Introduction & 8 Important Problems in Modern Operating Systems

Yubin Xia

Institute of Parallel and Distributed Systems (IPADS)

Shanghai Jiao Tong University

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

Outline

Why Choose This Course?

Course information

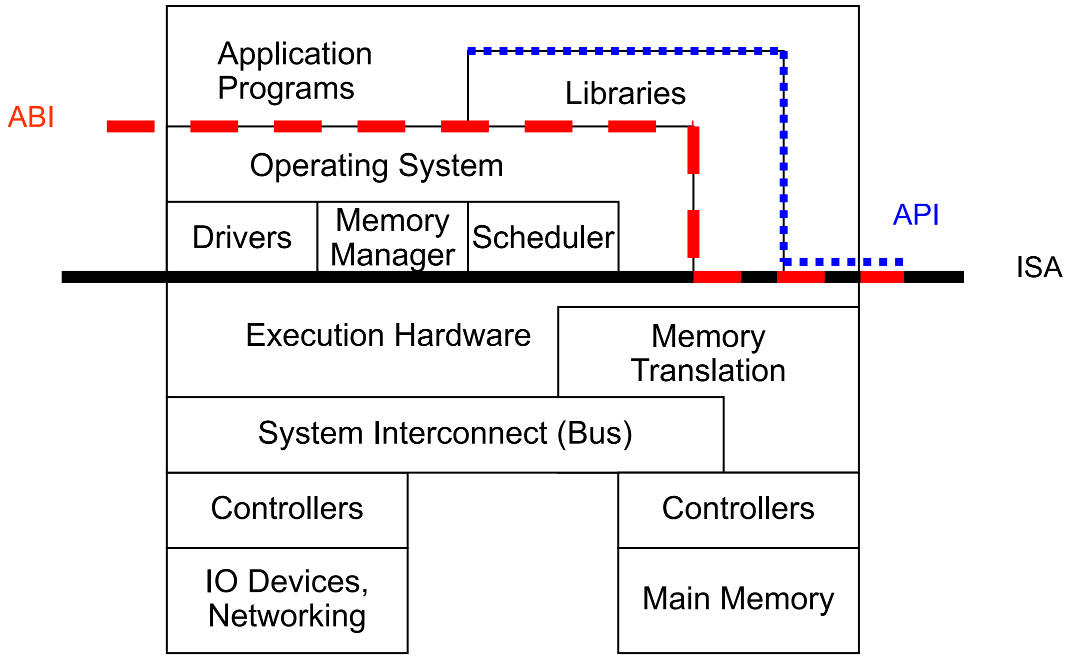
8 important problems of OS

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 API?
  - 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

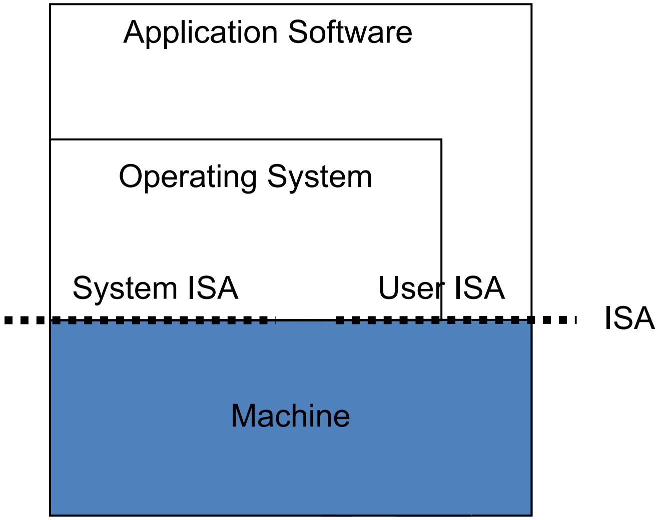


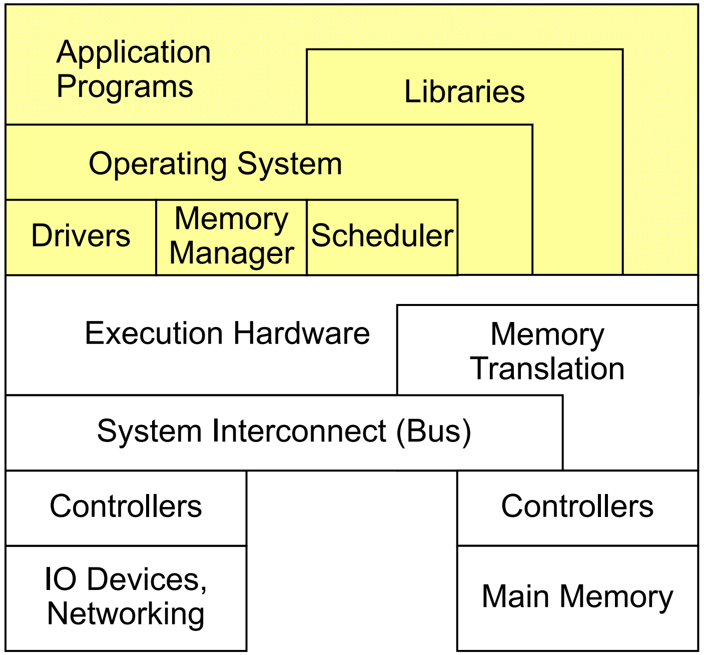
Architecture, Implementation Layers

- 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





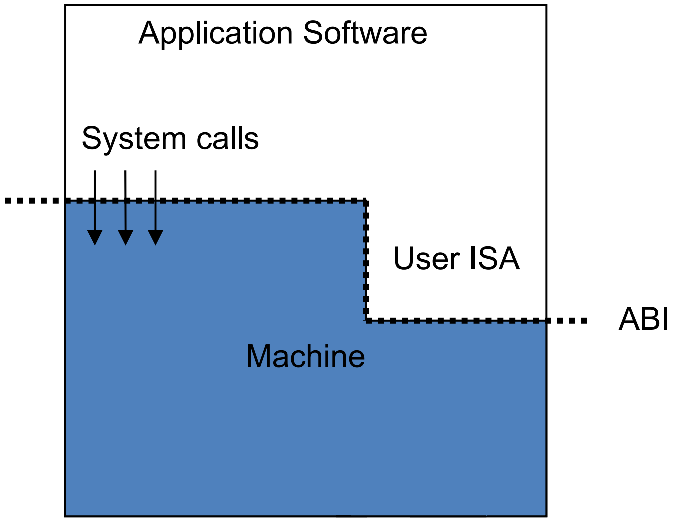
Question: ISA Emulation?

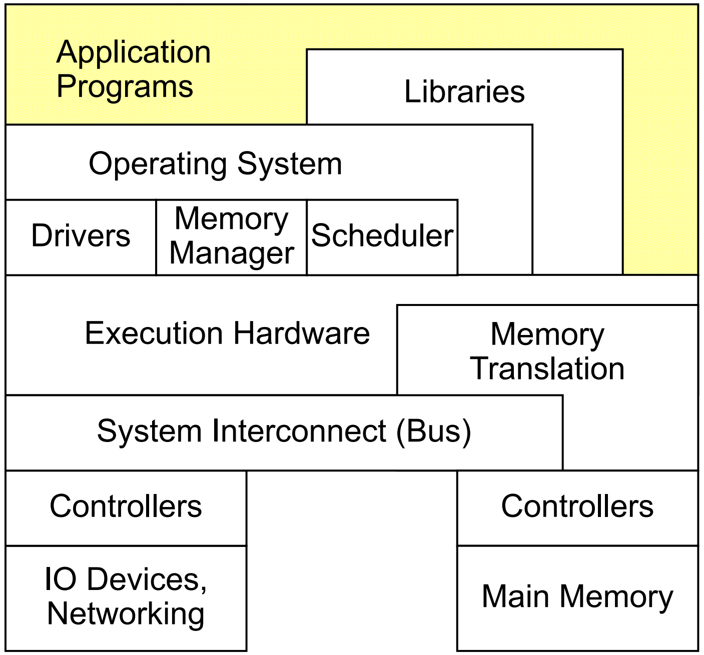
Architecture, Implementation Layers

- 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





Question: 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
  - **`clib`** in Unix environment : supports the UNIX/C programming language

Why Study OS?

Operating systems are 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

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 from this course
  - Become more effective 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 got 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?

Course information

8 problems of OS

Faculty Information

Instructor

XIA Yubin 夏虞斌 CHEN Rong 陈榕

Office: Room 3-402, Software Building

Office hour: 1:00pm~2:00pm, Friday

Contact:

xiayubin@sjtu.edu.cn, rongchen@sjtu.edu.cn

TA

DONG Mingkai	*MinGKaiDong@gmail.com*
YU Qianqian	*kiki_yu@foxmail.com*
ZHENG Yang	*tomsun.0.7@gmail.com*

Reference Books

xv6: a simple, Unix-like teaching operating system.
By Russ Cox, Frans Kaashoek, Robert Morris.
Online

Understanding Linux Kernel.

Labs

Implement JOS: adopted from MIT 6.828 course

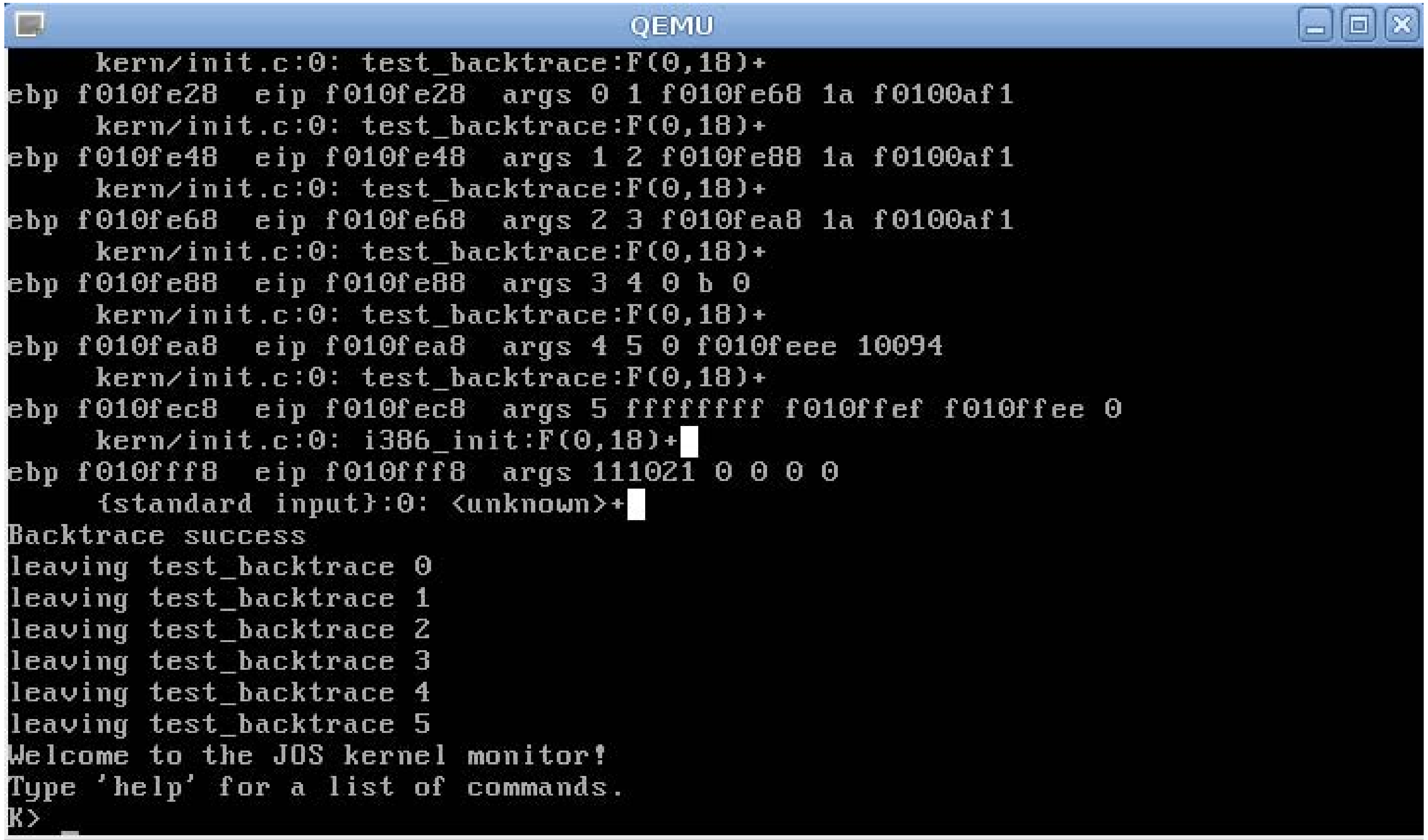
- Lab1: Booting a PC
- Lab2: Memory Management
- Lab3: User Environments
- Lab4: Preemptive Multitasking

*Lab 5: XXX

*Lab 6: XXX

Lab 1 Boot a PC

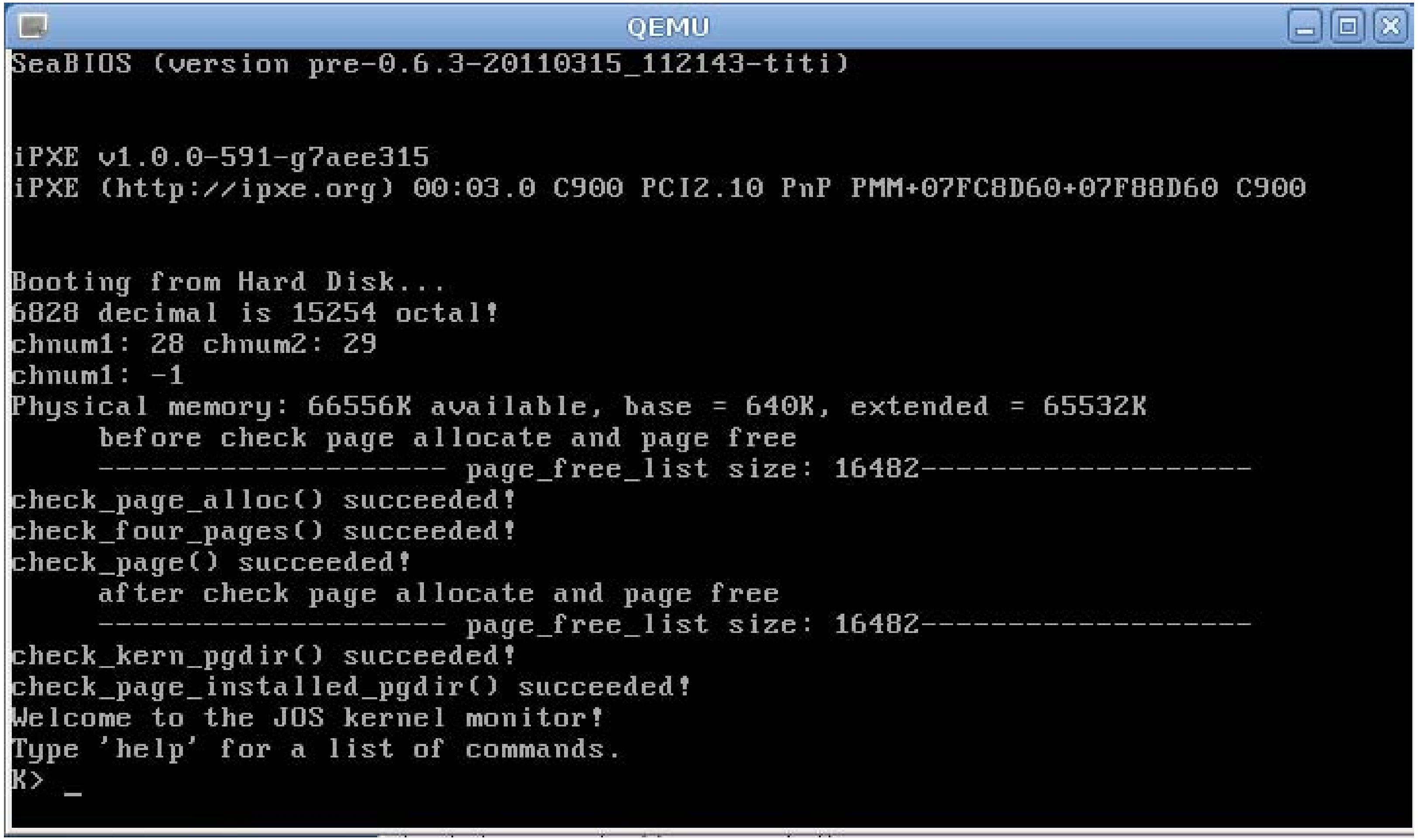
Familiar with PC boot process



```
     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 f010ffee 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

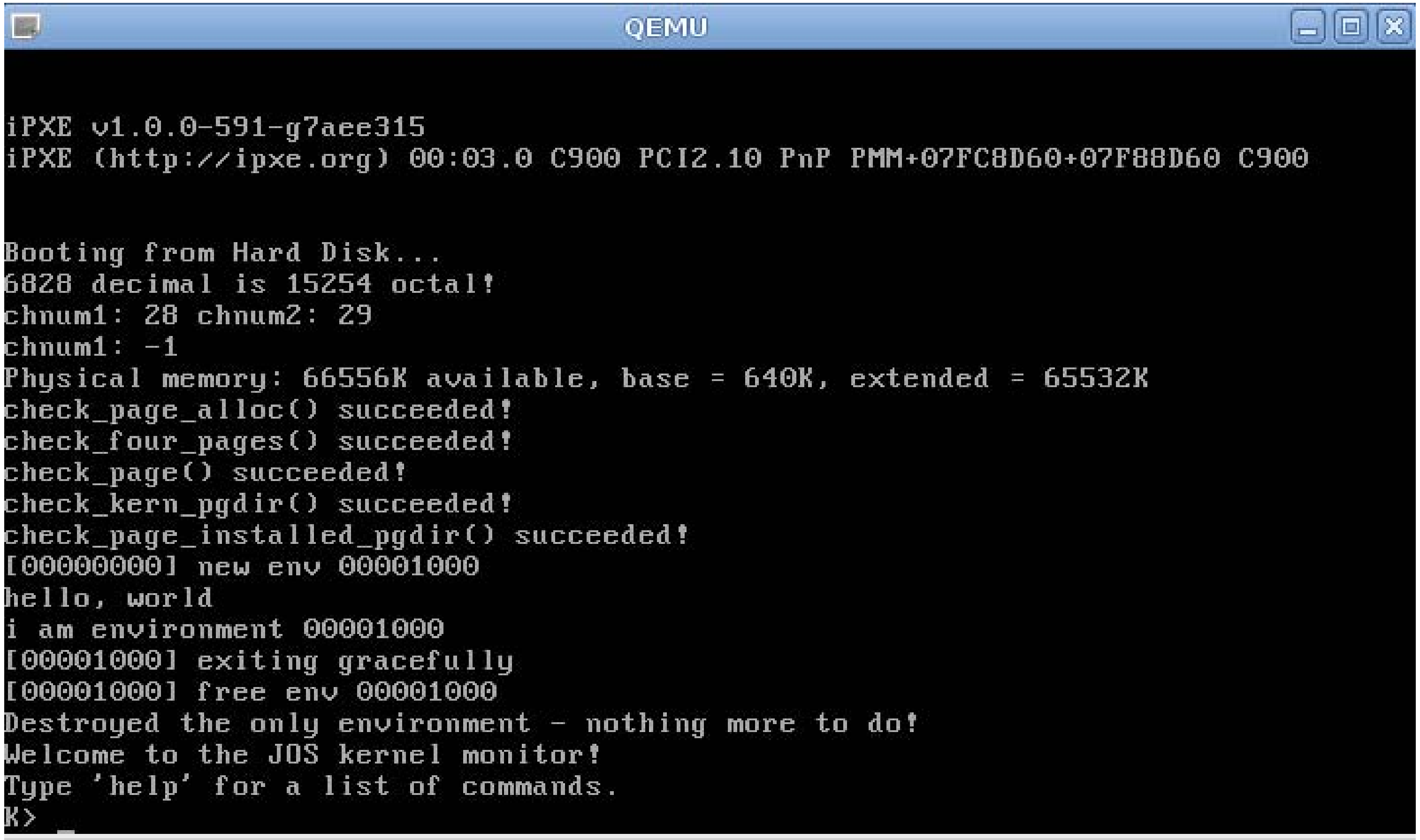


```
SeaBIOS (version pre-0.6.3-20110315_112143-titi)


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

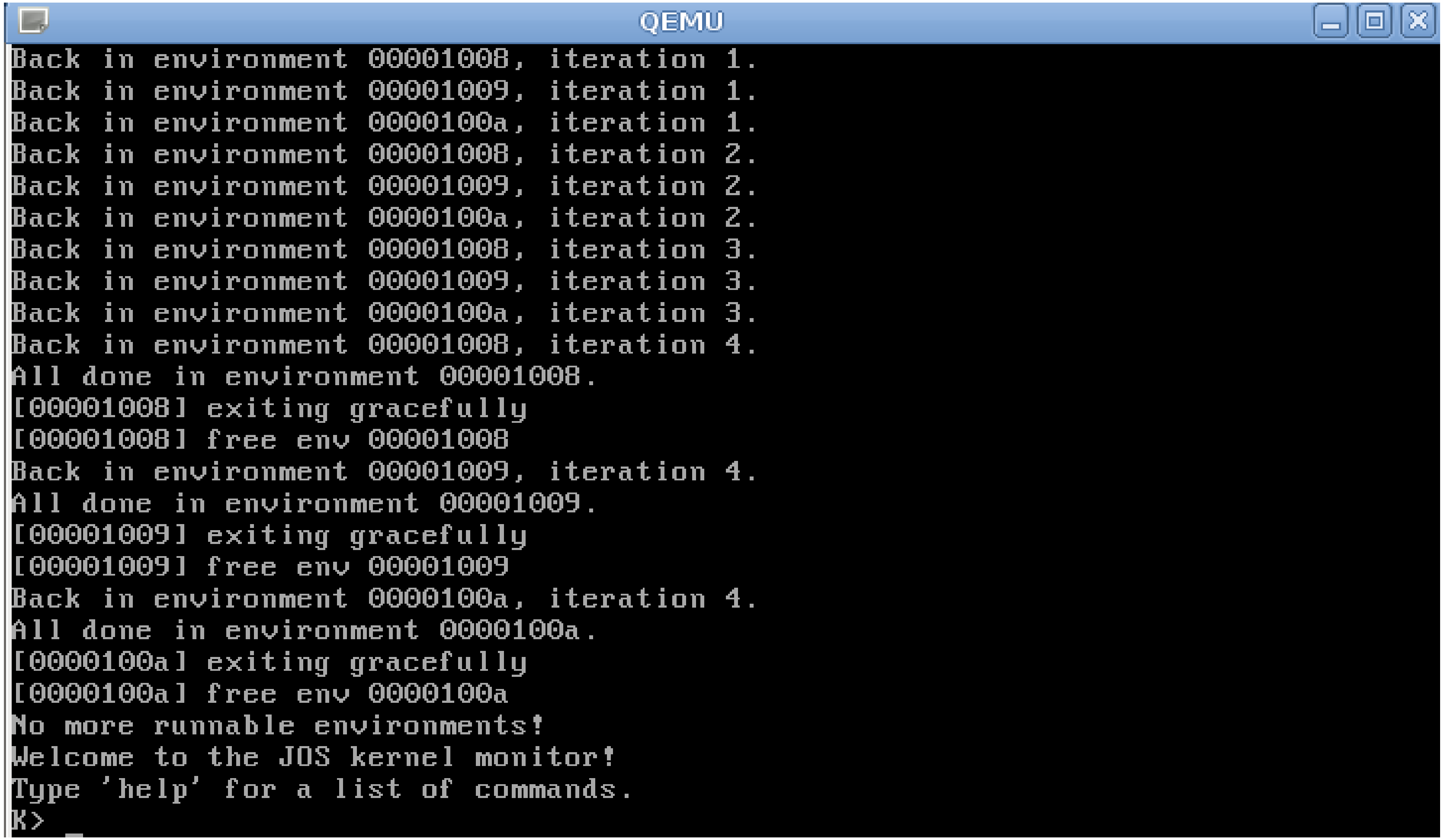


```
iPXE v1.0.0-591-g7aee315
iPXE (http://ipxe.org) 00:03.0 C900 PCI2.10 PnP PMM+07FC8D60+07F88D60 C900


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



```
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 in BBS/forum

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

8 IMPORTANT PROBLEMS IN OPERATING SYSTEMS

8 Important Problems
(Not a Rank)

Scale Up

Security & Privacy

Energy Efficiency

Mobility

Write Correct (Parallel) Code

Scale Out

Non-Volatile Storage

Virtualization

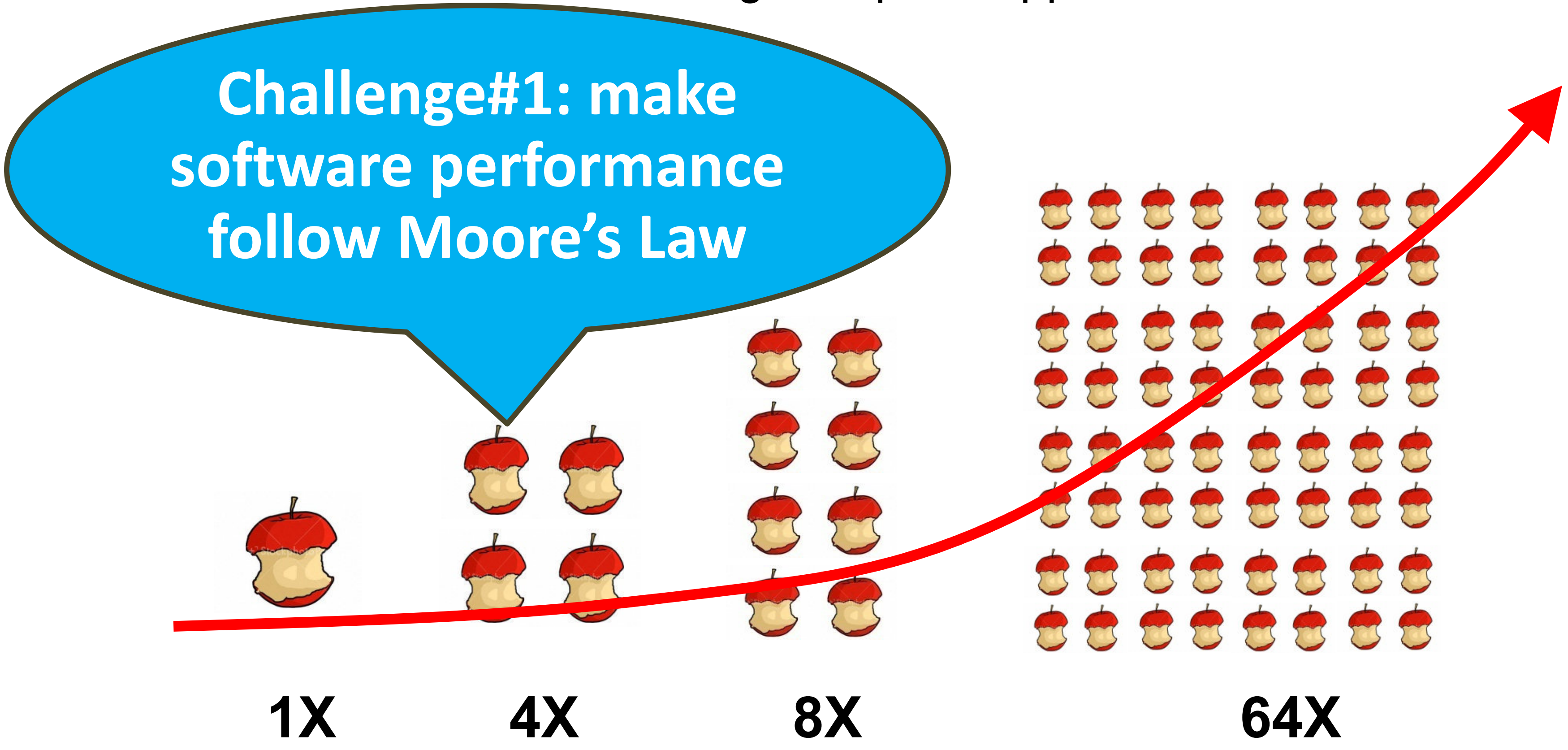
#1: Scale up (or Performance Scalability)

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 manages 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

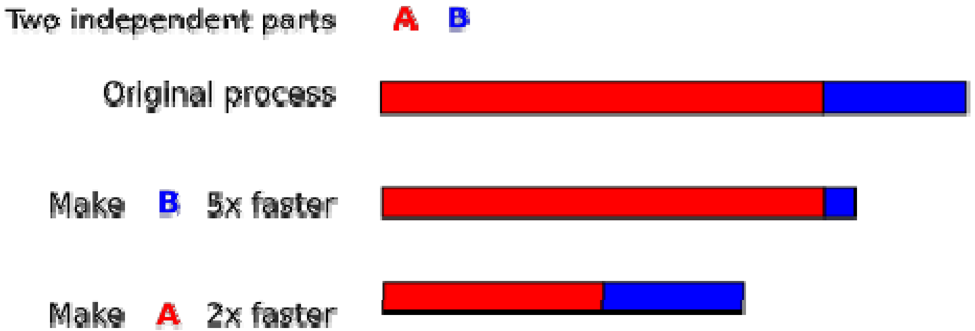
What is scalability?

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, ...)

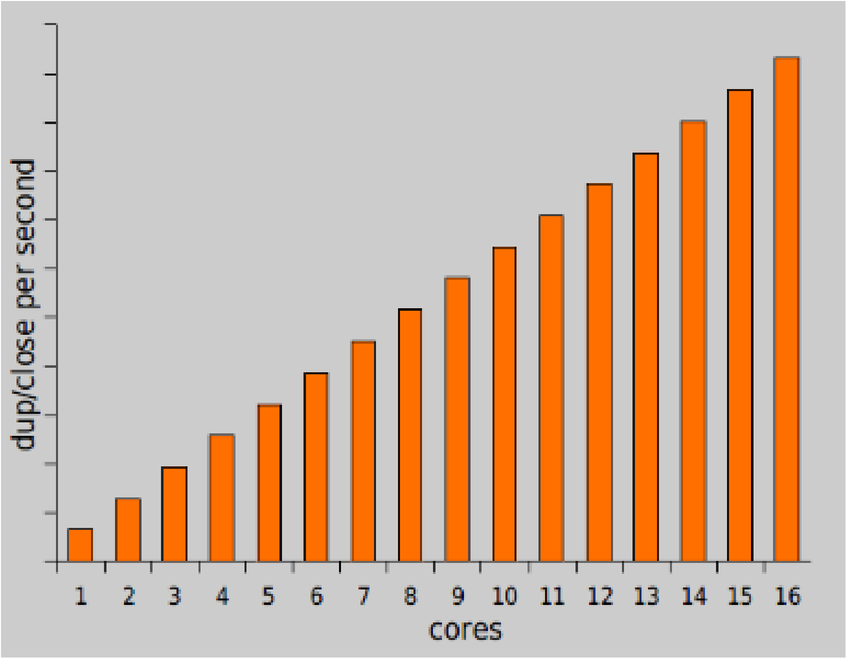
$$\frac{1}{(1-P)+\frac{P}{S}}$$

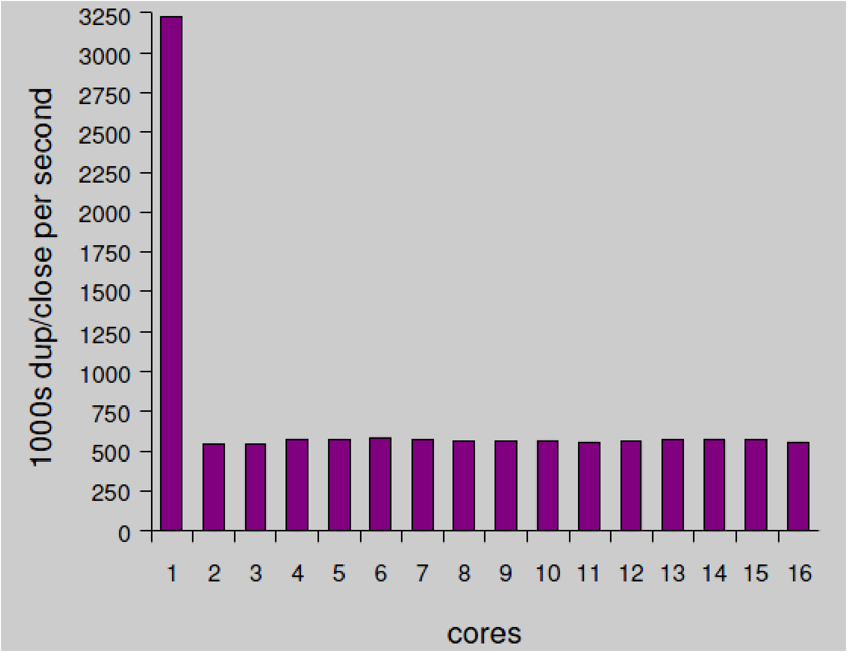


Motivating example: file descriptors

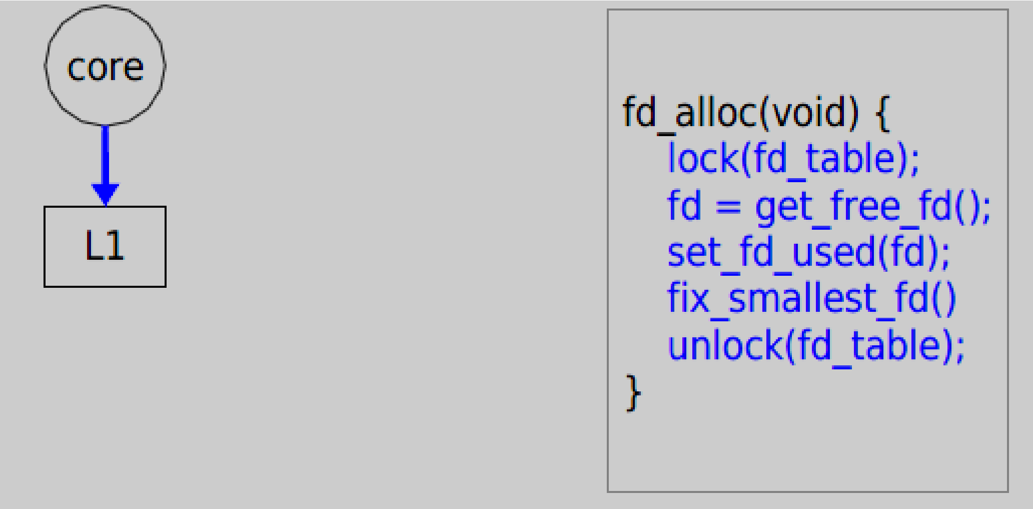
Ideal FD performance graph

Actual FD performance





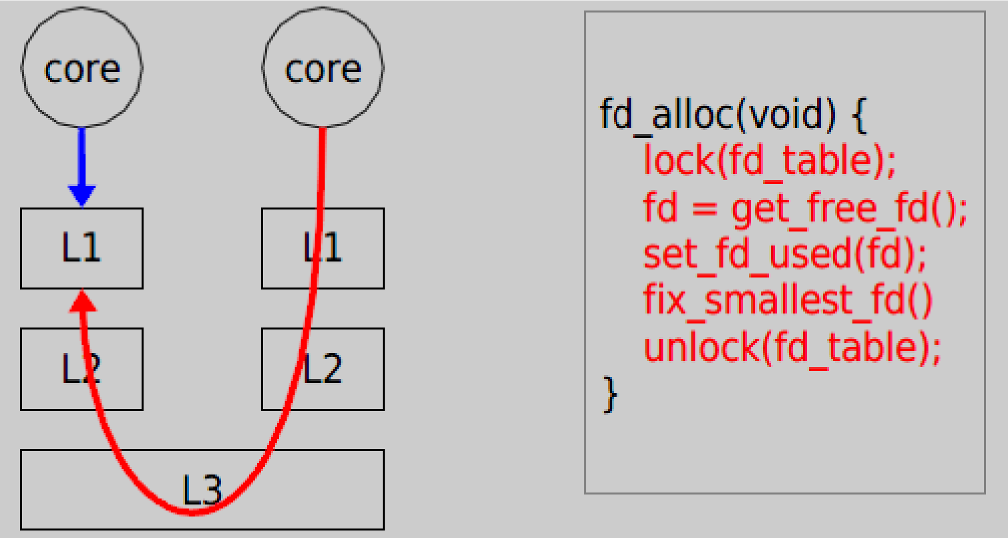
Why throughput drops?



```
fd_alloc(void) {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd()
    unlock(fd_table);
}
```

Load fd_table data from L1 in 3 cycles.

Why throughput drops?

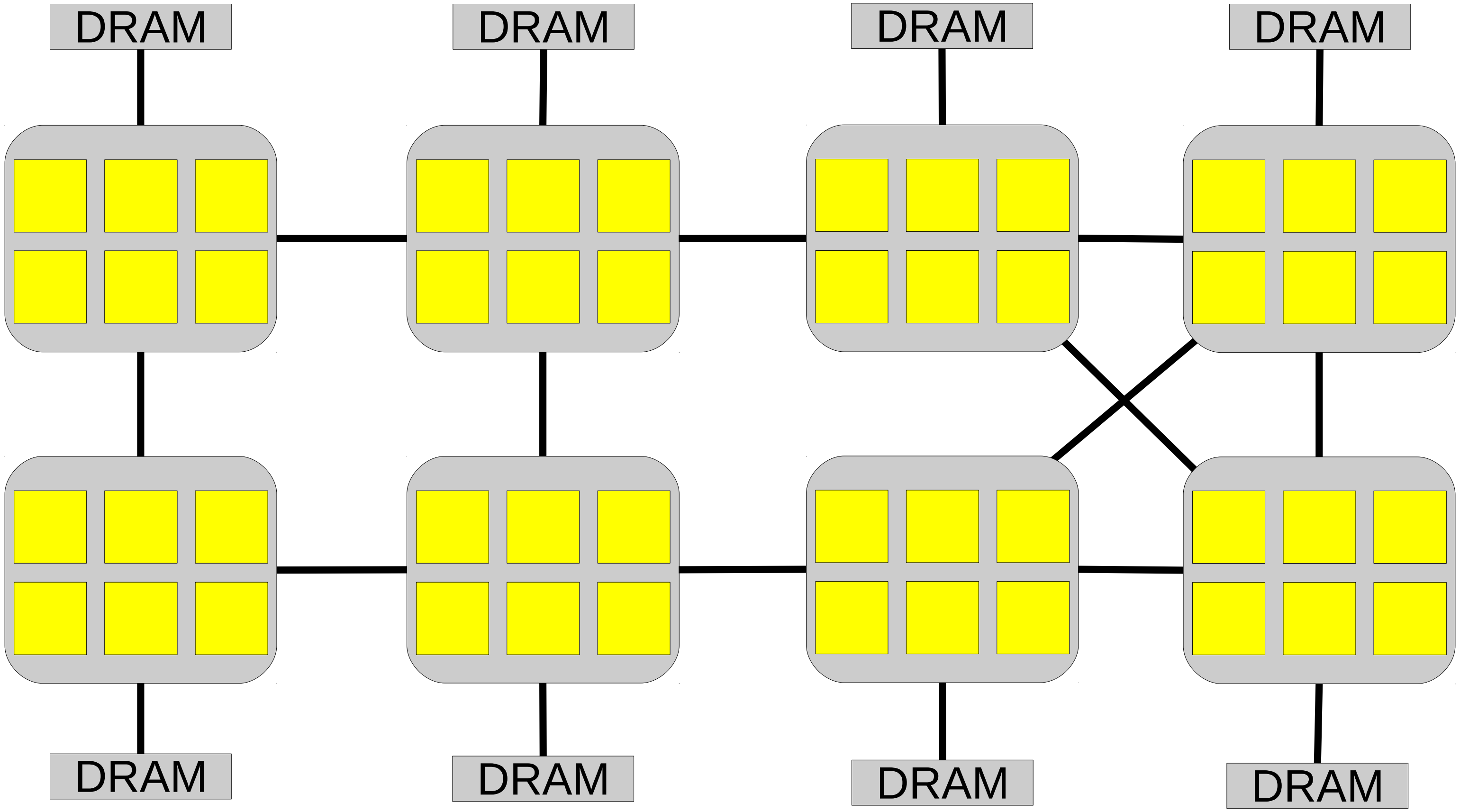


```
fd_alloc(void) {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd()
    unlock(fd_table);
}
```

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

You will fill this..

Issue#2: Security & Privacy

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

Example: The do_brk() Bug

```
Asmlinkage unsigned long sys_brk(unsigned long brk) {
…
/* OK, looks good – let it rip. */
-if (do_mmap(NULL, oldbrk, newbrk-oldbrk,
-PROT_READ|PROT_WRITE|PROT_EXEC,
-MAP_FIXED|MAP_PRIVATE, 0) != oldbrk)
+if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
goto out;
```

- do_mmap() checked things properly, do_brk() removed almost all checking
- Purpose of do_brk() was to speed up anonymous mmaps from sys_brk()

The do_brk() Fix

```
if (!len)
return addr;

+ if ((addr + len)) > TASK_SIZE || (addr + len) < addr)
+ return –EINVAL;
+
```

- The TASK_SIZE is typically set to 0xc0000000, the start of kernel memory

Issue#3: Power Efficiency

Power Efficiency

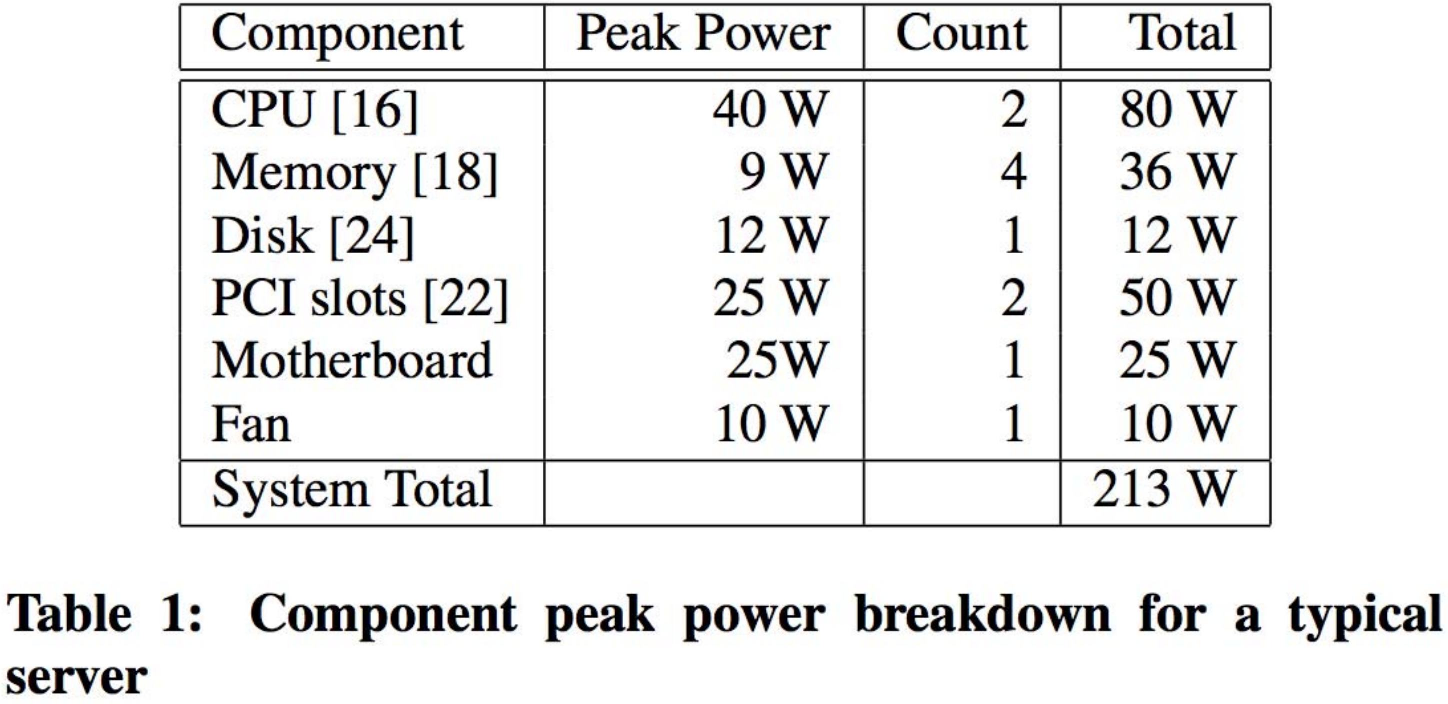
Greener World, Better Life!

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

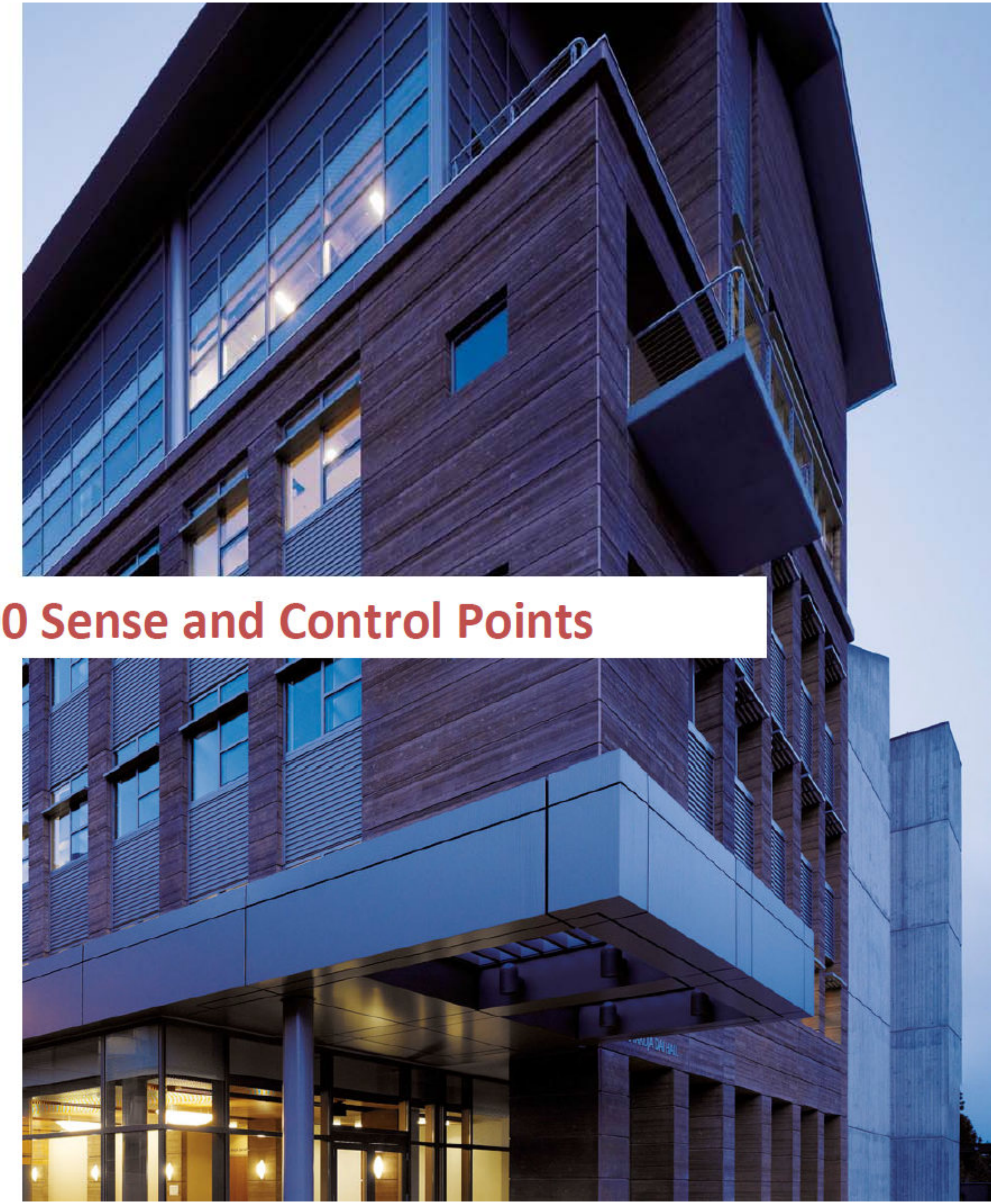
12 Variable Speed Fans

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

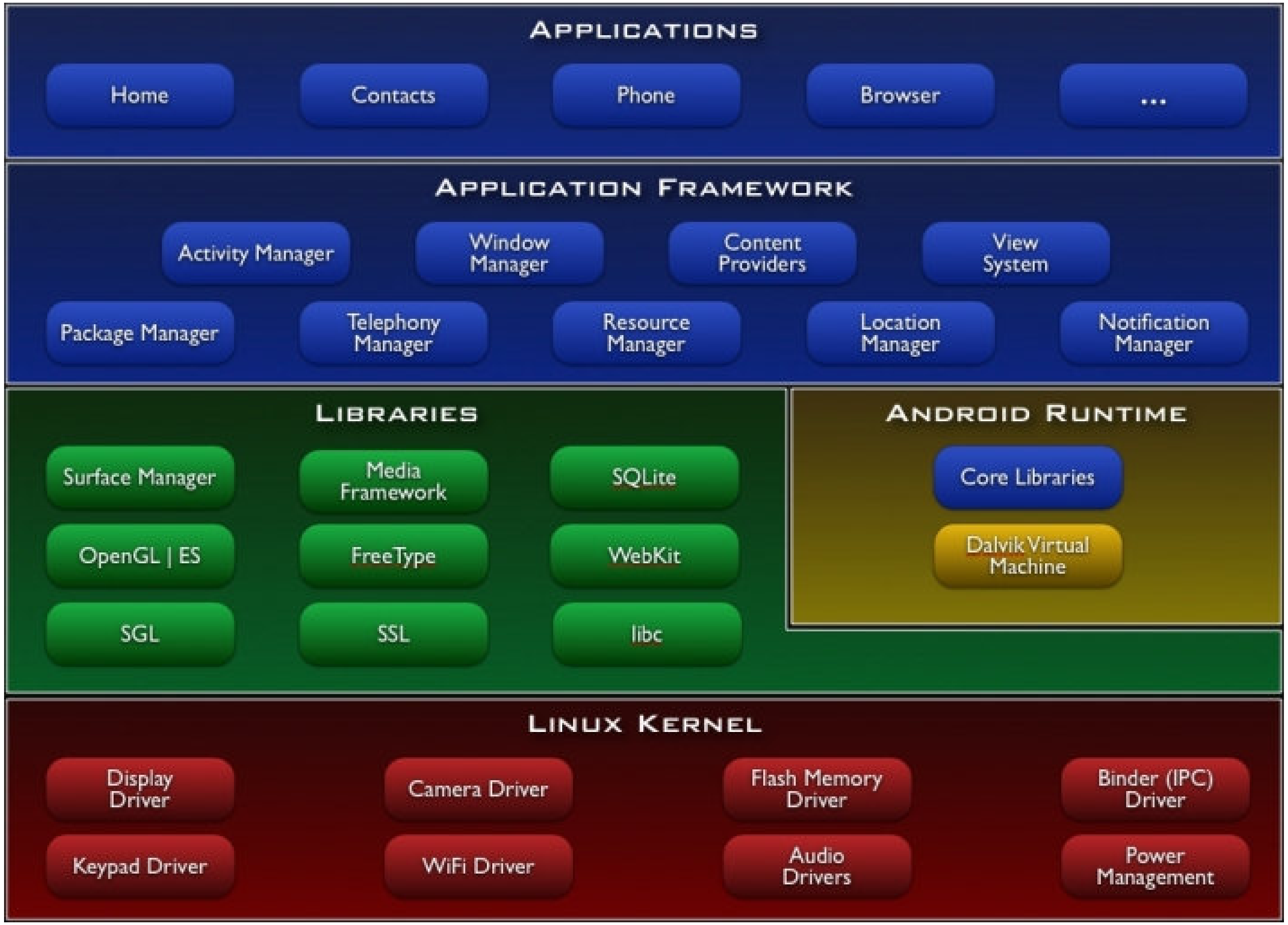
Palm WebOS

Windows Phone 7

Android Software Stack

- Linux kernel
- Libraries
- Android run time
  - core libraries
  - Dalvik virtual machine
- application layer
- application protocol

Android Architecture



APPLICATIONS

Home, Contacts, Phone, Browser, ...

APPLICATION FRAMEWORK

Activity Manager, Window Manager, Content Providers, View System

Package Manager, Telephony Manager, Resource Manager, Location Manager, Notification Manager

LIBRARIES

Surface Manager, Media Framework, SQLite

OpenGL | ES, FreeType, WebKit

SGL, SSL, libc

ANDROID RUNTIME

Core Libraries

Dalvik Virtual Machine

LINUX KERNEL

Display Driver, Camera Driver, Flash Memory Driver, Binder (IPC) Driver

Keypad Driver, WiFi Driver, Audio Drivers, Power Management

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 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

```
if (thd->proc_info) {
    …

    fputs(thd->proc_info, …)

    …
}
```

Thread 2

```
thd->proc_info = NULL;
```

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 reproduce

Provide better programming interfaces to ease programming

- Operating transactions

Issue#6: Scale Out (use distributed systems)

The Datacenter as a Computer

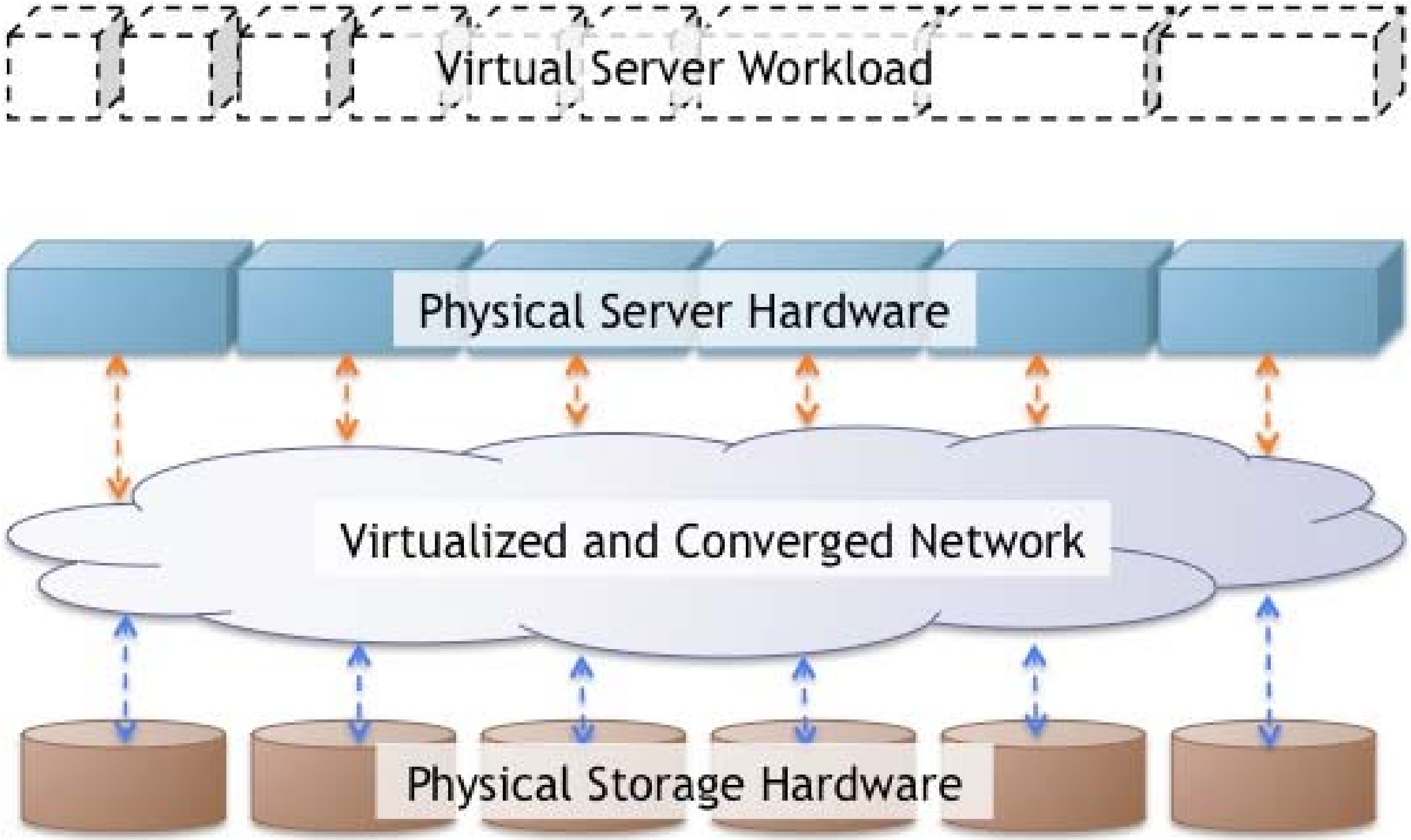
An Introduction to the Design of Warehouse-Scale Machines

Luiz André Barroso and Urs Hölzle
Google Inc.

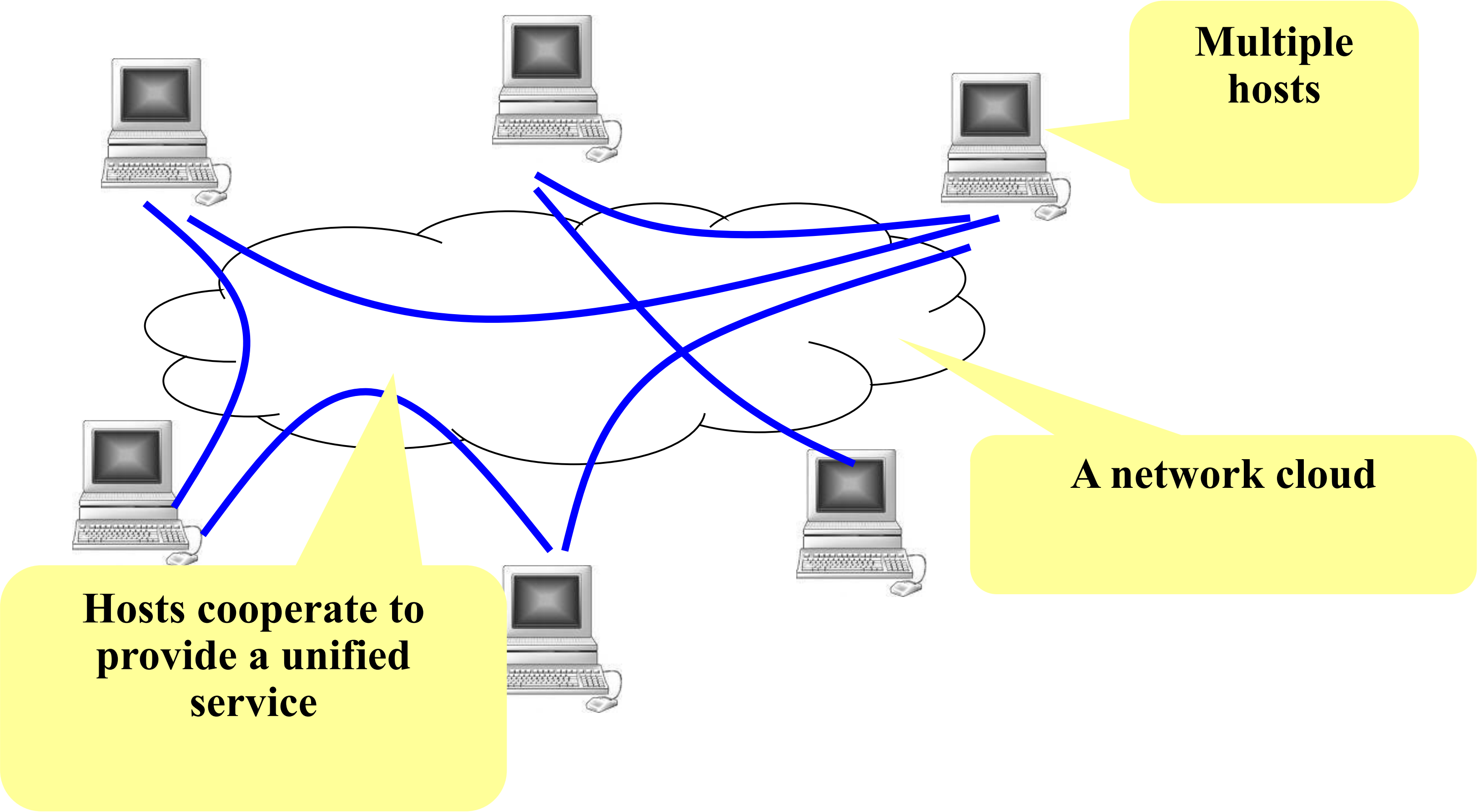
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE # 6

MORGAN & CLAYPOOL PUBLISHERS

The Virtual Datacenter Dream



What are distributed systems?



Multiple hosts

A network cloud

Hosts cooperate to provide a unified service

- 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

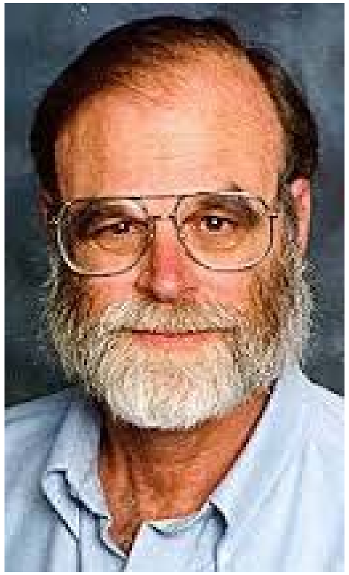
Tape is Dead
Disk is Tape
Flash is Disk
RAM Locality is King

Jim Gray

Microsoft

December 2006

In memory of Jim Gray



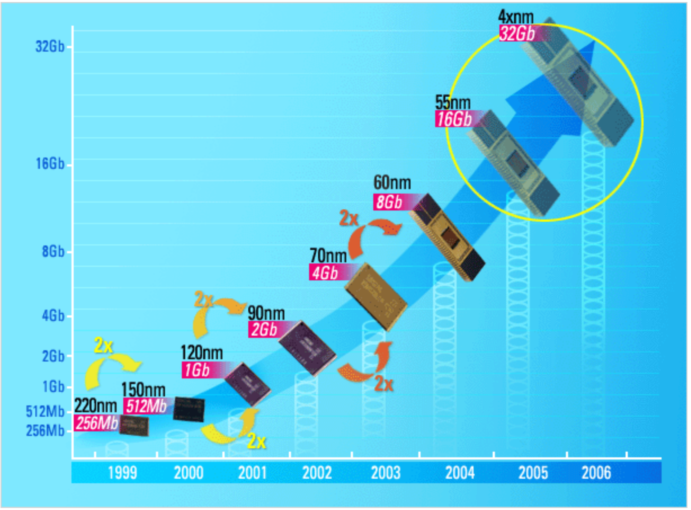
Tape Is Dead
Disk is Tape

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?



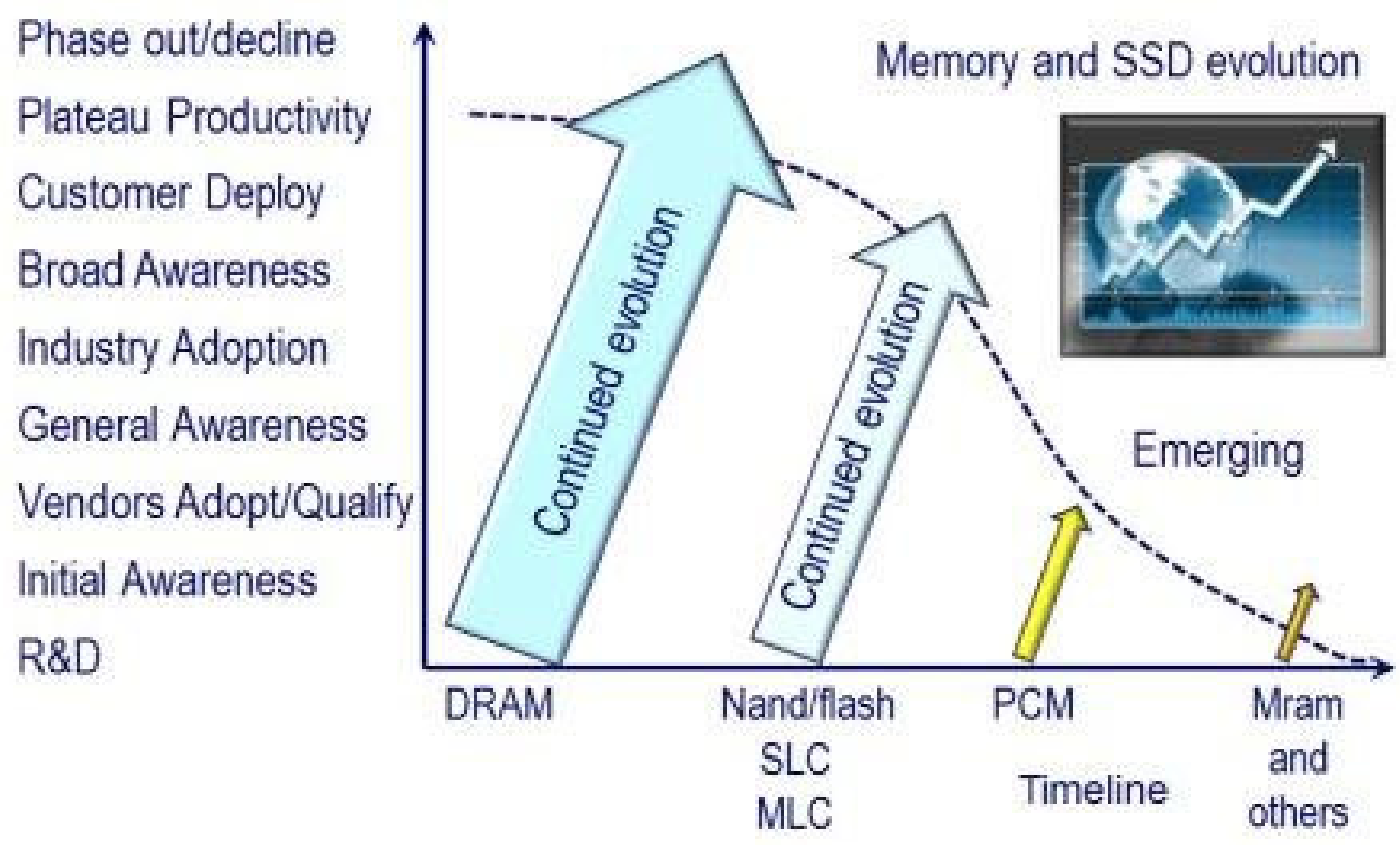
1995 16 Mb NAND flash chips
2005 16 Gb NAND flash
Doubled each year since 1995

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



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

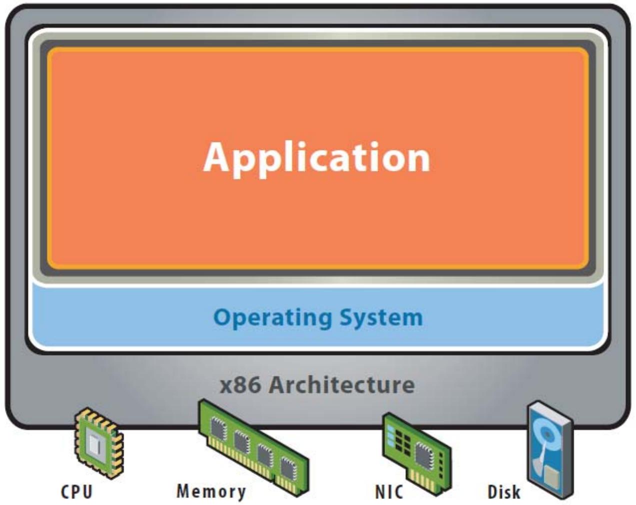
”Any problem in computer science can be solved with another level of indirection.”

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

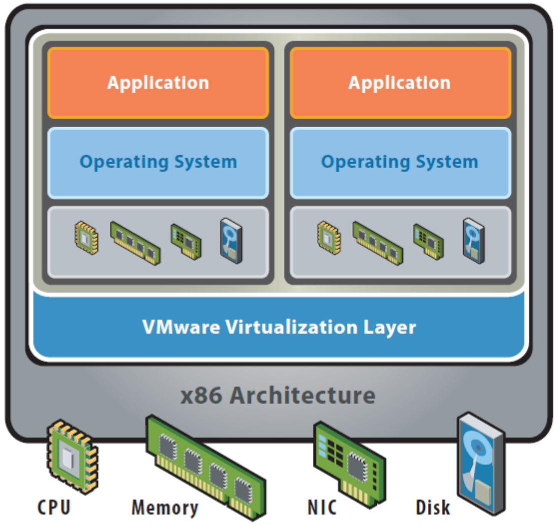
Concept of Virtualization

“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

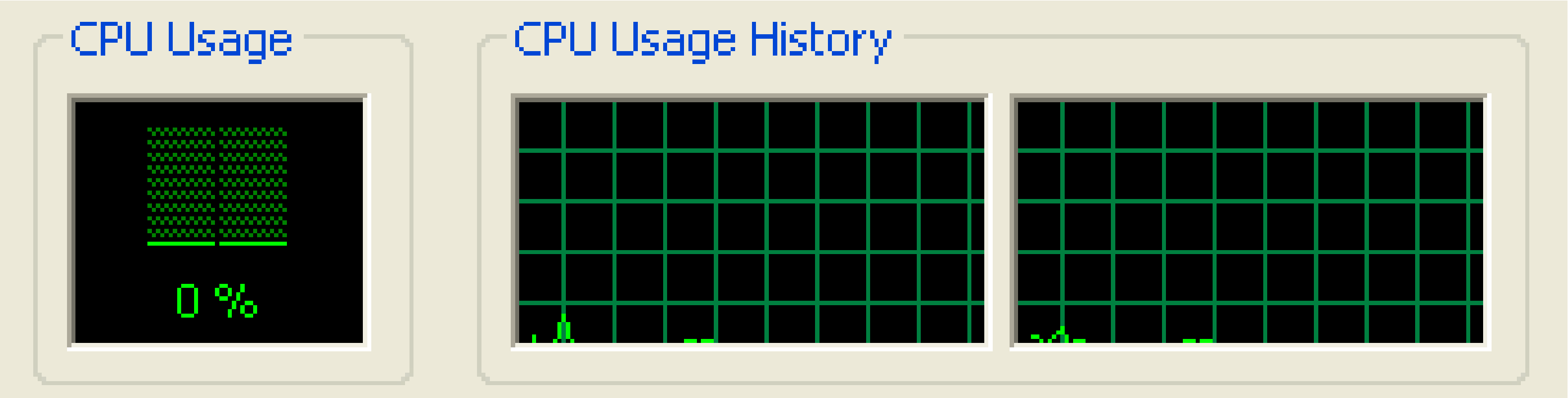
* “Virtualization in education”, IBM, 2007
•figures from: VMware white paper, virtualization overview

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

Next Class

Introduction to PC programming

Booting of an OS

Homework

- Writing your answer (in Chinese) to following questions

- What is do_brk()? How to exploit?
  - http://www.csee.umbc.edu/courses/undergraduate/421/Spring12/02/slides/ULKV.pdf
  - Give your own explaination

PC Programming & Booting

Yubin Xia & Rong Chen

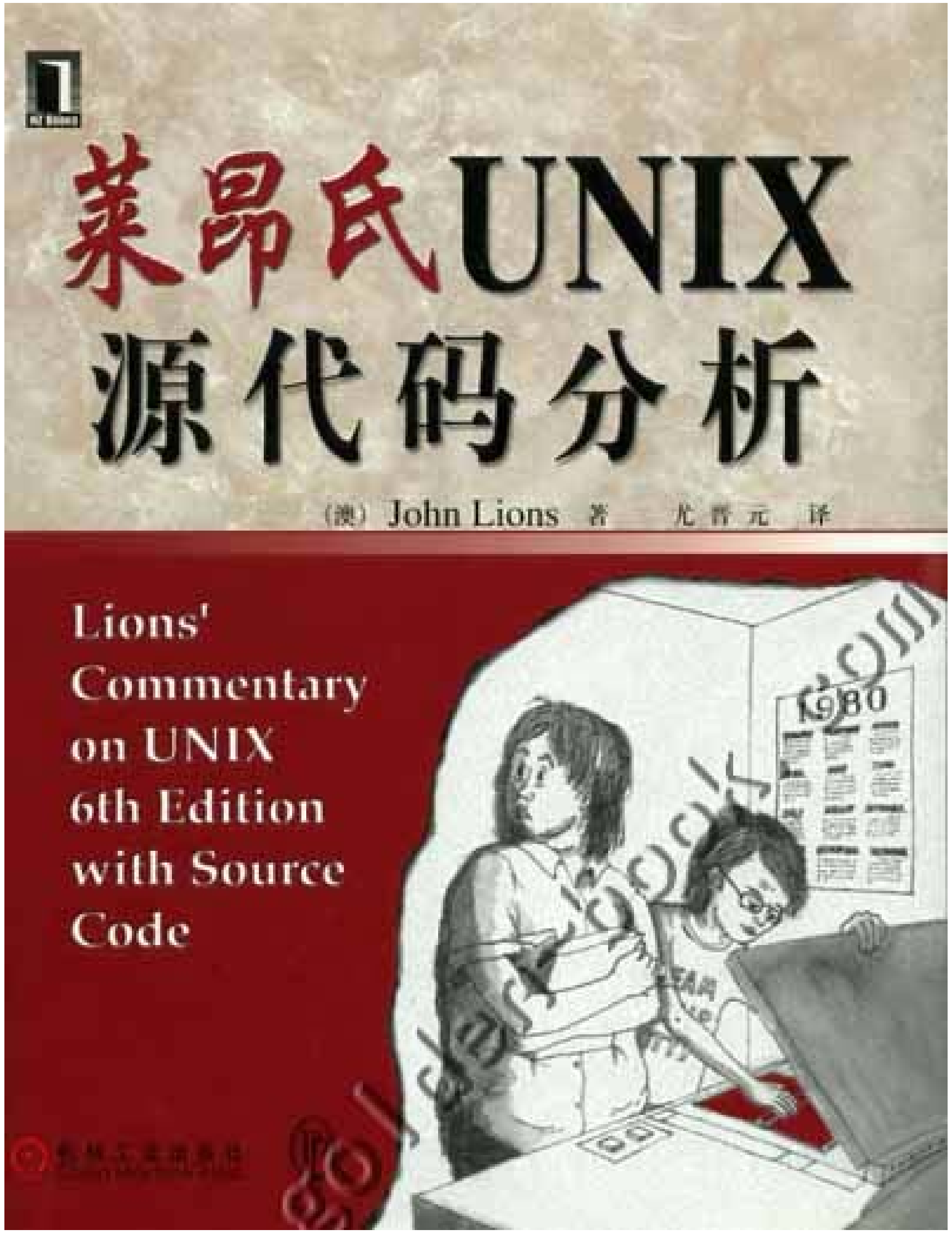
Our Tools

- QA Site:
  - http://ipads.se.sjtu.edu.cn/courses/qa
- Reference
  - 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!


莱昂氏UNIX
源代码分析
(澳) John Lions 著 尤晋元 译
Lions' Commentary on UNIX 6th Edition with Source Code
1980

A PC

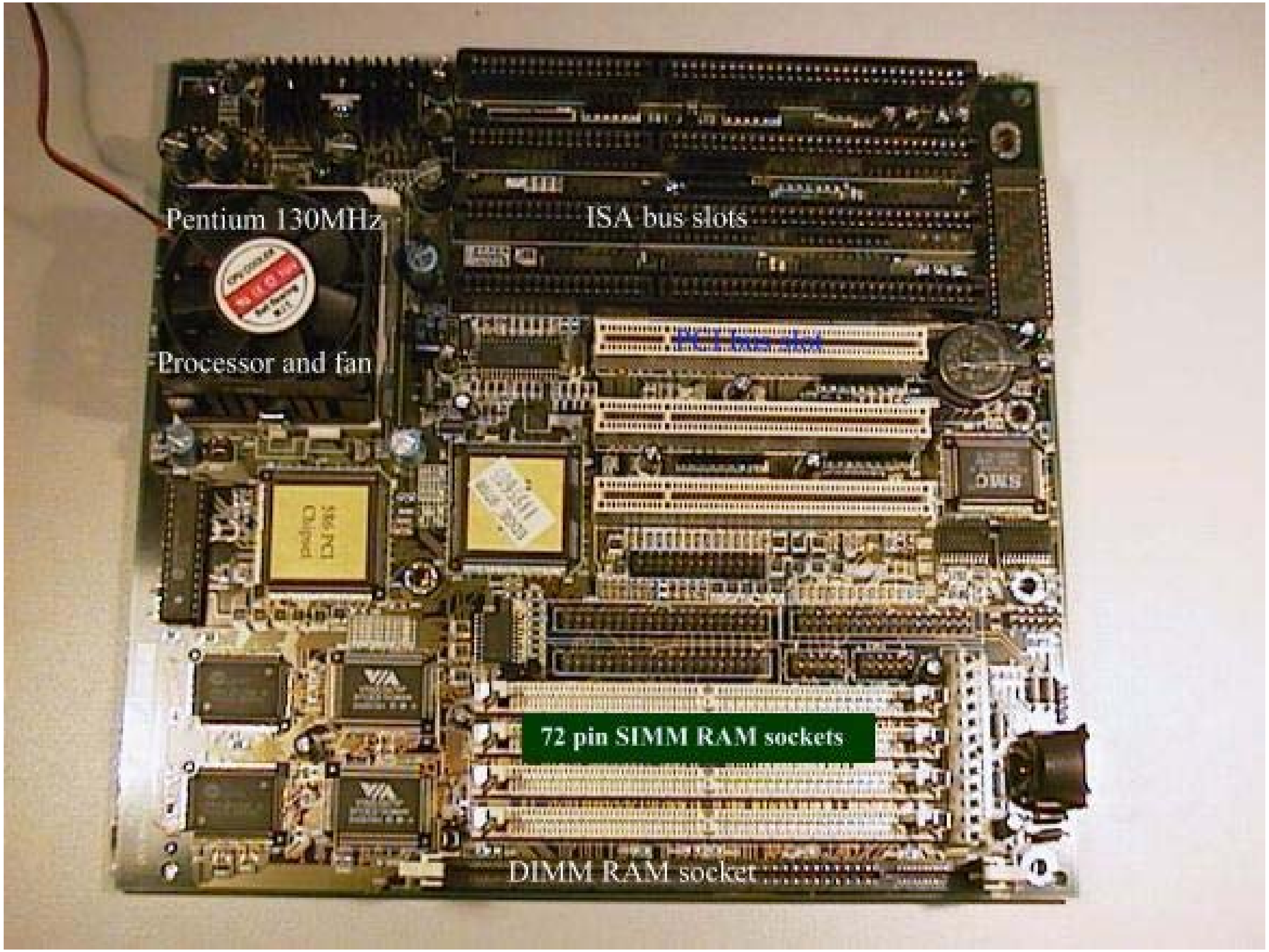
How to make it to do something useful?

Outline

- PC Architecture
- Memory
- Execution
- PC Emulation

- PC Architecture
- Memory
- Execution
- PC Emulation

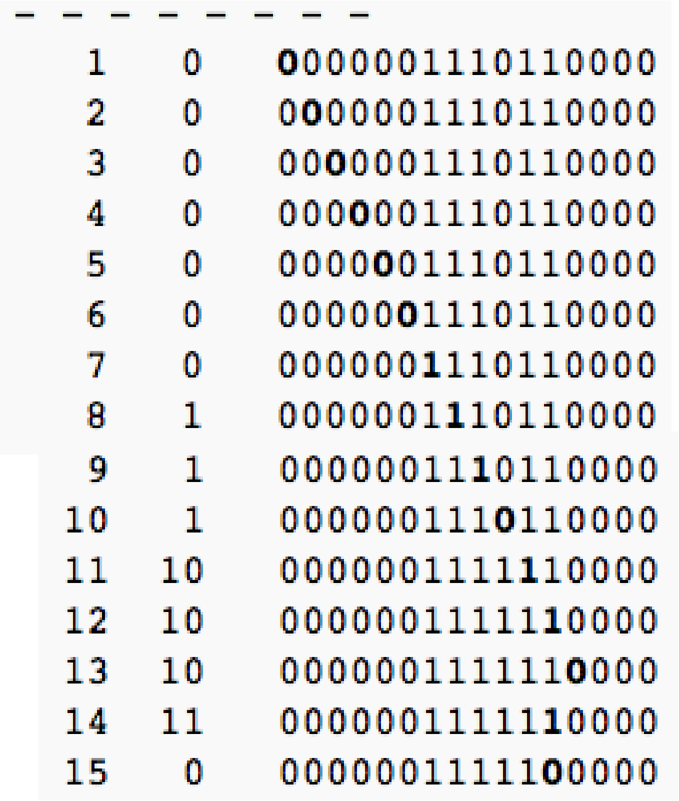
PC board


Pentium 130MHz
ISA bus slots
PCI bus slot
Processor and fan
72 pin SIMM RAM sockets
DIMM RAM socket

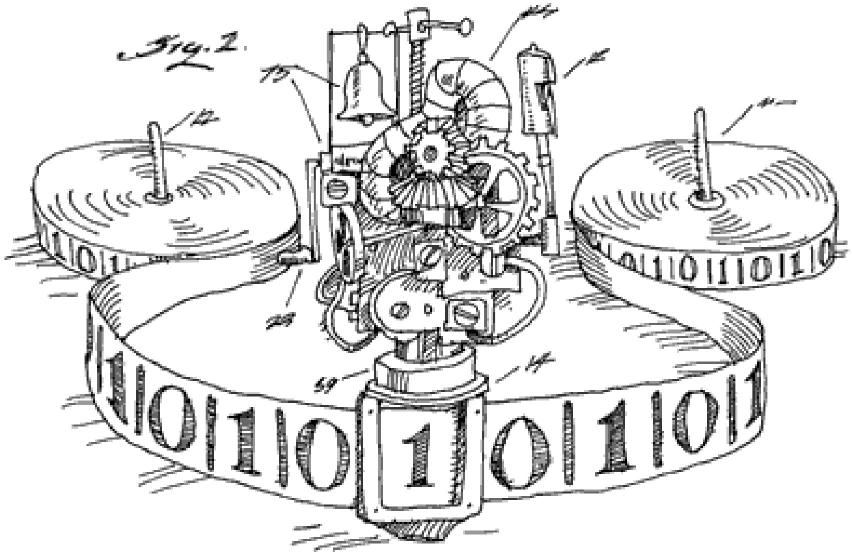
PC board - Nowadays

The Turing Machine

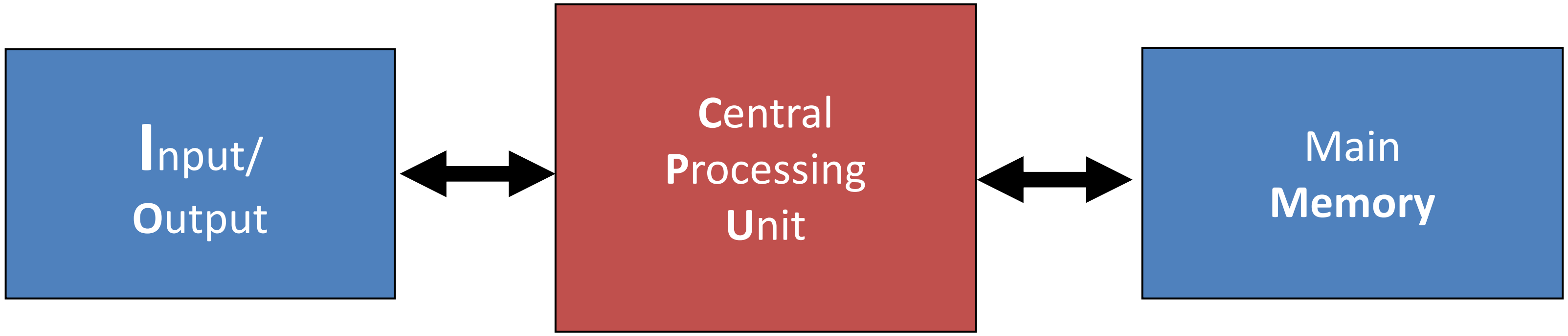
Calculate 3+2: 0000001110110000



```
- - - - - - - - -
   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    0000001111100000
```

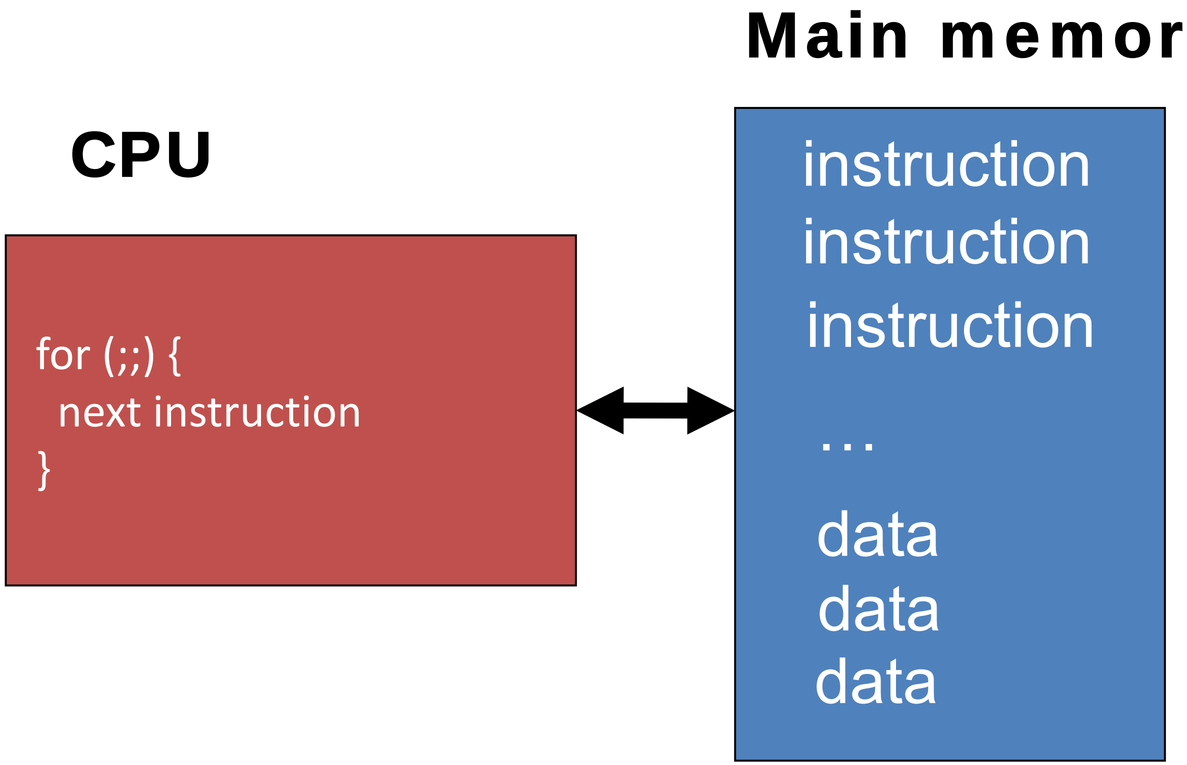


The von Neumann Model



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

The Stored Program Computer



Main memory

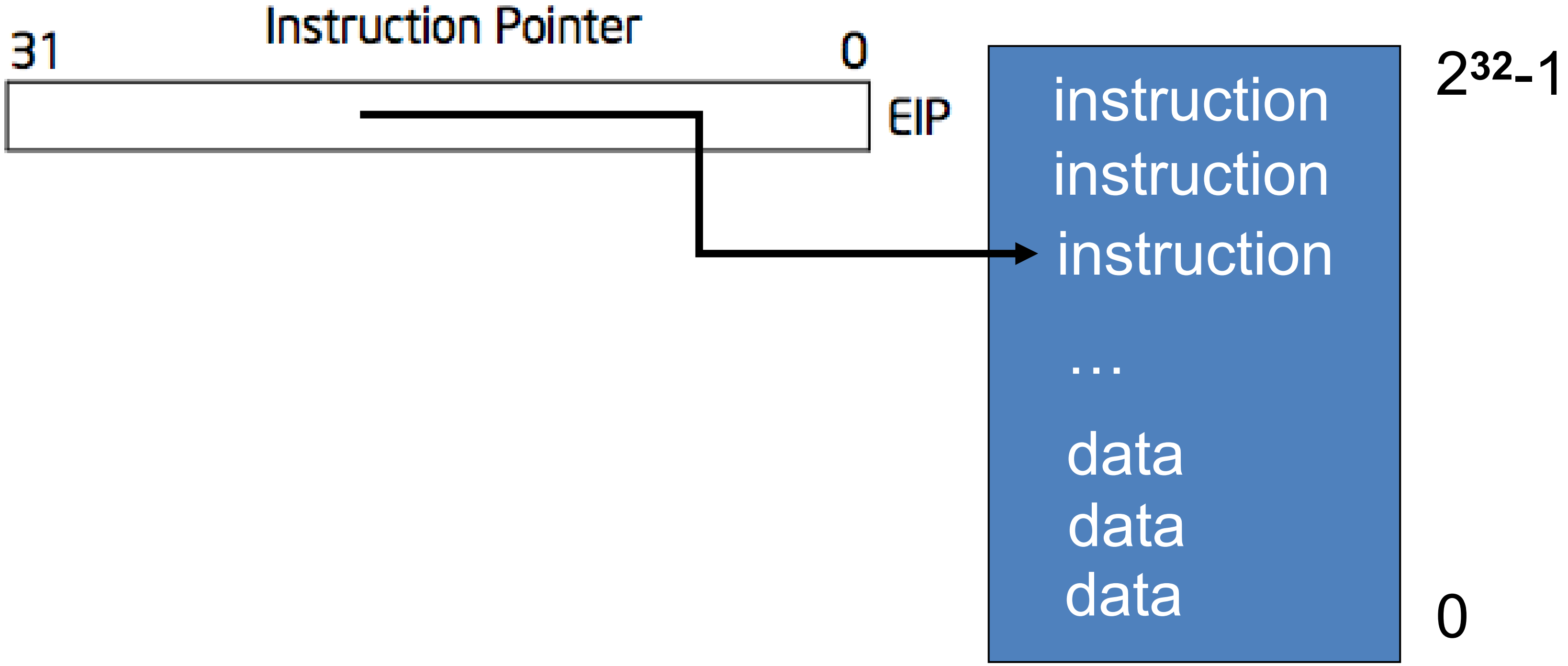
CPU

```
for (;;) {
  next instruction
}
```

instruction
instruction
instruction
…
data
data
data

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

x86 implementation



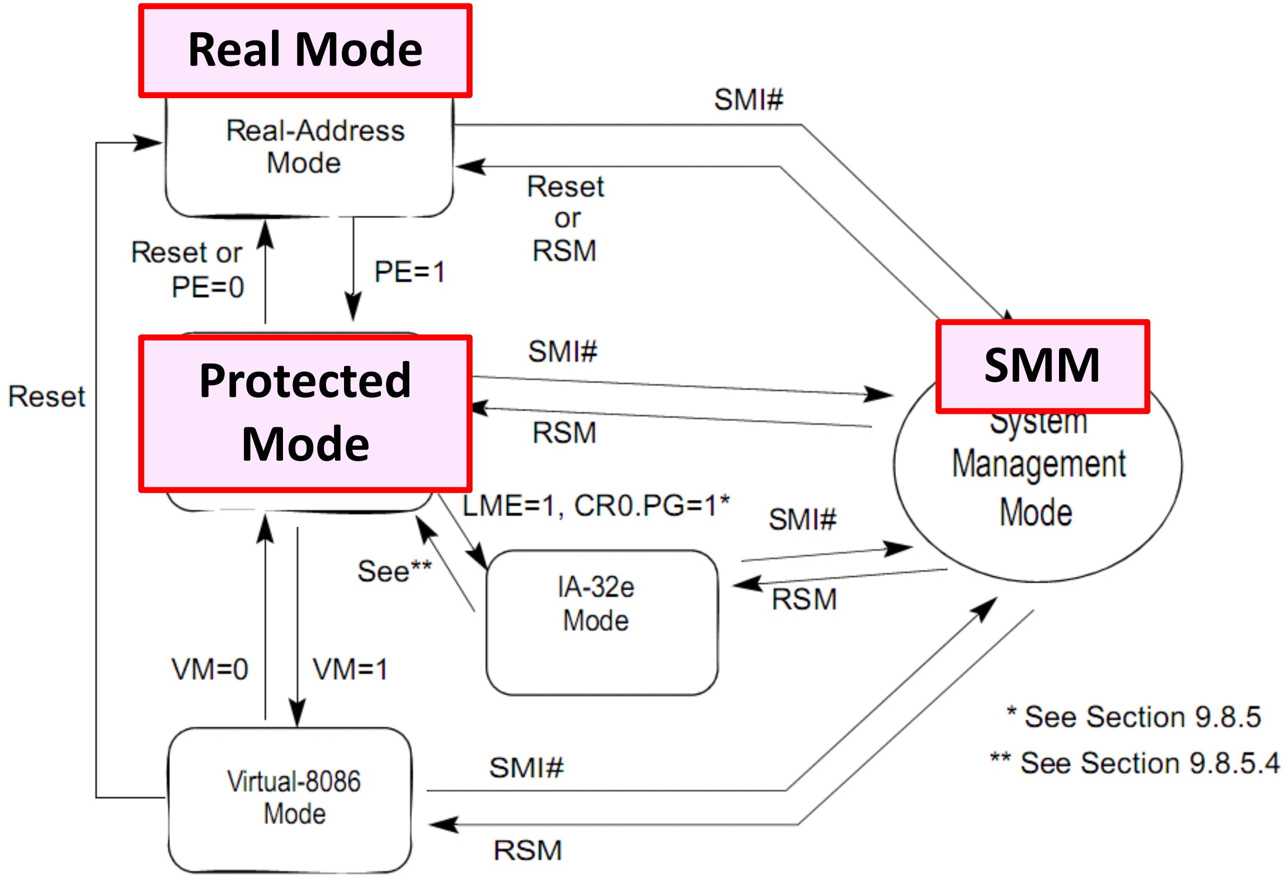
- **EIP** is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and cond. JMP

System Architecture Overview

Volume 3A: System Programming Guide, Part 1

- 2.1~2.5

- Modes

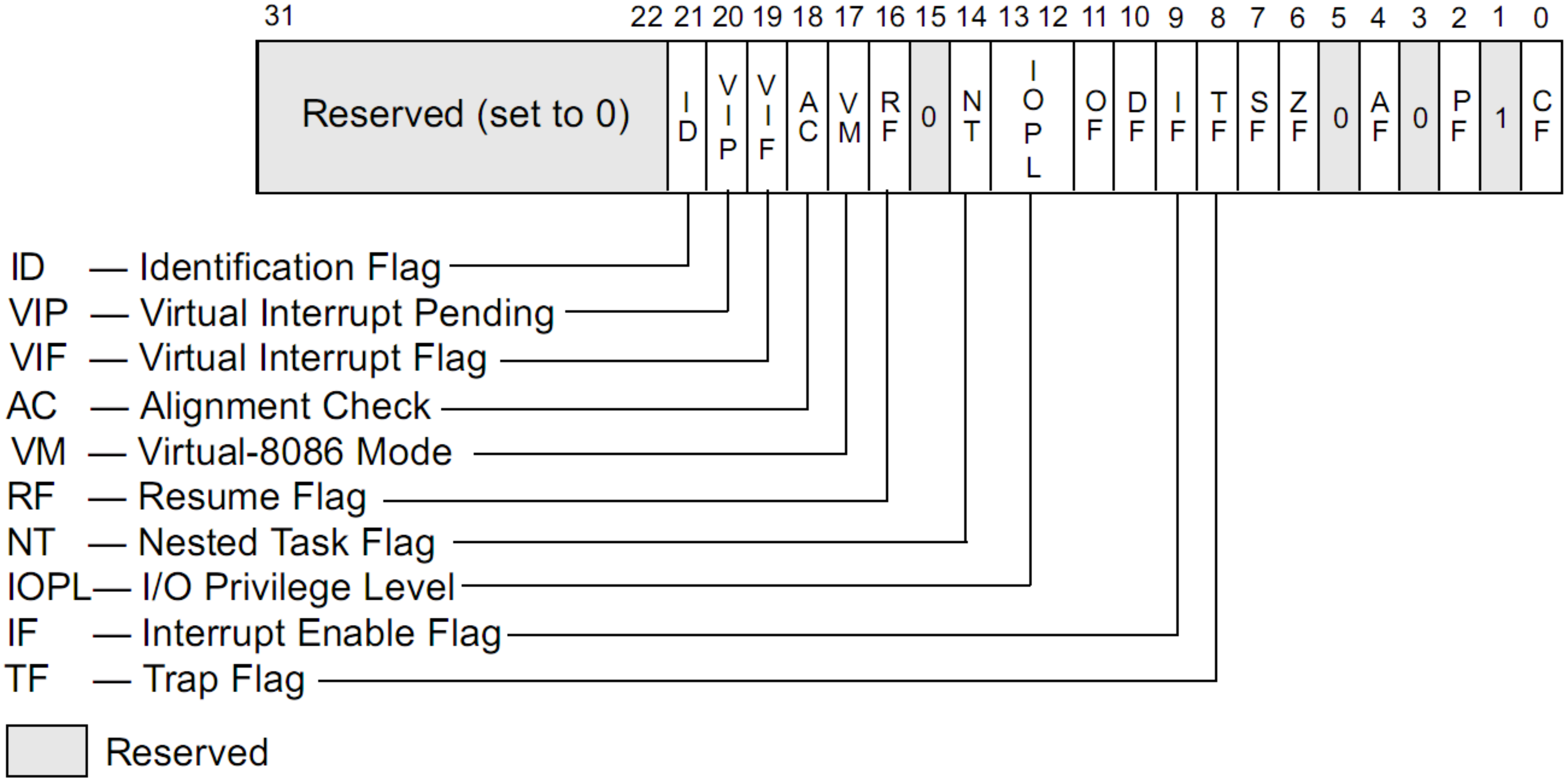


* See Section 9.8.5

** See Section 9.8.5.4

System Architecture Overview

- System Flags in the EFLAGS



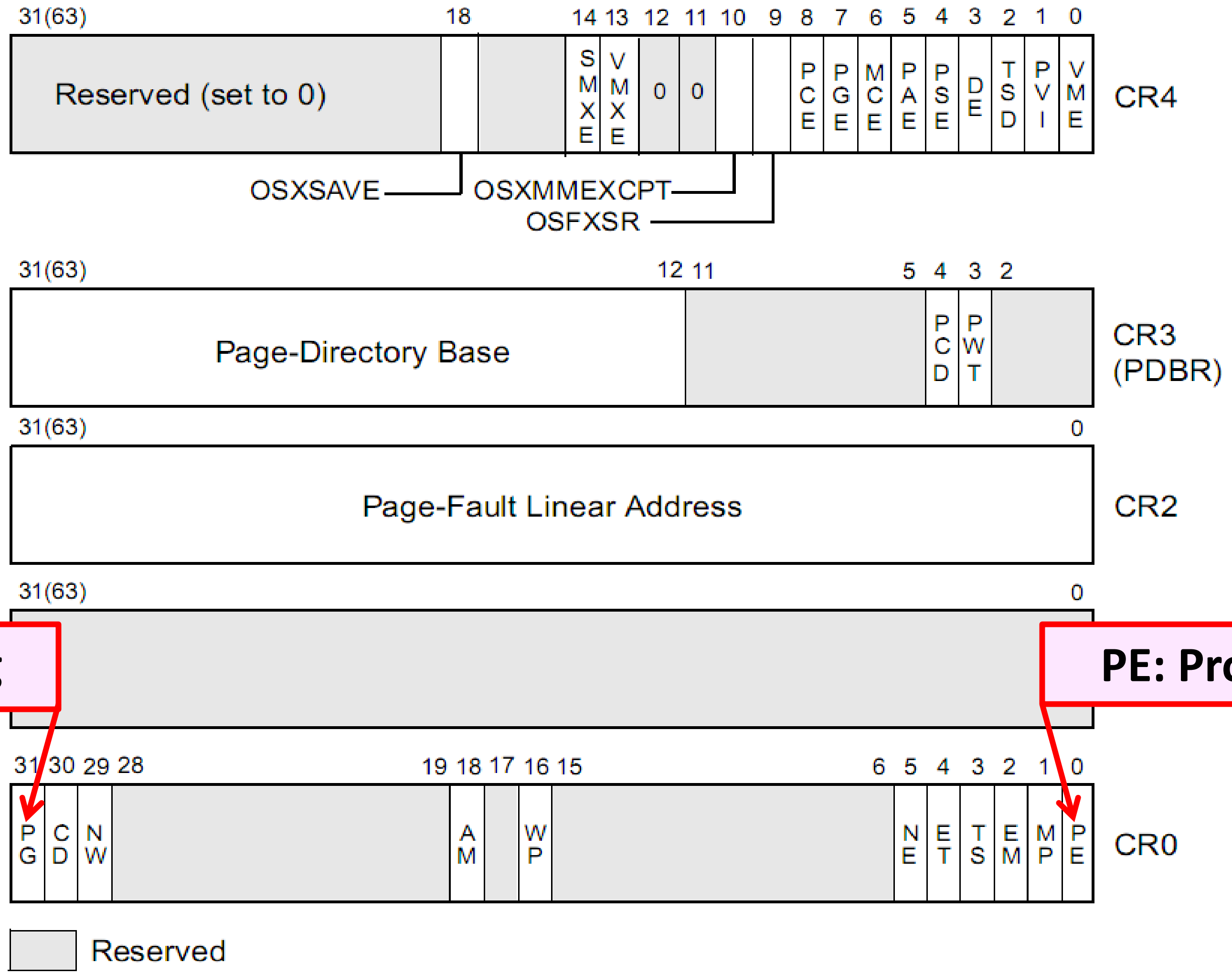
31	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved (set to 0)		ID	VIP	VIF	AC	VM	RF	0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

ID — Identification Flag
VIP — Virtual Interrupt Pending
VIF — Virtual Interrupt Flag
AC — Alignment Check
VM — Virtual-8086 Mode
RF — Resume Flag
NT — Nested Task Flag
IOPL— I/O Privilege Level
IF — Interrupt Enable Flag
TF — Trap Flag

▨ Reserved

System Architecture Overview

- Control Registers



PG: Paging

PE: Protection

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

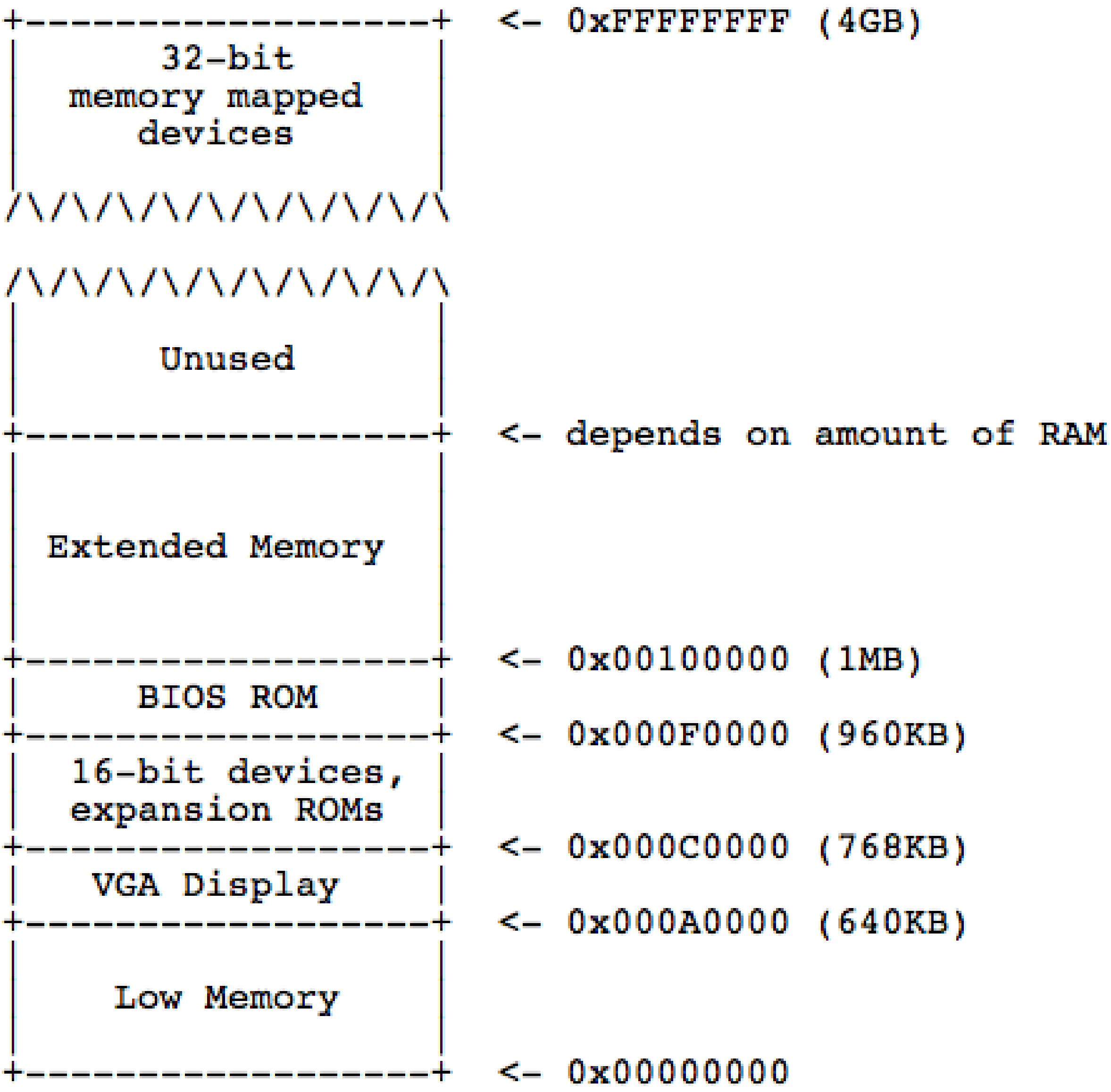
- 8086: 16-bits microprocessor
  - Real Mode: *physical addr = 16 * segment + offset*
  - Space: 64KB

- external address to 20-bits
  - Space: 1MB
  - The extra 4 bits come *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=f000 IP=fff0 =>  ADDR: ffff0`*

Memory Model

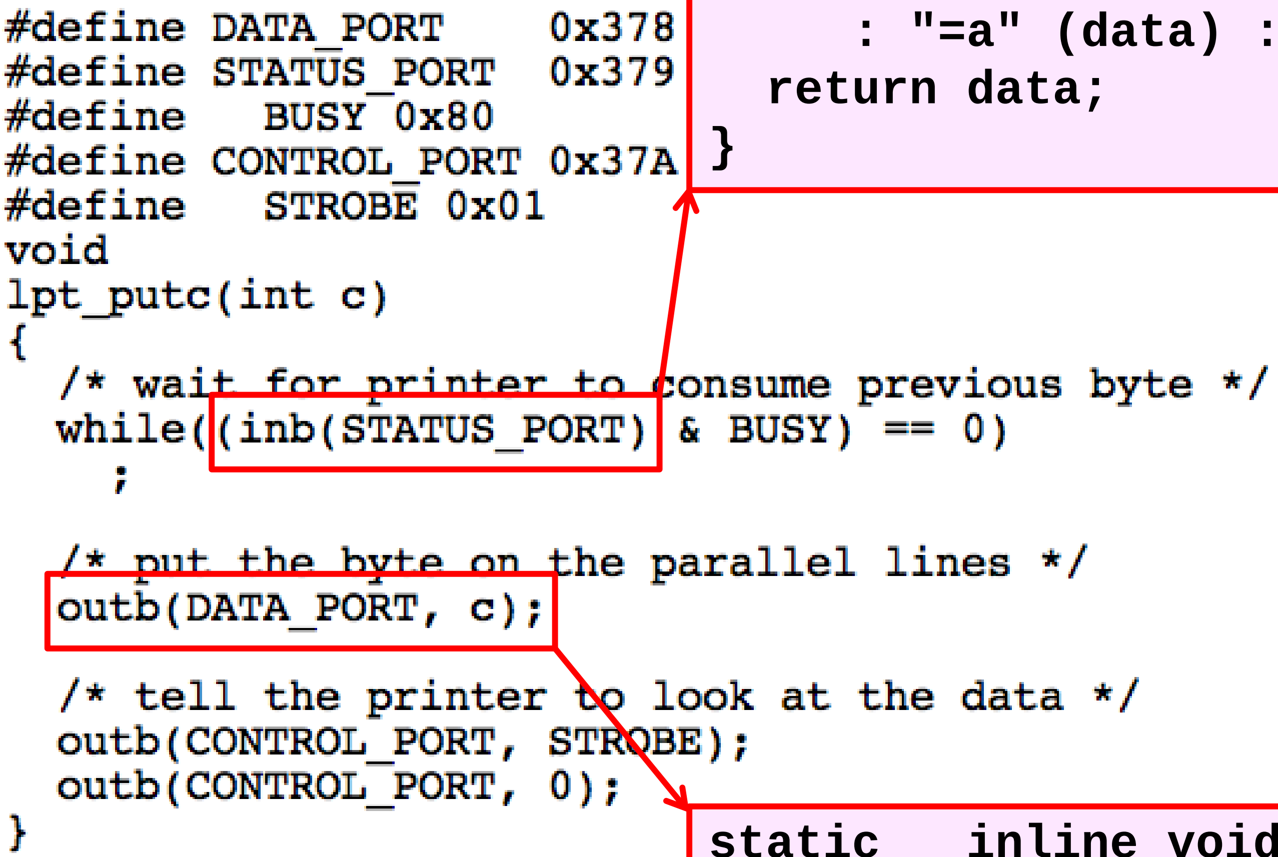
- 80386: 32-bit data and bus addresses
  - Protected Mode
- Now: the transition to 64-bit addresses
- Backwards compatibility:
  - Boots in 16-bit real mode, and switches to 32-bit protected mode
    - See: "*boot/boot.S*"

Memory Layout



```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

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));
}
```

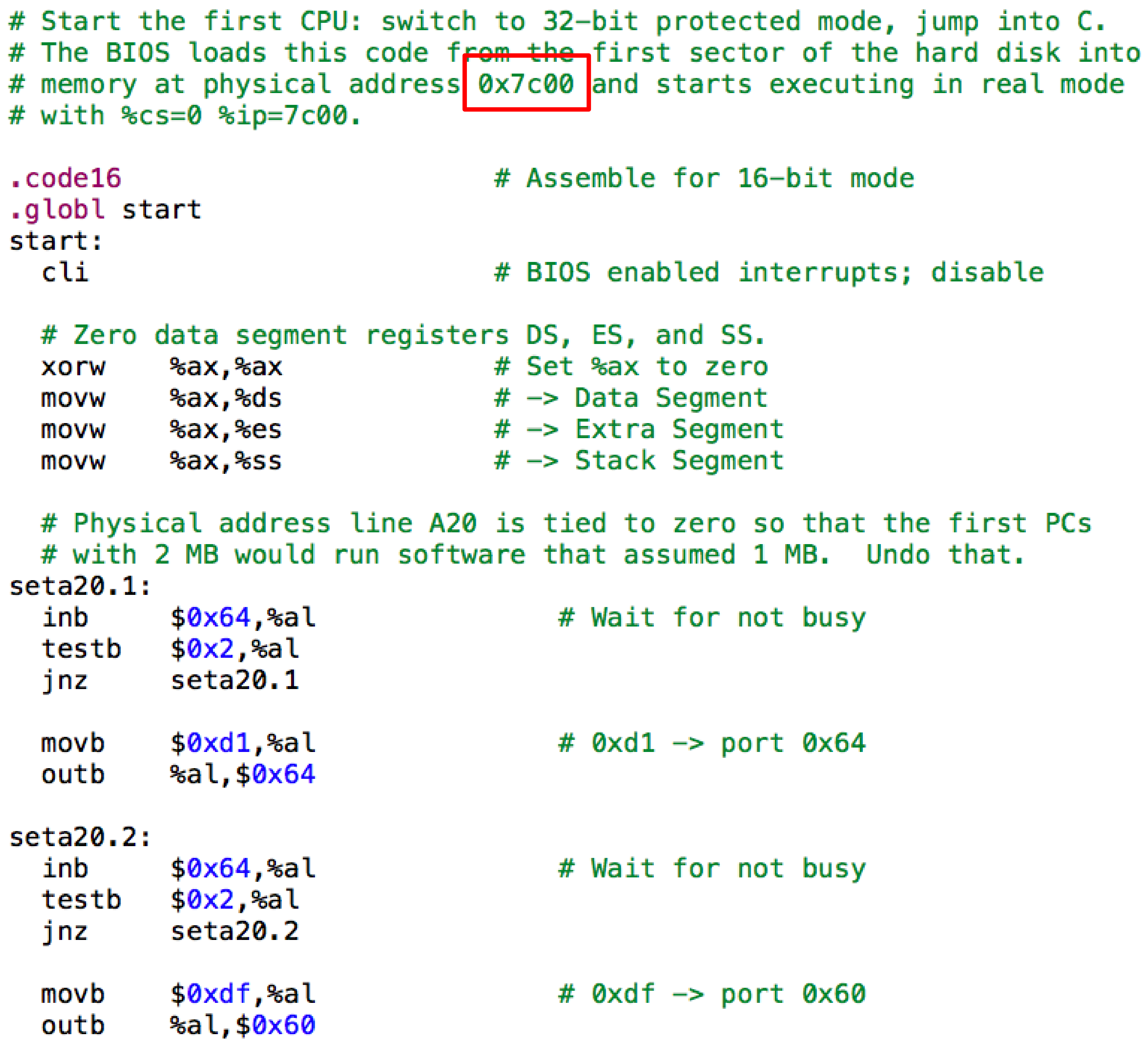
Memory-mapped I/O

- Use normal addresses
  - No need for special instructions
  - No 1024 limit
  - System controller routes to device

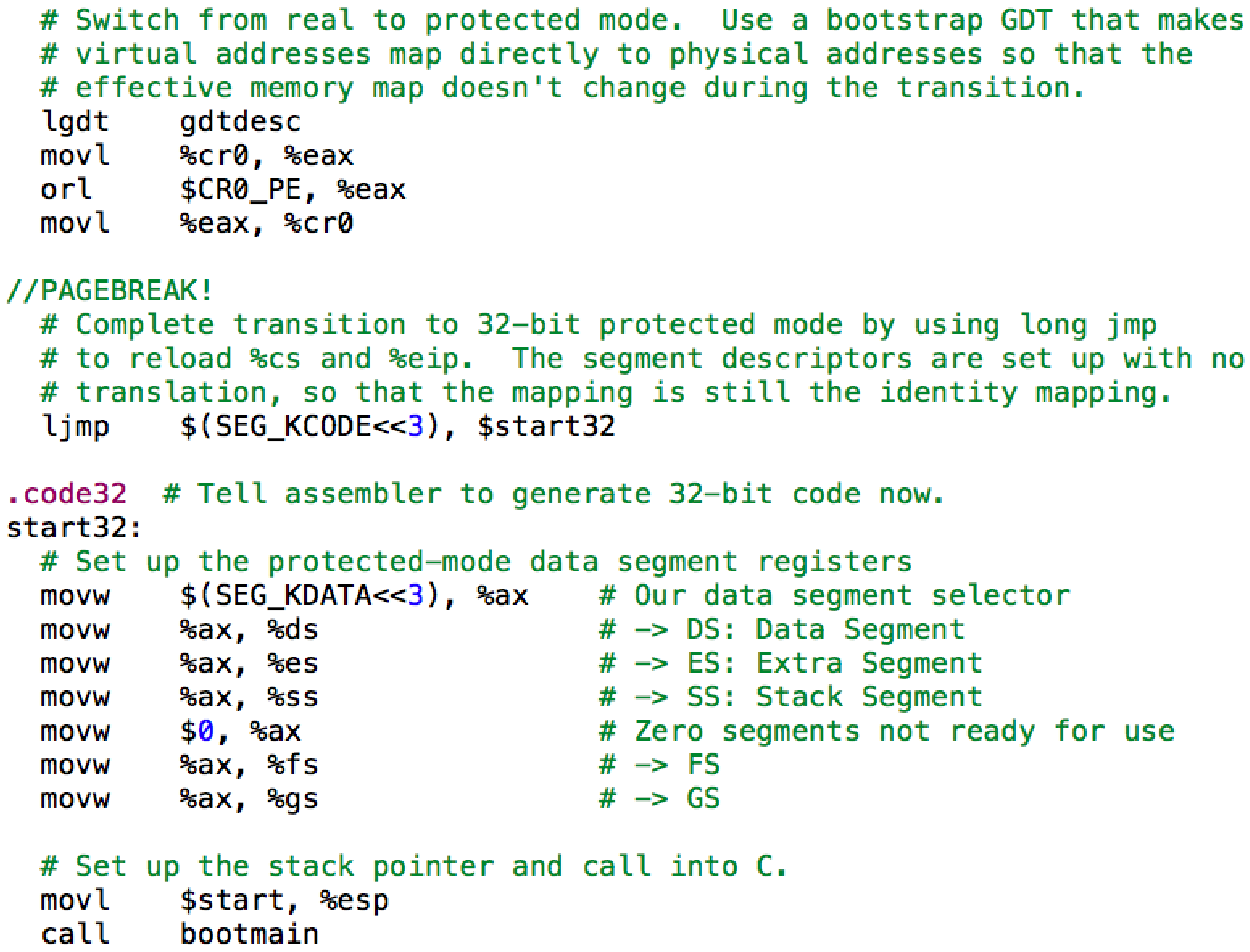
- Works like “magic” Memory
  - Addressed and accessed like memory
  - But does not behave like memory
  - Reads and writes have “side effects”
  - Read result can change due to external events
    - Recall: the “volatile” keyword

- PC Architecture
- Memory
- Execution
- PC Emulation

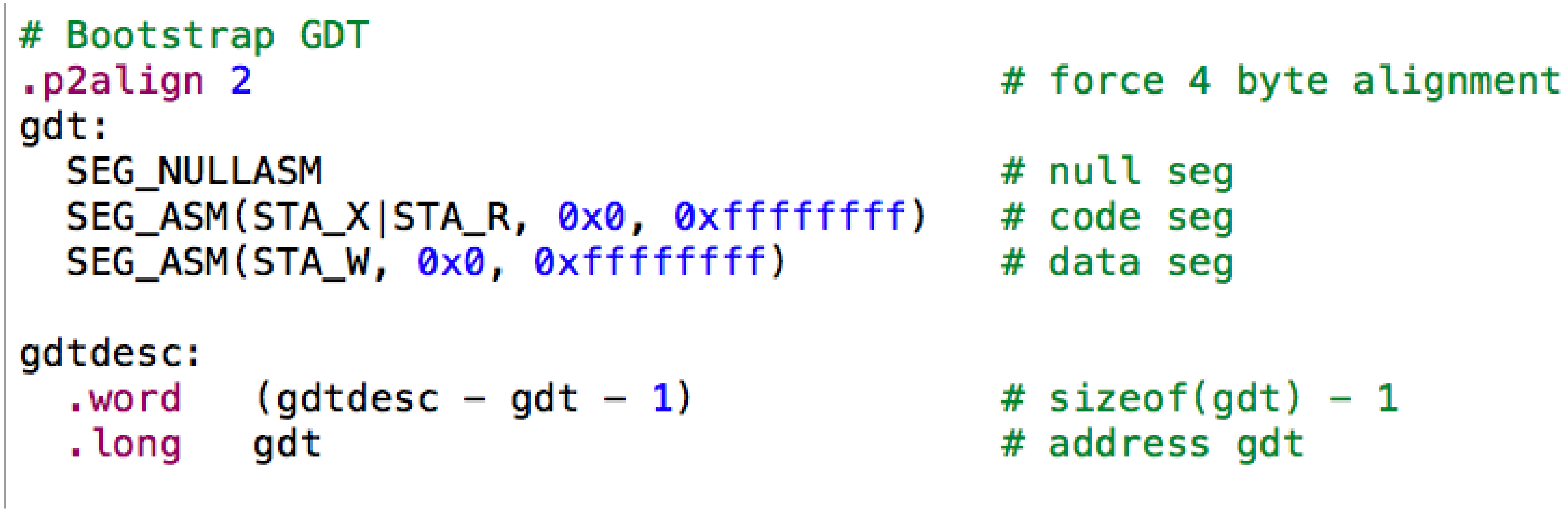
XV6 BOOTING CODE



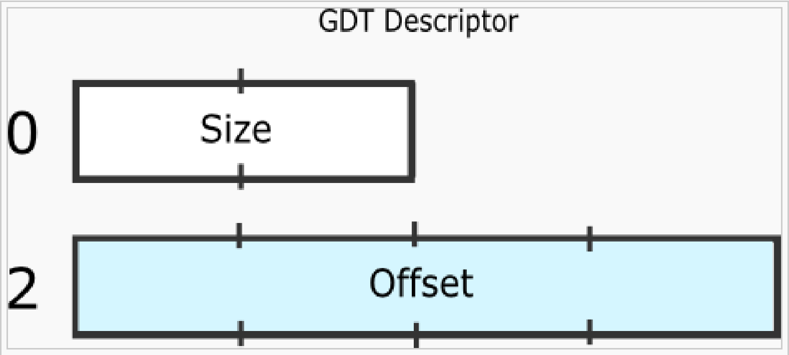
```
# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.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
```



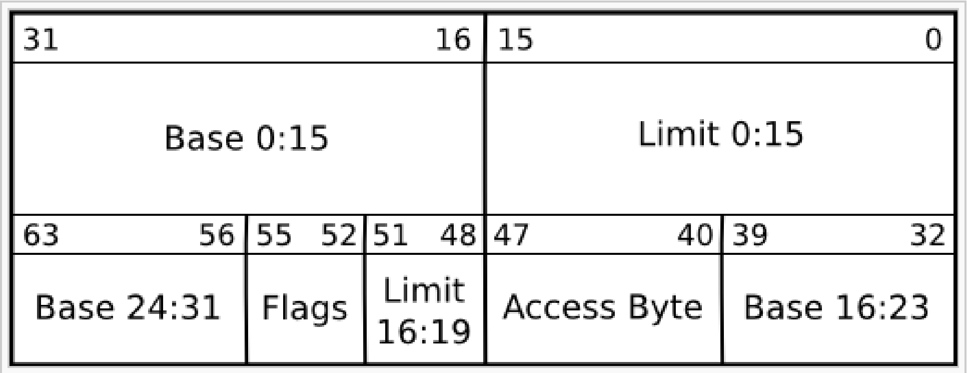
```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//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
```



GDTR

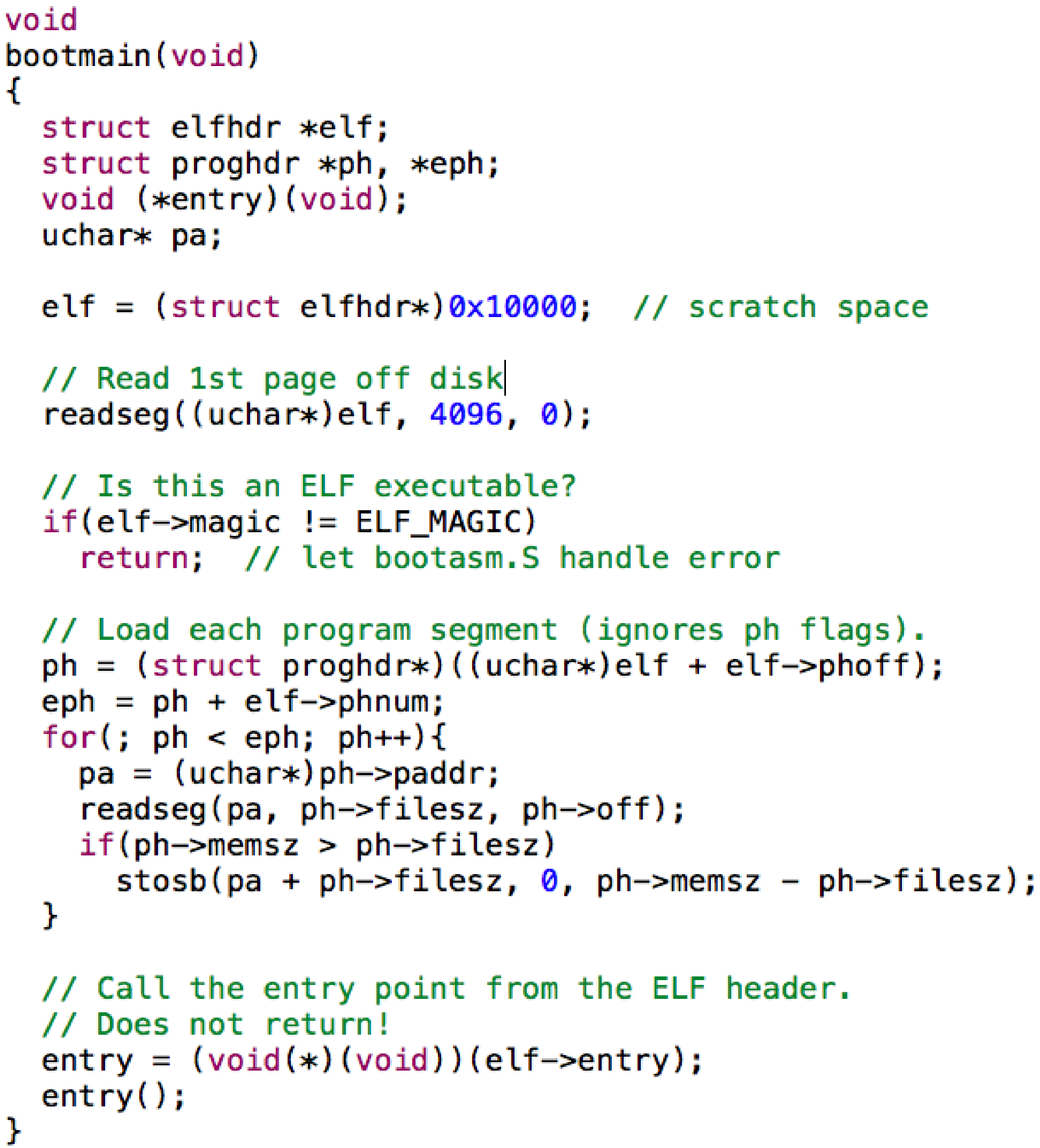


31 16	15 0
Base 0:15	Limit 0:15

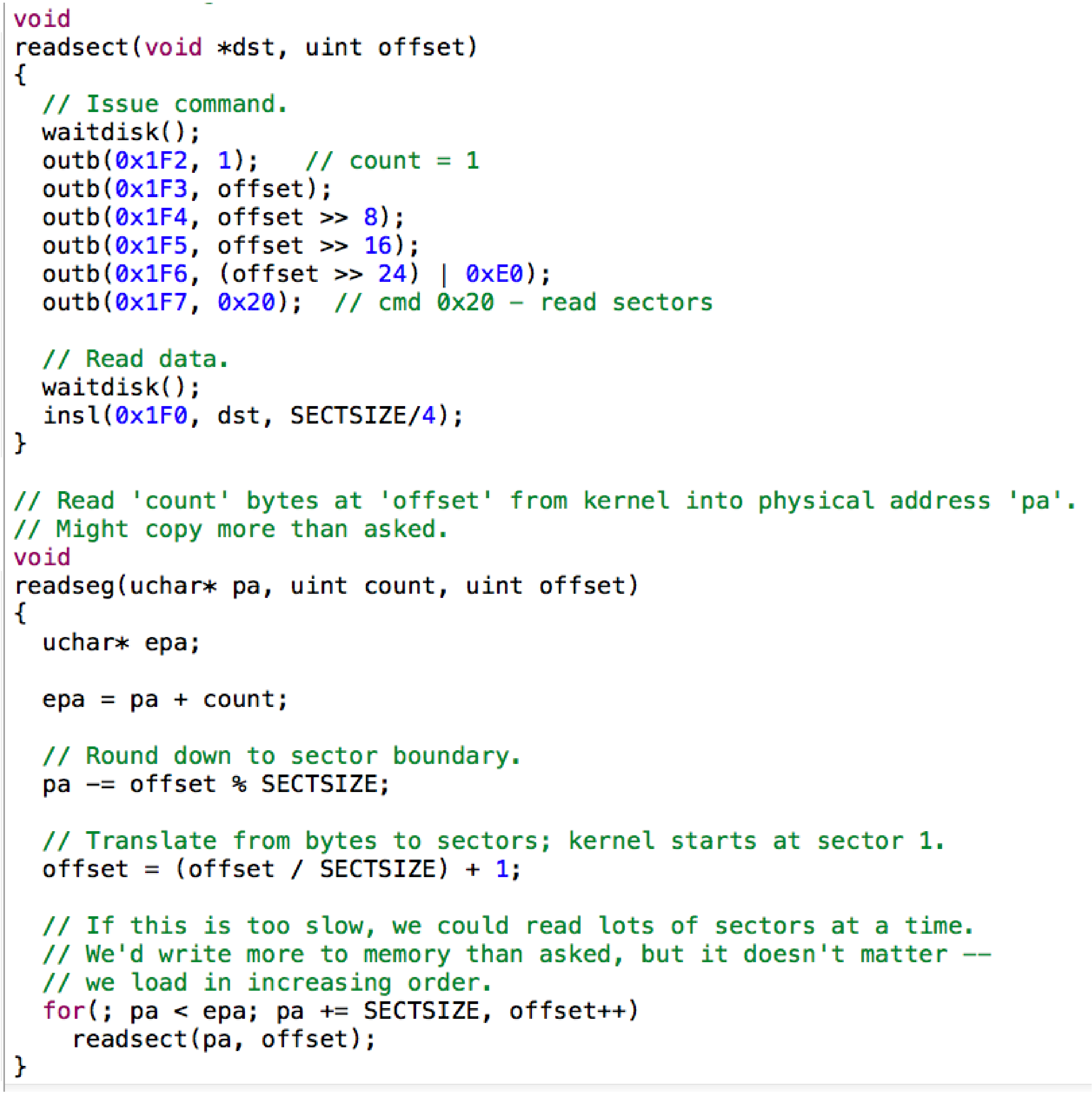
63 56	55 52	51 48	47 40	39 32
Base 24:31	Flags	Limit 16:19	Access Byte	Base 16:23

A GDT Entry

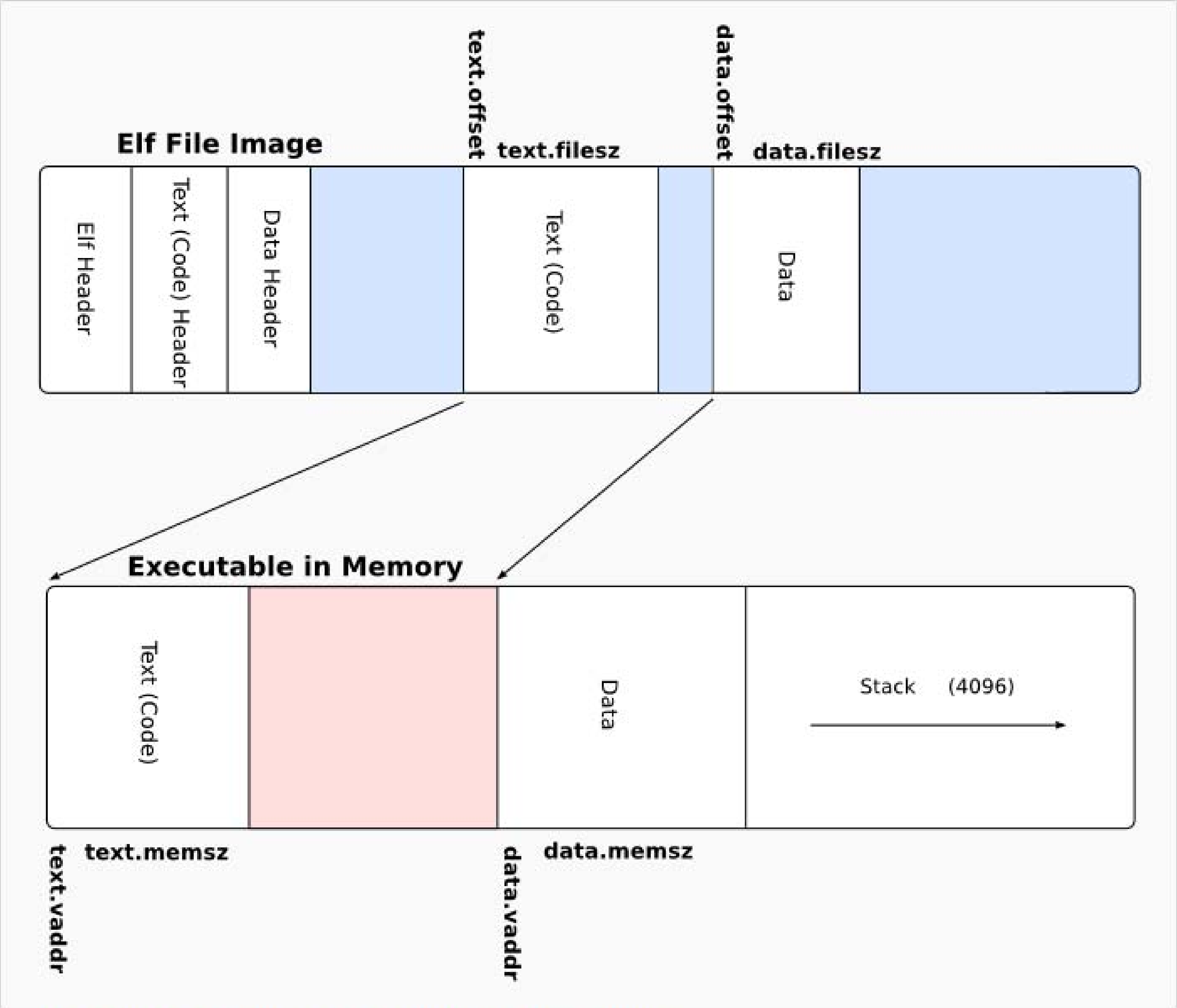
http://wiki.osdev.org/Global_Descriptor_Table



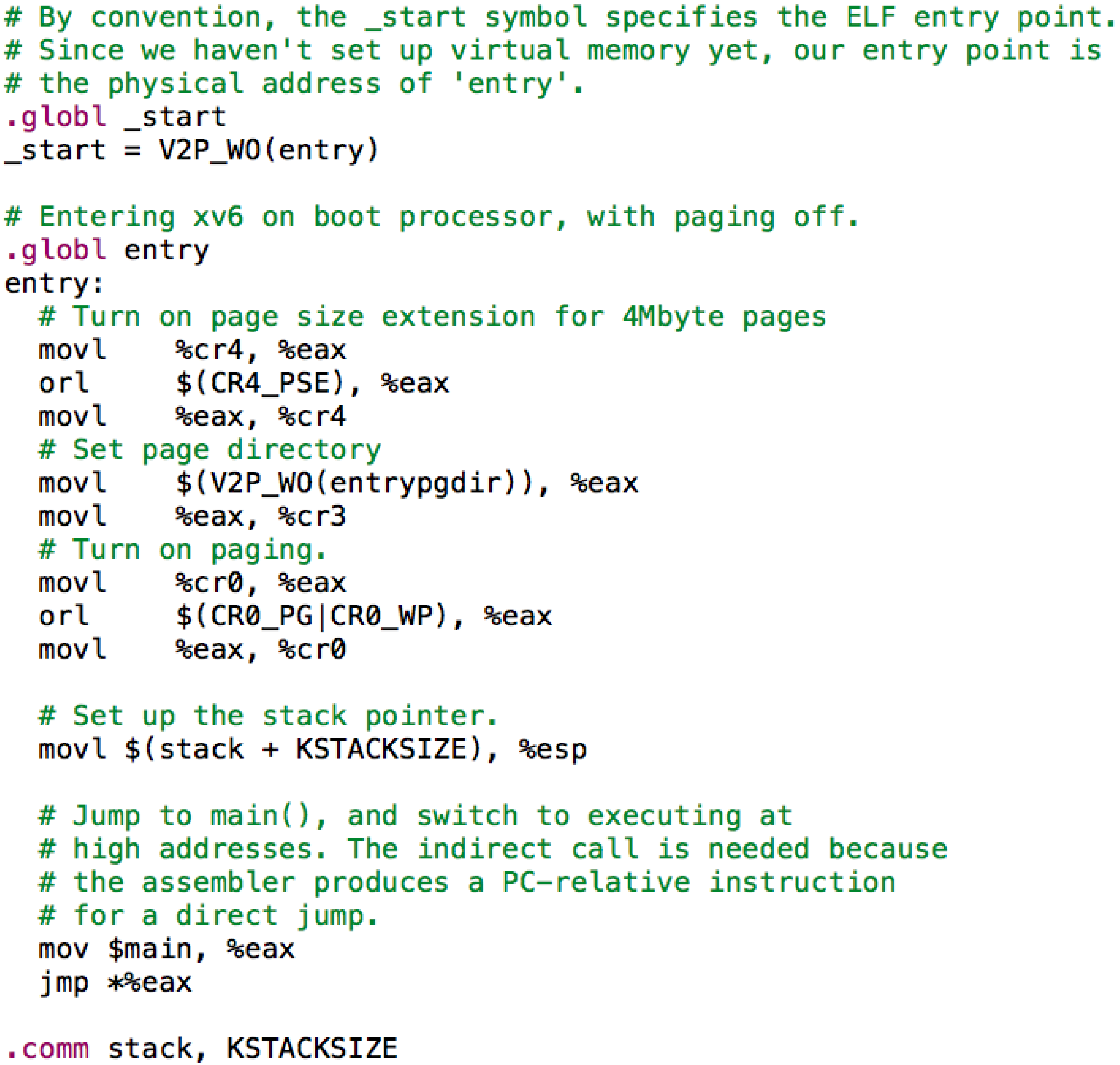
```
void
bootmain(void)
{
  struct elfhdr *elf;
  struct proghdr *ph, *eph;
  void (*entry)(void);
  uchar* pa;

  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();
}
```



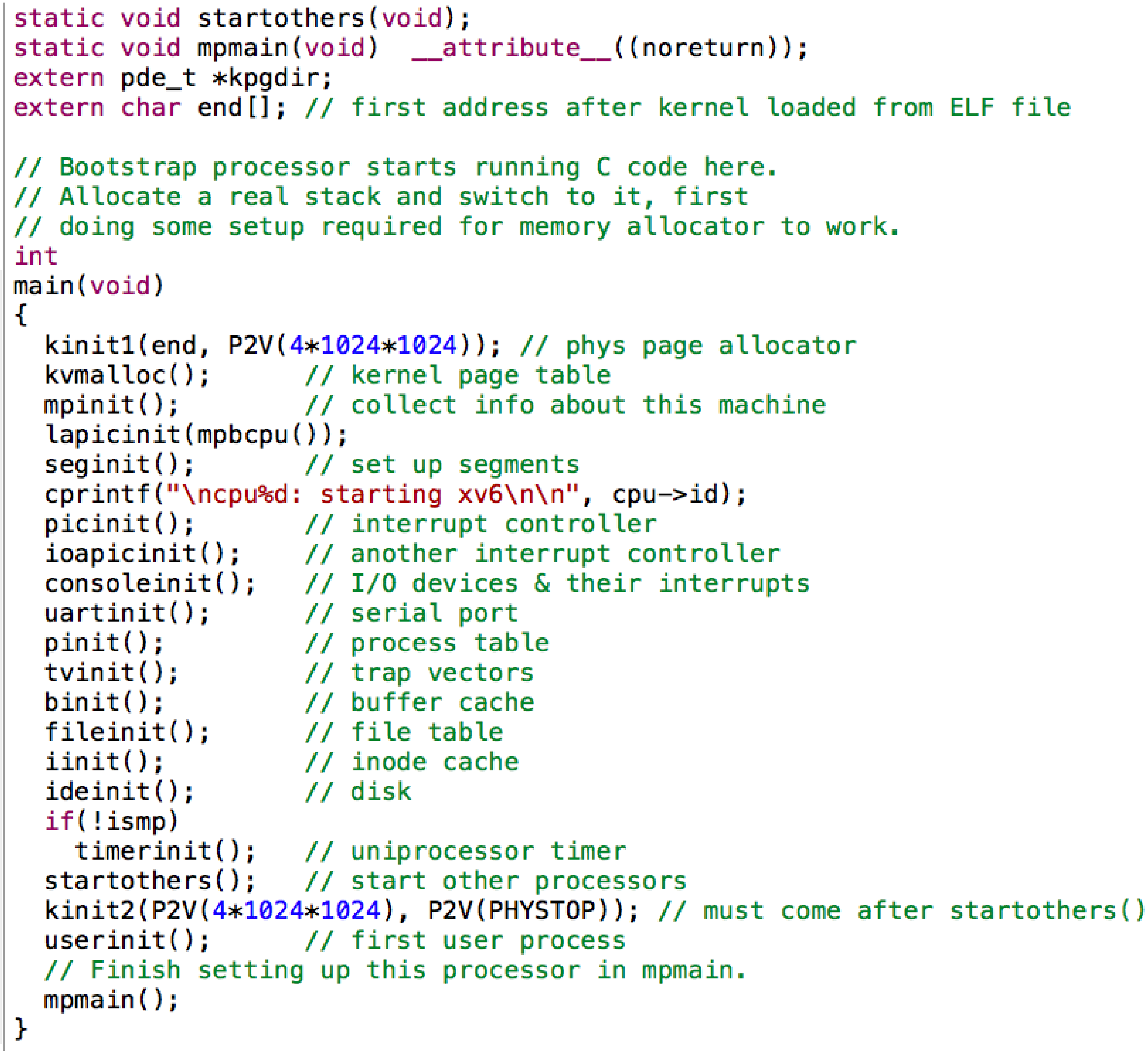
```
void
readsect(void *dst, uint offset)
{
  // Issue command.
  waitdisk();
  outb(0x1F2, 1);   // count = 1
  outb(0x1F3, offset);
  outb(0x1F4, offset >> 8);
  outb(0x1F5, offset >> 16);
  outb(0x1F6, (offset >> 24) | 0xE0);
  outb(0x1F7, 0x20);  // cmd 0x20 - read sectors

  // 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);
}
```



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_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```



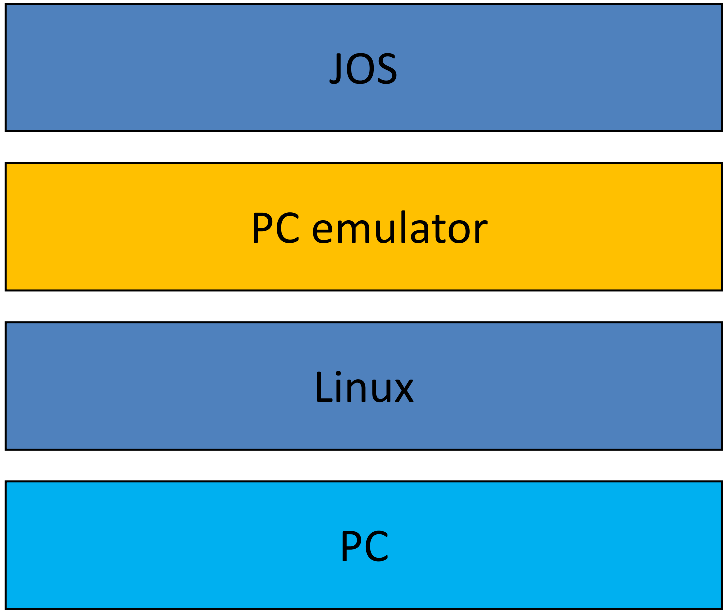
```
static void startothers(void);
static void mpmain(void)  __attribute__((noreturn));
extern pde_t *kpgdir;
extern char end[]; // first address after kernel loaded from ELF file

// 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

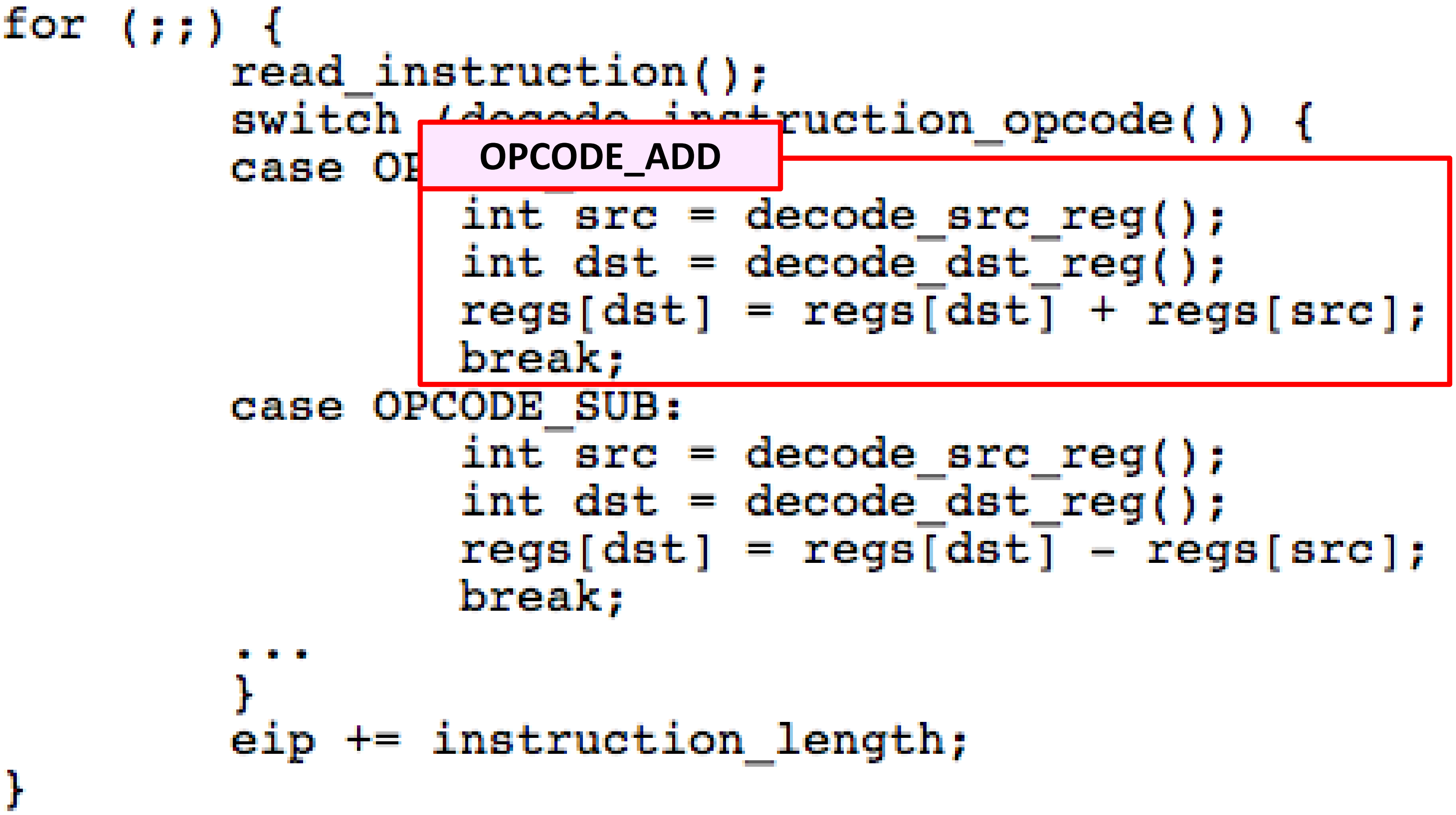
Development using PC emulator

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

Runs like a normal program on “host” operating system



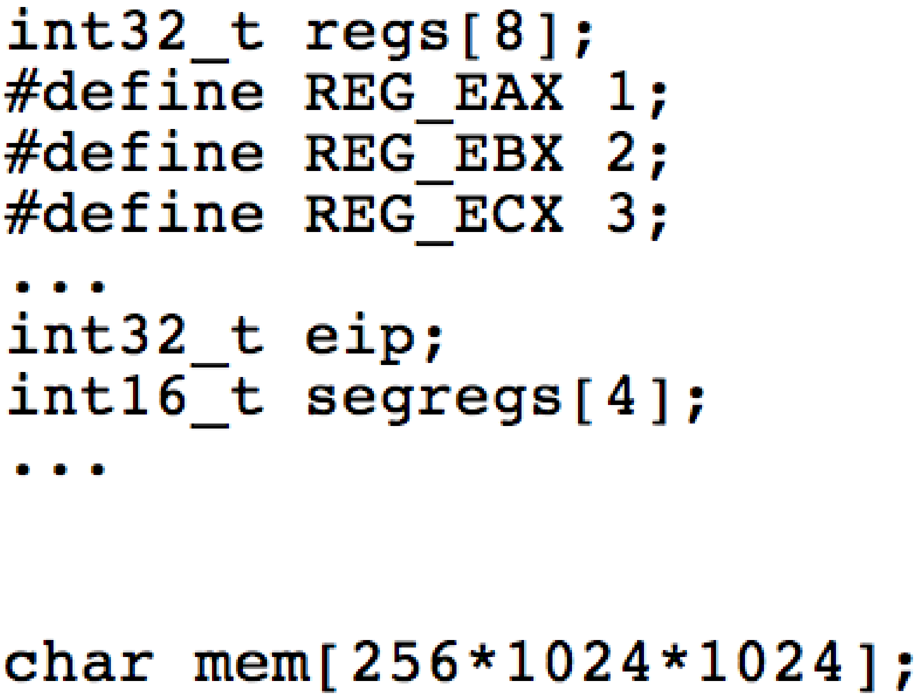
Emulation of CPU



```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OP
                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;
}
```

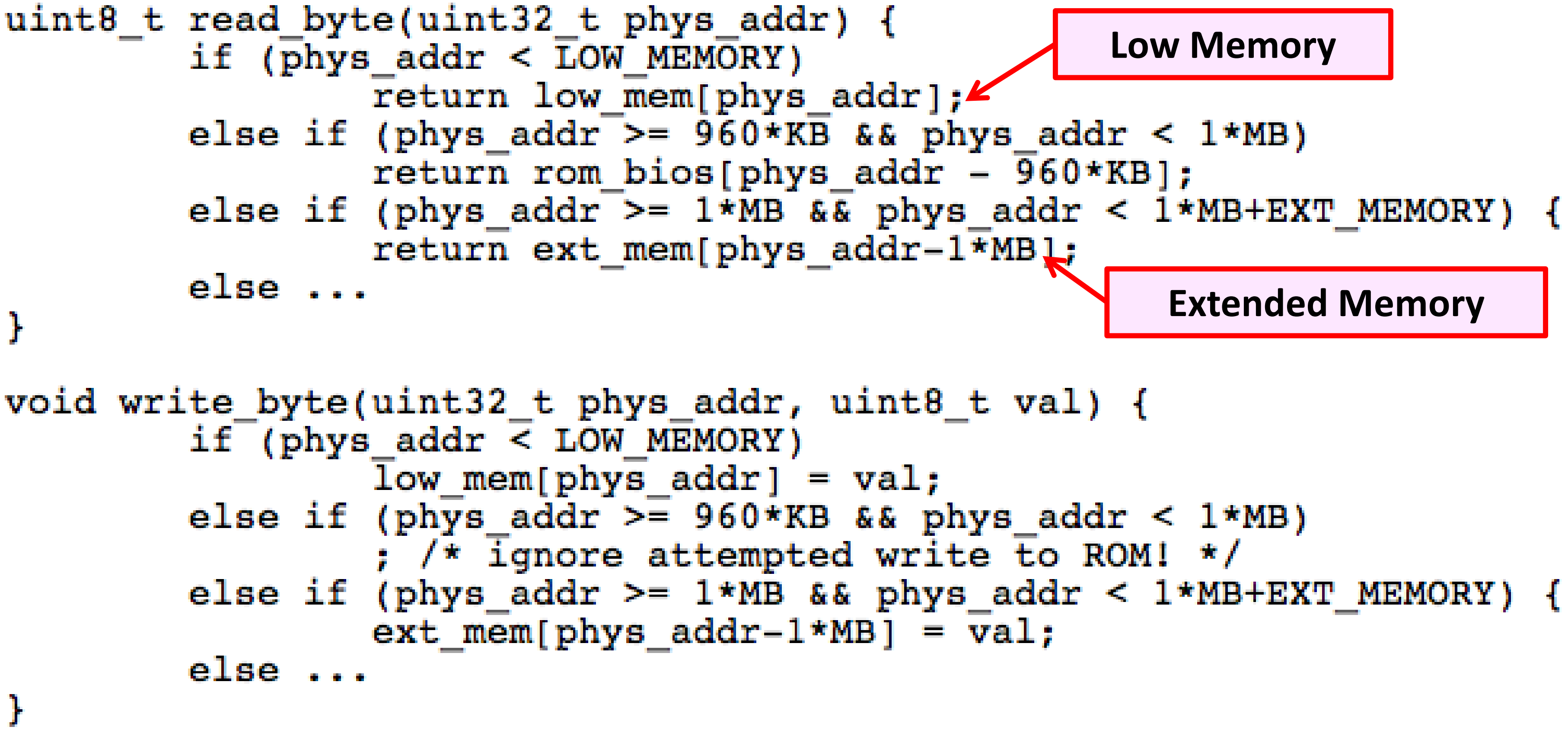
OPCODE_ADD

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 ...
}

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 ...
}
```

Low Memory

Extended Memory

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)

Xv6 Boot & OS Structure

Yubin Xia & Rong Chen

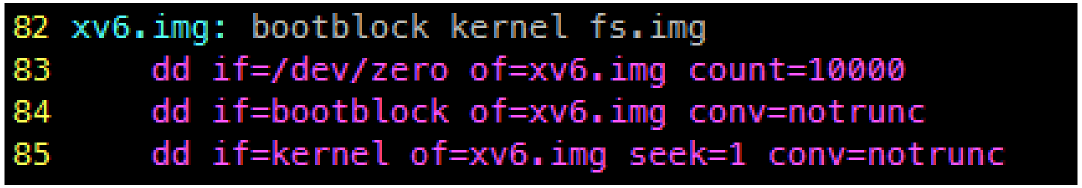
Some slides are based on Frans Kaashoek's slides
kaashoek@mit.edu

Review

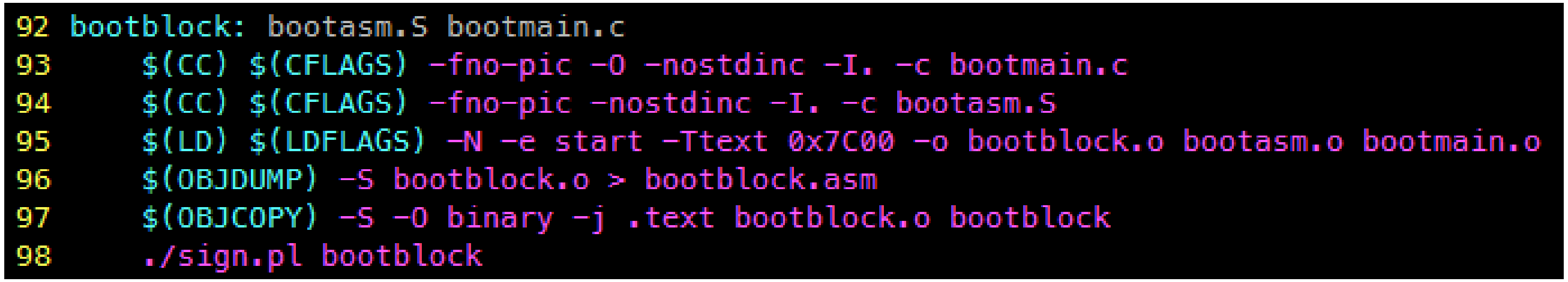
- PC Architecture
  - The von Neumann Model
- X86
  - Modes: real mode / protected mode / SMM / …
  - Registers: EFLAGS, CR[0-4], GDTR, IDTR, …
  - Memory model: 16-bit / 20-bit / 32-bit / 36-bit / 64-bit
    - Memory-mapped I/O
    - Low memory: 0-640 KB, extended memory: above 1MB
  - Booting
    - BIOS: 0xFFFF0 (what is seg, what is offset?)
    - First instruction: 0x7c00 (what instruction?)

XV6 BOOTING CODE

Makefile

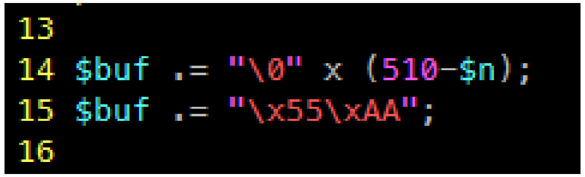


```
xv6.img: bootblock kernel fs.img
	dd if=/dev/zero of=xv6.img count=10000
	dd if=bootblock of=xv6.img conv=notrunc
	dd if=kernel of=xv6.img seek=1 conv=notrunc
```

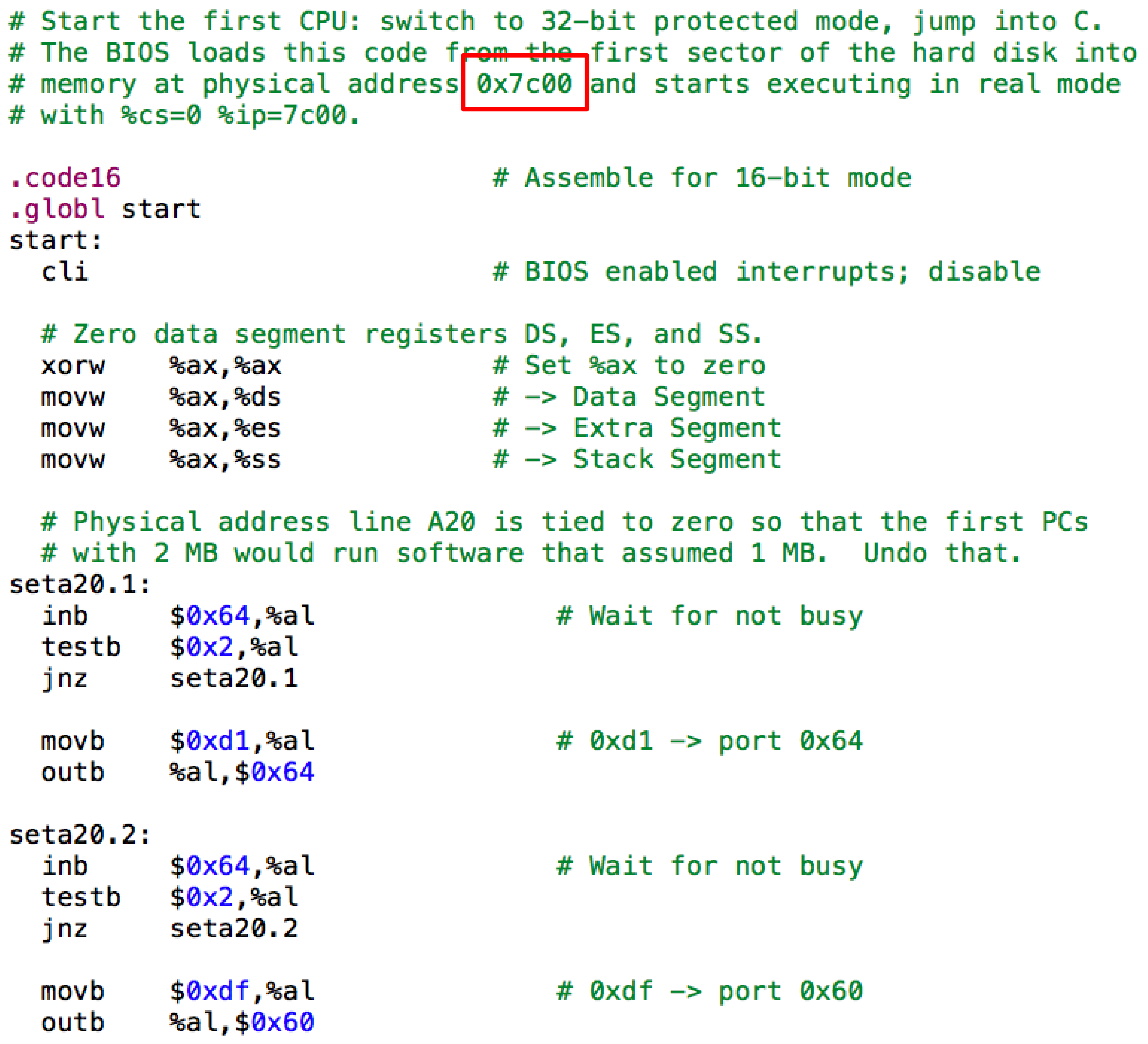


```
bootblock: bootasm.S bootmain.c
	$(CC) $(CFLAGS) -fno-pic -O -nostdinc -I. -c bootmain.c
	$(CC) $(CFLAGS) -fno-pic -nostdinc -I. -c bootasm.S
	$(LD) $(LDFLAGS) -N -e start -Ttext 0x7C00 -o bootblock.o bootasm.o bootmain.o
	$(OBJDUMP) -S bootblock.o > bootblock.asm
	$(OBJCOPY) -S -O binary -j .text bootblock.o bootblock
	./sign.pl bootblock
```

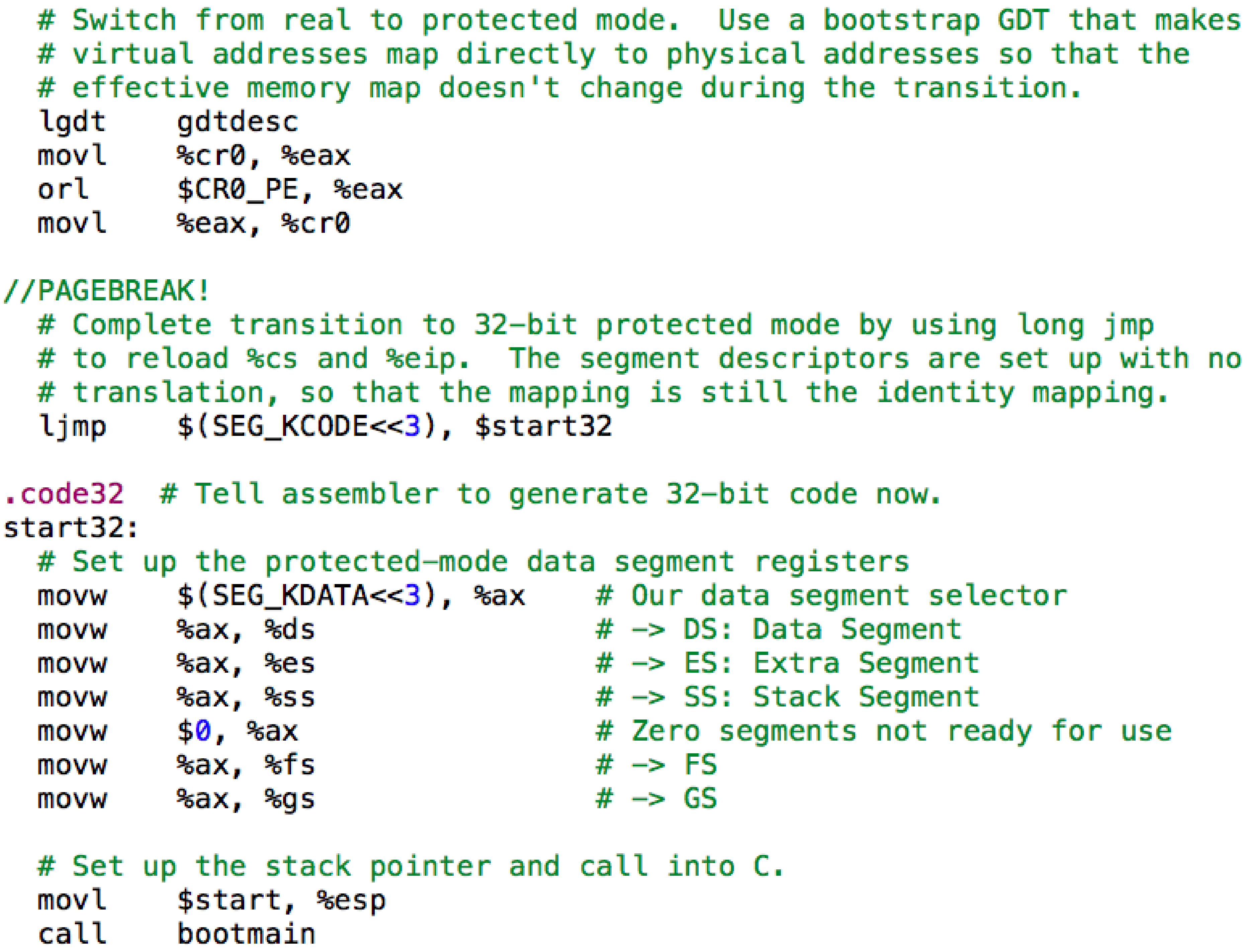
Sign.pl



```

$buf .= "\0" x (510-$n);
$buf .= "\x55\xAA";

```

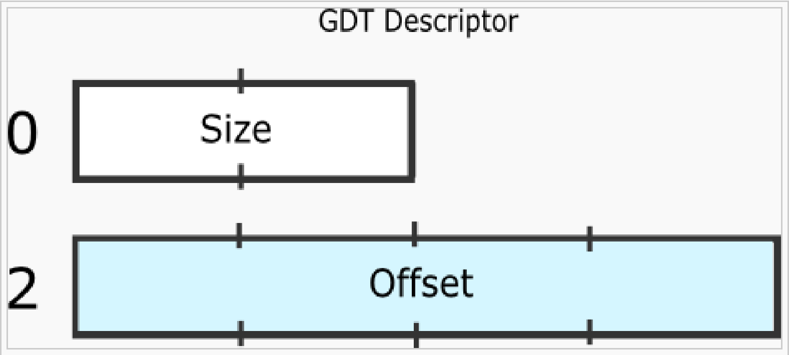


```
# Start the first CPU: switch to 32-bit protected mode, jump into C.
# The BIOS loads this code from the first sector of the hard disk into
# memory at physical address 0x7c00 and starts executing in real mode
# with %cs=0 %ip=7c00.

.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
```

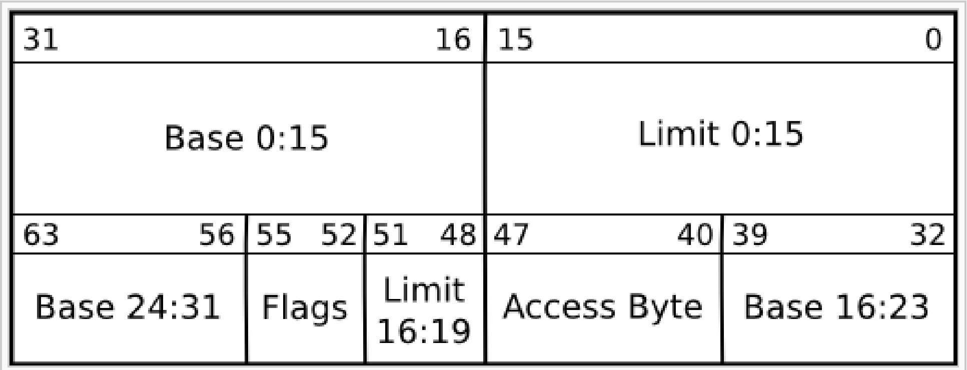


```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//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
```

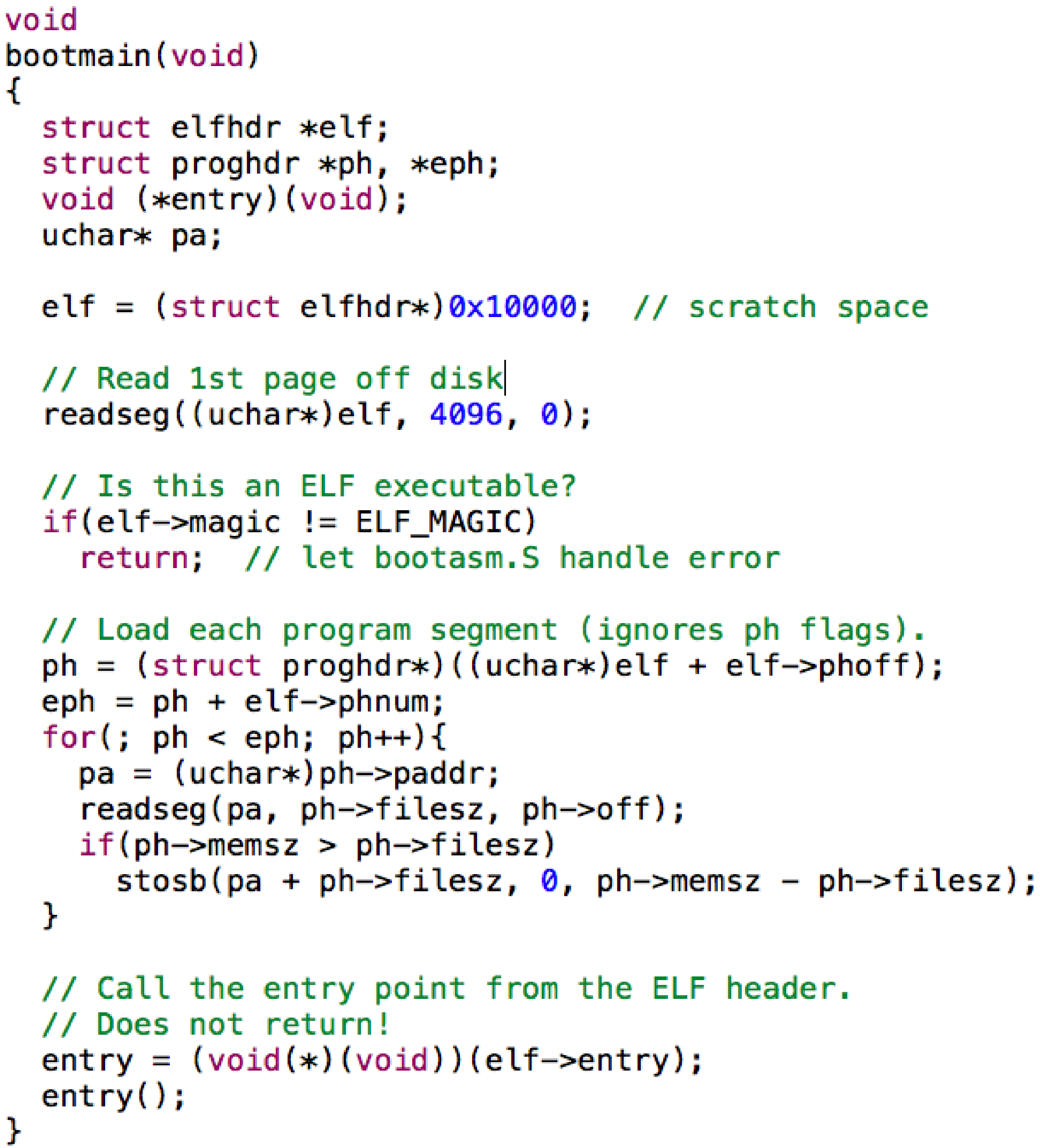


GDTR

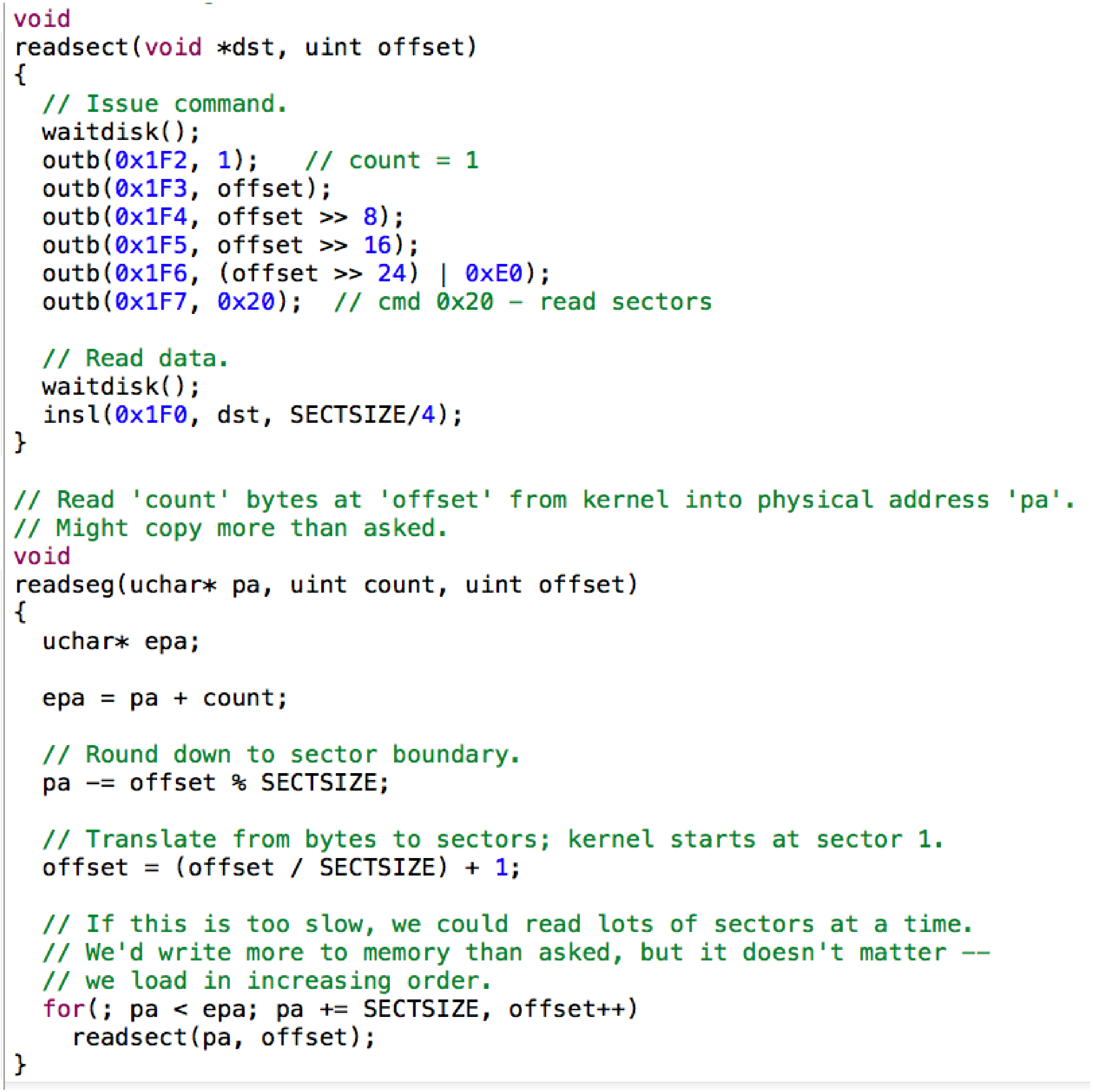


A GDT Entry

http://wiki.osdev.org/Global_Descriptor_Table



```
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 Header :

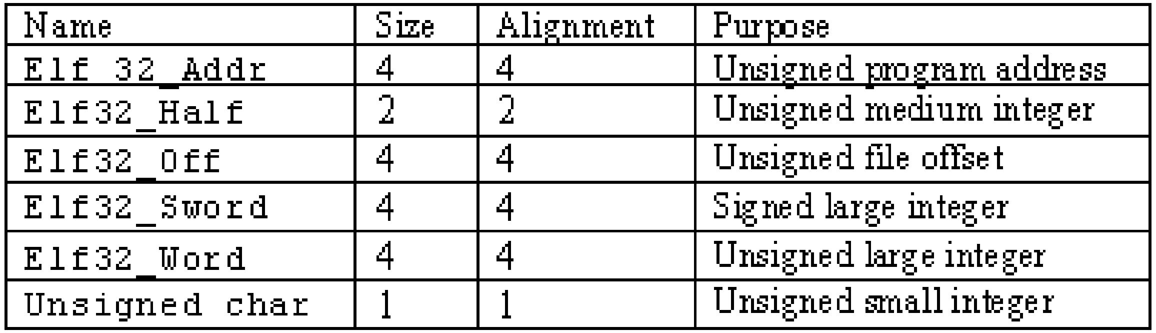
ELF header resides at the beginning and holds a road map describing the files organization

```
#define EI_NIDENT 16
typedef struct {
    unsigned char e_ident [EI_NIDENT] ;
    Elf32_Half e_type;
    Elf32_Half e_machine ;
    Elf32_Word e_version ;
    Elf32_Addr e_entry ;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags ;
    Elf32_Half e_ehsize ;
    Elf32_Half e_phentsize ;
    Elf32_Half e_phnum ;
    Elf32_Half e_shentsize ;
    Elf32_Half e_shnum ;
    Elf32_Half e_shstrndx ;
}Elf32_Ehdr ;
```

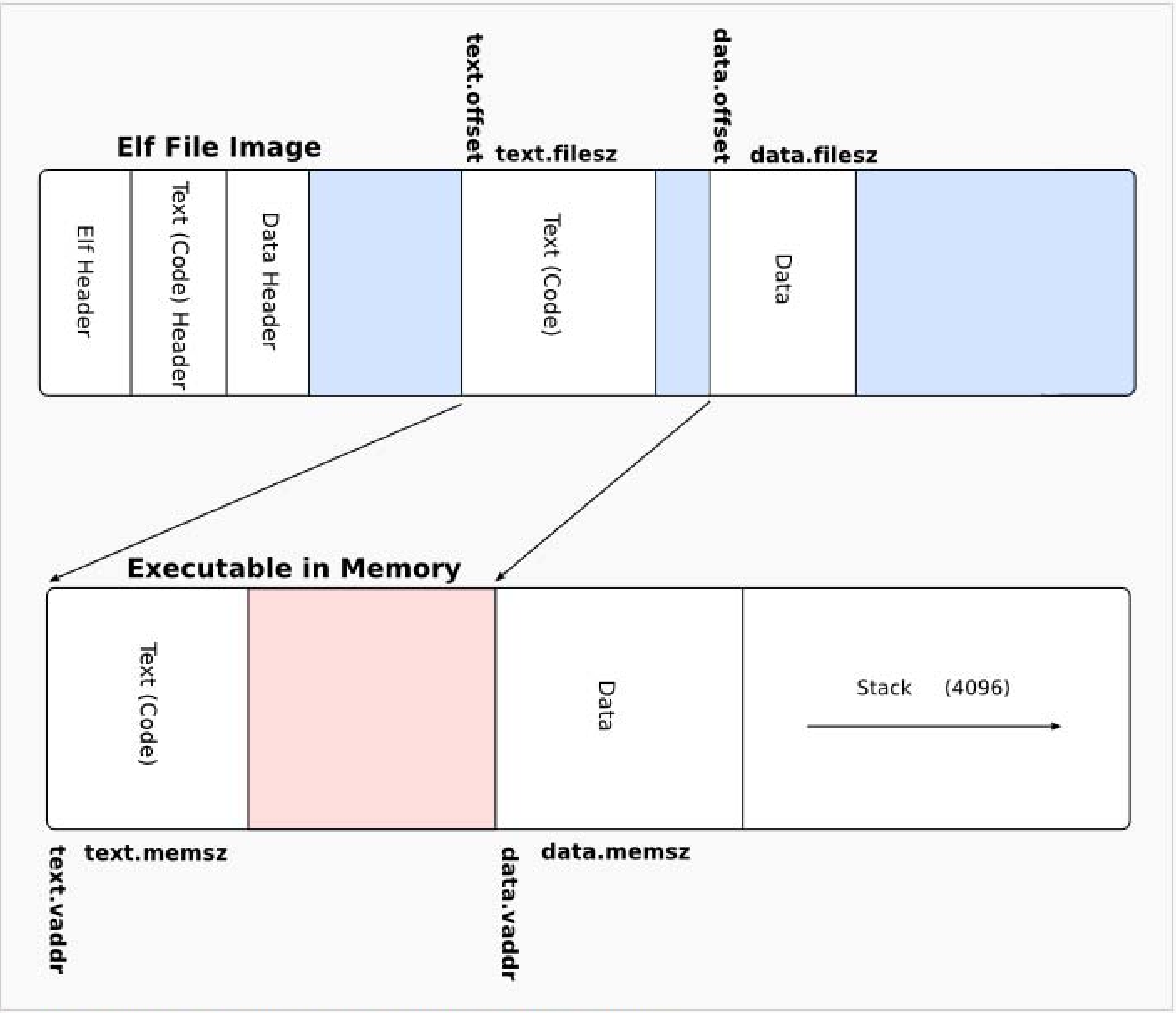
e_machine : specifies the required architecture :**0** = No machine … **3** = Intel 80386 **4** = Motorola 68000 …**8** = MIPS RS3000

e_entry : the virtual address to which the system first transfers control, when starting the process. If the file has no associated entry point, this member holds zero.

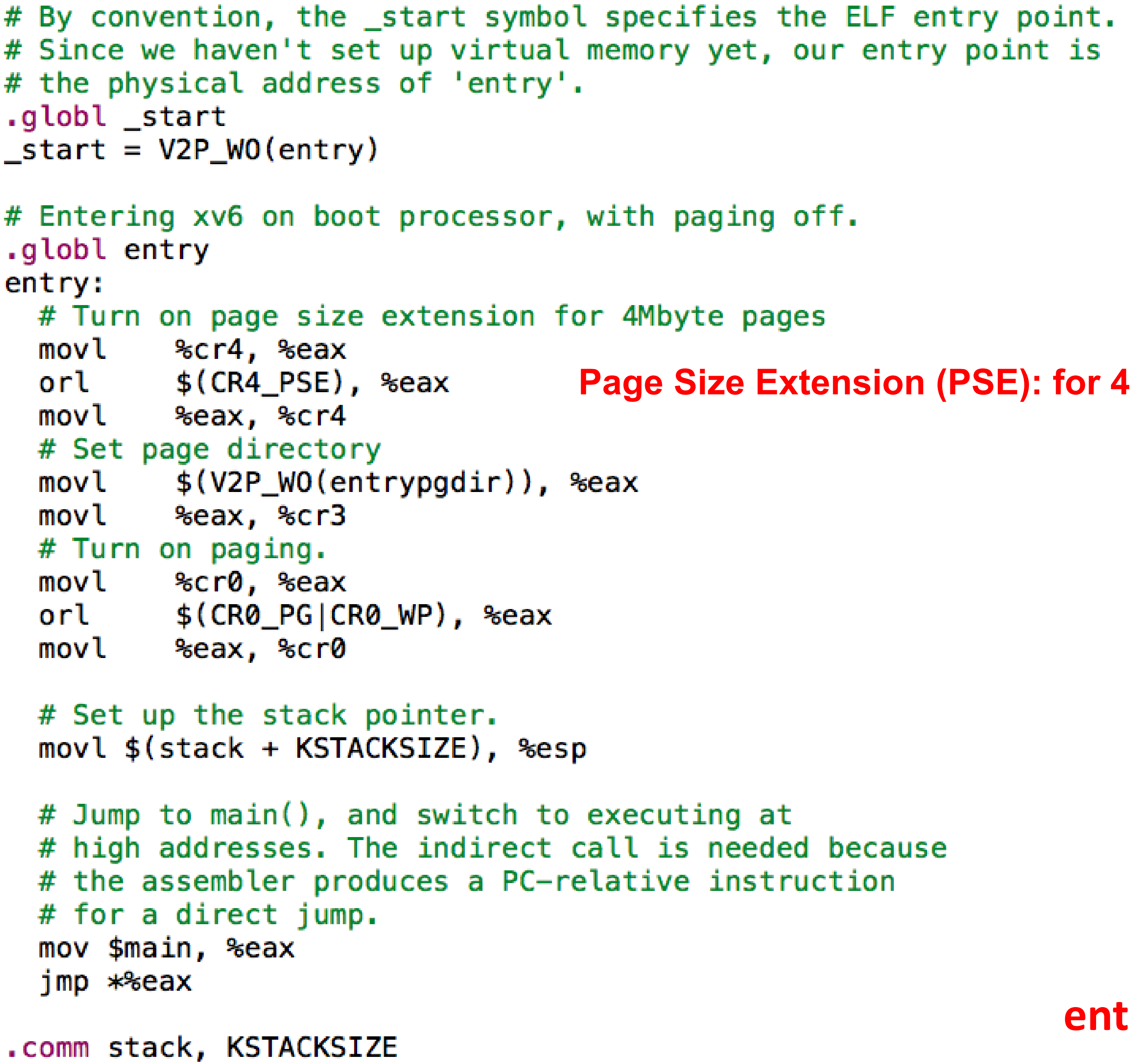
e_shoff : holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.



Name	Size	Alignment	Purpose
Elf 32_Addr	4	4	Unsigned program address
Elf32_Half	2	2	Unsigned medium integer
Elf32_Off	4	4	Unsigned file offset
Elf32_Sword	4	4	Signed large integer
Elf32_Word	4	4	Unsigned large integer
Unsigned char	1	1	Unsigned small integer



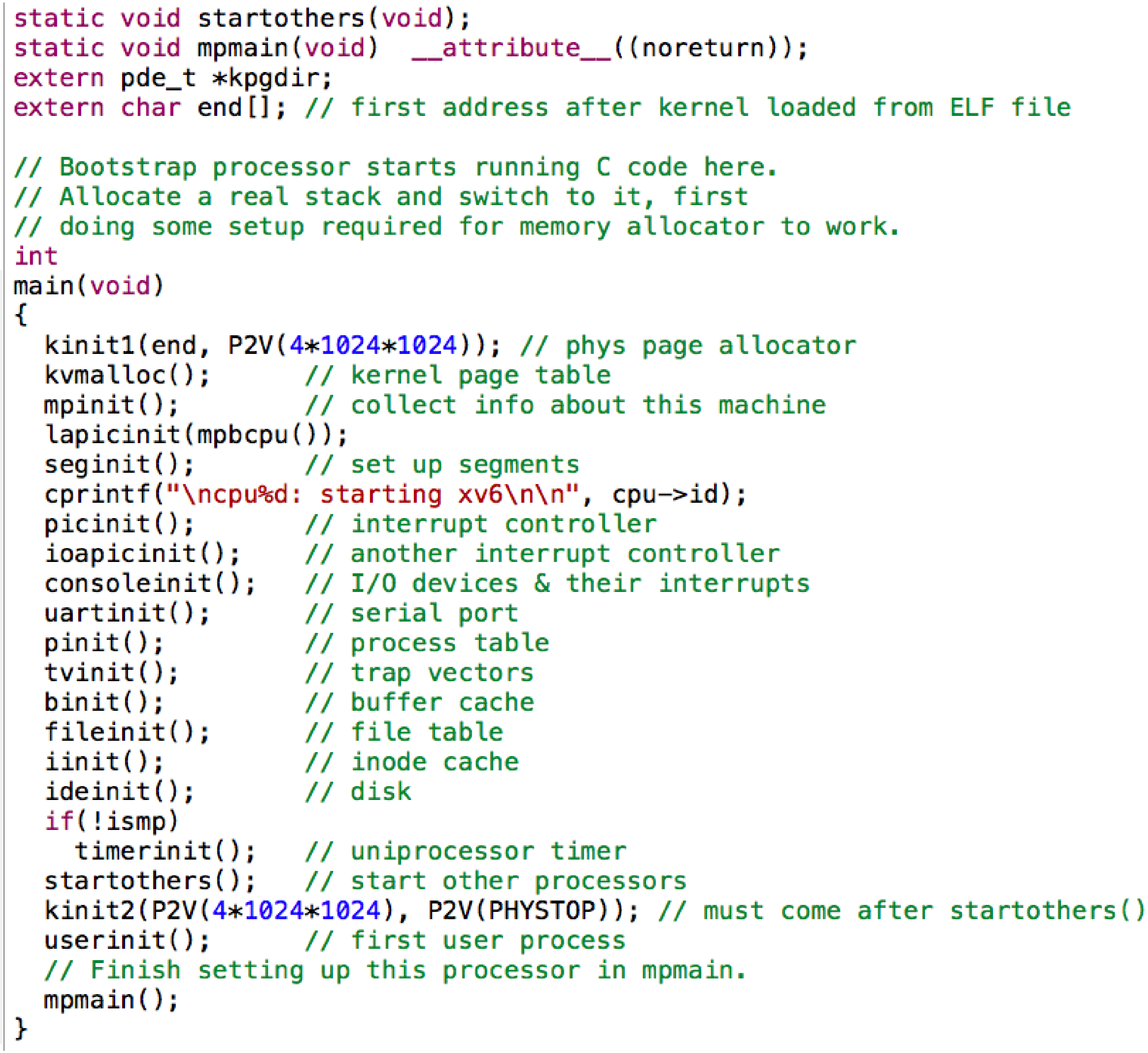
Executable image and elf binary can being mapped onto each other



```
# 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_WO(entry)

# Entering xv6 on boot processor, with paging off.
.globl entry
entry:
  # Turn on page size extension for 4Mbyte pages
  movl    %cr4, %eax
  orl     $(CR4_PSE), %eax
  movl    %eax, %cr4
  # Set page directory
  movl    $(V2P_WO(entrypgdir)), %eax
  movl    %eax, %cr3
  # Turn on paging.
  movl    %cr0, %eax
  orl     $(CR0_PG|CR0_WP), %eax
  movl    %eax, %cr0

  # Set up the stack pointer.
  movl $(stack + KSTACKSIZE), %esp

  # Jump to main(), and switch to executing at
  # high addresses. The indirect call is needed because
  # the assembler produces a PC-relative instruction
  # for a direct jump.
  mov $main, %eax
  jmp *%eax

.comm stack, KSTACKSIZE
```

Page Size Extension (PSE): for 4M pages

entry.S



```c
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();
}
```

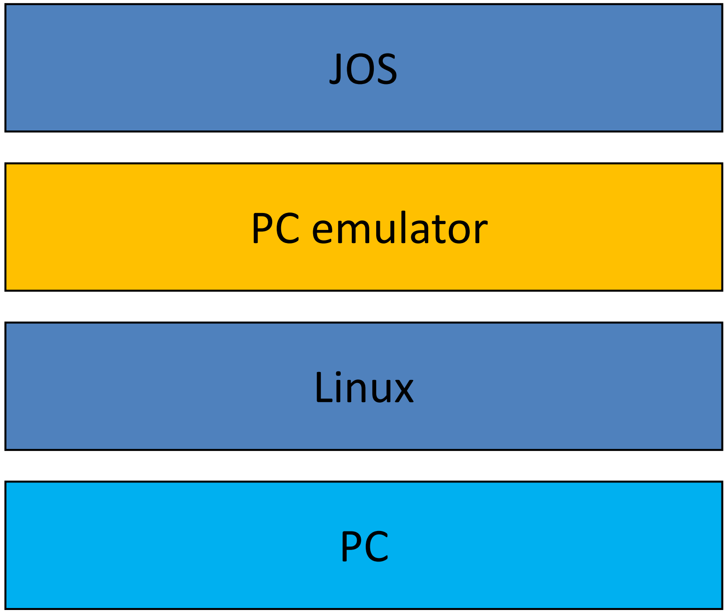
PC PROGRAMMING

- PC Architecture
- Memory
- Execution
- PC Emulation

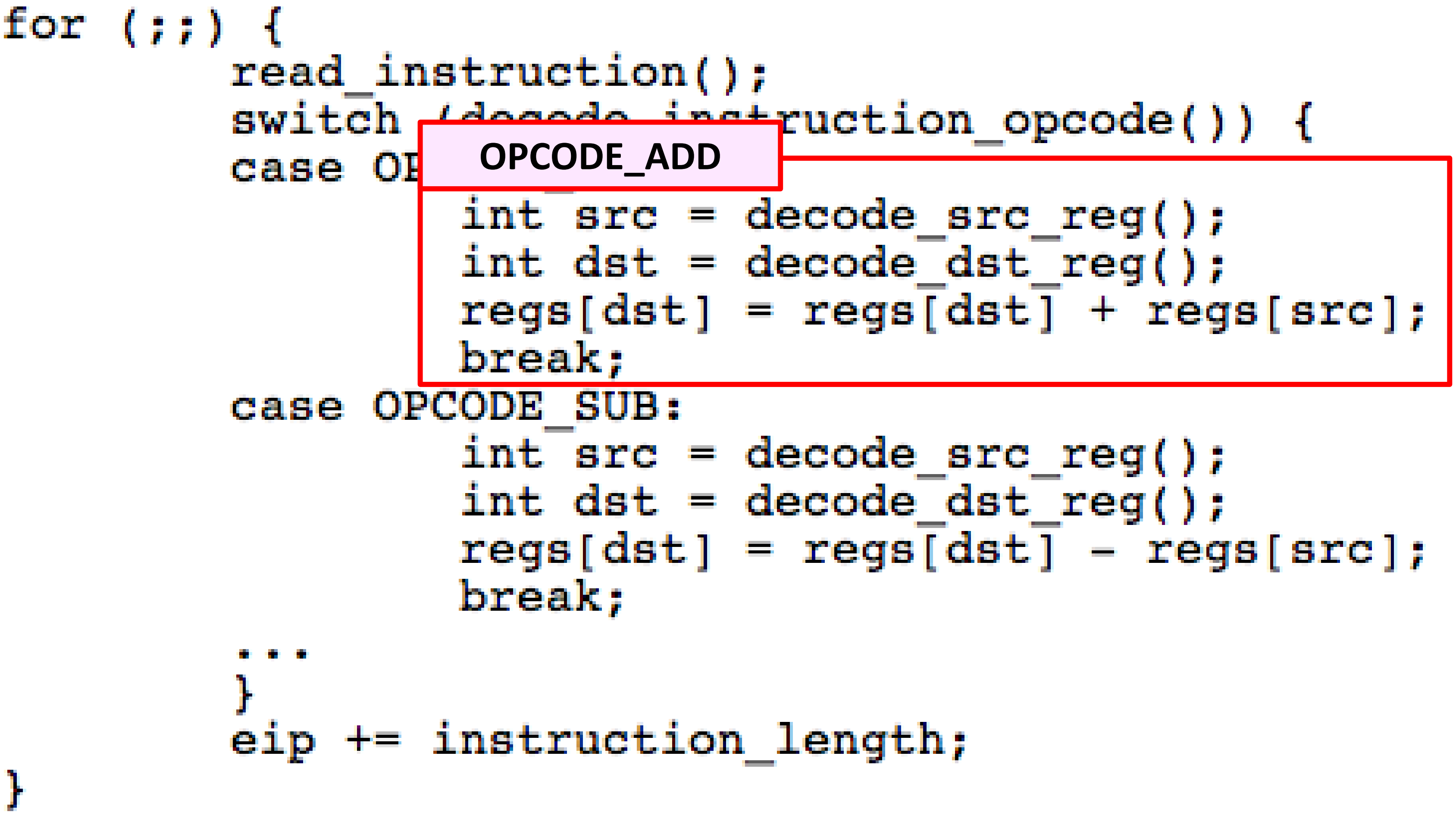
Development using PC emulator

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

Runs like a normal program on “host” operating system



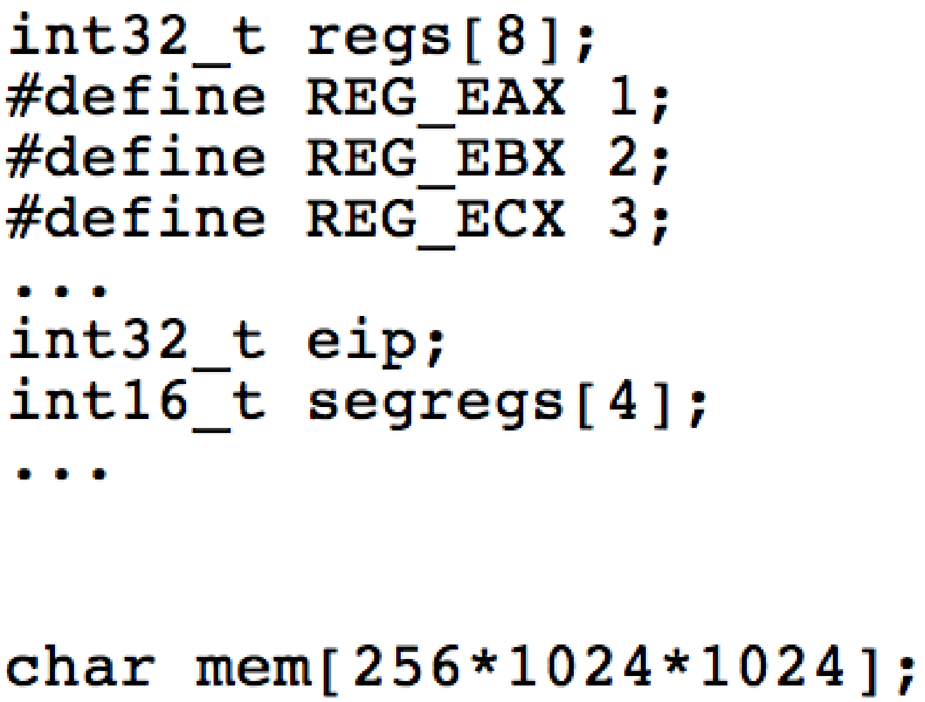
Emulation of CPU



```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OP
                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;
}
```

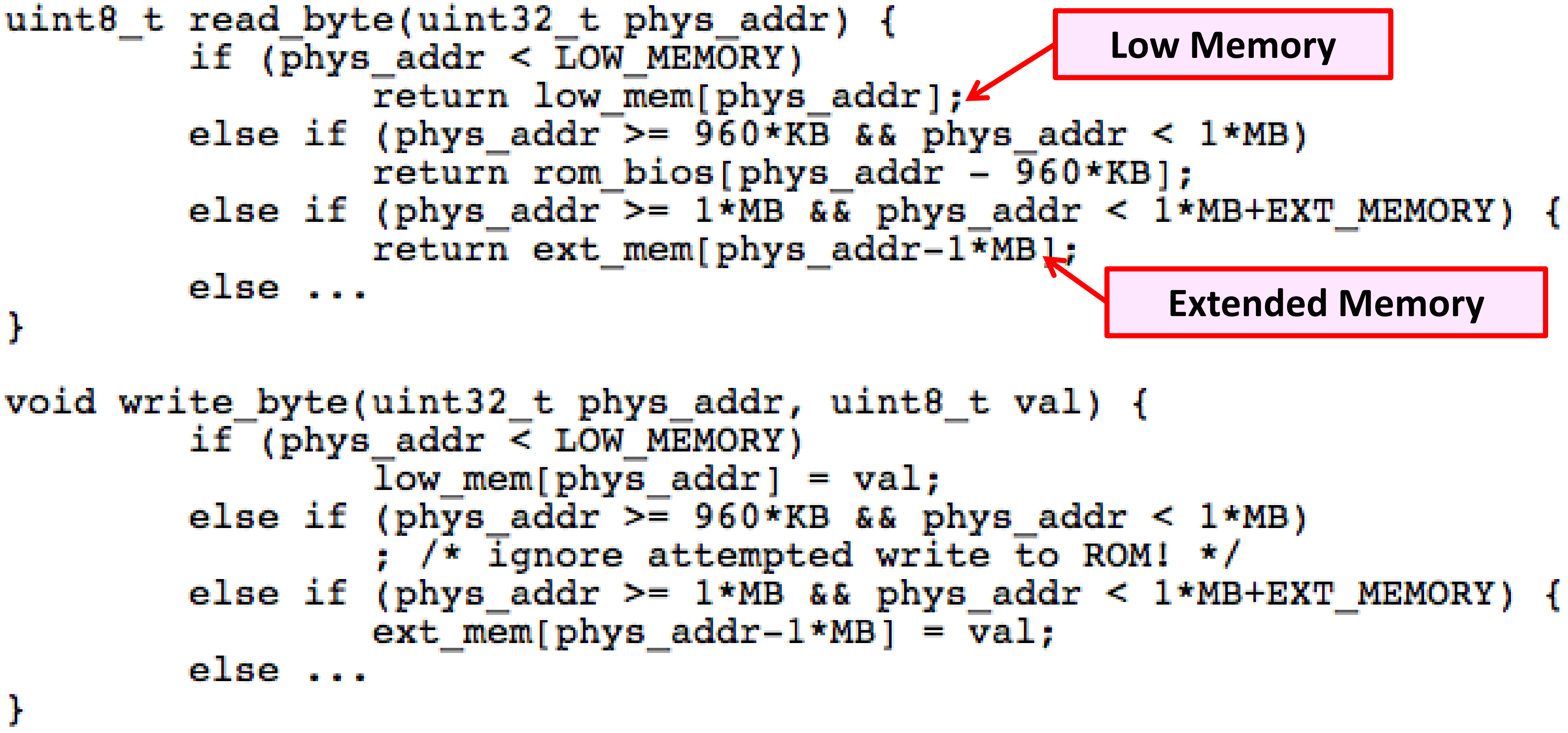
OPCODE_ADD

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 ...
}

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 ...
}
```

Low Memory

Extended Memory

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

OPERATING SYSTEM ARCHITECTURE

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

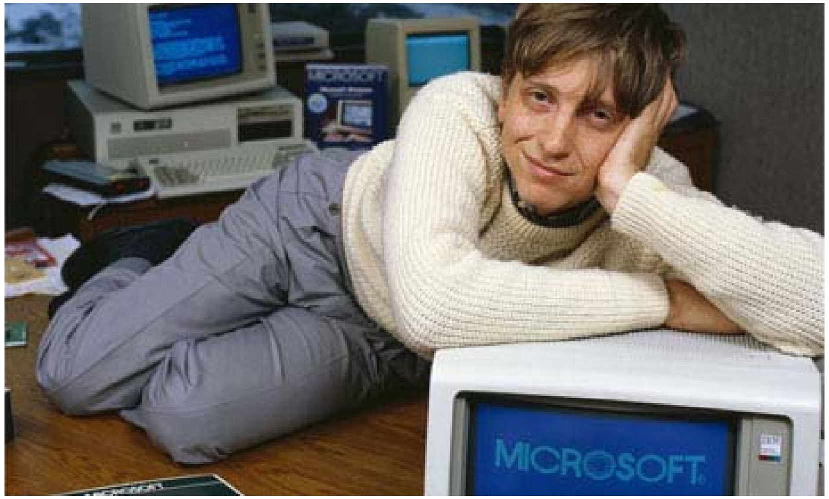
Policies decide what will be done

- The separation of policy from mechanism is a very important principle
- It allows maximum flexibility if policy decisions are to be changed later

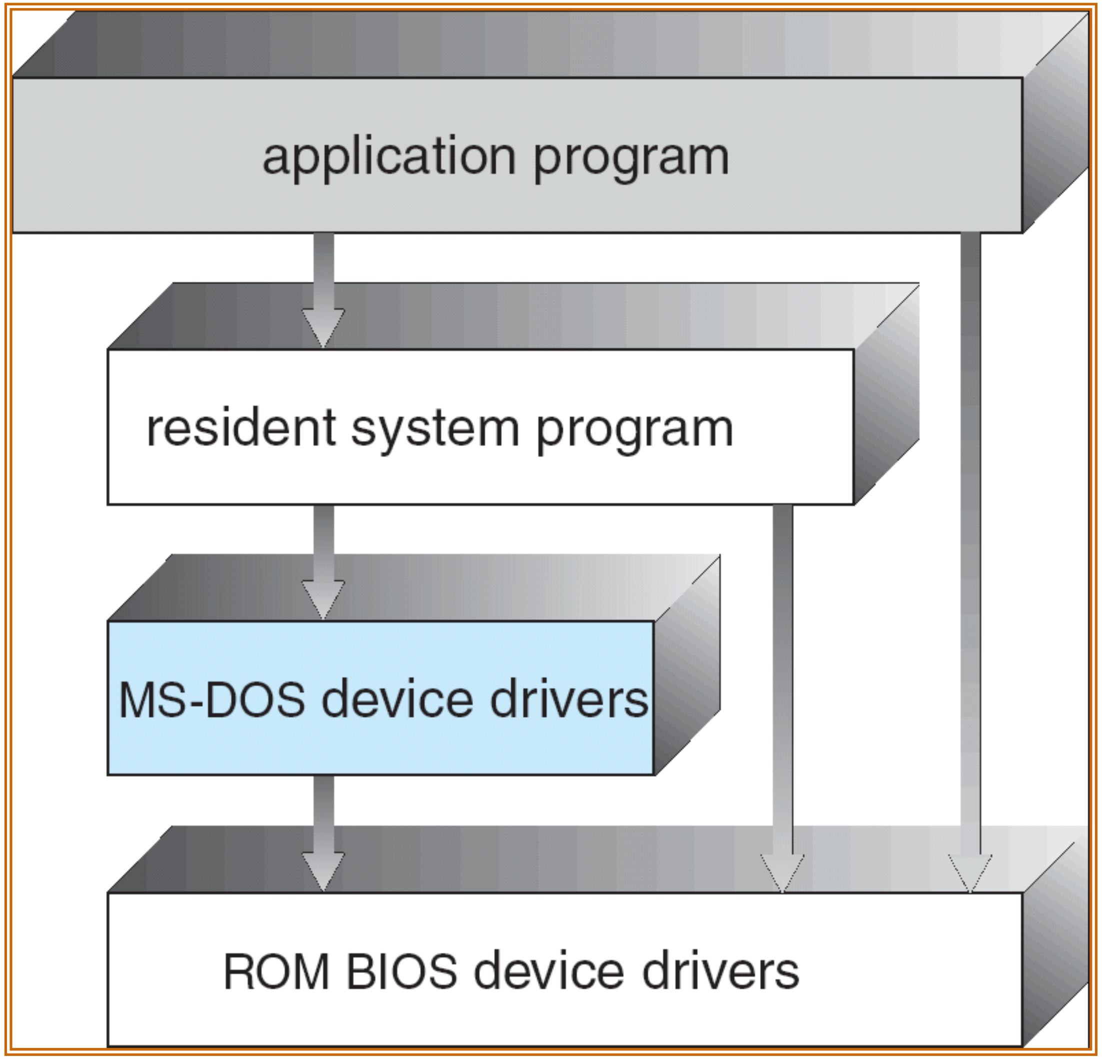
Simple Structure

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

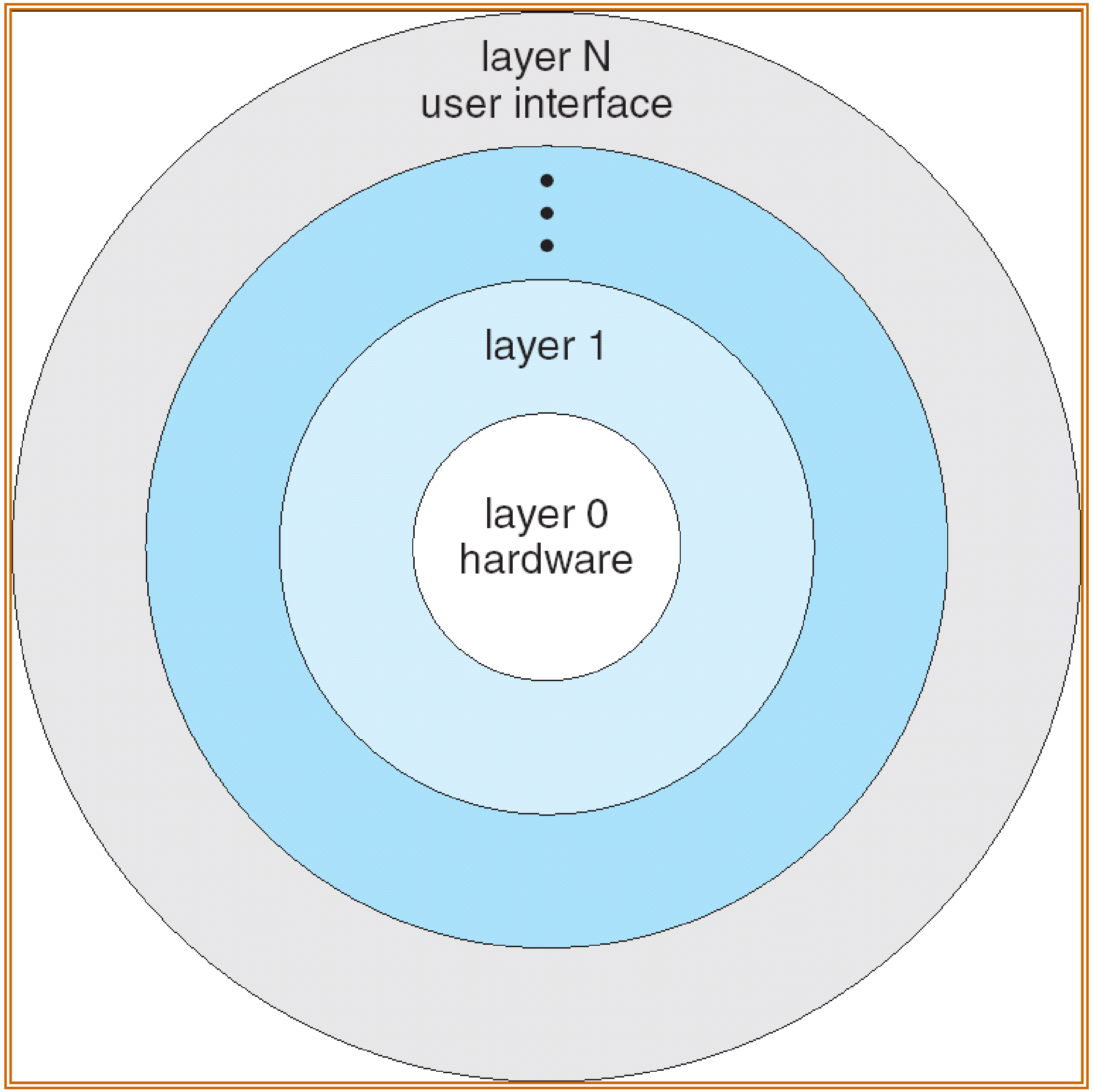
The operating system is divided into a number of layers (levels)

- 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

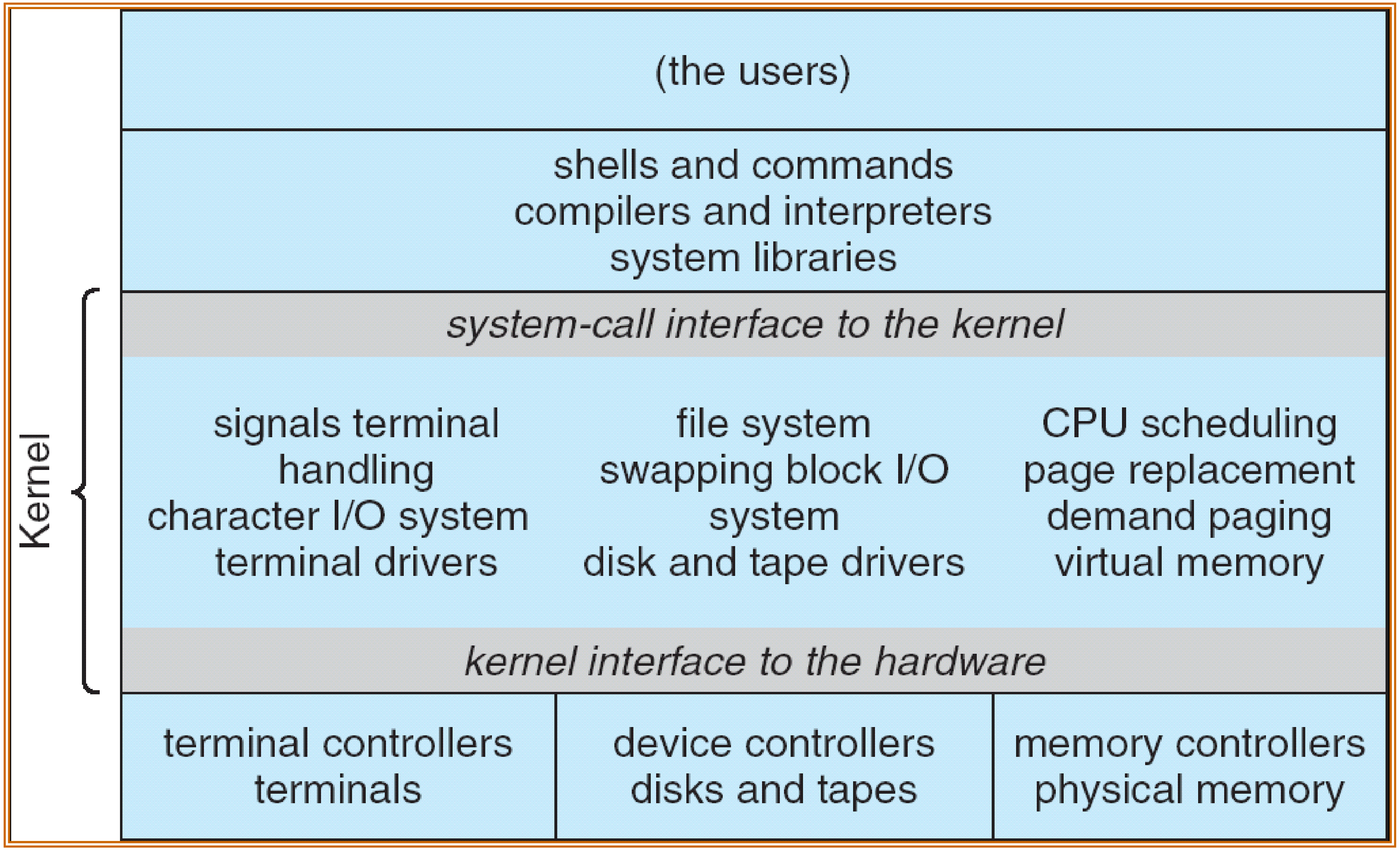
UNIX – limited by hardware functionality

- 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



(the users)

shells and commands
compilers and interpreters
system libraries

system-call interface to the kernel

signals terminal handling
character I/O system
terminal drivers

file system
swapping block I/O system
disk and tape drivers

CPU scheduling
page replacement
demand paging
virtual memory

kernel interface to the hardware

terminal controllers
terminals

device controllers
disks and tapes

memory controllers
physical memory

Kernel

Microkernel System Structure

Moves as much from the kernel into "*user*" space

Communication takes place between user modules using message passing

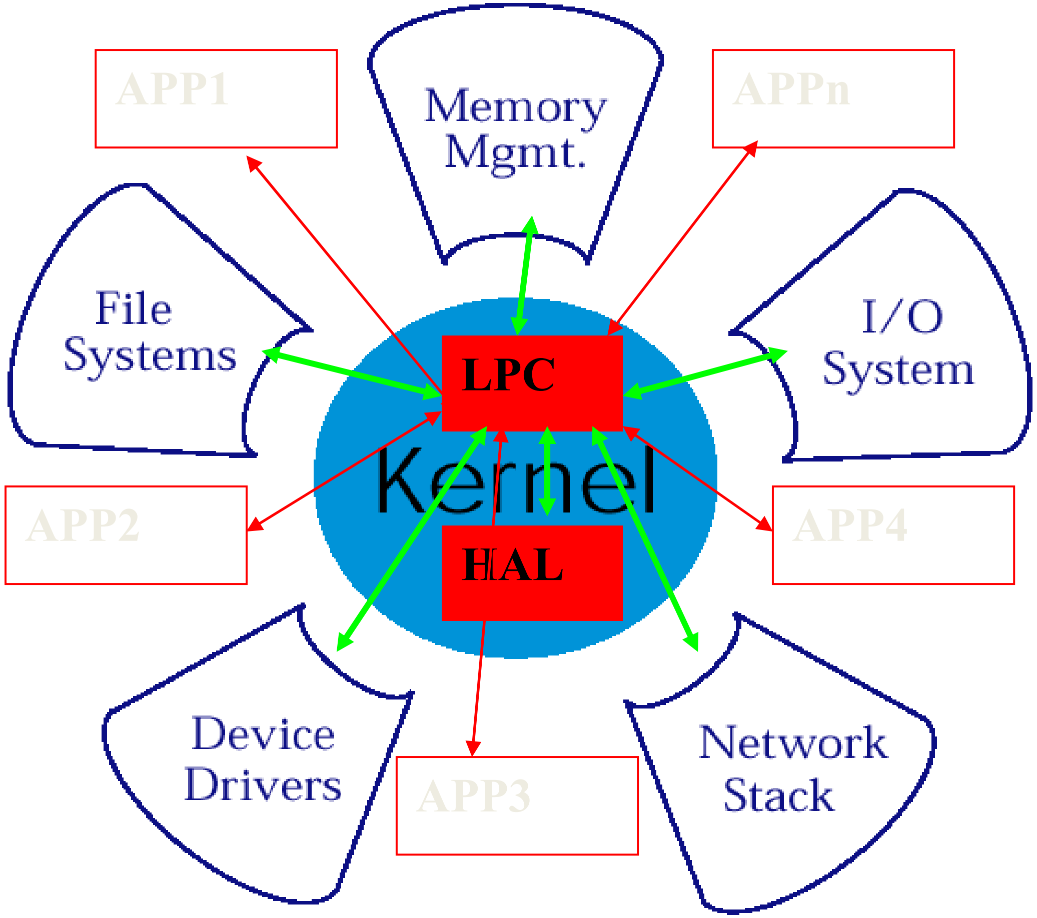
Benefits:

- 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

Detriments:

- Performance overhead of user space to kernel space communication

Microkernel Structure & Instances



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
- Summery

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).

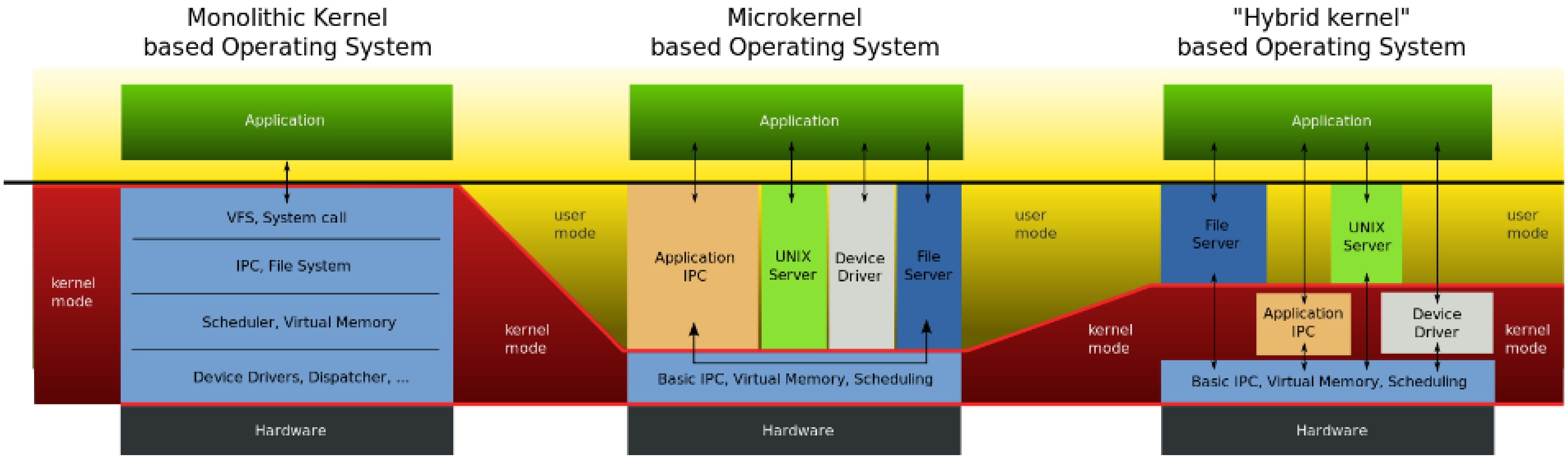
low-level address space management, thread management, and inter-process communication (IPC) and so on …

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
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel
mode
Hardware
Application
user
mode
Application
IPC
UNIX
Server
Device
Driver
File
Server
kernel
mode
Basic IPC, Virtual Memory, Scheduling
Hardware
Application
user
mode
File
Server
UNIX
Server
user
mode
Application
IPC
Device
Driver
kernel
mode
kernel
mode
Basic IPC, Virtual Memory, Scheduling
Hardware

Questions

What’s the shortcoming of those kernels

Hint: that’s the reason to create exokernel

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

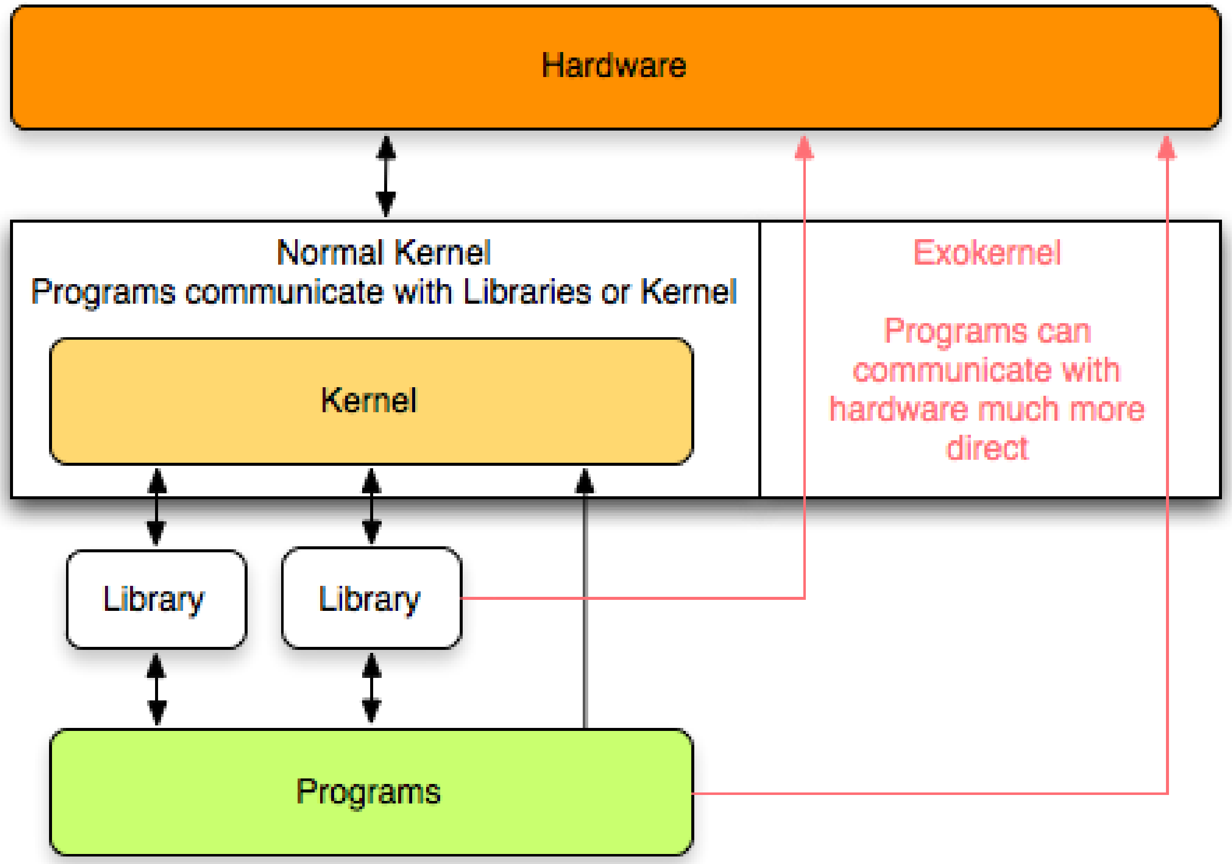
The libraries hide low-level resources

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

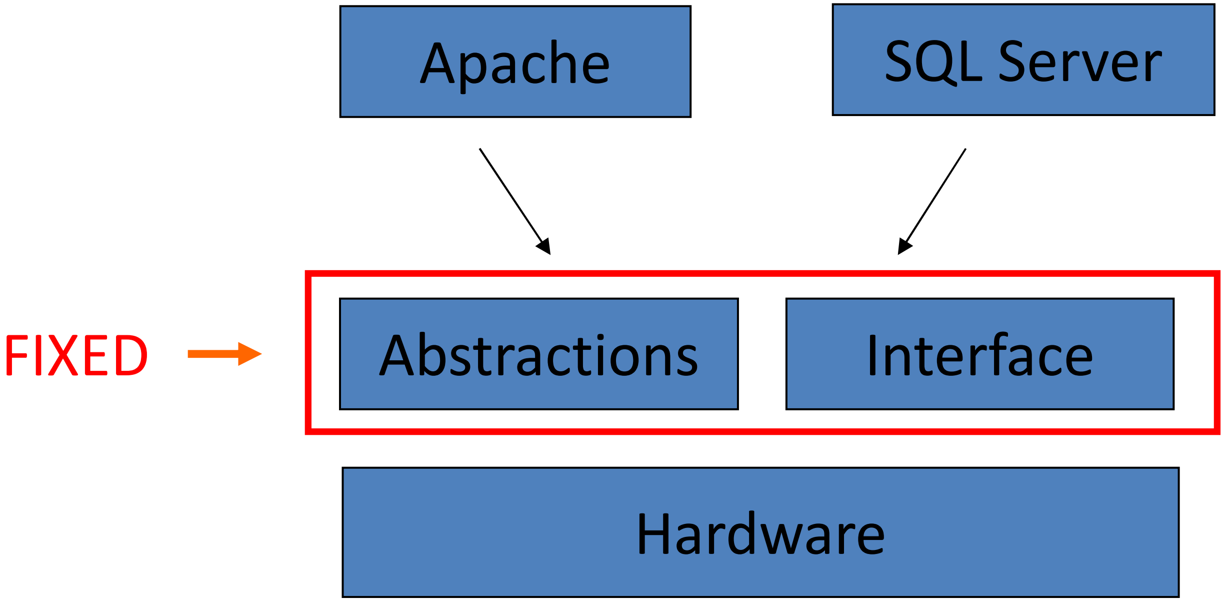
Exokernel Example 1


Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with hardware much more direct
Library
Library
Programs

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

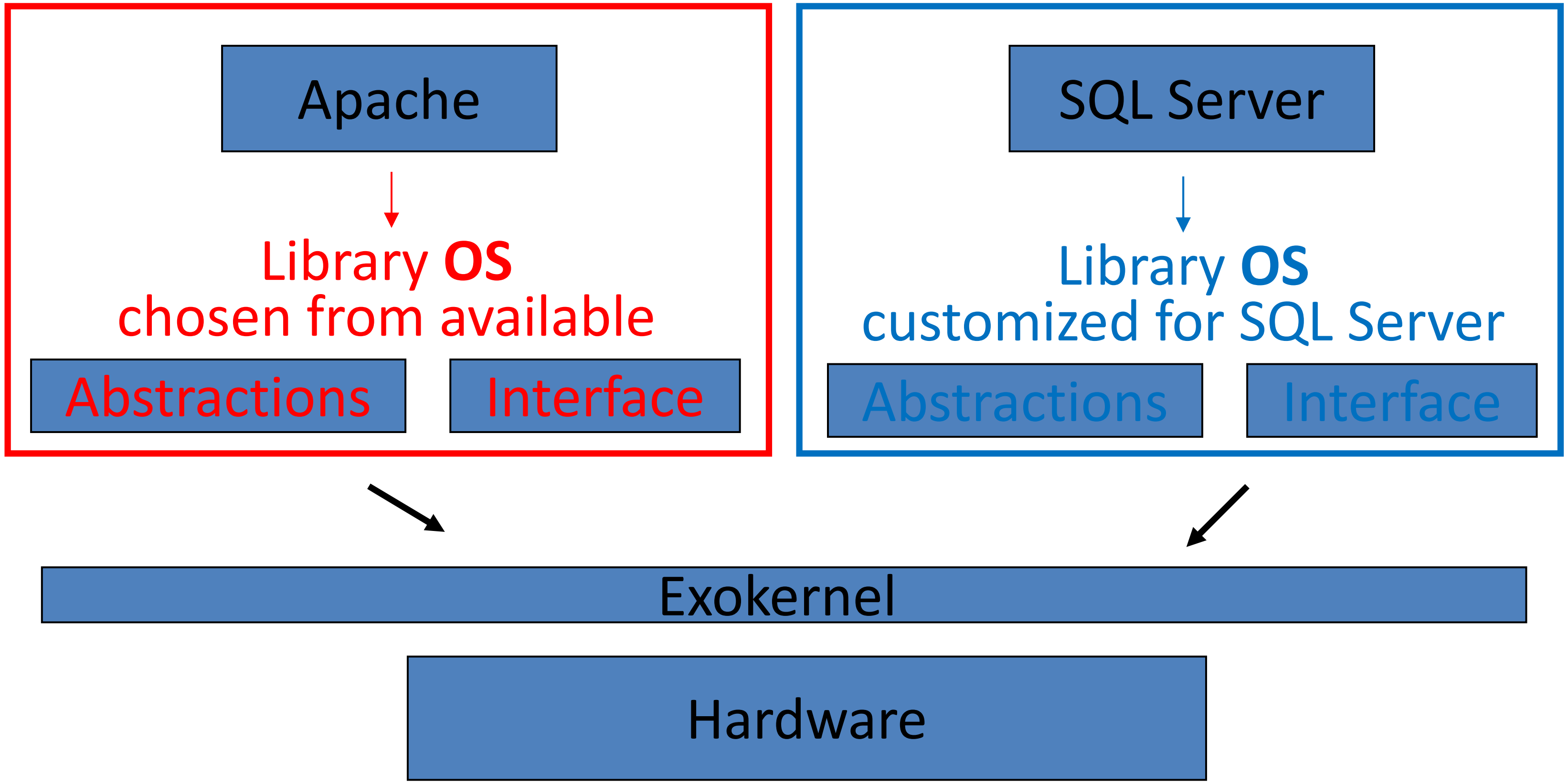
Exokernel Example 2

Traditional **OS**



Exokernel Example 2

Exo**kernel** + 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

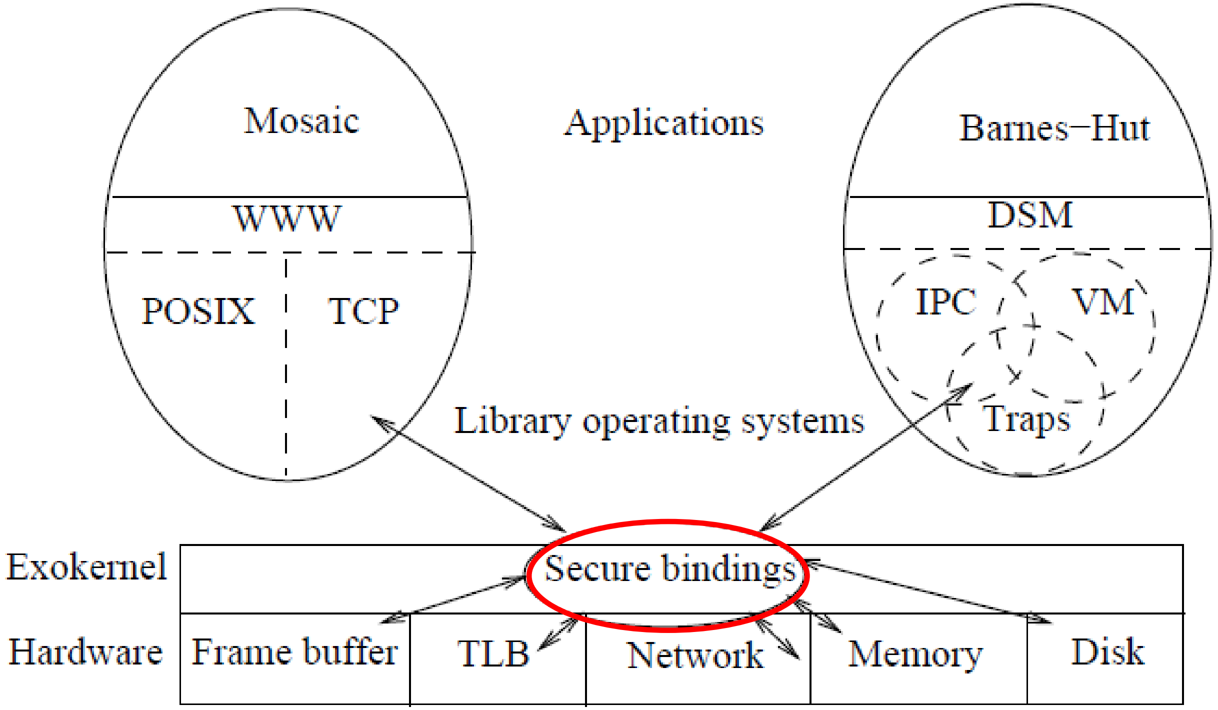
The abort protocol

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



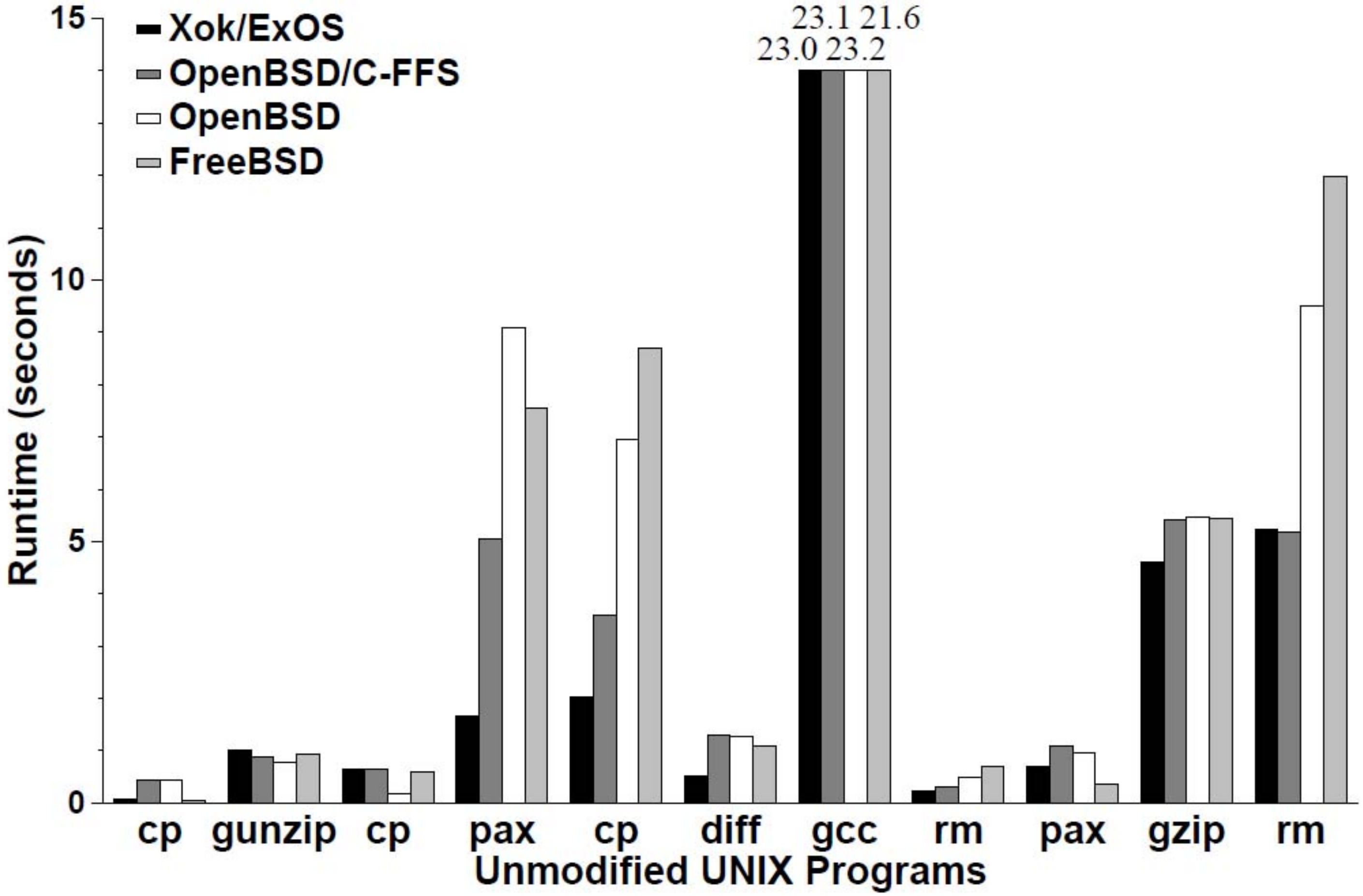
An example exokernel-based system consisting of a thin exokernel veneer that exports resources to library operating systems through secure bindings.

Implementation

Prototype (Xok / ExOS)

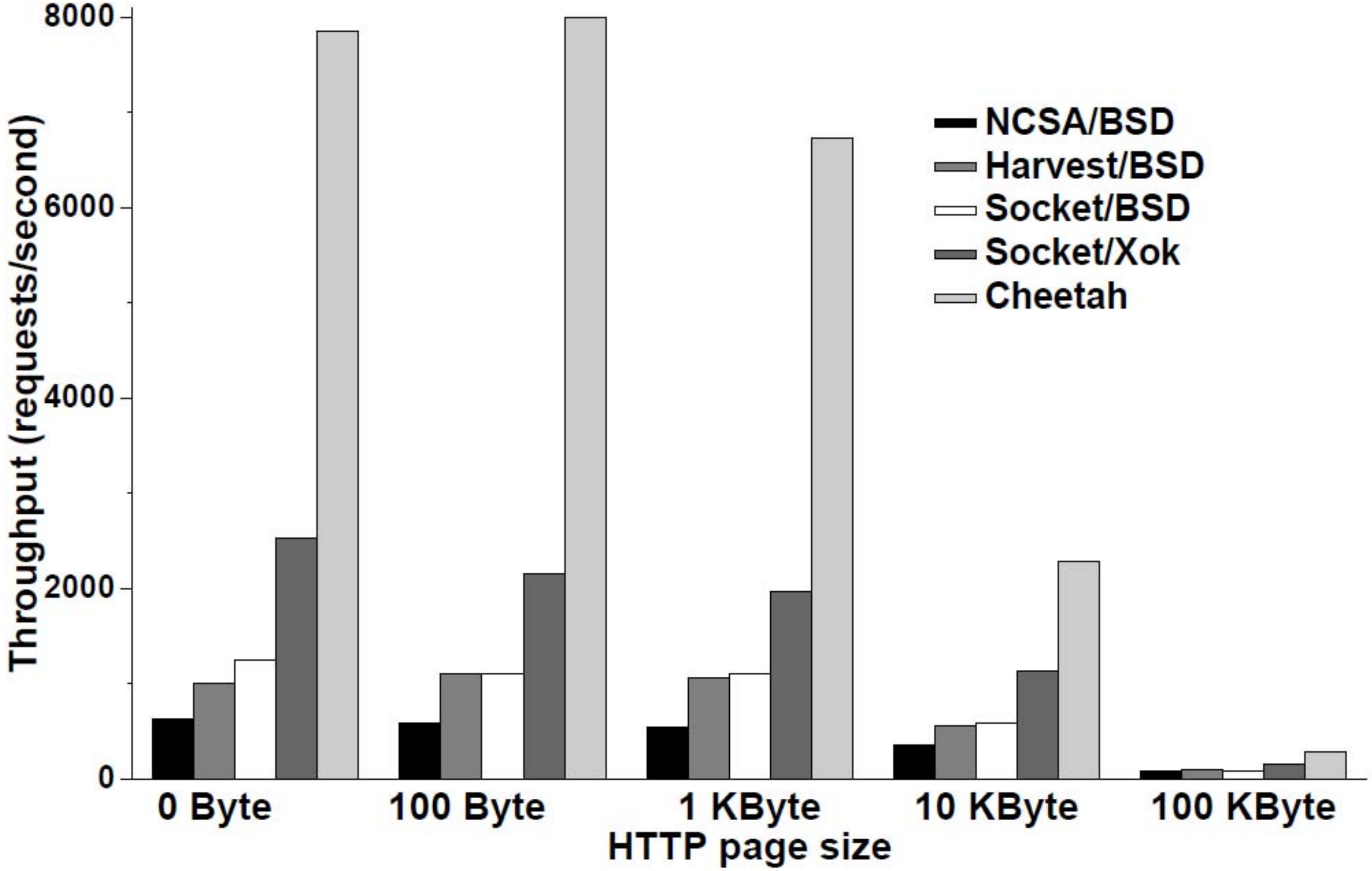
- Xok
  - **Exokernel**
  - Runs on x86 (Aegis runs on DEC)
  - Safely multiplexes the physical resources
- ExOS
  - **Library OS**
  - Manages fundamental OS abstractions at application level
    - completely within the address space of the application that is using it

Evaluation



Application Performance and Flexibility on Exokernel Systems, SOSP'97

Evaluation



Application-level control can significantly improve the performance of applications

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

Summary

OS evolve constantly

Exokernel: application manages resources

OS Structure & XV6 VM

Recap

- OS functionalities

- OS structure

Review

- Q: what does an OS do?
- Q: what are the components of an OS?

Kernels

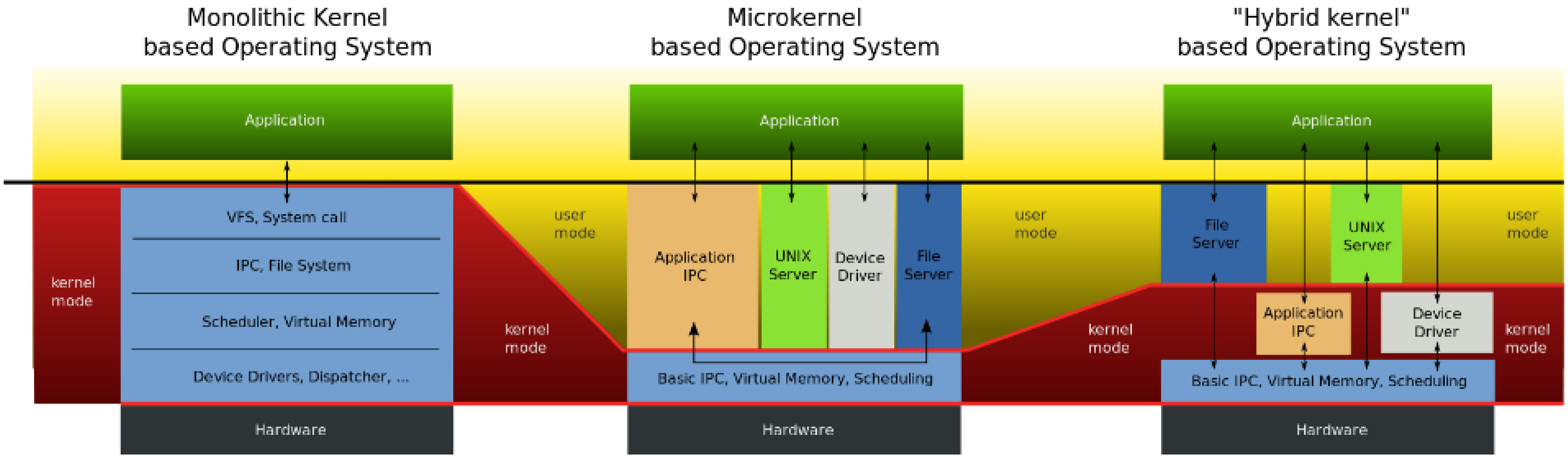
Kernel classification (wikipedia)

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…


Monolithic Kernel
based Operating System
Microkernel
based Operating System
"Hybrid kernel"
based Operating System
Application
VFS, System call
IPC, File System
Scheduler, Virtual Memory
Device Drivers, Dispatcher, ...
kernel
mode
Hardware
Application
user
mode
Application
IPC
UNIX
Server
Device
Driver
File
Server
kernel
mode
Basic IPC, Virtual Memory, Scheduling
Hardware
Application
user
mode
File
Server
UNIX
Server
user
mode
kernel
mode
Application
IPC
Device
Driver
kernel
mode
Basic IPC, Virtual Memory, Scheduling
Hardware

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

Microkernel System Structure

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

Communication takes place between user modules using message passing

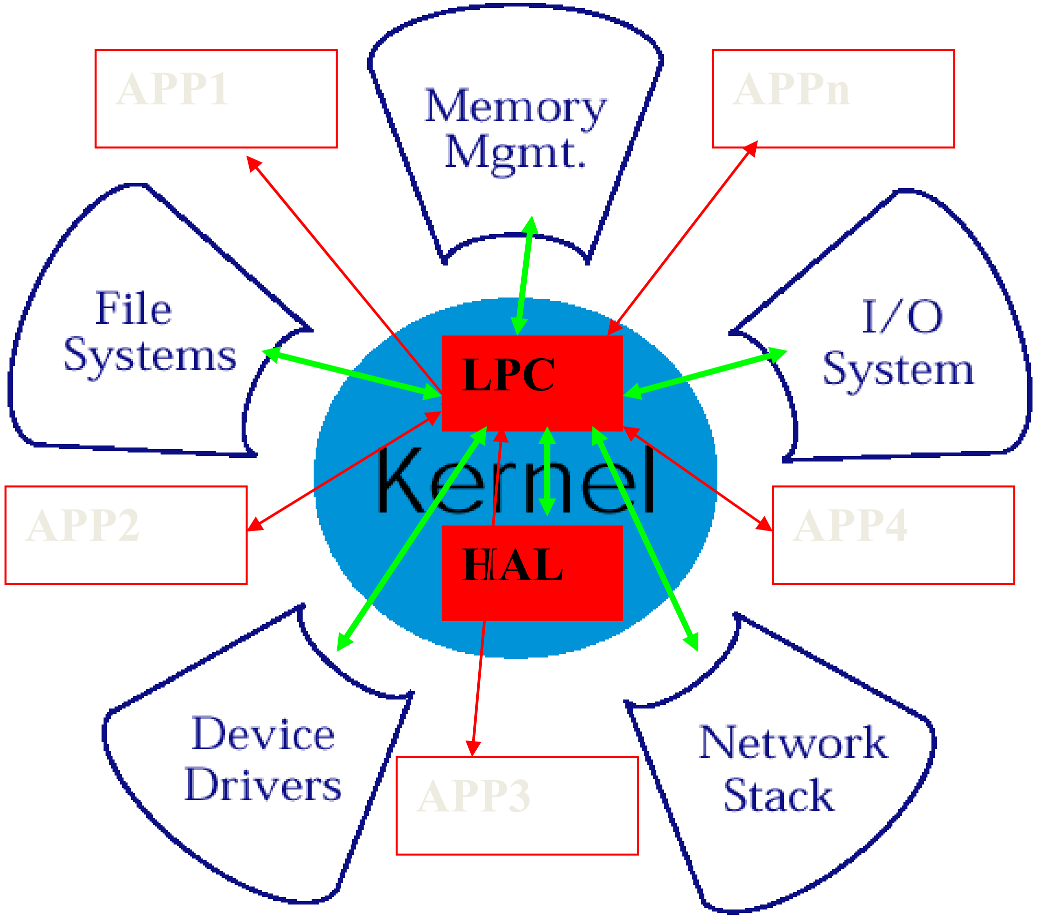
Benefits:

- 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

Detriments:

- Performance overhead of user space to kernel space communication

Microkernel Structure & Instances



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
- Summery

Questions

What’s the shortcoming of those kernels

Hint: that’s the reason to create exokernel

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

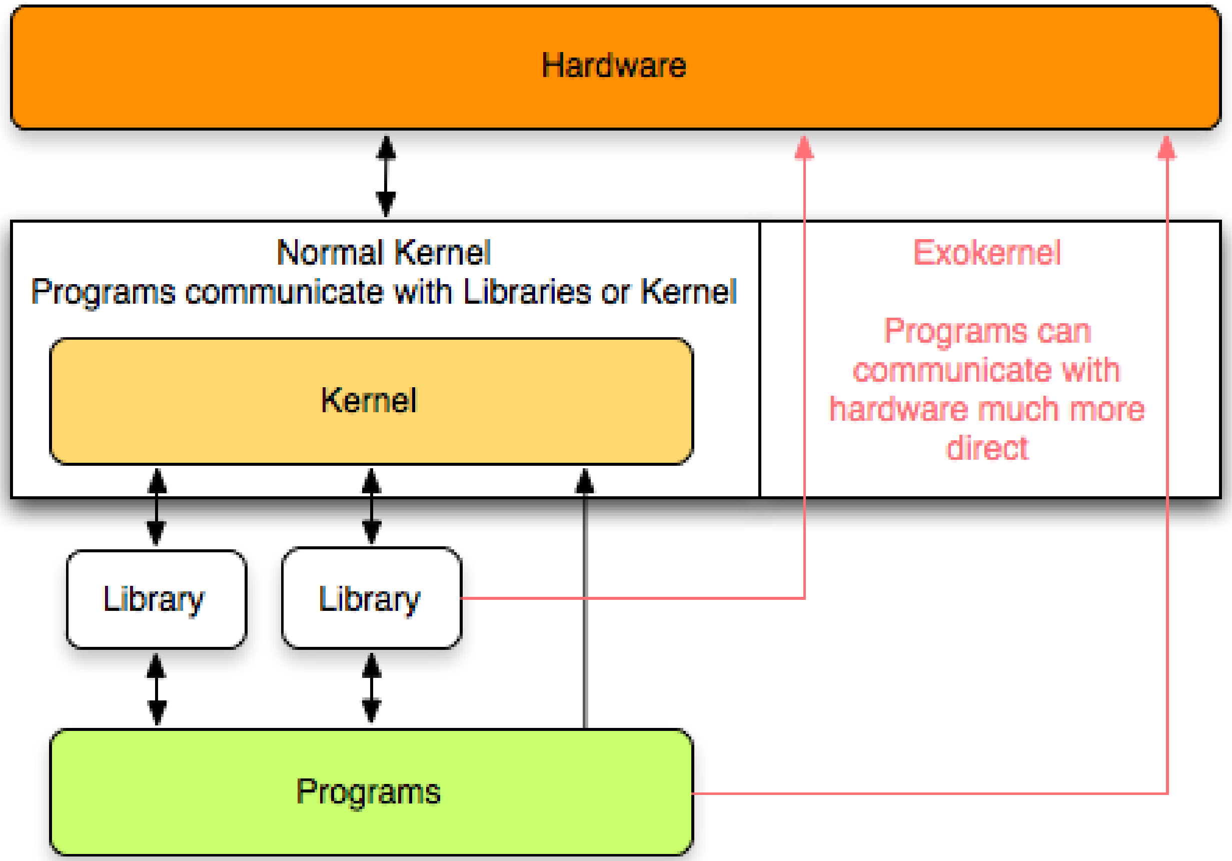
- The libraries hide low-level resources

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

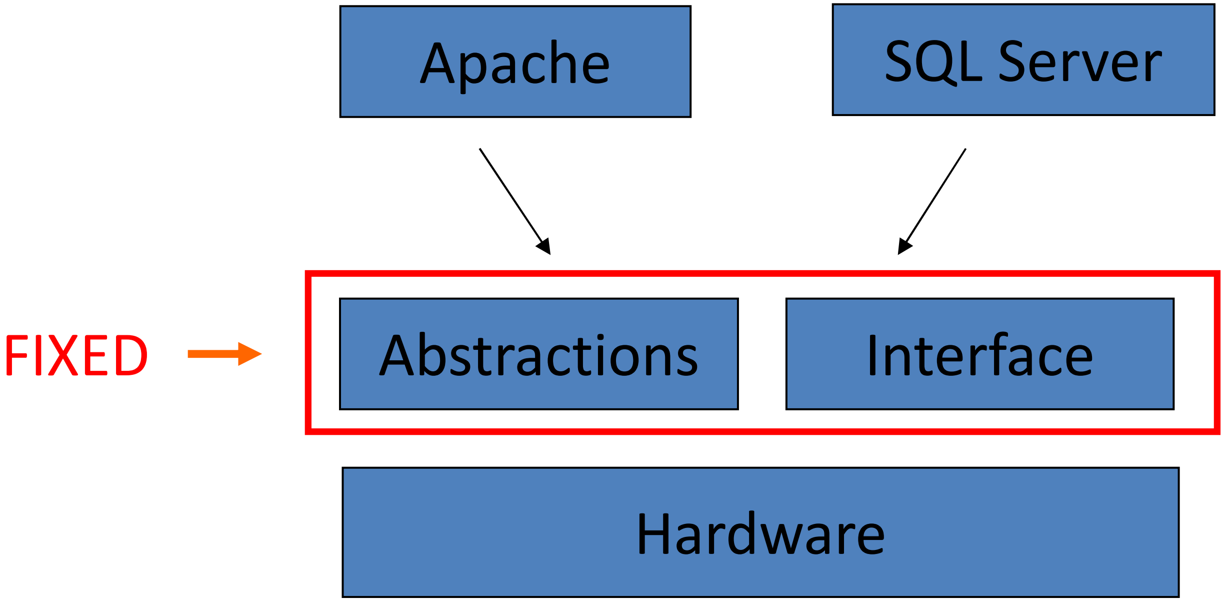
Exokernel Example 1


Hardware
Normal Kernel
Programs communicate with Libraries or Kernel
Kernel
Exokernel
Programs can communicate with hardware much more direct
Library
Library
Programs

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

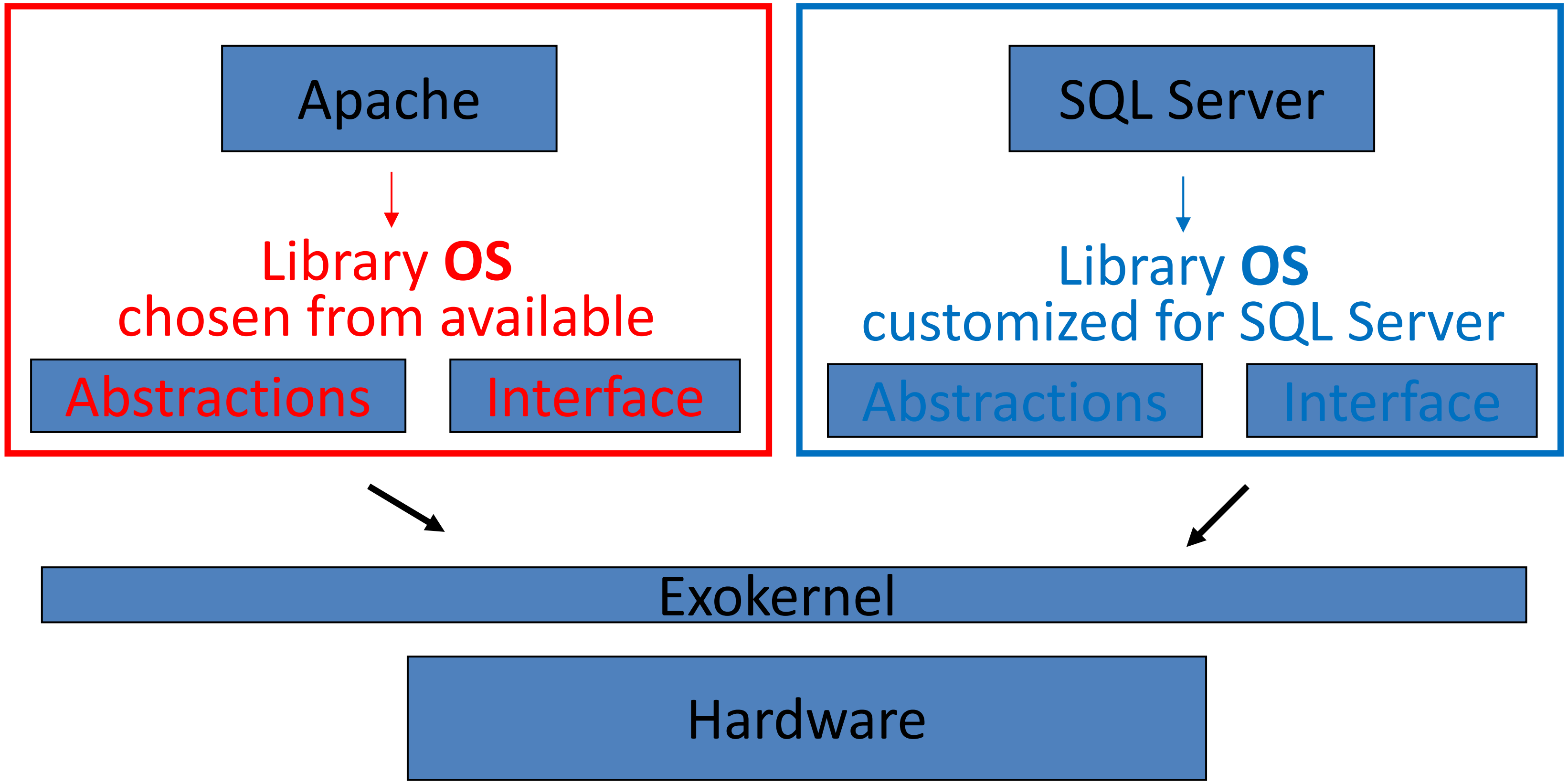
Exokernel Example 2

Traditional **OS**

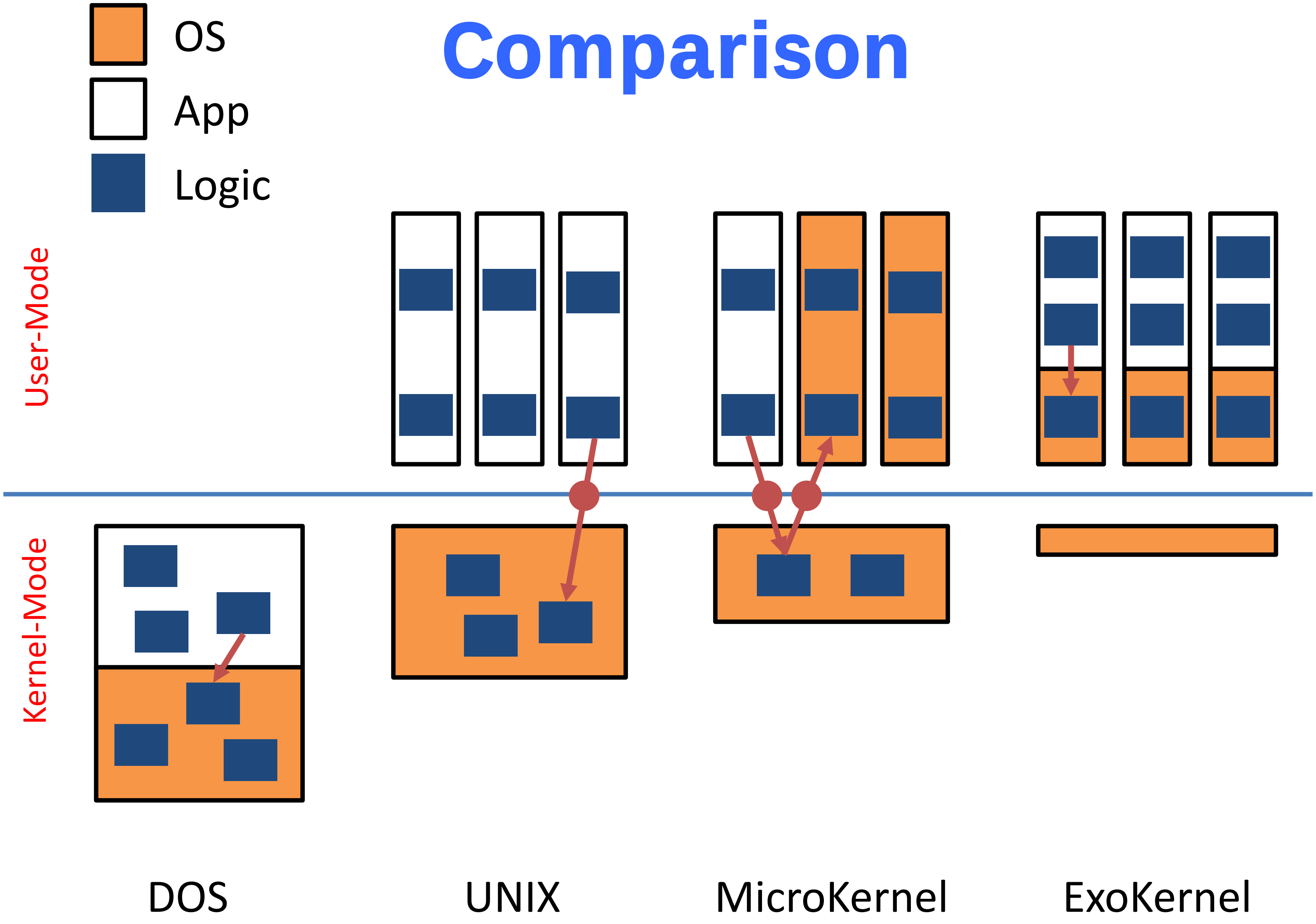


Exokernel Example 2

Exo**kernel** + Library **OS**



Comparison



🟧 OS

□ App

🟦 Logic

User-Mode

Kernel-Mode

DOS

UNIX

MicroKernel

ExoKernel

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

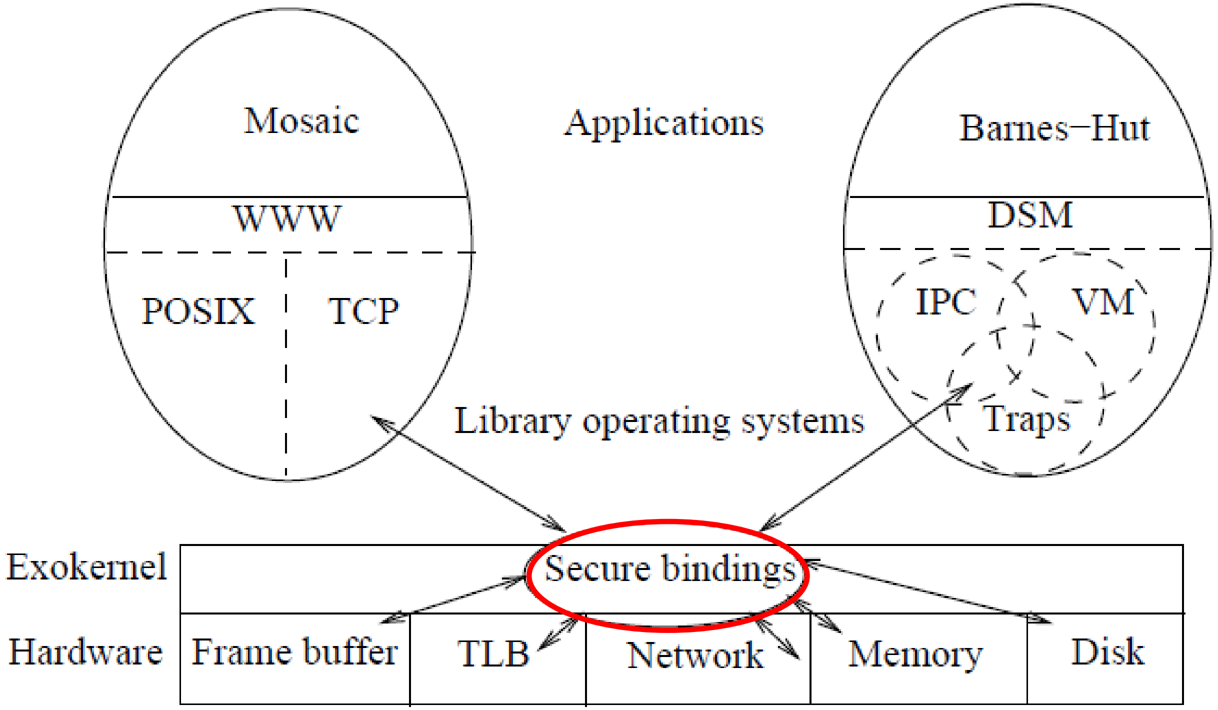
The abort protocol

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

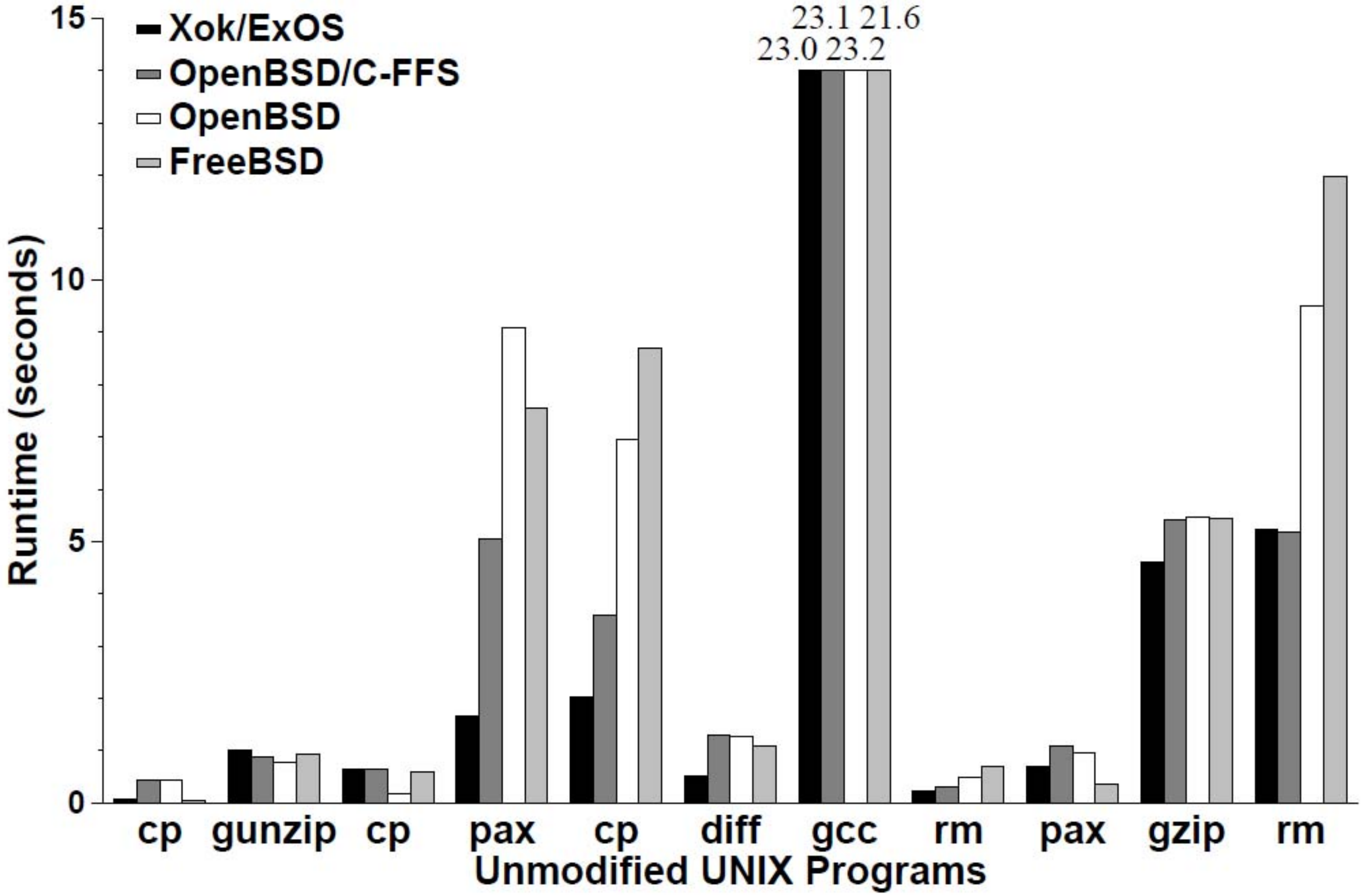
ExOS

Library OS

Manages fundamental OS abstractions at application level

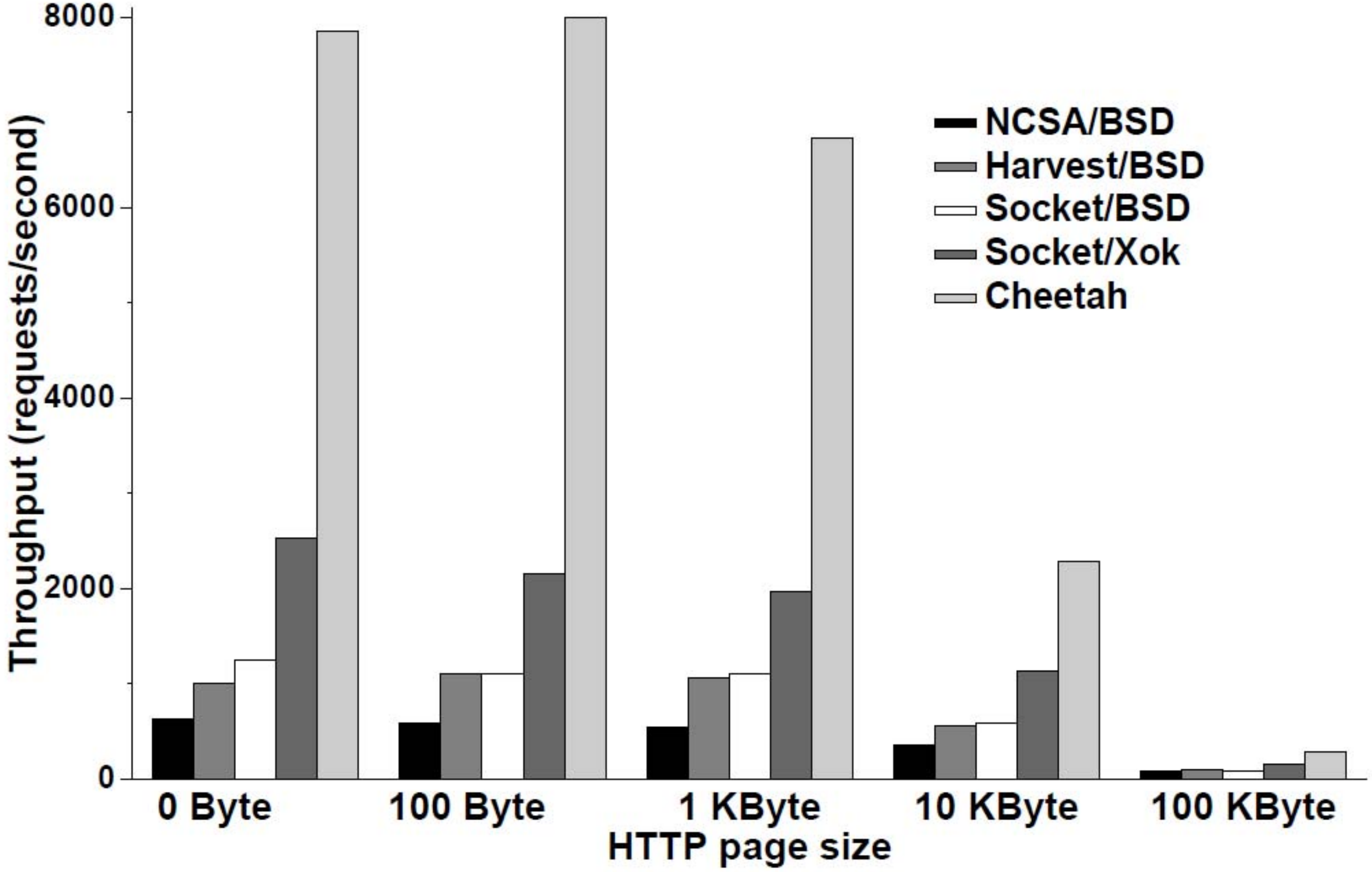
completely within the address space of the application that is using it

Evaluation



Application Performance and Flexibility on Exokernel Systems, SOSP'97

Evaluation



Application-level control can significantly improve the performance of applications

Questions

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’08

Homework

- Please read two papers with the same name
  - Are Virtual Machine Monitors Microkernels Done Right?
  - http://research.cs.wisc.edu/areas/os/ReadingGroup/OS/papers/hotos-ukernel.pdf
  - http://ssrg.nicta.com.au/publications/papers/Heiser_UL_06.pdf

- Please figure out what they were talking about, summarize their arguments, and give your own understanding

XV6 & VM

Rong Chen & Yubin Xia

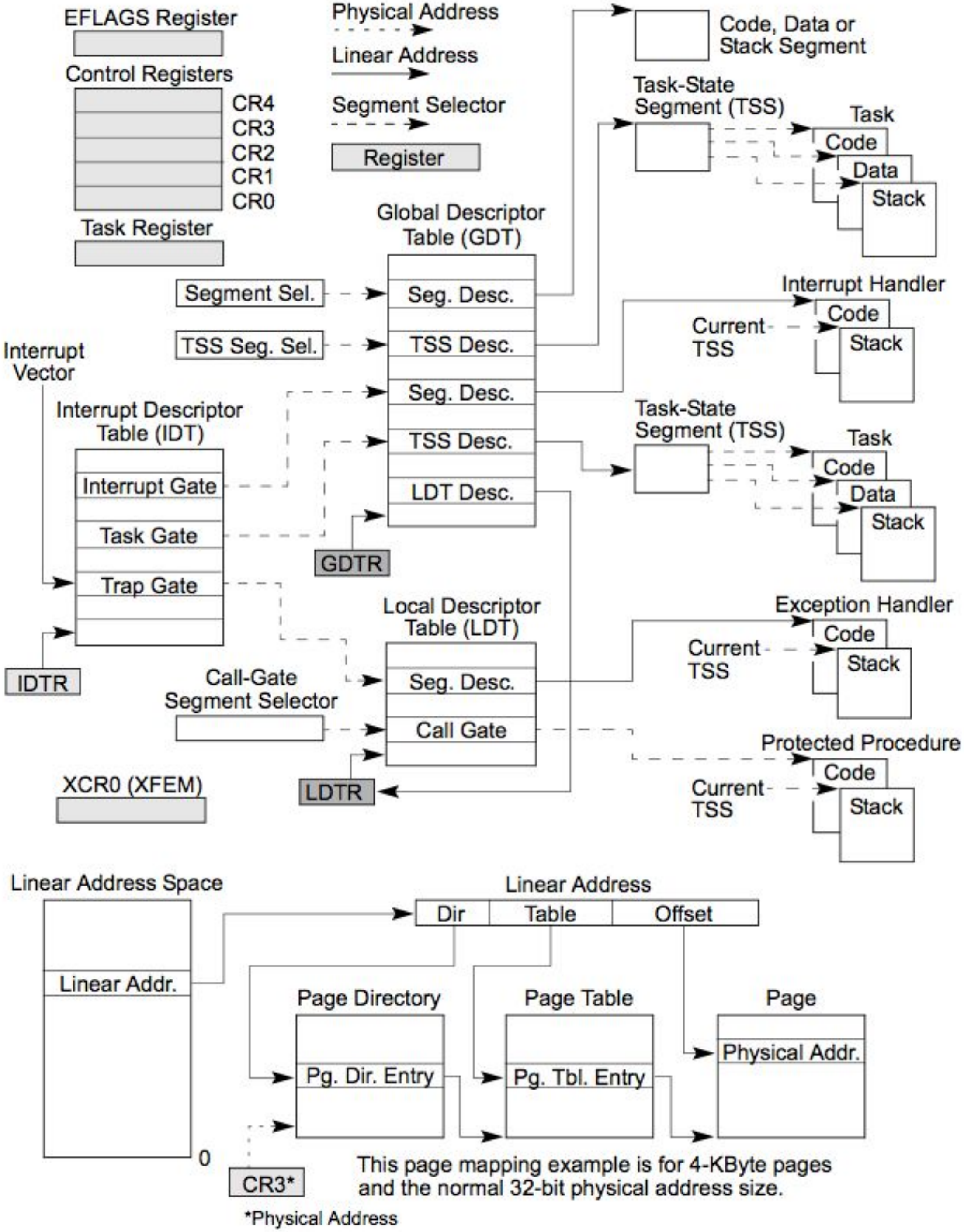
Some slides are based on Frans Kaashoek's slides
kaashoek@mit.edu

Outline

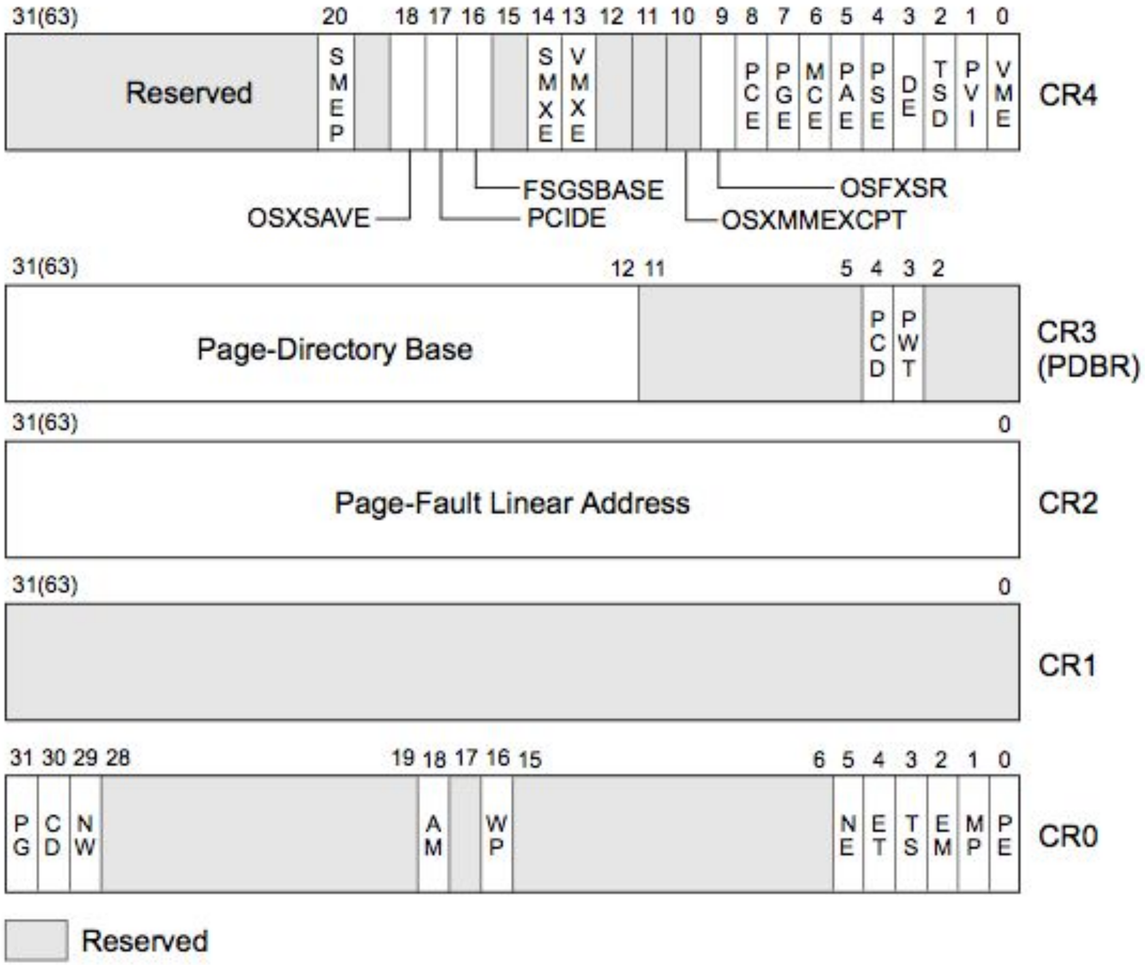
- Segment, Paging
- LA, VA, PA
- VM in xv6
  - Initial page table
  - Kernel page table
  - User page table
- Page Fault

VIRTUAL MEMORY

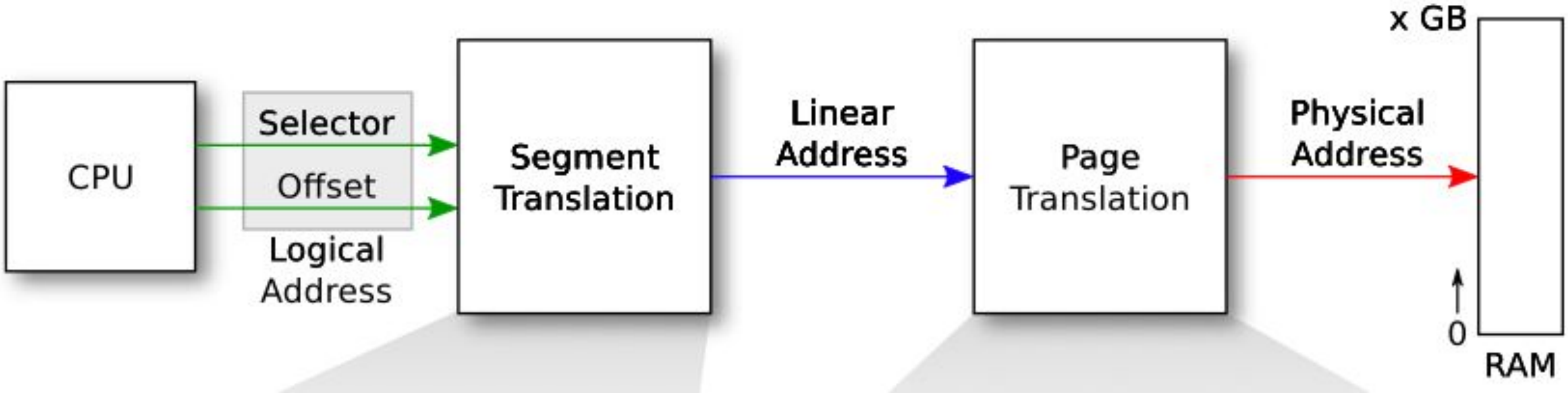
IA32 Arch Overview



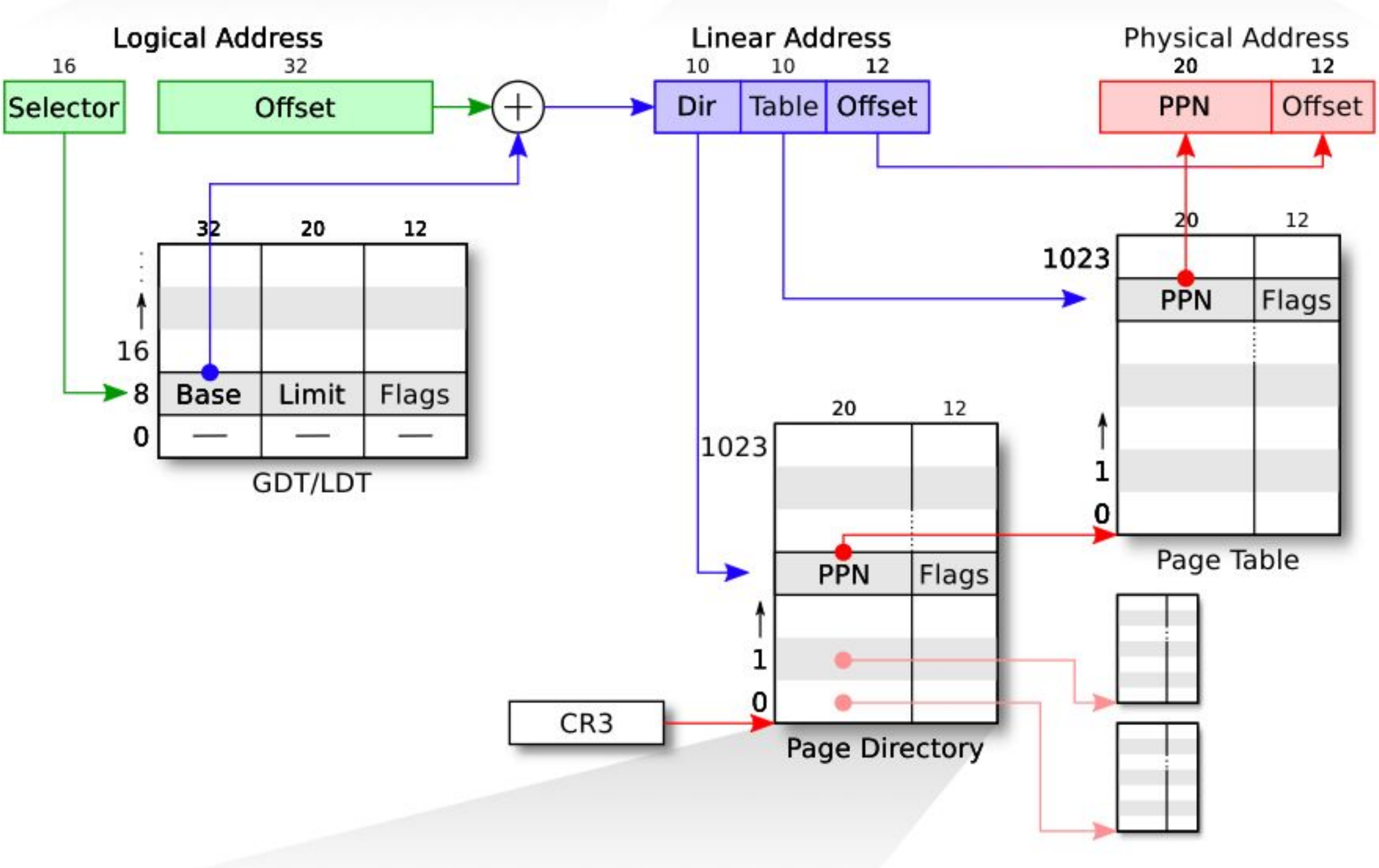
Control Registers



Protected-mode Address Translation



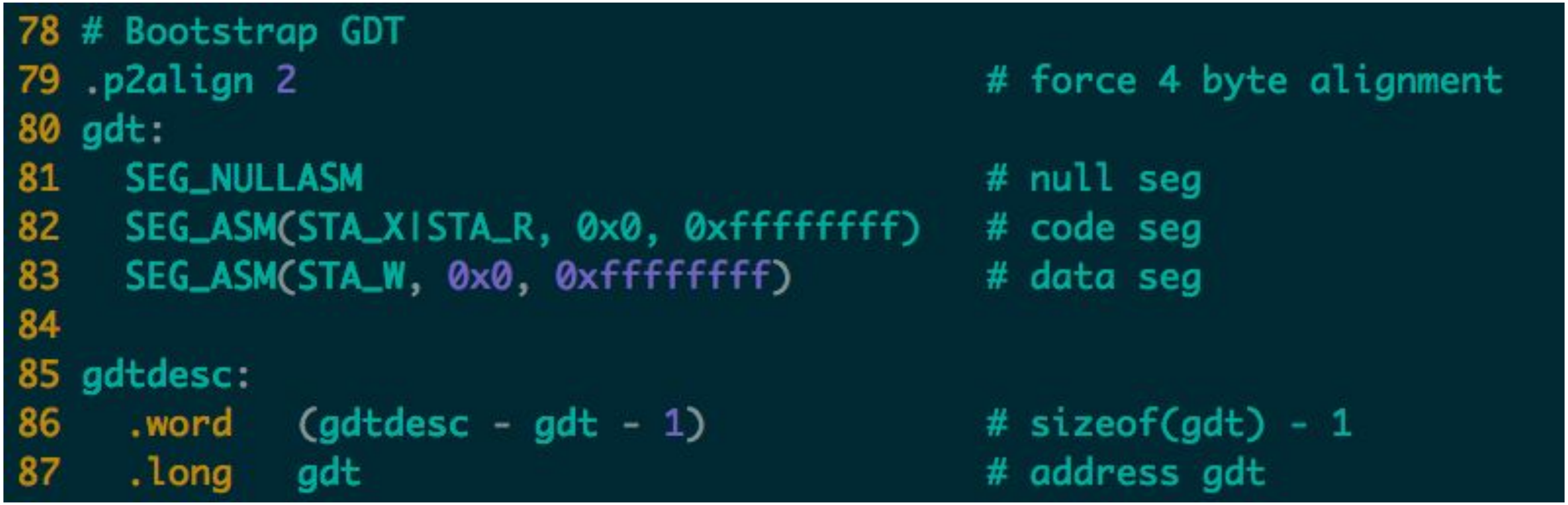
Protected Mode Address Translation



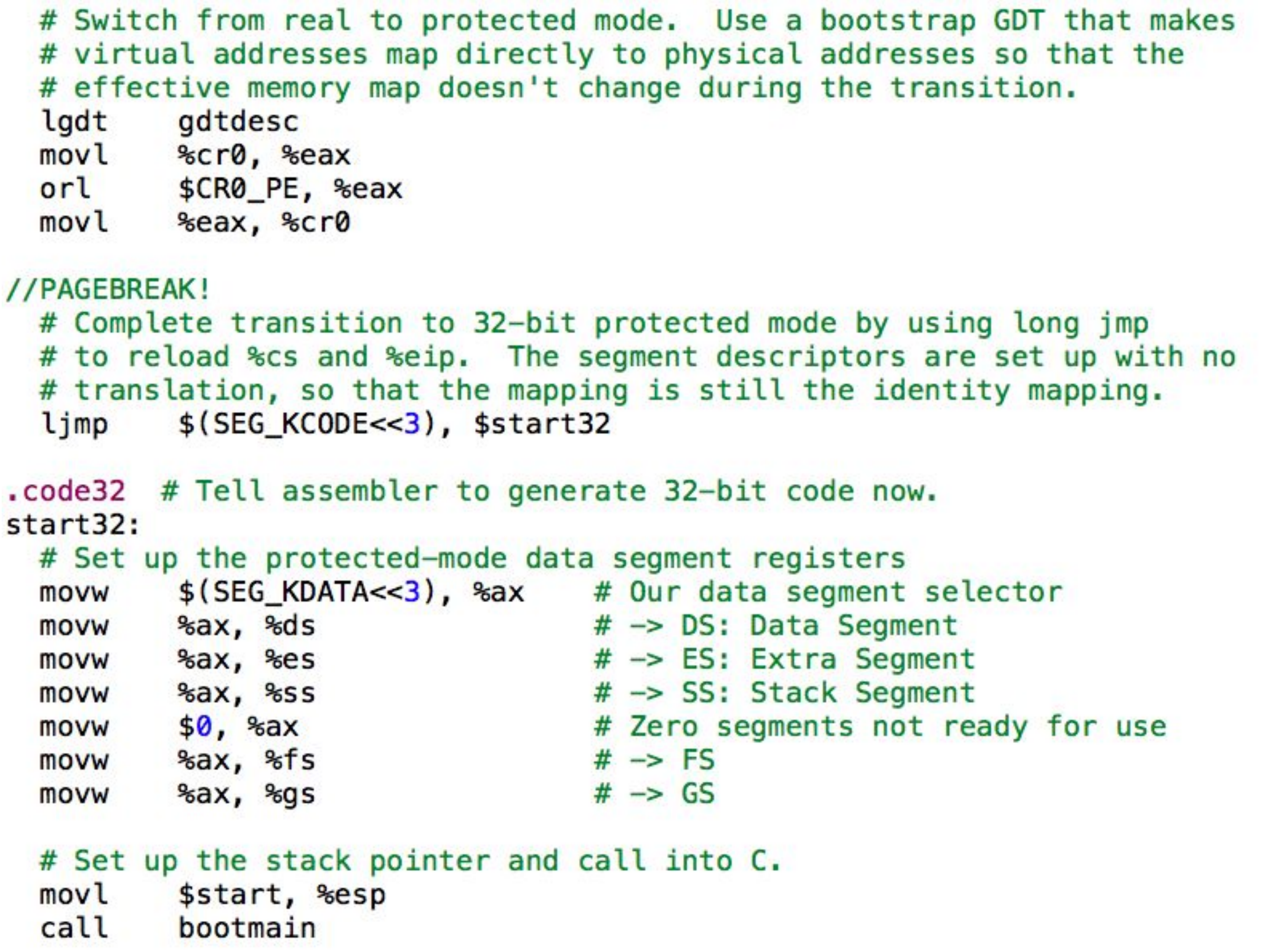
what's going to happen when the kernel enables segment?

- virtual address translation comes into effect?
  - presumably the enabling instruction has address segment-base + 0xffffc
  - The address may not contain any valid instruction!
  - Solution
    - Ensure that the addresses are the same before and after

Enable Segment



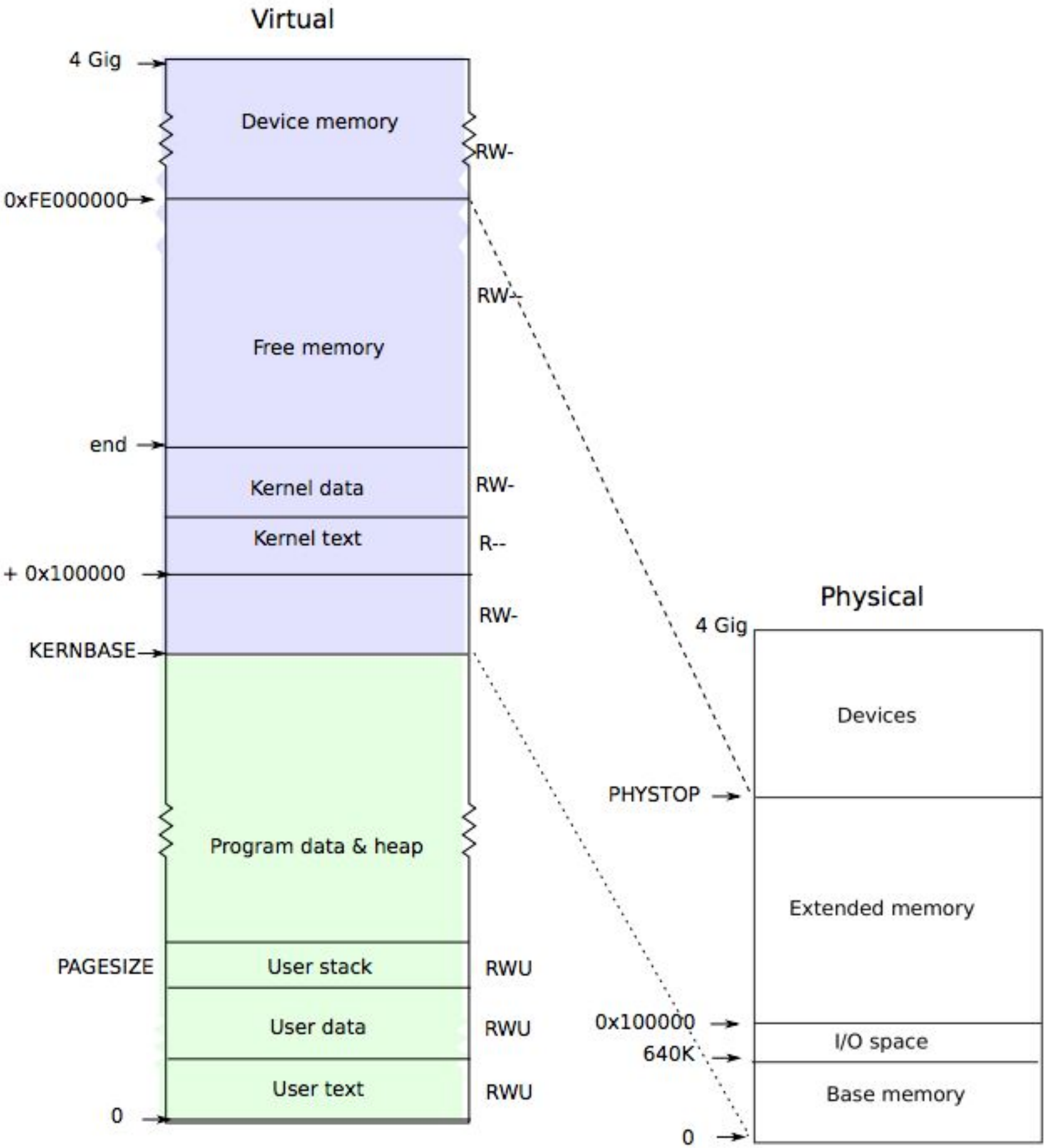
```
# Bootstrap GDT
.p2align 2                                # force 4 byte alignment
gdt:
  SEG_NULLASM                             # null seg
  SEG_ASM(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG_ASM(STA_W, 0x0, 0xffffffff)         # data seg

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



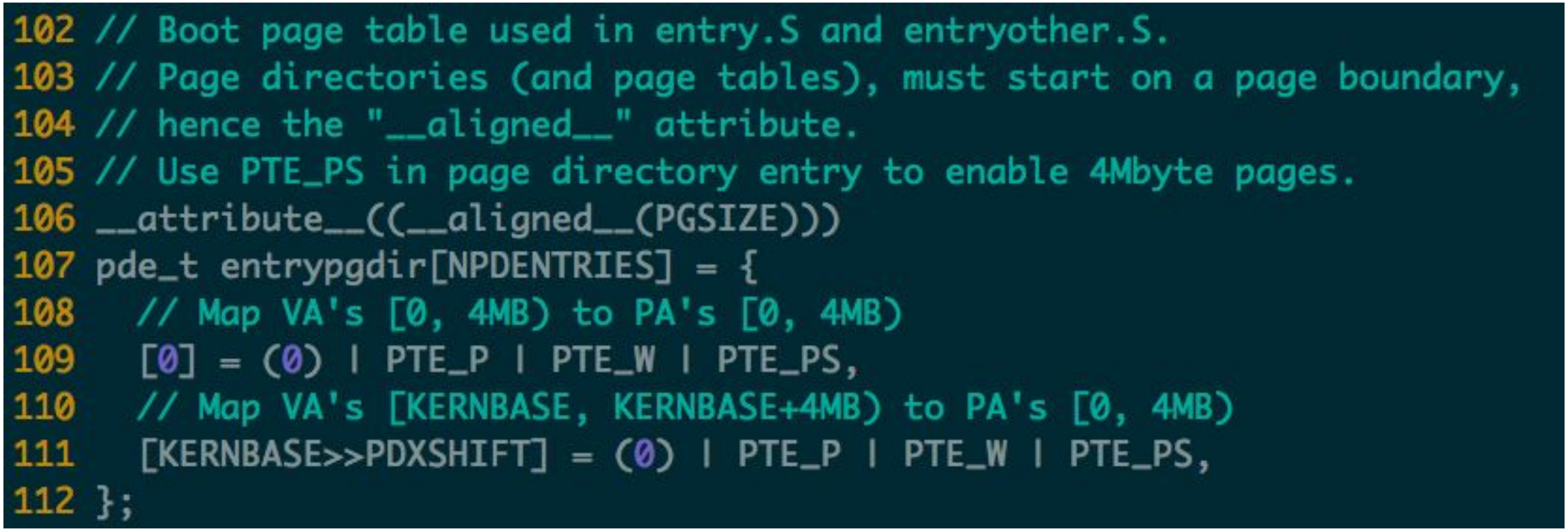
```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//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
```

what's going to happen when the kernel enables paging?

- virtual address translation comes into effect?
  - presumably the enabling instruction has address 0x100xxx
  - 0x100xxx maps to user memory, not kernel instructions!
  - solution
    - temporary page table with kernel mapped at both 0x80100000 and 0x100000
    - that's what the first few kernel instructions set up

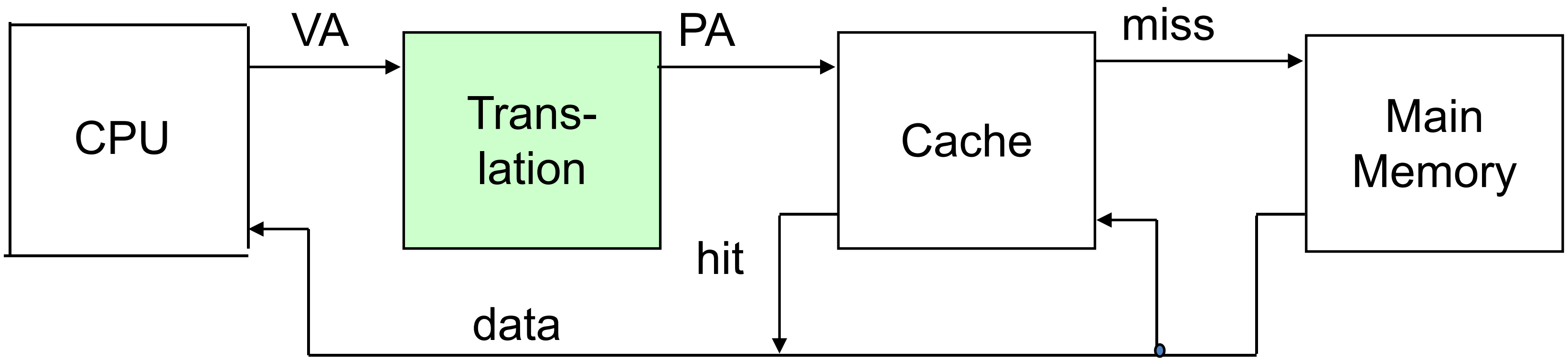

Virtual
4 Gig
Device memory
RW-
0xFE000000
RW-
Free memory
end
Kernel data
RW-
Kernel text
R--
+ 0x100000
RW-
KERNBASE
Program data & heap
PAGESIZE
User stack
RWU
User data
RWU
User text
RWU
0
Physical
4 Gig
Devices
PHYSTOP
Extended memory
0x100000
I/O space
640K
Base memory
0

entrypgdir

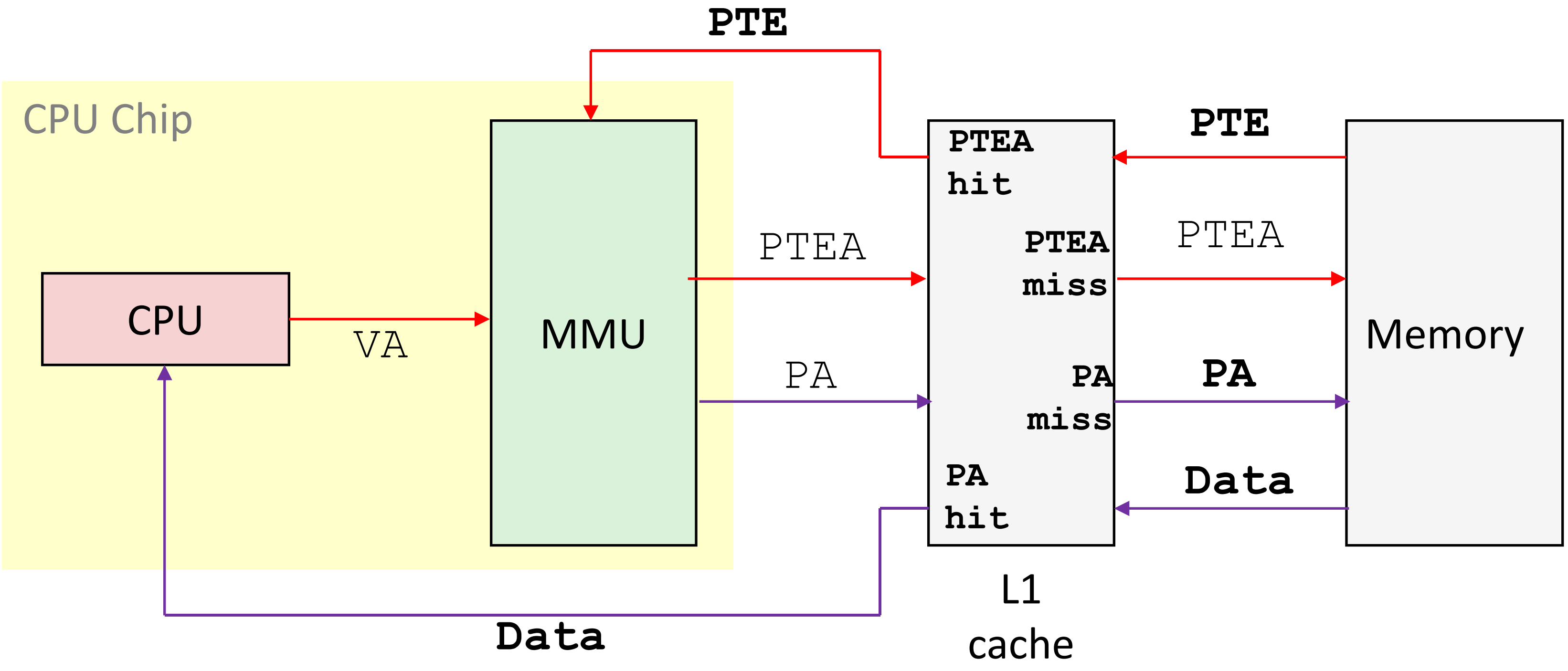


```
// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
// Use PTE_PS in page directory entry to enable 4Mbyte pages.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [0, 4MB) to PA's [0, 4MB)
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

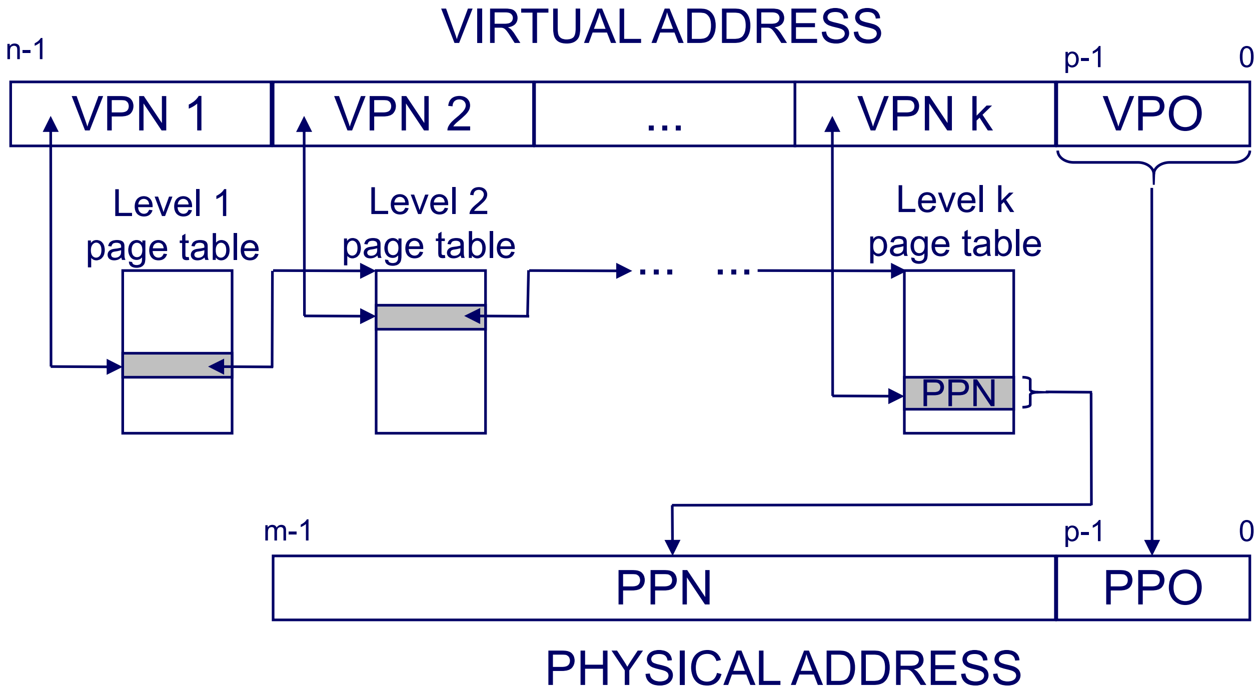
Recap: Integrating Caches and VM



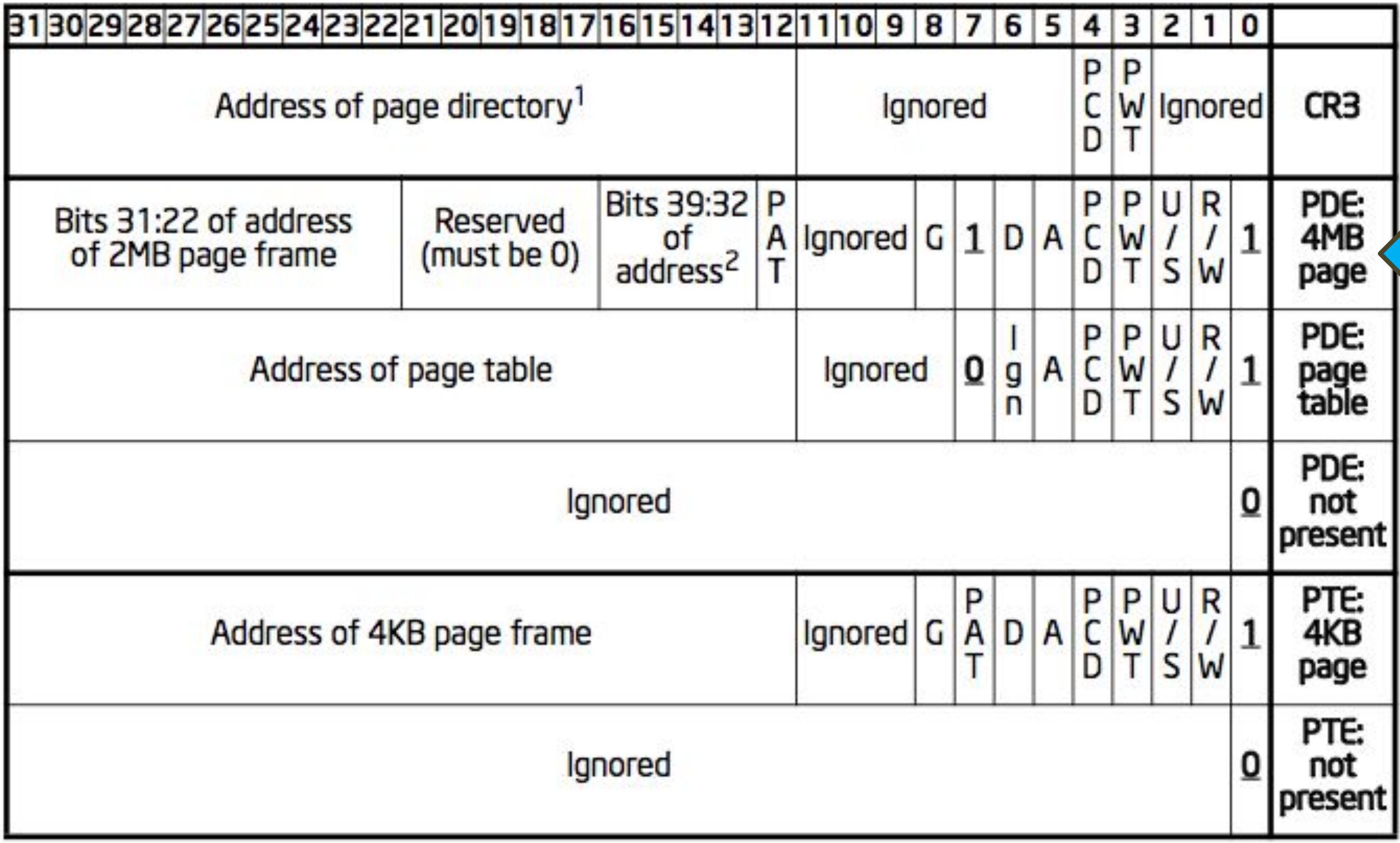
Recap: Integrating Caches and VM



Translating with a k-level Page Table

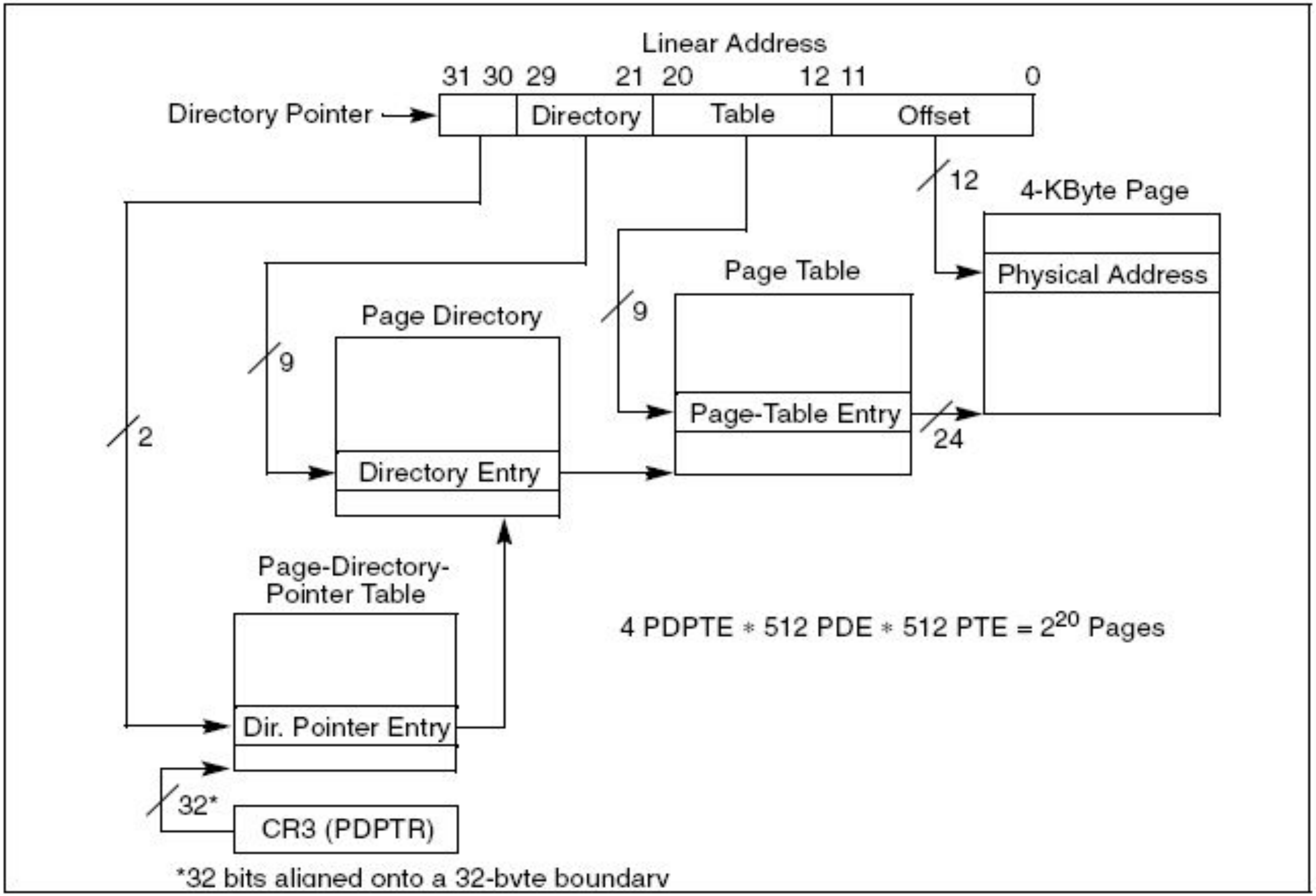


Super Page: 4M Page



<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td><td></td></tr>
<tr><td colspan="20">Address of page directory[1]</td><td colspan="7">Ignored</td><td>PCD</td><td>PWT</td><td colspan="3">Ignored</td><td>CR3</td></tr>
<tr><td colspan="10">Bits 31:22 of address of 2MB page frame</td><td colspan="5">Reserved (must be 0)</td><td colspan="4">Bits 39:32 of address[2]</td><td>PAT</td><td colspan="3">Ignored</td><td>G</td><td>1</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: 4MB page</td></tr>
<tr><td colspan="20">Address of page table</td><td colspan="4">Ignored</td><td>0</td><td>Ign</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PDE: page table</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PDE: not present</td></tr>
<tr><td colspan="20">Address of 4KB page frame</td><td colspan="3">Ignored</td><td>G</td><td>PAT</td><td>D</td><td>A</td><td>PCD</td><td>PWT</td><td>U/S</td><td>R/W</td><td>1</td><td>PTE: 4KB page</td></tr>
<tr><td colspan="31">Ignored</td><td>0</td><td>PTE: not present</td></tr>
</table>

Address translation: PAE-4k



4 PDPTE * 512 PDE * 512 PTE = 2^{20} Pages

*32 bits aligned onto a 32-byte boundary

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

Address translation: PAE-2M

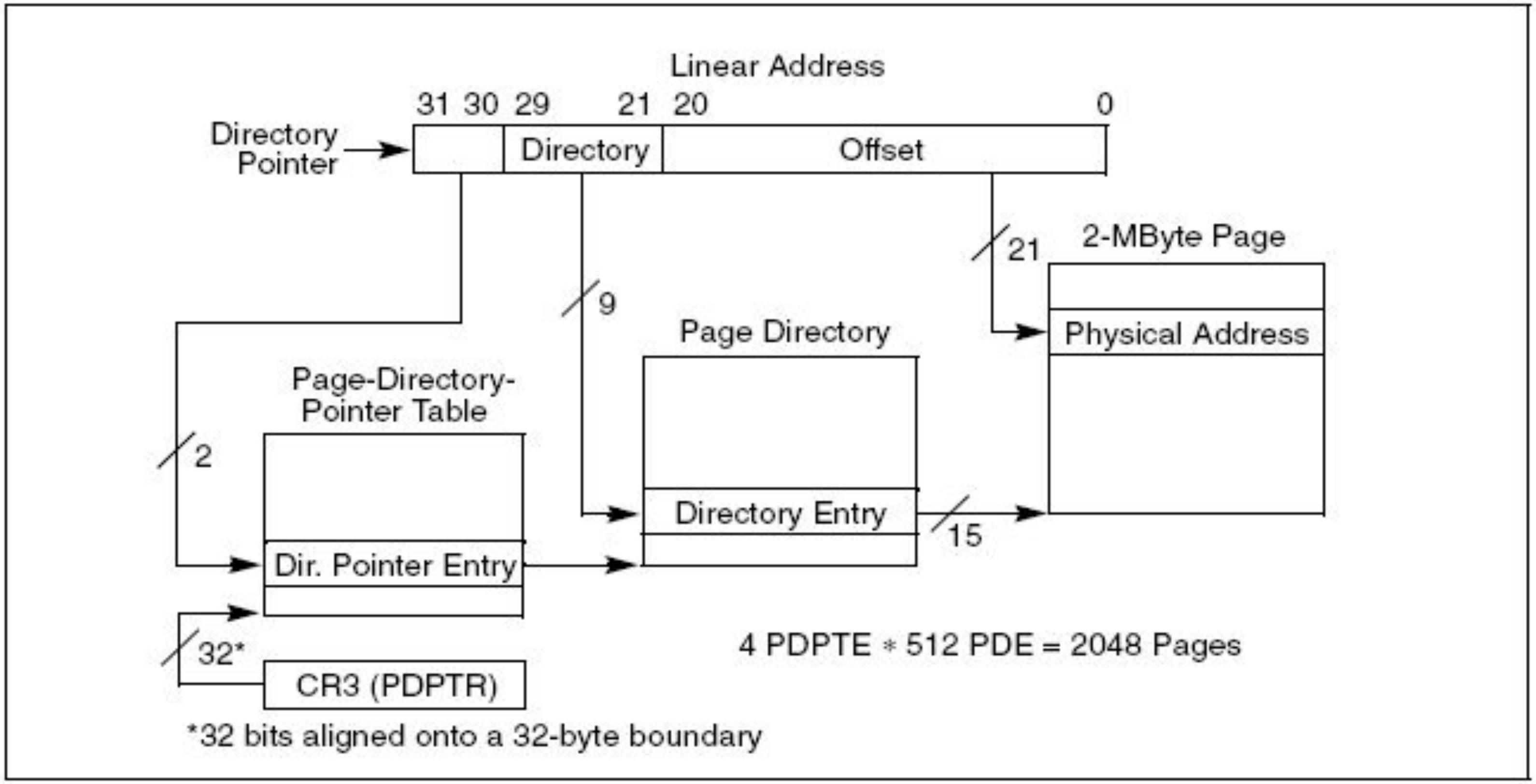


Figure 3-19. Linear Address Translation With PAE Enabled (2-MByte Pages)

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:

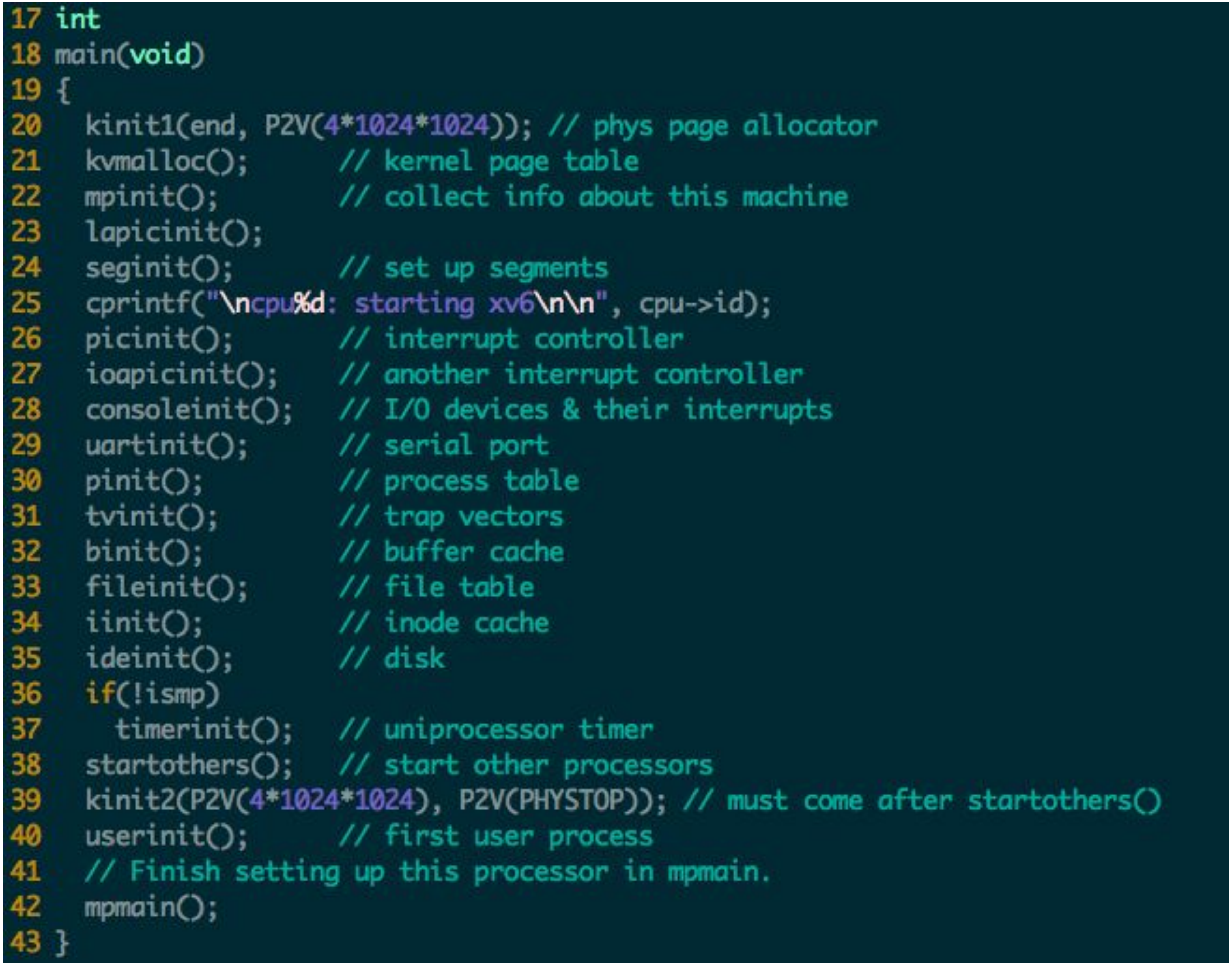
a) Dump the content of a large file of several hundred megabytes mapped to the RAM by mmap() function to another location:

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

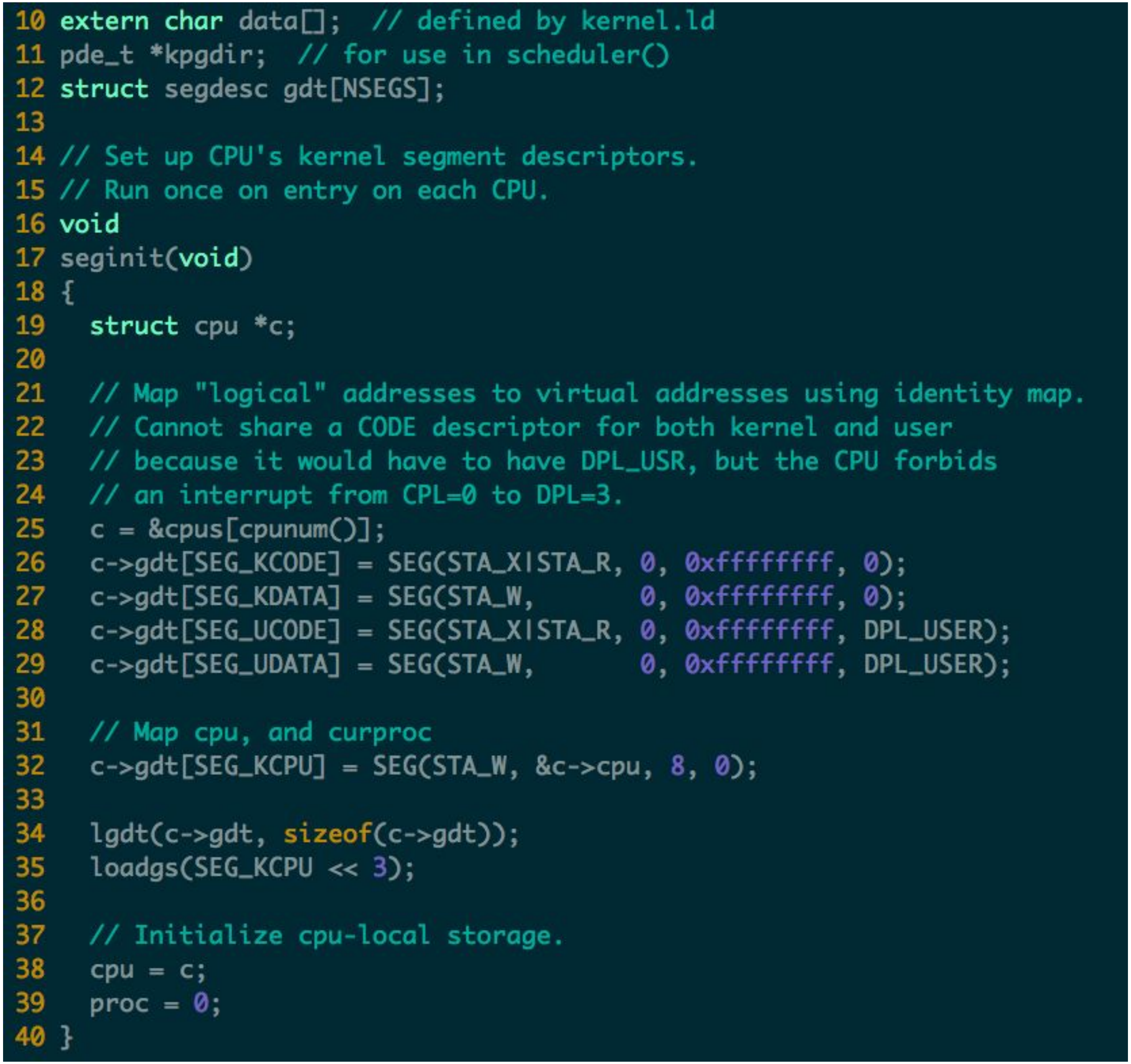
b) Do the following calculation, where A, B are int pointers:

```
for(i=0; i<0x40000; ++i){
    A[i] += B[i] & 0x3FFFFF;
}
```

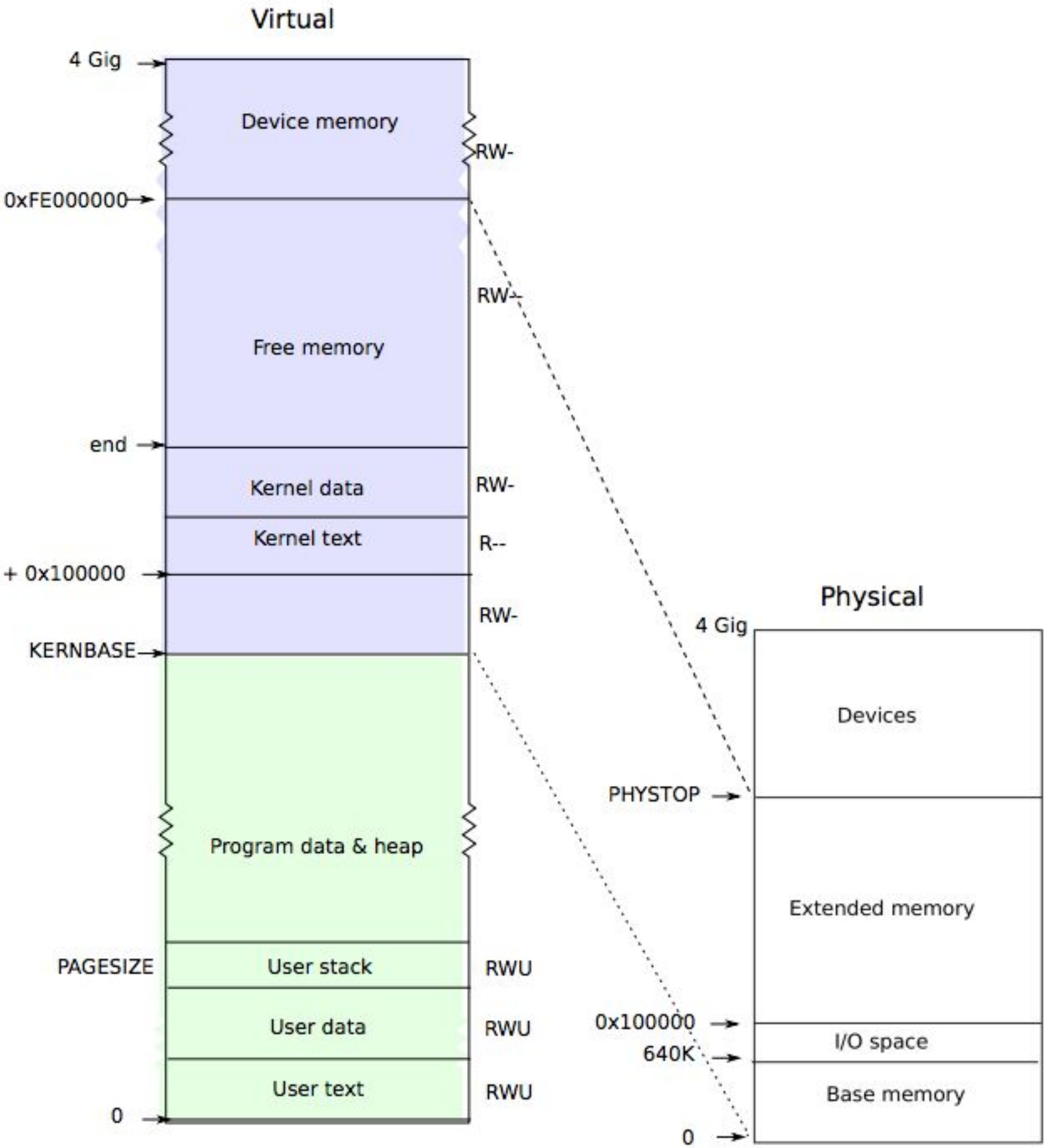
Which sized (4KB or 4MB) page performs better in each scenario? Give your reasons.



```c
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  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();
}
```



```
extern char data[];  // defined by kernel.ld
pde_t *kpgdir;  // for use in scheduler()
struct segdesc gdt[NSEGS];

// Set up CPU's kernel segment descriptors.
// Run once on entry on each CPU.
void
seginit(void)
{
  struct cpu *c;

  // Map "logical" addresses to virtual addresses using identity map.
  // Cannot share a CODE descriptor for both kernel and user
  // because it would have to have DPL_USR, but the CPU forbids
  // an interrupt from CPL=0 to DPL=3.
  c = &cpus[cpunum()];
  c->gdt[SEG_KCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, 0);
  c->gdt[SEG_KDATA] = SEG(STA_W,       0, 0xffffffff, 0);
  c->gdt[SEG_UCODE] = SEG(STA_X|STA_R, 0, 0xffffffff, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(STA_W,       0, 0xffffffff, DPL_USER);

  // Map cpu, and curproc
  c->gdt[SEG_KCPU] = SEG(STA_W, &c->cpu, 8, 0);

  lgdt(c->gdt, sizeof(c->gdt));
  loadgs(SEG_KCPU << 3);

  // Initialize cpu-local storage.
  cpu = c;
  proc = 0;
}
```


Virtual
4 Gig
Device memory
RW-
0xFE000000
RW-
Free memory
end
Kernel data
RW-
Kernel text
R--
+ 0x100000
RW-
KERNBASE
Program data & heap
PAGESIZE
User stack
RWU
User data
RWU
User text
RWU
0
Physical
4 Gig
Devices
PHYSTOP
Extended memory
0x100000
I/O space
640K
Base memory
0

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

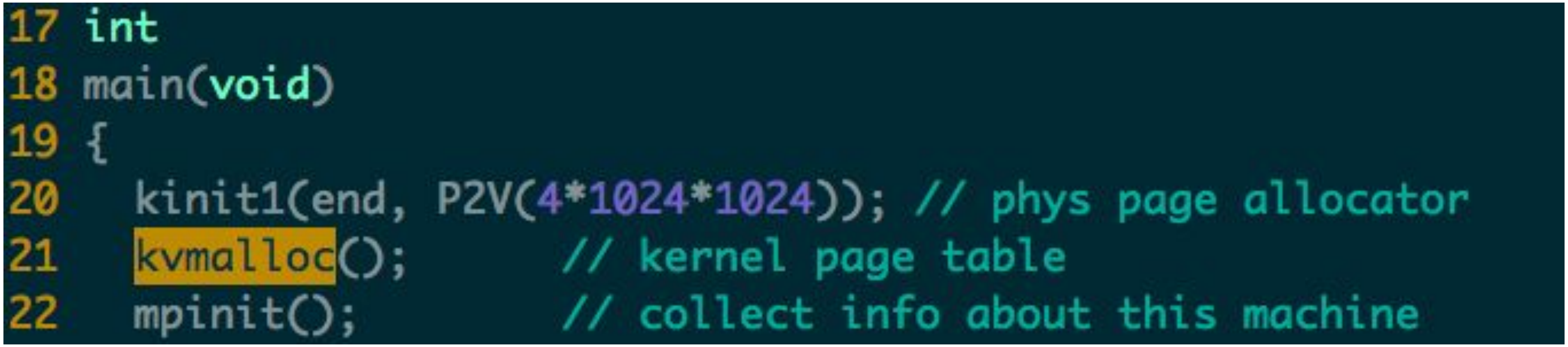
- more memory-mapped devices

4 KB pages

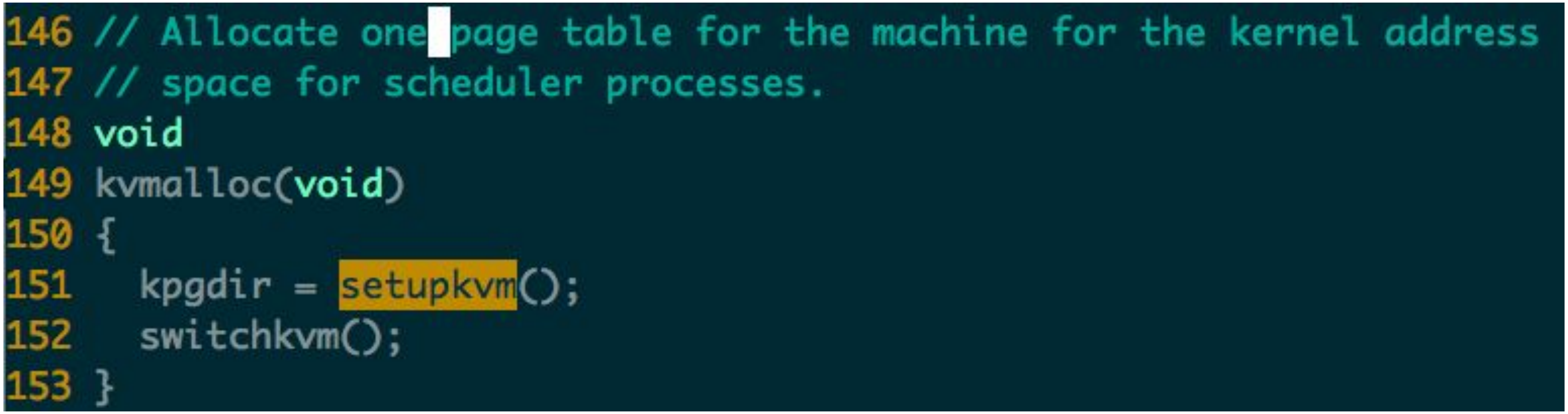
entrypgdir was simple

- 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 phys mem
- solution: x86 MMU supports 4 KB pages

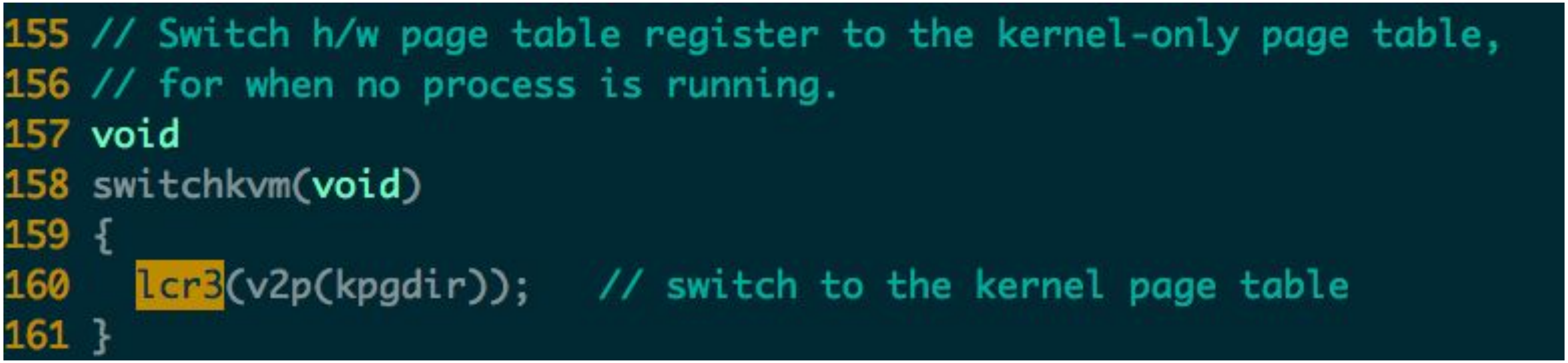
kvmalloc



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



```
// Allocate one page table for the machine for the kernel address
// space for scheduler processes.
void
kvmalloc(void)
{
  kpgdir = setupkvm();
  switchkvm();
}
```



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

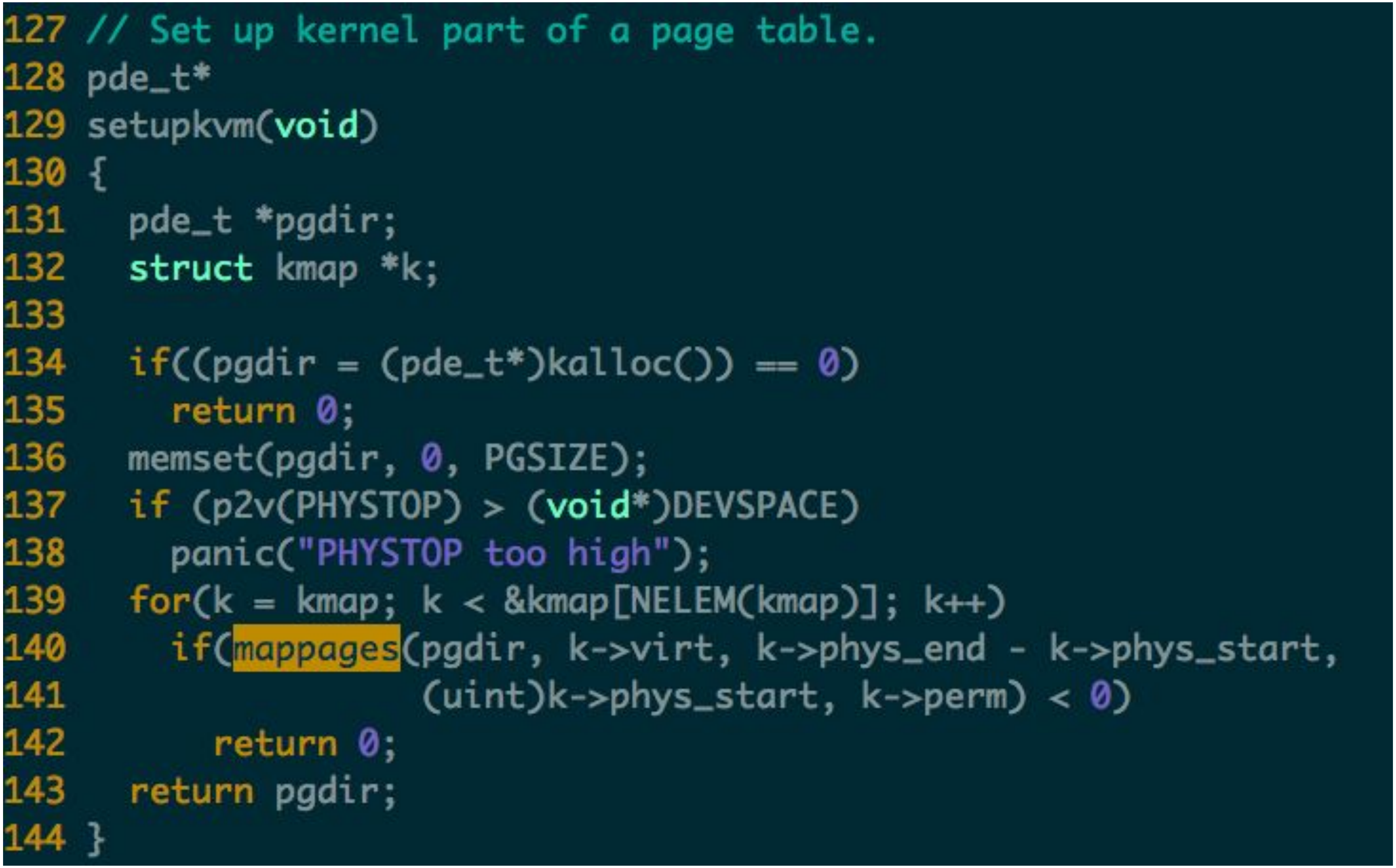
Setup_kvm

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[]



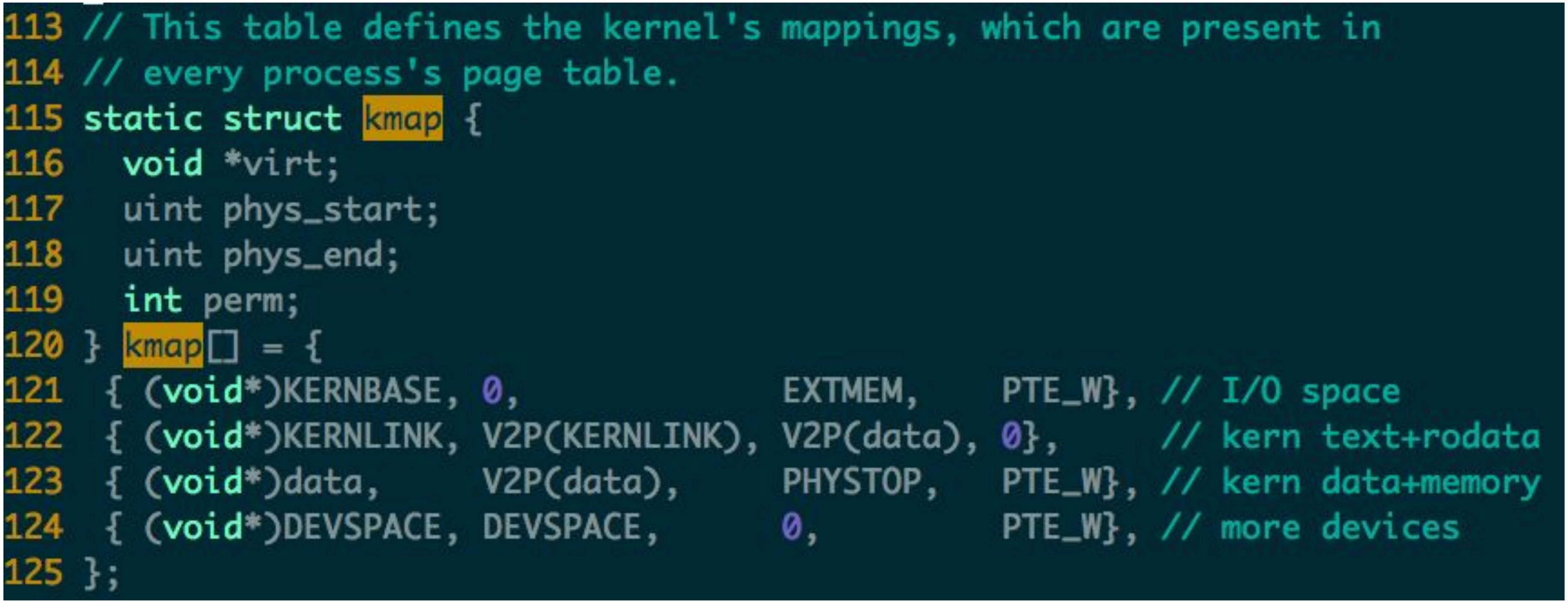
```c
// Set up kernel part of a page table.
pde_t*
setupkvm(void)
{
  pde_t *pgdir;
  struct kmap *k;

  if((pgdir = (pde_t*)kalloc()) == 0)
    return 0;
  memset(pgdir, 0, PGSIZE);
  if (p2v(PHYSTOP) > (void*)DEVSPACE)
    panic("PHYSTOP too high");
  for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
    if(mappages(pgdir, k->virt, k->phys_end - k->phys_start,
                (uint)k->phys_start, k->perm) < 0)
      return 0;
  return pgdir;
}
```

what is kmap[]?

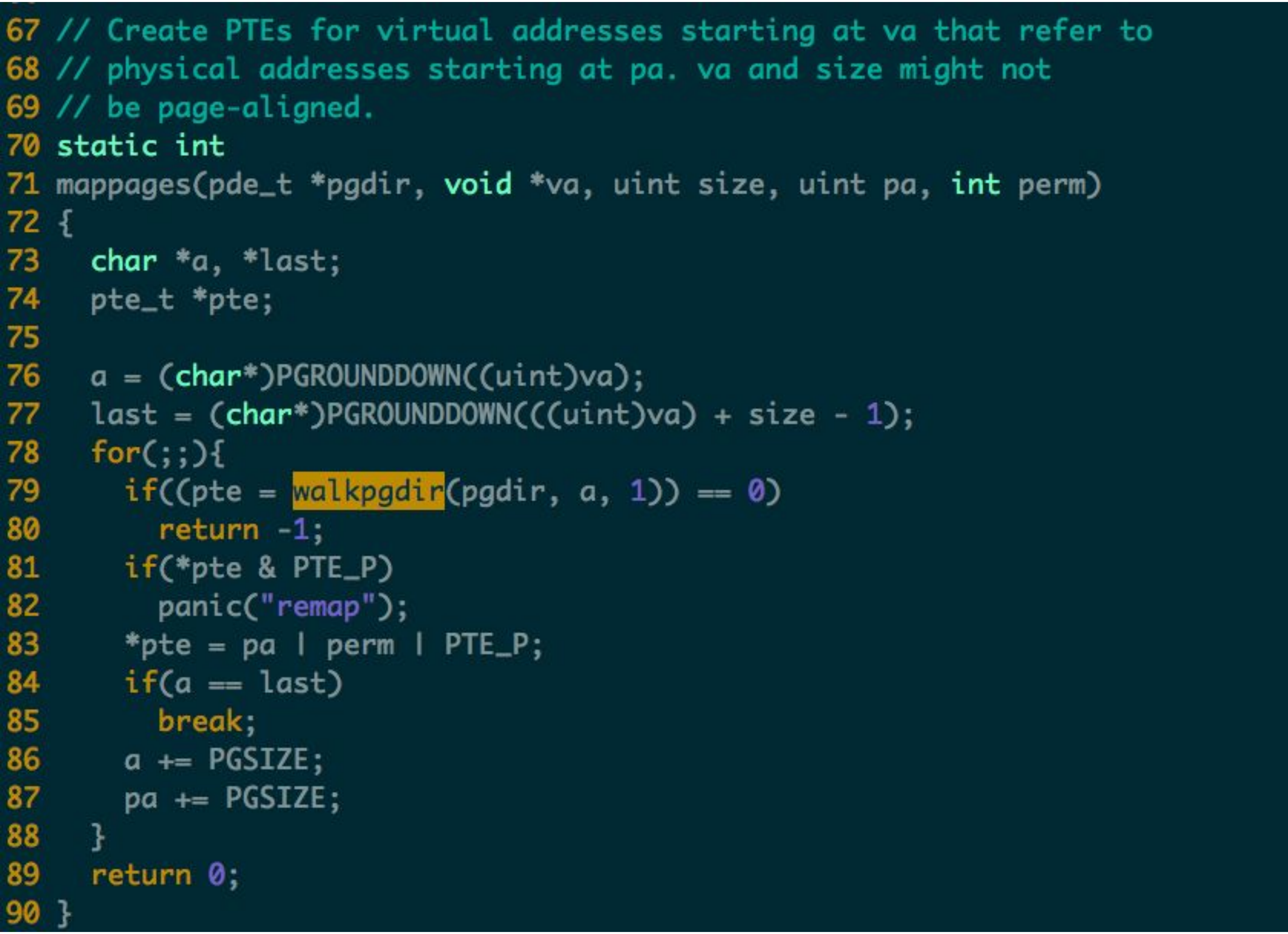
an entry for each region of kernel virtual address space

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



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



```
// Create PTEs for virtual addresses starting at va that refer to
// physical addresses starting at pa. va and size might not
// be page-aligned.
static int
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
  char *a, *last;
  pte_t *pte;

  a = (char*)PGROUNDDOWN((uint)va);
  last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
  for(;;){
    if((pte = walkpgdir(pgdir, a, 1)) == 0)
      return -1;
    if(*pte & PTE_P)
      panic("remap");
    *pte = pa | perm | PTE_P;
    if(a == last)
      break;
    a += PGSIZE;
    pa += PGSIZE;
  }
  return 0;
}
```

mappages

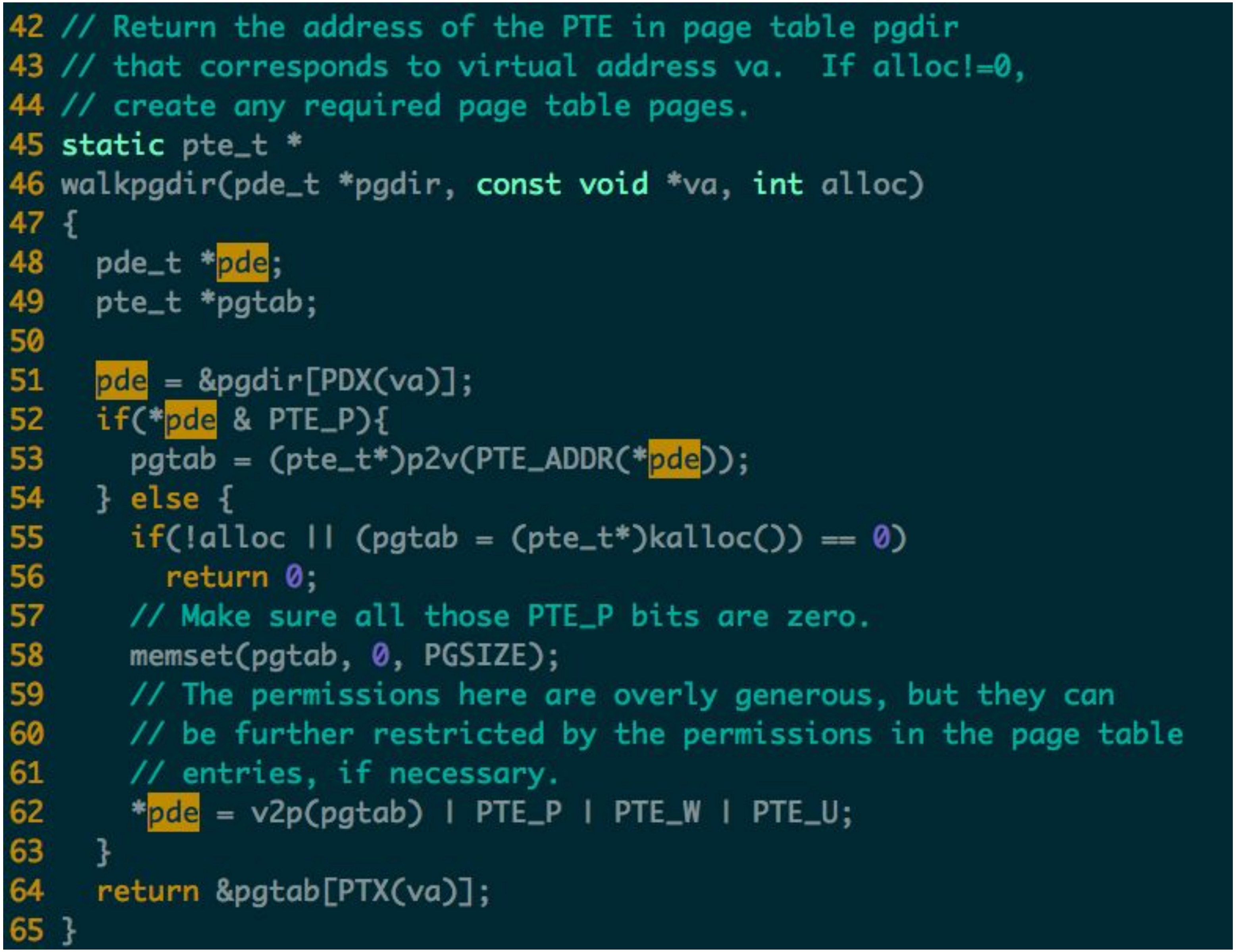
arguments are PD, va, size, pa, perm

adds mappings from a range of va's to corresponding pa's

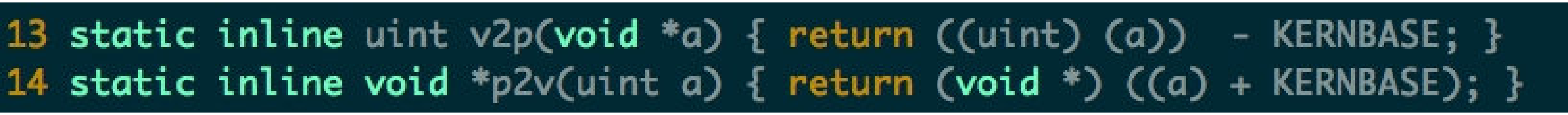
rounds b/c other uses pass in non-page-aligned addresses

for each page-aligned address in the range

call walkpgdir to find address of PTE need the PTE's address (not just content) b/c we want to modify



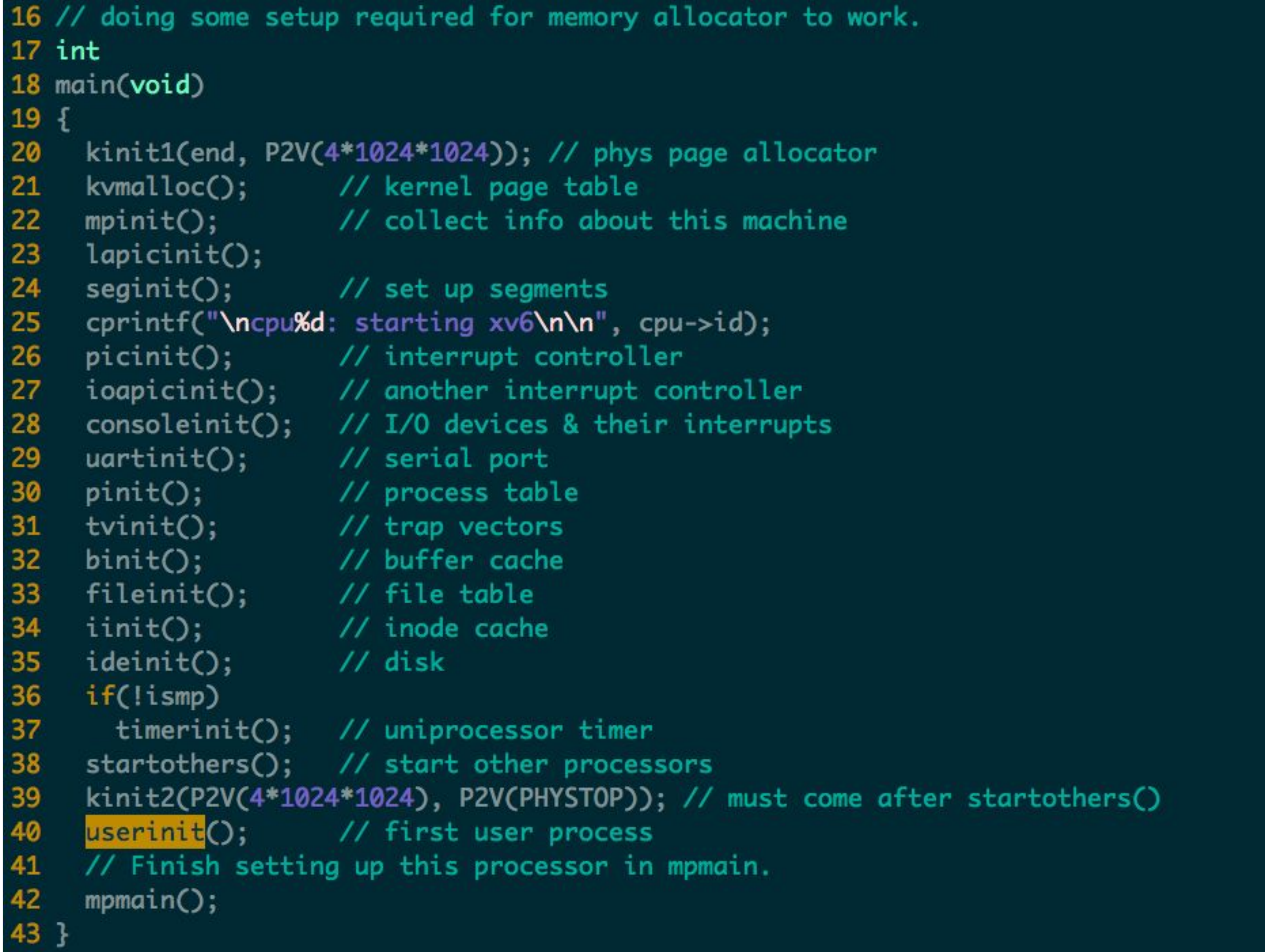
```c
// Return the address of the PTE in page table pgdir
// that corresponds to virtual address va.  If alloc!=0,
// create any required page table pages.
static pte_t *
walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```



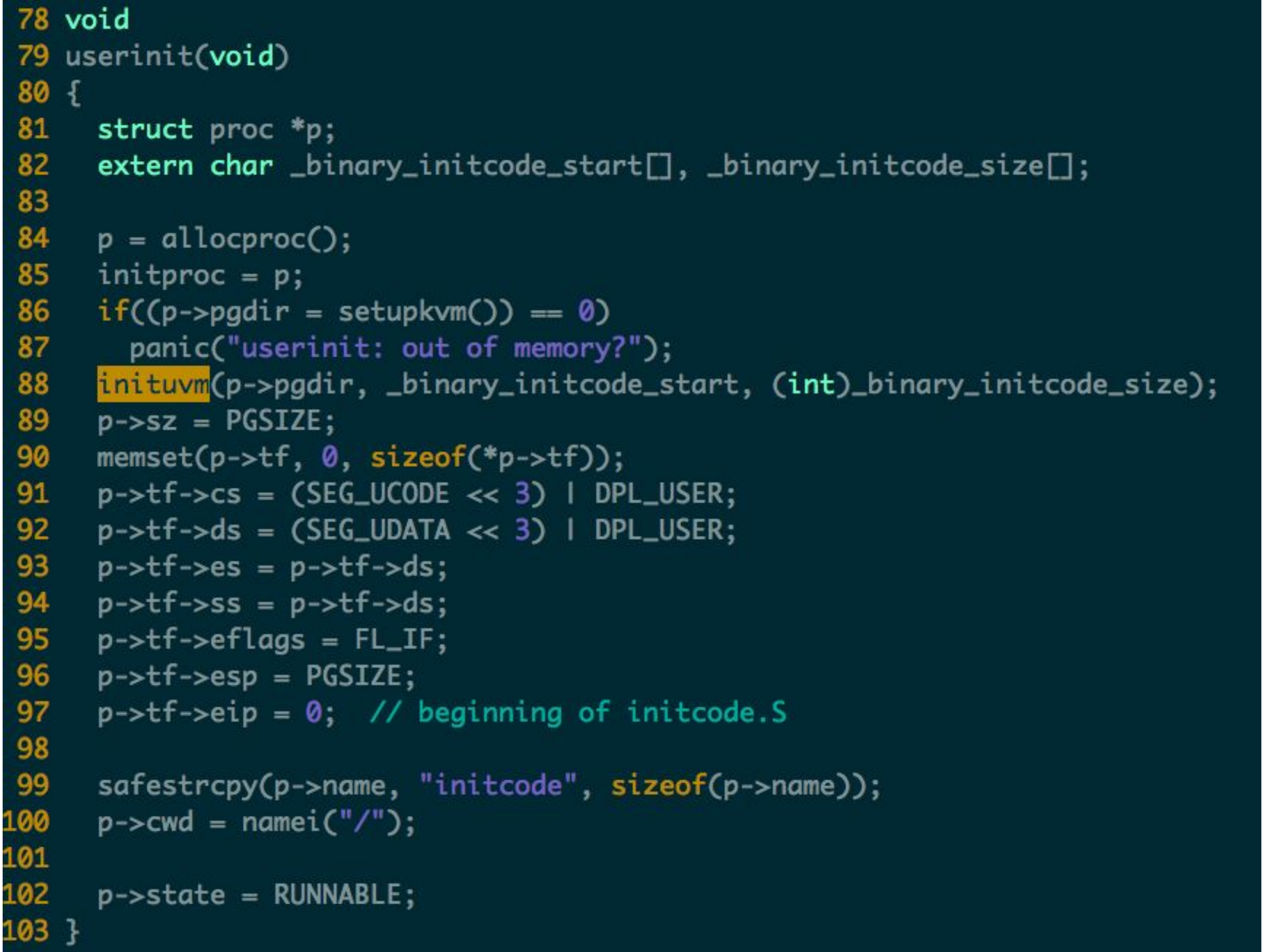
```c
static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }
```

walkpgdir

- mimics how the paging h/w finds the PTE for an address
- a little complex b/c 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



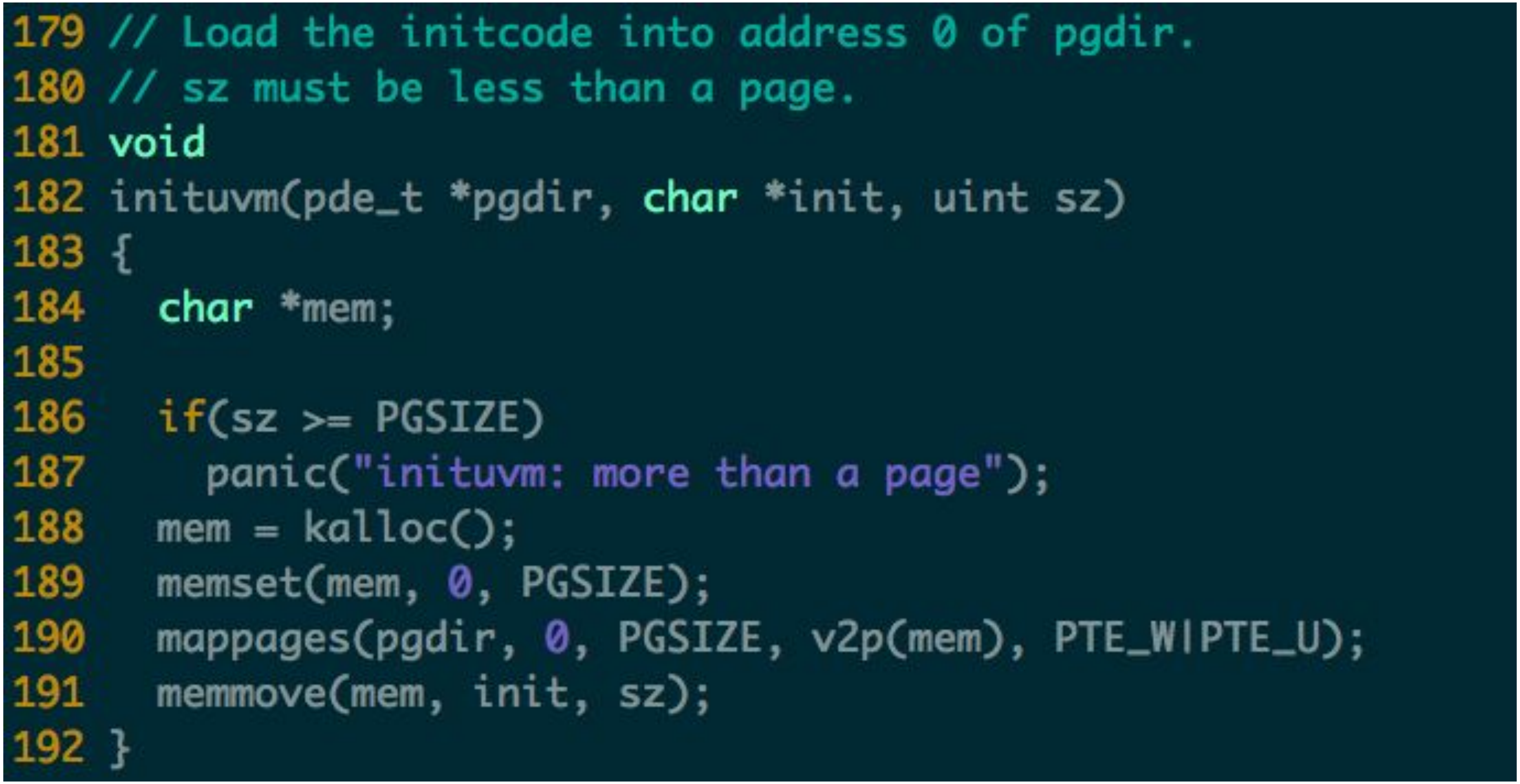
```c
// doing some setup required for memory allocator to work.
int
main(void)
{
  kinit1(end, P2V(4*1024*1024)); // phys page allocator
  kvmalloc();      // kernel page table
  mpinit();        // collect info about this machine
  lapicinit();
  seginit();       // set up segments
  cprintf("\ncpu%d: 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();
}
```



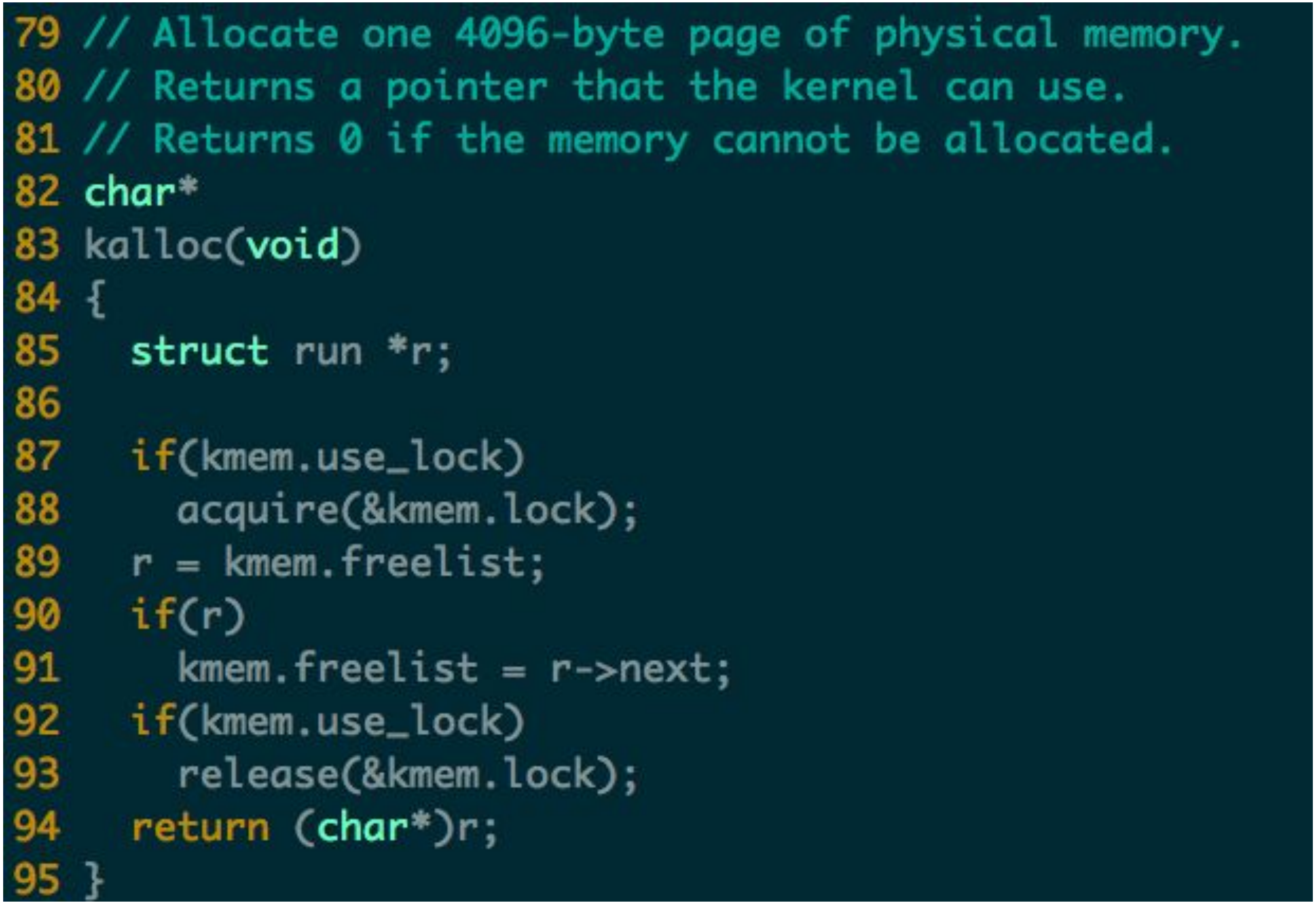
```
void
userinit(void)
{
  struct proc *p;
  extern char _binary_initcode_start[], _binary_initcode_size[];

  p = allocproc();
  initproc = p;
  if((p->pgdir = setupkvm()) == 0)
    panic("userinit: out of memory?");
  inituvm(p->pgdir, _binary_initcode_start, (int)_binary_initcode_size);
  p->sz = PGSIZE;
  memset(p->tf, 0, sizeof(*p->tf));
  p->tf->cs = (SEG_UCODE << 3) | DPL_USER;
  p->tf->ds = (SEG_UDATA << 3) | DPL_USER;
  p->tf->es = p->tf->ds;
  p->tf->ss = p->tf->ds;
  p->tf->eflags = FL_IF;
  p->tf->esp = PGSIZE;
  p->tf->eip = 0;  // beginning of initcode.S

  safestrcpy(p->name, "initcode", sizeof(p->name));
  p->cwd = namei("/");

  p->state = RUNNABLE;
}
```

User VM mapping

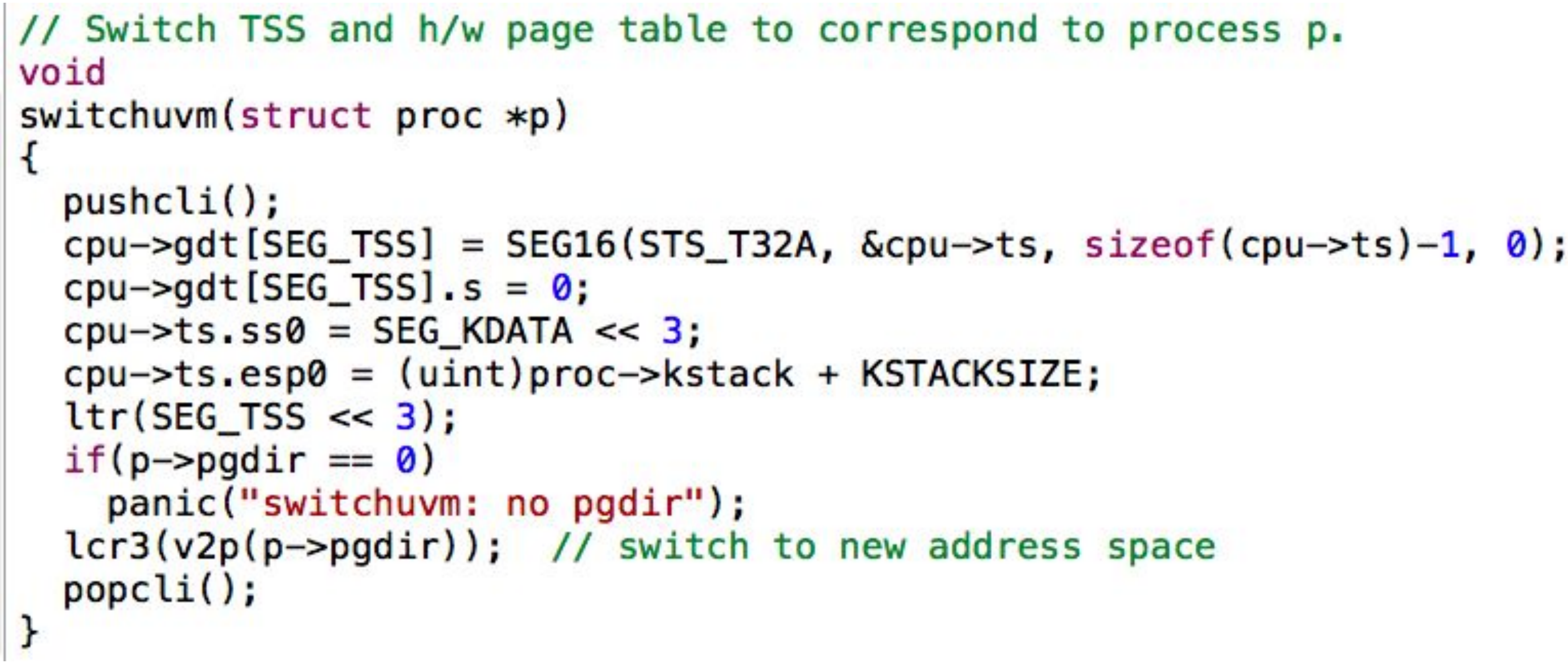
- Create a process
  - Inituvm : initialized user VM
  - Loaduvm: load program segment
- Context switch
  - switchuvm
- Grow/shrink/free VM
  - allocuvm
  - deallocuvm
  - Freeuvm
- Fork
  - copyuvm



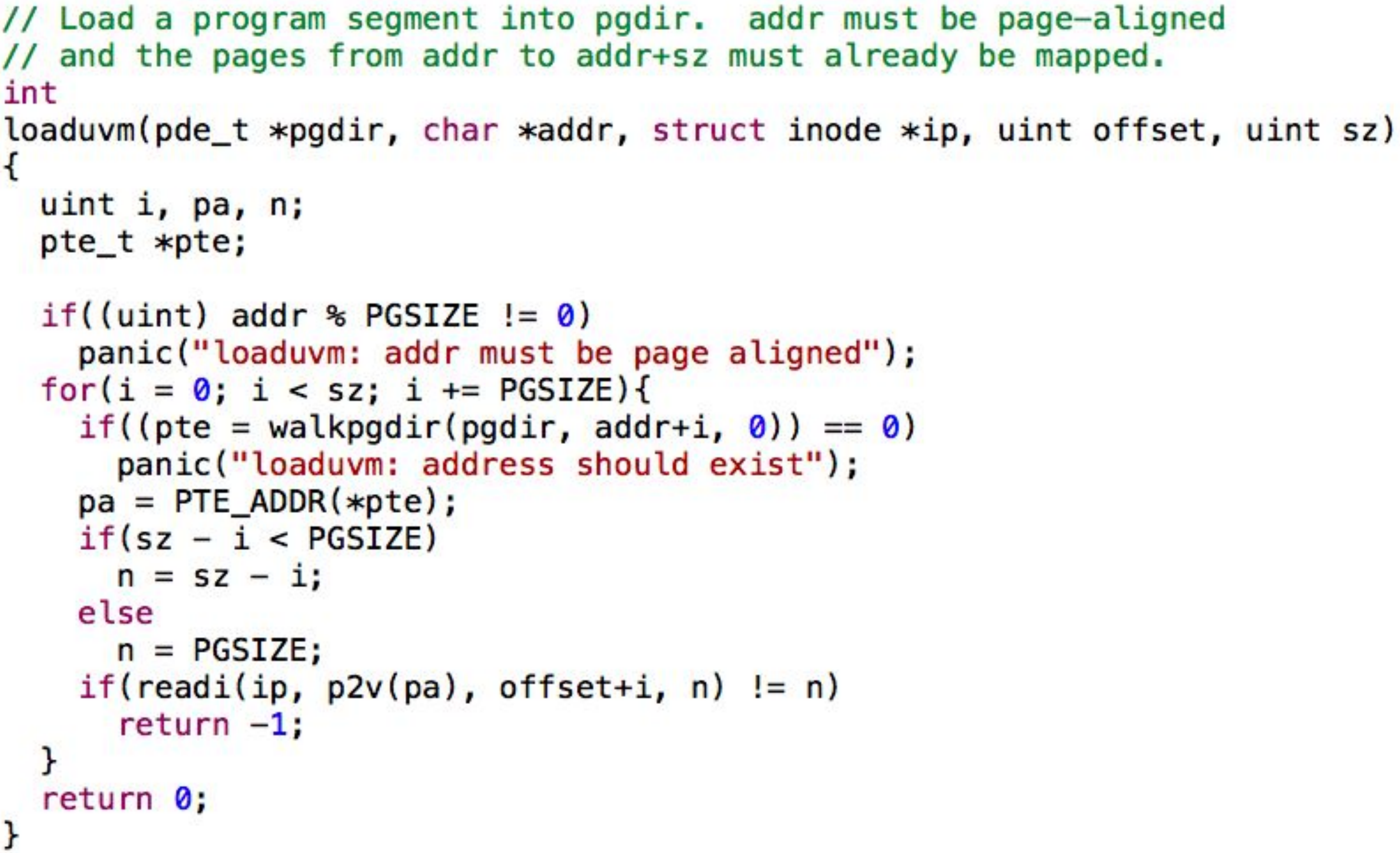
```c
// Load the initcode into address 0 of pgdir.
// sz must be less than a page.
void
inituvm(pde_t *pgdir, char *init, uint sz)
{
  char *mem;

  if(sz >= PGSIZE)
    panic("inituvm: more than a page");
  mem = kalloc();
  memset(mem, 0, PGSIZE);
  mappages(pgdir, 0, PGSIZE, v2p(mem), PTE_W|PTE_U);
  memmove(mem, init, sz);
}
```



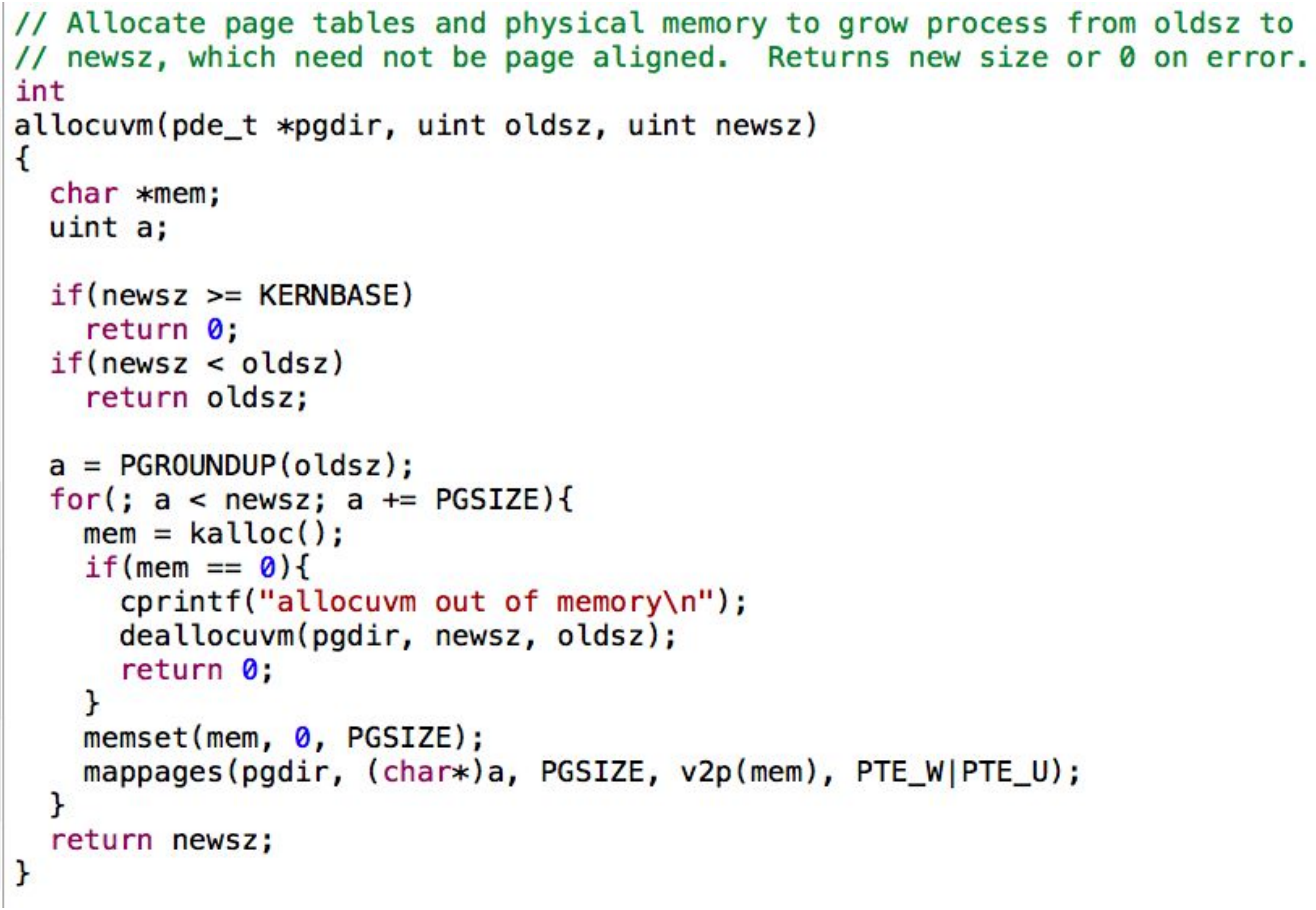
```c
// Allocate one 4096-byte page of physical memory.
// Returns a pointer that the kernel can use.
// Returns 0 if the memory cannot be allocated.
char*
kalloc(void)
{
  struct run *r;

  if(kmem.use_lock)
    acquire(&kmem.lock);
  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  if(kmem.use_lock)
    release(&kmem.lock);
  return (char*)r;
}
```



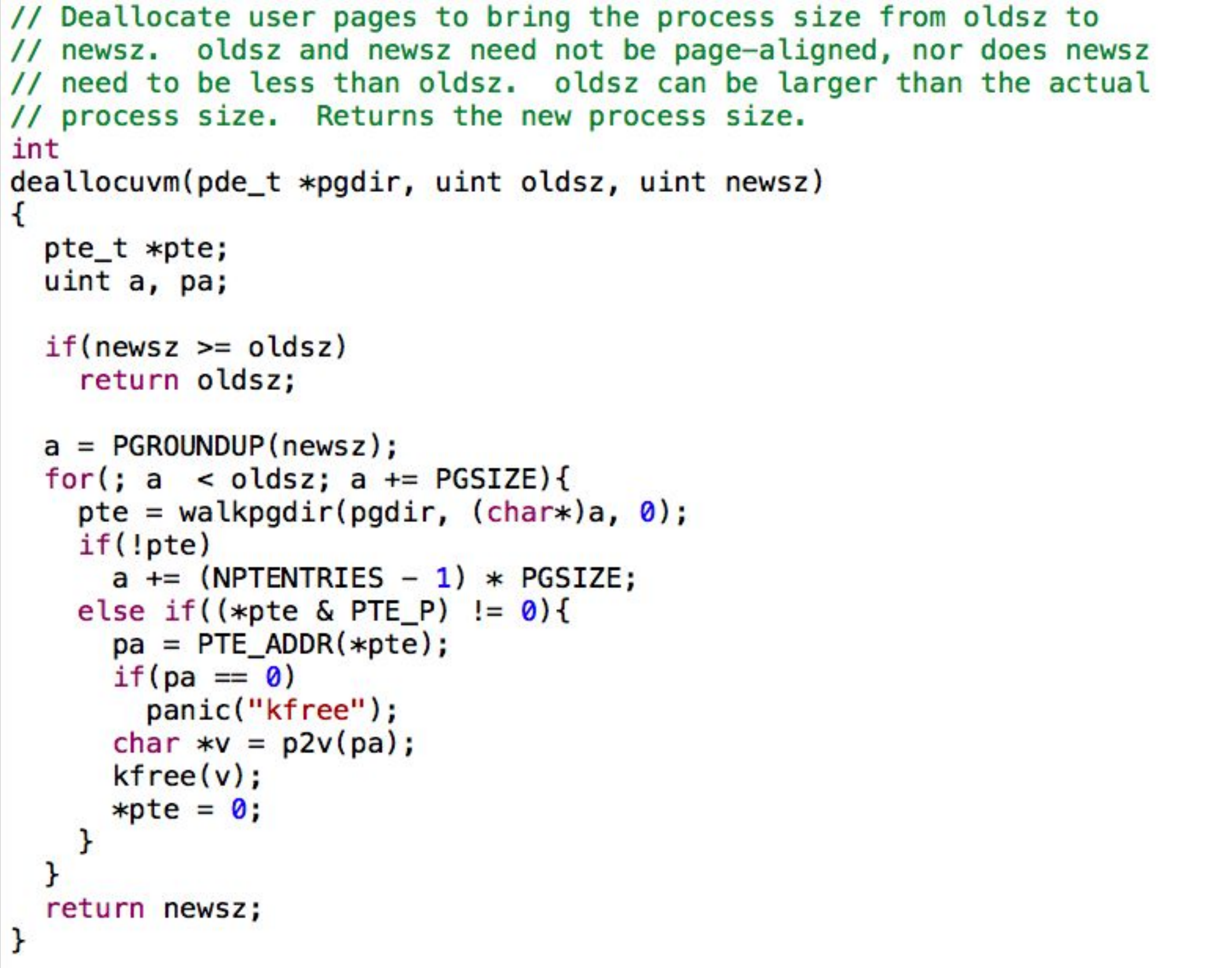
```
// Switch TSS and h/w page table to correspond to process p.
void
switchuvm(struct proc *p)
{
  pushcli();
  cpu->gdt[SEG_TSS] = SEG16(STS_T32A, &cpu->ts, sizeof(cpu->ts)-1, 0);
  cpu->gdt[SEG_TSS].s = 0;
  cpu->ts.ss0 = SEG_KDATA << 3;
  cpu->ts.esp0 = (uint)proc->kstack + KSTACKSIZE;
  ltr(SEG_TSS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



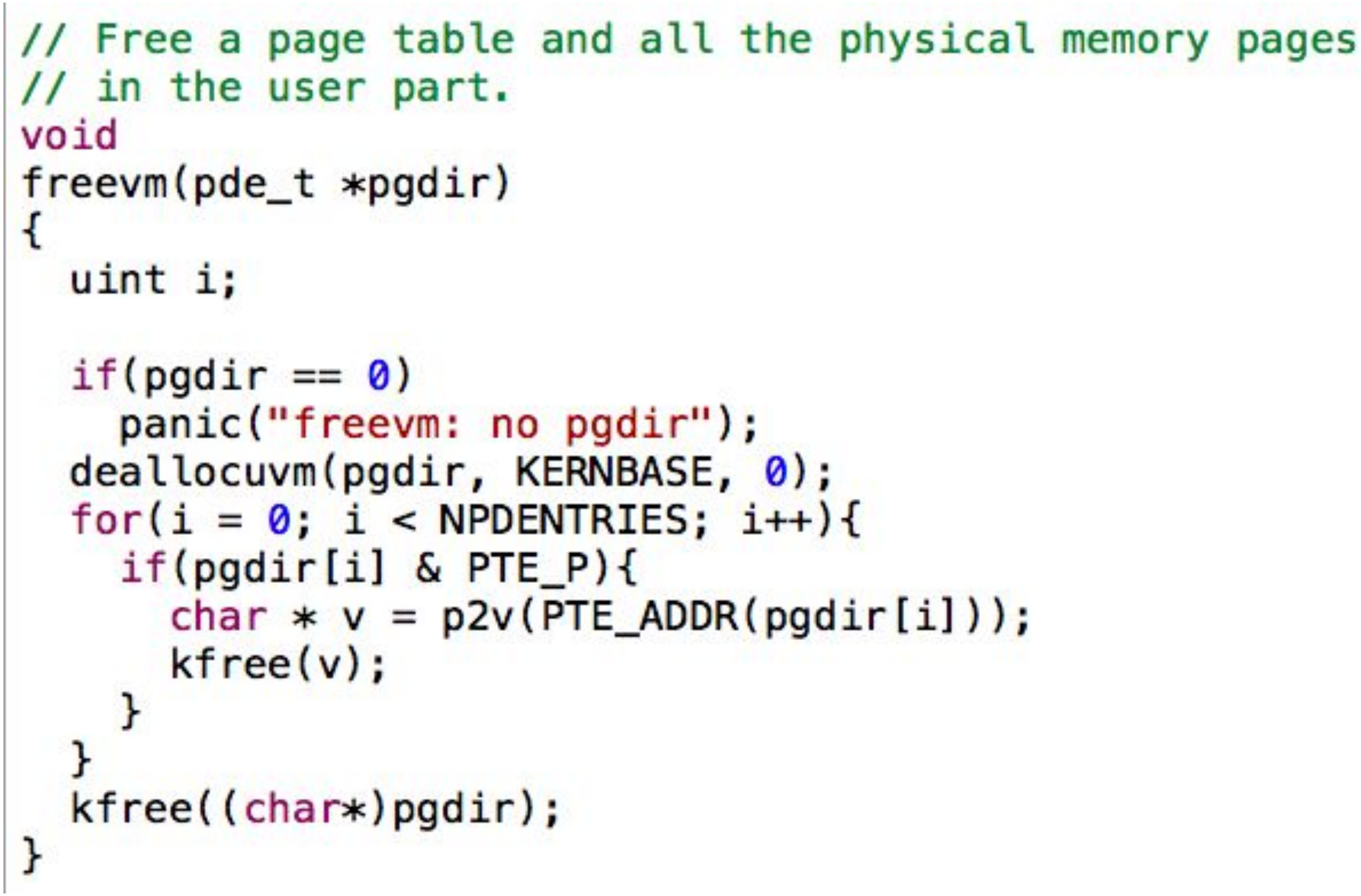
```
// Load a program segment into pgdir.  addr must be page-aligned
// and the pages from addr to addr+sz must already be mapped.
int
loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz)
{
  uint i, pa, n;
  pte_t *pte;

  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;
}
```



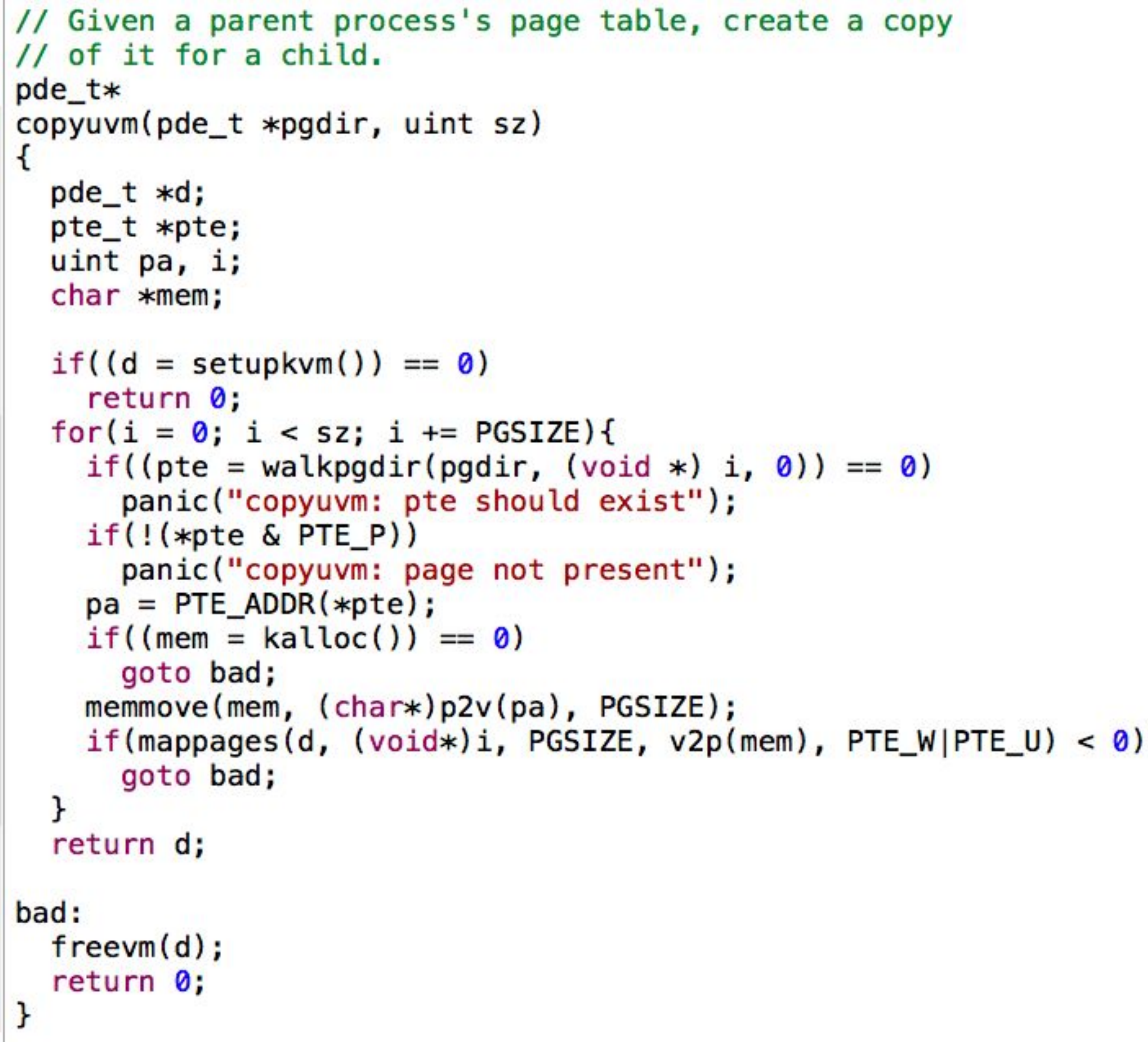
```
// Allocate page tables and physical memory to grow process from oldsz to
// newsz, which need not be page aligned.  Returns new size or 0 on error.
int
allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
{
  char *mem;
  uint a;

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

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



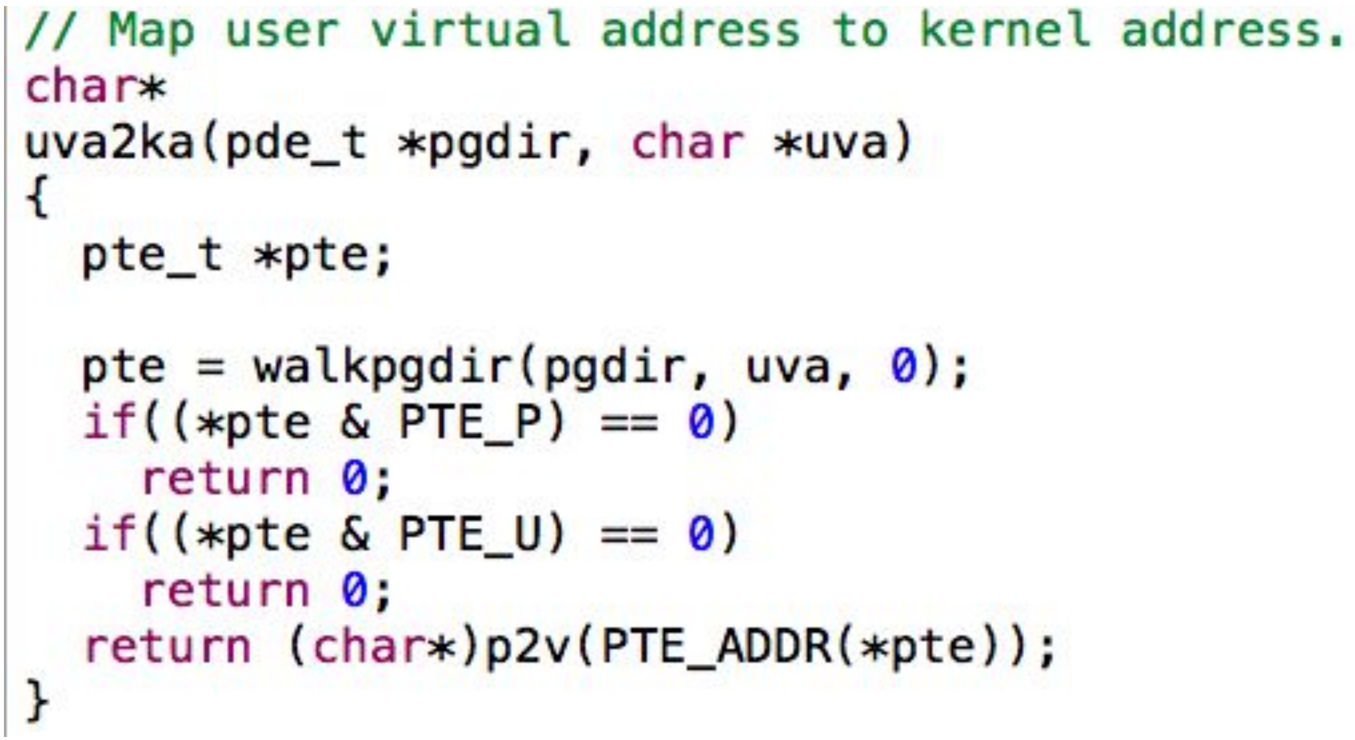
```
// Deallocate user pages to bring the process size from oldsz to
// newsz.  oldsz and newsz need not be page-aligned, nor does newsz
// need to be less than oldsz.  oldsz can be larger than the actual
// process size.  Returns the new process size.
int
deallocuvm(pde_t *pgdir, uint oldsz, uint newsz)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

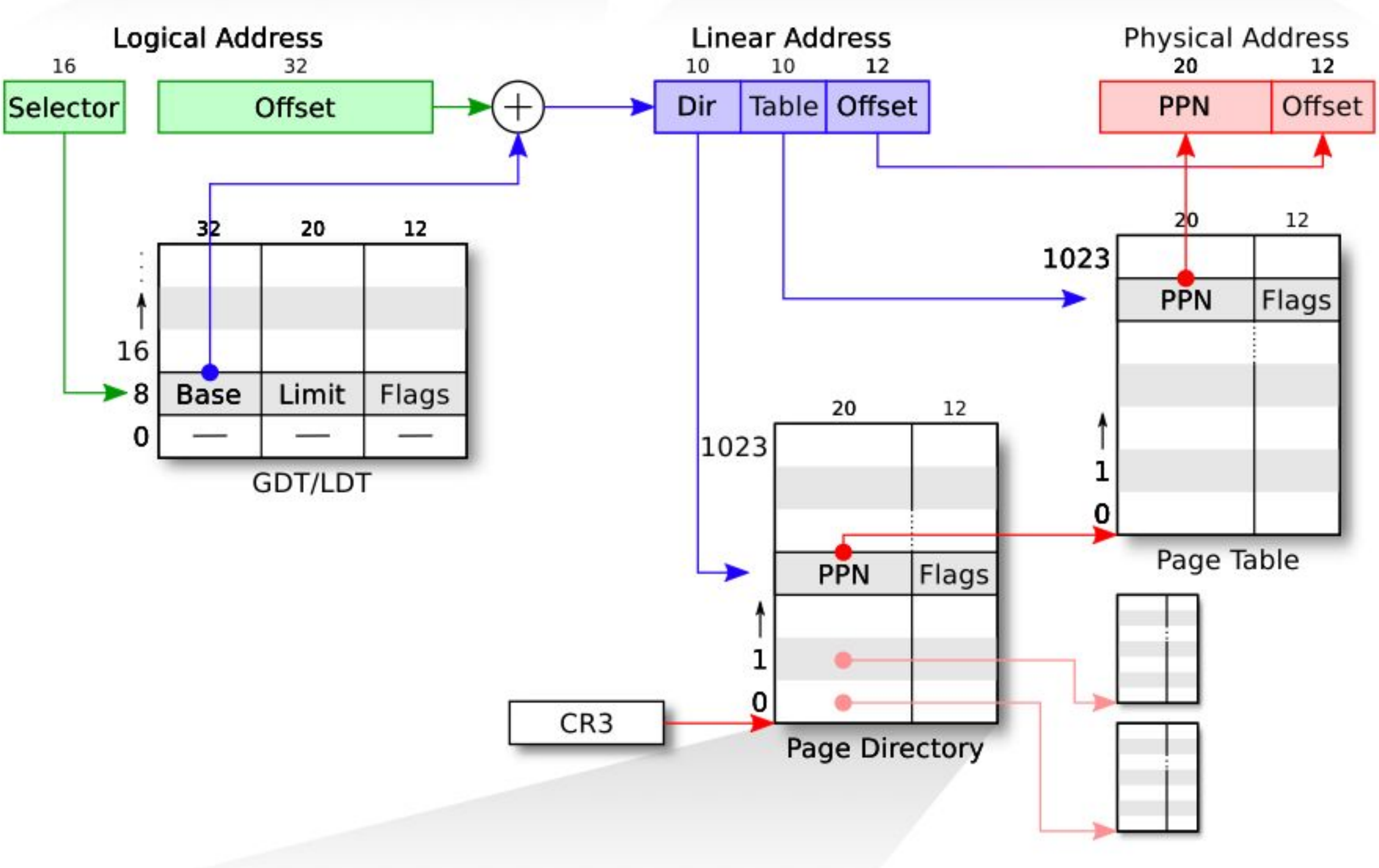
Processes

Rong Chen, Yubin Xia

IPADS, SJTU

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

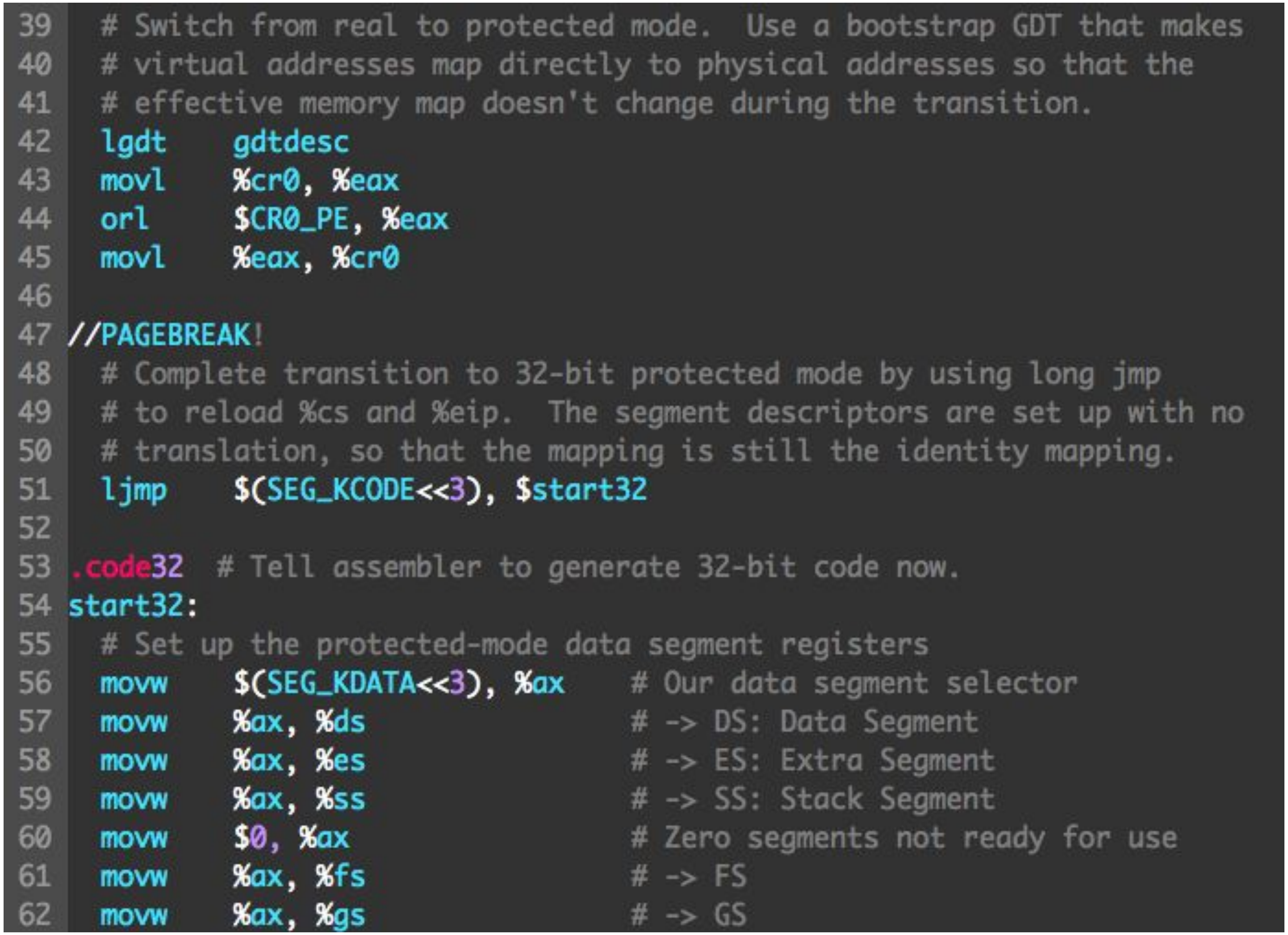
Review: LA->LA->PA



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)

Question in Last Class



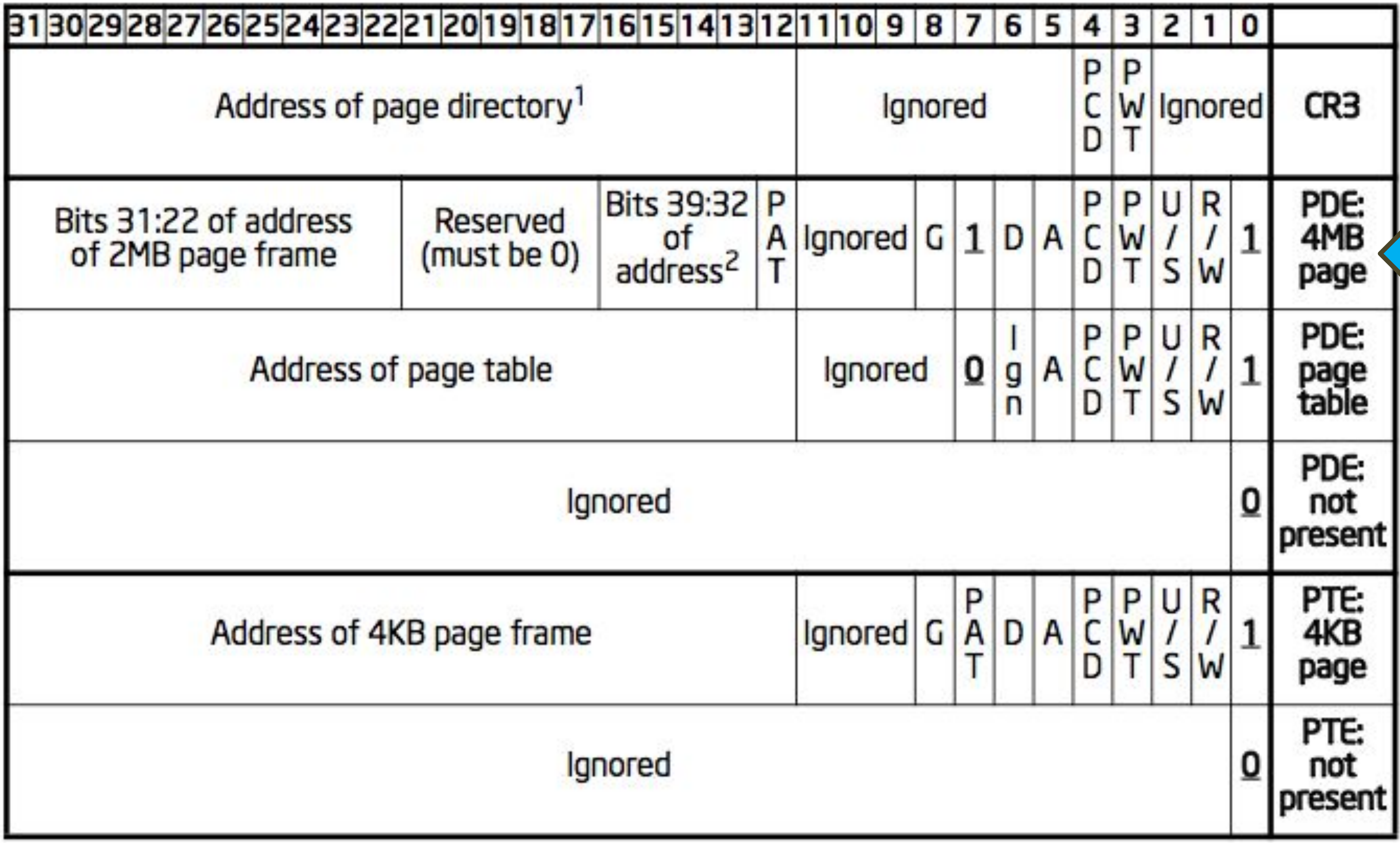
```
  # Switch from real to protected mode.  Use a bootstrap GDT that makes
  # virtual addresses map directly to physical addresses so that the
  # effective memory map doesn't change during the transition.
  lgdt    gdtdesc
  movl    %cr0, %eax
  orl     $CR0_PE, %eax
  movl    %eax, %cr0

//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
```

Explanation

- Intel’s manual 9.9.1
  - The JMP or CALL instruction immediately after the MOV CR0 instruction changes the flow of execution and serializes the processor.

- Xv6-book-rev7
  - Enabling protected mode does not immediately change how the processor translates logical to physical addresses; it is only when one loads a new value into a segment register that the processor reads the GDT and changes its internal segmentation settings. One cannot directly modify %cs, so instead the code executes an ljmp (far jump) instruction (8453), which allows a code segment selector to be specified. The jump continues execution at the next line (8456) but in doing so sets %cs to refer to the code descriptor entry in gdt. That descriptor describes a 32-bit code segment, so the processor switches into 32-bit mode.

Super Page: 4M Page



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 page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)					Bits 39:32 of address[2]				PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

PROCESS IN XV6

Processes Outline

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

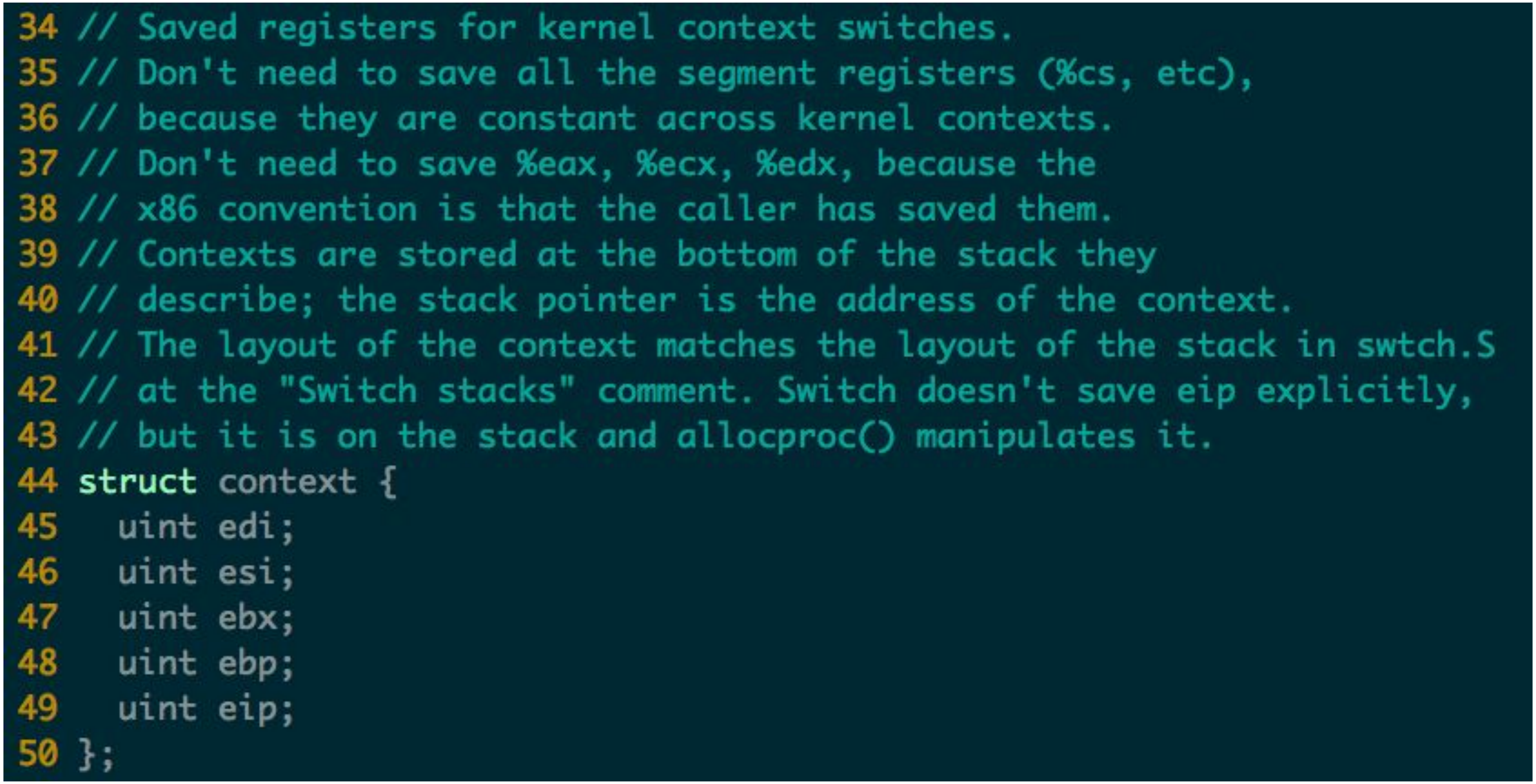
Process: Big Picture

- More programs than processors
  - How to share the limited number of processors among the programs?
- Observation: most programs don't 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.
  - What that state is depends on the processor
    - On x86, it is the processor registers.

Threads in xv6



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

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

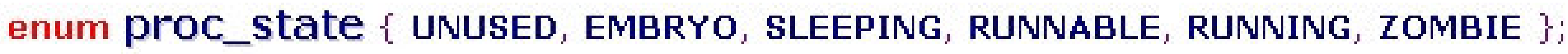
- A process includes:
  - program counter, stack, data section

- In xv6 all *user* programs contain one thread of computation and one address space

Process State

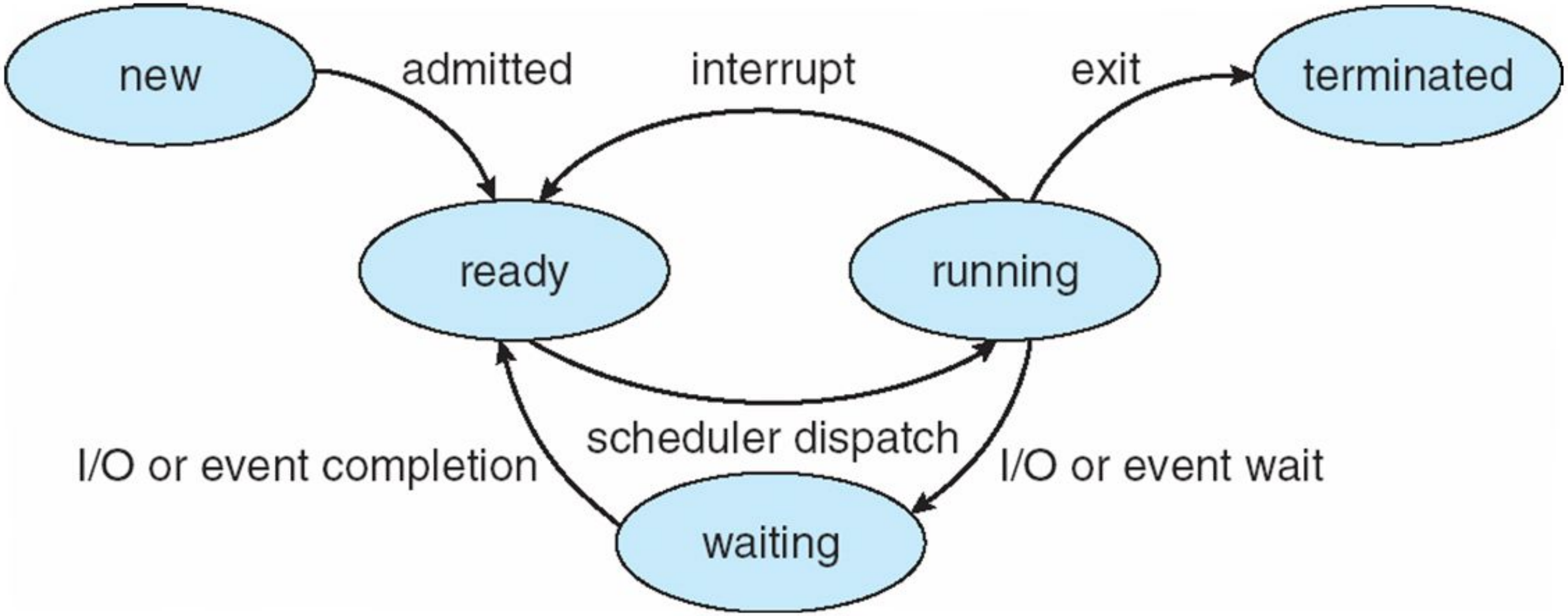
- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has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

- In xv6



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

Diagram of Process State

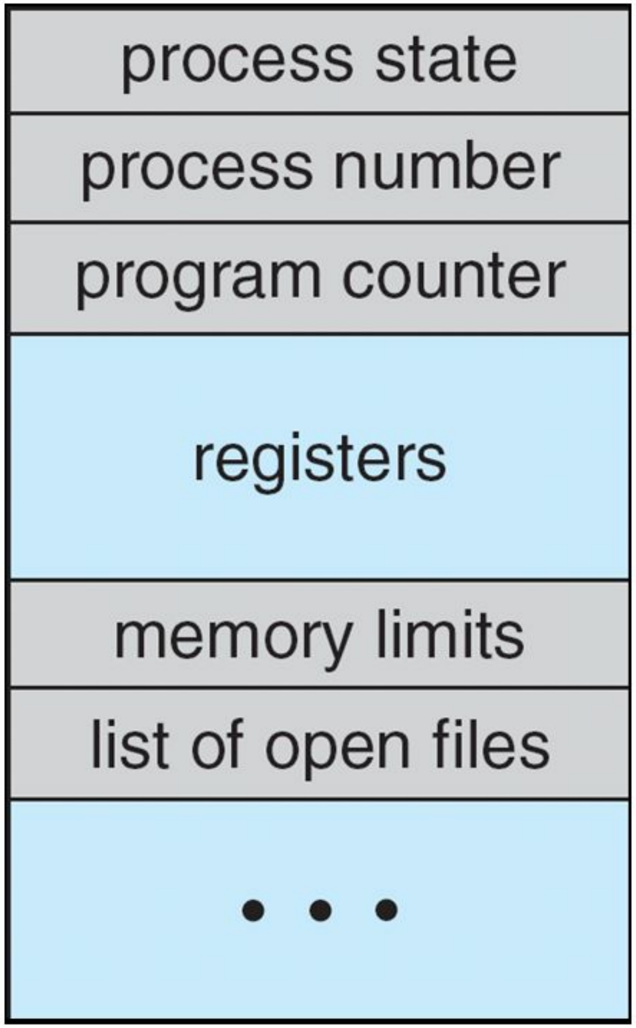


Process Control Block (PCB)

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 state

process number

program counter

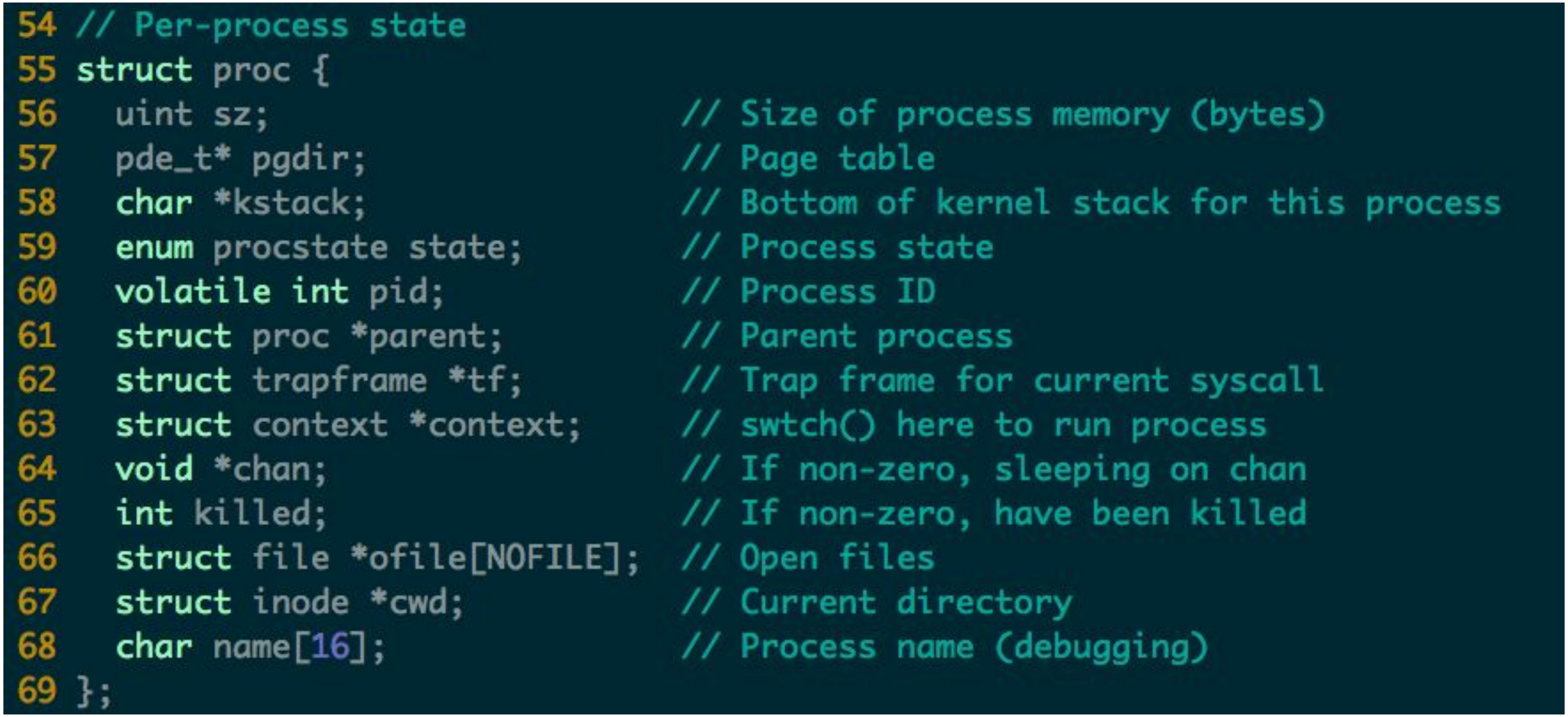
registers

memory limits

list of open files

• • •

Process in xv6 (proc.c)

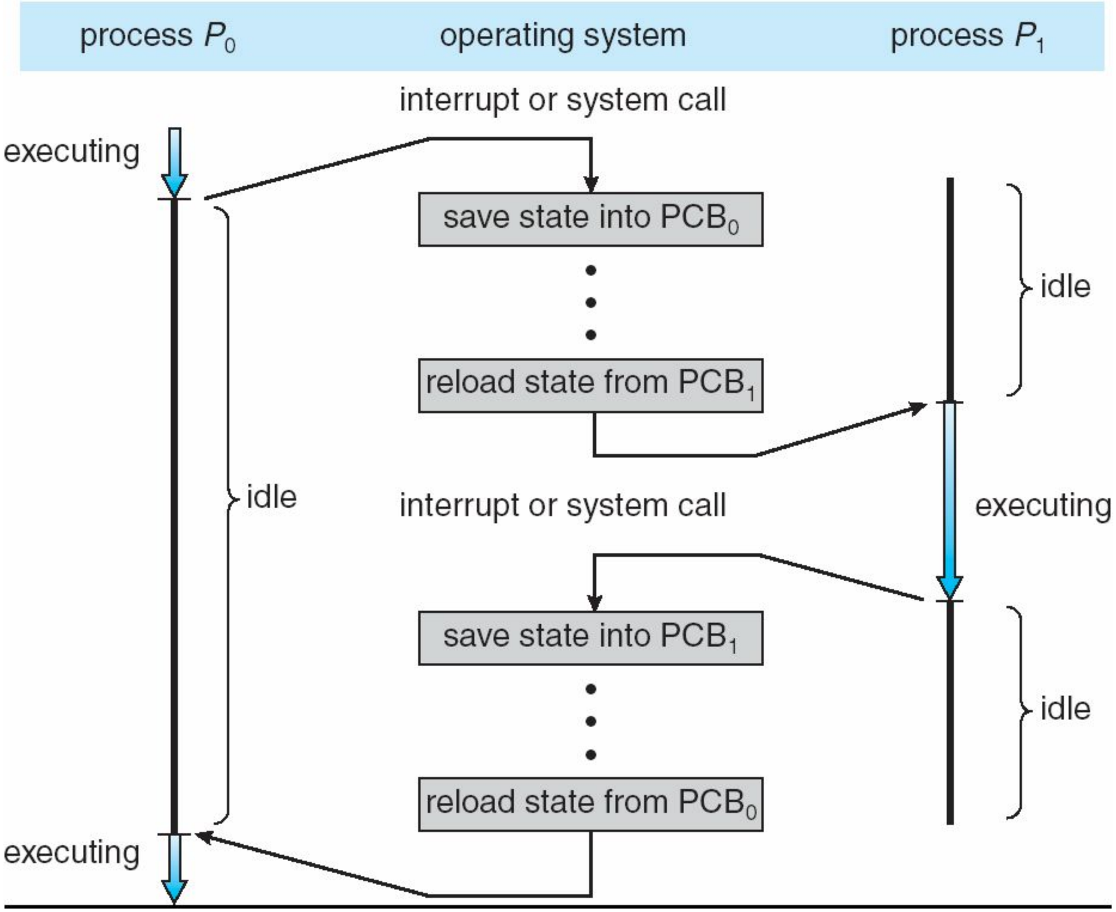


```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

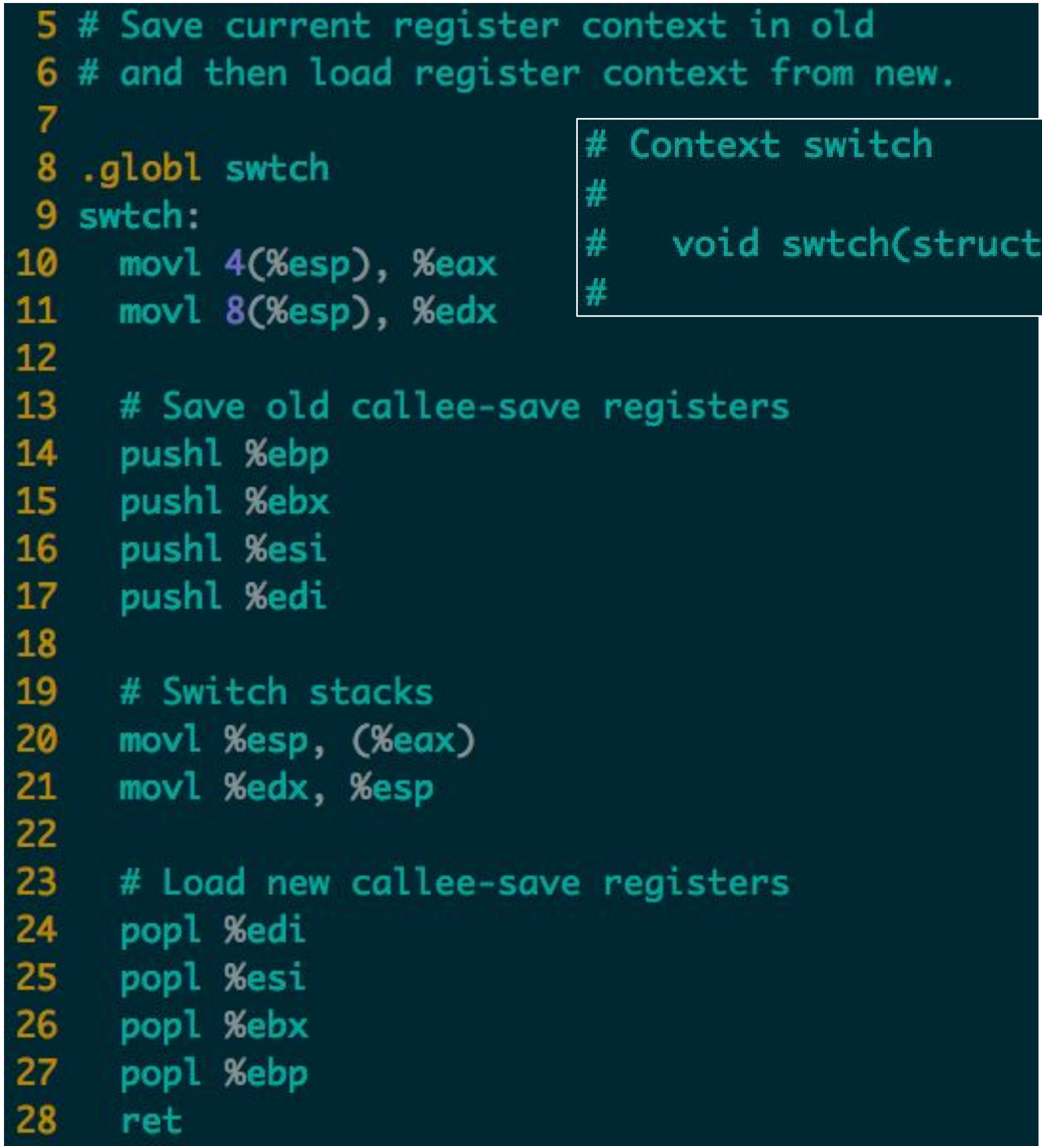
Context Switch

- 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

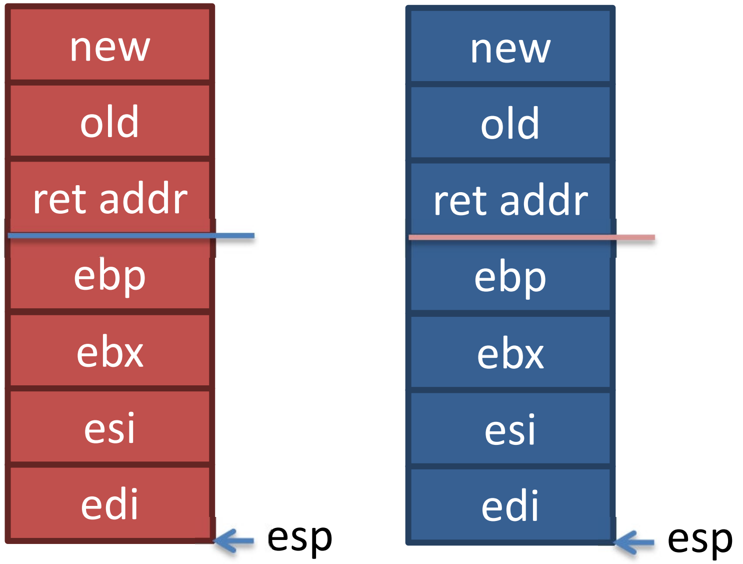


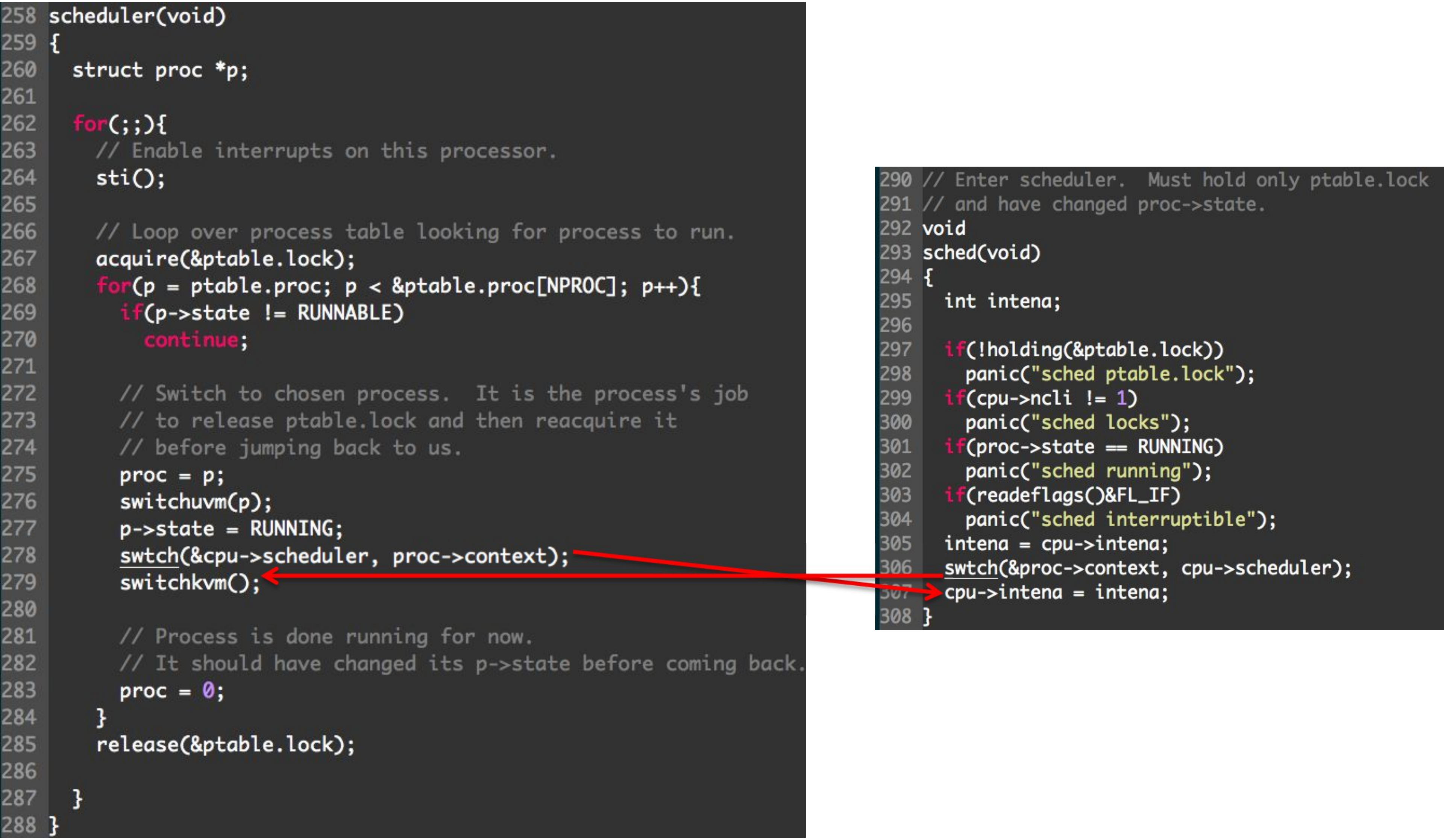
Context Switch in xv6 (swtch.S)



```
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
```





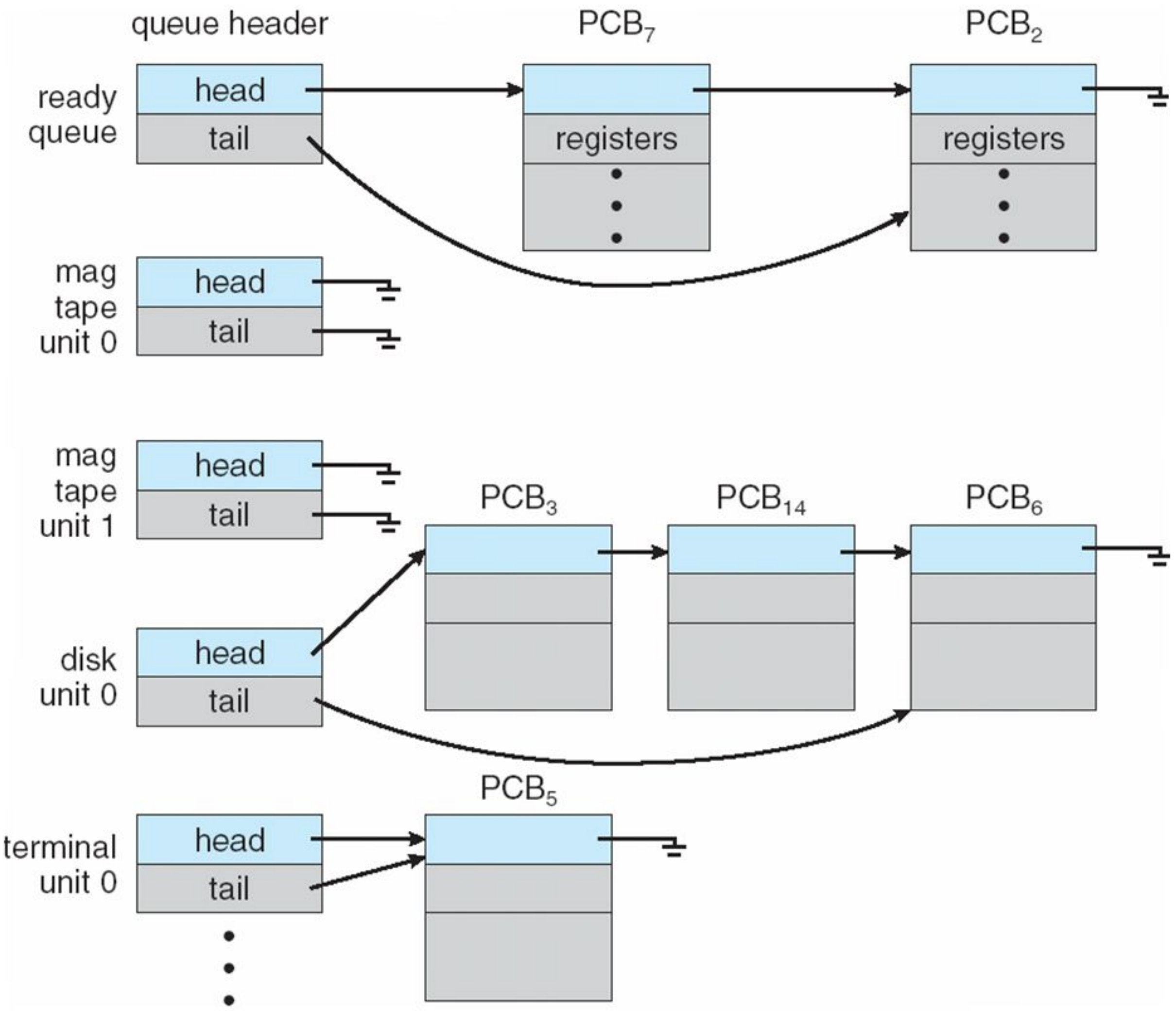
```
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

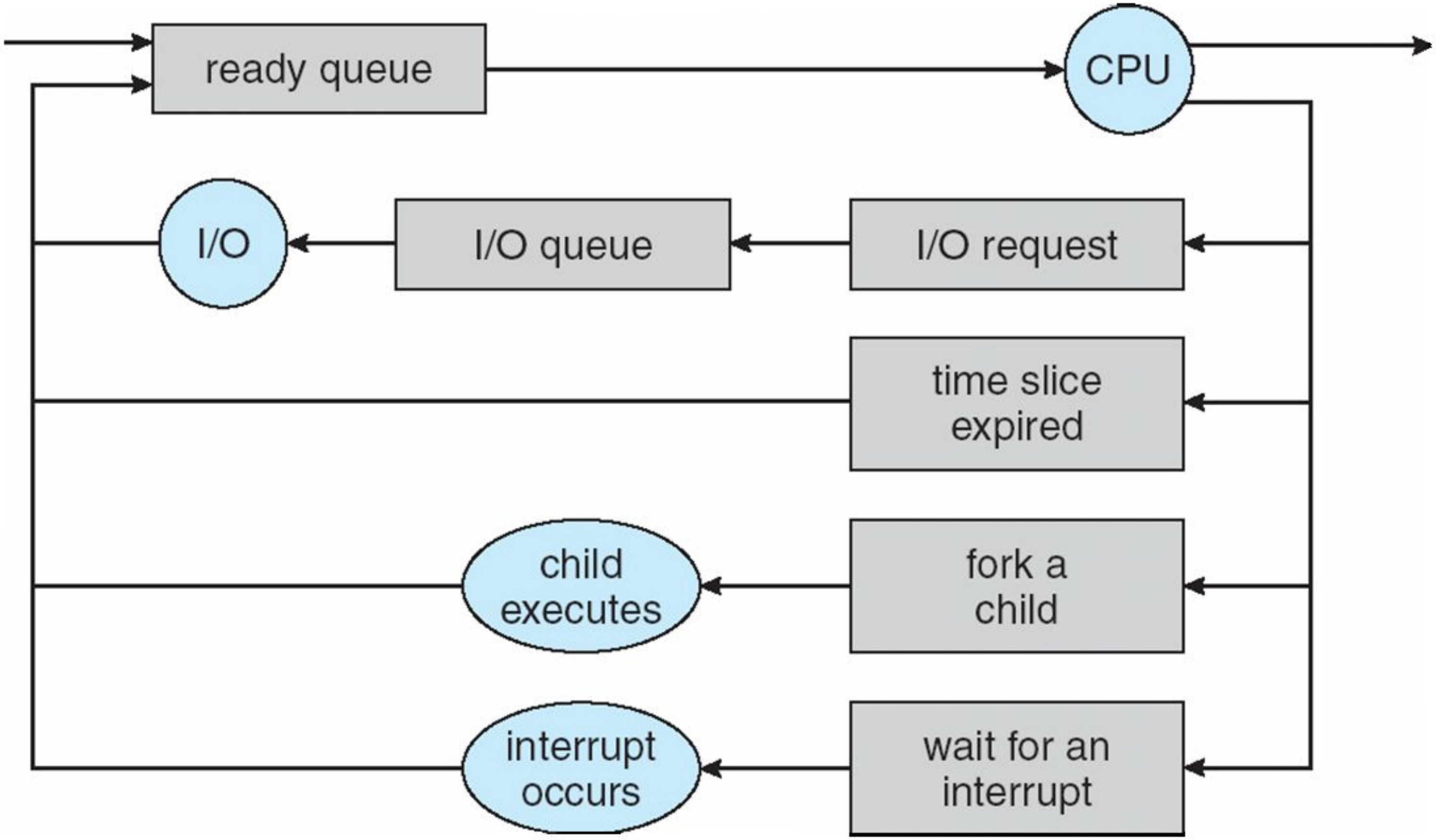
Process Scheduling Queues

- **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

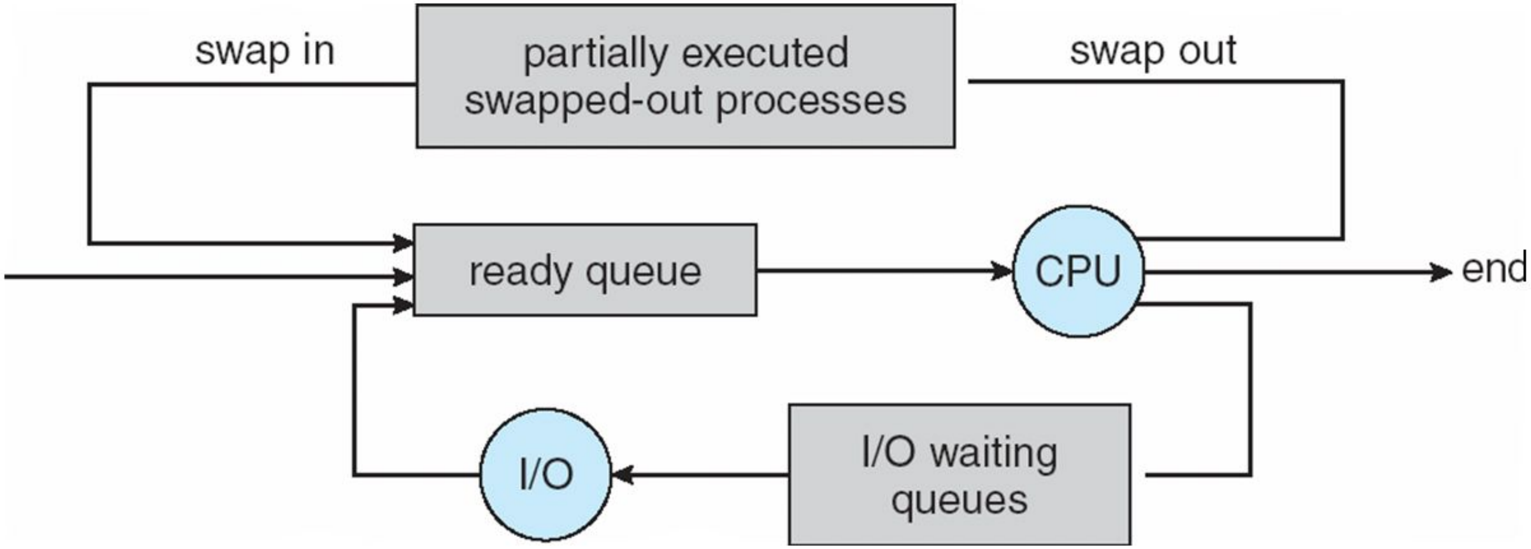


Schedulers

- **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

Addition of Medium Term Scheduling



Schedulers (Cont)

- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)

- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (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

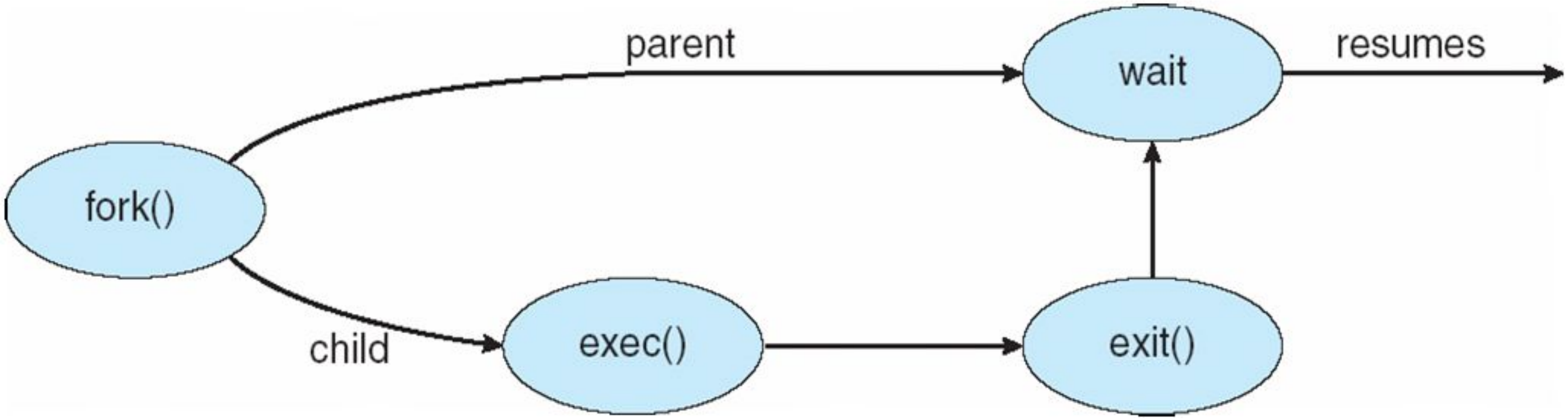
- **Parent** process create **children** processes
  - forming a tree of processes
  - Generally, process identified and managed via **a process identifier** (**pid**)

- Resource sharing policies
  - share all resources
  - share subset of parent’s resources
  - share no resources

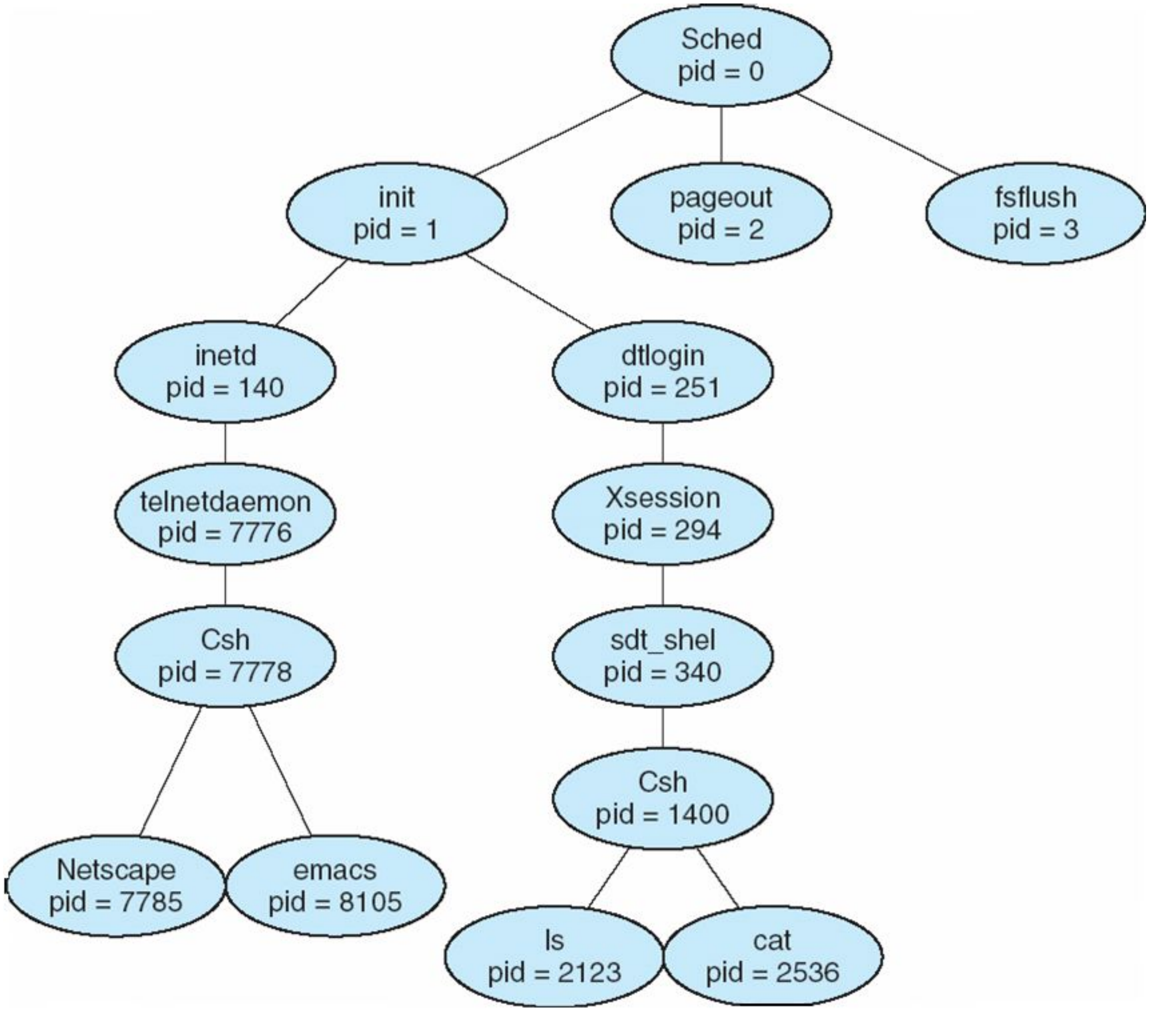
Process Creation (Cont)

- 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

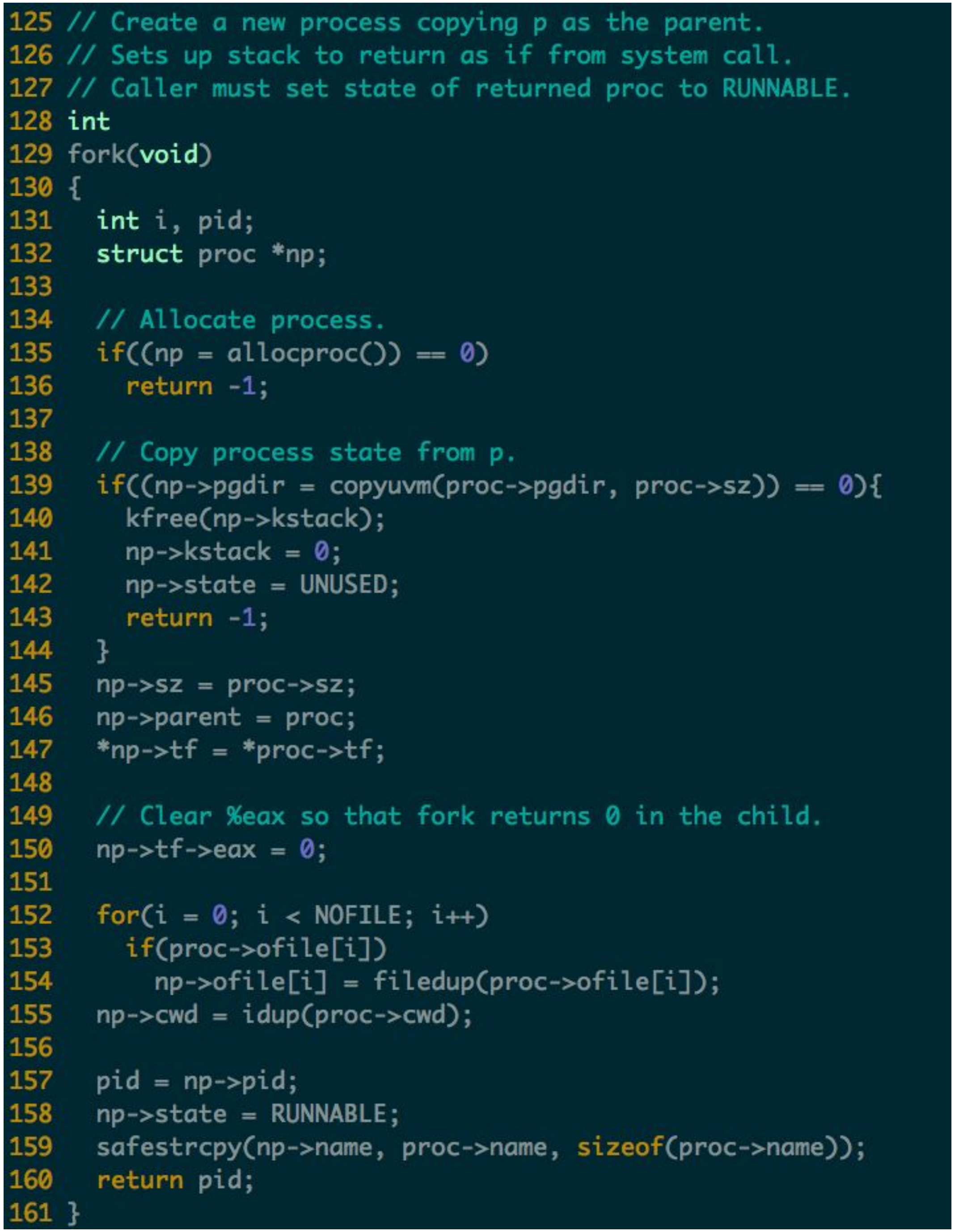
Process Creation



A tree of processes on a typical Solaris



Process Creation in xv6



```c
// Create a new process copying p as the parent.
// Sets up stack to return as if from system call.
// Caller must set state of returned proc to RUNNABLE.
int
fork(void)
{
  int i, pid;
  struct proc *np;

  // Allocate process.
  if((np = allocproc()) == 0)
    return -1;

  // Copy process state from p.
  if((np->pgdir = copyuvm(proc->pgdir, proc->sz)) == 0){
    kfree(np->kstack);
    np->kstack = 0;
    np->state = UNUSED;
    return -1;
  }
  np->sz = proc->sz;
  np->parent = proc;
  *np->tf = *proc->tf;

  // Clear %eax so that fork returns 0 in the child.
  np->tf->eax = 0;

  for(i = 0; i < NOFILE; i++)
    if(proc->ofile[i])
      np->ofile[i] = filedup(proc->ofile[i]);
  np->cwd = idup(proc->cwd);

  pid = np->pid;
  np->state = RUNNABLE;
  safestrcpy(np->name, proc->name, sizeof(proc->name));
  return pid;
}
```

Read following code

- allocproc()

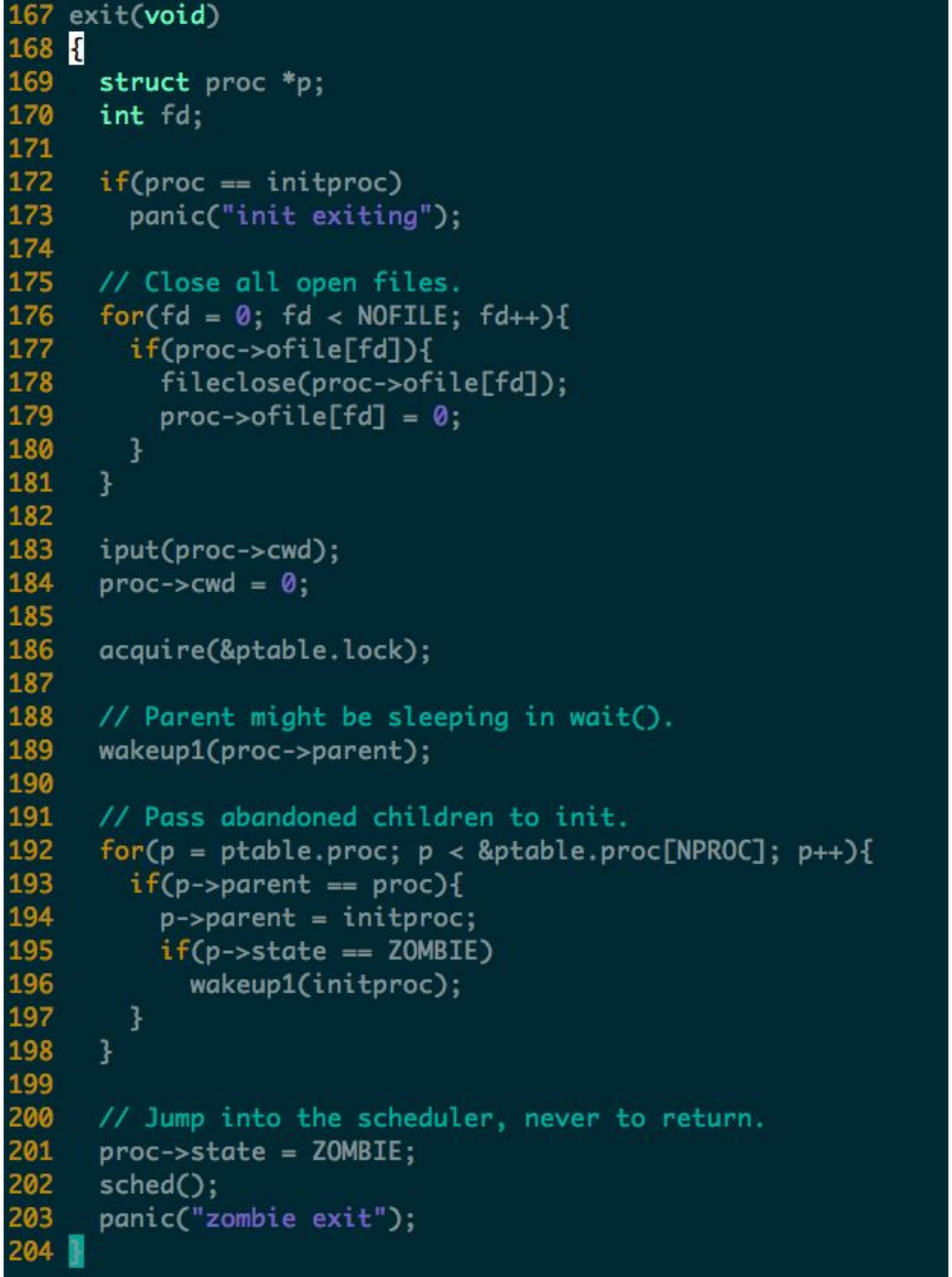
Process Termination

- 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**

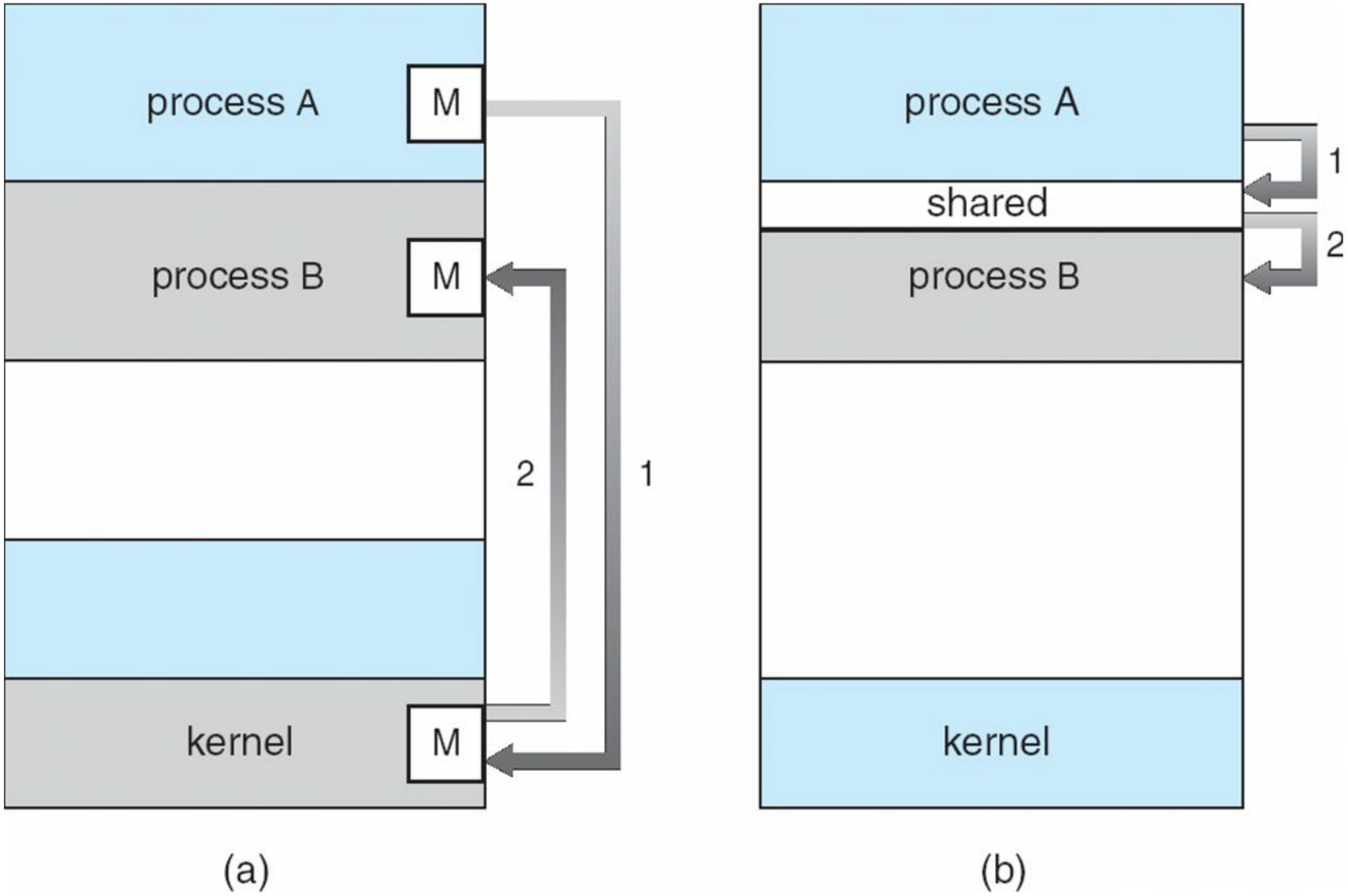


```
exit(void)
{
  struct proc *p;
  int fd;

  if(proc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(proc->ofile[fd]){
      fileclose(proc->ofile[fd]);
      proc->ofile[fd] = 0;
    }
  }

  iput(proc->cwd);
  proc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(proc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == proc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  proc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

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 BUFFER_SIZE-1 elements

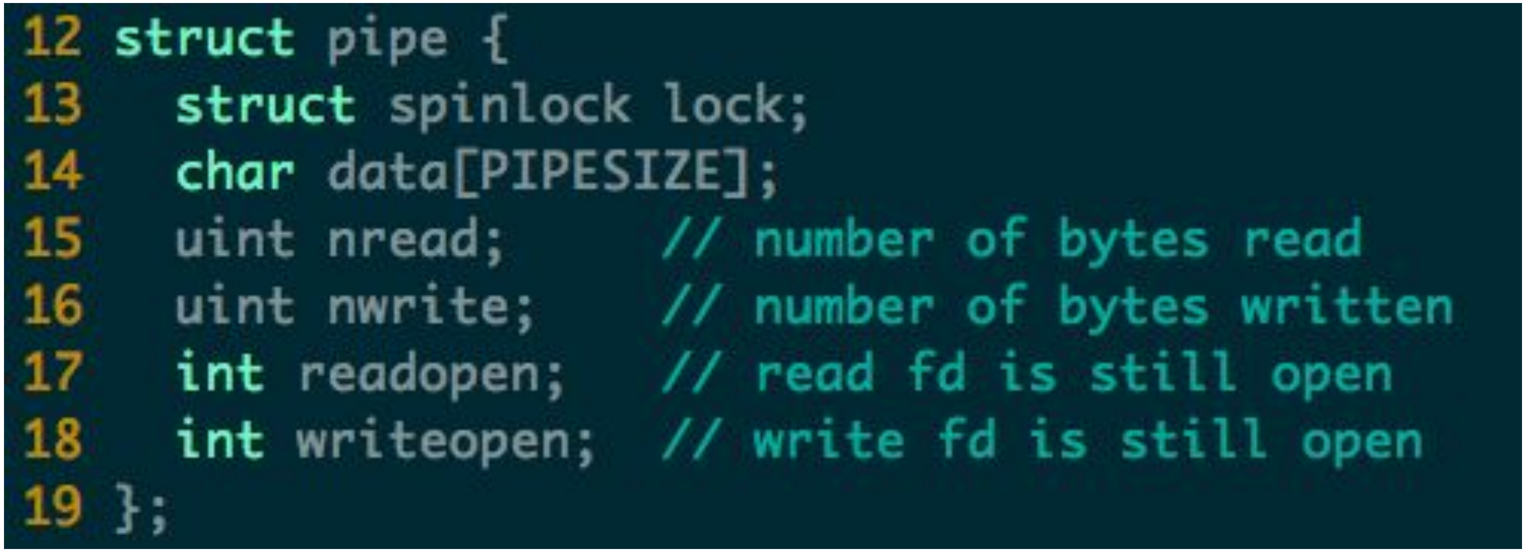
Bounded-Buffer – Producer

```
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;
 }
```

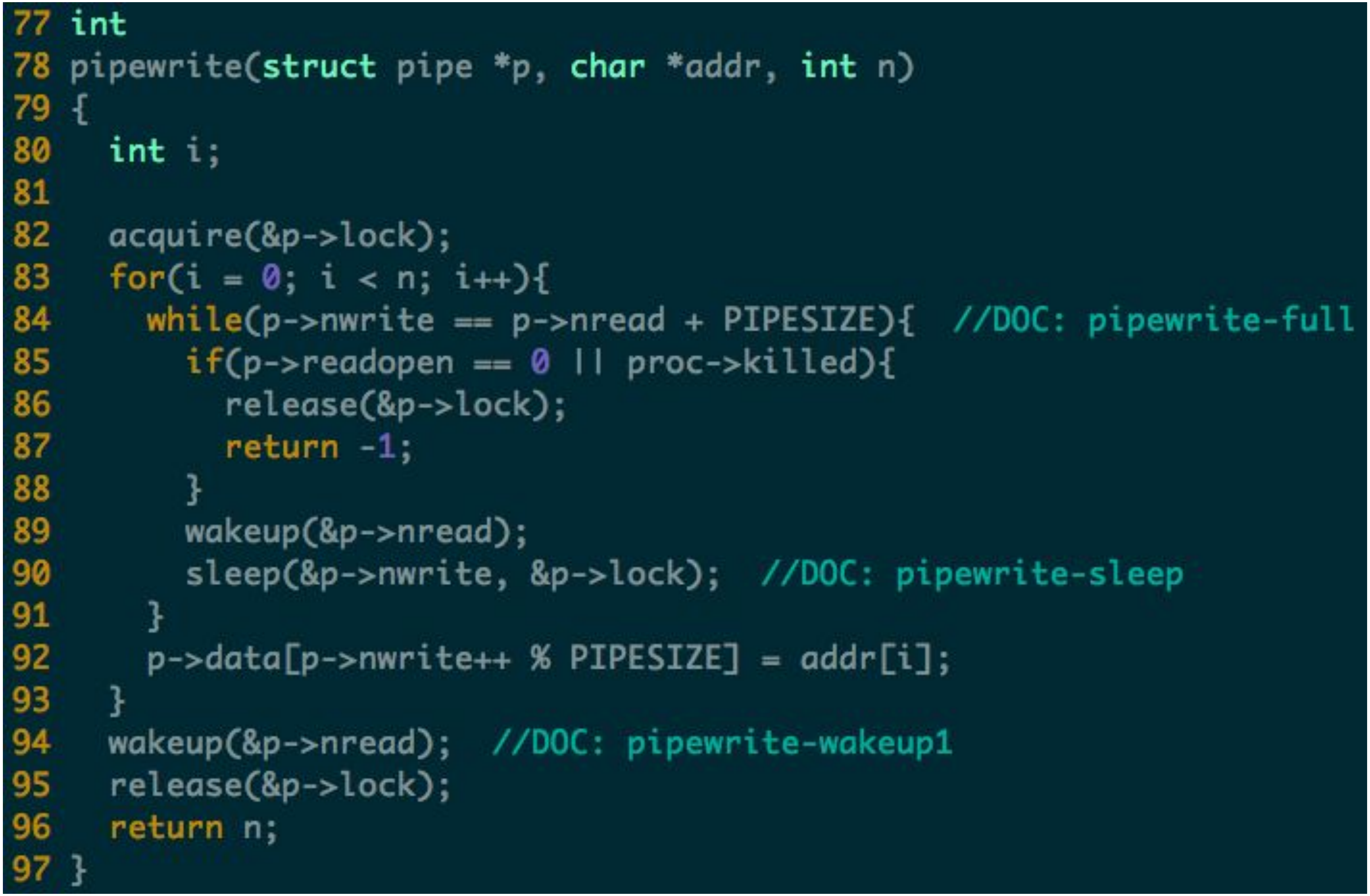
Bounded Buffer – Consumer

```
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;
  }
```

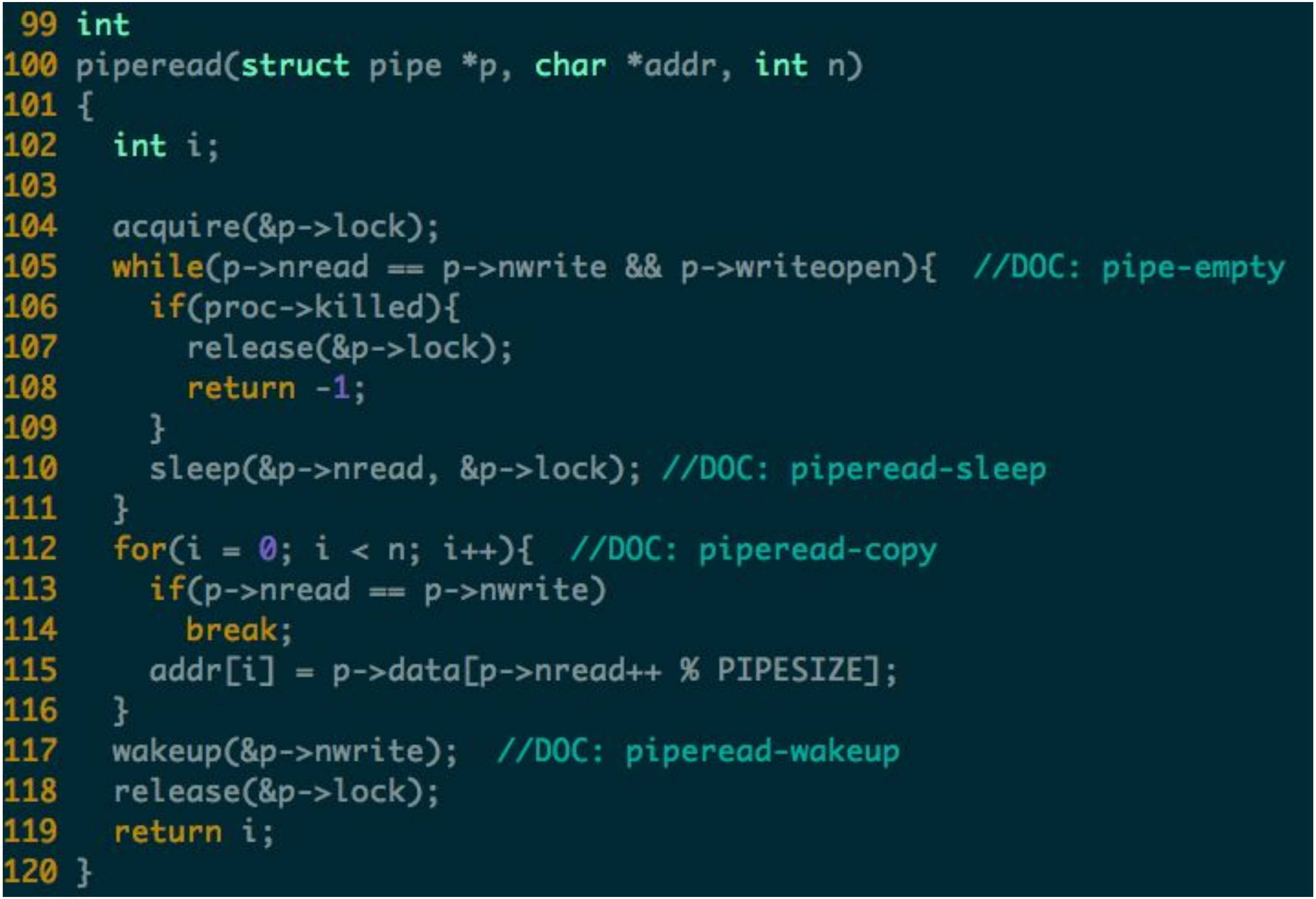
Example of IPC in xv6: Pipe (pipe.c)



```
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```



```c
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```



```c
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

Interprocess Communication – 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);
```

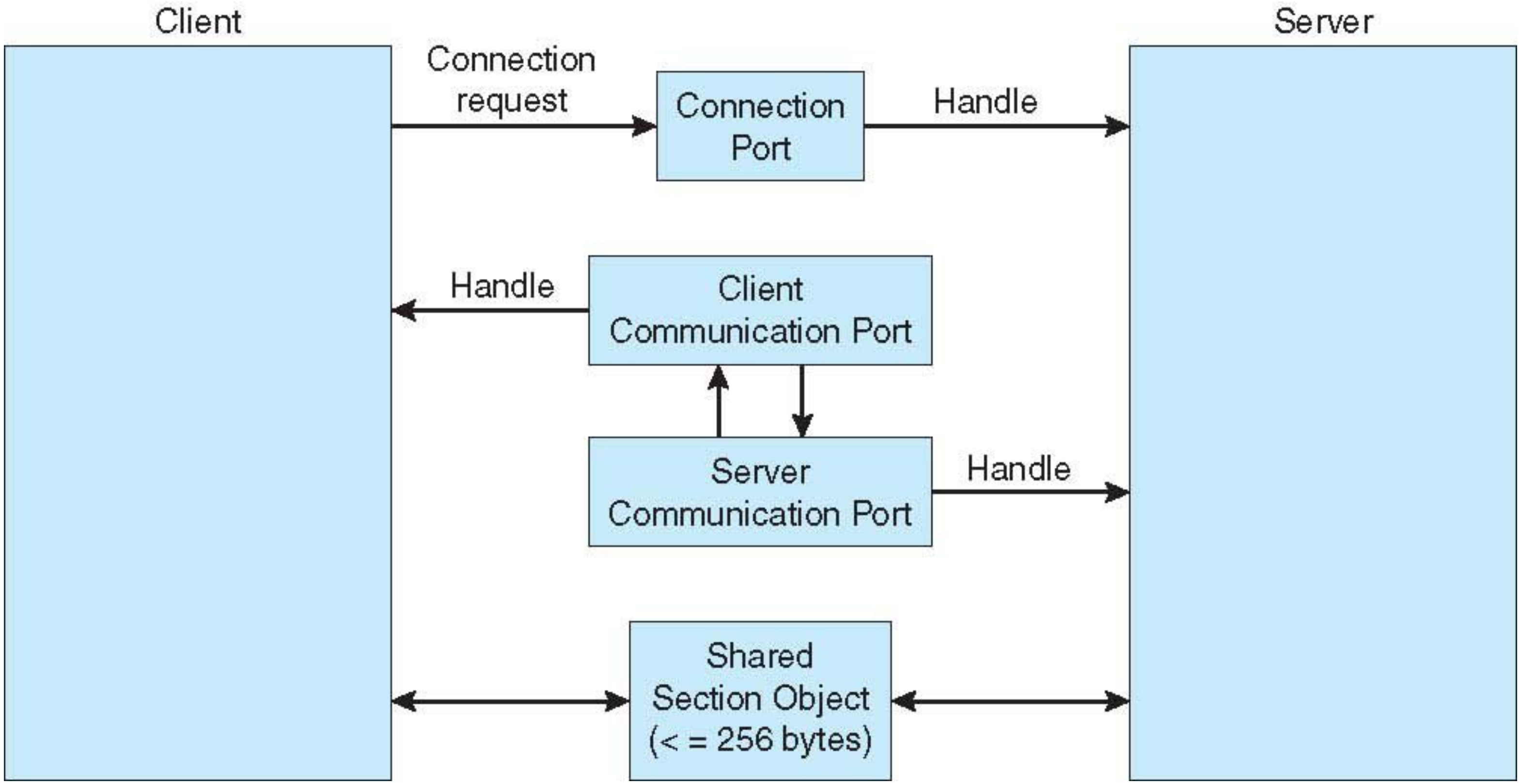
Examples of IPC Systems – WinXP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels

Examples of IPC Systems – WinXP (cont.)

- Communication works as follows:
  - The client opens a handle to the subsystem’s connection port object
  - The client sends a connection request
  - The server creates two private communication ports and returns the handle to one of them to the client
  - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies

Local Procedure Calls in Windows XP


Client
Server
Connection request
Connection Port
Handle
Handle
Client Communication Port
Server Communication Port
Handle
Shared Section Object (< = 256 bytes)

Communications in Client-Server Systems

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

INTERRUPT, TRAP, SYS CALL

Concepts

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

Requirements

- 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

IDT

- IDT: interrupt descriptor 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*
  - *n* specifies the index into the IDT

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 <= 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

Kernel Stack

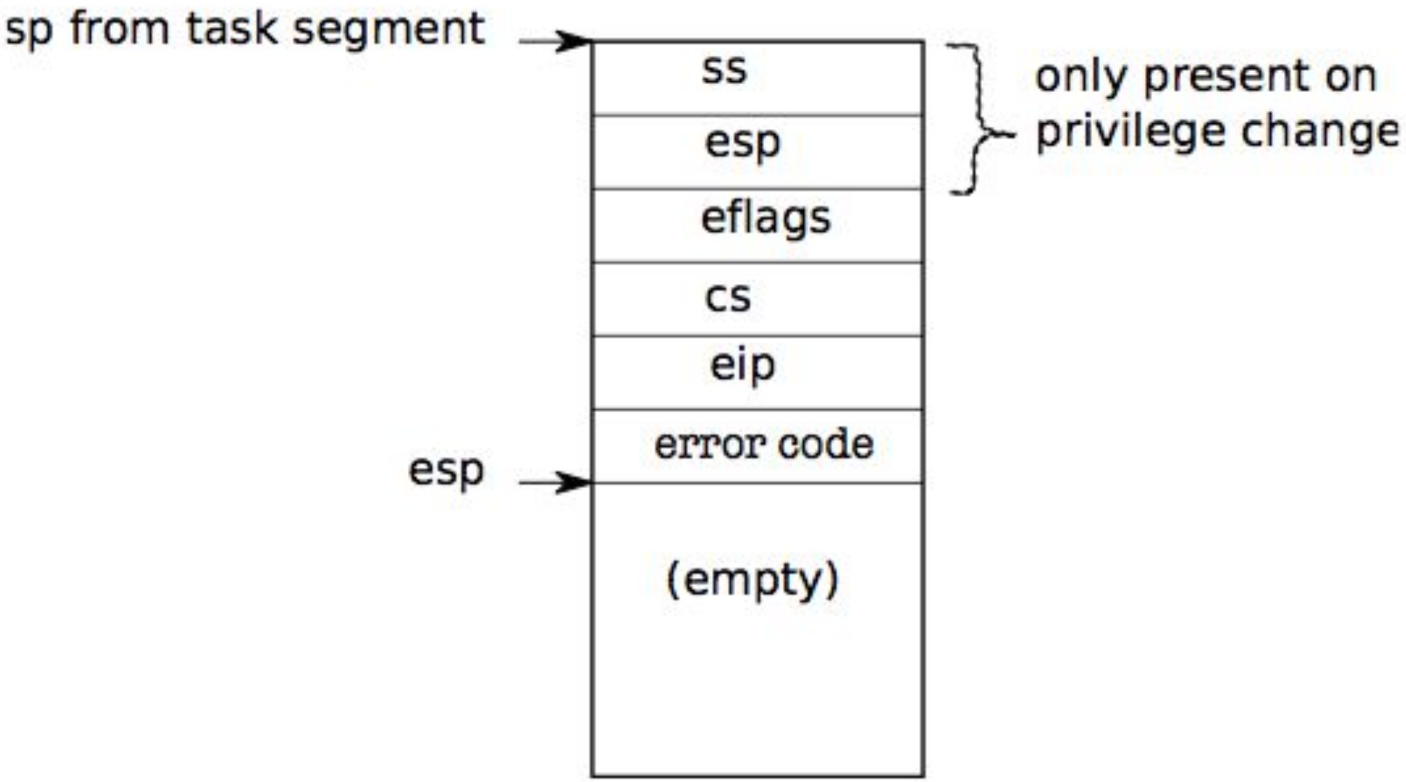
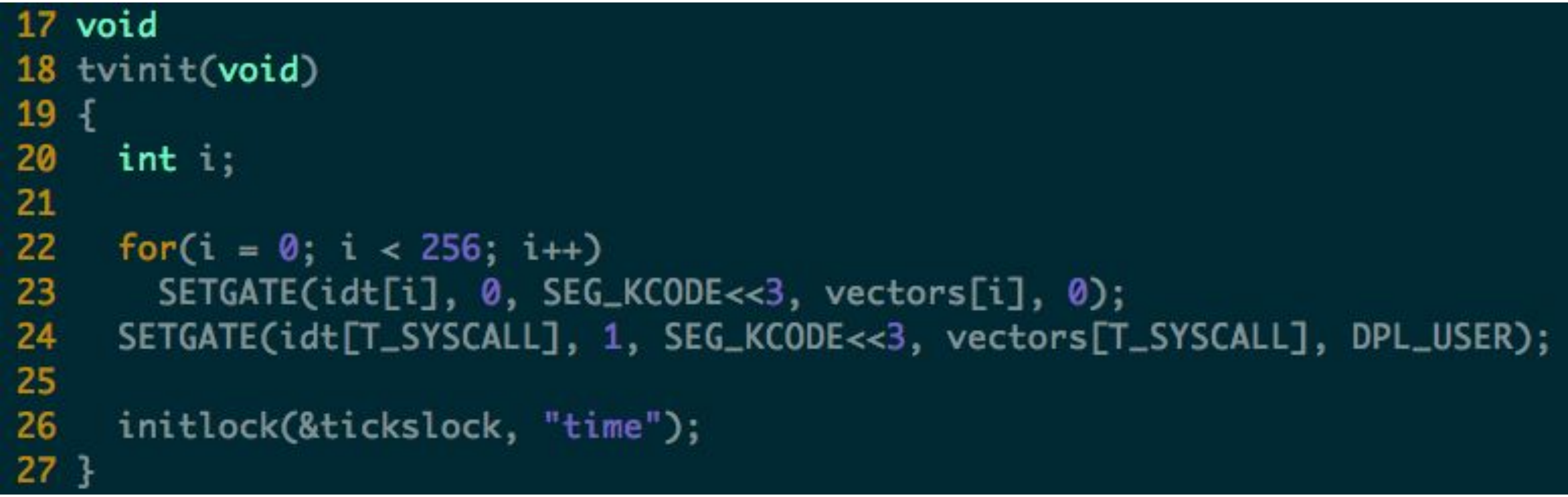


Figure 3-1. Kernel stack after an int instruction.

tvinit



```c
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```

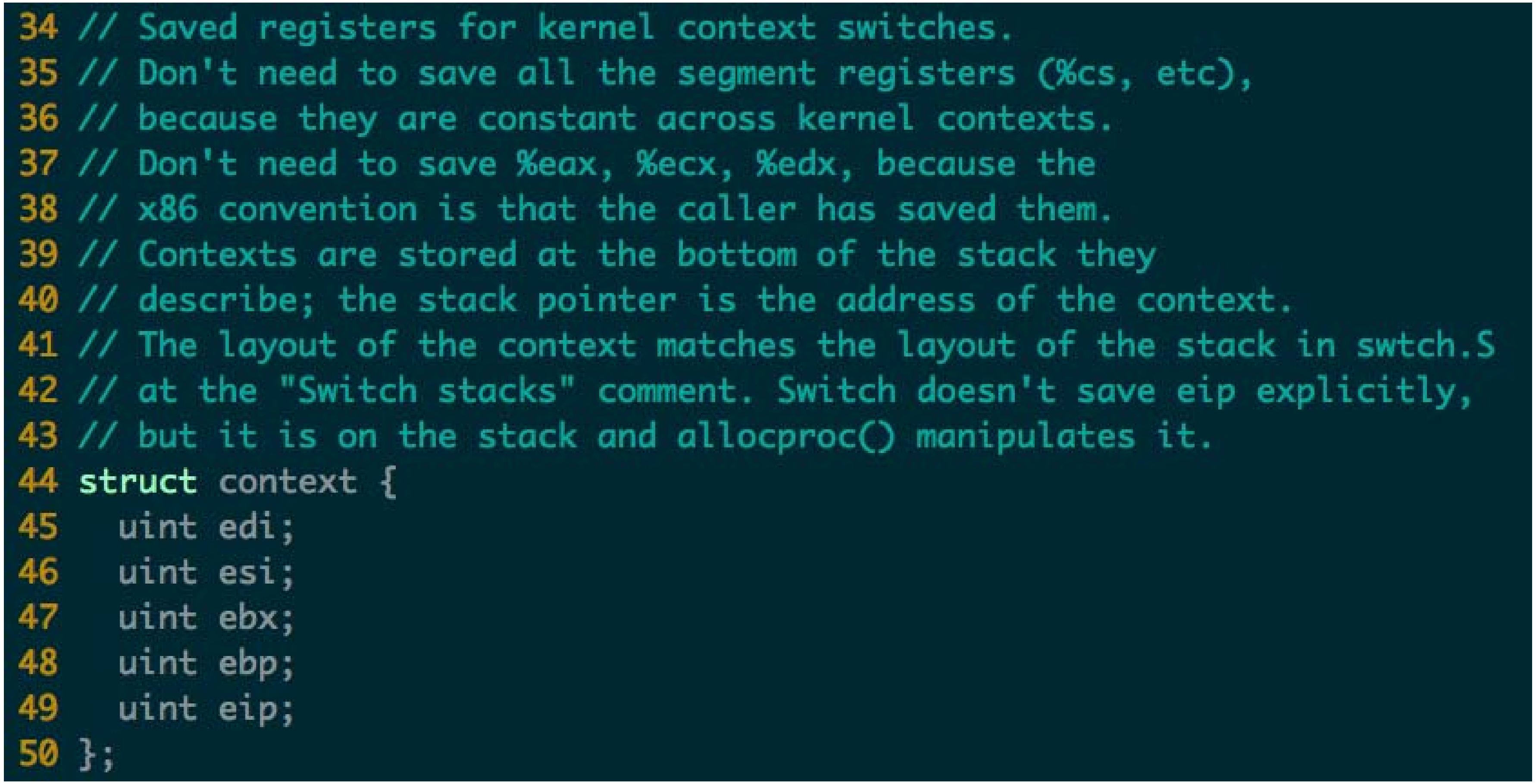
IPC

Yubin Xia, Rong Chen

IPADS, SJTU

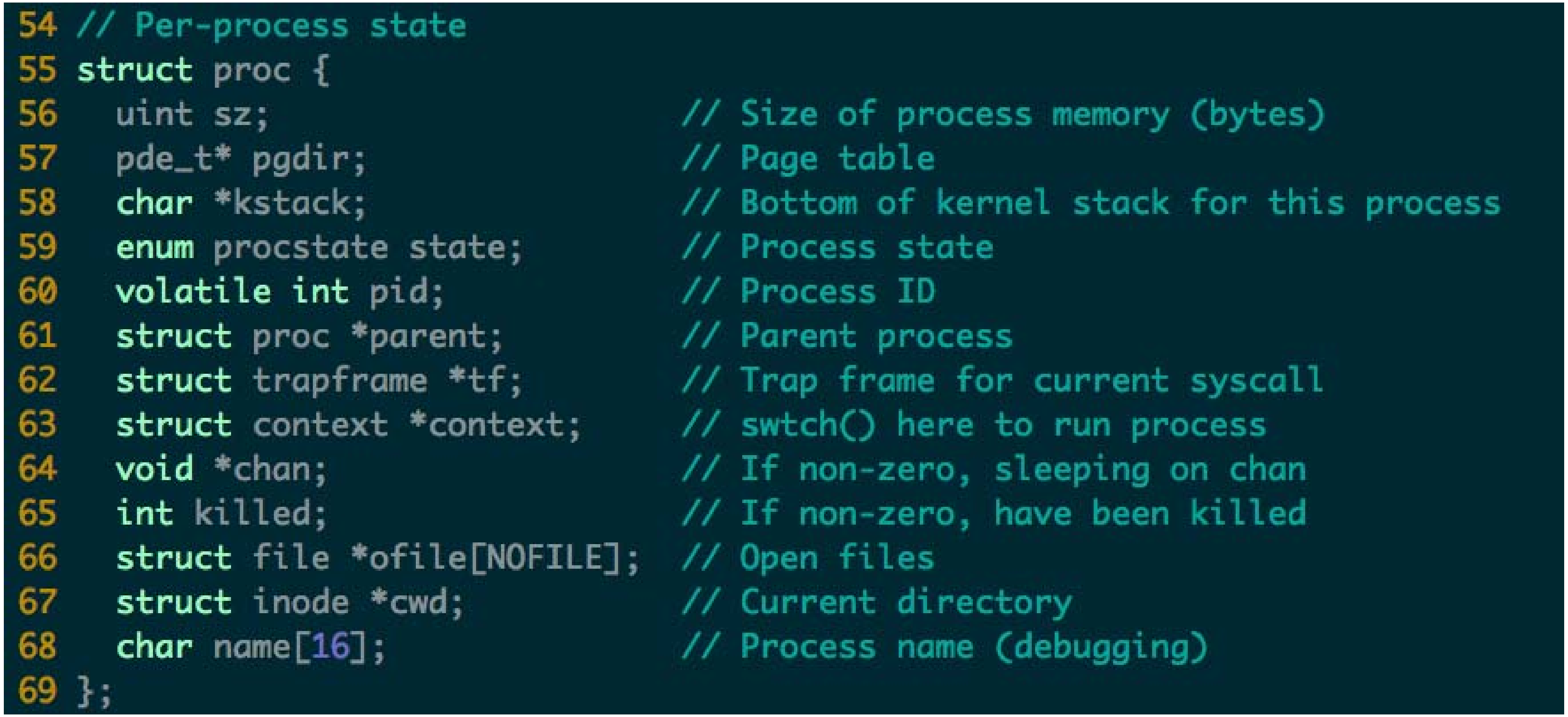
ACKs: Some slides are adapted from the textbook's original slides and Frans's OS course notes

Review: Threads in xv6



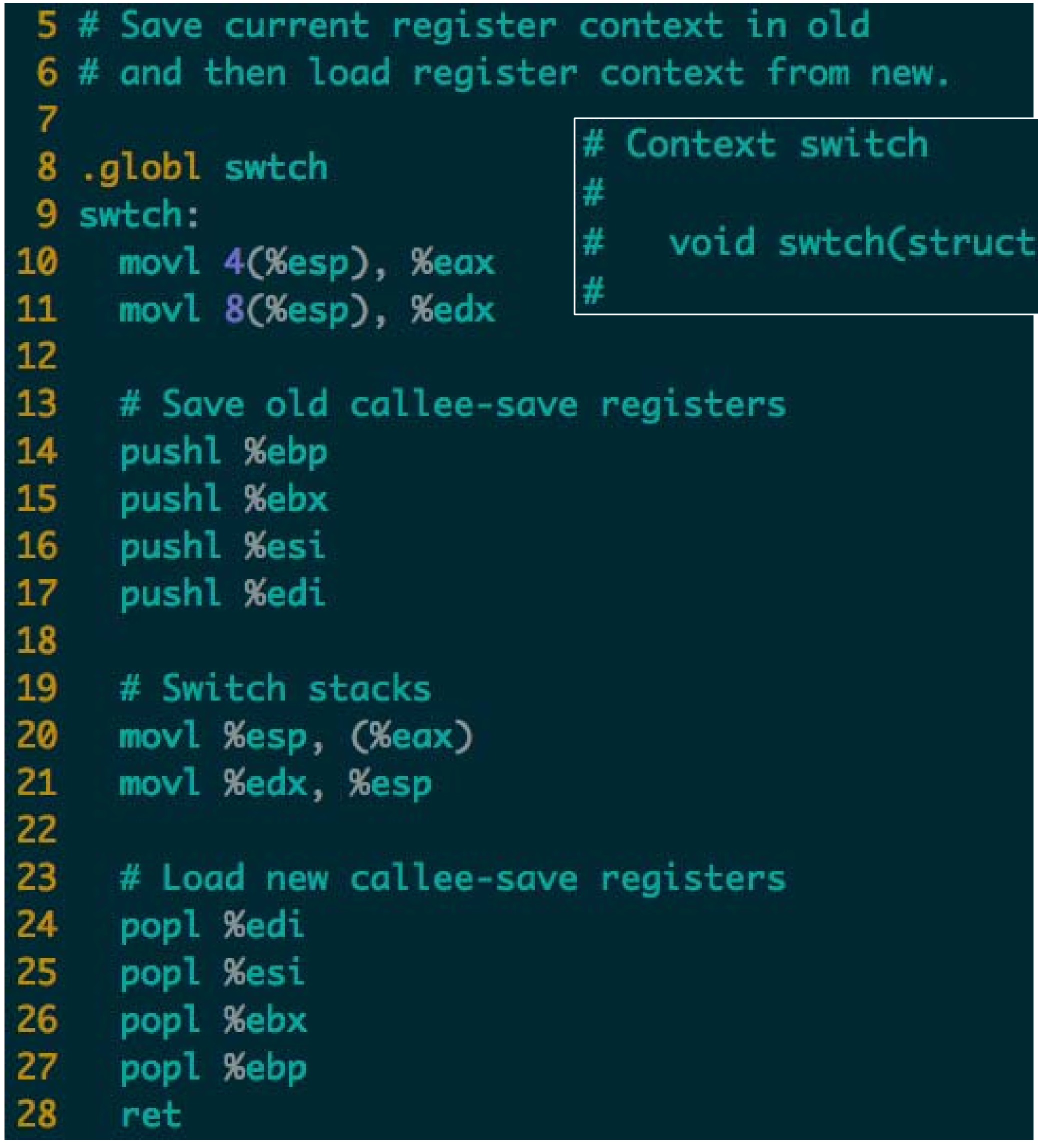
```
// Saved registers for kernel context switches.
// Don't need to save all the segment registers (%cs, etc),
// because they are constant across kernel contexts.
// Don't need to save %eax, %ecx, %edx, because the
// x86 convention is that the caller has saved them.
// Contexts are stored at the bottom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

Review: Process in xv6 (proc.c)



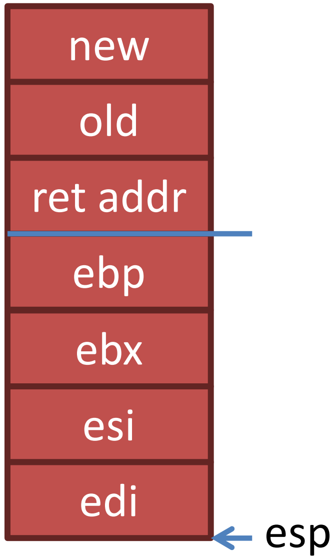
```c
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};
```

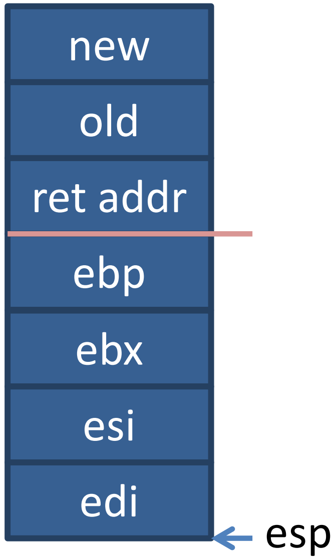
Review: Context Switch in xv6



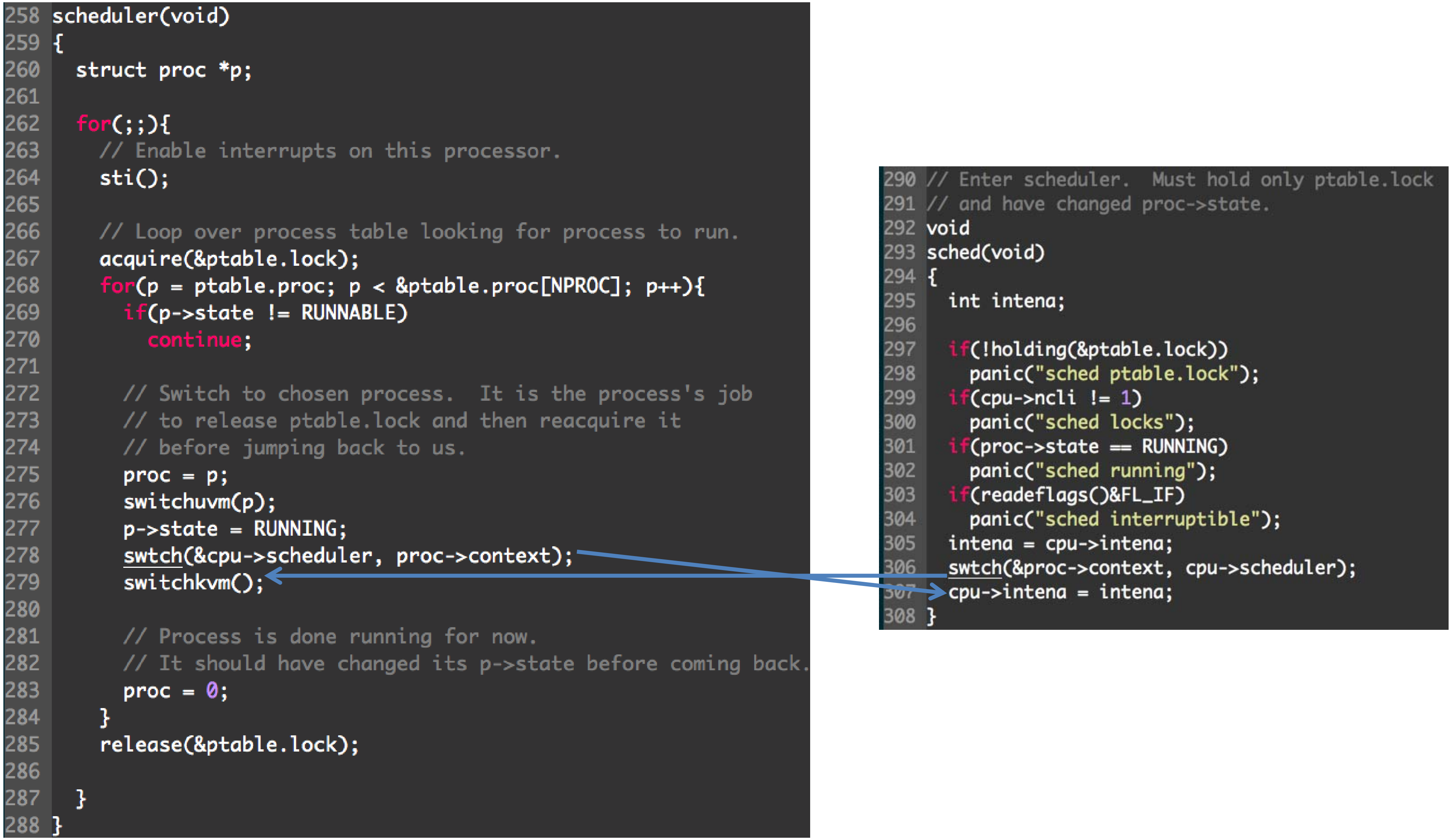
```
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-save registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-save registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

```
# Context switch
#
#   void swtch(struct context **old, struct context *new);
#
```





Review: Context Switch in xv6



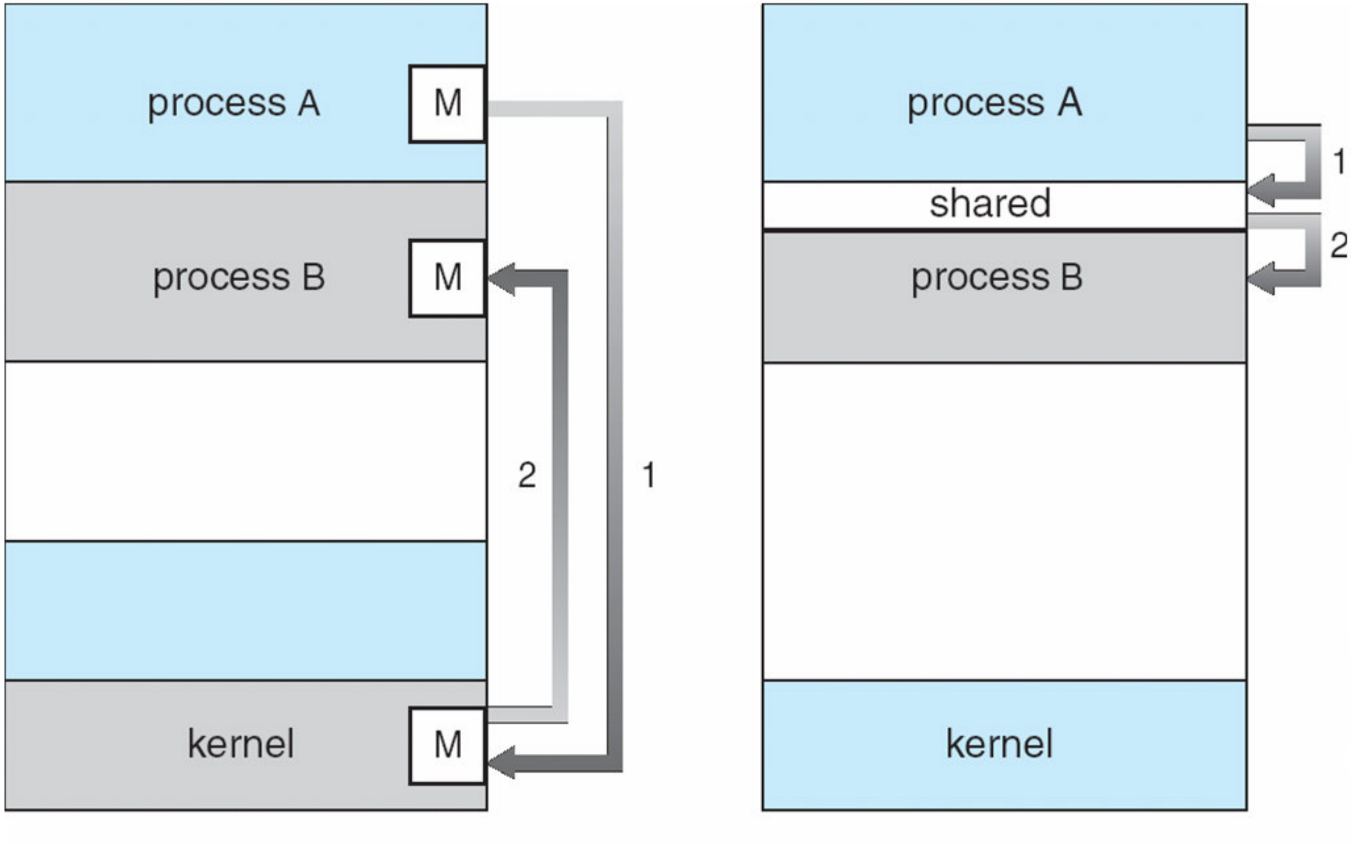
```
scheduler(void)
{
  struct proc *p;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      proc = p;
      switchuvm(p);
      p->state = RUNNING;
      swtch(&cpu->scheduler, proc->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      proc = 0;
    }
    release(&ptable.lock);

  }
}
```

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state.
void
sched(void)
{
  int intena;

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(cpu->ncli != 1)
    panic("sched locks");
  if(proc->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = cpu->intena;
  swtch(&proc->context, cpu->scheduler);
  cpu->intena = intena;
}
```

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



(a)

(b)

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.

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);
```

Shared Memory Control

- The shmctl system call permits the user to perform a number of generalized control operations on an existing shared memory segment
  - The first, shmid, is a valid shared memory segment identifier generated by a prior shmget system call.
  - The second argument, cmd, specifies the operation shmctl is to perform.
  - The third argument, buf, is a reference to a structure of the type shmid_ds.

Using a File as Shared Memory

- mmap system call can be used to map a file to a process's virtual memory address space.
- In many ways mmap is more flexible than its shared memory system call counterpart.
- Once a mapping has been established, standard system calls rather than specialized system calls can be used to manipulate the shared memory object.
- Unlike memory, the contents of a file are nonvolatile and will remain available even after a system has been shut down (and rebooted).

Interprocess Communication – 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 msg to process P
  - **receive**(*Q, message*) – receive a msg 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

- 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 – WinXP

- Message-passing centric via local procedure call (LPC) facility
  - Only works between processes on the same system
  - Uses ports (like mailboxes) to establish and maintain communication channels

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 BUFFER_SIZE-1 elements

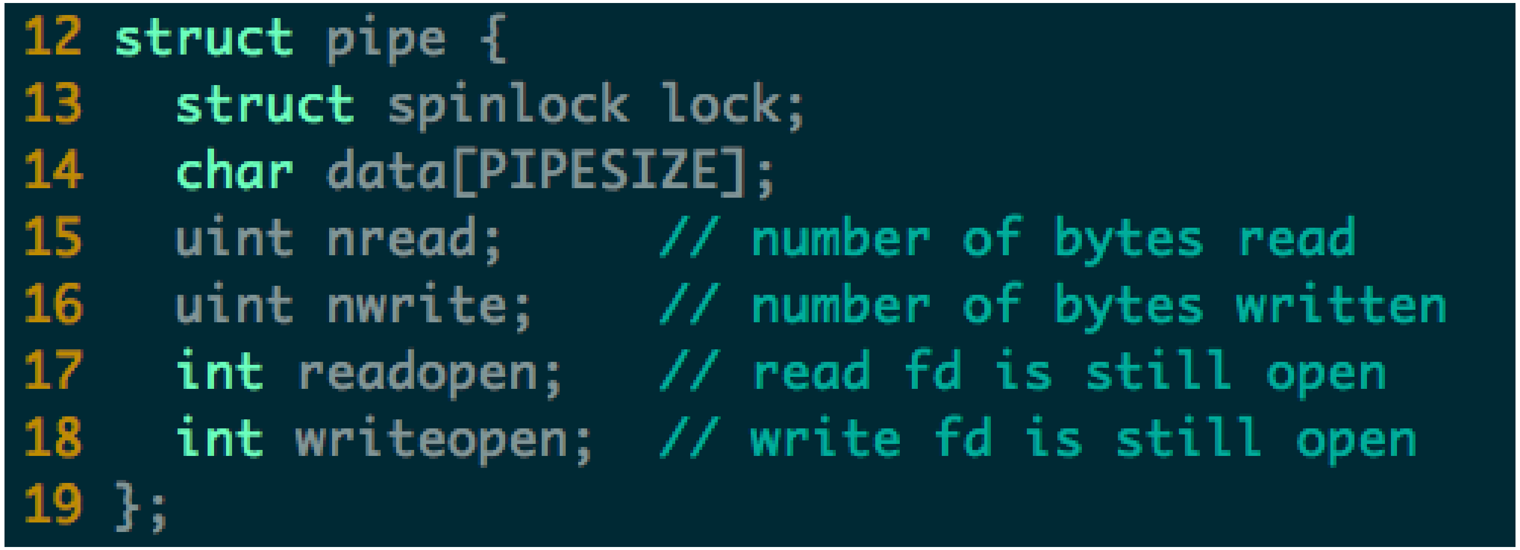
Bounded-Buffer – Producer

```
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;
 }
```

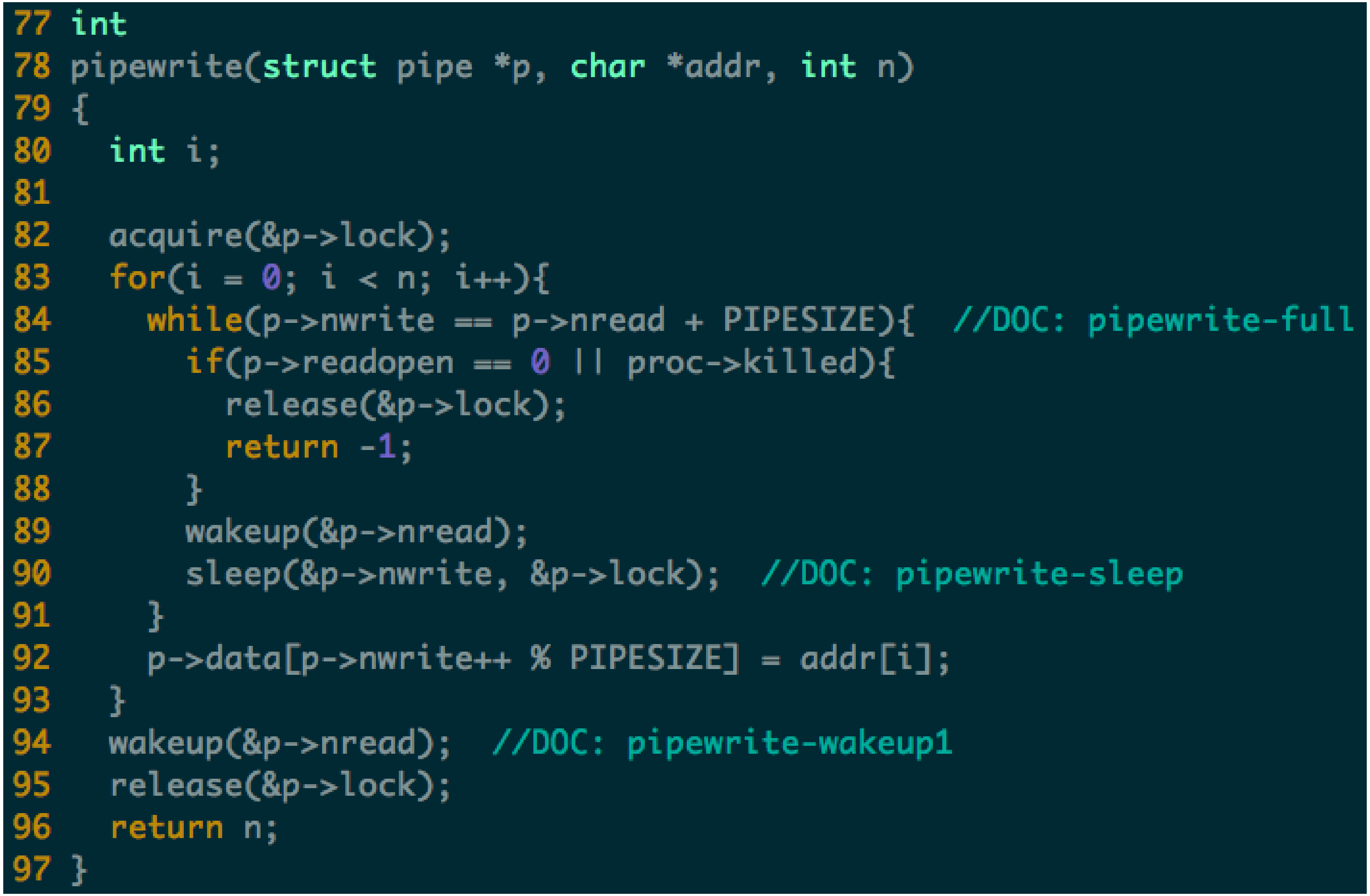
Bounded Buffer – Consumer

```
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;
 }
```

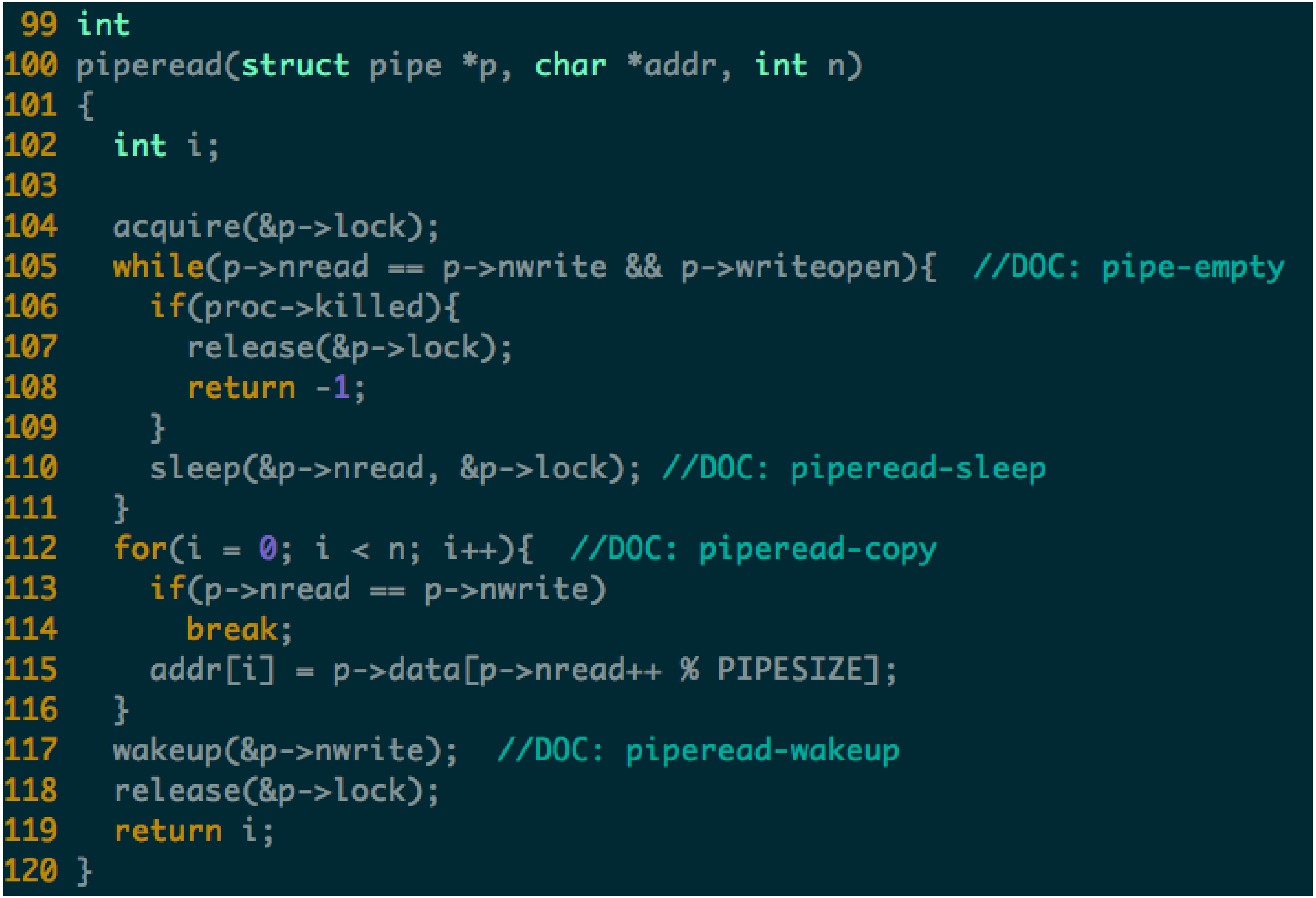
Example of IPC in xv6: Pipe (pipe.c)



```c
struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```



```c
int
pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}
```



```c
int
piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

Communications in Client-Server Systems

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

INTERRUPT, TRAP, SYS CALL

Concepts

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

Requirements

- 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

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 <= 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

Kernel Stack

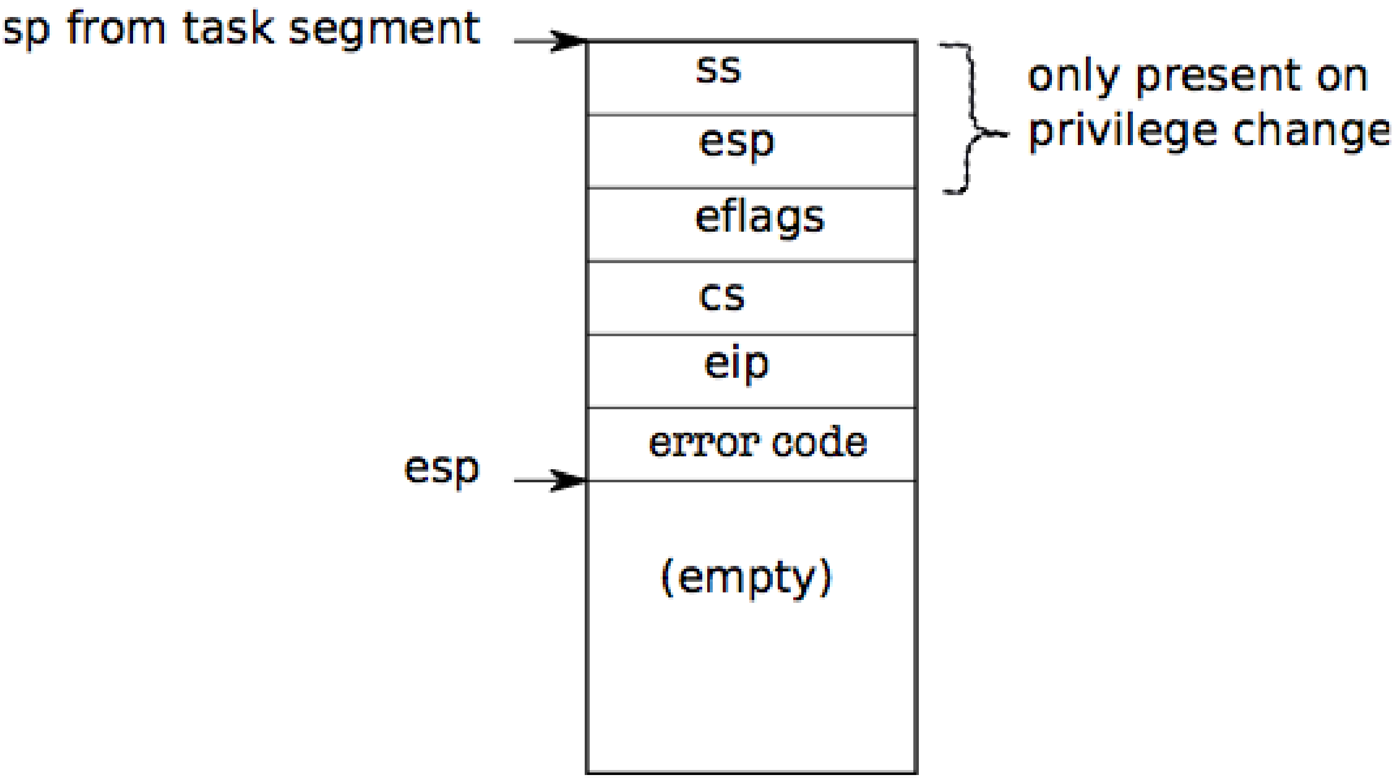


Figure 3-1. Kernel stack after an int instruction.

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)

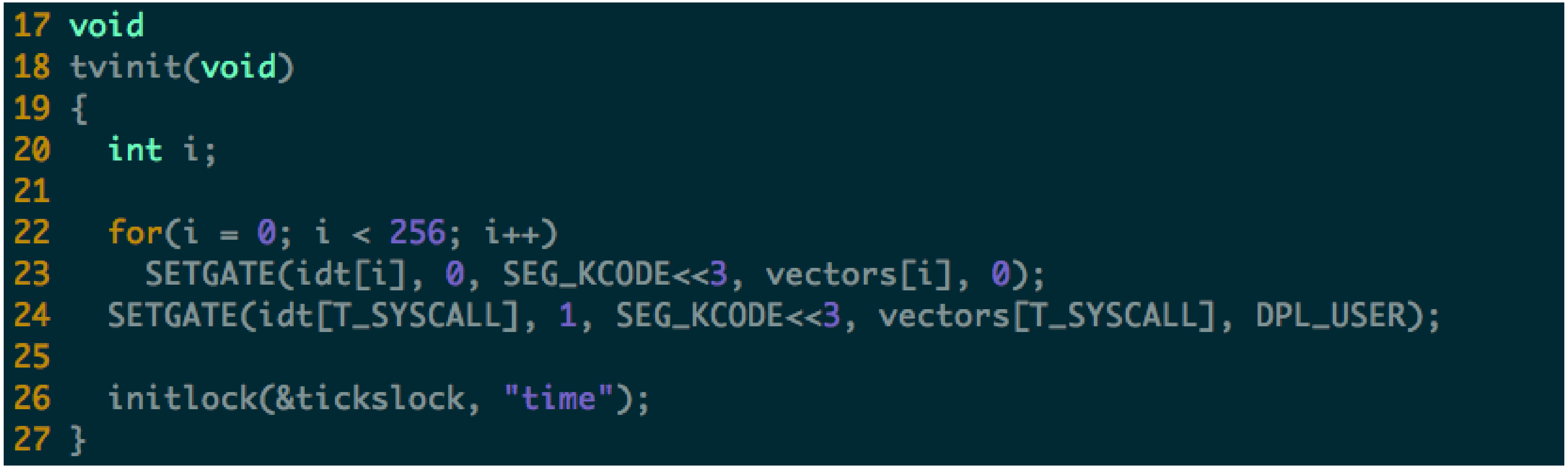
- We must program the device to generate an interrupt, and arrange that a CPU receives the interrupt.

Vectors, IDT

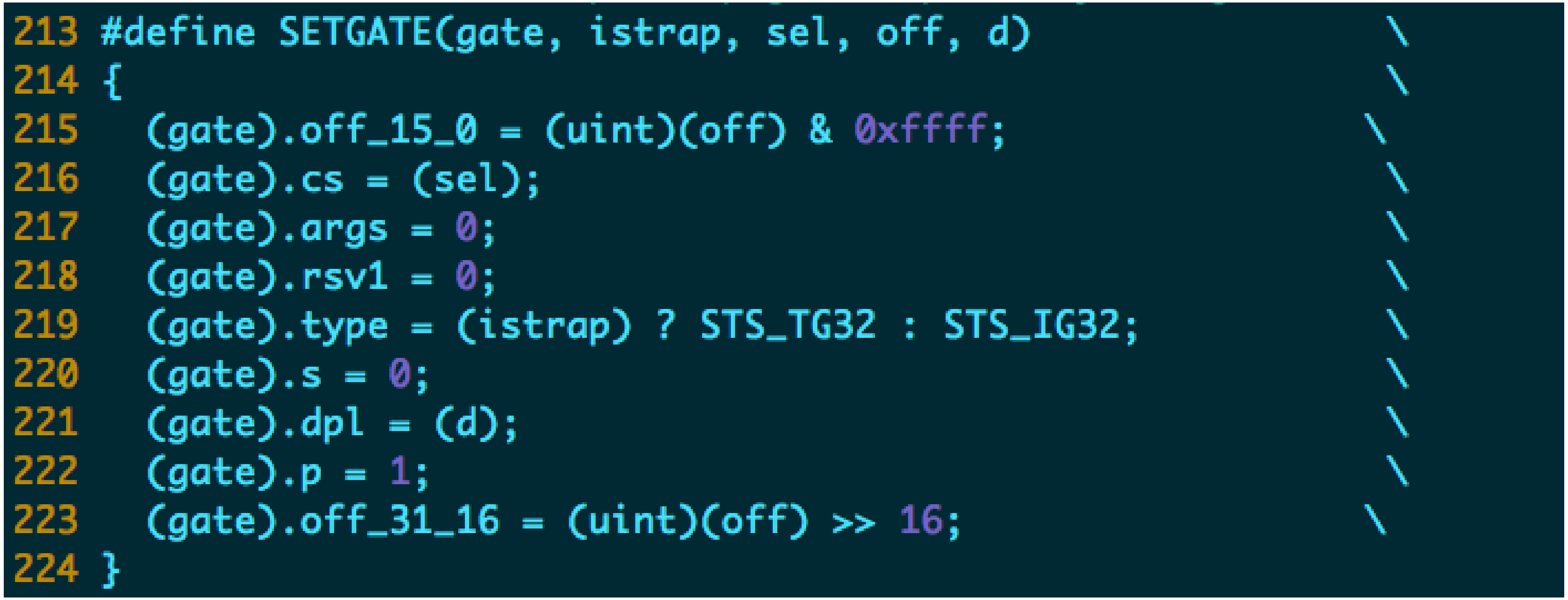
- Vector: index (0-255) into descriptor table (IDT)
  - Special register: idtr points to table (use lidt to load)
- IDT: table of "gate descriptors"
  - Segment selector + offset for handler
  - Descriptor Privilege Level (DPL)
  - Gates (slightly different ways of entering kernel)
    - **Task gate**: includes TSS to transfer to (not used by Linux)
    - **Interrupt gate**: disables further interrupts
    - **Trap gate**: further interrupts still allowed
- Vector assignments
  - Exceptions, NMI are fixed
  - Maskable interrupts can be assigned as needed

Gates

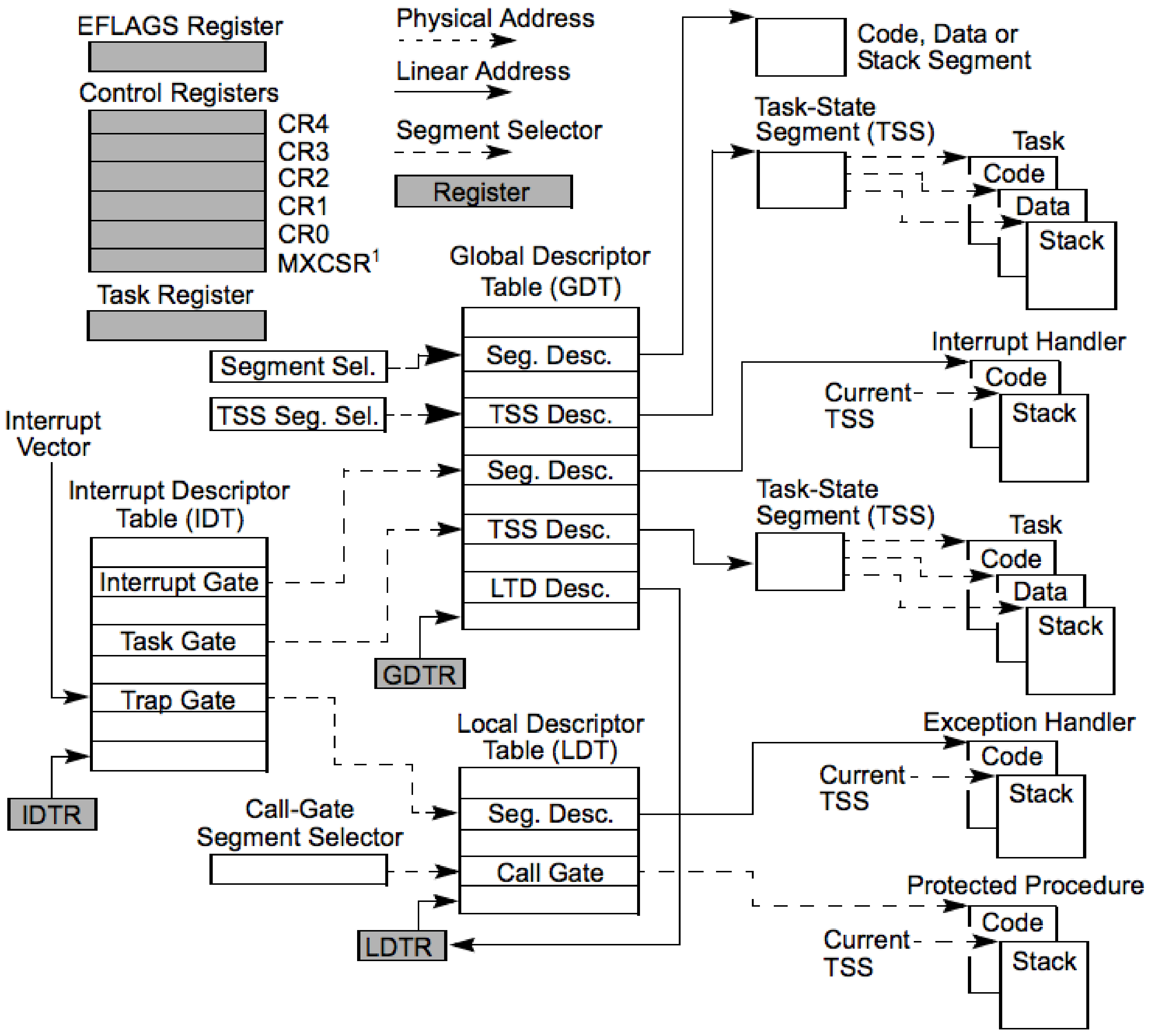
- Interrupt gate
  - E.g., vector 32 in XV6: interrupt
  - Clears IF, so no recursive interrupts
- Trap gate
  - E.g., vector 64 in XV6: system call
  - Interrupt
- Task Gate
  - Includes TSS to transfer to (not used by Linux)



```c
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



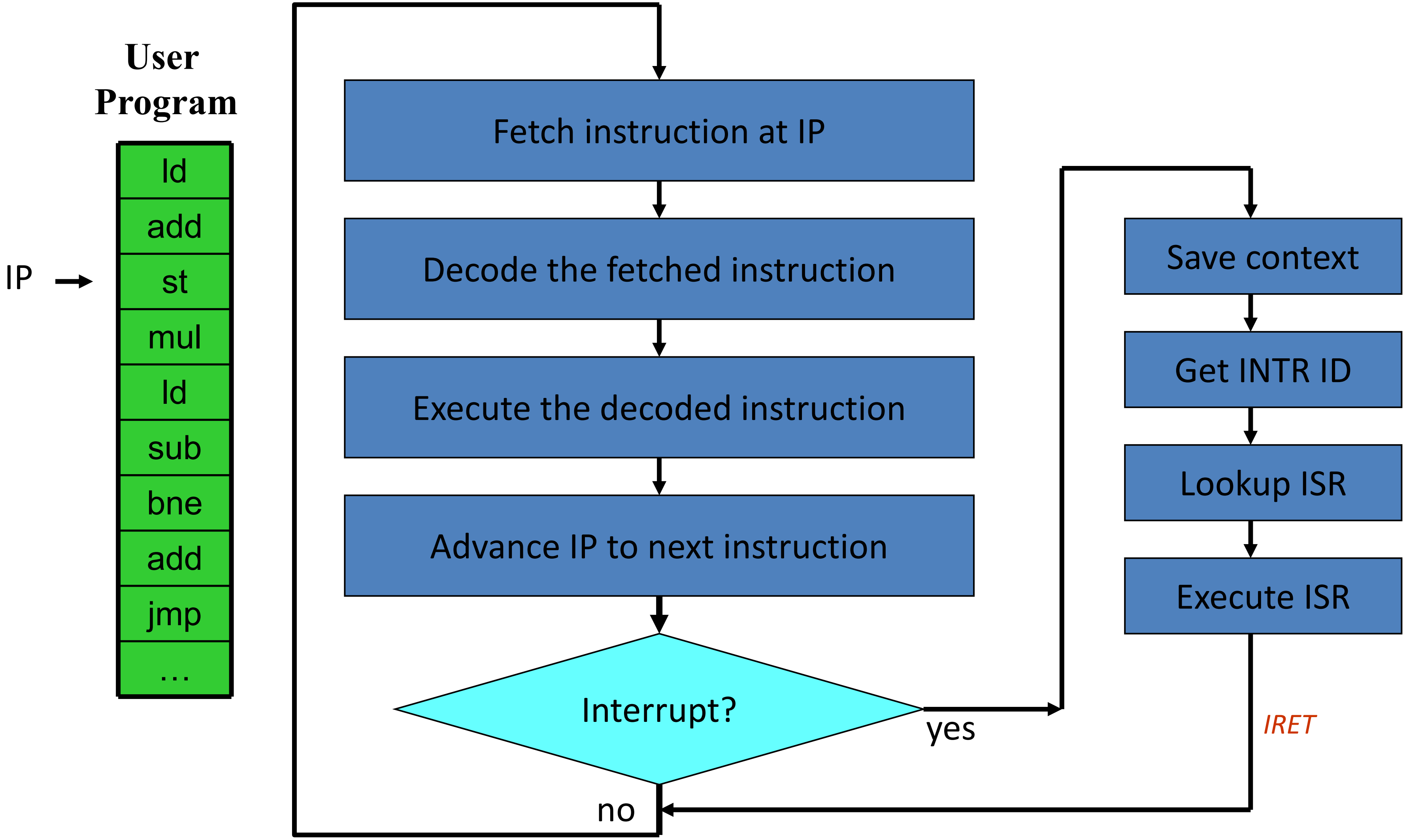
```c
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}
```

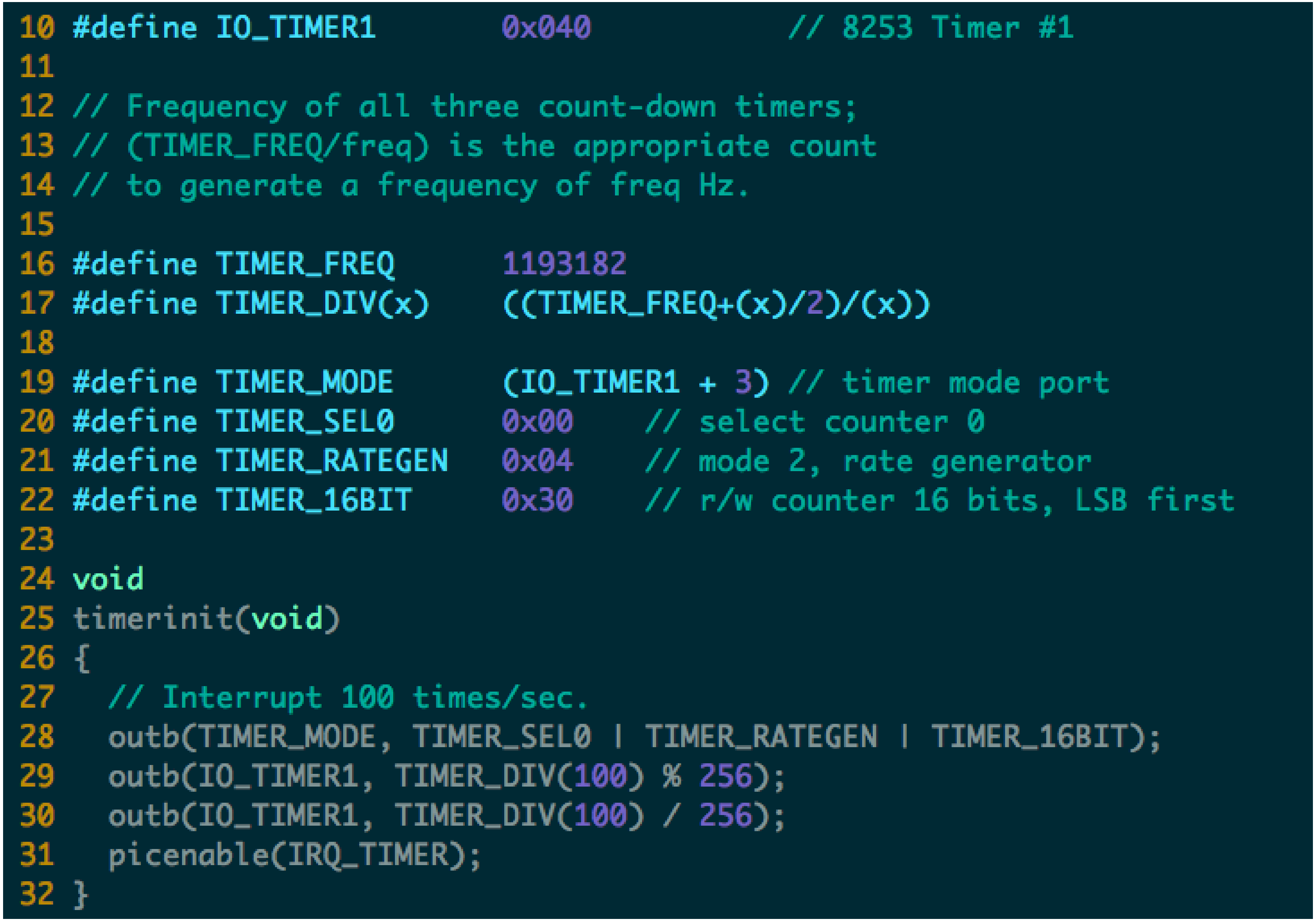

EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
MXCSR[1]
Task Register
Physical Address
Linear Address
Segment Selector
Register
Code, Data or
Stack Segment
Task-State
Segment (TSS)
Task
Code
Data
Stack
Global Descriptor
Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LTD Desc.
GDTR
Interrupt Handler
Code
Current
TSS
Stack
Task-State
Segment (TSS)
Task
Code
Data
Stack
Interrupt
Vector
Interrupt Descriptor
Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
IDTR
Exception Handler
Code
Current
TSS
Stack
Local Descriptor
Table (LDT)
Seg. Desc.
Call Gate
Call-Gate
Segment Selector
LDTR
Protected Procedure
Code
Current
TSS
Stack

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

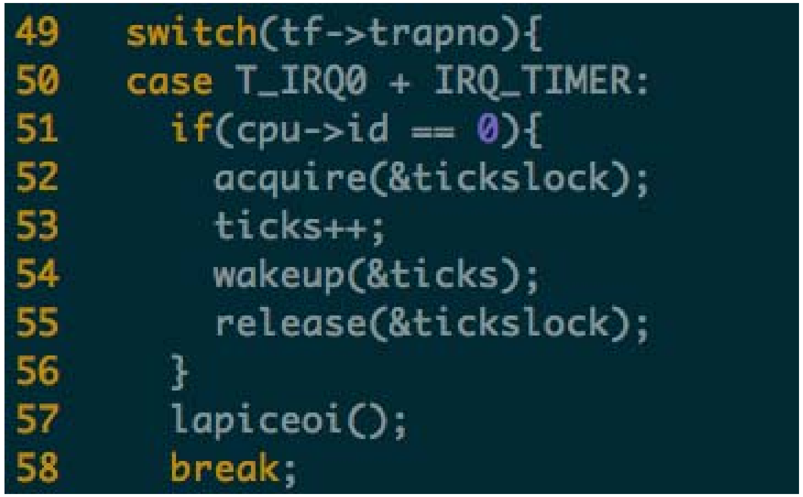
CPU’s ‘fetch-execute’ cycle



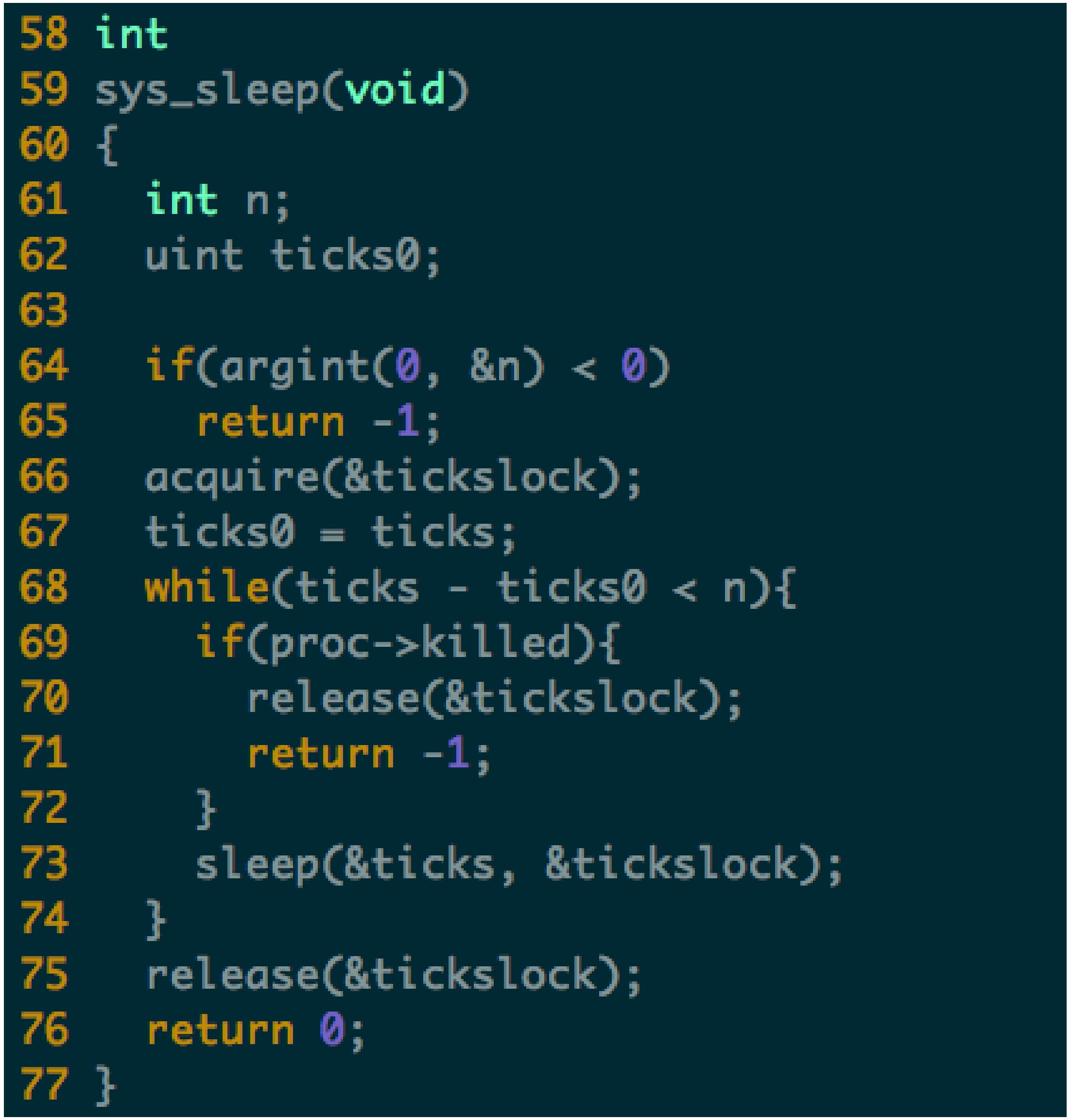


```c
#define IO_TIMER1       0x040           // 8253 Timer #1

// Frequency of all three count-down timers;
// (TIMER_FREQ/freq) is the appropriate count
// to generate a frequency of freq Hz.

#define TIMER_FREQ      1193182
#define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))

#define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
#define TIMER_SEL0      0x00    // select counter 0
#define TIMER_RATEGEN   0x04    // mode 2, rate generator
#define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first

void
timerinit(void)
{
  // Interrupt 100 times/sec.
  outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
  outb(IO_TIMER1, TIMER_DIV(100) % 256);
  outb(IO_TIMER1, TIMER_DIV(100) / 256);
  picenable(IRQ_TIMER);
}
```

Timer Interrupt



```
switch(tf->trapno){
case T_IRQ0 + IRQ_TIMER:
  if(cpu->id == 0){
    acquire(&tickslock);
    ticks++;
    wakeup(&ticks);
    release(&tickslock);
  }
  lapiceoi();
  break;
```

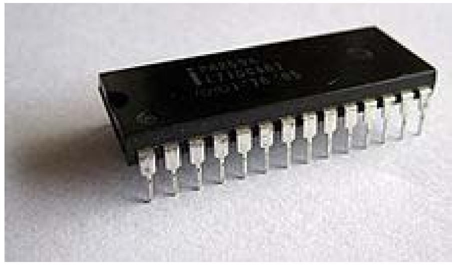


```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

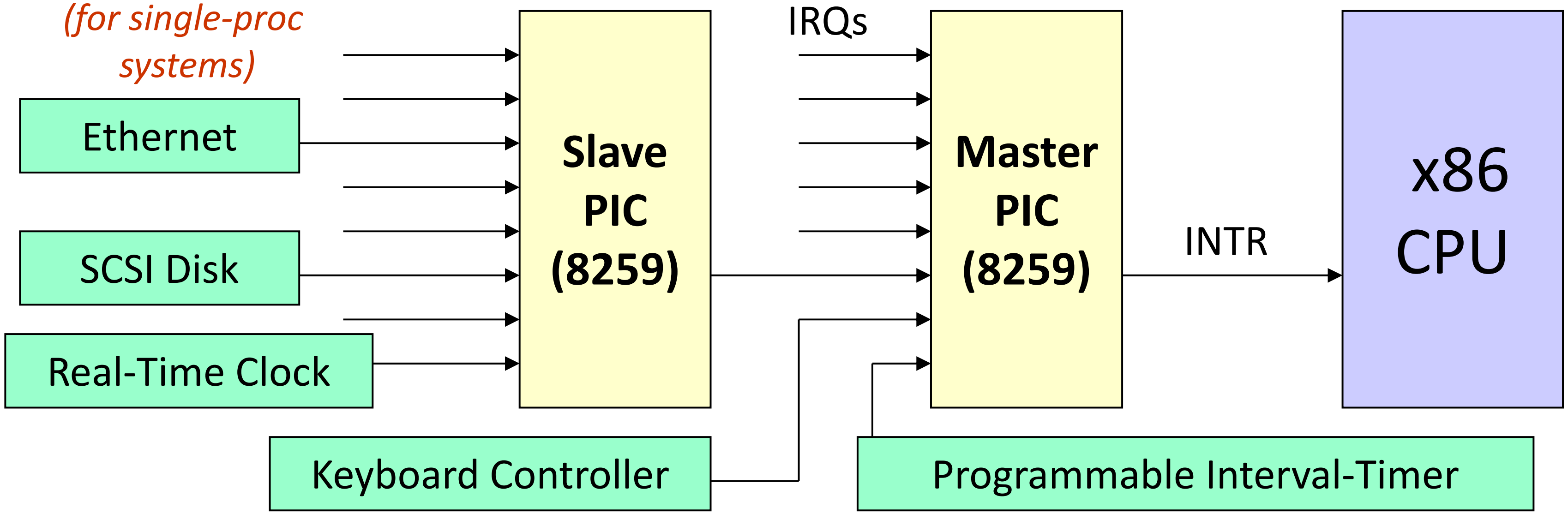
Early Boards

- 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
- XV6 works correctly on uni-processor

Interrupt Hardware

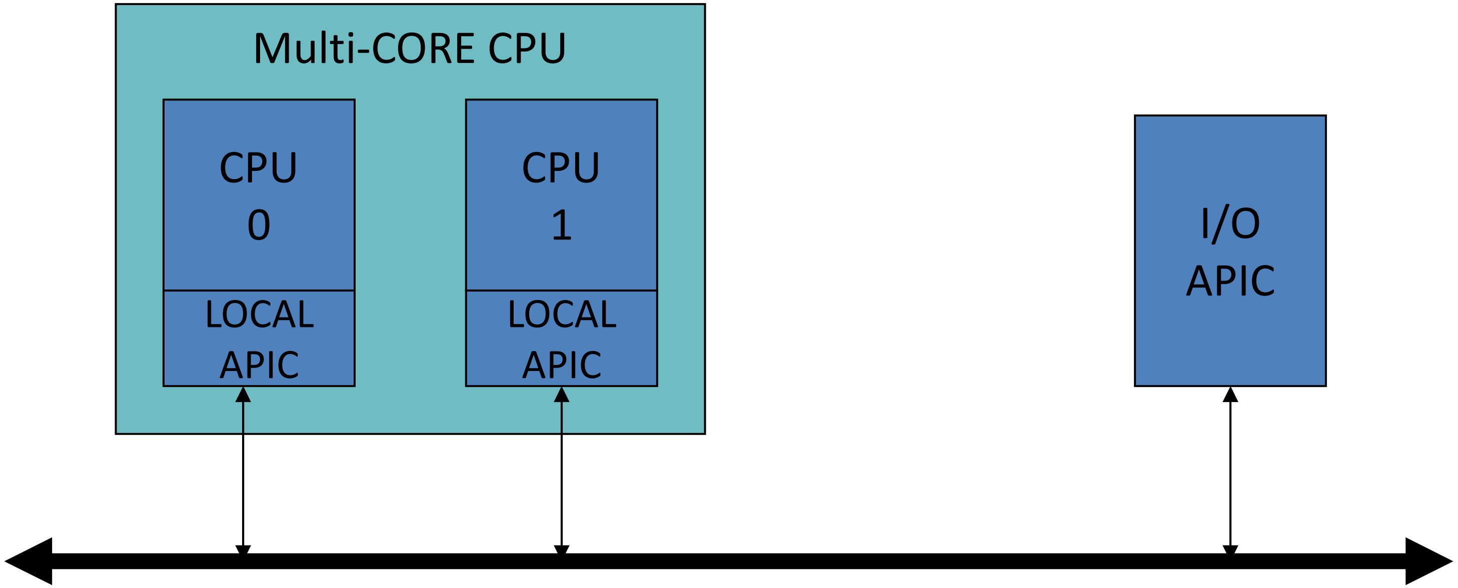


Legacy PC Design (for single-proc systems)



- 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)

Multiple Logical 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)

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
  - 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

Multi-processor PC Boards

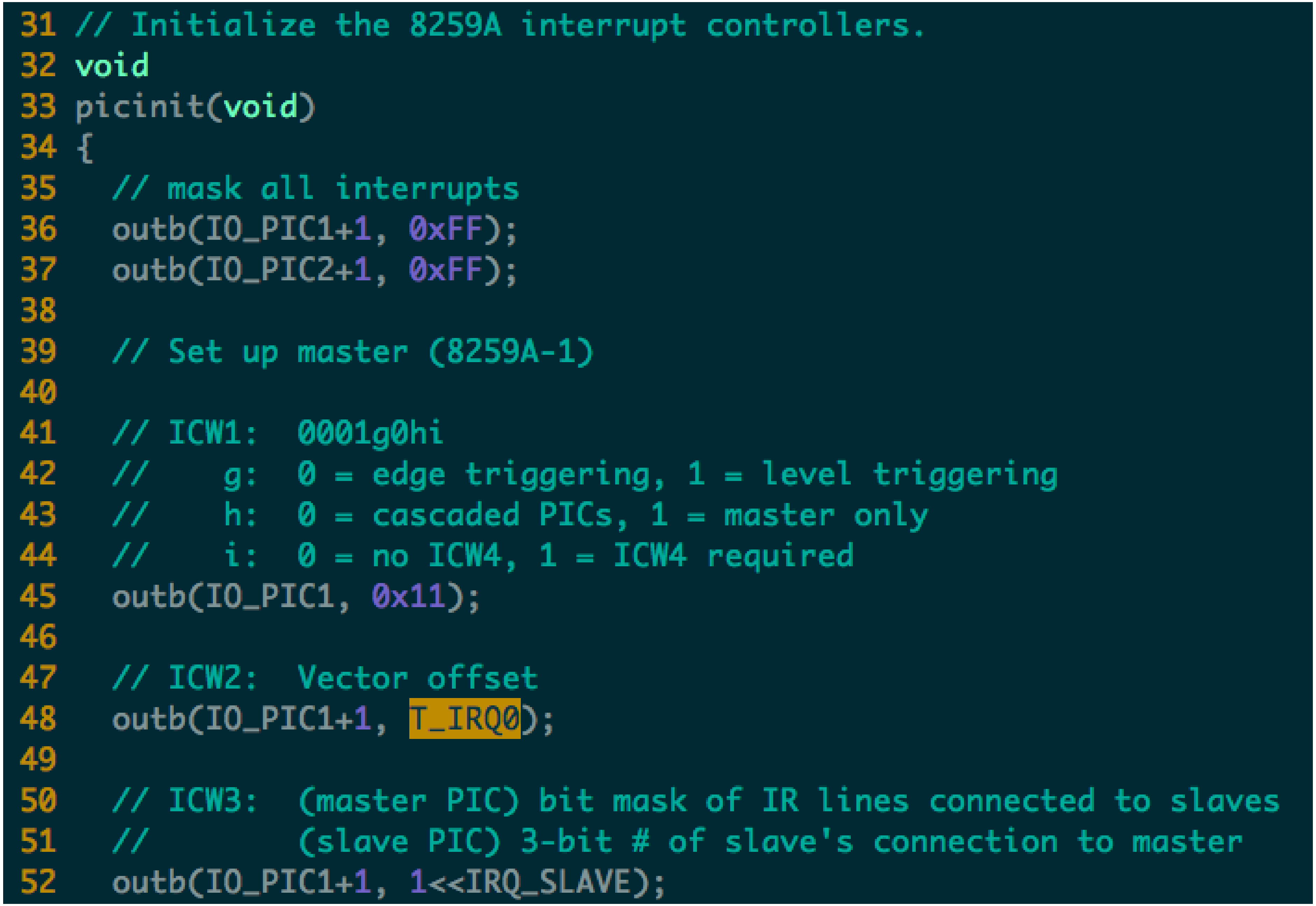
- Two parts in XV6
  - IO APIC, for the I/O system
    - ioapic.c
  - Local APIC, for each processor
    - lapic.c
- XV6 is designed for multi-processor
  - Each processor must be programmed to receive interrupts

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

Assigning Vectors to IRQs

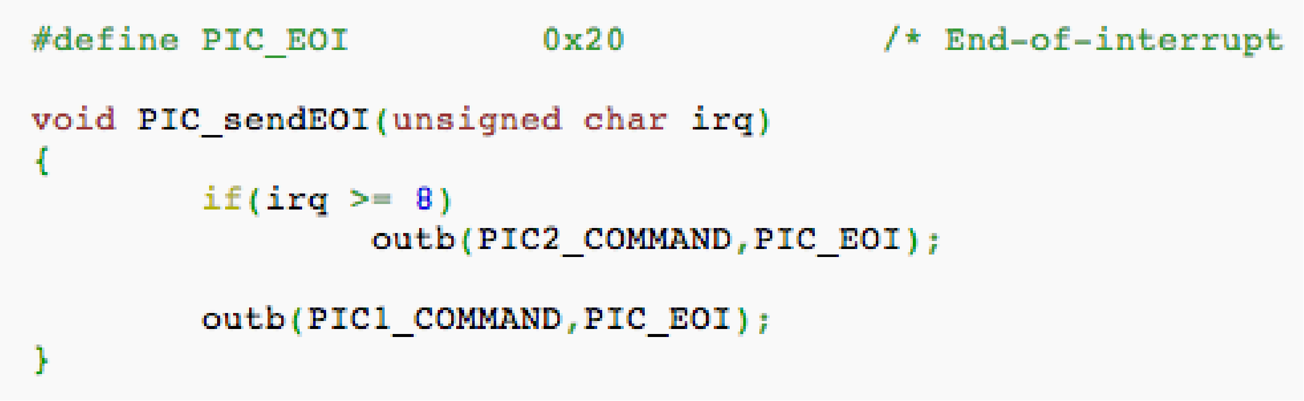
- Vector: index (0-255) into interrupt descriptor table
- Vectors usually IRQ# + 32
  - Below 32 reserved for non-maskable intr & exceptions
  - Maskable interrupts can be assigned as needed
  - Vector 128 used for syscall
  - Vectors 251-255 used for IPI



```c
// Initialize the 8259A interrupt controllers.
void
picinit(void)
{
  // mask all interrupts
  outb(IO_PIC1+1, 0xFF);
  outb(IO_PIC2+1, 0xFF);

  // Set up master (8259A-1)

  // ICW1:  0001g0hi
  //    g:  0 = edge triggering, 1 = level triggering
  //    h:  0 = cascaded PICs, 1 = master only
  //    i:  0 = no ICW4, 1 = ICW4 required
  outb(IO_PIC1, 0x11);

  // ICW2:  Vector offset
  outb(IO_PIC1+1, T_IRQ0);

  // ICW3:  (master PIC) bit mask of IR lines connected to slaves
  //        (slave PIC) 3-bit # of slave's connection to master
  outb(IO_PIC1+1, 1<<IRQ_SLAVE);
```

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 excuting?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximizing Parallelism

- You want to 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 as soon as possible
- Most devices don’t interfere with each others’ data structures; there’s 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 isn’t 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

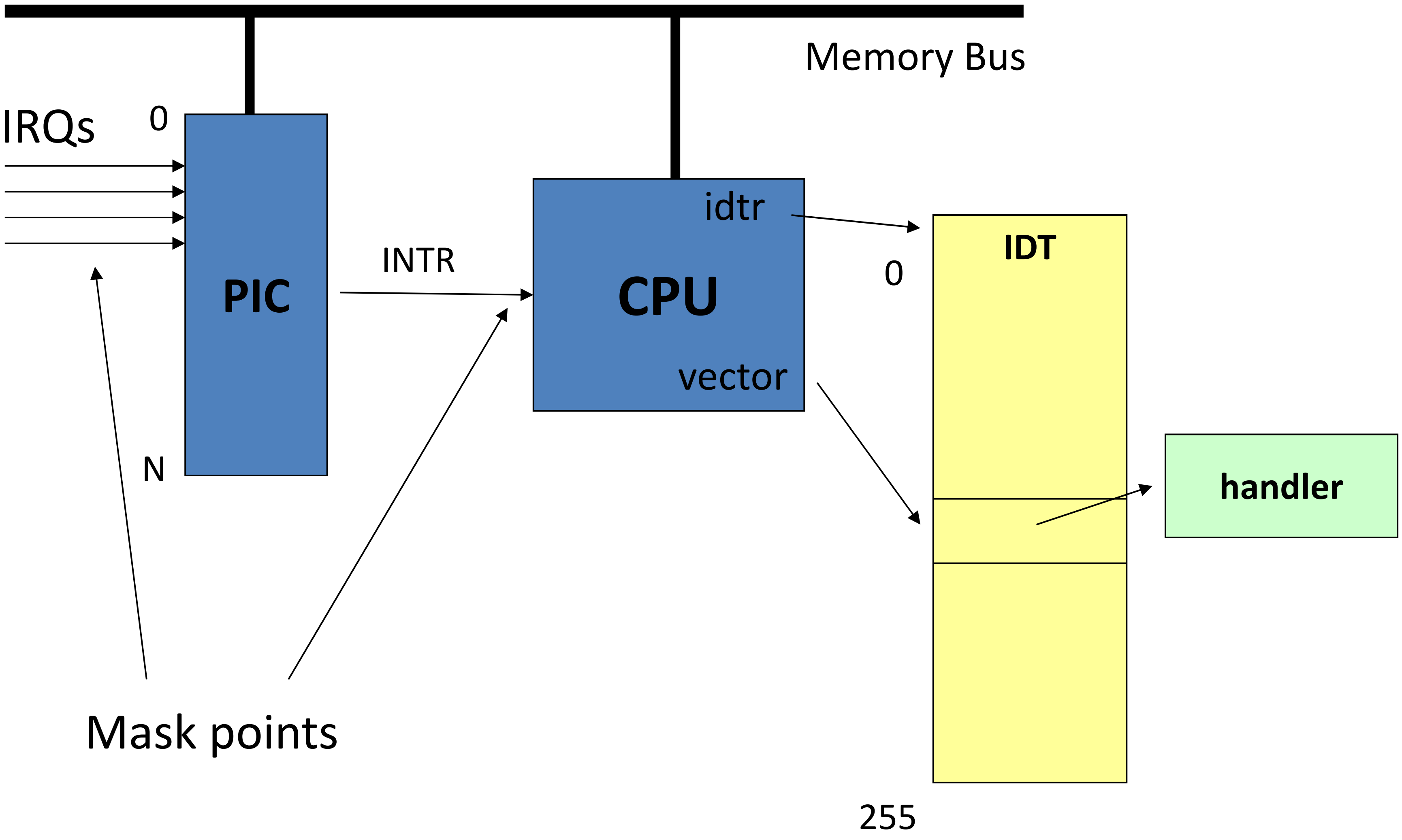
- Interrupts can be interrupted
  - By different interrupts; handlers need not be reentrant
  - Small portions execute with interrupts disabled
  - Interrupts remain pending until acked by CPU
- Exceptions can be interrupted
  - By interrupts (devices needing service)
- Exceptions can nest two levels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

Interrupt Masking

- Two different types: global and per-IRQ
- Global — delays all interrupts
- Selective — individual IRQs can be masked selectively
- Selective masking is usually what’s needed — interference most common from two interrupts of the same type

- NMI (Non-Maskable Interrupt)

Putting It All Together



/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:

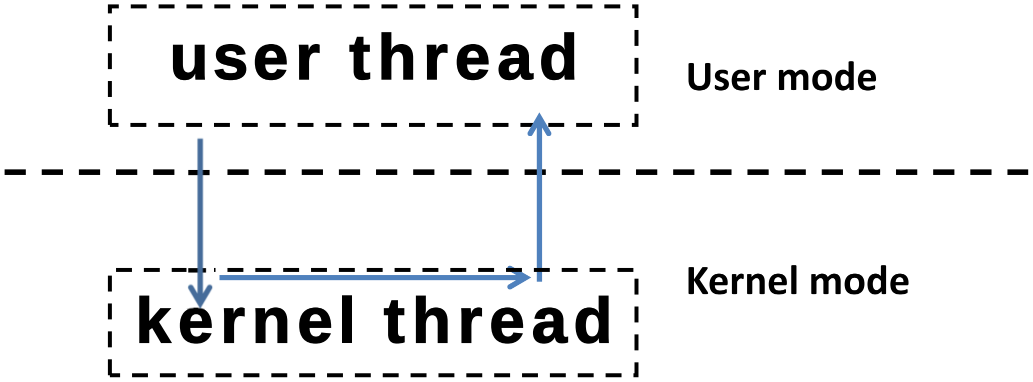
```
$ cat /proc/pci
PCI devices found:
  Bus  0, device   0, function  0:
    Host bridge: PCI device 8086:2550 (Intel Corp.) (rev 3).
      Prefetchable 32 bit memory at 0xe8000000 [0xebffffff].
  Bus  0, device  29, function  1:
    USB Controller: Intel Corp. 82801DB USB (Hub #2) (rev 2).
      IRQ 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].
```

Summary: Interrupts vs Exceptions

- Varying terminology but for Intel:
  - **Interrupt** (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)
    - 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)

Linux System Call

- One system call will cause two context switches between User and Kernel



Questions

1) How can this mode reduce the frequency of mode switching? (2’)

2) The simplest way to implement this mode is use one kernel thread to process all system calls. Describe how it works on a single core machine. (Hint: how do the user threads and the kernel thread cooperate?) (6’)

3) What is the drawback of using just one kernel thread? (2’)

Questions

4) How to mitigate this drawback? (3’) (Please provide the details)

5) In a multi-core machine, we can dedicate one core to only run the kernel threads and the rest cores to run user threads. Please describe the pros and cons. (3’)

Flexible System Call

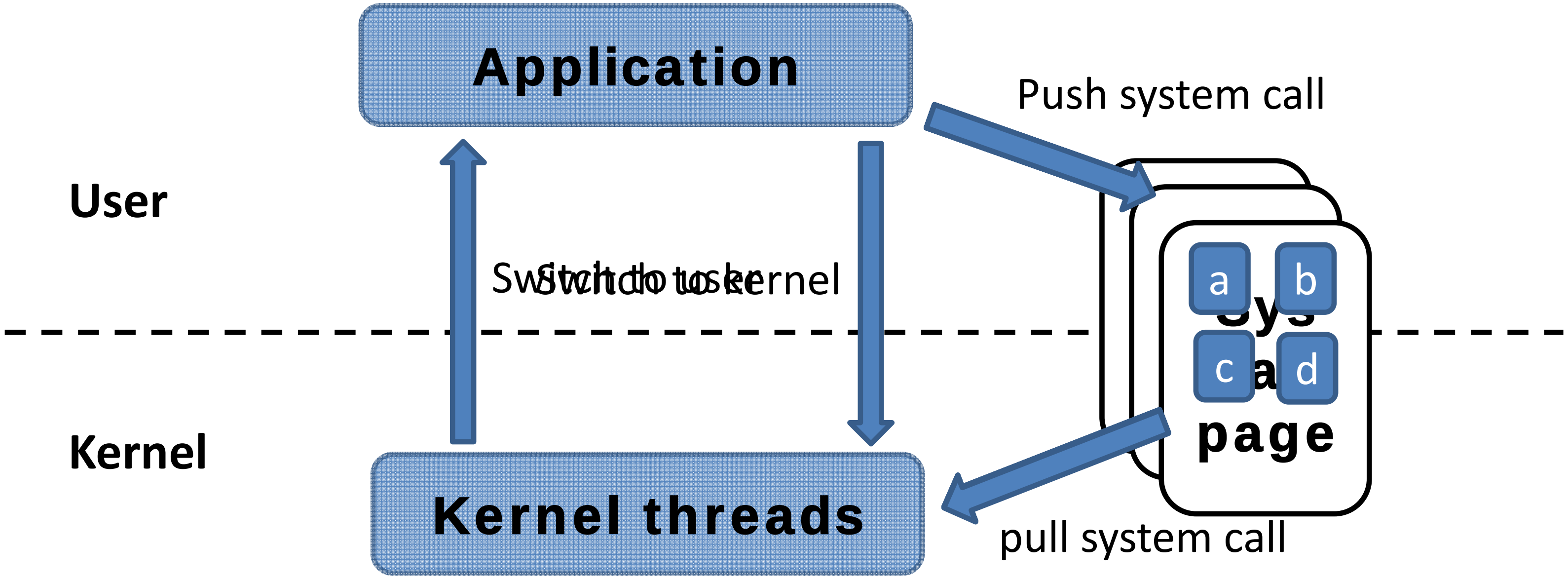
- Problem 2-2
  - New system call mechanism – Flexible System Call
    - no direct mapping between user threads and kernel threads
    - *system call page*
    - User threads can push the system call requests into the system call page
    - kernel threads will poll the system call requests out the system call page

Flexible System Call

- Basic Idea
  - “Flexible System Call Scheduling with Exception-Less System Calls”
  - University of Toronto
  - OSDI 2010

Flexible System Call

- Basic Idea

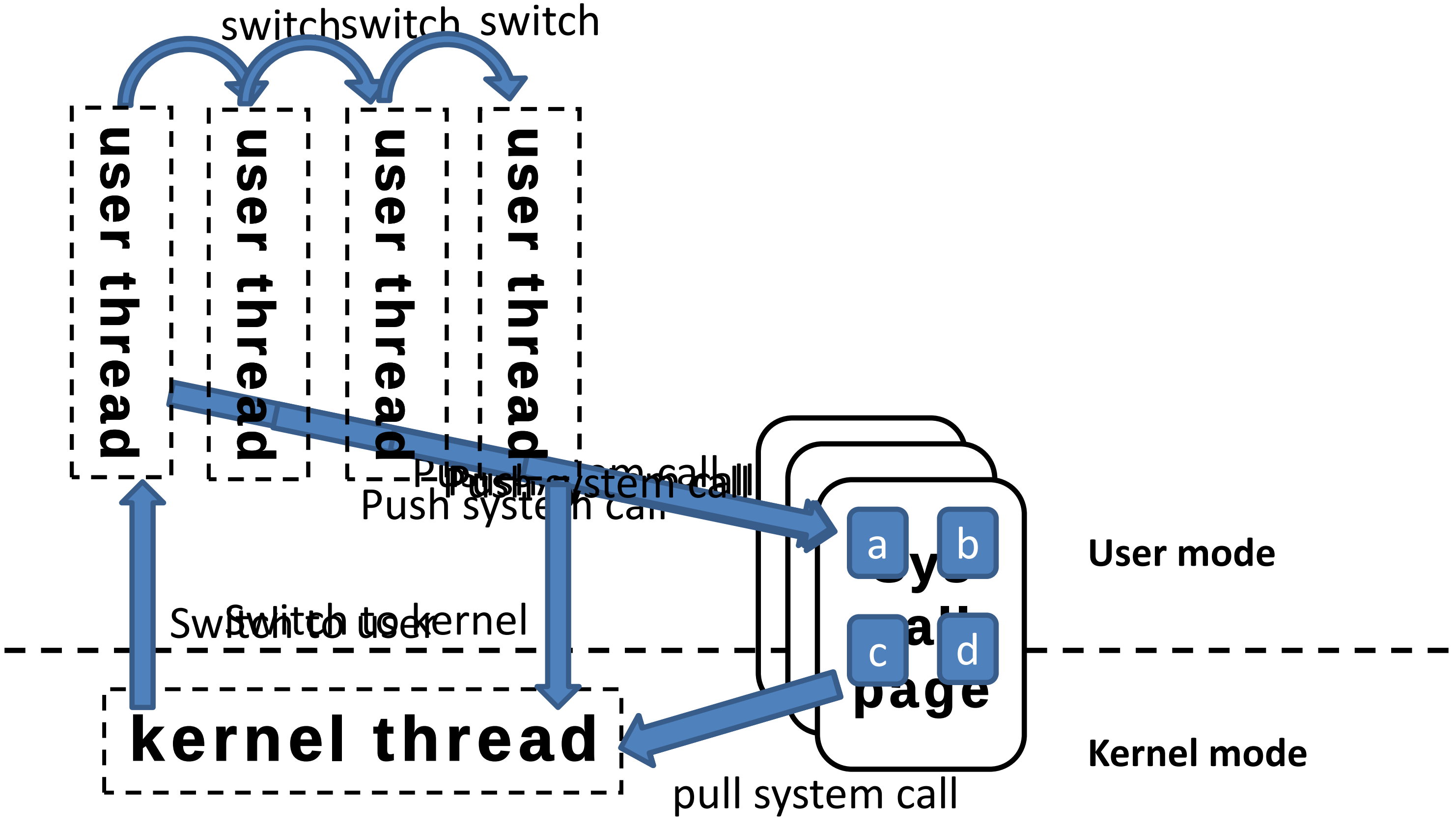


Problem 2-2.1

- How can this mode reduce the frequency of mode switching?
  - Batching system calls using system call page
  - During one switch-around
    - Kernel threads can process multiple system call requests together

Problem 2-2.2

- How it works in a single core machine?



Problem 2-2.3

- What is the drawback of using just one kernel thread?
  - Blocked system calls (e.g. read from disk file) can block the kernel thread
  - It will block the whole machine (not the process)

Problem 2-2.4

- How to mitigate this drawback?
  - We can use multiple kernel threads.
  - When one kernel thread is blocked schedule another to continue processing the left system call requests

Problem 2-2.5

- In a multi-core machine, we can dedicate one core to only run the kernel threads and the rest cores to run user threads. Please describe the pros and cons

Problem 2-2.5-Sol

- Pros:
  - No need to mode switching
  - Most of kernel data/code will not be shared across cores
- Cons:
  - Remote data exchanges between user/kernel threads are expensive (cross cores data access)
  - Kernel thread core maybe overloaded
  - Kernel thread core maybe idle

Interrupt, Trap, SYS CALL

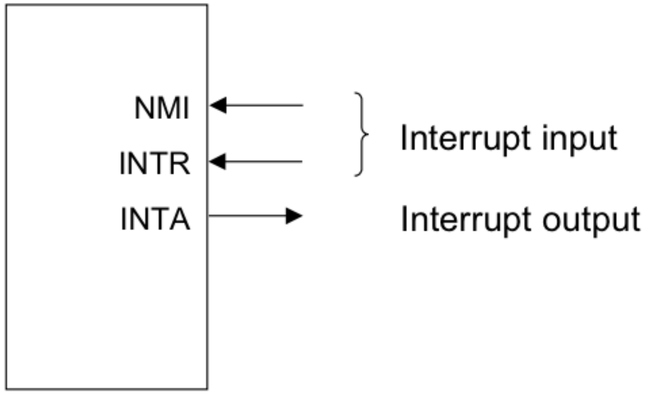
Haibo Chen, Yubin Xia

IPADS, SJTU

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

Interrupt Sources

1. **Hardware interrupt:** external
   - *Nonmaskable interrupt* (NMI) input pin
   - *Interrupt* (INTR) input pin



1. **Software interrupt:** execution of the Interrupt instruction, **INT**
2. **Error condition:** If some error condition occur by the execution of an instruction. E.g.,
   - *divide-by-zero interrupt*: If u attempt to divide an operand by zero, the 8086 will automatically interrupt the currently executing program

Terms

- Vector, Interrupt vector, Trap number
- IRQ: Interrupt Request
- Soft IRQ: A mechanism to implement bottom half

- Interrupt, trap, fault, exception
- Software interrupt / system call

- IDT: Interrupt Descriptor Table
- ISP: Interrupt Service Procedure

Requirements

- 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

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.

Types of Interrupts

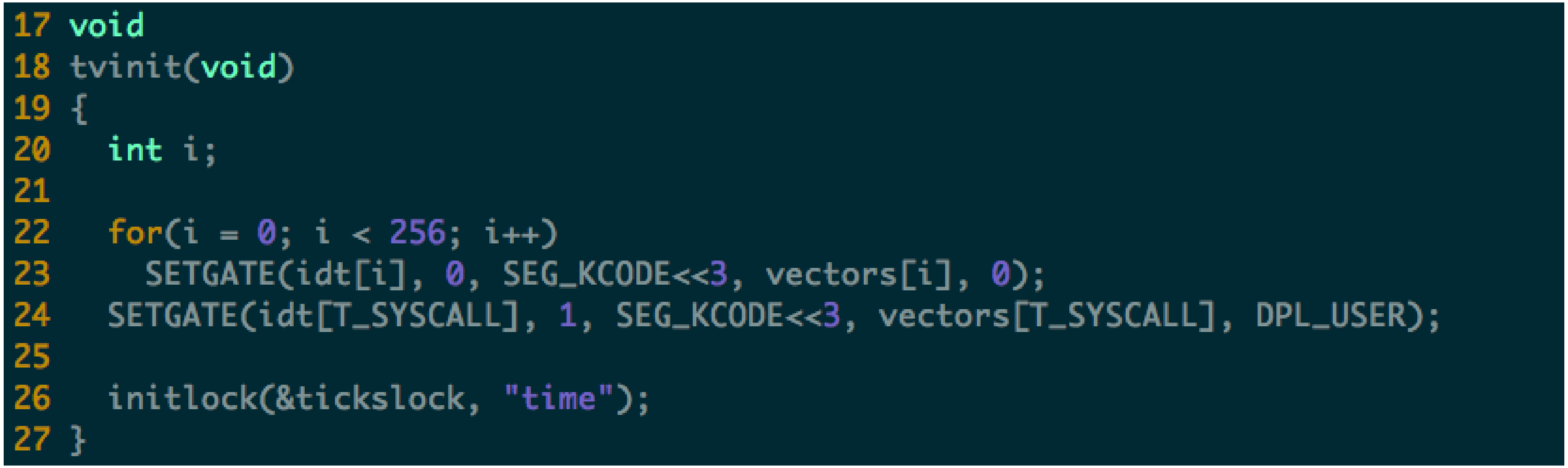
- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed
- Synchronous (also called *exceptions*)
  - *Processor-detected* exceptions:
    - *Faults* — correctable; offending instruction is *retried*
    - *Traps* — often for debugging; instruction is *not* retried
    - *Aborts* — major error (hardware failure)
  - *Programmed* exceptions:
    - Requests for kernel intervention (software interrupt/syscalls)

System Call

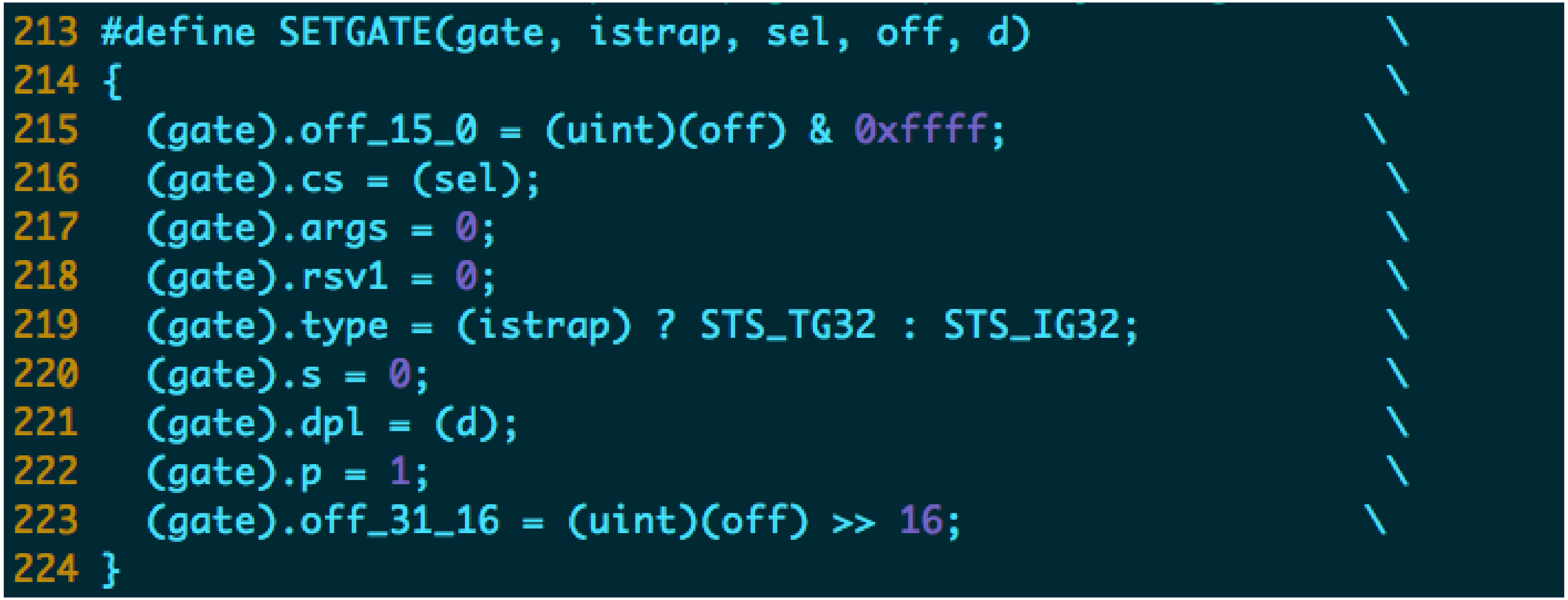
1. Fetch the *n*'th descriptor from the IDT, where *n* is the argument of int.
2. Check that CPL in %cs is <= DPL, where DPL is the privilege level in the descriptor.
3. Save %esp and %ss in a CPU-internal registers, but only if the target segment selector's PL < CPL.
4. Load %ss and %esp from a task 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

Vectors, IDT

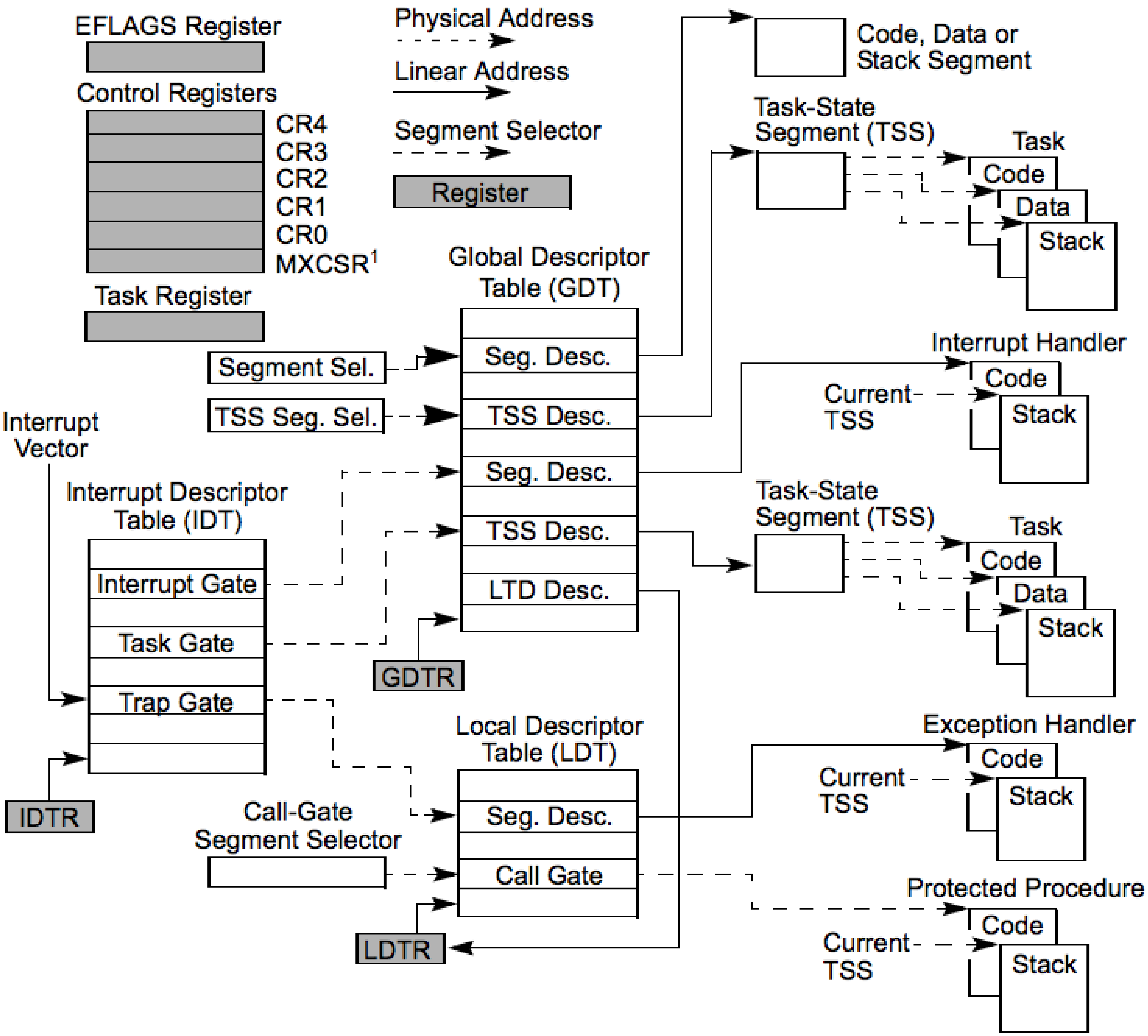
- Vector: index (0-255) into descriptor table (IDT)
  - Special register: idtr points to table (use lidt to load)
- IDT: table of “gate descriptors”
  - Segment selector + offset for handler
  - Descriptor Privilege Level (DPL)
  - Gates (slightly different ways of entering kernel)
    - **Task gate**: includes TSS to transfer to (not used by Linux)
    - **Interrupt gate**: disables further interrupts
    - **Trap gate**: further interrupts still allowed
- Vector assignments
  - Exceptions, NMI are fixed
  - Maskable interrupts can be assigned as needed



```
void
tvinit(void)
{
  int i;

  for(i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);

  initlock(&tickslock, "time");
}
```



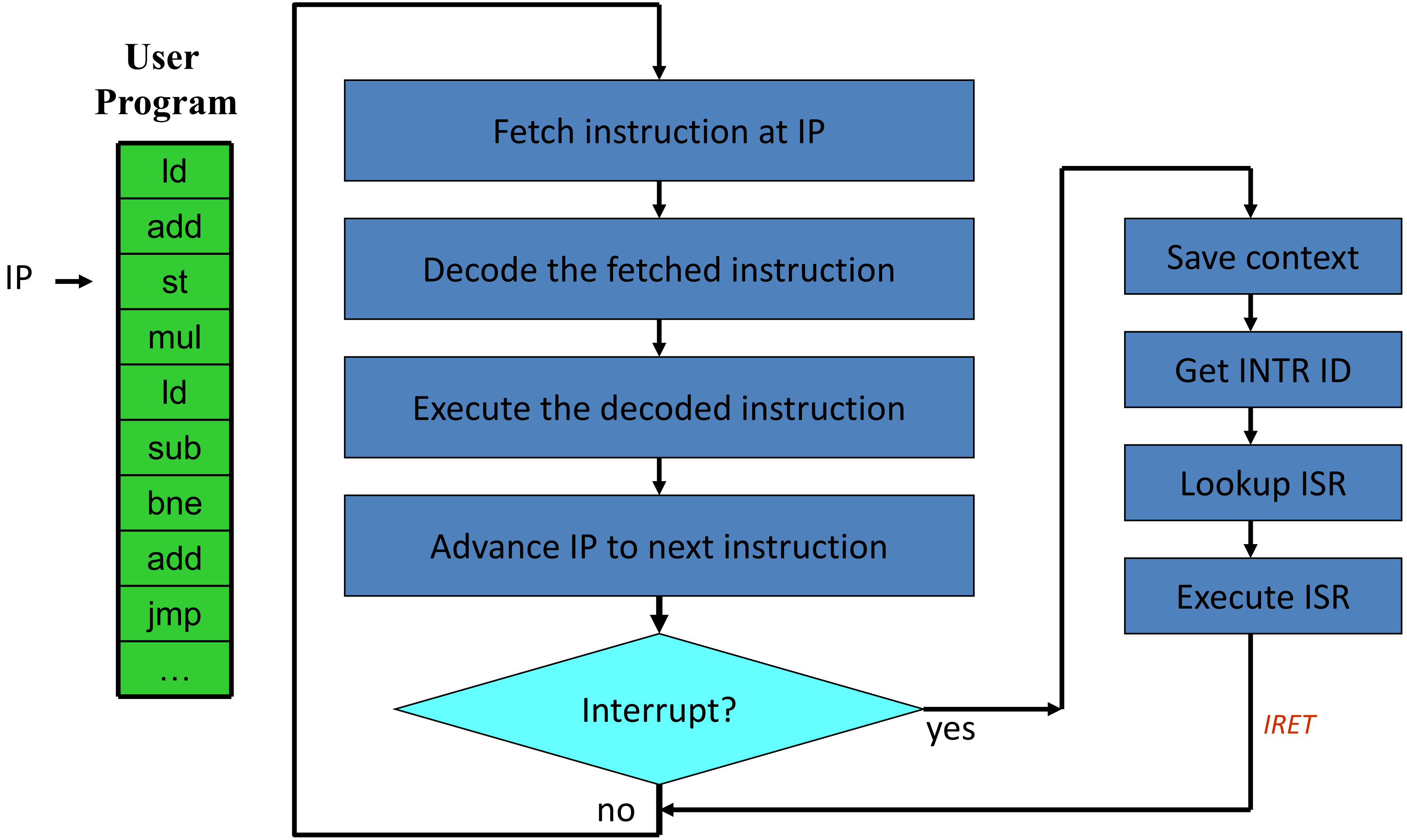
```
#define SETGATE(gate, istrap, sel, off, d)                \
{                                                         \
  (gate).off_15_0 = (uint)(off) & 0xffff;                \
  (gate).cs = (sel);                                      \
  (gate).args = 0;                                        \
  (gate).rsv1 = 0;                                        \
  (gate).type = (istrap) ? STS_TG32 : STS_IG32;           \
  (gate).s = 0;                                           \
  (gate).dpl = (d);                                       \
  (gate).p = 1;                                           \
  (gate).off_31_16 = (uint)(off) >> 16;                  \
}
```


EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
MXCSR[1]
Task Register
Physical Address
Linear Address
Segment Selector
Register
Code, Data or Stack Segment
Task-State Segment (TSS)
Task
Code
Data
Stack
Global Descriptor Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LTD Desc.
GDTR
Interrupt Handler
Code
Current TSS
Stack
Interrupt Vector
Interrupt Descriptor Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
IDTR
Task-State Segment (TSS)
Task
Code
Data
Stack
Local Descriptor Table (LDT)
Seg. Desc.
Call Gate
Call-Gate Segment Selector
LDTR
Exception Handler
Code
Current TSS
Stack
Protected Procedure
Code
Current TSS
Stack

Intel-Reserved ID-Numbers

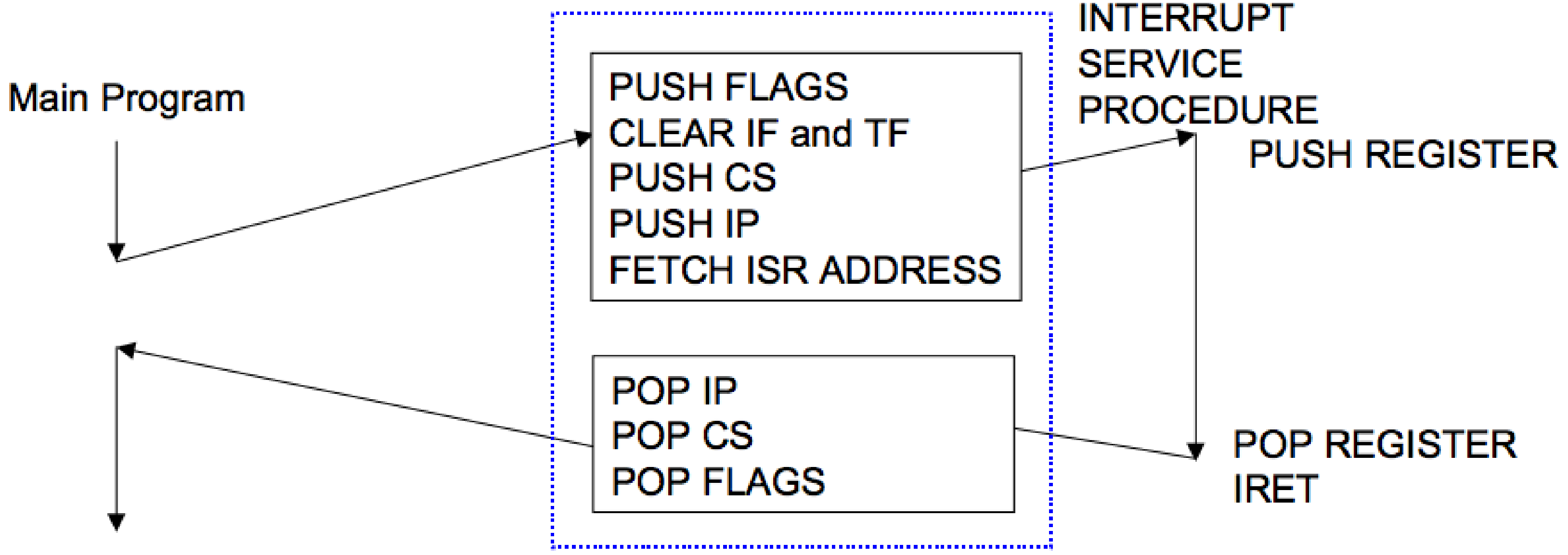
- Of the 256 possible interrupt ID numbers, Intel reserves the first 32 for ‘exceptions’
- OS’s such as Linux are free to use the remaining 224 available interrupt ID numbers for their own purposes (e.g., for service-requests from external devices, or for other purposes such as system-calls)
- Examples:
  - 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

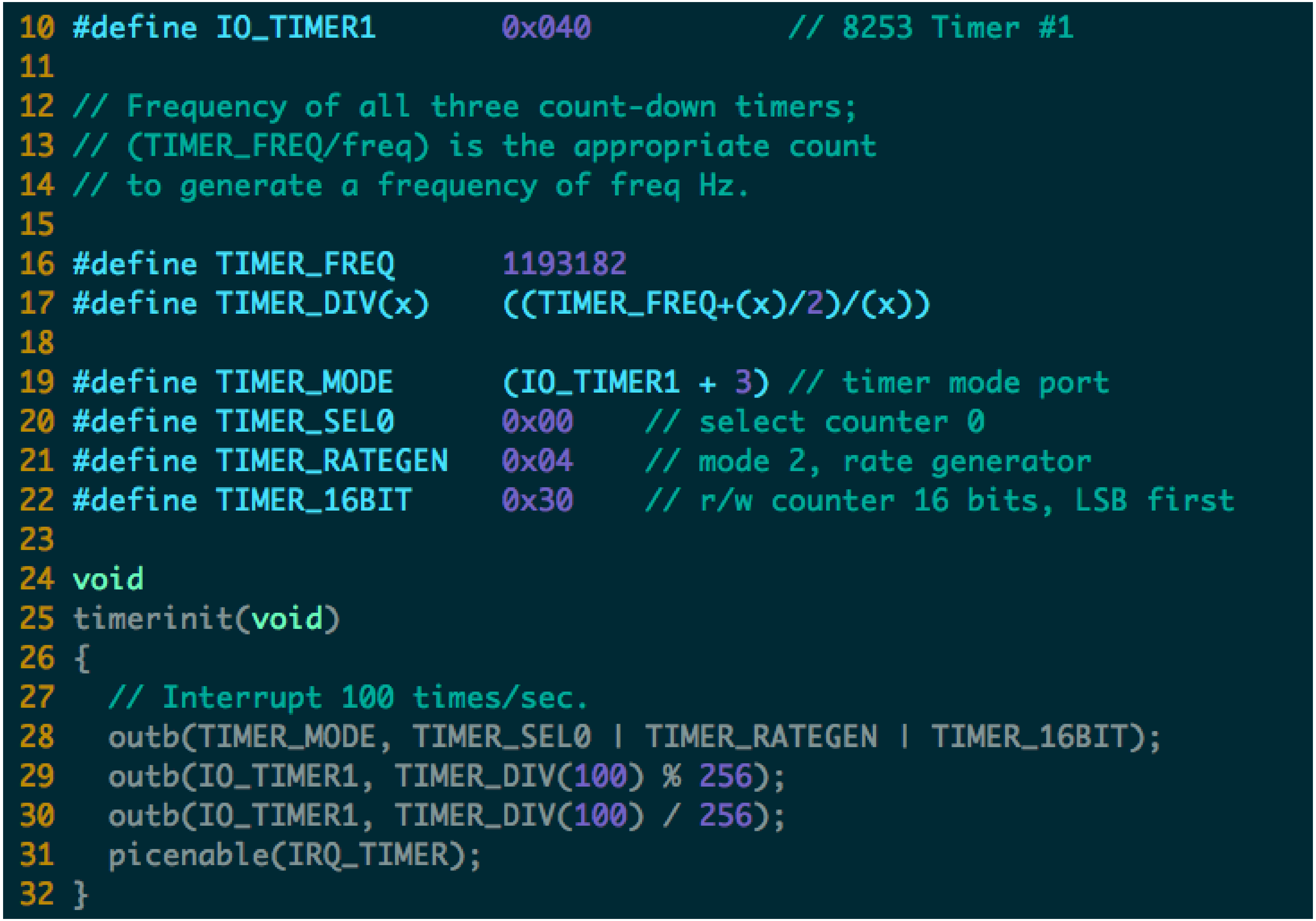
CPU’s ‘fetch-execute’ cycle



Interrupt Response Sequence

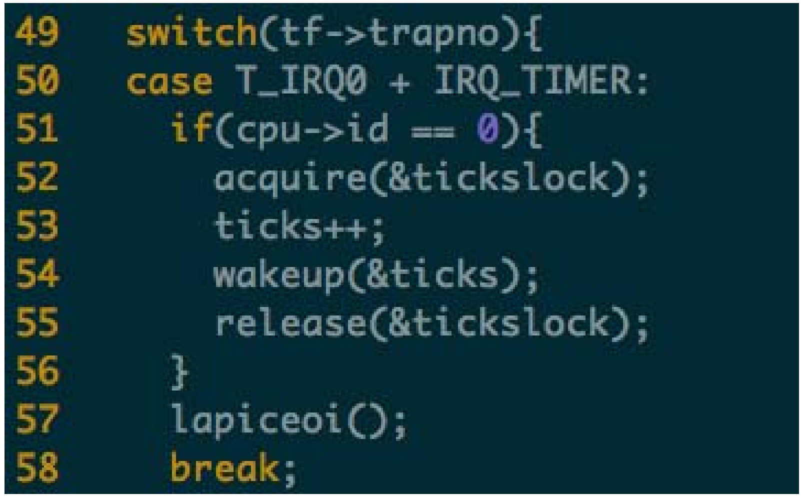
1. PUSH Flag Register
2. Clear IF and TF
3. PUSH CS
4. PUSH IP
5. Fetch Interrupt vector contents and place into both IP and CS to start the Interrupt Service Procedure (ISP)



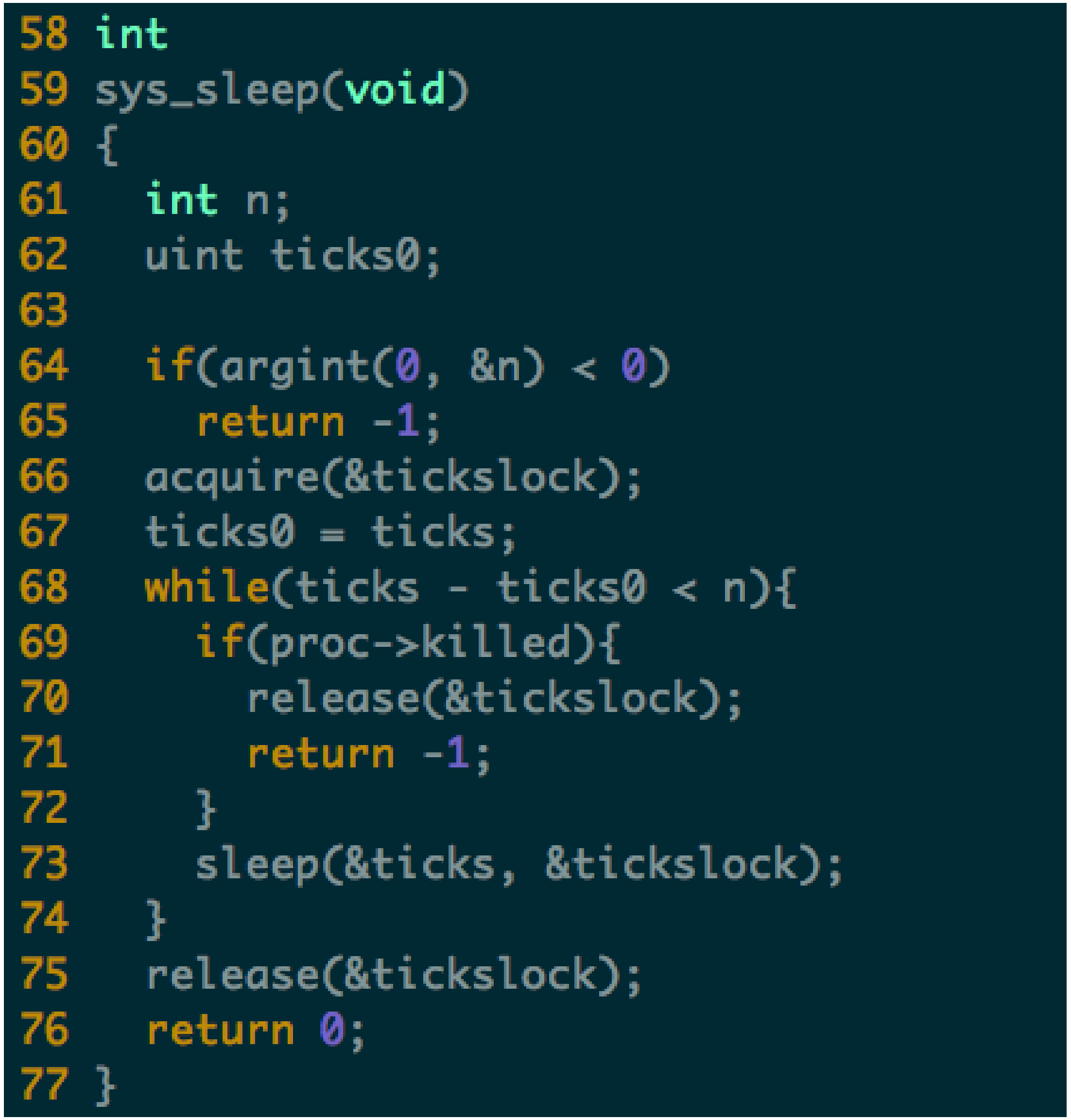


```c
#define IO_TIMER1       0x040           // 8253 Timer #1

// Frequency of all three count-down timers;
// (TIMER_FREQ/freq) is the appropriate count
// to generate a frequency of freq Hz.

#define TIMER_FREQ      1193182
#define TIMER_DIV(x)    ((TIMER_FREQ+(x)/2)/(x))

#define TIMER_MODE      (IO_TIMER1 + 3) // timer mode port
#define TIMER_SEL0      0x00    // select counter 0
#define TIMER_RATEGEN   0x04    // mode 2, rate generator
#define TIMER_16BIT     0x30    // r/w counter 16 bits, LSB first

void
timerinit(void)
{
  // Interrupt 100 times/sec.
  outb(TIMER_MODE, TIMER_SEL0 | TIMER_RATEGEN | TIMER_16BIT);
  outb(IO_TIMER1, TIMER_DIV(100) % 256);
  outb(IO_TIMER1, TIMER_DIV(100) / 256);
  picenable(IRQ_TIMER);
}
```

Timer Interrupt

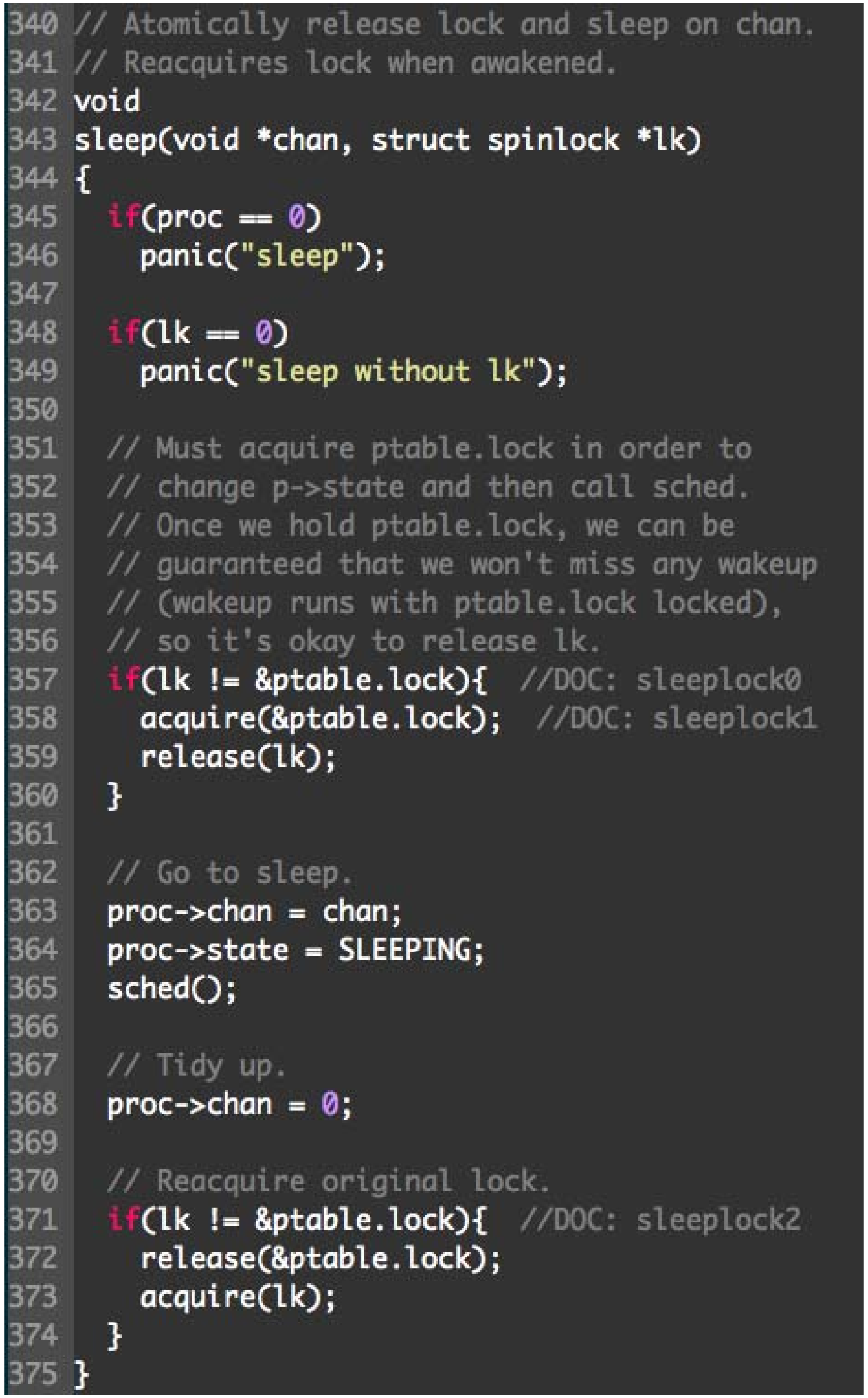


```
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
```

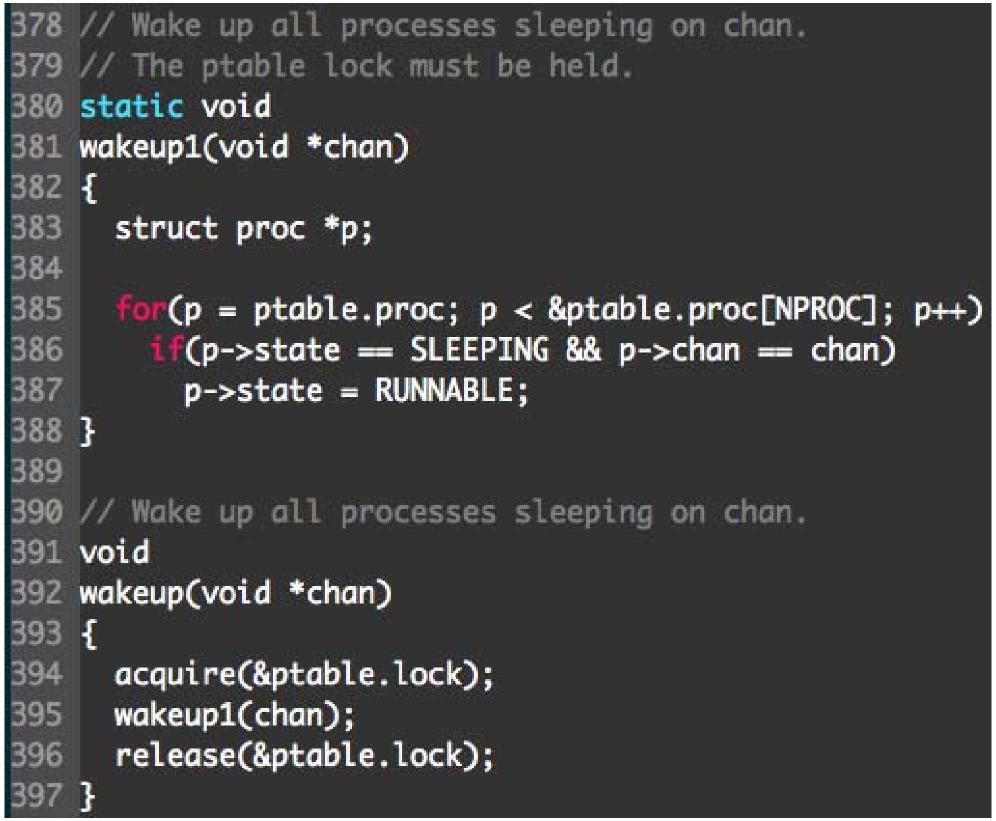


```
int
sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

Sleep & Wakeup



```c
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```



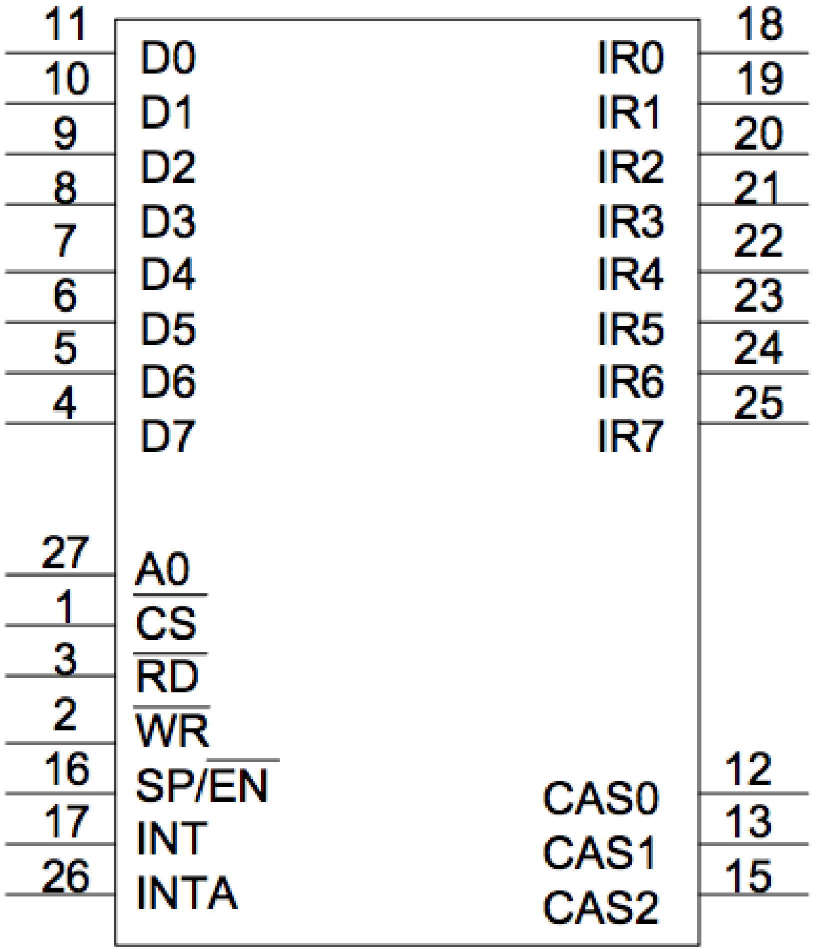
```c
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

Early Boards

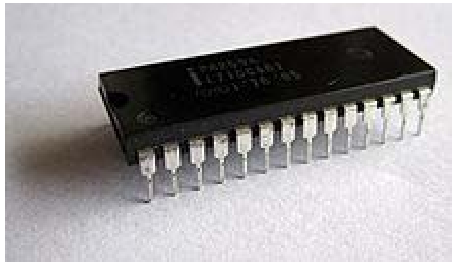
- PIC (Programmable Interrupt Controller)
  - picirq.c
  - 8259A chip
  - Each PIC handles 8 interrupts, and multiplex them on the interrupt pin of the processor
  - PIC can be cascaded
    - Master: 0 to 7
    - Slave: 8 to 15
  - Read timer.c for timer init

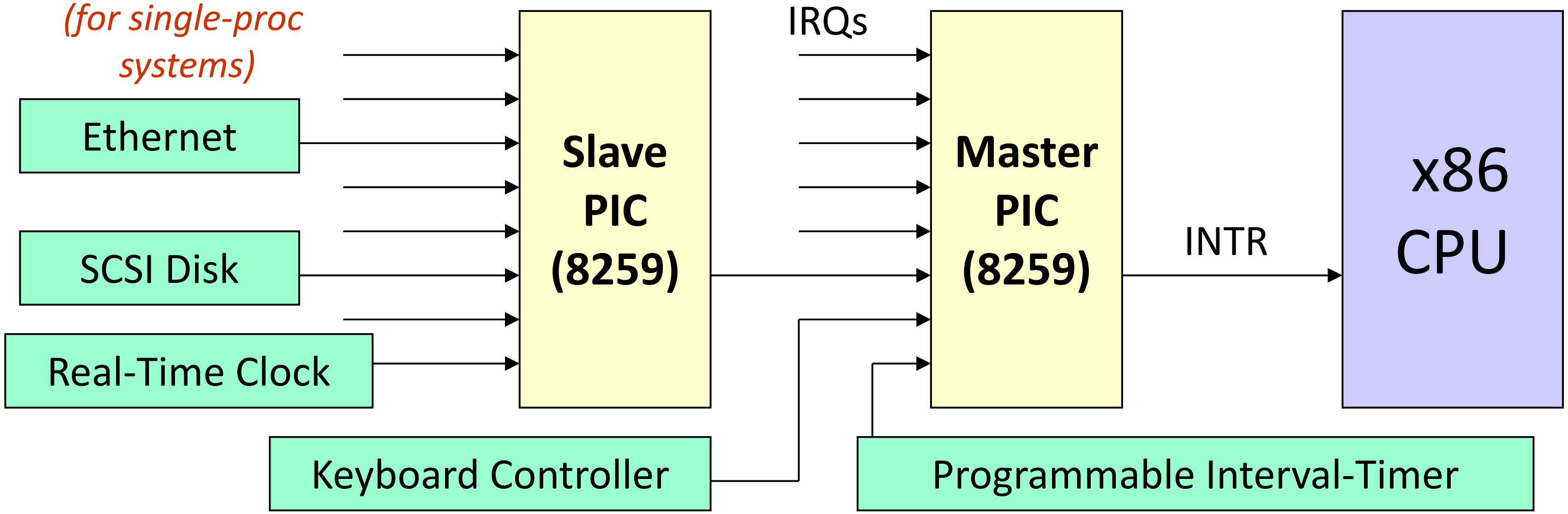
8259A Programmable Interrupt Controller (PIC)

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



Interrupt Hardware





Legacy PC Design (for single-proc systems)

Ethernet

SCSI Disk

Real-Time Clock

Slave PIC (8259)

IRQs

Master PIC (8259)

INTR

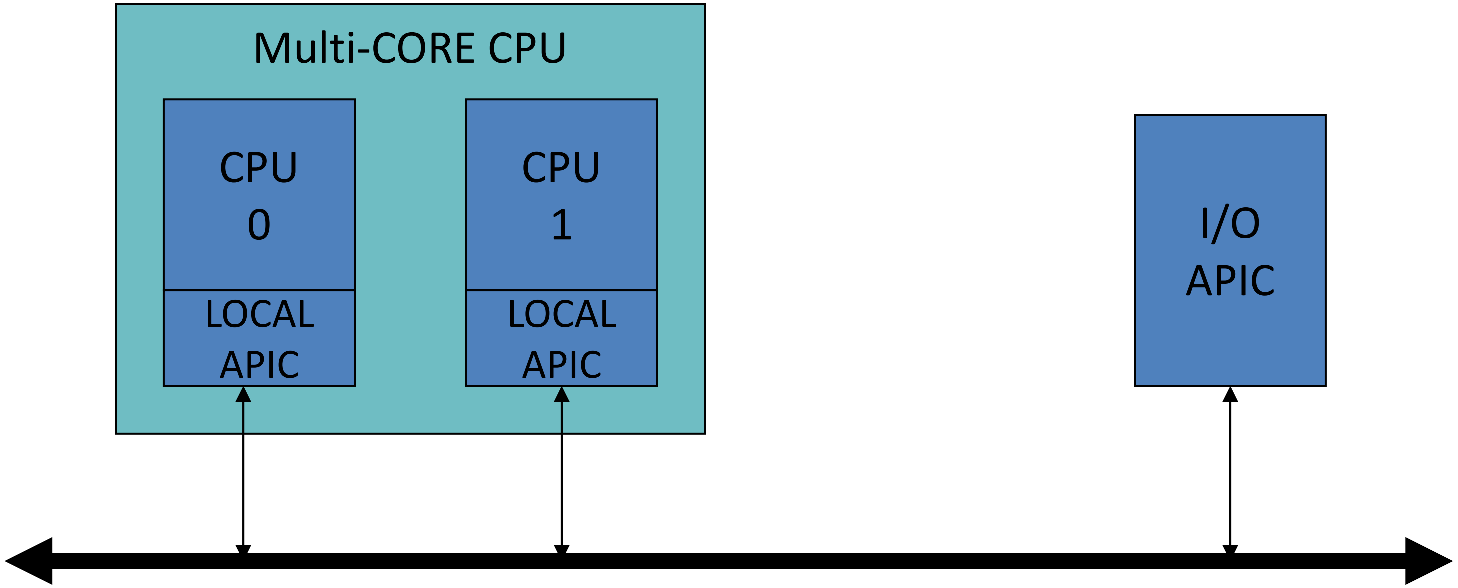
x86 CPU

Keyboard Controller

Programmable Interval-Timer

- 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)

Multiple Logical Processors



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

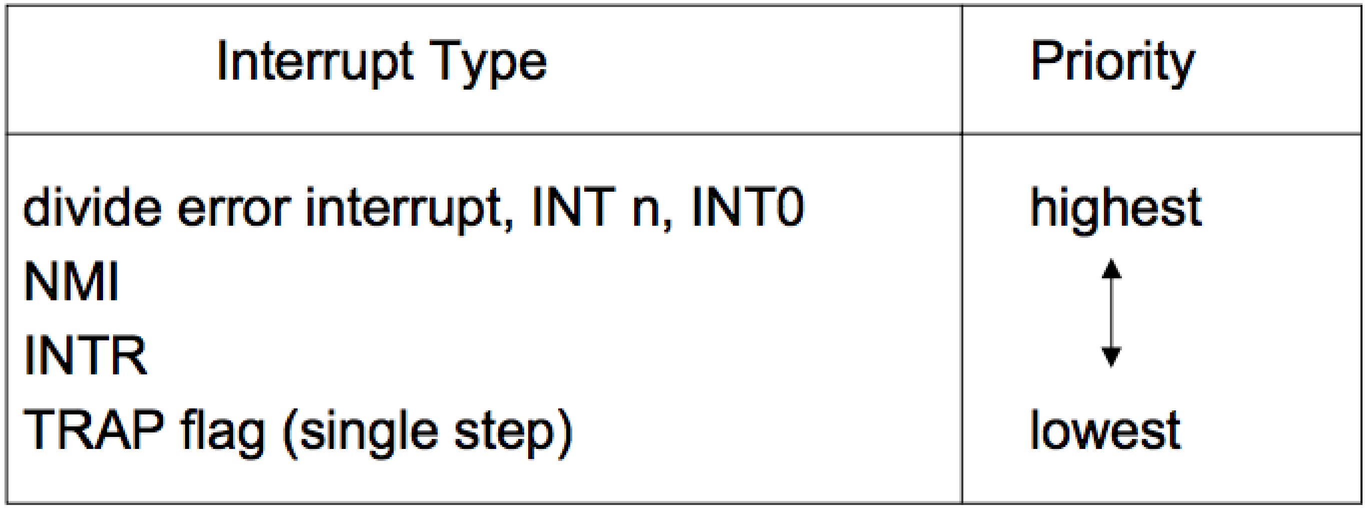
(The legacy PICs are masked when the APICs are enabled)

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
  - 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

Interrupt Priority

- When different type of interrupt (ie. 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, INT0	highest
NMI	↕
INTR	
TRAP flag (single step)	lowest

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

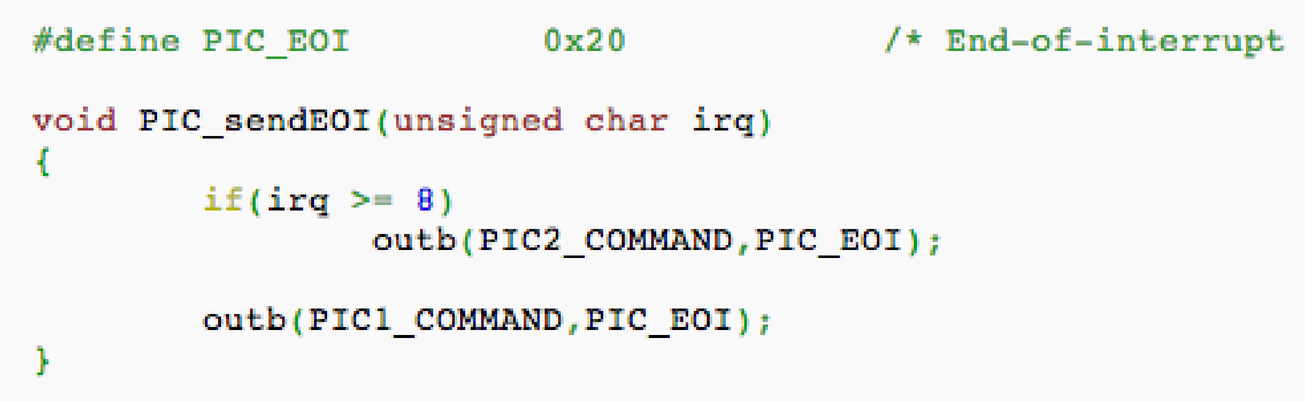
- IRQ assignment is hardware-dependent
  - Sometimes it’s hardwired, sometimes it’s set physically, sometimes it’s programmable
  - PCI bus usually assigns IRQs at boot
- Some IRQs are fixed by the architecture
  - IRQ0: Interval timer
  - IRQ2: Cascade pin for 8259A
- Linux device drivers request IRQs 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

Assigning Vectors to IRQs

- Vector: index (0-255) into interrupt descriptor table
- Vectors 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

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 excuting?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximizing Parallelism

- You want to 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 as soon as possible
- Most devices don’t interfere with each others’ data structures; there’s 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 isn’t 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) shouldn’t have bugs
  - Page fault is possible (trying to touch user data)

Interrupt Masking

- Two different types: global and per-IRQ
  - Global — delays all interrupts
  - Selective — individual IRQs can be masked selectively
  - Selective masking is usually what’s needed — interference most common from two interrupts of the same type

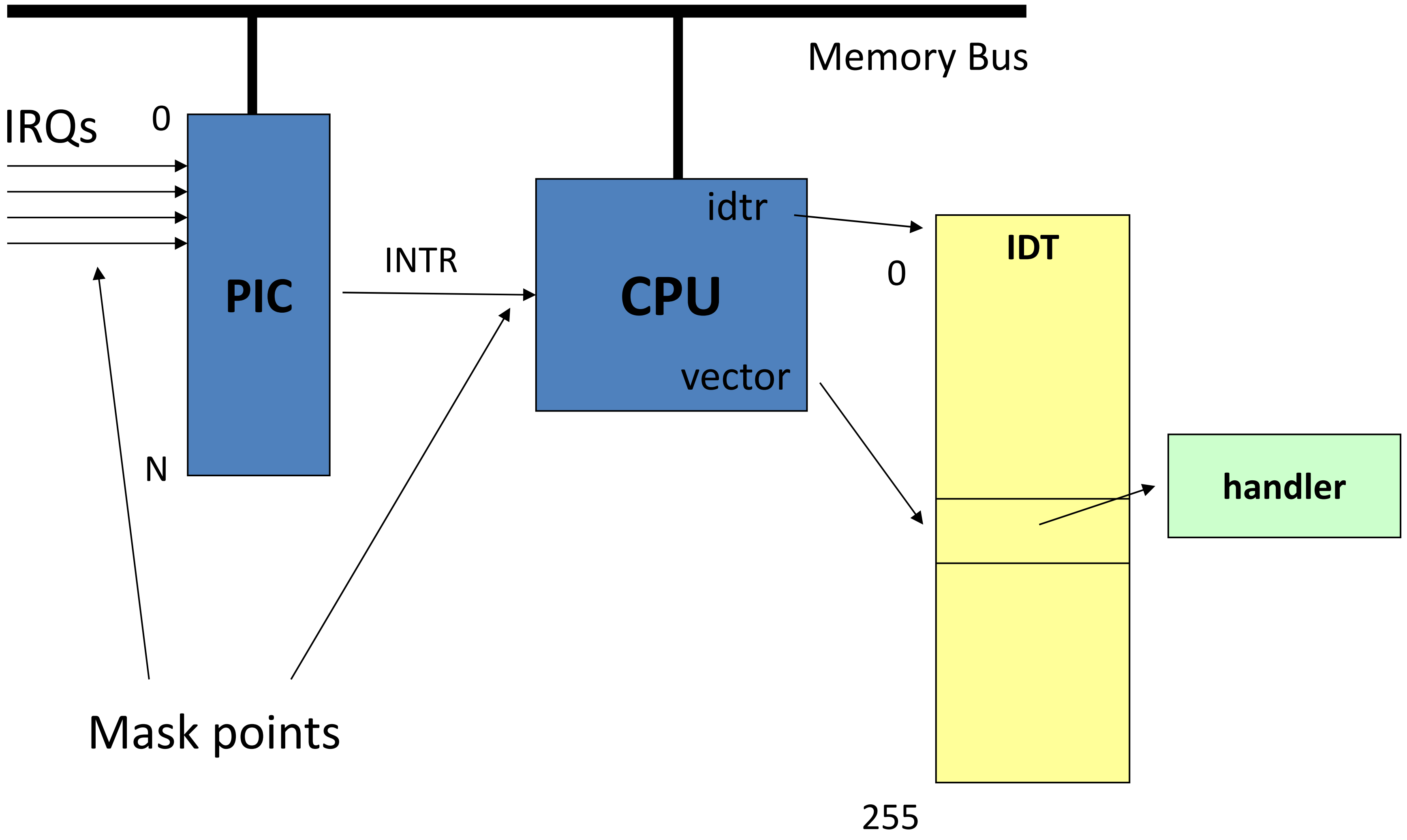
- NMI (Non-Maskable Interrupt)

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.

Putting It All Together



/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 [0xebffffff].
  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)

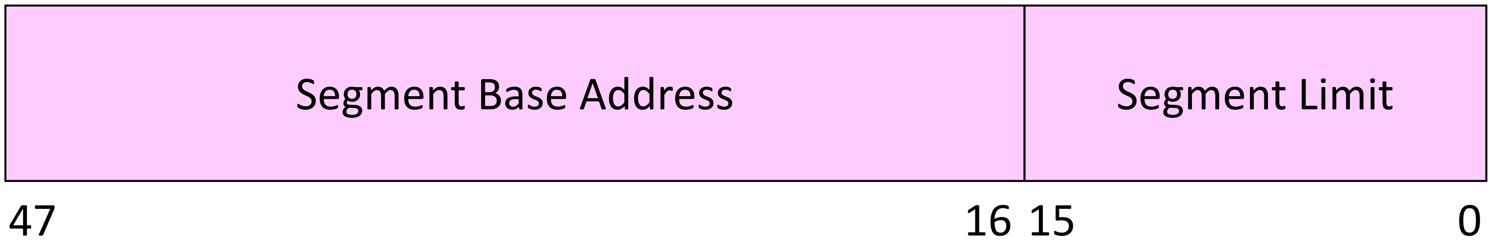
  defines entry-points for the various code-routines that will handle all ‘interrupts’ and ‘exceptions’

- 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:



Segment Base Address

Segment Limit

47

16 15

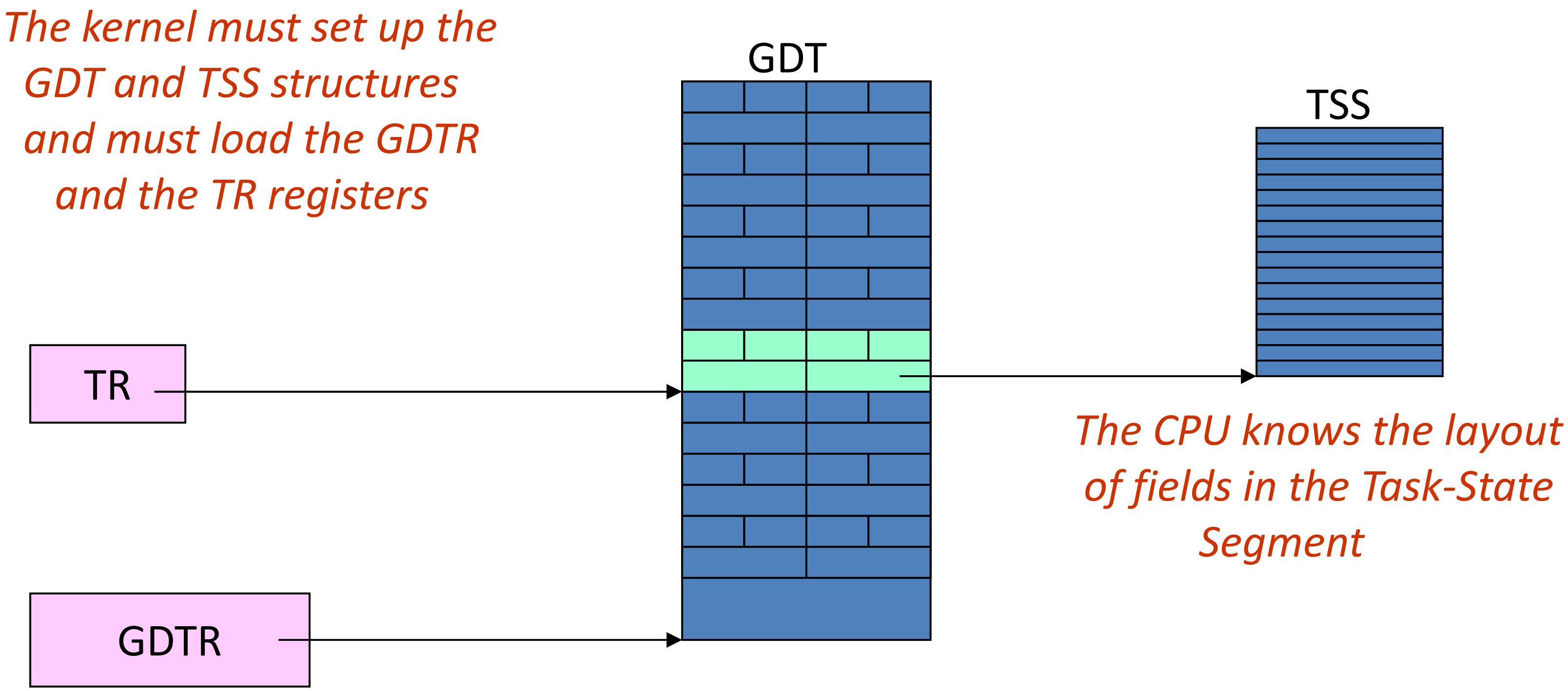
0

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

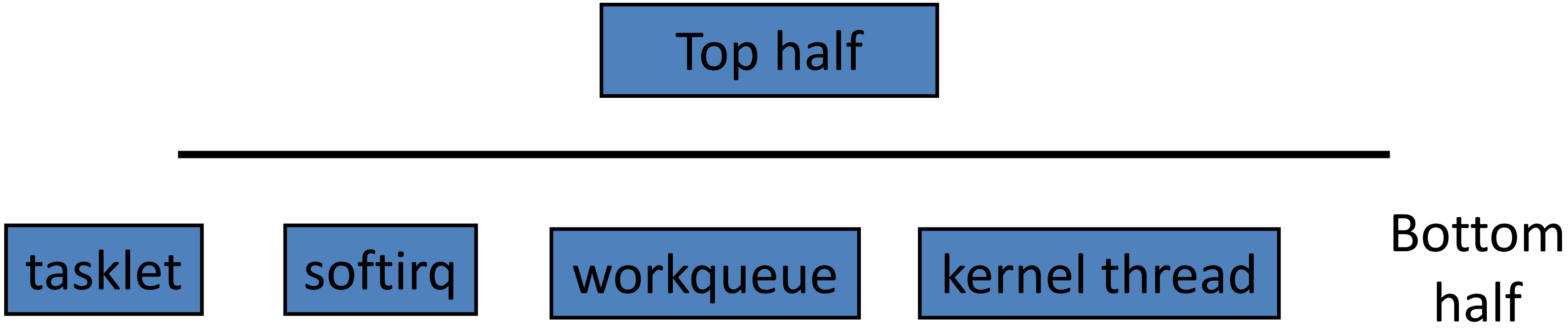
BOTTOM HALF

Summary: Interrupts vs Exceptions

- Varying terminology but for Intel:
  - **Interrupt** (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)

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

tasklet

softirq

workqueue

kernel thread

Bottom half

Top Half: Do it Now!

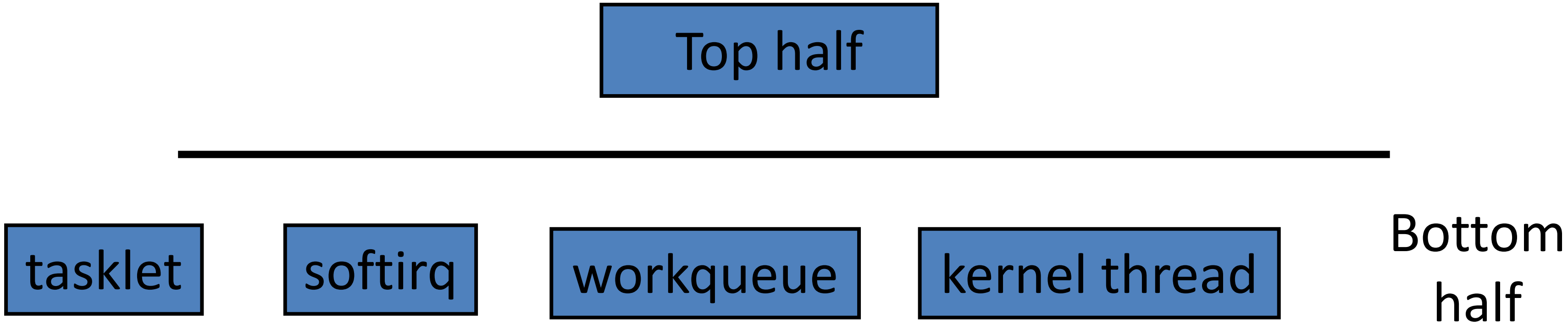
- Technically *is* the interrupt handler
- Perform minimal, common functions: save registers, unmask *other* interrupts. Eventually, undoes that: restores registers, returns to previous context.
  - Often written in assembler
- IRQ is typically masked for duration of top half
- Most important: call proper 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’re 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:

Priority	*Type*
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

I/O

Yubin Xia, Rong Chen

IPADS, SJTU

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

Review

- Interrupt concepts
  - IDT, Trap Num, Interrupt vector, IRQ, System call number, …
  - Vectors usually IRQ# + 32
  - Triple fault

- Interrupt process
  - Device, PIC (8259A), IDT, trap()
  - CPU checks interrupt every cycle, if interrupt enabled
  - Masked interrupt, umaskable interrupt

Intel-Reserved ID-Numbers

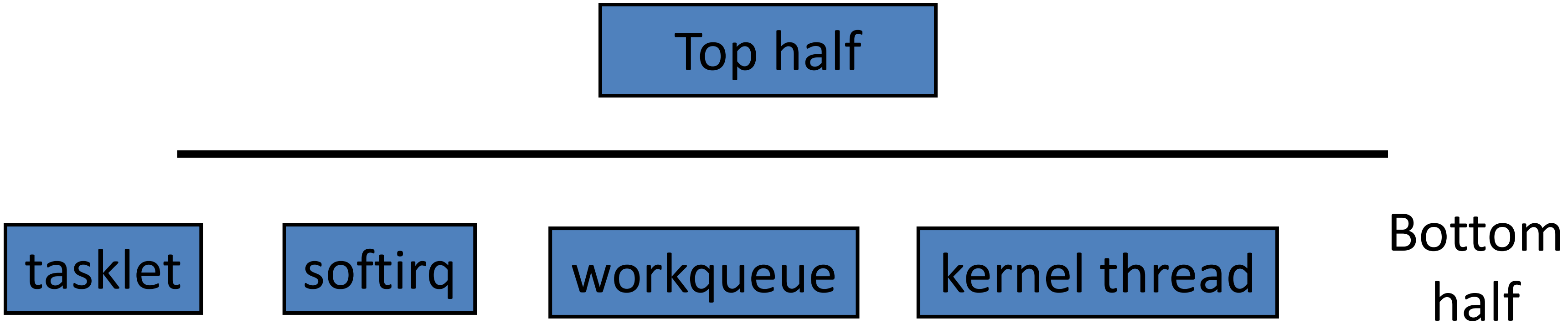
- Of the 256 possible interrupt ID numbers, Intel reserves the first 32 for ‘exceptions’
- OS’s such as Linux are free to use the remaining 224 available interrupt ID numbers for their own purposes (e.g., for service-requests from external devices, or for other purposes such as system-calls)
- Examples:
  - 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

Summary: Interrupts vs Exceptions

- Varying terminology but for Intel:
  - **Interrupt** (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)

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)



Warning: No Process Context

- Interrupts (as opposed to exceptions) are not associated with particular instructions
- They’re 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 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()`**

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:

Priority	*Type*
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

FLEXSC

Flexible System Call

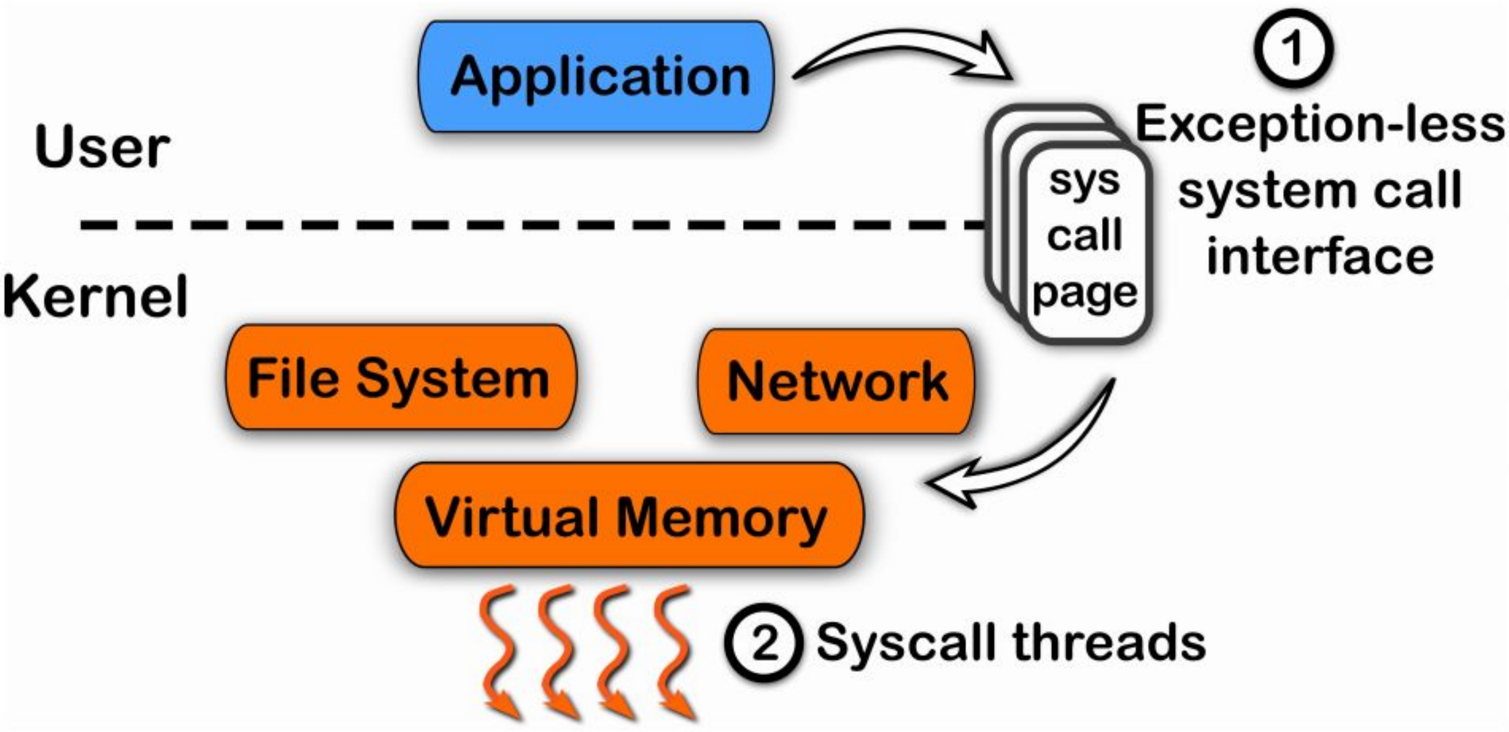
- Basic Idea
  - “Flexible System Call Scheduling with Exception-Less System Calls”
  - University of Toronto
  - OSDI 2010

Flexible System Call

- New system call mechanism – Flexible System Call
  - no direct mapping between user threads and kernel threads
  - *system call page*
  - 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

FlexSC: Another Way to System Call

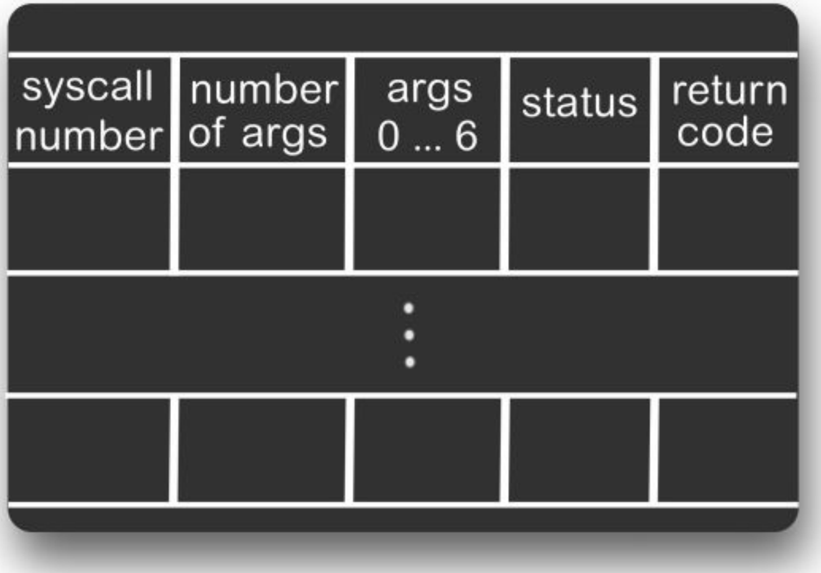
- Exception-less syscall: remove synchronicity by decoupling invocation from execution



Exception-less System Call

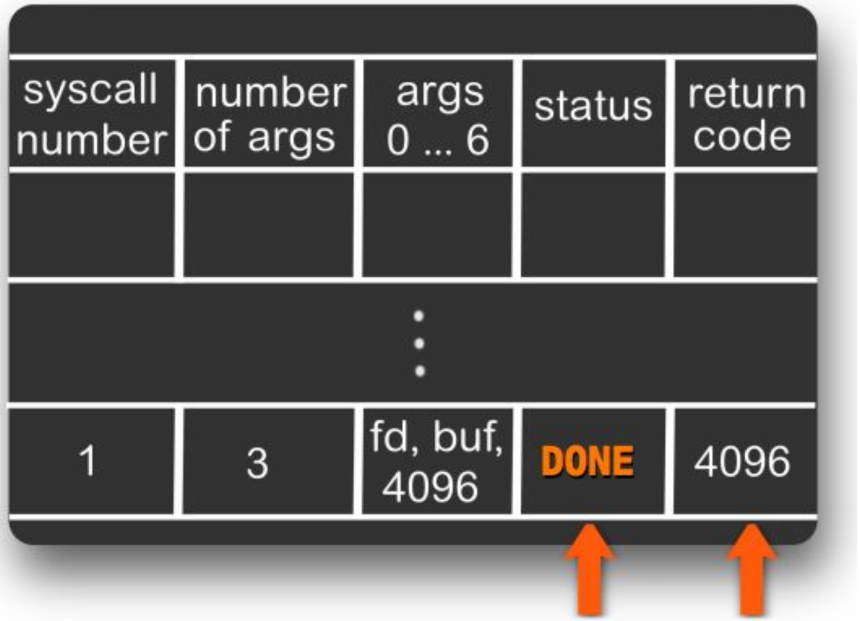
```
write(fd, buf, 4096);
```

```
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
		⋮		

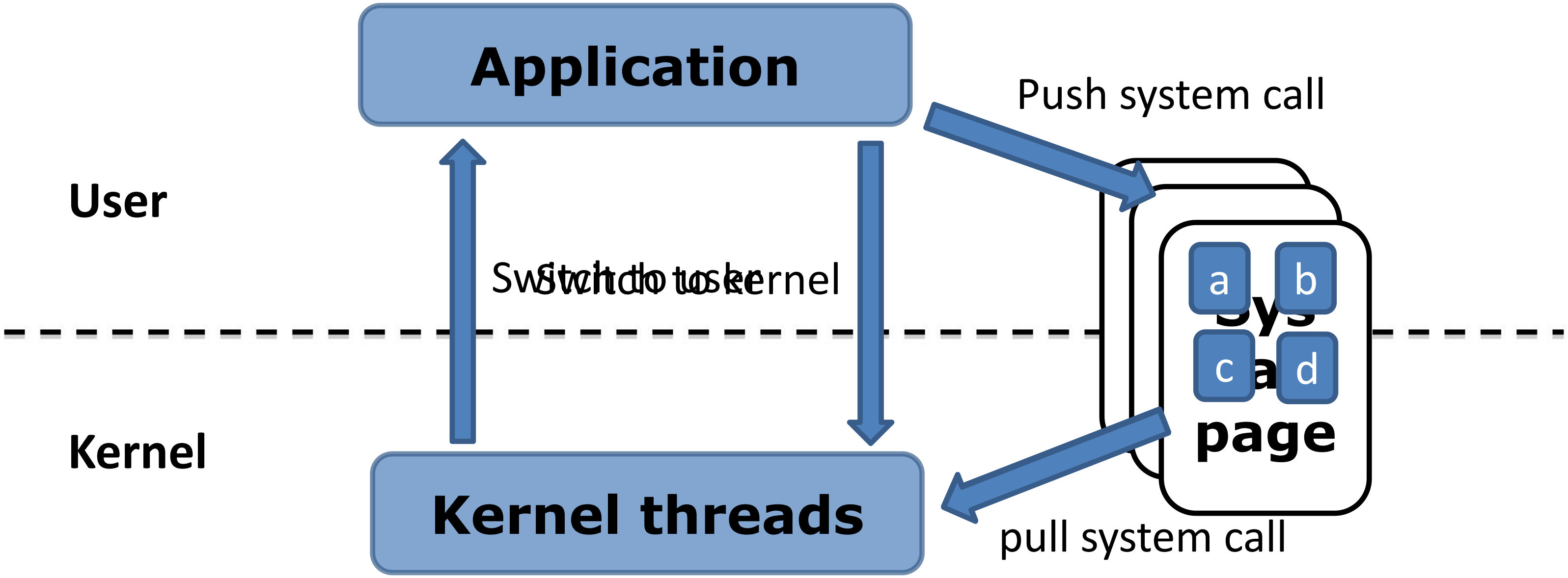
```
write(fd, buf, 4096);

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
		⋮		
1	3	fd, buf, 4096	DONE	4096

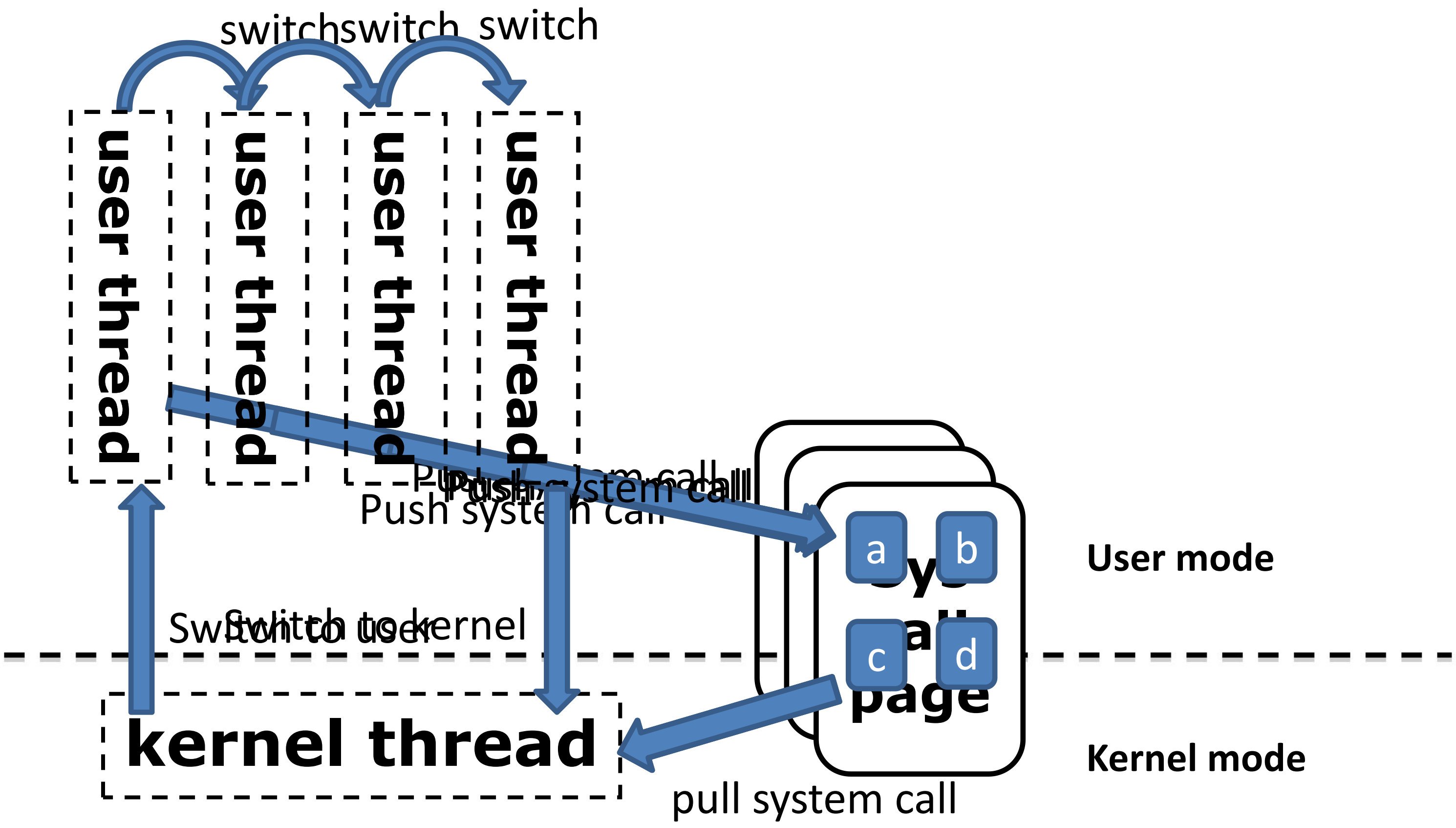
Flexible System Call

- Basic Idea



Running on a single core

- How it works in a single core machine?

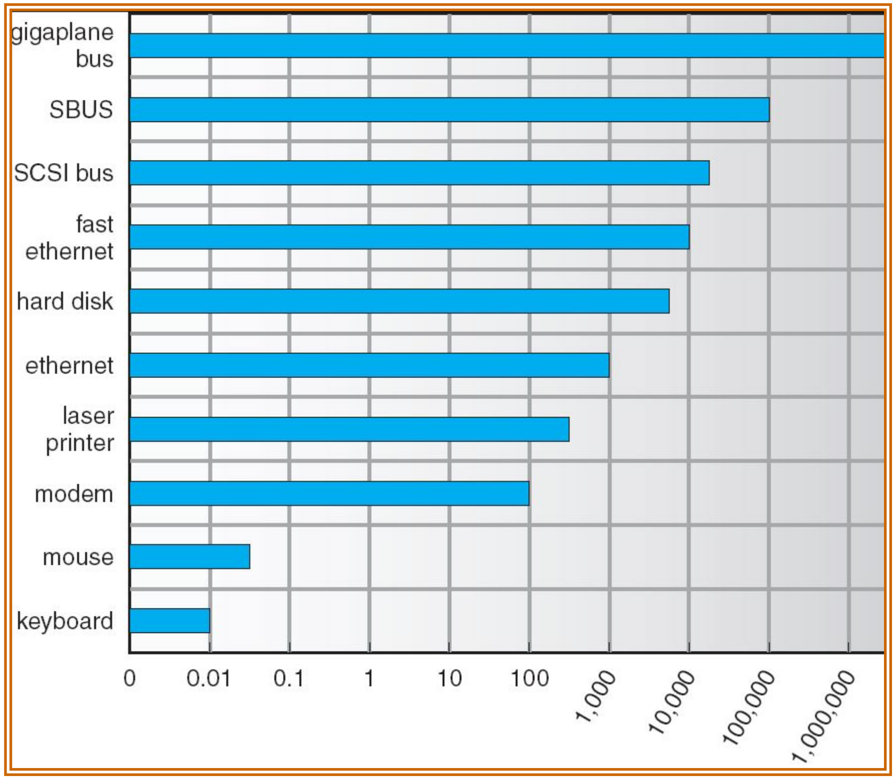


I/O

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

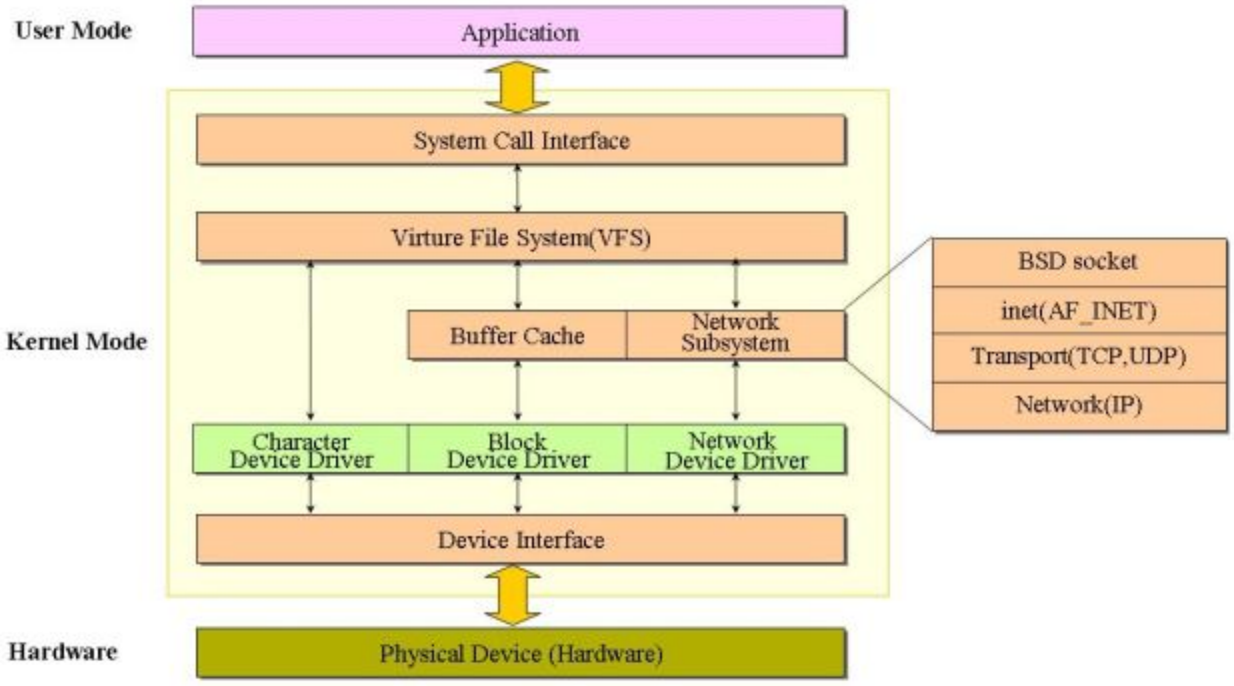
- Provides uniform interfaces, despite wide range of different devices
  - This code works on many different devices:

```
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

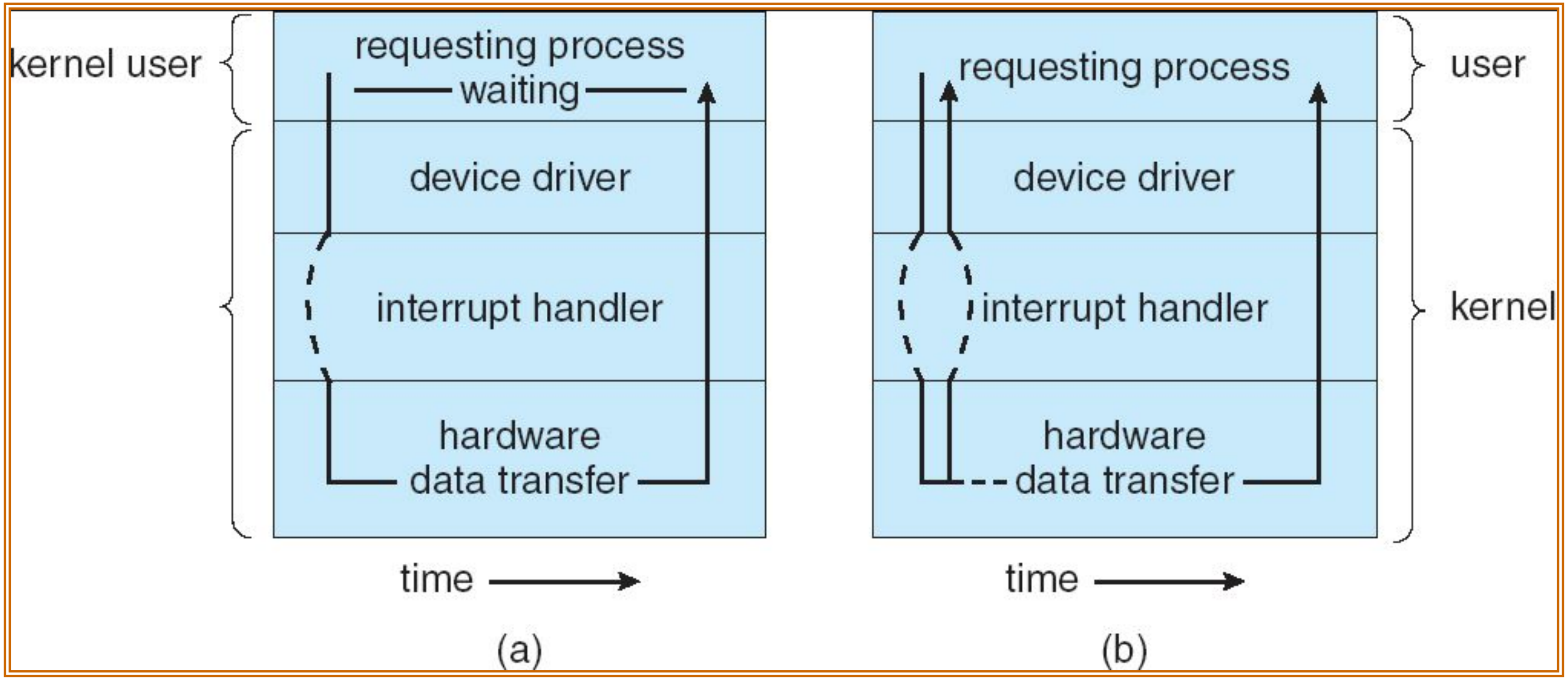
Block Devices and Network Devices

- 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
  - 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

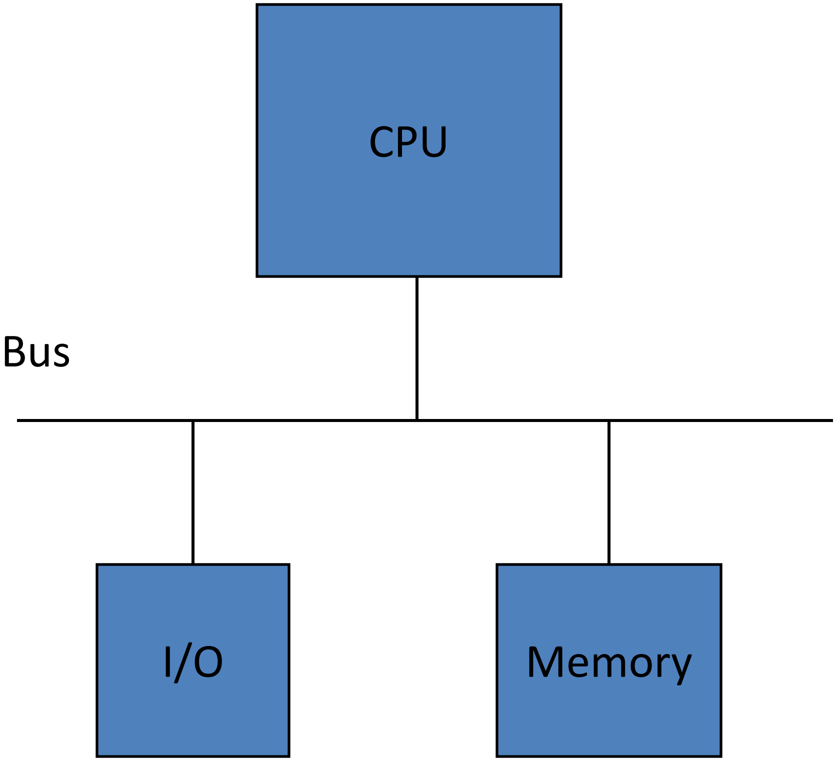
How Does User Deal with Timing?

- 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

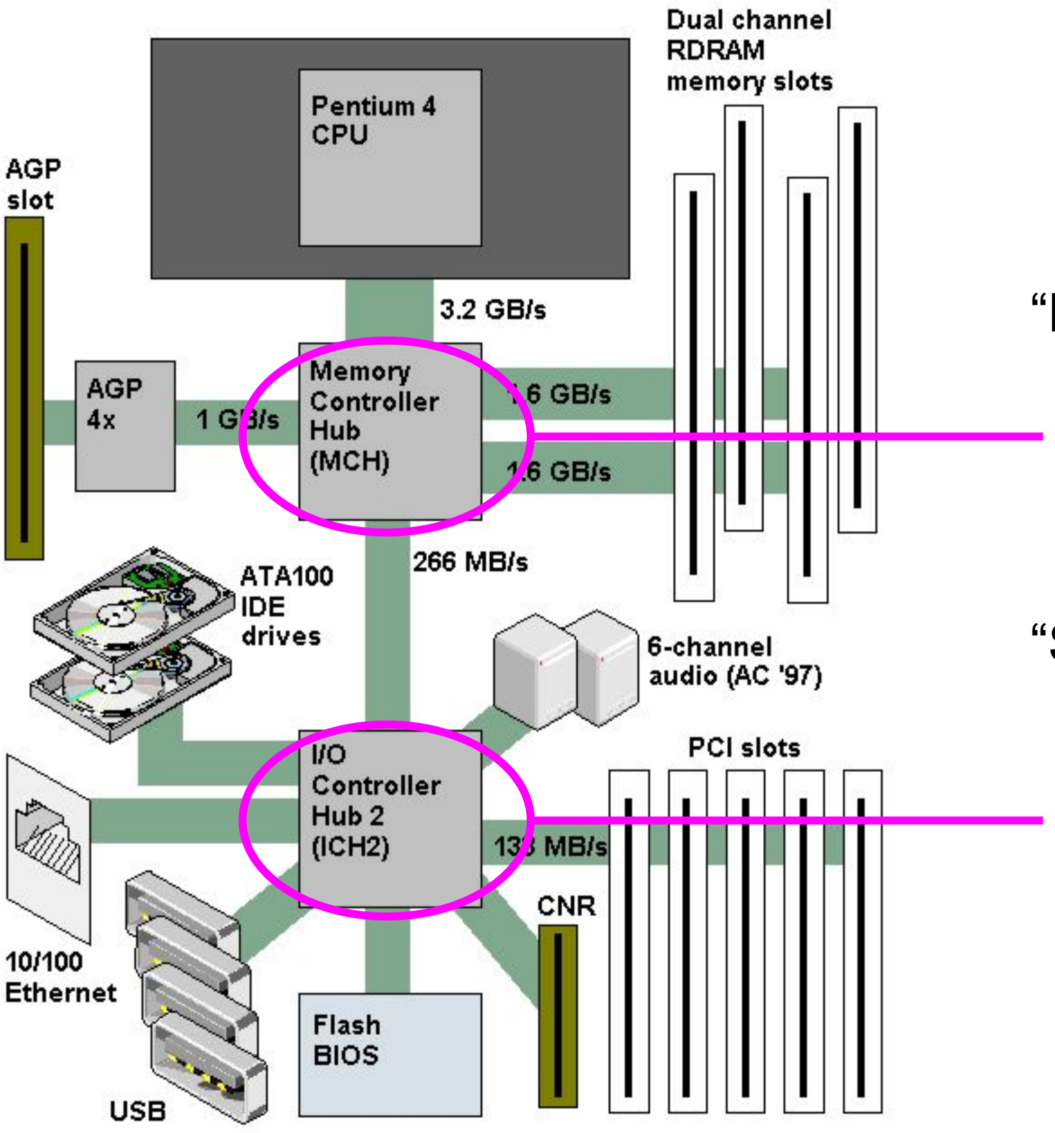


I/O Architecture: The Very Abstract View



I/O Architecture: A Modern Example

From Computer Desktop Encyclopedia
© 2001 The Computer Language Co. Inc.



“Northbridge”

- Memory
- AGP/PCI-Express
- Built-in display

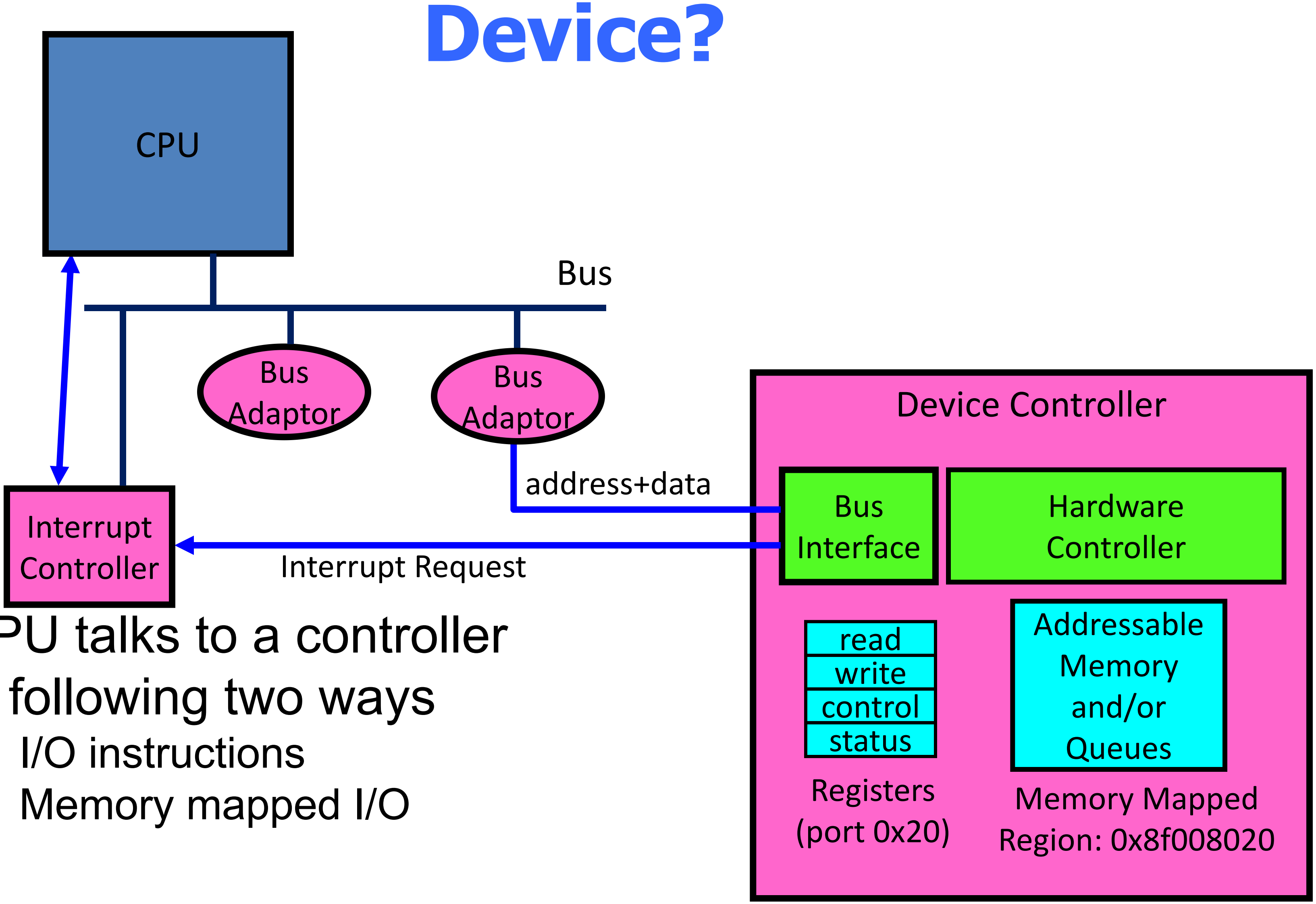
“Southbridge”

- ATA/IDE
- PCI bus
- USB/Firewire bus
- Serial/Parallel ports
- DMA controller
- Interrupt controller
- RTC, ACPI, BIOS, …

I/O Hardware

- I/O controllers
  - Interface between CPU and I/O devices
  - Provides CPU with special instructions and registers
- I/O addresses
  - “Names” for CPU to control the I/O hardware
  - Memory locations or port numbers
- OS mechanism
  - Use I/O instruction and I/O address to control a device
  - 3 types of interactions with I/O hardware: polling, interrupt-driven, and DMA

How does CPU Actually Connect to Device?



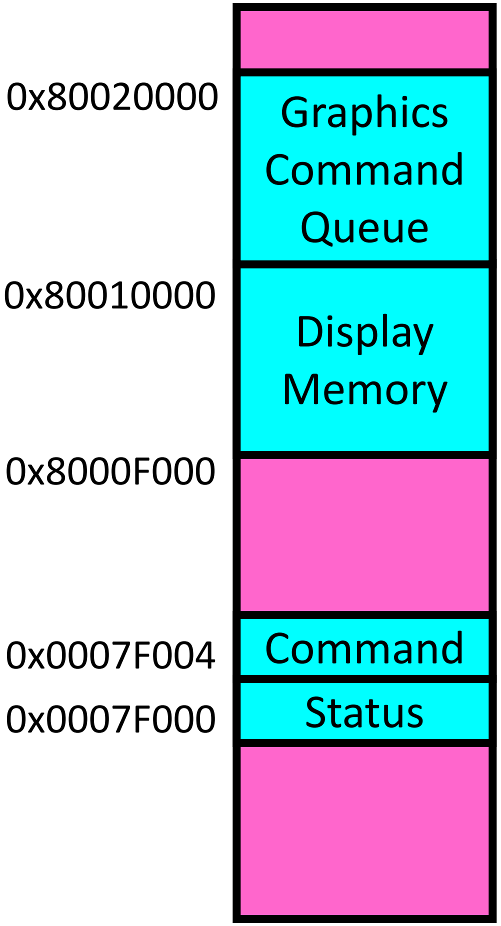
- CPU talks to a controller
- In following two ways
  - I/O instructions
  - Memory mapped I/O

I/O Instructions and Memory-Mapped I/O

- I/O instructions
  - Access device’s registers through I/O port numbers
  - Special CPU instructions dealing with I/O
  - Example from the Intel architecture: out 0x21,AL
- Memory mapped I/O (MMIO)
  - Device’s registers/memory appear in CPU’s physical address space
  - I/O accomplished with memory load/store instructions
  - Mapped by MMU, addresses set by hardware jumpers or programming at boot time
  - Can be protected with page tables

Example: Memory-Mapped Display Controller

- Simply writing to display memory changes image on screen
  - Addr: 0x8000F000—0x8000FFFF
  - Also called the "frame buffer"
- One may write graphics description to command-queue area
  - E.g., enter a set of triangles that describe some scene
- Writing to the command register may cause on-board graphics hardware to do something
  - Like execute commands in the command-queue area
  - Say render the above scene

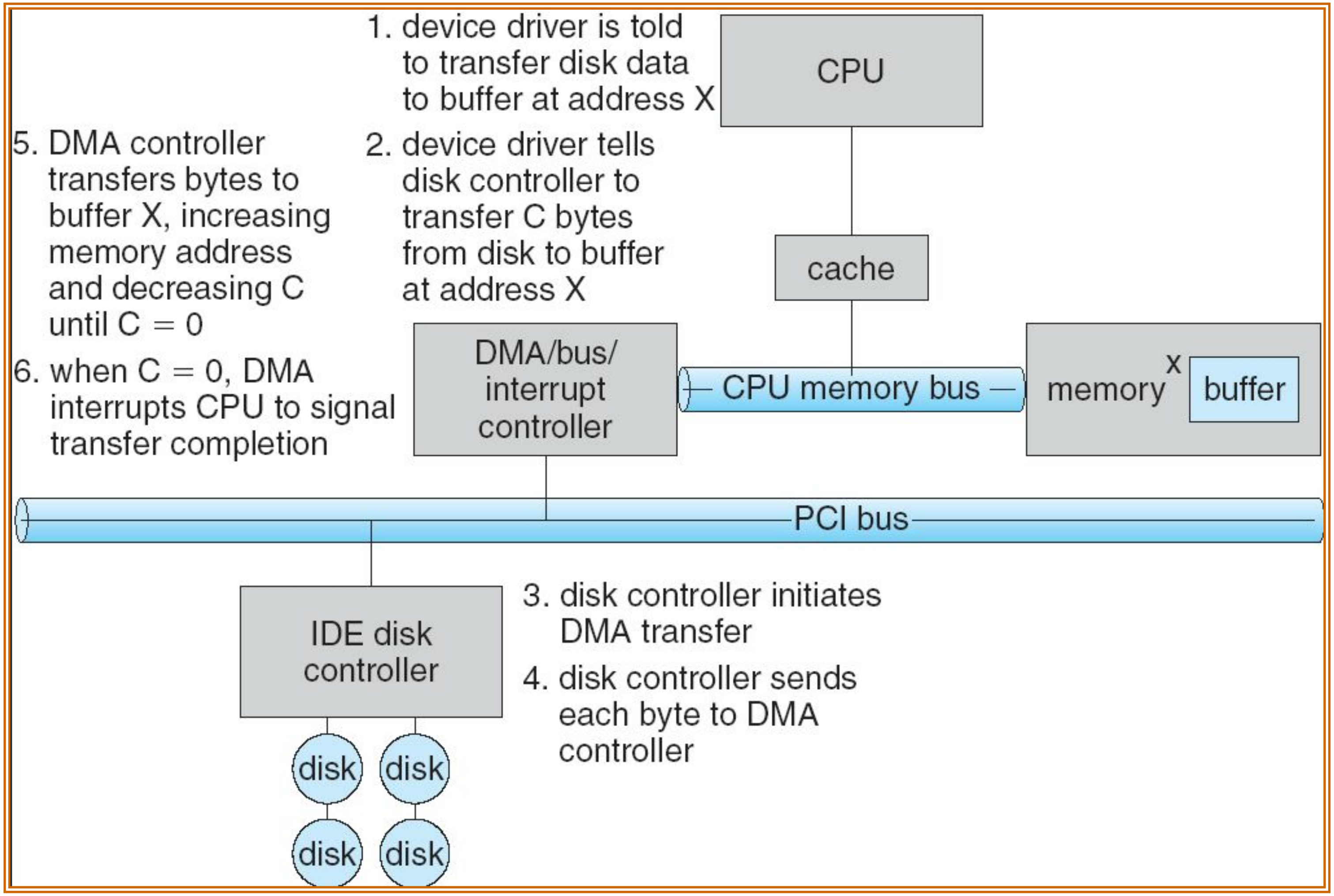


Physical Address Space

Transfering Data To/From Controller

- Programmed I/O (PIO):
  - Each byte transferred via processor in/out or load/store
  - Pro: Simple hardware, easy to program
  - Con: Consumes processor cycles proportional to data size
  - For small/simple I/O
- Direct Memory Access (DMA):
  - Give controller access to memory bus
  - Ask it to transfer data to/from memory directly
  - Pro: device transfers data without burdening CPU
  - Con: need setup
  - For high throughput I/O

Steps in a DMA Transfer



1. device driver is told to transfer disk data to buffer at address X

2. device driver tells disk controller to transfer C bytes from disk to buffer at address X

3. disk controller initiates DMA transfer

4. disk controller sends each byte to DMA controller

5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0

6. when C = 0, DMA interrupts CPU to signal transfer completion

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

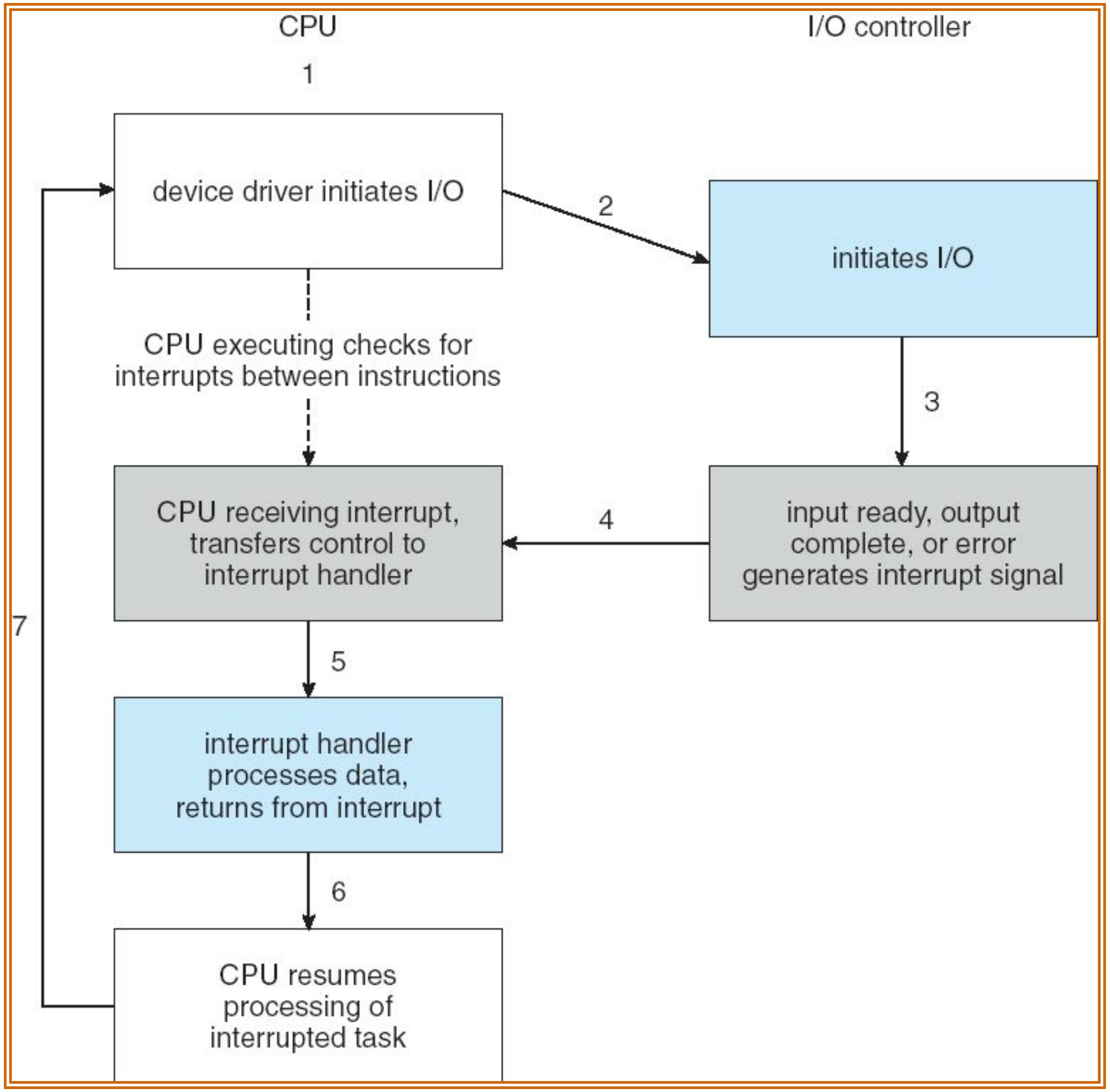
- CPU sets up interrupt handler vector before I/O
- CPU issues I/O request and continues other tasks
- I/O device processes the I/O request
- I/O device triggers CPU interrupt-request line
- Interrupt handler receives interrupts and dispatches to correct handler

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

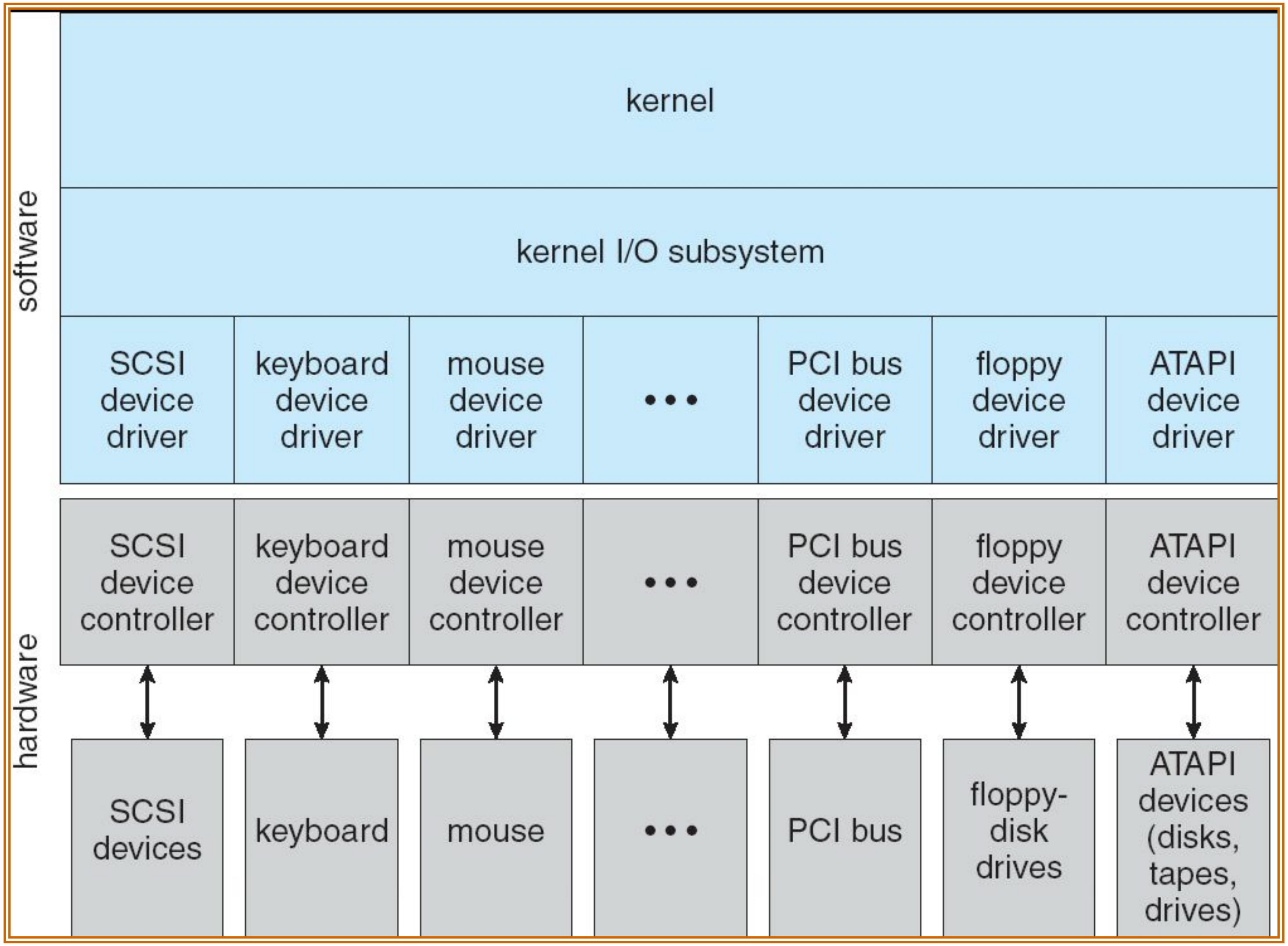
Interrupt-Driven I/O Cycle


CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Direct Memory Access (DMA)

- OS steps to use DMA:
  - Set up DMA channel (address, registers, direction)
  - Prepare and register interrupt service routine
  - Tell the DMA controller to start
  - When transfer completes, controller will interrupt

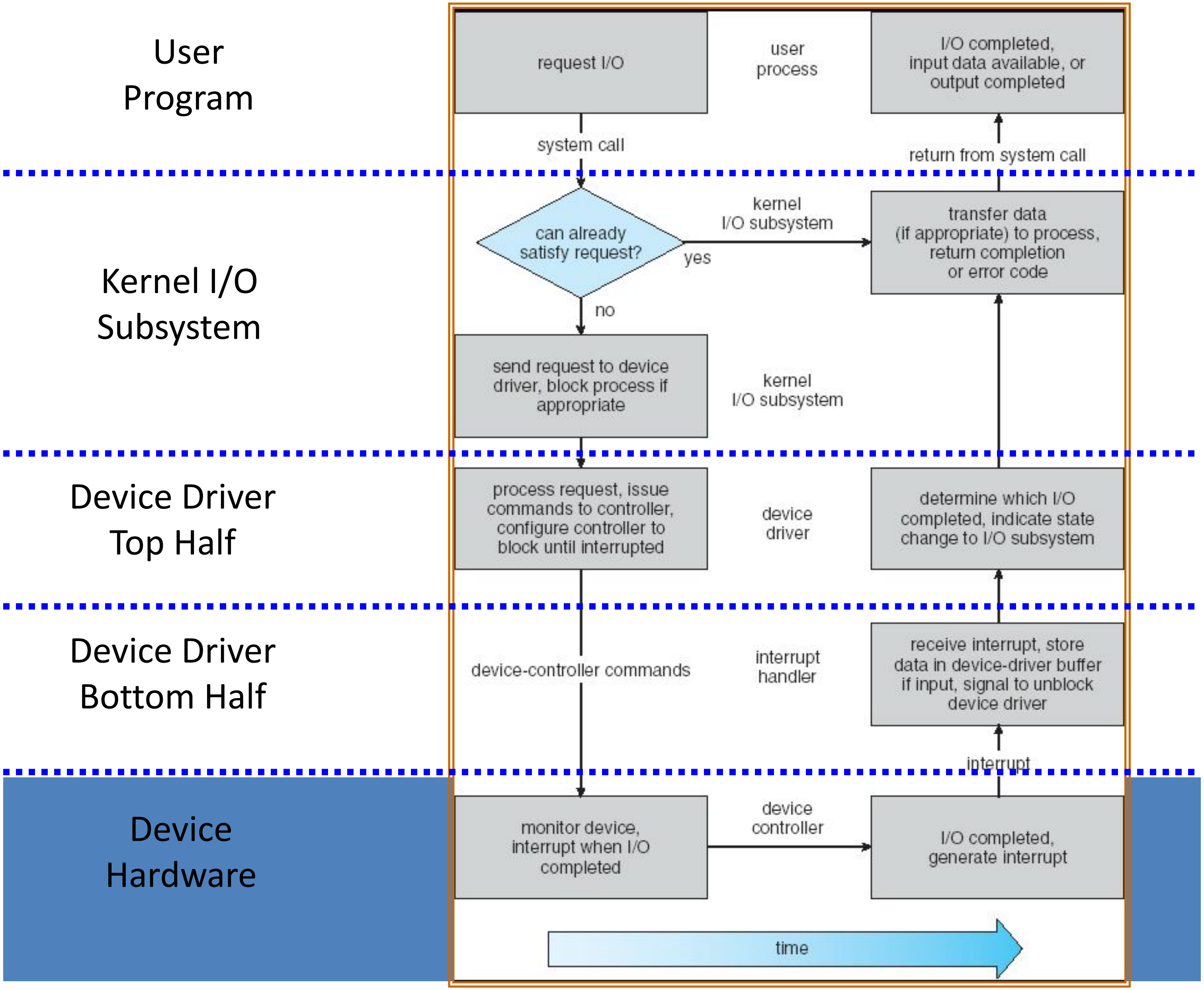
A Kernel I/O Structure



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



User Program

Kernel I/O Subsystem

Device Driver Top Half

Device Driver Bottom Half

Device Hardware

request I/O

user process

I/O completed, input data available, or output completed

system call

return from system call

can already satisfy request?

kernel I/O subsystem

yes

transfer data (if appropriate) to process, return completion or error code

no

send request to device driver, block process if appropriate

kernel I/O subsystem

process request, issue commands to controller, configure controller to block until interrupted

device driver

determine which I/O completed, indicate state change to I/O subsystem

device-controller commands

interrupt handler

receive interrupt, store data in device-driver buffer if input, signal to unblock device driver

interrupt

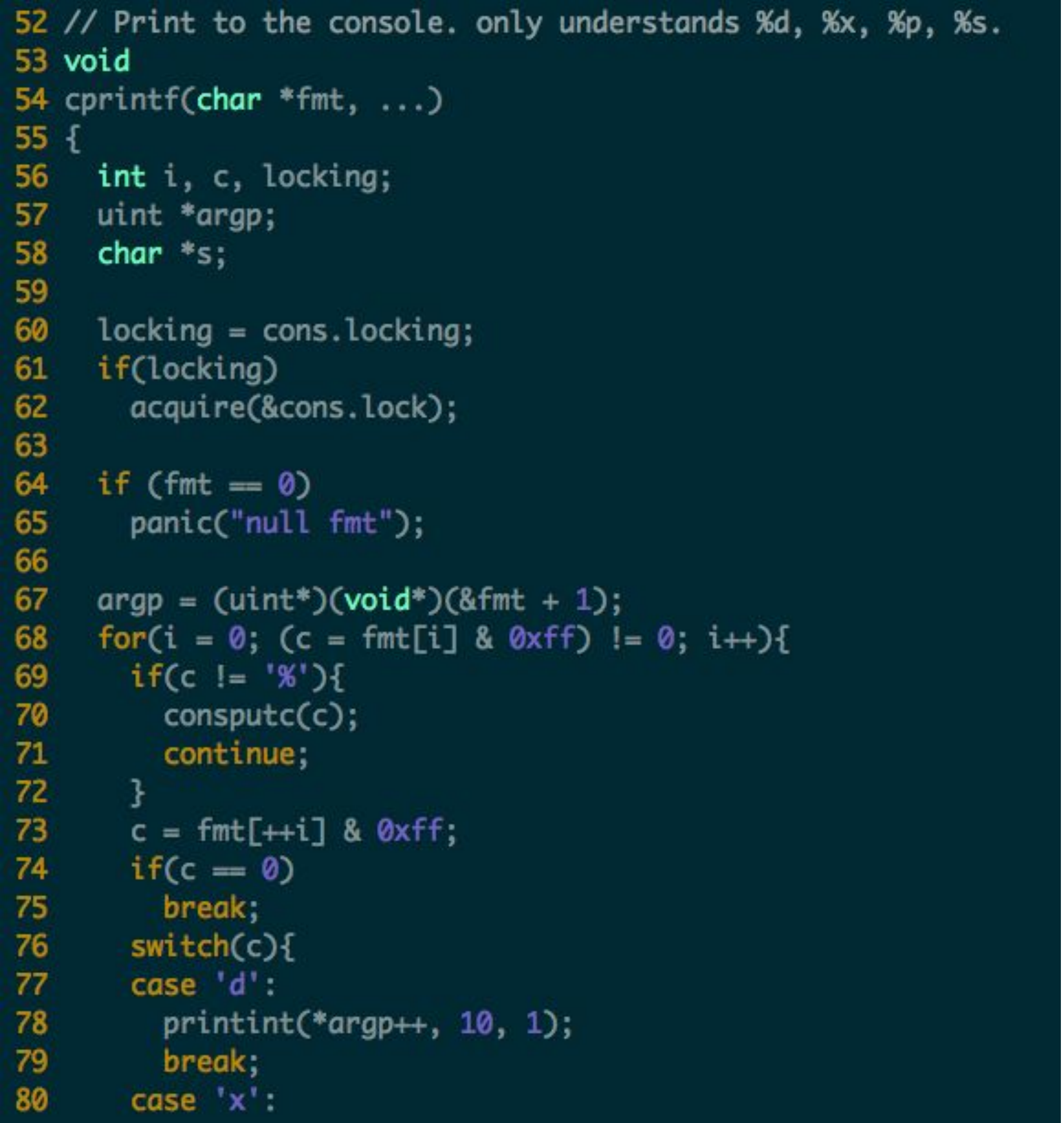
monitor device, interrupt when I/O completed

device controller

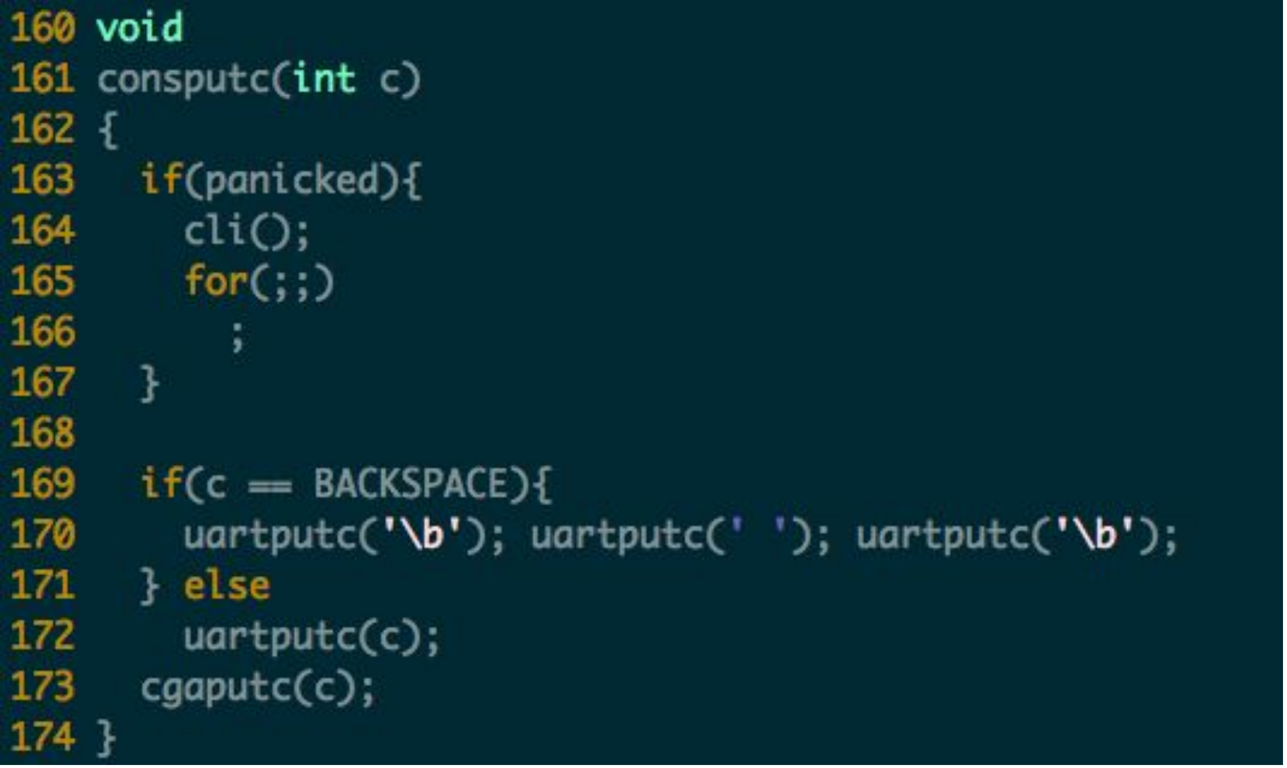
I/O completed, generate interrupt

time

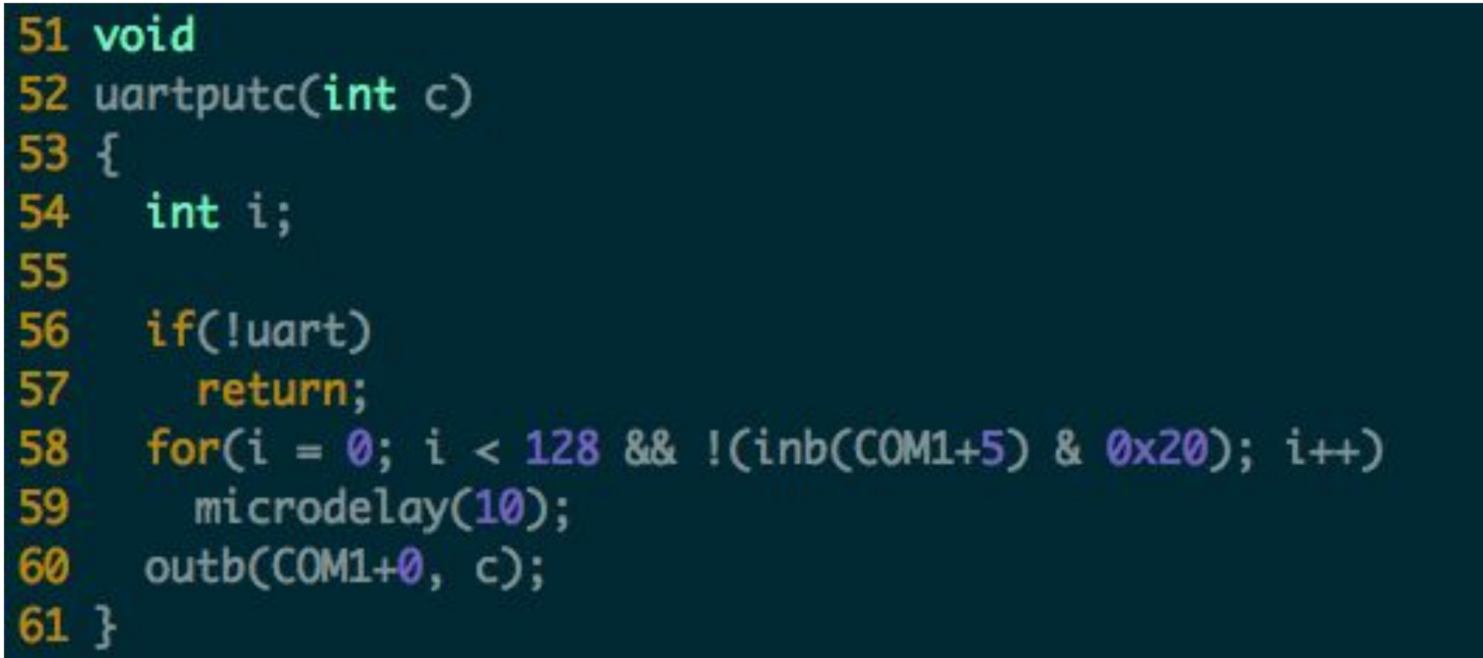
Example: Char Device



```c
// Print to the console. only understands %d, %x, %p, %s.
void
cprintf(char *fmt, ...)
{
  int i, c, locking;
  uint *argp;
  char *s;

  locking = cons.locking;
  if(locking)
    acquire(&cons.lock);

  if (fmt == 0)
    panic("null fmt");

  argp = (uint*)(void*)(&fmt + 1);
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
    if(c != '%'){
      consputc(c);
      continue;
    }
    c = fmt[++i] & 0xff;
    if(c == 0)
      break;
    switch(c){
    case 'd':
      printint(*argp++, 10, 1);
      break;
    case 'x':
```

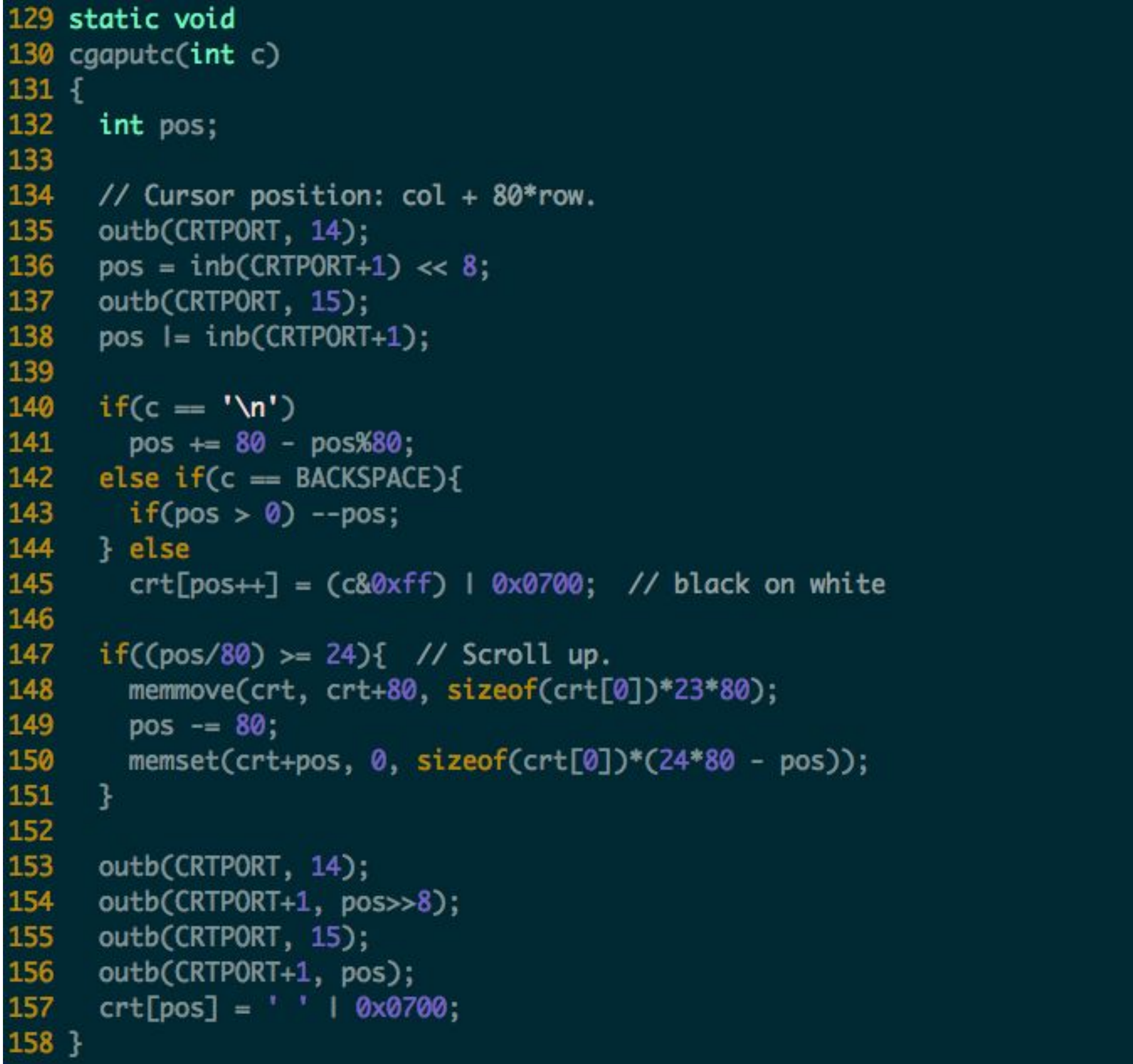


```c
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}
```



```c
void
uartputc(int c)
{
  int i;

  if(!uart)
    return;
  for(i = 0; i < 128 && !(inb(COM1+5) & 0x20); i++)
    microdelay(10);
  outb(COM1+0, c);
}
```

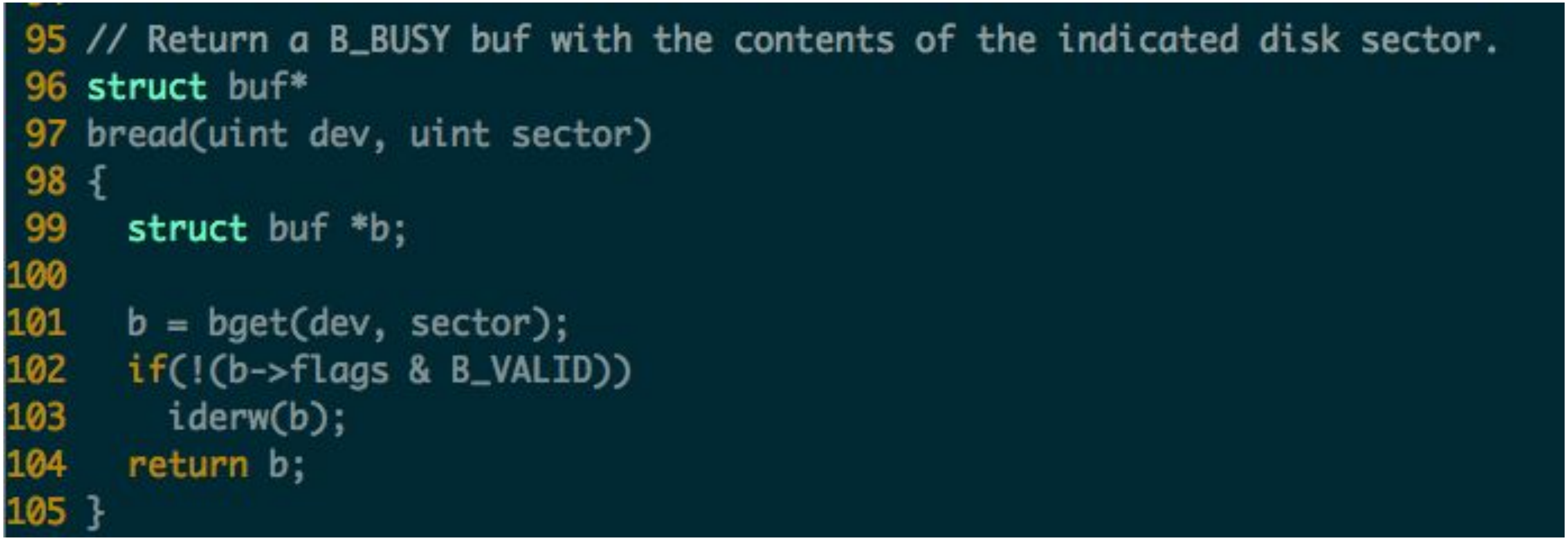
CGA Operation



```
static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```

Recall : read

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



```
// Return a B_BUSY buf with the contents of the indicated disk sector.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
}
```

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

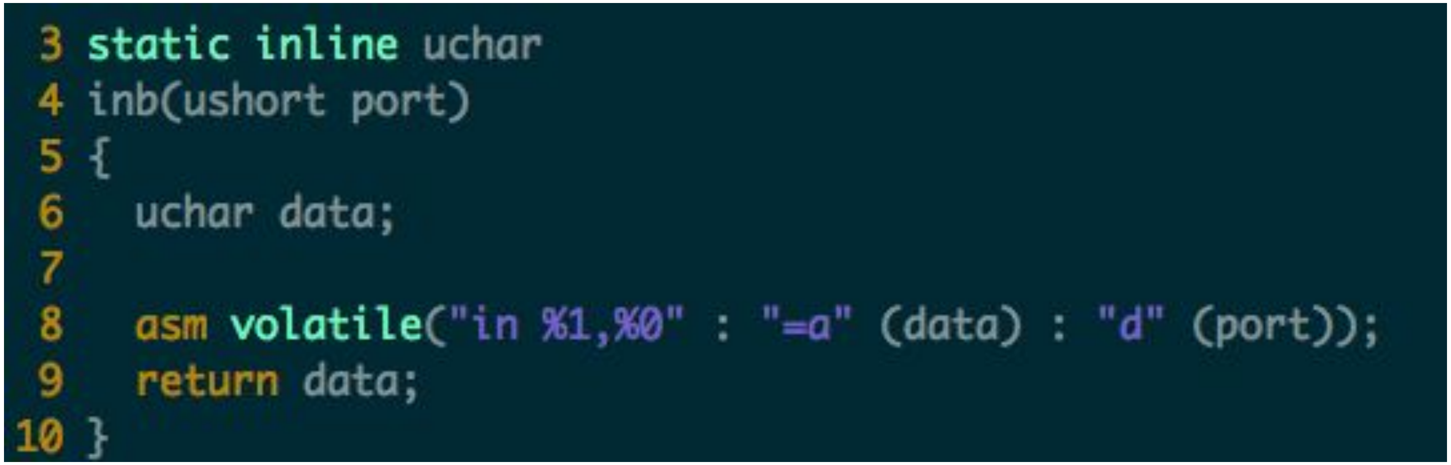
IDE Specification

- Xv6 uses a relative old IDE specification
  - Not PCI negotiation

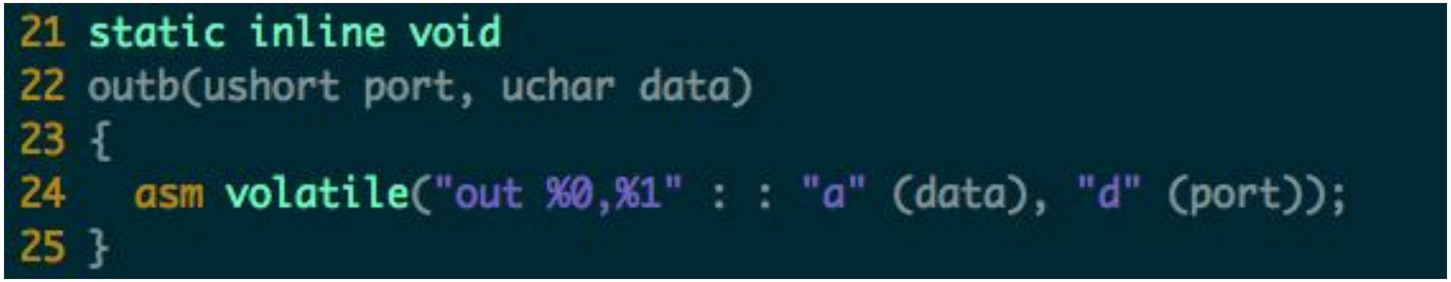
- IDE Ports
  - 0x1F0-0x1F7
    - 0x1f0: write/read port
    - 0x1f2: number of sectors
    - 0x1f3-0x1f5: sector number
    - 0x1f6: diskno and sector number
    - 0x1f7: command registers, status bit

  - 0x3F6 interrupt control line

I/O instruction in xv6

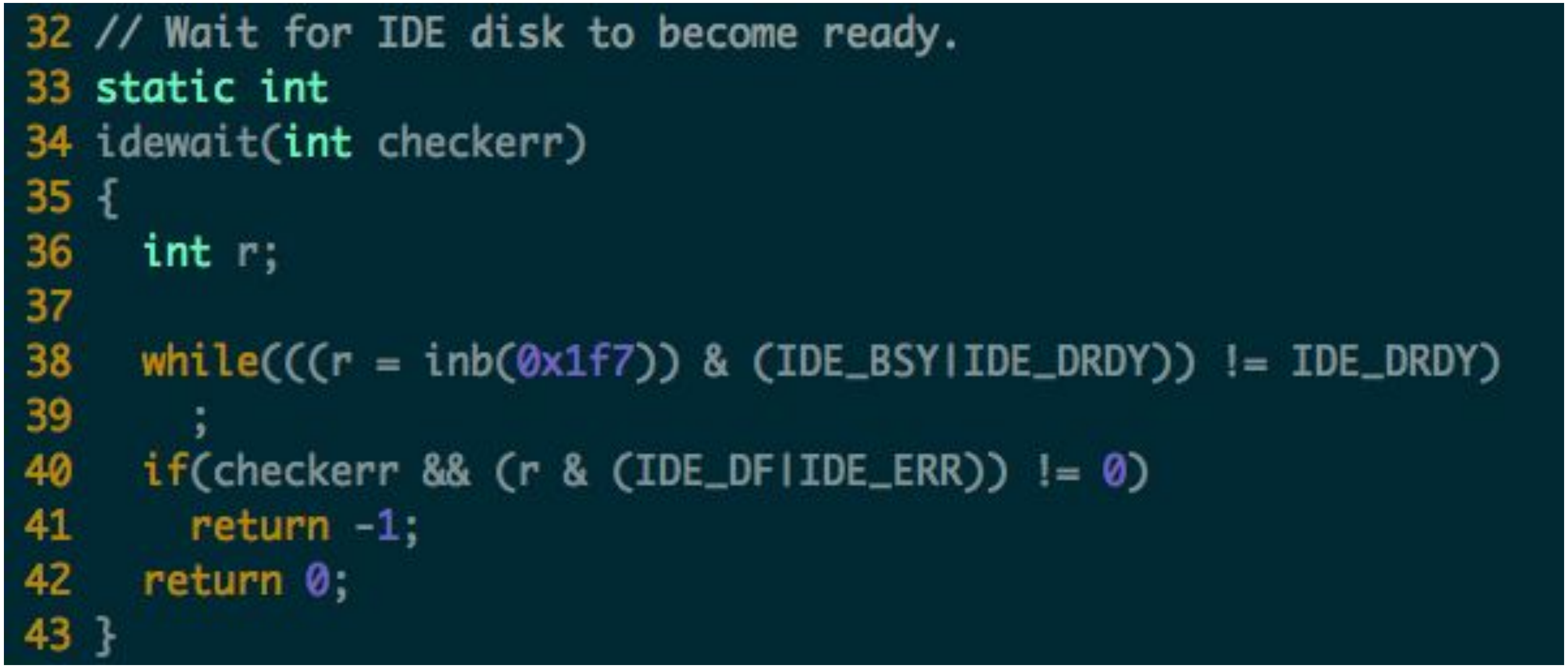


```c
static inline uchar
inb(ushort port)
{
  uchar data;

  asm volatile("in %1,%0" : "=a" (data) : "d" (port));
  return data;
}
```



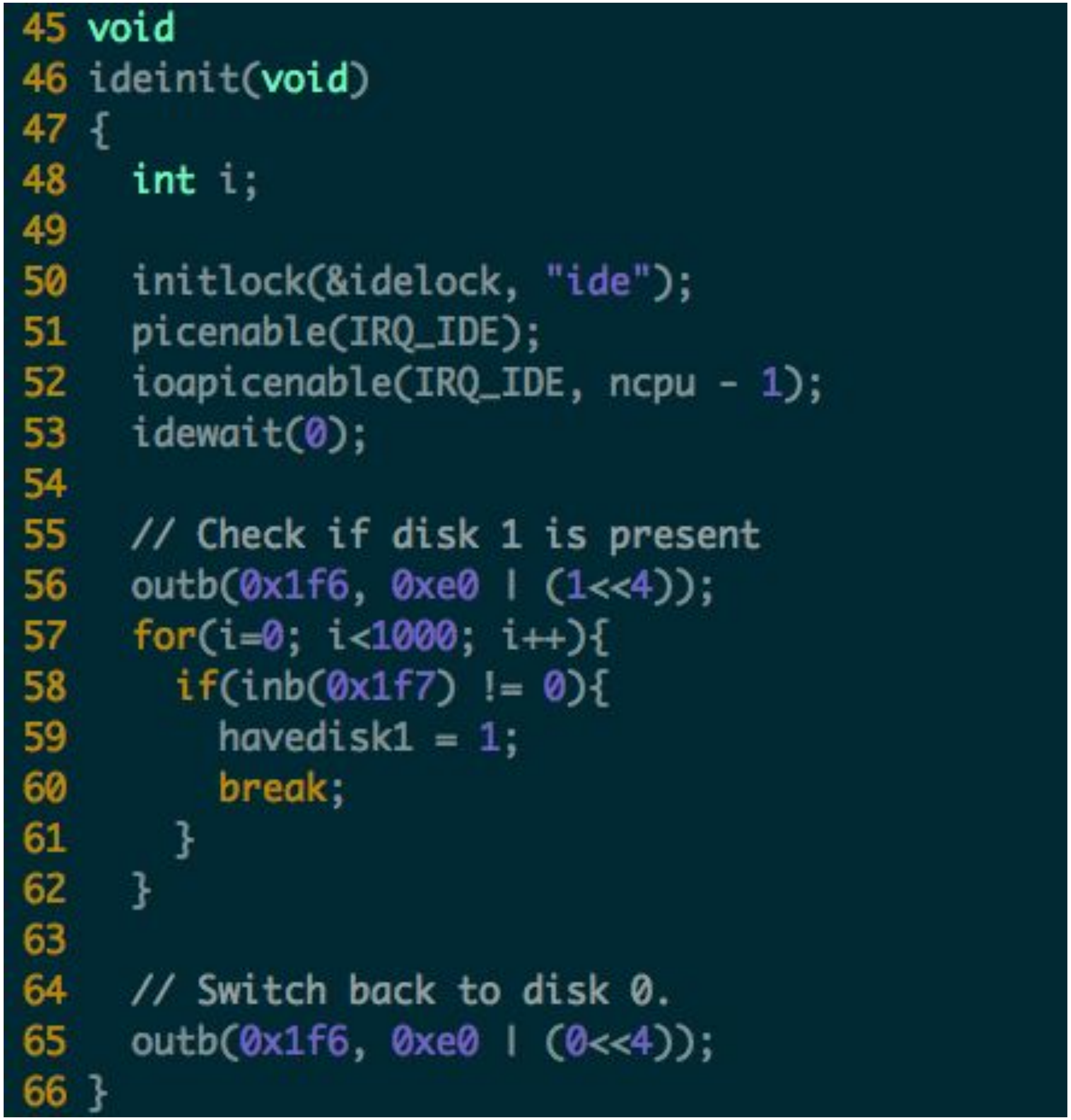
```c
static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}
```

Polling Mechanism



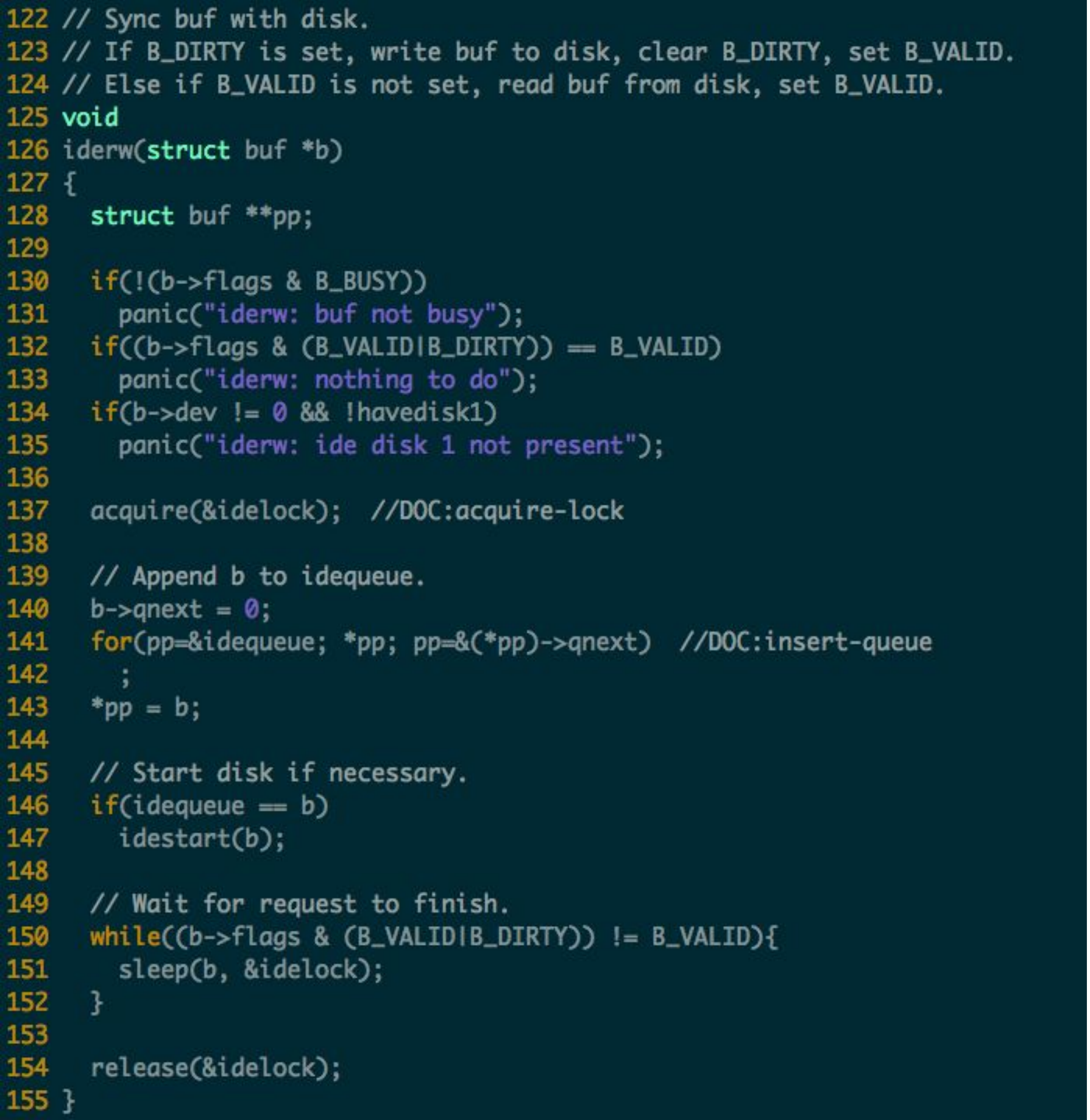
```c
// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
```

IDE Initialization



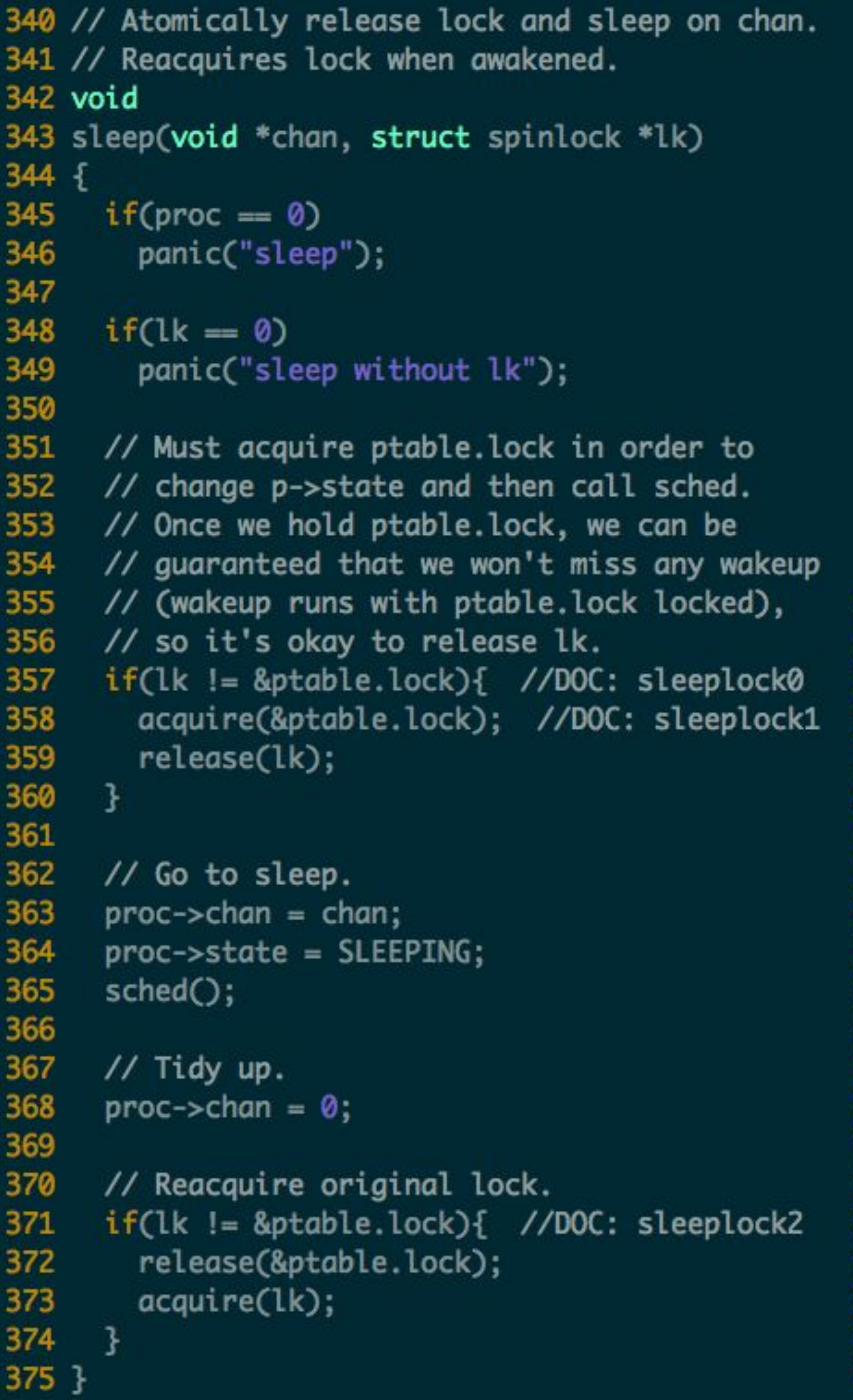
```c
void
ideinit(void)
{
  int i;

  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}
```

Synchronous I/O (iderw)

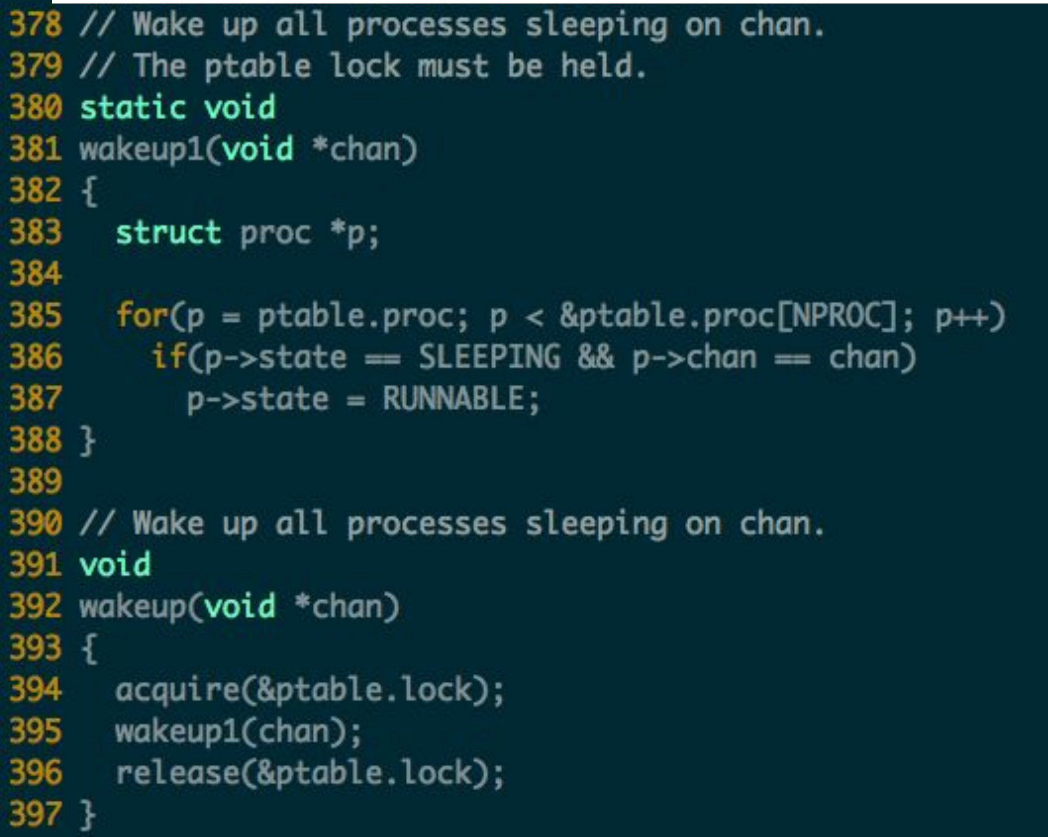


```
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);  //DOC:acquire-lock

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
}
```

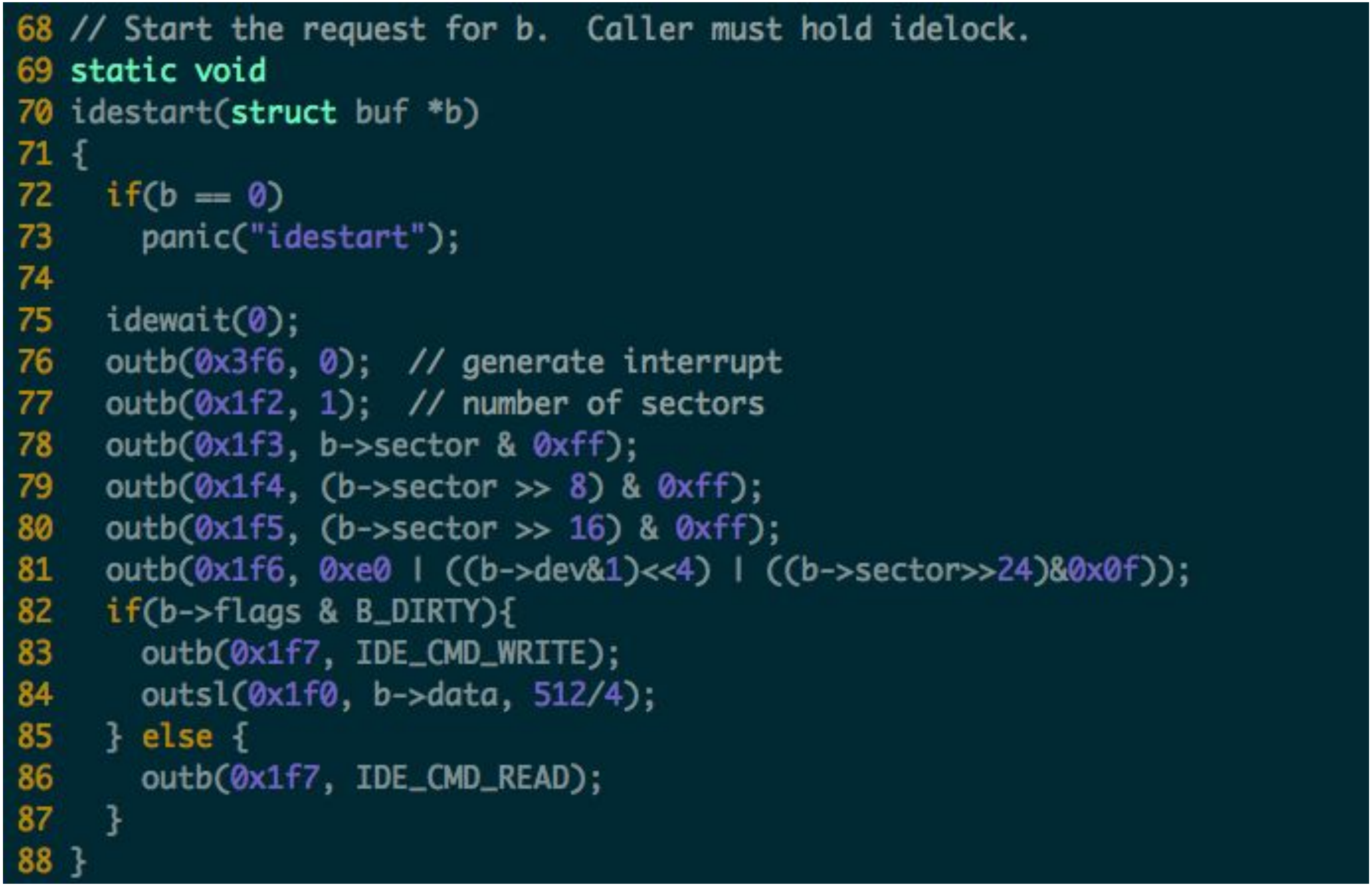
Sleep and Wakeup Mechanism in xv6



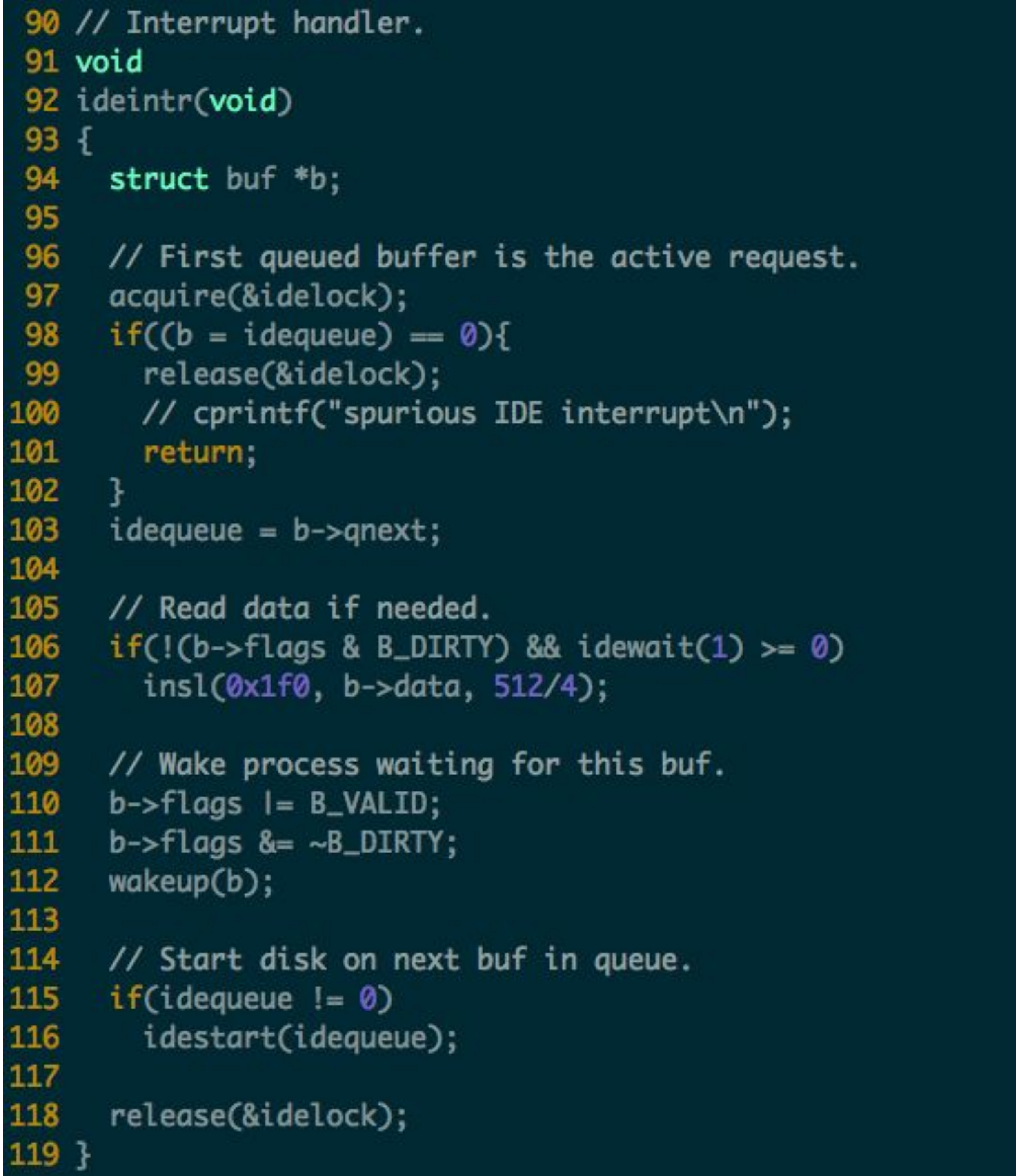
```c
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```



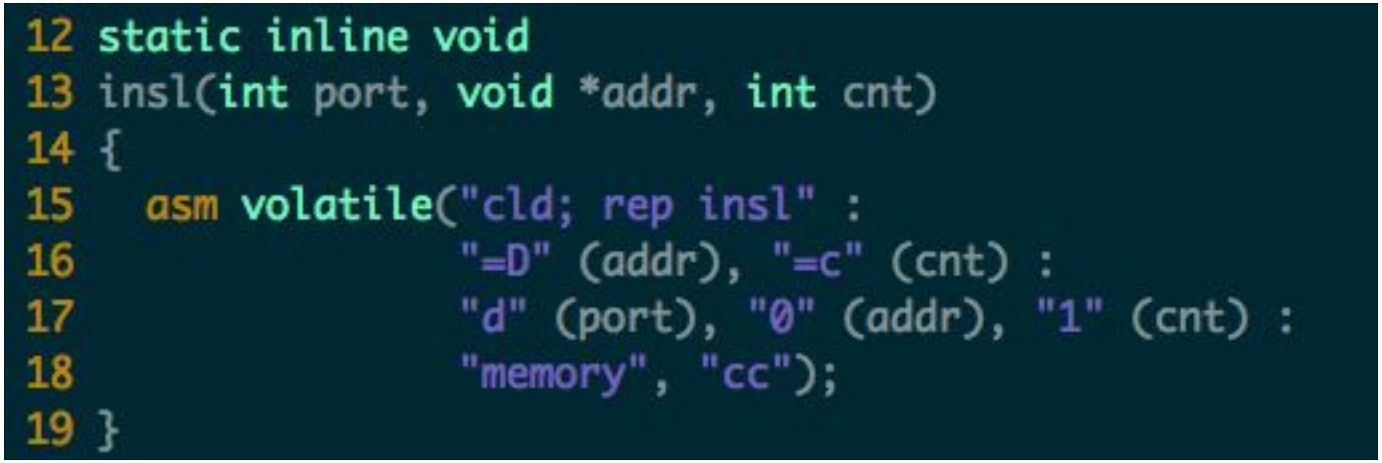
```c
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```



```
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```



```c
// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```



```
static inline void
insl(int port, void *addr, int cnt)
{
  asm volatile("cld; rep insl" :
               "=D" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "memory", "cc");
}
```

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
- iderw() maintains a queue of requests
  - Adds the buffer b to the end of the queue
  - If b is the first, then call idestart(), if not, just wait
  - Then just sleep, instead of polling
- ideintr() will handle the first request
  - Then start other request if any

I/O

Yubin Xia, Rong Chen

IPADS, SJTU

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

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
- Three types: block, char, network
- Blocking VS. non-blocking
- Sync VS. Async
- PIO, MMIO, DMA
- Top-half & bottom-half

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

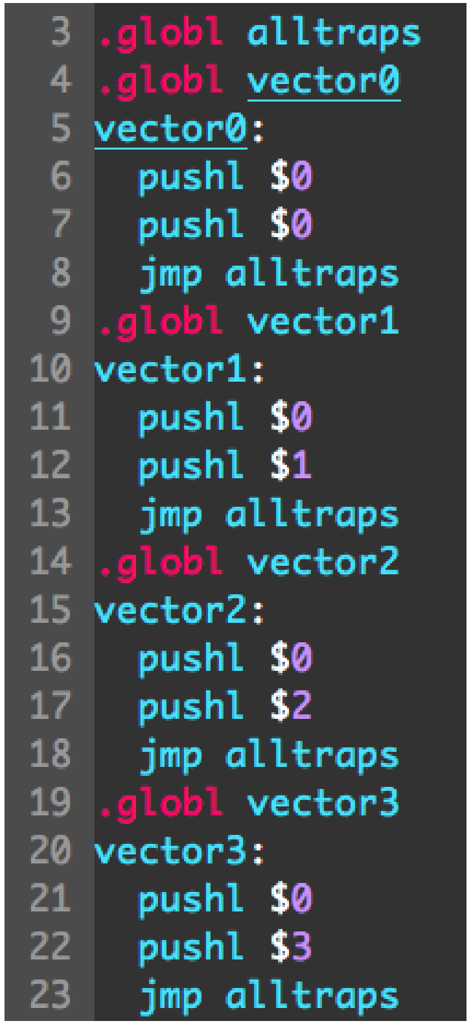
- CPU sets up interrupt handler vector before I/O
- CPU issues I/O request and continues other tasks
- I/O device processes the I/O request
- I/O device triggers CPU interrupt-request line
- Interrupt handler receives interrupts and dispatches to correct handler

Interrupt Driven

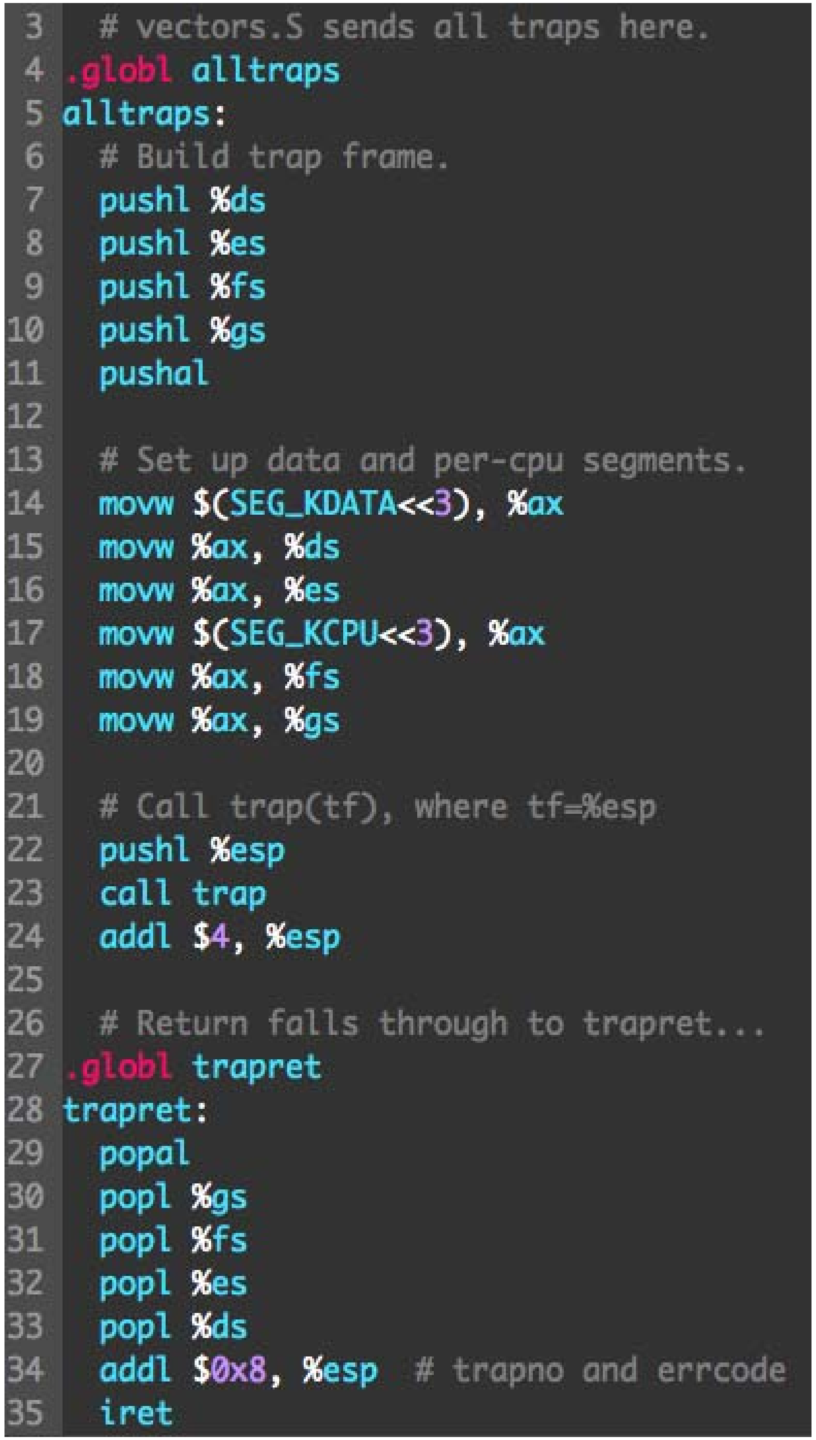
- Pro: handles unpredictable events well
- Con: interrupts relatively high overhead

- 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

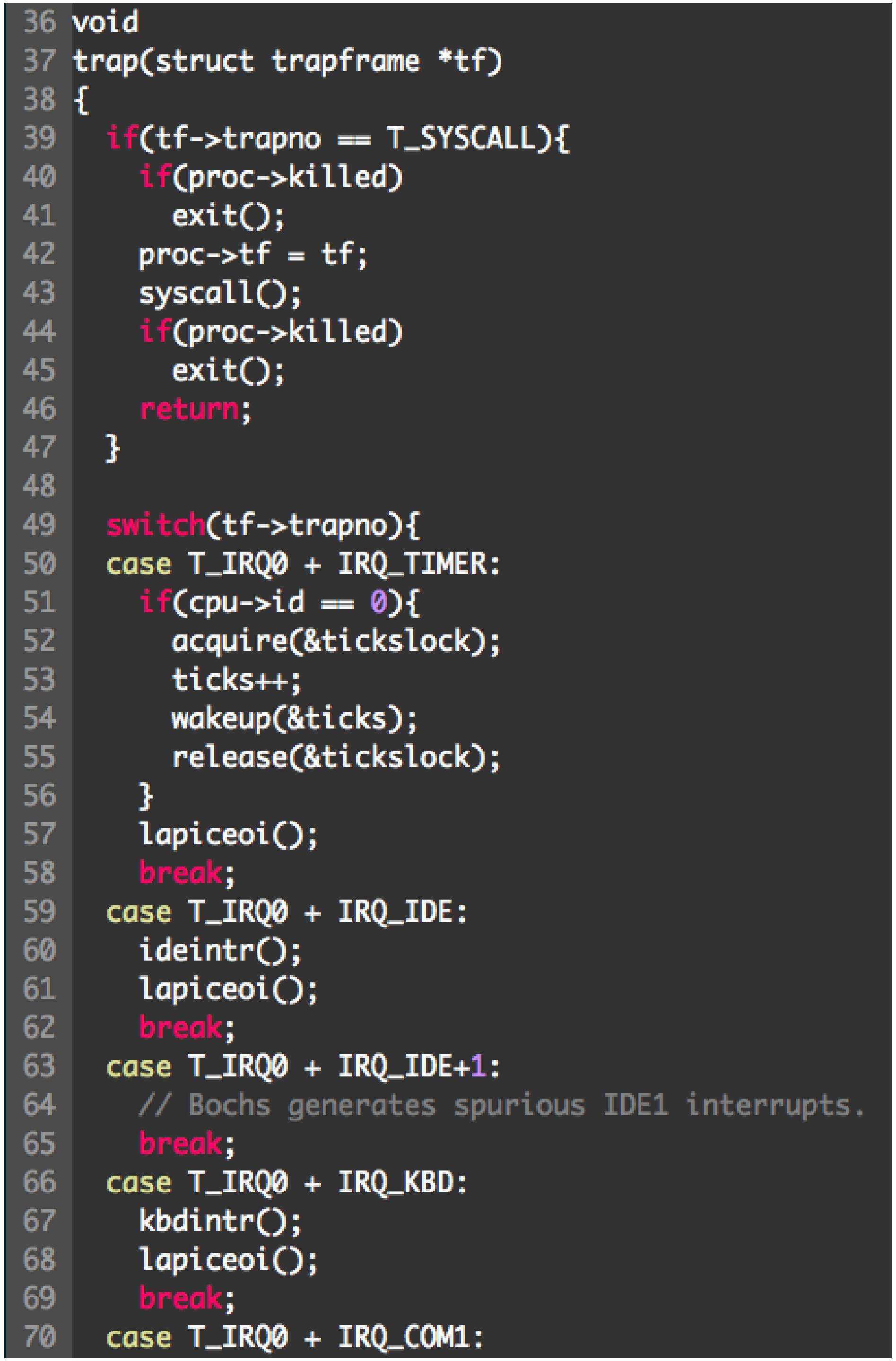
Press ‘a’ to display ‘a’



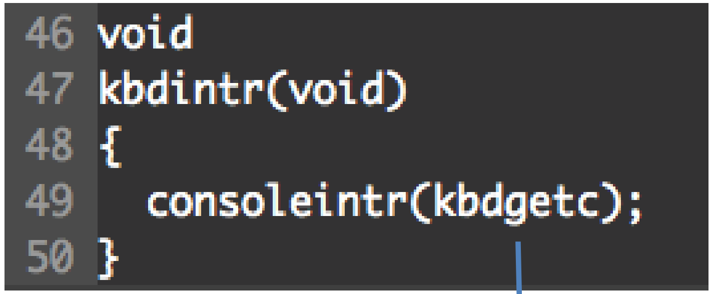
```
.globl alltraps
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
.globl vector1
vector1:
  pushl $0
  pushl $1
  jmp alltraps
.globl vector2
vector2:
  pushl $0
  pushl $2
  jmp alltraps
.globl vector3
vector3:
  pushl $0
  pushl $3
  jmp alltraps
```



```
  # vectors.S sends all traps here.
.globl alltraps
alltraps:
  # Build trap frame.
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal

  # Set up data and per-cpu segments.
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds
  movw %ax, %es
  movw $(SEG_KCPU<<3), %ax
  movw %ax, %fs
  movw %ax, %gs

  # Call trap(tf), where tf=%esp
  pushl %esp
  call trap
  addl $4, %esp

  # Return falls through to trapret...
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

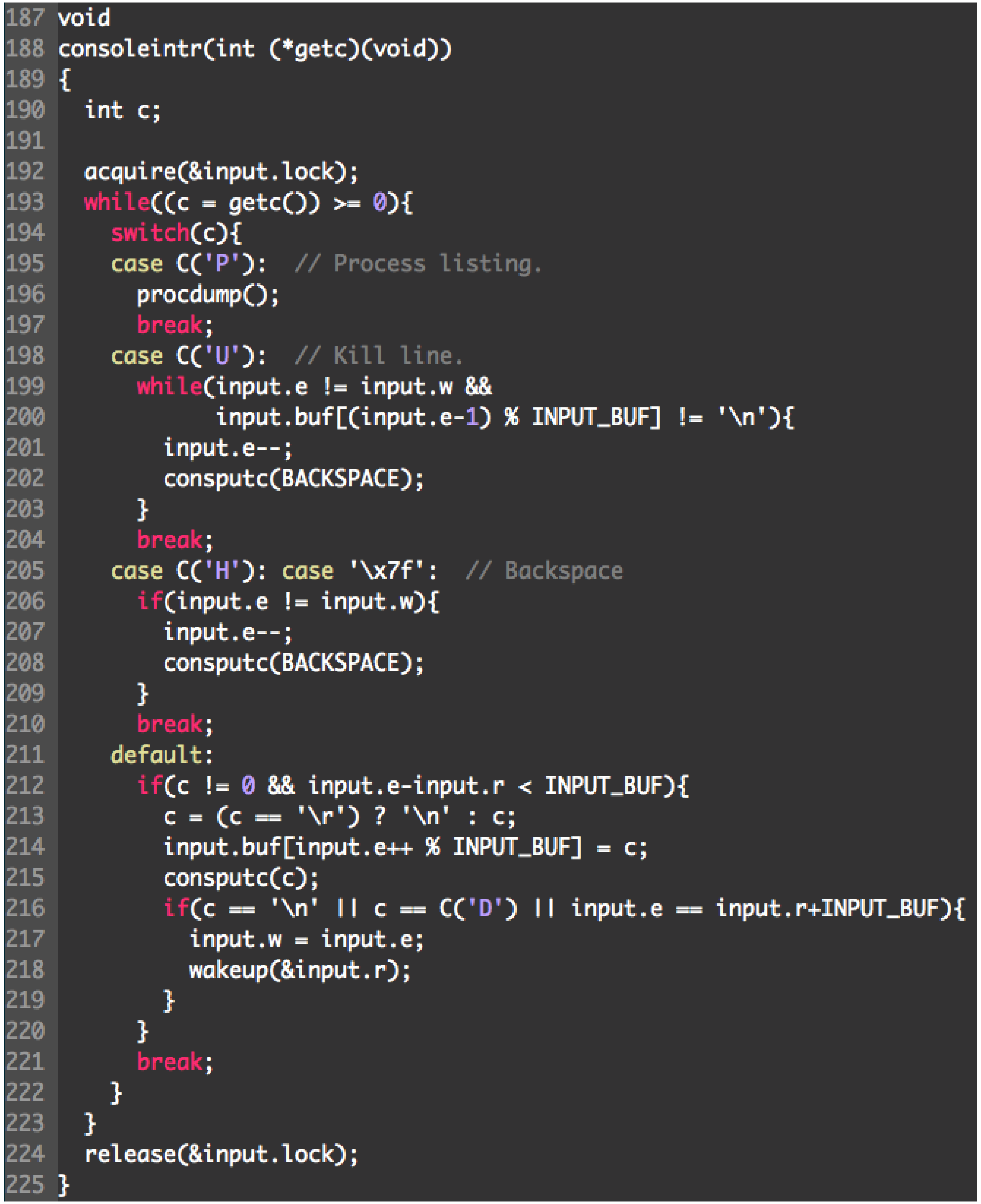


```c
void
trap(struct trapframe *tf)
{
  if(tf->trapno == T_SYSCALL){
    if(proc->killed)
      exit();
    proc->tf = tf;
    syscall();
    if(proc->killed)
      exit();
    return;
  }

  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    if(cpu->id == 0){
      acquire(&tickslock);
      ticks++;
      wakeup(&ticks);
      release(&tickslock);
    }
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE:
    ideintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_IDE+1:
    // Bochs generates spurious IDE1 interrupts.
    break;
  case T_IRQ0 + IRQ_KBD:
    kbdintr();
    lapiceoi();
    break;
  case T_IRQ0 + IRQ_COM1:
```

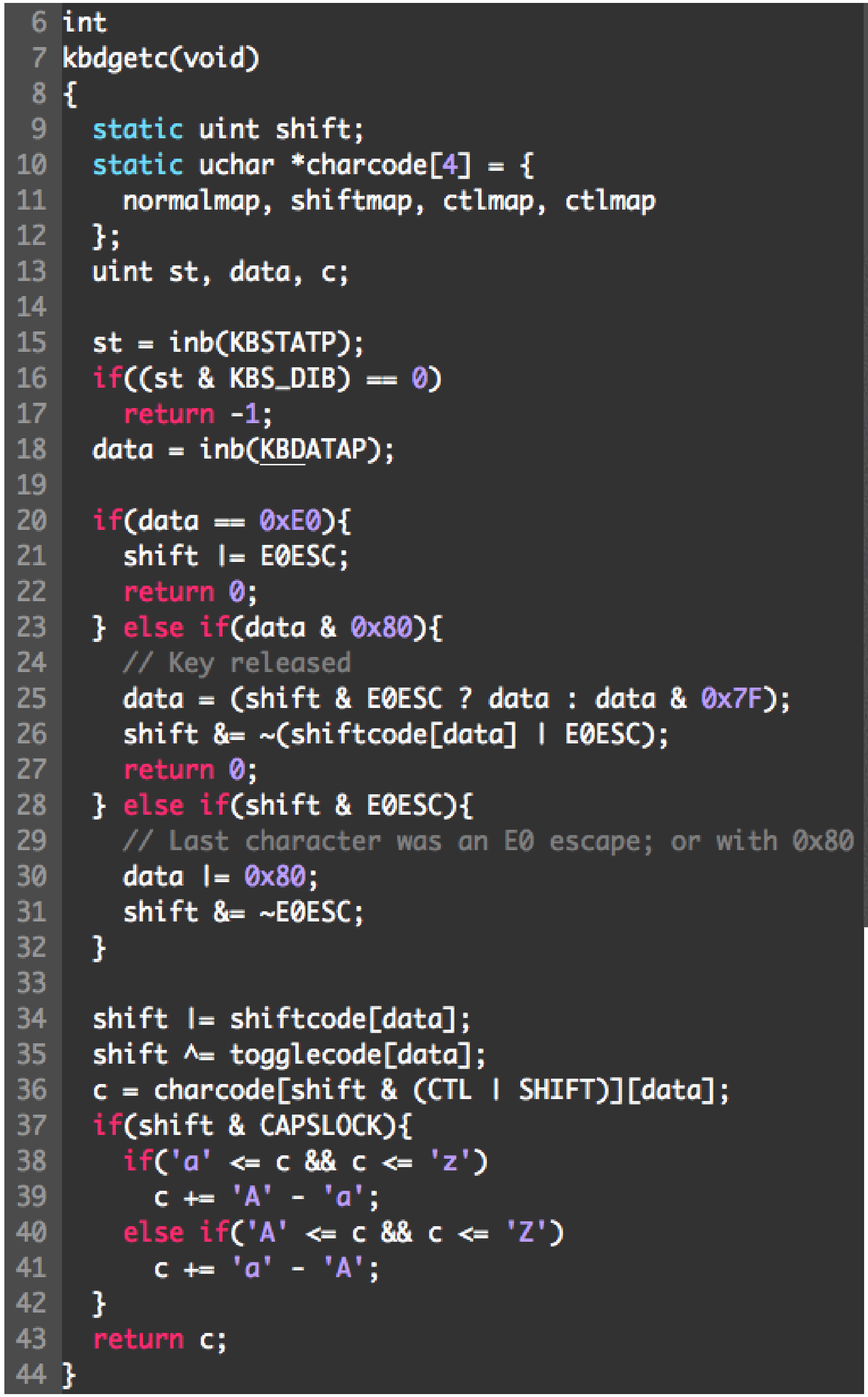


```c
void
kbdintr(void)
{
  consoleintr(kbdgetc);
}
```

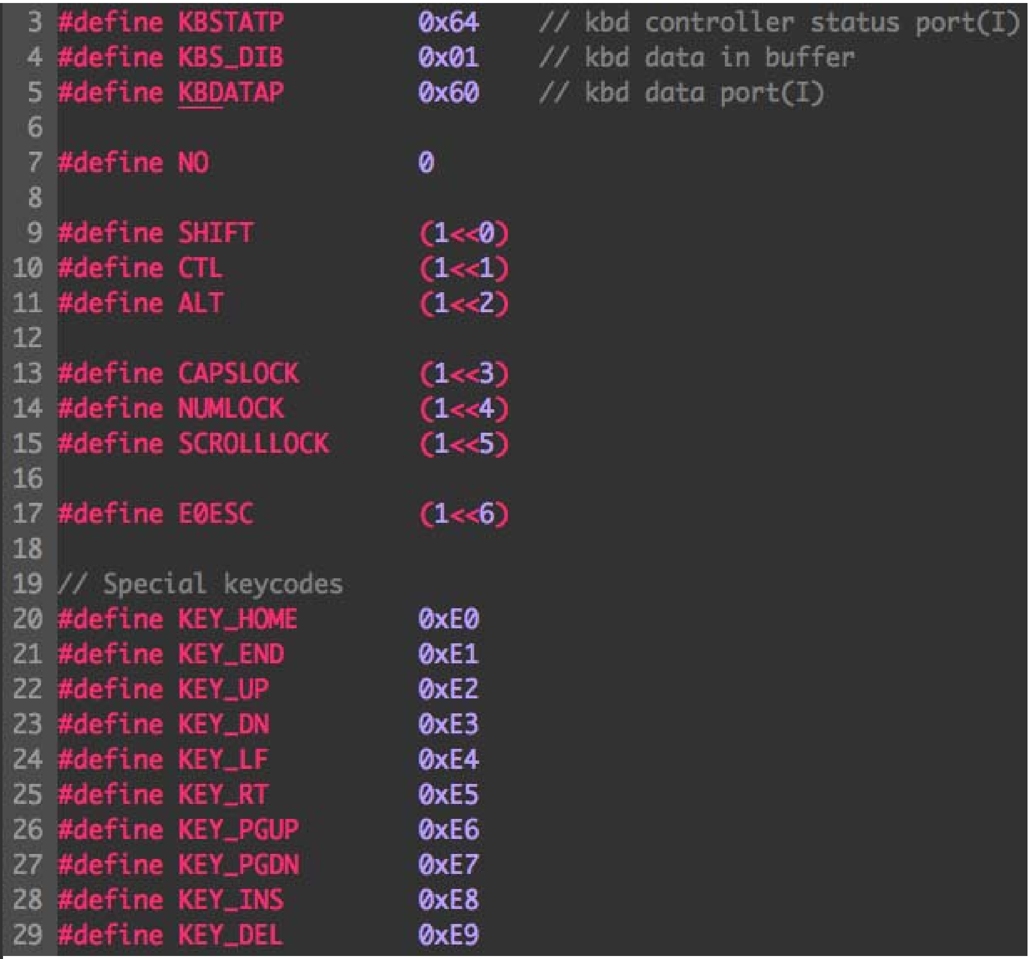
Function pointer



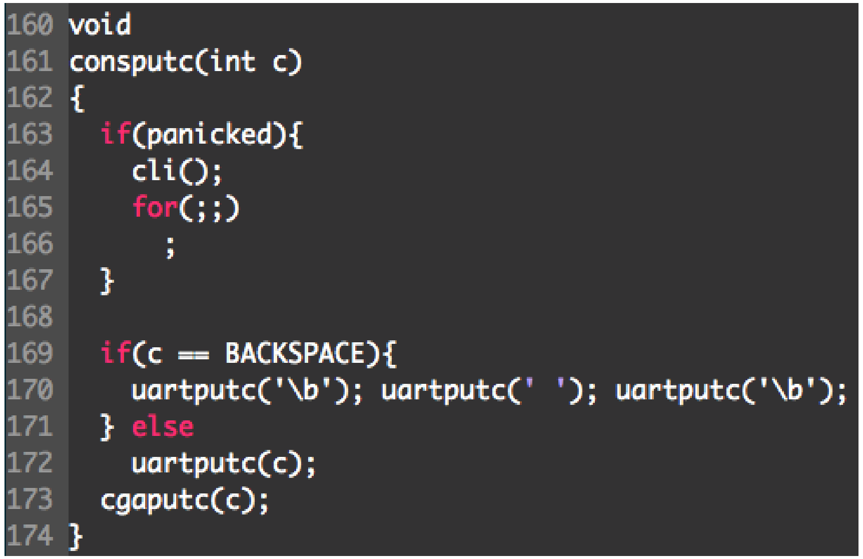
```c
void
consoleintr(int (*getc)(void))
{
  int c;

  acquire(&input.lock);
  while((c = getc()) >= 0){
    switch(c){
    case C('P'):  // Process listing.
      procdump();
      break;
    case C('U'):  // Kill line.
      while(input.e != input.w &&
            input.buf[(input.e-1) % INPUT_BUF] != '\n'){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    case C('H'): case '\x7f':  // Backspace
      if(input.e != input.w){
        input.e--;
        consputc(BACKSPACE);
      }
      break;
    default:
      if(c != 0 && input.e-input.r < INPUT_BUF){
        c = (c == '\r') ? '\n' : c;
        input.buf[input.e++ % INPUT_BUF] = c;
        consputc(c);
        if(c == '\n' || c == C('D') || input.e == input.r+INPUT_BUF){
          input.w = input.e;
          wakeup(&input.r);
        }
      }
      break;
    }
  }
  release(&input.lock);
}
```



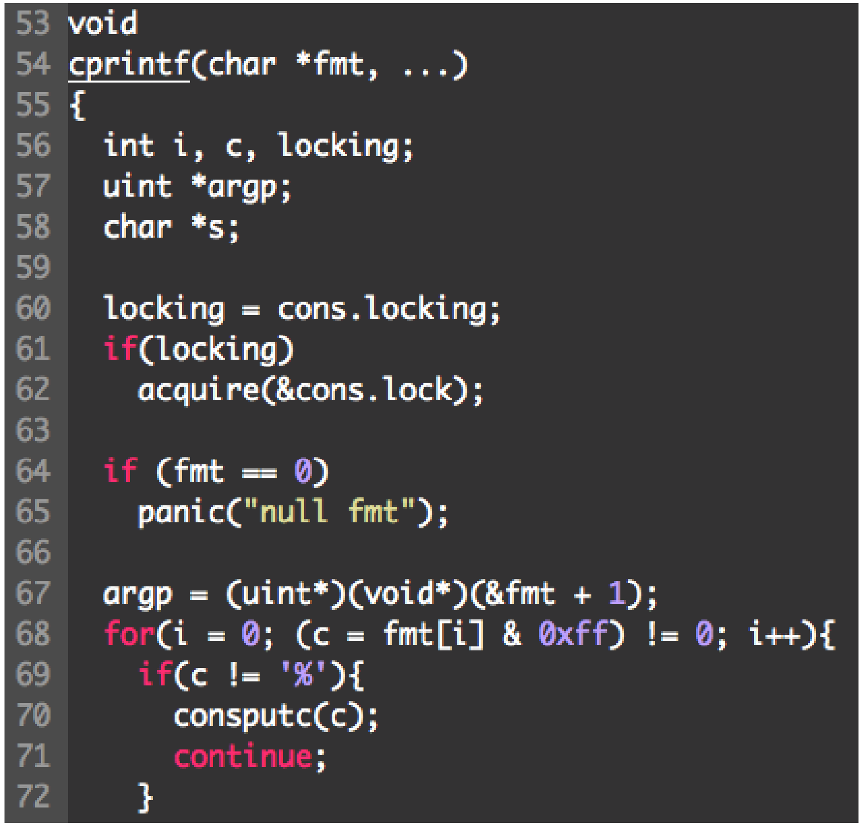
```
int
kbdgetc(void)
{
  static uint shift;
  static uchar *charcode[4] = {
    normalmap, shiftmap, ctlmap, ctlmap
  };
  uint st, data, c;

  st = inb(KBSTATP);
  if((st & KBS_DIB) == 0)
    return -1;
  data = inb(KBDATAP);

  if(data == 0xE0){
    shift |= E0ESC;
    return 0;
  } else if(data & 0x80){
    // Key released
    data = (shift & E0ESC ? data : data & 0x7F);
    shift &= ~(shiftcode[data] | E0ESC);
    return 0;
  } else if(shift & E0ESC){
    // Last character was an E0 escape; or with 0x80
    data |= 0x80;
    shift &= ~E0ESC;
  }

  shift |= shiftcode[data];
  shift ^= togglecode[data];
  c = charcode[shift & (CTL | SHIFT)][data];
  if(shift & CAPSLOCK){
    if('a' <= c && c <= 'z')
      c += 'A' - 'a';
    else if('A' <= c && c <= 'Z')
      c += 'a' - 'A';
  }
  return c;
}
```



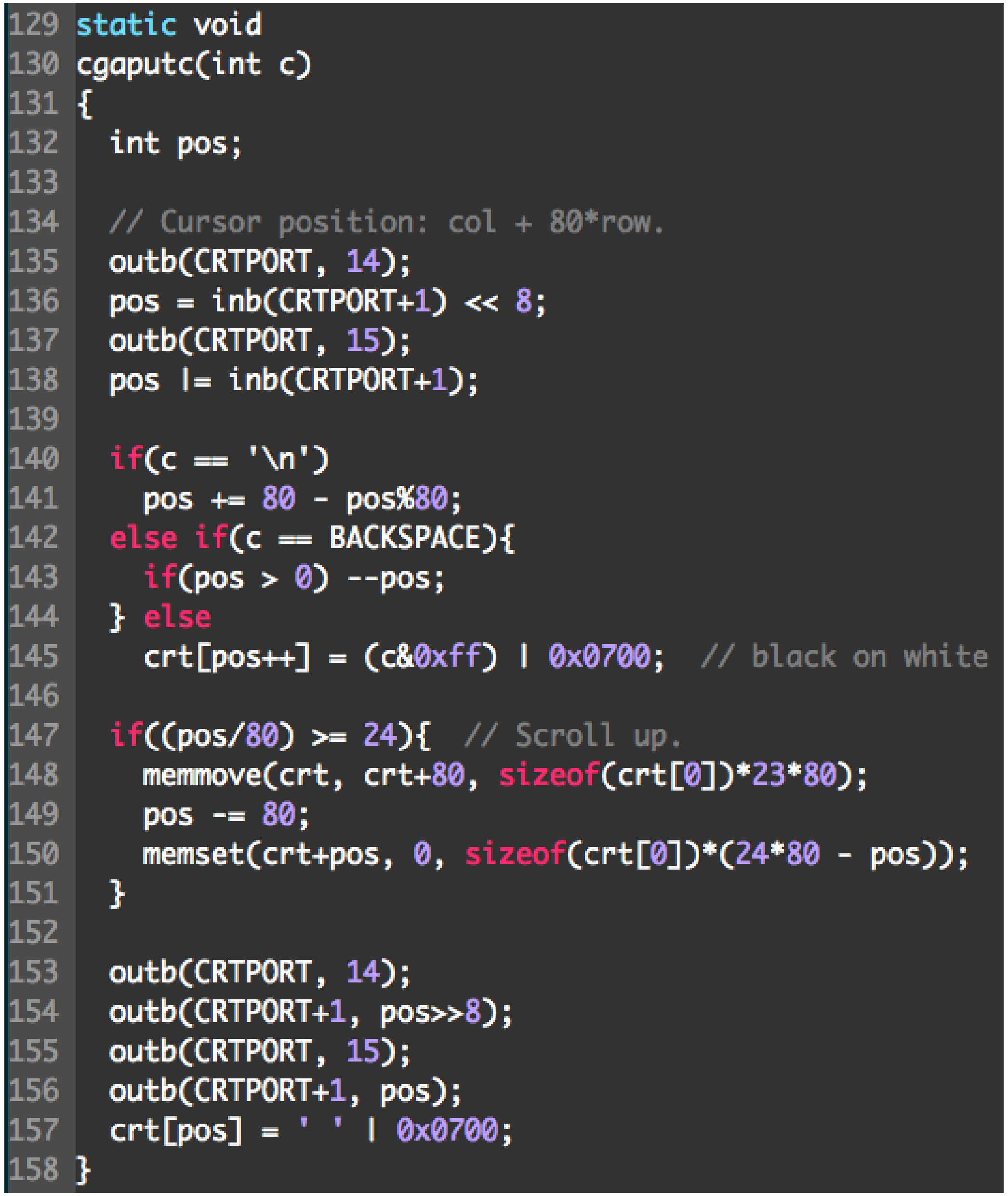
```
#define KBSTATP         0x64    // kbd controller status port(I)
#define KBS_DIB         0x01    // kbd data in buffer
#define KBDATAP         0x60    // kbd data port(I)

#define NO              0

#define SHIFT           (1<<0)
#define CTL             (1<<1)
#define ALT             (1<<2)

#define CAPSLOCK        (1<<3)
#define NUMLOCK         (1<<4)
#define SCROLLLOCK      (1<<5)

#define E0ESC           (1<<6)

// Special keycodes
#define KEY_HOME        0xE0
#define KEY_END         0xE1
#define KEY_UP          0xE2
#define KEY_DN          0xE3
#define KEY_LF          0xE4
#define KEY_RT          0xE5
#define KEY_PGUP        0xE6
#define KEY_PGDN        0xE7
#define KEY_INS         0xE8
#define KEY_DEL         0xE9
```



```
void
consputc(int c)
{
  if(panicked){
    cli();
    for(;;)
      ;
  }

  if(c == BACKSPACE){
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else
    uartputc(c);
  cgaputc(c);
}
```



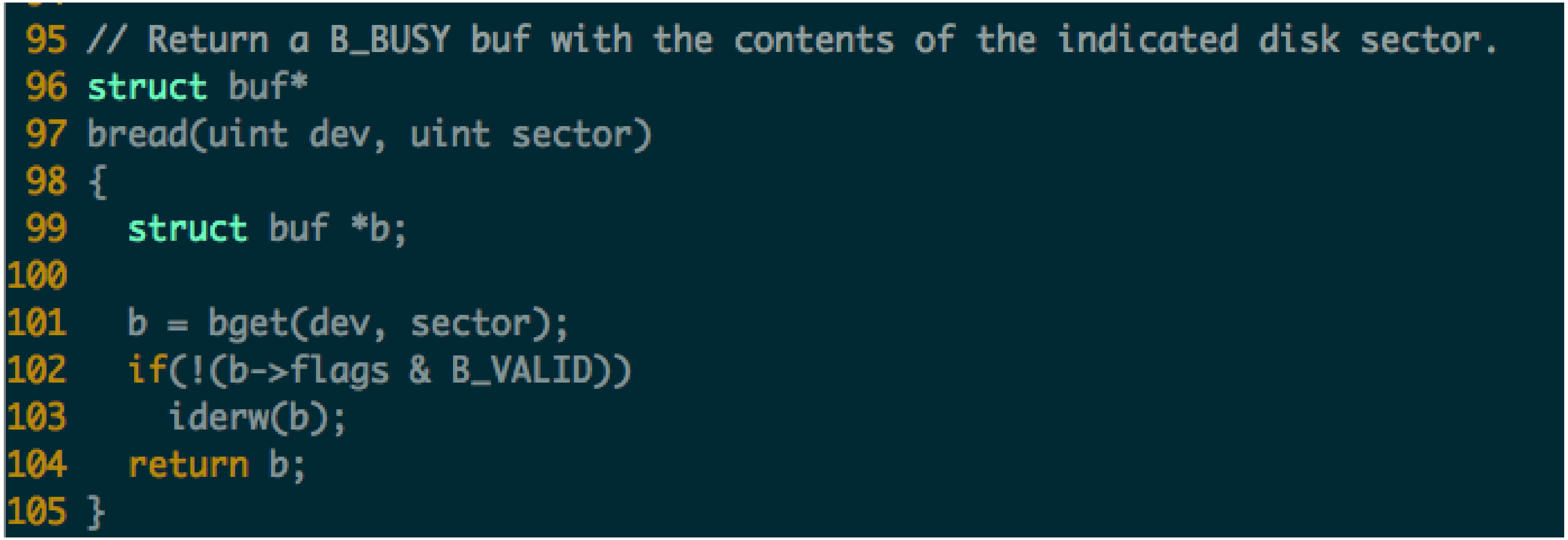
```
void
cprintf(char *fmt, ...)
{
  int i, c, locking;
  uint *argp;
  char *s;

  locking = cons.locking;
  if(locking)
    acquire(&cons.lock);

  if (fmt == 0)
    panic("null fmt");

  argp = (uint*)(void*)(&fmt + 1);
  for(i = 0; (c = fmt[i] & 0xff) != 0; i++){
    if(c != '%'){
      consputc(c);
      continue;
    }
```



```
static void
cgaputc(int c)
{
  int pos;

  // Cursor position: col + 80*row.
  outb(CRTPORT, 14);
  pos = inb(CRTPORT+1) << 8;
  outb(CRTPORT, 15);
  pos |= inb(CRTPORT+1);

  if(c == '\n')
    pos += 80 - pos%80;
  else if(c == BACKSPACE){
    if(pos > 0) --pos;
  } else
    crt[pos++] = (c&0xff) | 0x0700;  // black on white

  if((pos/80) >= 24){  // Scroll up.
    memmove(crt, crt+80, sizeof(crt[0])*23*80);
    pos -= 80;
    memset(crt+pos, 0, sizeof(crt[0])*(24*80 - pos));
  }

  outb(CRTPORT, 14);
  outb(CRTPORT+1, pos>>8);
  outb(CRTPORT, 15);
  outb(CRTPORT+1, pos);
  crt[pos] = ' ' | 0x0700;
}
```

Recall : read

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



```
// Return a B_BUSY buf with the contents of the indicated disk sector.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
}
```

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

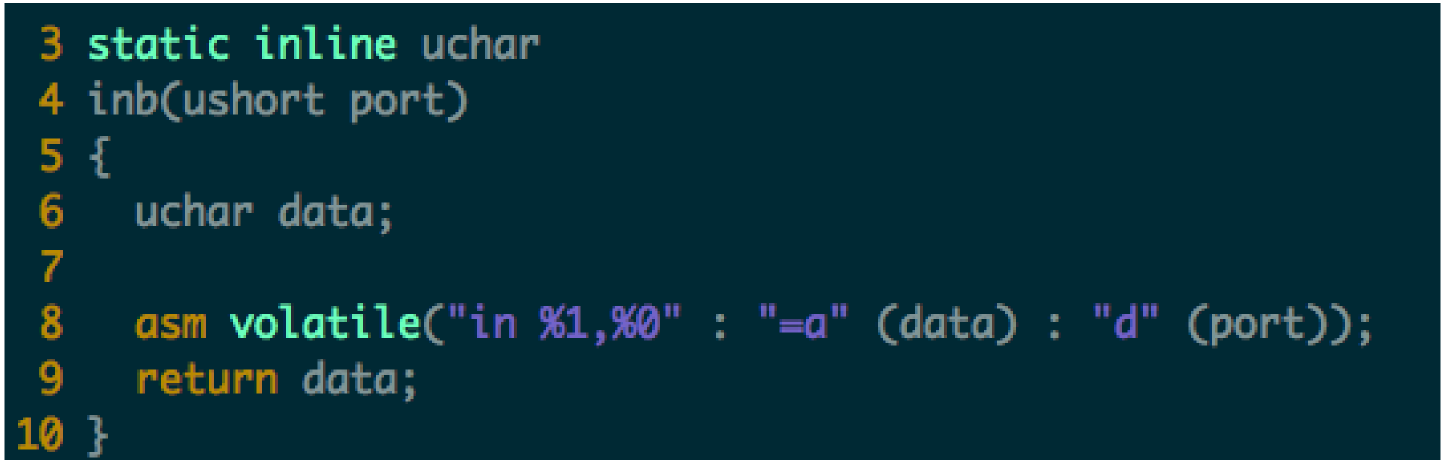
IDE Specification

- Xv6 uses a relative old IDE specification
  - Not PCI negotiation

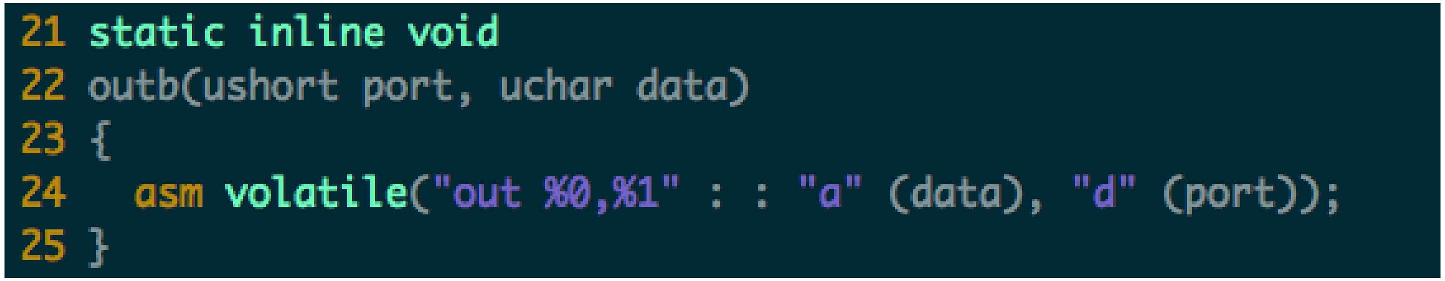
- IDE Ports
  - 0x1F0-0x1F7
    - 0x1f0: write/read port
    - 0x1f2: number of sectors
    - 0x1f3-0x1f5: sector number
    - 0x1f6: diskno and sector number
    - 0x1f7: command registers, status bit

  - 0x3F6 interrupt control line

I/O instruction in xv6

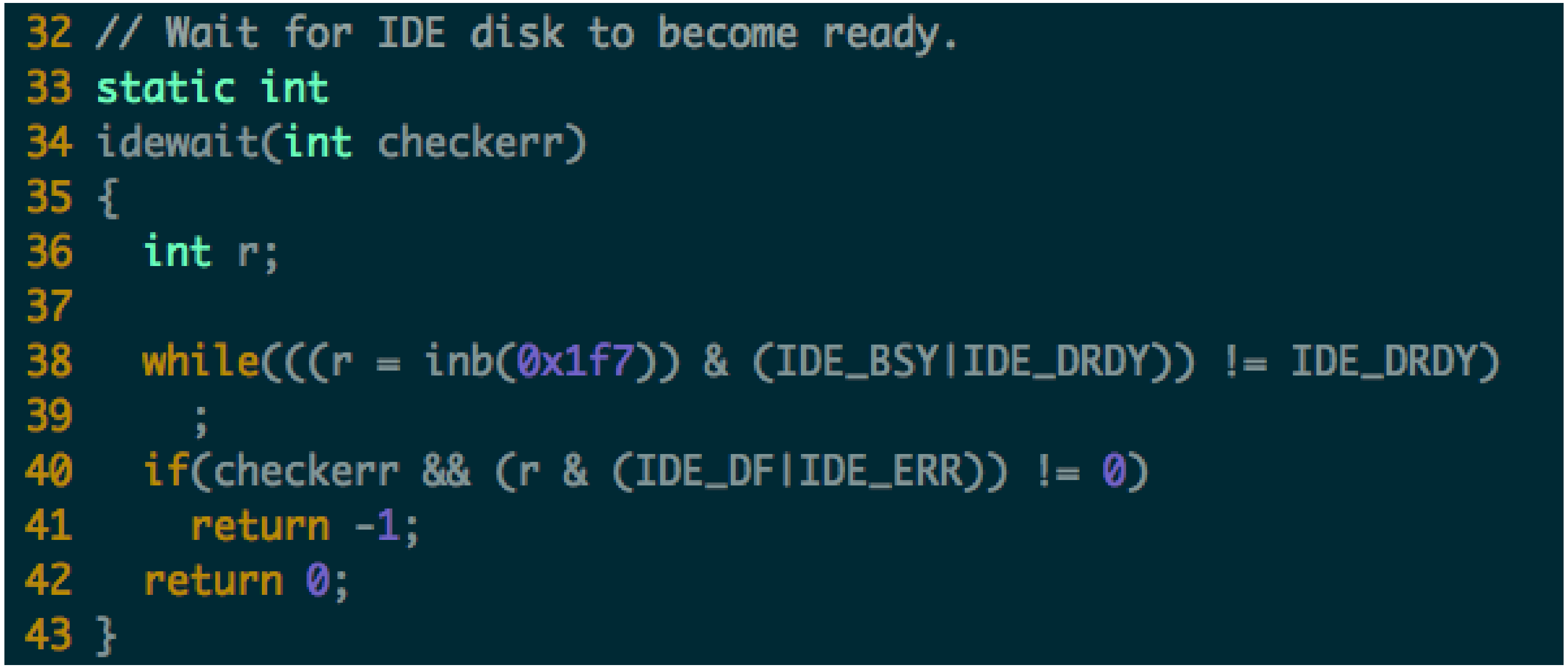


```c
static inline uchar
inb(ushort port)
{
  uchar data;

  asm volatile("in %1,%0" : "=a" (data) : "d" (port));
  return data;
}
```



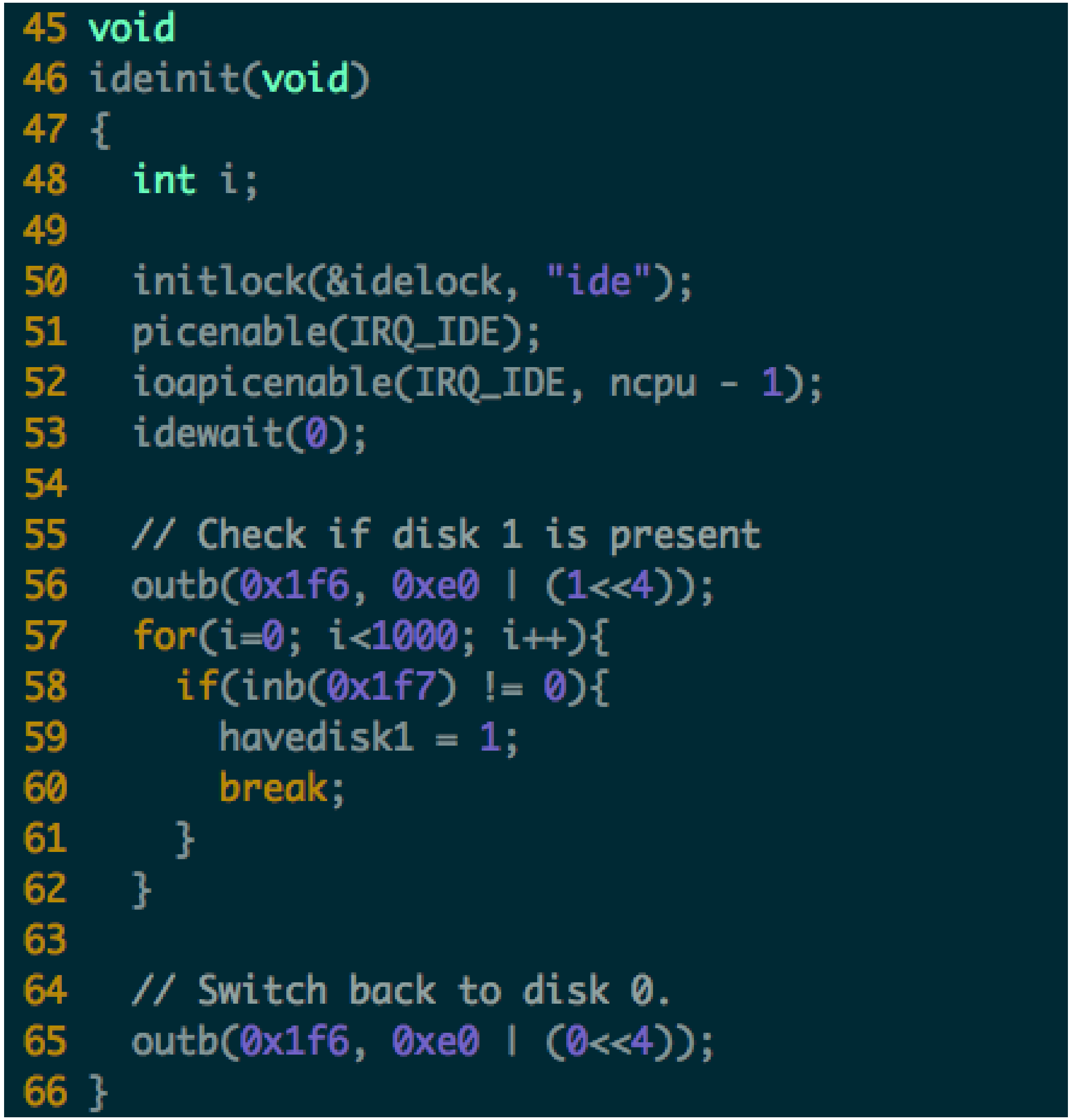
```c
static inline void
outb(ushort port, uchar data)
{
  asm volatile("out %0,%1" : : "a" (data), "d" (port));
}
```

Polling Mechanism



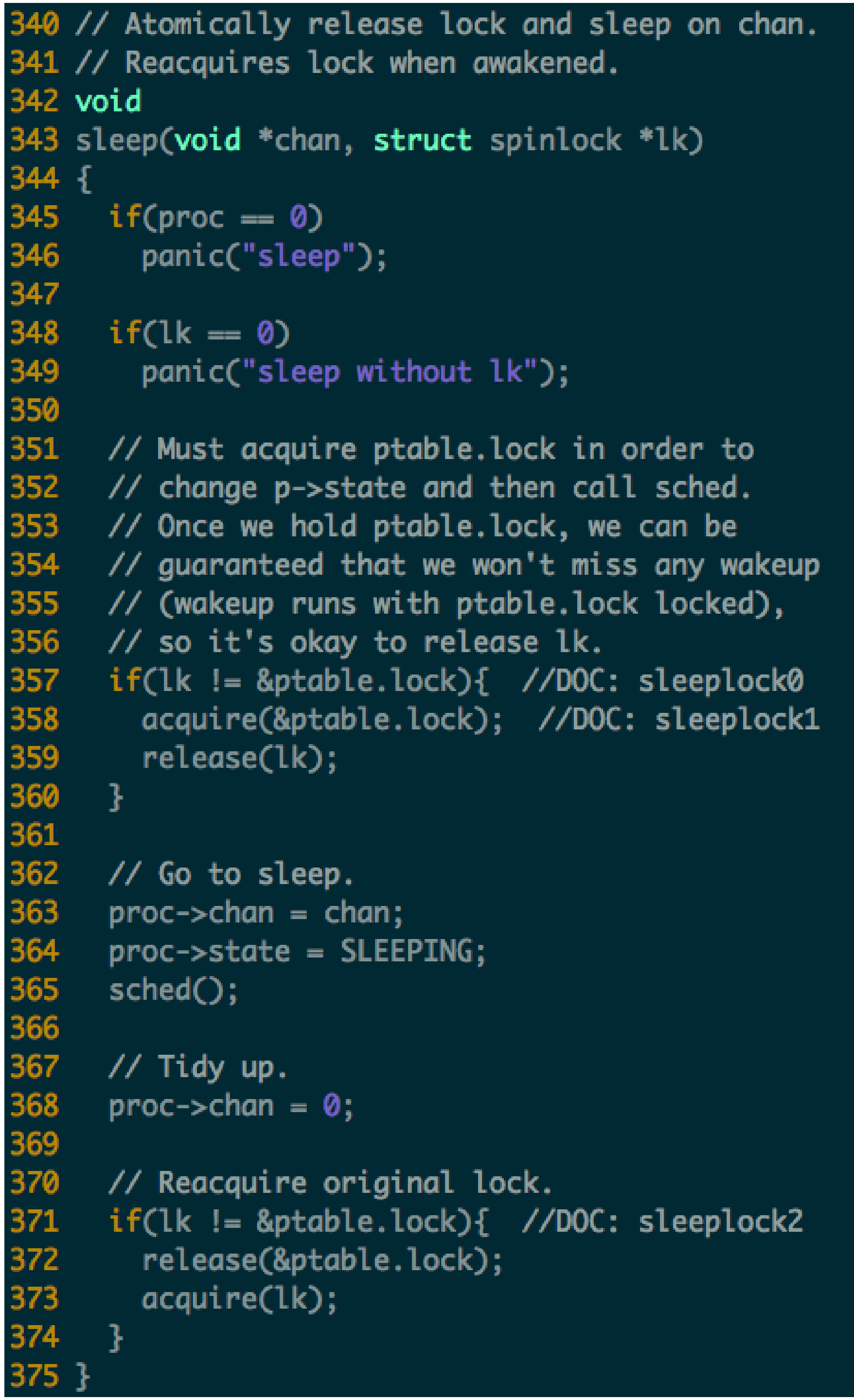
```
// Wait for IDE disk to become ready.
static int
idewait(int checkerr)
{
  int r;

  while(((r = inb(0x1f7)) & (IDE_BSY|IDE_DRDY)) != IDE_DRDY)
    ;
  if(checkerr && (r & (IDE_DF|IDE_ERR)) != 0)
    return -1;
  return 0;
}
```

IDE Initialization

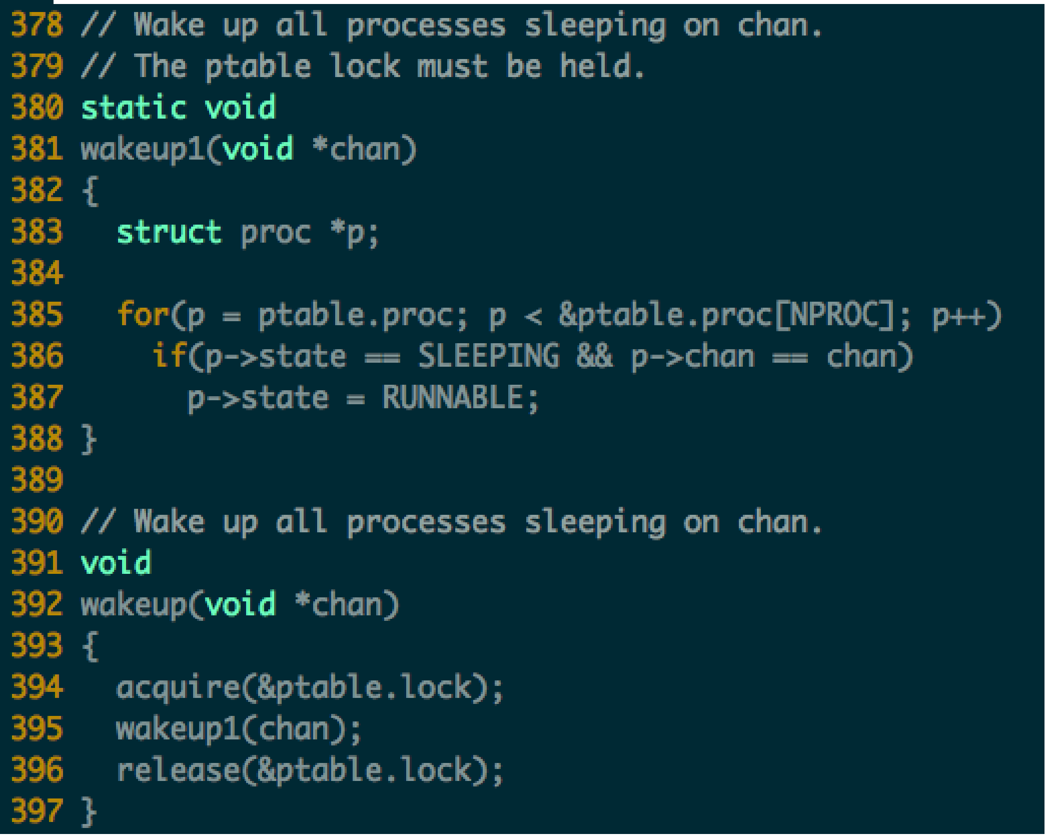


```c
void
ideinit(void)
{
  int i;

  initlock(&idelock, "ide");
  picenable(IRQ_IDE);
  ioapicenable(IRQ_IDE, ncpu - 1);
  idewait(0);

  // Check if disk 1 is present
  outb(0x1f6, 0xe0 | (1<<4));
  for(i=0; i<1000; i++){
    if(inb(0x1f7) != 0){
      havedisk1 = 1;
      break;
    }
  }

  // Switch back to disk 0.
  outb(0x1f6, 0xe0 | (0<<4));
}
```

Sleep and Wakeup Mechanism in xv6

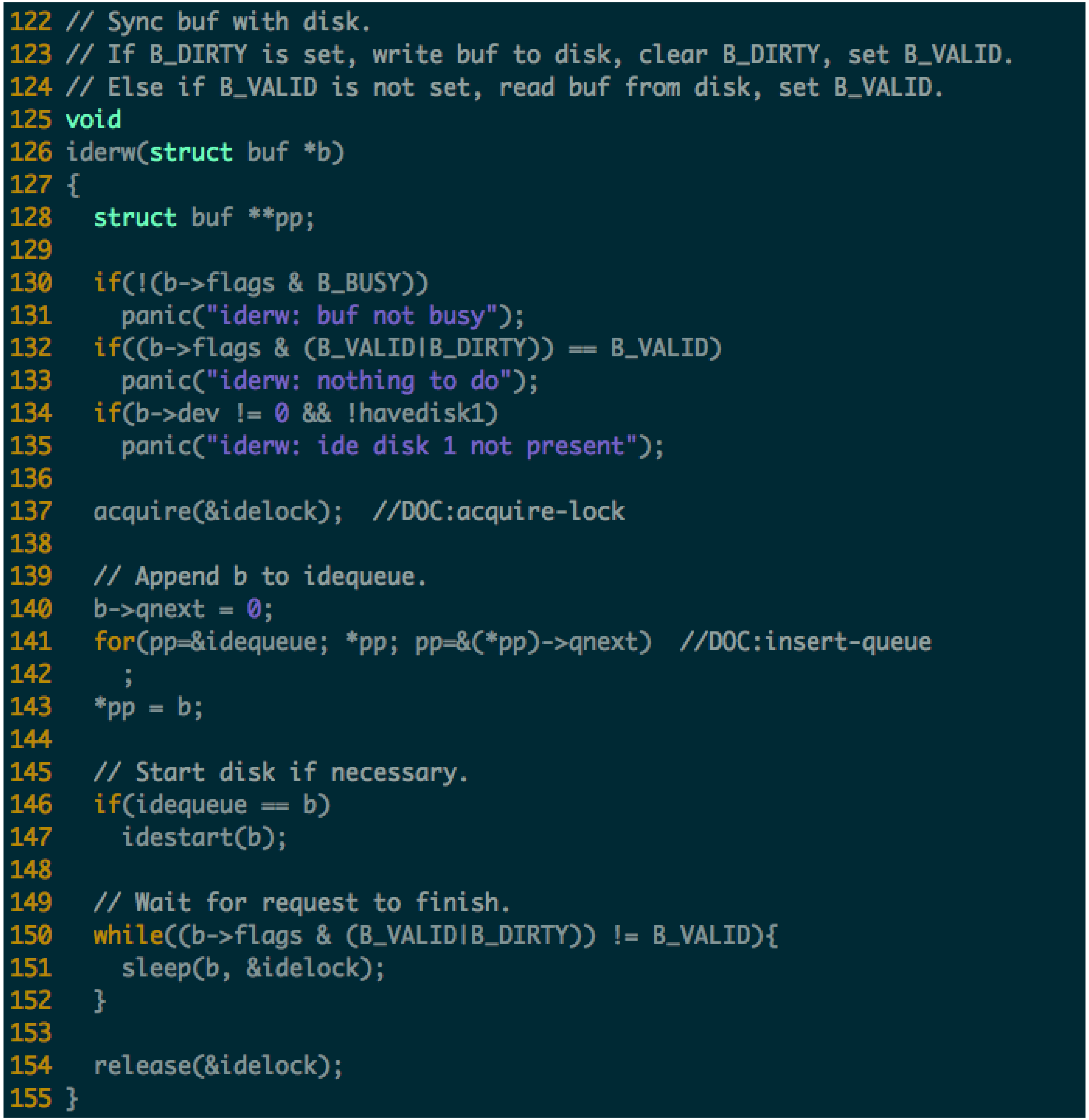


```
// Atomically release lock and sleep on chan.
// Reacquires lock when awakened.
void
sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  // Go to sleep.
  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  // Tidy up.
  proc->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

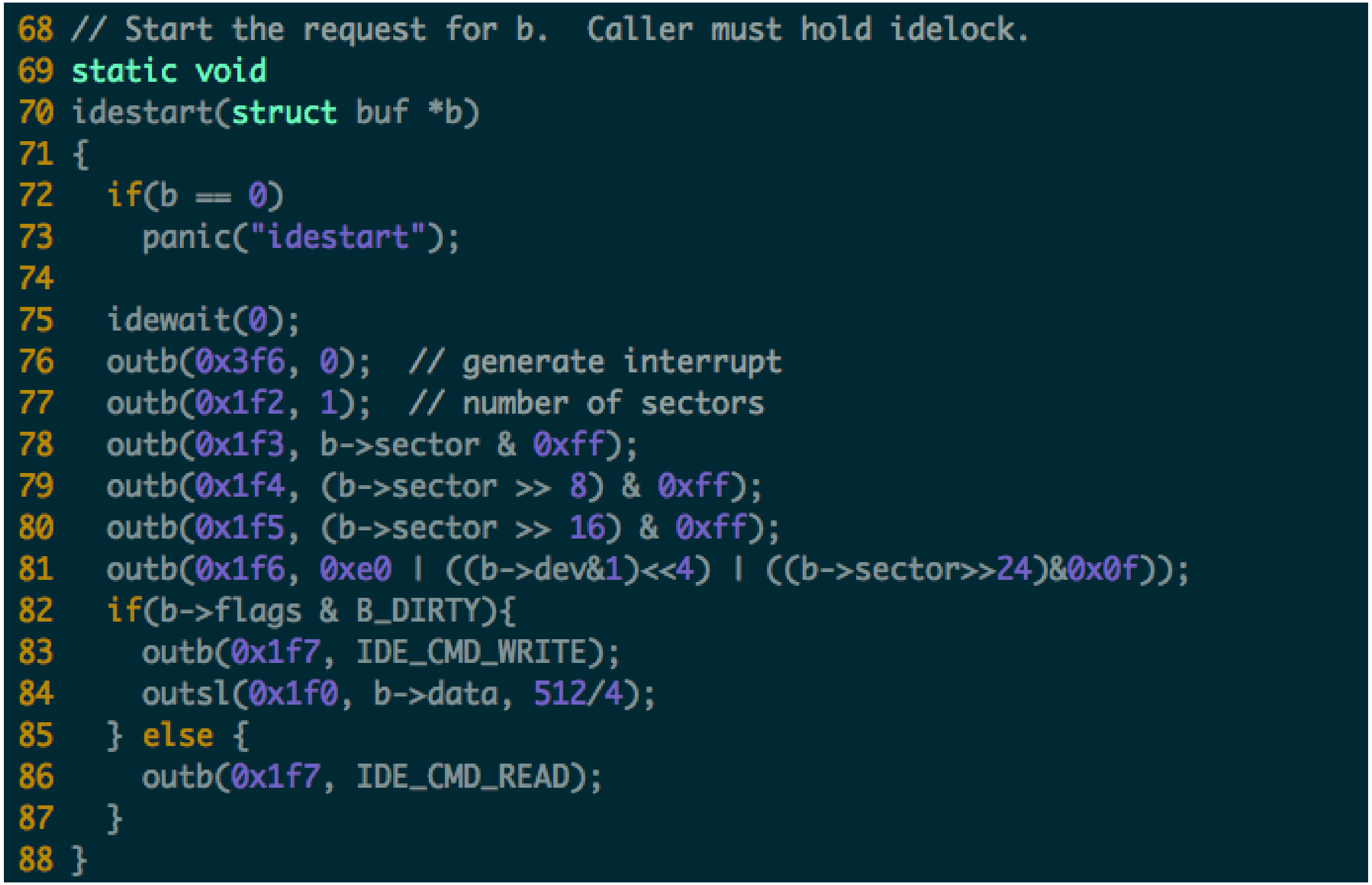


```
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void
wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

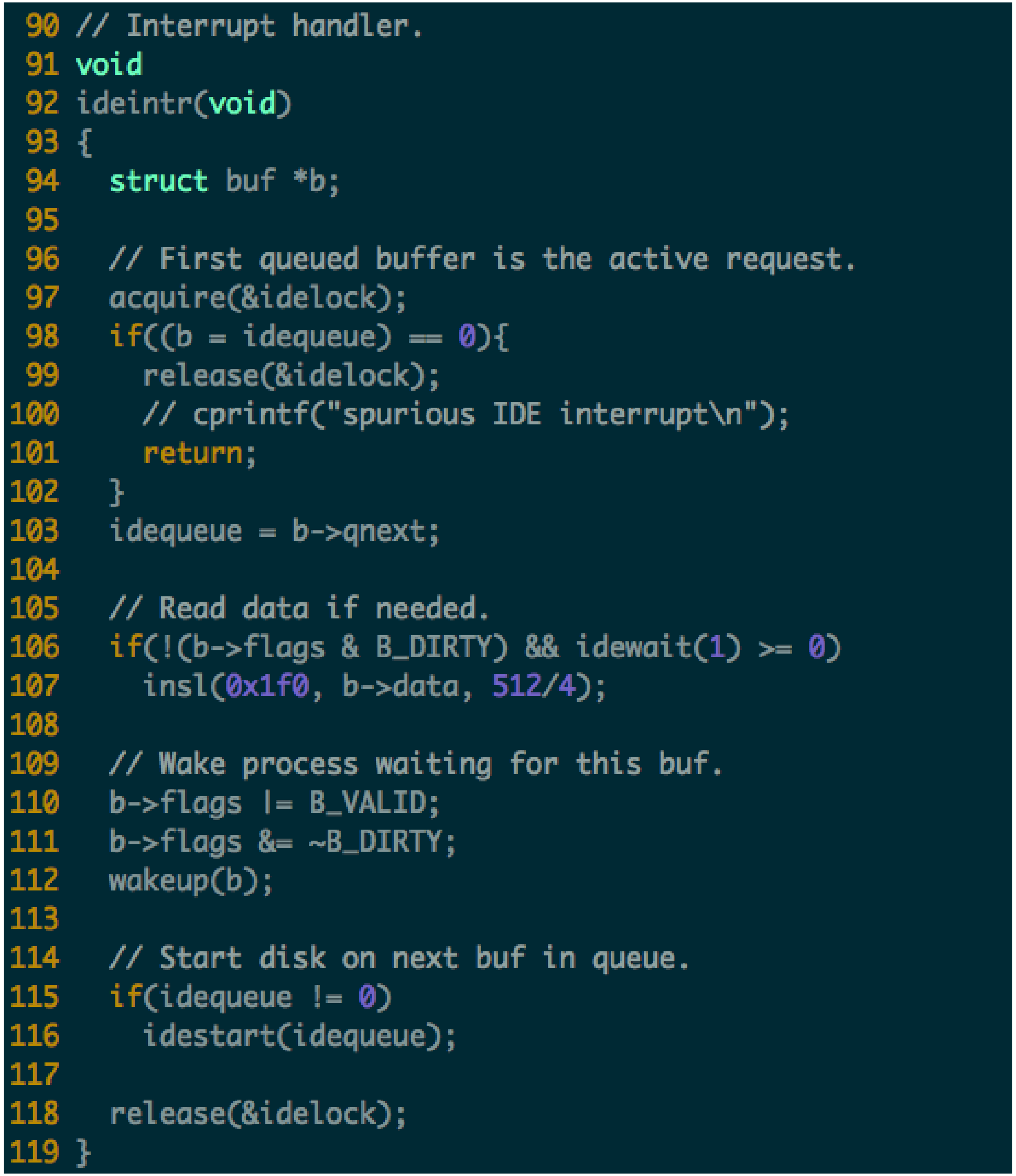
Synchronous I/O (iderw)



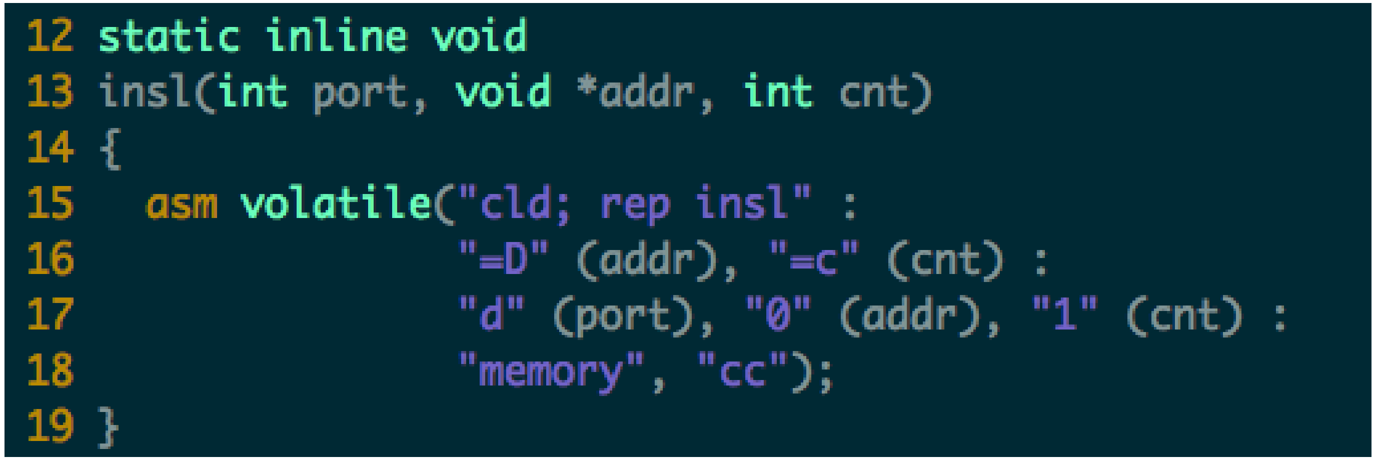
```c
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void
iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);  //DOC:acquire-lock

  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
}
```



```
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, 1);  // number of sectors
  outb(0x1f3, b->sector & 0xff);
  outb(0x1f4, (b->sector >> 8) & 0xff);
  outb(0x1f5, (b->sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, 512/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```



```
// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```



```c
static inline void
insl(int port, void *addr, int cnt)
{
  asm volatile("cld; rep insl" :
               "=D" (addr), "=c" (cnt) :
               "d" (port), "0" (addr), "1" (cnt) :
               "memory", "cc");
}
```

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
- iderw() maintains a queue of requests
  - Adds the buffer b to the end of the queue
  - If b is the first, then call idestart(), if not, just wait
  - Then just sleep, instead of polling
- ideintr() will handle the first request
  - Then start other request if any

XV6, FAT32, NTFS, Ext4

Yubin Xia & Rong Chen

Xv6 FS Design

- Implements a minimal Unix file system interface
  - Superblock
  - Inode
  - Dentry

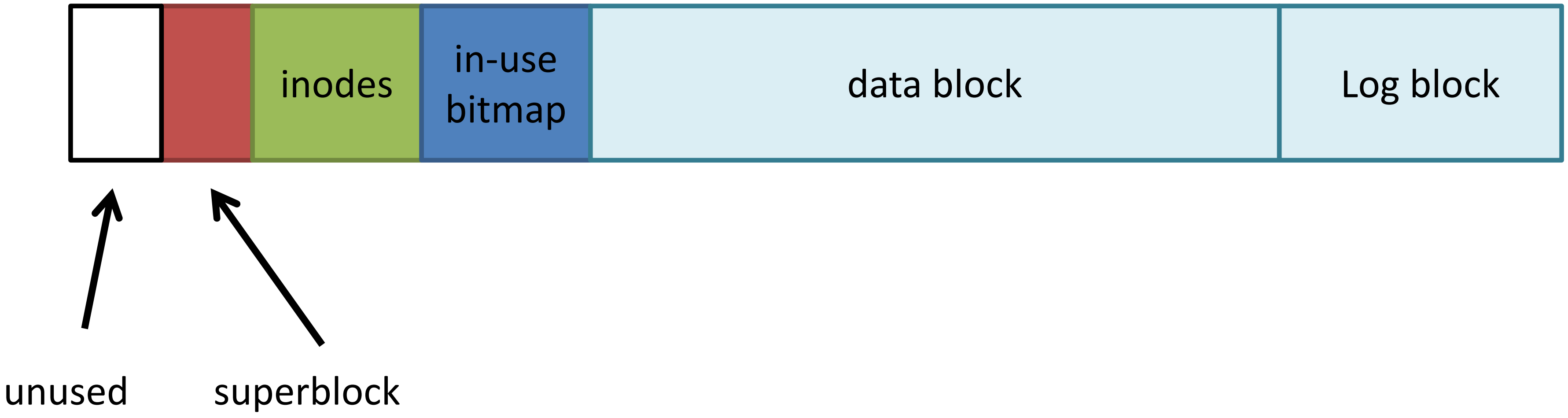
- Limitations
  - not pay attention to file system layout
  - doesn't do any disk scheduling.
  - Its cache is write-through
    - simplifies keeping disk data structures consistent
    - bad for performance.

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 (ninodes / inodes_per_block)
  - in-use bitmap
  - data block
  - Log block

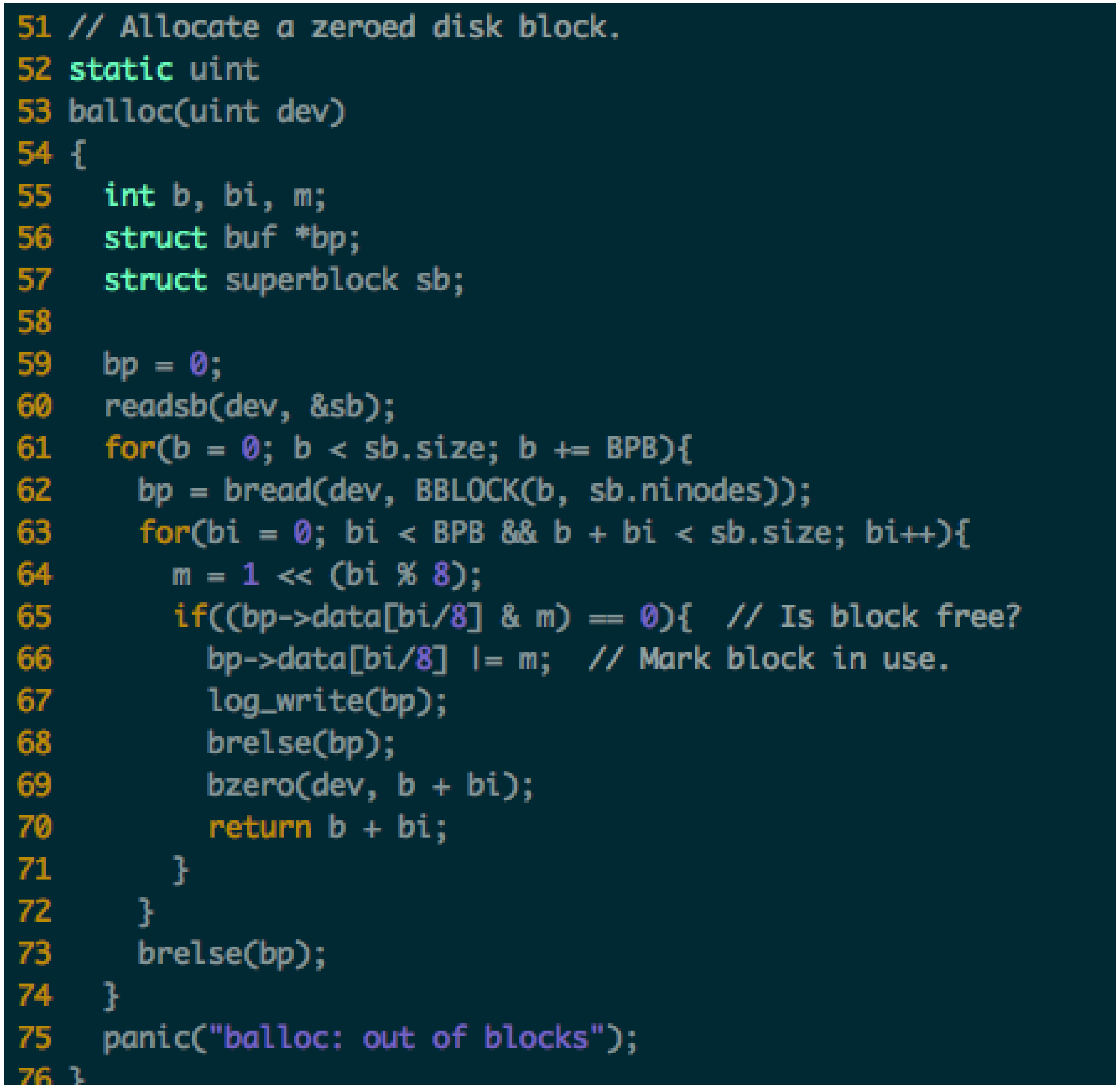


Recap: Free-Space List Bit Vector

- Represent the list of free blocks as a bit vector:
  - 11111111111111001110101011101111...
  - If bit i = 0 then block i is free, otherwise it is allocated
- Simple to use but this can be a big vector:
  - 160GB disk -> 40M blocks -> 5MB worth of bits
  - However, if free sectors are uniformly distributed across the disk
  - then the expected number of bits that must be scanned before finding a “0” is n/r, where
    - n = total number of blocks on the disk
    - r = number of free blocks
  - If a disk is 90% full, then the average number of bits to be scanned is 10, independent of the size of the disk

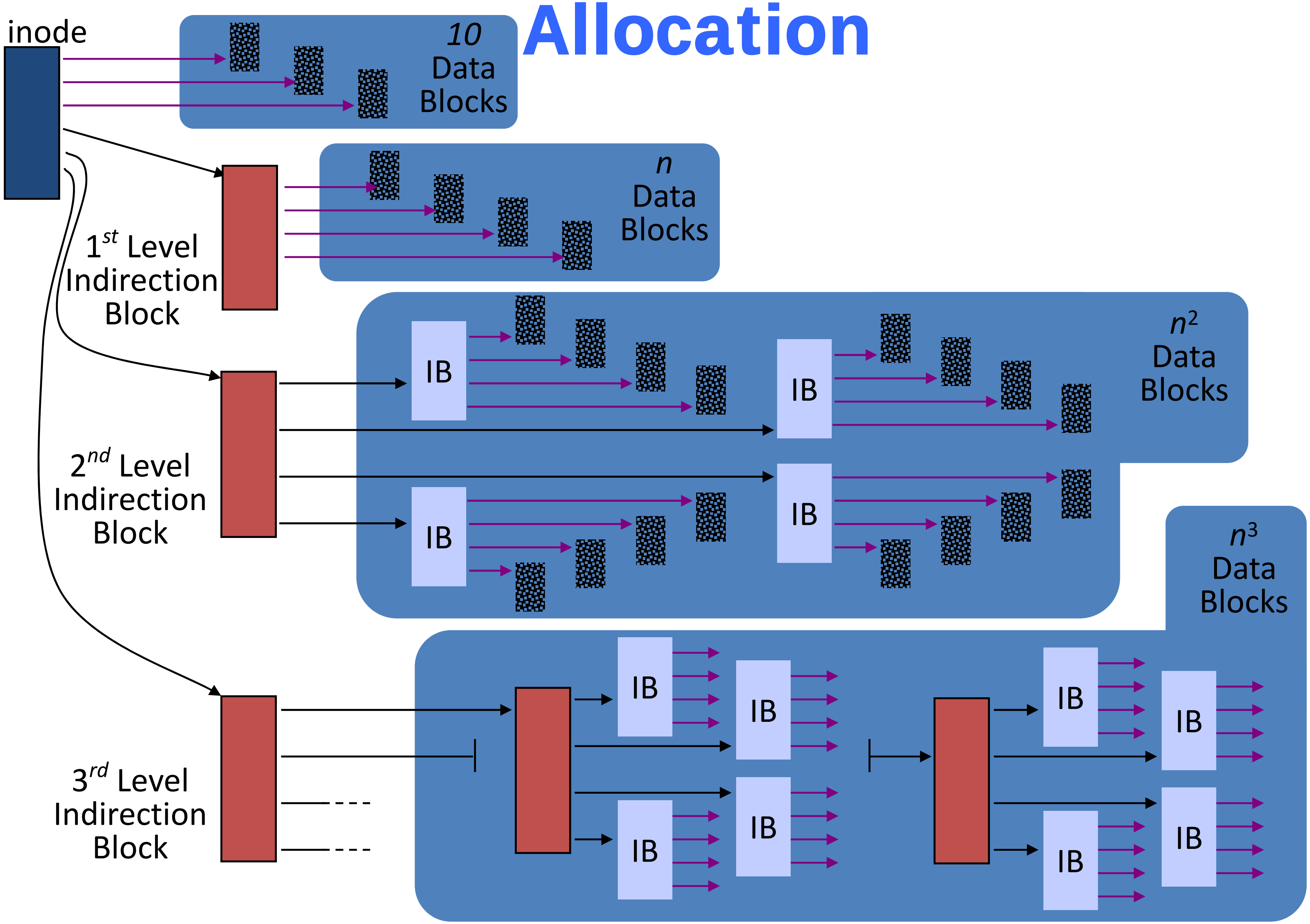
Managing Disk Blocks

- Use a bitmap to maintain availability of disk blocks



```c
// 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, BBLOCK(b, sb.ninodes));
    for(bi = 0; bi < BPB && b + bi < sb.size; bi++){
      m = 1 << (bi % 8);
      if((bp->data[bi/8] & m) == 0){  // Is block free?
        bp->data[bi/8] |= m;  // Mark block in use.
        log_write(bp);
        brelse(bp);
        bzero(dev, b + bi);
        return b + bi;
      }
    }
    brelse(bp);
  }
  panic("balloc: out of blocks");
}
```

Recap: Multi-level Indexed Allocation



Files

- 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.)

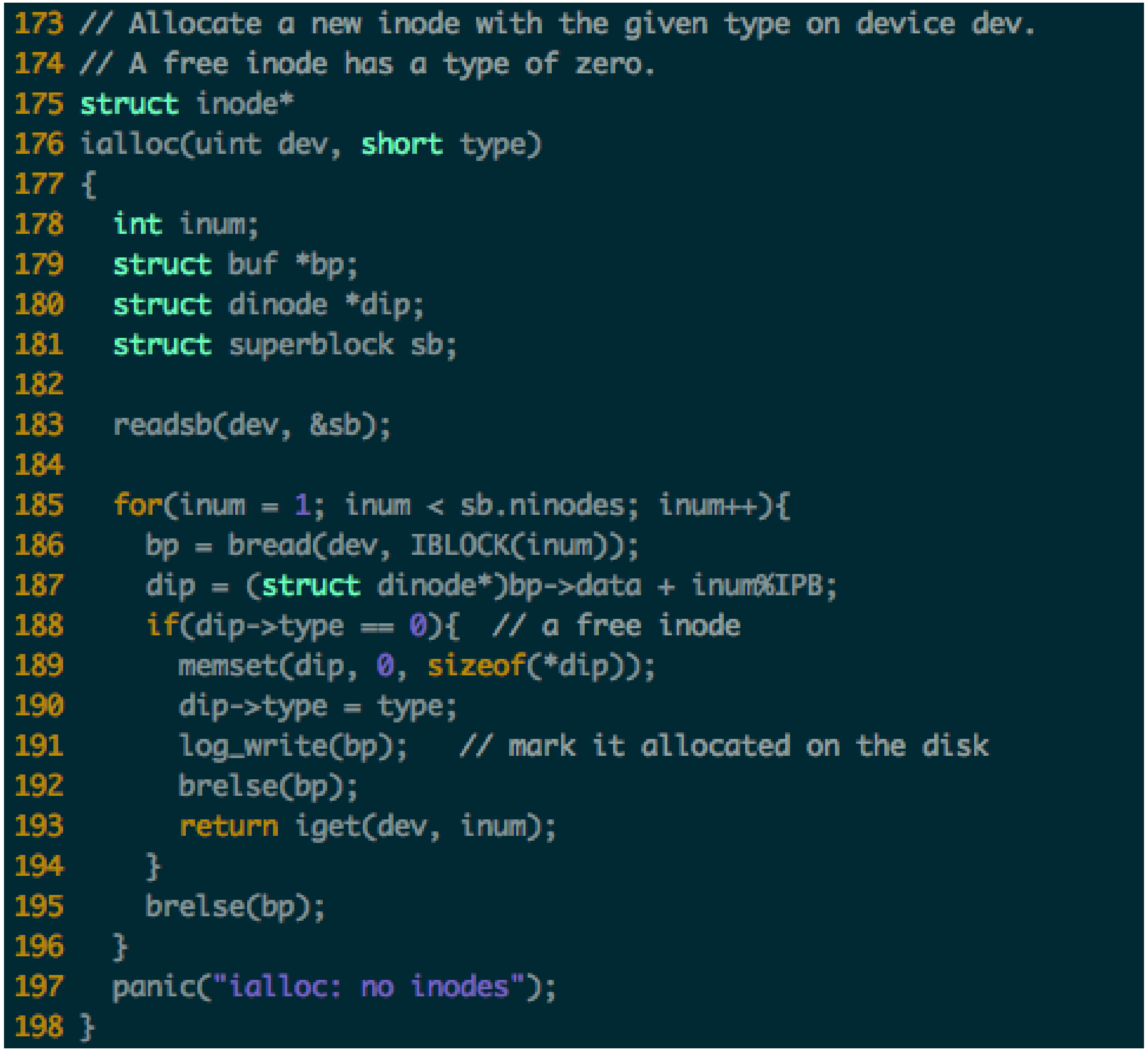
- File size
  - Small: up to 12 block addresses (*NADDRS – 1*)
  - Large: Last address in the inode as a disk address for a block with 128 disk addresses (512 / 4)
    - At most 12 * 512 + 128 * 512 bytes

Directories

- Directories are special files
  - Files with a bit of structure to them

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

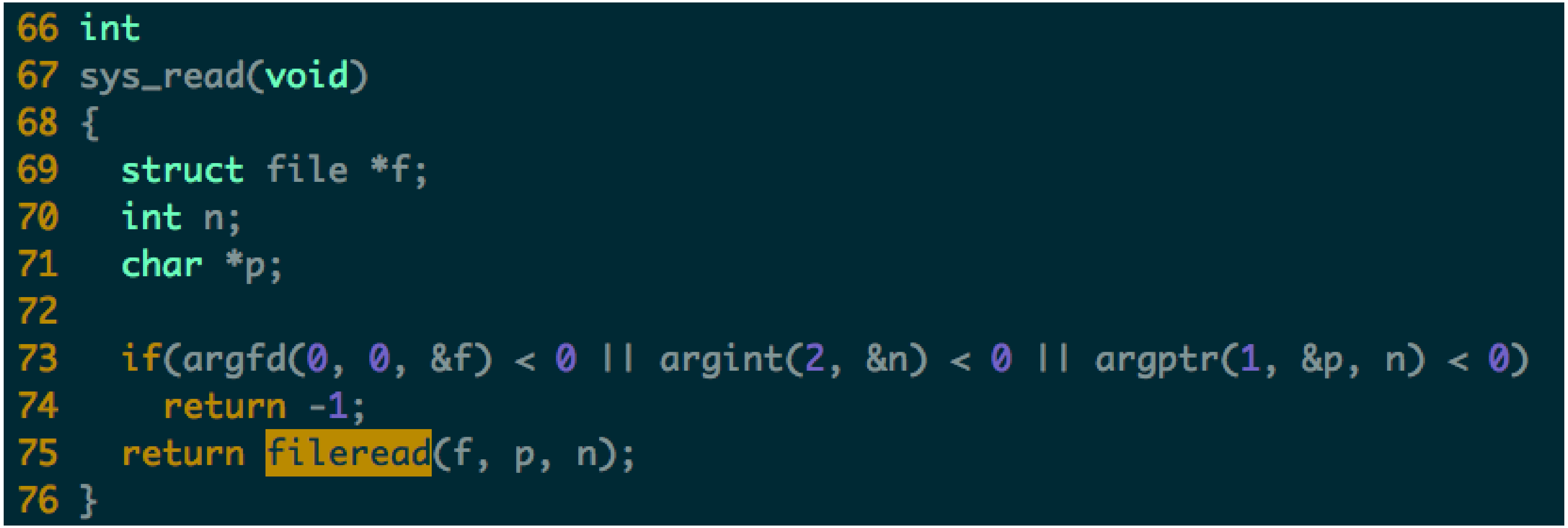
Allocating an inode



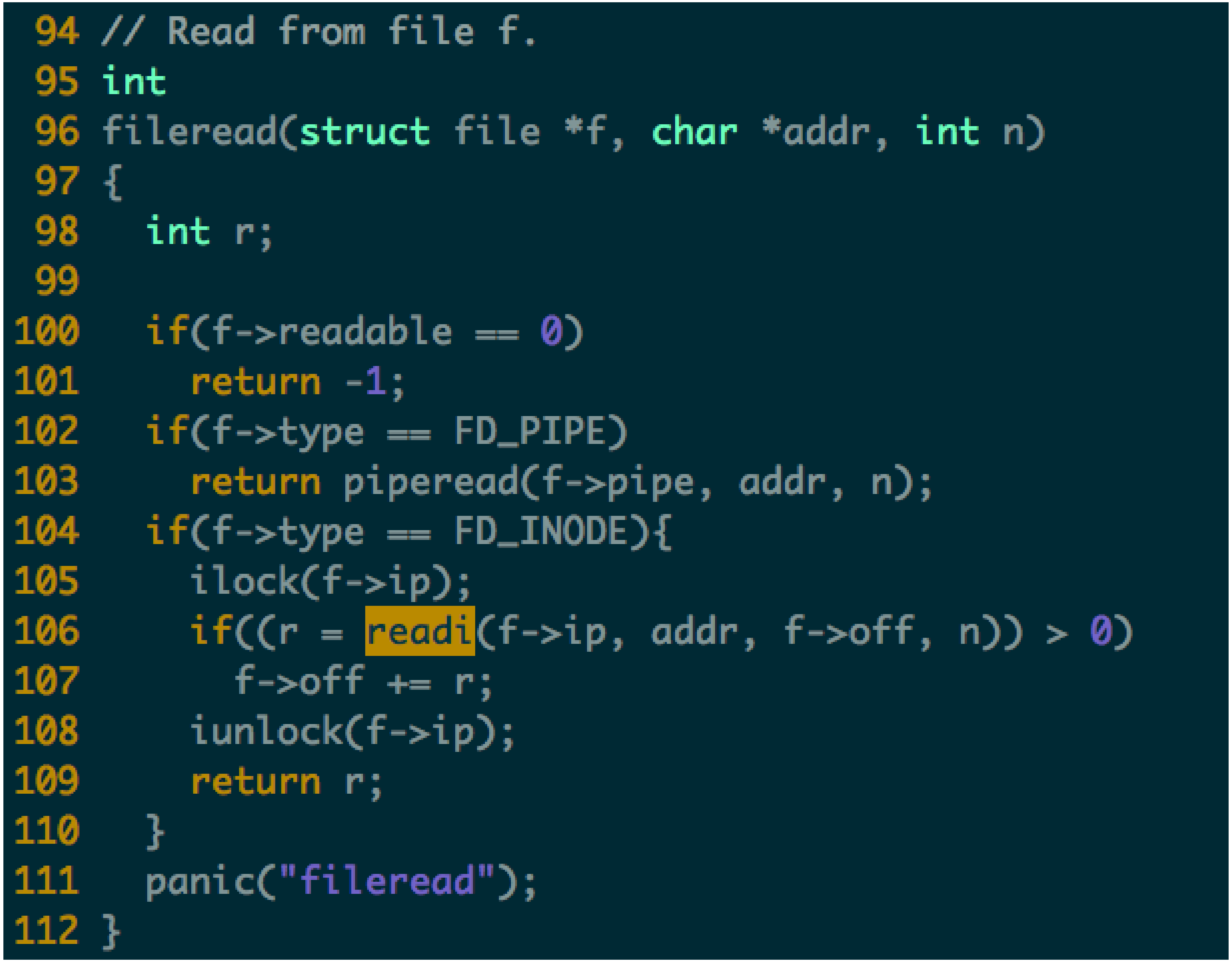
```
// Allocate a new inode with the given type on device dev.
// A free inode has a type of zero.
struct inode*
ialloc(uint dev, short type)
{
  int inum;
  struct buf *bp;
  struct dinode *dip;
  struct superblock sb;

  readsb(dev, &sb);

  for(inum = 1; inum < sb.ninodes; inum++){
    bp = bread(dev, IBLOCK(inum));
    dip = (struct dinode*)bp->data + inum%IPB;
    if(dip->type == 0){  // a free inode
      memset(dip, 0, sizeof(*dip));
      dip->type = type;
      log_write(bp);   // mark it allocated on the disk
      brelse(bp);
      return iget(dev, inum);
    }
    brelse(bp);
  }
  panic("ialloc: no inodes");
}
```

Case Study: read(fd, buf, 100)

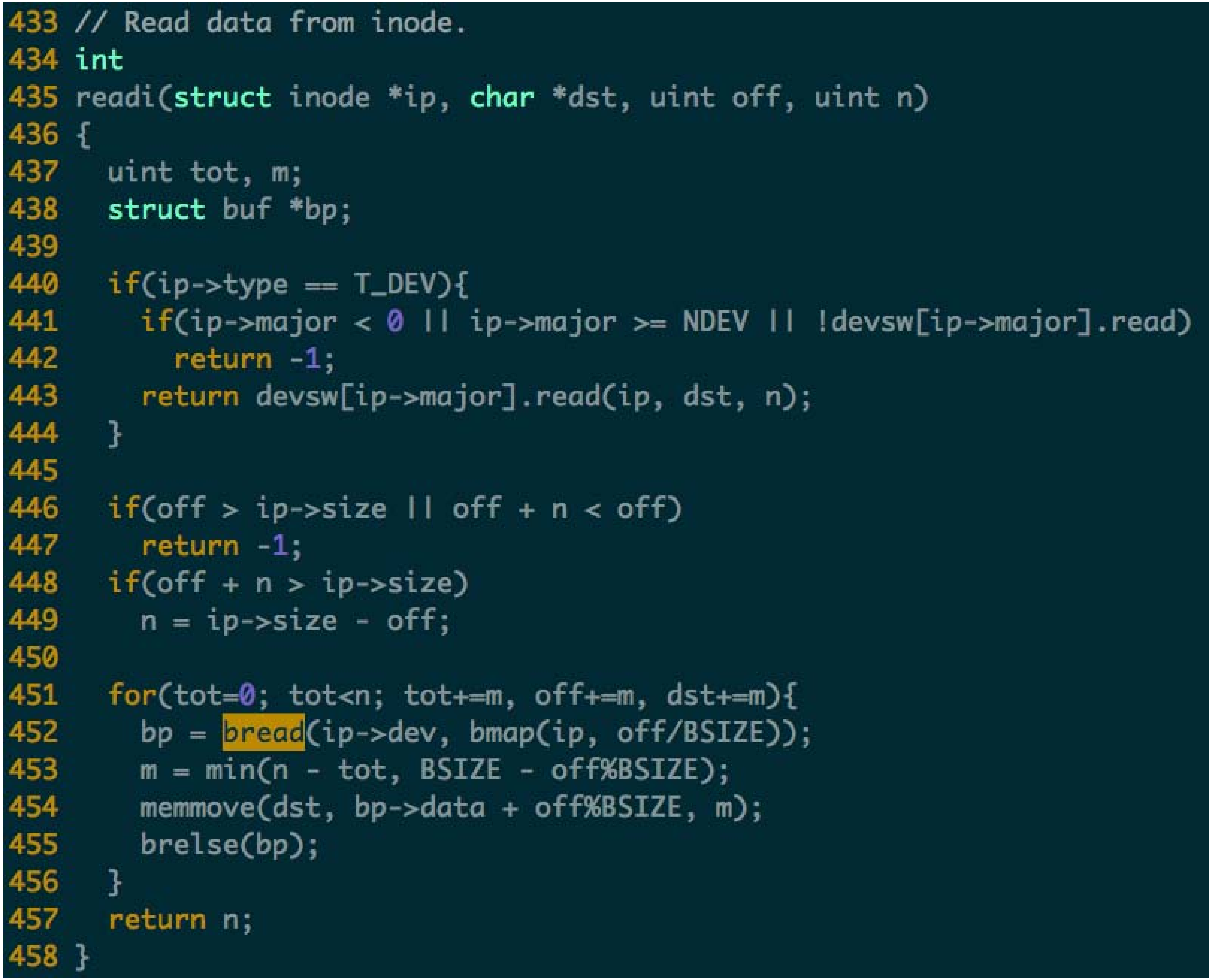
- sys_read()->fileread()->readi()
  - 1. If it’s a special device file, use predefined operation
  - 2. call bread() to get file buffer



```c
int
sys_read(void)
{
  struct file *f;
  int n;
  char *p;

  if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) < 0)
    return -1;
  return fileread(f, p, n);
}
```



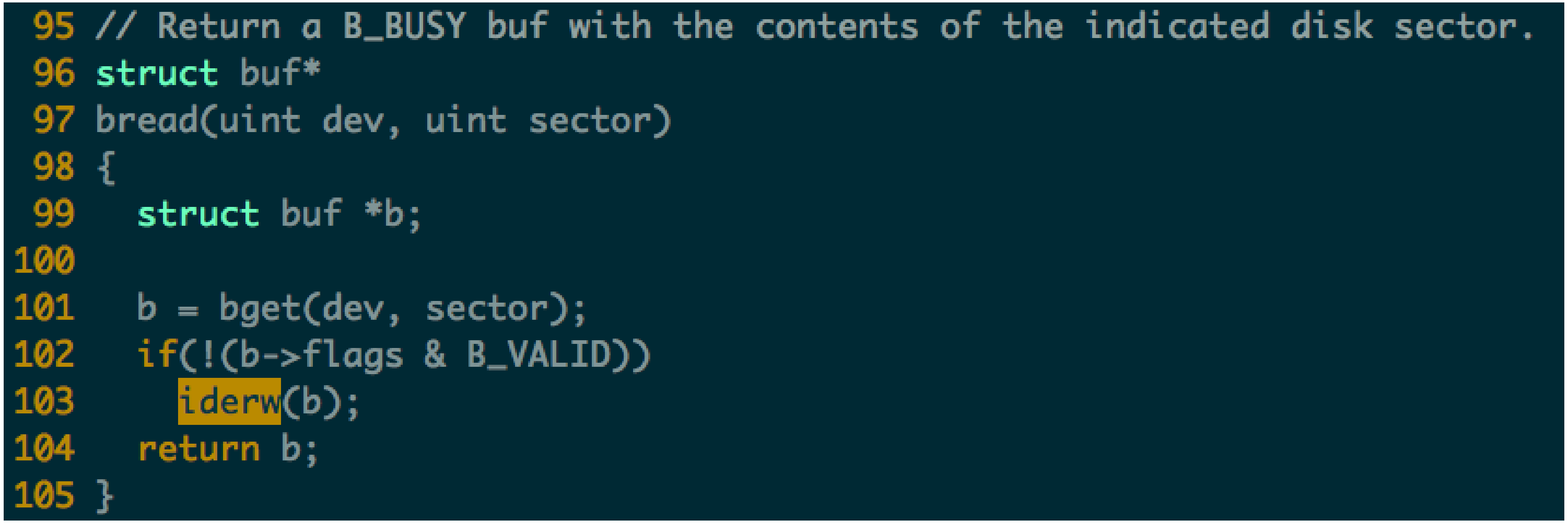
```
// Read from file f.
int
fileread(struct file *f, char *addr, int n)
{
  int r;

  if(f->readable == 0)
    return -1;
  if(f->type == FD_PIPE)
    return piperead(f->pipe, addr, n);
  if(f->type == FD_INODE){
    ilock(f->ip);
    if((r = readi(f->ip, addr, f->off, n)) > 0)
      f->off += r;
    iunlock(f->ip);
    return r;
  }
  panic("fileread");
}
```



```c
// Read data from inode.
int
readi(struct inode *ip, char *dst, uint off, uint n)
{
  uint tot, m;
  struct buf *bp;

  if(ip->type == T_DEV){
    if(ip->major < 0 || ip->major >= NDEV || !devsw[ip->major].read)
      return -1;
    return devsw[ip->major].read(ip, dst, n);
  }

  if(off > ip->size || off + n < off)
    return -1;
  if(off + n > ip->size)
    n = ip->size - off;

  for(tot=0; tot<n; tot+=m, off+=m, dst+=m){
    bp = bread(ip->dev, bmap(ip, off/BSIZE));
    m = min(n - tot, BSIZE - off%BSIZE);
    memmove(dst, bp->data + off%BSIZE, m);
    brelse(bp);
  }
  return n;
}
```

Case Study: read (cont.)

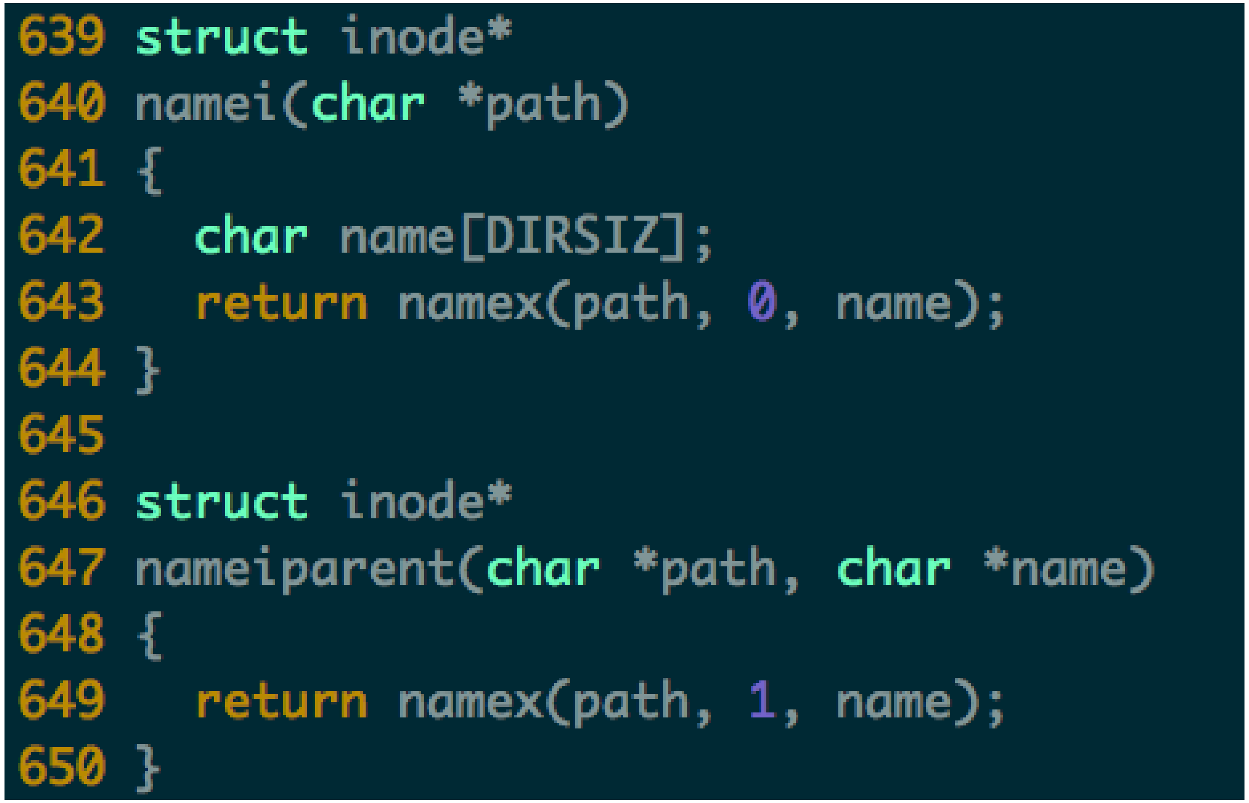
- readi()->bread()
  - **Call bget() to read from buffer cache first**
  - If not found, call ide_rw() to read from disk, and mark as valid.



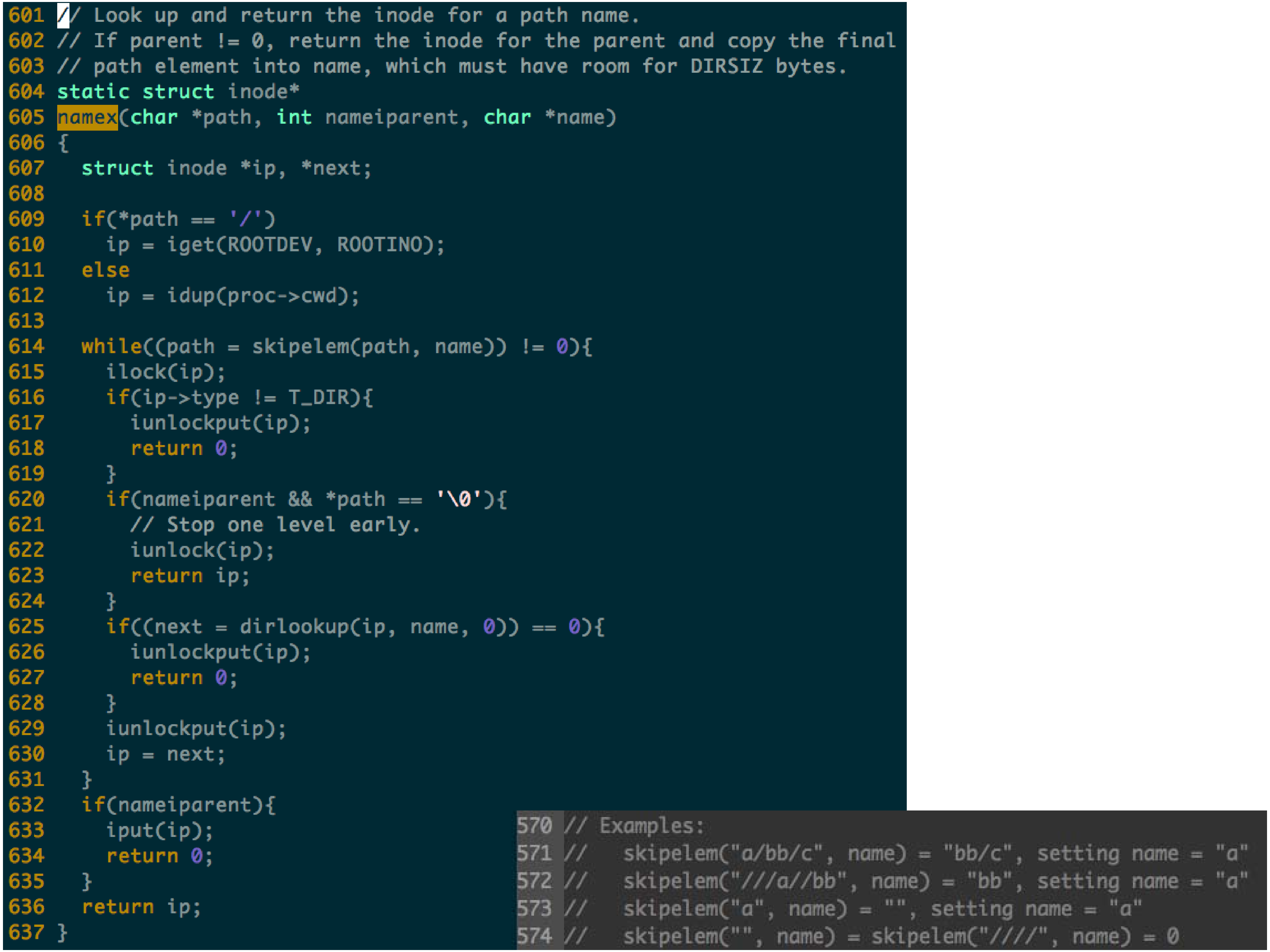
```
// Return a B_BUSY buf with the contents of the indicated disk sector.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b;

  b = bget(dev, sector);
  if(!(b->flags & B_VALID))
    iderw(b);
  return b;
}
```

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

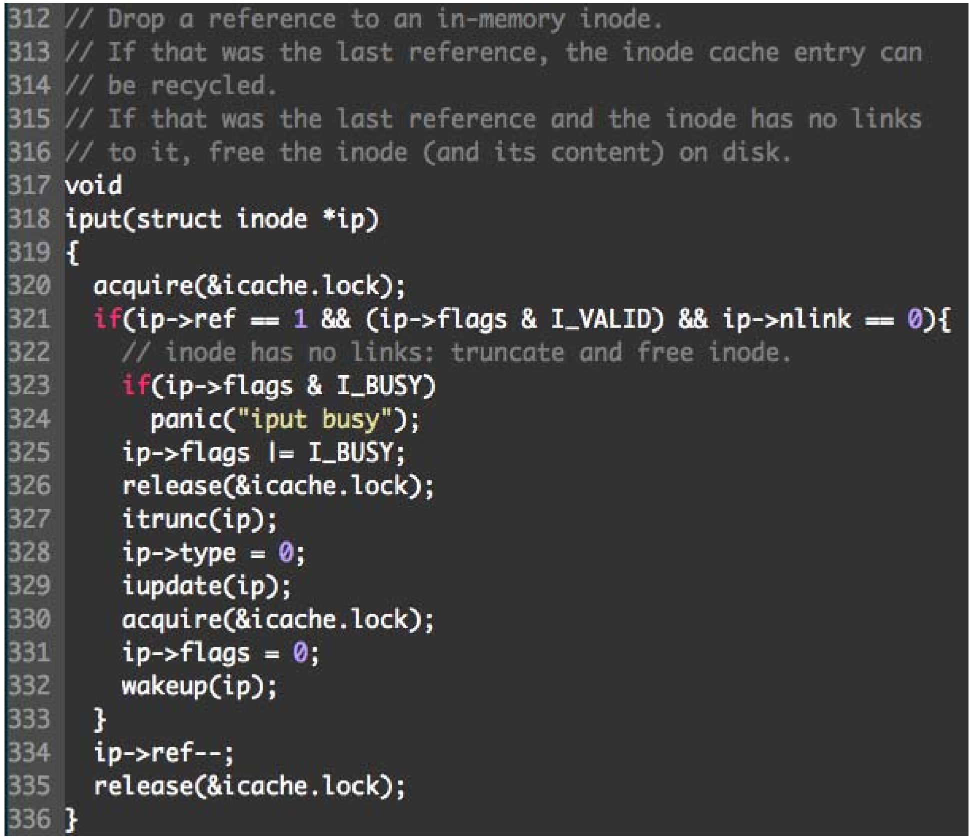


```c
struct inode*
namei(char *path)
{
  char name[DIRSIZ];
  return namex(path, 0, name);
}

struct inode*
nameiparent(char *path, char *name)
{
  return namex(path, 1, name);
}
```

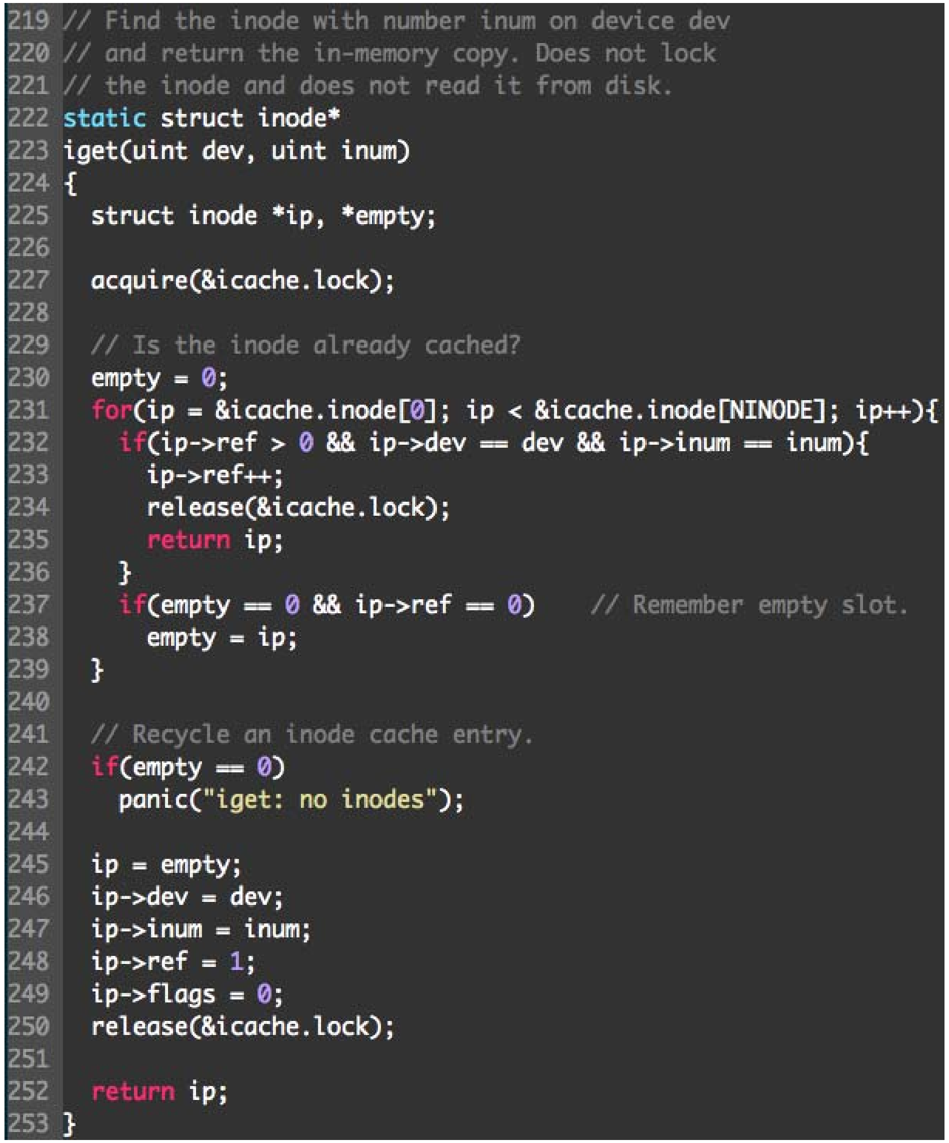


```c
// Look up and return the inode for a path name.
// If parent != 0, return the inode for the parent and copy the final
// path element into name, which must have room for DIRSIZ bytes.
static struct inode*
namex(char *path, int nameiparent, char *name)
{
  struct inode *ip, *next;

  if(*path == '/')
    ip = iget(ROOTDEV, ROOTINO);
  else
    ip = idup(proc->cwd);

  while((path = skipelem(path, name)) != 0){
    ilock(ip);
    if(ip->type != T_DIR){
      iunlockput(ip);
      return 0;
    }
    if(nameiparent && *path == '\0'){
      // Stop one level early.
      iunlock(ip);
      return ip;
    }
    if((next = dirlookup(ip, name, 0)) == 0){
      iunlockput(ip);
      return 0;
    }
    iunlockput(ip);
    ip = next;
  }
  if(nameiparent){
    iput(ip);
    return 0;
  }
  return ip;
}
```

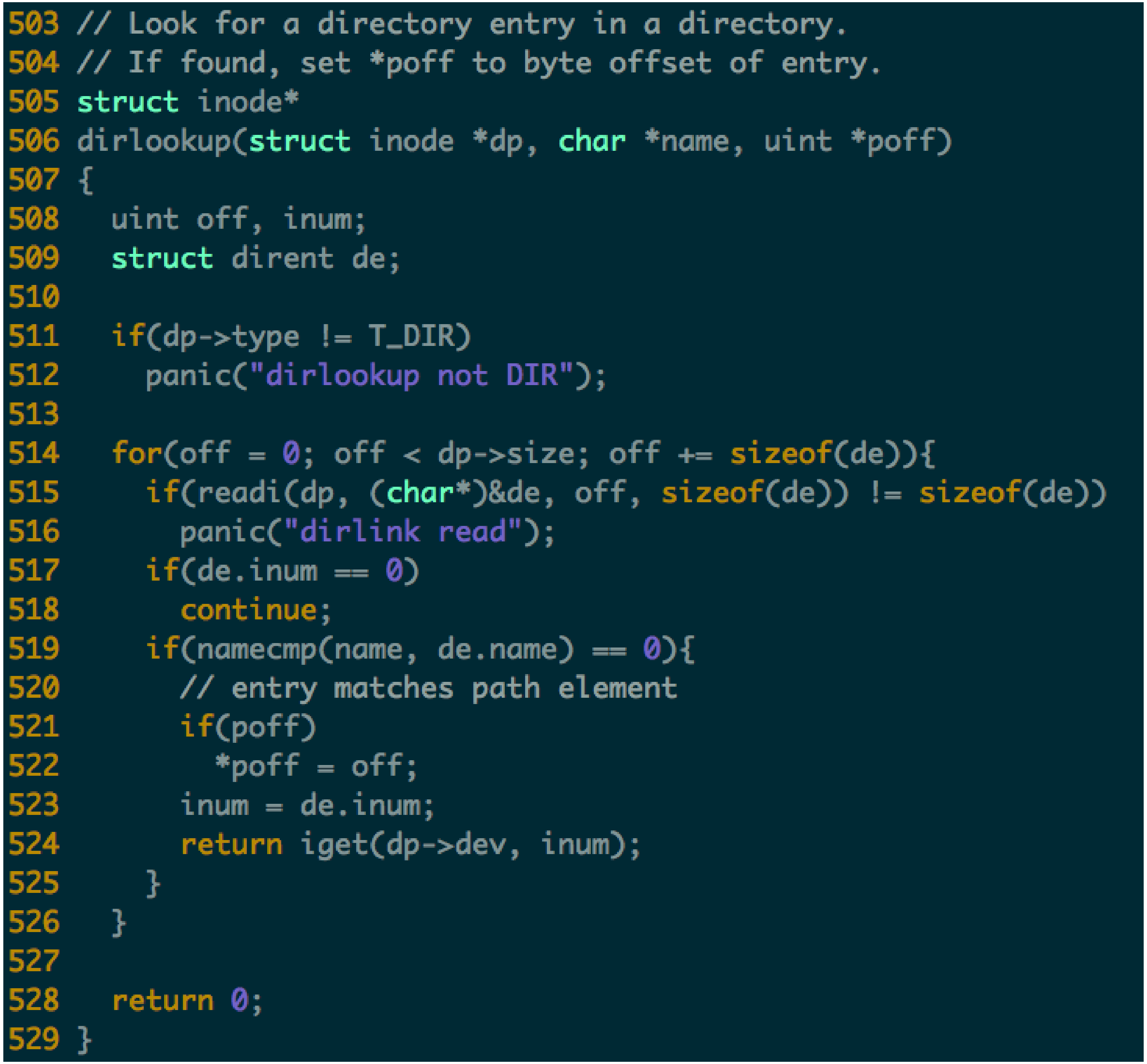
```c
// Examples:
//   skipelem("a/bb/c", name) = "bb/c", setting name = "a"
//   skipelem("///a//bb", name) = "bb", setting name = "a"
//   skipelem("a", name) = "", setting name = "a"
//   skipelem("", name) = skipelem("////", name) = 0
```



```c
// Drop a reference to an in-memory inode.
// If that was the last reference, the inode cache entry can
// be recycled.
// If that was the last reference and the inode has no links
// to it, free the inode (and its content) on disk.
void
iput(struct inode *ip)
{
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode has no links: truncate and free inode.
    if(ip->flags & I_BUSY)
      panic("iput busy");
    ip->flags |= I_BUSY;
    release(&icache.lock);
    itrunc(ip);
    ip->type = 0;
    iupdate(ip);
    acquire(&icache.lock);
    ip->flags = 0;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}
```



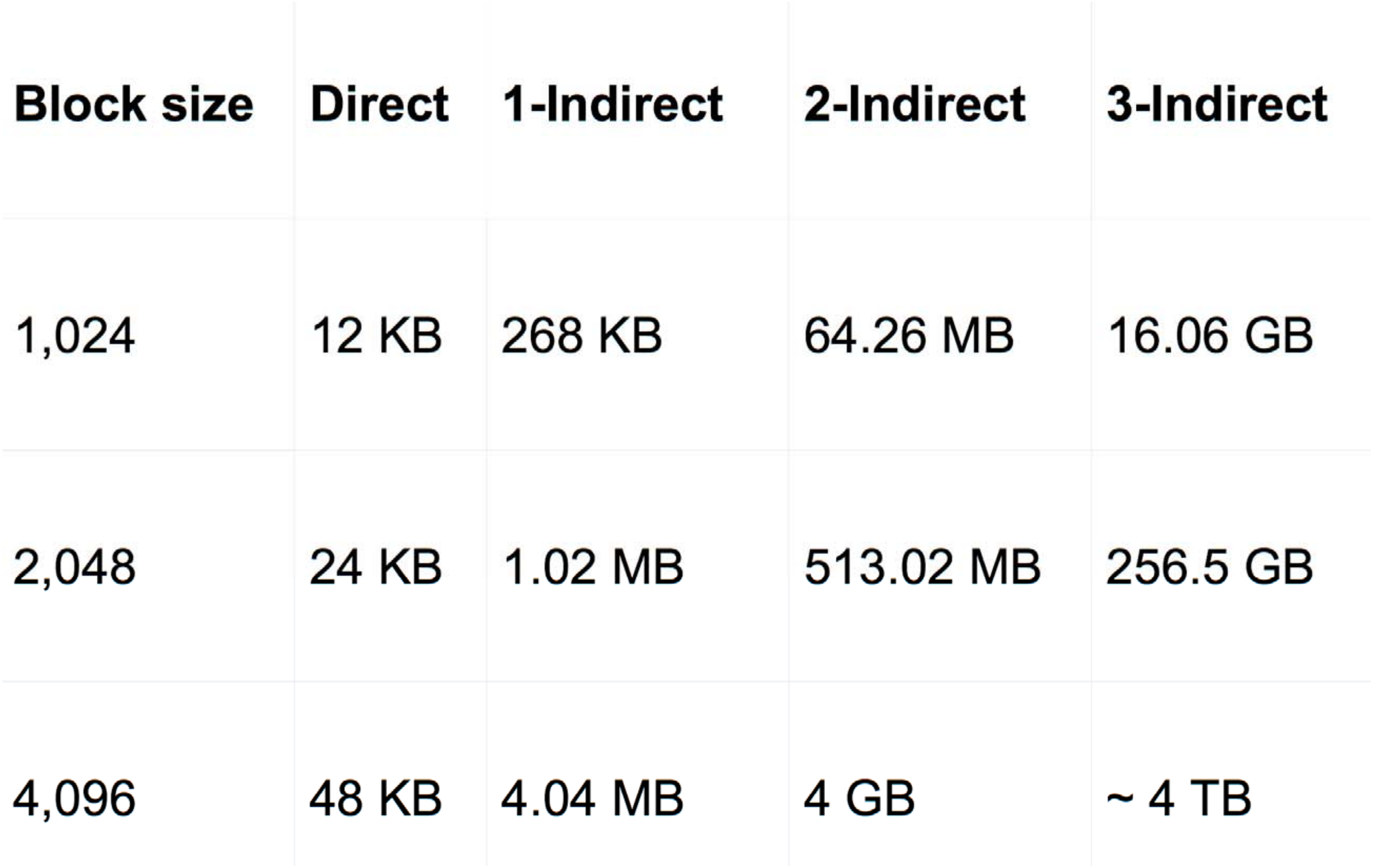
```c
// Find the inode with number inum on device dev
// and return the in-memory copy. Does not lock
// the inode and does not read it from disk.
static struct inode*
iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)    // Remember empty slot.
      empty = ip;
  }

  // Recycle an inode cache entry.
  if(empty == 0)
    panic("iget: no inodes");

  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->flags = 0;
  release(&icache.lock);

  return ip;
}
```



```c
// Look for a directory entry in a directory.
// If found, set *poff to byte offset of entry.
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```

EXT4 FILE SYSTEM

File-size upper limits for data block addressing



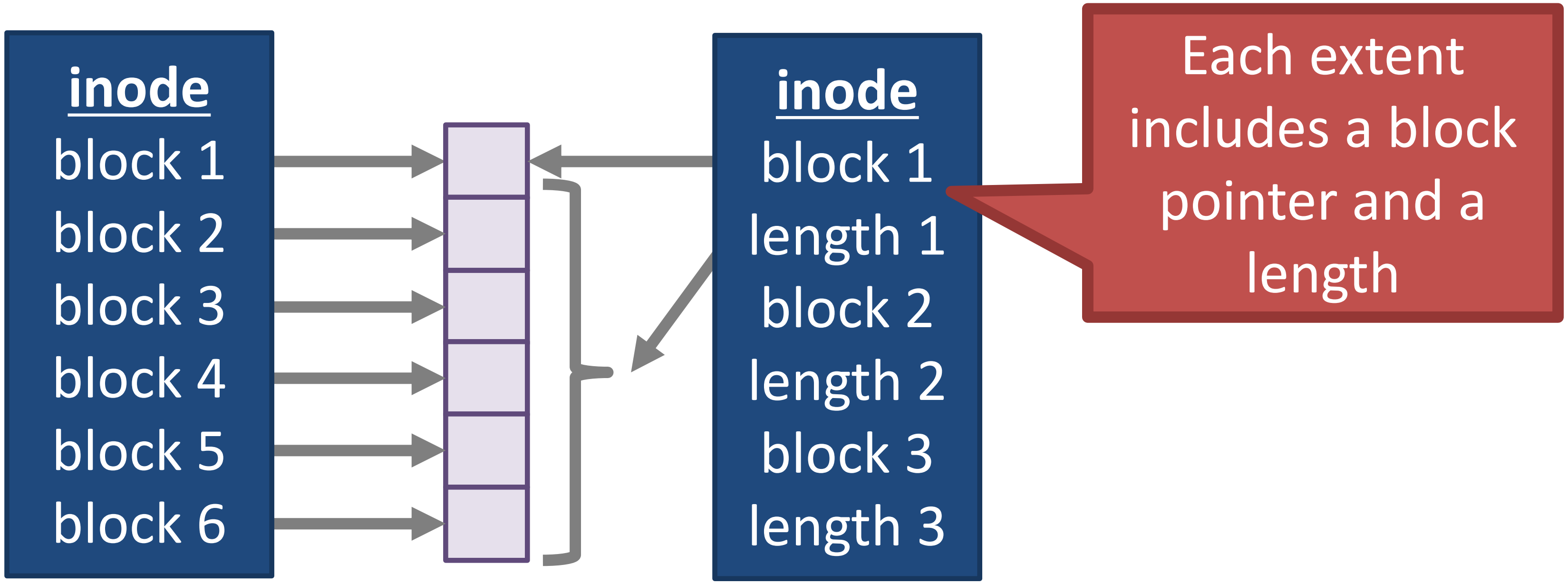
Block size	Direct	1-Indirect	2-Indirect	3-Indirect
1,024	12 KB	268 KB	64.26 MB	16.06 GB
2,048	24 KB	1.02 MB	513.02 MB	256.5 GB
4,096	48 KB	4.04 MB	4 GB	~ 4 TB

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
  - a 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



Each extent includes a block pointer and a length

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

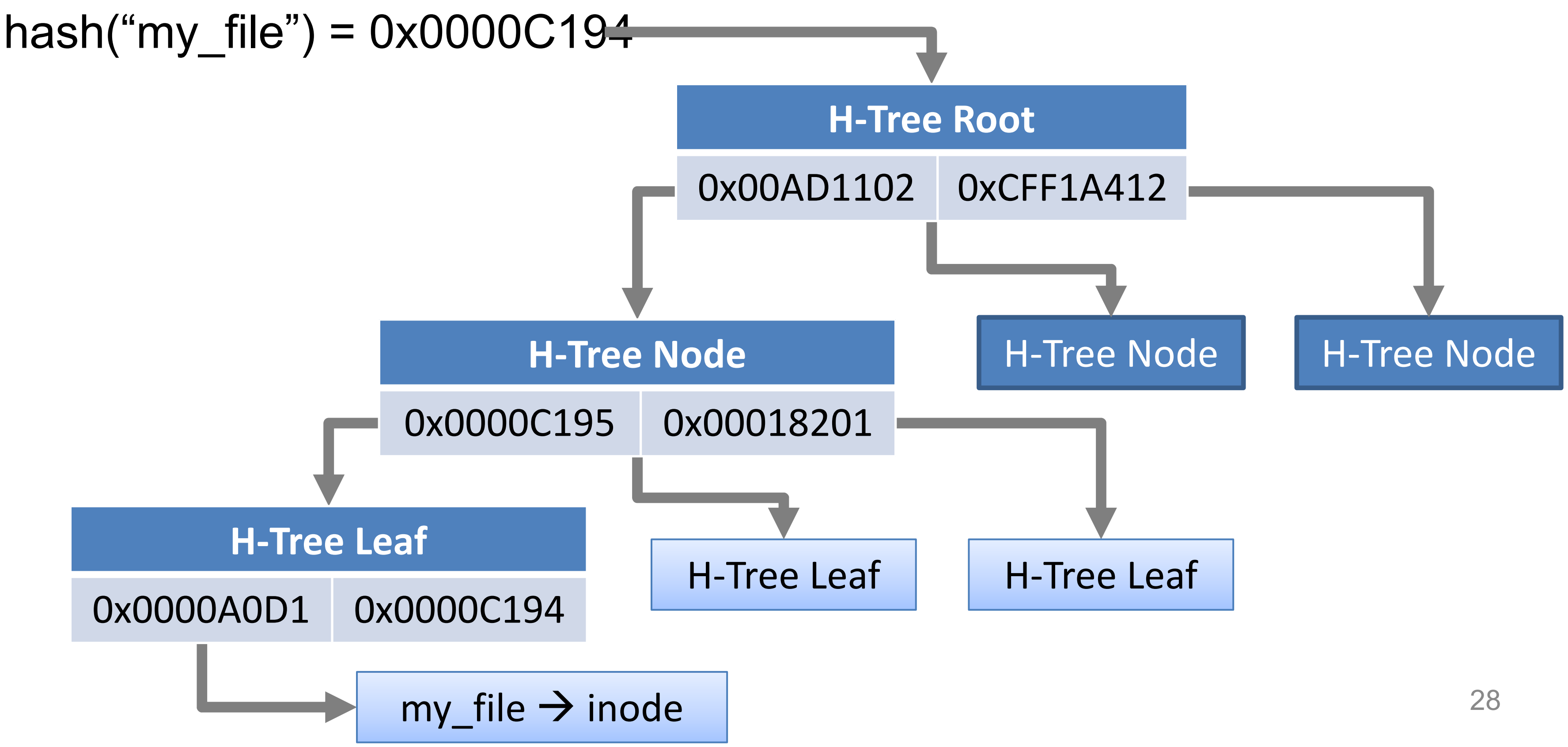
- In ext, ext2, and ext3, each directory is a file with a list of entries
  - Entries are not stored in sorted order
  - Some entries may be blank, if they have been deleted
- Problem: searching for files in large directories takes O(n) time
  - Practically, you can't store >10K files in a directory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

FAT FILE SYSTEM

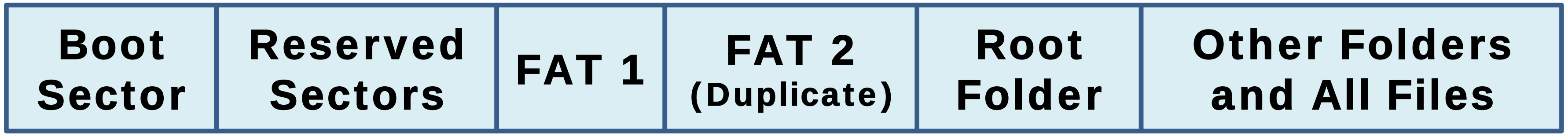
Clusters and Sectors

- 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

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

FAT 1

- Original FAT

FAT 2 (Duplicate)

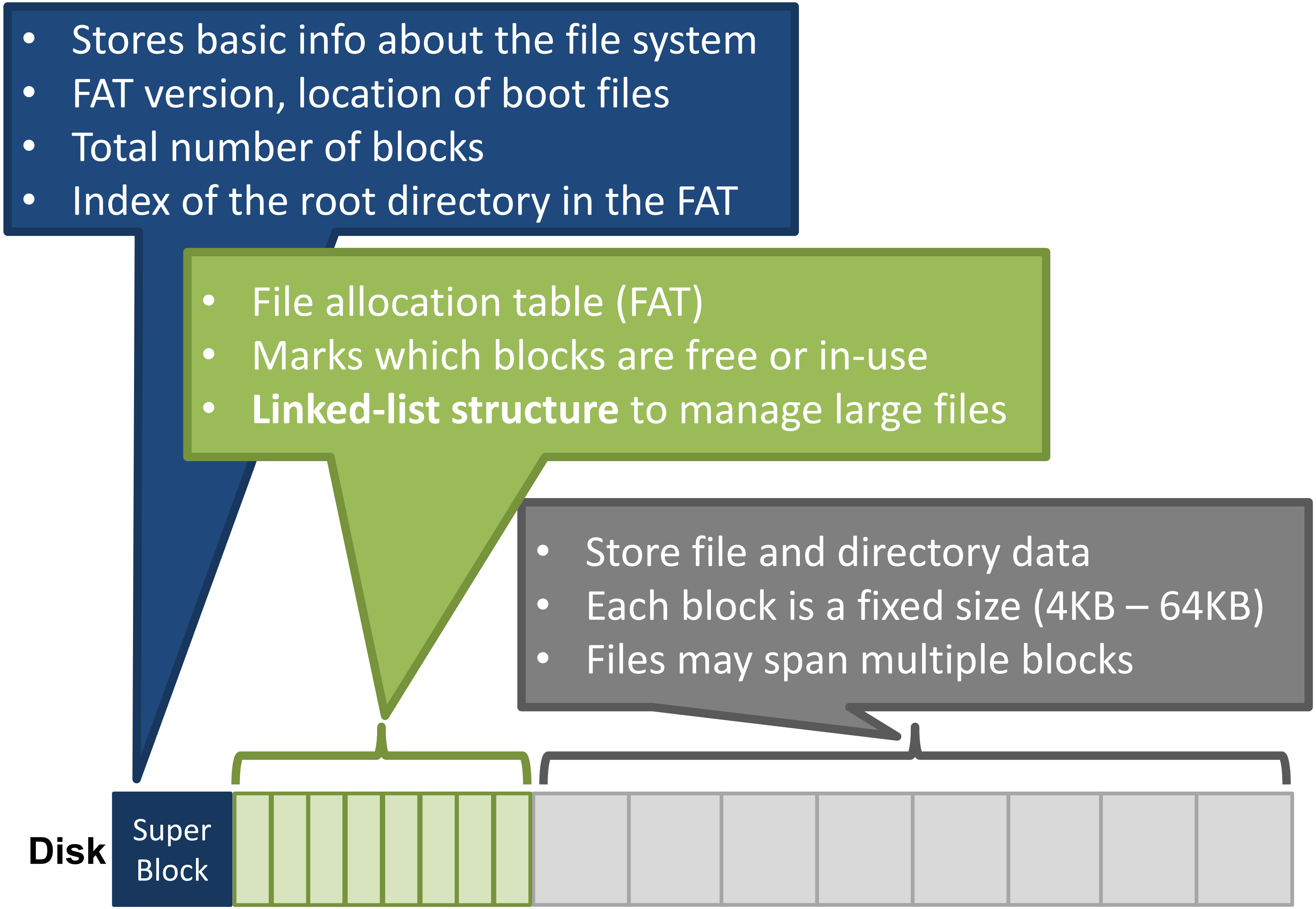
- Backup copy of FAT

Root Folder

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

Other Folder and All Files

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



- 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

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

Component of Boot Sector

0x00	0x03	0x0B	0x40	0x5A	0x01FE
Jump	OEM ID	BPB	Ex-BPB	Bootstrap Code	Marker
3Bs	8Bs	53Bs	26Bs	420Bs	2Bs

- Jump: skip the next several non-executable bytes
- OEM ID: a string of characters
  - the name and version number of OS that formatted the volumes
- BPB & Ex-BPB: information of volume
  - Byte Per Sector, Sectors Per Cluster, Number of FAT, Root Entries, Media Descriptor, Sectors Per Fat, … …
- Bootstrap Code: executable code to start the OS
- Marker: end of boot sector

FAT 1 and FAT 2

- File Allocation Table
  - identifies all clusters as
    - unused, cluster in use by a file, bad cluster, last cluster in a file
  - FAT 2 is used to consistency-checking program

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 create
Last Access Date	2B	Date file was last accessed
Last Modification Time and Date	4B	Time and data file was last modified
First Cluster	2B	Starting cluster number in the file allocation table
File Size	4B	Size of the file

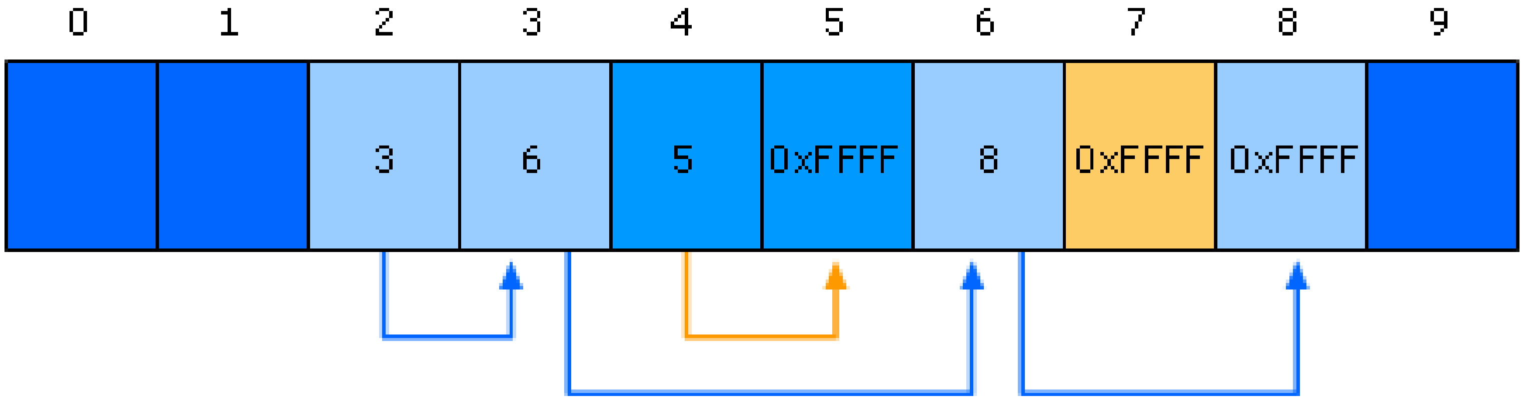
- FAT File System
  - FAT Architecture
  - FAT Physical Structure
  - FAT Processes and Interactions

Startup Process

- **BIOS** and CPU initiate the power-on self test
- BIOS find the boot device
- BIOS loads the first physical sector (**MBR**)
- Transfer CPU execution to MBR
- MBR scans the partition table
- Load a copy of the **Boot Sector** for active partition
- Transfers CPU execution to Boot Sector

File Processing on FAT Clusters

- Store file information
  - Continues to store file info in the next available cluster
    - when the file requires space greater than the cluster’s size.
  - e.g.
    - Three Files: 1 with (2-3-6-8), 2 with(4-5)   3 with (7)

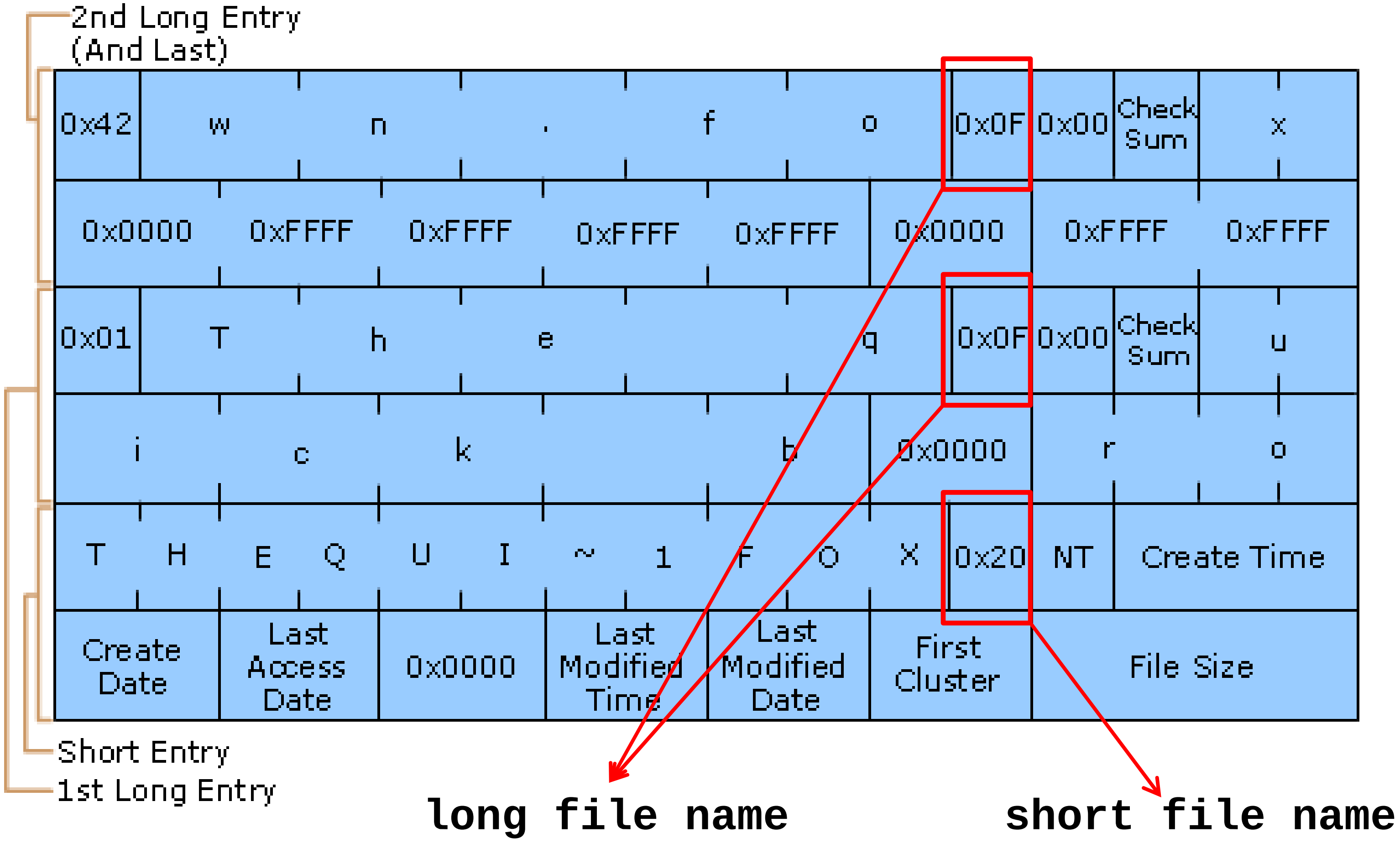


File Naming

- 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”



FAT: The Good and the Bad

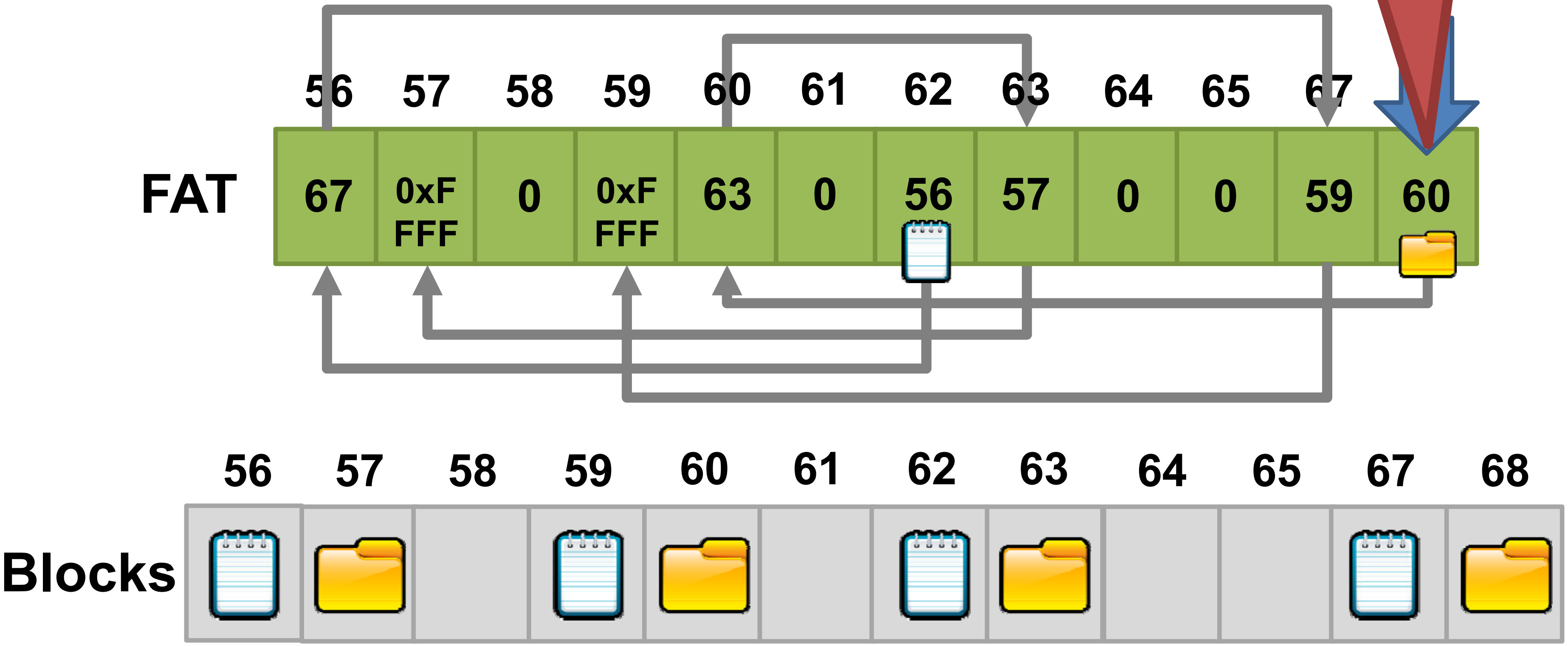
- The Good – FAT supports:
  - Hierarchical tree of directories and files
  - Variable length files
  - Basic file and directory meta-data
- The Bad
  - At most, FAT32 supports 2TB disks
  - Locating free chunks requires scanning the entire FAT
  - Prone to internal and external fragmentation
    - Large 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); // 4 4KB blocks
```

FAT may have very low spatial locality, thus a lot of random seeking



	56	57	58	59	60	61	62	63	64	65	67	
FAT	67	0xFFFF	0	0xFFFF	63	0	56	57	0	0	59	60

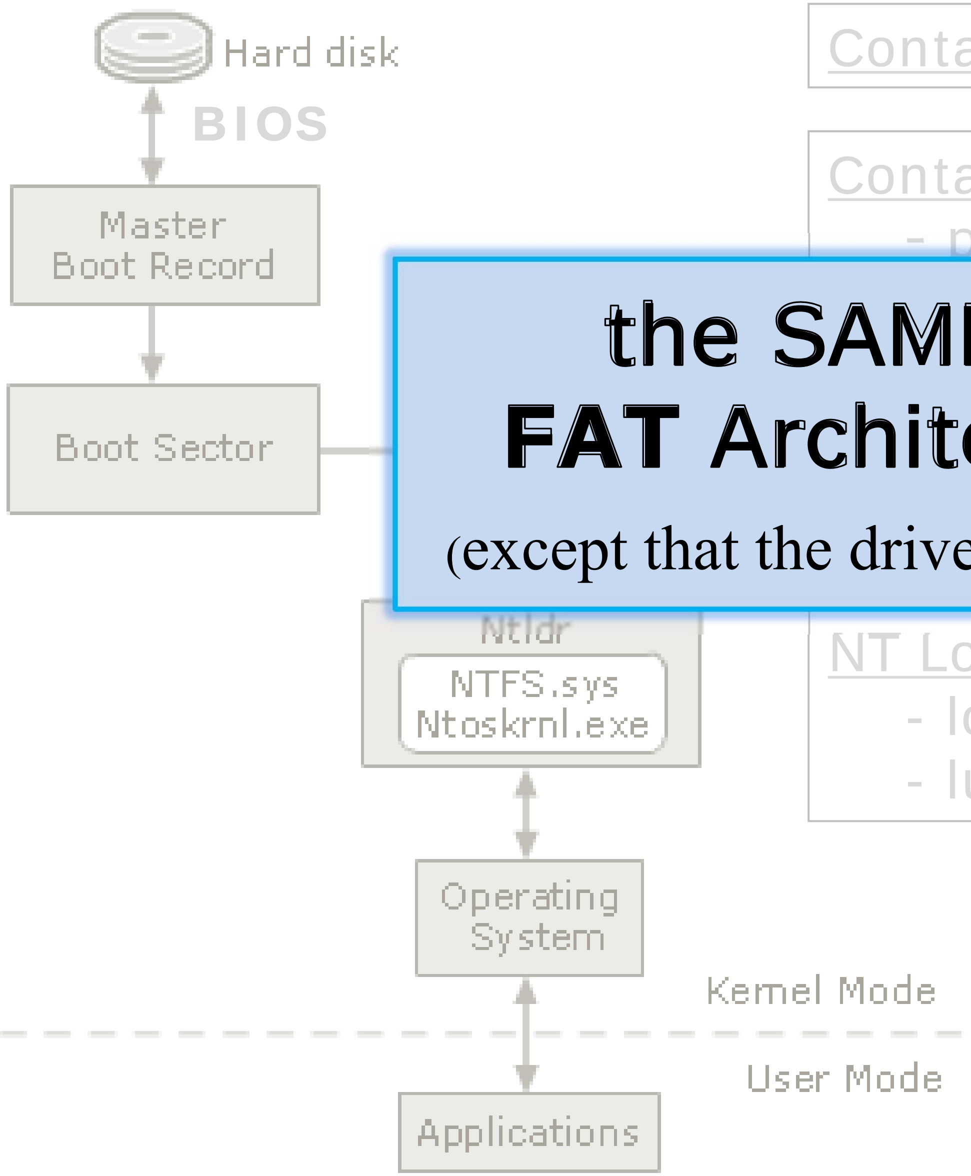
	56	57	58	59	60	61	62	63	64	65	67	68
Blocks	🗒	📁		🗒	📁		🗒	📁			🗒	📁

NTFS

NTFS File System

- NTFS Architecture
- NTFS Physical Structure
- NTFS Processes and Interactions

NTFS Architecture



Contains one or more partitions

Contains executable code
- partition table

olume
uctures
d *Ntldr*)

NT Loader
- load file system
- lunch boot menu (*boot.ini*)

the SAME to
FAT Architecture

(except that the driver is ***NTFS.sys***)

- NTFS File System
  - NTFS Architecture
  - NTFS Physical Structure
  - NTFS Processes and Interactions

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 don’t 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 no contained within MFT

Master File Table

- Attributes of files
- Attributes of folders

(*Most Important*)

Master File Table Copy

- a copy of MFT

NTFS Boot Sector

- NTFS 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-ba
  - Original Equipment Manufacturer Identification (OEM ID)
  - BIOS Parameter Block (PBP)
  - Extended BPB
  - Executable boot code

the SAME to
FAT File System

Component Boot Sector

0x00	0x03	0x0B	0x24	0x54	0x01FE
Jump	OEM ID	BPB	Ex-BPB	Bootstrap Code	Marker
3Bs	8Bs	25Bs	48Bs	426Bs	2Bs

the SAME to

FAT File System

(except the size of components)

- Jump: skip the next several non-executable bytes
- OEM ID: a
  - the name formatted the volumes
- BPB & Ex-BPB: information of volume
  - Byte Per Sector, Sectors Per Cluster, Media Descriptor, Total Sectors, Logical Cluster Number of the “MFT”, … …
- Bootstrap Code: executable code to start the OS
- Marker: end of boot sector

Master File Table

- Master File Table (MFT)
  - A relational database
  - Rows: file records
  - Columns: file attributes
  - Contains at least on entry for every files
- Metadata files
  - Describe the MFT
  - The first 16 records of MFT for metadata files
  - Begin with a dollar sign. e.g. $Mft, $MftMirr, $LogFile, …

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-data 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*

- NTFS File System
  - NTFS Architecture
  - NTFS Physical Structure
  - NTFS Processes and Interactions

Startup Process

- **BIOS** and CPU initiate the power-on self test
- BIOS find the boot device
- BIOS loads the first physical sector (**MBR**)
- Transfer CP
- MBR scans
- Load a copy of the **Boot Sector** for active partition
- Transfers CPU execution to Boot Sector

the SAME to
FAT File System

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

Compression of Files and Folders

- Moving Files or Folders within an NTFS Volume
  - keeps its compression state regardless of the compression state of the folder it is moved
- Copying Files or Folders within an NTFS Volume
  - takes on the compression state of the folder it is copied to.
- Between FAT and NTFS Volumes
  - FAT -> NTFS: takes on the compression attribute of the target folder
  - NTFS -> FAT: uncompressed

Hard Links

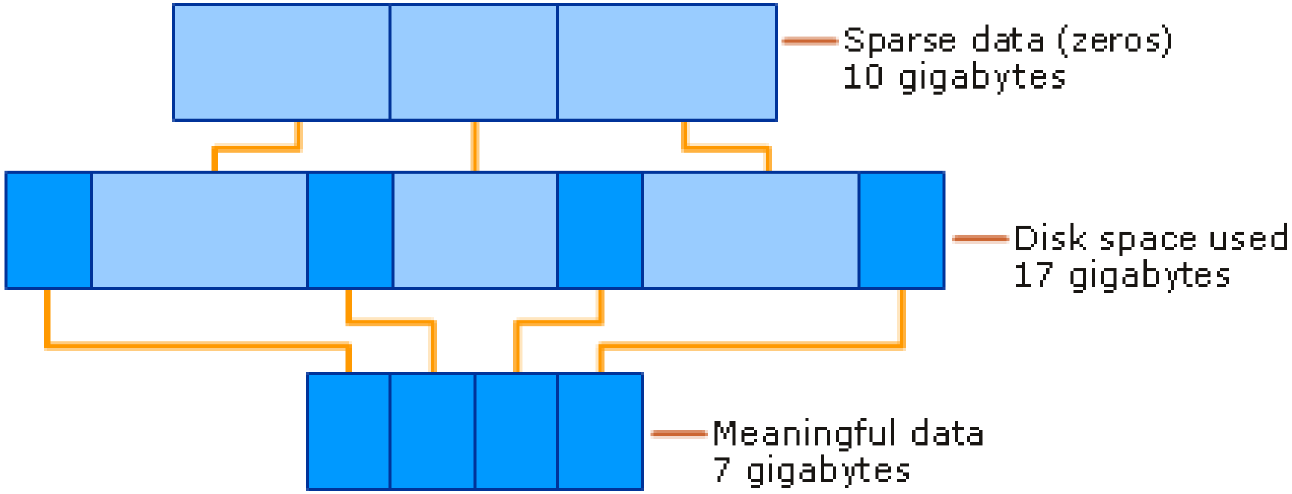
- Hard Links
  - Add a directory entry for the hard link without duplicating the original file
  - Application can modify a file by using any of its hard Links
  - Can’t give a file different security descriptors on a per-hard-link basis

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

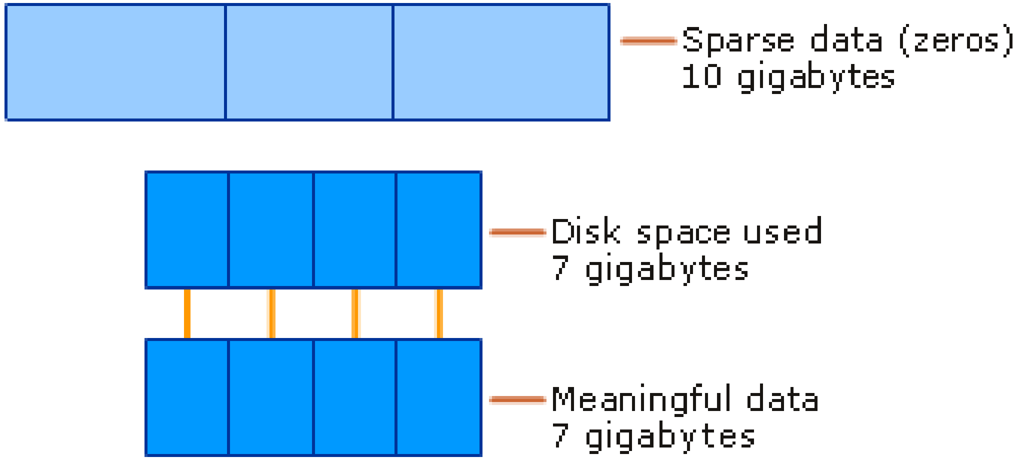


Sparse data (zeros)
10 gigabytes

Disk space used
17 gigabytes

Meaningful data
7 gigabytes

With Sparse File Attribute Set



Sparse data (zeros)
10 gigabytes

Disk space used
7 gigabytes

Meaningful data
7 gigabytes

File System Recoverability

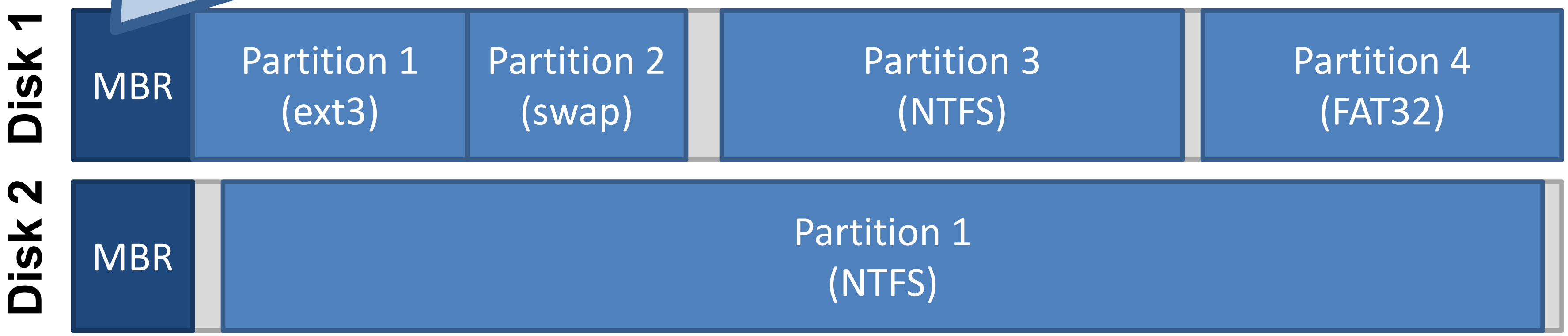
- Recovering NTFS File Structures
  - View each operation as a transaction and manages each one as an integral unit
  - Each transaction
    - Records the metadata operations in a log file cached in memory
    - Records the actual metadata operations in memory
    - Marks the transaction in the cached log file as committed
    - Flushes the log file to disk
    - Flushes the actual metadata operations to disk

MBR & MOUNT

The Master Boot Record

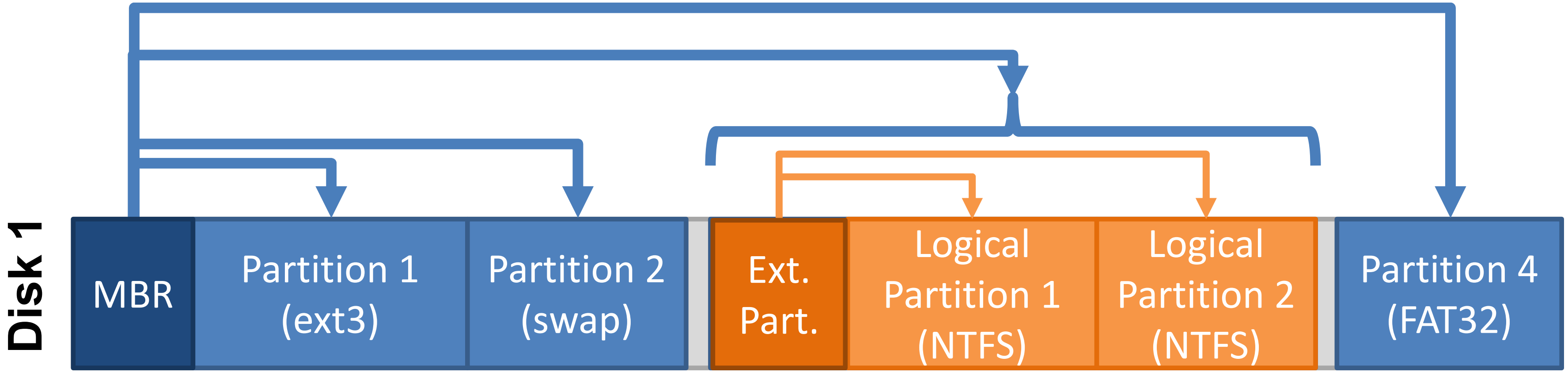
Address		Description	Size (Bytes)
Hex	Dec.		
0x000	0	Bootstrap code area	446
0x1BE	446	Partition Entry #1	16
0x1CE	462	Partition Entry #2	16
0x1DE	478	Partition Entry #3	16
0x1EE	494	Partition Entry #4	16
0x1FE	510	Magic Number	2
		Total:	**512**

Includes the starting LBA and length of the partition



Extended Partitions

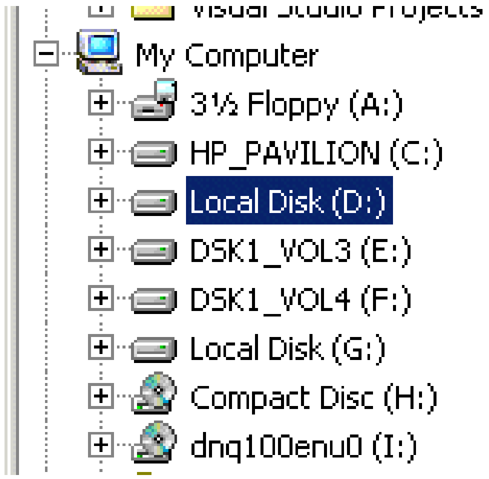
- In some cases, you may want >4 partitions
- Modern OSes support extended partitions



- 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

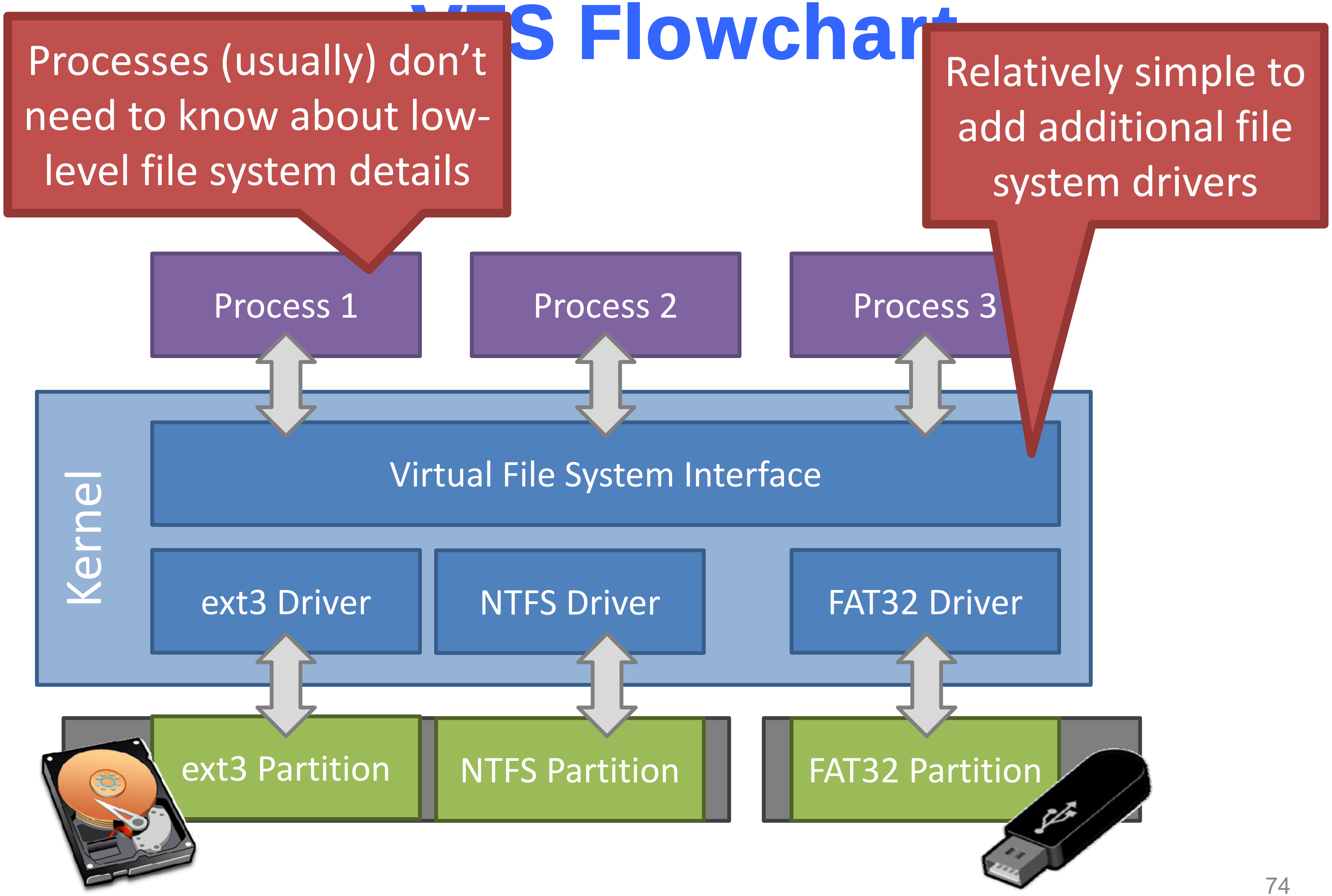
1. Read the super block for the target file system
   - Contains meta-data about the file system
   - Version, size, locations of key structures on disk, etc.
1. Determine the mount point
   - On Windows: pick a drive letter
   - On Linux: mount the new file system under a specific directory

```
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
```

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
Process 1
Process 2
Process 3
Kernel
Virtual File System Interface
ext3 Driver
NTFS Driver
FAT32 Driver
ext3 Partition
NTFS Partition
FAT32 Partition

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
- What does it mean to “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

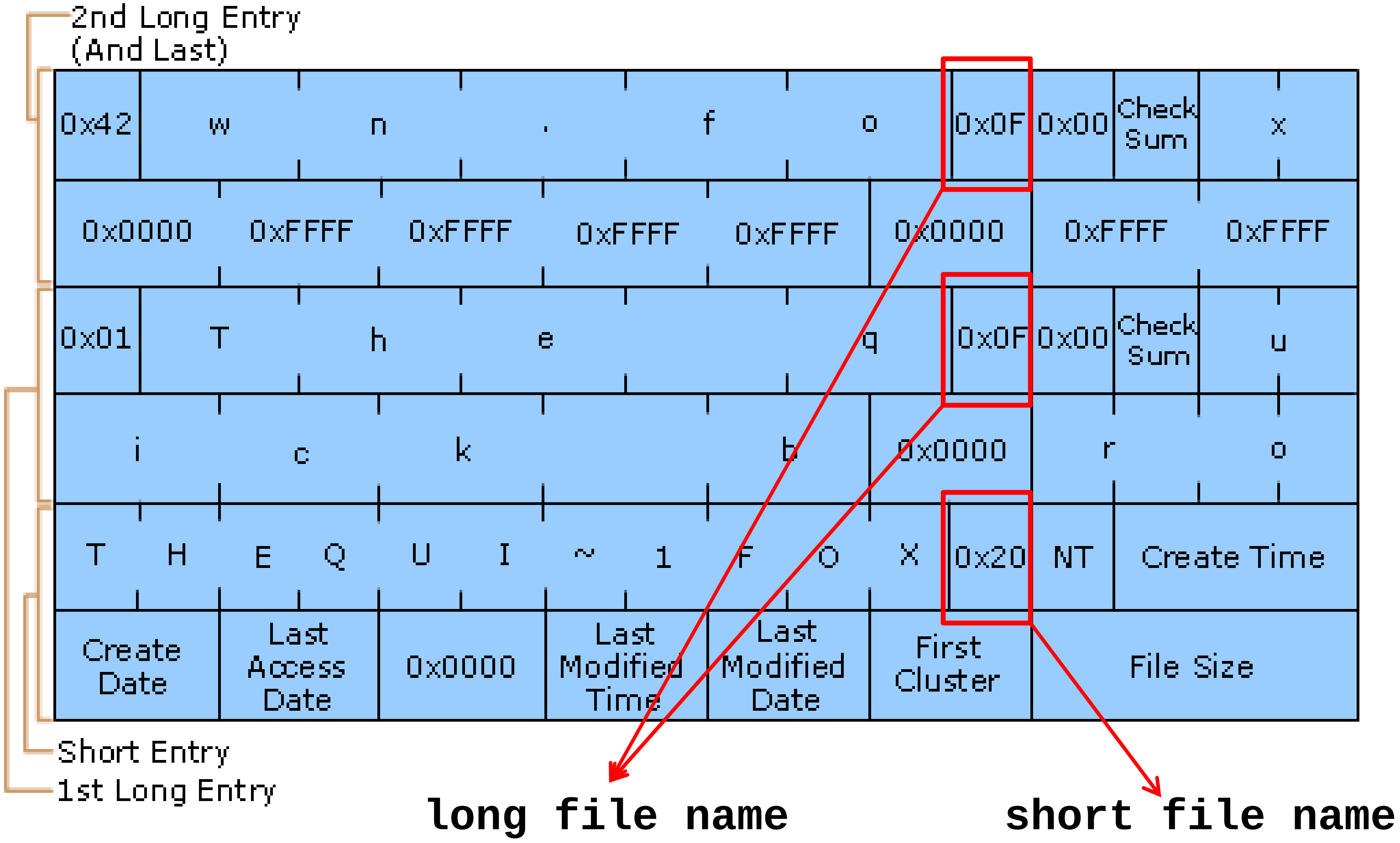
File System Durability & Crash Recovery

Yubin Xia & Rong Chen

Some of the materials are adopted from Frans'
6.828 course

Review: File Naming in FAT32

e.g. file with "The quick brown.fox" and "Thequi~1.fox"



2nd Long Entry (And Last)

0x42 w n . f o 0x0F 0x00 Check Sum x

0x0000 0xFFFF 0xFFFF 0xFFFF 0xFFFF 0x0000 0xFFFF 0xFFFF

0x01 T h e q 0x0F 0x00 Check Sum u

i c k b 0x0000 r o

T H E Q U I ~ 1 F O X 0x20 NT Create Time

Create Date | Last Access Date | 0x0000 | Last Modified Time | Last Modified Date | First Cluster | File Size

Short Entry

1st Long Entry

long file name

short file name

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

Bind Short Name and Long Name

```
int i,j,chknum=0;

for (i=11; i>0; i--)
{
    chksum = ((chksum & 1) ? 0x80 : 0)
             + (chksum >> 1)
             + shortname[j++];
}
```

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?

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

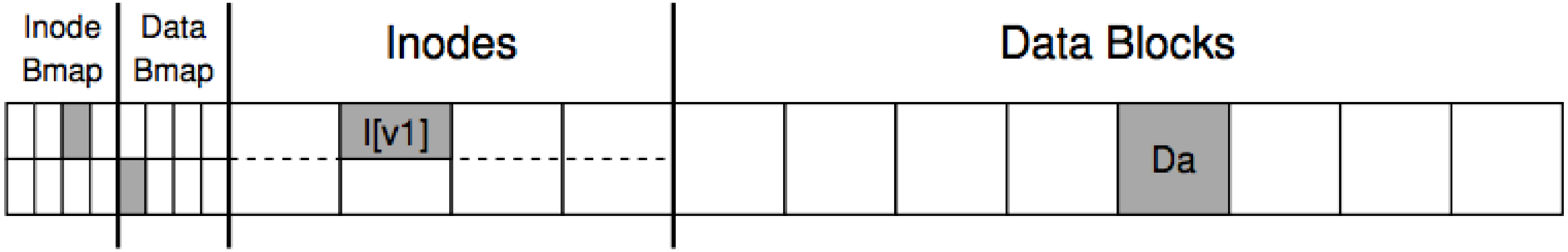
“Here is all of my data. But some of the metadata is wrong.”

Crash recovery is much harder than performance

"what if a crash occurred at this point?”

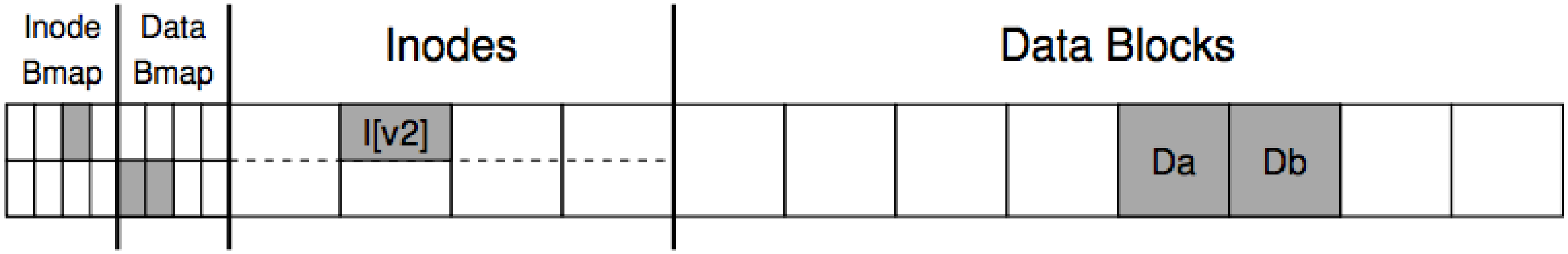
An Example: Append a File

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



An Example: Append a File

- Inside of I[v2]:
  - owner : yubin
  - 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 (Db) 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 (Db)
- 2. The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2])
- 3. The bitmap (B[v2]) and data block (Db) 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: 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 meta-data, such as file size

2. Soft update (FreeBSD fs modified on FFS)

   Soft update, 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 readir ally, not seeking
- still a long tim size

Consider the

- Scan the disk writes
- Just like find ntire 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?

Flash File System and GFS

Rong

Outline

- Flash file system
- Google file system

INTRO TO FLASH FILE SYSTEM

What is flash file system?

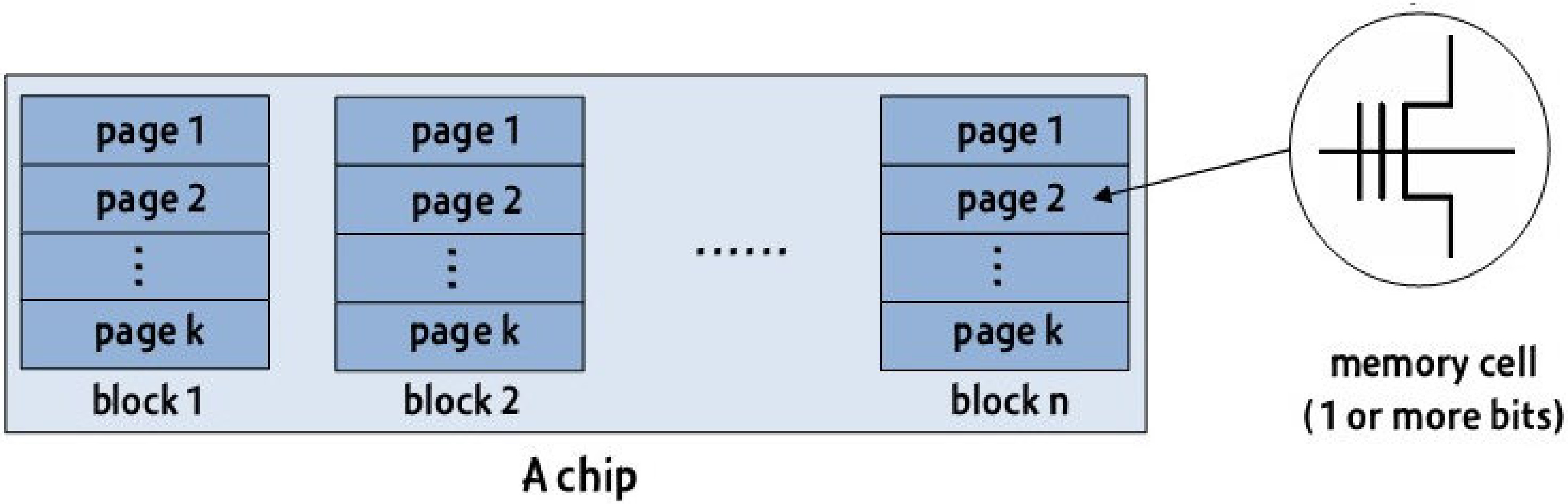
- Flash file system is designed for storing files on flash memory devices

- Can traditional file system be used on flash file system?
  - Of course, NO.

- So, what is the big difference between flash disk and magetic disk

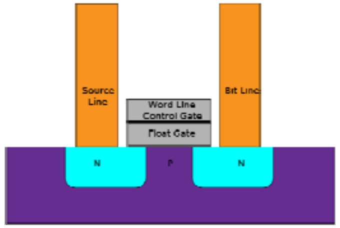
Flash disk organization (1/2)

- Flash disk organization
  - A **chip** (e.g. 1GB) => **blocks** (e.g. 512KB) => **pages** (e.g. 4KB) => **cells**

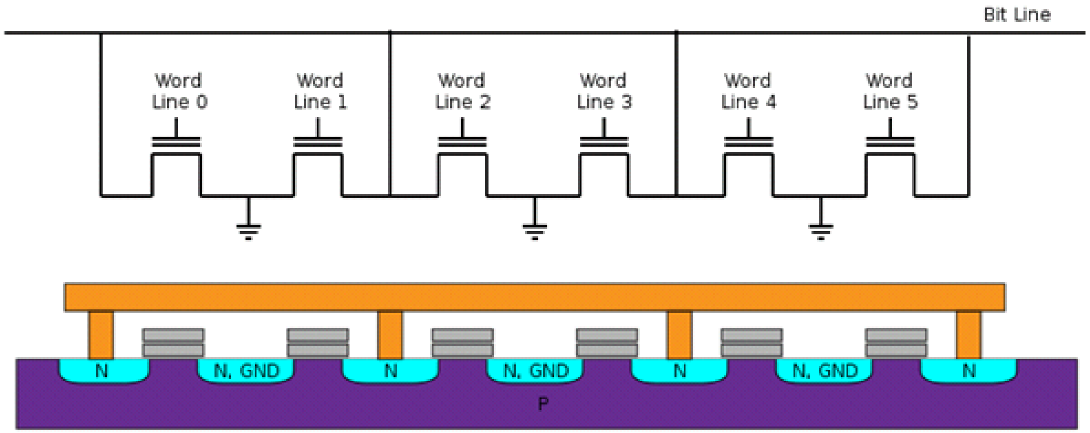


Flash disk organization (2/2)

- 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)

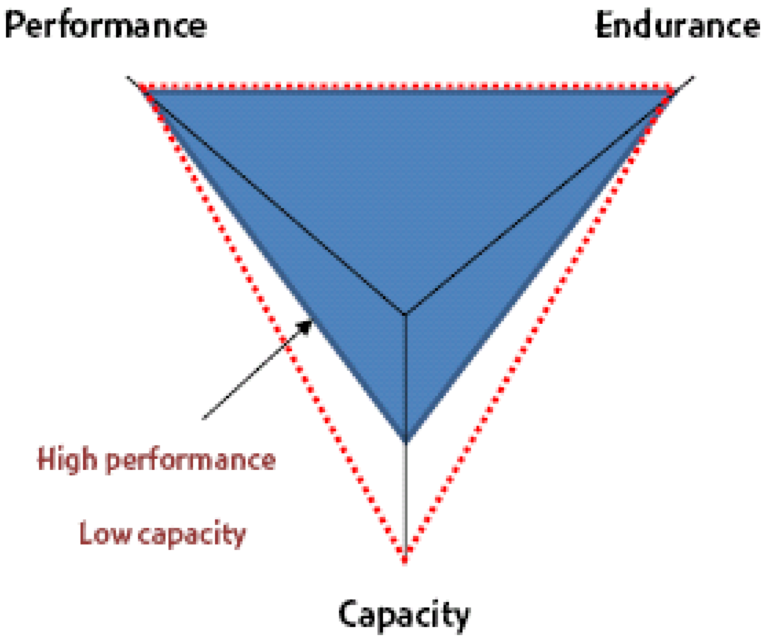


(Single-Level Cell, SLC)

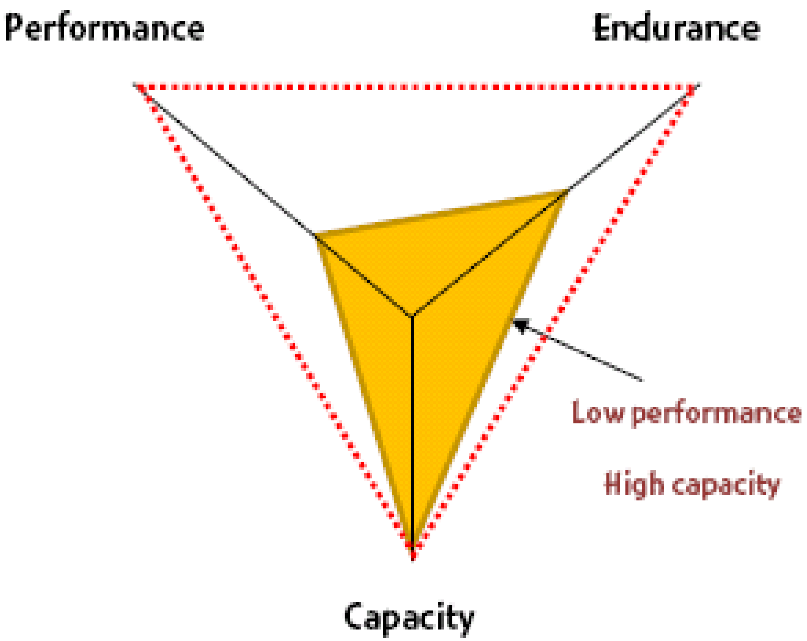


(Multi-Level Cell, MLC)

Comparisons of SLC and MLC Flashes



SLC Flash Memory
(1 bit / cell)



MLC Flash Memory
(2 bits / cell)

Flash disk characters (1/2)

- Asymmetric **read/write** and **erase** operations
  - A “**page**” is a unit of read/write
  - A “**block**” is a unit of erase

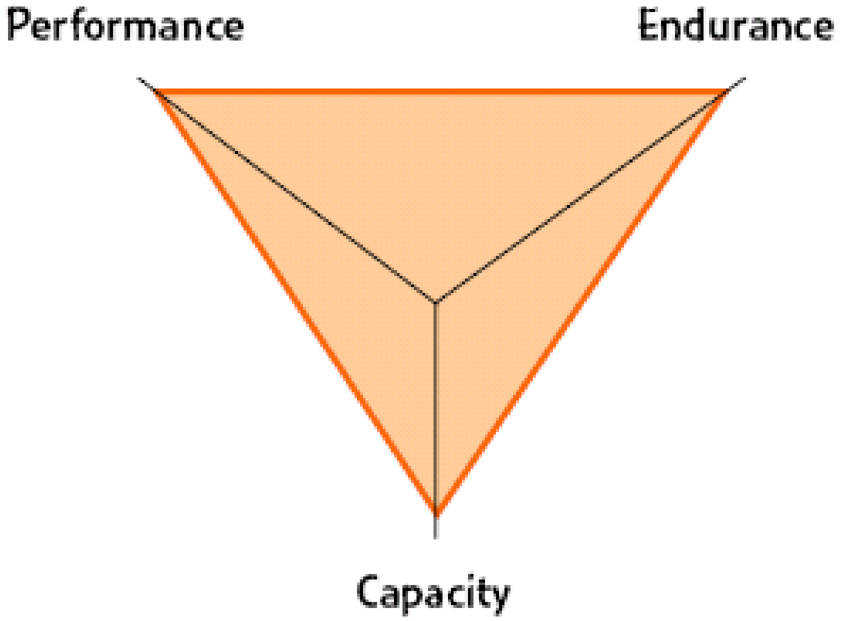
- Physical restrictions
  - **Erase-before-write** restriction
  - The number of erase cycles allowed for each block is limited

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)

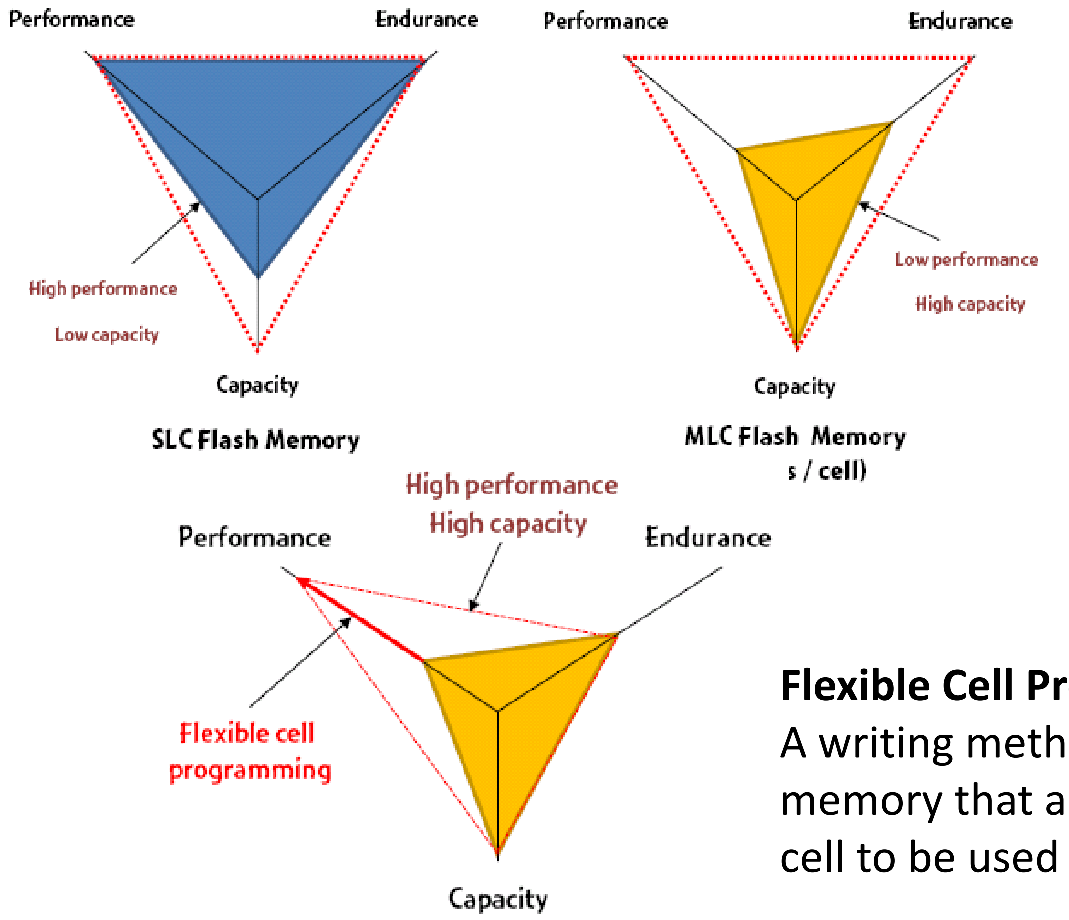
A real flash file system study - flexFS

- 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

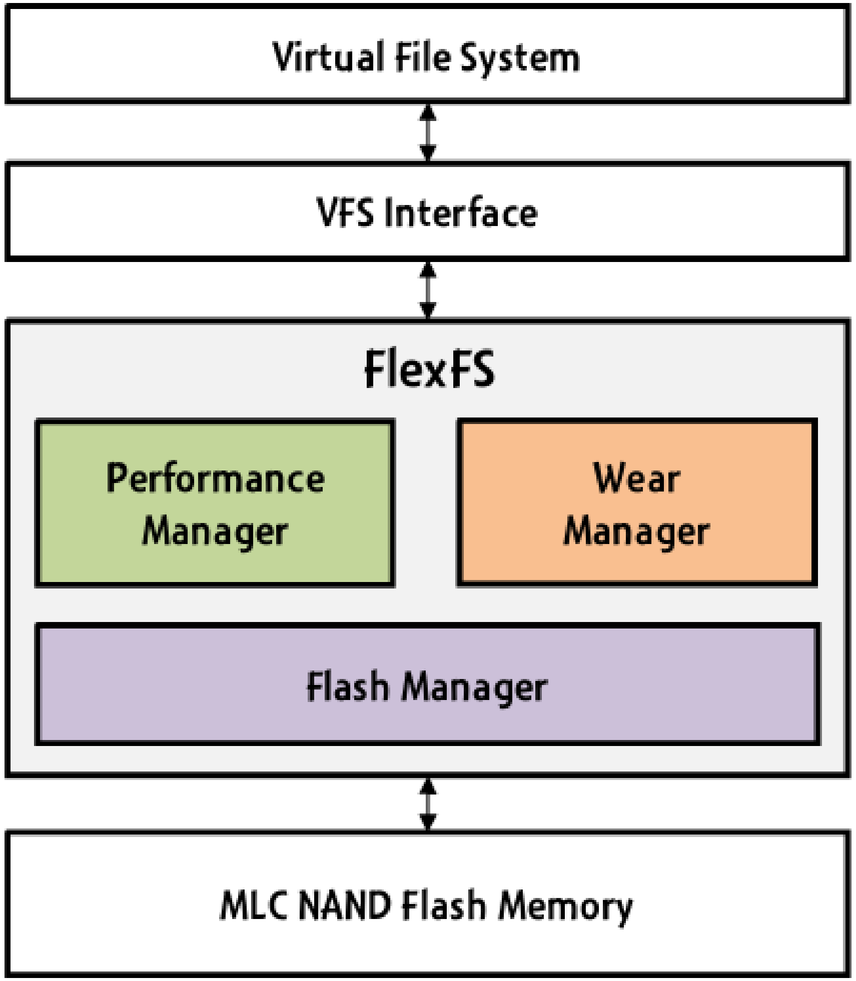


Flexible Cell Programming:
A writing method of MLC flash memory that allows each memory cell to be used as SLC or MLC

Our Approach

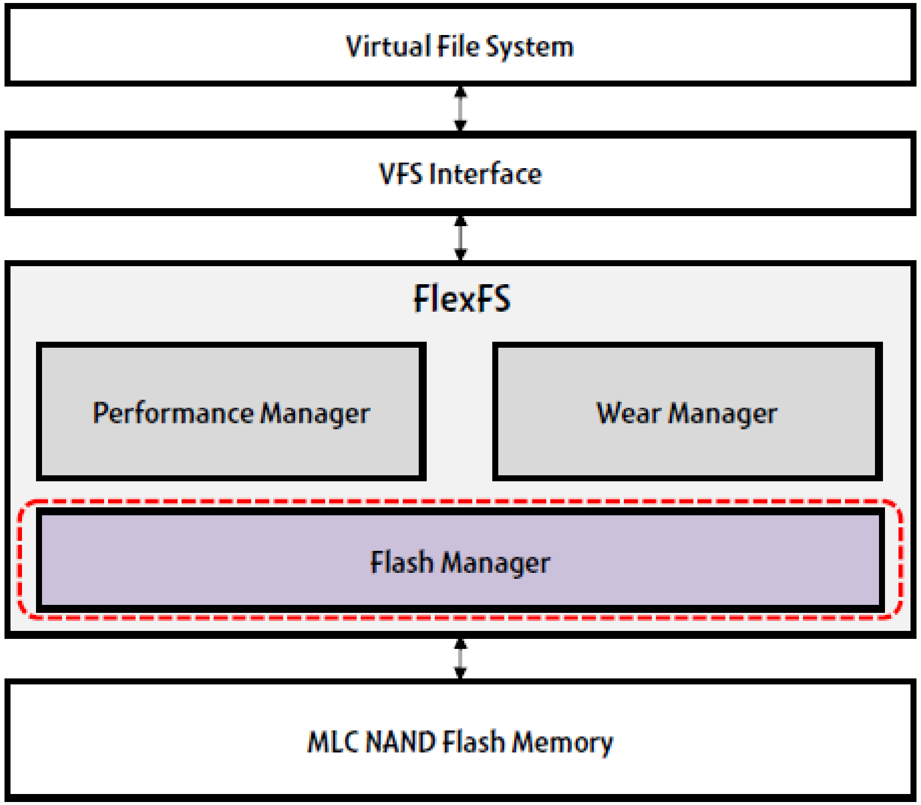
- Proposes a flash file system called FlexFS
  - Exploits flexible cell programming
  - Provides the high performance of SLC flash memory and the capacity of MLC flash memory
  - Provides a mechanism that copes with a poor wear characteristic of MLC flash memory
  - Designed for mobile systems, such as mobile phones

Architecture of FlexFS



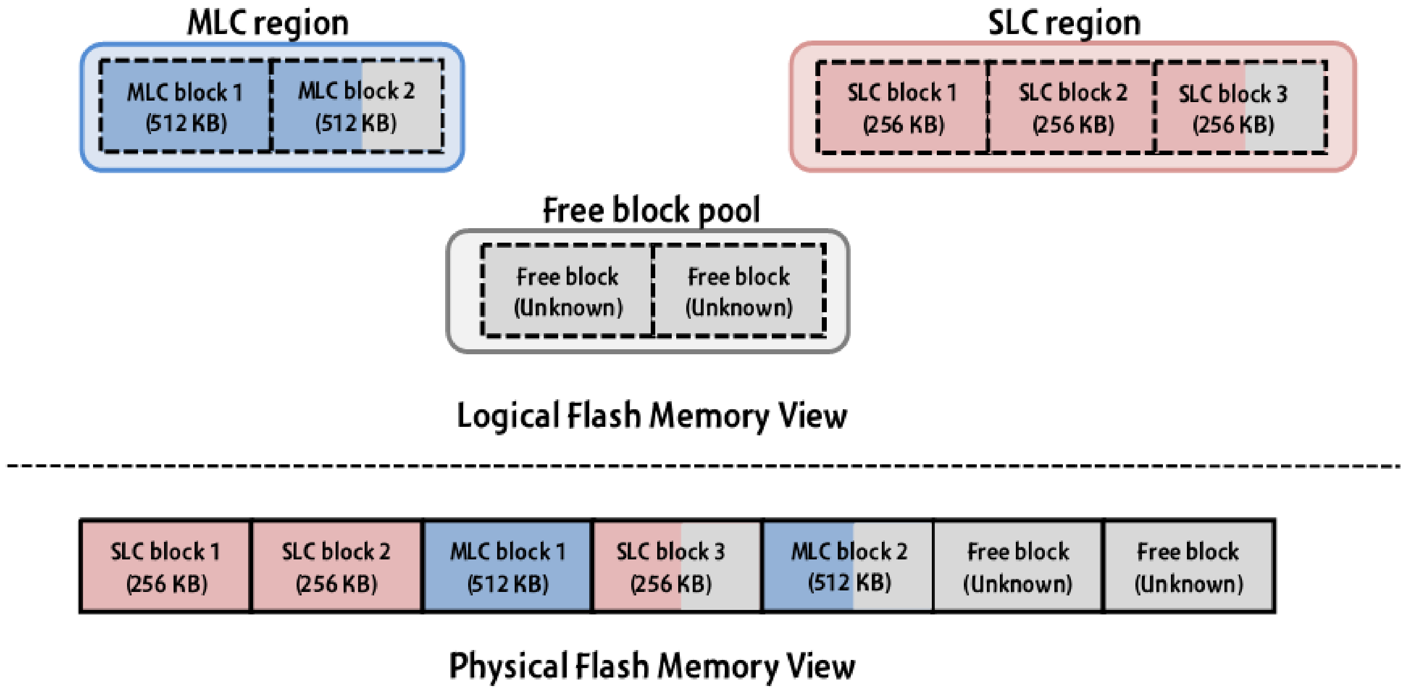
- Flash Manager
  - Manages heterogeneous cells
- Performance manager
  - Exploits I/O characteristics
  - To achieve the high performance and high capacity
- Wear manager
  - Guarantees a reasonable lifetime
  - Distributes erase cycles evenly

Overall Architecture

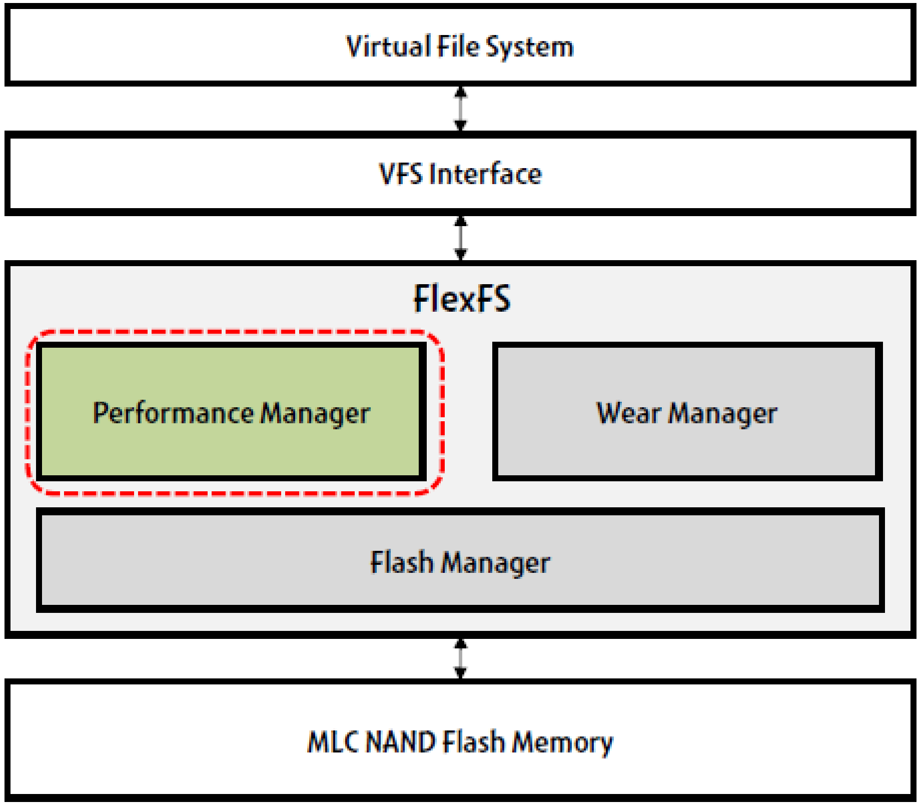


FlexFS - Flash Manager

- 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



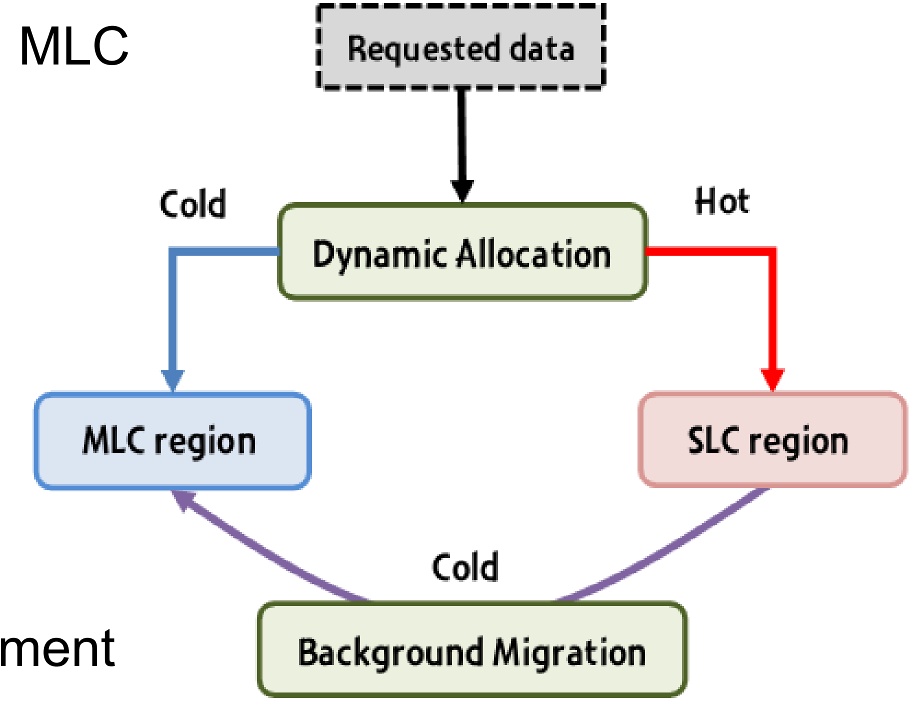
Overall Architecture



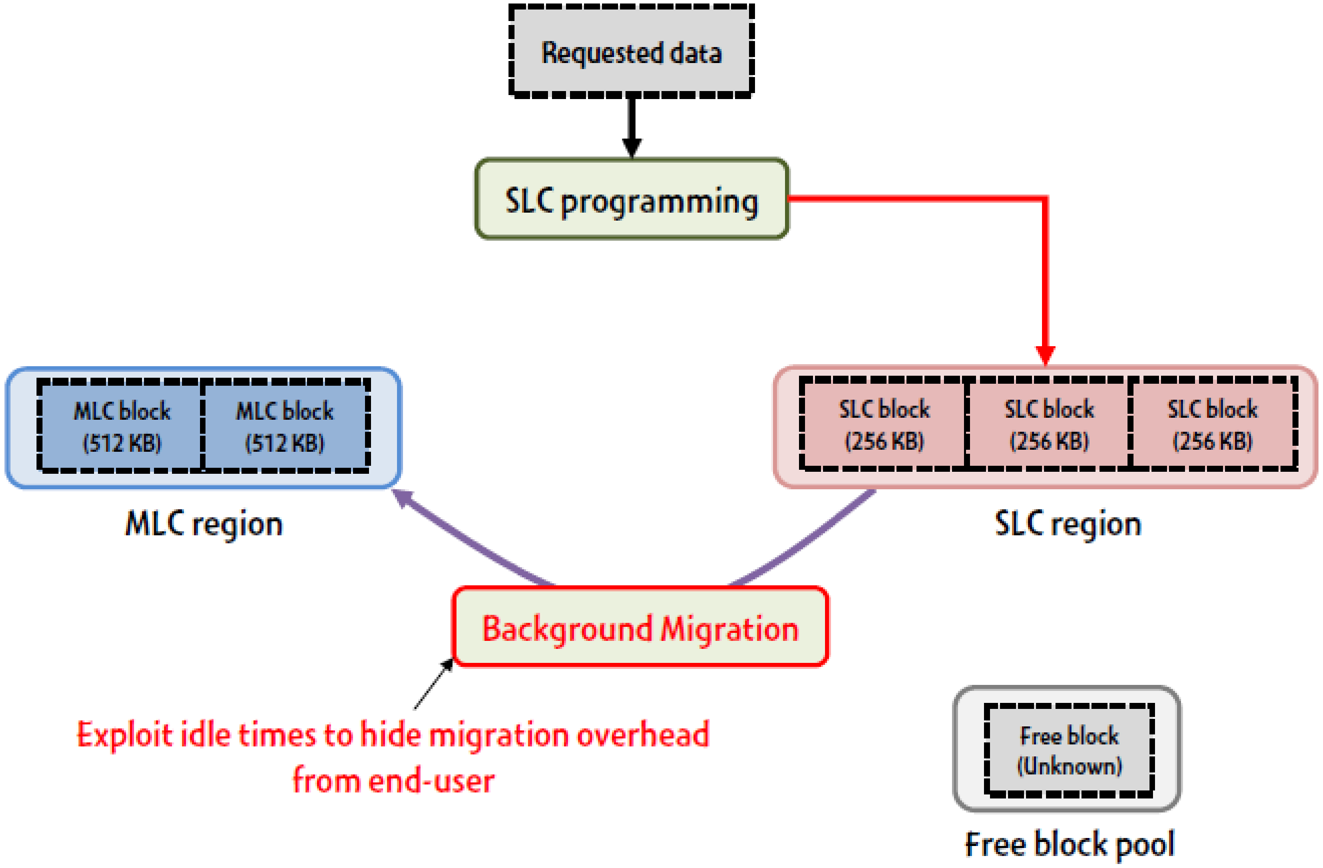
FlexFS – Performance Manager

- 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



Requested data

SLC programming

MLC block (512 KB)

MLC block (512 KB)

MLC region

SLC block (256 KB)

SLC block (256 KB)

SLC block (256 KB)

SLC region

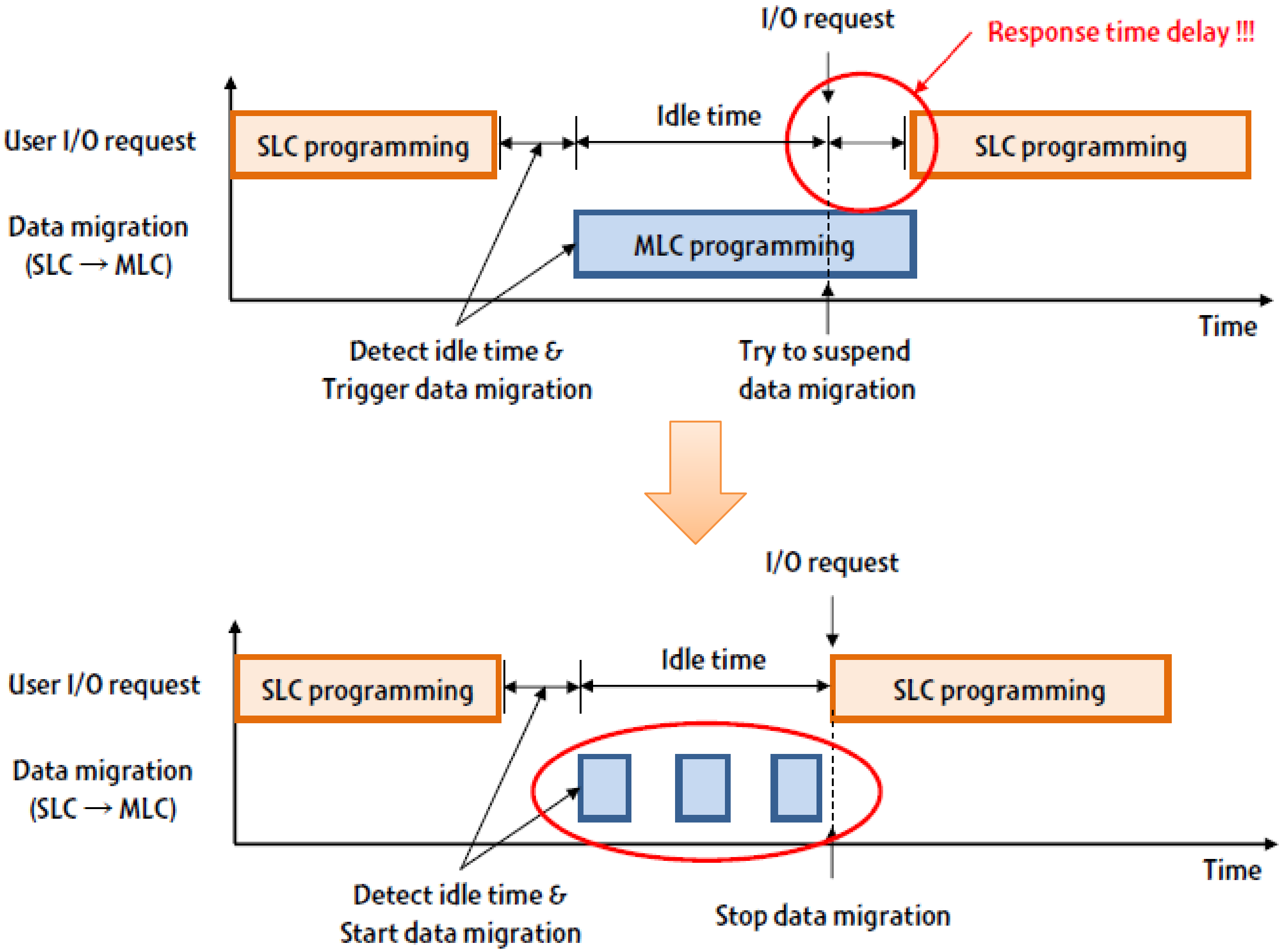
Background Migration

Exploit idle times to hide migration overhead from end-user

Free block (Unknown)

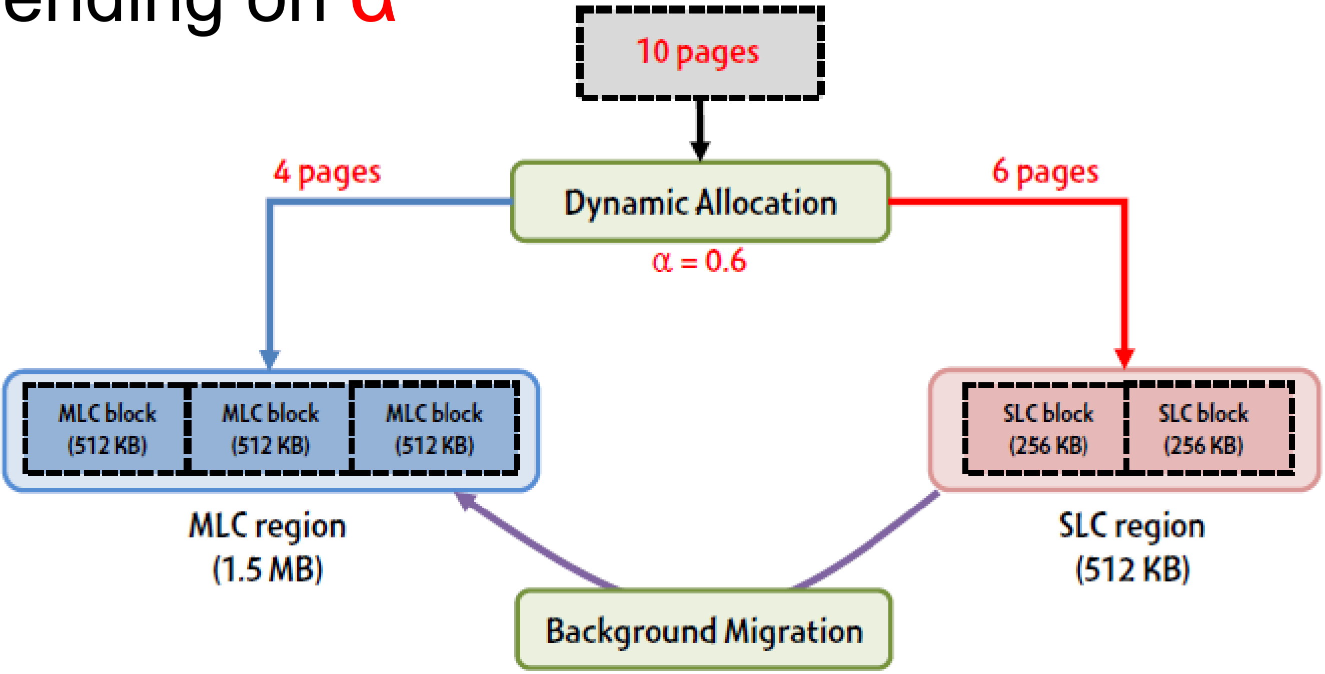
Free block pool

Background Migration



Dynamic Allocation

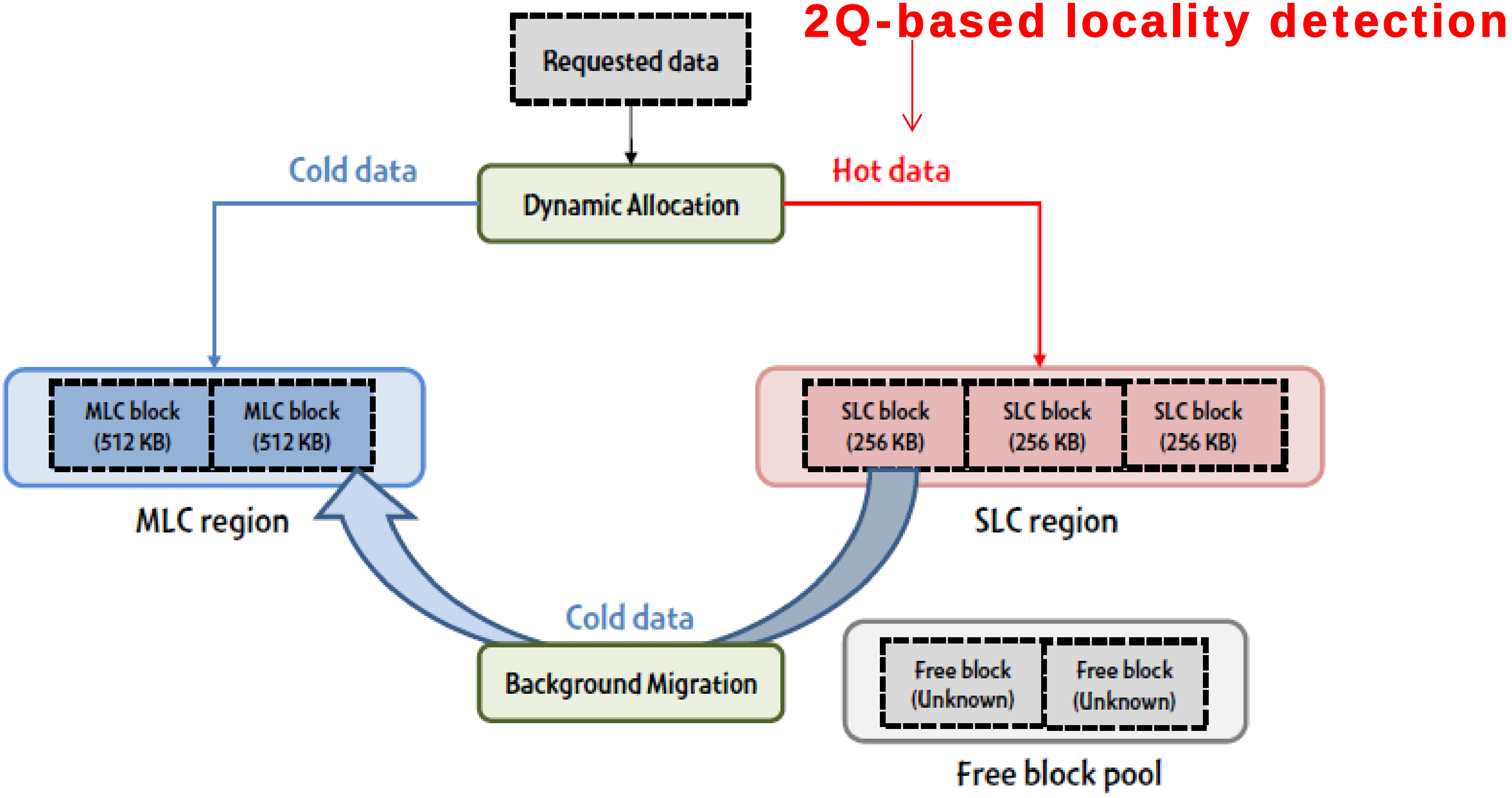
- Distributes the incoming data across two regions depending on α



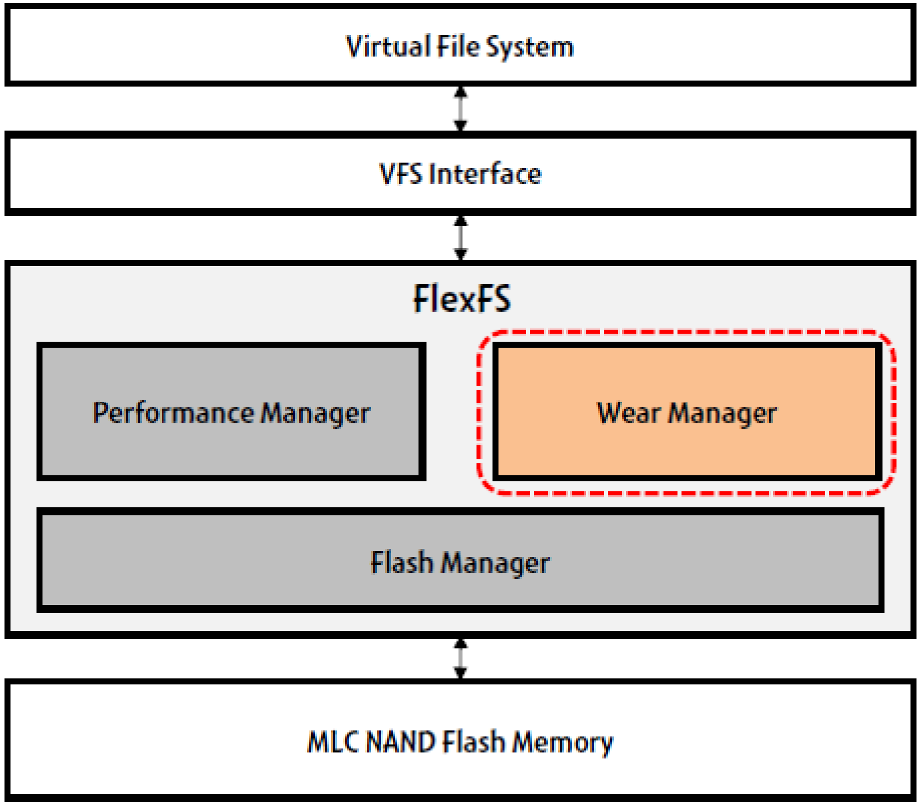
$$\alpha = \frac{T_{predict}}{N_p \cdot T_{copy}} \quad (\text{If } T_{predict} \geq N_p \cdot T_{copy}, \text{ then } \alpha = 1.0)$$

Locality-aware Data Management

- Data migration for hot data is unnecessary

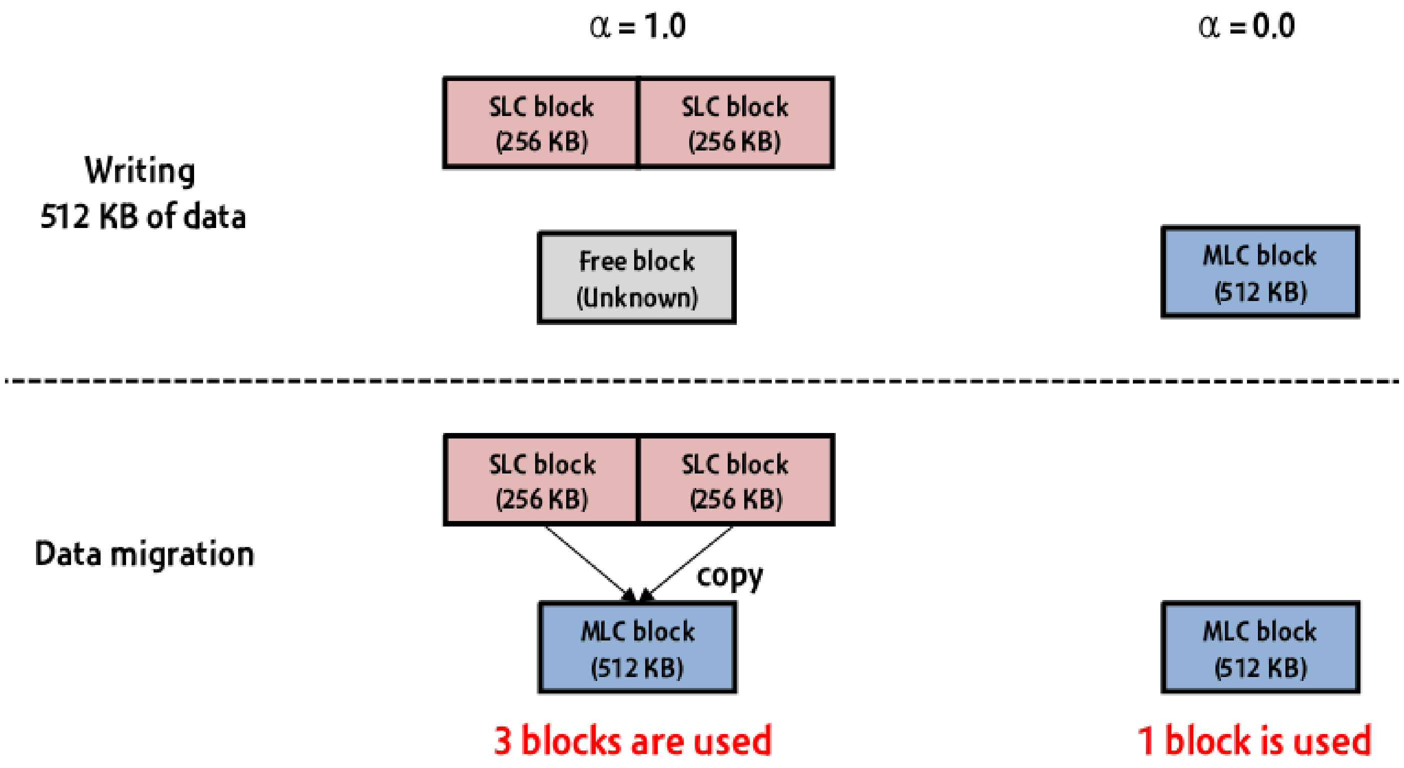


Overall Architecture

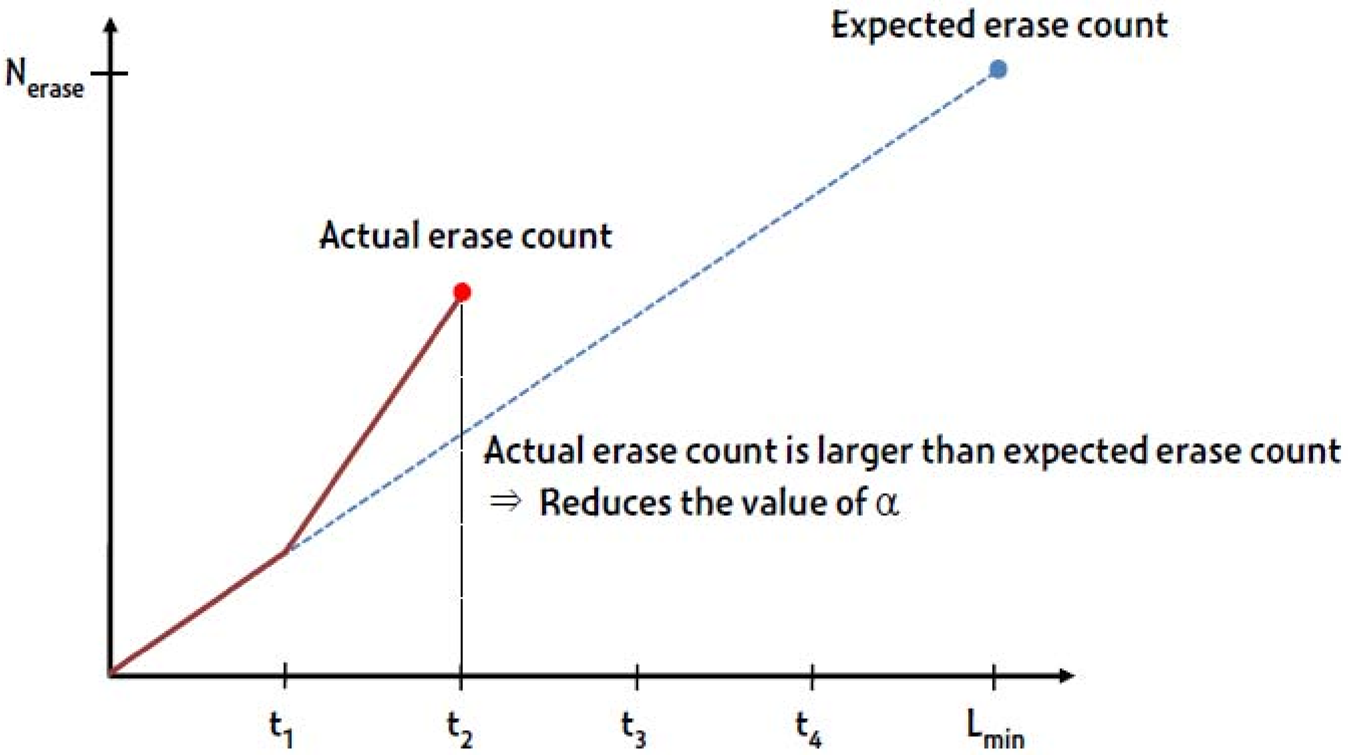


FlexFS– wearing rate control

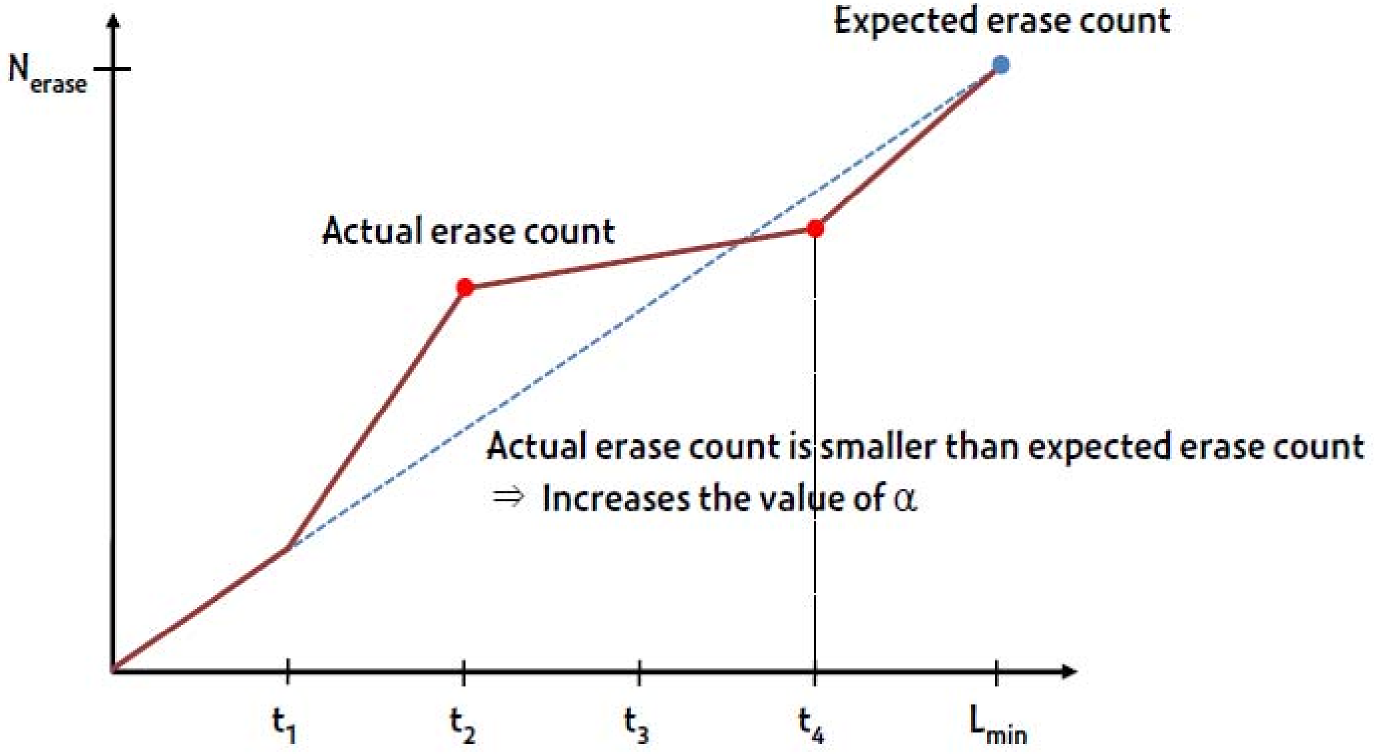
- How FlexFS control the wearing rate
  - The wearing rate is directly proportional to the value of α



Wearing Rate Control

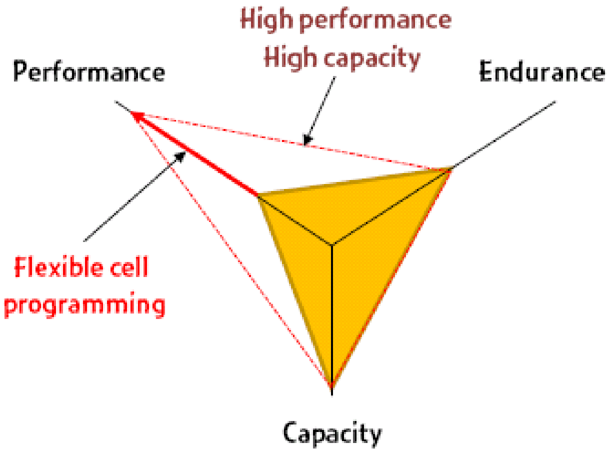


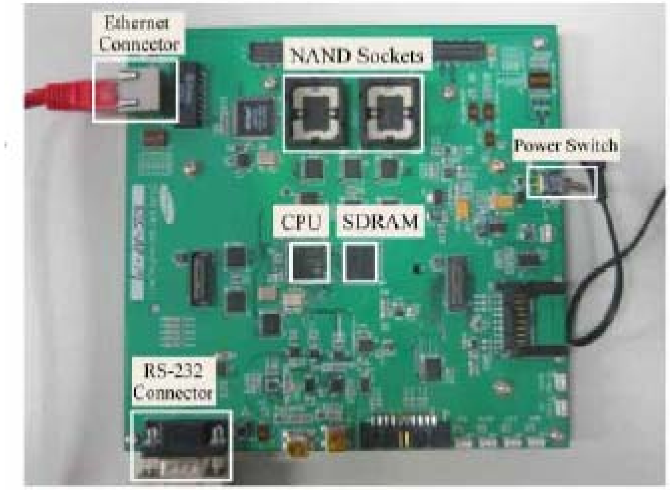
Wearing Rate Control



Conclusion

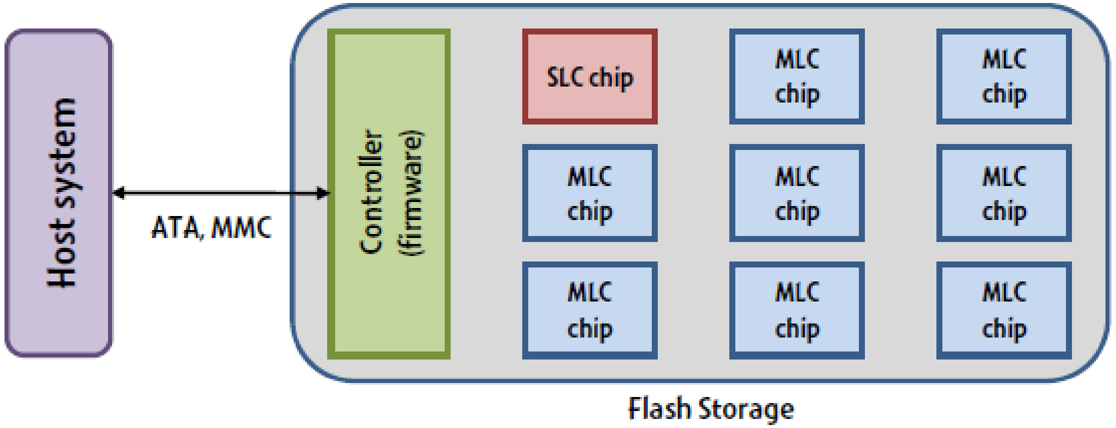
- Propose a new file system for MLC NAND flash memory
  - Exploits the flexible cell programming to achieve the SLC performance and MLC capacity while ensuring a reasonable lifetime





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



INTRO TO GFS

Introduction to google file system

- Shares same goals as previous distributed file systems, 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

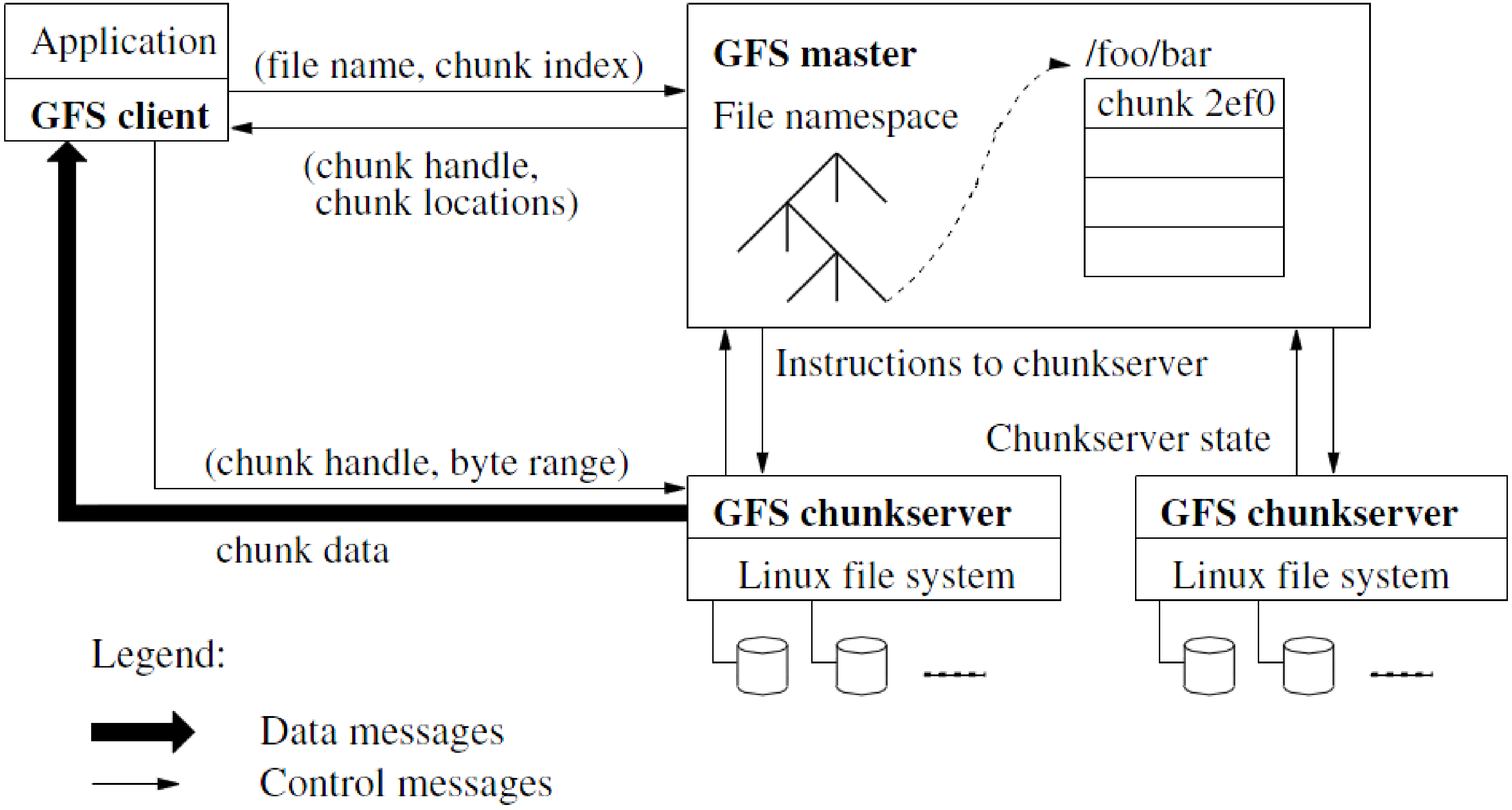
Design Assumptions

- 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**

Architecture



Architecture Design

- **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

Single Master

- 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

- 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

- Master monitors chunk location through *heartbeat* messages with each chunk.

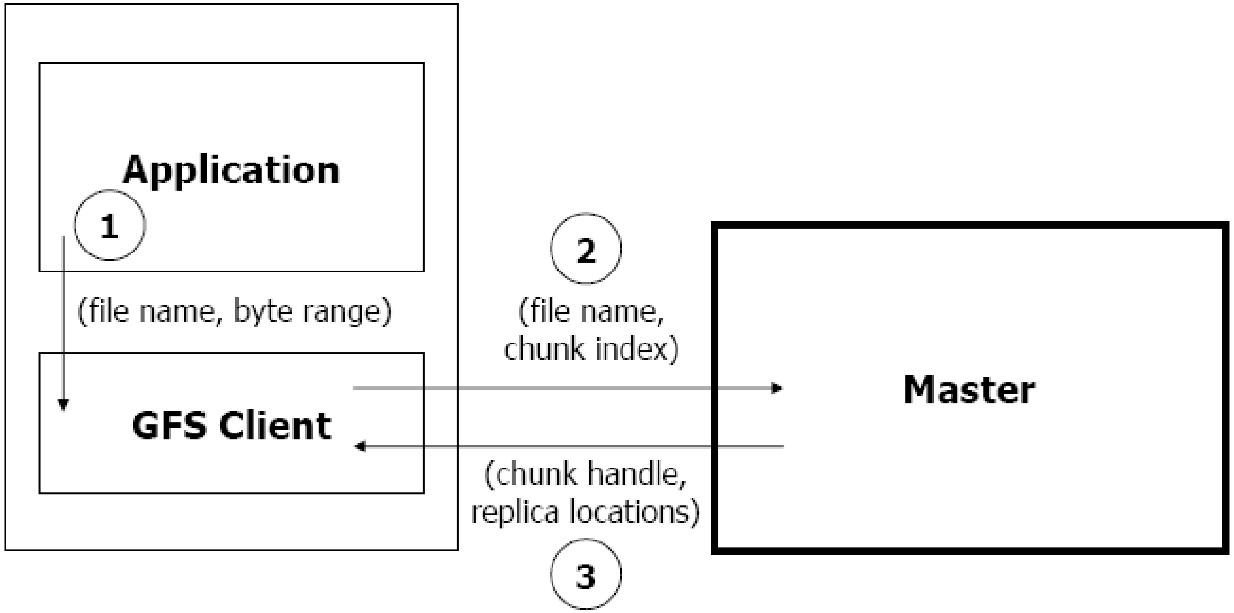
Chunkserver

- 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

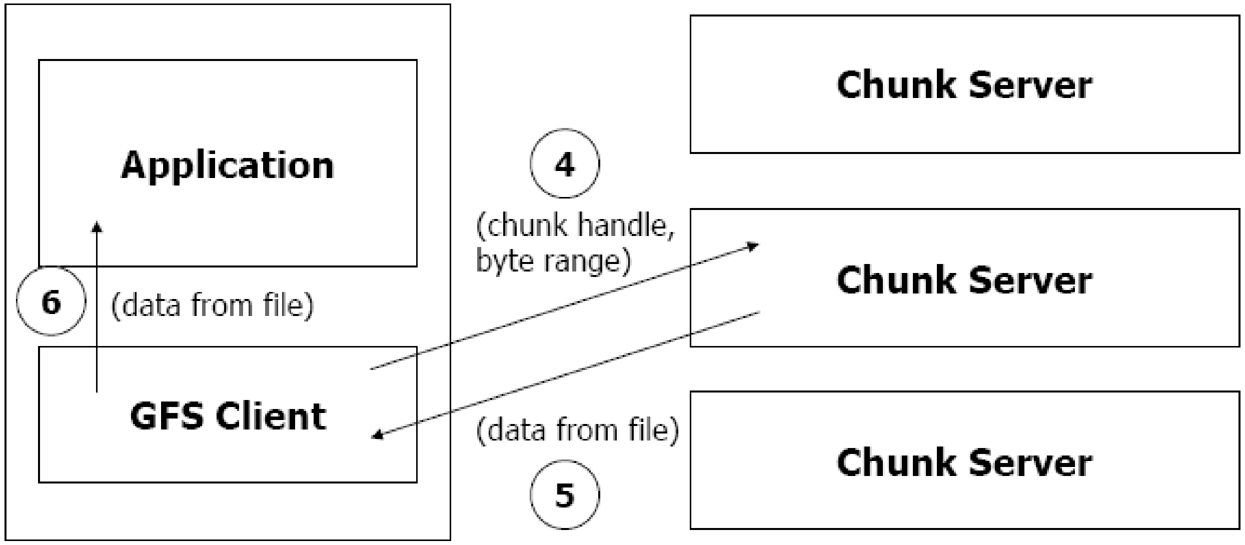
System Interactions

- 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



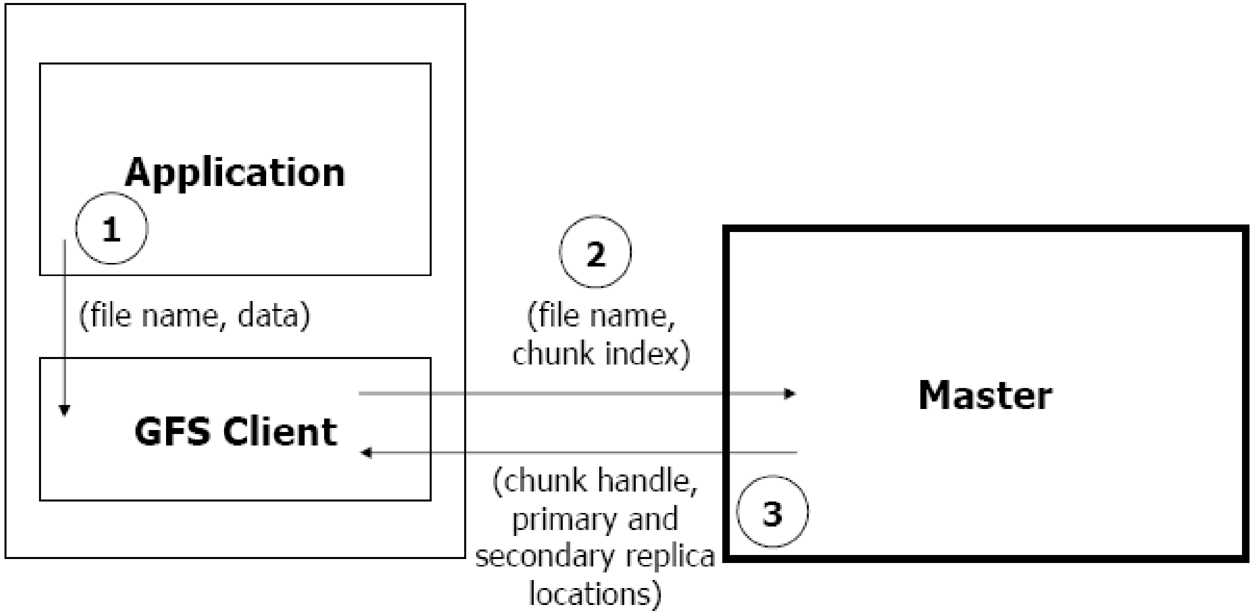
Read Algorithm



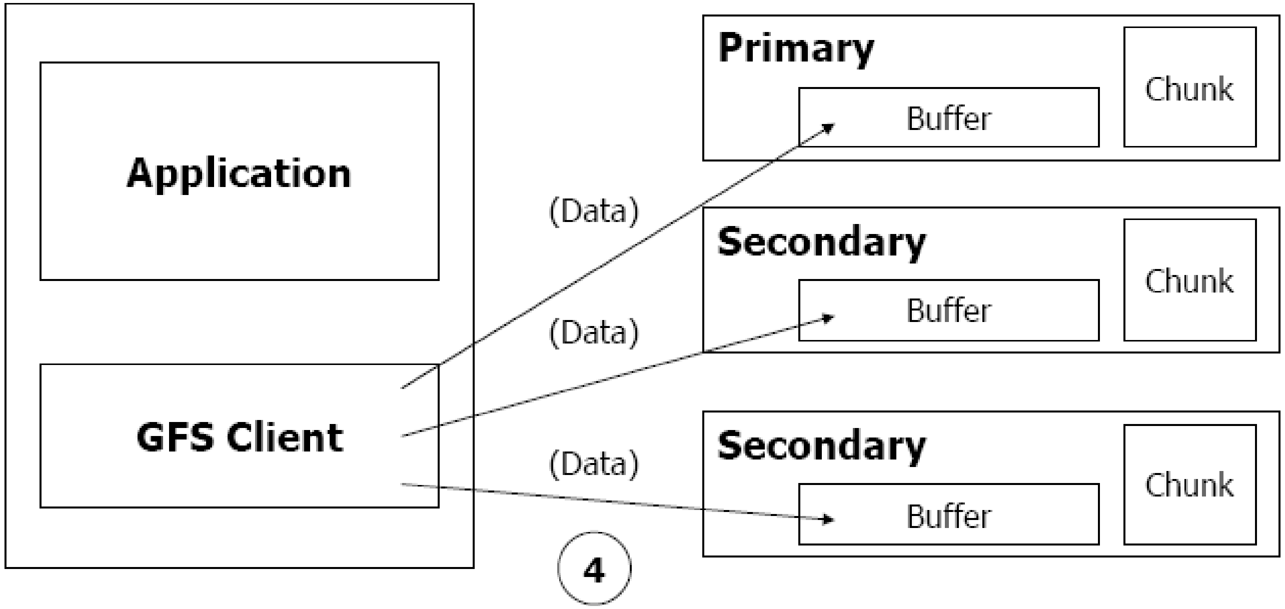
Read Algorithm

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.

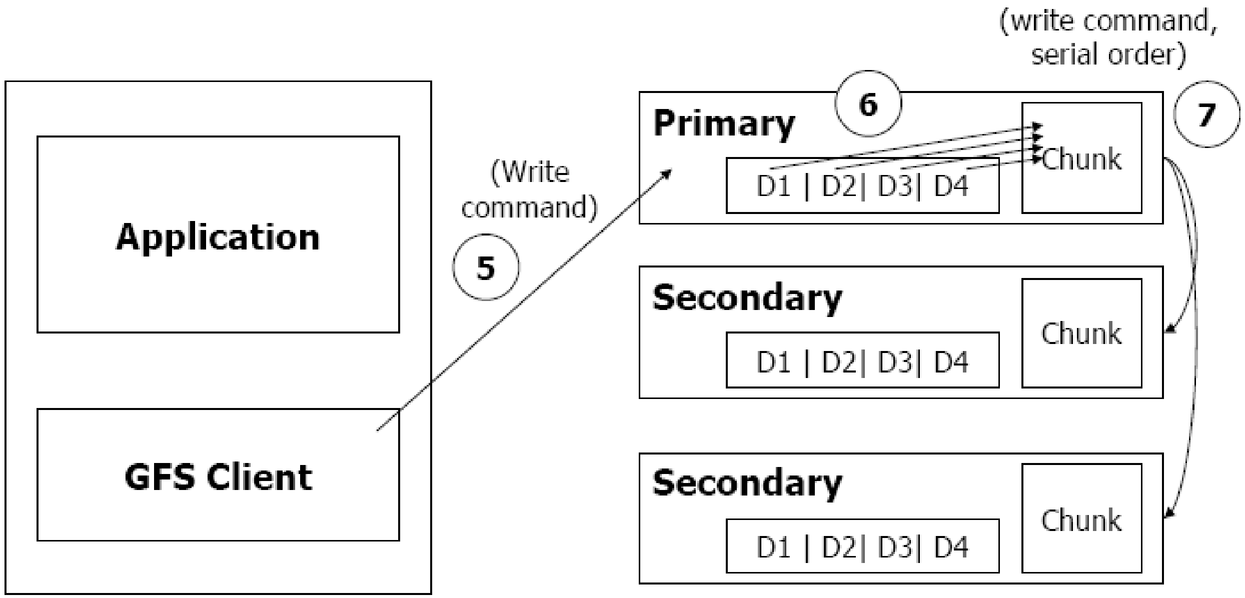
Write Algorithm



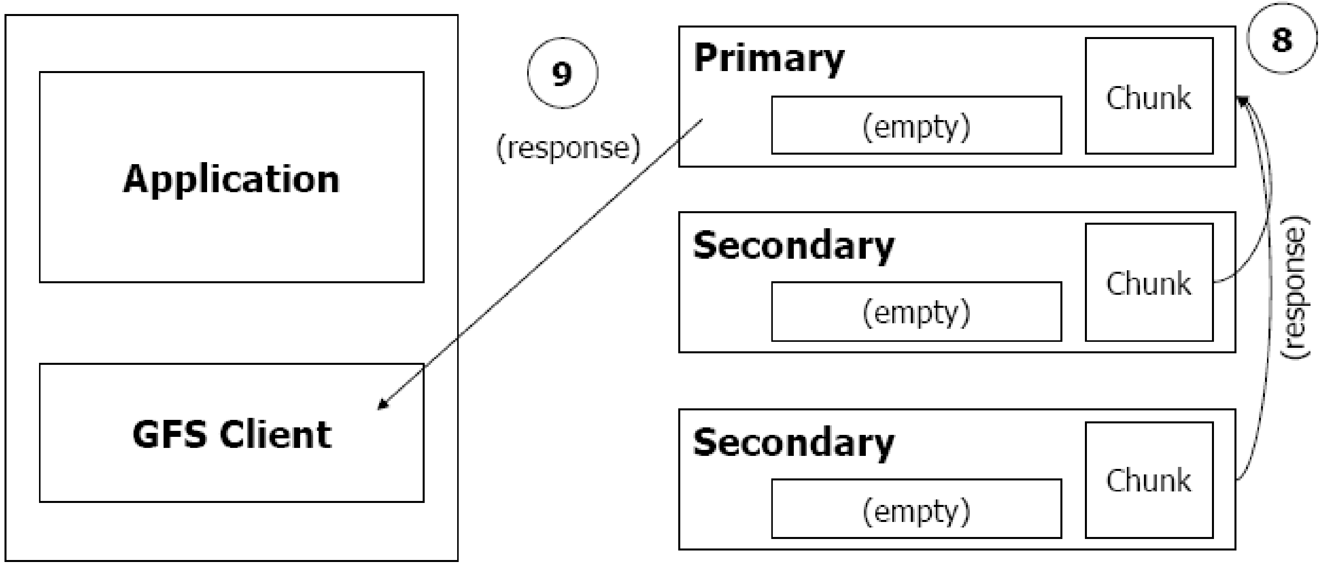
Write Algorithm



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

- Metadata fit in memory
- 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

Thanks

- Next class
  - Journaling

Journaling: xv6 and ext3

Yubin Xia & Rong Chen

Ack: slides adopted from Frans' 6.828 course

Outline

- Recap
- Journaling
- Xv6 journaling
- Ext3 journaling

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: Terms for properties of fs ops

What effects will app see after restart + recovery

- creat(“a”); fd = creat(“b”); write(fd,…); crash

Durable: 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

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

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

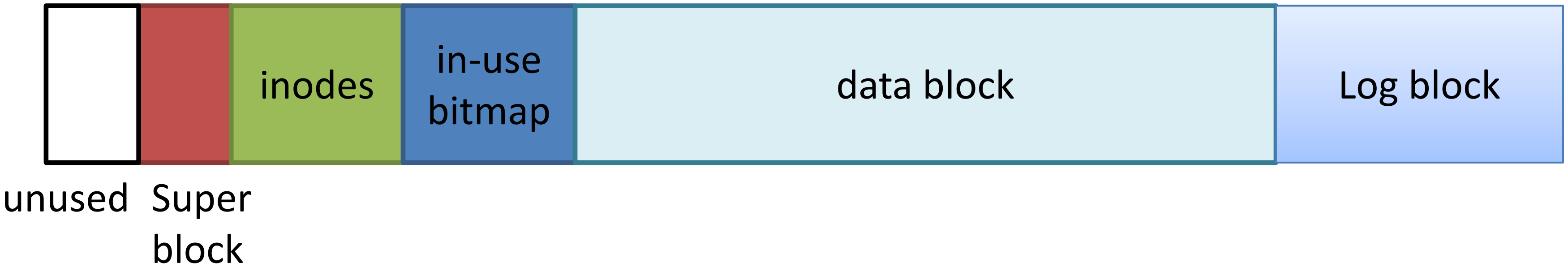
To ensure atomicity:

- 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

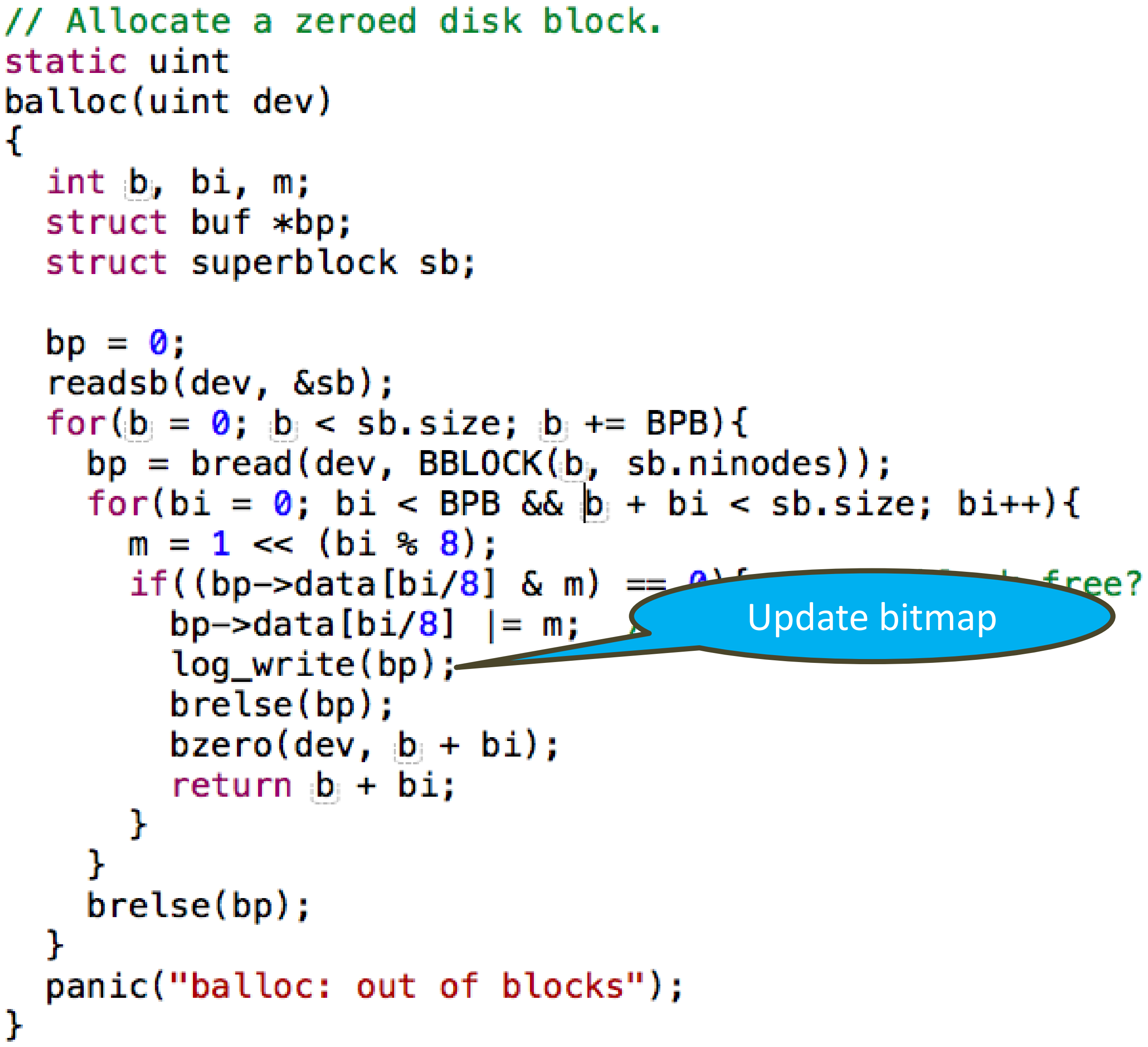
Xv6's simple logging

FS has a log on disk syscall:

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

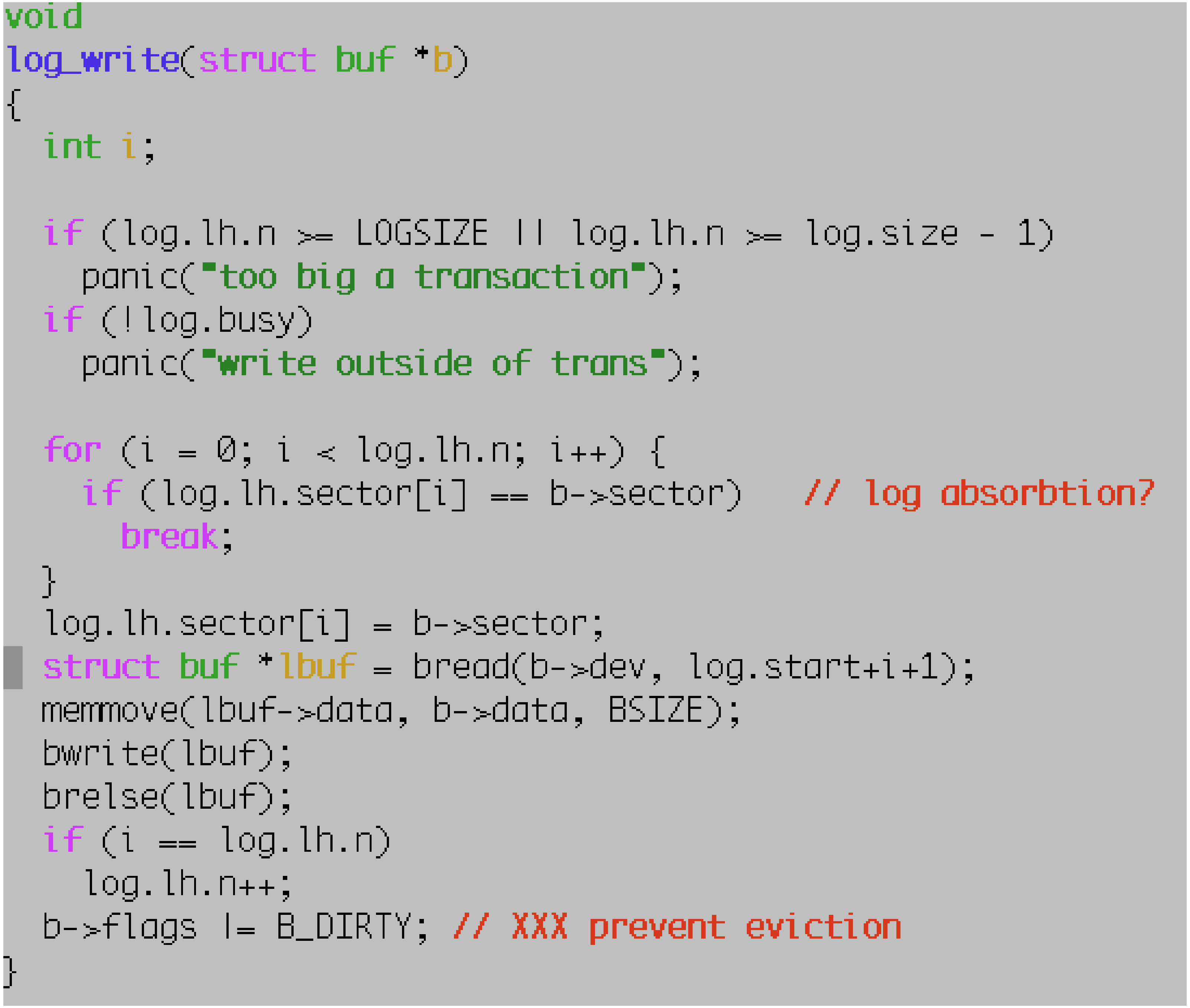


Recap: balloc



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

Update bitmap



```
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:

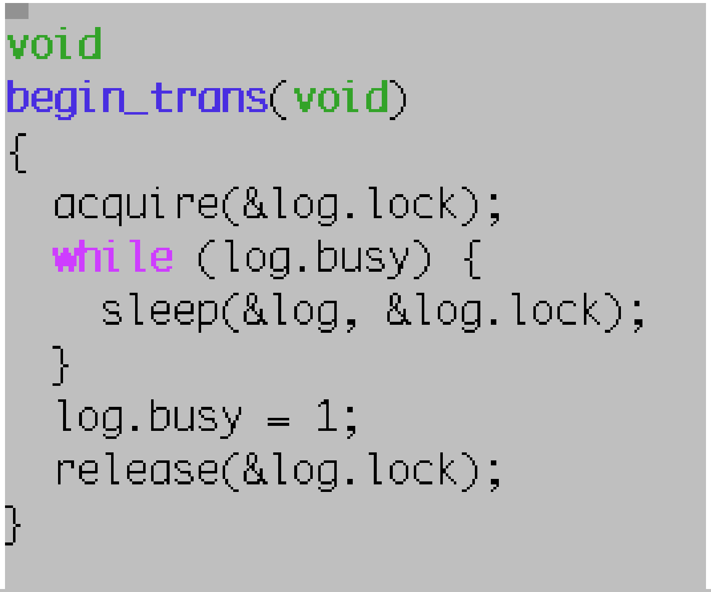
- record sector #
- append buffer content to log
- leave modified block in buffer cache (but do not write)

commit_trans():

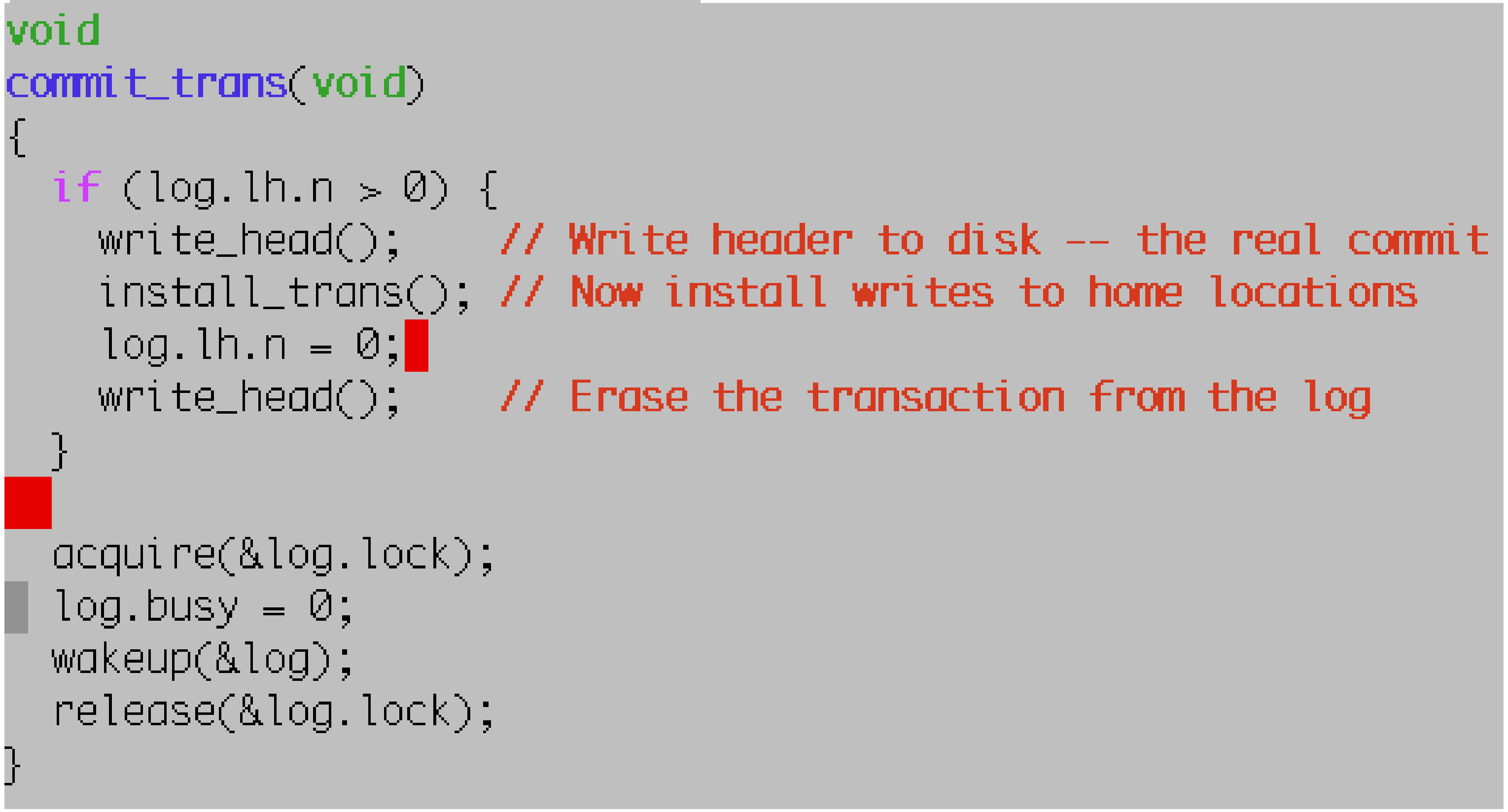
- record "done" and sector #s in log
- do the writes
- erase "done" from log

recovery:

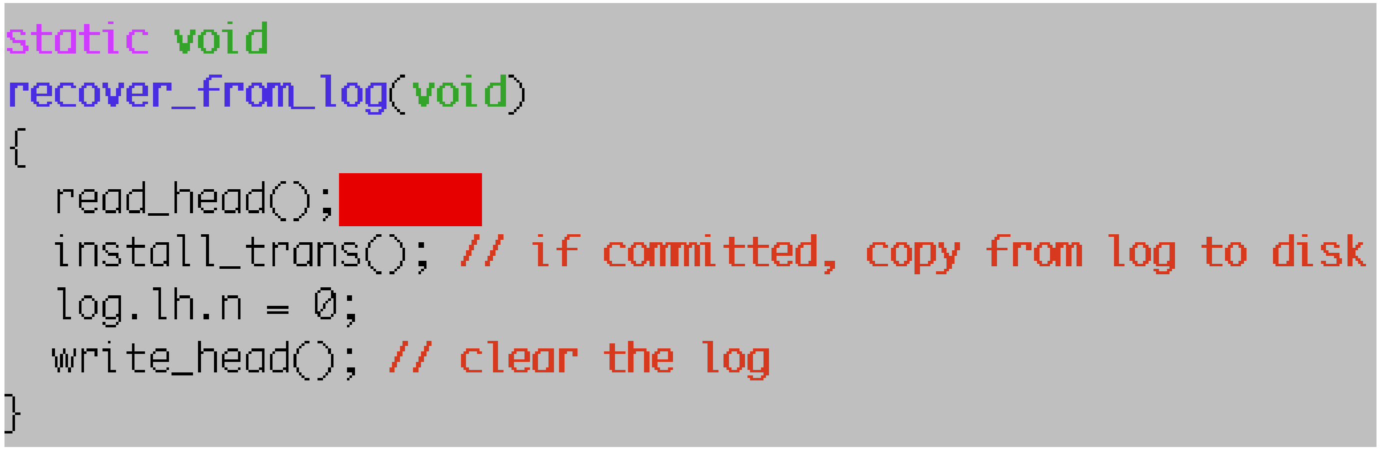
- if log says "done": copy blocks from log to real locations on disk



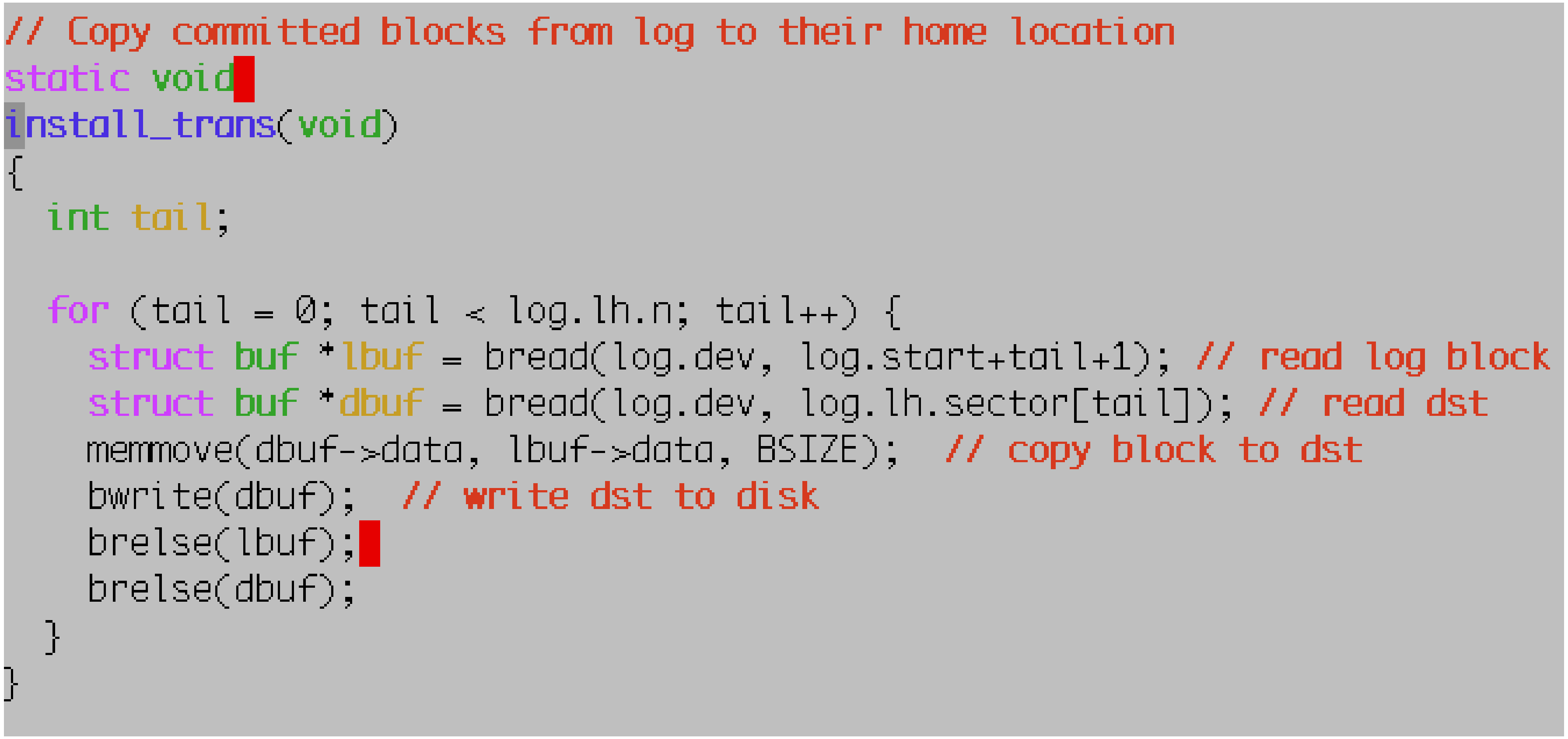
```
void
begin_trans(void)
{
  acquire(&log.lock);
  while (log.busy) {
    sleep(&log, &log.lock);
  }
  log.busy = 1;
  release(&log.lock);
}
```



```
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;
    write_head();    // Erase the transaction from the log
  }

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



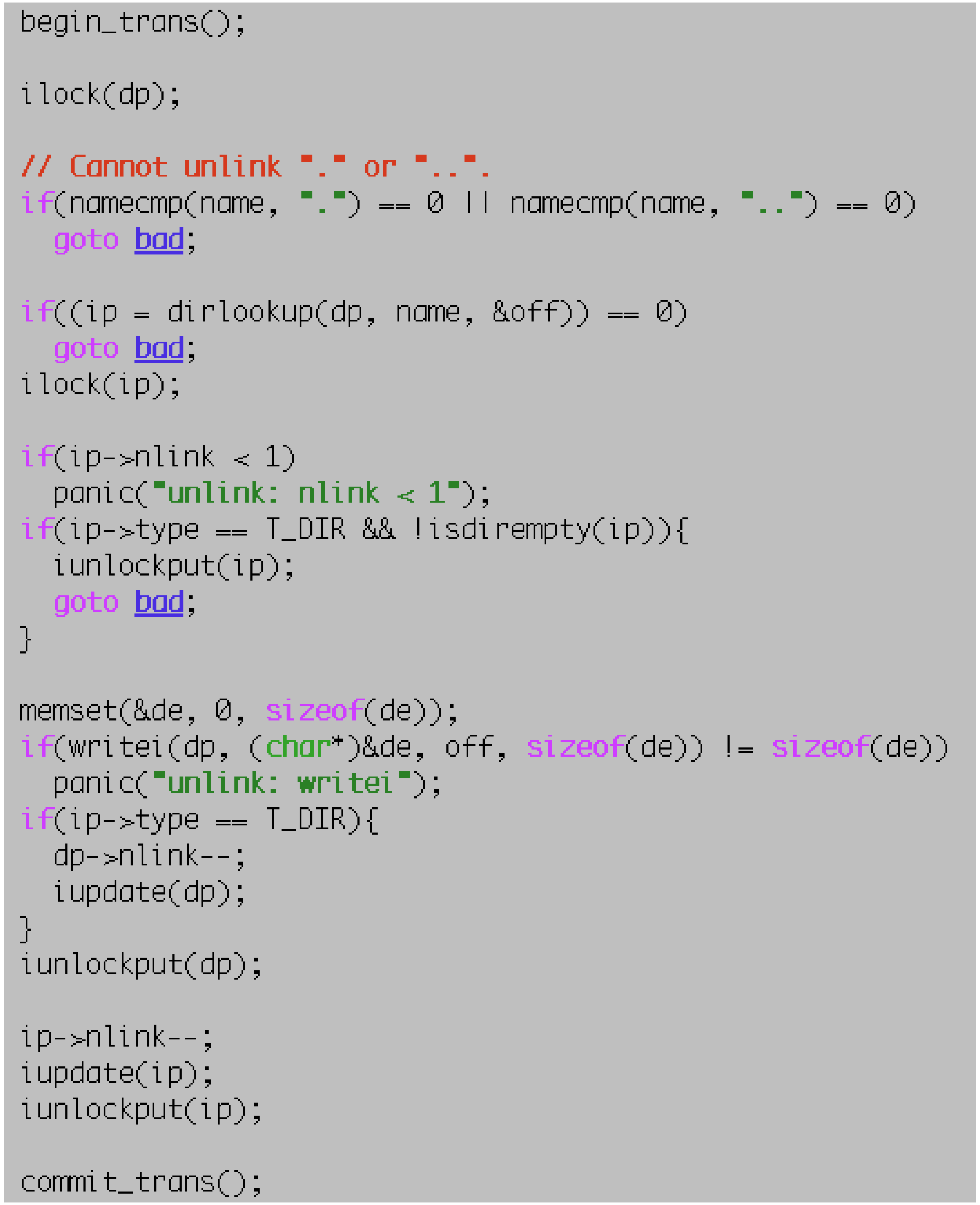
```
static void
recover_from_log(void)
{
  read_head();
  install_trans(); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
```



```
// 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);
    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

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"

ext3 adds a log to ext2, a previous xv6-like log-less file system

has many modes, start with "journaled data"

- log contains both metadata and file content blocks

Ext3 Structures

in-memory write-back block cache

in-memory list of blocks to be logged, per-transaction

on-disk FS

on-disk circular log file

What's in the ext3 log?

superblock: starting offset and starting seq #

descriptor blocks: magic, seq, block #s

data blocks (as described by descriptor)

commit blocks: magic, seq

How does ext3 get good perf.?

batches many syscalls per commit

defers copying cache block to log until it commits log to disk

hopes multiple sycalls modified same block

- thus many syscalls but only one copy of block in log
- "write absorbtion"

Syscall process

h = start()

get(h, block #)

- warn logging system we'll modify cached block
    - added to list of blocks to be logged
- prevent writing block to disk until after xaction commits

modify the blocks in the cache

stop(h)

guarantee: all or none

stop() does *not* cause a commit

Ext3 transaction

[circle set of cache blocks in this transaction]

while "open", adds new syscall handles, and remembers their block #s

only one open transaction at a time

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

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)

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 uncommited 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 xaction -- 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 xaction to finish
  - before letting any in next start
  - so that ops in old xaction don't read modifications of next xaction

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
  - modern disk interfaces can avoid wasted rotation

What if a crash?

crash may interrupt writing last xaction to log on disk

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

may have written some block cache to disk (home)

- but only for fully committed xactions, not partial last one

How does recovery work?

1. find the start and end of the log
    - log "superblock" at start of log file
    - log superblock has start offset and seq# of first transaction
    - scan until bad record or not the expected seq #
    - go back to last commit record
    - crash during commit -> last transaction ignored during recovery
2. replay all blocks through last completed xaction, 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

what if block in T1 has been dirtied in cache by T2?

- can't write that block to FS on disk
- note ext3 only does copy-on-write while T1 is commiting
  - after T1 commit, T2 dirties only block copy in cache
- so can't free T1 until T2 commits, so block is in log
  - T2's logged block contains T1's changes

Question

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

- can we free T1 to free up log space?
- maybe not, due to previous issue, T2 maybe dirtied a block in T1

- deadlock!

Solution: Reservations

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

block the sycall from starting until enough free space

may need to commit open transaction, then free older transaction

- OK since reservations mean all started sys calls can complete + commit

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 sys call?
  - would log many more blocks, no absorption
  - 10 ms per syscall, rather than 0 ms
  - (“Rethink the Sync” [OSDI’06] addresses this problem)

Issues with ext3

disks usually have write caches and re-order writes, for performance

- sometimes hard to turn off (the disk lies)
- often with re-ordering enabled for speed, out of ignorance

bad news if disk writes commit block before preceding stuff

- then recovery replays "descriptors" with random block #s!
- and writes them with random content!

how to solve this problem?

- using checksum

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

Xv6 vs. ext3

does ext3 fix the xv6 log performance problems?

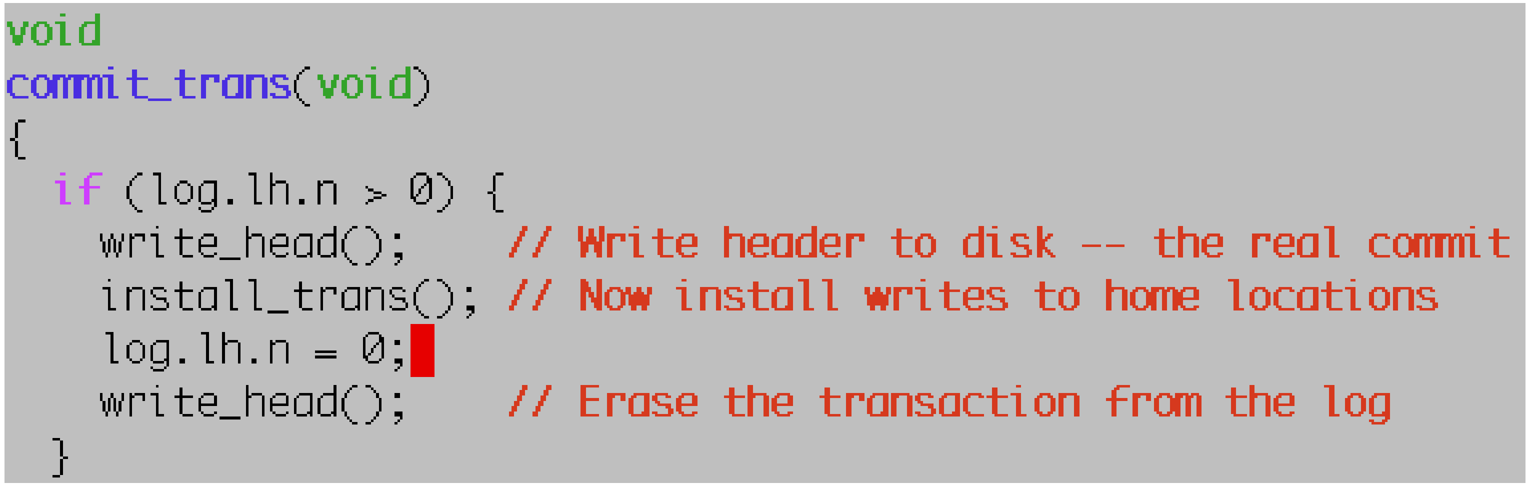
- only one transaction at a time -- yes
- synchronous write to on-disk log -- yes, but 5-second window
- tiny update -> whole block write -- yes (indirectly)
- synchronous writes to home locations after commit – yes

ext3 very successful

- but: no checksum -- ext4
- but: not efficient for applications that use fsync()

Questions

- In commit_trans, there are three steps
  - Write_head: log “done” to disk
  - Install_trans: checkpoiting metadata to home location
- Can these two operations get reordered or mixed together?
- How can we make then reordered?



```
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;
    write_head();     // Erase the transaction from the log
  }
```

T2 starts while T1 is committing to log on disk

- what if syscall in T2 wants to write block in prev xaction?
- can't be allowed to write buffer that T1 is writing to disk
  - then new syscall's write would be part of T1
  - crash after T1 commit, before T2, would expose update
- T2 gets a separate copy of the block to modify
  - T1 holds onto old copy to write to log
- are there now *two* versions of the block in the buffer cache?
  - no, only the new one is in the buffer cache, the old one isn't
- does old copy need to be written to FS on disk?
  - no: T2 will write it

Correctness w/ ordered mode

A. rmdir, re-use block for file, ordered write of file, crash before rmdir or write committed

- now scribbled over the directory block

fix: defer free of block until freeing operation forced to log on disk

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 ..

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

Final Tidbit

- open a file, then unlink it
  unlink commits
  file is open, so unlink removes dir ent but doesn't free blocks crash
  nothing interesting in log to replay
  inode and blocks not on free list, also not reachably by any name
- will never be freed! oops

- solution: add inode to linked list starting from FS superblock commit that along with remove of dir ent
  - recovery looks at that list, completes deletions

Scalable Locking

Rong Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/haibo_chen

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

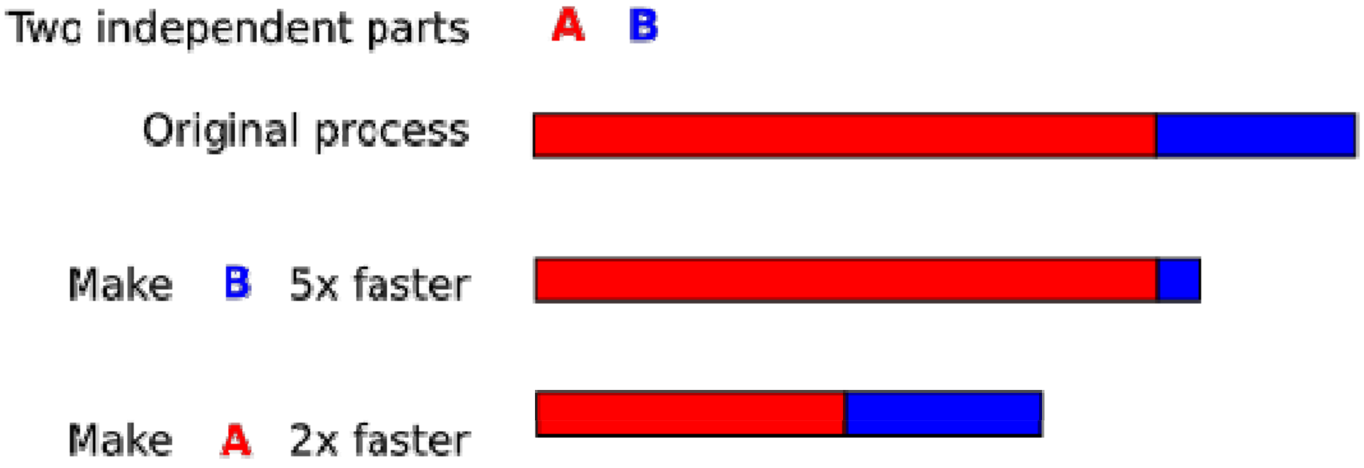
- If multiple writers
- Using lock to ensure single writer principle

Locking

- ACQUIRE & RELEASE
- Need atomicity
    - RSM: read-set-memory

Amdahl's Law

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



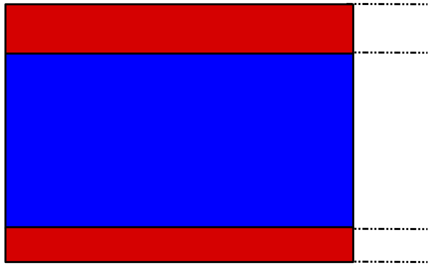
Two independent parts A B

Original process

Make B 5x faster

Make A 2x faster

Critical-section efficiency



Lock Acquisition (T_a)

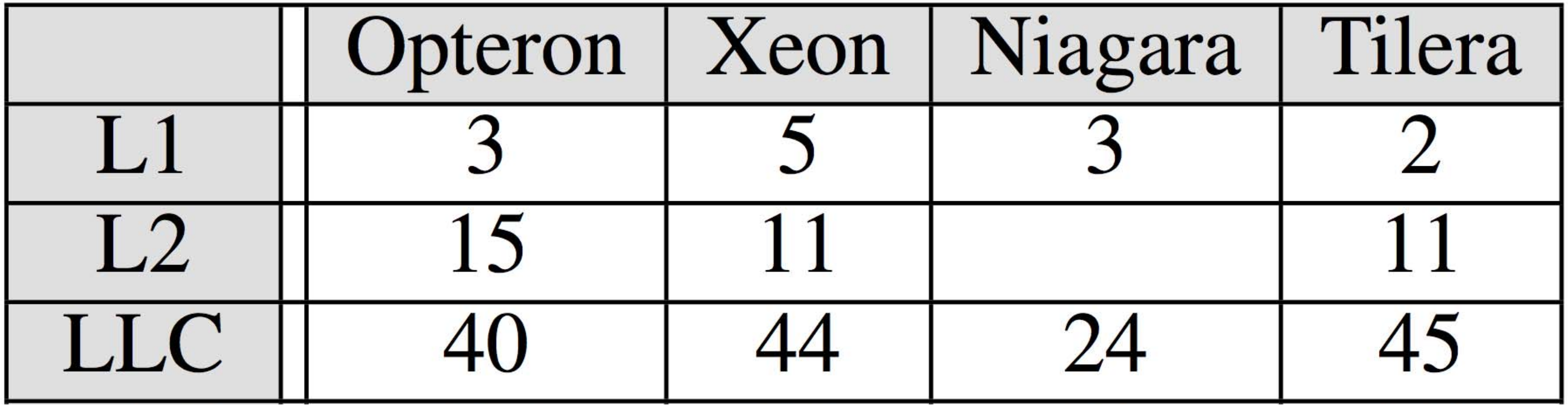
Critical Section (T_c)

Lock Release (T_r)

$$\text{Critical-section efficiency} = \frac{T_c}{T_c + T_a + T_r}$$

Ignoring lock contention and cache conflicts in the critical section

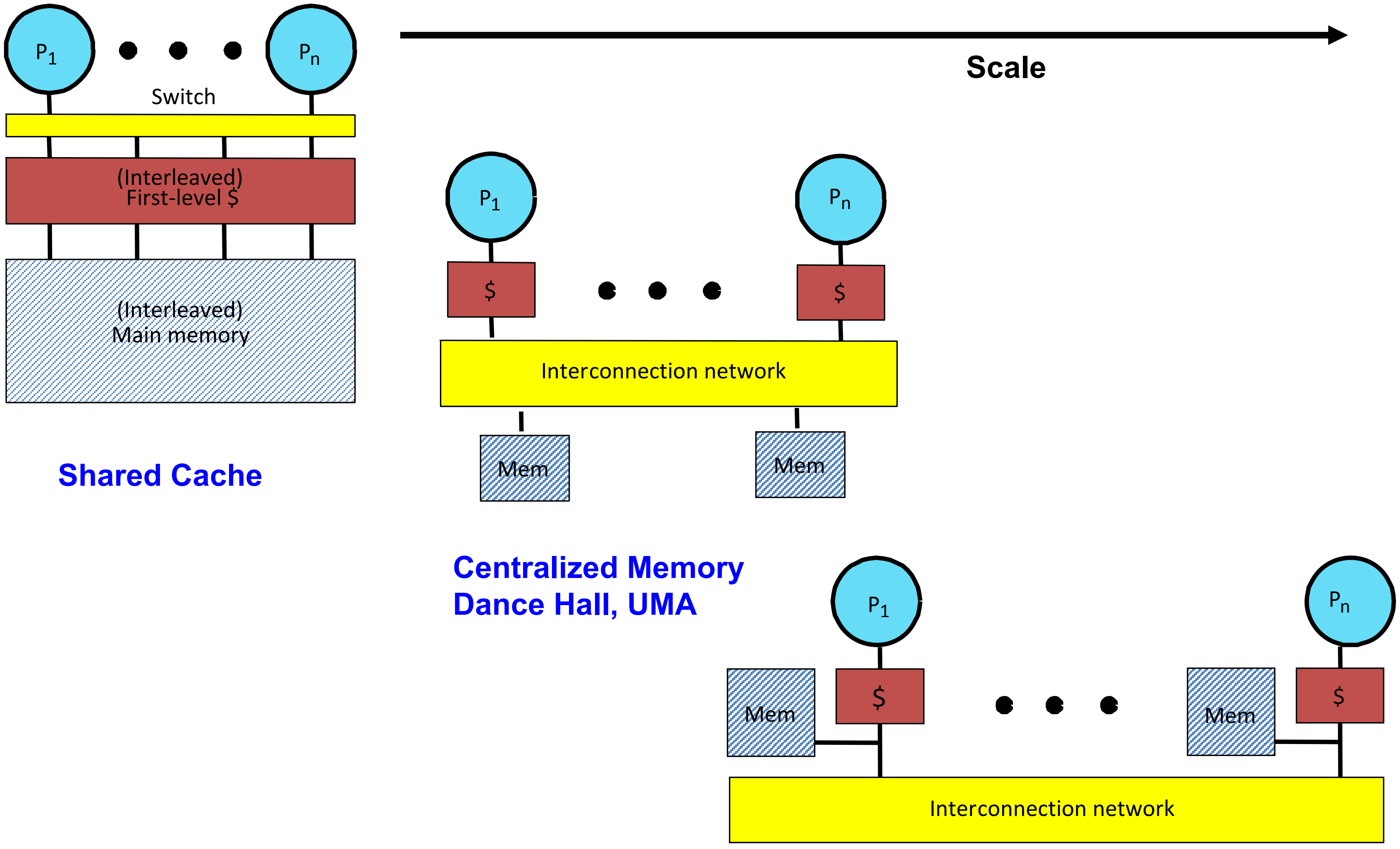
Some numbers you may want to know



	Opteron	Xeon	Niagara	Tilera
L1	3	5	3	2
L2	15	11		11
LLC	40	44	24	45

Source: *Everything you always wanted to know about synchronization*, but were afraid to ask.

Shared Memory Multiprocessors

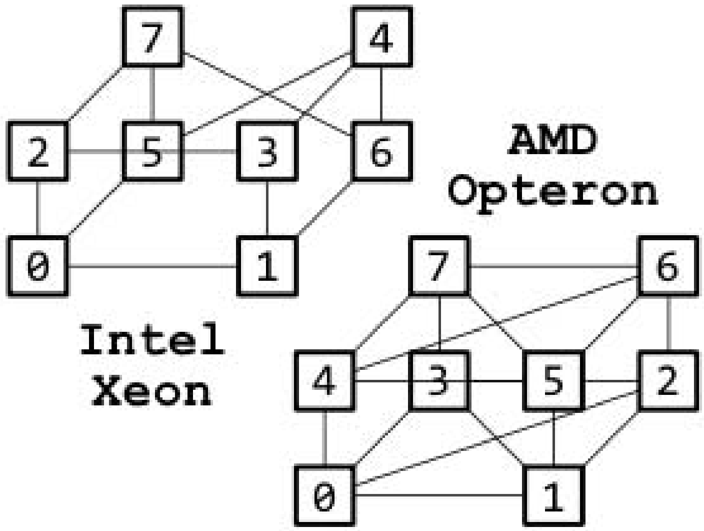


Shared Cache

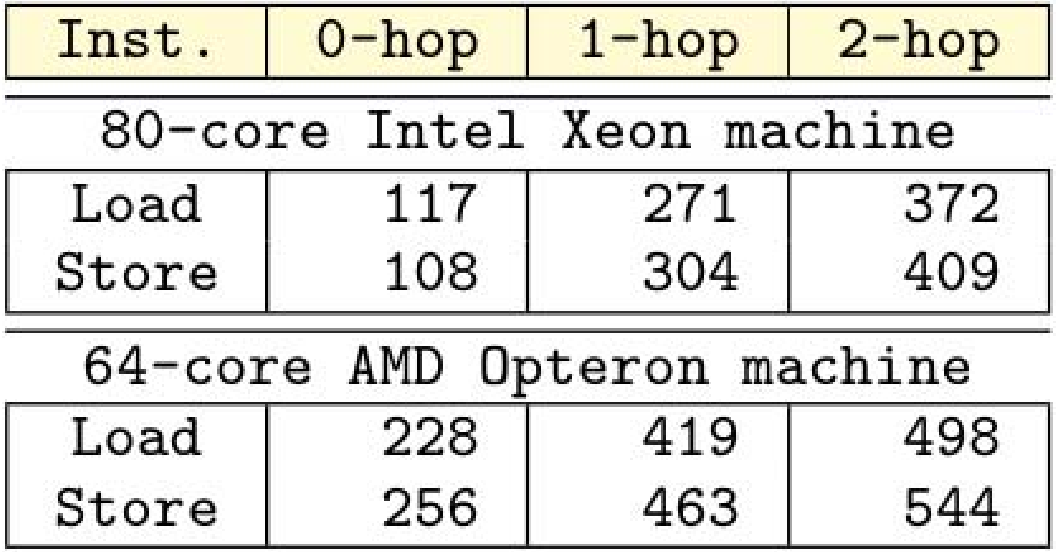
Centralized Memory
Dance Hall, UMA

Distributed Memory (NUMA)

More numbers you may want to know



(a) System topology



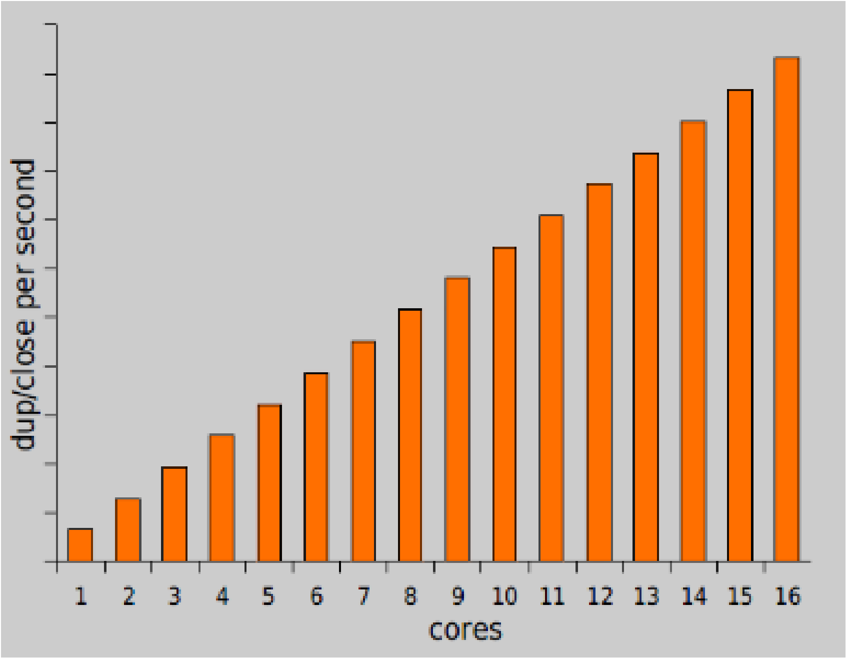
Inst.	0-hop	1-hop	2-hop
80-core Intel Xeon machine			
Load	117	271	372
Store	108	304	409
64-core AMD Opteron machine			
Load	228	419	498
Store	256	463	544

(b) Latencies (cycles) on the distance

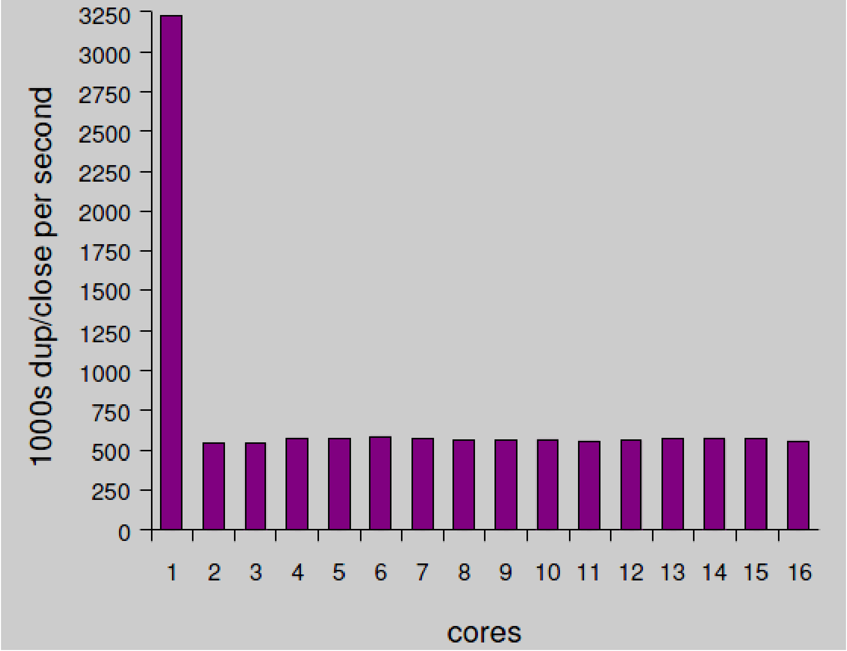
```
fd_alloc(void) {
   lock(fd_table);
   fd = get_free_fd();
   set_fd_used(fd);
   fix_smallest_fd()
   unlock(fd_table);
}
```

Motivating example: file descriptors

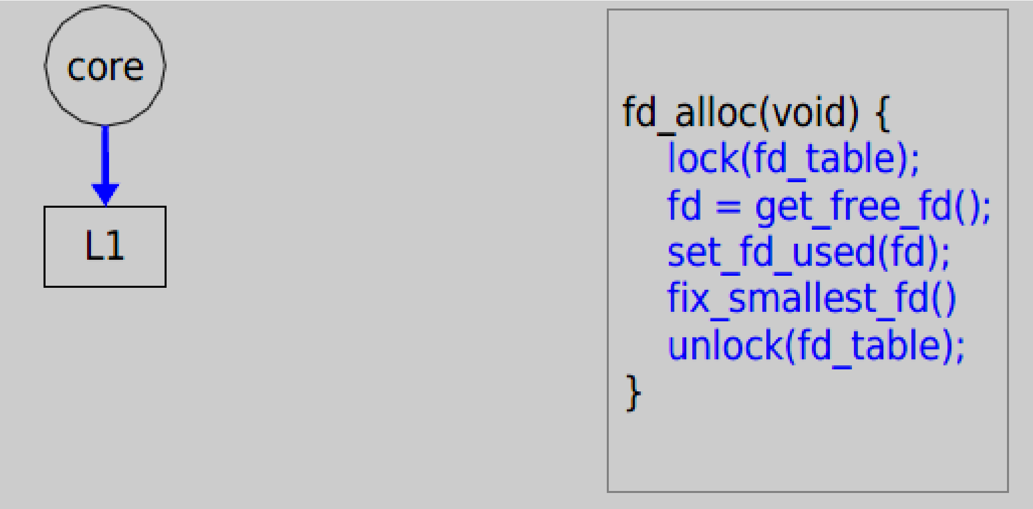
Ideal FD performance graph



Actual FD performance



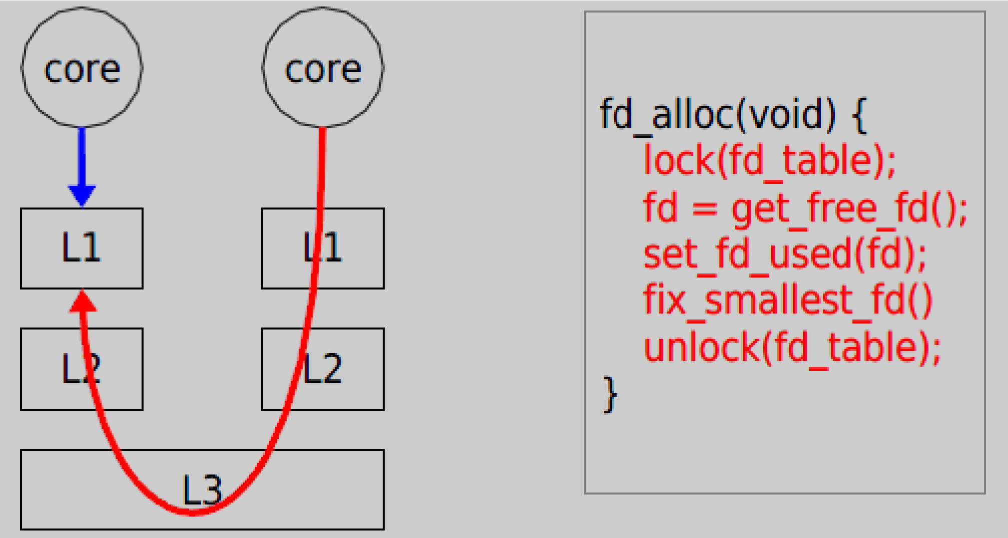
Why throughput drops?



```
fd_alloc(void) {
   lock(fd_table);
   fd = get_free_fd();
   set_fd_used(fd);
   fix_smallest_fd()
   unlock(fd_table);
}
```

Load fd_table data from L1 in 3 cycles.

Why throughput drops?



```
fd_alloc(void) {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd()
    unlock(fd_table);
}
```

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

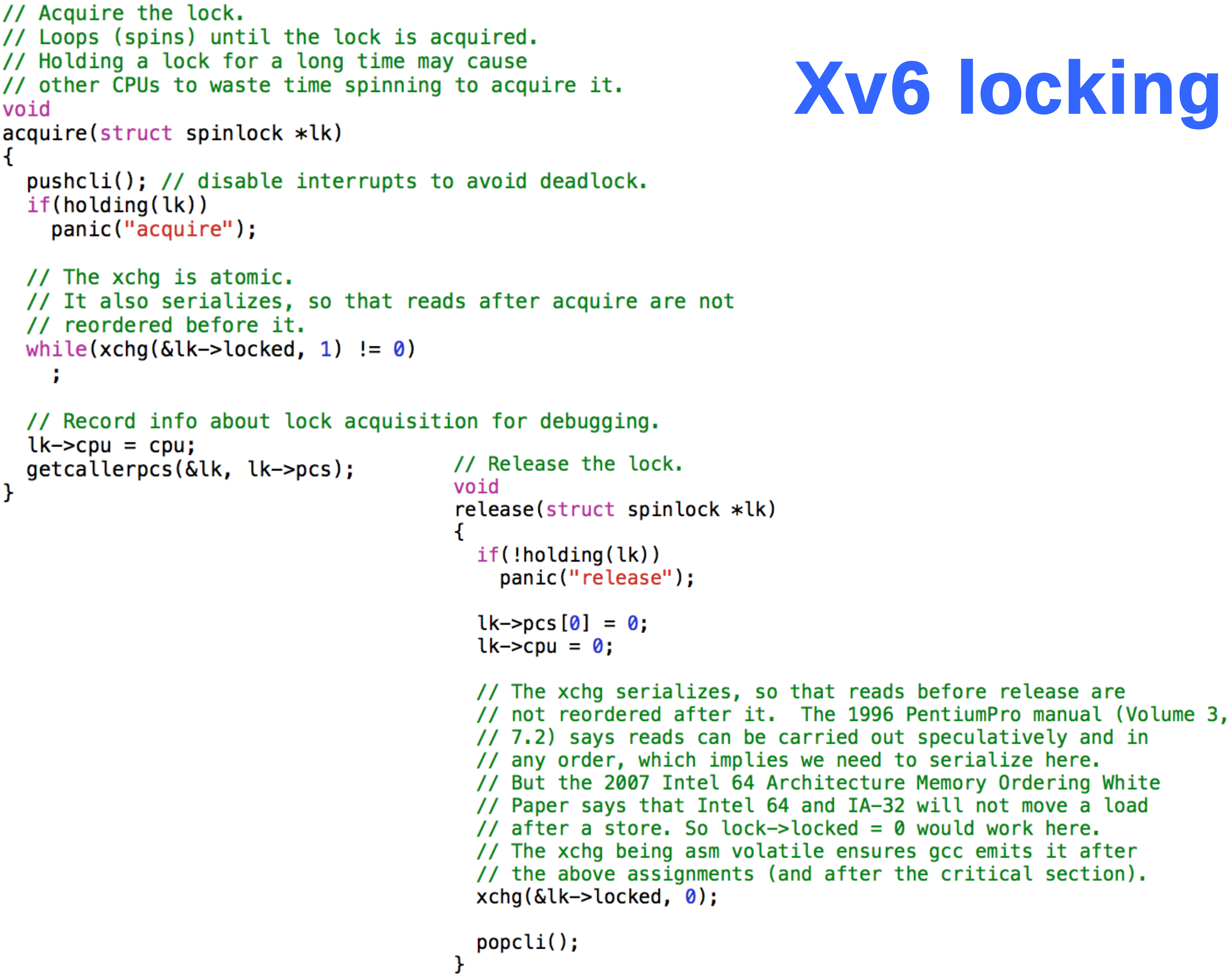
But they are dangerous

Why they are 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);
}
```

```
// Release the lock.
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;
    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++;
}
```

Currently serving which one

Ticket for the next comer

Add and get
original value

Questions

How do current_ticket and next_ticket work together to guarantee mutual exclusion?

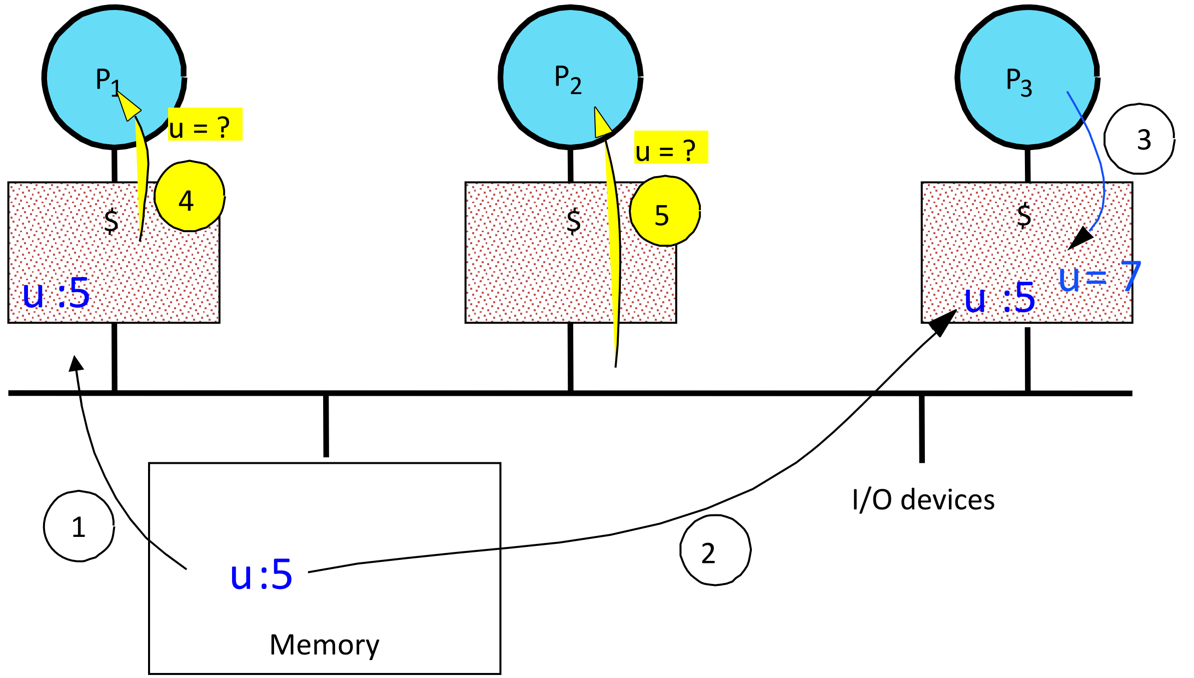
Background on cache coherence

On multi-core processors, each core has it's own private cache

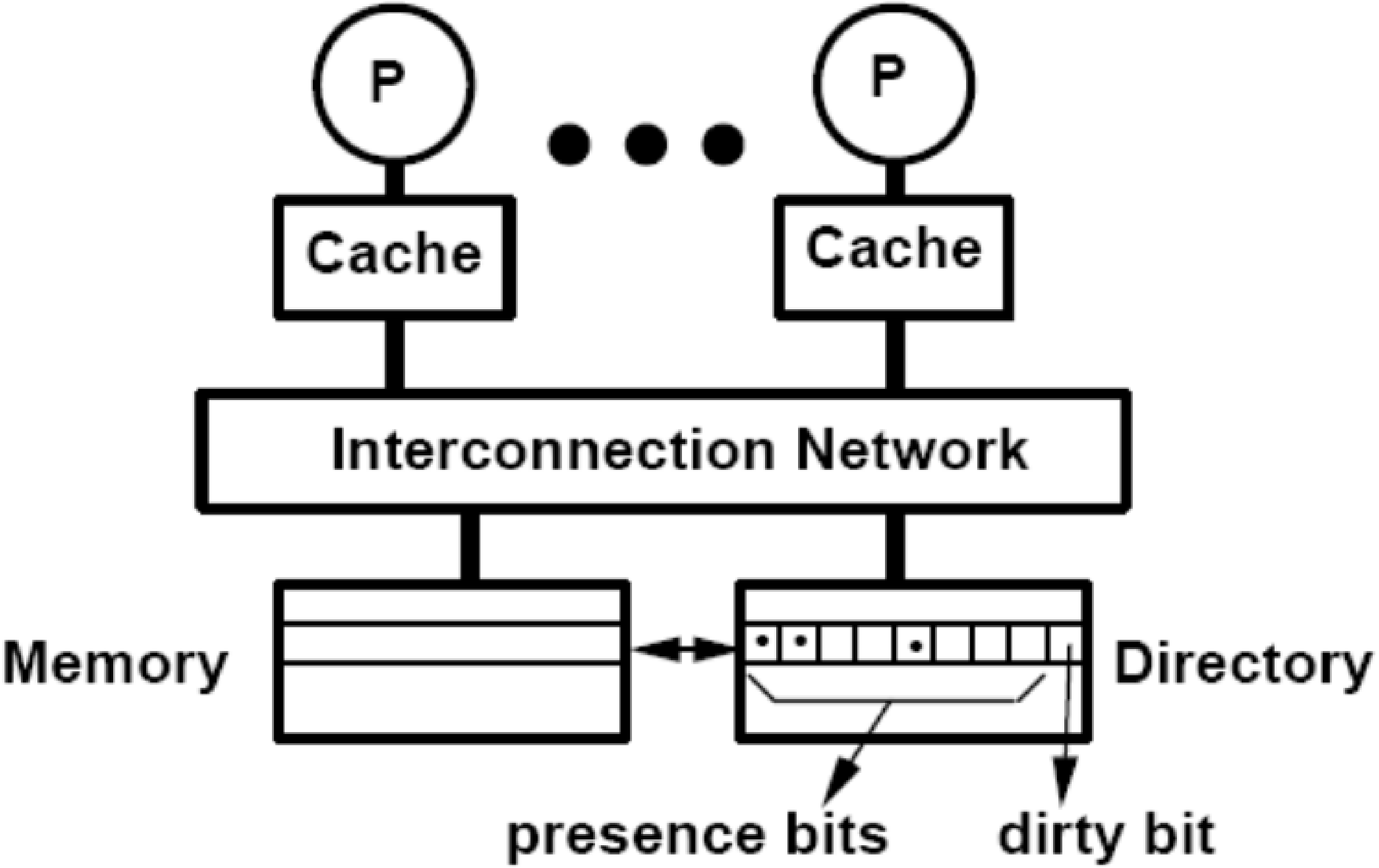
Need to keep the cache content in sync when some CPU modifies some memory

Accomplished by cache coherence protocol

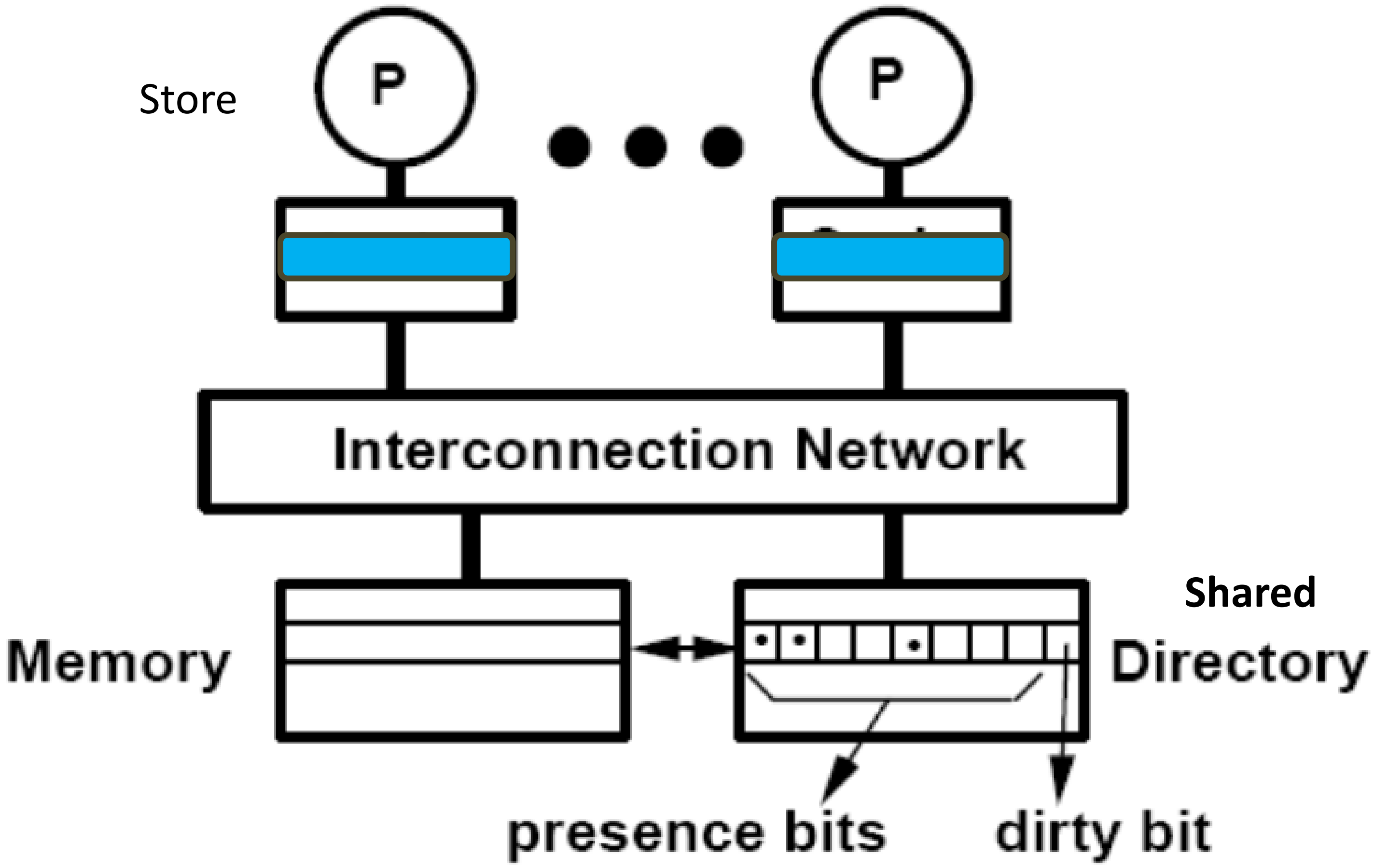
Example Cache Coherence Problem



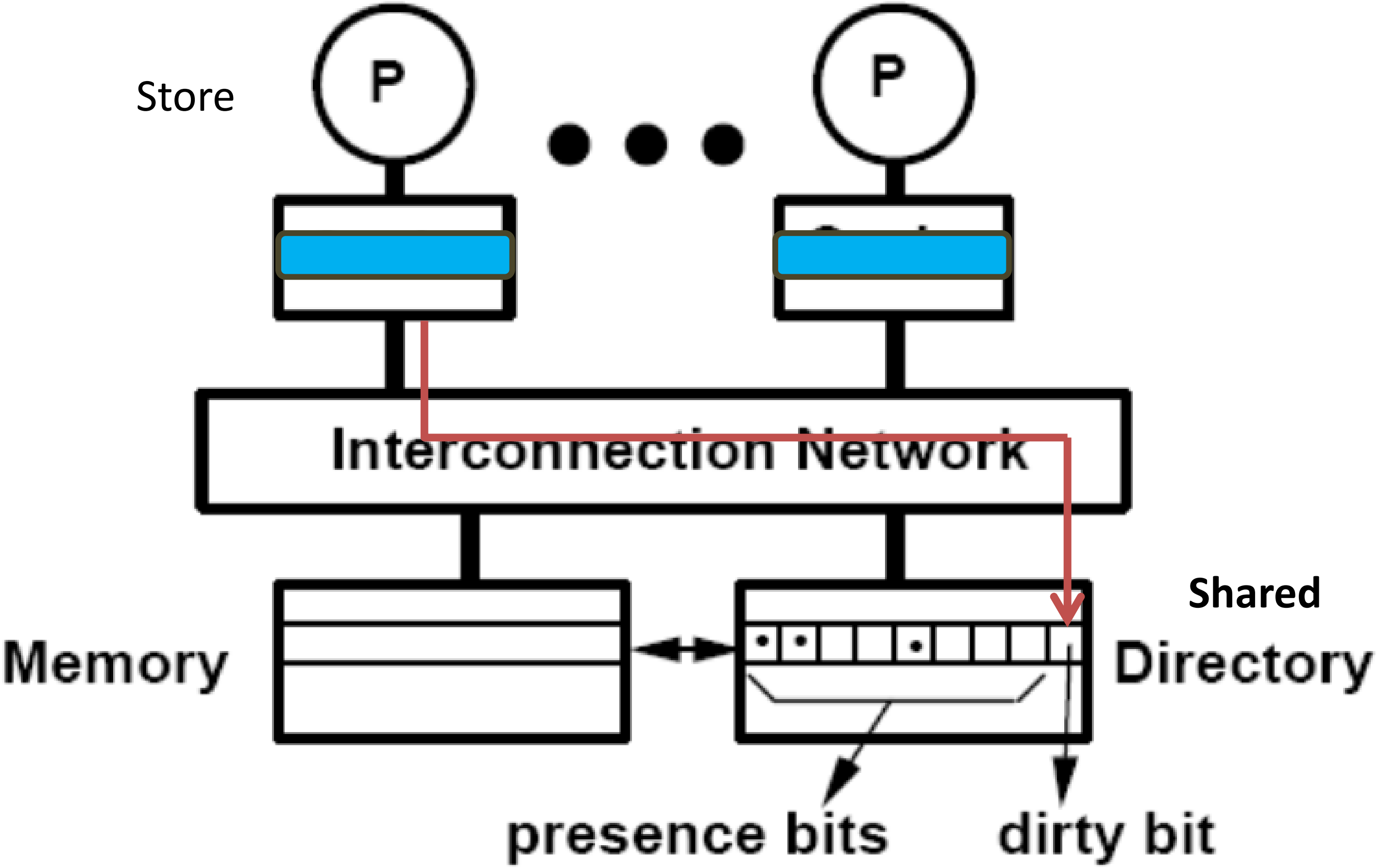
Directory-based cache coherence



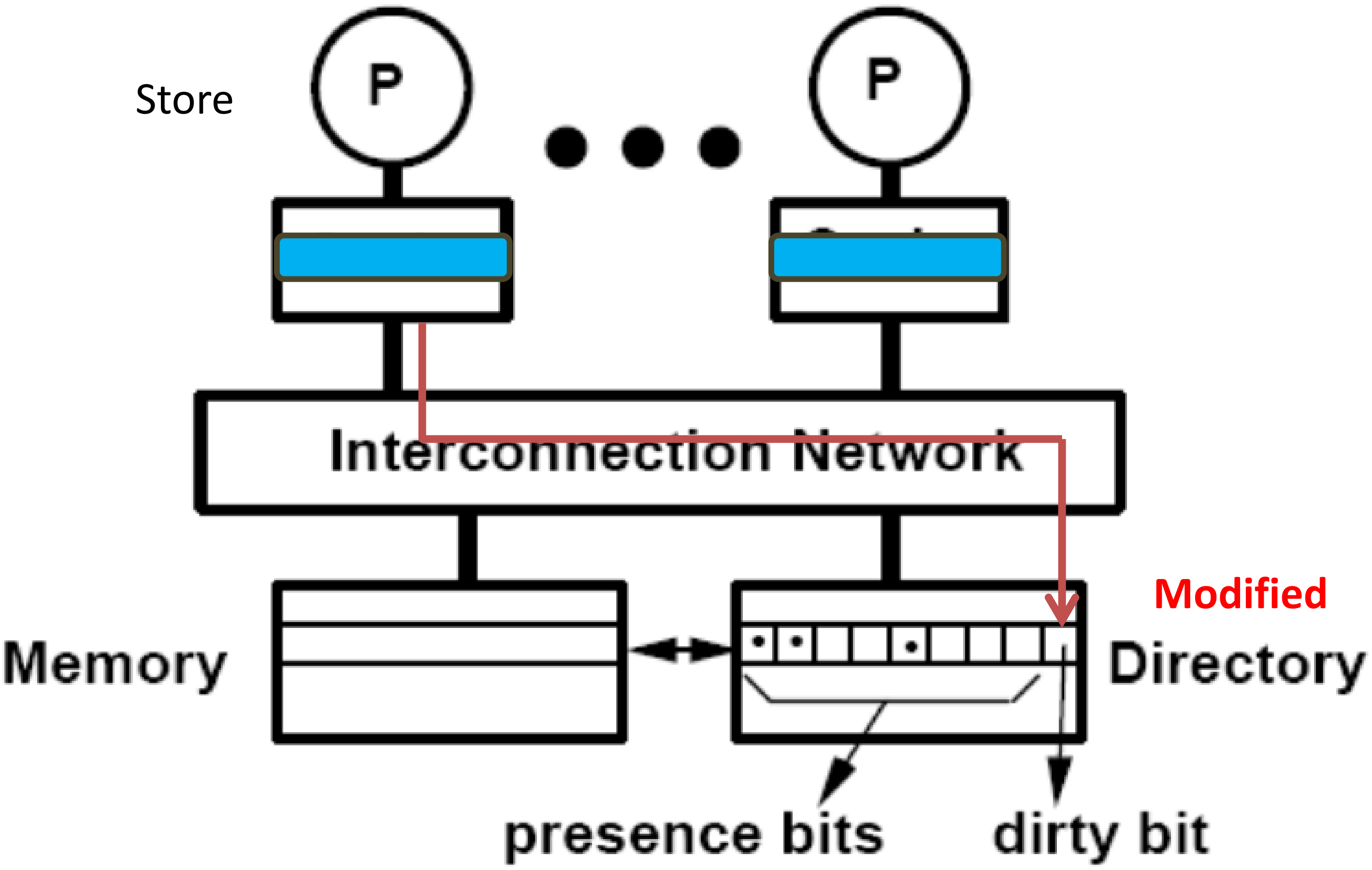
Directory-based cache coherence



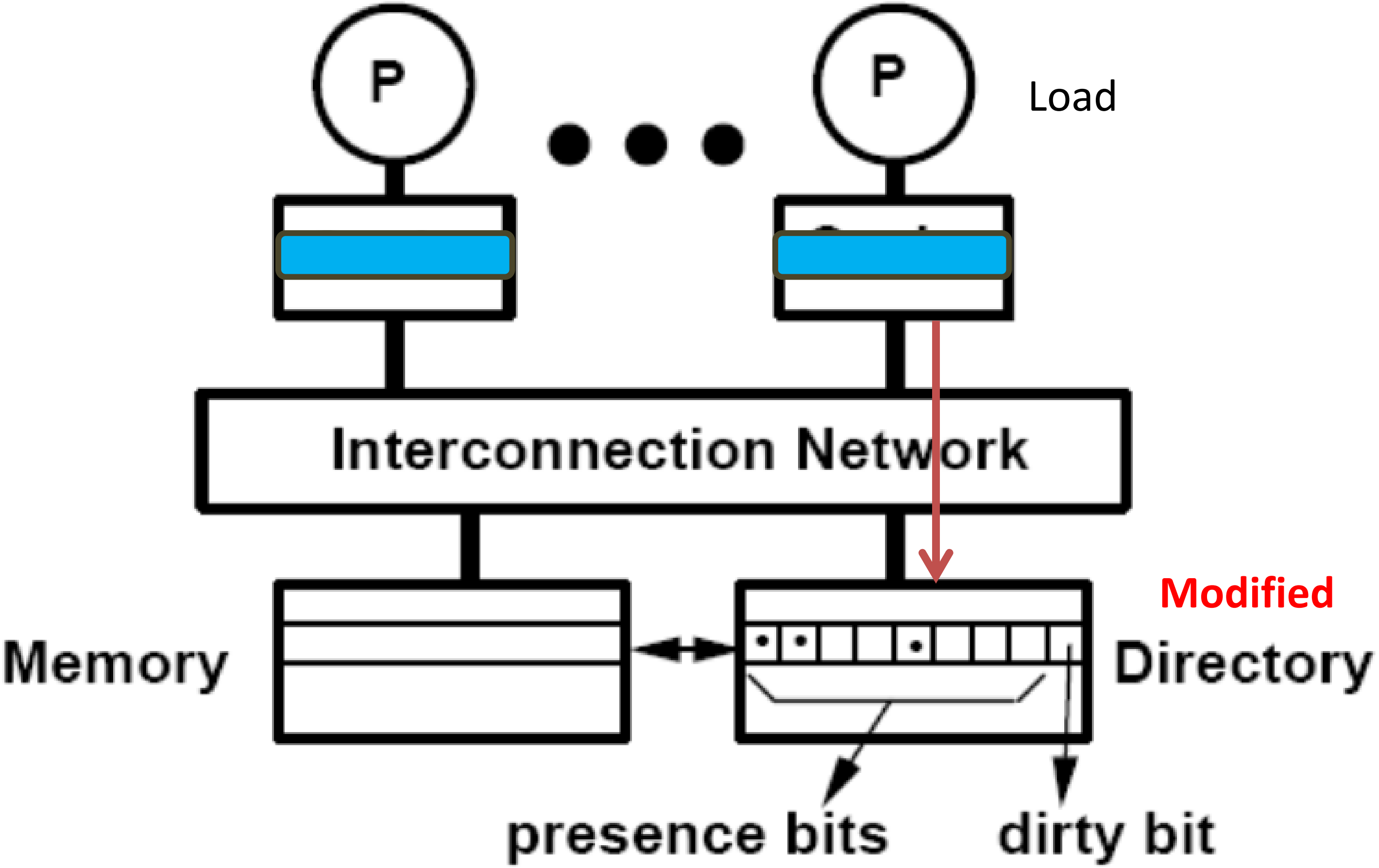
Directory-based cache coherence



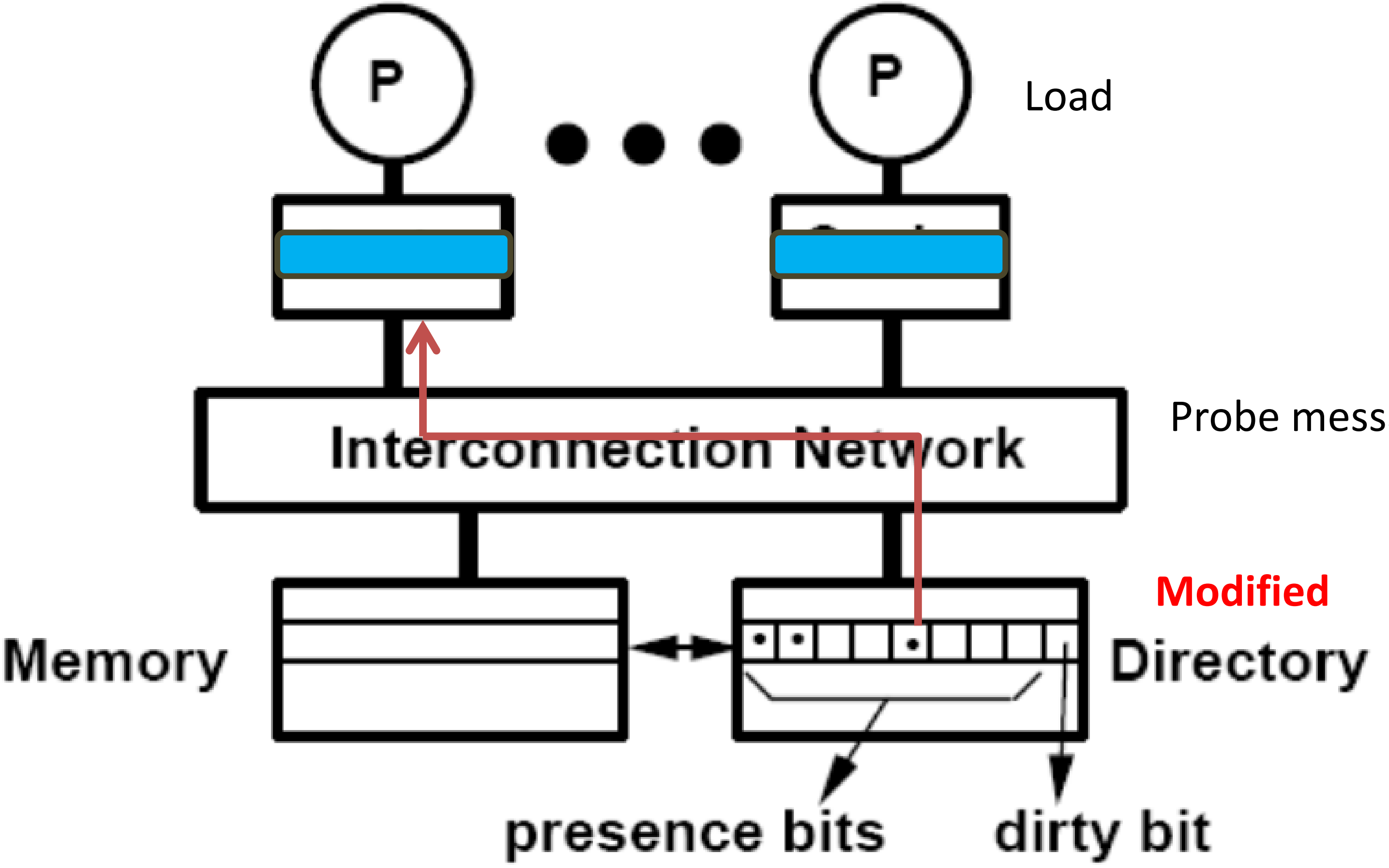
Directory-based cache coherence



Directory-based cache coherence



Directory-based cache coherence

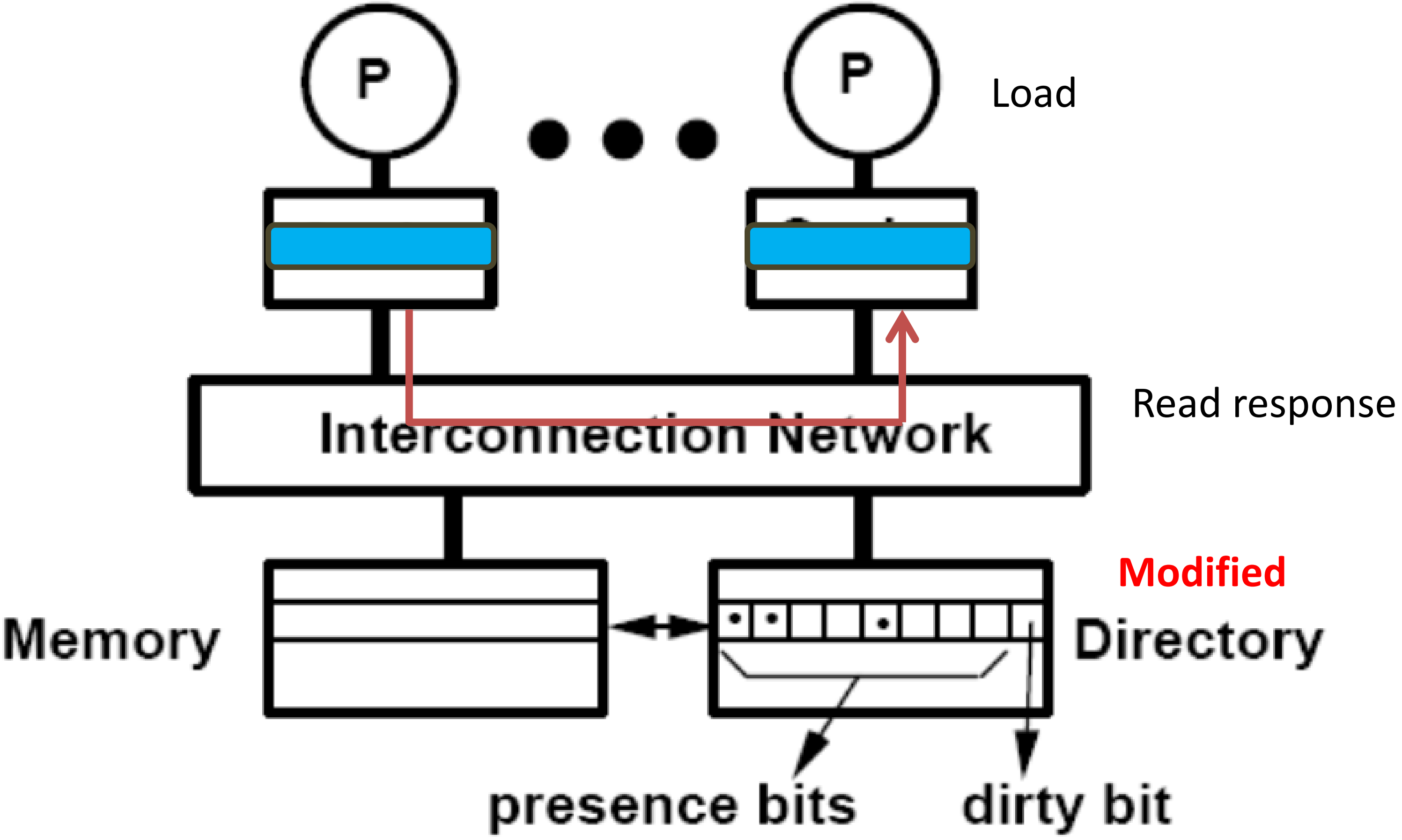


Load

Probe message

Modified

Directory-based cache coherence

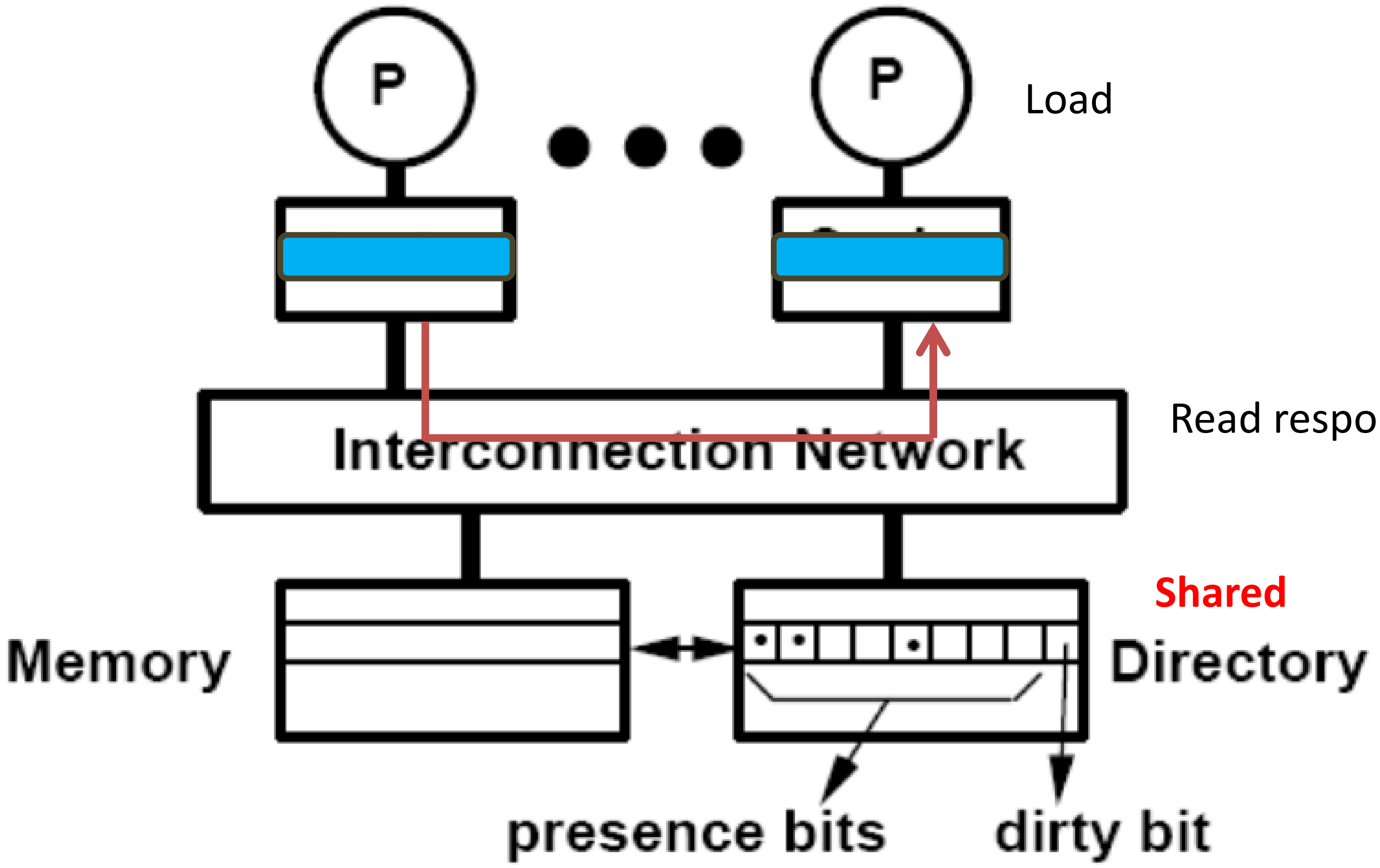


Load

Read response

Modified

Directory-based cache coherence



Load

Read response

Shared

A few notes 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

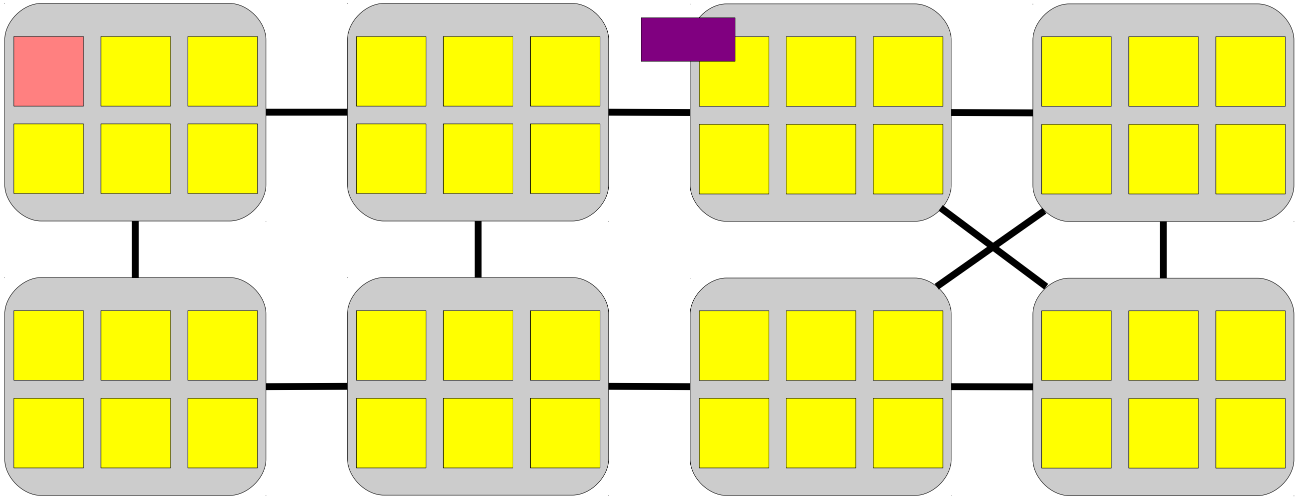
- They get value of the lock from the lock holder
- More readers means the next lock holder need to wait more time to get the lock

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;
}
```



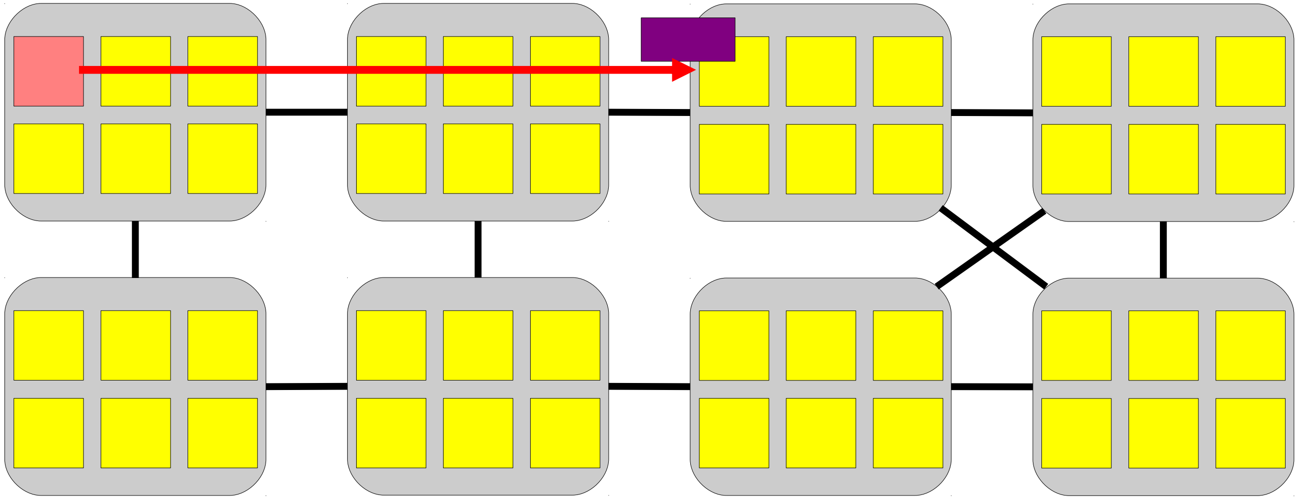
Allocate a ticket
read current ticket and spin

```
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



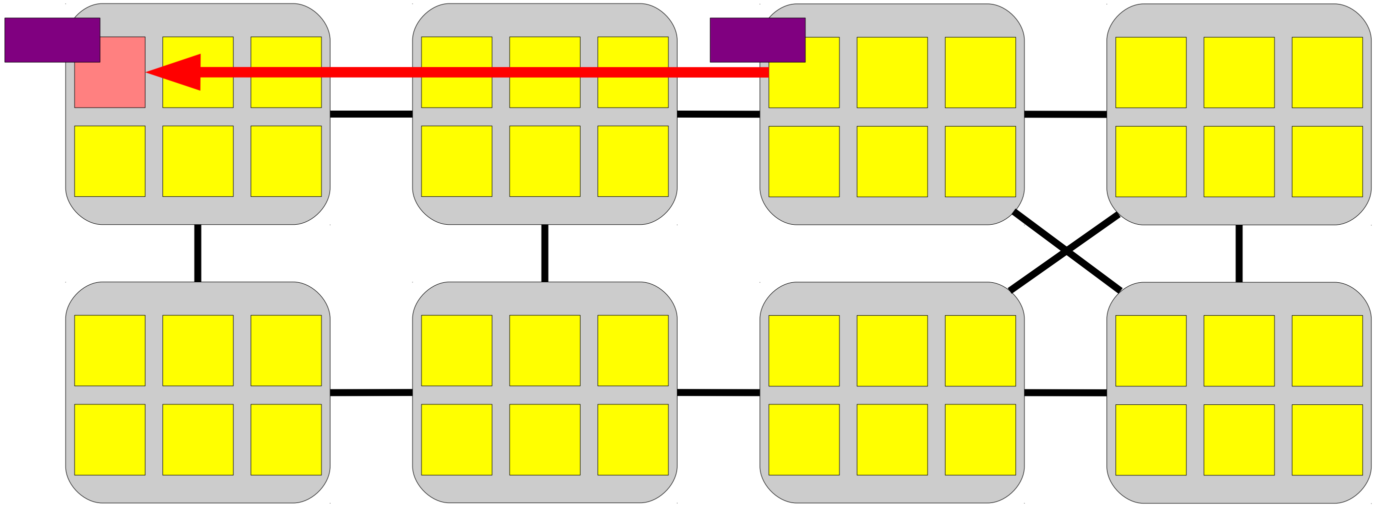
Allocate a ticket
read current ticket and spin

```
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



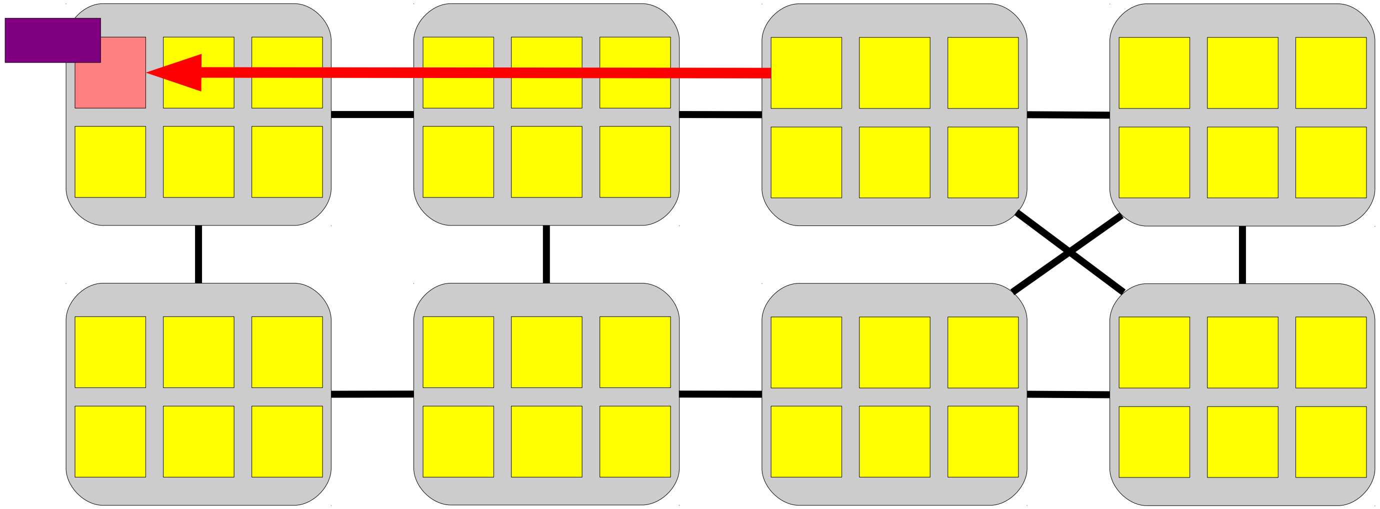
Allocate a ticket
read current ticket and spin

```
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

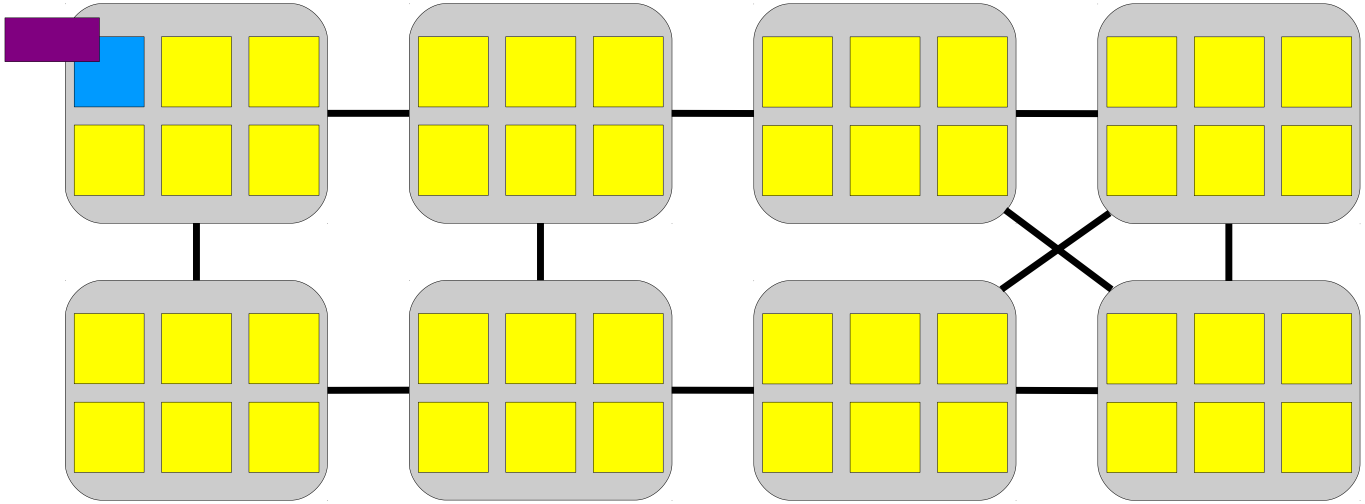


update the ticket value

```
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++;
}
```

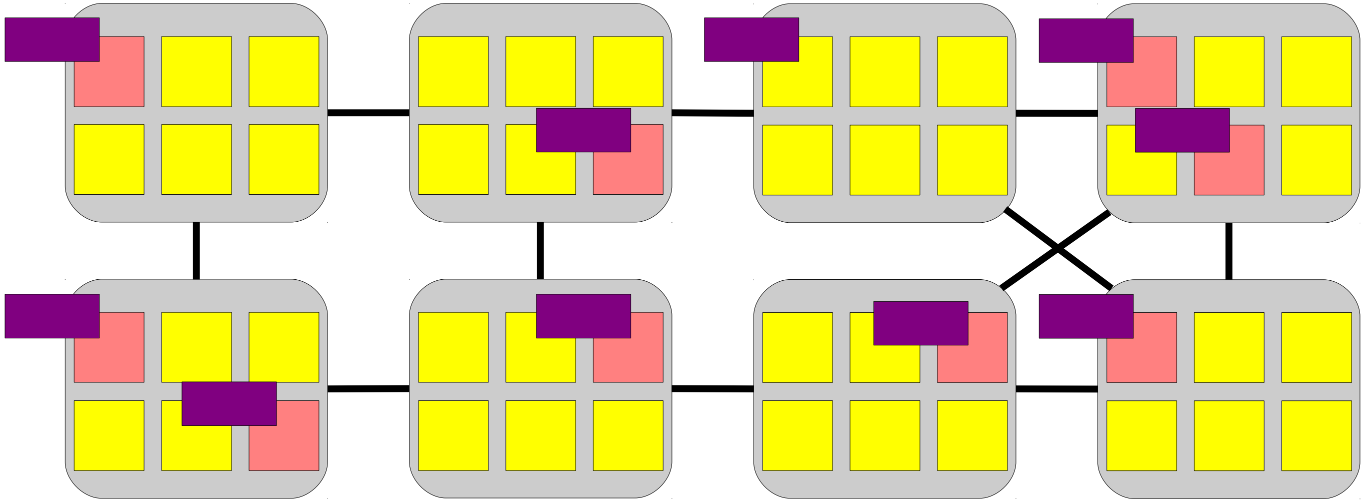
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```



```
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;
}
```

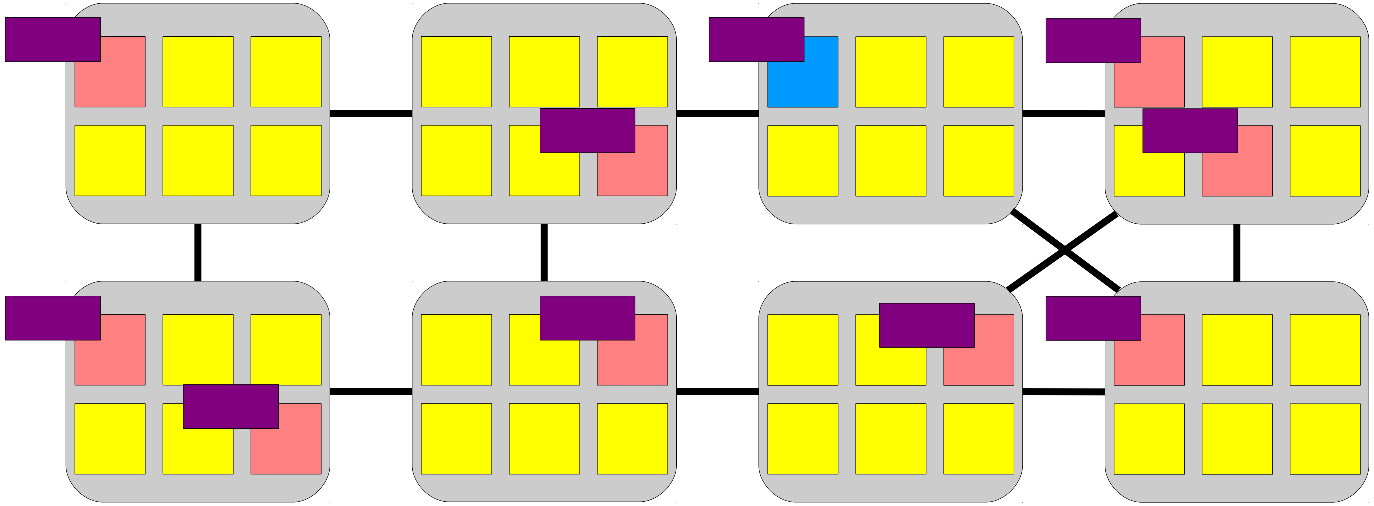


Lock shared by all core

```
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;
}
```



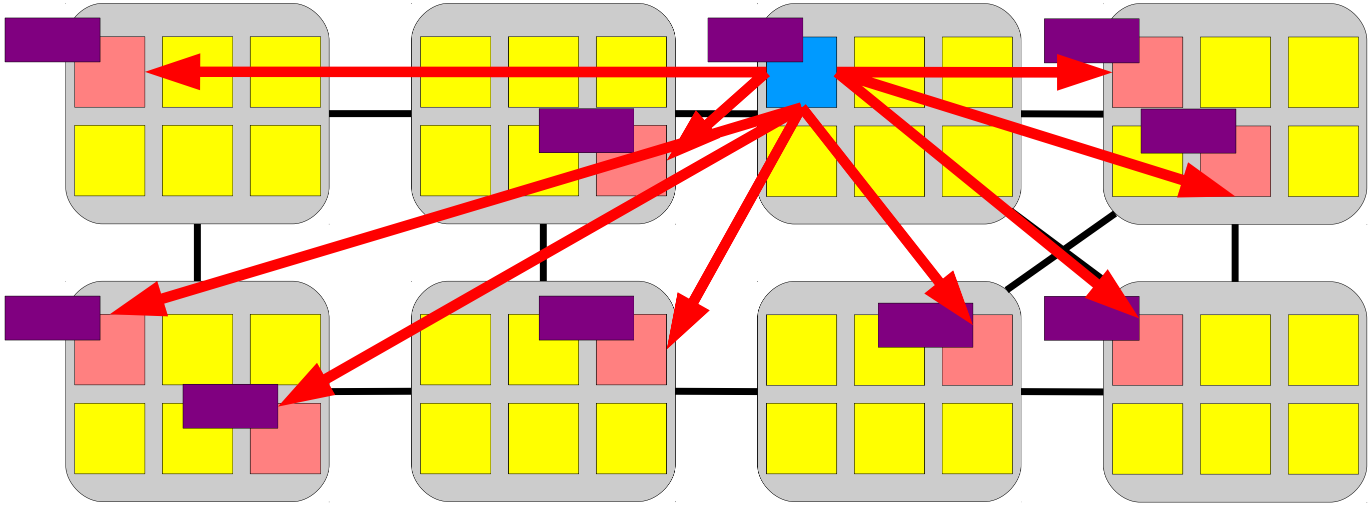
Lock holder update ticket

```
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



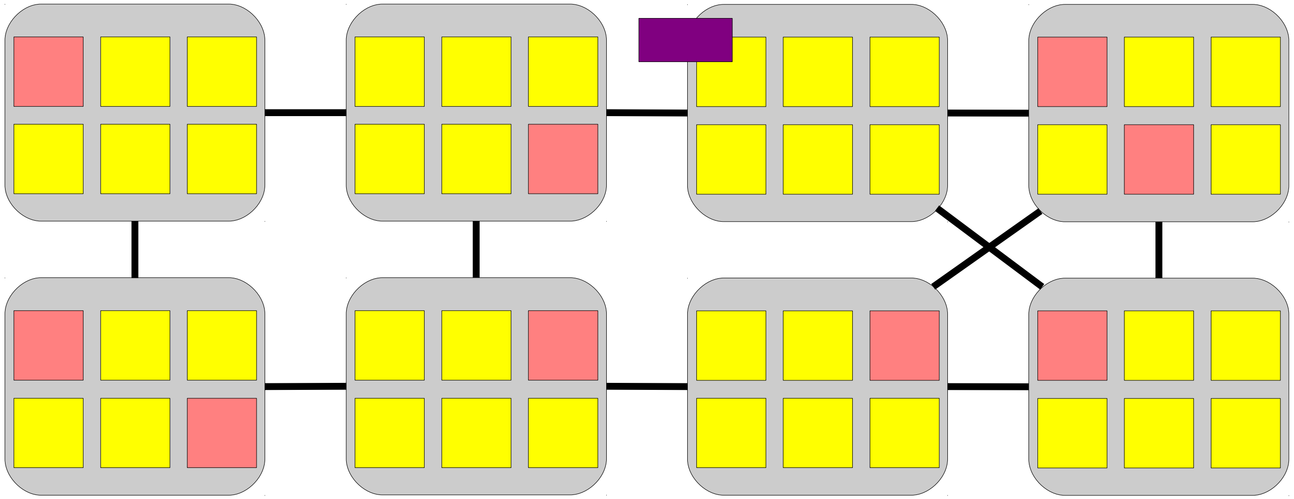
Lock holder update ticket

```
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

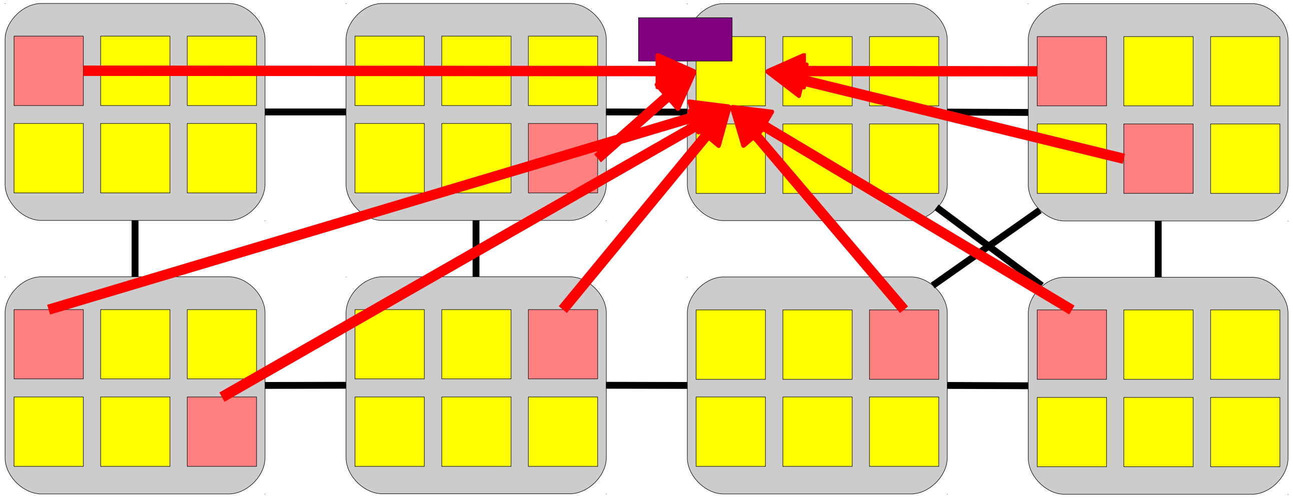


Lock holder update ticket

```
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;
}
```



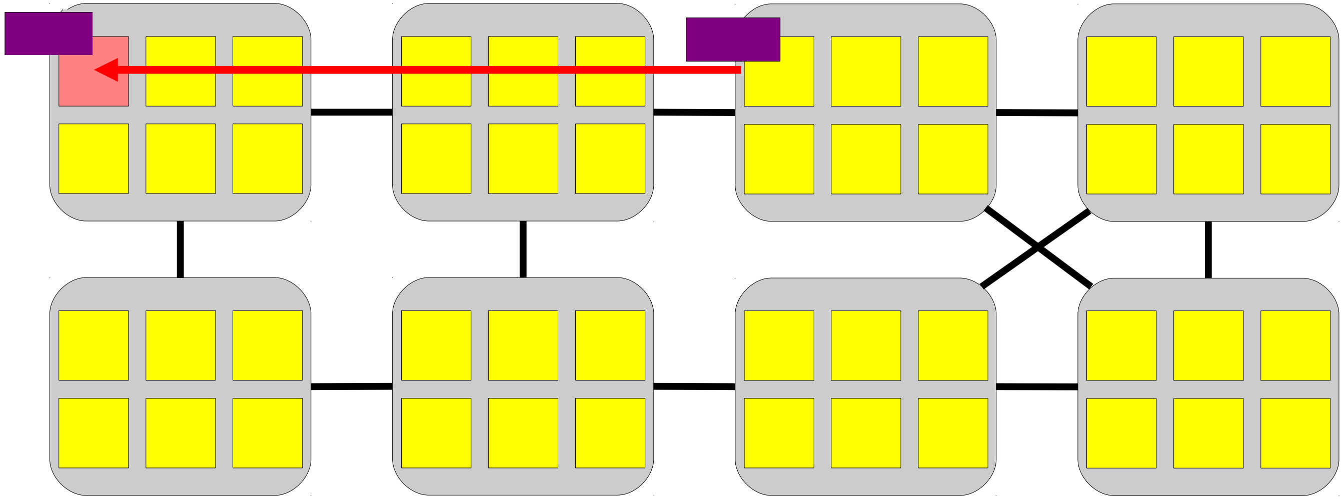
All waiters read the lock

```
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!



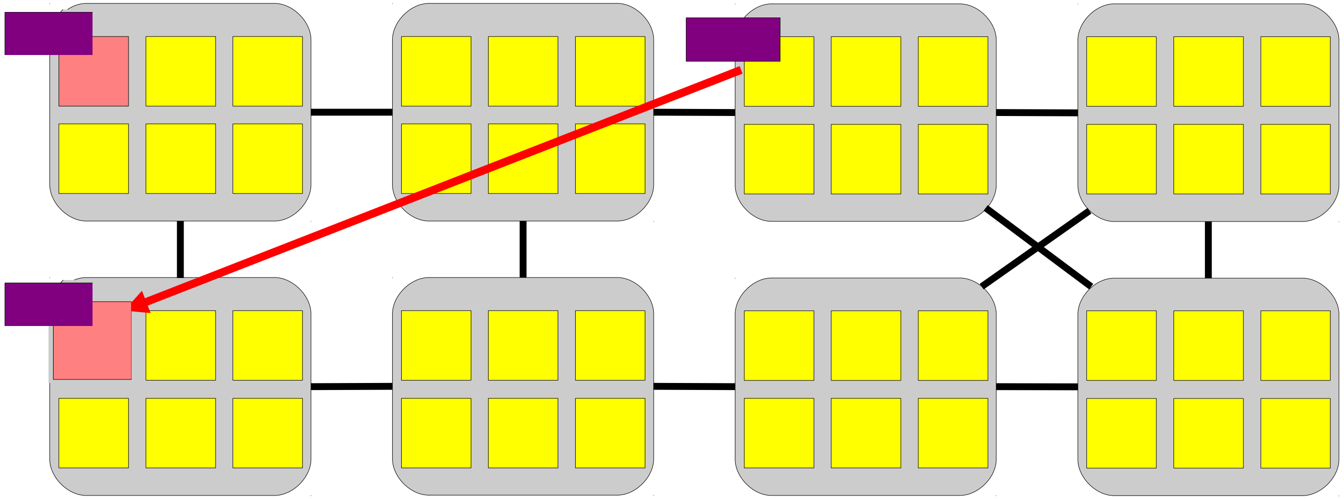
All waiters read the lock

```
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



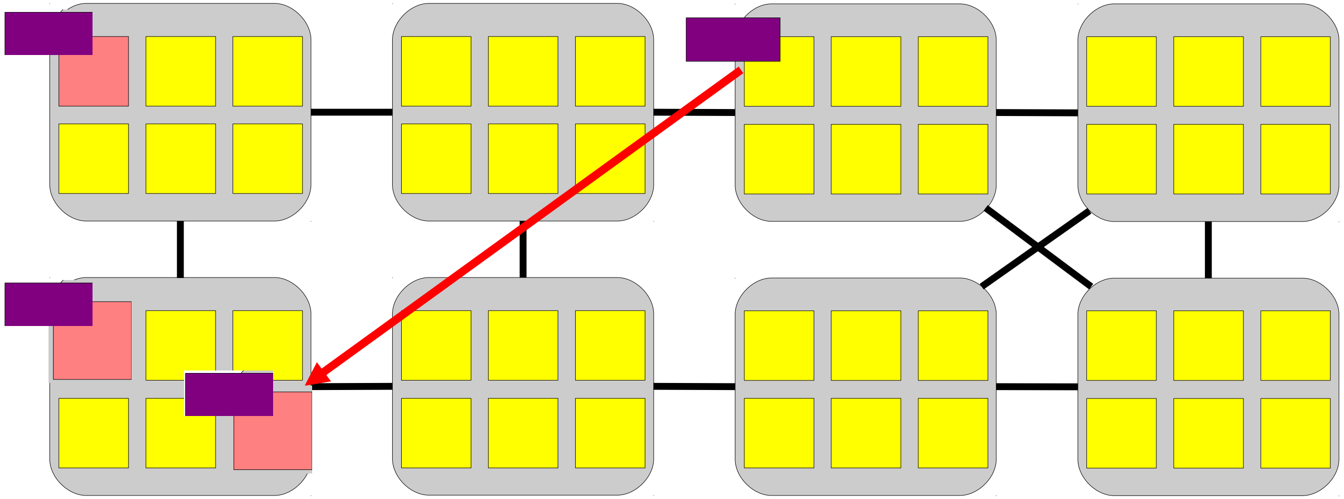
All waiters read the lock

```
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



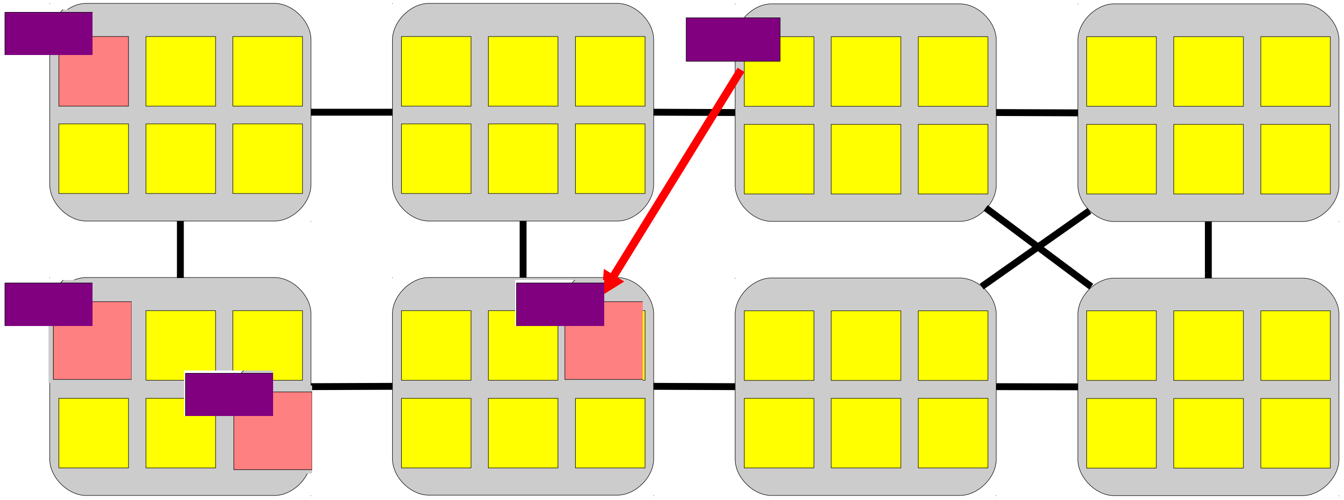
All waiters read the lock

```
void spin_lock(spinlock_t *lock)
{
    t = atomic_inc(lock->next_ticket);
    while (t != lock->current_ticket)
        ;  /* Spin */
}
```

```
void spin_unlock(spinlock_t *lock)
{
    lock->current_ticket++;
}
```

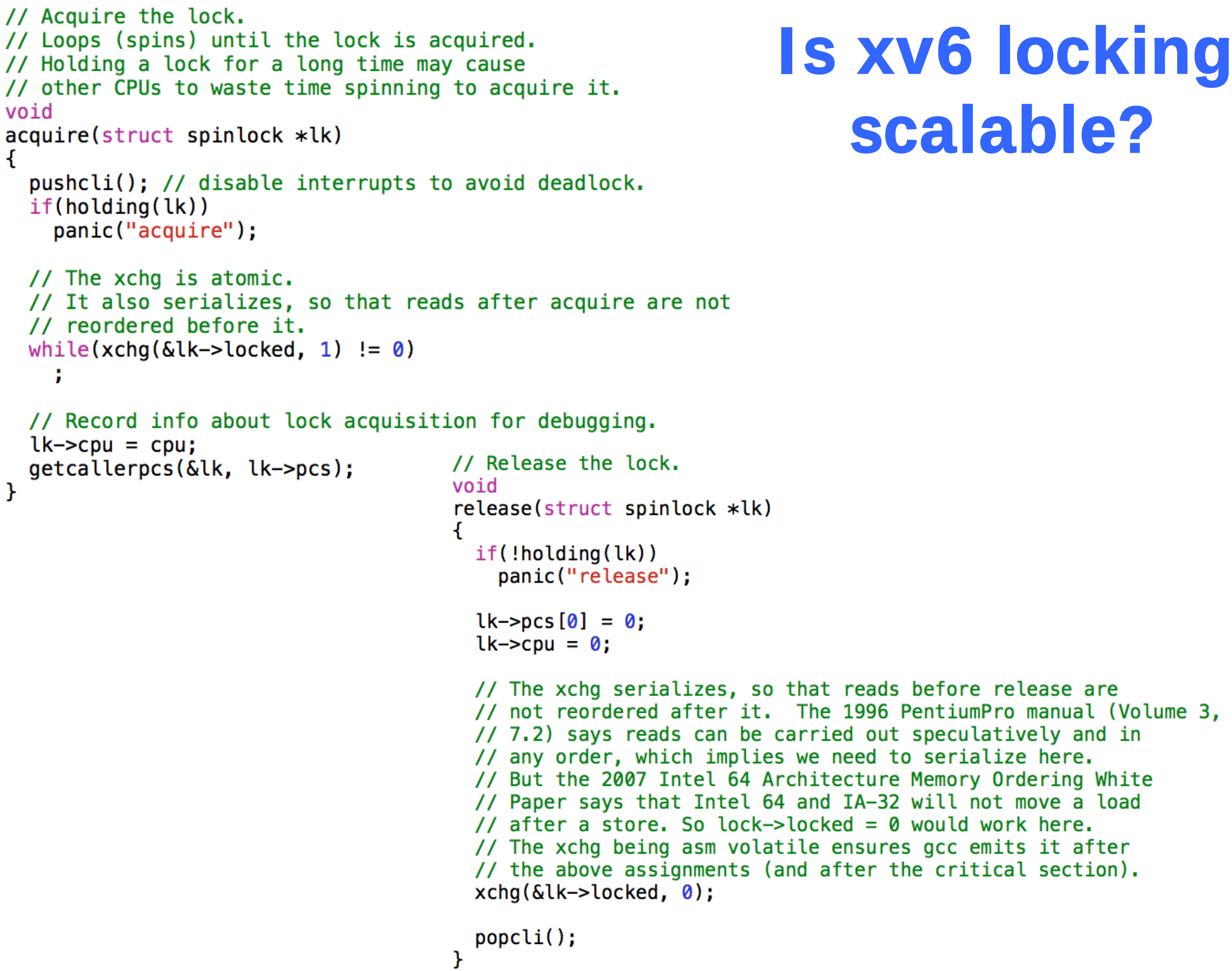
```
struct spinlock_t {
    int current_ticket;
    int next_ticket;
}
```

Previous lock holder notifies next lock holder after sending out N/2 replies



All waiters read the lock

Is xv6 locking scalable?



```c
// 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);
}
```

```c
// Release the lock.
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();
}
```

SCALABLE LOCKING

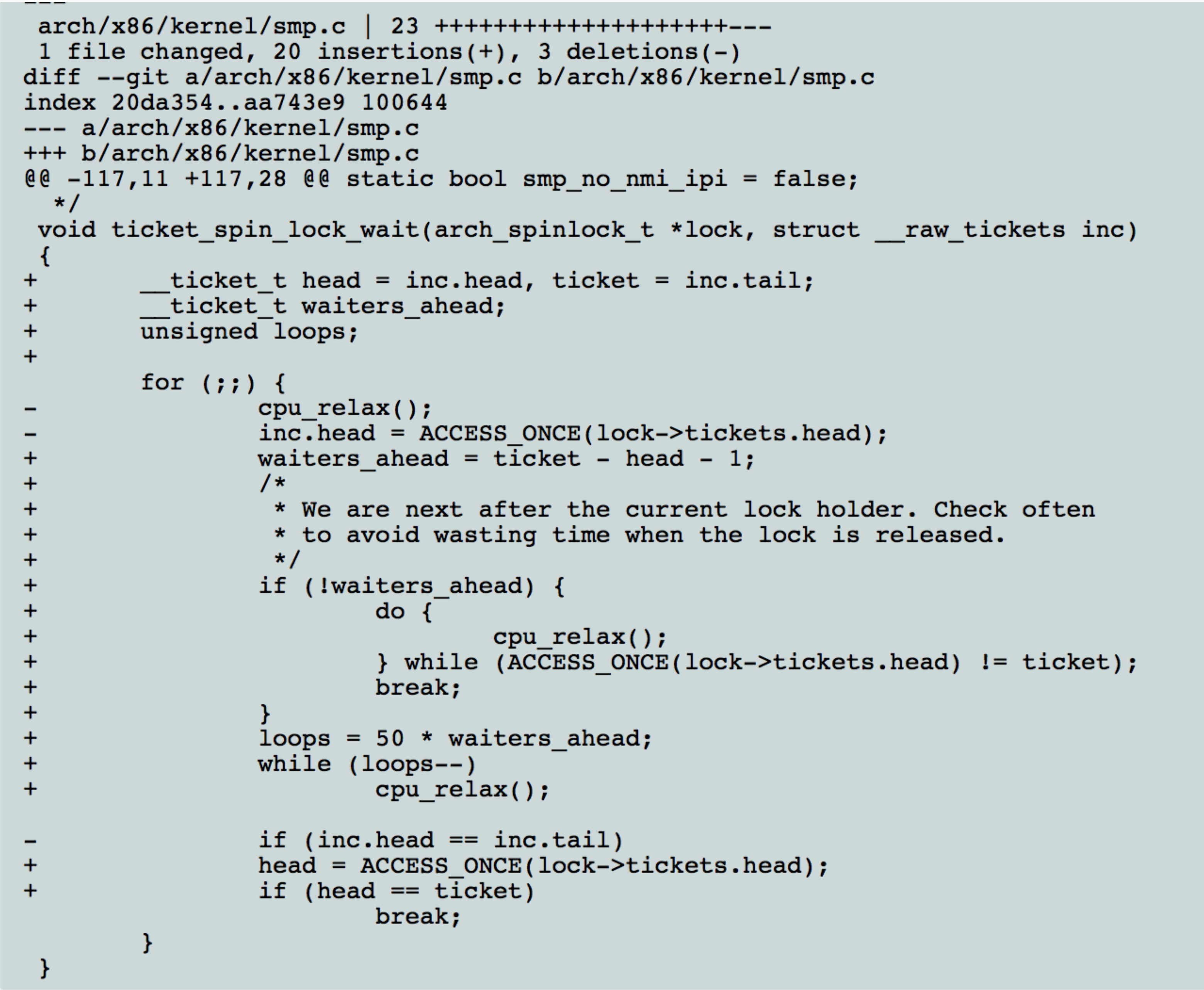
Making ticket spinlock scalable

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?

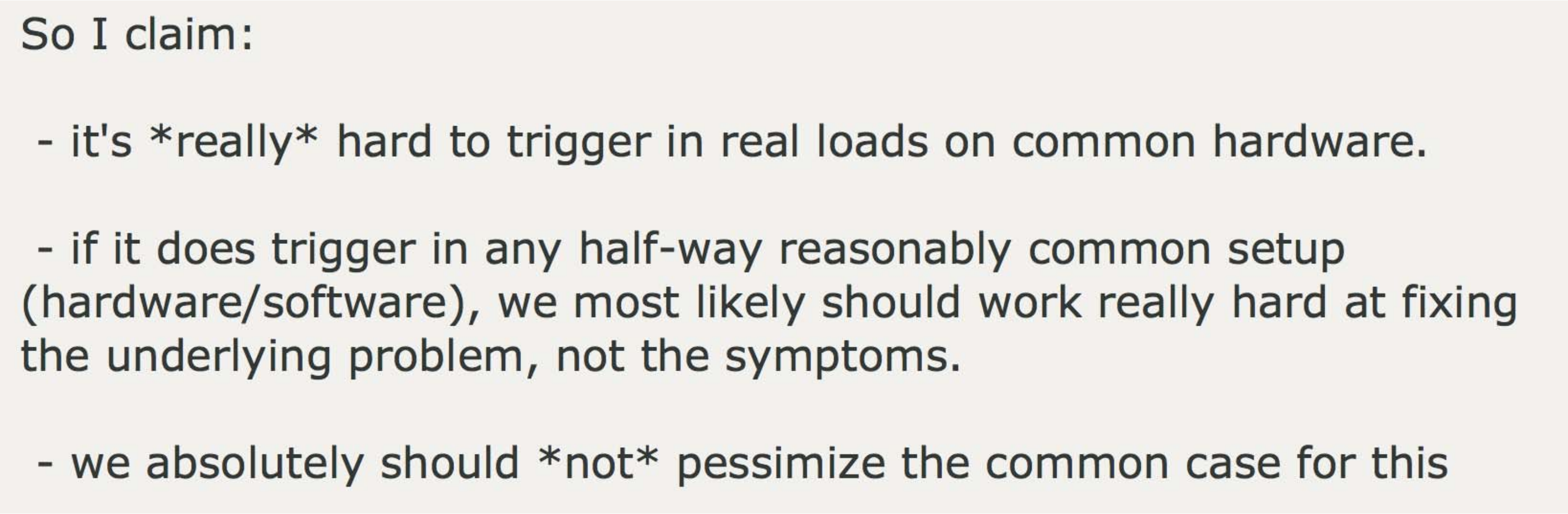
Example Fix



```
---
 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

Thanks

- Next class
  - MCS Lock
  - Read-writer lock, Lock-free synchronization

Synchronization Constructs

Rong Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/haibo_chen

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

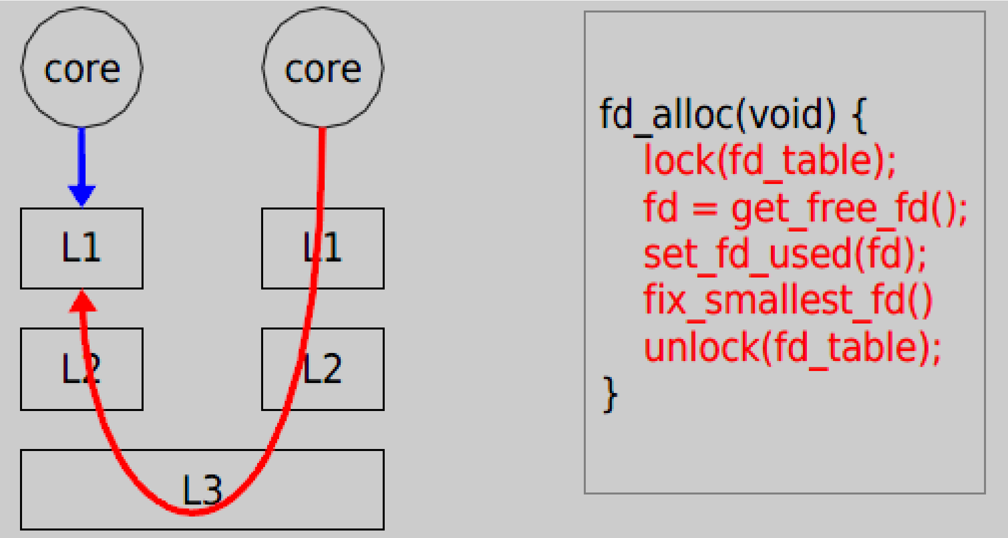
Recap: What is scalability?

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, ...)

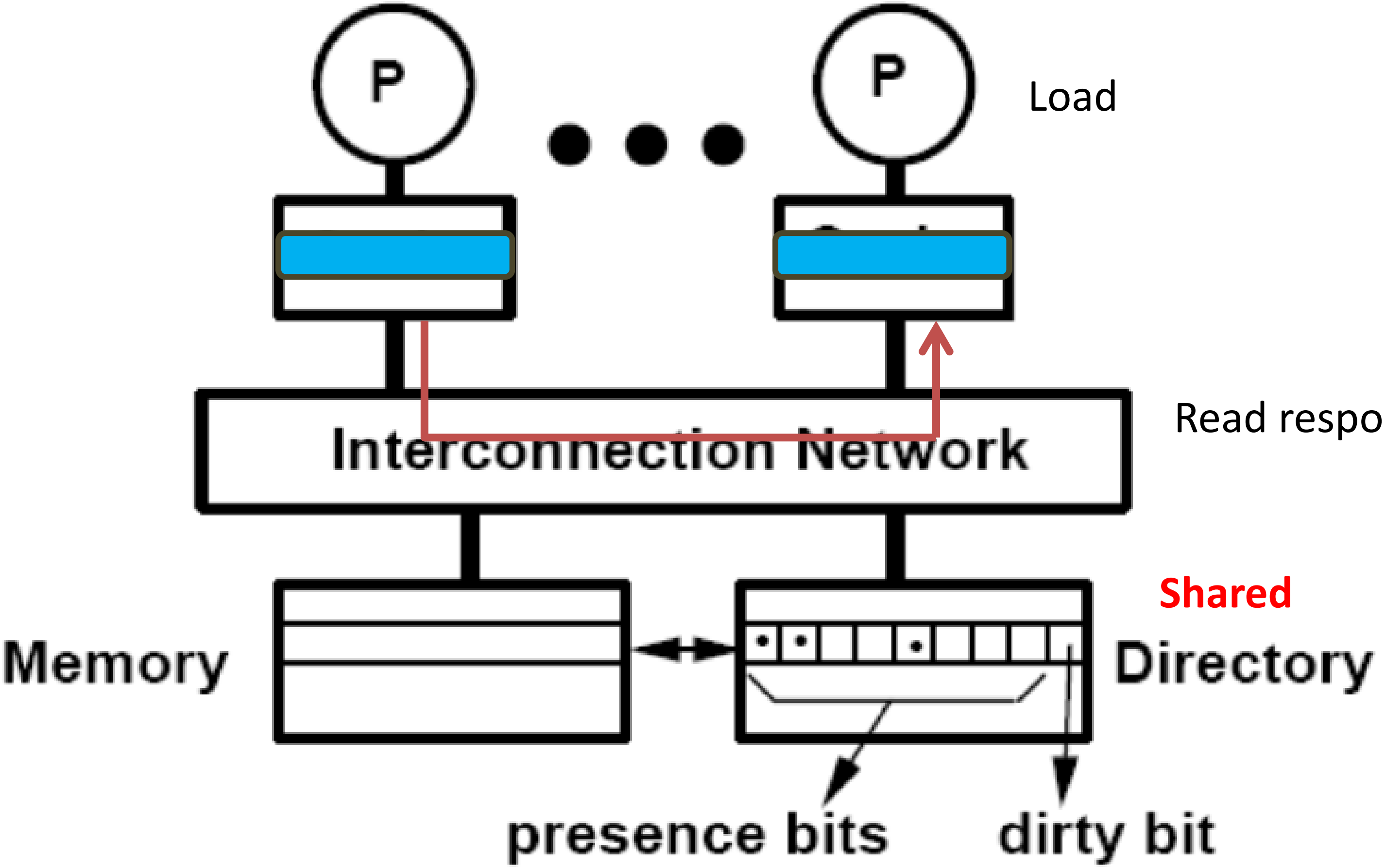
Recap: Why throughput drops?



```
fd_alloc(void) {
    lock(fd_table);
    fd = get_free_fd();
    set_fd_used(fd);
    fix_smallest_fd()
    unlock(fd_table);
}
```

Now it takes 121 cycles!

Recap: Directory-based cache coherence



Load

Read response

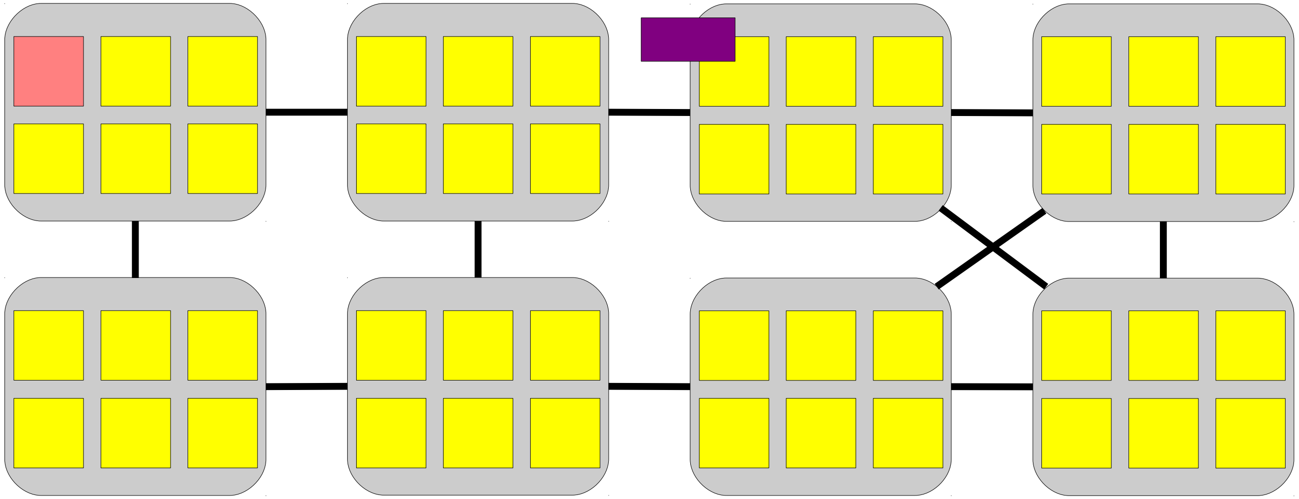
Shared

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;
}
```



Synchronization Constructs

MCS lock

Read-write Lock

Lock-free

Using scalable locks

Many existing scalable locks

Main idea is to avoid contending on a single cache line

Example

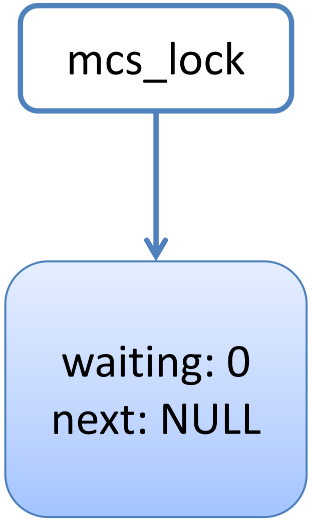
- MCS (John M. Mellor-Crummey and Michael L. Scott)
- K42

General idea of MCS lock

mcs_lock

NULL

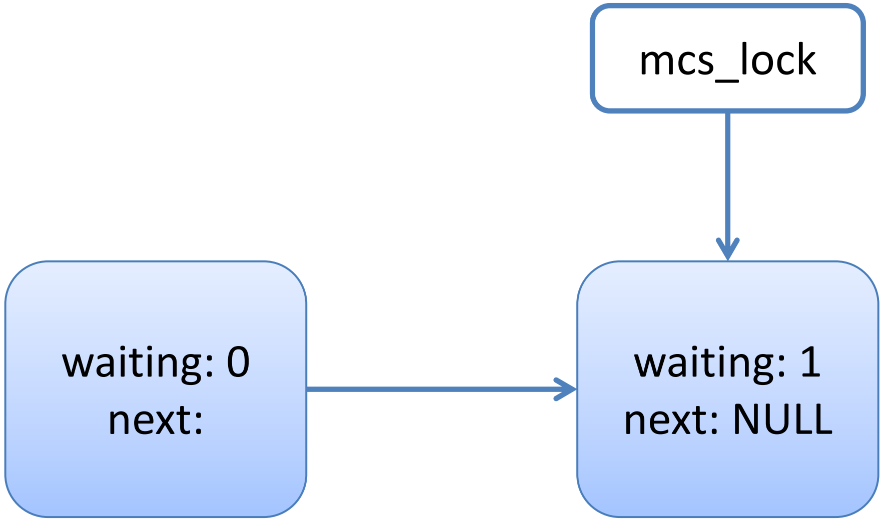
General idea of MCS lock



Use compare and swap to change
mcs_lock point to self node

Check previous node
NULL in this case, no need to wait

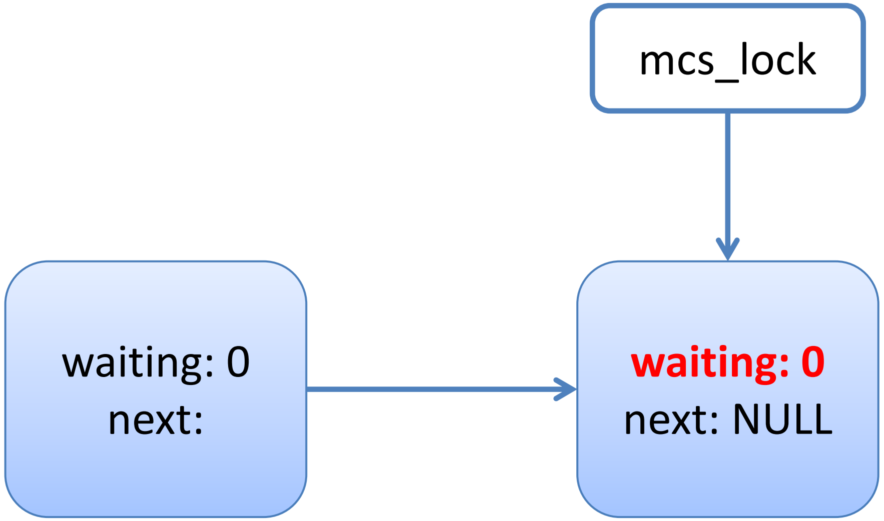
General idea of MCS lock



Previous node is not NULL
Current waiting is 1

Wait until lock holder
set waiting to 0

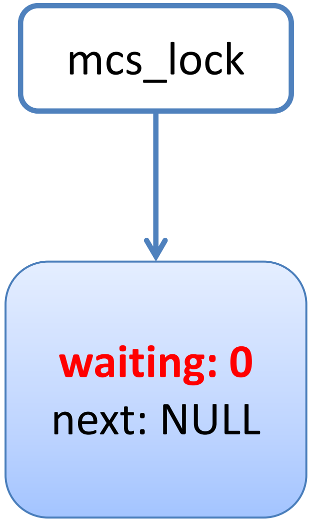
General idea of MCS lock



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

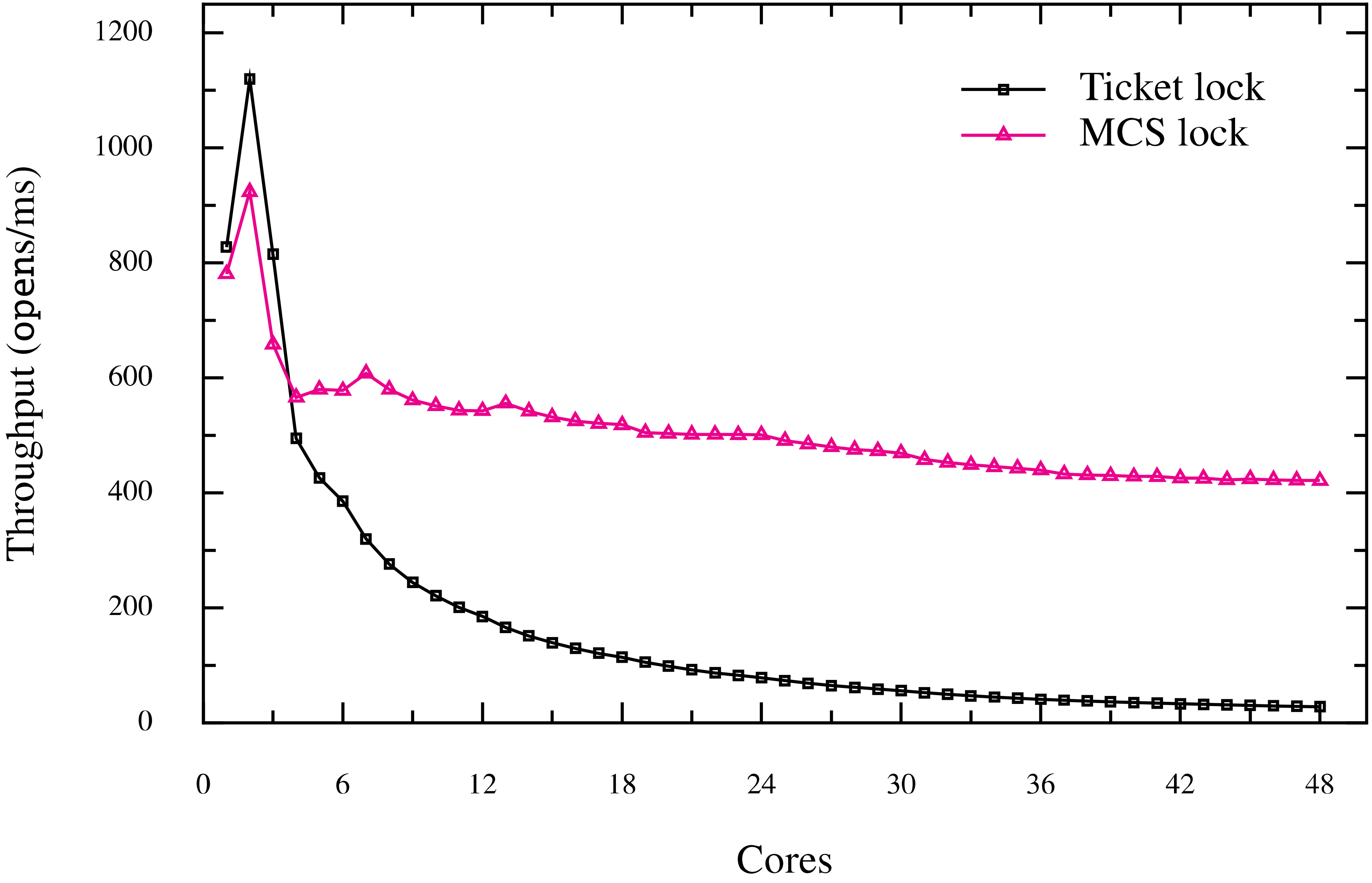
Wait until lock holder
set waiting to 0

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;
}
```

Using scalable locks: FOPS



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()

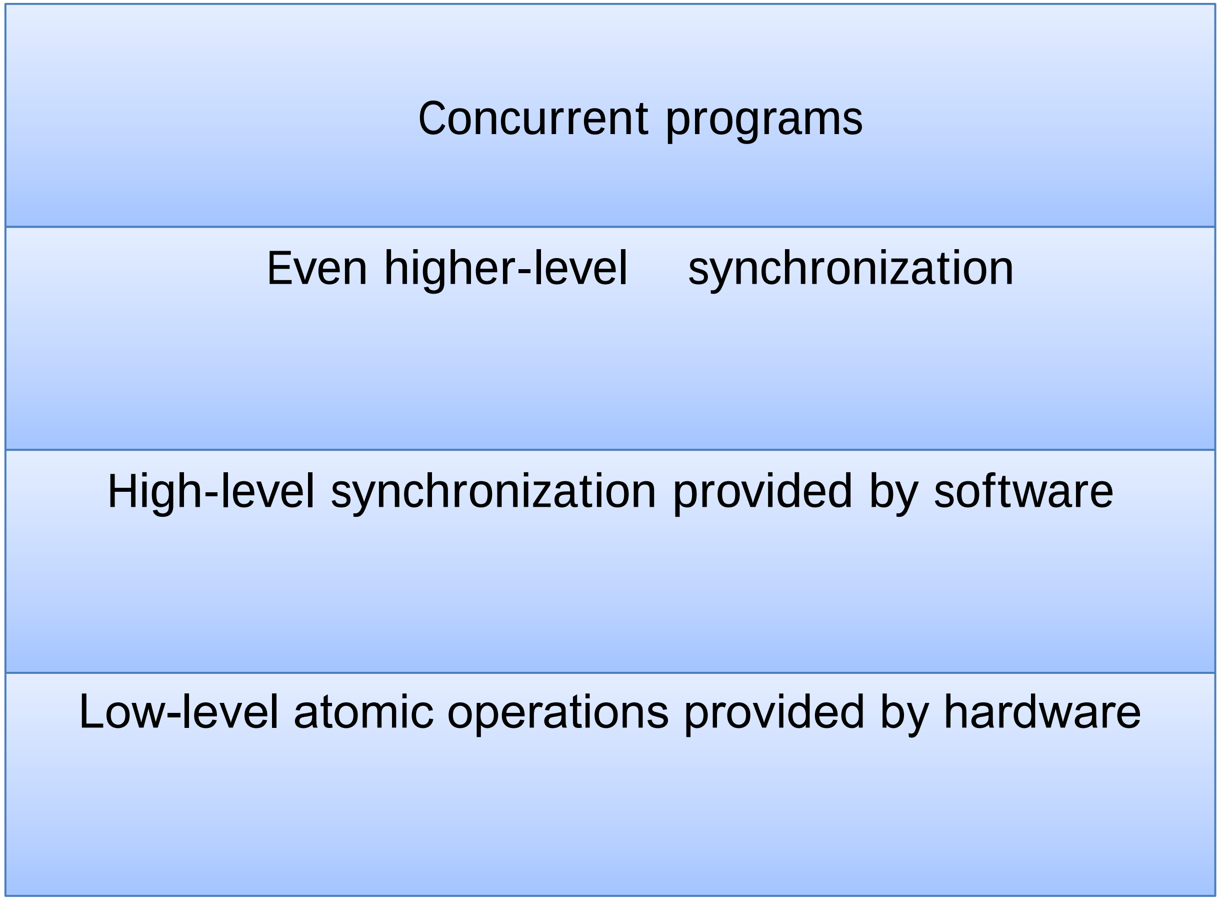
Many threads can be in between a readStart and readerFinish

Only one thread can be between writerStart and writierFinish

How to do with reader – writer locks?

```
GetEvents(date) {
 readerStart(maRWLock);
 List events = EventMap.find(date).copy();
 readerFinish(mapRWLock);
 return events;
}
AddEvent(data, newEvent) {
 writerStart(maRWLock);
 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);
}
```

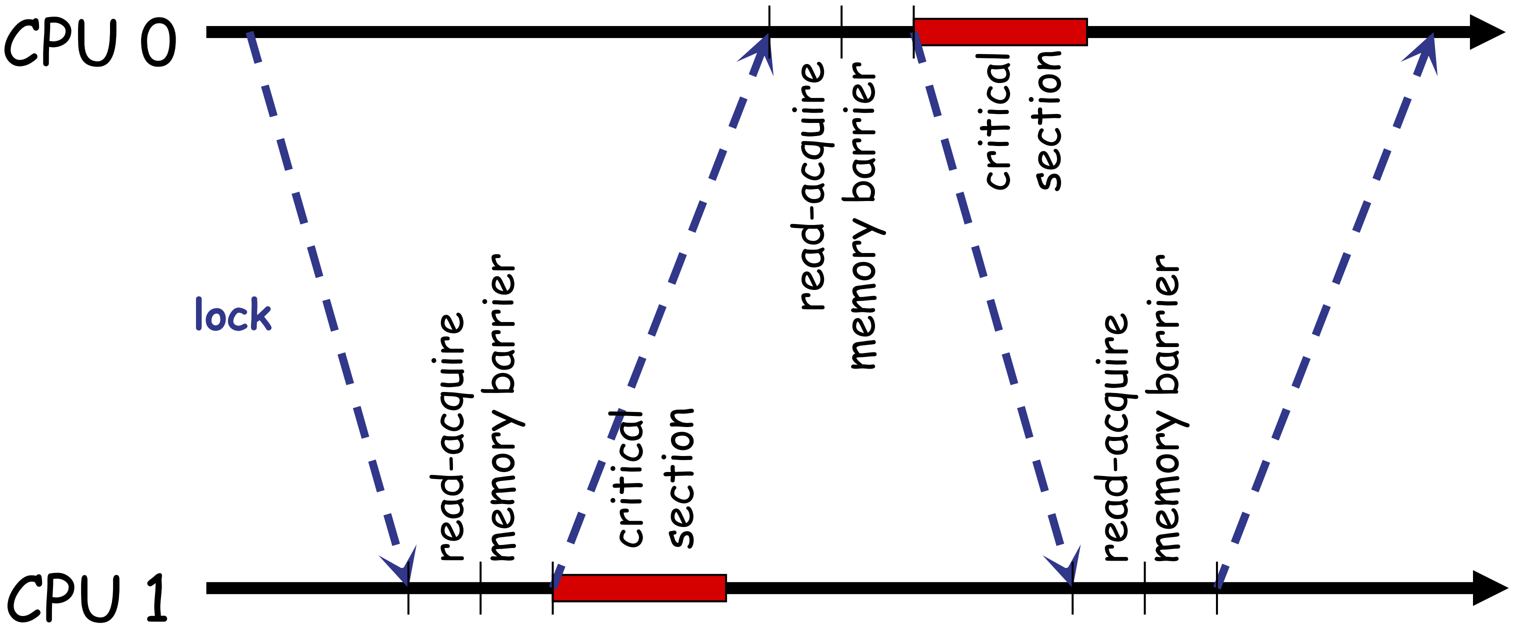
```
writerFinish() {
 lock(lockRW);

 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);
}
```

Scalability of reader/writer locking



Reader/writer locking does not scale due to critical section efficiency!

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);
```

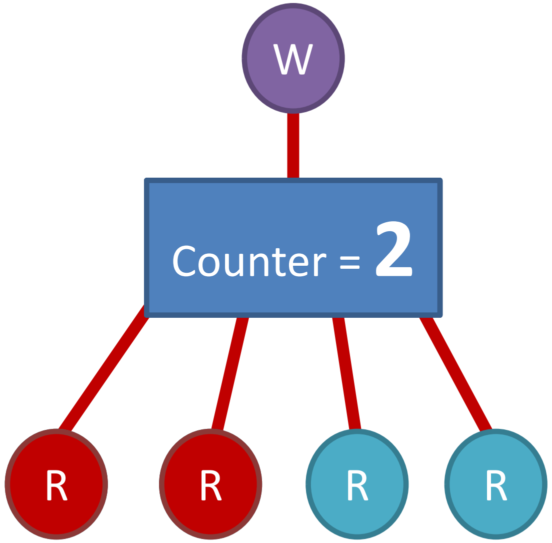
Scalability of rwlock?

Reading

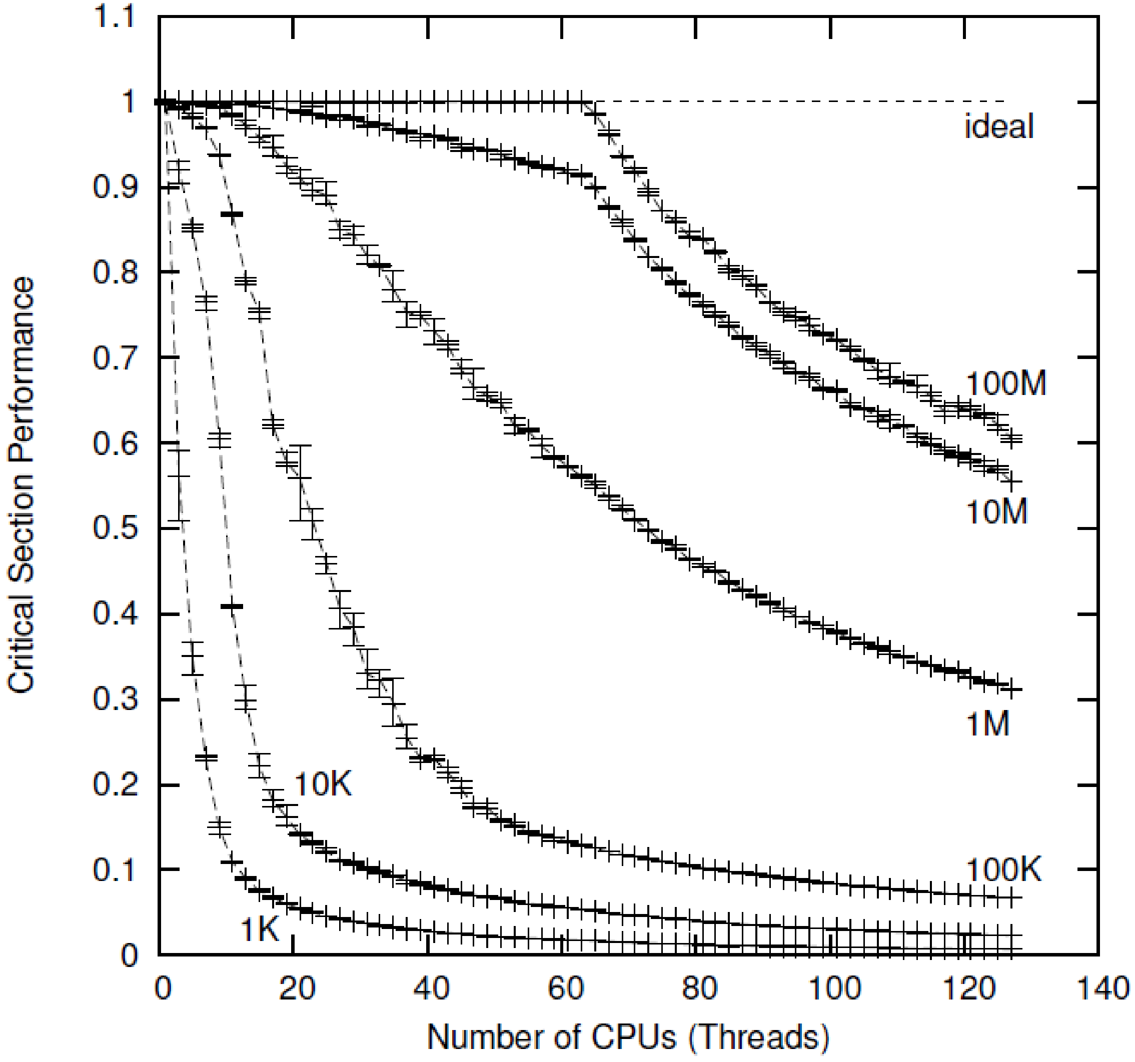
Counter -> #readers

Writing

Counter = -1



Traditional RW lock performance



Problem

Reader Lock Acquisition

add_and_featch(&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

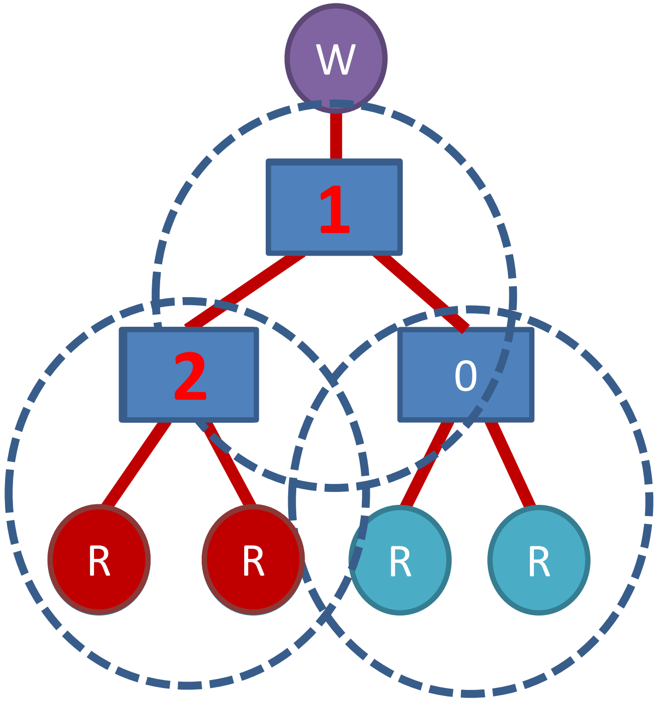
- Reduce time to wait for message (GOLL)

Idea 2

- Reduce #message (BR lock)

GOLL

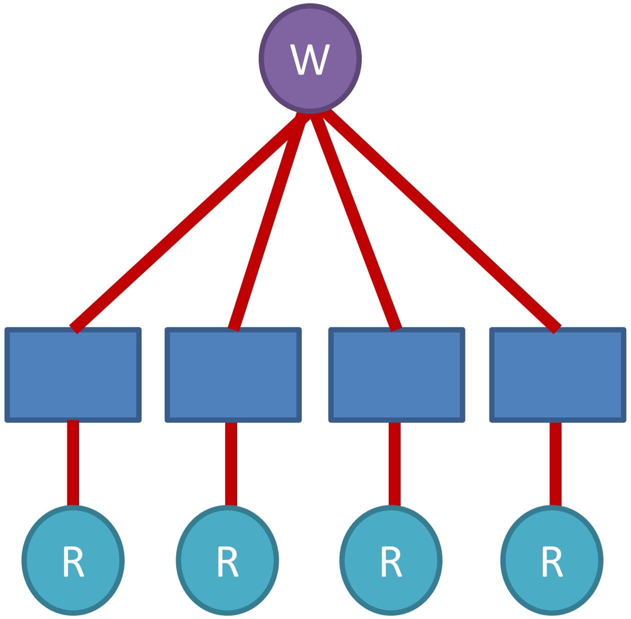
Reduce time to wait for message



Waiting domain is split into pieces
Reduce #remote messages

BR lock

Reduce #message



No reader messages if there’s no writer

Thanks

Next class

- Concurrency bugs, race detection

Bug Survey

Rong Chen
Institute of Parallel and Distributed Systems
Shanghai Jiao Tong University
http://ipads.se.sjtu.edu.cn/rong_chen

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(maRWLock);
 List events = EventMap.find(date).copy();
 readerFinish(mapRWLock);
 return events;
}
AddEvent(data, newEvent) {
 writerStart(maRWLock);
 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

Summary

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

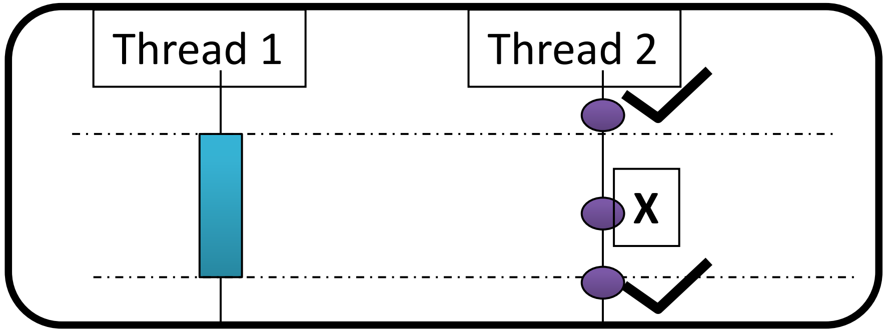
Categories:

- Atomicity violation
  - Desired atomicity of certain code region is violated
- Order violation
  - The desired order between two (sets of) accesses is flipped

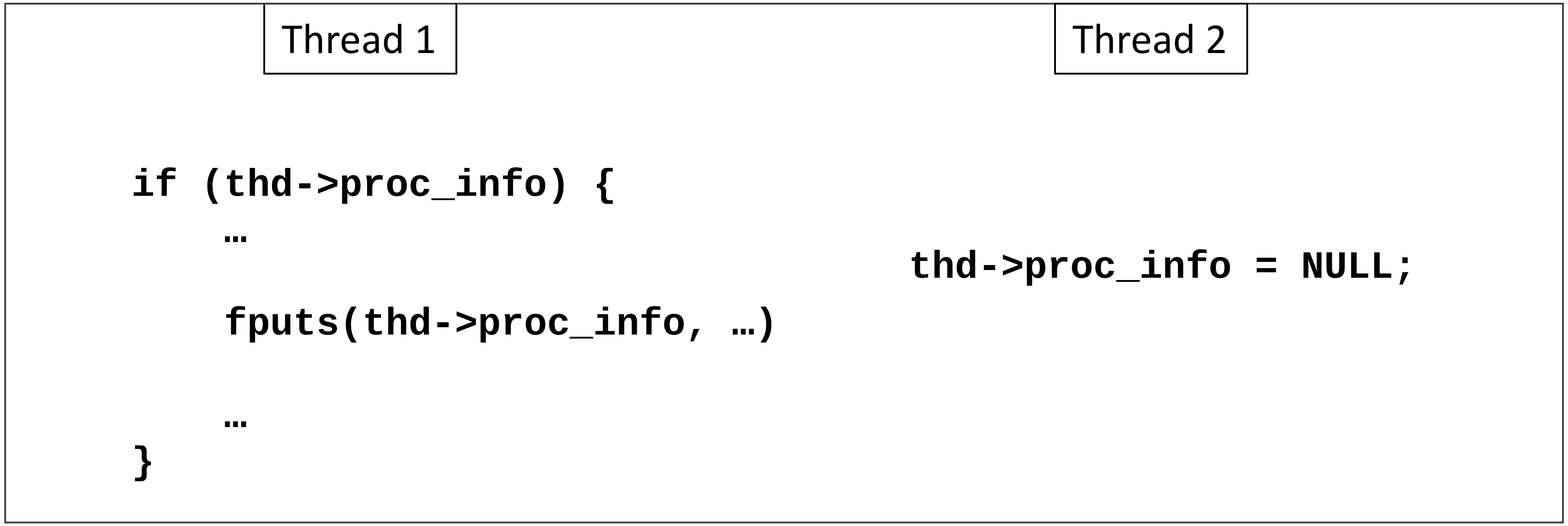
Others

Atomicity violation

The desired atomicity of certain code region is violated



Example of atomicity violation



Thread 1

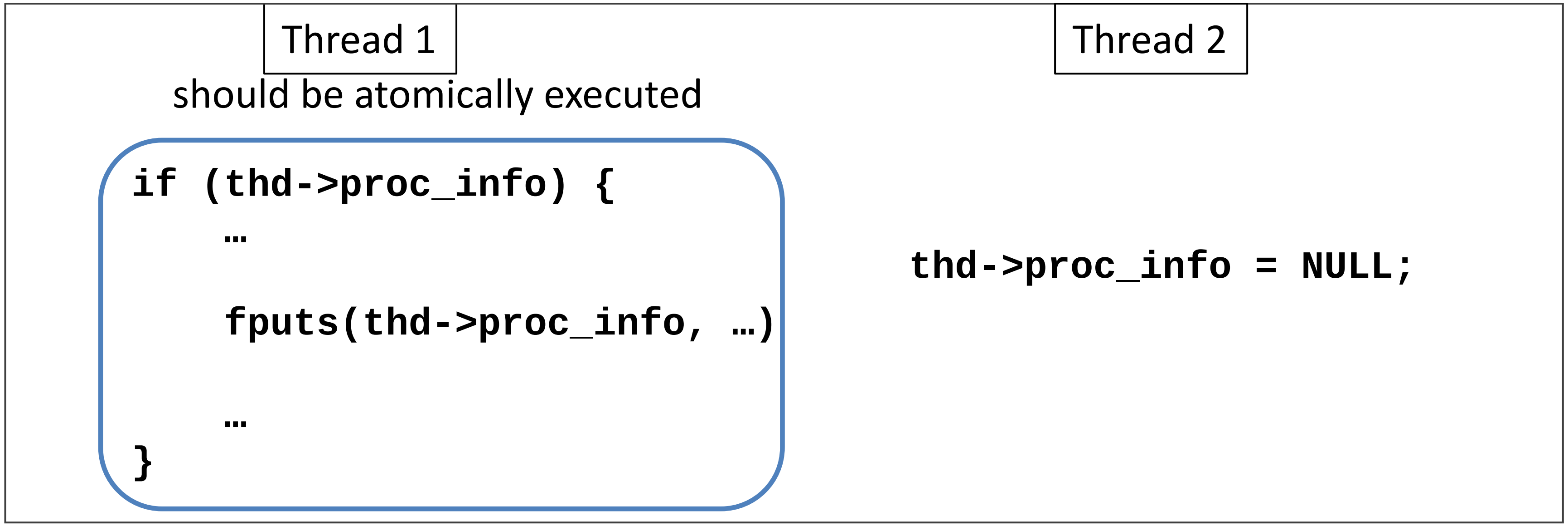
```
if (thd->proc_info) {
    …

    fputs(thd->proc_info, …)

    …
}
```

Thread 2

```
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp

Example of atomicity violation



Thread 1

should be atomically executed

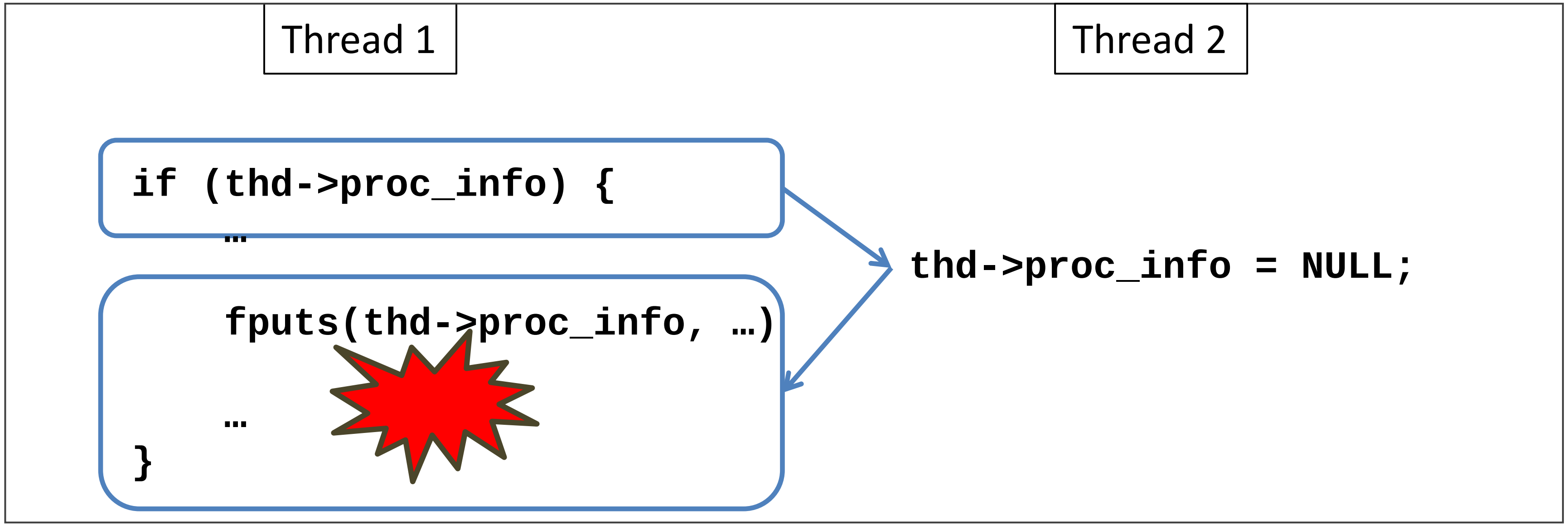
```
if (thd->proc_info) {
    …

    fputs(thd->proc_info, …)

    …
}
```

Thread 2

```
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp

Example of atomicity violation



Thread 1

```
if (thd->proc_info) {
    …
    fputs(thd->proc_info, …)

    …
}
```

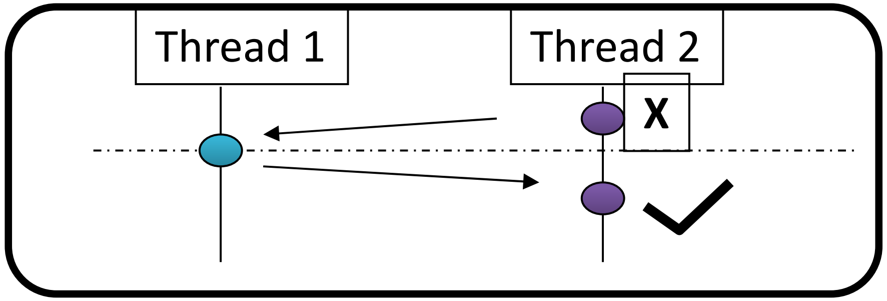
Thread 2

```
thd->proc_info = NULL;
```

MySQL ha_innodb.hpp

Order violation

- The desired order between two (sets of) accesses is flipped



Example of order violation

Thread 1

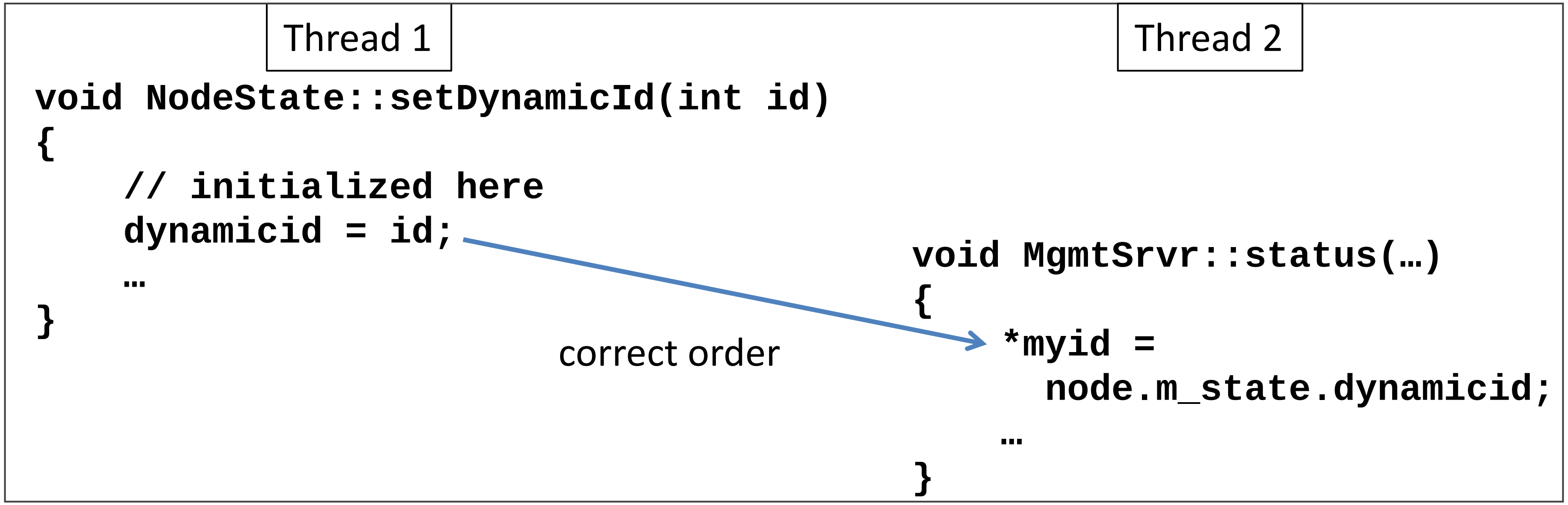
```
void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    …
}
```

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;
    …
}
```

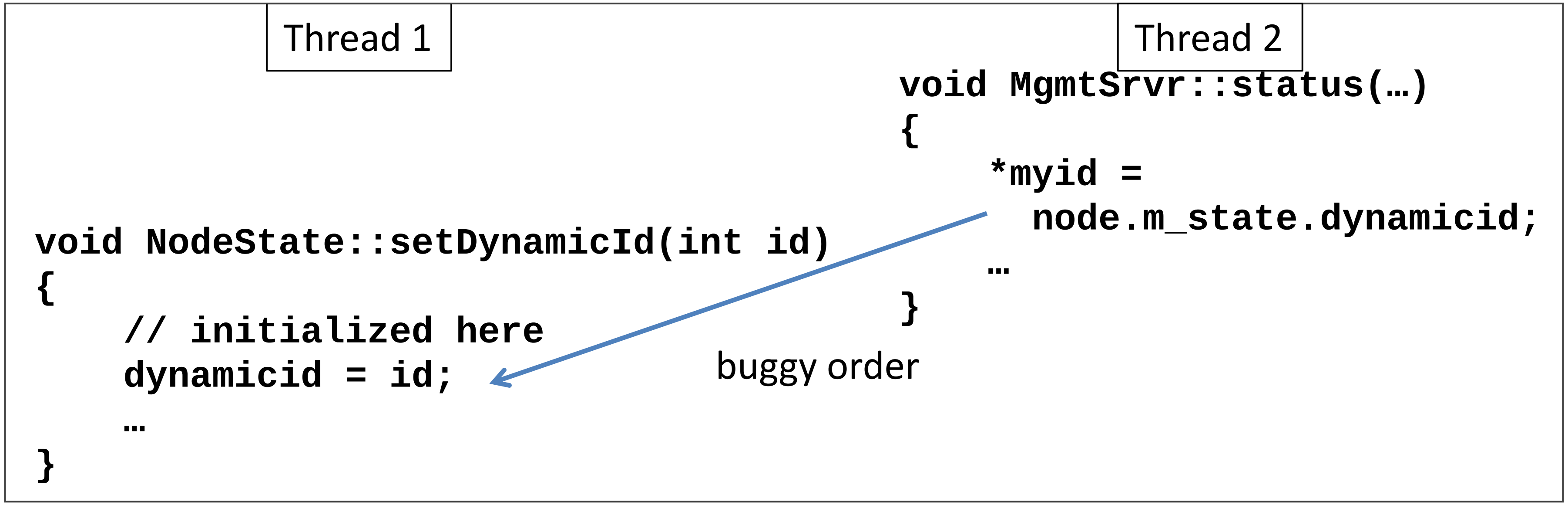
correct order

Thread 2

```
void MgmtSrvr::status(…)
{
    *myid =
      node.m_state.dynamicid;
    …
}
```

MySQL NodeState.hpp

Example of order violation



Thread 1

```
void NodeState::setDynamicId(int id)
{
    // initialized here
    dynamicid = id;
    …
}
```

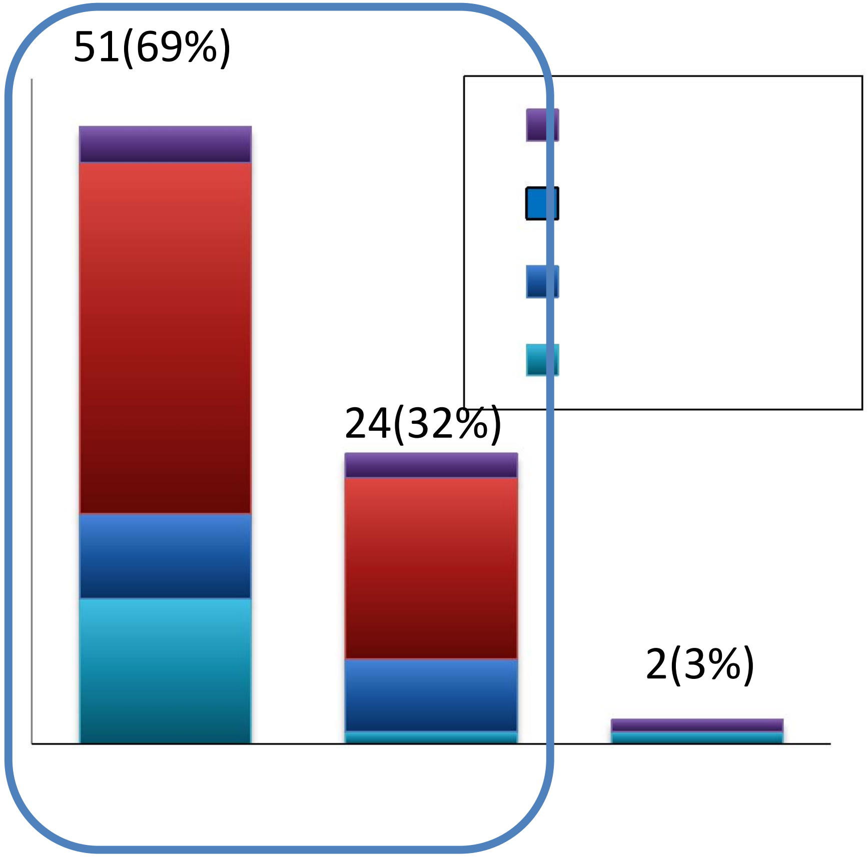
Thread 2

```
void MgmtSrvr::status(…)
{
    *myid =
      node.m_state.dynamicid;
    …
}
```

buggy order

MySQL NodeState.hpp

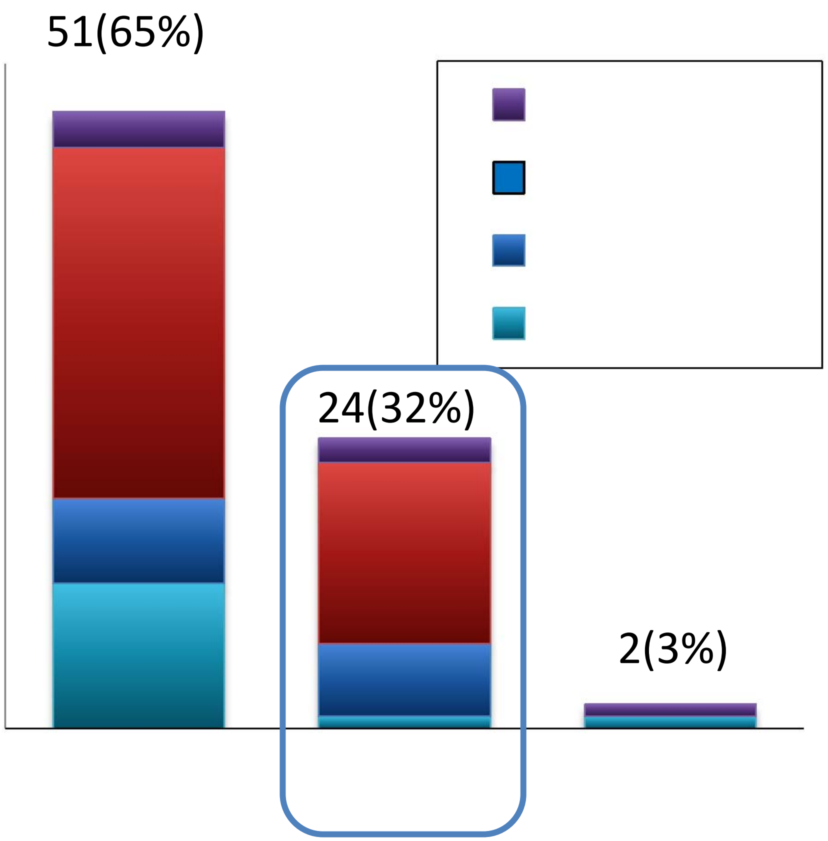
Non-deadlock bug pattern



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

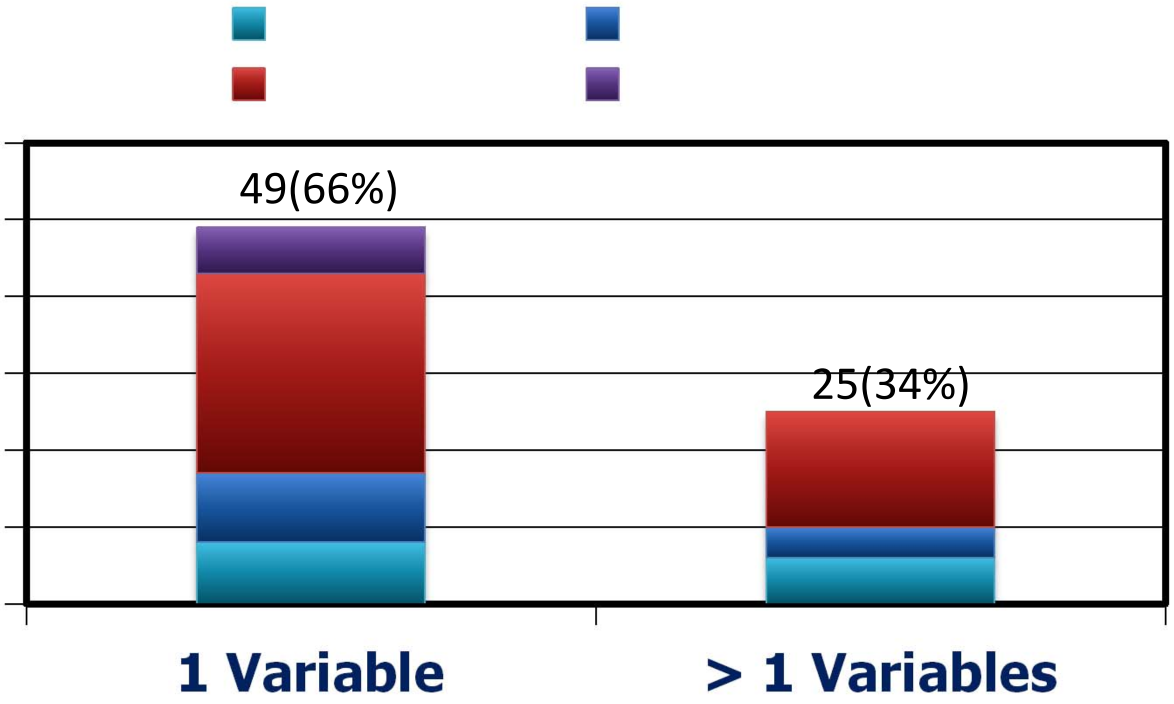
How many threads are involved?

How many variables are involved?

How many accesses are involved?

Single Variable vs. Multiple Variable

Findings



Single Variable vs. Multiple Variable

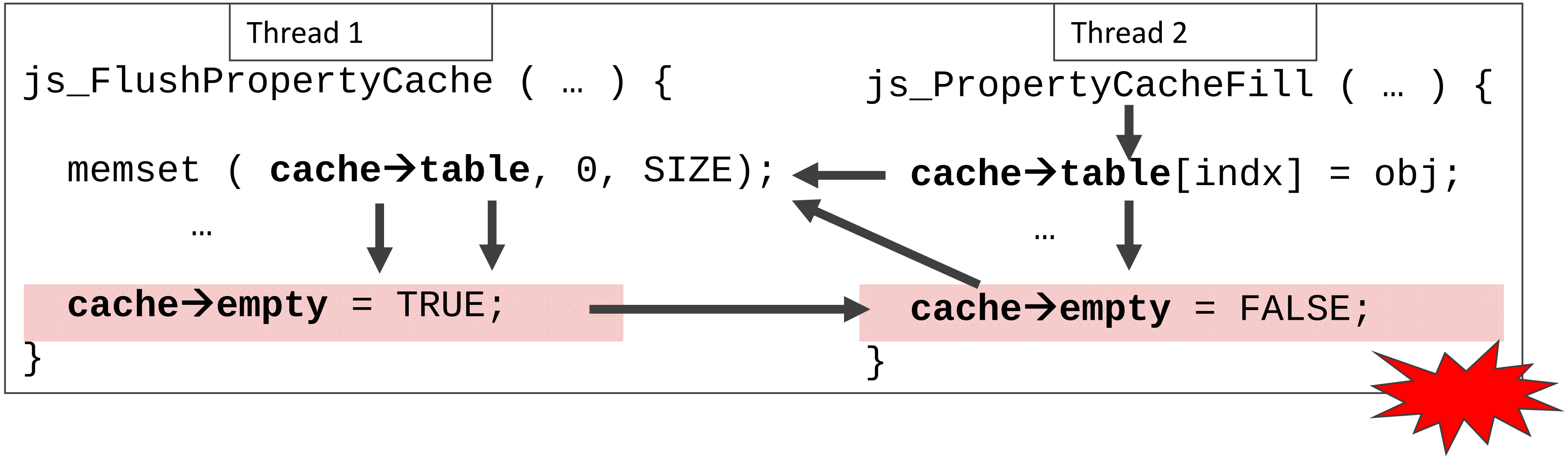
Single variables are more common

- 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



Thread 1

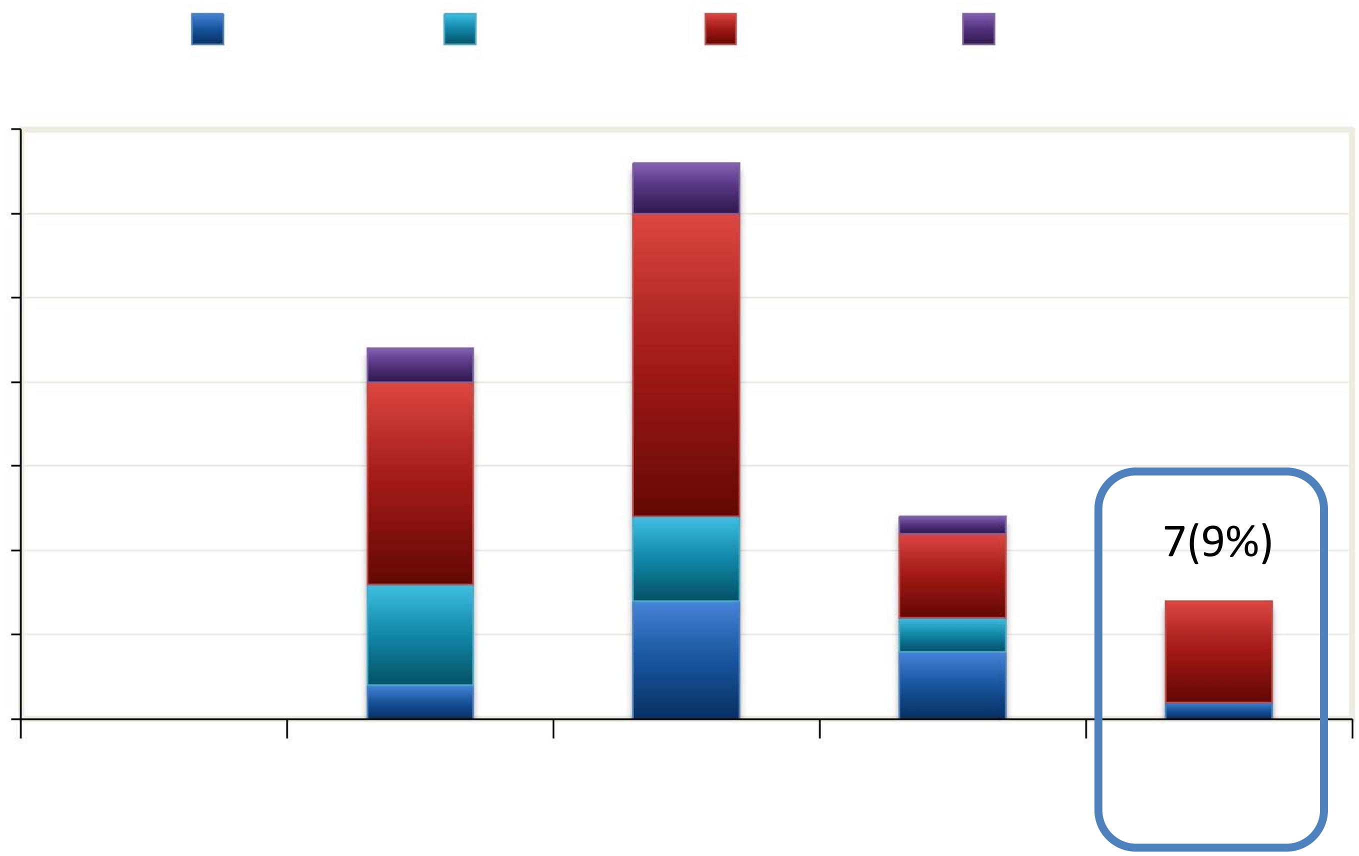
```
js_FlushPropertyCache ( … ) {

  memset ( cache→table, 0, SIZE);
      …
  cache→empty = TRUE;
}
```

Thread 2

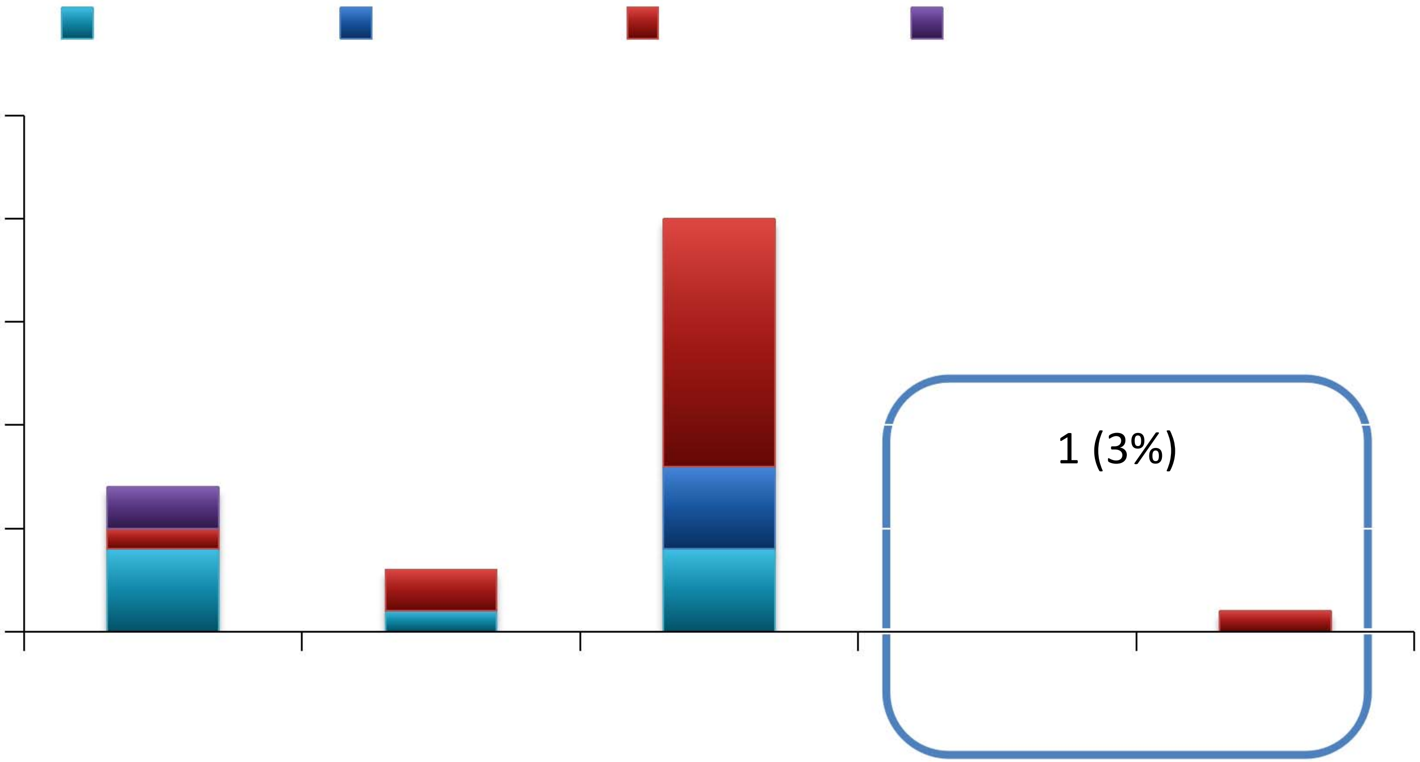
```
js_PropertyCacheFill ( … ) {

  cache→table[indx] = obj;
      …
  cache→empty = FALSE;
}
```

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 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 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

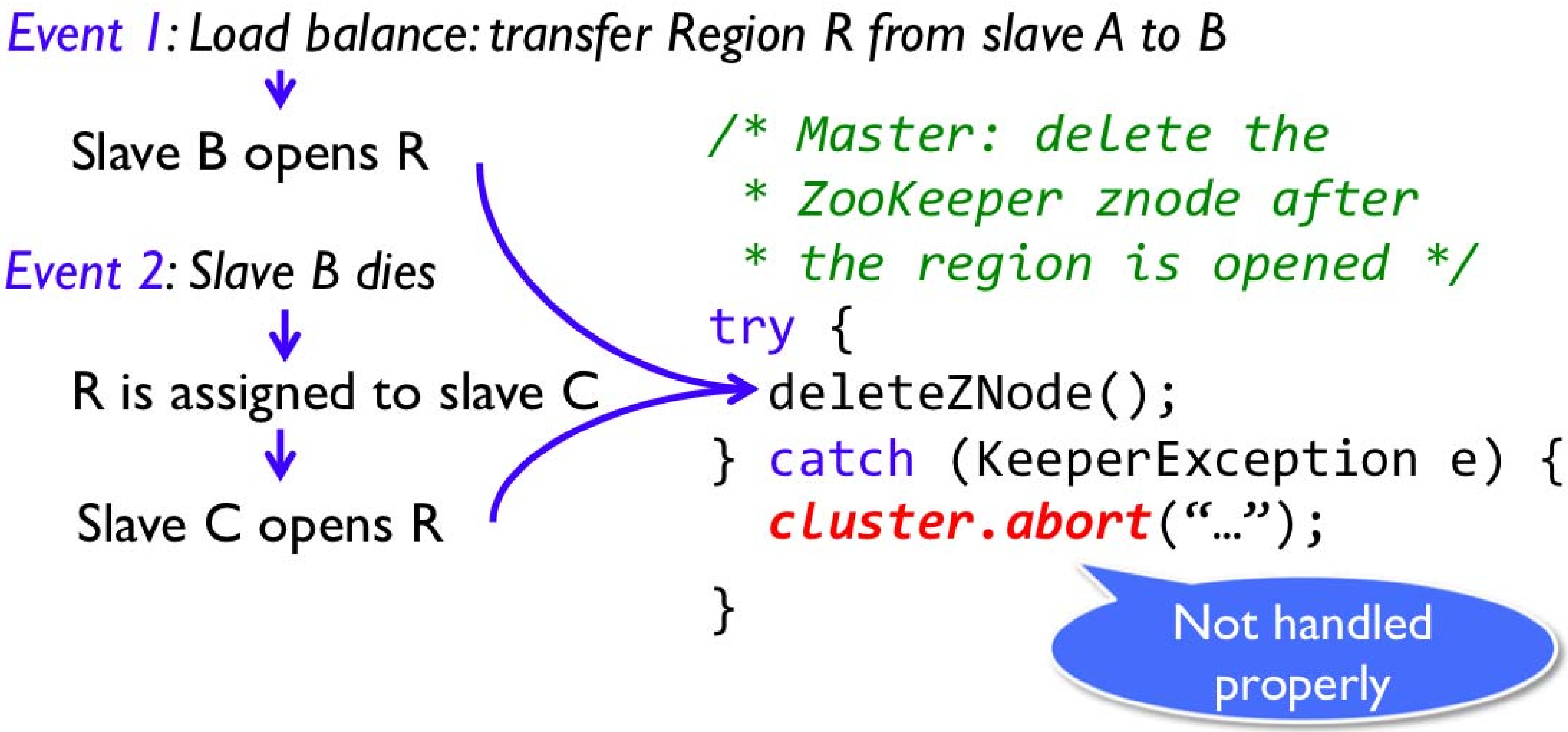
Fixing concurrency bugs is not trivial and not easy to get right

- Support from automated tools is needed

BUGS IN EXCEPTION HANDLERS

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

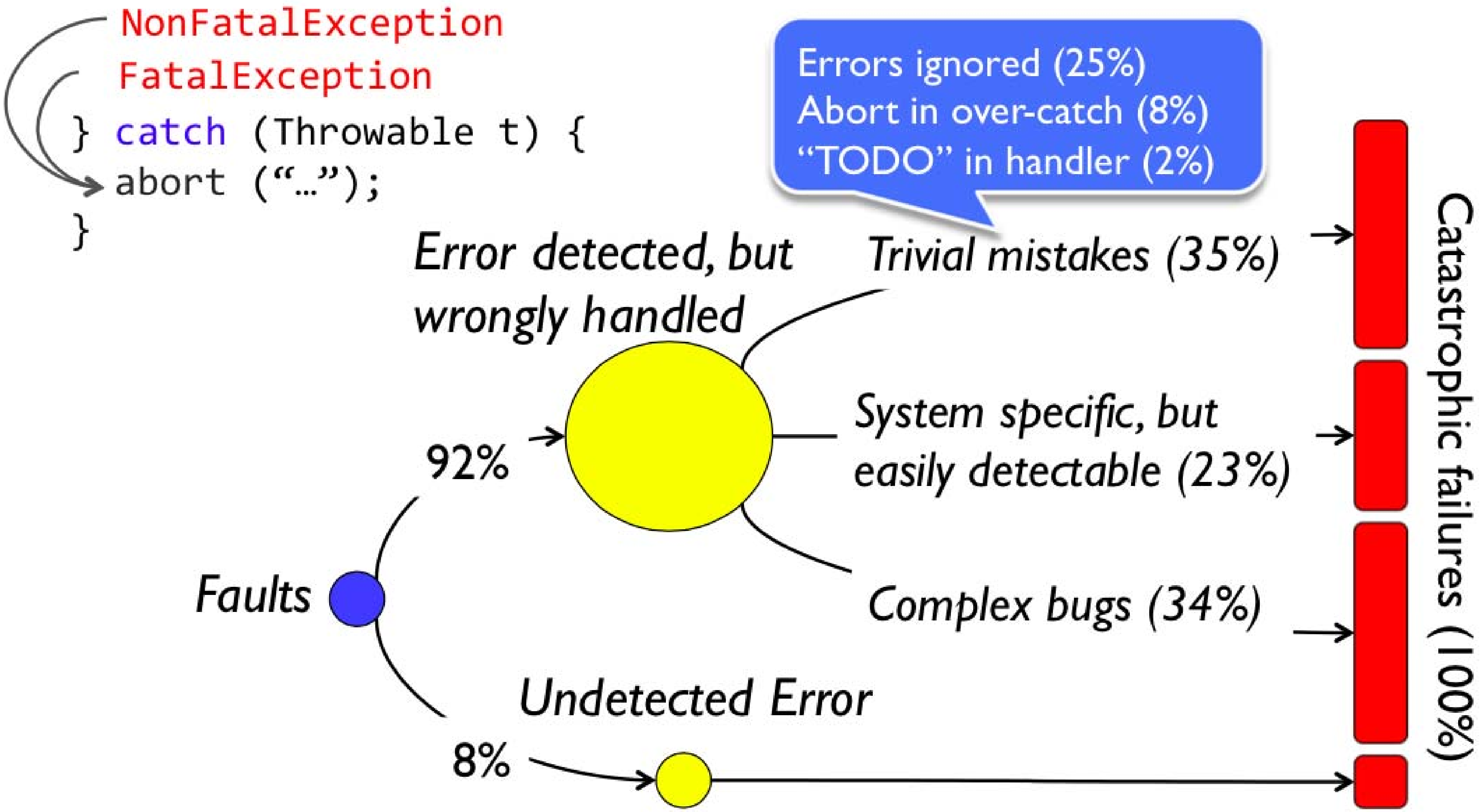
Slave C opens R

```
/* 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



```
  NonFatalException
  FatalException
} catch (Throwable t) {
  abort ("…");
}
```

Errors ignored (25%)
Abort in over-catch (8%)
"TODO" in handler (2%)

Error detected, but wrongly handled
Trivial mistakes (35%)
92%
System specific, but easily detectable (23%)
Faults
Complex bugs (34%)
Undetected Error
8%
Catastrophic failures (100%)

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.

What Bugs Means to You?

This Lecture

An example analysis of typical OS bugs

Describe several rules on OS bug finding

- Based on Invariants

Invariants in JOS

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 MUST belief:

- p is not null

A check (p == 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: forall 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 (iocommand.Direction == XFER_WRITE){
   if (copy_to_user(...)) {
        cmd_free(NULL, c);
        if (buff != NULL) kfree(buff);
        return( -EFAULT);
  }
}
if (iocommand.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

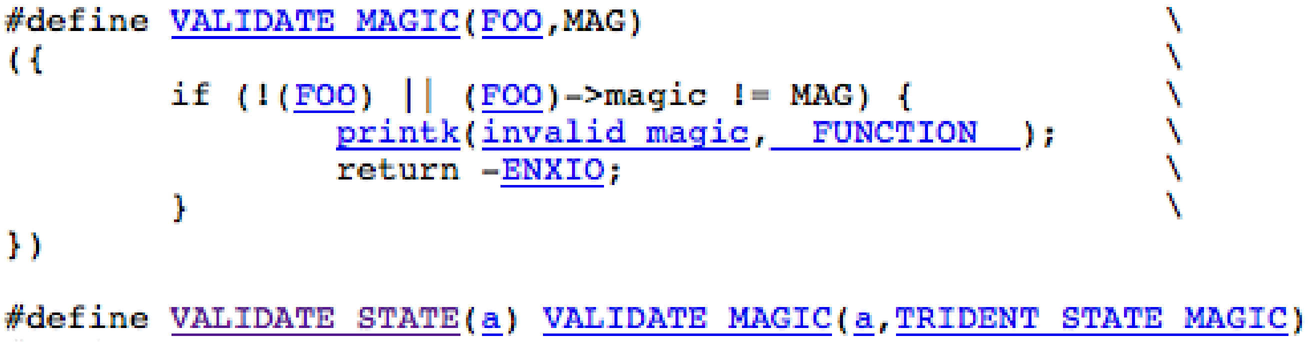
Checking derived lock functions

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)
```

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

Next Class

Data Race

Deadlock

Data Race & DeadLock

Rong Chen

Outline

Happens-before Race Detection

Lockset-Based Race Detection

Deadlocks

Detecting deadlocks using Lockset Analysis

Data Race Detection

What is Race?

Data Race

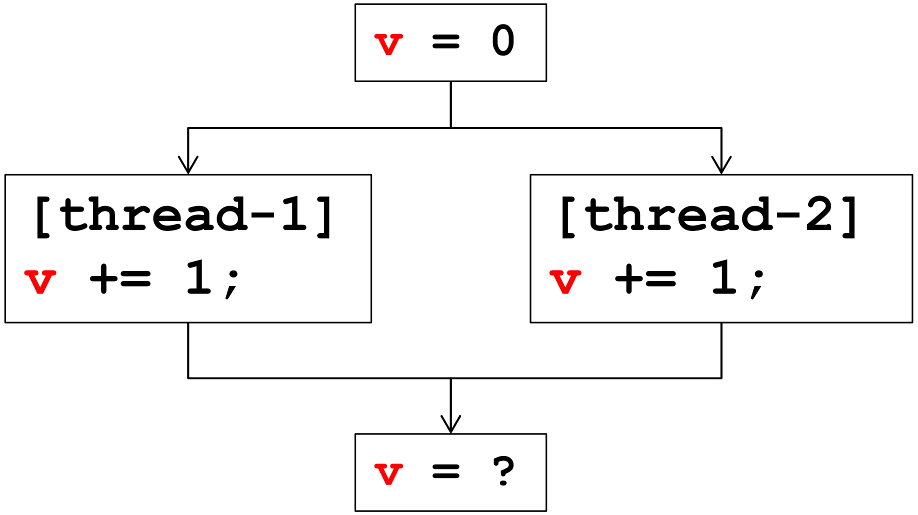
An undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but the operations must be done in the proper sequence in order to be done correctly

- multithread
- distributed Programs

Key reason

- separate processes or threads of execution depends on same shared state

Data Race



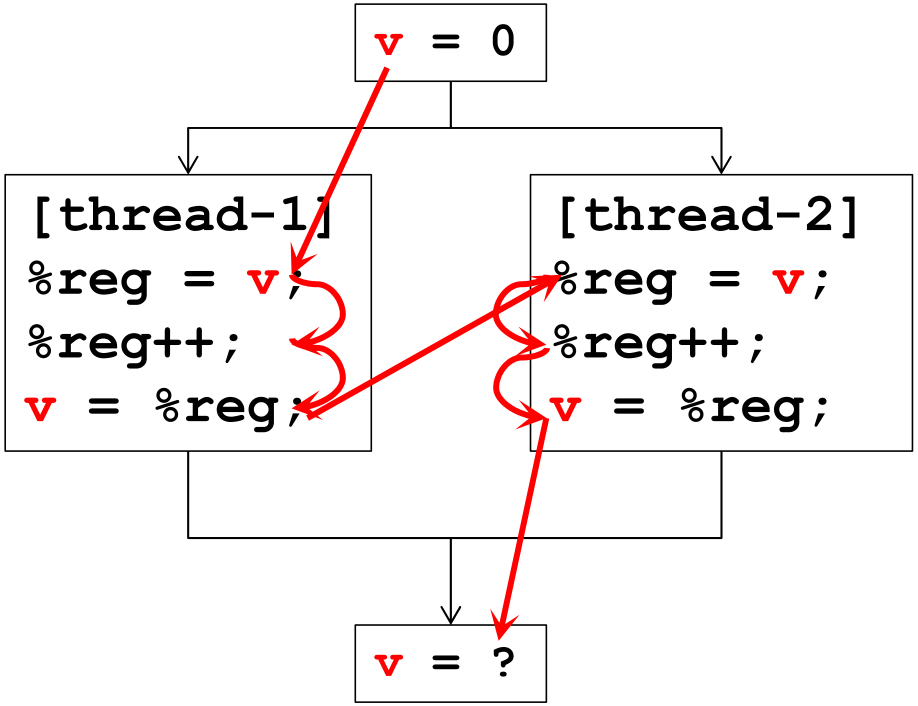
v = 0

[thread-1]
v += 1;

[thread-2]
v += 1;

v = ?

Data Race

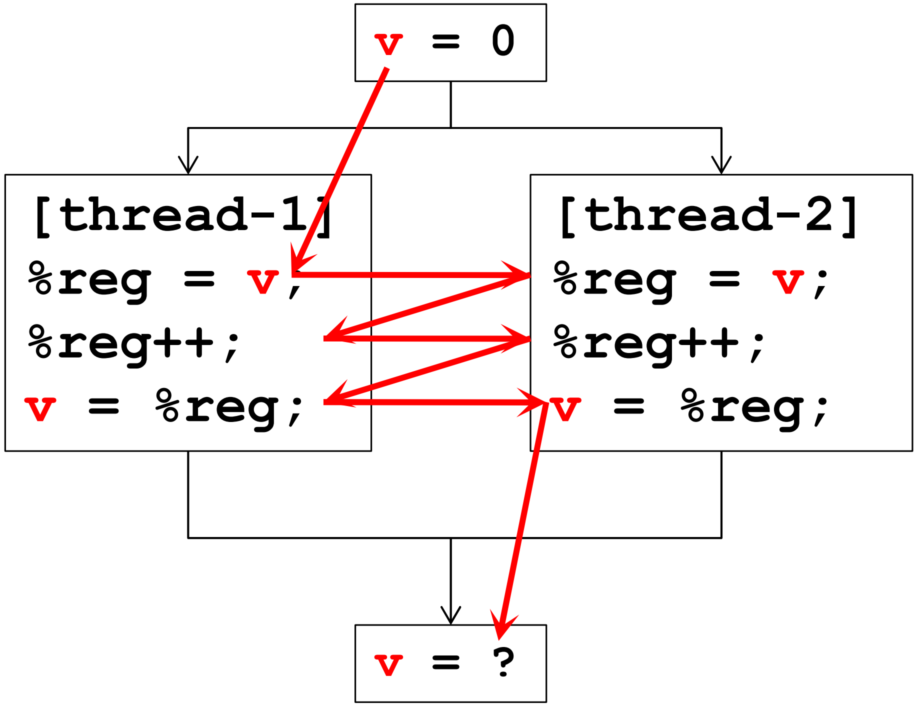


v = 0 (memory)

```
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
```

```
[thread-1]
%reg = v;
%reg++;
v = %reg;
```

```
[thread-2]
%reg = v;
%reg++;
v = %reg;
```

```
v = ?
```

```
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)
```

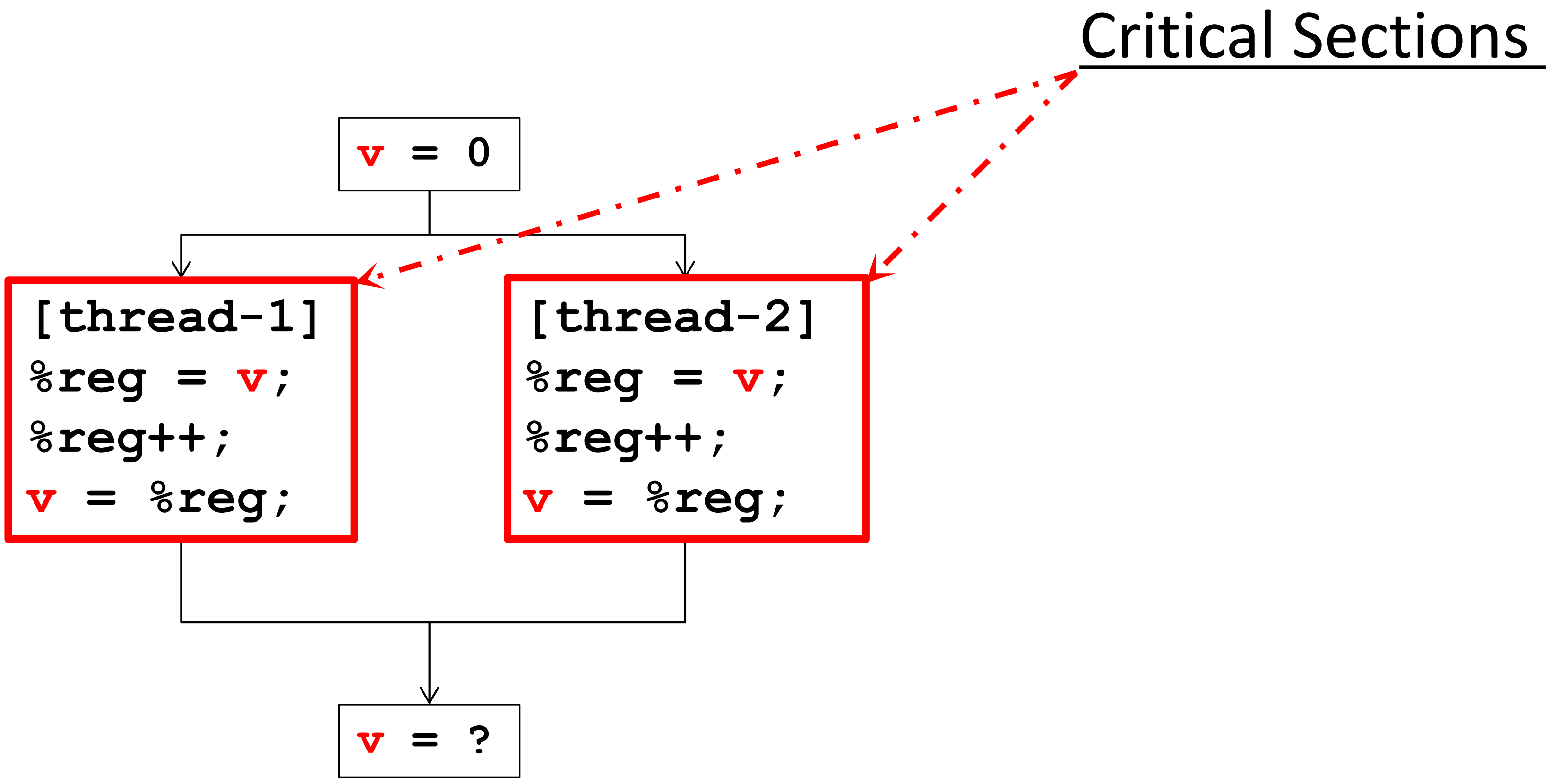
```
v = 1; (memory)
```

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



Critical Sections

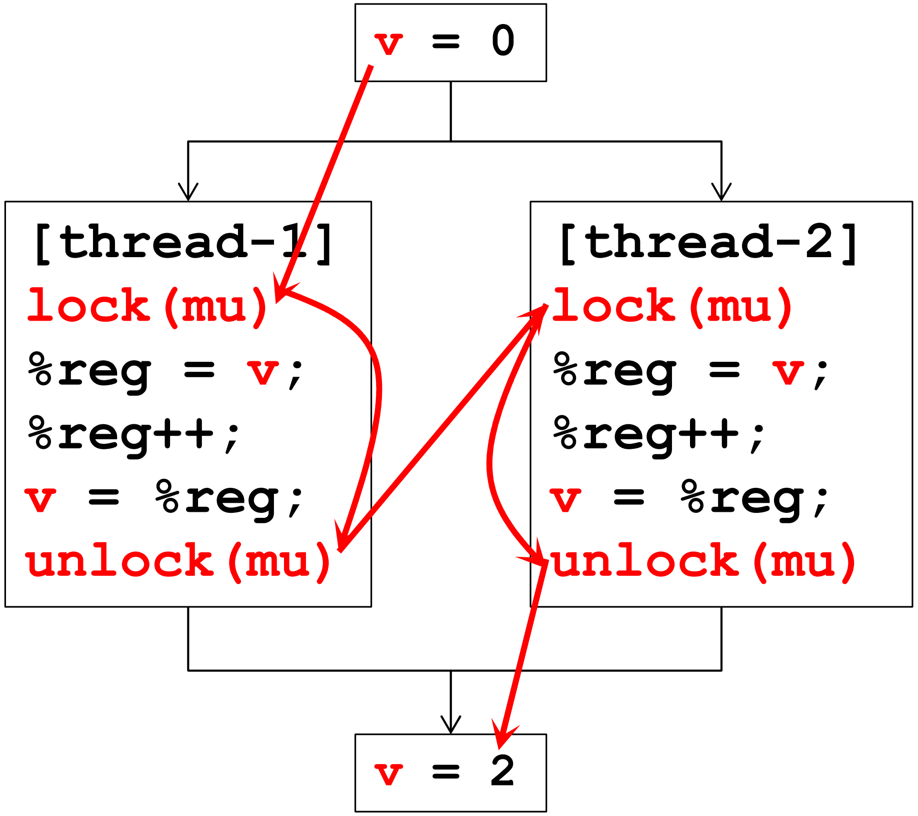
```
v = 0
```

```
[thread-1]
%reg = v;
%reg++;
v = %reg;
```

```
[thread-2]
%reg = v;
%reg++;
v = %reg;
```

```
v = ?
```

Remove Race Condition



```
v = 0
```

```
[thread-1]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)
```

```
[thread-2]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)
```

```
v = 2
```

<u>Mutually-exclusive</u>

- 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:

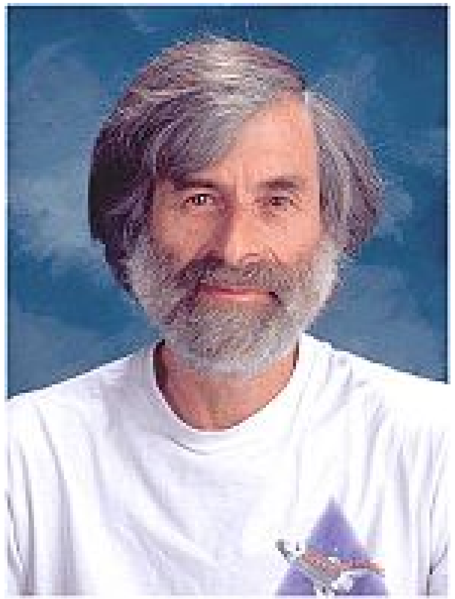
- Happens-before based
- Lockset based

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



*Times, Clocks, and Ordering of Events in a Distributed System

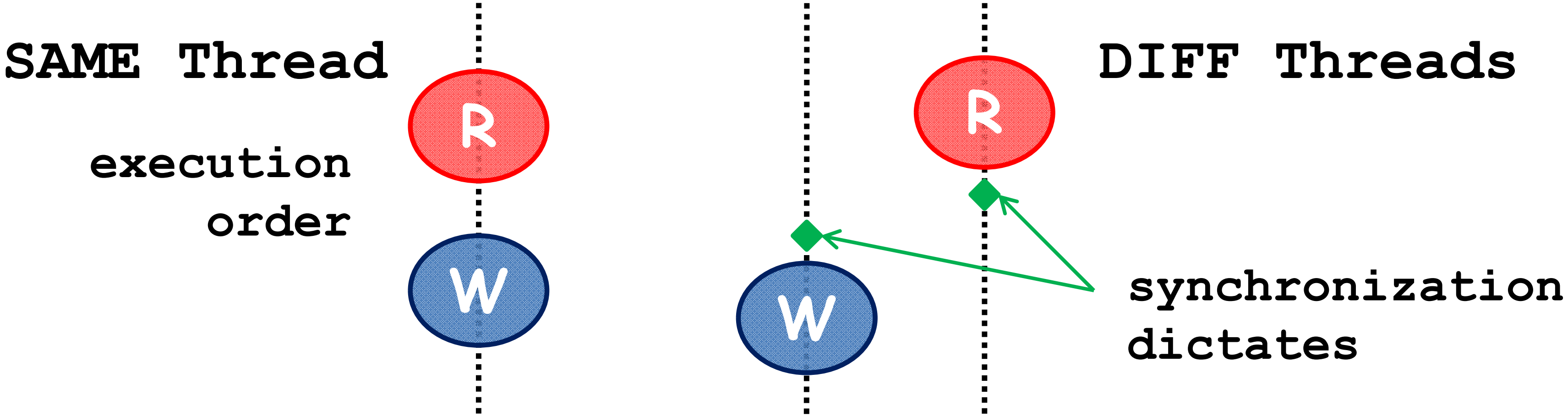
Rules in HB Relation

For the events A and B,

HB1: On the same sequential thread,
A→B if A executes before B.

HB2: On the different threads,
A→B if there is a synchronization that dictates A precedes B.

HB3: If A→B and B→C then A→C



SAME Thread

execution order

DIFF Threads

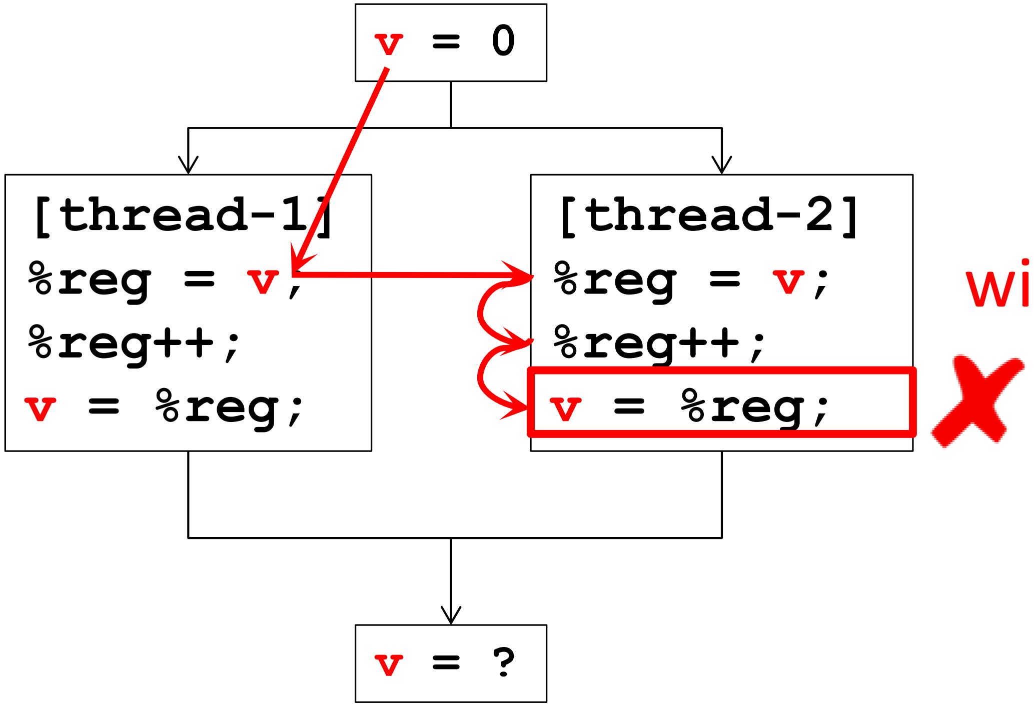
synchronization dictates

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



```
v = 0
```

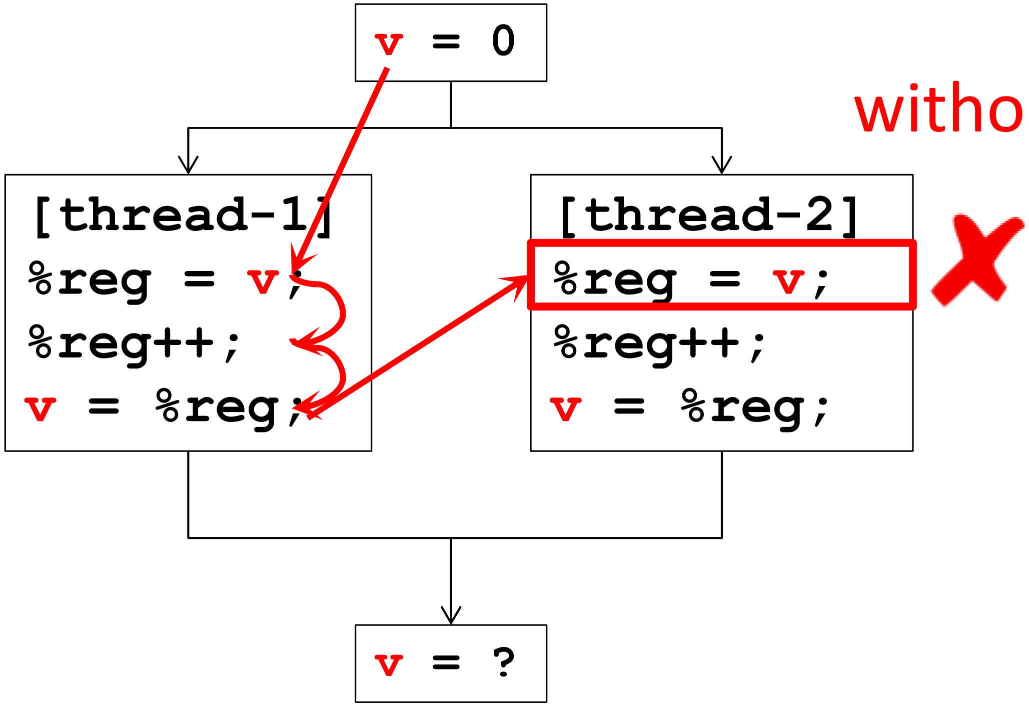
```
[thread-1]
%reg = v;
%reg++;
v = %reg;
```

```
[thread-2]
%reg = v;
%reg++;
v = %reg;
```

```
v = ?
```

without HB relation !!

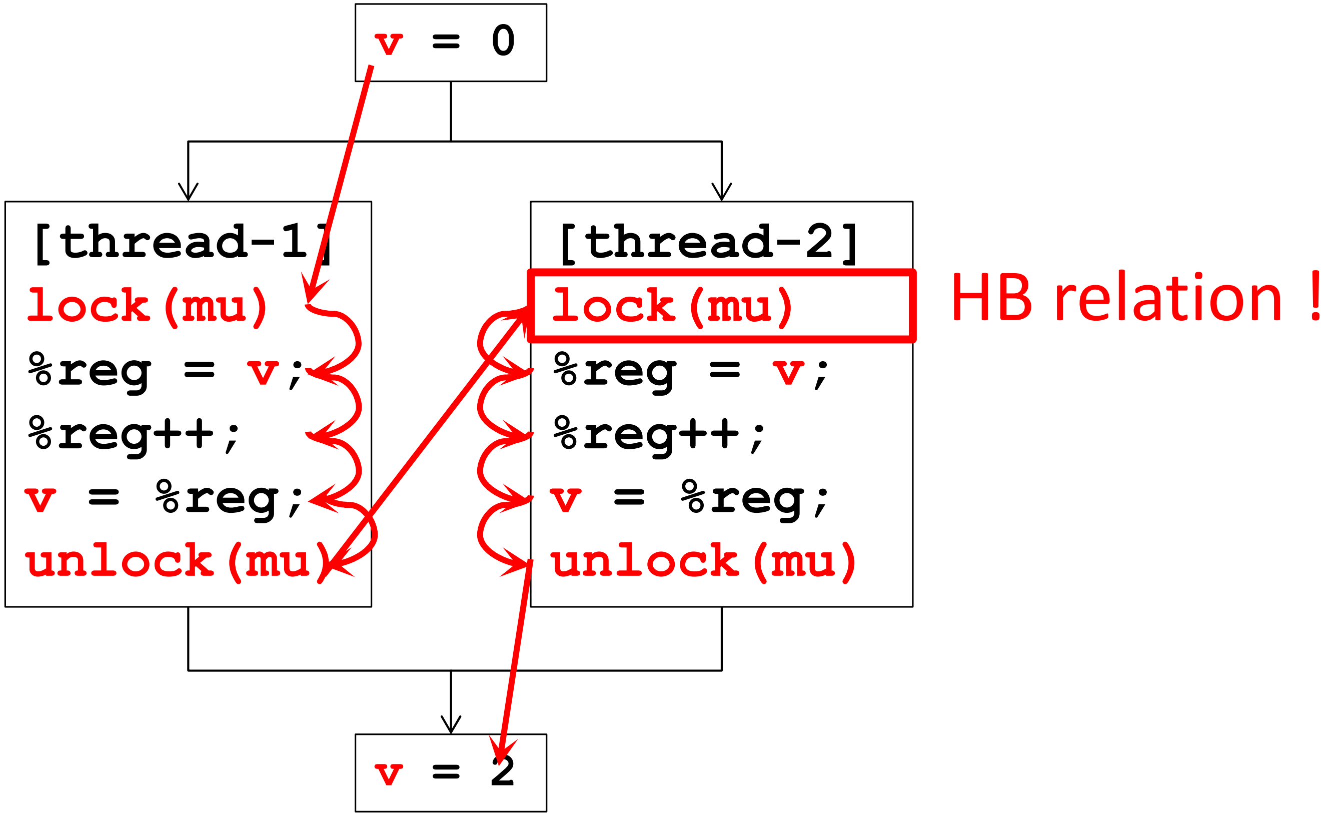
Example



```
v = 0

[thread-1]
%reg = v;
%reg++;
v = %reg;

[thread-2]
%reg = v;
%reg++;
v = %reg;

v = ?
```

without HB relation !!

Example



```
v = 0
```

```
[thread-1]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)
```

```
[thread-2]
lock(mu)
%reg = v;
%reg++;
v = %reg;
unlock(mu)
```

HB relation !

```
v = 2
```

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 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) ∩ locks_held(t)

   if C(v)={ }, then issue a warning

Example

Programs	locks_held(t)	C(v)
	{ }	{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);		
	{mu2}	
v = v + 1;		{} Warning!!
unlock(mu2);		

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

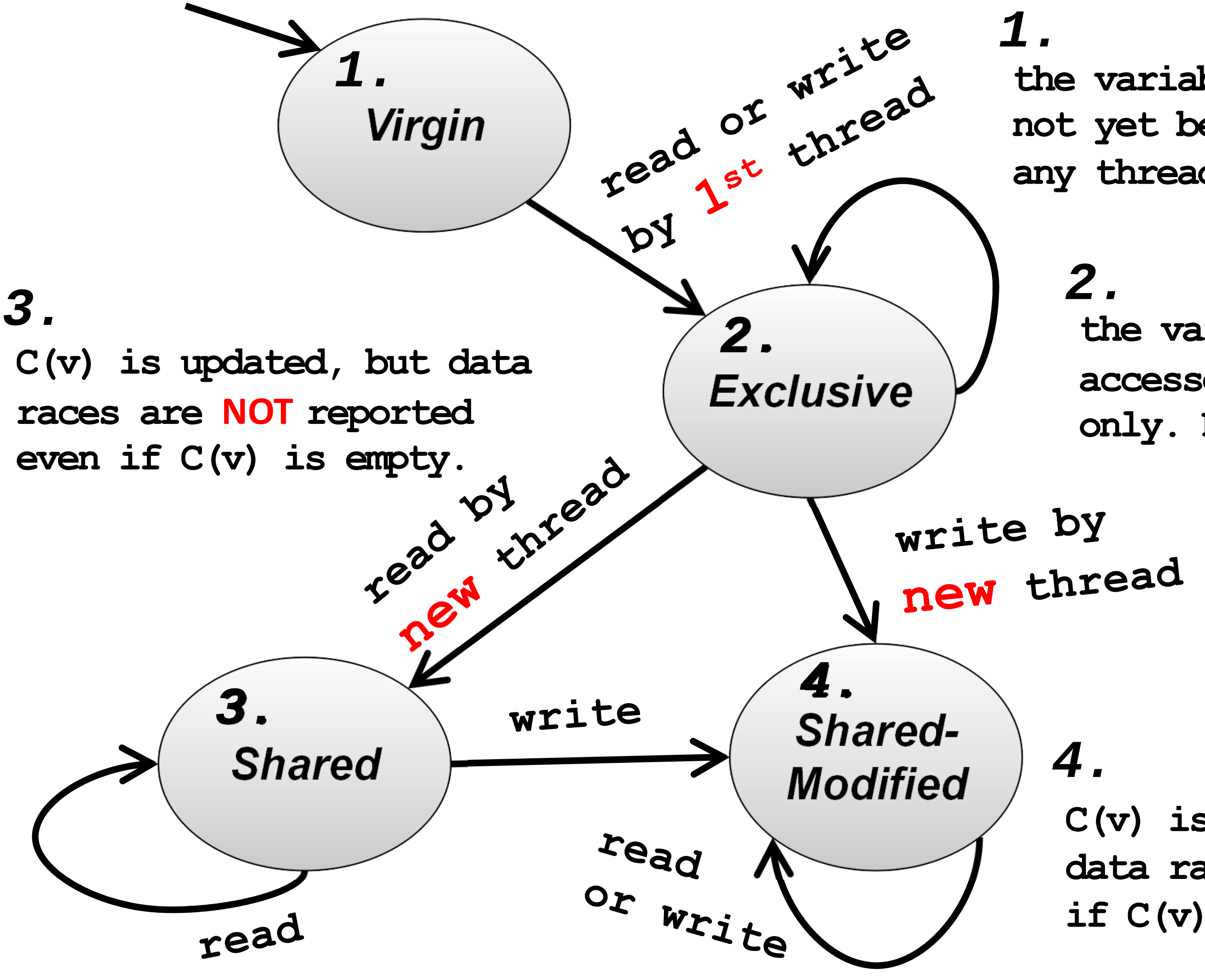
#3 Read-Write Lock

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



1.
the variable is NEW and has not yet been referenced by any thread.

2.
the variable has been accessed by ONE thread only. Not update C(v)

3.
C(v) is updated, but data races are NOT reported even if C(v) is empty.

4.
C(v) is updated, and data races are reported if C(v) is empty.

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) ∩ locks_held(t)
  if Cr(v)={ }, then issue a warning
4.On each write to v by thread t,
  set Cw(v):= Cw(v) ∩ 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

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
PERSONNEL
search ID: mban1277
"Thanks for coming in. We'll get back to you as soon as we lower our expectations."

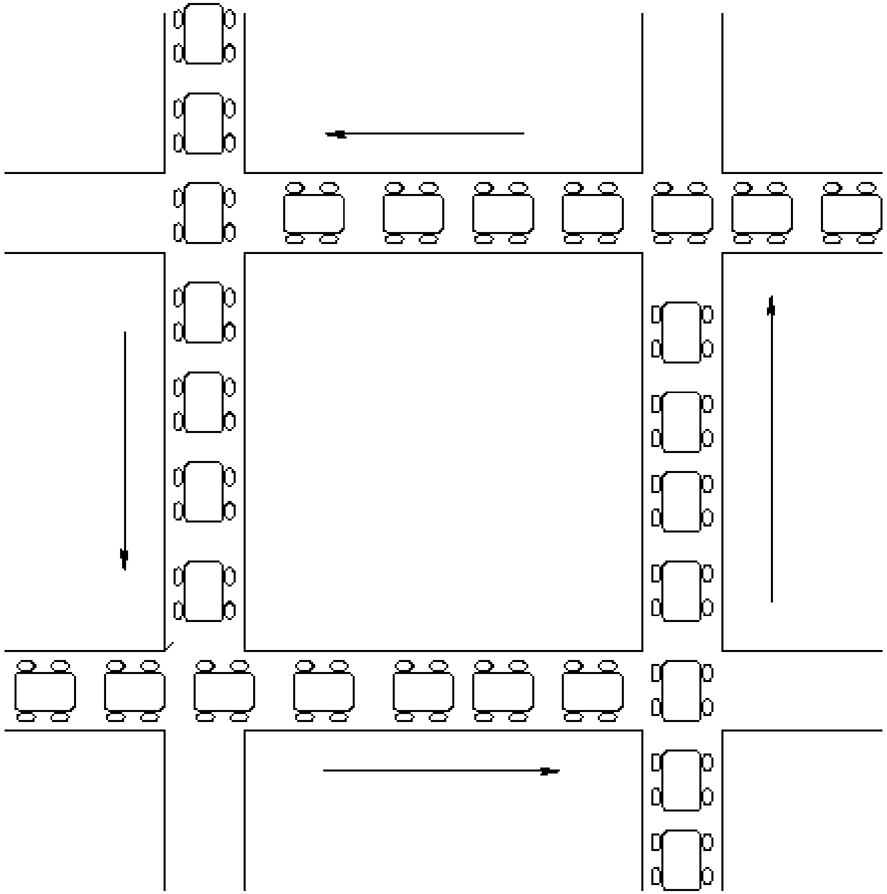
This guy has no experience!!!

How can I get experiences without a job!!!

Deadlock Definition

A set of processes is deadlocked when

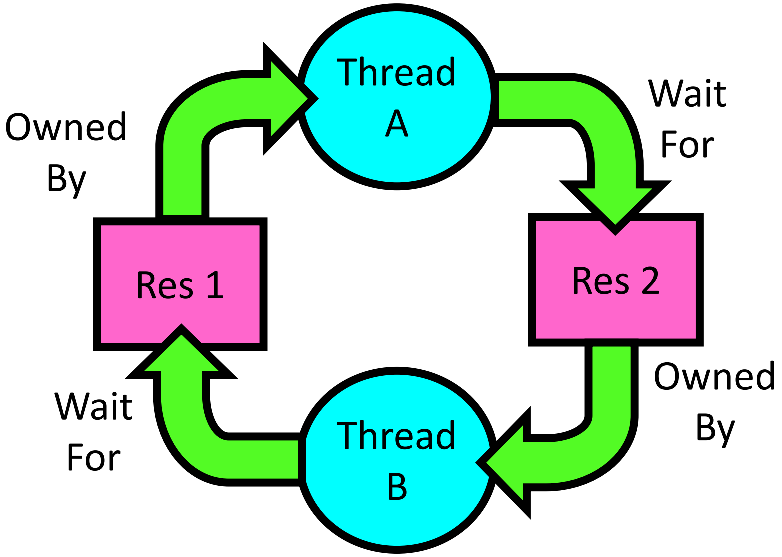
every process in the set is waiting for an event that can only be generated by some process in the set



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 {T1, …, Tn} of waiting threads
  - T1 is waiting for a resource that is held by T2
  - T2 is waiting for a resource that is held by T3
  - …
  - Tn is waiting for a resource that is held by T1

A Graph Theoretic Model of Deadlock

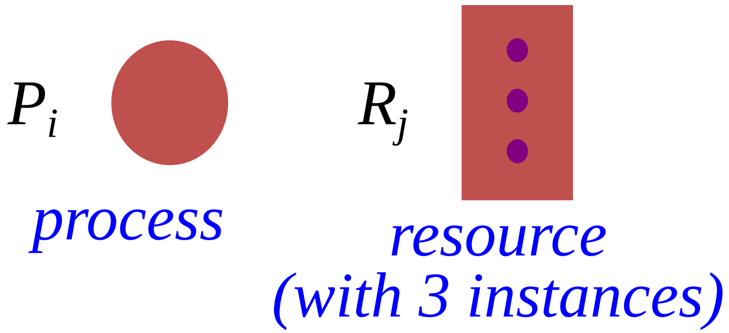
Basic components of any resource allocation problem

Processes and resources

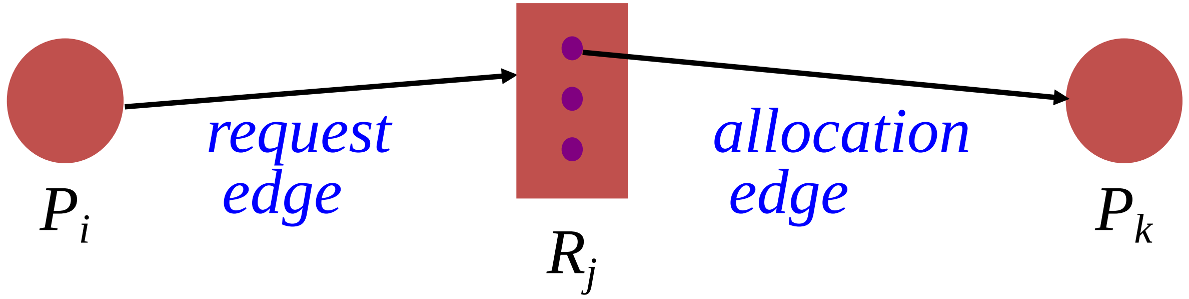
Model the state of a computer system as a directed graph (called resource allocation graph, or RAG)

$G = (V, E)$

V = the set of vertices = $\{P_1, \ldots, P_n\} \cup \{R_1, \ldots, R_m\}$



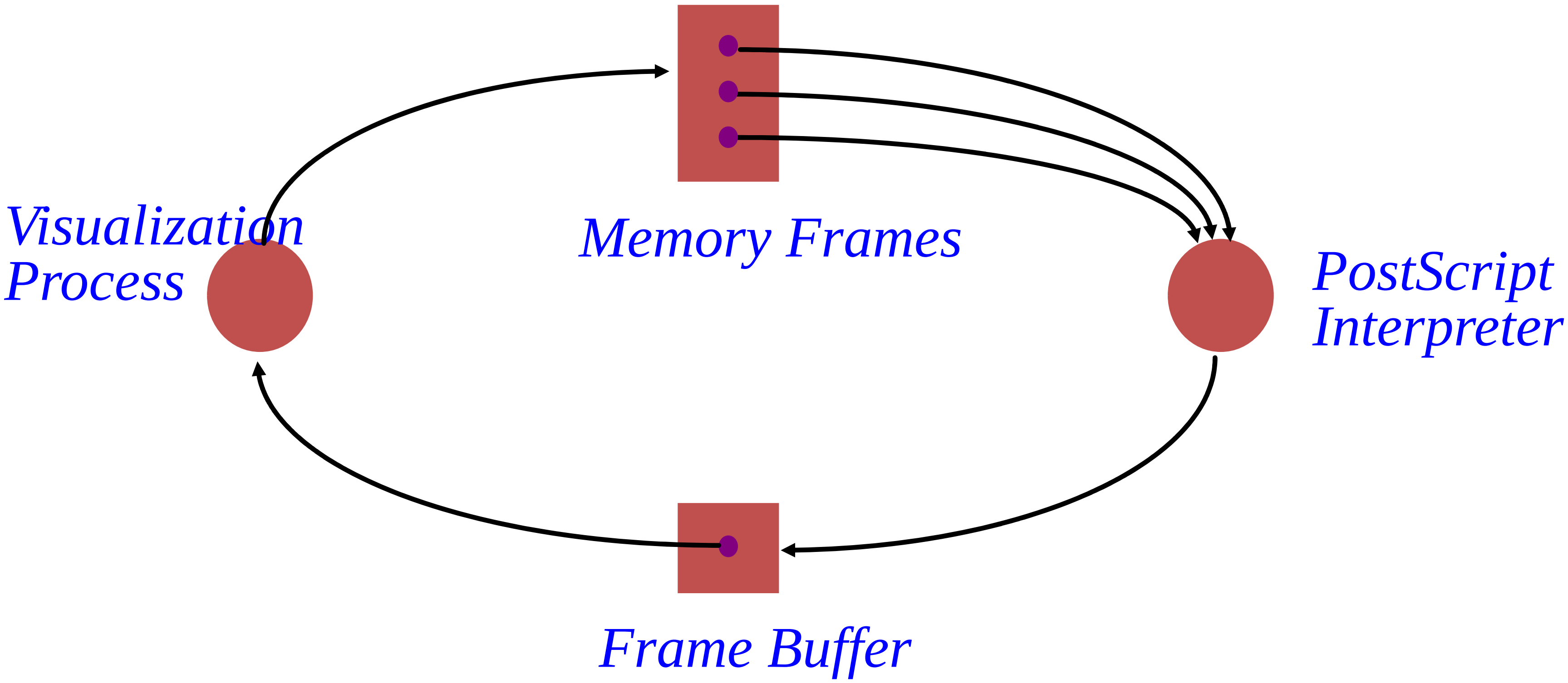
E = the set of edges = {edges from a resource to a process} $\cup$ {edges from a process to a resource}



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*} ∪ {*memory frames*, *frame buffer lock*}

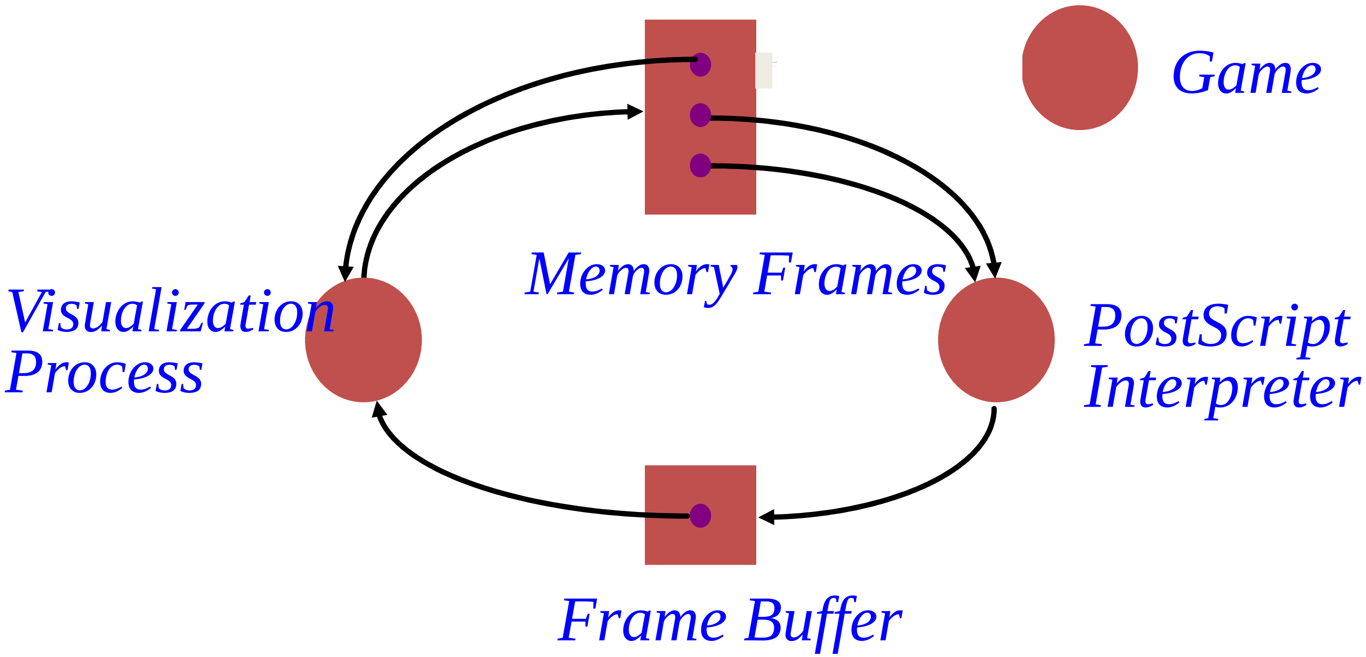


Resource Allocation Graphs & Deadlock

Theorem: If a resource allocation graph does not contain a cycle then no processes are deadlocked

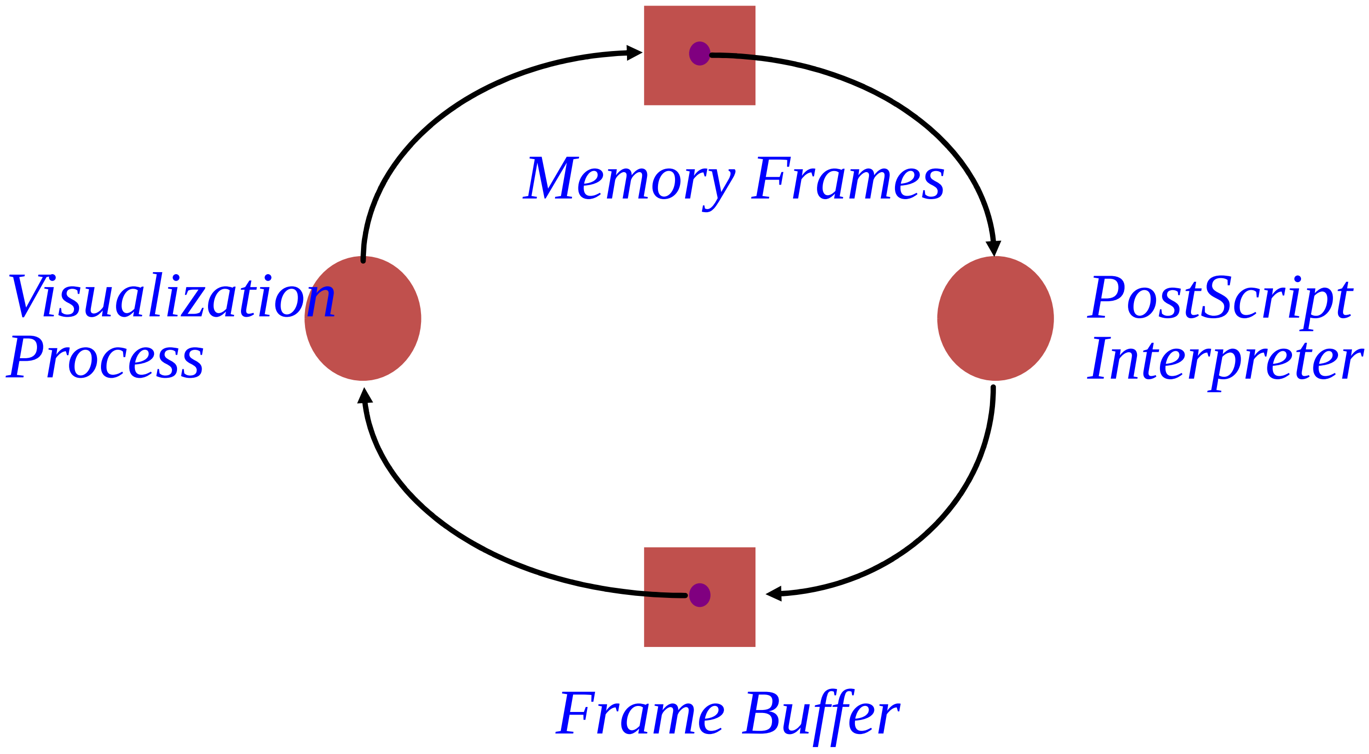
A cycle in a RAG is a necessary condition for deadlock

Is the existence of a cycle a sufficient condition?



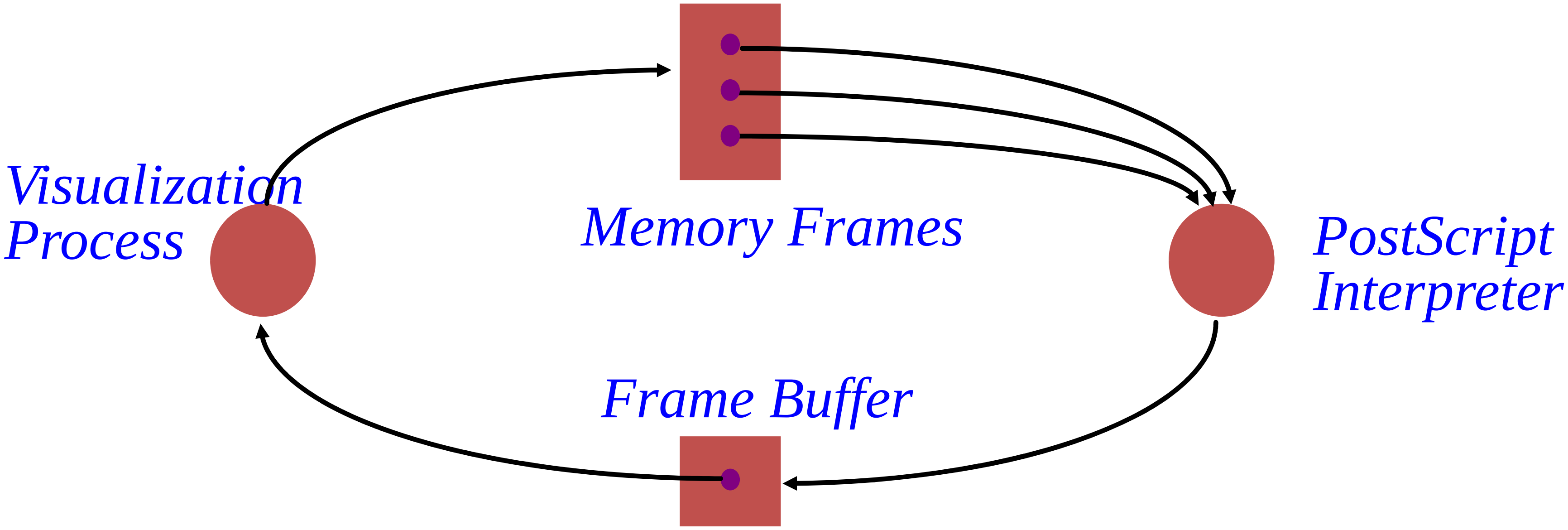
Single Resource RAG & Deadlocks

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

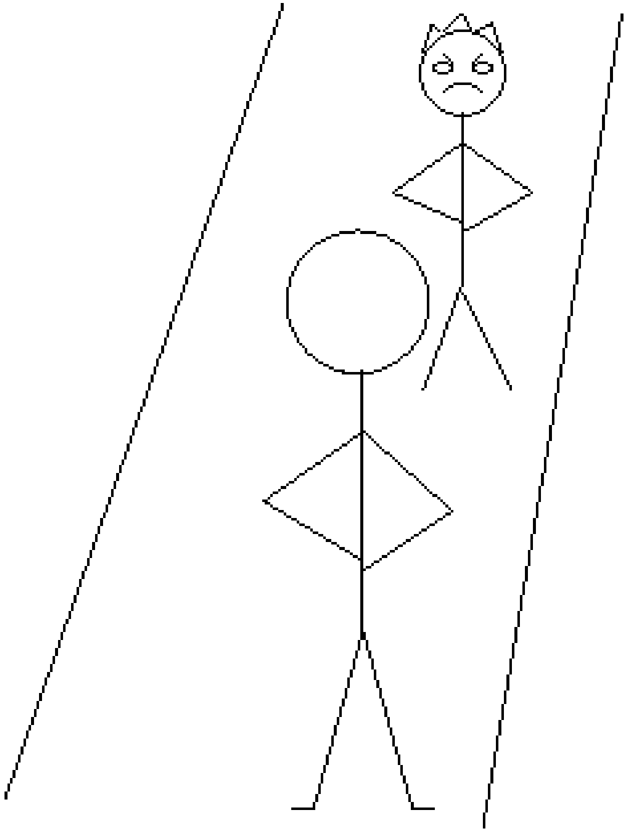
- Abort all deadlocked processes & reclaim their resources?
- Abort one process at a time until all cycles in the RAG are eliminated?
  - Start from low priority process?
  - Start from processes with most allocation of resources?

LiveLock

Similar to a deadlock

The states of the processes constantly change with regard to one another

none progressing

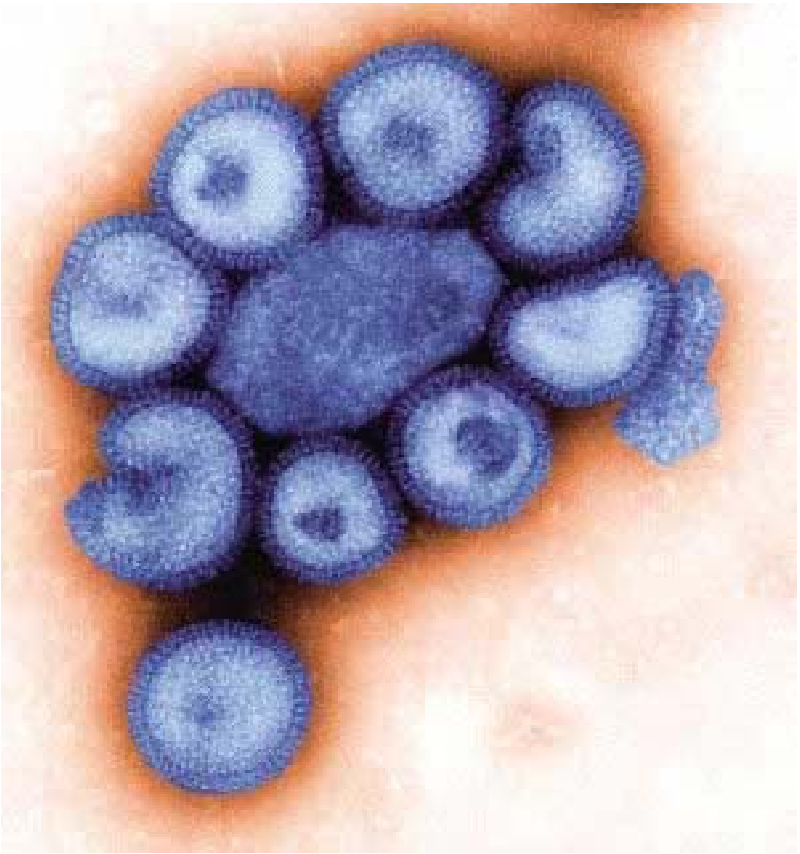


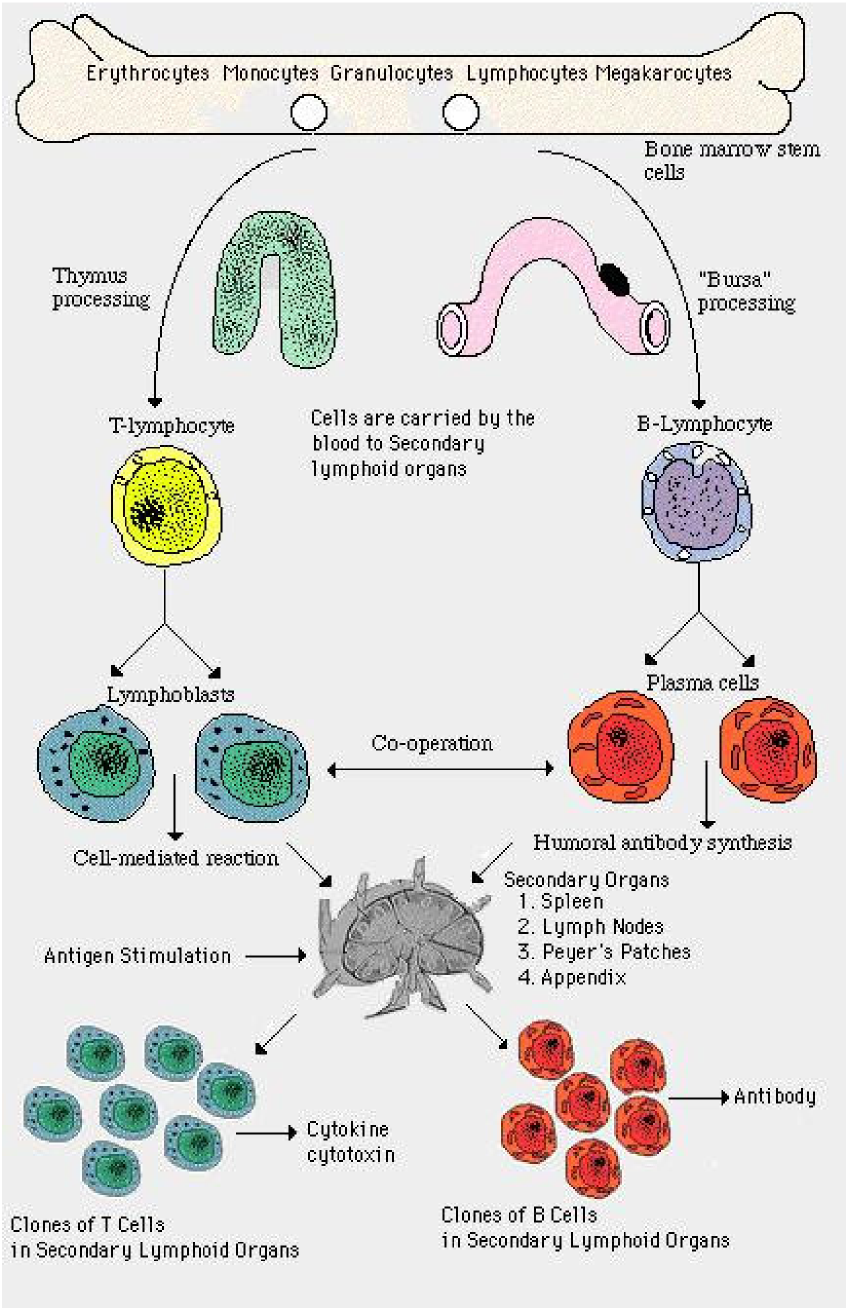
Deadlock Pitfalls in Reality

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.





Erythrocytes
Monocytes
Granulocytes
Lymphocytes
Megakaryocytes
Bone marrow stem cells
Thymus processing
"Bursa" processing
T-lymphocyte
Cells are carried by the blood to Secondary lymphoid organs
B-Lymphocyte
Lymphoblasts
Plasma cells
Co-operation
Cell-mediated reaction
Humoral antibody synthesis
Secondary Organs
1. Spleen
2. Lymph Nodes
3. Peyer's Patches
4. Appendix
Antigen Stimulation
Antibody
Cytokine cytotoxin
Clones of T Cells in Secondary Lymphoid Organs
Clones of B Cells in Secondary Lymphoid Organs

Dimmunix - Teaser

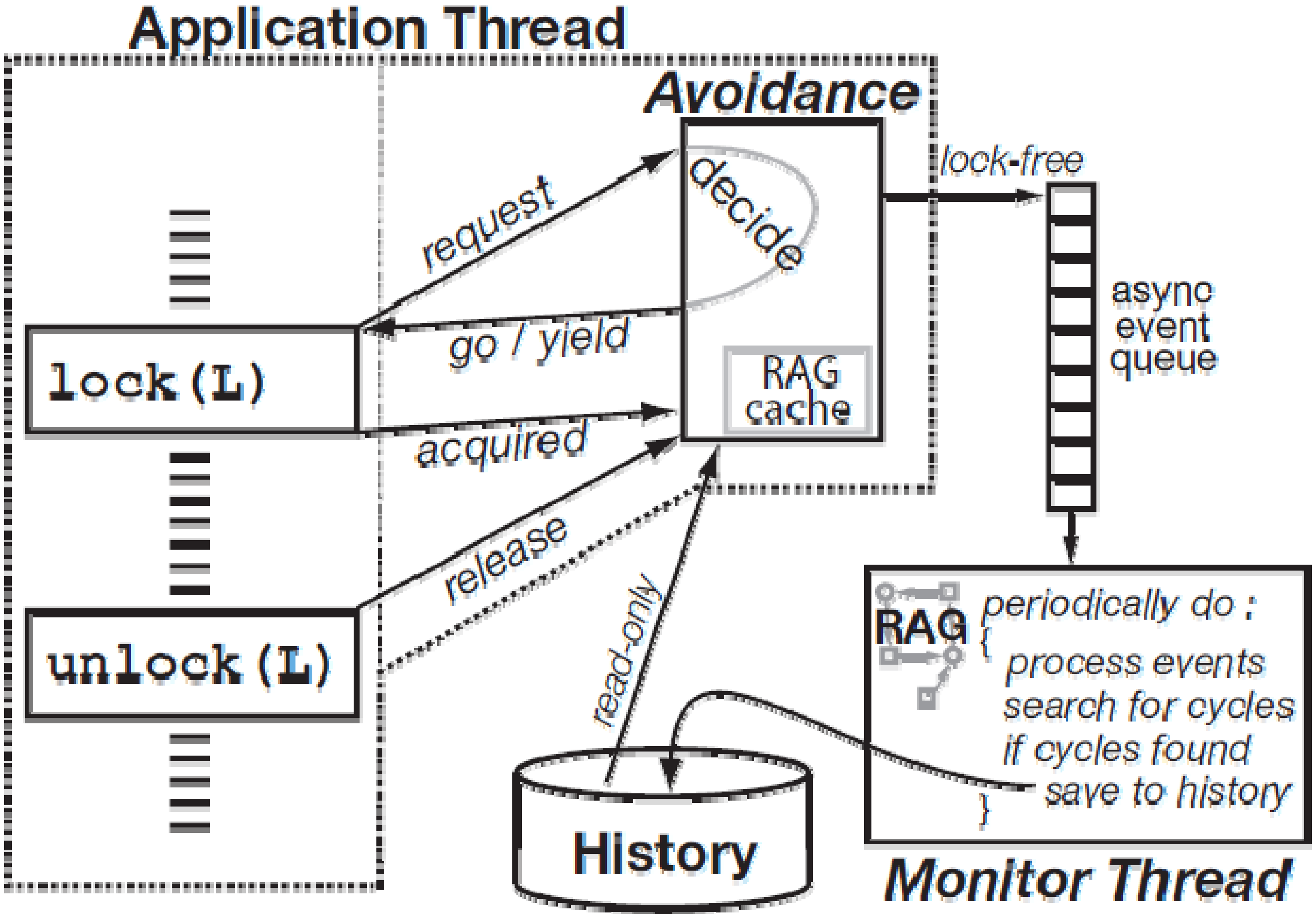
So, it is often hard to handle a deadlock. But what if…

- *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 it's *signature* and subsequently avoids entering the same pattern.

Dimmunix Architecture

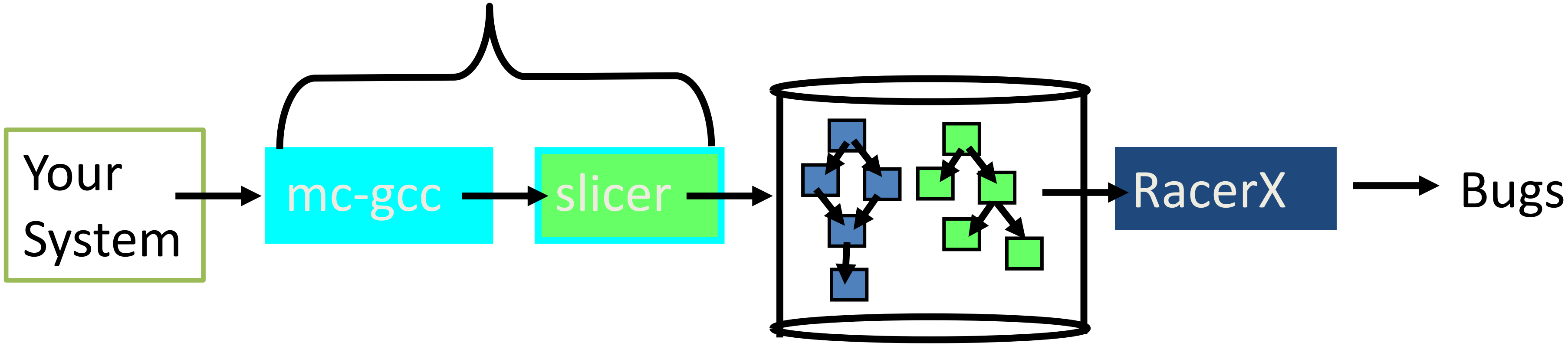


RaceX: Static LockSet Analysis for DeadLock

Slides from Dawson Engler@Stanford

The RacerX experience

- How to use:
  - List locking functions & entry points. Small:
    - Linux: 18 + 31, FreeBSD: 30 + 36, System X: 50 + 52
  - Emit trees from source code (2x cost of compile)



  - Run RacerX over emitted trees
    - Links all trees into global control flow graph (CFG)
    - Checks for deadlocks & races
    - ~2-20 minutes for Linux.
  - Post-process to rank errors (most of IQ spent here)
  - Inspect

Lockset analysis

- Lockset: set of locks currently held [Eraser]
  - For each root, do a flow-sensitive, inter-procedural DFS traversal computing lockset at each statement

```
initial   ➔ lockset = { }
lock(l)    ➔ lockset = lockset U { l }
unlock(l) ➔ lockset = lockset - { l }
```

  - Speed: If stmt s was visited before with lockset ls, stop.

- Inter-procedural:
  - Routine can exit with multiple locksets: resume DFS w/ each after callsite.
  - Record <in-ls, {out-ls}> in fn summary. If ls in summary, grab cached out-ls’ s and skip fn body.

Lockset

```
   connect() {
→    lock(a);          { a }
→    open_conn();      { a }
     send();
   }
                           { a }            summary:
                                              { a } ➔?
                           open_conn() {
                       →     if (x)                  { a }
                       →       lock(b);            { a, b }
                       →     else                    { a }
                       →       lock(c);            { a, c }

                       →   }                   { a, b }  { a, c }
```

Lockset

```
connect() {
   lock(a);           { a }
   open_conn();       { a }
   send();         { a, b }, {a, c}
}
```

summary:
{ a } ➔ { a, b }, {a, c}

{ a, b }, {a, c}

```
open_conn() {
   if (x)                { a }
      lock(b);         { a, b }
   else                  { a }
      lock(c);         { a, c }

}                   { a, b }, {a, c}
```

Big picture: Deadlock detection

- Pass 1: constraint extraction
  - emit 1-level locking dependencies during lockset analysis

lock(a);
lock(b); → "a→b"

lock(b);
lock(a); → "b→a"

- Pass 2: constraint solving
  - Compute transitive closure & flag cycles.
  - "a→b→a" : T1 acquires a, T2 acquires b, boom.
- Ranking:
  - Global locks over local
  - Depth of callchain & number of conditionals (less better)
  - Number of threads involved (fewer MUCH better)

Simplest deadlock example

```
// 2.5.62/drivers/char/rtc.c
int rtc_register(rtc_task_t *task) {
  spin_lock_irq(&rtc_lock);
  //...
  spin_lock(&rtc_task_lock);
  if (rtc_callback) {
     spin_unlock(&rtc_task_lock);
     spin_unlock_irq(&rtc_lock);
```

```
//2.5.62/drivers/char/rtc.c
rtc_unregister(rtc_task_t *task)
{
  spin_lock_irq(&rtc_task_lock);
  //...
  spin_lock(&rtc_lock);
```

– Constraint extraction emits "rtc_lock→rtc_task_lock" and "rtc_task_lock→rtc_lock"

– Constraint solving flags cycle: T1 acquires rtc_lock, T2 acquires rtc_task_lock. Boom.

– Ranked high: only two threads, global locks, local error.

False positive trouble

- Most FPs from bogus locks in lockset
  - Typically caused by mishandled data dependencies
- Oversimplified typical example
  - Naïve analysis will think four paths rather than two, including false one that holds lock a at line 5.

```
1: if(x)                {}
2:       lock(a);      {a}
3: if(x)                {a}
4:       unlock(a);
5: lock(b);             {a}    “a→b”
```

  - Inter-procedural analysis makes this much worse.
  - Could add path-sensitivity, but undecidable in general

Unlockset analysis

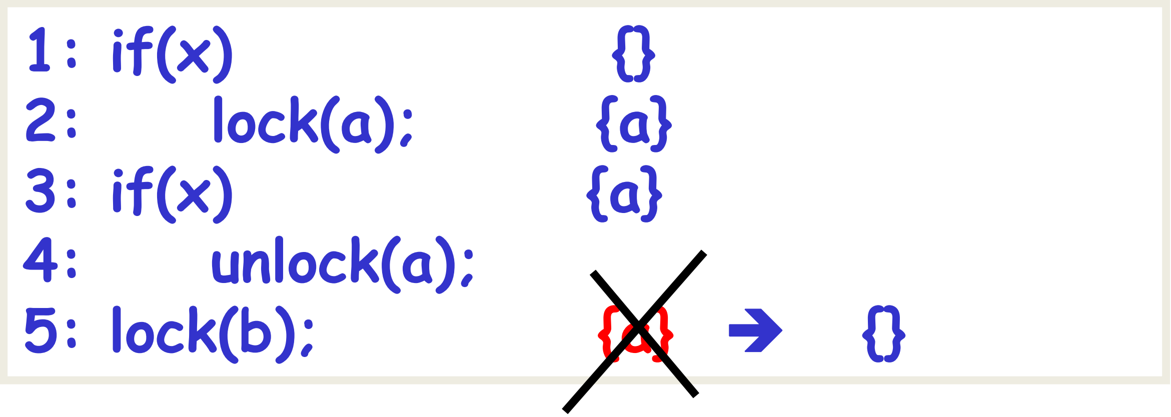
- Observations:
  - In practice, all false positives due to the A in “A➔B”, most because A goes “too far”
  - We had unconsciously adopted pattern of inspecting errors where there was an explicit unlock of “A” after “A➔B” since that strongly suggested “A” was held.

```
// 2.5.62/drivers/char/rtc.c
rtc_register(rtc_task_t *task) {
  spin_lock_irq(&rtc_lock);
  //...
  spin_lock(&rtc_task_lock);
  if (rtc_callback) {
      spin_unlock(&rtc_task_lock);
      spin_unlock_irq(&rtc_lock);
```

rtc_lock➔rtc_task_lock

Unlockset analysis

- At statement S remove any lock L from lockset if there exists no successor statement S’ reachable from S that contains an unlock of L.



```
if(x)                 {}
      lock(a);        {a}
if(x)                 {a}
      unlock(a);
lock(b);              {a}  →   {}
```

- Key: lockset holds exactly those locks the analysis can handle. Scales with analysis sophistication.

Unlockset implementation sketch

- Essentially compute reaching definitions
  - Run lockset analysis in reverse from leaves to roots
  - Unlockset holds all locks that will be released

```
initial    ➔ unlockset = { }
lock(l)    ➔ unlockset = unlockset  - { l }
unlock(l) ➔ unlockset = unlockset U { l }
s.unlockset = s.unlockset U unlockset
```

  - During lockset analysis:

```
lockset = intersect(s.unlockset, lockset);
```

- Main complication: function calls.
  - Different locks released after different callsites. Don't want to mix these up (context sensitivity)

Deadlock results

System	Confirmed	Unconfirmed	False
System X	2	3	7
Linux 2.5.62	4	8	6
FreeBSD	2	3	6
Total	8	14	19

- A bit surprised at the low bug counts
  - Main reason seems to be not that many locks held simultaneously
  - < 1000 unique constraints, only so many chances for error.

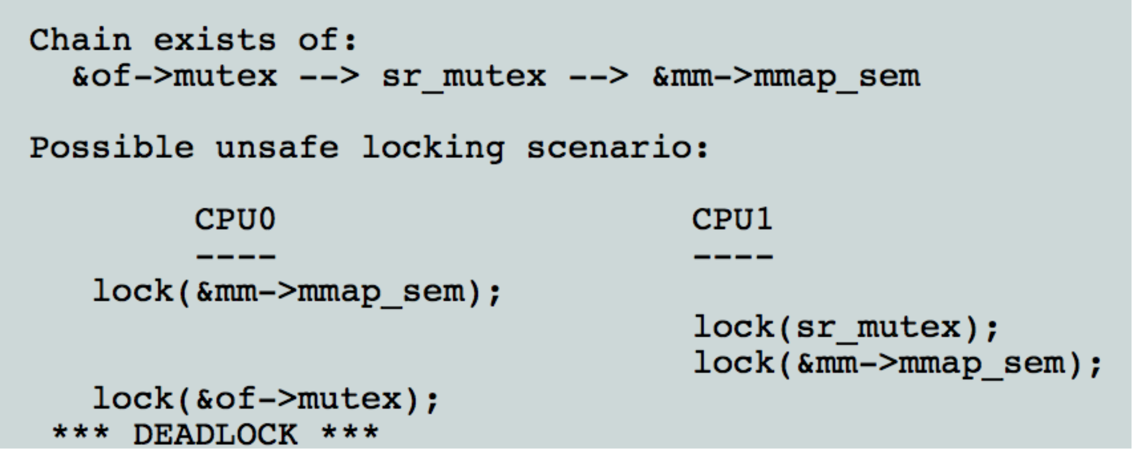
The most surprising error

```
// Entered holding scsiLock
int FindHandle(int handleID) {
   prevIRQL = SP_LockIRQ(&handleArrayLock, …);
   Validate(handle);
   ...
int Validate(handle) {
    ASSERT(SP_IsLocked(&scsiLock));
    while (adapter->openInProgress) {
       CpuSched_Wait(&adapter->openInProgress,
                 CPUSCHED_WAIT_SCSI, &scsiLock);
      SP_Lock(&scsiLock);
```

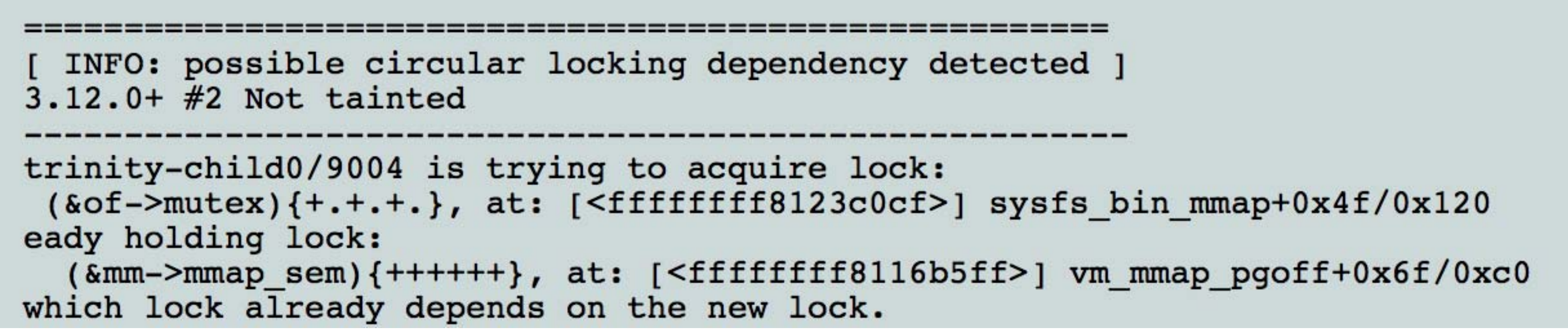
- T1 enters FindHandle with scsiLock, calls Validate, calls CpuSched_wait (rel scsiLock, sleep w/ handleArrayLock)
- T2 acquires scsiLock and calls FindHandle. Boom.

Lock Analysis in Real World

- Lockdep analysis
  - https://www.kernel.org/doc/Documentation/lockdep-design.txt



```
Chain exists of:
  &of->mutex --> sr_mutex --> &mm->mmap_sem

Possible unsafe locking scenario:

        CPU0                    CPU1
        ----                    ----
   lock(&mm->mmap_sem);
                                lock(sr_mutex);
                                lock(&mm->mmap_sem);
   lock(&of->mutex);
 *** DEADLOCK ***
```



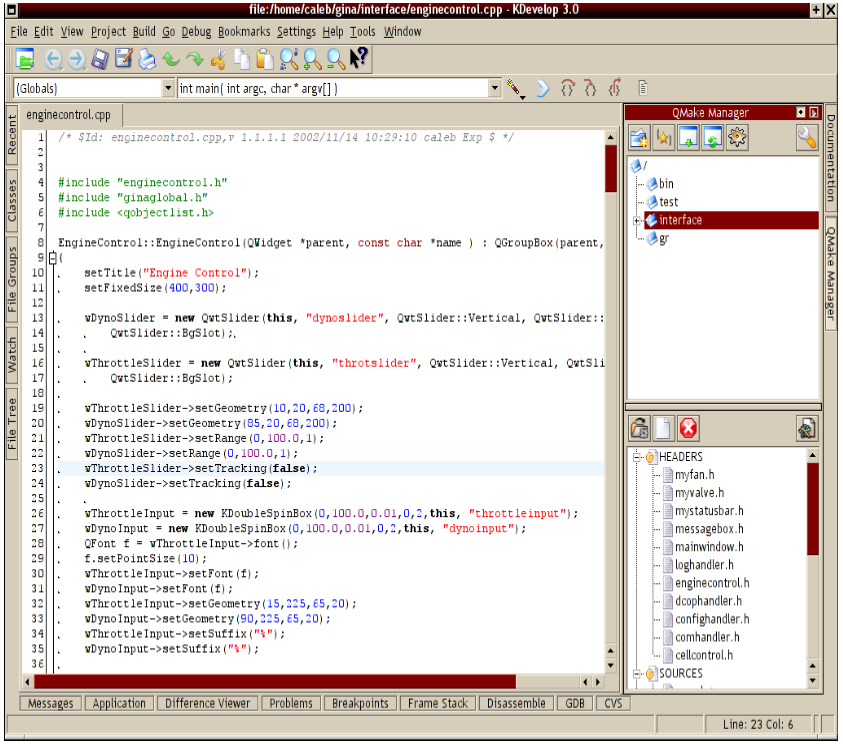
```
======================================================
[ INFO: possible circular locking dependency detected ]
3.12.0+ #2 Not tainted
-------------------------------------------------------
trinity-child0/9004 is trying to acquire lock:
 (&of->mutex){+.+.+.}, at: [<ffffffff8123c0cf>] sysfs_bin_mmap+0x4f/0x120
eady holding lock:
  (&mm->mmap_sem){++++++}, at: [<ffffffff8116b5ff>] vm_mmap_pgoff+0x6f/0xc0
which lock already depends on the new lock.
```

Valgrind: a useful tool for program debugging/analysis

What is Valgrind

- Linux, x86, program analyzer
- Advantages
  - powerful
  - no recompiles, just binaries
  - any language (!)
  - extensible, configurable
  - free, GPL'd goodness

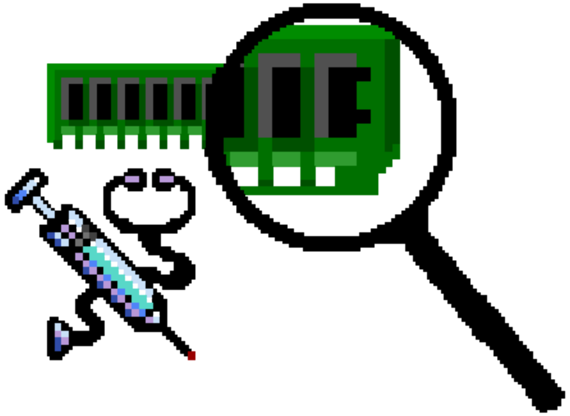
- Speed slowdown

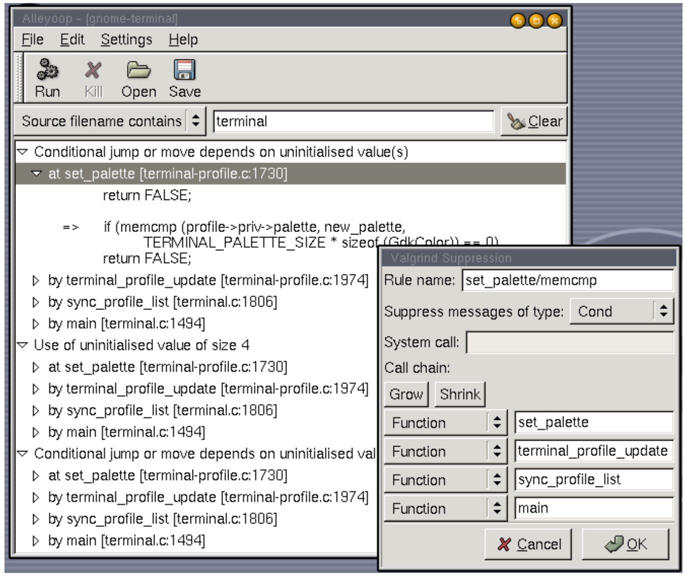


```
/* $Id: enginecontrol.cpp,v 1.1.1.1 2002/11/14 10:29:10 caleb Exp $ */


#include "enginecontrol.h"
#include "ginaglobal.h"
#include <qobjectlist.h>

EngineControl::EngineControl(QWidget *parent, const char *name ) : QGroupBox(parent,
{
    setTitle("Engine Control");
    setFixedSize(400,300);

    wDynoSlider = new QwtSlider(this, "dynoslider", QwtSlider::Vertical, QwtSlider::
        QwtSlider::BgSlot);


    wThrottleSlider = new QwtSlider(this, "throtslider", QwtSlider::Vertical, QwtSli
        QwtSlider::BgSlot);

    wThrottleSlider->setGeometry(10,20,68,200);
    wDynoSlider->setGeometry(85,20,68,200);
    wThrottleSlider->setRange(0,100.0,1);
    wDynoSlider->setRange(0,100.0,1);
    wThrottleSlider->setTracking(false);
    wDynoSlider->setTracking(false);

    wThrottleInput = new KDoubleSpinBox(0,100.0,0.01,0,2,this, "throttleinput");
    wDynoInput = new KDoubleSpinBox(0,100.0,0.01,0,2,this, "dynoinput");
    QFont f = wThrottleInput->font();
    f.setPointSize(10);
    wThrottleInput->setFont(f);
    wDynoInput->setFont(f);
    wThrottleInput->setGeometry(15,225,65,20);
    wDynoInput->setGeometry(90,225,65,20);
    wThrottleInput->setSuffix("%");
    wDynoInput->setSuffix("%");

```

What are the existing skins

- MemCheck
  - Heavy duty memory checker
- AddrCheck
  - Faster, cheaper
- Cachegrind
  - Caching info
- Helgrind
  - Thread semantics
- Others





Source filename contains: terminal

```
▽ Conditional jump or move depends on uninitialised value(s)
  ▽ at set_palette [terminal-profile.c:1730]
              return FALSE;

      =>      if (memcmp (profile->priv->palette, new_palette,
                   TERMINAL_PALETTE_SIZE * sizeof (GdkColor)) == 0)
              return FALSE;
  ▷ by terminal_profile_update [terminal-profile.c:1974]
  ▷ by sync_profile_list [terminal.c:1806]
  ▷ by main [terminal.c:1494]
▽ Use of uninitialised value of size 4
  ▷ at set_palette [terminal-profile.c:1730]
  ▷ by terminal_profile_update [terminal-profile.c:1974]
  ▷ by sync_profile_list [terminal.c:1806]
  ▷ by main [terminal.c:1494]
▽ Conditional jump or move depends on uninitialised val
  ▷ at set_palette [terminal-profile.c:1730]
  ▷ by terminal_profile_update [terminal-profile.c:1974]
  ▷ by sync_profile_list [terminal.c:1806]
  ▷ by main [terminal.c:1494]
```

Rule name: set_palette/memcmp

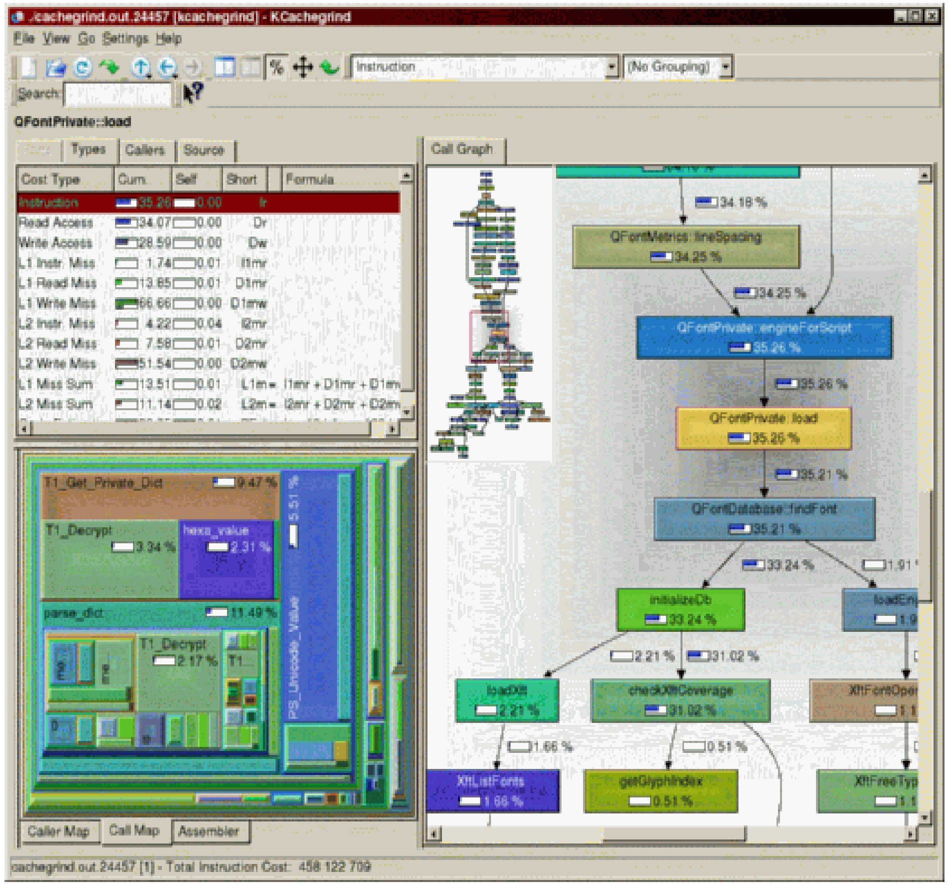
Suppress messages of type: Cond

System call:

Call chain:

Function	set_palette
Function	terminal_profile_update
Function	sync_profile_list
Function	main

Other features



./cachegrind.out.24457 [kcachegrind] - KCachegrind

QFontPrivate::load

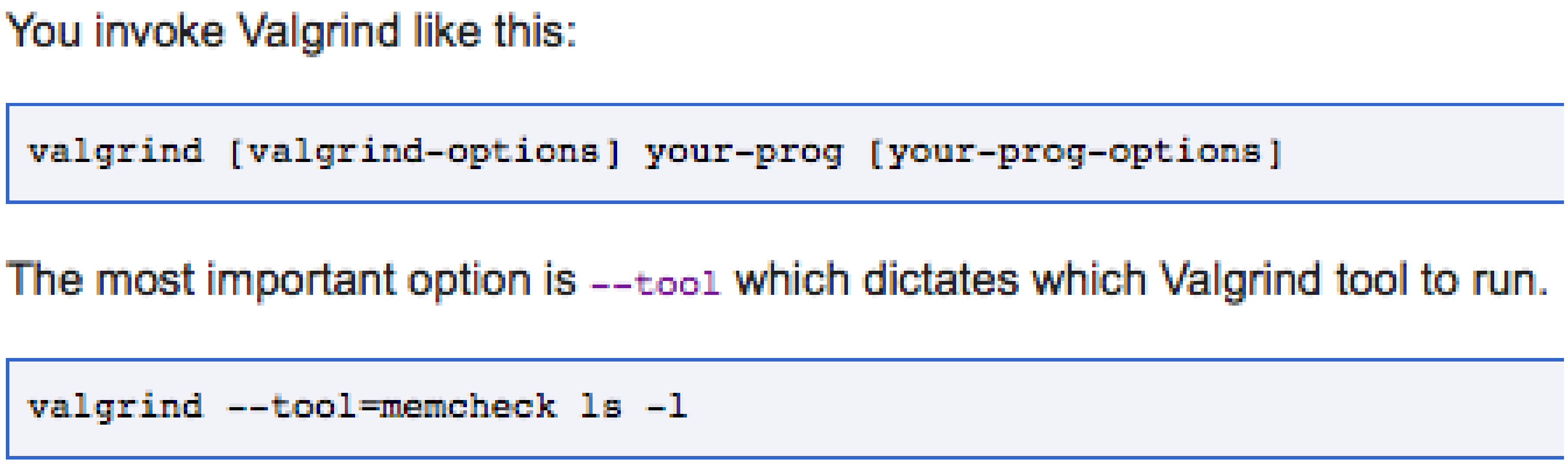
Cost Type	Cum.	Self	Short	Formula
Instruction	35.26	0.00	Ir	
Read Access	34.07	0.00	Dr	
Write Access	28.59	0.00	Dw	
L1 Instr. Miss	1.74	0.01	I1mr	
L1 Read Miss	13.85	0.01	D1mr	
L1 Write Miss	66.66	0.00	D1mw	
L2 Instr. Miss	4.22	0.04	I2mr	
L2 Read Miss	7.58	0.01	D2mr	
L2 Write Miss	51.54	0.00	D2mw	
L1 Miss Sum	13.51	0.01	L1m =	I1mr + D1mr + D1mw
L2 Miss Sum	11.14	0.02	L2m =	I2mr + D2mr + D2mw

cachegrind.out.24457 [1] - Total Instruction Cost: 458 122 709

- Callgraphs
- Multiple other patches
  - Open file descriptors
- Suppressions
  - Selective understanding
- Ignore System libs
  - Yes, bugs
- /dev/random

How to Use it

- Apt-get install valgrind



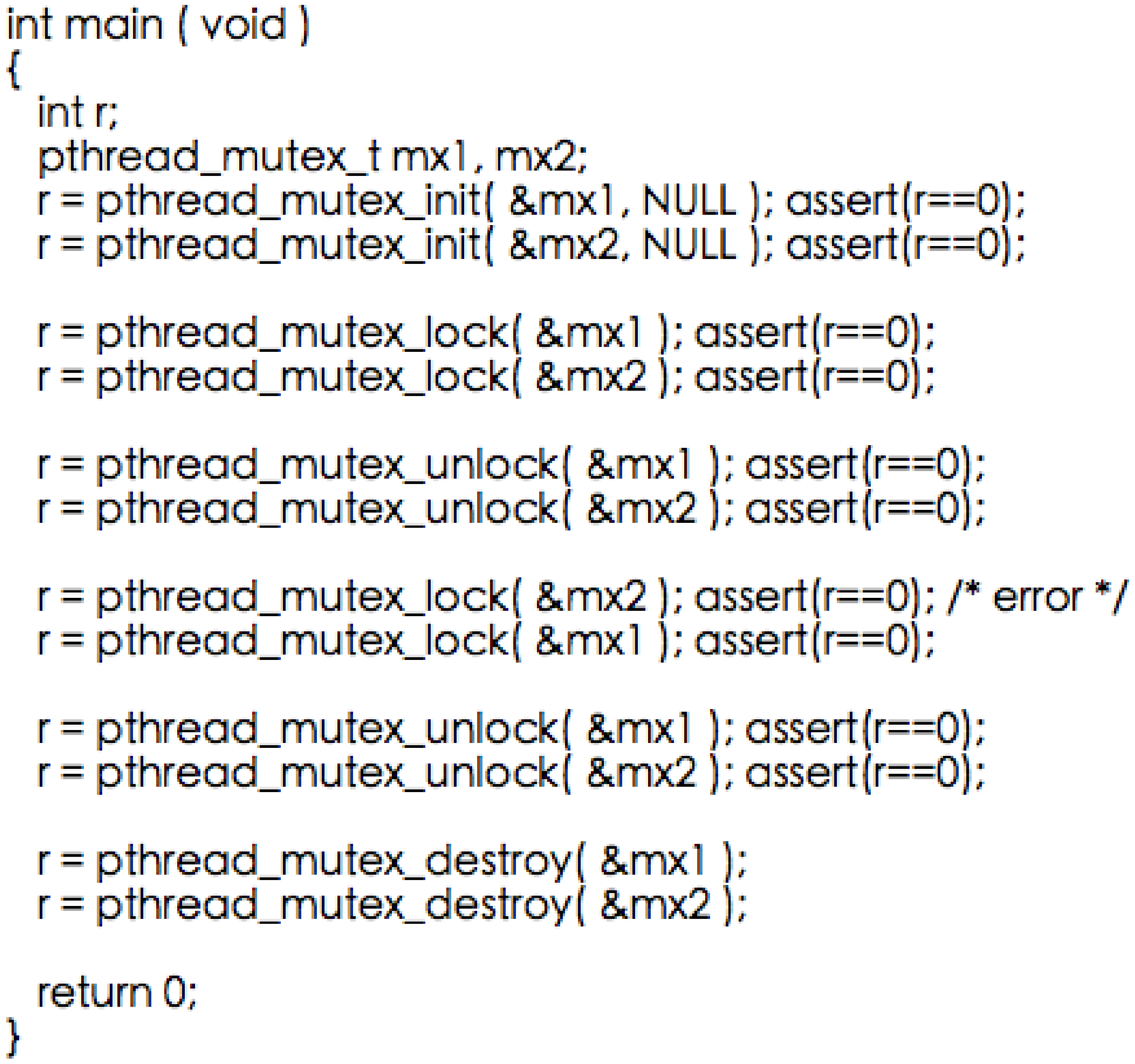
You invoke Valgrind like this:

```
valgrind [valgrind-options] your-prog [your-prog-options]
```

The most important option is `--tool` which dictates which Valgrind tool to run.

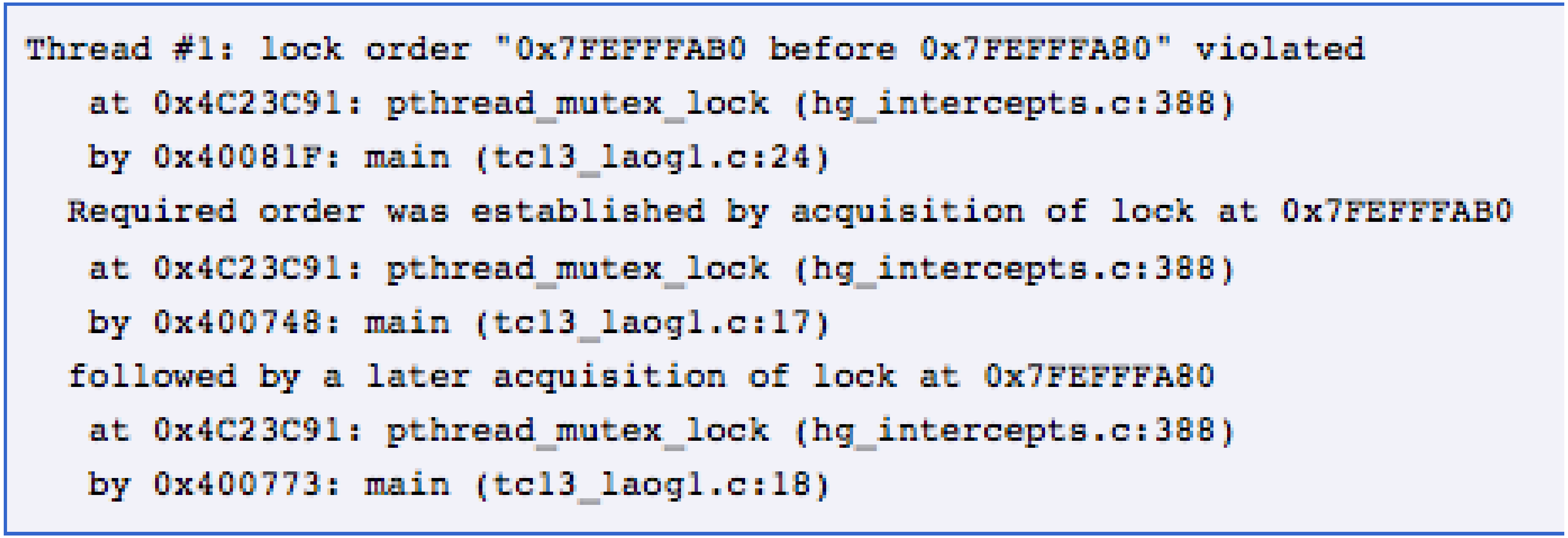
```
valgrind --tool=memcheck ls -l
```

Example code



```
int main ( void )
{
   int r;
   pthread_mutex_t mx1 , mx2;
   r = pthread_mutex_init( &mx1 , NULL ); assert(r==0);
   r = pthread_mutex_init( &mx2, NULL ); assert(r==0);

   r = pthread_mutex_lock( &mx1 ); assert(r==0);
   r = pthread_mutex_lock( &mx2 ); assert(r==0);

   r = pthread_mutex_unlock( &mx1 ); assert(r==0);
   r = pthread_mutex_unlock( &mx2 ); assert(r==0);

   r = pthread_mutex_lock( &mx2 ); assert(r==0); /* error */
   r = pthread_mutex_lock( &mx1 ); assert(r==0);

   r = pthread_mutex_unlock( &mx1 ); assert(r==0);
   r = pthread_mutex_unlock( &mx2 ); assert(r==0);

   r = pthread_mutex_destroy( &mx1 );
   r = pthread_mutex_destroy( &mx2 );

   return 0;
}
```

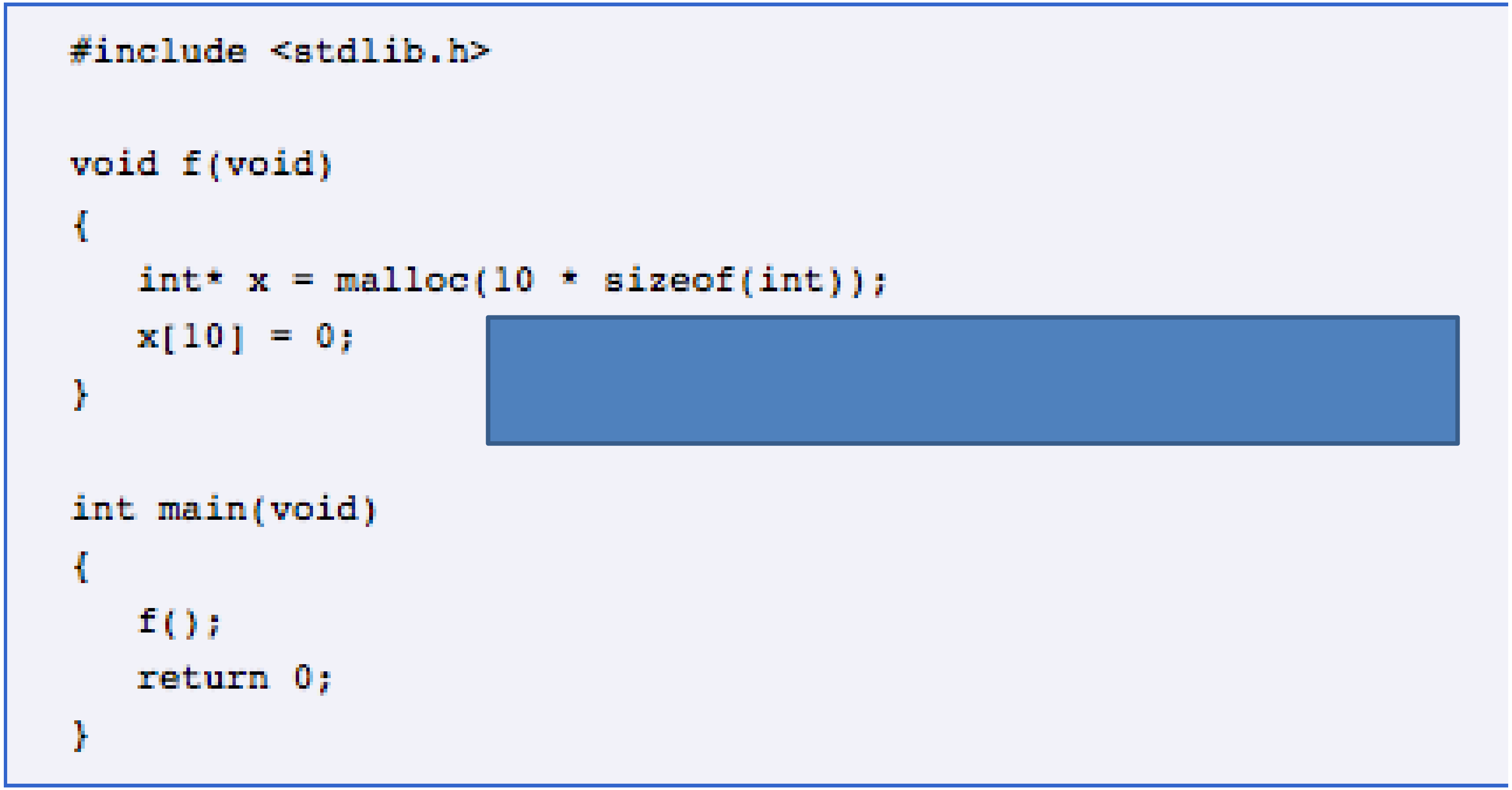
How Valgrind Detect it



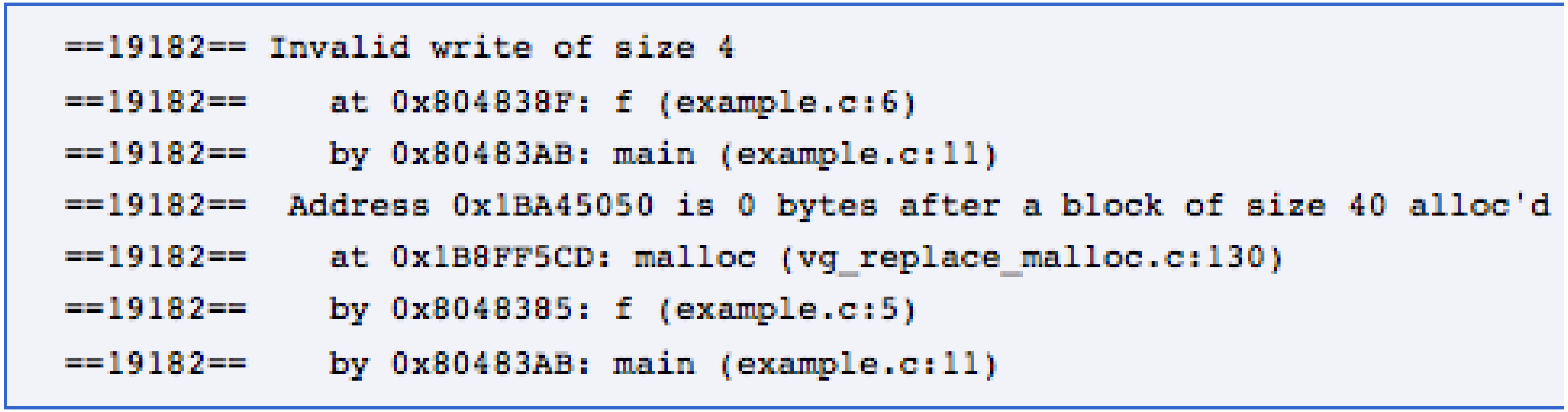
```
Thread #1: lock order "0x7FEFFFAB0 before 0x7FEFFFA80" violated
   at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
   by 0x40081F: main (tc13_laog1.c:24)
  Required order was established by acquisition of lock at 0x7FEFFFAB0
   at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
   by 0x400748: main (tc13_laog1.c:17)
  followed by a later acquisition of lock at 0x7FEFFFA80
   at 0x4C23C91: pthread_mutex_lock (hg_intercepts.c:388)
   by 0x400773: main (tc13_laog1.c:18)
```

MANY OTHER USEFUL FEATURES

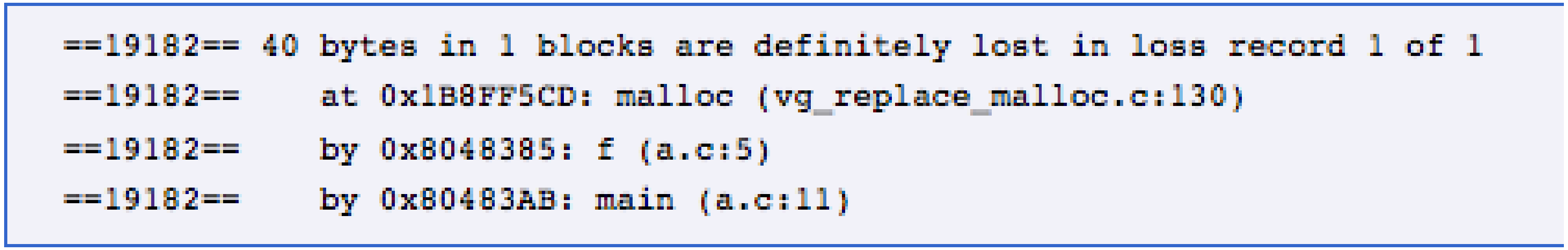
Example Uses of MemCheck



```
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```



```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```



```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

Cachegrind: a cache profiler

Cachegrind

- Cache behavior is crucial
  - L1 misses: ~10 cycles
  - L2 misses: ~200 cycles
- But difficult to predict
- Cachegrind gives three outputs:
  - Total hit/miss counts and ratios (I1, D1, L2)
  - Per-function hit/miss counts (sorted from most to least)
  - Per-line hit/miss counts (source code annotations)
- Source code annotations are the most useful
  - Most fine-grained data
  - Data that programmers can act on to speed up their programs

Sample output

```
--------------------------------------------------------------------------------
        Ir I1mr I2mr        Dr D1mr D2mr        Dw      D1mw      D2mw
--------------------------------------------------------------------------------
14,789,396  547  544 6,329,792  751  689 2,111,757 1,113,292 1,094,855  PROGRAM TOTALS

--------------------------------------------------------------------------------
        Ir I1mr I2mr        Dr D1mr D2mr        Dw      D1mw      D2mw  file:function
--------------------------------------------------------------------------------
14,688,273    1    1 6,294,531    0    0 2,098,178 1,113,088 1,094,656  example.c:main

--------------------------------------------------------------------------------
-- Auto-annotated source: example.c
--------------------------------------------------------------------------------
       Ir I1mr I2mr        Dr D1mr D2mr        Dw      D1mw      D2mw

        .    .    .         .    .    .         .         .         .  int main(void)
       10    0    0         0    0    0         1         0         0  {
        .    .    .         .    .    .         .         .         .     int i, j, a[1024][1024];
        .    .    .         .    .    .         .         .         .
    4,100    1    1     2,049    0    0         1         0         0     for (i = 0; i < 1024; i++) {
4,198,400    0    0 2,098,176    0    0     1,024         0         0        for (j = 0; j < 1024; j++) {
5,242,880    0    0 2,097,152    0    0 1,048,576    65,536    56,320           a[i][j] = 0;   // fast
5,242,880    0    0 2,097,152    0    0 1,048,576 1,047,552 1,038,336           a[j][i] = 0;   // slow
        .    .    .         .    .    .         .         .         .        }
        .    .    .         .    .    .         .         .         .     }
        1    0    0         0    0    0         0         0         0     return 0;
        2    0    0         2    0    0         0         0         0  }
```

How Cachegrind works

- Each instruction is instrumented
  - Call to a C cache simulation function
  - Different functions for loads, stores, modifies
  - Some combining of C calls for efficiency
- Each source code line gets a *cost centre*
  - Holds counters: accesses, hits and misses
  - Uses debug info to map each instruction to a cost centre
- Online simulation (i.e. no trace gathering)
- Cost centres dumped to file at end
  - Simple but compact text format
  - Post-processing script produces previous slide's output

Cache simulation

- Approximates an AMD Athlon hierarchy
  - I1, D1, inclusive L2
  - Write-allocate
  - LRU replacement
- Each cache is command-line configurable:
  - Cache size
  - Line size
  - Associativity
- On x86/AMD64 can use CPUID to auto-detect these parameters
- Simulation can be replaced easily

Inaccuracies

- Imperfect address trace
  - No kernel code
  - Other processes ignored (arguably good)
  - Conversion to Valgrind's IR changes a very small number of loads/stores
- Incorrect addresses
  - Virtual addresses
  - Memory layout and thread scheduling is different under Cachegrind compared to native
- Prefetches and cache-bypassing are ignored
  - Difficult to handle well without detailed microarchitectural simulation
- Still useful for general insights

How is it used?

- Characterization:
  - Program A vs. program B
  - Cache hierarcy A vs. cache hierarchy B

- Optimisation:
  - Identifies cache-unfriendly code
  - Fixing such code requires non-trivial insight
    - But easier (i.e. not impossible!) than fixing without this data

- Evaluation of optimizations:
  - Program A vs. optimized program A

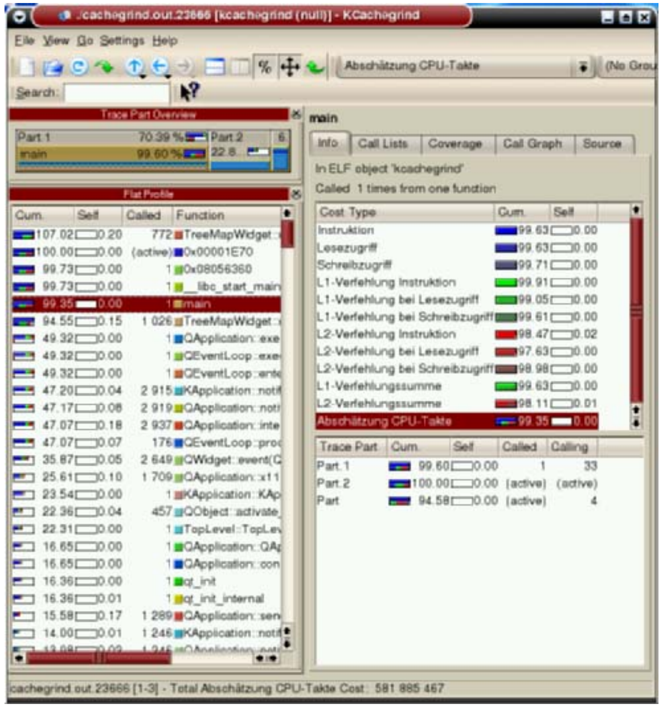
Cachegrind summary

- Cachegrind is a cache simulator
- Gives total, per-function and per-line hit/miss counts
- Simulation is imperfect, but still useful
- Used for characterization, optimisation and evaluation

Callgrind: a call graph profiler

Callgrind

- Extension of Cachegrind
- By Josef Weidendorfer
- Also provides:
  - Call graph information
  - Graphical results viewer (KCachegrind)
    - Allows interactive browsing of results
    - Accepts Cachegrind results also
  - Greater selectivity of what code is profiled



Cum.	Self	Called	Function
107.02	0.20	772	TreeMapWidget:
100.00	0.00	(active)	0x00001E70
99.73	0.00	1	0x08056360
99.73	0.00	1	__libc_start_main
99.35	0.00	1	main
94.55	0.15	1 026	TreeMapWidget:
49.32	0.00	1	QApplication::exe
49.32	0.00	1	QEventLoop::exe
49.32	0.00	1	QEventLoop::ente
47.20	0.04	2 915	KApplication::notif
47.17	0.08	2 919	QApplication::notif
47.07	0.18	2 937	QApplication::inte
47.07	0.07	176	QEventLoop::proc
35.87	0.05	2 649	QWidget::event(C
25.61	0.10	1 709	QApplication::x11
23.54	0.00	1	KApplication::KAp
22.36	0.04	457	QObject::activate_
22.31	0.00	1	TopLevel::TopLev
16.65	0.00	1	QApplication::QAp
16.65	0.00	1	QApplication::con
16.36	0.00	1	qt_init
16.36	0.01	1	qt_init_internal
15.58	0.17	1 289	QApplication::sen
14.00	0.01	1 246	KApplication::notif
13.08	0.02	1 246	QApplication::noti

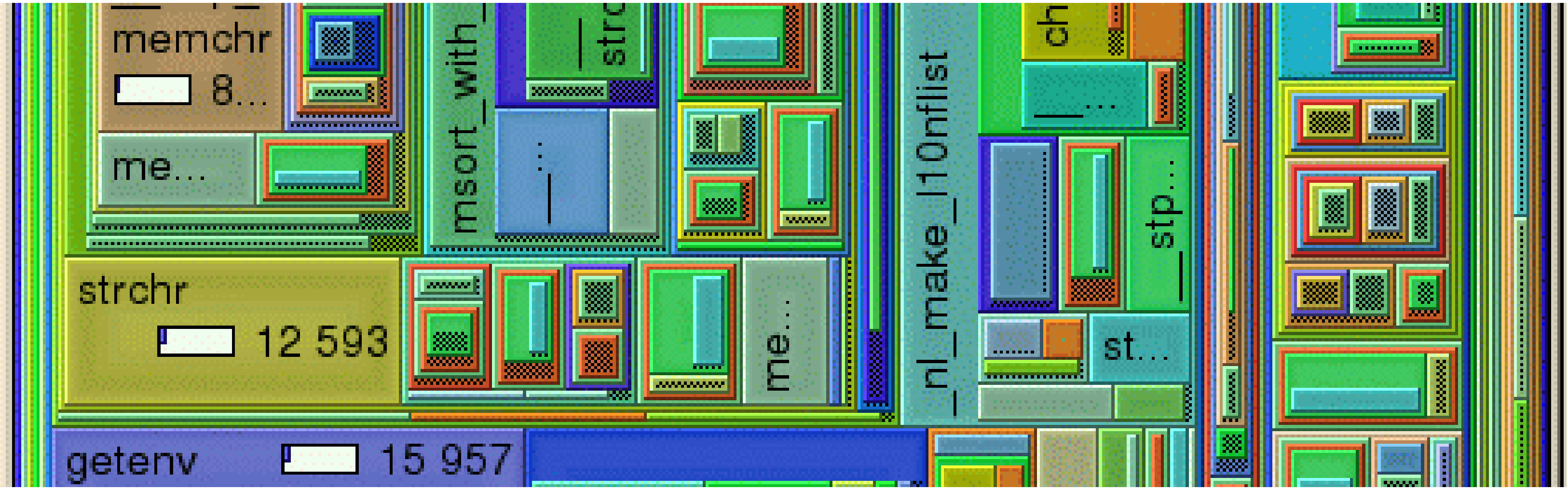
In ELF object 'kcachegrind'
Called 1 times from one function

Cost Type	Cum.	Self
Instruktion	99.63	0.00
Lesezugriff	99.63	0.00
Schreibzugriff	99.71	0.00
L1-Verfehlung Instruktion	99.91	0.00
L1-Verfehlung bei Lesezugriff	99.05	0.00
L1-Verfehlung bei Schreibzugriff	99.61	0.00
L2-Verfehlung Instruktion	98.47	0.02
L2-Verfehlung bei Lesezugriff	97.63	0.00
L2-Verfehlung bei Schreibzugriff	98.98	0.00
L1-Verfehlungssumme	99.63	0.00
L2-Verfehlungssumme	98.11	0.01
Abschätzung CPU-Takte	99.35	0.00

Trace Part	Cum.	Self	Called	Calling
Part.1	99.60	0.00	1	33
Part.2	100.00	0.00	(active)	(active)
Part	94.58	0.00	(active)	4

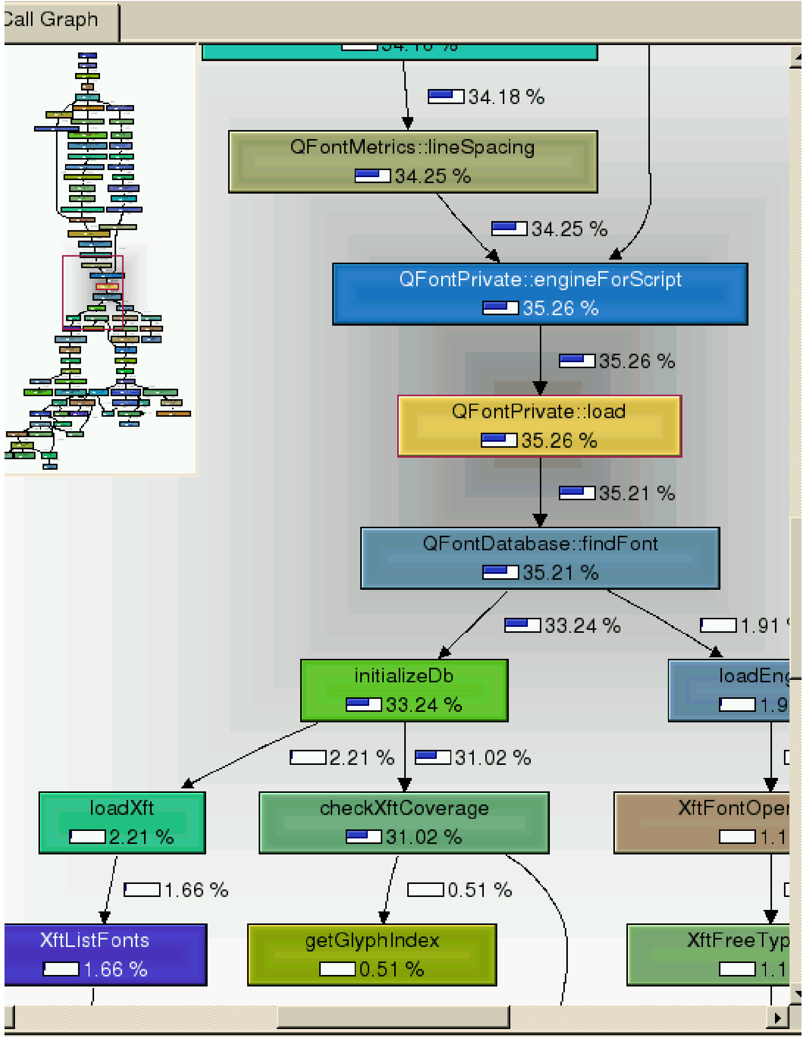
cachegrind.out.23666 [1-3] - Total Abschätzung CPU-Takte Cost: 581 885 467

KCachegrind's tree-map view



- Box sizes represent relative counts
- Nesting of boxes represents call chains
- Interactive: can drill down through boxes

KCachegrind's call graph view



- Shows whole call graph
- Boxes show count proportions
- Interactive

Selective profiling

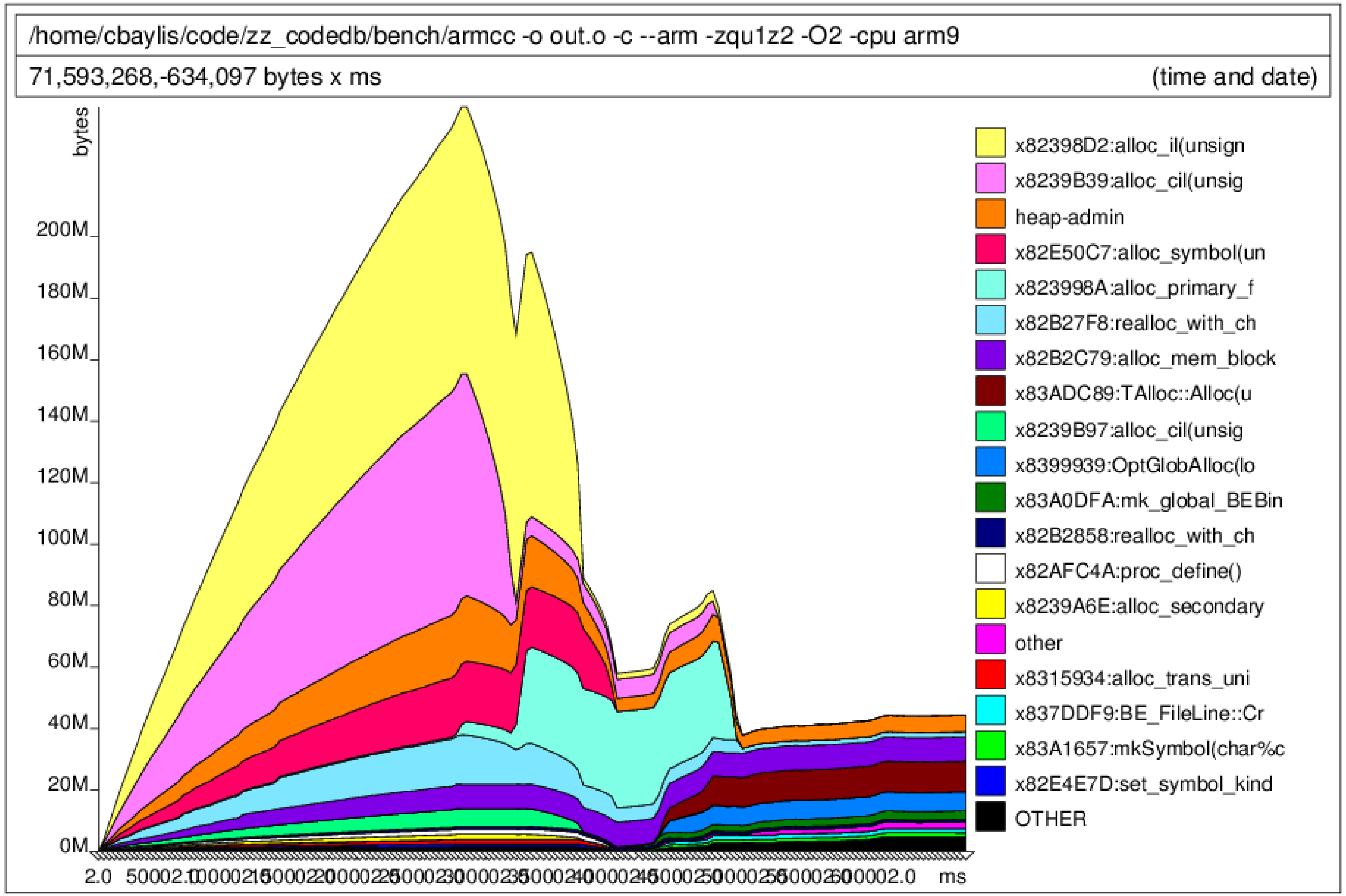
- Can dump counts at particular times
  - At termination (same as Cachegrind)
  - Periodically (every N code blocks)
  - At entry/exit of named functions
  - At particular program points (using client requests)
  - At any time (by invoking a separate script)
- Counters are zeroed after each dump
- Can choose which events to count
  - Instructions
  - Memory events (for cache simulation)
  - Function entries/exits

An interesting difficulty

- Callgrind maintains a call stack
  - For tracking function entries/exits
- Several difficulties:
  - `setjmp/longjmp`
  - Tail recursion
  - Dynamic linking
    - Calls through jump tables
    - Jump table patched on first call after loading
  - Stack switching
- Missed entries/exits can throw everything out

Massif: a heap profiler

Massif heap graph



/home/cbaylis/code/zz_codedb/bench/armcc -o out.o -c --arm -zqu1z2 -O2 -cpu arm9

71,593,268,-634,097 bytes x ms (time and date)

Massif

- Measures heap and stack
  - Each heap allocation site is a band
  - Stack is a band
- Also produces HTML output
  - Represents the call graph underlying allocations
  - Users can drill down through calling chains from allocation sites
- Simple interaction with Valgrind’s core
  - Only uses function wrapping
  - No instrumentation of code blocks
  - Complexity in the tool, not at the core/tool boundary

Thanks

Next class

- Virtualization

System Virtualization

Yubin Xia
Software School
Shanghai Jiao Tong University

Some Slides adapted from
VMWare's academic course plan

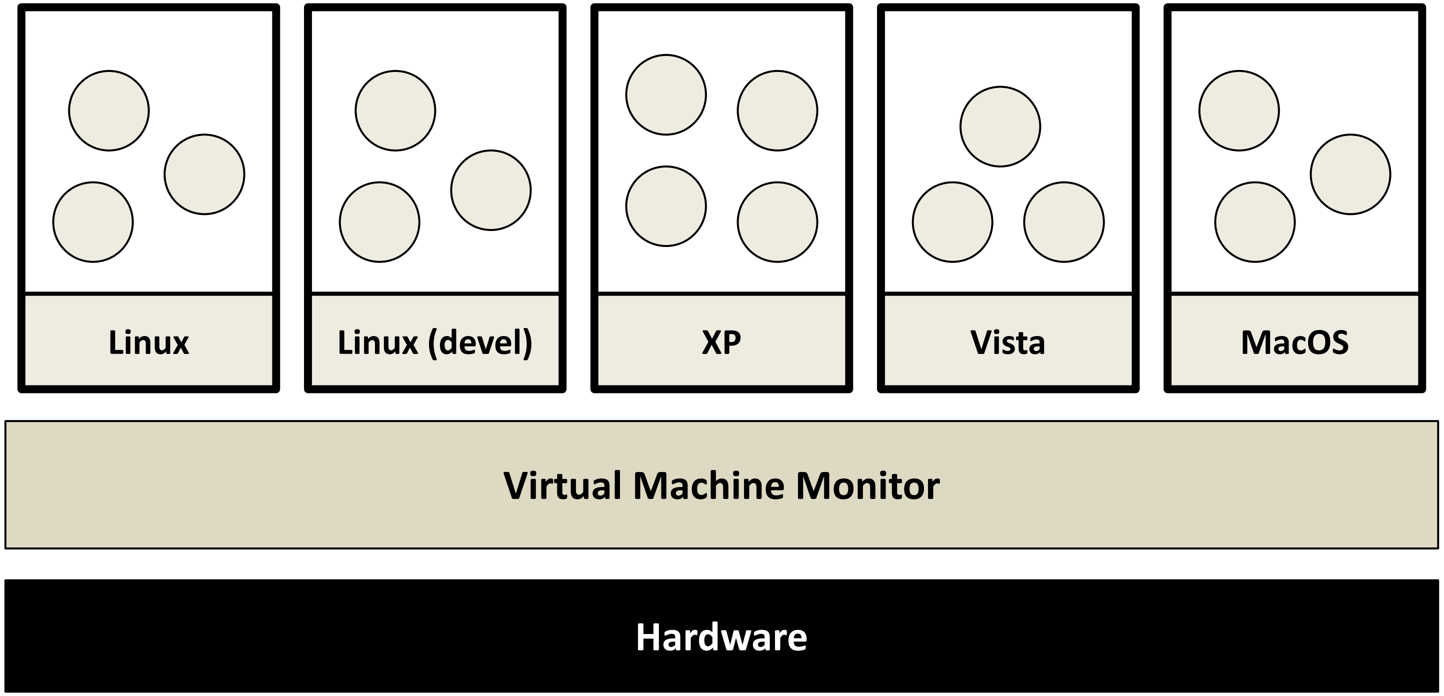
Outline

- Virtualizing CPU

- Virtualizing Memory

- Virtualizing I/O

What is Virtualization



Virtualization Properties

- Isolation
- Encapsulation
- Interposition

Isolation

- Fault Isolation
  - Fundamental property of virtualization
- Software Isolation
  - Software versioning
  - DLL Hell
- 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
- Ex
  - Compression
  - Encryption
  - Profiling
  - Translation

Why Not the OS?

- It about interfaces
  - VMMs operate at the hardware interface
  - Hardware interface are typically smaller, better defined than software interfaces

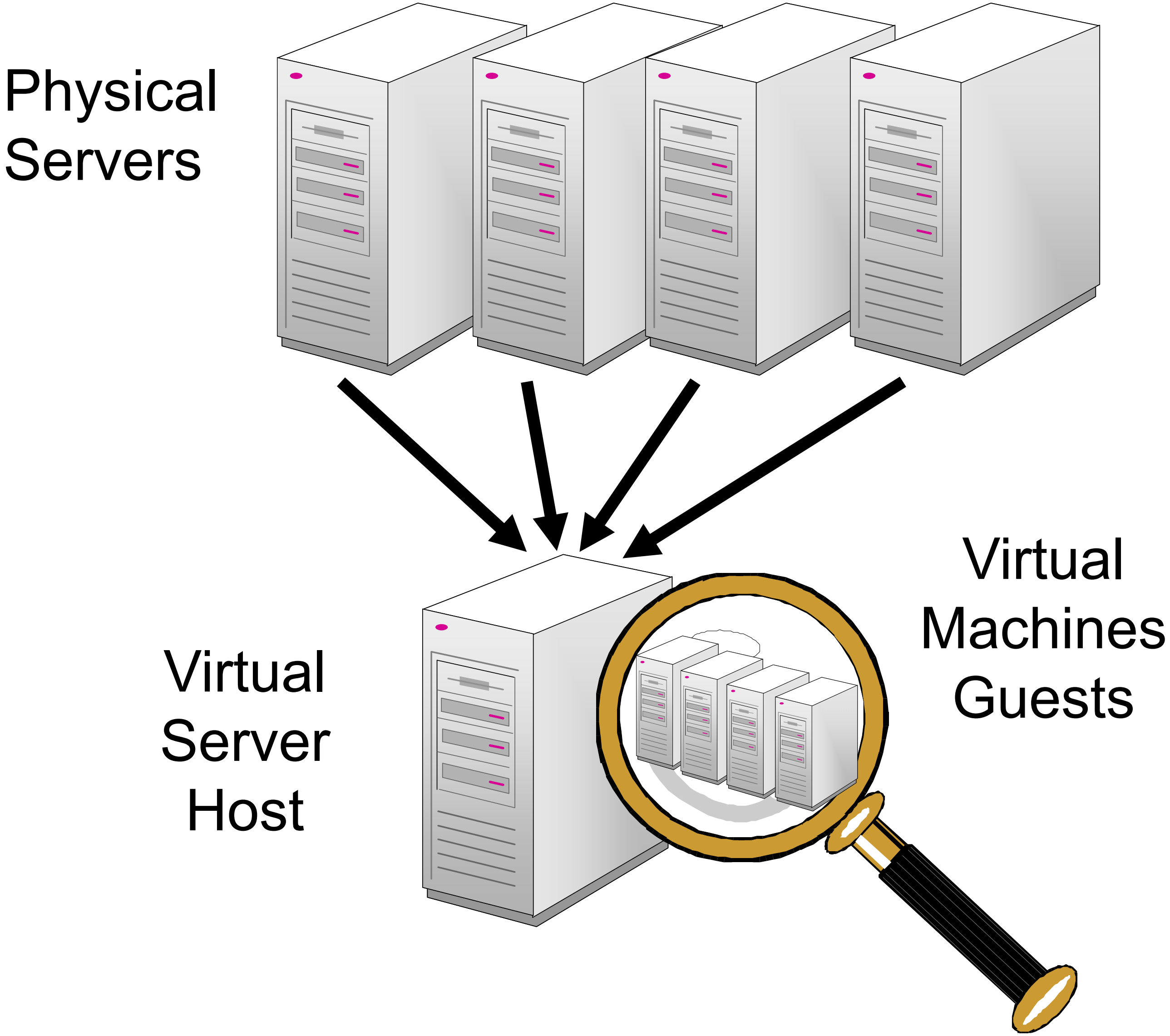
- Microkernel for commodity Operating Systems

- Disadvantages of being in the monitor
  - Low visibility into what the guest is doing

Virtualization benefits

- Increased resource utilization:
  - Server consolidation
- Mobility
- Enhanced Security
- Trusted Computing
- Test and Deployment
- …

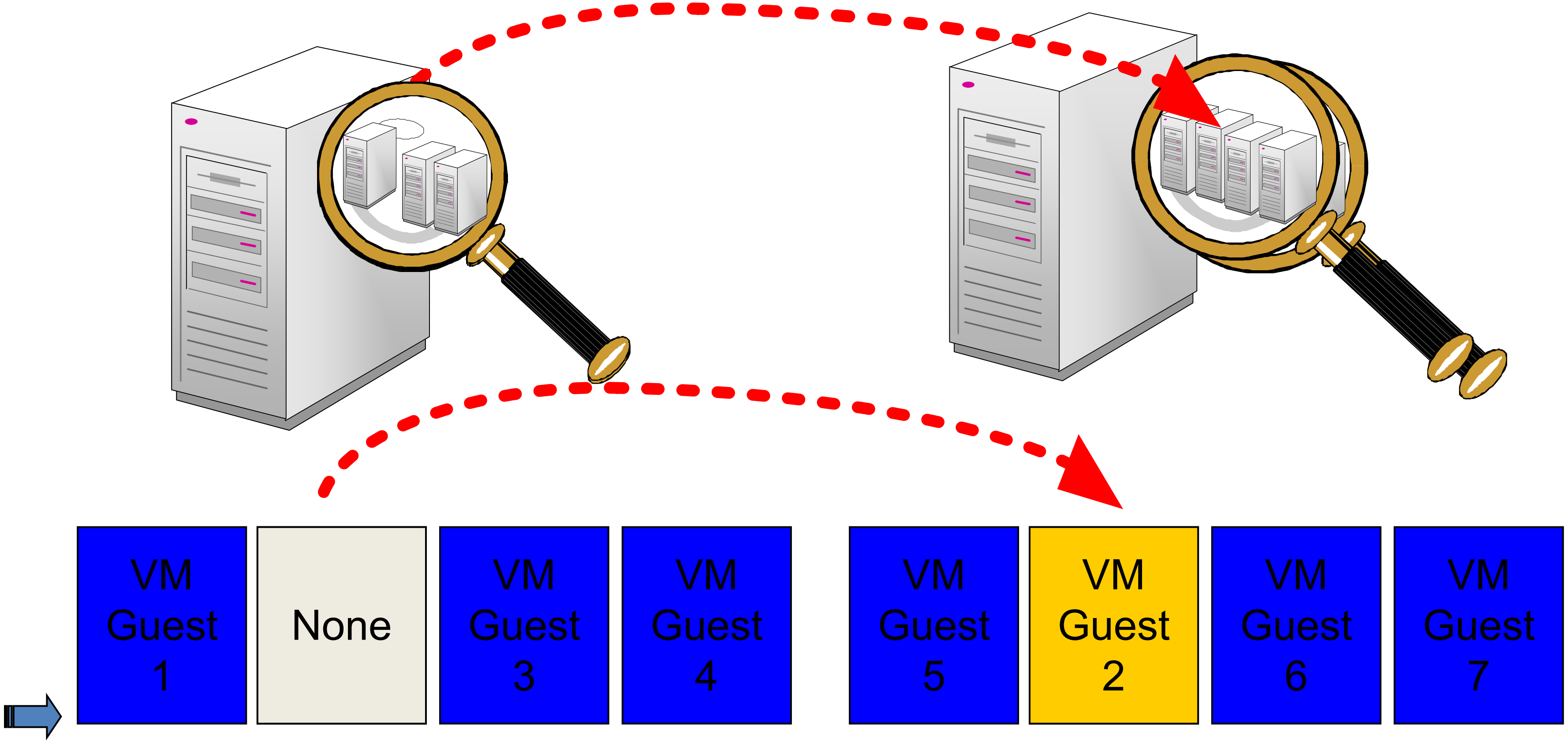
Virtualization: server consolidation



Mobility: load balance

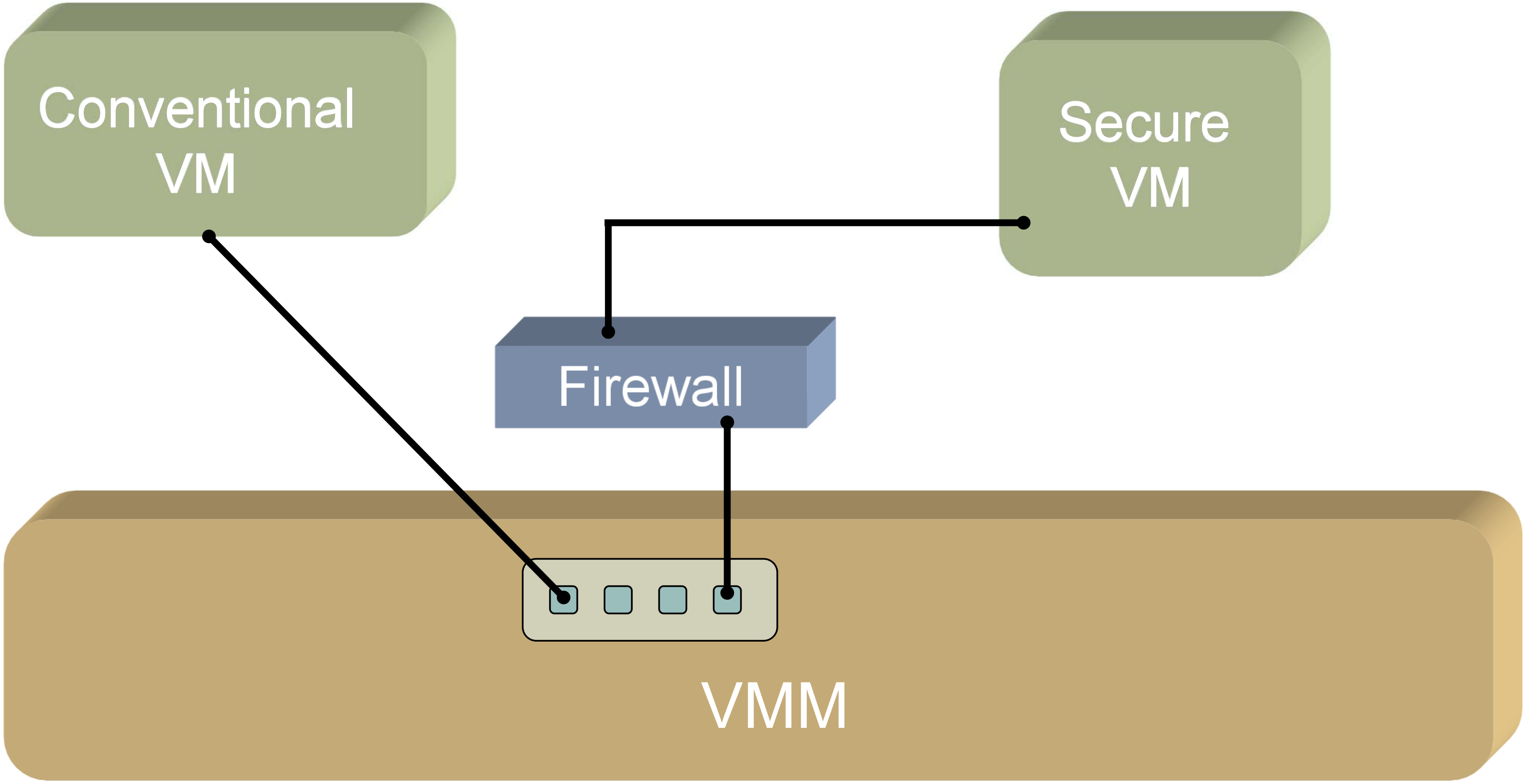
Server 1
CPU Utilization = 90% 70%

Server 2
CPU Utilization = 50% 70%

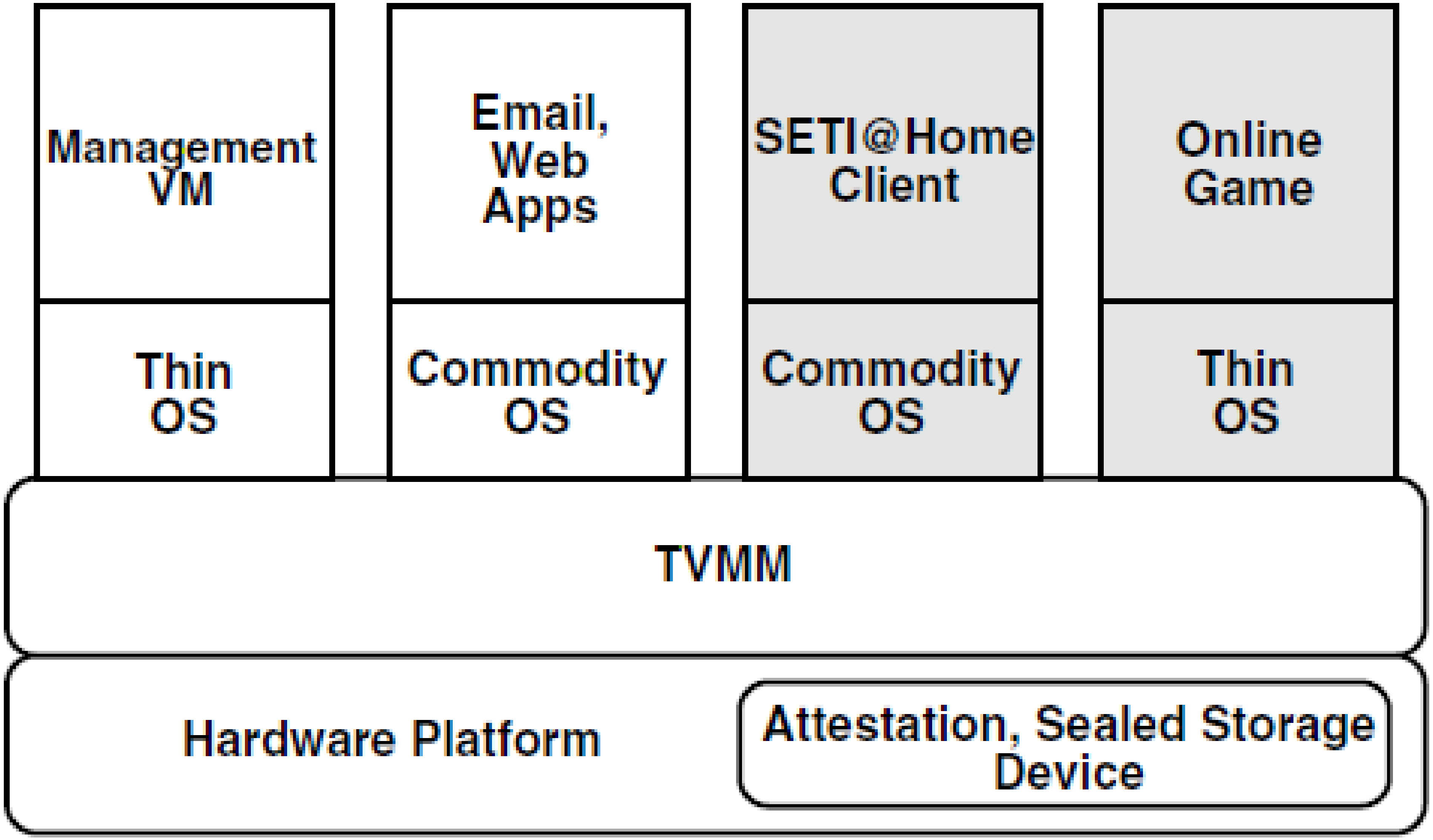


VM Guest 1	None	VM Guest 3	VM Guest 4	VM Guest 5	VM Guest 2	VM Guest 6	VM Guest 7

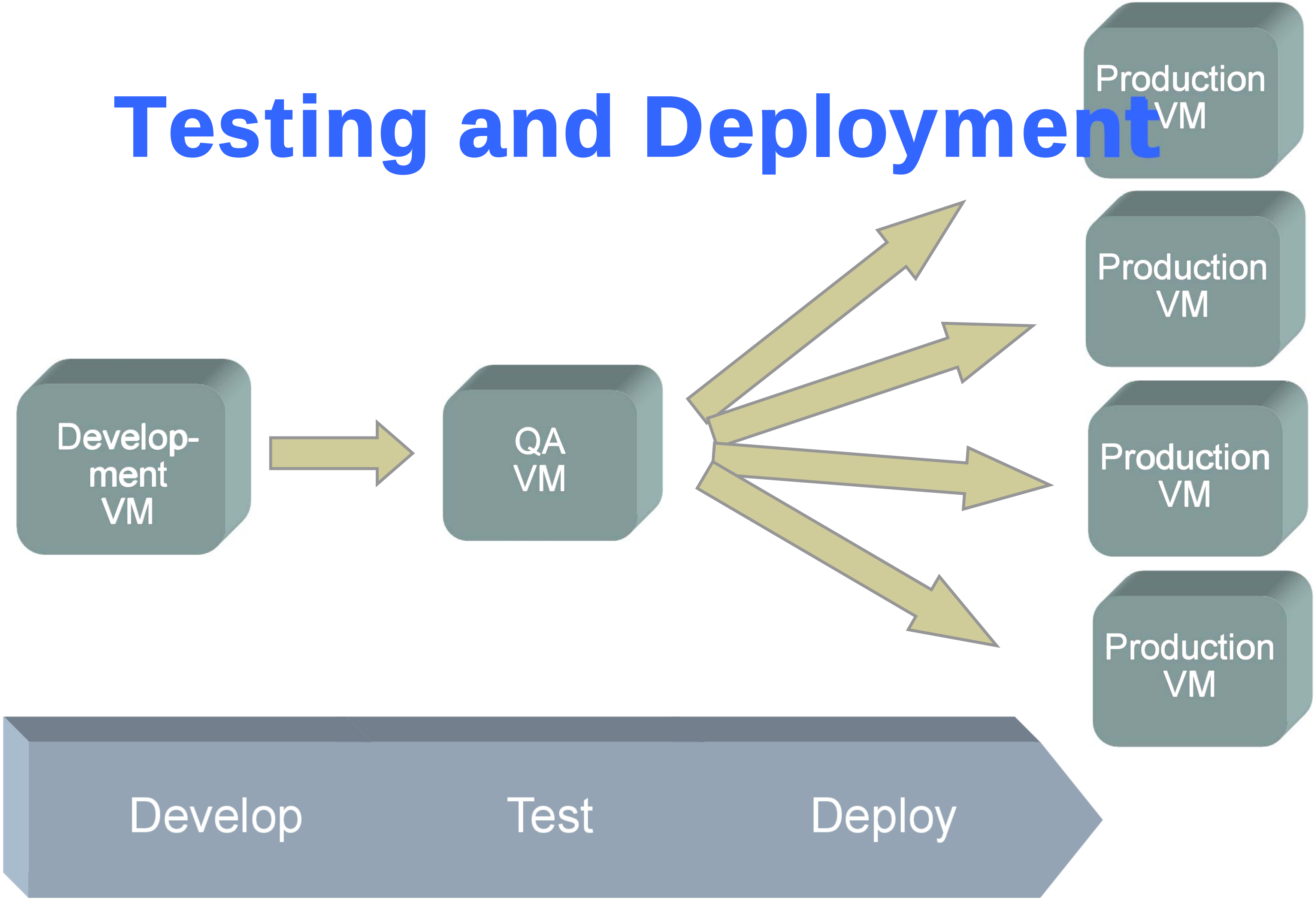
Enhanced Security



Trusted Computing



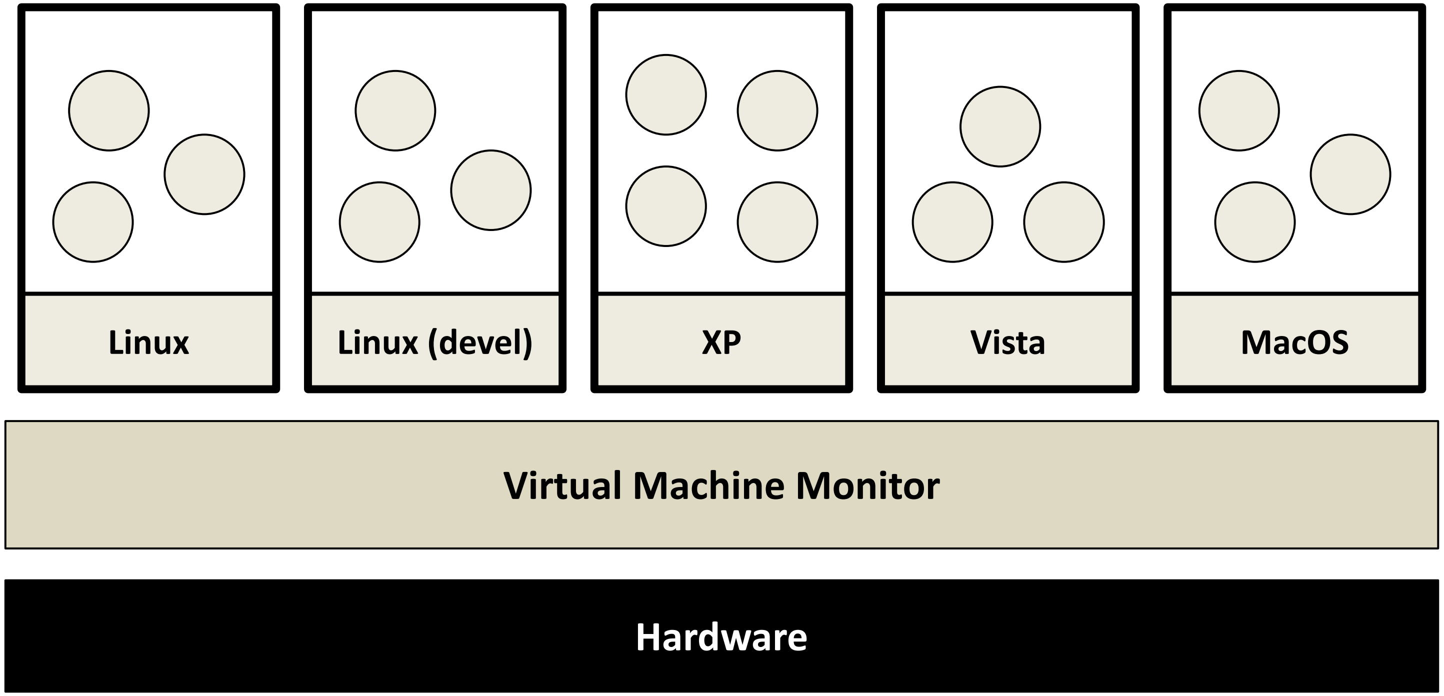
Testing and Deployment



System Virtual Machine Monitor Architectures

- Traditional
- Hosted
  - VMware Workstation
- Hybrid
  - VMware ESX
  - Xen
- Hypervisor

Traditional

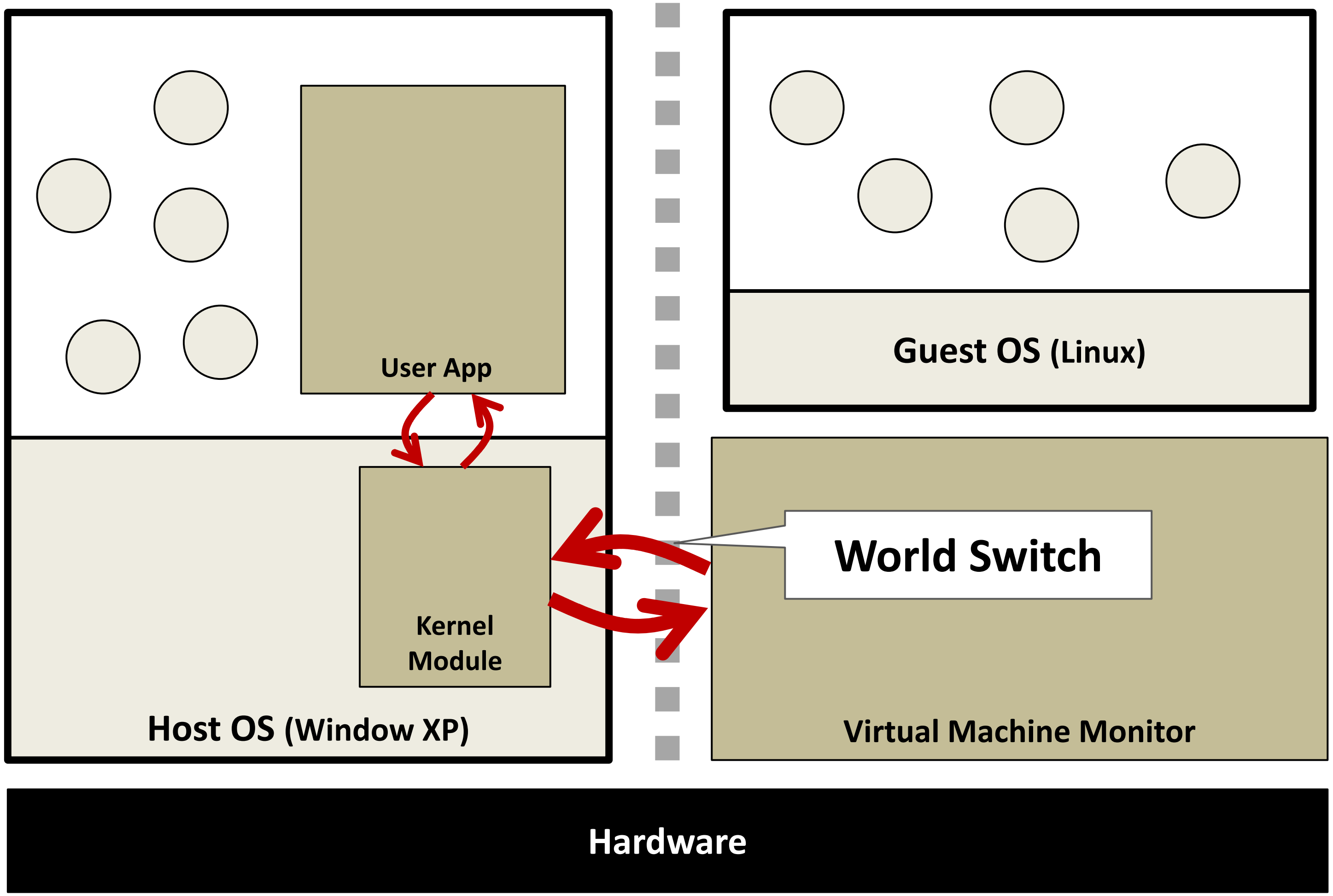


- Examples: IBM VM/370, Stanford DISCO

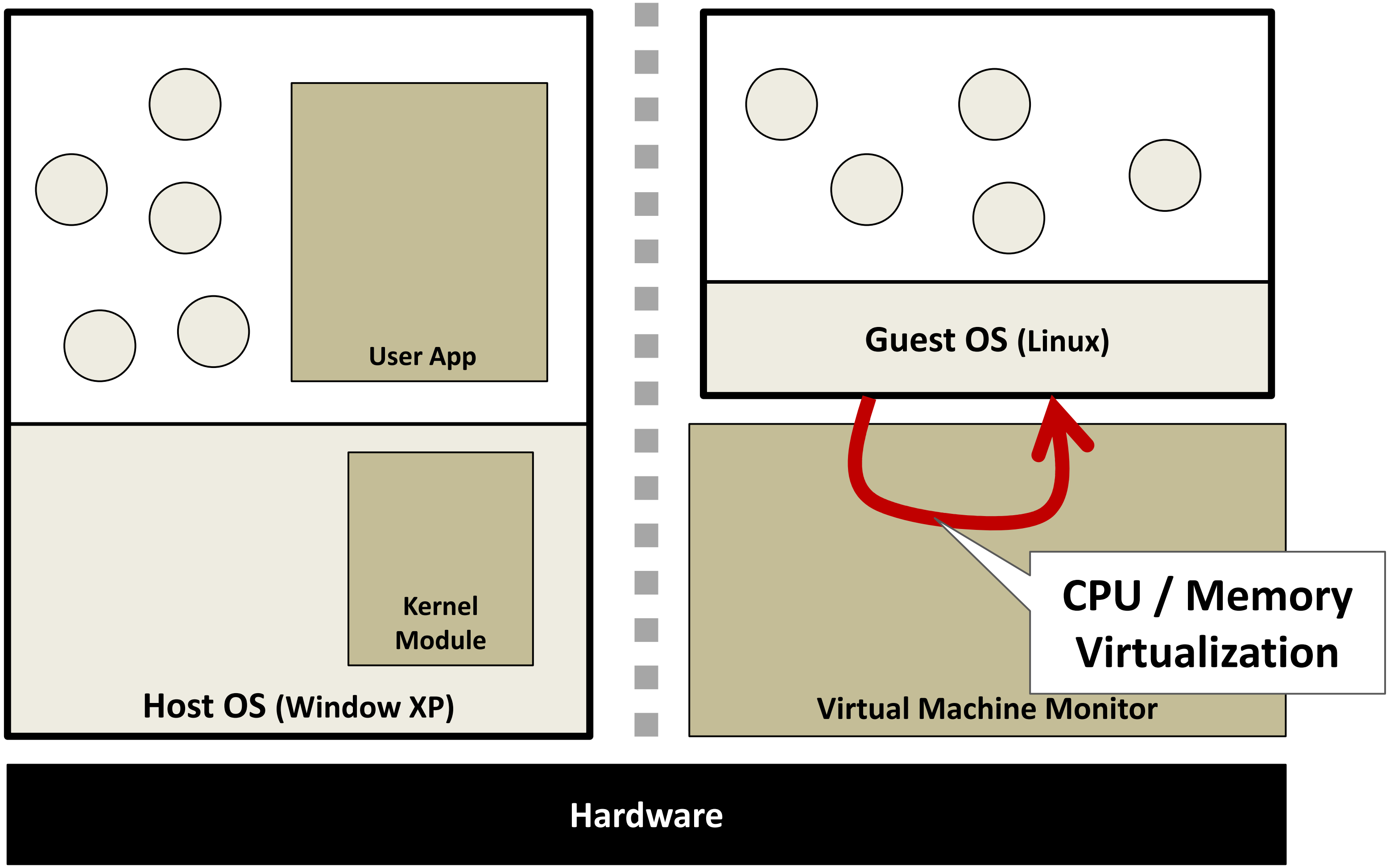
Hosted Virtual Machines

- 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

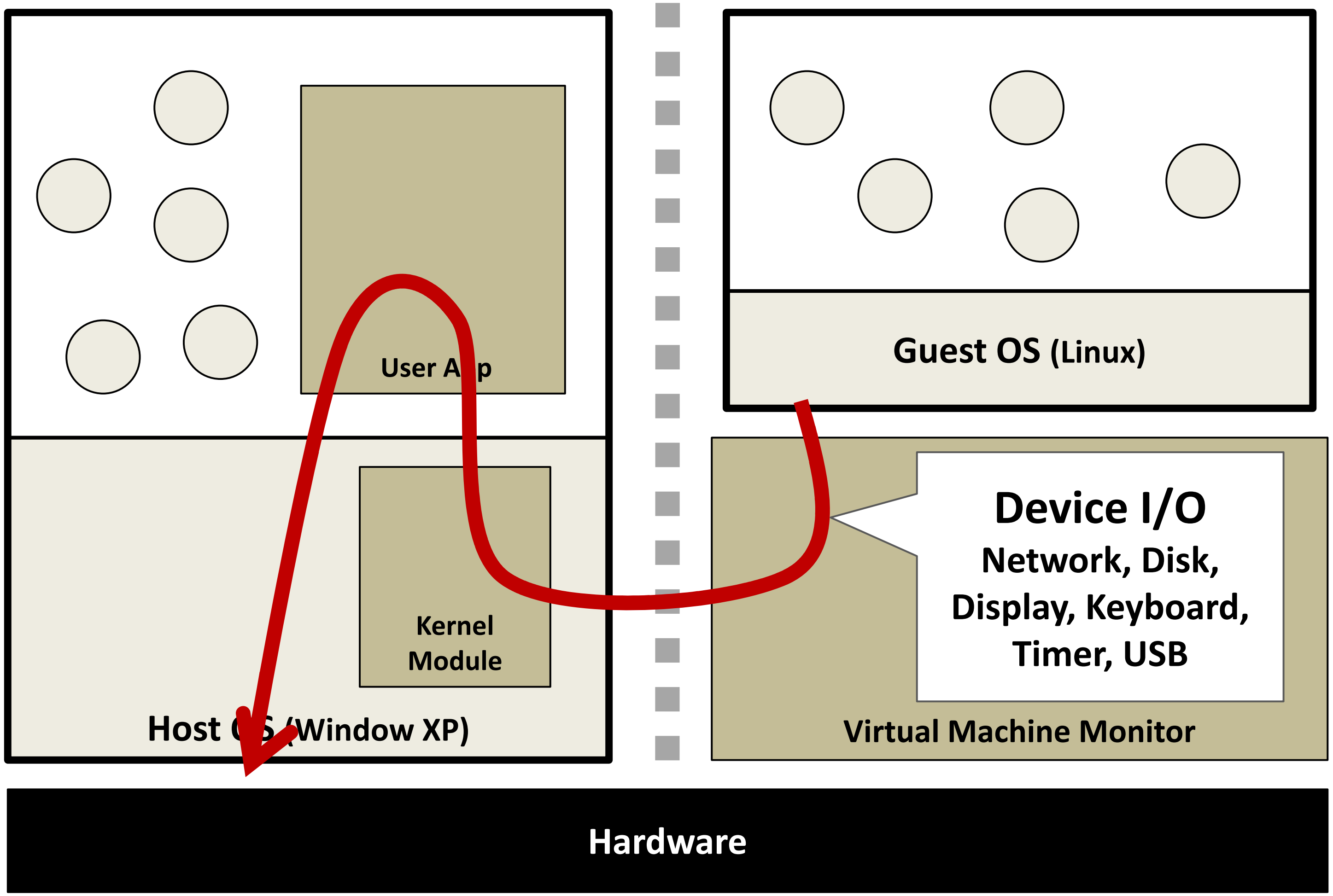
Hosted Monitor Architecture



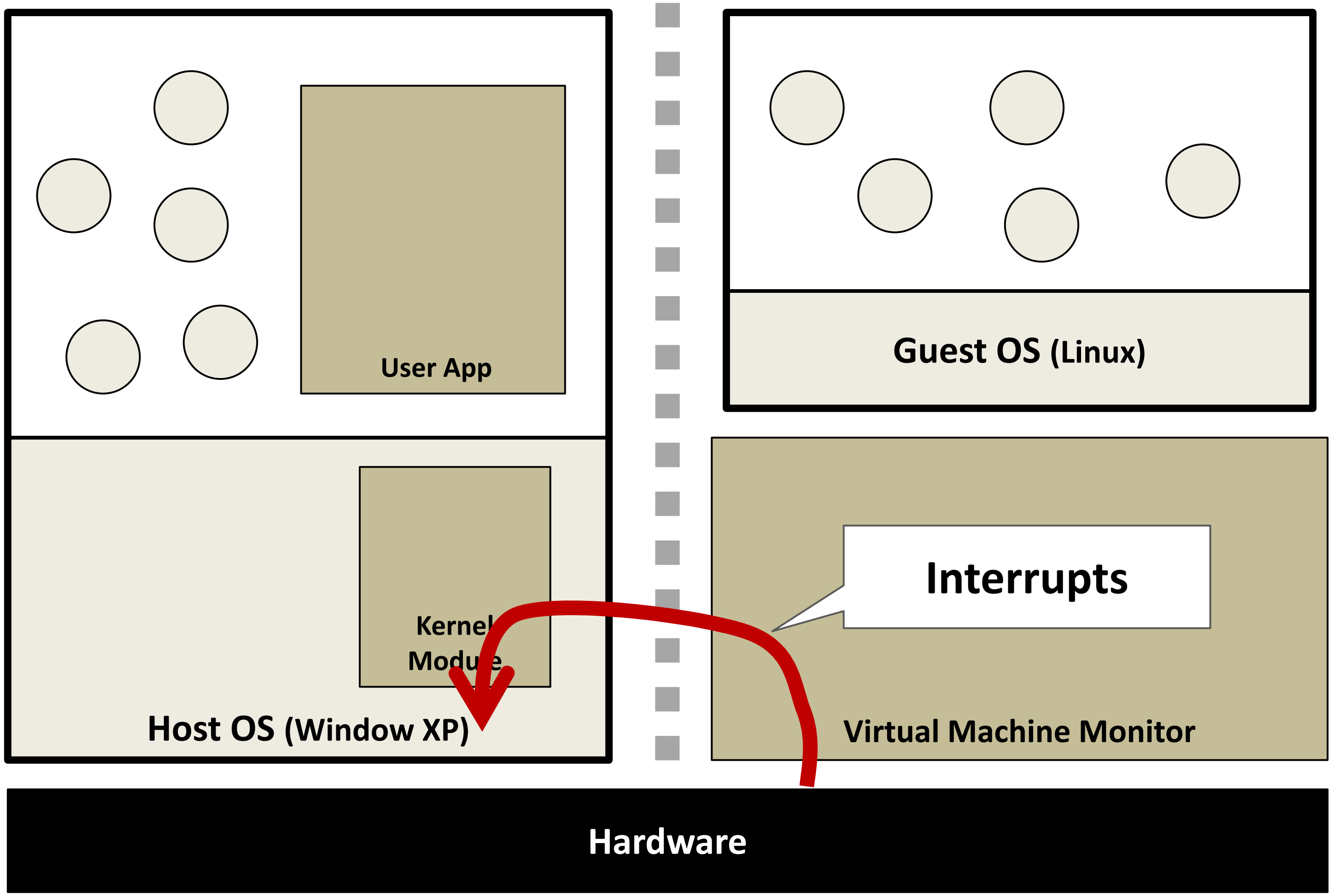
Hosted Monitor Architecture



Hosted Monitor Architecture



Hosted Monitor Architecture



Hosted Architecture Tradeoffs

- Positives
  - Installs like an application
    - No disk partitioning needed
    - Virtual disk is a file on host file system
    - No host reboot needed
  - Runs like an application
    - Uses host schedulers
- Negatives
  - I/O path is slow
    - Requires world switch
  - Relies on host scheduling
    - May not be suitable for intensive VM workloads

VMware ESX 2.0

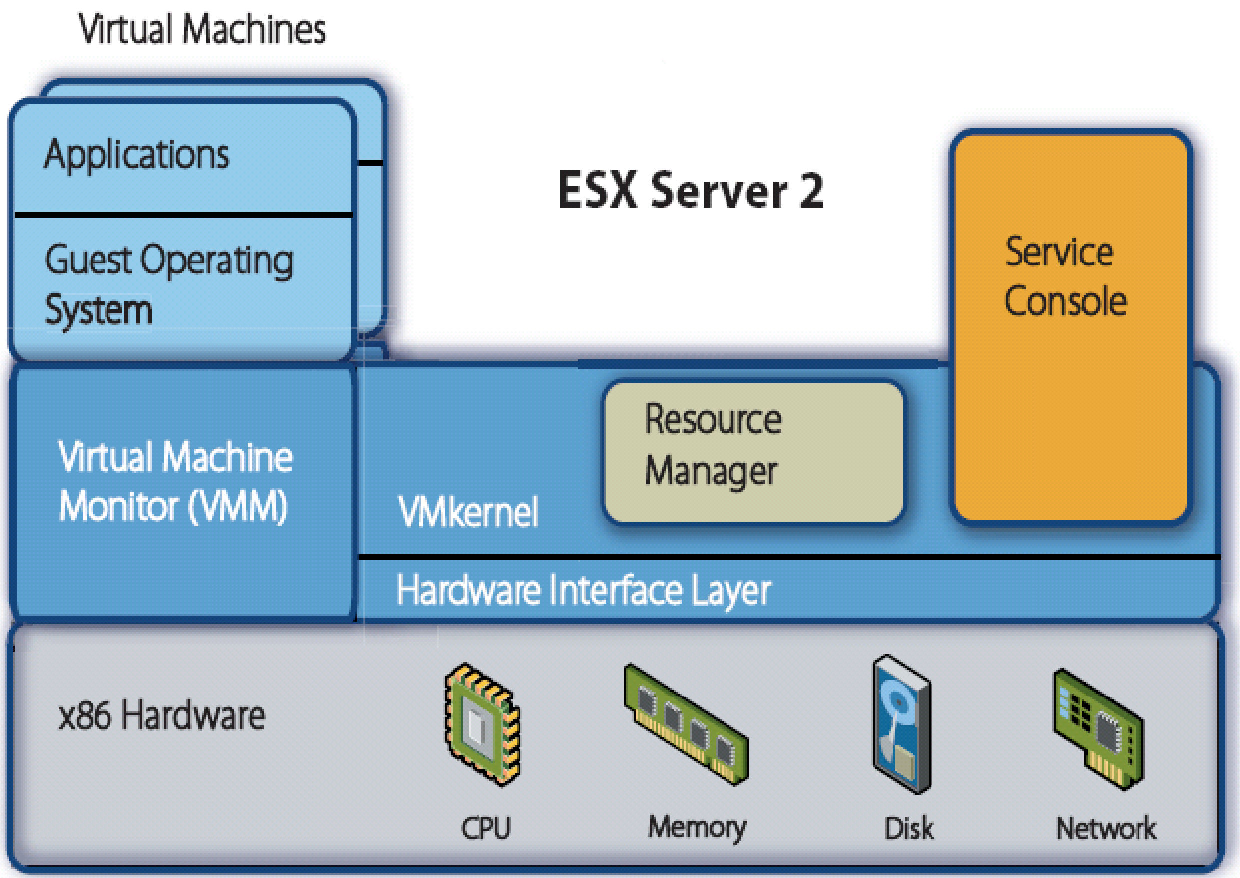
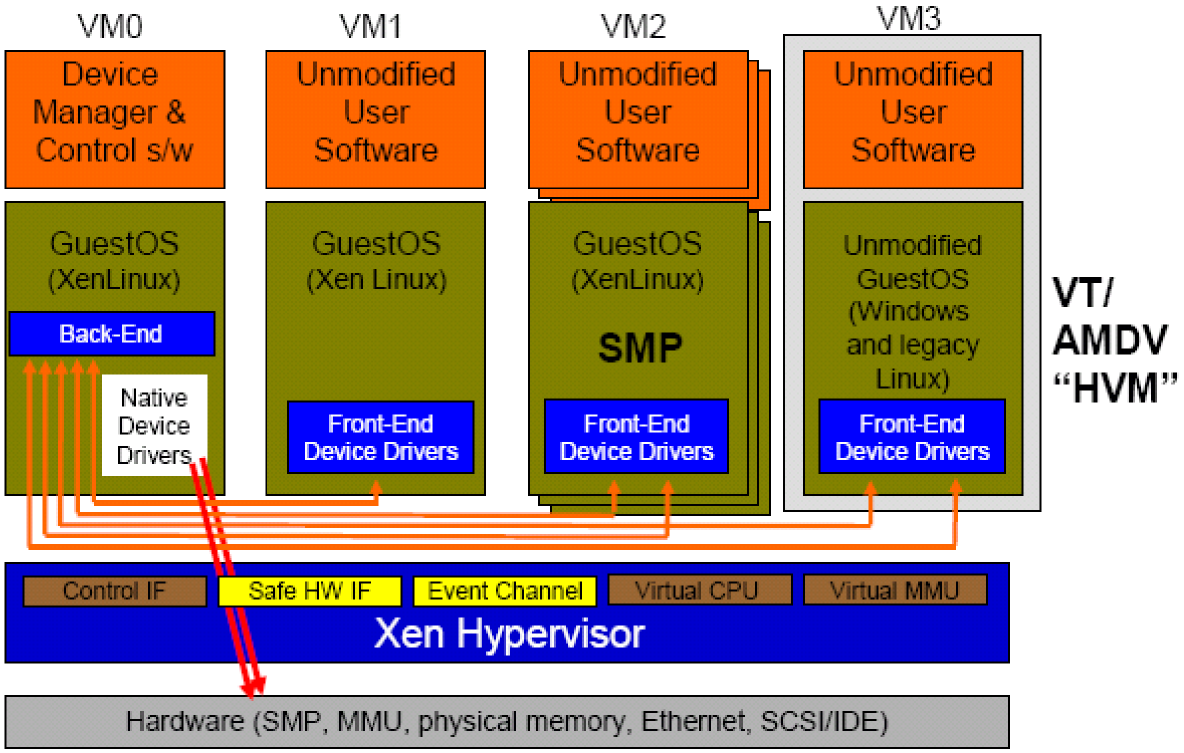


Figure 1: ESX Server architecture

Source: http://www.vmware.com/pdf/esx2_performance_implications.pdf

Hybrid Ex 2 - Xen 3.0

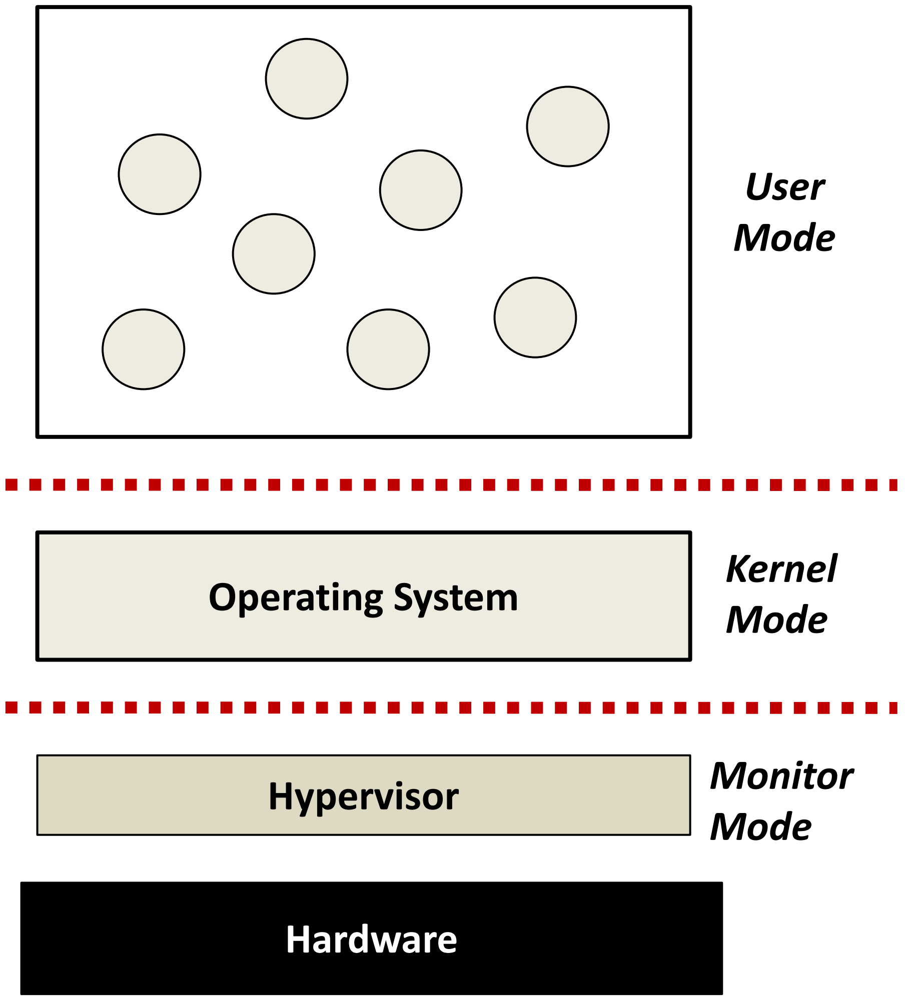
- Para –virtualization
  - Linux Guest
- Hardware-supported virtualization
  - Unmodified Windows
- Isolated Device Drivers



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



CPU VIRTUALIZATION

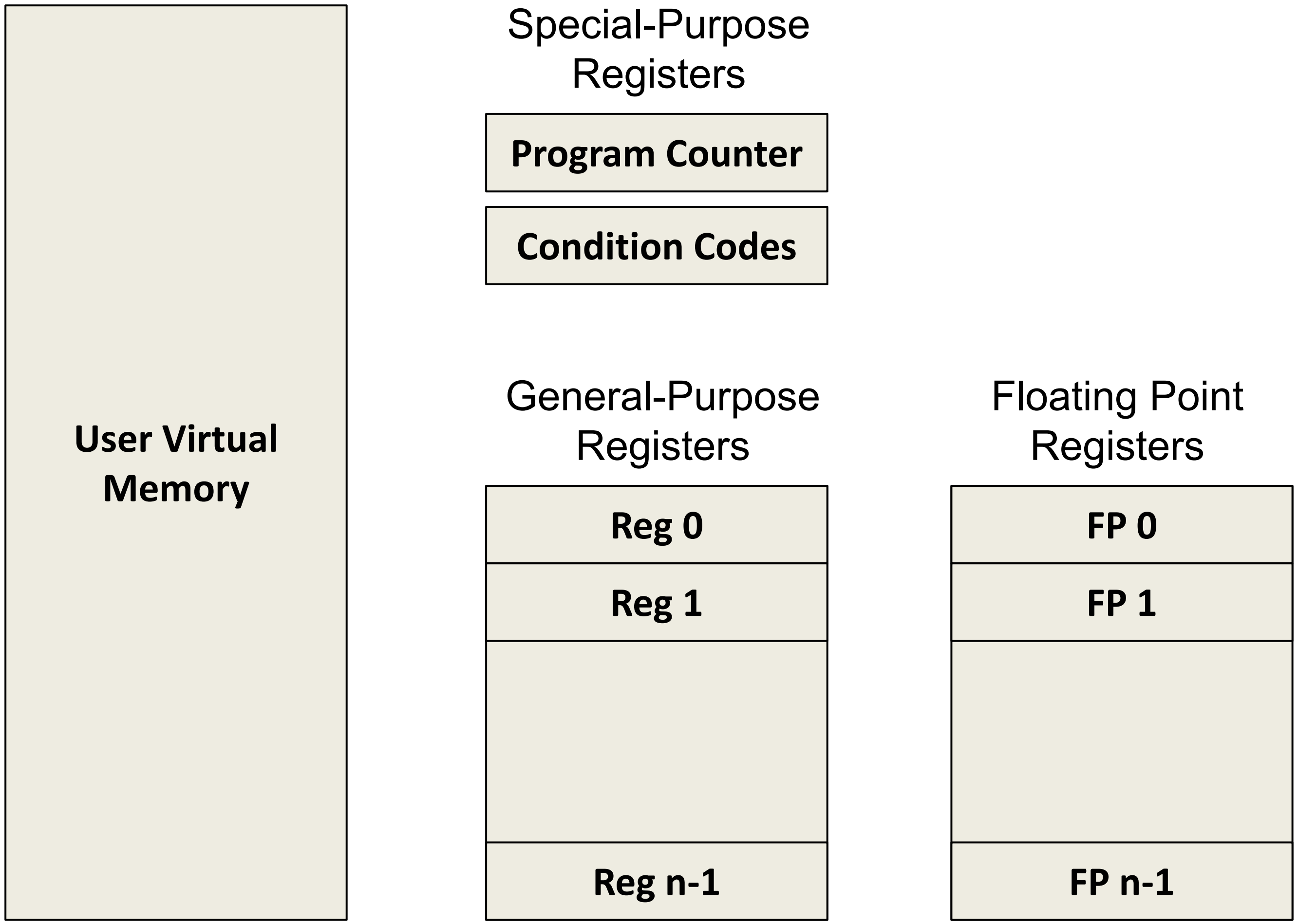
Recap: CPU Organization

- Instruction Set Architecture (ISA)

  Defines:

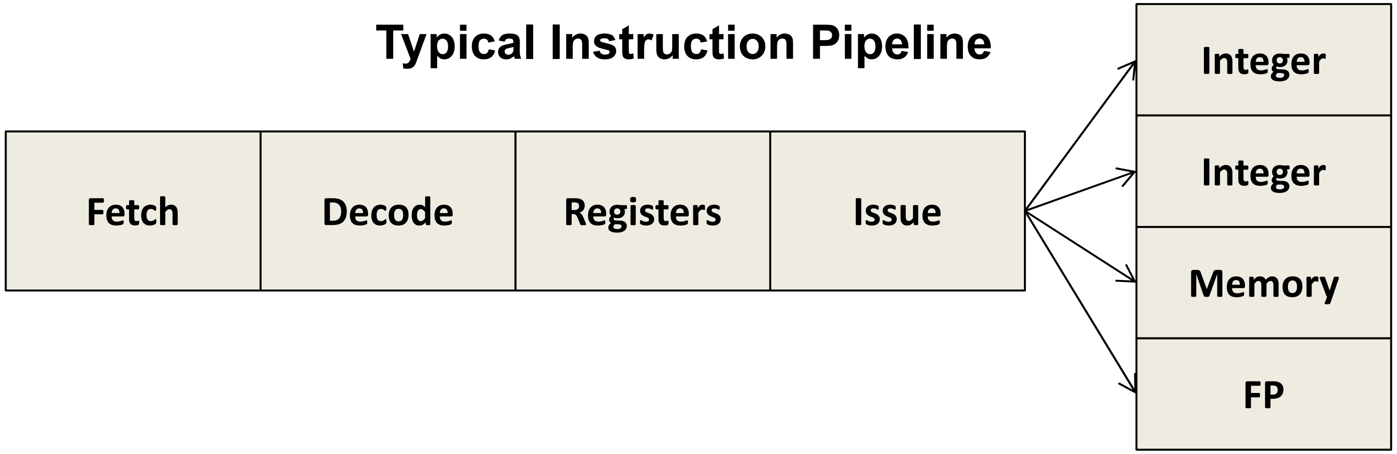
  - the state visible to the programmer
    - registers and memory
  - the instruction that operate on the state
- ISA typically divided into 2 parts
  - User ISA
    - Primarily for computation
  - System ISA
    - Primarily for system resource management

Recap: User ISA - State



Recap: User ISA – Instructions

Typical Instruction Pipeline

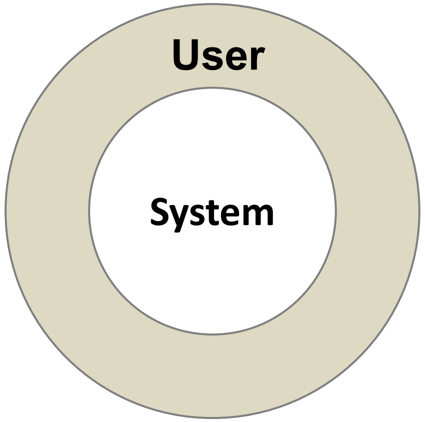


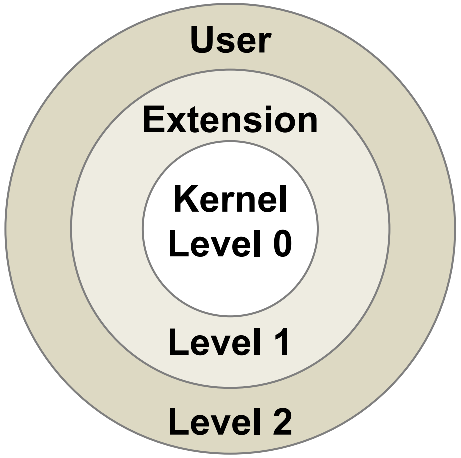
Integer	Memory	Control Flow	Floating Point
Add Sub And Compare …	Load byte Load Word Store Multiple Push …	Jump Jump equal Call Return …	Add single Mult. double Sqrt double …

Instruction Groupings

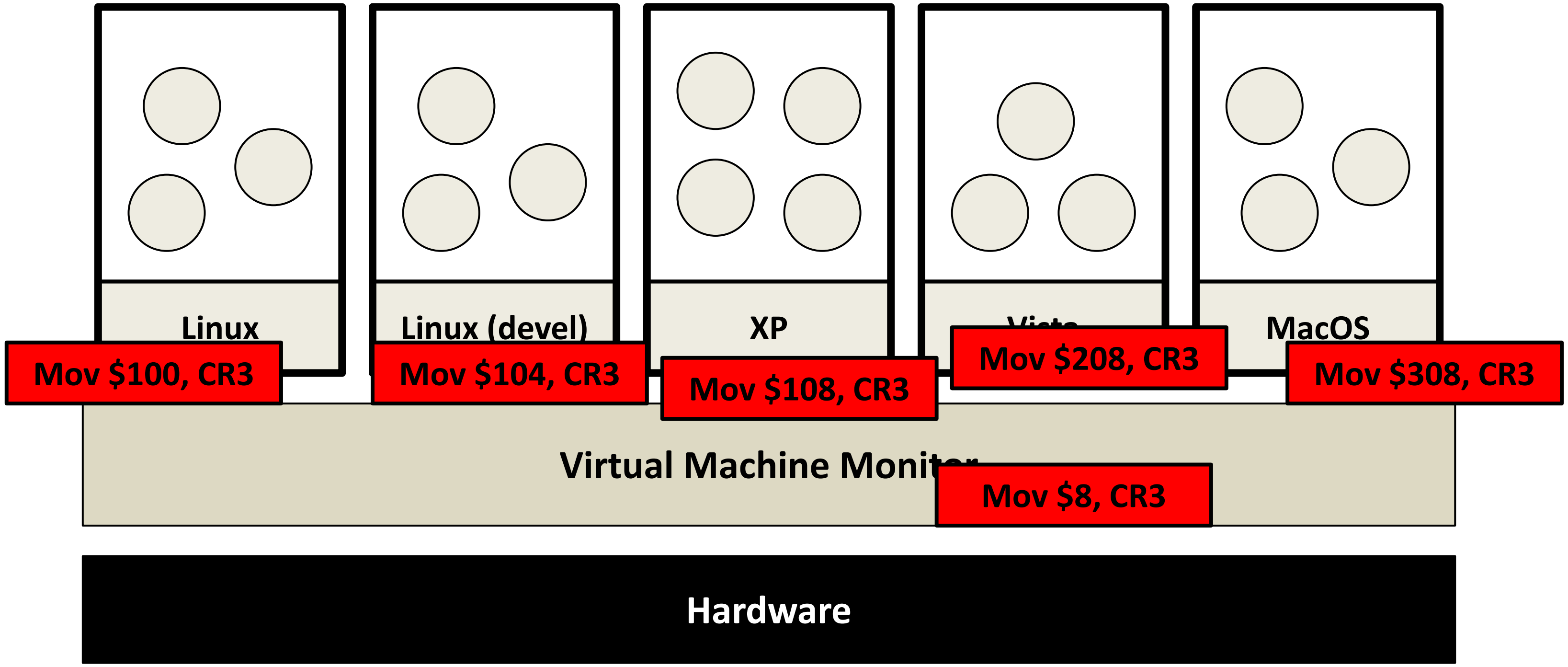
Recap: System ISA

- Privilege Levels
- Control Registers
- Traps and Interrupts
  - Hardcoded Vectors
  - Dispatch Table
- System Clock
- MMU
  - Page Tables
  - TLB
- I/O Device Access





Challenges to Virtualized System ISA



Virtualizing the System ISA

- Hardware needed by monitor
  - Ex: monitor must control real hardware interrupts
- Access to hardware would allow VM to compromise isolation boundaries
  - Ex: access to MMU would allow VM to write any page
- So…
  - All access to the virtual System ISA by the guest must be emulated by the monitor in software.
  - System state kept in memory.
  - System instructions are implemented as functions in the monitor.

Example: CPUState

```
static struct {
   uint32  GPR[16];
   uint32  LR;
   uint32  PC;
   int     IE;
   int     IRQ;
} CPUState;




void CPU_CLI(void)
{
   CPUState.IE = 0;
}

void CPU_STI(void)
{
   CPUState.IE = 1;
}
```

- Goal for CPU virtualization techniques
  - Process normal instructions as fast as possible
  - Forward privileged instructions to emulation routines

Virtualization VS. Multiplexing

- Why not run virtual machines just as user apps?
  - Virtual machine includes guest kernel and guest apps
  - What is the problem?

Definition

- Sensitive Instructions
  - Instructions that inspect or change system states of a processor
  - E.g., mov to CR registers

- Privileged Instructions
  - Instructions that executed at high privileged level

- What if an instruction is sensitive but not privileged?

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

x86 virtualization challenge

- The IA-32 instruction set contains 17 sensitive, unprivileged instructions:
  - Sensitive register instructions: read or change sensitive registers and/or memory locations such as a clock register or interrupt registers:
    - SGDT, SIDT, SLDT, SMSW, PUSHF, POPF
  - Protection system instructions: reference the storage protection system, memory or address relocation system:
    - LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV

Solutions

- Instruction Interpretation
- Trap-and-emulate
- Binary Translation
- Para-virtualization

Instruction Interpretation

- Emulate Fetch/Decode/Execute pipeline in software

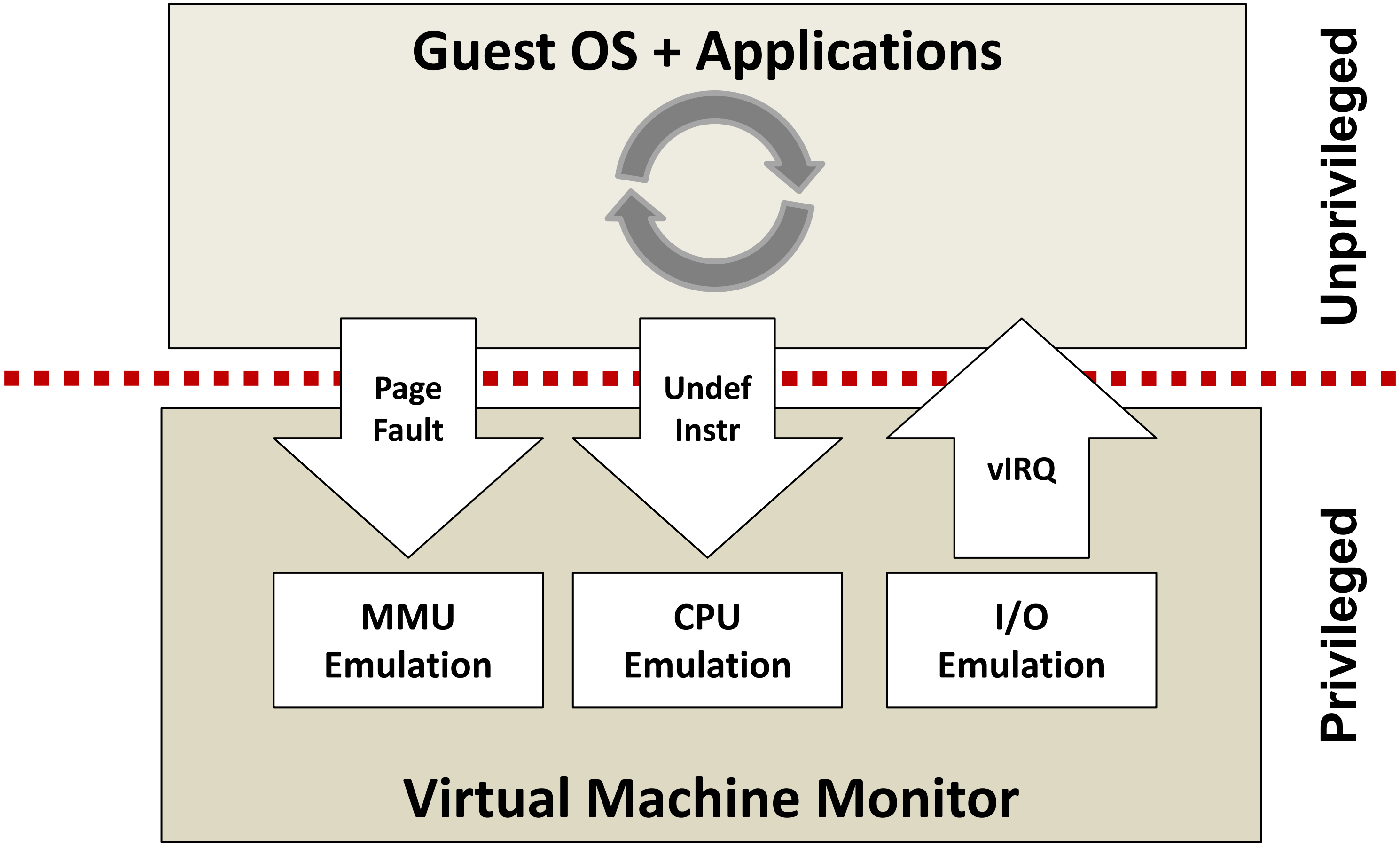
- Positives
  - Easy to implement
  - Minimal complexity

- Negatives
  - Slow!

Example: Virtualizing the Interrupt Flag w/ Instruction Interpreter

```
void CPU_Run(void)
{
   while (1) {
      inst = Fetch(CPUState.PC);

      CPUState.PC += 4;

      switch (inst) {
      case ADD:
         CPUState.GPR[rd]
            = GPR[rn] + GPR[rm];
         break;
      …
      case CLI:
         CPU_CLI();
         break;
      case STI:
         CPU_STI();
         break;
      }

      if (CPUState.IRQ
          && CPUState.IE) {
         CPUState.IE = 0;
         CPU_Vector(EXC_INT);
      }
   }
}
void CPU_CLI(void)
{
   CPUState.IE = 0;
```

Trap and Emulate



Issues with Trap and Emulate

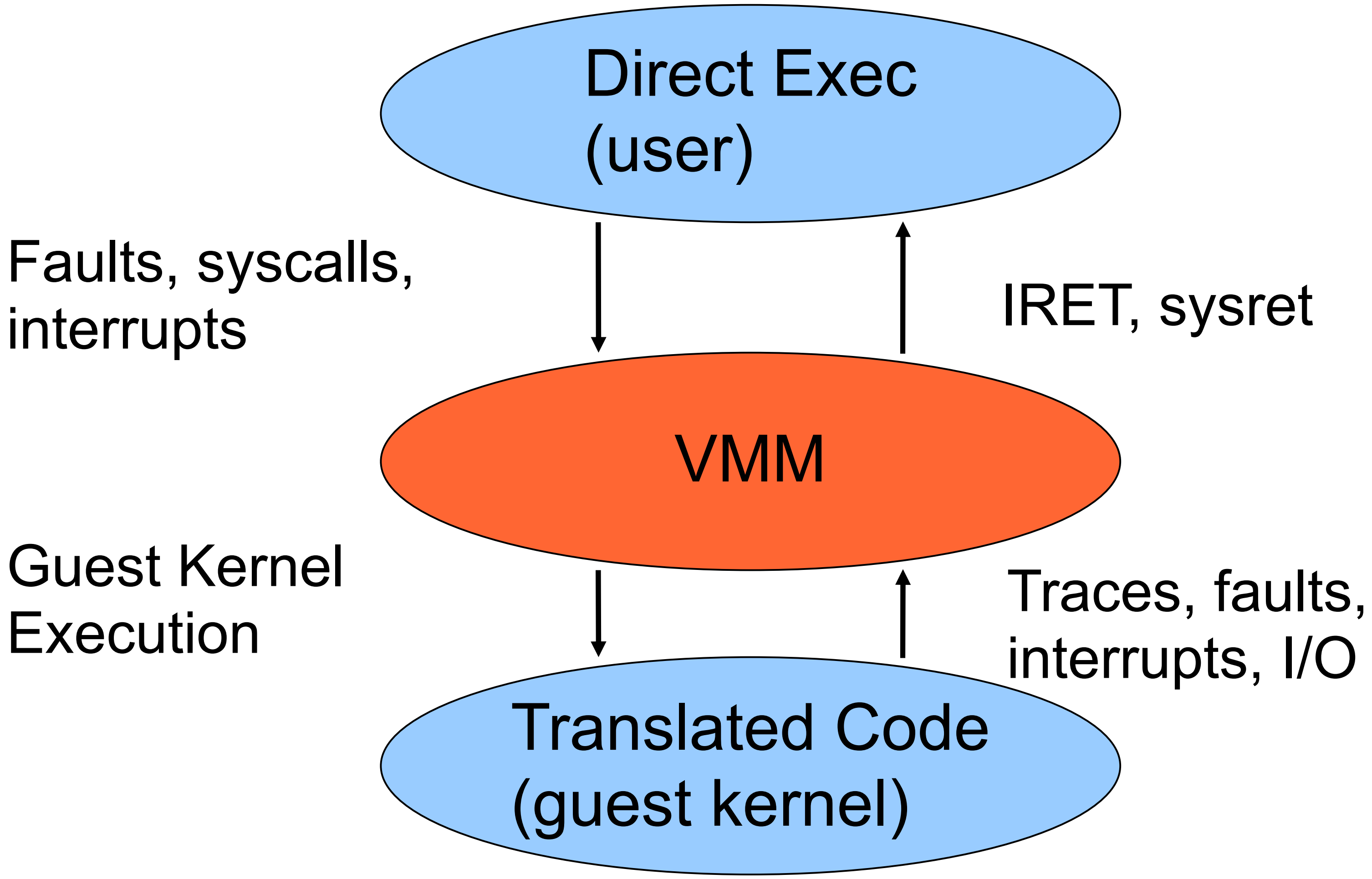
- Not all architectures support it
- Trap costs may be high
- Monitor uses a privilege level
  - Need to virtualize the protection levels

“Strictly Virtualizable”

A processor or mode of a processor is strictly virtualizable if, when executed in a lesser privileged mode:

- all instructions that access privileged state trap
- all instructions either trap or execute identically
- …

Software VMM



Direct Exec
(user)

Faults, syscalls,
interrupts

IRET, sysret

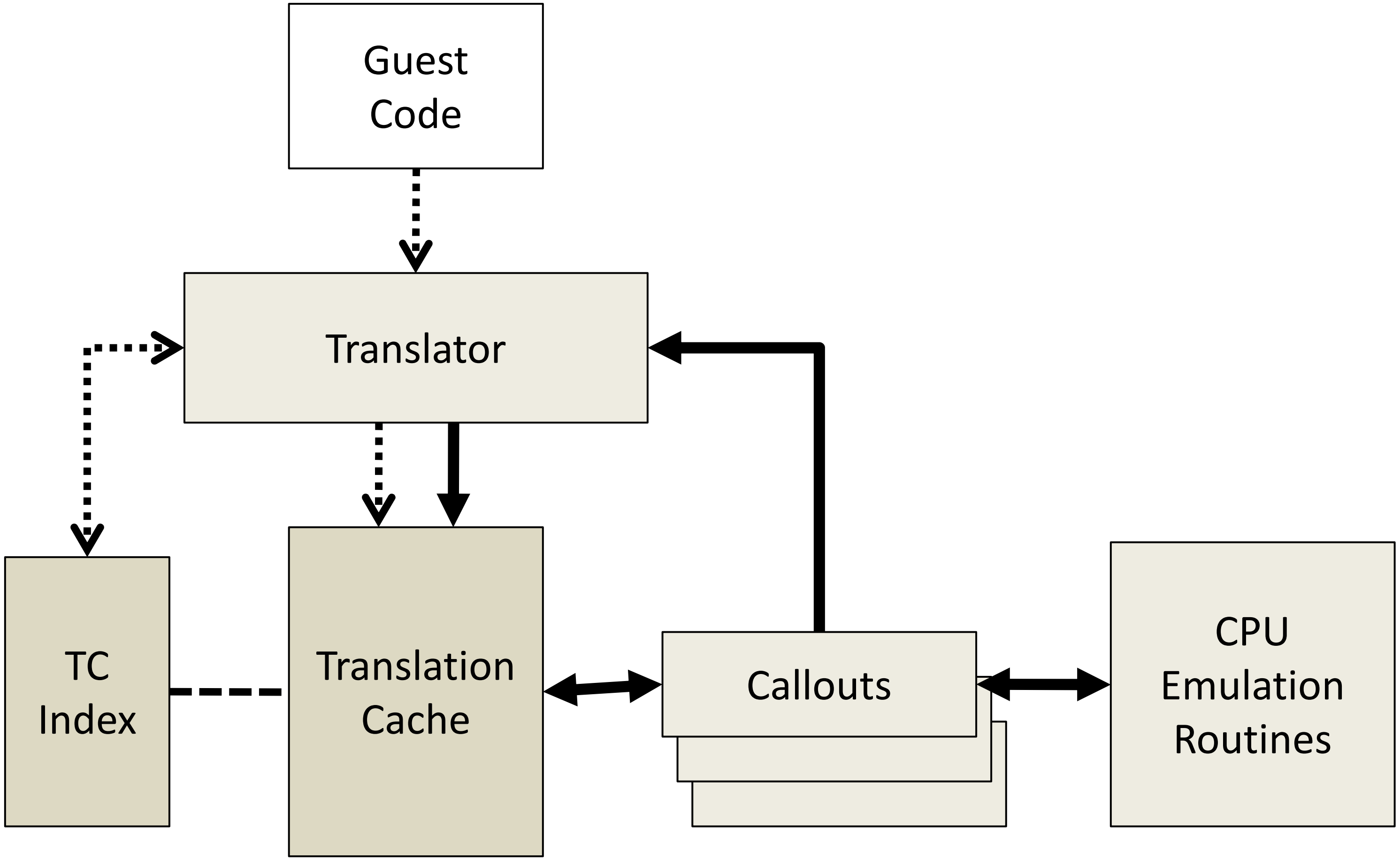
VMM

Guest Kernel
Execution

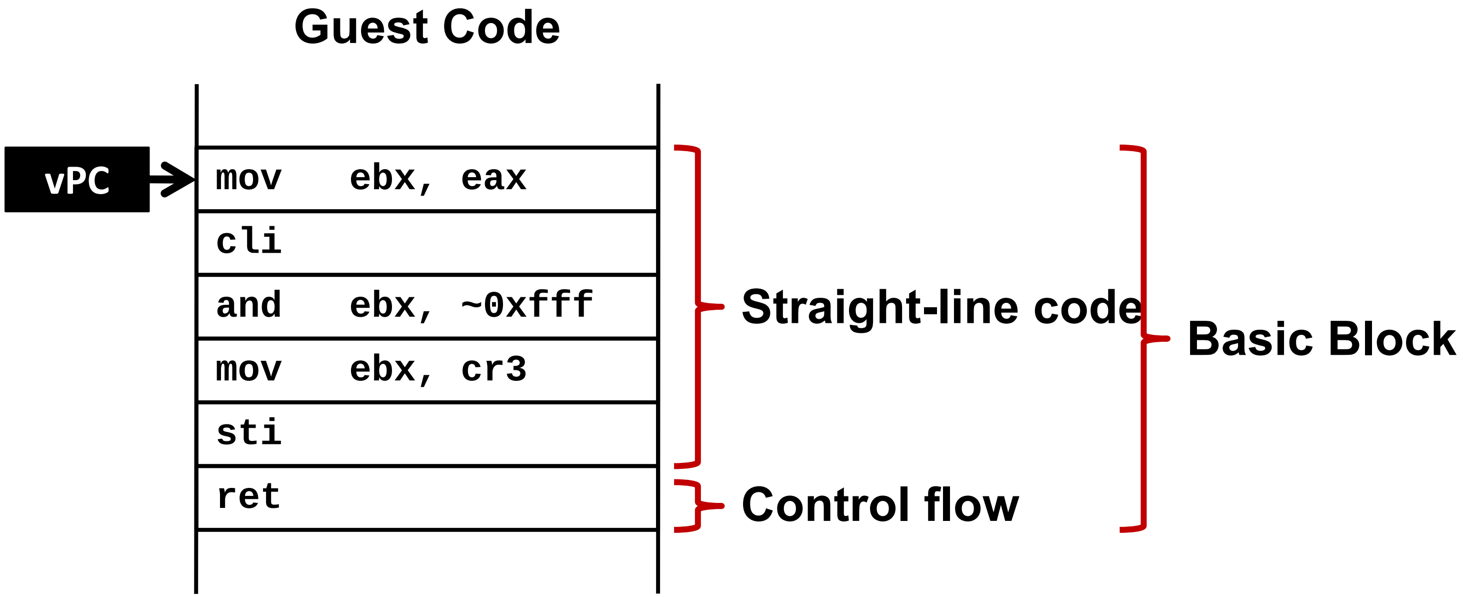
Traces, faults,
interrupts, I/O

Translated Code
(guest kernel)

Binary Translator

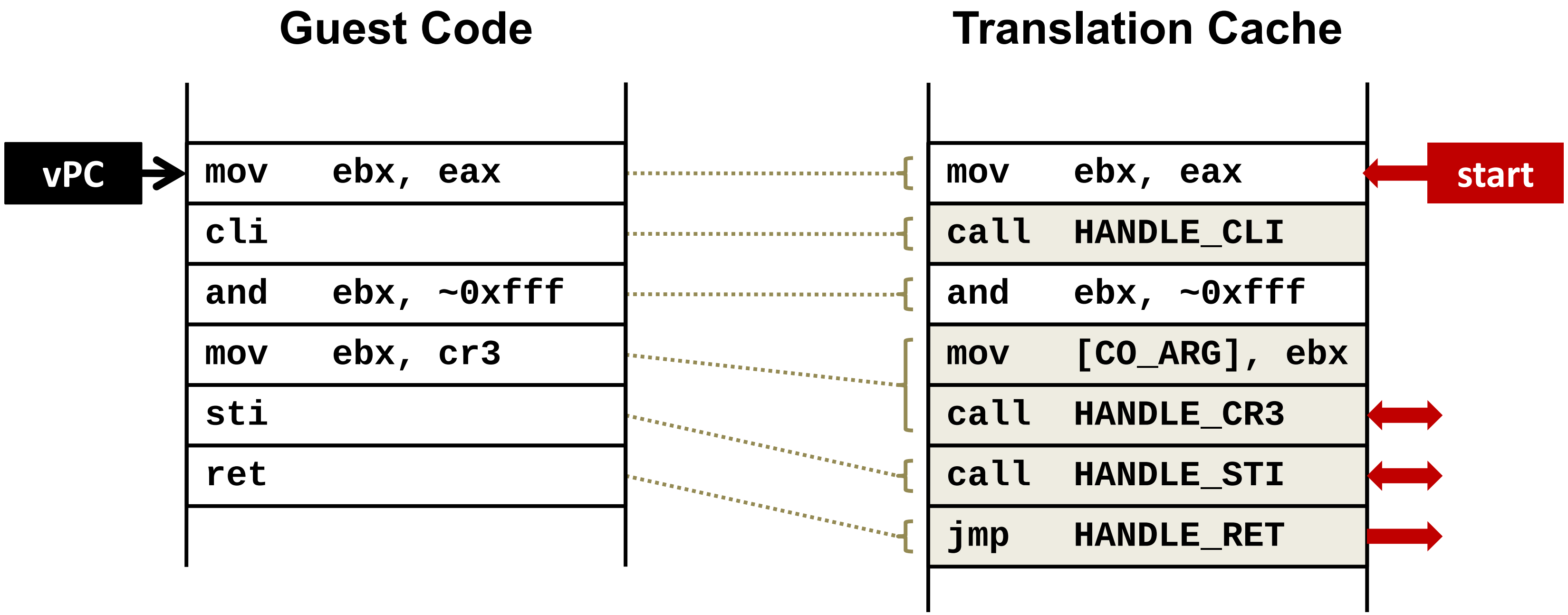


Basic Blocks



```
mov    ebx, eax
cli
and    ebx, ~0xfff
mov    ebx, cr3
sti
ret
```

Binary Translation



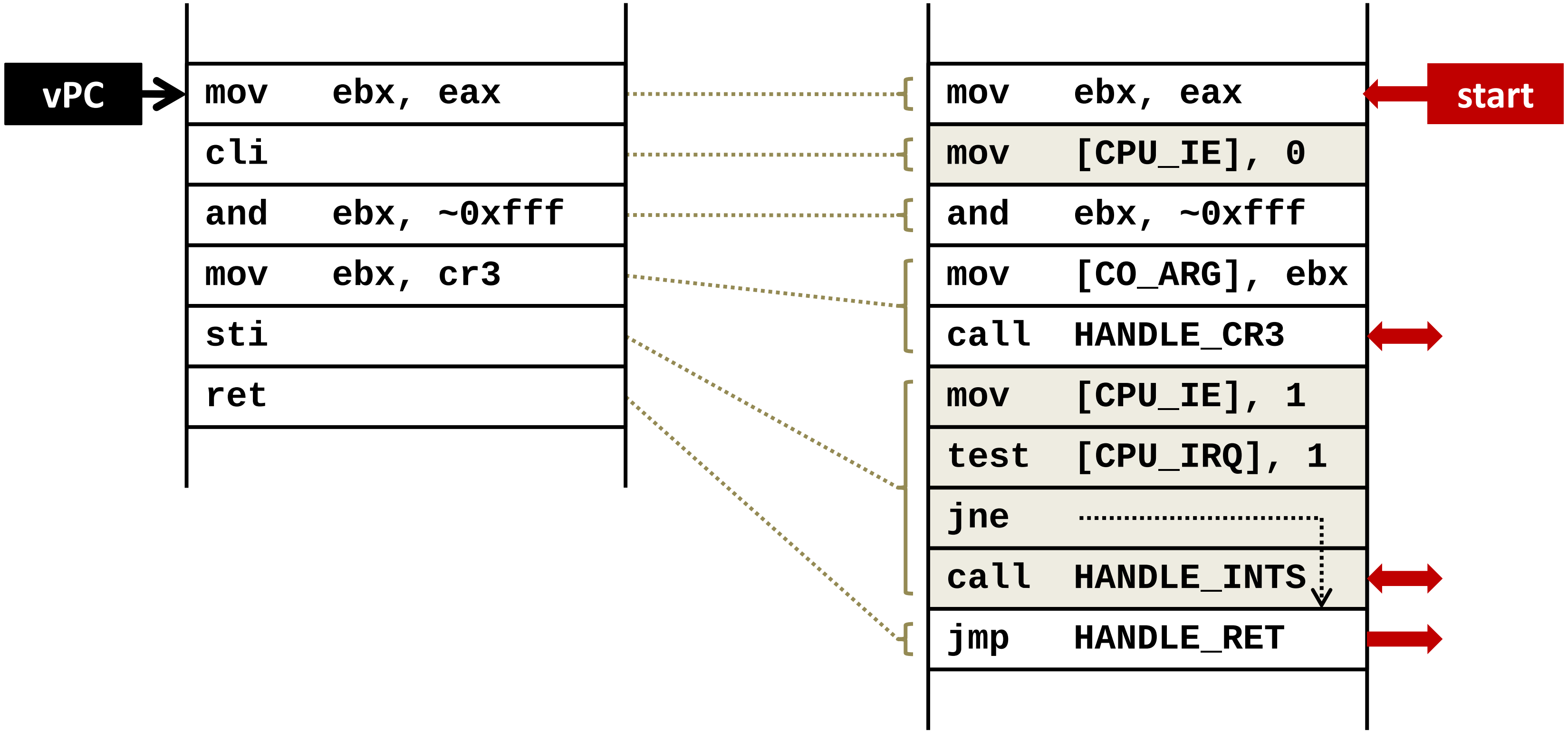
Guest Code

```
mov   ebx, eax
cli
and   ebx, ~0xfff
mov   ebx, cr3
sti
ret
```

Translation Cache

```
mov   ebx, eax
call  HANDLE_CLI
and   ebx, ~0xfff
mov   [CO_ARG], ebx
call  HANDLE_CR3
call  HANDLE_STI
jmp   HANDLE_RET
```

Binary Translation



Guest Code

```
mov     ebx, eax
cli
and     ebx, ~0xfff
mov     ebx, cr3
sti
ret
```

Translation Cache

```
mov     ebx, eax
mov     [CPU_IE], 0
and     ebx, ~0xfff
mov     [CO_ARG], ebx
call    HANDLE_CR3
mov     [CPU_IE], 1
test    [CPU_IRQ], 1
jne
call    HANDLE_INTS
jmp     HANDLE_RET
```

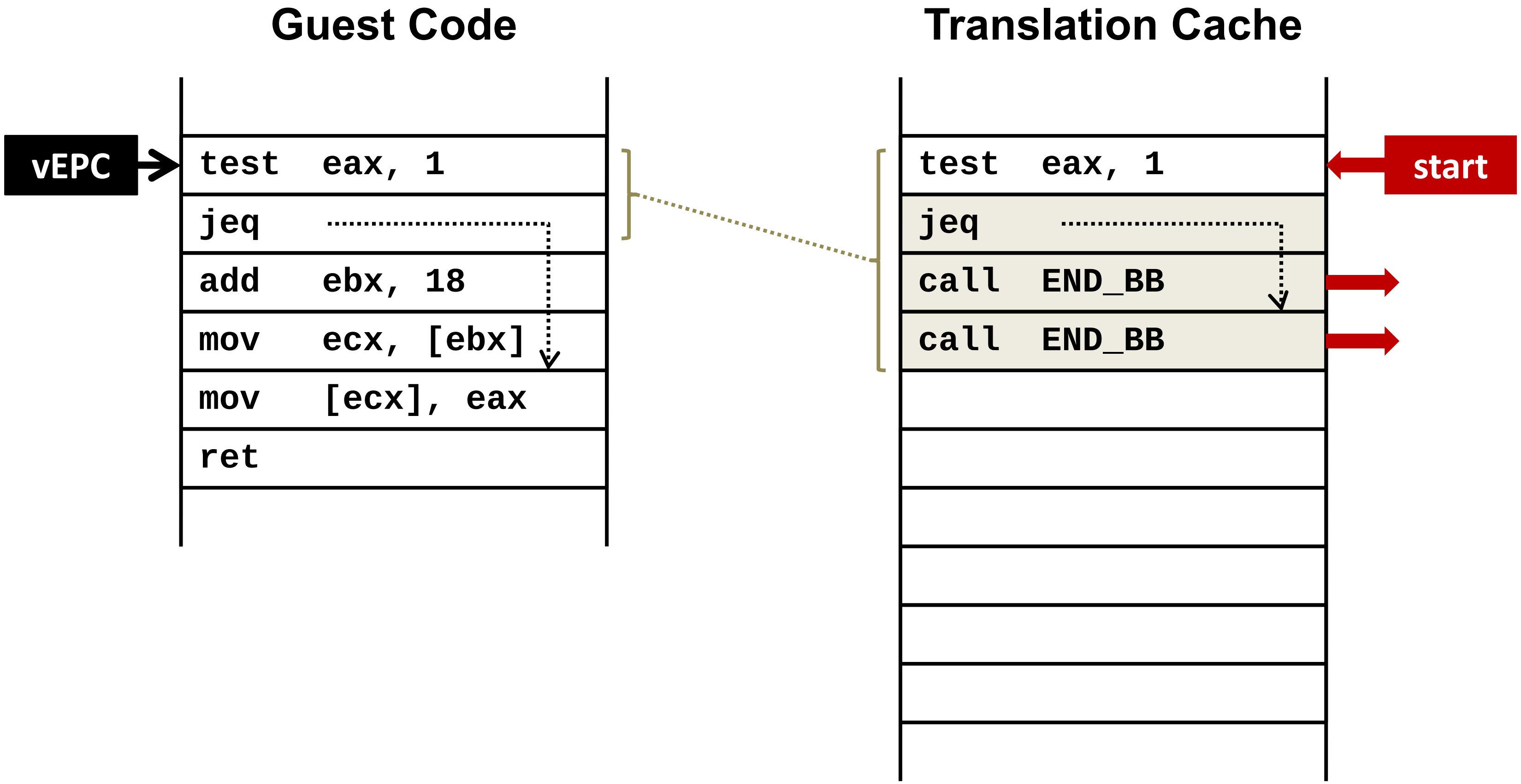
Basic Binary Translator

```
void BT_Run(void)
{
   CPUState.PC = _start;
   BT_Continue();
}

void BT_Continue(void)
{
   void *tcpc;

   tcpc = BTFindBB(CPUState.PC);

   if (!tcpc) {
      tcpc = BTTranslate(CPUState.PC);
   }

   RestoreRegsAndJump(tcpc);
}







void *BTTranslate(uint32 pc)
{
   void *start = TCTop;
   uint32 TCPC = pc;
```

Basic Binary Translator – Part 2

```
void BT_CalloutSTI(BTSavedRegs regs)
{
   CPUState.PC = BTFindPC(regs.tcpc);
   CPUState.GPR[] = regs.GPR[];

   CPU_STI();

   CPUState.PC += 4;

   if (CPUState.IRQ
         && CPUState.IE) {
      CPUVector();
      BT_Continue();
      /* NOT_REACHED */
   }

   return;
}
```

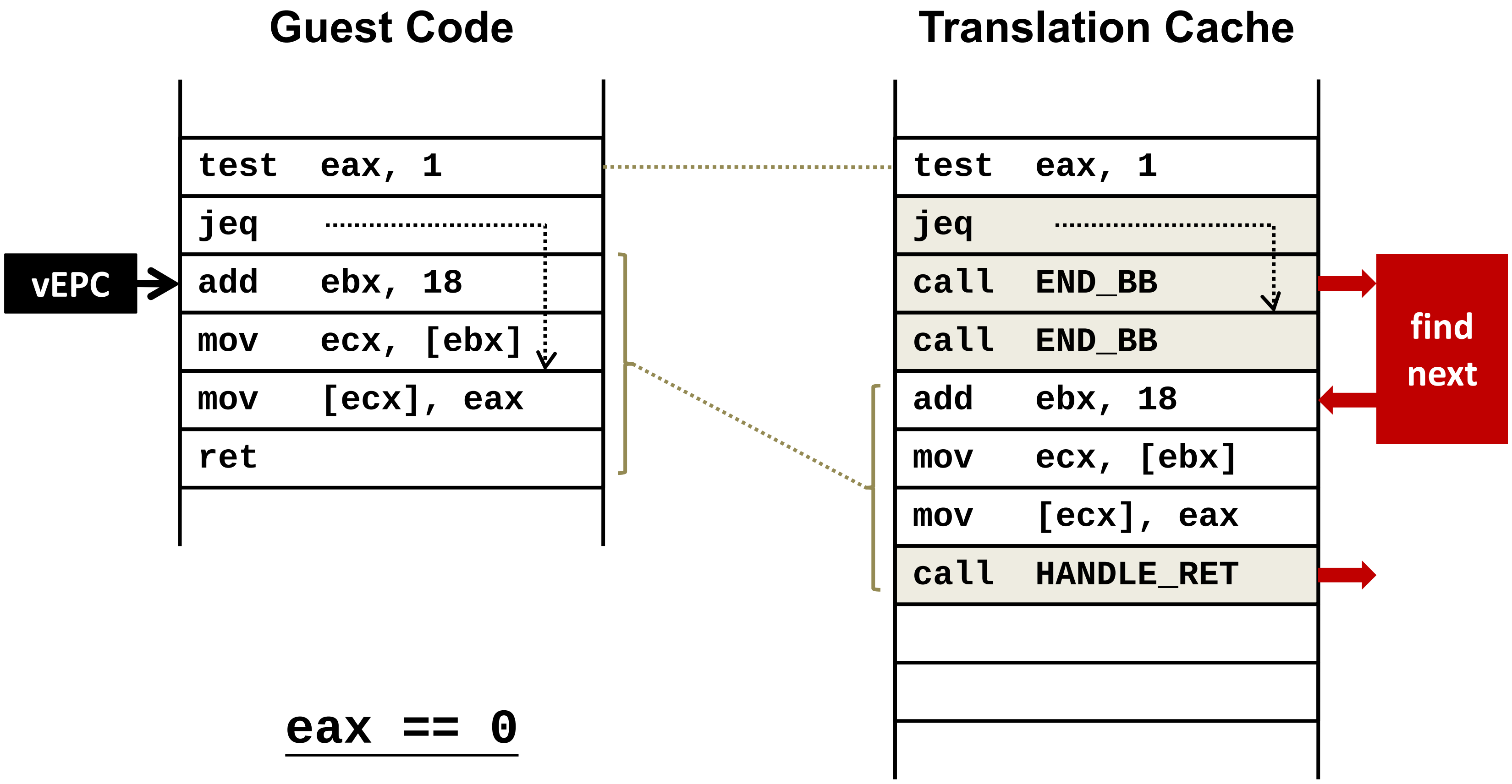
Controlling Control Flow



```
test  eax, 1
jeq
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
ret
```

```
test  eax, 1
jeq
call  END_BB
call  END_BB
```

Controlling Control Flow



Guest Code

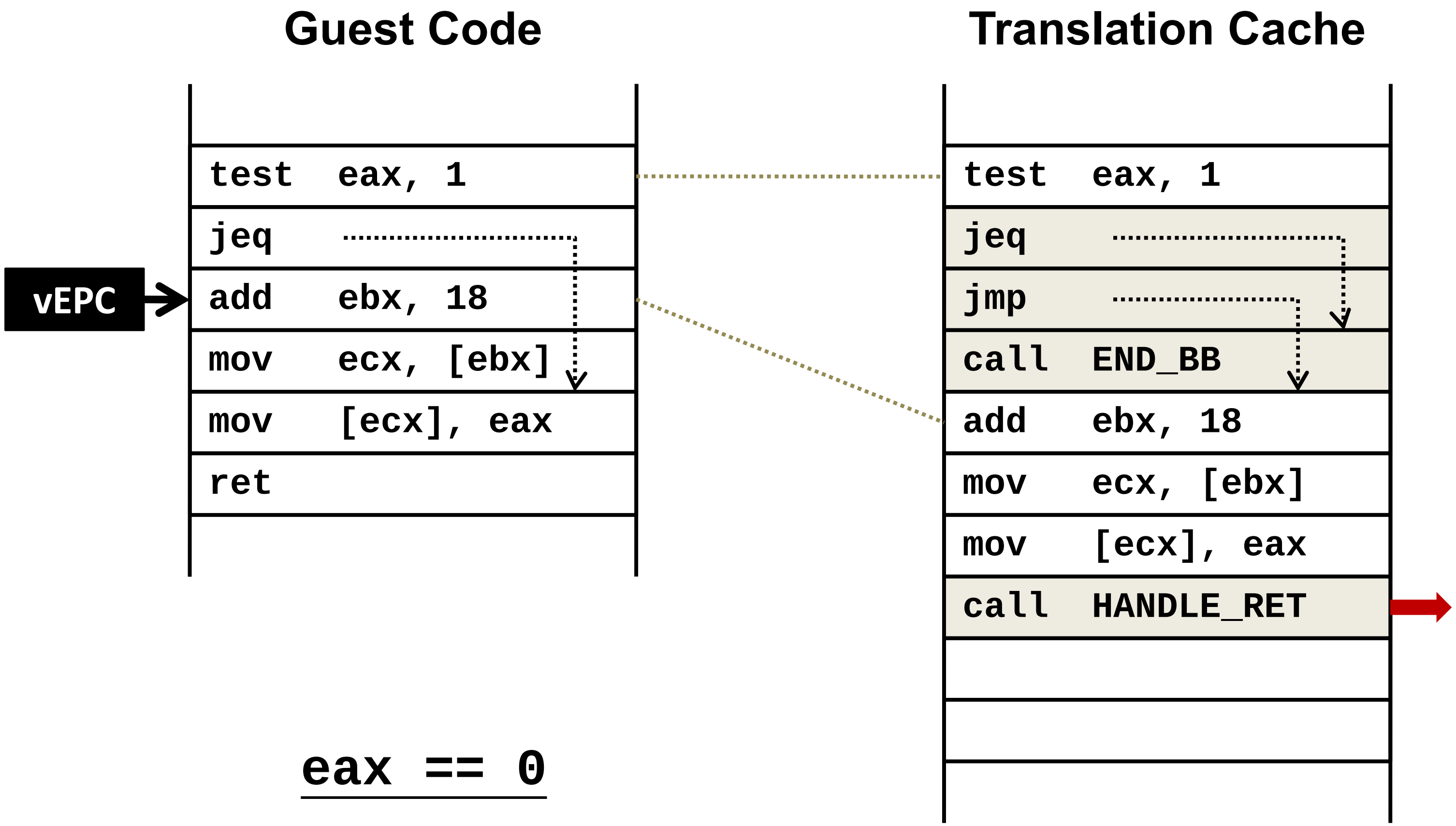
```
test  eax, 1
jeq
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
ret
```

Translation Cache

```
test  eax, 1
jeq
call  END_BB
call  END_BB
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
call  HANDLE_RET
```

eax == 0

Controlling Control Flow



Guest Code

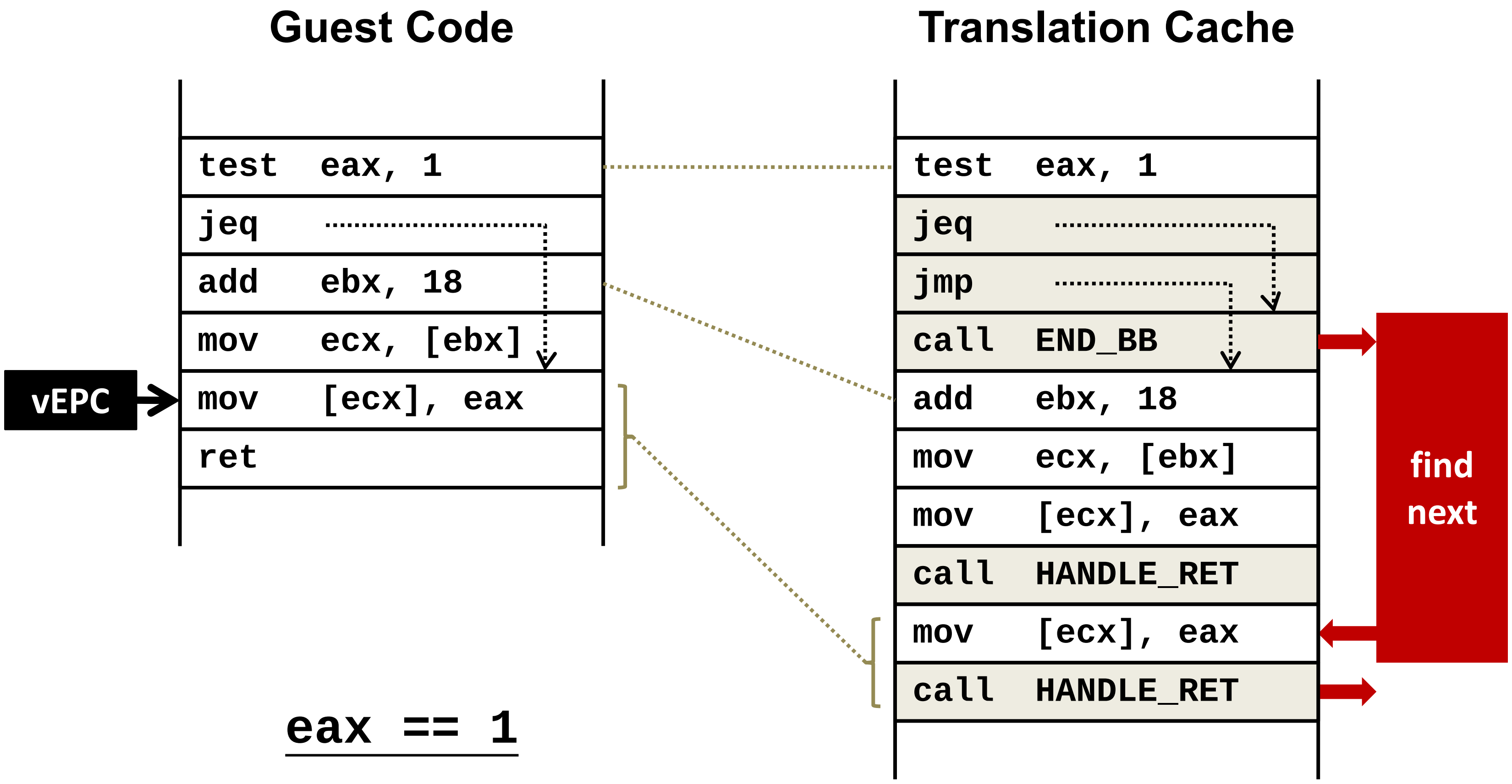
```
test  eax, 1
jeq
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
ret
```

Translation Cache

```
test  eax, 1
jeq
jmp
call  END_BB
add   ebx, 18
mov   ecx, [ebx]
mov   [ecx], eax
call  HANDLE_RET
```

`eax == 0`

Controlling Control Flow



Guest Code

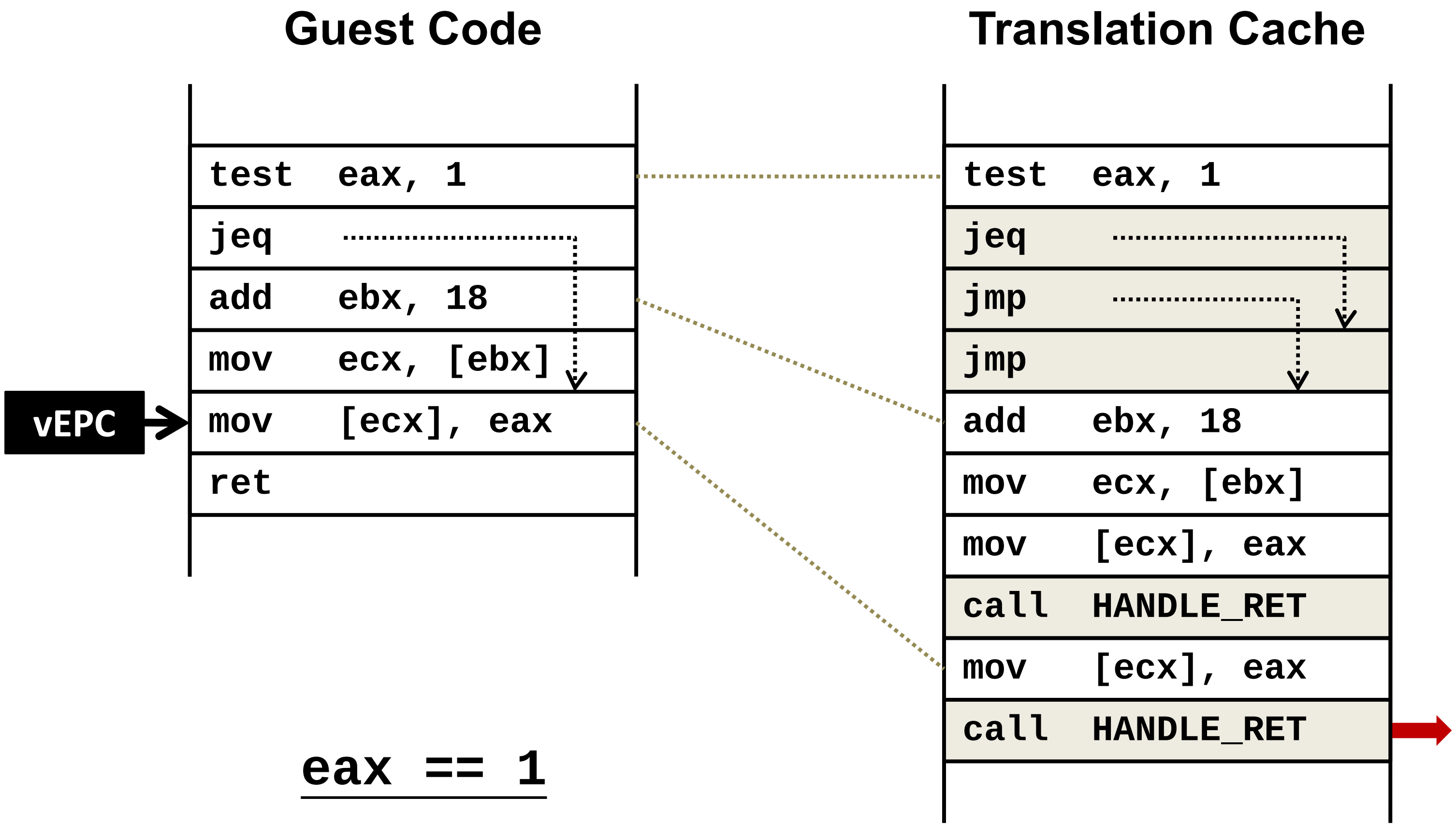
```
test   eax, 1
jeq
add    ebx, 18
mov    ecx, [ebx]
mov    [ecx], eax
ret
```

eax == 1

Translation Cache

```
test   eax, 1
jeq
jmp
call   END_BB
add    ebx, 18
mov    ecx, [ebx]
mov    [ecx], eax
call   HANDLE_RET
mov    [ecx], eax
call   HANDLE_RET
```

Controlling Control Flow



Guest Code

```
test   eax, 1
jeq
add    ebx, 18
mov    ecx, [ebx]
mov    [ecx], eax
ret
```

vEPC

Translation Cache

```
test   eax, 1
jeq
jmp
jmp
add    ebx, 18
mov    ecx, [ebx]
mov    [ecx], eax
call   HANDLE_RET
mov    [ecx], eax
call   HANDLE_RET
```

eax == 1

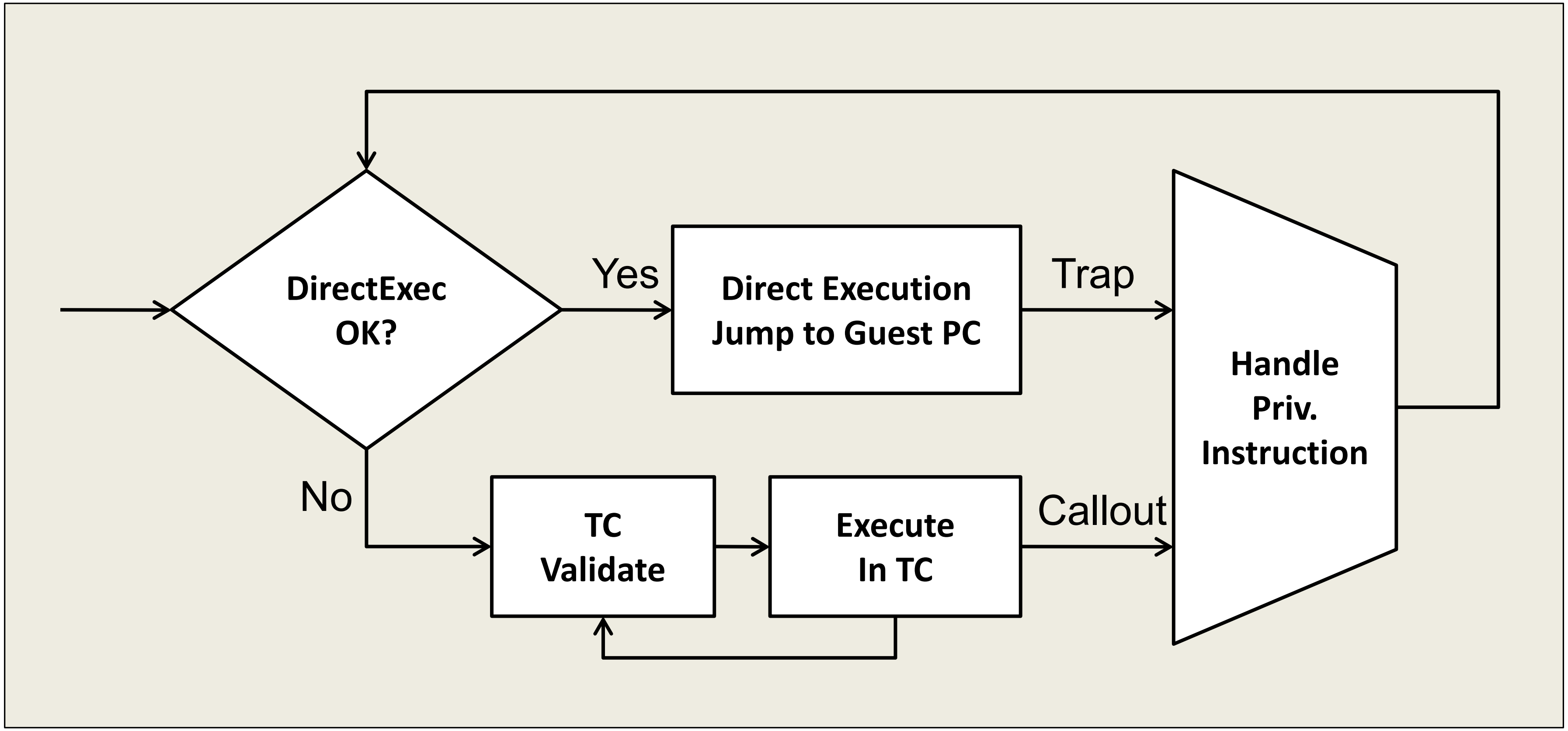
Issues with Binary Translation

- Translation cache index data structure
- PC Synchronization on interrupts
- Self-modifying code
  - Notified on writes to translated guest code

Other Uses for Binary Translation

- Cross ISA translators
  - Digital FX!32
- Optimizing translators
  - H.P. Dynamo
- High level language byte code translators
  - Java
  - .NET/CLI

Hybrid Approach



- Binary Translation for the Kernel
- Direct Execution (Trap-and-emulate) for the User
- U.S. Patent 6,397,242

Para-virtualization (e.g., Xen)

- Modify operating systems to let OS to cooperate with VMM
  - Let OS runs in Ring1 and user apps run in Ring3
  - Change sensitive instructions to explicit calls to the VMM
  - No need to trap and emulate

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 CR3 / Page Tables

- Terminology: 3 types of address now
  - GVA->GPA->HPA (Guest virtual. Guest physical. Host physical)
  - Guest VM's page table contains guest physical address
  - Hardware page table must point to host physical address
- Setting CR3 to point to guest page table would not work
  - Processes in guest VM might access host physical address 0~1GB
  - But those HPA might not belong to that guest VM

One Solution: Shadow Pages

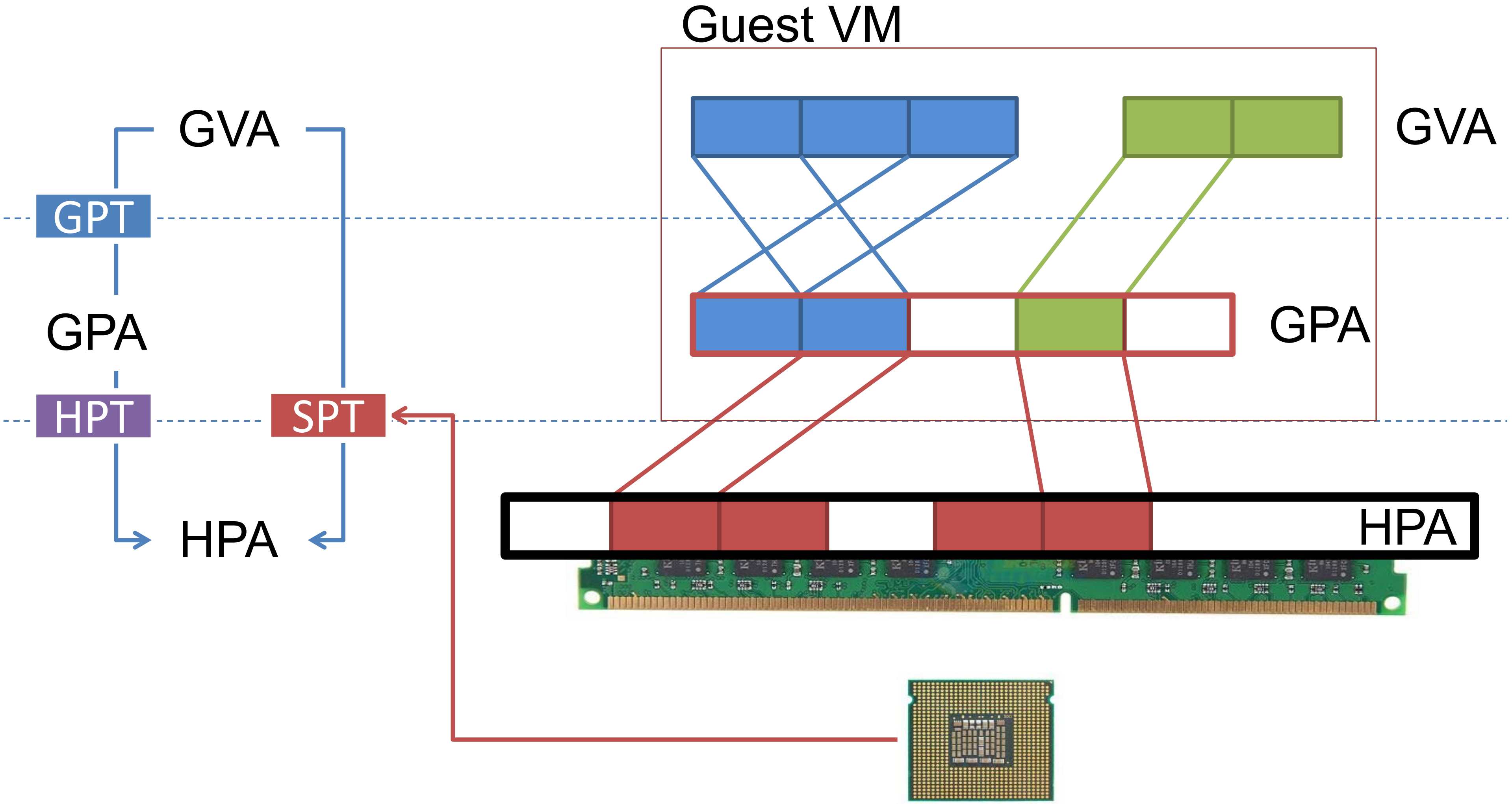
1. VMM intercepts guest OS setting the virtual CR3
2. VMM iterates over the guest page table, constructs a corresponding shadow PT
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow PT

(Shadow PT is per process)

Shadow Page Table

31 12	11 9	8	7	6	5	4	3	2	1	0
Physical-Page Base Address	AVL	G	P A T	D	A	P C D	P W T	U / S	R / W	P

```
set_ptp(guest_pt):
    for gva in 0 .. $2^{20}$:
        if guest_pt[gva] & PTE_P:
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt
```


Guest VM
GVA
GPT
GPA
HPT
SPT
HPA
GVA
GPA
HPA

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

31 12	11 9	8	7	6	5	4	3	2	1	0
Physical-Page Base Address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

```
set_ptp(guest_pt, kmode):
    for gva in 0 .. $2^{20}$:
        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
```

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)

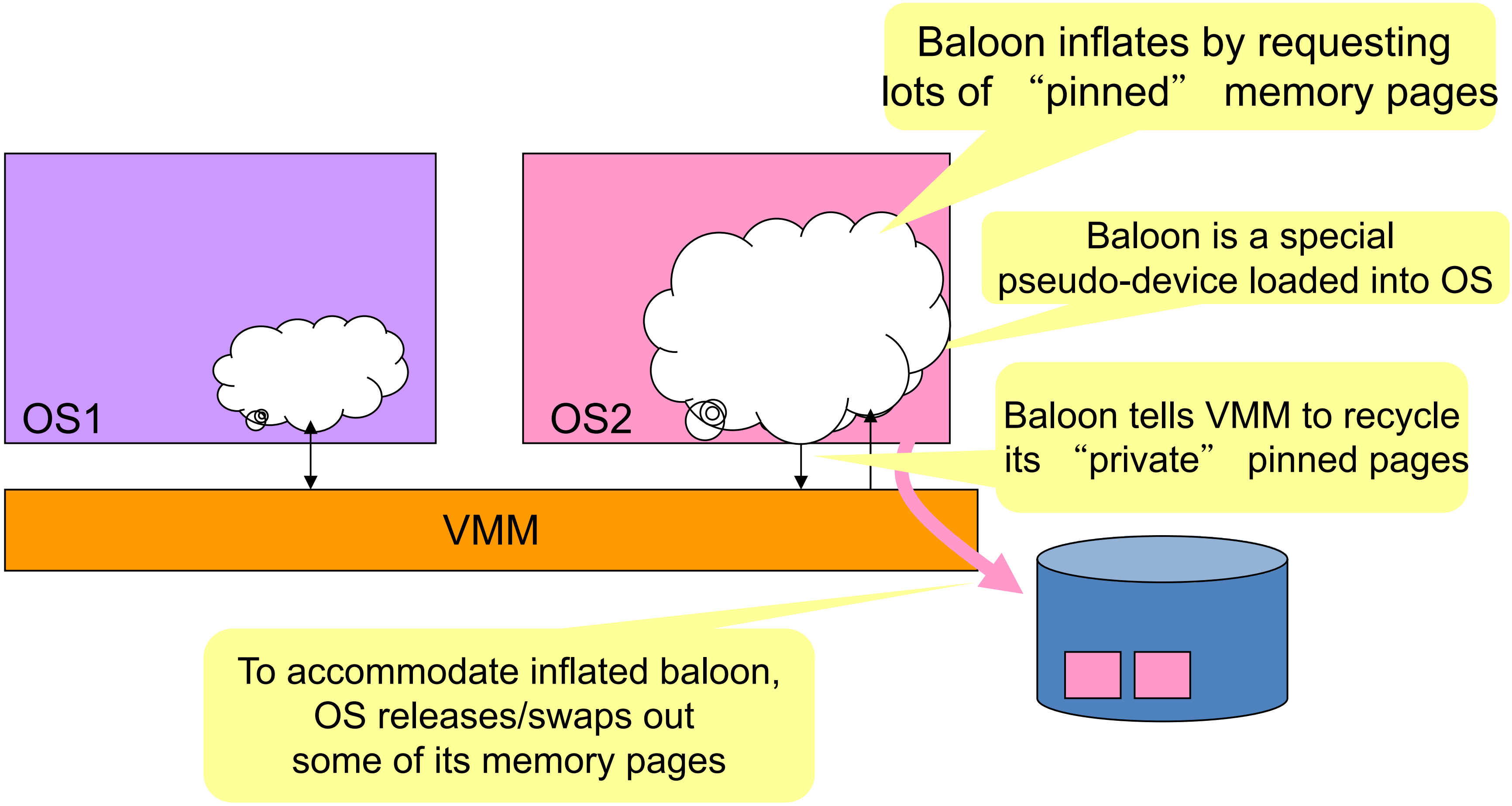
Managing Memory in VMM

- 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 Balooning



Baloon inflates by requesting lots of “pinned” memory pages

Baloon is a special pseudo-device loaded into OS

Baloon tells VMM to recycle its “private” pinned pages

To accommodate inflated baloon, OS releases/swaps out some of its memory pages

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

System Virtualization

Yubin Xia & Rong Chen

Software School

Shanghai Jiao Tong University

Some Slides adapted from
VMWare’s academic course plan

Review: CPU Virtualization

- Instruction Interpretation
- Trap-and-emulate
- Binary Translation
- Para-virtualization

MEMORY VIRTUALIZATION

Review: 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

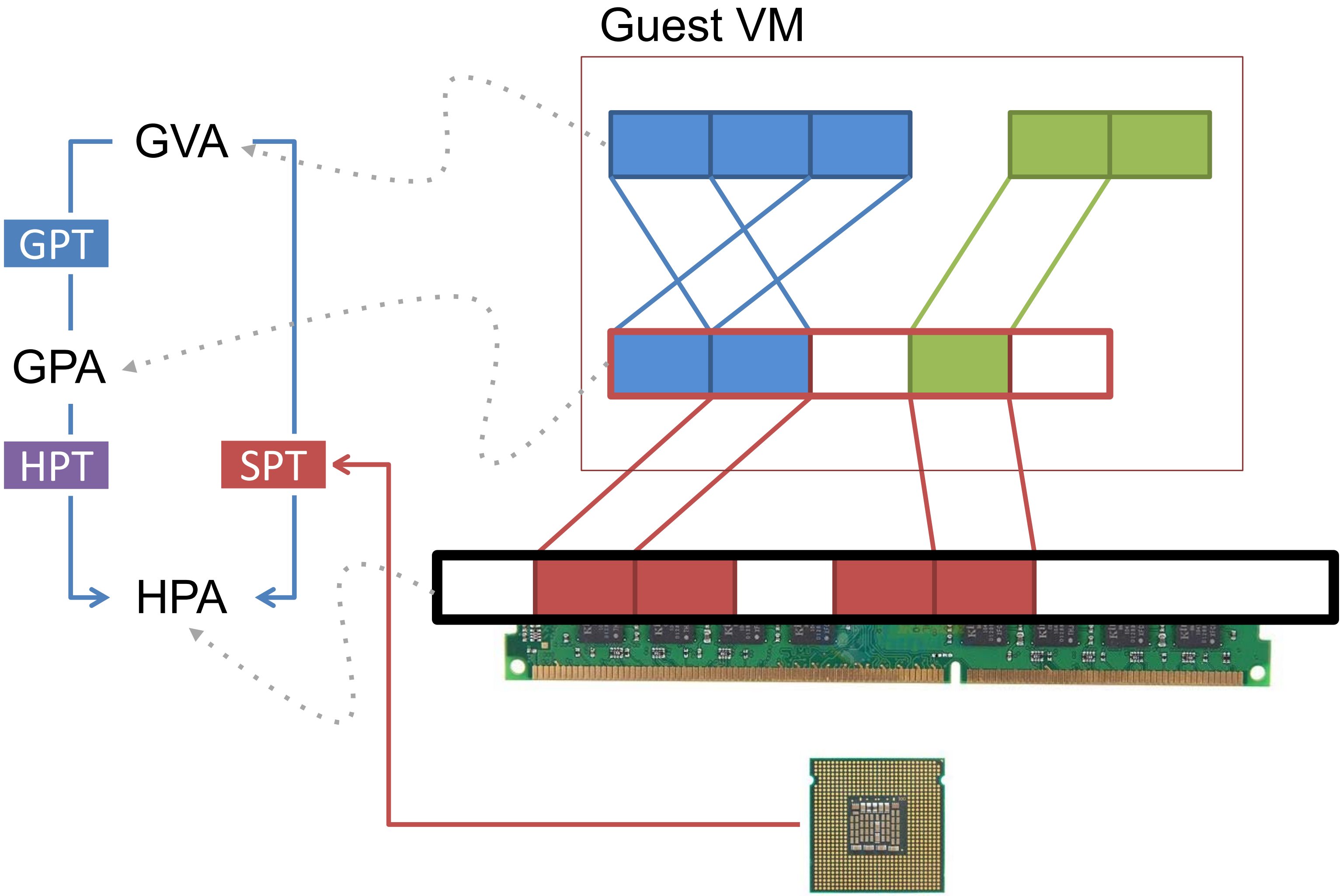
Review: Shadow Pages

- Shadow Paging
  - VMM intercepts guest OS setting the virtual PTP register
  - VMM iterates over the guest page table, constructs a corresponding shadow PT
  - In shadow PT, every guest physical address is translated into host physical address
  - Finally, VMM loads the host physical address of the shadow PT
  - Shadow PT is per process

Shadow Page Table

31 12	11 9	8	7	6	5	4	3	2	1	0
Physical-Page Base Address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

```
set_ptp(guest_pt):
    for gva in 0 .. $2^{20}$:
        if guest_pt[gva] & PTE_P:
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt
```


Guest VM
GVA
GPT
GPA
HPT
SPT
HPA

Protect Kernel-only Pages

31 12	11 9	8	7	6	5	4	3	2	1	0
Physical-Page Base Address	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

```
set_ptp(guest_pt, kmode):
    for gva in 0 .. $2^{20}$:
        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
```

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)

Review: Hardware Support for Memory Virtualization

- Hardware Support for Memory Virtualization
  - 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

Hardware-assisted Virtualization

VT-x Modes

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

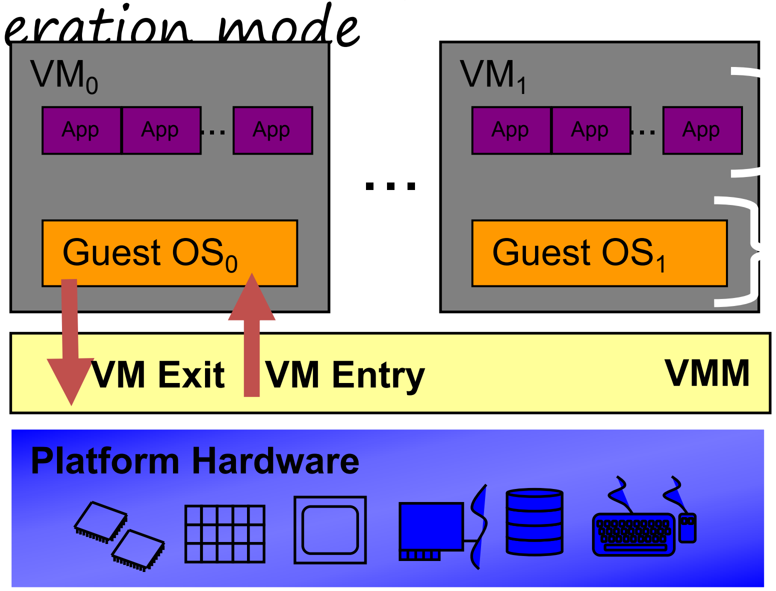
VT-x transition mechanisms

VM exit

- From VMX non-root operation mode to VMX root operation mode

VM entry

- from VMX root operation mode to VMX non-root operation mode



VTX non root mode / Ring 3

VTX non root mode / Ring 0

VTX root mode / Ring 0

Virtual Machine Control Structure (VMCS)

Data structure that manages VM entries and VM exits.

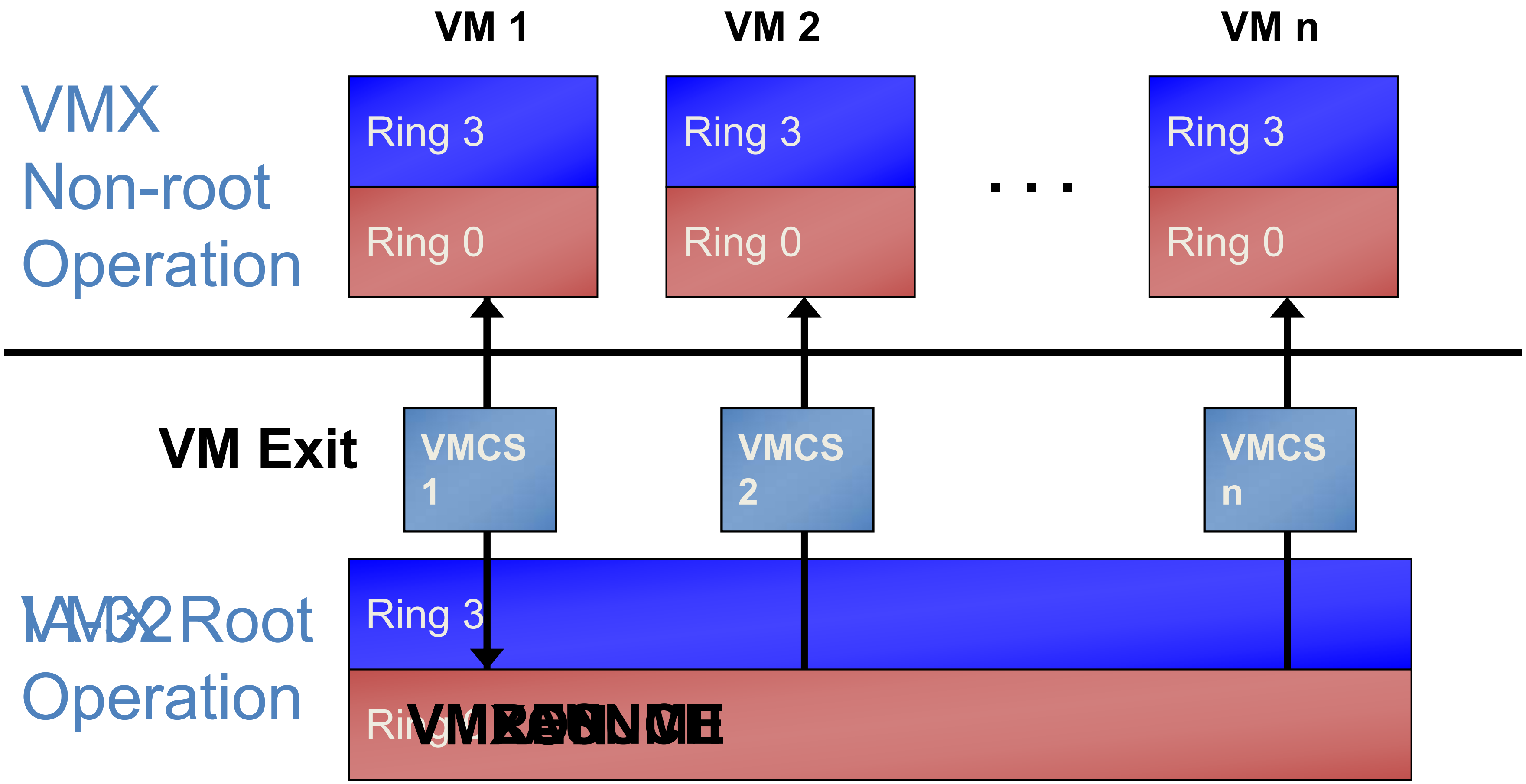
VMCS is logically divided into:

- Guest-state area
- Host-state area.
- VM-execution control fields
- VM-exit control fields
- VM-entry control fields
- VM-exit information fields

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.

VT-x Operations



VM 1

VM 2

VM n

VMX Non-root Operation

Ring 3

Ring 0

Ring 3

Ring 0

. . .

Ring 3

Ring 0

VM Exit

VMCS 1

VMCS 2

VMCS n

IA-32

VMX Root Operation

Ring 3

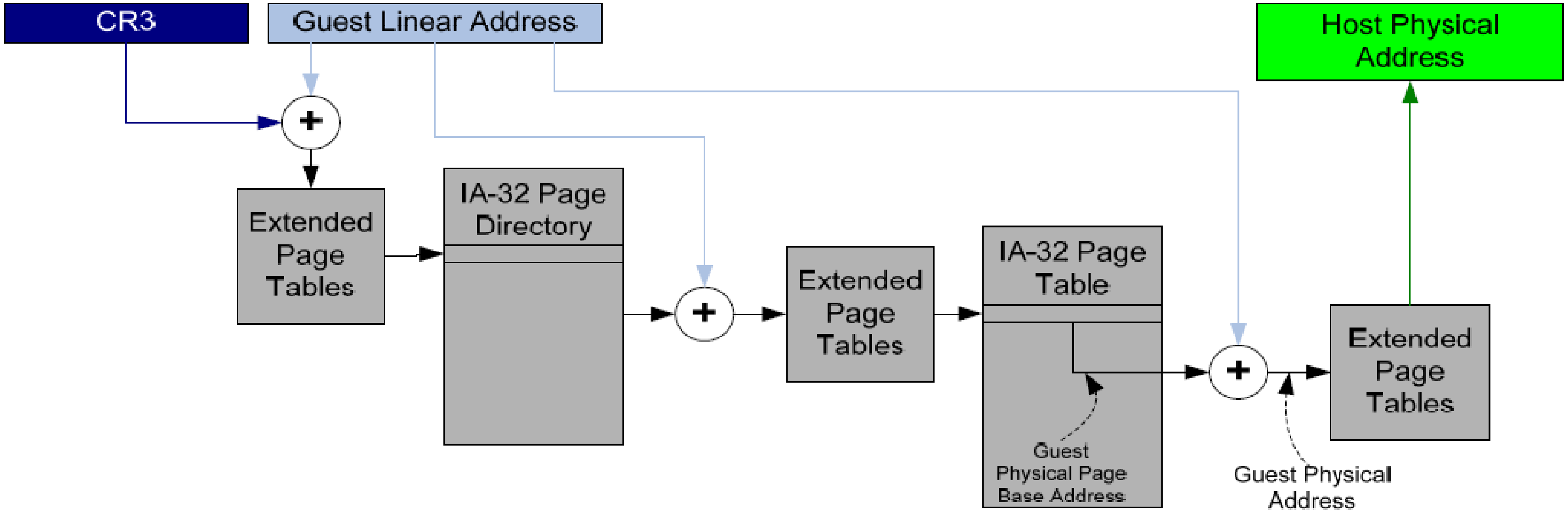
Ring 0

VMXON

VMLAUNCH

VMRESUME

VT-x extension: Extended Page Table (EPT)

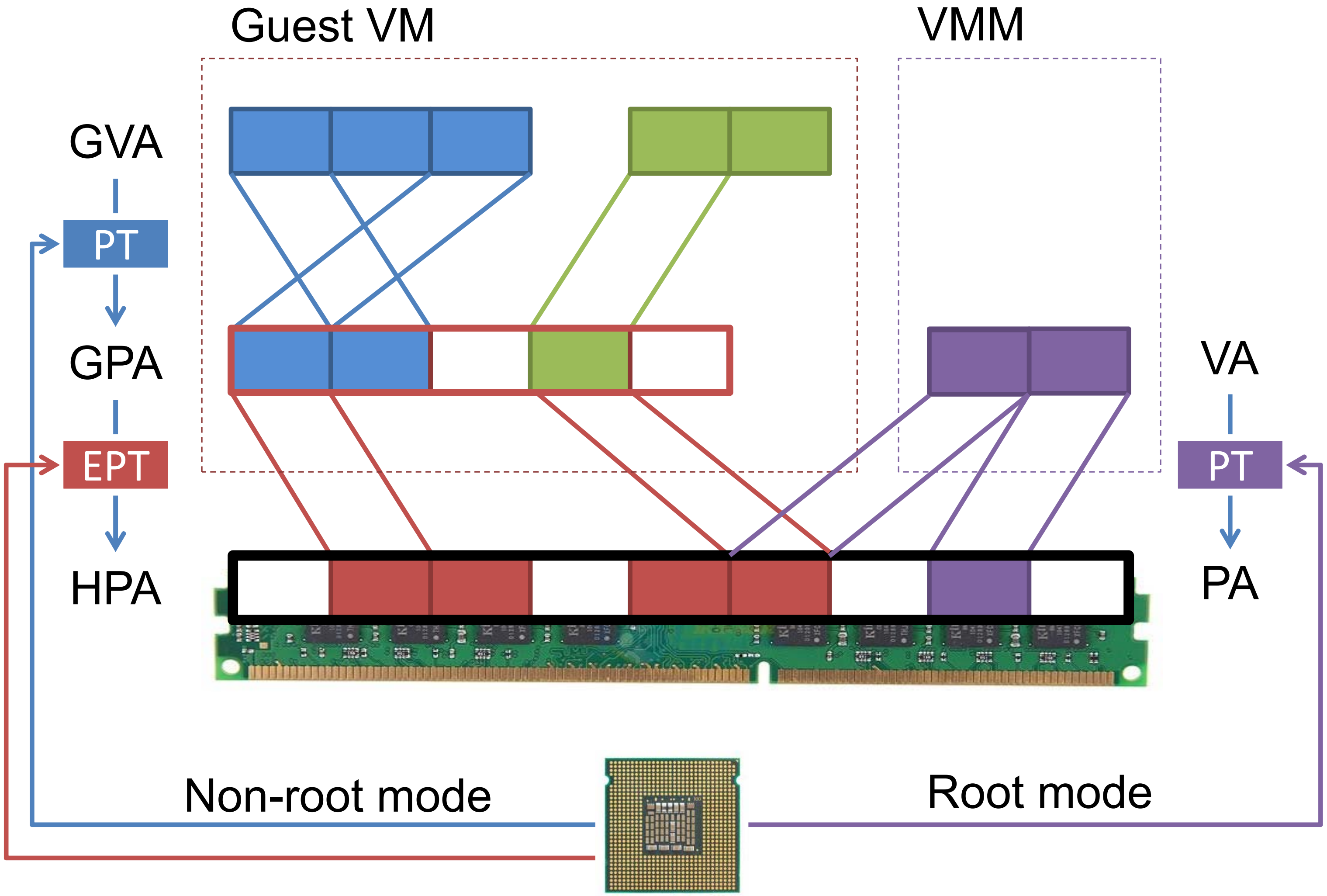


All guest-physical addresses go through extended page tables

Includes address in CR3, address in PDE, address in PTE, etc.

Reduces the frequency of VM exits to VMM.

The net effect of both implementations (EPT or NPT) is to allow the guest OS to own and manage its own page table, and not force the host to get involved.


Guest VM
VMM
GVA
PT
GPA
EPT
HPA
VA
PT
PA
Non-root mode
Root mode

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.

MANAGE VM MEMORY

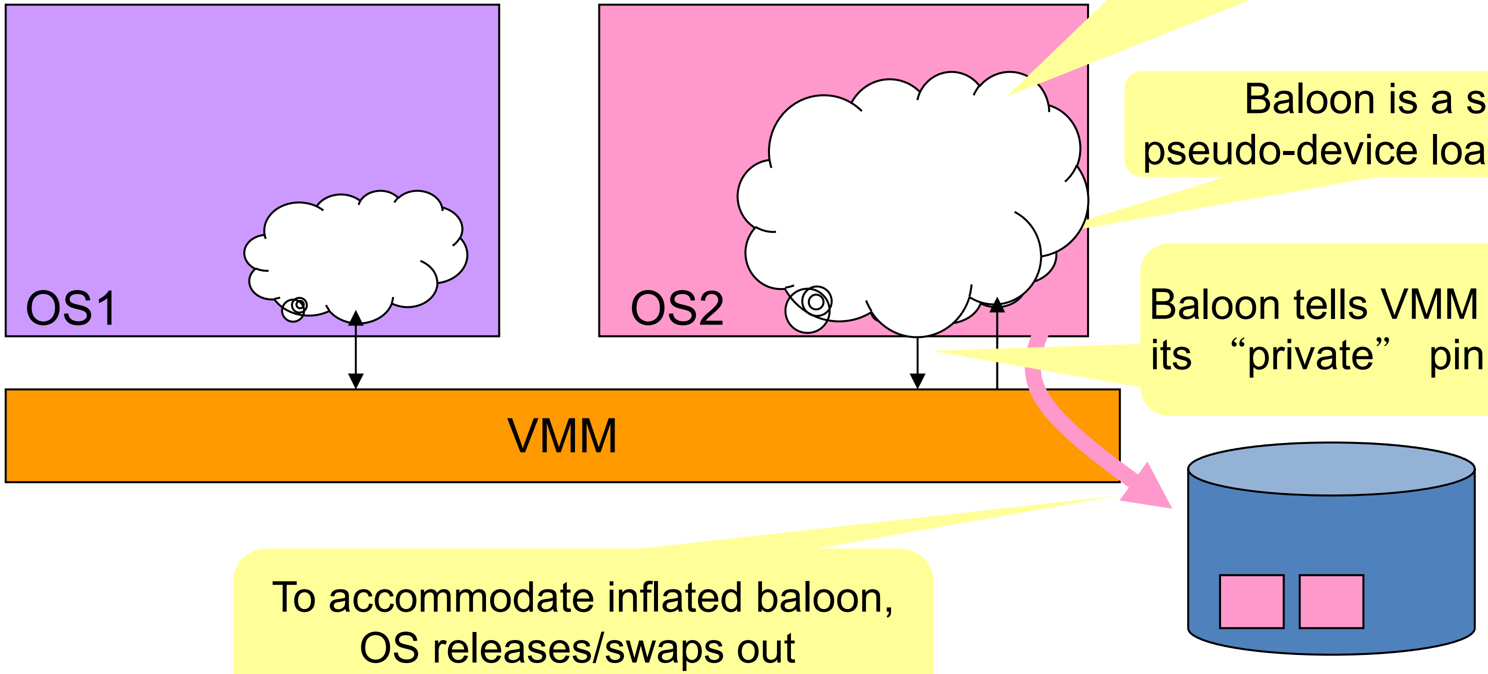
Managing Memory in VMM

- 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 Balooning



Baloon inflates by requesting lots of “pinned” memory pages

Baloon is a special pseudo-device loaded into OS

Baloon tells VMM to recycle its “private” pinned pages

To accommodate inflated baloon, OS releases/swaps out some of its memory pages

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

I/O VIRTUALIZATION

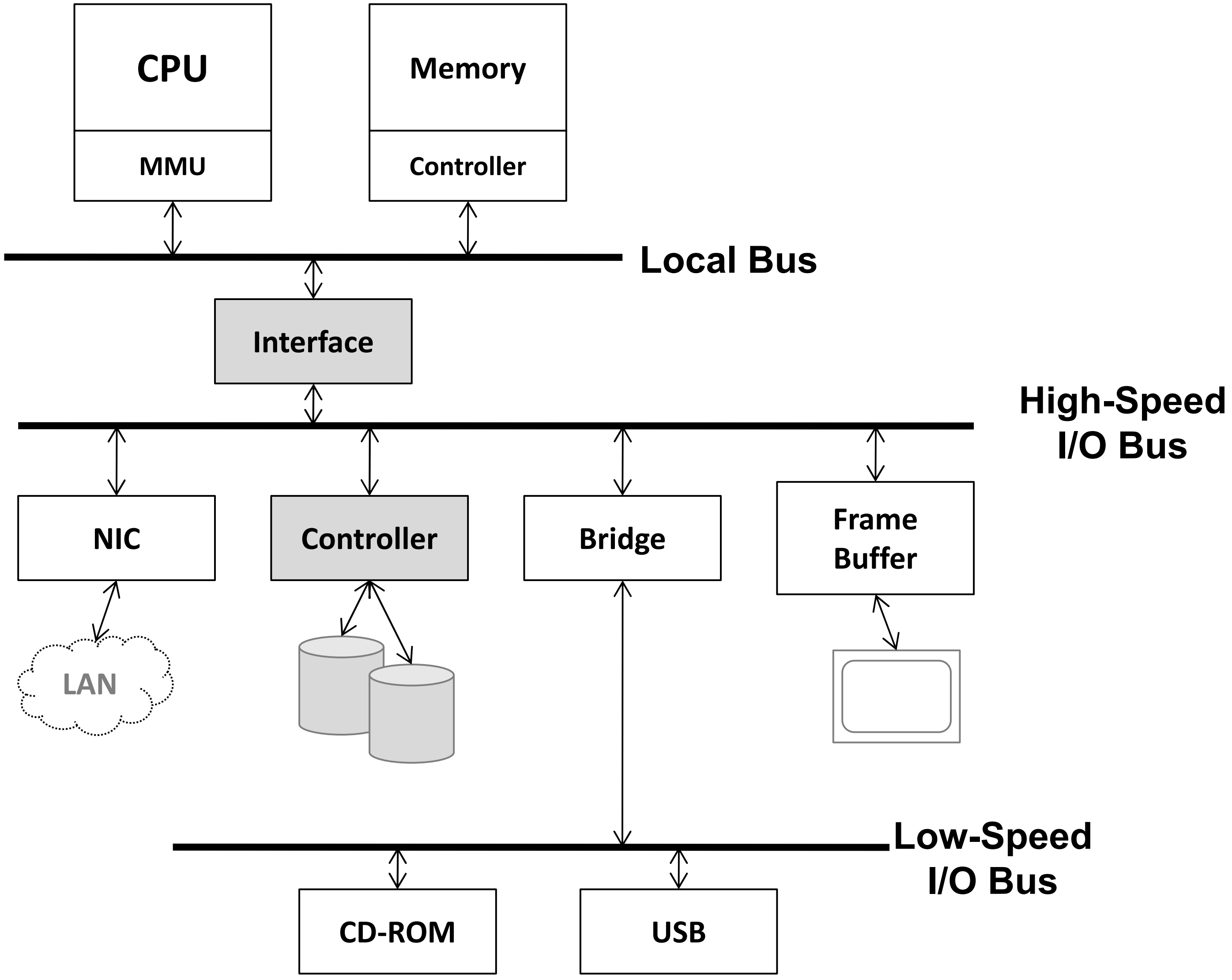
Outline

Types of Device Virtualization

- Direct Access
- Emulated
- Para-virtualized

Hardware assisted I/O virtualization

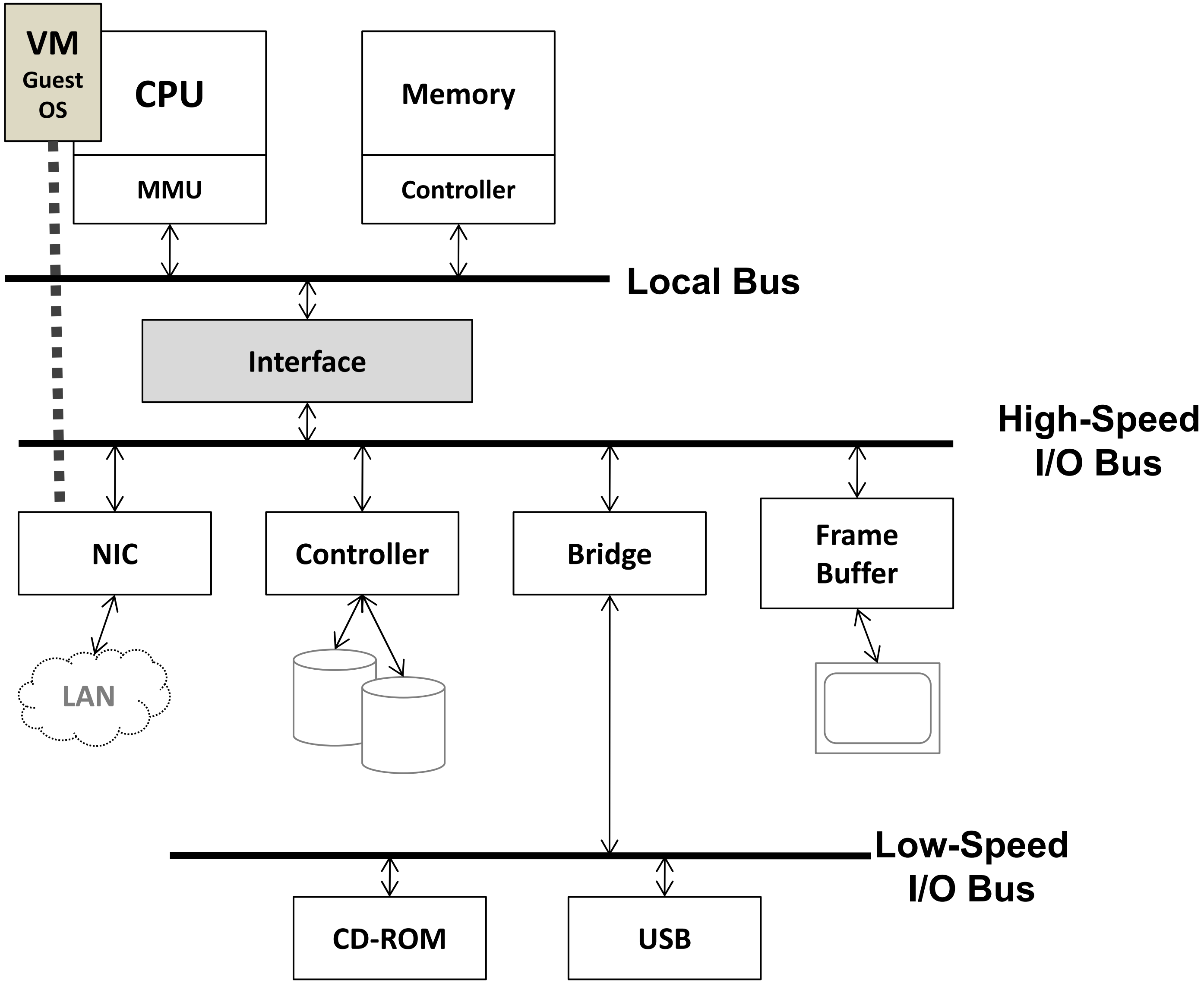
Computer System Organization



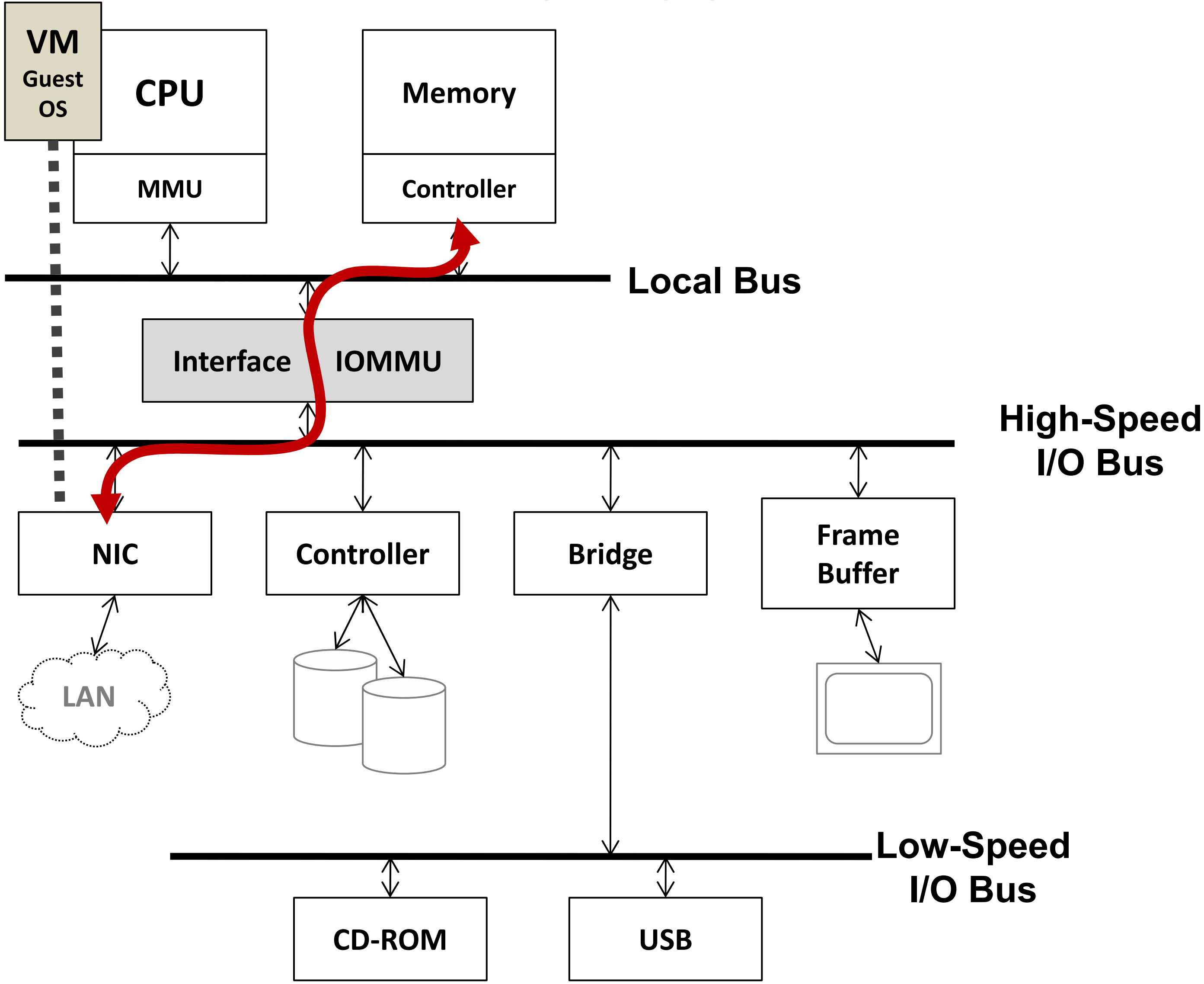
Device Virtualization

- Goals
  - Isolation
  - Multiplexing
  - Speed
  - Mobility
  - Interposition
- Device Virtualization Strategies
  - Direct Access
  - Emulation
  - Para-virtualization

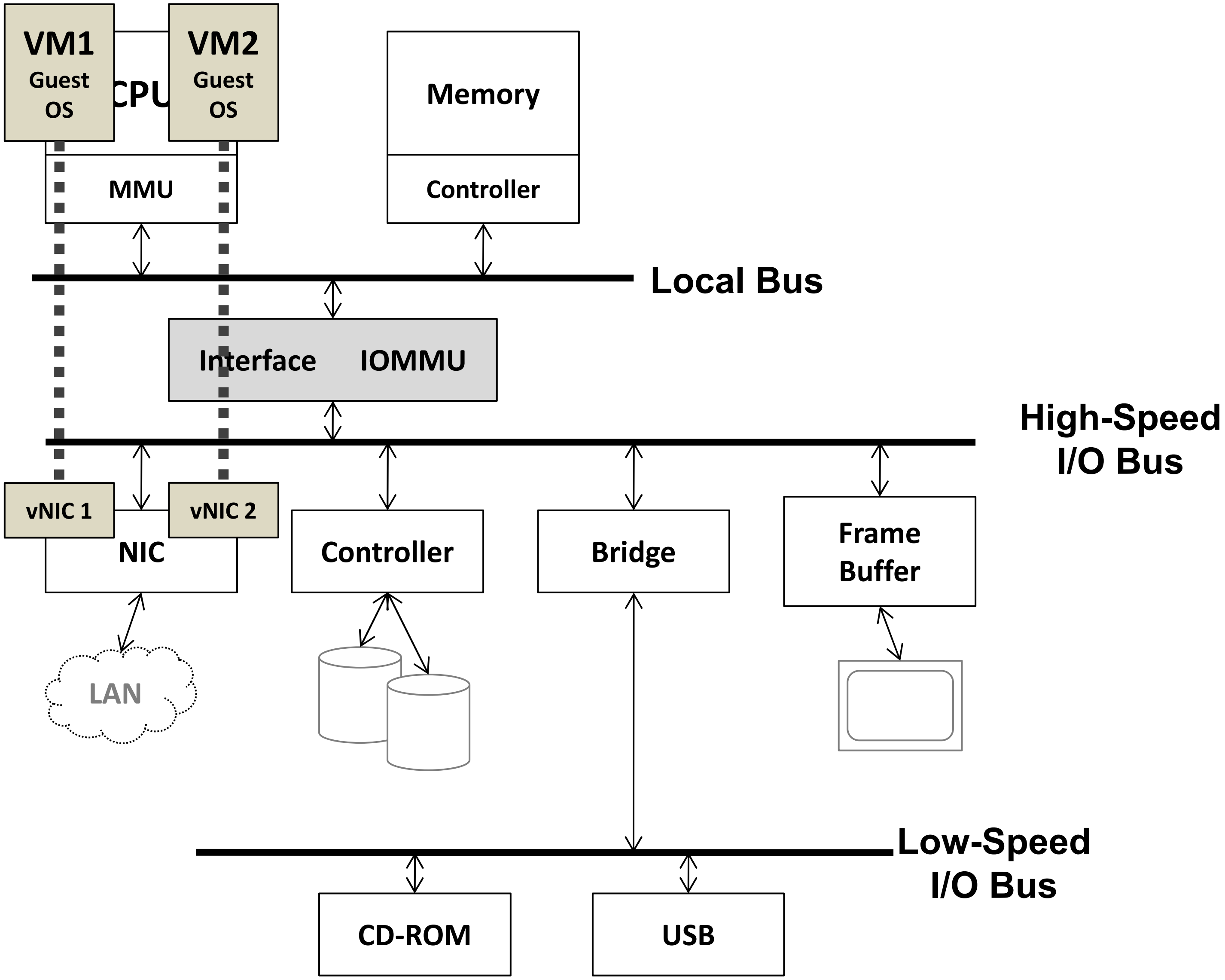
Direct Access Device



Memory Isolation w/ Direct Access Device



Virtualization Enabled Device



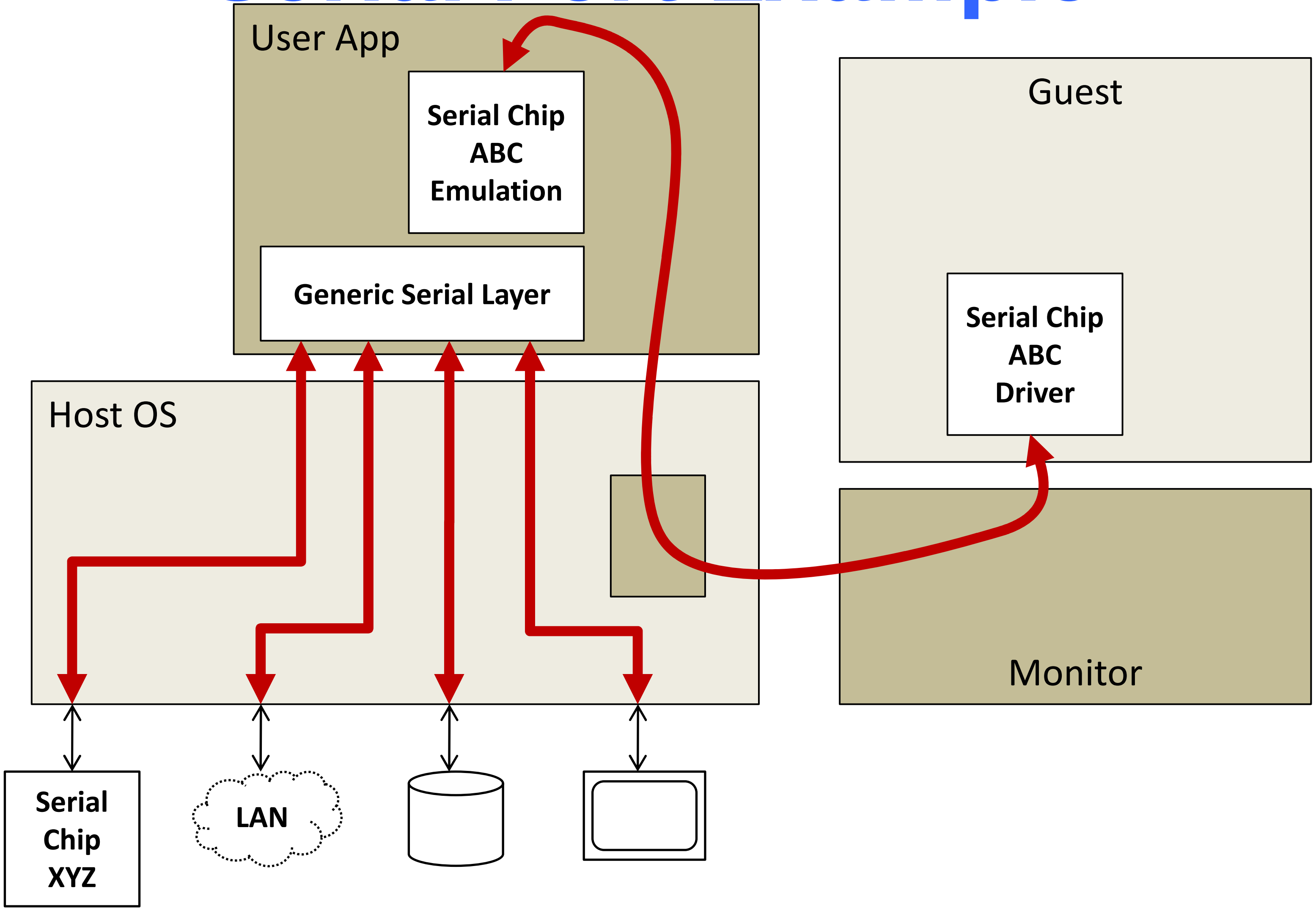
Direct Access Device Virtualization

- 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

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



Emulated Devices

Positives

- Platform stability
- Allows interposition
- No special hardware support needed
  - Isolation, multiplexing implemented by monitor

Negatives

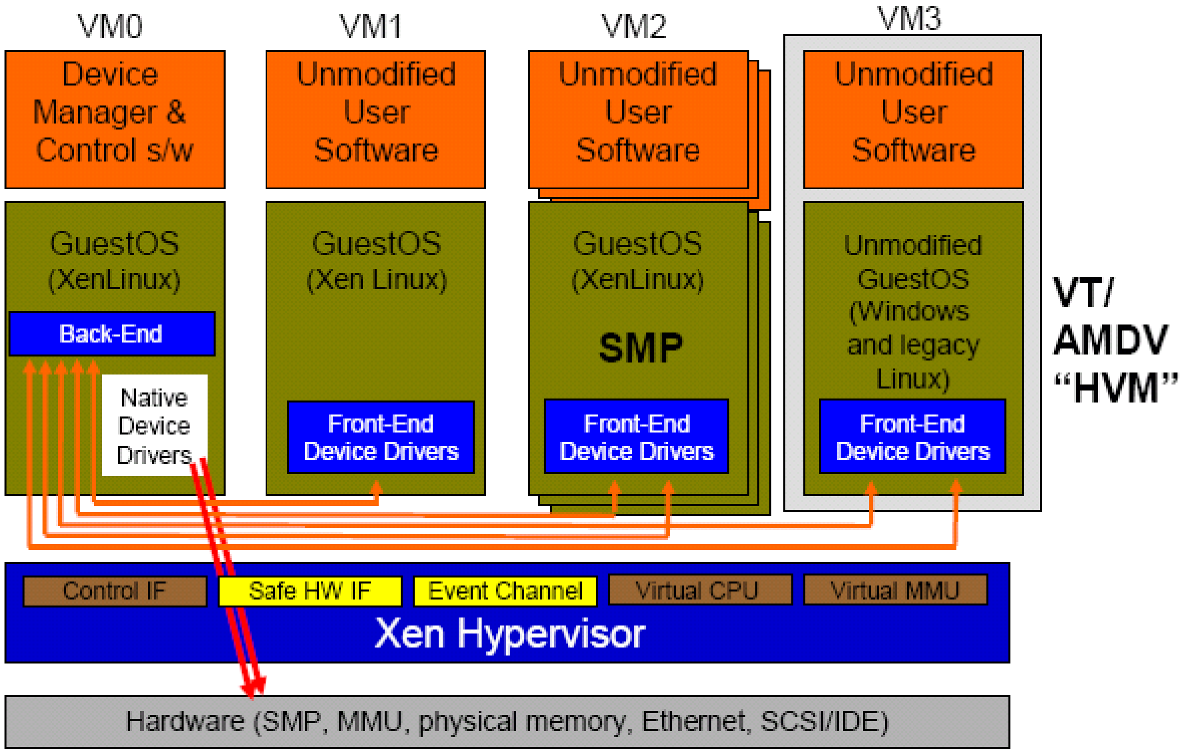
- Can be slow
- Drivers needed in monitor or host

Para-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 lguest

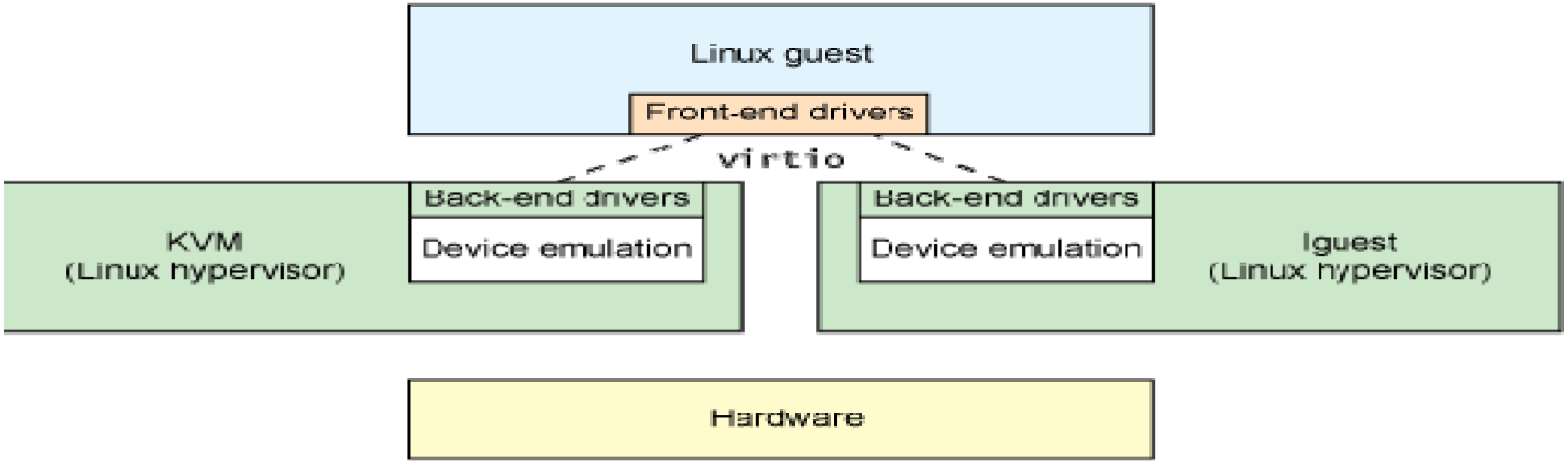
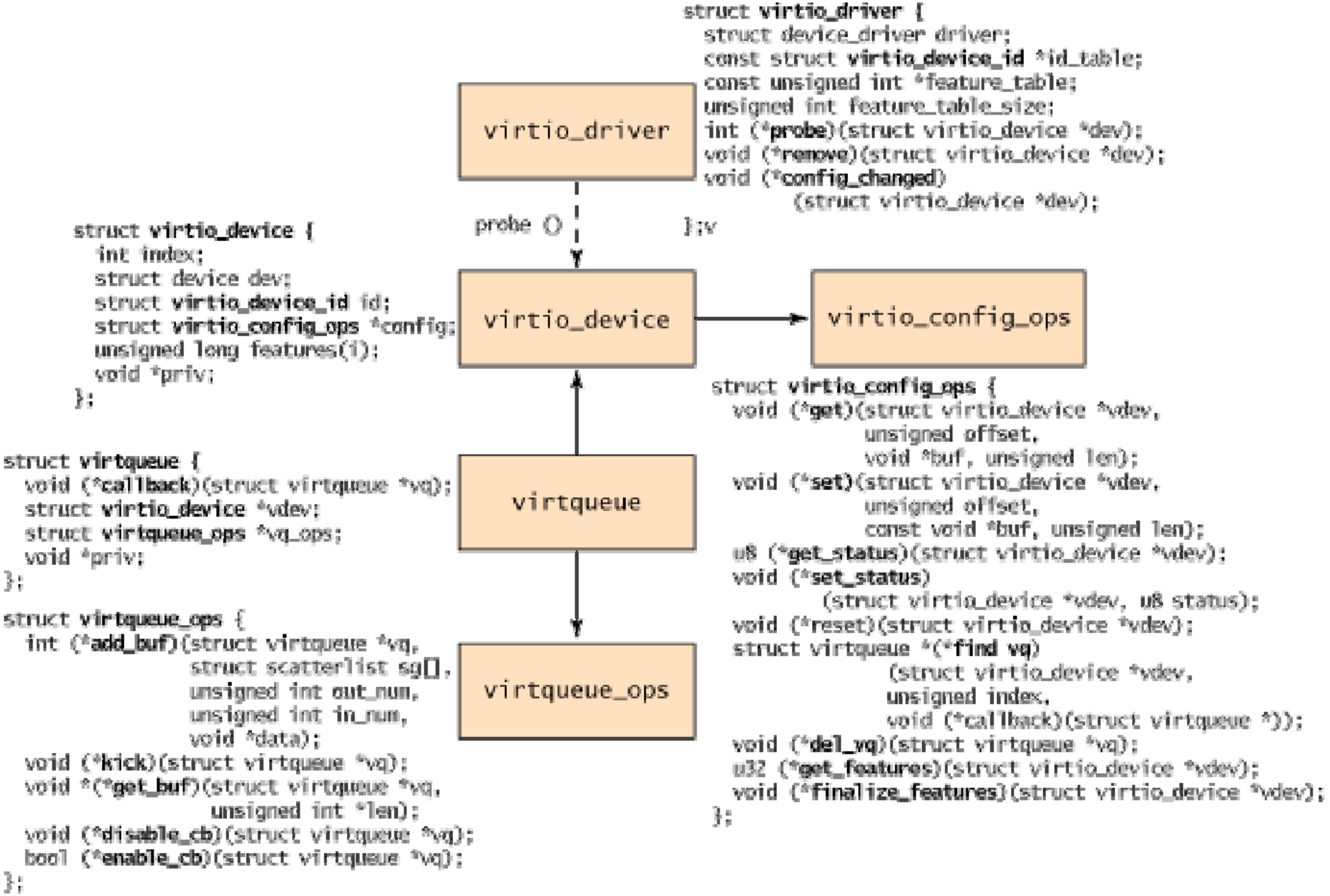


Figure from: http://www.ibm.com/developerworks/cn/linux/l-virtio/figure2.gif

VirtIO



```
struct virtio_driver {
  struct device_driver driver;
  const struct virtio_device_id *id_table;
  const unsigned int *feature_table;
  unsigned int feature_table_size;
  int (*probe)(struct virtio_device *dev);
  void (*remove)(struct virtio_device *dev);
  void (*config_changed)
          (struct virtio_device *dev);
};v
```

```
struct virtio_device {
  int index;
  struct device dev;
  struct virtio_device_id id;
  struct virtio_config_ops *config;
  unsigned long features(1);
  void *priv;
};
```

```
struct virtio_config_ops {
  void (*get)(struct virtio_device *vdev,
              unsigned offset,
              void *buf, unsigned len);
  void (*set)(struct virtio_device *vdev,
              unsigned offset,
              const void *buf, unsigned len);
  u8 (*get_status)(struct virtio_device *vdev);
  void (*set_status)
          (struct virtio_device *vdev, u8 status);
  void (*reset)(struct virtio_device *vdev);
  struct virtqueue *(*find vq)
                (struct virtio_device *vdev,
                unsigned index,
                void (*callback)(struct virtqueue *));
  void (*del_vq)(struct virtqueue *vq);
  u32 (*get_features)(struct virtio_device *vdev);
  void (*finalize_features)(struct virtio_device *vdev);
};
```

```
struct virtqueue {
  void (*callback)(struct virtqueue *vq);
  struct virtio_device *vdev;
  struct virtqueue_ops *vq_ops;
  void *priv;
};
```

```
struct virtqueue_ops {
  int (*add_buf)(struct virtqueue *vq,
                 struct scatterlist sg[],
                 unsigned int out_num,
                 unsigned int in_num,
                 void *data);
  void (*kick)(struct virtqueue *vq);
  void *(*get_buf)(struct virtqueue *vq,
                   unsigned int *len);
  void (*disable_cb)(struct virtqueue *vq);
  bool (*enable_cb)(struct virtqueue *vq);
};
```

http://www.ibm.com/developerworks/cn/linux/l-virtio/

Hardware-assisted I/O Virtualization

Motivation

- Emulated devices
  - Slow performance
  - High implementation complexity

- Para-virtualized I/O virtualization
  - Needs modification to guest OS, e.g., Linux, Windows
  - May still have suboptimal performance

- Hardware-assisted I/O virtualization
  - Provide efficient hardware support for I/O virtualization
  - E.g., Intel VT-d

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 invovlement

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.

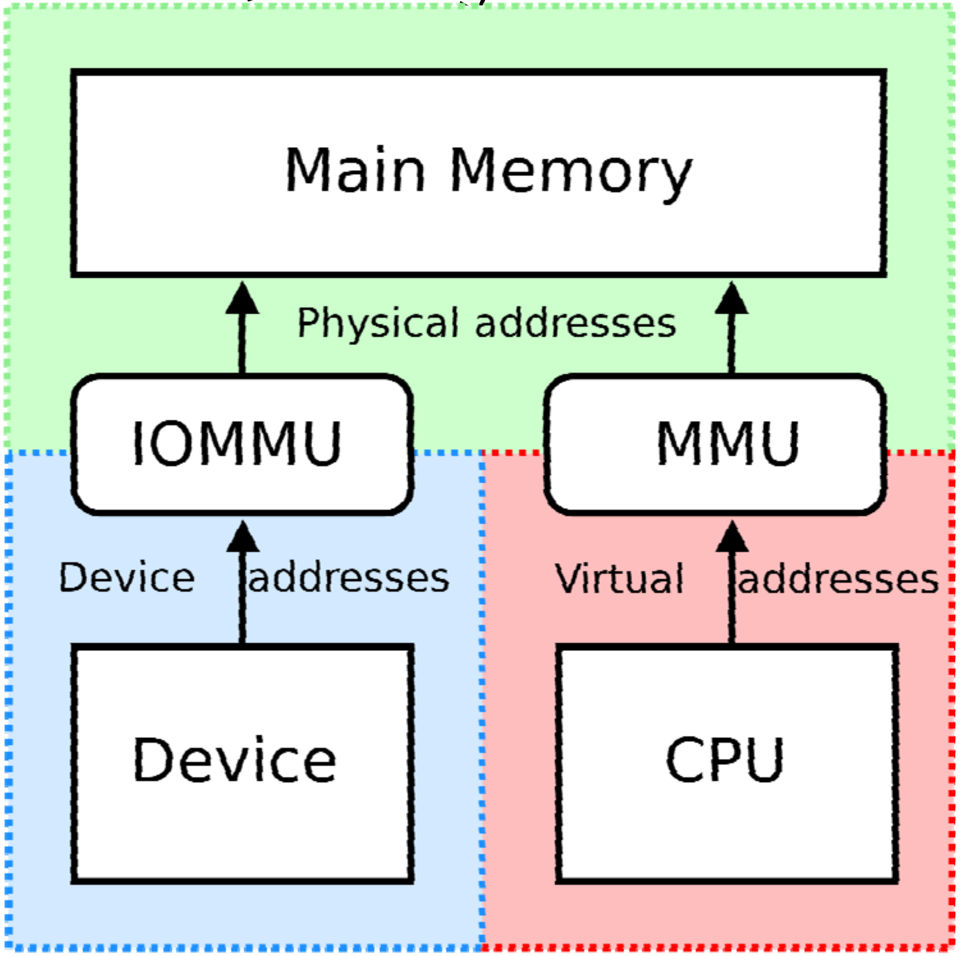
Supports the remapping of I/O DMA transfers and device-generated interrupts.

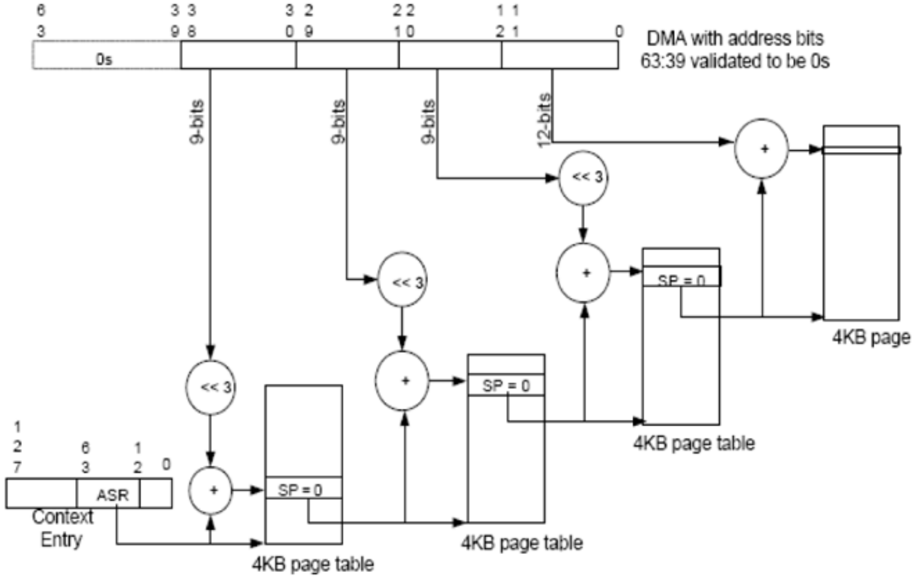
Provides flexibility to support multiple usage models that may run un-modified, special-purpose, or "virtualization aware" guest OSs.

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

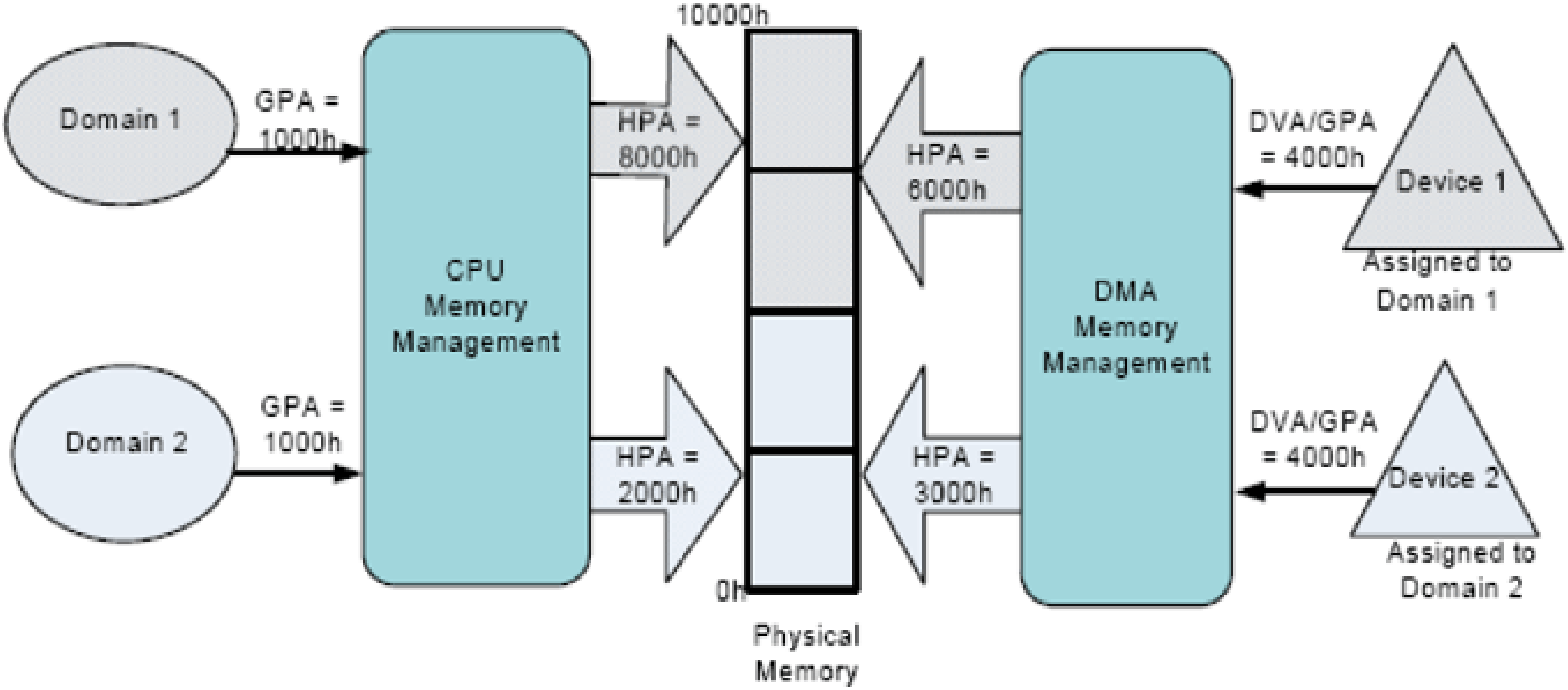




DMA Remapping

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

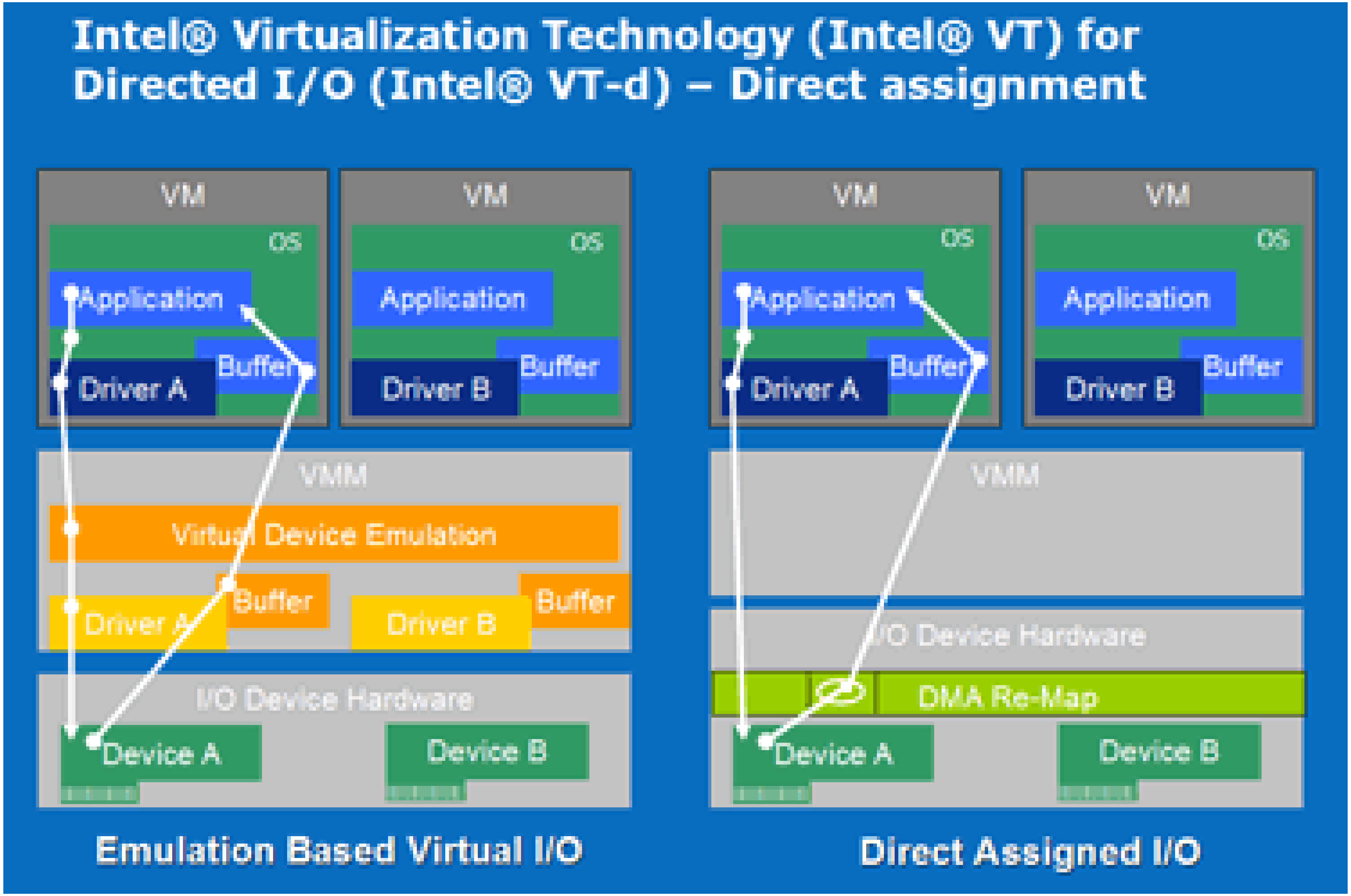
The interrupt requests generated by I/O devices must be controlled by the VMM

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



Device multiplexiing:
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.

- 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

Single Root I/O Virtualization

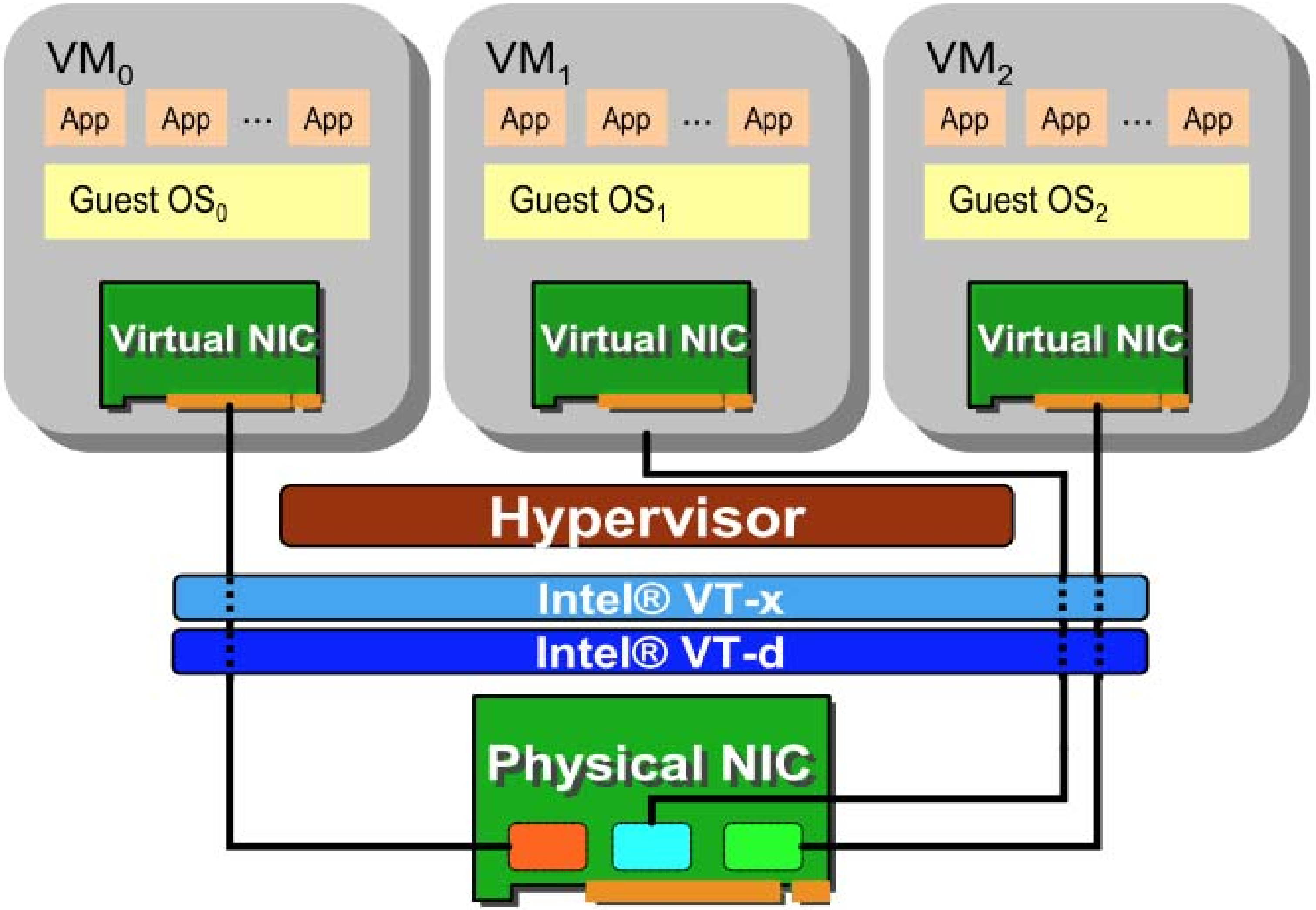
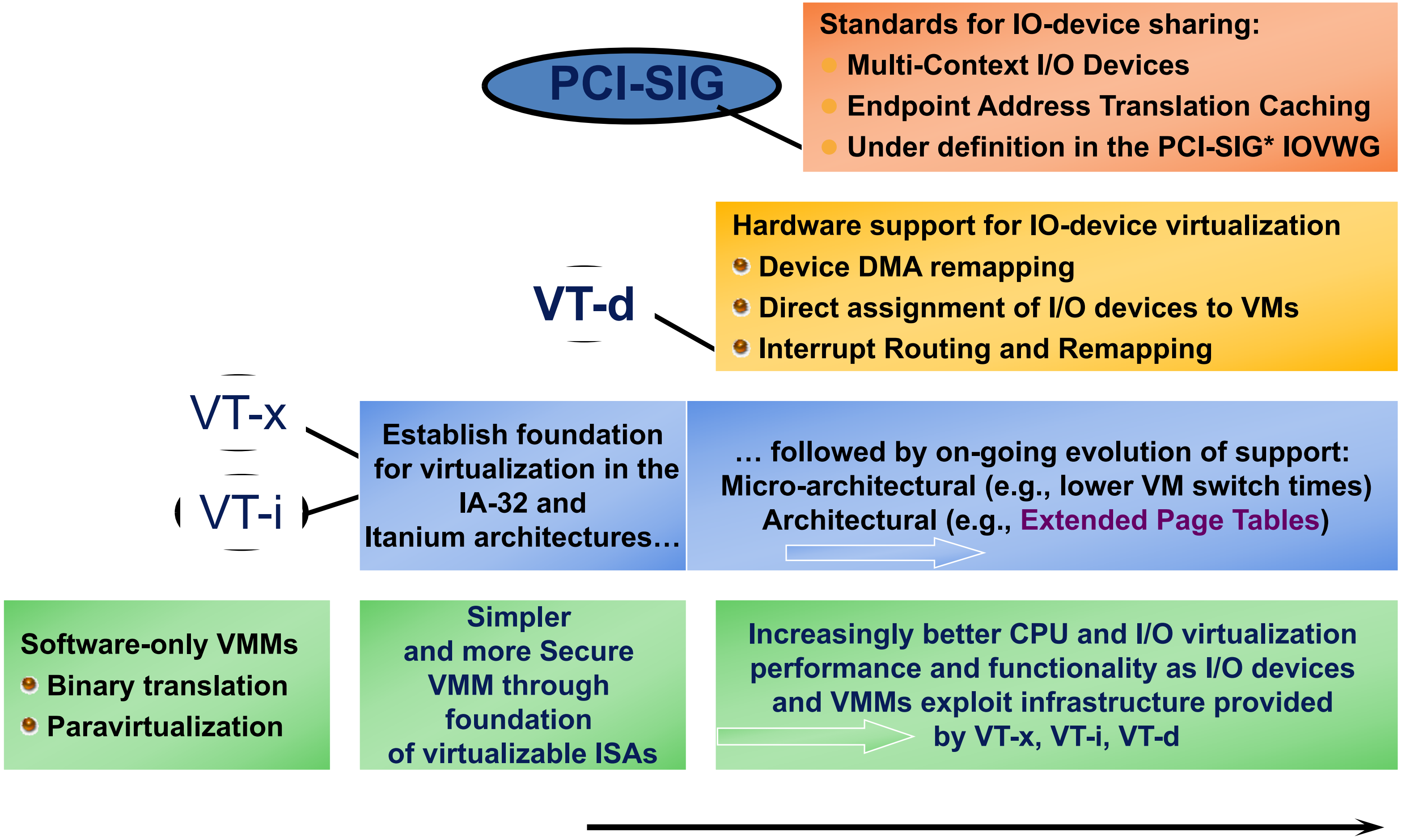


Figure 1. Natively Shared

Intel Virtualization Technology Evolution



PCI-SIG

Standards for IO-device sharing:

- Multi-Context I/O Devices
- Endpoint Address Translation Caching
- Under definition in the PCI-SIG* IOVWG

VT-d

Hardware support for IO-device virtualization

- Device DMA remapping
- Direct assignment of I/O devices to VMs
- Interrupt Routing and Remapping

VT-x

VT-i

Establish foundation for virtualization in the IA-32 and Itanium architectures…

… followed by on-going evolution of support:
Micro-architectural (e.g., lower VM switch times)
Architectural (e.g., Extended Page Tables)

Software-only VMMs

- Binary translation
- Paravirtualization

Simpler and more Secure VMM through foundation of virtualizable ISAs

Increasingly better CPU and I/O virtualization performance and functionality as I/O devices and VMMs exploit infrastructure provided by VT-x, VT-i, VT-d