



pollev.com/cse333a



Two Polls: About how long did Exercise 15 & 16 take you?

- A. [0, 2) hours
- B. [2, 4) hours
- C. [4, 6) hours
- D. [6, 8) hours
- E. 8+ Hours
- F. I didn't submit / I prefer not to say

Introduction to Concurrency

CSE 333 Winter 2026

Instructor: Amber Hu Justin Hsia

Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

Relevant Course Information

- ❖ Homework 4 due next Thursday (3/12)
 - You can still use **two** late days (until Sunday, 3/15)

- ❖ Final Exam on Wednesday, 3/18 12:30-2:20 PM (110 min)
 - KNE 110 & 120
 - See [Ed Post #630](#) for details
 - Final review session on Friday, 3/13
 - CSE2 G10 and on Zoom 4:30-6:30 PM
 - One sheet of handwritten notes
 - Not cumulative

Some Common HW4 Bugs

- ❖ Your server works, but is really, really slow
 - Check the 2nd argument to the `QueryProcessor` constructor
- ❖ Funny things happen after the first request
 - Make sure you're not destroying the `HTTPConnection` object too early (*e.g.*, falling out of scope in a while loop)
- ❖ Server crashes on a blank request
 - Make sure that you handle the case that `read()` (or `WrappedRead()`) returns `0`

Lecture Outline

- ❖ **From Query Processing to a Search Server**
- ❖ Concurrency and Concurrency Methods

Building a Web Search Engine

❖ We have:

■ Some indexes

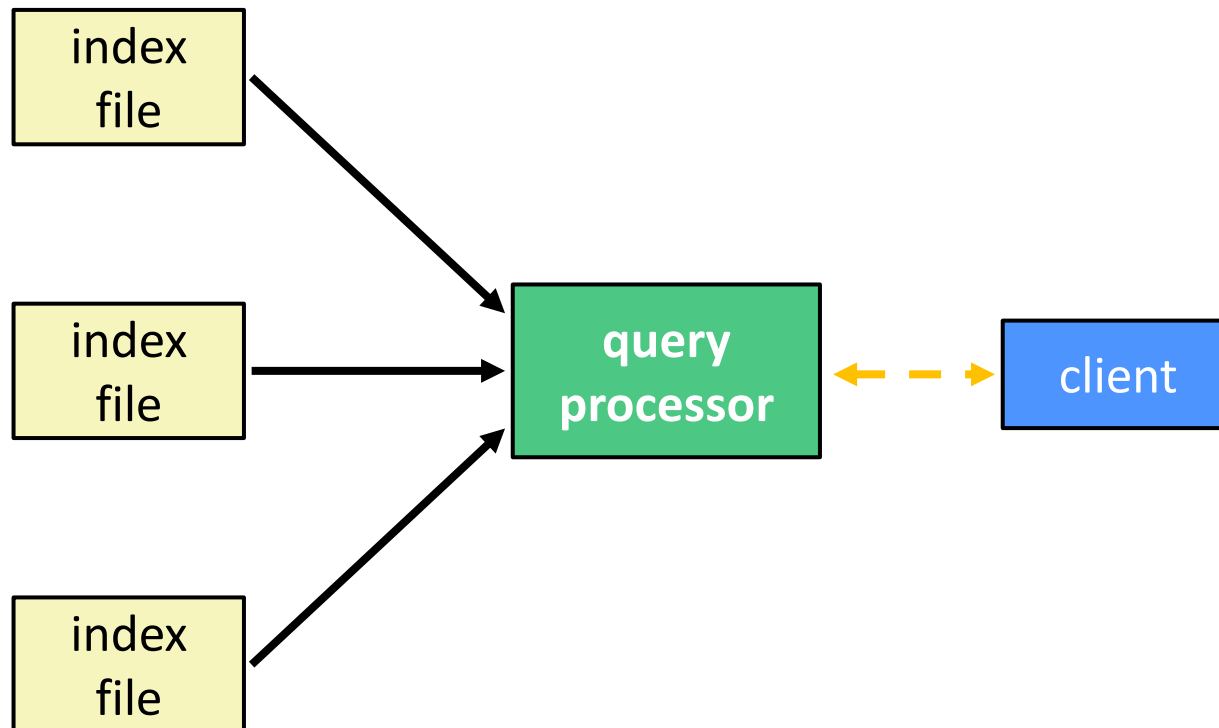
- A map from *<word>* to *<list of documents containing the word>*
- This is probably sharded over multiple files

Take Distributed to learn more!
CSE 452

■ A query processor

- Accepts a query composed of multiple words
- Looks up each word in the index
- Merges the result from each word into an overall result set

Search Engine Architecture



Sequential Search Engine (Pseudocode)

```
fn Lookup(string word) -> doclist:
    bucket = hash(word)
    hitlist = file.read(bucket)
    foreach hit in hitlist:
        doclist.append(file.read(hit))
    return doclist

fn main():
    SetupServerToReceiveConnections()
    while (true):
        string query_words[] = GetNextQuery()
        results = Lookup(query_words[0])
        foreach word in query[1..n]:
            results = results.intersect(Lookup(word))
        Display(results)
```

Handwritten annotations in red:

- "returns" with an arrow pointing to the return type `doclist` in the `Lookup` function signature.
- "disk I/O" with arrows pointing to `file.read(bucket)` and `file.read(hit)`.
- "network I/O" with an arrow pointing to `GetNextQuery()`.
- "network I/O" with an arrow pointing to `Display(results)`.

Full implementation in `searchserver_sequential/`

Why Sequential?

❖ Advantages:

- Super(?) simple to build/write

❖ Disadvantages:

- Incredibly poor performance
 - One slow client will cause *all* others to block
 - Poor utilization of resources (CPU, network, disk)

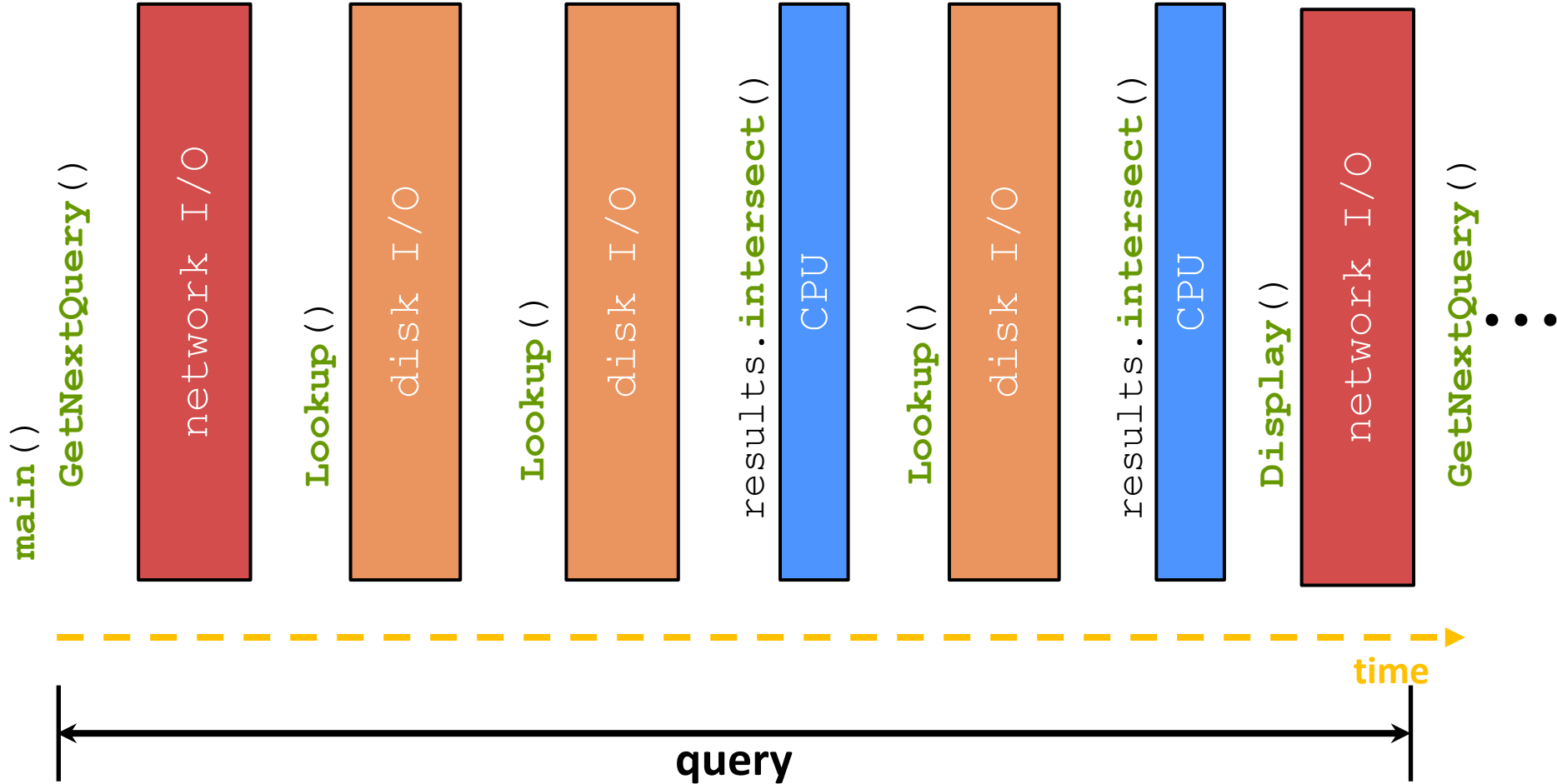
Execution Timeline: a Multi-Word Query

3-word query:

ocean

whale



ravenous



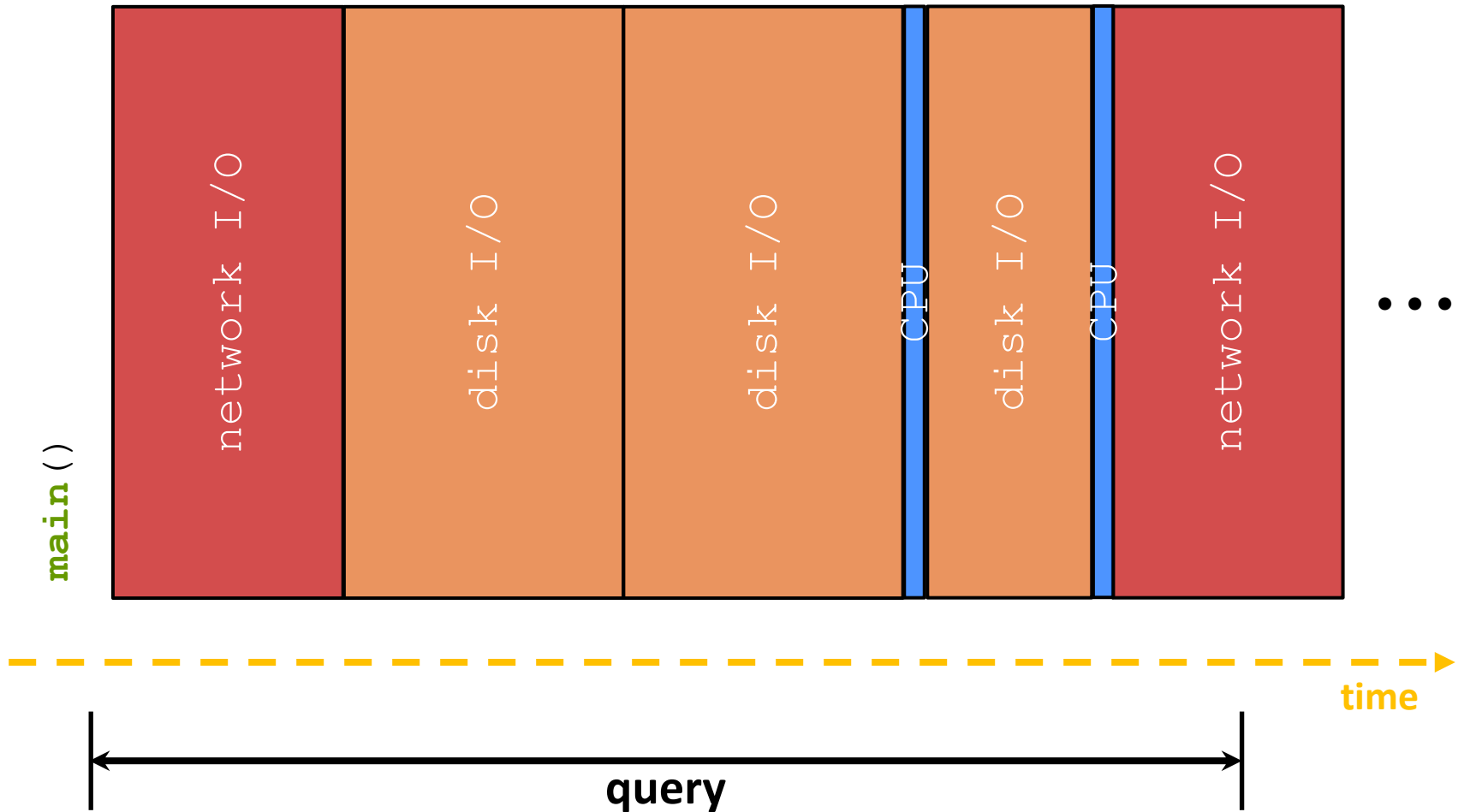
What About I/O-caused Latency?

- ❖ Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

Numbers Everyone Should Know	
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zip	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from network	10,000,000 ns
Read 1 MB sequentially from disk	30,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns



Execution Timeline: (Loosely) To Scale



Multiple (Single-Word) Queries

Q# is the Query Number

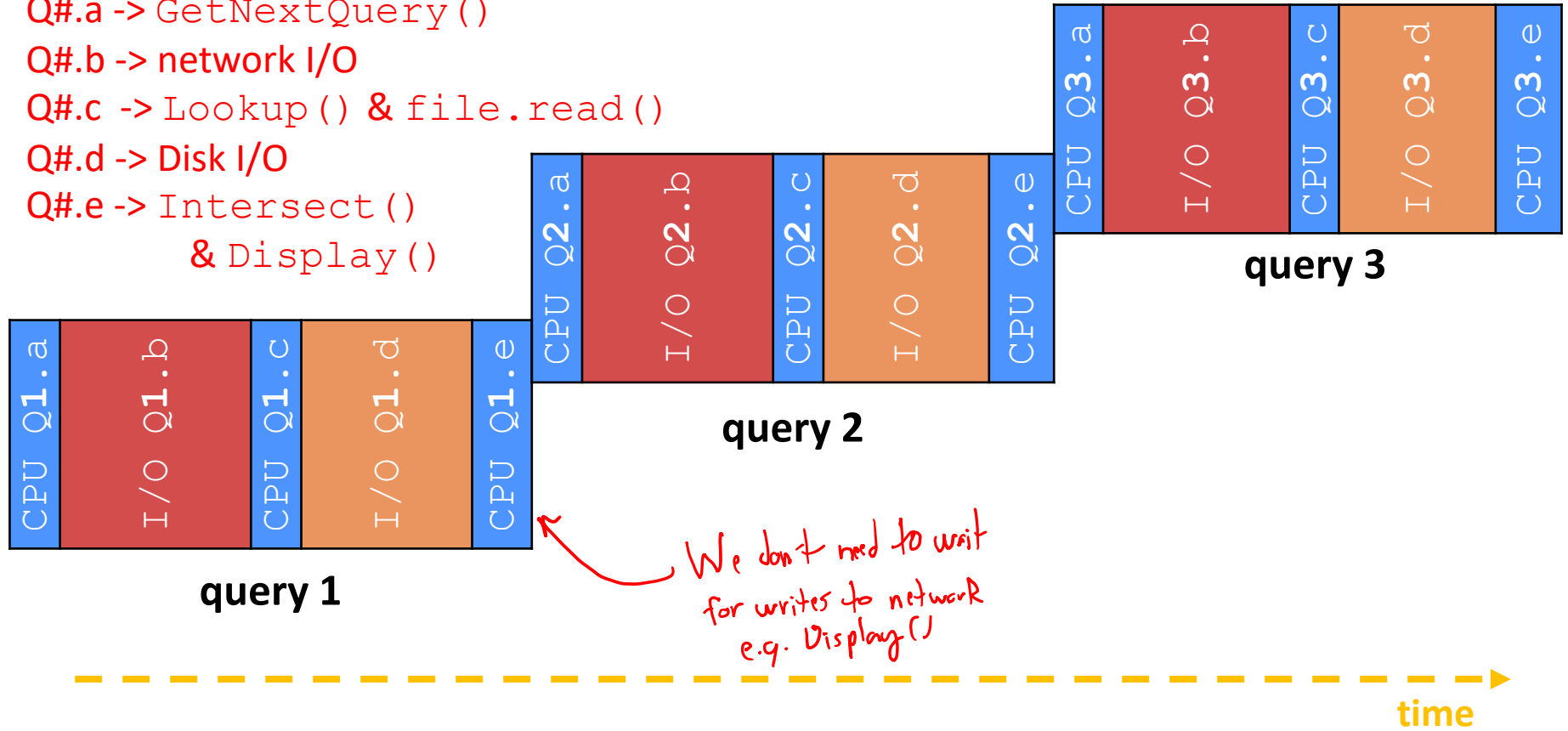
Q#.a -> GetNextQuery()

Q#.b -> network I/O

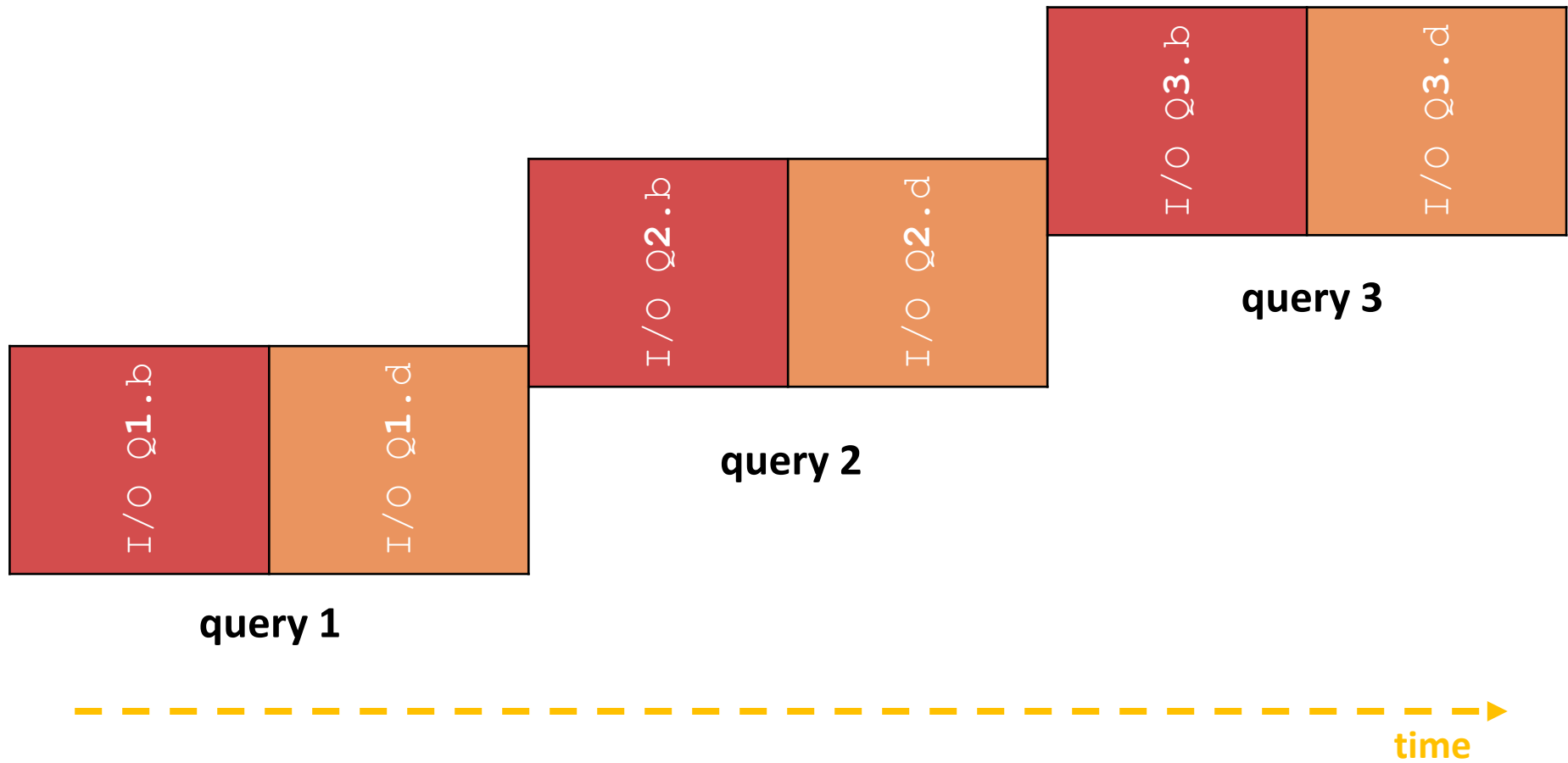
Q#.c -> Lookup() & file.read()

Q#.d -> Disk I/O

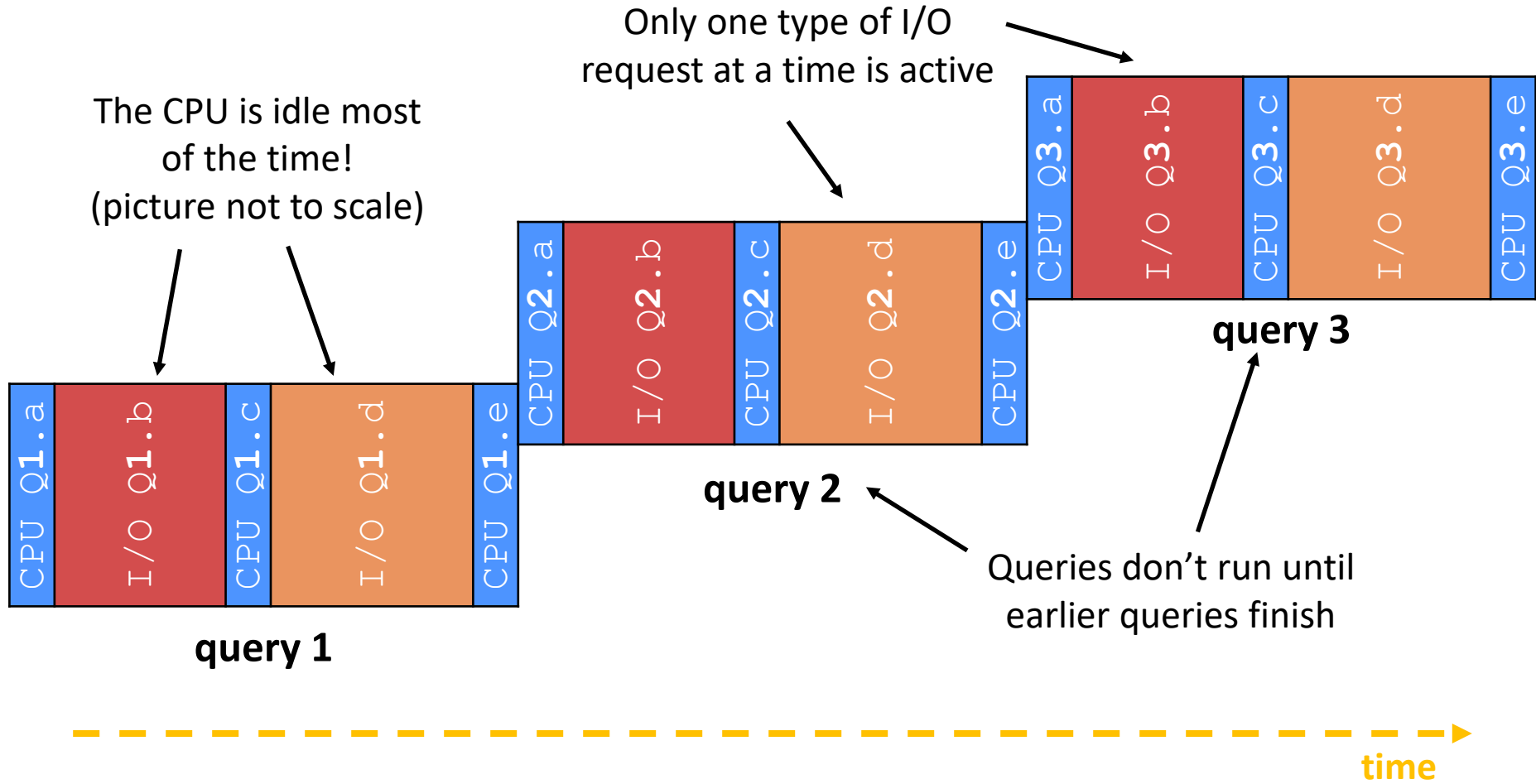
Q#.e -> Intersect()
& Display()



Multiple Queries: (Loosely) To Scale



Sequential Issues



Sequential Can Be Inefficient

- ❖ Only one query is being processed at a time
 - All other queries queue up behind the first one
 - And clients queue up behind the queries ...
- ❖ Even while processing one query, the CPU is idle the vast majority of the time
 - It is *blocked* waiting for I/O to complete
 - Disk and network I/O can be very slow (10 million times slower ...)
- ❖ Only one type of I/O request is active at a time
 - Missed opportunities to speed I/O up
 - Separate devices in parallel, better scheduling of a single device, etc.

Lecture Outline

- ❖ From Query Processing to a Search Server
- ❖ **Concurrency and Concurrency Methods**

Concurrency

- ❖ Concurrency != parallelism
 - **Concurrency is working on multiple tasks with overlapping execution times**
 - Parallelism is executing multiple CPU instructions *simultaneously*
↳ requires hardware support
- ❖ Our search engine could run concurrently in multiple different ways:
 - Example: Issue ***I/O requests*** against different files/disks simultaneously
 - Could read from several index files at once, processing the I/O results as they arrive
 - Example: Execute multiple ***queries*** at the same time
 - While one is waiting for I/O, another can be executing on the CPU

A Concurrent Implementation

- ❖ Use multiple “workers”
 - As a query arrives, create a new worker to handle it
 - The worker reads the query from the network, issues read requests against files, assembles results and writes to the network
 - The worker alternates between consuming CPU cycles and blocking on I/O
 - The OS context switches between workers
 - While one is blocked on I/O, another can use the CPU
 - Multiple workers’ I/O requests can be issued at once

- ❖ **So what should we use for our “workers”?**

Worker Option 1: Processes (Review)

- ❖ Processes can use `fork` to “clone” itself
 - These two processes have a parent-child relationship
 - Work almost entirely independent of each other
- ❖ The major components of a **process** are:
 - An address space to hold data and instructions *OS affords private virtual address space*
 - Open resources such as file descriptors *fork() copies fd table*
 - Current state of execution
 - Includes values of registers (including program counter and stack pointer) and parts of memory (the Stack, in particular)

Why Processes?

❖ Advantages:

- Processes are isolated from one another
 - No shared memory between processes
 - If one crashes, the other processes keep going
- No need for language support (OS provides `fork`)

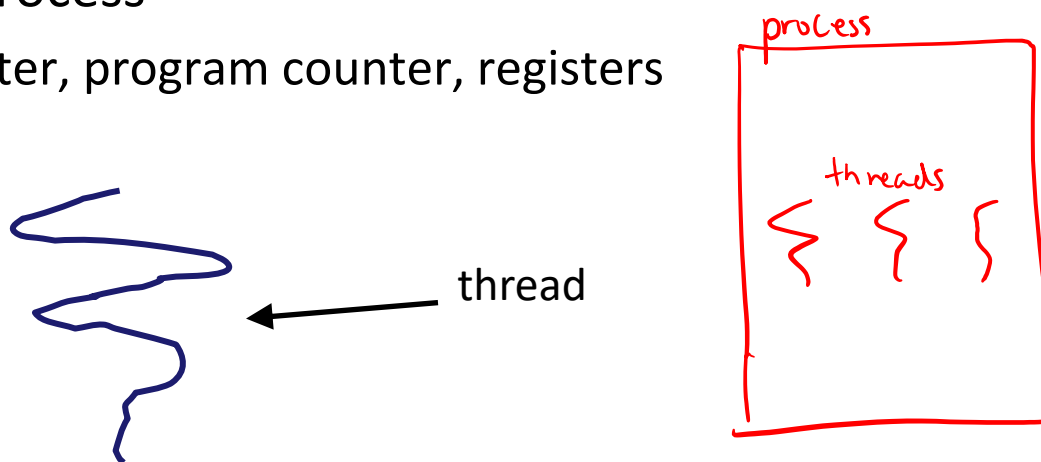
❖ Disadvantages:

- A lot of overhead during creation and context switching
- Cannot easily share memory between processes
 - Typically must communicate through the file system *slow!*
 - *e.g.*, POSIX pipe/FIFO: one way data stream between two processes

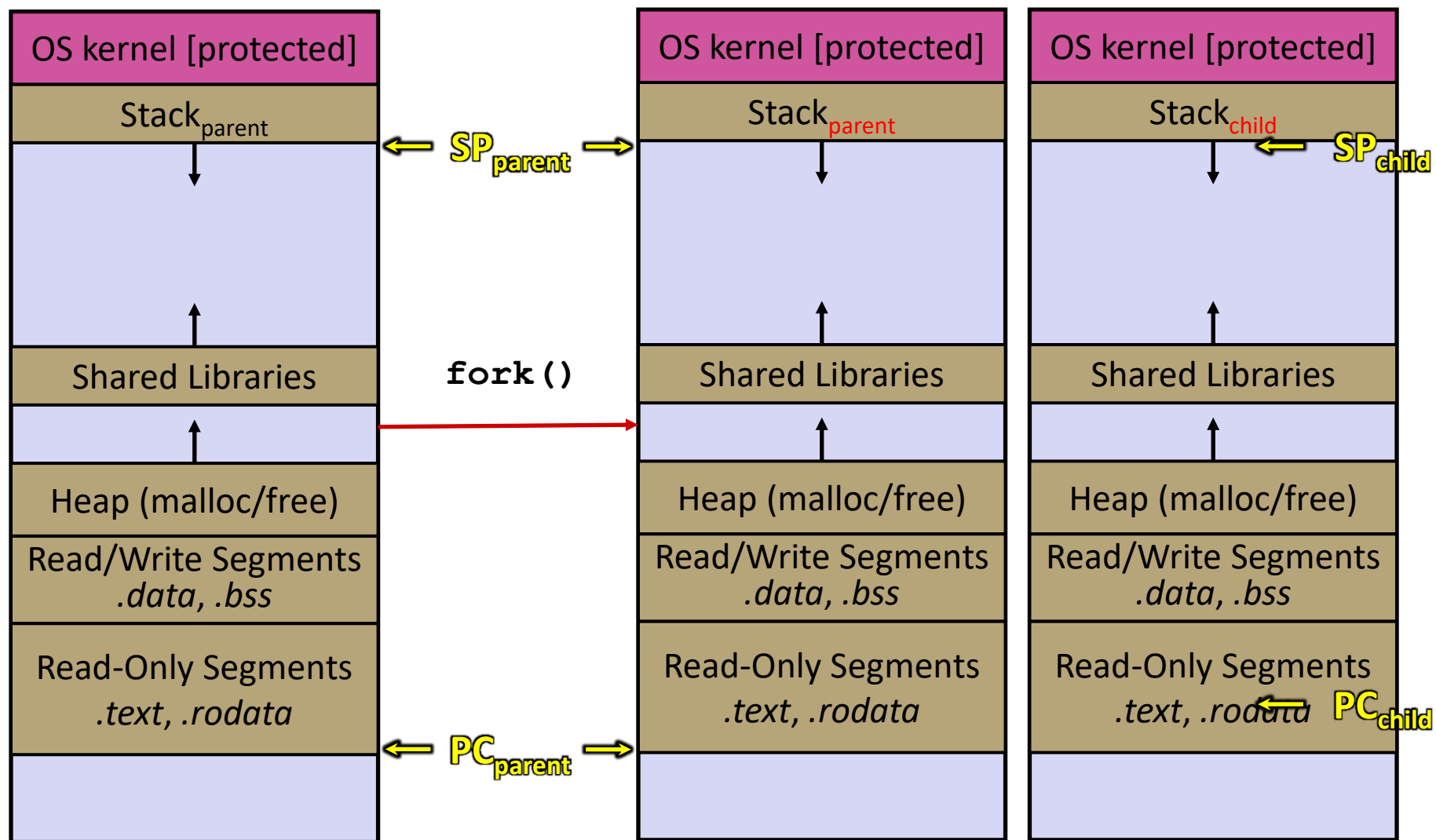
More on Lecture 25

Worker Option 2: Threads

- ❖ From within a process, we can separate out the concept of a “thread of execution” (**thread** for short)
 - Processes are the containers that hold shared resources and attributes
 - e.g., address space, file descriptors, security attributes
 - Threads are independent, sequential execution streams (*units of scheduling*) within a process
 - e.g., stack, stack pointer, program counter, registers



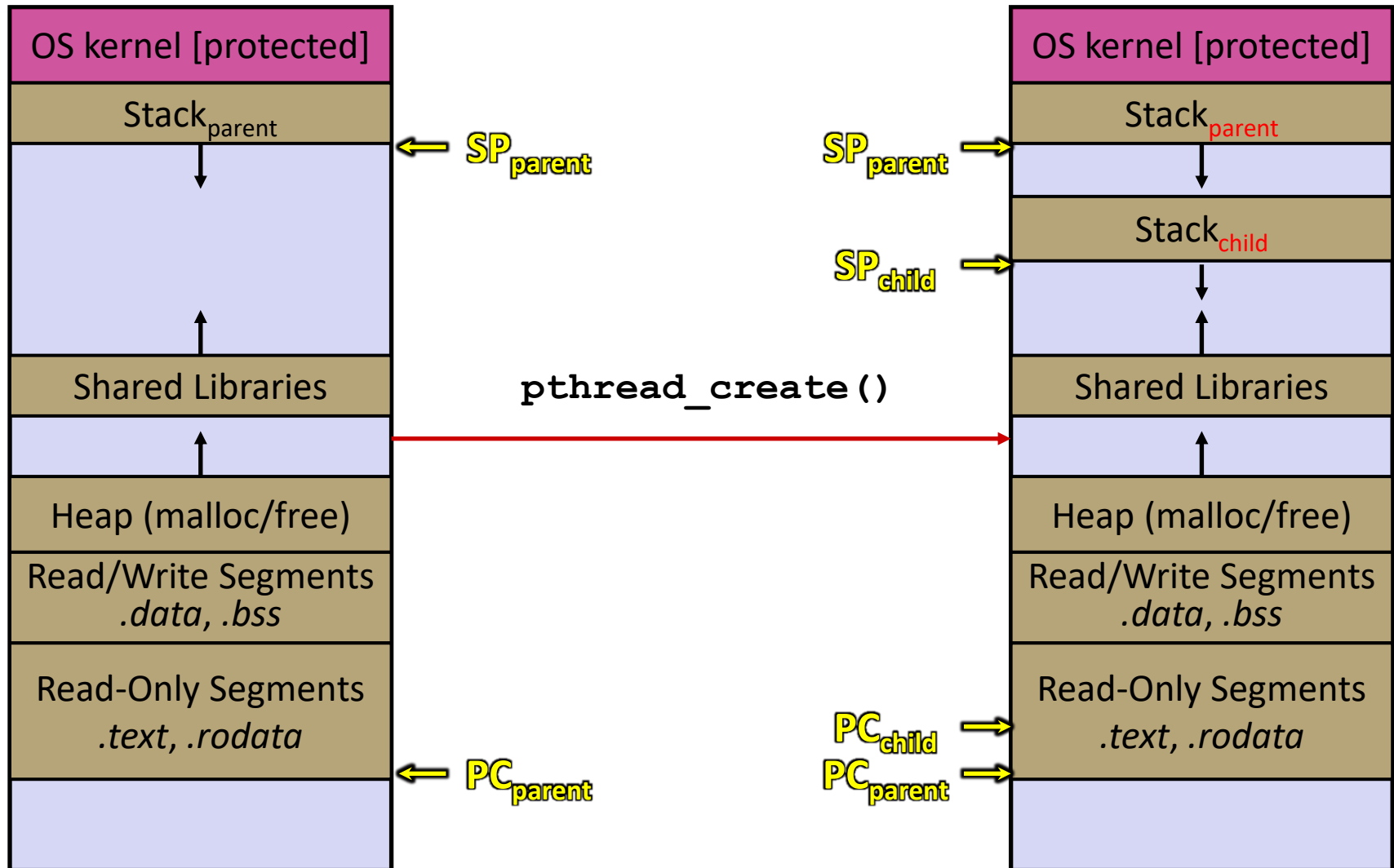
Threads vs. Processes



Before

After

Threads vs. Processes



Before

After

Multi-threaded Search Engine (Pseudocode)

```
fn main():  
  while (true):  
    string[] query_words = GetNextQuery()  
    CreateThread(job=ProcessQuery, input=query_words)
```

↳ returns

```
fn Lookup(string word) -> doclist:  
  bucket = hash(word)  
  hitlist = file.read(bucket)  
  foreach hit in hitlist  
    doclist.append(file.read(hit))  
  return doclist  
  
fn ProcessQuery(string[] query_words):  
  results = Lookup(query_words[0])  
  foreach word in query[1..n]:  
    results = results.intersect(Lookup(word))  
  Display(results)
```

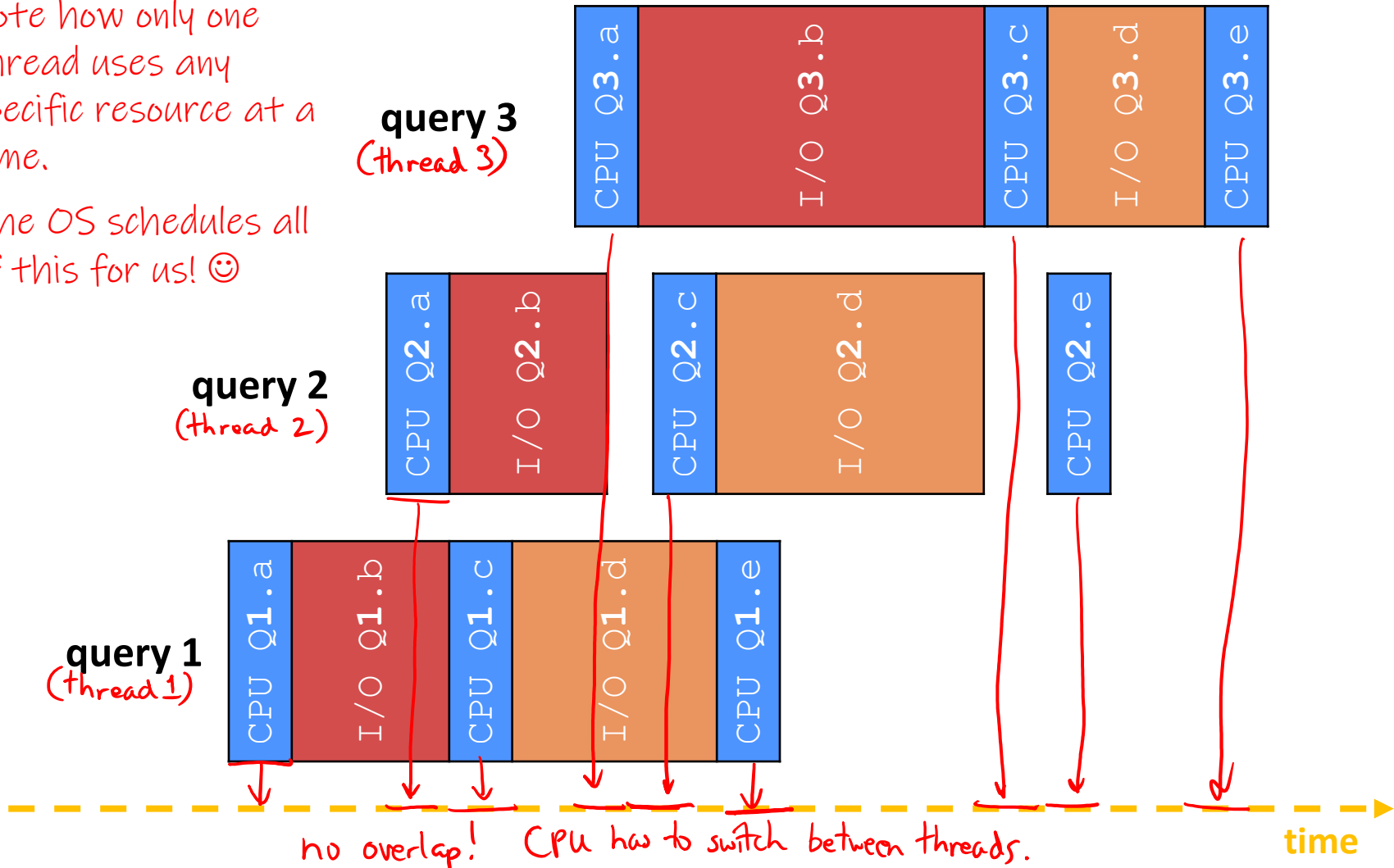
All we did was put the code into a function, and create a thread that invokes it!

Multi-threaded Search Engine (Execution)

(still one CPU)

Note how only one thread uses any specific resource at a time.

The OS schedules all of this for us! 😊



Why Threads?

❖ Advantages:

- You (mostly) write sequential-looking code
- Less overhead than processes during creation and context switching
- Threads can run in parallel if you have multiple CPUs/cores

OS does this automatically

❖ Disadvantages:

- If threads share data, you need **locks** or other synchronization
 - Very bug-prone and difficult to debug
- Threads can introduce overhead
 - Lock contention, context switch overhead, and other issues
- Need language support for threads



Poll Everywhere

pollev.com/cse333a



Which of the following statements about threads and processes are TRUE?

A. Each thread has its own Stack and ~~Heap~~

Yes

No

B. Each thread runs on a separate CPU core

C. Threads within the same process share file descriptors

D. Parent and child processes initially share file descriptors

Clone, including fd's

E. We're lost...

Alternative 1: Non-blocking I/O

- ❖ Reading from the network can truly *block* your program
 - Remote computer may wait arbitrarily long before sending data
- ❖ Non-blocking I/O (network, console)
 - Your program enables non-blocking I/O on its file descriptors *O_NONBLOCK: seefcntl.h*
 - Your program issues **read()** and **write()** system calls
 - If the read/write would block, the system call returns immediately
 - Program can ask the OS which file descriptors are readable/writable *(errno == EWOULDBLOCK/EAGAIN)*
 - Use **poll()**, **epoll()** (Linux), or **kqueue()** (macOS)
 - Set a timeout parameter
 - 0 ms = non-blocking
 - 100 ms = short blocking
 - -1 ms = block until ready

Alternative 2: Async I/O via Events

- ❖ Using **asynchronous** I/O, your program (almost never) *blocks* on I/O
- ❖ **Event-driven** programming is the main paradigm for async
 - When you need to read more data, you register interest in the data with the OS and then switch to a separate task
 - The OS handles the details of issuing the read from the disk, network, console, etc.
 - When data becomes available, the OS lets you know by delivering an **event**
 - It is common to register interest by storing a pointer to the function you would like to run when the event happens
 - This is referred to as a “**callback function**”

“When data is ready, run my callback function”

Event-Driven Programming (Pseudocode)

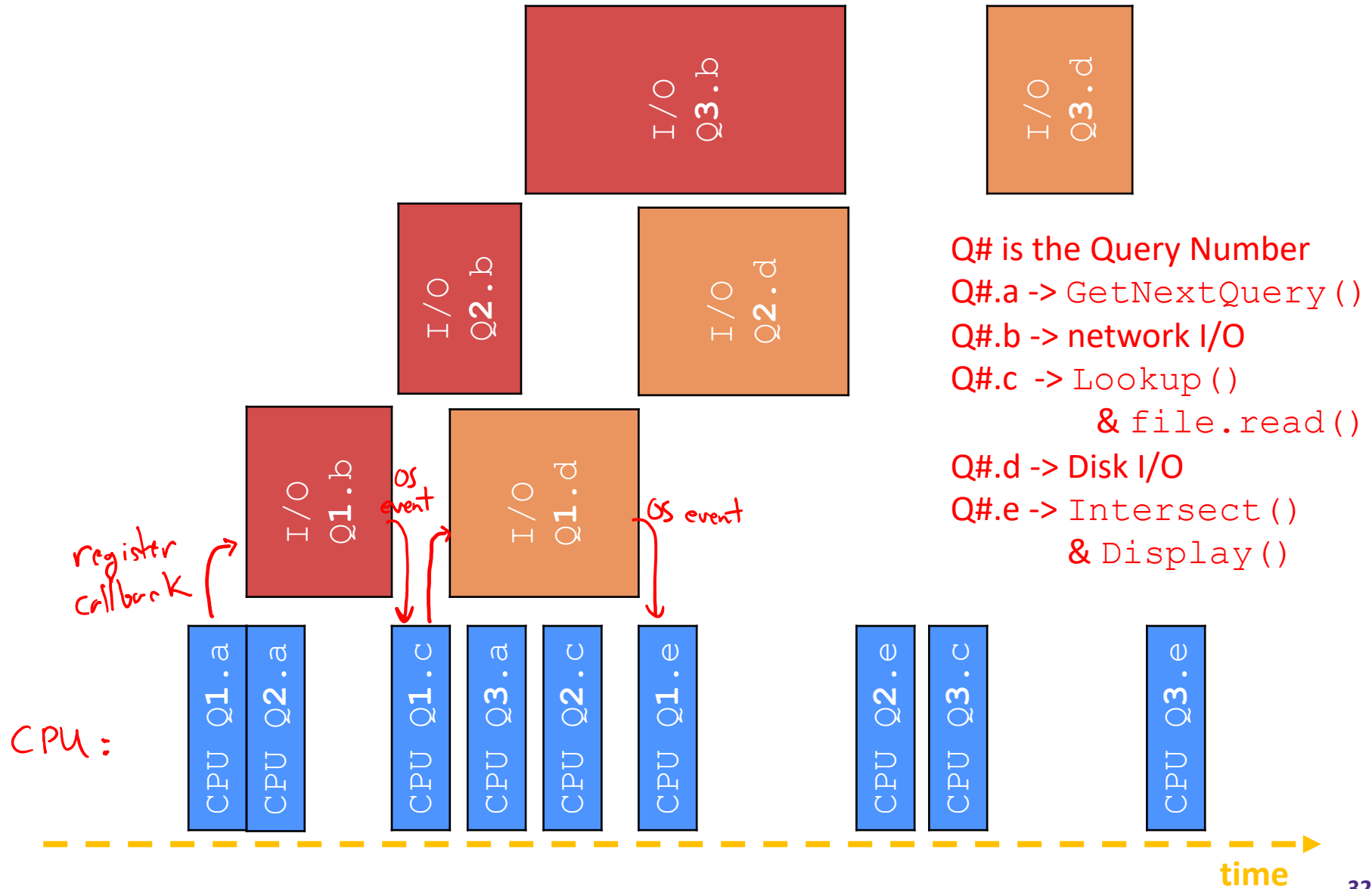
```
let fd_group = event_group()

fn main():
  let listener_fd = listen("localhost:8080")
  fd_group.register_callback(listener_fd, AcceptClient)
  "when ready to accept"
  while(true):
    let event = event_poll_wait(fd_group) blocks until next event
    event.callback(event.fd, event.arg)

fn AcceptClient(listener_fd):
  let connection_fd = accept(listener_fd) guaranteed not to block
  fd_group.register_callback(connection_fd, HandleRequest)
  "when ready to read from conn"

fn HandleRequest(connection_fd):
  let request_text = read(connection_fd) guaranteed not to block
  let filename = extract_filename(request_text)
  let file_fd = open(filename)
  let file_body = read(file_fd) will block
  write(connection_fd, file_body)
  close(file_fd)
```

Asynchronous, Event-Driven



Why Events?

❖ Advantages:

- Don't have to worry about locks and data races
- For some kinds of programs, especially GUIs, leads to a very simple and intuitive program structure
 - One event handler for each UI event

Java Script

❖ Disadvantages:

- Can lead to complex structure for programs that do lots of network I/O
 - Sequential code gets broken up into a jumble of small event handlers
 - You have to package up all task state between handlers
 - “Callback Hell”, “Callback Pyramid”

Event-Driven vs. Threads



- ❖ ["Node.js Is Bad Ass Rock Star Tech" \(2012\)](#) [1:15-2:15]
- ❖ *"[JavaScript] uses a single-threaded loop that dispatches events to handlers. It's a proven model that solves some concurrency problems, but at the cost of **code complexity**..."*
- ❖ *[You must be] on the lookout for blocking operations that should be **split into little pieces**, each perfectly tailored to optimize throughput...*
 - *Do you know what this reminds me of?... The invention of threads.*
- ❖ *Threading libraries do exactly what you are doing manually. They break up pieces of code to be executed, intermittently switching from instructions that are waiting on I/O to instructions that are ready to run, but your sequential code stays intact. **You may recall sequential code. That's the code you can read.**"*

Outline (next two lectures)

- ❖ We'll look at different `searchserver` implementations
 - Concurrent via dispatching threads – `pthread_create()`
 - Introduced in section tomorrow!
 - Concurrent via forking processes – `fork()`

- ❖ Reference: *Computer Systems: A Programmer's Perspective*, Chapter 12 (CSE 351 book)