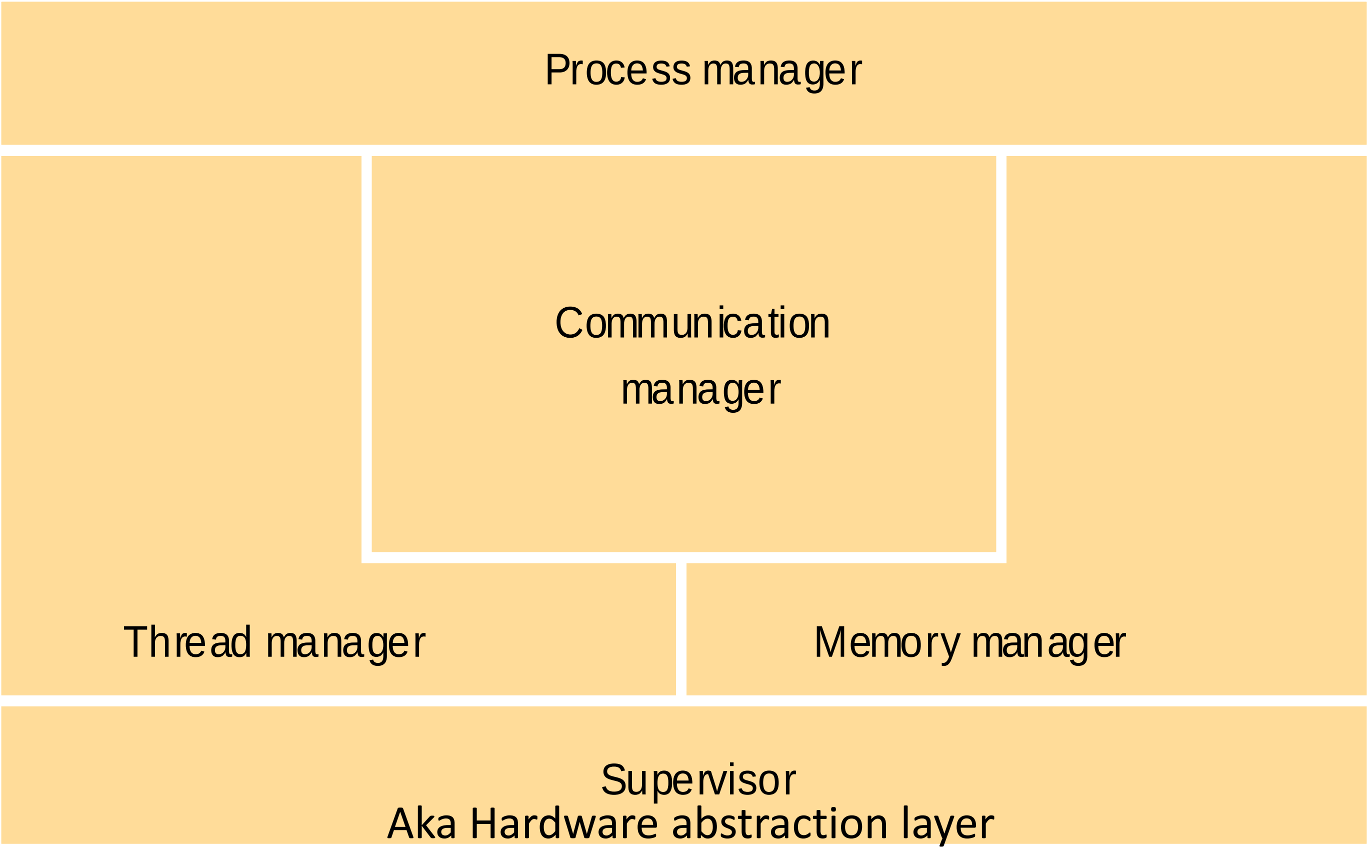
Distributed Systems
OS and Networking Fundamentals

Lecture 02

Michele Albano

Credits to
Bryant & O'Hallaron
Kurose & Ross
Brian Nielsen

Core OS functionality



- Processes & threads
  - Synchronization
  - Scheduling
  - Deadlock
  - Interprocess communication
- Memory management, address translation, and virtual memory
- Operating system management of I/O
  - Device Drivers
  - Network protocol stack
- File systems
- Security & protection
- Nice user interface(s)

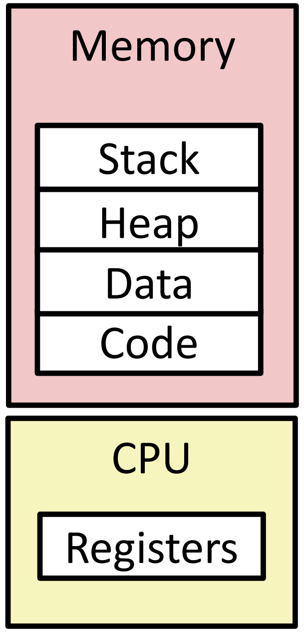
OS Kernel = The critical part of the OS that has complete control over the resources
- Executes in processor's "priviledged mode"

Processes

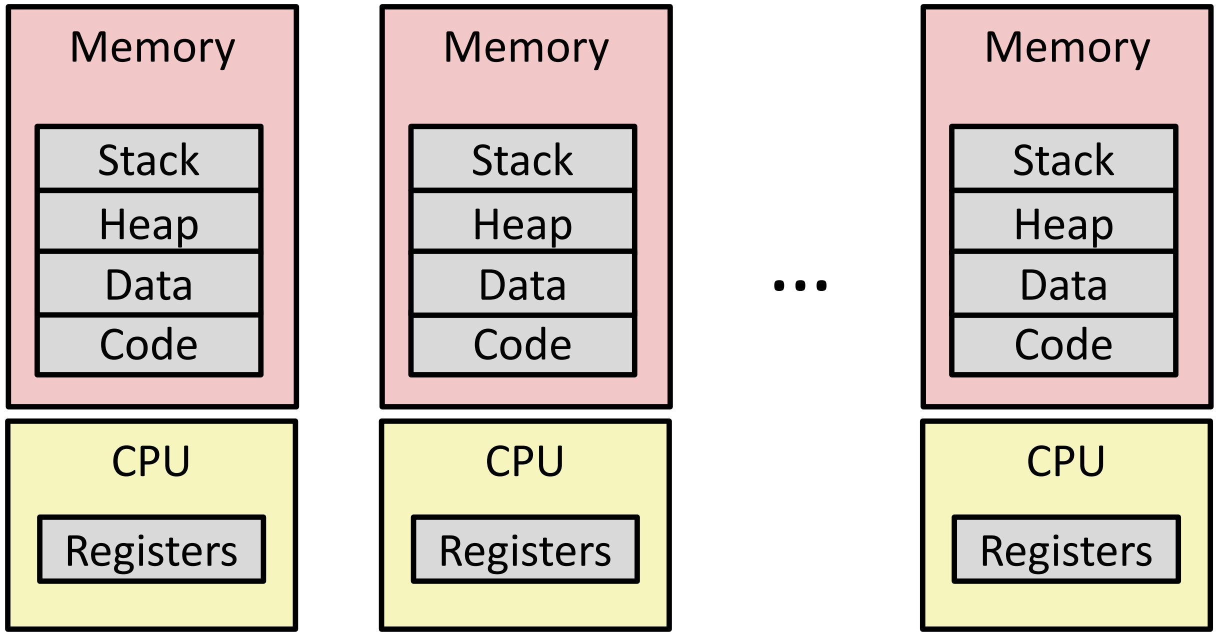
Processes

- Definition: A *process* is a running program (an instance of a program).
  - One of the most profound ideas in computer science
  - Not the same as “program” or “processor”

- Process provides each program with two key abstractions:
  - ***Logical control flow***
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called <u>*context switching*</u>
  - ***Private address space***
    - Each program seems to have exclusive use of main memory.
    - Provided by a technique called <u>*virtual memory*</u>



The Multiprocessing Illusion

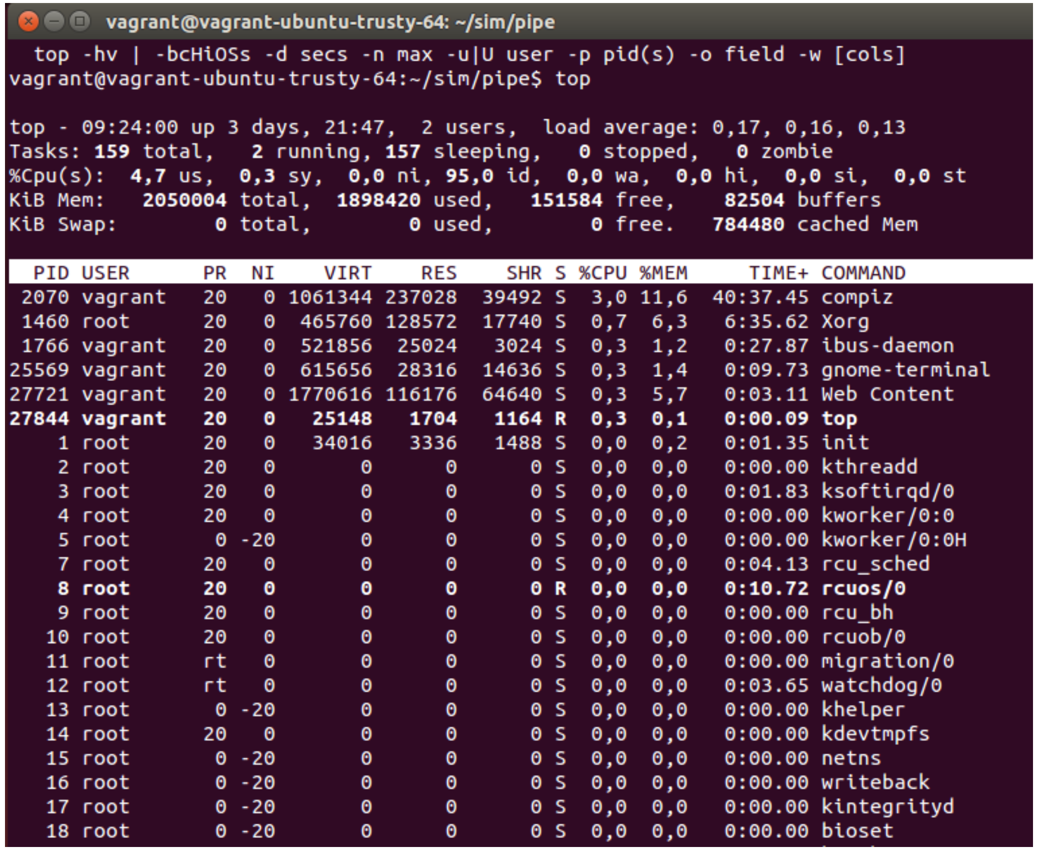


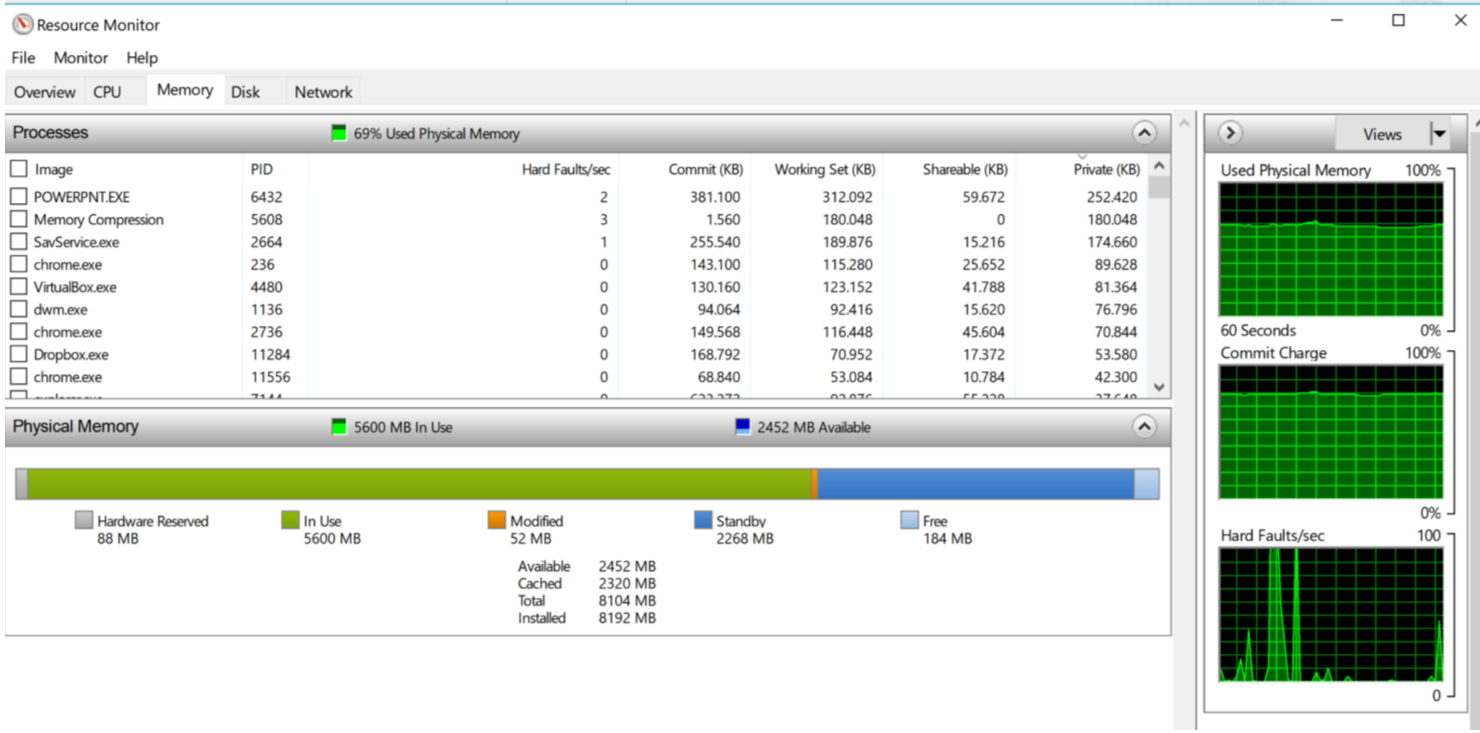
- Computer runs many processes simultaneously
  - Applications for one or more users
    - Web browsers, email clients, editors, …
  - Background tasks
    - Monitoring network & I/O devices

Running Processes

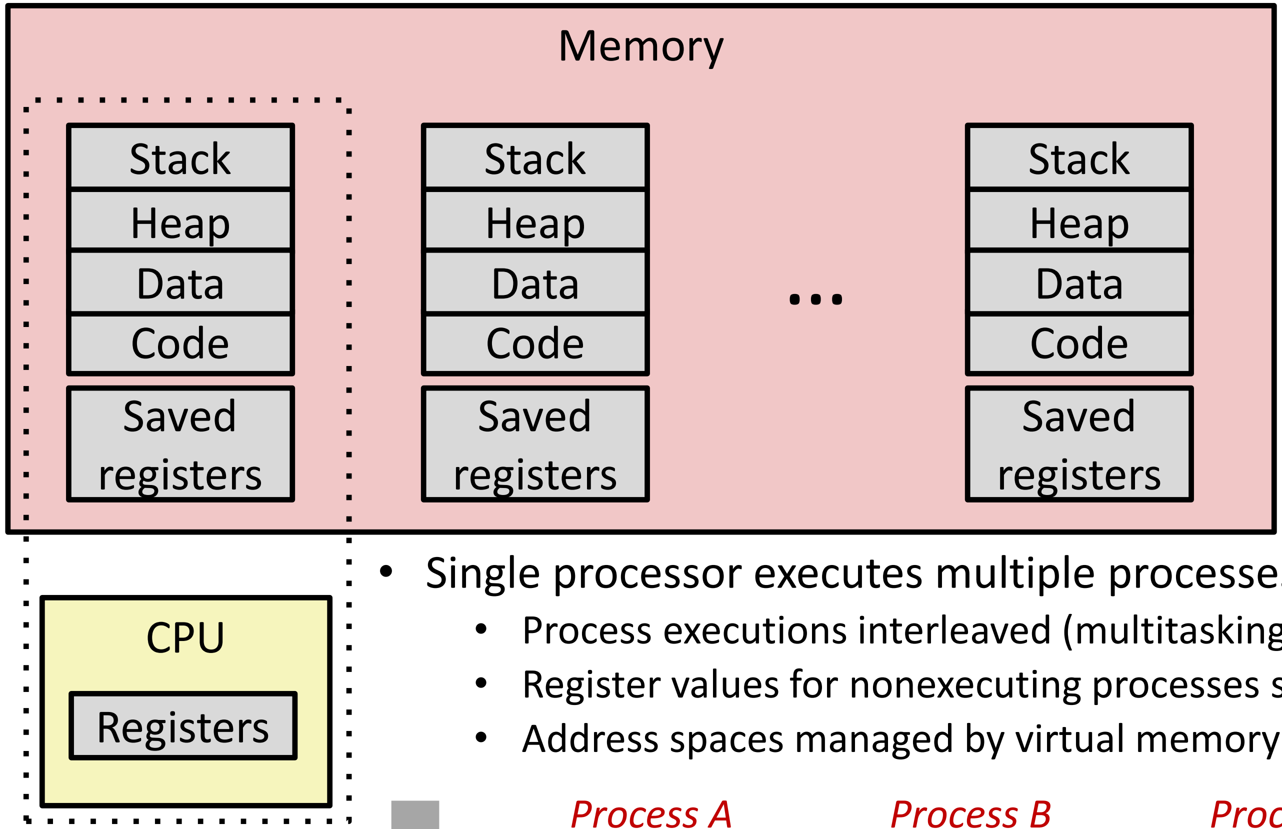
Process = A running program = A private "address space"+ one or more "execution threads"

- Ordinary user program
- Systems programs and network services (Daemon)
- ~100 such "programs" active "simultaneously"

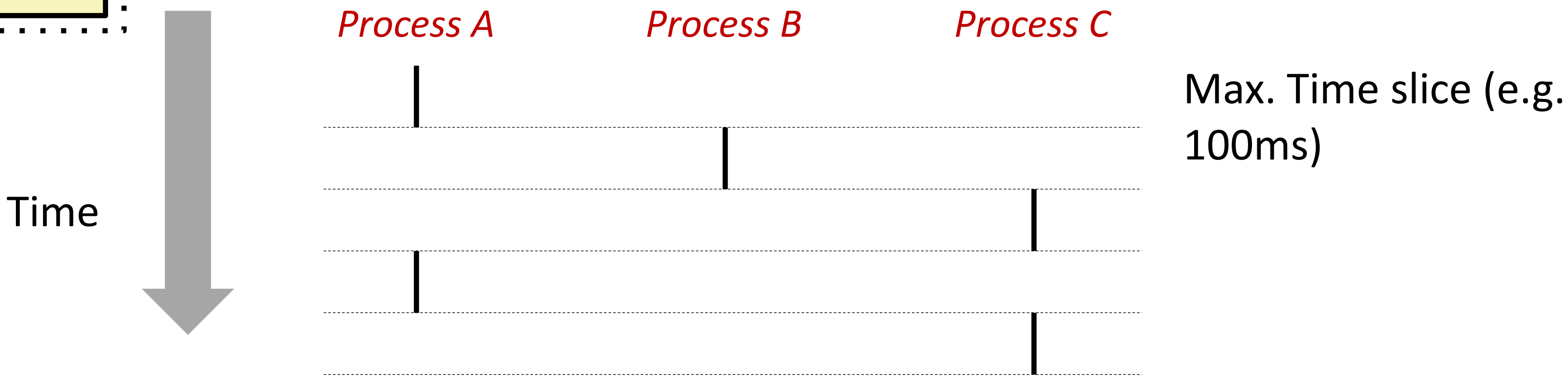




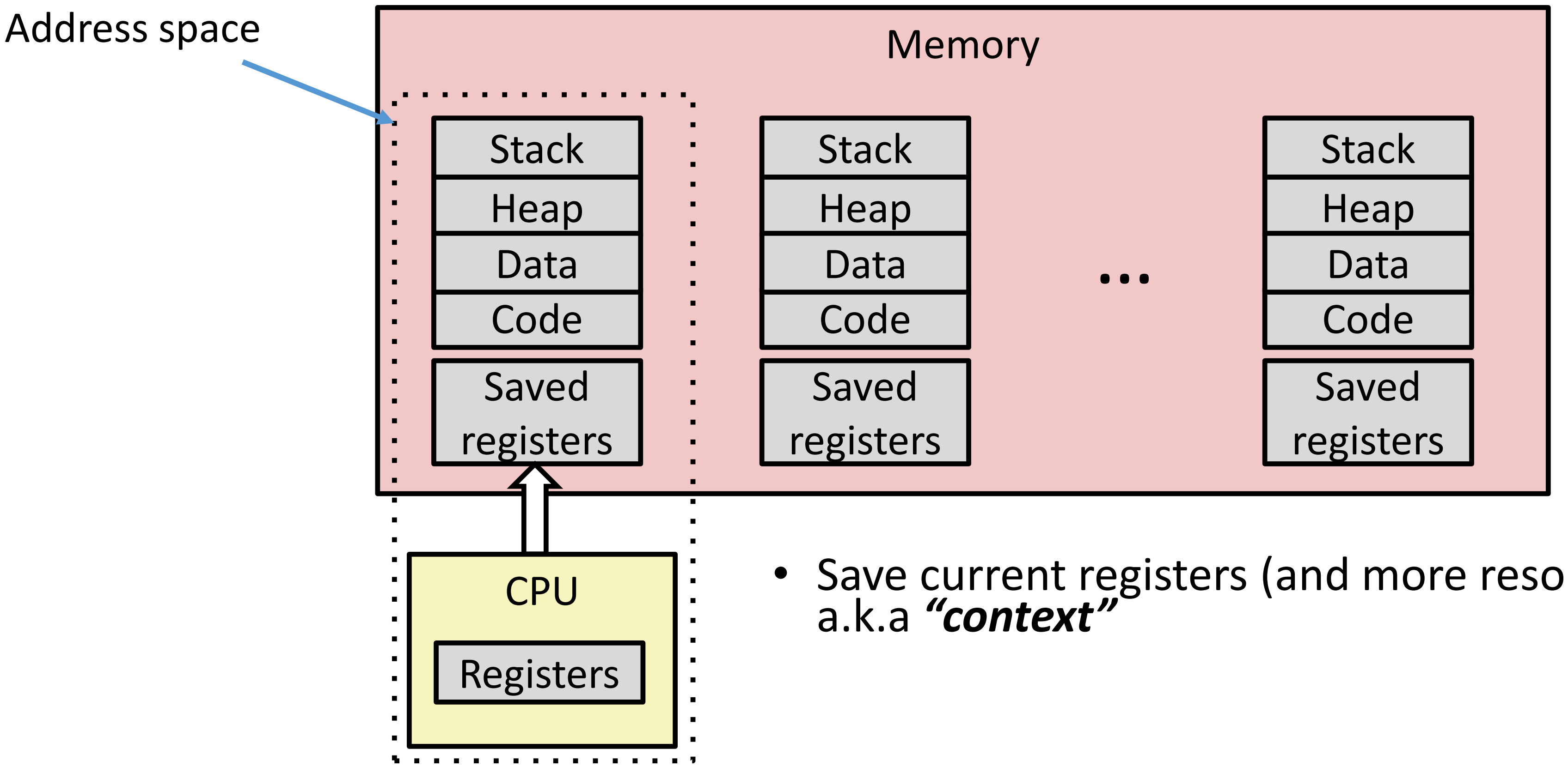
Multiprocessing: The (Traditional) Reality



- Single processor executes multiple processes concurrently
  - Process executions interleaved (multitasking)
  - Register values for nonexecuting processes saved in memory
  - Address spaces managed by virtual memory system (later in course)

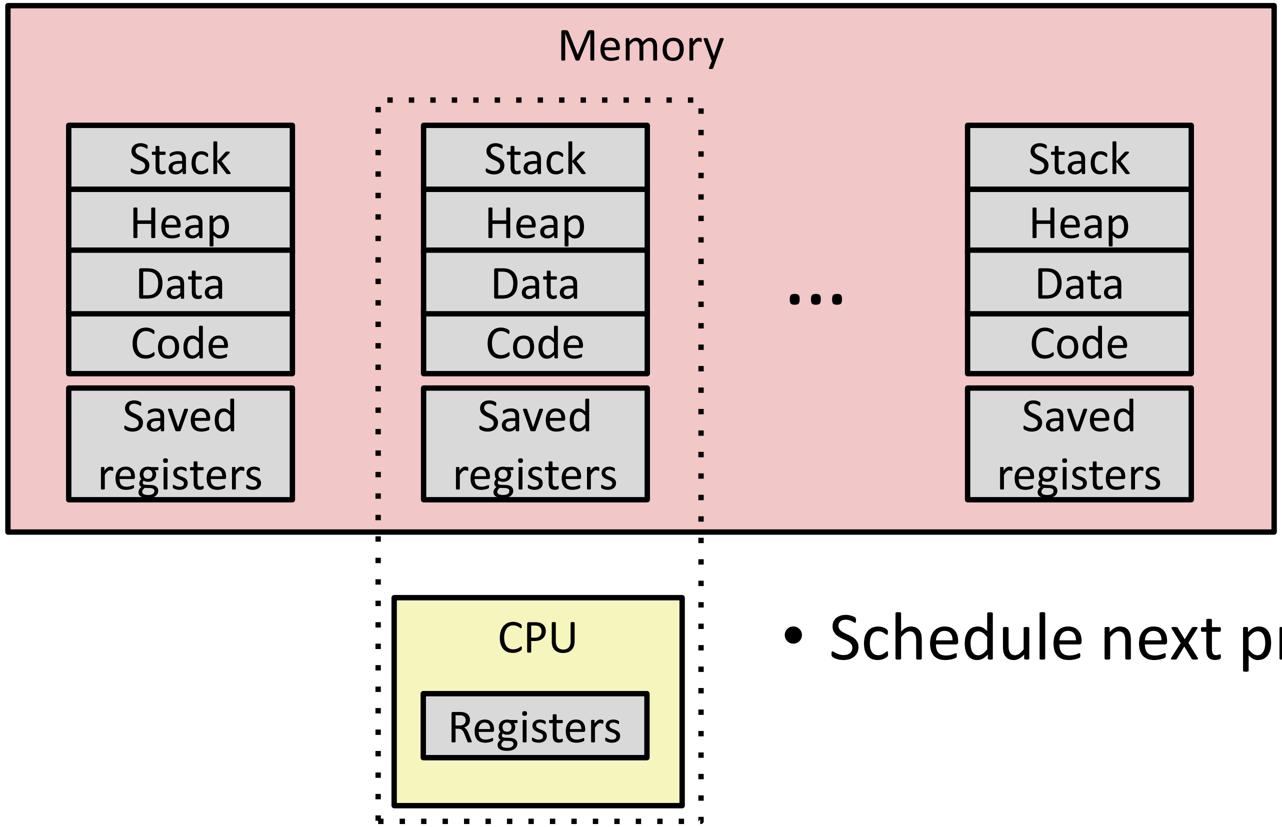


Multiprocessing: The (Traditional) Reality



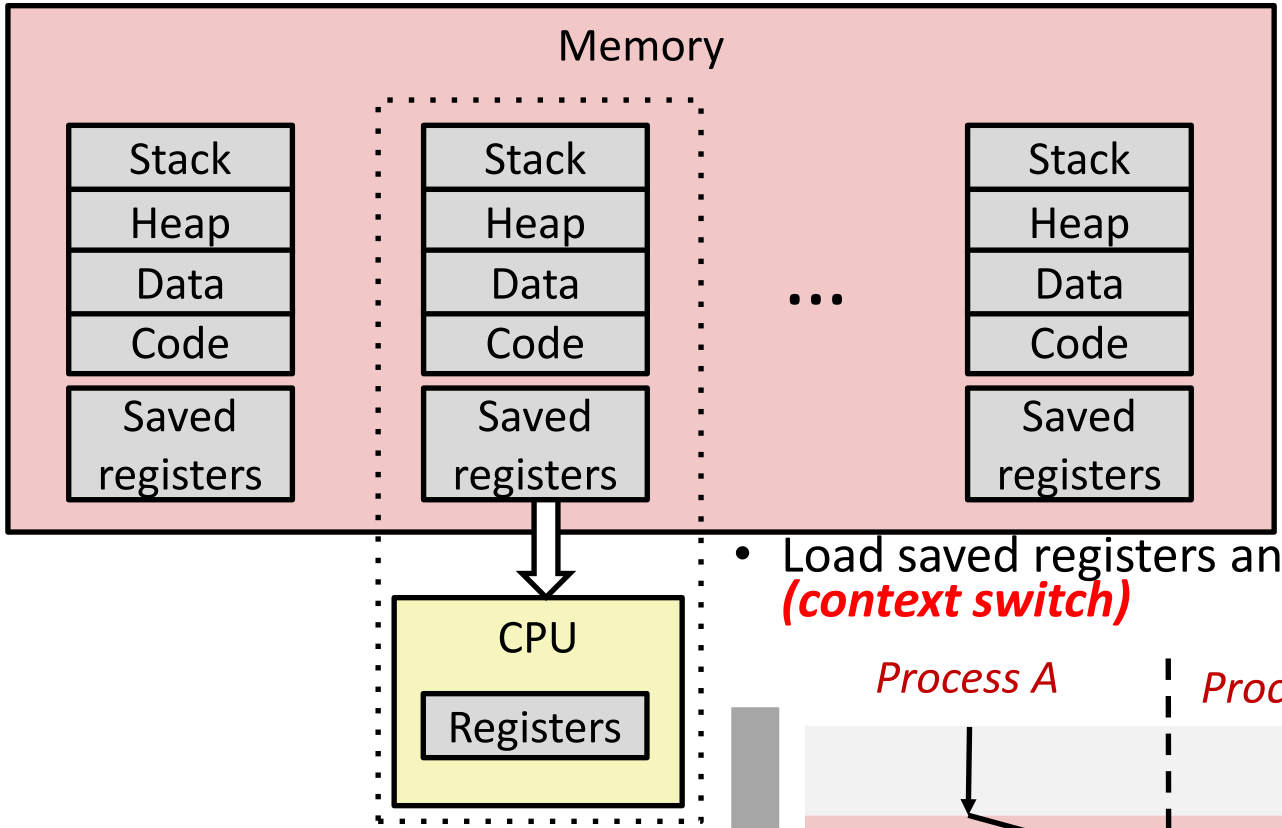
- Save current registers (and more resource information) in memory a.k.a ***"context"***

Multiprocessing: The (Traditional) Reality

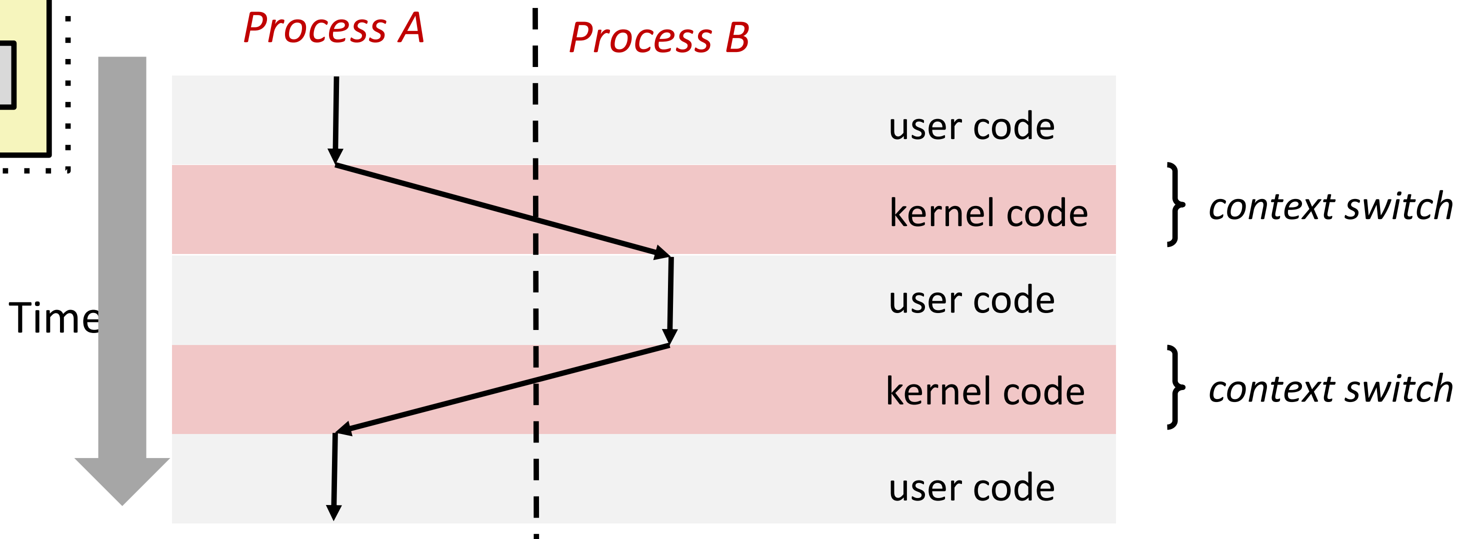


- Schedule next process for execution

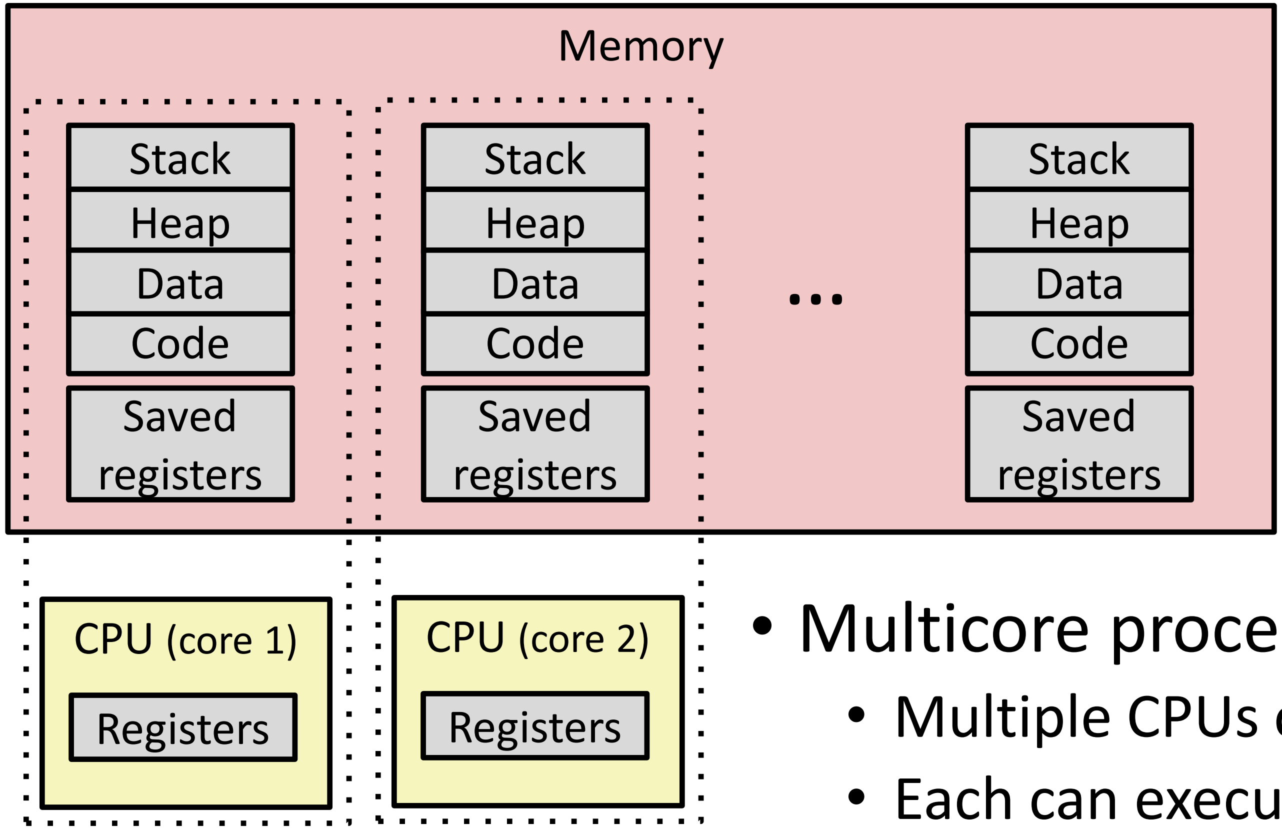
Multiprocessing: The (Traditional) Reality



- Load saved registers and switch address space ***(context switch)***

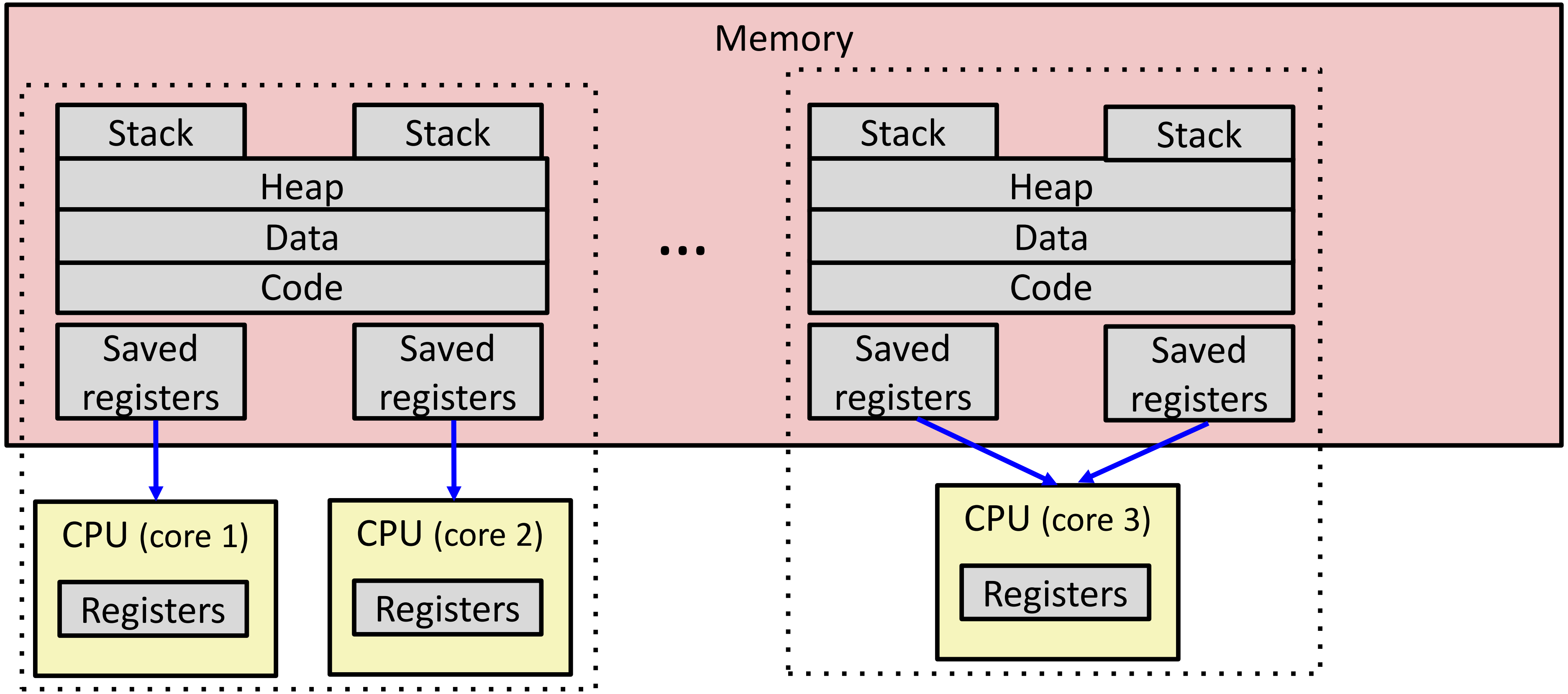


Multiprocessing: The (Modern) Reality 1



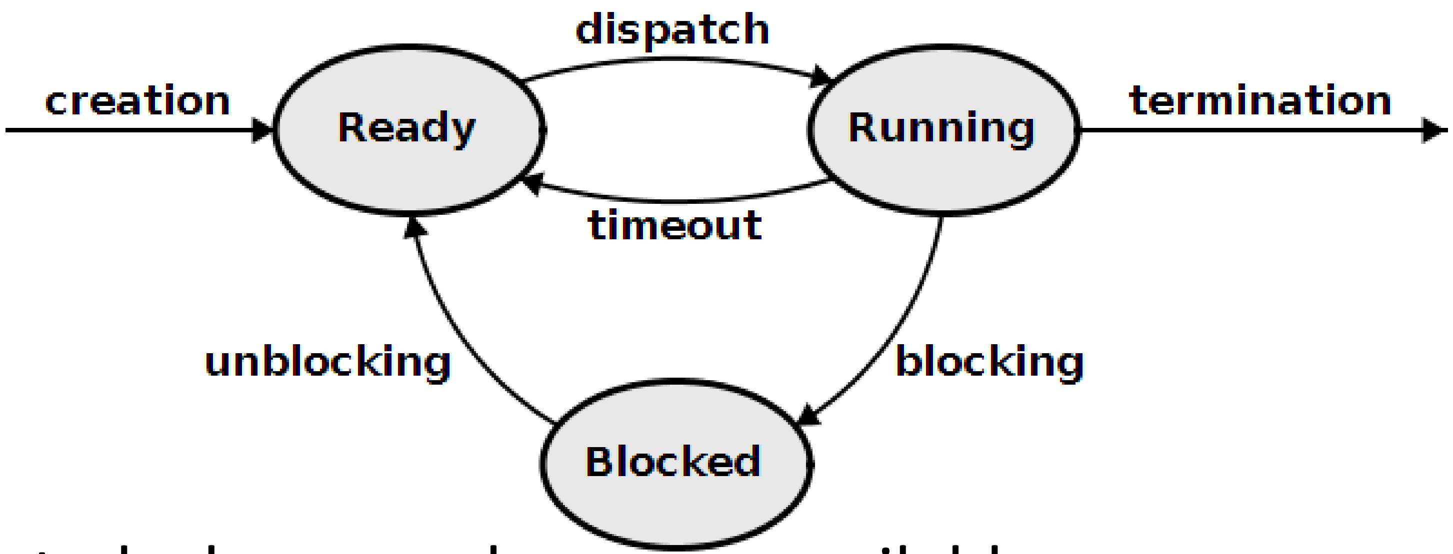
- Multicore processors
    - Multiple CPUs on single chip
    - Each can execute a separate process
- True “parallelism”

Multi-threading : The (Modern) Reality 2



- Process contains several “execution threads”
- Shares data+code
- Has private stack+context

Thread state model



- **Ready:** can be executed when core becomes available
- **Running:** currently executing on a core
- **Blocked:** waiting for resource
  - I/O device
  - Synchronization (e.g. lock/semaphore/monitor)

Exceptions and I/O

Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU's *control flow* (or *flow of control*)

Physical control flow

Time

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

Altering the Control Flow

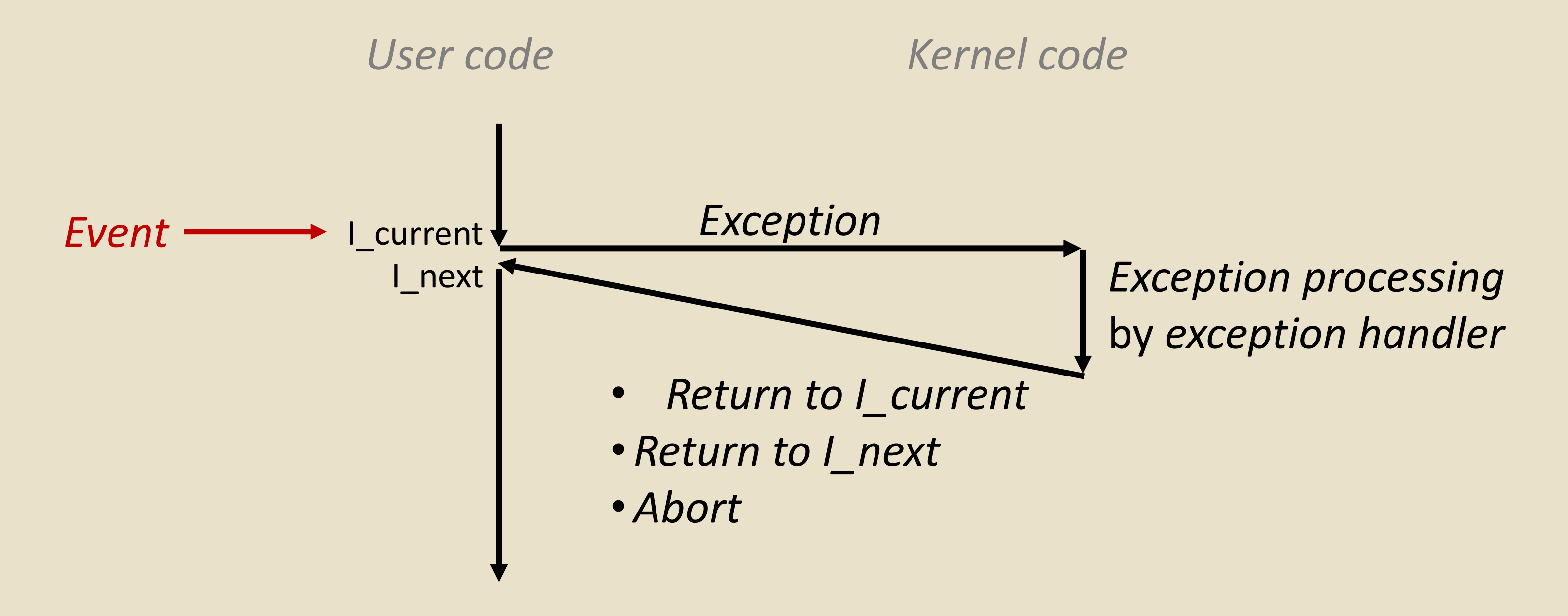
- Up to now: two mechanisms for changing control flow:
  - Jumps and branches (if, for(), switch(),…)
  - Call and return

  React to changes in ***program state***

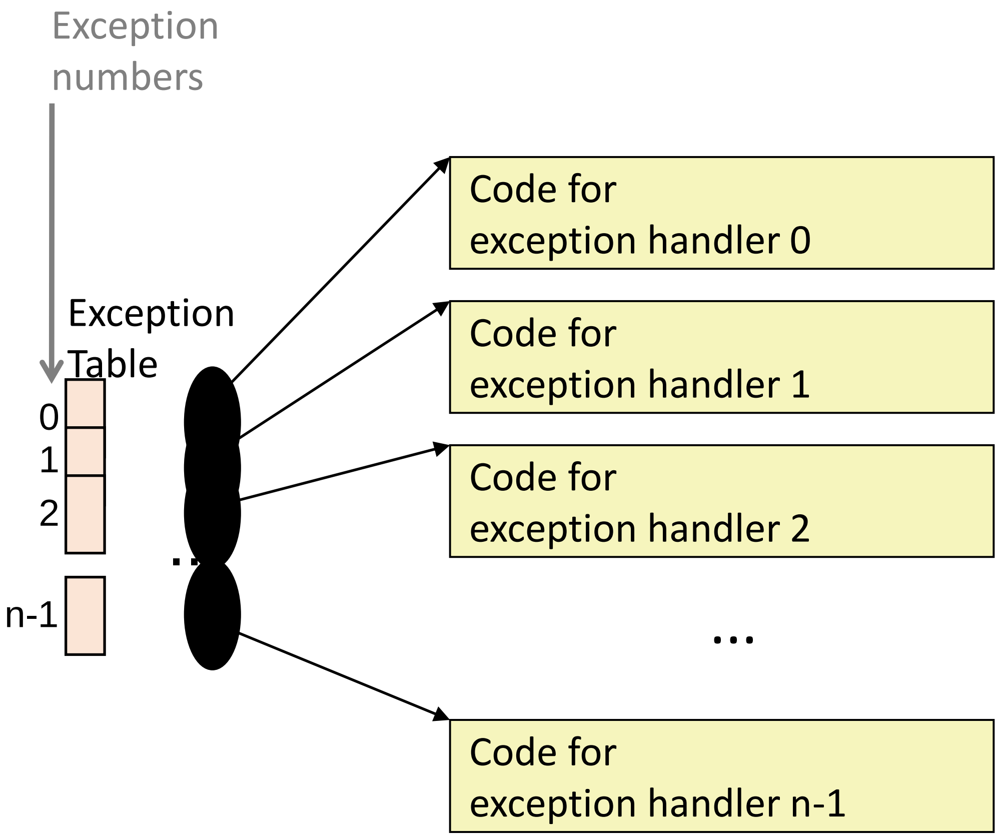
- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero, reads inaccessible memory
  - User hits Ctrl-C at the keyboard
  - System timer expires

Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Exception Tables



- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's *interrupt pin*
  - Handler returns to "next" instruction

- Examples:
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - I/O interrupt from external device
    - Hitting Ctrl-C at the keyboard
    - Arrival of a packet from a network

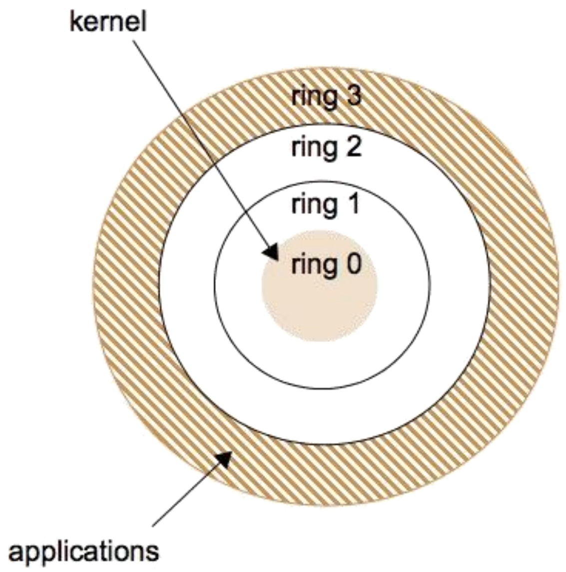
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check

Processor Protection levels

- Most processors have several protection levels
  - User-mode and Kernel mode
  - Privileged instruction
    - Can only be executed in kernel mode
    - Typically instructions that may change hardware resources in a "bad" way for other processes or OS
    - E.g., Disable interrupts, or modify page table base register
    - Causes processor to generate an exception and pass control to kernel-code

  - Non-Privileged instruction
    - All other instructions
    - Normal arithmetic operation
    - Move memory word to/from Register, Push/Pop



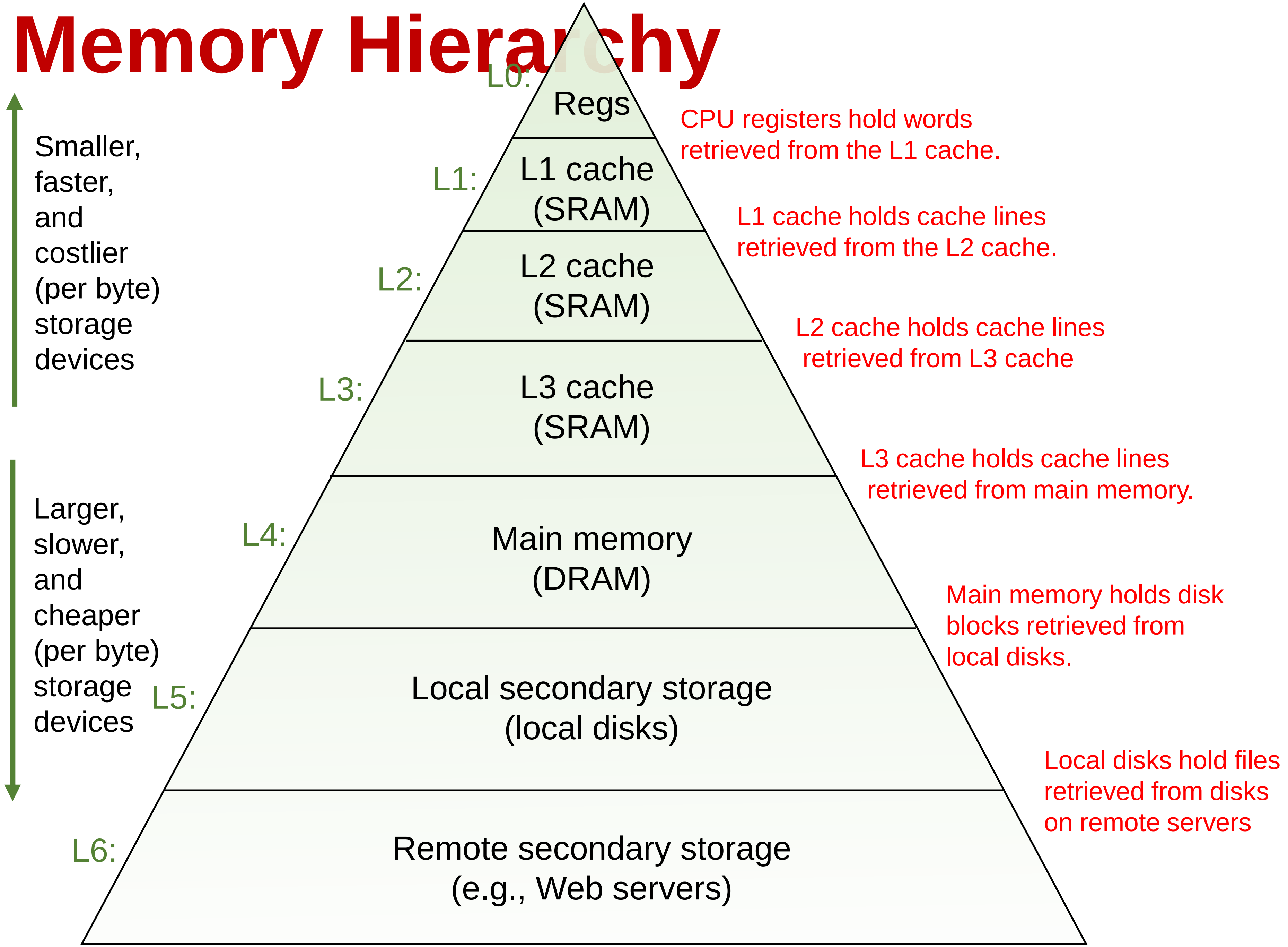
System Calls

- **Each system call has a unique ID number**
- **All resource requests must be validated and granted by OS**
- **Examples:**

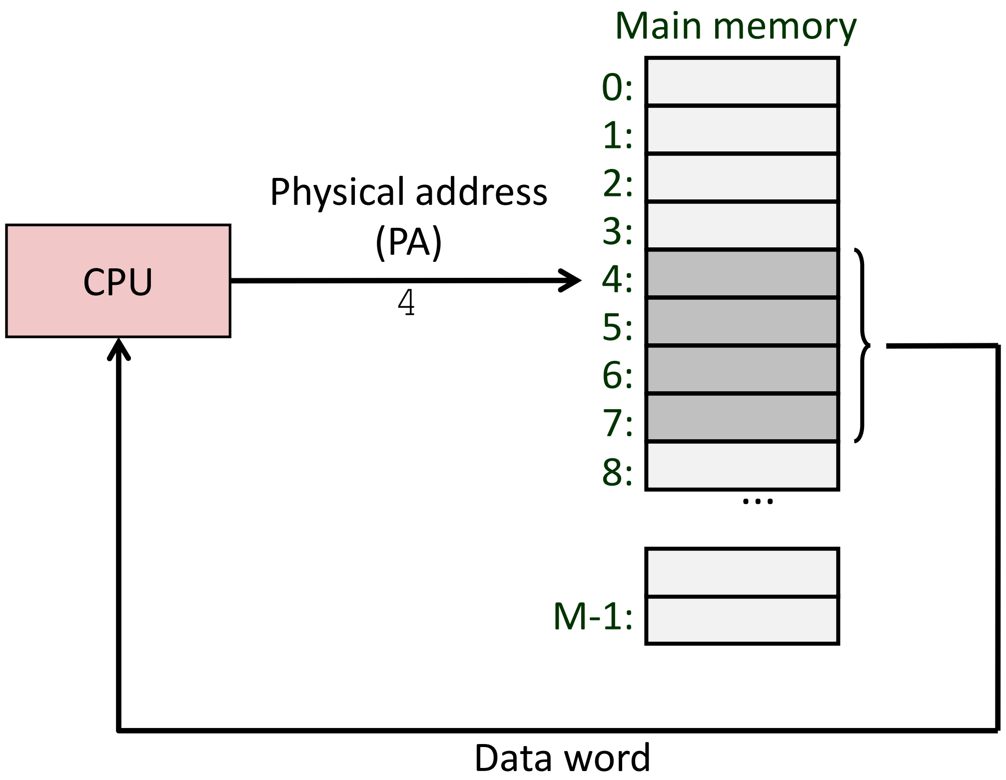
Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Virtual Memory

The Memory Hierarchy

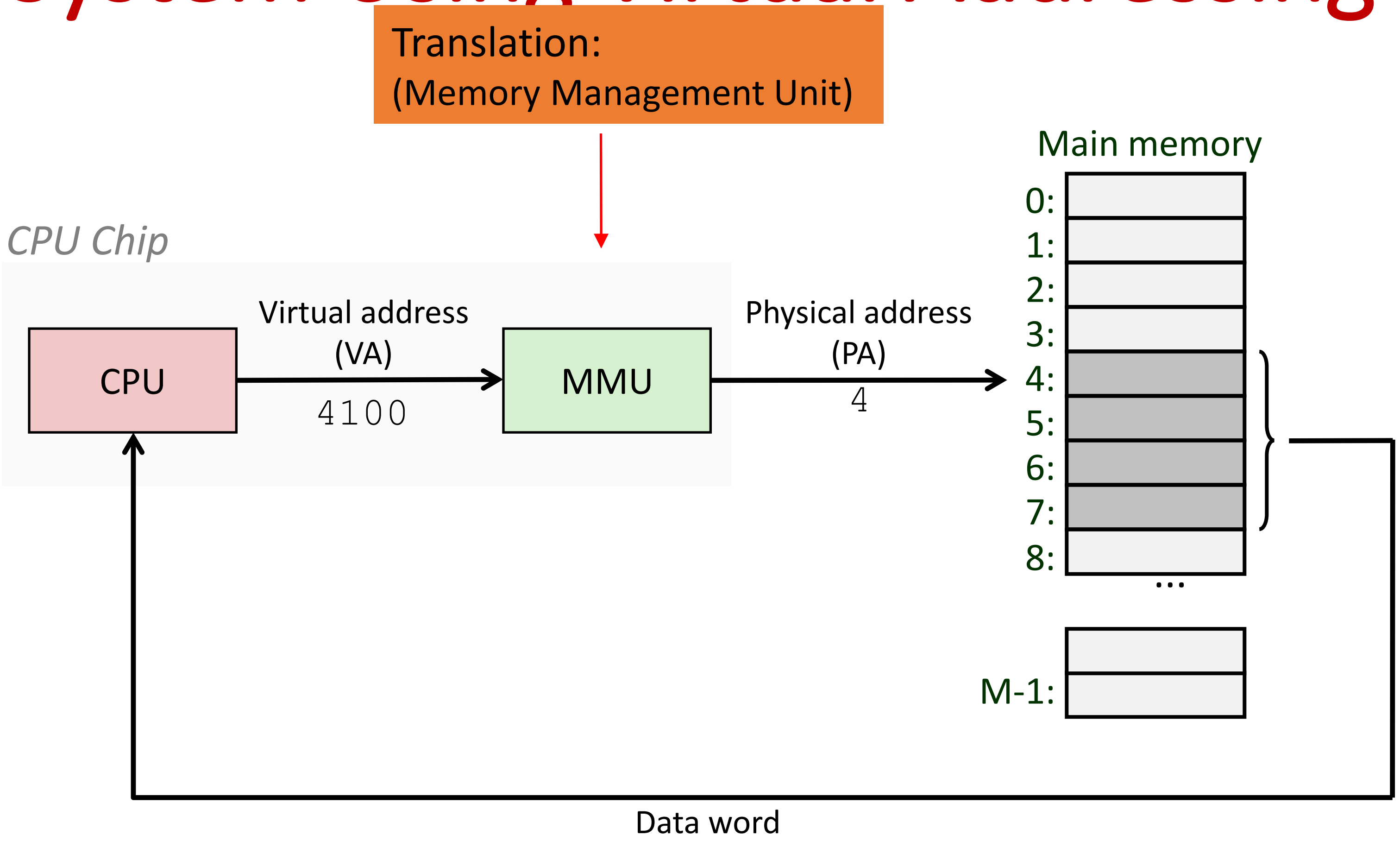


A System Using Physical Addressing



- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing



- Used in all modern PC's, servers, laptops, and smart phones
- One of the great ideas in computer science

Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space
  - Execute more programs than physical memory permits
  - Provide larger address spaces than physical memory permits

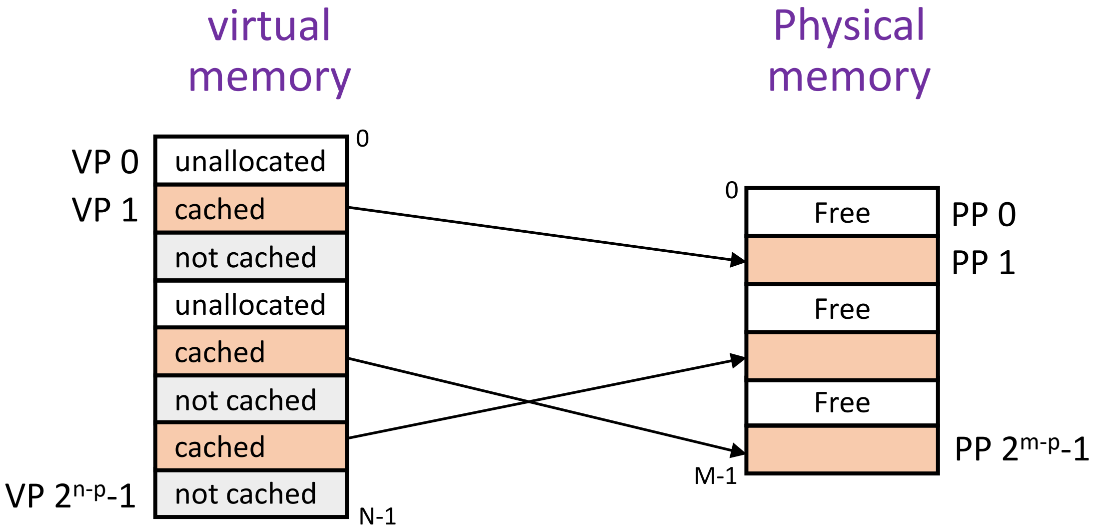
- Simplifies memory management
  - Simplifies keeping track of free/used memory
  - Reduces waste
  - Each process gets the same uniform linear address space

- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

VM for caching

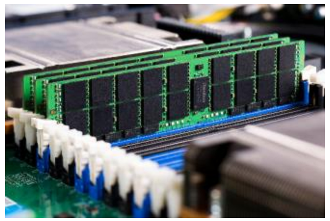
- *Conceptually Virtual memory* is an array of N consecutive bytes stored on disk.
- Its contents are "cached" in *physical memory* (*DRAM cache*)
  - In VM, cached blocks are called *pages* (of $P = 2^p$ bytes)

Eq. page size : $=2^{12}$ bytes=4KB



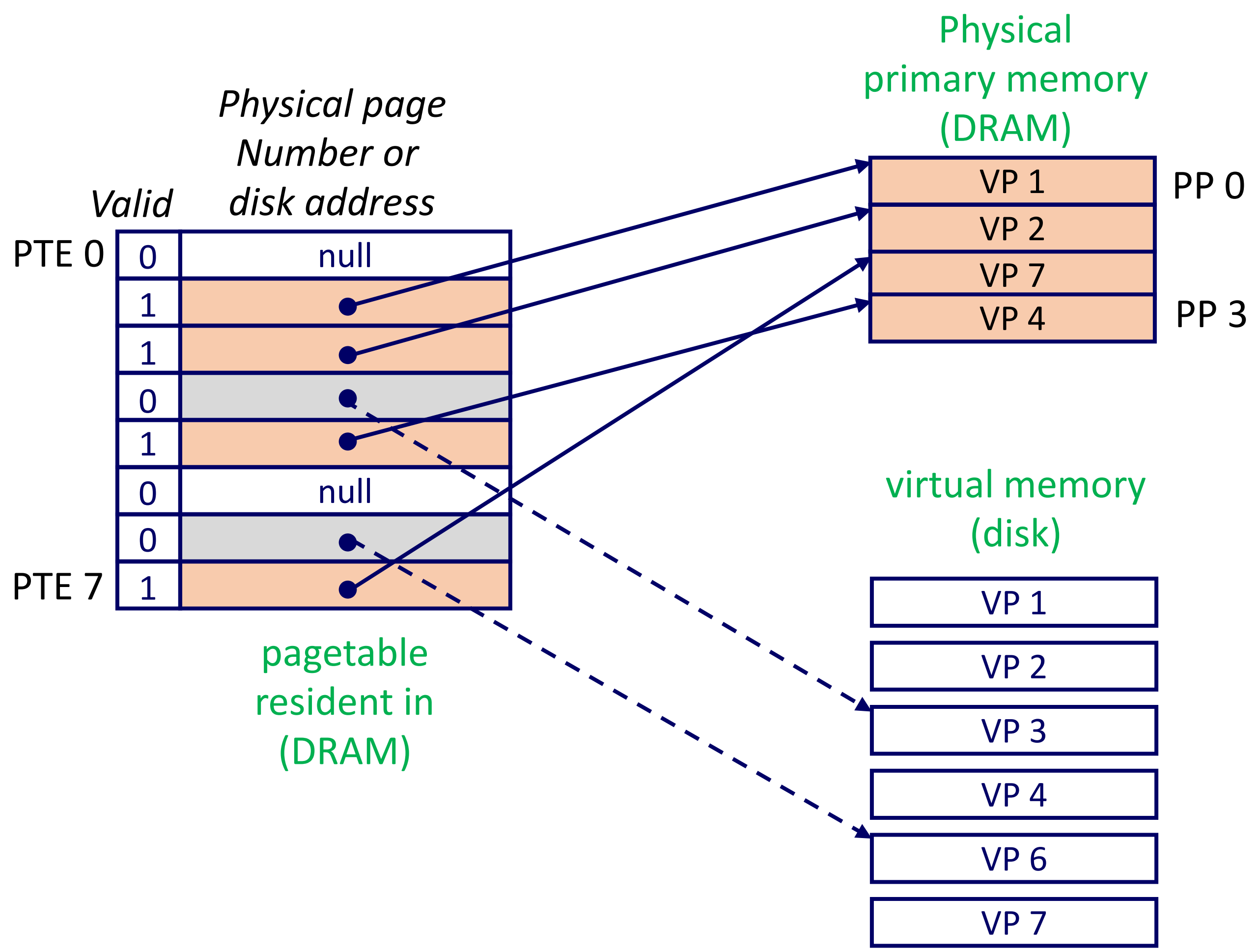
virtual pages (VPs) stored on secondary memory (disk)

Physical Pages(PPs) cached in primary memory (DRAM)



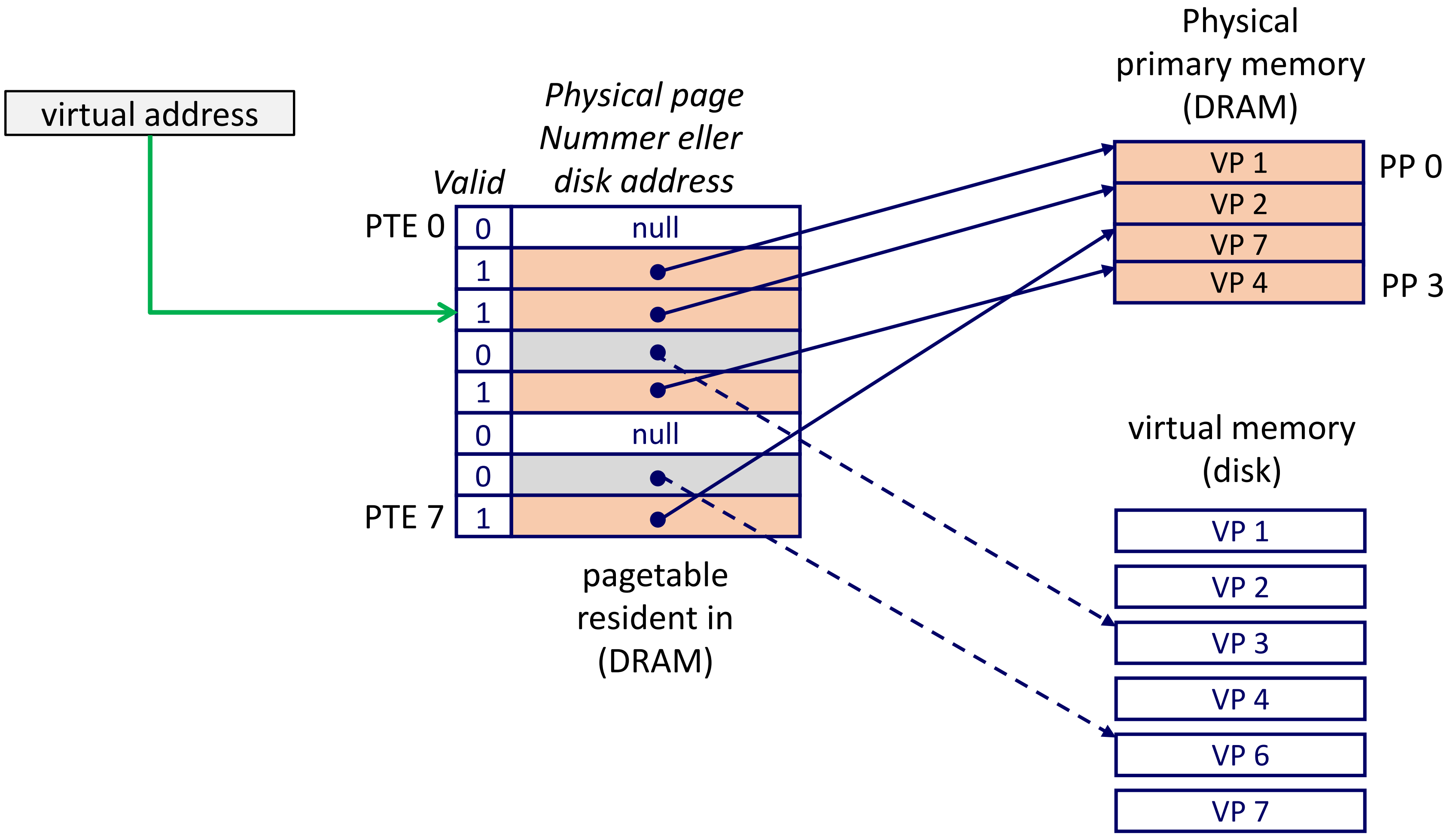
Data Structure for VM (Page Table)

- A *page table* is an array of PTEs (page table entries) that maps virtual pages to physical pages
  - Every process has its own, stored in kernel memory



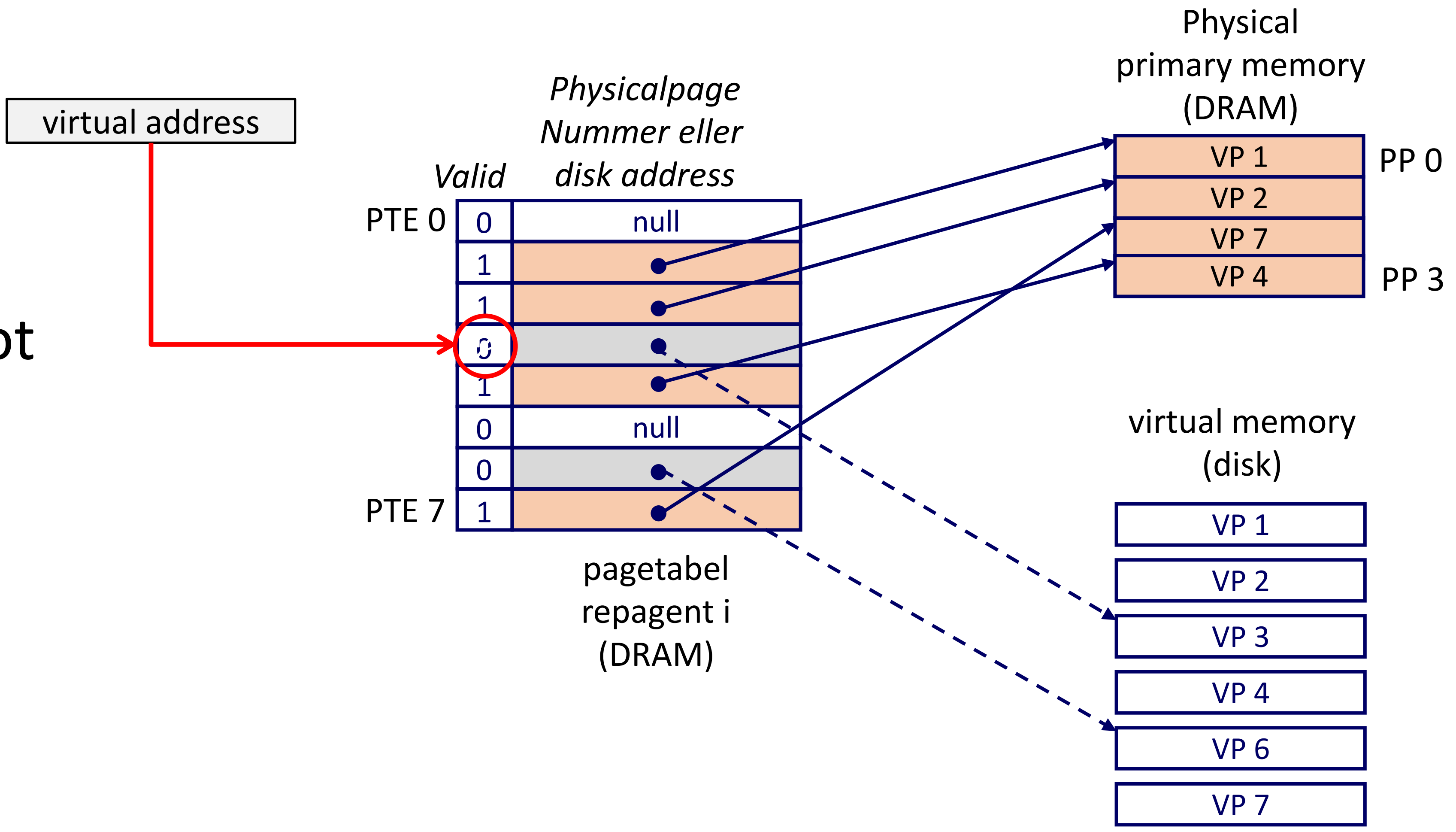
Page Hit

- *Page Hit:* issues a reference a word in VM that is present in physical memory (DRAM cache hit)



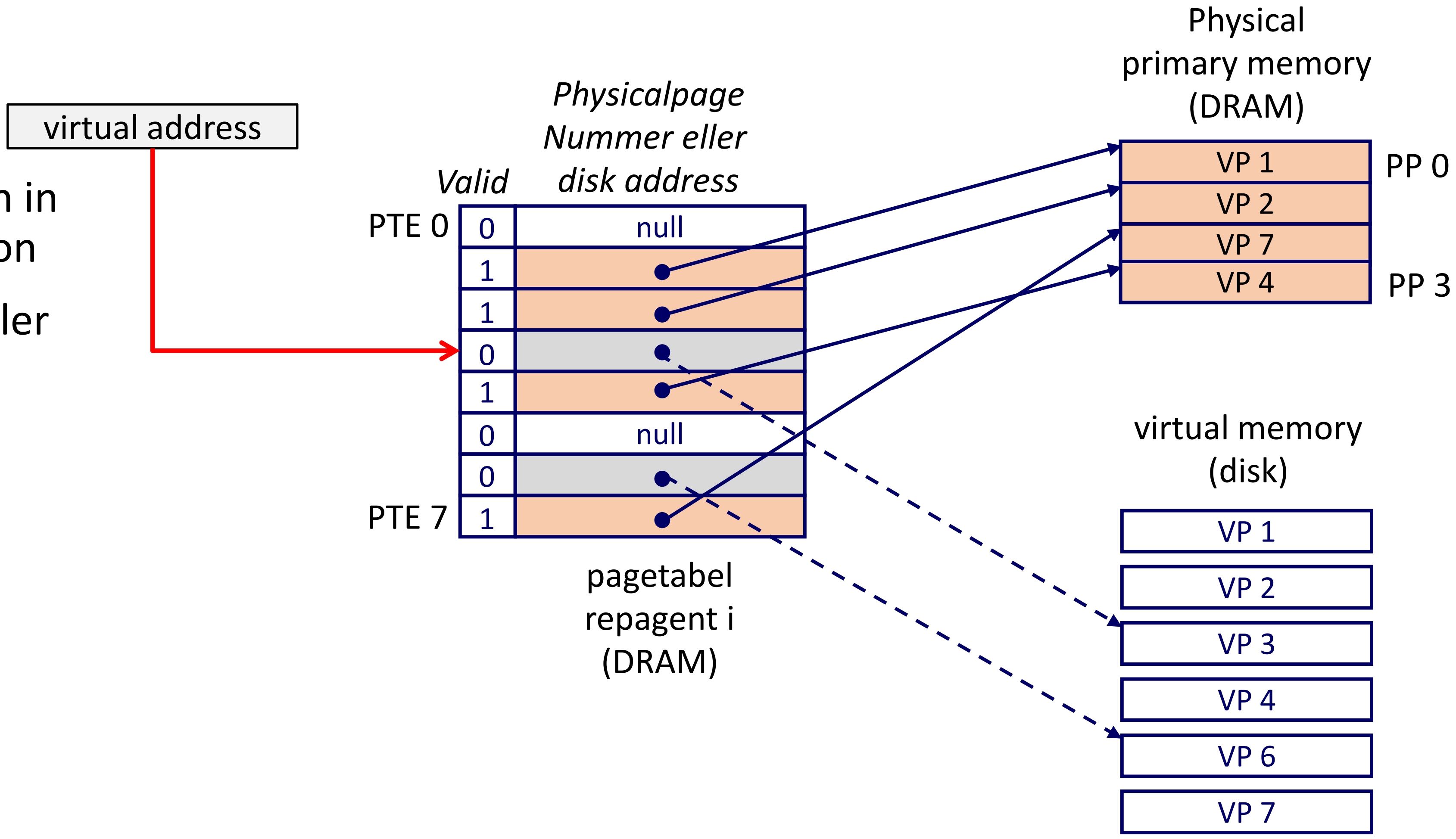
Page Fault

- *Page Fault: Processor issues* reference to a word in VM that is not present in physical memory
  (DRAM cache miss)



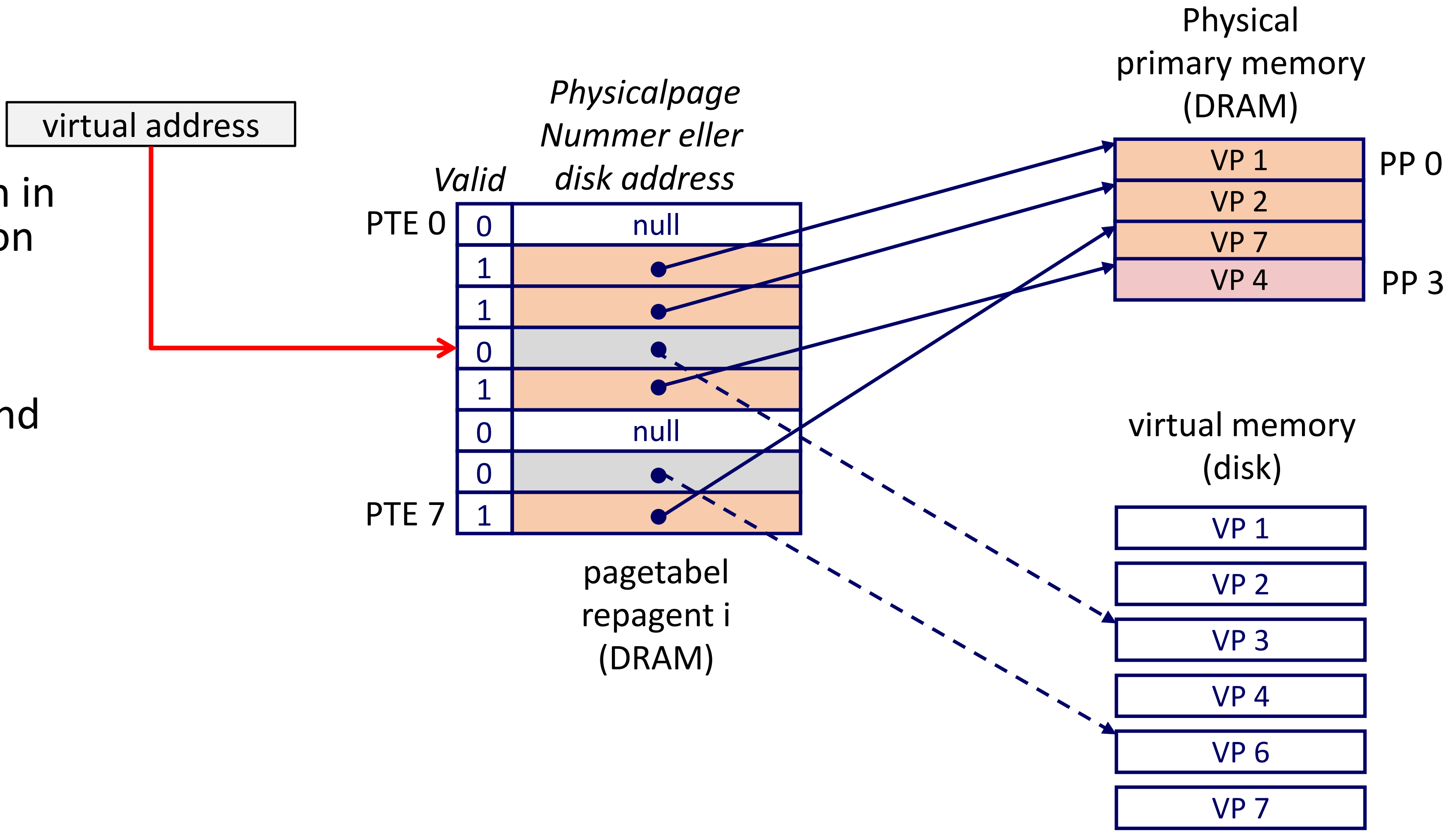
Handling Page Fault

- Page-fault causes an exception in processor : Page-fault exception
- Wakes up OS page-hault handler



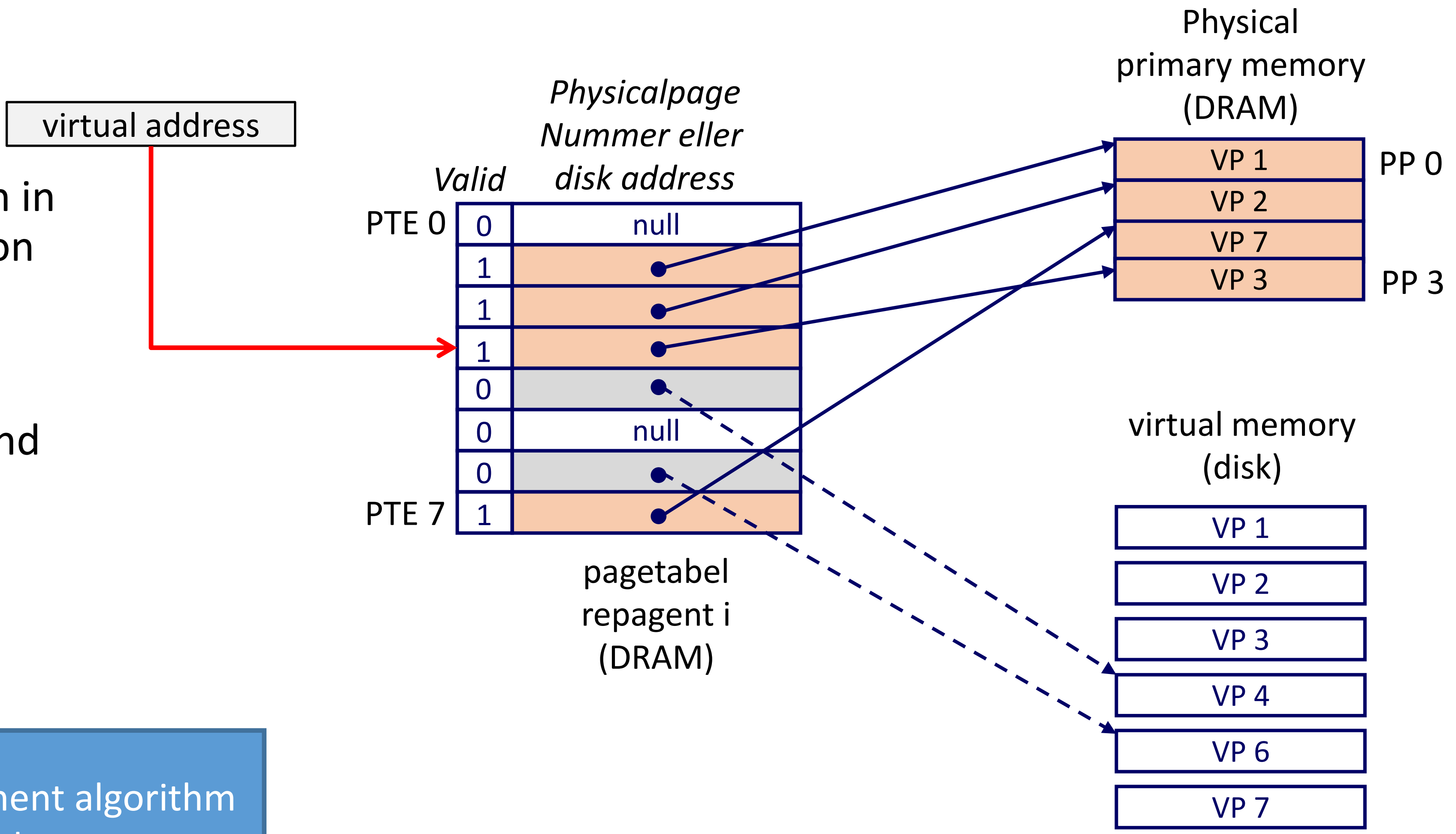
Handling Page Fault

- Page-fault causes an exception in processor : Page-fault exception
- Wakes up OS
- OS' page-hault handler finds a suitable "victim" to sacrifise and evict from cache (here VP 4)



Handling Page Fault

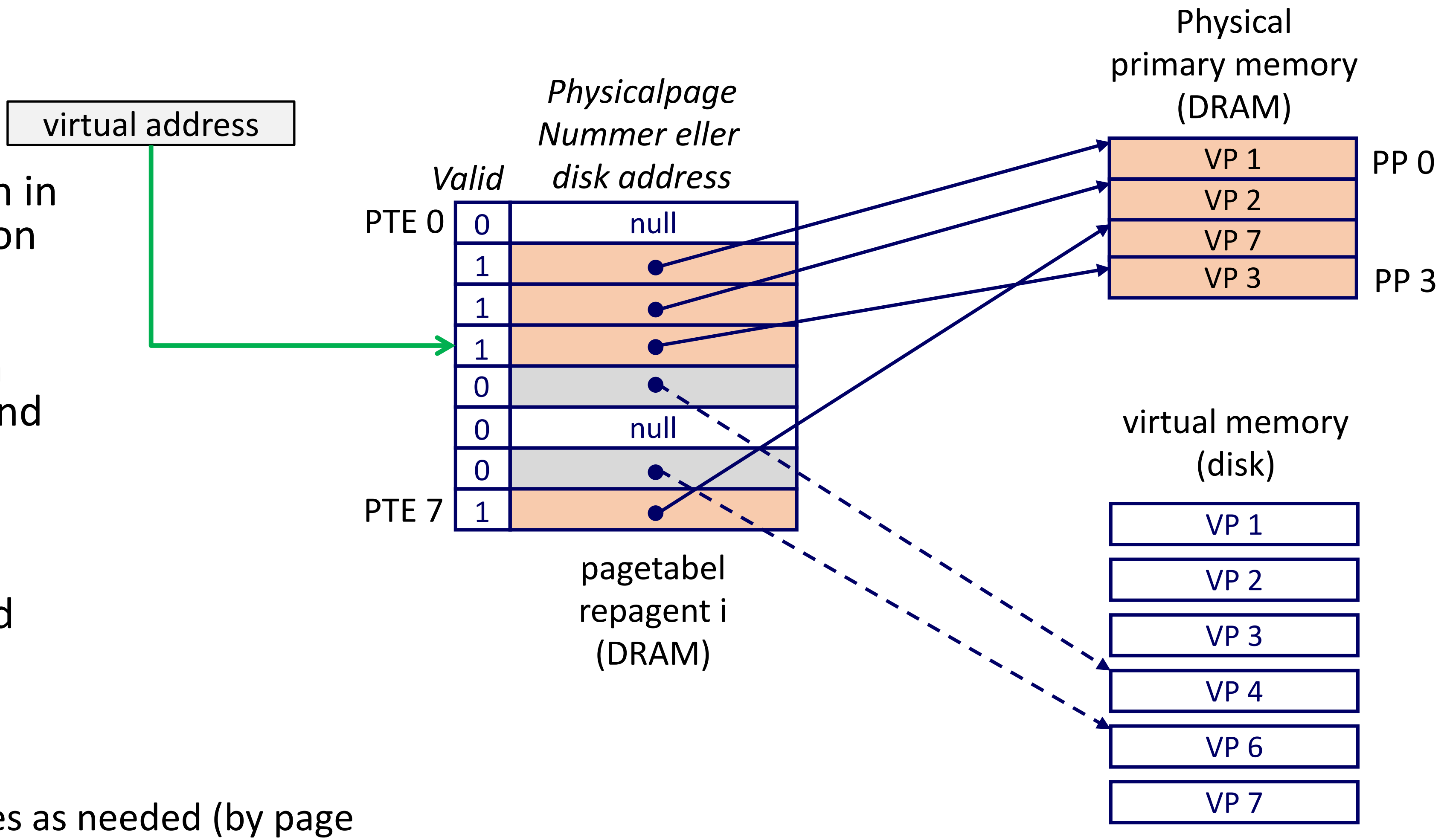
- Page-fault causes an exception in processor : Page-fault exception
- Wakes up OS
- OS' page-hault handler finds a suitable "victim" to sacrifise and evict from cache (here VP 4)
  - Saves VP4 to disk if modified
  - Loads VP3 from disk



Page replacement algorithm typically implements an approximation to Least Recently Used (LRU)

Handling Page Fault

- Page-fault causes an exception in processor : Page-fault exception
- Wakes up OS
- OS’ page-hault handler finds a suitable ”victim” to sacrifise and evict from cache (here VP 4)
  - Saves VP4 to disk if modified
  - Loads VP3 from disk
- Restart instruction that caused fault



OBS: The system reads only pages as needed (by page faults): *demand paging*

Disclaimer

- We have not looked at the actual detailed mechanisms
- Many more hard/software tricks are neede to make this work effectively
  - MMU
  - Translation look-aside buffer
  - Page tables in multiple levels
  - More details and control bits in page table
  - Good interaction with the L1-L3 caching system

Locality saves your performance again!

- VM seems to be hoplessly inefficient, but essentially works because of ”localitet”
  - Well-behaving programs uses the
    - same data repeatedly (temporal locality), or
    - data nearby (spatial locality)
- At any point in time, programs tends to only use a subset of its virtual pages. Active pages = *working set*
  - Programs with good locality has smaller working sets

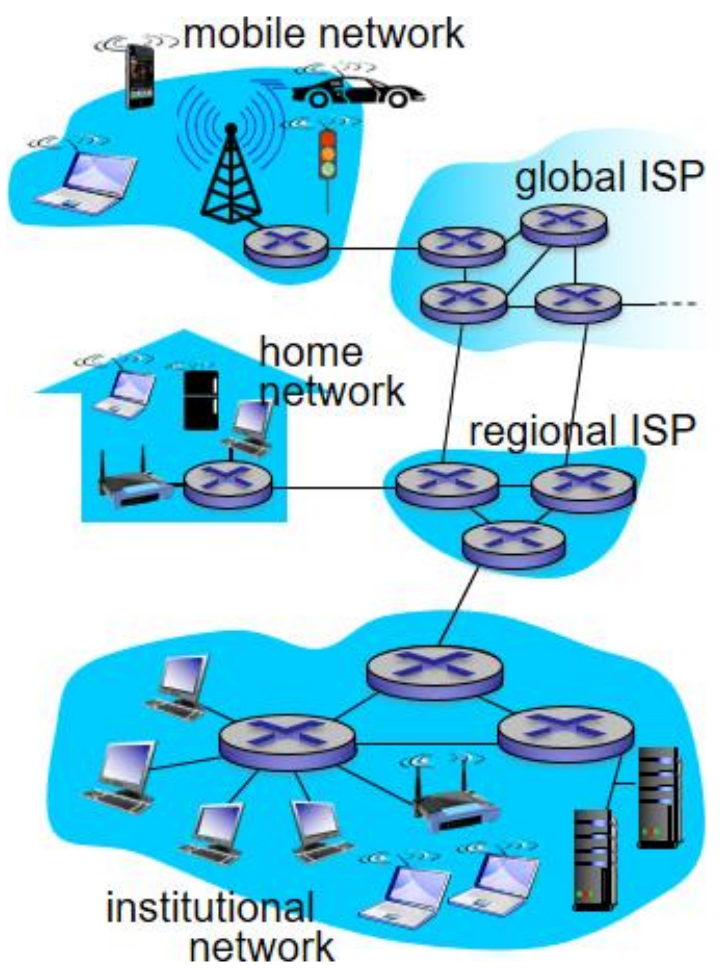
- IF (working set < primary memory)
  - Good performance for a process (after a initial cold-start period)
- IF (SUM(working set) > primary memory)
  - *Meltdown (Thrashing): Hopeless performance where the OS keeps performing page-replacements – access time dominated by disk = slooooooooowwwwww*

Internetworking

Internet structure and Packet-switching

A simple view of (Inter-)Network Structure:

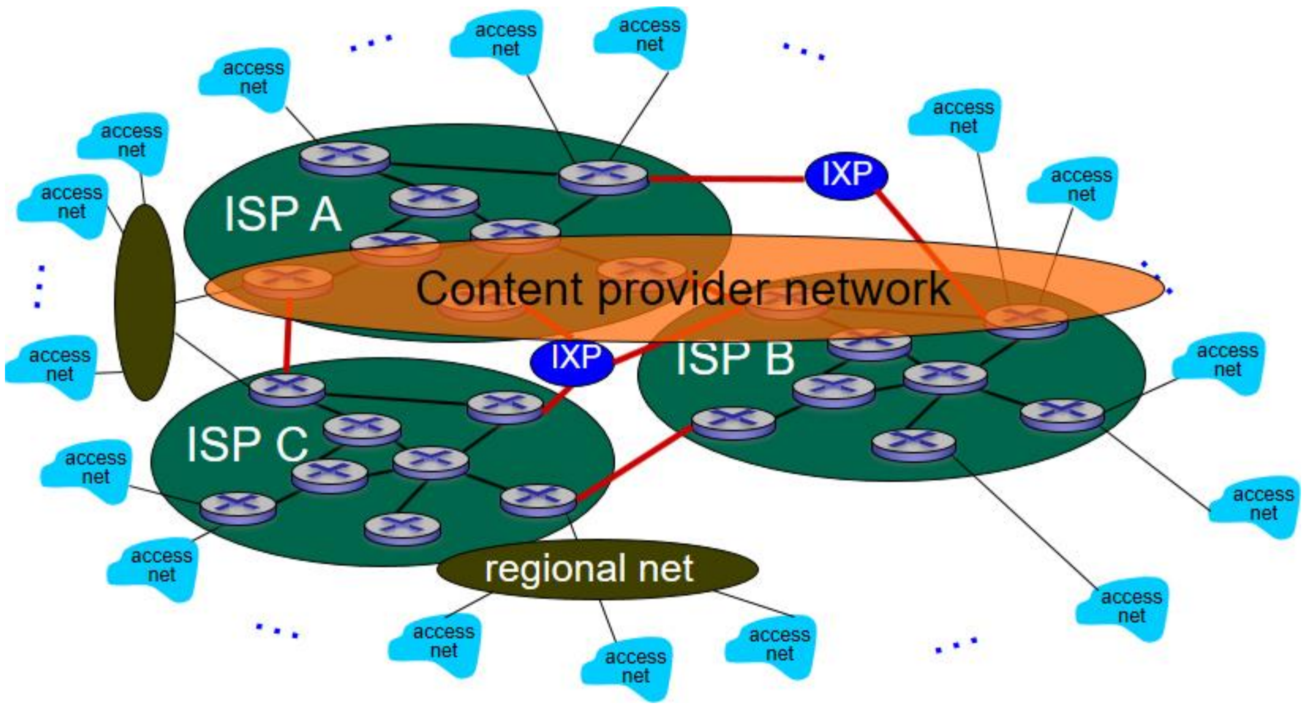
- **network edge:**
  - hosts: clients and servers
  - servers often in data centers
- **access networks, physical media***:* wired, wireless communication links
- **network core:**
  - interconnected routers
  - network of networks

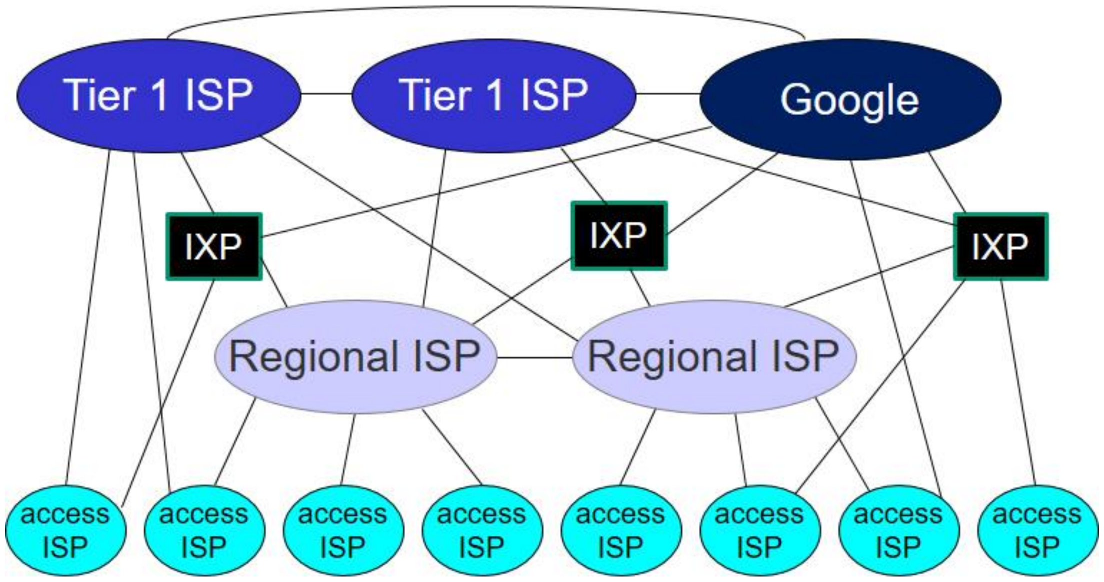


Internet Structure: Network of Networks

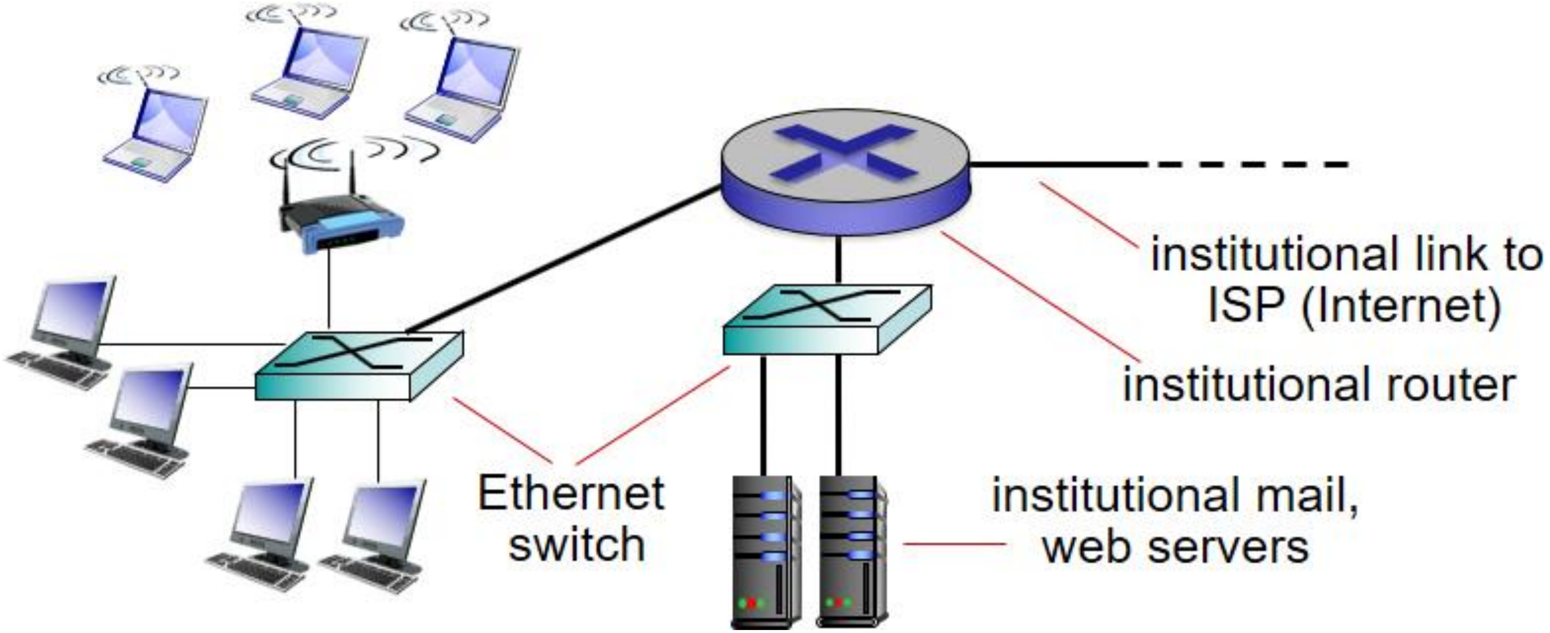
A complex structure of ***inter-connected networks*** among

- ISPs, Telecommunications providers, Content provider networks (e.g., Google, Microsoft, Akamai)
- At center: small number of well-connected large networks
- Data centers concentrate many servers (“the cloud”), Most traffic is content from data centers (esp. video)
- Between networks, traffic exchange is set by business agreements: Peering and transit payments





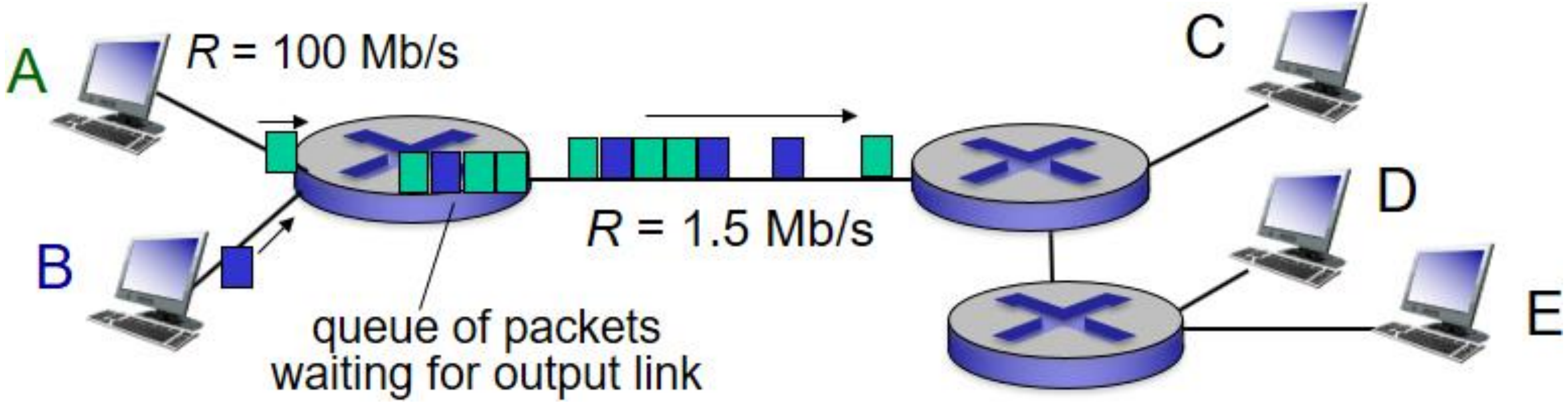
Enterprise Access Networks (Ethernet)



- typically used in companies, universities, etc.
- 10 Mbps, 100Mbps, 1Gbps, 10Gbps transmission rates
- today, end systems typically connect into Ethernet switch

Packet Switching: Queueing

- **packet-switching**: hosts break application-layer messages into packets
  - forward packets from one router to the next, across links on path from source to destination
  - each packet transmitted at full link capacity
- **store and forward:** entire packet must arrive at router before it can be transmitted on next link



- **great for bursty data:** resource sharing (statistical multi-plexing), simpler, no call setup
- **queuing and loss:**

  if arrival rate (in bits) to link exceeds transmission rate of link for a period of time:

  packets will queue, wait to be transmitted on link

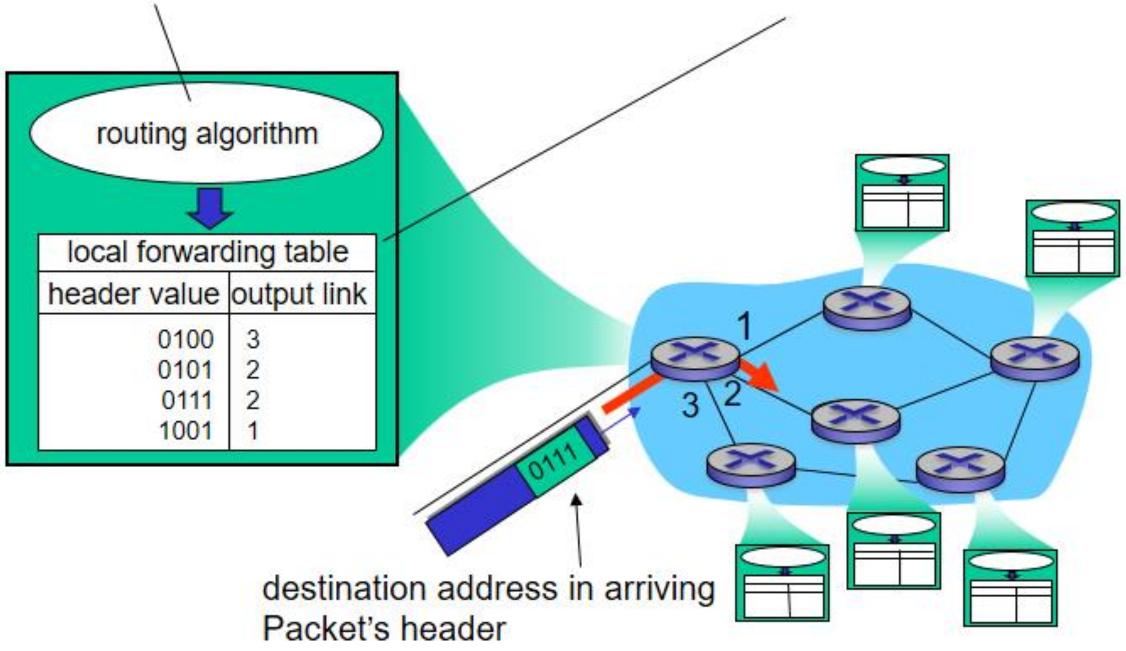
  packets can be dropped (lost) if memory (buffer) fills up
- Alternative principle : circuit switching

Two Key Network-Core Functions

routing: determines source-destination route taken by packets

- routing algorithms

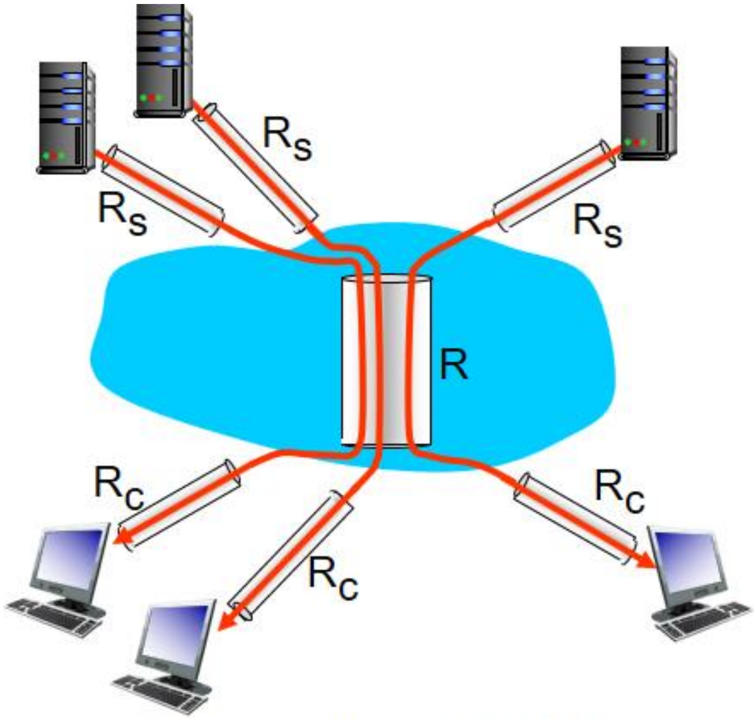
forwarding: move packets from router's input to appropriate router output



Throughput: Internet Scenario

- per-connection end-end throughput

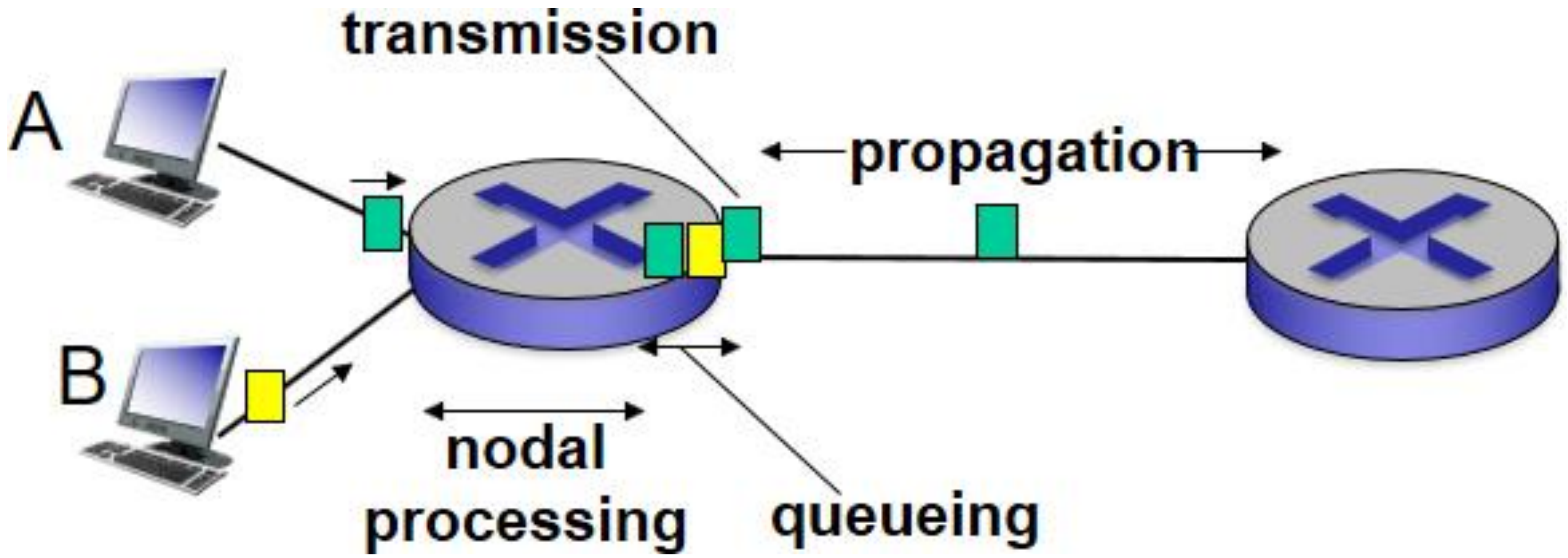
- in practice: R_c or R_s is often bottleneck



10 connections (fairly) share backbone bottleneck link *R* bits/sec

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Four Sources of Packet Delay



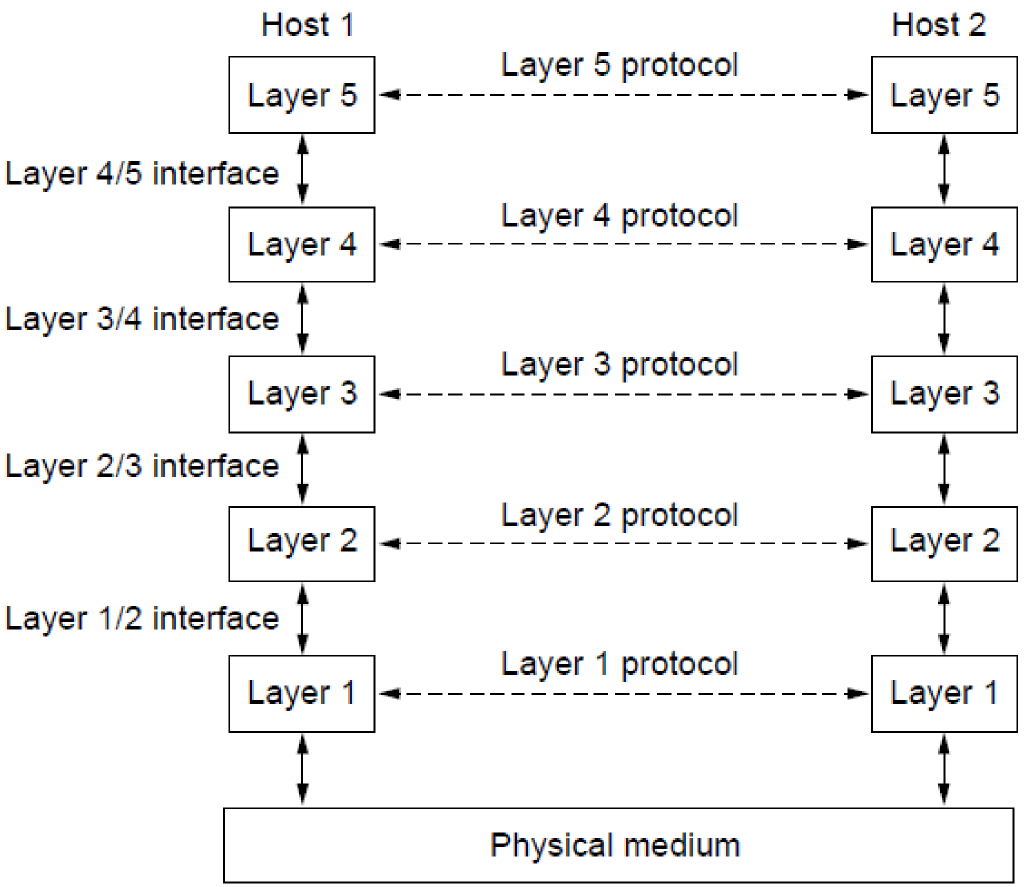
$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

Internet Protocol Stack

Protocol Layers

- Protocol layering is the main structuring method used to divide up network functionality.
- Make abstractions that hide specific complexities/problems of the underlying system
- **A protocol** is a set of rules that describe how messages are exchanged and formatted

  - Each protocol instance talks virtually to its peer
  - Each layer communicates only by using the one below
  - Lower layer services are accessed by an interface
  - At bottom, messages are carried by the medium



Why Layering?

- dealing with complex systems:
- explicit structure allows identification, relationship of complex system’s pieces
  - layered reference model for discussion
- modularization eases maintenance, updating of system
  - change of implementation of layer’s service transparent to rest of system
  - IPv4 to IPv6 change does not affect hardware layer

OSI Reference Model

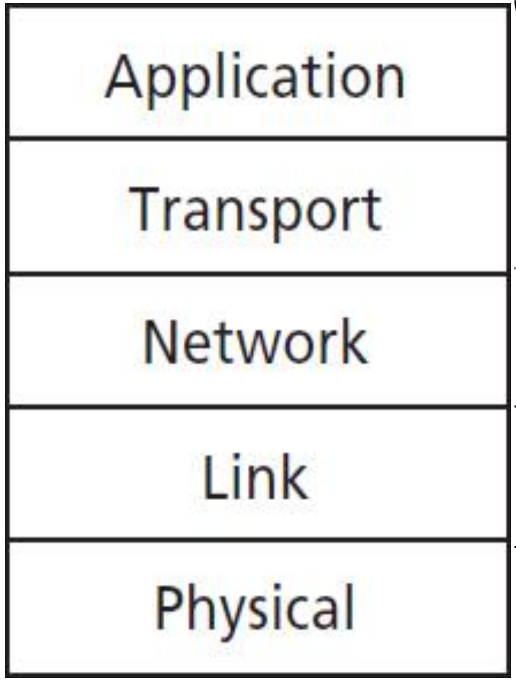
- A principled, international standard, seven layer model to connect different systems

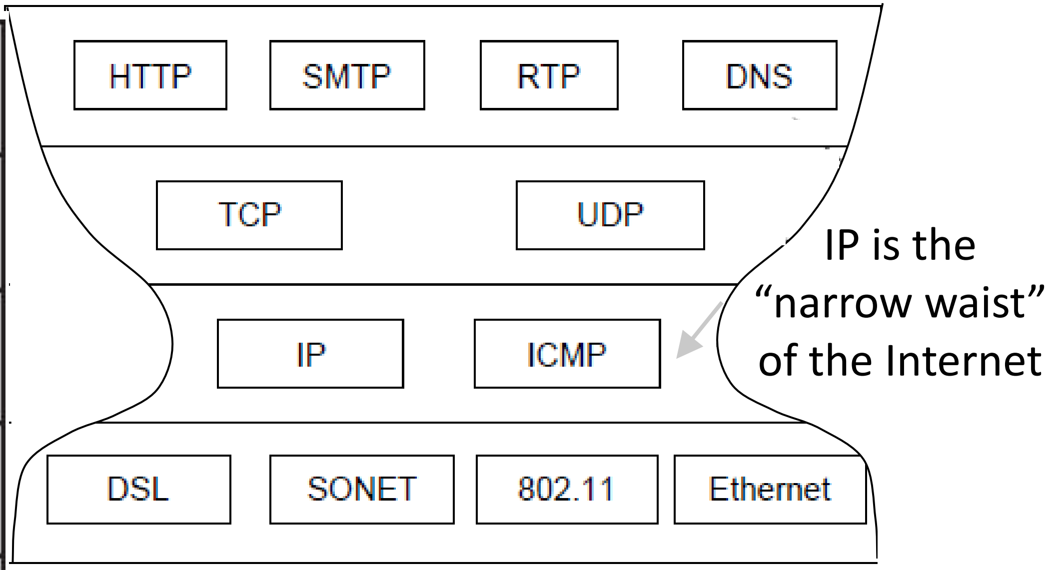
7	Application	– Provides functions needed by users
6	Presentation	– Converts different representations: allow applications to interpret meaning of data, e.g., encryption, compression, machine-specific conventions
5	Session	– Manages task dialog: synchronization, checkpointing, recovery of data exchange
4	Transport	– Provides end-to-end delivery
3	Network	– Sends packets over multiple links
2	Data link	– Sends frames of information
1	Physical	– Sends bits as signals

TCP/IP Reference Model

- A four layer model derived from experimentation; omits some OSI layers and uses the IP as the network layer.

- **application:** supporting network applications
  - FTP, SMTP, HTTP
- **transport:** process-process data transfer
  - TCP, UDP
- **network:** routing of datagrams from source to destination
  - IP, routing protocols
- **link:** data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- **physical:** bits "on the wire"





Protocols are shown in their respective layers

Critique of OSI & TCP/IP

OSI:

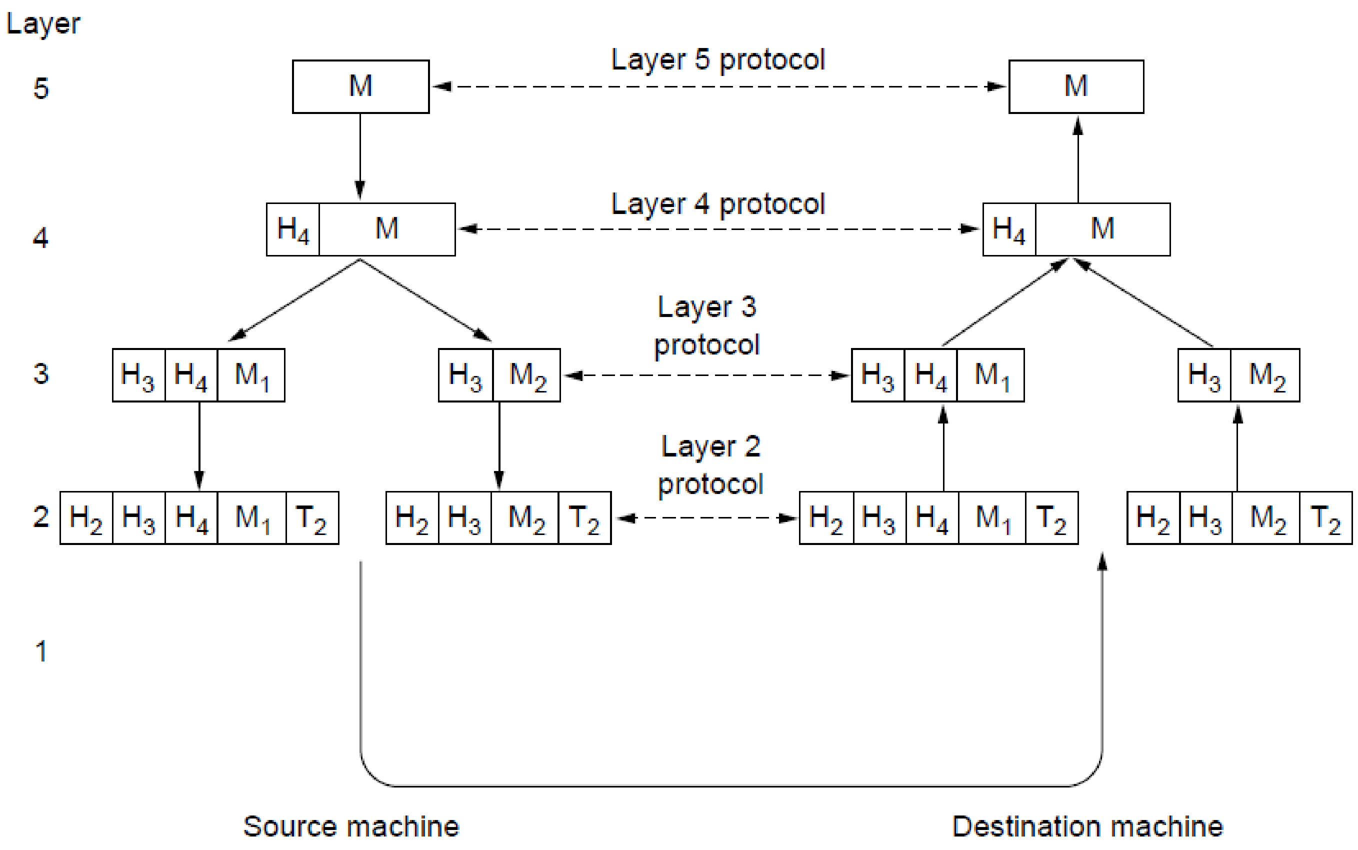
- \+ Very influential model with clear concepts
- Models, protocols and adoption all bogged down by politics and complexity

TCP/IP:

- \+ Very successful protocols that worked well and thrived
- Weak model derived after the fact from protocols

Encapsulation

- Control flow goes down the protocol layers at the sender, and up at the receiver
- Header-payload model
- Each lower layer adds its own <u>header</u> (with control information) to the message to transmit and removes it on receive
    - "Enveloping"
- Layers may also split and join messages, etc: Segmentation and re-assembly



Link Layer

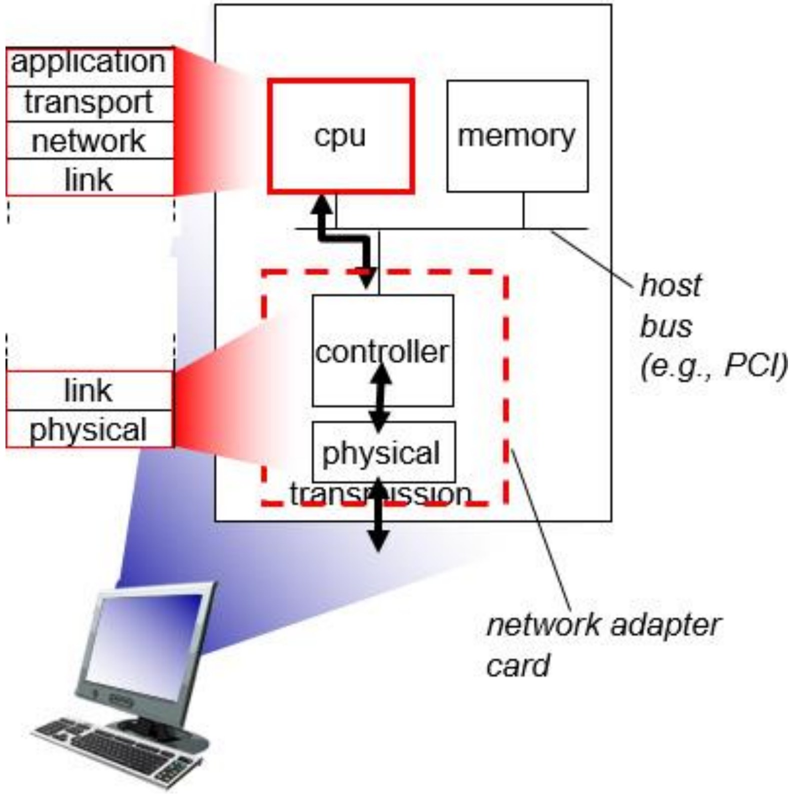
Link Layer: Context

- Datagram transferred by different link protocols over different links:
  - e.g., Ethernet on first link, frame relay on intermediate links, 802.11 on last link
- each link protocol provides different services
  - e.g., may or may not provide reliable data transmission over link
- Many tasks

- **framing, link access:**
  - encapsulate datagram into frame, adding header, trailer
  - channel access if shared medium
  - "MAC" addresses used in frame headers to identify source, destination
    - different from IP address!
- **reliable delivery between adjacent nodes**
  - seldom used on low bit-error link (fiber, some twisted pair)
  - wireless links: high error rates
- **flow control:**
  - pacing between adjacent sending and receiving nodes
- **error detection:**
  - errors caused by signal attenuation, noise.
  - receiver detects presence of errors:
    - signals sender for retransmission or drops frame
- **error correction:**
  - receiver identifies **and corrects** bit error(s) without resorting to retransmission
- **half-duplex and full-duplex**
  - with half duplex, nodes at both ends of link can transmit, but not at same time

Where is the Link Layer Implemented?

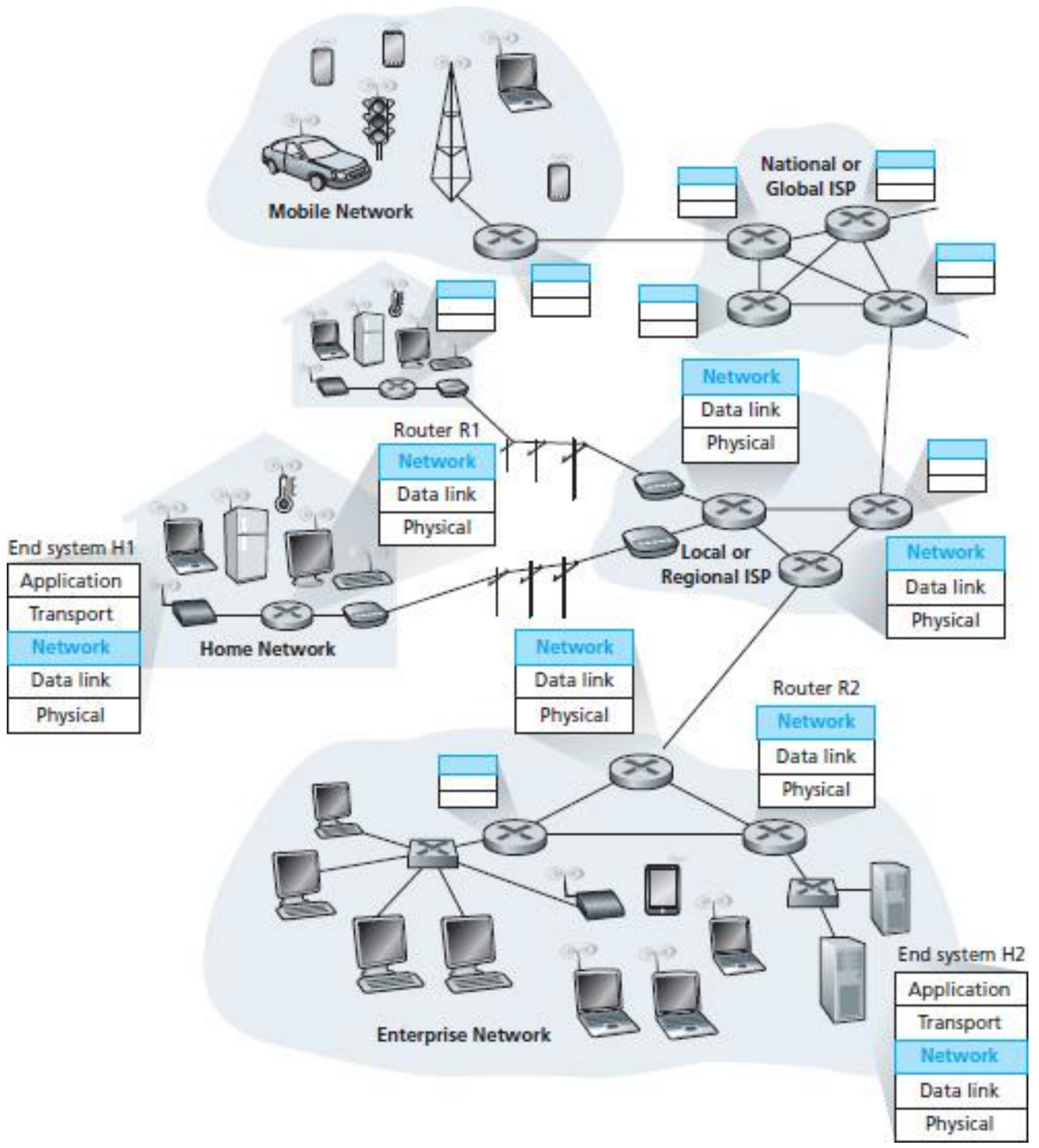
- in each and every host
- link layer implemented in "adaptor" (aka **network interface card** NIC) or on a chip
  - Ethernet card, 802.11 card; Ethernet chipset
  - implements link, physical layer
- attaches into host's system buses
- combination of hardware, software, firmware
- **MAC (or LAN or physical or Ethernet) address:**
  - function: **used 'locally" to get frame from one interface to another physically-connected interface (same network, in I P-addressing sense)**
  - 48 bit MAC address (for most LANs) burned in NIC ROM, also sometimes software settable
  - e.g.: 1A-2F-BB-76-09-AD



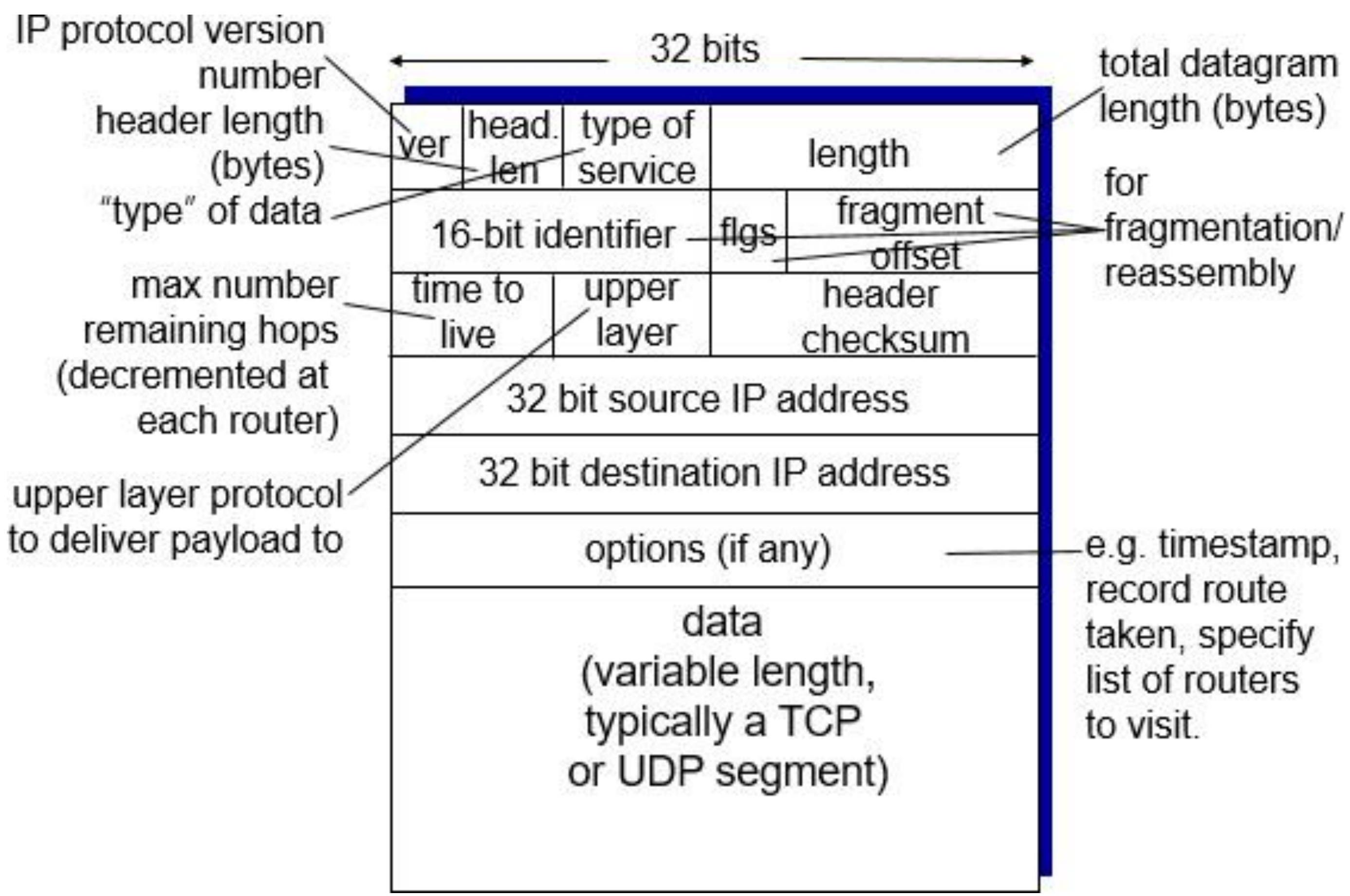
IP layer

Network Layer

- transport segment from sending to receiving host
- on sending side encapsulates segments into datagrams
- on receiving side, delivers segments to transport layer
- network layer protocols in **every** host, router
- router examines header fields in all IP datagrams passing through it
- Routing algorithm (dynamically) computes forwarding table at routers
- DISCLAIMER: We omit any discussion of routing algorithms!



IP Datagram Format

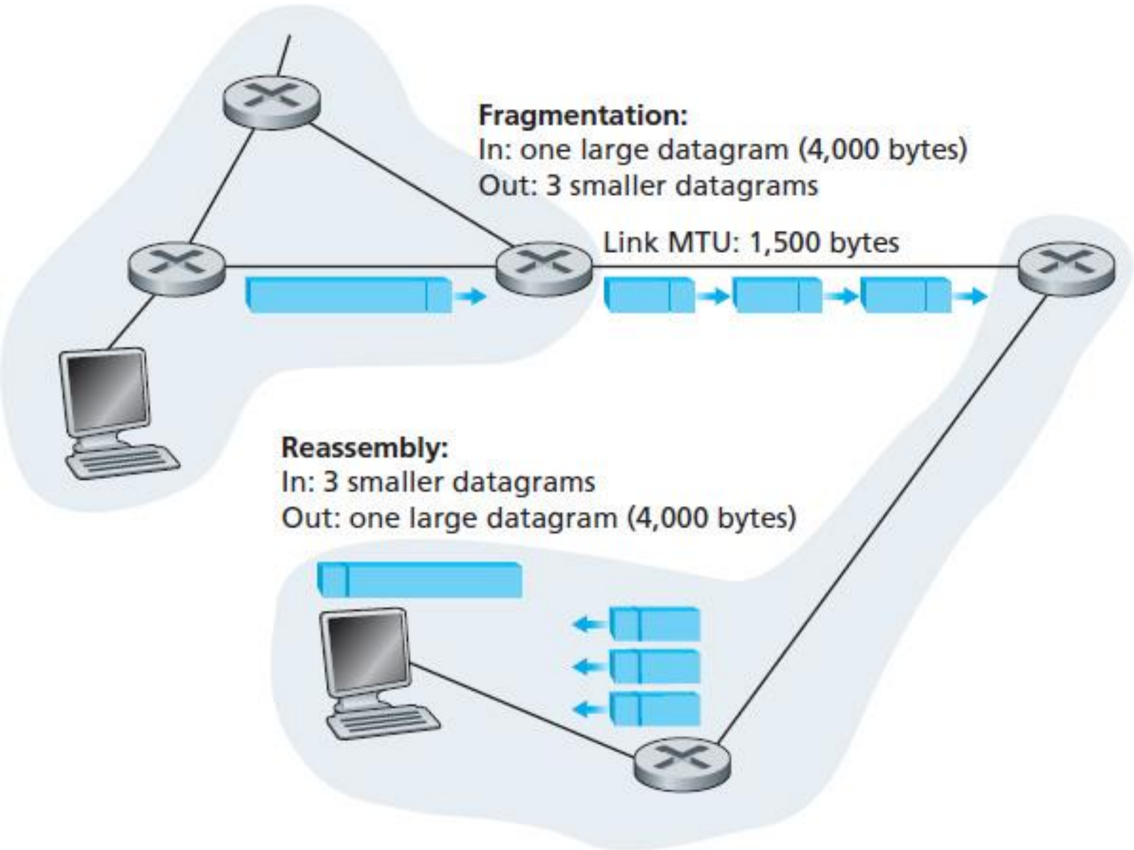


how much overhead?

- 20 bytes of TCP
- 20 bytes of IP
- = 40 bytes + app layer overhead

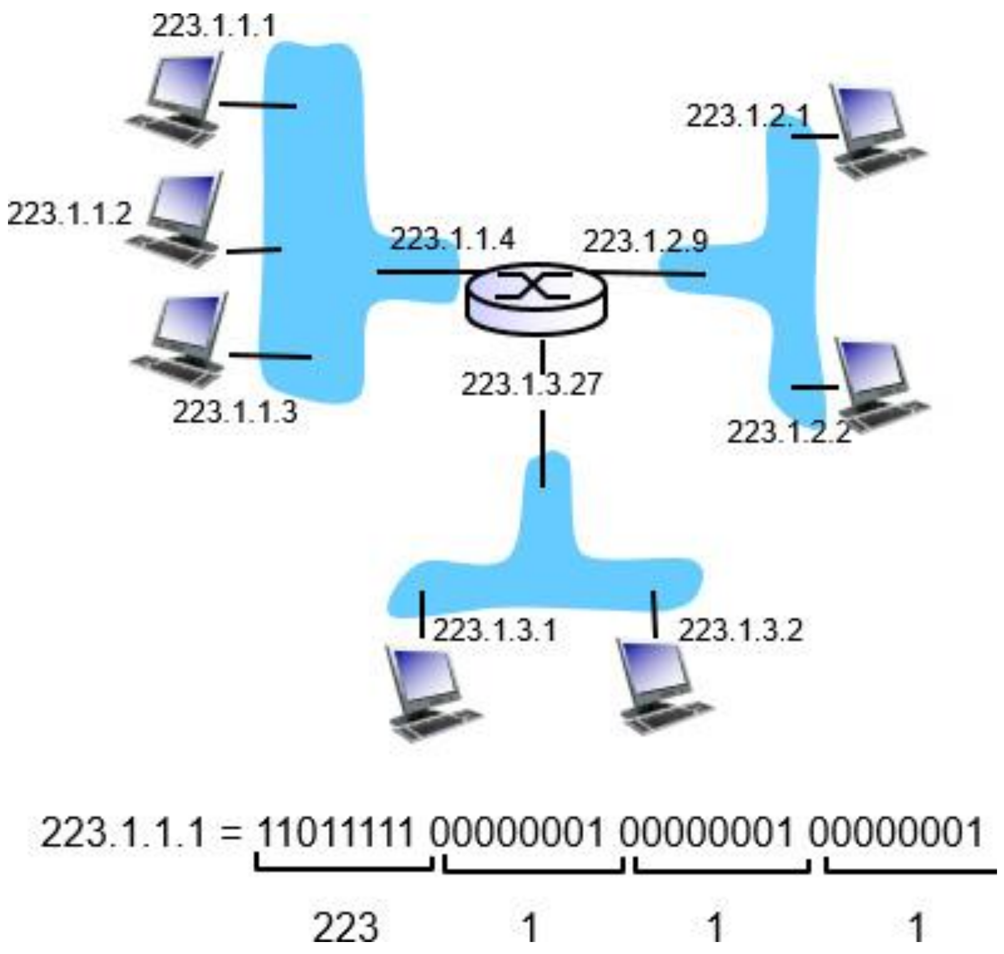
IP Fragmentation, Reassembly

- theoretically largest IP datagram = 2^{16} ~65K
- network links have MTU (max.transfer size) - largest possible link-level frame
  - different link types, different MTUs
- large IP datagram divided (“fragmented”) within net
  - one datagram becomes several datagrams
  - “reassembled” only at final destination
  - IP header bits used to identify, order related fragments



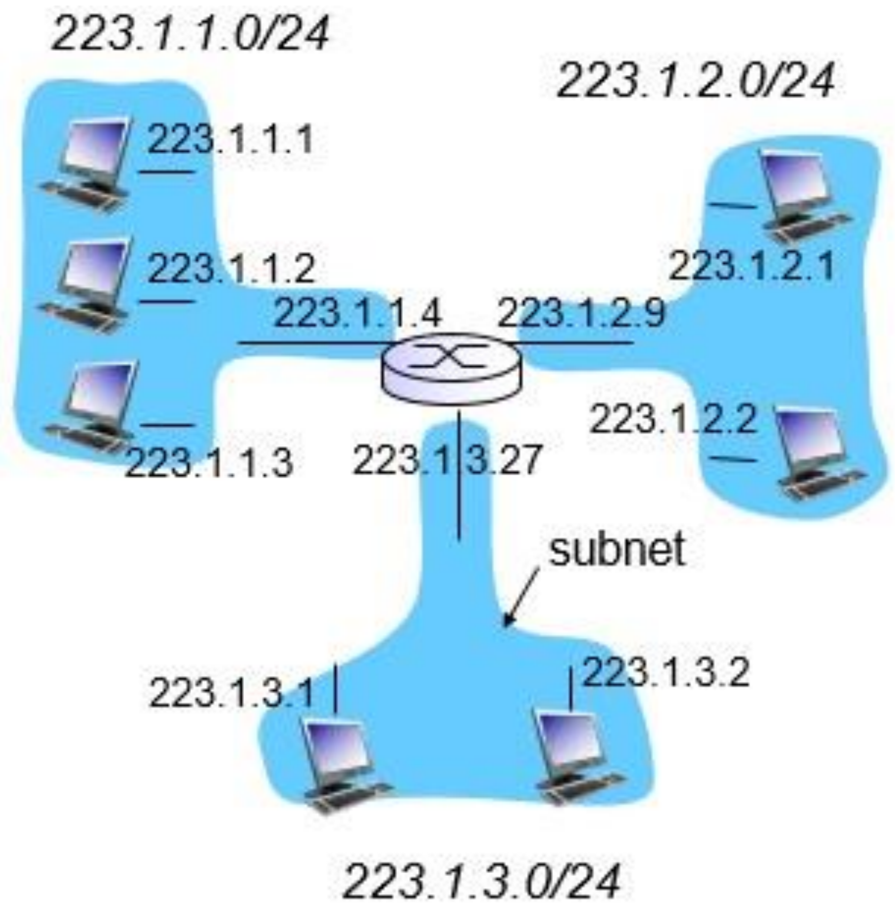
IP Addressing: Introduction

- **IP address (IPv4):** 32-bit identifier for host, router **interface**
- **interface:** connection between host/router and physical link
    - Router’s typically have multiple interfaces
    - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)
- **IP addresses associated with each interface**



Subnets

- **IP address:**
  - subnet part - high order bits
  - host part - low order bits
- **What's subnet ?**
  - device interfaces with same subnet part of IP address
  - can physically reach each other **without intervening router**
- **recipe**
  - to determine the subnets, detach each interface from its host or router, creating islands of isolated networks
  - each isolated network is called a **subnet**

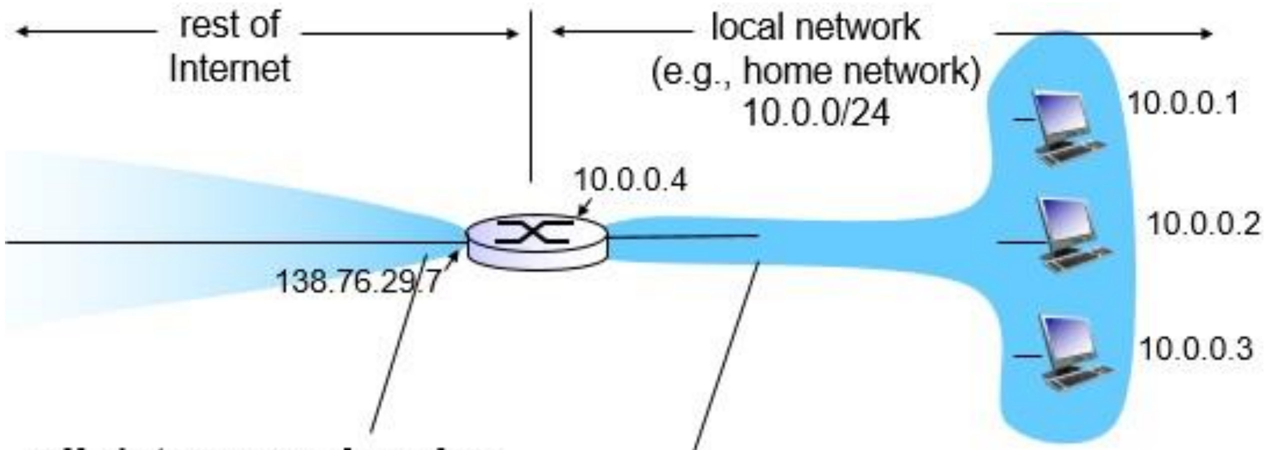


subnet mask: /24

NAT: Network Address Translation

motivation: local network uses just one IP address as far as outside world is concerned:

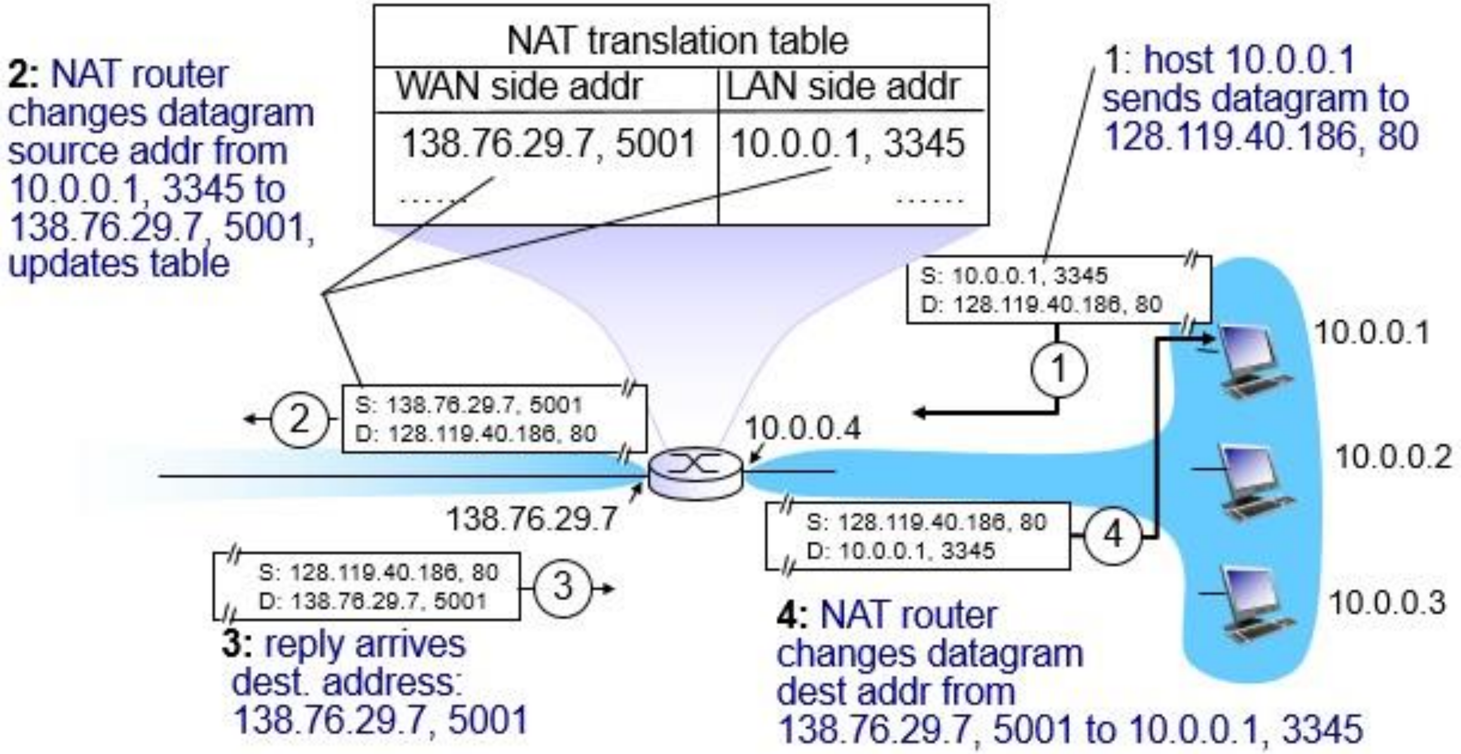
- range of addresses not needed from ISP: just one IP address for all devices
- (Dirty?!) trick to save IP addresses.
- can change addresses of devices in local network without notifying outside world
- can change ISP without changing addresses of devices in local network
- devices inside local net not explicitly addressable, visible by outside world (a security plus)



all datagrams **leaving** local network have **same** single source NAT IP address: 138.76.29.7,different source port numbers

datagrams with source or destination in this network have 10.0.0/24 address for source, destination (as usual)

NAT: Network Address Translation



- 16-bit port-number field: 60,000 simultaneous connections with a single LAN-side address!
- NAT is controversial:
  - routers should only process up to layer 3
  - address shortage should be solved by IPv6
  - NAT traversal: what if client wants to connect to server behind NAT? P2P applications?!

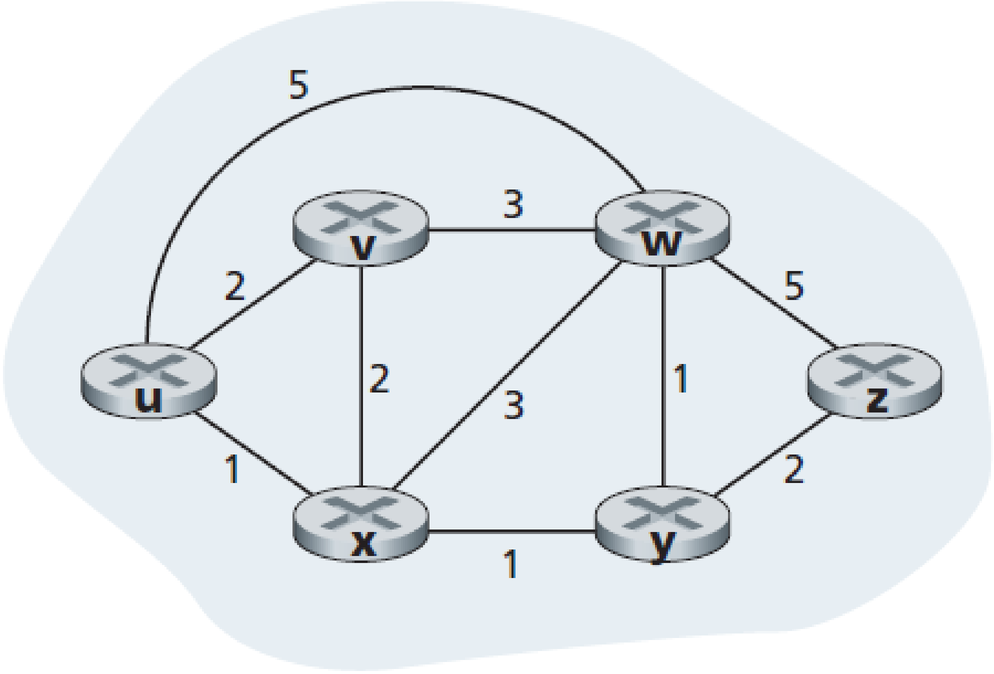
Routing

Routing Protocols

Routing protocol goal: determine “good” paths (a.k.a. routes), from sending host to receiving host

- path: sequence of routers packets will traverse in going from given initial source host to given final destination host
- need to define “good”:
  - least “cost”,
  - “fastest”,
  - “least congested”

Graph Abstraction of the Network

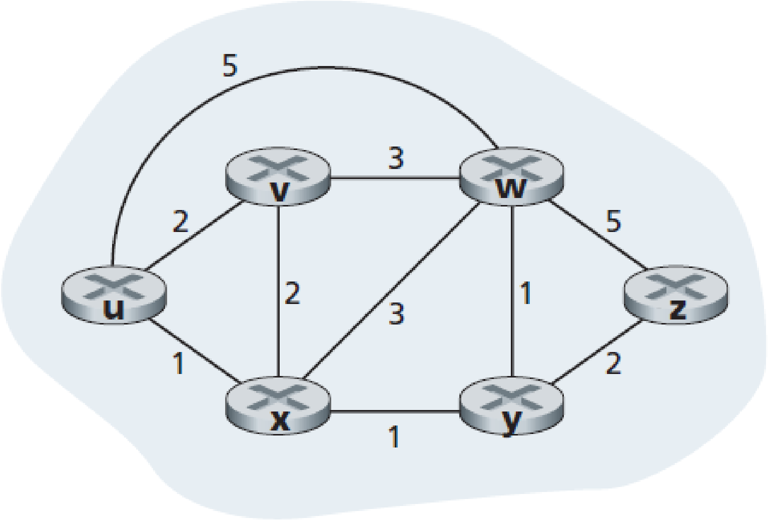


graph: G = (N, E)

N = set of routers = { u, v, w, x, y, z }

E = set of links ={ (u,v), (u,x), (v,x), (v,w), (x,w), (x,y), (w,y), (w,z), (y,z) }

Graph Abstraction: Costs



$c(x, x') = \text{cost of link } (x, x')$ e.g., $c(w, z) = 5$

cost could always be 1, or inversely related to bandwidth, or inversely related to congestion

cost of path $(x_1, x_2, x_3, \ldots, x_p) = c(x_1, x_2) + c(x_2, x_3) + \ldots + c(x_{p-1}, x_p)$

key question*:* what is the least-cost path between u and z?

routing algorithm*:* algorithm that finds that least cost path

Routing Algorithm Classification

Q: global or decentralized information?

global:

- all routers have complete topology, link cost info
- **"link state" algorithms**

decentralized:

- router knows physically-connected neighbors, link costs to neighbors
- iterative process of computation, exchange of info with neighbors
- **"distance vector" algorithms**

Q: static or dynamic?

static:

- routes change slowly over time

dynamic:

- routes change more quickly
  - periodic update
  - in response to link cost changes

Example of routing algorithm

A Link-State Routing Algorithm

Dijkstra's algorithm

- net topology, link costs known to all nodes
  - accomplished via "link state broadcast"
  - all nodes have same info
- computes least cost paths from one node ('source') to all other nodes
  - gives **forwarding table** for that node
- iterative: after k iterations, know least cost path to k destinations

notation:

- **c(x,y):** link cost from node
  - **C(x,y) =** ∞ if x and y are not connected directly
- **D(v):** current value of cost of path from source to dest. v
- **p(v):** predecessor node along path from source to v
- **N':** set of nodes whose least cost path definitively known

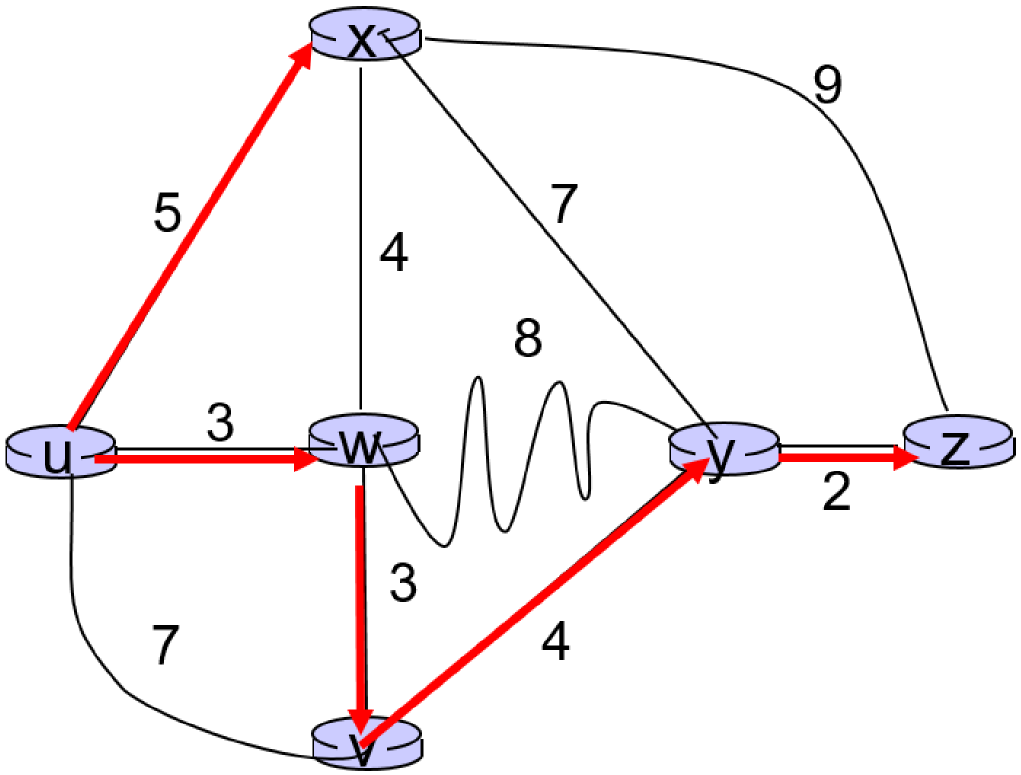
Dijsktra’s Algorithm

```
Initialization:
  N' = {u}
  for all nodes v
    if v adjacent to u
        then D(v) = c(u,v)
    else D(v) = ∞

Loop
   find w not in N' such that D(w) is a minimum
   add w to N'
   update D(v) for all v adjacent to w and not in N' :
      D(v) = min( D(v), D(w) + c(w,v) )
   /* new cost to v is either old cost to v or known
    shortest path cost to w plus cost from w to v */
until all nodes in N' `
```

Dijkstra's Algorithm: Example

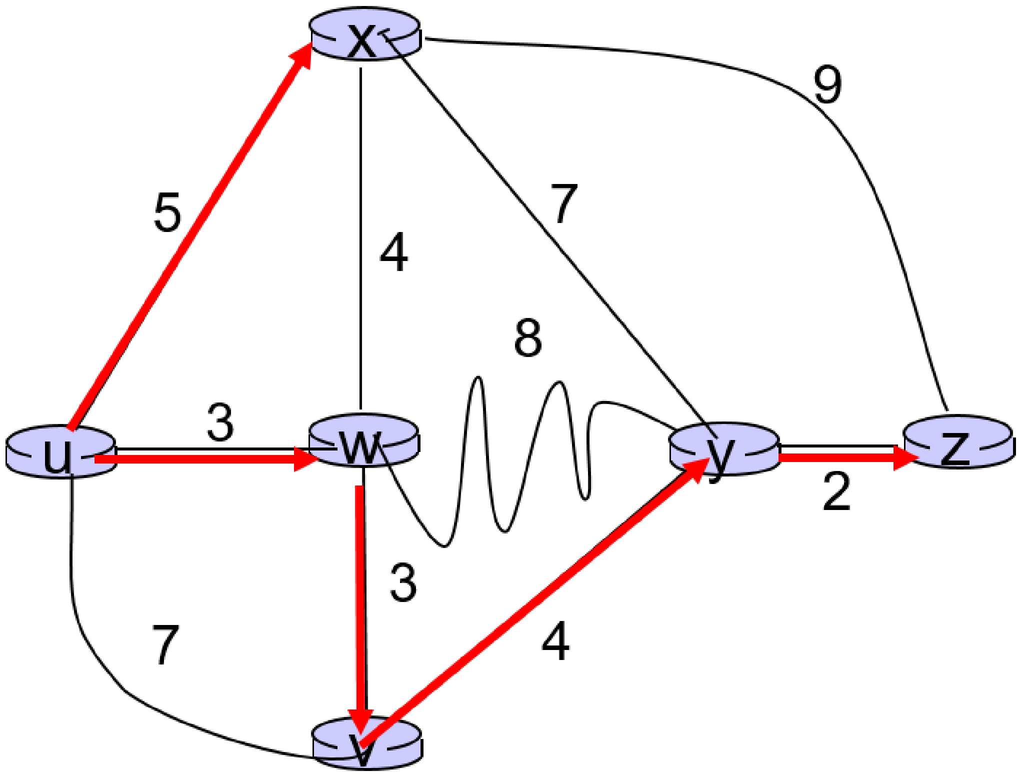
Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	3,u	5,u	∞	∞



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra's Algorithm: Example

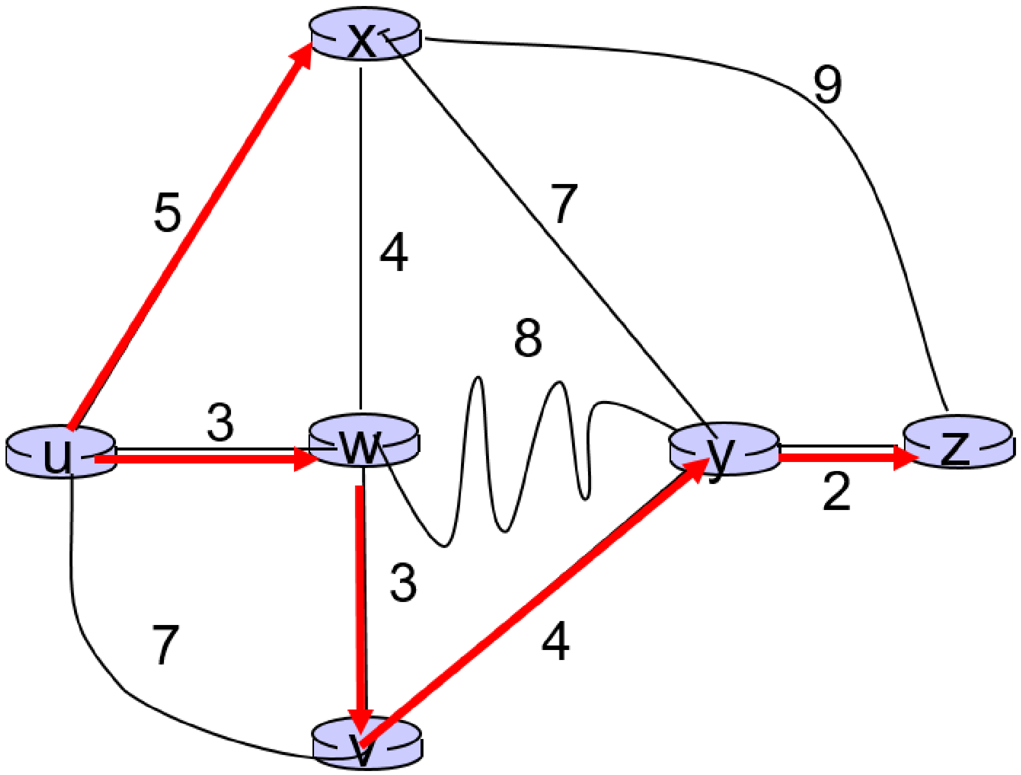
Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	3,u	5,u	∞	∞



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra's Algorithm: Example

Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞

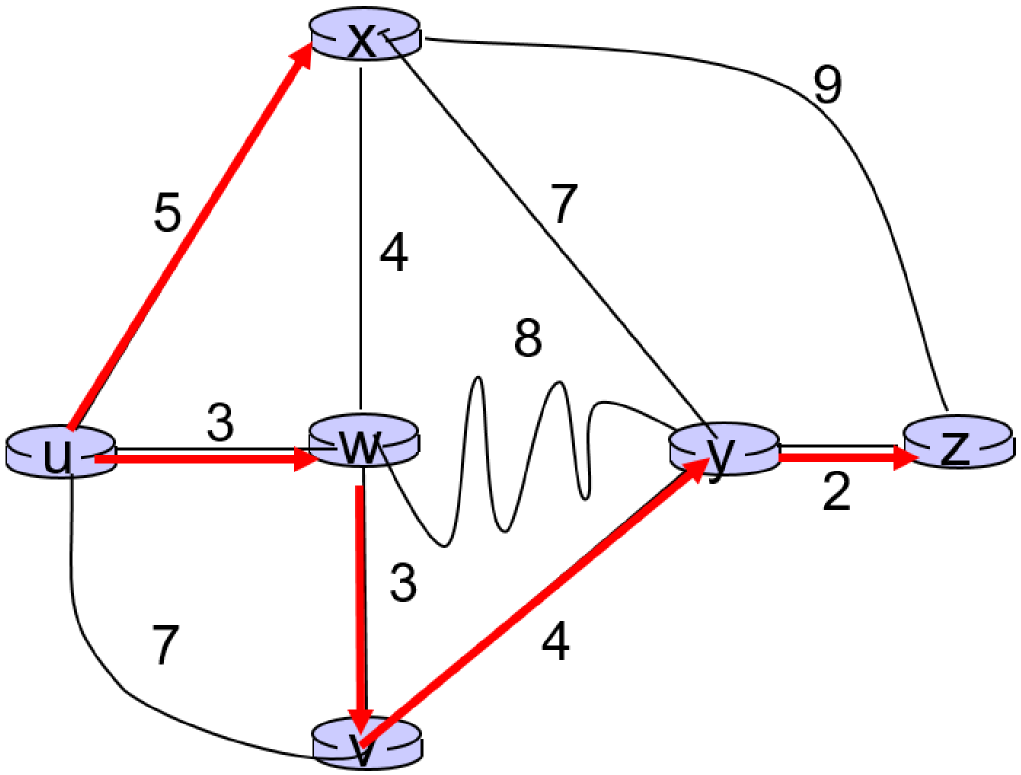


- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra’s Algorithm: Example

Step	N’	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞

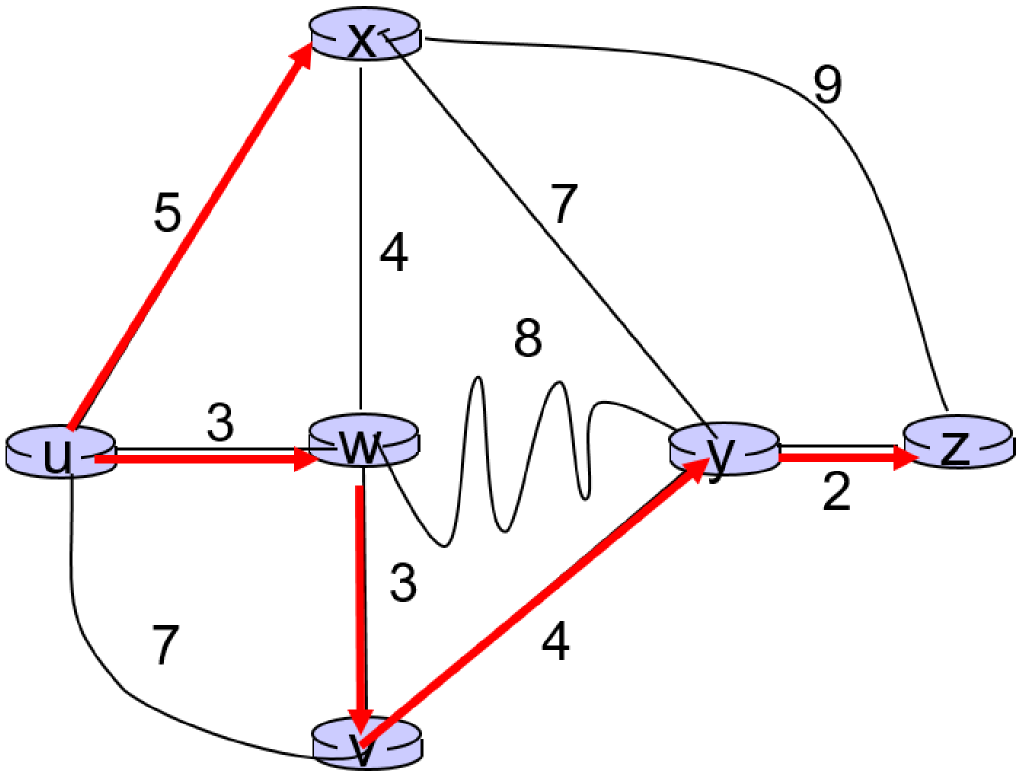
- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



Dijkstra's Algorithm: Example

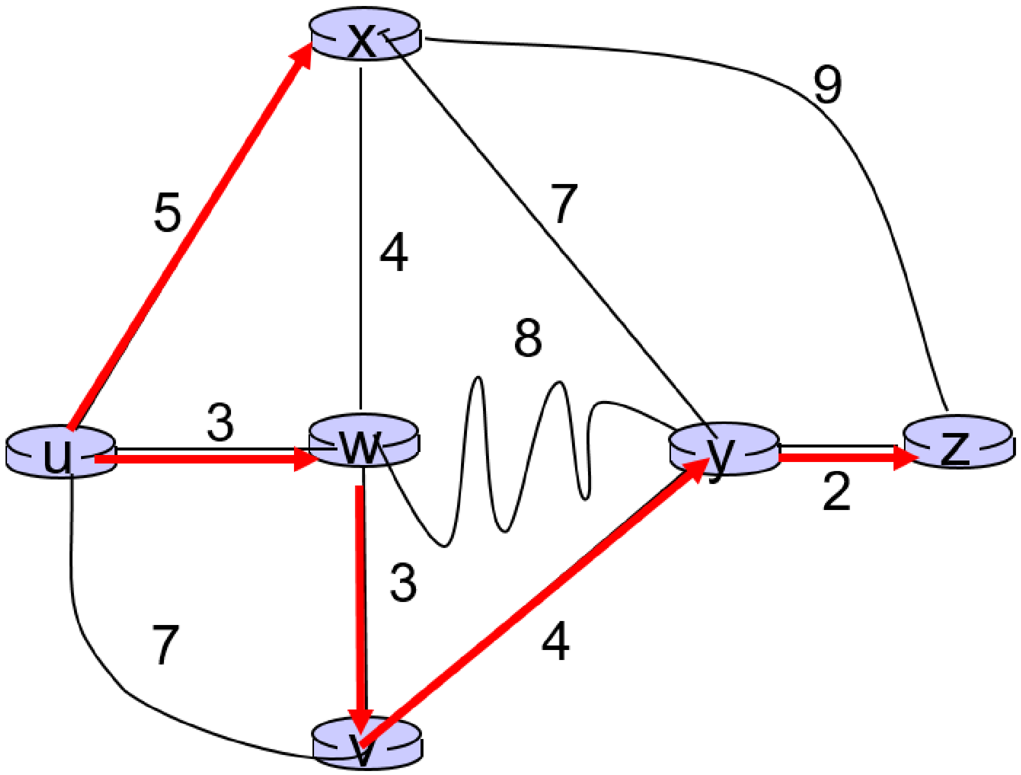
Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	6,w			11,w	14,x

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



Dijkstra's Algorithm: Example

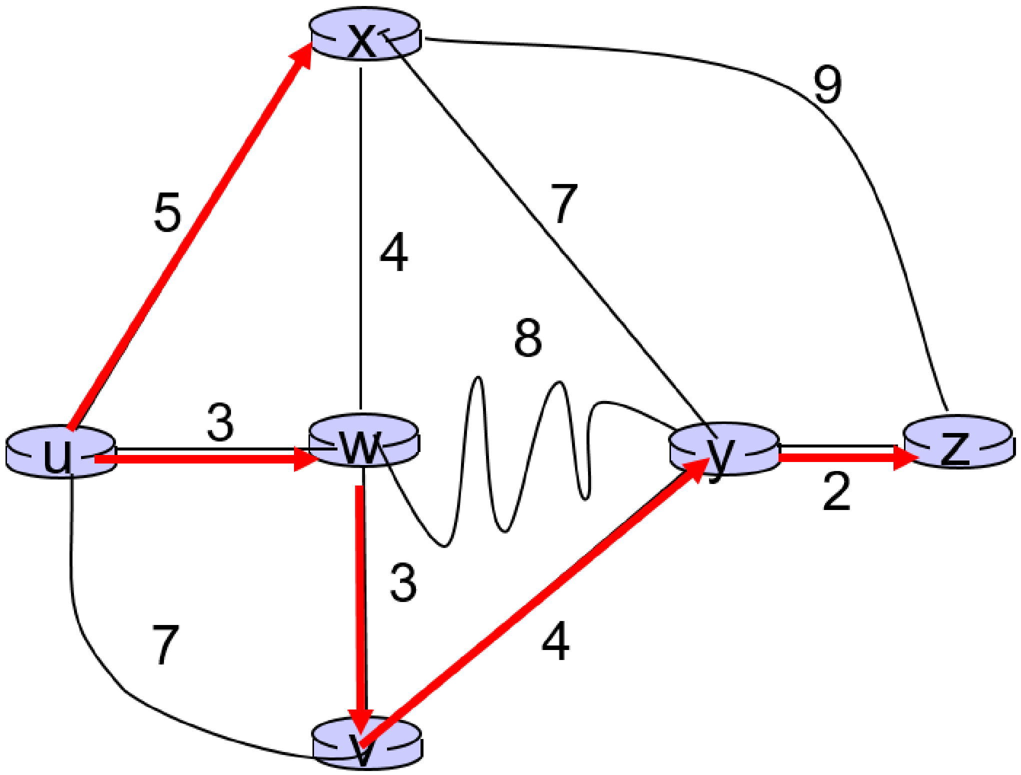
Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	6,w			11,w	14,x



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra’s Algorithm: Example

Step	N’	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	**6,w**			11,w	14,x
3	uwxv				10,v	14,x

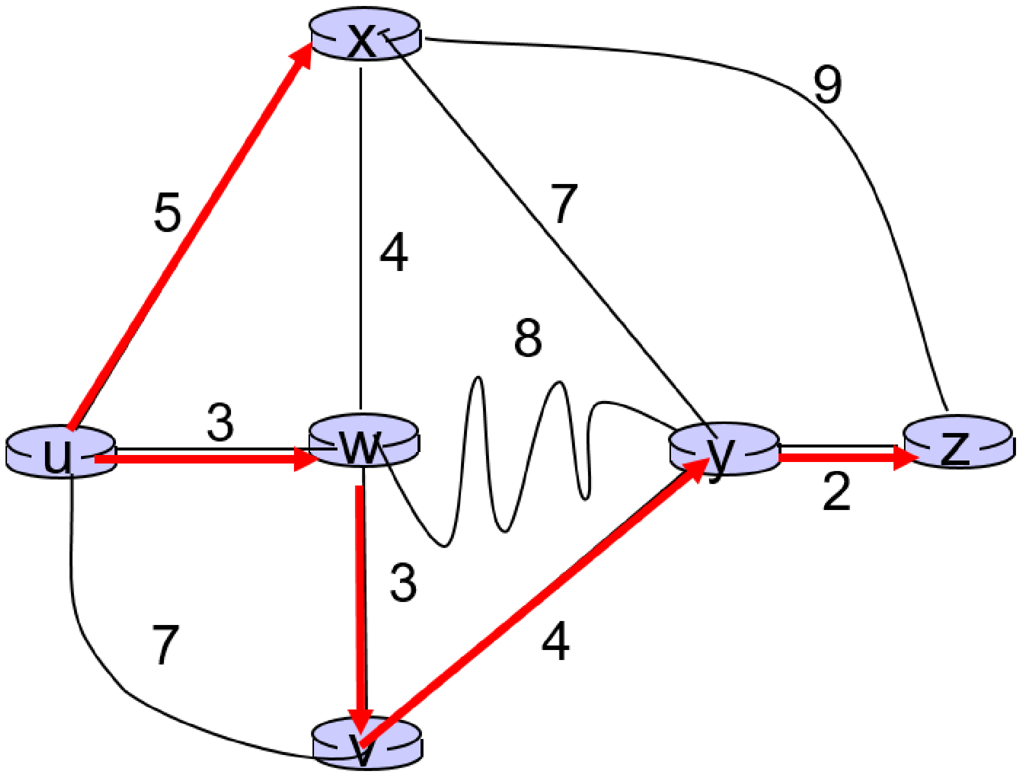


- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra’s Algorithm: Example

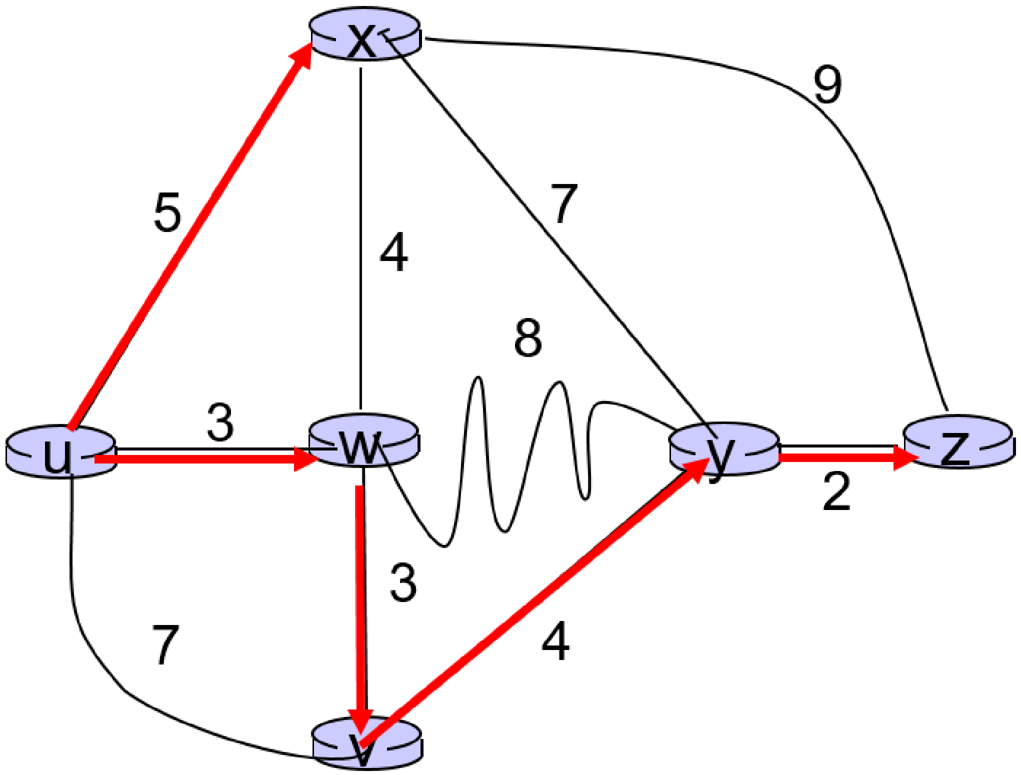
Step	N’	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	**6,w**			11,w	14,x
3	uwxv				10,v	14,x

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



Dijkstra's Algorithm: Example

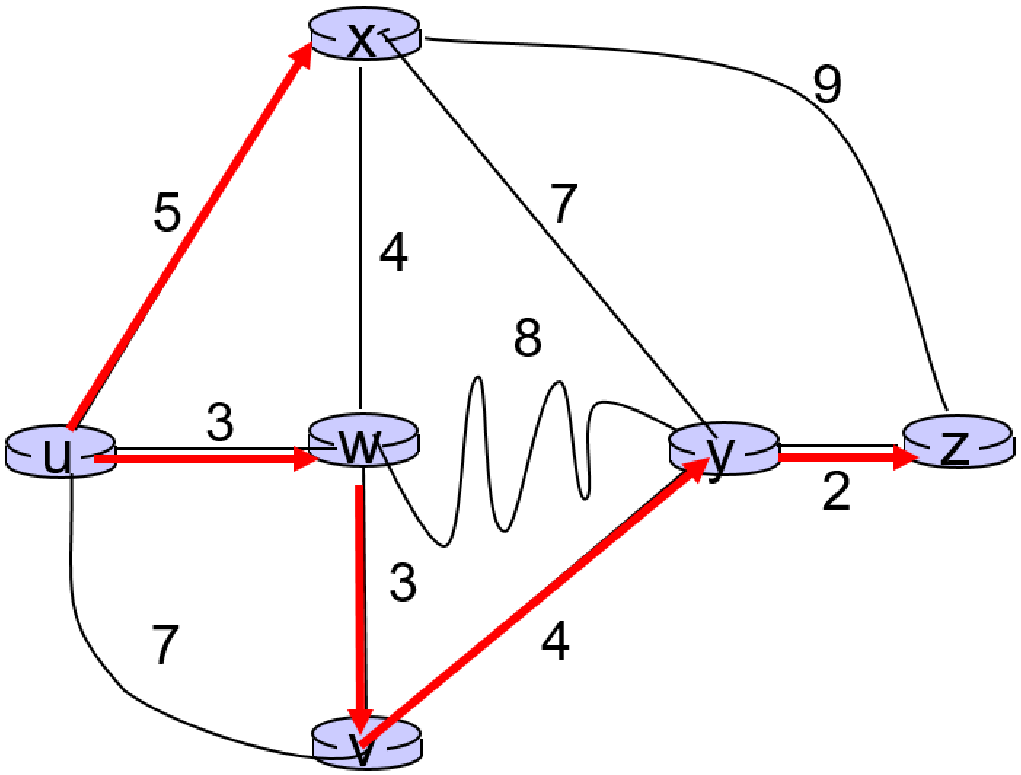
Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	**6,w**			11,w	14,x
3	uwxv				**10,v**	14,x
4	uwxvy					12,y



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra’s Algorithm: Example

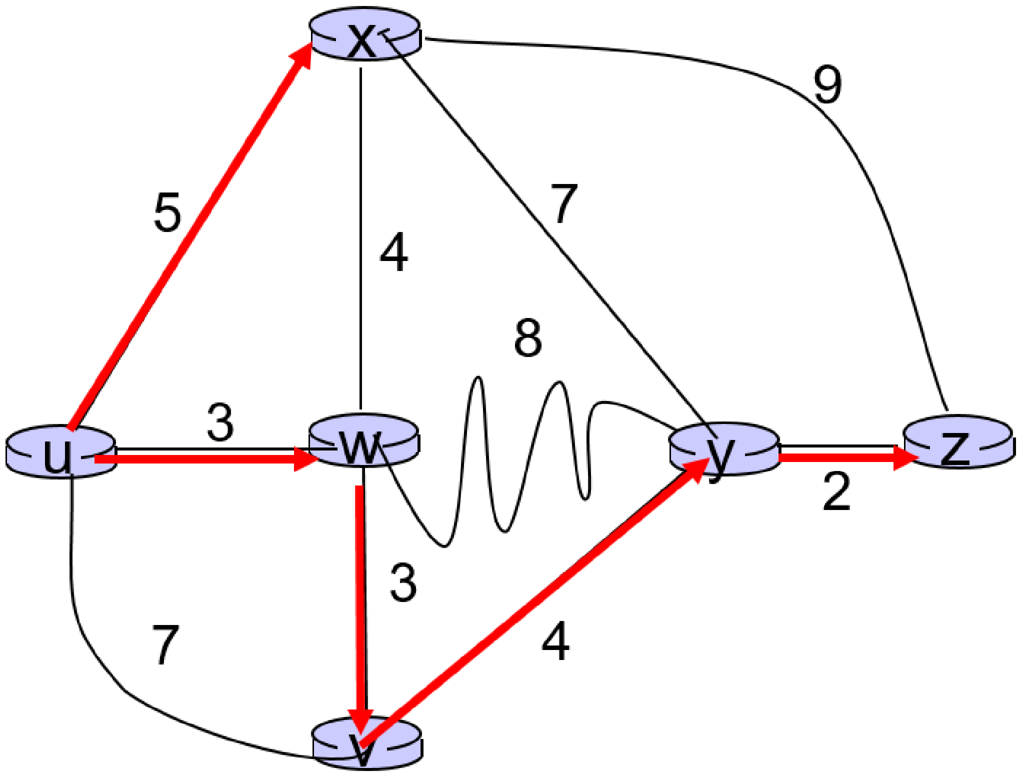
Step	N’	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	**6,w**			11,w	14,x
3	uwxv				**10,v**	14,x
4	uwxvy					12,y



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

Dijkstra's Algorithm: Example

Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	**3,u**	5,u	∞	∞
1	uw	6,w		**5,u**	11,w	∞
2	uwx	**6,w**			11,w	14,x
3	uwxv				**10,v**	14,x
4	uwxvy					**12,y**
5	uwxvyz					



- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)

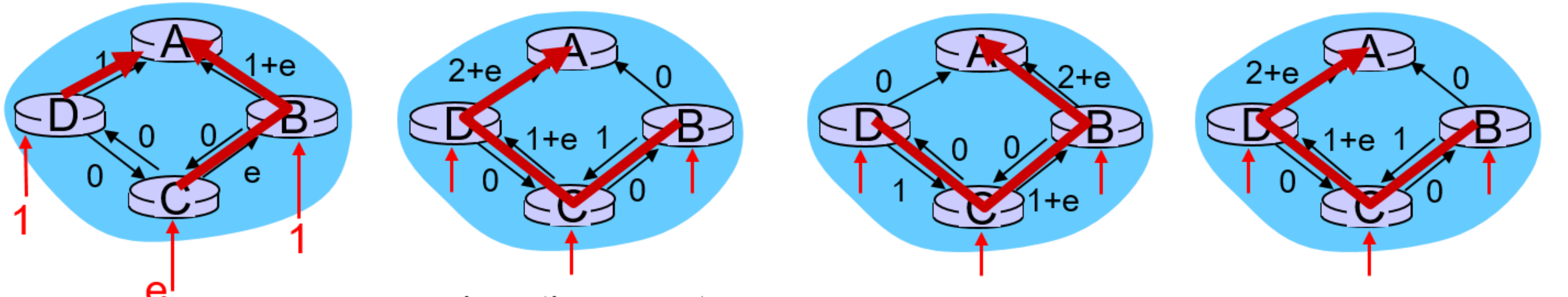
Dijkstra's Algorithm, Discussion

algorithm complexity: n nodes

- each iteration: need to check all nodes, w, not in N
- $n(n+1)/2$ comparisons: $O(n^2)$
- more efficient implementations possible: O(n log n)

oscillations possible:

- e.g., support link cost equals amount of carried traffic:



initially

given these costs, find new routing…. resulting in new costs

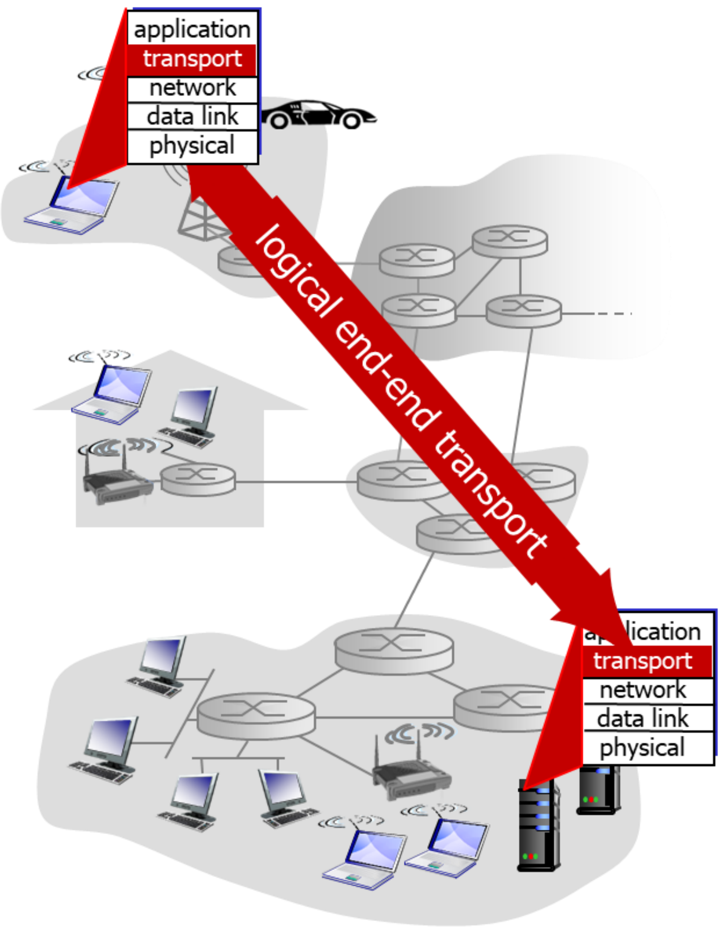
given these costs, find new routing…. resulting in new costs

given these costs, find new routing…. resulting in new costs

Transport Layer Protocols

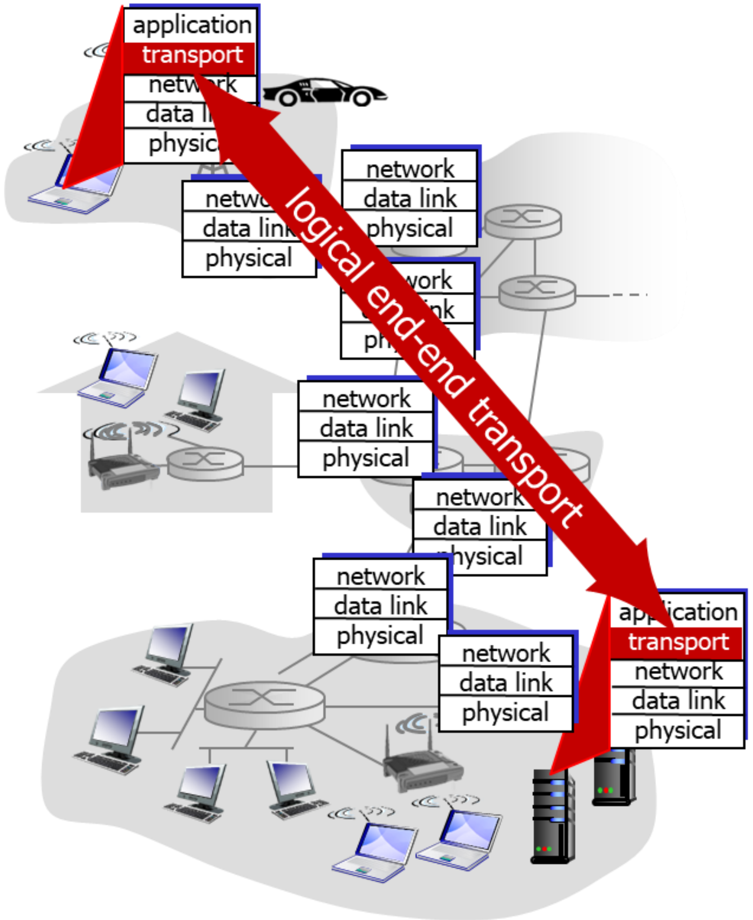
Transport Services and Protocols

- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Internet Transport-Layer Protocols

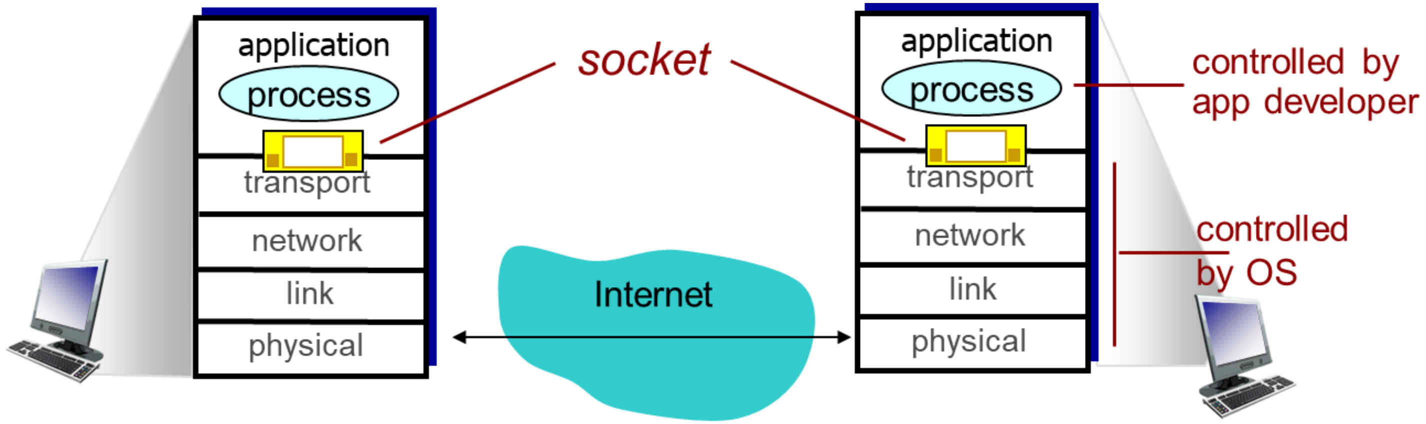
- **“Transmission control protocol (TCP)”**
  - Connection Oriented
  - Reliable*), in-order delivery, byte-stream
    - congestion control: throttle sender when network overloaded
    - flow control: sender won’t overwhelm receiver
    - connection setup
- ***“User datagram Protocol (UDP)”***
  - Connectionless, unreliable,
    - no-frills extension of “best-effort” IP
    - Message-loss
    - Messages delivered out-of-order
  - Streaming multimedia apps (loss tolerant, rate sensitive), DNS, SNMP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Reliable=if neither sender nor receiver fails during transmissions, and if network failures doesn’t persist (too long), data is eventually delivered!

Sockets

- The main API for programming network applications
- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



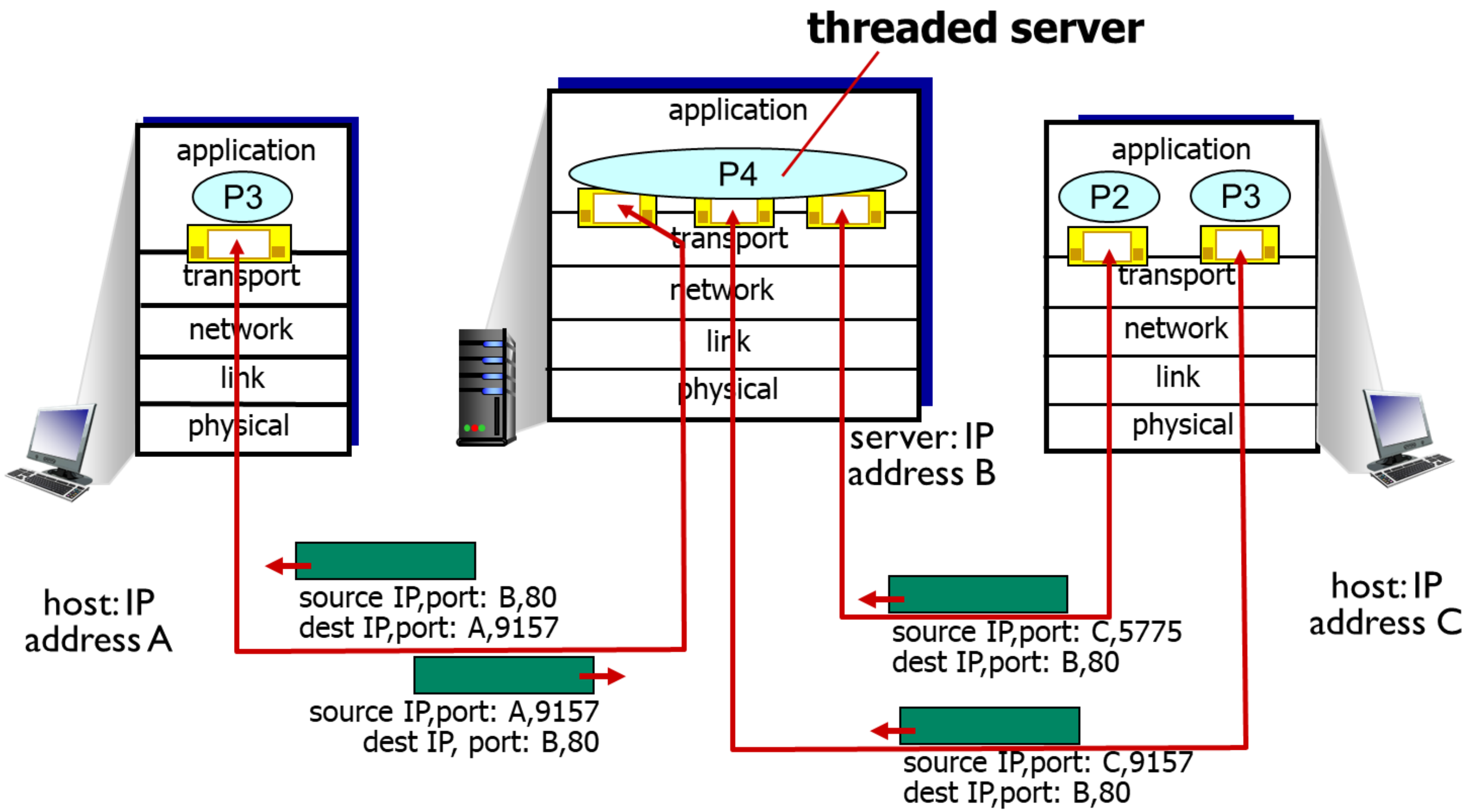
Addressing Processes

- to receive messages, process must have identifier
- host device has unique 32-bit IP address
  - Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, many processes can be running on same host
- identifier includes both IP address and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
  - to send HTTP message to gaia.cs.umass.edu web server:
    - IP address: 128.119.245.12
    - port number: 80

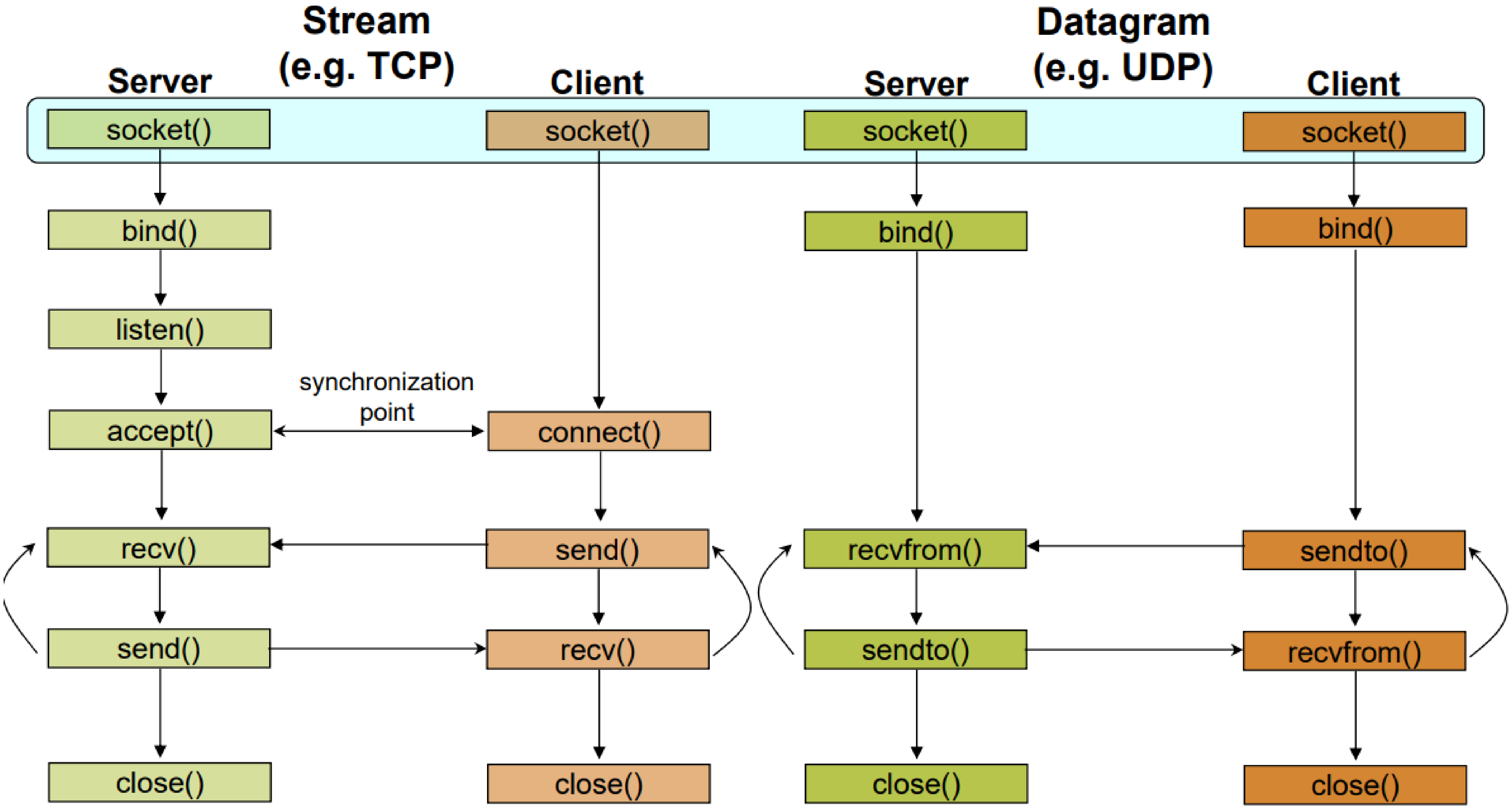
Well-known ports: https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers

Connection-Oriented Demux: Example

- TCP socket identified by 4-tuple:
  - **source IP address**
  - **source port number**
  - **dest IP address**
  - **dest port number**
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request



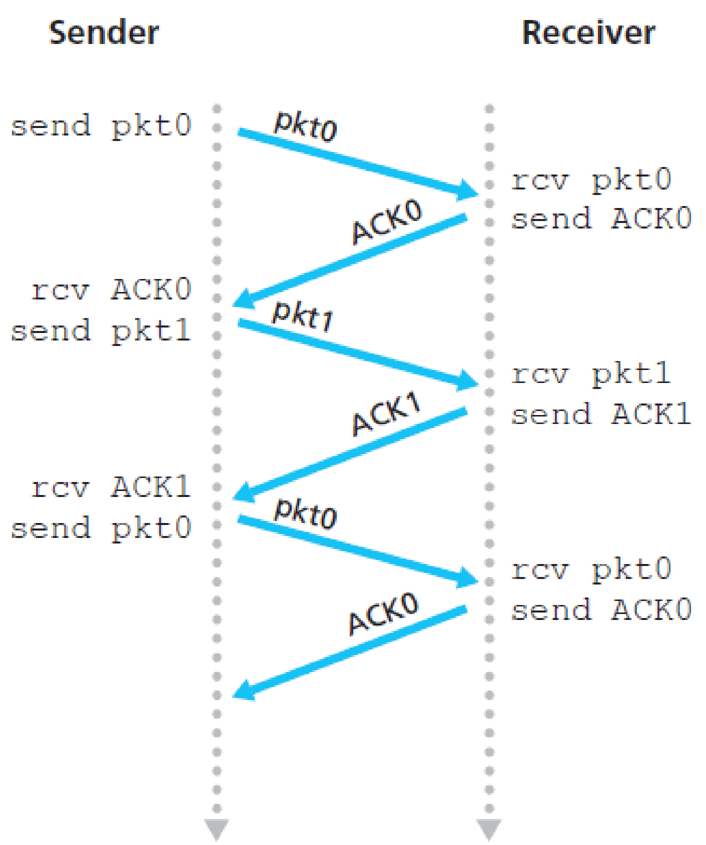
The Socket API



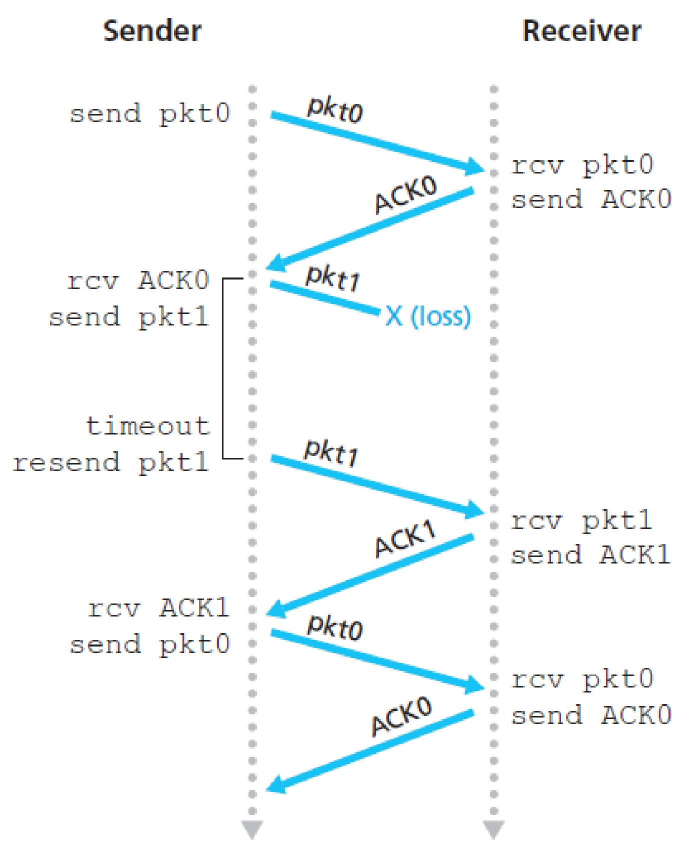
Always check the return value of these calls !!!!

A simple reliable protocol (1 of 2)

(a) no loss

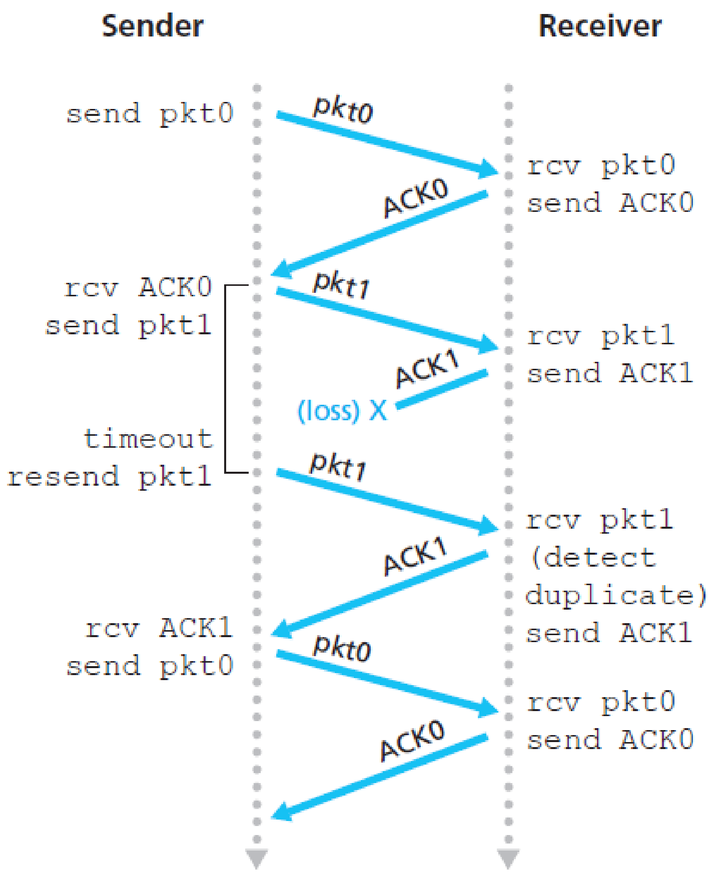


(b) packet loss

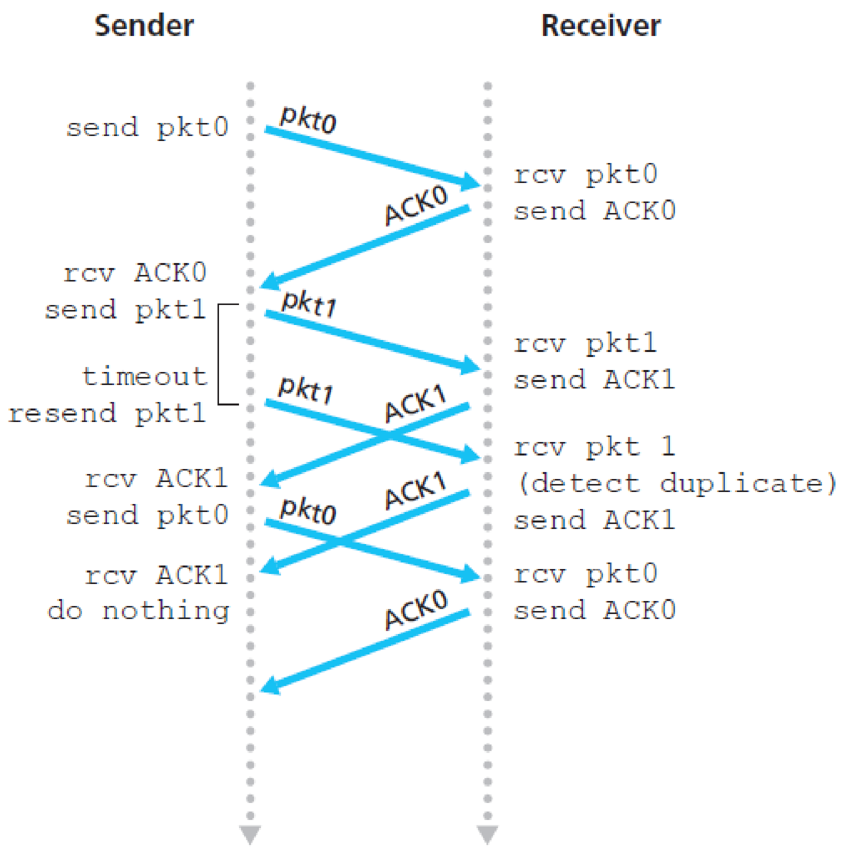


A simple reliable protocol (2 of 2)

(c) ACK loss



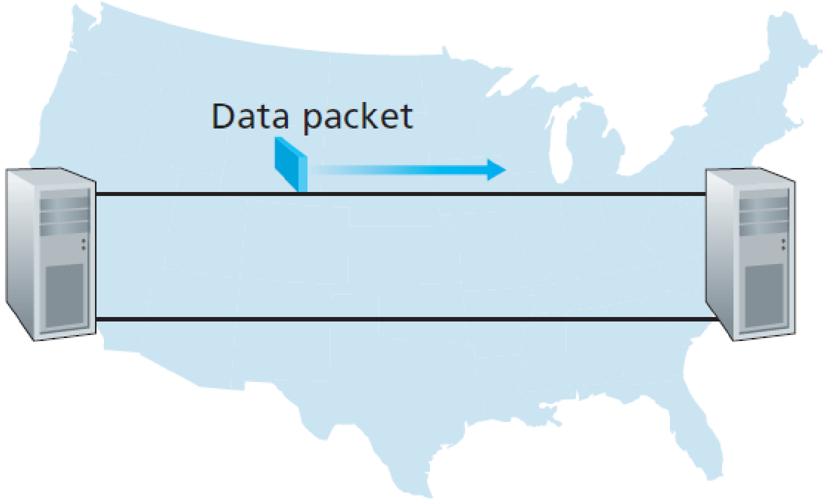
(d) premature timeout/ delayed ACK



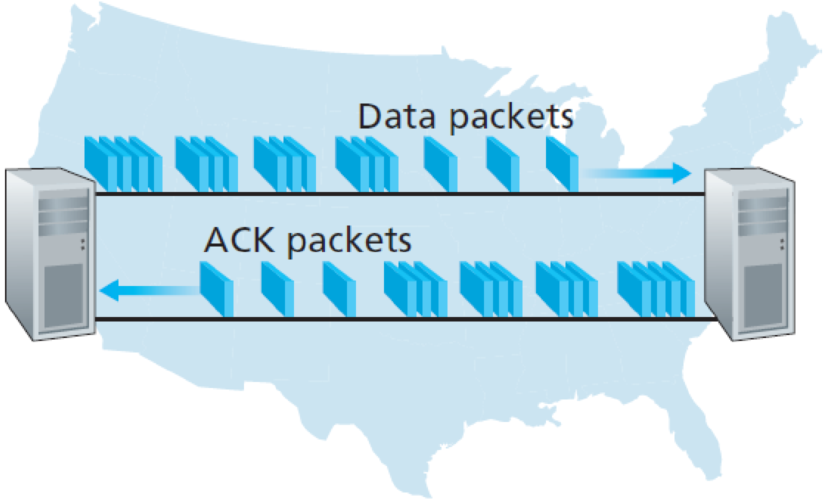
"The alternating bit protocol"

Pipelined Protocols

- **pipelining:** sender allows multiple, “in-flight”, yet-to-be acknowledged pkts
  – range of sequence numbers must be increased
  – buffering at sender and/or receiver



a. A stop-and-wait protocol in operation

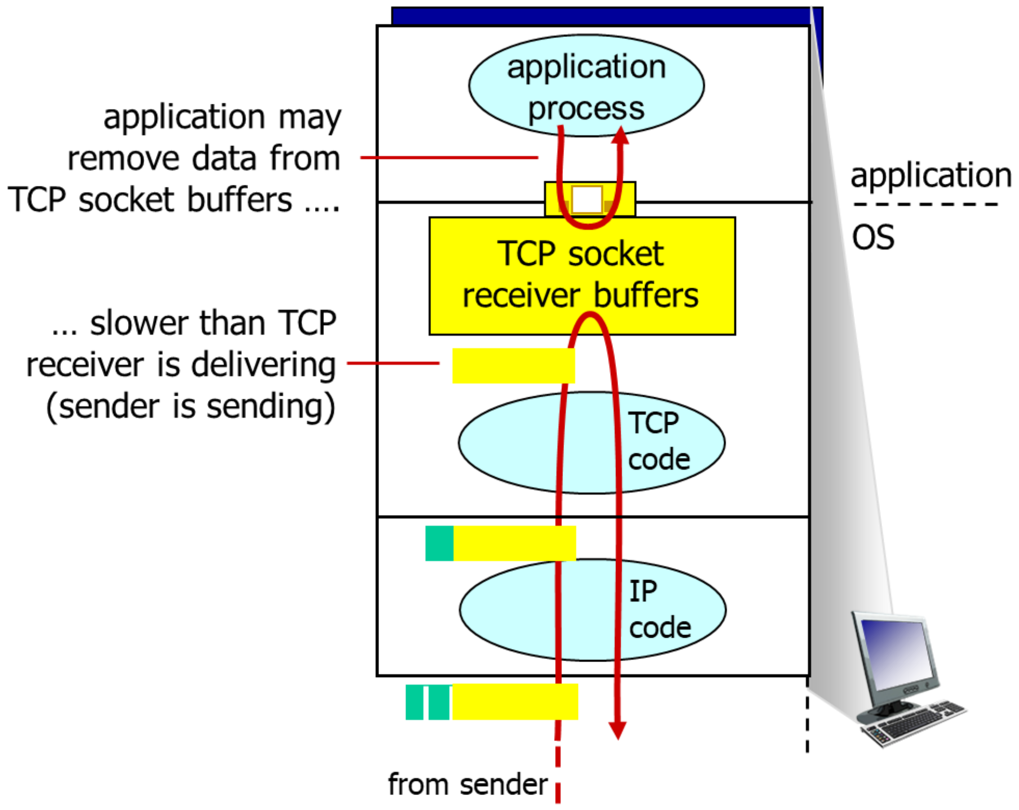


b. A pipelined protocol in operation

- two generic forms of pipelined protocols: **go-Back-N, selective repeat**

TCP Flow Control (1 of 2)

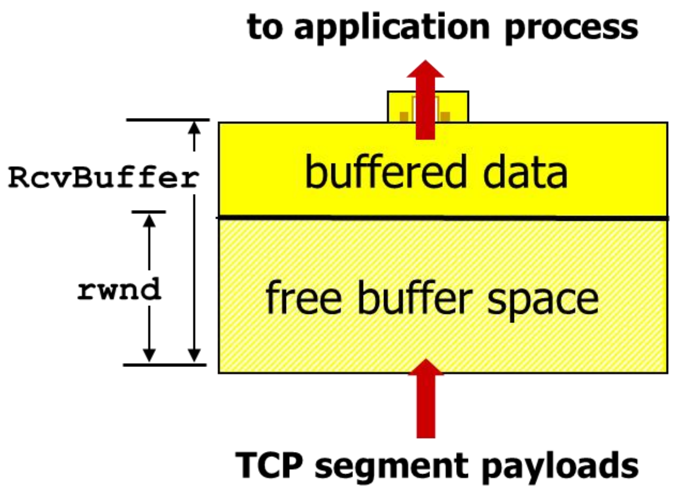
- flow control
- receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast



receiver protocol stack

TCP Flow Control (2 of 2)

- receiver "advertises" free buffer space by including "receive window" (rwnd) value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust RcvBuffer
- sender limits amount of unacked ("in-flight") data to receiver's rwnd value
- guarantees receive buffer will not overflow



receiver-side buffering

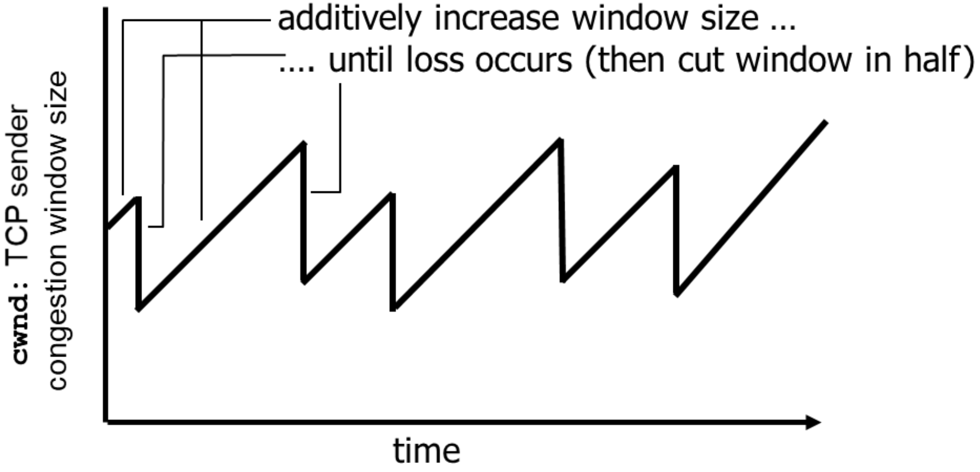
Principles of Congestion Control

- **congestion:**
  - informally: “too many sources sending too much data too fast for network to handle”
  - different from flow control!
  - manifestations:
    - lost packets (buffer overflow at routers)
    - long delays (queueing in router buffers)

TCP Congestion Control

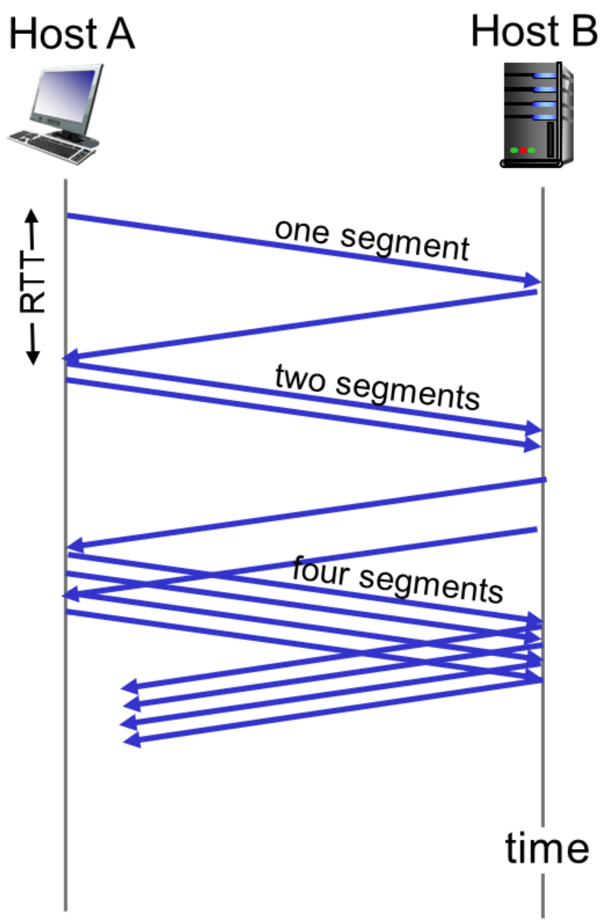
- **Additive Increase Multiplicative Decrease (AIMD)**
- **approach:** sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase***:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - **multiplicative decrease**: cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



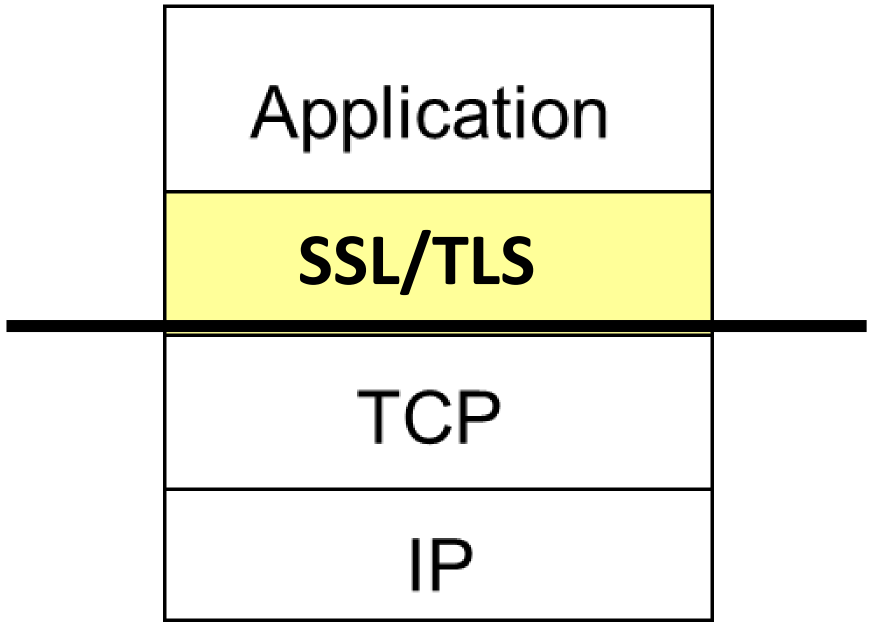
Congestion Control: TCP “Slow” Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- **summary:** initial rate is slow but ramps up exponentially fast
  - Slow-start is not slow



Securing TCP

- TCP & UDP: no encryption
- Secure Socket Layer
- First widely deployed internet security protocol
- mechanisms: [Woo 1994], implementation: Netscape
  - supported by almost all browsers, web servers
  - Netscape ~1993
  - https
  - billions $/year over SSL
- provides
  - **confidentiality**
  - **integrity**
  - **end-point authentication**
- **Now TLS: transport layer security**, RFC 2246
- **SSL/TLS is at app layer**
  - apps use TLS libraries, that “talk” to T C P

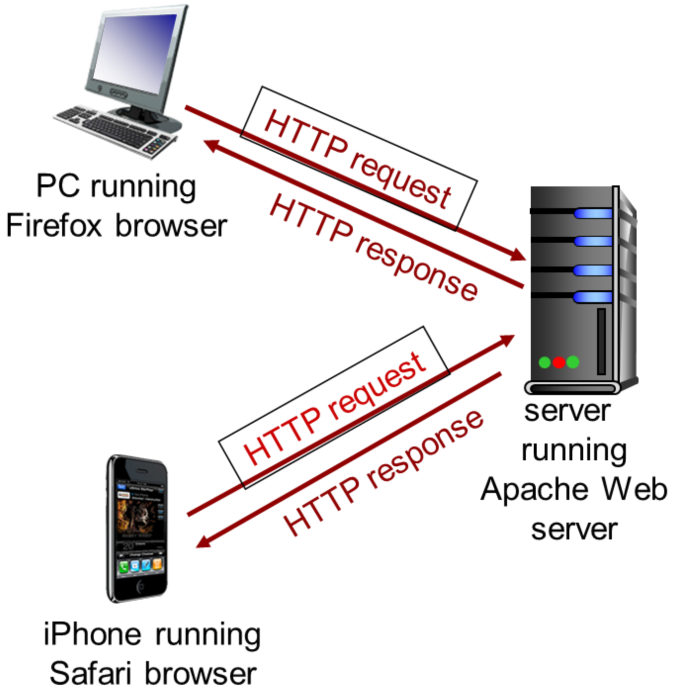


An application layer protocol

HTTP Overview https://tools.ietf.org/html/rfc7231

HTTP: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - **client:** browser that requests, receives, (using HT TP protocol) and "displays" Web objects
  - **server:** Web server sends (using HTTP protocol) objects in response to requests
- client initiates TCP connection (creates socket) to server, port 80



Server hosts a number of **resources** (identified through URI's)

- An HTTP request often causes the server to execute code that computes the resulting represenation
  - Dynamic web pages, DB access,
  - REST APIs

HTTP Messages

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

header lines

carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

Non-Persistent HTTP: Response Time

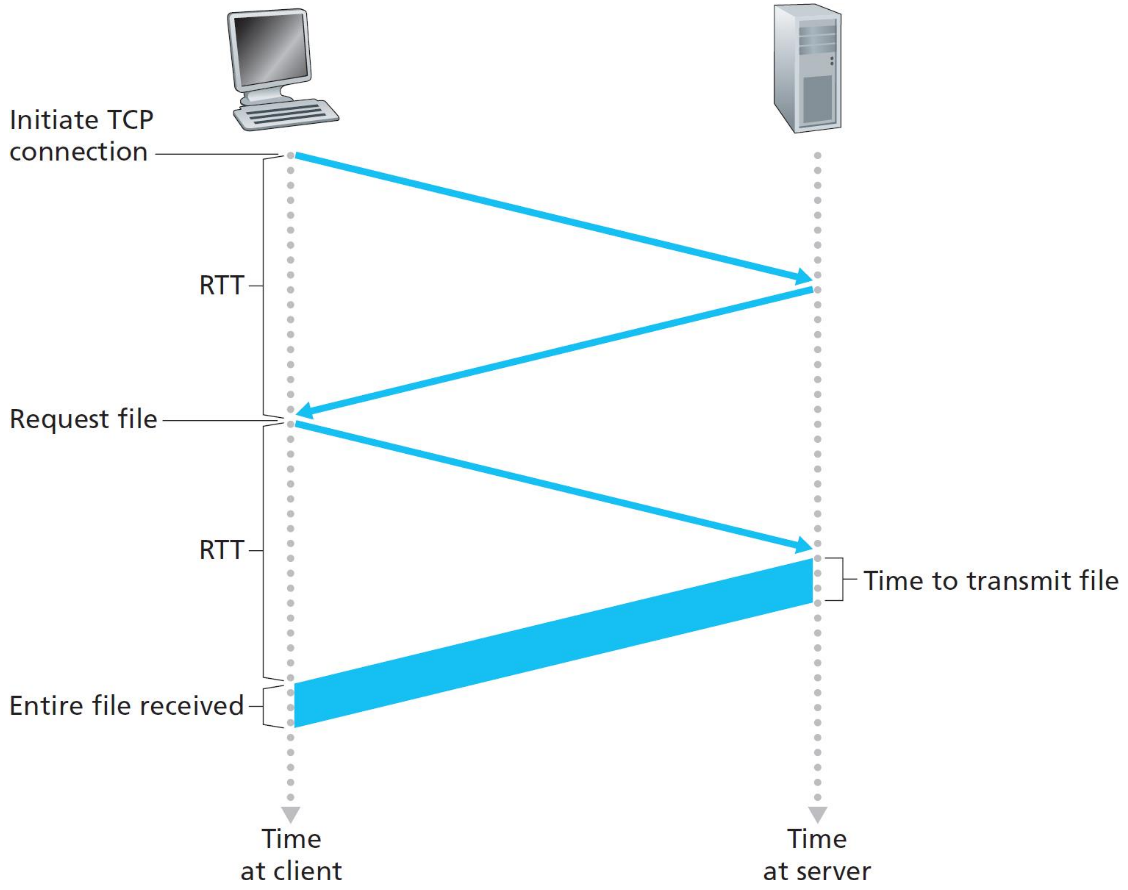
RTT (definition): time for a small packet to travel from client to server and back

HTTP response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time = 2RTT+ file transmission time
- If a page contains multiple objects:
  - Sequentially repeat this process: slow
  - Open multiple parallel connections (but browsers may limit the number, and is more demanding on server)

Persistent HTTP:

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- HTTP/2



Stateless Servers: HTTP Overview

HTTP is "stateless"

- server maintains no information about past client requests
- In general, maintaining state in stateful services/protocols can be problematic/complex
    - Scalability
    - If server/client crashes, their views of "state" may be inconsistent, must be reconciled
    - Eg. Service that maintains a mutex lock

A statefull protocol maintains state at sender/receiver over multiple transactions

A website/application **may maintain state using cookies carried in HTTP messages**

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

HTTP/1.1 Methods

- GET
- POST
  - Providing a block of data, such as the fields entered into an HTML form
  - Append (or create)
- PUT:
  - Replace (create) state of the target ressource
- DELETE
  - deletes association to the resource specified in the URL field
- CONNECT, OPTIONS, TRACE

GET, HEAD, OPTIONS, and TRACE methods should be safe

*An **method is idempotent** if the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request.*

- PUT, DELETE, and safe request (GET) methods are to be idempotent.
- invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

GET (Safe methods responses) are **cacheable**

POST Responses are only cacheable when explicitly permitted using cache control header fields.)

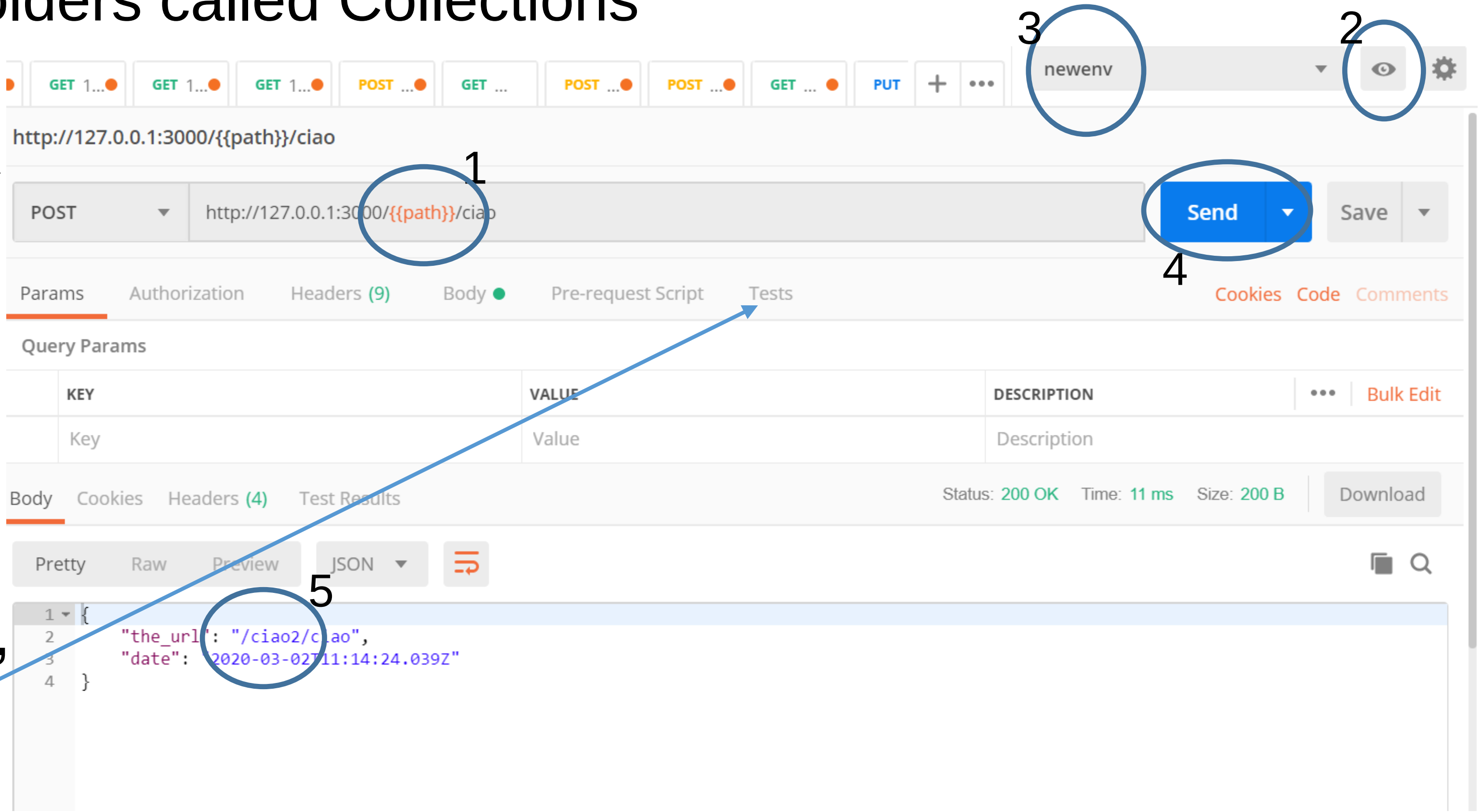
DELETE & PUT: invalidates cached items, Reponses not cacheable

YOUR CODE MUST RESPECT THESE

Useful tools

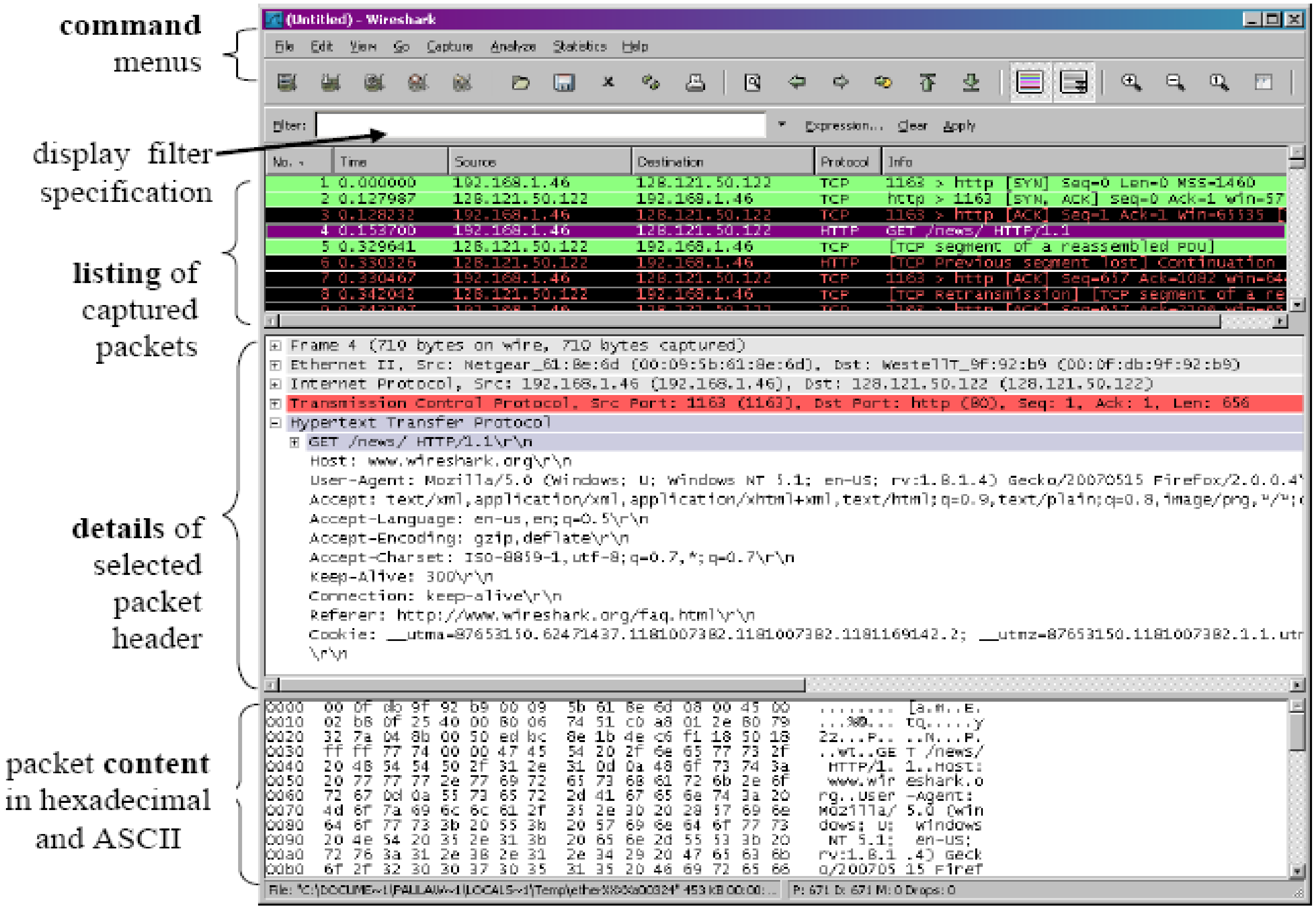
Postman

- Tool useful to test APIs
  - Set up GET/PUT/POST/DELETE/etc requests and see responses
  - Save requests to repeat later (History)
  - Organize requests into folders called Collections
- Parametrization of requests ➔ ➔ ➔ ➔ ➔
  1. Parametrize using {{..}}
  2. Set up an environment
  3. Apply it
  4. Execute the request
  5. See the result
- Possible to create “tests”
- Automate with CLI



Wireshark

- Program to intercept network communication and look into it
- Useful for example to verify if we are receiving data from a given server
- Do not forget to execute it as root / super-user



Wireshark vs TLS

General idea: ask chrome to dump the Pre_Master_Key (which is central to all TLS protocols)

Thus:

- Set up the environment variable SSLKEYLOGFILE
- Start wireshark to collect packets
  - Tell wireshark to use the key file (Preferences -> Protocols -> TLS)
- Restart chrome, and load a https web page
- Tell wireshark to “follow” a TLS session
- **REMOVE variable SSLKEYLOGFILE!!!**