Review

CS 4410

Operating Systems

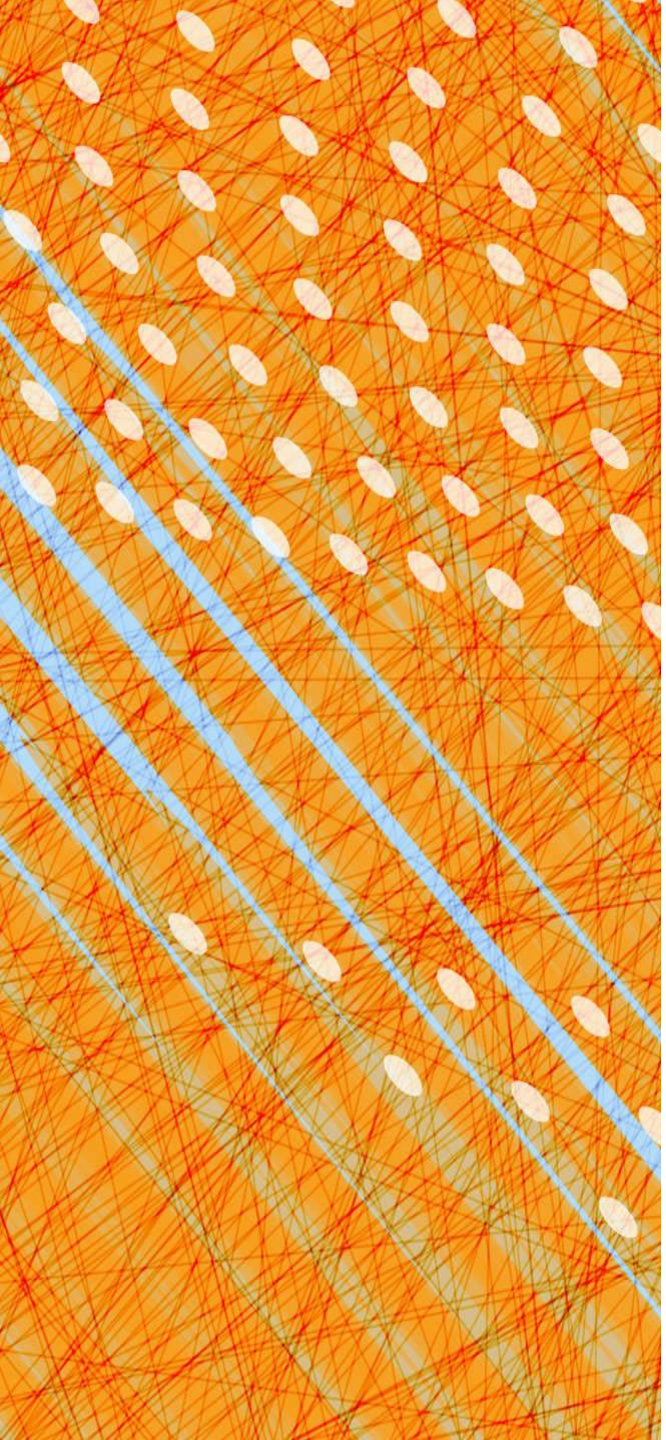
Summer 2019

Edward Tremel

Main OS Topics

- Architectural Support (HW/SW interface)
- Processes and Threads
- Scheduling
- Synchronization
- Virtual Memory
- Disks and Filesystems
- Networking

Cornell CIS



Architectural Support

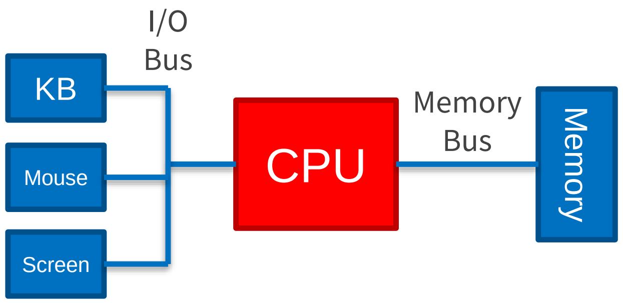
CORNELL UNIVERSITY FOUNDED A.D. 1865

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Device Interfacing Techniques

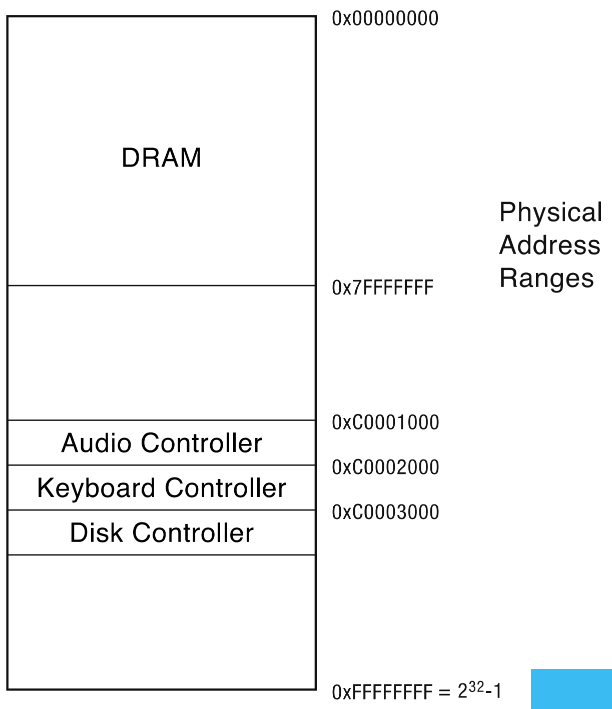
Programmed I/O

- CPU has dedicated, special instructions
- CPU has additional wires (I/O bus)
- Instruction specifies device and operation



Memory-mapped I/O

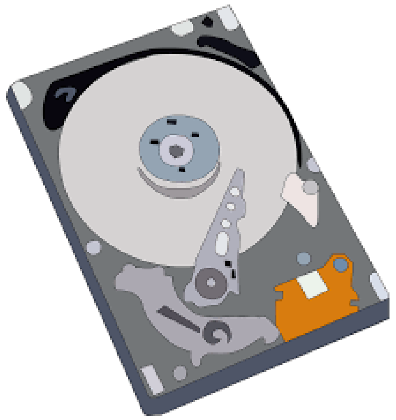
- Device communication goes over memory bus
- Reads/Writes to special addresses converted into I/O operations by dedicated device hardware
- Each device appears as if it is part of the memory address space
- **Predominant device interfacing technique**



I/O Summary

Interrupt-driven operation with memory-mapped I/O:
- CPU initiates device operation (*e.g.*, read from disk): writes an operation descriptor to a designated memory location
- CPU continues its regular computation
- The device asynchronously performs the operation
- When the operation is complete, interrupts the CPU

Bulk Data Transfers: Use **DMA**
- CPU sets up DMA request
- Device puts data on bus, RAM accepts it
- Device interrupts CPU when all done



Supporting dual mode operation

1. Privilege mode bit (0=kernel, 1=user)

   Where? x86 → EFLAGS reg., MIPS →status reg.

2. Privileged instructions

   user mode → no way to execute unsafe insns

3. Memory protection

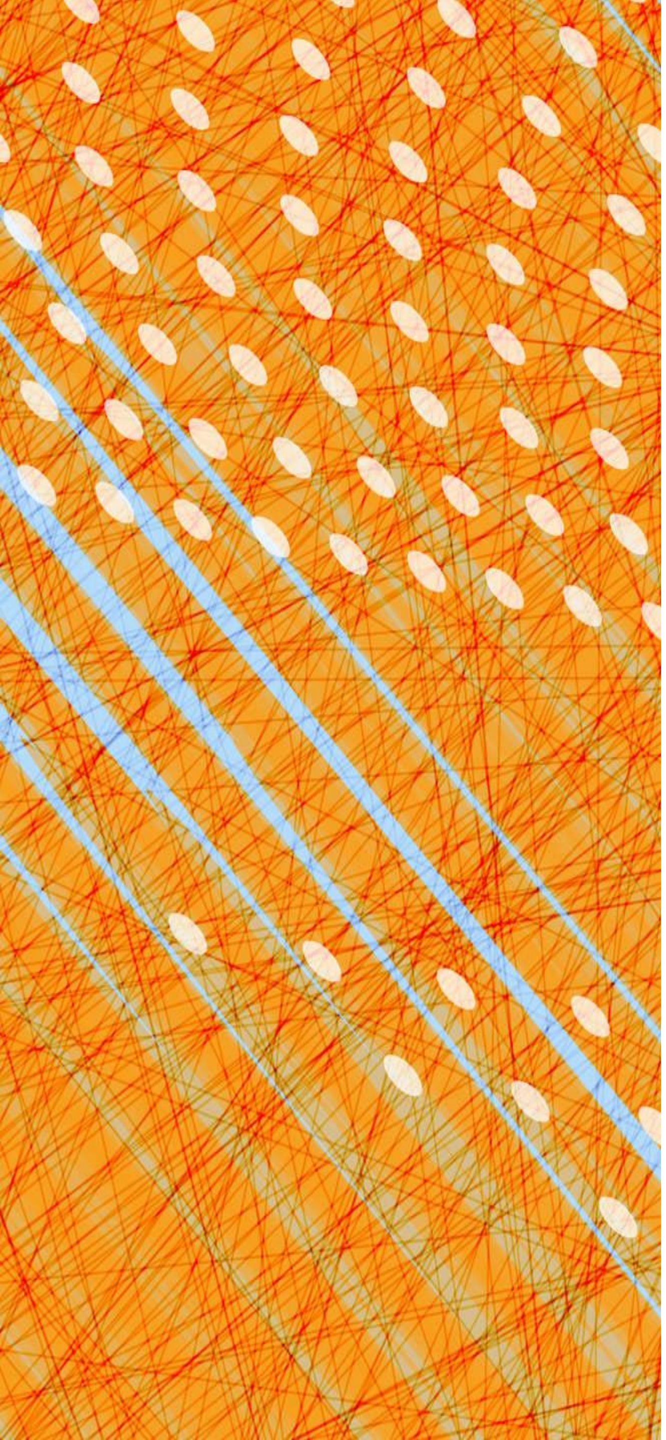
   user mode → memory accesses outside a process' memory region are prohibited

4. Timer interrupts

   kernel must be able to periodically regain control from running process

5. Efficient mechanism for switching modes

   must be fast because it happens a lot!



Processes and Threads

Process Control Block (PCB)

For each process, the OS has a PCB containing:

- location in memory
- location of executable on disk
- which user is executing this process
- process privilege level
- process identifier (`pid`)
- process arguments (for identification with `ps`)
- process status (Ready, waiting, finished, *etc.*)
- register values
- scheduling information
- PC, SP, eflags/status register

    ... and more!

Usually lives on the kernel stack

Creating and Managing Processes

`fork`	Create a child process as a clone of the current process. Returns to both parent and child. Returns child pid to parent process, 0 to child process.
`exec (prog, args)`	Run the application **prog** in the current process with the specified arguments.
`wait(pid)`	Pause until the child process has exited.
`exit`	Tell the kernel the current process is complete, and its data structures (stack, heap, code) should be garbage collected. Why not necessarily PCB?
`kill (pid, type)`	Send an interrupt of a specified type to a process. (a bit of a misnomer, no?)

[UNIX]

Signals (virtualized interrupt)

Allow applications to behave like operating systems.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctrl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGTSTP	Stop until next SIGCONT	Stop signal from terminal (e.g. ctrl-z from keyboard)

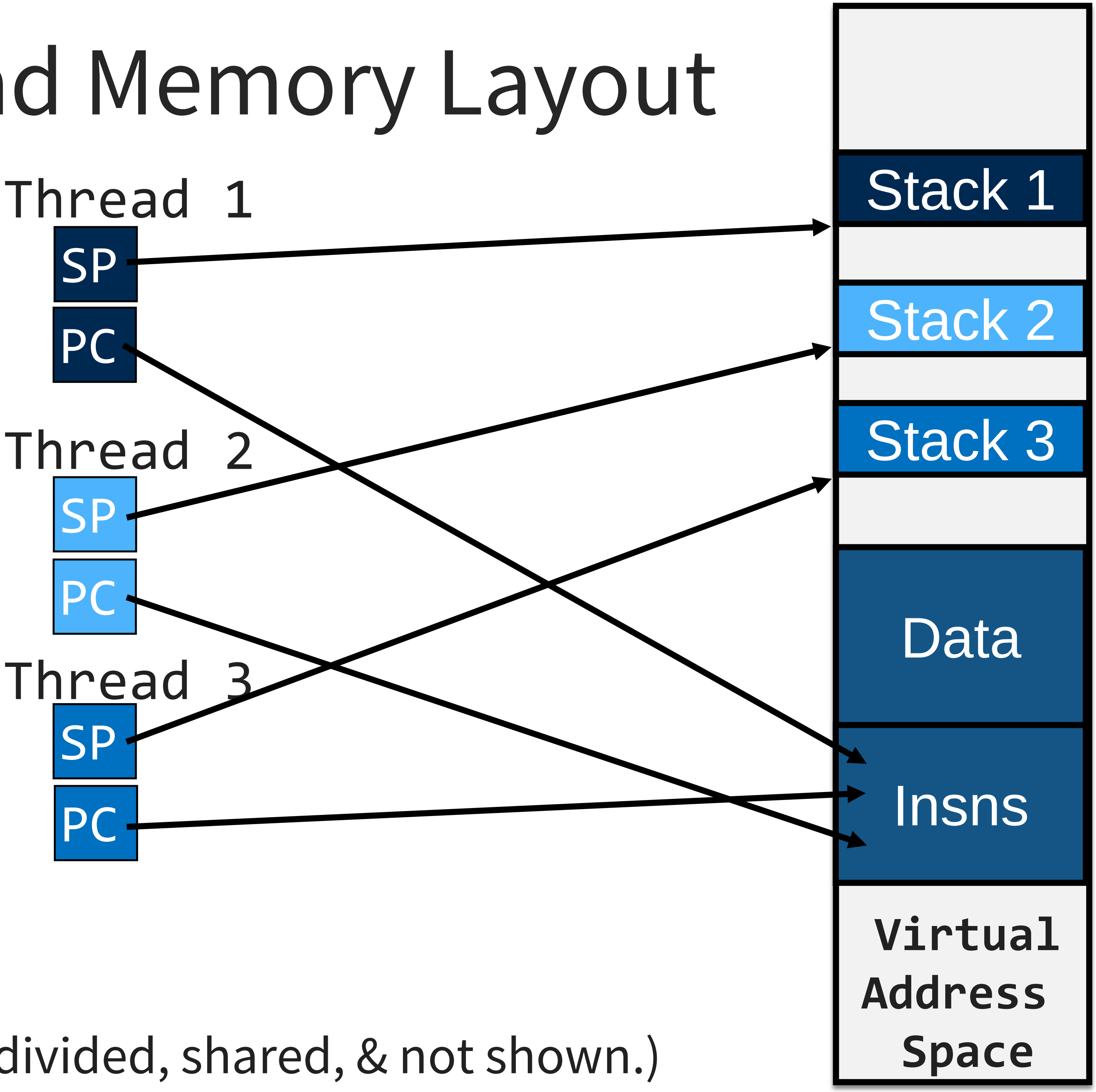
[UNIX]

Process vs. Thread

Process:

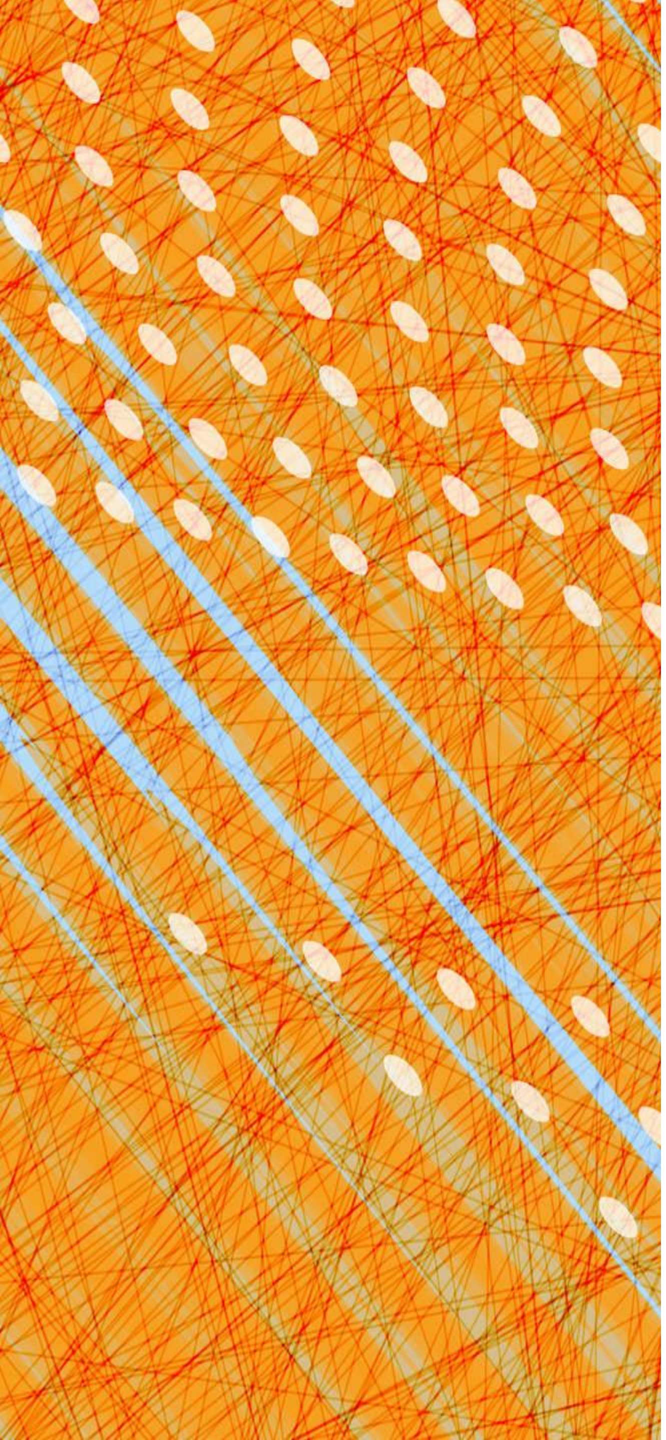
- Privilege Level
- Address Space
- Code, Data, Heap
- Shared I/O resources
- One or more Threads:
    - Stack
    - Registers
    - PC, SP

Thread Memory Layout



(Heap subdivided, shared, & not shown.)

Scheduling



Kernel Operation (conceptual, simplified)

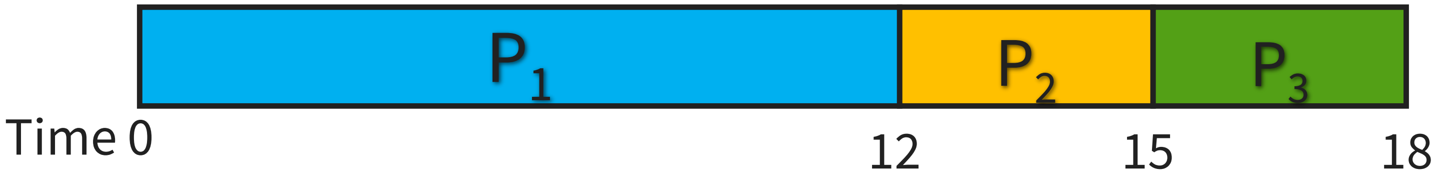
```
1.   Initialize devices
2.   Initialize "first process"
3.    while (TRUE) {
        • while device interrupts pending
          - handle device interrupts
        • while system calls pending
          - handle system calls
        • if run queue is non-empty
          - select process and switch to it
        • otherwise
          - wait for device interrupt
}
```

First In First Out (FIFO)

Processes P_1, P_2, P_3 with compute time 12, 3, 3

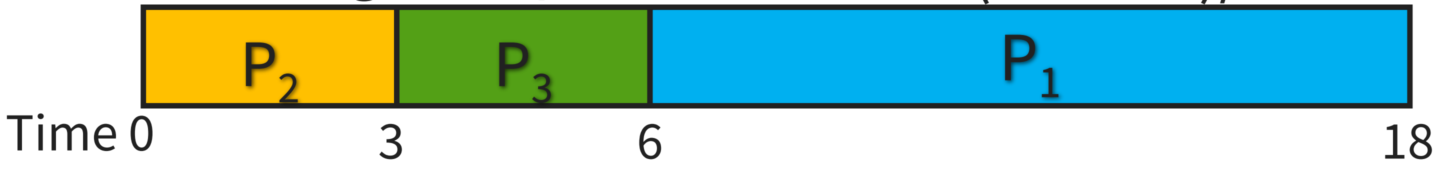
Scenario 1: arrival order P_1, P_2, P_3

Average Response Time: (12+15+18)/3 = 15



Scenario 2: arrival order P_2, P_3, P_1

Average Response Time: (3+6+18)/3 = 9



Note: this is always non-preemptive

FIFO Roundup

+ Simple
+ Low-overhead
+ No Starvation
+ Optimal avg. response time if all tasks same size

– Poor avg. response time if tasks have variable size

– Average response time very sensitive to arrival time

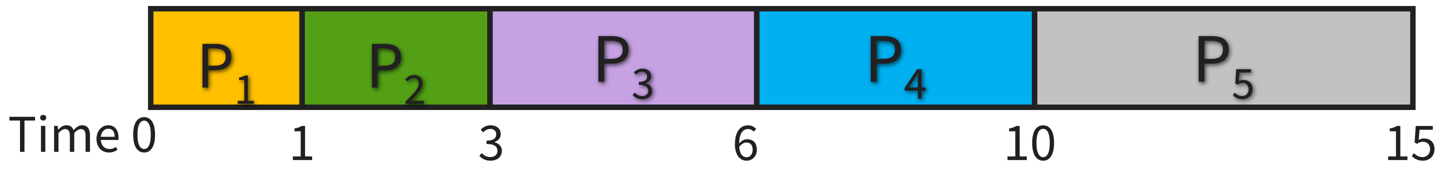
– Not responsive to interactive tasks

Shortest Job First (SJF)

Schedule in order of estimated completion[†] time

Scenario : each job takes as long as its number

Average Response Time: (1+3+6+10+15)/5 = 7



Would another schedule improve avg response time?

†with preemption, remaining time

SJF Roundup

+ Optimal average response time (when jobs available simultaneously)

– Pessimal variance in response time

– Needs estimate of execution time
– Can starve long jobs
– Frequent context switches

Round Robin (RR)

- Each process allowed to run for a quantum
- Context is switched (at the latest) at the end of the quantum

What is a good quantum size?

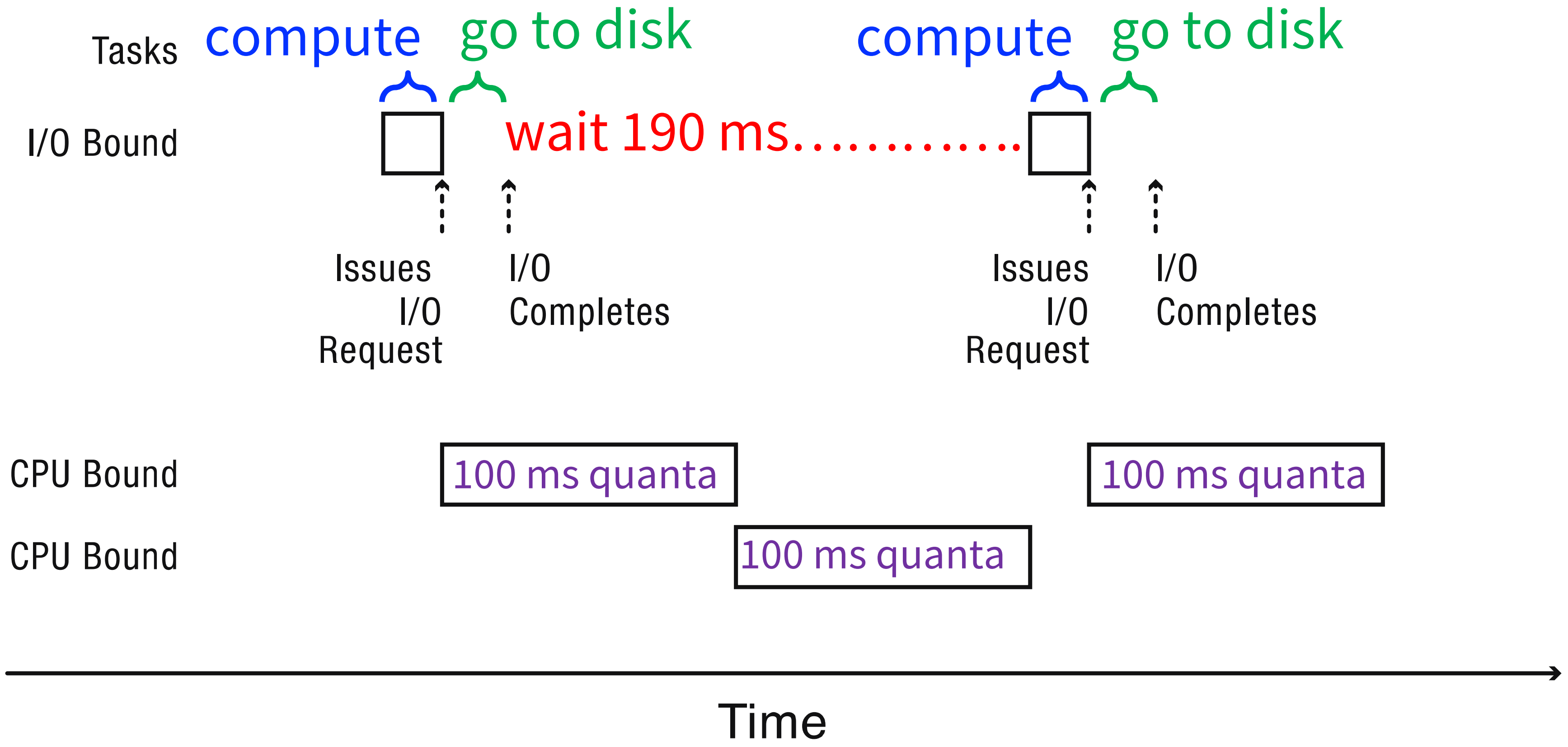
- Too long, and it morphs into FIFO
- Too short, and much time lost context switching
- Typical quantum: about 100X cost of context switch (~100ms vs. << 1 ms)

More Problems with Round Robin

Mixture of one I/O Bound tasks + two CPU Bound Tasks

I/O bound: compute, go to disk, repeat

→ *RR doesn't seem so fair after all….*



RR Roundup

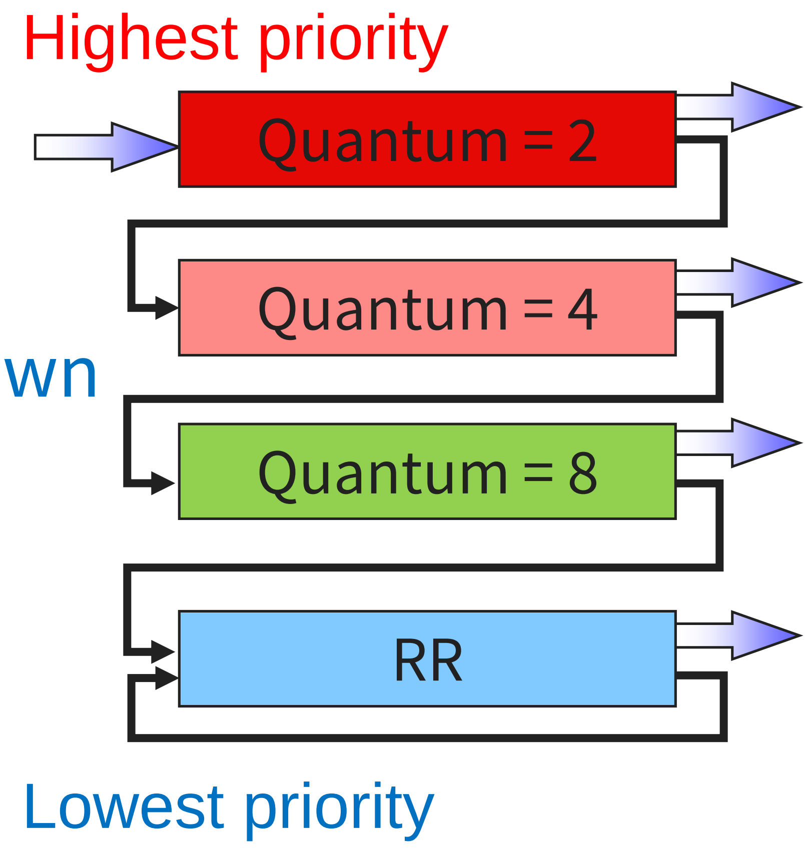
+ No starvation
+ Can reduce response time
+ Low Initial waiting time

– Overhead of context switching
– Mix of I/O and CPU bound

– Particularly bad for simultaneous, equal length jobs

Multi-Level Feedback Queues

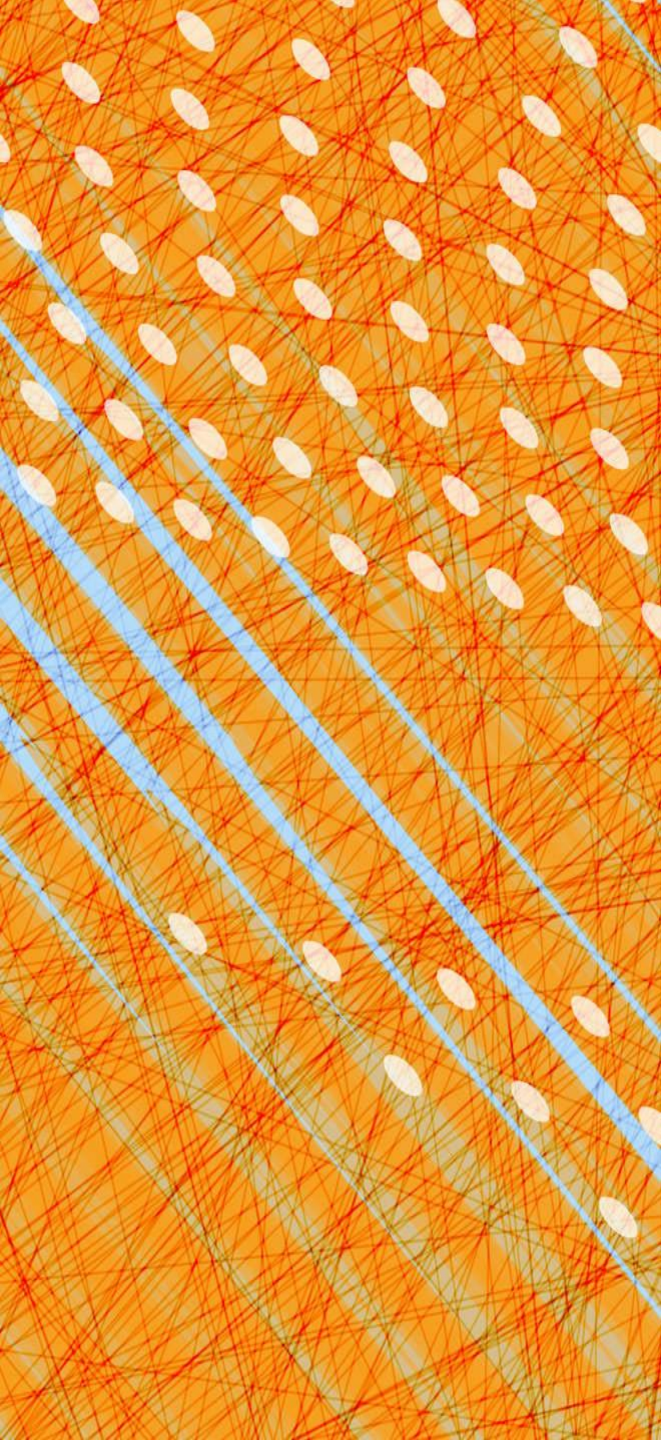
- Like multilevel queue, but assignments are not static
- Jobs start at the top
  - Use your quantum? move down
  - Don't? Stay where you are



Need parameters for:

- Number of queues
- Scheduling alg. per queue
- When to upgrade/downgrade job

Synchronization



[Dijkstra 1962]

What is a Semaphore?

Dijkstra introduced in the THE Operating System

Stateful:

- a **value** (incremented/decremented atomically)
- a queue
- a lock

Interface:

- Init(starting value)
- **P (procure)**: decrement, "consume" or "start using"
- **V (vacate)**: increment, "produce" or "stop using"

No operation to read the value!

Dutch 4410: P = Probeer ('Try'), V = Verhoog ('Increment', 'Increase by one')

Implementation of P and V

P():
- block (**sit on Q**) til n > `0`
- when so, decrement **value** by 1

V():
- increment **value** by 1
- **resume a thread waiting on Q (if any)**

Implementation requires:
- TAS spinlocks
- System calls for sleep and wake

```
P() {
    acquire(&guard);
    while(n <= 0) {
        waiting.enq(self);
        release(&guard);
        sleep();
        acquire(&guard);
    }
    n -= 1;
    release(&guard);
}
```

```
V() {
    acquire(&guard);
    n += 1;
    if(!waiting.empty()) {
       wake(waiting.deq());
    }
    release(&guard);
}
```

Semaphore's count:

- must be initialized!
- keeps state
  - reflects the sequence of past operations
  - `>0` reflects number of future P operations that will succeed

Not possible to:

- read the count
- grab multiple semaphores at same time
- decrement/increment by more than 1!

Producer-Consumer with Semaphores

```
Shared:
int buf[N];
int in = 0, out = 0;
Semaphore mutex_in(1), mutex_out(1);
Semaphore empty(N), filled(0);
```

```
void produce(int item)
{
   empty.P(); //need space
   mutex_in.P();
   buf[in] = item;
   in = (in+1)%N;
   mutex_in.V();
   filled.V(); //new item!
}
```

```
int consume()
{
   filled.P(); //need item
   mutex_out.P();
   int item = buf[out];
   out = (out+1)%N;
   mutex_out.V();
   empty.V(); //more space!
   return item;
}
```

Condition Variables

A mechanism to wait for events

3 operations on `Condition Variable x`

- `x.wait()`: sleep until woken up (could wake up on your own)
- `x.signal()`: wake at least one process waiting on condition (if there is one). No history associated with signal.
- `x.broadcast()`: wake all processes waiting on condition

!! NOT the same thing as UNIX wait & signal !!

Using Condition Variables

You must hold the monitor lock to call these operations.

To wait for some condition:

```
    while not some_predicate():
        CV.wait()
```

- atomically releases monitor lock & yields processor
- as CV.wait() returns, lock automatically reacquired

When the condition becomes satisfied:

`CV.broadcast()`: wakes up all threads

`CV.signal()`: wakes up at least one thread

Kid and Cook Threads

Running

```
Monitor BurgerKing {
  Lock mlock

  int numburgers = 0
  condition hungrykid

  kid_eat:
   with mlock:
     while (numburgers==0)
          hungrykid.wait()
     numburgers -= 1

  makeburger:
   with mlock:
     ++numburger
     hungrykid.signal()
}
```

```
kid_main() {

   play_w_legos()
   BK.kid_eat()
   bathe()
   make_robots()
   BK.kid_eat()
   facetime_Edward()
   facetime_grandma()
   BK.kid_eat()
}
```

```
cook_main() {

  wake()
  shower()
  drive_to_work()
  while(not_5pm)
    BK.makeburger()
  drive_to_home()
  watch_got()
  sleep()
}
```

Readers and Writers

```
Monitor ReadersNWriters {

 int waitingWriters=0, waitingReaders=0, nReaders=0, nWriters=0;
 Condition canRead, canWrite;

BeginWrite()
  with monitor.lock:
    ++waitingWriters
    while (nWriters >0 or nReaders >0)
      canWrite.wait();
    --waitingWriters
    nWriters = 1;

EndWrite()
  with monitor.lock:
    nWriters = 0
    if WaitingWriters > 0
      canWrite.signal();
    else if waitingReaders > 0
      canRead.broadcast();
}
```

```
void BeginRead()
  with monitor.lock:
    ++waitingReaders
    while (nWriters>0 or waitingWriters>0)
      canRead.wait();
    --waitingReaders
    ++nReaders


void EndRead()
  with monitor.lock:
    --nReaders;
    if (nReaders==0 and waitingWriters>0)
      canWrite.signal();
```

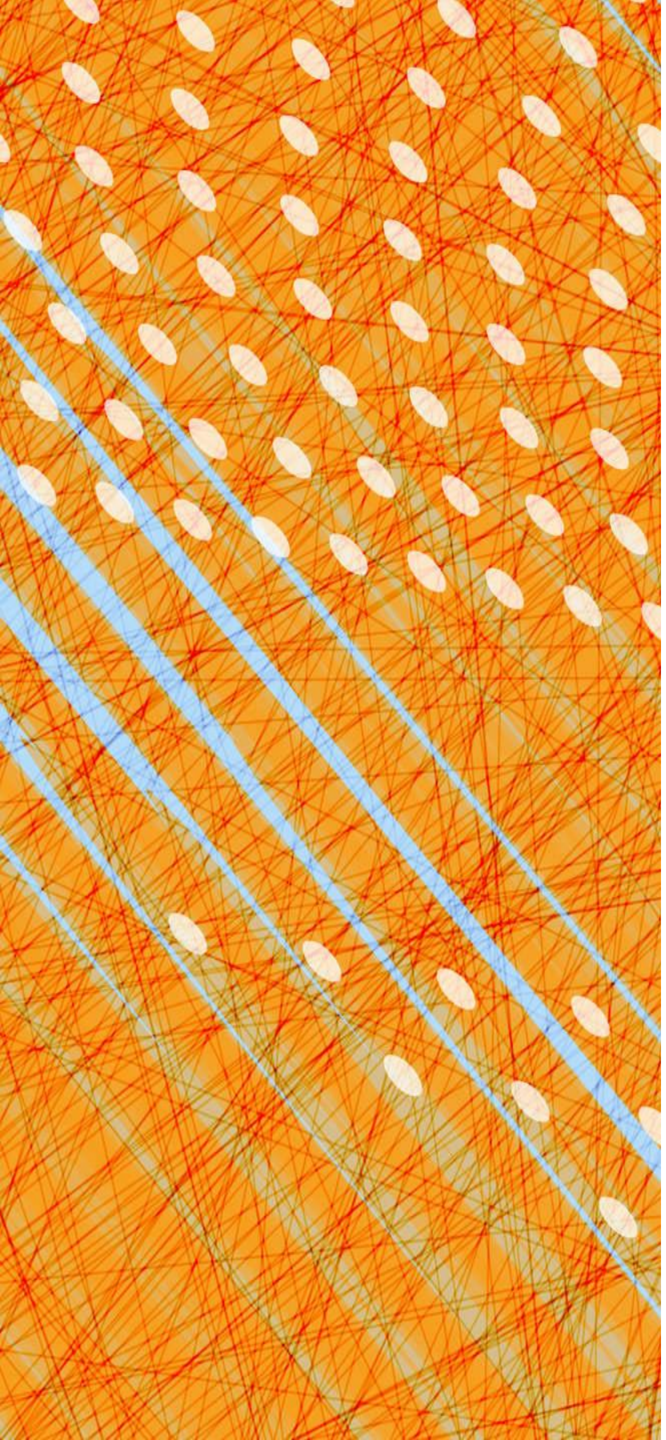
Checkin with 2 condition variables

```
self.allCheckedIn = Condition(self.lock)
self.allLeaving = Condition(self.lock)

def checkin():
  with self.lock:
    nArrived++
    if nArrived < nThreads:          // not everyone has checked in
      while nArrived < nThreads:
        allCheckedIn.wait()            // wait for everyone to check in
    else:
      nLeaving = 0                   // this thread is the last to arrive
      allCheckedIn.broadcast()       // tell everyone we're all here!
    nLeaving++
    if nLeaving < nThreads:                // not everyone has left yet
      while nLeaving < nThreads:
        allLeaving.wait()               // wait for everyone to leave
    else:
      nArrived = 0                   // this thread is the last to leave
      allLeaving.broadcast()         // tell everyone we're outta here!
```

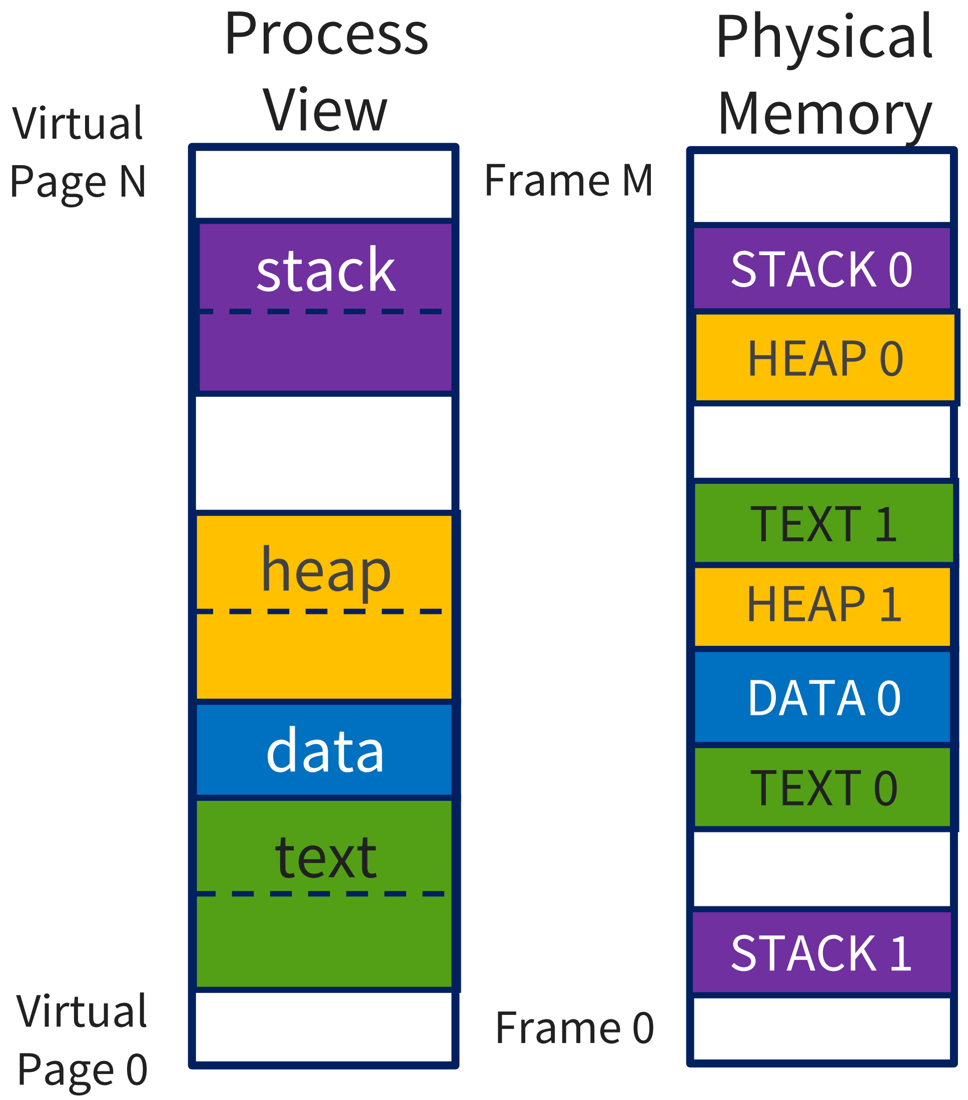
Implementing barriers is not easy.
Solution here uses a "double-turnstile"

Virtual Memory



Paged Translation

TERMINOLOGY ALERT:
Page: the data itself
Frame: physical location



No more external fragmentation!

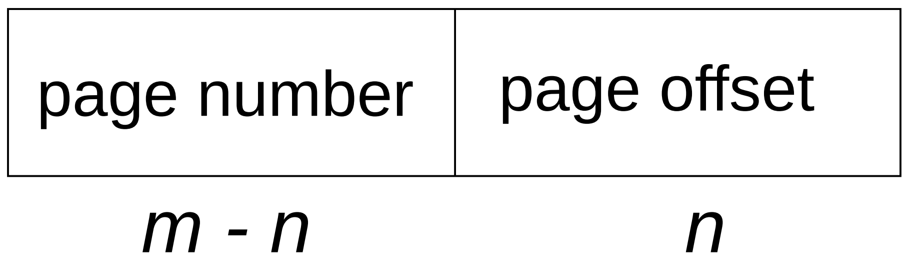
Logical Address Components

Page number – Upper bits

- Must be translated into a physical frame number

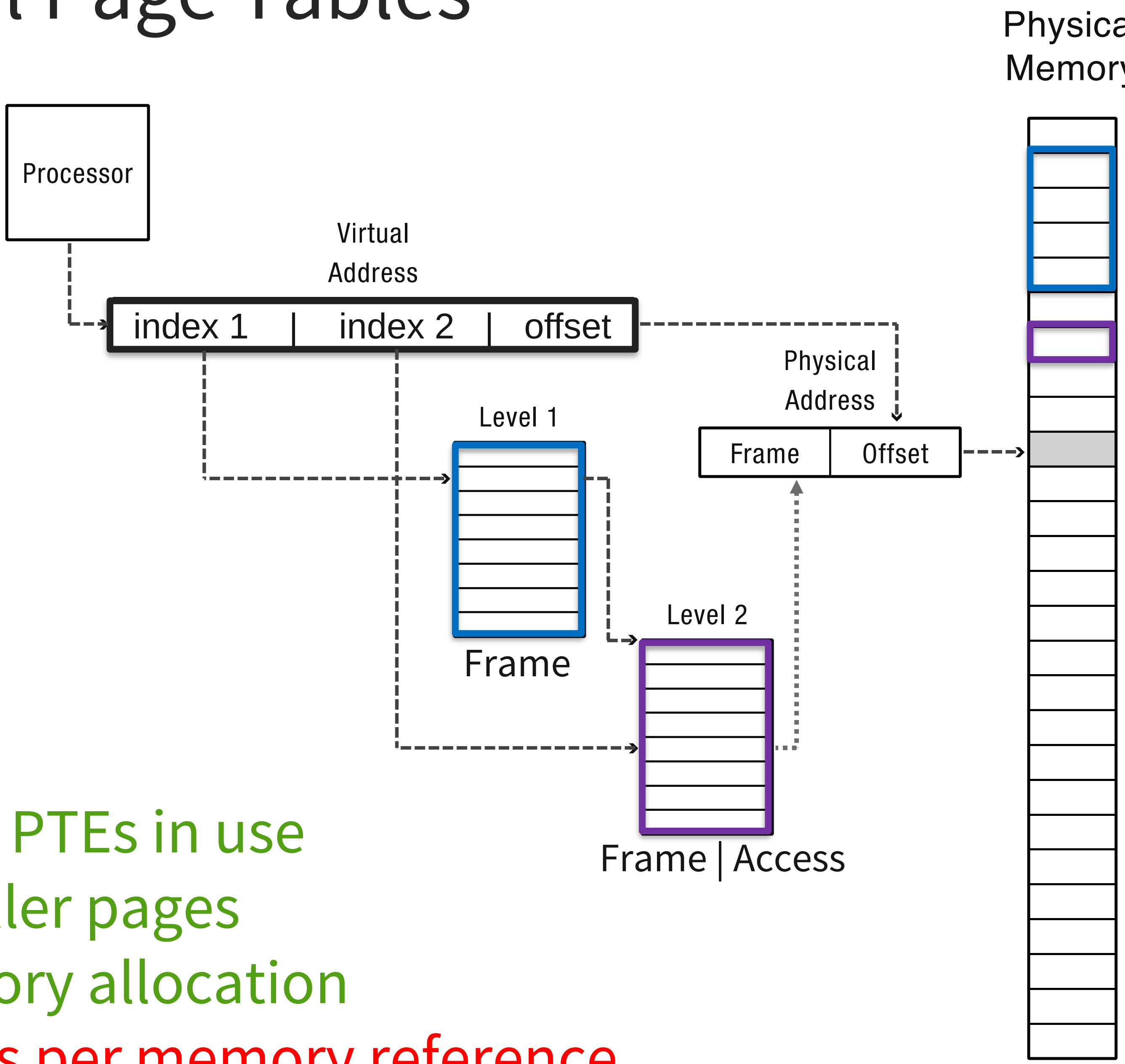
Page offset – Lower bits

- Does not change in translation



For given logical address space 2^m and page size 2^n

Multi-Level Page Tables

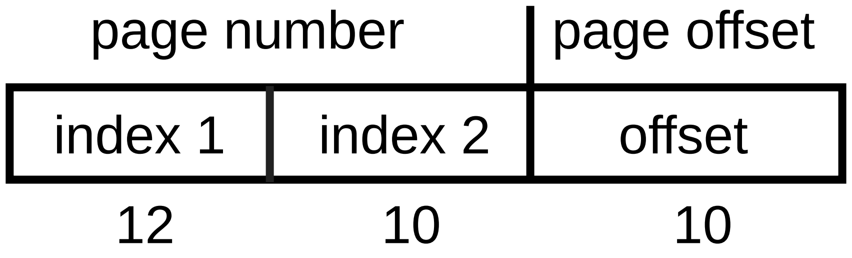


+ Allocate only PTEs in use
+ Can use smaller pages
+ Simple memory allocation
− ***more*** lookups per memory reference

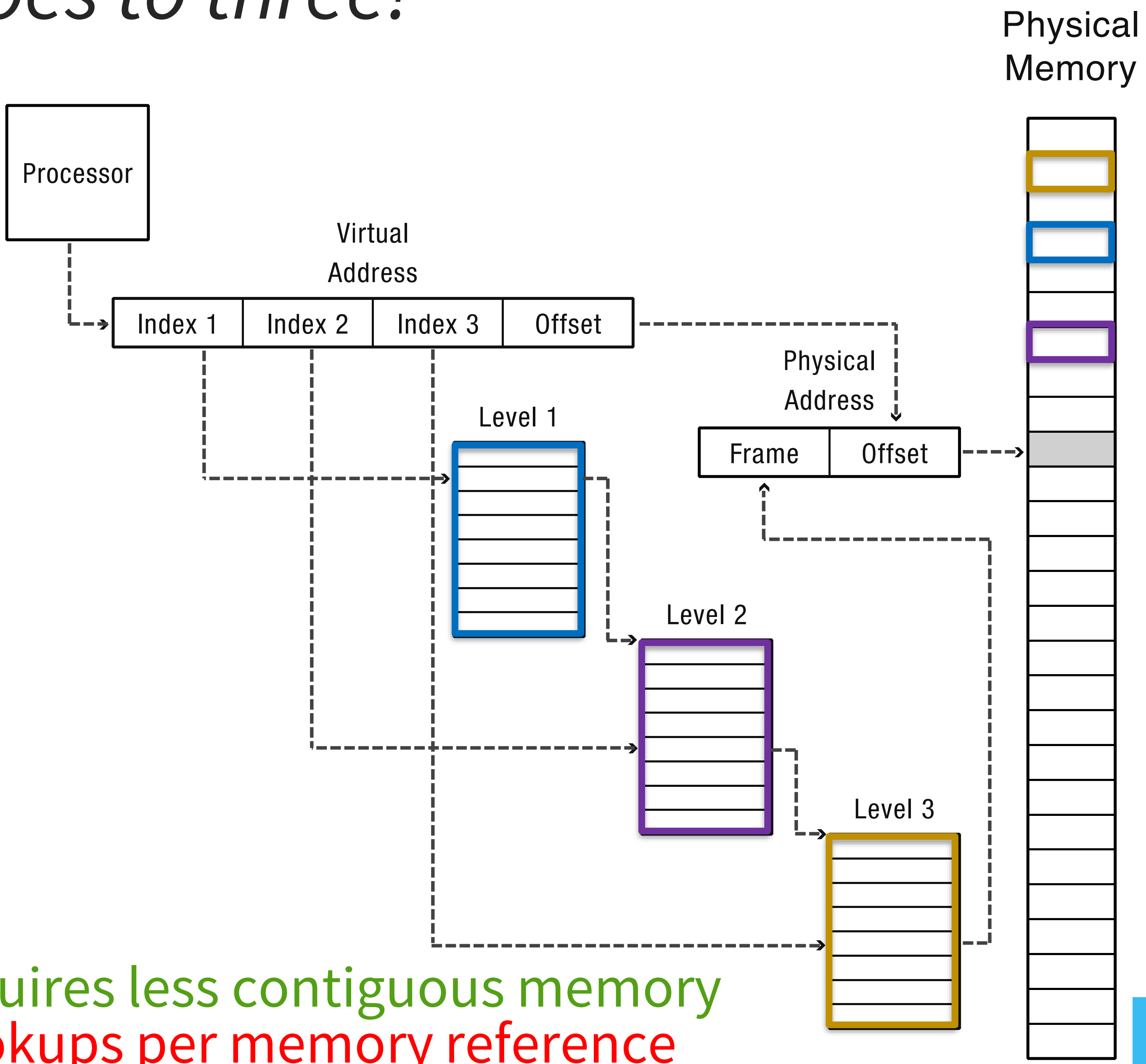
Two-Level Paging Example

32-bit machine, 1KB page size

- Logical address is divided into:
  - a page offset of 10 bits (1024 = 2^10)
  - a page number of 22 bits (32-10)
- Since the page table is paged, the page number is further divided into:
  - a 12-bit first index
  - a 10-bit second index
- Thus, a logical address is as follows:



This one goes to three!



+ First Level requires less contiguous memory

− ***even more*** lookups per memory reference

Complete Page Table Entry (PTE)

Valid	Protection R/W/X	Ref	Dirty	Index

Index is an index into:

- table of memory frames (if bottom level)
- table of page table frames (if multilevel page table)
- backing store (if page was swapped out)

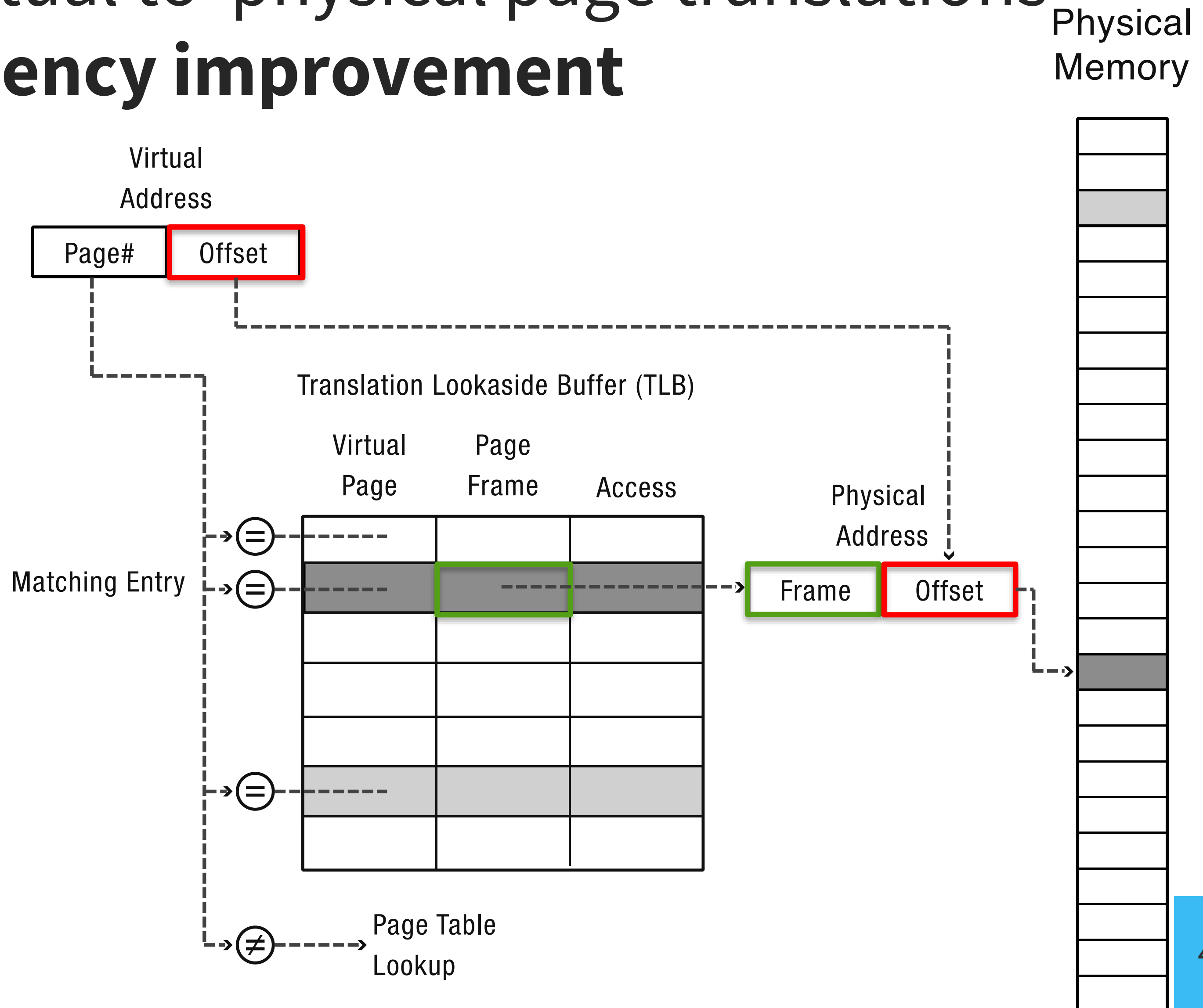
Synonyms:

- Valid bit == Present bit
- Dirty bit == Modified bit
- Referenced bit == Accessed bit

Translation Lookaside Buffer (TLB)

Cache of virtual to physical page translations

Major efficiency improvement



(the contents of) **A Virtual Page Can Be**

Mapped

- to a physical frame

Not Mapped (→ Page Fault)

- in a physical frame, but not currently mapped
- still in the original program file
- zero-filled (heap/BSS, stack)
- on backing store ("paged or swapped out")
- illegal: not part of a segment
  → Segmentation Fault

When a page needs to be brought in…

- Find a free frame
  - or evict one from memory (next slide)
  - which one? (next lecture)
- Issue disk request to fetch data for page
  - what to fetch? (requested page or more?)
- Block current process
- Context switch to new process
- When disk completes, set valid bit to 1 (& other permission bits), put current process in ready queue

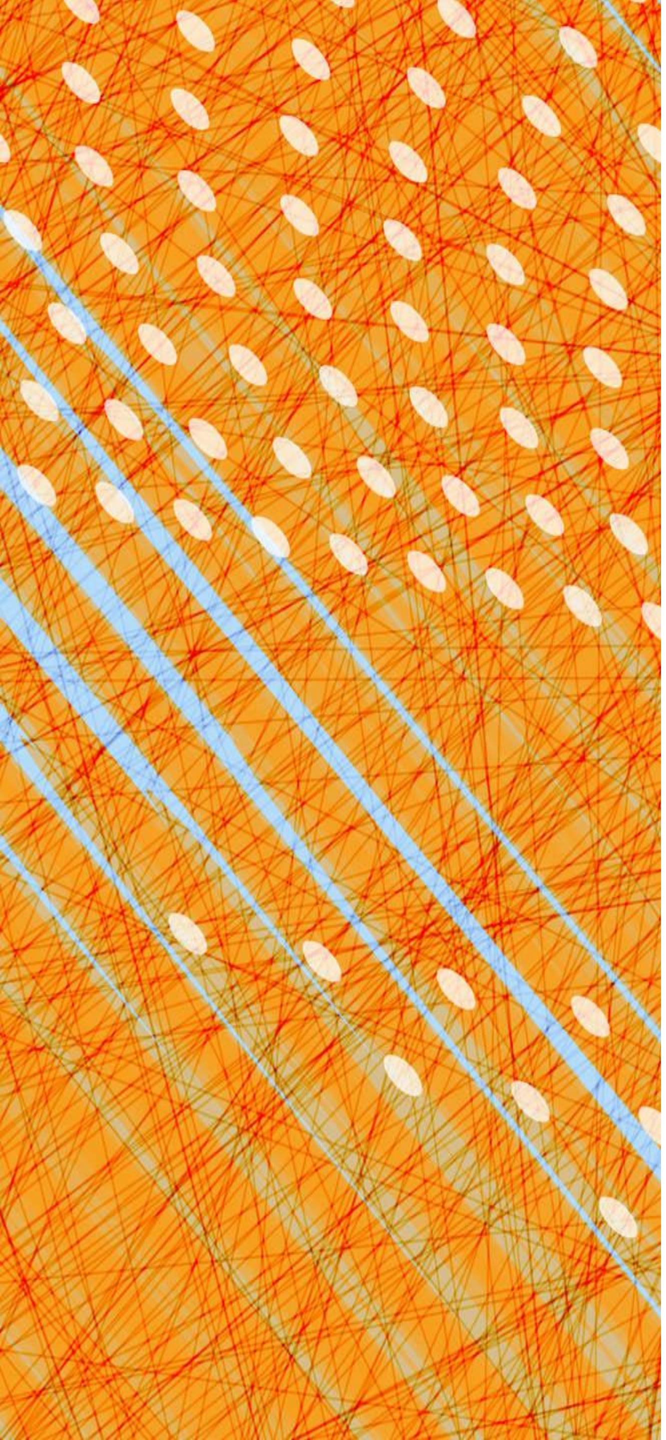
When a page is swapped out…

- Find all page table entries that refer to old page
  - Frame might be shared
  - Core Map (frames → pages)
- Set each page table entry to invalid
- Remove any TLB entries
  - Hardware copies of now invalid PTE
  - “TLB Shootdown”
- Write changes on page back to disk, if needed
  - Dirty/Modified bit in PTE indicates need
  - Text segments are (still) on program image on disk

Valid	Protection R/W/X	Ref	Dirty	Index

Page Replacement Algorithms

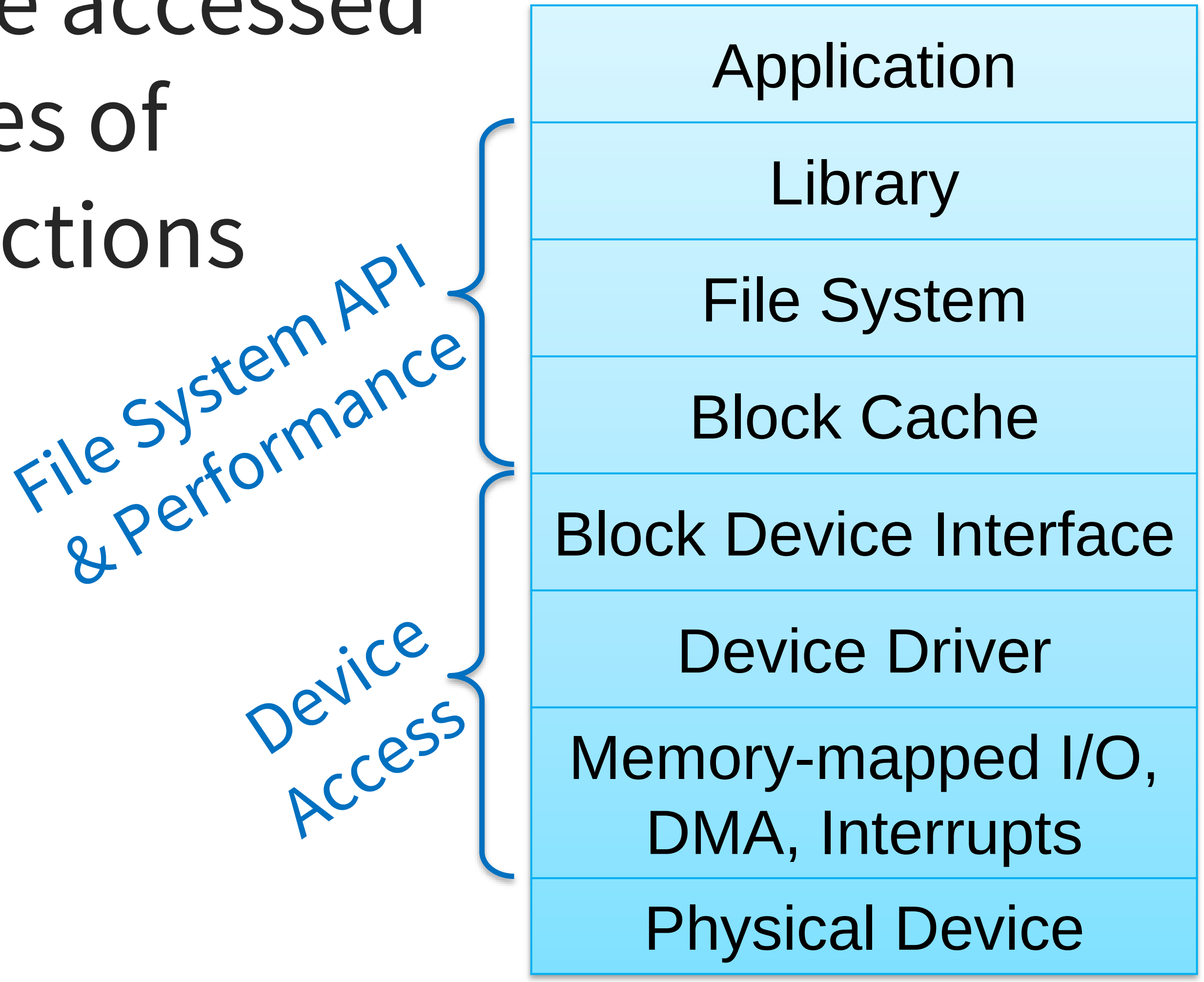
- **Random:** Pick any page to eject at random
  - Used mainly for comparison
- **FIFO:** The page brought in earliest is evicted
  - Ignores usage
- **OPT:** Belady's algorithm
  - Select page not used for longest time
- **LRU:** Evict page that hasn't been used for the longest
  - Past could be a good predictor of the future
- **MRU:** Evict the most recently used page
- **LFU:** Evict least frequently used page



Filesystems

The abstraction stack

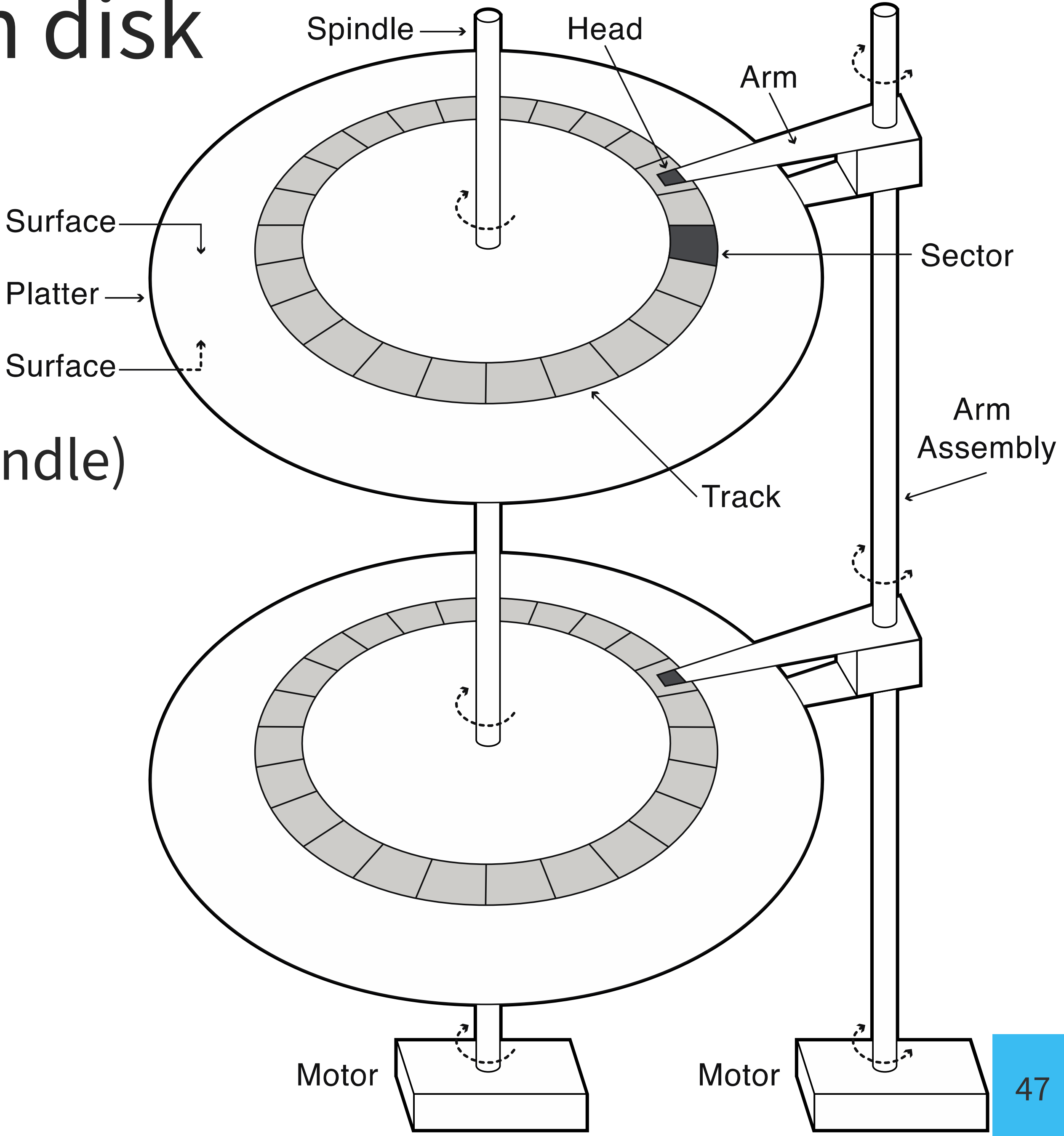
I/O systems are accessed through a series of layered abstractions



Reading from disk

Must specify:

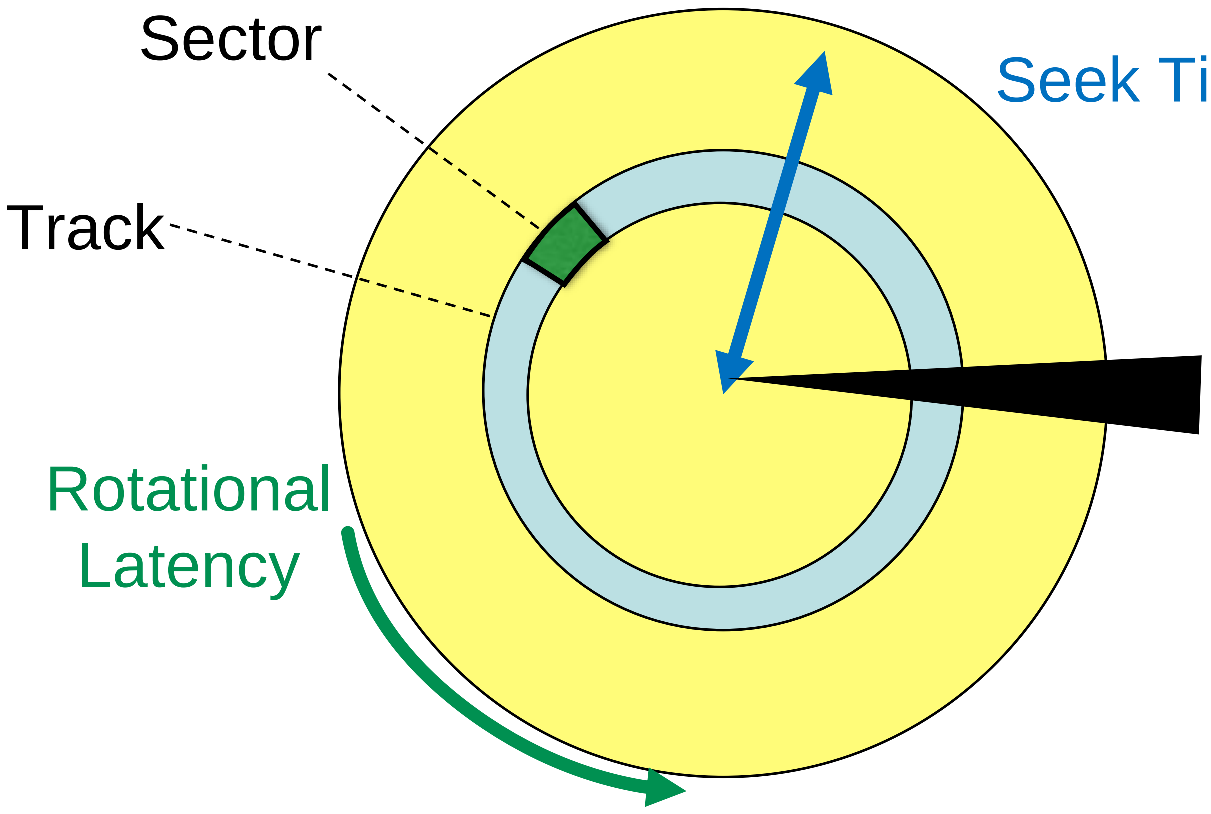
- cylinder # (distance from spindle)
- head #
- sector #
- transfer size
- memory address



Disk overheads

*Disk Latency = **Seek Time** + **Rotation Time** + Transfer Time*

- **Seek:** to get to the track (5-15 millisecs (ms))
- **Rotational Latency:** to get to the sector (4-8 millisecs (ms))
  (on average, only need to wait half a rotation)
- **Transfer:** get bits off the disk (25-50 microsecs (μs)

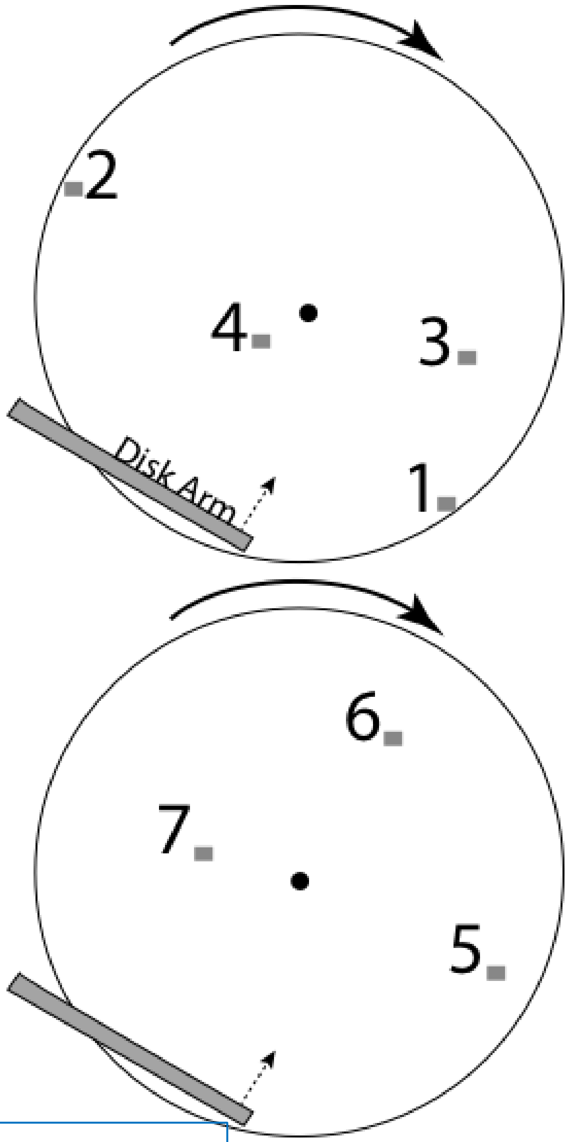


Disk Scheduling: C-SCAN

Circular list treatment:

- head moves from one end to other
- servicing requests as it goes
- reaches the end, returns to beginning
- no requests serviced on return trip

+ More uniform wait time than SCAN



C-SCAN Schedule?

Total Head movement?

Head pointer @ 53

Queue: 98, 183, 37, 122, 14, 124, 65, 67

Implementation Basics

Directories

- file name → file number

Index structures

- file number → block

Free space maps

- find a free block; better: find a free block *nearby*

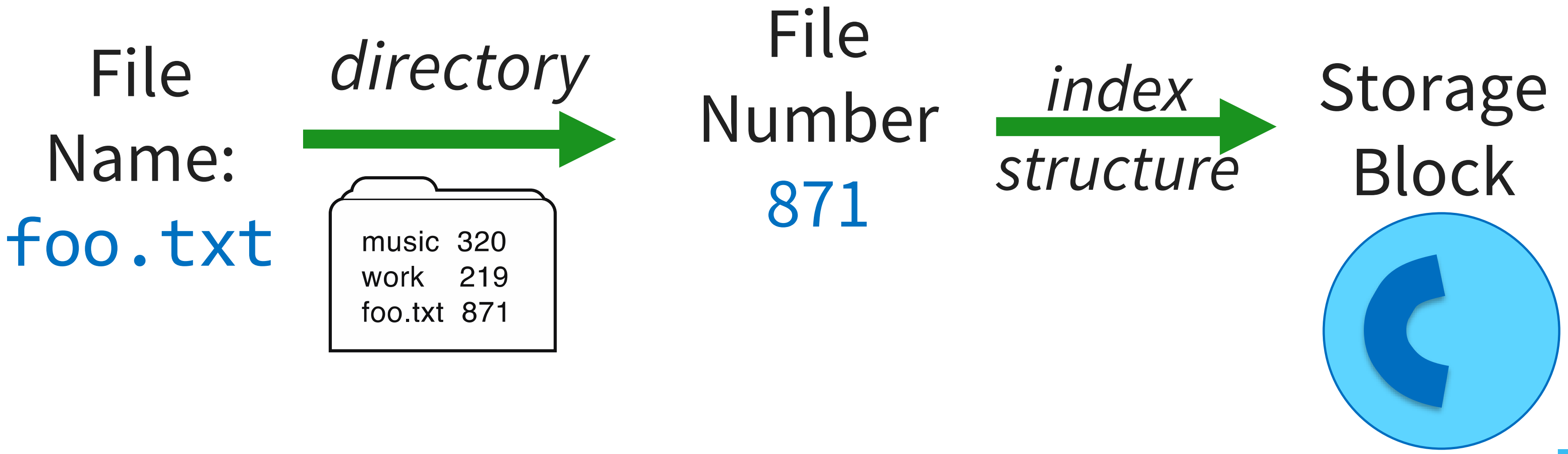
Locality heuristics

- policies enabled by above mechanisms
    - group directories
    - make writes sequential
    - defragment

Directory

Directory: provides names for files

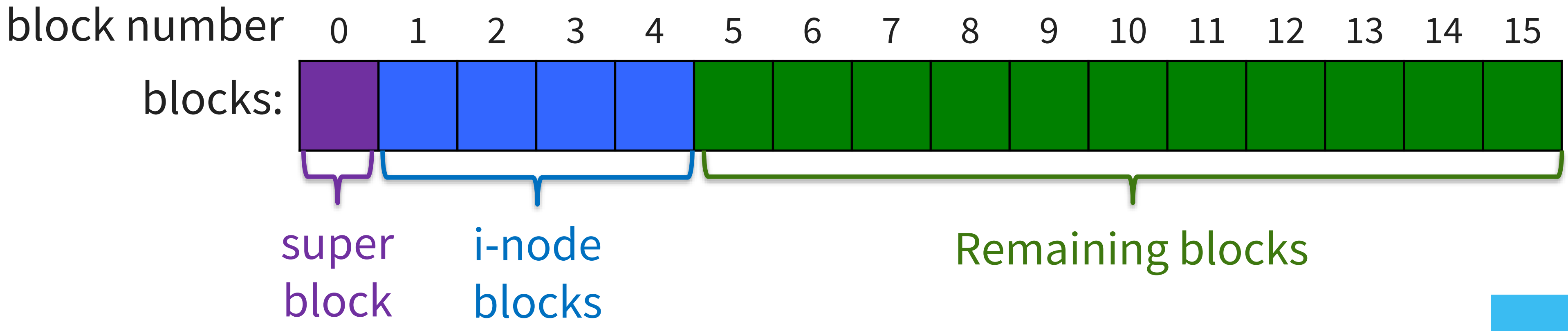
- a list of human readable names
- a mapping from each name to a specific underlying file or directory



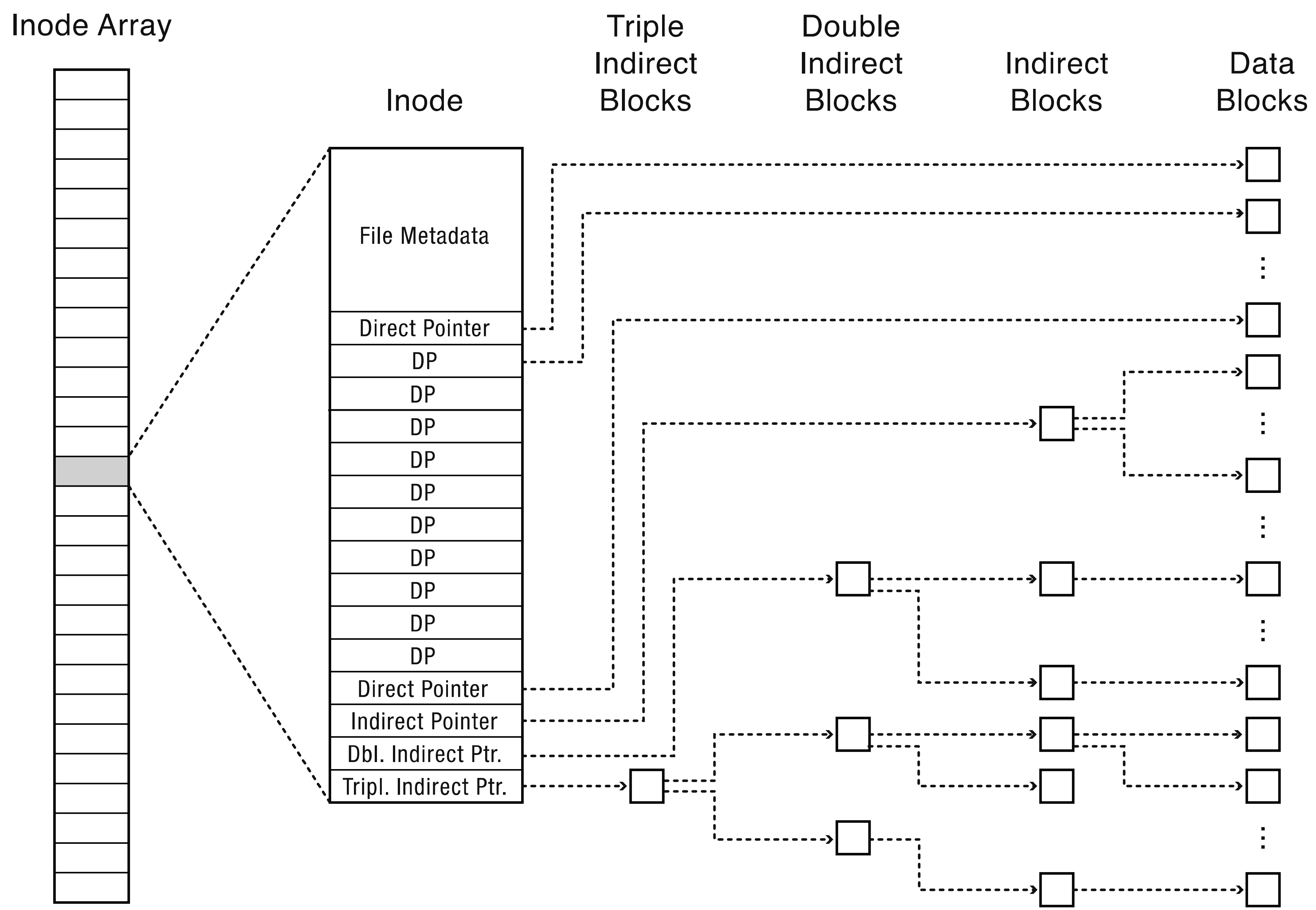
FFS Superblock

Identifies file system's key parameters:

- type
- block size
- inode array location and size
  (or analogous structure for other FSs)
- location of free list

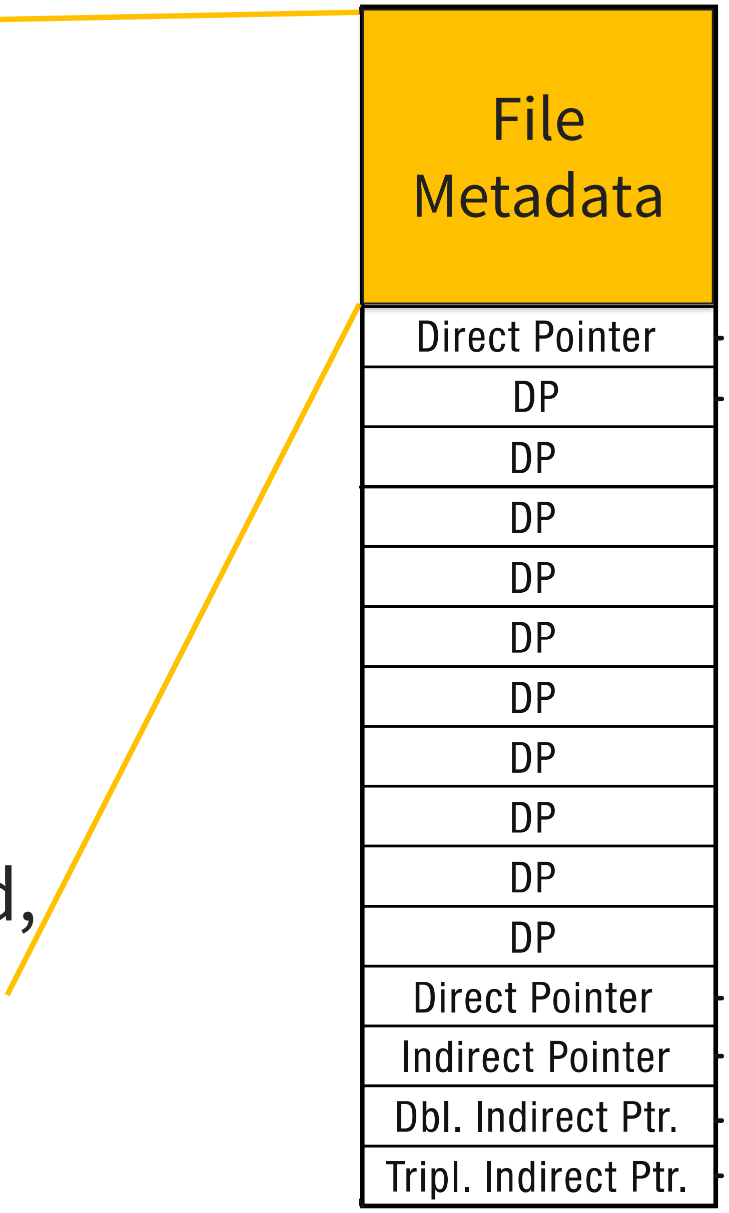


FFS: Index Structures



What else is in an inode?

- Type
  - ordinary file
  - directory
  - symbolic link
  - special device
- Size of the file (in #bytes)
- # links to the i-node
- Owner (user id and group id)
- Protection bits
- Times: creation, last accessed, last modified

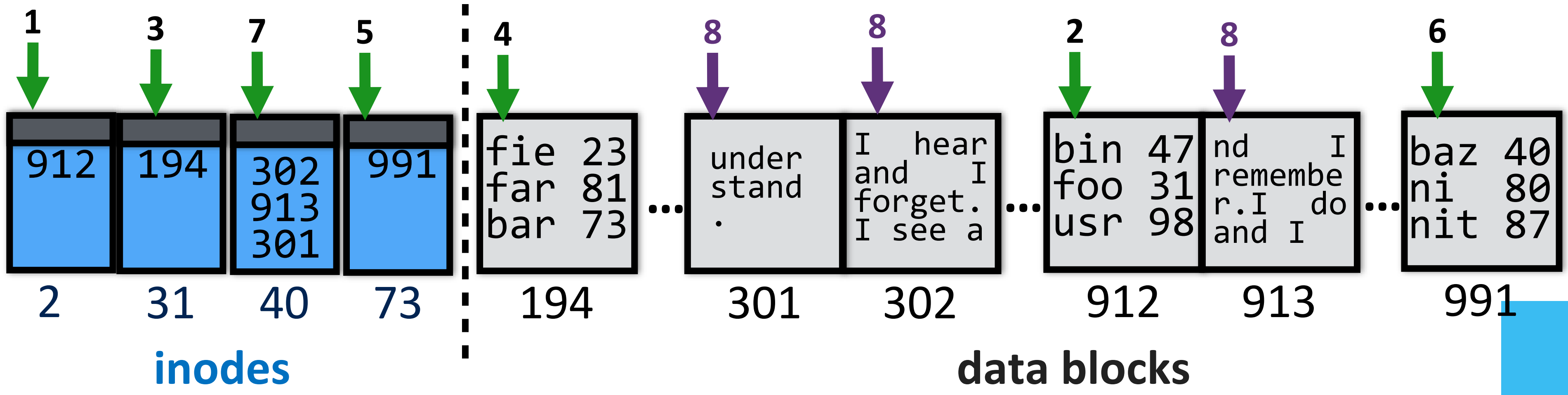


FFS: Steps to reading `/foo/bar/baz`

Read & Open:

(1) inode #2 (root always has inumber 2), find root's blocknum (912)
(2) root directory (in block 912), find foo's inumber (31)
(3) inode #31, find foo's blocknum (194)
(4) foo (in block 194), find bar's inumber (73)
(5) inode #73, find bar's blocknum (991)
(6) bar (in block 991), find baz's inumber (40)
(7) inode #40, find data blocks (302, 913, 301)
(8) data blocks (302, 913, 301)

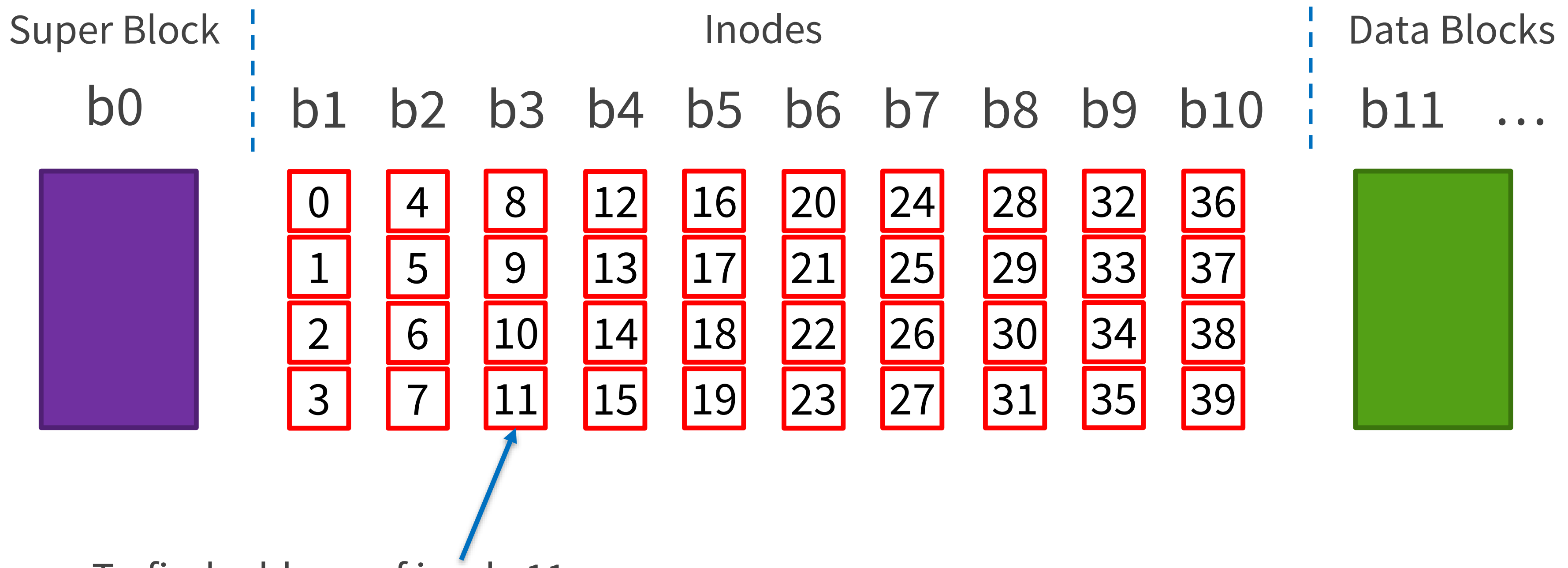
Caching allows first few steps to be skipped



Finding inodes in FFS

- Use inode number to index into inode array

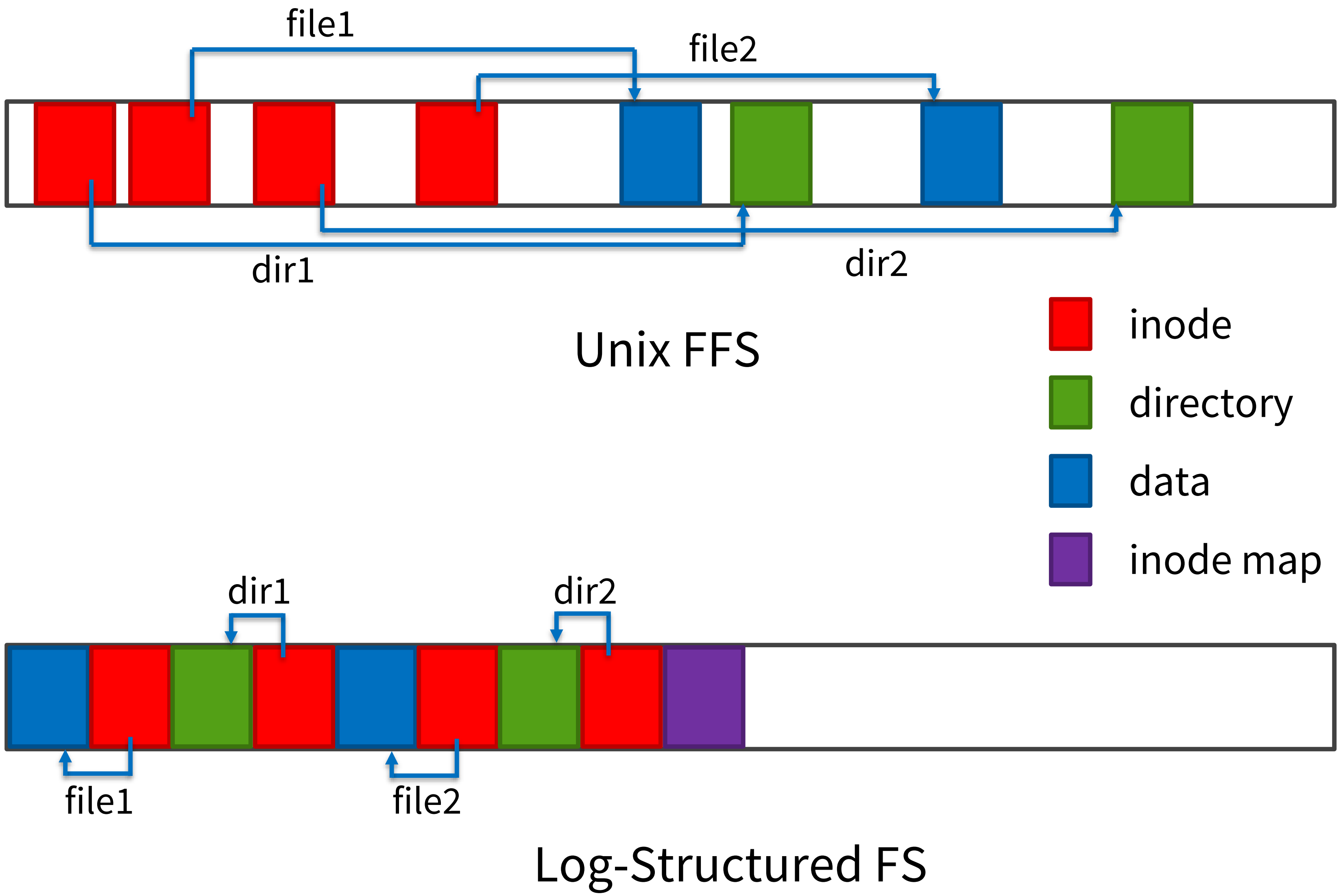
512 bytes/block
128 bytes/inode



To find address of inode 11:
addr(b1) + 11 * size(inode)

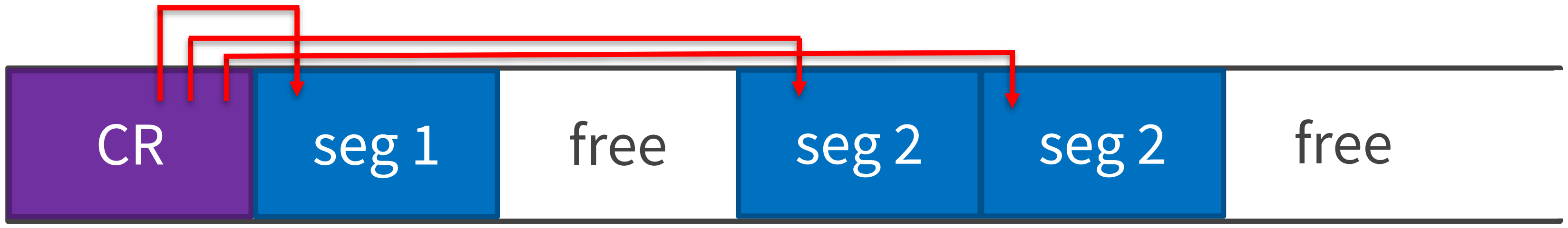
LFS vs FFS

Blocks written to create two 1-block files: dir1/file1 and dir2/file2



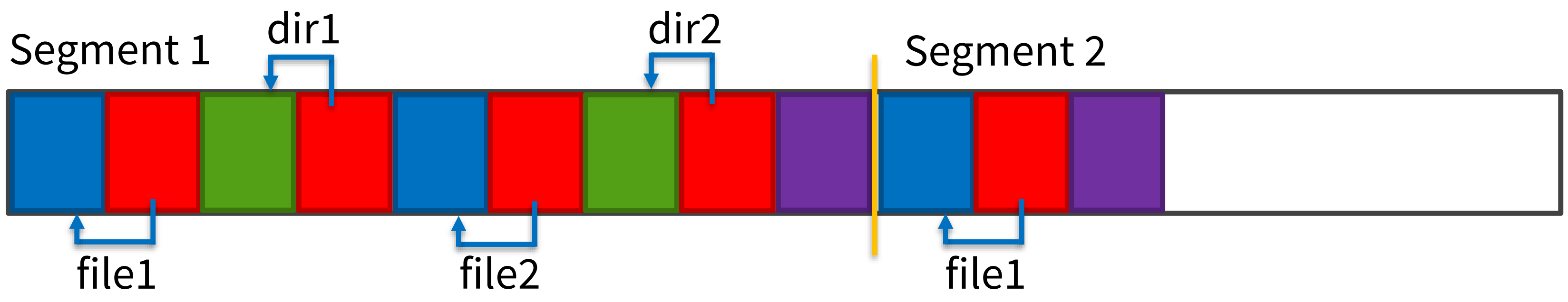
Finding Inodes in LFS

- Inode map: a table indicating where each inode is on disk
  - Normally cached in memory
  - Inode map blocks are written as part of the segment when updated
  - Still no seeking to write to imap ☺
- How do we find the blocks of the Inode map?
  - Listed in a fixed checkpoint region, updated periodically – same function as superblock in FFS



Overwriting Data in LFS

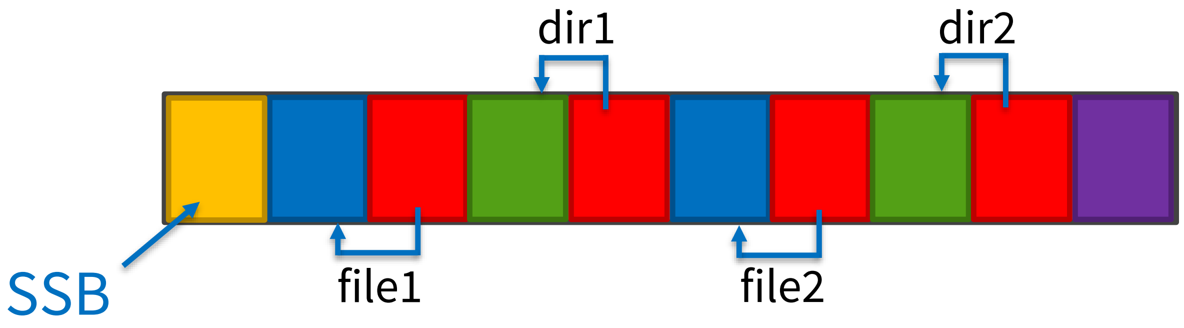
- To change data in block 1, create a new block 1
  - Update the inode (create a new one)
  - Update the imap



No need to change dir1, since file1 still has the same inode number

Segment Summary Block

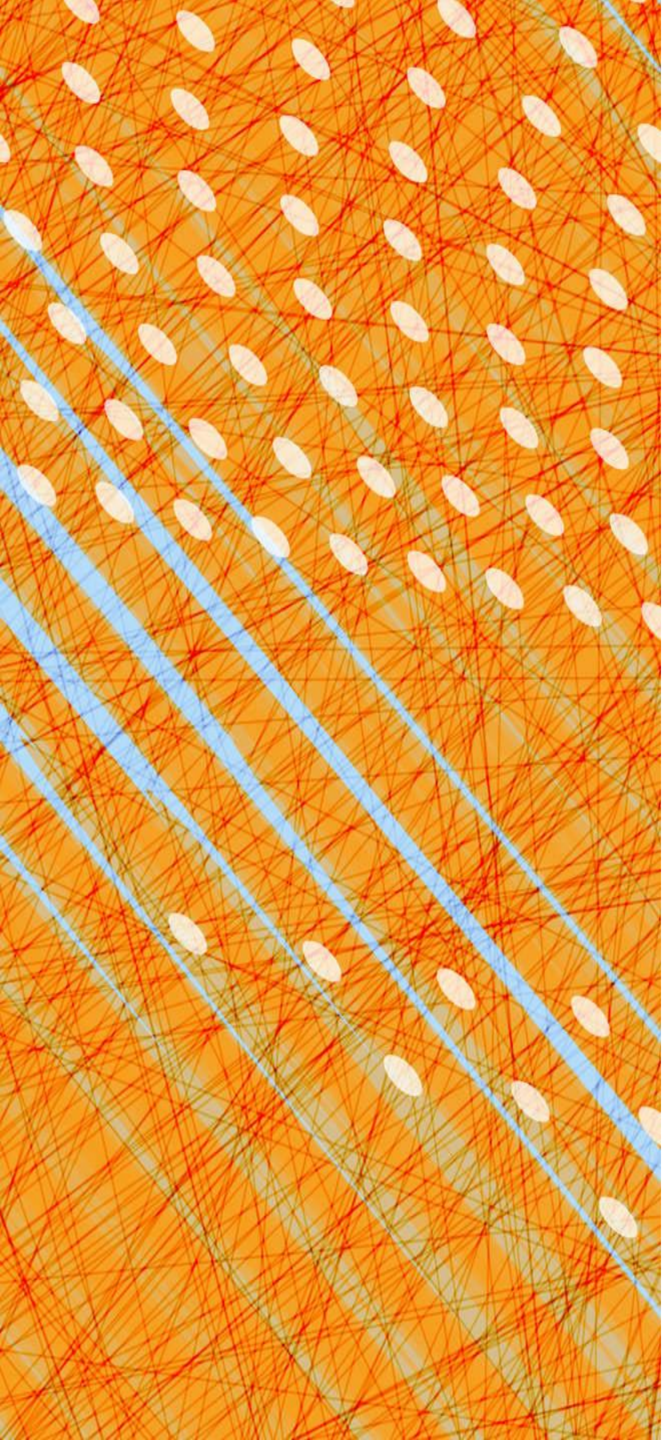
- Kept at the beginning of each segment
- For each data block in segment, SSB holds
  - The file the data block belongs to (inode#)
  - The offset (block#) of the data block within the file



Segment Summary Block

- During cleaning, to determine whether data block D is live:
  - Use inode# to find in imap where inode is currently on disk
  - Read inode (if not already in memory)
  - Check whether pointer for block *block#* refers to D's address
  - If not, D is dead
- Update file's inode with correct pointer if D is live and compacted to new segment

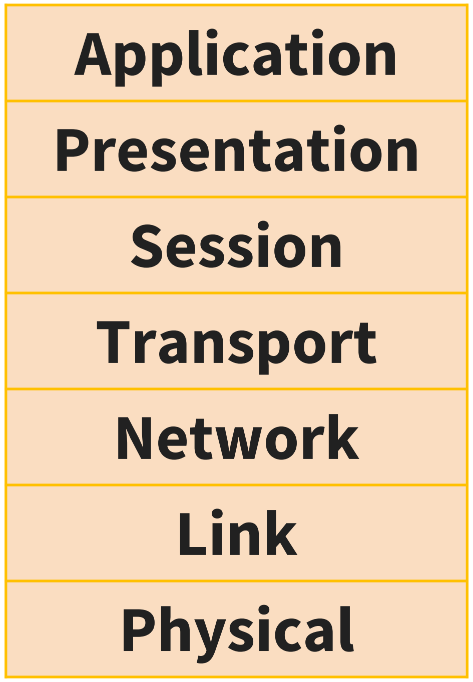
Networking



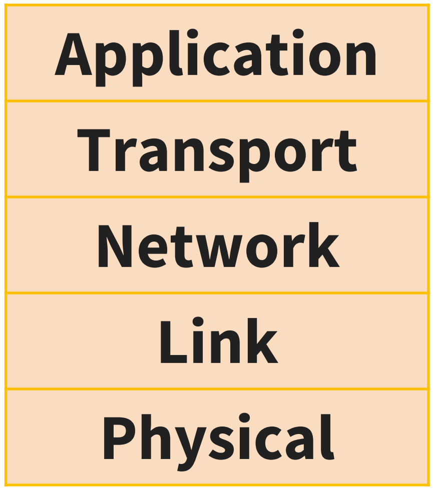
Network Layering

Network abstraction is usually *layered*

- Like Object Oriented-style inheritance
- Also like the hw/sw stack



Proposed 7-Layer ISO/OSI reference model (1970's)



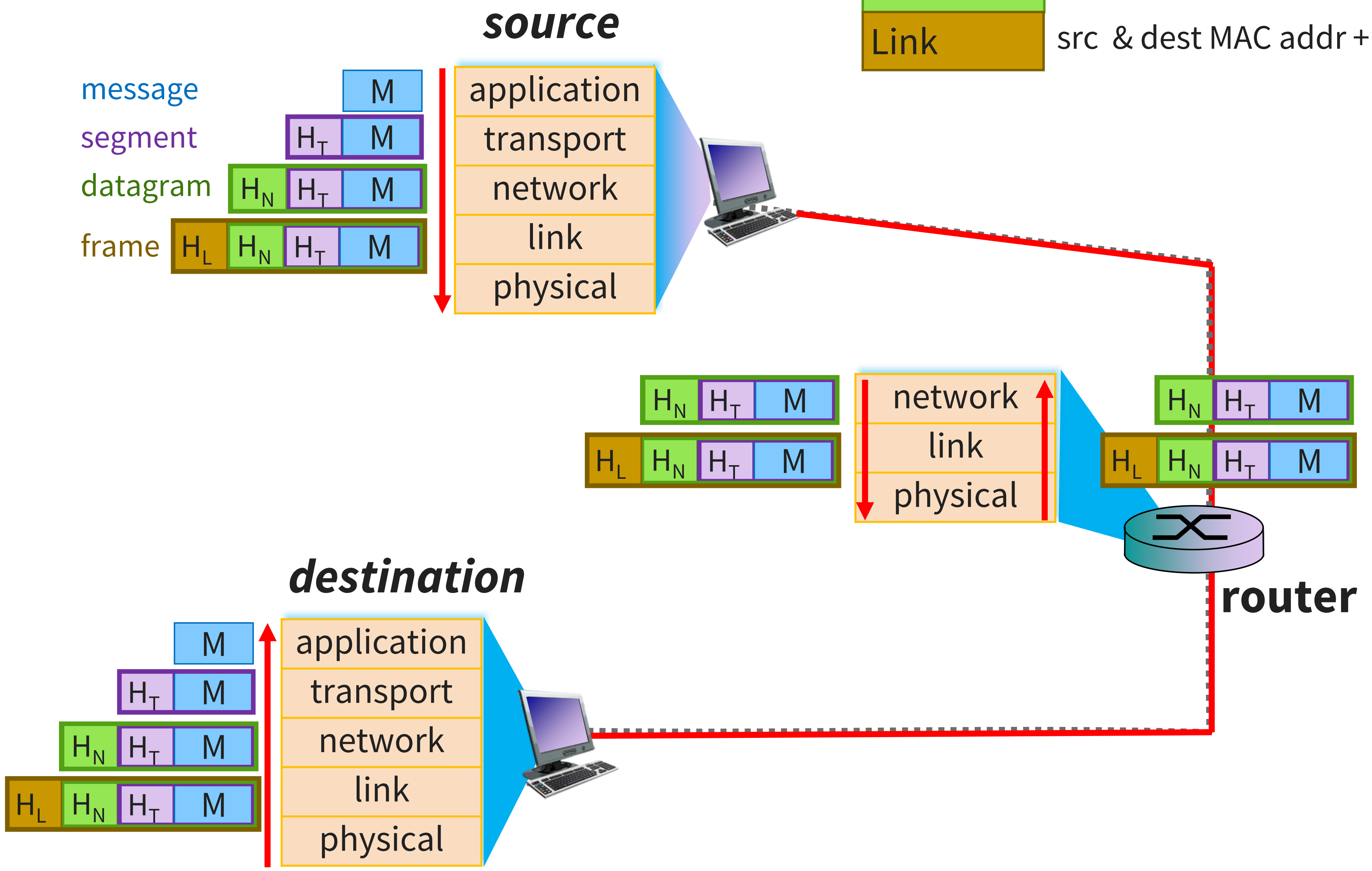
Actual 5-Layer Internet Protocol Stack

Internet Protocol Stack

Application	exchanges **messages**	HTTP, FTP, DNS
Transport	Transports messages; exchanges **segments**	TCP, UDP
Network	Transports segments; exchanges **datagrams**	IP, ICMP (ping)
Link	Transports datagrams; exchanges **frames**	Ethernet, WiFi
Physical	Transports frames; exchanges **bits**	wires, signal encoding

Encapsulation

Headers	
Transport	src & dst ports + …
Network	src & dest IP addr + …
Link	src & dest MAC addr + …



DNS Lookup

1. the client asks its local nameserver
2. the local nameserver asks one of the *root nameservers*
3. the root nameserver replies with the address of the authoritative nameserver
4. the server then queries that nameserver
5. repeat until host is reached, cache result.

Example: Client wants IP addr of www.amazon.com

1. Queries root server to find com DNS server
2. Queries `.com` DNS server to get `amazon.com` DNS server
3. Queries `amazon.com` DNS server to get IP address for `www.amazon.com`

Transport services and protocols

User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- no-frills extension of best-effort IP

"**Unreliable** Datagram Protocol"

Transmission Control Protocol (TCP)

- **reliable, in-order delivery**
- congestion control
- flow control
- connection setup

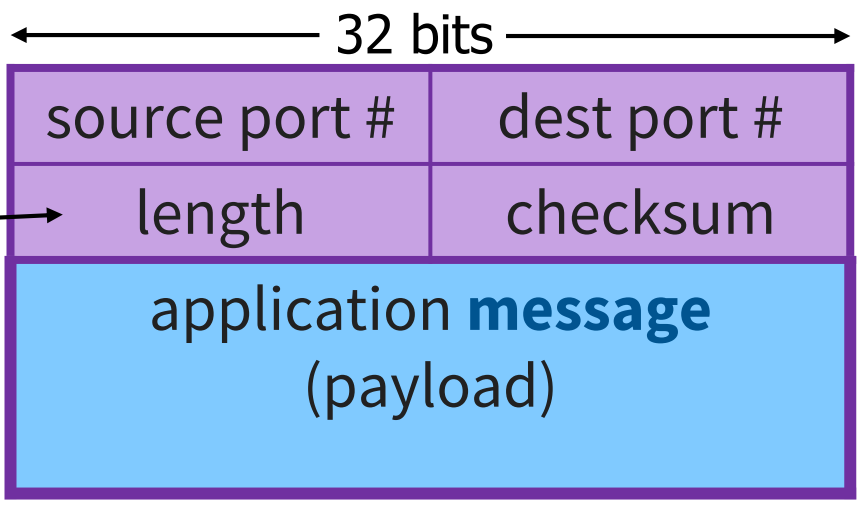
"**Trusty** Control Protocol"

Both provide:

- port numbers to identify sending/receiving processes
- additional headers inside IP packet

UDP Segment Format

length (in bytes) of UDP segment, including header



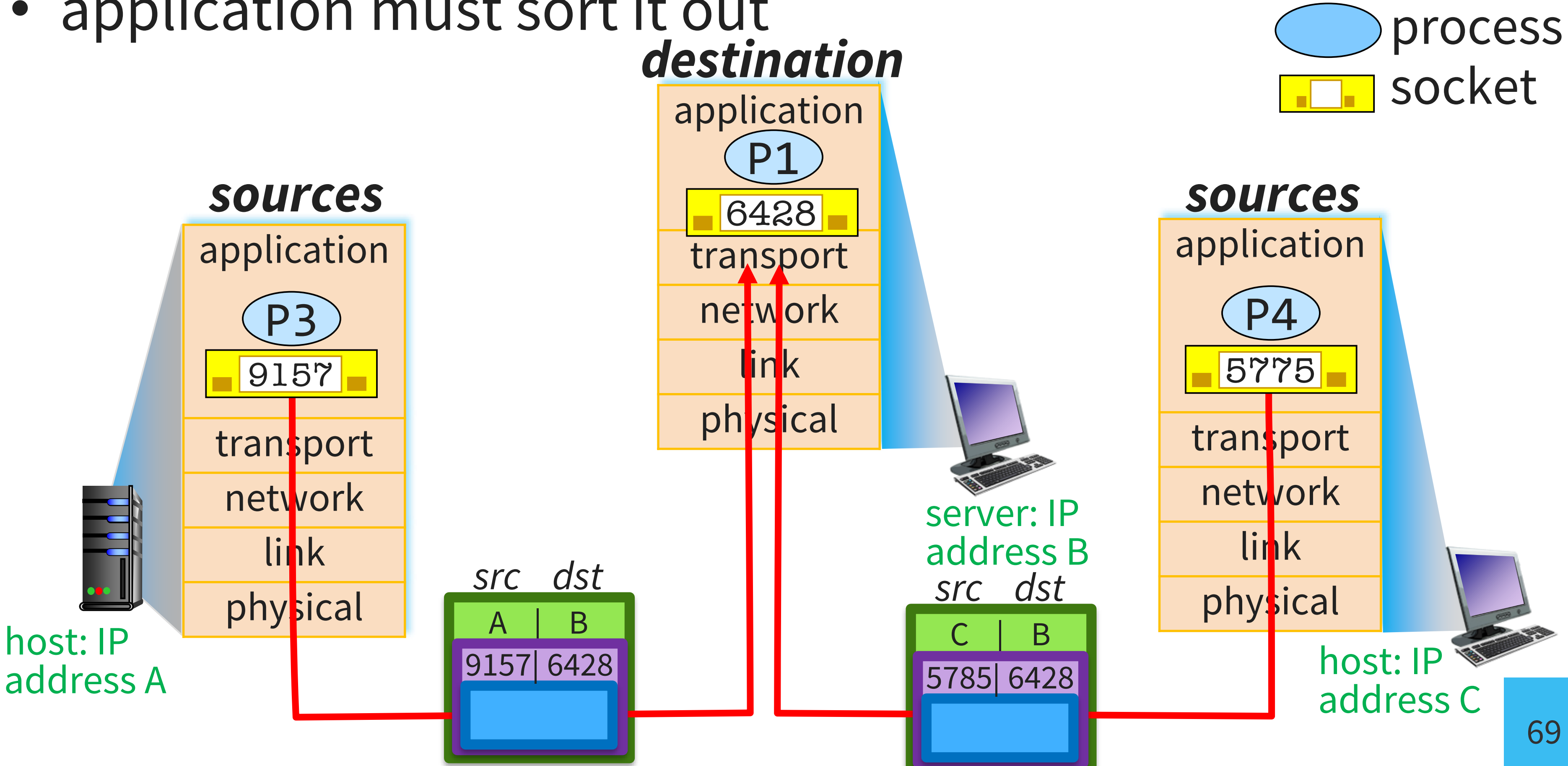
UDP header size: 8 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

UDP Sockets and Ports

Host receives 2 UDP segments:

- checks dst port, directs segment to socket w/that port
- *different src IP or port* but *same dst port* → *same socket*
- application must sort it out



TCP Segment Format

HL: header len

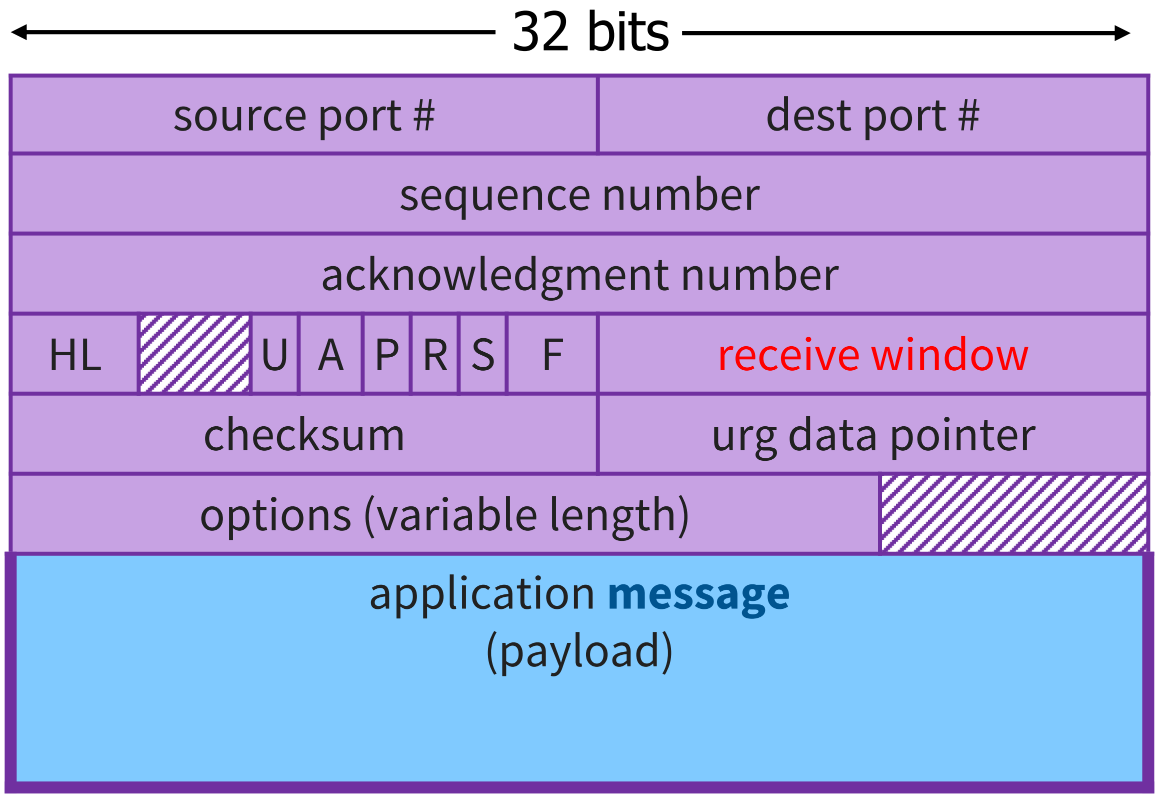
U: urgent data

A: ACK # valid

P: push data now

RST, SYN, FIN: connection commands (setup, teardown)

# bytes receiver willing to accept



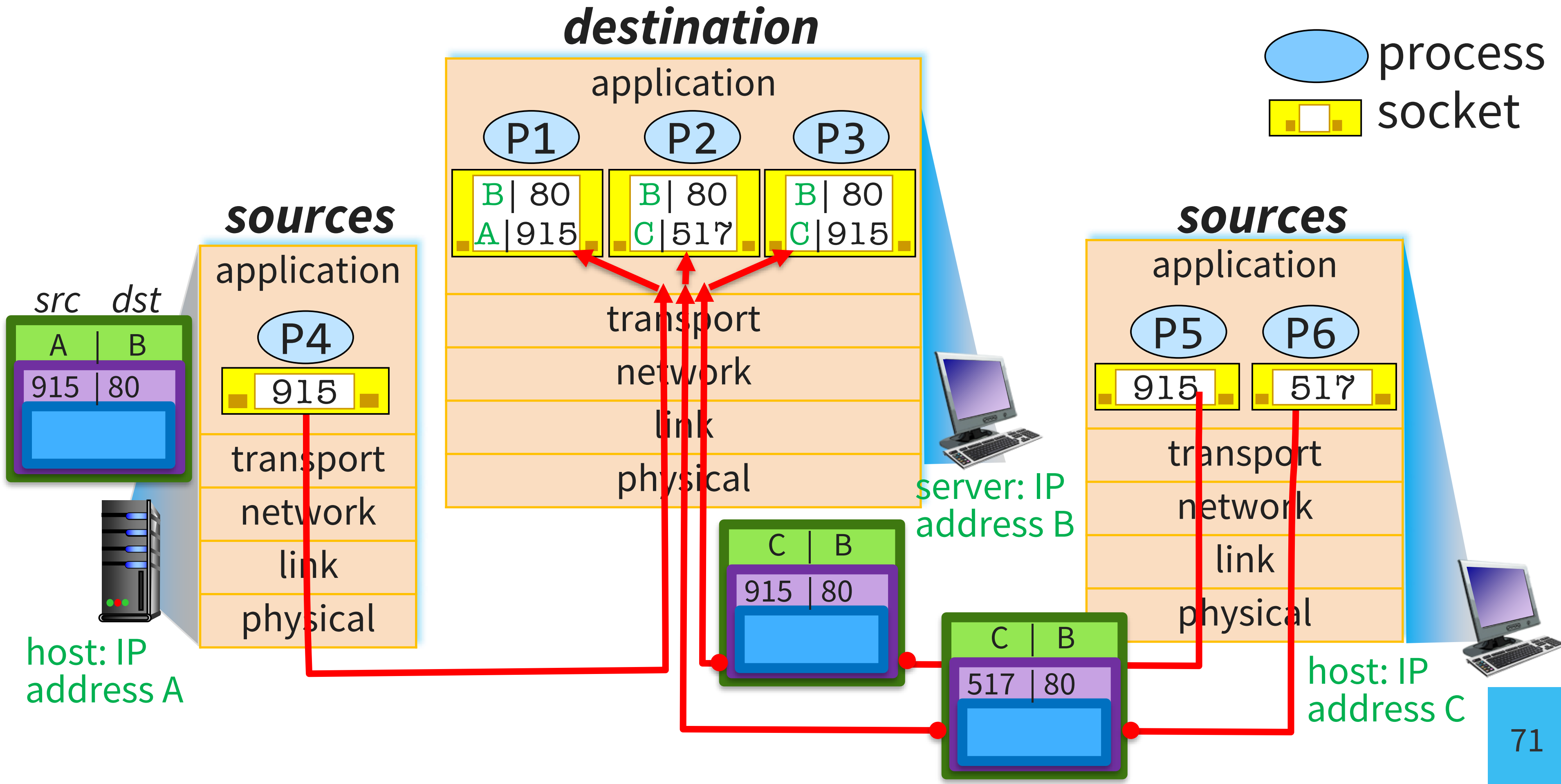
TCP header size: 20-60 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

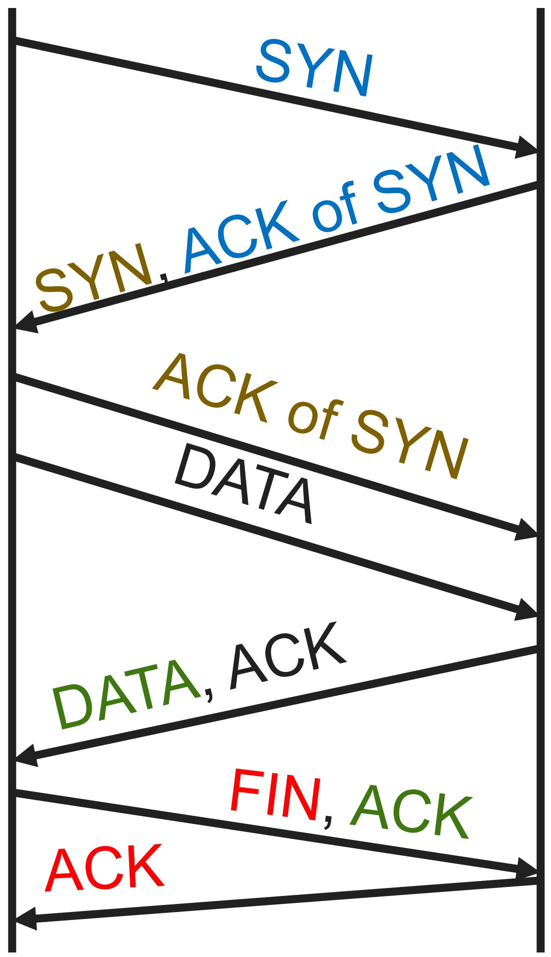
TCP Sockets and Ports

Host receives 3 TCP segments:

- all destined to IP addr B, port 80
- demuxed to different sockets with socket's 4-tuple



TCP Usage Pattern



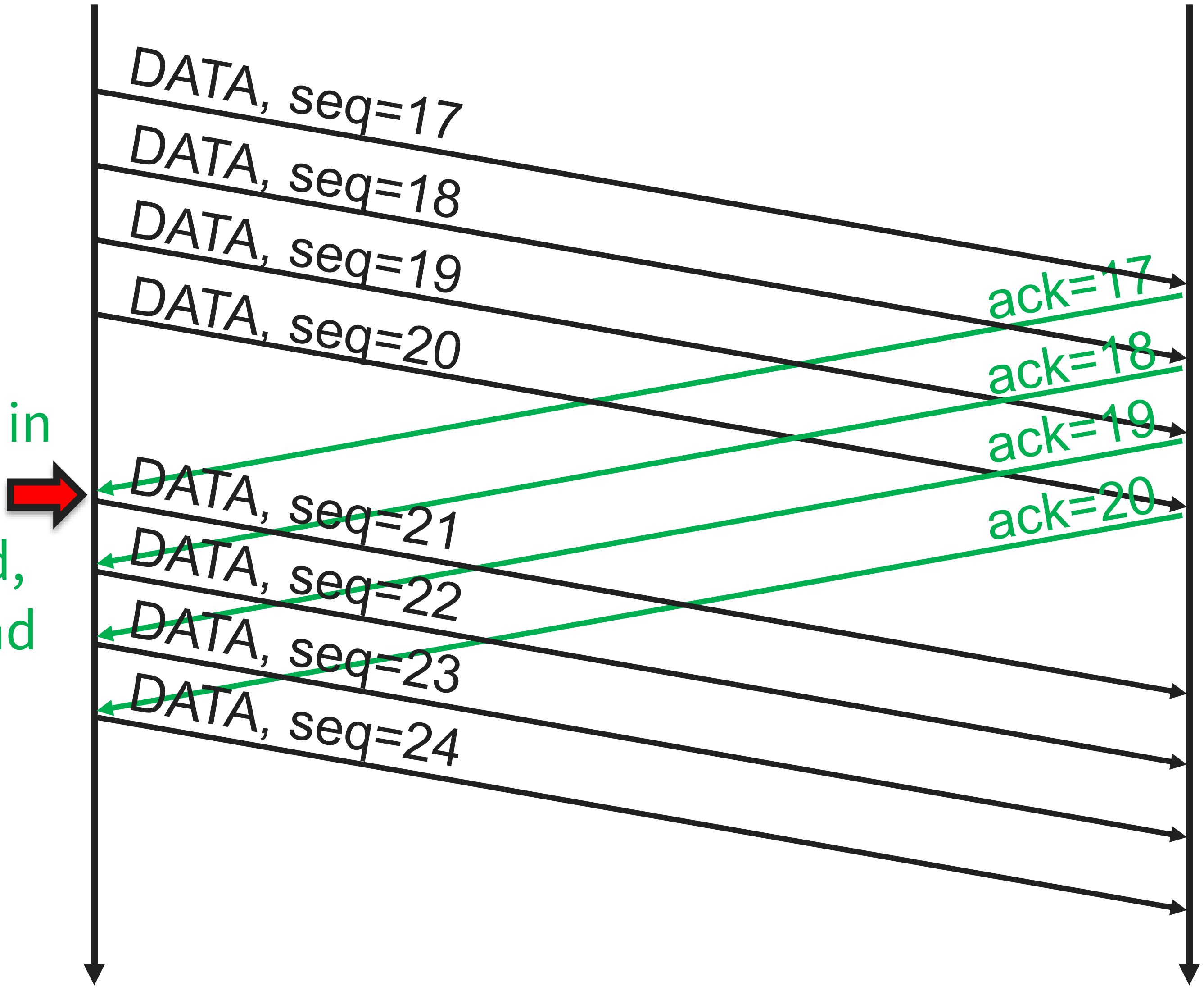
3 round-trips:

1. set up a connection
2. send data & receive a response
3. tear down connection

FINs work (mostly) like SYNs to tear down connection

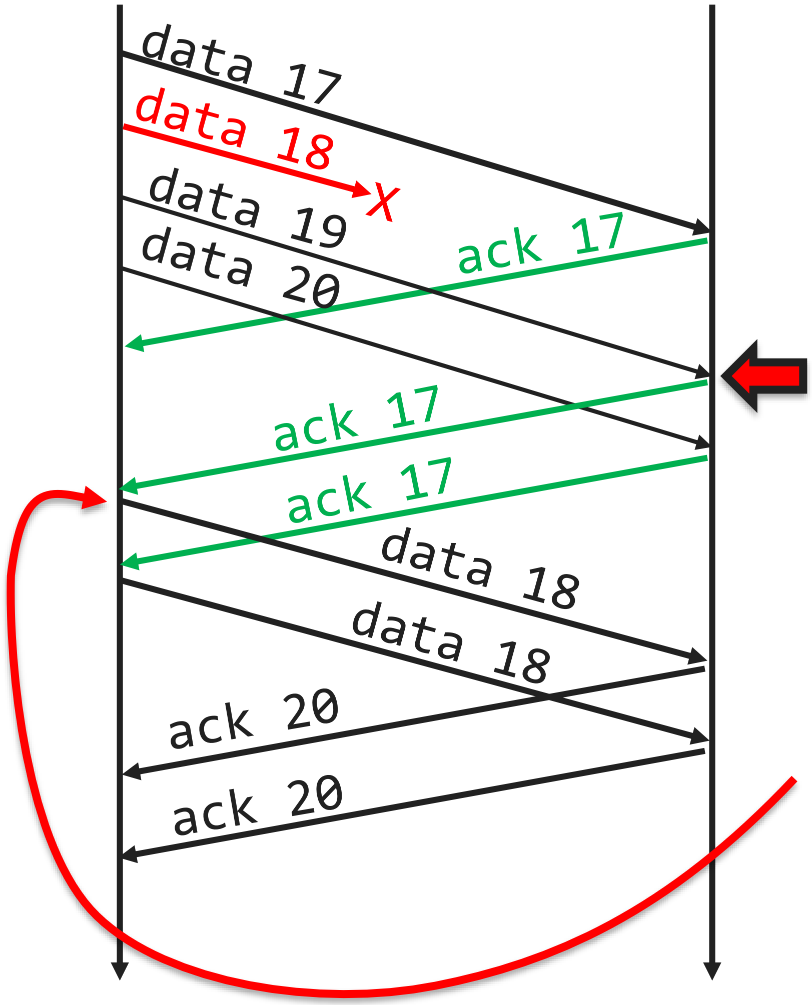
Need to wait after a FIN for straggling packets

TCP Congestion Window



When first item in window is acknowledged, sender can send the 5th item.

TCP **Fast Retransmit**



Receiver detects a lost packet (*i.e.*, a missing seq), ACKs the last id it successfully received

Sender can detect the loss without waiting for timeout

TCP Congestion Control

Additive-Increase/Multiplicative-Decrease (**AIMD**):

- window size++ every RTT if no packets dropped
- window size/2 if packet is dropped
  - drop evident from the acknowledgments

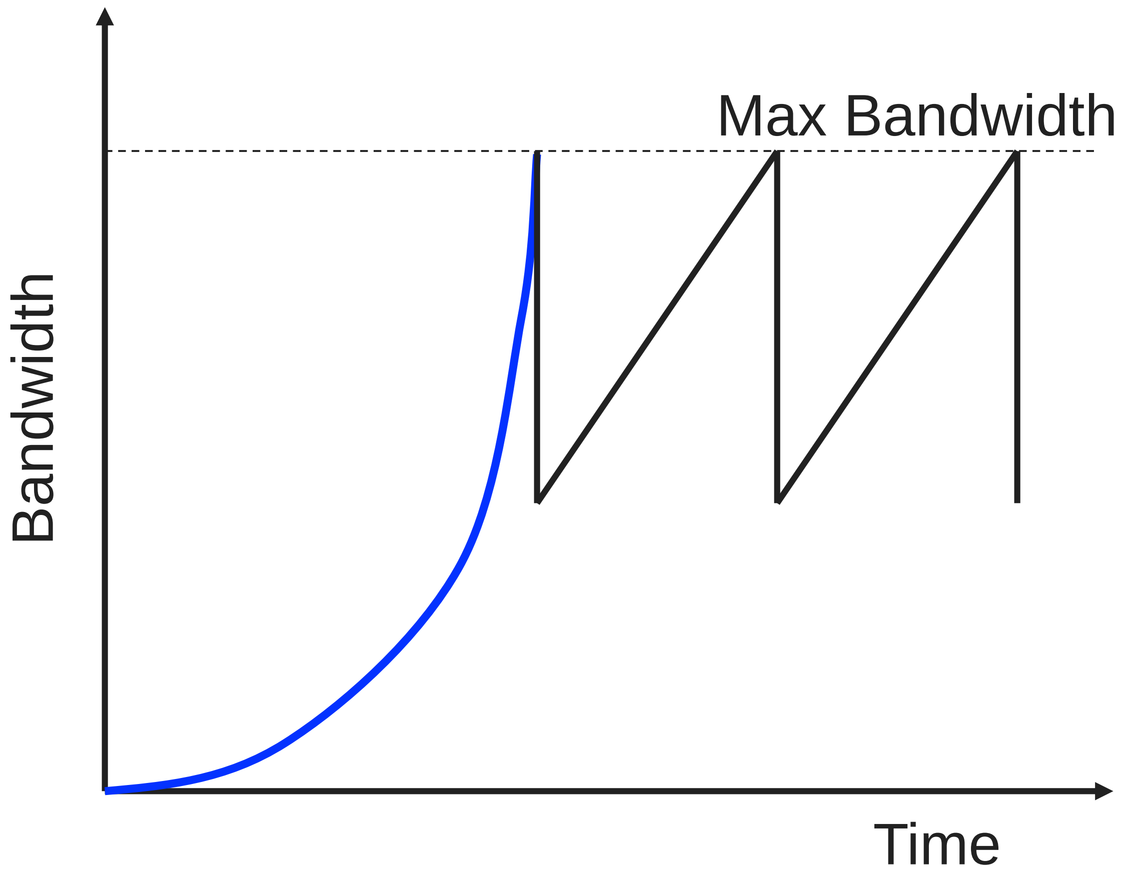
→ slowly builds up to max bandwidth, and hover there

- - Does not achieve the max possible
- + Shares bandwidth well with other TCP connections

This linear-increase, exponential backoff in the face of congestion is termed ***TCP-friendliness***

TCP Slow Start

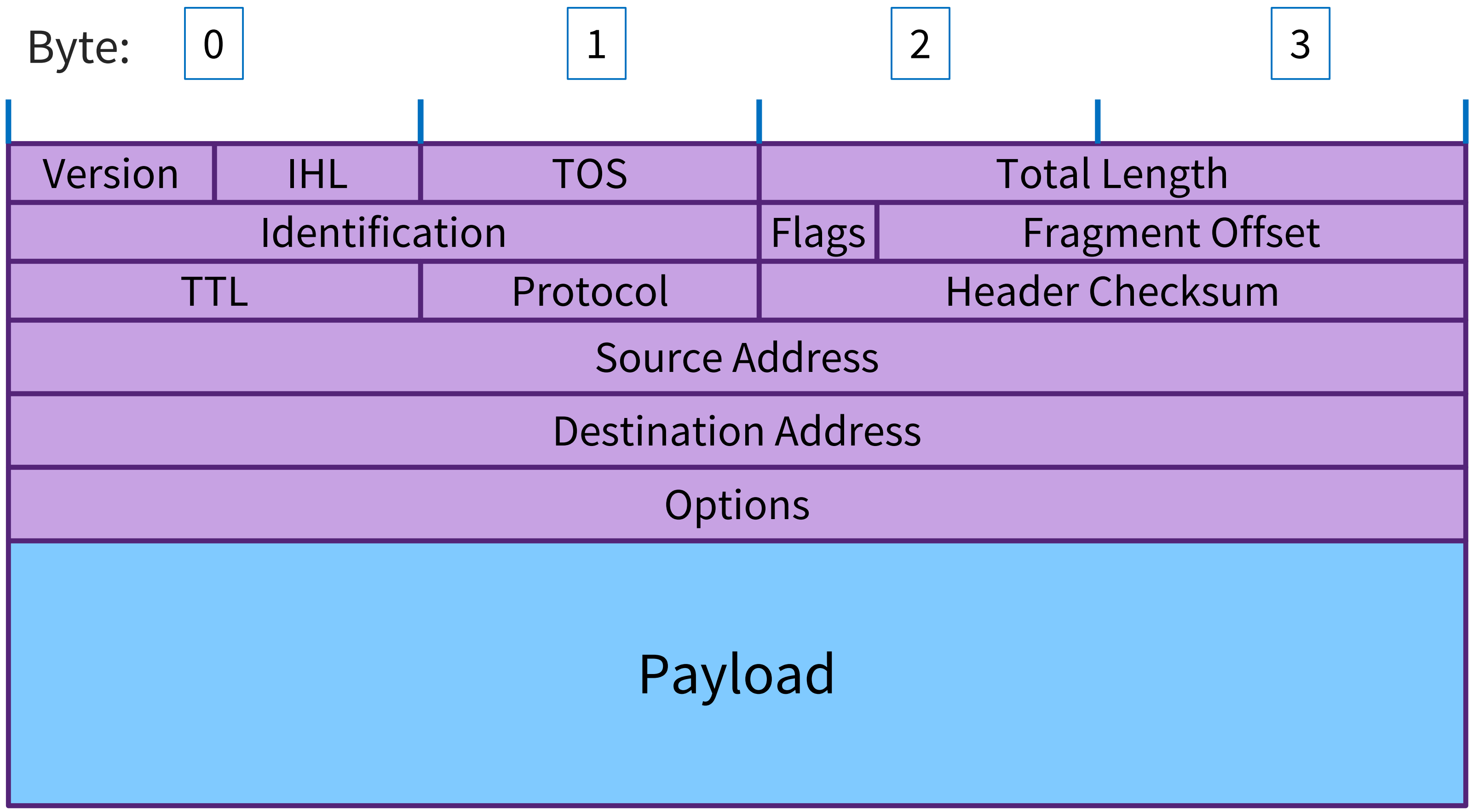
- Initial phase: **exponential increase**
- Assuming no other losses in the network except those due to bandwidth



IP

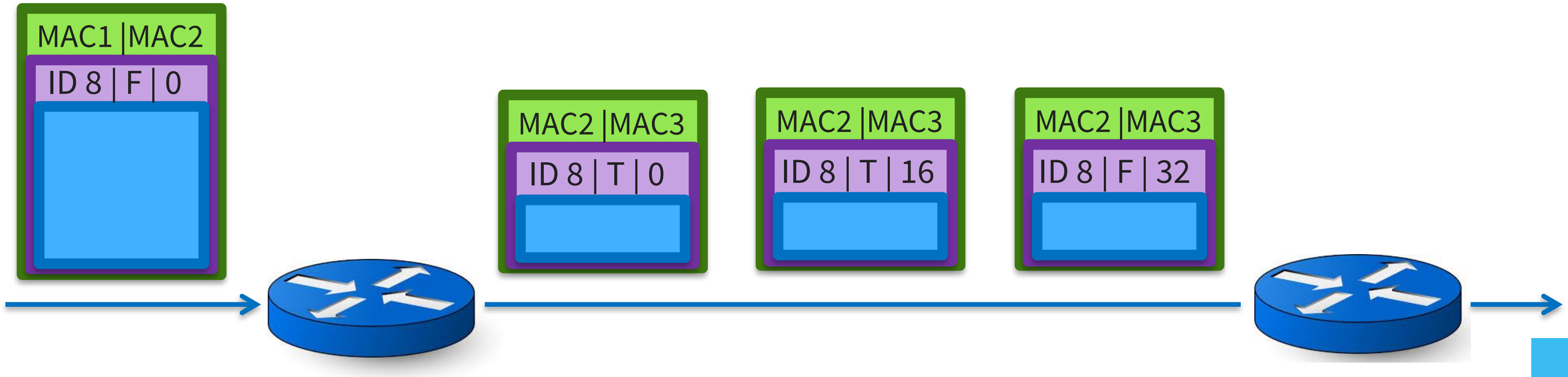
- Internetworking protocol
  - Network layer
- Common address format
- Common packet format for the Internet
  - Specifies what packets look like
  - *Fragments* long packets into shorter packets
  - *Reassembles* fragments into original shape
- IPv4 vs IPv6
  - IPv4 is what most people use
  - IPv6 more scalable and clears up some of the messy parts

IPv4 packet layout



IP Fragmentation Mechanics

- Source assigns each datagram an “identification”
- At each hop, IP can divide a long datagram into N smaller datagrams
- Sets the More Fragments bit except on the last packet
- Receiving end puts the fragments together based on *Identification* and *More Fragments* and *Fragment Offset (times 8)*



Routing Table

- Maps IP address to interface or port and to MAC address
- Longest Prefix Matching
- Your laptop/phone has a routing table too!

Address/Mask	IF or Port	MAC
128.84.216/23	en0	`c4:2c:03:28:a1:39`
127/8	lo0	`127.0.0.1`
128.84.216.36/32	en0	`74:ea:3a:ef:60:03`
128.84.216.80/32	en0	`20:aa:4b:38:03:24`
128.84.217.255/32	en0	`ff:ff:ff:ff:ff:ff`
130.18/16	en1	`c8:d4:58:1a:32:de`

Prefix of address to match

Number of bits in prefix

Netmask: a "1" for each bit that matters
For /16, netmask is 255.255.0.0

Router Function

often implemented in hardware

```
for ever:
    receive IP packet p
    if isLocal(p.dest): return localDelivery(p)
    if --p.TTL == 0: return dropPacket(p)
    matches = { }
    for each entry e in routing table:
        if p.dest & e.netmask == e.address & e.netmask:
            matches.add(e)
    bestmatch = matches.maxarg(e.netmask)
    forward p to bestmatch.port/bestmatch.MAC
```

Destination: 128.84.216.33
Entry: 128.84.216.0/23
Netmask: 255.255.254.0

Dest & Netmask = 128.84.216.0
Entry & Netmask = 128.84.216.0