

pollev.com/naomila

Tell me one thing you remember about networking from last week (:

CSE333 Systems Programming

Intro to Concurrency

Instructor:

Naomi Alterman

Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister

Advay Patil

Aviel Wood

Angela Wu

Jennifer Xu

Administrivia

- ❖ Networking exercises due this week (Ex10 just now, Ex11 this Friday)
- ❖ Concurrency exercise and HW4 will be due next week
- ❖ Then the final exam, and you're done!

Lecture Outline

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

20 years later...

- ❖ World has changed since HTTP/1.1 was adopted
 - Web pages were a few hundred KB with a few dozen objects on each page, now several MB each with hundreds of objects (JS, graphics, ...) & multiple domains per page
 - Much larger ecosystem of devices (phones especially)
 - Many hacks used to make HTTP/1.1 performance tolerable
 - Multiple TCP sockets from browser to server
 - Caching tricks; JS/CSS ordering and loading tricks; cookie hacks
 - Compression/image optimizations; splitting/sharding requests
 - etc., etc. ...

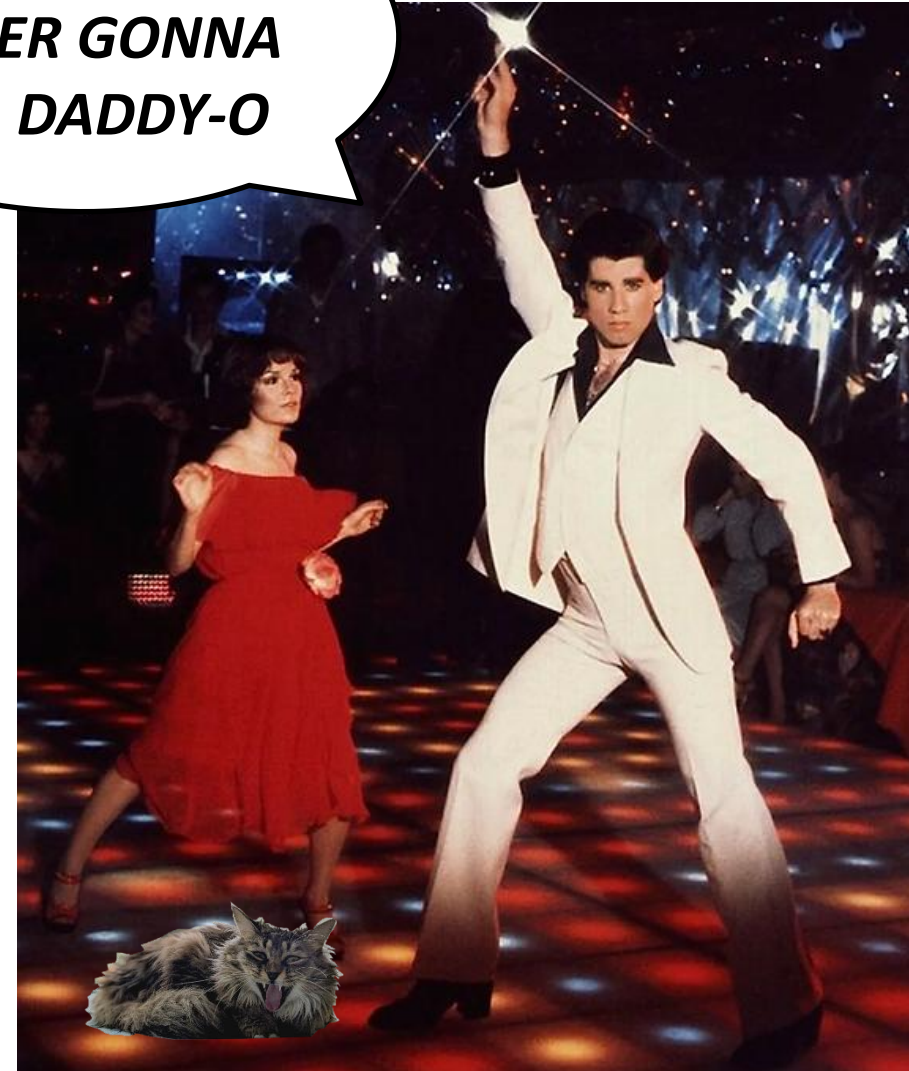
Beyond HTTP/1.x

- ❖ HTTP/2 standardized in 2015
 - Based on Google's "SPDY" protocol
 - Binary, not plaintext!
 - Multiple data streams "multiplexed" on single TCP connections
- ❖ HTTP/3 standardized in 2022
 - Switched from TCP to QUIC (a new transport layer protocol from Google)
 - Does multiplexing better (TCP was holding us baaaack 🙄)
- ❖ All use same core request/response model (URLs and methods like GET, POST, ...)
- ❖ All existing implementations incorporate TLS encryption (https)
- ❖ Widely adopted
 - All major browsers
 - 30% of websites speak HTTP/3
 - 60% speak HTTP/2
 - All speak HTTP/1.1

Beyond sockets?

- ❖ Sockets are a 40 year old API
 - Is there anything we'd have done differently in retrospect?
 - Are operating systems the same as they were in the 1970s?
 - Is the *internet* the same as it was in the 1970s?
- ❖ In practice, a shocking amount of the internet landscape has stayed the same
 - *It is a profoundly well engineered set of systems!!*

**AF_INET IS
NEVER GONNA
DIE, DADDY-O**




The modern internet

- ❖ In practice, a few important differences (beyond programming language and scale):
 - We **maintain sessions across connections** *all the time*, and expect continuous service
 - **Privacy and security** are more important than ever
 - **Metered** connections (connections on which you pay by-the-byte) are much more common

Apple's Network.framework

- ❖ An OS API for TCP/UDP-level network access that stands next to sockets
 - Includes affordances for modern network usage
 - <https://developer.apple.com/videos/play/wwdc2018/715/>
- ❖ Why now?
 - Most non-systems code does networking at the “application” layer of the OSI model and doesn't actually touch the sockets API
 - This makes it *much easier* to change the sockets abstraction
(*Fewer codebases to update, fewer angry developers to placate*)

Higher level transports

- ❖ A lot of internet traffic is **facilitated** by HTTP connections, if not directly transported over them
 - Eg: you open a zoom call using a web URL, but the call itself is peer-to-peer over UDP ()
- ❖ **Websockets** (circa 2011):
 - https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
 - Application-level communication
 - Uses HTTP for **connection setup** and TCP for **transport**
 - An “easy byte pipe”
 - Case study: <http://slither.io/>

Careful about reinventing the wheel

- ❖ I've seen a lot of networking research over the past decade and a half
 - And *a lot* of the same mistakes get made now that were getting made back then
- ❖ To re-invent an architecture, you need to understand it **as deeply as the people who built it**
 - Otherwise you're just pursuing an exercise in vanity
- ❖ (*"okay, thanks grandma :P"*)

Lecture Outline

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

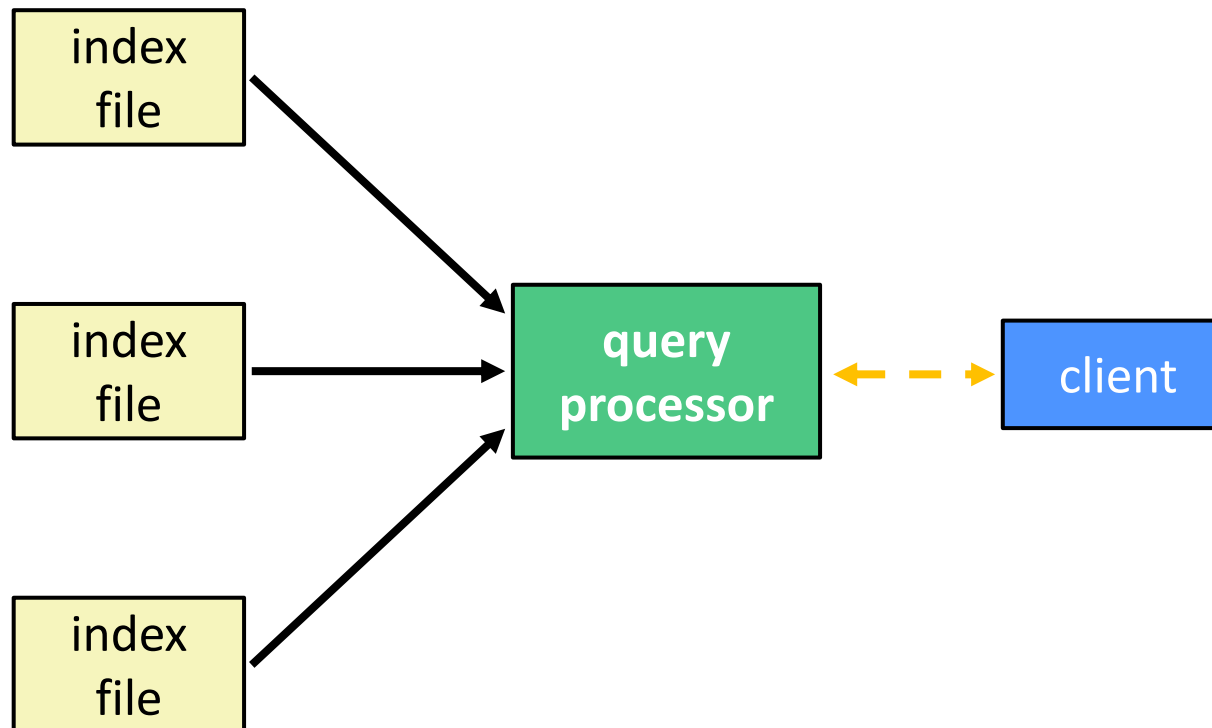
hw4 demo

- ❖ Multithreaded Web Server (333gle)
 - Don't worry – multithreading has mostly been written for you
 - `./http333d PORT STATIC_DIR INDEX_FILE+`
 - Some security bugs to fix, too

Building a Web Search Engine

- ❖ We have:
 - Some indices
 - A map from *<word>* to *<list of documents containing the word>*
 - This is probably *sharded* over multiple files
 - 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)
```

Full implementation in [searchserver_sequential/](#)

Why Sequential?

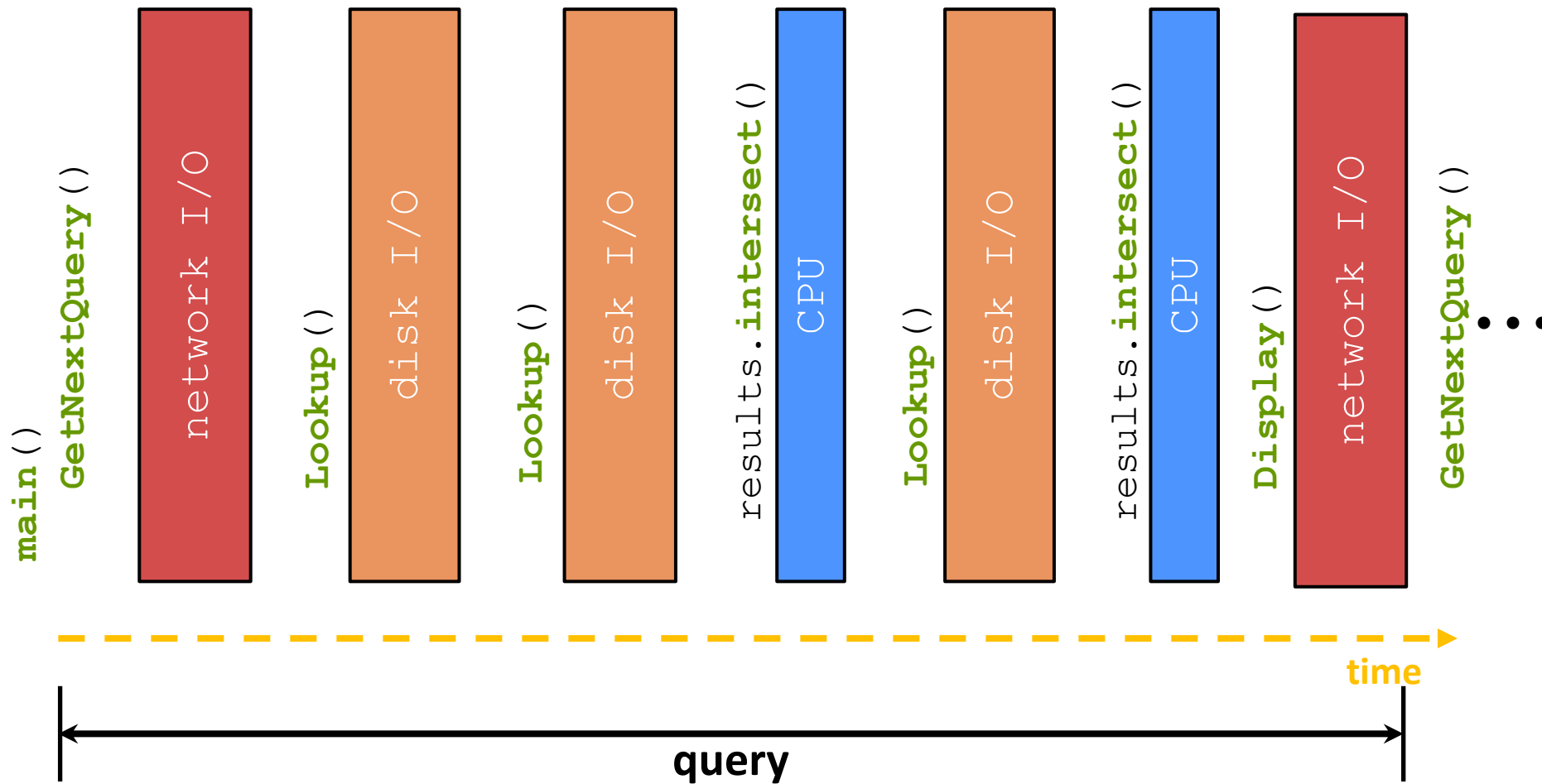
❖ Advantages:

- Super simple (hahaha 🤔) 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



What About I/O-caused Latency?

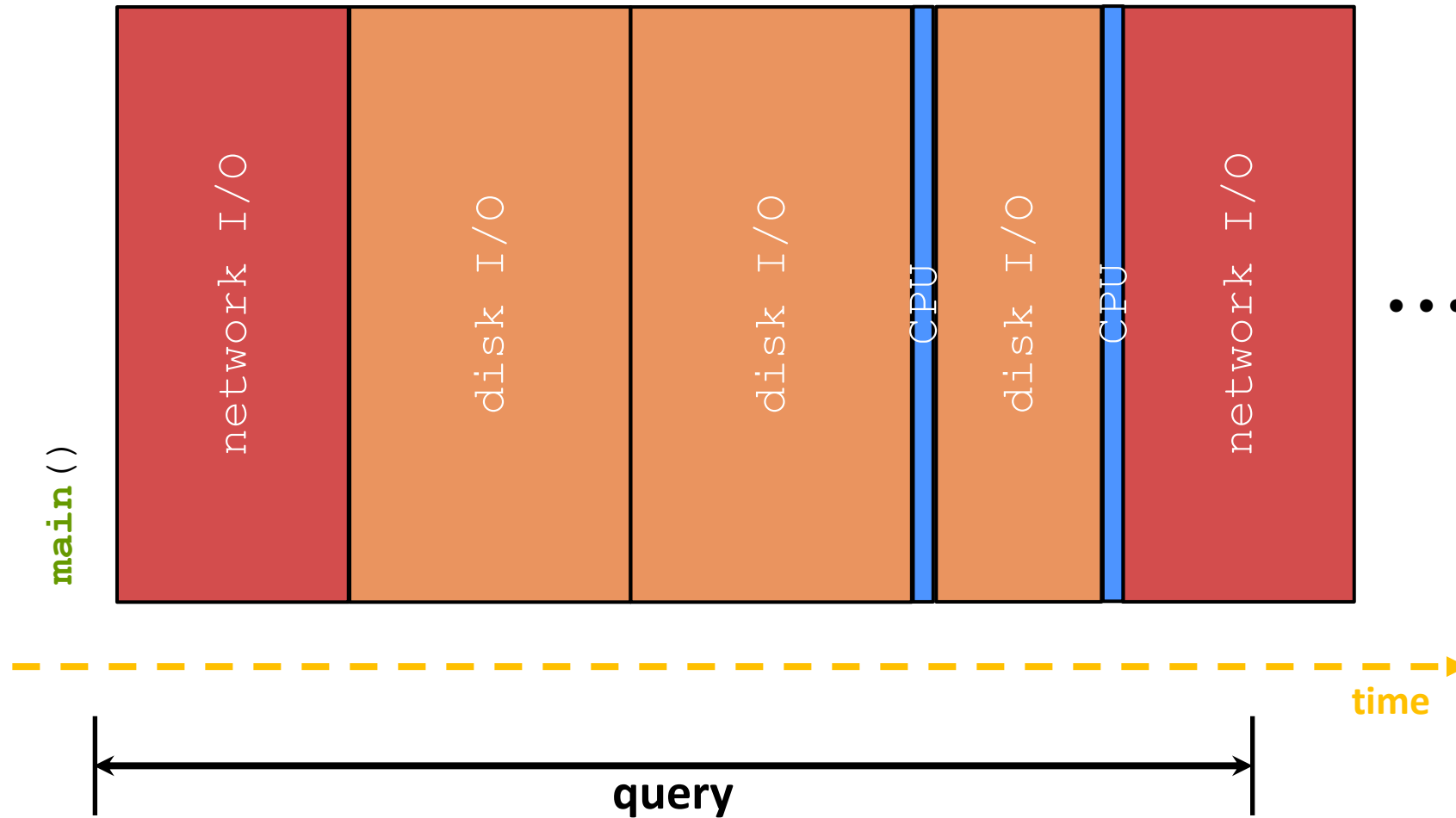
- ❖ Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

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



Google

Execution Timeline: (Loosely) To Scale



Multiple (Single-Word) Queries

Q# is the Query Number

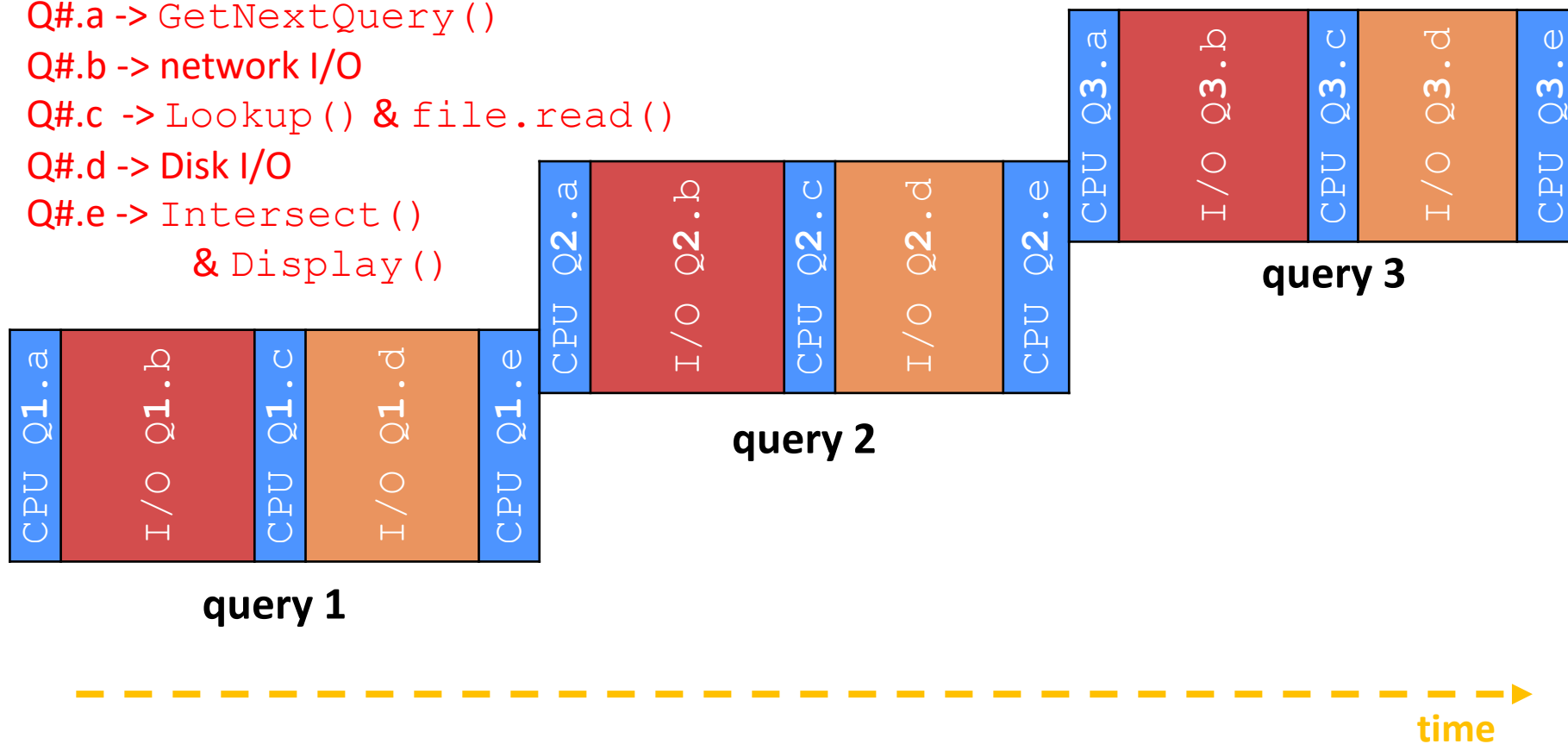
Q#.a -> `GetNextQuery()`

Q#.b -> network I/O

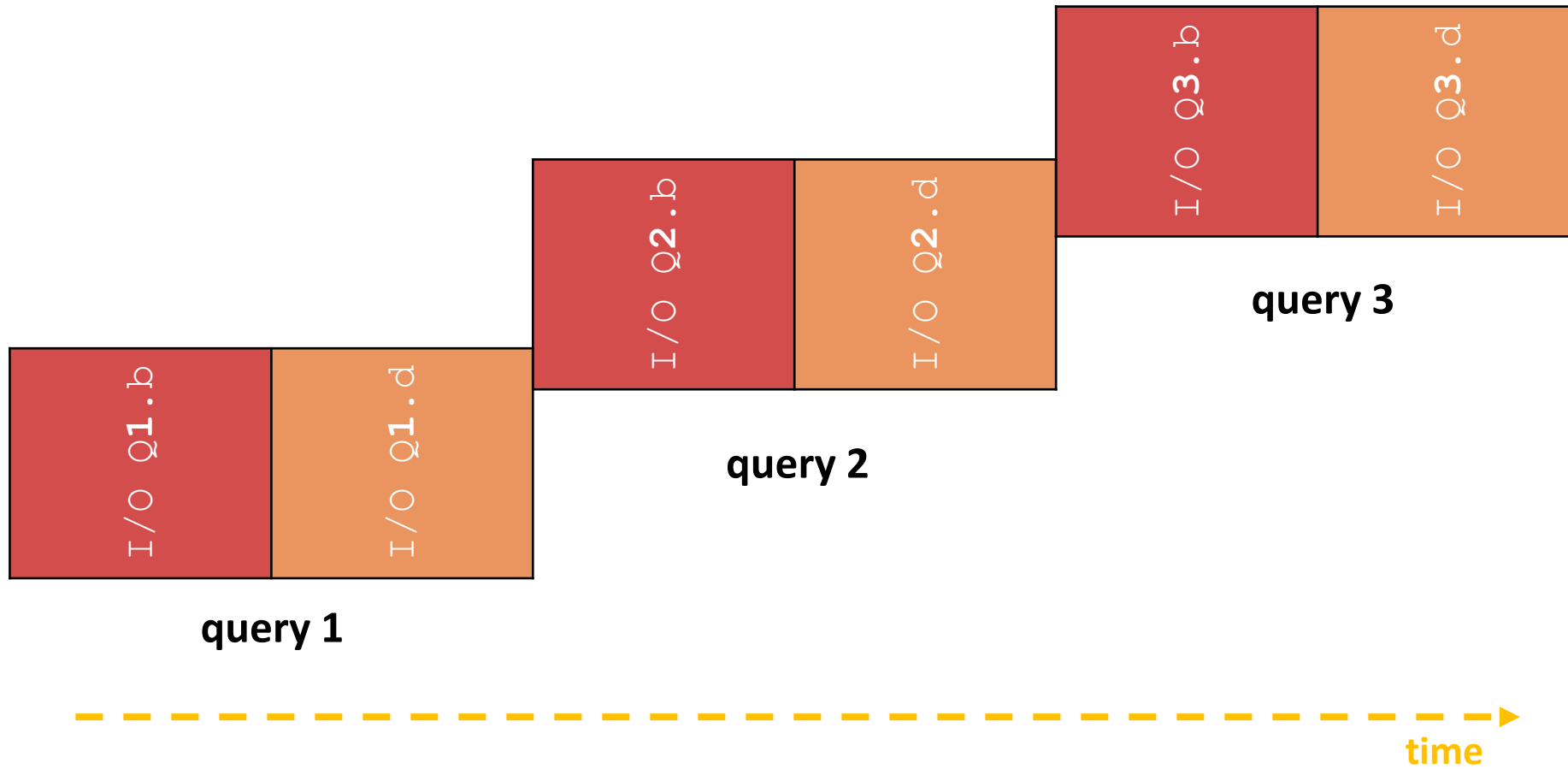
Q#.c -> `Lookup()` & `file.read()`

Q#.d -> Disk I/O

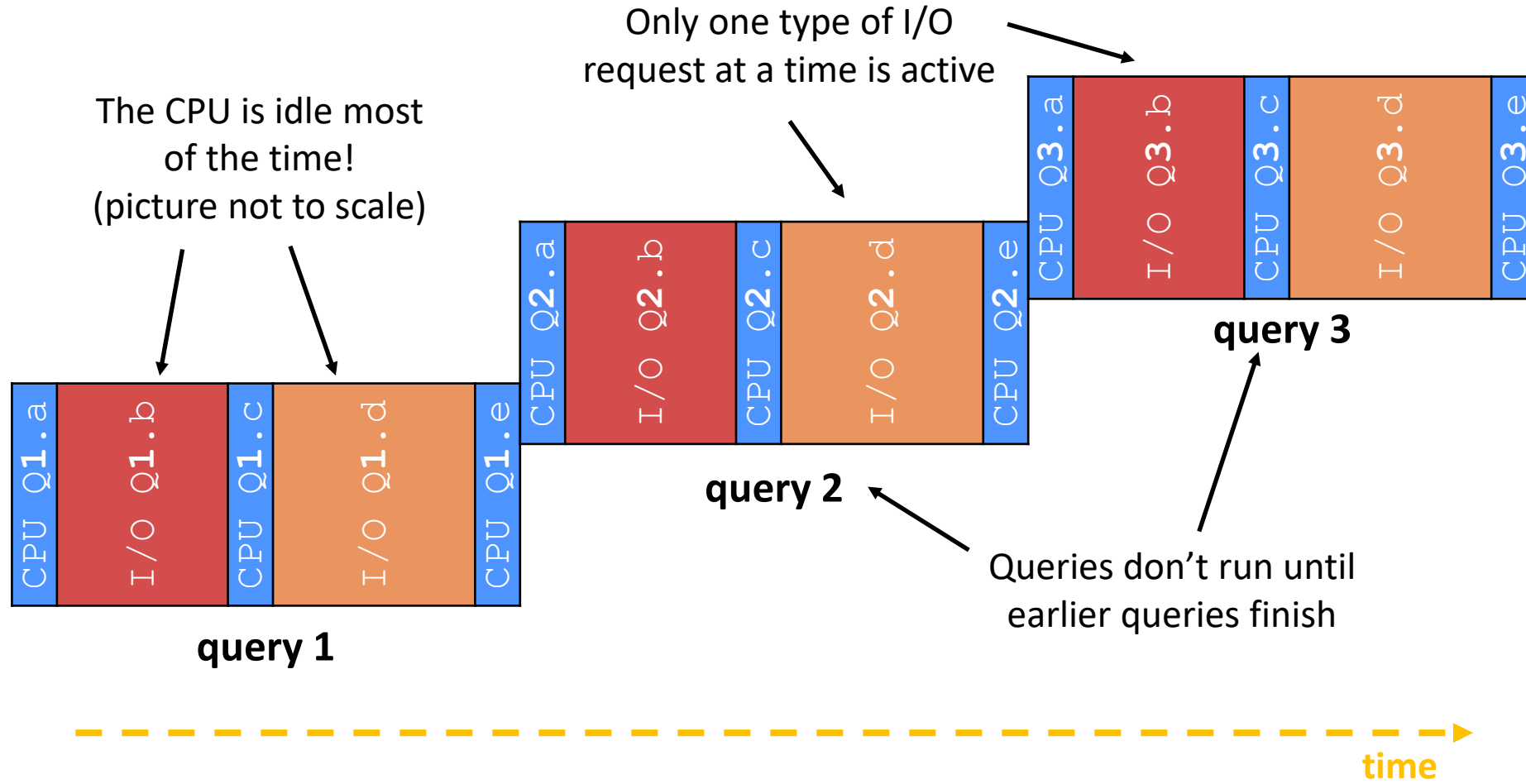
Q#.e -> `Intersect()`
& `Display()`



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

- ❖ Beyond Sockets
- ❖ 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*
- ❖ 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
 - Open resources such as file descriptors
 - 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:

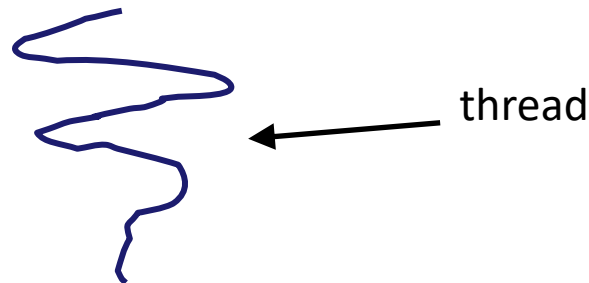
- Processes are isolated from one another
 - No shared memory between processes
 - If one crashes, the other processes keep going
- No need for language support (OS provides `fork`)

❖ Disadvantages:

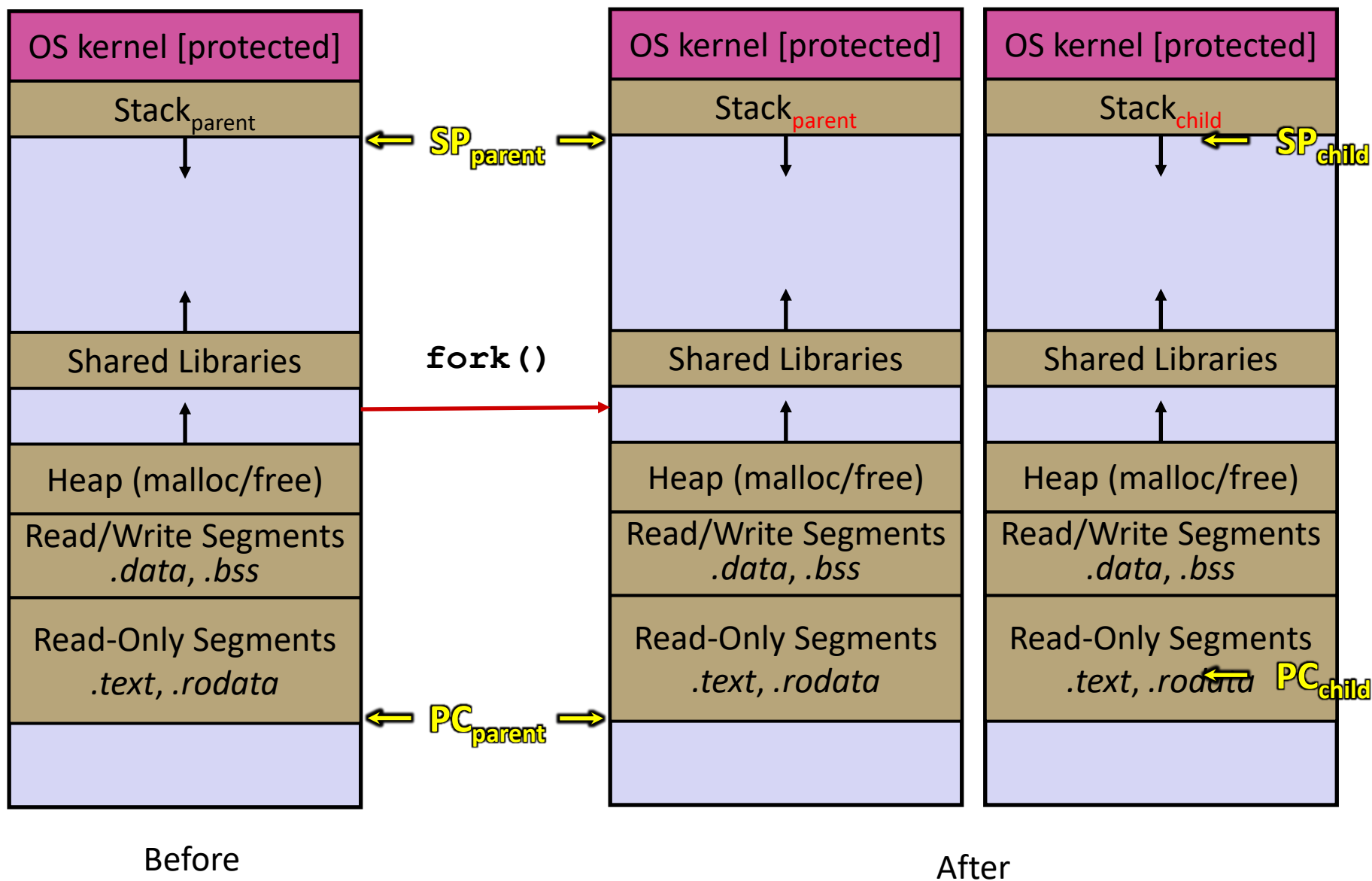
- A lot of overhead during creation and context switching
- Cannot easily share memory between processes
 - Typically must communicate through the file system
 - *e.g.*, POSIX pipe/FIFO: one way data stream between two processes

Worker Option 2: Threads

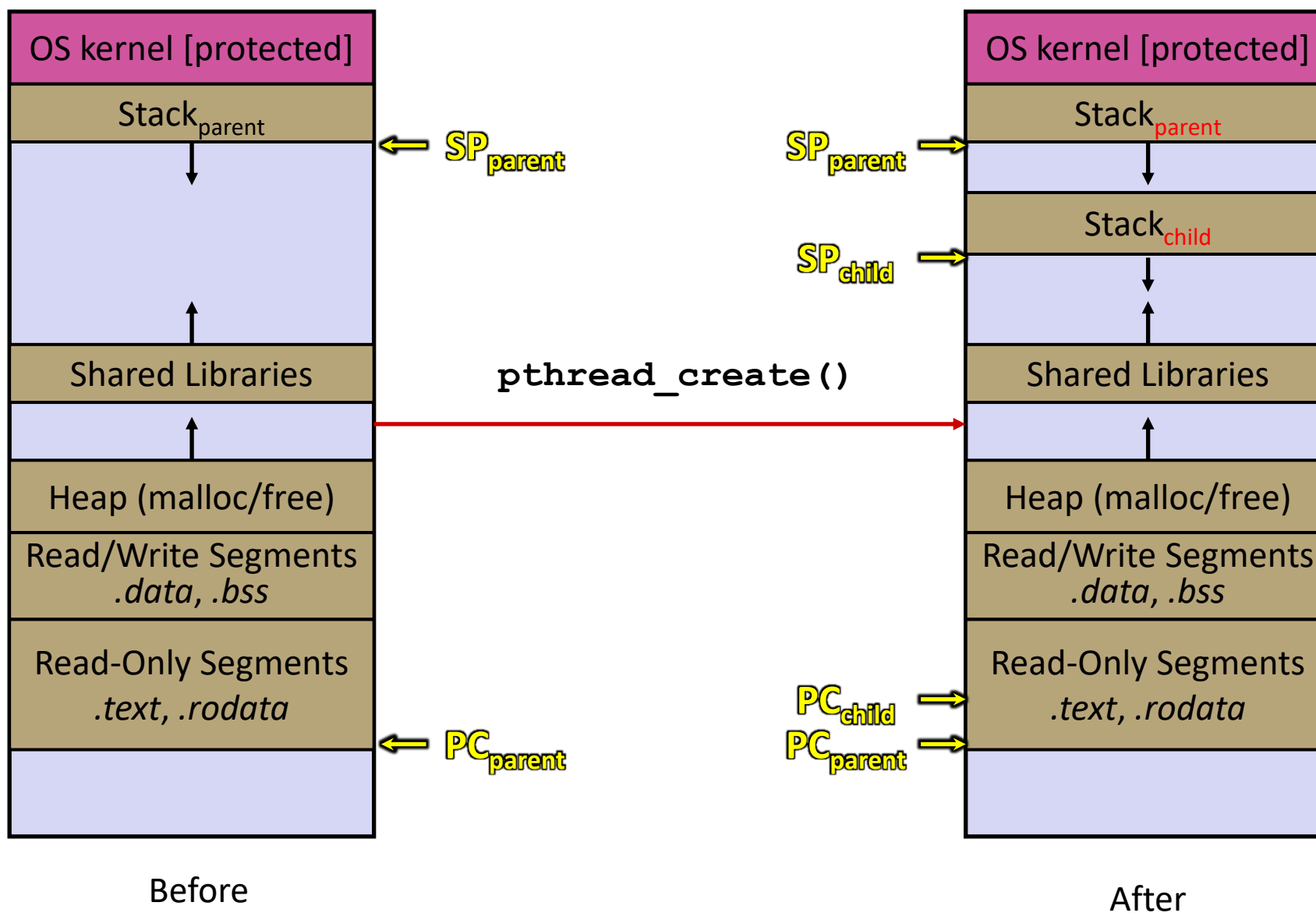
- ❖ From within a process, we can separate out the concept of a “thread of execution” (**thread** for short)
 - Processes are the containers that hold shared resources and attributes
 - *e.g.*, address space, file descriptors, security attributes
 - Threads are independent, sequential execution streams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



Processes in memory



Threads in memory



Multi-threaded Search Engine (Pseudocode)

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

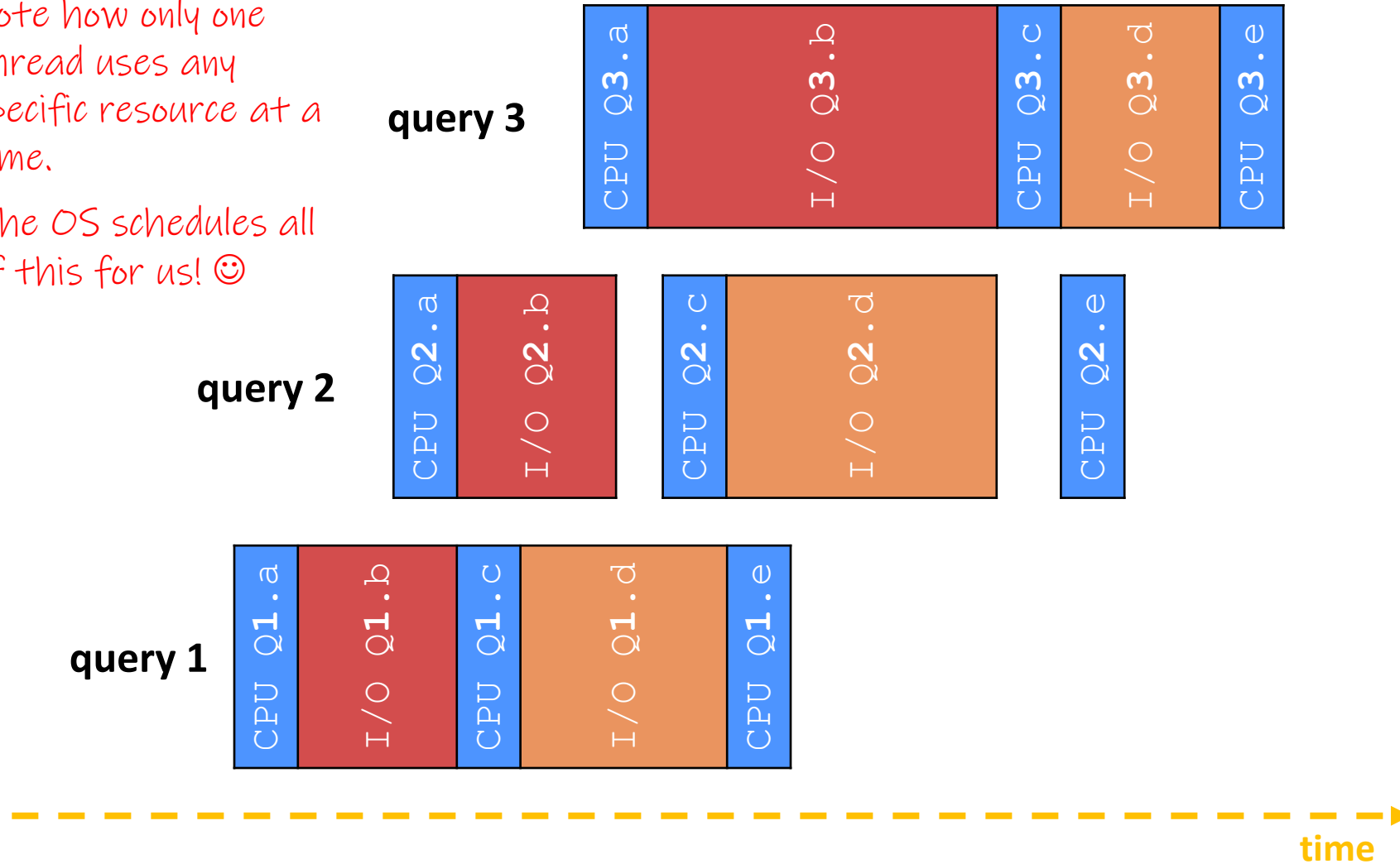
```
fn Lookup(string word) -> doclist:  
  bucket = hash(word)  
  hitlist = file.read(bucket)  
  foreach hit in hitlist  
    doclist.append(file.read(hit))  
  return doclist  
  
fn ProcessQuery(string[] query_words):  
  results = Lookup(query_words[0])  
  foreach word in query[1..n]:  
    results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

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

❖ 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/naomila



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

- A. Each thread has its own Stack and Heap
- B. Each thread runs on a separate CPU core
- C. Threads within the same process share file descriptors
- D. Parent and child processes do not share file descriptors
- E. Future me will know this, present me is lost