Introduction

Coping with the complexity of computer system

Instructors

Binyu Zang 臧斌宇
binyu.zang@gmail.com

Yubin Xia 夏虞斌
xiayubin@gmail.com

Teaching Assistants

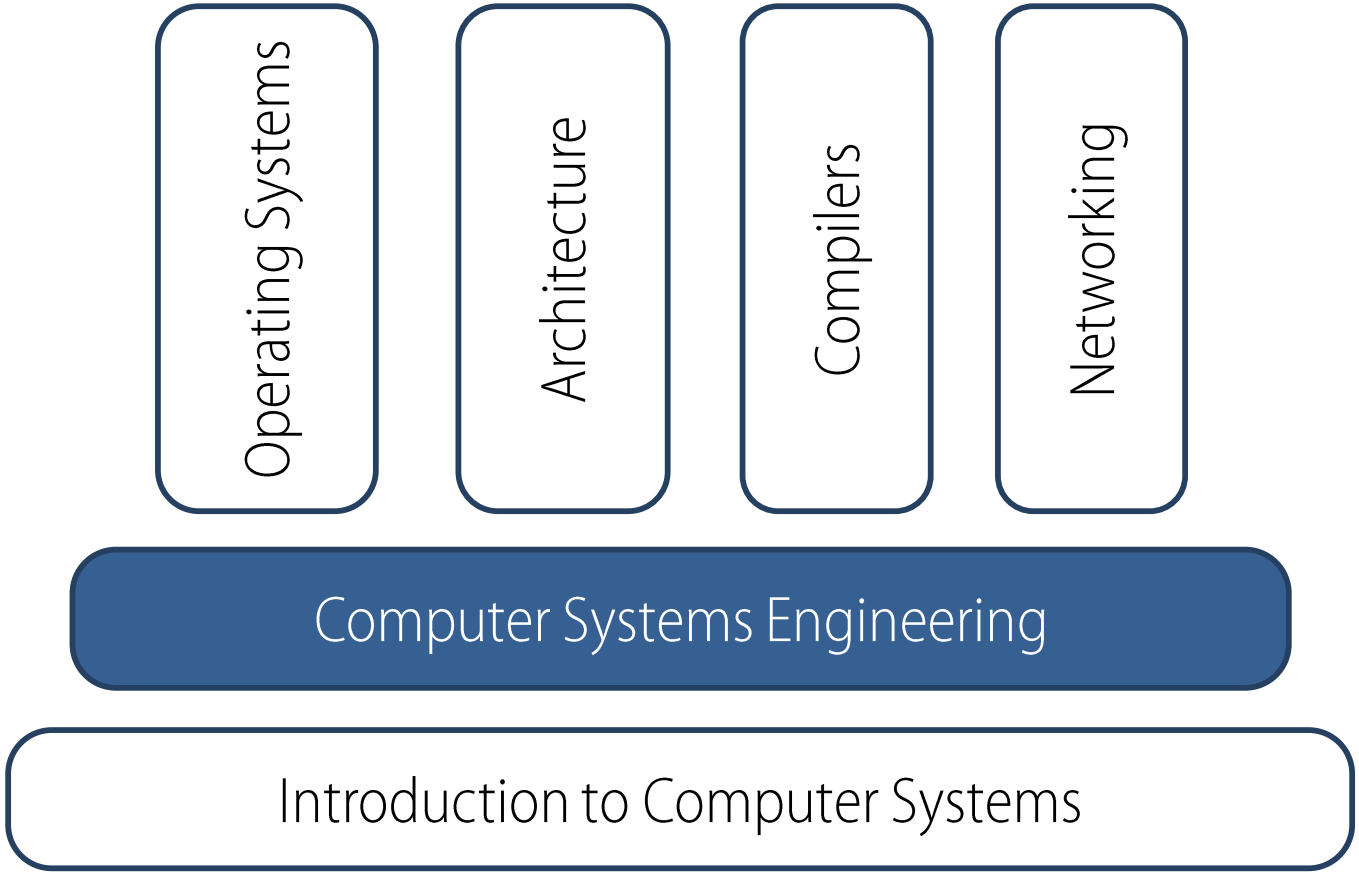
- Qianqian Yu
- Jinyu Gu
- Lei Shi
- Yang Zheng

Textbook

- Principles of Computer System Design: An Introduction
  - Jerome H. Saltzer & M. Frans Kaashoek, June 2009.

- Others
  - Computer Systems: An Integrated Approach to Architecture and Operating Systems
  - Distributed Systems: Principles and Paradigms, 2nd Edition
  - Papers and articles for recitation

Where is CSE in Courses

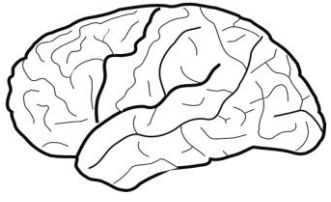


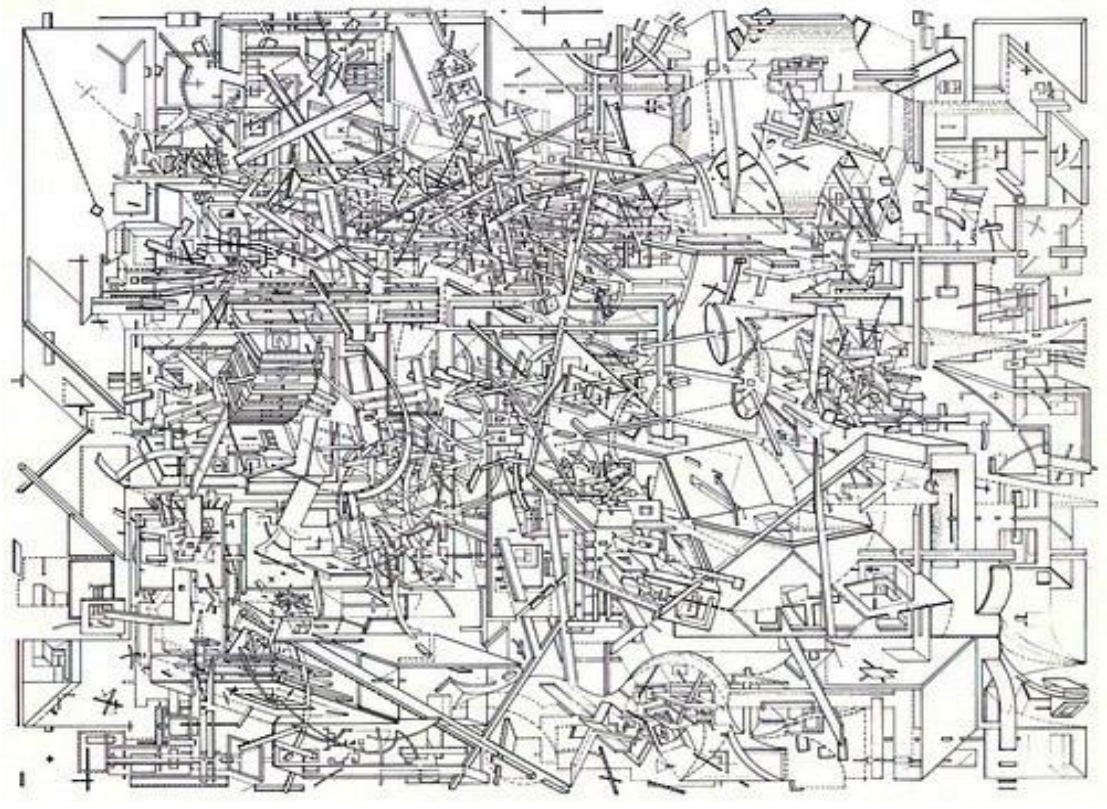
What is a System?

- System = *Interacting set of components with a specified behavior at the interface with its environment*
  - Examples: Web, Android, Linux, File system

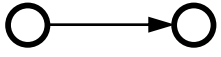
- This lecture: study and design of systems, their components, and internals

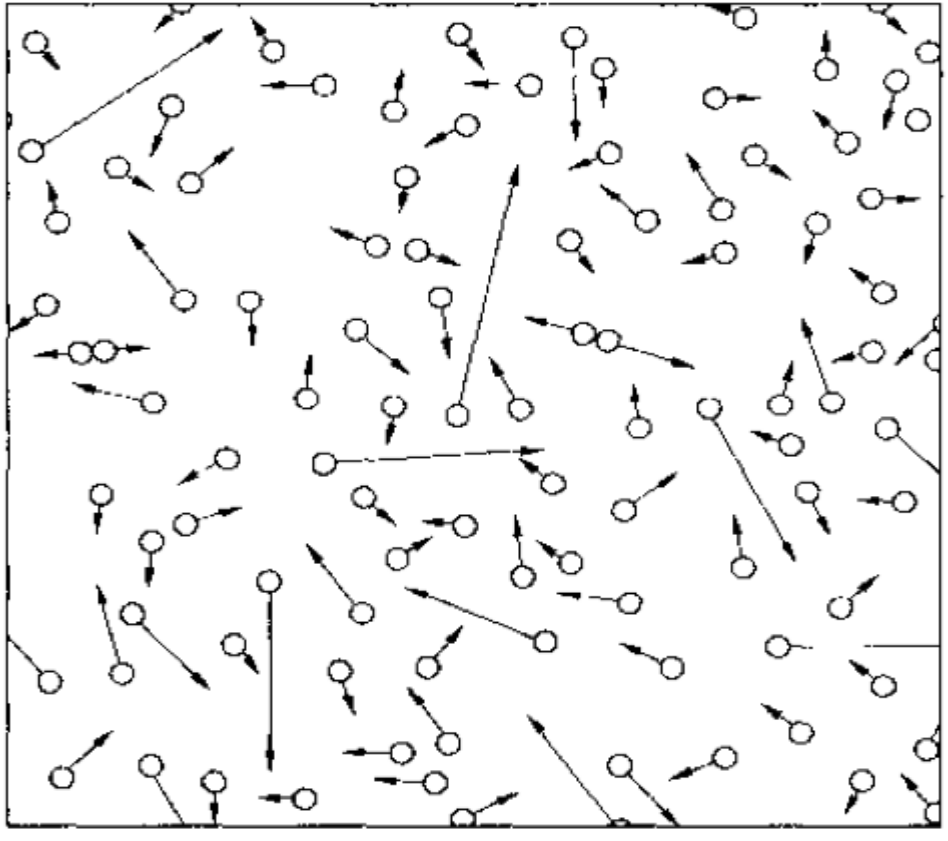
The Problem: Complexity of the Systems





An Example: The Gas System





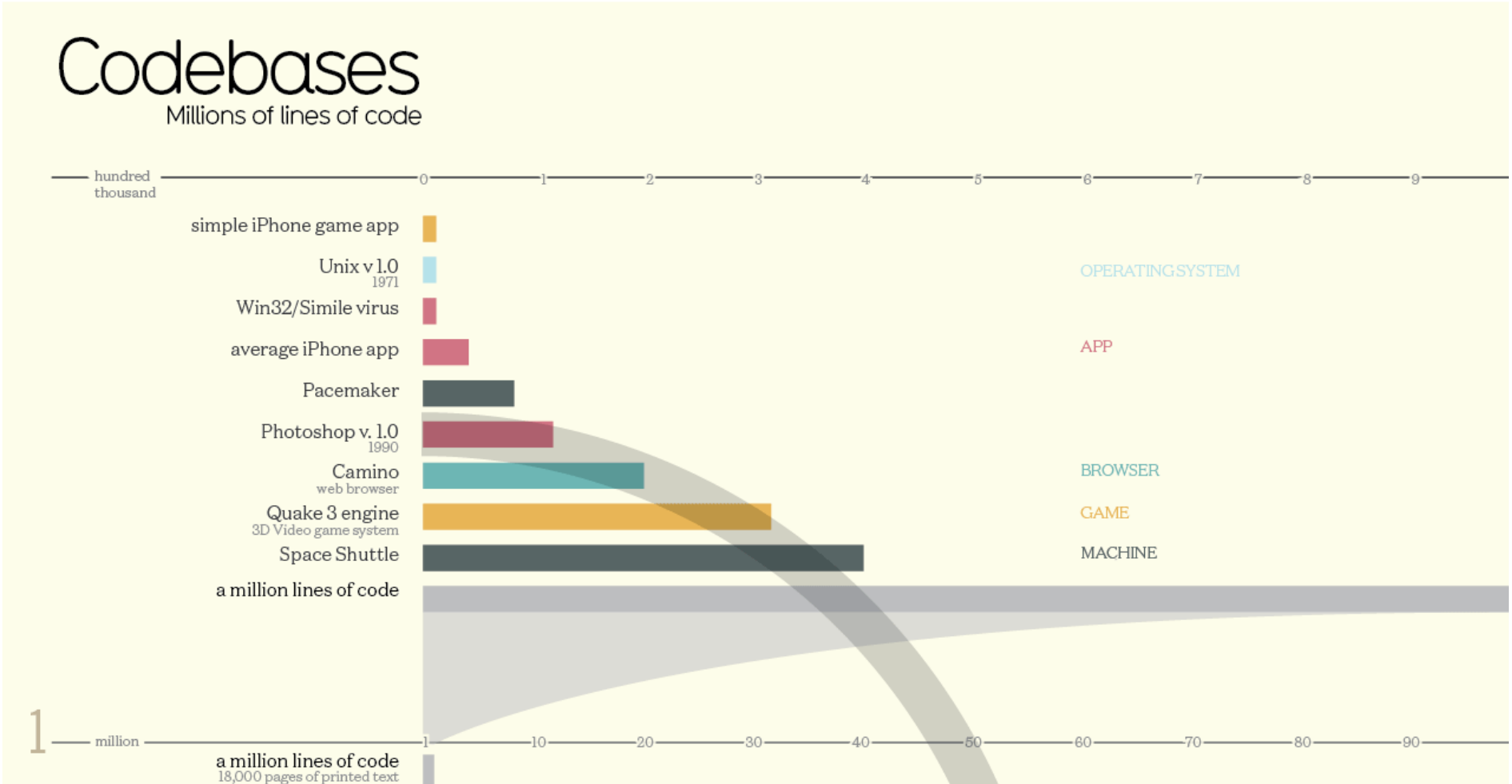
Compare with the Computer Systems

- Programming / Data Structure
  - LOC (Lines Of Code): From hundreds to millions
- Operating System / Architecture
  - Cores: from one to hundreds
- Network
  - Nodes: from two to millions
- Web Service
  - Clients: from tens to millions

Example Complex System: the Linux Kernel

- 1975 Unix kernel: 10,500 LOC (Lines of Code)
- 2008 Linux 2.6.24 line counts:
  - 85,000 processes
  - 430,000 sound drivers
  - 490,000 network protocols
  - 710,000 file systems
  - 1,000,000 different CPU architectures
  - 4,000,000 drivers
  - 7,800,000 Total

LOC (Lines Of Code)



Complexity of Computer Systems

- Hard to define; symptoms:
  - Large number of components
  - Large number of connections
  - Irregular
  - No short description
  - Many people required to design/maintain

- Technology rarely the limit
  - Indeed tech opportunity is the problem
  - Limit is usually designers' understanding

SYSTEM COMPLEXITY

Problem Types

- Emergent properties (surprise!)
  - The properties that are not considered at design time
- Propagation of effects
  - Small change -> big effect
- Incommensurate scaling
  - Design for small model may not scale
- Trade-offs
  - Waterbed effect

1. Emergent Properties

- Features
  - No evident in the individual components of a system
  - But show up when combining those components
  - Might also be called surprises
  - An unalterable fact of life: some things turn up only when a system is built

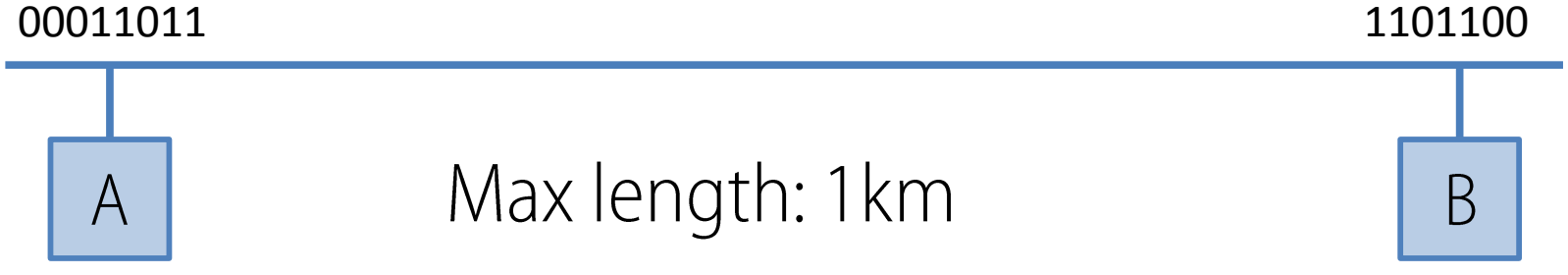
1. Emergent Properties (Cont.)

- The Millennium Bridge
  - For pedestrians over the River Thames in London
  - Pedestrians synchronize their footsteps when the bridge sways, causing it to sway even more

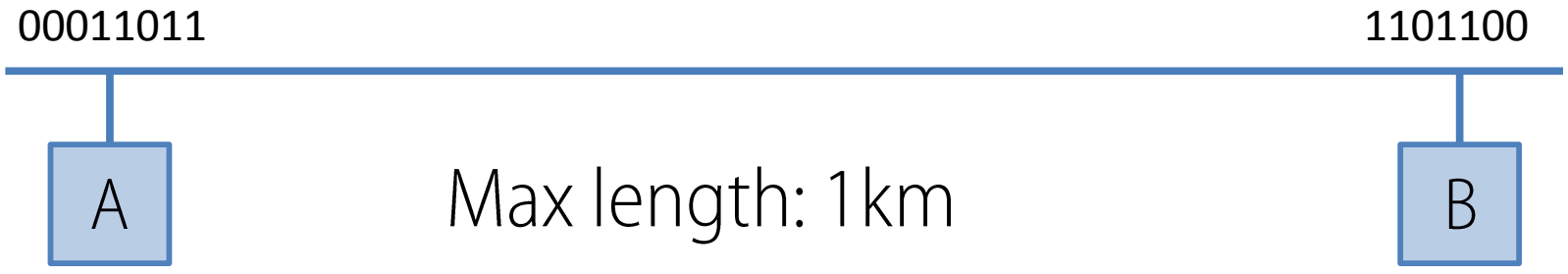
  - It had to be closed after only a few days

Emergent Property Example: Ethernet

- All computers share single cable
- Goal is reliable delivery
- Listen while sending to detect collisions
  - If two nodes sends data at the same time, then both cancel and wait for a random time



Does Collision Detection Work?



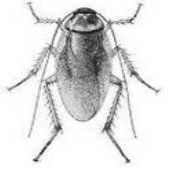
- What if A finishes sending before data from B arrives?
  - 1km at 60% speed of light = 5 ms (microseconds)
  - Original Ethernet Spec: 3 Mbit/sec
    - A can send 15 bits before bit 1 arrives at B
    - A must keep sending for 2*5 ms (to detect collision when first bit from B arrives)
  - Minimum packet size is 5*2*3 = 30 bits
  - The default header is 5 bytes (40 bits), so no problem!

3 Mbit/s to 10 Mbit/s

- First Ethernet standard: 10 Mbit/s, 2.5 km wire
  - Must send for 2*12.5 μseconds = 250 bits @ 10 Mb/s
  - Header was 14 bytes
  - Needed to pad packets to at least 250 bits (32 bytes)
- Emergent property: Minimum packet size!
  - The 250-bit minimum packet size is a surprise

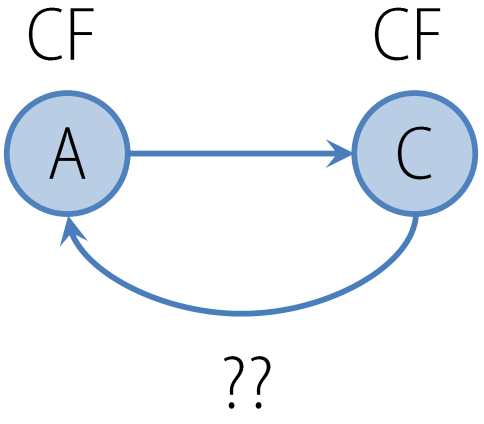
2. Propagation of Effects [Cole'69]

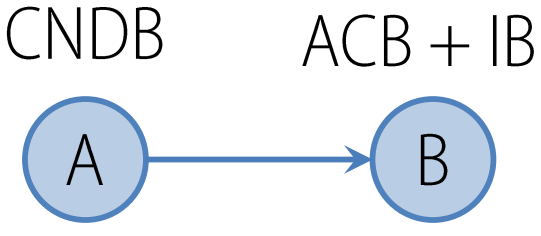
- WHO: tried control malaria in North Borneo
  - Sprayed villages with DDT
  - Wiped out mosquitoes, but ….
  - Roaches collected DDT in tissue
  - Lizards ate roaches and became slower
  - Easy target for cats
  - Cats didn't deal with DDT well and died
  - Forest rats moved into villages
  - Rats carried the bacillus for the plague
- WHO replaced malaria with the plague



Example: No Small Changes

- Phone network features
  - CF: Call Forwarding
  - CNDB: Call Number Delivery Blocking
    - The caller's number should be hidden
  - ACB: Automatic Call Back
  - IB: Itemized Billing





- A calls B, B is busy
- Once B is done, B automatically calls A
- A's (caller) number appears on B's bill

3. Incommensurate Scaling

- As a system increases in size or speed, not all parts of it follow the same scaling rules
  - So things stop working
- The mathematical description
  - Different parts of the system exhibit different orders of growth

3. Incommensurate Scaling (Cont.)

- Galileo in 1638
  - To illustrate briefly, I have sketched a bone whose natural length has been increased three times and whose thickness has been multiplied until, for a correspondingly large animal, it would perform the same function which the small bone performs for its small animal. From the figures here shown you can see how out of proportion the enlarged bone appears.

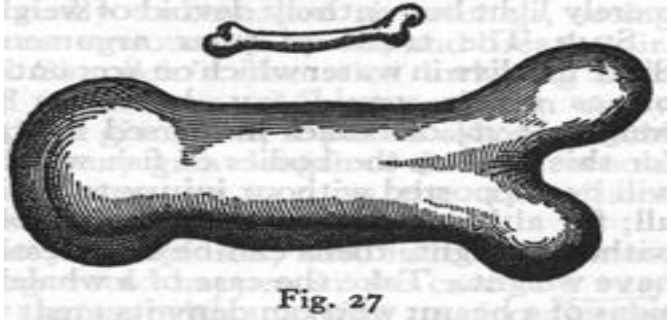


Fig. 27

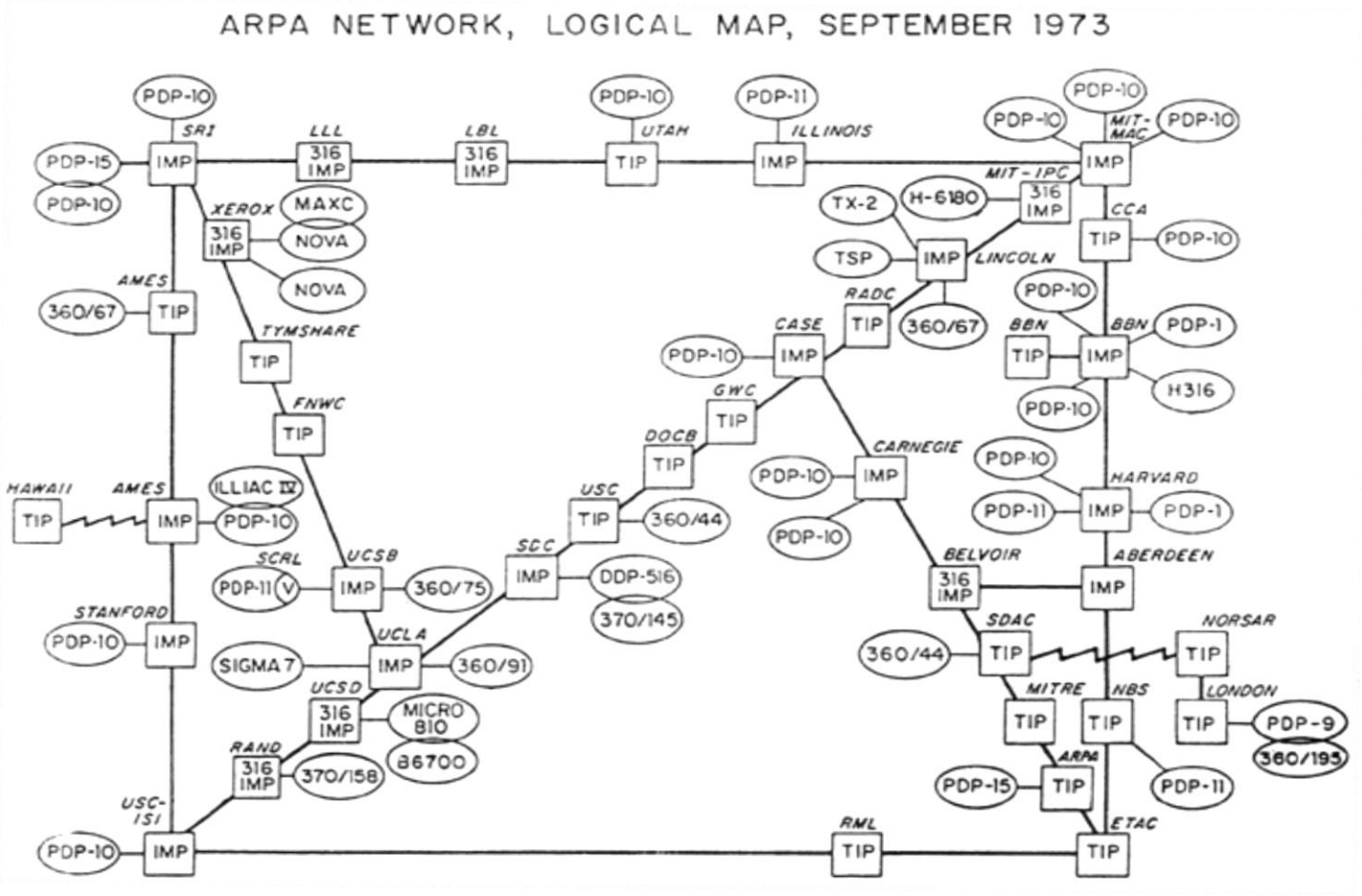
3. Incommensurate Scaling (Cont.)

- Galileo in 1638
  - Clearly then if one wishes to maintain in a great giant the same proportion of limb as that found in an ordinary man he must either find a harder and stronger material for making the bones, or he must admit a diminution of strength in comparison with men of medium stature; for if his height be increased inordinately he will fall and be crushed under his own weight.

3. Incommensurate Scaling (Cont.)

- Galileo in 1638
  - Whereas, if the size of a body be diminished, the strength of that body is not diminished in the same proportion; indeed the smaller the body the greater its relative strength. Thus a small dog could probably carry on his back two or three dogs of his own size; but I believe that a horse could not carry even one of his own size.

3. Incommensurate Scaling (Cont.)



Example: Scaling the Internet

- Size routing tables (for shortest paths): $O(n^2)$
  - Hierarchical routing on network numbers
  - Address: 16 bit network number and 16 bit host number
- Limited networks (2^{16})

- Solutions:
  - NAT (Network Address Translators) and IPv6

4. Trade-offs

- General Models
  - Limited amount of goodness
  - Maximize the goodness
  - Avoid wasting
  - Allocate where helps most
- Waterbed Effect
  - Pushing down on a problem at one point
  - Causes another problem to pop up somewhere else

4. Trade-offs (Cont.)

- Binary Classification
  - We wish to classify a set of things into two categories
    - Based on presence or absence of some property
  - But we lack a direct measure of that property
  - So we identify instead some indirect measure
    - Known as a proxy

4. Trade-offs (Cont.)

- Binary Classification (Cont.)
  - Occasionally this scheme misclassifies something
  - By adjusting parameters of the proxy
  - The designer may be able to
    - reduce one class of mistakes
    - but only at the cost of increasing some other class of mistakes

How to Handle?

- Ideally, the Constructive Theory
  - Allows the designer systematically to
    - Synthesize a system from its specifications
    - Make necessary trade-offs with precision
  - In some fields
    - Communication systems
    - Linear control systems
    - Design of bridge and skyscrapers (to a certain extent)

How to Handle? (Cont.)

- In Computer Systems
  - "We find that we were born too soon" -- Our textbook
  - The problems
    - We work almost entirely by analyzing ad hoc examples rather than by synthesizing
    - So, in place of a well-organized theory, we use case studies

Signs of Complexity

- Webster's Definition
  - "Difficult to understand"
- Signs of complexity (like diagnosis in medicine)
  - Large number of components
  - Large number of interconnections
  - Many irregularities
  - A long description
  - A team of designers, implementers, or maintainers

Limit the Levels of Complexity

- All systems are *indefinitely*
  - The deeper one digs, the more signs of complexity turn up
  - A computer -> gates -> electrons -> quarks -> …

- Abstraction: limits the depth of digging

M.A.L.H

COPING WITH COMPLEXITY

M.A.L.H

- **Modularity**
  - Split up system
  - Consider separately
- **Abstraction**
  - Interface/Hiding
  - Avoid propagation of effects
- **Layering**
  - Gradually build up capabilities
- **Hierarchy**
  - Reduce connections
  - Divide-and-conquer

Modularity

- Analyze or design the system as a collection of interacting subsystems
  - Subsystems called modules
  - Divide-and-conquer technique

- The simplest, most important tool for reducing complexity
  - Be able to consider interactions among the components within a module
  - Without simultaneously thinking about the components that are inside other modules

Modularity (Cont.)

- Example
  - Debugging a program with N statements
  - Number of bugs is proportional to its size
  - Bugs are randomly distributed

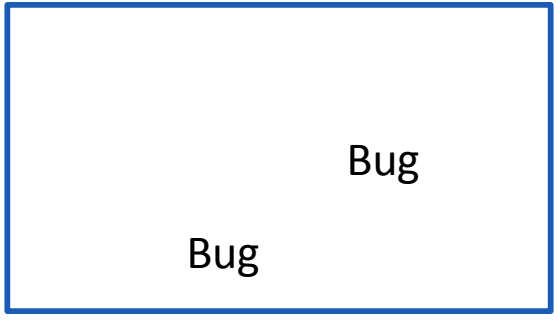
$$BugCount \sim N$$
$$DebugTime \sim N \times BugCount \sim N^2$$

Original

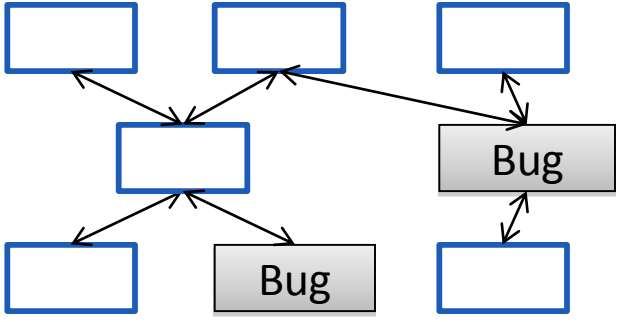
$$DebugTime \sim \left(\frac{N}{K}\right)^2 \times K \sim \frac{N^2}{K}$$

With Modularity

Modularity (Cont.)



Original System



System with Modularity

Abstraction

- Abstraction
  - Treat a module based on external specifications, no need for details inside
- Principles to divide a module
  - Follow natural or effective boundaries
  - Fewer interactions among modules (Chap.4 & 5)
  - Less propagation of effects

Abstraction (Cont.)

- Examples
  - DVD players
  - Registers (circuits to remember states)
  - Procedure call
  - Applications with window interfaces
  - Games, spreadsheet, web browsers

Abstraction (Cont.)

- Minimizing interconnections among modules may be defeated
  - Unintentional or accidental interconnections, arising from implementation errors
  - Well-meaning design attempts to sneak past modular boundaries
    - Improve performance
    - Meet some other requirement

Abstraction (Cont.)

- Software is particularly subject to this problem
  - The modular boundaries provided by the separately compiled subprograms are actually somewhat soft
  - Is easily penetrated by errors in
    - using pointers
    - filling buffers
    - calculating array indices

Abstraction (Cont.)

- System designers prefer techniques
  - Enforce modularity by
    - Interposing impenetrable walls between modules
  - Assure that there can be no
    - Unintentional or hidden interconnections

Abstraction (Cont.)

- Failure Containment
  - A module does not meet its abstract interface specifications
  - Limiting the impact of faults
    - Well-designed and properly enforced modular abstractions
    - Control propagation of effects
  - Modules are the units of fault containment

Abstraction (Cont.)

- The Robustness Principle
  - Be tolerant of inputs and strict on outputs
    - Suppress noise or errors
    - Not propagate or amplify them
  - A module should accept its input values
    - If it is still apparent how to sensibly interpret them
    - Even if they are not within specified ranges
  - A module should construct its outputs
    - Conservatively in accordance with its specification
    - Making them even more accurate or more constrained than the specification requires if possible

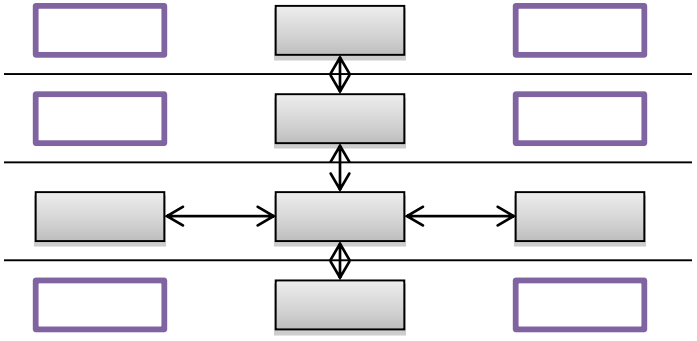
Abstraction (Cont.)

- The Safety Margin Principle
  - Keep track of the distance to the cliff, or you may fall over the edge
  - Abnormal inputs indicate something is going wrong
  - Track and report out-of-tolerance inputs
  - Shake-out mode & production mode

Layering

- Goal
  - Reduce module interconnections even more
- How to do it
  - Build a set of mechanisms first(a lower layer)
  - Use them to create a different complete set of mechanisms (an upper layer)
- General rule: A module in one layer only interacts with:
  - Its peers in the same layer, and
  - Modules in the next lower layer / next higher layer

Layering (Cont.)



Application layer

OS layer

Processor & memory layer

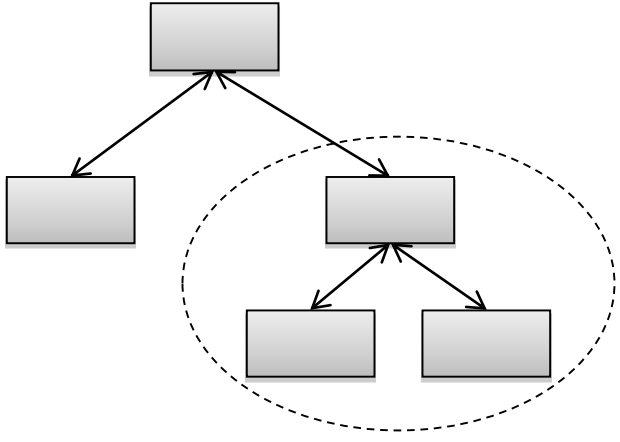
Memory cells & gates layer

- House:
  - Inner layer of studs, joist, rafter (shape & strength)
  - Layer of sheathing and drywall (wind out)
  - Layer of siding, flooring and roof tiles (watertight)
  - Cosmetic layer of paint (looks good)
- Algebra:
  - integer, complex number, polynomials, polynomials with polynomial coefficients

Hierarchy

- Hierarchy: another module organization
  - Start with a small group of modules
  - Assemble them into a stable, self-contained subsystem with well defined interface
  - Assemble a group of subsystems to a larger subsystem

- Example
  - 1 manager leads N employees
  - 1 higher manager leads N lower managers



Hierarchy (Cont.)

- There are many striking examples of hierarchy
  - from microscopic biological systems
  - to the assembly of Alexander's empire
  - Offers compelling arguments
    - under evolution, hierarchical designs have a better chance of survival
- Constrains interactions
  - Permitting them only among the components of a subsystem
- Reduces the number of potential interactions among modules from square-law to linear

COMPUTER SYSTEMS ARE DIFFERENT

Computer Systems are Different

- Computer systems are the same as all other systems (plausible)
  - Certain common problems show up in all complex systems
  - The techniques that have been devised for coping with complexity are universal
- Computer systems are different
  - The complexity is not limited by physical laws
  - The rate of change of technology is unprecedented

Unbounded Composition

- Two properties of computer systems
  - 1. Mostly digital
  - 2. Controlled by software
  - Both relax the limits on complexity arising from physical laws in other systems

Computer System: Coping with Complexity

- M.A.L.H are NOT enough
  - Hard to choose the right modularity
  - Hard to choose the right abstraction
  - Hard to choose the right layer
  - Hard to choose the right hierarchy

Class Plan

- **Naming**: Glue between modules
- **Operating systems**: Enforced modularity within a machine
- **Network**: Enforced modularity between machines
- **Reliability and transactions**: Handing hardware failures
- **Security**: Handling malicious failures

Scores

- Exam: 45%
- Lab: 30%
- Recitation: 15%
- Homework: 10%

Naming Scheme

to achieve modularity

Our Web-site

- http://ipads.se.sjtu.edu.cn/courses/cse

Review

- Challenge in computer system: complexity
  - Emergent properties
  - Propagation of efforts
  - Incommensurate scaling
  - Trade-offs
- Solutions
  - M.A.L.H.

Naming: the glue of modues

NAMING SCHEME

Naming in General

- ipads.se.sjtu.edu.cn – hostname
- steven@apple.com - email
- steven – username
- EAX - x86 processor register name
- main() - function name
- WebBrowser - class name
- /courses/cse/index.html - path name (fully-qualified)
- index.html - path name (relative)
- http://ipads.se.sjtu.edu.cn/courses/cse/index.html - URL
- 13918275839- Phone number
- 202.120.40.188 - IP Address

Use Name to Achieve Modularity

- Retrieval: e.g., using URL to get a web page
- Sharing: e.g., passing an object reference to a function
  - Save space as well: only sending the name, not the object
- Hiding: e.g., using a file name without knowing file system
  - Can support access control: use an object only if knowing its name
  - E.g., Windows has many undocumented API
- User-friendly identifiers: e.g., "homework.txt" instead of 0x051DE540
- Indirection: e.g., OS can move the location of the file data without notifying the user
  - Have you ever defragmented your hard driver?

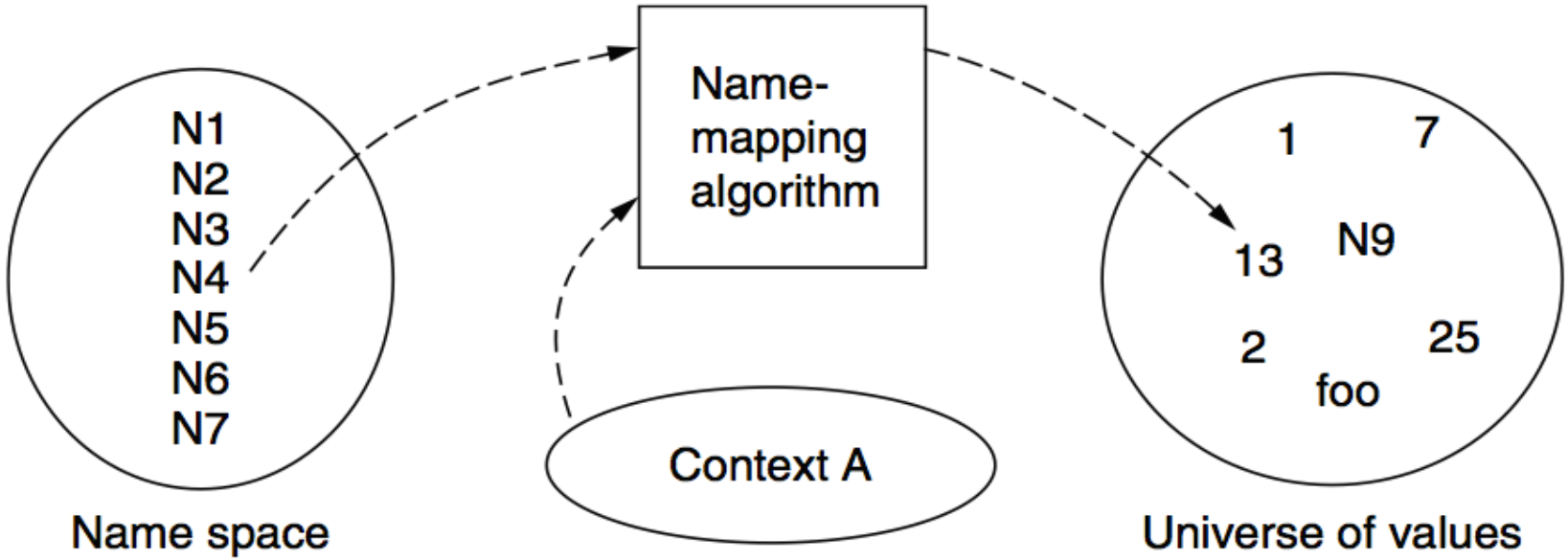
Address in Naming

- Software uses names in an obvious way
  - E.g., memory addresses
- Hardware modules connected to a bus
  - Use bus addresses (a kind of name) for interconnection

Naming Schemes

- 1. Set of all possible names
  - You cannot use 'for' as a variable in C
- 2. Set of all possible values
- 3. Look-up algorithm to translate a name into a value (or set of values, or "none")

Naming Model



Naming Terminology

- Binding – A mapping from a name to value
  - Unbind is to delete the mapping
  - A name that has a mapping is bound
- A name mapping algorithm resolves a name

Naming Context

- Type-1: context and name are separated
- Type-2: context is part of the name

- Name spaces with only one possible context are called *universal name spaces*
  - Example: credit card number

Determining Context - 1

- Hard code it in the resolver
  - Examples: Many universal name spaces work this way
- Embedded in name itself
  - cse@sjtu.edu.cn:
    - Name = "cse"
    - Context = "sjtu.edu.cn"
  - /ipads.se.sjtu.edu.cn/courses/cse/README :
    - Name = "README"
    - Context = "/ipads.se.sjtu.edu.cn/courses/cse"

Determining Context - 2

- Taken from environment (Dynamic)
  - Unix cmd: "rm foo":
    - Name = "foo", context is current dir
    - *Question: how to find the binary of "rm" command?*
  - Read memory 0x7c911109:
    - Name = "0x7c911109",
    - Context is thread's address space
- Many errors in systems due to using wrong context

Name Mapping Algorithms - 1

- Table lookup
  - Find name in a table
    - E.g., Phone book
  - Context: which table?
    - Implicit VS. explicit
    - Default context

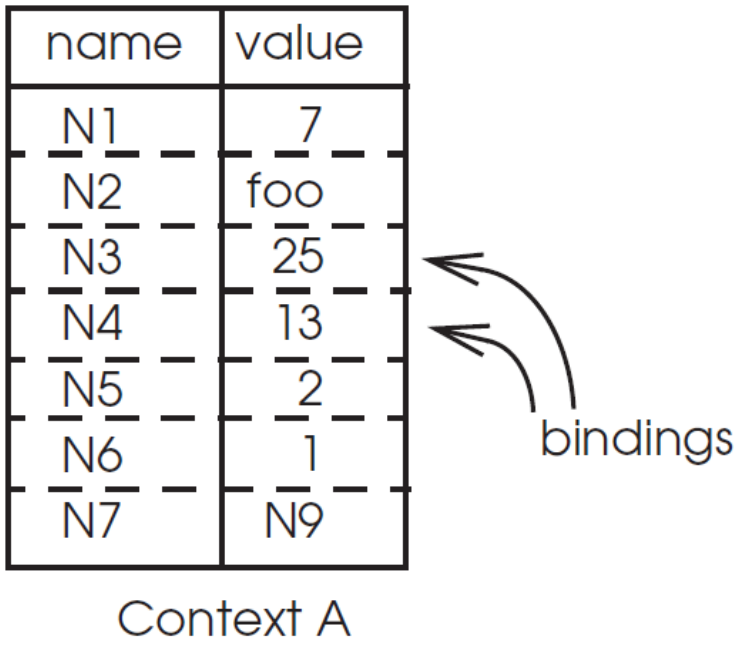


Table Lookup

Name Mapping Algorithms - 2

- Recursive lookup
  - E.g., "/usr/bin/rm"
  - First find "usr" in "/", then find "bin" in "/usr", then "rm"
  - Each look-up process is the same
- Multiple lookup
  - *Recall: how to find "rm" without absolute name?*
  - $PATH
    - E.g., "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
  - Look-up in a predefined list of context

Interpreter Naming API

- *value* ← RESOLVE(*name*, *context*)
  - Return the mapping of *name* in the *context*
- *status* ← BIND(*name*, *value*, *context*)
  - Establish a *name* to *value mapping* in the *context*
- *status* ← UNBIND(*name*, *context*)
  - Delete name from *context*
- *list* ← ENUMERATE(*context*)
  - Return a list of all bindings
- *result* ← COMPARE(*name1*, *name2*)
  - Check if *name1* and *name2* are equal

FAQ of Naming Scheme - 1

- What is the syntax of names?
- What are the possible value?
- What context is used to resolve names?
- Who specifies the context?
- Is a particular name global (context-free) or local?

FAQ of Naming Scheme - 2

- Does every name have a value?
  - Or, can you have "dangling" names?
- Can a single name have multiple values?
- Does every value have a name?
  - Or, can you name everything?
- Can a single value have multiple names?
  - Or, are there synonyms?
- Can the value corresponding to a name change over time?

Domain Name Service

CASE: DNS

DNS: Binding IP and Domain Name

- Names: hostname strings
  - E.g., www.sjtu.edu.cn
- Values: IP addresses
  - E.g., 202.120.2.119
- Look up algorithm
  - Resolves a hostname to an IP address so that your machine knows where to send packets

Address can be seen as a Type of Name

- An address itself is a type of name
  - A structured name that is used to locate an object
  - Recall your labs in ICS on socket
    - The program uses IP address to identify the server
  - On Internet
    - The router will know where to send a packet with source IP
- Hostname has no such semantic
  - A router does not know how to send a packet to "baidu.com"

Why Not Just Using IP Address?

- IPs are structured in a particular way for routing
  - You cannot pick your favorite four numbers as your IP
    - *Note: usually an address cannot be picked, e.g., your house addr.*
  - While host names are not using such structured
- One benefit
  - User-friendliness

Retrieval
Sharing
Hiding
User-friendly identifiers *
Indirection

IP Address ABC

- IP addresses are categorized to 5 types, A to E

Type	Net #	IP Range	Host #	Private IP
A	126 (27-2)	0.0.0.0-127.255.255.255	16777214	10.0.0.0-10.255.255.255
B	16384 (214)	128.0.0.0–191.255.255.255	65534	172.16.0.0-172.31.255.255
C	2097152(221)	192.0.0.0-223.255.255.255	254	192.168.0.0-192.168.255.255

- Type D and E are special ones
  - Type D: multicast address, started with 1110
  - Other special ones: e.g., 127.0.0.1-127.255.255.255

Questions on DNS

- Can a single name have multiple values?
  - Yes
  - This allows a web server to balance its load over multiple machines
  - Also allows a client to choice a nearest IP to access
- Can a single value have multiple names?
  - Yes
  - This allows server consolidation

Questions on DNS

- Can the value corresponding to a name change?
  - Yes
  - This allows to change the physical machine (with different IP) that stores the data without changing the hostname
  - So the changing is hidden to clients

The 3rd Part: the Look-up Algorithm

- History
  - Each machine kept a "hosts.txt" for address binding
    - E.g., "r900 202.120.224.83"
  - Using table look-up to resolve the binding
  - This method cannot scale in Internet
    - Using a different way of storing data
    - Thus requires a different look-up algorithm
  - 1984, four Berkeley students wrote BIND
    - Still the dominant DNS software in use

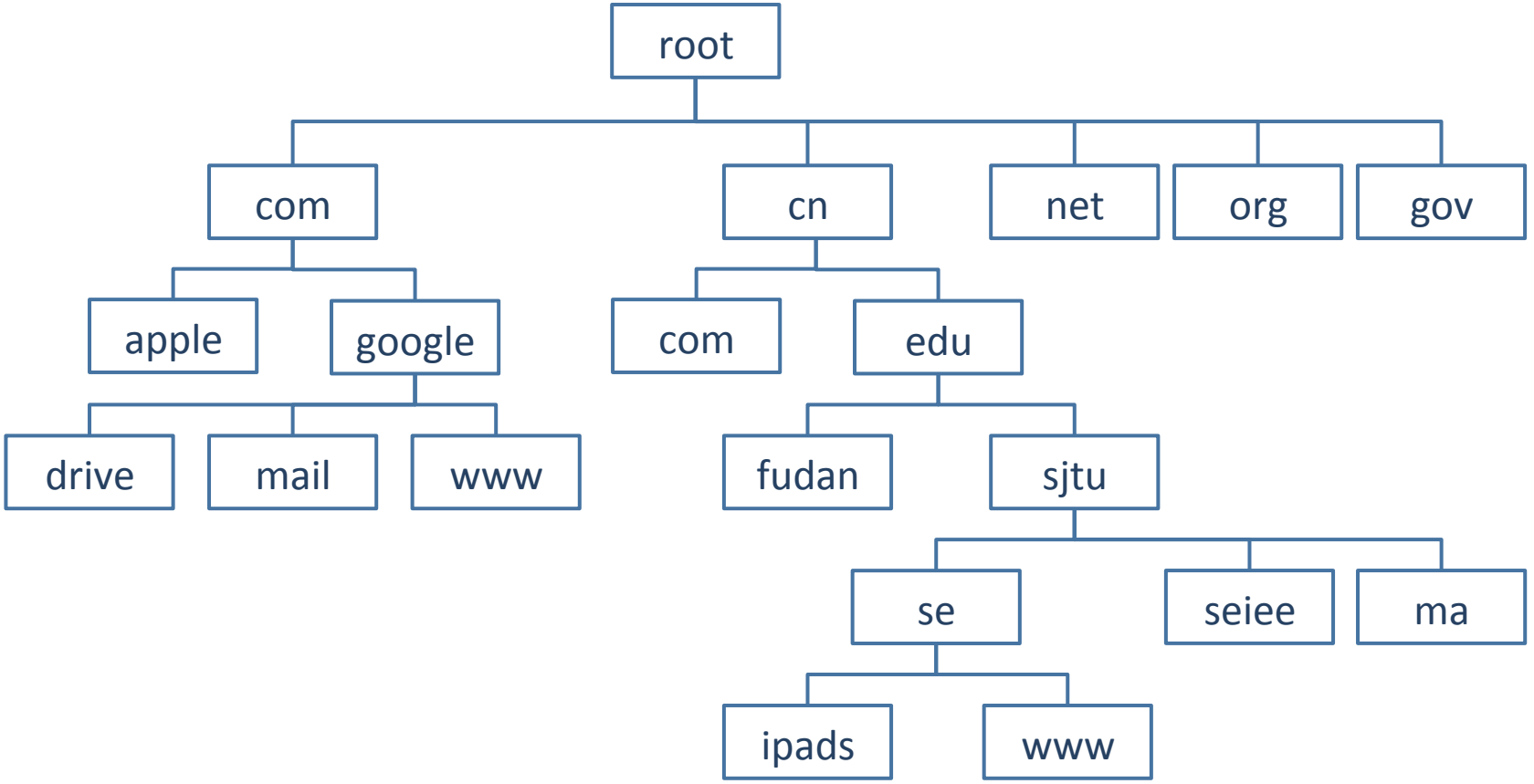
Distributing Responsibility

- The binding
  - Too large to be stored on a single machine
  - Thus, the data are stored on many machines
    - As known as "name servers"
- How to know which name server has a particular binding?
  - Solution: structure the hostname
  - Names have a hierarchy, e.g., com, net, gov, correspond to "zones"
  - Zones are mapped to name servers

Name Servers

- The root zone
  - Maintained by ICANN, non-profit
- The ".com" zone
  - Maintained by VeriSign, add for money
- The ".sjtu.edu.cn" zone
  - Maintained by SJTU

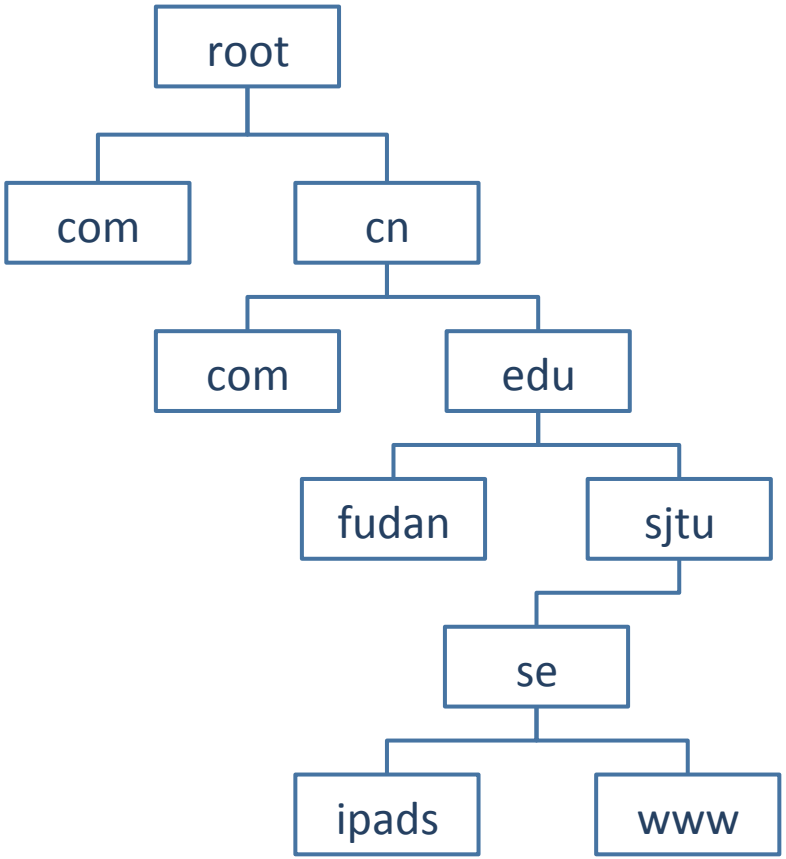
DNS Hierarchy (a partial view)



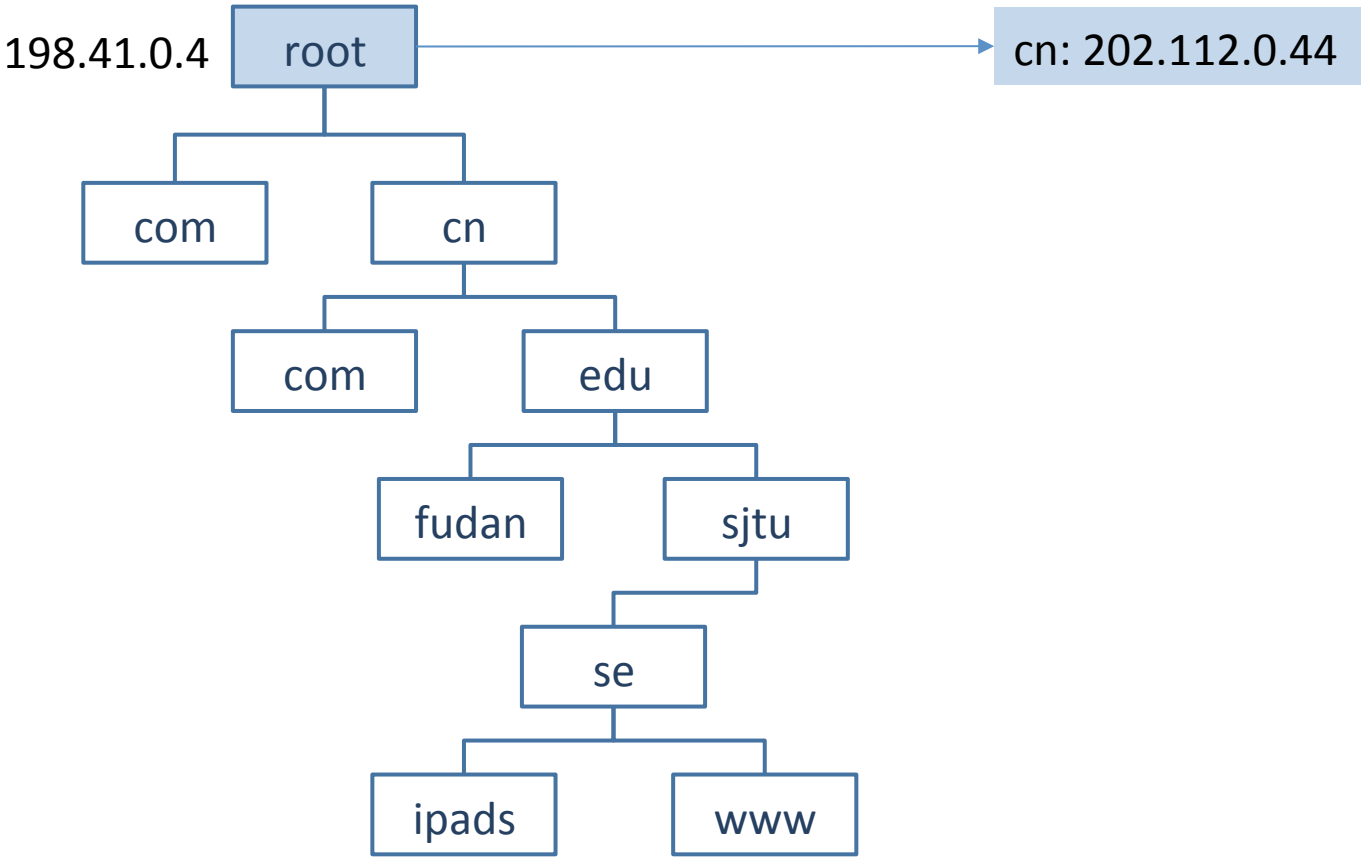
Basic DNS Look-up Algorithm

- Example: lookup IP of *"ipads.se.sjtu.edu.cn"*
- Traverse the name hierarchy from the root
  - The root will tell us the "cn" name server IP,
  - which will tell us the "edu.cn" name server IP,
  - which will tell us the "sjtu.edu.cn" name server IP,
  - which will tell us the "se.sjtu.edu.cn" name server IP,
  - which finally tells us the "ipads.se.sjtu.edu.cn" IP
- Such algorithm is called delegation

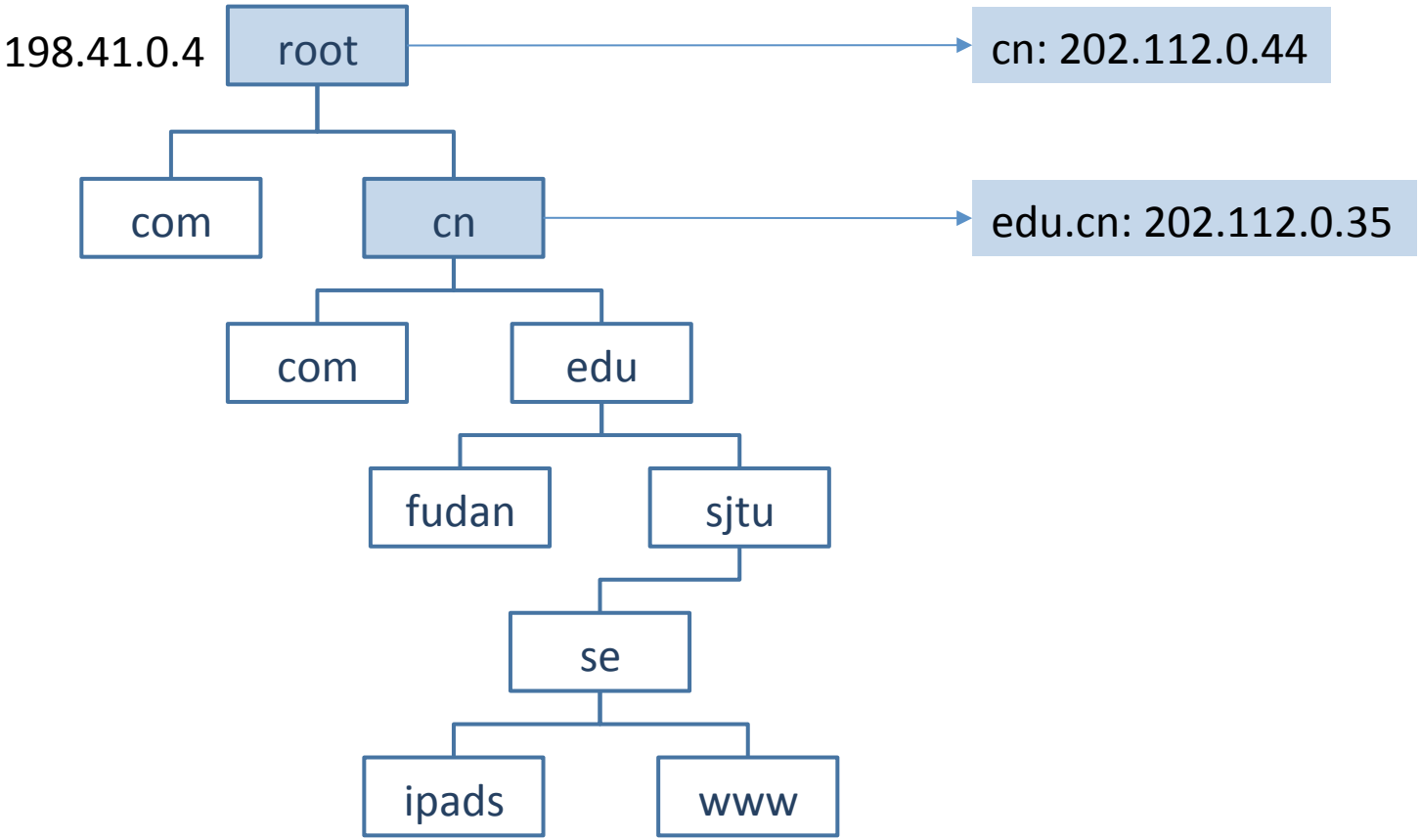
DNS Lookup



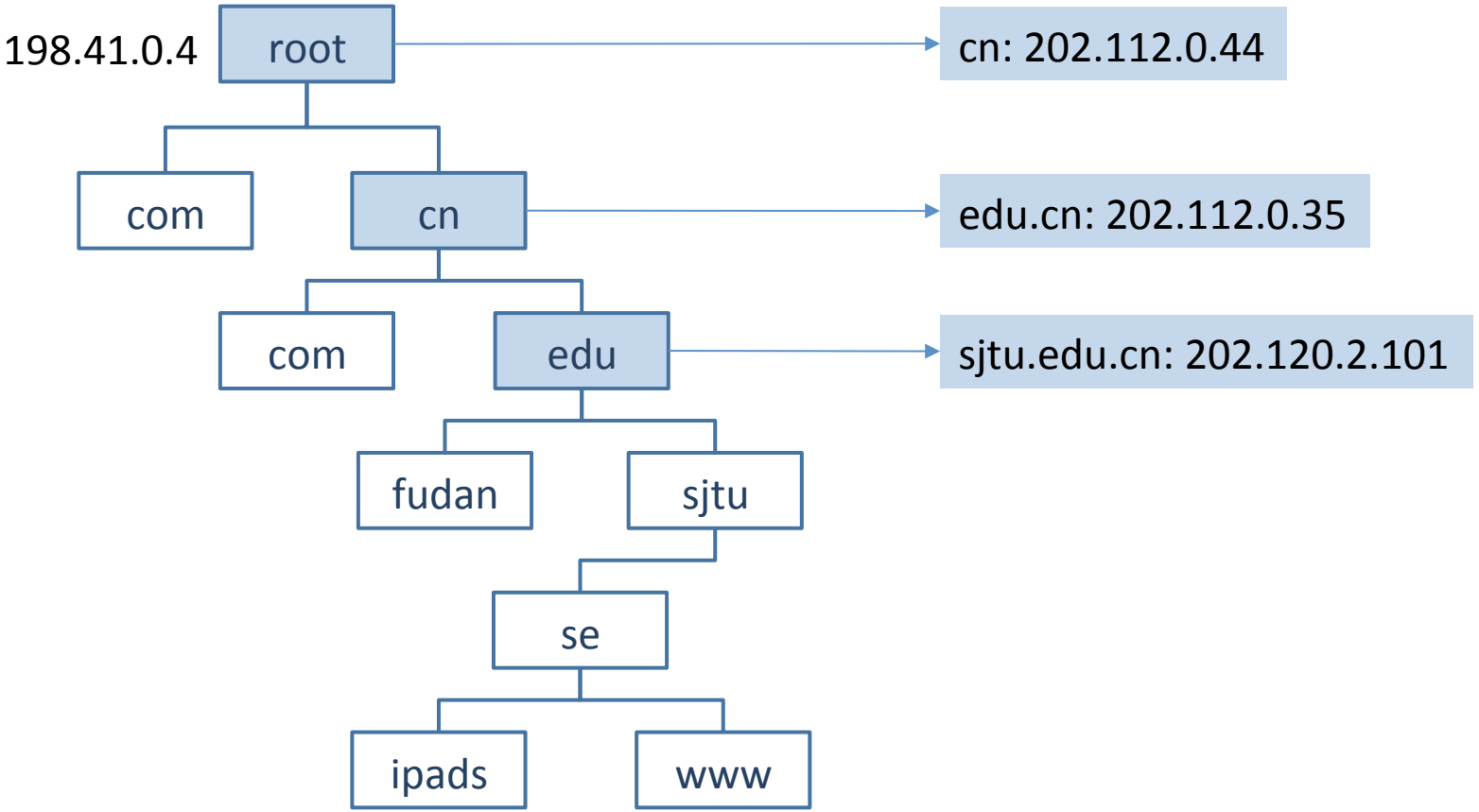
DNS Lookup


198.41.0.4
root
cn: 202.112.0.44
com
cn
com
edu
fudan
sjtu
se
ipads
www

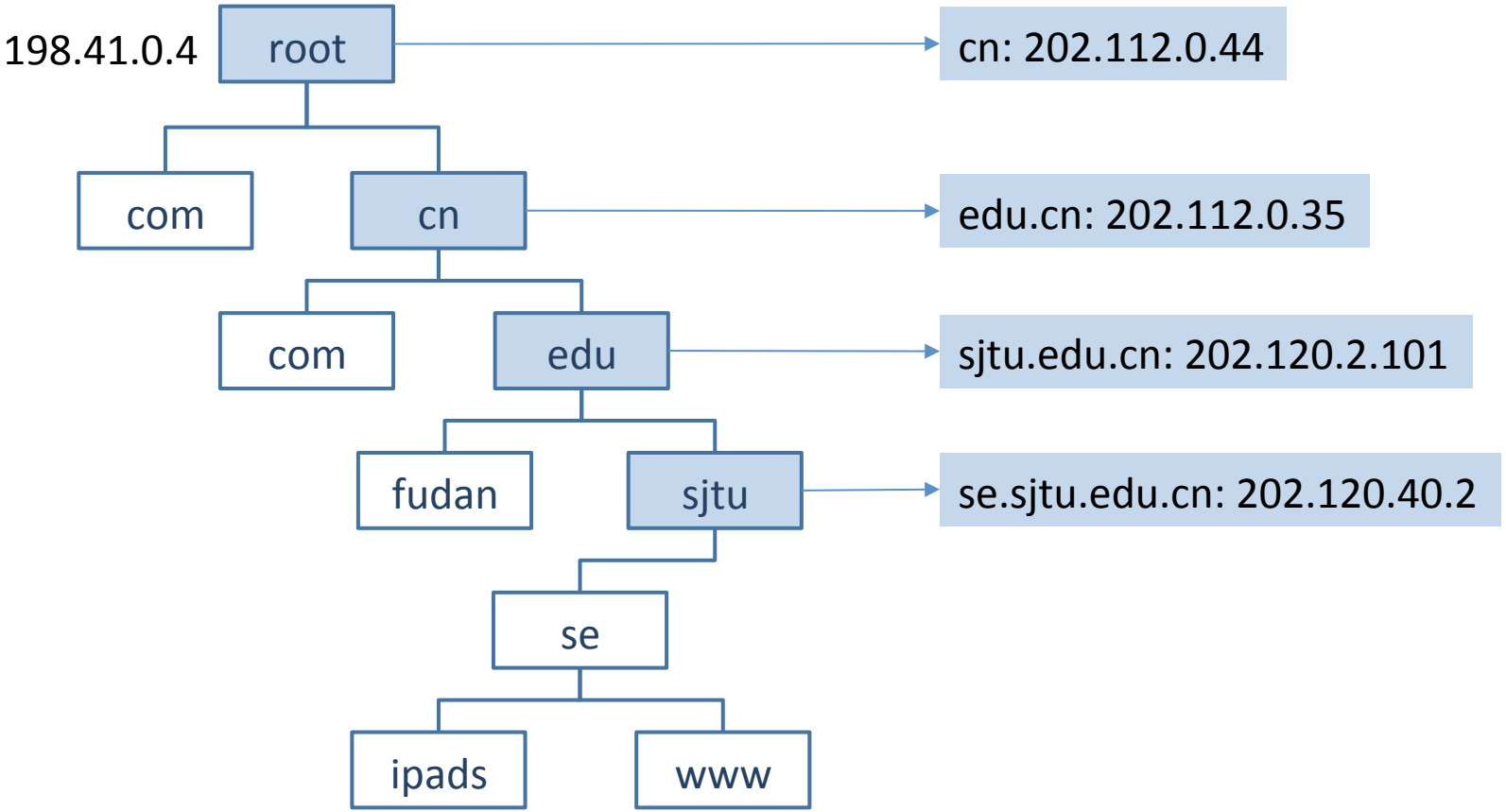
DNS Lookup


198.41.0.4
root
cn: 202.112.0.44
com
cn
edu.cn: 202.112.0.35
com
edu
fudan
sjtu
se
ipads
www

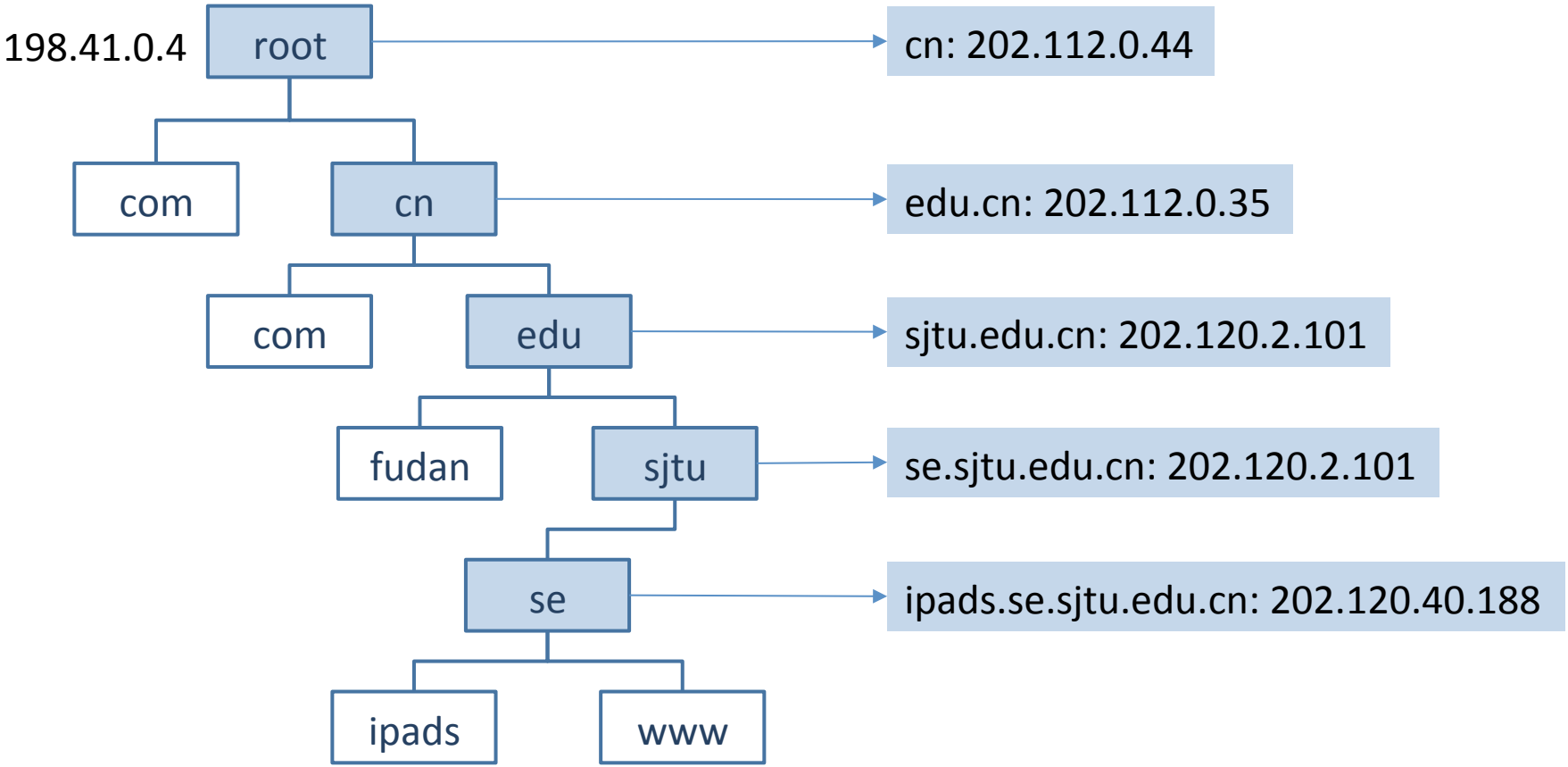
DNS Lookup


198.41.0.4
root
com
cn
com
edu
fudan
sjtu
se
ipads
www
cn: 202.112.0.44
edu.cn: 202.112.0.35
sjtu.edu.cn: 202.120.2.101

DNS Lookup


198.41.0.4
root
com
cn
com
edu
fudan
sjtu
se
ipads
www
cn: 202.112.0.44
edu.cn: 202.112.0.35
sjtu.edu.cn: 202.120.2.101
se.sjtu.edu.cn: 202.120.40.2

DNS Lookup


198.41.0.4
root
com
cn
com
edu
fudan
sjtu
se
ipads
www
cn: 202.112.0.44
edu.cn: 202.112.0.35
sjtu.edu.cn: 202.120.2.101
se.sjtu.edu.cn: 202.120.2.101
ipads.se.sjtu.edu.cn: 202.120.40.188

Context in DNS

- Names in DNS are global (context-free)
  - A hostname means the same thing everywhere in DNS
- Actually, it should be "ipads.se.sjtu.edu.cn."
  - A hostname is a list of domain names concatenated with dots
  - The root domain is unnamed, i.e., "." + blank
  - You can also look-up a hostname like "web"
    - Resolving "web."
    - Resolving "web" + additional default context (it's system specific)

Fault Tolerant

- Each zone can have multiple name servers
  - A delegation usually contains a list of name servers
  - If one name server is down, the others can be used

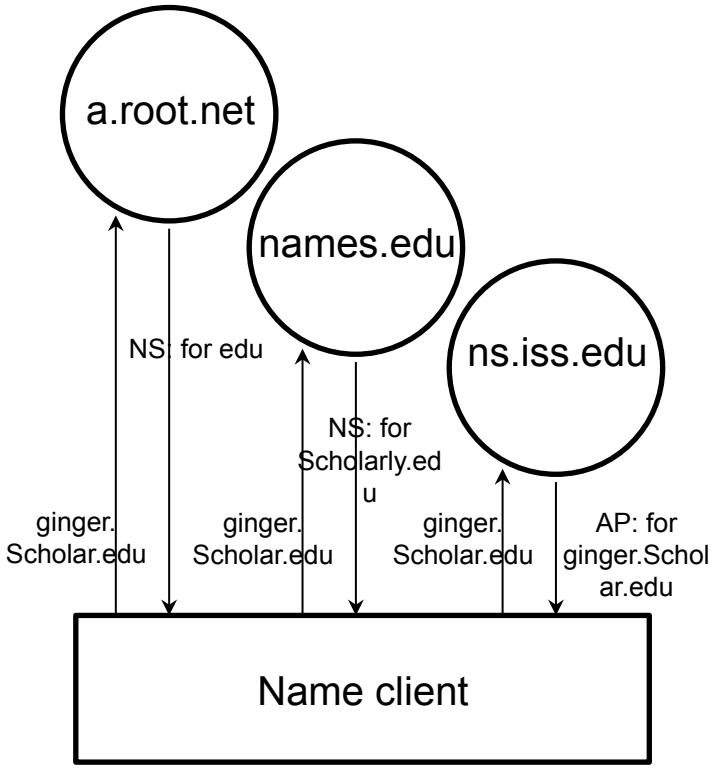
Three Enhancements on Look-up Algorithm

- 1. The initial DNS request can go to any name server, not just the root server
  - Even on your own machine: /etc/hosts
  - You can specific your name servers in /etc/resolv.conf
  - If no record, just returns address of the root server
  - *Question: what are the benefits?*

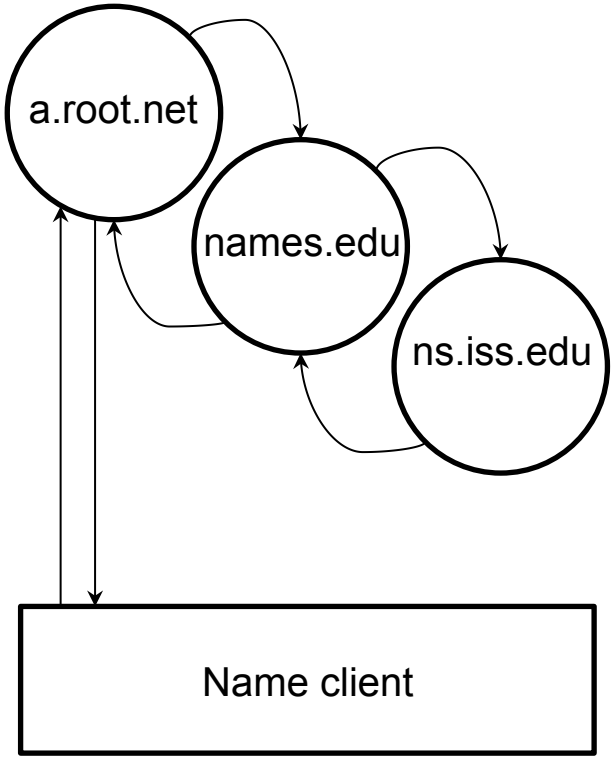
Three Enhancements on Look-up Algorithm

- 2. Recursion
  - A client asks a name server "www.baidu.com"
  - The name server does all the lookup through the tree and return the IP of baidu to the client
  - Usually, a name server has a better network connection

DNS Request Process



Non-Recursion



Recursion

Three Enhancements on Look-up Algorithm

- 3. Caching
  - DNS clients and name servers keep a cache of names
    - Your browser will not do two look-ups for one address
  - Cache has expire time limit
    - Controlled by a time-to-live parameter in the response itself
    - E.g., SJTU sets the TTL for *www.sjtu.edu.cn*
  - Trade-off
    - High TTLs VS. low TTLs (Question: what are the tradeoffs?)

Combine These Enhancements

- If
  - Many machines at SJTU use the SJTU name server for their initial DNS query
  - The name server offers recursive querying and caching
- Then
  - The name server's cache will holding many bindings
  - Performance benefits from this large cache

Other Features of DNS

- At least two identical replica servers
  - 80 replicas of the root name server in 2008
  - Replicas are placed separated around the world
- Organization's name server
  - Several replicas in campus
    - To enable communications within the organization
  - At least one out of the campus
    - To validate the address for outside world

Name Discovery in DNS

- A client must discover the name of a nearby name server
  - Name discovery broadcast to ISP at first time
  - Ask network manager
  - A user must discover the domain name of a service
  - Ask by email, Google

Why was DNS designed in this way?

BEHIND THE DESIGN

Benefits of Hierarchical Design

- Hierarchies delegate responsibility
- Each zone is only responsible for a small portion
- Hierarchies also limit interaction between modules

Good Points on DNS Design

- Global names (assuming same root servers)
  - No need to specific a context
  - DNS has no trouble generating unique names
  - The name can also be user-friendly
- Scalable in performance
  - Simplicity: look-up is simple and can be done by a PC
  - Caching
  - Delegation: many name severs handle lookups

Good Points on DNS Design

- Scalable in management
  - Each zone makes its own policy decision on binding
  - Hierarchy is great here
- Fault tolerant
  - If one name server breaks, other will still work
  - Duplicated name server for a same zone

Bad Points on DNS Design

- Policy
  - Who should control the root zone, .com zone, etc? Government?
- Significant load on root servers
  - Many DNS clients starts by talking to root server
  - Many queries for non-existent names, becomes a DoS
- Security
  - How does a client know if the response is correct?
  - How does VeriSign know "change Amazon.com IP" is legal?

DNS Security

- DNS Authentication
  - Cache inconsistency
- DNS Hijack
  - Cutting the binding between name and IP
- Solution: /etc/hosts, dnsmasq, OpenDNS, etc.
  - DNS DoS
  - BAOFENG.com & DNSPod
  - 2009-5-18: DNSPod is attacked and banned
  - 2009-5-19: The Internet in China is almost down
- DNS shield to defend against DoS attack

Problem 4.5

- While browsing the Web, you click on a link that identifies an Internet host named www.cslab.scholarly.edu. Your browser asks your Domain Name System (DNS) name server, *M*, to find an Internet address for this domain name. Under what conditions is each of the following statements true of the name resolution process?

Problem 4.5

A. To answer your query, *M* must contact one of the root name servers.

B. If *M* answered a query for *www.cslab.scholarly.edu* in the past, then it can answer your query without asking any other name server.

C. M must contact one of the name servers for cslab.scholarly.edu to resolve the domain name.

Problem 4.5

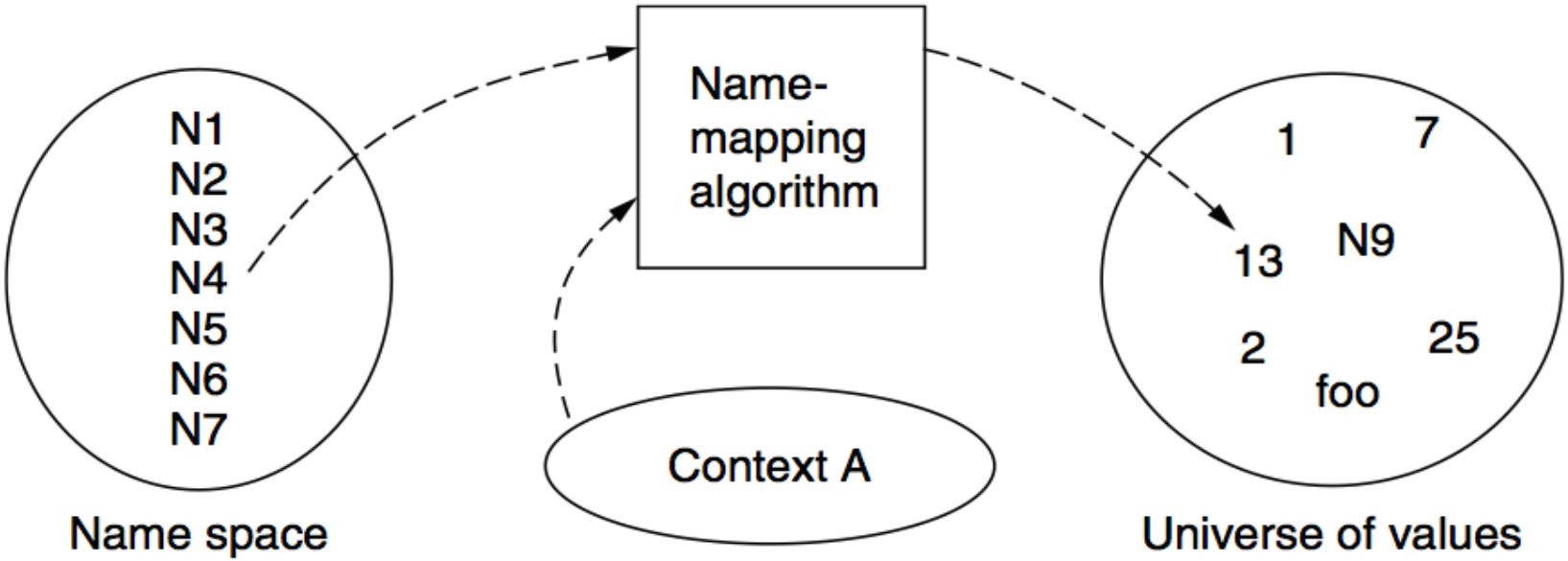
D. If *M* has the current Internet address of a working name server for *scholarly.edu* cached, then that name server will be able to directly provide an answer.

E. If *M* has the current Internet address of a working name server for *cslab.scholarly.edu* cached, then that name server will be able to directly provide an answer.

Naming in File System

A demonstration of naming, modularity and layering

Review: Naming Model



Review: Use Name to Achieve Modularity

- Retrieval: e.g., using URL to get a web page
- Sharing: e.g., passing an object reference to a function
  - Save space as well: only sending the name, not the object
- Hiding: e.g., using a file name without knowing file system
  - Can support access control: use an object only if knowing its name
  - E.g., Windows has many undocumented API
- User-friendly identifiers: e.g., "homework.txt" instead of 0x051DE540
- Indirection: e.g., OS can move the location of the file data without notifying the user
  - Have you ever defragmented your hard driver?

Review: Good Points on DNS Design

- Global names (assuming same root servers)
  - No need to specific a context
  - DNS has no trouble generating unique names
  - The name can also be user-friendly
- Scalable in performance
  - Simplicity: look-up is simple and can be done by a PC
  - Caching
  - Delegation: many name severs handle lookups

Review: Good Points on DNS Design

- Scalable in management
  - Each zone makes its own policy decision on binding
  - Hierarchy is great here
- Fault tolerant
  - If one name server breaks, other will still work
  - Duplicated name server for a same zone

Review: Bad Points on DNS Design

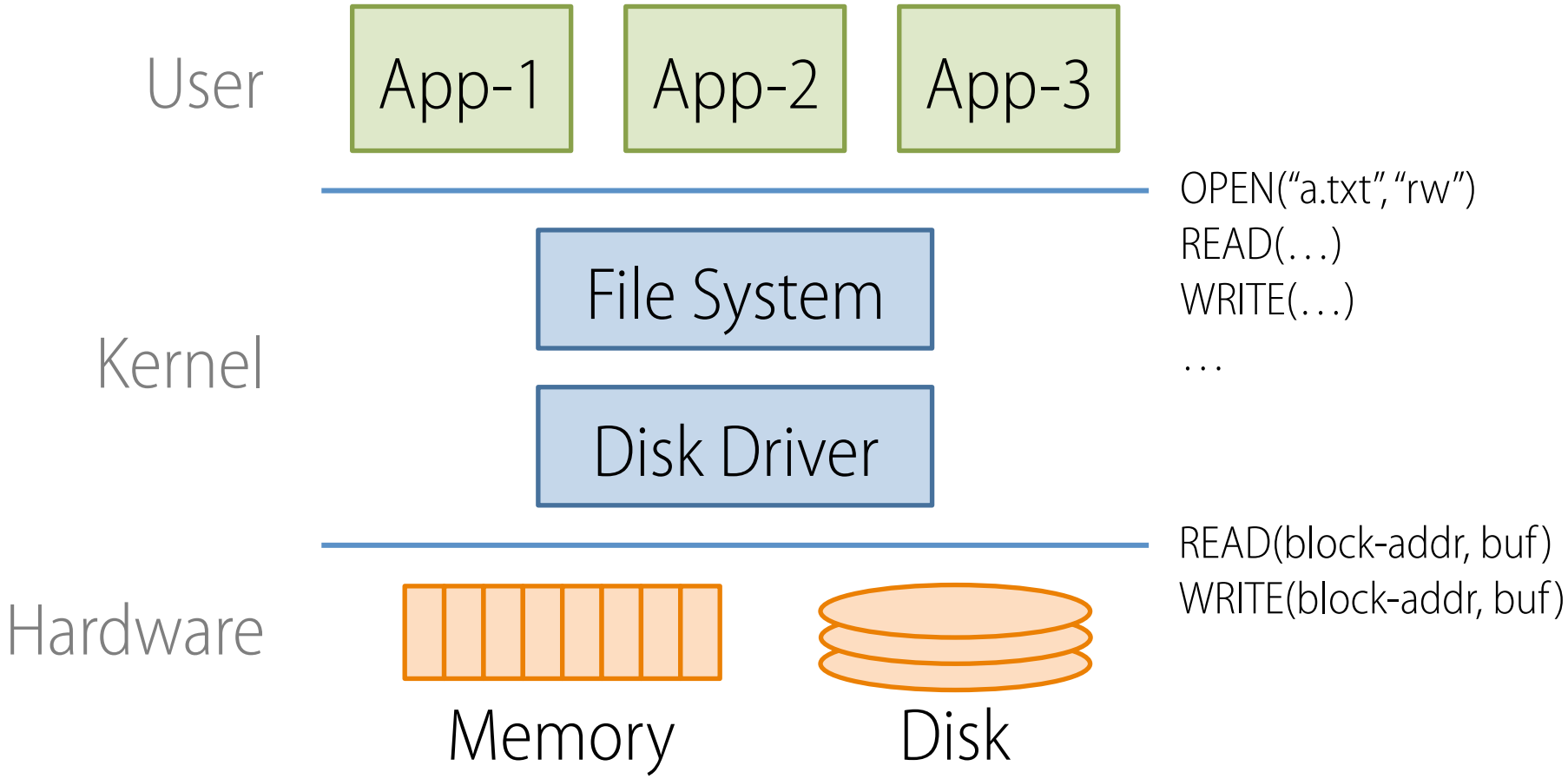
- Policy
  - Who should control the root zone, .com zone, etc? Government?
- Significant load on root servers
  - Many DNS clients starts by talking to root server
  - Many queries for non-existent names, becomes a DoS
- Security
  - How does a client know if the response is correct?
  - How does VeriSign know "change Amazon.com IP" is legal?

FILE SYSTEM OF UNIX-V6

File

- File is a high-level version of the memory abstraction
  - *Recall: Abstraction VS. Virtualization*
- A file has two key properties
  - It is durable & has a name
- System layer implements files using modules from hardware layer
  - Divide-and-conquer strategy
  - Makes use of several hidden layers of machine-oriented names (addresses), one on another, to implement files
  - Maps user-friendly names to these files

The Big Picture



Abstraction: API of UNIX File System

- OPEN, READ, WRITE, SEEK, CLOSE
- FSYNC
- STAT, CHMOD, CHOWN
- RENAME, LINK, UNLINK, SYMLINK
- MKDIR, CHDIR, CHROOT
- MOUNT, UNMOUNT

FILE SYSTEM: SOFTWARE LAYER

The Naming Layers of the UNIX FS (version 6)

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
Path name layer	Organize files into naming hierarchies.	↓
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑
File layer	Organize blocks into files.	machine-oriented names
Block layer	Identify disk blocks.	↓

Block Layer

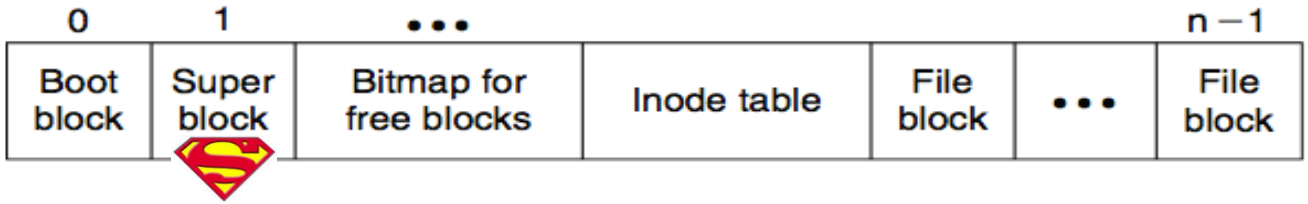
Block num	Disk Block

- Block size: a trade-off
  - Neither too small or too big
- Name mapping: block number -> block
  - *Name-mapping algorithm*
    - **procedure** BLOCK_NUMBER_TO_BLOCK (**integer** *b*) **returns** *block*
      return *device*[*b*]
  - *Context*
    - The storage device (e.g. disk) itself
    - Binds block numbers to physical blocks
  - *Name discovery*
    - Super block 

Super Block

Block num	Disk Block

- One superblock per file system
  - Kernel reads superblock when mount the FS
- Superblock contains
  - Size of the blocks
  - Number of free blocks
  - A list of free blocks
  - Index to next free block
  - Lock field for free block and free inode lists
  - Flag to indicate modification of superblock
  - Size of the inode list
  - Number of free inodes
  - A list of free inodes
  - Index to next free inode



File Layer

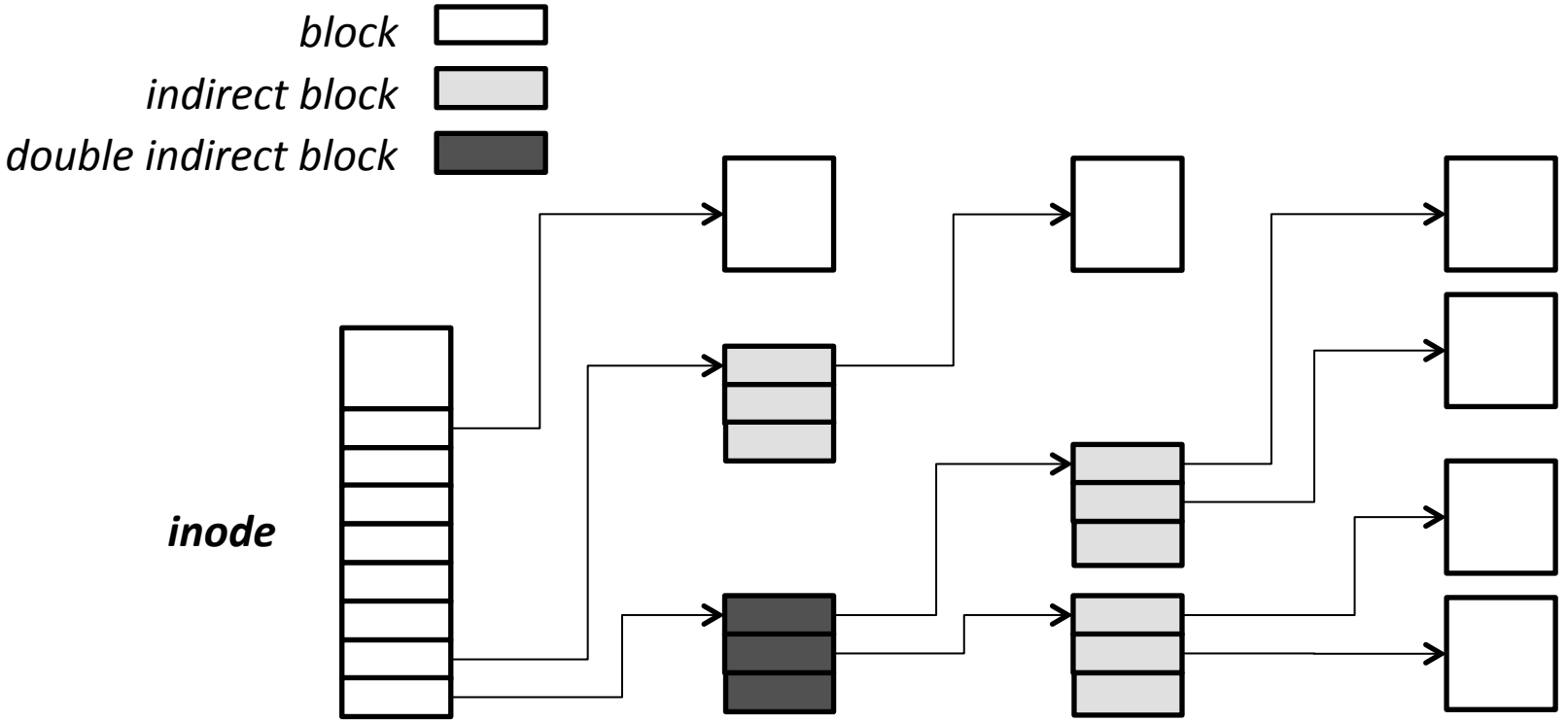
File (inode)	Block num	Disk Block

- File requirements
  - Store items that are larger than one block
  - May grow or shrink over time
  - A file is a linear array of bytes of arbitrary length
  - Record which blocks belong to each file
- inode (index node)
  - A container for metadata about the file

```
structure inode
    integer block_numbers[N]
    integer size
```

inode for Larger Files

File (inode)	Block num	Disk Block



File Layer

File (inode)	Block num	Disk Block

- Name mapping: index number -> block number
  - Index number can be seen as block-size *offset* within a file
- Context: the inode itself
- Name mapping algorithm

```
procedure INDEX_TO_BLOCK_NUMBER (inode instance i, integer index) returns integer
    return i.block_numbers[index]

procedure INODE_TO_BLOCK (integer offset, inode instance i) returns block
    o ← offset / BLOCKSIZE
    b ← INDEX_TO_BLOCK_NUMBER (i, o)
    return BLOCK_NUMBER_TO_BLOCK (b)
```

Choices Other Than inode

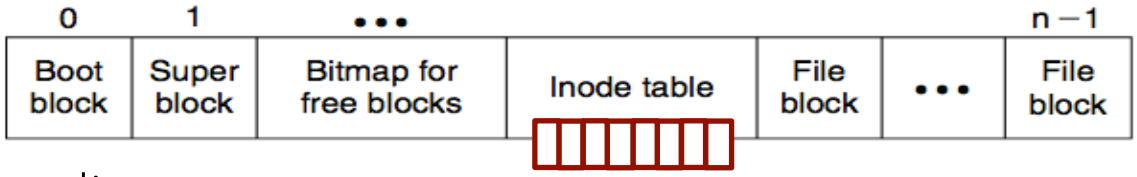
- Method-1:
  - Use continue blocks
  - Re-allocate if the file expands
  - E.g., data in memory
  - Why not?
- Method-2: Use Linked List
  - Each block links to its next block
  - Use special one as EOF (End of File)
  - E.g., FAT32
  - Why not?
- How to integrate different FS?
  - vnode
  - Interface is similar with inode

inode Number Layer

Inode num	File (inode)	Block num	Disk Block

- Name mapping: inode number -> inode
- Context: the inode table
- Name-mapping algorithm: inode table
  - At a fixed location on storage

```
procedure INODE_NUMBER_TO_INODE (integer inode_number) returns inode
    return inode_table[inode_number]
```



- Name discovery
  - Track which inode number are in use
  - E.g. free list, a field in inode

Put Layers so far Together

Inode num	File (inode)	Block num	Disk Block

```
1    procedure INODE_NUMBER_TO_BLOCK (integer offset, integer inode_number)
2                                                    returns block
3        inode instance i ← INODE_NUMBER_TO_INODE (inode_number)
4        o ← offset / BLOCKSIZE
5        b ← INDEX_TO_BLOCK_NUMBER (i, o)
6        return BLOCK_NUMBER_TO_BLOCK (b)
```

- Needs more user-friendly name
  - Numbers are convenient names only for computer
- Numbers change on different storage device

File Name Layer

File name	Inode num	File (inode)	Block num	Disk Block

- File name
  - Hide metadata of file management
  - Files and I/O devices

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
```

File name	Inode number
program	10
paper	12

- Name mapping algorithm
  - Mapping table saved in directory
  - Default context: current working directory
  - Context reference is also inode number
    - The directory itself is a file
  - **procedure** NAME_TO_INODE_NUMBER (**character string** *filename*, **integer** *dir*) **returns integer**
    return LOOKUP (*filename*, *dir*)
  - Max length of a name is 14 bytes in UNIX version 6

LOOKUP in a Directory

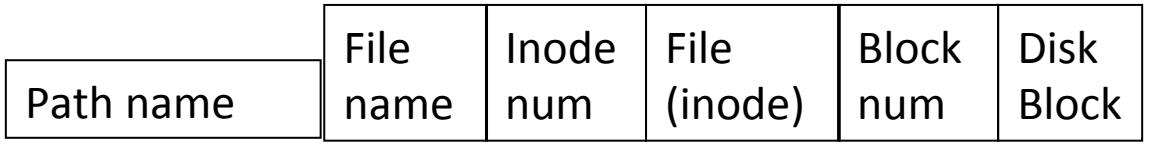
File name	Inode num	File (inode)	Block num	Disk Block

```
1     procedure LOOKUP (character string filename, integer dir) returns integer
2         block instance b
3         inode instance i ← INODE_NUMBER_TO_INODE (dir)
4         if i.type ≠ DIRECTORY then return FAILURE
5         for offset from 0 to i.size – 1 do
6             b ← INODE_NUMBER_TO_BLOCK (offset, dir)
7             if STRING_MATCH (filename, b) then
8                 return INODE_NUMBER (filename, b)
9             offset ← offset + BLOCKSIZE
10        return FAILURE
```

- Name compare method: STRING_MATCH
- LOOKUP("program", dir) will return 10

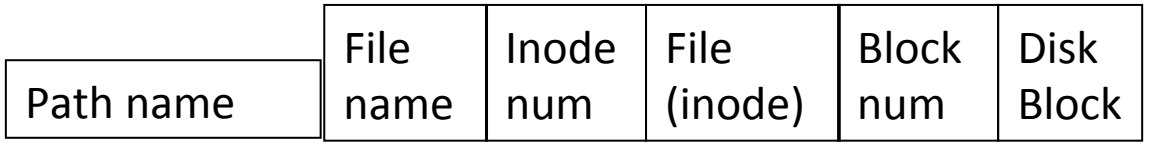
- Next Problem: too many files

Path Name Layer



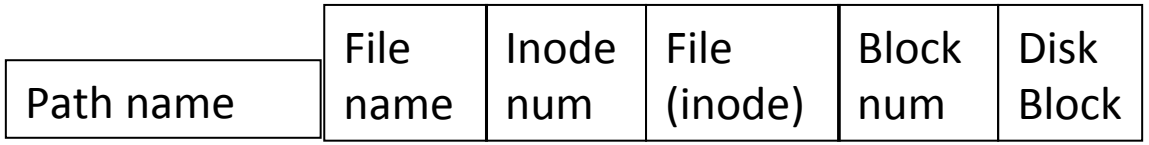
- Hierarchy of directories and files
  - Structured naming: E.g. "projects/paper"
- Name-mapping algorithm
  -
```
procedure PATH_TO_INODE_NUMBER (character string path, integer dir) returns integer
    if (PLAIN_NAME (path)) return NAME_TO_INODE_NUMBER (path, dir)
    else
        dir ← LOOKUP (FIRST (path), dir)
        path ← REST (path)
        return PATH_TO_INODE_NUMBER (path, dir)
```
  - PLAIN_NAME returns true if no '/' in the path
- Context: the working directory

Links



- LINK: shortcut for long names
  - LINK("Mail/inbox/new-assignment", "assignment")
  - Turns strict hierarchy into a directed graph
    - Users cannot create links to directories -> acyclic graph
  - Different names, same inode number
- UNLINK
  - Remove the binding of filename to inode number
  - If UNLINK last binding, put inode/blocks to free-list
    - A reference counter is needed

Links

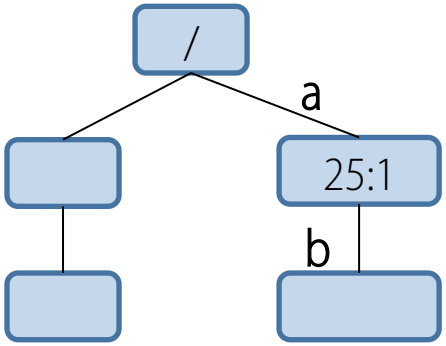


- Reference count
  - An inode can bind multiple file names
  - +1 when LINK, -1 when UNLINK
  - A file will be deleted when reference count is 0
  - No cycle allowed
    - Except for '.' and '..'
    - Naming current and parent directory with no need to know their names

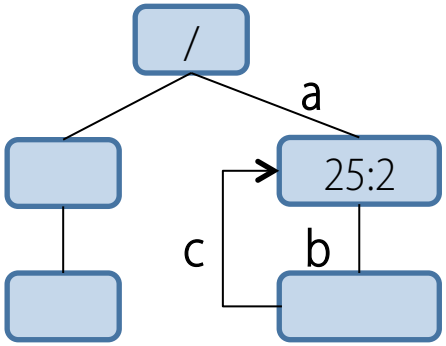
```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
```

No Cycle for LINK

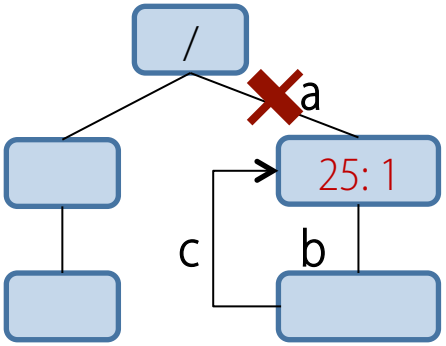
Path name	File name	Inode num	File (inode)	Block num	Disk Block



- /a/b is a directory
- The refcnt of a is 1
- a's inode num is 25



- LINK ("/a/b/c", a")
- Cause a cycle!
- Refcnt of a is 2



- UNLINK ("/a")
- Refcnt of a is 1, so the inode 25 is not deleted
- Now inode 25 is dis-connected from graph

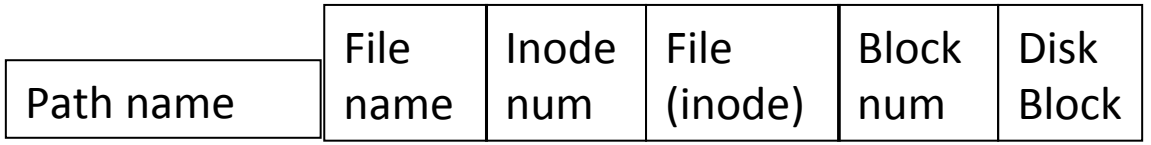
Renaming - 1

Path name	File name	Inode num	File (inode)	Block num	Disk Block

```
1    UNLINK (to_name)
2    LINK (from_name, to_name)
3    UNLINK (from_name)
```

- Text edit usually save editing file in a tmp file
  - Edit in .a.txt.swp, then rename .a.txt.swp to a.txt
- What if the computer fails between 1 & 2?
  - *to_name* will be lost, which surprises the user
  - Need atomic action in chap-9

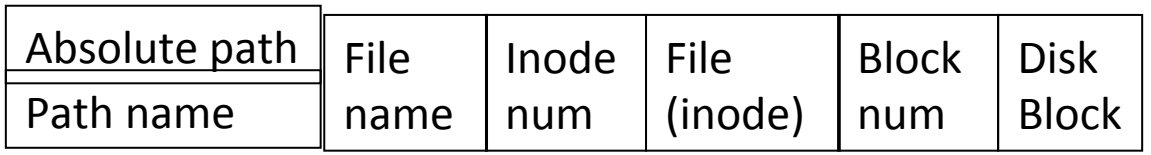
Renaming - 2




```
1    LINK (from_name, to_name)
2    UNLINK (from_name)
```

- Weaker specification without atomic actions
  - Changes the inode number in the directory entry for *to_name* to the inode number of *from_name*
  - Removes the directory entry for *from_name*
  - If fails between 1 & 2, must increase reference count of *from_name*'s inode on recovery
  - If *to_name* already exist, it will always exist even if machine fails between 1 & 2
  - (The *to_name* file is UNLINKed in the first LINK function)

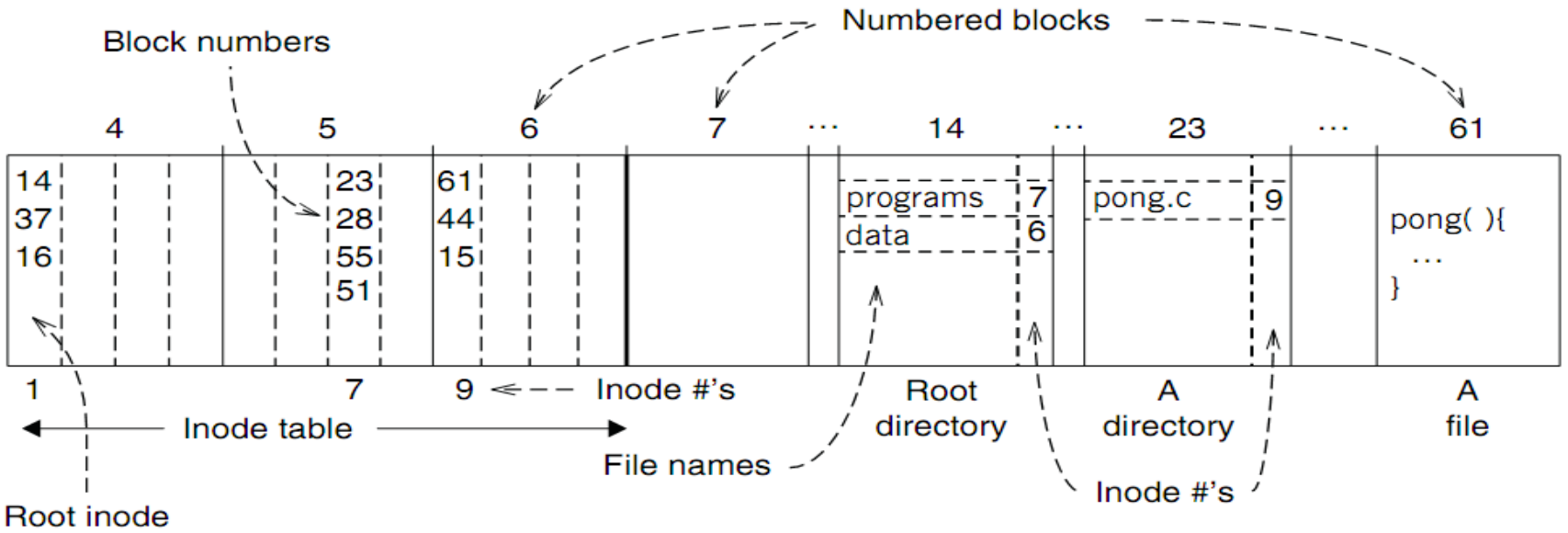
Absolute Path Name Layer



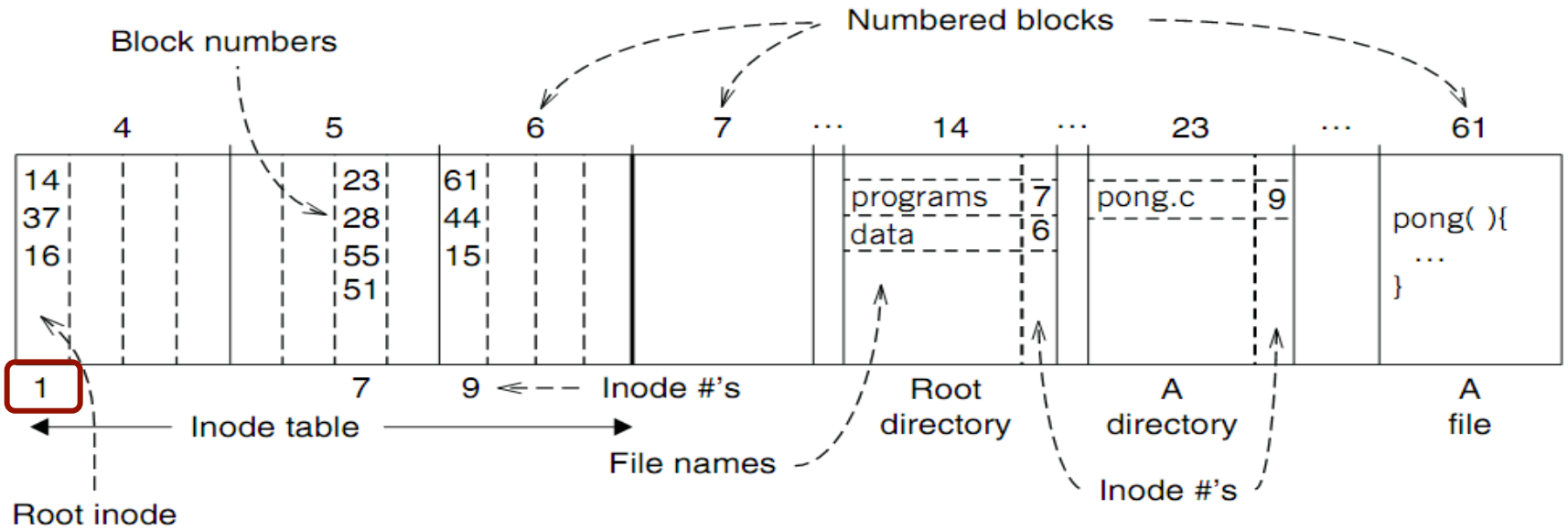
- HOME directory
  - Every user's default working directory
  - Problem: no sharing of HOME files between users
- Context: the root directory
  - A universal context for all users
  - Well-known name: '/'
  - Both '/.' and '/..' are linked to '/'

```
procedure GENERALPATH_TO_INODE_NUMBER (character string path) returns integer
    if (path[0] = "/") return PATH_TO_INODE_NUMBER(path, 1)
    else return PATH_TO_INODE_NUMBER(path, wd)
```

An Example: Find Blocks of "/programs/pong.c"

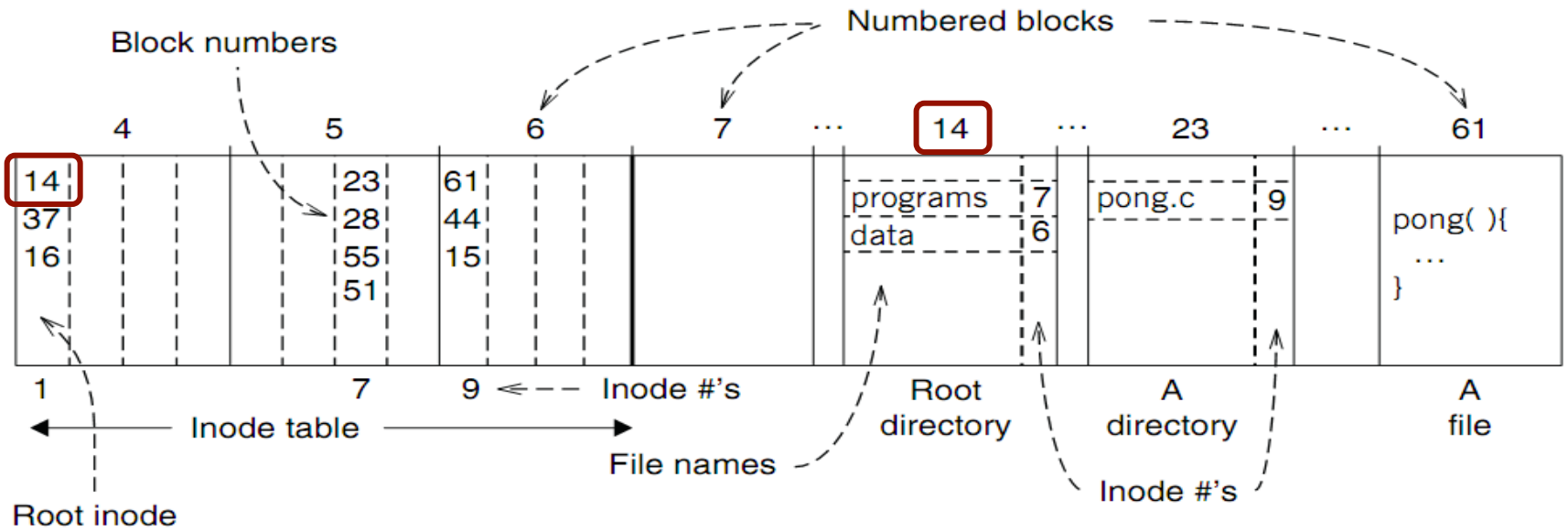


An Example: Find Blocks of "/programs/pong.c"



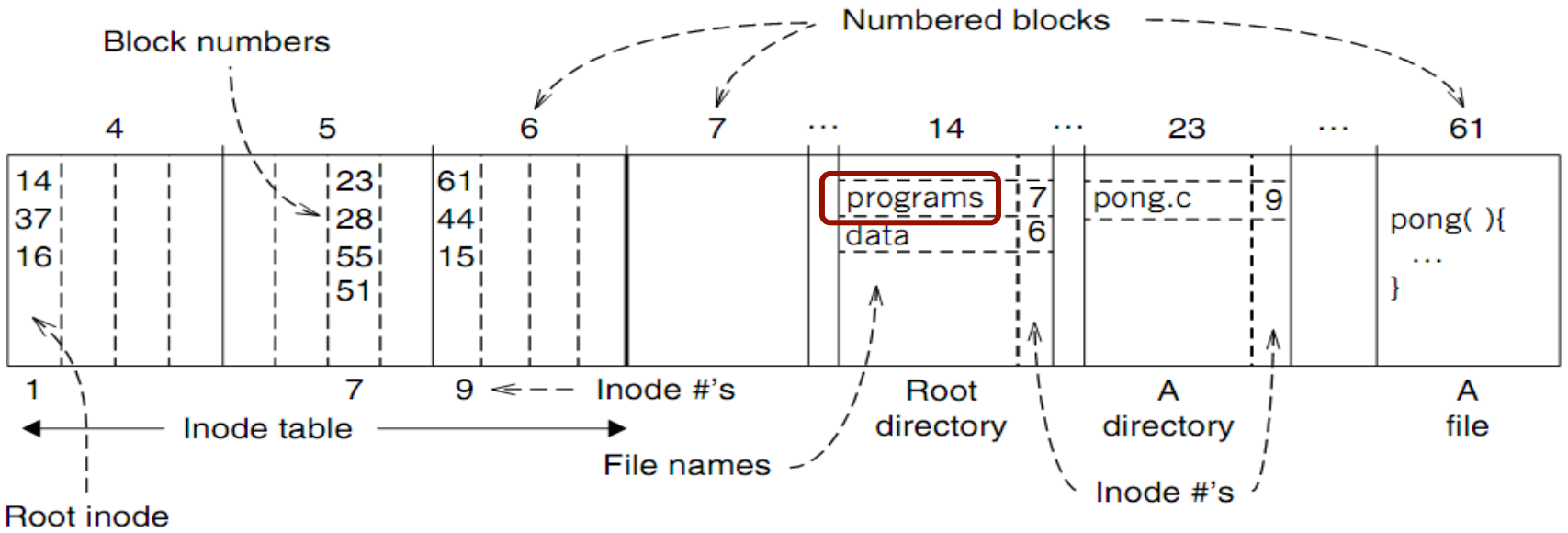
- '/' root directory: inode is 1

An Example: Find Blocks of "/programs/pong.c"



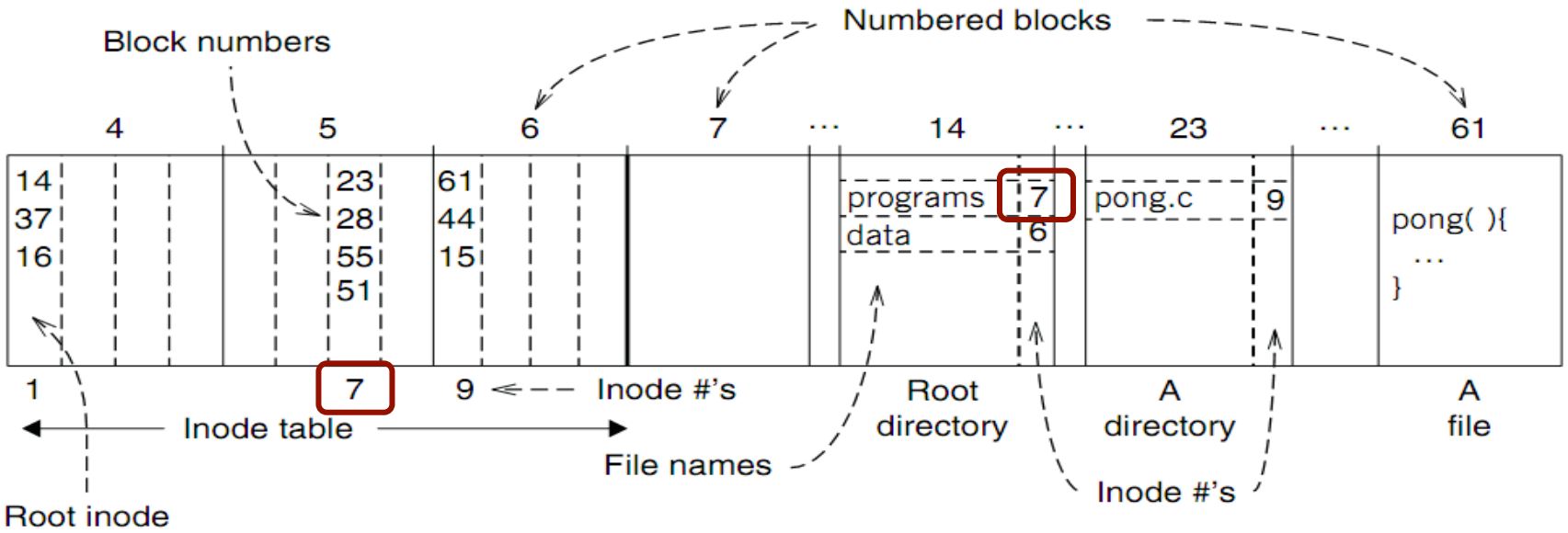
- Find the first directory in '/' by block number

An Example: Find Blocks of "/programs/pong.c"



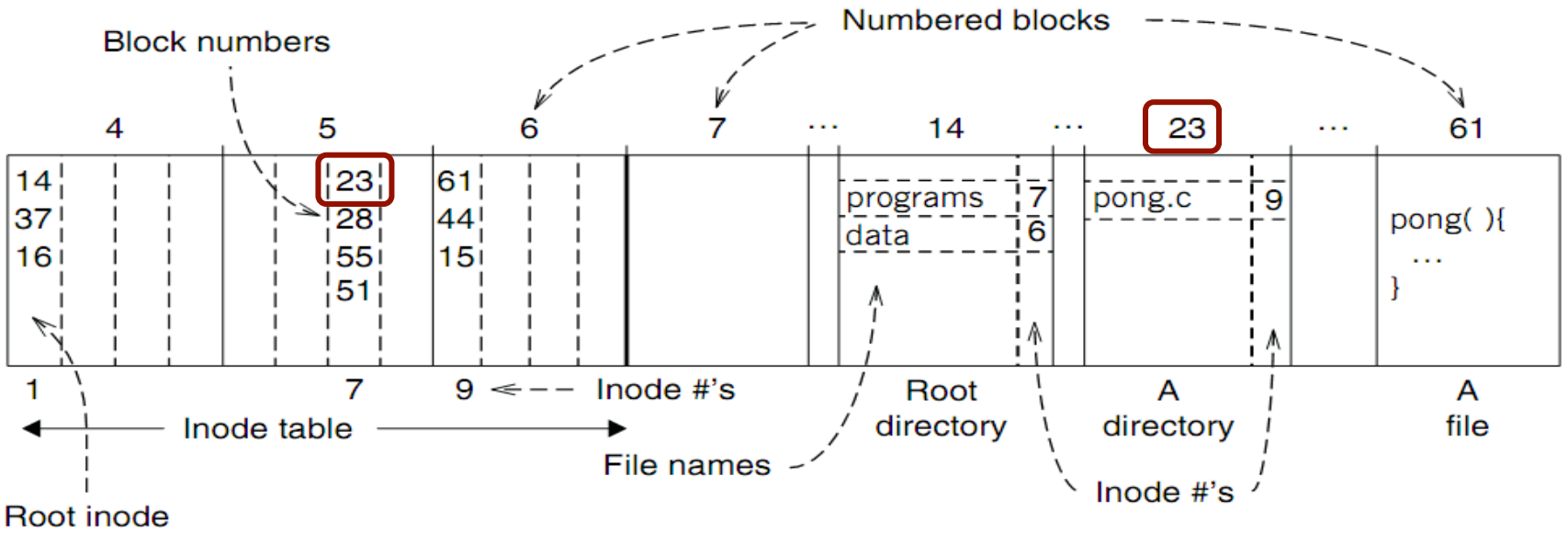
- Find '/programs' by comparing name

An Example: Find Blocks of "/programs/pong.c"



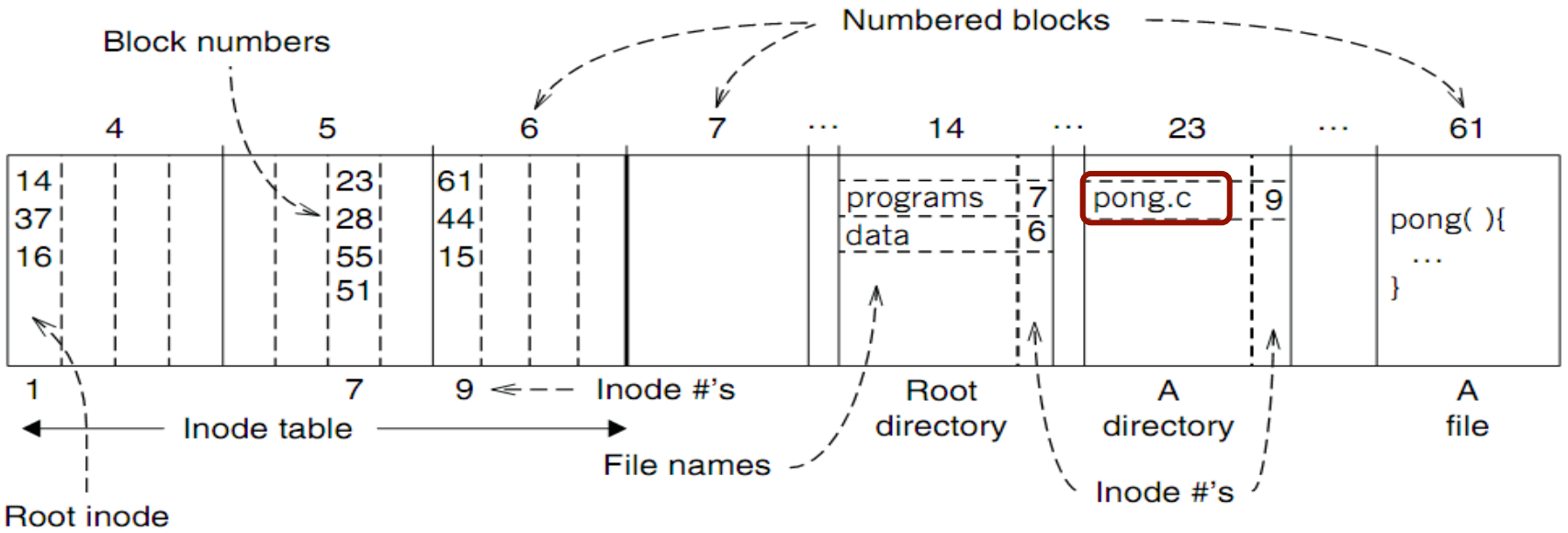
- Find '/programs' inode by its inode number 7

An Example: Find Blocks of "/programs/pong.c"



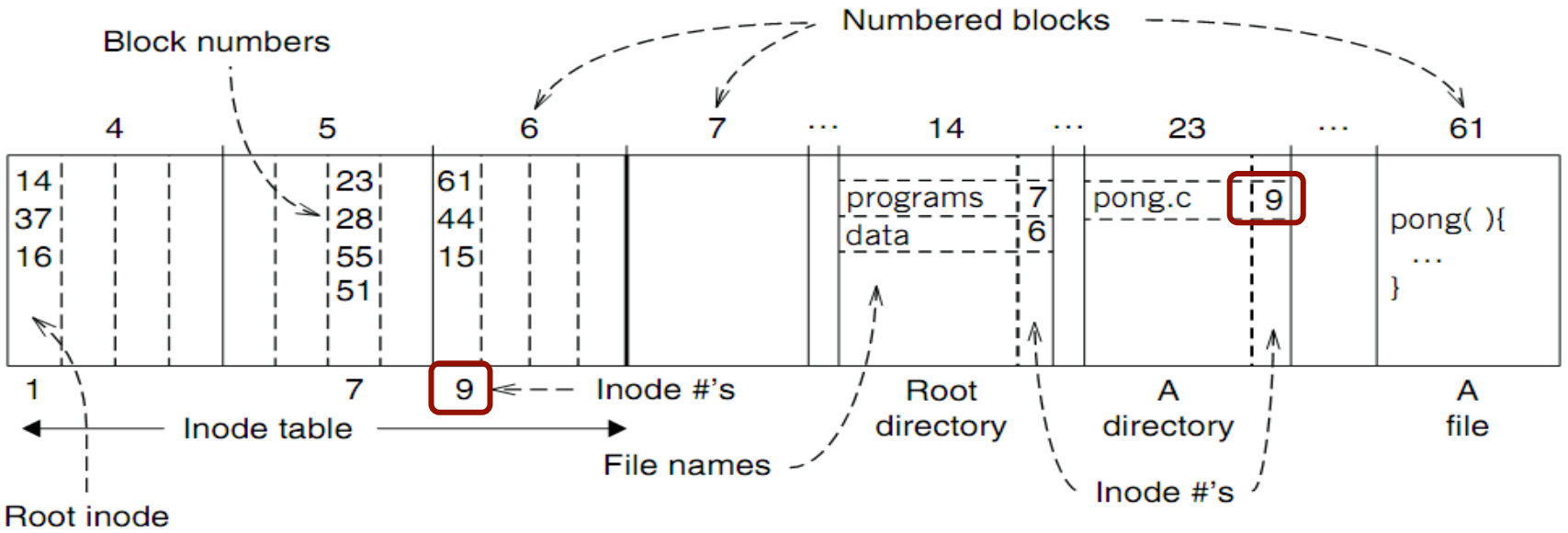
- Find the first file in '/programs/'

An Example: Find Blocks of "/programs/pong.c"



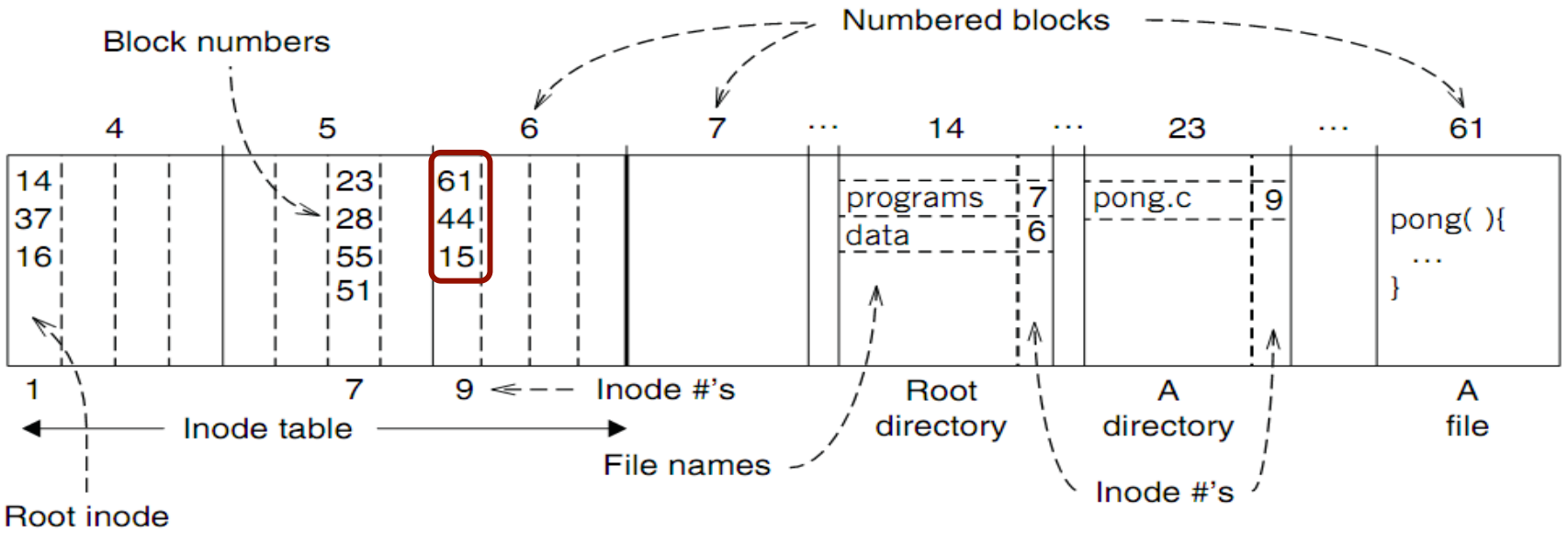
- Find '/programs/pong.c' by comparing its name

An Example: Find Blocks of "/programs/pong.c"



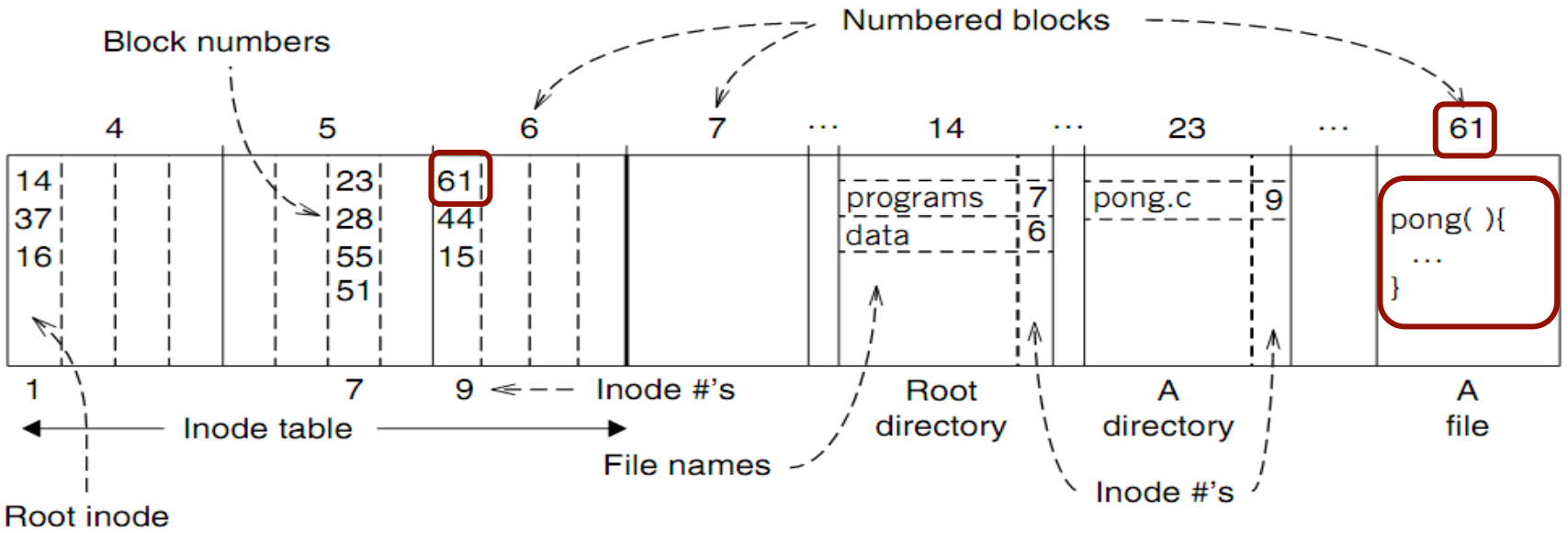
- Find inode of '/programs/pong.c' by the inode number 9

An Example: Find Blocks of "/programs/pong.c"



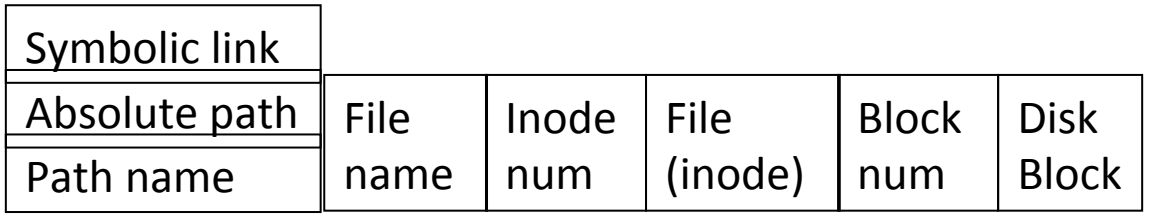
- Find block number of '/programs/pong.c'

An Example: Find Blocks of "/programs/pong.c"



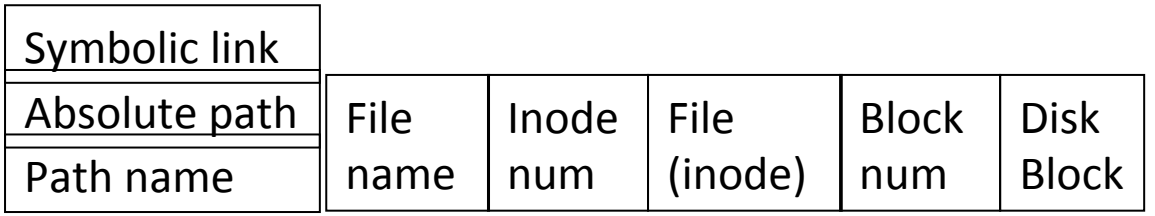
- Find data of block 61 by its block number
  - And data of block 44 & 15

Symbolic Link Layer



- MOUNT
  - Records the device and the root inode number of the file system in memory
  - Record in the in-memory version of the inode for "/dev/fd1" its parent's inode
  - UNMOUNT undoes the mount
- Change to the file name layer
  - If LOOKUP runs into an inode on which a file system is mount, it uses the root inode of that file system for the lookup

Symbolic Link Layer



- Name files on other disks
  - Inode is different on other disks
  - Supports to attach new disks to the name space
- Two options
  - Make inodes unique across all disks ✗
  - Create synonyms for the files on the other disks
- Soft link (symbolic link)
  - SYMLINK
  - Add another type of inode
  - Context: the directory hierarchy

Naming in File System

A demonstration of naming, modularity and layering

Review: File

- File is a high-level version of the memory abstraction
  - *Recall: Abstraction VS. Virtualization*
- A file has two key properties
  - It is durable & has a name
- System layer implements files using modules from hardware layer
  - Divide-and-conquer strategy
  - Makes use of several hidden layers of machine-oriented names (addresses), one on another, to implement files
  - Maps user-friendly names to these files

Review: The Naming Layers of the UNIX FS (version 6)

Layer	Purpose	
Symbolic link layer	Integrate multiple file systems with symbolic links.	↑
Absolute path name layer	Provide a root for the naming hierarchies.	user-oriented names
Path name layer	Organize files into naming hierarchies.	↓
File name layer	Provide human-oriented names for files.	machine-user interface
Inode number layer	Provide machine-oriented names for files.	↑
File layer	Organize blocks into files.	machine-oriented names
Block layer	Identify disk blocks.	↓

Review: Two Types of Links (Synonyms)

- Add link "assignment" to "Mail/new-assignment"
  - Hard link
    - No new file is created, just add a binding between a string and an existing inode
    - Target inode reference count is increased
    - If target file is deleted, the link is still valid
  - Soft link
    - A new file is created, the data is the string "Mail/new-assignment"
    - Target inode reference count is not increased
    - If target file is deleted, the link is not valid
- Soft link can create cycle by SYMLINK("a", "a")

Symbolic Link Layer

- Another interesting behavior of soft link
  - Current directory is "/Scholarly/programs/www"
  - This *wd* contains a soft link
    - "CSE-web" -> "Scholarly/programs/www"
  - Run following commands
    - CHDIR ("CSE-web")
    - CHDIR ("..")
  - What is the current directory?
    - ".." is resolved in the new default context, by bash

Decouple Modules with Indirection

Table 2.3 The UNIX Naming Layers, with Details of the Naming Scheme of Each Layer

Layer	Names	Values	Context	Name-Mapping Algorithm	
Symbolic link	Path names	Path names	The directory hierarchy	PATHNAME_TO_GENERAL_PATH	↑
Absolute path name	Absolute path names	Inode numbers	The root directory	GENERALPATH_TO_INODE_NUMBER	user-oriented names
Path name	Relative path names	Inode numbers	The working directory	PATH_TO_INODE_NUMBER	↓
File name	File names	Inode numbers	A direc-tory	NAME_TO_INODE_NUMBER	machine-user interface
Inode number	Inode numbers	Inodes	The inode table	INODE_NUMBER_TO_INODE	↑
File	Index numbers	Block numbers	An inode	INDEX_TO_BLOCK_NUMBER	machine-oriented names
Block	Block numbers	Blocks	The disk drive	BLOCK_NUMBER_TO_BLOCK	↓

Summary of File System

- File name is not part of a file (neither data nor metadata!)
  - Name is not a part of an inode
  - Name is actually the data of a directory
  - An inode can have several names (hard LINK)
- Hard links are equal
  - If a file has two name mapping, both are links (instead of a link and a name)
- Directory size is small
  - Only mapping from name to inode number
  - “Folder” is somewhat misleading

FILE SYSTEM API

Implementing the File System API

- API
  - CHDIR, MKDIR
  - LINK, UNLINK, RENAME
  - SYMLINK
  - MOUNT, UNMOUNT
  - OPEN, READ, WRITE, CLOSE
  - FSYNC
- Implemented as System Call to User Apps
  - Set of function pointers
  - Specific to each FS (mount point)

File Meta-data

- Owner ID
  - User ID and group ID that own this inode
- Types of permission
  - Owner, group, other
  - Read, write, execute
- Time stamps
  - Last access (by OPEN)
  - Last modification (by WRITE)
  - Last change of inode (by LINK)

```
structure inode
    integer block_numbers[N]
    integer size
    integer type
    integer refcnt
    integer userid
    integer groupid
    integer mode
    integer atime
    integer mtime
    integer ctime
```

OPEN File

- Check user's permission
- Update last access time
- Return a short name for a file
  - *fd*: file descriptor
  - Used by READ, WRITE, CLOSE

File Descriptor

- Each process starts with three open files
  - Standard in: *fd* = 0
  - Standard out: *fd* = 1
  - Standard error: *fd* = 2
- Can also use *fd* to name opened devices
  - Keyboard, display, etc.
  - Allow a designer not to worry about input/output
    - Just read from *fd* 0 and write to *fd* 1
- Each process has its own *fd* name space

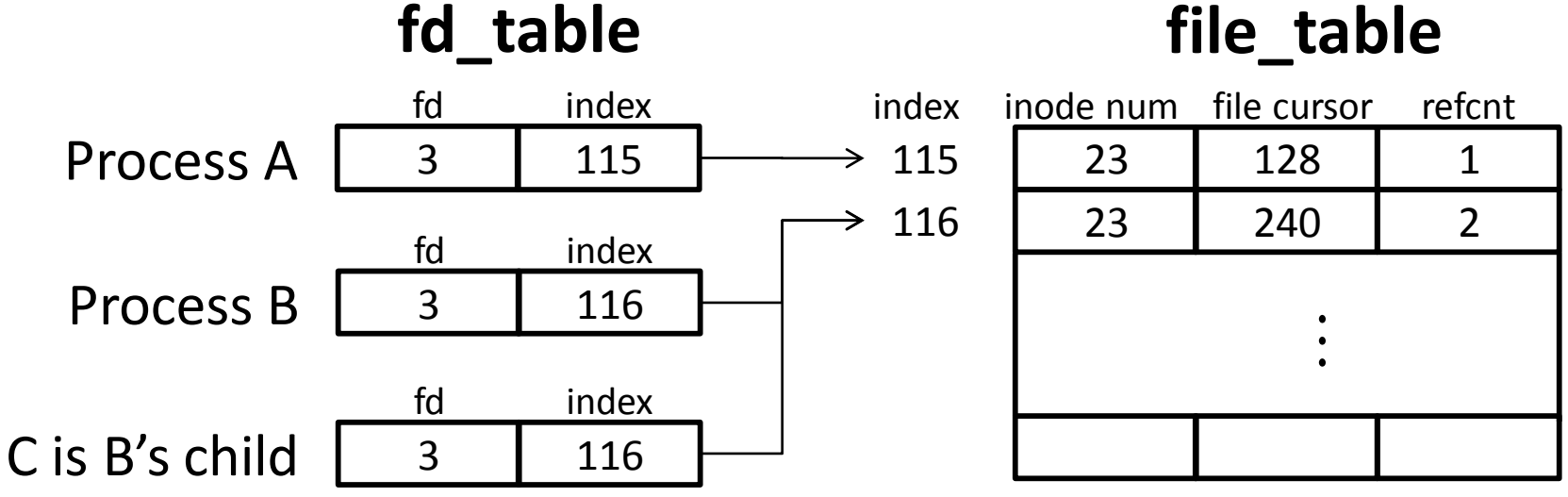
File Cursor

- File cursor
  - Keep track of operation position within a file
- Sharing cursor
  - Parent passes its *fd* to its child
    - In UNIX, child inherits all open *fds* from its parent
  - Allow parent and child to share a output file
- Not sharing cursor
  - Two processes open the same file

fd_table & file_table

- One *file_table* for the whole system
  - Records information for opened files
    - Inode number, file cursor, reference count of opening processes
    - Children can share the cursor with their parent
- One *fd_table* for each process
  - Records mapping of *fd* to index of the *file_table*

File Cursor Sharing



- Process A, B and C all open just one file with inode number 23
- Process A and B open the same file, not share file cursor
- Process B and C share the file cursor

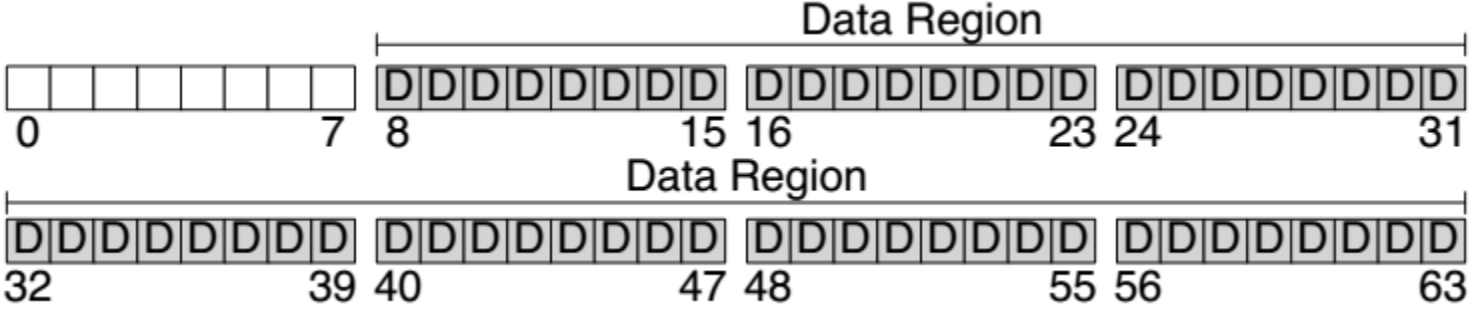
OPEN Implementation

```
1    procedure OPEN (character string filename, flags, mode)
2        inode_number ← PATH_TO_INODE_NUMBER (filename, wd)
3        if inode_number = FAILURE and flags = O_CREATE then      // Create the file?
4            inode_number ← CREATE (filename, mode)               // Yes, create it.
5        if inode_number = FAILURE then
6            return FAILURE
7        inode ← INODE_NUMBER_TO_INODE (inode_number)
8        if PERMITTED (inode, flags) then   // Does this user have the required permissions
9            file_index ← INSERT (file_table, inode_number)
10           fd ← FIND_UNUSED_ENTRY (fd_table)   // Find entry in file descriptor table
11           fd_table[fd] ← file_index           // Record file index for file descriptor
12           return fd                           // Return fd
13       else return FAILURE                     // No, return a failure
```

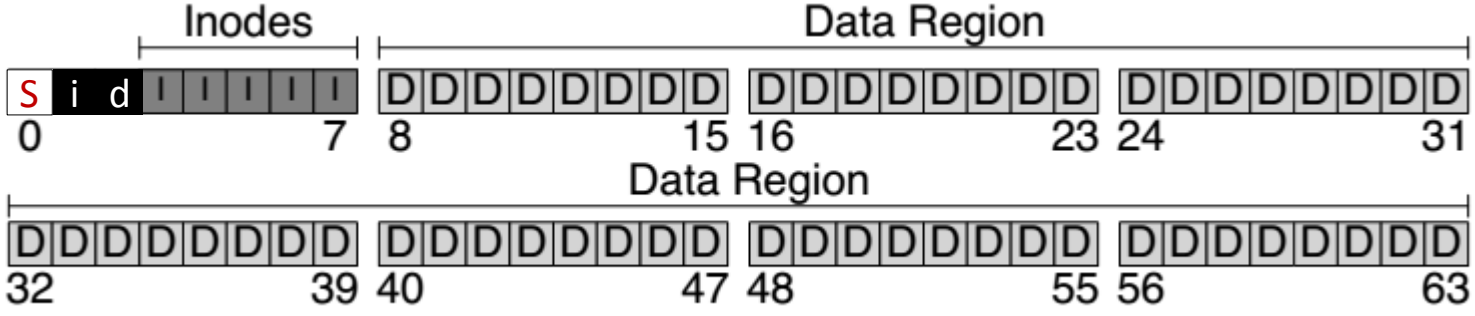
READ Implementation

```
1     procedure READ (fd, character array reference buf, n)
2         file_index ← fd_table[fd]
3         cursor ← file_table[file_index].cursor
4         inode ← INODE_NUMBER_TO_INODE (file_table[file_index].inode_number)
5         m = MINIMUM (inode.size – cursor, n)
6         atime of inode ← NOW ()
7         if m = 0 then return END_OF_FILE
8         for i from 0 to m – 1 do {
9             b ← INODE_NUMBER_TO_BLOCK (i, inode_number)
10            COPY (b, buf, MINIMUM (m – i, BLOCKSIZE))
11            i ← i + MINIMUM (m – i, BLOCKSIZE)
12        file_table[file_index].cursor ← cursor + m
13        return m
```

Disk Layout of a Simple File System



At the Head of a Disk Partition



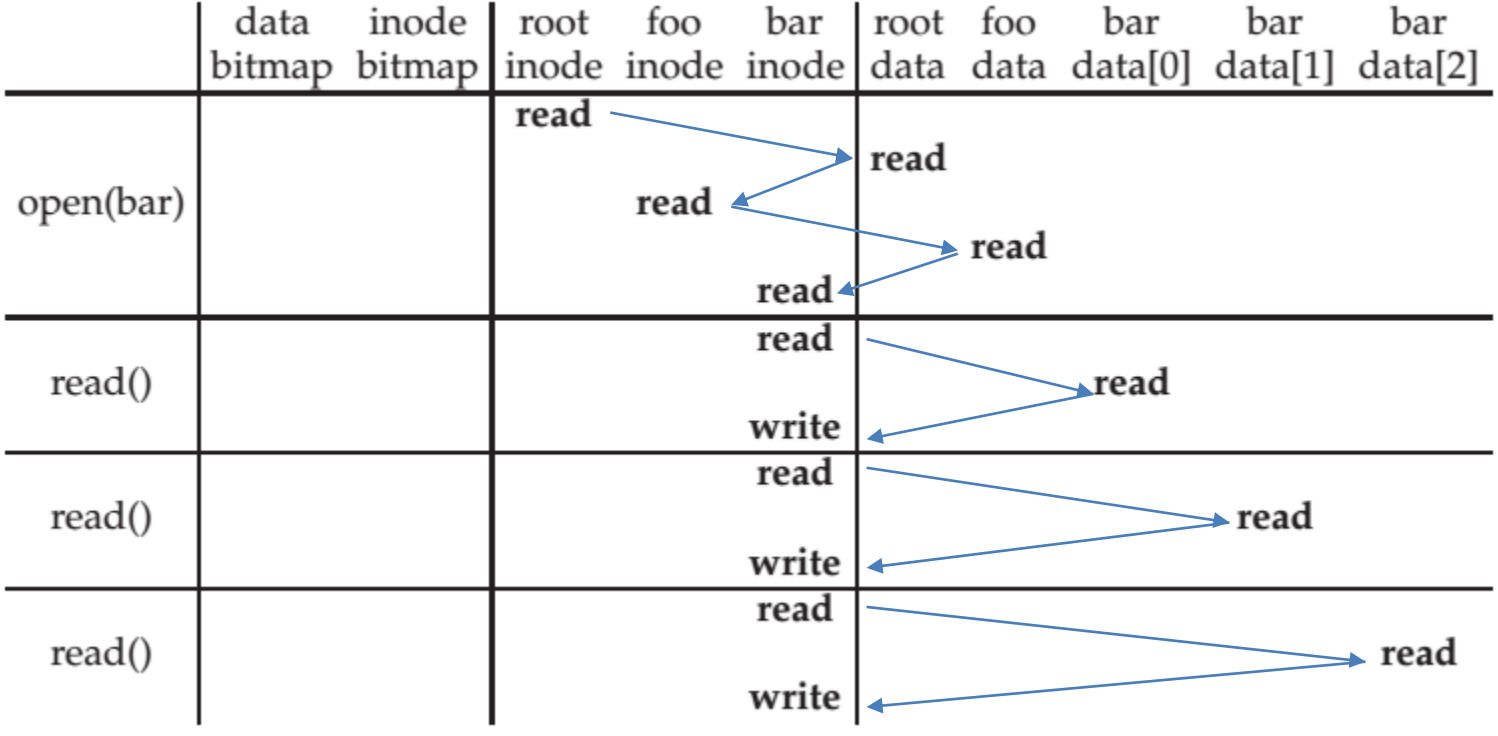
- i: inode free block bitmap
- d: data free block bitmap
- S: super-block
  - How many inodes: 80
  - How many data blocks: 56
  - Where the inode table begins: block 3
  - …
  - The magic number to identify the file system type

The super-block is used when the file system is mounted

Question

- How many read and write in a OPEN?
- How many read and write in a READ?
- How many read and write in a CREATION?

File Open & Read Timeline




```
open("/foo/bar", O_RDONLY)
```

File Creation Timeline

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)		read write	read	read	read write write	read	read write			
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read write					write

WRITE & CLOSE

- WRITE is similar to READ
  - Allocate new block if necessary
  - Update inode's *size* and *mtime*
- CLOSE
  - Free the entry in the fd_table
  - Decrease the reference counter in file table
  - Free the entry in file table if counter is 0
- Failures in the middle may cause inconsistency
  - E.g. a block is allocated from on-disk free list, but no inode records that block yet, then the block is lost

Questions

- When writing, which order is preferred?
  - Allocate new blocks, write new data, update size
  - Allocate new blocks, update size, write new data
  - Update size, allocate new blocks, write new data

Delete after OPEN but before CLOSE

- One process has a file open
- Another process removes the last name pointing to that file
  - Reference counter is now 0
- The inode isn't freed until the first process calls CLOSE

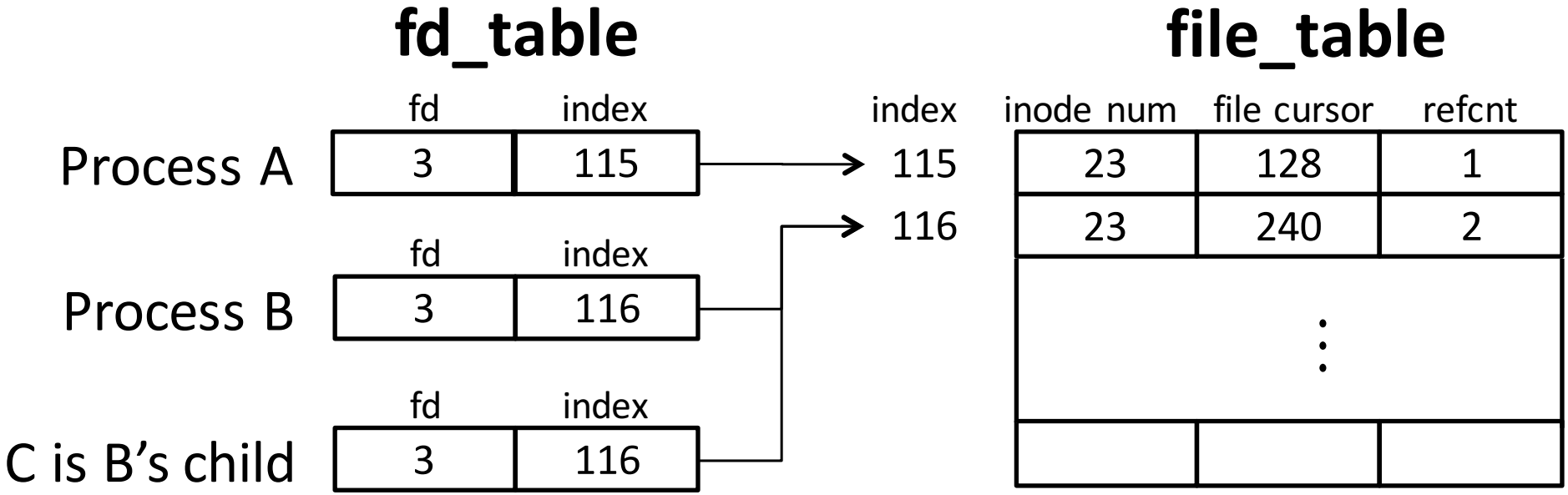
FSYNC

- Block cache
  - Cache of recently used disk blocks
  - Read from disk if cache miss
  - Delay the writes for batching
  - Improve performance
  - Problem: may cause inconsistency if fail before write
- FSYNC
  - Ensure all changes to the file have been written to the storage device

Naming in Hardware

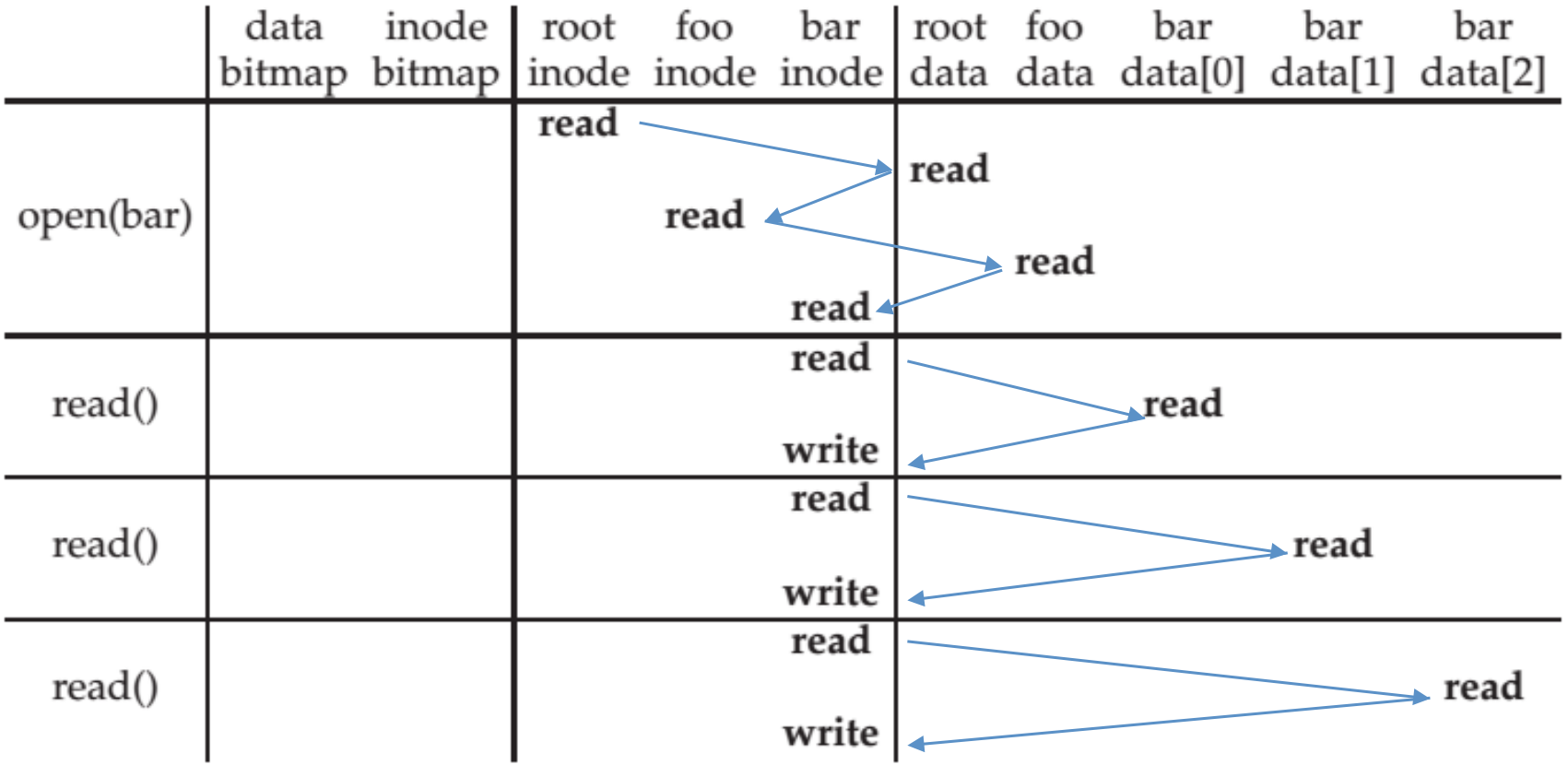
Memory bus, Interrupt and DMA

Review: File Cursor Sharing



- Process A, B and C all open just one file with inode number 23
- Process A and B open the same file, not share file cursor
- Process B and C share the file cursor

Review: File Open & Read Timeline




```
open("/foo/bar", O_RDONLY)
```

FSYNC

- Block cache
  - Cache of recently used disk blocks
  - Read from disk if cache miss
  - Delay the writes for batching
  - Improve performance
  - Problem: may cause inconsistency if fail before write
- FSYNC
  - Ensure all changes to the file have been written to the storage device

Review: Naming Scheme

- Typically, a name can be an address, an index or a string
- A value can be an address, an index, a string, a data structure, a file, a web page, a data-block, etc.
- A context can be a table, an inode, a file system, a directory, a disk, a web domain, etc.
- A lookup algorithm can be table searching, nested, multiple lookup
- Name discovery method can be well-known, enumeration, etc.

BUS: A HARDWARE LAYER

A Hardware Layer: the Bus

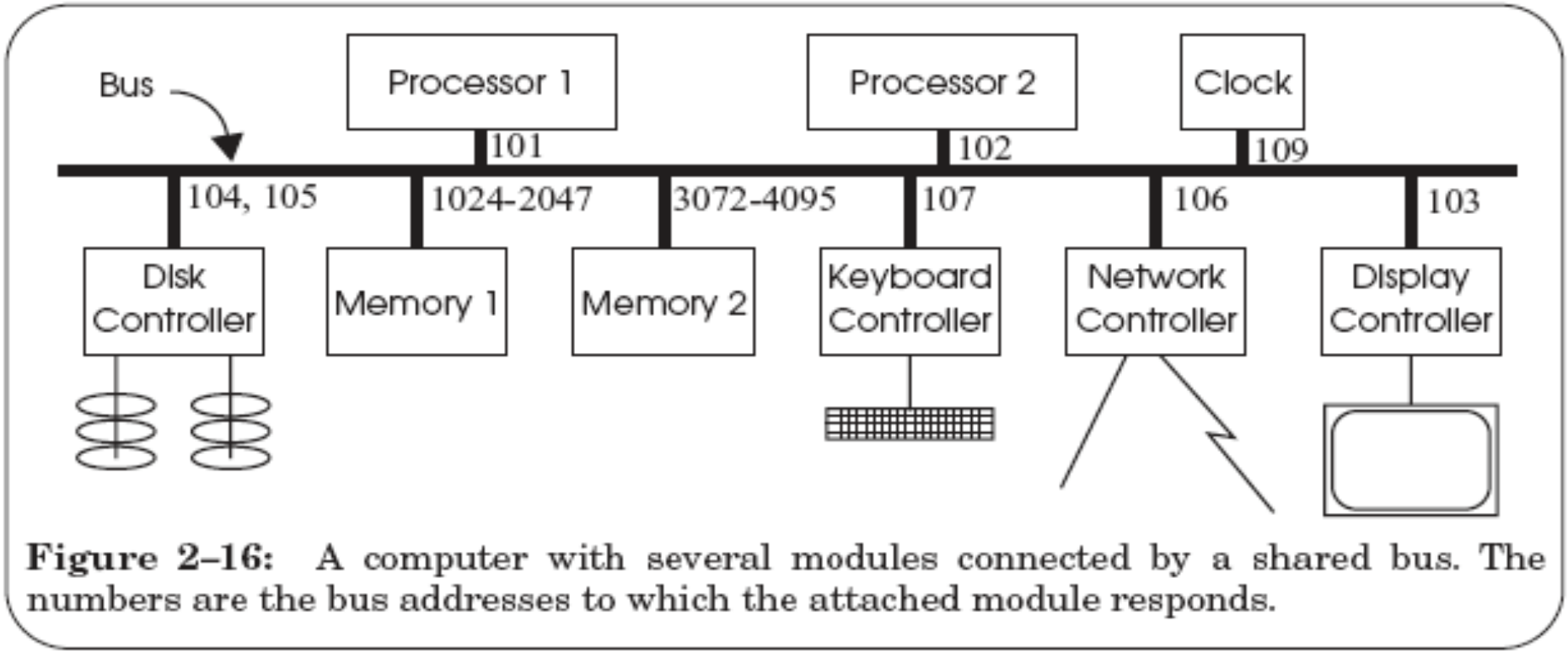


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

Bus Features

- A set of wires
  - Comprising address, data, control lines that connect to a bus interface on each module
- Broadcast link: every module hears every message
  - Bus address: identify the intended recipient, as the name
- Bus arbitration protocol
  - Decide which module may send or receive a message at any particular time
  - Bus arbiter (optional): a circuit to choose which modules can use the bus

Bus Transaction

1. Source module requires exclusive use of the bus: the data sender
2. Source module places a bus address of the destine module on the bus
3. Source module signals READY wire to alert the other module
4. The destine module singles ACKNOWLEDGE wire after copied the data
   - If synchronized, then READY & ACKNOWLEDGE are not needed, just need to check the address lines on each clock cycle
5. Source module releases the bus

Memory load example: LOAD 1742, R1

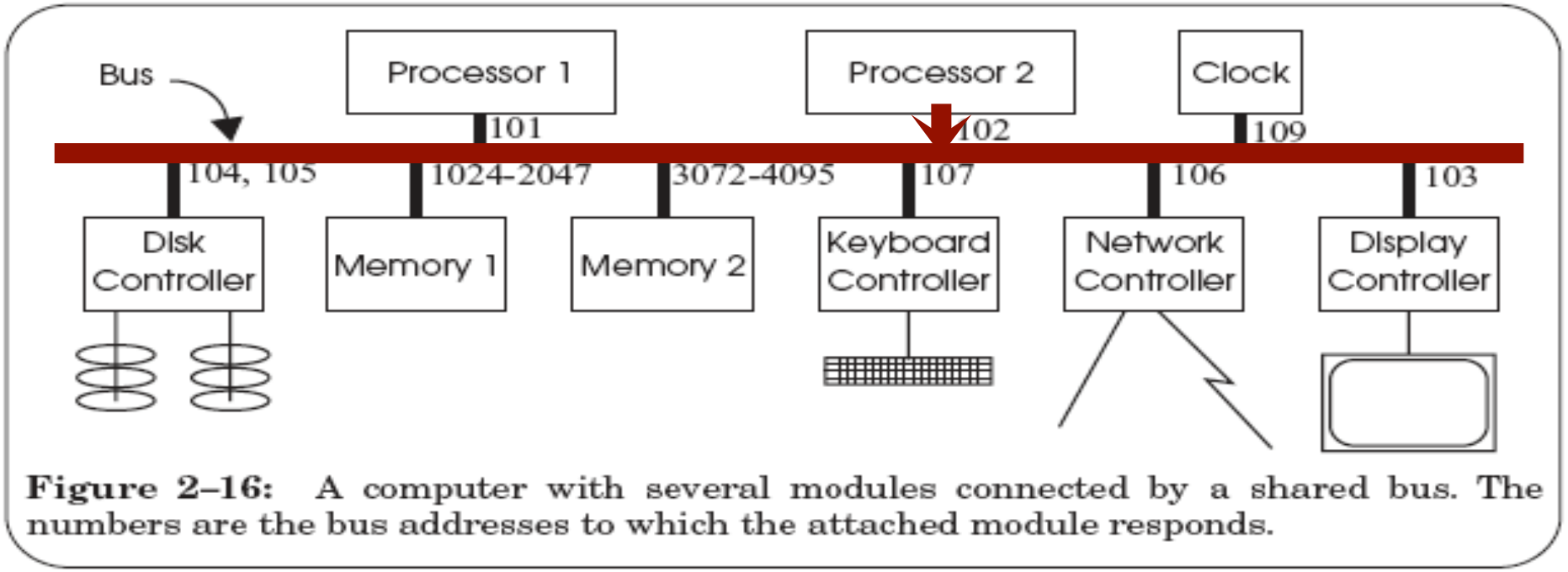


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

Processor #2 => all bus modules: {1742, READ, 102}

Memory load example: LOAD 1742, R1

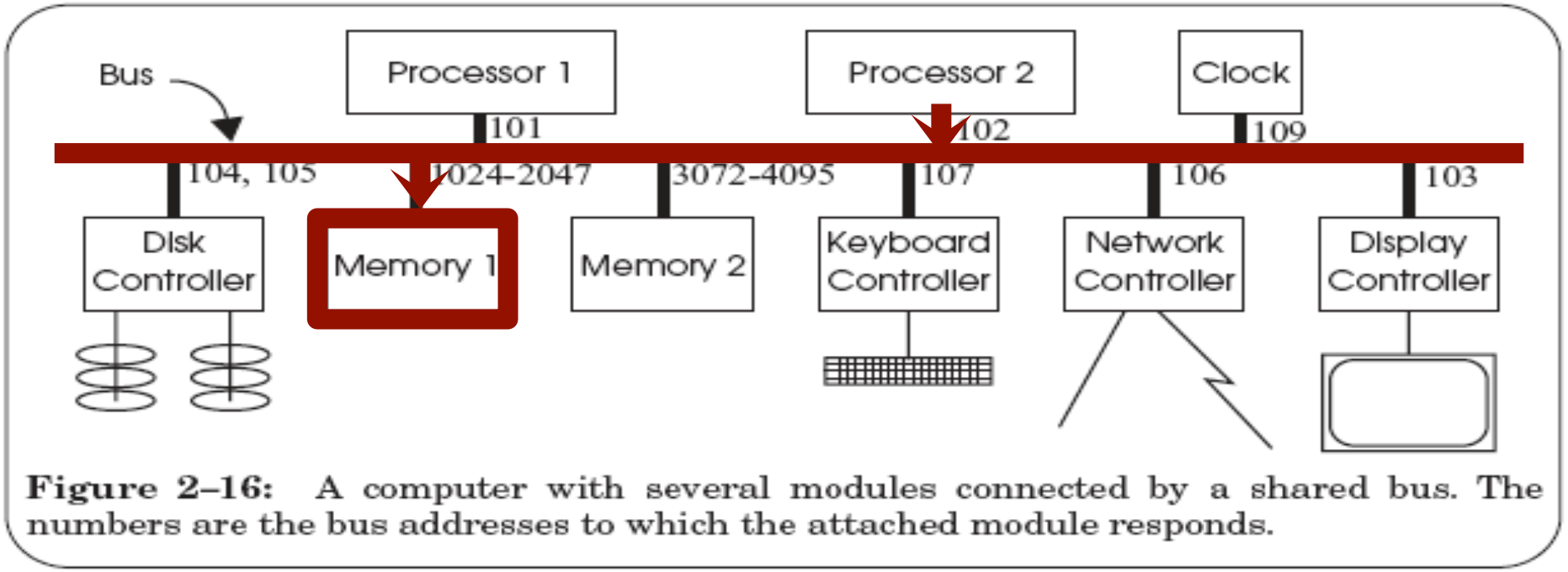


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

- Memory1 recognizes the address is within its range
  - By examining just a few high-order address bits

Memory load example: LOAD 1742, R1

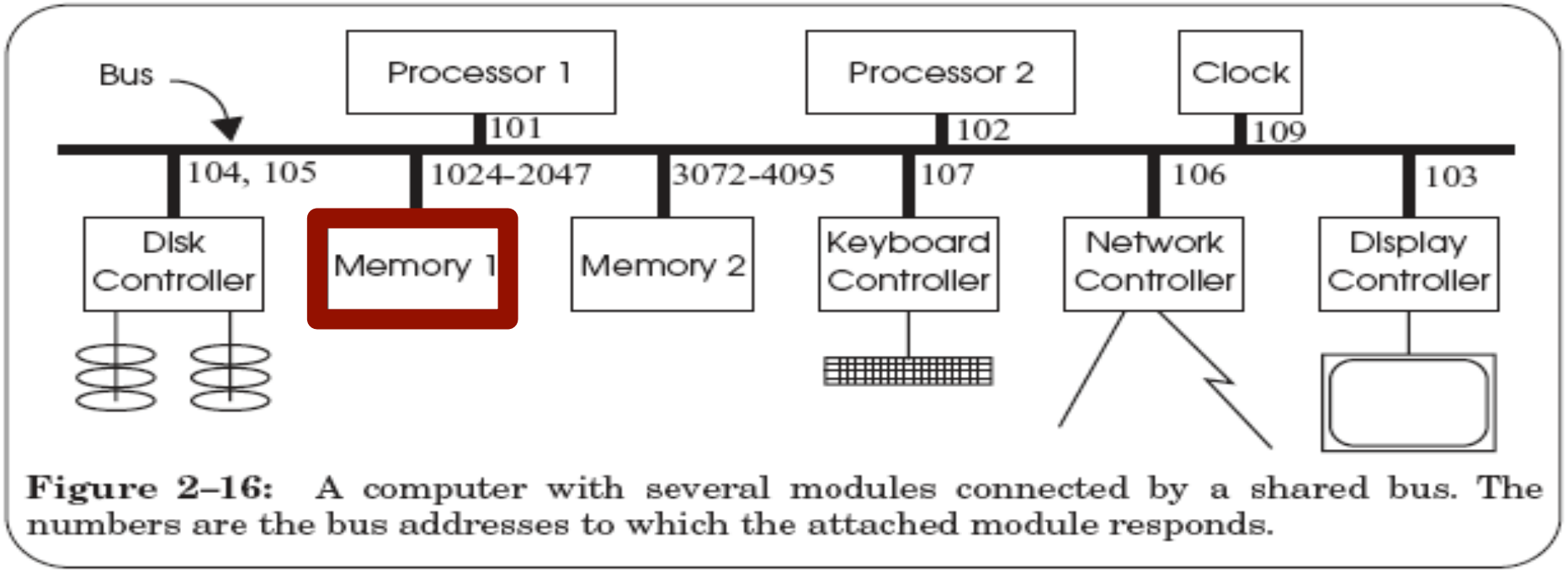


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

- Memory1 acknowledges and processor2 releases the bus
- Memory1 performs the internal operation to get the value
  - value <- READ (1742)

Memory load example: LOAD 1742, R1

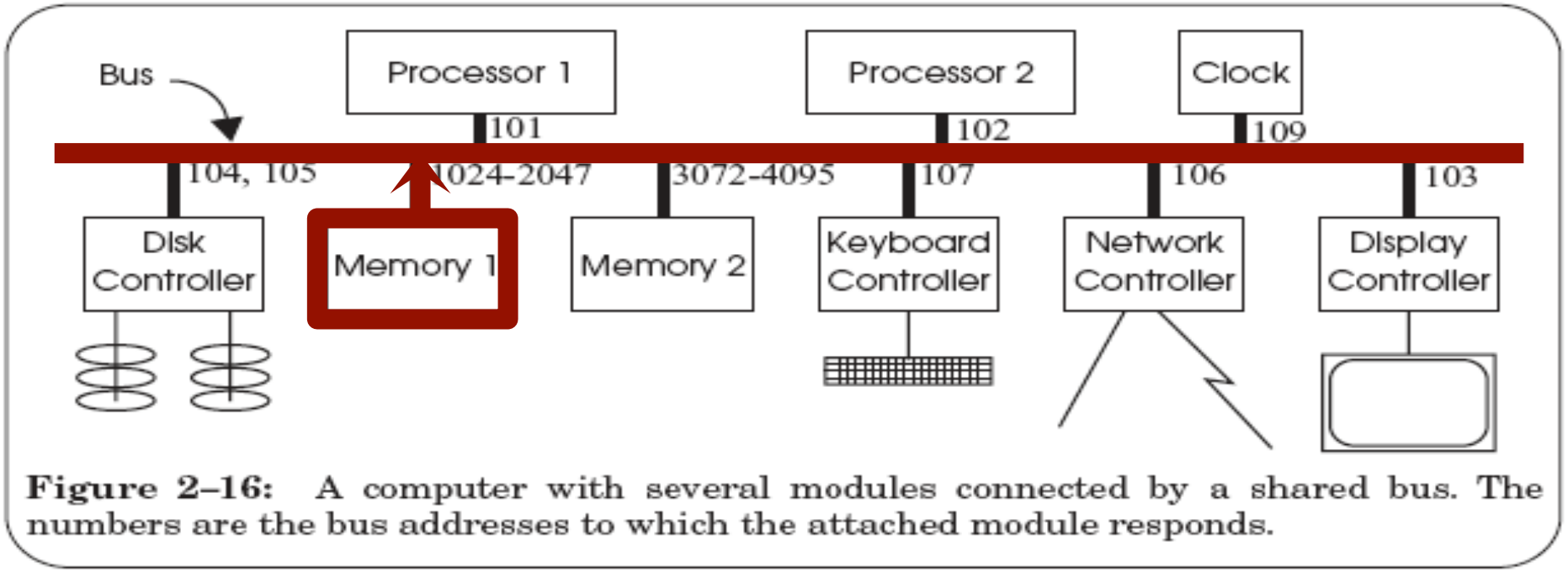


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

- Memory1 => all bus modules: {102, value}

Memory load example: LOAD 1742, R1

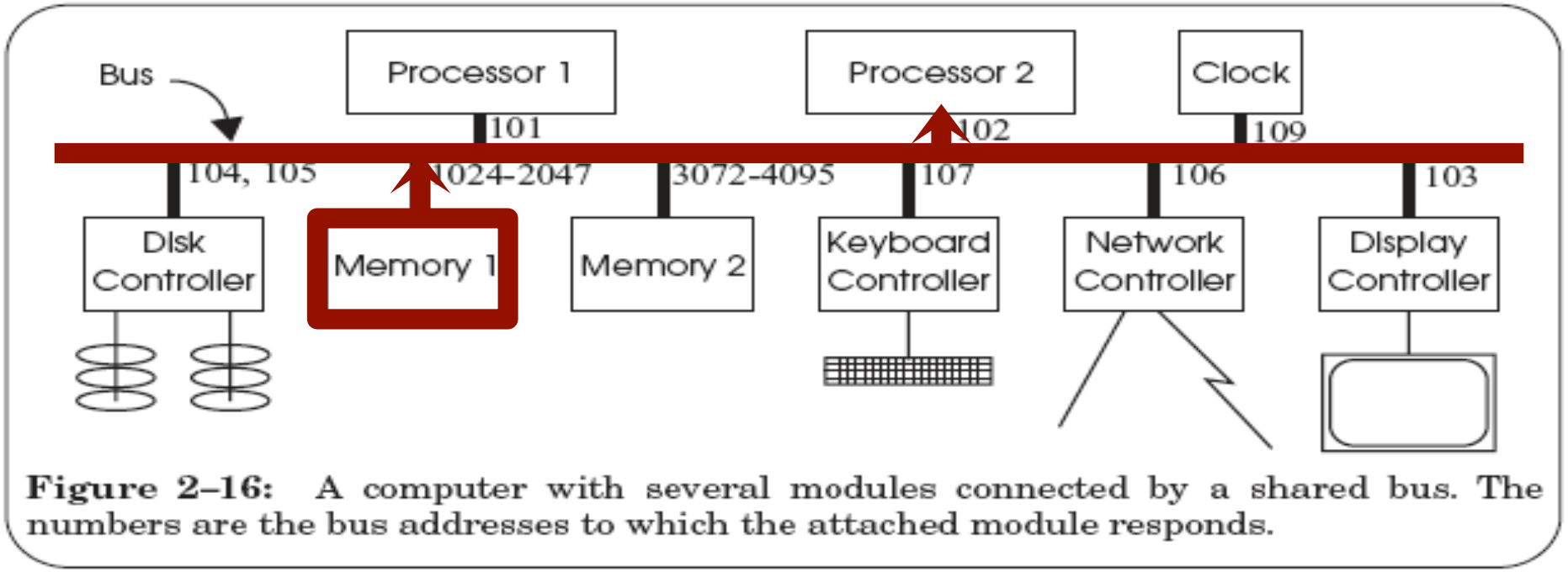


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

- Processor2 is waiting for this result, just copies the data on the bus to its register R1

Memory load example: LOAD 1742, R1

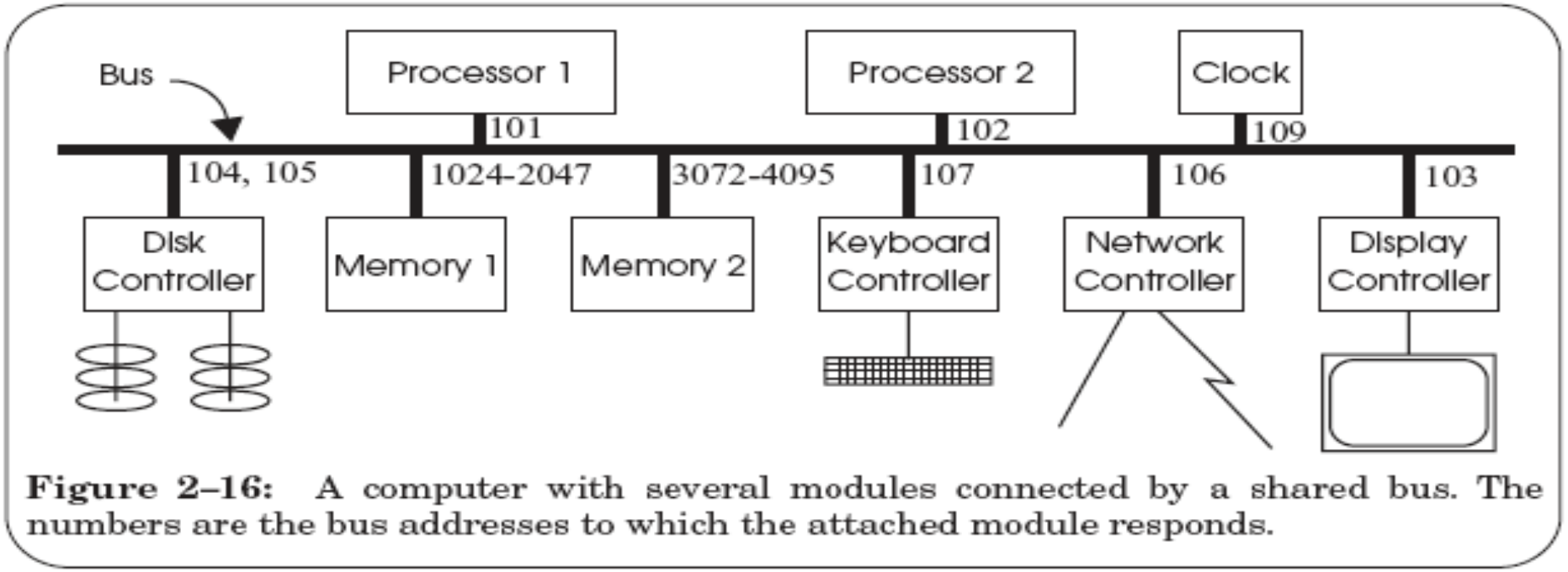


Figure 2–16: A computer with several modules connected by a shared bus. The numbers are the bus addresses to which the attached module responds.

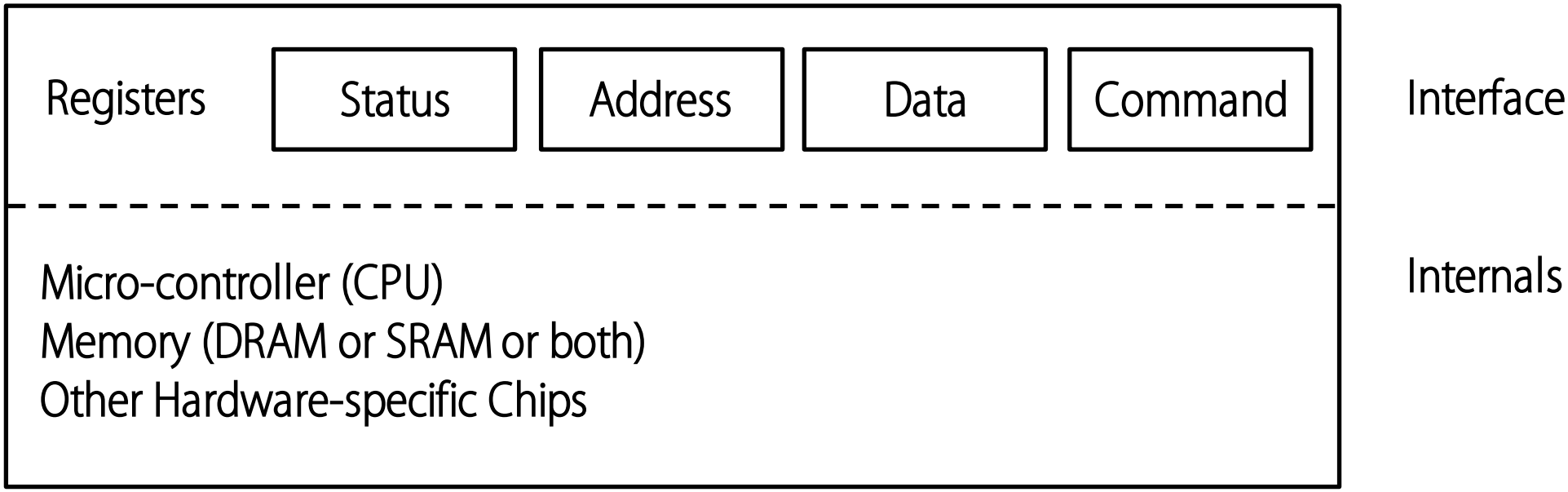
- Processor2 acknowledges and memory1 releases the bus

Sync VS. Async

- Sync data transfer
  - The source and destination cooperate through a shared clock
- Async data transfer
  - The source and destination cooperate through explicit signal line, like acknowledge line

INTERACT WITH DEVICE: INTERRUPT

A Canonical Device



A Canonical Protocol

```
While(STATUS == BUSY)
   ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
   (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

A Canonical Protocol

```
While(STATUS == BUSY)
    ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
    (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

- Polling: CPU waits until the device is ready to receive a command by repeatedly reading the status register

A Canonical Protocol

```
While(STATUS == BUSY)
   ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
   (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

- If this were a disk, then multiple writes would need to take place to transfer a disk block (say 512 Bytes) to the device
- When the main CPU is involved with the data movement (as in this example protocol), we refer to it as programmed I/O (PIO)

A Canonical Protocol

```
While(STATUS == BUSY)
    ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
    (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

- OS writes a command to the command register; doing so implicitly lets the device know that both the data is present and that it should begin working on the command

A Canonical Protocol

```
While(STATUS == BUSY)
   ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
   (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

- OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished
- It may then get an error code to indicate success or failure

A Canonical Protocol

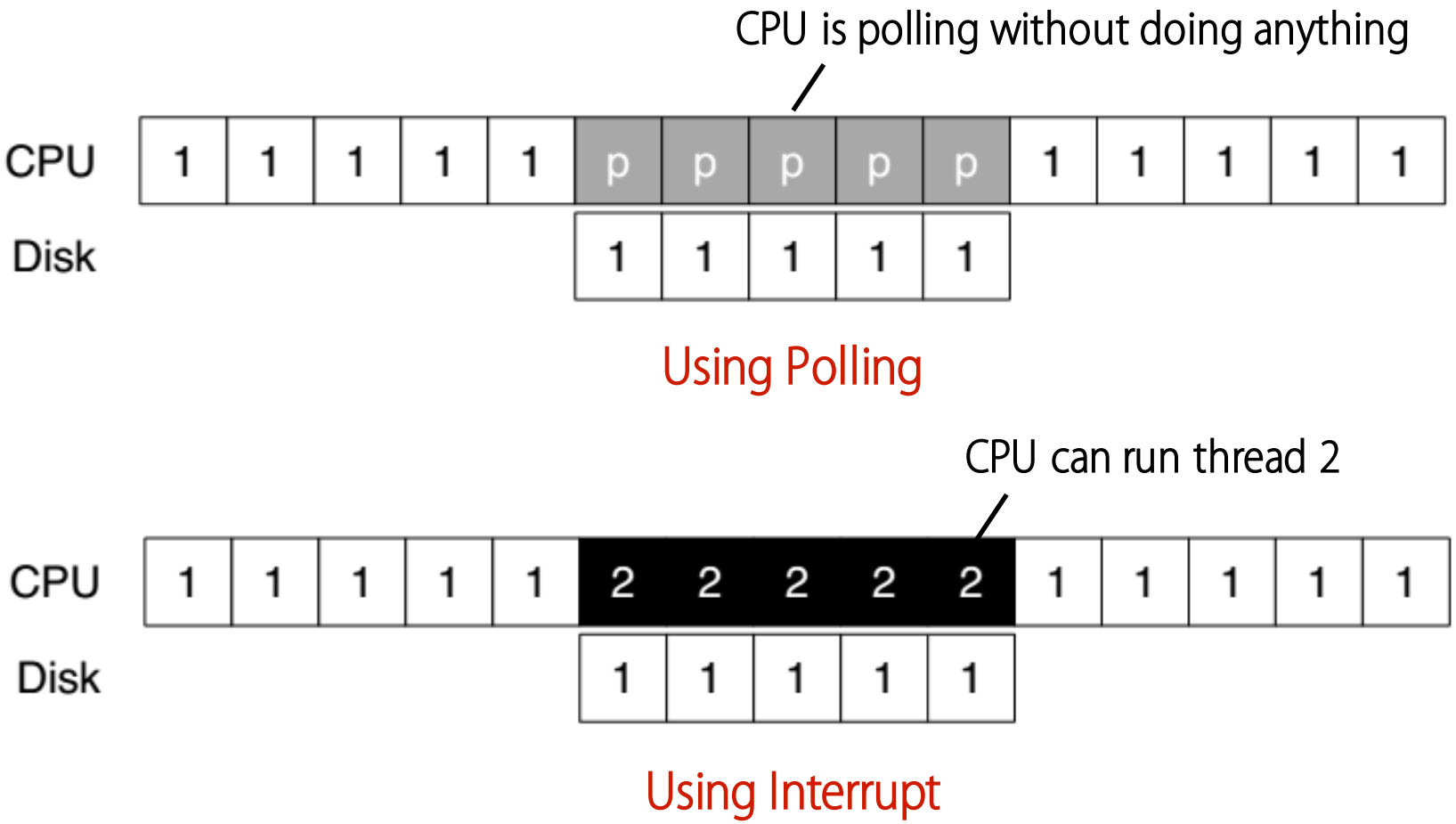
```
While(STATUS == BUSY)
   ; //wait until device is not busy
Write data to DATA register and address to ADDRESS register
Write command to COMMAND register
   (Doing so starts the device and executes the command )
While(STATUS == BUSY)
    ; //wait until device is done with your request
```

- Problem: polling wastes too much CPU
- Solution: using interrupt

Lowering CPU Overhead With Interrupts

- Instead of polling, the OS can issue a request, put the calling process to sleep, and context switch to another task
- When the device finishes, it will raise a hardware interrupt
- The CPU jumps into the OS at a pre-determined interrupt service routine (ISR) or more simply an interrupt handler
- The handler is just a piece of operating system code that will finish the request
  - For example, by reading data and perhaps an error code from the device
  - and wake the process waiting for the I/O, which can then proceed as desired

Lowering CPU Overhead With Interrupts



Example of Interrupt: Keyboard

- When user depresses a key, keyboard SENDs a message to the processor containing the key value
- As the processor is not ready, its bus interface:
  - copies the data into a temporary register,
  - acknowledges the keyboard,
  - SENDs an interrupt signal to the processor
- The processor handles the interrupt in next cycle
  - SENDs the value over the bus to memory module
- Suitable for slow device, not suitable for disk

Problem of Interrupt: Livelock

- Using interrupts arises in networks
  - When a huge stream of incoming packets each generate an interrupt it is possible for the OS to **livelock**
  - Livelock: the CPU only processes interrupts and never allows a user-level process to run and actually service the requests
- Solution: hybrid
  - Default using interrupts
  - When an interrupt happens, handle it and polling for a while to solve subsequence requests
  - If no further request or time-out, fall back to interrupt again
  - Used in Linux network driver with the name *NAPI* (New API)

Another Optimization for Interrupt

- A device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU
- While waiting, other requests may soon complete, and thus multiple interrupts can be **coalesced** into a single interrupt delivery, thus lowering the overhead of interrupt processing

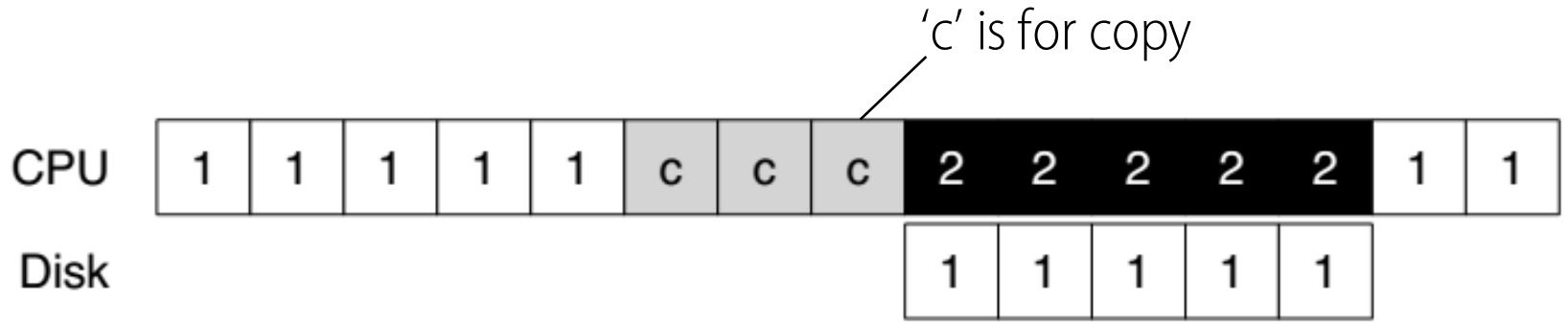
- Note: waiting too long will increase the latency of a request, a common trade-off in systems

INTERACT WITH DEVICE: DMA

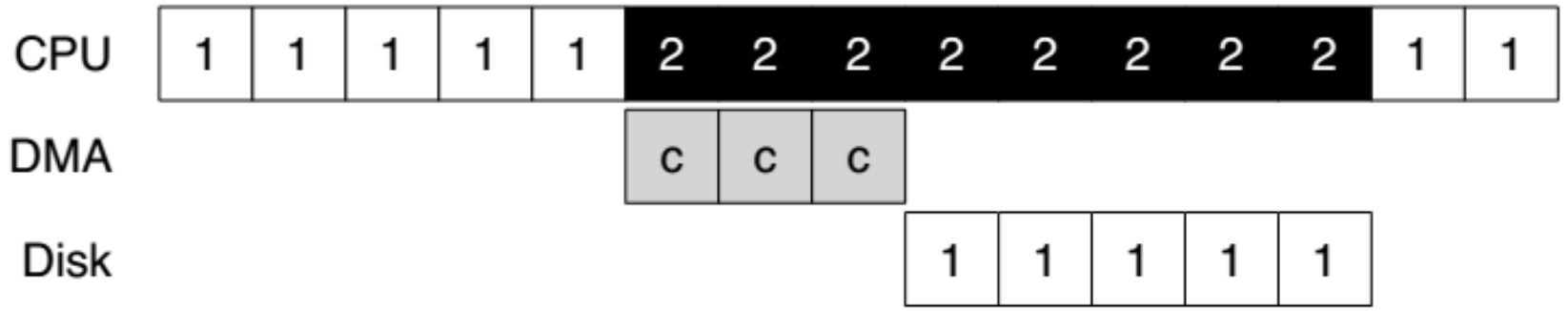
DMA for Disk Device

- DMA (Direct Memory Access)
  - A processor SENDs a request to a disk controller to READ a block of data
  - Including the address of a buffer in memory
- The disk SENDs the data directly to memory
  - Incrementing the memory address appropriately

DMA for Disk Device



Without DMA



With DMA

DMA for Disk Device

- Benefits of DMA
  - Relieve the CPU's load to execute other program
  - Reduce one transfer (original two)
  - Take better advantage of long message if the bus supports
  - Amortize the overhead of the bus protocol

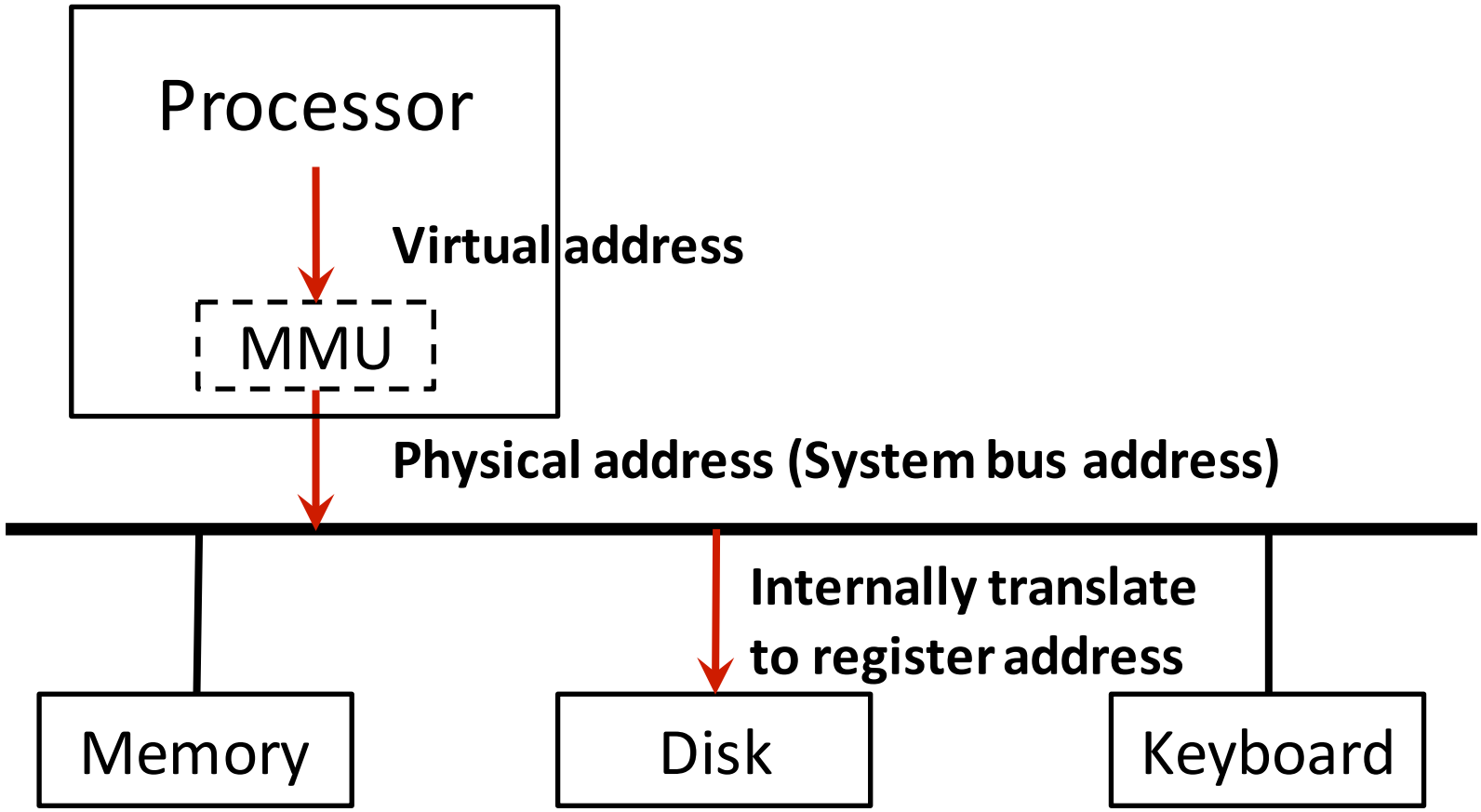
Methods of Device Interaction

- How should the hardware communicate with a device? Should there be explicit instructions? Or are there other ways to do it?
- Two primary methods
  - I/O instructions: on x86, IN and OUT instructions
    - Must be executed in privileged mode (kernel mode)
  - Memory-mapped I/O: using LOAD and STORE
    - Can also be executed in unprivileged mode (user mode)

Memory Mapped I/O

- Use LOAD and STORE instructions to address the register and buffer of the I/O modules
  - Just like access memory
  - Address is overloaded name with location info
- Provide a uniform interface to bus modules
  - MMU translates virtual address to physical address
    - Physical address is system bus address
  - I/O modules translate bus address to register address internally

Memory Mapped I/O



NOTE: Volatile Address if using MMIO

```
void main(void)
{
    void          *pdev = (void *) 0x40400000;
    size_t         size = (1024*1024);
    int           *base;
    volatile int  *pcid, cid;

    base = mmap(pdev, size, PROT_READ|PROT_WRITE,
                MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
    if (base == MAP_FAILED) errx(1, "mmap failure");

    pcid = (int *) (((void *) base) + 0xf0704);
    cid = *pcid;
    printf("cid = %d\n", cid);
    cid = *pcid;
    printf("cid = %d\n", cid);

    munmap(base, size);
}
```

If no volatile, then the compiler will think the two printf are redundant and will eliminate the second memory load operation

Another Volatile Example

```
#include <stdio.h>
void main()
{
    int i = 10;
    int a = i;

    printf("i= %d\n",a);

     // Change value of i
      __asm {
          mov dword ptr [ebp-4], 20h
    }

    int b = i;
    printf("i= %d\n",b);
}
```

Is volatile needed here?

Memory Mapped I/O Combined with DMA

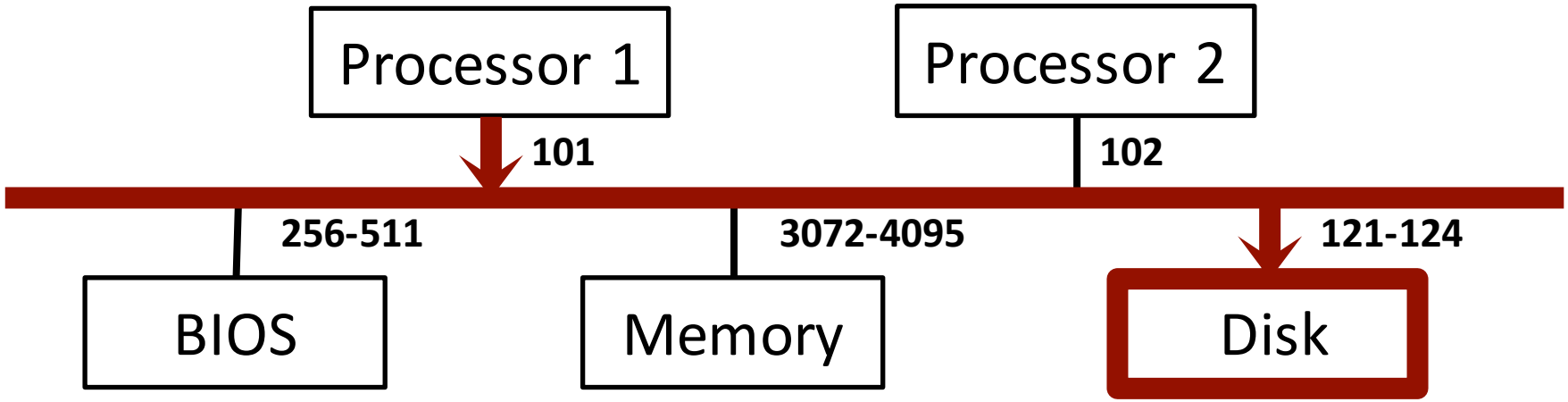
bus address	control register
121	*sector_number*
122	*DMA_start_address*
123	*DMA_count*
124	*control*

```
R1 ← 11742; R2 ← 3328; R3 ← 256; R4 ← 1;
STORE 121,R1          // set sector number
STORE 122,R2          // set memory address register
STORE 123,R3          // set byte count
STORE 124,R4          // start disk controller running
```

disk controller #1 ⇒ *all bus modules*: {3328, *block*[1]}
disk controller #1 ⇒ *all bus modules*: {3336, *block*[2]}
etc . . .

DMA Example

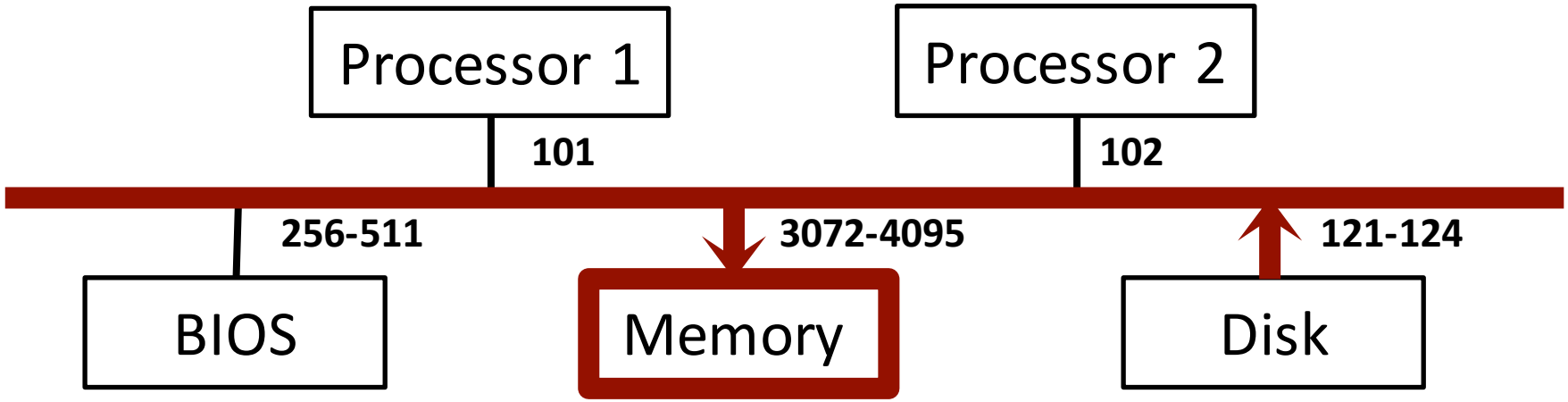
bus address	control register
121	*sector_number*
122	*DMA_start_address*
123	*DMA_count*
124	*control*



- Processor #1 => all bus modules: {121, WRITE, 11742}
  - Disk acknowledge and write the value 11742 to its control register
- Processor #1 => all bus modules: {122, WRITE, 3328}
- Processor #1 => all bus modules: {123, WRITE, 256}
- Processor #1 => all bus modules: {124, WRITE, 1}

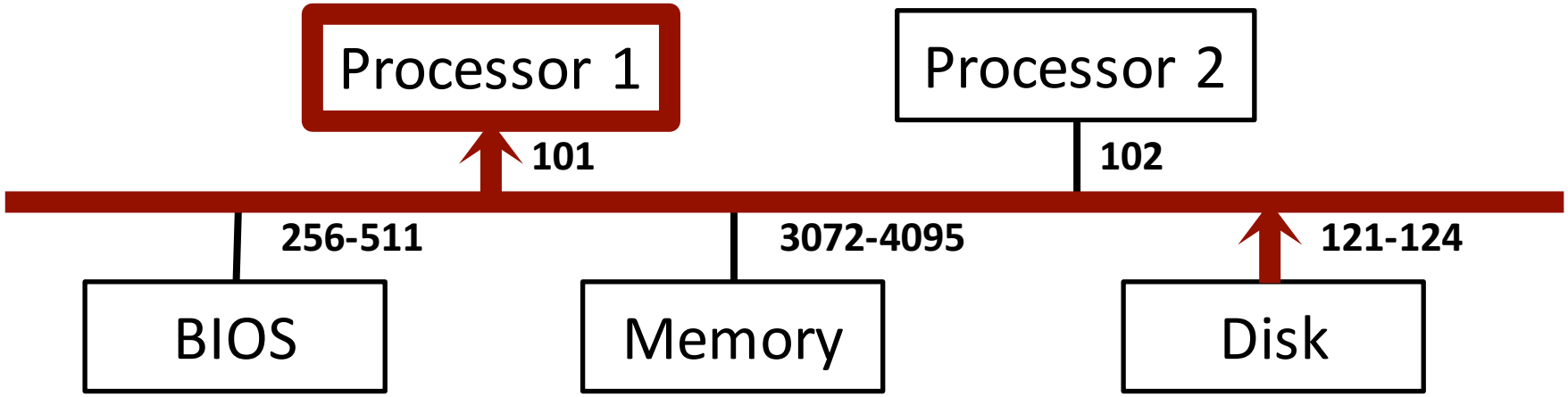
DMA Example

```
R1 ← 11742; R2 ← 3328; R3 ← 256; R4 ← 1;
STORE 121,R1                    // set sector number
STORE 122,R2                    // set memory address register
STORE 123,R3                    // set byte count
STORE 124,R4                    // start disk controller running
```



- Disk => all bus modules: {3328, WRITE, *data[11742]*}
  - Memory acknowledge and save data[11742]
- Disk => all bus modules: {3329, WRITE, *data[11743]*}
- ... *(loop)*
- Disk => all bus modules: {3583, WRITE, *data[11997]*}

DMA Example



- When transferring is finished, disk controller SENDs message to the processor
  - Just like keyboard controller does when press a key
- Processor will enter interrupt handler next cycle
- Now the processor knows that the DMA is done

Summary

- How does a CPU interact with physical memory?
  - Through system bus that connects each other
  - Using physical address to name memory content
- How does a CPU interact with a device?
  - Also using physical address
  - Polling, interrupt and DMA
  - I/O instruction or MMIO

Questions

- How are the physical addresses assigned?
  - Memory physical addresses: by BIOS
  - Some devices (e.g., keyboard, IDE): fixed for all time
  - Other devices: assigned by the OS

Questions

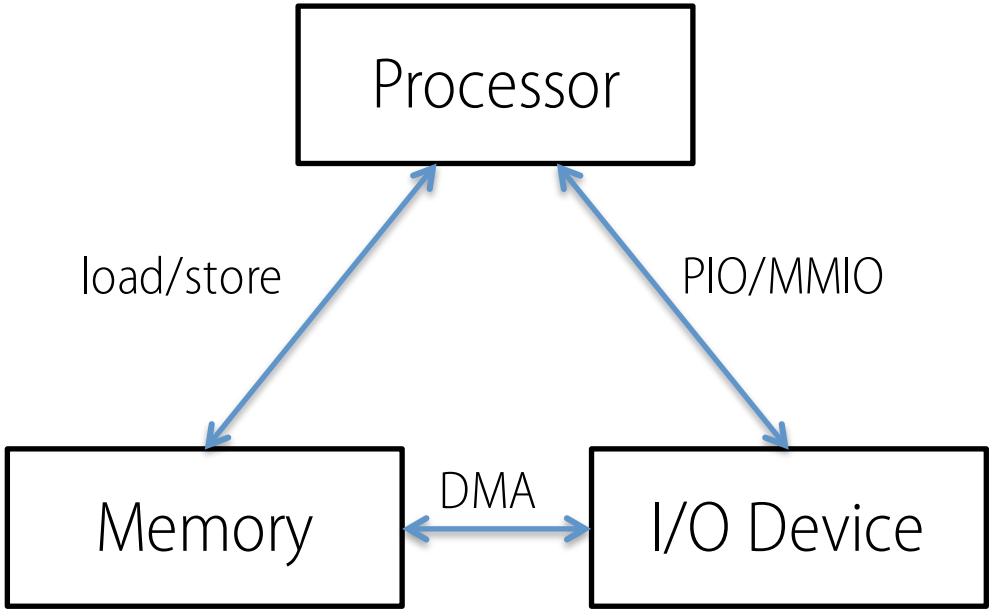
- Why not map the whole disk to memory?
  - So that the CPU can access a byte on the disk directly by system bus
  - 1. Too large
  - 2. Too slow

The principle of least astonishment:

People are part of the system. The design should match the user's experience, expectations, and mental models

Summary

- Memory Load/Store
  - Between CPU and memory
  - Physical memory address space
- I/O Operations
  - MMIO: map device memory and registers into physical address space
  - E.g., frame buffer
- DMA
  - Also using physical address



- Name of a disk
  - File name: /dev/sda1
    - As a special type of inode
    - 8,0 as (major, minor)
  - PCI address: 19:00.0
    - SCSI storage controller: LSI Logic / Symbios Logic SAS1068E PCI-Express Fusion-MPT SAS (rev 08)

Client & Server

Enforced modularity

Why Enforced Modularity?

- Modularity itself is not enough
  - Programmers make mistakes
  - Mistakes propagate easily
  - A way to strengthen the modularity is needed

Modularity in the Code

```
1    procedure MEASURE (func)
2        start ← GET_TIME (SECONDS)
3        func () // invoke the function
4        end ← GET_TIME (SECONDS)
5        return end – start
```

```
1    procedure GET_TIME (units)
2        time ← CLOCK
3        time ← CONVERT_TO_UNITS (time, units)
4        return time
```

- Modularity in the code
  - E.g. hide CLOCK's physical address
  - E.g. hide time unit
  - No need to change MEASURE on another computer, only change GET_TIME
  - … but not enough

Soft Modularity

- Error leaks from one module to another
- Function call
  - Stack discipline
  - Procedure calling convention

Potential Problems in Calling - 1

- Errors in callee may corrupt the caller's stack
  - The caller may later compute incorrect results or fails
- Callee may return somewhere else by mistake
  - The caller may lose control completely and fail
- Callee may not store return value in R0
  - The caller may read error value and compute incorrectly
- Caller may not save temp registers before the call
  - The callee may change the registers and causes the caller compute incorrectly

Potential Problems in Calling - 2

- Callee may have disasters (e.g. divided by 0)
  - Caller may terminate too, known as fate sharing
- Callee may change some global variable that it shouldn't change
  - Caller and callee may compute incorrectly or fail altogether
  - Even other procedures may be affected

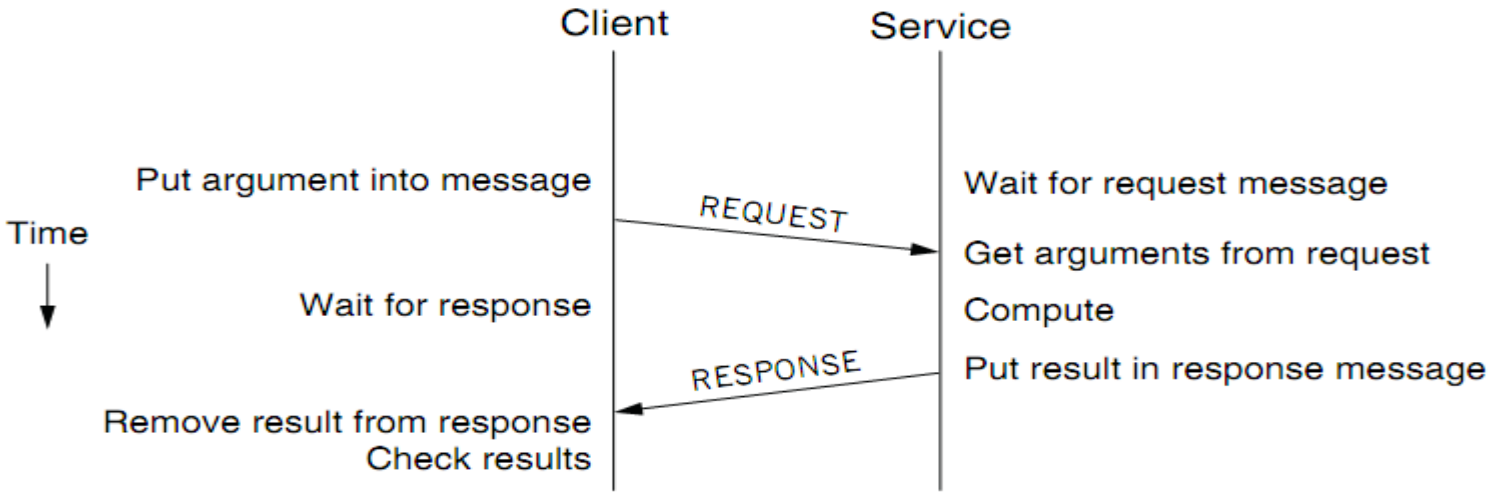
- Procedure call contract provides only soft modularity
  - Attained through specifications
  - No way to force the interactions among modules to their defined interfaces

Enforced Modularity is Needed

- Using external mechanism
  - Limits the interaction among modules
  - Reduces the opportunities for propagation of errors
  - Allows verification that a user uses a module correctly
  - Prevent an attacker from penetrating the security of a module

Client/Service Organization

- Limit interactions to explicit messages
  - Client: request; service: response or reply
  - On different computers, connected by a wire
    - Chap 5 will explain how to get them into one computer



C/S Model

- Client/Service model
  - Separates functions (abstraction)
  - Enforces that separation (enforced modularity)
  - Reduce fate sharing but not eliminate it

Multiple Clients and Services

- One services can work for multiple clients
- One client can use several services
- One module can be both a client and a service
  - Printer as a service for printing request
  - Also a client of the file service

Trusted Intermediaries

- Service as trusted 3rd party
  - Run critical procedures as a service
  - E.g., a file service to keep clients' file distinct, email service
  - Enforces modularity among multiple clients
  - Ensures that a fault in one client has limited effect on another client

- Thin client computing (the ultimate version of C/S?)
  - Simplify clients by having the trusted intermediary provide most functions
  - Only the trusted intermediaries run on powerful computers
  - What if the powerful computers fail…

Remote Procedure Call

RPC

RPC

- RPC（Remote Procedure Call)
  - Allow a procedure to execute in another address space without coding the details for the remote interaction
- RPC History
  - Idea goes back in 1976
  - Sun's RPC: first popular implementation on Unix
    - Used as the basis for NFS
- RMI (Remote Method Invocation)
  - Object-oriented version of RPC, e.g. in Java

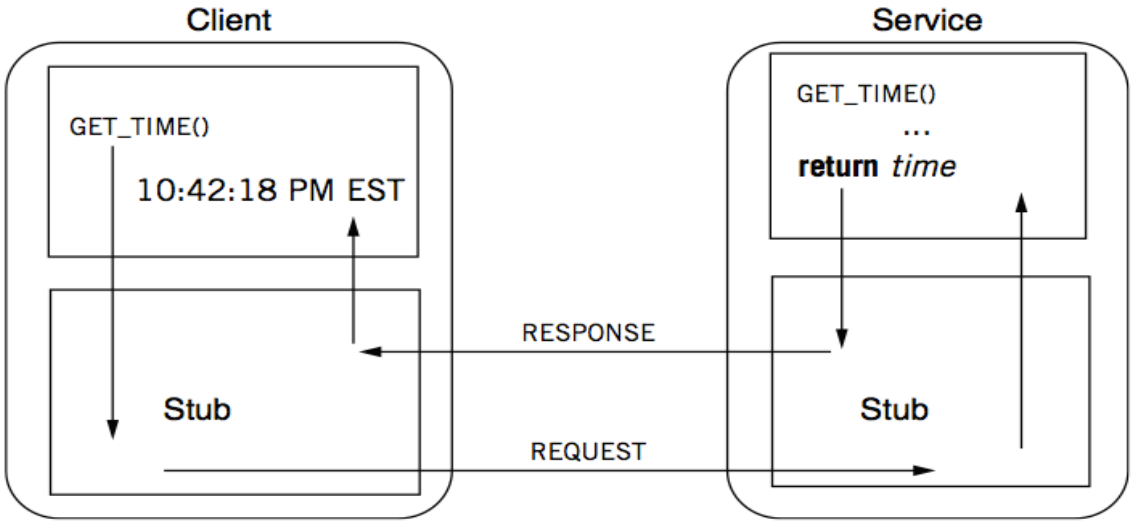
Back to the Example

```
1    procedure MEASURE (func)
2        start ← GET_TIME (SECONDS)
3        func () // invoke the function
4        end ← GET_TIME (SECONDS)
5        return end – start
```

```
1    procedure GET_TIME (units)
2        time ← CLOCK
3        time ← CONVERT_TO_UNITS (time, units)
4        return time
```

- Implement MEASURE and GET_TIME with RPC
  - On different computers
  - Marshaling: big-endian & little-endian converting

RPC Calling Process



- Message send/receive
  - SEND_MESSAGE (NameForTimeService, {"Get time", unit})
  - RECEIVE_MESSAGE (NameForTimeService)

RPC Stub

- Client stub
  - Marshal the arguments into a message
  - Send the message
  - Wait for a response
- Service stub
  - Wait for a message
  - Unmarshal the argument
  - Call the procedure (e.g. GET_TIME)
- Stub
  - Hide marshaling and communication details

Client Program

Client program

```
1       procedure MEASURE (func)
2           SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
3           response ← RECEIVE_MESSAGE (NameForClient)
4           start ← CONVERT2INTERNAL (response)
5           func ()       // invoke the function
6           SEND_MESSAGE (NameForTimeService, {"Get time", CONVERT2EXTERNAL(SECONDS)})
7           response ← RECEIVE_MESSAGE (NameForClient)
8           end ← CONVERT2INTERNAL (response)
9           return end – start
```

Service Program

```
10      procedure TIME_SERVICE ()
11          do forever
12              request ← RECEIVE_MESSAGE (NameForTimeService)
13              opcode ← GET_OPCODE (request)
14              unit ← CONVERT2INTERNAL(GET_ARGUMENT (request))
15              if opcode = "Get time" and (unit = SECONDS or unit = MINUTES) then
16                  time ← CONVERT_TO_UNITS (CLOCK, unit)
17                  response ← {"OK", CONVERT2EXTERNAL (time)}
18              else
19                  response ←{"Bad request"}
20              SEND_MESSAGE (NameForClient, response)
```

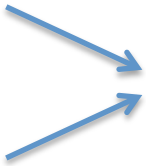
Question

- What is inside a *message*?
  - Service ID (e.g., function ID)
  - Service parameter (e.g., function parameter)
  - Using marshal / unmarshal

RPC Request

- UDP header...
  - Xid
  - call/reply
  - rpc version
  - program #
  - program version
  - procedure #
  - auth stuff
  - arguments

X is short for "transaction"
Client reply dispatch uses xid
Client remembers the xid of each call



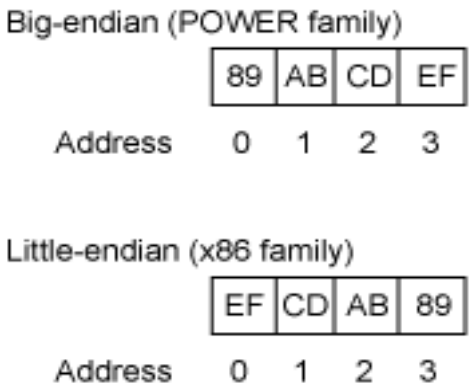
Server dispatch uses prog#, proc#

RPC Reply

- UDP header …
  - Xid
  - call/reply
  - accepted? (vs bad rpc version, or auth failure)
  - auth stuff
  - success? (vs bad prog/proc #)
  - results

Marshal / Unmarshal

- Marshal
  - Convert an object into an array of bytes with enough annotation so that the unmarshall procedure can convert it back into an object
- Why Marshal?
  - Serialization is not easy
  - E.g., big endian VS. little endian
    - 0x89ABCDEF on POWER and Intel
    - Big endian on network:
      - htons(), ntohs()
  - What about an array?
  - What about a pointer?



Automatic Stub Generation

- Automatic generation
  - Generate stubs from an interface specification
- Process
  - Tool to look at argument and return types
  - Generate *marshal* and *unmarshal* code
  - Generate stub procedures
  - Saves programming (thus less error)
  - Ensures agreement on e.g. argument types
    - E.g., consistent function ID

RPC System Components

- 1. Standards for wire format of RPC msgs and data types
- 2. Library of routines to marshal / unmarshal data
- 3. Stub generator, or RPC compiler, to produce "stubs"
  - For client: marshal arguments, call, wait, unmarshal reply
  - For server: unmarshal arguments, call real fn, marshal reply
- 4. Server framework:
  - Dispatch each call message to correct server stub
- 5. Client framework:
  - Give each reply to correct waiting thread / callback
- 6. Binding: how does client find the right server?

Client Framework

- Keeps track of outstanding requests
  - For each, *xid* and caller's thread / callback
- Matches replies to caller
- Might be multiple callers using one socket
  - NFS client in kernel
- Usually handles timing out and retransmitting

Server Framework

- In a threaded system:
  - Create a new thread per request
    - Master thread reads socket[s]
  - Or a fixed pool of threads
    - Use a queue if too many requests
    - E.g., NFS server
  - Or just one thread, serial execution
    - Simplifies concurrency
    - X server

Concurrent RPC

- Key feature: support for concurrent RPCs
  - If one RPC takes multiple blocking steps, can the server serve another one in the meantime?
  - For example, DNS service routine is an RPC client
  - May also avoid deadlock if one sends RPC to itself

RPC != PC

- Different in three ways
  - RPCs can reduce fate sharing
    - Won't crash
    - Set a time-out for message lost or service failure
  - RPCs introduce new failures: no response
  - RPCs take more time
    - Stub itself may cost more than a service
    - Consider using RPC in device driver: time-out error

RPC: Failure Handling

- 1. At least once
  - Client keeps resending until server responds
  - Only OK without side effects
- 2. At most once
  - Client keeps resending (may time-out)
  - Server remembers requests and suppress duplicates
- 3. Exactly once
  - This is often what we really want
  - Hard to do in a useful way, later in Chap-8 and 10
- Most RPC system do #1 or #2

Most network systems and applications also have these considerations

Other Differences

- Language support
  - Some features don't combine well with RPC
    - Inter procedure communicate through global variable
  - Data structure that contains pointers
    - Different binding of address on client and server
- Different semantic now days
  - Any client/service interaction in which the request is followed by a response

Review

- Why C/S?
  - Programmers make mistakes
  - Mistakes propagate easily
  - Enforce modularity

CASE STUDY: NFS

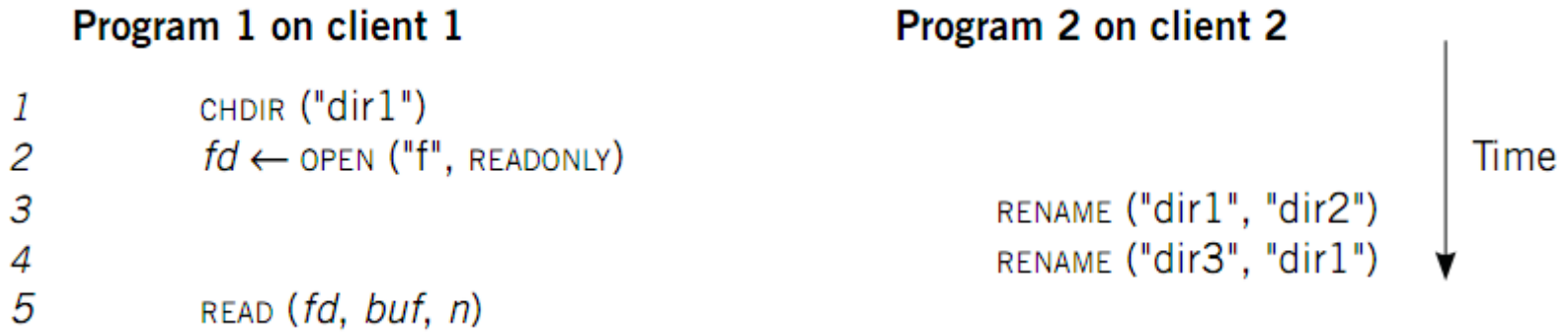
Case: NFS

- NFS (Network File System)
  - Sun, 1980s, workstations
  - Centralized management: easy to share & backup
  - Cost benefit: workstations without disks
- NFS goals
  - Compatibility with existing applications
  - Easy to deploy
  - OS independent: clients in DOS
  - Efficient enough to be tolerable

Naming Remote Files and Directories

- Transparent to programs
  - User cannot tell whether NFS or not
- Mounter
  - # mount –t nfs 10.131.250.6:/nfs/dir /mnt/nfs/dir
- File handle
  - File system identifier: for server to identify the FS
  - Inode number: for server to locate the file
  - Generation number: for server to maintain consistency
  - Usable across server failures, e.g. reboot
  - Why not put path name in the handle?

Corner Case 1: Rename After Open



- UNIX Spec:
  - Program 1 should read "dir2/f"
  - NFS should keep the spec

Corner Case 2: Delete After Open

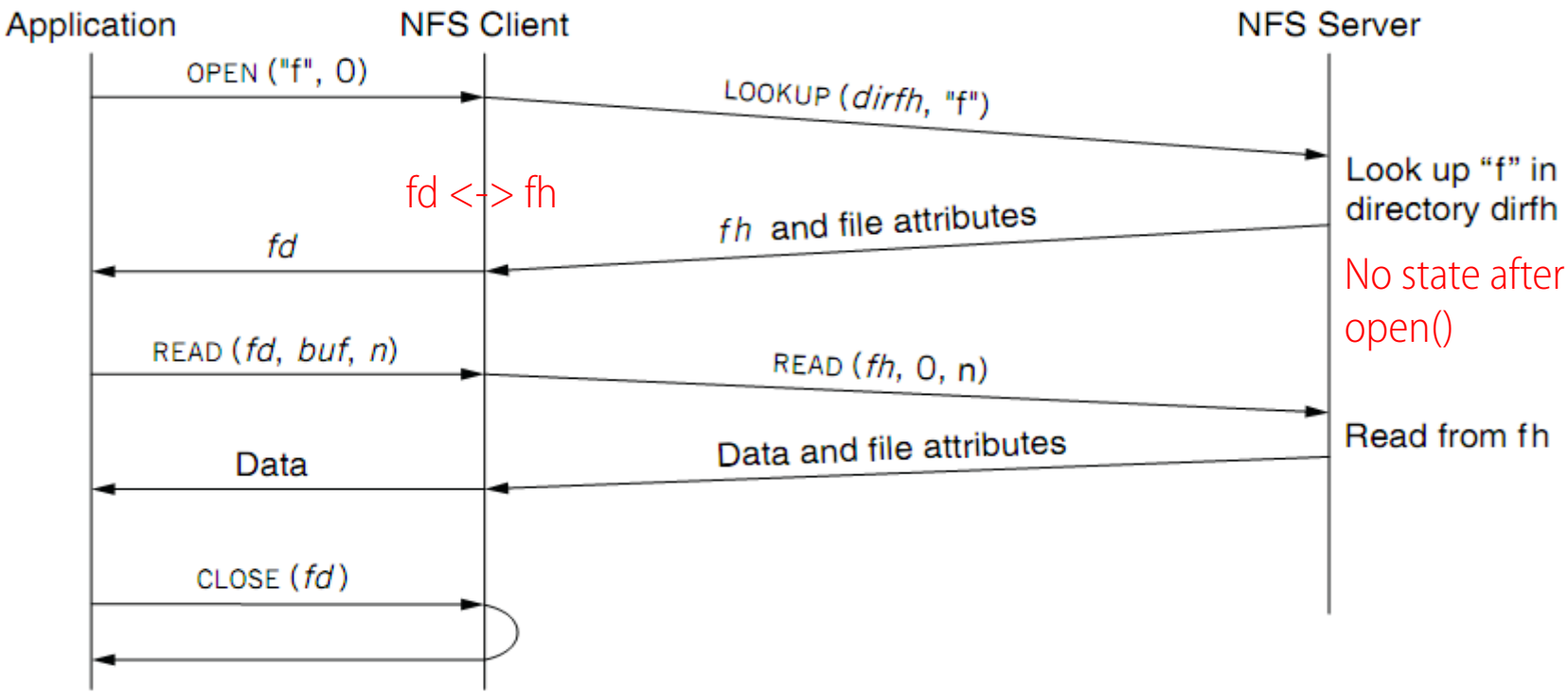
- UNIX spec:
  - On local FS, program 2 will read the old file
- How to avoid program 2 reading new file?
  - Generation number
  - "stale file handler"
- Not the same as UNIX spec! It's a tradeoff.

RPC in NFS

Table 4.1 NFS Remote Procedure Calls	
Remote Procedure Call	**Returns**
NULL ()	Do nothing.
LOOKUP (*dirfh*, *name*)	fh and file attributes
CREATE (*dirfh*, *name*, *attr*)	fh and file attributes
REMOVE (*dirfh*, *name*)	status
GETATTR (*fh*)	file attributes
SETATTR (*fh*, *attr*)	file attributes
READ (*fh*, *offset*, *count*)	file attributes and data
WRITE (*fh*, *offset*, *count*, *data*)	file attributes
RENAME (*dirfh*, *name*, *tofh*, *toname*)	status
LINK (*dirfh*, *name*, *tofh*, *toname*)	status
SYMLINK (*dirfh*, *name*, *string*)	status
READLINK (*fh*)	string
MKDIR (*dirfh*, *name*, *attr*)	fh and file attributes
RMDIR (*dirfh*, *name*)	status
READDIR (*dirfh*, *offset*, *count*)	directory entries
STATFS (*fh*)	file system information

OPEN?
CLOSE?

RPC in NFS


Application
NFS Client
NFS Server
OPEN ("f", 0)
LOOKUP (dirfh, "f")
Look up "f" in directory dirfh
fd <-> fh
fh and file attributes
fd
No state after open()
READ (fd, buf, n)
READ (fh, 0, n)
Read from fh
Data
Data and file attributes
CLOSE (fd)

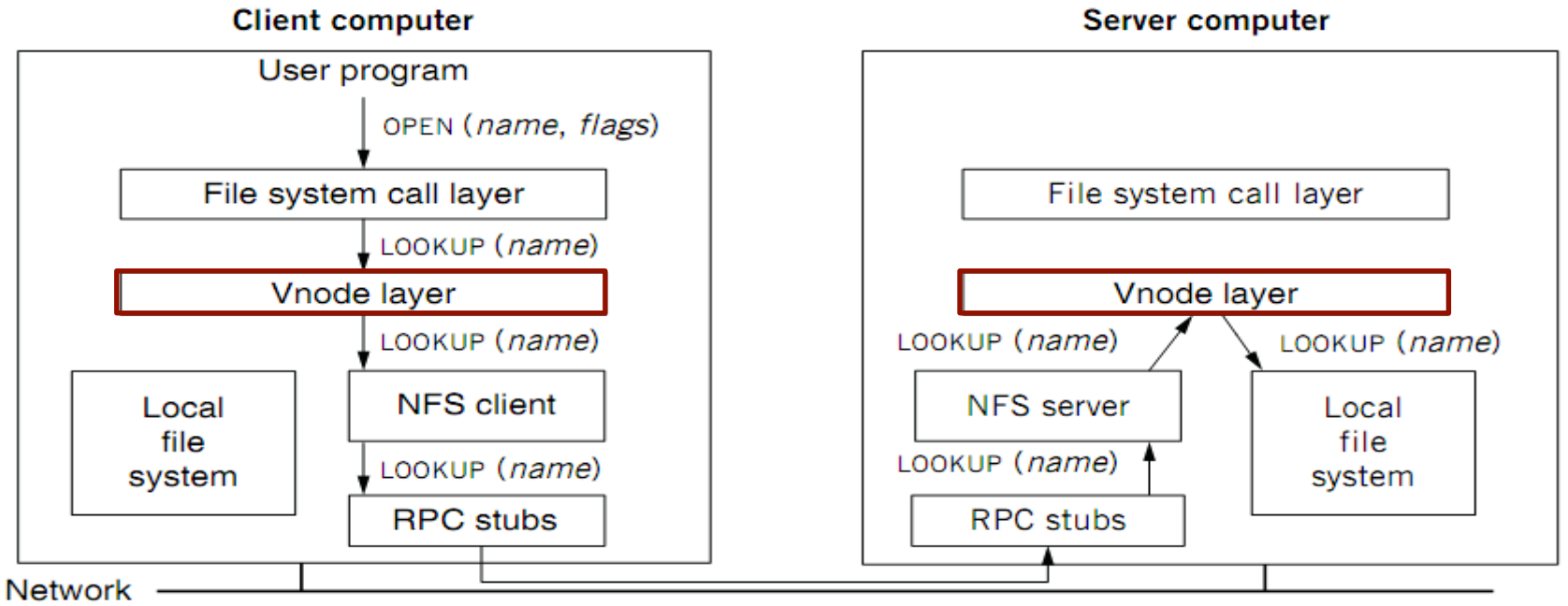
Stateless on NFS server

- Stateless on NFS server
  - Each RPC contains all the information
- What about file cursor?
  - Client maintains the state, e.g. the file cursor
- Client can repeat a request until it receives a reply
  - Server may execute the same request twice
  - Solution: each RPC is tagged with a transaction number, and server maintains some "soft" state: reply cache
  - What if the server fails between two same requests?

Extend the UNIX FS to support NFS

- Vnode
  - Abstract whether a file or dir is local or remote
  - In volatile memory, why?
  - Support several different local file system
  - Where should vnode layer be inserted?
- Vnode API
  - OPEN, READ, WRITE, CLOSE…
  - Code of fd_table, current dir, name lookup, can be moved up to the file system call layer

Extend the UNIX FS to support NFS



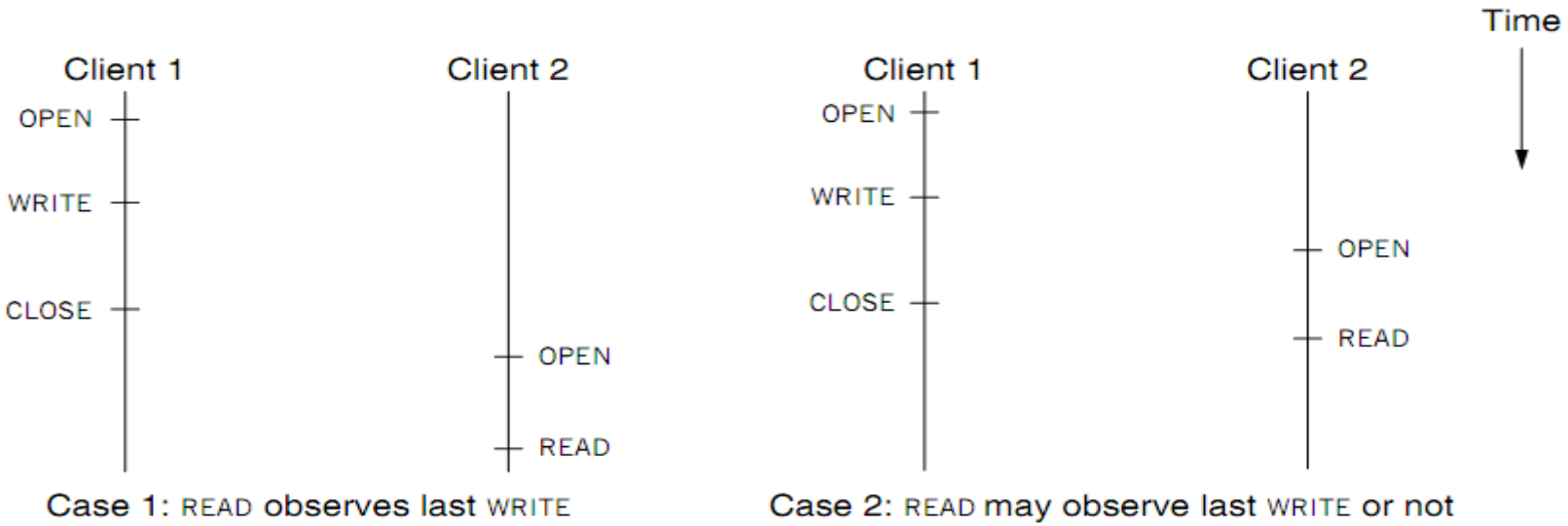
Cache on the Client

- NFS client maintains various caches
  - Stores vnode for every open file
    - Know the file handles
  - Recently used vnodes, attributes, recently used blocks, mapping from path name to vnode
- Cache benefits
  - Reduce latency
  - Less RPC, reduce load on server
- Cache coherence is needed

Coherence

- Read/write coherence
  - On local file system, READ gets newest data
  - On NFS, client has cache
  - NFS could guarantee read/write coherence for every operation, or just for certain operation
- Close-to-open consistency
  - Higher data rate
  - GETATTR when OPEN, to get last modification time
  - Compare the time with its cache
  - When CLOSE, send cached writes to the server

Coherence



Two cases of close-to-open consistency

- More consistency on chapter 9 and 10

Computer System Engineering, Spring 2015. (IPADS, SJTU)

Operating System

Enforce modularity on a single machine

C/S using Virtual Computers

- Enforce Modularity: Pros
  - Each program has its own virtual computer
  - Module isolation
  - Error containment
  - Programmer can think each one independently
- Enforce Modularity: Cons
  - A power failure knocks out all virtual computers
  - Hacker exploits bugs in virtualization layer
    - Only one of several defense lines
  - All of these are caused by one reason: sharing

Major Abstractions of a Virtual Computer

- Memory
  - Virtual memory address space
  - OS gives modules their own memory name space
- Communication
  - Bounded buffer virtual link
  - OS gives modules an inter-module communication channel for passing messages
- Interpreter
  - Thread
  - OS gives modules a slice of a CPU

C/S on a Single Physical Machine

in order to enforce modularity + build an effective operating system

programs shouldn't be able to refer to (and corrupt) each others' memory	→	Virtual memory
programs should be able to communicate	→	Assume that they don't need to (for today)
programs should be able to share a CPU without one program halting the progress of the other	→	Assume one program per CPU (for today)

Virtualization VS. Abstraction

OPERATING SYSTEMS

Virtualization VS. Abstraction

- Operating systems: enforce modularity on a single machine via virtualization and abstraction

Operating System

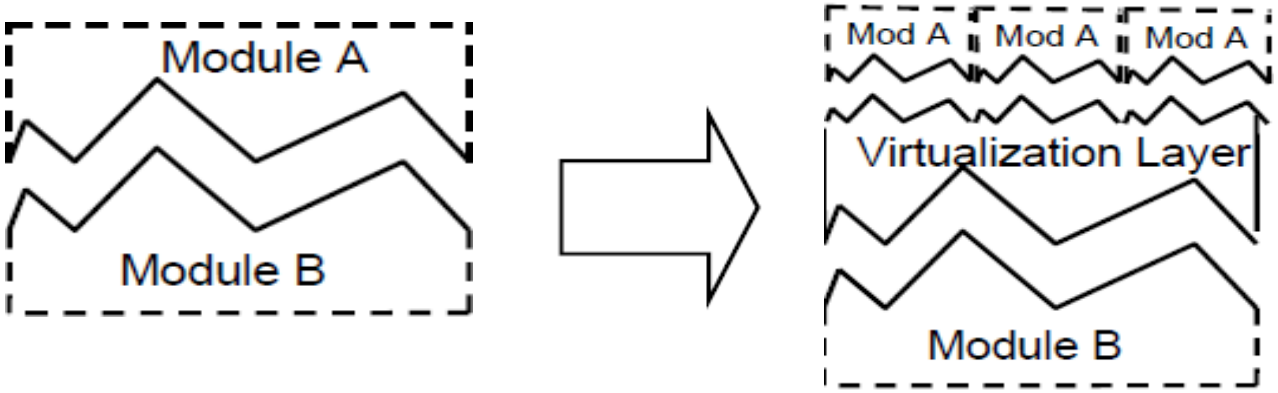
- Goals
  - Multiplexing
  - Protection
  - Cooperation
  - Portable
  - Performance
- Technologies
  - Virtualization, e.g., memory and CPU
  - Abstraction, e.g., FS, bounded buffer, window

Virtualization

- Widely used term involving a layer of indirection top of some abstraction

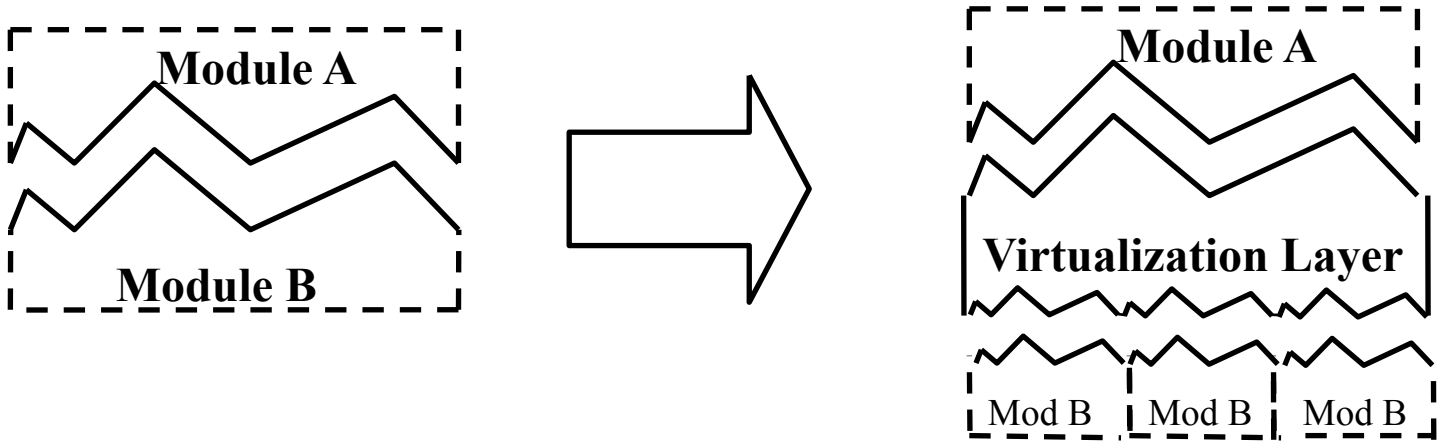
Virtual Resource or Resources
Virtualization Layer
Real (physical) Resource or Resources

Virtualization: Multiplexing



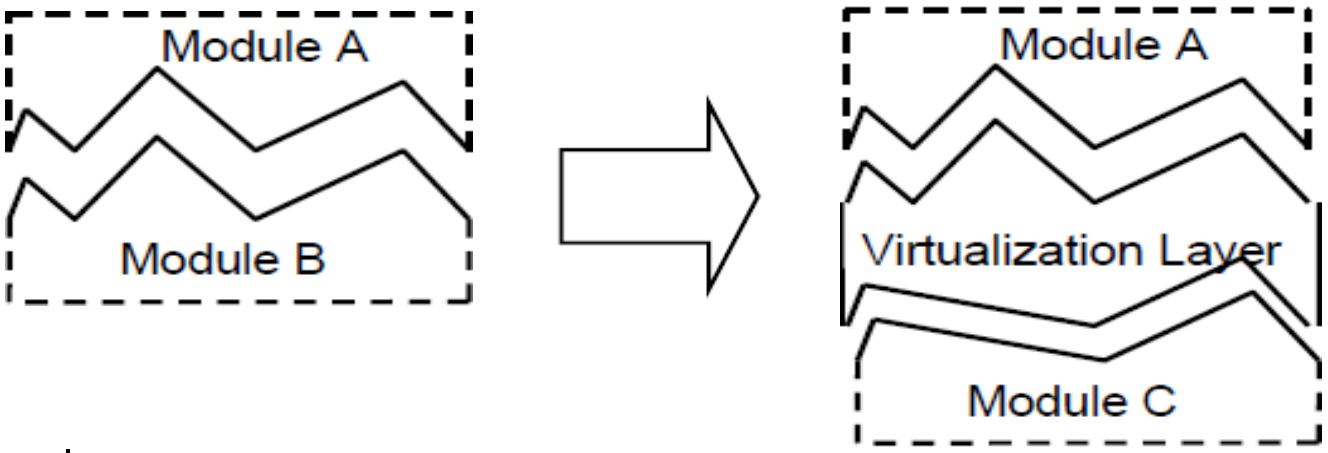
- **Examples:**
  - **Thread:** 1 CPU looks like N CPUs
  - **Virtual Memory**: 1 memory looks like N memories
  - **Virtual Circuit**: 1 channel looks like N channels
  - **Virtual Machine**: 1 machine looks line N machines

Virtualization: Aggregation



- Examples:
  - RAID disks: N disks look like one disk
  - Channel bounding: N channels look like one (e.g., the NICs)
  - Compute side: GPUs

Virtualization: Emulation

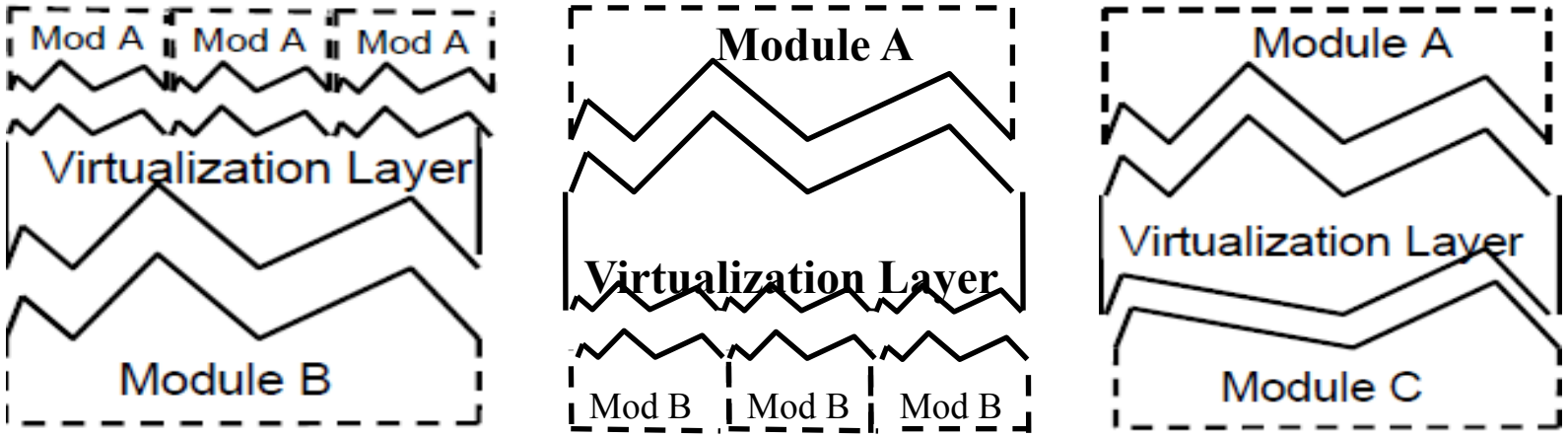


- Examples:
  - **RAM Disk**: Make a RAM memory act like a very fast disk
  - **Apple's Rosetta Technology**: Make a x86 CPU look like a Power PC CPU; *btrans* by Intel
  - **Virtual Machines**: VMware, Xen, KVM, Qemu, etc.

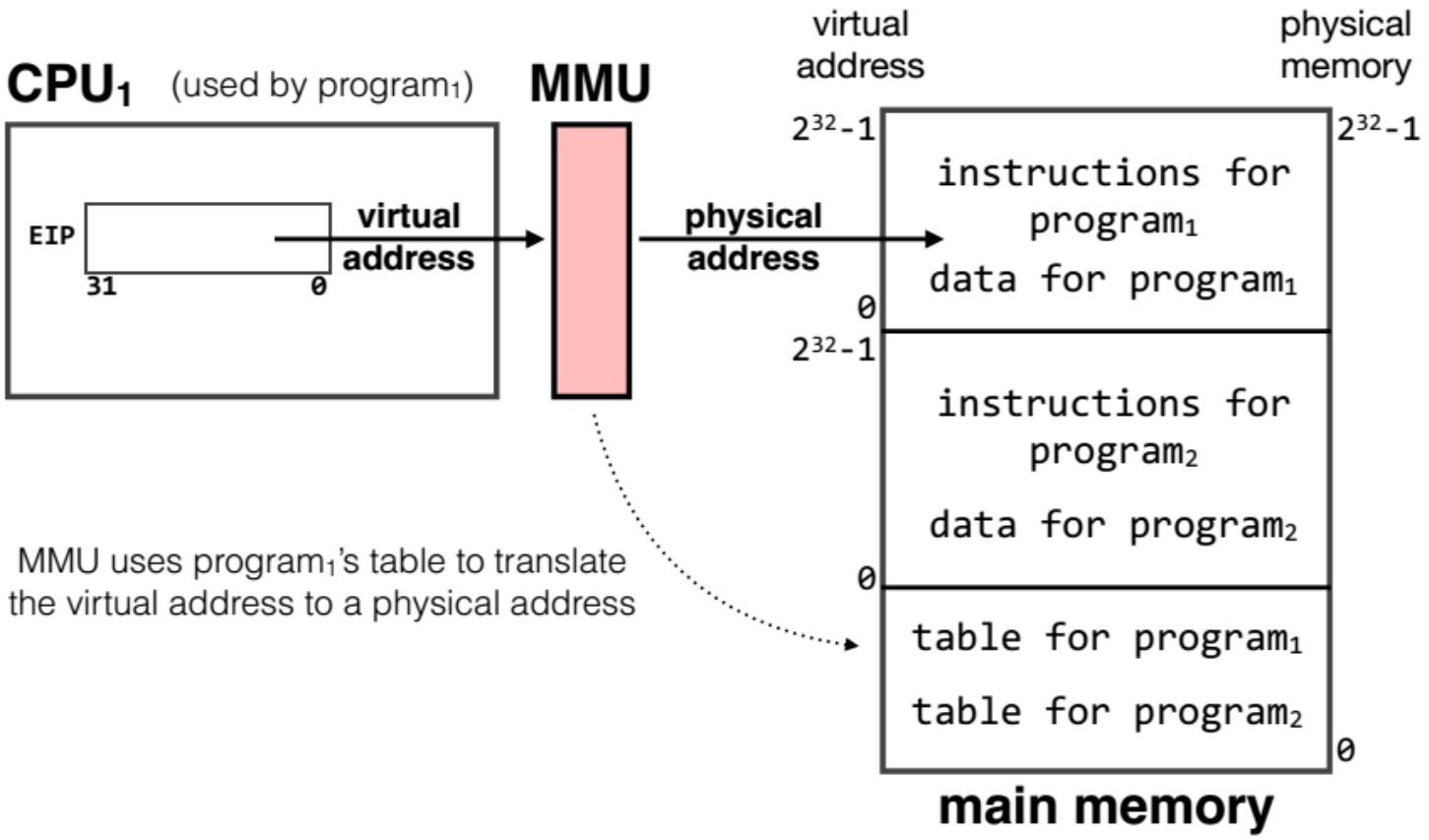
VIRTUAL MEMORY

Virtual Memory

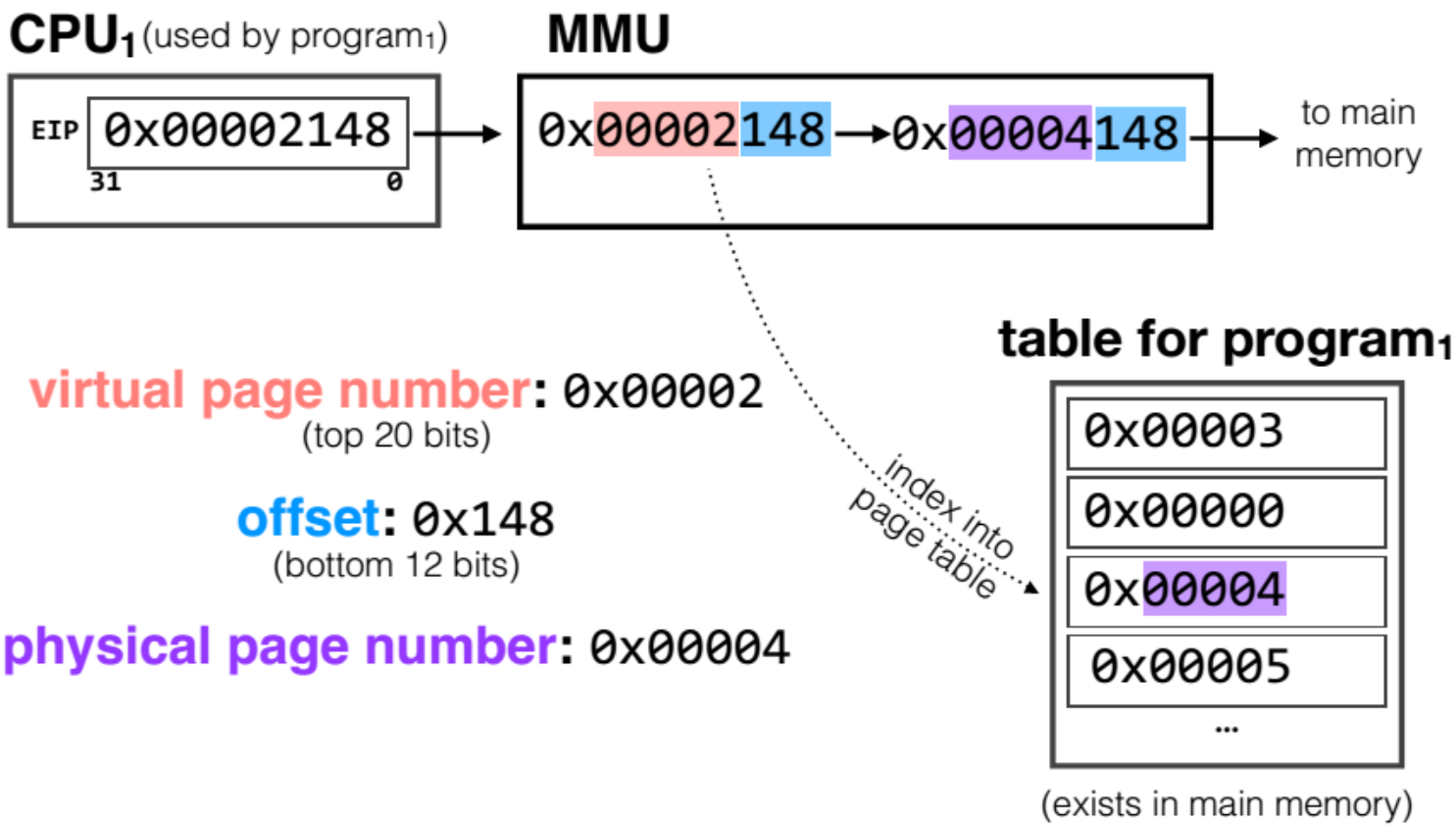
- Virtual memory: which type(s)?



Virtualize Memory

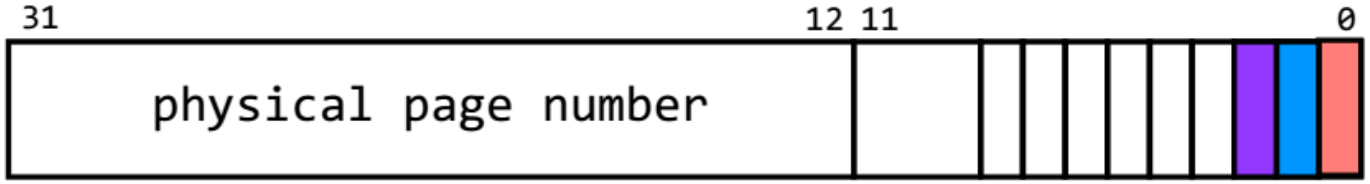


Using Page Tables



Page Table Entries

- Page table entries are 32 bits because they contain a 20-bit physical page number and 12 bits of additional information



present (P) bit: is the page currently in DRAM?

read/write (R/W) bit: is the program allowed to write to this address?

user/supervisor (U/S) bit: does the program have access to this address?

KERNEL MODE VS. USER MODE

Why Kernel Mode?

- How to protect page tables from applications?
  - Only some privileged code can access the page table
  - Must save the page tables in some memory that applications cannot access
- Other Isolation
  - Special instructions: e.g., change CR3
  - I/O channels

Kernel Mode Bit

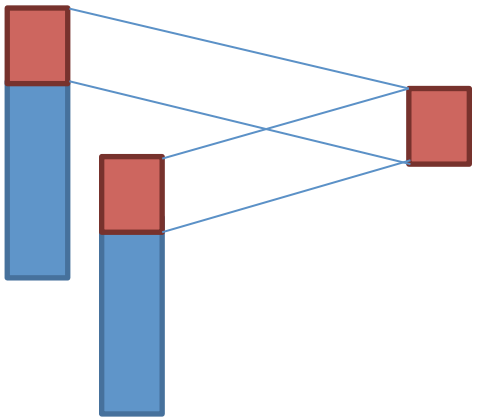
- Kernel/user bit on the processor
  - Add one bit to the processor to indicate current mode
  - Change the value of domain registers only in kernel mode
    - Generate an illegal instruction exception otherwise
  - Change the mode bit only in kernel mode
- Kernel/user bit in page table entry
  - Only the kernel can access some specific memory region

Kernel and Address Spaces

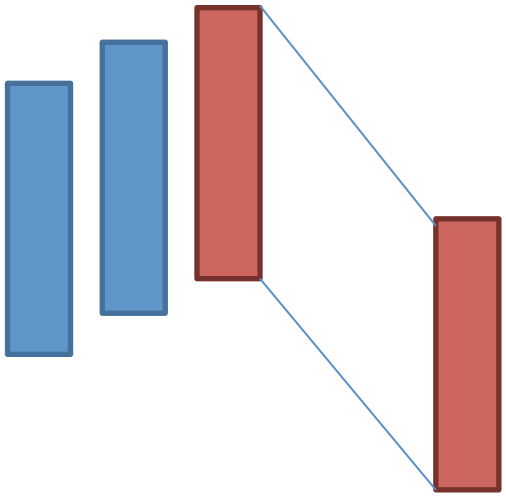
- Each address space include a mapping of the kernel into its address space
  - Only the user-mode bit must be changed when mode switches
  - The kernel sets the permissions for kernel pages to KERNEL-ONLY
- Kernel has its own separate address space
  - Inaccessible to user-level threads
  - Extend the SVC instruction to switch the page map address register

Two Design of Kernel Address Space

- Type-1: In Linux, the kernel uses 3-4G virtual memory space while applications use 0-3G virtual memory space



- Type-2: Giving the kernel a separated memory space just like an application, thus both the kernel and the applications can have 4G memory space



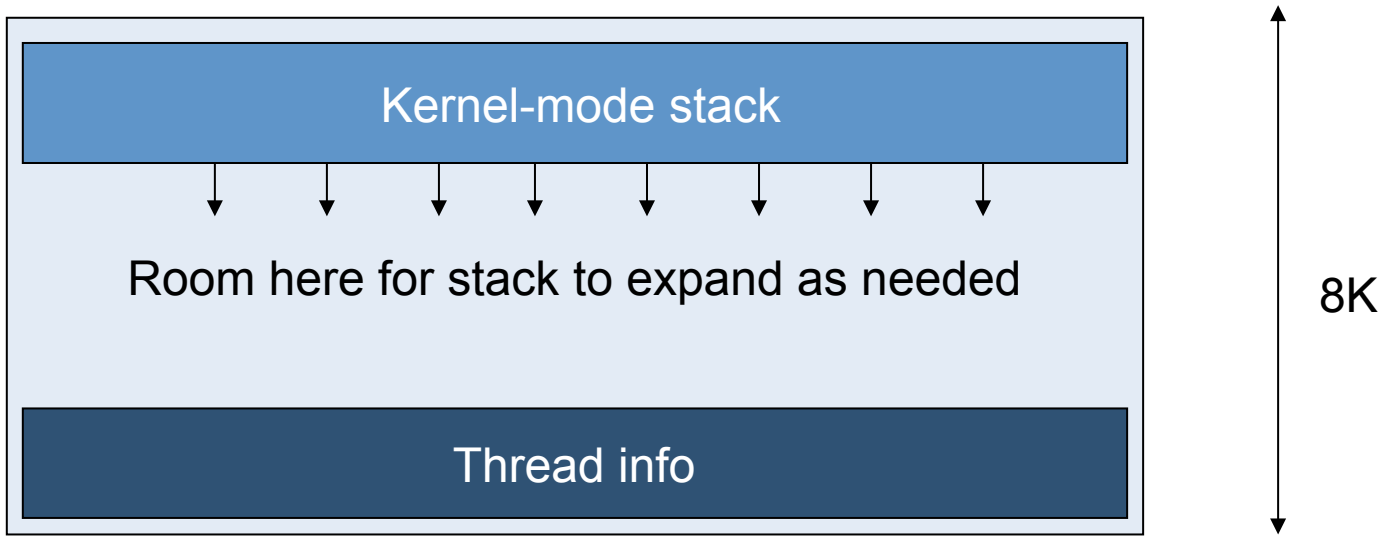
Kernel Stack in Linux

- Kernel uses its own stack
  - Kernel uses a different stack for each thread
  - It is accessible only to the kernel but is part of the process image
  - Save task's registers (e.g., user's ESP) & function parameters and return-addresses
  - Kernel also uses a different stack for each CPU for handling interrupt
- Kernel stack is small
  - In Linux, less than 8K (2 pages)
  - The stack is empty each time *iret* to user (no state!)

Kernel Stack in Linux

- Linux uses part of a task's kernel-stack to store that task's thread_info.

```
2157 union thread_union {
2158         struct thread_info thread_info;
2159         unsigned long stack[THREAD_SIZE/sizeof(long)];
2160 };
```



Kernel Stack in Linux

- Is kernel stack large enough?
  - Sometimes, no… *stack overflows* will occur
  - E.g., on older kernels, filesystem / software RAID code being interrupted by network code with iptables active
  - Solution: increase the stack size

Kernel/User Mode Switching

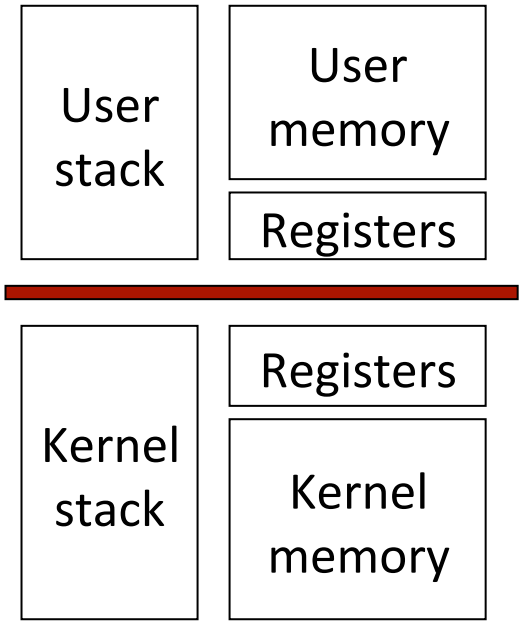
- The machine starts in K mode
- K->U: using *iret*
- U->K:
  - Hardware
    - Clock interrupt, Network/Disk interrupt
  - Software
    - Exception, System call (e.g., int 0x80)

System Call: Library Stubs in User Mode

- **Use read(fd, buf, size) as an example:**

```
int read( int fd, char * buf, int size)
{
    move fd, buf, size to R1, R2,
    move READ to R0
    int $0x80
    move result to Rresult
}
```

Linux: 80
NT: 2E



System Call: Entry Point in Kernel Mode

- **Assume passing parameters in registers**

  EntryPoint:

  switch to kernel stack

  save context

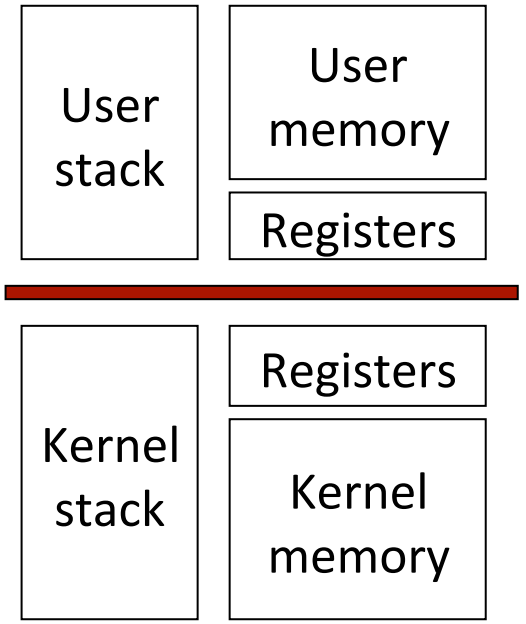
  check R_0

  call the real code pointed by R_0

  restore context

  switch to user stack

  iret (change to user mode and return)

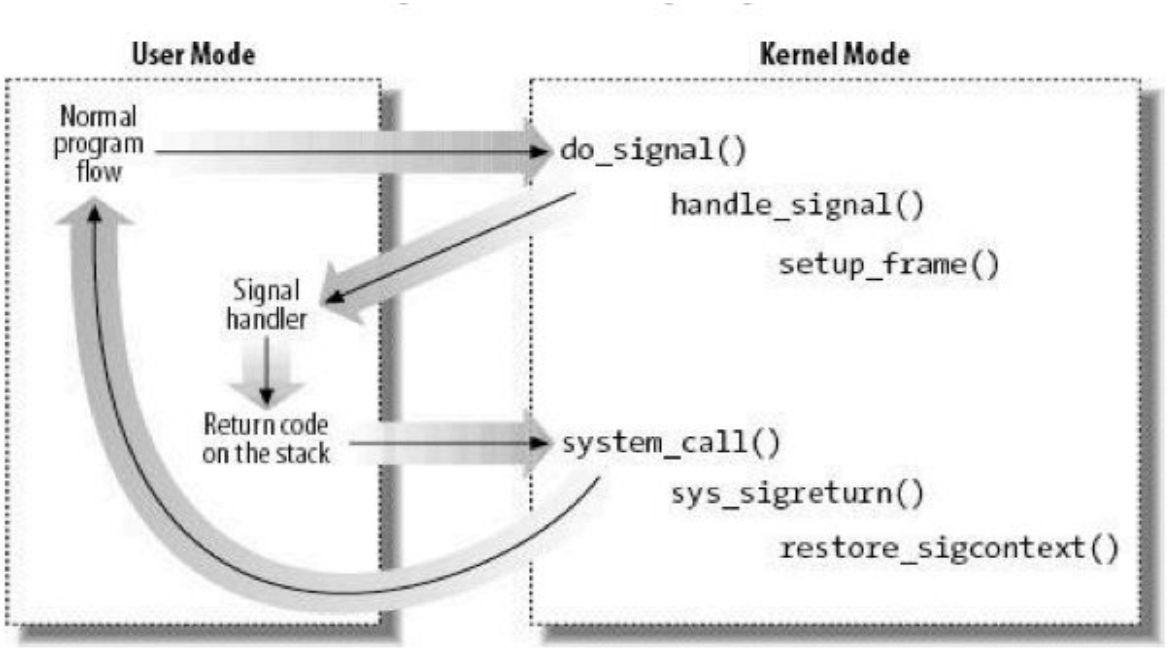


Call from Kernel to User

- UNIX Signal
  - Register signal handler
  - Kernel checks signal before *iret*
  - Kernel calls handler if a signal is pending
    - How?

```
void foo(int signo) {
  printf("You wanna kill me?\n");
  return;
}

int main() {
  signal(SIGINT, foo);
  while (1) {
    printf("I'm still alive\n");
    sleep (1)
  }
}
```

Signal Handler Process



from "Understanding Linux Kernel", 3rd Edition

TYPES OF OS STRUCTURE

Monolithic Kernel

- The kernel must be a trusted intermediary for the memory manager hardware,
  - Many designers also make the kernel the trusted intermediary for all other shared devices
    - The clock, display, disk
  - Modules that manage these devices must be part of the kernel program
    - The window manager, network manager , file manager
- Monolithic kernel
  - Most of the operating system runs in kernel mode

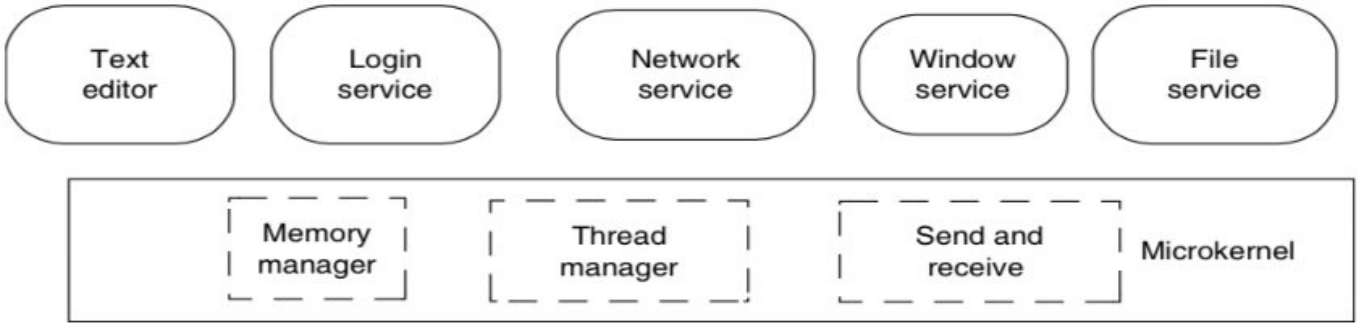
Monolithic Kernel

- Pros
  - Relatively few crossings
  - Shared kernel address space
  - Performance
- Cons
  - Flexibility
  - Stability
  - Experimentation

Microkernel

- We would like to keep the kernel small
- Reduce the number of bugs
- Restrict errors in the module which generates the error
  - The file manager module error may overwrite kernel data structures unrelated to the file system
  - Causing unrelated parts of the kernel to fail

Microkernel



- System modules run in user mode in their own domain
- Microkernel itself implements a minimal set of abstractions
  - Domains to contain modules
  - threads to run programs
  - virtual communication links

Microkernel

- Pros
  - Easier to develop services
  - Fault isolation
  - Customization
  - Smaller kernel => easier to optimize
- Cons
  - Lots of boundary crossings
  - Really poor performance

Microkernel VS. Monolithic Kernel

- Few microkernel OS
  - Mach, L4
- Most widely-used operating systems have a mostly monolithic kernel
  - the GNU/Linux operating system
  - the file and the network service run in kernel mode
  - the X Window system runs in user mode

Microkernel VS. Monolithic Kernel

1. The system is unusable if a critical service fails
   - No matter in user mode or kernel mode
2. Some services are shared among many modules
   - It's easier to implement these services as part of the kernel program, which is already shared among all modules
3. The performance of some services is critical
   - E.g., the overhead of SEND and RECEIVE supervisor calls may be too large

Microkernel VS. Monolithic Kernel

4. Monolithic systems can enjoy the ease of debugging of microkernel systems
   - good kernel debugging tools
5. It may be difficult to reorganize existing kernel programs
   - There is little incentive to change a kernel program that already works
   - If the system works and most of the errors have been eradicated
     - the debugging advantage of microkernel begins to evaporate
     - the cost of SEND and RECEIVE supervisor calls begins to dominate

Microkernel VS. Monolithic Kernel

- How to choose
  - a working system and a better designed, but new system
  - don't switch over to the new system unless it is much better
- The overhead of switching
  - learning the new design
  - re-engineering the old system to use the new design
  - rediscovering undocumented assumptions
  - discovering unrealized assumptions

Microkernel VS. Monolithic kernel

- The uncertainty of the gain of switching
  - The claims about the better design are speculative
  - There is little experimental evidence that
    - microkernel-based systems are more robust than existing monolithic kernels

Bounded Buffer and Lock

Virtualizing communication link

Review: Virtualization VS. Abstraction

- C/S on a single computer
- Operating systems: enforce modularity on a single machine via virtualization and abstraction
- Virtualization
  - Multiplexing, aggregation, emulation
- Kernel: have higher privilege
  - So that the page tables are managed
  - Has its own stacks (as application contexts and interrupt contexts)

Kernel/User Mode Switching

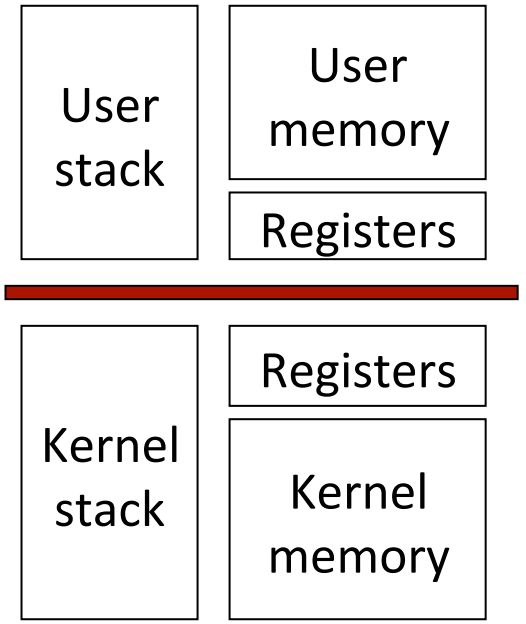
- The machine starts in K mode
- K->U: using *iret*
- U->K:
  - Hardware
    - Clock interrupt, Network/Disk interrupt
  - Software
    - Exception, System call (e.g., int 0x80)

System Call: Library Stubs in User Mode

- **Use read(fd, buf, size) as an example:**

```
int read( int fd, char * buf, int size)
{
    move fd, buf, size to R1, R2,
    move READ to R0
    int $0x80
    move result to Rresult
}
```

Linux: 80
NT: 2E



System Call: Entry Point in Kernel Mode

- **Assume passing parameters in registers**

  EntryPoint:

  switch to kernel stack

  save context

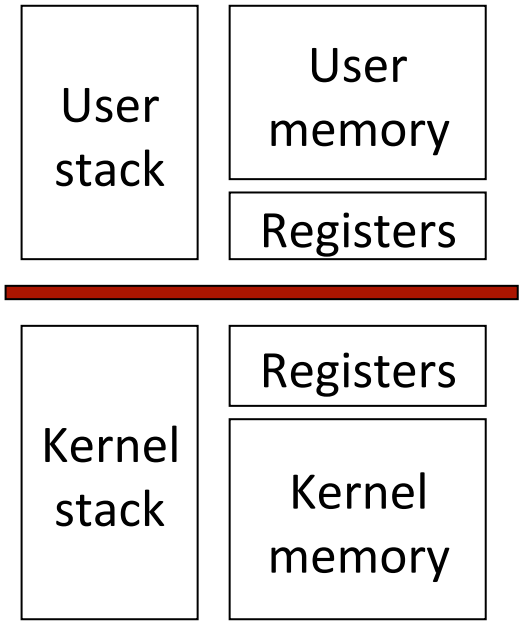
  check R_0

  call the real code pointed by R_0

  restore context

  switch to user stack

  iret (change to user mode and return)

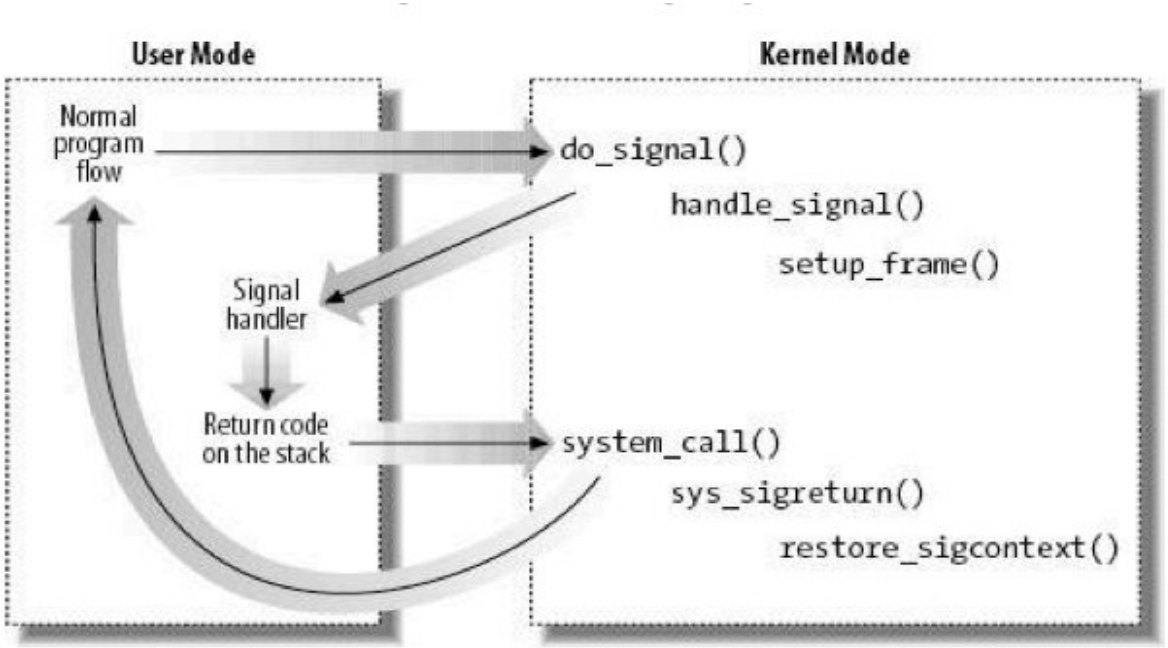


Call from Kernel to User

- UNIX Signal
  - Register signal handler
  - Kernel checks signal before *iret*
  - Kernel calls handler if a signal is pending
    - How?

```
void foo(int signo) {
  printf("You wanna kill me?\n");
  return;
}

int main() {
  signal(SIGINT, foo);
  while (1) {
    printf("I'm still alive\n");
    sleep (1)
  }
}
```

Signal Handler Process



from "Understanding Linux Kernel", 3rd Edition

Enforcing Modularity via Virtualization

in order to enforce modularity + build an effective operating system

programs shouldn't be able to refer to (and corrupt) each others' memory → Virtual memory

programs should be able to communicate → Bounded buffer (virtualize communication links)

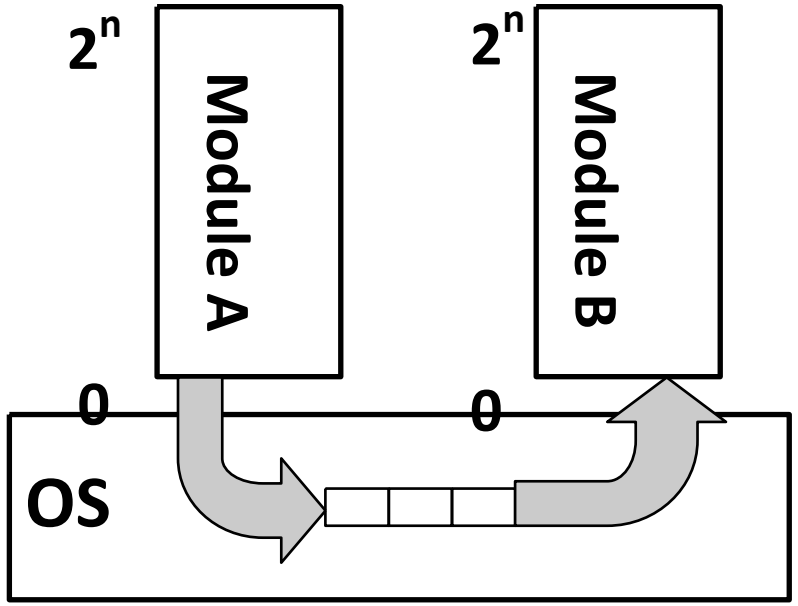
programs should be able to share a CPU without one program halting the progress of the other → Assume one program per CPU (for today)

Our Assumption So Far

- One CPU per programmer
- Programs don't need to communication
  - Today we'll deal with this
- No faults in the OS

VIRTUAL LINK: BOUNDED BUFFER

Virtual Communication Links



Very similar with the C/S model

Enforce Modularity for Bounded Buffer

- Adopt the same message-passing paradigm in C/S
  - To enable communication while keep isolation
  - Like RPC

- Share buffer in kernel
  - User mode cannot access the buffer directly
  - Applications use API to operate the buffer
  - Must transition to kernel mode to copy messages into/from the shared buffer

Bounded Buffer and its API

- Bounded buffer: a buffer that stores (up to) N messages

- Bounded buffer API
  - ✓send (m)
  - ✓m <- receive()

Enforce Modularity for Bounded Buffer

- SEND() and RECEIVE()
  - Supervisor calls (e.g., system calls)
  - Copy the message from/to the thread's domain to/from the shared buffer
  - Programs running in kernel mode is written carefully
  - Transitions between user mode and kernel mode can be expensive

Virtual Communication Links

- Producer and Consumer Problem
  - Sender must wait when buffer is full
  - Receiver must wait when buffer is empty
  - Need *sequence coordination*

BB FOR SINGLE SENDER

Bounded Buffer Send

```
send(bb, m):
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            return
```

Cannot switch the two

Bounded Buffer Send/Receive

```
send(bb, m):
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            return


receive(bb):
    while True:
        if bb.in > bb.out:
            m ← bb.buf[bb.out mod N]
            bb.out ← bb.out + 1
            return m
```

Send/Receive Implementation Assumptions

1. Single producer & Single consumer
2. Each on own CPU
3. *in* and *out* don't overflow
4. read/write coherence
5. *in* and *out* ensure before-or-after atomicity
6. The result of executing a statement becomes visible to other threads in program order
   - Compilers cannot optimize it

Concurrency

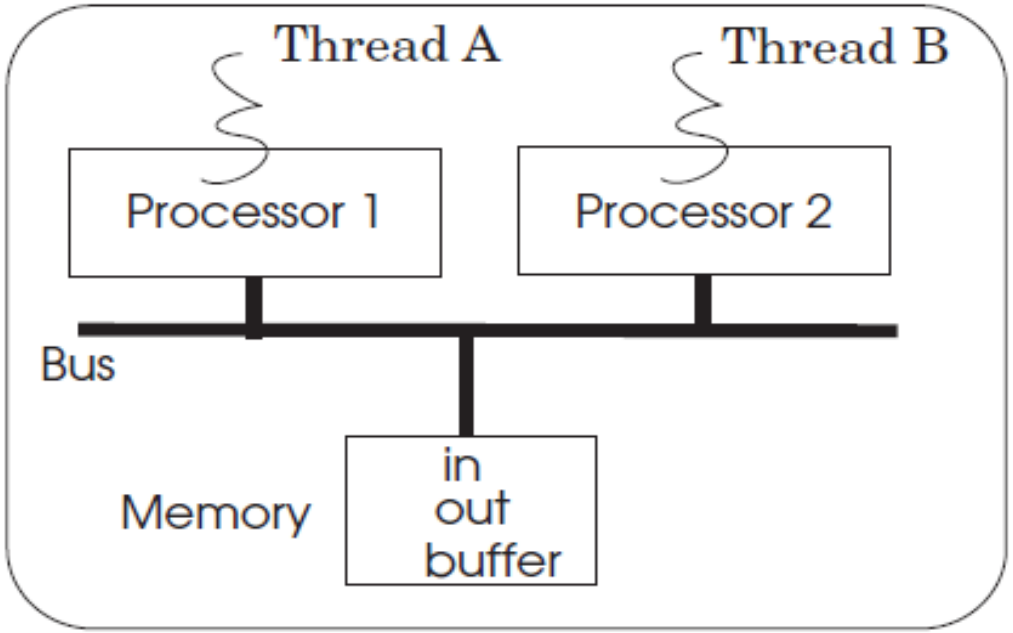
- This implementation works on two CPUs
  - One for a sender and one for a receiver
  - which is surprising!

Multiple Senders and Receivers

RACE CONDITION IF MULTI-SENDERS

Multiple Senders

- Multiple senders may send at the same time



Case 1

A

bb.in = 0, bb.out = 0

write m1 to buf[0]

set bb.in = 1

B

bb.in = 1, bb.out = 0

write m2 to buf[1]

set bb.in = 2

Everything works fine!

Case 2

A	B
bb.in = 0, bb.out = 0	
	bb.in = 0, bb.out = 0
write m1 to buf[0]	
	write m2 to buf[0]
set bb.in = 1	
	set bb.in = 1
m1 lost!	

Race Condition

- Race condition
  - Timing dependent error involving shared state
  - Whether it happens depends on how threads scheduled
  - Considering "i++" by 2 threads concurrently, with i =0:

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

Before-or-After Atomicity

- **volatile int in;**
- Works on 32 and 64 bit machines

```
mov    in,     %eax
add    $0x1, %eax
mov    %eax, in
```

Before-or-After Atomicity

- **volatile long long int in;**
- Works only on 64bit machines, not 32bit

```
mov   in,     %eax
mov   in+4,   %edx
add   $0x1,   %eax
adc   $0x0,   %edx
mov   %eax,  in
mov   %edx,  in+4
```

Race Condition — Hard to Control

- Must make sure all possible schedules are safe
  - Number of possible schedules permutations is huge
  - Bad schedules that will and will not work sometimes
- They are intermittent
  - Small timing changes between invocations might result in different behavior which can hide bug (e.g., Therac-25)
  - Heisenbugs (Heisenberg)
    - DMT (Deterministic Multi-Threading)
    - Record and Replay

Before-or-After

LOCK, TO THE RESCUE, OR NOT?

Locks: Before-or-after Atomicity

- Widely used concept in systems
- With a bunch of different names
  - Database community
    - Isolation and Isolated actions
  - Operating systems community
    - Mutual exclusion (mutex) and Critical sections
  - Computer architecture community
    - Atomicity and Atomic actions

Using Locks

- Developer must figure out the possible race conditions and insert locks to prevent them
  - Puts a heavy load on the developer
- Place acquire/release of locks in the code
  - Nothing is automatic
  - Forget one place, then you have race conditions
  - Forget to release a lock or try to acquire it twice, and you have likely have a deadlock
  - How to avoid that?

Lock API

- acquire(lock)
- release(lock)

- lock(lock)
- unlock(lock)

Send with Locking: Correct?

```
send(bb, message):
    while True:
        if bb.in – bb.out < N:
            acquire(bb.send_lock)
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
            return
```

Send with Locking: the Correct Version

```
send(bb, message):
    acquire(bb.send_lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
            return
```

LOCKS FOR FILE SYSTEM

Locks for File System

- Consider a file system that has a move() function
  - Move a file from one directory to another
  - Step-1: removing the file from directory 1
    - unlink from directory 1
  - Step-2: Place the file in directory
    - link to directory 2

move(dir1, dir2, filename)

unlink(dir1, filename)

link(dir2, filename)

Approach 1: Coarse-grained Locking

```
move(dir1, dir2, filename):
  acquire(fs_lock)
  unlink(dir1, filename)
  link(dir2, filename)
  release(fs_lock)
```

```
move(dir1, dir2, file1.txt)
move(dir3, dir4, file2.txt)
```

Forbid possible concurrency

Approach 2: Fine-grained Locking

```
move(dir1, dir2, filename):
  acquire(dir1.lock)
  unlink(dir1, filename)
        release(dir1.lock)
        acquire(dir2.lock)
  link(dir2, filename)
  release(dir2.lock)
```

Where is the file?

Approach 3: Fine-grained Locking + Holding Both Locks

```
move(dir1, dir2, filename):
    acquire(dir1.lock)
        acquire(dir2.lock)
    unlink(dir1, filename)
    link(dir2, filename)
        release(dir1.lock)
    release(dir2.lock)
```

A: move(M, N, file1.txt)
B: move(N, M, file2.txt) // M and N swapped

Deadlock!

DEADLOCK

Approach 4: Fine-grained Locking + Solving Deadlock

```
move(dir1, dir2, filename):
    if dir1.inum < dir2.inum:
        acquire(dir1.lock)
        acquire(dir2.lock)
    else:
        acquire(dir2.lock)
        acquire(dir1.lock)
    unlink(dir1, filename)
    link(dir2, filename)
        release(dir1.lock)
    release(dir2.lock)
```

- dir.inum is the inumber of a directory
- Requires global reasoning about all locks
- Need a way to ensure locks are acquired in the same order

- Aka., ordered locking

Lock

Against race condition

Review: Enforcing Modularity via Virtualization

in order to enforce modularity + build an effective operating system

programs shouldn't be able to refer to (and corrupt) each others' memory	→	Virtual memory
programs should be able to communicate	→	Bounded buffer (virtualize communication links)
programs should be able to share a CPU without one program halting the progress of the other	→	Assume one program per CPU (for today)

Review: Bounded Buffer Send/Receive

```
send(bb, m):
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            return

receive(bb):
    while True:
        if bb.in > bb.out:
            m ← bb.buf[bb.out mod N]
            bb.out ← bb.out + 1
            return m
```

If the buffer is unavailable yet, wait by spinning

Review: Multiple Sender

A	B
bb.in = 0, bb.out = 0	
	bb.in = 0, bb.out = 0
write m1 to buf[0]	
	write m2 to buf[0]
set bb.in = 1	
	set bb.in = 1
m1 lost!	

Review: Send with Locking

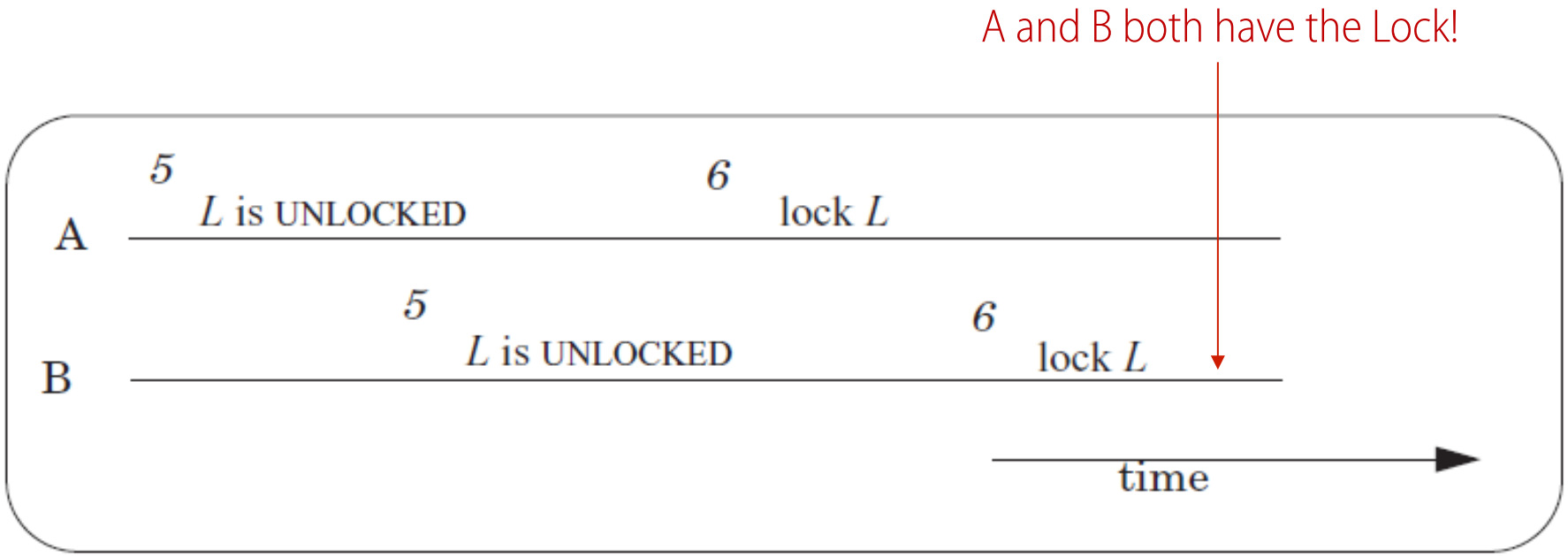
```
send(bb, message):
    acquire(bb.send_lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] <- message
            bb.in <- bb.in + 1
            release(bb.send_lock)
            return
```

IMPLEMENTING THE LOCK

Lock Implementation (Incorrect)

```
structure lock {
    integer state;
}
void acquire (lock refrerence L) {
    while L.state = LOCKED {;}          // spin until L is UNLOCKED
    L.state = LOCKED;                   // the while test failed, got the lock
}
void release (lock reference L) {
    L.state = UNLOCKED;
}
```

Race Condition Still Exists!



Operation-1: Read L to check if its state is LOCKED

Operation-2: Write L to change its state to LOCKED

Not atomic!

Need another lock?

RSM: READ and SET Memory (by Hardware)

```
1  procedure RSM (reference mem) // RSM memory location mem
2      do atomic
3          r ← mem                    // Load value stored at mem into r
4          mem ← LOCKED               // Store LOCKED into memory location
5      return r
```

Implementing ACQUIRE and RELEASE

```
1  procedure ACQUIRE (lock reference L)
2      R1 ← RSM (L.state)              // read and set lock L
3      while R1 = LOCKED do            // was it already locked?
4          R1 ← RSM (L.state)          // yes, do it again, till we see it wasn't
5
6  procedure RELEASE (lock reference L)
7      L.state ← UNLOCKED
```

RSM: Test-and-Set

- Test-and-Set
  - Load the memory, return its original value
  - Set the memory to 1 only if its original value is not 1
  - The load and set are combined as an atomic operation
- ACQUIRE()
  - `while TEST_AND_SET(L) == LOCKED do nothing`
- Other Versions of RSM
  - Compare-and-swap(v1, m, v2)
  - Non-blocking

Using the One-Writer Principle (Software)

```
shared boolean flag[N]                                  // one boolean per thread

1   procedure RSM (lock reference L)                    // set lock L and return old value
2       do forever                                      // me is my index in flag
3           flag[me] ← TRUE                             // warn other threads
4           if ANYONE_ELSE_INTERESTED (me) then         // is another thread warning us?
5            flag[me] ← FALSE                           // yes, reset my warning, try again
6           else
7            R ← L.state                                // set R to value of lock
8            L.state ← LOCKED                           // and set the lock
9            flag[me] ← FALSE
10           return R
11
12  procedure ANYONE_ELSE_INTERESTED (me)               // is another thread updating L?
13      for i from 0 to N–1 do
14          if i ≠ me and flag[i] = TRUE then return TRUE
15      return FALSE
```

Correctness

- Two threads
  - Case-1:
    - A: Write A, Read B, ----------
    - B: ----------, Write B, Read A
  - Case-2:
    - A: Write A, ----------, Read B, ----------
    - B: ----------, Write B, ----------, Read A
  - Case-3:
    - A: Write A, ---------- , Read B
    - B: ----------, Write B, Read A

Bootstrapping

- The faulty ACQUIRE has a multi-step operation on a shared variable (the lock)
  - We must ensure in some way that ACQUIRE itself is a before-or-after action
- Once ACQUIRE is a before-or-after action
  - We can use it to turn arbitrary multi-step operations on shared variables into before-or-after actions

Bootstrapping

- Solve the narrow problem using some specialized method
  - Might work for only that case
  - It takes advantage of the specific situation
- The general solution then consists of two parts:
  - a method for solving the special case
  - a method for reducing the general problem to the special case

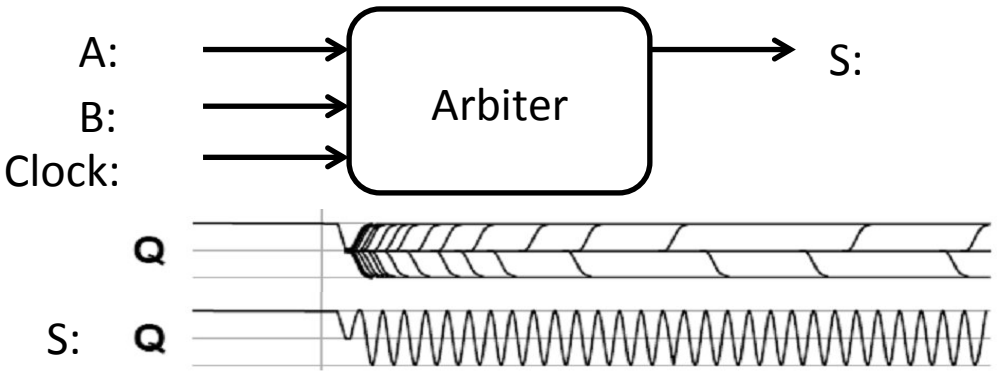
Bootstrapping (In the case of ACQUIRE)

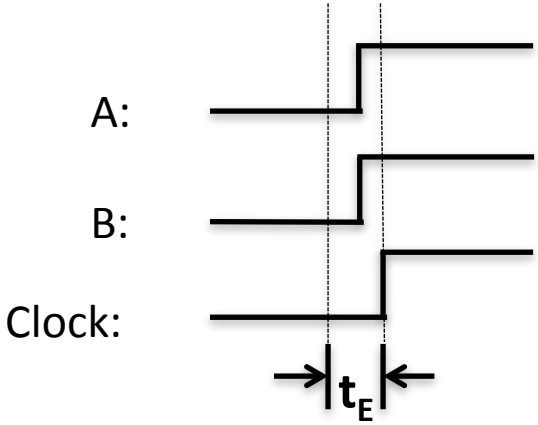
- The solution for the specific problem is
  - Hardware: building a special hardware instruction that is itself is a before-or-after action
  - Software: by some extremely careful programming

Assumptions

- Bus Arbiter
  - Guarantees that any single LOAD or STORE is a before-or-after action with respect to every other LOAD and STORE
- Each Entry of the Shared Array is:
  - in its own memory cell
  - the memory provides read/write coherence for memory cells
- The instructions execute in program order

Bus Arbiter Problem





- Two async inputs (A, B), one sync output
  - Choose between two asynchronous input
- If inputs are closely spaced
  - Output may be oscillating
  - Waiting longer only reduce the probability of oscillating

LOCK PERFORMANCE

Lock Granularity

- Systems can be distinguished by number of locks and the amount of share data they protect
- Developer must choose a locking scheme to provides the need amount of concurrency
  - Frequently concurrency leads to complexity

Lock Granularity

- Course-grain – Few locks protecting more data
  + Simpler, easier to reason about
  - Less concurrency
- Fine-grain – Many locks protecting smaller pieces of data
  + High amount of concurrency
  - More complex
  - Overhead for locks

Simple Example of Granularity

- Allocate a single lock and acquire the lock every time you enter the subsystem and release it when you leave the subsystem
  - Advantage: Simple. As long as subsystem doesn't call into itself recursively everything is fine
  - Disadvantage: No concurrency
- Frequently done if a subsystem doesn't need concurrency while other parts of the system does
  - Example: Many large software projects
    - Many modern operating systems (Example: Unix/Linux, NT)

```
1     shared structure buffer                  // A shared bounded buffer
2           message instance message[N] // with a maximum of N messages
3           long integer in initially 0      // Counts number of messages put in the buffer
4           long integer out initially 0     // Counts number of messages taken out of the buffer
5           lock instance buffer_lock initially UNLOCKED// Lock to order sender and receiver

6     procedure SEND (buffer reference p, message instance msg)
7           ACQUIRE (p.buffer_lock)
8           while p.in – p.out = N do                 // Wait until there room in the buffer
9                 RELEASE (p.buffer_lock)             // While waiting release lock so that RECEIVE
10                ACQUIRE (p.buffer_lock)             // can remove a message
11          p.message[p.in modulo N] ← msg            // Put message in the buffer
12          p.in ← p.in + 1                           // Increment in
13          RELEASE (p.buffer_lock)

14    procedure RECEIVE (buffer reference p)
15          ACQUIRE (p.buffer_lock)
16          while p.in = p.out do                     // Wait until there is a message to receive
17                RELEASE (p.buffer_lock)             // While waiting release lock so that SEND
18                ACQUIRE (p.buffer_lock)             // can add a message
19          msg ← p.message[p.out modulo N]           // Copy item out of buffer
20          p.out ← p.out + 1                         // Increment out
21          RELEASE (p.buffer_lock)
22          return msg
```

buffer_lock

One-writer Principle

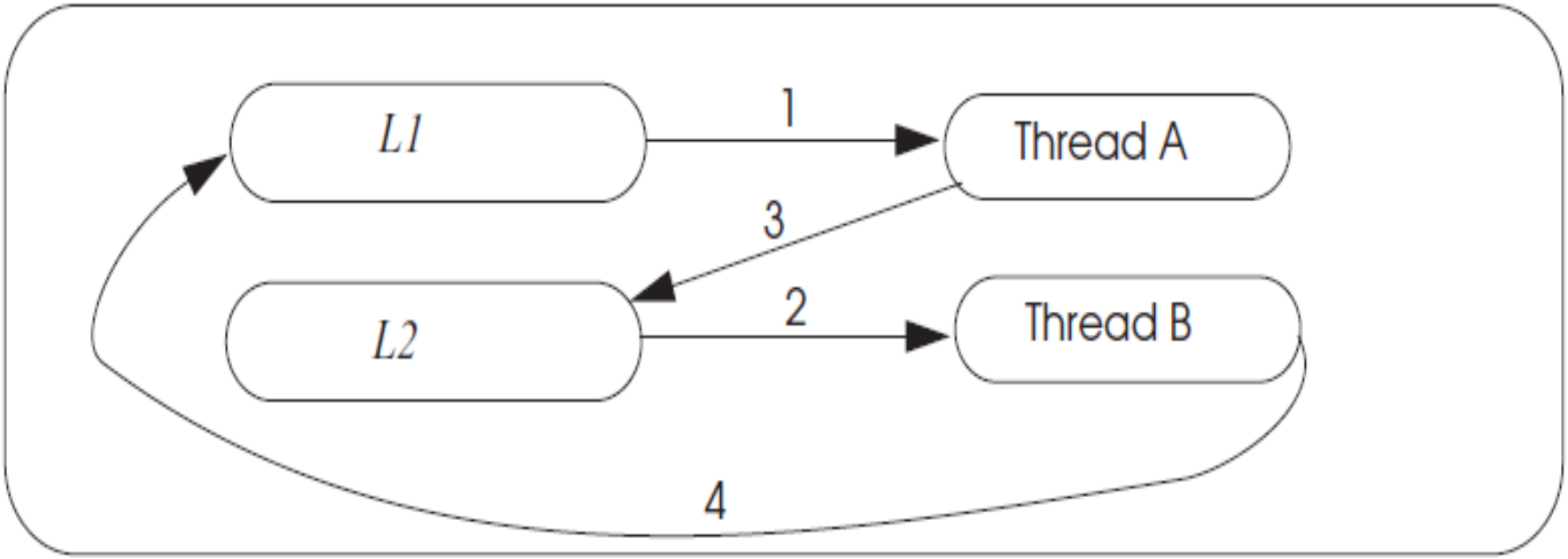
- If each variable has only one writer
  - Coordination becomes easier
  - Concurrency and read-only data is easy
  - Guide: Make as much data as you can have only a single writer
- Privatization: Make data private to a thread
    - Allocate on thread stack
    - Array indexed by thread_id()
      - E.g. privateData[thread_id()]...
- Focus locking scheme on data shared read/write

DEADLOCK, LIVELOCK, OPTIMIZATION

Deadlock

- If we had modified the code so that both threads acquire the locks in the same order
  - (*L1* and then *L2, or vice versa)*
  - No deadlock could have occurred
- Again, *small changes* in the order of statements can result in good or bad behavior

Deadlock *wait-for* Graph



Deadlock Theory

Four necessary and sufficient conditions for deadlock

1. Limited access
   - Resource can only be shared with finite users.
2. No preemption
   - Once resource granted, cannot be taken away.
3. Multiple independent requests (hold and wait)
   - Don't ask all at once (wait for next resource while holding current one)
4. Cycle in wait for graph

Deadlock & Making Progress

- Inevitable if using locks in concurrency
  - 1. Waiting for one another
  - 2. Waiting for a lock by some deadlocked one
  - Correctness arguments ensures correctness, but no progress
- Methods
  - Pessimistic ones: take a priori action to prevent
  - Optimistic ones: detect deadlocks then fix up

Methods for Solving Deadlock

- Lock ordering (pessimistic)
  - Number the locks uniquely
  - Require transactions acquire locks in order
  - Problem: some app may not predict all of the locks they need before acquiring the first one

Methods for Solving Deadlock

- Backing out (optimistic)
  - Allow acquire locks in any order
  - If it encounters an already-acquired lock with an number lower than one it has previously acquired itself, then
    - UNDO: Back up to release its higher-numbered locks
    - Wait for the lower-numbered lock and REDO

Methods for solving deadlock

- Timer expiration (optimistic)
  - Set a timer at begin_transaction, abort if timeout
  - If still no progress, another one may abort
  - Problem: how to chose the interval?

Methods for solving deadlock

- Cycle detection (optimistic)
  - Maintain a wait-for-graph in the lock manager
    - Shows owner and waiting ones
    - Check when transaction tries to acquire a lock
  - Prevent cycle (deadlock)
    - Select some cycle member to be a victim

Livelock

- An interaction among a group of threads
  - Each thread is repeatedly performing some operations
    - E.g., context saving/restoring
  - But never able to complete the whole sequence of operations
    - E.g., process the network packets

Computer System Engineering, Spring 2015. (IPADS, SJTU)

Thread

Virtualize the CPU

Virtualizing CPU

THREAD

Sharing a Processor Among Multiple Threads

- Observation
  - Many threads spend most of the time waiting
    - Waiting for specific condition
    - Most modules are consumers, e.g., mail reader
  - The processor can switch to another one
- Synonyms
  - Time sharing
  - Processor multiplexing
  - Multiprogramming, multithreading, multitasking

YIELD()

- SEND & RECEIVE
  - Using spin loop
  - Waist processor
- Using YIELD
  - When a thread calls YIELD(), gives up the CPU
  - The state changes from RUNNING to RUNNABLE
  - When YIELD() returns, the thread gets another CPU

```
procedure SEND (buffer reference p, message
    ACQUIRE (p.buffer_lock)
    while p.in − p.out = N do
        RELEASE (p.buffer_lock)
        YIELD ()
        ACQUIRE (p.buffer_lock)

    p.message[p.in modulo N] ← msg
    p.in ← p.in + 1
    RELEASE (p.buffer_lock)
```

```
procedure RECEIVE (buffer reference p)
    ACQUIRE (p.buffer_lock)
    while p.in = p.out do
        RELEASE (p.buffer_lock)
        YIELD ()
        ACQUIRE (p.buffer_lock)

    msg ← p.message[p.out modulo N]
    p.out ← p.out + 1
    RELEASE (p.buffer_lock)
    return msg
```

Implementing YIELD()

- Assumptions
  - A fixed number of threads, e.g., 7
  - Fewer processors than threads
  - Threads share memory address space
  - Threads are already running
    - Don't care creating or terminating
- 4 Procedures
  - ENTER_PROCESSOR_LAYER, GET_THREAD_ID, EXIT_PROCESSOR_LAYER, SCHEDULER()
- 2 Tables
  - *processor_table*: record current thread ID
  - *thread_table*: record thread SP
    - Protected by *thread_table_lock*

Thread Layer and Processor Layer

- A thread runs in thread layer
- A thread calls YIELD, enters processor layer
- Saves the state of the running thread
  - General purpose regs + PC + SP + CR3
- Chose another runnable thread
- Exit the processor layer and enter thread layer
- The new thread runs in thread layer

Implementing YIELD()

YIELD: among the most mysterious [code] in an OS

Yield in a Nutshell

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread                    } suspend
    threads[id].state = RUNNABLE             } current
    threads[id].sp = SP                      } thread

    do:                                      } choose
        id = (id + 1) mod N                  } new
    while threads[id].state ≠ RUNNABLE       } thread

    threads[id].state = RUNNING              } resume
    SP = threads[id].sp                      } new
    cpus[CPU].thread = id                    } thread
    release(t_lock)
```

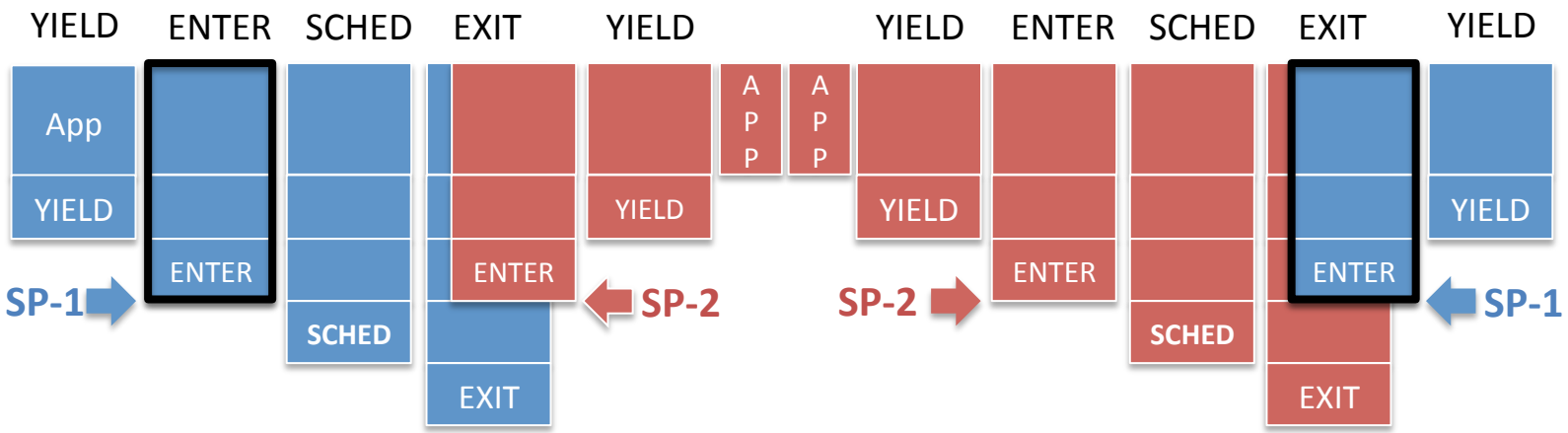
Context Switch

```
procedure YIELD ()
    ACQUIRE (thread_table_lock)
    ENTER_PROCESSOR_LAYER (GET_THREAD_ID())
    RELEASE (thread_table_lock)
    return
```

```
procedure ENTER_PROCESSOR_LAYER (this_thread)
    thread_table[this_thread].state ← RUNNABLE
    thread_table[this_thread].topstack ← SP
    SCHEDULER()
    return
```

```
procedure EXIT_PROCESSOR_LAYER (new)
    SP ← thread_table[new].topstack
    return
```

```
procedure SCHEDULER()
    j = GET_THREAD_ID()
    do
        j ← (j + 1) modulo 7
    while thread_table[j].state ≠ RUNNABLE
    thread_table[j].state ← RUNNING
    processor_table[CPUID].thread_id ← j
    EXIT_PROCESSOR_LAYER (j)
    return
```



Assumptions

- Several Assumptions
  - 1. Only fixed threads
  - 2. All threads are runnable
  - 3. Schedule using round-robin

Allocate a Thread

- thread_id ←ALLOCATE_THREAD (starting_procedure, address_space_id)
  - Allocate a new thread in address space *address_space_id*
  - The new thread is to begin with a call to the procedure specified in the argument *starting_procedure*
  - Return
    - An identifier that names the just-created thread.
    - An error

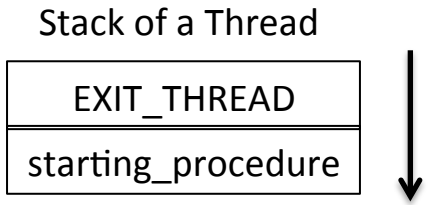
Allocate a Thread

- Allocate a range of memory in address space id
  - Used as the stack for procedure calls
- Selects a processor
- Set the processor's PC to *starting_procedure*
- Set the processor's SP to the bottom of the allocated stack

- thread_id <- **ALLOCATE_THREAD** (starting_procedure, address_space_id)

Creating a Thread

- ALLOCATE_THREAD
  - Allocate a new stack
  - Push address of *EXIT_THREAD()* as return address
  - Push address of *starting_procedure* as return address
    - *starting_procedure* is the start address of a thread
  - Initialize an entry in *thread_table*
  - Set state to RUNNABLE



Start the Scheduler

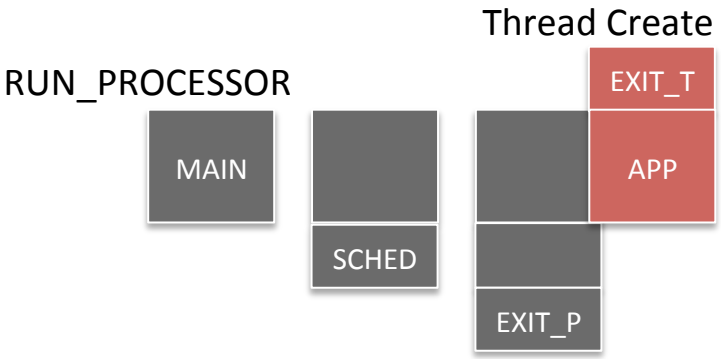
- Processor Thread
  - Create a separate thread for each processor first
  - Its stack is not used for most of the time
    - Only used when running scheduler()

RUN_PROCESSOR & Create Thread

```
procedure RUN_PROCESSORS ()
    for each processor do
        allocate stack and set up a processor thread
        shutdown ← FALSE
        SCHEDULER ()
        deallocate processor thread stack
        halt processor
```

```
procedure SCHEDULER ()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
            if thread_table[i].state = RUNNABLE then
                thread_table[i].state ← RUNNING
                processor_table[CPUID].thread_id ← i
                EXIT_PROCESSOR_LAYER (CPUID, i)
                if thread_table[i].kill_or_continue = KILL then
                    thread_table[i].state ← FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONTINUE
        RELEASE (thread_table_lock)
    return                                          // Go shut dow
```

```
procedure EXIT_PROCESSOR_LAYER (processor, tid)  //
    processor_table[processor].topstack ← SP     //
    SP ← thread_table[tid].topstack              //
    return
```



Thread Exit

```
1 procedure EXIT_THREAD()
2         ACQUIRE (threadtable_lock)
3         threadtable[tid].kill_or_continue ← KILL
4         ENTER_PROCESSOR_LAYER (GET_THREAD_ID (), CPUID)
```

Destroy a Thread

- DESTROY_THREAD
  - One thread kills another thread
  - Problem: the target thread may be running, thus the calling thread cannot just free its resource
  - Solution: asynchronization: set KILL and return
    - Wait for the thread to be destroyed when SCHEDULER

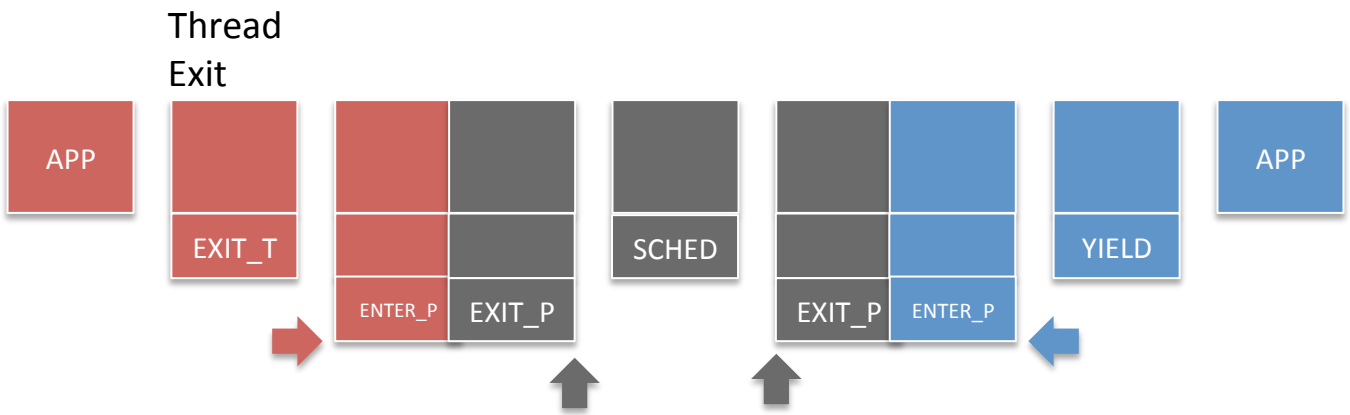
Thread Exit

```
1  procedure EXIT_THREAD()
2      ACQUIRE (thread_table_lock)
3      thread_table[tid].kill_or_continue ← KILL
4      ENTER_PROCESSOR_LAYER (GET_THREAD_ID (), CPUID)
```

```
procedure ENTER_PROCESSOR_LAYER (tid, processor)
    thread_table[tid].state ← RUNNABLE
    thread_table[tid].topstack ← SP                    //
    SP ← processor_table[processor].topstack           //
    return
```

```
procedure EXIT_PROCESSOR_LAYER (processor, tid)        //
    processor_table[processor].topstack ← SP           //
    SP ← thread_table[tid].topstack                    //
    return
```

```
procedure SCHEDULER ()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
            if thread_table[i].state = RUNNABLE then
                thread_table[i].state ← RUNNING
                processor_table[CPUID].thread_id ← i
                EXIT_PROCESSOR_LAYER (CPUID, i)
                if thread_table[i].kill_or_continue = KILL then
                    thread_table[i].state ← FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONTINUE
        RELEASE (thread_table_lock)
    return                                             // Go shut dow
```



The New Scheduler()

```
16 procedure SCHEDULER ()
17     while shutdown = FALSE do
18  ACQUIRE (threadtable_lock)
19  for i from 0 until 7 do
20     if threadtable[i].state = RUNNABLE then
21           threadtable[i].state ← RUNNING
22           processor_table[CPUID].thread_id ← i
23           EXIT_PROCESSOR_LAYER (CPUID, i)
24           if threadtable[i].kill_or_continue = KILL then
25            threadtable[i].state ← FREE
26            DEALLOCATE(threadtable[i]. stack)
27            threadtable[i]. kill_or_continue = CONTINUE
28  RELEASE (threadtable_lock)
29    return              // Go shut down this processor
```

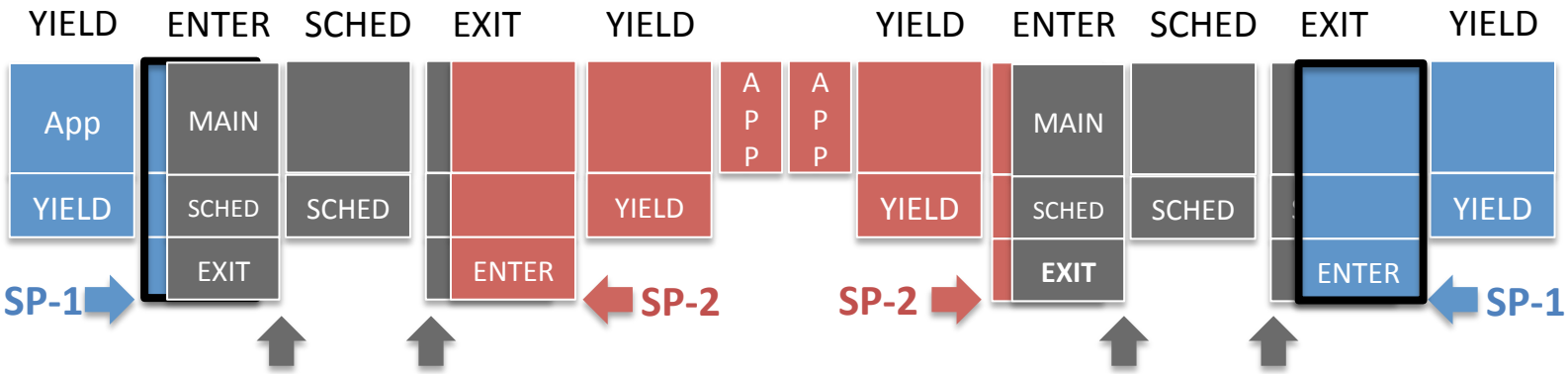
Context Switch

```
procedure YIELD ()
    ACQUIRE (thread_table_lock)
    ENTER_PROCESSOR_LAYER (GET_THREAD_ID(), CPUID)
    RELEASE (thread_table_lock)
    return
```

```
procedure ENTER_PROCESSOR_LAYER (tid, processor)
    thread_table[tid].state ← RUNNABLE
    thread_table[tid].topstack ← SP                    //
    SP ← processor_table[processor].topstack           //
    return
```

```
procedure EXIT_PROCESSOR_LAYER (processor, tid)        //
    processor_table[processor].topstack ← SP           //
    SP ← thread_table[tid].topstack                    //
    return
```

```
procedure SCHEDULER ()
    while shutdown = FALSE do
        ACQUIRE (thread_table_lock)
        for i from 0 until 7 do
            if thread_table[i].state = RUNNABLE then
                thread_table[i].state ← RUNNING
                processor_table[CPUID].thread_id ← i
                EXIT_PROCESSOR_LAYER (CPUID, i)
                if thread_table[i].kill_or_continue = KILL then
                    thread_table[i].state ← FREE
                    DEALLOCATE(thread_table[i].stack)
                    thread_table[i].kill_or_continue = CONTINUE
        RELEASE (thread_table_lock)
    return                                             // Go shut dow
```



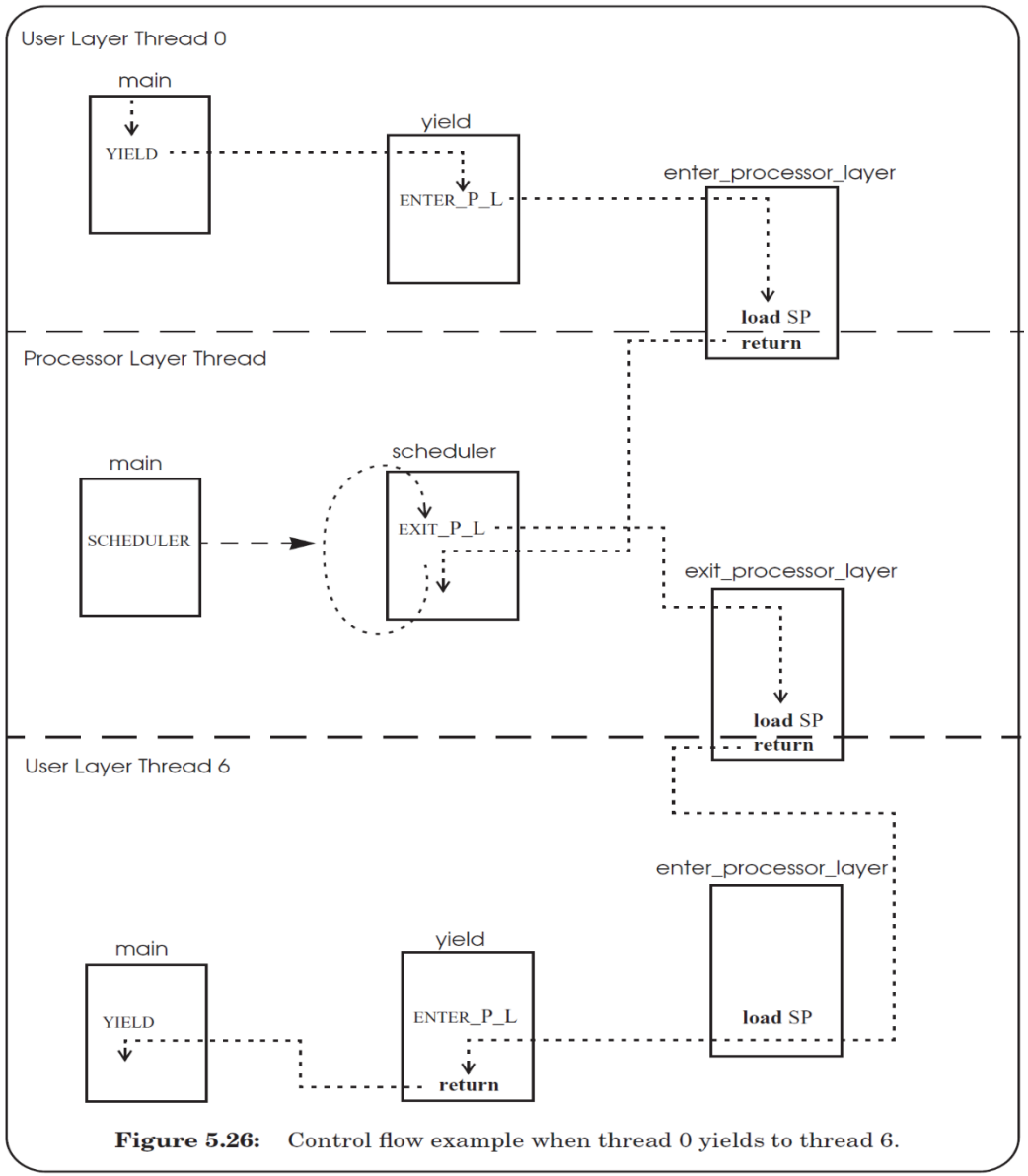
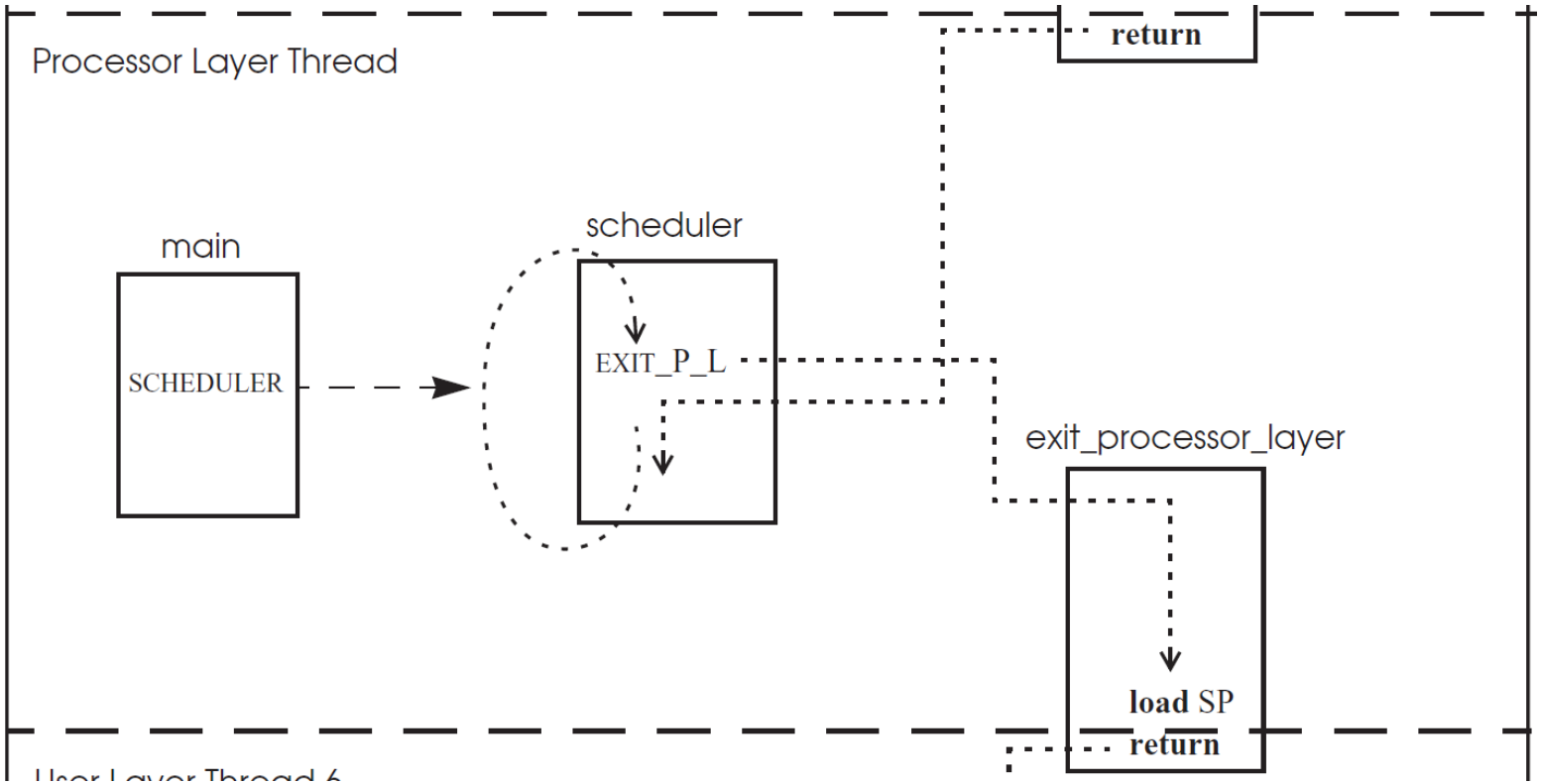


Figure 5.26: Control flow example when thread 0 yields to thread 6.


Processor Layer Thread
return
main
scheduler
SCHEDULER
EXIT_P_L
exit_processor_layer
load SP
return

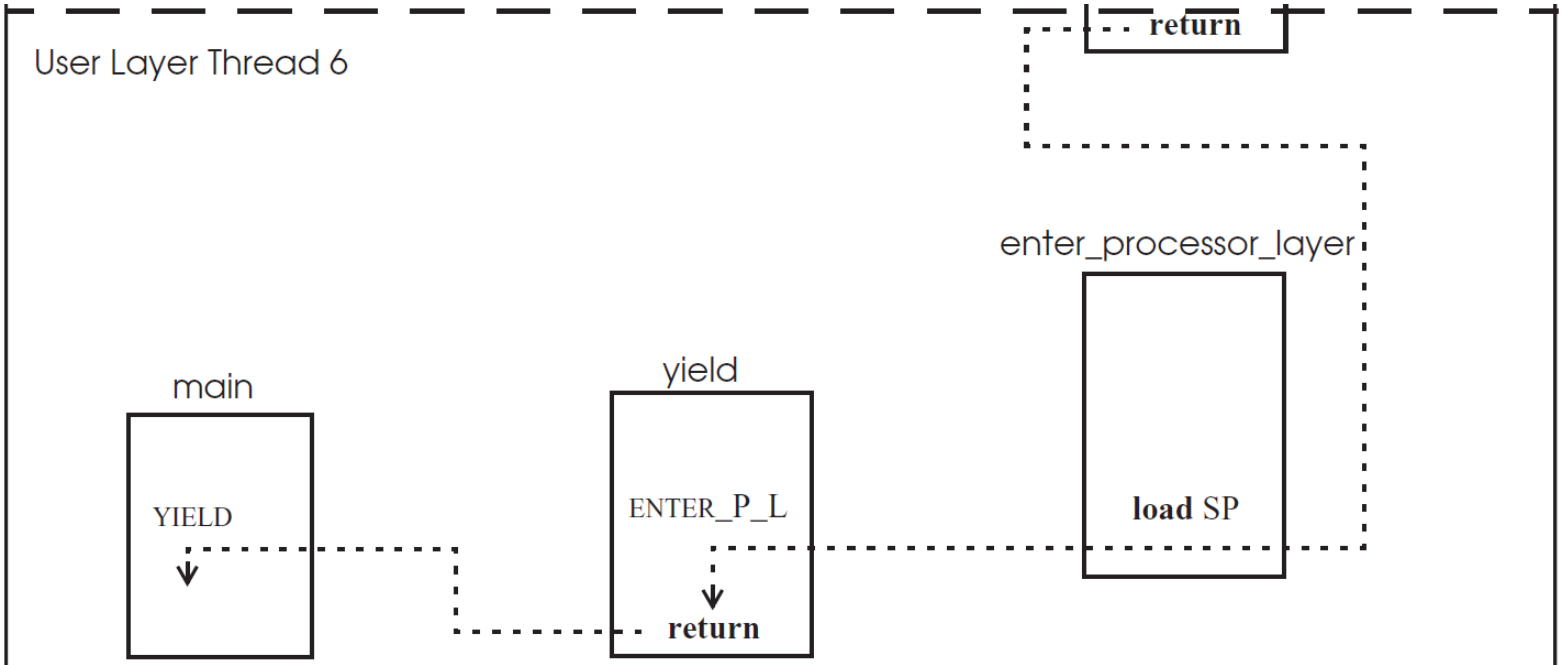
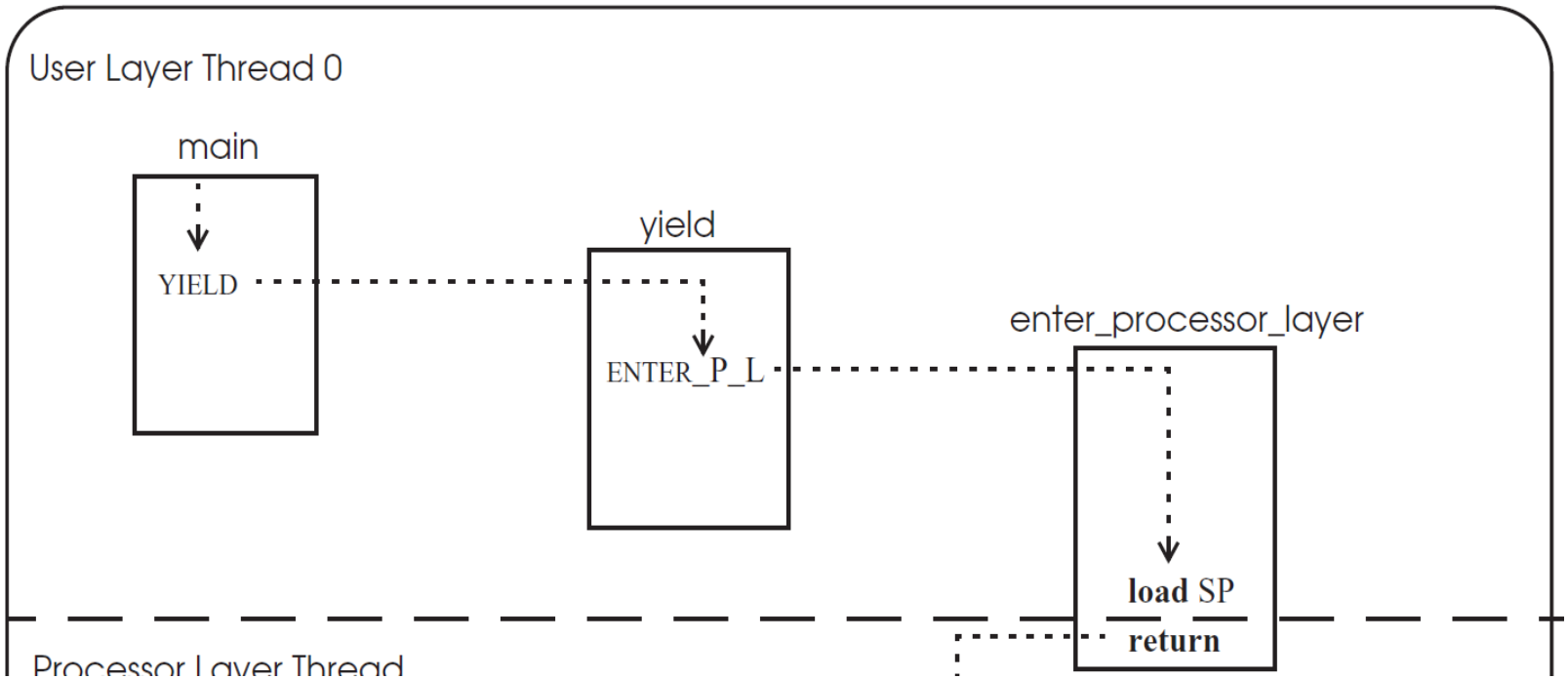
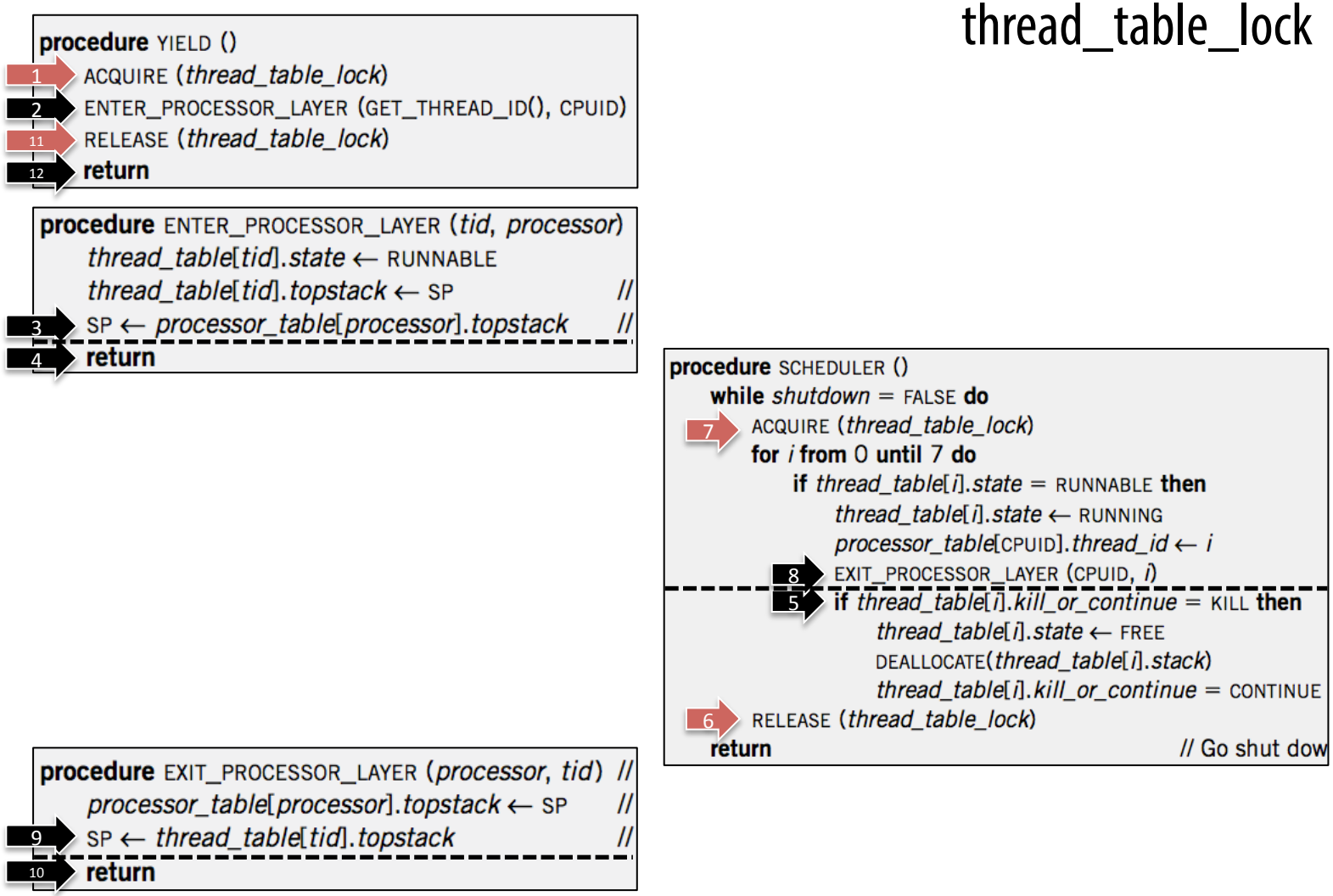


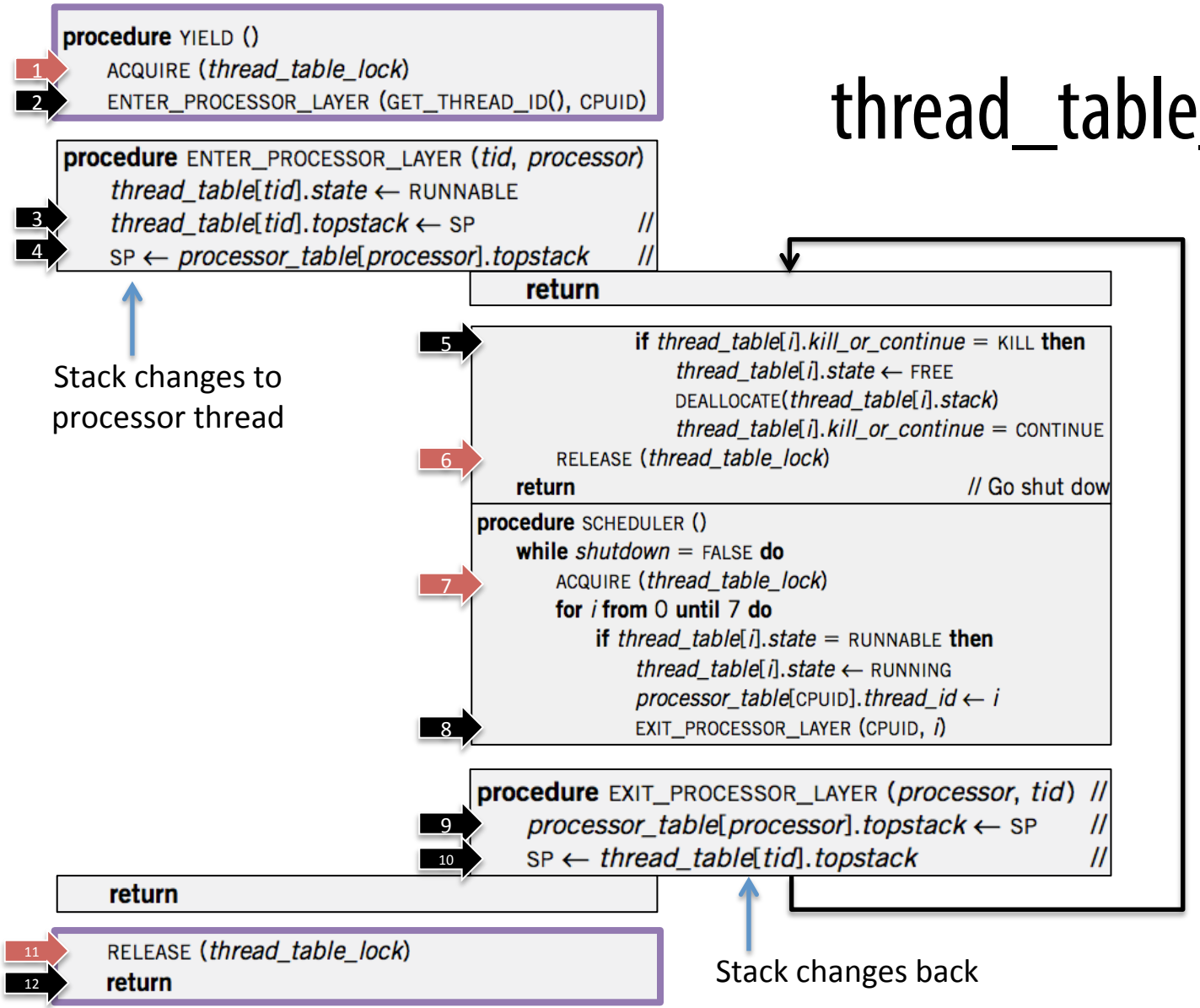
Figure 5.26: Control flow example when thread 0 yields to thread 6.


User Layer Thread 0
main
YIELD
yield
ENTER_P_L
enter_processor_layer
load SP
return
Processor Layer Thread

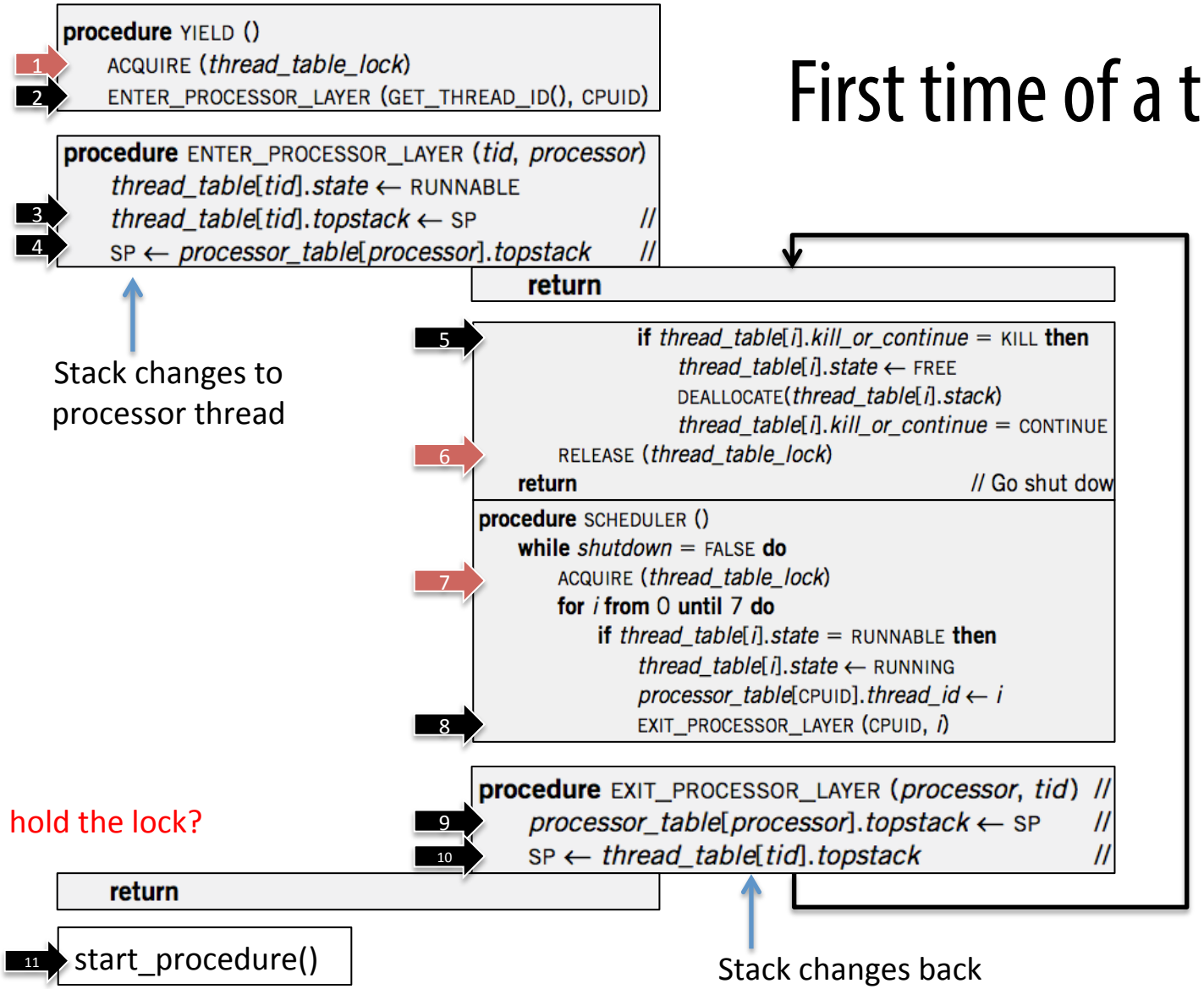
thread_table_lock



thread_table_lock



First time of a thread



Q: still hold the lock?

Preemptive Scheduling

- Non-preemptive Scheduling
  - A thread continues to run until it gives up its processor
- Cooperative Scheduling (cooperative multitasking)
  - Every thread is supposed to call YIELD periodically
- Preemptive Scheduling
  - The thread manager force a thread to give up its processor after a time interval

Implement Preemptive Scheduling

- Set the interval timer of a clock device
- When the timer expires
  - The clock triggers an interrupt
  - Switching to kernel mode in the processor layer to invoke clock interrupt handler
- The clock interrupt handler invokes an exception handler
  - Runs in the thread layer
  - Forces the currently running thread to yield

Problem: Before-or-after Again

- Interrupt Handler Calls Exception Handler
  - Interrupt handler
    - Trigger: clock interrupt
    - Context: Processor
  - Exception handler
    - Trigger: YIELD(), EXIT_THREAD()
    - Context: Thread
  - Problem
    - The contexts are different, e.g., the double-acquire lock problem
    - Should also ensures before-or-after
    - Solution: *disable interrupt when acquire thread_table_lock, and enable interrupt when release the lock*

Address Space Isolation between Threads

- Virtual address space
  - Thread manager is aware of the address space
  - Switch address space while doing thread switch
  - ENTER_PROCESSOR_LAYER saves PMAR in the thread_table
  - EXIT_PROCESSOR_LAYER loads PMAR of the new thread
- Sharing thread manager
  - Map both the instructions and the data of the thread manager into the same set of virtual addresses in every virtual address space

Computer System Engineering, Spring 2015. (IPADS, SJTU)

OS Structures

Monolithic kernel, Micro-kernel, Virtual Machine

Enforcing Modularity via Virtualization

in order to enforce modularity + build an effective operating system

programs shouldn't be able to refer to (and corrupt) each others' memory	→	Virtual memory
programs should be able to communicate	→	Bounded buffer (virtualize communication links)
programs should be able to share a CPU without one program halting the progress of the other	→	Thread (virtualize processors)

today: can we rely on the kernel to work properly?

Kernel Complexity

- 1975: Unix v6

  10,500 total lines of kernel code

- 2012: Linux 3.2

  300,000 lines: header files (data structures, APIs)
  490,000 lines: networking
  530,000 lines: sound
  700,000 lines: support for 60+ file systems
  1,880,000 lines: support for 25+ CPU architectures
  5,620,000 lines: drivers

  9,930,000 total lines of code

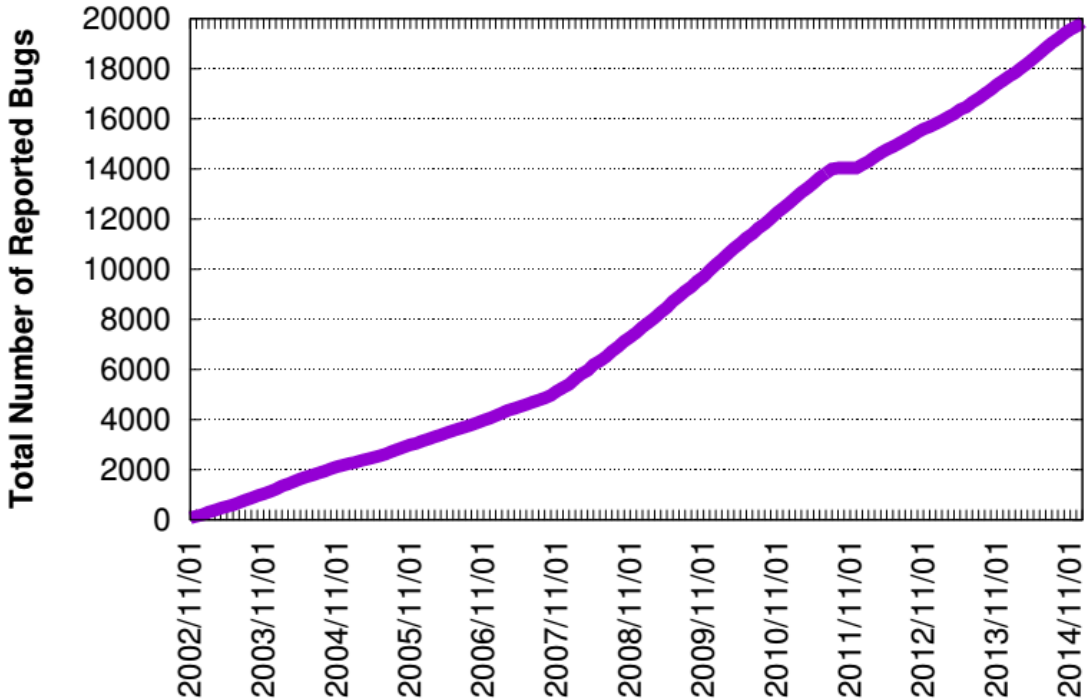
Monolithic Kernel

- Linux is Monolithic
  - No enforced modularity (e.g., Linux)
  - Many good software engineering techniques used in practice
  - But ultimately a bug in the kernel can affect entire system
- How can we deal with the complexity of such systems?
  - One result of all this complexity: bugs!

Linux Fails

- A kernel bug can
  - Cause the whole Linux system to fail
- Is it a good thing that Linux lasted this long?
  - Problems can be hard to detect, even if they may still do damage
  - E.g., maybe my files are now corrupted, but system didn't crash
  - Worse: adversary can exploit a bug to gain unauthorized access

Linux Bugs



Source: Bugzilla.kernel.com, count of all bugs currently marked NEW, ASSIGNED, REOPENED, RESOLVED, VERIFIED, or CLOSED, by creation data

TYPES OF OS STRUCTURE

Monolithic Kernel

- The kernel must be a trusted intermediary for the memory manager hardware,
  - Many designers also make the kernel the trusted intermediary for all other shared devices
    - The clock, display, disk
  - Modules that manage these devices must be part of the kernel program
    - The window manager, network manager , file manager
- Monolithic kernel
  - Most of the operating system runs in kernel mode

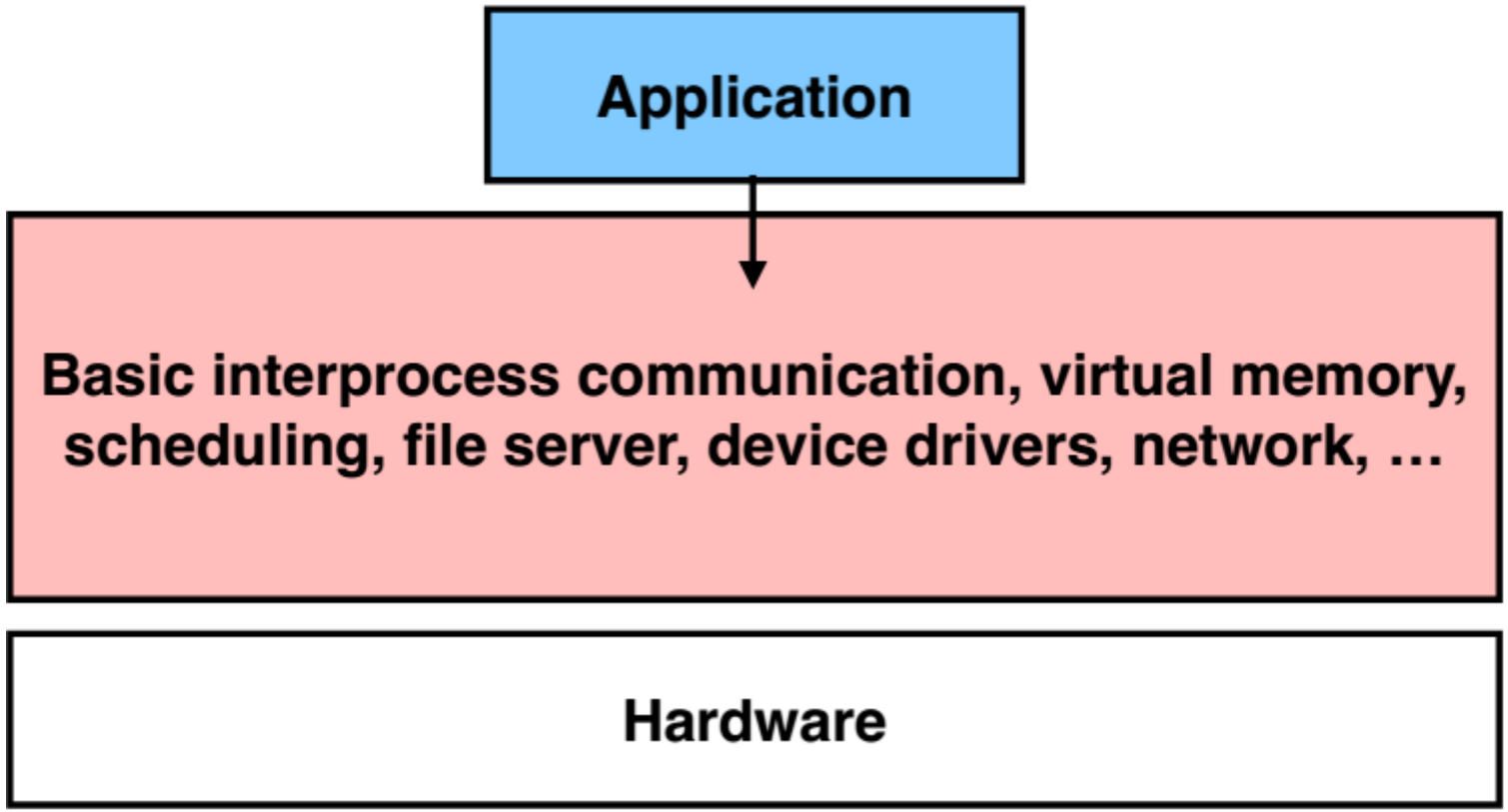
Monolithic Kernel

- Pros
  - Relatively few crossings
  - Shared kernel address space
  - Performance
- Cons
  - Flexibility
  - Stability
  - Experimentation

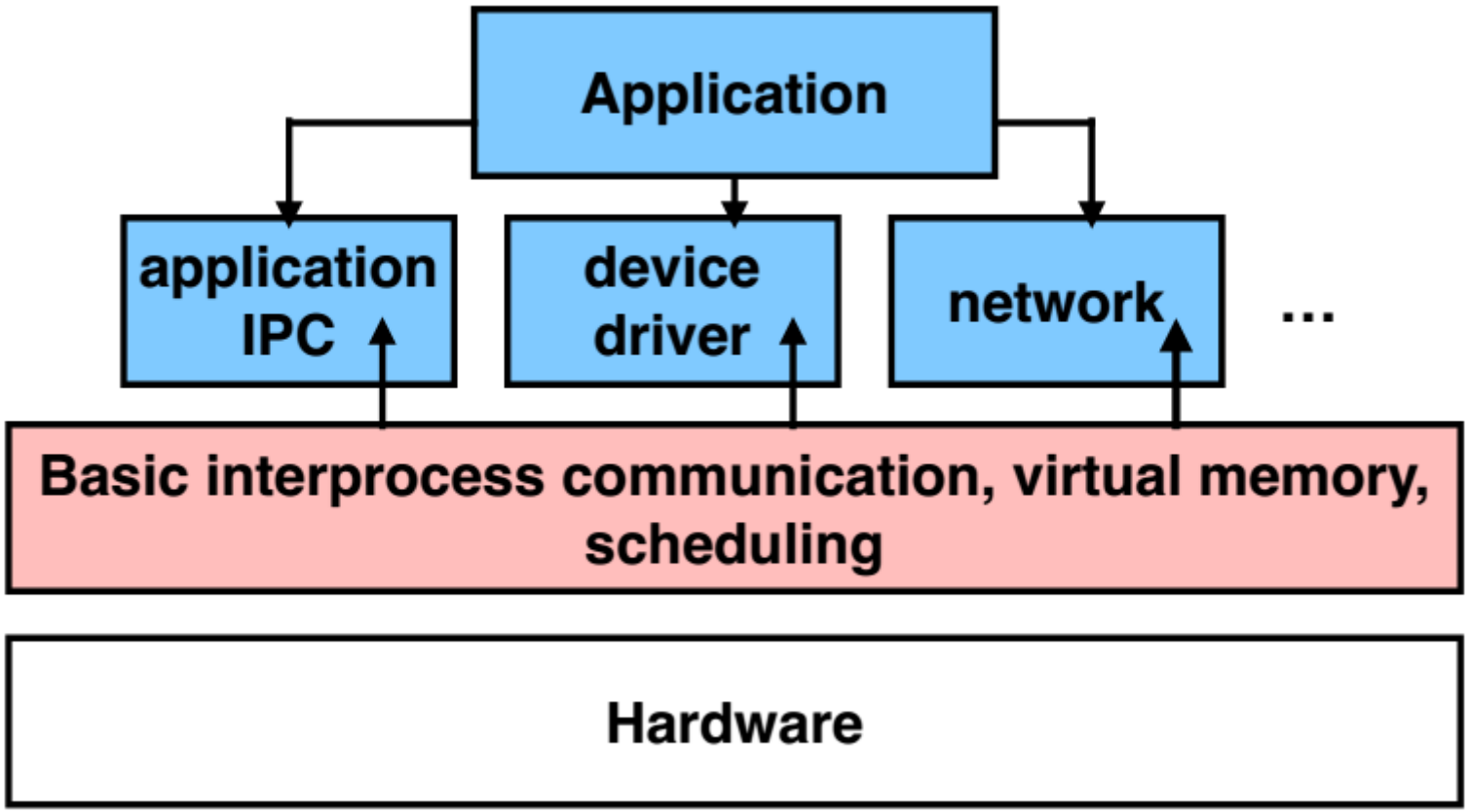
Microkernel

- We would like to keep the kernel small
- Reduce the number of bugs
- Restrict errors in the module which generates the error
  - The file manager module error may overwrite kernel data structures unrelated to the file system
  - Causing unrelated parts of the kernel to fail

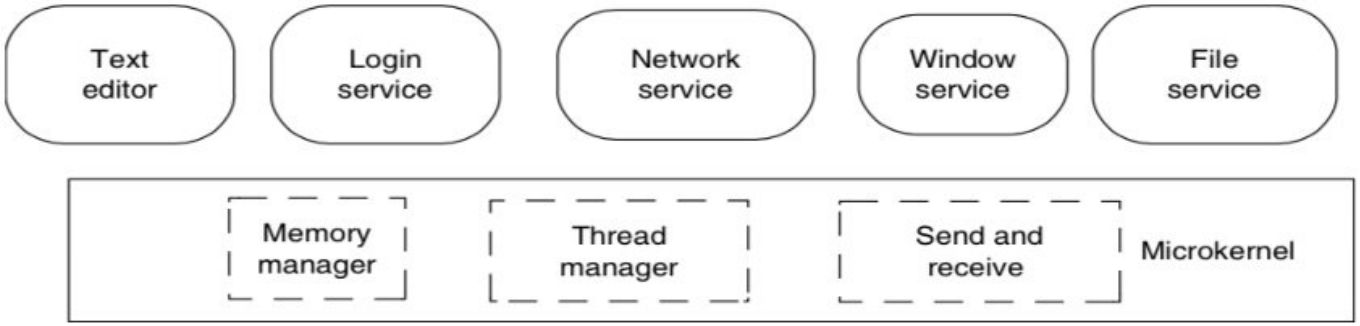
Monolithic Kernel: No Enforced Mudularity within the Kernel



Microkernel: Enforce Modularity by Putting Subsystems in User Space



Microkernel



- System modules run in user mode in their own domain
- Microkernel itself implements a minimal set of abstractions
  - Domains to contain modules
  - threads to run programs
  - virtual communication links

Microkernel

- Pros
  - Easier to develop services
  - Fault isolation
  - Customization
  - Smaller kernel => easier to optimize
- Cons
  - Lots of boundary crossings
  - Really poor performance

Microkernel VS. Monolithic Kernel

- Few microkernel OS
  - Mach, L4
- Most widely-used operating systems have a mostly monolithic kernel
  - the GNU/Linux operating system
  - the file and the network service run in kernel mode
  - the X Window system runs in user mode

Microkernel VS. Monolithic Kernel

1. The system is unusable if a critical service fails
   - No matter in user mode or kernel mode
2. Some services are shared among many modules
   - It's easier to implement these services as part of the kernel program, which is already shared among all modules
3. The performance of some services is critical
   - E.g., the overhead of SEND and RECEIVE supervisor calls may be too large

Microkernel VS. Monolithic Kernel

4. Monolithic systems can enjoy the ease of debugging of microkernel systems
   - good kernel debugging tools
5. It may be difficult to reorganize existing kernel programs
   - There is little incentive to change a kernel program that already works
   - If the system works and most of the errors have been eradicated
     - the debugging advantage of microkernel begins to evaporate
     - the cost of SEND and RECEIVE supervisor calls begins to dominate

Microkernel VS. Monolithic Kernel

- How to choose
  - a working system and a better designed, but new system
  - don't switch over to the new system unless it is much better
- The overhead of switching
  - learning the new design
  - re-engineering the old system to use the new design
  - rediscovering undocumented assumptions
  - discovering unrealized assumptions

Microkernel VS. Monolithic Kernel

- The uncertainty of the gain of switching
  - The claims about the better design are speculative
  - There is little experimental evidence that
    - microkernel-based systems are more robust than existing monolithic kernels

Another Solution

- Problem:
  - Deal with Linux kernel bugs without redesigning Linux from scratch

- One idea: run different programs on different computers
  - Each computer has its own Linux kernel; if one crashes, others not affected
  - Strong isolation (all interactions are client-server), but often impractical
  - Can't afford so many physical computers: hardware cost, power, space, ..

VIRTUAL MACHINE

Run multiple Linux on a Single Computer?

- Virtualization + Abstractions
  - New constraint: compatibility, because we want to run existing Linux kernel
  - Linux kernel written to run on regular hardware
  - No abstractions, pure virtualization

- Approach is called "virtual machines" (VM):
  - Each virtual machine is often called a *guest*
  - The equivalent of a kernel is called a "virtual machine monitor" (VMM)
  - The VMM is often called the *host*

Why Virtual Machine?

- Consolidation
  - Run several different OS on a single machine
- Isolation
  - Keep the VMs separated as error container
  - Fault tolerant
- Maintenance
  - Easy to deploy, backup, clone, migrate
- Security
  - VM introspection
  - Antivirus out of the OS

Complete Machine Simulation

```
#define REG_EAX 1;
int32_t eip;
int32_t regs[8];
int32_t segregs[4];
...
for (;;) {
   read_instruction();
   switch (decode_instruction_opcode()) {
     case OPCODE_ADD:
       int src = decode_src_reg();
       int dst = decode_dst_reg();
       regs[dst] = regs[dst] + regs[src];
       break;
     case ..
  }
  eip += instruction_length;
}
```

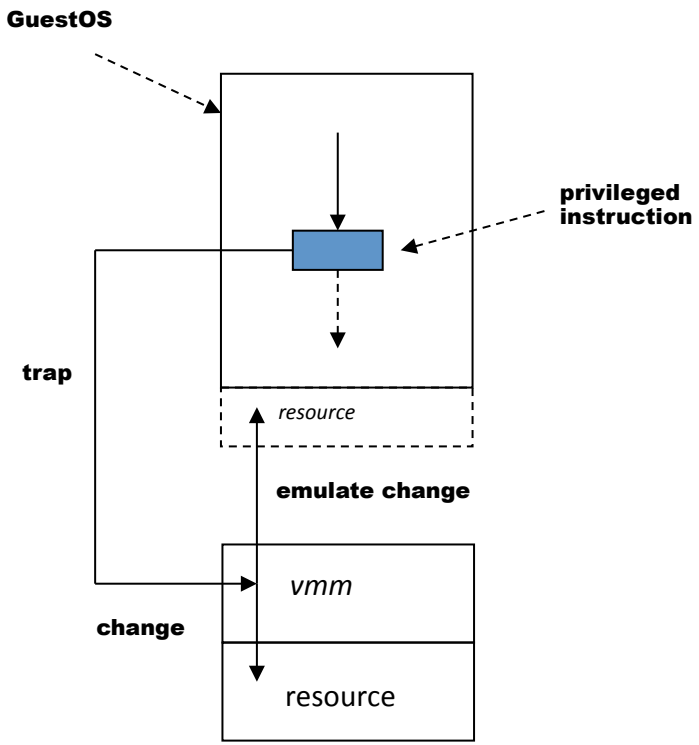
Complete Machine Simulation

- Emulate every single instruction from the guest OS
  - Often too slow in practice
  - Want to run most instructions directly on the CPU, like a regular OS kernel

Trap and Emulate

- Idea
  - Why not run most of the instruction on hardware directly?
  - If the ISA of guest and host are the same
- Problem
  - Some privileged instruction cannot be run by untrusted guest OS
  - E.g., cli (disable interrupt), load cr3 (change the page table)
- Solution
  - Trap these privileged instruction and emulate them in software

Trap and Emulate



- De-privileging guest kernel
  - VMM emulates the effect on system or hardware resources of privileged instructions whose execution traps into the VMM
  - Typically achieved by running Guest OS at a lower hardware priority level than the VMM
    - E.g., ring-3
  - Problematic on some architectures where privileged instructions do not trap when executed at deprivileged priority

Virtualization

- Memory virtualization
  - Enable each guest VM has its own virtual MMU
  - Keep isolation between guest VMs
- CPU virtualization
  - Enable each guest VM has its own kernel and user modes
  - Keep isolation between guest's kernel and user modes
- I/O virtualization
  - Enable each guest VM has its own virtual devices

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

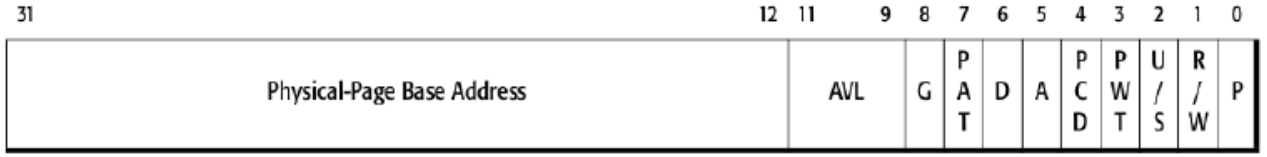
Virtualizing the PTP Register & Page Tables

- Terminology: 3 levels of addressing now
  - Guest virtual. Guest physical. Host physical
  - Guest VM's page table contains guest physical address
  - Hardware page table must point to host physical address (actual DRAM locations)
- Setting hardware PTP register to point to guest page table would not work
  - Processes in guest VM might access host physical address 0~1GB
  - But those host physical addresses might not belong to that guest VM

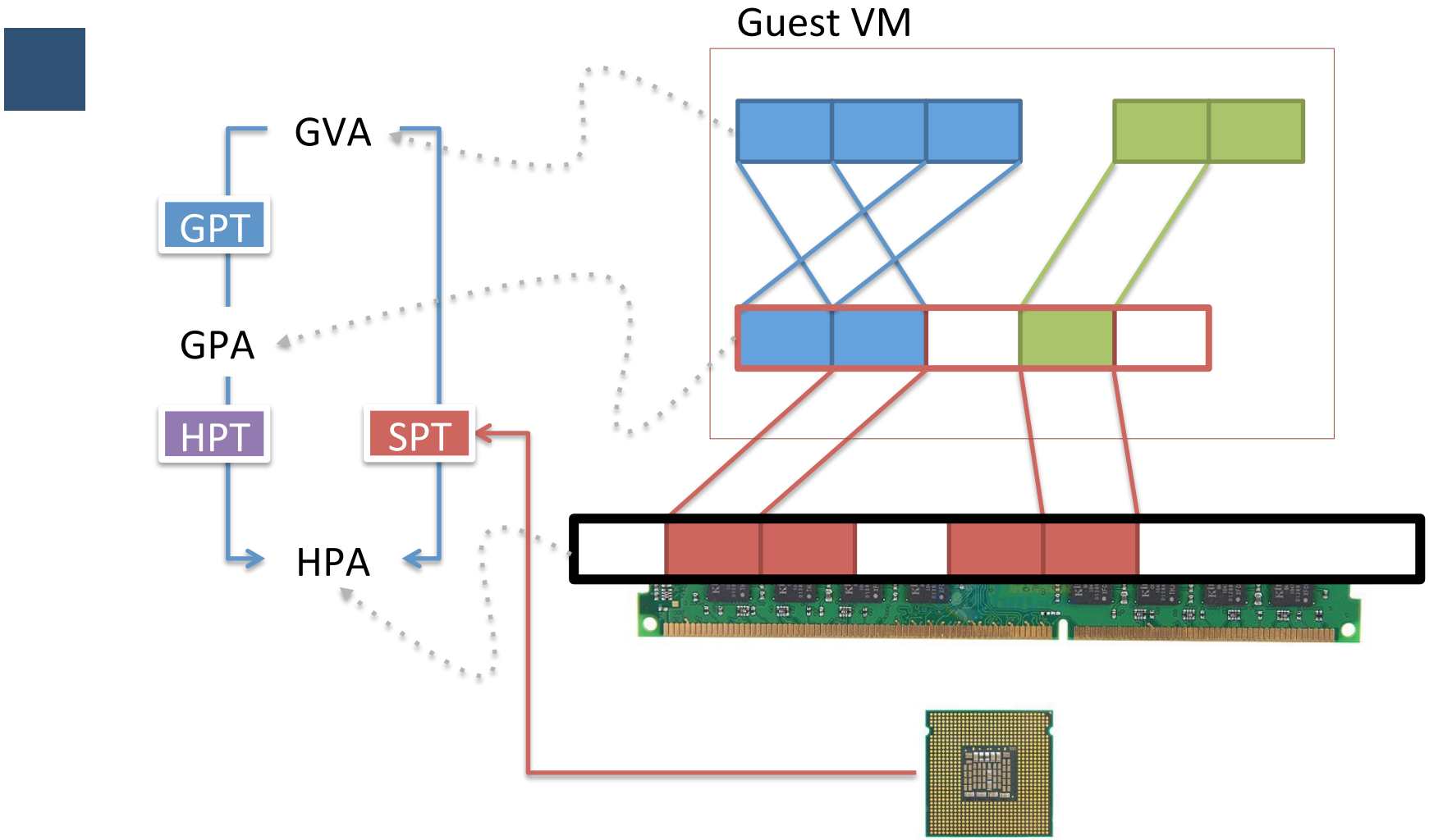
One Solution: Shadow Pages

- 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




```
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
```

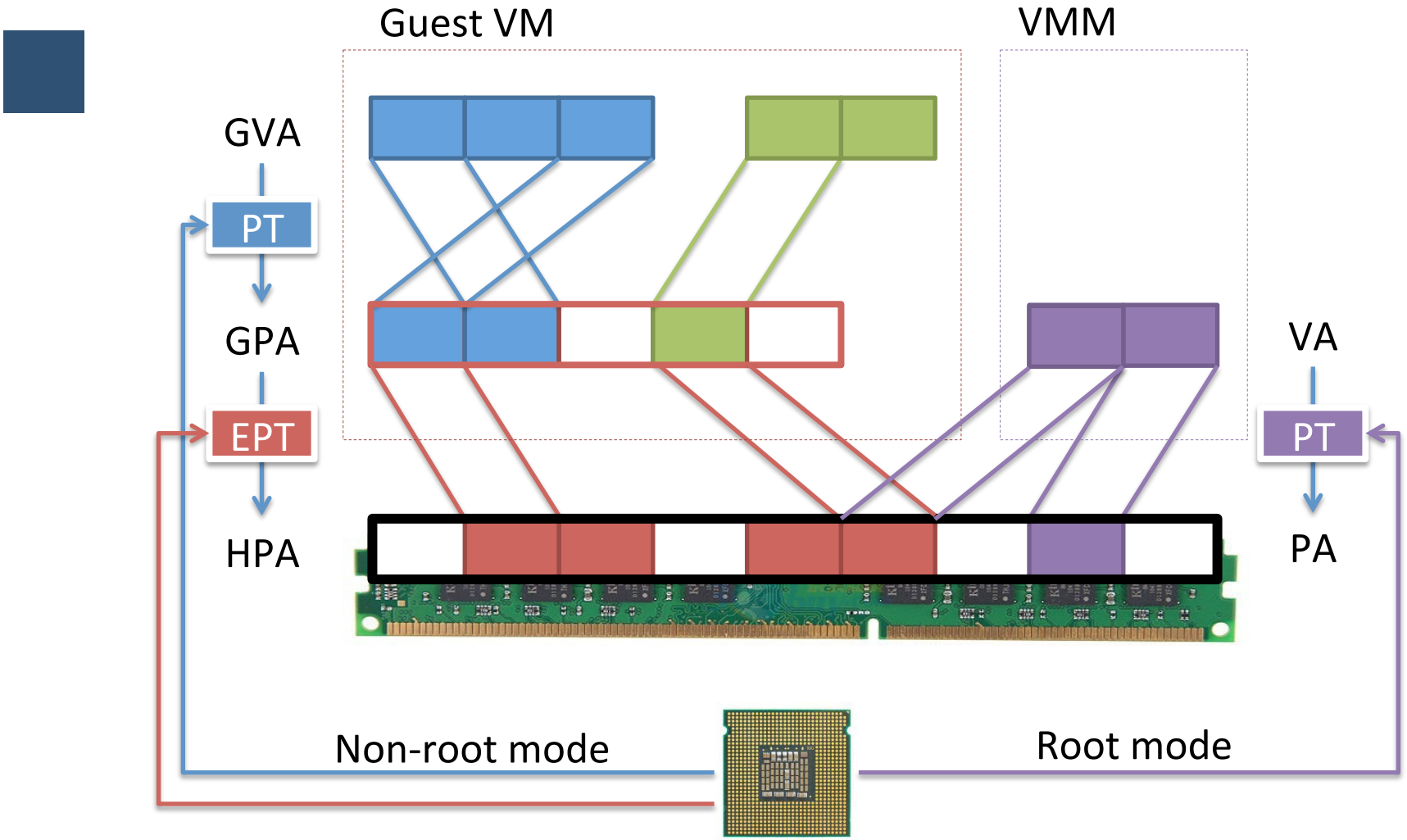

Guest VM
GVA
GPT
GPA
HPT
SPT
HPA

What if Guest OS Modifies Its Page Table?

- 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

Another Solution: Hardware Support

- 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


Guest VM
VMM
GVA
PT
GPA
EPT
HPA
VA
PT
PA
Non-root mode
Root mode

CPU VIRTUALIZATION

Virtualizing the U/K bit

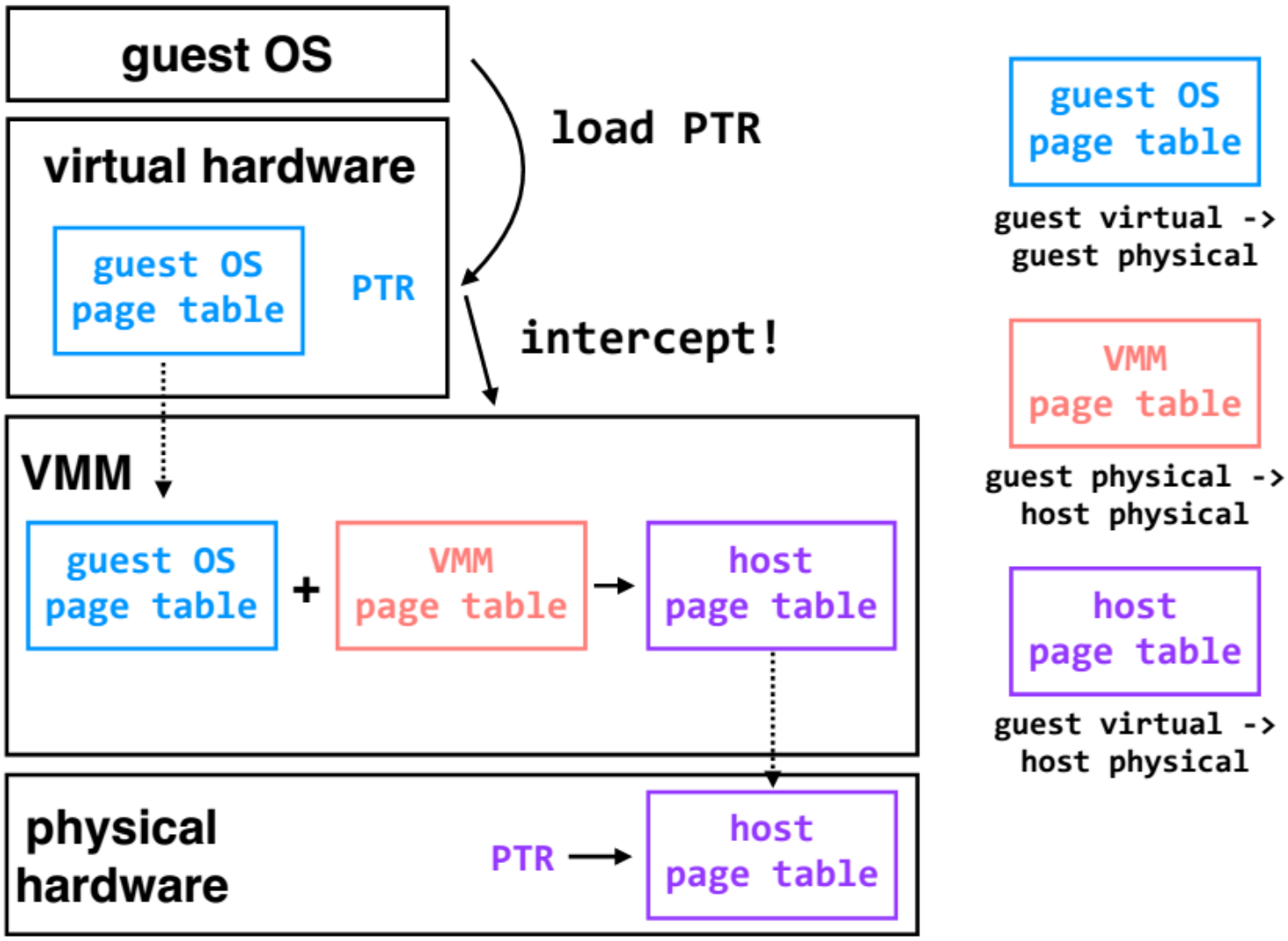
- Hardware U/K bit should be set to 'U':
  - Otherwise guest OS can do anything
- Behavior affected by the U/K bit:
  - 1. Whether privileged instructions can be executed (e.g., load PTP)
  - 2. Whether pages marked "kernel-only" in page table can be accessed

Virtual U/K Bit

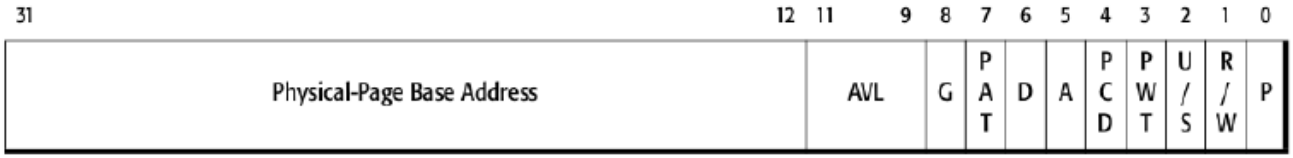
- Implementing a virtual U/K bit:
  - VMM stores the current state of guest's U/K bit (in some memory location)
  - Privileged instructions in guest code will cause an exception (U mode)
    - VMM exception handler's job will be to emulate these instructions
  - "trap and emulate"
    - E.g., if load PTP instruction runs in virtual K mode, load shadow PT. Otherwise, send an exception to the guest OS, so it can handle it

PTP Exception

```
set_ptp_exception(guest_pt):
    acquire(t_lock)
    id = cpus[CPU].thread
    if threads[id].vm_k:
        set_ptp(guest_pt)
    else:
        ## deliver exception to guest VM
    release(t_lock)
```


guest OS
load PTR
virtual hardware
guest OS
page table
PTR
intercept!
VMM
guest OS
page table
+
VMM
page table
host
page table
physical
hardware
PTR
host
page table
guest OS
page table
guest virtual ->
guest physical
VMM
page table
guest physical ->
host physical
host
page table
guest virtual ->
host physical

Virtual U/K Bit




```
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)

Sequence Coordination

Order matters

CONDITION VARIABLE

Condition Variables

- Condition Variable
  - A condition variable (cv) object names a condition that a thread can wait for
  - A waiting thread will be waked up only if the condition is ready
- API:
  - WAIT(cv, lock): go to sleep (release lock while sleeping, re-acquire later)
  - NOTIFY(cv): wake up any threads that have gone to sleep w/ same cv

Send with wait/notify (Incorrect)

```
send(bb, m):
    acquire(bb.lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            release(bb.lock)
            return
        release(bb.lock)
        yield()
        acquire(bb.lock)
```

```
send(bb, m):
    acquire(bb.lock)
    while True:
        if bb.in – bb.out < N:
            bb.buf[bb.in mod N] ← m
            bb.in ← bb.in + 1
            release(bb.lock)
            notify(bb.empty)
            return
        release(bb.lock)
        wait(bb.full)
        acquire(bb.lock)
```

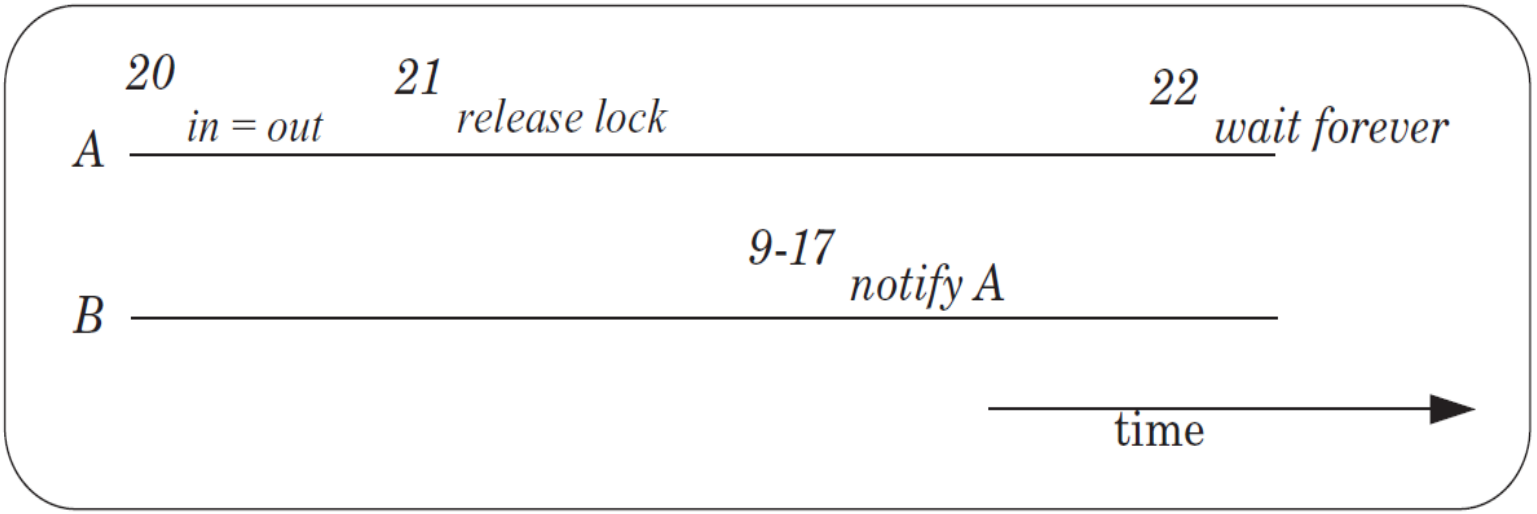
Send with wait/notify

- Do we really need a while loop with CV?
  - Strawman alternative: if buffer is full, wait() and then store message
  - Problem: many senders might wake up from wait() after just one receive()
  - Must re-check that there's still space after we re-acquired the lock
- What happens if send() notifies but there's no receiver waiting?
  - No one gets woken up, but that's OK: a later receiver will re-check (in>out)

The Lost Notify Problem

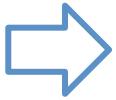
- Condition variable itself has no memory/state
  - wait() and then notify(): wait() returns
  - notify() and then wait(): wait() does not return
    - This is potentially prone to race conditions
    - The lock argument to wait() helps solve this -- will return to it later

The Lost Notify Problem



Send with wait(bb.full, bb.lock)

```
send(bb, m):
   acquire(bb.lock)
   while True:
      if bb.in – bb.out < N:
         bb.buf[bb.in mod N] ← m
         bb.in ← bb.in + 1
         release(bb.lock)
         notify(bb.empty)
         return
      release(bb.lock)
      wait(bb.full)
      acquire(bb.lock)
```



```
send(bb, m):
   acquire(bb.lock)
   while True:
      if bb.in – bb.out < N:
         bb.buf[bb.in mod N] ← m
         bb.in ← bb.in + 1
         release(bb.lock)
         notify(bb.empty)
         return
      release(bb.lock)
      yield()
      acquire(bb.lock)
      wait(bb.full, bb.lock)
```

Why Does it Works?

- Before-or-after Atomicity
  - Suppose two thread: T0 & T1
    - T0 check/wait called before OR after T1 update
  - 1. T0 check -> T0 wait -> T1 update -> T1 notify
    - Not lost
  - 2. T1 update -> T0 check
    - No wait, so no lost notify
    - In this case, T0 doesn't go to sleep because the check sees T1's update
- In practice, condition associated with data structure, which has a lock
  - This is the lock that's associated with the cv/wait

Wait

```
wait(cvar, lock):
    acquire(t_lock)
    release(lock)
    threads[id].cvar = cvar
    threads[id].state = WAITING
    yield_wait()      # will be a little different than yield
    release(t_lock)
    acquire(lock)
```

Wait and notify

```
wait(cvar, lock):
    acquire(t_lock)
    release(lock)
    threads[id].cvar = cvar
    threads[id].state = WAITING
    yield_wait()      # will be a little different than yield
    release(t_lock)
    acquire(lock)

notify(cvar):
    acquire(t_lock)
    for i = 0 to N-1:
        if threads[i].cvar == cvar && threads[i].state == WAITING:
            threads[i].state = RUNNABLE
    release(t_lock)
```

Recall: Original Yield

```
yield():
    acquire(t_lock)
    id = cpus[CPU].thread                    } suspend
    threads[id].state = RUNNABLE             } current
    threads[id].sp = SP                      } thread

    do:                                      } choose
        id = (id + 1) mod N                  } new
    while threads[id].state ≠ RUNNABLE       } thread

    threads[id].state = RUNNING              } resume
    SP = threads[id].sp                      } new
    cpus[CPU].thread = id                    } thread
    release(t_lock)
```

Yield for wait, first attempt

```
yield_wait():
    ~~acquire(t_lock)~~
    id = cpus[CPU].thread
    ~~threads[id].state = RUNNABLE~~
    threads[id].sp = SP

    do:
        id = (id + 1) mod N
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    SP = threads[id].sp
    cpus[CPU].thread = id
    ~~release(t_lock)~~
```

Yield for wait

```
yield_wait():
    id = cpus[CPU].thread
    threads[id].sp = SP
    SP = cpus[CPU].stack

    do:
        id = (id + 1) mod N
        release(t_lock)
        acquire(t_lock)
    while threads[id].state ≠ RUNNABLE

    threads[id].state = RUNNING
    SP = threads[id].sp
    cpus[CPU].thread = id
```

} switch to this CPU's kernel stack

} choose new thread, but allow other CPUs to notify()

} resume new thread

Interrupt When Yield?

- What happens if timer interrupt occurs when CPU is running yield / yield_wait?
  - Problem: t_lock already locked
    - When timer tries to call yield(), acquire(t_lock) will hang forever
  - Solution: hardware mechanism to disable interrupts
    - Disable interrupts before acquiring t_lock, re-enable after releasing it

```
yield_wait():
        id = cpus[CPU].thread
        threads[id].sp = SP
        SP = cpus[CPU].stack

        do:
                id = (id + 1) mod N
                release(t_lock)
                enable_interrupt()
                disable_interrupt()
                acquire(t_lock)
        while threads[id].state != RUNNABLE

        threads[id].state = RUNNING
        SP = threads[id].sp
        cpus[CPU].thread = id
```

Interrupt When Yield?

- What if timer interrupt comes in when yield_wait() briefly releases t_lock?
  - Interrupts not masked because yield_wait() released t_lock
  - Can't call yield: we would put the wrong thread to sleep!
  - Solution:
    - First set cpus[].thread to null in yield_wait()
    - Don't call yield if cpus[].thread is null in interrupt handler

```
yield_wait():
        id = cpus[CPU].thread
        cpus[CPU].thread = null
        threads[id].sp = SP
        SP = cpus[CPU].stack

        do:
                id = (id + 1) mod N
                release(t_lock)
                enable_interrupt()
                disable_interrupt()
                acquire(t_lock)
        while threads[id].state != RUNNABLE

        threads[id].state = RUNNING
        SP = threads[id].sp
        cpus[CPU].thread = id
```

Summary for Condition Variable

- Previous Solution: Polling
  - Spinning
  - Periodically yield() and recheck
- Condition Variable
  - Wait for the condition to be ready
  - Still need loop to recheck

EVENTCOUNT & SEQUENCER

Single Sender and Single Receiver

```
procedure SEND (buffer reference p, message
    AWAIT (p.out, p.in − N)
    p.message[READ(p.in) modulo N] ← msg
    ADVANCE (p.in)

procedure RECEIVE (buffer reference p)
    AWAIT (p.in, p.out)
    msg ← p.message[READ(p.out) modulo N]
    ADVANCE (p.out)
    return msg
```

4 Primitives

- AWAIT (eventcount, value)
- ADVANCE (eventcount)
- TICKET (sequencer)
- READ (eventcount or sequencer)

Primitives for Sequence Coordination

- AWAIT (eventcount, value) is a before-or-after action
  - Compares *eventcount* to *value*
  - If *eventcount* exceeds *value*
    - AWAIT returns to its caller.
  - If *eventcount* is less than or equal to *value*
    - Changes the state of the calling thread to WAITING
    - Places *value* and the name of *eventcount* in this thread's entry in the thread table
    - Yields its processor

Primitives for Sequence Coordination

- ADVANCE (eventcount) is a before-or-after action
  - Increments *eventcount* by one
  - Searches the thread table for threads that are waiting on this *eventcount*
  - For each one it finds,
    - If *eventcount* now exceeds the *value* for which that thread is waiting
    - Changes that thread's state to RUNNABLE

Primitives for Sequence Coordination

- TICKET (sequencer) is a before-or-after action
  - Returns a non-negative value that increases by one on each call
  - Two threads concurrently calling TICKET on the same sequencer
    - Receive different values
    - The ordering of the values returned corresponds to the time ordering of the execution of TICKET

Primitives for Sequence Coordination

- READ (eventcount or sequencer) is a before-or-after action
  - Returns to the caller the current value of the variable
  - Having an explicit READ procedure is to assure before-or-after atomicity for *eventcounts* and sequencers whose value may grow to be larger than a memory cell

Multiple Senders and Single Receiver

```
procedure SEND (buffer reference p, message
    AWAIT (p.out, p.in − N)
    p.message[READ(p.in) modulo N] ← msg
    ADVANCE (p.in)

procedure RECEIVE (buffer reference p)
    AWAIT (p.in, p.out)
    msg ← p.message[READ(p.out) modulo N]
    ADVANCE (p.out)
    return msg
```

```
procedure SEND (buffer reference p, message
    t ← TICKET (p.sender)
    AWAIT (p.in, t)
    AWAIT (p.out, READ(p.in) − N)
    p.message[READ(p.in) modulo N] ← msg
    ADVANCE (p.in)
```

No loop! Is it OK?

Implementation

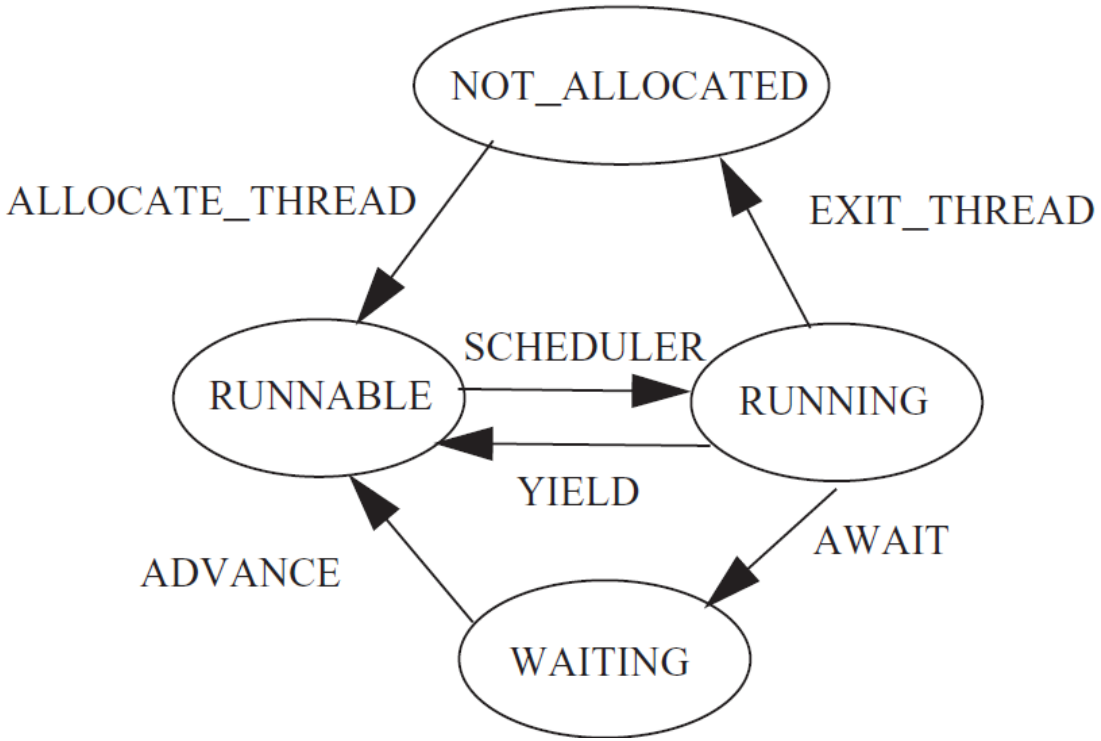
```
procedure AWAIT (eventcount reference event, value)
    ACQUIRE (thread_table_lock)
    id ← GET_THREAD_ID ()
    thread_table[id].event ← event
    thread_table[id].value ← value
    if event.count ≤ value then thread_table[id].state ← WAITING
    ENTER_PROCESSOR_LAYER (id, CPUID)
    RELEASE (thread_table_lock)
```

```
procedure ADVANCE (eventcount reference event)
    ACQUIRE (thread_table_lock)
    event.count ← event.count + 1
    for i from 0 until 7 do
        if thread_table[i].state = WAITING and thread_table[i].event = event and
                                        event.count > thread_table[i].value then
            thread_table[i].state ← RUNNABLE
    RELEASE (thread_table_lock)
```

```
procedure TICKET (sequencer reference s)
    ACQUIRE (thread_table_lock)
    t ← s.ticket
    s.ticket ← t + 1
    RELEASE (thread_table_lock)
    return t
```

```
procedure READ (eventcount reference event)
    ACQUIRE (thread_table_lock)
    e ← event.count
    RELEASE (thread_table_lock)
    return e
```

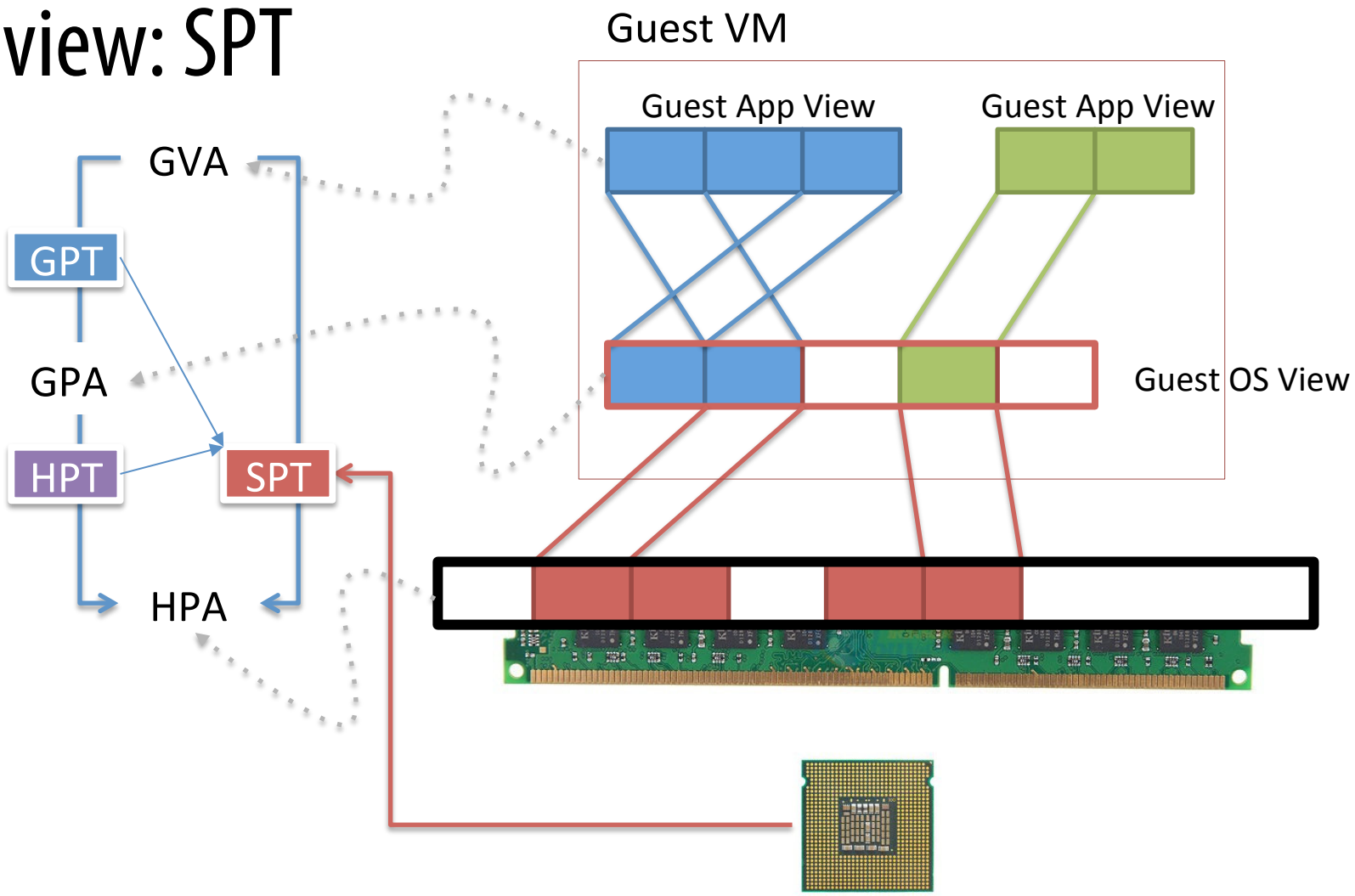
Thread State Diagram



System Performance

Design for performance

Review: SPT


Guest VM
Guest App View
Guest App View
GVA
GPT
GPA
HPT
SPT
HPA
Guest OS View

Enforcing Modularity via Virtualization

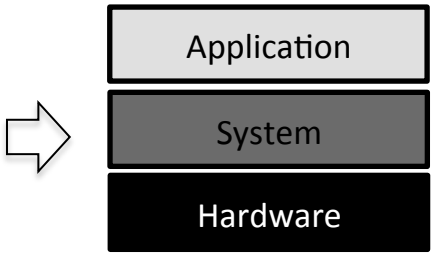
in order to enforce modularity + build an effective operating system

programs shouldn't be able to refer to (and corrupt) each others' memory	→	Virtual memory
programs should be able to communicate	→	Bounded buffer (virtualize communication links)
programs should be able to share a CPU without one program halting the progress of the other	→	Thread (virtualize processors)

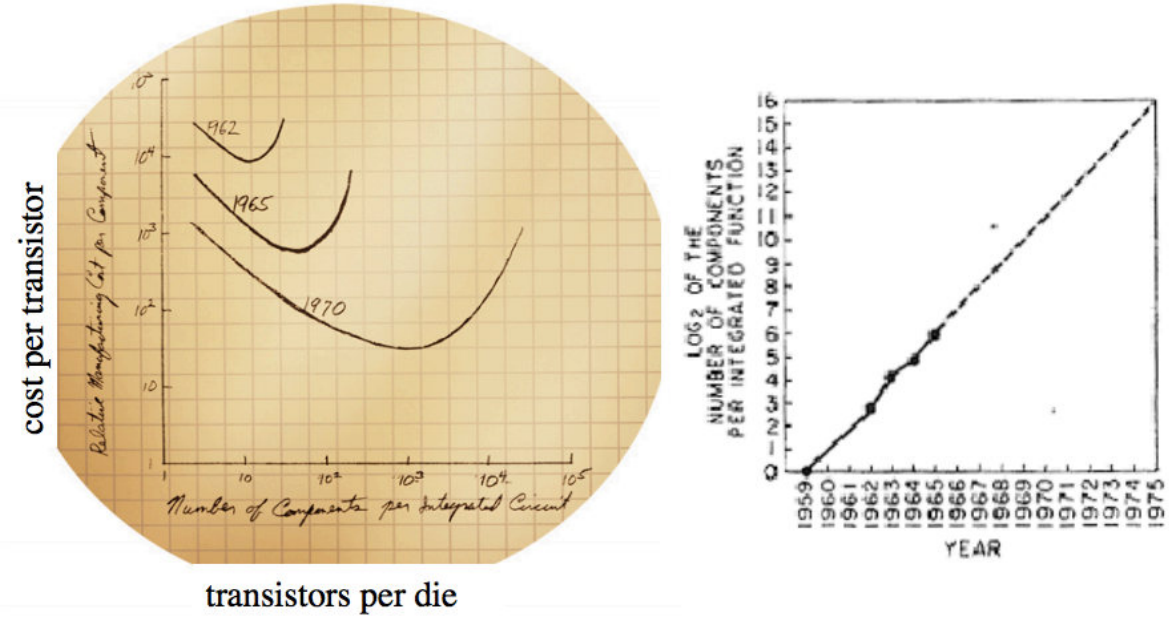
today: can we get systems to not just work, but to work well?

Improving Performance

- Get faster hardware
- Fix application: better algorithm, fewer features (not in CSE)
- General system optimization techniques:
  - Batching
  - Caching
  - Concurrency
  - Scheduling



Moore's Law



"Cramming More Components Onto Integrated Circuits", *Electronics*, April 1965

Performance Metrics

Computer Systems can be viewed as interconnected modules

★ Capacity
★ Utilization
★ Latency
★ Throughput

Performance Metrics: Capacity

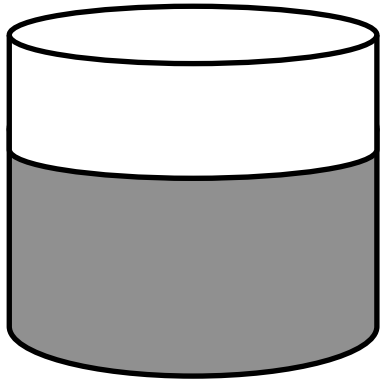
Capacity – Size or amount of a service

Blocks on storage or memory devices

Cycles on a processor

Requests per second for a web server

Bits per second on a network link



Performance Metrics: Utilization

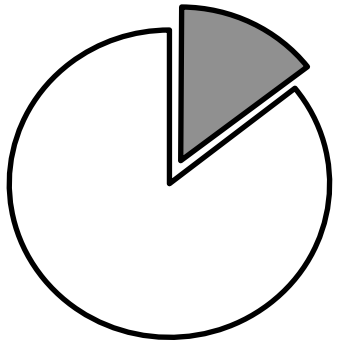
Utilization – Fraction of capacity used by a workload

CPU is 50% busy, rest in idle loop

75% of disk blocks are allocated

Webserver is running at 20% of capacity

20% of the time network is busy



Performance Metrics: Latency

The time it takes from start to finish

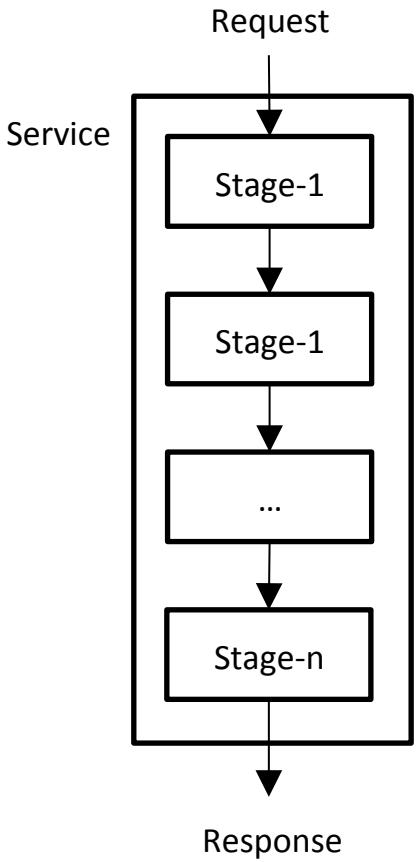
Webserver – Time from request to response

Network – Time from send() until receive() done

Procedure – Time from call until return

For a multi-step process (Steps A + B)

$Latency_{A+B} \geq LatencyA + Latency_{B}$



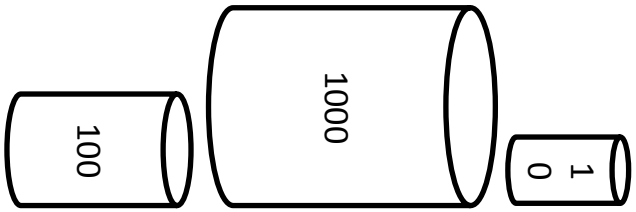
Performance Metrics: Throughput

Rate at which work is done

Webserver – Requests per second

Network – Bytes per second

Procedure – Calls per second



Pipeline of modules

$\text{Thoughput}_{A+B} \leq \text{minimum}(\text{Throughput}_A, \text{Throughput}_B)$

Bottleneck

Relationship between Latency & Throughput

When processing is serial:

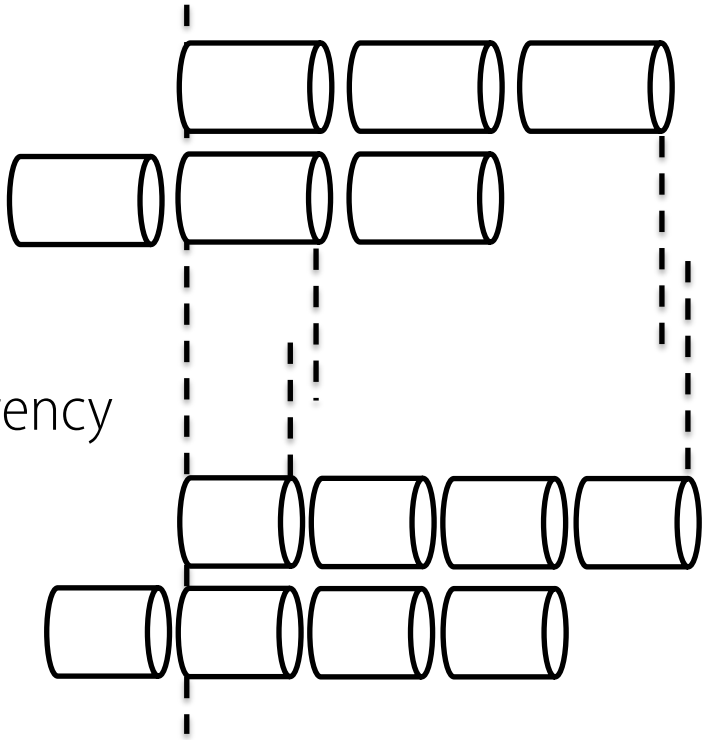
Throughput = 1 / (Latency)

When processing is parallel:

Throughput has *no* direct relationship with latency

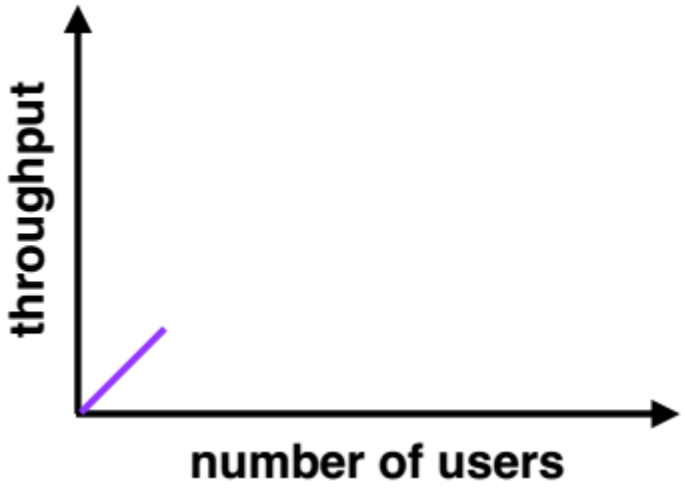
How to improve throughput?

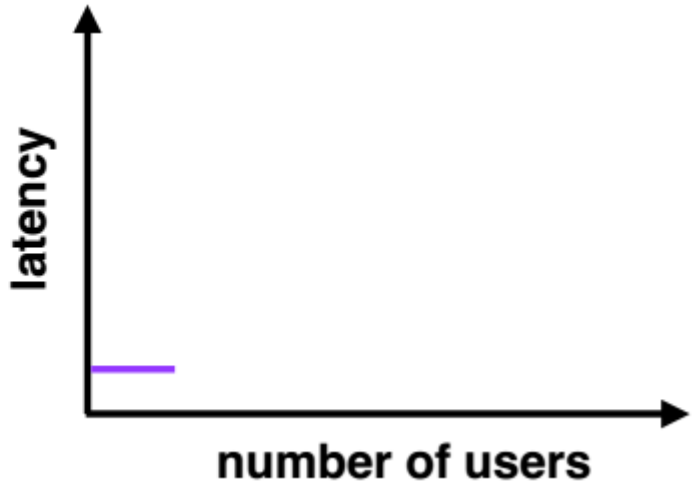
Reduce latency & Increase paralleling



Throughput and Latency

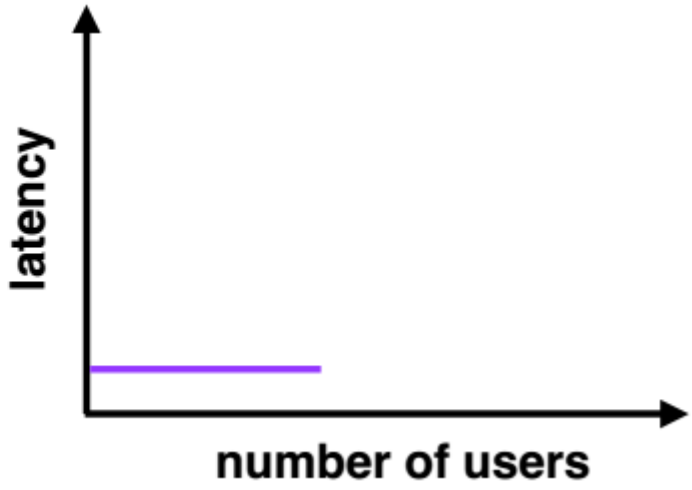
- Few users
  - Low latency
  - Low throughput (few users = few requests)





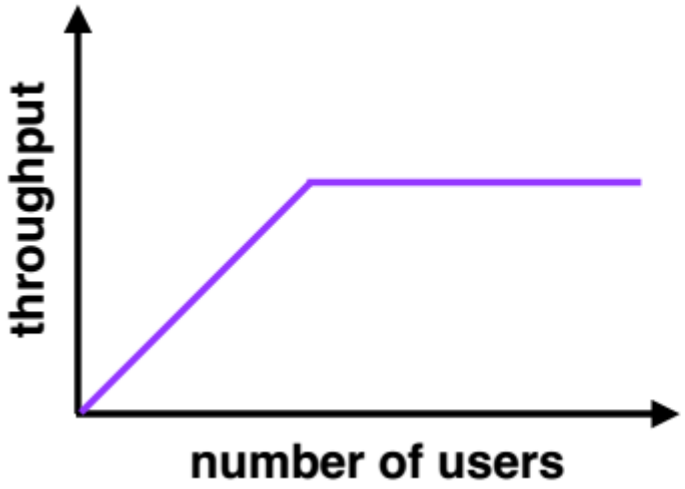
Throughput and Latency

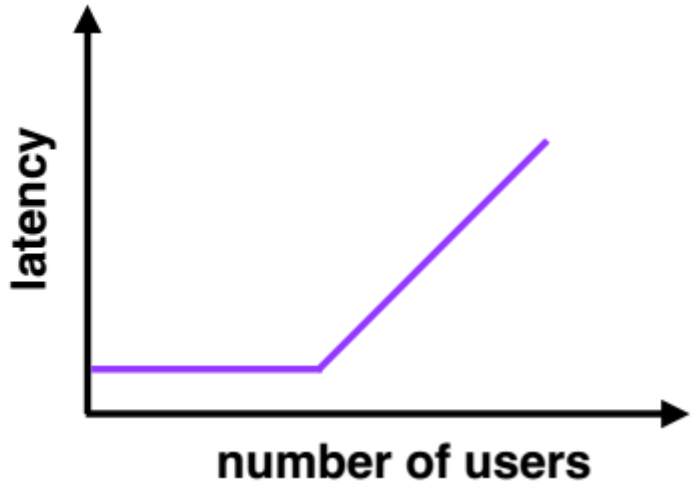
- Moderate users
  - Low latency (new users consume previously idle resources)
  - High throughput (more users = more requests)



Throughput and Latency

- Many users
  - High latency (requests queue up)
  - Throughput plateaus (cannot serve requests any faster)



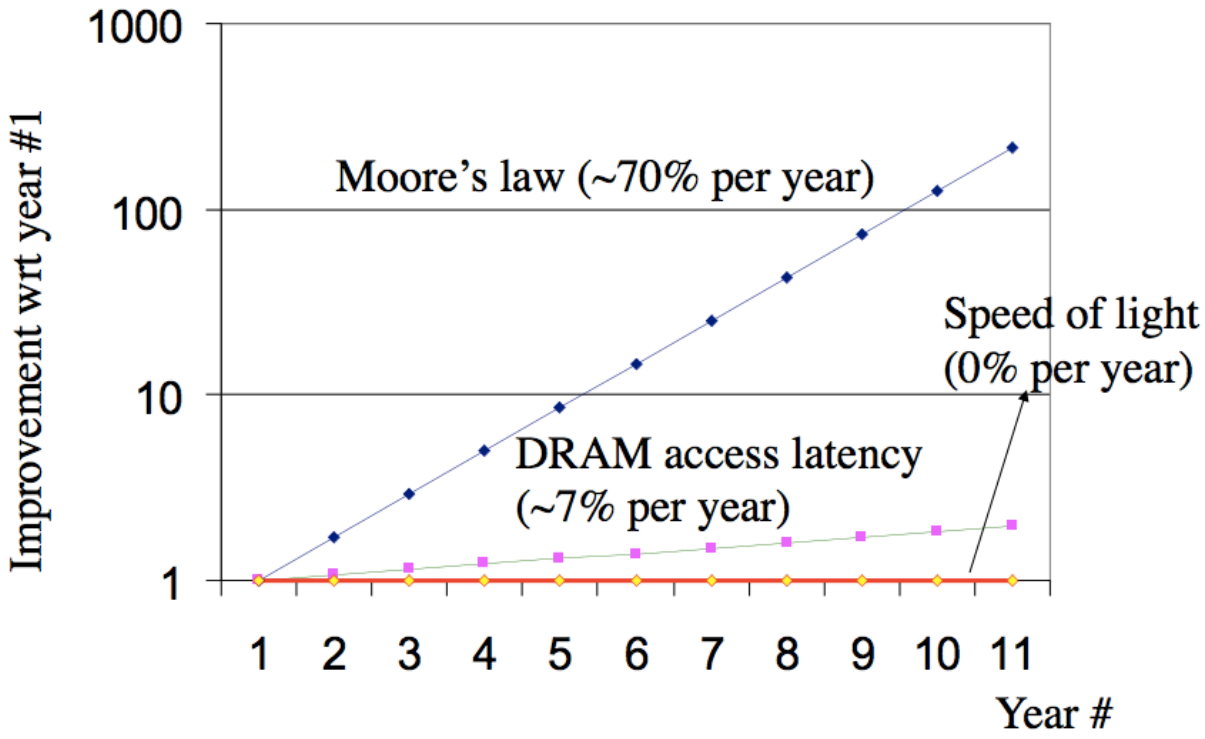


Improving Bottlenecks

- Latency is the harder of the problems
  - Frequently fundamental limits
  - Speed of CPU
  - Speed of communication link (e.g., light)

- Throughput frequently is a resource limit
  - Example: Buy a fatter pipe but it costs more
  - Famous saying: You can buy throughput but you can't bribe god

Latency Improves Slowly

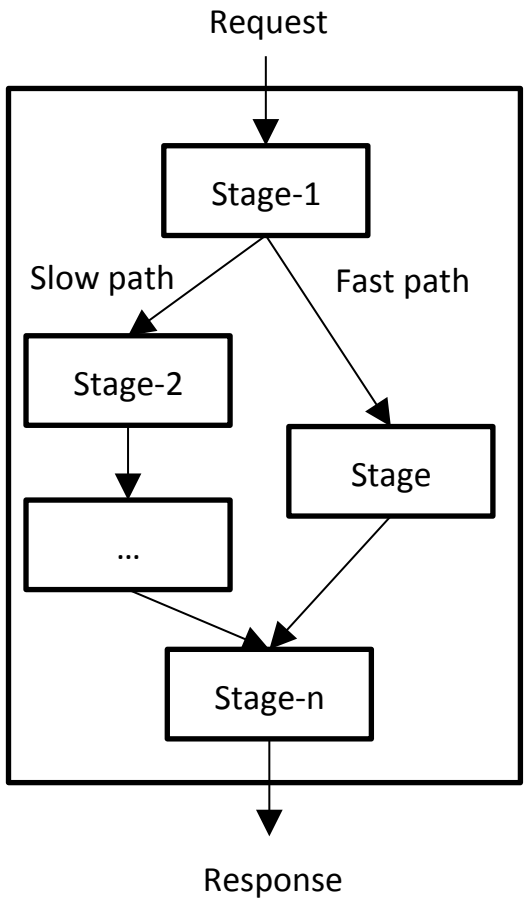


Fast Path for Latency Optimizations

- Use knowledge of the workload
- Make the common cases faster

```
if (WorksOnFastPath(request))
        DoFastCase(request);
Else
        DoNormalCase(request);
```

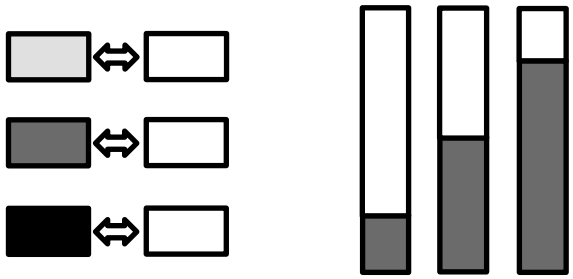
- E.g., cache hit VS. cache miss



Fast Path for Latency Optimizations

- AverageLatency = Frequencyfast x Latencyfast + Frequencyslow x Latencyslow

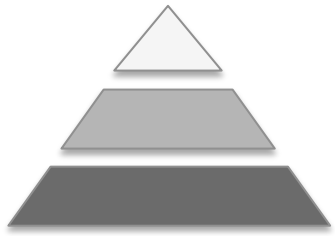
- Is introducing a fast path worth the effort?
  - depends on the difference of latency,
  - and the frequency using fast path,
  - which is dependent on the workload



- Many workloads don't have a uniform distribution of requests,
  - thus introducing a fast path works well

Caching: Classic Fast Path Optimization

- Use knowledge of workload: memory abstractions
  - Keep common requests in a fast local memory
  - Check the local memory every access



- CPU cache – Cache 1ns, Memory 100ns, 90% hit rate
  - AverageLatency = 0.9 x 1ns + 0.1 x 100ns = 10.9 ns (100ns -> 10.9ns)

- Used everywhere there are memory abstractions
  - Processors, file systems, TLBs, browsers, DNS, etc.

Another Fast Path Example

- Consider a request stream of different types
- Some are very simple and some are very complex

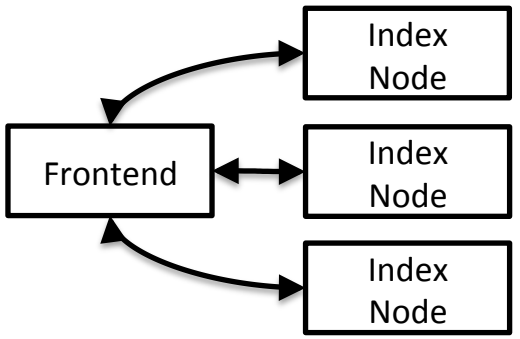
- Optimization:

```
if (FastPath(request)) {
      Handle simple cases in highly optimized code
} else {
      Handle request in full generality
}
```

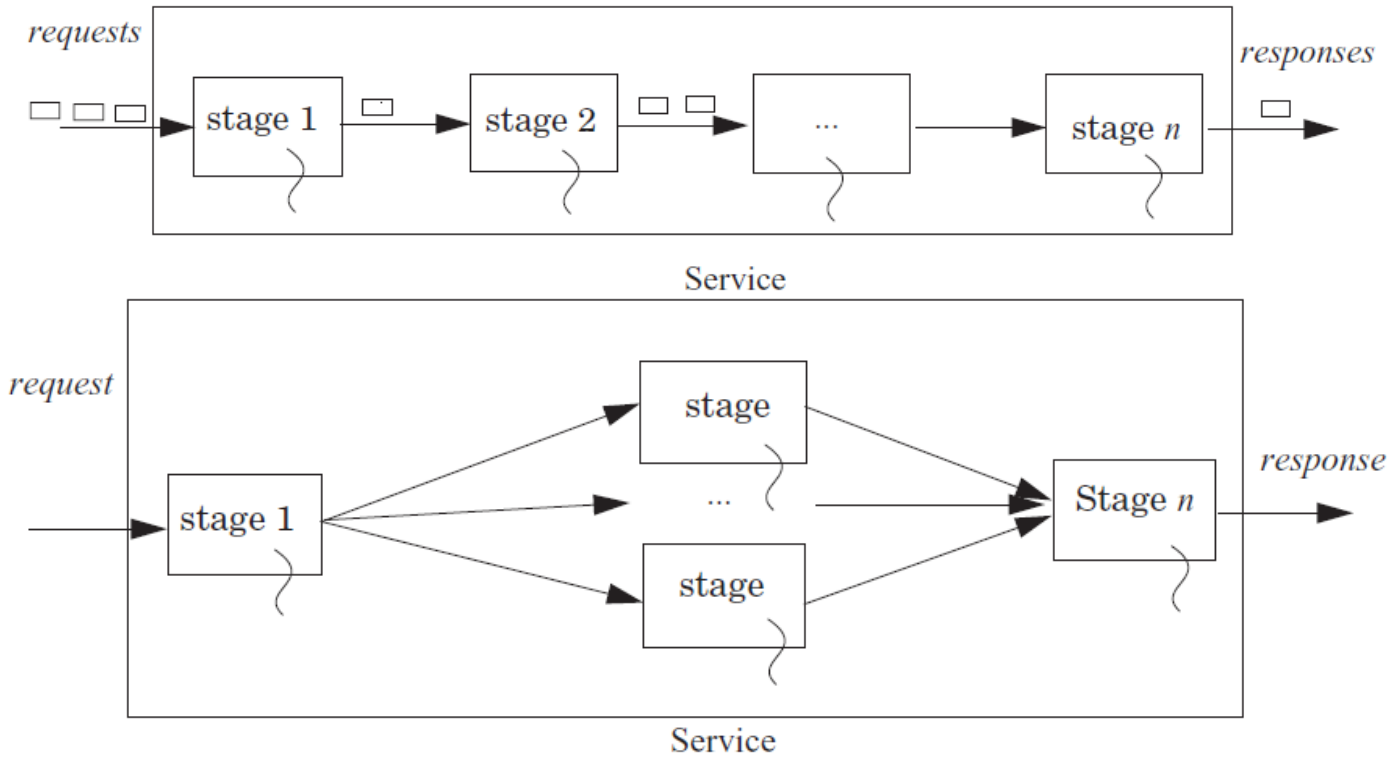
Reducing Latency using Concurrency

- Example: Google search engine
  - Splits the index of the Web up in *n* pieces, each piece on a machine
  - Frontend sends copies of requests and combines the response

- If subtasks are independent to each other
  - Can use n threads to get a speedup of n
  - Hard to get the ideal speedup
    - 10 months a man -> 1 month 10 men?

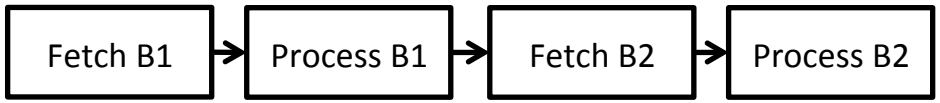


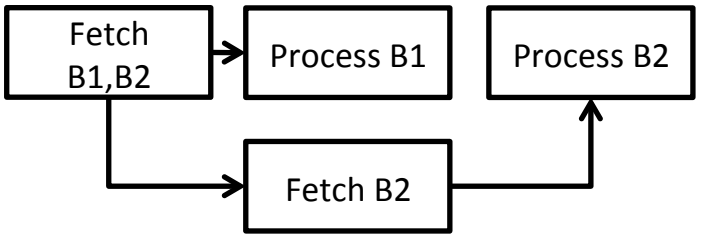
Using Concurrency to Improve Throughput



Hide Latency by Overlapping

- Fetch (5 time units) and process (5 units) 2 blocks (20 units)
  - Fetch B1; Process B1; Fetch B2; Process B2;
- Fetch 2 blocks asynchronously (15 units)
  - Send request for B1,B2;
  - Receive B1; Process B1; Receive B2; Process B2
- Processors prefetching memory requests
  - File systems prefetching files
  - Browser prefetching pages





Queuing and Overload

- Overload: whenever the offered load is greater than the capacity of a service for some duration
  - E.g., 7 threads on 3 processors, must wait in queue (RUNNABLE)

- The queuing theory
  - Unit is the average service time
  - The waiting time in a queue: $1/(1-\rho)$, ρ is the utilization
  - Once the utilization approaches 1, the delay will grow without bound

Queuing and Overload

- One solution
  - Make the capacity match the offered load of requests
- Example:
  - CPU: one instruction per ns; memory: respond takes 10 ns
  - CPU must make a memory request 10 ns in advance
  - Memory must serve 10 requests concurrently
    - If half instructions are memory request, then only 5 concurrently
    - But cannot predicate the pattern of memory access

Queuing and Overload

- If overload appears short period of time
  - A queue handles short bursts of too much demand by time-averaging with adjacent periods when there is excess capacity
- If overload persists over long periods of time
  - Increase the capacity of the system
  - Shed load: reduce or limit the offered load

Queuing and Overload

- Offered load: the rate of arrival of requests for service
- Using bounded buffer to control offered load
  - Self managing, if the a source needs the results of the output
  - If source makes no request at all, then offered load decreases
  - If source holds the request and resubmits it later, offered load doesn't decrease
  - Put a quota on the source (e.g., max threads per application)
- Reducing the offered load when a stage becomes overloaded: e.g., swapping

FIGHTING BOTTLENECKS

Why Performance Bottleneck?

1. Physical limitation
   - Speed of light
   - The capacity of memory
   - …
2. Sharing
   - Several users share a device
   - Several clients share a server

How to Improve Performance?

1. Measure the system to find the bottleneck
2. Relax the bottleneck
   - Batch requests
   - Cache data
   - Exploit concurrency
   - Exploit parallelism

Batching

- Frequently it is possible to group requests for more efficiency
  - Amortize overheads of processing
  - Schedule requests for better performance

- Examples: N messages vs. one message with N requests Disks:
  - Always transfer a bunch of bytes (sector or block)
  - Schedule the disk motions

Dallying

- Procrastination sometimes helps
  - Temp files – deleted before written to disk

- Examples:
  - Caches – Write back policy: Write absorption
  - Database transactions – group commit
    - Might hold your ATM request until others arrive to do batching

Speculation

- Guessing at an operation and performing it ahead a time
  - Need to be able to undo changes if wrong

- Examples:
  - Processors guess at branches, memory access
  - File systems guess at next file block needed (*prefetching*)
  - Easier if space of possibilities is small
  - Branch prediction versus value prediction

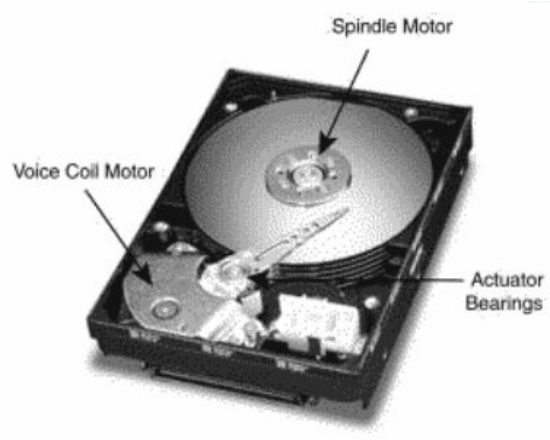
Challenges

- Batching, dallying, and speculation introduce complexity
  - They introduce concurrency
  - Coordination is difficult to get right

CASE: I/O BOTTLENECK

The I/O Bottleneck

- Bits read from a disk encounter two potential transfer rate limits
  - The rate at which bits spin under the heads on their way to a buffer
  - The rate at which the I/O channel or I/O bus can transfer the contents



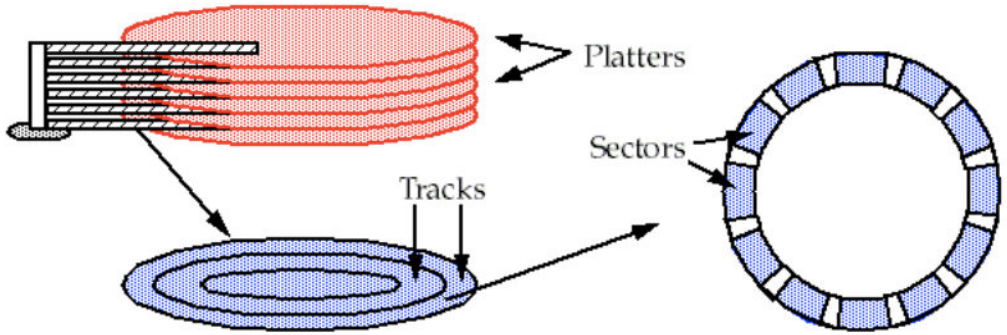
The I/O Bottleneck

- Average seek latency: about 8 milliseconds
  - Time to move the head over 1/3 of the disk

- The average rotational latency: 4.17 milliseconds
  - The disks spin at 7200 rotations/minute
  - One rotation every 8.33 milliseconds
  - On average, the disk has to wait a half rotation for the desired block to be under the disk head

A Typical Modern 400 Gigabyte Disk

- Has 16,383 cylinders, or about 24 megabytes per cylinder
- Would have 8 two-sided platters and thus 16 read/write heads
- Would be 24/16 = 1.5 megabytes per track
- When rotating at 7200 revolutions per minute
- The bits will go by a head at 180 megabytes per second



Bottleneck

- The IDE bus: 66 MB/S is common
- The Serial ATA-3 bus: 6 Gbps

- With IDE bus: The electronics would be the bottleneck
  - at 66 MB/S
- With SATA-3 bus: The mechanics would be the bottleneck
  - at 180 MB/S

Latency of 4-KB Data Accessing (Using IDE)

- The latency of reading a 4 kilobyte block chosen at random:
- *avg seek time + avg rotation latency + transmission of 4 kilobytes*
- $= 8 + 4.17 + (4 / (66 \times 1024)) \times 1000$ milliseconds
- $= 8 + 4.17 + 0.06$ milliseconds
- $= 12.23$ milliseconds

- The throughput for reading randomly-chosen blocks one by one is:
- $= 1000/12.23 \times 4$ kilobytes per second
- $= 327$ kilobytes/second

Round-1: No Optimization

```
1     in ← OPEN ("in", READ)                  // open "in" for reading
2     out ← OPEN ("out", WRITE)  // open "out" for reading
3
4     while not ENDOFFILE (in) do
5               block ← READ (in, 4096)     // read 4 kilobyte block from in
6               block ← COMPUTE (block)     // compute for 1 millisecond
7               WRITE (out, block, 4096)    // write 4 kilobyte block to out
8     CLOSE (in)
9     CLOSE (out)
```

- *reading 4 kilobyte + 1 millisecond of computation + writing 4 kilobyte*
- = 12.23 + 1+ 12.23 milliseconds
- = 25.46 milliseconds

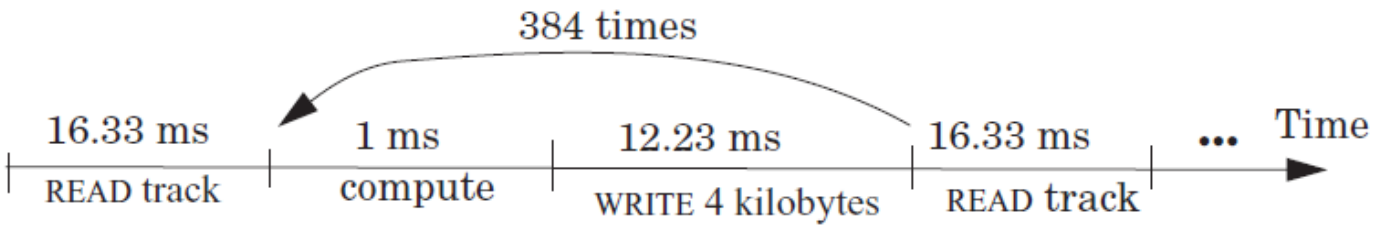
Round-2

- Modify the file system
  - to layout the blocks of a file contiguously
  - to *prefetch* an entire track of data on each read

- *Average seek time + 1 rotational delay*
- = 8 + 8.33 milliseconds = 16.33 milliseconds

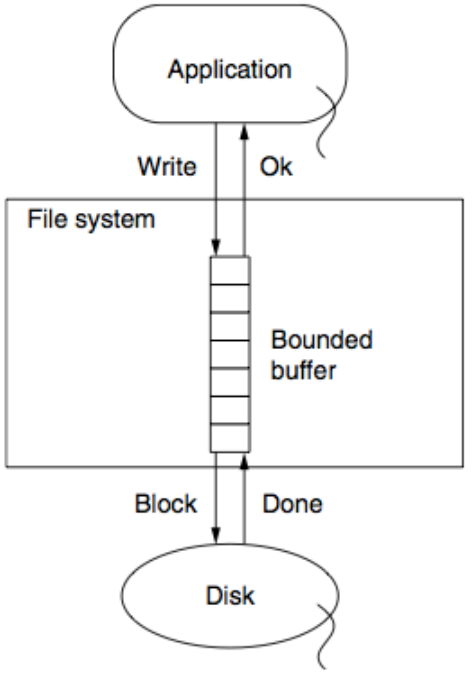
Round-2

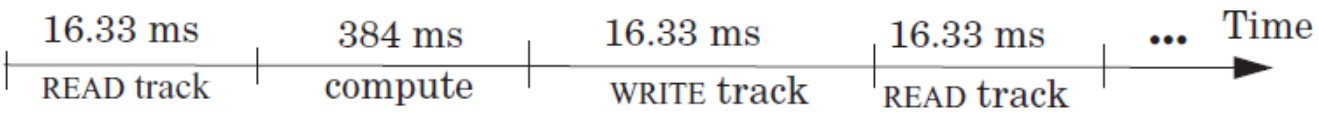
- File system issues 1 read request per 384 (1.5 MB per track / 4 KB) loop iterations
- The average time for 384 iterations is:
- *reading 1536 kilobyte + 384× (1 millisecond of computation + writing 4 kilobyte)*
- = 16.33 + 384 × (1 + 12.23) millisecond
- = 16.33 + 5080.32 milliseconds
- = 5096.65 milliseconds.
- Thus, the average time for a loop iteration is 5096.65/384 = 13.27 milliseconds



Round-3

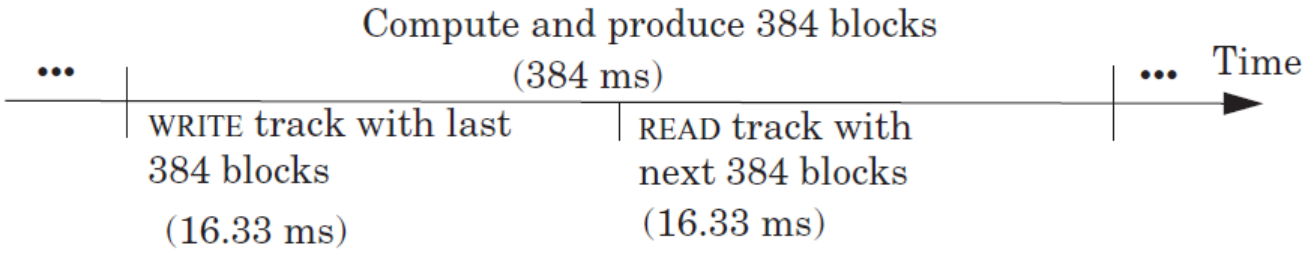
- Improve by dallying & batching write requests
- Write to buffer in RAM and flush when buffer is full
- Latency = (16.33 + 384 + 16.33)/384 milliseconds
- = 1.09 milliseconds





Round-4

- Prefetch the next track before the 385th READ
- Overlap computation and I/O completely
- The average time around the loop is 1 millisecond
- The bottleneck now is the CPU



Computer System Engineering, Spring 2015. (IPADS, SJTU)

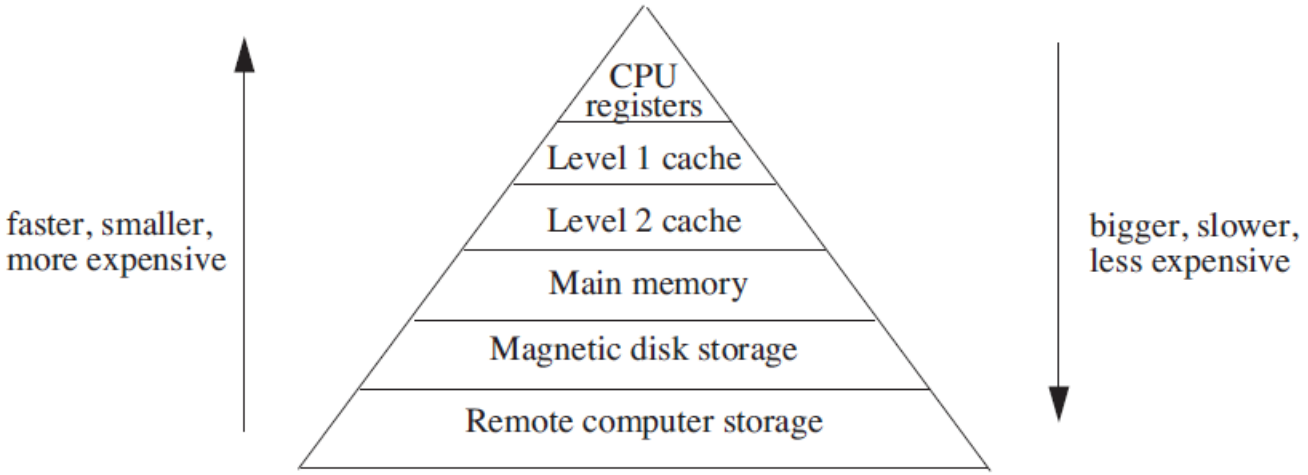
Performance

Multi-level Memory

CACHE POLICIES

Analyzing Multilevel Memory Systems

- Only consider the adjacent pair of levels
  - View it as a two level multilevel memory system
  - The first one is primary device, the second is secondary device



Locality of Reference and Working Sets

- All information items stored in the memory must not have equal frequency of use

$$AverageLatency = R_{hit} \times Latency_{primary} + R_{miss} \times Latency_{Secondary}$$

- If accesses to every cell of the primary and secondary devices were of equal frequency,

$$AverageLatency = \frac{S_{primary}}{S_{primary} + S_{secondary}} \times T_{primary} + \frac{S_{secondary}}{S_{primary} + S_{secondary}} \times T_{secondary}$$

- The primary device is L2 cache with 1 nanosecond latency and the secondary device is main memory with 10 nanoseconds latency
  - 0.99 + 0.10 = 1.09 nanoseconds

Locality of Reference and Working Sets

- In many situations most memory references are to a small set of addresses for significant periods of time
- As the application progresses, the area of concentration of access shifts, but its size still typically remains small
- Called "locality of reference"
  - Temporal locality and spatial locality
  - *Thrashing*: repeated movement of data back and forth between two levels

Multilevel Memory Management Policies

- Each level of multilevel memory system can be characterized by 4 items
  - The string of references directed to that level
  - The bring-in policy for that level
  - The removal policy for that level
  - The capacity of the level

Page-removal Policies

- First-in, first-out (FIFO) page-removal policy
  - Remove the page that has been in the primary device the longest
- Belady's anomaly
  - Performance drops with a larger primary device capacity!
- Optimal (OPT) page-removal policy
  - Choose for removal the page that will not be needed for the longest time
- Least-recently-used (LRU) page-removal policy
  - The page in the primary device that has not been used for the longest time

First-in, First-out (FIFO) Page-Removal Policy

- The page for removal is the one that has been in the primary device the longest

time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device contents	- - -	0 - -	0 1 -	0 1 2	3 1 2	3 0 2	3 0 1	4 0 1	4 0 1	4 0 1	4 2 1	4 2 3	Pages brought in
remove	-	-	-	0	1	2	3	-	-	0	1	-	
bring in	0	1	2	3	0	1	4	-	-	2	3	-	9

Belady's Anomaly

- Increase of missing-page exception numbers with a larger primary device capacity

time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device contents	- - - -	0 - - -	0 1 - -	0 1 2 -	0 1 2 3	0 1 2 3	0 1 2 3	4 1 2 3	4 0 2 3	4 0 1 3	4 0 1 2	3 0 1 2	Pages brought in
remove	-	-	-	-	-	-	0	1	2	3	4	0	
bring in	0	1	2	3	-	-	4	0	1	2	3	4	10

OPT (Optimal) Page-Removal Policy

- Always choose for removal the page that will not be needed for the longest time

time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device contents	- - -	0 - -	0 1 -	0 1 2	0 1 3	0 1 3	0 1 3	0 1 4	0 1 4	0 1 4	2 1 4	3 1 4	Pages brought in
remove	-	-	-	2	-	-	3	-	-	0	2	-	
bring in	0	1	2	3	-	-	4	-	-	2	3	-	7

time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
primary device contents	- - - -	0 - - -	0 1 - -	0 1 2 -	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 4	0 1 2 4	0 1 2 4	0 1 2 4	3 1 2 4	Pages brought in
remove	-	-	-	-	-	-	3	-	-	-	0	-	
bring in	0	1	2	3	-	-	4	-	-	-	3	-	6

LRU

Table 6.5 The LRU Page-Removal Policy with a Three-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	0	3	Pages brought in
	–	–	1	1	2	1	1	1	1	1	1	1	
	–	–	–	2	3	3	3	4	4	4	2	2	
Remove	–	–	–	1	–	2	3	–	–	4	0	1	
Bring in	0	1	2	3	–	1	4	–	–	2	3	4	9

Table 6.6 The LRU Page-Removal Policy with a Four-Page Primary Device

Time	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Reference string	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**	
Primary device contents	–	0	0	0	0	0	0	0	0	0	0	0	Pages brought in
	–	–	1	1	1	1	1	1	1	1	1	1	
	–	–	–	2	2	2	2	4	2	2	2	2	
	–	–	–	–	3	3	3	3	4	4	4	3	
Remove	–	–	–	–	–	–	2	–	–	–	4	0	
Bring in	0	1	2	3	–	–	4	–	–	–	3	4	8

Least-recently-used (LRU) Page Removal Policy

- A program that runs from top to bottom
- the virtual memory that is larger than primary device
- LRU: always evicts exactly the wrong page

Reference string	**0**	**1**	**2**	**3**	**4**	**0**	**1**	**2**	**3**	**4**	**0**	**1**	**2**
4-page primary device	F	F	F	F	F	F	F	F	F	F	F	F	F

Reference string	**0**	**1**	**2**	**3**	**4**	**0**	**1**	**2**	**3**	**4**	**0**	**1**	**2**
4-page primary memory	F	F	F	F	F				F				F

Comparative Analysis of Different Policies

- Ways of deciding the following two things:
  - How large the primary memory device should be?
  - Which page removal policy to use?

- Collecting traces of the reference strings of typical programs that are to be run

- Simulate the multilevel memory manager with:
  - Different configurations - Size of primary device
  - Page removal policy - Traces of memory access

Stack Algorithms

- Subset property
  - For the optimal policy, at all times
  - the pages it keeps in the 3-page memory
  - is a subset of that it keeps in the 4-page memory
- No Belady's anomaly if subset property holds
  - At all times and
  - For every possible capacity of primary device
  - It creates a total ordering for pages at a given time

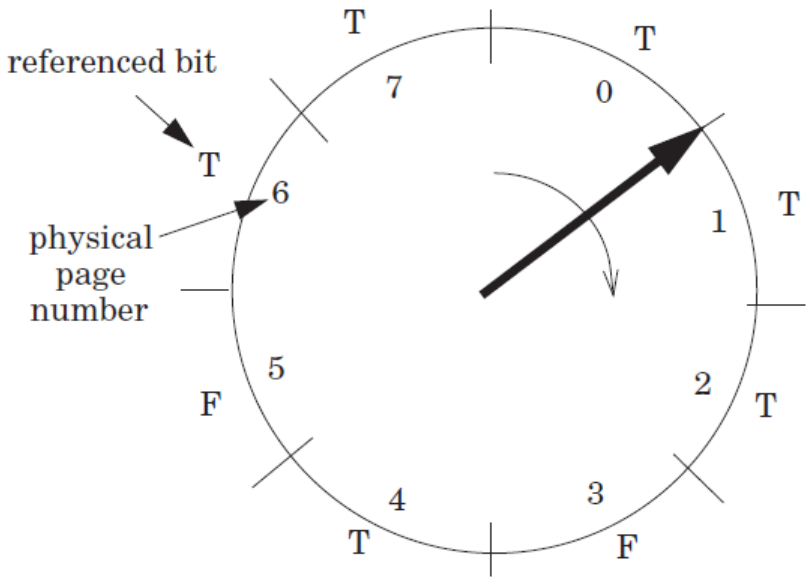
Stack Algorithms for LRU

- It can perform a simulation for all possible primary memory size with a single pass through a given reference string

time	1	2	3	4	5	6	7	8	9	10	11	12	
reference string	0	1	2	3	0	1	4	0	1	2	3	4	
stack contents after reference	0 - - - -	1 0 - - -	2 1 0 - -	3 2 1 0 -	0 3 2 1 -	1 0 3 2 -	4 1 0 3 2	0 4 1 3 2	1 0 4 3 2	2 1 0 4 3	3 2 1 0 4	4 3 2 1 0	number of moves in
size1 in/out	0/-	1/0	2/1	3/2	0/3	1/0	4/1	0/4	1/0	2/1	3/2	4/3	12
size 2 in/out	0/-	1/-	2/0	3/1	0/2	1/3	4/0	0/1	1/4	2/0	3/1	4/2	12
size 3 in/out	0/-	1/-	2/-	3/0	0/1	1/2	4/3	-/-	-/-	2/4	3/0	4/1	10
size 4 in/out	0/-	1/-	2/-	3/-	-/-	-/-	4/2	-/-	-/-	2/3	3/4	4/0	8
size 5 in/out	0/-	1/-	2/-	3/-	-/-	-/-	4/-	-/-	-/-	-/-	-/-	-/-	5

Efficiency of Page-Removal Policies

- Clock page-removal algorithm
  - Base on a hardware setting bit
  - Reference bit
  - Move clockwise
  - If T, set F
  - Choose as the victim otherwise
- Random removal policy
  - For TLB
- Directed mapping

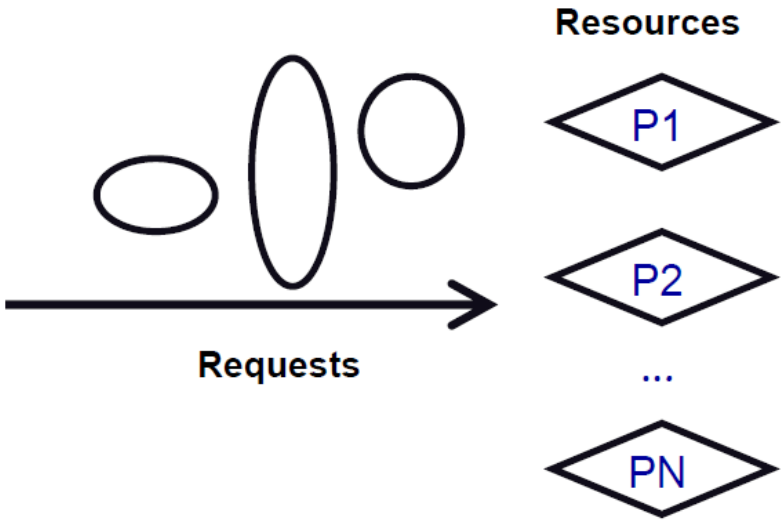


SCHEDULING

Scheduling

- Algorithm that assigns requests to resources

- Processor time – Threads
- Physical memory – Address spaces
- Printers – Printer jobs
- Disks – Disk requests
- Networks – Packets
- Memory bus – Memory requests

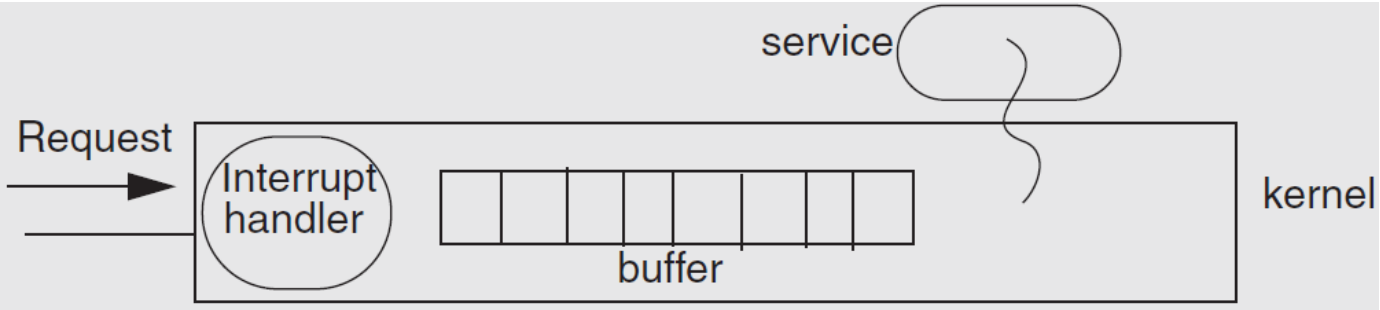


Challenges of Scheduling

- Easy – if number of requests $\leq$ resources
- Lack of information
  - Packet scheduling and streaming media
- Lack of mechanism to enforce policies
  - Give high priority to CPU scheduler, but if can't get to the disk or network …
- Getting mechanism right
  - Many schedulers break rather than gracefully degrade under load

Example: Livelock under Overload

- Web news server overloaded
  - Most of the time is spend to handle interrupt and drop requests
  - No progress at all!



The Ideal Scheduler

- Mini latency: How long a service takes? E.g., Move mouse, cursor moves
- Max throughput: Max jobs / time. E.g., Web server pages/second
- Low overheads: Scheduler doesn't use any resources
- Fairness: everyone gets to make progress, no one starves
- Scales linearly up to capacity: Graceful handling of overload

- System-level goal may conflict with the needs of individual threads
  - System: minimal preemption
  - Apps: finish ASAP

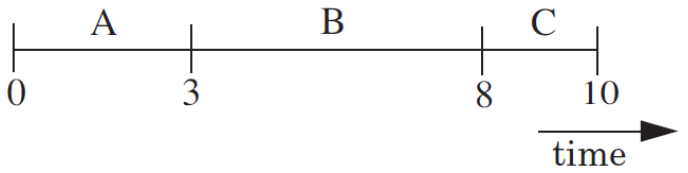
Measuring the Request's Response

- Turn-around time: - The length of time from when a request arrives at a service until it completes

- Response time: The length of time from when a request arrives at a service until it starts producing output

- Waiting time: The length of time from when a request arrives at a service until the service starts processing the request

First Come First Server (FCFS)



- Fairness paramount
- Good for printer Scheduler

Job	Arrival time	Amount of work
A	0	3
B	1	5
C	3	2

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Wait time till job is done
A	0	3	0	3	0	3
B	1	5	3	8	2	7
C	3	2	8	10	5	7
Total waiting					7	

First Come First Server (FCFS)

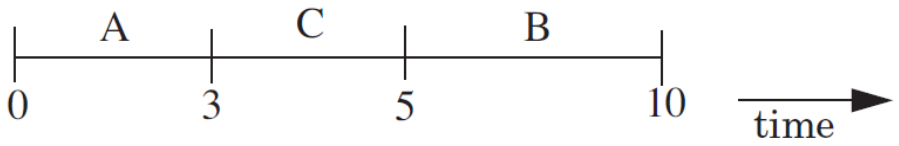
- For the processor
  - One thread that periodically waits for I/O but mostly computes
  - Several threads that perform mostly I/O operations

- For the scheduler
  - It runs the I/O-bound threads first
  - Then runs the computation intensive thread
  - All quickly finished computations and waiting for I/O

Convoy Effect

- When I/O devices are idle
  - The I/O-bound threads will quickly finish computations and start I/O operations
  - The processor-bound thread will run for a long time
  - I/O-bound operations will finish I/O operation and queuing up for computation
  - All the I/O devices will become idle!
- When processor is idle
  - The processor-bound thread finishes computation and starts I/O operation
  - The I/O-bound threads will quickly finish computations and start I/O operations
  - Processor will become idle!

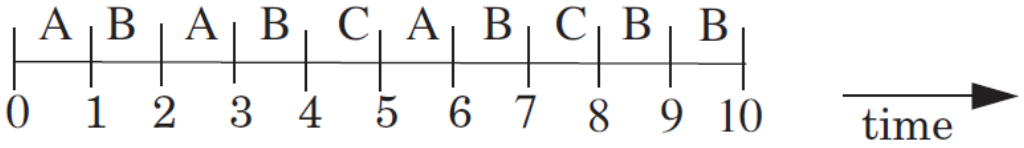
Shortest-Job-First



- Chooses the job that has the shortest expected run time
- Require a prediction of the running time of a job before running
- Starvation: several short jobs makes a long job starve

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Waiting time till job is done
A	0	3	0	3	0	3
B	1	5	5	10	4	9
C	3	2	3	5	0	2
			Total waiting		4	

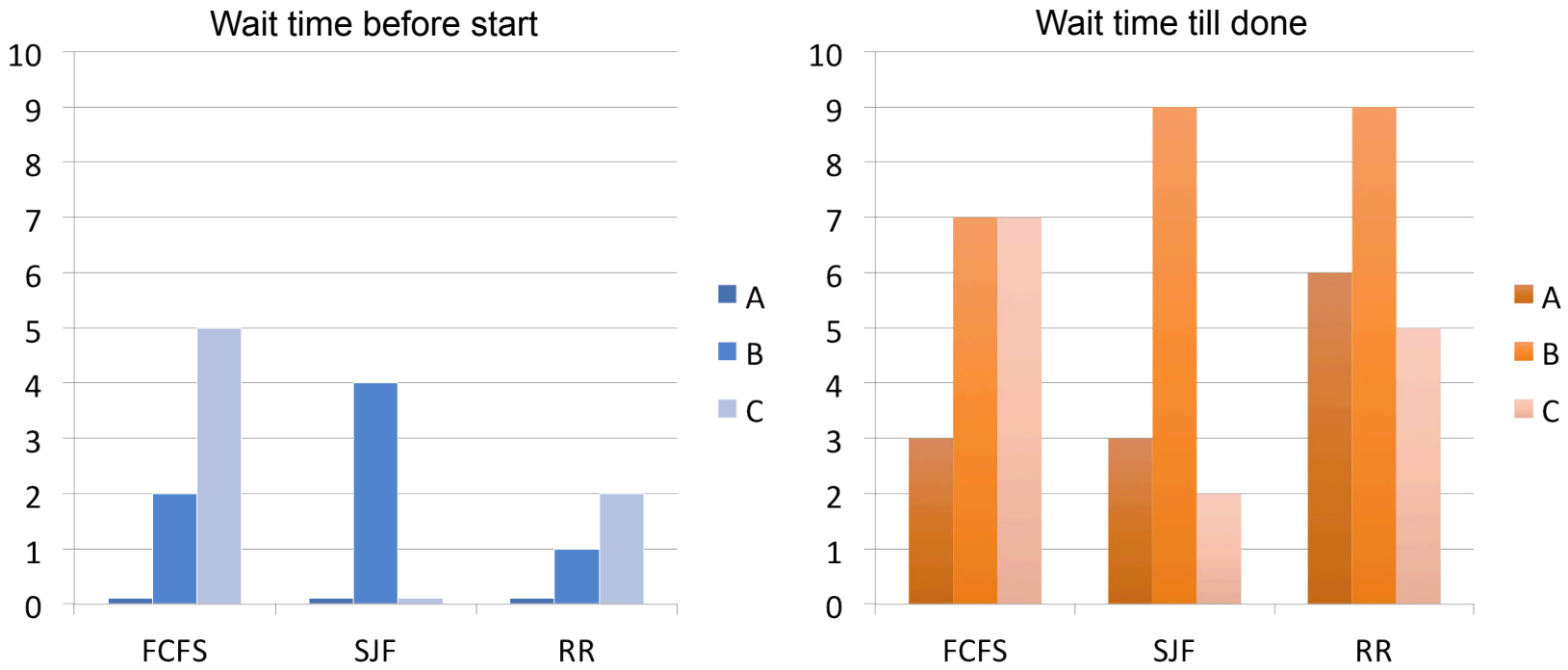
Round-Robin



- Select the first job from queue as in the first-come first-serve policy
- Stop the job after some period of time, and select a new job

Job	Arrival time	Amount of work	Start time	Finish time	Waiting time till job starts	Waiting time till job is done
A	0	3	0	6	0	6
B	1	5	1	10	1	9
C	3	2	5	8	2	5
				Total waiting	3	

Comparing the Three Policies



Priority Scheduling Policy

- Priority Scheduling Policy
  - Assign each job a priority number (static vs. dynamic)
  - Select the job with the highest priority number
  - Must have some rule to break ties
- Modern CPU schedulers have
  - Priorities: Run job with the highest priority
  - Round Robin: Alternate among jobs of same priority, preempt a job if it holds CPU too long

CPU Scheduling Policy

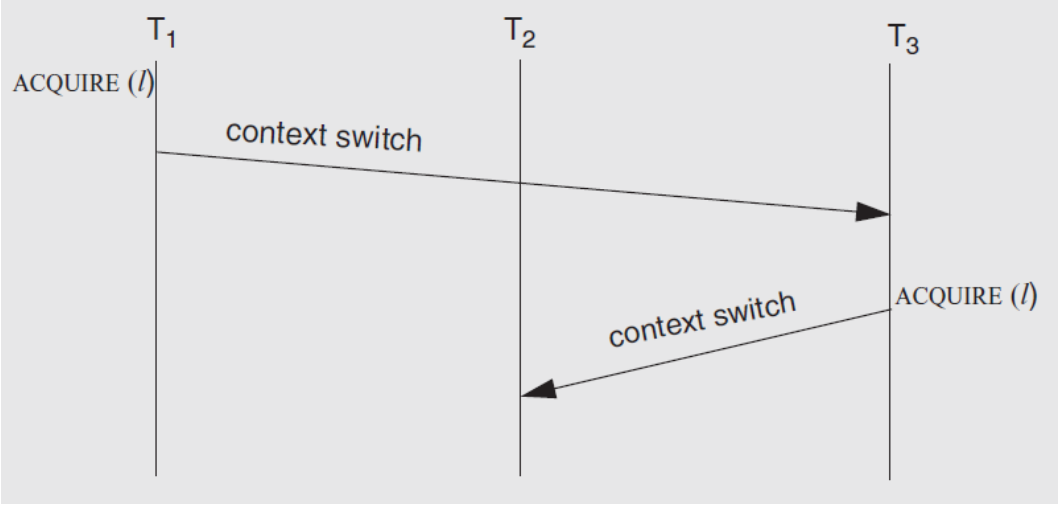
- Assign priorities to jobs
  - Give high priorities to I/O-bound jobs
  - Give low priorities to CPU-bound jobs
  - Handles I/O-bound and CPU-bound jobs nicely

CPU Scheduling Policy

- Challenge: distinguish I/O-bound and CPU-bound
- Look at past behavior to predict the future
  - Linux: Priority of job is function of amount of CPU it has used
- Counterintuitive:
  - More CPU a job uses/needs, the lower its priority
  - The less CPU it uses, the higher its priority
  - Avoids starvation problems: Not running gives you a higher priority

The Priority Inversion Problem

- Problem: what if a high priority thread waits for a low priority one which holding a lock?
- Solution: priority inheritance



Real-Time Scheduling

- Processes are not time insensitive
- Missed deadline = incorrect behavior

- Soft real time: Display video frame every 30th of sec
- Hard real time: "apply-breaks" process in your car

- Scheduling more than one thing: memory, network bandwidth, CPU all at once

Earliest Deadline First (EDF)

- Keep the queue of jobs sorted by deadline
- Run the first of the queue
- Minimize the total lateness of all the jobs

$$\left(\sum_{i=1}^{n} \frac{C_i}{P_i} \right) \leq 1$$

- How to know deadline?
  - Specified by tasks themselves
  - In the form of (C, P), C for second, P for period
  - The scheduler rejects new task if sum(Ci/Pi) > 1

Disk Scheduling Goals

- Minimize time spent waiting on disk mechanisms
  - Moving the disk arm
  - Waiting for sector to rotate under disk heads

- Fairness
  - Should not make requests wait too long

Disk Scheduling Goals

Request	Movement	Time
Seek 1	0 → 0	0*t*
Seek 2	0 → 90	90*t*
Seek 3	90 → 5	85*t*
Seek 4	5 → 100	95*t*
Total		270*t*

Request	Movement	Time
Seek 1	0 → 0	0*t*
Seek 2	0 → 5	5*t*
Seek 3	5 → 90	85*t*
Seek 4	90 → 100	10*t*
Total		100*t*

Disk Scheduling Goals

Request	Movement	Time
Seek 1	0 → 0	0*t*
Seek 2	0 → 5	5*t*
Seek 3	5 → 1	4*t*
Seek 4	1 → 90	89*t*
Seek 5	90 → 100	10*t*
Total		108*t*

Shortest-Seek-First

Request	Movement	Time
Seek 1	0 → 0	0*t*
Seek 2	0 → 5	5*t*
Seek 3	5 → 90	85*t*
Seek 4	90 → 100	10*t*
Seek 5	100 → 1	99*t*
Total		199*t*

Elevator Algorithm

Computer System Engineering, Spring 2015. (IPADS, SJTU)

Introduction to Network

As a component, and as a system itself

Why Learn About Networks?

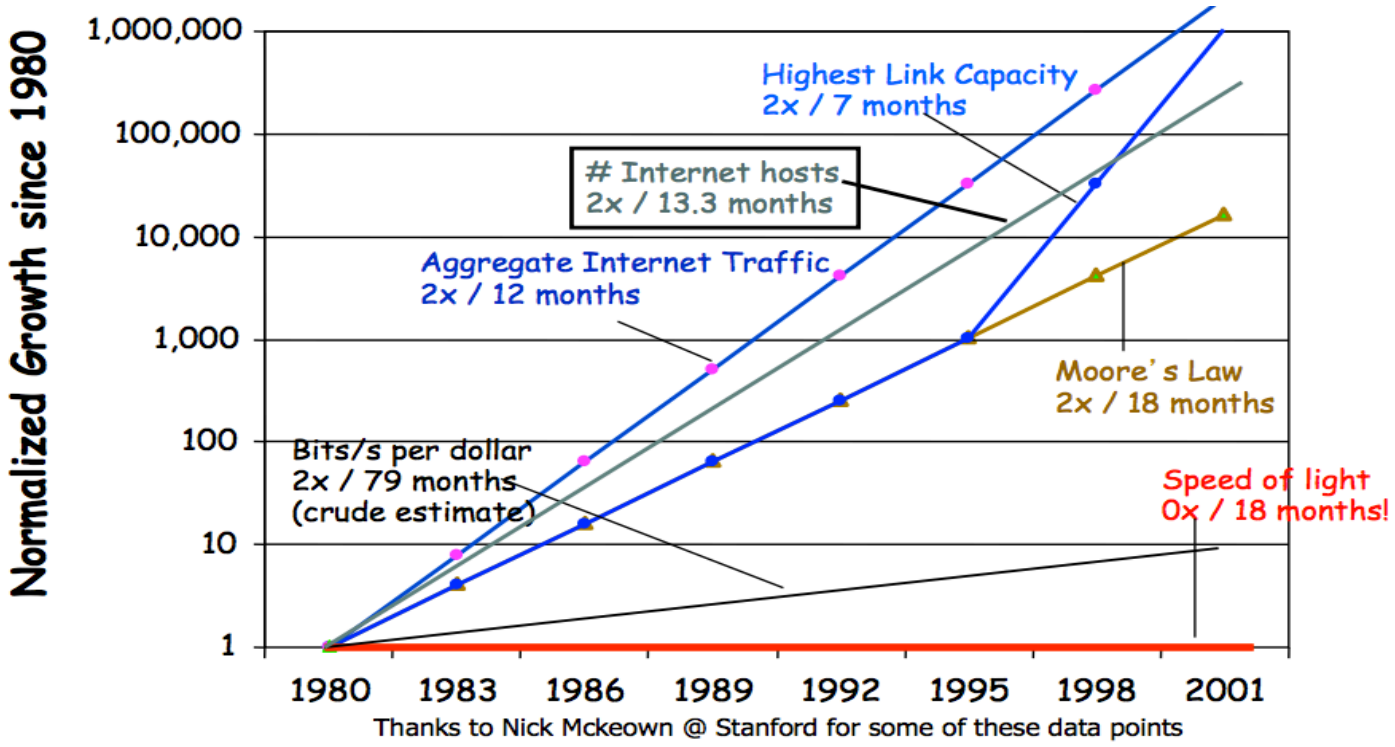
- Many computer systems use the network
- The Internet is an interesting example of a successful system

Why do We Need Network?

- We need network to transfer data

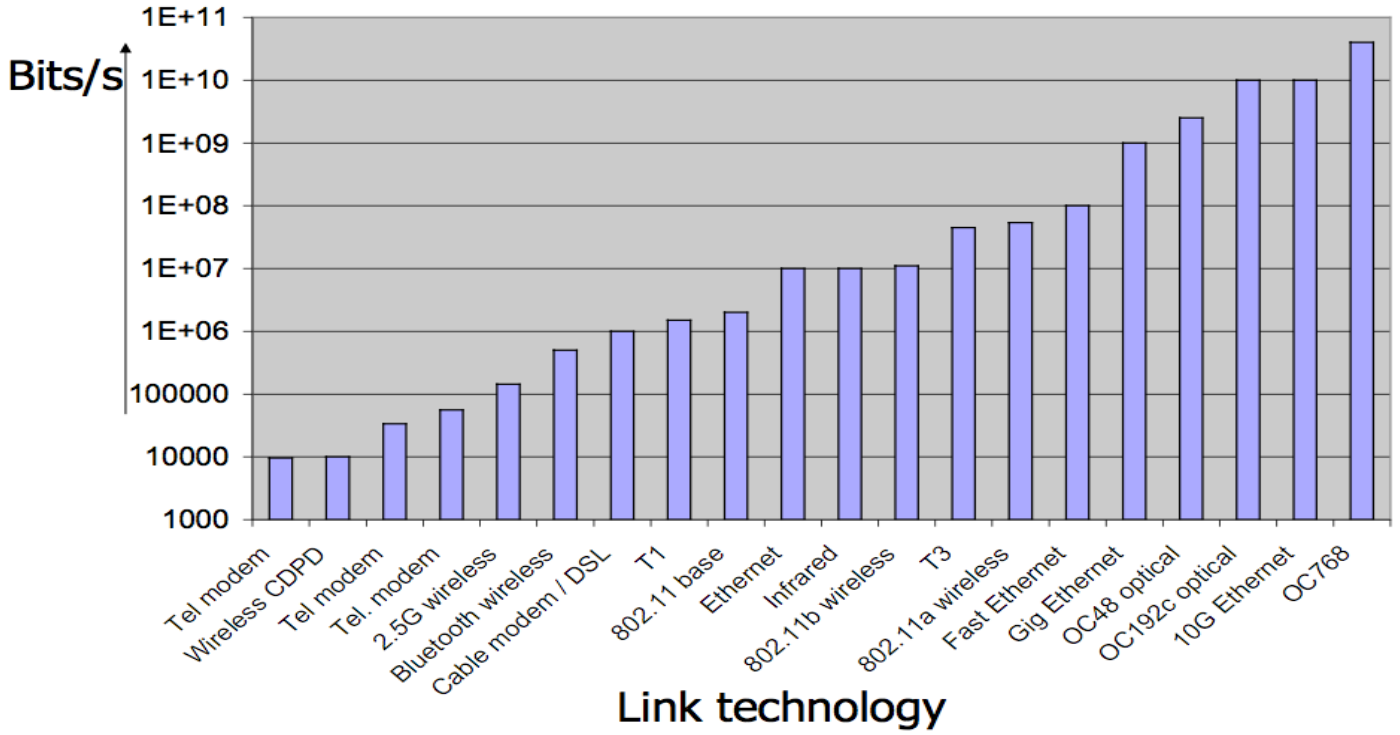
- Considerations
  - Data size
  - Data type
  - Transfer speed: throughput vs. latency
  - Transfer distance
  - Transfer method
  - …

d(tech)/dt for Networks



Thanks to Nick Mckeown @ Stanford for some of these data points

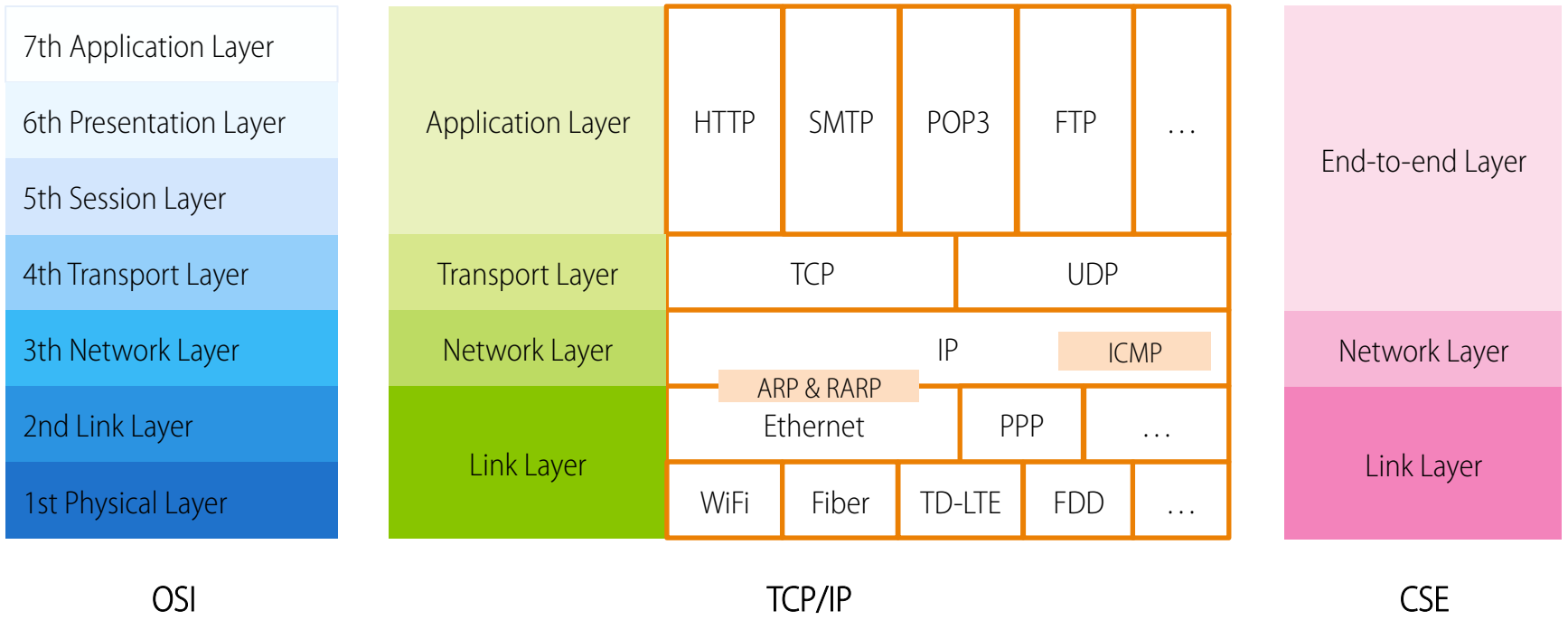
Networks are Heterogeneous



Layers in Network

- Link layer
  - Moving data directly from one point to another
- Network layer
  - Forwarding data through intermediate points to the place it is wanted
- End-to-end layer
  - Everything else required to provide a comfortable application interface
- Application
  - Can be thought of as a fourth layer
  - Not part of the network

OSI, TCP/IP & Protocol Stack

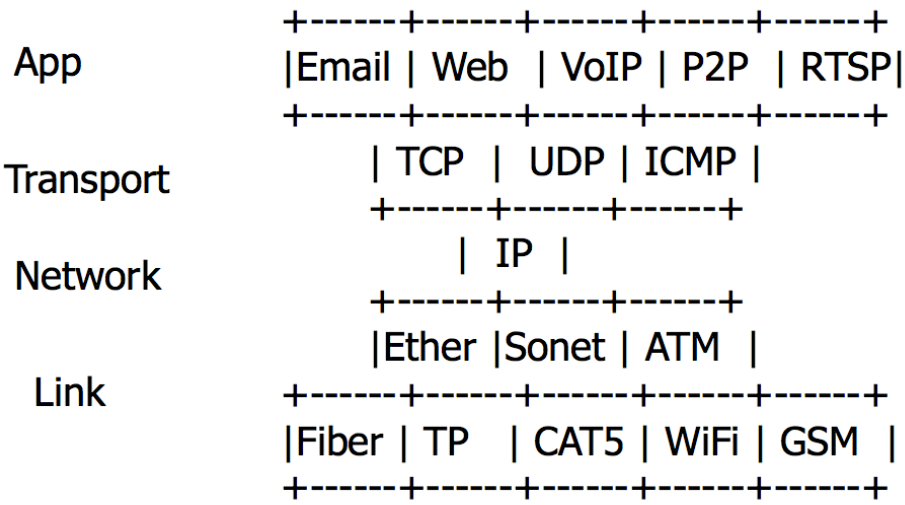


Question

- What do we talk when we talk about Internet
  - Which layer does the Internet reside in?

The Internet "Hour Glass"

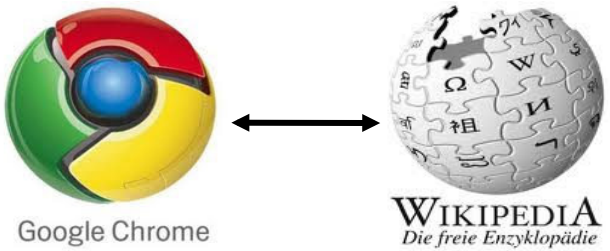
- More people, more useful
  - Value to me = N
  - Value to society is N^2
- Network, dumb vs. smart
  - Standardize vs. flexibility
- Network is a black box
  - Simplify the system that uses it



"Everything over IP, and IP over everything"

Application Layer

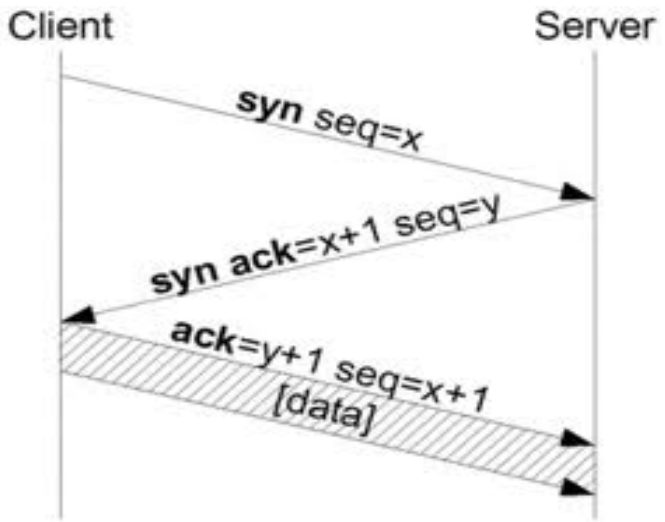
- Entities
  - Client and server
  - End-to-end connection
- Name space:
  - URL
- Protocols
  - HTTP, FTP, POP3, SMTP, etc.
- What to care?
  - Content of the data: video, text, …



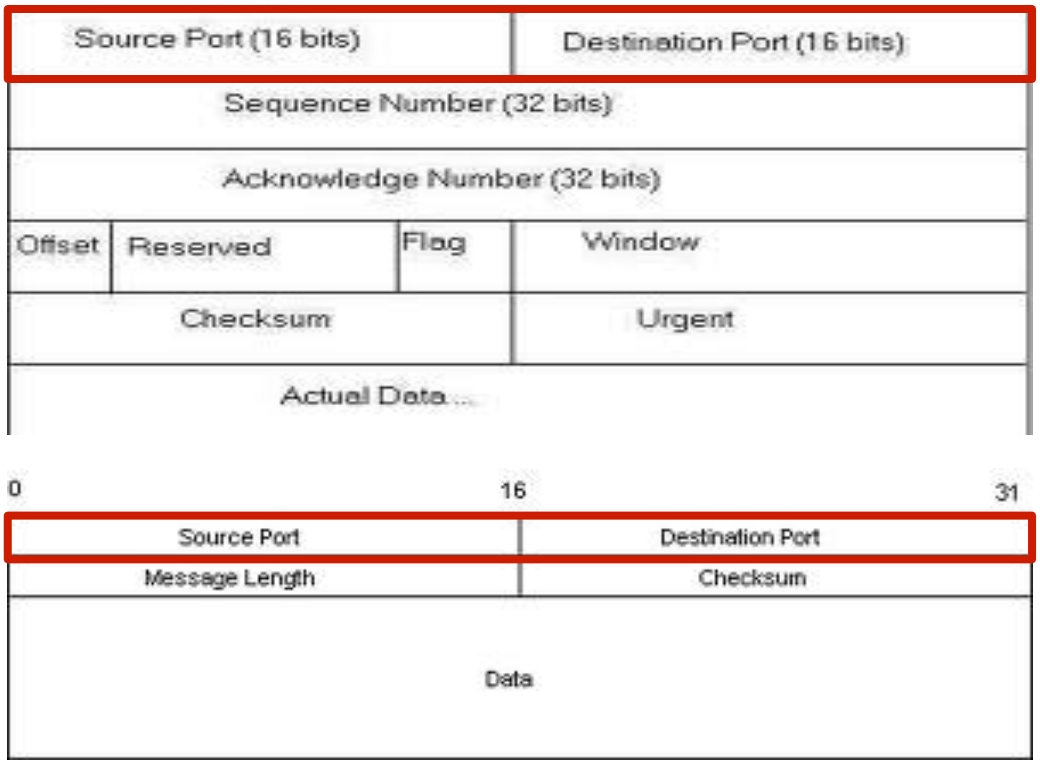

```
<html>
  <head>
    <title>Google</title>
    <script>window.google=..... </script>
  </head>
  <body>  ...  </body>
</html>
```

Transport Layer

- Entities
  - Sender and receiver
  - Proxy, firewall, etc.
  - End-to-end connection
- Name space: *port* number
- Protocols: TCP, UDP, etc.
- What to care?
  - TCP: Retransmit packet if lost
  - UDP: nothing



Packet Format of TCP & UDP

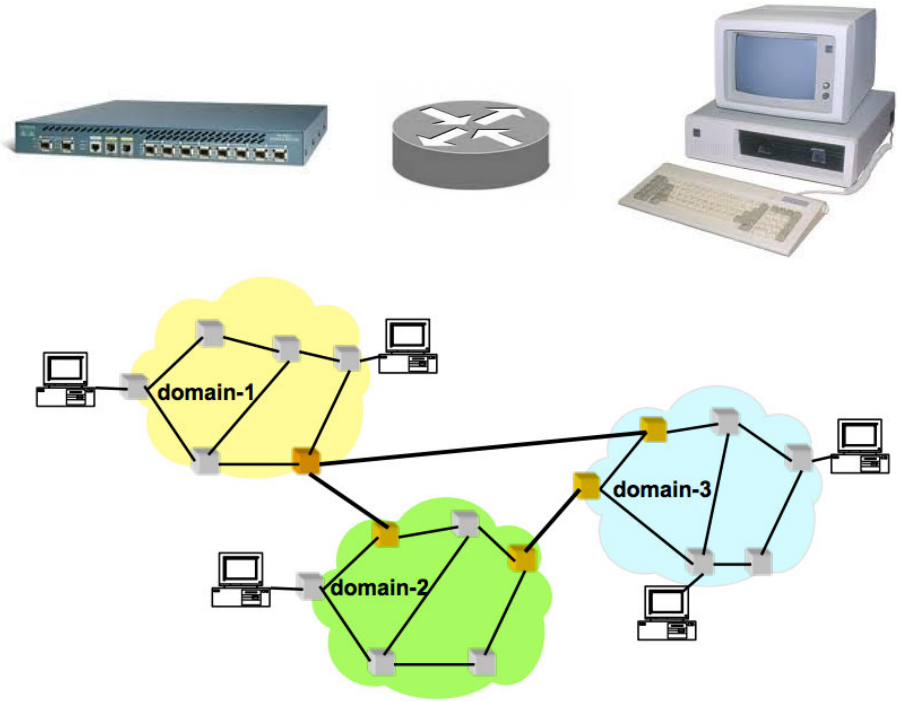


TCP

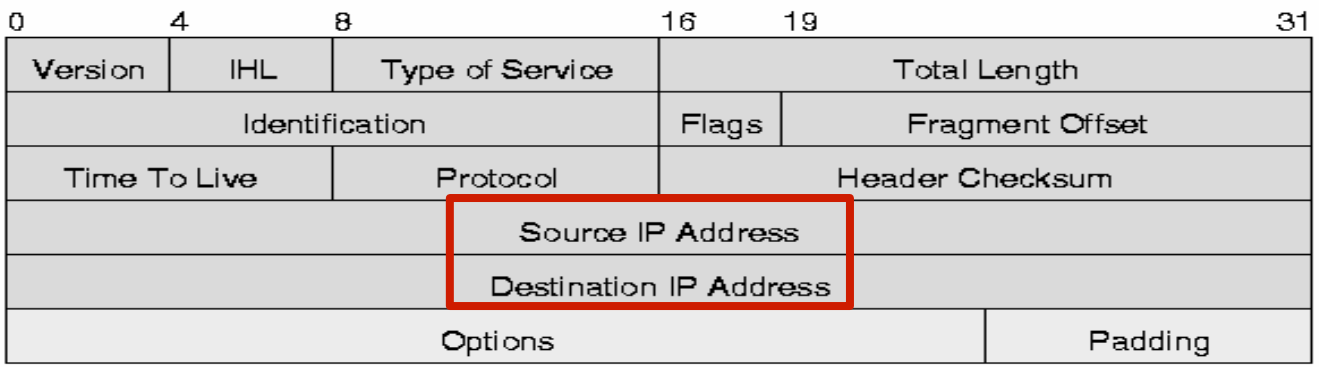
UDP

Network Layer (the Internet Layer, IP Layer)

- Network entities
  - Gateway, bridge
  - Router, etc.
- Name space
  - IP address
- Protocols
  - IP, ICMP (ping)
- What to care?
  - Next hop decided by route table



IP Datagram (Packet, Package)

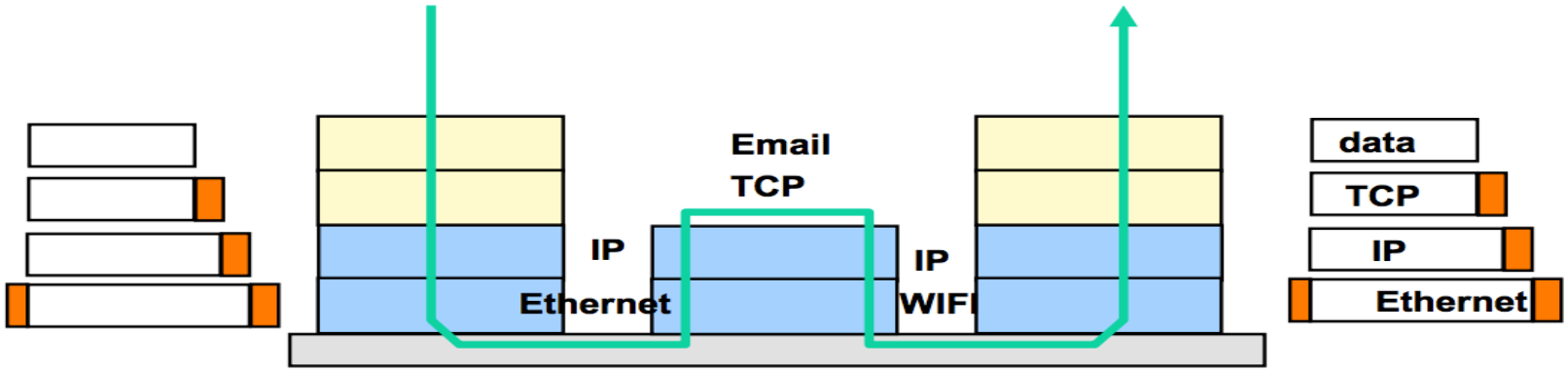


Header

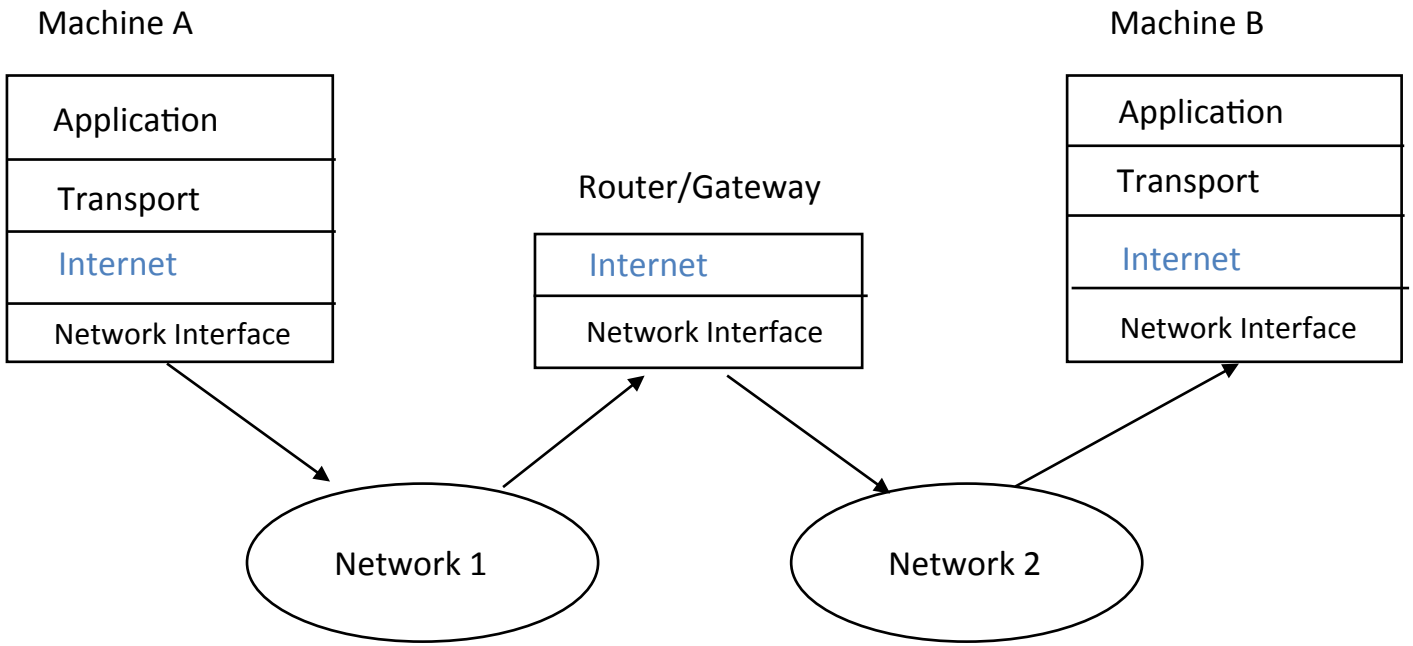
```
10101011101010101010010101010100101010100
11010010101010010101111111010000011101111
10100001011101010100110101011110100000101
00100000000010101000011010000111111010101
..... 10110110010101000110010010101l0
                                  Data
```

TCP/IP Architecture: Internet Layer

- Each layer adds/strips off its own header
- Each layer may split up higher-level data
- Each layer multiplexes multiple higher layers
- Each layer is (mostly) transparent to higher layers



TCP/IP Architecture: Internet Layer



IPv4 Address

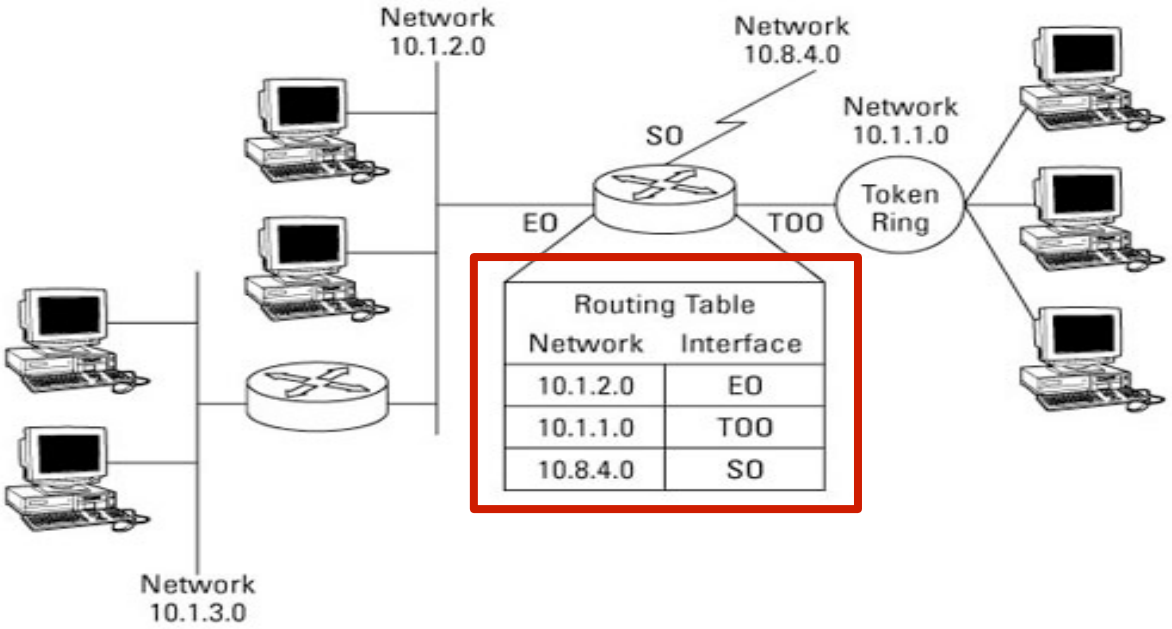
Historical classful network architecture

Class	Leading address bits	Range of first octet	Network ID format	Host ID format	Number of networks	Number of addresses
A	0	0 - 127	a	b.c.d	$2^7 = 128$	$2^{24} = 16777216$
B	10	128 - 191	a.b	c.d	$2^{14} = 16384$	$2^{16} = 65536$
C	110	192 - 223	a.b.c	d	$2^{21} = 2097152$	$2^8 = 256$

Private IPv4 network ranges

	Start	END	No. of addresses
24-bit Block (/8 prefix, 1 × A)	10.0.0.0	10.255.255.255	16777216
20-bit Block (/12 prefix, 16 × B)	172.16.0.0	172.31.255.255	1048576
16-bit Block (/16 prefix, 256 × C)	192.168.0.0	192.168.255.255	65536

IP Route Table



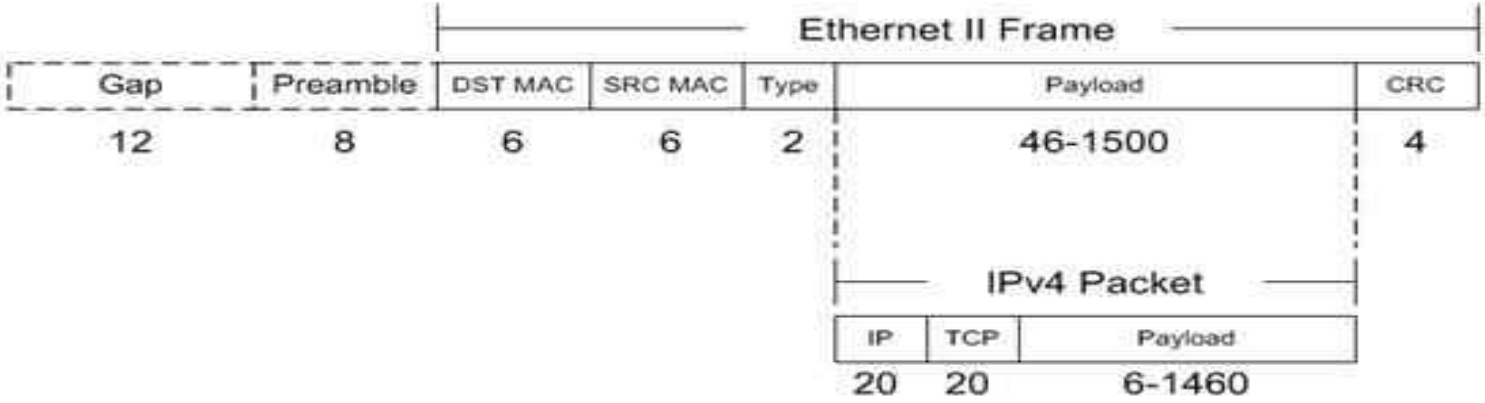
ICMP Protocol

- Ping
  - Test the reachability of a host
  - Measure the RTT (Round Trip Time)
- "Ping of death" attack
  - Sending a ping larger than 65535 bytes
  - Many computer could not handle and then crash
  - Fixed since 1997 & 1998
- "Ping flooding" attack
  - DoS (Denial of Service)

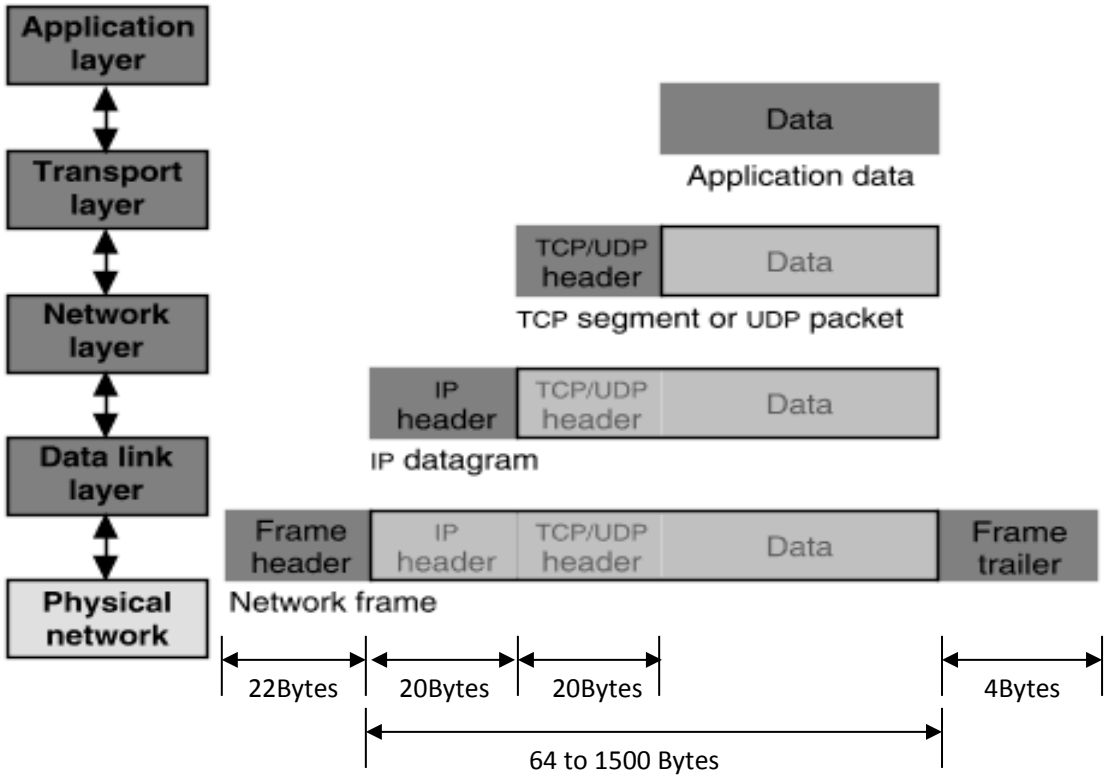
Link Layer (& Physical Layer)

- Network entities
  - Hub, switcher, etc.
  - Twisted line, cable line, etc.
- Name space
  - No name needed
- Protocols
  - Ethernet, ATM, etc.
  - ARP, RARP, etc.
- What to care?
  - Physical transfer, error detection, etc.

Frame Format of Ethernet



Packet Encapsulation



From a node to its physical neighbor

LINK LAYER

The Link Layer

- The bottom-most layer of the three layers
- Purpose: moving data directly from one physical location to another

- 1. Physical transmission
- 2. Multiplexing the link
- 3. Framing bits & bit sequences
- 4. Detecting transmission errors
- 5. Providing a useful interface to the up layer

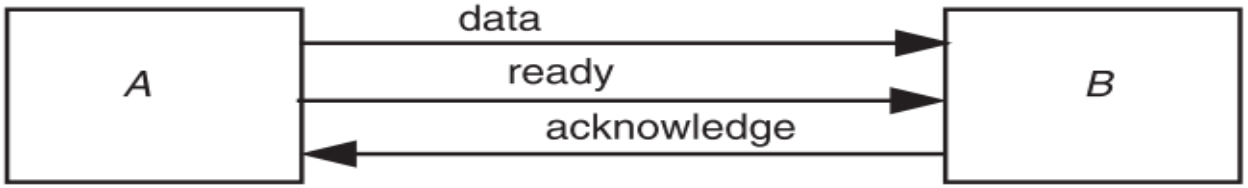
Physical Transmission using Shared Clock

- Example-1: moving a bit from register-1 to register-2 on the same chip
  - Run a wire to connect output of reg-1 to input of reg-2
  - Wait till reg-1's output has settled & signal has propagated to reg-2
  - Reg-2 read input the next clock tick
  - Assumption: propagation can be done within one clock

- How to send data between two modules without sharing a clock?

Physical Transmission without Shared Clock

- Three-wire ready/acknowledge protocol
  - 1. A places data on data line
  - 2. A changes value on the ready line
  - B sees the ready line change, reads value on the data line, then changes the acknowledge line



- B: when to look at the data line?
- A: when to stop holding the bit value on the data line?

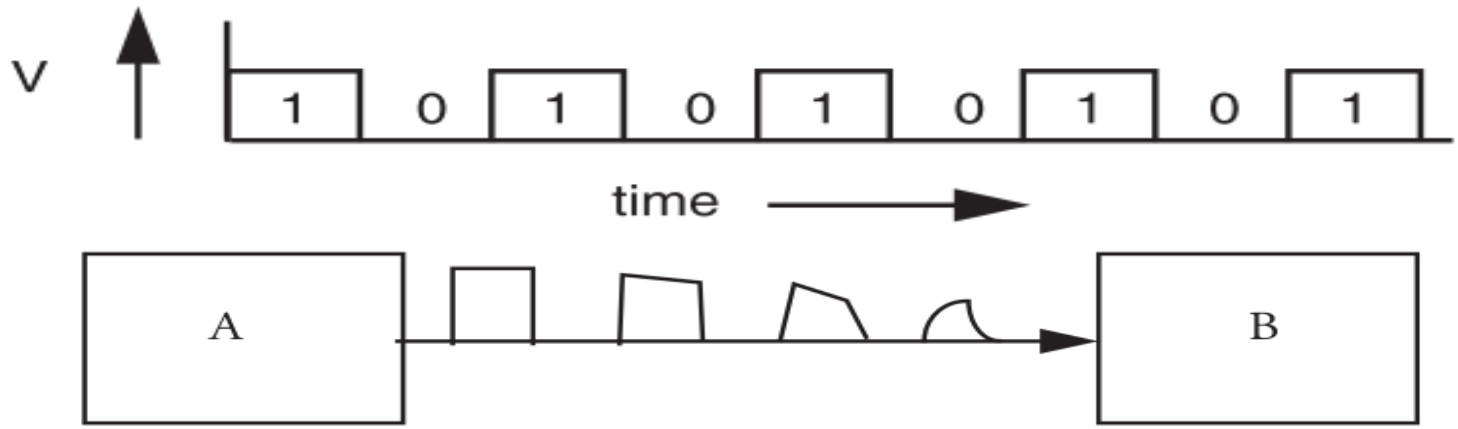
Parallel Transmission

- Propagation time Δt
  - It takes more than $2\Delta t$ to send one bit
  - The max data rate is $1/(2\Delta t)$
- Parallel transmission
  - Use N parallel data lines to achieve $N/(2\Delta t)$
  - E.g. SCSI, printer, etc.

Parallel vs. Serial

- Ready/acknowledge protocol
  - Δt grows significantly, which limits the data rate
- Serial transmission
  - Send a stream of bits down a single line
  - Without waiting for any response from the receiver
  - Expect the receiver can recover the bits with no additional signal
  - Higher rates, longer distance, fewer wires
  - E.g., USB, SATA

Signal Transmission on Analog Line



- It is hard for B to understand the signal
  - B doesn't have a copy of A's clock, so when to sample the signal?

VCO: Voltage Controlled Oscillator

- How to make two ends agree on the data rate without clock line?

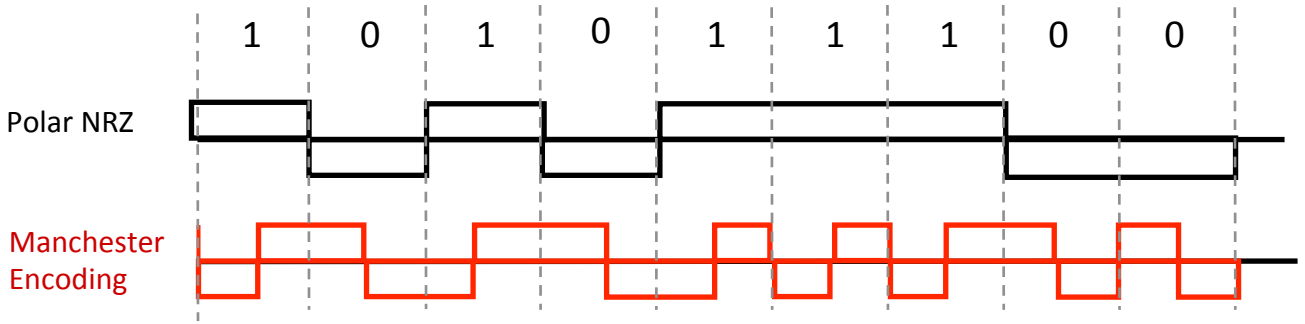
- The receiver run a VCO at about the same data rate
- VCO's output is multiplied by the voltage of incoming signal
- The product is suitably filtered and sent back to adjust the VCO

- VCO will finally be *locked* to both the frequency and phase of the arriving signal: phase-locked loop
- Then the VCO becomes a clock source for the receiver

- Problem: if no transition in the stream (e.g., a lot of zero), the phase-locked loop cannot synchronize

Manchester Code

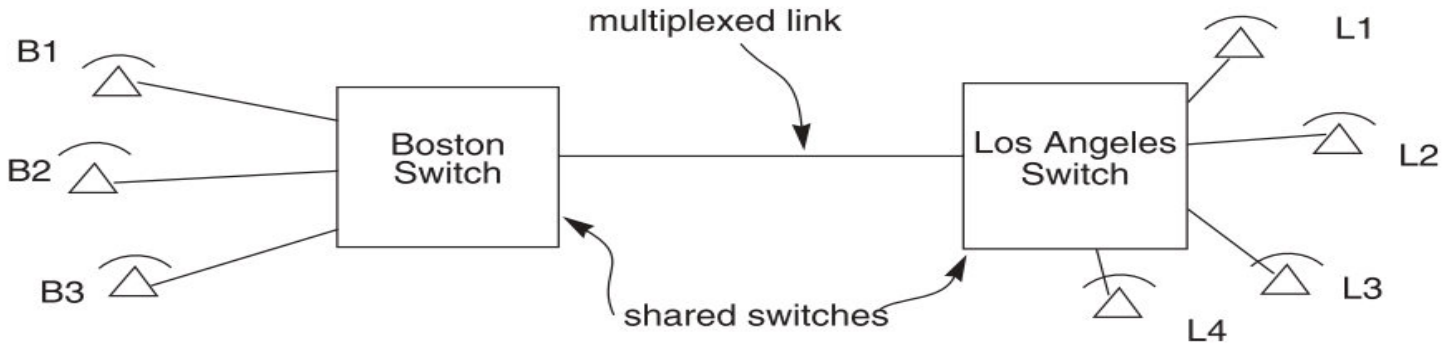
- Solution: sender encodes the data to ensure transitions
- Phase encoding: at least 1 level transition for a bit
  - Manchester code: 0 -> 01, 1 -> 10
  - Max data rate is only half, but simple enough



Sharing a Connection

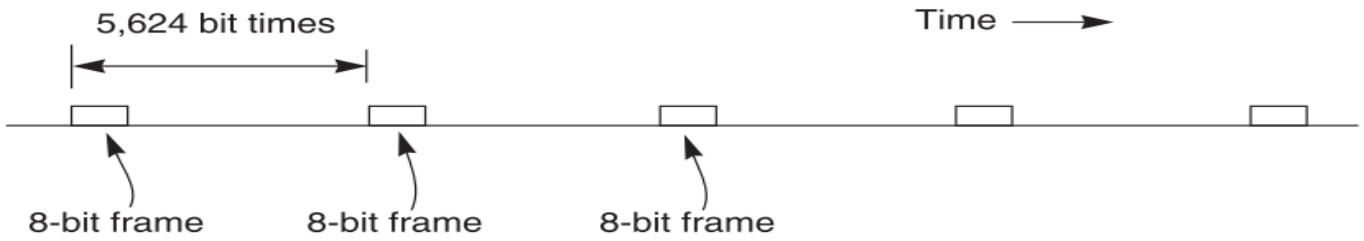
- Isochronous communication (telephone communication)
  - Needs prior arrangement between switches
  - Connection: set up and tear down
  - Stream: continuous bits flows out of a phone
- Asynchronous communication (data communication)
  - Message: burst, ill-suited to fixed size and spacing of isochronous frames
  - Connectionless, asynchronous

Isochronous Multiplexing



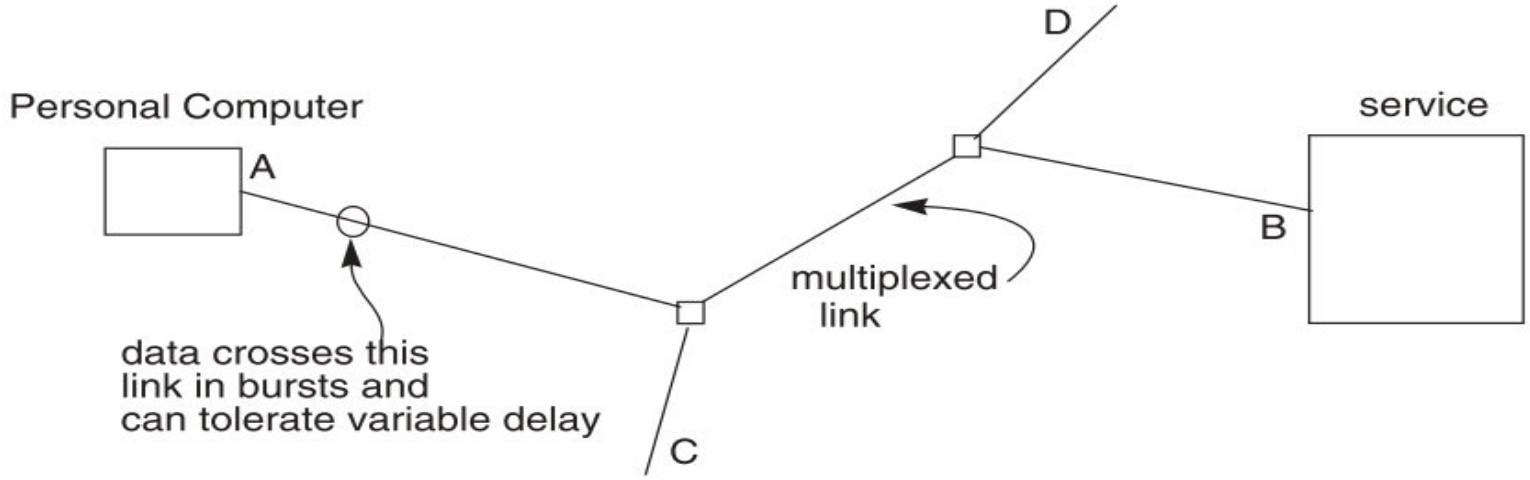
- Telephone network
  - Leverage "virtual link" for connection
  - "network is busy" when no available time slot

Isochronous - TDM



- 64 Kbps each phone, 45 Mbps link
- 8-bit block (frame), 8000 frames per second
- 5624 bit times or 125 us
- 703 simultaneous conversations
- Q: Why the voice is still *continuous*, instead of *fragmented*?

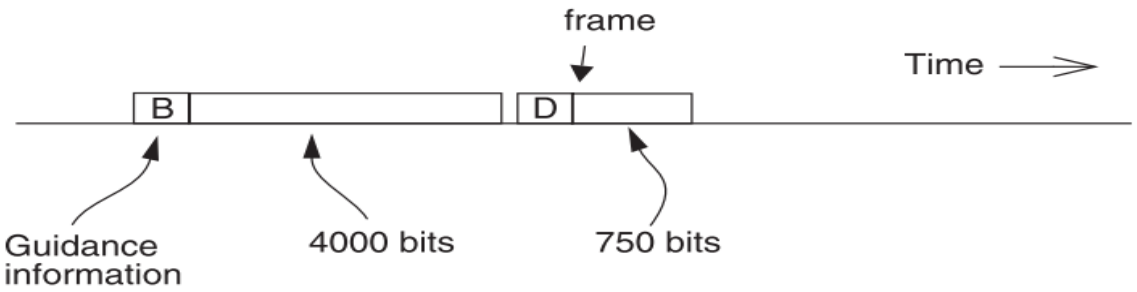
Data Communication Network



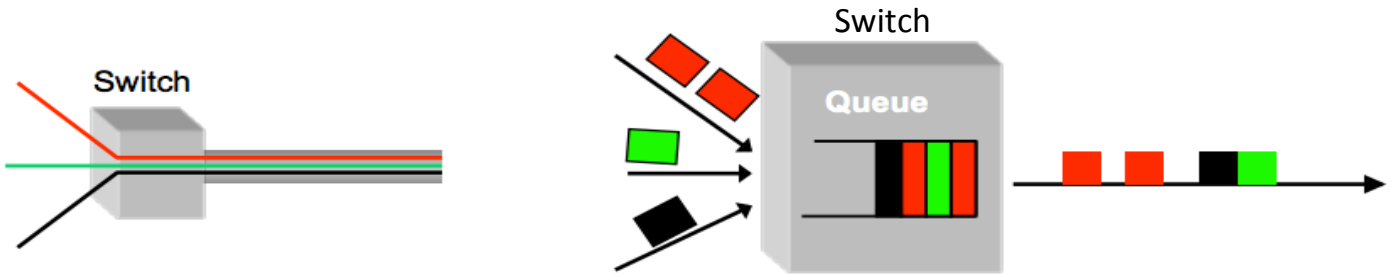
- Data communication network usually contains burst communication
- Different from the telephone network

Frame and Packet: Asynchronous Link

- Frame can be of any length, carried at any time that the link is free
- Packet: a variable-length frame with its guidance info
- Connectionless transmission: no state maintained
- Segment and reassemble
- Packet voice: replacing many parts of isochronous network



Multiplexing / Demultiplexing



- Multiplex using a queue: switch need memory/buffer
- Demultiplex using information in packet header
  - Header has destination
  - Switch has a forwarding table that contains information about which link to use to reach a destination

Framing Frames

- Where a frame begins and ends
- Independent from framing bits
  - Some model separates link layer to 2: one for bits and one for frames
- Simple method
  - Choose a pattern of bits, e.g. 7 one-bits in a row, as a frame-separator
  - Bit stuffing: if data contains 6 ones in a row, then add an extra bit (0)

Error Handling

- Error detection code
  - Adding redundancy: e.g., checksum at the end
- What to do if detect an error
  - Error correction code: with enough redundancy
    - Where noise is well understood, e.g., disk
  - Ask sender to resend: sender holds frame in buffer
  - Let receiver discard the frame
  - Blending these techniques

Coding: Incremental Redundancy

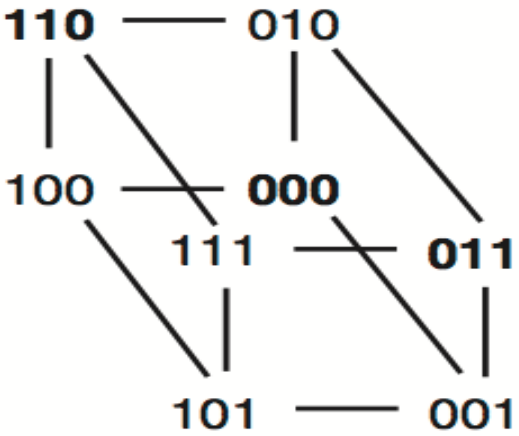
- Forward error correction
  - Perform coding before storing or transmitting
  - Later decode the data without appealing to the creator
- Hamming distance
  - Number of 1 in $A \oplus B$, $\oplus$ is exclusive OR (XOR)
  - If H-distance between every legitimate pair is 2
    - 000101, can only detect 1-bit flip
  - If H-distance between every legitimate pair is 3
    - Can only correct 1 bit flip
  - If H-distance between every legitimate pair is 4
    - Can detect 2-bit flip, correct 1-bit flip

100101
000111

100101
010111

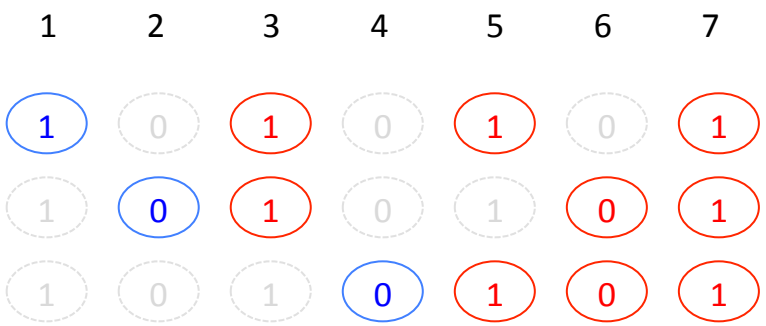
Example-1: Simple Parity Check

- 2 bits -> 3 bits
  - Detect 1-bit errors
  - 8 patterns total
- Only 4 correct patterns
  - 00 -> 000
  - 11 -> 110
  - 10 -> 101
  - 01 -> 011
- Hamming distance of this code is 2
  - 1-bit flipping will cause incorrect pattern



Example-2: 4-bit -> 7-bit

- 4 bits -> 7 bits (56 using only extra 7)
  - 3 extra bits to distinguish 8 cases
  - e.g. 1101 -> 1010101
- Correct 1-bit errors
  - 1010101 -> 1010001 : P1 & P4 not match
  - 1010101 -> 1110101 : P2 not match



$$P_1 = P_7 \oplus P_5 \oplus P_3$$
$$P_2 = P_7 \oplus P_6 \oplus P_3$$
$$P_4 = P_7 \oplus P_6 \oplus P_5$$

Not Match	Error
None	None
P1	P1
P2	P2
P4	P4
P1 & P2	P3
P1 & P4	P5
P2 & P4	P6
P1 & P2 & P4	P7

Computer System Engineering, Spring 2015. (IPADS, SJTU)

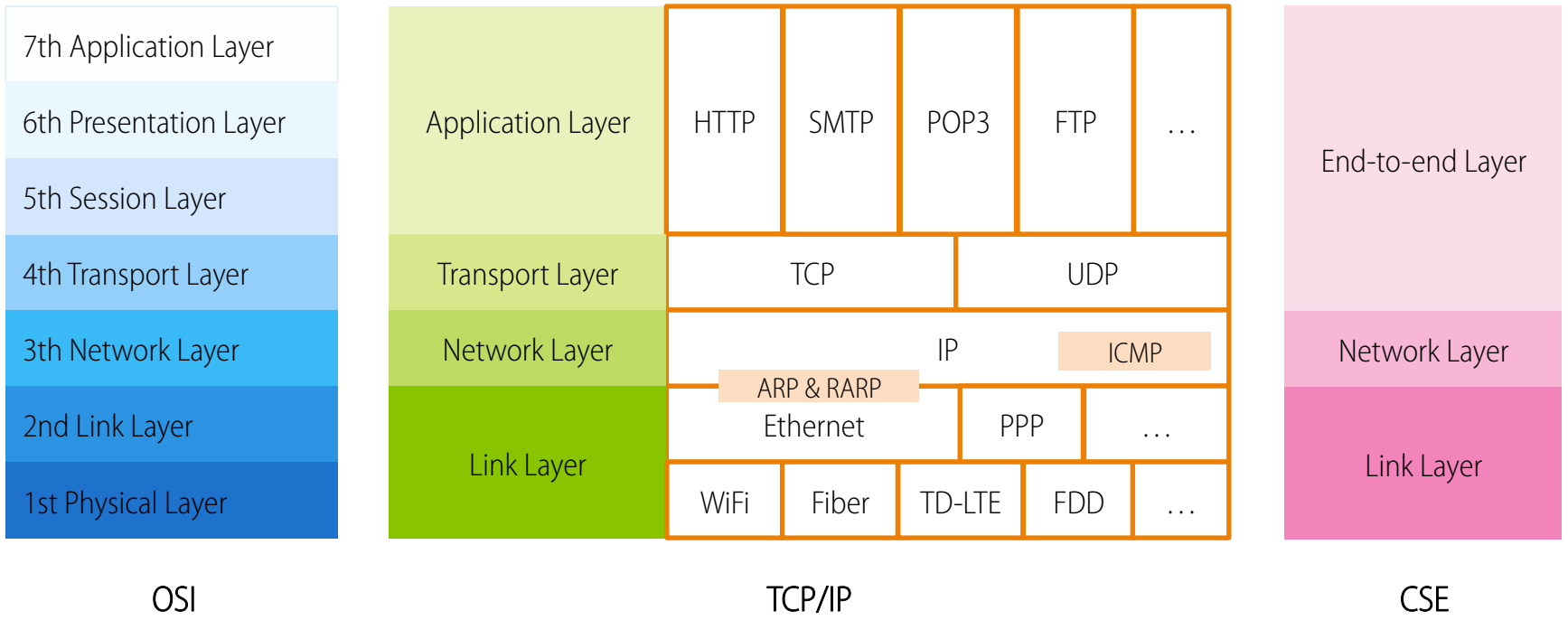
Network Layer

It's all about routing

Why Learn About Networks?

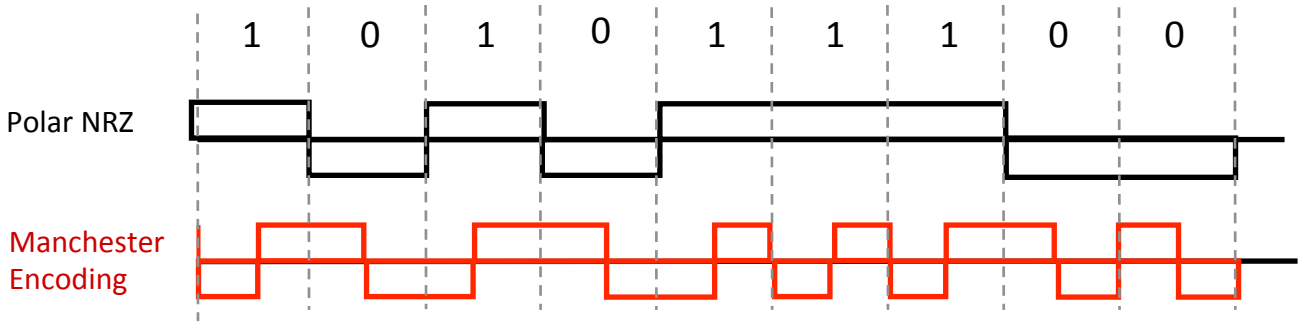
- Many computer systems use the network
- The Internet is an interesting example of a successful system

Review: OSI, TCP/IP & Protocol Stack



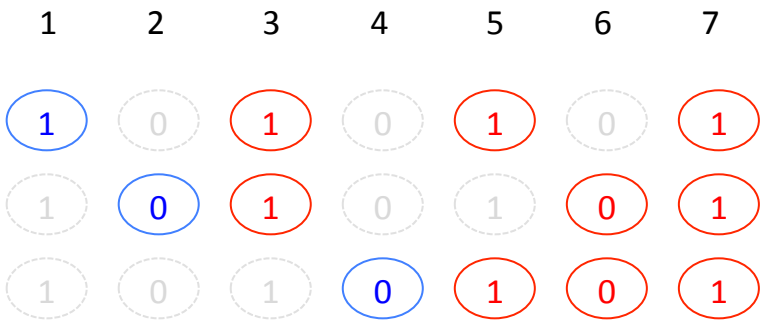
Review: Manchester Code

- Solution: sender encodes the data to ensure transitions
- Phase encoding: at least 1 level transition for a bit
  - Manchester code: 0 -> 01, 1 -> 10
  - Max data rate is only half, but simple enough



Review: 4-bit -> 7-bit Encoding

- 4 bits -> 7 bits (56 using only extra 7)
  - 3 extra bits to distinguish 8 cases
  - e.g. 1101 -> 1010101
- Correct 1-bit errors
  - 1010101 -> 1010001 : P1 & P4 not match
  - 1010101 -> 1110101 : P2 not match



$$P_1 = P_7 \oplus P_5 \oplus P_3$$
$$P_2 = P_7 \oplus P_6 \oplus P_3$$
$$P_4 = P_7 \oplus P_6 \oplus P_5$$

Not Match	Error
None	None
P1	P1
P2	P2
P4	P4
P1 & P2	P3
P1 & P4	P5
P2 & P4	P6
P1 & P2 & P4	P7

IP: Best-effort Network

- Best-effort network
  - If it cannot dispatch, may discard a packet
- Guaranteed-delivery network
  - Also called store-and-forward network, no discarding data
  - Work with complete messages rather than packets
  - Uses disk for buffering to handle peaks
  - Tracks individual message to make sure none are lost
- In real world
  - No absolute guarantee
  - Guaranteed-delivery: higher layer; best-effort: lower layer

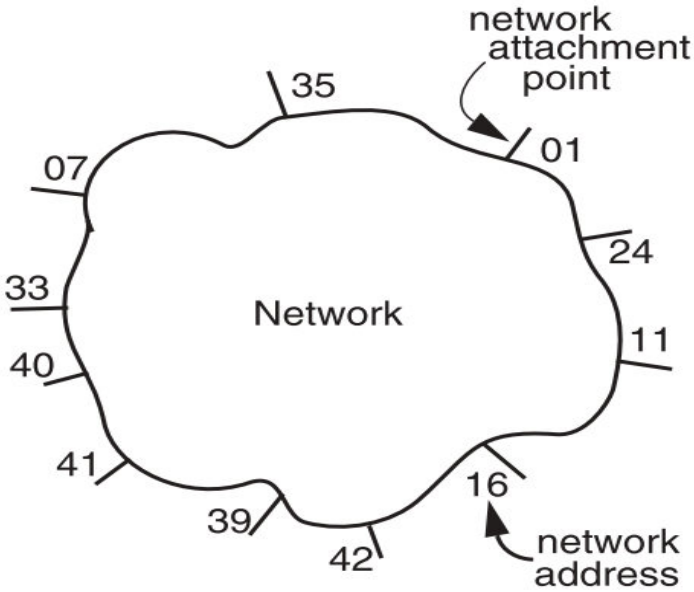
Duplicate Packets and Suppression

- Discarding packets is common case
  - Many network protocol includes timeout and resend mechanism

- When a congested forwarder discards a packet
  - Client doesn't receive a response as quickly as originally hoped
  - Users may prepared for duplicate requests and responses
  - Detecting duplicates may or may not be important

The Network Layer

- Addressing interface
  - Network attachment points
  - Network address
  - Source & destination

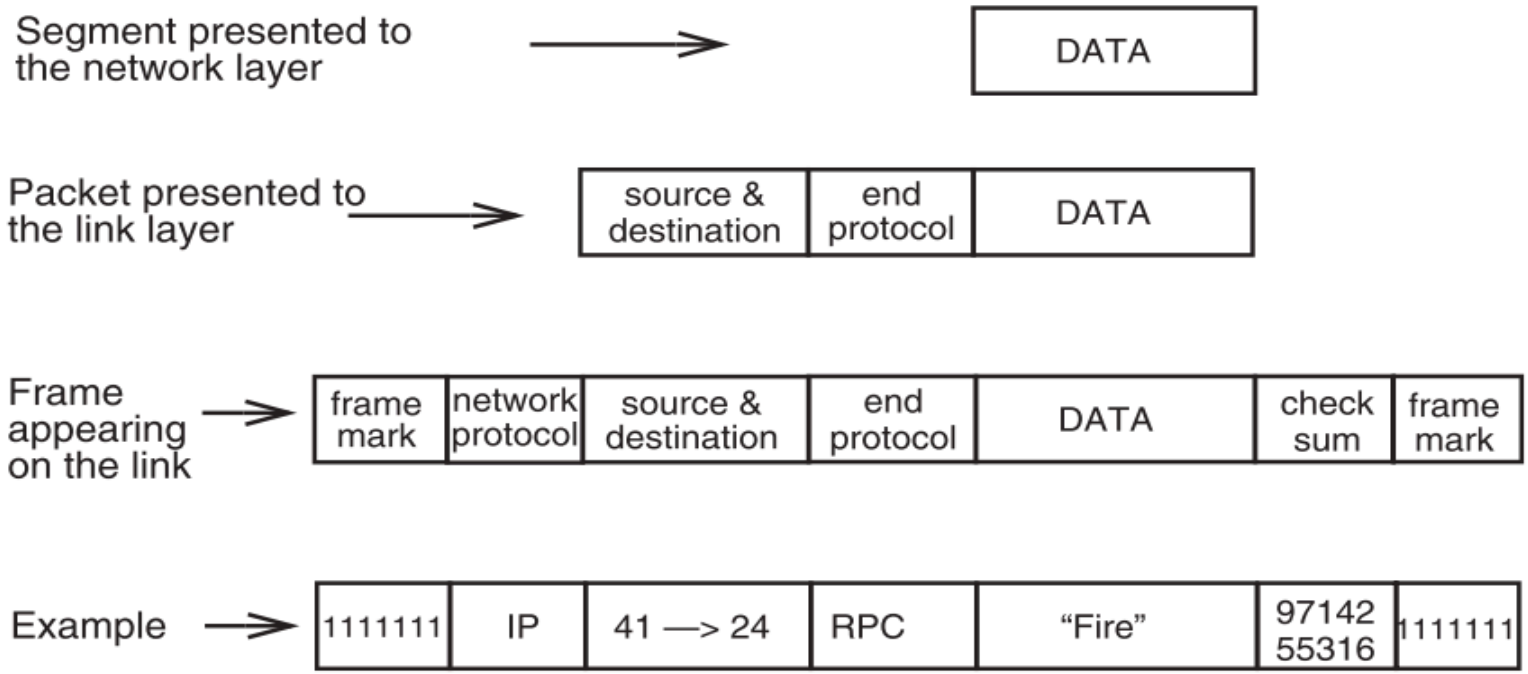


- NETWORK_SEND (segment_buffer, destnation, network_protocol, end_layer_protocol)
- NETWORK_HANDLE (packet, network_protocol)

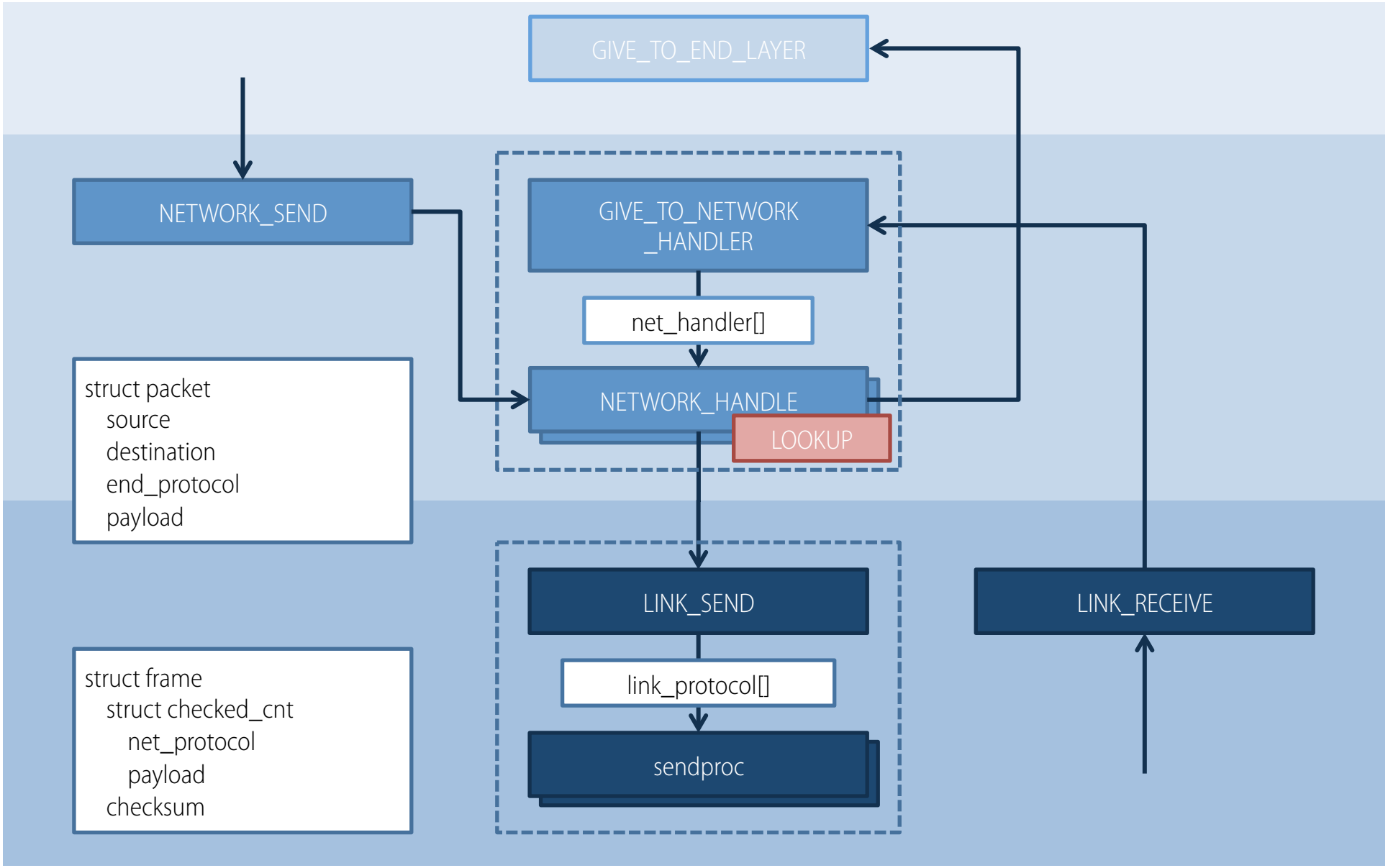
Network Layer Interface

```
structure packet
    bit_string source
    bit_string destination
    bit_string end_protocol
    bit_string payload
```

Accumulation of Headers and Trailers



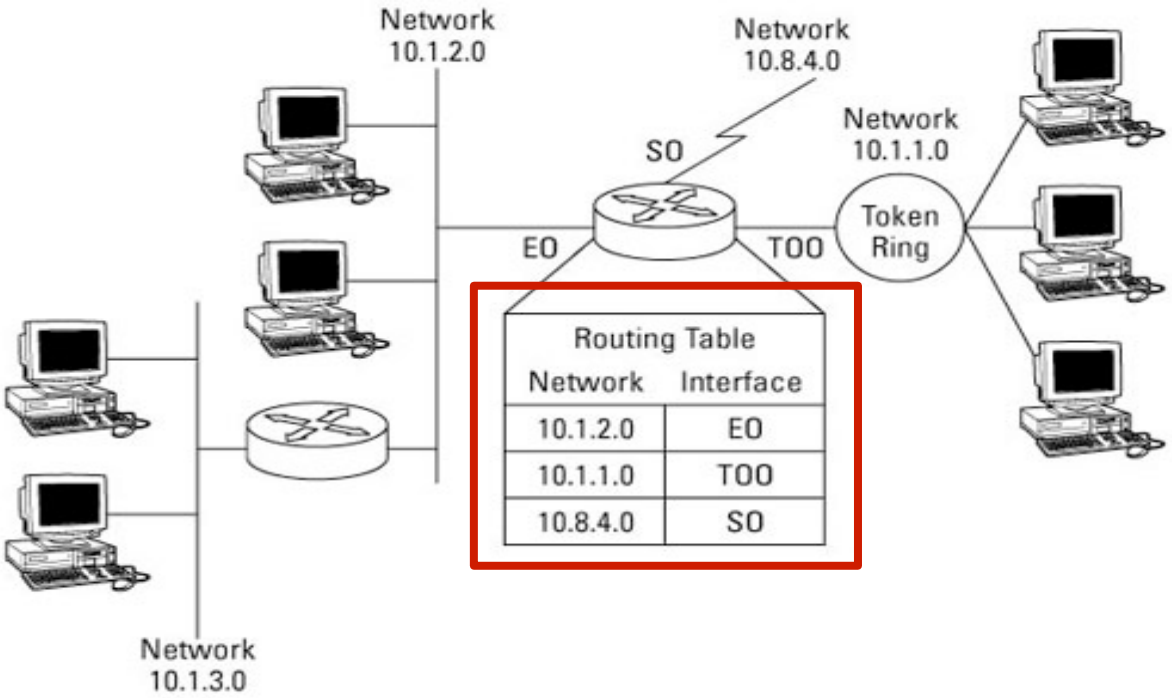
```
1 procedure NETWORK_SEND (segment_buffer, destination,
2                         net_protocol, end_protocol)
3     packet instance outgoing_packet
4     outgoing_packet.payload ← segment_buffer
5     outgoing_packet.end_protocol ← end_protocol
6     outgoing_packet.source ← MY_NETWORK_ADDRESS
7     outgoing_packet.destination ← destination
8     NETWORK_HANDLE (outgoing_packet, net_protocol)

9 procedure NETWORK_HANDLE (net_packet, net_protocol)
10    packet instance net_packet
11    if net_packet.destination != MY_NETWORK_ADDRESS then
12        next_hop ← LOOKUP (net_packet.destination, forwarding_table)
13        LINK_SEND (net_packet, next_hop, link_protocol, net_protocol)
14    else
15        GIVE_TO_END_LAYER (net_packet.payload,
16                           net_packet.end_protocol, net_packet.source)
```


GIVE_TO_END_LAYER
NETWORK_SEND
GIVE_TO_NETWORK
_HANDLER
net_handler[]
NETWORK_HANDLE
LOOKUP
struct packet
source
destination
end_protocol
payload
LINK_SEND
LINK_RECEIVE
link_protocol[]
sendproc
struct frame
struct checked_cnt
net_protocol
payload
checksum

Managing the Forwarding Table: Routing

- Routing (or path-finding)
  - Constructing the tables
- Impractical by hand
  - Determining the best paths requires calculation
  - Recalculating the table when links change
  - Recalculating the table when link fails
  - Adapt according to traffic congestion
- Static routing vs. adaptive routing
  - Adaptive routing requires exchange of info

IP Route Table

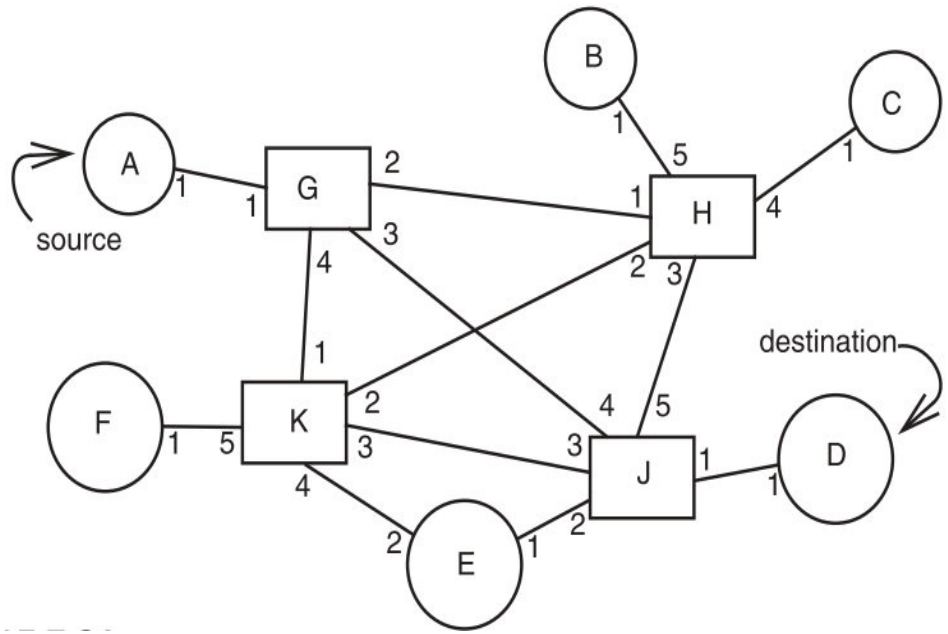


Path Vector Exchange

- Each participant maintains a path vector
  - A complete path to some destination
  - E.g. zero-length path to itself
  - Gradually learns about other paths
  - Construct a new forwarding table from its new path vector
- Algorithm
  - Advertising
  - Path selection

to	path
G	< >

Path Vector Exchange



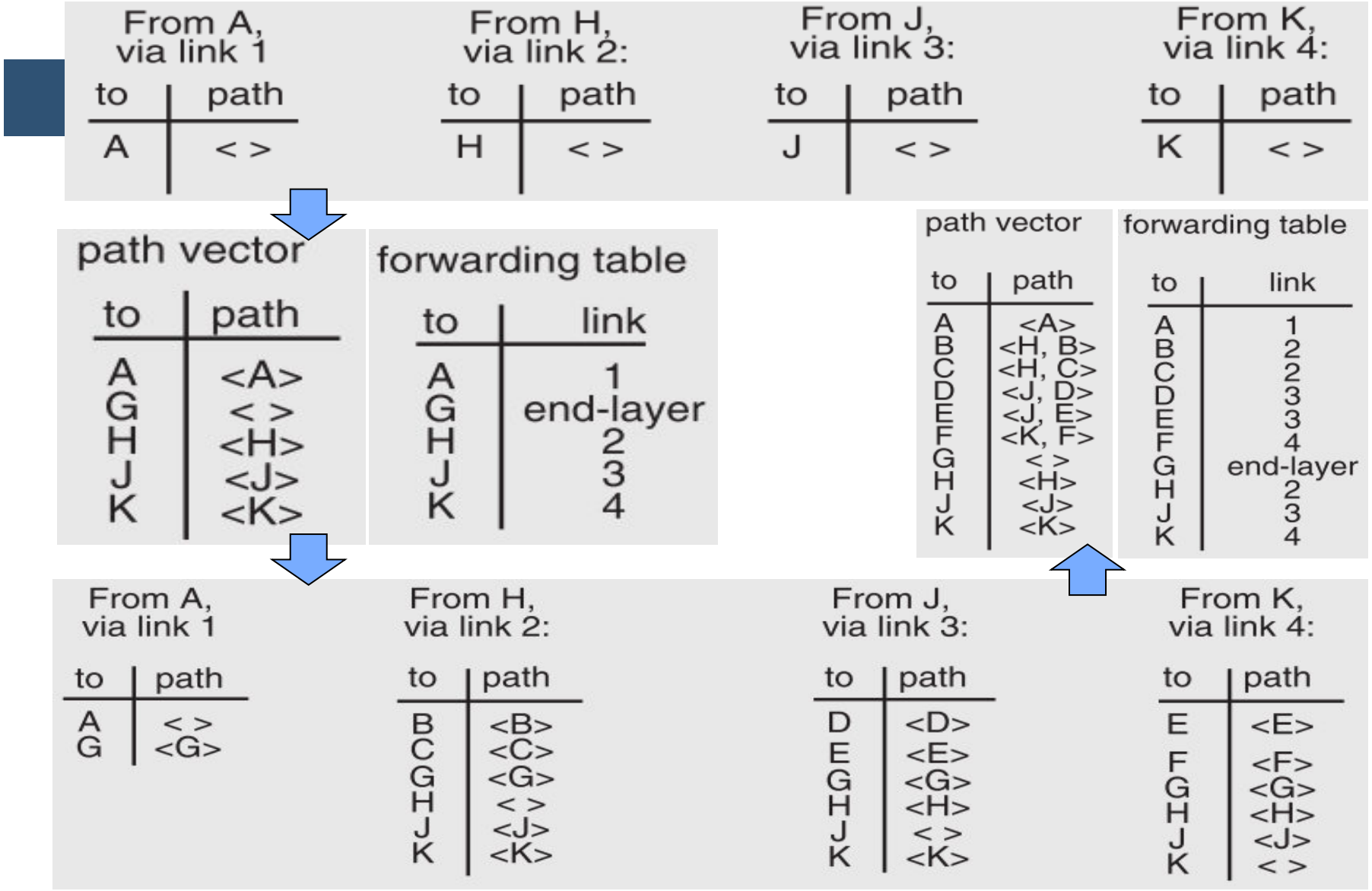
A

destination	link
A	end-layer
all other	1

G

destination	link
A	1
B	2
C	2
D	3
E	4
F	4
G	end-layer
H	2
J	3
K	4

- Need coordination to ensure no loop


From A, via link 1
to | path
A | < >
From H, via link 2:
to | path
H | < >
From J, via link 3:
to | path
J | < >
From K, via link 4:
to | path
K | < >
path vector
to | path
A | <A>
G | < >
H | <H>
J | <J>
K | <K>
forwarding table
to | link
A | 1
G | end-layer
H | 2
J | 3
K | 4
path vector
to | path
A | <A>
B | <H, B>
C | <H, C>
D | <J, D>
E | <J, E>
F | <K, F>
G | < >
H | <H>
J | <J>
K | <K>
forwarding table
to | link
A | 1
B | 2
C | 2
D | 3
E | 3
F | 4
G | end-layer
H | 2
J | 3
K | 4
From A, via link 1
to | path
A | < >
G | <G>
From H, via link 2:
to | path
B | <B>
C | <C>
G | <G>
H | < >
J | <J>
K | <K>
From J, via link 3:
to | path
D | <D>
E | <E>
G | <G>
H | <H>
J | < >
K | <K>
From K, via link 4:
to | path
E | <E>
F | <F>
G | <G>
H | <H>
J | <J>
K | < >

Path vector exchange routing algorithm

```
// Maintain routing and forwarding tables.
vector associative array  // vector[d_addr] contains path
                             to destination d_addr
neighbor_vector instance of vector // A path vector
                              received from some neighbor
my_vector instance of vector // My current path vector.
addr associative array // addr[j] is the address of the
                          network attachment point at the
                          other end of link j. my_addr is
                          address of my network attachment
                          point. A path is a parsable list
                          of addresses, e.g. {a,b,c,d}
```

Path vector exchange routing algorithm

```
procedure main()
  SET_TYPE_HANDLER (HANDLE_ADVERTISEMENT,
                    exchange_protocol)
  clear my_vector;
  do occasionally
    for each j in link_ids do
      status ← SEND_PATH_VECTOR (j, my_addr, my_vector,
                                 exch_protocol)
      if status != 0 then
        clear new_vector
        FLUSH_AND_REBUILD (j)
```

Path vector exchange routing algorithm

```
procedure HANDLE_ADVERTISEMENT (advt, link_id)
  addr[link_id] ← GET_SOURCE (advt)
  neighbor_vector ← GET_PATH_VECTOR (advt)
  for each neighbor_vector.d_addr do
    new_path ← {addr[link_id], neighbor_vector[d_addr]}
    if my_addr is not in new_path then
      if my_vector[d_addr] = NULL) then
        my_vector[d_addr] ← new_path
      else
        my_vector[d_addr] ← SELECT_PATH (new_path,
                                         my_vector[d_addr])
  FLUSH_AND_REBUILD (link_id)
```

Path vector exchange routing algorithm

```
procedure SELECT_PATH (new, old)
  if first_hop(new) = first_hop(old) then return new
  else if length(new) ≥ length(old) then return old
  else return new

procedure FLUSH_AND_REBUILD (link_id)
  for each d_addr in my_vector
    if first_hop(my_vector[d_addr]) = addr[link_id]
                  and new_vector[d_addr] = NULL
      then
      delete my_vector[d_addr]
  REBUILD_FORWARDING_TABLE (my_vector, addr)
```

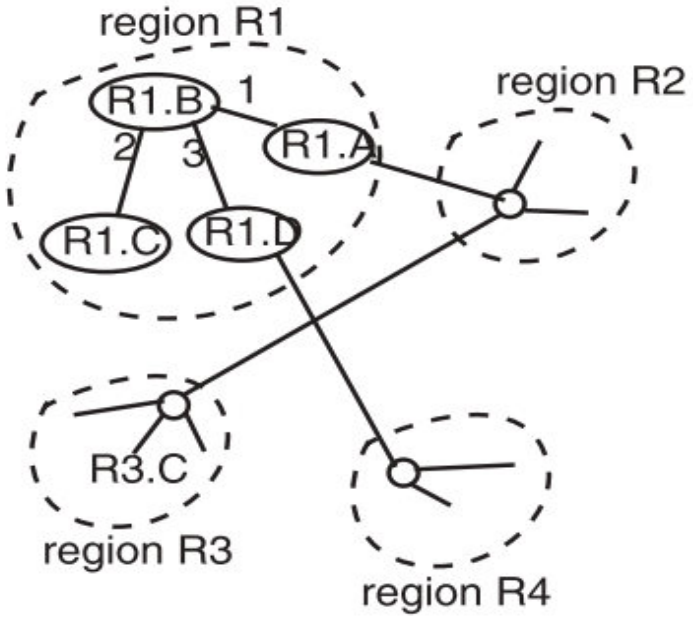
Question on Path Vector

- How do we avoid permanent loops?
  - When a node updates its paths, it never accepts a path that has itself
- What happens when a node hears multiple paths to the same destination?
  - It picks the better path
- What happens if the graph changes?
  - Algorithm deals well with new links
  - To deal with links that go down, each router should discard any path that a neighbor stops advertising

Hierarchical Address Assignment & Routing

- Two problems of the implementation
  - Every attachment point must have a unique address
  - The path vector grows in size with the number of attachment points
- Hierarchy
  - Tow parts of network address: region & station, e.g., “11,75”
  - Regions correspond to the set of closely-connected entities
  - Region 11 has only 1 entry in other region routers’ table
  - First forward to region, then to station

Hierarchical Address Assignment & Routing



forwarding table in R1.B

region forwarding section

to	link
R1	local
R2	1
R3	1
R4	3

local forwarding section

to	link
R1.A	1
R1.B	end-layer
R1.C	2
R1.D	3

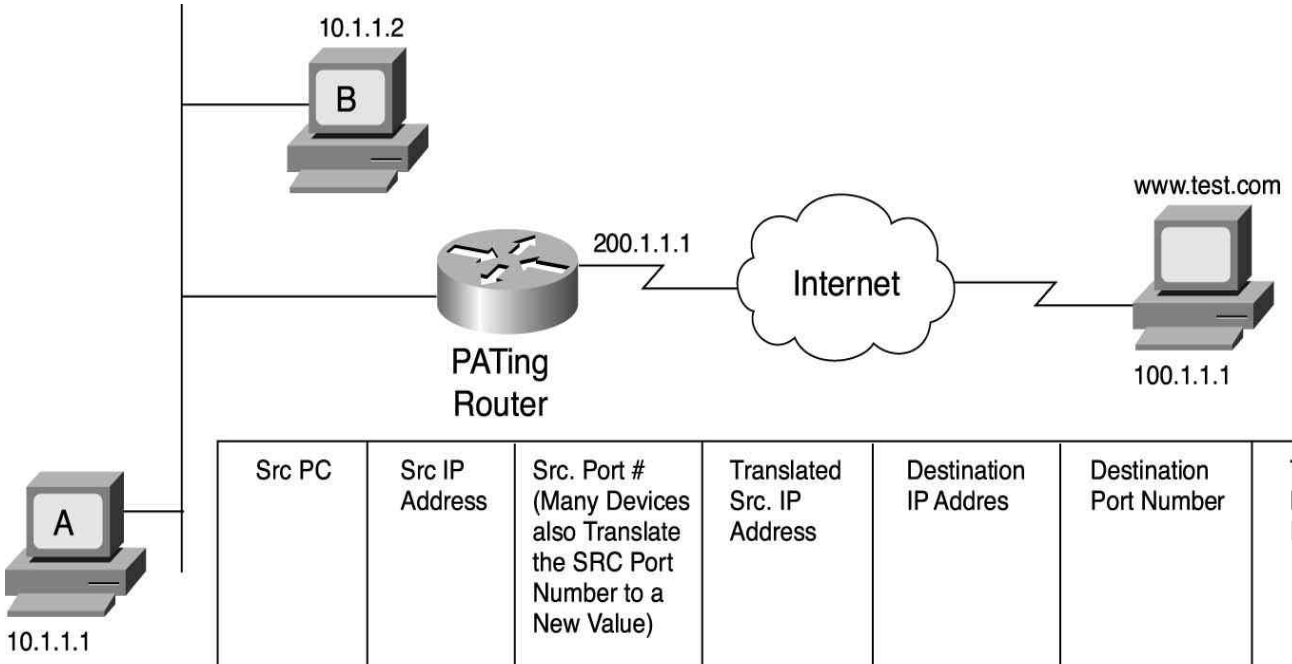
Hierarchical Address Assignment & Routing

- Problems introduced by hierarchy: more complex
  - Binding address with location
    - has to change address after changing location
  - Paths may no longer be the shortest possible
    - Algorithm has less detailed information
- More about hierarchy
  - Can extend to more levels
  - Different places can have different levels

NAT (Network Address Translation)

- Private network
  - Public routers don't accept routes to network 10
- NAT router: bridge the private networks
  - Router between private & public network
  - Send: modify source address to temp public address
  - Receive: modify back by looking mapping table
- Limitations
  - Some end-to-end protocols place address in payloads
  - The translator may become the bottleneck
  - What if two private network merge?

NAT



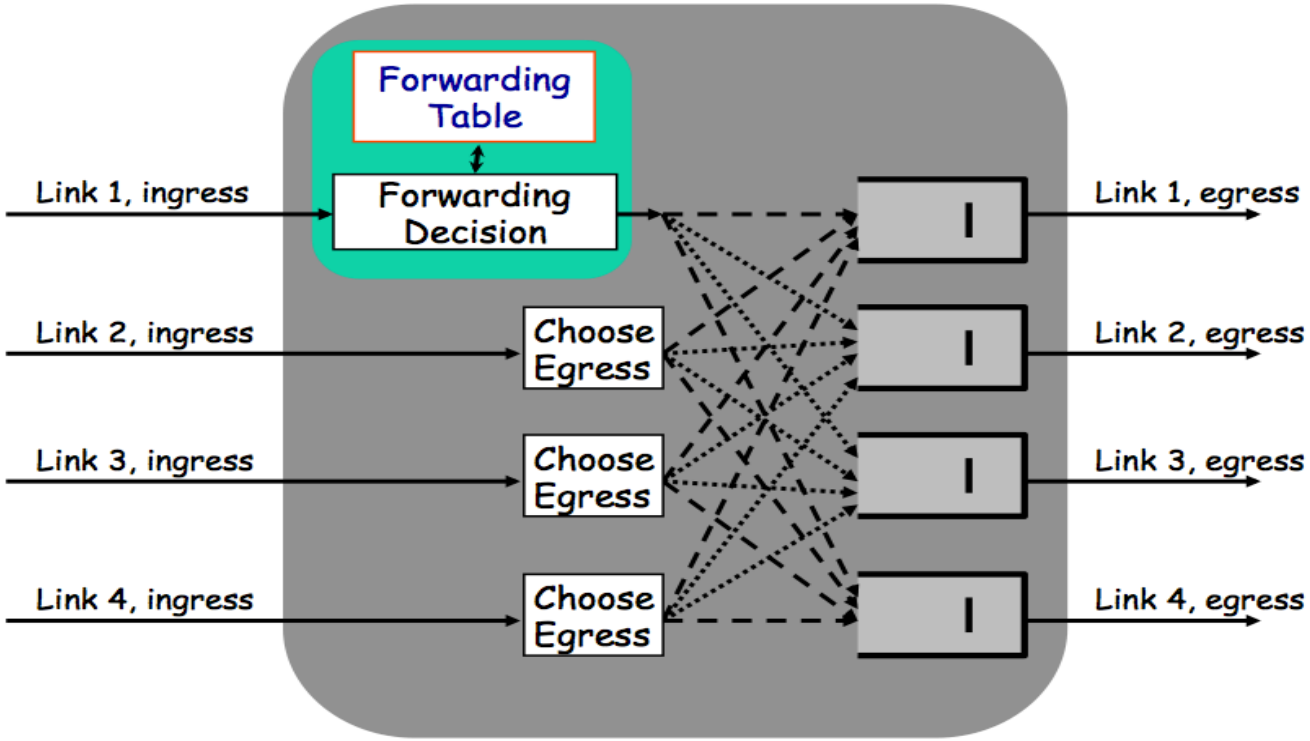
PAT Table

Src PC	Src IP Address	Src. Port # (Many Devices also Translate the SRC Port Number to a New Value)	Translated Src. IP Address	Destination IP Addres	Destination Port Number	Translated Destination IP Address
A	10.1.1.1	3759	200.1.1.1	100.1.1.1	80	100.1.1.1
B	10.1.1.2	2119	200.1.1.1	100.1.1.1	80	100.1.1.1

Control-plane VS. Data-plane

- Control-plane
  - Control the data flow by defining rules
  - E.g., the routing algorithm
- Data-plane
  - The data-path which copies data according to the rules
  - Performance critical
  - E.g., the IP forwarding process
- Interface: the routing table
  - Control-plane write the table, data-plane read the table

Inside a Router

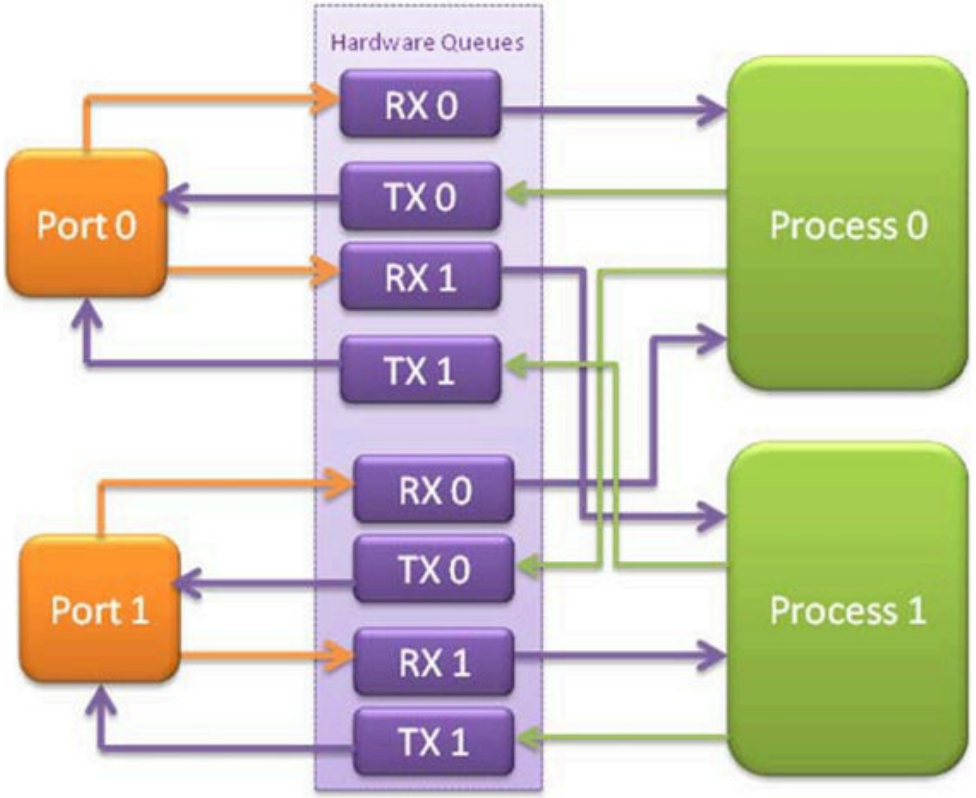


Forwarding an IP Packet

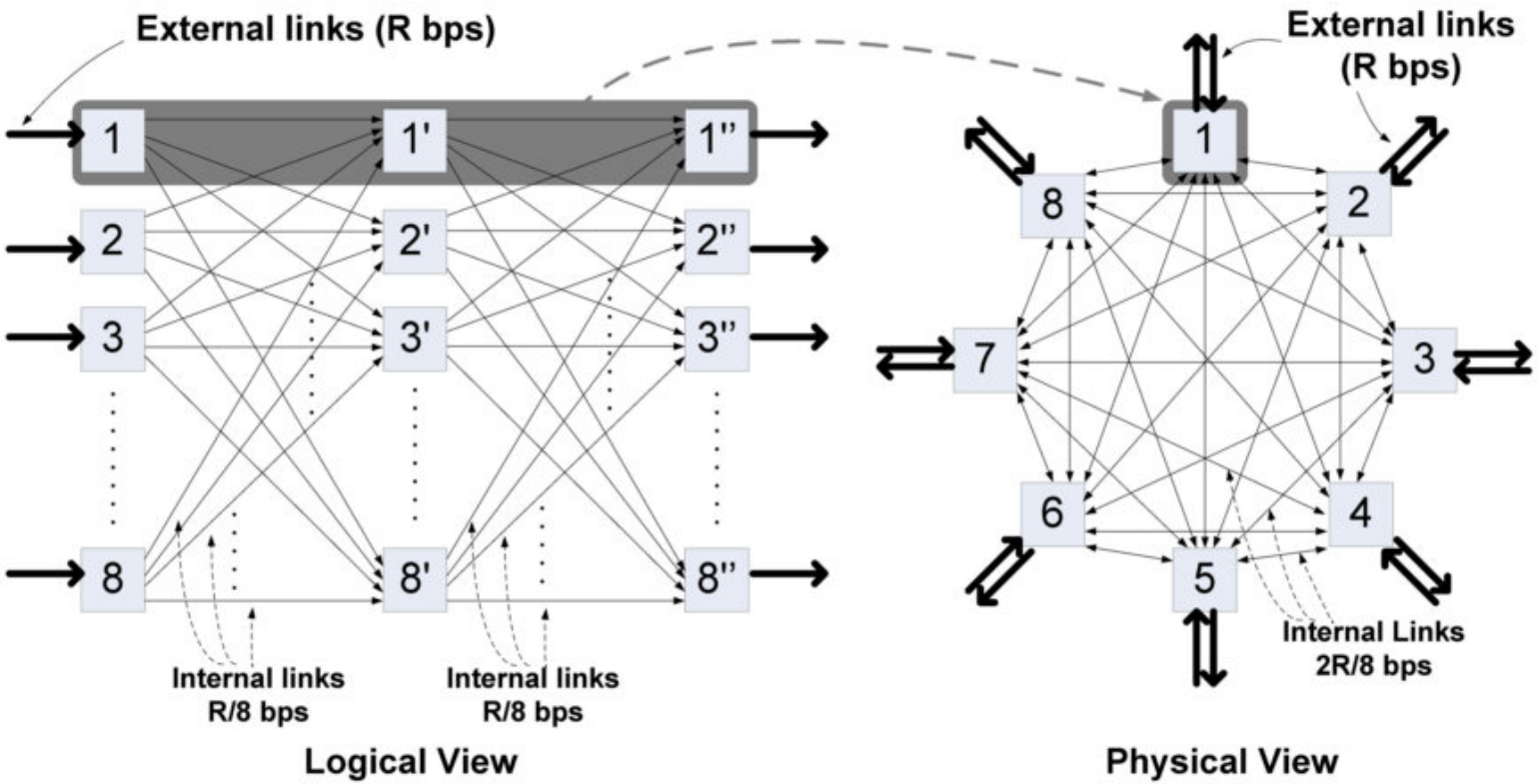
- Lookup packet's DST in forwarding table
  - If known, find the corresponding outgoing link
  - If unknown, drop packet
- Decrement TTL (Time To Live)
  - Drop packet if TTL is zero
- Update header checksum
- Forward packet to outgoing port
- Transmit packet onto link

Data-plane Case: Intel's DPDK

- DPDK: Data Plane Development Kit
- NIC
  - Has several ports
  - A port has RX/TX
- Processor
  - Read packets from RX
    - Polling
  - Find output port
  - Write packets to TX

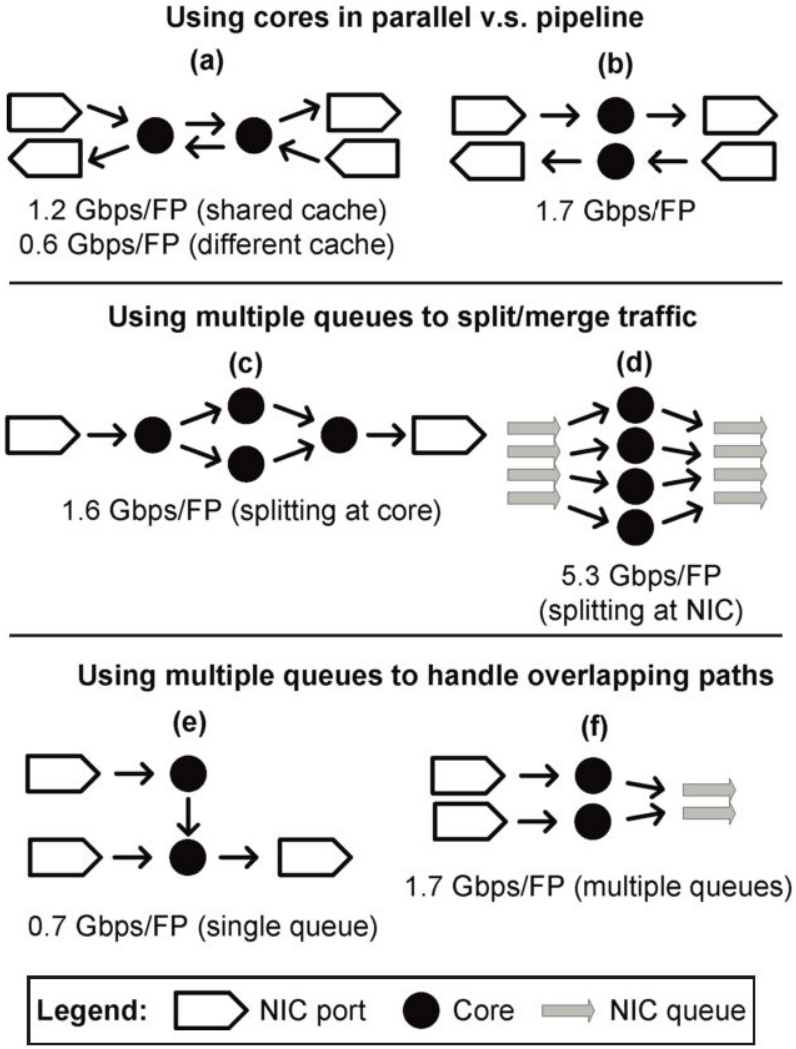


RouteBricks [SOSP'09]



RouteBricks [SOSP'09]

- The performance of parallel and pipeline



Reporting Network Layer Errors

- The buffers of the router were full, so the packet had to be discarded
- The buffers of the router are getting full - please stop sending so many packets
- The region identifier part of the target address does not exist
- The station identifier part of the target address does not exist
- The end type identifier was not recognized
- The packet is larger than the maximum transmission unit of the next link
- The packet hop limit has been exceeded

- What about sending report when checksum is error?

Reporting Network Layer Errors

- Cross layers error message
  - Originates in the network layers, is delivered to the end-to-end layer
  - Violating the separation of layers?
- Error reporting protocol: best-effort
  - Reliable protocol adds a lot
  - Can be thought of hints, not essential
  - E.g. ICMP (ping)
  - Hop limit exceeded
  - Learn the smallest MTU by "MTU exceeded" error

Mapping Internet to Ethernet

CASE: ETHERNET MAPPING

Case Study: Mapping Internet to Ethernet

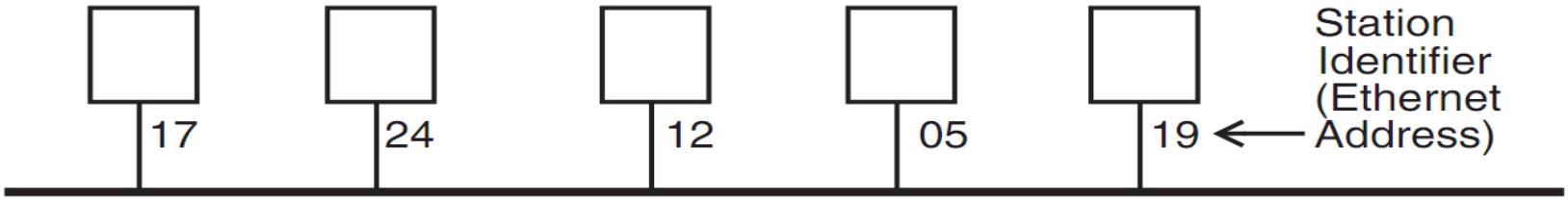
- Listen-before-sending rule, collision
- Ethernet: CSMA/CD
  - Carrier Sense Multiple Access with Collision Detection
- Ethernet type
  - Experimental Ethernet, 3mpbs
  - Standard Ethernet, 10 mbps
  - Fast Ethernet, 100 mbps
  - Gigabit Ethernet, 1000 mbps

Overview of Ethernet

- A half duplex Ethernet
  - The max propagation time is less than the 576 bit times, the shortest allowable packet
  - So that two parties can detect a collision together
  - If collision: wait random first time, exponential backoff if repeat
- A full duplex & point-to-point Ethernet
  - No collisions & the max length of the link is determined by the physical medium

leader	destination	source	type	data	checksum
64 bits	48 bits	48 bits	16 bits	368 to 12,000 bits	32 bits

Broadcast Aspects of Ethernet



- Broadcast network
  - Every frame is delivered to every station
  - (Compare with forwarding network)
- ETHERNET_SEND
  - Pass the call along to the link layer
- ETHERNET_HANDLE
  - Simple, can even be implemented in hardware

Broadcast Aspects of Ethernet

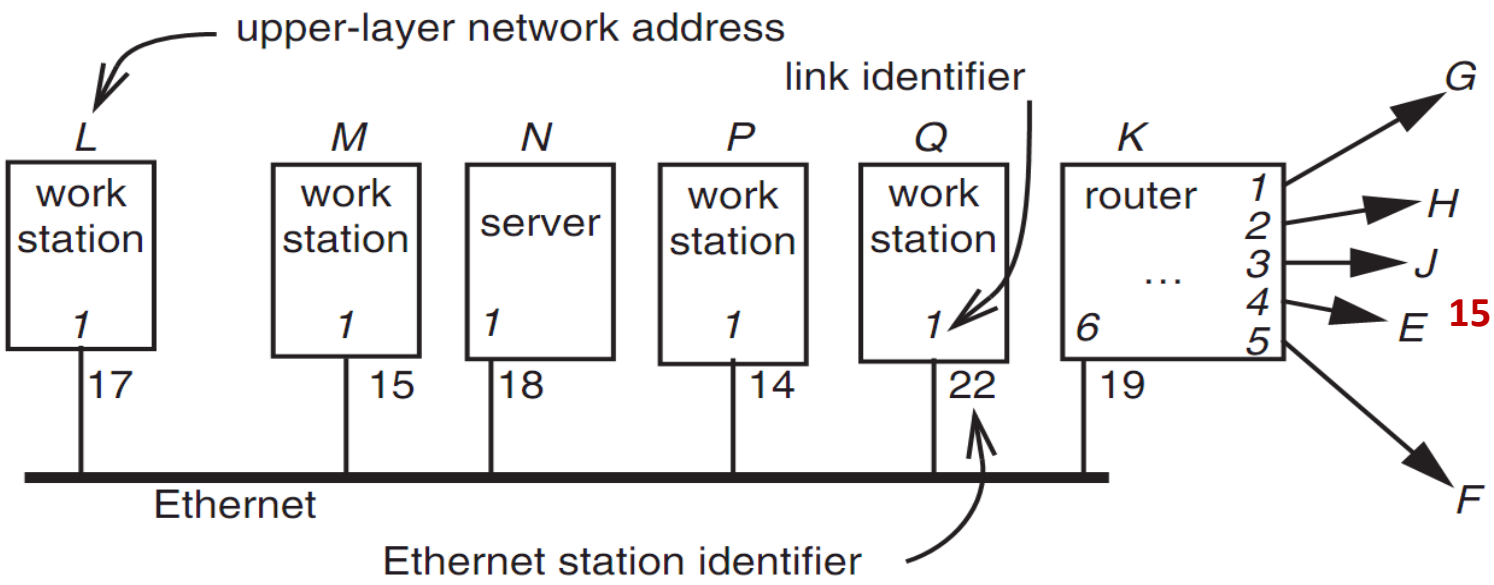
```
procedure ETHERNET_HANDLE (net_packet, length)
  destination ← net_packet.target_id
  if destination = my_station_id

    then
    GIVE_TO_END_LAYER (net_packet.data,
                       net_packet.end_protocol,
                       net_packet.source_id)
  else
    ignore packet
```

no need to do any forwarding

Broadcast Aspects of Ethernet

```
procedure ETHERNET_HANDLE (net_packet, length)
  destination ← net_packet.target_id
  if destination = my_station_id
      or destination = BROADCAST_ID
    then
    GIVE_TO_END_LAYER (net_packet.data,
                       net_packet.end_protocol,
                       net_packet.source_id)
  else
    ignore packet
```

Layer Mapping: Attach Ethernet to Forwarding Network



- L sends a RPC to N by sending to station 18 of link 1
- L sends a RPC to E by sending to K, E may have 15 as address, as well as M

Layer Mapping

- The Internet network layer
  - NETWORK_SEND (data, length, RPC, INTERNET, N)
  - NETWORK_SEND (data, length, RPC, ENET, 18)
- L must maintain a table

internet address	Ethernet/ station
M	enet/15
N	enet/18
P	enet/14
Q	enet/22
K	enet/19
E	enet/19

ARP (Address Resolution Protocol)

- NETWORK_SEND ("where is M?", 11, ARP, ENET, BROADCAST)
- NETWORK_SEND ("M is at station 15", 18, ARP, ENET, BROADCAST)
- L ask E's Ethernet address, E does not hear the Ethernet broadcast, but the router at station 19 does, and it sends a suitable ARP response instead
- Manage forwarding table as a cache

internet address	Ethernet/ station
M	enet/15

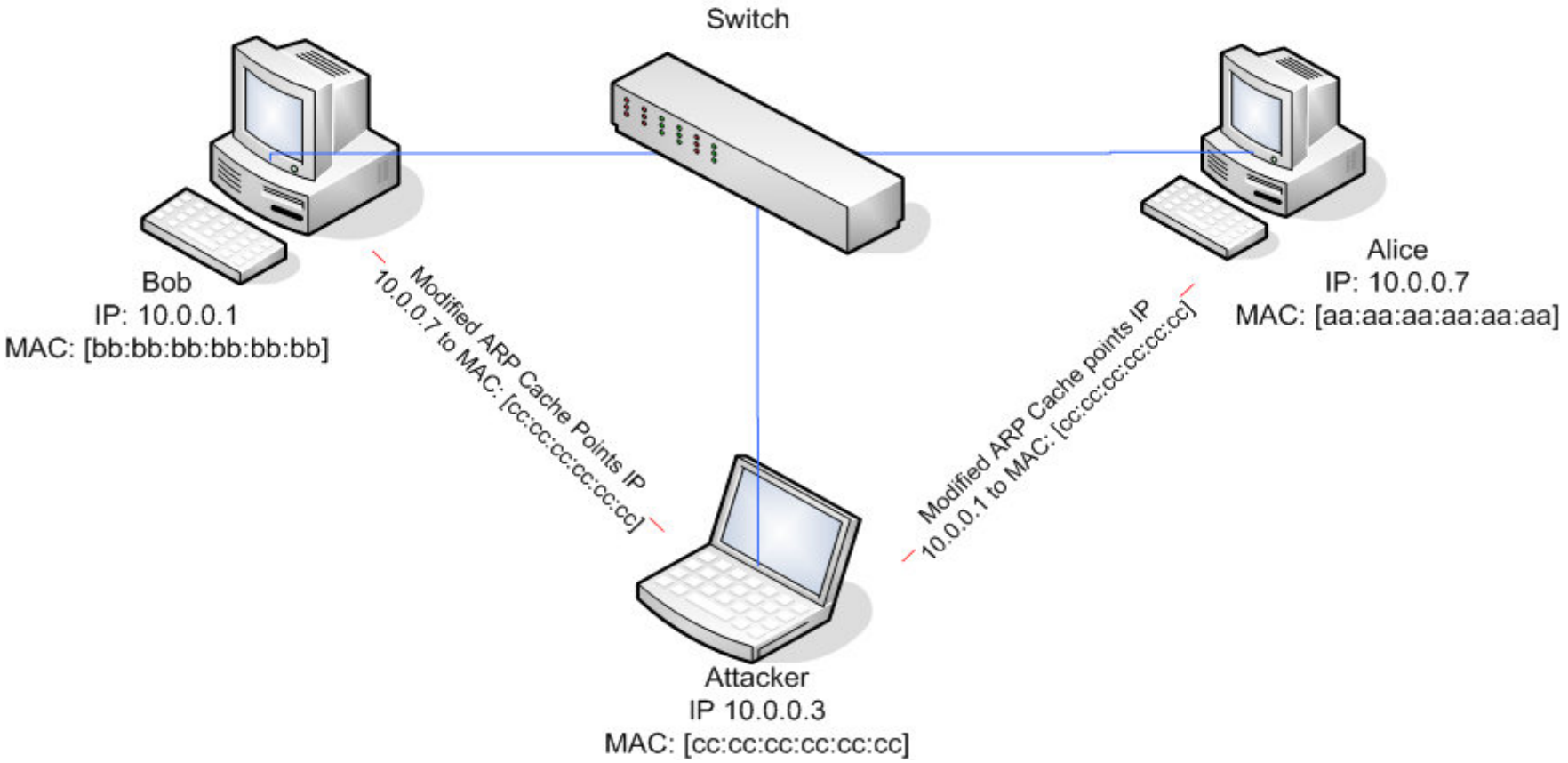
internet address	Ethernet/ station
M	enet/15
E	enet/19

ARP & RARP Protocol

Hardware Type (16 bits)		Protocol Type (16 bits)
HA Length (8 bits)	PA Length (8 bits)	Operation (16 bits)
Sender Hardware Address (Octets 0-3)		
Sender Hardware Address (Octets 4-5)		Sender Protocol Address (Octets 0-1)
Sender Protocol Address (Octets 2-3)		Target Hardware Address (Octets 0-1)
Target Hardware Address (Octets 2-5)		
Target Protocol Address (Octets 0-3)		

- Name mapping: IP address <-> MAC address

ARP Spoofing: A Design Flaw



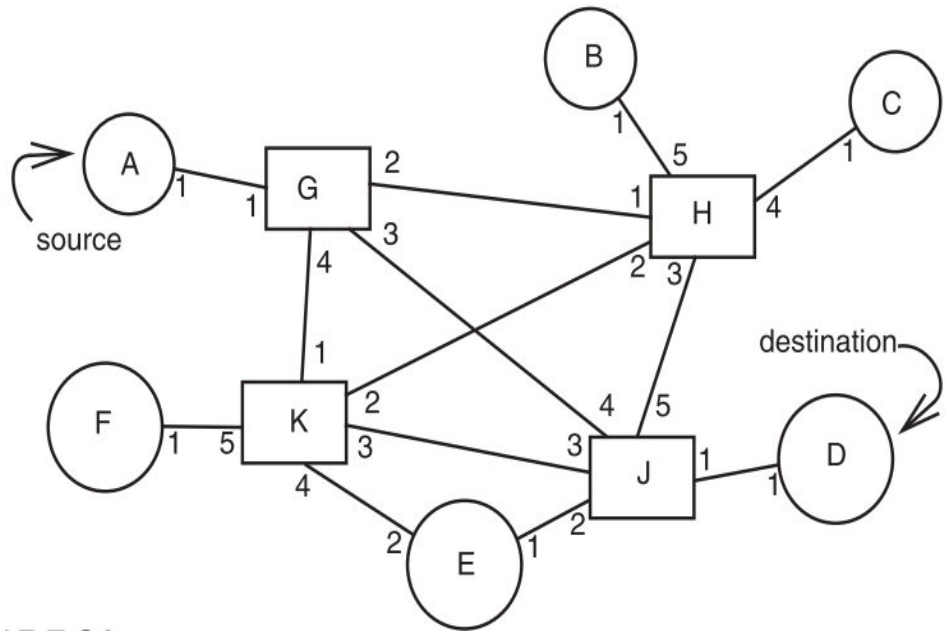
End-to-end Layer

Best-effort is not enough

Review: IP: Best-effort Network

- Best-effort network
  - If it cannot dispatch, may discard a packet
- Guaranteed-delivery network
  - Also called store-and-forward network, no discarding data
  - Work with complete messages rather than packets
  - Uses disk for buffering to handle peaks
  - Tracks individual message to make sure none are lost
- In real world
  - No absolute guarantee
  - Guaranteed-delivery: higher layer; best-effort: lower layer

Review: Path Vector Exchange



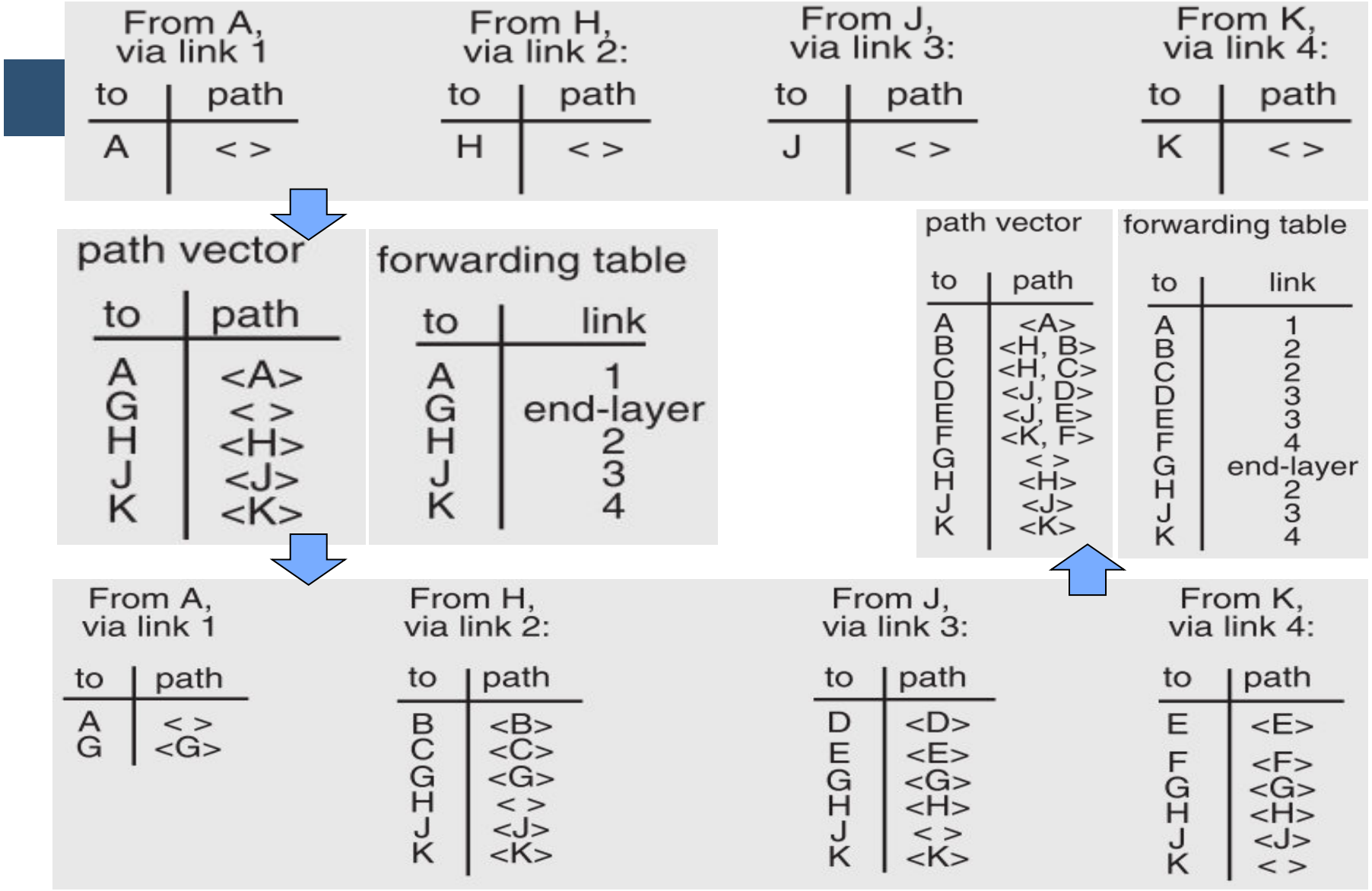
A

destination	link
A	end-layer
all other	1

G

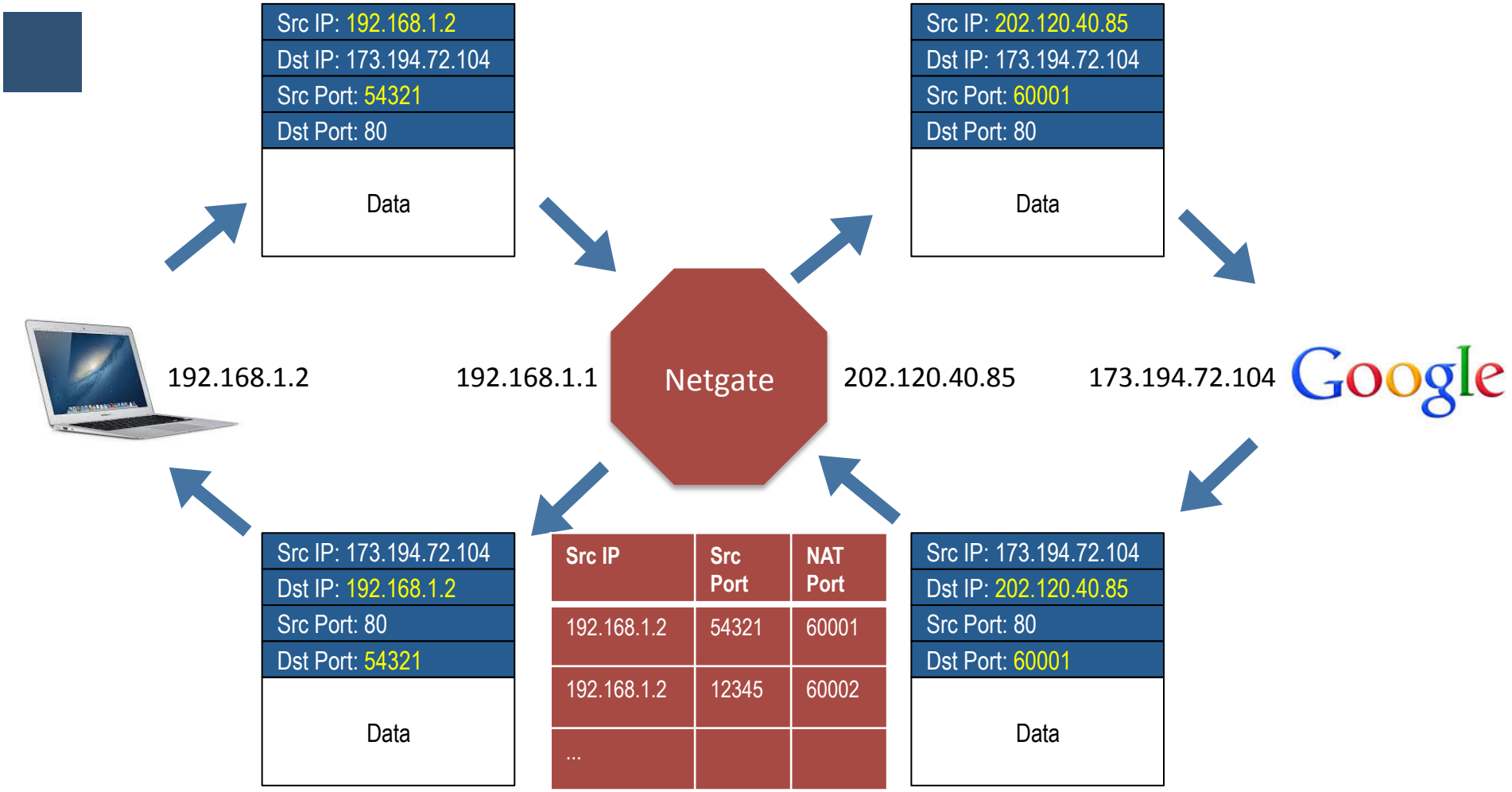
destination	link
A	1
B	2
C	2
D	3
E	4
F	4
G	end-layer
H	2
J	3
K	4

- Need coordination to ensure no loop


From A, via link 1
to | path
A | < >
From H, via link 2:
to | path
H | < >
From J, via link 3:
to | path
J | < >
From K, via link 4:
to | path
K | < >
path vector
to | path
A | <A>
G | < >
H | <H>
J | <J>
K | <K>
forwarding table
to | link
A | 1
G | end-layer
H | 2
J | 3
K | 4
path vector
to | path
A | <A>
B | <H, B>
C | <H, C>
D | <J, D>
E | <J, E>
F | <K, F>
G | < >
H | <H>
J | <J>
K | <K>
forwarding table
to | link
A | 1
B | 2
C | 2
D | 3
E | 3
F | 4
G | end-layer
H | 2
J | 3
K | 4
From A, via link 1
to | path
A | < >
G | <G>
From H, via link 2:
to | path
B | <B>
C | <C>
G | <G>
H | < >
J | <J>
K | <K>
From J, via link 3:
to | path
D | <D>
E | <E>
G | <G>
H | <H>
J | < >
K | <K>
From K, via link 4:
to | path
E | <E>
F | <F>
G | <G>
H | <H>
J | <J>
K | < >

Review: NAT (Network Address Translation)

- Private network
  - Public routers don't accept routes to network 10
- NAT router: bridge the private networks
  - Router between private & public network
  - Send: modify source address to temp public address
  - Receive: modify back by looking mapping table
- Limitations
  - Some end-to-end protocols place address in payloads
  - The translator may become the bottleneck
  - What if two private network merge?


Src IP: 192.168.1.2
Dst IP: 173.194.72.104
Src Port: 54321
Dst Port: 80
Data
Src IP: 202.120.40.85
Dst IP: 173.194.72.104
Src Port: 60001
Dst Port: 80
Data
192.168.1.2
192.168.1.1
Netgate
202.120.40.85
173.194.72.104
Google
Src IP: 173.194.72.104
Dst IP: 192.168.1.2
Src Port: 80
Dst Port: 54321
Data
Src IP
Src Port
NAT Port
192.168.1.2
54321
60001
192.168.1.2
12345
60002
...
Src IP: 173.194.72.104
Dst IP: 202.120.40.85
Src Port: 80
Dst Port: 60001
Data

Review: Control-plane VS. Data-plane

- Control-plane
  - Control the data flow by defining rules
  - E.g., the routing algorithm
- Data-plane
  - The data-path which copies data according to the rules
  - Performance critical
  - E.g., the IP forwarding process
- Interface: the routing table
  - Control-plane write the table, data-plane read the table

Review: Mapping IP to Ethernet

- The Internet network layer
  - NETWORK_SEND (data, length, RPC, INTERNET, N)
  - NETWORK_SEND (data, length, RPC, ENET, 18)
- L must maintain a table

internet address	Ethernet/ station
M	enet/15
N	enet/18
P	enet/14
Q	enet/22
K	enet/19
E	enet/19

ARP (Address Resolution Protocol)

- NETWORK_SEND ("where is M?", 11, ARP, ENET, BROADCAST)
- NETWORK_SEND ("M is at station 15", 18, ARP, ENET, BROADCAST)
- L ask E's Ethernet address, E does not hear the Ethernet broadcast, but the router at station 19 does, and it sends a suitable ARP response instead
- Manage forwarding table as a cache

internet address	Ethernet/ station
M	enet/15

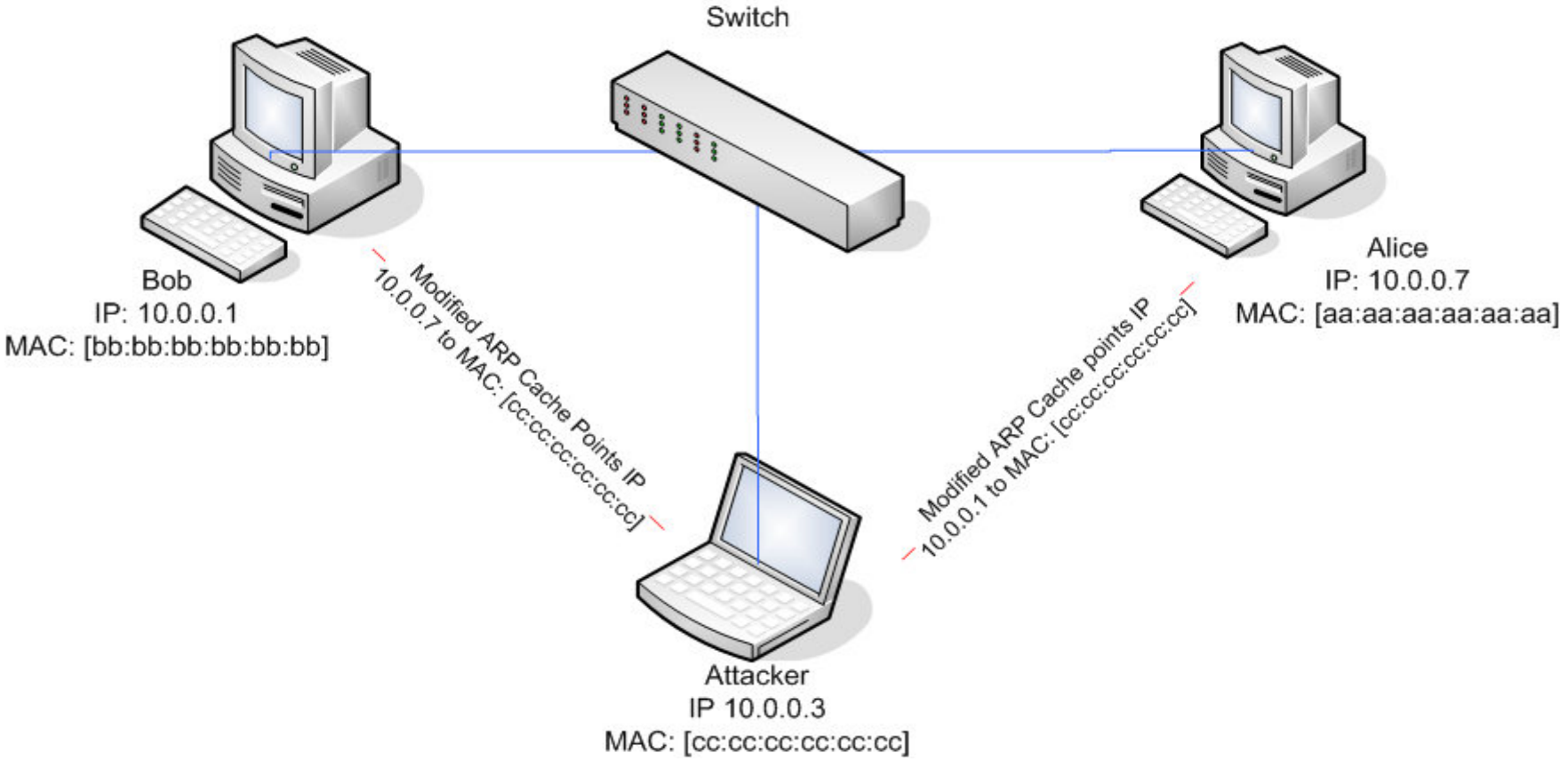
internet address	Ethernet/ station
M	enet/15
E	enet/19

ARP & RARP Protocol

Hardware Type (16 bits)		Protocol Type (16 bits)
HA Length (8 bits)	PA Length (8 bits)	Operation (16 bits)
Sender Hardware Address (Octets 0-3)		
Sender Hardware Address (Octets 4-5)		Sender Protocol Address (Octets 0-1)
Sender Protocol Address (Octets 2-3)		Target Hardware Address (Octets 0-1)
Target Hardware Address (Octets 2-5)		
Target Protocol Address (Octets 0-3)		

- Name mapping: IP address <-> MAC address

ARP Spoofing: A Design Flaw



END-TO-END LAYER

The End-to-end Layer

- Network layer has no guarantees on:
  - Delay - Order of arrival
  - Certainty of arrival - Accuracy of content
  - Right place to deliver

- End-to-end layer
  - No single design is likely to suffice
  - Transport protocol for each class of application

Famous Transport Protocols

- UDP (User Datagram Protocol)
  - Be used directly for some simple applications
  - Also be used as a component for other protocols
- TCP (Transmission Control Protocol)
  - Keep order, no missing, no duplication
  - Provision for flow control
- RTP (Real-time Transport Protocol)
  - Built on UDP
  - Be used for streaming video or voice, etc.
- No "one size fits all"

Assurance of End-to-end Protocol

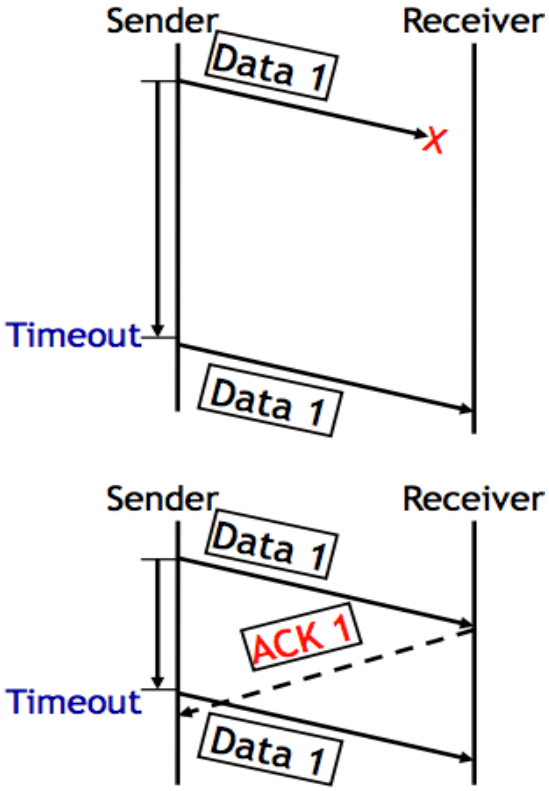
1. Assurance of at-least-once delivery
2. Assurance of at-most-once delivery
3. Assurance of data integrity
4. Assurance of stream order & closing of connections
5. Assurance of jitter control
6. Assurance of authenticity and privacy
7. Assurance of end-to-end performance

1. Assurance of At-least-once Delivery

- RTT (Round-trip time)
  - *to_time + process_time + back_time (ack)*
- At least once on best effort network
  - Send packet with nonce
  - Sender keeps a copy of the packet
  - Resend if timeout before receiving acknowledge
  - Receiver acknowledges a packet with its nonce
- Try limit times before returning error to app

1. Assurance of At-least-once Delivery

- Dilemma
  - 1. The data was not delivered
  - 2. The data was delivered, but no ACK received
  - No way to know which situation
- At-least-once delivery
  - No absolute assurance for at-least-once
  - Ensure if it is possible to get through, the message will get through eventually
  - Ensure if impossible to confirm delivery, app will know
  - No assurance for no-duplication



How to Decide Timeout?

- Fixed timer: dilemma of fixed timer
  - Too short: unnecessary resend
  - Too long: take long time to discover lost packets
- Adaptive timer
  - E.g., adjust by currently observed RTT, set timer to 150%
  - Exponential back-off: doubling from a small timer
- NAK (Negative AcKnowledgment)
  - Receiver sends a message that lists missing items
  - Receiver can count arriving segments rather than timer
  - Sender can have no timer (only once per stream)

Fixed Timer is Evil

- Fixed Timers Lead to Congestion Collapse in NFS
- Emergent Phase Synchronization of Periodic Protocols
- Wisconsin Time Server Meltdown

Congestion Collapse in NFS

- Using at-least-once with stateless interface
  - Persistent client: repeat resending forever
  - Server: FIFO
- Timeout when queuing and the client will resend
  - The server re-execute the resent request and waste time
  - When the queue becomes longer, waste more time and collapse
- Lesson: *Fixed timers are always a source of trouble, sometimes catastrophic trouble*

Emergent Phase Synchronization of Periodic Protocols

- Periodic polling
  - E.g. picking up mail, sending "are-you-there?"
  - A workstation sends a broadcast packet every 5 minutes
  - All workstations try to broadcast at the same time
- Each workstation
  - Send a broadcast
  - Set a fixed timer
- *Lesson: Fixed timers have many evils. Don't assume that unsynchronized periodic activities will stay that way*

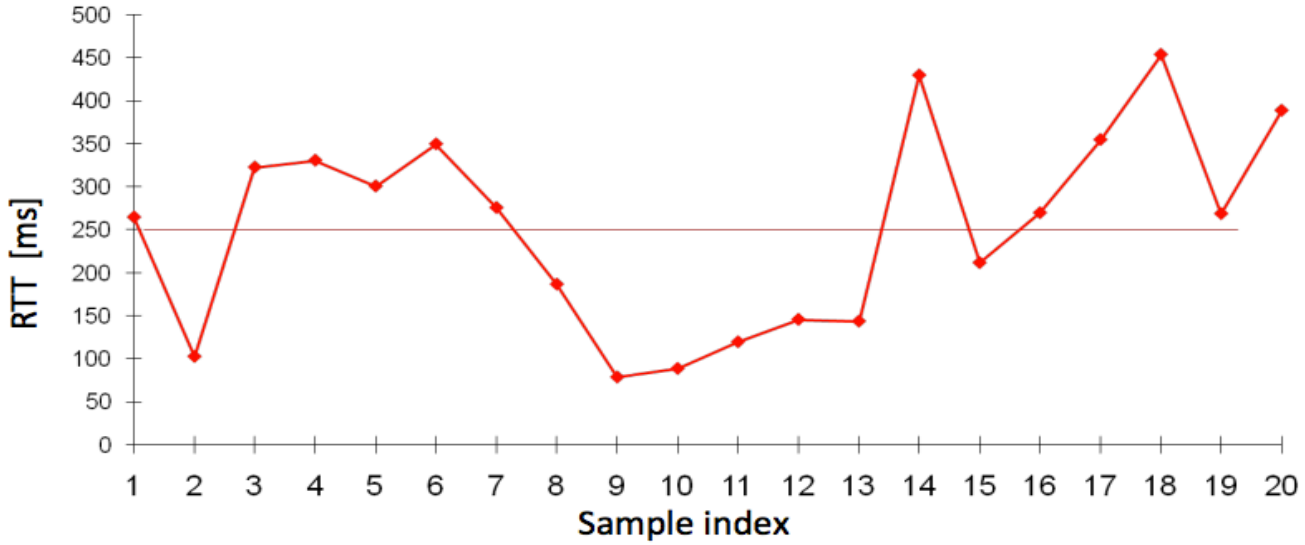
Wisconsin Time Server Meltdown

- NETGEAR added a feature to wireless router
  - Logging packets -> timestamp -> time server (SNTP) -> name discovery -> 128.105.39.11
  - Once per second until receive a response
  - Once per minute or per day after that
- Wisconsin Univ.
  - On May 14, 2003, at about 8:00 a.m
  - From 20,000 to 60,000 requests per second, filtering 23457
  - After one week, 270,000 requests per second, 150Mbps

Wisconsin Time Server Meltdown

- Lesson(s)
  - Fixed timers, again
  - Fixed Internet address
  - The client implements only part of a protocol
    - There is a reason for features likes the "*go away*" response in SNTP

RTT Could be Highly Variable



Example from a TCP connection over a wide-area wireless link
Mean RTT = 0.25 seconds; Std deviation = 0.11 seconds!

Can't set timeout to an RTT sample; need to consider variations

Calculating RTT and Timeout (in TCP)

- Exponentially Weighted Moving Average
  - Estimate both the average *rtt_avg* and the deviation *rtt_dev*
  - Procedure *calc_rtt*(rtt_sample)
    - `rtt_avg = a*rtt_sample + (1-a)*rtt_avg; /* a = 1/8 */`
    - `dev = absolute(rtt_sample – rtt_avg);`
    - `rtt_dev = b*dev + (1-b)*rtt_dev;  /* b = 1/4 */`
  - Procedure *calc_timeout(rtt_avg, rtt_dev)*
    - `Timeout = rtt_avg + 4*rtt_dev`

2. Assurance of At-most-once Delivery

- At-least-once delivery
  - Remember state at the sending side
  - Tends to generate duplicated requests
- At-most-once delivery
  - Maintains a table of nonce at the receiving side
  - The table may grow indefinitely
    - Space and search time
    - Tombstones (something that cannot be deleted *forever*)
  - Another way: Make the application tolerate duplicated requests
    - Recall the NFS collapse case: server wastes time to execute duplicated requests

Duplicate Suppression

- Monotonically increasing sequence number
  - Receiver discards smaller nonce, only holds the last nonce, one per sender (tombstone)
- Use a different port for each new request
  - Should never reuse the old port number (the old port is now tombstone)
- Accept the possibility of making a mistake
  - E.g., if sender always gives up after five RTT (cannot ensure at-least-once), then receiver can safely discard nonces that are older than five RTT
  - It is possible that a packet finally shows up after long delay (solution: wait long time)
- Receiver crashes and restarts: lose the table
  - One solution is to use a new port number each time the system restarts
  - Another is to ignore all packets until the number of RTT has passed since restarting, if sender tries limit times
- Anyway, duplicate suppression makes the system complex

3. Assurance of Data Integrity

- Data integrity
  - Receiver gets the same contents as sender
- Reliable delivery protocol
  - Sender: adds checksum to the end-to-end layer
  - Receiver: recalculates the checksum, discards if not match
- Is it redundant since link layer provides checksum?
  - Other errors, e.g., in memory copying
- The assurance is not absolute
  - What if a packet is misdelivered and the receiver ACK?

4. Segments and Reassembly of Long Messages

- Bridge the difference between message and MTU
  - Message length: determined by application
  - MTU: determined by network
- Segment contains ID for where it fits
  - E.g., "message 914, segment 3 of 7"
  - Can be used for *at-least-once* and *at-most-once* delivery
- Reassembly
  - Out-of-order, mingled with other message's segments
  - Allocating a buffer large enough to hold the message
  - Keeping a checklist for segments not arrived

When Out of Order. . .

- Solution-1: Receiver only ACK in order packets, discards others
  - Waste of bandwidth
- Solution-2: ACK every packet and hold early packets in buffer, release the buffer when all in order
  - Need using large buffer when waiting for a bad packet
- Solution-3: Combine the two above
  - Discard if buffer is full
  - New problem: how much buffer?
- Speedup for common case
  - NAK to avoid timeout
  - If NAKs are causing duplicates, stop NAKs
- *TCP is based on ACK, not NAK*

Closing of Connections

- Open a stream
  - Create a record to keep track of which elements have been sent, received, acknowledged
- Close a stream
  - When finish, it needs to report an end-of-stream
  - Both ends need to agree last element is OK and then close
  - 1. Alice sends close request to Bob with stream record ID
  - 2. Bob checks and agrees, sends a close ACK
  - 3. Alice receives ACK, turn off sender, discard record
  - 4. Alice sends "all done" to Bob **<- What if duplicate?**
  - 5. Bob receives "all done" and discard stream record

5. Assurance of Jitter Control

- Real-time
  - When reliability is less important than timely delivery
  - A few error in a movie may not be noticed
  - Jitter: variability in delivery time
- Strategy
  - Basic: delay all arriving segments

5. Assurance of Jitter Control

- Measure the distribution of delays in a chart showing delay time vs. frequency of that delay
- Choose an acceptable frequency of delivery failure
- Determine D_{long} that longer than 99% delay
- Determine the shortest delay, D_{short}
- Calculate number of segment buffer:

$$\text{Number of segment buffers} = \frac{D_{\text{long}} - D_{\text{short}}}{D_{\text{headway}}}$$

  - $D_{headway}$ is average delay between arriving segments

6. Assurance of Authenticity and Privacy

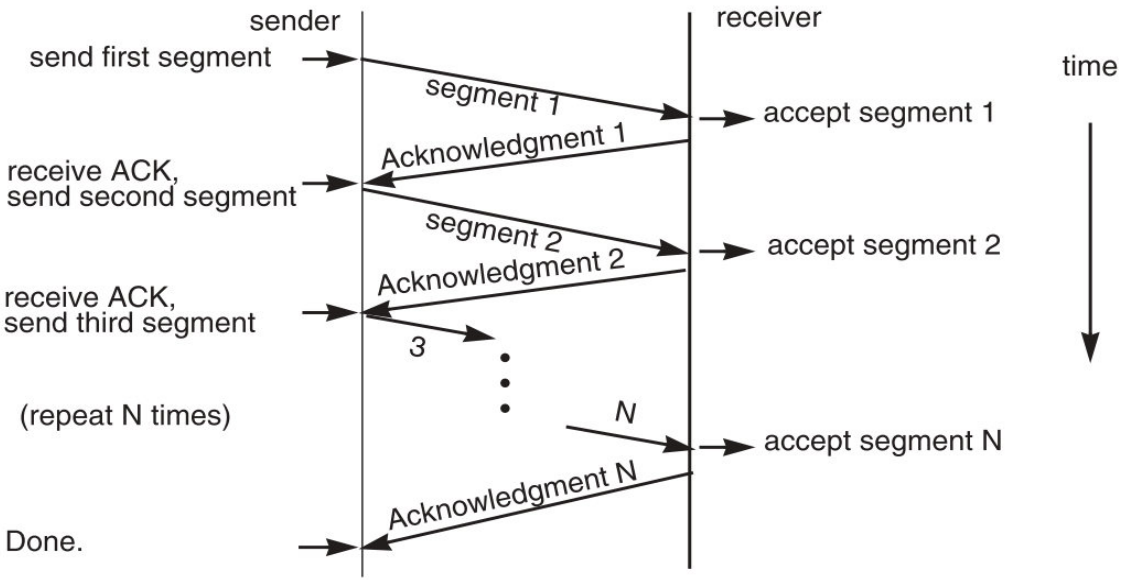
- Internet is dangerous
  - Hostile intercepts and maliciously modifies packets
  - Violate a protocol with malicious intent
- Key-based mathematical transformations to data
  - Sign and verify: establish the authenticity of the source and integrity of contents
  - Encrypt and decrypt: maintain privacy of contents
- Consideration
  - False sense of security, worse than no assurance

6. Security: Asymmetric Encryption

- Public Key VS. Private Key
  - Public key: encrypt to identify *reader* (only me can read this)
  - Private key: encrypt to identify *writer* (yes, it's me who wrote this)
  - Poor performance, so just used to exchange symmetric key
- Questions
  - What is a certificate? Why using a CA (Certificate Authority)?
  - How to exchange a symmetric key in HTTPS or SSH?
  - What is the root of trust?

7. End-to-end Performance

- Next lecture



Congestion Control

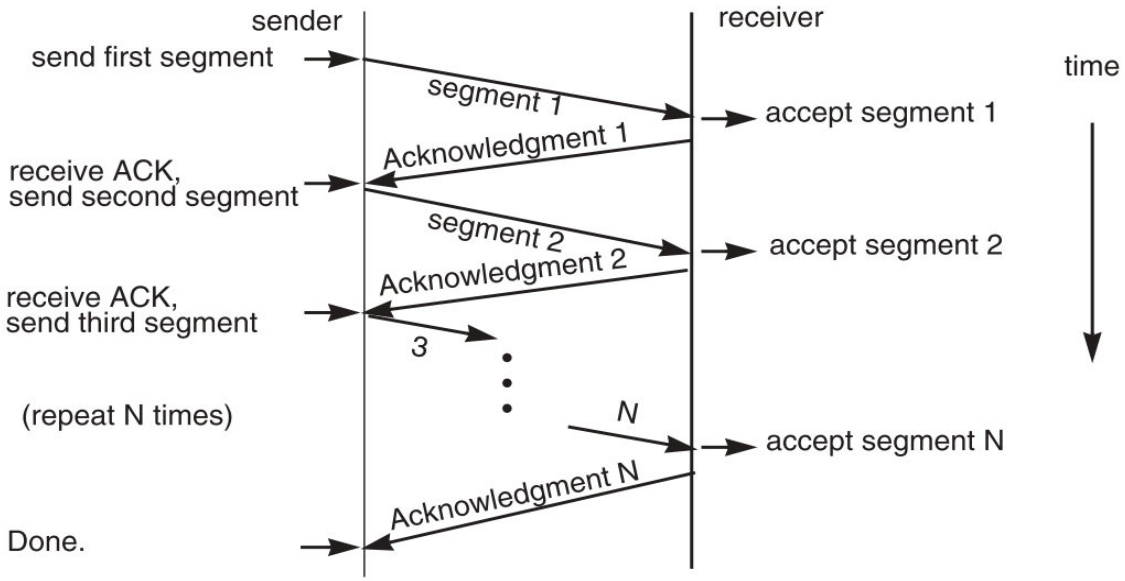
Sliding window & queue management

Review: Assurance of End-to-end Protocol

1. Assurance of at-least-once delivery
2. Assurance of at-most-once delivery
3. Assurance of data integrity
4. Assurance of stream order & closing of connections
5. Assurance of jitter control
6. Assurance of authenticity and privacy
7. Assurance of end-to-end performance

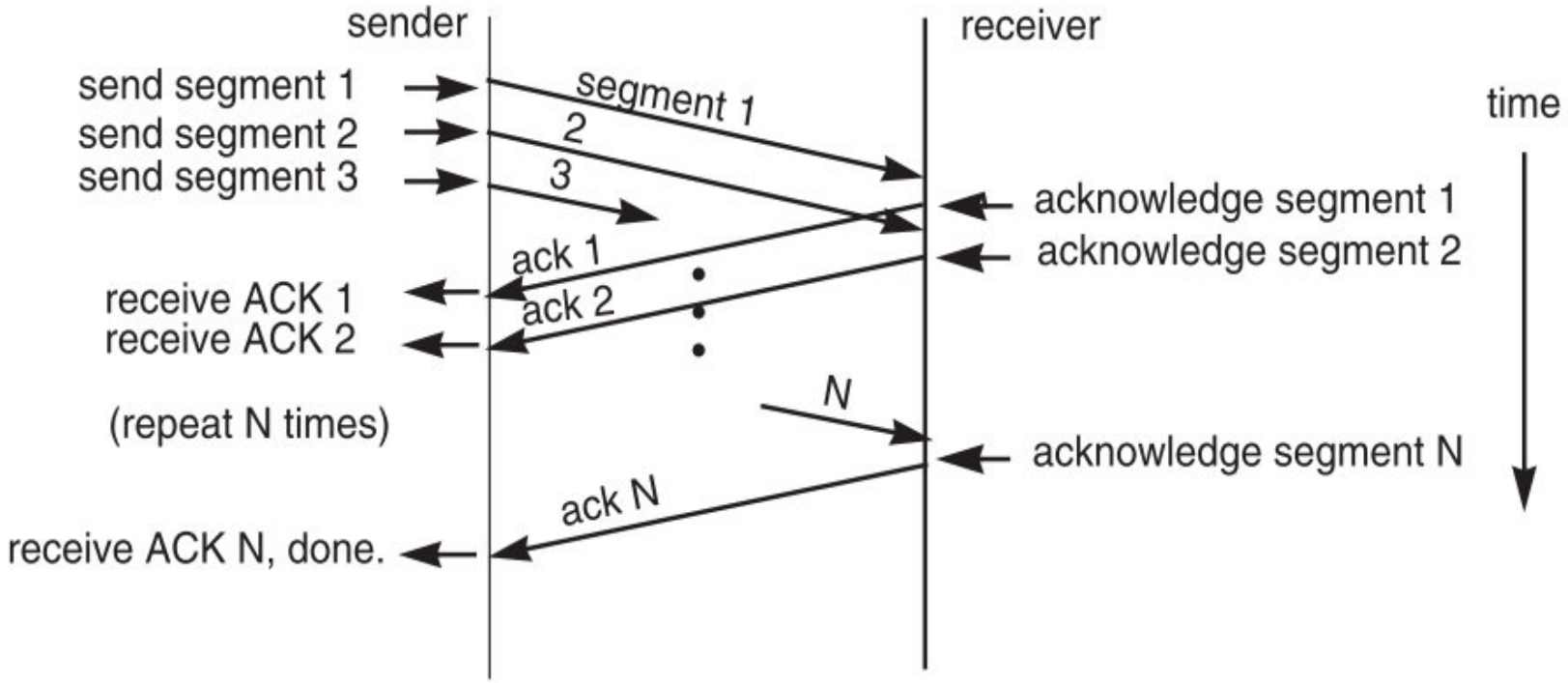
7. End-to-end Performance

- Multi-segment message questions
  - Trade-off between complexity and performance
  - Lock-step protocol



Overlapping Transmissions

- Pipelining technique

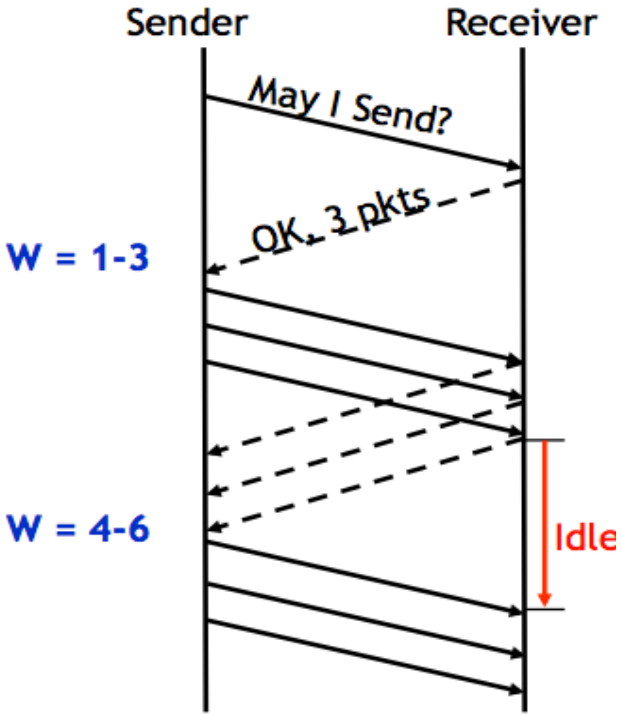


Overlapping Transmissions: Problems

- Packets or ACK may be lost
  - Sender holds a list of segments sent, check it off when receives ACK
  - Set a timer (according to RTT) for last segment

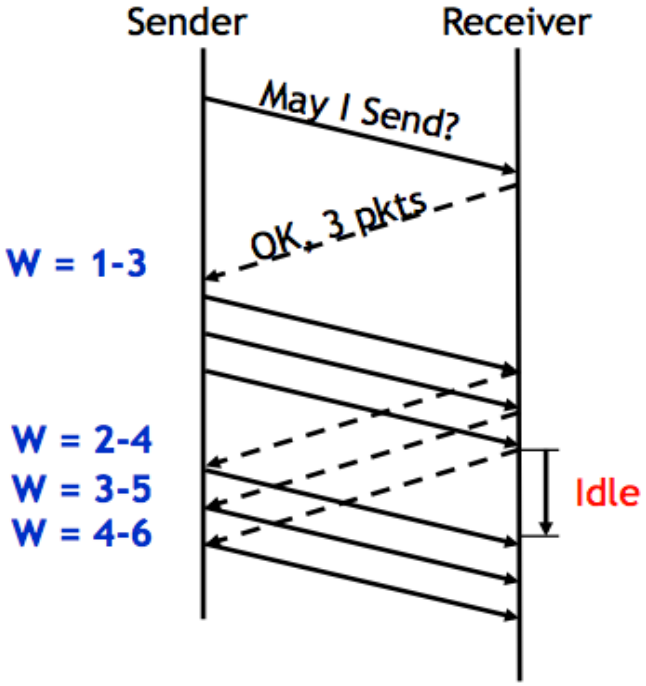
- If list of missing ACK is empty, OK
- If timer expires, resend packets and another timer

Fixed Window



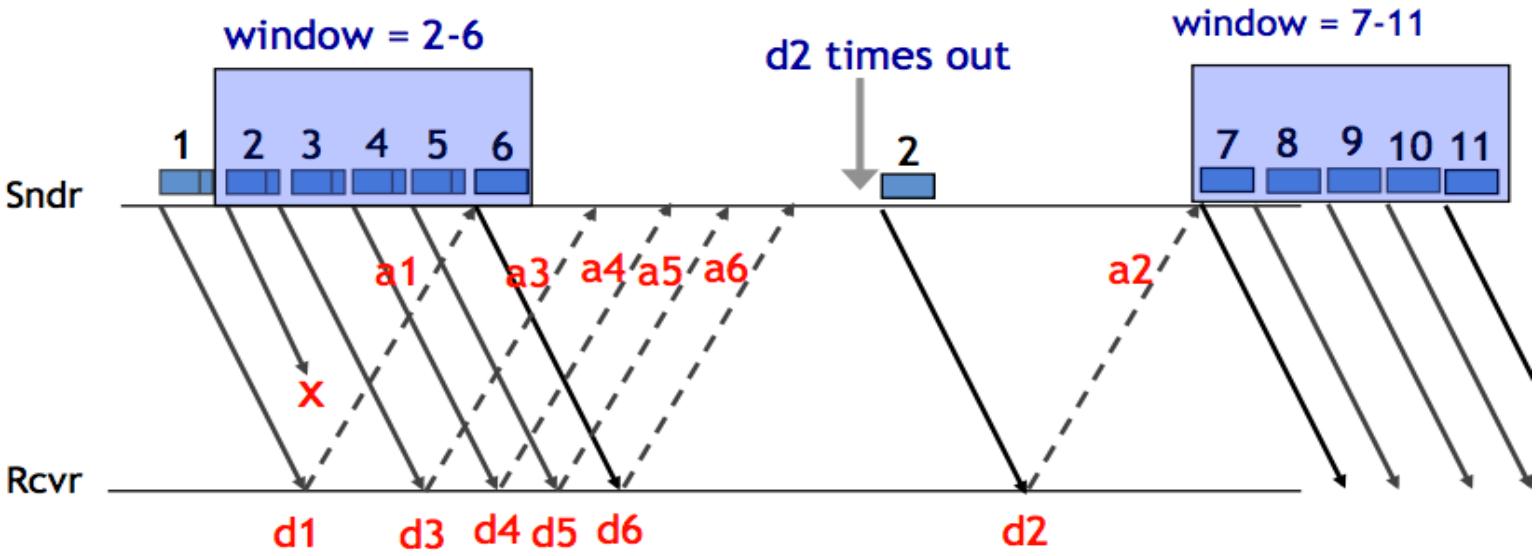
- Receiver tells the sender a window size
- Sender sends window
- Receiver acks each packet as before
- Window advances when all packets in previous window are acked
  - E.g., packets 4-6 sent, after 1-3 ack'd
- If a packet times out -> resend packets
- Still much idle time

Sliding Window



- Sender advances the window by 1 for each in-sequence ACK it receives
  - Reduces idle periods
  - Pipelining idea
- But what's the correct value for the window?
  - We'll revisit this question
  - First, we need to understand windows

Handling Packet Loss



Sender advances the window on arrivals of in-sequence acks

→ Can't advance on a3's arrival

Recovery of Lost Data Segments with Windows

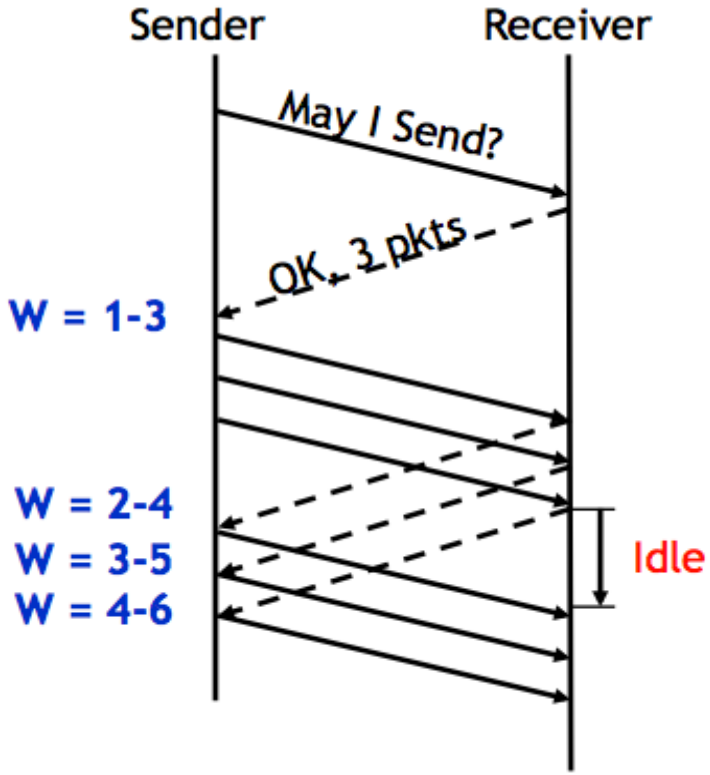
- When to take action on sender's checklist?
  - Solution-1: Timestamp and timeout
    - Associate each segment in the list with a timestamp
    - After more than one RTT, resend
  - Solution-2: Numbering segments
    - Receiver sends a NAK
- If sender resends a segment, should the available window increase?
  - Dilemma: original segment is lost? ACK is lost?
  - Presume receiver discard duplicated segments
  - Same answer in both cases: no increase

Recovery of Lost Data Segments with Windows

- What if a permission message is lost?
  - No data will be received
- Resend permission message?
  - May get twice as much
  - Incremental value is fragile -> cumulative total is better
    - E.g., using "send 1-3" instead of "send 3 more"
  - Similar with "in" and "out" pointer in bounded-buffer
- Rate-matching problem
  - Blizzard of packets arising from a newly-opened window
  - Need cooperation between end-to-end protocol & network forwarders

Chose the Right Window Size

- If window is too small
  - Long idle time
  - Underutilized network
- If window too large
  - Congestion



Sliding Window Size

window size ≥ round-trip time × bottleneck data rate

- Sliding window with one segment in size
  - Data rate is window size / RTT
- Enlarge window size to bottleneck data rate
  - Data rate is window size / RTT
- Enlarge window size further
  - Data rate is still bottleneck
  - Larger window makes no sense

- Receive 500 KBps
- Sender 1 MBps
- RTT 70ms
- A segment carries 0.5 KB

- Sliding window size = 35KB (70 segment)

Self-pacing: Sliding Window Size

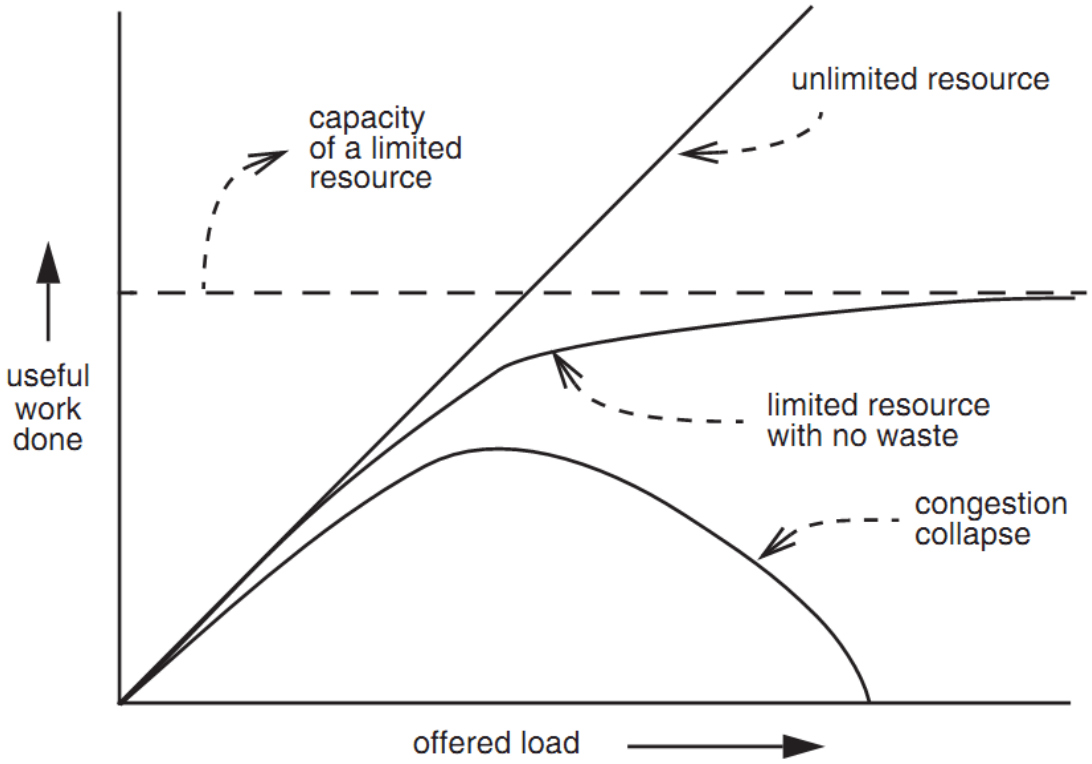
- Although the sender doesn't know the bottleneck, it is sending at exactly that rate
- Once sender fills a sliding window, cannot send next data until receive ACK of the oldest data in the window
- The receiver cannot generate ACK faster than the network can deliver data elements
- RTT estimation still needed

CONGESTION CONTROL

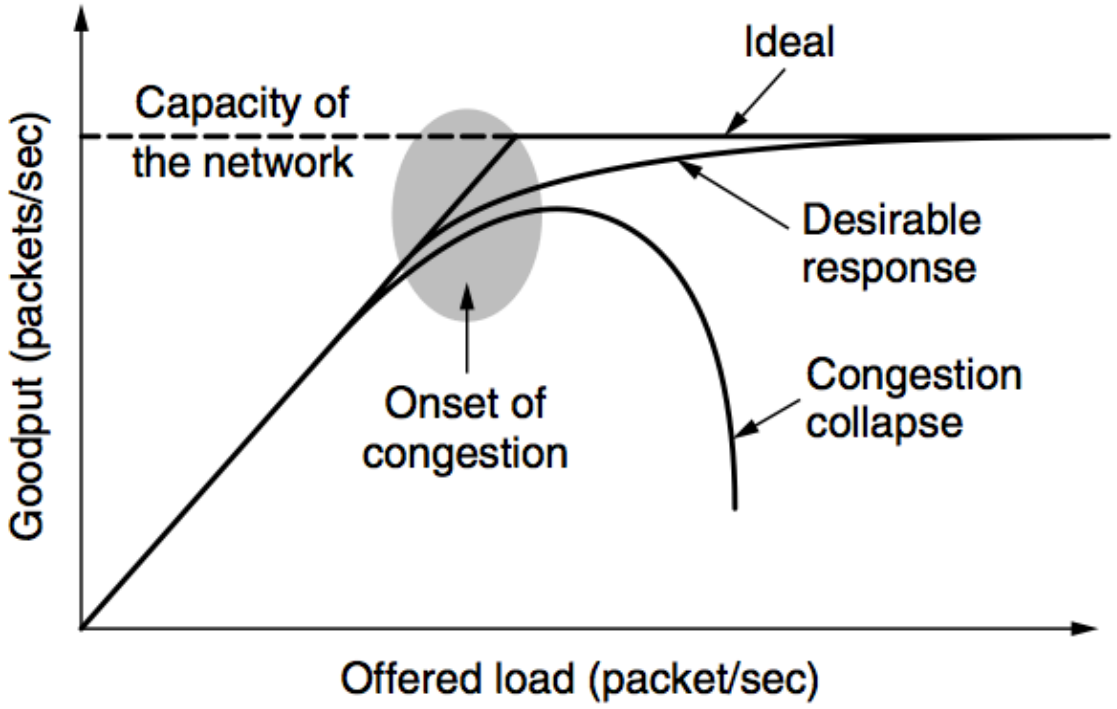
Congestion

- Definition: Too many packets present in (a part of) the network causes packet delay and loss that degrades performance.
- Network & e2e layers *share the responsibility* for handling congestion
- 1. Network layer
  - Directly experiences the congestion
  - Ultimately determine what to do with the excess packets
- 2. End-to-end layer
  - Control to reduce the sending rate, is the most effective way

Congestion Collapse

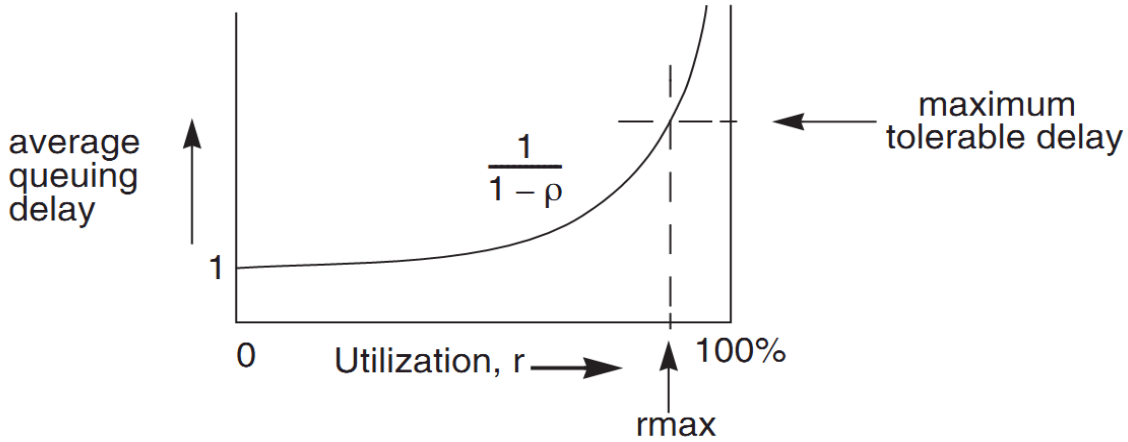


Network Congestion



Congestion Control

- Requires cooperation of more than one layer



- A shared resource, and demands from several statistically independent sources, there will be fluctuations in the arrival of load, and thus in the length of queue and time spent waiting

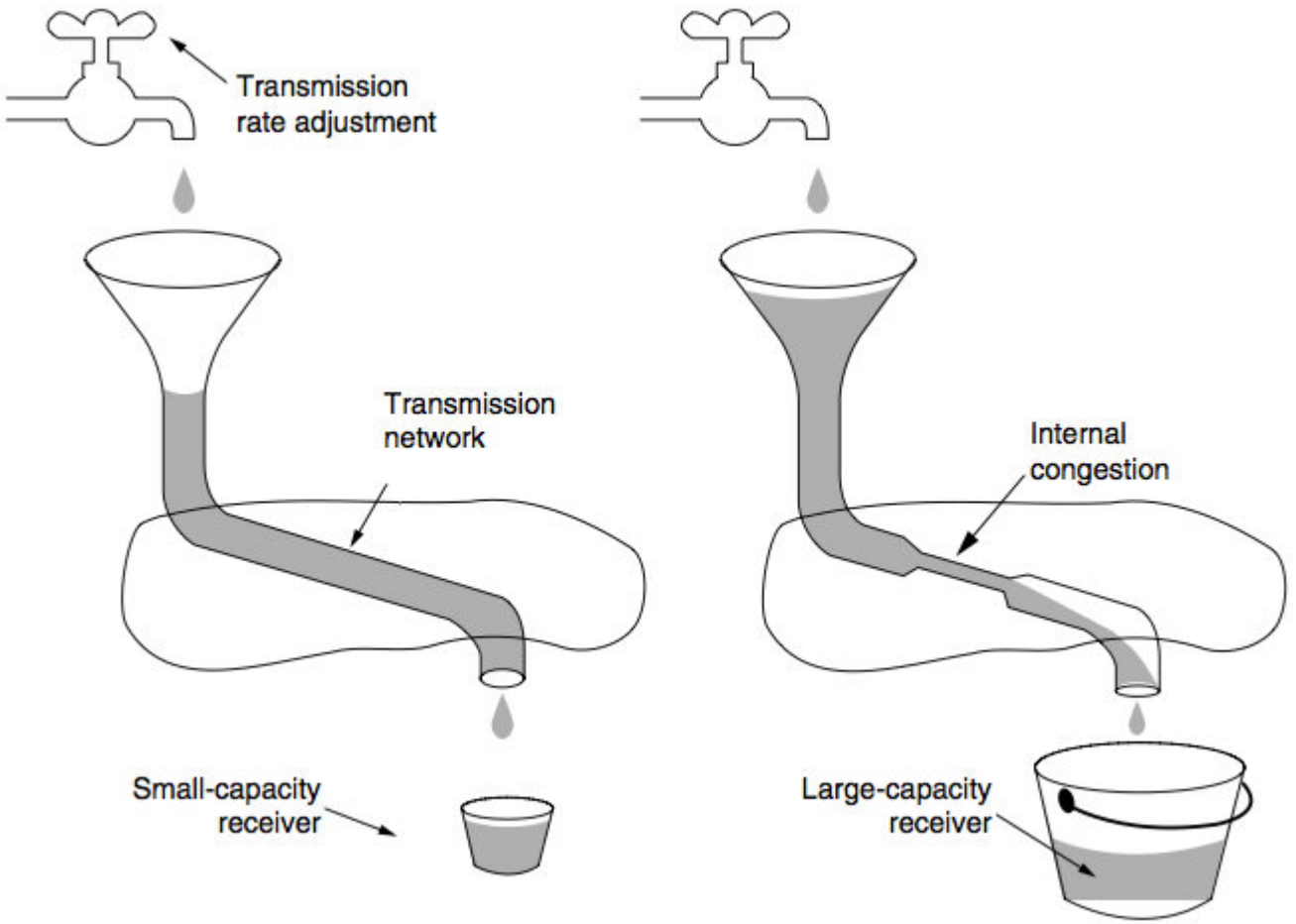
Why Congest?

- If all of a sudden, streams of packets begin arriving on three or four input lines and all need the same output line, a queue will build up
- If there is insufficient memory to hold all of them, packets will be lost
- Adding more memory may help up to a point, but
  - Nagle (1987) realized that if routers have an infinite amount of memory, congestion gets worse, not better.
  - This is because by the time packets get to the front of the queue, they have already timed out (repeatedly) and duplicates have been sent

Congestion Control vs. Flow Control

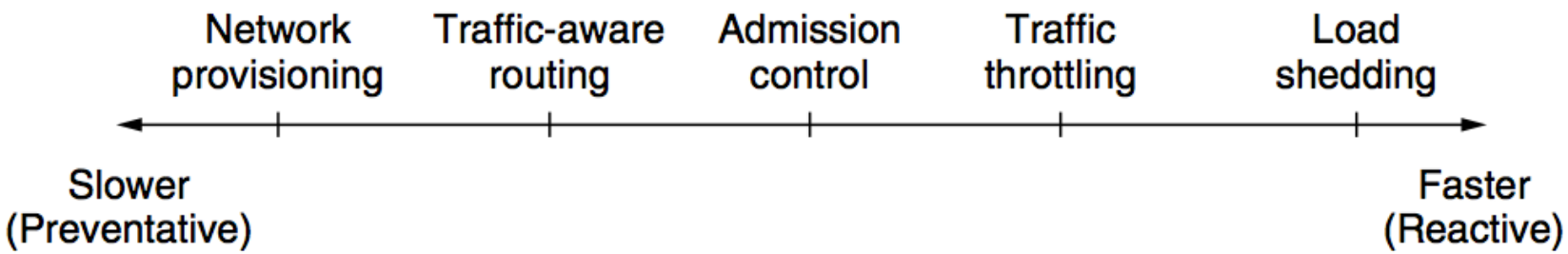
- Congestion control
  - Makes sure the network is able to carry the offered traffic
  - A global issue, involving the behavior of all the hosts and routers
  - E.g., 1000 PCs send packets at 100-Kbps over a 1-Mbps lines
- Flow control
  - Cares the traffic between a particular sender and a particular receiver
  - Makes sure that a fast sender cannot continually transmit data faster than the receiver is able to absorb it
  - E.g., a supercomputer sends 100-Gbps over fiber to a 1-Gbps PC

2 Types of Congestion



Approaches to Congestion Control

- 1. Network provisioning: add network resources
- 2. Traffic-aware routing: tailor route to traffic patterns
- 3. Admission control: refuse new connection (in virtual-circuit)
- 4. Traffic throttling: request source to slow down, or slow down itself
- 5. Load shedding: just discard packets



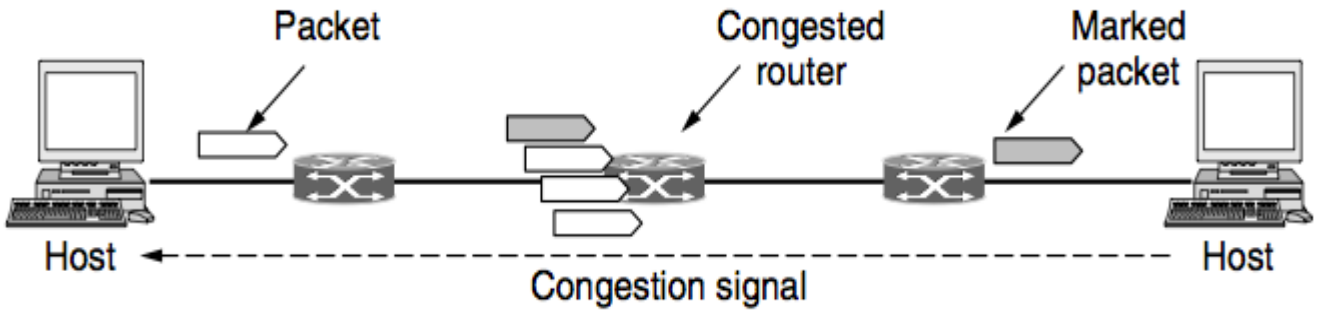
Traffic Throttling

- When congestion is imminent, the network (routers) must tell the senders to throttle back their transmissions and slow down

- First, routers must determine when congestion is approaching
  - the utilization of the output links
  - buffering of queued packets inside the router (most useful)
  - the number of packets that are lost due to insufficient buffering

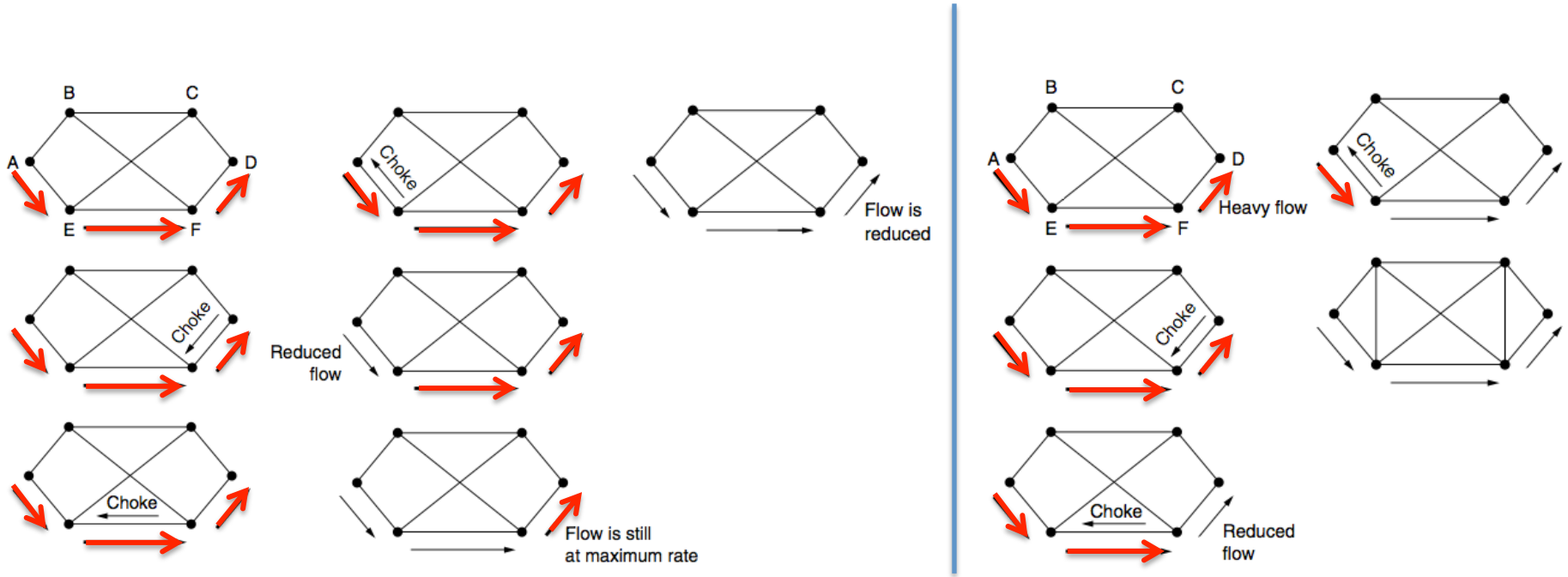
Traffic Throttling

- Choke packets
  - Router sends choke packet to source, source reduce load by, e.g., 50%
  - Must be sent at low rate to avoid congestion
- Explicit Congestion Notification
  - Tag IP packet to the dest (2 bits in IP header), and then back to the source
  - The "back" process is end-to-end, e.g., in TCP layer



Traffic Throttling

- Hop-by-Hop Backpressure



Load Shedding: Setting Window Size

- For performance:
  - *window size ≥ round-trip time × bottleneck data rate*
- For congestion control:
  - *window size ≤ min(RTT x bottleneck data rate, Receiver buffer)*
  - Congestion window (cwnd)
- 2 windows become 1
  - to achieve best performance and avoid congestion

Congestion Control

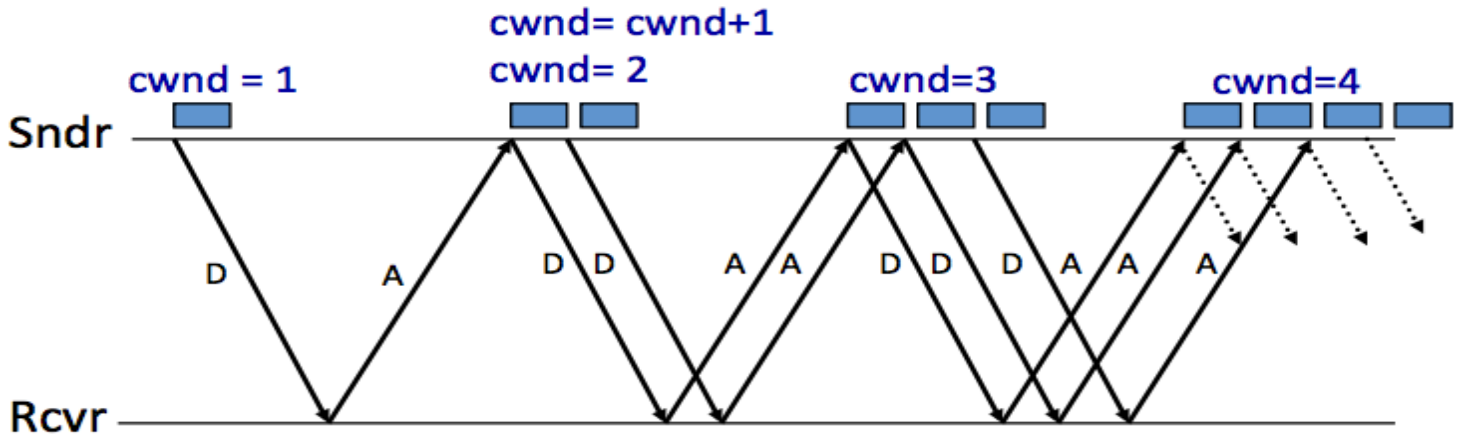
Basic Idea:

- Increase cwnd slowly
- If no drops -> no congestion yet
- If a drop occurs -> decrease cwnd quickly

Use the idea in a distributed protocol that achieves

- Efficiency: i.e., uses the bottleneck capacity efficiently
- Fairness, i.e., senders sharing a bottleneck get equal throughput (if they have demands)

AIMD (Additive Increase, Multiplicative Decrease)

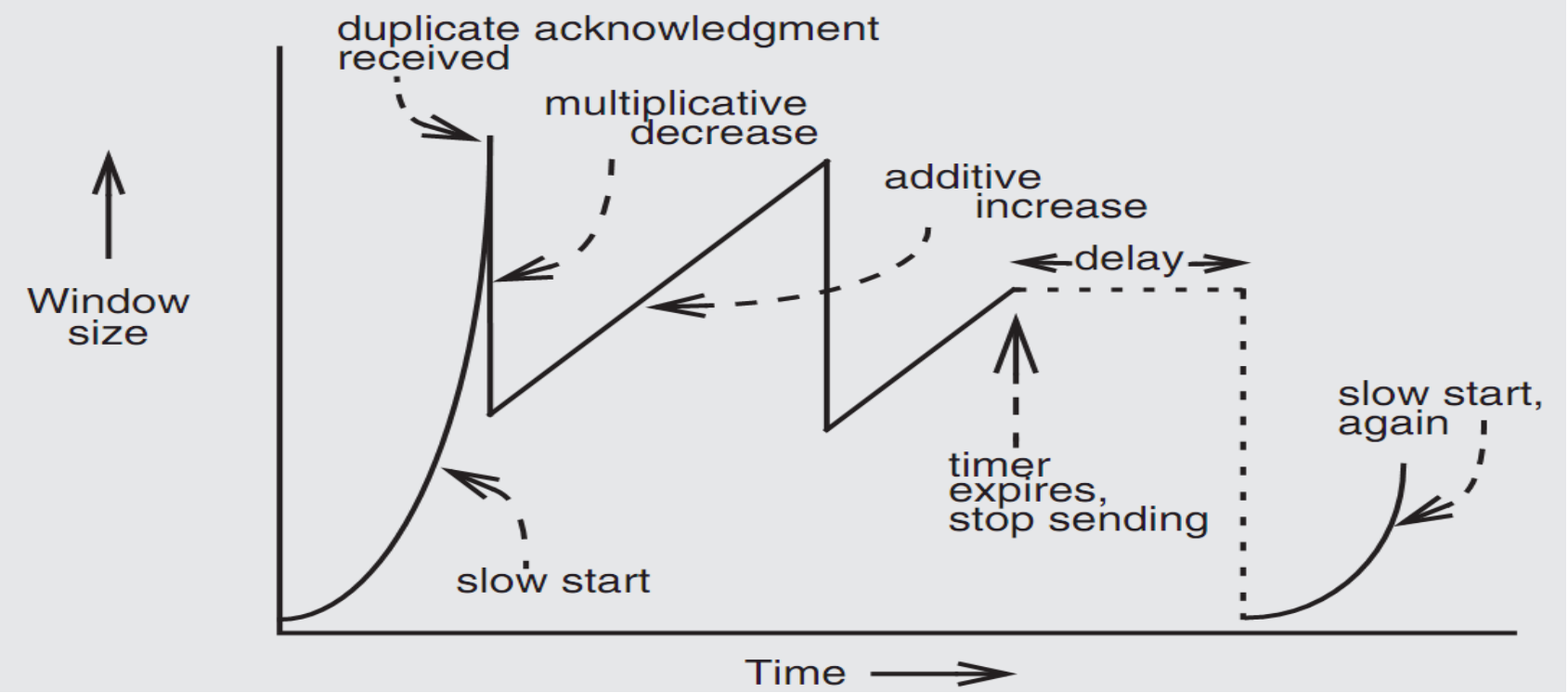


- Every RTT:
  - No drop: cwnd = cwnd + 1
  - A drop: cwnd = cwnd / 2

Problems with AIMD

- Increases very slowly at the beginning
- Initial window size is 1
  - Probably too small in practice
- Solution: do multiplicative increase at the beginning
  - $cwnd_{init} = 1$
  - initially, do $cwnd \leftarrow 2 * cwnd$ each RTT until we hit congestion
  - Named "slow start" (even though it's exponentially fast!)

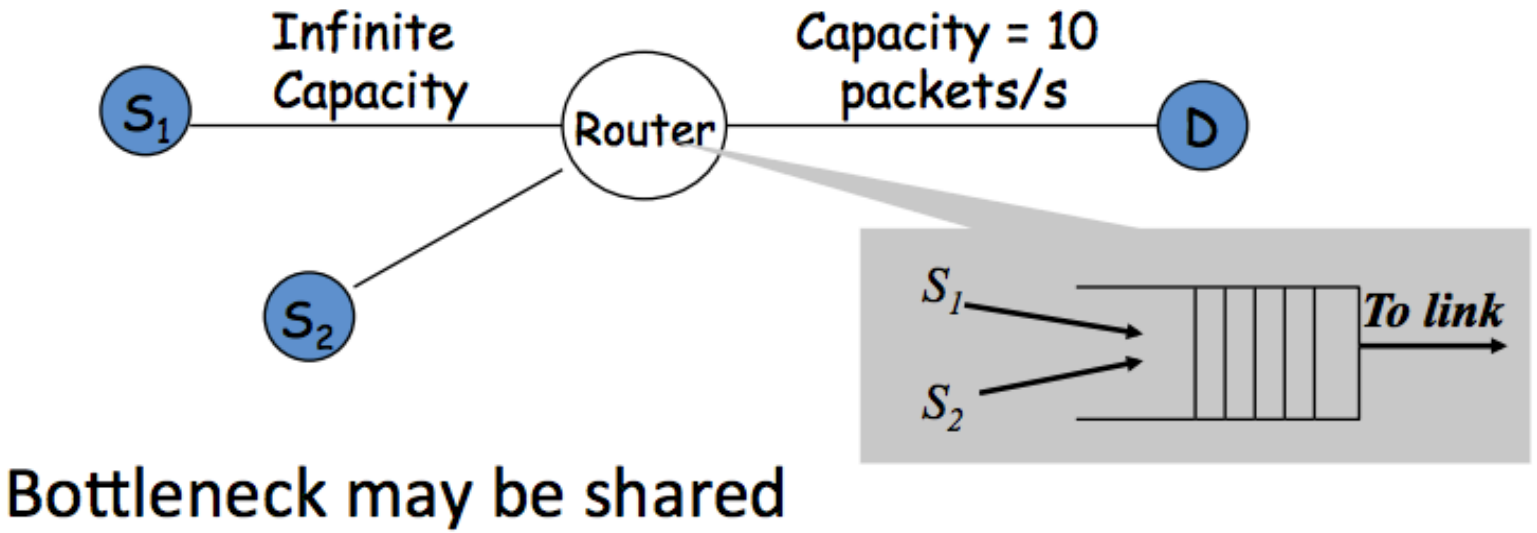
Retrofitting TCP



Retrofitting TCP

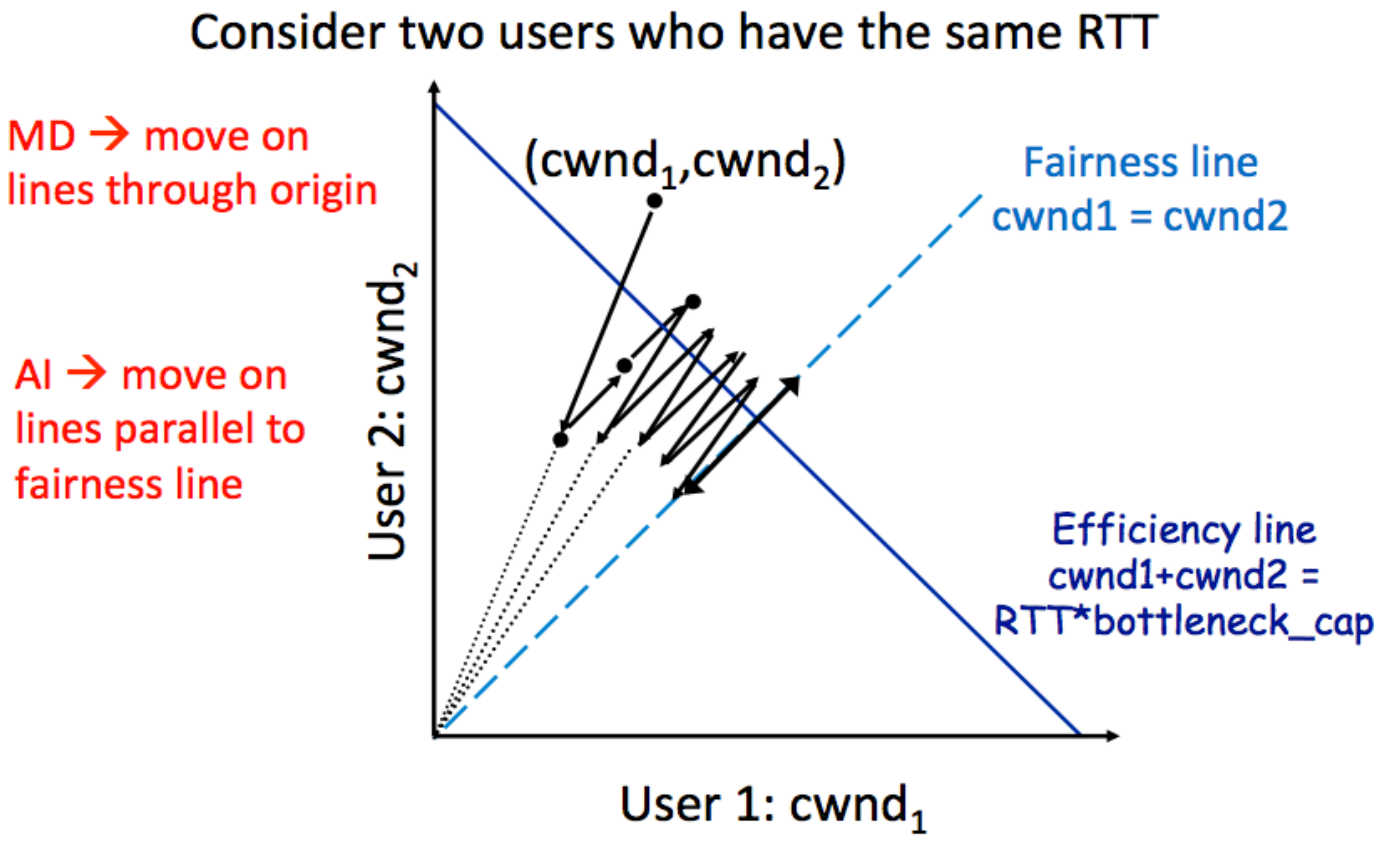
- 1. Slow start: one packet at first, then double until
  - Sender reaches the window size suggested by the receiver
  - All the available data has been dispatched
  - Sender detects that a packet it sent has been discarded
- 2. Duplicate ACK
  - When receiver gets an out-of-order packet, it sends back a duplicate of latest ACK
- 3. Equilibrium
  - Additive increase & multiplicative decrease
- 4. Restart, after waiting a short time

Fairness between Links



Bottleneck may be shared

AIMD Leads to Efficiency and Fairness



Q: Why not Additive Decrease

- It does not converge to fairness
  - from a congested point, (x',y'), reducing each by 1 worsens fairness and takes us away from the "ideal" outcome

Weakness of TCP

- If routers have too much buffering, causes long delays
- Packet loss is not always caused by congestion
  - Consider wireless network
- TCP doesn't perform well in datacenters
  - High bandwidth, low delay situations
- TCP has a bias against long RTTs
  - Throughput inversely proportionally to RTT
  - Consider when sending packets really far away vs really close
- Assumes cooperating sources, which isn't always a good assumption

Summary of Congestion Window

- Reliability Using Sliding Window
  - Tx Rate = W / RTT
- Congestion Control
  - W = min(Receiver_buffer, cwnd)
  - cwnd is adapted by the congestion control protocol to ensure efficiency and fairness
  - TCP congestion control uses AIMD which provides fairness and efficiency in a distributed way

QUEUE MANAGEMENT

Congestion Control is A Cross-layer Problem

- The end-to-end layer (TCP) is using network layer's feedback to adapt
  - The feedback: packet drops
- Routers have buffer
  - If buffer is full -> long delay
  - When use a buffer? the network is enough (to absorb burst)
  - When not use a buffer? there's bottleneck on the network
  - What we don't want is: queues that never drain

Packet Drop VS. Delay

- Problem: TCP reacts to packet drops, but packets are not dropped until the queues are full
  - It may be too late
- Observation: the delay increases as the buffer increases
- TCP senders back off when observe an increase in delay
  - Earlier, before the queues are full
  - But, how to distinguish between full-queue and transient queue?
    - Both increase the delay, but we don't want TCP back-off for the latter

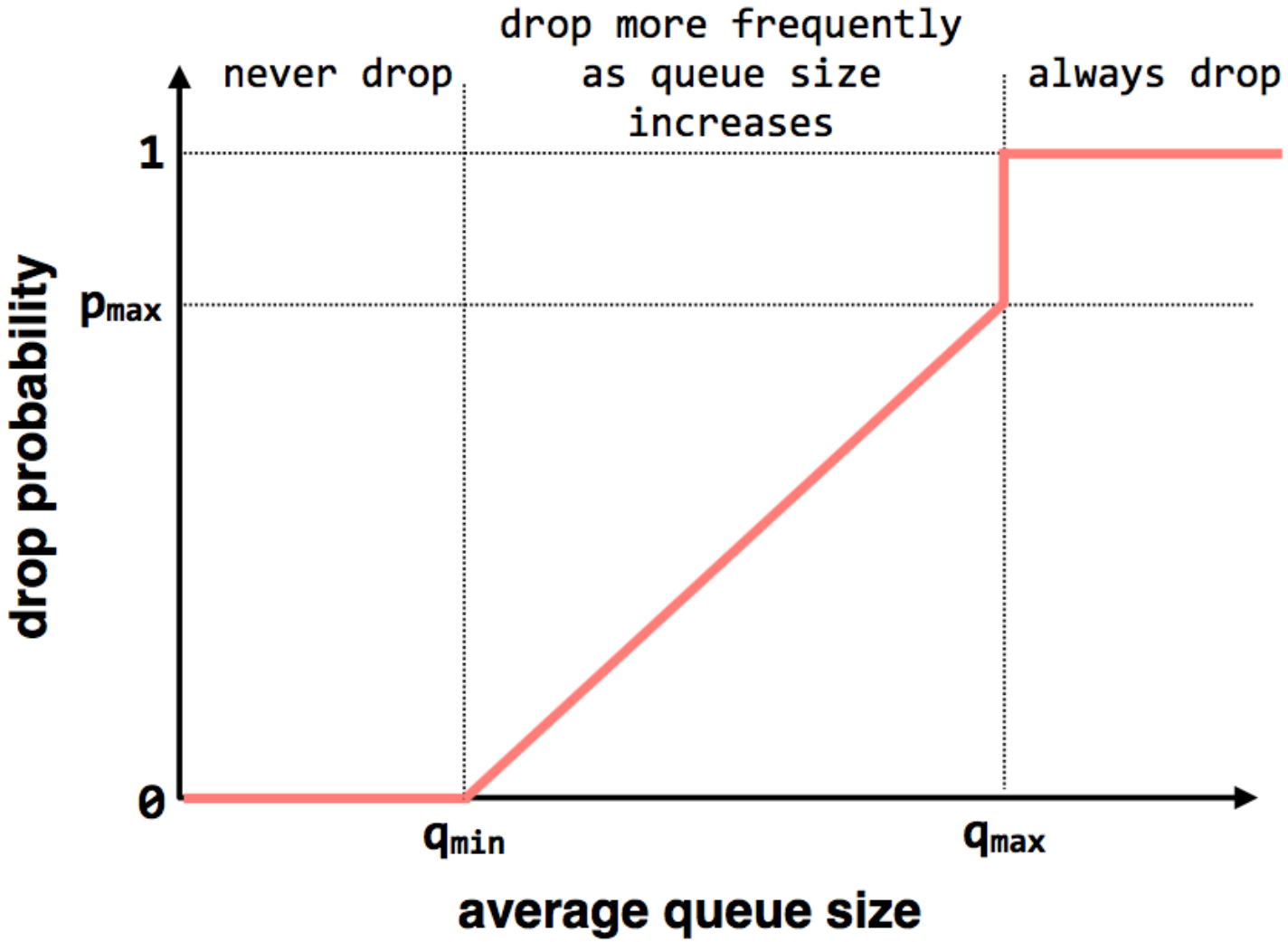
DropTail

- Drop a packet if and only if the queue is full
- Synchronization problem
  - Consider multiple sources, all with bursty type
  - 1. All sources burst
  - 2. Packet dropped from all
  - 3. All sources throttle back, reducing utilization
  - 4. Sources increase, repeat from 1
  - All the flows get synchronized!
- Flow synchronization = decreased network utilization

RED (Random Early Detection)

- Drop before the queue is full to give flows an early signal
  - RED spreads the drops out, while DropTail does this all at once

- + Queue length does not oscillate as much
- + Smooth change in the drop rate with congestion
- + Flows get desynchronized


never drop
drop more frequently
as queue size
increases
always drop
1
p_{max}
0
drop probability
q_{min}
q_{max}
average queue size

Explicit Congestion Notification (ECN)

- Idea: Mark packets instead of dropping them
  - A particular bit in the header
  - As a signal of congestion
  - No packet drop at all
  - The source needs to cooperate (same as RED)

RED/ECN VS. DropTail

- Pros
  - Smaller persistent queues => smaller delays
  - Less dramatic oscillation
  - Less biased against bursty traffic (in theory)
- Cons
  - More complex
  - Hard to pick parameters (min_q, max_q, etc.)
    - Depend on the number of flows in the network, the bottleneck, etc. If not chosen well, the oscillation can be even worse

Traffic Differentiation

- Youku stream VS. large email
  - Consider FIFO and round-robin
  - Priority based scheduling
- Good or bad?
  - It's hard to decide the granularity
    - Per-app? per-flow? per-destination? per-source?
  - How to ensure fairness?

Fault Tolerant

Reliable System from Unreliable Components

"It's good to learn from your mistakes. It's better to learn from other people's mistakes."

– Warren Buffett

War Story

War Stories: MAXC & Alto

MAXC

- A time-sharing computer system
  - Using Intel 1103: 1KB memory
- Fault tolerant
  - Extra bits for each 36-bit
  - Single-error-correction
  - Double-error-detection
- Result
  - Solidly reliable
  - No errors were reported

Alto

- A personal workstation
  - Using the same Intel memory chip
- Fault tolerant
  - One parity bit for each 16-bit
  - Error-detection only
- Result
  - Frequent memory-parity failures!
    1. Bit-map display: Bravo
    2. Pattern-sensitive 1103

War Stories: MAXC & Alto

- Lesson-1:
  - There is no such thing as a small change in a large system
  - A new software can bring down previously working hardware
  - You are never quite sure just how close to the edge of the cliff you are standing
- Lesson-2:
  - Experience is a primary source of information about failures
  - It is nearly impossible, without specific prior experience, to predict what kinds of failures you will encounter in the field

War Stories: MAXC & Alto, Chap-2

- Back to MAXC: why so few errors at that time?
  - Hardware reports both corrected errors and uncorrectable errors
  - But software only logged uncorrectable errors!

- Lessons
  - Safety margin principle is important when system implement automatic error masking
  - Otherwise, you may be standing on the edge again

War stories: MAXC & Alto, Chap-3

- New machine: Alto-2
  - Using new memory chip with 4096 bits
  - Single-error-correction, double-error-detection, again, flawlessly!
  - Two years later, an implementation mistake is discovered
  - Neither error correction nor detection was working on 1/4 cards!

- Lessons
  - Never assume the h/w does as it says in the spec
  - It is harder than it looks to test the FT features of a FT sys

Nowadays

- Memory chip does become better
  - Errors often don't lead to failures
  - Bit-flip can be benign
- But there are still errors
  - Sometimes, failure is blamed on something else
  - You never know it h/w error or s/w error

The U.S. National Archives

- Preserve electronic records e.g., email
  - Daily incremental backups
  - periodic complete backups
  - audit logs for tracking actions
- June 18-21, 1999, 43,000 emails disappeared
  - The audit log had been turned off for better performance
  - The contractor's employees never backup
  - Archives never verify the existence of backup copies

Lesson: Avoid rarely used components, must be tested periodically

Hospital Operating Room in Newark, New Jersey

- Hospital operating room, with three backup generators
- August 14, 2003, a widespread power grid failure
  - One backup generator caught fire from an oil leak
  - Another backup generator shut down due to overload
  - Hospital engineering turned off many circuit
  - Main power was interrupted to the OR

1. Rarely used component may not have been maintained properly
2. Human makes mistakes

World-Flight of Northwest Airline

- 2:05 p.m. on March 23, 2000
  - All communications dropped out
  - Six cables were accidentally bored through
  - Both primary and secondary!
  - Pilots resorted to manual procedures
  - Radio links were used
  - 125 flights had to be cancelled

1. MTTF depend on that replicas are independent
2. FT measuring is difficult to test

British: Telehouse's Electricity

- Safe against "fire, flooding, bombs, sabotage"
  - Especial protection against power failure
  - Including two independent connections to power grid
  - A room full of batteries, two diesel generators
  - Can detect failures automatically then switch
  - May 8, 1997…"It was due to human error.", which is not detectable

1. Identifying each potential fault and evaluate the risk is first step
2. People are part of the system, mistakes made by authorized operators are typically a

Kerosene Light in Radin, Poland

- A town of Radin has a electricity generator
  - One day the machine broke
  - Darkness descended upon all of the streets
  - Even worse than the days without electricity
  - Every house had a kerosene light

1. Single point of failure: Centralization reduces robustness
2. Adding redundancy to a centralized design takes planning and adds to the

The SOHO Mission Interrupt

- SOHO spacecraft was lost on June 25, 1998
  - "… a direct result of operational errors, a failure to adequately monitor spacecraft status, and an erroneous decision which …"
  - Five distinct direct causes of the loss
  - Three indirect causes in design process
  - Three indirect causes in operations procedures

1. Complex systems fail for complex reasons
2. When some components are people,

Basic Concepts

Fault, Error, Failure

Fault, Error, Failure

- Fault can be latent or active
  - If active, get wrong data or control signals
- Error is the results of active **fault**
  - E.g. violation of assertion or invariant of spec
  - Discovery of errors is ad hoc (formal specification?)
- Failure happens if an **error** is not detected and masked
  - Not producing the intended result at an interface

Failure in System and Subsystem

- The failure of a subsystem is a fault of a system
  - The fault may cause an error that leads to the failure of the larger subsystem
  - Unless the larger subsystem detects the error and masks it
- Examples
  - A flat tire -> detects the error by failure of a subsystem
  - Miss an appointment -> the person notices a failure
  - Change to a spare tire -> masked the error

Different Types of Faults

Fault Types	Types
Software fault	Programming mistakes
Hardware fault	A gate whose output is stuck at the value ZERO
Design fault	Assign too little memory in a telephone switch
Implementation fault	Installing less memory than the design called for
Operations fault	Running a weekly payroll twice last Friday
Environment fault	Lightning strikes a power line causing a voltage surge
Transient fault	Cosmic ray caused single bit flipped (soft

More Examples

- Stuck-at-ZERO in a memory chip
  - It's a persistent fault, but is not active
- When the bit should contain a ONE
  - The fault is active and the value is in error
- If cosmic ray flips 1 more bit, and system can only handle 1-bit error
  - Two bits may cause a failure of the module
- If someone tests the module
  - If the error is masked, the masking should be reported

The Fault Tolerance Design Process (1/2)

- 1. Begin to develop a fault-tolerance model
  - Identify every potential fault
  - Estimate the risk of each fault
  - Where the risk is too high, design methods to detect the errors
- 2. Apply modularity to contain the damage
- 3. Design and implement procedures to mask
  - Temporal redundancy. Retry using the same components
  - Spatial redundancy. Have different components do the operation
- 4. Update the fault-tolerance model
  - To account for those improvements

The Fault Tolerance Design Process (2/2)

- 5. Iterate the design and the model
  - until the probability of untolerated faults is low enough that it is acceptable
- 6. Observe the system in the field
  - Check logs of how many errors the system is successfully masking. (Always keep track of the distance to the edge of the cliff)
  - Perform postmortems on failures and identify all of the reasons for each failure
- 7. Use the logs of masked faults and the postmortem reports to revise and improve the FT model and reiterate the design

Tolerating Active Faults

- Do nothing: let higher layer solve the failure
  - *The more layers, the more difficult*
- Be fail-fast: report the problem
  - *E.g., Ethernet stops sending and broadcasts when collision*
- Be fail-safe: transfer incorrect values to acceptable ones
  - *E.g., blinking red light in all directions*
- Be fail-soft: continues to operate with degradation
  - *E.g., airplane with three engines continues to fly if one has failure*
- Mask the error: makes incorrect value right

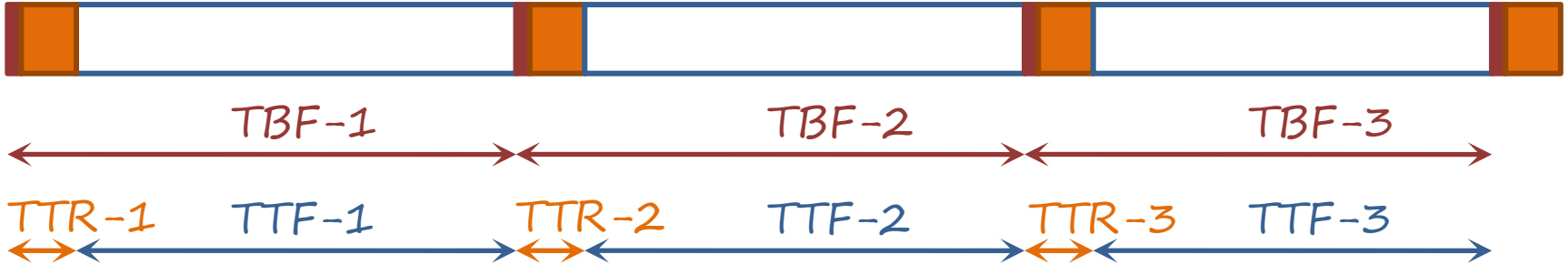
MTTF & Availability

MTTF, MTTR, MTBF

- MTTF: mean time to failure
- MTTR: mean time to repair
- MTBF: mean time between fail
- *MTBF = MTTF + MTTR*

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i$$

$$MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i$$



MTBI

Web page sna

Data Sheet

Barracuda® 7200.10

Experience the industry's proven flagship perpendicular 3.5-inch hard drive

80 GB to 750 GB • SATA 1.5Gb/s or 3Gb/s and PATA 100

Key Advantages

- First 3.5-inch drive to utilize capacity- and reliability-boosting perpendicular recording technology
- First drive to reach 750 GB—a full year ahead of competition—enabling new solutions for data-intensive applications.
- Industry's most proven and established desktop hard drive available today—more than 16 million shipped to date*
- "One-stop shopping" with a broad range of capacity, cache and interface options for all your computing needs
- Best-in-class environmental specifications and reliability features
- Adaptive Fly Height offers consistent read/write performance from the beginning to the end of your computing workload.
- Clean Sweep automatically calibrates your drive.
- Directed Offline Scan runs diagnostics when storage access is not needed.
- RoHS-compliant design assures an environmentally conscious product.
- Enhanced G-Force Protection™ defends against handling damage.
- Seagate® SoftSonic™ motor enables whisper-quiet operation.

Best-Fit Applications

Desktop and High-Performance PCs

- Gamer PCs
- Workstations
- High-end PCs
- Desktop RAID
- Mainstream PCs
- Point-of-sale devices/ATMs
- USB/FireWire/eSATA personal external storage

*16 million Barracuda 7200.10 drives shipped as of 4/18/07

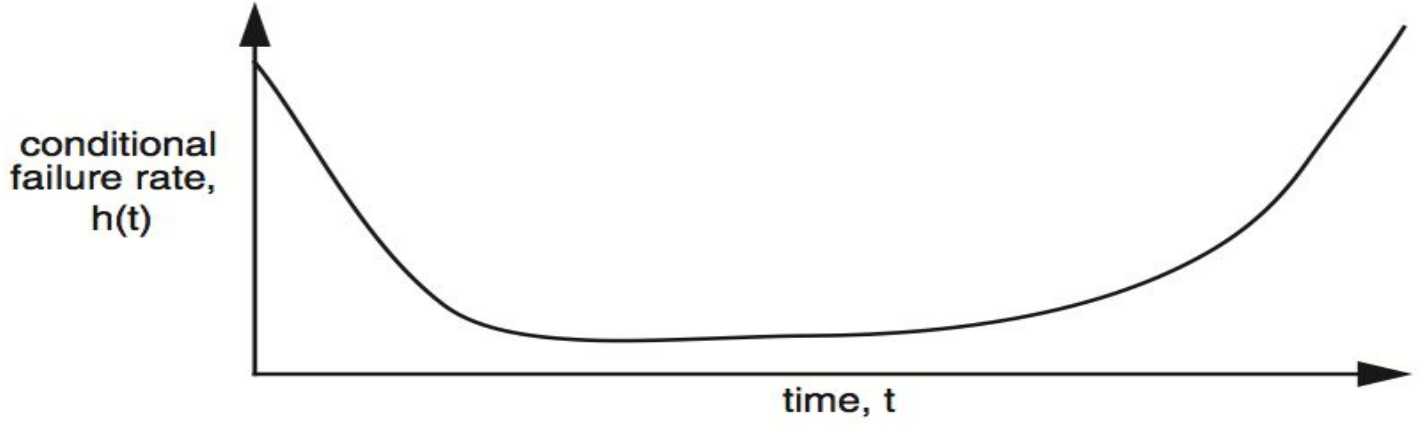
Contact Start-Stops	50,000
Nonrecoverable Read Errors per Bits Read	1 per 10^{14}
Mean Time Between Failures (MTBF, hours)	700,000
Annualized Failure Rate (AFR)	0.34%

Measuring MTBF

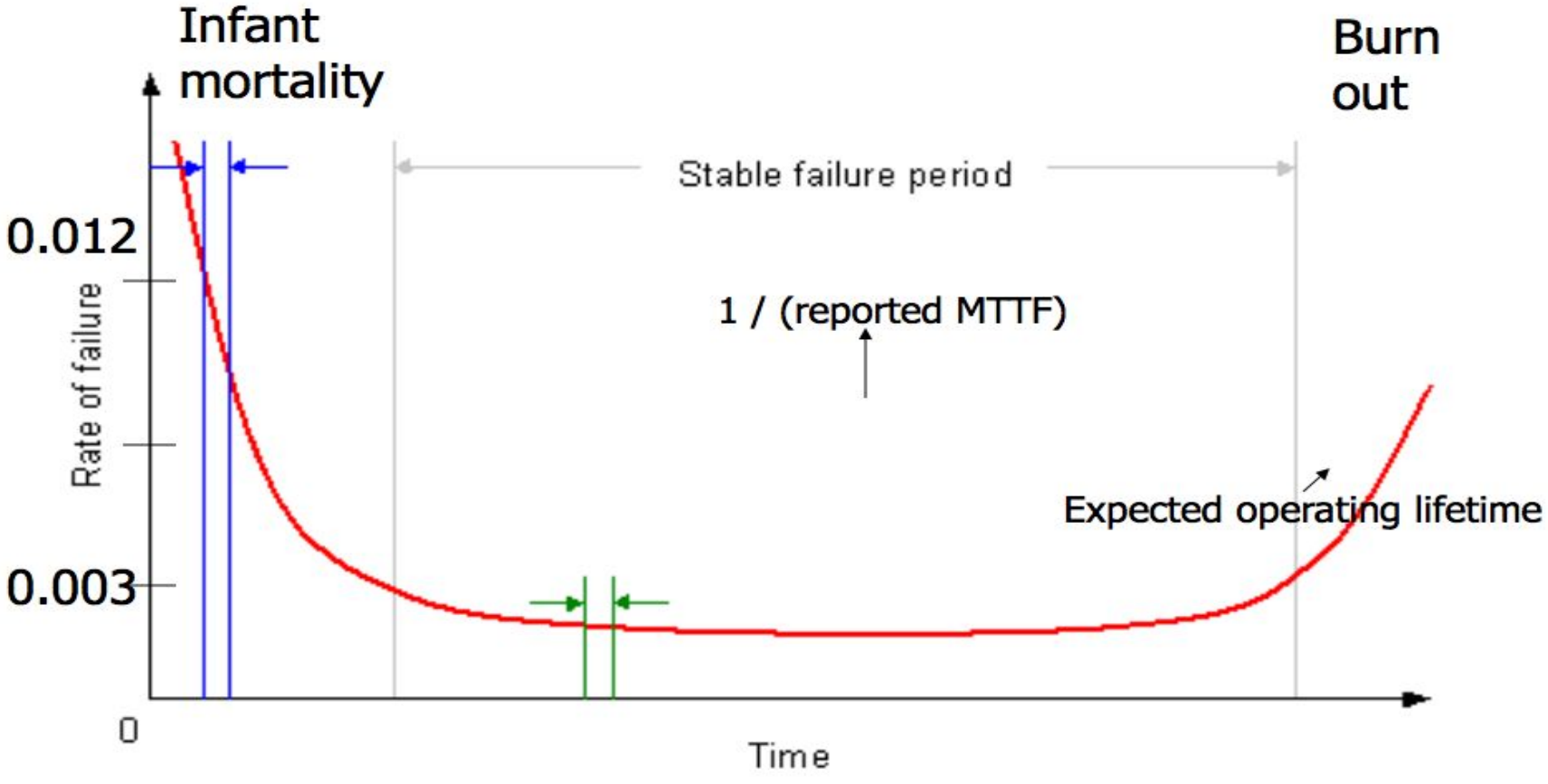
- Two purposes: *evaluating* & *predicting*
  - The more reliable, the longer it takes to evaluate
  - Not measure directly, but use proxies to estimate its value
- One way to measure "MTBF"
  - E.g. 3.5-inch disk's MTTF is 700,000 hours (80 years!)
  - Guess: ran 7,000 disks for 1,000 hours and 10 failed
  - If the failure process were *memoryless*, then OK
  - "expected operational lifetime" is only 5 years

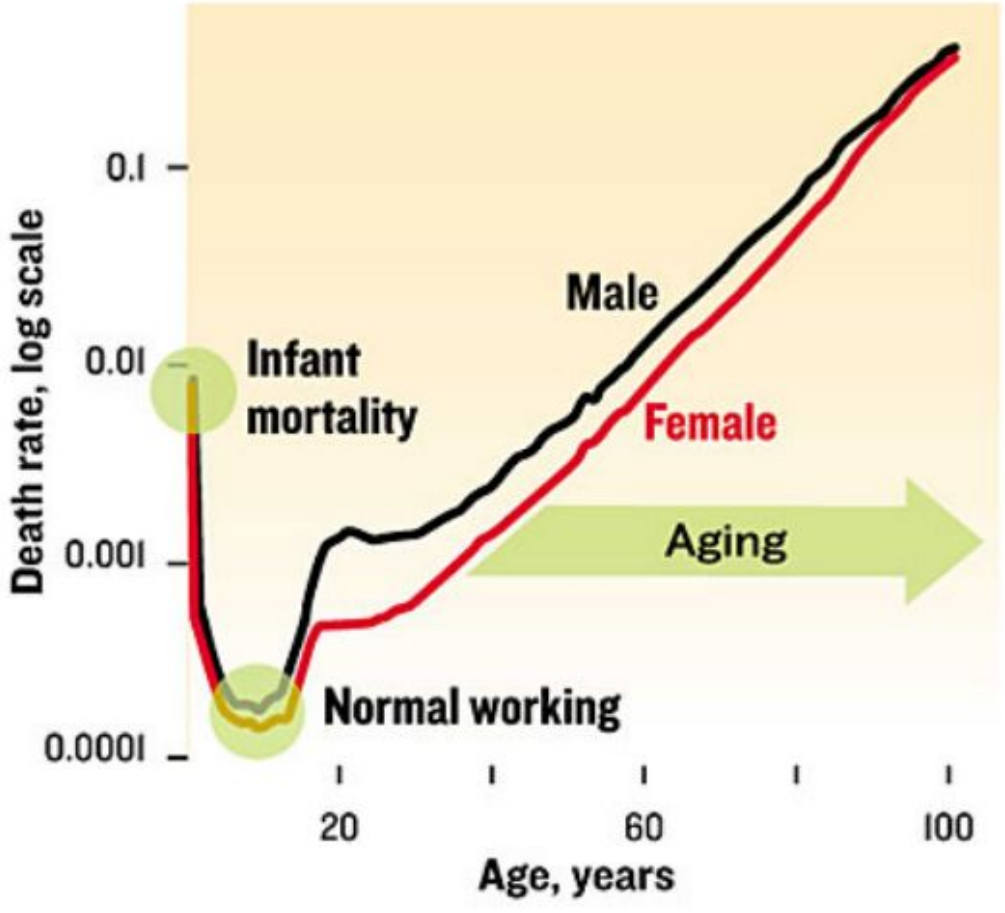
Measuring MTBF

- Bathtub curve
  - Infant mortality: gross manufacturing defects
  - Burn out: accumulated ware and tear cause failure
  - Burn in: run for a while before shipping



The Bathtub Curve





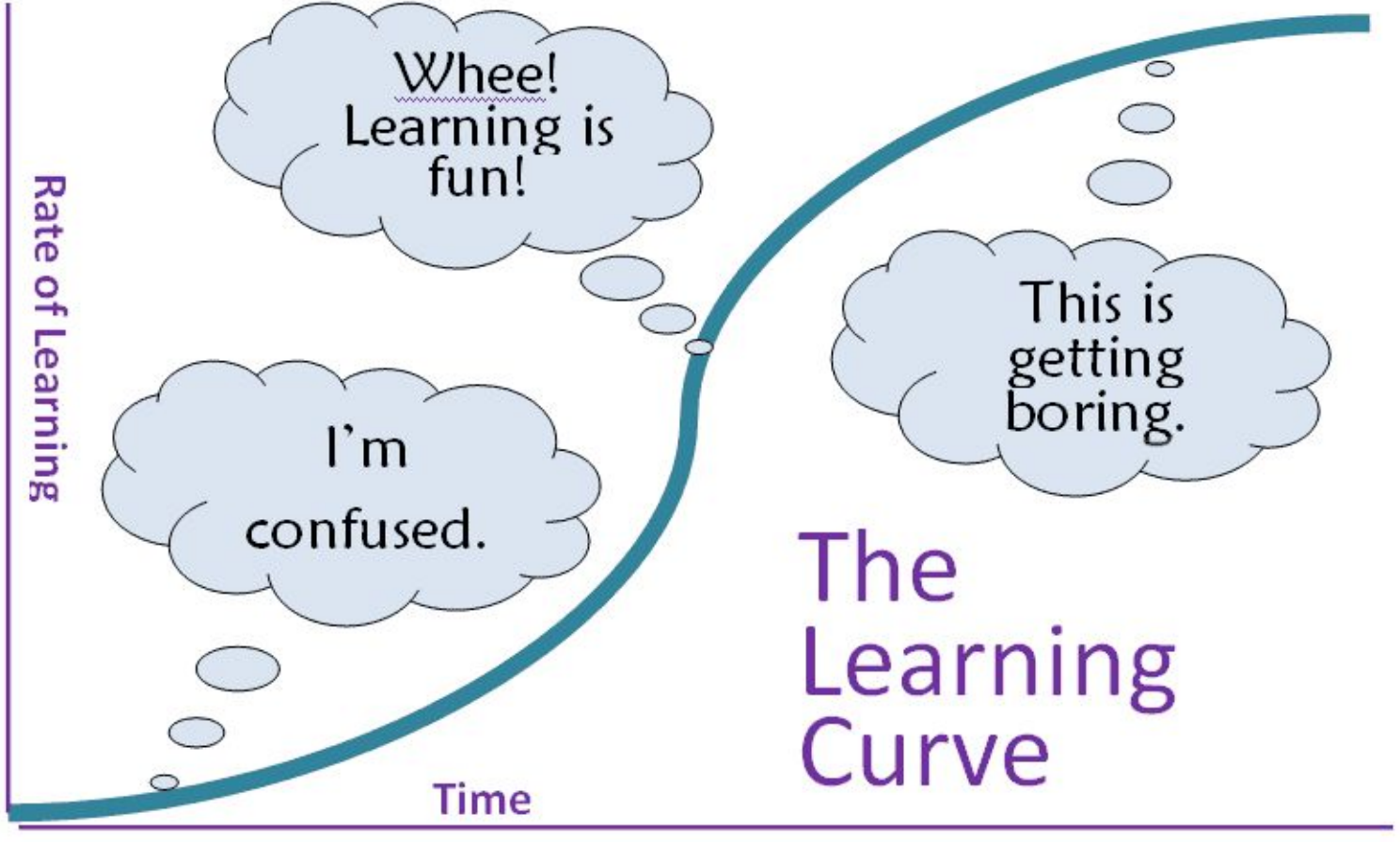
Human Mortality Rates (US, 1999)

From: L. Gavrilov & N. Gavrilova, "Why We Fall Apart," IEEE Spectrum, Sep. 2004.
Data from http://www.mortality.org

The Learning Curve

- Learning curve
  - The first components coming out of a new production line tend to have more failures
  - Result of design of iteration

The Learning Curve



Frequency Hardware Replaceme

COM1	
Component	%
Power supply	34.8
Memory	20.1
Hard drive	**18.1**
Case	11.4
Fan	8.0
CPU	2.0
SCSI Board	0.6
NIC Card	1.2
LV Power Board	0.6
CPU heatsink	0.6

10,000 machines

Pr(failure in 1 year) ~.3

Schroeder and Gibson, FAST 2008

Availability

- Counting the number of nines
  - E.g. 99.9% has 3-nines availability
  - Often used in marketing
- Corresponding down time
  - E.g. 3-nines -> 8 hour/year
  - E.g. 5-nines -> 5 min/year
  - E.g. 7-nines -> 3 sec/year
  - Without any information about MTTF
- More fine-grained
  - Fail-soft, as in 8.3

Availability in Practice

- Carrier airlines (2002 FAA fact book)
  - 99.9993% availability, 41 accidents, 6.7M departures
- 911 Phone service (1993 NRIC report)
  - 99.994%, 29 minutes per line per year
- Standard phone service (various sources)
  - 99.99+%, 53+ minutes per line per year
- End-to-end Internet Availability
  - 95% - 99.6%

Redundancy

Systematically Applying Redundancy

- Masking error
  - Analog system designer: margin
  - Digital system designer: redundancy
- Outline
  - Coding: incremental redundancy
  - Replication: massive redundancy
  - Voting
  - Repair

Recall: Coding for Incremental Redundancy

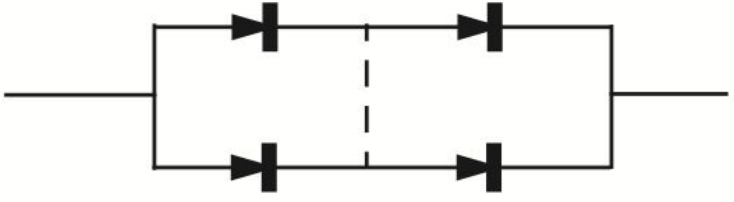
- Forward error correction
  - Perform coding before storing or transmitting
  - Later decode the data without appealing to the creator
- Hamming distance
  - Number of 1 in A $\oplus$ B , $\oplus$ is exclusive OR (XOR)
  - If H-distance between every legitimate pair is 2
    - 000101, can only detect 1-bit flip
  - If H-distance between every legitimate pair is 3
    - Can only correct 1 bit flip
  - If H-distance between every legitimate pair is 4
    - Can detect 2-bit flip, correct 1-bit flip

100101
000111

100101
010111

Replication: Massive Redundancy

- Redundancy in bridge building
  - Material strength: 5 or 10 times as strong as min
  - Heavy-handed, but simple and effective
- Replication
  - Replicas: identical multipl
  - E.g. Quad component by Shannon & Moore
    - Can tolerate single short circuit and single open circuit, and others
  - Mask failures silently



Voting

- NMR: N-modular redundancy (supermodule)
  - E.g. TMR (3MR) and 5MR
  - Voter: compare the outputs with the same inputs
  - Can be applied at any level of module
- Fail-vote: NMR with a majority voter
  - Raise an alert if any replicas disagree with majority
  - Signal a failure if no majority
  - Fail-fast only if any two replicas fail in different ways
    - If two replicas fail in same way, then not fail-fast

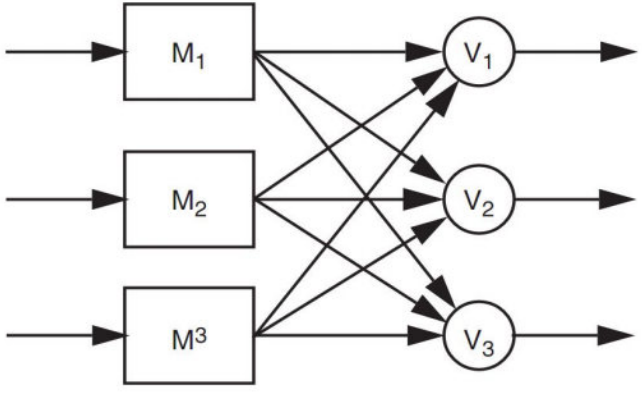
$$R_{supermodule} = R^3 + 3R^2(1-R) = 3R^2 - 2R^3$$

Voting

- The probability that an incorrect result will be accepted by the voter is that it is no more than:

$$(1-R_{supermodule}) = (1-3R^2+2R^3)$$

- Assume that the voter is perfectly reliable, which is not practical
  - Voter should also be replicated
  - Everything should be replicated
  - Recursive: voters belong to next module
  - Final voter should be the client



Voting

- TMR can improve reliability
  - If R(T) = 0.999, TMR's R(T) = 0.999997
- But MTTF can be smaller
  - If MTTF is 6,000 hours and fails independently, and the mechanism of engine failure is memoryless
  - 6,000 hours in only 2,000 hours of flying, first fail
  - 3,000 hours next and cause the second fail
  - 5,000 hours < 6,000 hours

Mean time to first failure	2000 hours (three engines)
Mean time from first to second failure	3000 hours (two engines)
Total mean time to system failure	5000 hours

MTTF-replica and MTTF-system

- If MTTF-replica = 1, N replicas total
  - Expected time until 1st failure is MTTF-replica/N
  - Expected time from then until the 2nd is MTTF-replica/(N-1)
  - Expected time until the system of N replicas fails is

  - $MTTF_{system} = 1 + 1/2 + 1/3 + \ldots(1/N)$
- If mission time is long compared with MTTF-replica
  - Simple replication escalates the cost while providing little benefit

Repair

- Return to fail-vote TMR supermodel
  - Requires at least two replicas be working
  - The rate of failure of 2 replicas is 2/MTTF
  - $Pr(\text{supermodule fails while waiting for repair}) = \frac{2 \times MTTR}{MTTF}$

    1/3000
  - Once replaced, expect to fly another 2000 hours until the next engine failure
  - If we have unlimited replacement for 10,000 hours

$$MTTF_{\text{supermodule}} = \frac{MTTF_{replica}}{3} \times \frac{MTTF_{replica}}{2 \times MTTR_{replica}} = \frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}}$$

- MTTF = 6 million hours

- In another word
  - Although MTTF is reduced by the factor of 3
  - The availability of repair has increased MTTF by a factor equal to the ratio of the MTTF to MTTR of the remaining 2 engines

- Disk with 3 replicas
  - If MTTF is 5 years, repair in 10 hours

$$\frac{(MTTF_{replica})^2}{6 \times MTTR_{replica}} = \frac{(5 \text{ years})^2}{6 \cdot (10 \text{ hours})/(8760 \text{ hours/year})} = 3650 \text{ years}$$

Review Overly Optimistic Assumptions

- Disks fail independently
  - Same vendor, same fault
  - Earthquake, whose MTTF is less than 3650 years
- Disk failures are memoryless
  - Bathtub curve
- Repair is also a memoryless process
  - Stock enough spares
- Repair is done flawlessly
  - Replace the wrong disk, forget to copy data, etc.

Redundancy to software and data

Applying Redundancy to Software and Data

- Software and data
  - Soft: reduce the impact of programming errors
  - Data: reduce the impact of any kind of hw, sw, or operational error that may affect integrity
  - N-version programming, valid construction, firewall
- Outline
  - Tolerating software faults
  - Tolerating soft (and other) faults by separating state
  - Durability and durable storage
  - Magnetic disk fault tolerance

Tolerating Software Faults

- NMR requires independence among the replicas
  - E.g. DNS software evolved for different OSes
- N-version programming
  - Commission several teams of programmers
  - Run several versions in parallel and compare results
  - Same training? Same language? Same ambiguities?
  - Bit-for-bit identical results are difficult
  - Boeing 777 aircraft (N=3) & Space Shuttle (N=2)
  - Can also apply in hardware replica

Tolerating Software Faults

- Valid construction
  - Devising spec and programming tech that avoid faults in the first place
  - Test tech that systematically root out faults
  - Repaired once and for all before deploying
  - Software: once it is made correct, it stays that way
    - Hard to get correct: patch for patch for patch
    - Changing is required
    - Tension between valid construction and design for iteration
    - Later maintainer may make things worse

Separating States

- Residue of faults is *inevitable*
  - Both software and hardware
- Distributed states
  - Non-volatile storage, volatile memory, processor registers, kernel tables, etc.
  - Makes containment of errors problematic
  - Stop/repair/resume is usually unrealistic
- Separating states
  - 1. State that the system can safely abandon when fail
  - 2. State whose integrity the system should preserve despite failure

Separating States

- Upon detecting a failure
  - Abandon all state in the first category
  - Just maintain the integrity of second category
  - Starting a new set of threads with a clean states
    - Working only with the second category
  - Firewall is needed
- Natural firewall
  - Non-volatile storage: GET/PUT interface, as a bottleneck
  - Volatile storage: READ/WRITE interface
  - Stop the system before it reaches the next PUT
    - Making the volatile storage the error containment boundary

50

Durability and Durable Storage

- Durability
  - How long a result of action be preserved after the action completes
- 1. Durability no longer than the lifetime of the thread that created the data
  - Place the data in volatile memory is enough
  - E.g. CHDIR: change working directory
  - E.g. registers and cache of a processor
- 2. Durability for times short compared with expected operational lifetime of non-volatile storage media
  - Writing one copy of the data in the nonvolatile storage
  - E.g. data in a cache that writes through to a non-volatile medium
  - E.g. writing nonce to non-volatile memory in RPC
  - E.g. temporary copies in word-processing system

Durability and Durable Storage

- 3. Durability for times comparable to the expected operational lifetime of non-volatile media
  - Placing replicas of the data on independent instances of the non-volatile media
- 4. Durability for many multiples of the expected operational lifetime of non-volatile media
  - Aka. preservation
  - Copying data from one non-volatile medium to another before the first one deteriorates or becomes obsolete
  - Also consider software's lifetime

Magnetic Disk Fault Tolerance

- Using disk as durable storage
  - Low cost, large capacity, non-volatility
  - Internal power can prevent data loss at power-off
- Three/four nested layers
  - Raw storage, fail-fast storage, careful storage
  - Optional: durable storage
- Fail-fast storage
  - In software first, then migrate to firmware of the disk controller
  - Usually includes a RAM buffer
  - Reason: easy to track the source of failure, not as memory

Magnetic Disk Fault Modes

- Mechanical wear and tear
- A bumping may cause a head to hit the surface
  - Head crush may also create cloud of dust
  - Results in several sectors decaying: decay set
- Electronic components in the controller age
  - E.g. clock timing and signal detection circuits
  - Cause previously good data to become unreadable, or bad data to be written
  - Soft or hard errors
- Seek error
  - Arm moves to a wrong track

System Faults

- Two threats to the integrity of data from outside
  - If the power fails in the middle of a disk write
    - Sector maybe partly written
  - If the OS fails during the writing
    - Data could be affected, even if the disk is perfect and rest of the system is fail-fast
    - All the contests of volatile memory are at risk

1. Raw Disk Storage

- Untolerated error
  - Soft error: dust particles on the surface of the disk
  - Hard error: a spot on the disk may be defective
  - Hard error: previously information may decay
  - Seek error: read on the wrong track
  - Power fails during a RAW_PUT, partly write data
  - OS crashes during RAW_PUT and scribbles over disk buffer

```
RAW_SEEK (track)    // Move read/write head into position.
RAW_PUT (data)      // Write entire track.
RAW_GET (data)      // Read entire track.
```

2. Fail-fast Disk Storage

- Detected errors
  - FAIL_FAST_GET checks the error-detection code, simply reports status = BAD
  - FAIL_FAST_PUT reads back and checks fail, also report status = BAD
  - FAIL_FAST_SEEK reads track number in the first sector and finds not match, report status = BAD
  - Caller of FAIL_FAST_PUT tells it to bypass the verification step, but next FAIL_FAST_GET should detect it
  - Power fails during FAIL_FAST_PUT, later FAIL_FAST_GET should discover checksum fails (reserve of power is needed)
- Untolerated errors
  - OS crashed during FAIL_FAST_PUT and scribbles buffer
  - The data of some sector decays that is undetectable (rarely)

status ← FAIL_FAST_SEEK (*track*)

status ← FAIL_FAST_PUT (*data*, *sector_number*)

status ← FAIL_FAST_GET (*data*, *sector_number*)

3. Careful Disk Storage

- Tolerated errors
  - Soft read, write, or seek error
  - Mask by repeatedly retrying until status = OK
  - If retry count exceeds some limit, it gives up and reports
- Detected errors
  - Hard errors after several retry, by setting status = BAD
  - Power fails during a CAREFUL_PUT
- Untolerated errors
  - Crash corrupts data
  - Data decays undetectably

```
status ← CAREFUL_SEEK (track)
status ← CAREFUL_PUT (data, sector_number)
status ← CAREFUL_GET (data, sector_number)
```

3. Careful Disk Storage

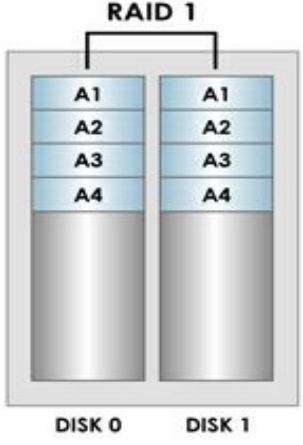
```
1   procedure CAREFUL_GET (data, sector_number)
2       for i from 1 to NTRIES do
3           if FAIL_FAST_GET (data, sector_number) = OK then
4               return OK
5       return BAD

6   procedure CAREFUL_PUT (data, sector_number)
7       for i from 1 to NTRIES do
8           if FAIL_FAST_PUT (data, sector_number) = OK then
9               return OK
10      return BAD
```

- Revectoring
  - Write data on a spare sector elsewhere on the same disk
  - Add an entry to an internal disk mapping table
    - Future GETs and PUTs use that spare one

	raw layer	fail-fast layer	careful layer	durable layer	more durable layer
soft read, write, or seek error	failure	detected	masked		
hard read, write error	failure	detected	detected	masked	
power failure interrupts a write	failure	detected	detected	masked	
single data decay	failure	detected	detected	masked	
multiple data decay spaced in time	failure	detected	detected	detected	masked
multiple data decay within T_d	failure	detected	detected	detected	failure*
undetectable decay	failure	failure	failure	failure	failure*
system crash corrupts write buffer	failure	failure	failure	failure	detected

Durable Storage: RAID 1

- Errors on reading are detected by the fail-fast layer
  - Only read one copy, unless its bad
- Virtual sector number
  - Usually involve different disks
- Tolerate error
  - Hard errors reported by careful layer are masked by reading from other replicas
- Untolerated error
  - Decay on the same sector of all the replicas, status = BAD
  - OS crashes during a DURABLE_PUT
  - Decay in a way that is undetectable



status ← DURABLE_PUT (*data*, *virtual_sector_number*)
status ← DURABLE_GET (*data*, *virtual_sector_number*)

Improving on RAID 1

- Clerk: periodically checks for decays
  - T_d as the period, must short enough
- Tolerated error
  - Hard errors reported by careful layer are masked by reading from one of the other replicas
  - Data of a single decay set decays, is discovered by the clerk, and is repaired, all within T_d of the decay event
- Untolerated error
  - The OS crashes during DURABLE_PUT
  - All decay sets fail with T_d
  - The data of some sector decays in a way undetectable

Detecting Errors Caused by System Crashes

- Outside help is need
  - Either from OS or the application
  - Calculates and includes an end-to-end checksum before initiating the disk write
  - Any program checks first before using

All-or-Nothing

Atomicity of single operation

Transaction & Atomicity

Atomicity

- An action is atomic
  - If there is no way for a higher layer to discover the internal structure of its implementation

- Atomicity = All-or-Nothing + Before-or-After
  - Atomicity of single operation
  - Atomicity of concurrent operations

All-or-Nothing Atomicity

- From the point of view of a procedure that invokes an atomic action
  - The atomic action always appears either to complete as anticipated, or to do nothing
  - This consequence is the one that makes atomic actions useful in recovering from failures

Before-or-After Atomicity

- From the point of view of a concurrent thread
  - An atomic action acts as though it occurs either completely before or completely after every other concurrent atomic action
  - This consequence is the one that makes atomic actions useful for coordinating concurrent threads

All-or-Nothing and Before-or-After Atomicity

- 1. Data abstraction
  - Hide the internal structure of data
- 2. Client/server organization
  - Hide the internal structure of major subsystems
- 3. Atomicity
  - Hide the internal structure of an action
- Enforce industrial-strength modularity
  - Guarantee absence of unanticipated interactions among components of a complex system
- The implementer's point of view
  - Painful

Golden Rule of Atomicity

- Never modify the only copy!

Example: the IBM System/370

- Multi-operand character-editing instruction
  - TRANSLATE contains 3 arguments
    - Two address: string and table
    - 8-bit counter: length
    - Takes 1 byte at a time from string, as an offset in table, get the byte in table, then replace the byte in string
  - Problem: page fault
  - Solution: dry run
    - Hidden copy of register make no change to memory
    - Maybe several dry run for one instruction
    - Then run it again
  - Problem: what if another process snatch a page?

Commit Point

ALL_OR_NOTHING_PUT

```
1.  procedure ALMOST_ALL_OR_NOTHING_PUT (data, all_or_nothing_sector)
2.                  CAREFUL_PUT(data, all_or_nothing_sector.S1)
3.      CAREFUL_PUT (data, all_or_nothing_sector.S2)
4.      CAREFUL_PUT (data, all_or_nothing_sector.S3)
5.  procedure ALL_OR_NOTHING_GET (reference date,all_or_nothing_sector)
6.      CAREFUL_GET (data1, all_or_nothing_sector.S1)
7.      CAREFUL_GET (data2, all_or_nothing_sector.S2)
8.      CAREFUL_GET (data3, all_or_nothing_sector.S3)
9.      if (data1 = data2) data ← data1
10.     else data ← data3
```

ALL_OR_NOTHING_PUT

1. **procedure** ALL_OR_NOTHING_PUT (*data, all_or_nothing_sector*)
2. CHECK_AND_REPAIR (*all_or_nothing_sector*)
3. ALMOST_ALL_OR_NOTHING_PUT (*data*, *all_or_nothing_sector*)

4. **procedure** CHECK_AND_REPAIR (*all_or_nothing_sector*)
// Ensure copies match
5. CAREFUL_GET (*data1*, *all_or_nothing_sector.S1*)
6. CAREFUL_GET (*data2*, *all_or_nothing_sector.S2*)
7. CAREFUL_GET (*data3*, *all_or_nothing_sector.S3*)

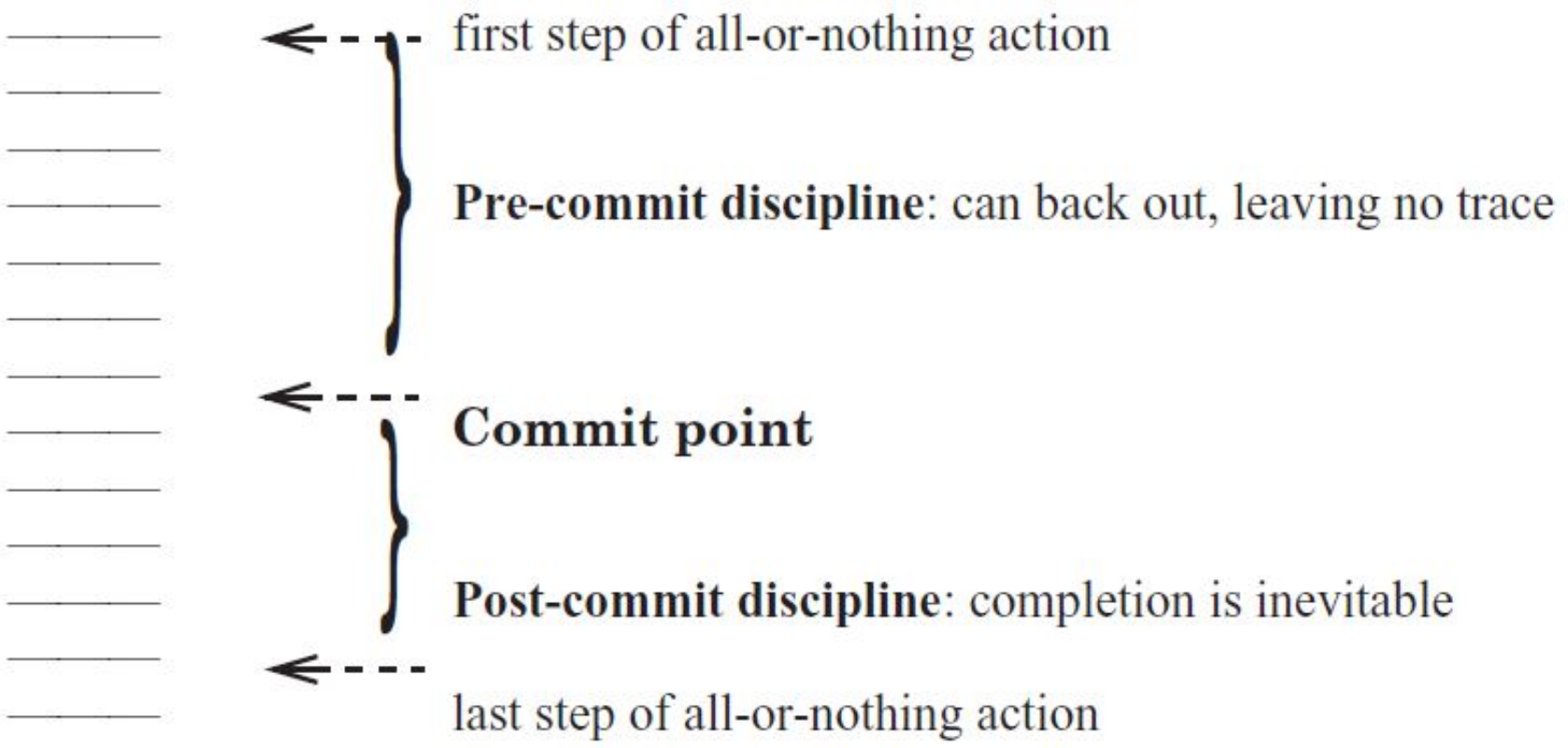
11

ALL_OR_NOTHING_PUT

8. **if** (*data1* = *data2*) **and** (*data2* = *data3*) **return** // State 1 or 7, no repair
9. **if** (*data1* = *data2*)
10. CAREFUL_PUT (*data1, all_or_nothing_sector.S3*) **return** // State 5 or 6.
11. **if** (*data2* = *data3*)
12. CAREFUL_PUT (*data2, all_or_nothing_sector.S1*) **return** // State 2 or 3.
13. CAREFUL_PUT (*data1, all_or_nothing_sector.S2*) // State 4, go to state 5
14. CAREFUL_PUT (*data1, all_or_nothing_sector.S3* // State 5, go to state 7

data state:	1	2	3	4	5	6	7
sector *S1*	old	bad	new	new	new	new	new
sector *S2*	old	old	old	bad	new	new	new
sector *S3*	old	old	old	old	old	bad	new

Commit Point



13

Shadow Copy

Bank Account Transfer

```
xfer(bank, a, b, amt):
    bank[a] = bank[a] - amt
    bank[b] = bank[b] + amt

audit(bank):
    sum = 0
    for acct in bank:
        sum = sum + bank[acct]
    return sum
```

audit(bank):

<- sum=200

<- sum=150

<- sum=200

Shadow Copy

```
xfer(bank, a, b, amt):
    bank[a] = read_accounts(bankfile)
    bank[a] = bank[a] - amt
    bank[b] = bank[b] + amt
    write_accounts(#bankfile)
    rename("#bankfile", bankfile)
```


16

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 12
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 1
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 1

17

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode **13**
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 1
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 1

18

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 1
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

19

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 1

Problem

- Two names point fnew's inode, but refcount is 1
- Problem is that we don't know which reference is the correct one

Second Try: Increase ref-count First

```
rename(x, y):
    newino = lookup(x)
    oldino = lookup(y)

    incref(newino)
    change y's dirent to newino
    decref(oldino)
    remove x's dirent
    decref(newino)
```

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 12
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 1
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

25

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode **13**
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 1
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 2

rename("#bank", "bank")

- Directory data blocks:
  - filename "bank" → inode 13
  - filename "#bank" → inode 13
- inode 12:
  - data blocks: 3, 4, 5
  - refcount: 0
- inode 13:
  - data blocks: 6, 7, 8
  - refcount: 1

Recovery After Crash

```
salvage(disk):
  for inode in disk.inodes:
    inode.refcnt =
      find_all_refs(disk.root_dir, inode)
  if exists(“#bank”):
    unlink(“#bank”)
```


30

Shadow Copy

- Write to a copy of data, atomically switch to new copy
- Switching can be done with one all-or-nothing operation (sector write)
- Requires a small amount of all-or-nothing atomicity from lower layer (disk)
- Main rule: only make one write to current/live copy of data
  - In our example, sector write for rename
  - Creates a well-defined commit point

31

Shadow Copy

- Does the shadow copy approach work in general?
  - + Works well for a single file
  - – Hard to generalize to multiple files or directories
    - Might have to place all files in a single directory, or rename subdirs
  - – Requires copying the entire file for any (small) change
  - – Only one operation can happen at a time
  - – Only works for operations that happen on a single computer, single disk

All-or-nothing In Transaction

Transactions: A Programming Model

- All-or-nothing ("Atomic" in the database literature, but "All-or-nothing" in 6.033)
- Before-or-after ("Isolation")
- Effects persist ("Durable)
- "Consistent": satisfies higher-level constraints (e.g., all salaries > 0)
- Aka "ACID"

Example: Bank Account App

```
xfer(bank, a, b, amt):
    copy(bank, tmp)
    tmp[a] = tmp[a] - amt
    tmp[b] = tmp[b] + amt
    rename(tmp, bank)
```

Shadow Copy Abort VS. Commit

```
xfer(bank, a, b, amt):
        copy(bank, tmp)
        tmp[a] = tmp[a] – amt
        tmp[b] = tmp[b] + amt
        if tmp[a] < 0:
                print “Not enough funds”
                unlink(tmp)
        else:
                rename(tmp, bank)
```

Transaction Terminology

```
xfer(bank, a, b, amt):
    begin
      bank[a] = bank[a] - amt
      bank[b] = bank[b] + amt
      if bank[a] < 0:
            print "Not enough funds"
            abort
      else:
            commit
```

Consider the Bank Account Example

- Two accounts: A and B.
  - Accounts start out empty.
  - Run these all-or-nothing actions

```
begin
    A = 100
    B = 50
    commit

    begin
    A = A - 20
    B = B + 20
    commit

    begin
    A = A + 30
    --CRASH--
```


38

A Log Sample

TID	T1	T1	T1	T2	T2	T2	T3
OLD	A=0	B=0		A=100	B=50		A=80
NEW	A=100	B=50	COMMIT	A=80	B=70	COMMIT	A=110

```
begin
A = 100
B = 50
commit

begin
A = A - 20
B = B + 20
commit

begin
A = A + 30
--CRASH--
```

- Begin
- Write variable
- Read variable
- Commit
- Abort

Logging

- What happens when a program runs now?
  - begin: allocate a new transaction ID
  - write variable: append an entry to the log
  - read variable: scan the log looking for last committed value
    - As an aside: how to see your own updates?
    - Read uncommitted values from your own tid

Read with a Log

```
read(log, var):
    commits = { }
    for record r in log[len(log)-1] .. log[0]:
        if (r.type == commit):
            commits = commits + r.tid
        if (r.type == update)
                and (r.tid in commits)
                and (r.var == var):
            return r.new_val
```

Logging

- Commit: write a commit record
  - Expectedly, writing a commit record is the "commit point" for action, because of the way read works (looks for commit record)
  - However, writing log records better be all-or-nothing
    - One approach, from last time: make each record fit within one sector
- Abort:
  - Do nothing or write a record?
    - could write an abort record, but not strictly needed
- Recover from a crash: do nothing

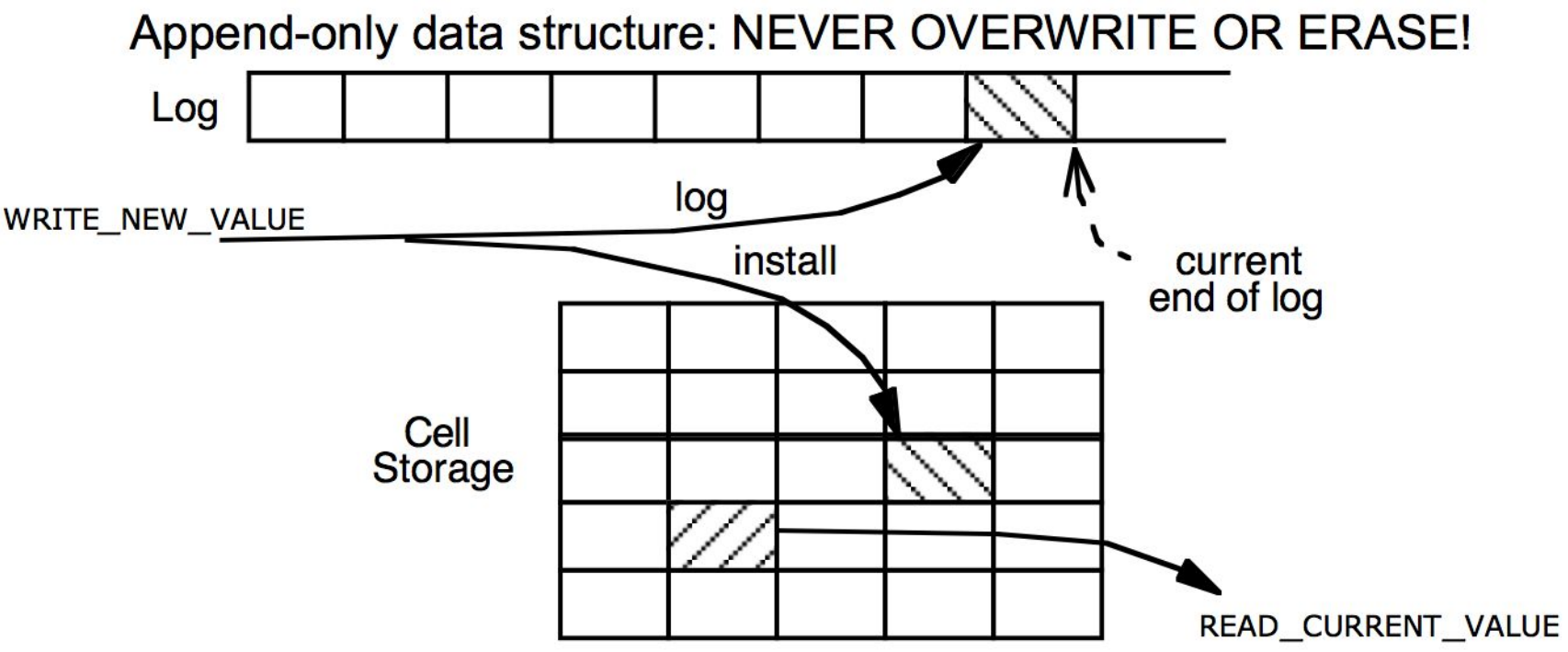
Performance of Read

- What's the performance of this log-only approach?
  - Write performance is probably good: sequential writes, instead of random.
  - (Since we aren't using the old values yet, we could have skipped the read.)
- Read performance is terrible: scan the log for every read!
  - Crash recovery is instantaneous: nothing to do.

Performance Optimization

- How can we optimize read performance?
  - Keep both a log and "cell storage"
  - Log is just as before: authoritative, provides all-or-nothing atomicity
  - Cell storage: provides fast reads, but cannot provide all-or-nothing
- Terminology
  - "log" an update when it's written to the log
  - "install" an update when it's written to cell storage

Cell Storage (Home) + Log Storage



Read / write with Cell Storage

```
read(var):
    return cell_read(var)


write(var, value):
    log.append(cur_tid, update,
                var, read(var), value)
    cell_write(var, value)
```

Read / write with Cell Storage

- Two questions we have to answer now:
  - How to update both the log and cell storage when an update happens?
  - How to recover cell storage from the authoritative log after crash?

47

Order Matters

- Ordering of logging and installing
  - The two together don't have all-or-nothing atomicity
  - Can crash in-between, so just one of the two might have taken place
- What happens if we install first and then log?
  - If we crash, no idea what happened to cell storage, or how to fix it
- The corresponding rule for logging is the "Write-ahead-log protocol" (WAL)
  - Log the update before installing it
- If we crash, log is authoritative and intact, can repair cell storage
  - Can think of it as not being the only copy, once it's in the log

Recovering Cell Storage

- What happens if we log an update, install it, but then abort/crash?
  - Need to undo that installed update.
  - Plan: scan log, determine what actions aborted ("losers"), undo them.
- Why do we have to scan backwards?
  - Need to undo newest to oldest
  - Also need to find outcome of action before we decide whether to undo

Recovering Cell Storage from Log

```
recover(log):
  done = { }, aborted = { }
  for record r in log[len(log)-1] .. log[0]:
      if r.type == commit or r.type == abort:
        done = done + r.tid
      if r.type == update and r.tid not in done:
        cell_write(r.var, r.old_val) # undo
      aborted = aborted + r.tid
  for tid in aborted: log.append(tid, abort)
```


50

Performance Now

- Writes might still be OK
  - but we do write twice: log & install
- Reads are fast
  - just look up in cell storage
- Recovery requires scanning the entire log
- Remaining performance problems:
  - We have to write to disk twice
  - Scanning the log will take longer and longer, as the log grows

Optimization 1: Defer Installing Updates

- Storing installing in a cache
  - Writes can now be fast: just one write, instead of two
    - Hope that variable is modified several times in cache before flush
  - Reads go through the cache, since cache may contain more up-to-date values
  - Atomicity problem: cell storage (on disk) may be out-of-date
    - Is it possible to have changes that should be in cell storage, but aren't?
    - yes: might not have flushed the latest commits
    - Is it possible to have changes that shouldn't be in cell storage, but are?
    - yes: flushed some changes that then aborted (same as before)

Cached read/write

```
read(var):
    if var not in cache:
        # may evict others from cache to cell store
        cache[var] = cell_read(var)
    return cache[var]

write(var, value):
    log.append(cur_tid, update,
               var, read(var), value)
    cache[var] = value
```

Recovery

- Need to go through and re-apply changes to cell storage
  - undo every abort (even if it had an explicit record)
  - redo every commit
- Don't treat actions with an abort record as "done"
  - there might be leftover changes from them in cell storage
- Re-do any actions that are committed (in the "done" set now)

Recovery for Cached Writes

```
recover(log):
    done = { }
    for record r in log[len(log)-1] .. log[0]:
        if r.type == commit or r.type == abort:
            done = done + r.tid
        if r.type == update and r.tid not in done:
            cell_write(r.var, r.old_val) # undo
    for record r in log[0] .. log[len(log)-1]:
        if r.type == update and r.tid in done:
            cell_write(r.var, r.new_val) # redo
```

Optimization 2: Truncate the Log

- Current log grows without bound: not practical.
  - What part of the log can be discarded?
    - Must know the outcome of every action in that part of log
    - Cell storage must reflect all of those log records (commits, aborts).
  - Truncating mechanism (assuming no pending actions):
    - Flush all cached updates to cell storage
    - Write a checkpoint record, to save our place in the log.
    - Truncate log prior to checkpoint record
    - (Often log implemented as a series of files, so can delete old log files.)
  - With pending actions, delete before checkpoint & earliest undecided record

Before-or-After

Atomicity of current operations

Before-or-after

Before-or-after atomicity

- Transaction
  - All-or-nothing: single operation, crash
  - Before-or-after: multiple operations, concurrency
  - In the midst of multiple-step atomic action
- Definition of "correctness" (as in 9.1.5)
  - If every result is guaranteed to be one that could have been obtained by some purely serial application of those same actions
  - Serializability

Concurrent actions

```
xfer(a, b, amt):
    begin
    a = a – amt
    b = b + amt
    commit

interest(rate):
    begin
    for each account x:
        x = x * (1+rate)
    commit
```

Serial executions

- What happens if we have A=100, B=50, and concurrent xfer(A,B,10) & int(0.1)?
- Serial executions:
  - .. -> [xfer] -> A=90, B=60 -> [int] -> A=99, B=66
  - .. -> [int] -> A=110, B=55 -> [xfer] -> A=100, B=65

Schedule1

xfer:	int:
RA [100]	
WA [90]	
	RA [90]
	WA [99]
RB [50]	
WB [60]	
	RB [60]
	WB [66]

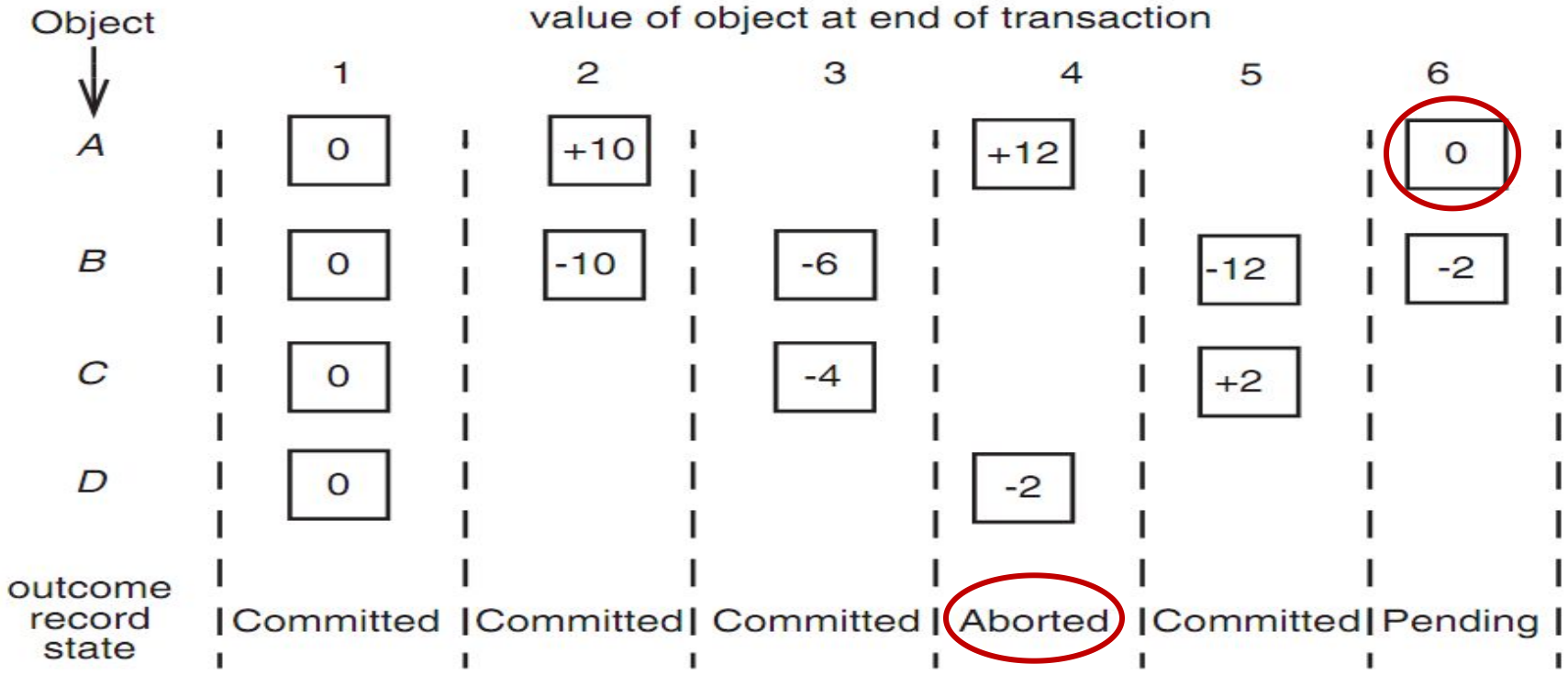
Schedule 2

xfer:	int:
RA [100]	
	RA [100]
WA [90]	
	WA [110]
RB [50]	
WB [60]	
	RB [60]
	WB [66]

Simple serialization

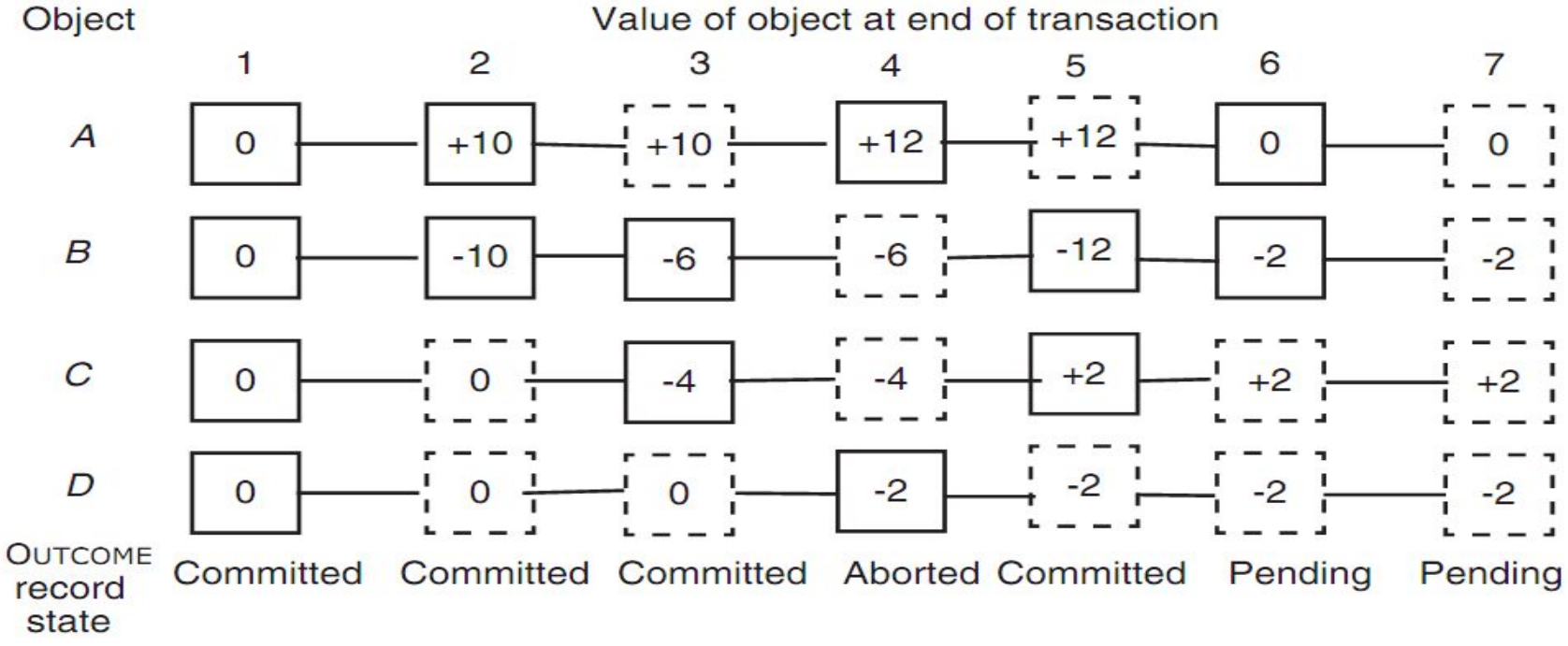
```
1  procedure BEGIN_TRANSACTION ()
2      id ← NEW_OUTCOME_RECORD (PENDING)          // Create, initialize, assign id.
3      previous_id ← id − 1
4      wait until previous_id.outcome_record.state ≠ PENDING
5      return id
```

- Simple serialization: similar as lock-step
  - Transaction n waits transaction n-1 to complete
- Drawbacks: too strict
  - Prevents all concurrency among transactions
  - Suitable for applications without many transactions
- Next of this chapter are nothing but optimizations



transaction

1: initialize all accounts to 0
2: transfer 10 from *B* to *A*
3: transfer 4 from *C* to *B*
4: transfer 2 from *D* to *A* (aborts)
5: transfer 6 from *B* to *C*
6: transfer 10 from *A* to *B*



- More relaxed disciplines that still guarantee correctness
- We don't care when things happen
  - Transaction-3 can create new version of C before transaction-2
  - Transaction-4 can run concurrently with transaction-3

Mark-point Discipline

- Step-1: wait for pending version
- Step-2: mark data to be updated
  - Create a pending versions of every variable it intends to modify --- mark point
  - Announce when it is finished doing so
    - By MARK_POINT_ANNOUNCE, simple set a flag
- Step-3: keep discipline of mark point:
  - No transaction can begin reading its inputs until the preceding transaction has reached its mark point or is no longer pending

```
1   procedure TRANSFER (reference debit_account, reference credit_account,
2                                                                  amount)
3       my_id ← BEGIN_TRANSACTION ()
4       NEW_VERSION (debit_account, my_id)
5       NEW_VERSION (credit_account, my_id)
6       MARK_POINT_ANNOUNCE (my_id);
7       xvalue ← READ_CURRENT_VALUE (debit_account, my_id)
8       xvalue ← xvalue - amount
9       WRITE_VALUE (debit_account, xvalue, my_id)
10      yvalue ← READ_CURRENT_VALUE (credit_account, my_id)
11      yvalue ← yvalue + amount
12      WRITE_VALUE (credit_account, yvalue, my_id)
13      if xvalue > 0 then
14          COMMIT (my_id)
15      else
16          ABORT (my_id)
17          signal("Negative transfers are not allowed.")
```

WRITE_NEW_VERSION() using Markpoint

```
1   procedure NEW_VERSION (reference data_id, this_transaction_id)
2       if this_transaction_id.outcome_record.mark_state = MARKED then
3           signal ("Tried to create new version after announcing mark point!")
4       append new version v to data_id
5       v.value ← NULL
6       v.action_id ← transaction_id

7   procedure WRITE_VALUE (reference data_id, new_value, this_transaction_id)
8       starting at end of data_id repeat until beginning
9           v ← previous version of data_id
10          if v.action_id = this_transaction_id
11              v.value ← new_value
12              return
13      signal ("Tried to write without creating new version!"))
```

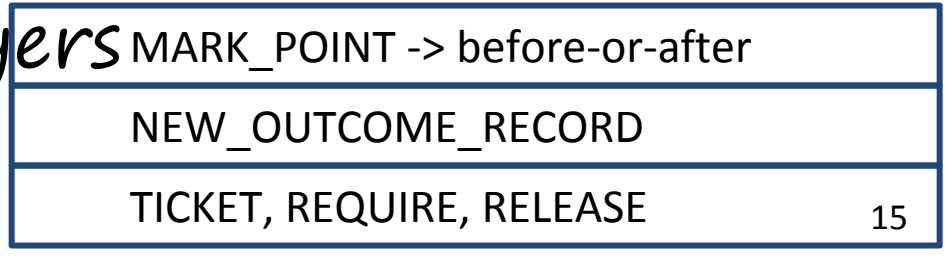
```
1  procedure READ_CURRENT_VALUE (data_id, caller_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id          // Get next older version
4          a ← v.action_id  // Identify the action a that created it
5          s ← a.outcome_record.state               // Check action a's outcome record
6          if s = COMMITTED then
7              return v.value
8          else skip v                              // Continue backward search
9      signal ("Tried to read an uninitialized variable!")
```

Waiting, instead of skipping

```
1  procedure READ_CURRENT_VALUE (data_id, this_transaction_id)
2      starting at end of data_id repeat until beginning
3          v ← previous version of data_id
4          last_modifier ← v.action_id
5          if last_modifier ≥ this_transaction_id then skip v        // Keep searching
6          wait until (last_modifier.outcome_record.state ≠ PENDING)
7          if (last_modifier.outcome_record.state = COMMITTED)
8              then return v.state
9              else skip v                                          // Resume search
10     signal ("Tried to read an uninitialized variable")
```

Mark-point Discipline

- Distribute delays
  - Some in BEGIN_TRANSACTION()
  - Some in READ_CURRENT_VALUE()
- Bootstrapping
  - Goal: before-or-after for general programs
  - Solution: special case of a "new outcome record"
- Three layers



```
1   procedure BEGIN_TRANSACTION ()
2       id ← NEW_OUTCOME_RECORD (PENDING)          // Create, initialize, assign id.
3       previous_id ← id − 1
4       wait until previous_id.outcome_record.state ≠ PENDING
5       return id
```

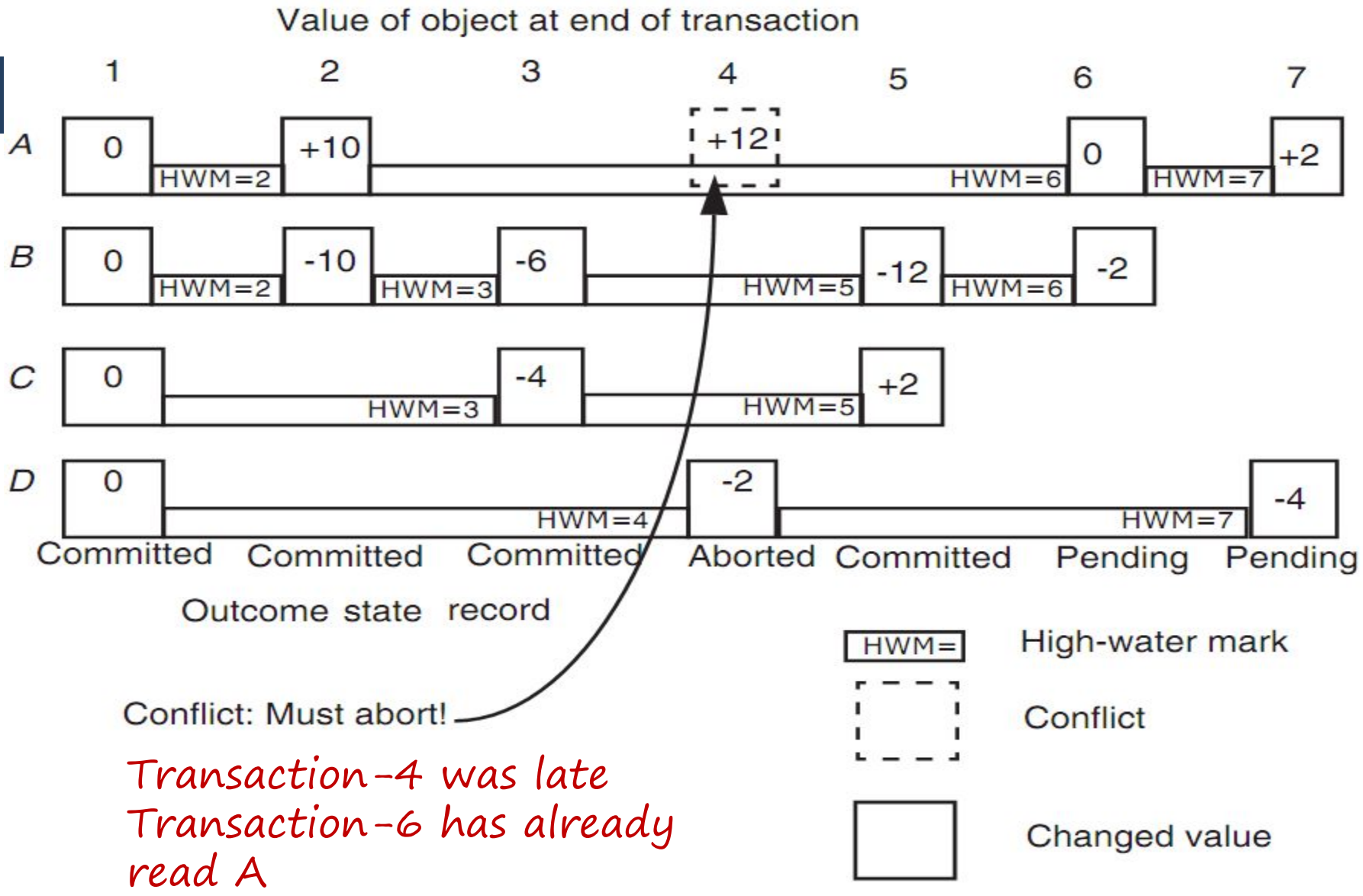
```
1   procedure BEGIN_TRANSACTION ()
2       id ← NEW_OUTCOME_RECORD (PENDING)
3       previous_id ← id - 1
4       wait until (previous_id.outcome_record.mark_state = MARKED)
5           or (previous_id.outcome_record.state ≠ PENDING)
6       return id

7   procedure NEW_OUTCOME_RECORD (starting_state)
8       ACQUIRE (outcome_record_lock)          // Make this a before-or-after action.
9       id ← TICKET (outcome_record_sequencer)
10      allocate id.outcome_record
11      id.outcome_record.state ← starting_state
12      id.outcome_record.mark_state ← NULL
13      RELEASE (outcome_record_lock)
14      return id

15  procedure MARK_POINT_ANNOUNCE (reference this_transaction_id)
16      this_transaction_id.outcome_record.mark_state ← MARKED
```

Mark-point: no deadlock

- Wait only earlier transaction
  - Earliest ones wait no one
  - Guarantee progress
    - Lock will not guarantee progress,
    - requires additional mechanisms to ensure no deadlock
- Two minor points
  - Reduce to simple serialization discipline
    - If wait to announce mark point until commit or abort
  - Two possible errors
    - Never call NEW_VERSION after mark point
    - Never try to write a value without new version

Read-capture: Optimistic Atomicity

- Pessimistic methods
  - Presume that interference is likely
  - Prevent any possibility of interference actively
  - Simple serialization & mark-point
- Optimistic methods
  - Allow write in any order and at any time
  - With the risk that "sorry, interfere write, you must abort, clear the history and then retry"
  - Read-capture discipline



Transaction-4 was late
Transaction-6 has already read A

```
1   procedure READ_CURRENT_VALUE (reference data_id, value, caller_id)
2       starting at end of data_id repeat until beginning
3           v ← previous version of data_id
4           if v.action_id ≥ caller_id then skip v
5           examine v.action_id.outcome_record
6               if PENDING then
7                   WAIT for v.action_id to COMMIT or ABORT
8                   if COMMITTED then
9                       v.high_water_mark ← max(v.high_water_mark, caller_id)
10                      return v.value
11                  else skip v                          // Continue backward search
12      signal ("Tried to read an uninitialized variable!")

13  procedure NEW_VERSION (reference data_id, caller_id)
14      if (caller_id < data_id.high_water_mark)         // Conflict with later reader.
15      or (caller_id < (LATEST_VERSION[data_id].action_id))     // Blind write conflict.
16      then ABORT this transaction and terminate this thread
17      add new version v at end of data_id
18      v.value ← 0
19      v.action_id ← caller_id

20  procedure WRITE_VALUE (reference data_id, new_value, caller_id)
21      locate version v of data_id.history such that v.action_id = caller_id
22              (if not found, signal ("Tried to write without creating new version!"))
23      v.value ← new_value
```

Read-capture's Correctness

- Correctness
  - 1. WAIT for PENDING in READ ensures that transaction n will wait for k to commit or abort (k<n)
  - 2. High-water mark in READ and test in NEW_VERSION ensures transaction j will abort if n has read the object (j < n)
  - 3. Therefore, every value that READ returns to transaction n will include effect of 1…n-1
  - 4. Therefore, every transaction n will act as if it serially follows transaction n-1
- Price of optimism
  - Later transaction may cause earlier ones to abort
  - Suitable for those application without a lot of data interference

Locking

Pragmatics: lock

- Lock: a flag associated with a data object to warn concurrent actions not to read or write the object
  - ACQUIRE (A.lock) / RELEASE (A.lock)
    - Only one will succeed
  - Problems
    - Easy to make error to race
    - Difficult to find out why
  - Three steps
    - Discipline specifies which locks must be acquired and when
    - Establish a compelling line of reasoning that concurrent transactions that follow the discipline will ensure before-or-after
    - Interpose a lock manager to enforces the discipline

System-wide locking

- System-wide lock
  - begin_transaction
    ACQUIRE (System.lock)
    …
  - …
    RELEASE (System.lock)
    end_transaction
  - Allow only one transaction to run at a time
  - Serialize potentially concurrent transactions in the order that they call ACQUIRE

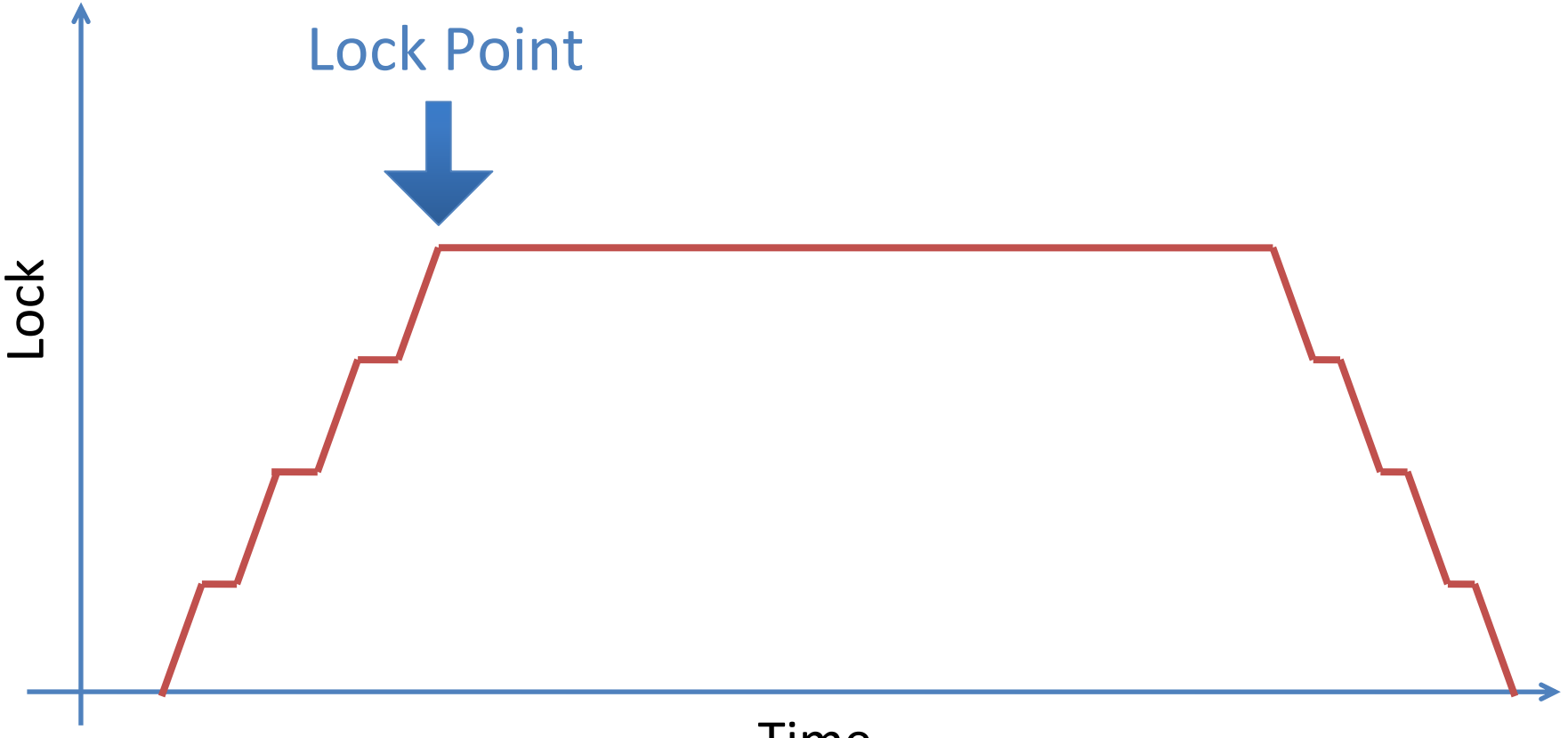
id ← NEW_OUTCOME_RECORD ()
preceding_id ← *id* - 1
wait until *preceding_id.outcome_record.value* ≠ PENDING
…
COMMIT (*id*) [or ABORT (*id*)]

ACQUIRE (*System.lock*)
…
RELEASE (*System.lock*)

Simple Locking

- Simple locking
  - 1. Acquire a lock for every shared data in advance
  - 2. Release locks only after commit or abort
- Lock point (similar as mark-point)
  - Lock set: locks acquired when reaches lock point
  - Lock manager's enforcement
    - Intercept read/write/commit/abort, and check
- Problems
  - How to enumerate all shared object to access?
  - The set of might access may be larger than does access

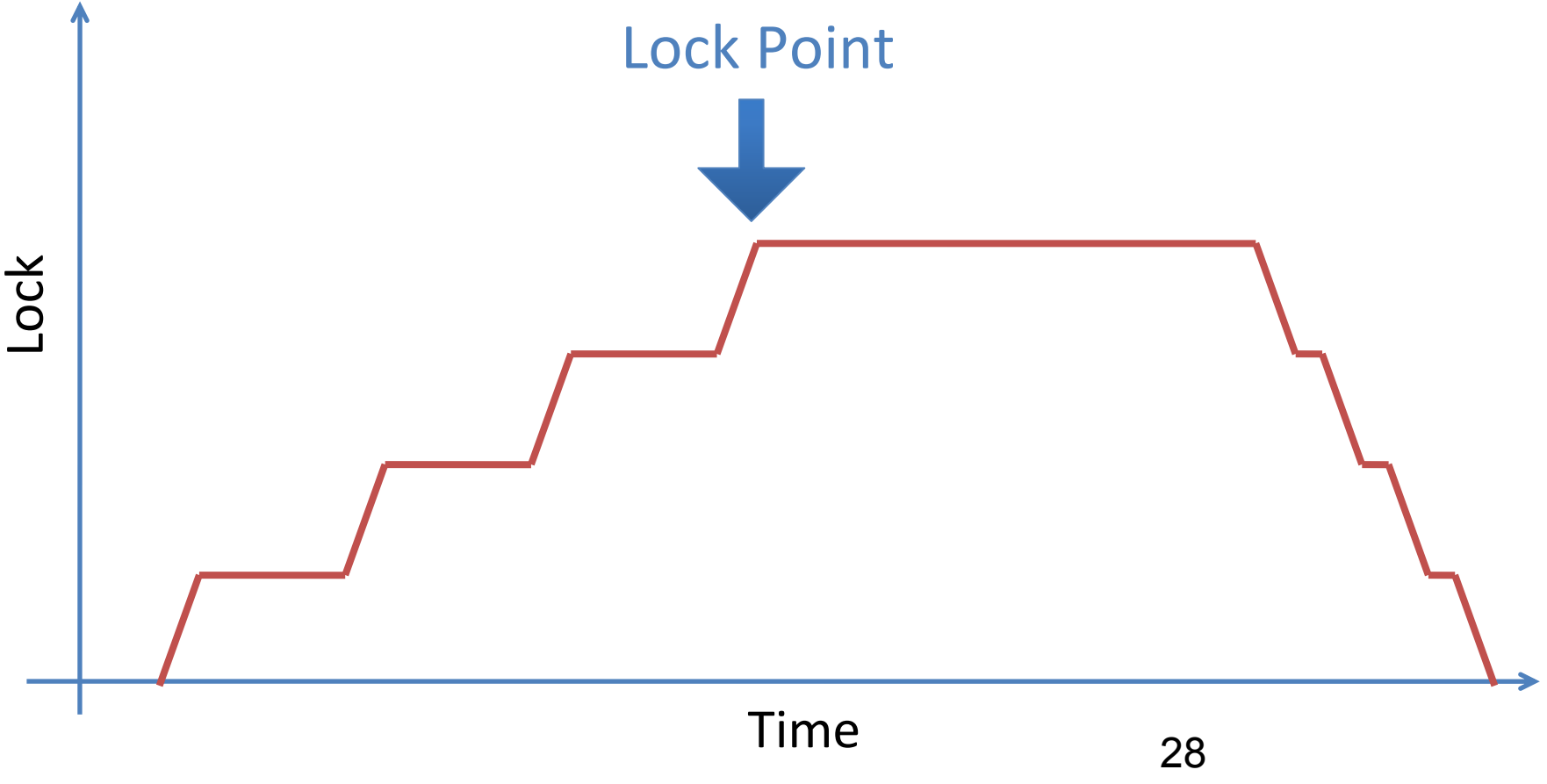
Simple Locking



Two-phase Locking

- Two-phase locking
  - Avoids to know lock set in advance
  - Acquire locks as it proceeds, access data as soon as it acquires the lock
- Constraints
  - Not release any locks until passes lock point
  - Only release a lock if never need to read/write again

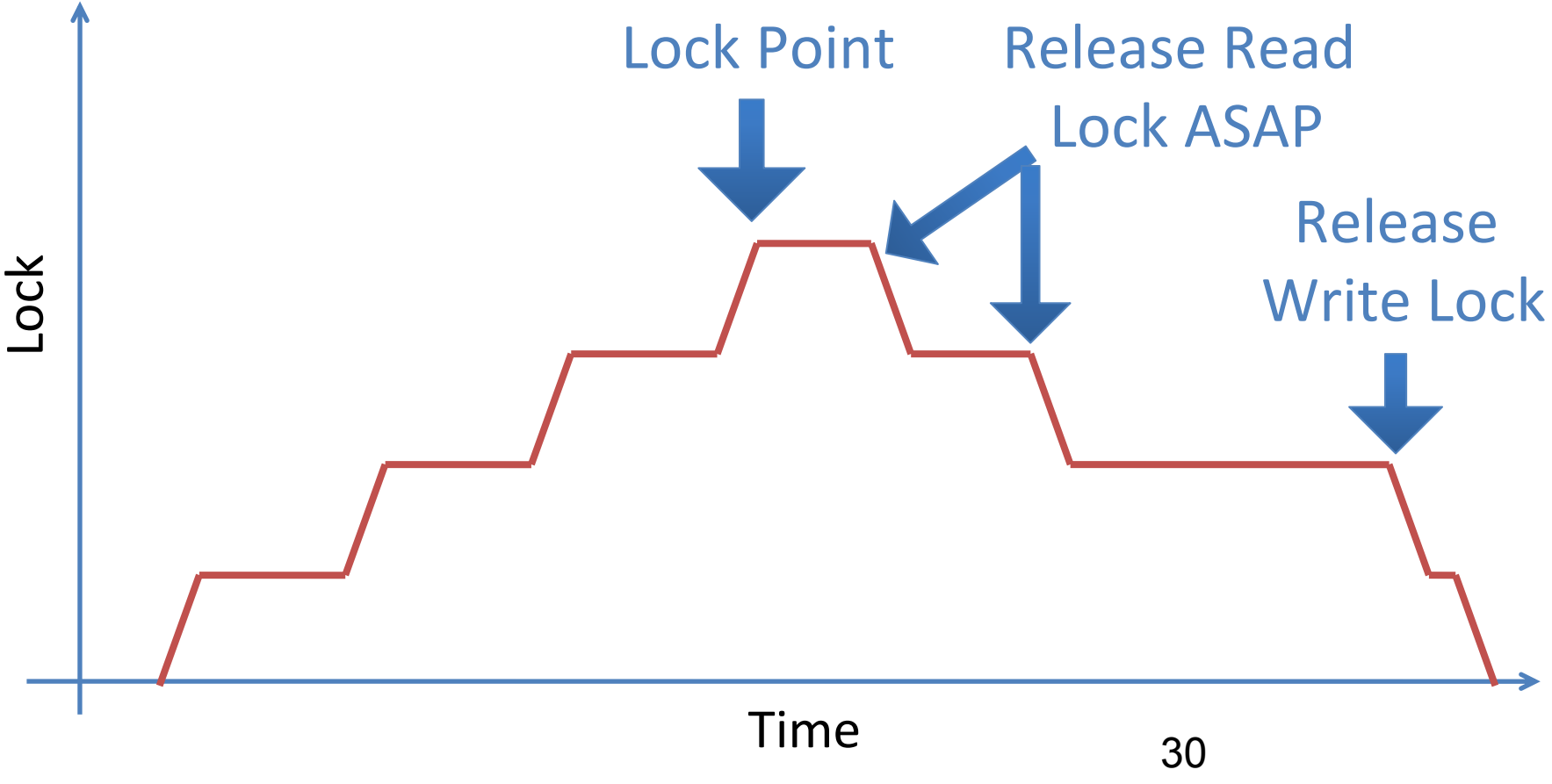
Two-phase Locking



Performance Optimizations

- Lock compatibility modes
  - Multiple-reader, single-writer protocol
    - Any number of readers is safe
    - Only one writer, wait for all reader to finish
    - Suitable for applications with a lot of reading
    - A writer may be delayed indefinitely
  - More specific, more complex

Using Read-write Lock



Two-phase locking

- Unnecessary blocking
  - Example
    - T1: READ X
    - T2: WRITE Y
    - T1: WRITE Y
    - T1's & T2's lock sets intersect at Y
  - Two-phase locking prevents interleaving
    - But T1/T2/T1 = T2/T1/T1
    - NP-complete

Interactions between locks and logs

- Transaction abort
  - Restore its changed data before releasing lock
  - Just like committed transactions doing nothing
- System recovery
  - Whether the locks themselves should be logged?
    - No pending after recovery, thus no locks
    - Locks are in volatile memory
  - Non-complete transactions have no overlapping lock sets at the moment of crash

Performance Optimizations

- Physical locking VS. logical locking
  - Choose lock granularity
    - E.g. change 6-byte object of a 1000-byte disk sector, or change 1500-byte object on two disk sectors
    - Which to lock: the object or sector?
  - Logical locking
    - If objects are small: more concurrency, more logging
  - Physical locking
    - New logical layer between app and disk
      - E.g. data object management and garbage collection
    - Tailor the logging and locking design to match disk granularity
      - Disk sectors rather than object is a common practice

Deadlock & Making Progress

- Inevitable if using locks in concurrency
  - 1. Waiting for one another
  - 2. Waiting for a lock by some deadlocked one
  - Correctness arguments ensures correctness, but no progress
- Methods
  - Pessimistic ones: take a priori action to prevent
  - Optimistic ones: detect deadlocks then fix up

Methods for solving deadlock

- Lock ordering (pessimistic)
  - Number the locks uniquely
  - Require transactions acquire locks in order
  - Problem: some app may not predict all of the locks they need before acquiring the first one
- Backing out (optimistic)
  - Allow acquire locks in any order
  - If it encounters an already-acquired lock with an number lower than one it has previously acquired itself, then
    - UNDO: Back up to release its higher-numbered locks
    - Wait for the lower-numbered lock and REDO

Methods for solving deadlock

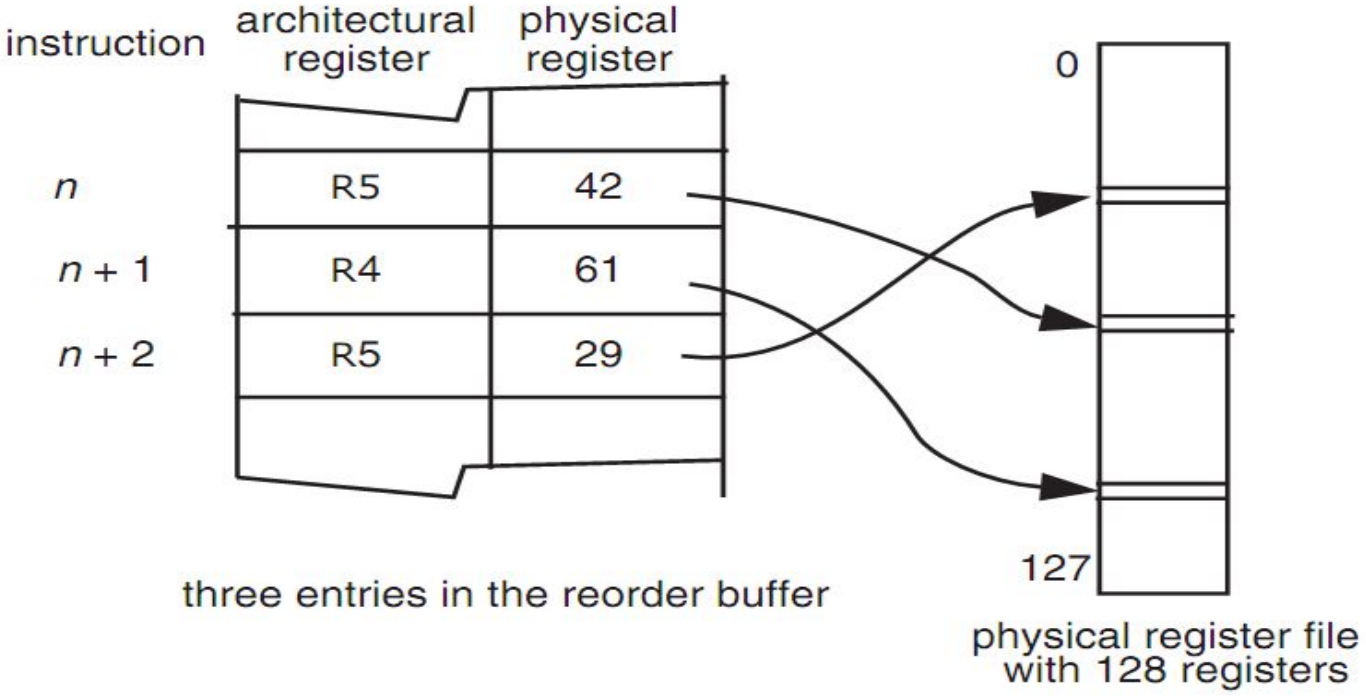
- Timer expiration (optimistic)
  - Set a timer at begin_transaction, abort if timeout
  - If still no progress, another one may abort
  - Problem: how to chose the interval?
- Cycle detection (optimistic)
  - Maintain a wait-for-graph in the lock manager
    - Shows owner and waiting ones
    - Check when transaction tries to acquire a lock
  - Prevent cycle (deadlock)
    - Select some cycle member to be a victim

History Version & Isolation

Using version histories

- Register renaming
  - Pentium-4 only has 8 architectural register
  - Has 128 physical register
  - Renaming by reorder buffer
    - Assigning a slot in the reorder buffer
      - NEW_OUTCOME_RECORD & NEW_VERSION
    - Committing instruction
      - WRITE_VALUE & COMMIT

Register renaming




```
n       R5 ← R4 × R2          // Write a result in register five.
n + 1   R4 ← R5 + R1          // Use result in register five.
n + 2   R5 ← READ (117492)    // Write content of a memory cell in register five.
```

Using version histories

- Oracle database's serializable
  - Snapshot isolation
  - When a transaction begins
    - System takes a snapshot of every committed value
    - Read all of its inputs from that snapshot
  - If two concurrent transaction modify the same variable
    - The first one to commit wins
    - Aborts the other one with "serialization error"

Using version histories

- Transaction memory
  - Allow concurrent threads without locks
    - Mark the beginning of an atomic instruction sequence with a “begin transaction” instruction
    - Direct all STORE to a hidden copy
    - Check interference at end
  - Even more optimistic than read-capture
  - Most useful if interference is possible but unlikely
  - Hardware or software implementation

Snapshot Isolation

- Setup: table with doctors, oncall=true

- T1:

```
select count(*) from doctors where oncall=true;
If count > 1
    update doctors set oncall=false where username = 'alice';
```

- T2:

```
select count(*) from doctors where oncall=true;
If count > 1
    update doctors set oncall=false where username = 'bob';
```

Read-committed Isolation

- Setup: table with doctors, oncall=false

- T1:
  select count(*) from doctors where oncall=false;

  select count(*) from doctors where oncall=false;
  commit;
- T2:
  update doctors set oncall=true where username = 'bob';
  commit;

Summary

- Serialization (before-or-after)
  - Simple serialization
  - Mark-point
  - Read-capture
- Other Isolation
  - Read uncommitted
  - Read committed <- PostgreSQL default
  - Repeat Read
  - Serializability
  - Snapshot isolation <- Oracle db
- Locking
  - System-wide locking
  - Simple locking
  - Two-phase locking
- Deadlock
  - Prevention
  - Detection
  - Solving

Transaction: multi-site

Multiple-site Atomicity: Distributed Two-phase Commit

- Multiple-site atomicity
  - A transaction requires several transactions at different sites in a best-effort network
    - 1 machine holds accounts A-K
    - 1 machine holds accounts K-Z
  - Message may be lost, delayed or duplicated
  - Use RPC to communicate
    - Ensure at-least-once by persistent sender
    - Ensure at-most-once by duplicate suppression
    - However, neither is enough to ensure atomicity here
  - Combine three components
    - At-lease-once, at-most-once, one-site transaction

Two-phase Commit

- Phase-1: preparation / voting
  - Lower-layer transactions either aborts or tentatively committed
  - Higher-layer transaction evaluate lower situation
- Phase-2: commitment
  - If top-layer, then COMMIT or ABORT
  - If nested itself, then become tentatively committed

Multiple-site Atomicity

- Worker: Bob, Charles, Dawn
  - Does three transactions: X, Y, Z
- Coordinator: Alice
  - Create a higher-layer transaction
  - Send three messages to the three workers
- Challenge: un-reliable communication

From:Alice
To: Bob
Re: my transaction 271

Please do X as part of my transaction.

From:Bob
To: Alice
Re: your transaction 271

My part X is ready to commit.

Commit Phase-1

From:Alice
To: Bob
Re: my transaction 271

PREPARE to commit X.

From:Bob
To:Alice
Re: your transaction 271

I am PREPARED to commit my part. Have you decided to commit yet? Regards.

Bob: tentative committed

State: PREPARED

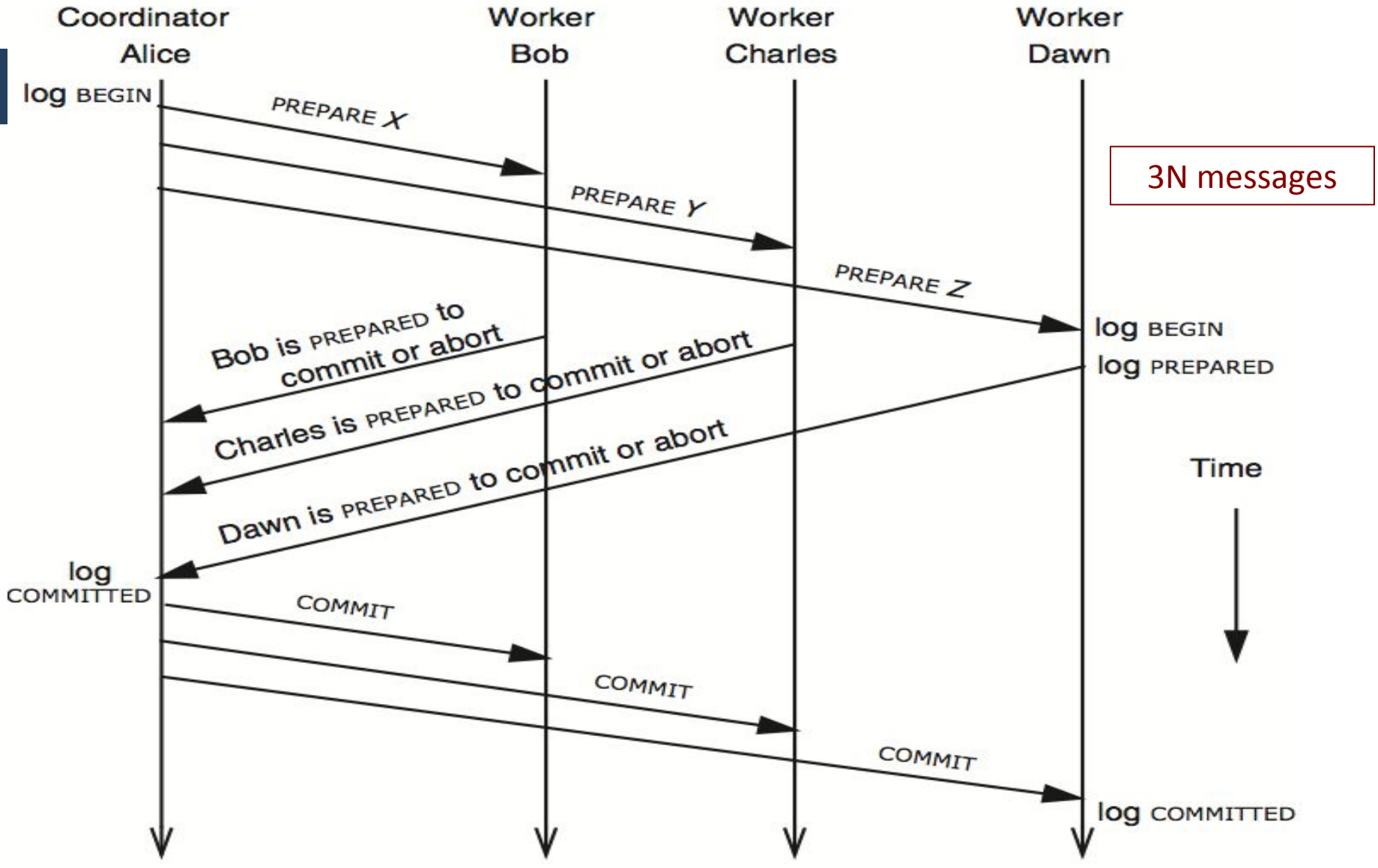
Commit Phase-2

From:Alice
To:Bob
Re: my transaction 271

My transaction committed. Thanks for your help.

Bob: COMMITTED

Bob will perform post-commit


Coordinator
Alice
Worker
Bob
Worker
Charles
Worker
Dawn
log BEGIN
PREPARE X
PREPARE Y
PREPARE Z
3N messages
log BEGIN
log PREPARED
Bob is PREPARED to commit or abort
Charles is PREPARED to commit or abort
Dawn is PREPARED to commit or abort
Time
log COMMITTED
COMMIT
COMMIT
COMMIT
log COMMITTED

Multiple-site Atomicity

- Coordinator
  - Collect some ABORT or nothing: ABORT or assign the work to another worker
  - Collect all COMMIT: then COMMIT
- Worker
  - When receive nothing: resend PREPARED
    - Coordinator will send current state if it receives duplicate message
  - When receive COMMIT: then COMMIT

Worker Crash Recovery

- If a worker uses logs and locks crashes
  - 1. Classify any PREPARED transaction as a tentative winner that it should restore to the PREPARED state
  - 2. If worker uses lock, recovery reacquires any locks the PREPARED transaction was holding when fail
  - 3. Restart the persistent sender to learn the current status of higher-layer transaction
- If a worker uses version history
  - Only step-3 is needed

Variation of 3N protocol

- Carry PREPARE/PREPARED in initial RPC req/rep
  - Lose the ability to ABORT when PREPARED
  - Must remain on the knife edge
  - It is preferred to delay the PREPARE/PREPARED pair
- 4N: with ACKs
  - Coordinator safely discard its outcome record after collecting all ACKs
- Presumed commit
  - Expect most transactions commit
  - Space-efficient representation of COMMITTED: non-existence
    - COMMIT by destroy the outcome record, but not when ABORT
  - Answers inquiry about a non-existent outcome record by sending a COMMITTED response

Unsolved Problem

- What if the coordinator crash after sending PREPARE but before sending COMMIT?
  - Workers are left in PREPARED with no way to proceed
- A problem that cannot be solved
  - Several workers will do their parts eventually, not necessarily simultaneously
  - Alice would in trouble if she had required simultaneous action

Review problem set 37: two-phase commit

- First phase: PREPARE / vote
  - First, the coordinator sends a PREPARE message to each of the workers.
  - For each worker, if it is able to commit, it writes a log record indicating it is entering the PREPARED state and send a YES vote to the coordinator; otherwise it votes NO.
- Second phase-1: COMMIT or ABORT
  - If all YES, coordinator logs a COMMIT and sends a COMMIT to all workers, which in turn log a COMMIT record. ABORT is similar.
- Second phase-2: ACKNOWLEDGMENT / END
  - After they receive the transaction outcome, workers send an ACKNOWLEDGMENT message to the coordinator.
  - Once the coordinator has received all ACK, it logs an END record.
- Crash recovery
  - Workers that have not learned the outcome of a transaction periodically contact the coordinator asking for the outcome.
  - If the coordinator does not receive an ACKNOWLEDGMENT from some worker, after a timer expiration it resends the outcome to that worker, persistently if necessary

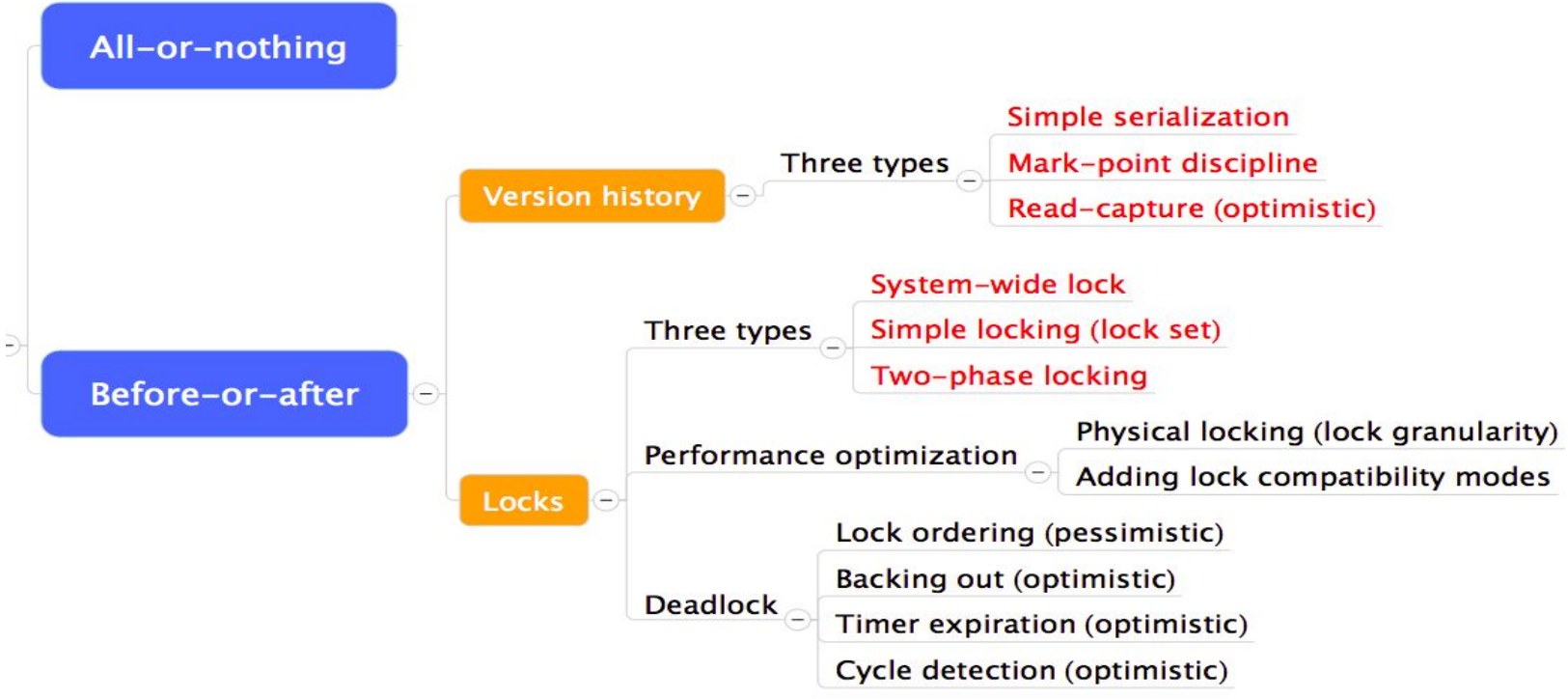
Q 37.8 For each of the following situations, of the above actions choose the best action that a worker or coordinator should take.

A. The coordinator crashes, finds log records for a transaction but no COMMIT record
B. The coordinator crashes, finds a COMMIT record for a transaction but no END record indicating the transaction is complete
C. A worker crashes, finds a PREPARE record for a transaction
D. A worker crashes, and finds log records for a transaction, but no PREPARE or COMMIT records

- W1. Abort the transaction by writing an "abort" record
- W2. Commit the transaction by writing a "commit" record
- W3. Resend its vote to the server and ask it for transaction outcome

- C1. Abort the transaction by writing an "abort" record
- C2. Do nothing
- C3. Send commit messages to the workers

Review so far



57

Case studies: machine language atomicity

- Complex instruction sets: the general electric 600 line
  - Indirect and tally: set by one-bit of indirect address
  - Load register A from Y indirect.
    - First Y++, then A = [Y.old]
    - Can be used to sweep through a table
  - Problem: page fault?
    - The indirect word and operand are in different pages
    - Then the state of the processor is "half-way of a instruction"
  - Solution: save internal state: 216-bit
    - Programmer get info not in the API
    - Be careful when restore

The Apollo desktop computer and Motorola M68k

- Problem: ISA is not atomic when adding VM, and multiple processors
- Solution: install two 68k
  - When the first encounters a page fault, stops
  - The second fetches the missing page, restarts the first one
- Solution-2:
  - Modify compilers to generate atomic instruction only
  - More tricky but risky
- Solution-3:
  - Save internal state