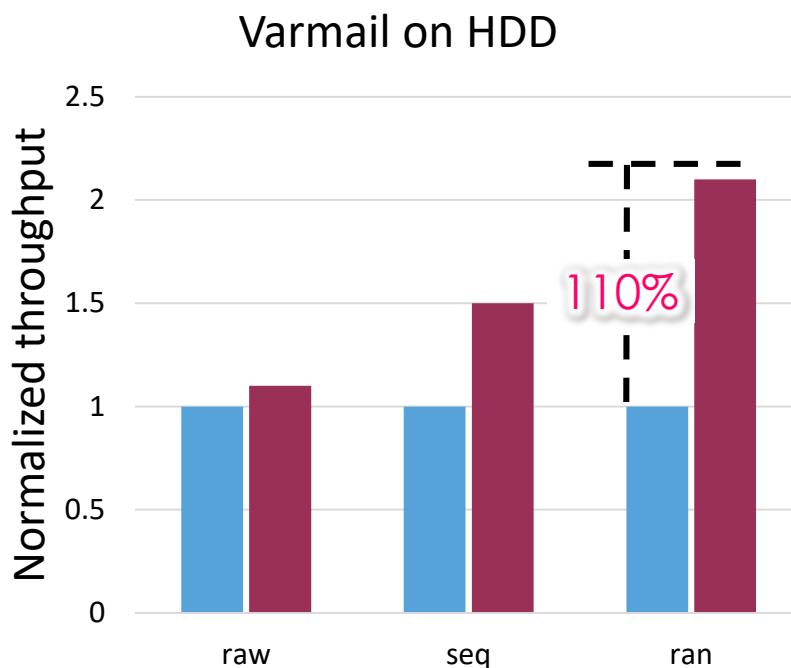
No-data mode:



Micro-benchmark



Macro-benchmark



FS: FAT32 & NTFS

Yubin Xia

FAT-16
FAT-32
exFAT

FAT FILE SYSTEM

Clusters and Sectors

- Sector: smallest storage unit on disk
 - 512 bytes
- Cluster: smallest allocated disk space to hold file
 - Data clusters are located after metadata of partition
 - Different cluster sizes depending on volume size

Volume Size	FAT32 Cluster Size
<32MB	Not Supported
32MB ~ 64MB	512 bytes
65MB ~ 128MB	1KB
129MB ~ 256MB	2KB
257MB ~ 8GB	4KB
8GB ~ 16GB	8KB
16GB ~ 32GB	16KB

Default FAT Cluster Sizes

Organization of an FAT Volume

Boot Sector	Reserved Sectors	FAT 1	FAT 2 (Duplicate)	Root Folder	Other Folders and All Files
-------------	------------------	-------	-------------------	-------------	-----------------------------

Boot Sector

- Layout of the volume
- File system structure
- Boot code

Root Folder

- Describes the files and folders in the root of the partition

FAT 1

- Original FAT

Other Folder and All Files

- Contains the data for the files and folders within the file system.

FAT 2 (Duplicate)

- Backup copy of FAT

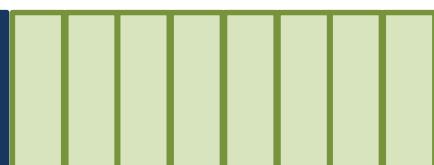
- Stores basic info about the file system
- FAT version, location of boot files
- Total number of blocks
- Index of the root directory in the FAT

- File allocation table (FAT)
- Marks which blocks are free or in-use
- **Linked-list structure** to manage large files

- Store file and directory data
- Each block is a fixed size (4KB – 64KB)
- Files may span multiple blocks

Disk

Super Block



FAT Boot Sector

- FAT Boot Sector
 - Locate at the first logical sector of each partition
 - Created when you format a volume
 - End with a 2-byte sector marker (always 0x55AA)
- Component
 - An x86-based CPU jump instruction
 - Original Equipment Manufacturer Identification (OEM ID)
 - BIOS Parameter Block (BPB)
 - Extended BPB
 - Executable boot code

FAT 1 and FAT 2

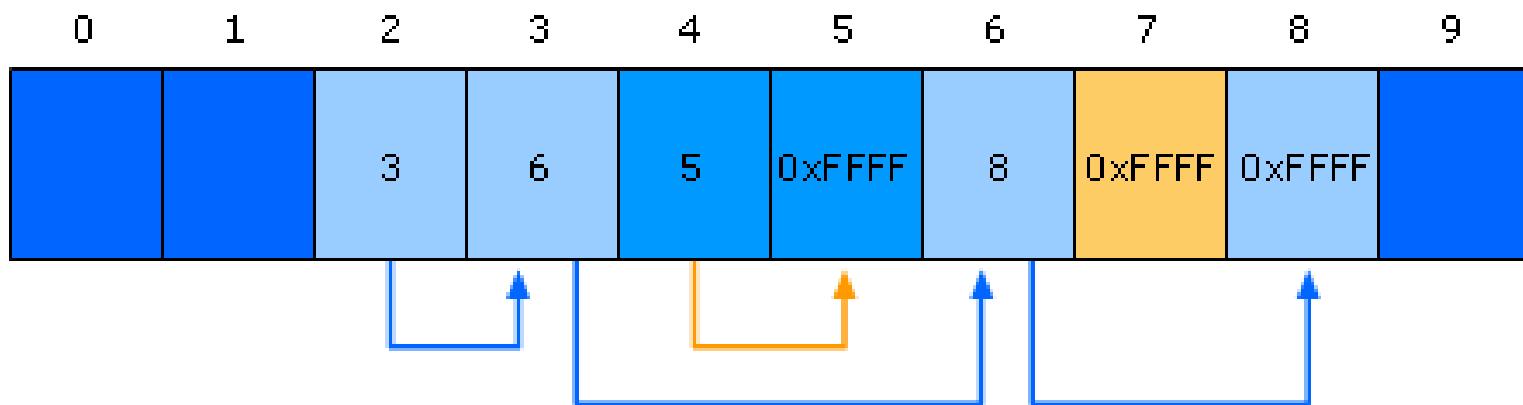
- File Allocation Table (FAT)
 - Identifies all clusters as
 - unused, cluster in use by a file, bad cluster, last cluster in a file
 - FAT 2 is used for consistency-checking program

A Sample of FAT

00009199616	F8 FF FF OF	FF FF FF FF	FF FF FF OF	FF FF FF OF	0yy
00009199632	FF FF FF OF	06 00 00 00	07 00 00 00	08 00 00 00	yyy
00009199648	09 00 00 00	FF FF FF OF	00 00 00 00	00 00 00 00	yy
00009199664	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	
00009199680	FF FF FF OF	12 00 00 00	13 00 00 00	14 00 00 00	yyy
00009199696	15 00 00 00	FF FF FF OF	17 00 00 00	FF FF FF OF	yy
00009199712	FF FF FF OF	yyy			
00009199728	FF FF FF OF	yy			
00009199744	FF FF FF OF	22 00 00 00	FF FF FF OF	24 00 00 00	yyy
00009199760	FF FF FF OF	26 00 00 00	FF FF FF OF	28 00 00 00	yyy
00009199776	FF FF FF OF	2A 00 00 00	FF FF FF OF	2C 00 00 00	yy
00009199792	FF FF FF OF	2E 00 00 00	FF FF FF OF	30 00 00 00	yy
00009199808	FF FF FF OF	32 00 00 00	FF FF FF OF	34 00 00 00	yy
00009199824	35 00 00 00	FF FF FF OF	37 00 00 00	FF FF FF OF	5 y
00009199840	39 00 00 00	FF FF FF OF	3B 00 00 00	FF FF FF OF	9 y
00009199856	FF FF FF OF	3E 00 00 00	3F 00 00 00	40 00 00 00	yyy
00009199872	41 00 00 00	42 00 00 00	43 00 00 00	44 00 00 00	A
00009199888	45 00 00 00	46 00 00 00	FF FF FF OF	FF FF FF OF	E
00009199904	49 00 00 00	4A 00 00 00	4B 00 00 00	4C 00 00 00	I
00009199920	4D 00 00 00	4E 00 00 00	FF FF FF OF	50 00 00 00	M

File Processing on FAT Clusters

- Store file information in FAT
 - Continues to store file info in the next available cluster
 - when the file requires space greater than the cluster's size
 - Three Files: 1 with (2-3-6-8), 2 with (4-5), 3 with (7)



FAT Root Folder

- FAT Root Folder Structure

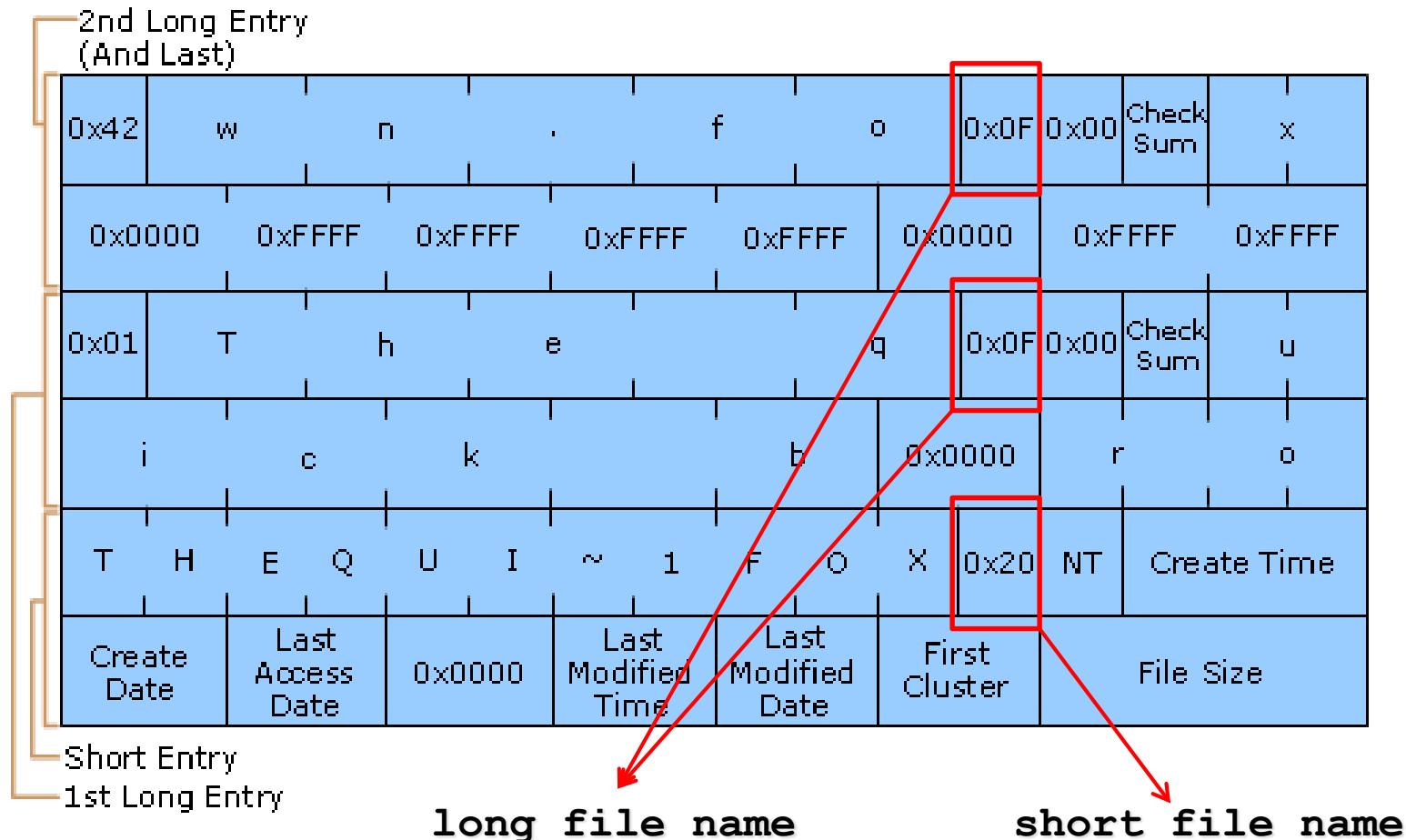
Root Folder Entry	Size	Description
Name	11B	Name in 8.3 format
Attribute Byte	1B	Information about entry (e.g. archive, system, hidden, and read-only)
Creation Time and Date	5B	Time and date file was created
Last Access Date	2B	Date file was last accessed
Last Modification Time and Date	4B	Time and date file was last modified
First Cluster	2B	Starting cluster number in the file allocation table
File Size	4B	Size of the file

File Naming

- File Names in Windows Server 2003
 - Support long file name, up to 255 characters
 - Generate a 8.3 short file name
- Support Long File Name
 - The main folder entry stores 8.3 short file name
 - The secondary folder entries store long file name
 - Each stores 13 characters in Unicode

File Naming (cont.)

e.g. file with “The quick brown.fox” and “Thequi~1.fox”



FAT32 Long File Name

- Using 0xF in directory entry
 - Means this entry is part of long file name
 - 0xF is not used in DOS/WIN32 (consider as invalid)
- Using *unicode* for storage
 - 2 bytes (64 bits)
- The short (8.3) file name is still there
 - The system cannot work without short name

Short Name Generation

- What if there is already a file with name THEQUI~1FOX?
 - Then use THEQUI~2FOX
 - If still conflict, using THEQUI~3FOX
 - If still conflict, using ...
 - If still conflict, using T~999999FOX
 - If still conflict, error

Question

- What is the process of searching a file by its long name?

File Name Search in ExFAT

- Search name by hash value first
 - Hash the upper case version of the file name
 - Each record in the directory is searched by comparing the hash value
 - When a match is found, the filenames are compared to ensure that the proper file was located in case of collisions
 - Only two characters have to be compared, why?

FAT: The Good and the Bad

- The Good – FAT supports:
 - Hierarchical tree of directories and files
 - Variable length files
 - Basic file and directory meta-data
- The Bad
 - At most, FAT32 supports 2TB disks
 - Locating free chunks requires scanning the entire FAT
 - Prone to internal and external fragmentation
 - Large blocks → internal fragmentation
 - **Reads require a lot of random seeking**

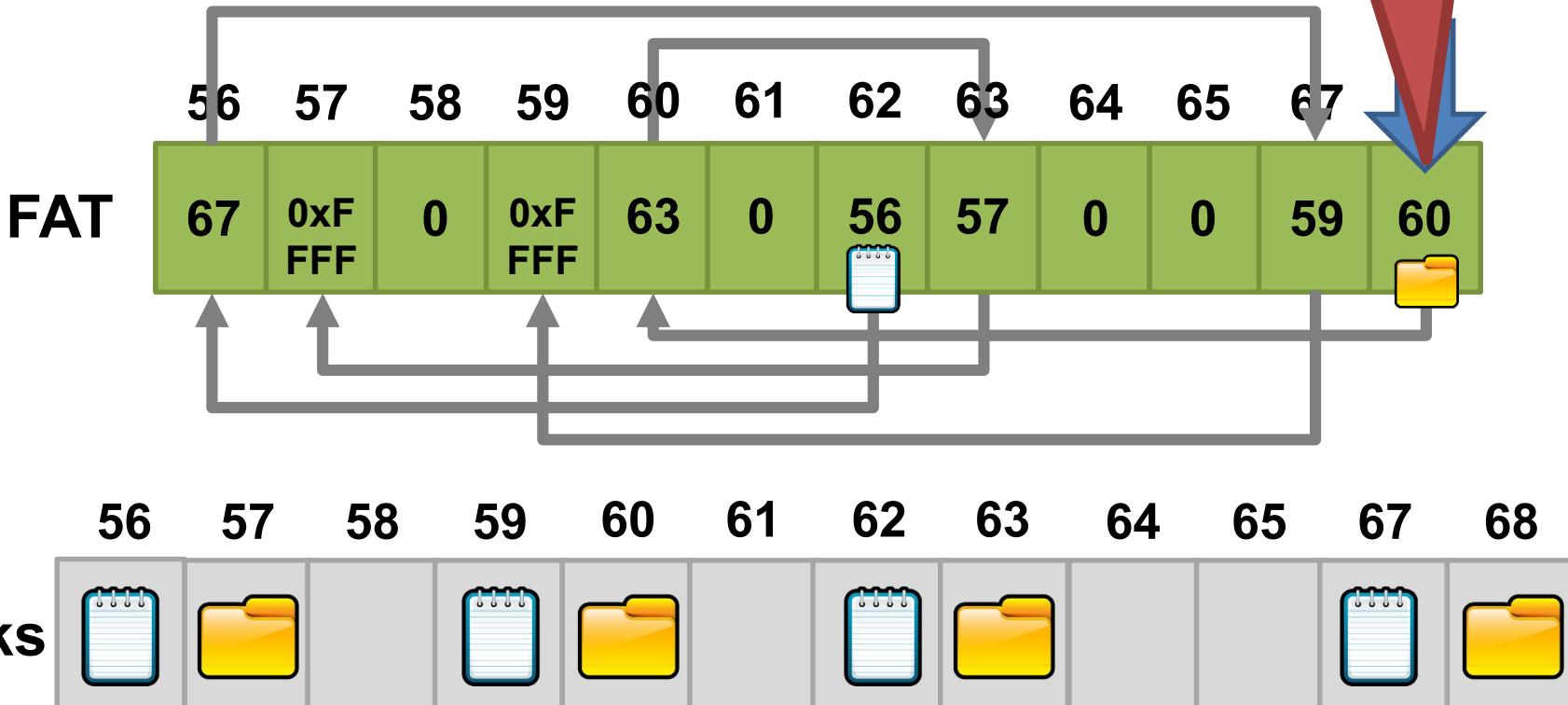
Lots of Seek!

- Consider the following code:

```
int fd = open("my_file.txt", "r");
```

```
int r = read(fd, buffer, 1024 * 4 * 4); // 4x4KB blocks
```

FAT may have very low spatial locality, thus a lot of random seeking



NTFS

NTFS Clusters

- Cluster: smallest allocated disk space to hold file
 - Sequentially logical cluster numbers from the beginning of the partition
 - Clusters start at sector ZERO (*different to FAT*)
 - Floppy disks do not use NTFS
 - Different cluster sizes depending on volume size

Volume Size	NTFS Cluster Size
7MB ~ 512MB	512 bytes
513MB ~ 1GB	1KB
1GB ~ 2GB	2KB
2GB ~ 2TB	4KB

Default NTFS Cluster Sizes

Organization of an NTFS Volume

NTFS Boot Sector	Master File Table	File System Data	Master File Table Copy
-----------------------------	------------------------------	-----------------------------	-----------------------------------

NTFS Boot Sector

- Layout of the volume
- File system structure
- Boot code

File System Data

- Data not contained within MFT

Master File Table

- Attributes of files
- Attributes of folders

(Most Important)

Master File Table Copy

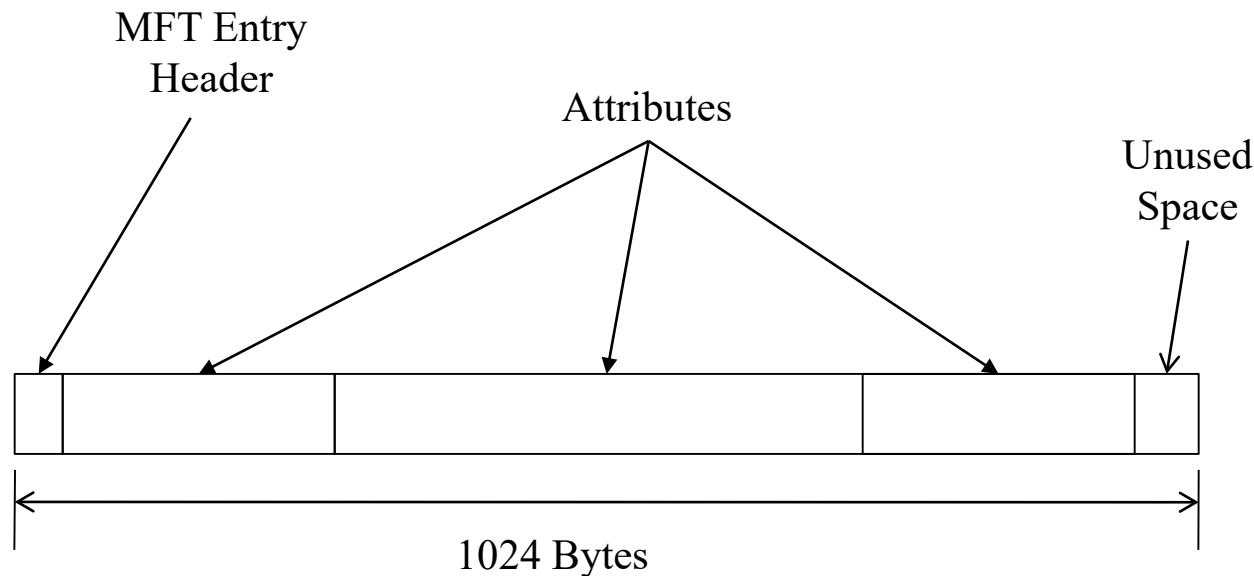
- a copy of MFT

Master File Table

- Master File Table (MFT)
 - A relational database
 - Rows: file records
 - Columns: file attributes
 - Contains at least one entry for every file
- Metadata files
 - Describe the MFT
 - The first 16 records of MFT for metadata files
 - Begin with a dollar sign, e.g., \$Mft, \$MftMirr, \$LogFile, ...

MFT

- The MFT is an array of file records
- Each record is 1024 bytes
- The first record in the MFT is for the MFT itself
- The name of the MFT is \$MFT
- The first 16 records in the MFT are reserved for metadata files



Metadata Files in MFT

System File	Name	Rec	Purpose of File
Master File Table	\$Mft	0	Contains file record for each file and folder on a NTFS volume.
Master File Table Mirror	\$MftMirr	1	Duplicate image of the first four records of MFT
Log File	\$LogFile	2	Restores metadata for fast recovery
Volume File	\$Volume	3	Contains information about the volume
Attribute Definitions	\$AttrDef	4	Lists attribute names, numbers, and descriptions
Root File Name Index	.	5	The root file
Cluster Bitmap	\$Bitmap	6	Represents the volume by showing free and unused clusters.

Metadata Files in MFT (cont.)

System File	Name	Rec	Purpose of File
Boot Sector	\$Boot	7	Includes the BPB and additional bootstrap loader code
Bad Cluster file	\$BadClus	8	Contains bad clusters for a volume
Security File	\$Secure	9	Contains unique security descriptors for all files
Upcase Table	\$Upcase	10	Used to match Unicode uppercase characters
NTFS Extension File	\$Extend	11	Optional extensions such as quotas, reparse point data ...
		12-15	Reserved for future use

MFT Zone

- MFT contains a record for each file and folder on the volume
 - File and folder records are 1KB each
- MFT Zone
 - Exclusive use of the MFT
 - Prevent fragmentation
 - Dedicate 12.5% of volume by default (!)
 - On demand allocation for additional MFT zone

NTFS File Record Attributes

- File Attributes
 - View each file and folder as a set of file attributes
 - Resident Attributes vs. Nonresident Attributes
 - Small files and folders are entirely contained within the file's MFT record (*typically, less than 900 bytes*)
 - Types:
 - Standard Info: access mode, timestamp, link count ...
 - Attribute List: locations of all nonresident attributes
 - File Name: long and short file names
 - Data / Index: data of file or index of folder
 - Object ID: volume-unique file identifier
 - ...

NTFS File Record Attributes

- Index of Folder
 - Folder records contain index information (directory entry)
 - Organized B-tree structure for large folders
- Last Access Time
 - The most up-to-date LAT is always stored in memory
 - Written to disk
 - WHERE: *file's attribute AND directory entry for the file*
 - WHEN: *more than an hour OR update other file attributes*

File Naming

- File Names in Windows Server 2003
 - Support long file name, up to 255 characters
 - NTFS generate a 8.3 short file name
- HOW to generate short file name
 - Deletes all of unsupported characters (e.g. space)
 - Deletes all extra periods (*abc.def.txt*)
 - Truncates the file name to six characters and appends a tilde(~) and a number
 - Translates all characters in file name and extension to uppercase

“This is test 1.txt” -> “THISIS~1.TXT”

Compression of Files and Folders

- NTFS Compression
 - Implemented within NTFS
 - Compression only in disk
 - Uncompress before moving data to memory
- Moving and Copying Files or Folders
 - Change compression state
 - Add overhead to the system

Hard Links

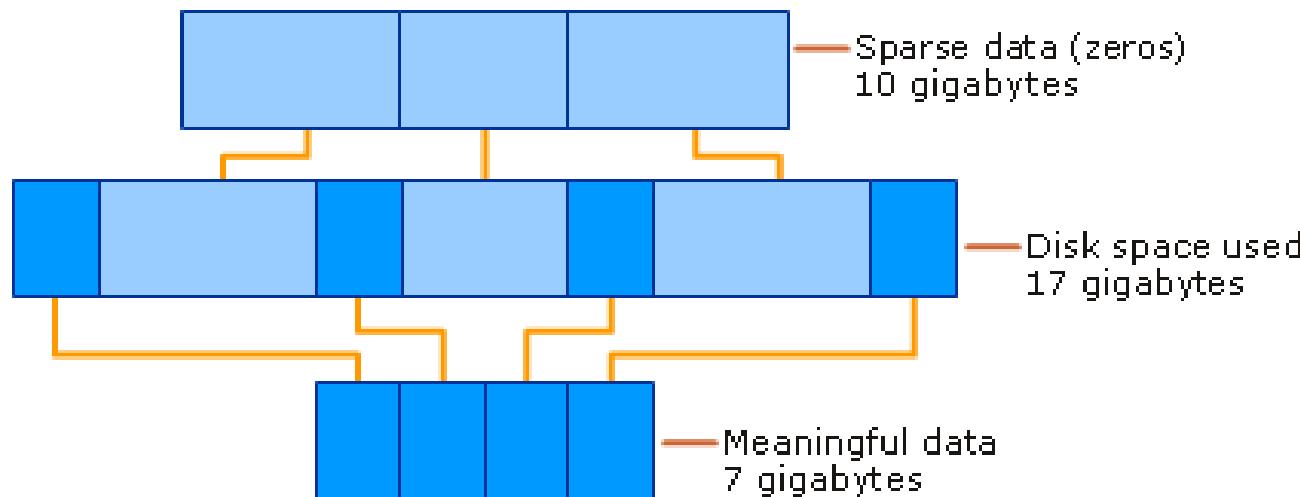
- Hard Links
 - Add a directory entry for the hard link without duplicating the original file
 - Application can modify a file by using any of its hard Links
 - *Cannot* give a file different security descriptors on a per-hard-link basis

Sparse Files

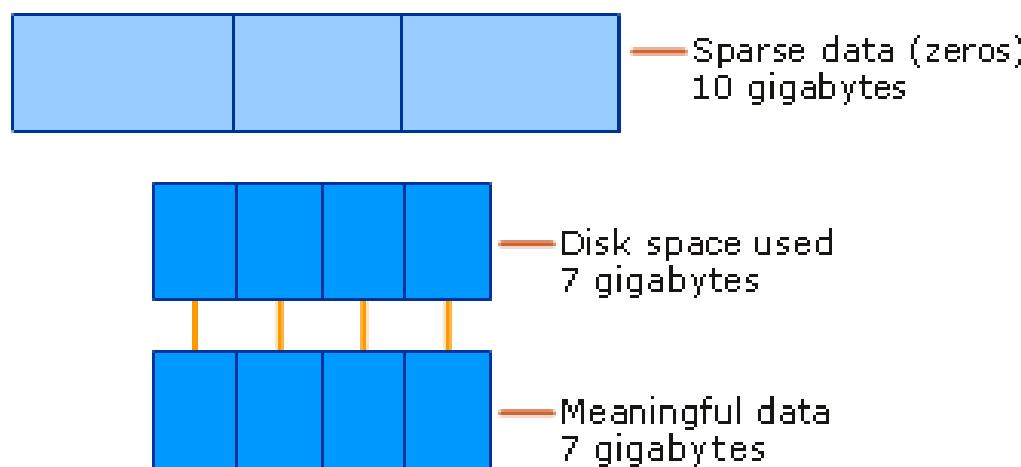
- Sparse Files
 - Focus on the file contain large sections of data composed of zeros
 - Store
 - Allocates disk clusters only for the data explicitly specified by the application
 - Non-specified ranges of the file are represented by non-allocated space on the disk
 - Read
 - From allocated ranges, return the data stored
 - From non-allocated ranges, return ZERO

Sparse Files (cont.)

Without Sparse File Attribute Set



With Sparse File Attribute Set

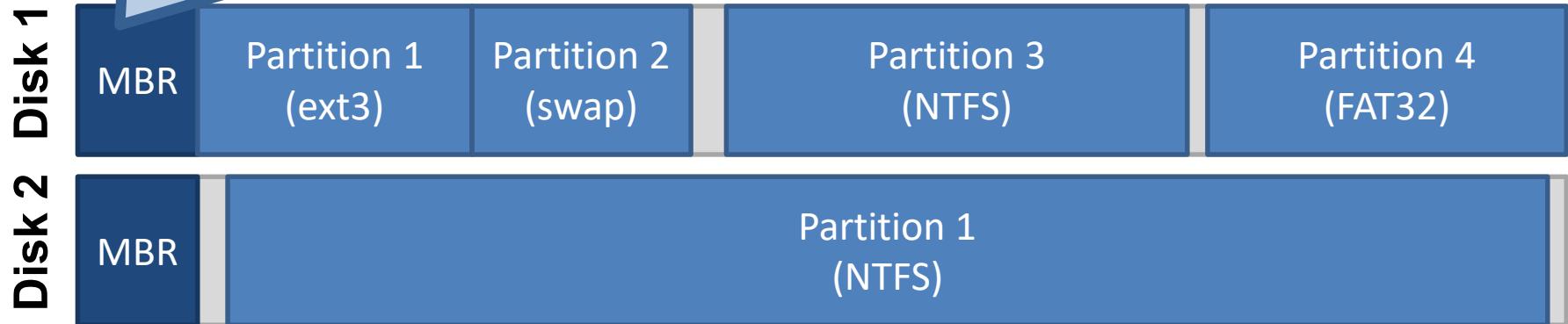


MBR & MOUNT

The Master Boot Record

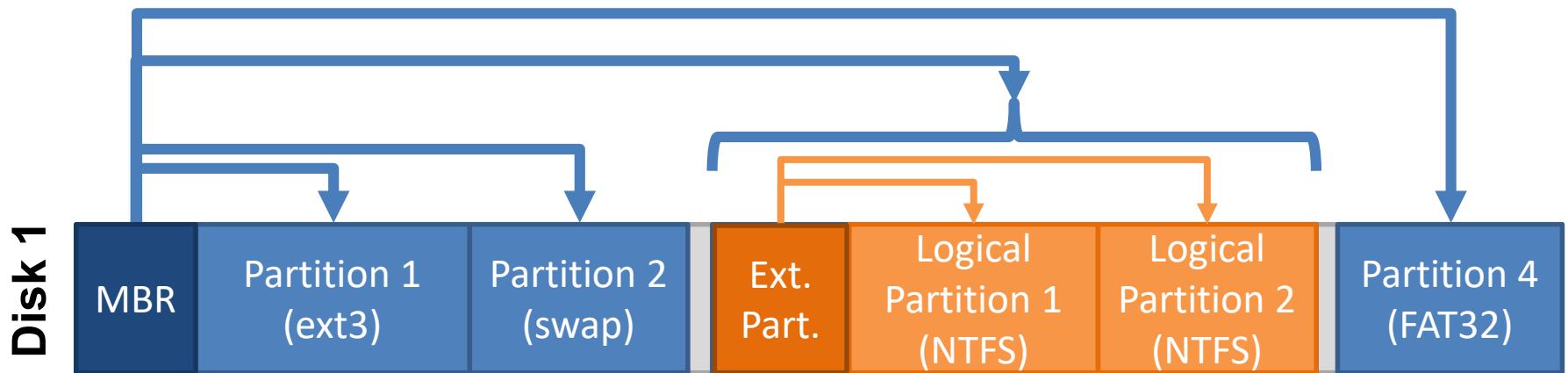
Address		Description	Size (Bytes)
Hex	Dec.		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2
		Total:	512

Includes the starting LBA and length of the partition



Extended Partitions

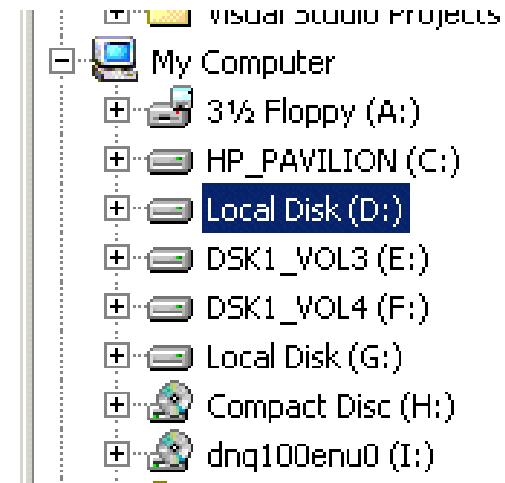
- In some cases, you may want >4 partitions
- Modern OSes support extended partitions



- Extended partitions may use OS-specific partition table formats (meta-data)
 - Thus, other OSes may not be able to read the logical partitions

Types of Root File Systems

- Windows exposes a multi-rooted system
 - Each device and partition is assigned a letter
 - Internally, a single root is maintained
- Linux has a single root
 - One partition is mounted as /
 - All other partitions are mounted somewhere under /
- Typically, the partition containing the kernel is mounted as / or C:



Mounting a File System

1. Read the **super block** for the target file system
 - Contains meta-data about the file system
 - Version, size, locations of key structures on disk, etc.
2. Determine the **mount point**
 - On Windows: pick a drive letter
 - On Linux: mount the new file system under a specific directory

Filesystem	Size	Used	Avail	Use%	Mounted on
/dev/sda5	127G	86G	42G	68%	/media/cbw/Data
/dev/sda4	61G	34G	27G	57%	/media/cbw/Windows
/dev/sdb1	1.9G	352K	1.9G	1%	/media/cbw/NDSS-2013

Mount isn't Just for Bootup

- When you plug storage devices into your running system, mount is executed in the background
- Example: plugging in a USB stick
- Why “safely eject” a device?
 - Flush cached writes to that device
 - Cleanly unmount the file system on that device



Homework

- Please learn about **btrfs**, use your own words to describe its system architecture, and states its pros and cons (you can compare it with ext4/fat/ntfs)
- (optional) you can also check ReiserFS and ZFS, they also have very interesting features

Flash FS

Yubin Xia

Transaction Semantics in xv6

begin_trans:

- need to indicate which group of writes must be atomic!
- lock -- xv6 allows only one transaction at a time

log_write:

- record sector #
- append buffer content to log
- leave modified block in buffer cache (but do not write)

commit_trans():

- record "done" and sector #s in log
- do the writes
- erase "done" from log

recovery:

- if log says "done": copy blocks from log to real locations on disk

Committing a transaction to disk

open a new transaction, for subsequent syscalls

mark transaction as done

wait for in-progress syscalls to stop()

(maybe it starts writing blocks, then waits, then writes again if needed)

write descriptor to log on disk w/ list of block #s

write each block from cache to log on disk

wait for all log writes to finish

append the commit record

now cached blocks allowed to go to homes on disk (but not forced)

Review: Ordered Mode vs Journaled Mode

journaling file content is slow, every data block written twice

perhaps not needed to keep FS internally consistent

can we just lazily write file content blocks?

no:

- if metadata updated first, crash may leave file pointing to blocks with someone else's data

ext3 ordered mode:

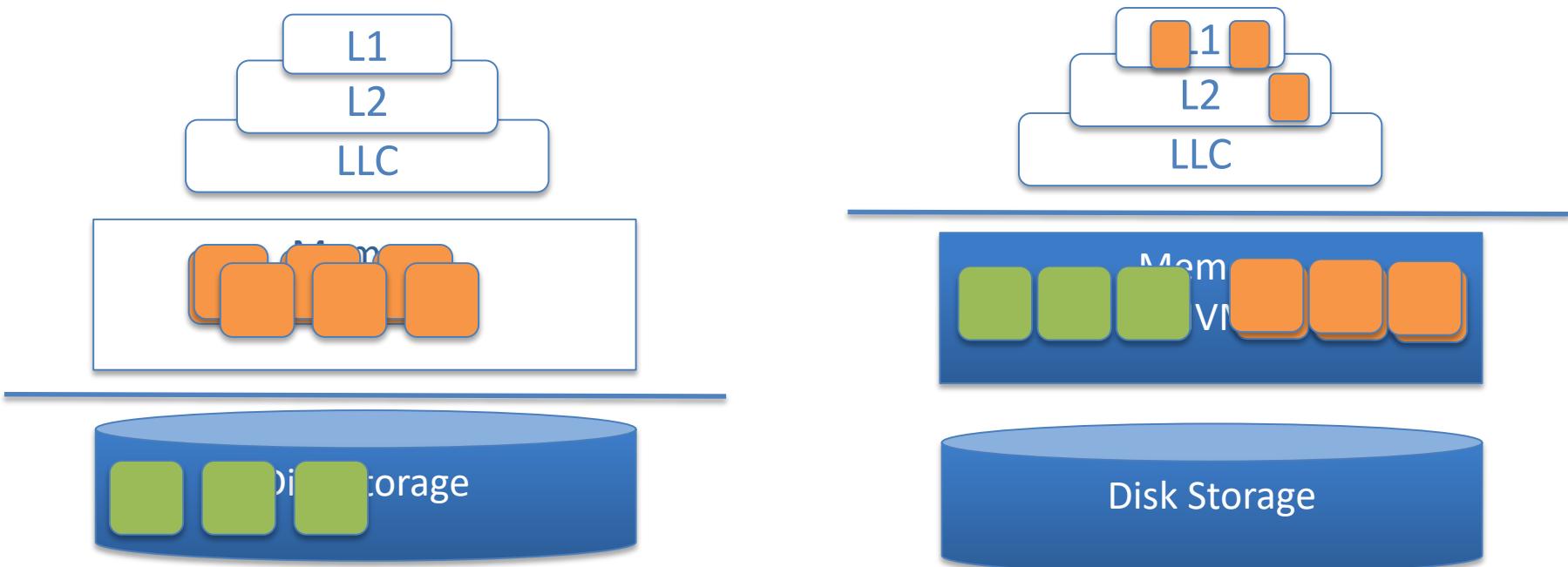
- write content block to disk before committing inode w/ new block #

- thus won't see stale data if there's a crash

most people use ext3 ordered mode

Similar Problem in NV-RAM

- Consider using NV-RAM as disk
 - CPU Cache now becomes the memory
 - Crash will lose data in cache (not written back)



Similar Problem in NV-RAM

- CPU has instructions to flush cache to memory
 - Clflush: cache line flush
 - Clflushopt: a similar version
 - Clwb: cache line write back
 - Pcommit: Persistence commit

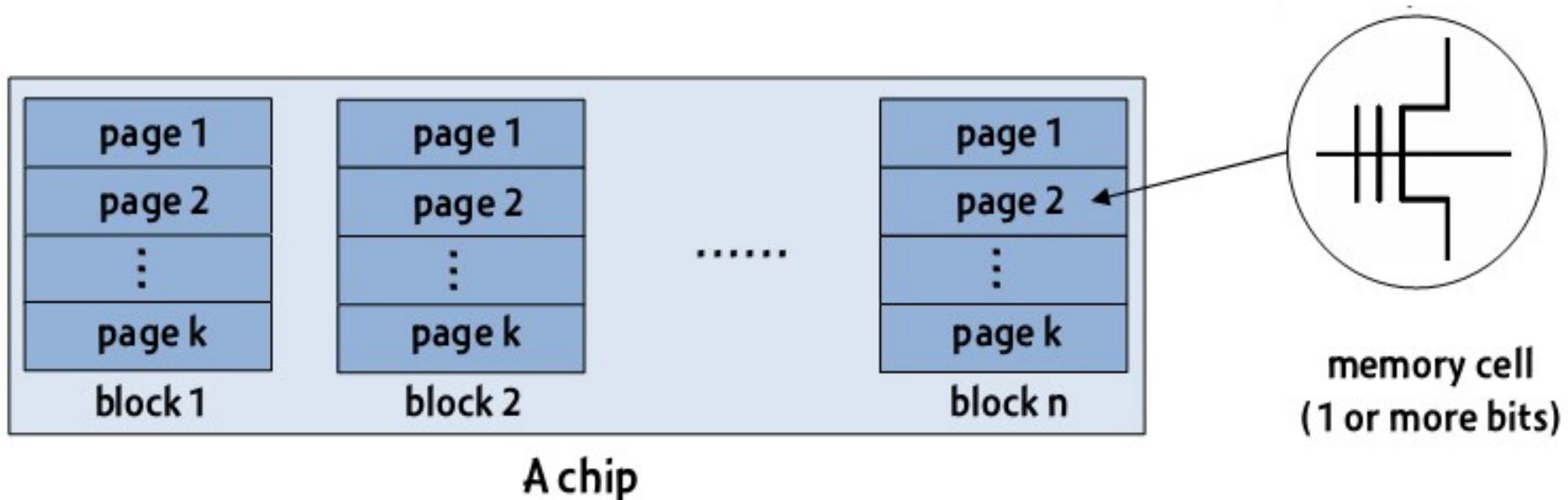
INTRO TO FLASH FILE SYSTEM

What is flash file system?

- Flash file system is designed for storing files on flash memory devices
- Can traditional file systems be used on flash file system?
 - Of course, NO.
- So, what is the big difference between flash disk and magnetic disk

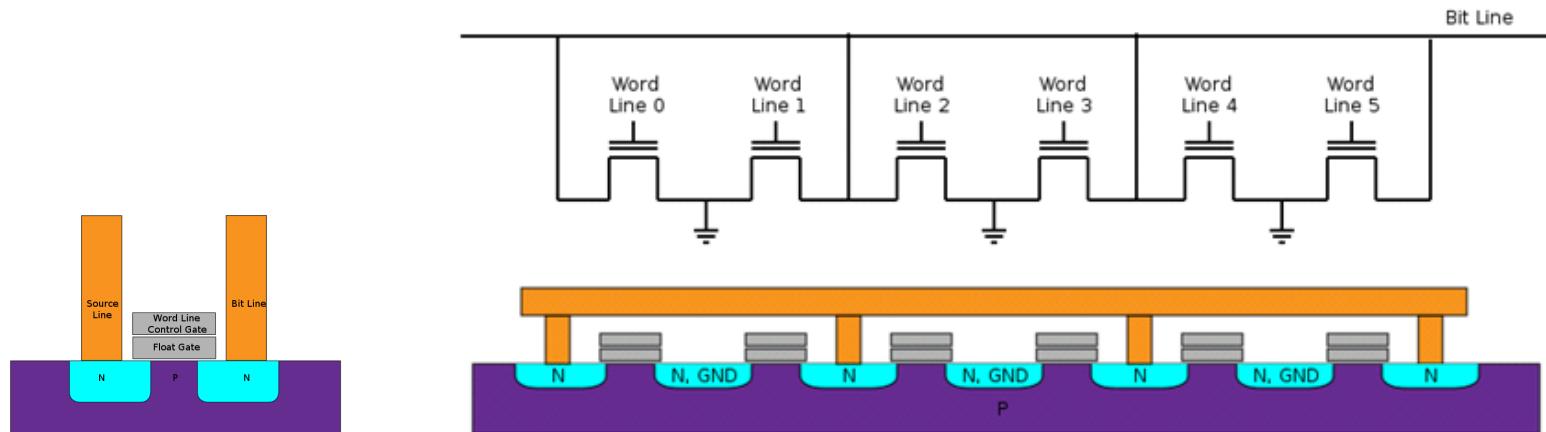
Flash disk organization (1/2)

- Flash disk organization
 - A **chip** (e.g. 1GB) => **blocks** (e.g. 512KB) => **pages** (e.g. 4KB) => **cells**



Flash disk organization (2/2)

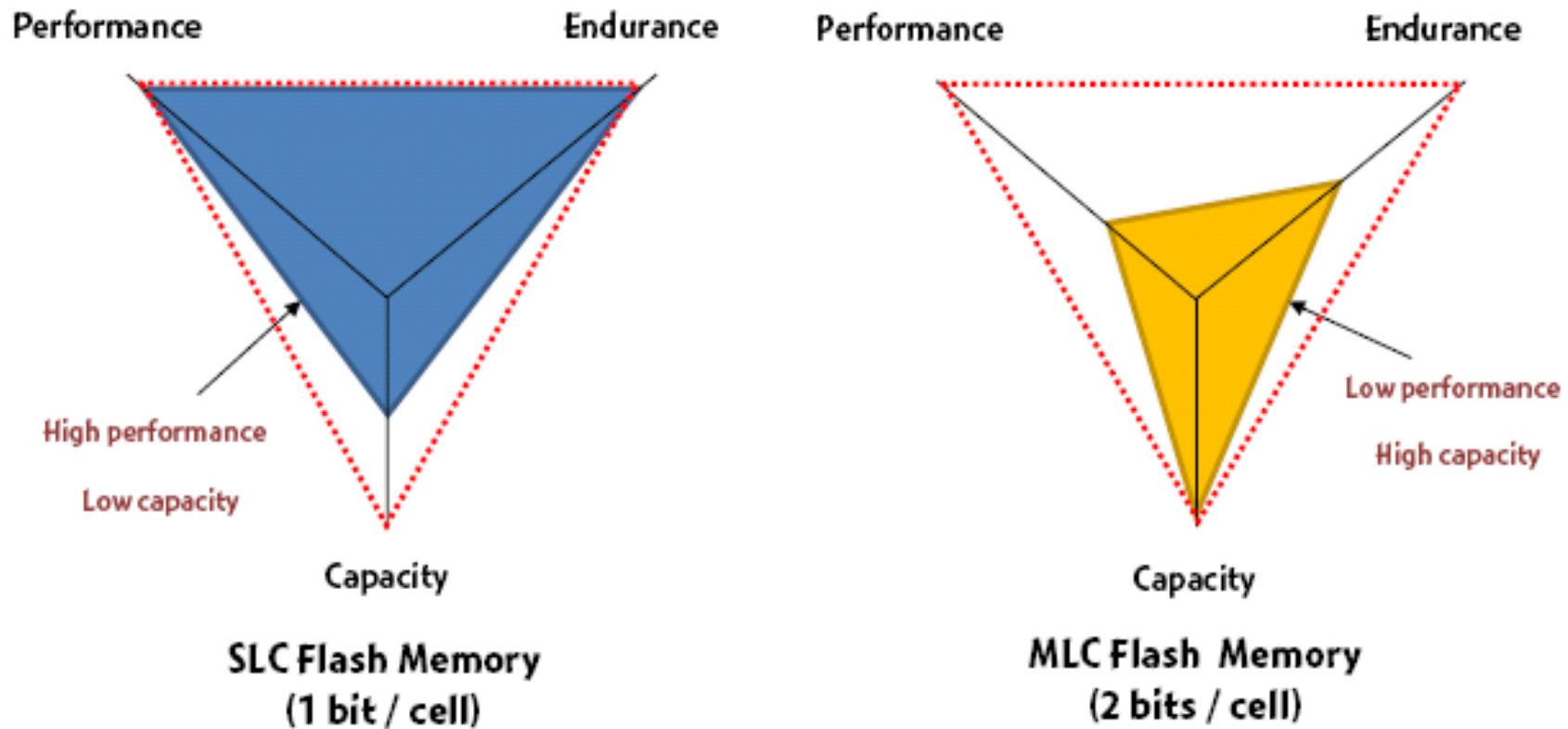
- Flash cell: a floating gate transistor
 - The number of electrons on the floating gate determines the threshold voltage V
 - The threshold voltage represents a logical bit value (0 or 1)



(Single-Level Cell, SLC)

(Multi-Level Cell, MLC)

Comparisons of SLC and MLC Flashes



Flash disk characters (1/2)

- Asymmetric **read/write** and **erase** operations
 - A “**page**” is a unit of read/write (8-16KB)
 - A “**block**” is a unit of erase (4-8MB)
- Physical restrictions
 - **Erase-before-write** restriction
 - The number of erase cycles allowed for each block is limited



Flash disk characteristics (2/2)

- Random access
 - Disk file system are optimized to avoid disk seeks whenever possible
 - Flash device impose no seek latency
- Wear leaving
 - Flash device tend to wear out when a single block is repeatedly overwritten
 - Designed to spread out writes evenly
- Heterogeneous cells
 - SLC (single level cell) and MLC (multi level cell)

Flash for File Storage

Simplest method – direct 1:1 mapping

- Good for read-only operations

- No wear levelling

- Very unsafe (on power loss)

Flash Translation Layer (FTL) keeps track of sectors

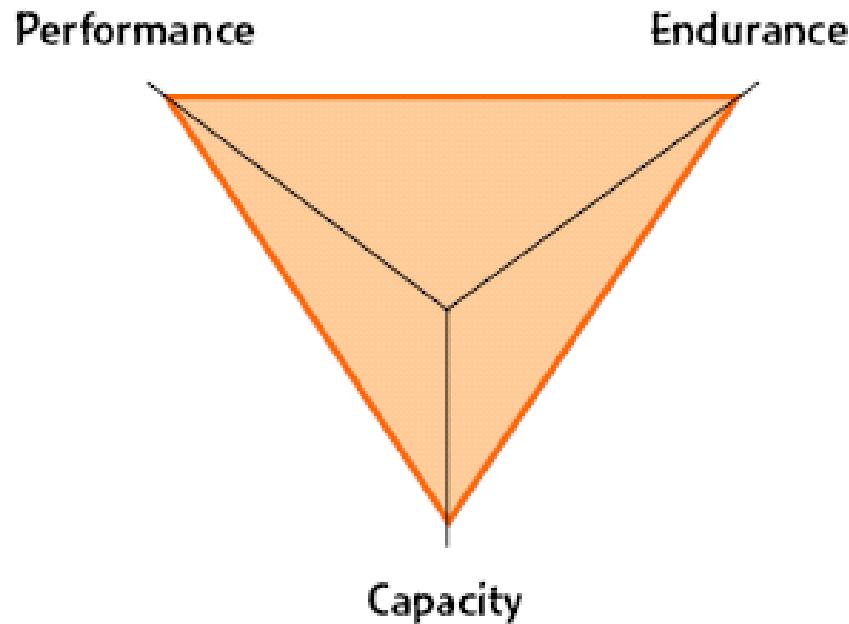
- Suitable for a writable FS

- Wear levelling, Reliable operation;

- But, one journalling FS on top of the other

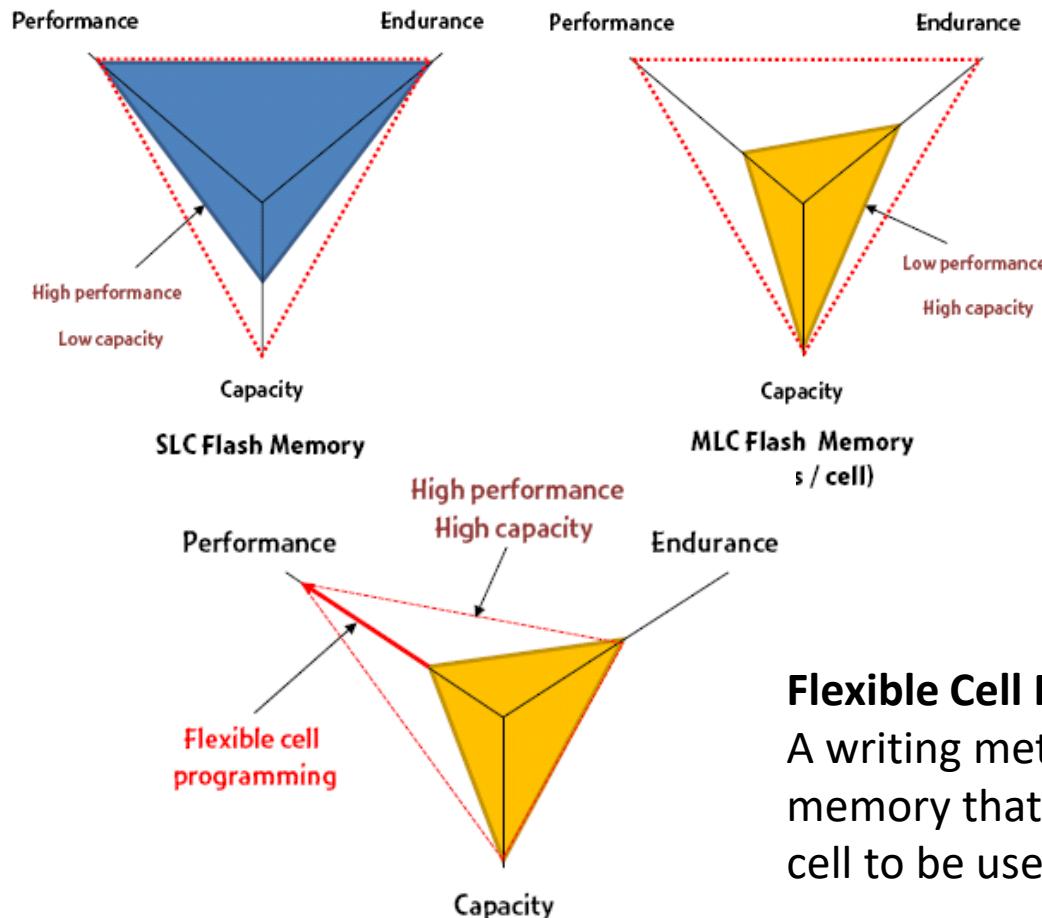
A real flash file system study - flexFS

- Consumers want to have a storage system with high performance, high capacity, and high endurance
- HOW ??



Flexible Cell Programming

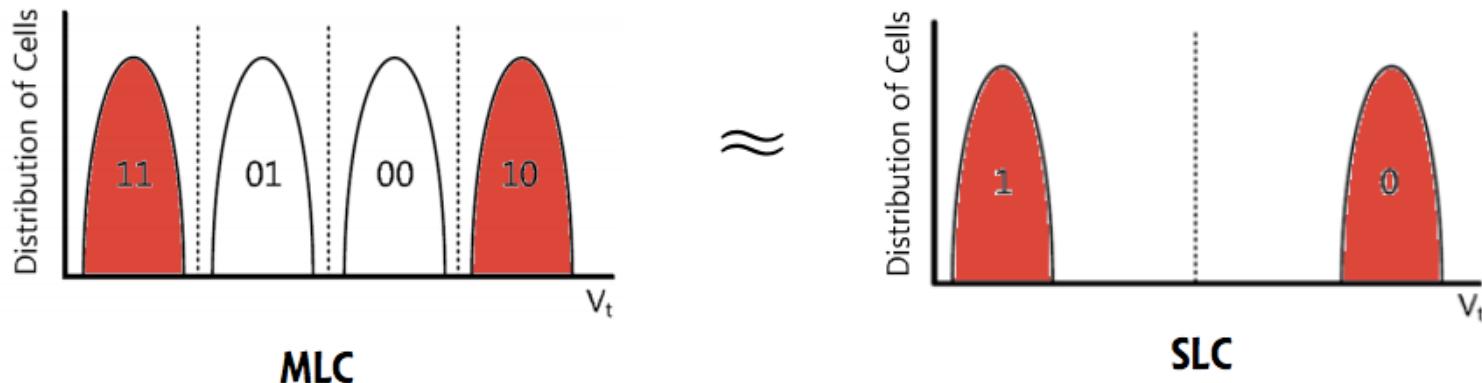
- Heterogeneous - Makes it possible to take benefits of two different types of NAND flash memory



Flexible Cell Programming:
A writing method of MLC flash memory that allows each memory cell to be used as SLC or MLC

Background

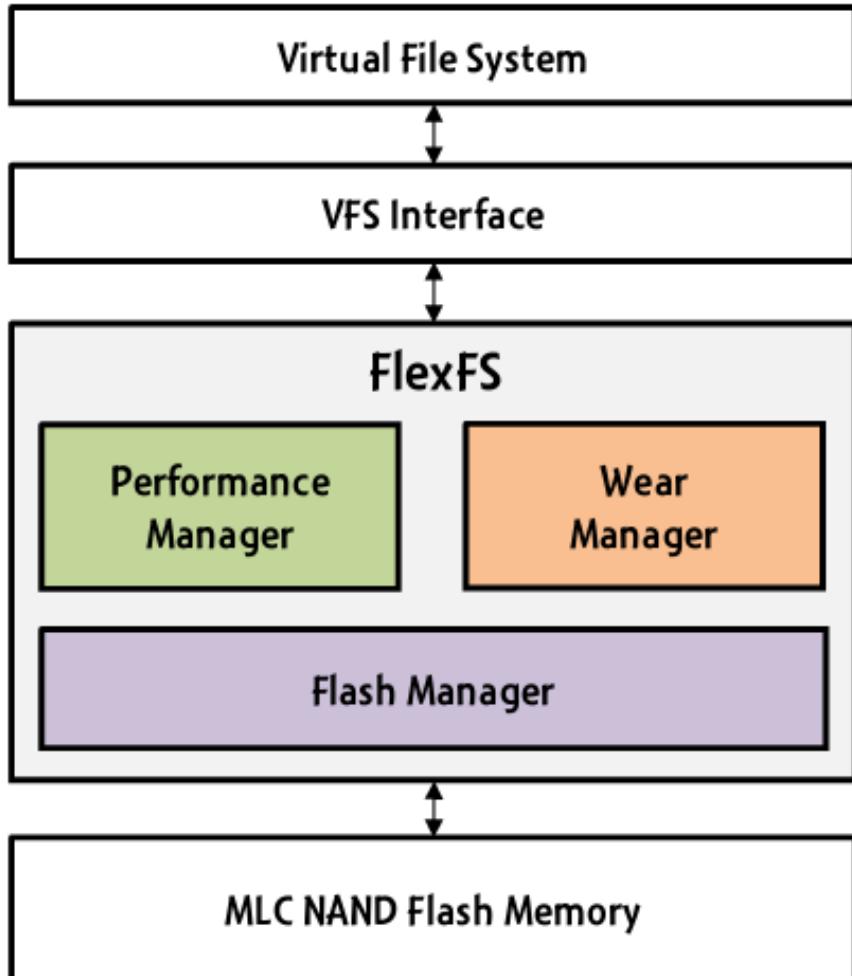
- (1) MLC programming method
 - Uses all four values of cell by writing data to both LSB and MSB bits
 - Low performance / High capacity (2 bits per cell)
- (2) SLC programming method
 - Uses only two values of cell by writing data to LSB bit (or MSB bit)
 - High performance / Low capacity (1 bit per cell)



The Approach

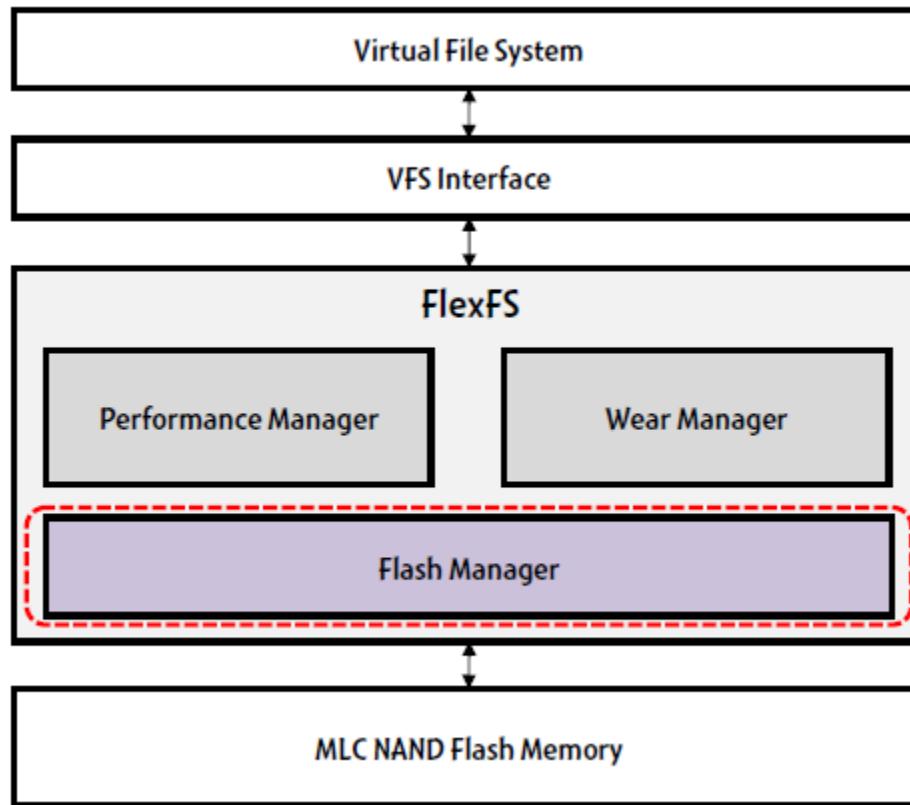
- Proposes a flash file system called FlexFS
 - Exploits **flexible cell programming**
 - Provides the **high performance** of SLC flash memory and the **capacity** of MLC flash memory
 - Provides a mechanism that copes with a poor **wear** characteristic of MLC flash memory
 - Designed for mobile systems, such as mobile phones

Architecture of FlexFS



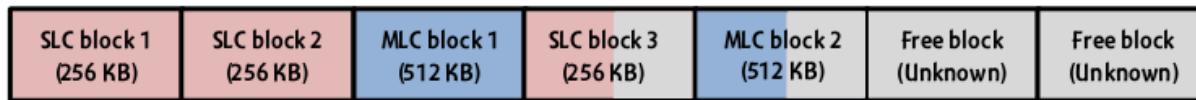
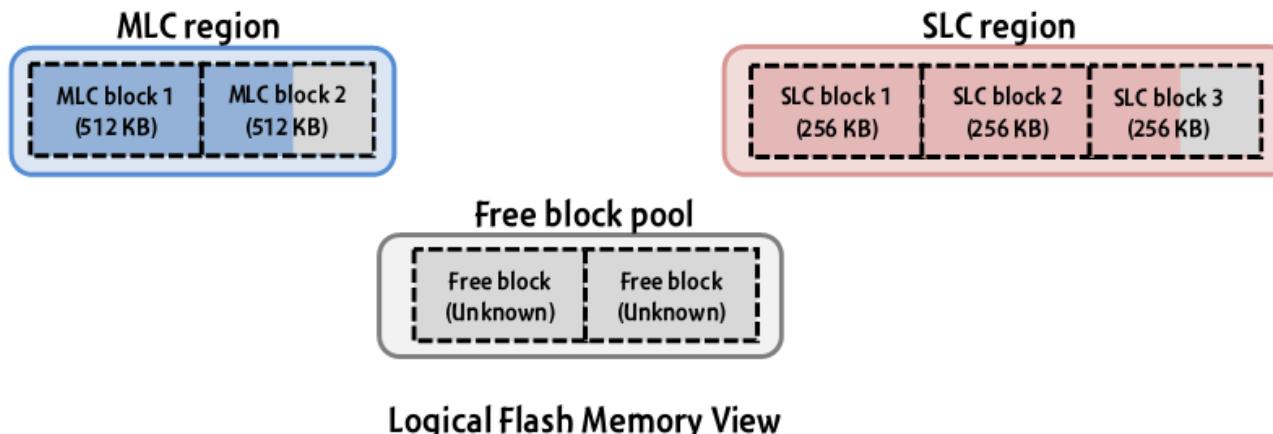
- Flash Manager
 - Manages heterogeneous cells
- Performance manager
 - Exploits I/O characteristics
 - To achieve the high performance and high capacity
- Wear manager
 - Guarantees a reasonable lifetime
 - Distributes erase cycles evenly

Overall Architecture



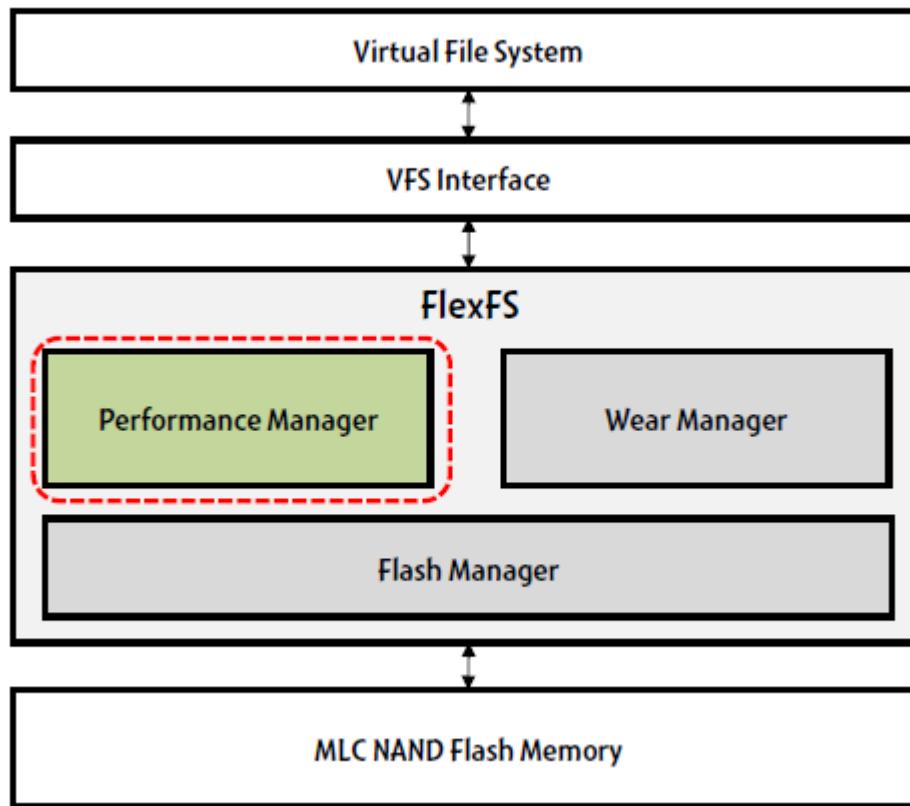
FlexFS - Flash Manager

- Handles Heterogeneous Cells
 - Three types of flash memory block: SLC block, MLC block, and free block
 - Manages them as two regions and one free block pool



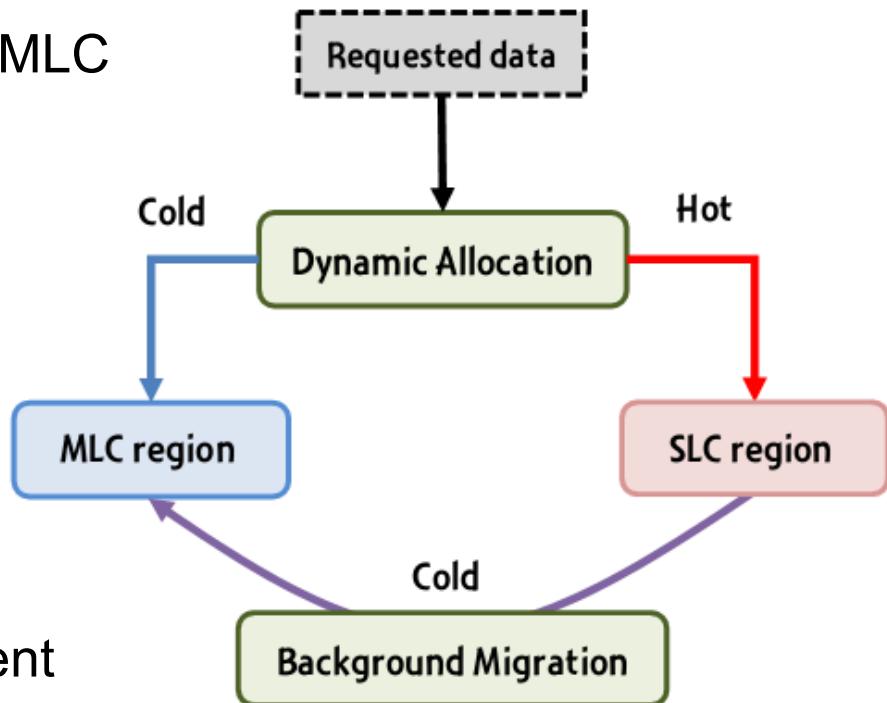
Physical Flash Memory View

Overall Architecture

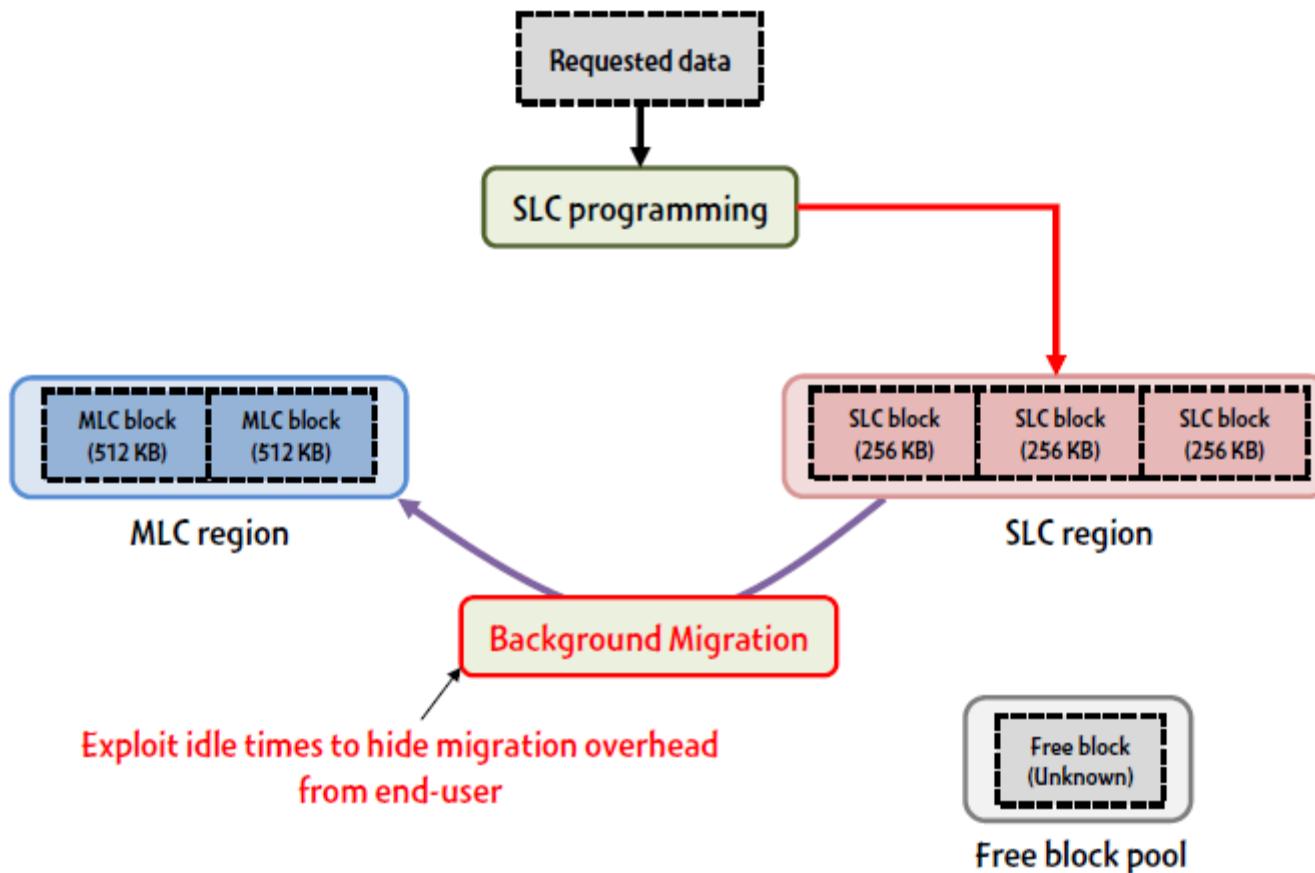


FlexFS – Performance Manager

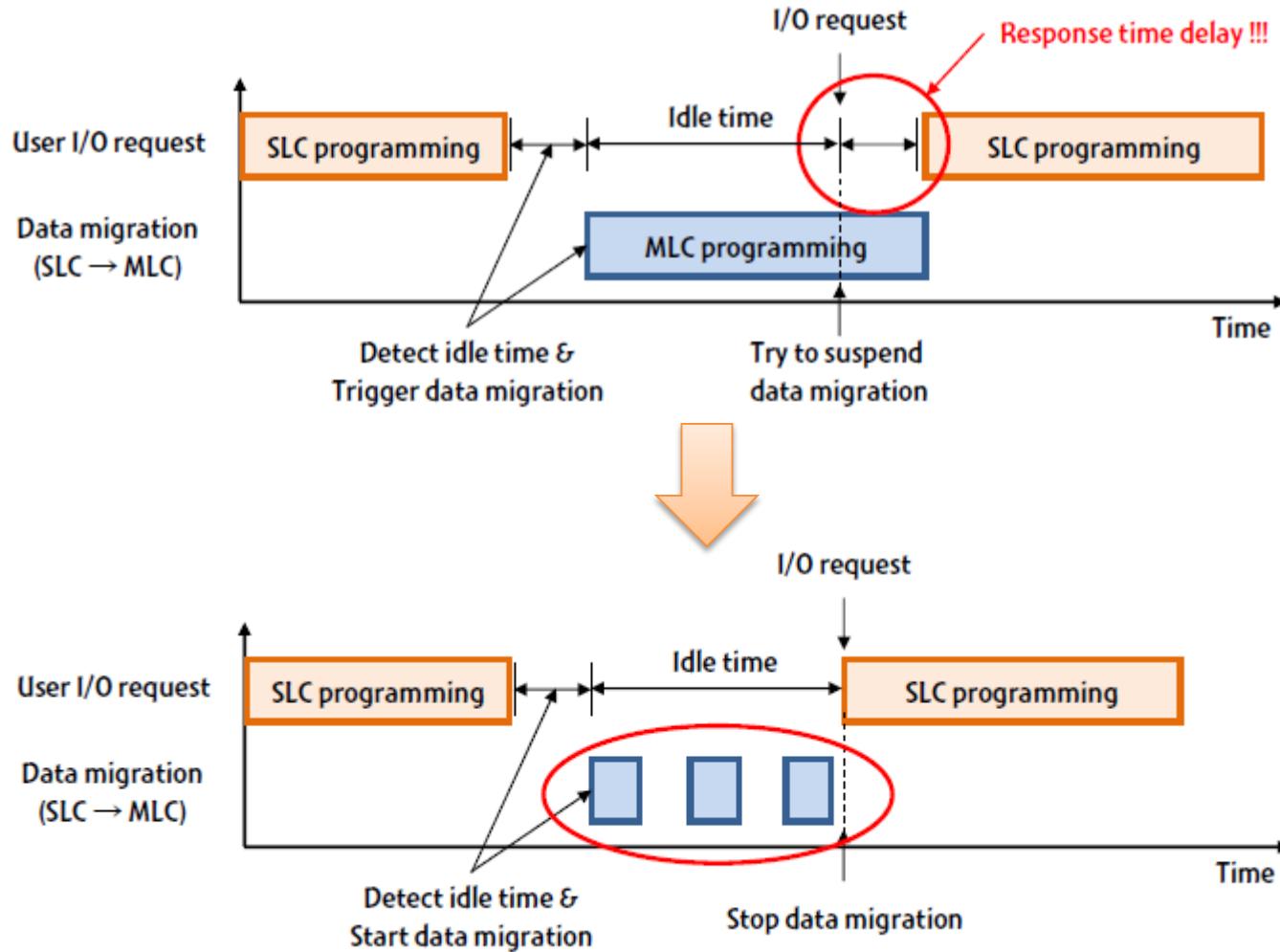
- Manages SLC and MLC regions
 - Provide SLC performance and MLC capacity
 - Exploits I/O characteristics, such as idle time and locality
- Three key techniques
 - Dynamic allocation
 - Background migration
 - Locality-aware data management



Baseline Approach



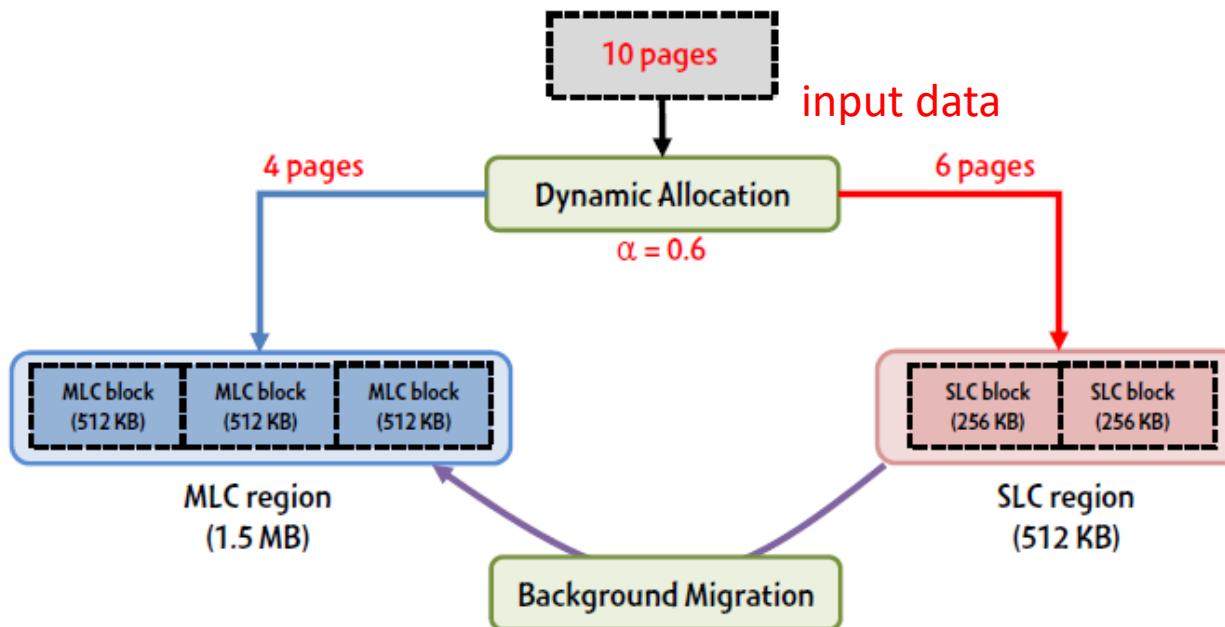
Background Migration



Dynamic Allocation

- Distributes the incoming data across two regions depending on α

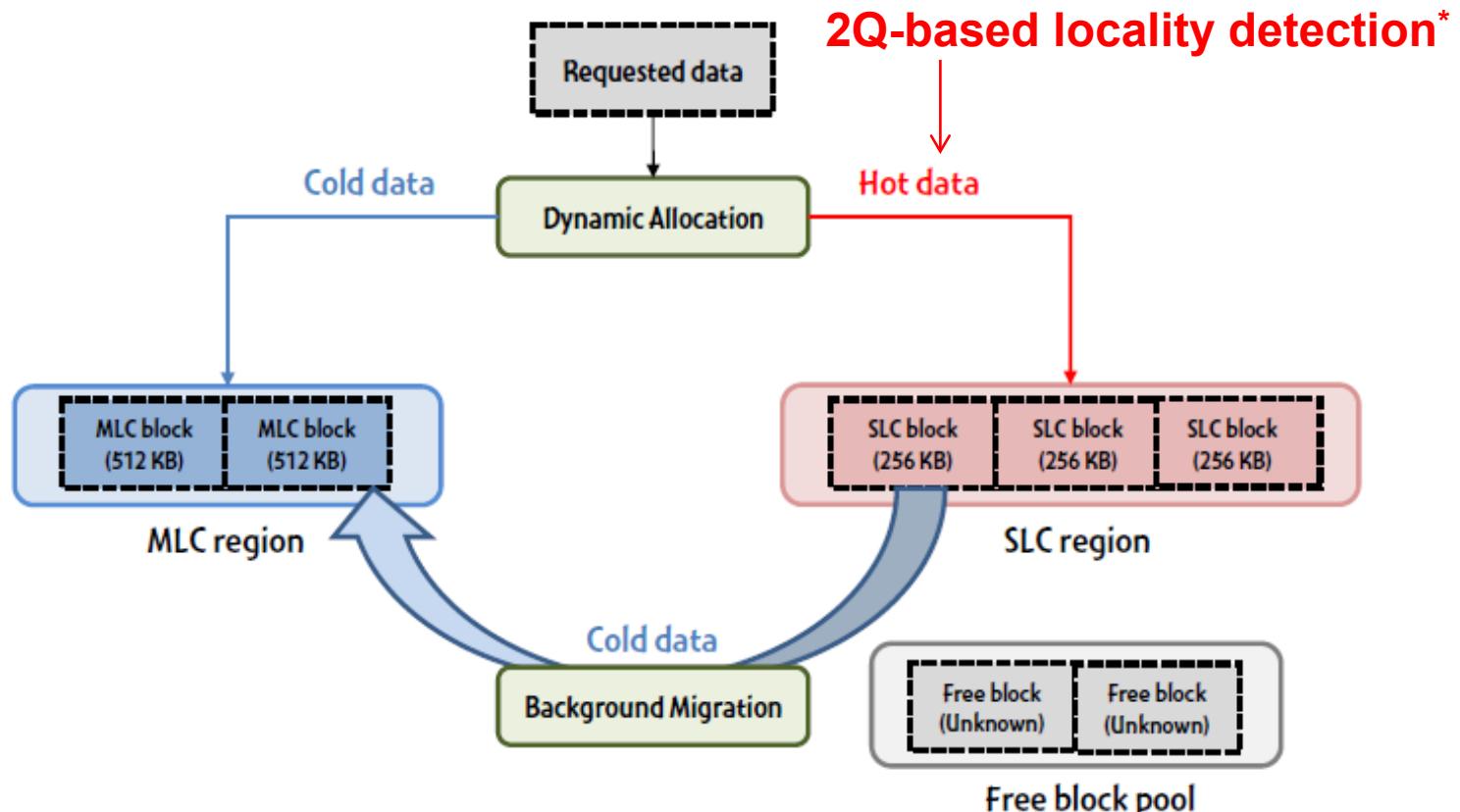
$$\alpha = \frac{T_{\text{predict}}}{N_p \cdot T_{\text{copy}}} \quad (\text{If } T_{\text{predict}} \geq N_p \cdot T_{\text{copy}}, \text{ then } \alpha = 1.0)$$



- T_{copy} is the time required to copy a single page from SLC to MLC
- N_p is the number of pages in SLC
- T_{predict} is the idle time of next time window predicted

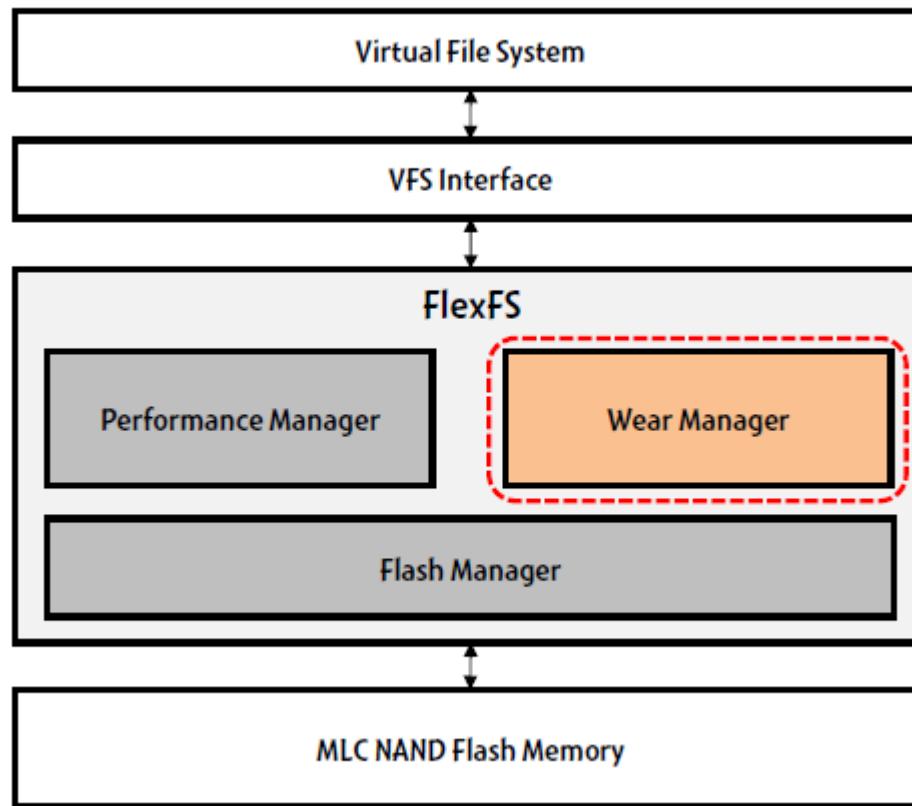
Locality-aware Data Management

- Data migration for hot data is unnecessary



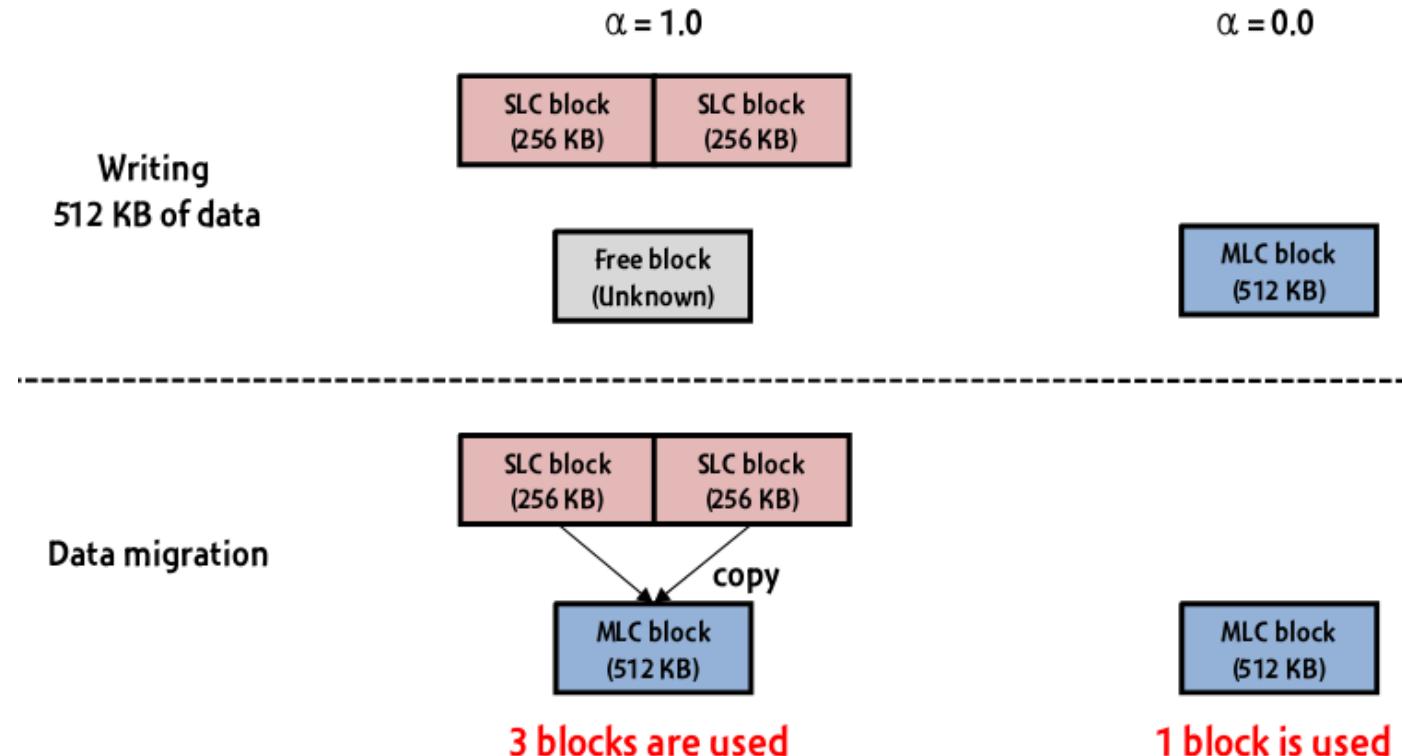
* E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," In Proceedings of the Conference on Management of Data (SIGMOD '93), May 1993.

Overall Architecture

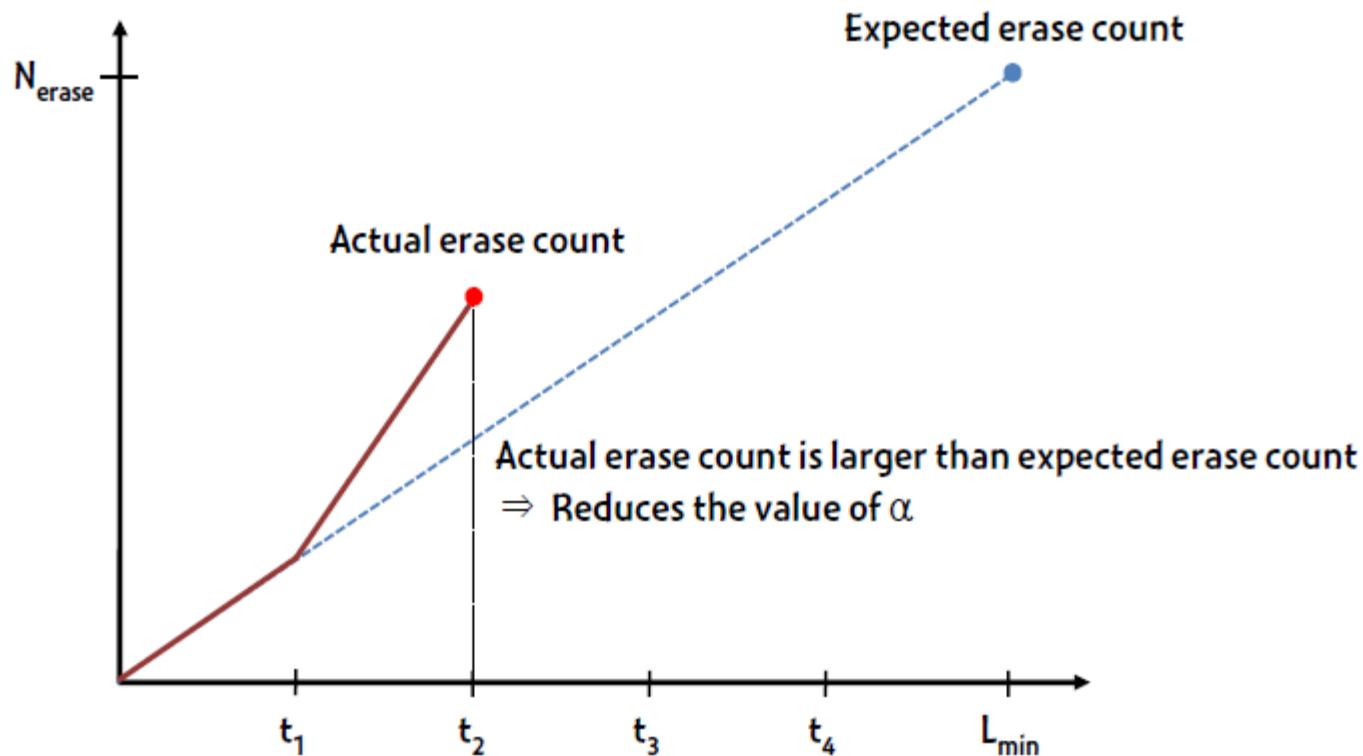


FlexFS– wearing rate control

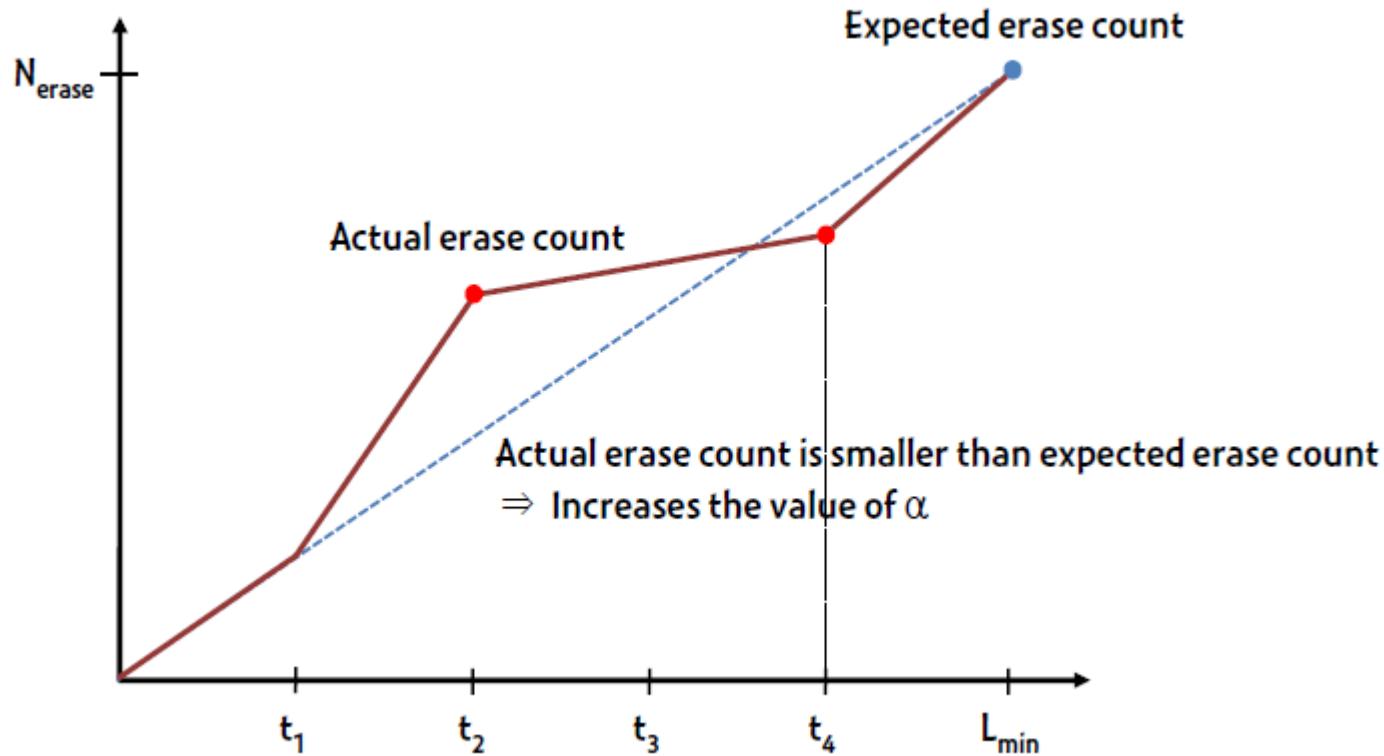
- How FlexFS control the wearing rate
 - The wearing rate is directly proportional to α



Wearing Rate Control

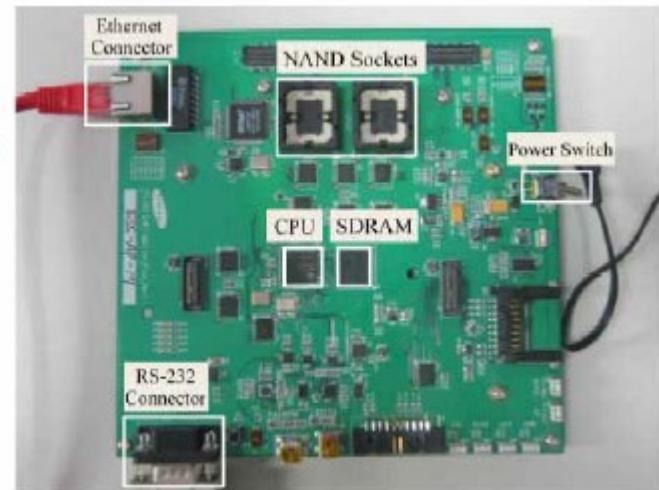
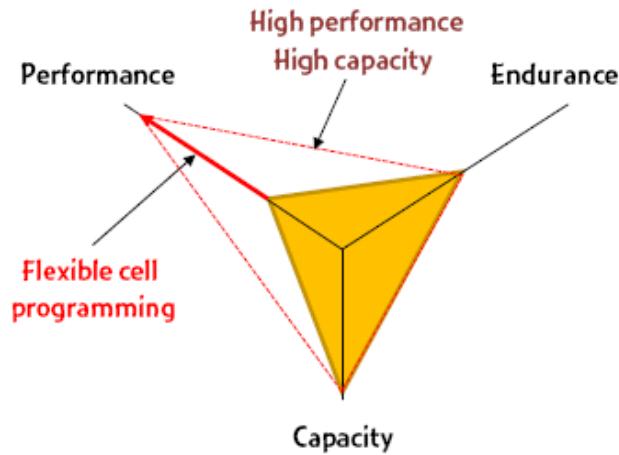


Wearing Rate Control



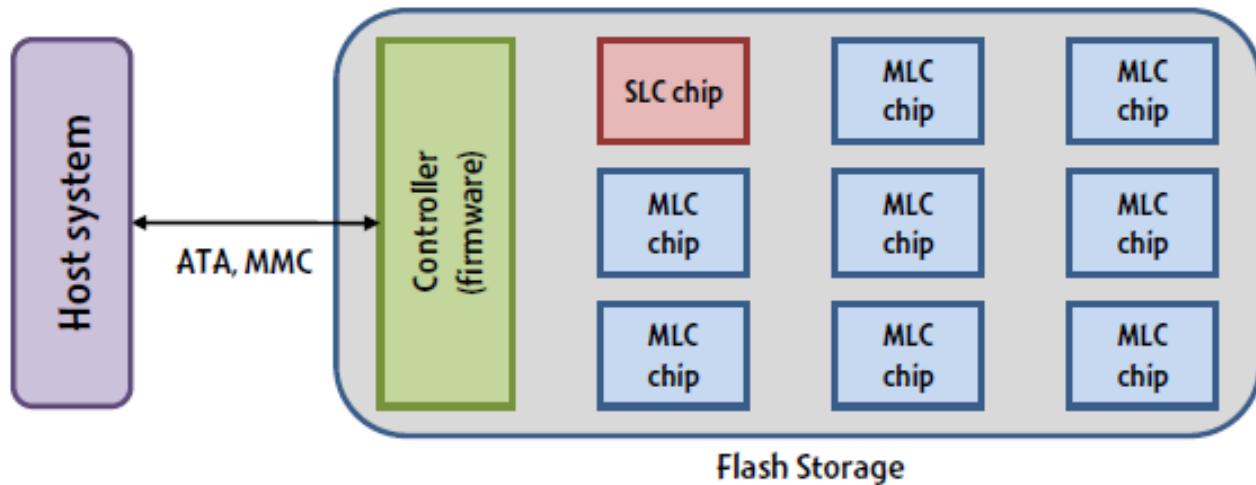
Conclusion

- Propose a new file system for MLC NAND flash memory
 - Exploits the flexible cell programming to achieve the SLC **performance** and MLC **capacity** while ensuring a reasonable **lifetime**



Other Solution

- SLC/MLC hybrid storage [Chang et al (2008), Park et al (2008), Im et al (2009)]
 - Composed of a single SLC chip and many MLC chips
 - Uses the SLC chip as a write buffer for MLC chips
 - Redirects frequently accessed small data into the SLC chip
 - Redirects bulk data into the MLC chips



Q: If using this design for an FS, which data should be put to SLC? Which to MLC?

LOG-BASED FILE SYSTEMS

Reevaluating Disk Performance

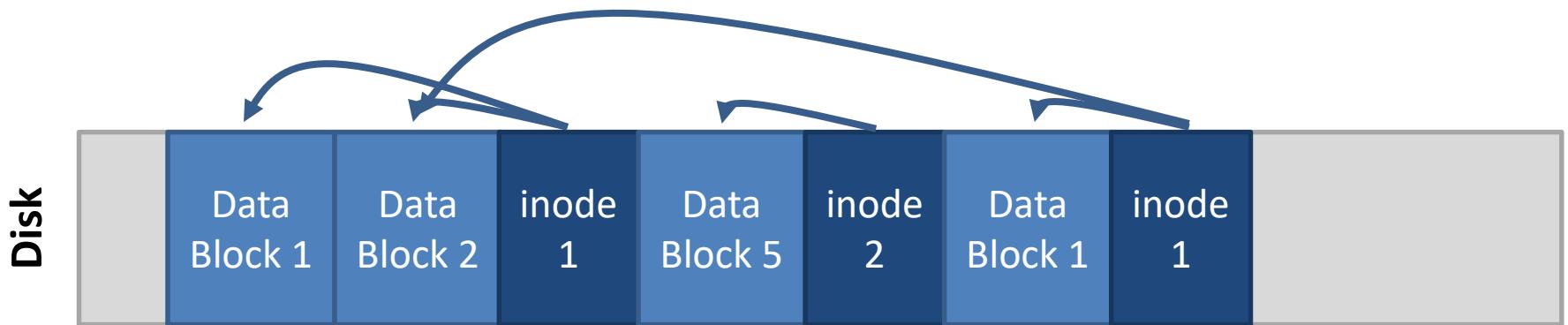
- How has computer hardware been evolving?
 - RAM has become cheaper and grown larger :)
 - Random access seek times have remained very slow :(
- This changing dynamic alters how disks are used
 - More data can be cached in RAM = less disk reads
 - Thus, writes will dominate disk I/O
- Can we create a file system that is optimized for sequential writes?

Log-structured File System

- Key idea: buffer all writes (including meta-data) in memory
 - Write these long segments to disk sequentially
 - Treat the disk as a circular buffer, i.e. don't overwrite
- Advantages:
 - All writes are large and sequential
- Big question:
 - How do you manage meta-data and maintain structure in this kind of design?

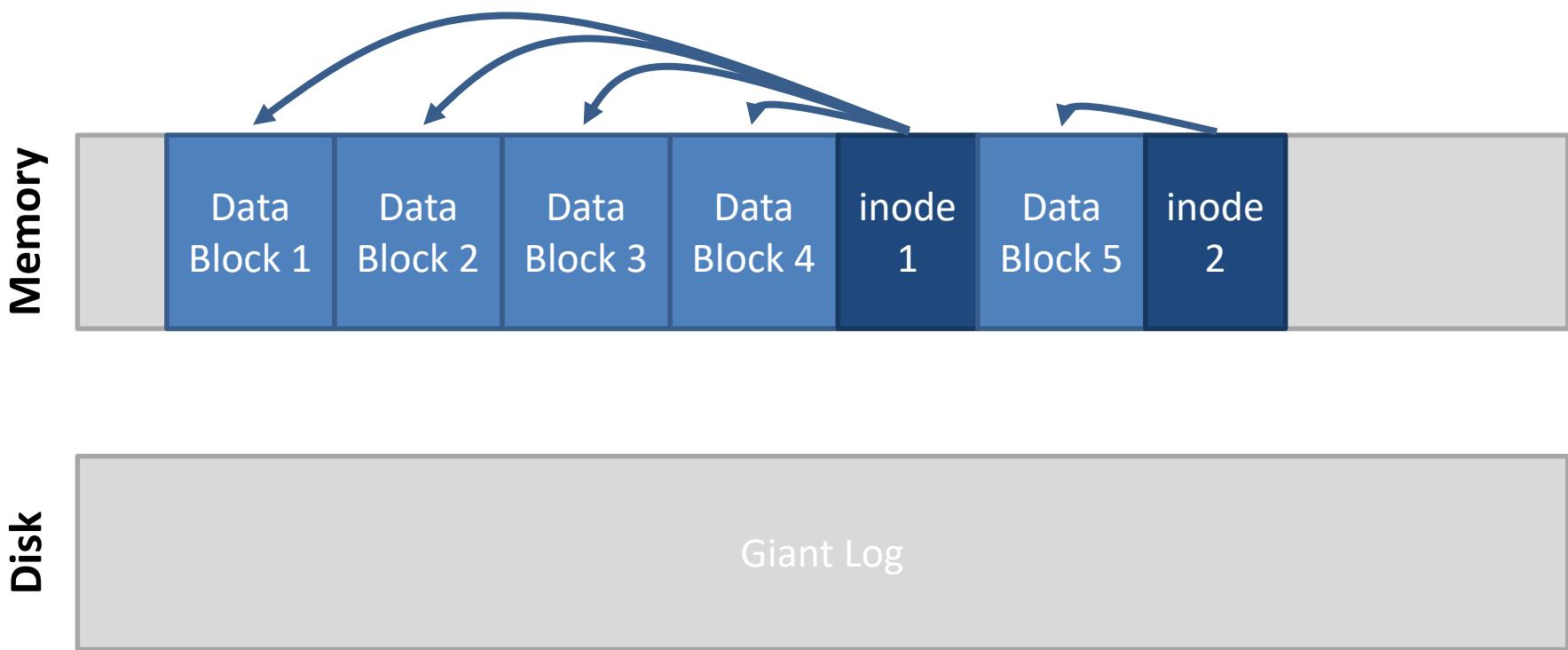
Treating the Disk as a Log

- Same concept as data journaling
 - Data and meta-data get appended to a log
 - Stale data isn't overwritten, its replaced



Buffering Writes

- LFS buffers writes in-memory into chunks

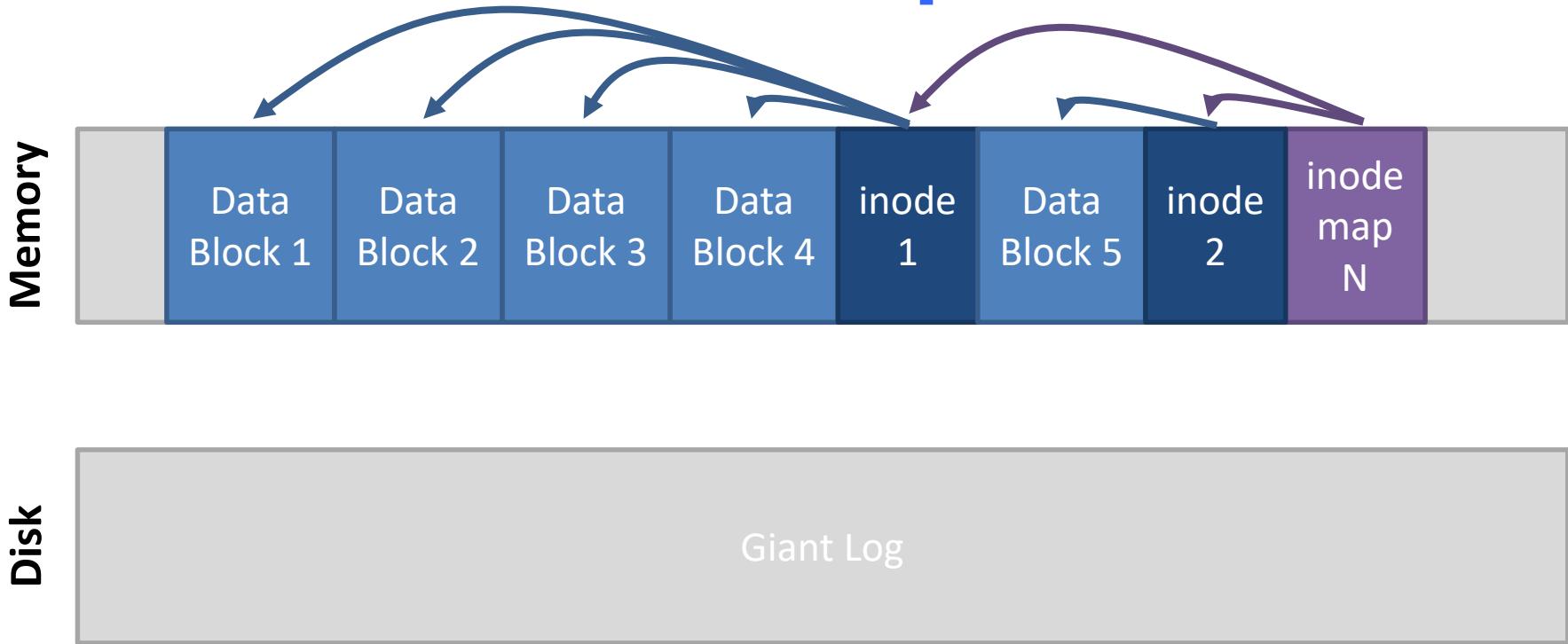


- Chunks get appended to the log once they are sufficiently large

How to Find inodes

- In a typical file system, the inodes are stored at fixed locations (relatively easy to find)
- How do you find inodes in the log?
 - Remember, there may be multiple copies of a given inode
- Solution: add a level of indirection
 - The traditional **inode map** can be broken into pieces
 - When a portion of the inode map is updated, write it to the log!

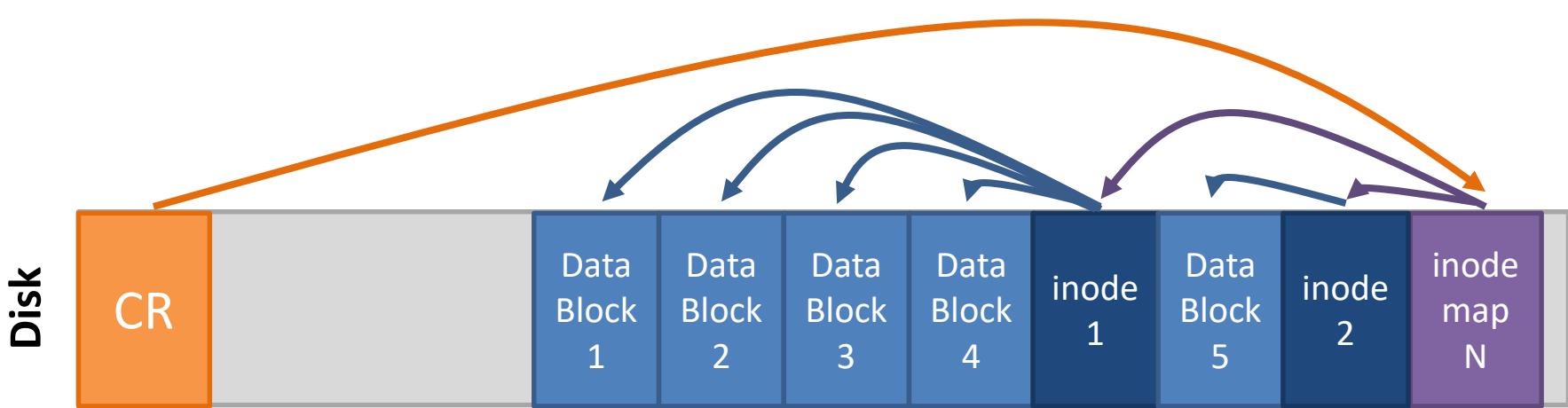
inode Maps



- New problem: the inode map is scattered throughout the log
 - How do we find the most up-to-date pieces?

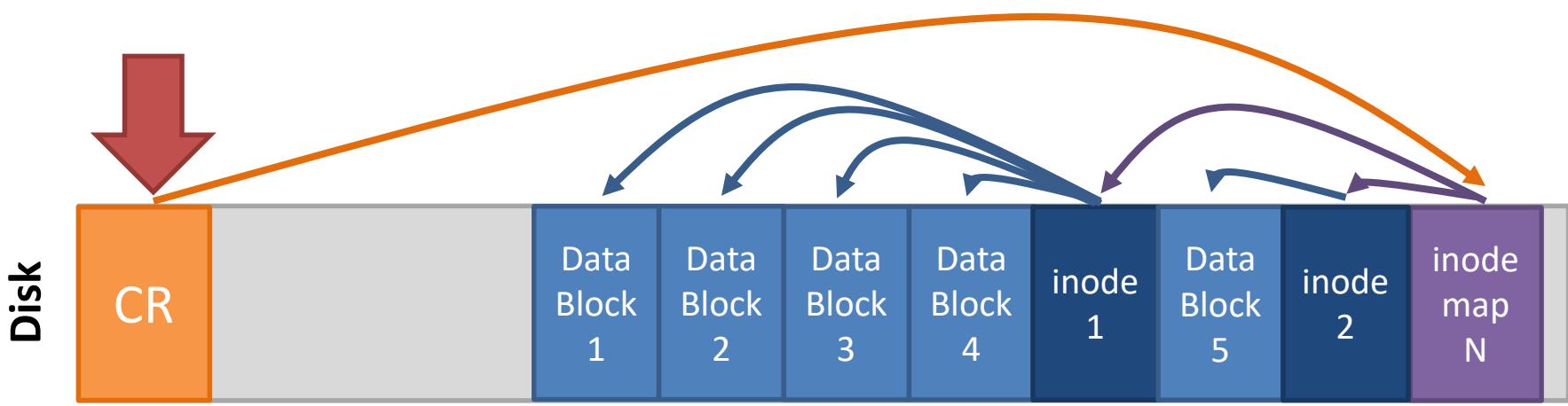
The Checkpoint Region

- The superblock in LFS contains pointers to all of the up-to-date inode maps
 - The **checkpoint region** is always cached in memory
 - Written periodically to disk, say ~30 seconds
 - Only part of LFS that isn't maintained in the log



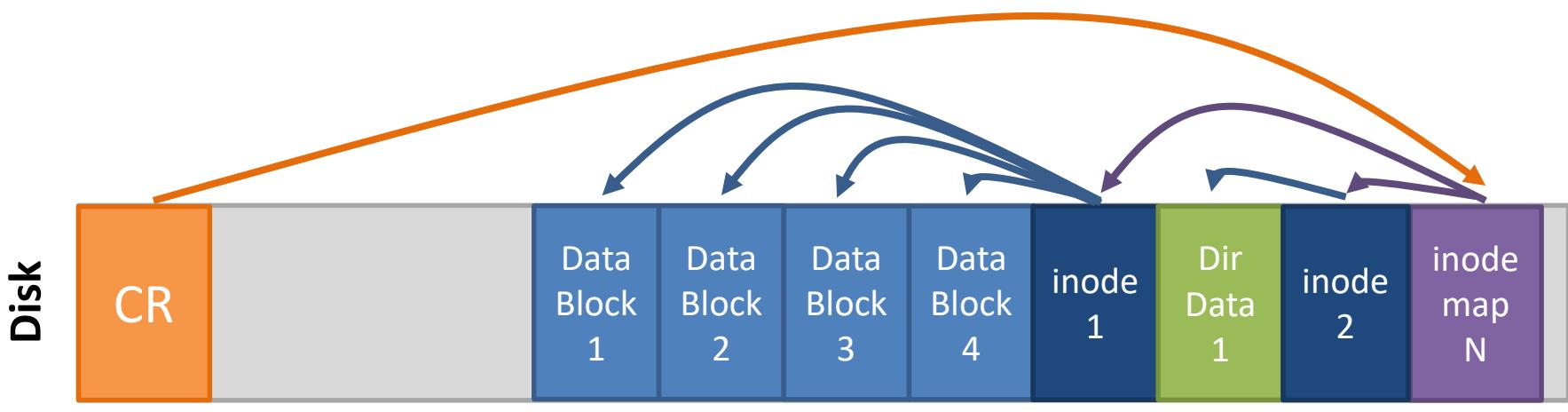
How to Read a File in LFS

- Suppose you want to read inode 1
 - 1. Look up inode 1 in the checkpoint region
 - inode map containing inode 1 is in sector X
 - 2. Read the inode map at sector X
 - inode 1 is in sector Y
 - 3. Read inode 1
 - File data is in sectors A, B, C, etc.



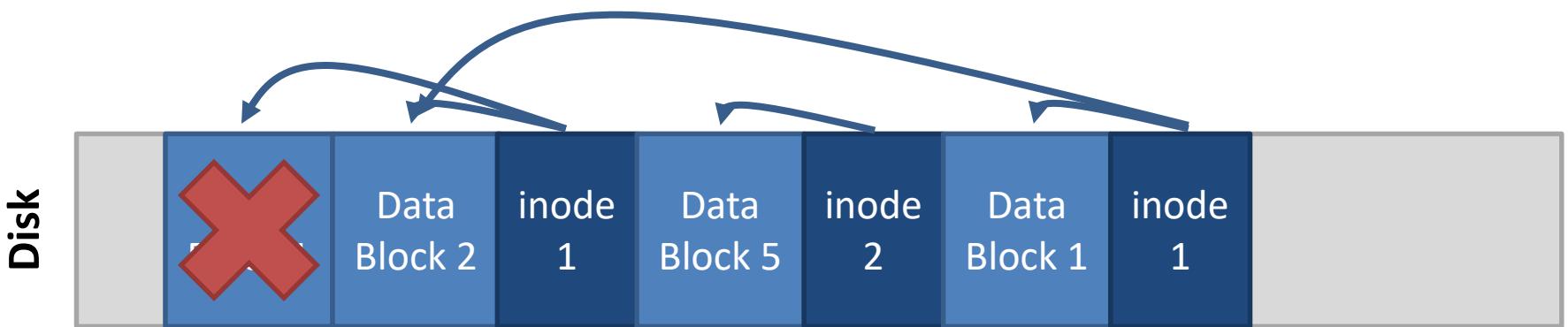
Directories in LFS

- Directories are stored just like in typical file systems
 - Directory data stored in a file
 - inode points to the directory file
 - Directory file contains name → inode mappings



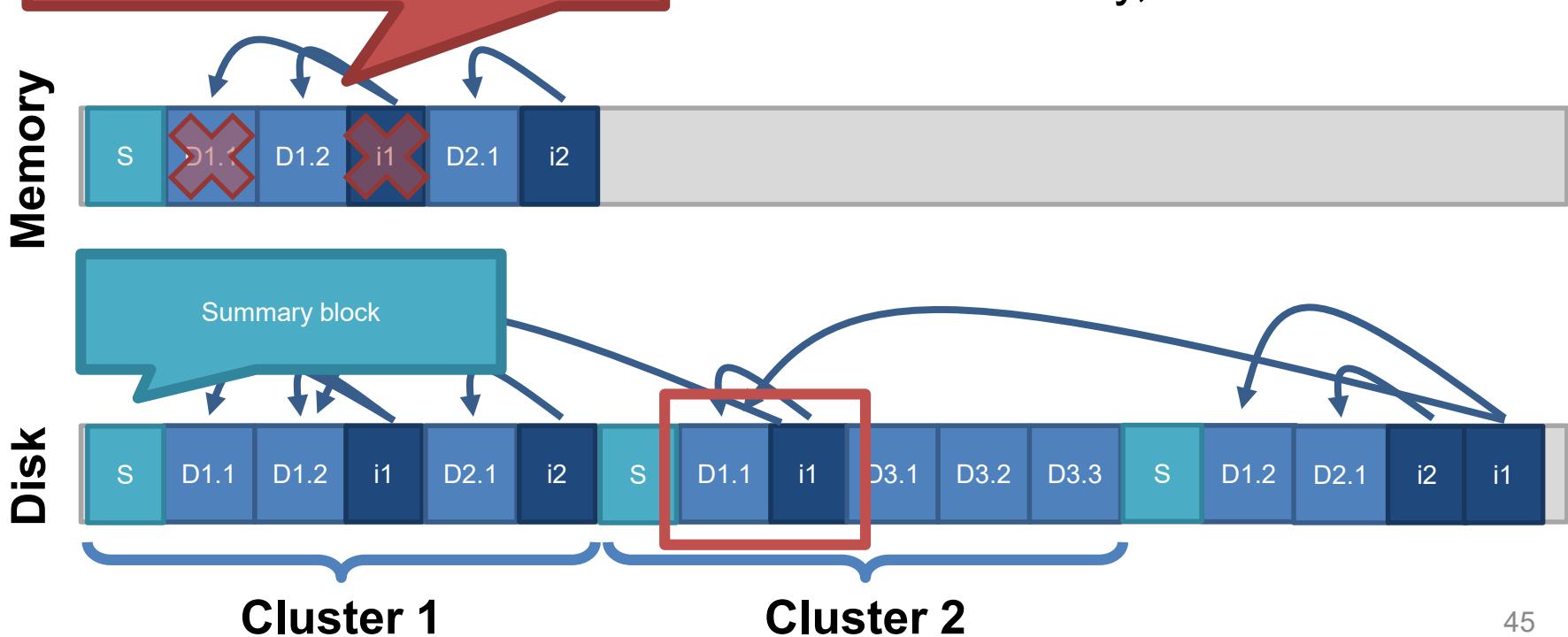
Garbage

- Over time, the log is going to fill up with stale data
 - Highly fragmented: live data mixed with stale data
- Periodically, the log must be garbage collected



Garbage Collection in LFS

- Each cluster has a summary block
 - Contains the ~~block \> inode~~ mapping for each block in the cluster
 - Which blocks are stale?
 - Pointers from other clusters are invisible



Segment Summary

- For each data block in the segment, the summary describes:
 - the **inode number N** of the inode that owns the block
 - the **offset F** of the block inside the inode's data region (e.g., "this block is at byte offset 4096")
- Using the summary, the cleaner determines whether a data block is live by:
 - reading inode N (either from the buffer cache or from disk via the inode map)
 - finding the appropriate data pointer for offset F, and seeing whether that pointer points to the data block in the segment being examined

Segment Summary

- When live data blocks are written to a new segment, the cleaner must also write new versions of the corresponding inodes and imap entries
- The inode map indicates which inodes are live, and each on-disk inode struct contains the inode number
 - So, when the cleaner finds inodes in an old segment, the cleaner can check the inode map to see whether the inodes should be written to a new segment

LFS Failure Recovery

- Checkpoint and roll-forward
- Recovery is very fast
 - No fsck, no need to check the entire disk
 - Recover the last checkpoint, and see how much data written after the checkpoint you can recover
 - Some data written after a checkpoint may be lost
 - Seconds versus hours

An Idea Whose Time Has Come

- LFS seems like a *very strange* design
 - Totally unlike traditional file system structures
 - Doesn't map well to our ideas about directory hierarchies
- Initially, people did not like LFS
- However, today its features are widely used

File Systems for SSDs

- Let's revisit SSD hardware constraints
 - To implement wear leveling, writes must be spread across the blocks of flash
 - Periodically, old blocks need to be garbage collected to prevent write-amplification
- Does this sound familiar?
- LFS is the ideal file system for SSDs!
- Internally, SSDs manage all files in a LFS
 - This is transparent to the OS and end-users
 - Ideal for wear-leveling and avoiding write-amplification

Copy-on-write

- Modern file systems incorporate ideas from LFS
- Copy-on-write semantics
 - Updated data is written to empty space on disk, rather than overwriting the original data
 - Helps prevent data corruption, improves sequential write performance
- Pioneered by LFS, now used in ZFS and btrfs
 - btrfs will probably be the next default file system in Linux

Versioning File Systems

- LFS keeps old copies of data by default
- Old versions of files may be useful!
 - Example: accidental file deletion
 - Example: accidentally doing `open(file, 'w')` on a file full of data
- Turn LFS flaw into a virtue
- Many modern file systems are **versioned**
 - Old copies of data are exposed to the user
 - The user may roll-back a file to recover old versions

[CCS'17] FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware

GFS and NFS

Yubin Xia

INTRO TO GFS

Background

- In 2003
 - 50+ GFS clusters
 - Each with thousands of storage nodes
 - Managing petabytes of data
 - GFS is under lots of applications and other systems in Google, such as bigtable, etc.
- Built on top of commodity machine and disks

Tailored for Google workloads

Introduction to GFS

- Shares the same goals as previous distributed file systems
 - Performance, Scalability, Reliability, Availability
- But with some differences:
 - Component **failures** are considered the norm rather than the exception
 - Files are **huge** by traditional standards.
 - Most files are mutated by **appending** new data rather than overwriting existing data
 - **Co-designing** the file system and applications increases flexibility in development

Design Assumptions

- System is built from **commodity hardware** which *fails* as the norm
 - The system must be able to detect and recover from such occurrences
- The system must be optimized to deal with **large** files (multi-GB)
- Two types of **reading**
 - Large streaming reads
 - Small random reads

Design Assumptions

- Large **sequential** writes
 - Appending data to files which are seldom altered again.
- Atomicity in **concurrent** writes from multiple clients
- High sustained **bandwidth** is more valuable than low **latency**

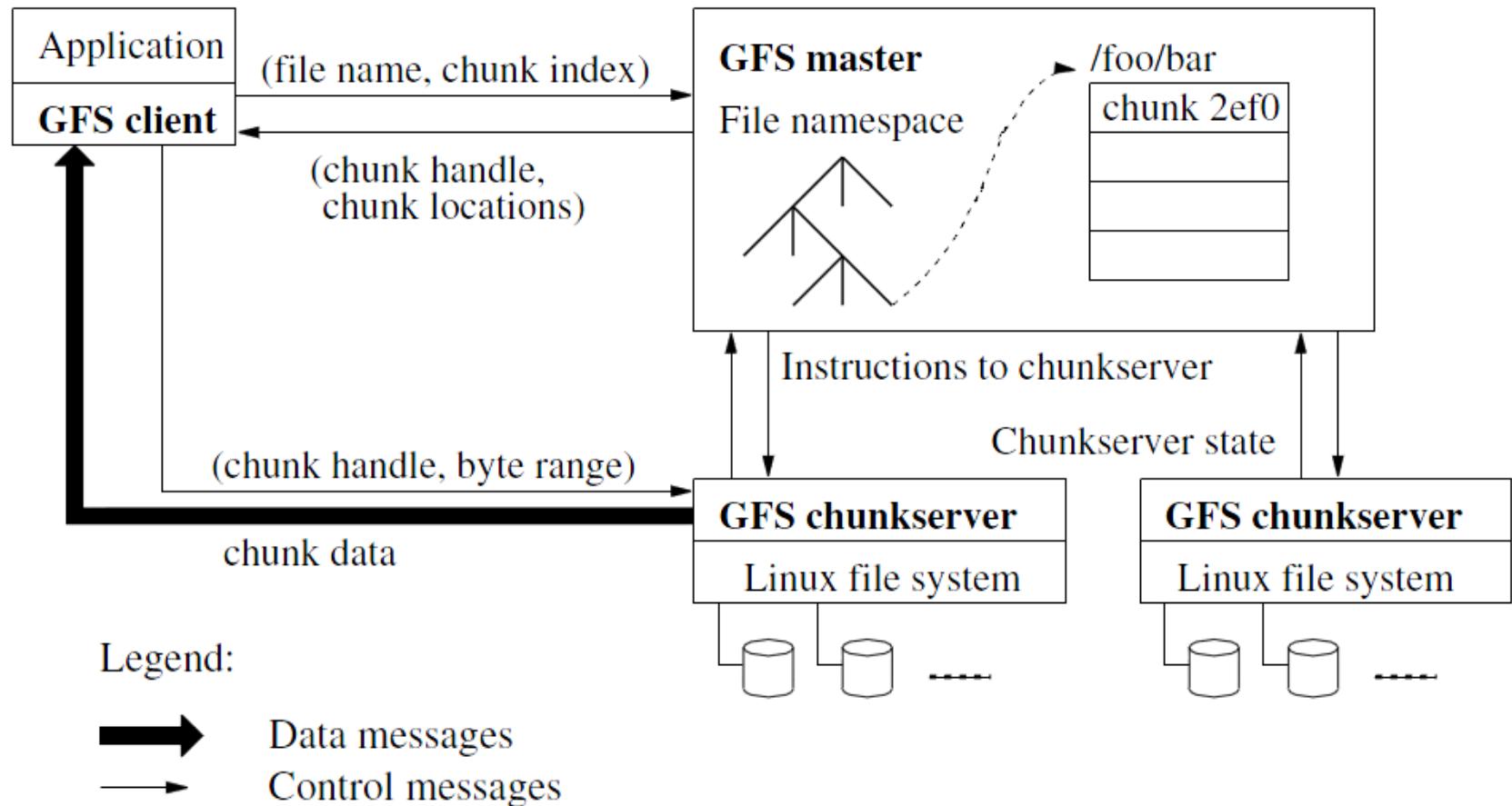
Typical workloads on GFS

- Two kinds of reads: **large streaming reads & small random reads**
 - Large streaming reads usually read 1MB or more
 - Oftentimes, applications read through contiguous regions in the file
 - Small random reads are usually only a few KBs at some arbitrary offset
- Also many large, **sequential writes** that append data to files
 - Similar operation sizes to reads
 - Once written, files are seldom modified again
 - Small writes at arbitrary offsets do not have to be efficient
- Multiple clients (e.g. ~100) **concurrently** appending to a single file
 - e.g. producer-consumer queues, many-way merging

GFS Interface

- Not POSIX-compliant, but supports typical file system operations: **create**, **delete**, **open**, **close**, **read**, and **write**
- **snapshot**: creates a copy of a file or a directory tree at low cost
- **record append**: allow multiple clients to append data to the same file concurrently
 - At least the very first append is guaranteed to be atomic

Architecture



Design Architecture

- **GFS Cluster** (accessed by clients)
 - Single Master + Multiple Chunkservers
- **Chunkserver**
 - Files of fixed sized chunks
 - Each chunk has a globally unique 64-bit chunk handle.
- **Master**
 - Maintains file system metadata
 - Namespace
 - Access Control Information
 - Mapping from files to chunks
 - Current locations of chunks

Design Architecture

- Principle: **data flow is decoupled from control flow**
 - Clients interact with the master for metadata operations
 - Clients interact directly with chunkservers for all files operations
 - This means performance can be improved by scheduling expensive data flow based on the network topology
- Neither the clients nor the chunkservers cache file data
 - Working sets are usually too large to be cached, chunkservers can use Linux's buffer cache

Single Master Node

- Clients do not read or write through the master
- The master relays relevant chunkserver location information to the client
- The client temporarily caches the chunkserver data and directly accesses the chunkserver
- Shadow Masters (fault tolerance)

Metadata on Master

- Three major types:
 - Chunk namespaces
 - Mapping from files to chunks
 - Location of chunk replicas
- Chunk location is not stored persistently and is instead read on each startup
- Metadata is stored in **memory**
 - Namespaces and file-to-chunk mappings are also stored persistently in operation log
- Master monitors chunk location through **heartbeat** messages with each chunk.

Metadata on Master

- For the namespace metadata
 - Master does not use any per-directory data structures
 - **No inodes!** No symlinks or hard links, either
 - Every file and directory is represented as a node in a lookup table, mapping pathnames to metadata
 - Stored efficiently using prefix compression
 - < 64 bytes per namespace entry
- Each node in the namespace tree has a corresponding read-write lock to manage concurrency
 - Because all metadata are stored in memory, the master can efficiently scan the entire state of the system periodically in the background
 - Master's memory capacity does not limit the size of the system

The Operation Log

- Only persistent record of metadata
- Also serves as a logical timeline that defines the serialized order of concurrent operations
- Master recovers its state by replaying the operation log
 - To minimize startup time, the master checkpoints the log periodically
 - The checkpoint is represented in a B-tree like form, can be directly mapped into memory, but stored on disk
 - Checkpoints are created without delaying incoming requests to master, can be created in ~1 minute for a cluster with a few million files

Chunkserver

- Contains chunks (blocks) of a fixed size
 - 64 MB
 - Fewer chunk location, fewer metadata
- Chunks are replicated regularly
 - Default: **3-way mirror** (across machines and racks)
- Talks with master through ***heartbeat*** messages
 - This let's master determines chunk locations and assesses state of the overall system
 - The chunkserver has the final word over what chunks it does or does not have on its own disks – not the master

Chunk Size

- 64 MB, a key design parameter (Much larger than most file systems)
- **Disadvantages:**
 - Wasted space due to internal fragmentation
 - Small files consist of a few chunks, which then get lots of traffic from concurrent clients
 - This can be mitigated by increasing the replication factor
- **Advantages:**
 - Reduces clients' need to interact with master (reads/writes on the same chunk only require one request)
 - Since client is likely to perform many operations on a given chunk, keeping a persistent TCP connection to the chunkserver reduces network overhead
 - Reduces the size of the metadata stored in master → metadata can be entirely kept in memory

Why a Single Master?

- The master now has global knowledge of the whole system, which drastically simplifies the design
- But the master is (hopefully) never the **bottleneck**
 - Clients never read and write file data through the master; client only requests from master which chunkservers to talk to
 - Master can also provide additional information about subsequent chunks to further reduce latency
 - Further reads of the same chunk don't involve the master, either

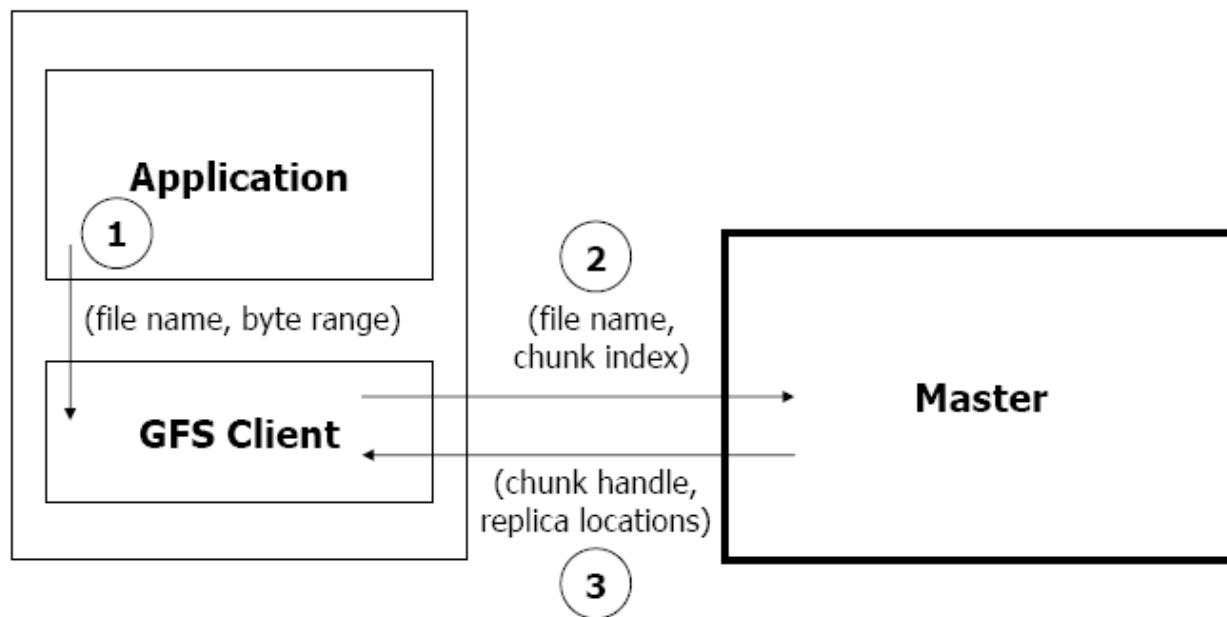
Why a Single Master?

- Master state is also replicated for reliability on multiple machines, using the operation log and checkpoints
 - If master fails, GFS can start a new master process at any of these replicas and modify DNS alias accordingly
- “Shadow” masters also provide read-only access to the file system, even when primary master is down
 - They read a replica of the operation log and apply the same sequence of changes
 - Not mirrors of master: they lag primary master by fractions of a second
 - This means we can still read up-to-date file contents while master is in recovery!

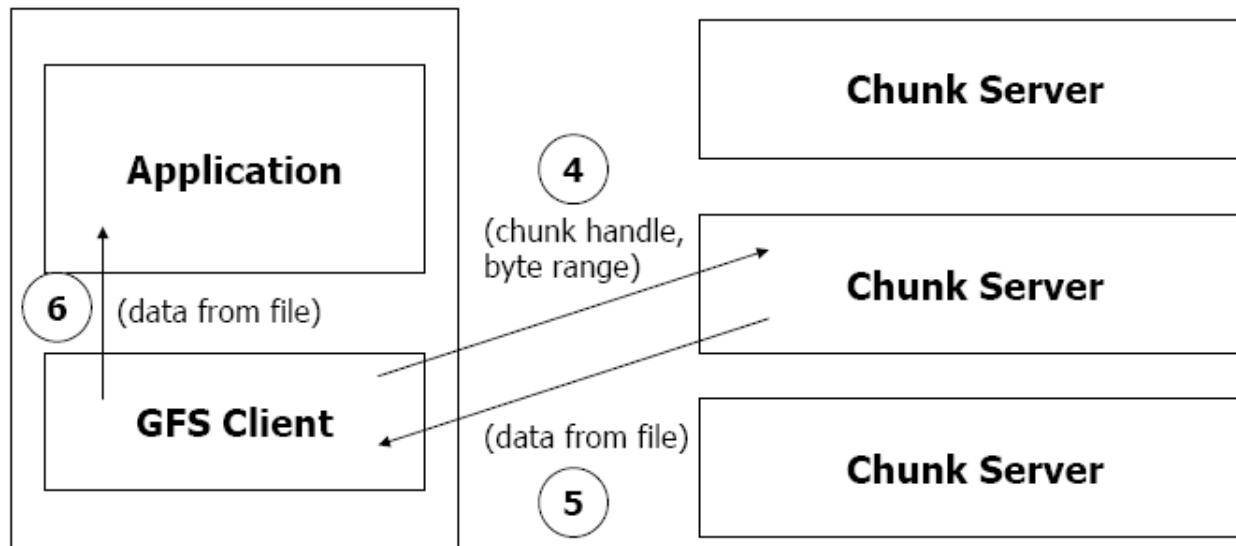
System Interactions

- Mutation
 - Write
 - Append
 - Acts on all of the chunk's replicas
- Lease
 - Initial timeout of 60 seconds, which can be renewed or revoked.
- Data Flow
 - Decoupled **control** flow from **data** flow
 - Data pushed linearly to avoid bottlenecks and high latency links

Read Algorithm



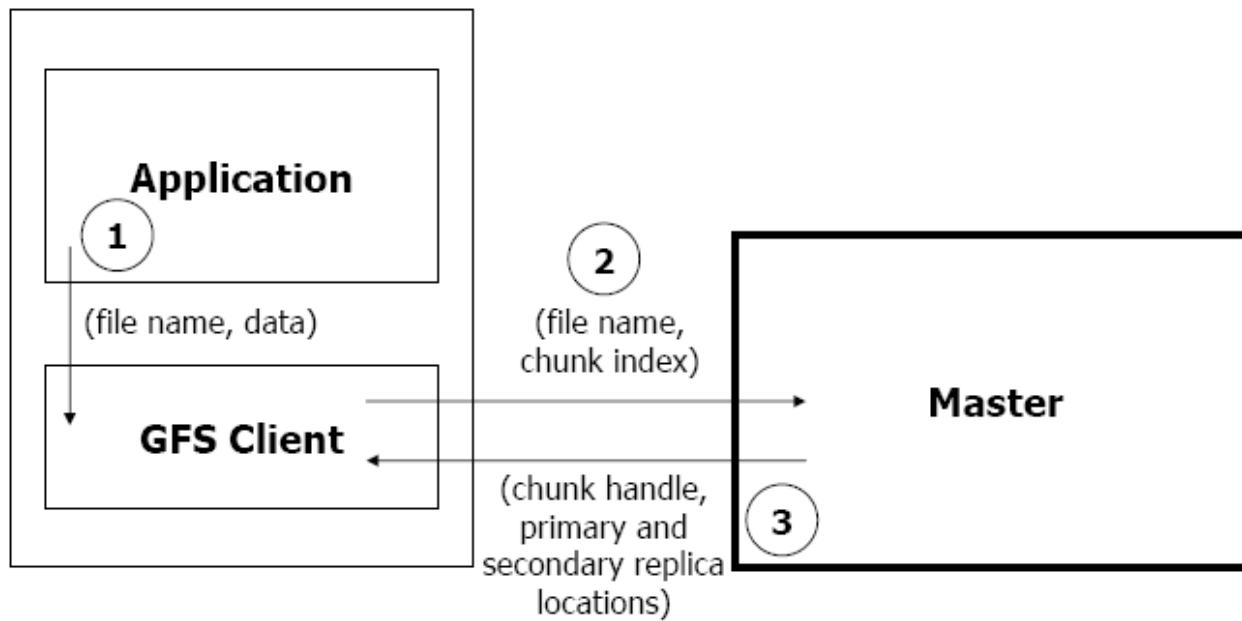
Read Algorithm



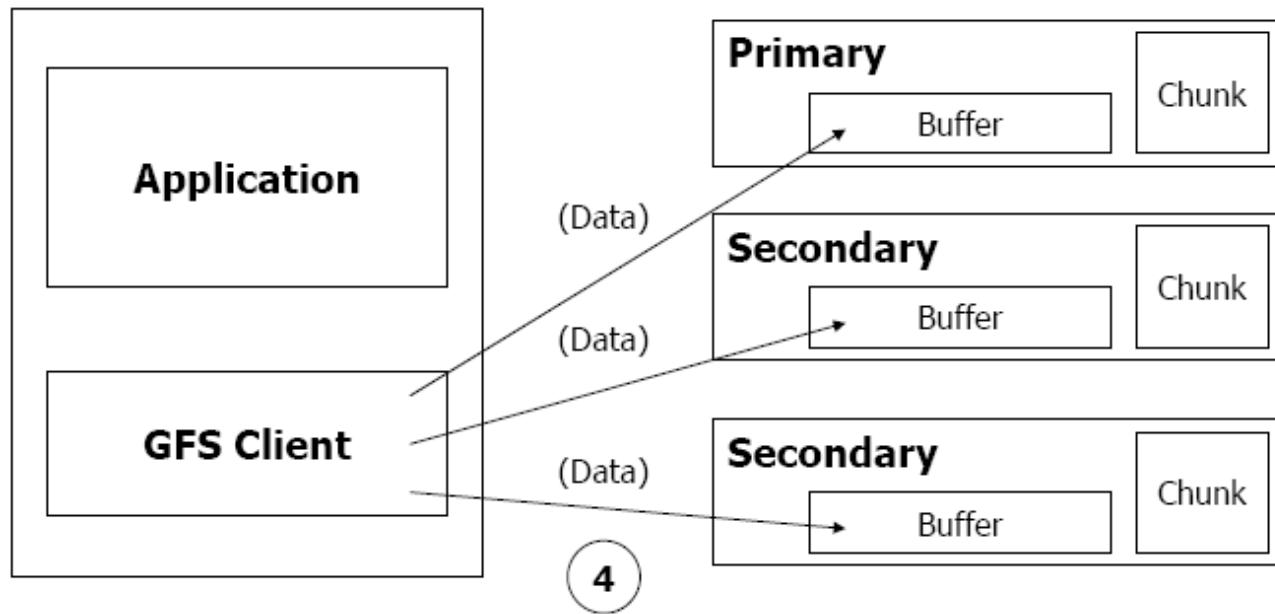
Read Algorithm

1. Application originates the read request.
2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and replica locations (i.e. chunk servers where the replicas are stored).
4. Client picks a location and sends the (chunk handle, byte range) request to that location.
5. Chunk server sends requested data to the client.
6. Client forwards the data to the application.

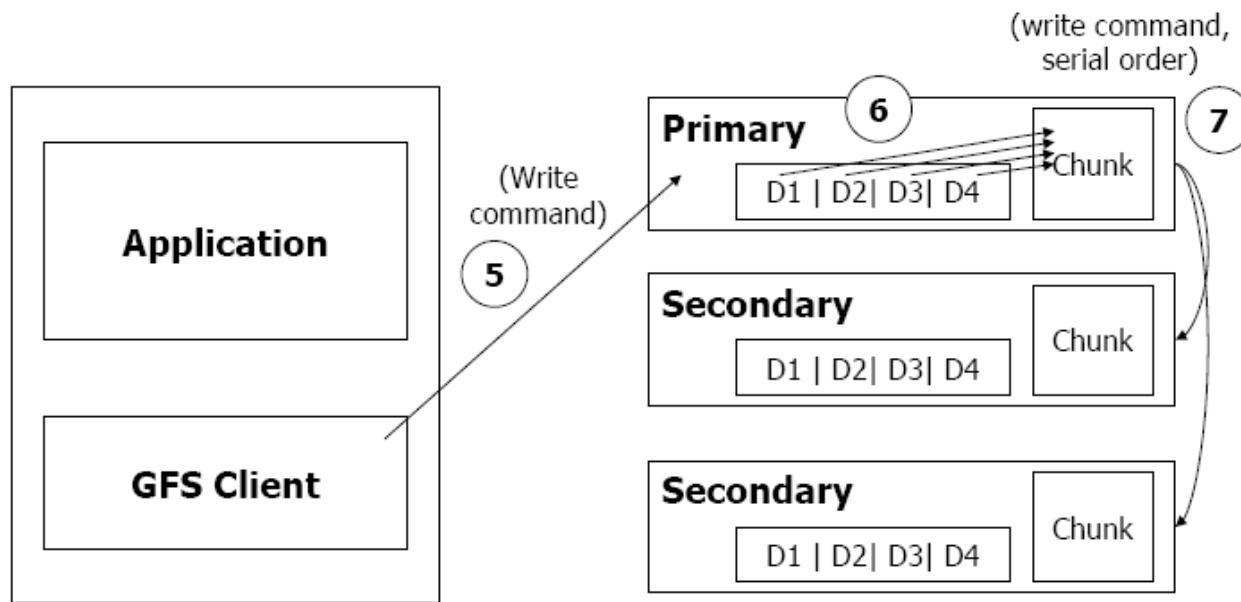
Write Algorithm



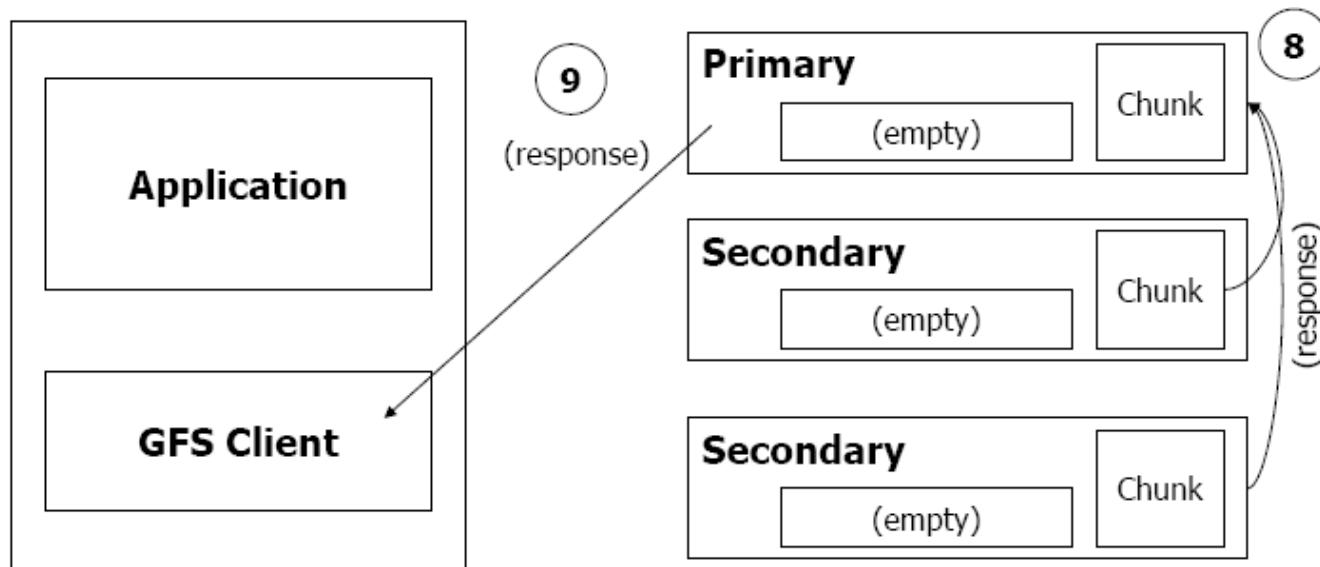
Write Algorithm



Write Algorithm



Write Algorithm



Write Algorithm

1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers' internal buffers.
5. Client sends write command to primary.

Write Algorithm

6. Primary determines **serial order** for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

Note: If write fails at one of chunk servers, client is informed and retries the write.

Master Operation

1. Replica Placement
2. Creation, Re-replication, Rebalancing
3. Garbage Collection
4. Stale Replica Detection

Fault Tolerance and Diagnosis

- Fast recovery
 - Master and chunkserver are designed to restore their states and start in seconds
- Chunk replication : 3-way mirrors
 - Across multiple machines, across multiple racks

Fault Tolerance and Diagnosis

- Master Mechanisms:
 - Log of all changes made to metadata
 - Periodic checkpoints of the log
 - Log and checkpoints replicated on multiple machines
 - Master state is replicated on multiple machines
 - “Shadow” masters for reading data if “real” master is down

Fault Tolerance and Diagnosis

- Data integrity
 - A chunk is divided into 64-KB blocks
 - Each with its 32-bit checksum
 - Verified at read and write times
 - Also background scans for rarely used data

Summary of GFS

- Runs on commodity hardware and is scalable
- Performs well for the specified tasks and assumptions previously mentioned
- Innovation
 - File system API tailored to stylized workload
 - Single-master design to simplify coordination
 - Metadata fit in memory

Summary of GFS

- Flat namespace
- Dedicated Care for Component Failure
 - hard disk failure, data corruption, network disconnection, etc.
- High-throughput
 - Minimized the master involvement
 - Chunk servers themselves send and receive the client data
 - The master leases authority to mutate chunks

INTRO TO NFS

Accessing Remote Files

FTP, telnet, ...

- Explicit access
- User-directed connection to access remote resource

We want more transparency

- Allow user to access remote resource just as local ones

NAS: Network Attached Storage

File Service Types

Upload/Download model

- Read file: copy file from server to client
- Write file: copy file from client to server

Advantage

- Simple

Problem

- Wasteful “what if client needs small piece?”
- Problematic “what if client doesn’t have enough space?”
- Consistency “what if others modify the same file?”

File Service Types

Remote access model

- File service provides functional interface
(create, delete, read, write, etc ...)

Advantage

- Client gets only what's needed
- Server can manage coherent view of file system

Problem

- Possible server and network congestion
 - Servers are accessed for duration of file access
 - Same data may be requested repeatedly

Remote File Service

File Service

- ❑ Provides file access interface to clients

Directory Service

- ❑ Maps textual names for file to internal locations that can be used by file service

Client module (driver)

- ❑ Client side interface for file & directory service
- ❑ If done right, helps provide access transparency
 - e.g. implement the FS under the VFS layer

Semantics of File Sharing

Sequential semantics

- Read returns result of **last write**

Easily achieved if

- Only one server
- Clients do not cache data

BUT

- Performance problem if no cache
- We can **write-through**
 - Must notify clients holding copies
 - Requires extra state, generates extra traffic

Semantics of File Sharing

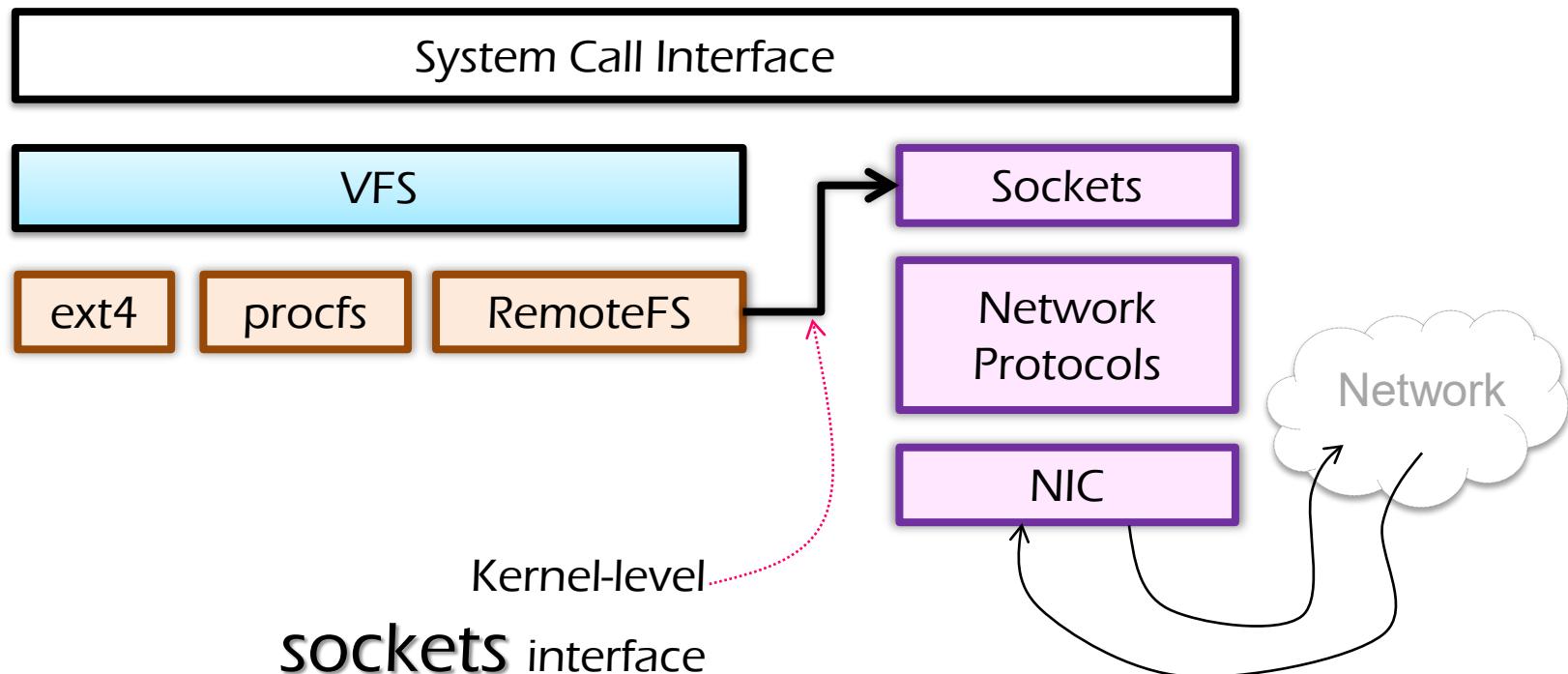
Session semantics

Relax the rules

- Changes to an open file are initially **visible** only to the process that modified it
- Last process to modify the file **wins**

Accessing Remote Files

Implement the client module as an FS under VFS



Server: Stateful or Stateless

Stateful →

Server maintains client-specific state

Shorter requests

Better performance in processing requests

Cache coherence is possible

- Server can know who's accessing what

File locking is possible

Server: Stateful or Stateless

Stateless → Server maintains no info for client

Each request must identify file and offsets

Server/Client can crash and recovery

- No state to lose

No open/close needed

No server space used for state (scalable)

Problems if file is deleted on server

File locking not possible

Caching

Hide latency to improve performance for repeated accesses by bringing the data closer to where it's needed

Three places to replicate data

- Server's buffer cache
- Client's buffer cache
- Client's disk



WARNING:
Potential cache
consistency problem

Approaches to Caching

Write-through

- Keep data **cached** at the client but **send** modifications to the server
- What if **another** client reads its own (out-of-data) cache copy?
- All accesses will require **checking** with server
- Or ... server maintains state of clients and sends **invalidations**

Approaches to Caching

Delayed writes (write-behind)

- Data is cached **locally** (watch out for consistency – others won't see updates)
- Remote files updated (~~immediately~~) periodically
- One bulk **write** is more efficient than lots of **little writes**

- Problem: semantics become ambiguous

Approaches to Caching

Read-ahead (prefetch)

- Request chunks of data **before** it is needed
- Minimize **wait** when it actually is needed

Write on close

- Admit that we have **session semantics**

Centralized control

- Keep track of who has what **open** and **cached** on each node
- Stateful file system with signaling traffic

NFS: Network File System

Design Goals

- Any machine can be a client or server
- Must support diskless workstations
- **Heterogeneous** system must be supported
 - Different HW, OS, underlying file system
- Access transparency
- Recovery from failure
 - Stateless, UDP, client retries
- High performance
 - Use caching and read-ahead

NFS Protocols: Mount

Mounting Protocol:

Request access to exported directory tree

- Client sends pathname to server
 - Requests permission to access contents

Client: parses **pathname**
contacts server for file **handle**

- Server returns file handle

Client: create in-memory VFS **inode** at mount point
internally points to **rnode** for remote files
(client keeps state, not the server)

NFS Protocols: Mount

Static mounting

- mount request contacts server

Server: add list of shared directories to `/etc/exports`

Client: `mount r900:/users/paul /home/paul`

NFS Protocols: Access

Directory and File Access Protocol:

Access files and directories (read, mkdir, ...)

- First, perform a **lookup** RPC
 - Returns file handle and attributes
- “**lookup**” is not like “**open**”
 - Establish state on the client **only**
(no information on server)
 - Call the NFS **lookup** function
- The **handle** passed as a parameter for other file access function
 - e.g., **read(handle, offset, count)**

NFS Performance

Usually **slower** than local

Improve by **caching** at client

- Goal: reduce number of remote ops
- Caching: **read, readlink, getattr, lookup, readdir**
 1. Cache file data at client (buffer cache)
 2. Cache file attribute information at client
 3. Cache pathname bindings for faster lookup

Server side

- Caching is “automatic” via buffer cache
- All NFS writes are **write-through** to disk

Validation

Inconsistencies may arise in NFS

Try to resolve inconsistency by validation

- Save timestamp of file
- When file opened or server contacted for new
 - 1. Compare last modification time
 - 2. If remote is more recent, invalidate cached data

Always invalidate data after some time

- open files (3 sec), directories (30 sec)

If data block is modified, it is:

- Marked dirty, then flushed on file close

Improving Read Performance

Transfer data in large chunks

- 8KB default

Read-ahead

- Optimize for sequential file access
- Send requests to read disk blocks before they are requested by the applications

Problem with NFS

File consistency

Assumes clocks are synchronized

Open with append can't be guaranteed to work

Locking cannot work

- Separate lock manager added (stateful)

No reference counting of open files

- You can delete a file opened by yourself/others

Global UID space assumed

...

Scalable Locking

Yubin Xia

Some slides adjusted from. Elsa L Gunter (UIUC), Jonathan Walpole (PSU)
Paul McKenney (IBM) Tom Hart (University of Toronto), Frans Kaashoek (MIT)

Outline

Scalability Tutorial

Non-scalable locks

Scalable locks

SCALABILITY TUTORIALS

What is scalability?

Application does N times as much work on N cores
as it could on 1 core

Scalability may be limited by Amdahl's Law:

Locks, shared data structures, ... Shared hardware
(DRAM, NIC, ...)

$$S_{\text{latency}}(s) = \frac{1}{1 - p + \frac{p}{s}}$$

Locking

Why do kernels normally use locks?

Locks support a concurrent programming style based on **mutual exclusion**

- Acquire lock on entry to critical sections

- Release lock on exit

- Block or spin if lock is held

- Only one thread at a time executes the critical section

Locks prevent concurrent access and enable *sequential reasoning* about critical section code

Sample: SEND & RECEIVE

Bounded Buffer

- Shared between modules
- Producers and consumers

Race Condition

- If multiple writers
- Using lock to ensure single writer principle

Locking

- ACQUIRE & RELEASE
- Need atomicity
- RSM: read-set-memory

Amdahl's Law

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

Two independent parts **A** **B**



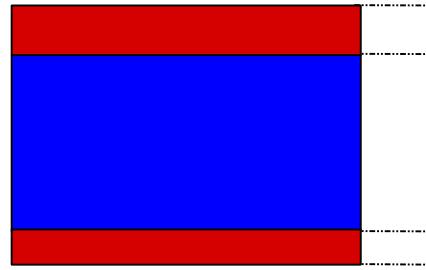
Make **B** 5x faster

The same horizontal bar is shown again, but the blue segment (B) is now much longer, indicating it has been sped up 5 times while the red segment (A) remains the same length.

Make **A** 2x faster

The same horizontal bar is shown again, but the red segment (A) is now much shorter, indicating it has been sped up 2 times while the blue segment (B) remains the same length.

Critical-section efficiency



$$\text{CriticalSection efficiency} = \frac{T_c}{T_c + T_a + T_r}$$

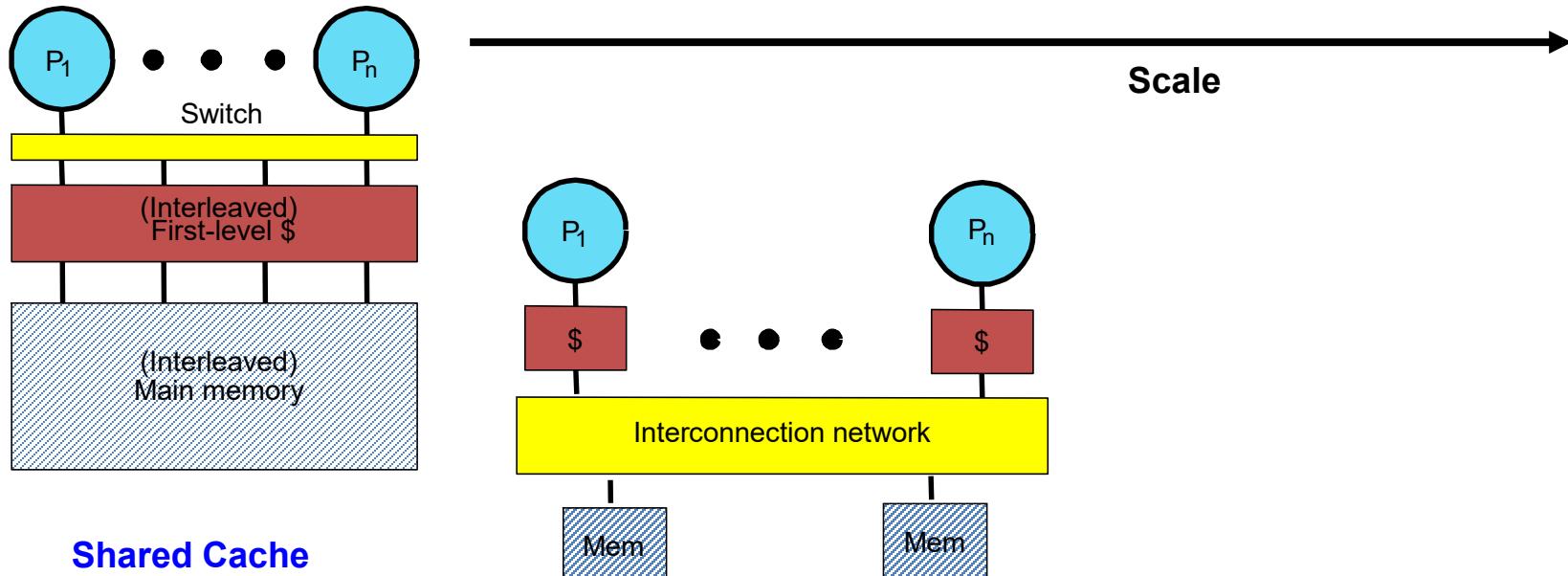
Ignoring lock contention and cache conflicts in the critical section

Some numbers you may want to know

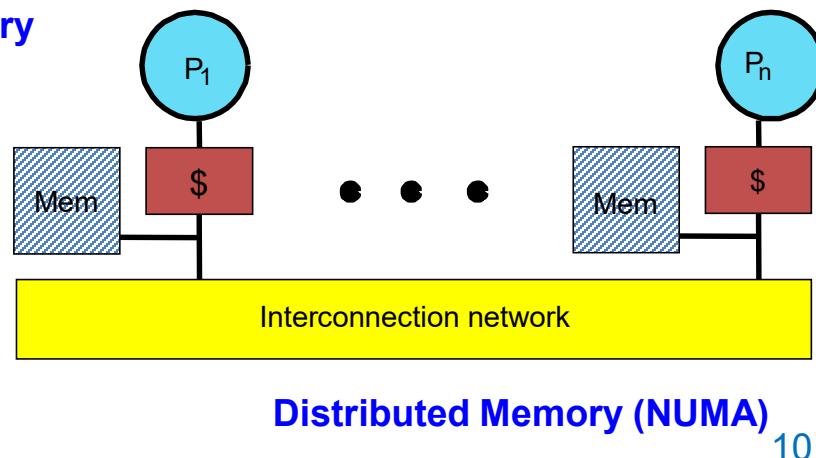
	Opteron	Xeon	Niagara	Tilera
L1	3	5	3	2
L2	15	11		11
LLC	40	44	24	45
RAM	136	355	176	118

Source: Everything you always wanted to know about synchronization, but were afraid to ask. [SOSP'13]

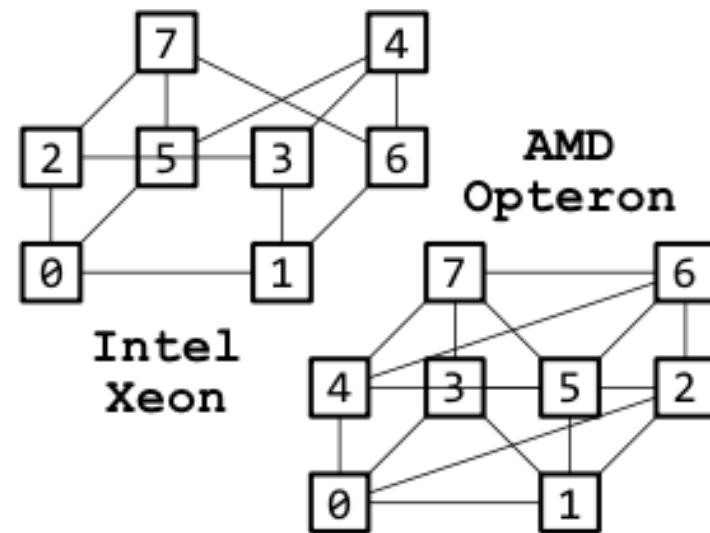
Shared Memory Multiprocessors



**Centralized Memory
Dance Hall, UMA**



More numbers you may want to know



(a) System topology

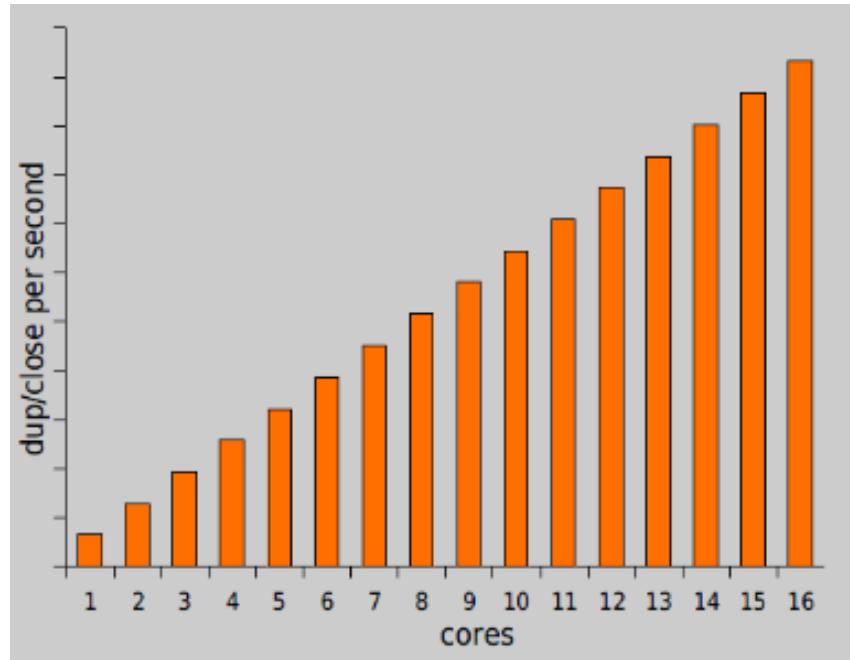
Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

(b) Latencies (cycles) on the distance

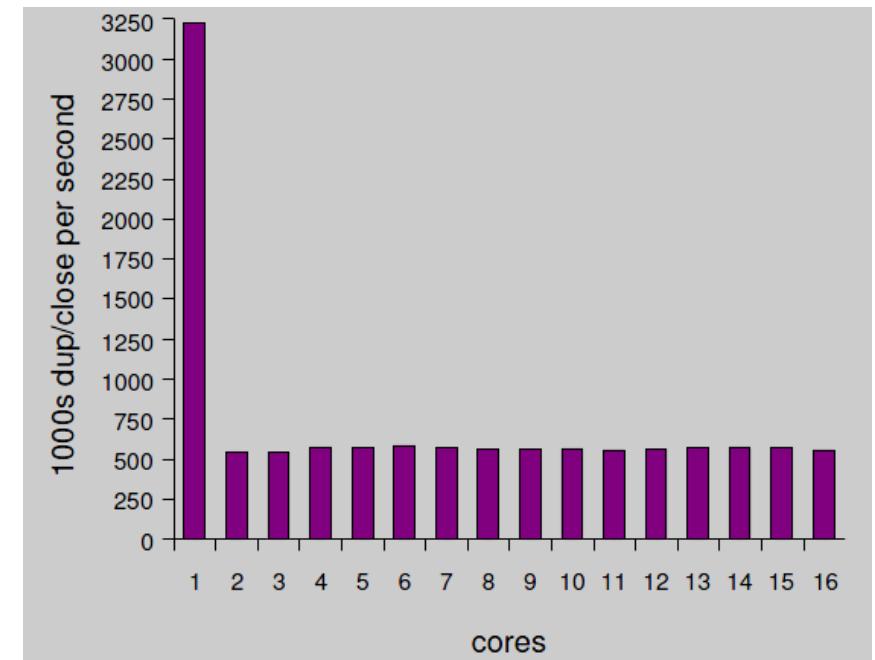
```
fd_alloc(void) {  
    lock(fd_table);  
    fd = get_free_fd();  
    set_fd_used(fd);  
    fix_smallest_fd()  
    unlock(fd_table);  
}
```

Motivating example: file descriptors

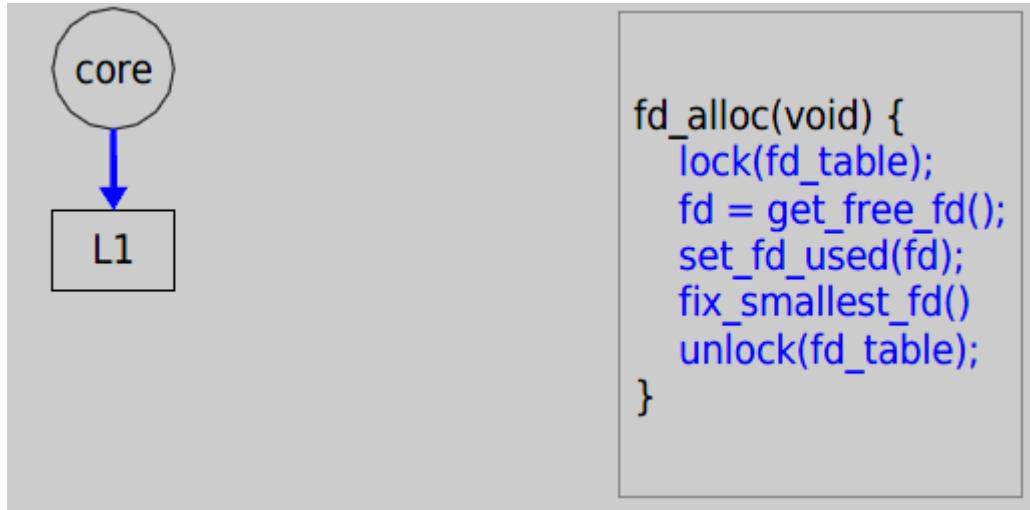
Ideal FD performance graph



Actual FD performance

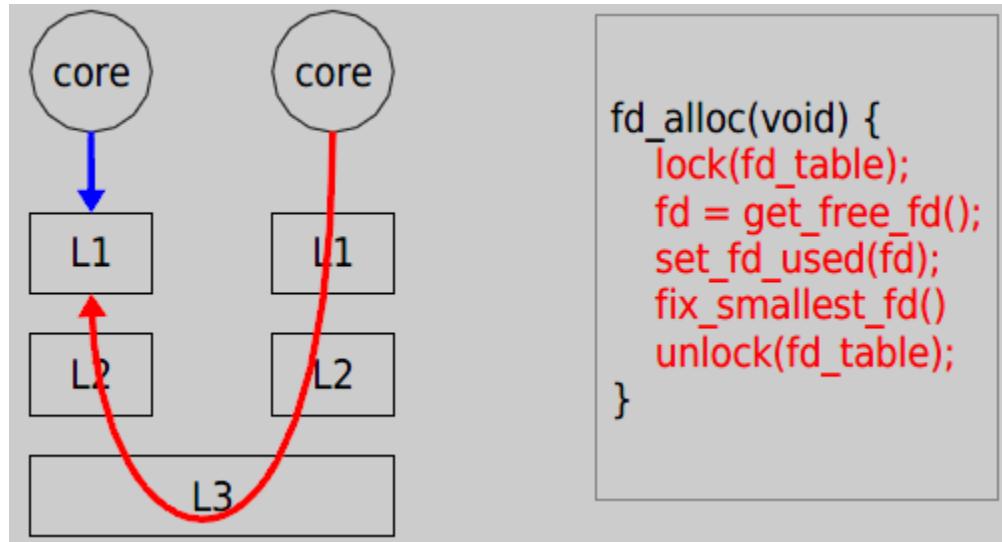


Why throughput drops?



Load fd_table data from L1 in 3 cycles.

Why throughput drops?



Now it takes **121** cycles!

**NON-SCALABLE LOCKING
ARE DANGEROUS**

Cause: Non-scalable locks

Non-scalable locks

- Such as spin locks

- Poor performance when highly contended

Many systems are using non-scalable locks

But they are dangerous

Why dangerous

Lead to **performance collapse** when adding a few more cores

Even tiny critical section will also lead to this performance collapse

Xv6 locking

```
// Acquire the lock.  
// Loops (spins) until the lock is acquired.  
// Holding a lock for a long time may cause  
// other CPUs to waste time spinning to acquire it.  
void  
acquire(struct spinlock *lk)  
{  
    pushcli(); // disable interrupts to avoid deadlock.  
    if(holding(lk))  
        panic("acquire");  
  
    // The xchg is atomic.  
    // It also serializes, so that reads after acquire are not  
    // reordered before it.  
    while(xchg(&lk->locked, 1) != 0)  
        ;  
  
    // Record info about lock acquisition for debugging.  
    lk->cpu = cpu;  
    getcallerpcs(&lk, lk->pcs);  
}  
  
void  
release(struct spinlock *lk)  
{  
    if(!holding(lk))  
        panic("release");  
  
    lk->pcs[0] = 0;  
    lk->cpu = 0;  
  
    // The xchg serializes, so that reads before release are  
    // not reordered after it. The 1996 PentiumPro manual (Volume 3,  
    // 7.2) says reads can be carried out speculatively and in  
    // any order, which implies we need to serialize here.  
    // But the 2007 Intel 64 Architecture Memory Ordering White  
    // Paper says that Intel 64 and IA-32 will not move a load  
    // after a store. So lock->locked = 0 would work here.  
    // The xchg being asm volatile ensures gcc emits it after  
    // the above assignments (and after the critical section).  
    xchg(&lk->locked, 0);  
  
    popcli();  
}
```

Case study: ticket spinlock

Normal spinlock has extremely noticeable unfairness

Even on a 8-core CPU

Ticket spinlock guarantees lock are granted to acquirers in order

Used in Linux kernel

But it's non-scalable lock

Pseudo code for ticket lock

```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

void spin_lock(spinlock_t *l) {
    int t = atomic_xadd(&l->next_ticket);
    while (t != l->current_ticket)
        ; // spin
}

void spin_unlock(spinlock_t *l) {
    l->current_ticket++;
}
```

Pseudo code for ticket lock

```
struct spinlock_t {  
    int current_ticket; ← Currently serving which one  
    int next_ticket; ← Ticket for the next comer  
}  
  
void spin_lock(spinlock_t *l) {  
    int t = atomic_xadd(&l->next_ticket); ← Add and get  
    while (t != l->current_ticket)  
        ; // spin  
}  
  
void spin_unlock(spinlock_t *l) {  
    l->current_ticket++;  
}
```

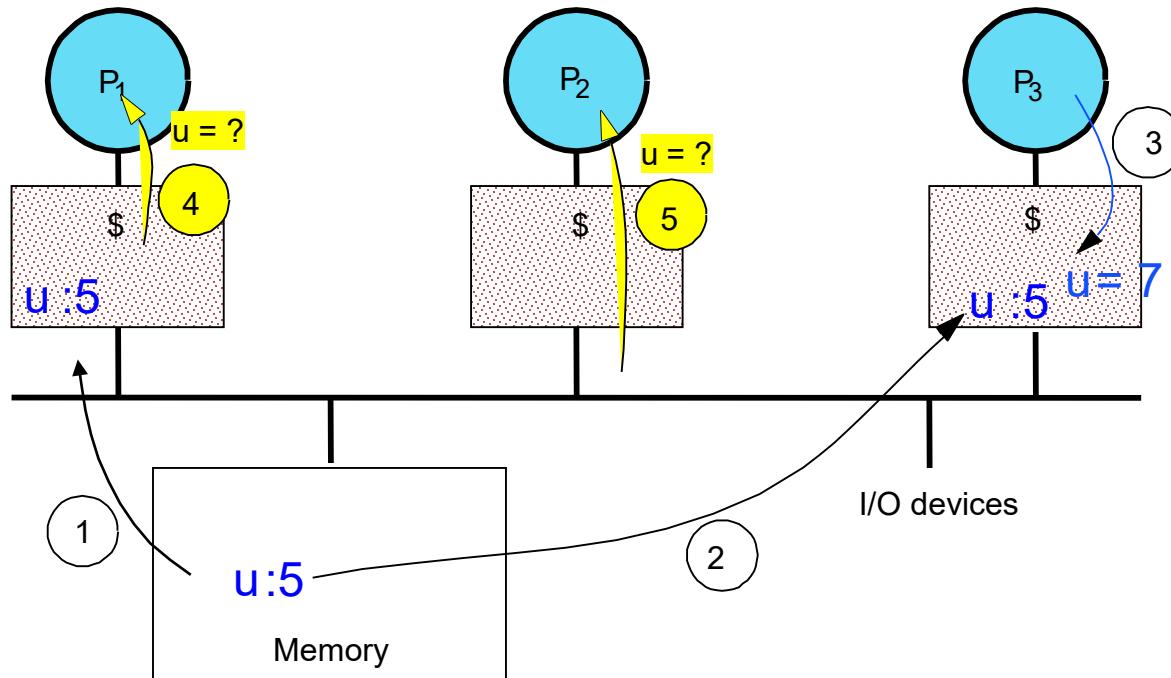
Background on cache coherence

On multi-core processors, each core has its own private cache

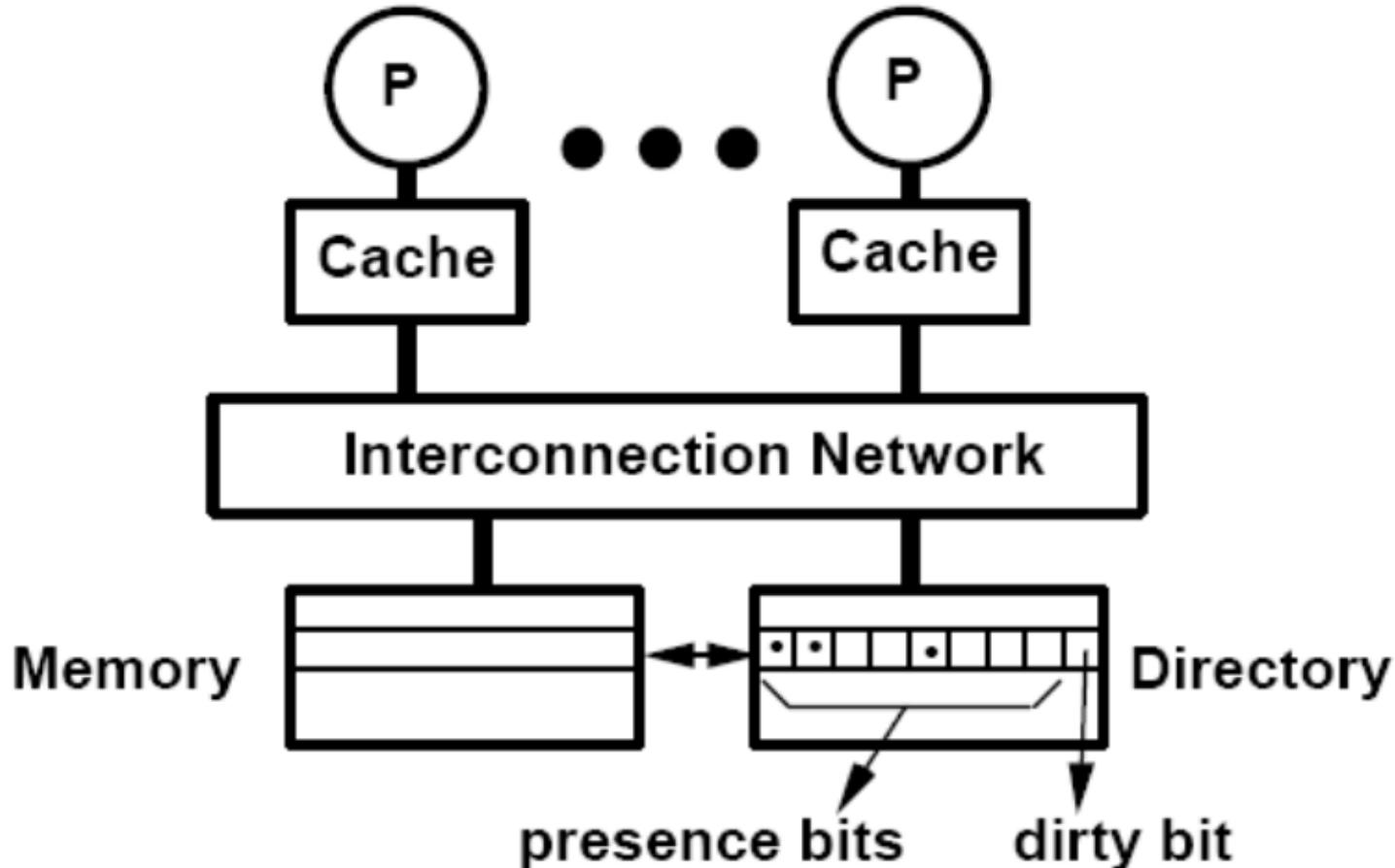
Need to keep the cache content in sync when some CPU modifies some memory

Accomplished by cache coherence protocol

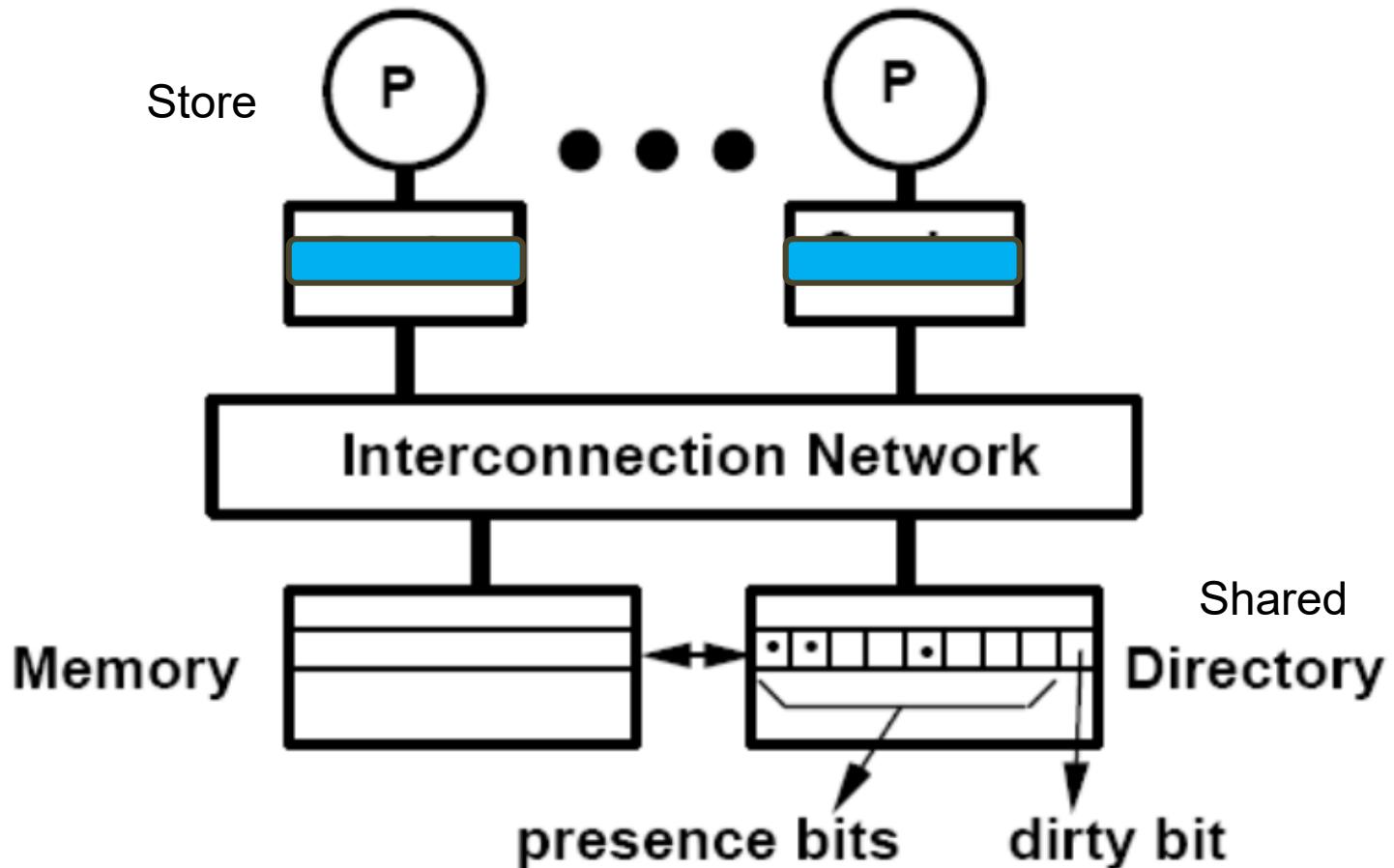
Example Cache Coherence Problem



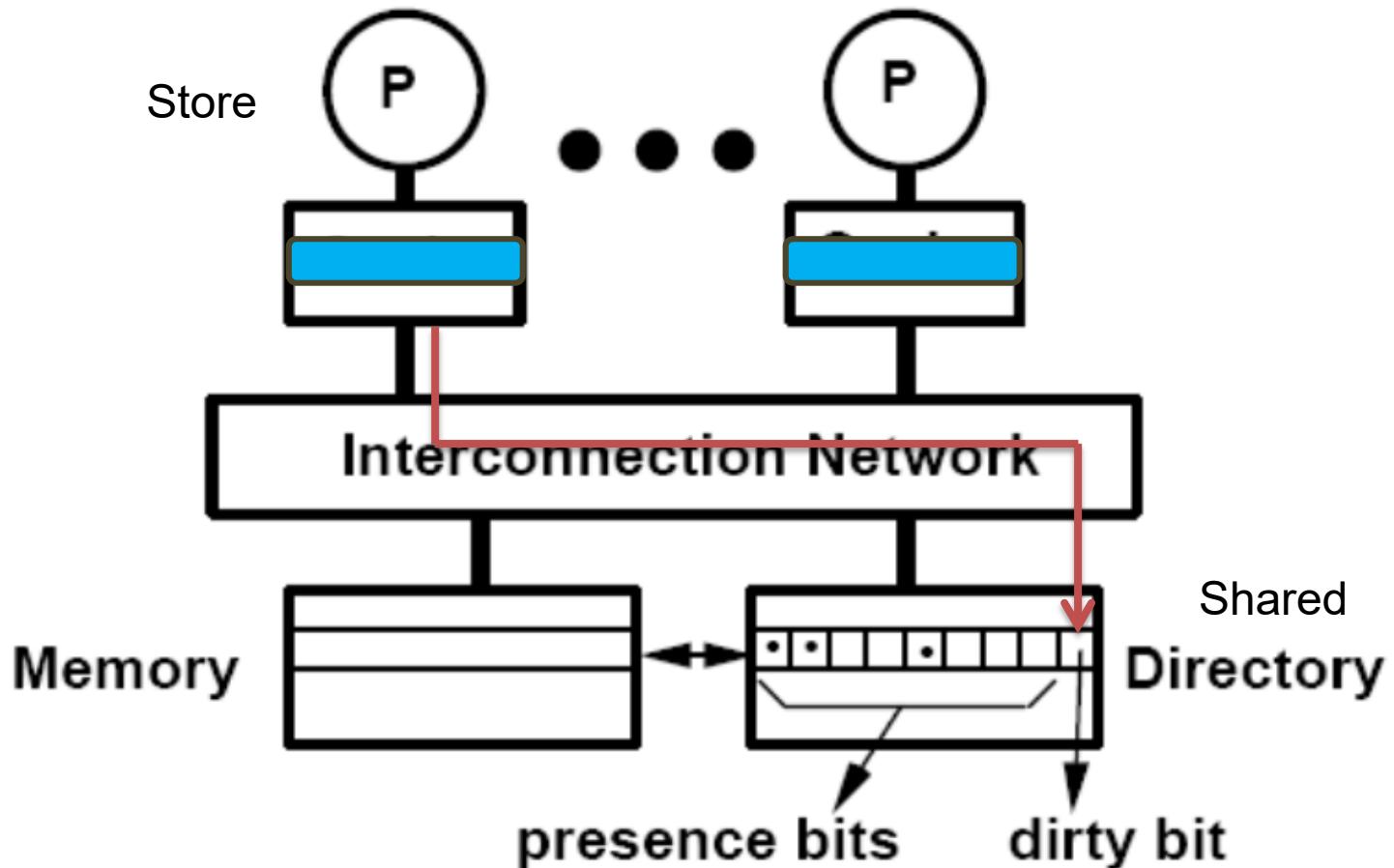
Directory-based cache coherence



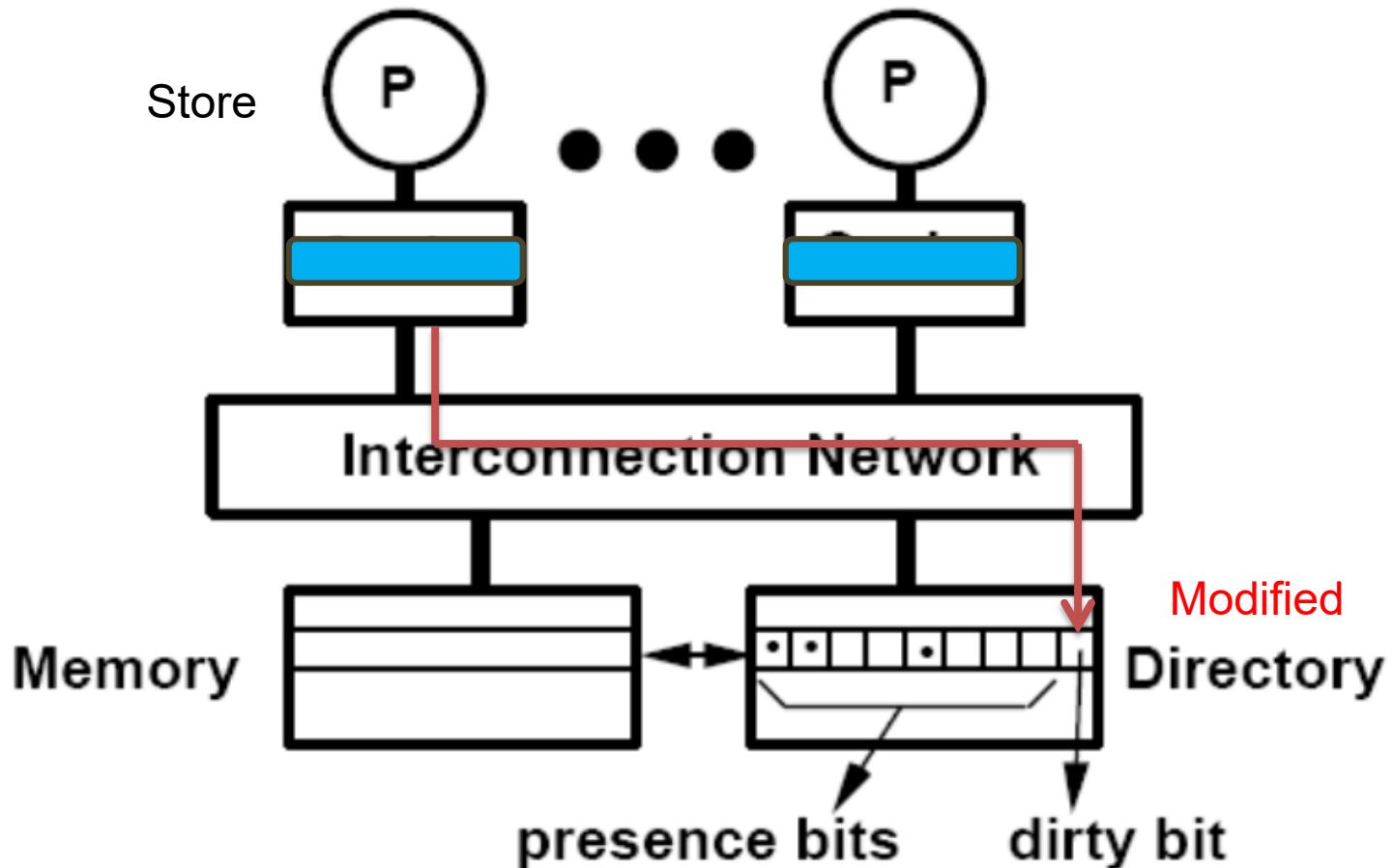
Directory-based cache coherence



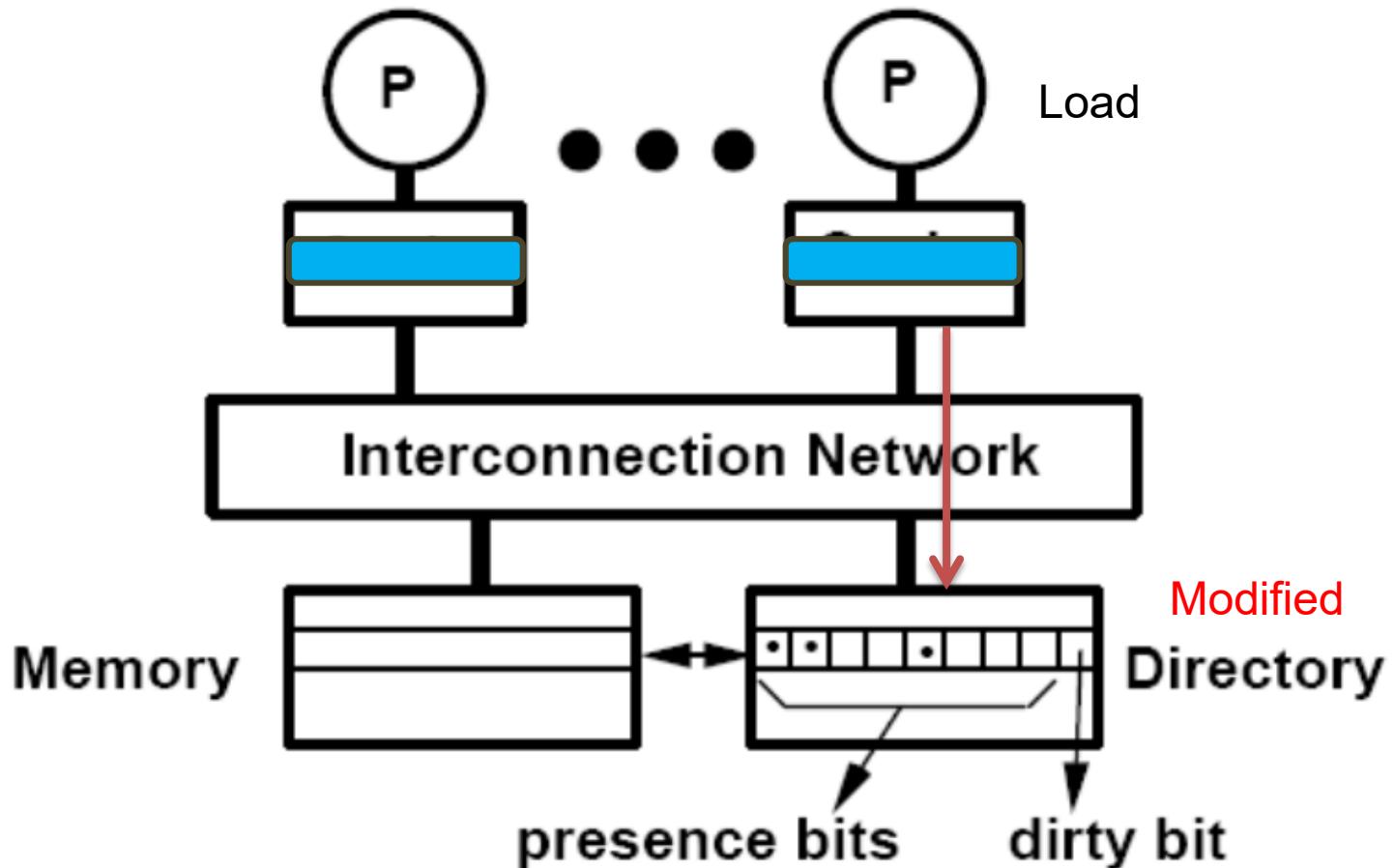
Directory-based cache coherence



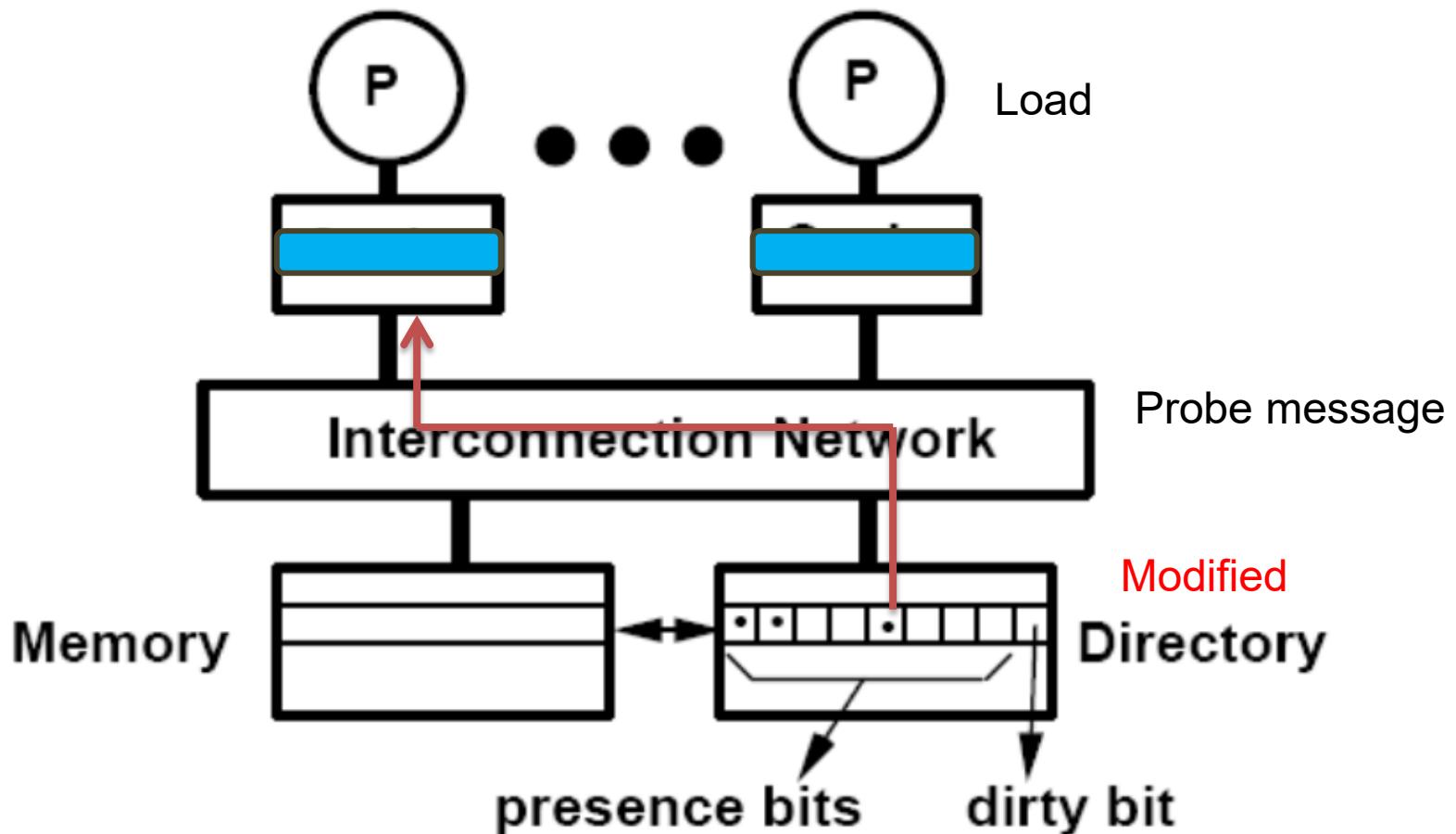
Directory-based cache coherence



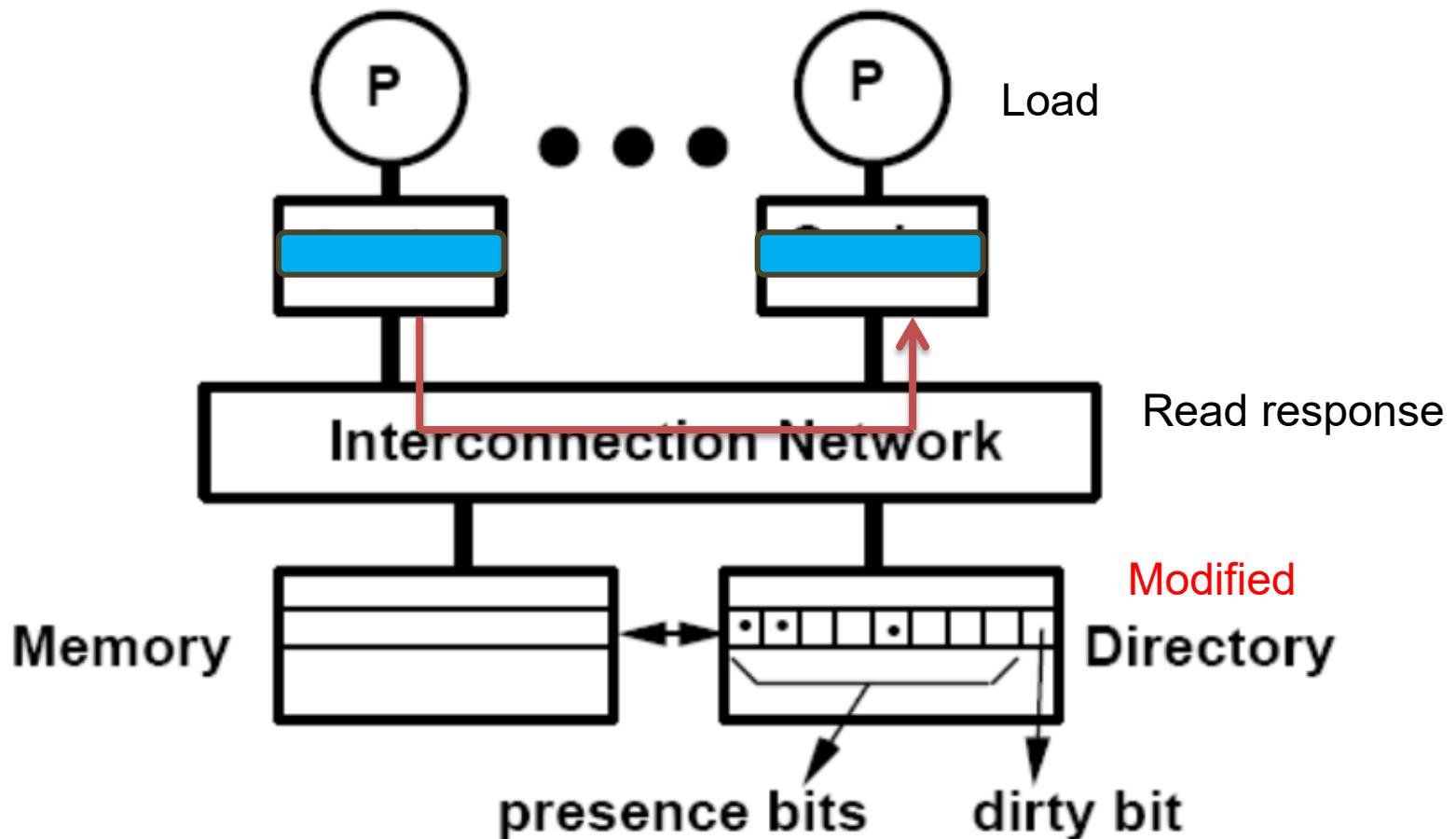
Directory-based cache coherence



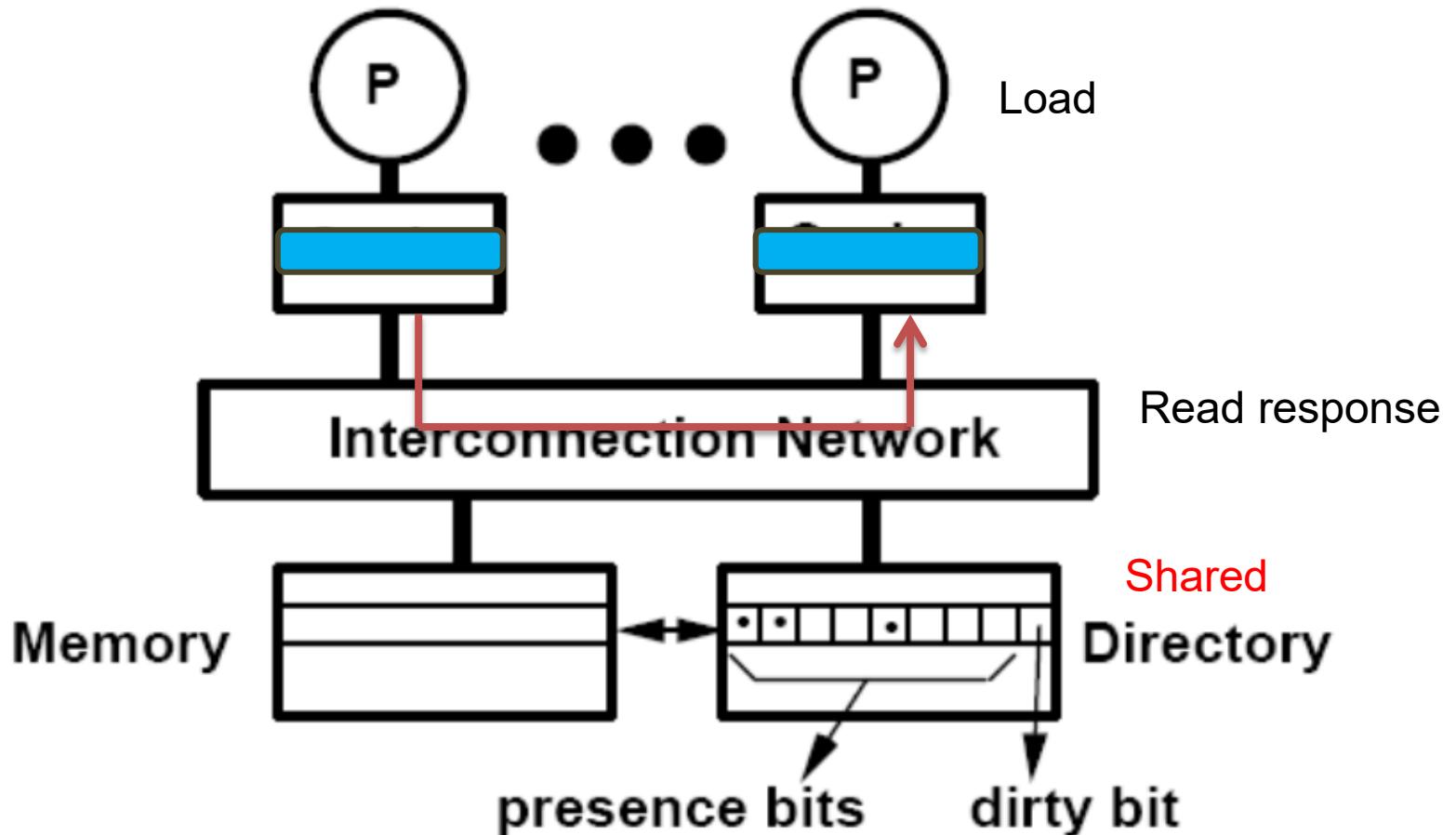
Directory-based cache coherence



Directory-based cache coherence



Directory-based cache coherence



A few notes on cache coherence

There may be more than a single directory
Especially for NUMA systems

Interconnect structure affects cache coherence performance

Directory is just one possible implementation
Snooping is another commonly used approach

Intuition of the collapse

Key point: read with modified cache line have to get data back from the owner

Coherence message are processed **sequentially**

Lock holder modifies cache holding the lock

Waiter is trying to read the lock

They get value of the lock from the lock holder

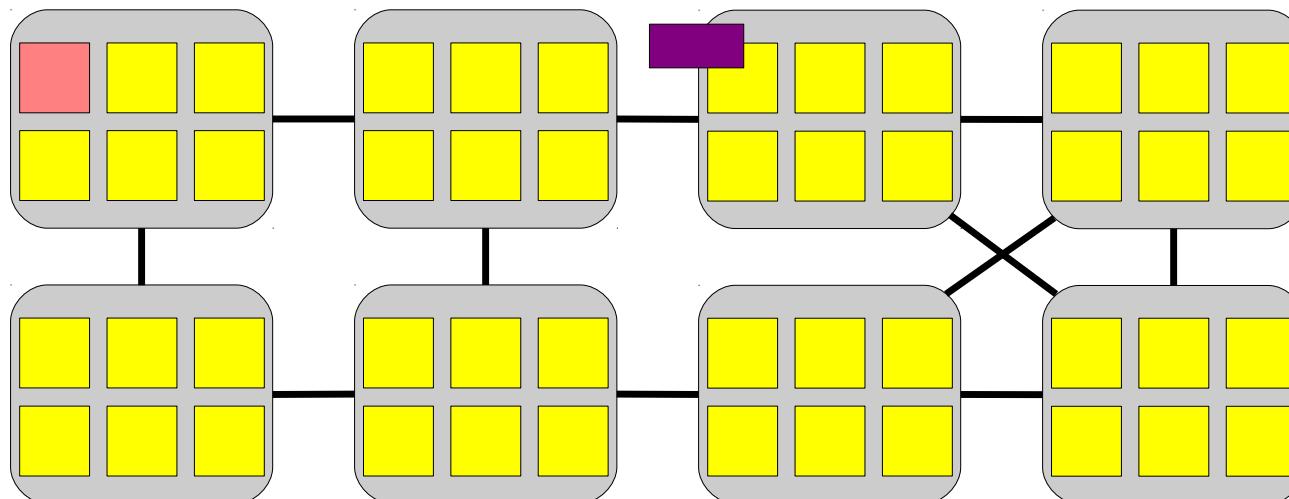
More readers means the next lock holder needs to wait more time to get the lock

Allocate a ticket read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



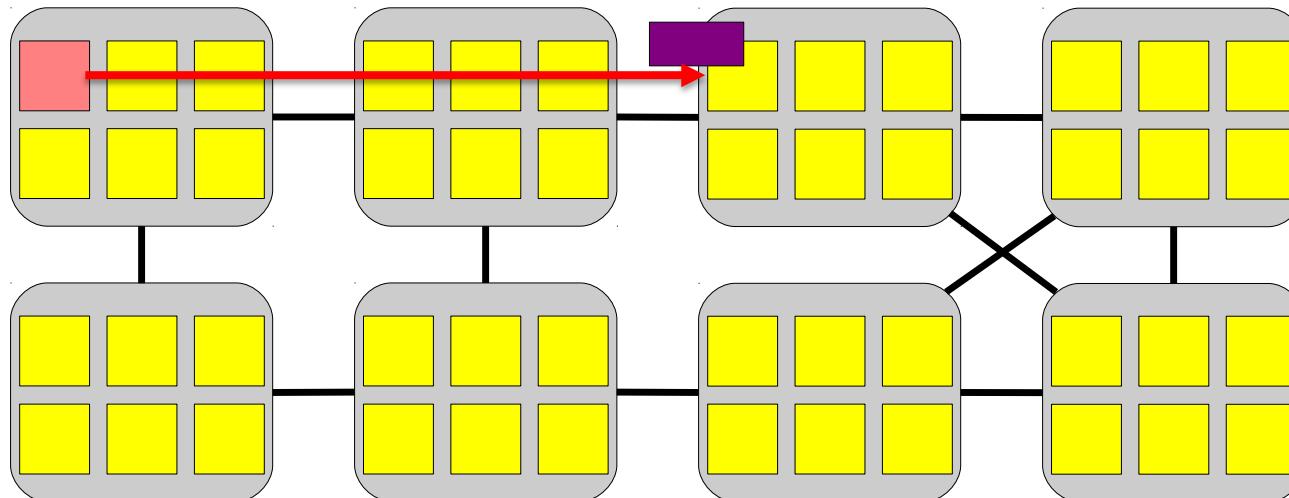
Allocate a ticket read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



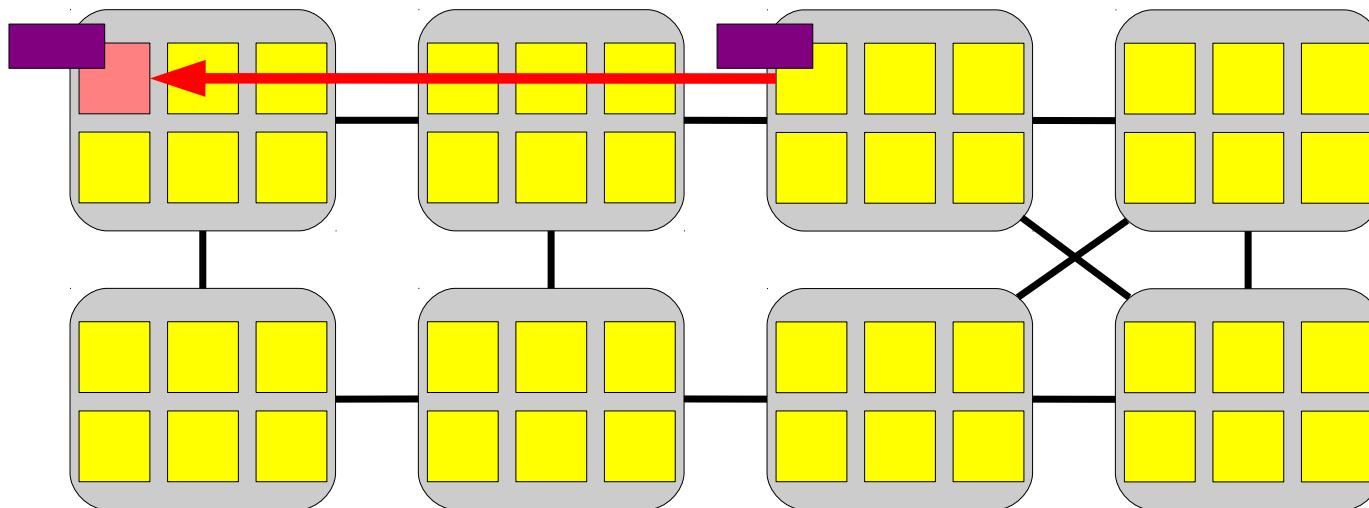
Allocate a ticket read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

cache coherence message



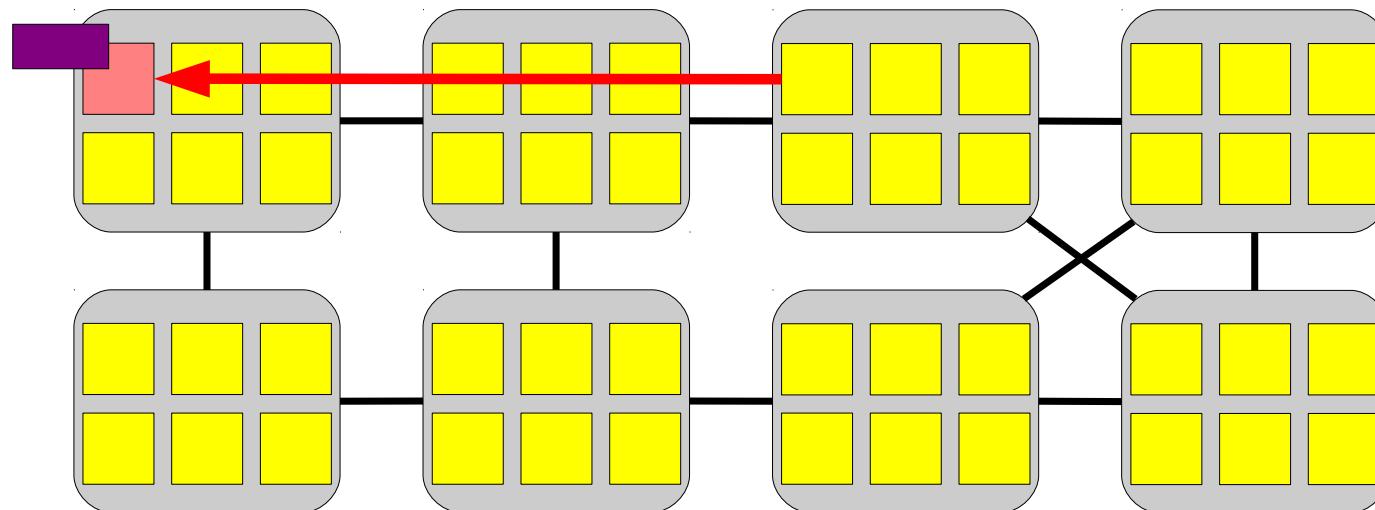
Allocate a ticket read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

120 ~ 420 cycles

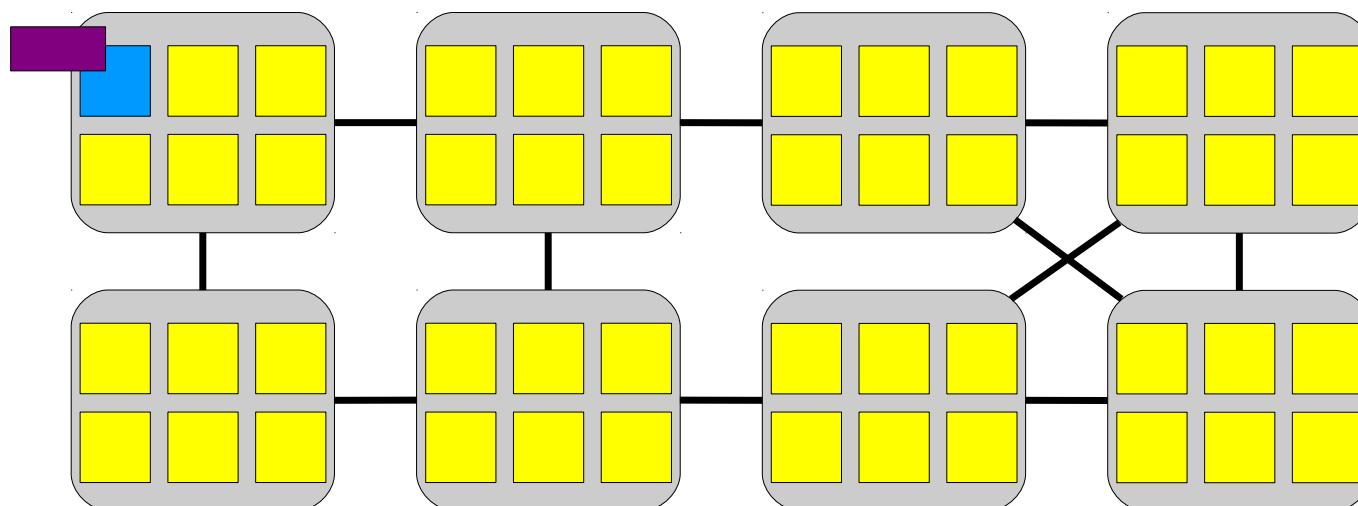


update the ticket value

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

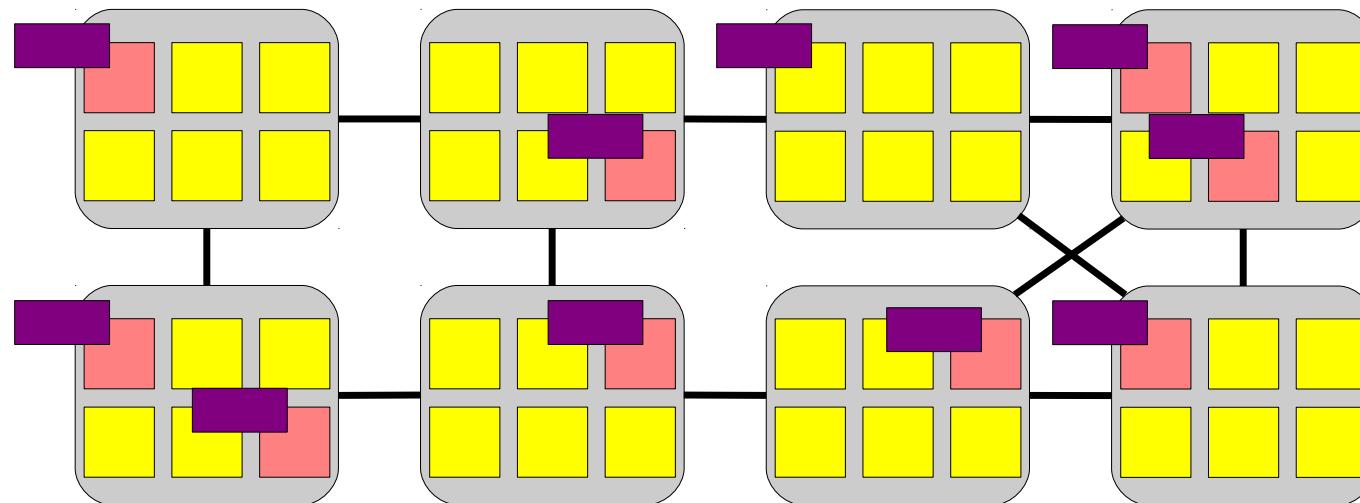
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



Lock shared by all core

```

void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}

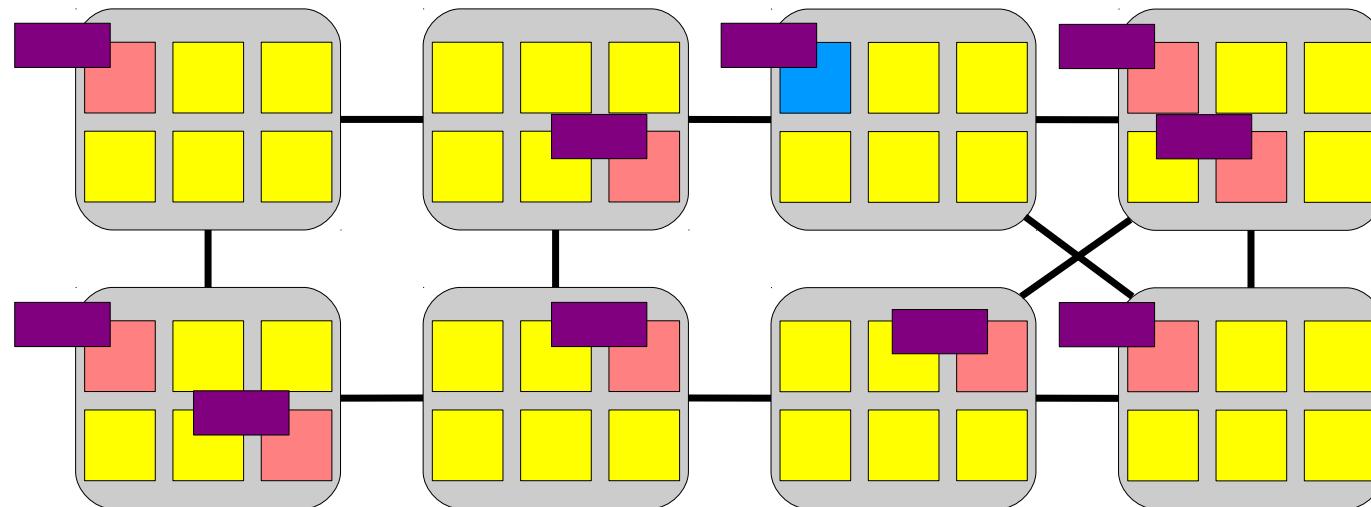
```

```

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

```



Lock holder update ticket

```

void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}

```

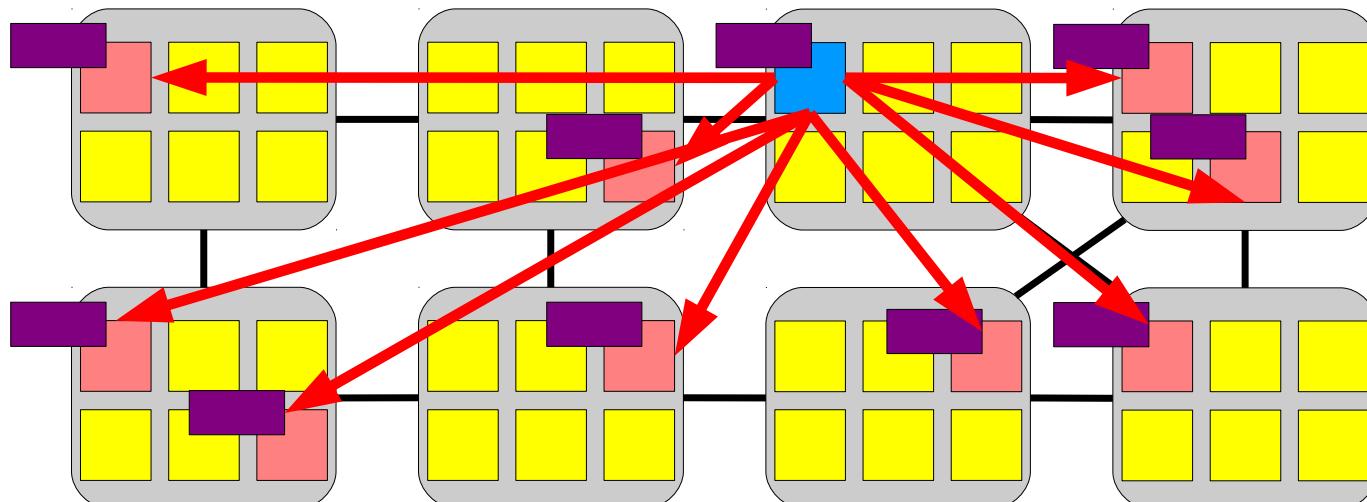
```

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

```

Invalidate message



Lock holder update ticket

```

void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}

```

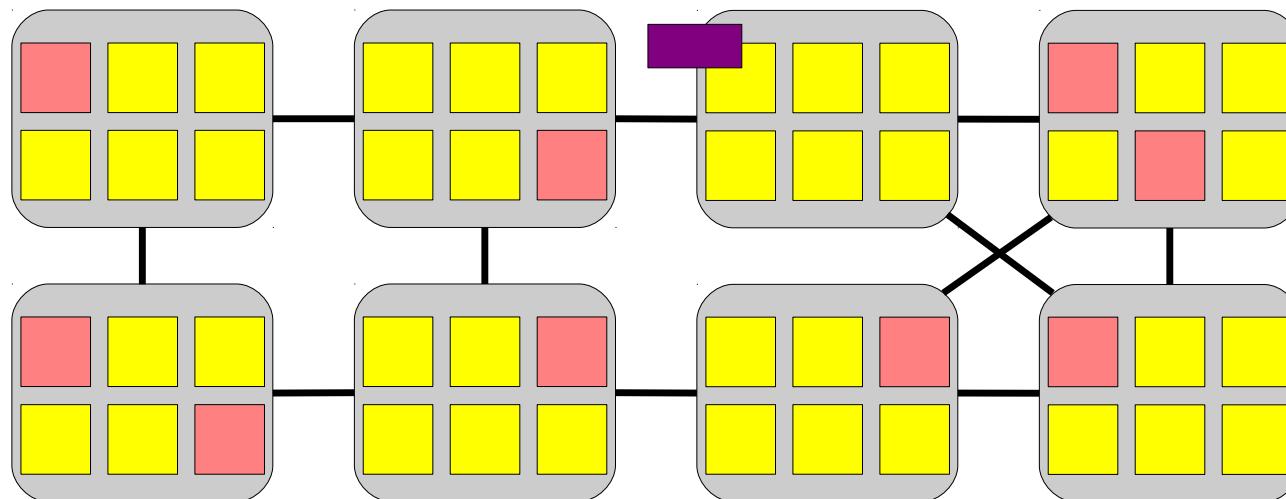
```

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

```

Only lock holder has lock in cache

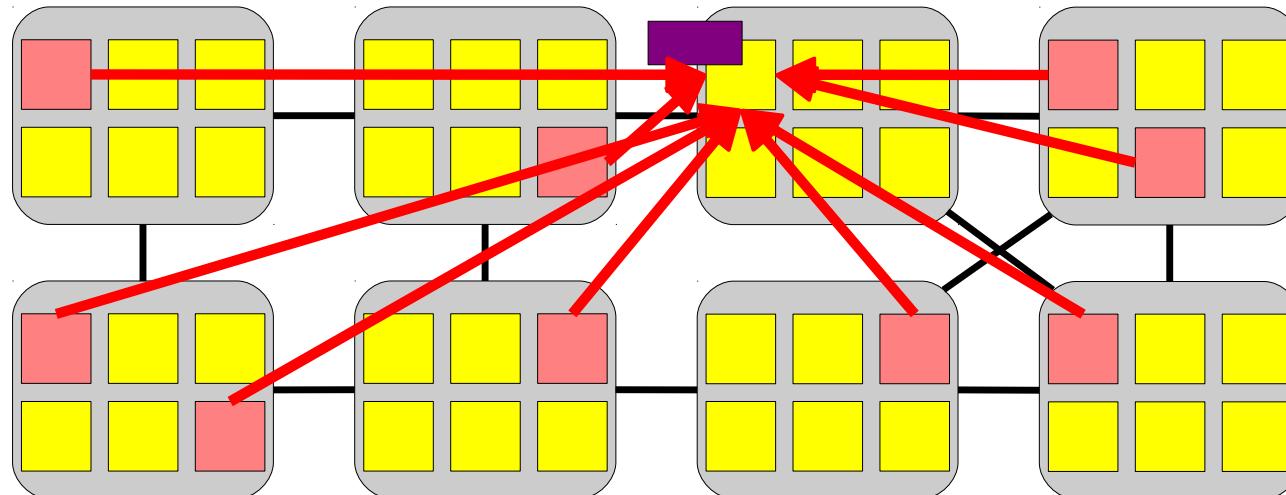


Lock holder update ticket

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



All waiters read the lock

```

void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}

```

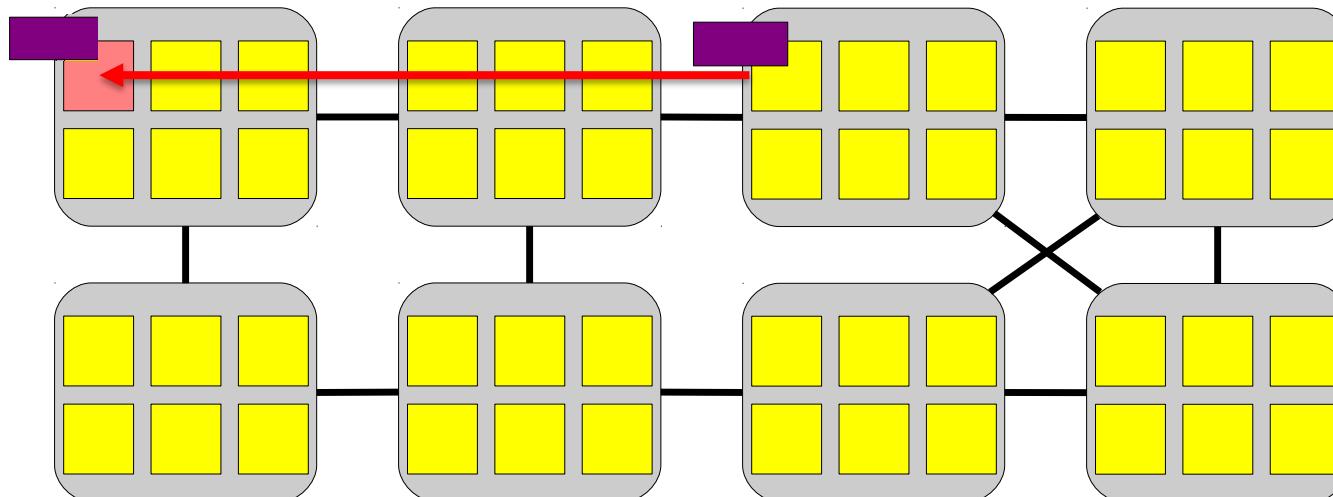
```

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

```

500 ~ 4000 cycles!



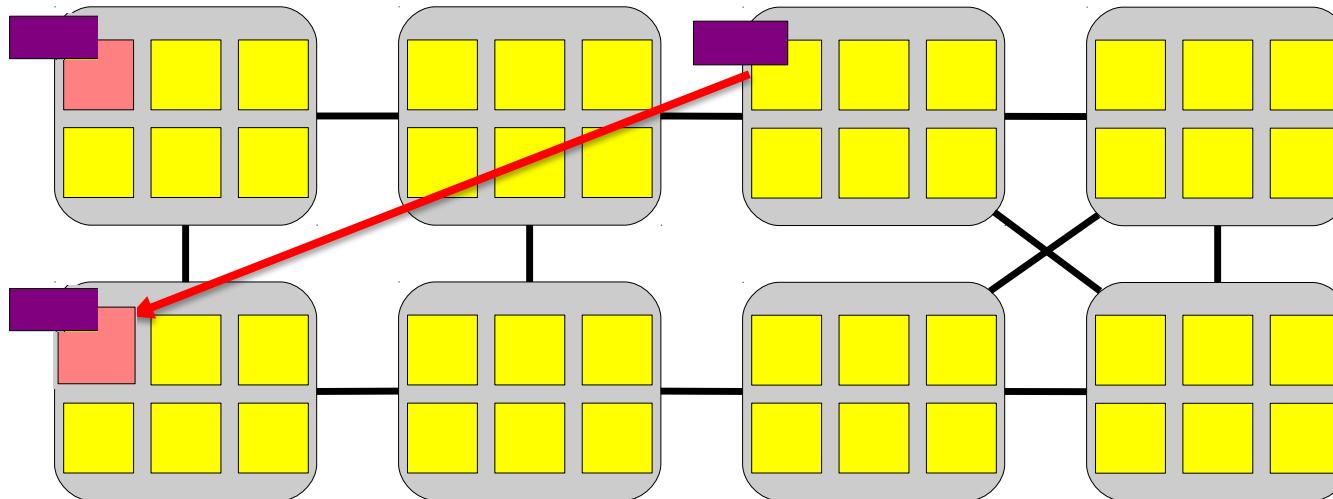
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



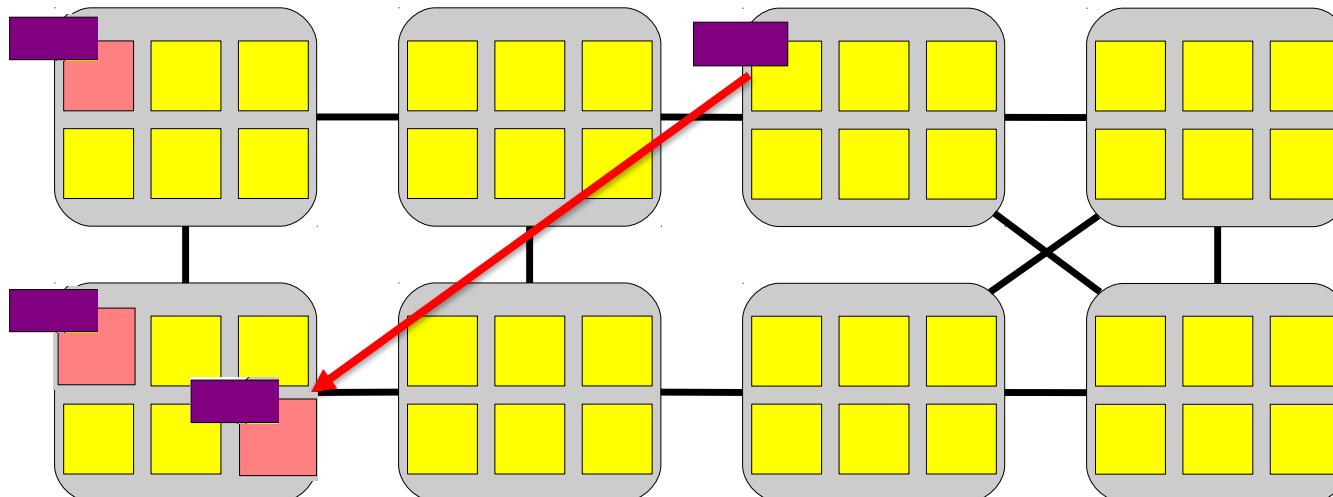
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Reply read request one by one



All waiters read the lock

```

void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}

```

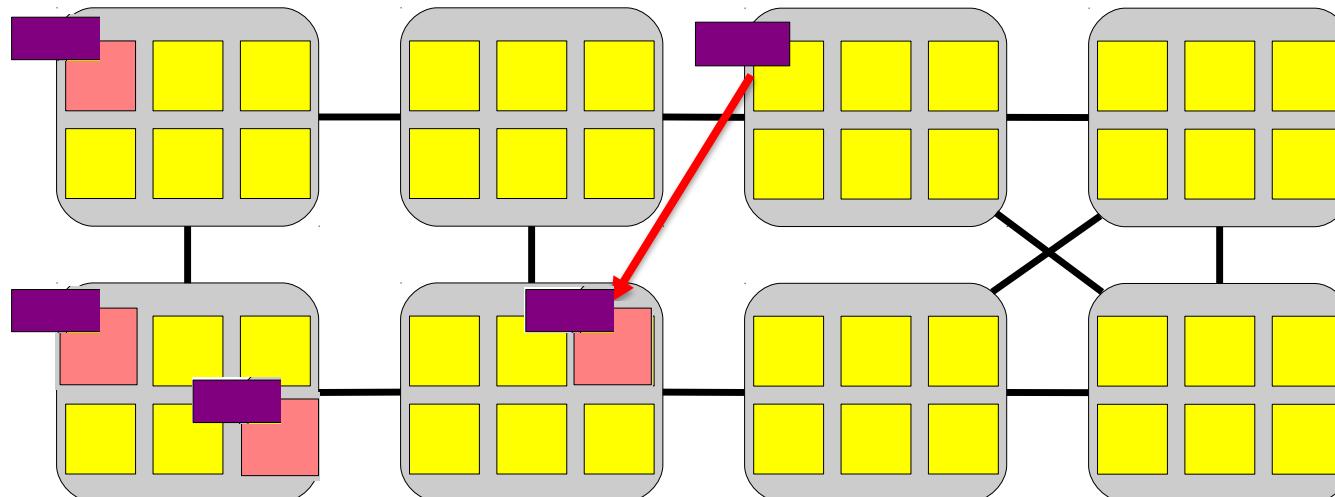
Previous lock holder notifies next lock holder after sending out $N/2$ replies

```

void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}

```



All waiters read the lock

SCALABLE LOCKING

Making ticket spinlock scalable

Common way: use proportional back-off

```
void spin_lock(spinlock_t *l) {  
    int t = atomic_xadd(&l->next_ticket);  
    while (t != lock->current_ticket)  
        // wait more time with each failure  
        ;  
}
```

Why this would work?

Is this a good way?

Example Fix

```
-- arch/x86/kernel/smp.c | 23 ++++++-----  
1 file changed, 20 insertions(+), 3 deletions(-)  
diff --git a/arch/x86/kernel/smp.c b/arch/x86/kernel/smp.c  
index 20da354..aa743e9 100644  
--- a/arch/x86/kernel/smp.c  
+++ b/arch/x86/kernel/smp.c  
@@ -117,11 +117,28 @@ static bool smp_no_nmi_ipi = false;  
 */  
void ticket_spin_lock_wait(arch_spinlock_t *lock, struct __raw_tickets inc)  
{  
+    __ticket_t head = inc.head, ticket = inc.tail;  
+    __ticket_t waiters_ahead;  
+    unsigned loops;  
+  
+    for (;;) {  
-        cpu_relax();  
-        inc.head = ACCESS_ONCE(lock->tickets.head);  
+        waiters_ahead = ticket - head - 1;  
+        /*  
+         * We are next after the current lock holder. Check often  
+         * to avoid wasting time when the lock is released.  
+         */  
+        if (!waiters_ahead) {  
+            do {  
+                cpu_relax();  
+            } while (ACCESS_ONCE(lock->tickets.head) != ticket);  
+            break;  
+        }  
+        loops = 50 * waiters_ahead;  
+        while (loops--)  
+            cpu_relax();  
  
-        if (inc.head == inc.tail)  
+        head = ACCESS_ONCE(lock->tickets.head);  
+        if (head == ticket)  
+            break;  
    }  
}
```

Linus' Response

<http://linux-kernel.2935.n7.nabble.com/PATCH-v5-0-5-x86-smp-make-ticket-spinlock-proportional-backoff-w-auto-tuning-td596698i20.html>

So I claim:

- it's *really* hard to trigger in real loads on common hardware.
- if it does trigger in any half-way reasonably common setup (hardware/software), we most likely should work really hard at fixing the underlying problem, not the symptoms.
- we absolutely should *not* pessimize the common case for this

Using scalable locks

Many existing scalable locks

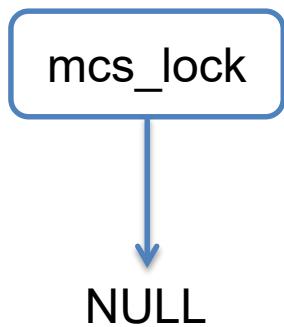
Main idea is to avoid contending on a single cache line

Example

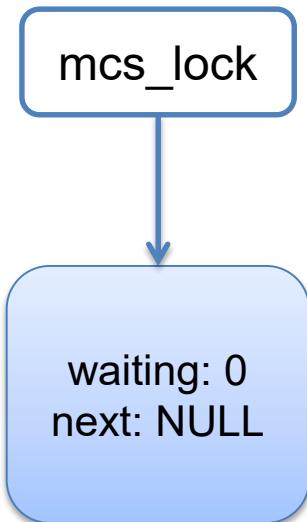
MCS (John M. Mellor-Crummey and Michael L. Scott)

K42

General idea of MCS lock



General idea of MCS lock

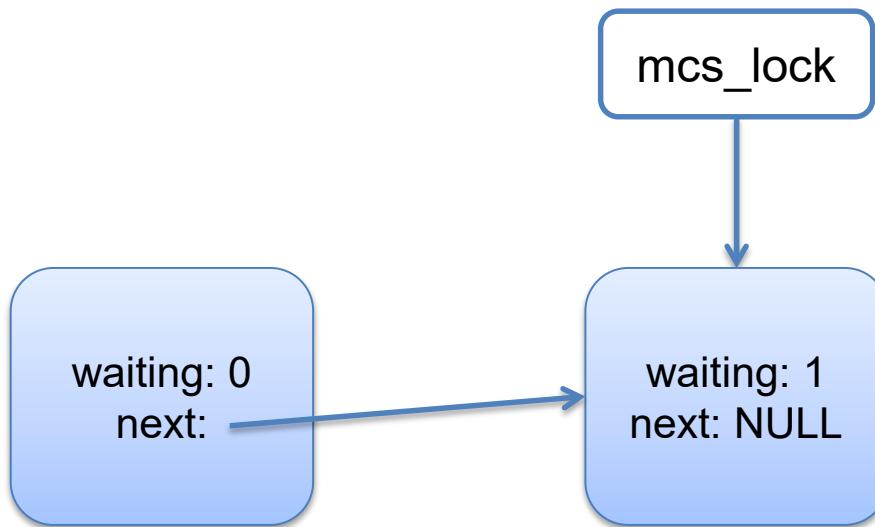


Use compare and swap to change
mcs_lock point to self node

Check previous node

- NULL in this case, no need to wait

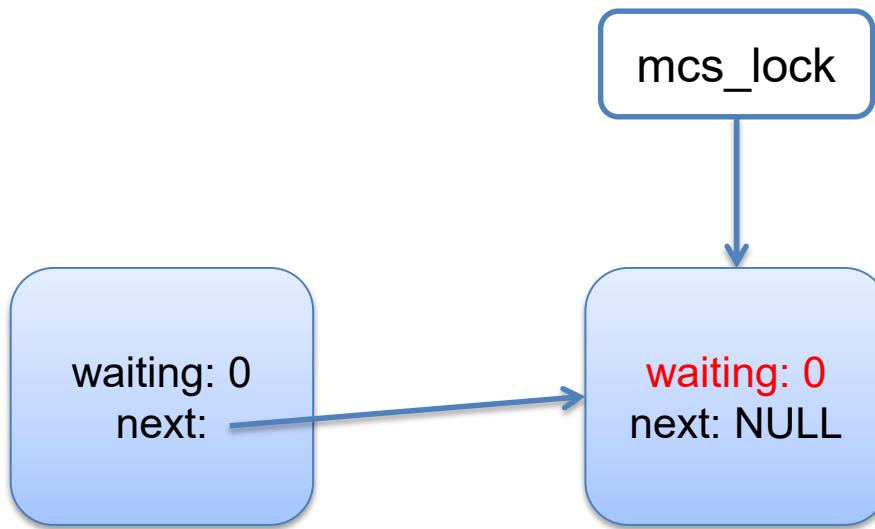
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

General idea of MCS lock



Previous node is not NULL
Current waiting is 1

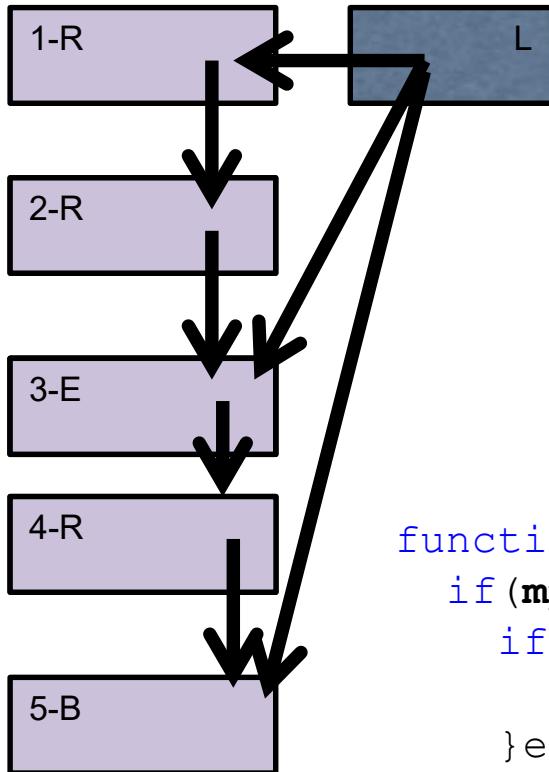
Wait until lock holder
set waiting to 0

Code

```
mcs_node{  
    mcs_node next;  
    int is_locked;  
}  
  
mcs_lock{  
    mcs_node queue;  
}  
  
function lock(mcs_lock lock, mcs_node my_node) {  
    my_node.next = NULL;  
    mcs_node predecessor =  
        fetch_and_store(lock.queue, my_node);  
    if(predecessor != NULL) {  
        my_node.is_locked = true;  
        predecessor.next = my_node;  
        while(my_node.is_locked) {};  
    }  
}
```

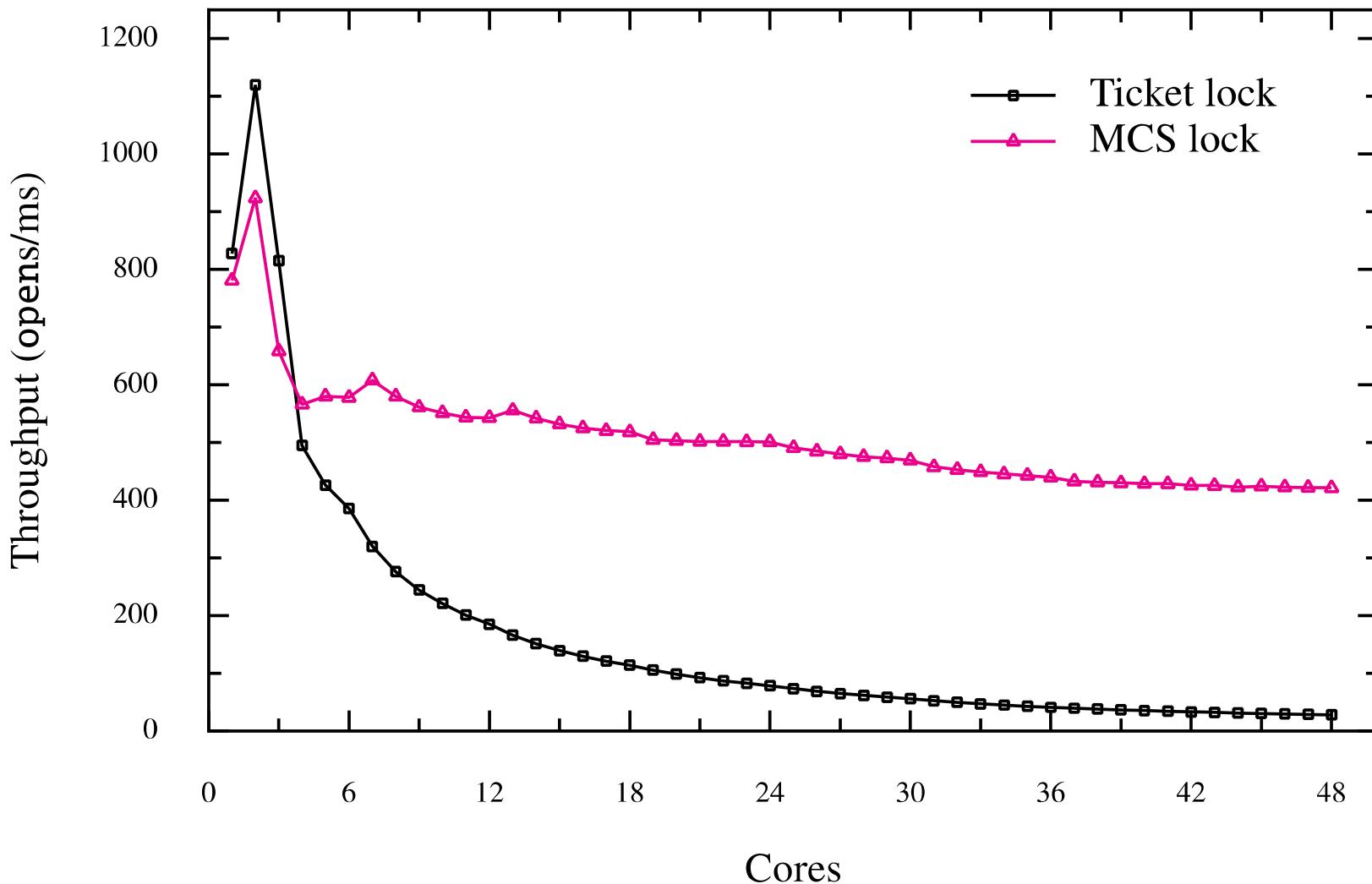
```
function unlock(mcs_lock lock, mcs_node my_node) {  
    if (my_node.next == NULL) {  
        if (compare_and_swap(lock.queue, my_node, NULL) {  
            return;  
        } else {  
            while (my_node.next == NULL) {};  
        }  
    }  
  
my_node.next.is_locked = false;  
}
```

MCS Locks



```
function unlock(mcs_lock lock, mcs_node my_node) {  
    if (my_node.next == NULL) {  
        if (compare_and_swap(lock.queue, my_node, NULL) {  
            return;  
        } else {  
            while (my_node.next == NULL) {};  
        }  
    }  
    my_node.next.is_locked = false;  
}
```

Using scalable locks: FOPS



Synchronization Constructs

Zhaoguo Wang

Some slides adjusted from. Elsa L Gunter (UIUC), Jonathan Walpole (PSU)
Paul McKenney (IBM) Tom Hart (University of Toronto)

Outline

Recap

MCS Lock

Read-write Lock

Lock-free coordination

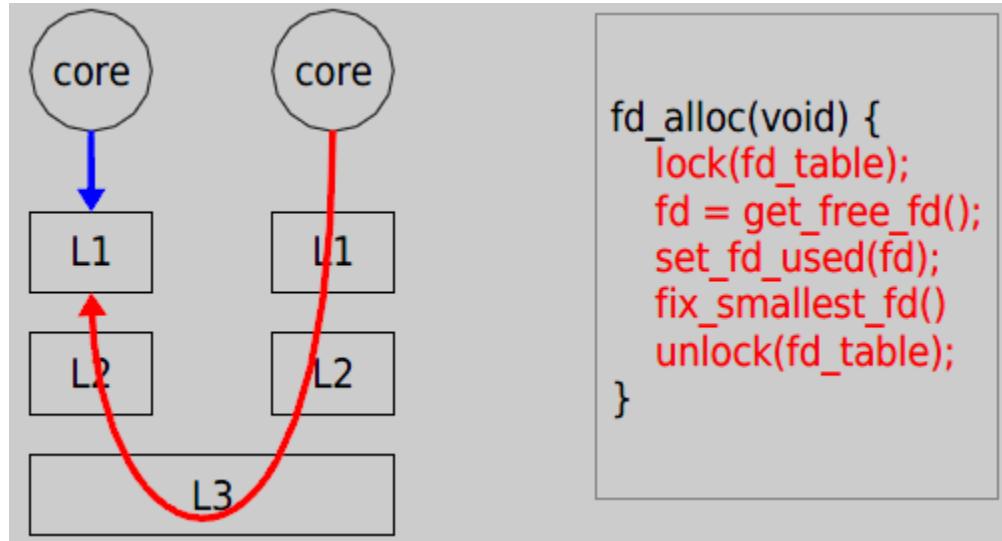
Recap: What is scalability?

Application does N times as much work on N cores
as it could on 1 core

Scalability may be limited by Amdahl's Law:

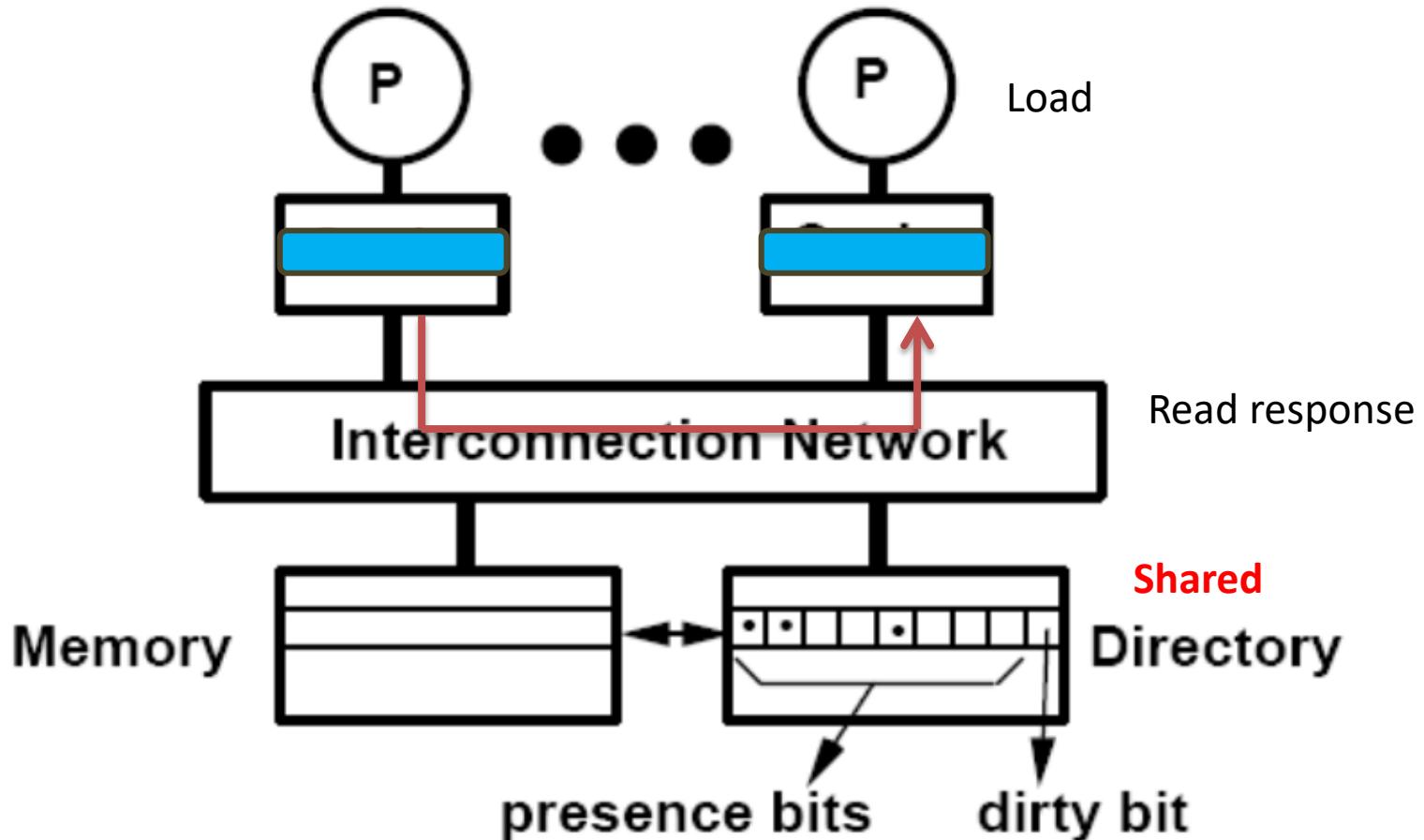
Locks, shared data structures, ... Shared hardware
(DRAM, NIC, ...)

Recap: Why throughput drops?



Now it takes 121 cycles!

Recap: Directory-based cache coherence

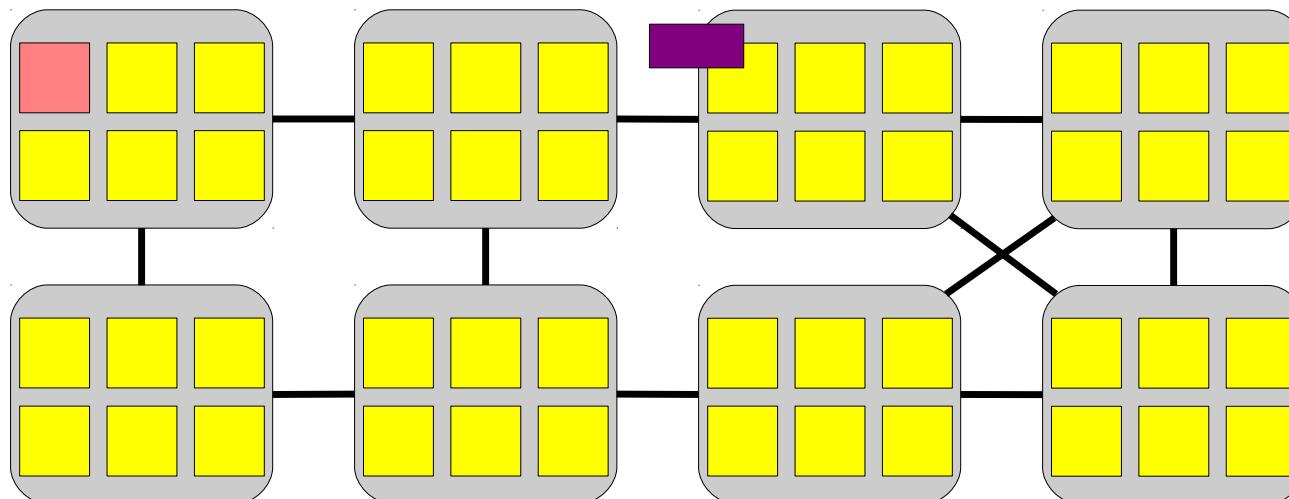


Recap: Allocate a ticket read current ticket and spin

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ; /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}

struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

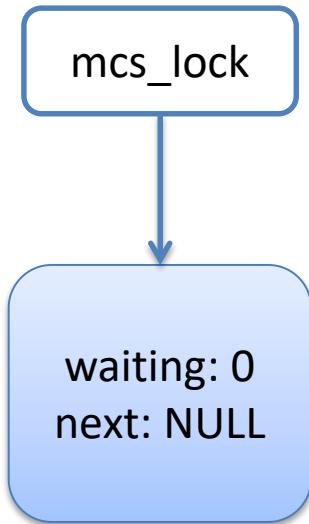


General idea of MCS lock

mcs_lock

NULL

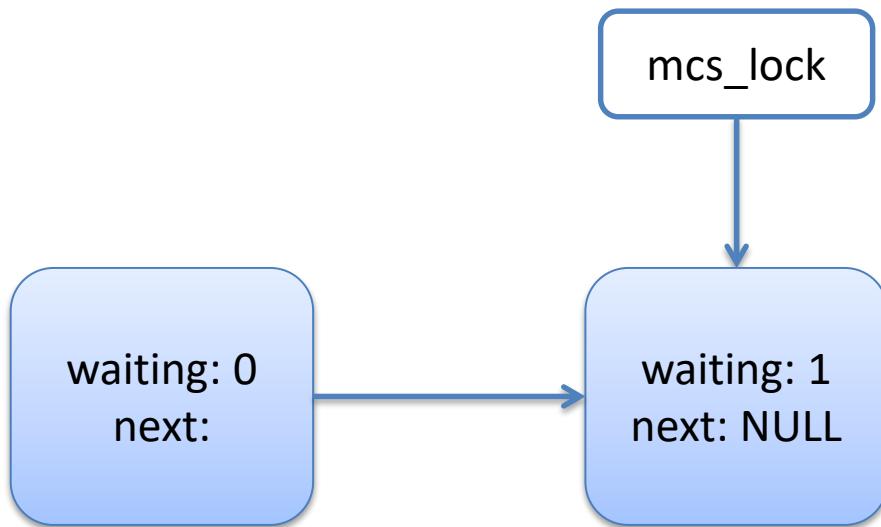
General idea of MCS lock



Use compare and swap to change
mcs_lock point to self node

Check previous node
NULL in this case, no need to wait

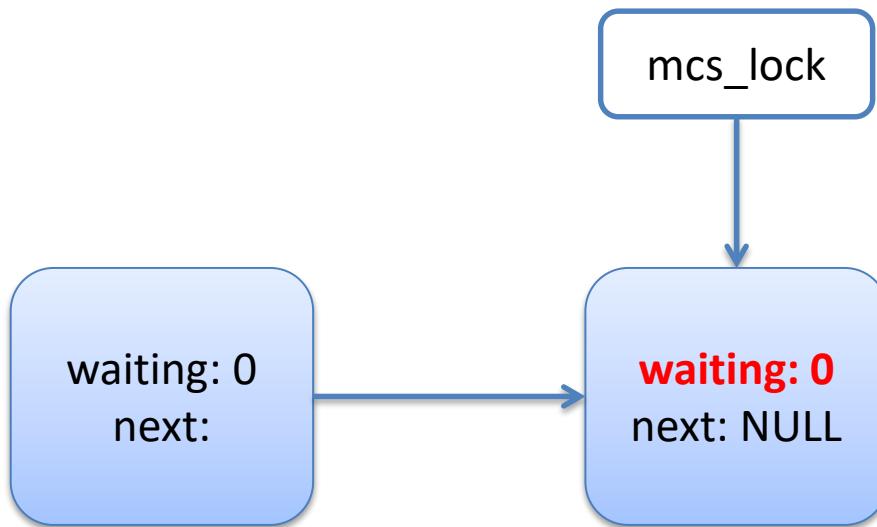
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

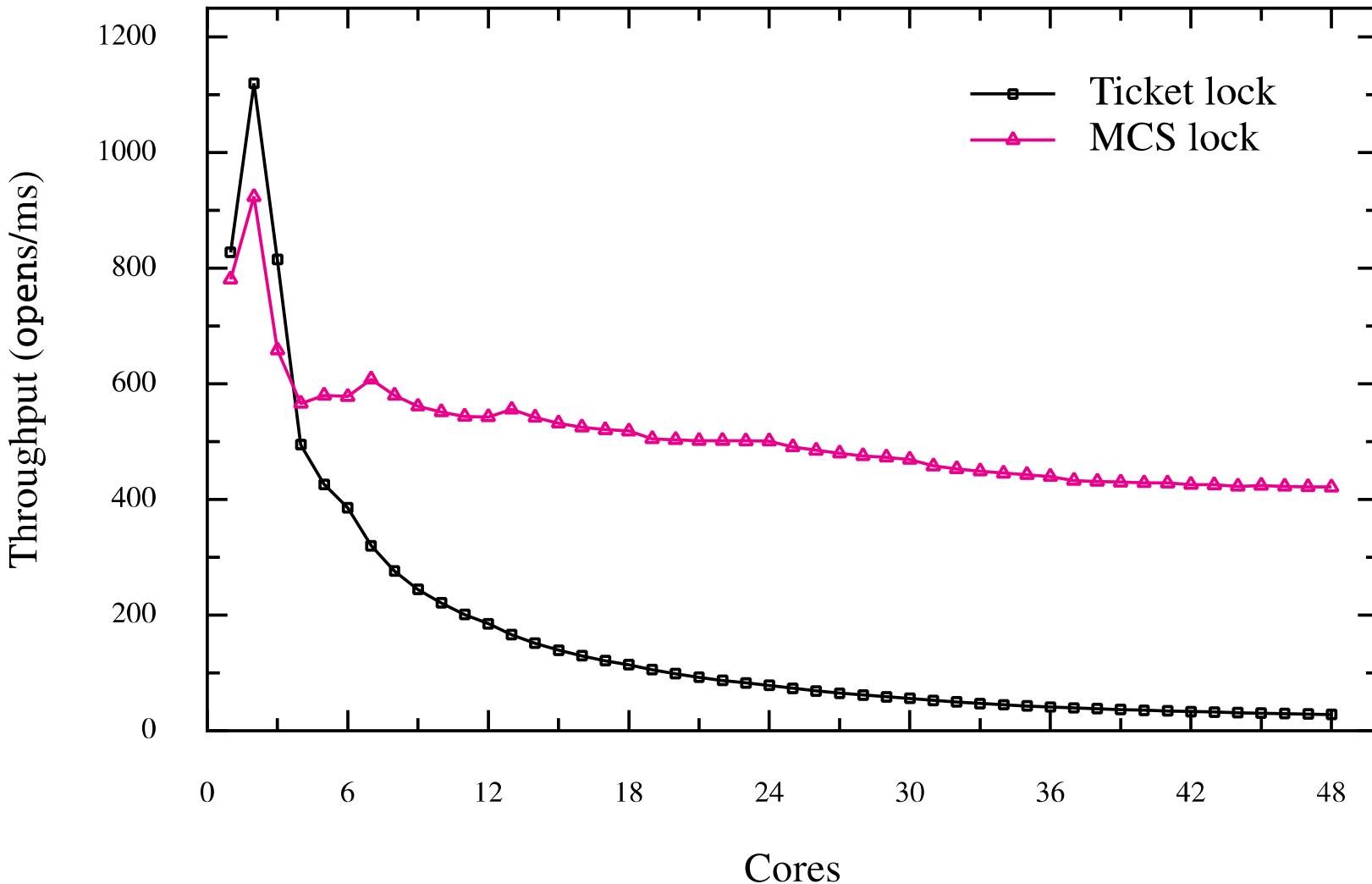
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

Using scalable locks: FOPS



Synchronization Constructs

Read-write Lock

Lock-free

READER-WRITER LOCK

Motivating Example: Calendar

Goal: online calendar for a class

Lots of people may read it at the same time

Only one person updates it (Prof, TAs)

Shared data

`map<date, listOfEvents> EventMap`

`listOfEvents GetEvents(date)`

`AddEvent(data, newEvent)`

Basic Code – Single Threaded

```
GetEvents(date) {  
    List events = EventMap.find(date).copy();  
    return events;  
}
```

```
AddEvent(data, newEvent) {  
    EventMap.find(date) += newEvent;  
}
```

Inefficient Multi-threaded code

```
GetEvents(date) {  
    lock(mapLock);  
    List events = EventMap.find(date).copy();  
    unlock(mapLock);  
    return events;  
}
```

```
AddEvent(data, newEvent) {  
    lock(mapLock);  
    EventMap.find(date) += newEvent;  
    unlock(mapLock);  
}
```

How to do with reader – writer locks?

```
GetEvents(date) {  
    List events = EventMap.find(date).copy();  
    return events;  
}
```

```
AddEvent(data, newEvent) {  
    EventMap.find(date) += newEvent;  
}
```

Reader – Writer Locks

Problem definition:

Shared data that will be read and written by multiple threads

Allow multiple readers to access shared data when no threads are writing data

A thread can write shared data only when no other thread is reading or writing the shared data

Interface

readerStart()

readerFinish()

writerStart()

writerFinish()

Many threads can be in between a **readStart** and **readerFinish**

Only one thread can be between **writerStart** and **writierFinish**

How to do with reader – writer locks?

```
GetEvents(date) {  
    readerStart(mapRWLock);  
    List events = EventMap.find(date).copy();  
    readerFinish(mapRWLock);  
    return events;  
}  
  
AddEvent(data, newEvent) {  
    writerStart(mapRWLock);  
    EventMap.find(date) += newEvent;  
    writerFinish(mapRWLock);  
}
```

Additional Layer of Synchronization

Concurrent programs

Even higher-level synchronization

High-level synchronization provided by software

Low-level atomic operations provided by hardware

Reader – Writer Locks using Monitors

Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code

Central Questions:

Shared Data?

Ordering Constraints?

How many Condition Variables?

Reader – Writer Locks using Monitors

Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code

Central Questions:

Shared Data? NumReaders, NumWriters

Ordering Constraints?

How many Condition Variables?

Reader – Writer Locks using Monitors

Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code

Central Questions:

Shared Data? NumReaders, NumWriters

Ordering Constraints?

readerStart must wait if there are writers

writerStart must wait if there are readers or writes

How many Condition Variables?

Reader – Writer Locks using Monitors

Note: Implement Reader/Writer Locks as an abstractions, *not* as an integrated part of code

Central Questions:

Shared Data? NumReaders, NumWriters

Ordering Constraints?

readerStart must wait if there are writers

writerStart must wait if there are readers or writes

How many Condition Variables?

One: condRW (no readers or writers)

Basic Implementation

```
readerStart() {
```

```
readerFinish() {
```

```
}
```

```
}
```

Basic Implementation

```
readerStart() {  
    lock(lockRW);  
  
    while(numWriters > 0){  
        wait(lockRW, condRW);  
    };  
  
    numReaders++;  
  
    unlock(lockRW);  
}  
  
readerFinish() {  
    lock(lockRW);  
  
    numReaders--;  
  
    broadcast(lockRW, condWR);  
  
    unlock(lockRW);  
}
```

Basic Implementation

```
writerStart() {  
    lock(lockRW);  
  
    while(numReaders > 0 ||  
          numWriters > 0){  
        wait(lockRW, condRW);  
    };  
  
    numWriters++;  
  
    unlock(lockRW);  
}  
  
writerFinish() {  
    lock(lockRW);  
  
    numWriters--;  
  
    broadcast(lockRW, condWR);  
  
    unlock(lockRW);  
}
```

Better Implementation

```
readerStart() {  
    lock(lockRW);  
  
    while(numWriters > 0){  
        wait(lockRW, condRW);  
    };  
  
    numReaders++;  
  
    unlock(lockRW);  
}  
  
readerFinish() {  
    lock(lockRW);  
  
    numReaders--.  
  
    if(numReaders == 0){  
        signal(lockRW, condWR);  
    };  
  
    unlock(lockRW);  
}
```

Reader/writer spin locks (rwlock) in Linux

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;  
  
read_lock(&mr_rwlock);  
/* critical section (read only) ... */  
read_unlock(&mr_rwlock);  
  
write_lock(&mr_rwlock);  
/* critical section (read and write) ... */  
write_unlock(&mr_rwlock);
```

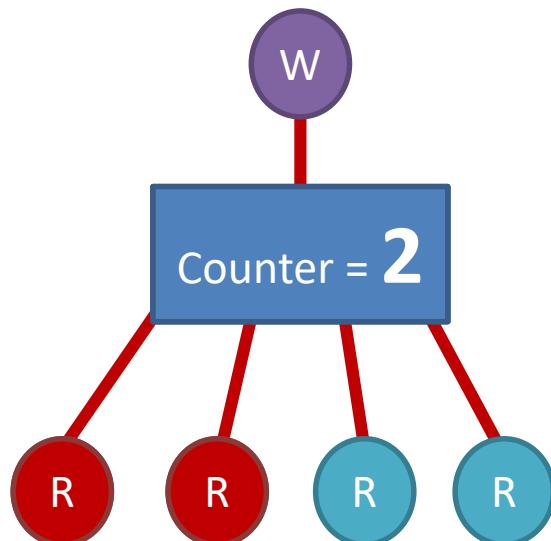
Scalability of rwlock?

Reading

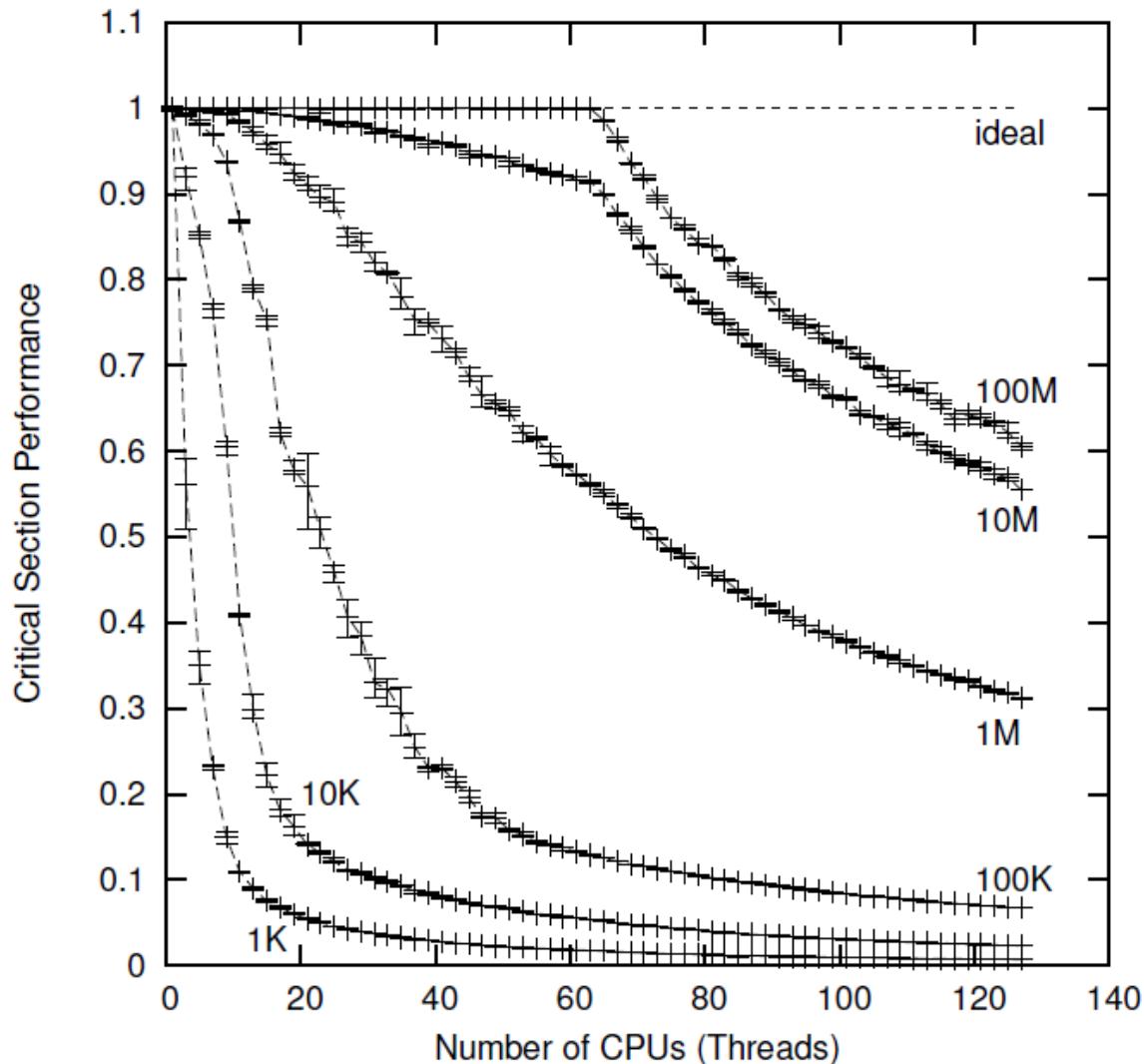
Counter -> #readers

Writing

Counter = -1



Traditional RW lock performance



Problem

Reader Lock Acquisition
add_and_fetch(&counter)

Need to wait for message (current #reader)
Need to send message (next reader)

Acquisition serialized by messages!!
Occupies a great deal of message bandwidth!!

Solutions

Idea 1

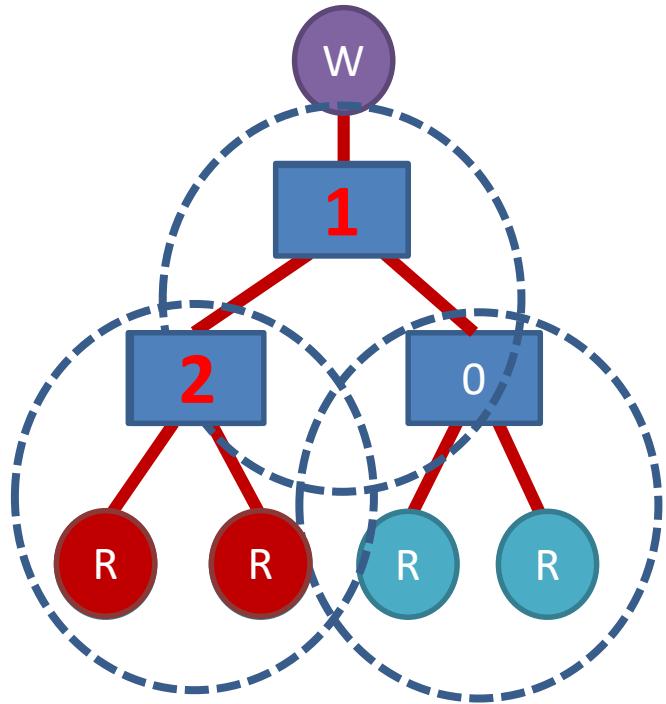
Reduce time to wait for message (GOLL)

Idea 2

Reduce #message (BR lock)

GOLL

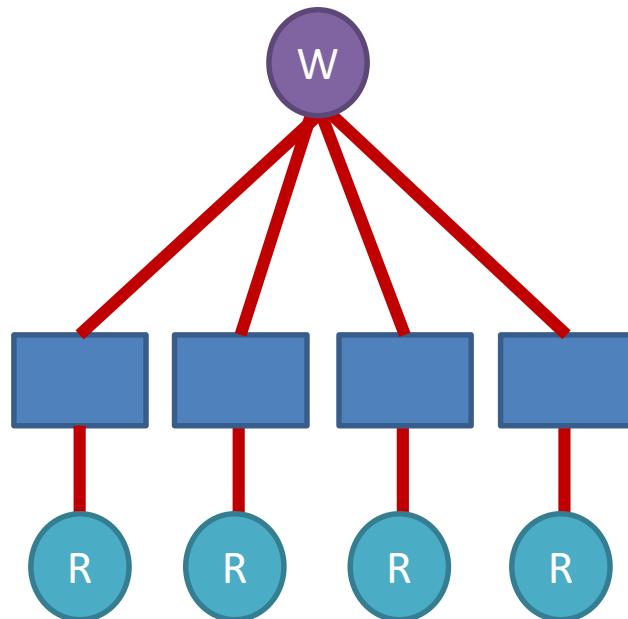
Reduce time to wait for message



Waiting domain is split into pieces
Reduce #remote messages

BR lock

Reduce #message



No reader messages if there's no writer

RCU SYNCHRONIZATION

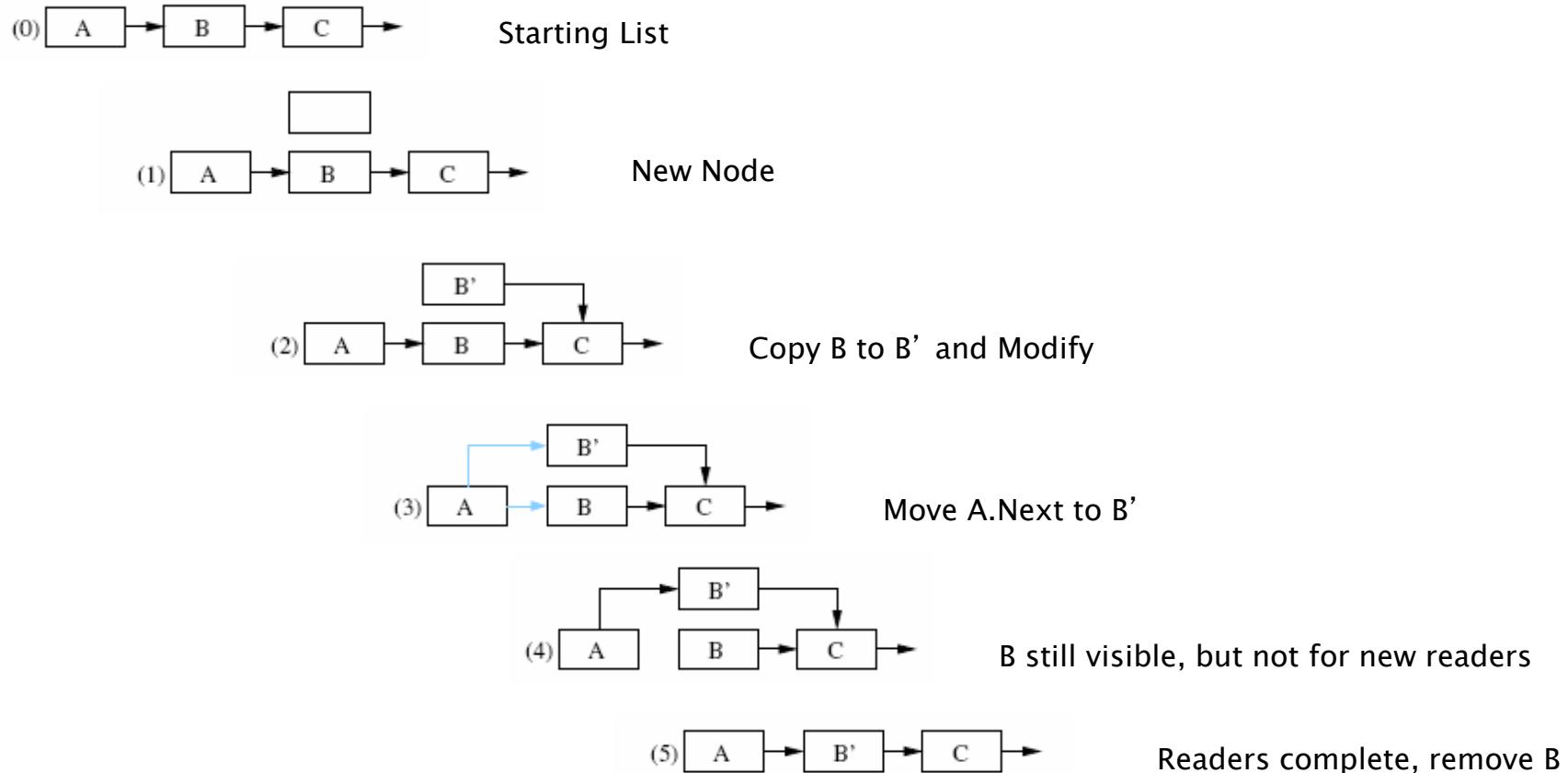
Overview of the basic idea

- Writers create new versions
 - Using locking or NBS to synchronize with each other
 - Register *call-backs* to destroy old versions when safe
 - `call_rcu()` primitive registers a call back with a reclaimer
 - Call-backs are deferred and memory reclaimed in batches
- Readers do not use synchronization
 - While they hold a reference to a version it will not be destroyed
 - Completion of read-side critical sections is “inferred” by the reclaimer from observation of quiescent states

RCU Overview

- Publish–Subscribe
 - insertion
 - reader–writer synchronization
- Wait for pre-existing readers to complete
 - deletion
 - change – wait for readers – free
 - safe memory reclamation
- Maintain multiple versions of update objects
 - for readers

RCU – List Update– Basic Strategy



Quiescent states and grace periods

- Example quiescent states
 - Context switch (non-preemptive kernels)
 - Voluntary context switch (preemptive kernels)
 - Kernel entry/exit
 - Blocking call
- Grace periods
 - A period during which every CPU has gone through a quiescent state

Read-Copy Update Performance

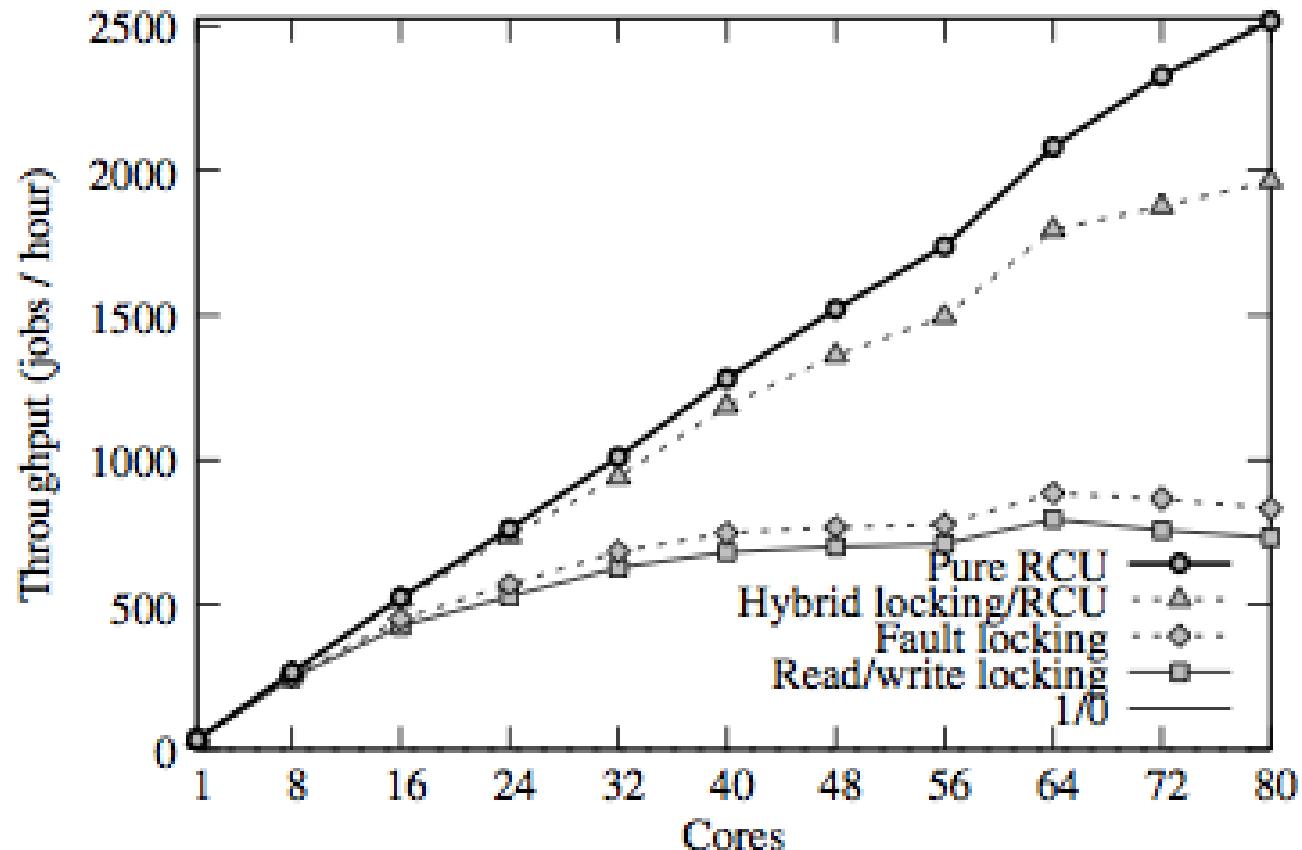


Figure 13. Metis throughput for each page fault concurrency design.

LOCK-FREE SYNCHRONIZATION

Optimistic synchronization

- Use lock-free, “optimistic” synchronization
 - Execute the critical section unconstrained, and check at the end to see if you were the only one
 - If so, continue. If not roll back and retry
- Synthesis uses no locks at all!

Locking is pessimistic

- Murphy's law: “If it can go wrong, it will...”
- In concurrent programming:
 - “If we can have a race condition, we will...”
 - “If another thread could mess us up, it will...”
- Solution:
 - Hide the resources behind locked doors
 - Make everyone wait until we're done
 - That is...if there was anyone at all
 - We pay the same cost either way

Optimistic synchronization

- The common case is often little or no contention
 - Or at least it should be!
 - Do we really need to shut out the whole world?
 - Why not proceed optimistically and only incur cost if we encounter contention?
- If there's little contention, there's no starvation
 - So we don't need to be “wait-free” which guarantees no starvation
 - Lock-free is easier and cheaper than wait-free
- Small critical sections really help performance

How does it work?

- Copy
 - Write down any state we need in order to retry
- Do the work
 - Perform the computation
- Atomically “test and commit” or retry
 - Compare saved assumptions with the actual state of the world
 - If different, *undo work*, and *start over* with new state
 - If preconditions still hold, commit the results and continue
 - This is where the work becomes visible to the world (ideally)

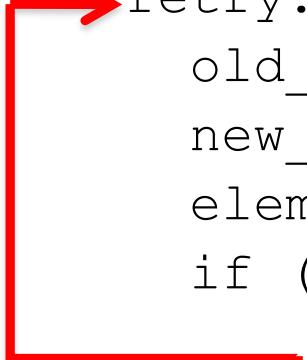
Example – stack pop

```
Pop() {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        if (CAS(old_SP, new_SP, &SP) == FAIL)  
            goto retry;  
    return elem;  
}
```

Example – stack pop

```
Pop() {  
    old_SP = SP;  
    new_SP = old_SP + 1;  
    elem = *old_SP;  
    if (CAS(old_SP, new_SP, &SP) == FAIL)  
        goto retry;  
    return elem;  
}
```

loop



Example – stack pop

```
Pop() {  
    retry:  
        old_SP = SP; // Global - may  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        if (CAS(old_SP, new_SP, &SP) == FAIL)  
            goto retry;  
    return elem;  
}
```

Locals -
won't change!

Global - may
change any time!

"Atomic"
read-modify-write
instruction

CAS

- CAS – single word Compare and Swap
 - An atomic read-modify-write instruction
 - Semantics of the single atomic instruction are:

```
CAS(copy, update, mem_addr)
{
    if (*mem_addr == copy) {
        *mem_addr = update;
        return SUCCESS;
    } else
        return FAIL;
}
```

Example – stack pop

```
Pop() {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        if (CAS(old_SP, new_SP, &SP) == FAIL)  
            goto retry;  
    return elem;  
}
```

Copy global to local → old_SP = SP;

Example – stack pop

```
Pop() {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        if (CAS(old_SP, new_SP, &SP) == FAIL)  
            goto retry;  
        return elem;  
}
```

Do Work →

Example – stack pop

```
Pop() {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP + 1;  
        elem = *old_SP;  
        Test → if (CAS(old_SP, new_SP, &SP) == FAIL)  
                goto retry;  
        return elem;  
}
```

Example – stack pop

```
Pop() {  
    retry:  
    Copy → old_SP = SP;  
    Do Work → new_SP = old_SP + 1;  
    elem = *old_SP;  
    Test → if (CAS(old_SP, new_SP, &SP) == FAIL)  
           goto retry;  
    return elem;  
}
```

What made it work?

- It works because we can atomically commit the new stack pointer value and compare the old stack pointer with the one at commit time
- This allows us to verify no other thread has accessed the stack concurrently with our operation
 - i.e. since we took the copy
 - Well, at least we know the address in the stack pointer is the same as it was when we started
 - Does this guarantee there was no concurrent activity?
 - Does it matter?
 - We have to be careful !

Stack push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

Stack push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1; Copy  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

Stack push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;             
        if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

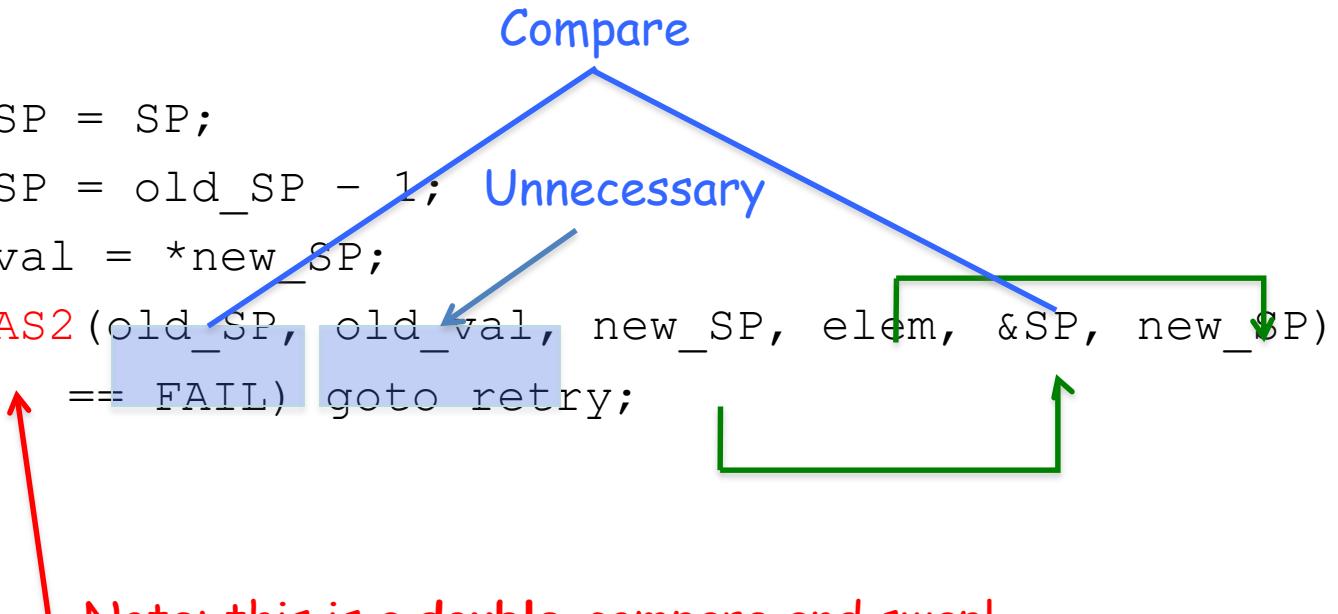
Stack push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if(CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```

Test and
commit

Stack push

```
Push(elem) {  
    retry:  
        old_SP = SP;  
        new_SP = old_SP - 1;  
        old_val = *new_SP;  
        if (CAS2(old_SP, old_val, new_SP, elem, &SP, new_SP)  
            == FAIL) goto retry;  
}
```



Note: this is a **double** compare and swap!
Its needed to atomically update both the
new item and the new stack pointer

Thanks

Next class

Concurrency bugs, race detection

Bug Survey

Yubin Xia

Some slides adjusted from Frans Kaashoek (MIT), Shan Lu (U. Wisc) and Ding Yuan (U. Toronto)

Outline

Recap

Understanding Concurrency bugs

OS Bugs

Data race detection

Recap: How to do with reader – write locks?

```
GetEvents(date) {  
    readerStart(mapRWLock);  
    List events = EventMap.find(date).copy();  
    readerFinish(mapRWLock);  
    return events;  
}  
  
AddEvent(data, newEvent) {  
    writerStart(mapRWLock);  
    EventMap.find(date) += newEvent;  
    writerFinish(mapRWLock);  
}
```

Recap: Basic Implementation

```
readerStart() {  
    lock(lockRW);  
  
    while(numWriters > 0){  
        wait(lockRW,condRW);  
    };  
  
    numReaders++;  
  
    unlock(lockRW);  
}  
  
readerFinish() {  
    lock(lockRW);  
  
    numReaders--;  
  
    broadcast(lockRW,condRW);  
  
    unlock(lockRW);  
}
```

Recap: Basic Implementation

```
writerStart() {  
    lock(lockRW);  
  
    while(numReaders > 0 ||  
          numWriters > 0){  
        wait(lockRW,condRW);  
    };  
  
    numWriters++;  
  
    unlock(lockRW);  
}  
  
writerFinish() {  
    lock(lockRW);  
  
    numWriters--;  
  
    broadcast(lockRW,condRW);  
  
    unlock(lockRW);  
}
```

Recap: Better Implementation

```
readerStart() {  
    lock(lockRW);  
  
    while(numWriters > 0){  
        wait(lockRW,condRW);  
    };  
  
    numReaders++;  
  
    unlock(lockRW);  
}
```

```
readerFinish() {  
    lock(lockRW);  
  
    numReaders--.  
  
    if(numReaders == 0){  
        signal(lockRW,condWR);  
    };  
  
    unlock(lockRW);  
}
```

CONCURRENCY BUG CHARACTERISTICS

Survey

105 **real-world** concurrency bugs from 4 large open source programs

Study from 4 dimensions

Bug patterns

Manifestation condition

Diagnosing strategy

Fixing methods

Implications for:



Bug detection
Software testing
PL design

Outline

Methodology

Findings and implications

- Bug pattern

- Bug manifestation

- Bug fixing

Conclusions

Application sources

	MySQL	Apache	Mozilla	OpenOffice
Software Type	Server	Server	Client	GUI
Language	C++/C	Mainly C	C++	C++
LOC (M line)	2	0.3	4	6
Bug DB history	6 years	7 years	10 years	8 years

**Different types of
real world applications**

Bug sources

	MySQL	Apache	Mozilla	OpenOffice	Total
Non-deadlock	14	13	41	6	74
Deadlock	9	4	16	2	31

Limitations

- No scientific computing applications
- No JAVA programs
- Never-enough bug samples

Non-Deadlock Bug Pattern

Classified based on root causes

Categories:

Atomicity violation

Desired atomicity of certain code region is violated

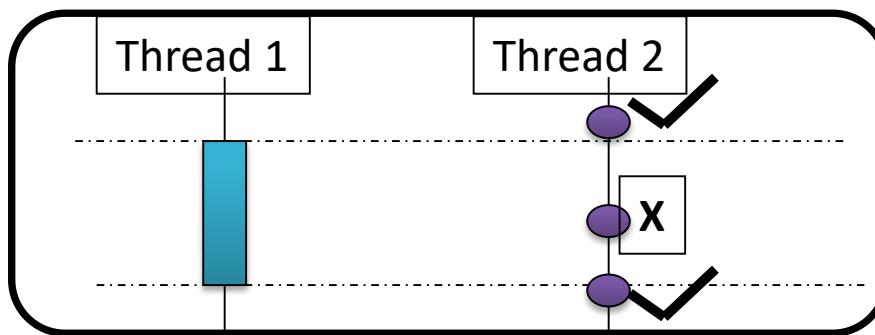
Order violation

The desired order between two (sets of) accesses is flipped

Others

Atomicity violation

The desired atomicity of certain code region is violated



Example of atomicity violation

Thread 1

Thread 2

```
if (thd->proc_info) {  
    ...  
    thd->proc_info = NULL;  
    fputs(thd->proc_info, ...)  
    ...  
}
```

MySQL ha_innodb.hpp

Example of atomicity violation

Thread 1

should be atomically executed

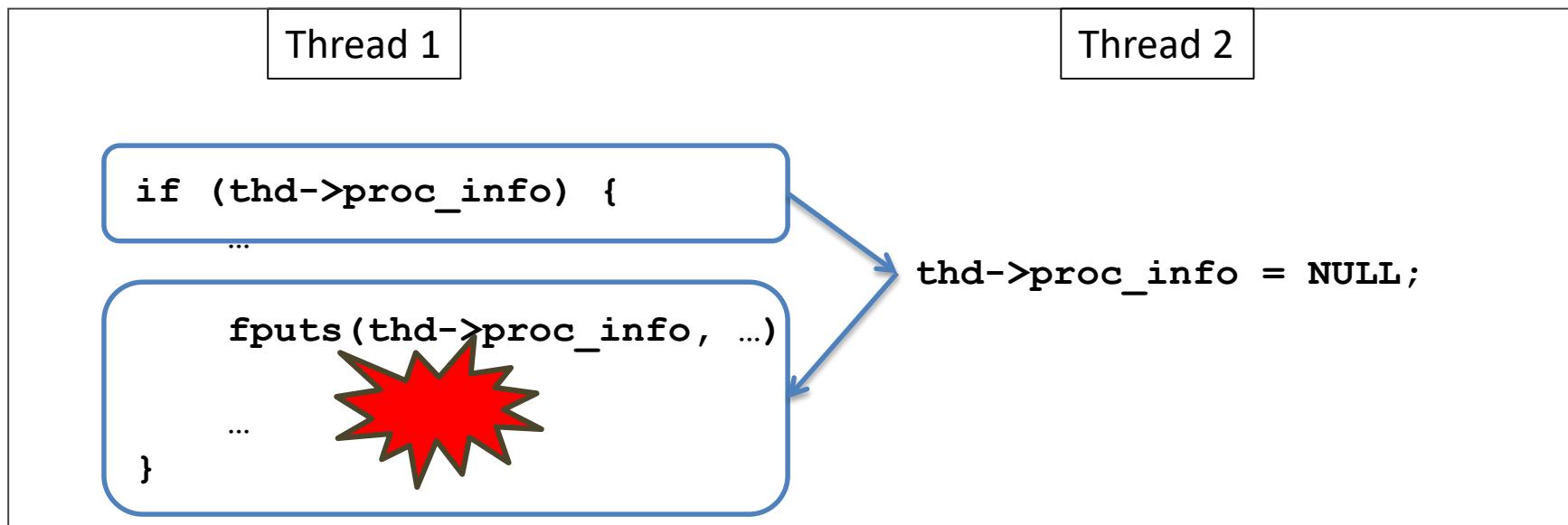
```
if (thd->proc_info) {  
    ...  
  
    fputs(thd->proc_info, ...)  
  
    ...  
}
```

Thread 2

```
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp

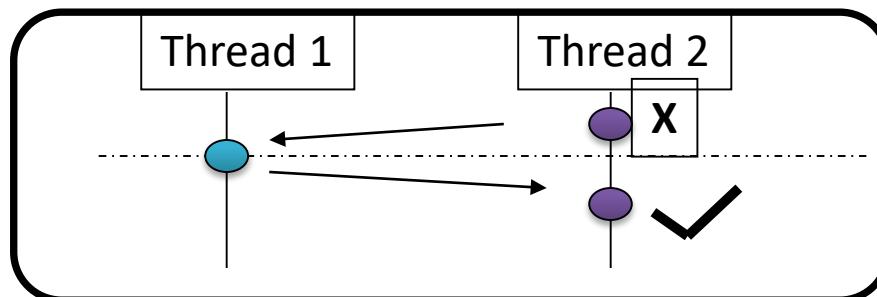
Example of atomicity violation



MySQL ha_innodb.hpp

Order violation

- The desired order between two (sets of) accesses is flipped



Example of order violation

Thread 1

```
void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    ...
}
```

Thread 2

```
void MgmtSrvr::status(...)
{
    *myid =
        node.m_state.dynamicid;
    ...
}
```

MySQL NodeState.hpp

Example of order violation

Thread 1

```
void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    ...
}
```

correct order

Thread 2

```
void MgmtSrvr::status(...)
{
    *myid =
        node.m_state.dynamicid;
    ...
}
```

MySQL NodeState.hpp

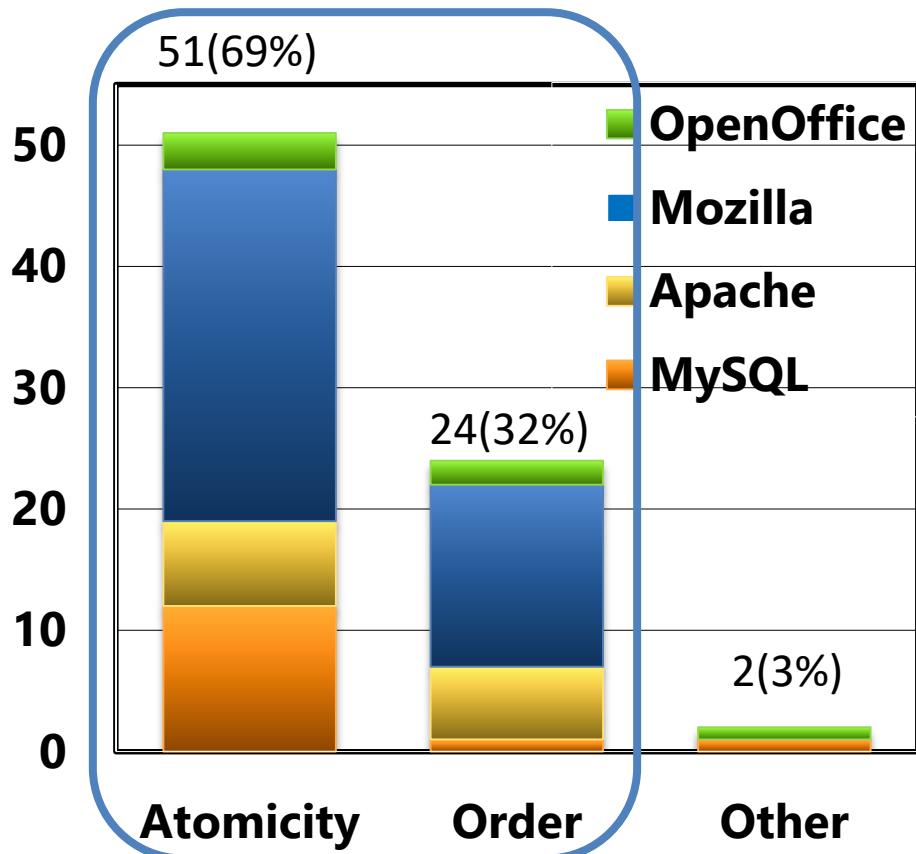
Example of order violation

Thread 1	Thread 2
<pre>void NodeState::setDynamicId(int id) { // initialized here dynamicid = id; ... }</pre>	<pre>void MgmtSrvr::status(...) { *myid = node.m_state.dynamicid; ... }</pre>

A blue arrow points from the line `dynamicid = id;` in Thread 1 to the line `*myid = node.m_state.dynamicid;` in Thread 2, with the text "buggy order" written below the arrow.

MySQL NodeState.hpp

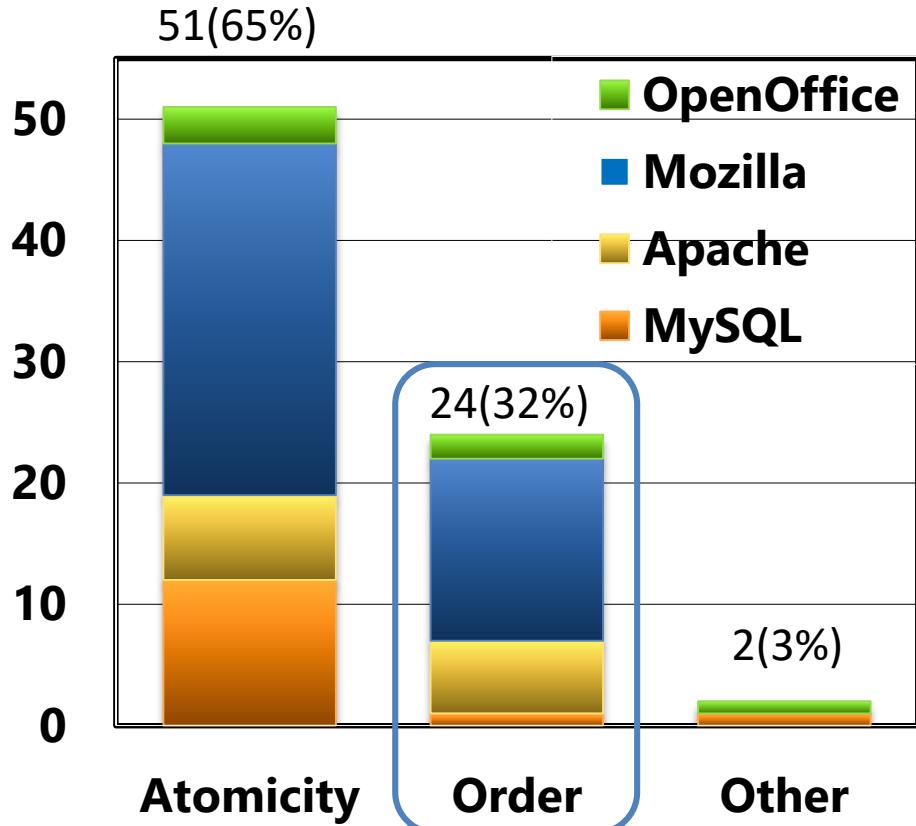
Non-deadlock bug pattern



Implications

We should focus on atomicity violation and order violation

Non-deadlock bug pattern



Implications

We should focus on atomicity violation and order violation

Bug detection tools for order violation bugs are desired

How to trigger a bug

Bug manifestation condition

A specific execution order among a smallest set of memory accesses

The bug is guaranteed to manifest, as long as the condition is satisfied

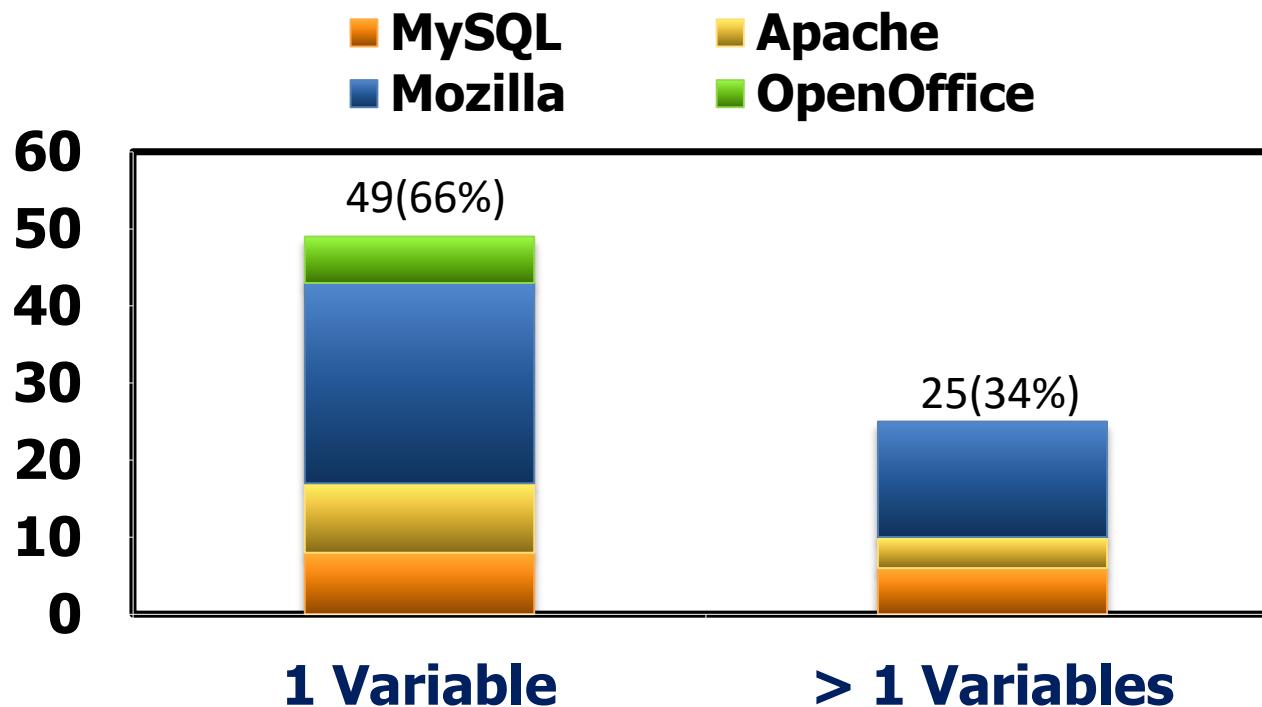
How many threads are involved?

How many variables are involved?

How many accesses are involved?

Single Variable vs. Multiple Variable

Findings



Single Variable vs. Multiple Variable

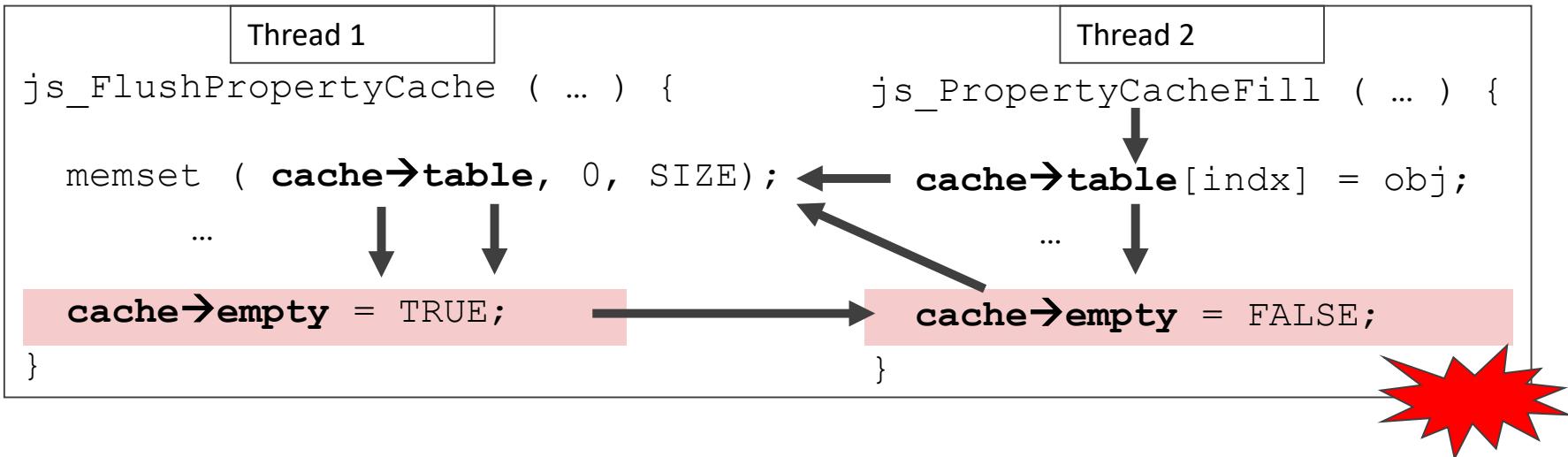
Single variables are more common

The widely-used simplification is reasonable

Multi-variable concurrency bugs are non-negligible

Techniques to detect multi-variable concurrency bugs
are needed

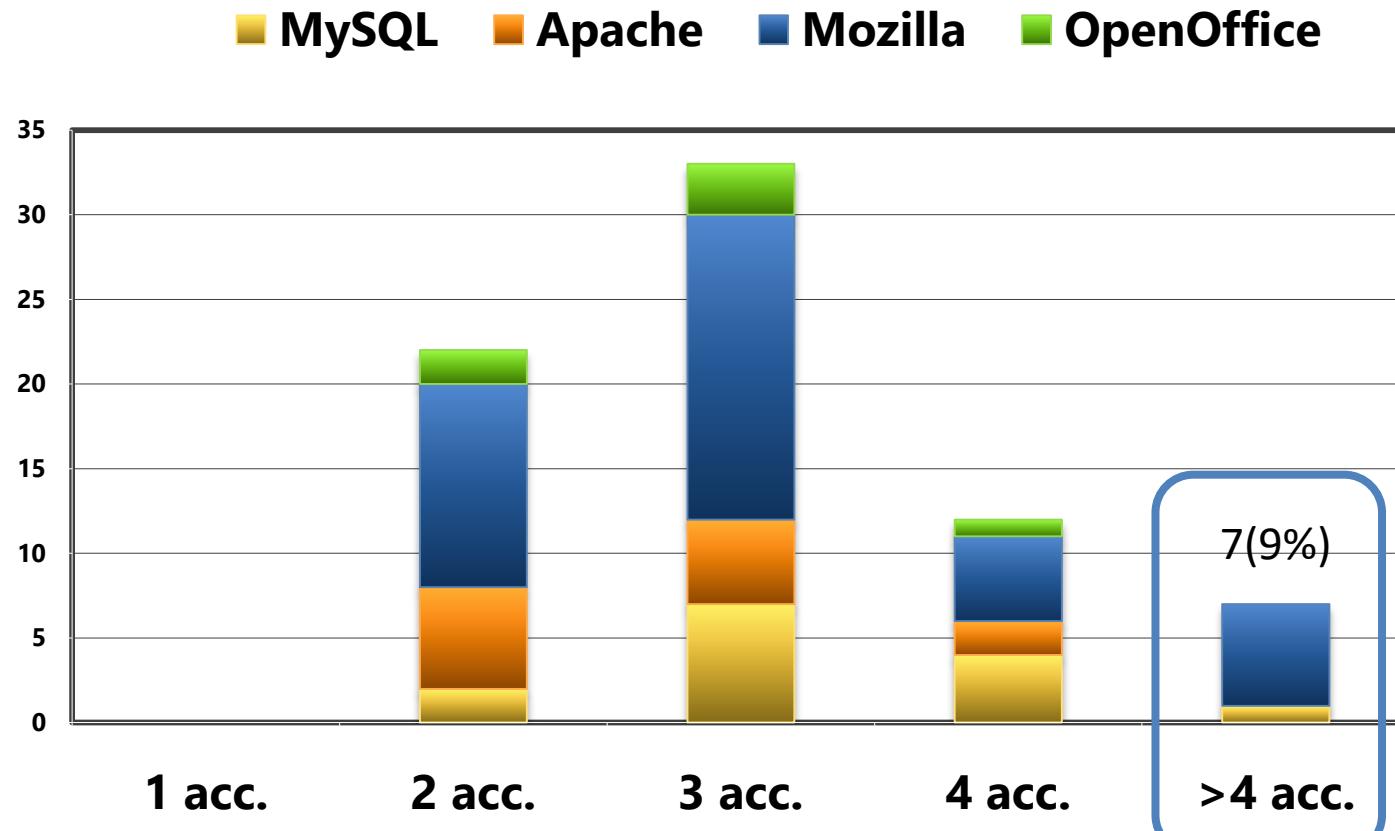
Multi-Variable Concurrency Bug Example



Control the order among accesses to any one variable
can **not** guarantee the bug manifestation

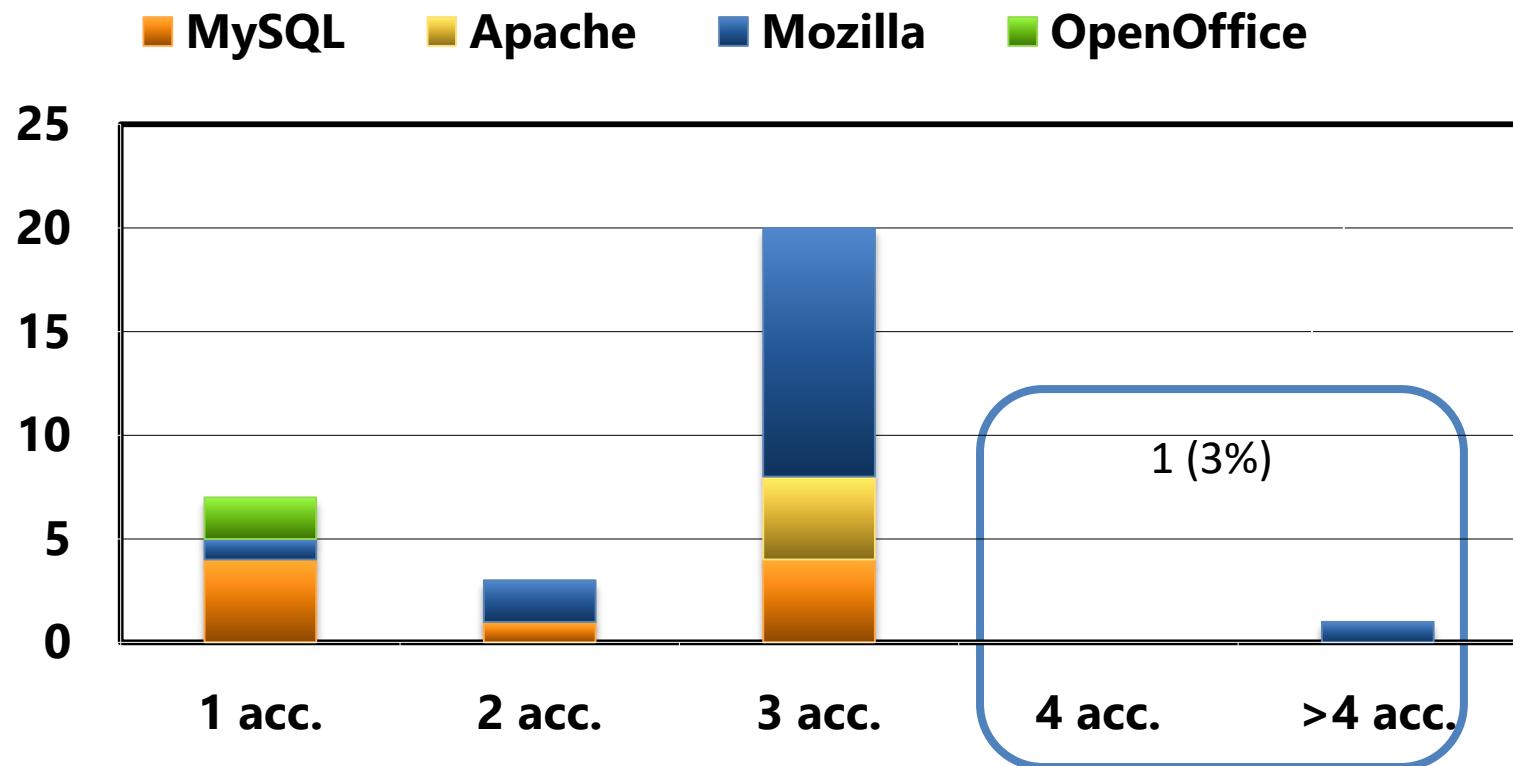
Non-deadlock bugs

Number of Accesses



Deadlock bugs

Number of Accesses



Implications

Only a few percentage of bugs need more than 4 access to trigger

Concurrent program testing can focus on small groups of accesses

The testing target shrinks from exponential to polynomial

Number of Threads Involved

101 out of 105 (96%) bugs involve at most two threads

Most bugs can be reliably disclosed if we check all possible interleaving between each pair of threads

Few bugs cannot

Example: Intensive resource competition among many threads causes unexpected delay

How Were Non-Deadlock Bugs Fixed?

Adding/changing locks 20 (27%)

Condition check 19 (26%)

Data-structure change 19 (26%)

Code switch 10 (13%)

Other 6 (8%)

Implications

No silver bullet for fixing concurrency bugs.

Lock usage information is not enough to fix bugs.

How Were Deadlock Bugs Fixed?

		Might introduce non-dead locks
Give up resource acquisition	19 (61%)	
Change resource acquisition order	7 (23%)	
Split the resource to smaller ones	1 (3%)	
Others	4 (13%)	

We need to pay attention to the correctness of “fixed” deadlock bugs

Other findings

Impact of concurrency bugs

- ~ 70% leads to program crash or hang

Reproducing bugs are critical to diagnosis

Programmers lack diagnosis tools

- Most are diagnosed via code review

- Reproduce bugs are extremely hard and directly determines the diagnosing time

60% 1st-time patches contain concurrency bugs
(old or new)

Summary

Bug detection needs to look at order-violation bugs and multi-variable concurrency bugs

Testing can target at more realistic interleaving coverage goals

Fixing concurrency bugs is not trivial and not easy to get right

Support from automated tools is needed

BUGS IN EXCEPTION HANDLERS

Study Methodology

- ▶ Randomly sampled 198 user-reported failures*
 - ▶ Carefully studied the discussion and related code/patch
 - ▶ Reproduced 73 to understand them
- ▶ 48 are **catastrophic** --- they affect all or a majority of users

Software	Program language	Sampled failures	
		Total	Catastrophic
Cassandra	Java	40	2
HBase	Java	41	21
HDFS	Java	41	9
Hadoop MapReduce	Java	38	8
Redis	C	38	8
Total	-	198	48

multiple events are required

User: “Sudden outage on the entire HBase cluster.”

Event 1: Load balance: transfer Region R from slave A to B

↓
Slave B opens R

Event 2: Slave B dies

↓
R is assigned to slave C

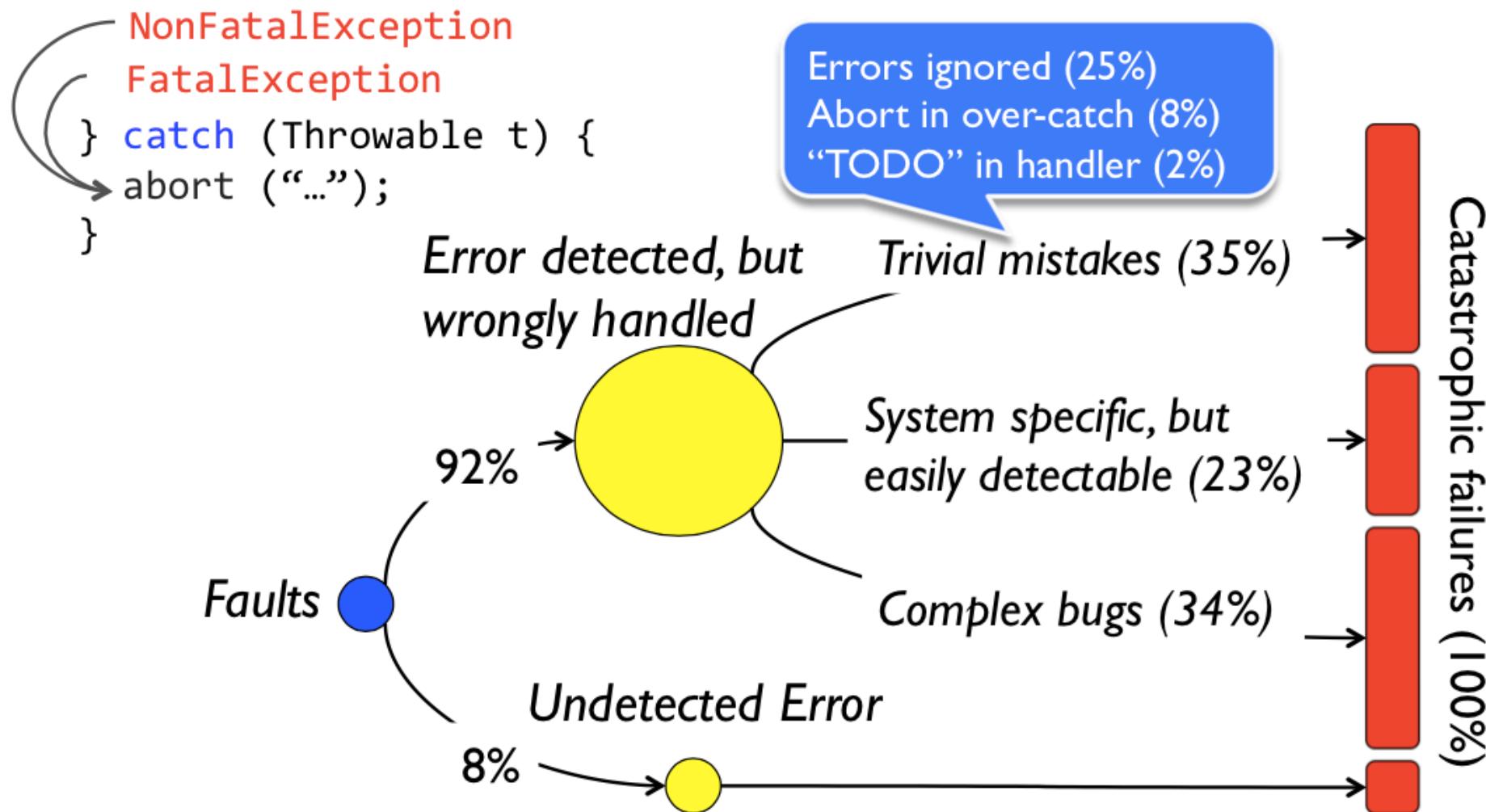
↓
Slave C opens R

```
/* Master: delete the
 * ZooKeeper znode after
 * the region is opened */
try {
    deleteZNode();
} catch (KeeperException e) {
    cluster.abort("...");
}
```

Not handled
properly

Breakdown of catastrophic failures

92% of catastrophic failures are the result of incorrect error handling



A failure caused by trivial mistake

User:

“MapReduce jobs hang when a rare Resource Manager restart occurs. I have to ssh to every one of our 4000 nodes in a cluster and kill all jobs.”

```
catch (RebootException) {  
    // TODO  
    LOG("Error event from RM: shutting down...");  
+    eventHandler.handle(exception_response);  
}
```

Why do developers ignore error handling?

- ▶ Developers think the errors *will never happen*
 - ▶ Code evolution may enable the errors
 - ▶ The judgment can be wrong
 - } `catch (IOException e) {`
// will never happen
{}
- ▶ Error handling is difficult
 - ▶ Errors can be returned by 3rd party libraries
 - } `catch (NoTransitionException e) {`
/ Why this can happen? Ask God not me. */*
{}}
- ▶ Feature development is prioritized

Other Findings

- ▶ Failures require no more than 3 nodes to manifest
 - * almost all (98%)
- ▶ Failures can be reproduced offline by unit tests
 - ▶ The triggering events are recorded in system log
 - * Logs are noisy: the median of the number of log messages printed by each failure is 824.
- ▶ Non-deterministic failures can still be deterministically reproduced
 - * at least one part of the timing dependency can be controlled by testers

Unexpected fun: comments in error handlers

```
/* If this happens, hell will unleash on earth. */  
  
/* FIXME: this is a buggy logic, check with alex. */  
  
/* TODO: this whole thing is extremely brittle. */  
  
/* TODO: are we sure this is OK? */  
  
/* I really thing we should do a better handling of these  
 * exceptions. I really do. */  
  
/* I hate there was no piece of comment for code  
 * handling race condition.  
 * God knew what race condition the code dealt with! */
```

OS BUGS

Bugs Cost??

Patriot missile defense system

28 **dead** soldiers, 98 wounded

Therac-25 medical device

Several people dead, others wounded

General Electric XA/21

50 million people left without water, electricity.

How to find bugs?

What is your belief set?

MUST set

MAY set

What is the implied sets?

Inconsistency means possible bugs!!

Trivial consistency: NULL pointers

* p implies a MUST belief:

p is *not* null

A check ($p == \text{NULL}$) implies two MUST beliefs:

POST: p is null on true path, not null on false path

PRE: p was unknown before check

```
/* 2.4.1: drivers/isdn/svmb1/capidrv.c */
if (!card)
    printk(KERN_ERR, "capidrv-%d: ...", card->contrnr...)
```

```
/* drivers/net/wan/sdla_chdlc.c:3948 */
if (!card) {
    lock_adapter_irq(&card->wandev.lock, &smp_flags);
    card->tty=NULL;
```

Null pointer fun

Use-then-check

```
/* 2.4.7: drivers/char/mxser.c */
struct mxser_struct *info = tty->driver_data;
unsigned flags;
if(!tty || !info->xmit_buf)
    return 0;
```

Contradiction/redundant checks

```
/* 2.4.7/drivers/video/tdfxfb.c */
fb_info.regbase_virt = ioremap_nocache(...);
if(!fb_info.regbase_virt)
    return -ENXIO;
fb_info.bufbase_virt = ioremap_nocache(...);

if(!fb_info.regbase_virt) {
    iounmap(fb_info.regbase_virt);
```

Internal Consistency: finding security holes

Applications are bad:

Rule: “do not dereference user pointer `<p>`”

One violation = security hole

Big Problem: which are the user pointers???

Sol’n: for-all pointers, cross-check two OS beliefs

“`*p`” implies safe kernel pointer

“`copyin(p)/copyout(p)`” implies dangerous user pointer

Error: pointer `p` has both beliefs.

Statistical: Deriving deallocation routines

Use-after free errors are horrible.

Problem: lots of undocumented sub-system free functions

Soln: derive behaviorally: pointer “p” not used after call
“foo(p)” implies **MAY** belief that “foo” is a free function

Conceptually: Assume all functions free all arguments
(in reality: filter functions that have suggestive names)

A bad free error

```
/* drivers/block/cciss.c:cciss_ioctl */  
if (ioccommand.Direction == XFER_WRITE) {  
    if (copy_to_user(...)) {  
        cmd_free(NULL, c);  
        if (buff != NULL) kfree(buff);  
        return (-EFAULT);  
    }  
}  
if (ioccommand.Direction == XFER_READ) {  
    if (copy_to_user(...)) {  
        cmd_free(NULL, c);  
        kfree(buff);  
    }  
}  
cmd_free(NULL, c);  
if (buff != NULL) kfree(buff);
```

“A must be followed by B”

“a(); ... b();” implies MAY belief that a() follows b()

You might believe a-b paired, or might be a coincidence

Checking derived lock functions

Simplest:

```
/* fs/proc/inode.c:41:de_put: */
lock_kernel();
if (!de->count) {
    printk("de_put: entry already free!\n");
    return;
}
unlock_kernel();
```

Evilest:

```
/* 2.4.1: drivers/sound/trident.c:trident_release:
lock_kernel();
card = state->card;
dmabuf = &state->dmabuf;
VALIDATE_STATE(state);

#define VALIDATE_MAGIC(FOO,MAG)
({
    if (!(FOO) || (FOO)->magic != MAG) {
        printk(invalid magic, _FUNCTION_);
        return -ENXIO;
    }
})

#define VALIDATE_STATE(a) VALIDATE_MAGIC(a,TRIDENT STATE MAGIC)
```

CP-Miner

- Many bugs are generated by copy-paste
- Find the pattern in the source
- OSDI'04!

```
( linux-2.6.6/arch/sparc64/prom/memory.c )
68 void __init prom_meminit(void)
69 {
    .....
92     for(iter=0; iter<num_regs; iter++) {
93         prom_phys_total[iter].start_adr =
94             prom_reg_memlist[iter].phys_addr;
95         prom_phys_total[iter].num_bytes =
96             prom_reg_memlist[iter].reg_size;
97         prom_phys_total[iter].theres_more =
98             &prom_phys_total[iter+1];
99     }
    .....
111    for(iter=0; iter<num_regs; iter++) {
112        prom_prom_taken[iter].start_adr =
113            prom_reg_memlist[iter].phys_addr;
114        prom_prom_taken[iter].num_bytes =
115            prom_reg_memlist[iter].reg_size;
```

iter].theres_more =
tal[iter+1]; // bug

Software	errors reported	bugs verified	careless programming	false alarms		
				(1)	(2)	(3)
Linux	421	28	21	151	41	57
FreeBSD	443	23	8	307	41	30
Apache	17	5	0	3	1	6
PostgreSQL	74	2	0	13	10	43

/* iComment : Bugs or Bad Comments ? */ [SOSP'07]

- Inconsistency between bugs and comments

drivers/scsi/in2000.c:

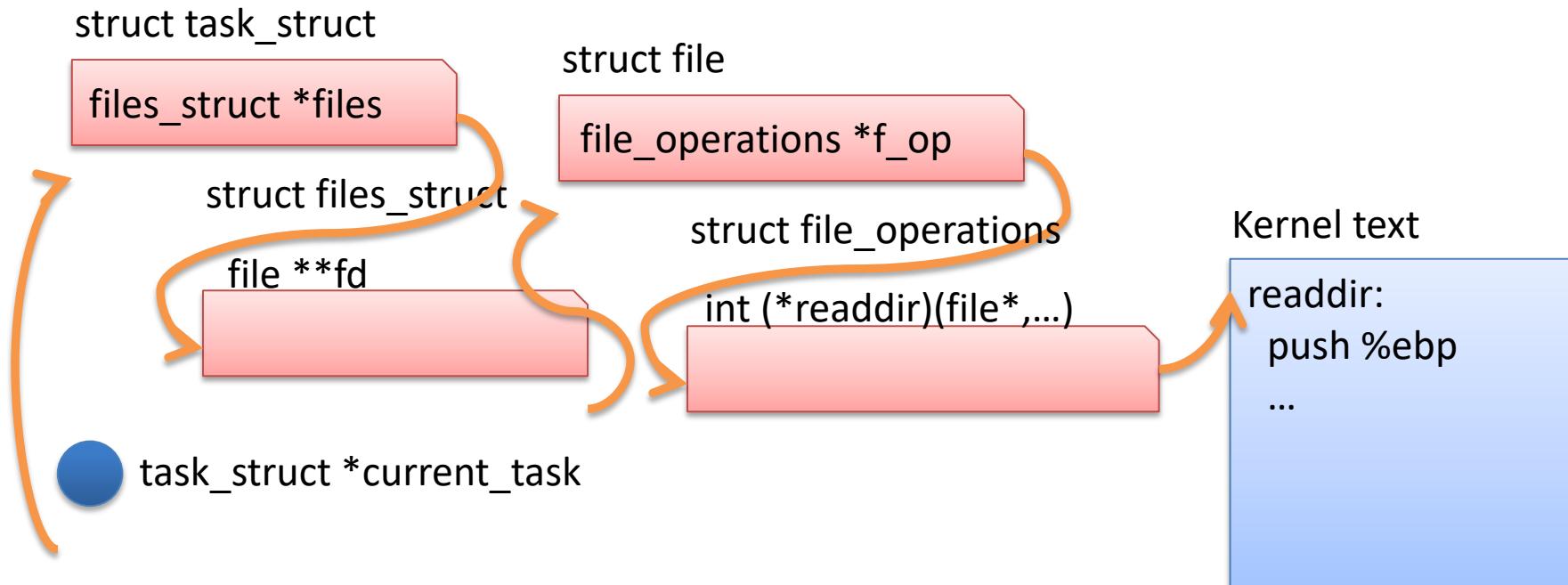
```
security/nss/lib/ssl/sslsnce.c:  
/* Caller must hold cache lock when calling this.*/  
static sslSessionID * ConvertToSID( ... ){ ... }  
...  
static sslSessionID *ServerSessionIDLookup(...){...  
    UnlockSet(cache, set);  
    ...  
    sid = ConvertToSID( ... );  
    ...  
}
```

*Assumption
in Comment.*

*Cache lock is
released
before calling
ConvertToSID()*

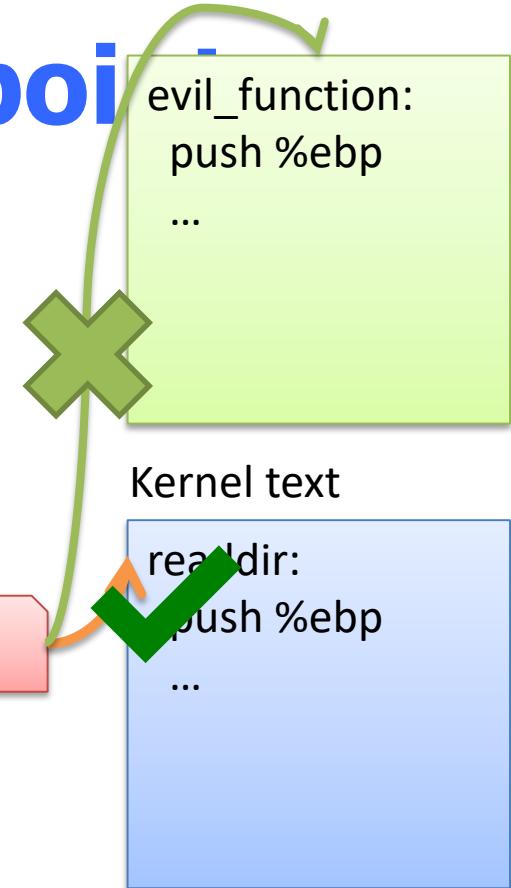
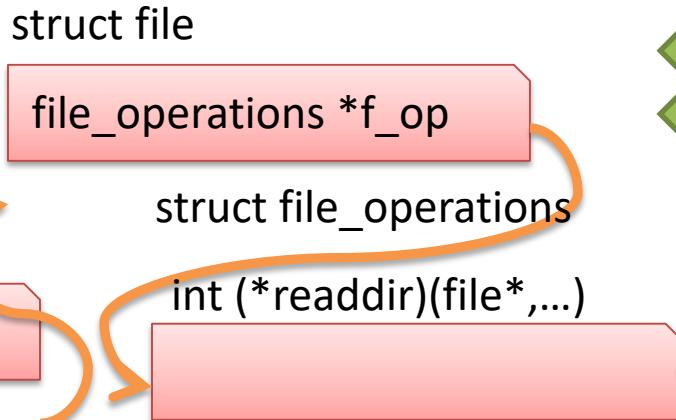
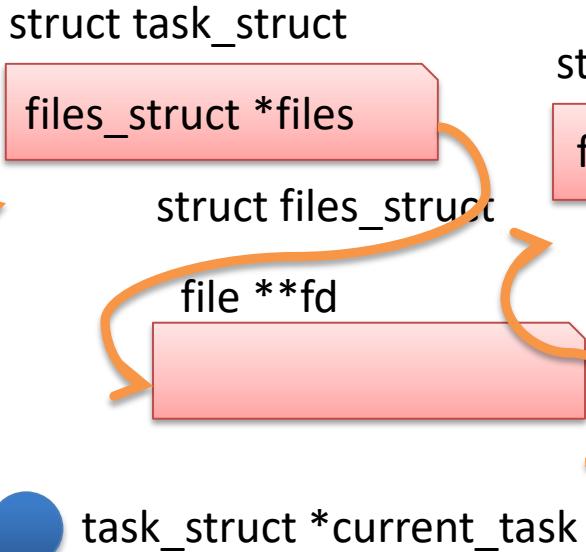
Mismatch!
Confirmed
by developers
as a bad
comment
after we
reported it.

Checking function pointers



- How does kernel get to function pointer?
 - Start at global root (symbol)
 - Traverse graph of data structures

Checking function pointers



- *State-based control flow integrity [Petroni & Hicks]*
 - Start at global root (symbol)
 - Traverse graph of data structures
 - Ensure function pointers point to valid entry points

General Rules

Do **not** call blocking functions with interrupts disabled or spin lock held

Check for **NULL** results

Do **not** allocate large stack variables

Do not **re-use** already-allocated memory

Check **user** pointers before using them in kernel mode

Release acquired locks

Unstated Rules in JOS

Interrupts are **disabled** in kernel mode

Only **env 1** has access to disk

All registers are saved & restored on context switch

Application code is **never** executed with CPL 0

Don't **allocate** an already-allocated physical page

Propagate error messages to user applications
(e.g., out of resources)

Unstated Rules in JOS

A spawned program should have open **only** file descriptors 0, 1, and 2.

User pointers should be run through TRUP before used by the kernel

Data Race & DeadLock

Yubin Xia

Review: Bug Survey

Bug detection needs to look at order-violation bugs and multi-variable concurrency bugs

Testing can target at more realistic interleaving coverage goals

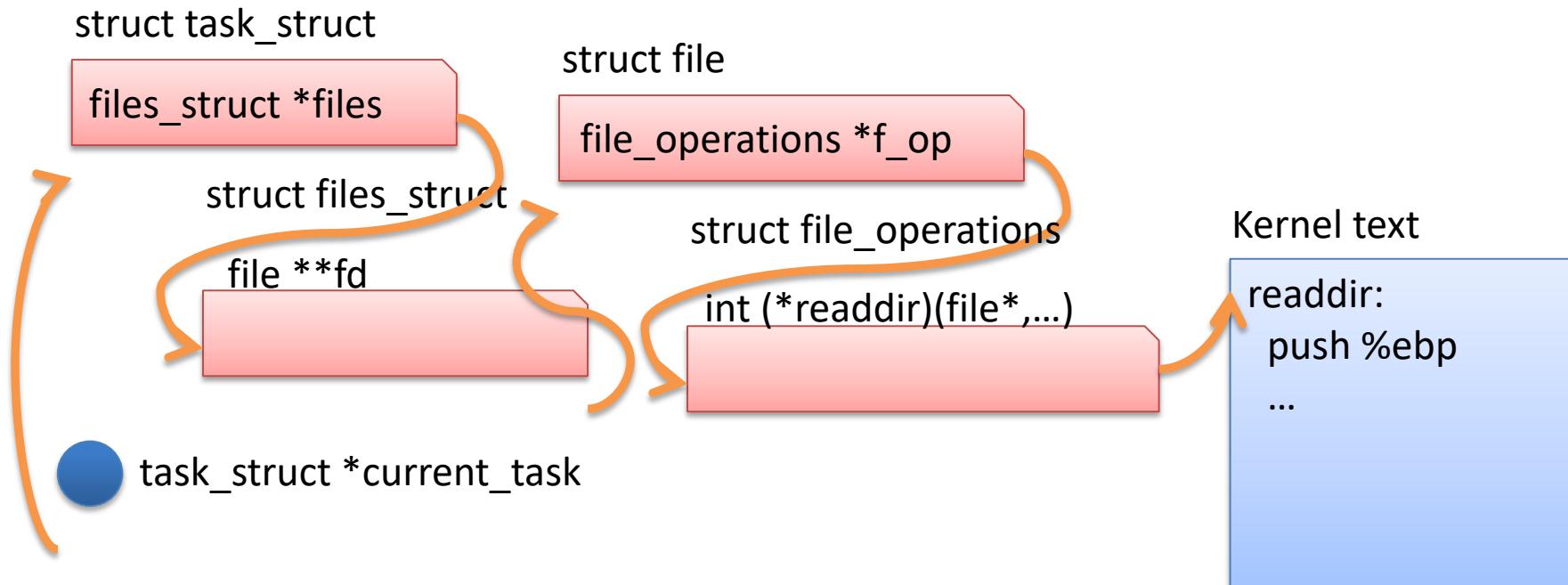
Fixing concurrency bugs is not trivial and not easy to get right

Support from automated tools is needed

Fancy Ways of Finding Bugs

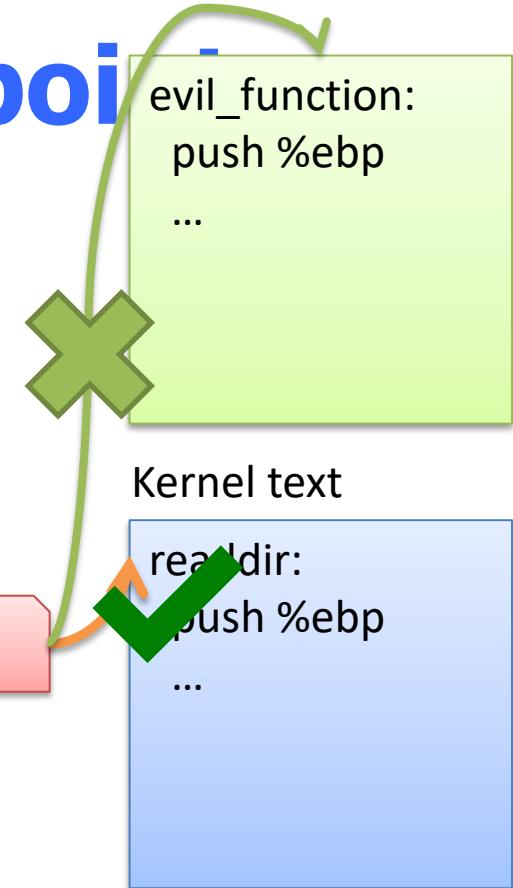
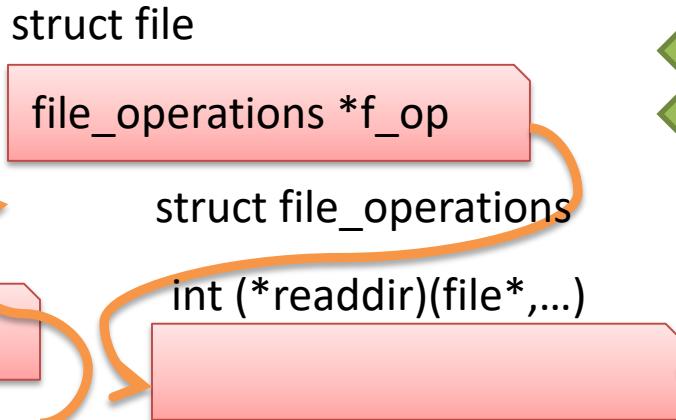
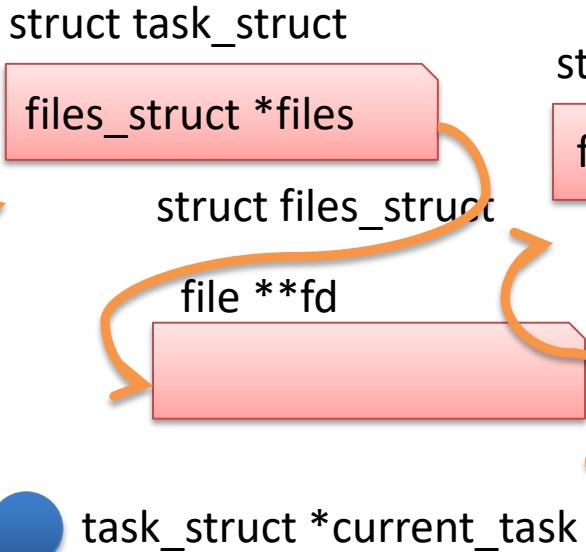
- Error handlers
- C-P pattern miner
- iComment
- Invariable in the kernel

Checking function pointers



- How does kernel get to function pointer?
 - Start at global root (symbol)
 - Traverse graph of data structures

Checking function pointers



- *State-based control flow integrity [Petroni & Hicks]*
 - Start at global root (symbol)
 - Traverse graph of data structures
 - Ensure function pointers point to valid entry points

Outline

Happens-before Race Detection

Lockset-Based Race Detection

Deadlocks

Detecting deadlocks using Lockset Analysis

Data Race Detection

What is Race?



Data Race

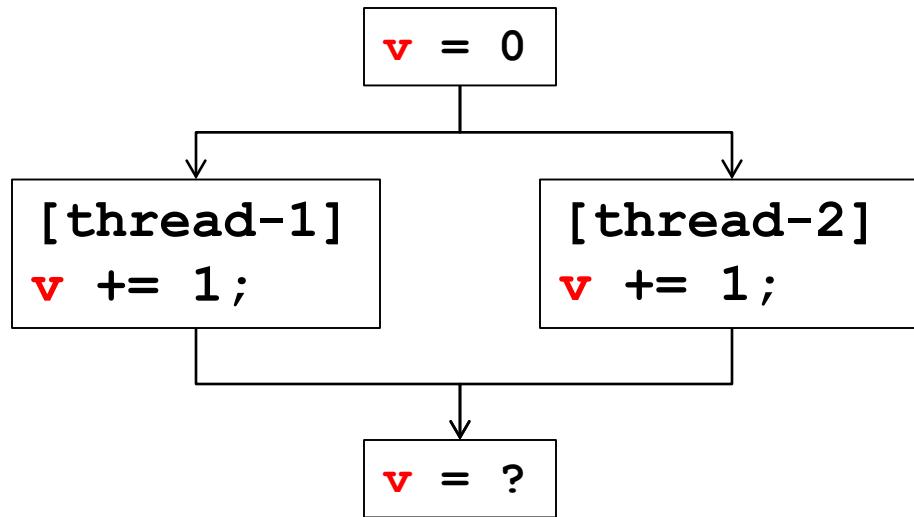
An undesirable situation that occurs when a device or system attempts to perform **two or more** operations at the **same time**, but the operations must be done in the **proper sequence** in order to be done correctly

- multithread
- distributed Programs

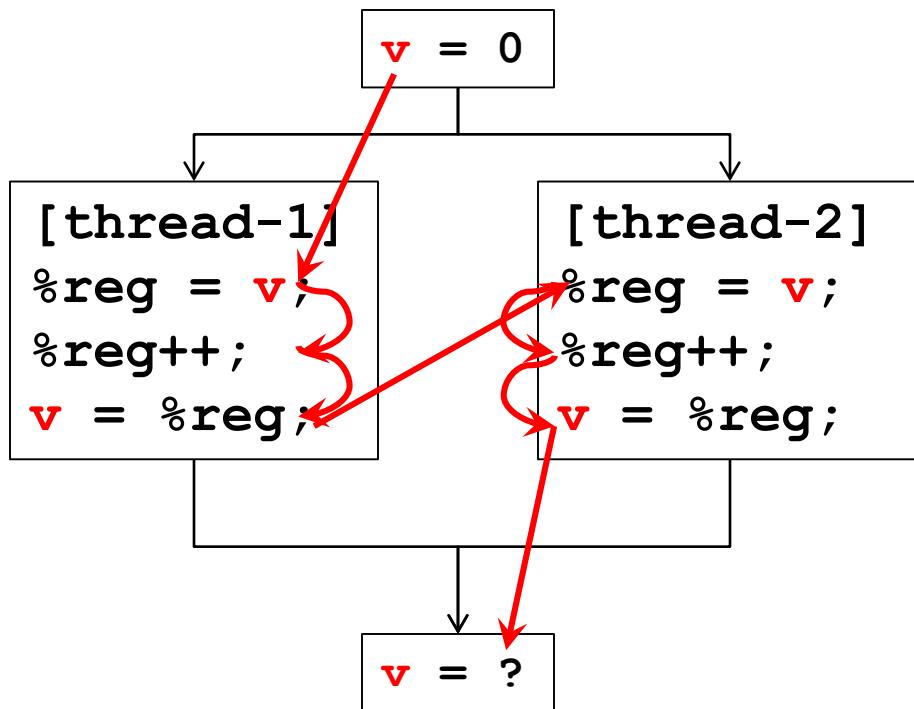
Key reason

- separate processes or threads of execution depends on **same shared state**

Data Race



Data Race



v = 0 (memory)

T1 reads 0 to %reg

T1 increments %reg to 1

T1 stores %reg to v (v = 1)

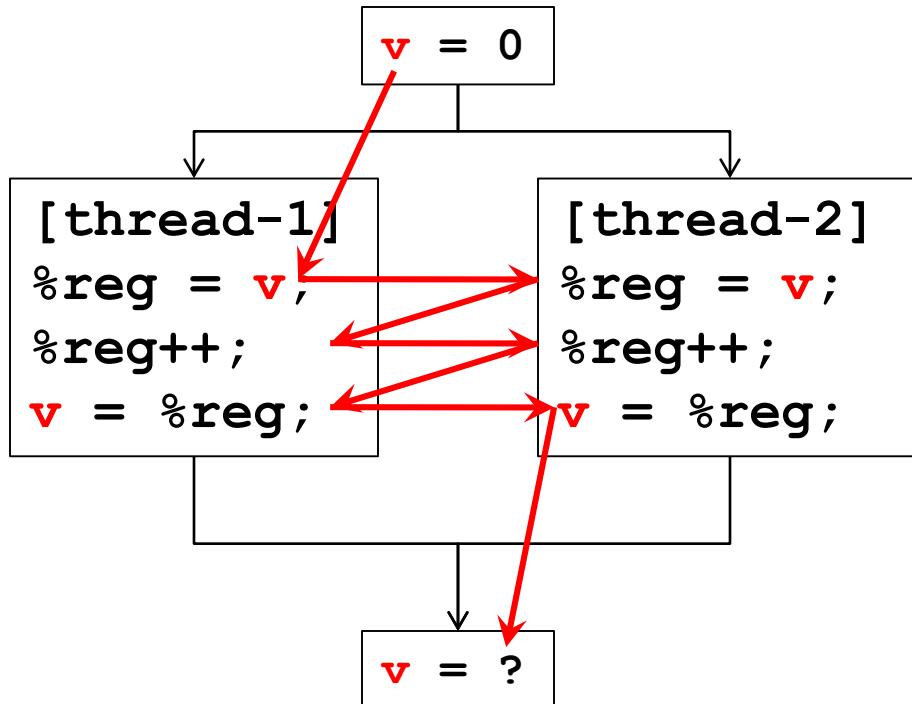
T2 reads 1 to %reg

T2 increments %reg to 2

T2 stores %reg to v (v = 2)

v = 2; (memory)

Data Race



$v = 0$ (memory)

T1 reads 0 to %reg
T2 reads 0 to %reg
T1 increments %reg to 1
T2 increments %reg to 1
T1 stores %reg to v ($v = 1$)
T2 stores %reg to v ($v = 1$)

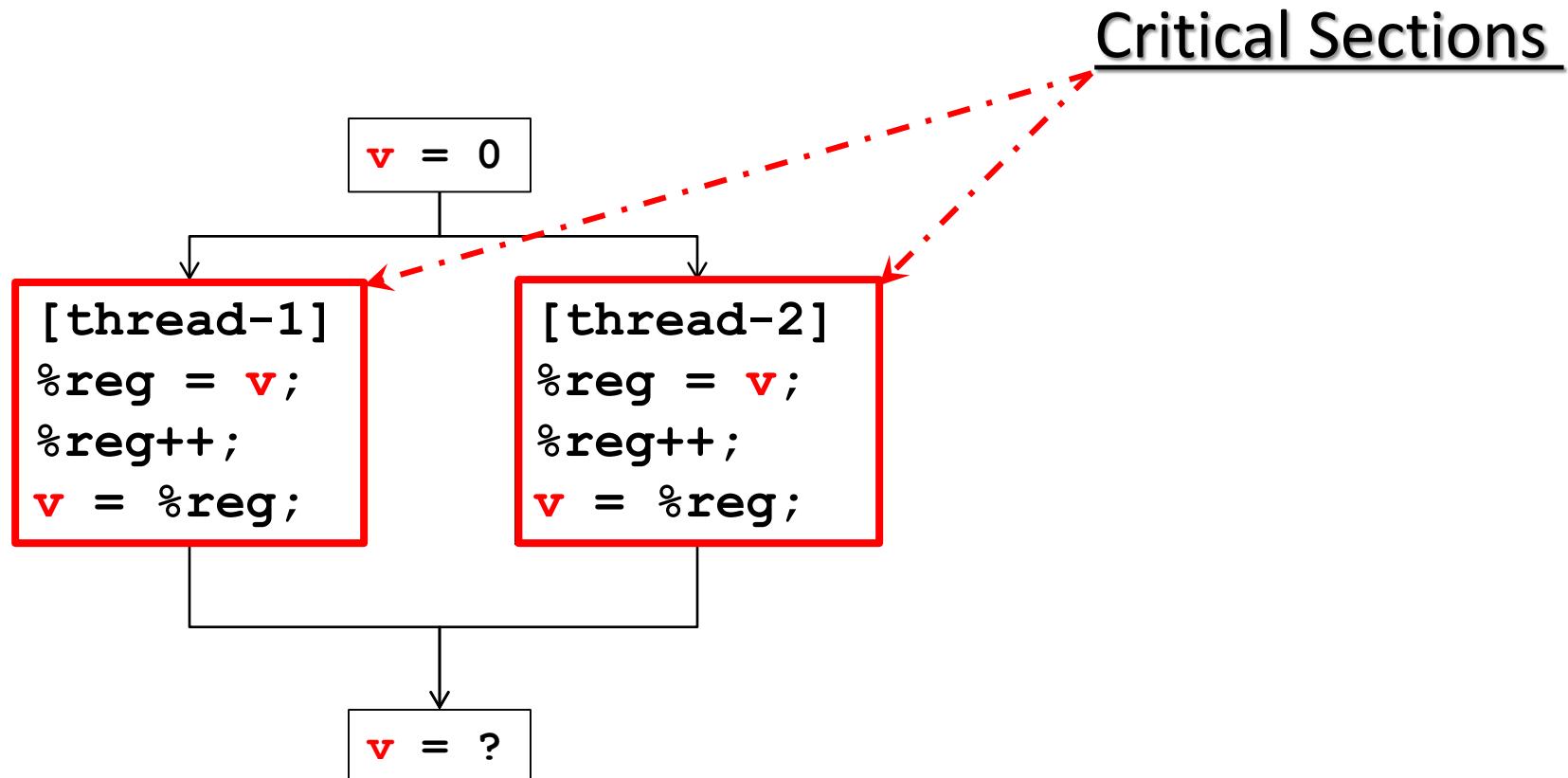
$v = 1$; (memory)

Data Race

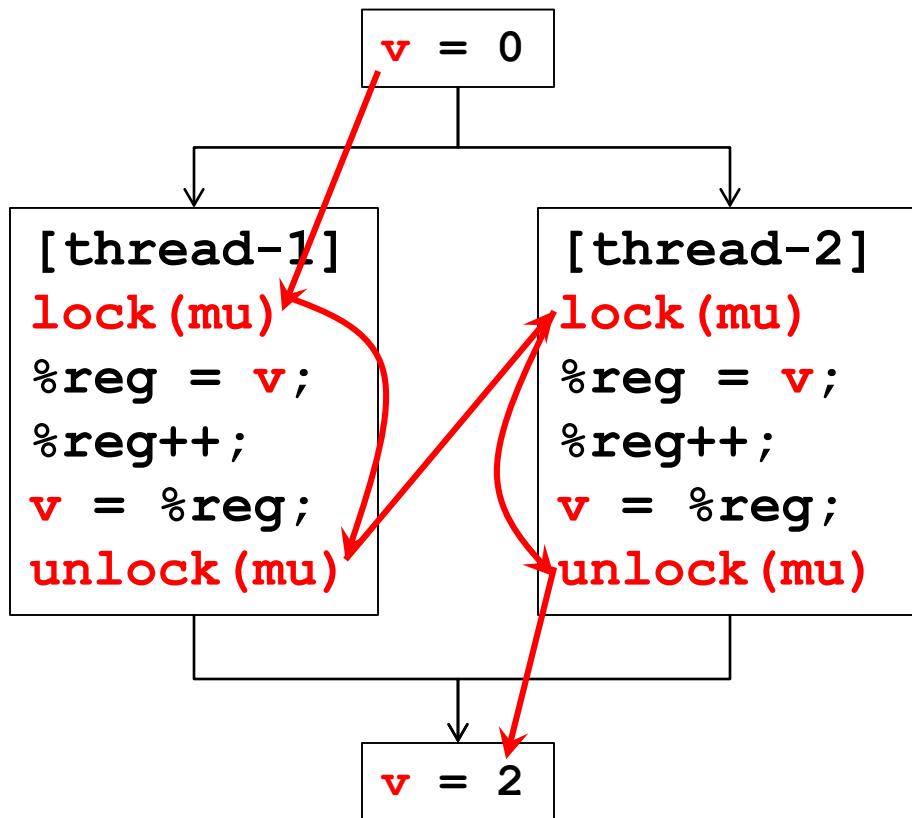
The definition of data race:

1. two concurrent threads access a **shared** variable
2. at least one access is a **write**
3. the threads use no explicit mechanism to prevent the accesses from being **simultaneous**.

Remove Race Condition



Remove Race Condition



Mutually-exclusive

- lock & unlock
- cli & sti

Data Race Bug

Thread-1

```
OpenInputStream()
{
    PostEvent();
    ...
    m_inputStream = ...
    ...
}
```

file: nsSocketTransport.cc

Thread-2

```
ProcessCurrentURL()
{
    WaitEvent();
    ...
    if (m_inputStream) {
        AsyncRead(m_inputStream);
    }
    ...
}
```

file: nsImapProtocol.cpp

*Data race bug in Mozilla

Data Race Detectors

Two major categories:

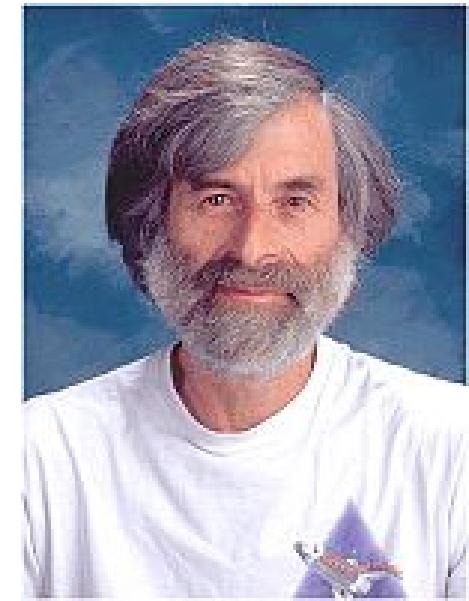
- Happens-before based
- Lockset based

HAPPENS-BEFORE BASED

Happens-before Relation

The **happens-before** relation is a means of **ordering** events based on the causal relationship of pairs of events in a concurrent system

- denoted: →
- Formulated by Leslie Lamport
- strict **partial order** on events
- without using physical clocks



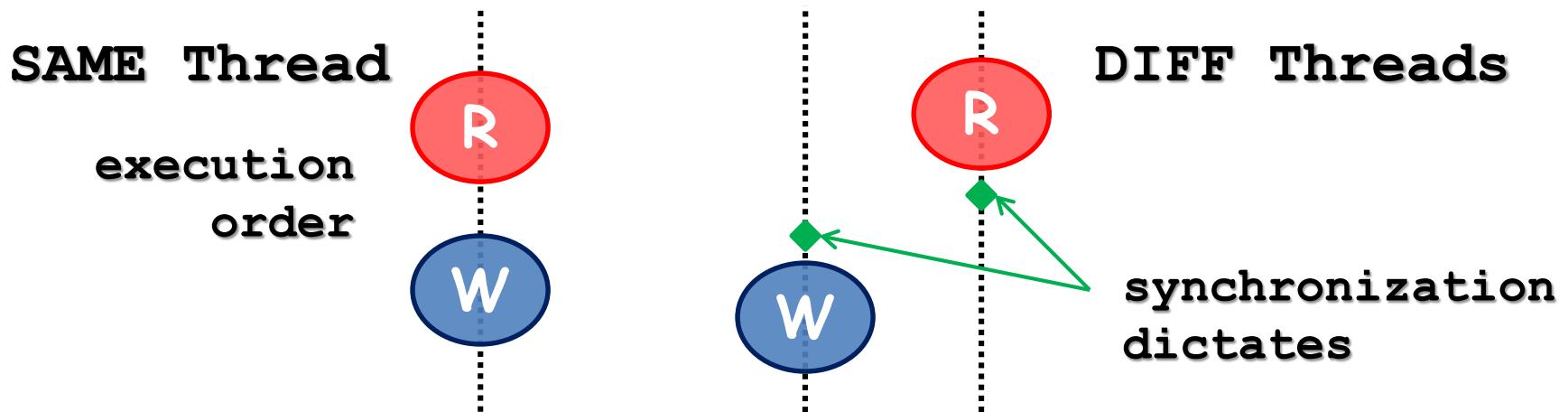
Rules in HB Relation

For the events A and B,

HB1: On the **same** sequential thread,
 $A \rightarrow B$ if A executes before B.

HB2: On the **different** threads,
 $A \rightarrow B$ if there is a synchronization that dictates A precedes B.

HB3: If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$

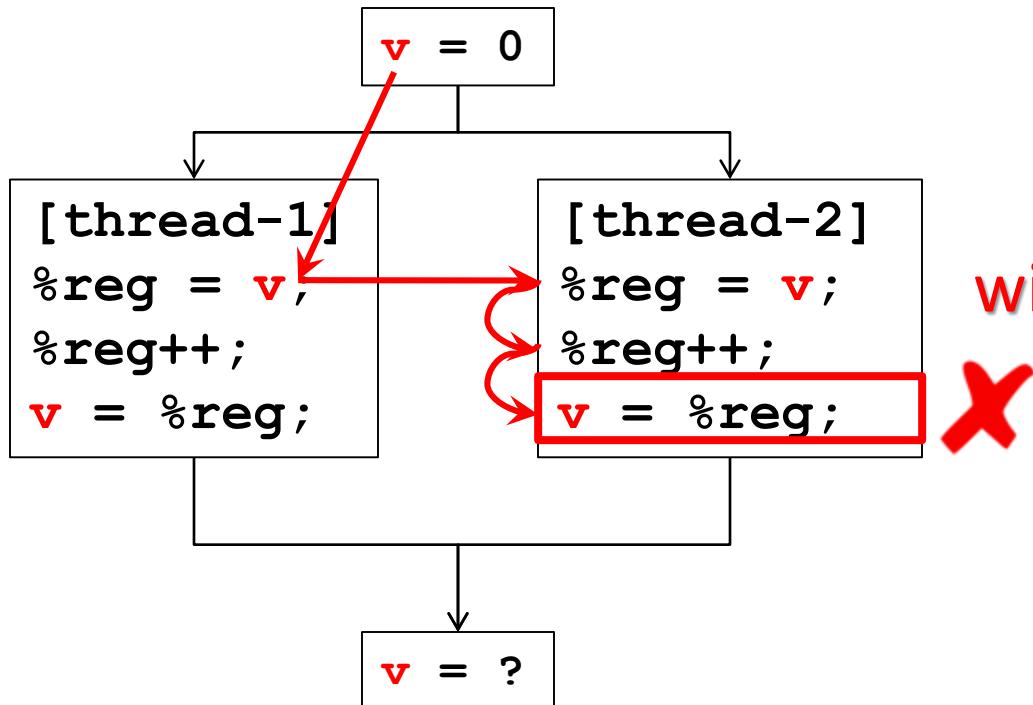


HB based Detectors

The definition of data race:

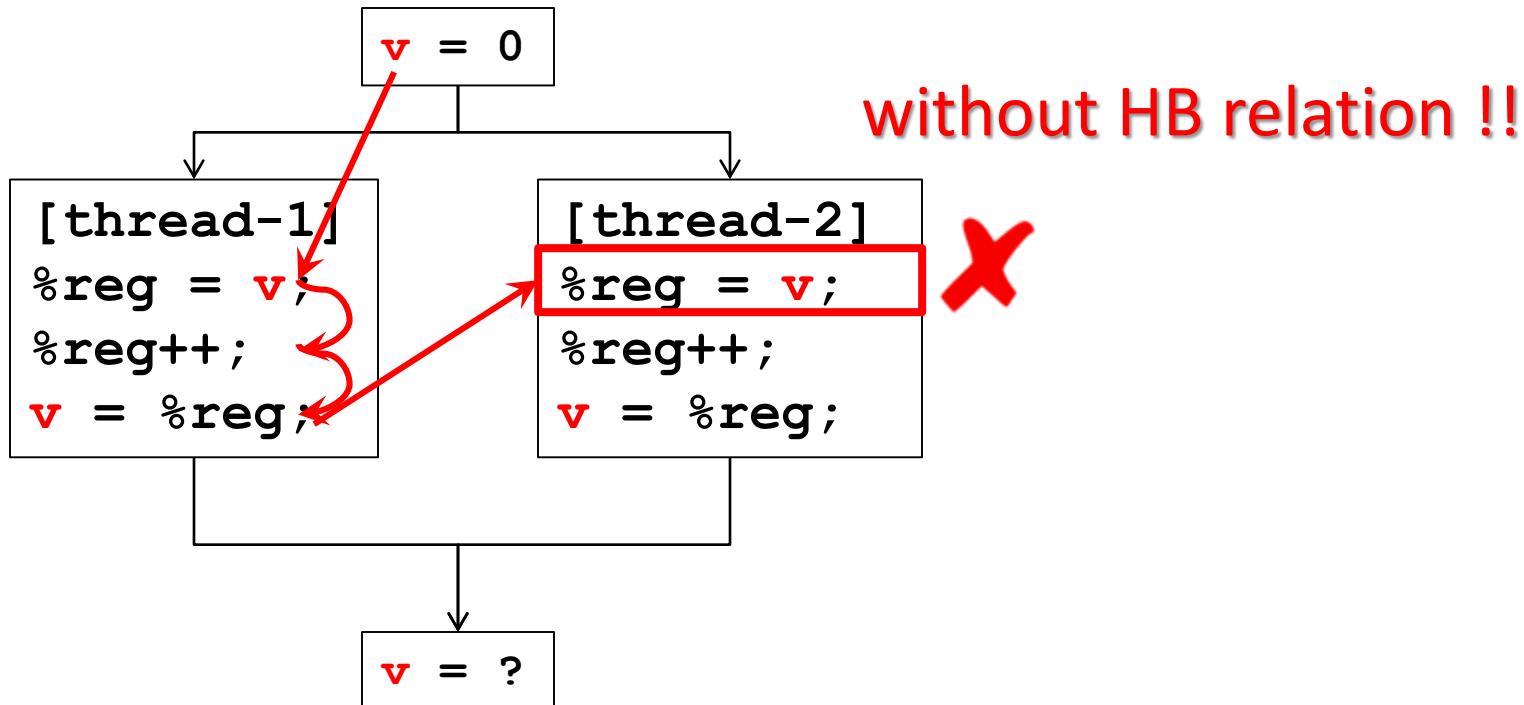
- 1.a pair of accesses to the **same** memory location
- 2.at least one access is a **write**
- 3.neither one **happens-before** the other

Example

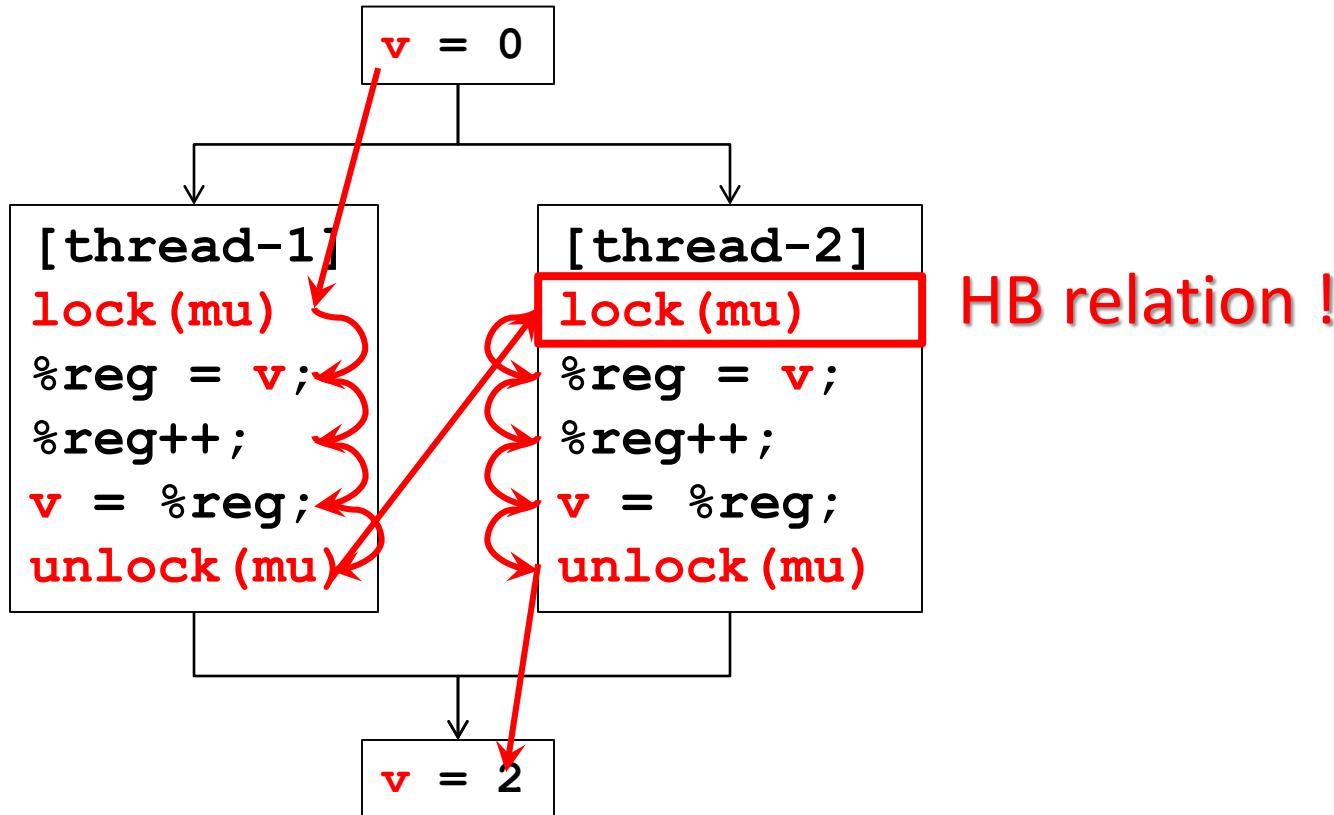


without HB relation !!

Example



Example



Pros and Cons

Pros:

Detect **true** data race

Cons:

Difficult to implement efficiently

- Each thread, shared-memory location, and concurrent access

Depend on the interleaving produced by
the scheduler

- Miss data race

LOCKSET BASED

Lock

The definition of lock:

- 1.a synchronization object used for **mutual exclusion**
- 2.a lock is either **available** or **owned** by a thread.
- 3.the operations on a lock **m** are **lock(m)** and **unlock(m)**

Lockset based Detectors

The definition of data race:

- 1.a pair of accesses to the **same** memory location
- 2.at least one access is a **write**
- 3.**No** lock protects all accesses to the same data

Lockset

Locking Discipline

- A programming policy that ensures the absence of data races
 - e.g. “**every variable shared between threads is protected by a mutual exclusion lock**”

Principle

- Check all shared memory accesses follow a consistent lock discipline
 - **monitors** all reads and writes, and **infer** the protection relation from the execution history

Algorithm of Lockset

The summary:

1. Let $\text{locks_held}(t)$ be the set of locks held by thread t
2. For each v , initialize $C(v)$ to the set of all locks
3. On each access to v by thread t ,
set $C(v) := C(v) \cap \text{locks_held}(t)$
if $C(v) = \{\}$, then issue a warning

Example

<u>Programs</u>	<u>locks held(t)</u>	<u>C(v)</u>
	{ }	{ mu1 , mu2 }
lock(mu1);	{ mu1 }	
lock(mu2);	{ mu1 , mu2 }	
v = v + 1;		{ mu1 , mu2 }
unlock(mu2);	{ mu1 }	
v = v + 2;		{ mu1 }
unlock(mu1);		
lock(mu2);	{ }	
v = v + 1;	{ mu2 }	
unlock(mu2);		{ } Warning!!

Three Challenges

#1 Initialization

- Shared variables are frequently initialized without holding a lock

#2 Read-only Shared Variable

- write once and read all the time without lock

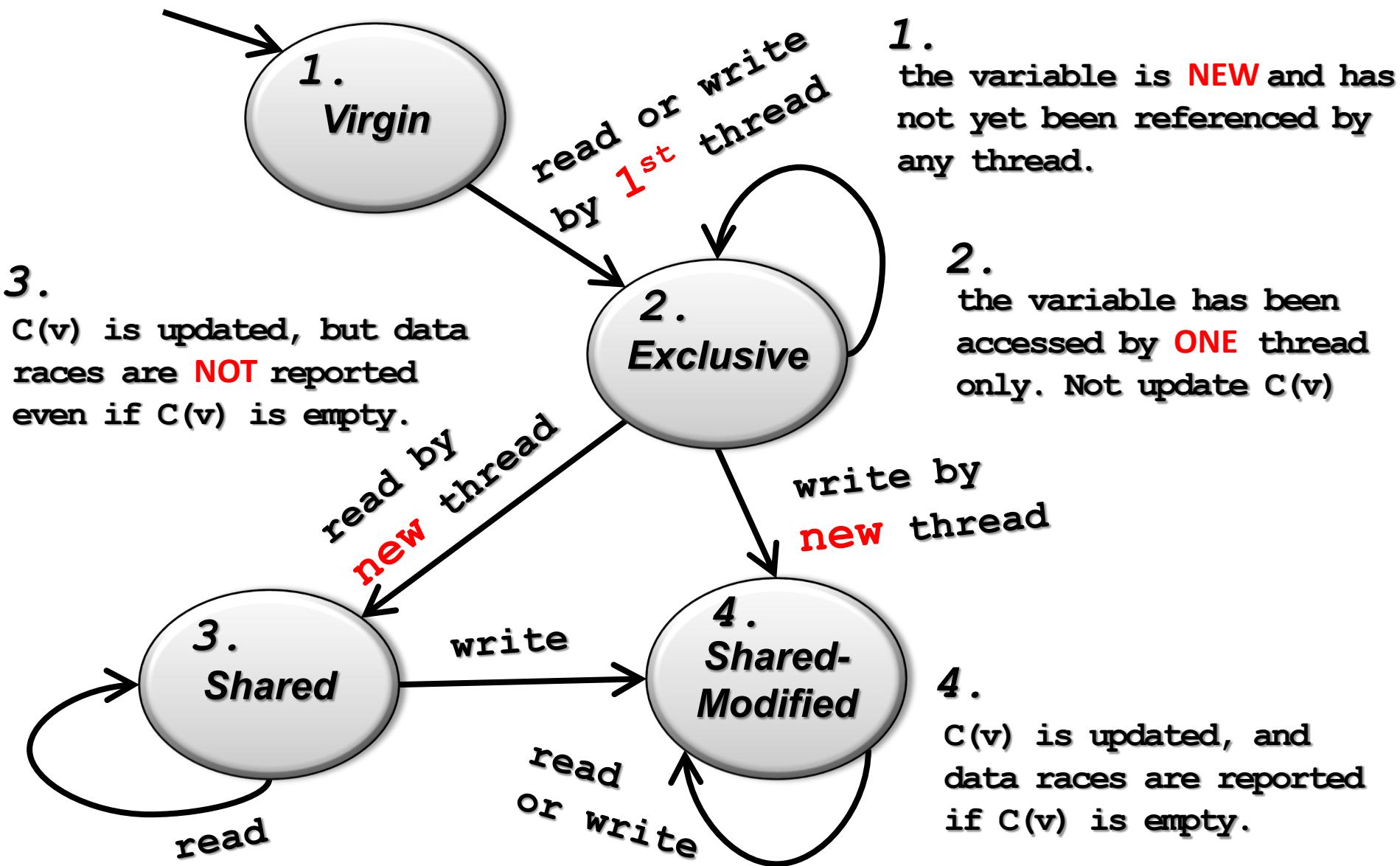
#3 Read-Write Lock

Solution to #1 and #2

Delay the refinement until the shared variable has been initialized

- No easy way to know when initialization is done
- Heuristic way: the first time to be accessed by the 2nd thread

Solution to #1 and #2



Solution to #3

Modify the algorithm:

1. Let `locks_held(t)` and `write_locks_held(t)` be the set of locks held in any mode and `write` mode by thread t
2. For each v, initialize $Cr(v)$ and $Cw(v)$ to the set of all locks
3. On each `read` to v by thread t,
set $Cr(v) := Cr(v) \cap locks_held(t)$
if $Cr(v) = \{ \}$, then issue a warning
4. On each `write` to v by thread t,
set $Cw(v) := Cw(v) \cap write_locks_held(t)$
if $Cw(v) = \{ \}$, then issue a warning

Pros and Cons

Pros:

More efficient way to detect data race

Predict data race that have not manifest

Cons:

Exists report false positive

- Memory reuse

Limits the synchronization method to lock

Deadlock

What is a deadlock



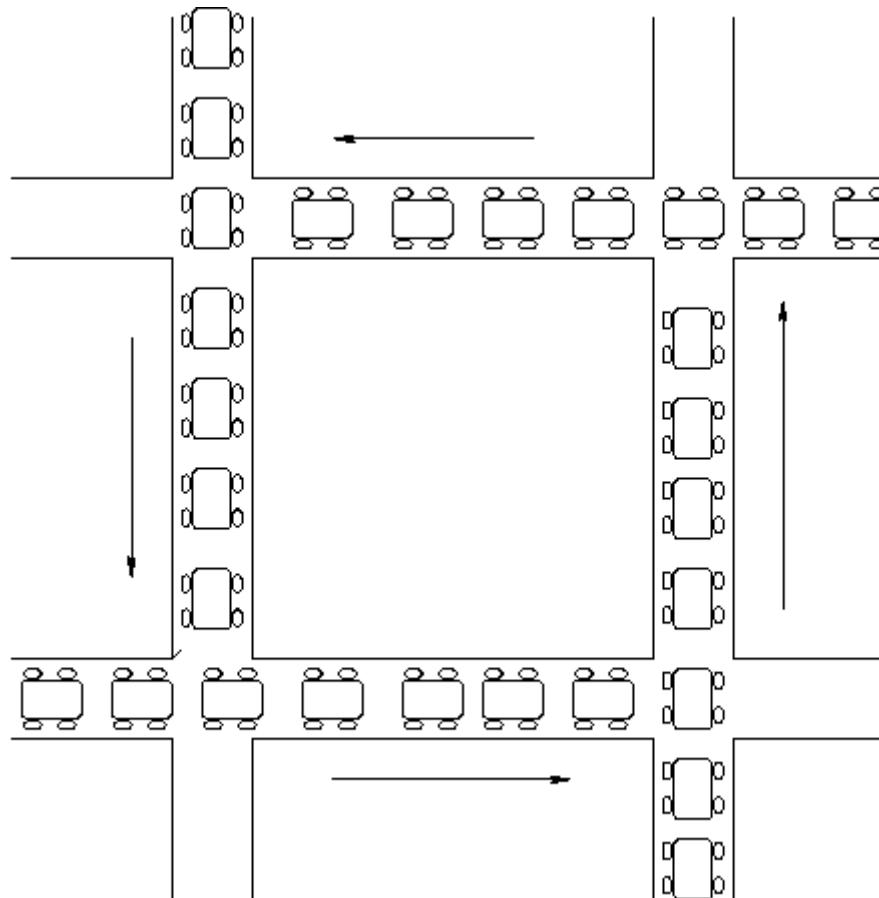
"Thanks for coming in. We'll get back to you as soon as we lower our expectations."

This guy has no experience!!!



Deadlock Definition

A set of processes is **deadlocked** when
every process in the set is waiting for an event that can
only be generated by some process in the set

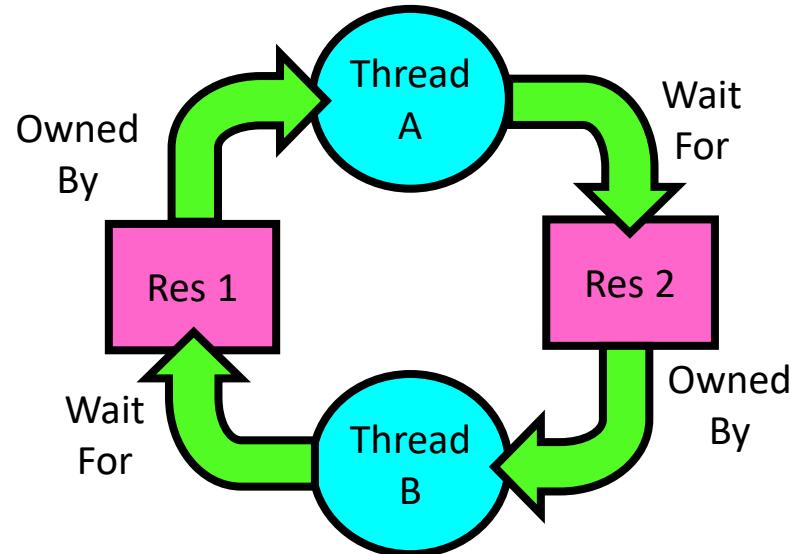


Deadlock

Circular waiting for resources

Thread A owns Res 1 and is waiting for Res 2

Thread B owns Res 2 and is waiting for Res 1



Four requirements for Deadlock

Mutual exclusion

Only one thread at a time can use a resource.

Hold and wait

Thread holding at least one resource is waiting to acquire additional resources held by other threads

No preemption

Resources are released only voluntarily by the thread holding the resource, after thread is finished with it

Circular wait

There exists a set $\{T_1, \dots, T_n\}$ of waiting threads

T_1 is waiting for a resource that is held by T_2

T_2 is waiting for a resource that is held by T_3

...

T_n is waiting for a resource that is held by T_1

A Graph Theoretic Model of Deadlock

Basic components of any resource allocation problem

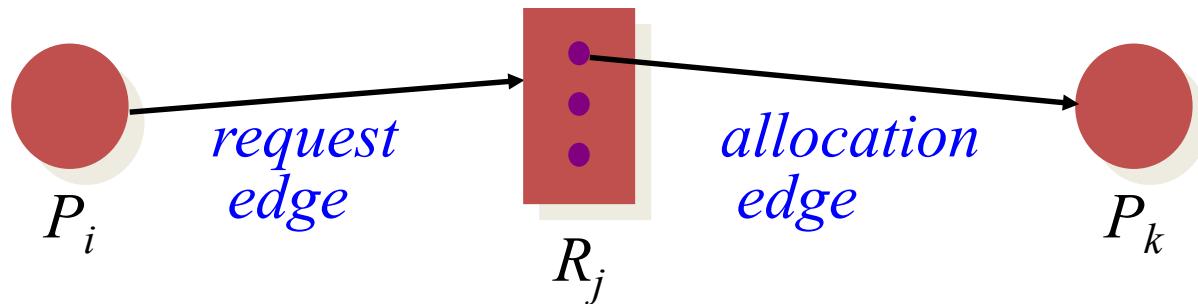
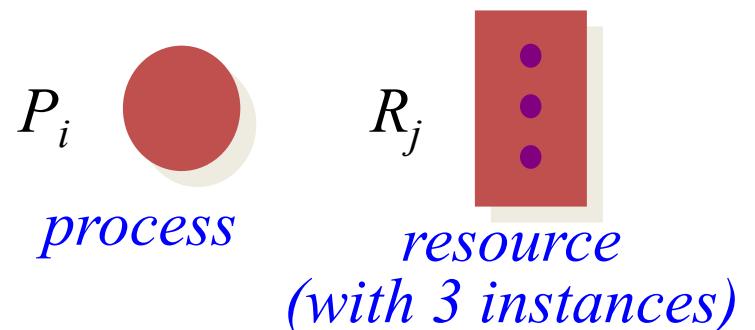
Processes and resources

Model the state of a computer system as a directed graph (called resource allocation graph, or RAG)

$$G = (V, E)$$

V = the set of vertices =
 $\{P_1, \dots, P_n\} \cup \{R_1, \dots, R_m\}$

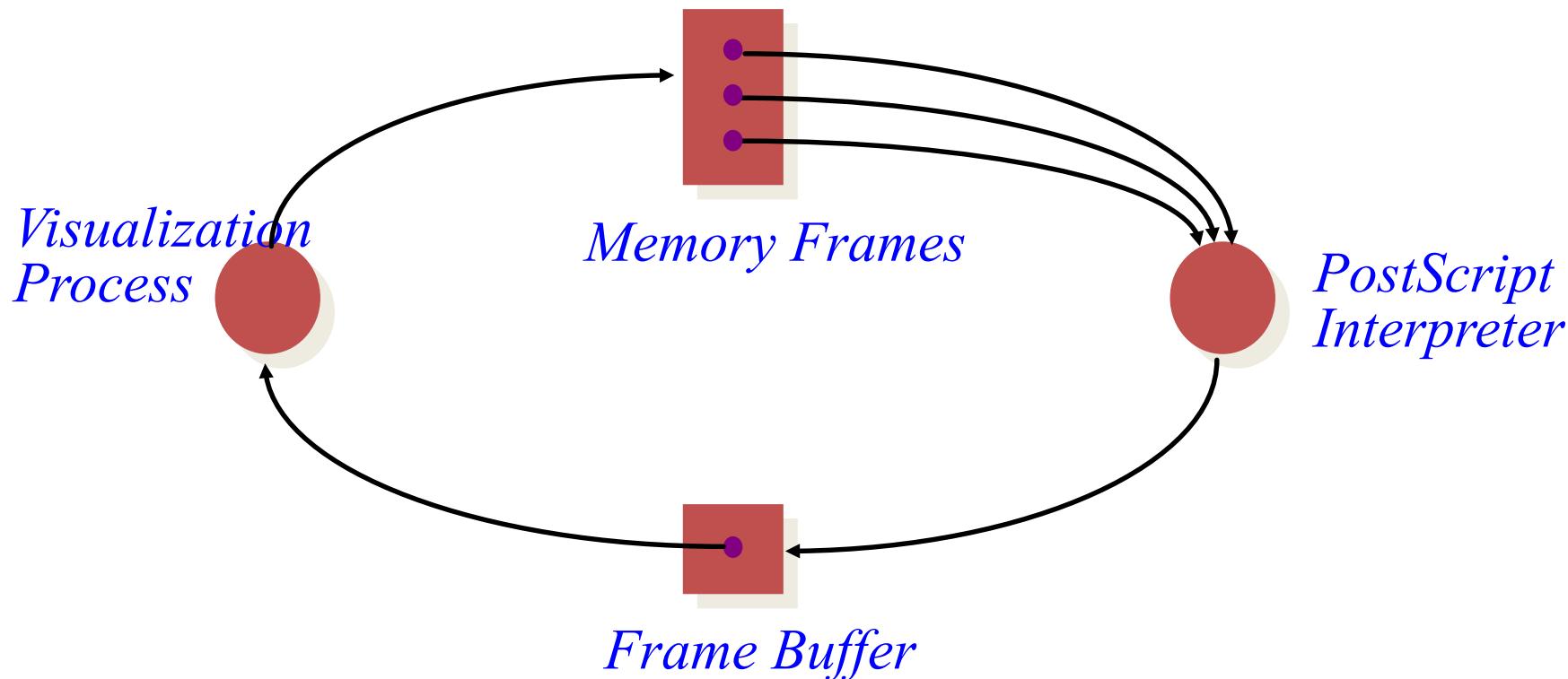
E = the set of edges =
{edges from a resource to a process} \cup {edges from a process to a resource}



Resource Allocation Graphs Example

A PostScript interpreter that is waiting for the **frame buffer lock** and a visualization process that is waiting for memory

$$V = \{PS\ interpret, visualization\} \cup \{memory\ frames, frame\ buffer\ lock\}$$

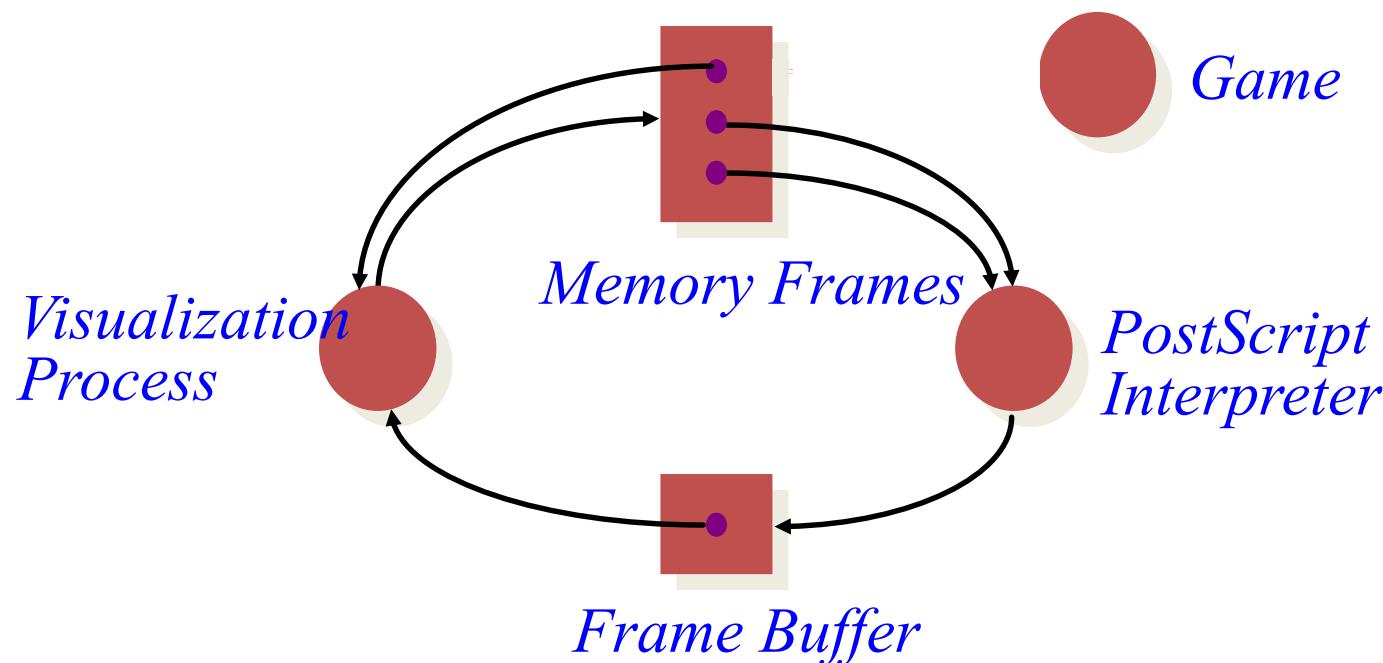


Resource Allocation Graphs & Deadlock

Theorem: If a resource allocation graph does not contain a **cycle** then no processes are deadlocked

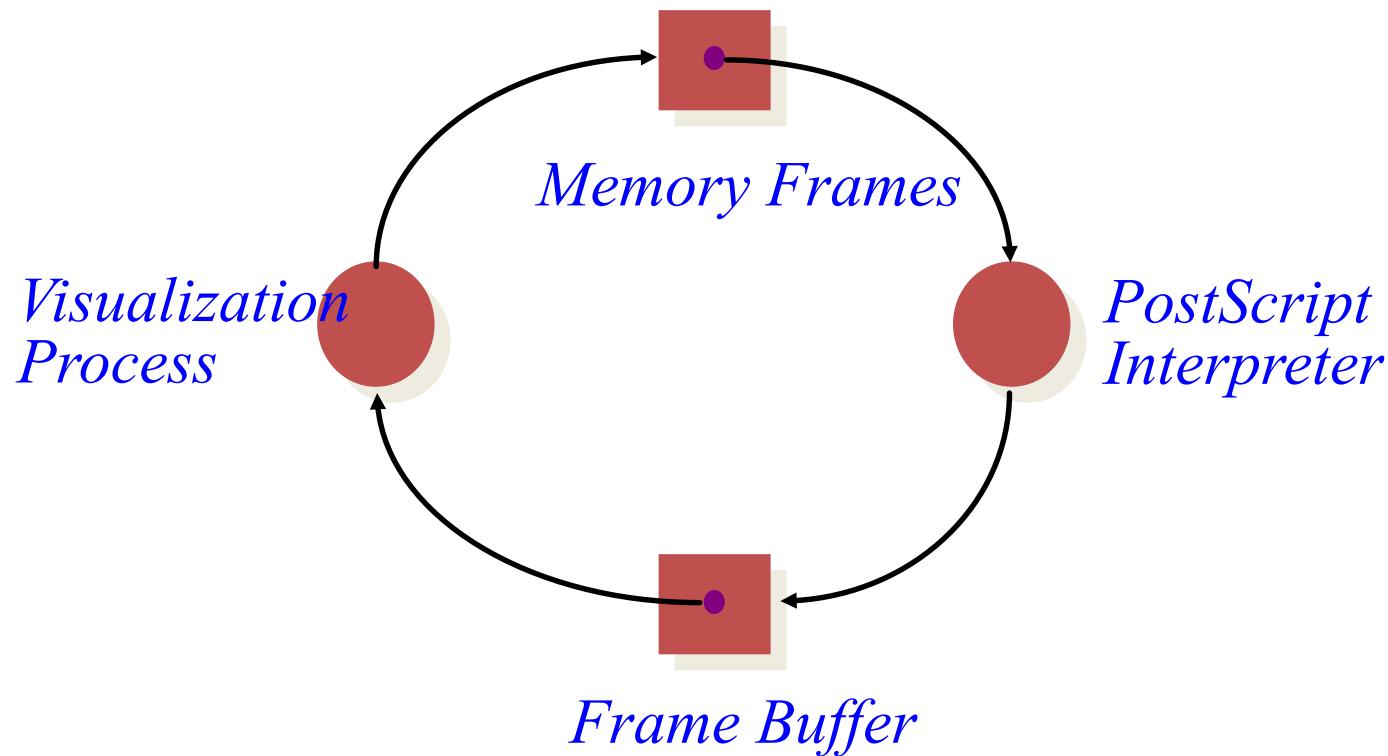
A cycle in a RAG is a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?



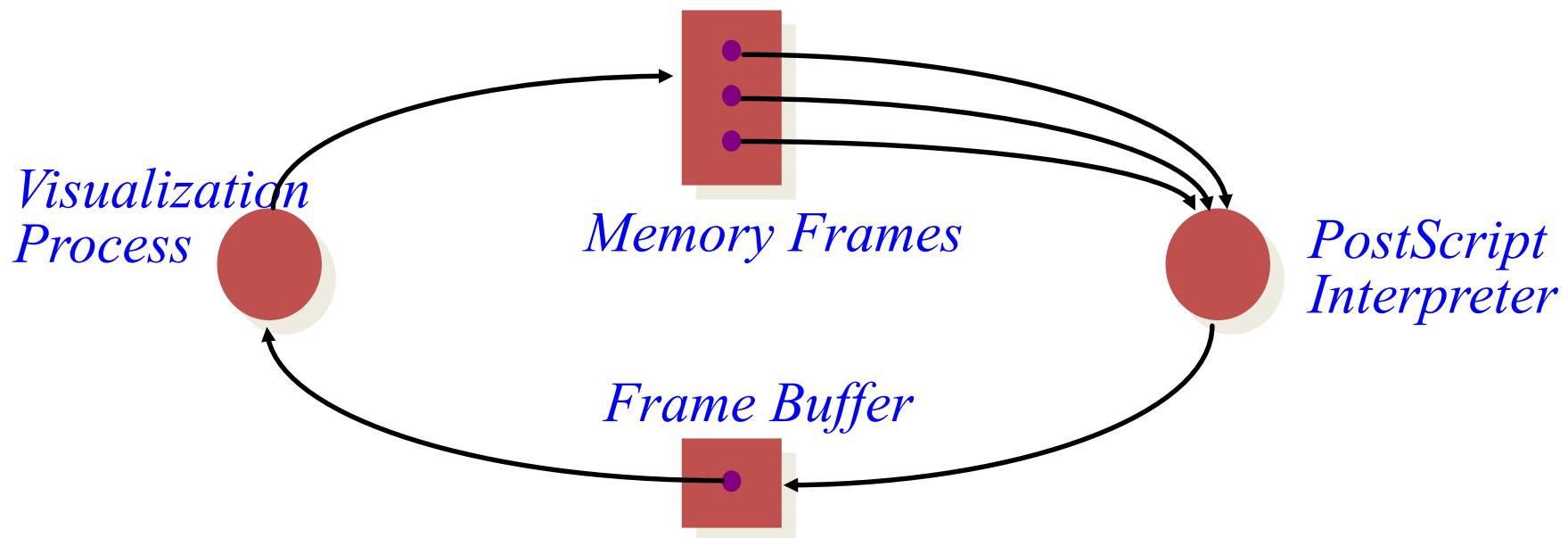
Single Resource RAG & Deadlocks

Theorem: If there is only a single unit of all resources then a set of processes are deadlocked iff there is a cycle in the resource allocation graph



An operational definition of deadlock

- A set of processes are deadlocked **iff** the following conditions hold simultaneously
 1. Mutual exclusion is required for resource usage
 2. A process is in a “hold-and-wait” state
 3. Preemption of resource usage is not allowed
 4. Circular waiting exists (a cycle exists in the RAG)



Dealing With Deadlock

Adopt some resource allocation protocol that ensures deadlock can never occur

Deadlock prevention/avoidance

Guarantee that deadlock will never occur

Generally breaks one of the following conditions:

Mutex, Hold-and-wait, No preemption, Circular wait

Deadlock detection and recovery

Admit the possibility of deadlock occurring and periodically check for it

On detecting deadlock, abort

Breaks the no-preemption condition

Deadlock Avoidance by Resource Ordering

Eliminate circular waiting by ordering all locks (or semaphores, or resources)

All code grabs locks in a predefined order

Problems?

Maintaining global order is difficult, especially in a large project

Global order can force a client to grab a lock earlier than it would like, tying up a resource for too long

Deadlock Detection & Recovery

Detection: periodic check RAG for cycles

How often should the OS check for deadlock?

After every resource request?

Only when we suspect deadlock has occurred?

Recovery: break the deadlock

Abort all deadlocked processes & reclaim their resources?

Abort one process at a time until all cycles in the RAG are eliminated?

Start from low priority process?

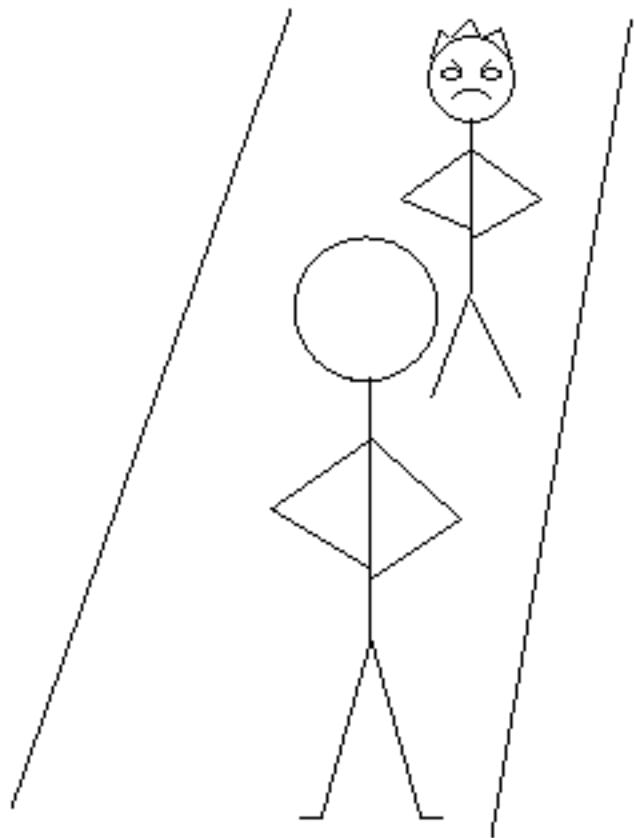
Start from processes with most allocation of resources?

LiveLock

Similar to a deadlock

The states of the processes constantly change with regard to one another

none progressing



Deadlock Pitfalls in Reality

Even **deadlock-free** code would deadlock once deployed

Due to dependencies on deadlock-prone third party libraries or runtimes

Examples: web browsers plug-ins, Java beans.

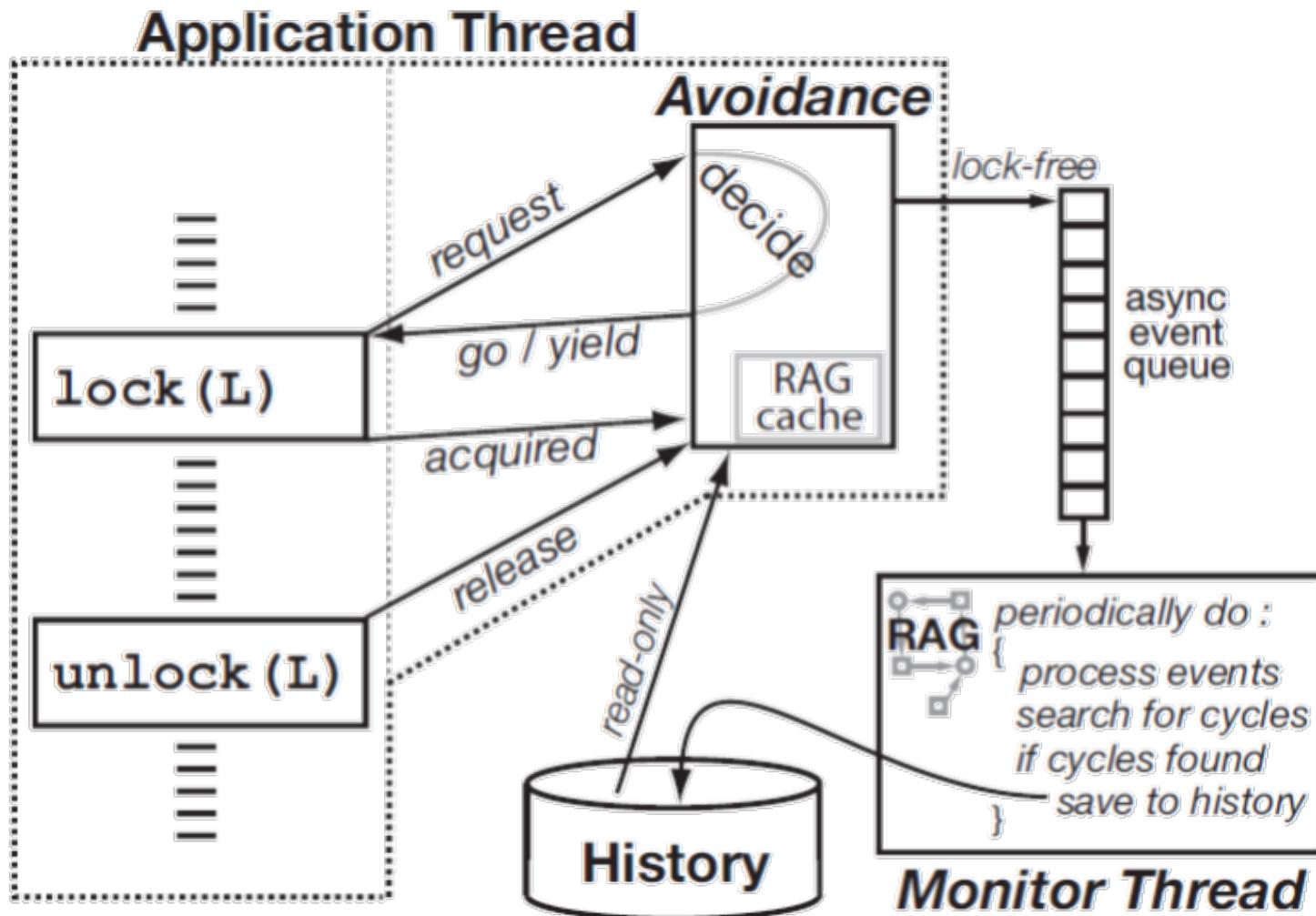
Upgrade of such libraries/runtimes can introduce new deadlocks during execution.

Dimmunix - Teaser

So, it is often **hard** to handle a deadlock. But what if...

- *Deadlock immunity:*
 - once afflicted by a given deadlock, develop resistance against future concurrences of that and similar deadlocks
- *Dimmunix* - a tool for developing deadlock immunity.
- The first time a deadlock pattern manifests
 - Dimmunix automatically captures its *signature* and subsequently avoids entering the same pattern.

Dimmunix Architecture



Introduction to Virtualization

Yubin Xia
Software School
Shanghai Jiao Tong University

Some Slides adapted from
VMWare's academic course plan

The Reality (1)– Computer is powerful

Processor

Core counts: about **20%** per year

Memory

DRAM capacity: about **60%** per year (4x every 3 years)

Memory speed: about **10%** per year

Cost per bit: improves about **25%** per year

Disk

capacity: about **60%** per year

Total use of data: **100%** per 9 months!

Network Bandwidth

Bandwidth increasing more than **50%** per year! (Nielsen's Law)

The Reality (2) – Business Trend

Low utilization of Computers:

Most are below 20%

Think: why it's Amazon, an e-commerce company, who develops cloud first?

Maintenance of machines is costly:

~5X to 10X cost of HW/SW

Service **outages** are frequent and expensive

65% of IT managers report that their websites were unavailable to customers over a 6-month period

25%: 3 or more outages

social effects: negative press, loss of customers who “click over” to competitor

Why not virtualize?

Today's
data center



Cloud
computing



Fault resist
Cost Effect

Easy to maintain
Virtual
resource
pool

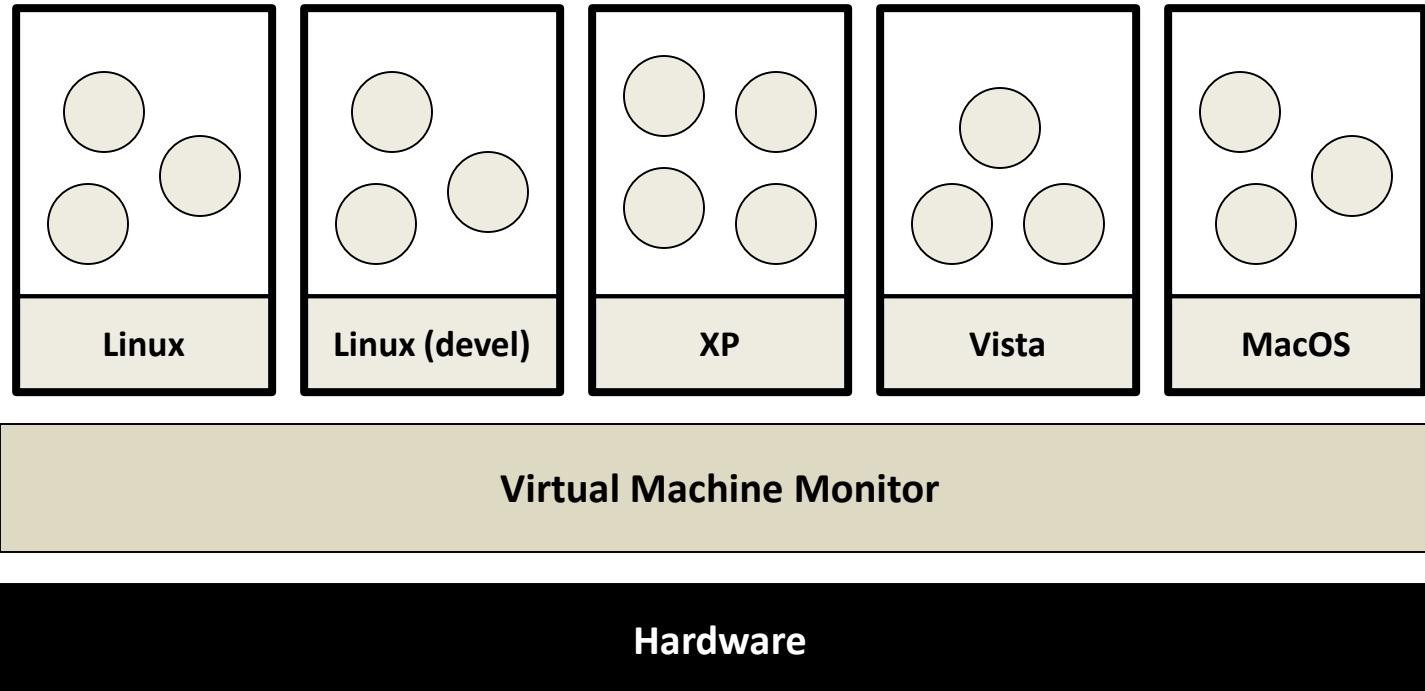
*“Any problem in computer science
can be solved with another level
of indirection.”*

*–David Wheeler in Butler Lampson’s
1992 ACM Turing Award speech.*

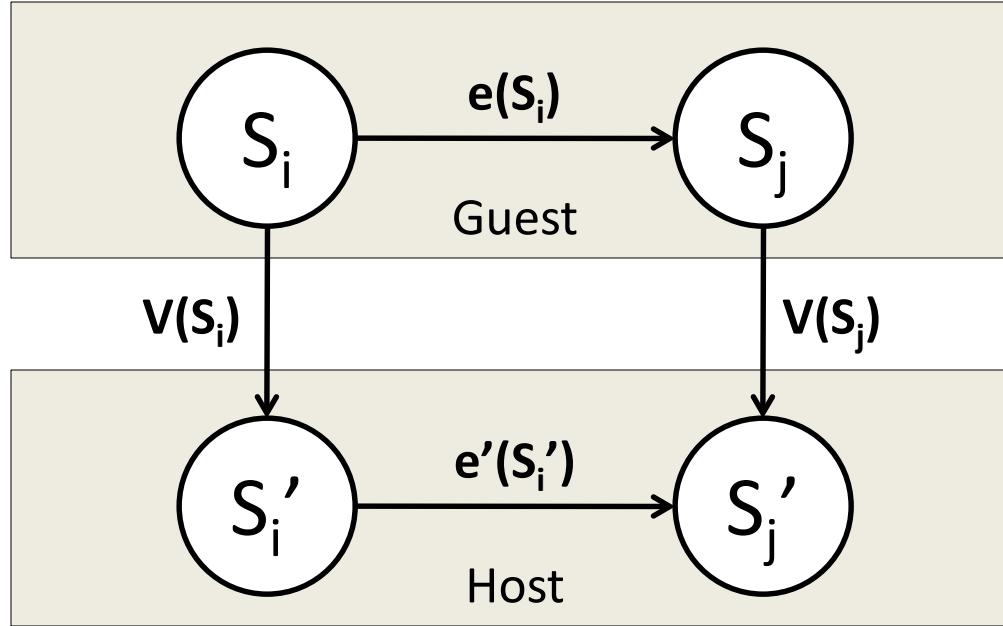
Outline

- What is virtualization?
- Virtualization classification
- Monitor Architectures

What is Virtualization



Isomorphism



Formally, virtualization involves the construction of an isomorphism from guest state to host state.

Virtualization Properties

- Isolation
- Encapsulation
- Interposition

Isolation

- Fault Isolation
 - Fundamental property of virtualization
- Software Isolation
 - Software versioning
 - The DLL-Hell in Windows
- Performance Isolation
 - Accomplished through scheduling and resource allocation

Encapsulation

- All VM state can be captured into a file
 - Operate on VM by operating on file
 - mv, cp, rm
- Complexity
 - Proportional to virtual HW model
 - Independent of guest software configuration

Interposition

- All guest actions go through monitor
- Monitor can inspect, modify, deny operations
- Enable
 - Compression
 - Encryption
 - Profiling
 - Translation

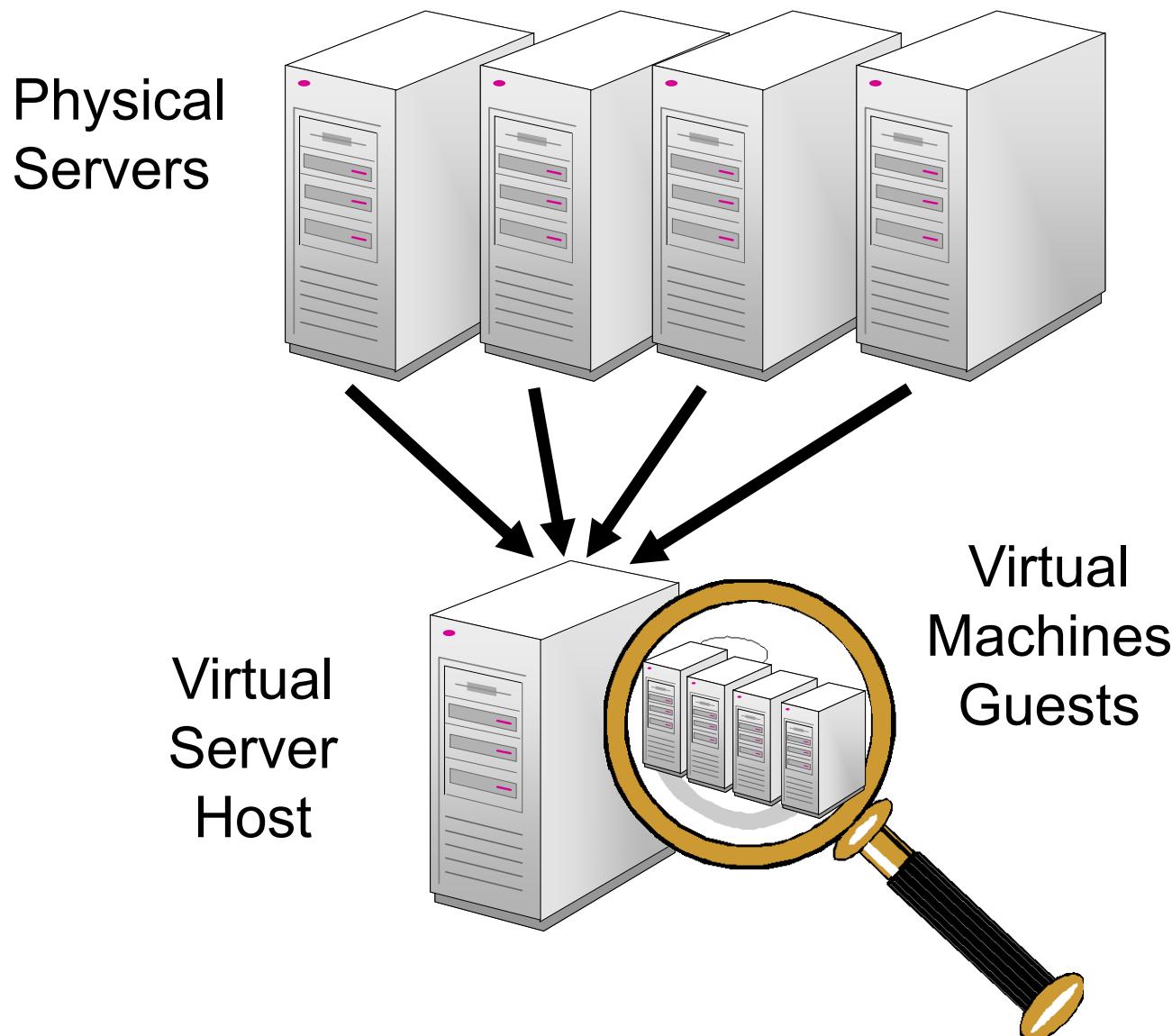
Why Not Using the OS?

- It's about interfaces
 - VMMs operate at the hardware interface
 - Hardware interface are typically **smaller, better** defined than software interfaces
- Microkernel for commodity Operating Systems
- Disadvantages of being in the monitor
 - Semantic gap: VMM barely knows what guest is doing
 - E.g., OS knows data and metadata, VMM not

Virtualization benefits

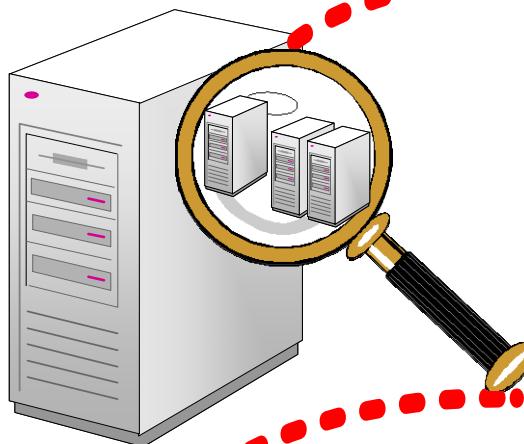
- Increased resource utilization:
 - Server consolidation
- Mobility
- Enhanced Security
- Trusted Computing
- Test and Deployment
- ...

Virtualization: server consolidation

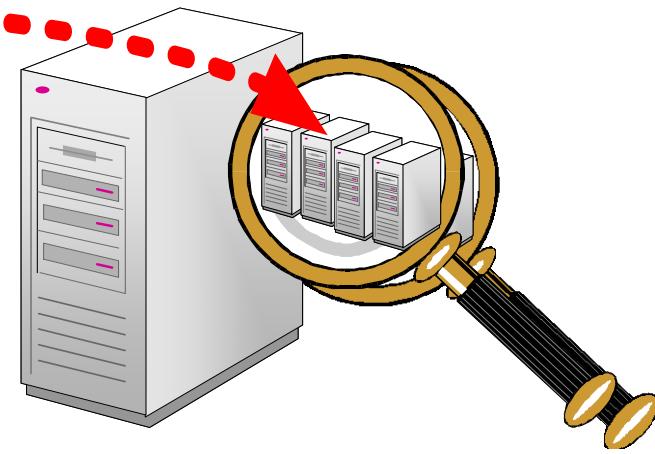


Mobility: load balance

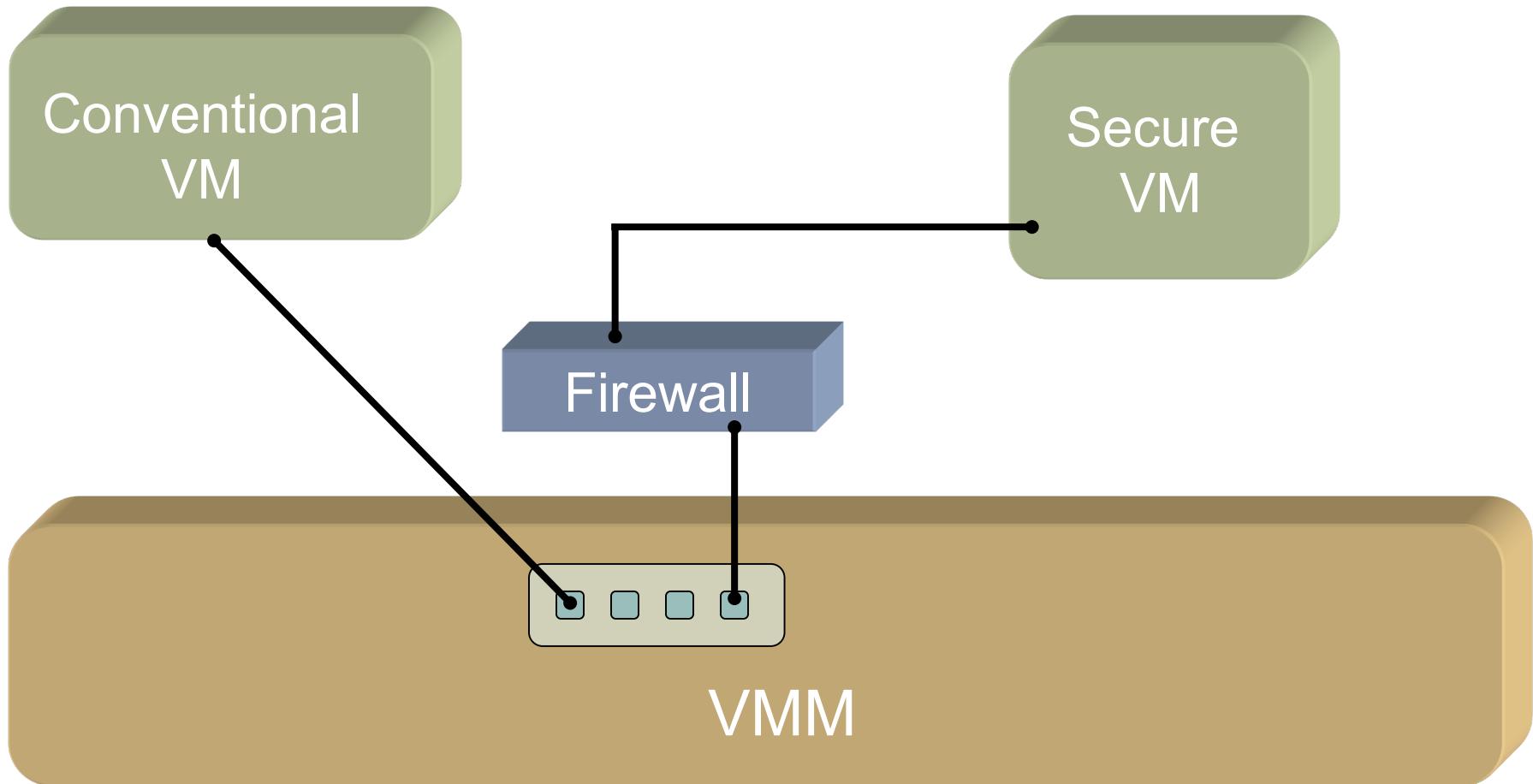
Server 1
CPU Utilization = 90%



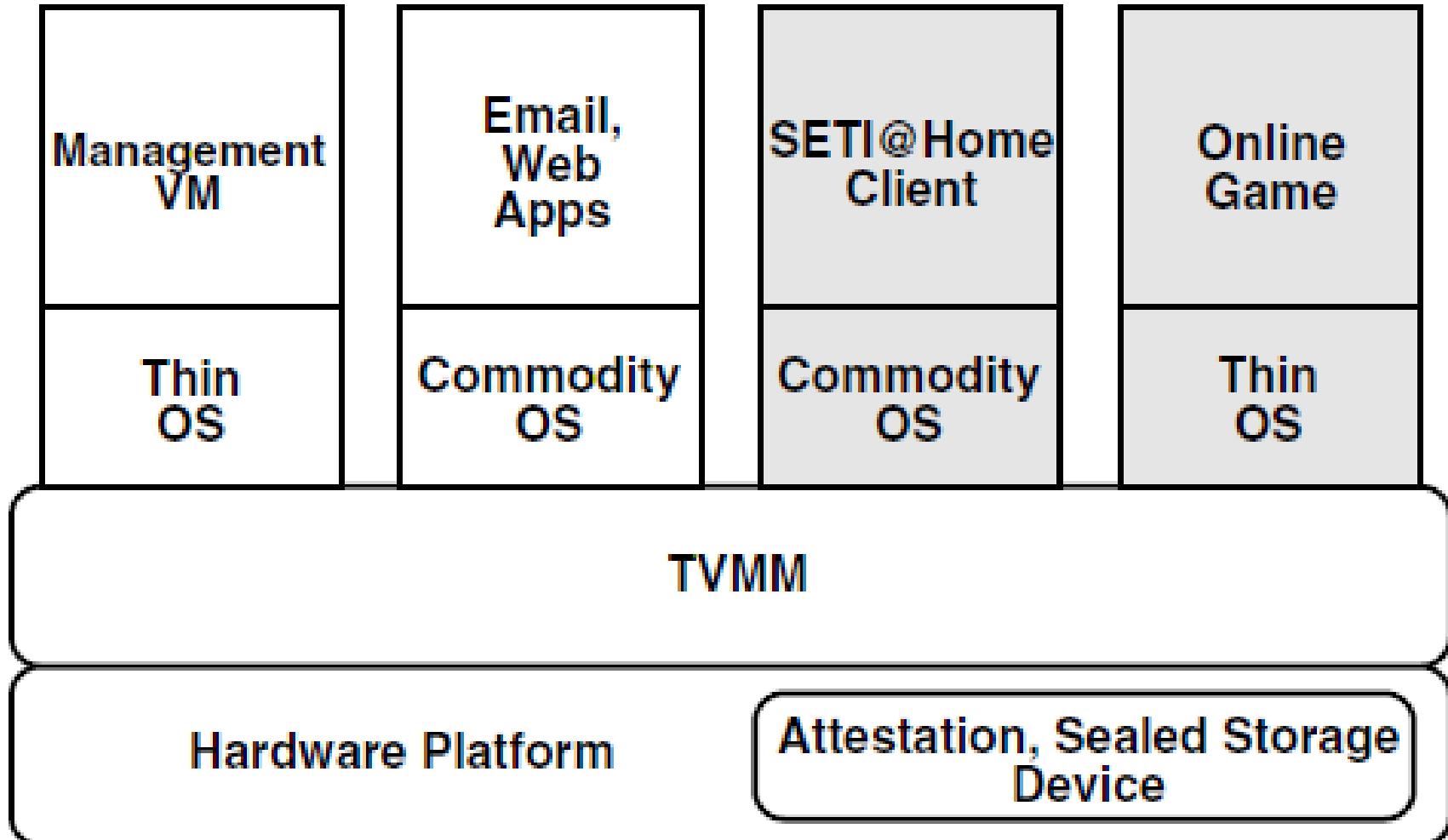
Server 2
CPU Utilization = 50%



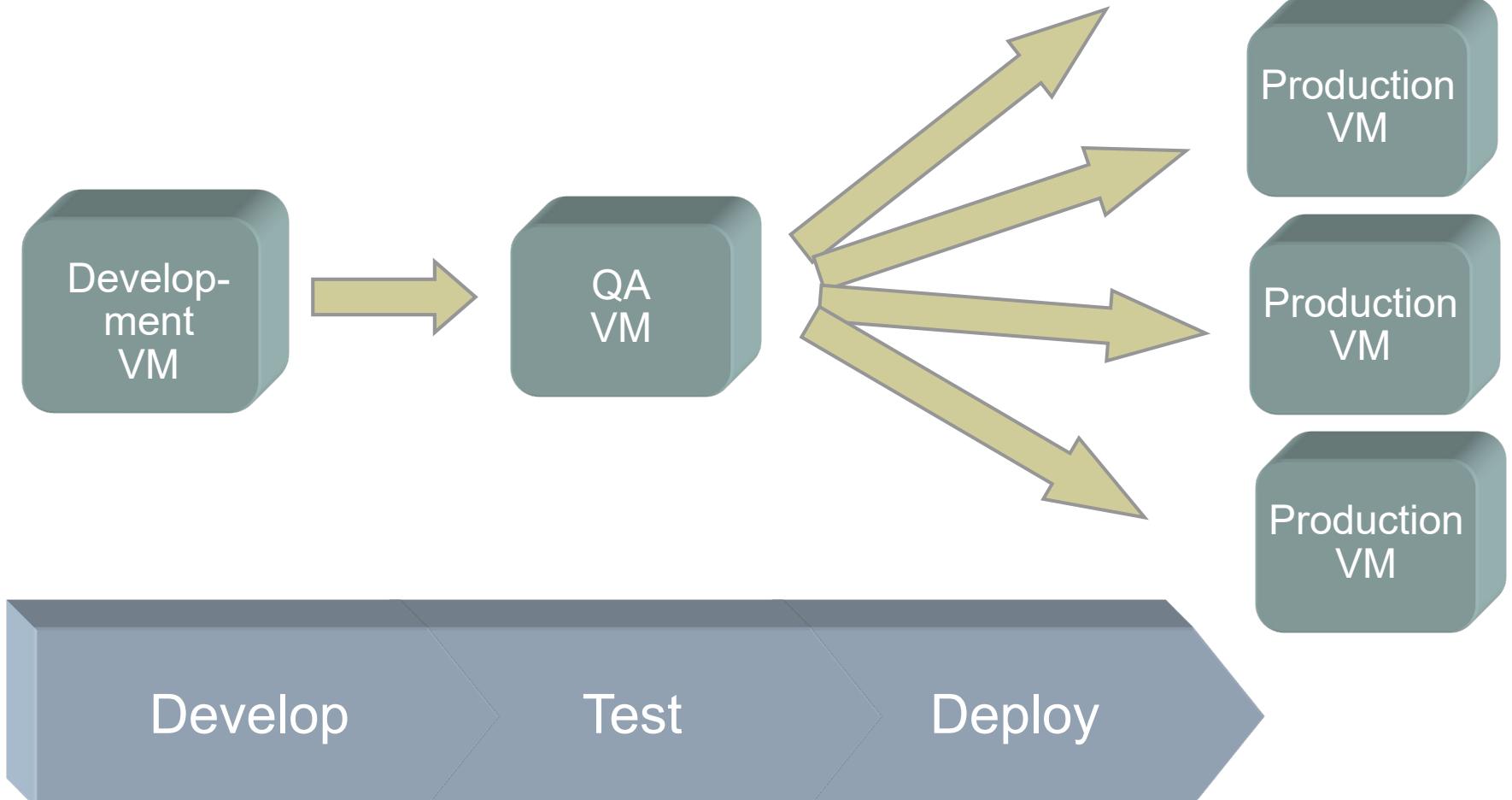
Enhanced Security



Trusted Computing



Testing and Deployment

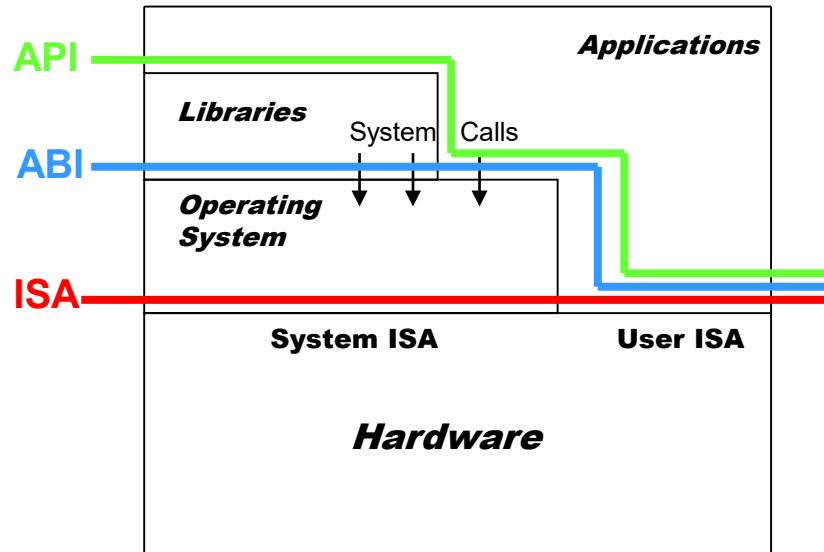


HOW TO VIRTUALIZE

How to Virtualize?

- Run an existing software component on another platform
 - Component: applications, OS
 - Platform: hardware, software
- Keep isolation
 - The component cannot access other's data or environment
- Keep transparency
 - No modification to the software component

Architecture & Interfaces



- **API** – application programming interface
- **ABI** – application binary interface
- **ISA** – instruction set architecture

Architecture, Implementation Layers

- Implementation Layer : ISA
 - Instruction Set Architecture
 - Divides hardware and software
 - User ISA and System ISA
 - User ISA example: mov \$0, %eax
 - System ISA examples: cli, sti, mov \$0, %cr0

Architecture, Implementation Layers

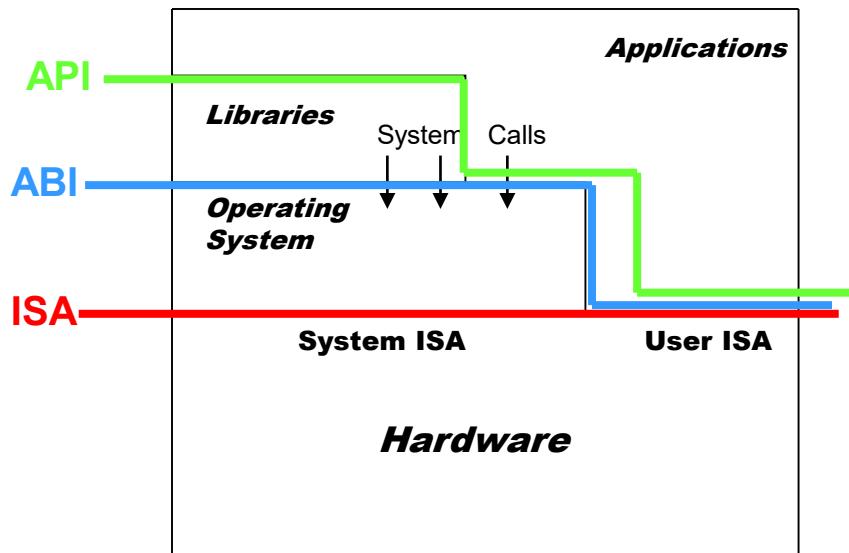
- Implementation Layer : ABI
 - Application Binary Interface
 - Provides a program with access to the hardware resource and services available in a system
 - Consists of User ISA and System Call Interfaces
 - Also contains calling convention

Architecture, Implementation Layers

- Implementation Layer : API
 - Application Programming Interface
 - Key element is Standard Library (or Libraries)
 - Typically defined at the source code level of High Level Language
 - **clib** in Unix environment : supports the UNIX/C programming language

Question: At Which Layer?

- Hello world
- Web game
- Dota
- Office 2013
- Windows 8
- Java applications
- Python scripts

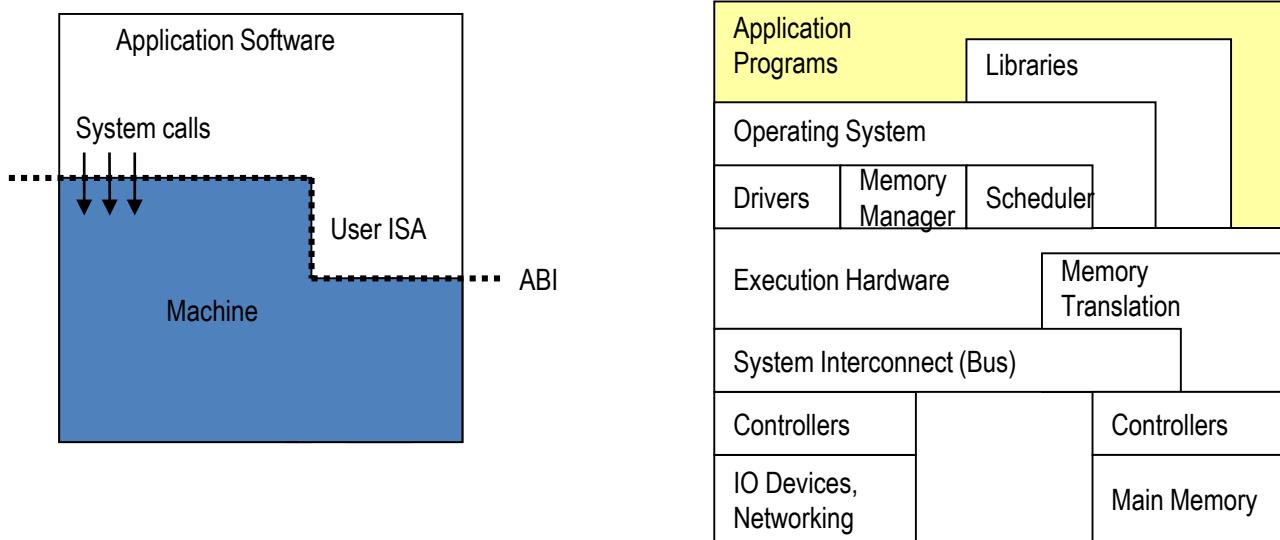


What is a VM and Where is the VM?

- What is “Machine”? 2 perspectives
- From the perspective of a process:
 - ABI provides interface between process and machine
- From the perspective of a system:
 - Underlying hardware itself is a machine
 - ISA provides interface between system and machine

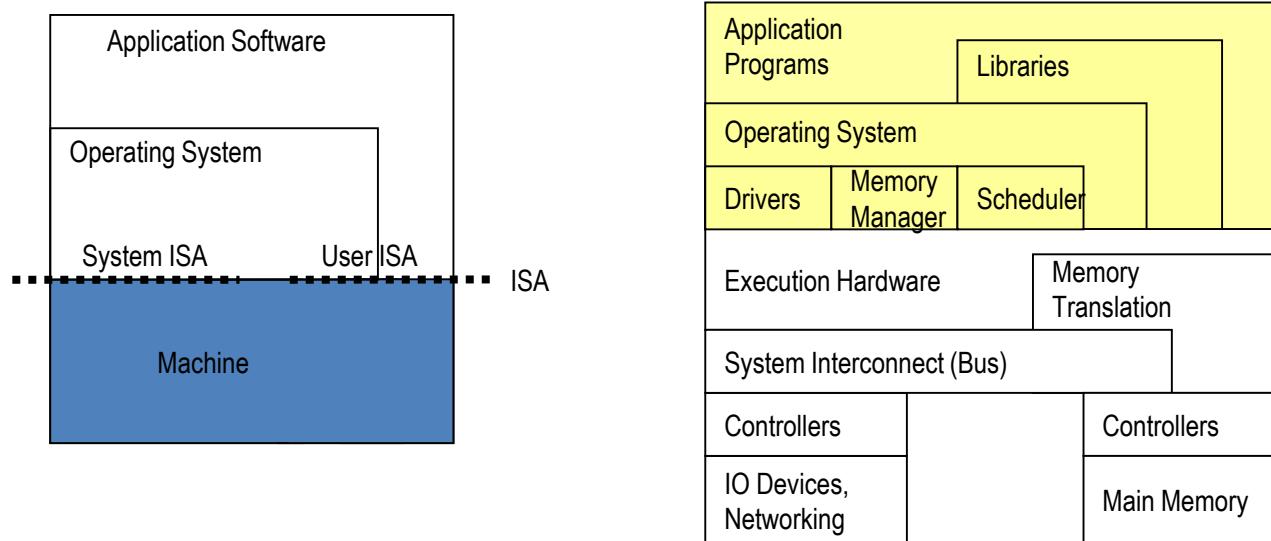
What is a VM and Where is the VM?

- Machine from the perspective of a process
 - ABI provides interface between process and machine



What is a VM and Where is the VM?

- Machine from the perspective of a system
 - ISA provides interface between system and machine

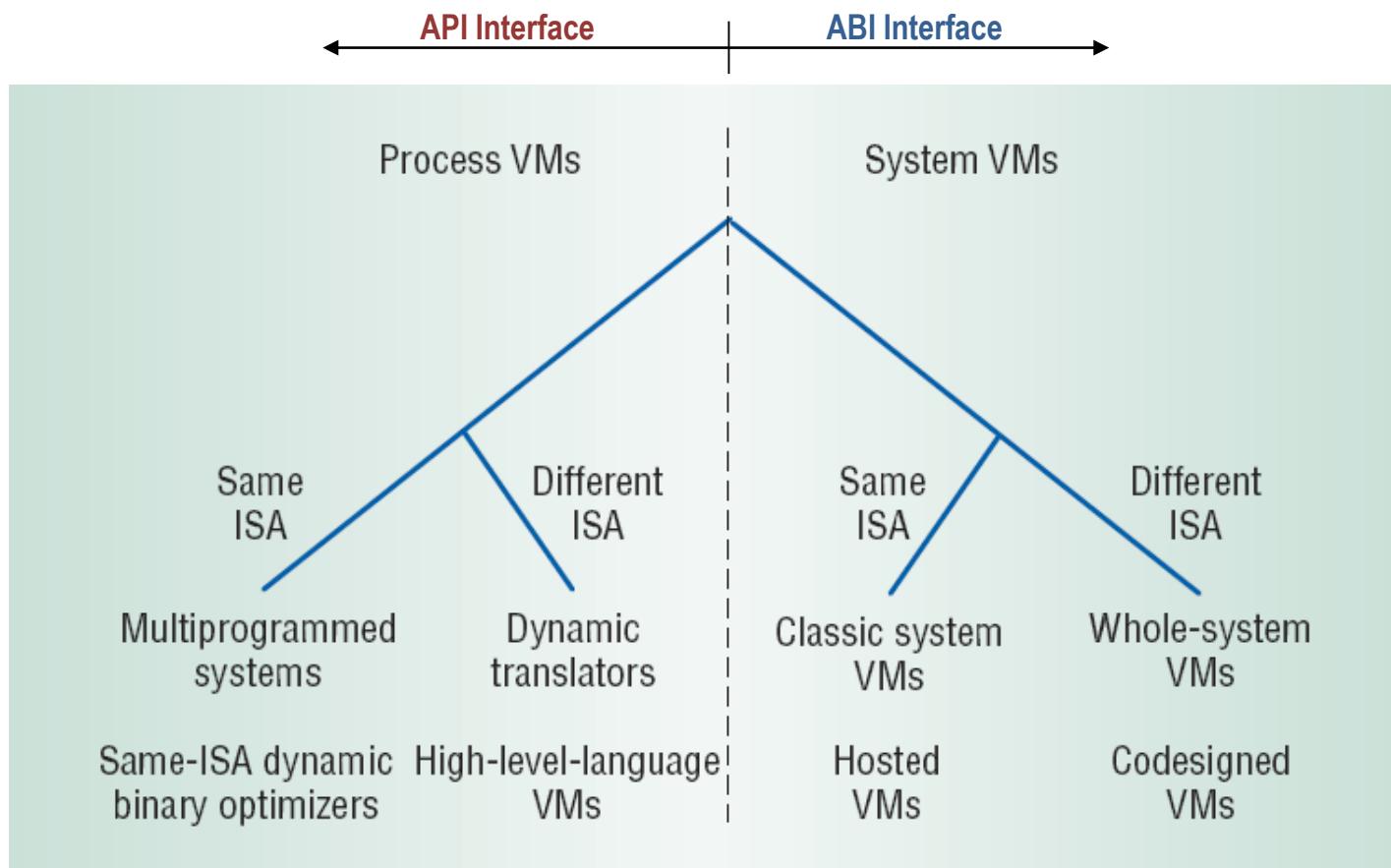


TYPES OF HYPERVISOR

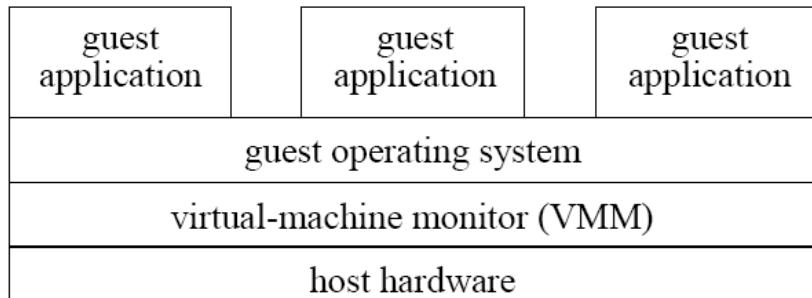
Types of Virtualization

- Process Virtualization
 - Language construction
 - Java, .NET
 - Cross-ISA emulation
 - Apple's 68000-PowerPC-Intel Transition
 - Application virtualization
 - Sandboxing, mobility
- Device Virtualization
 - RAID
- System Virtualization
 - VMware
 - Xen
 - Microsoft's Hyper-V (aka. Viridian)

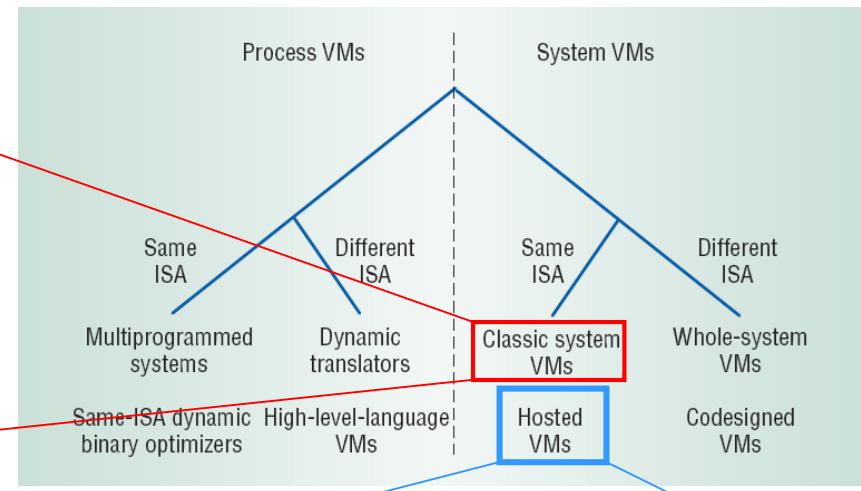
Design Space (Level vs. ISA)



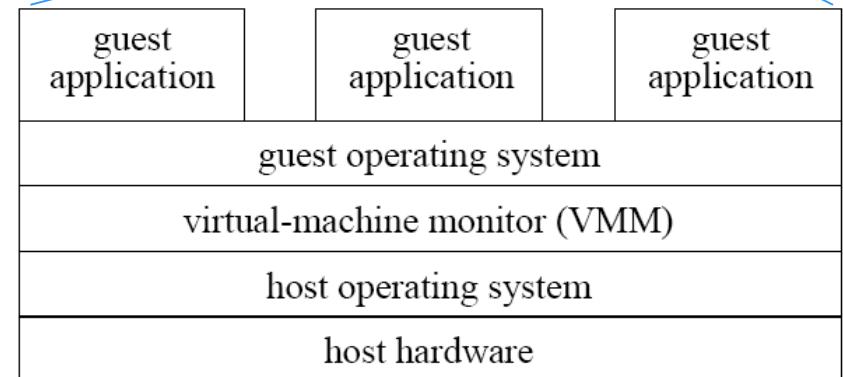
System VMs



Type 1



- **Type 1:** runs directly on hardware
 - High performance
 - e.g., Xen, VMware ESX Server
- **Type 2:** runs on Host OS
 - Ease of construction/installation
 - e.g., VMware Workstation

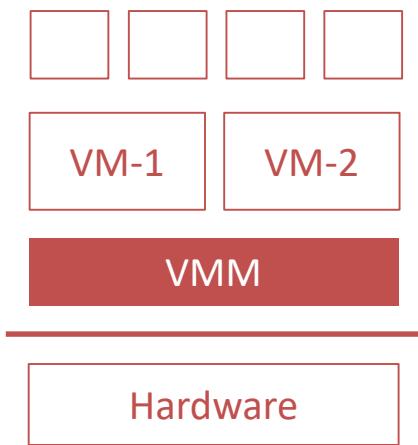


Type 2

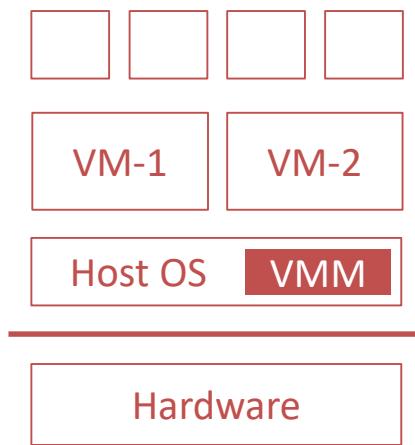
System Virtual Machine Monitor Architectures

- Traditional
- Hosted
 - VMware Workstation
- Hybrid
 - VMware ESX
 - Xen
- Hypervisor

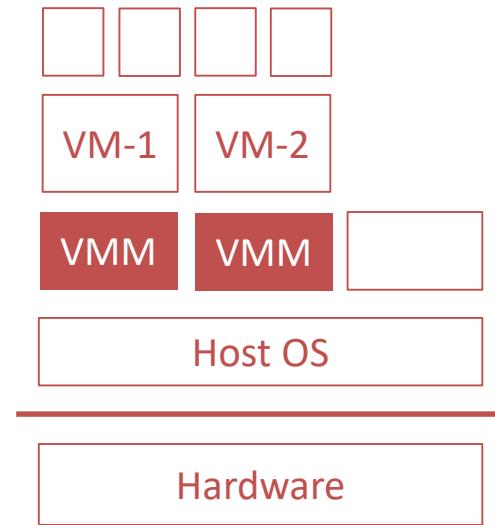
Different Architectures of VMM



Xen

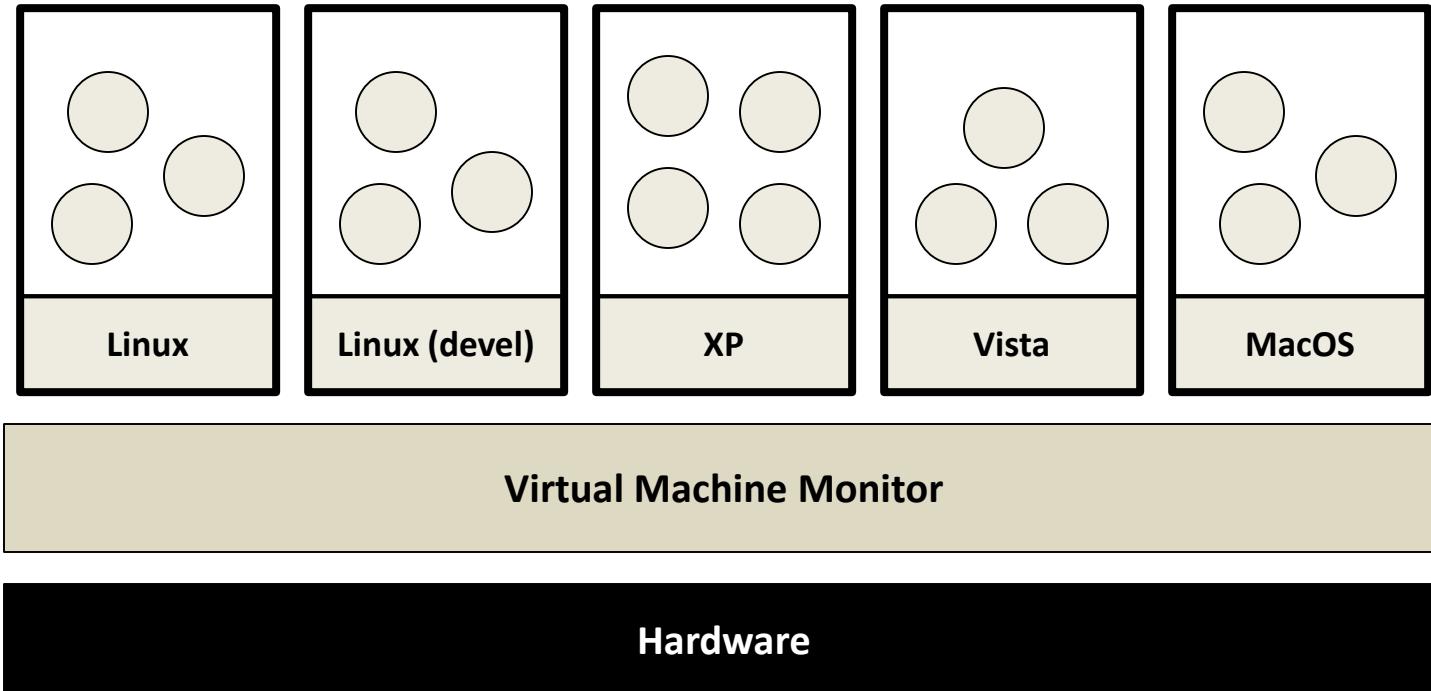


Linux KVM



Qemu

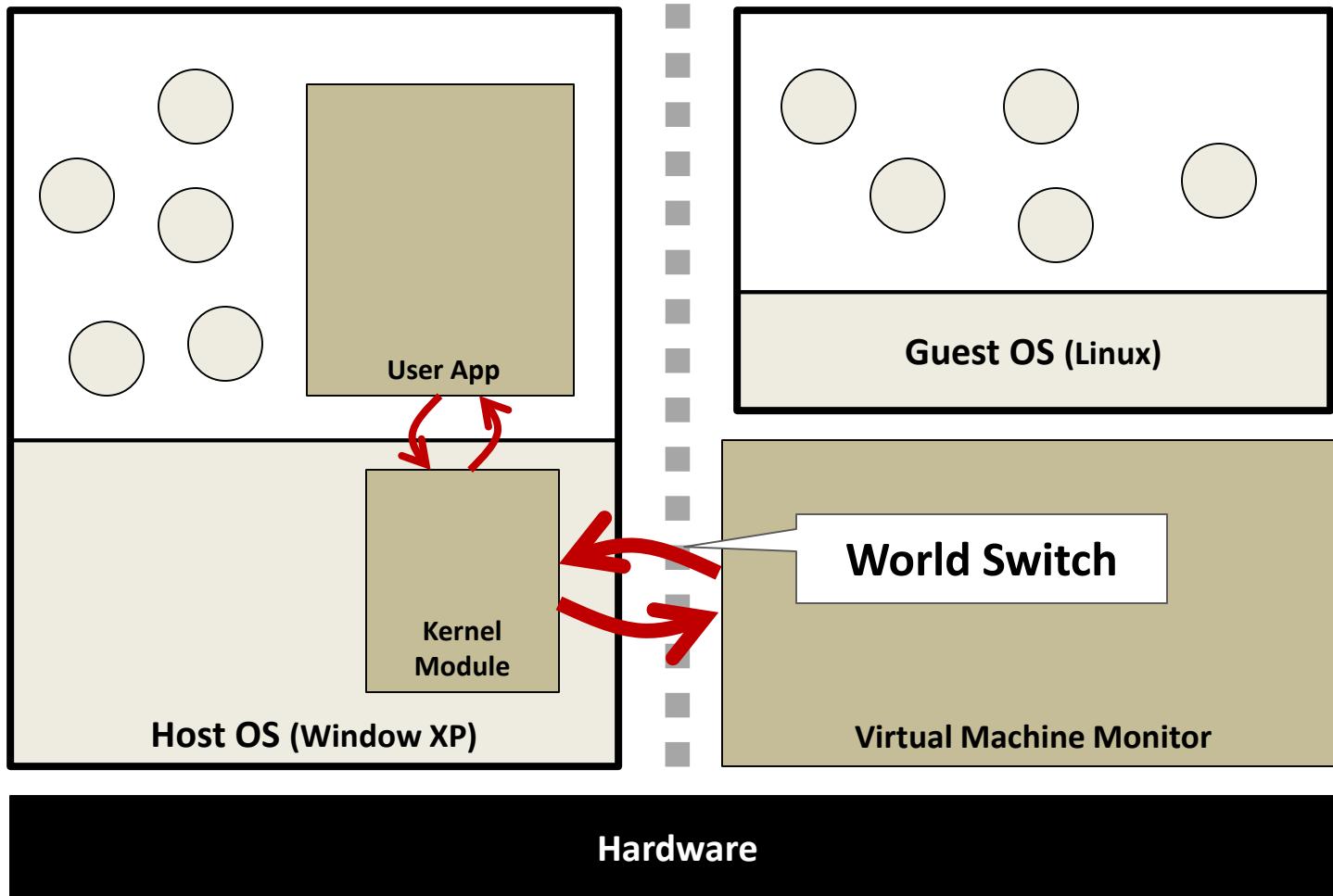
Type-1 Hypervisor (VMM)



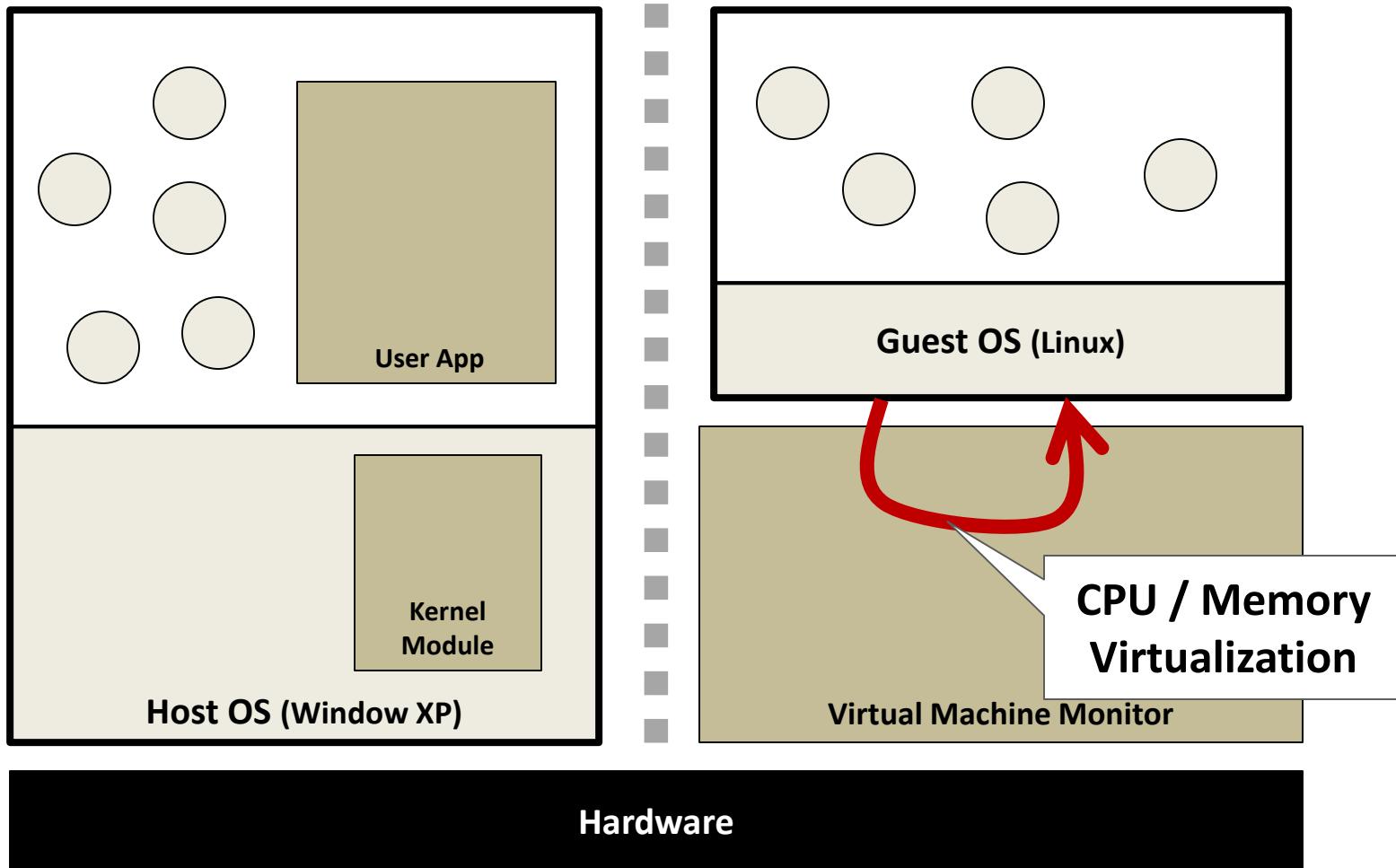
Type-2: Hosted Virtual Machines

- Goal:
 - Run Virtual Machines as an application on an existing Operating System
- Why
 - Application continuity
 - Reuse existing device drivers
 - Leverage OS support
 - File system
 - CPU Scheduler
 - VM management platform

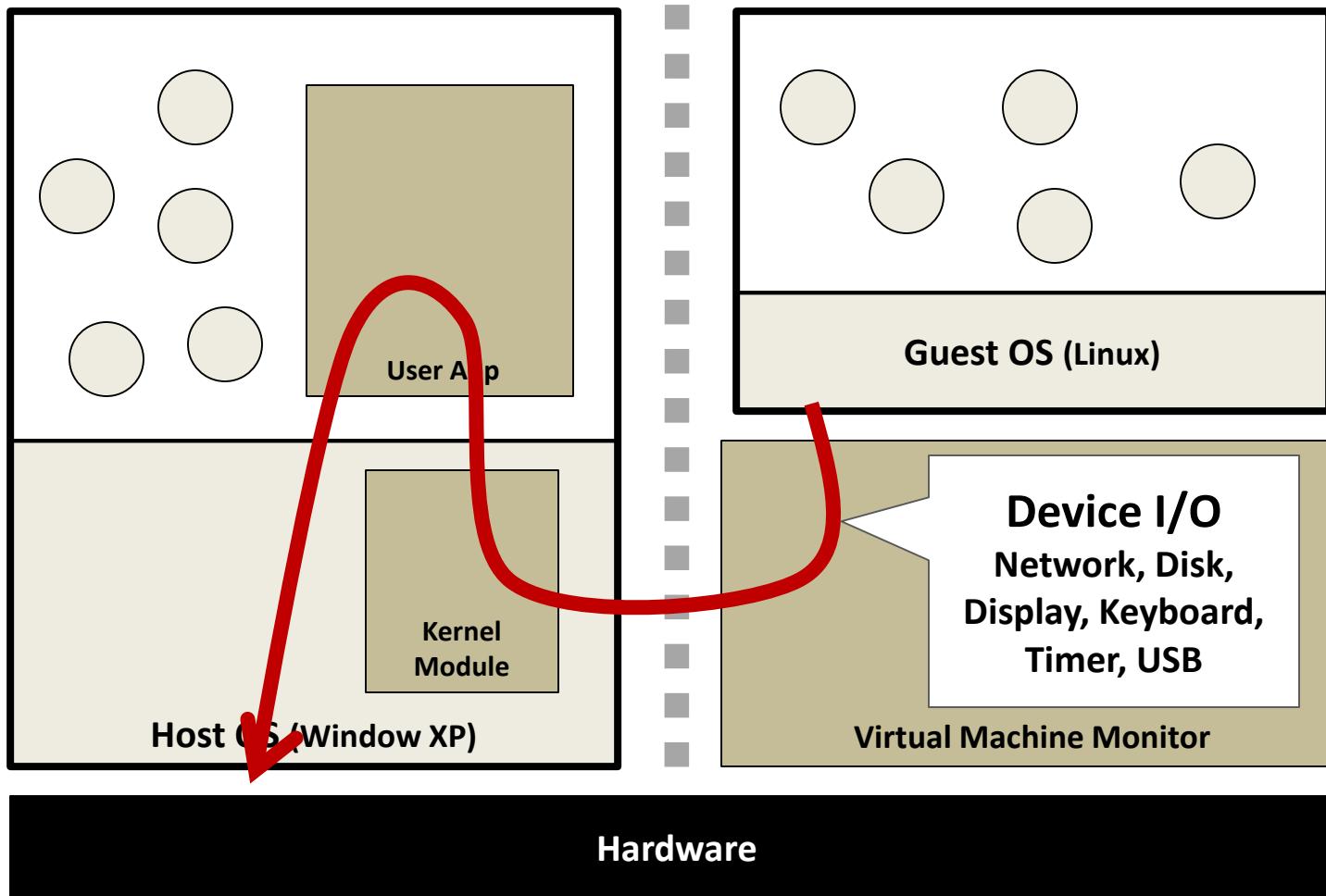
Hosted Monitor Architecture



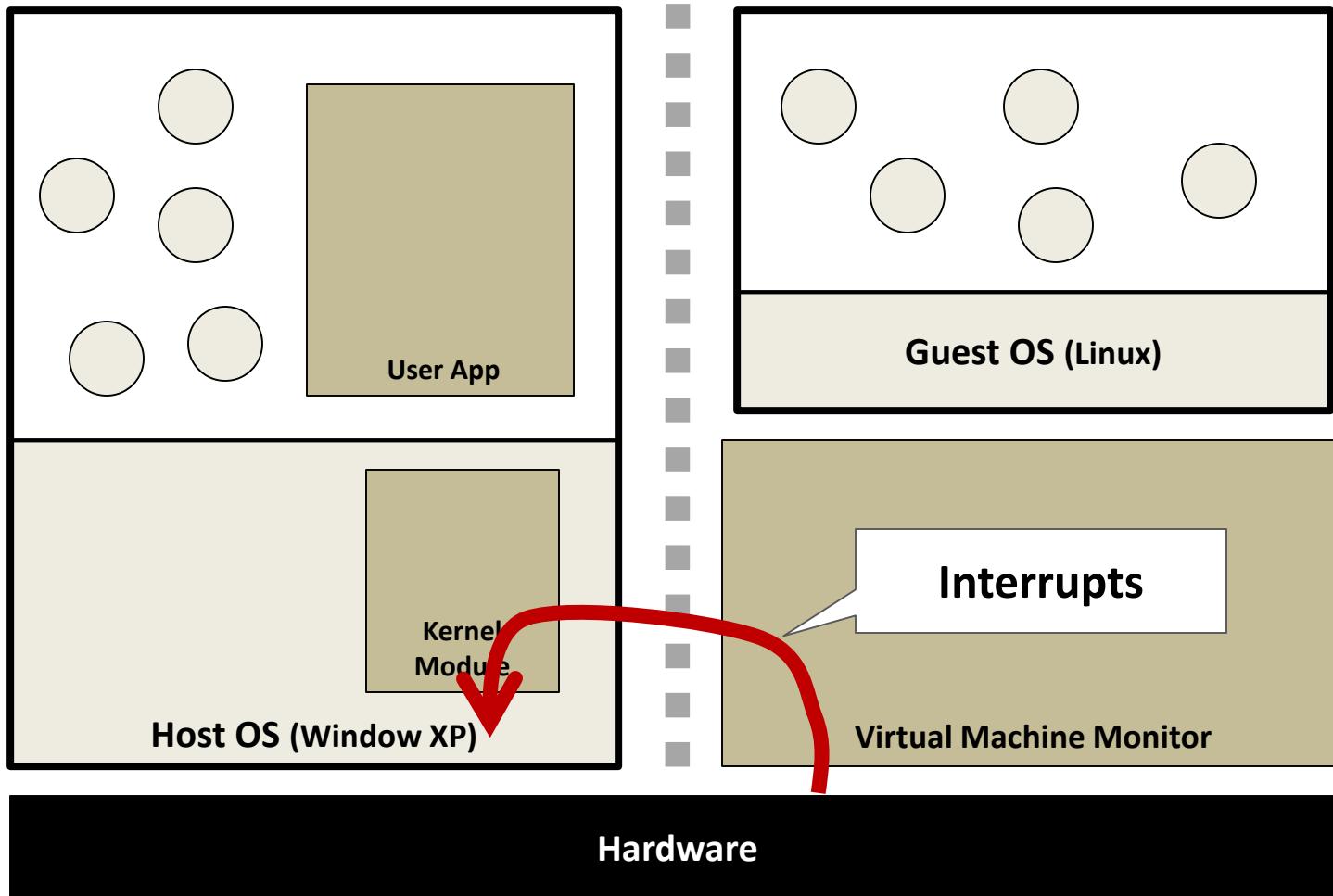
Hosted Monitor Architecture



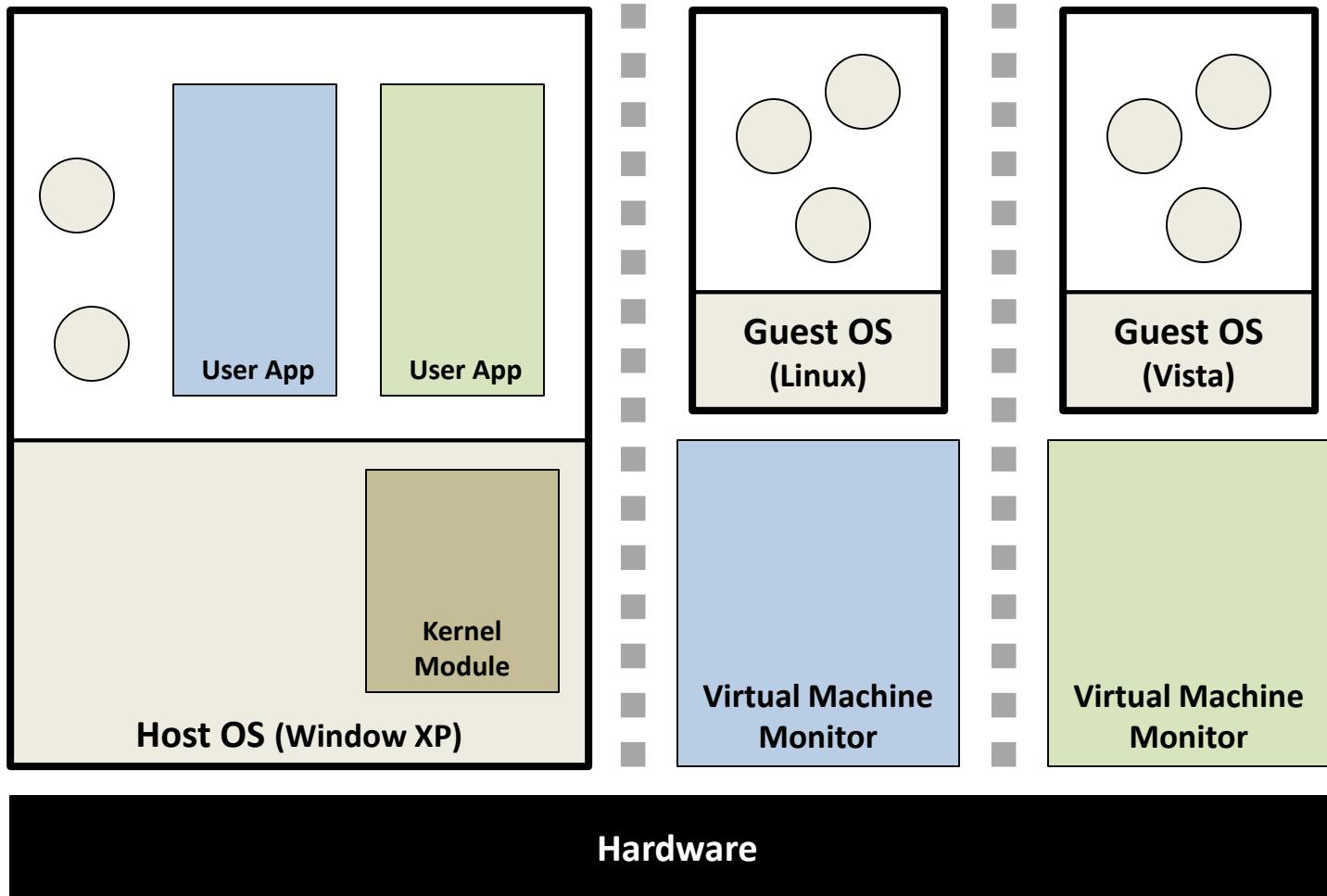
Hosted Monitor Architecture



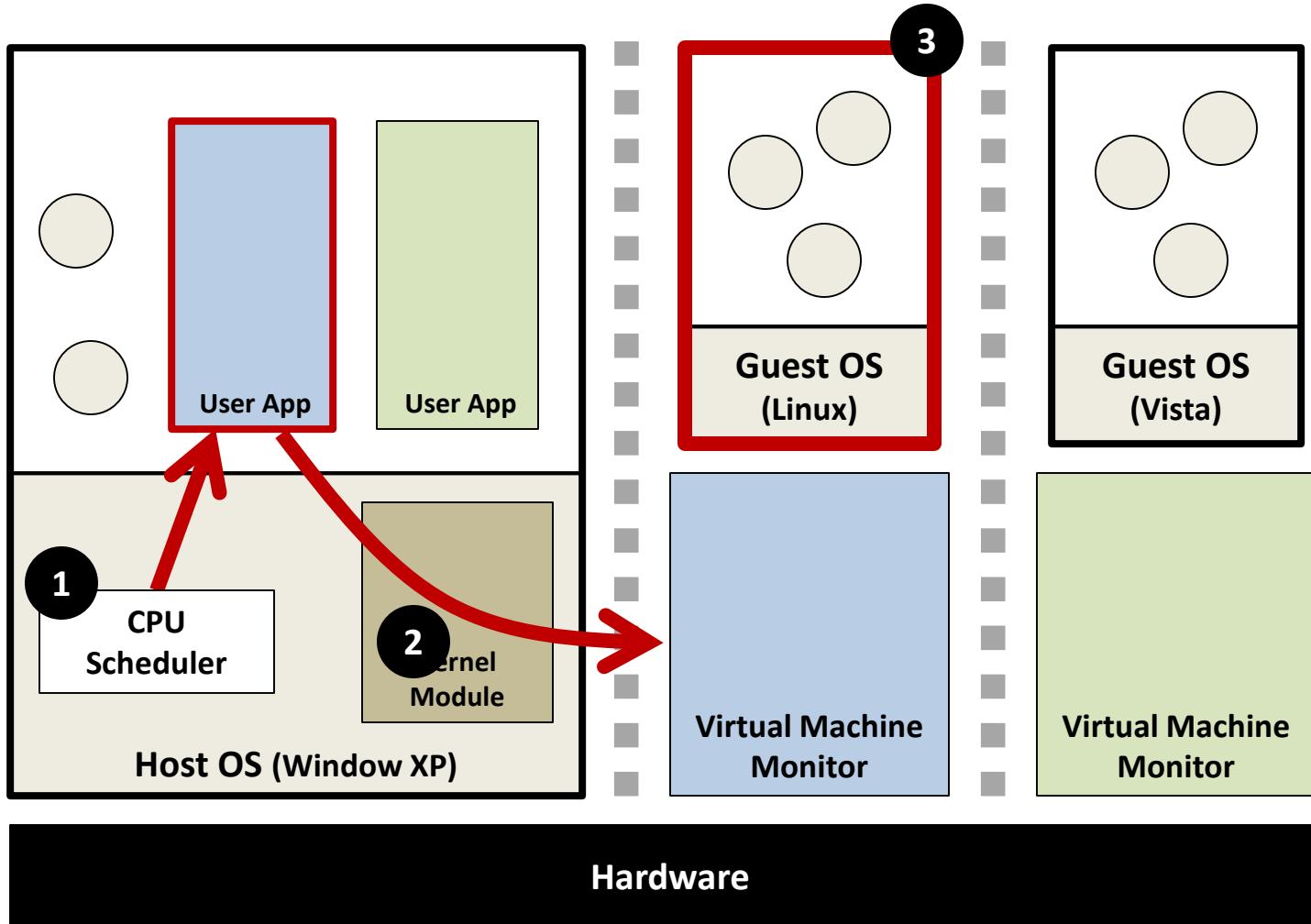
Hosted Monitor Architecture



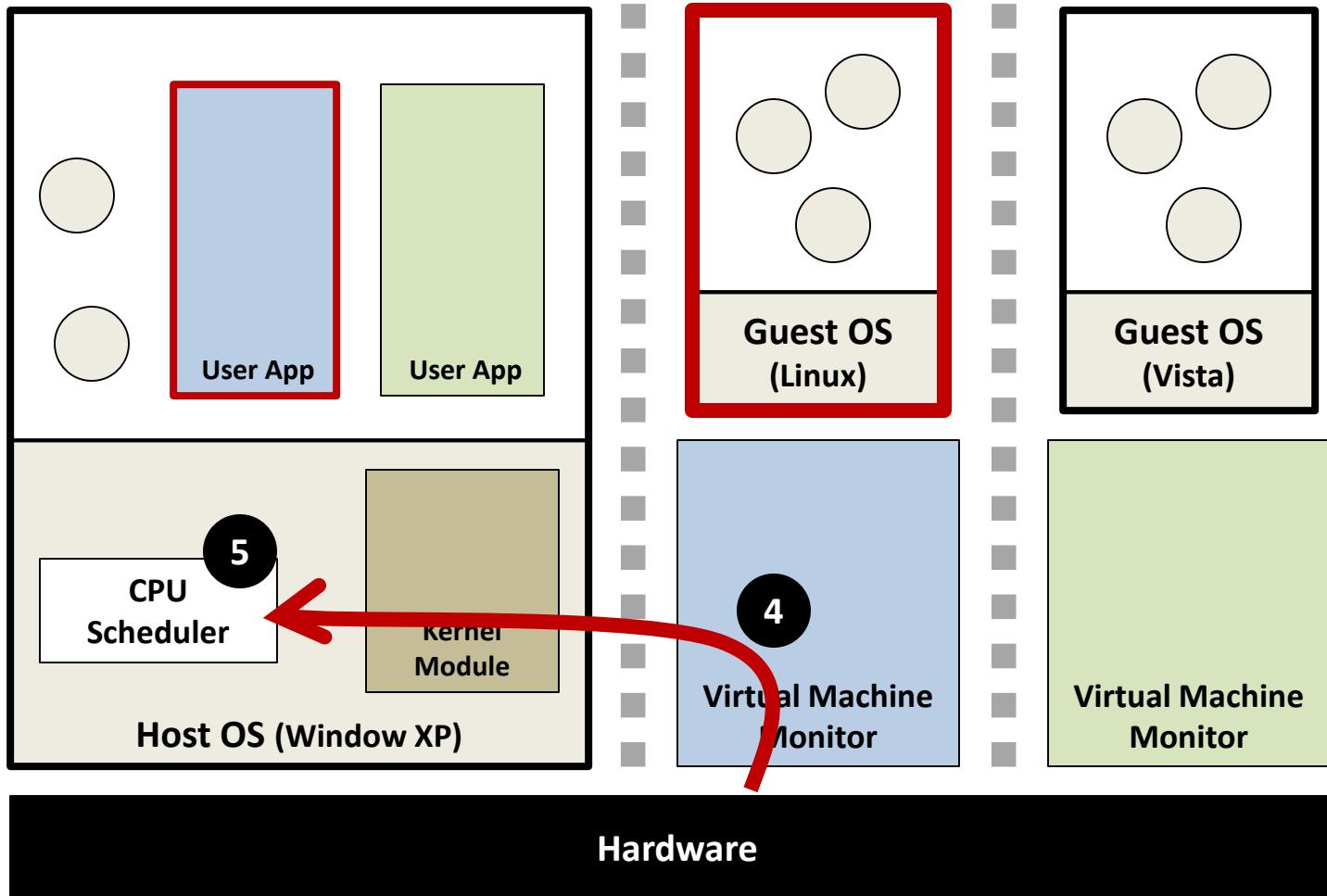
Hosted Monitor Scheduling



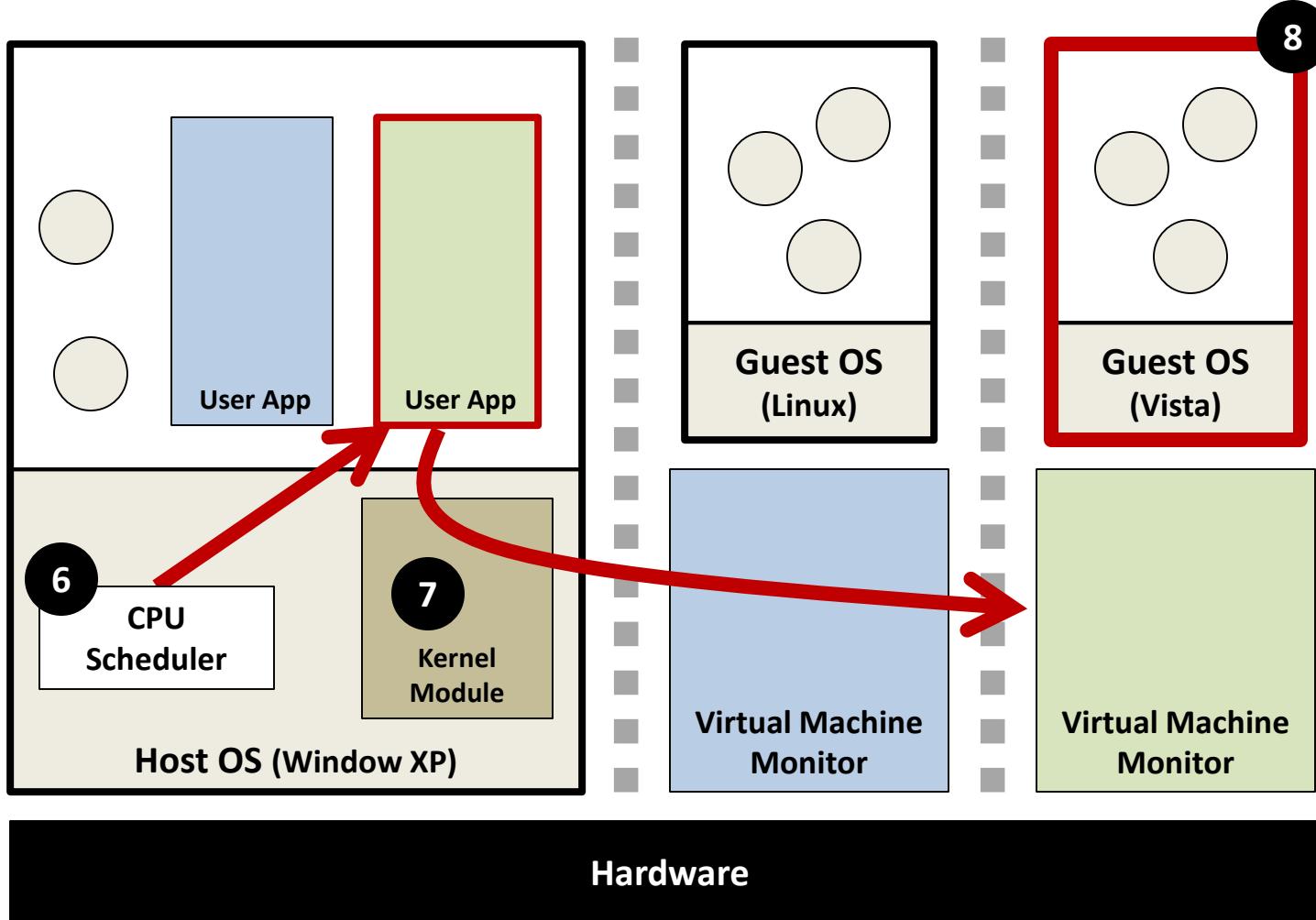
Hosted Monitor Scheduling



Hosted Monitor Scheduling



Hosted Monitor Scheduling



```

Ubuntu - VMware Workstation
File Edit View VM Team Windows Help
bzip2/compress.o
bzip2/decompress.o
bzip2/bzlib.o
bzip2/bzip2.o
Building dot
dot/dot.o
dot/dot
Building printenv
printenv/printenv.o
printenv/printenv
[sbox-SDK_ARMEL: ~/varmosa/arm86] > make clean
[sbox-SDK_ARMEL: ~/varmosa/arm86] > make
Building fib
fib/fib.o
fib/fib
Building bubble
bubble/bubble.o
bubble/bubble
Building bzip2
bzip2/blocksort.o
bzip2/huffman.o
bzip2/crcutable.o
bzip2/randtable.o
bzip2/compress.o
bzip2/decompress.o
bzip2/bzlib.o
bzip2/bzip2.o
Building dot
dot/dot.o
dot/dot
Building printenv
printenv/printenv.o
printenv/printenv
[sbox-SDK_ARMEL: ~/varmosa/arm86] > make clean
[sbox-SDK_ARMEL: ~/varmosa/arm86] > make
Building fib
fib/fib.o
fib/fib
Building bubble
bubble/bubble.o
bubble/bubble

```

devine@devine: ~\$

Windows Task Manager

File Options View Shut Down Help

Applications Processes Performance Networking Users

Show processes from all users

Processes: 117 CPU Usage: 45% Commit Charge: 859M / 3930M

Image Name	User Name	Session ID	CPU	CPU Time	Mem Usage
System Idle Process	SYSTEM	0	55	89:43:07	28 K
vmware-vmx.exe	Scott Devine	0	36	0:01:02	438,092 K
IEXPLORE.EXE	Scott Devine	0	04	0:02:34	37,920 K
taskmgr.exe	Scott Devine	0	02	0:00:06	7,308 K
S24EvMon.exe	SYSTEM	0	02	0:09:35	10,932 K
explorer.exe	Scott Devine	0	01	0:07:49	36,336 K
rundll32.exe	Scott Devine	0	00	0:00:00	4,312 K
qtask.exe	Scott Devine	0	00	0:00:00	3,316 K
ISUSPM.exe	Scott Devine	0	00	0:01:02	4,720 K
cssauth.exe	Scott Devine	0	00	0:00:01	12,188 K
hqtray.exe	Scott Devine	0	00	0:00:00	5,056 K
vmware-tray.exe	Scott Devine	0	00	0:00:00	19,636 K
AwaySch.EXE	Scott Devine	0	00	0:00:00	3,296 K
Hotsync.exe	Scott Devine	0	00	0:00:01	7,088 K
IEXPLORE.EXE	Scott Devine	0	00	0:01:41	7,064 K
FNP Licensing Servi...	SYSTEM	0	00	0:00:02	2,948 K
igfxpers.exe	Scott Devine	0	00	0:00:03	3,976 K
hkcmd.exe	Scott Devine	0	00	0:00:00	4,036 K
igfxtray.exe	Scott Devine	0	00	0:00:00	6,484 K
OUTLOOK.EXE	Scott Devine	0	00	0:00:04	4,172 K
Amsg.exe	Scott Devine	0	00	0:03:30	9,996 K
POWERPNT.EXE	Scott Devine	0	00	0:13:41	49,556 K
BbDevMgr.exe	Scott Devine	0	00	0:00:00	3,988 K
LPMGR.EXE	Scott Devine	0	00	0:00:00	7,860 K
DesktopMgr.exe	Scott Devine	0	00	0:00:01	3,184 K
pdservice.exe	Scott Devine	0	00	0:00:00	4,268 K
ACWLIcon.exe	Scott Devine	0	00	0:00:03	5,192 K
spoolsv.exe	SYSTEM	0	00	0:00:01	6,292 K



Hosted Architecture Tradeoffs

- **Positives**
 - Installs like an application
 - No disk partitioning needed
 - Virtual disk is a file on host file system
 - No host-rebooting needed
 - Runs like an application
 - Uses host's schedulers
- **Negatives**
 - I/O path is slow
 - Requires many world switches
 - Relies on host scheduling
 - May not be suitable for intensive VM workloads

VMware ESX 2.0

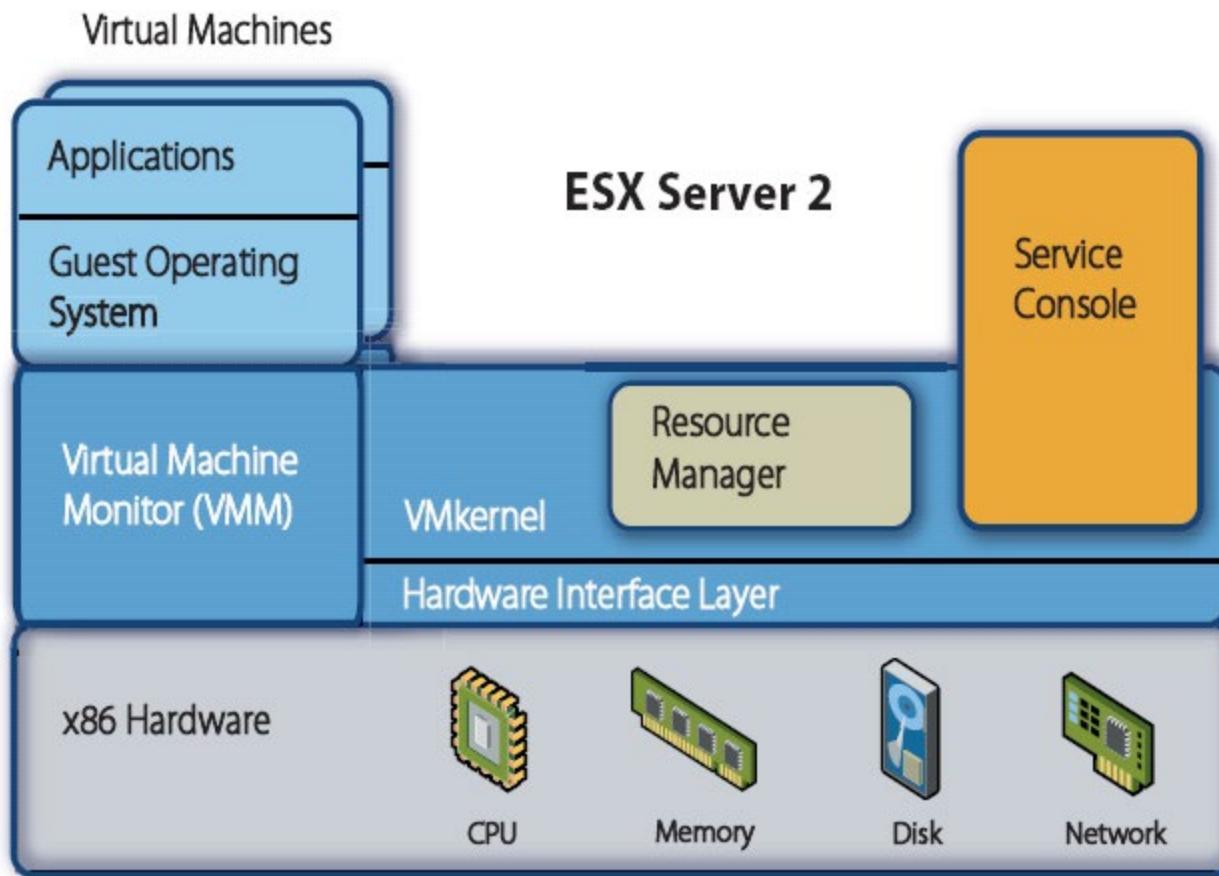
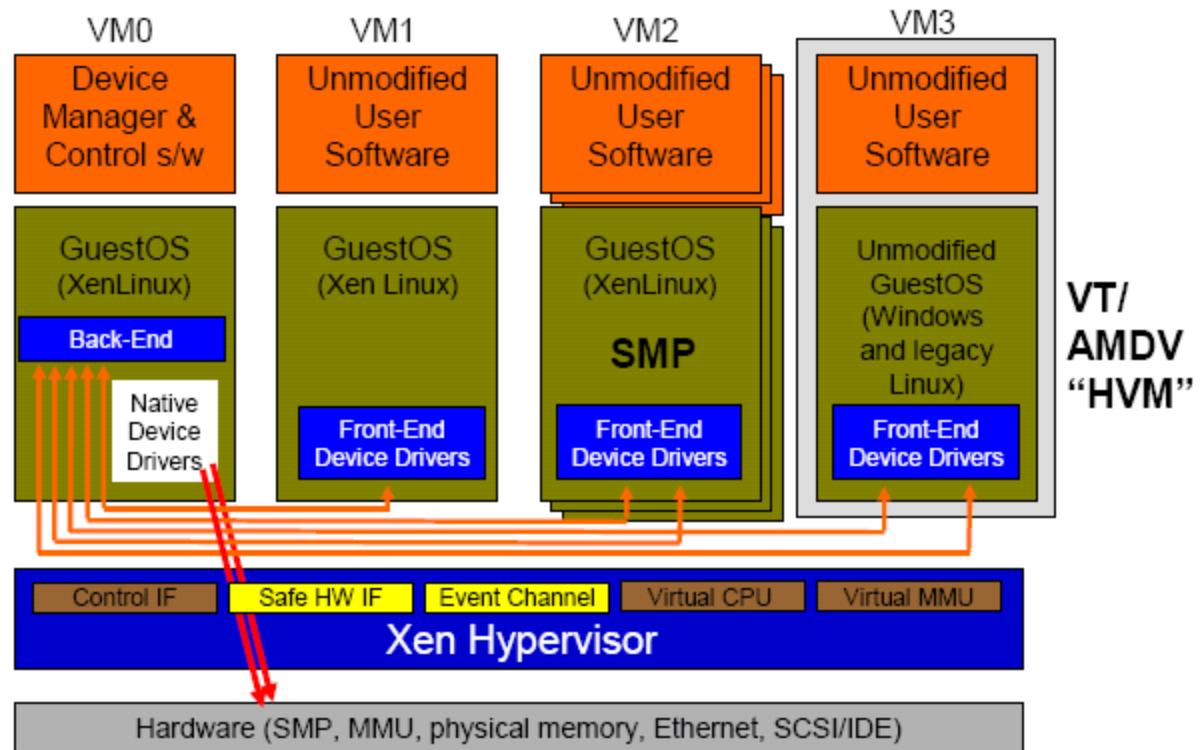


Figure 1: ESX Server architecture

Source: http://www.vmware.com/pdf/esx2_performance_implications.pdf

Hybrid Ex 2 - Xen 3.0

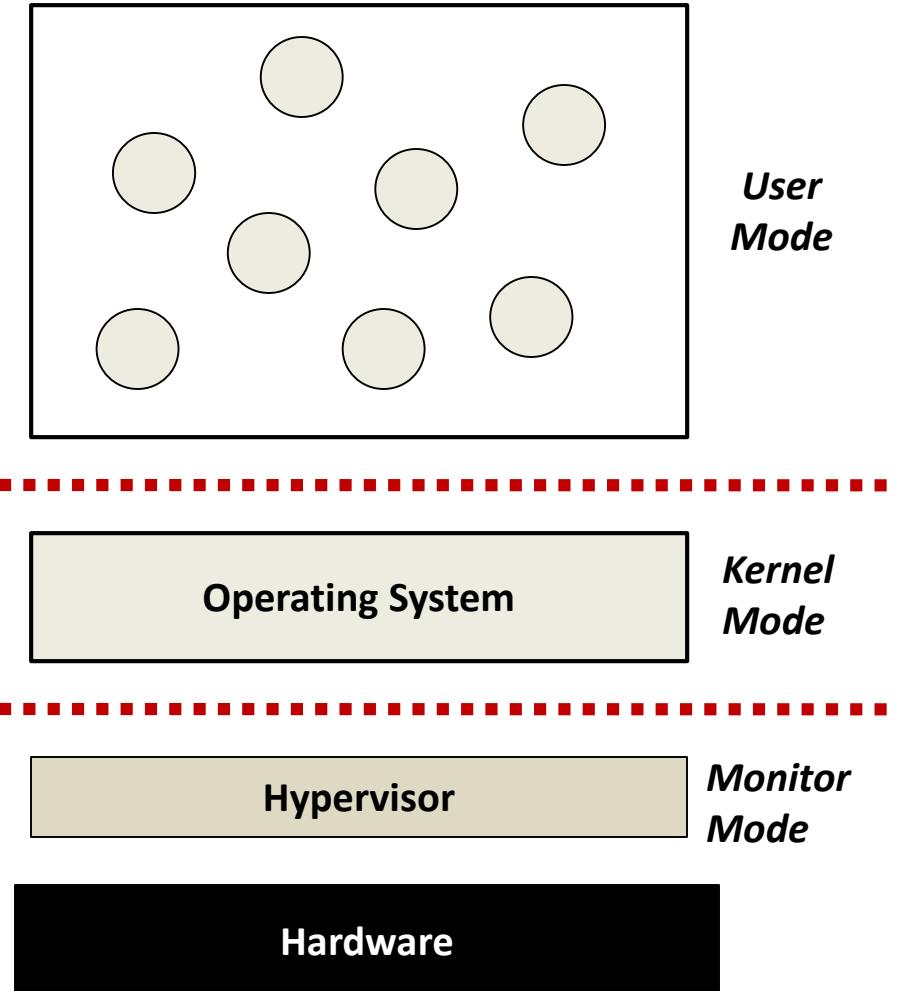
- Para – virtualization
 - Linux Guest
- Hardware-supported virtualization
 - Unmodified Windows
- Isolated Device Drivers



*Source: Ottawa Linux Symposium 2006 presentation.
<http://www.cl.cam.ac.uk/netos/papers/>*

Hypervisor

- Hardware-supported single-use monitor
- Characteristics
 - Small size
 - Runs in a special hardware mode
 - Guest OS runs in normal privileged level
- Uses
 - Security
 - System management
 - Fault tolerance

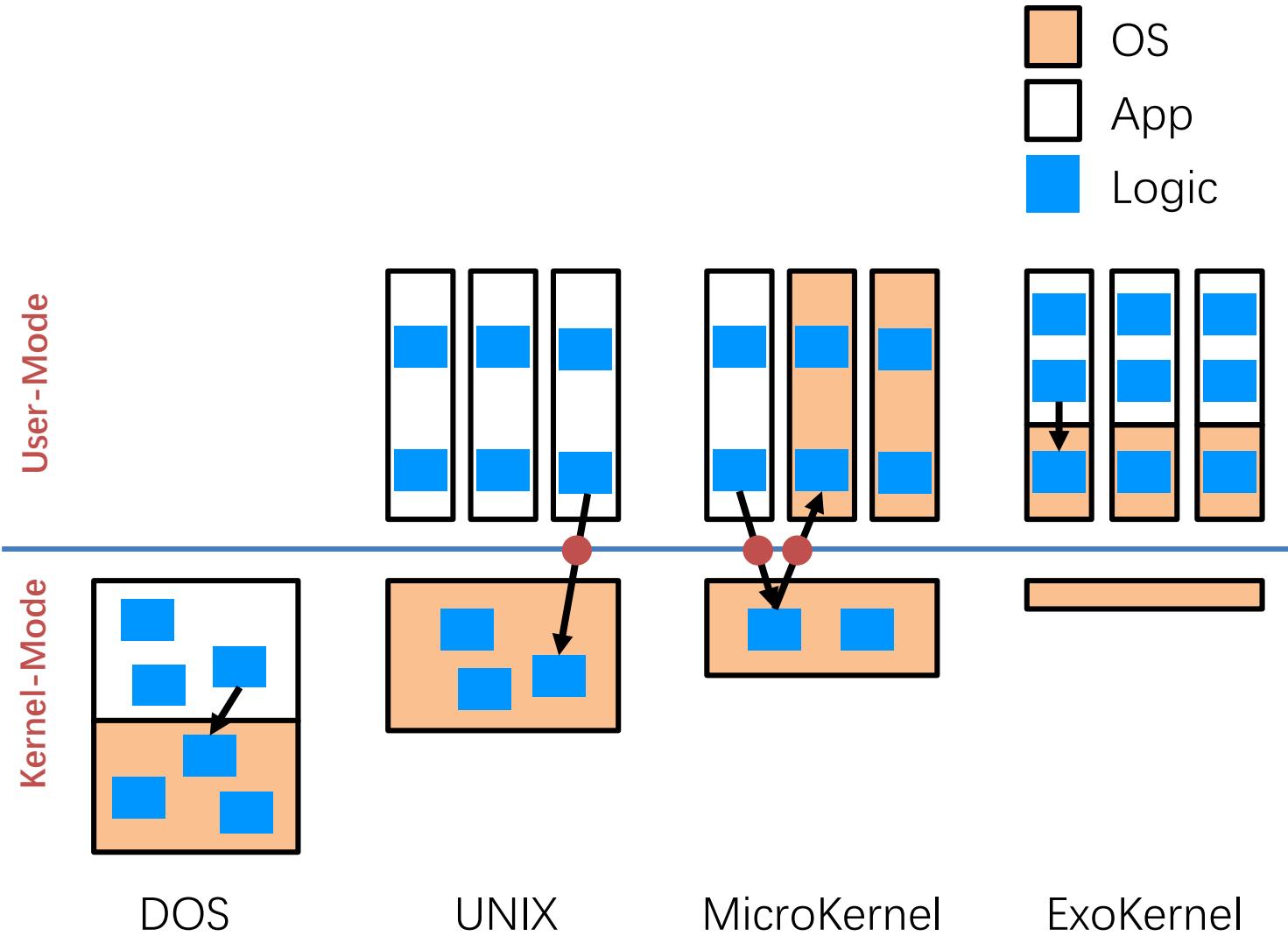


Virtualization: CPU and Memory

Yubin Xia
Software School
Shanghai Jiao Tong University

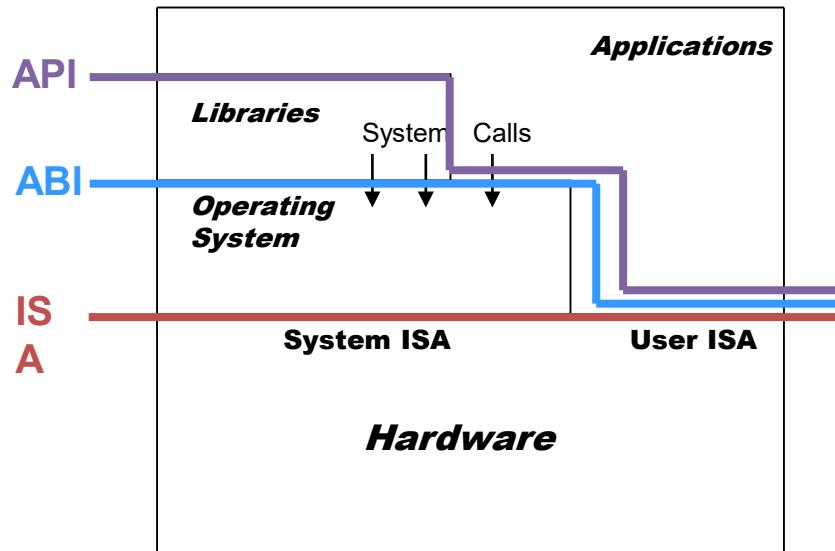
Some Slides adapted from
VMWare's academic course plan

Review: OS Structure



Virtualization Layers

- At which layer?
 - Hello world
 - Web game
 - Dota
 - Office 2013
 - Windows 8
 - Java applications
 - Python scripts
 - High Sierra



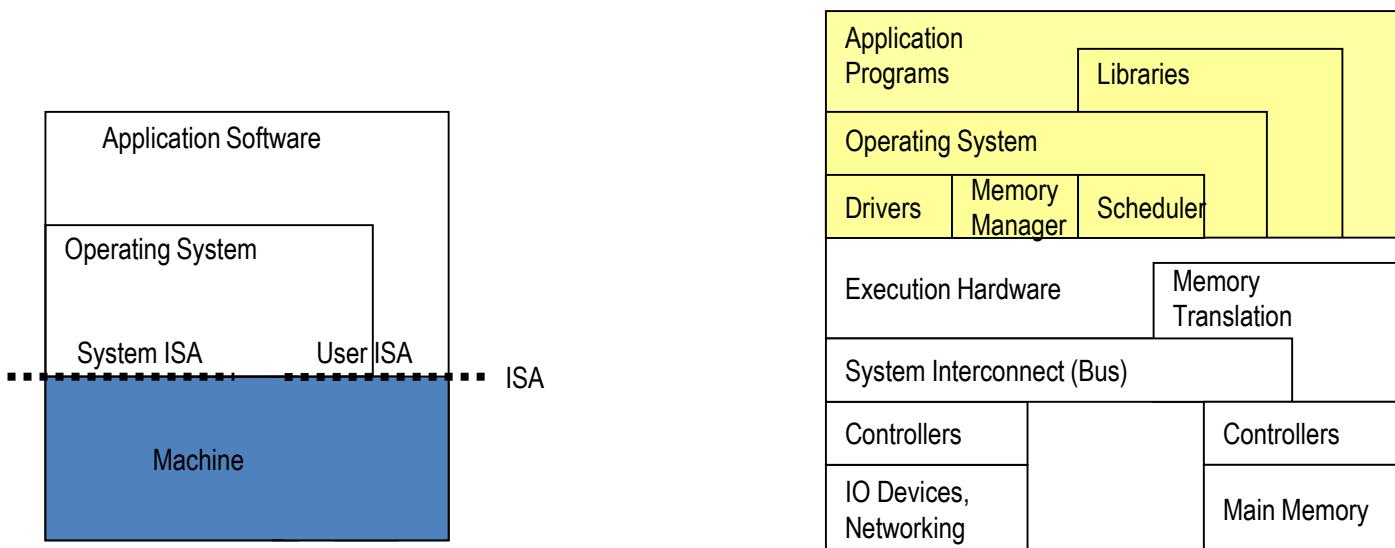
API – application programming interface

ABI – application binary interface

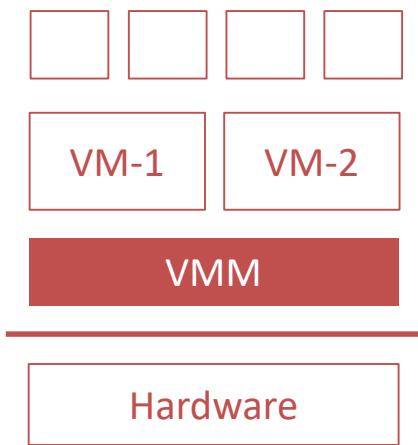
ISA – instruction set architecture

What is a VM and Where is the VM?

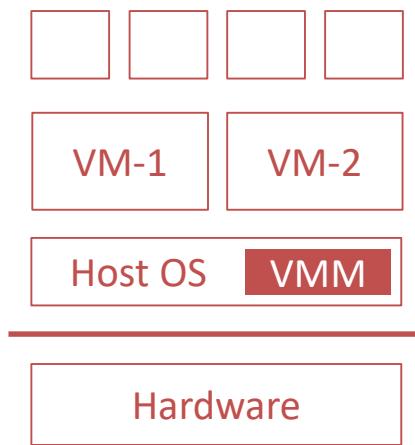
- Machine from the perspective of a system
 - ISA provides interface between system and machine



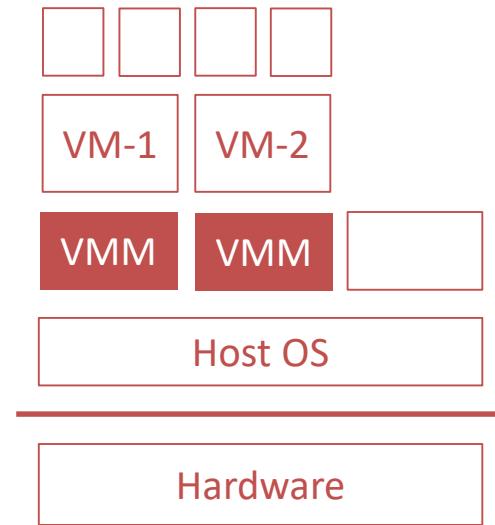
Different Architectures of VMM



Xen



Linux KVM



Qemu

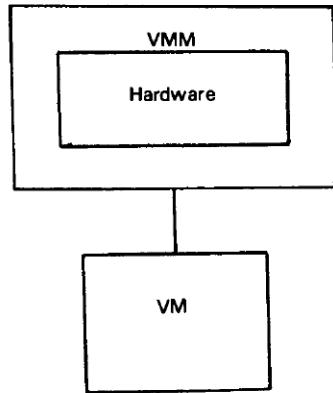
CPU VIRTUALIZATION

Formal Requirement of Virtualization

- 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

Principles in 1974

Fig. 1. The virtual machine monitor.



“an efficient, isolated duplicate of the real machine”

- **Efficiency**
 - Innocuous instructions should execute directly on hardware
- **Resource control**
 - Executed programs may not affect the system resources
- **Equivalence**
 - The behavior of a program executing under the VMM should be the same as if the program were executed directly on the hardware (except possibly for timing and resource availability)

Formal Requirements for Virtualizable Third Generation Architectures

Gerald J. Popek
University of California, Los Angeles
and
Robert P. Goldberg
Honeywell Information Systems and
Harvard University

Virtual machine systems have been implemented on a limited number of third generation computer systems, e.g. CP-67 on the IBM 360/67. From previous empirical studies, it is known that certain third generation computer systems, e.g. the DEC PDP-10, cannot support a virtual machine system. In this paper, model of a third-generation-like computer system is developed. Formal techniques are used to derive precise sufficient conditions to test whether such an architecture can support virtual machines.

Recap: Run OS as an Application

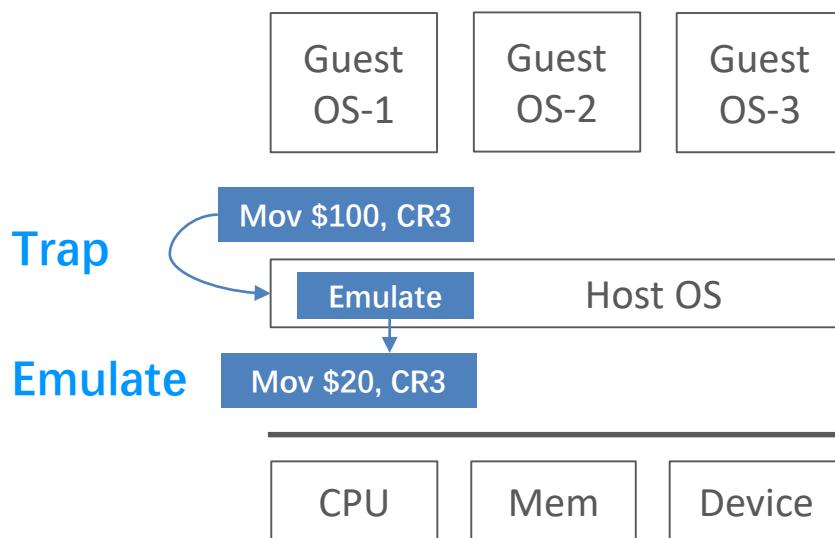
- Stuck at the very first instruction
 - **cli** : disable interrupt
 - It's a privilege instruction
 - **Cannot run in user mode!**
- Similar instructions
 - E.g., **change CR3, set IDT**, etc.
 - These instructions will change machine states
 - Aka., System ISA

```
9  
10 .code16  
11 .globl start  
12 start:  
13 cli  
14
```

xv6: bootasm.S

Solution: Trap & Emulate

- **Trap**: running privilege instructions in user-mode will trap to the VMM
- **Emulate**: those instructions are implemented as *functions* in the VMM
 - System states are kept in VMM's memory, and are changed according



X86 is not Strictly Virtualizable

- Example: the **popf** instruction
 - *popf* takes a word off the stack and puts it into the flags register
 - One flag in that register is the interrupt enable flag (IF)
 - At system level the IF flag is updated by **popf**
 - At user level the IF flag is *not* updated, and CPU silently drops updates to the IF
- There **17** such instructions in X86
 - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

How to Deal with the 17 Instructions?

1. **Instruction Interpretation**: emulate them by software
2. **Binary translation**: translate them to other instructions
3. **Para-virtualization**: replace them in the source code
4. **New hardware**: change the CPU

Sol-1: Instruction Interpretation

- Emulate **Fetch/Decode/Execute** pipeline in software
 - Emulate all the system status using memory
 - E.g., using an array **GPR[8]** for general purpose registers
 - None guest instruction executes directly on hardware
- E.g., Bochs
- Positives
 - Easy to implement & minimal complexity
- Negatives
 - **Very slow!**

Sol-2: Binary Translator

- Translate before execution
 - Translation unit is basic block (why?)
 - Each basic block -> code cache
 - Translate the 17 instructions to function calls
 - Implemented by the VMM
- E.g., VMware, Qemu

Issues with Binary Translation

- PC synchronization on interrupts
 - Now interrupt will only happen at basic block boundary
 - But on real machine, interrupt may happen at any instruction
- Carefully handle self-modifying code (SMC)
 - Notified on writes to translated guest code

Sol-3: Para-virtualization

- Modify OS and let it cooperate with the VMM
 - Change sensitive instructions to calls to the VMM
 - Also known as **hypervisor call**
 - Hypervisor call can be seen as trap
- E.g., Xen
 - Widely used by industry like Amazon's EC2

Sol-4: Hardware Supported CPU Virtualization

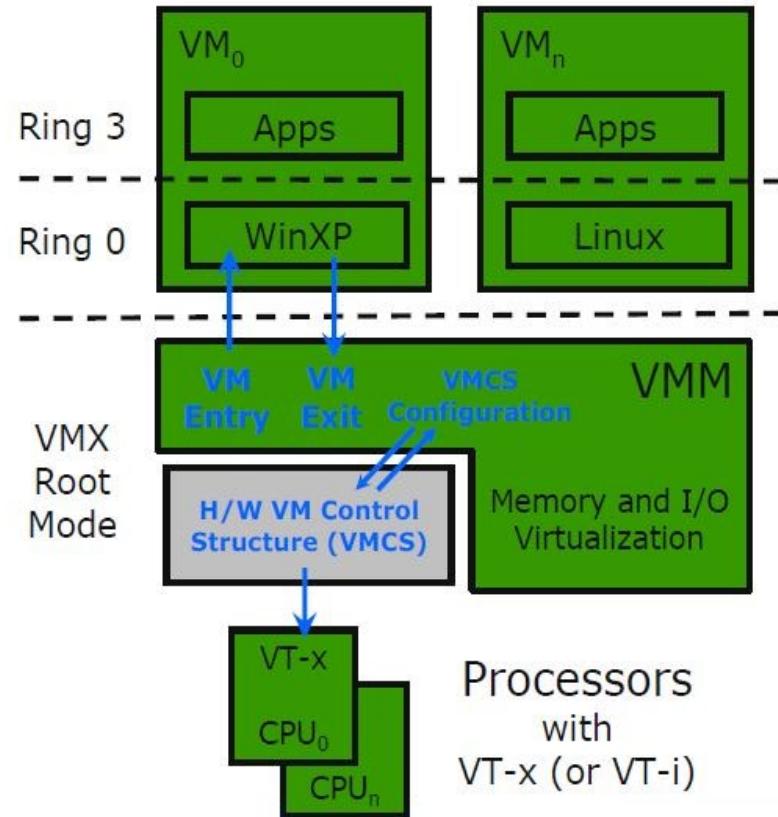
- VMX **root** operation:
 - Full privileged, intended for Virtual Machine Monitor
- VMX **non-root** operation:
 - Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

CPU Virtualization with VT-x

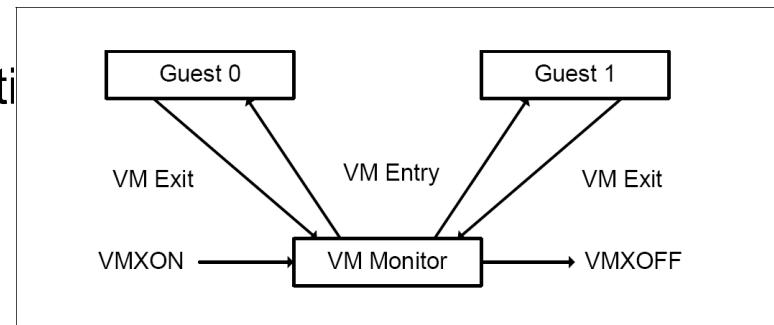
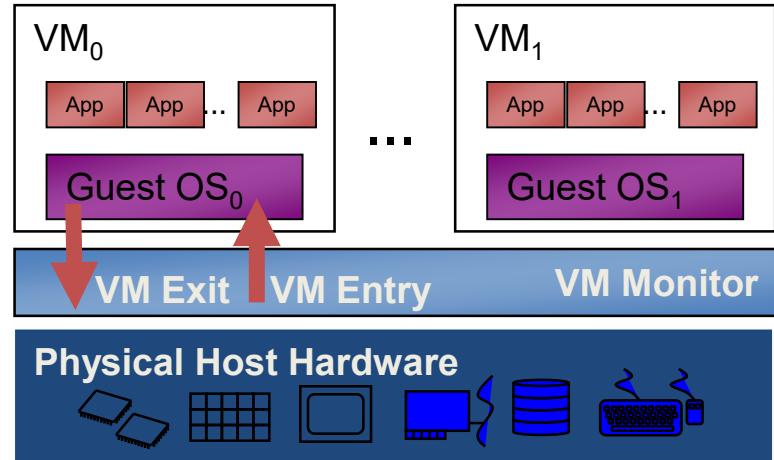
Guest OSes run at intended rings

- Two new mode:
 - Root and non-root
- VMX root operation:
 - Fully privileged, intended for VM monitor
- VMX non-root operation:
 - Not fully privileged, intended for guest software
 - Reduces Guest SW privilege w/o relying on rings
 - Solution to Ring Aliasing and Ring Compression



VM Entry and VM Exit

- VM Entry
 - Transition from VMM to Guest
 - Enters VMX non-root operation
 - Loads Guest state from VMCS
 - **VMLAUNCH** used on initial entry
 - **VMRESUME** used on subsequent entries
- VM Exit
 - **VMEXIT** instruction used on transition from Guest to VMM
 - Enters VMX root operation
 - Saves Guest state in VMCS
 - Loads VMM state from VMCS



Virtual Machine Control Structure (VMCS)

Data structure that manages VM entries and VM exits.

VM entries load processor state from the guest-state area.

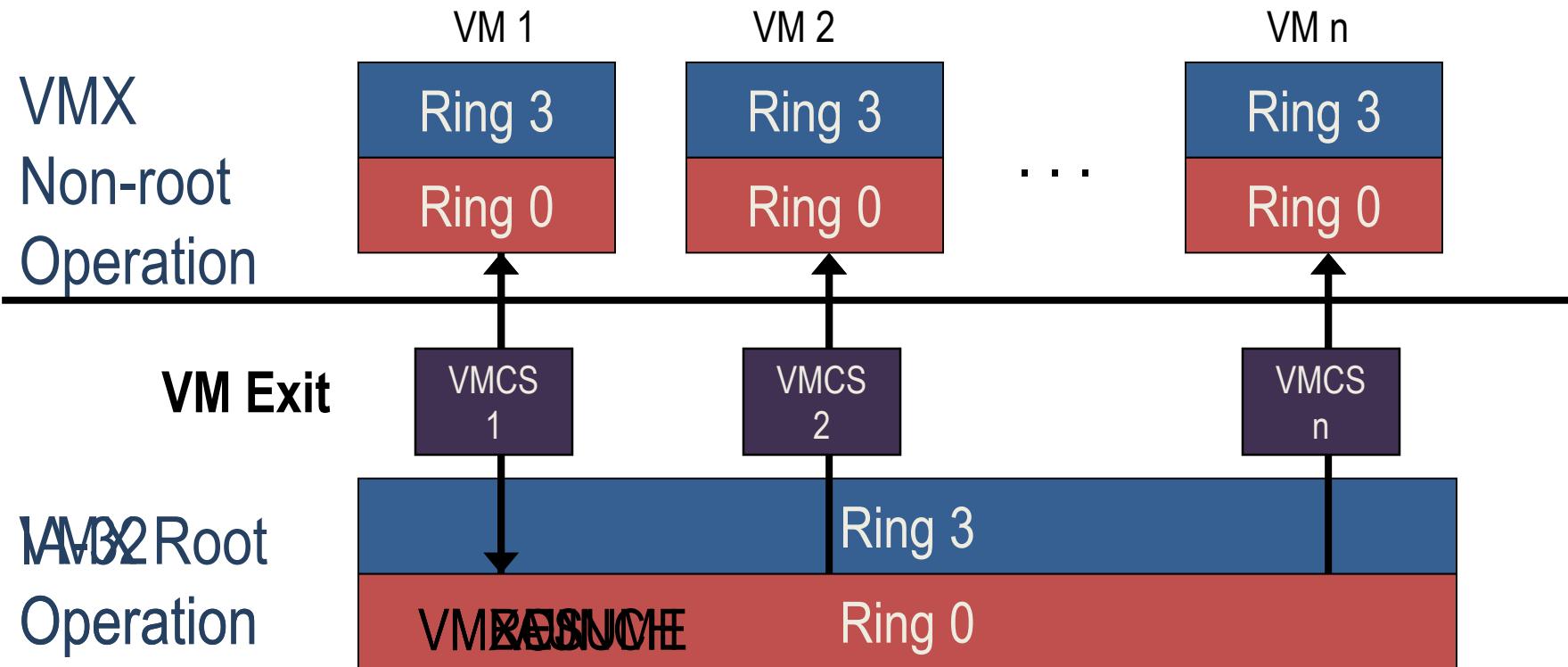
VM exits save processor state to the guest-state area and the exit reason, and then load processor state from the host-state area.

VMCS: VM Control Structure

The VMCS consists of six logical groups:

- **Guest-state area:** Processor state saved into the guest-state area on VM exits and loaded on VM entries.
- **Host-state area:** Processor state loaded from the host-state area on VM exits.
- **VM-execution control fields:** Fields controlling processor operation in VMX non-root operation.
- **VM-exit control fields:** Fields that control VM exits.
- **VM-entry control fields:** Fields that control VM entries.
- **VM-exit information fields:** Read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

VT-x Operations



VT-x New instructions

VMXON and VMXOFF

To enter and exit VMX-root mode.

VMLAUNCH: Used on initial transition from VMM to Guest

Enters VMX non-root operation mode

VMRESUME: Used on subsequent entries

Enters VMX non-root operation mode

Loads Guest state and Exit criteria from VMCS

VMEXIT

Used on transition from Guest to VMM

Enters VMX root operation mode

Saves Guest state in VMCS

Loads VMM state from VMCS

VMPTRST and VMPTRLD

To Read and Write the VMCS pointer.

VMREAD, VMWRITE, VMCLEAR

Read from, Write to and clear a VMCS.

MEMORY VIRTUALIZATION

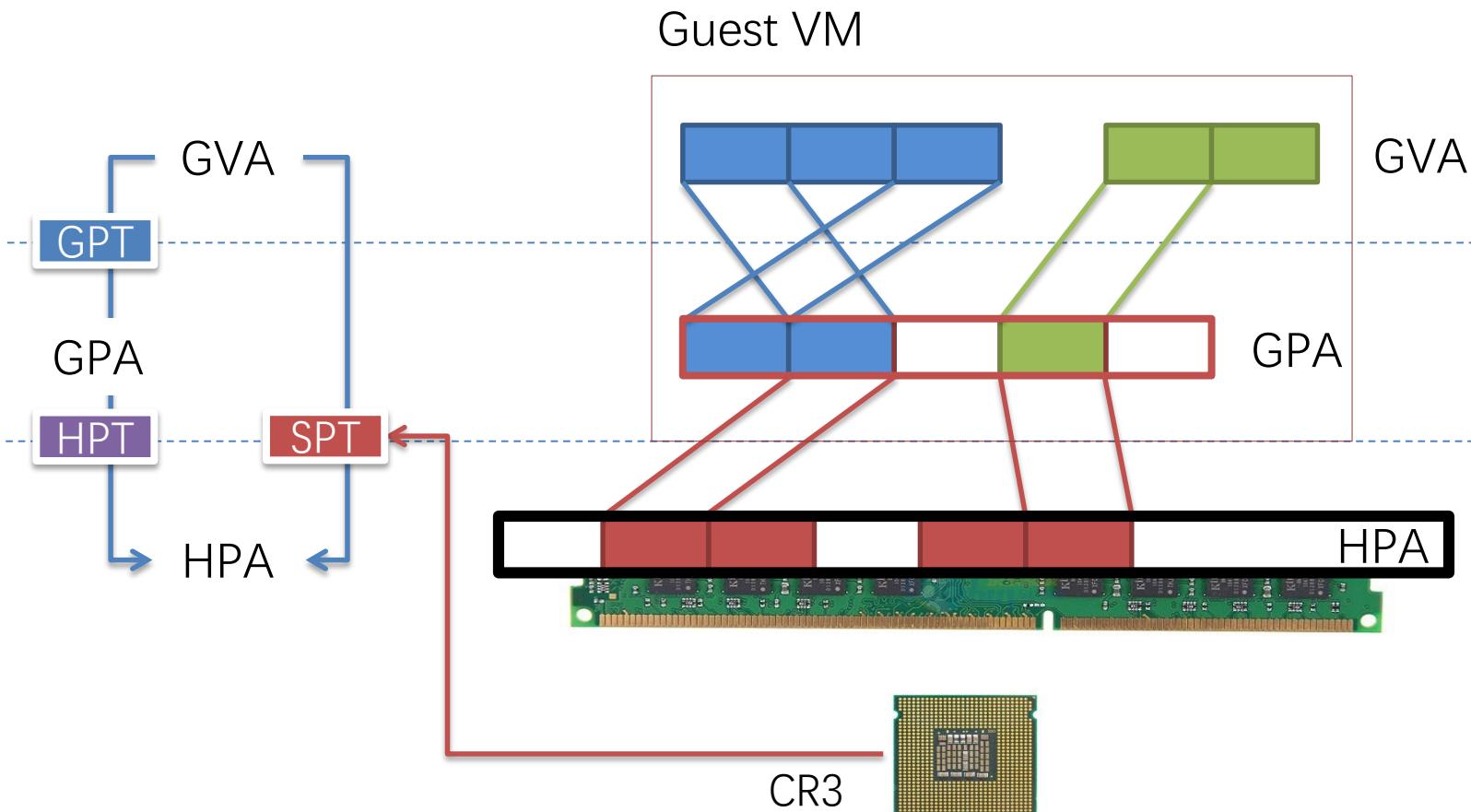
Virtualizing Memory

- VMM constructs a page table that maps guest address to host physical address
 - E.g., if guest VM has 1GB of memory, it can access memory address 0~1GB
 - Each guest VM has its own mapping for memory address 0, etc.
 - Different host physical address used to store data for those memory locations

Virtualizing the Page Tables

- Terminology: 3 types of address now
 - **GVA->GPA->HPA** (Guest virtual. Guest physical. Host physical)
 - Guest VM's page table contains GPA
- Setting CR3 to point to guest page table would not work
 - E.g., a processes in VM might access host physical address 0~1GB, which might not belong to that guest VM
 - Solution-1: **shadow paging**
 - Solution-2: **direct paging**
 - Solution-3: **new hardware**

Solution-1: Shadow Pages



Two Page Tables Become One

1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

Setup Shadow Page Table

```
set_cr3 (guest_page_table):  
    for GVA in 0 to 220  
        if guest_page_table[GVA] & PTE_P:  
            GPA = guest_page_table[GVA] >> 12  
            HPA = host_page_table[GPA] >> 12  
            shadow_page_table[GVA] = (HPA<<12) | PTE_P  
        else  
            shadow_page_table[GVA] = 0  
    CR3 = PHYSICAL_ADDR(shadow_page_table)
```

What if Guest OS Modifies Its Page Table?

- Real hardware would start using the new page table's mappings
 - Virtual machine monitor has a separate shadow page table
- Goal:
 - VMM needs to intercept when guest OS modifies page table, update shadow page table accordingly
- Technique:
 - Use the read/write bit in the PTE to mark those pages read-only
 - If guest OS tries to modify them, hardware triggers page fault
 - Page fault handled by VMM: update shadow page table & restart guest

Protect Kernel-only Pages

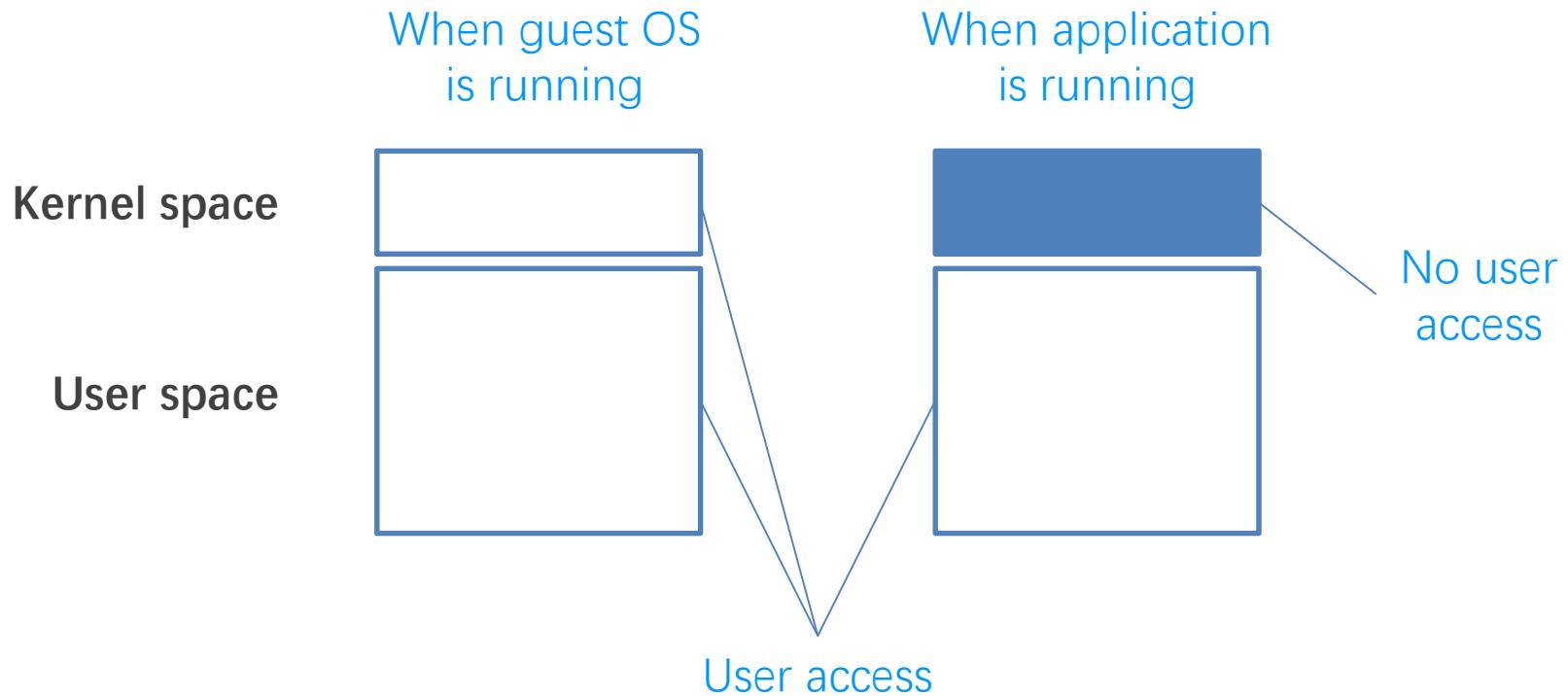
- How do we selectively allow / deny access to kernel-only pages in guest PT?
 - Hardware doesn't know about our virtual U/K bit
- Idea:
 - Generate **two** shadow page tables, one for U, one for K
 - When guest OS switches to U mode, VMM must invoke `set_ptp(current, 0)`

Protect Kernel-only Pages



```
set_ptp(guest_pt, kmode):
    for gva in 0 .. 220:
        if guest_pt[gva] & PTE_P and
            (kmode or guest_pt[gva] & PTE_U):
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P | PTE_U
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt
```

Two Memory Views of Guest VM



Sol-2: Direct Paging (Para-virtualization)

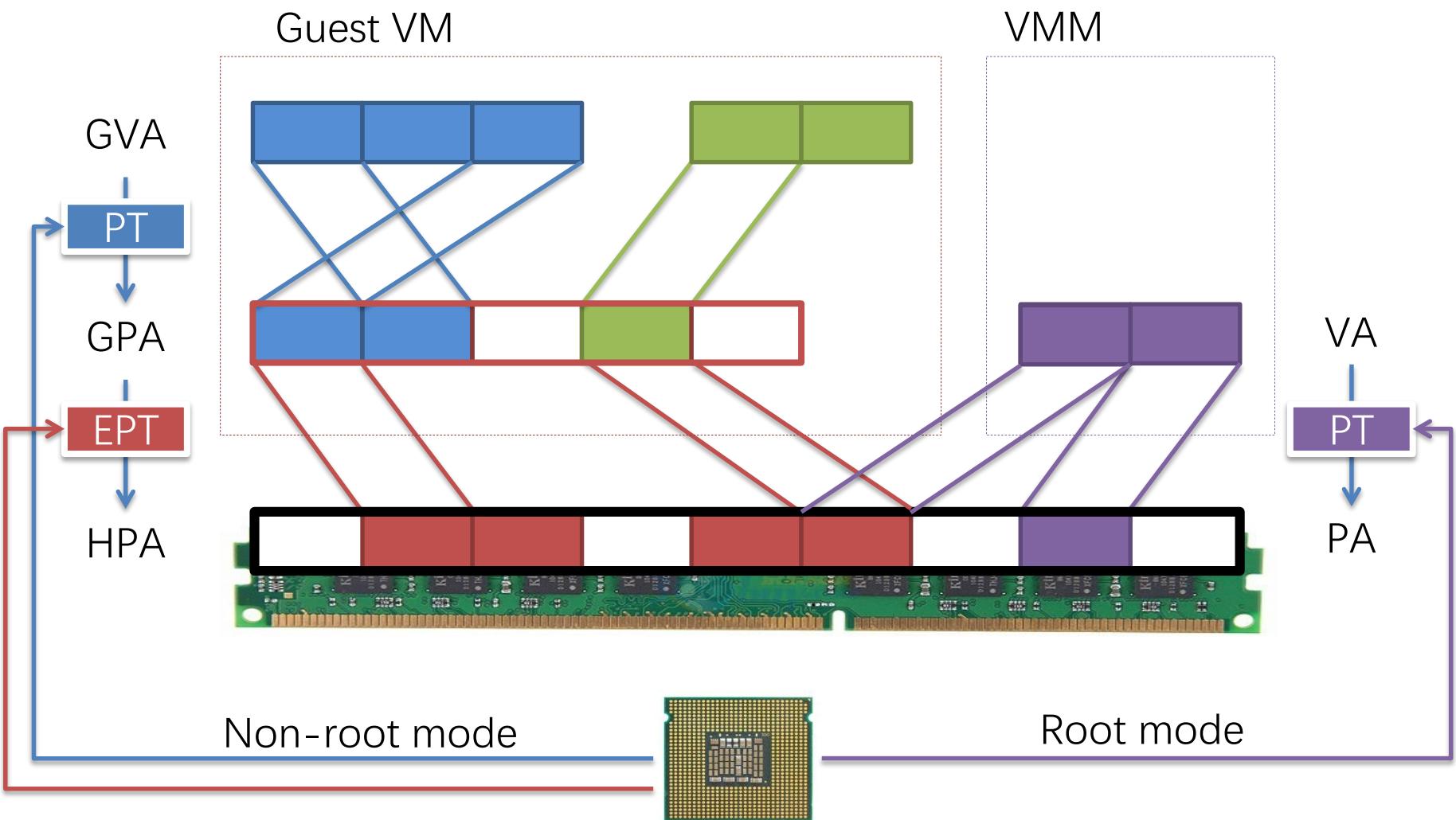
- Modify the guest OS
 - No GPA is needed, just GVA and HPA
 - Guest OS directly manages its HPA space
 - Use hypercall to let the VMM update the page table
 - The hardware CR3 will point to guest page table
- VMM will check all the page table operations
 - The guest page tables are read-only to the guest

Sol-2: Direct Paging (Para-virtualization)

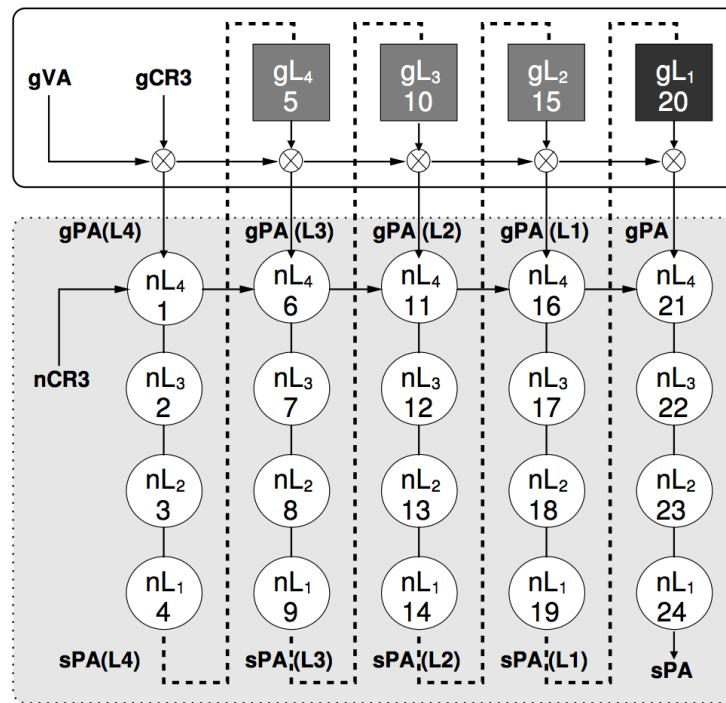
- Positive
 - Easy to implement and more clear architecture
 - Better performance: guest can batch to reduce trap
- Negatives
 - Not transparent to the guest OS
 - The guest now knows much info, e.g., HPA
 - May use such info to trigger *rowhammer* attacks

Sol-3: Hardware Supported Memory Virtualization

- Hardware implementation
 - Intel's EPT (Extended Page Table)
 - AMD's NPT (Nested Page Table)
- Another table
 - EPT for translation from **GPA to HPA**
 - EPT is controlled by the hypervisor
 - EPT is per-VM



EPT Increases Memory Access



One memory access from the guest VM may lead up to **20 memory accesses!**

CASE STUDY: VMWARE

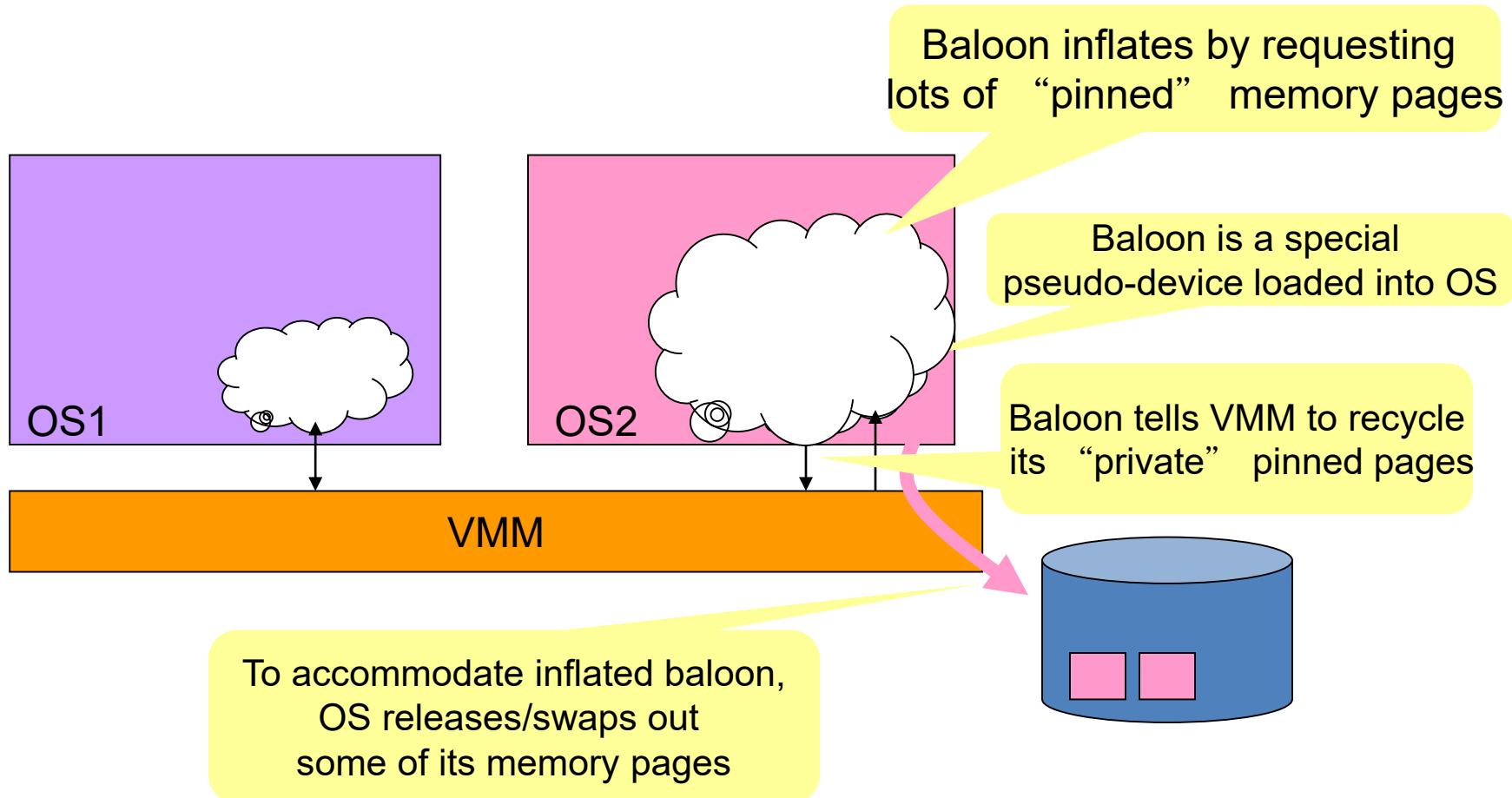
Managing Memory in VMM

- Configure VMs to use more memory than actually available
- What happens when running out of memory?
- Strawman: use LRU paging at VMM
 - OS already uses LRU → doubling paging
 - OS will recycle whatever page VMM just paged out
 - Better to do random eviction

Vmware ESX: Reclaiming Pages

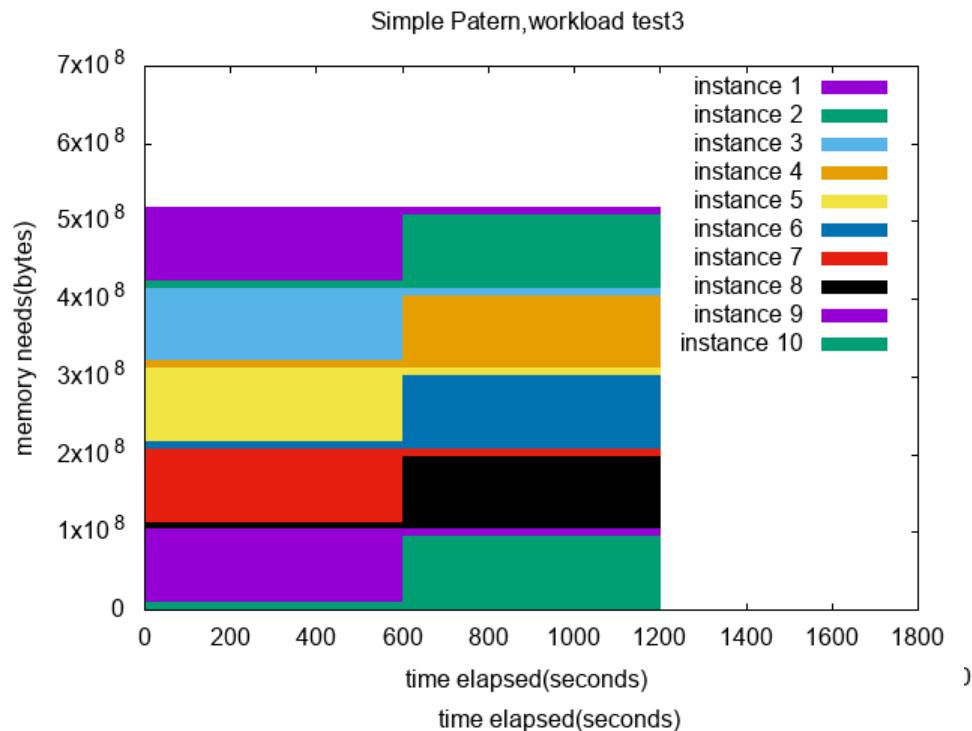
- Idea: trick OS to return memory to VMM
- OS is better at deciding what to swap
 - Normally OS uses all available memory
 - E.g. buffer cache contains old pages, OS won't discard if it does not need memory
- ESX trick: **balloon** driver

Memory Ballooning

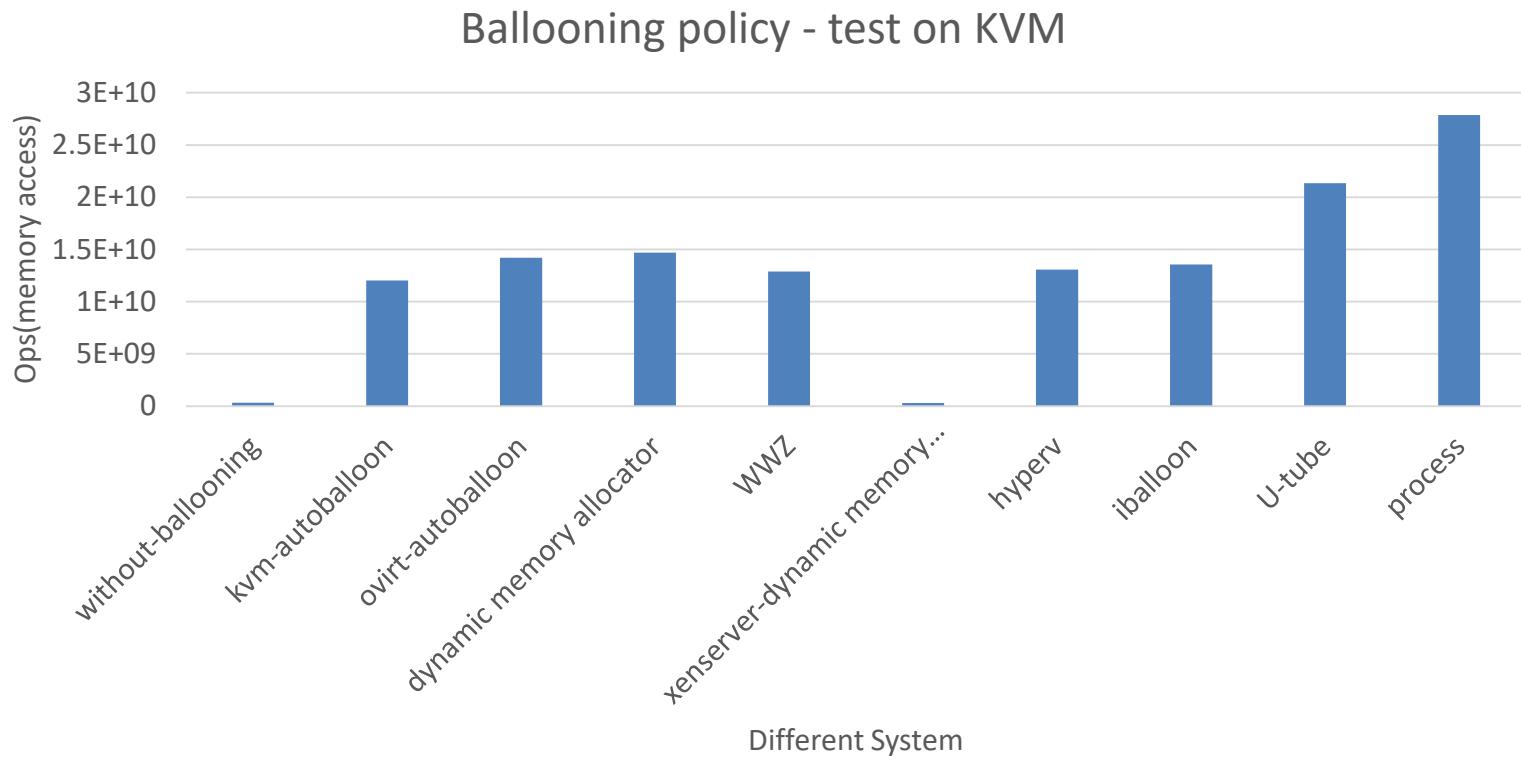


A simple pattern evaluation

- 10 small VMs configured with initial memory 90MB, maxmem 256MB.
 - Three tests with three pattern change intervals.



Ballooning policy analysis



ESX: Sharing Pages across VMs

- Many VMs run same OS and programs
 - Many Linux boxes with Apache server
- Idea: use 1 machine page for identical physical pages
- Periodically scan to find identical machine pages
 - Do copy-on-write to eliminate redundancy
- Optimization: use a hash table keyed by hash(content)
 - Allows quick lookup based on page content

Further Reading

- www.govirtual.com
- 系统虚拟化--原理与实现
 - <http://www.china-pub.com/45151&ref=xiangguan>

Virtualization: I/O

Yubin Xia
Software School
Shanghai Jiao Tong University

Some Slides adapted from
VMWare's academic course plan

Review: CPU Virtualization

- Trap and Emulation
- Instruction Interpretation
- Binary Translation
- Para-virtualization
- New Hardware (to fix the 17 instructions)

CASE STUDY: KVM & QEMU

KVM Introduction

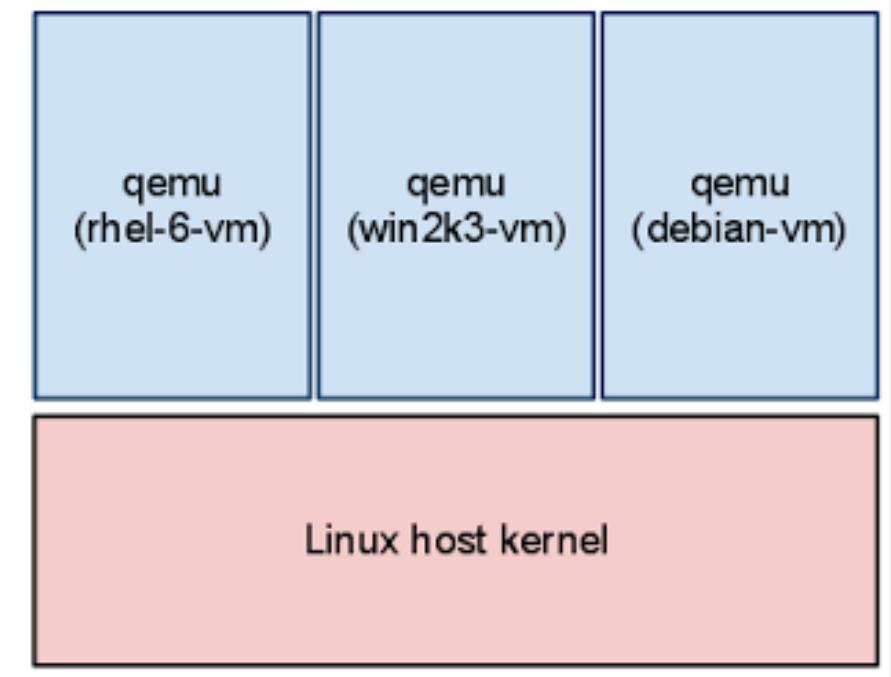
- KVM runs as a Linux module (`/dev/kvm`)
 - Leverage Linux functions
 - E.g., memory management, process scheduling
- Leverage hardware virtualization
 - Trap and emulate: for all the privileged instructions
 - Leverage EPT for address translation between GVA, GPA, HPA
- I/O virtualization
 - Leverage Qemu for convenience
 - Leverage SR-IOV for performance
 - Virtio for para-virtualization

KVM History

- Began development at Qumranet, bought by RedHat later 2008
- Merged to kernel in Feb 2007
- Also be ported to FreeBSD
- Now support x86, S/390, PowerPC, IA-64, ARM
- Support a lot of guest OSes without modification
- VirtIO: a paravirtualization device driver now work for Linux, *BSD, Plan 9 and Windows

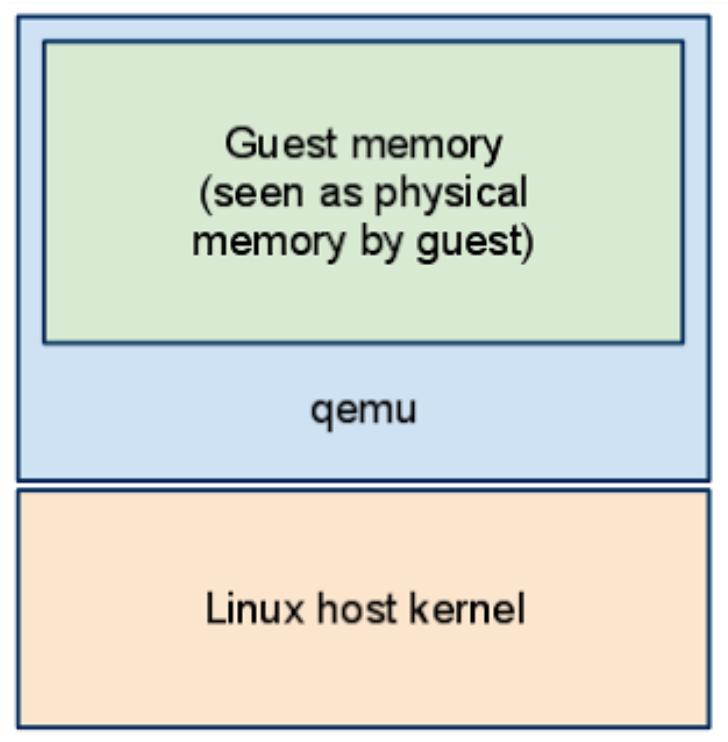
In KVM, VM is Just a Qemu Process

- Create VMs by running the *qemu* program
 - Aka., *qemu-kvm* or *kvm*
 - On a host that is running 3 VMs there are 3 *qemu* processes
- VM shut down, *Qemu* exits
 - Reboot without restart *qemu*



Allocate Memory as Guest Physical Memory

- Qemu allocates memory
- The memory acts as the "physical" memory as seen by the guest



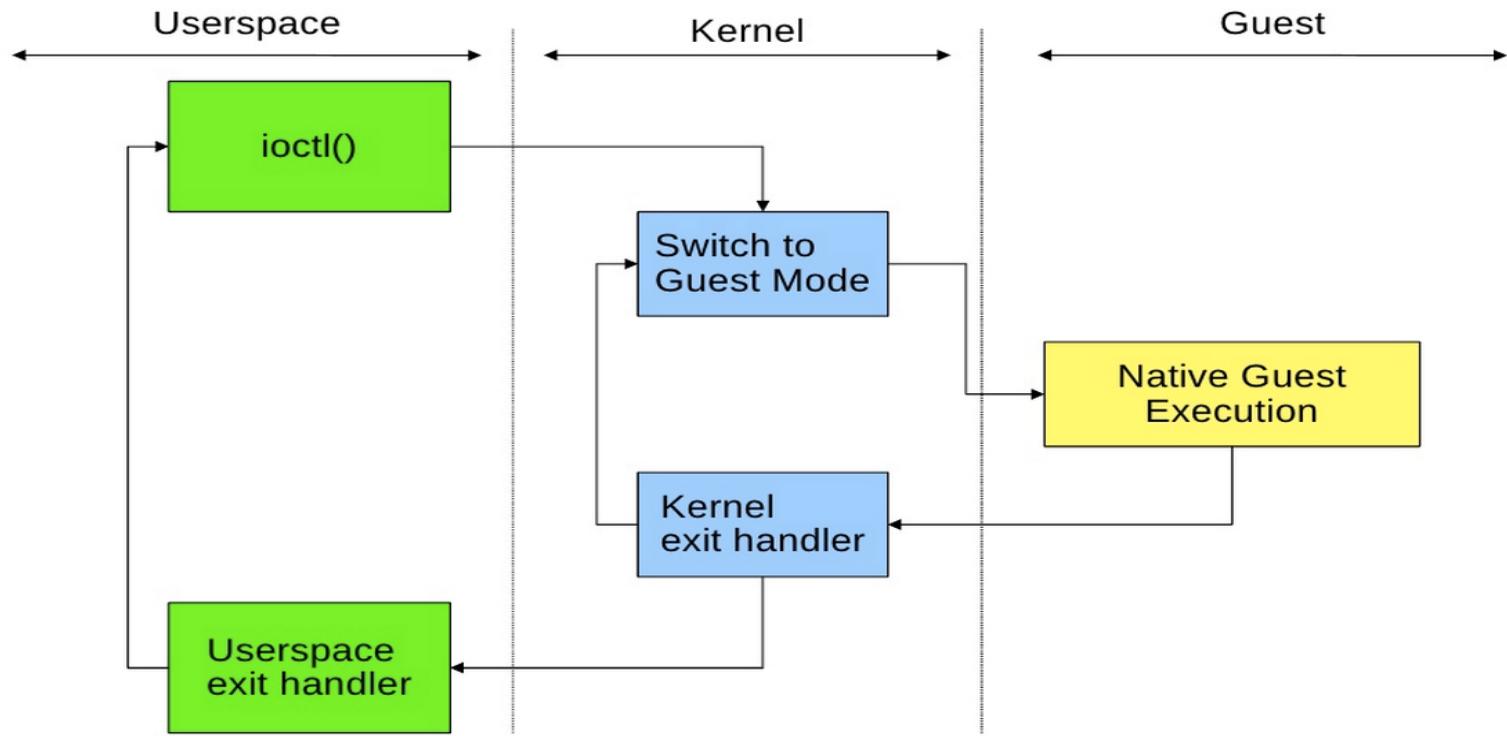
KVM Basic Flow

- Use /dev/kvm to operate hardware virtualization
 - ioctl with different parameters
 - CREATE_VM, CREATE_VCPU, RUN, etc.

Qemu-kvm Basic Flow

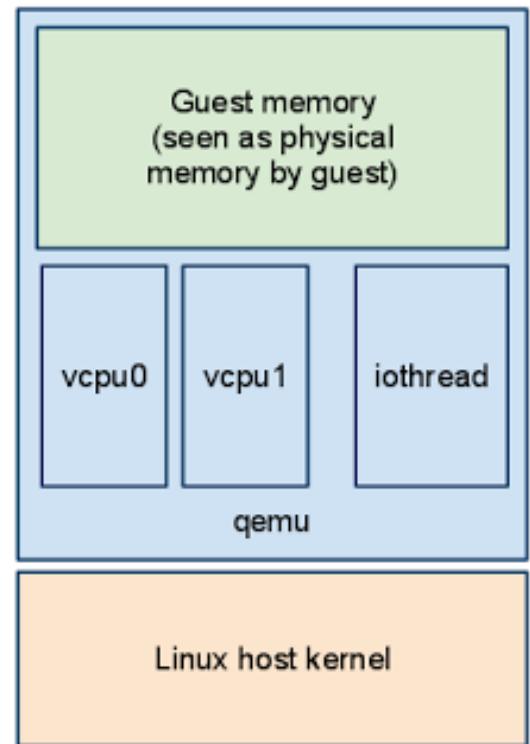
```
open ("/dev/kvm")
ioctl (KVM_CREATE_VM)
ioctl (KVM_CREATE_VCPU)
for (;;) {
    ioctl (KVM_RUN)           Invoke VMENTRY
    exit_reason = get_exit_reason();
    switch (exit_reason) {
        case KVM_EXIT_IO: /* ... */
            break;
        case KVM_EXIT_HLT: /* ... */
            break;
    }
}
```

KVM Execution



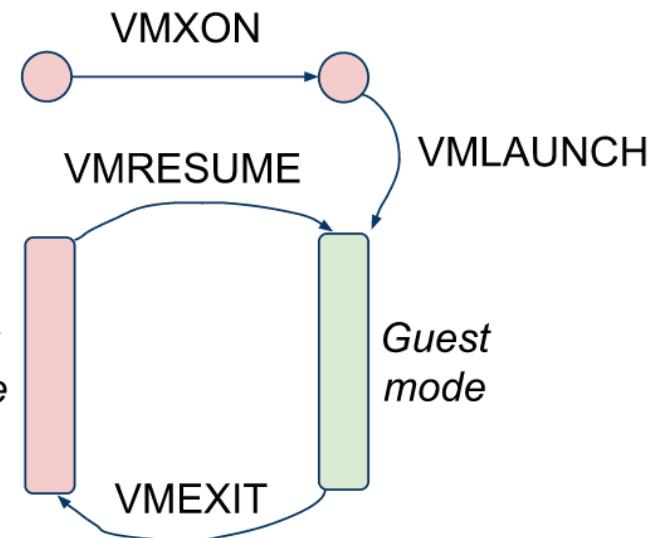
Qemu Process Seen from the Host

- Host kernel schedules Qemu as a regular process
- Host cannot see the processes running inside the guest
- Guests have a *vcpu* thread per virtual CPU
- A dedicated iothread runs a `select()` event loop to process I/O such as network packets and disk I/O completion

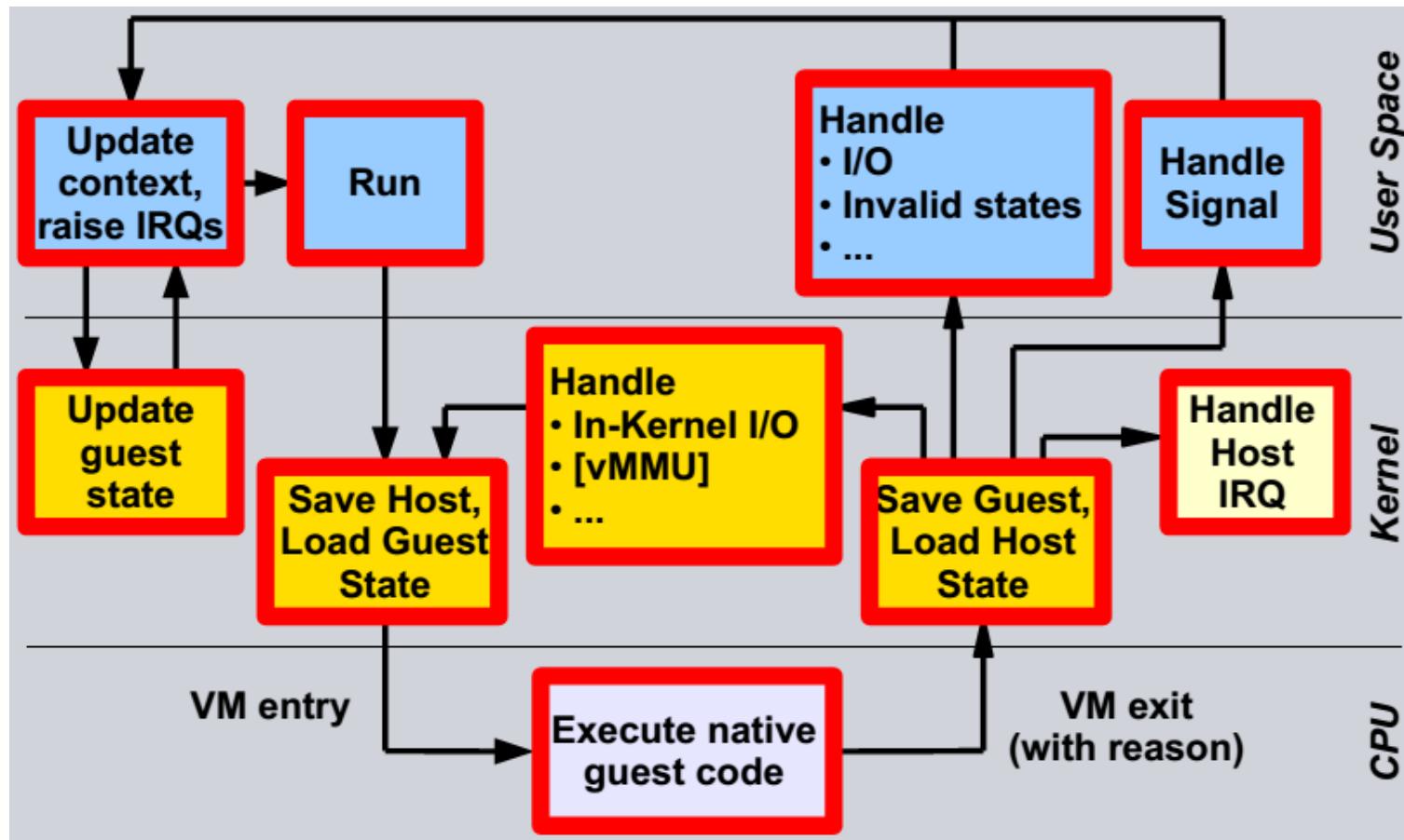


What Happens After ioctl(KVM_RUN)?

- The kernel finds the VMCS data structure
- Load VMCS to processor by VMENTRY
- The processor switches from root-mode to non-root
- The IP is changed to VMCS->IP, and start to run guest VM's code



VCPU Execution Flow (KVM View)



Architectural Advantages of the KVM Model

- Proximity of guest and user space hypervisor
 - Only one address space switch: guest ↔ host
 - Less rescheduling
- Massive Linux kernel reuse
 - Scheduler, Memory management with swapping, I/O stacks
 - Power management, Host CPU hot-plugging, ...
- Massive Linux user land reuse
 - Network configuration
 - Handling VM images
 - Logging, tracing, debugging

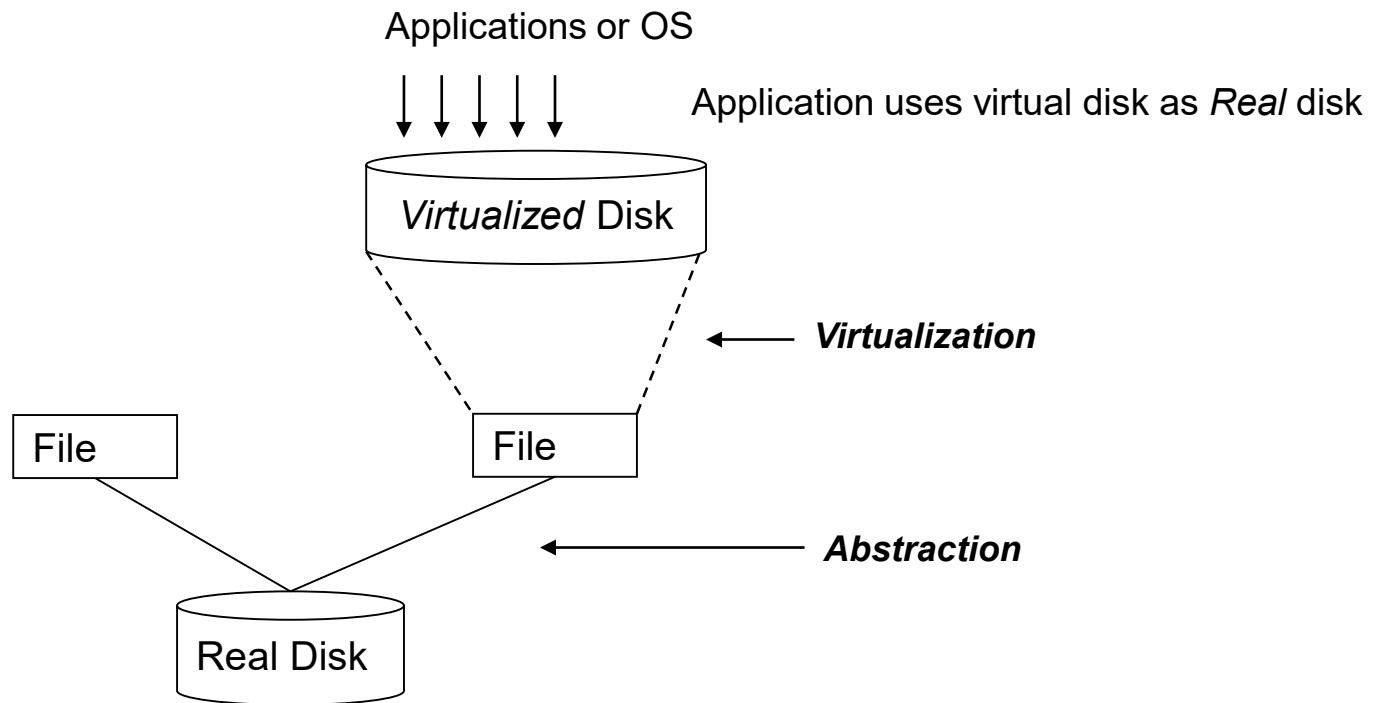
I/O VIRTUALIZATION

Device Virtualization

1. Emulated
2. Para-virtualized
3. Direct Access (Passthrough)
4. Hardware assisted I/O virtualization

EMULATED DEVICE

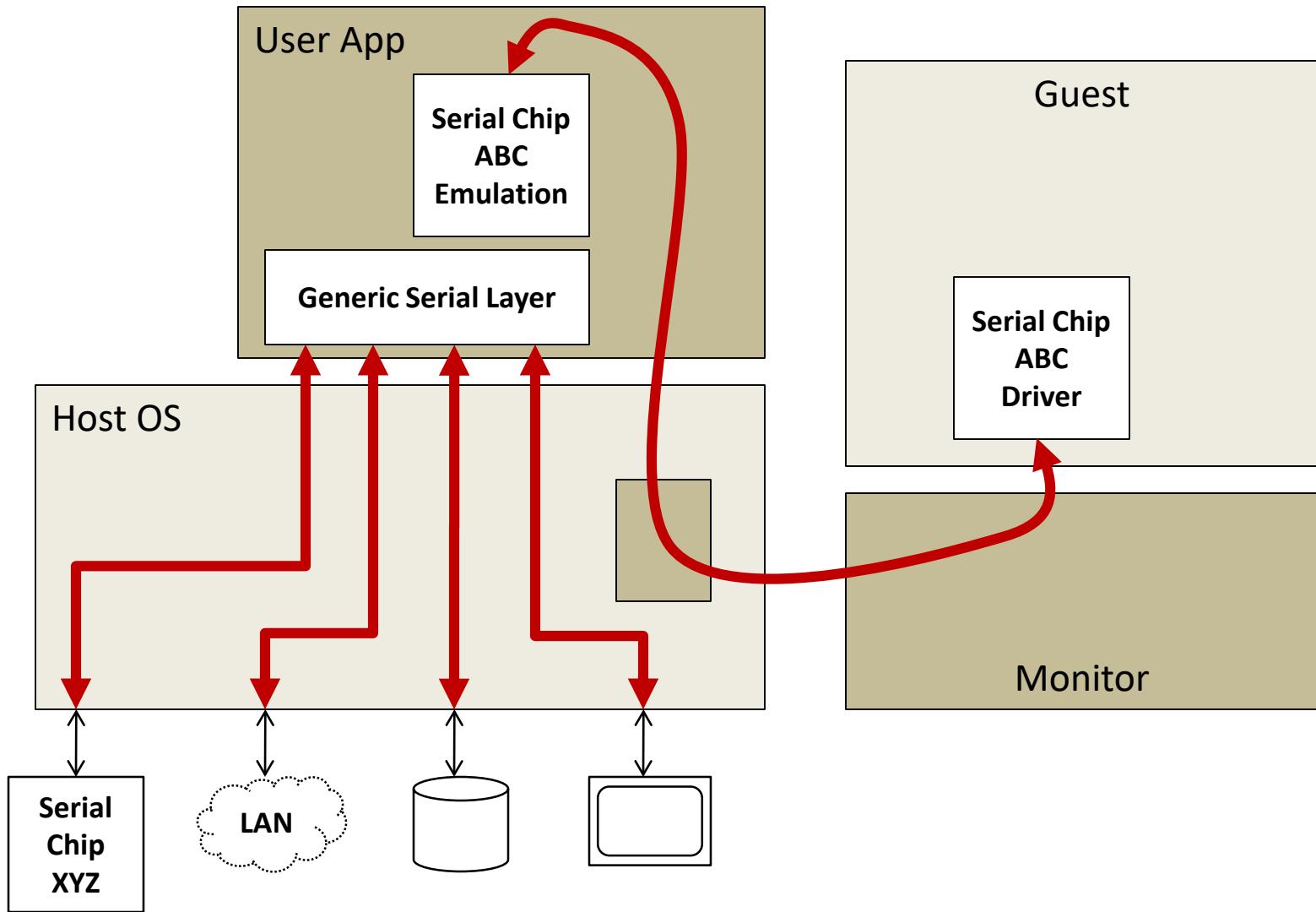
Abstraction & Virtualization



Emulated Devices

- Emulate a device in class
 - Emulated registers
 - Memory mapped I/O or programmed I/O
- Convert
 - Intermediate representation
- Back-ends per real device

Serial Port Example



Xen's Domain-0 (Full-Virtualization)

- Qemu for device emulation
 - User-level application running in domain-0
 - Implement NIC (e.g., 8139) in software
 - Each VM has its own Qemu instance running
- I/O request redirection
 - Guest VM's I/O requests are sent to domain-0
 - Then Domain-0 sends to Qemu
 - Eventually, Qemu will use domain-0's NIC driver

Emulated Devices

Positives

Platform stability

Allows interposition

No special hardware support needed

Isolation, multiplexing implemented by monitor

Negatives

Can be slow

Drivers needed in monitor or host

PARA-VIRTUALIZATION

Para-Virtualized Devices

Guest passes requests to Monitor at a higher abstraction level

- Monitor calls made to initiate requests

- Buffers shared between guest / monitor

Positives

- Simplify monitor

- Fast

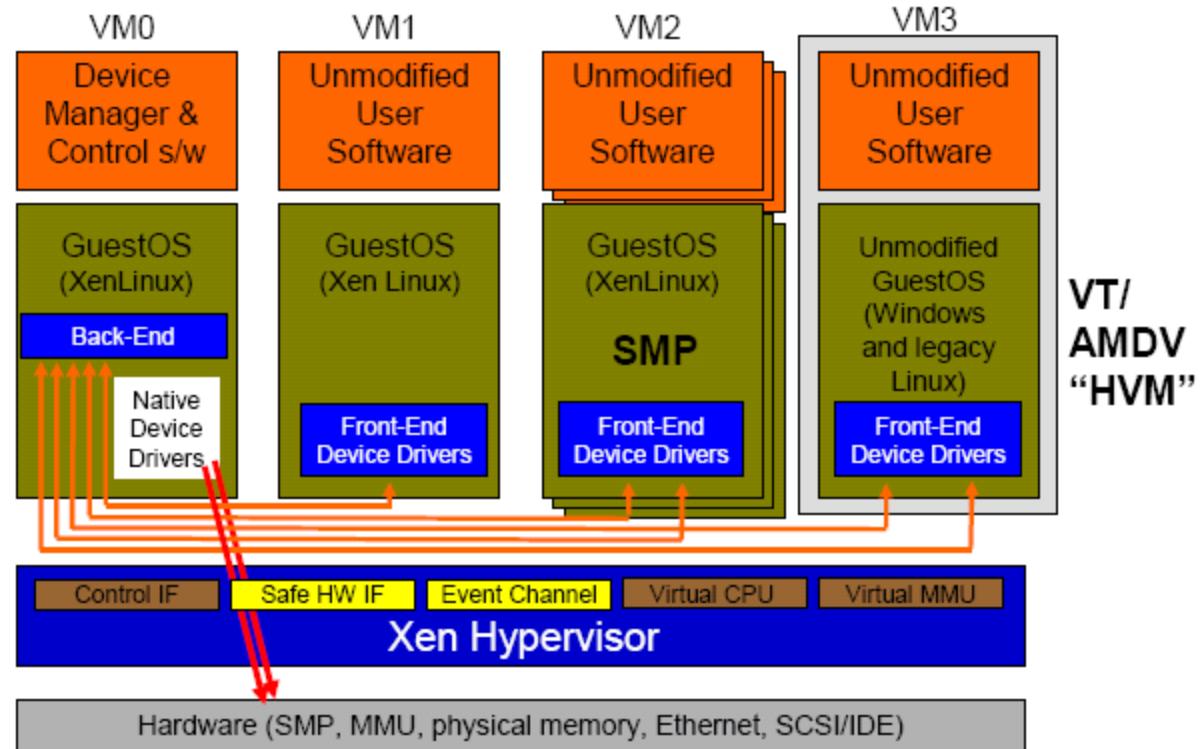
Negatives

- Monitor needs to supply guest-specific drivers

- Bootstrapping issues

Para-virtualized Devices: Front-end/Back-end

- Para – virtualization
 - Linux Guest
- Hardware-supported virtualization
 - Unmodified Windows
- Isolated Device Drivers



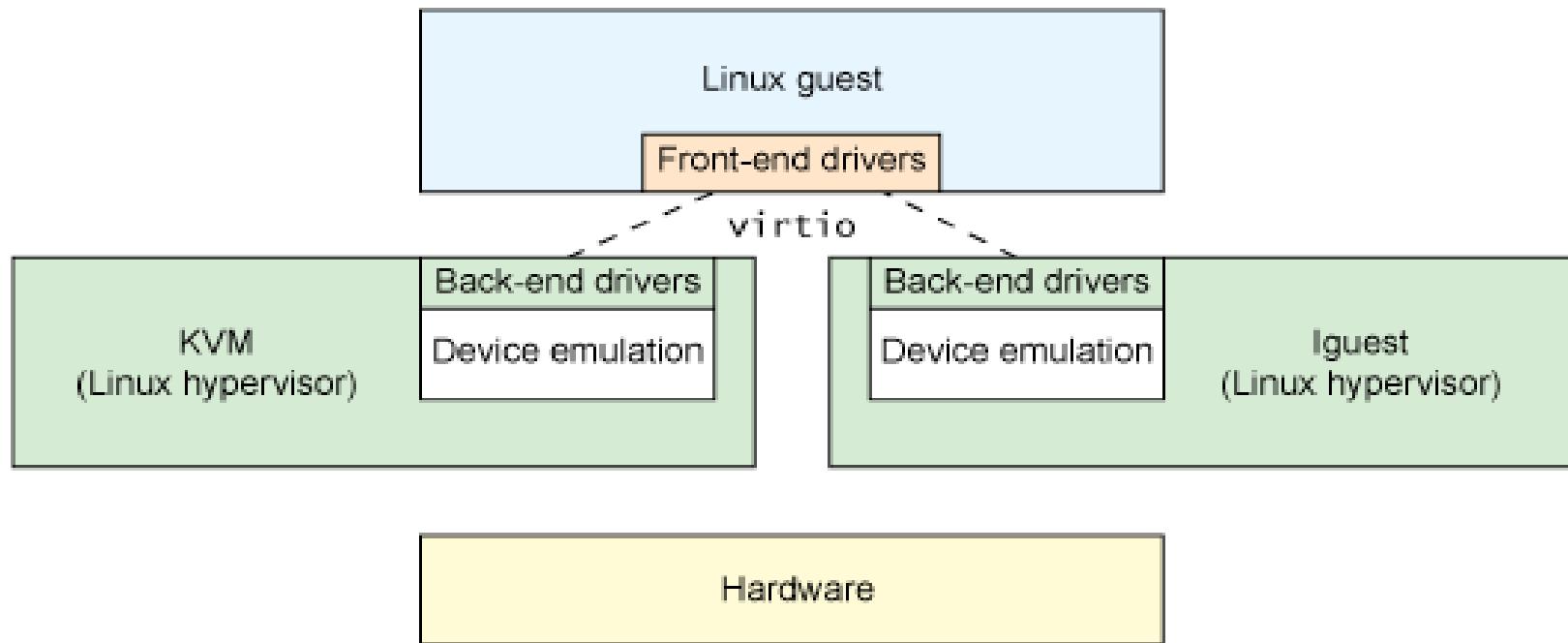
*Source: Ottawa Linux Symposium 2006 presentation.
<http://www.cl.cam.ac.uk/netos/papers/>*

VirtIO: Unified Para-virtualized I/O

Motivation: Linux has support for at least 8 virtualization platforms, each with its own para-virtualized I/O design/interfaces

VirtIO: to provide a unified I/O mode for para-virtualized device

Adopted by KVM and Iguest



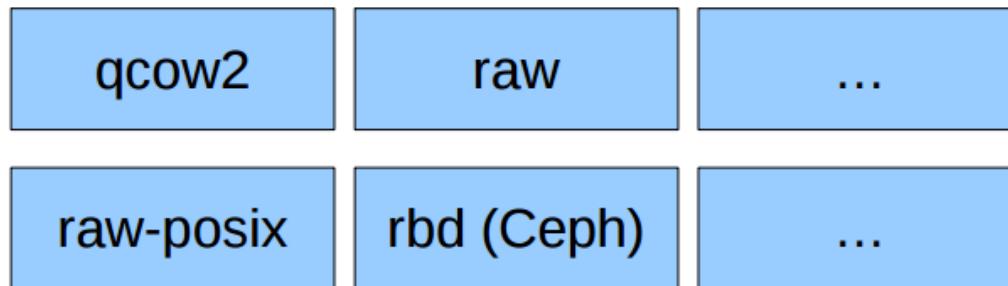
Virtio in KVM

- Virtio is a framework and set of drivers
 - A hypervisor-independent, domain-independent, bus-independent protocol for transferring buffers
 - A binding layer for attaching virtio to a bus (e.g., PCI)
 - Domain specific guest drivers (networking, storage, etc.)
 - Hypervisor specific host support

Storage in QEMU

Block drivers fall in two categories:

Formats – image file formats (qcow2, vmdk, etc)



Protocols – I/O transports (POSIX file, rbd/Ceph, etc)

Plus additional block drivers that interpose like quorum, blkdebug, blkverify

Storage stack



Guest – application plus full file system and block layer



QEMU – image format, storage migration, I/O throttling



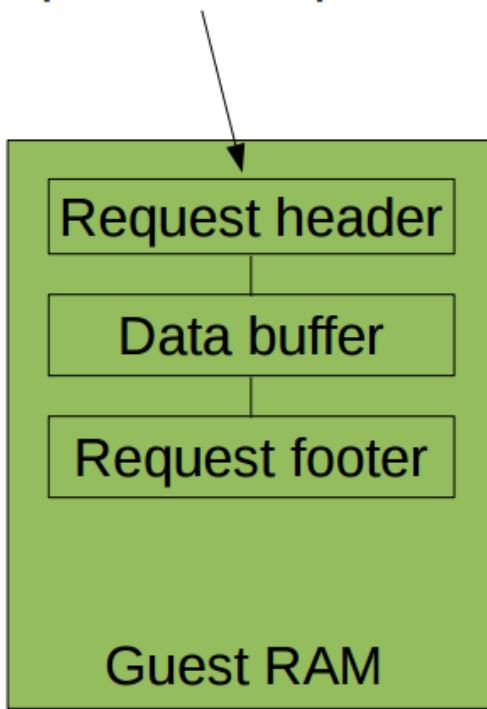
Host – full file system and block layer



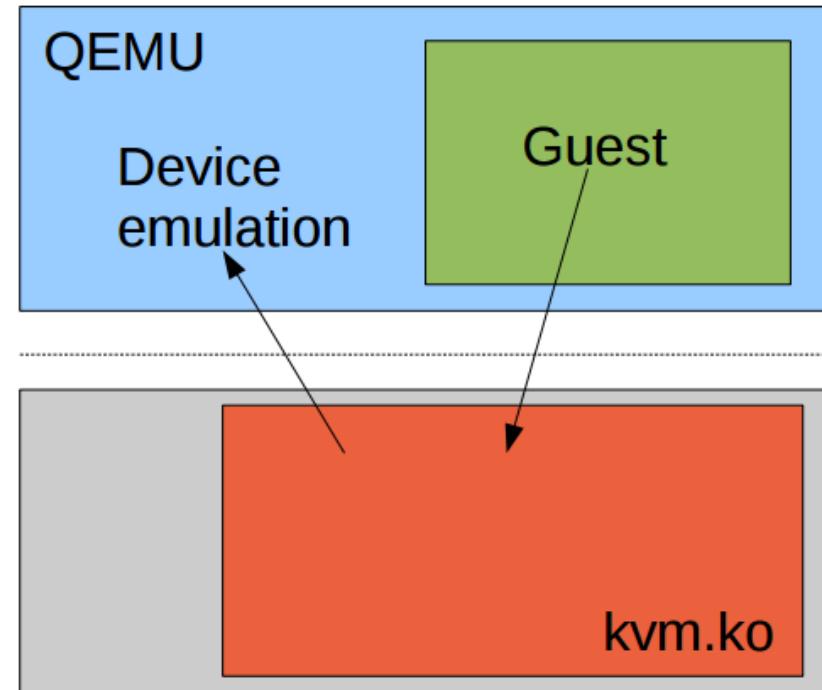
Beware double caching and anticipatory scheduling delays!

Walkthrough: virtio-blk disk read request (Part 1)

1. Guest fills in
request descriptors

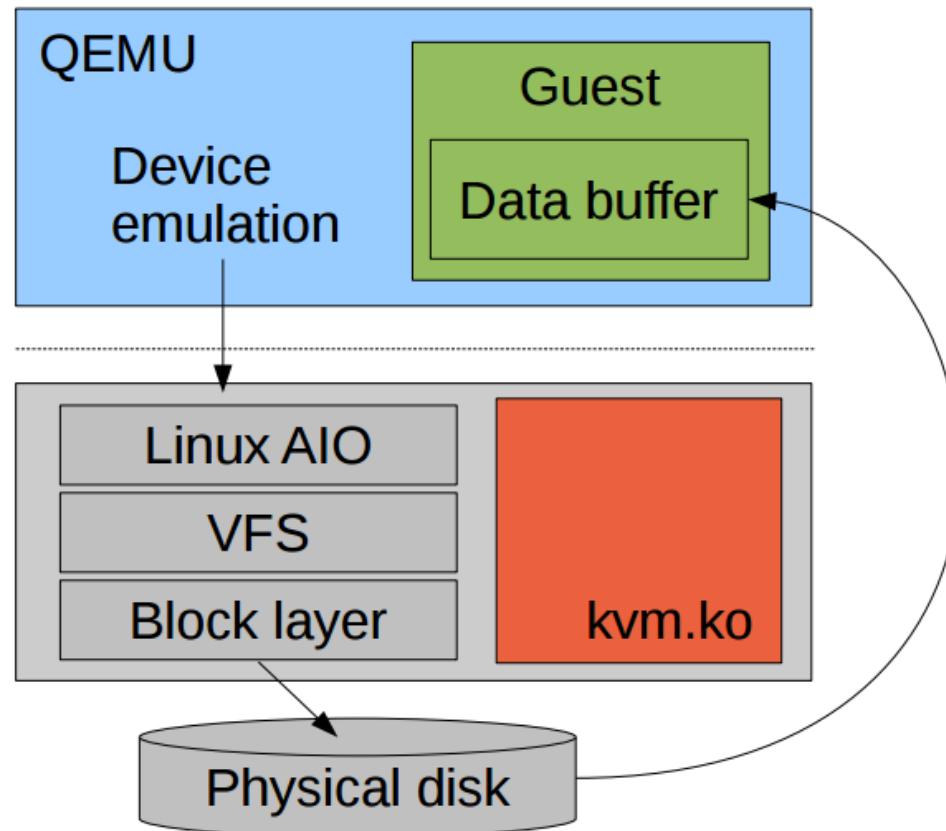


2. Guest writes to virtio-blk
virtqueue notify register



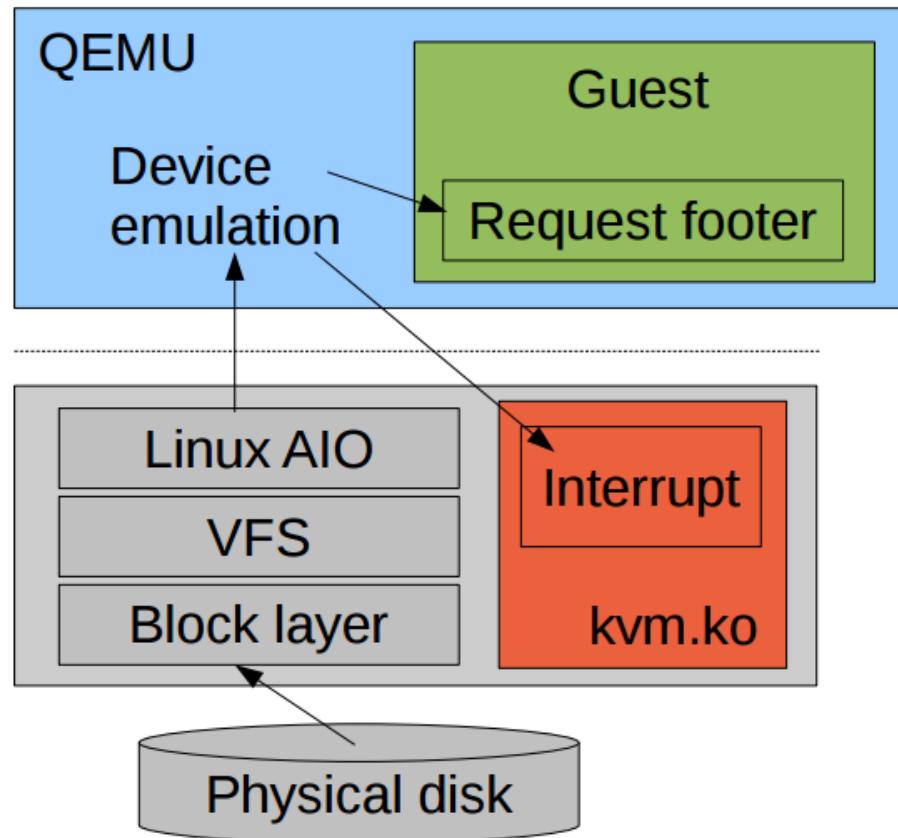
Walkthrough: virtio-blk disk read request (Part 2)

3. QEMU issues I/O request on behalf of guest



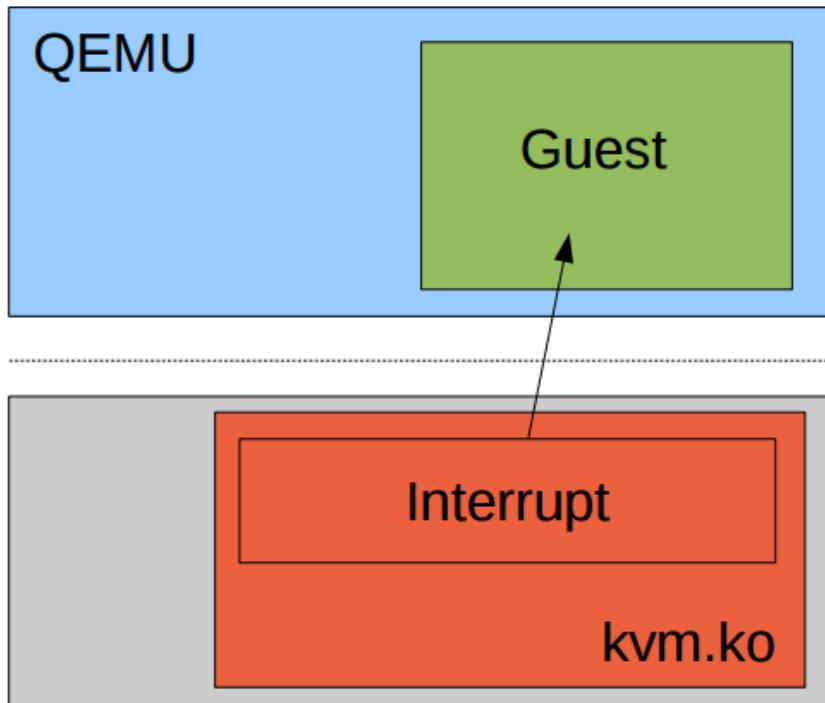
Walkthrough: virtio-blk disk read request (Part 3)

4. QEMU fills in request footer and injects completion interrupt

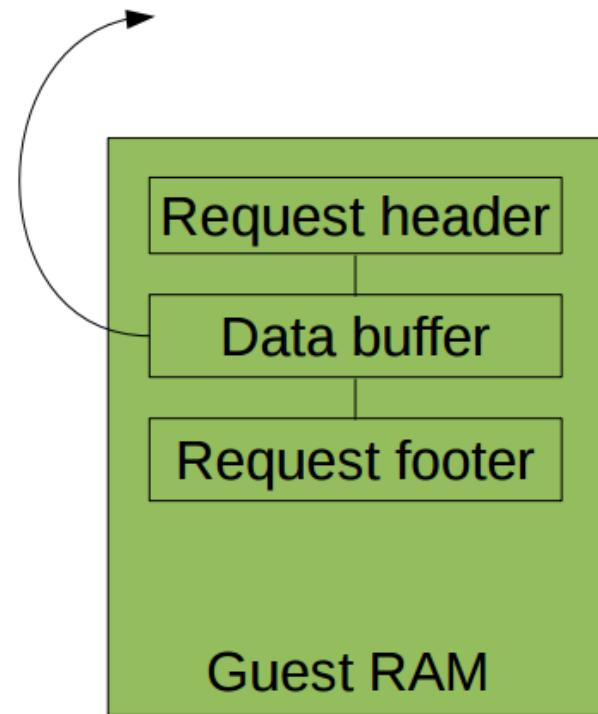


Walkthrough: virtio-blk disk read request (Part 4)

5. Guest receives interrupt and executes handler



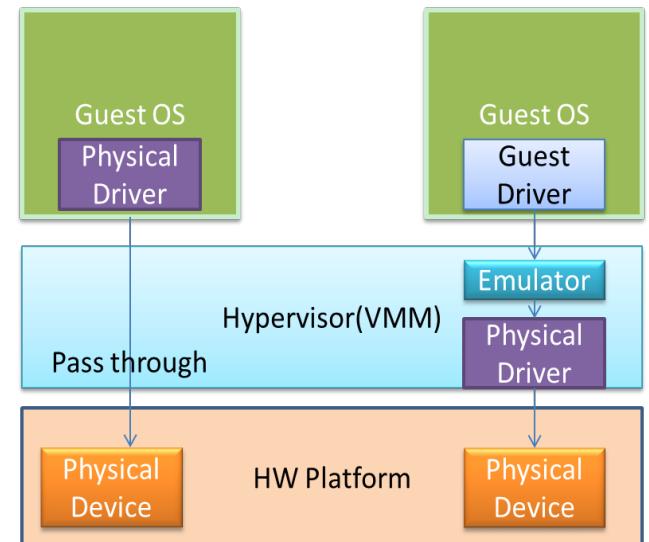
6. Guest reads data from buffer



HARDWARE ASSISTED

Directed I/O

- Software-based sharing adds overhead to each I/O due to emulation layer
 - This indirection has the additional affect of eliminating the use of hardware acceleration that may be available in the physical device.
- Directed I/O has added enhancements to facilitate memory translation and ensure protection of memory that enables a device to directly DMA to/from host memory.
 - Bypass the VMM's I/O emulation layer
 - Throughput improvement for the VMs



Direct Access Device Virtualization

- Allow Guest OS direct access to underlying device
- **Positives**
 - Fast
 - Simplify monitor
 - Limited device drivers needed
- **Negatives**
 - Need hardware support for safety (IOMMU)
 - Need hardware support for multiplexing
 - Hardware interface visible to guest
 - Limits mobility of VM
 - Interposition hard by definition

Issues to Address

I/O address translation

How to translate I/O address to host physical address

Interrupt mapping

How to route an interrupt correctly to a guest VM

Device multiplexing

How to multiplex a single hardware device among multiple VMs

Mostly importantly

Provide strong isolation, while reduce hypervisor involvement

VT-d: Intel® Virtualization Technology for Directed I/O

Provides the capability to ensure improved isolation of I/O resources for greater reliability, security, and availability.

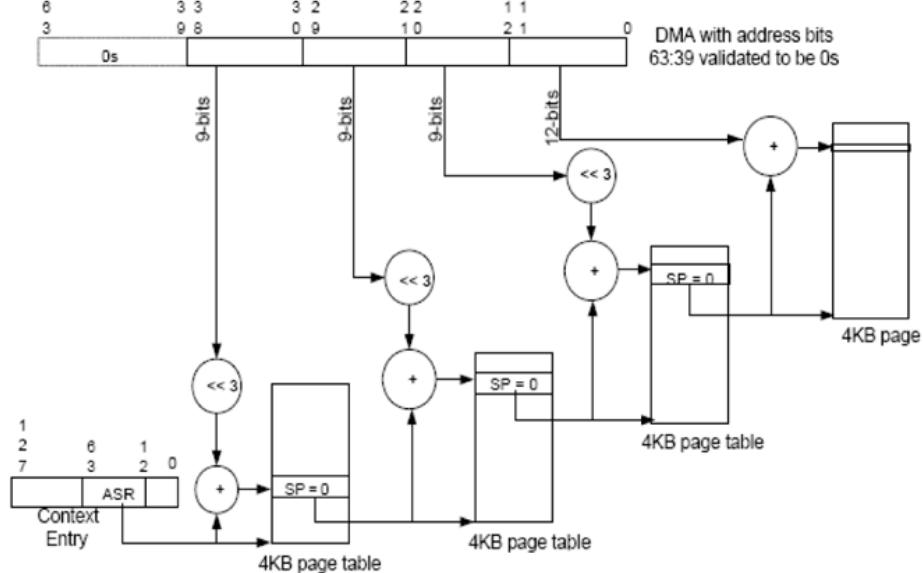
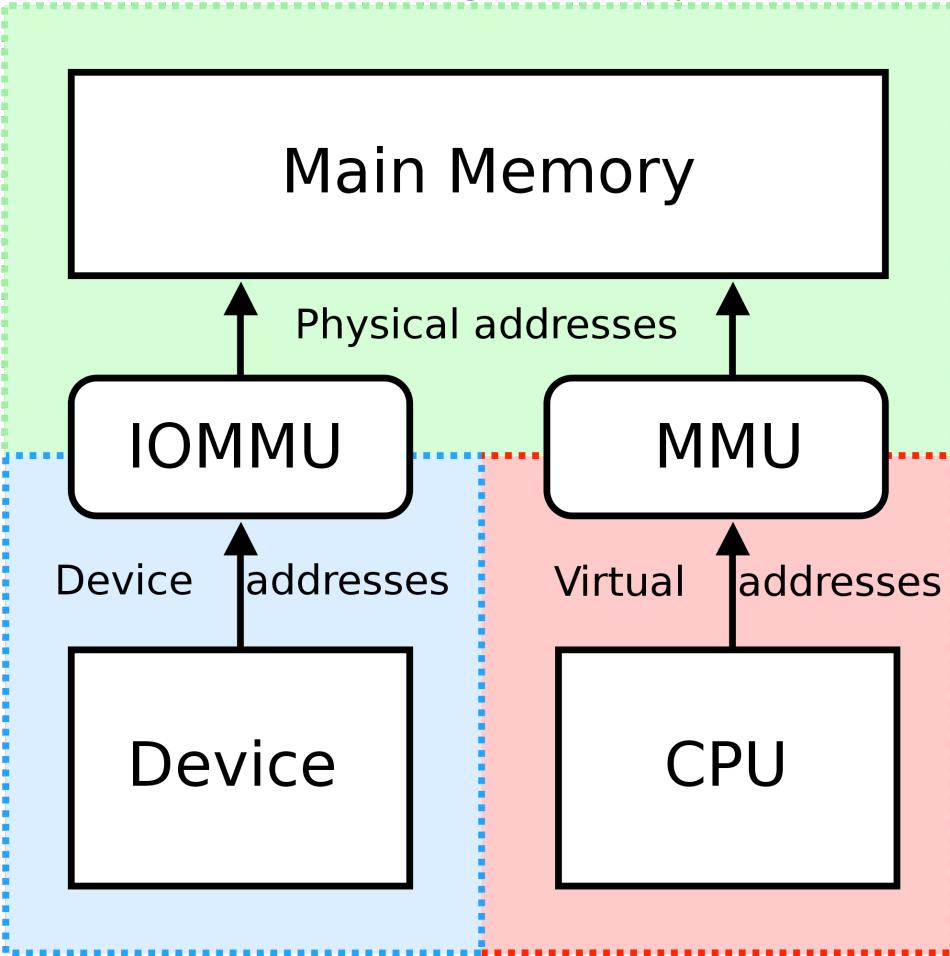
Supports the remapping of I/O DMA transfers and device-generated interrupts.

Provides flexibility to support multiple usage models that may run un-modified, special-purpose, or "virtualization aware" guest OSs.

Address Translation Services

VT-d architecture defines a multi-level page-table structure for DMA address translation

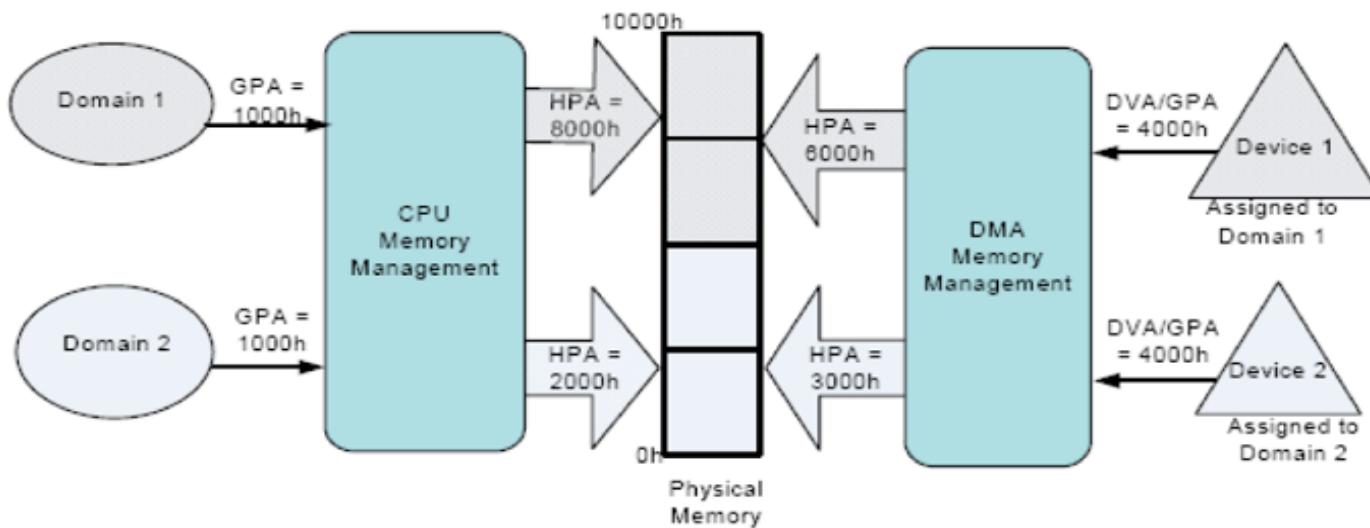
The multi-level page tables are similar to IA-32 processor page-tables, enabling software to manage memory at 4 KB or larger page granularity



DMA Remapping

DMA-remapping translates the address of the incoming DMA request to the correct physical memory address and perform checks for permissions to access that physical address

DMA-remapping hardware logic in the chipset sits between the DMA capable peripheral I/O devices and the computer's physical memory



VT-d Feature: Interrupt Remapping

The interrupt requests generated by I/O devices must be controlled by the VMM

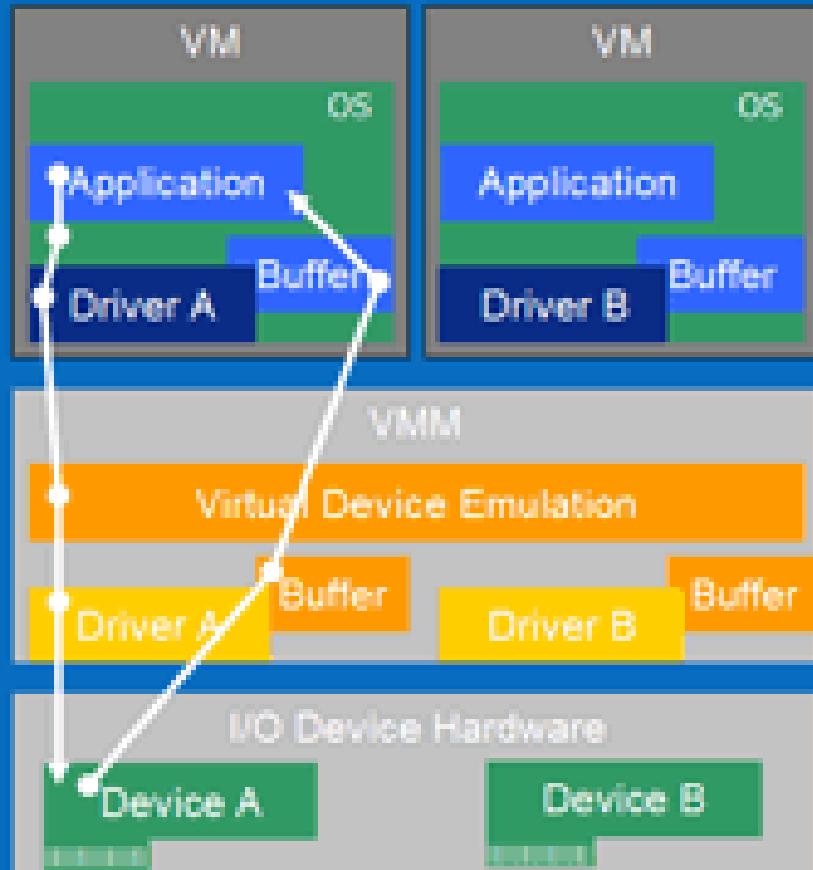
When the interrupt occurs, the VMM must present the interrupt to the guest. This is not accomplished through hardware.

The VT-d interrupt-remapping architecture addresses this problem by redefining the interrupt-message format.

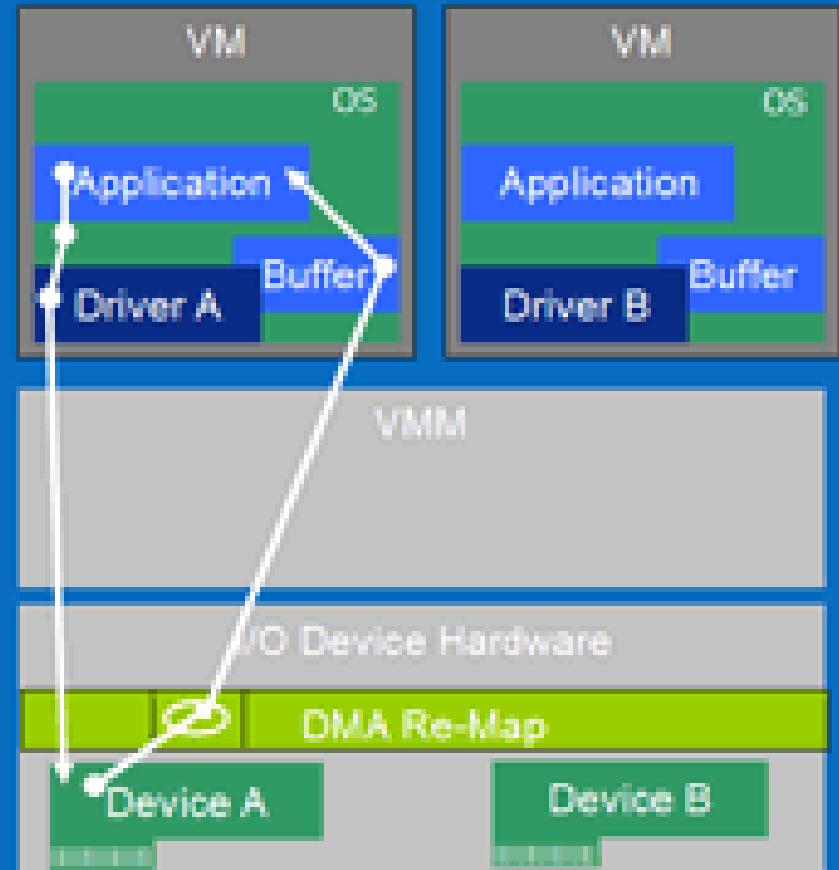
Interrupt requests specify a requester-ID and interrupt-ID, and remap hardware transforming these requests to a physical interrupt.

Interrupt Remapping

Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) – Direct assignment



Emulation Based Virtual I/O



Direct Assigned I/O

Drawbacks to Directed I/O

- One concern with direct assignment is that it has limited scalability
 - A physical device can only be assigned to one VM.
 - For example, a dual port NIC allows for direct assignment to two VMs. (one port per VM)
 - Consider for a moment a fairly substantial server of the very near future
 - 4 physical CPU's
 - 12 cores per CPU
 - If we use the rule that one VM per core, it would need 48 physical ports.

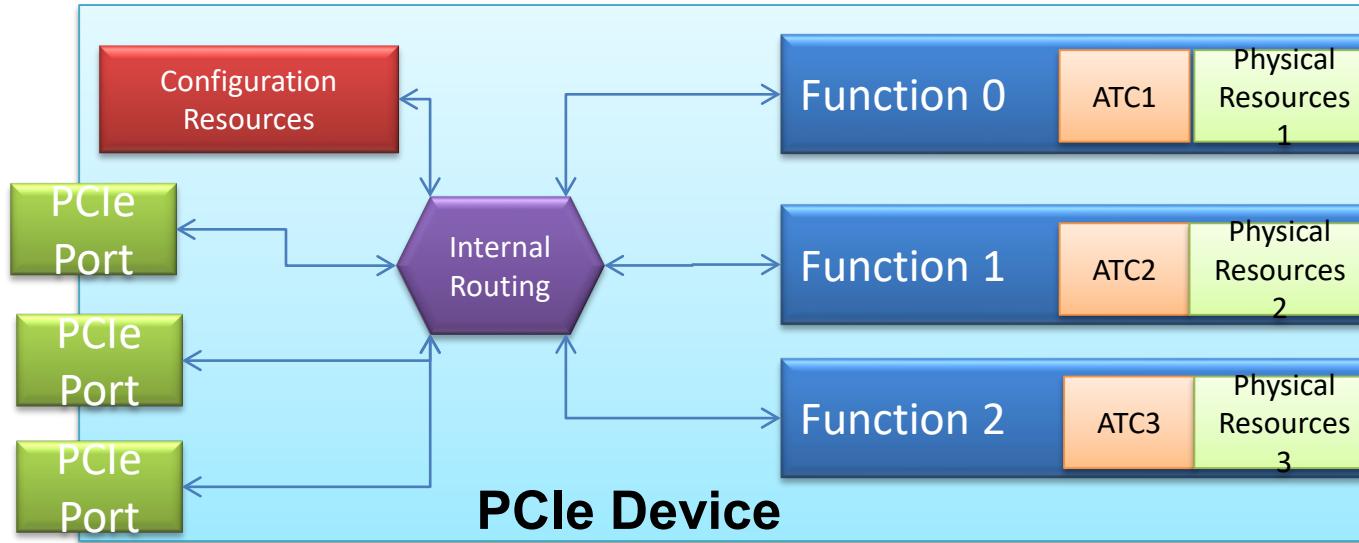
Device multiplexing: Single Root I/O Virtualization

- Single Root I/O Virtualization (SR-IOV) is a Peripheral Component Interconnect Special Interest Group (PCI-SIG) specification.
- SR-IOV provides a standard mechanism for devices to advertise their ability to be simultaneously shared among multiple virtual machines.
- SR-IOV allows for the partitioning of a PCI function into many virtual interfaces for the purpose of sharing the resources of a PCI Express* (PCIe) device in a virtual environment.

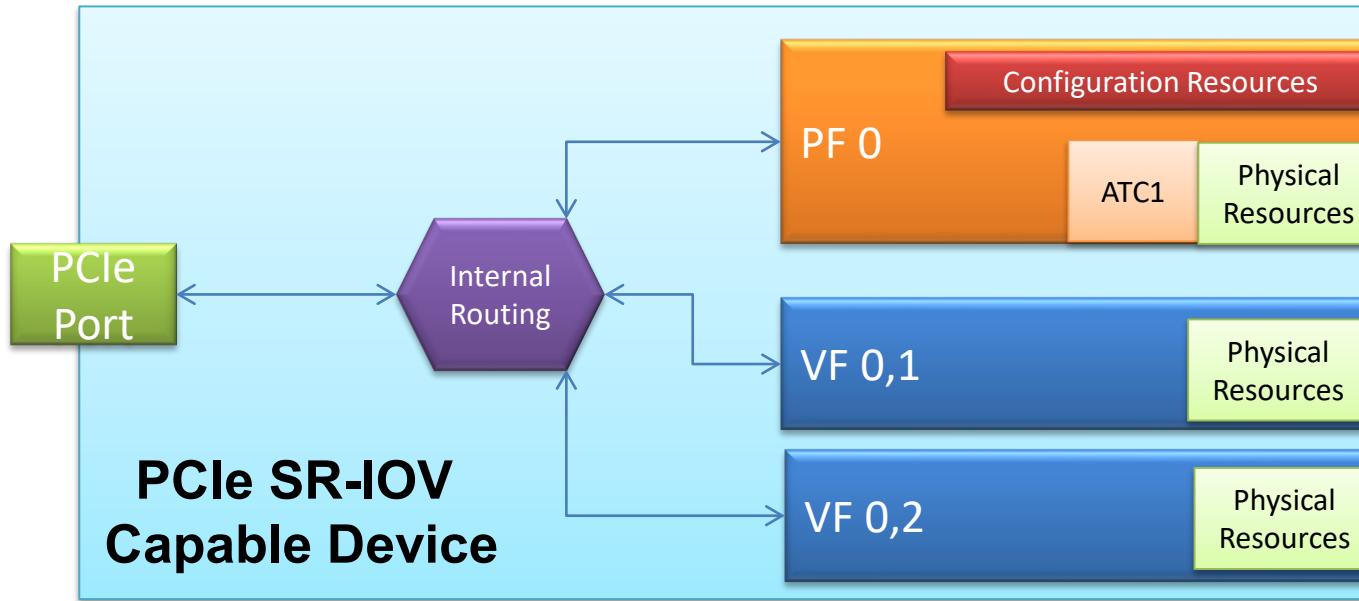
Device multiplexing: Single Root I/O Virtualization

- With SR-IOV:
 - SI's will get direct access to PCIe device functions
 - No more need for hypervisor (VI) to manage all system resources
- PCIe devices will have multiple virtual functions (VF's)
 - utilizable by multiple SI's
 - a single SI may also use multiple virtual functions
- Security of I/O Streams ensured by
 - Independency of control structures between VF's within one PCIe device
 - I/O address translation services
 - Interrupt remapping mechanisms

Physical V.S. Virtual



Physical



Virtual

Single Root I/O Virtualization

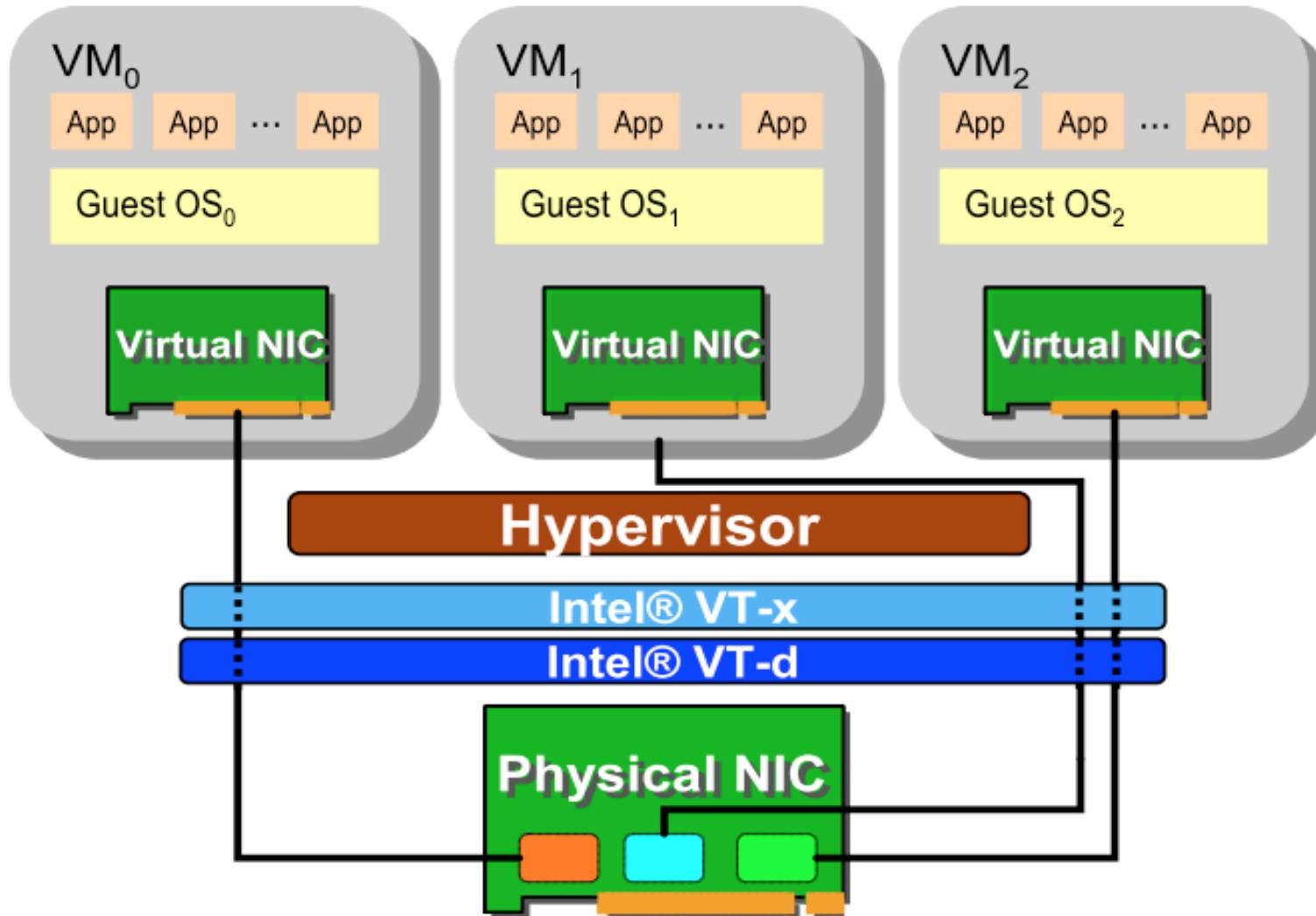
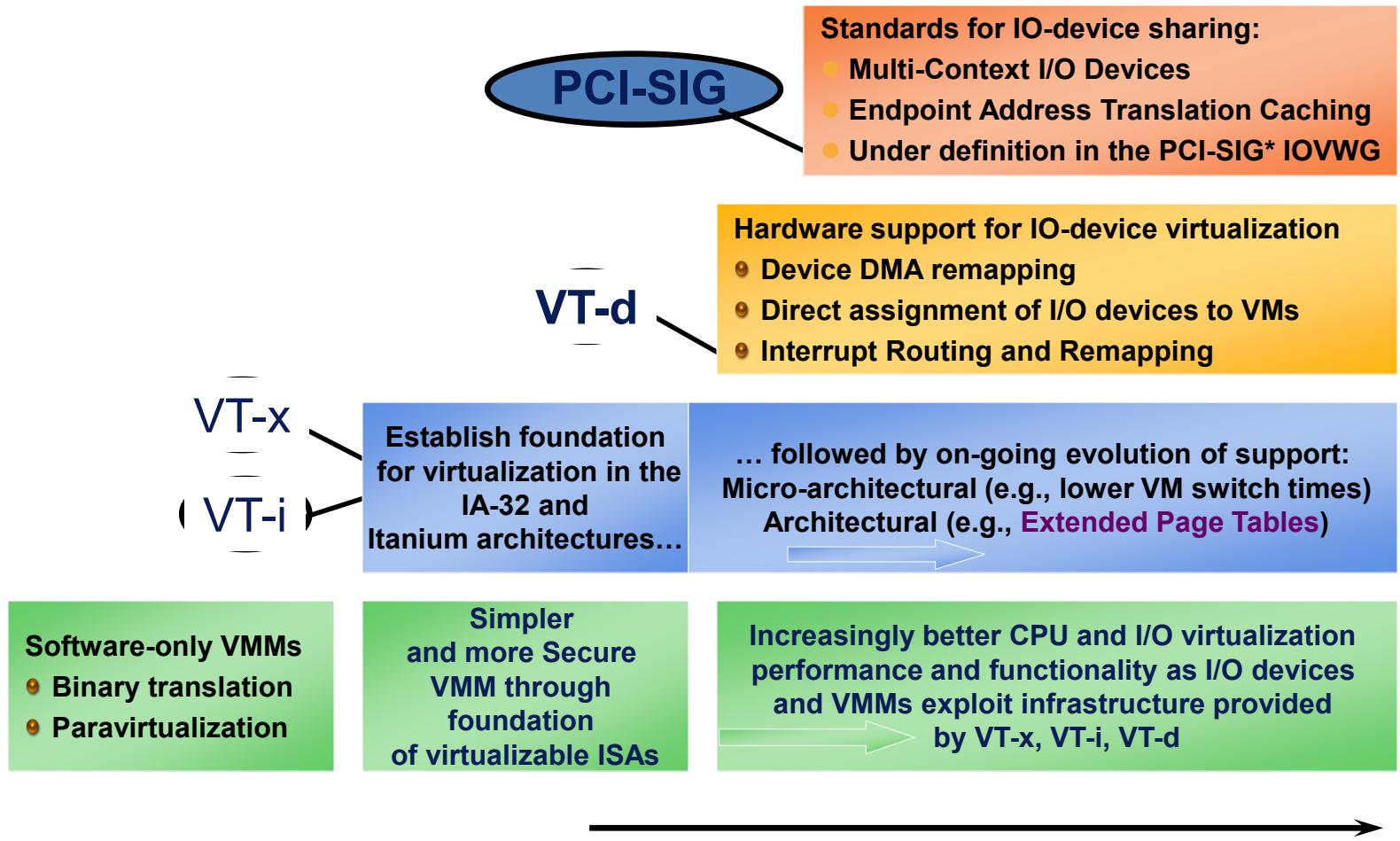


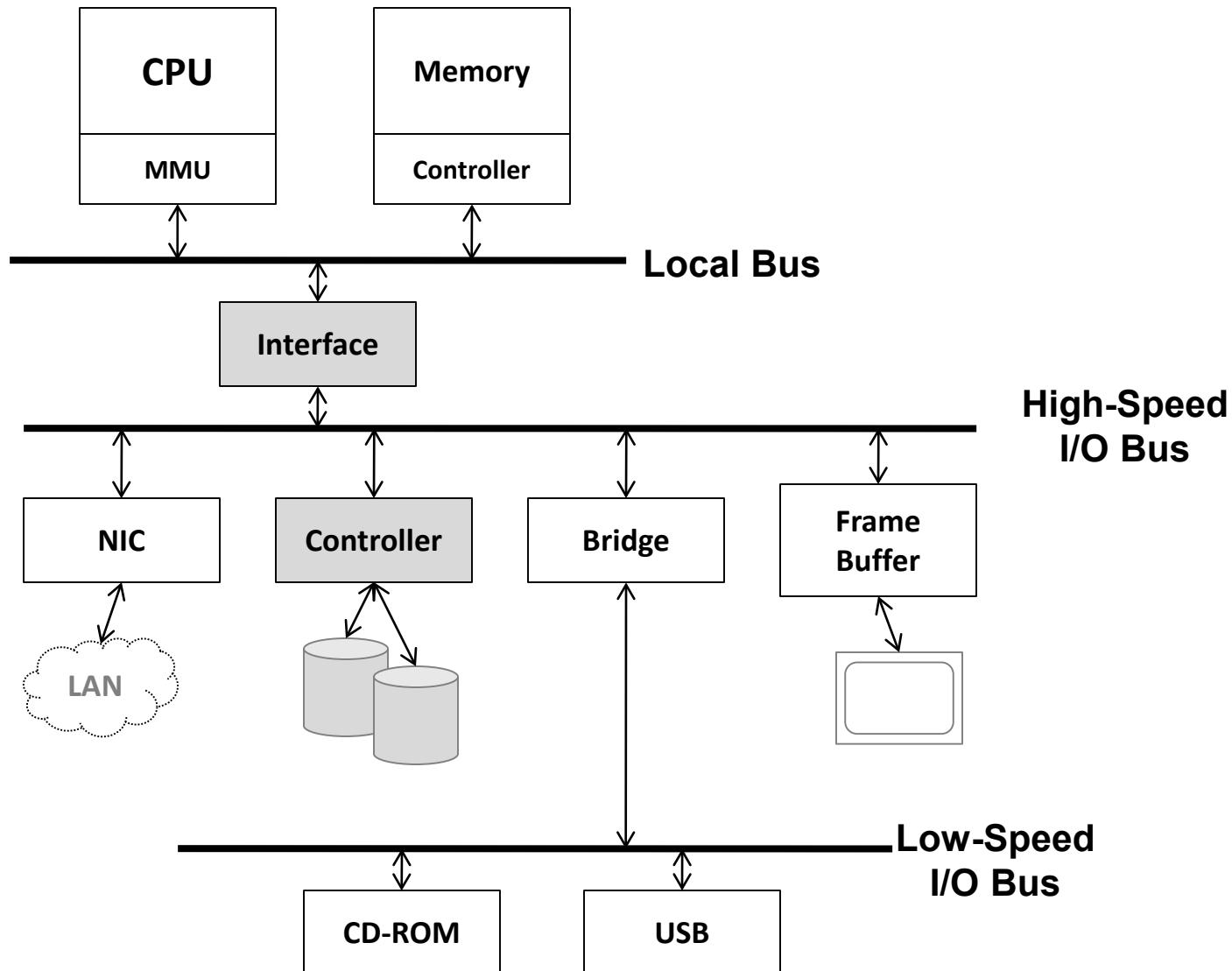
Figure 1. Natively Shared

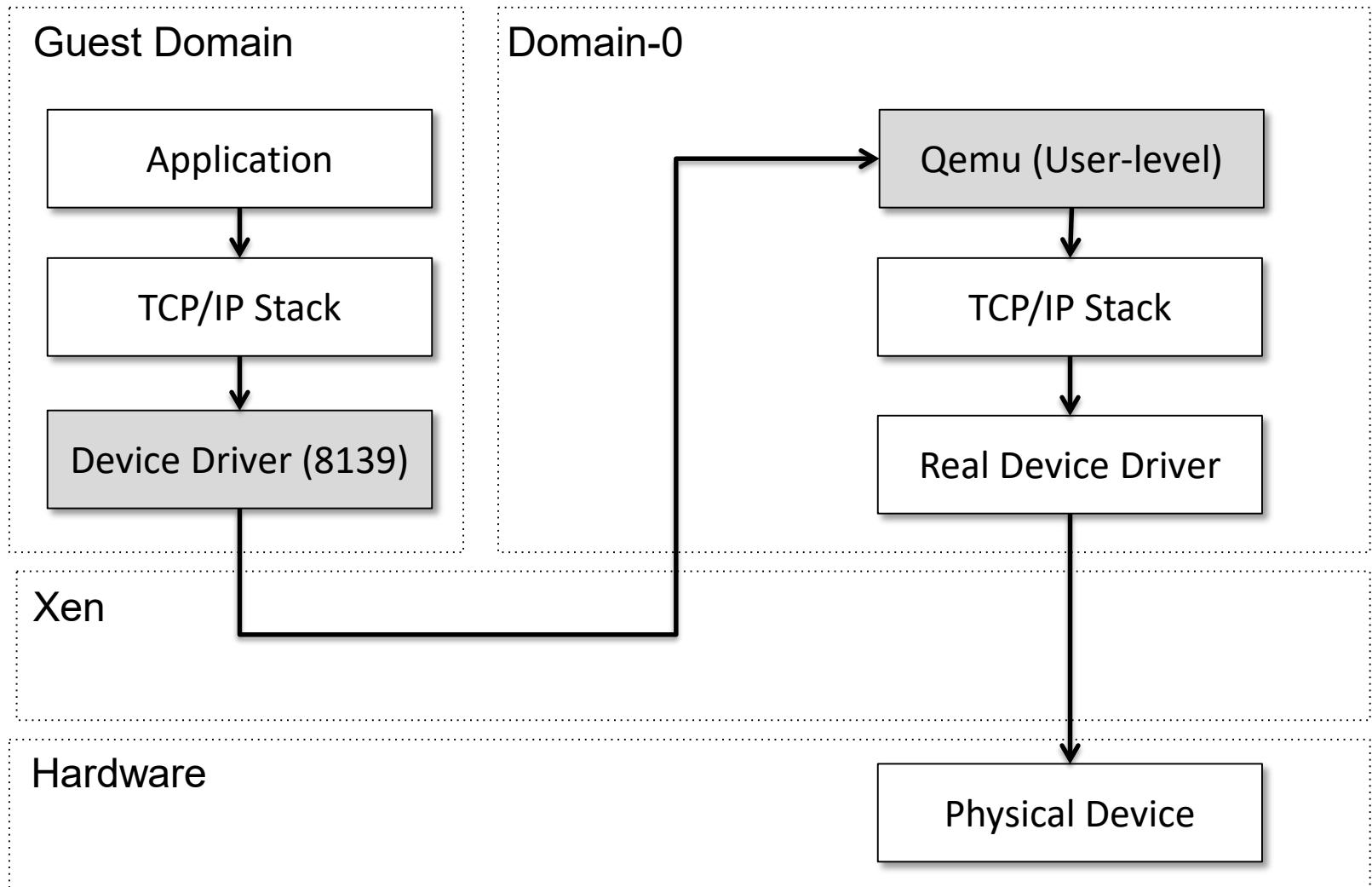
Intel Virtualization Technology Evolution

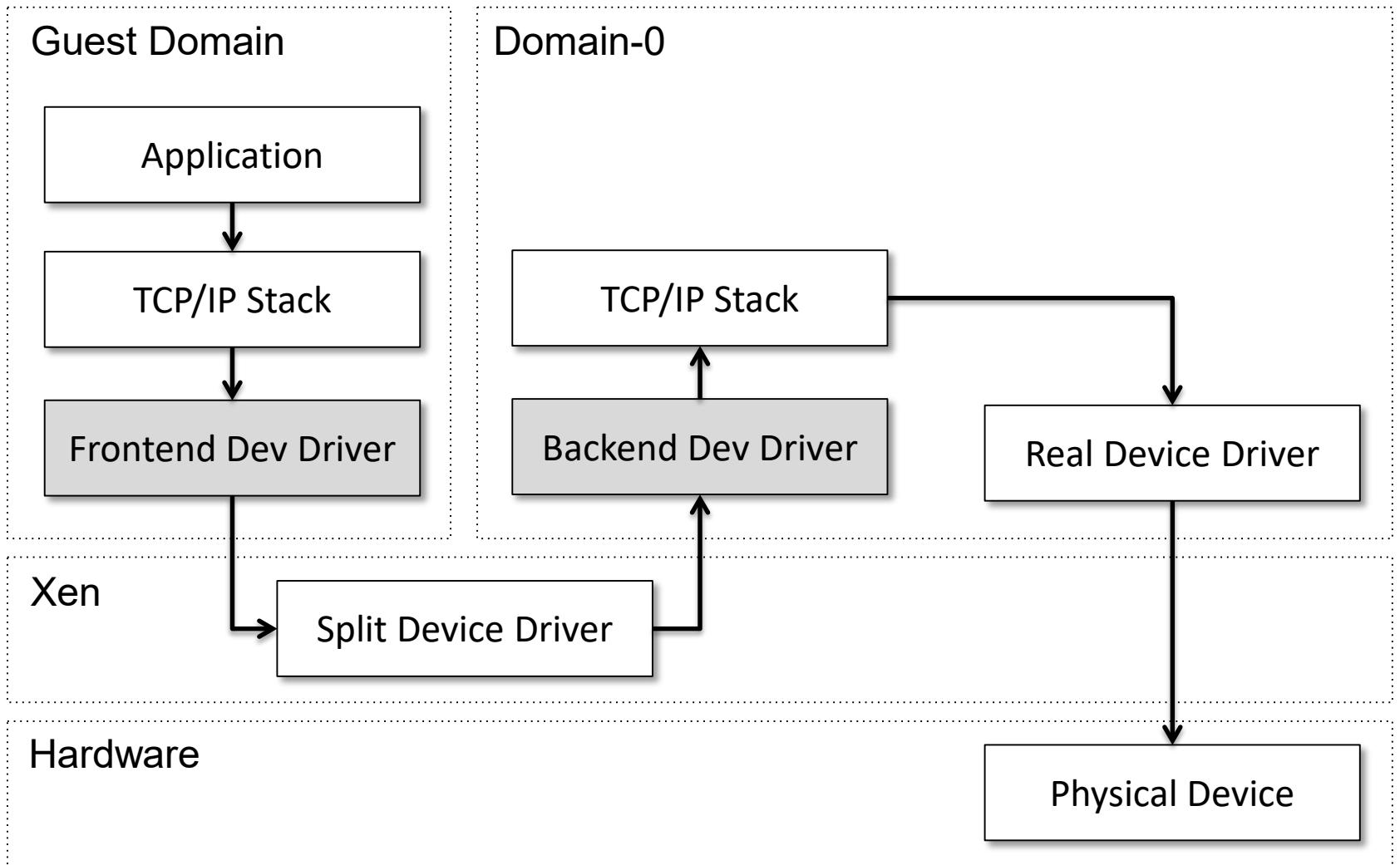


SUMMARY

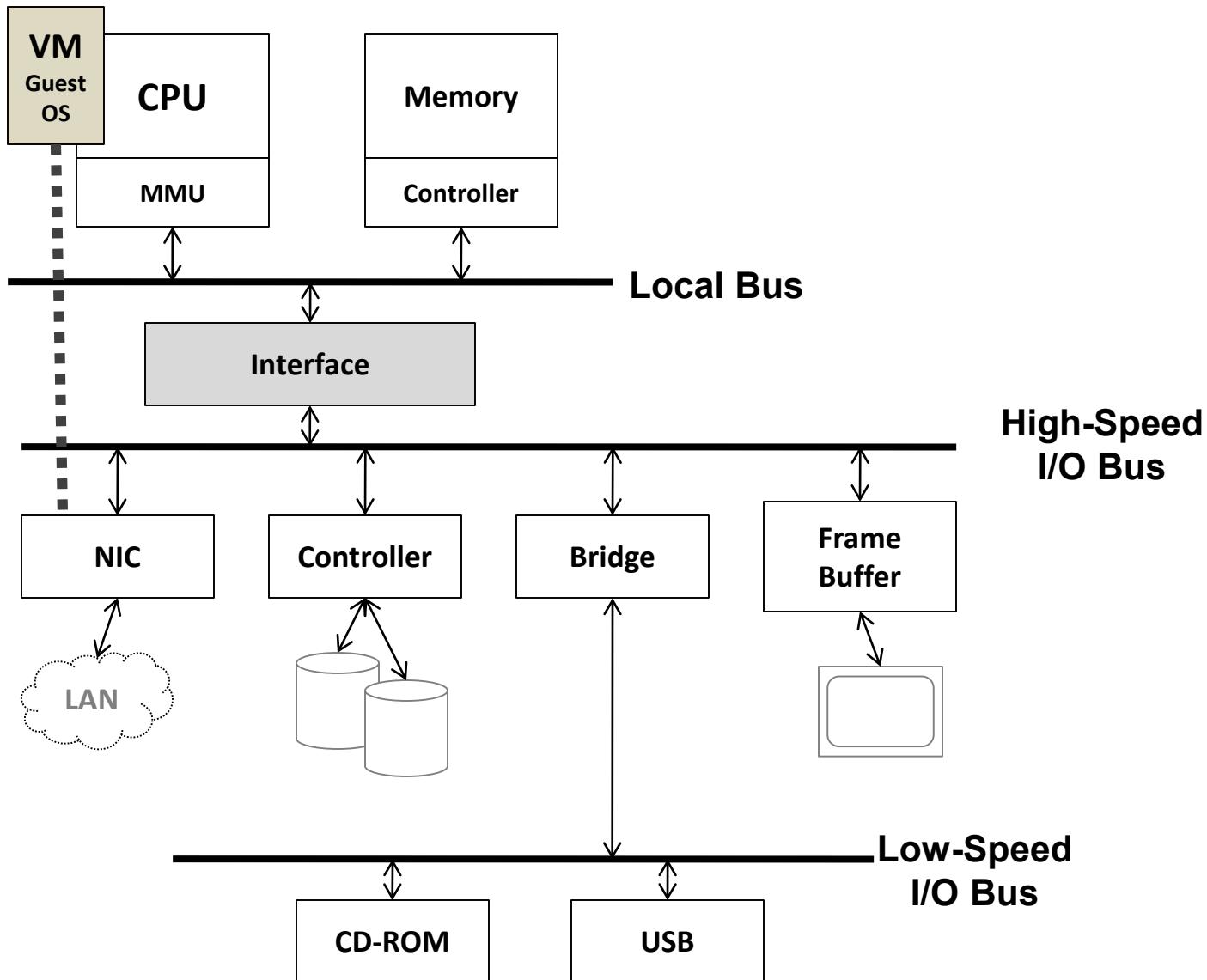
Computer System Organization



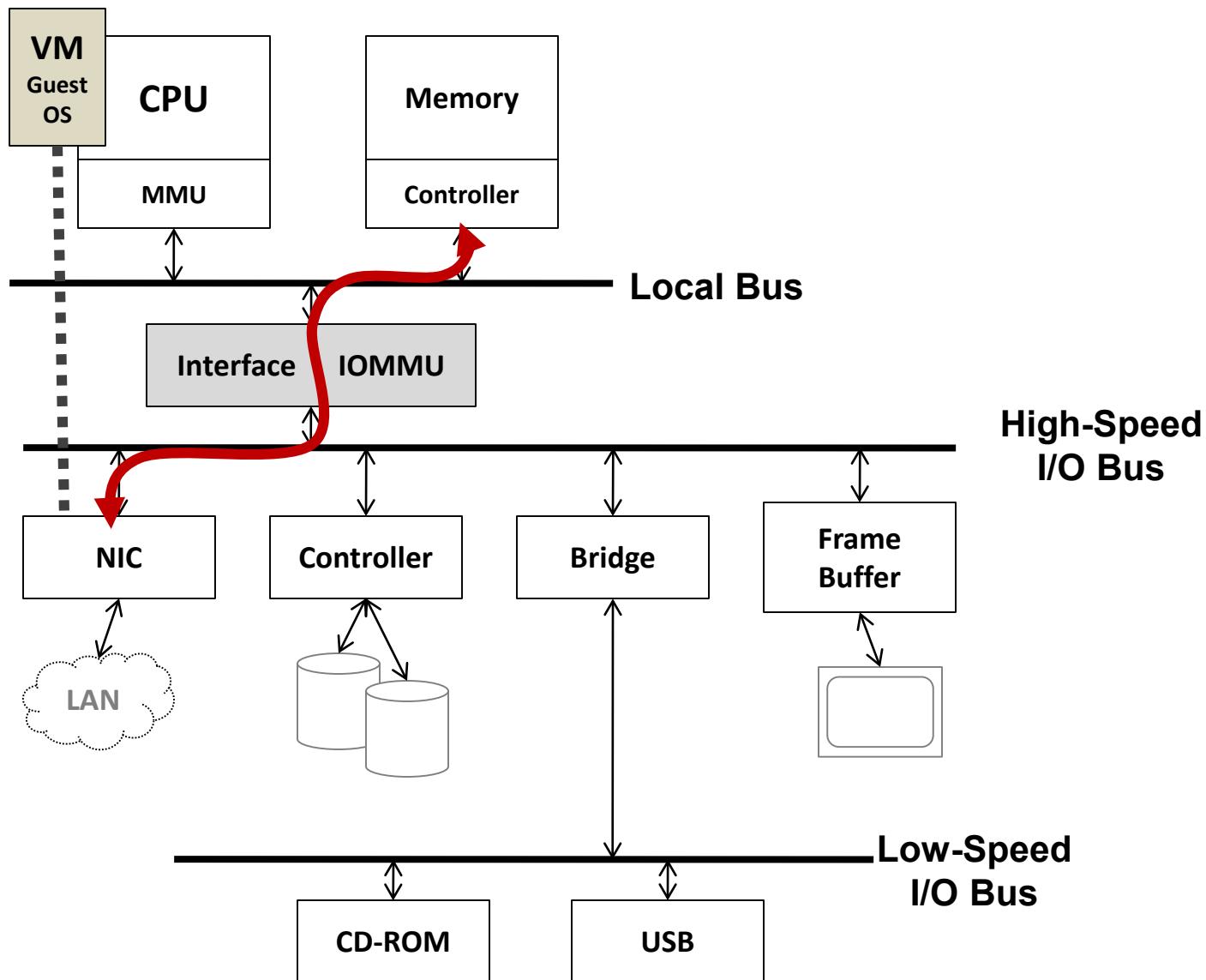




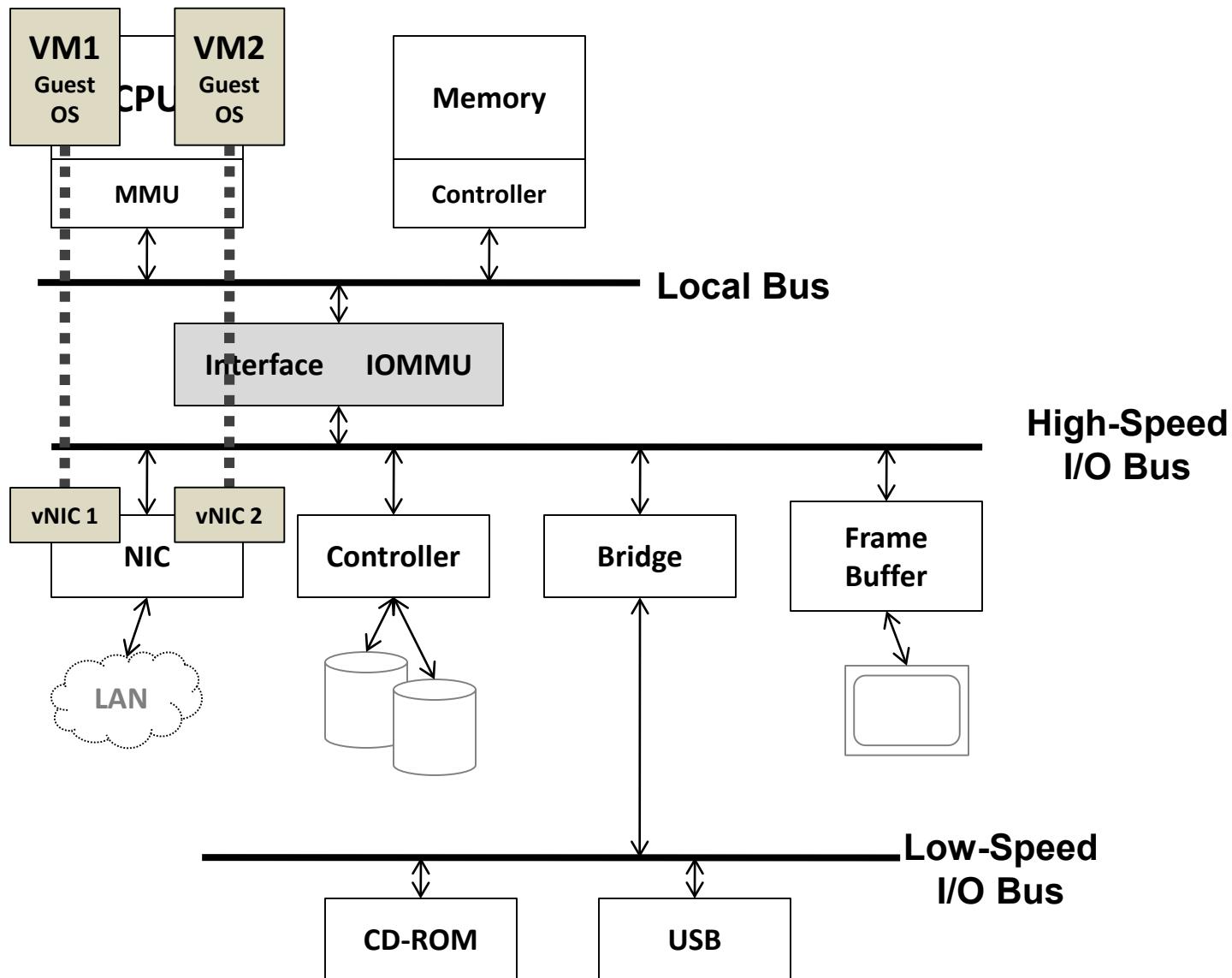
Summary of I/O Virtualization



Memory Isolation w/ Direct Access Device



Virtualization Enabled Device



Control Flow of Pressing 'a' and Displaying 'a'

- Keyboard triggers an interrupt. (Suppose currently VM is running)
- VMExit! Jump to host Linux's interrupt handler (non-root -> root)
- Host Linux driver gets the character 'a'
- Host Linux returns to Qemu (kernel -> user)
- Qemu notices an I/O inputs in its main loop, updates its emulated keyboard device states (write 'a' to an emulated register of the emulated keyboard)
- Qemu puts a virtual interrupt in guest's VMCS, and use ioctl to resume VM (user -> kernel) (root -> non-root)

Control Flow of Pressing 'a' and Displaying 'a'

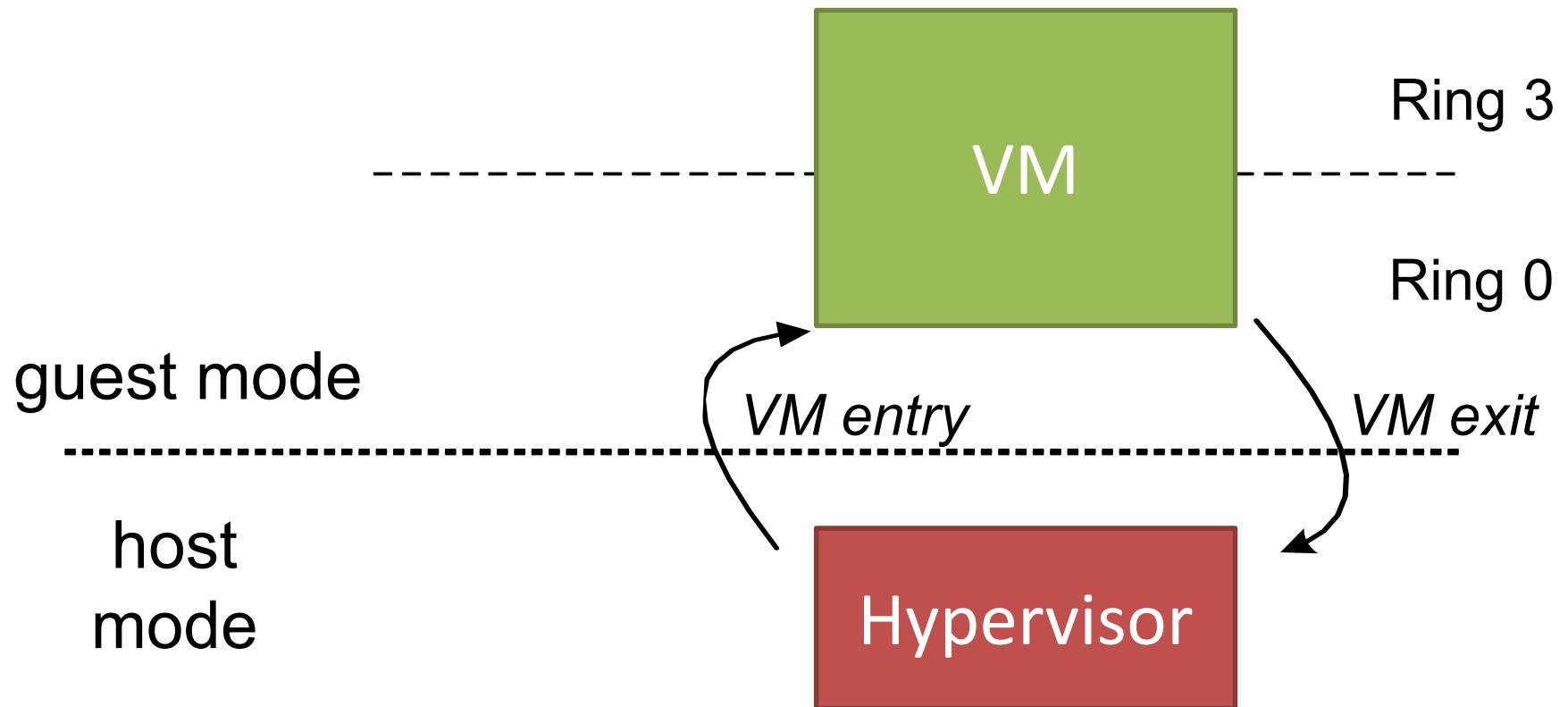
- VM finds the keyboard interrupt, jumps to interrupt handler (user -> kernel)
- VM driver starts to run, it tries to read keyboard's register to find out which key is pressed, by 'in 0x100 %eax', which will trap (non-root -> root)
- VMExit! Host Linux gives control to Qemu (kernel -> user)
- Qemu finds the reason is I/O instruction, then retrieves the instruction and emulates it, by reading the states of emulated keyboard, gets a value 'a' (which is just written by Qemu itself)
- Qemu updates VMCS->eax, and starts to run VM (user-> kernel) (root -> non-root)
- VM resumes and gets the value in %eax, which is 'a'
- Guest OS returns the value to user application (kernel -> user)

Virtualization Techs

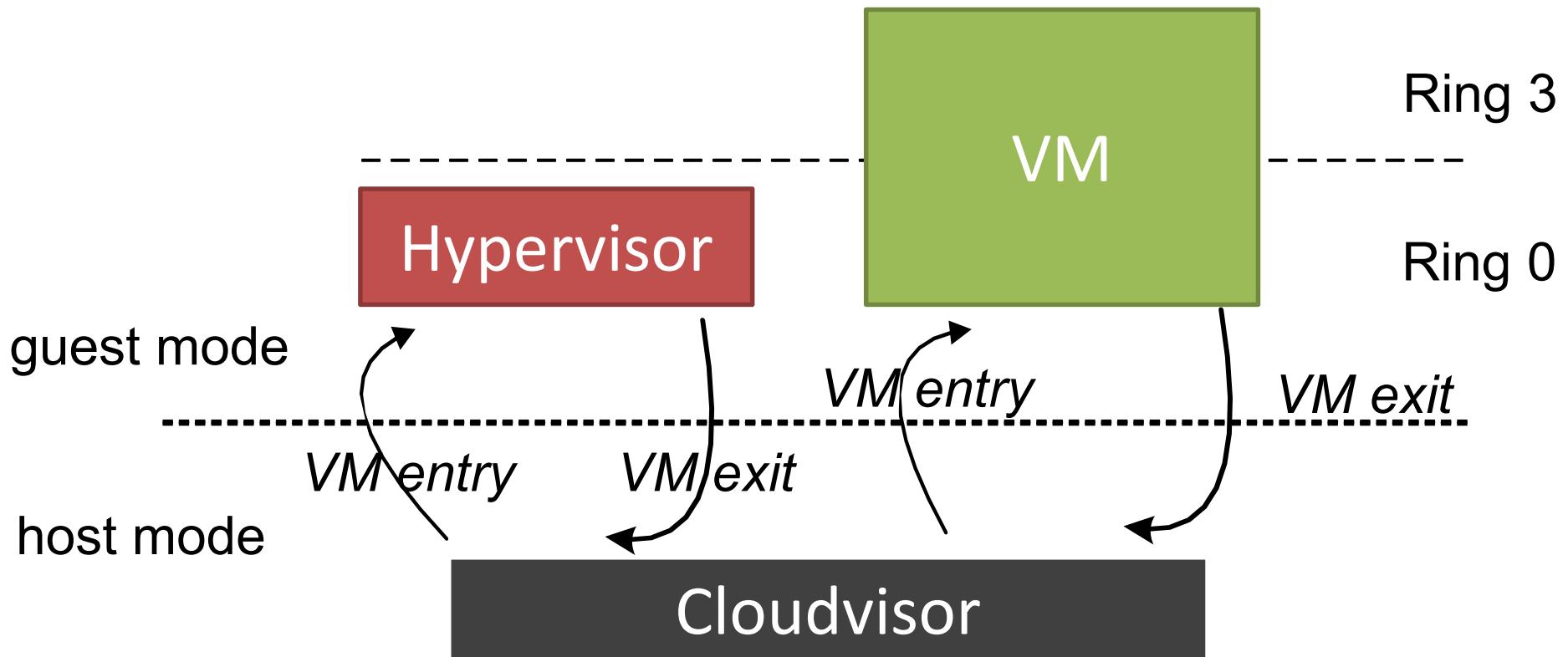
Virtualization	Software Solution	Hardware Solution
CPU	<ul style="list-style-type: none">• Trap & Emulate• Instruction interpretation• Binary translation• Para-virtualization: no 17 insns	<ul style="list-style-type: none">• VT-x<ul style="list-style-type: none">• Root / non-root mode• VMCS
Memory	<ul style="list-style-type: none">• Shadow page table• Separating page tables for U/K• Para-virtualization: Direct paging	<ul style="list-style-type: none">• EPT
I/O	<ul style="list-style-type: none">• Direct I/O• Device emulation• Para-virtualization: Front-end & back-end driver (e.g., virtio)	<ul style="list-style-type: none">• IOMMU• SR-IOV

FOR SECURITY

Virtualization Preliminary: VT-x



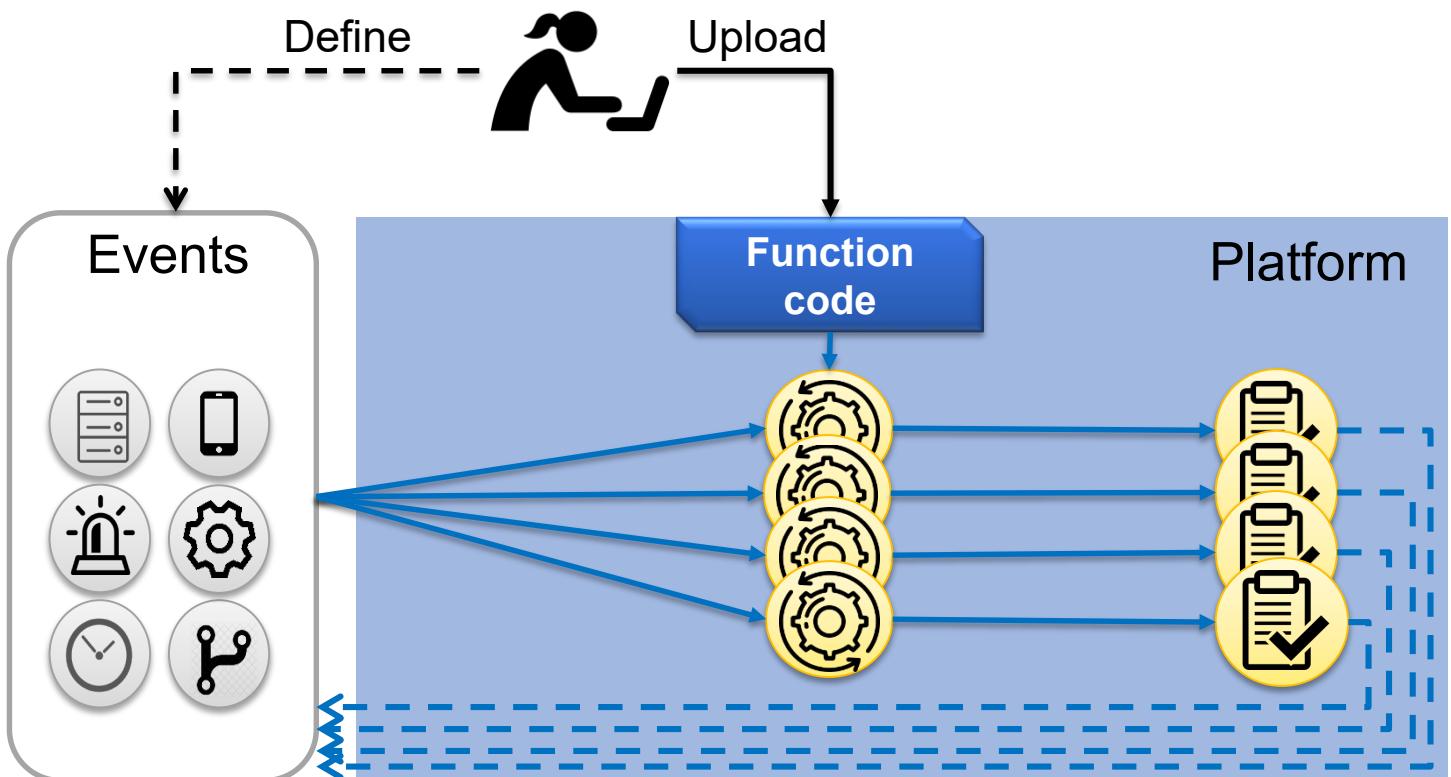
Interposition with CloudVisor



Serverless Computing

2019-6-11

Serverless Computing – Function as a Service (FaaS)





Advantages

- ✓ event-driven
 - stateless

Advantages

- ✓ event-driven
- ✓ auto-scale
 - instantiate new runtimes for new requests
 - destroy runtime when request finished
 - isolation between instances

Advantages

- ✓ event-driven
- ✓ auto-scale
- ✓ easier dev-ops
 - applications developers focus on application logic
 - no server management

Advantages

- ✓ event-driven
- ✓ auto-scale
- ✓ easier dev-ops
- ✓ fine-grained billing
 - cost-efficient for short-lasting applications
 - limited fine-grained (e.g. AWS Lambda: 100ms)

Serverless Computing

- c.f. Microservices

Serverless

- simple function
- auto-scale
- short running
- no server management

Microservice

- simple function
- scale, not automatic
- long running
- service manage required

Service Providers

	AWS Lambda	Azure Functions	Google Cloud Functions
Supported Languages	Node.js, Python, Java, C# and Go	C#, F#, Python, Java, Node.js, PHP	Node.js, Python
Persistent Storage	Completely stateless	Environment variables can be set, and can be stored into blob storage	Persistent storage available.
Max Code Size	50 MB compressed 250 MB uncompressed	None, you pay storage cost	100 MB compressed 500 MB uncompressed
Max Execution Time	900 secs.	600 secs.	540 secs.
Concurrent Functions	1K	depend on Triggers	1K
Event Triggers	AWS Services + API Gateway	Microsoft service requests + HTTP webhooks, APIM, Function proxy and bindings	HTTP + Cloud Pub/Sub + Cloud Storage + Direct Triggers

Source: RightScale

A Real-life Example – Dragon Quest

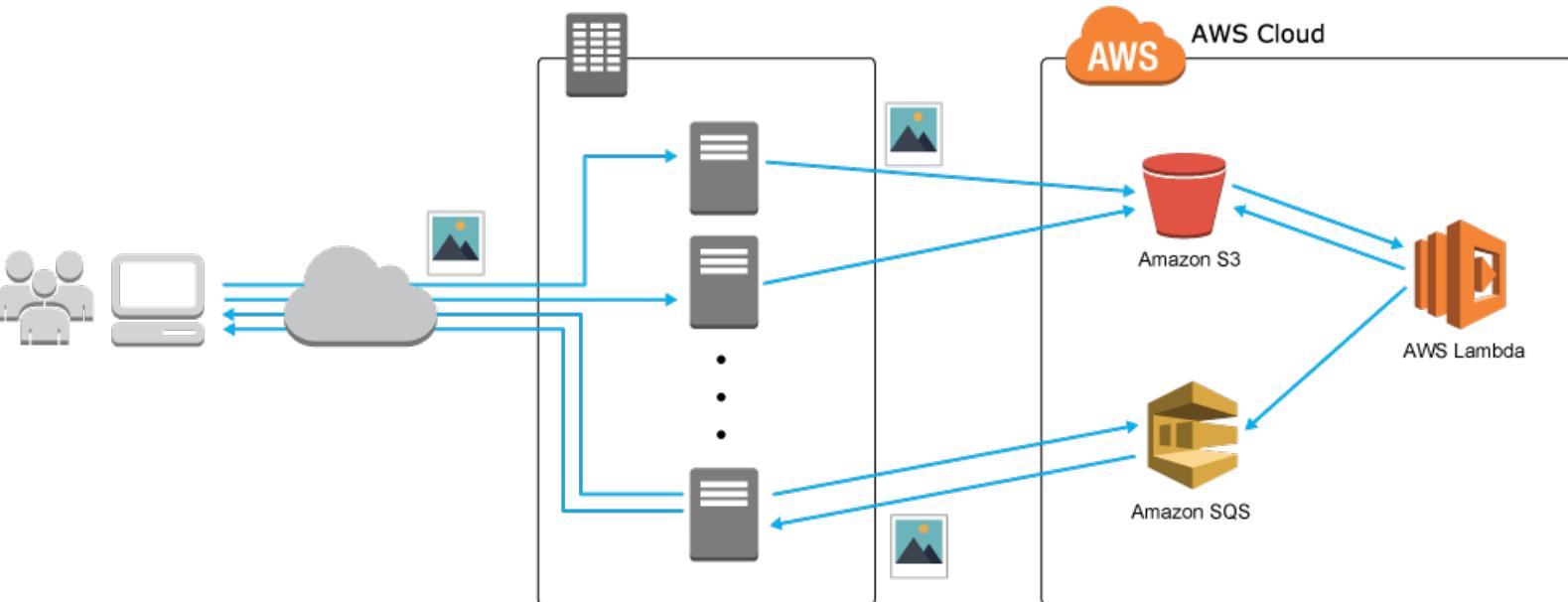
- **hundreds of thousands** of people play in the same world
- in-game screenshot function:
 - image processing (in server) includes: thumbnail creation, addition of a copyright watermark, etc.
 - resource intensive
 - 200 – 300 images per minute at usual, but as high as **6000 images per minute** on events such as New Year's Eve



in-game screenshots taken by customers

A Real-life Example – Dragon Quest

- image data uploaded to Amazon Simple Storage Server (S3)
- S3 event triggers AWS Lambda for image processing
- outputs from Amazon Simple Queue Service (SQS) are imported into the on-premises servers to save the processed data

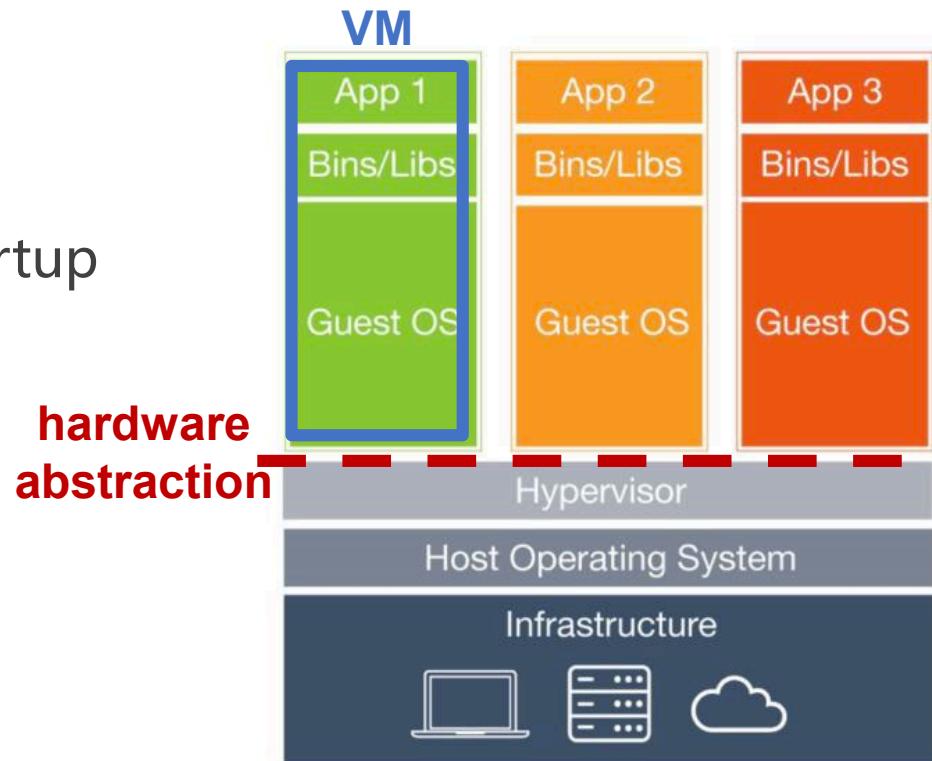


A Real-life Example – Dragon Quest

- **Benefits:**
 - reduced processing time
 - several hours -> 10 seconds
 - reduced processing cost
 - one twentieth of on premises
 - reduced infrastructure and labor cost
 - eliminating the labor associated with operations, maintenance, and server replacement

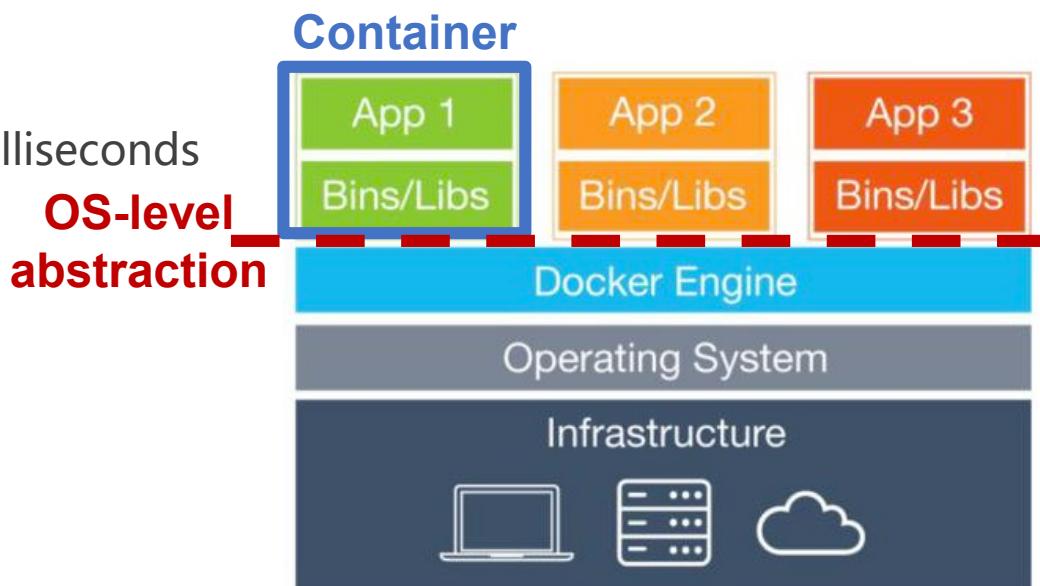
Sandbox Technologies – Virtual Machine

- each virtual machine with its own OS
- strong isolation
- heavy weight
- extremely slow startup



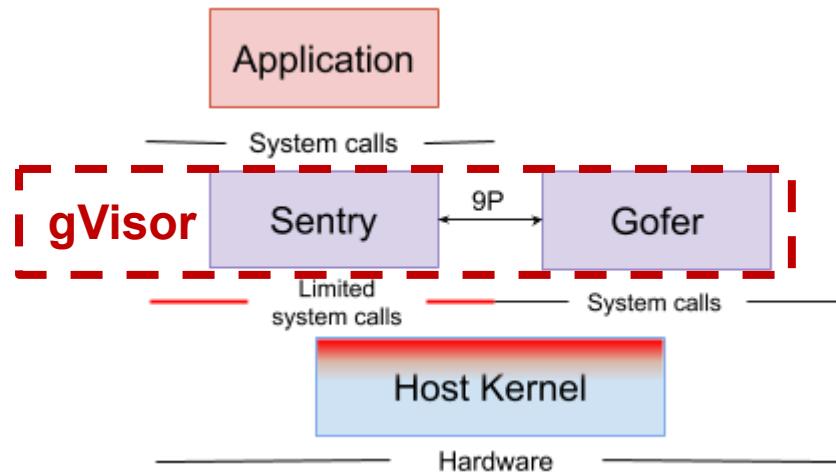
Sandbox Technologies – Container

- all containers share OS
- weak isolation
- light weight
- faster startup
 - hundreds of milliseconds



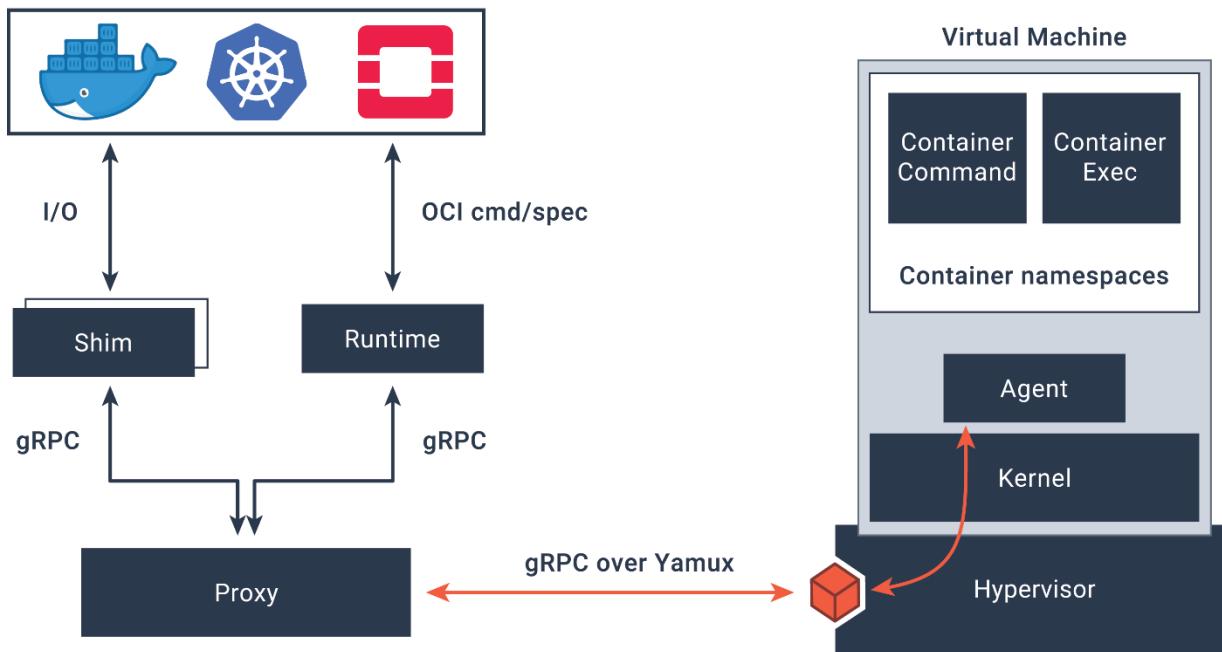
Sandbox Technologies – gVisor

- user-space kernel for containers, with stronger isolation
- Sentry handles most system calls from application
- Gofer handles I/O operations
- Sentry and Gofer communicate via 9P protocol



Sandbox Technologies – Kata Container

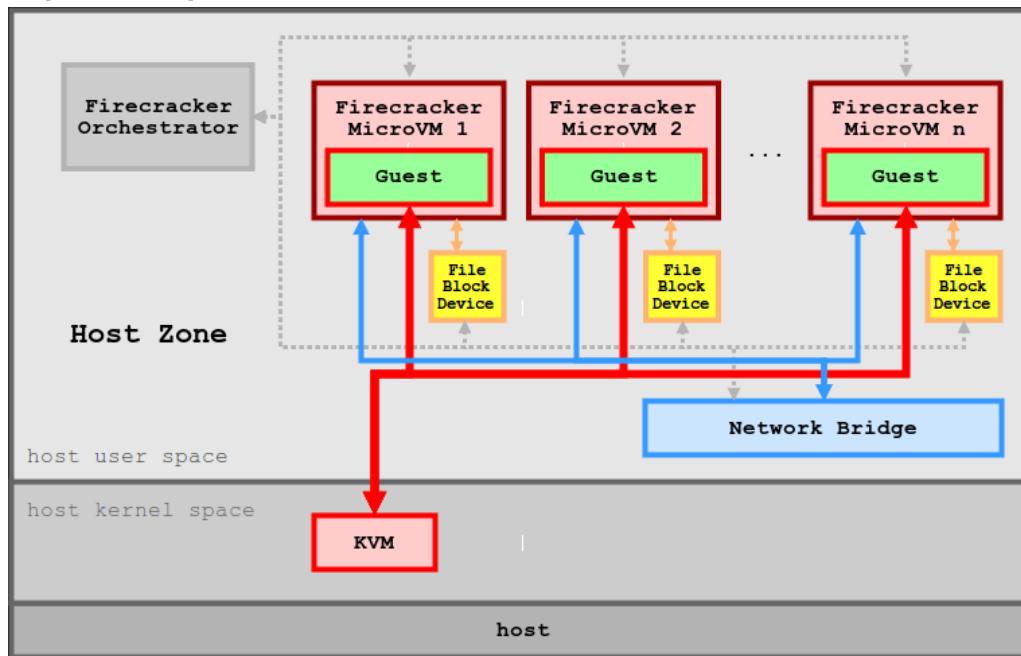
- secure container runtime with light-weight VM



Hypervisor serial interface

Sandbox Technologies – Firecracker

- microVM using KVM
- steady and fast startup (125ms)
- employed by AWS Lambda



Orchestration - Kubernetes



- a system for running and coordinating containers across a cluster of machines
- provides predictability, scalability, and high availability
- a natural fit for serverless platform implementation

Orchestration - Kubernetes

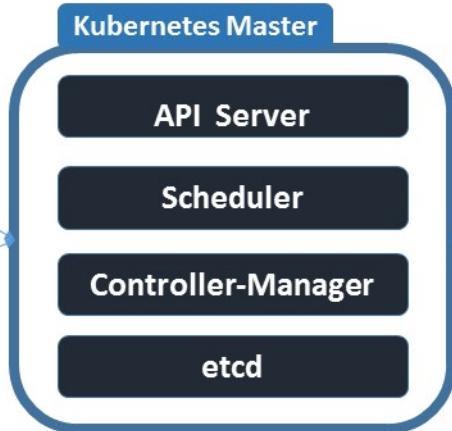


KUBERNETES ARCHITECTURE

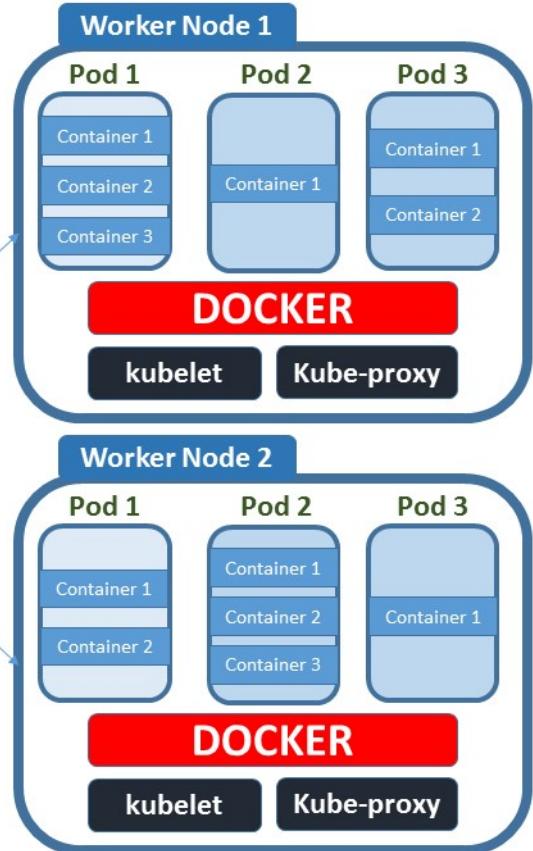
User Interface



kubectl

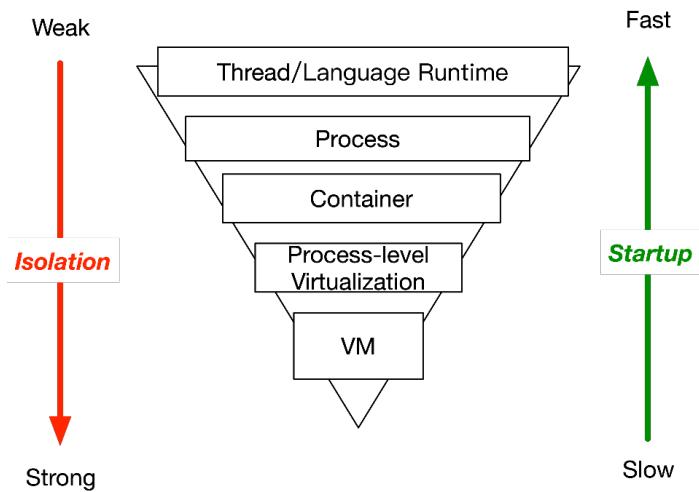


www.learnitguide.net



Challenges – Latency

- fast start-up and shut-down
- fast communication
- tail latency
- tradeoff with isolation level





Challenges – Resource

- resource footprint
- resource allocation/assignment speed
- resource utilization

Challenges – State handling

- real-life applications are often stateful
- intermediate states passed between chained functions
- via external storage (e.g. Amazon S3)
 - storage also needs to be fast, auto-scale, fine-grained

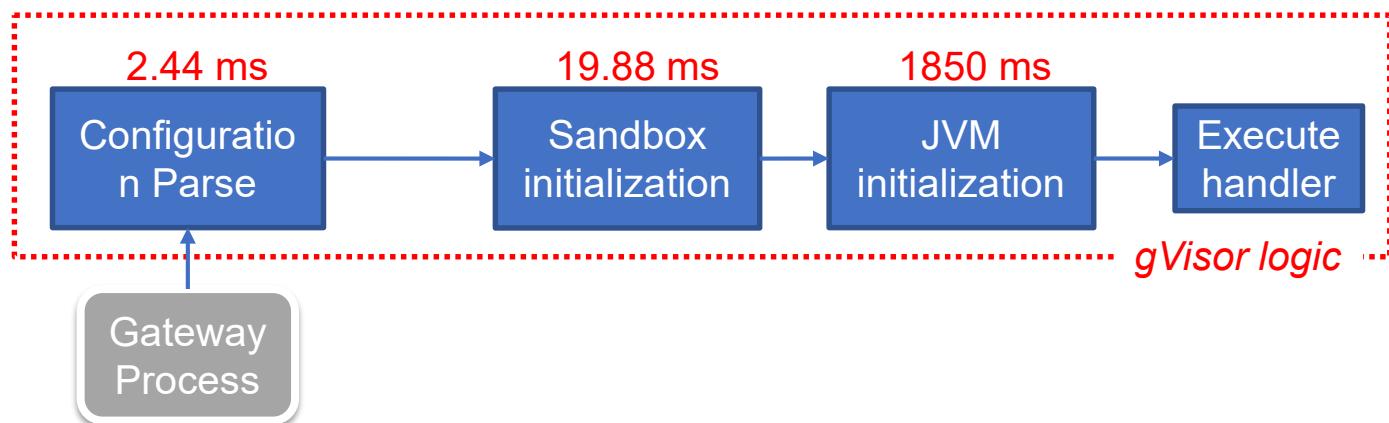


Latency introduced by initialization

INIT-LESS LOADING

Startup Latency Breakdown

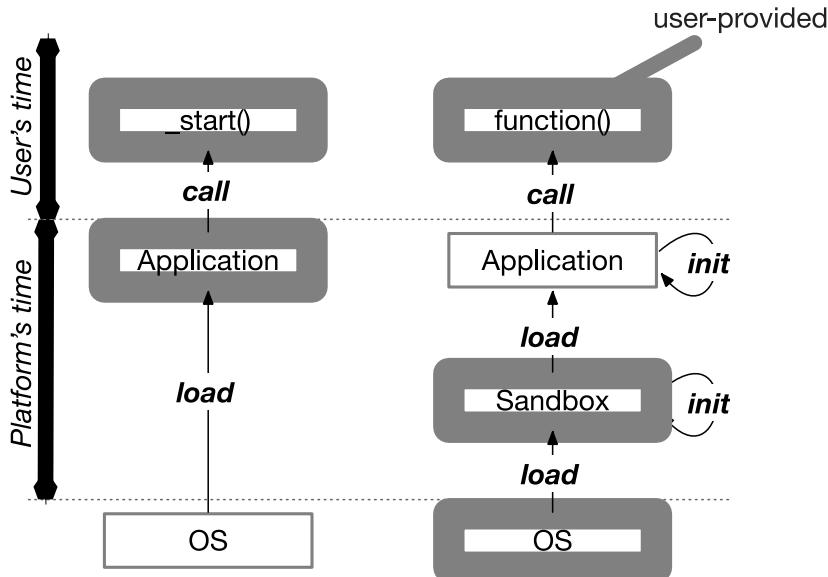
- **gVisor+SPECJbb as a case study**
 - Runtime sandbox initialization: 22.33 ms
 - Application initialization: **1850 ms**



The major of startup latency comes from **application initialization!**

Startup Latency Breakdown

- Initializations cause high startup latency for serverless applications

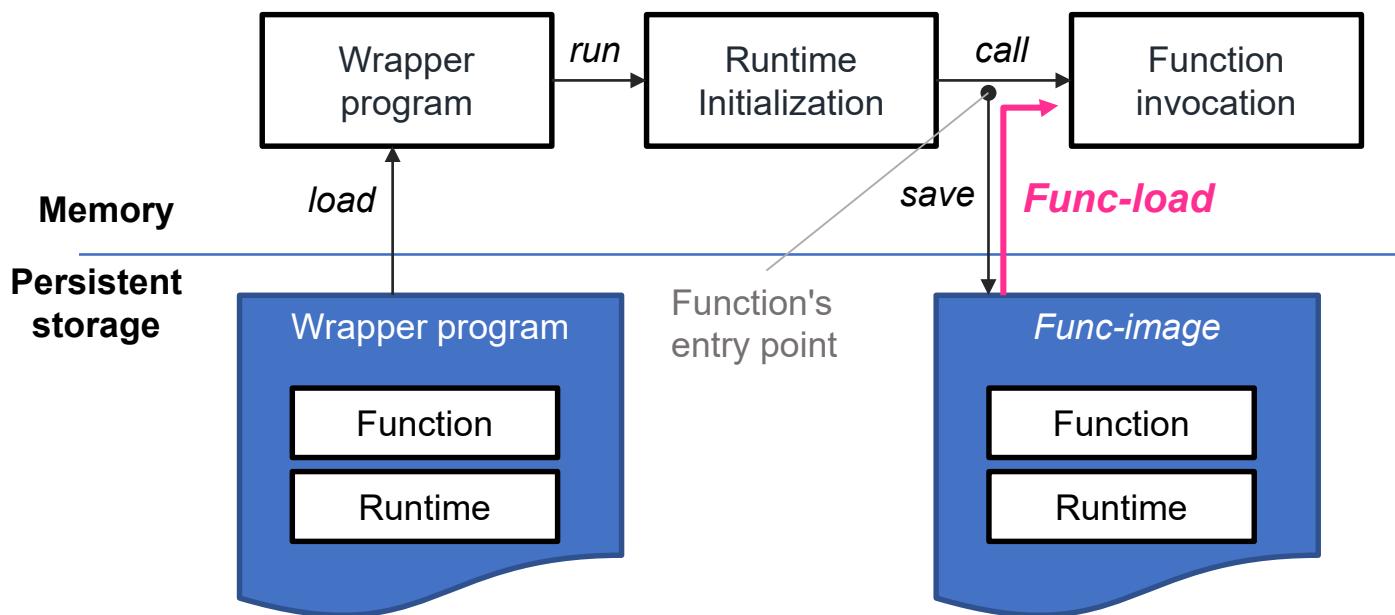


Can we eliminate the initialization phase?

(a) Traditional application (b) Serverless computing

Overview

- *func-image compilation and func-load*



Strawman Implementation

- *strawman*:
 - *Save (compilation)*: checkpoint
 - *Func-load*: restore
- *func-entry point*

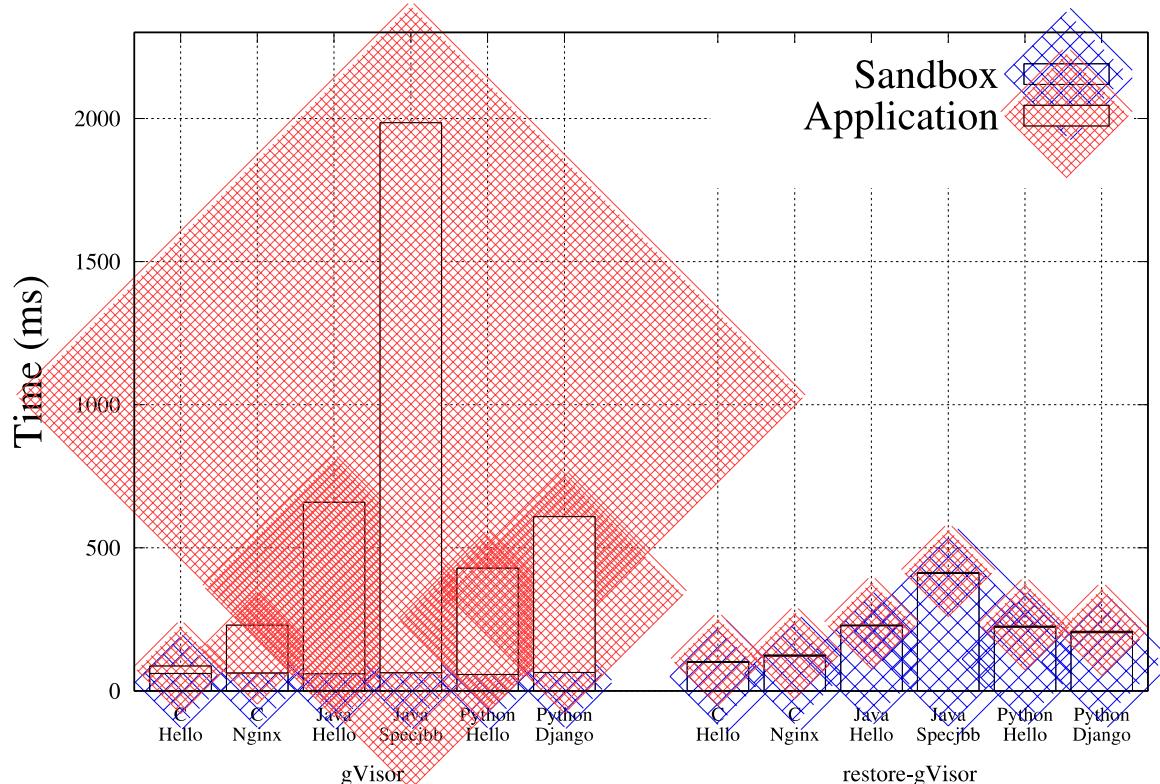
Listing 1 Example serverless function code of the init-less design.

```
1: package com.example.fn;  
2: public class HelloFunction {  
3:     @FunctionEntryPoint  
    public String hello(String name) {  
        return "Hello " + name;  
    }  
}
```

Serverless computing platforms	Default <i>func-entry point</i>	
AWS Lambda	before the invocation of <i>handler</i>	name + "!";
Oracle FN Project	before the invocation of <i>cmd</i>	
Google Cloud Functions	index.js/function.js/main in package.json	
IBM Cloud Function	before main() function	
Microsoft Azure Functions	before invocation of <i>entryPoint</i>	

Evaluation of Strawman Implementation

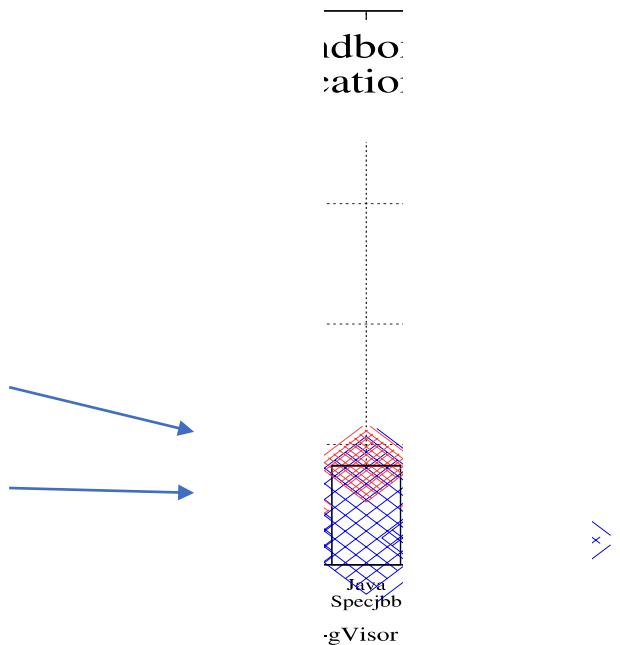
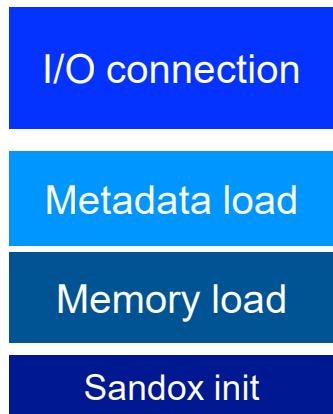
- Strawman: gVisor checkpoint/restore + *func-entry point*



2x-5x improvement than gVisor

Evaluation of Strawman Implementation

- Strawman: gVisor checkpoint/restore + *func-entry point*



Still can be improved!