

Concurrency: Intro and Threads

CSE 333 Autumn 2021

Instructor: Chris Thachuk

Teaching Assistants:

Arpad (John) Depaszthory

Ian Hsiao

Logan Gnanapragasam

Mengqi (Frank) Chen

Angela Xu

Khang Vinh Phan

Maruchi Kim

Cosmo Wang




pollev.com/cse333

How far are you in HW4?

- A. I have passed the test_suite
- B. I've finished ServerSocket
- C. I'm working on ServerSocket right now
- D. I have read the spec and some of the code
- E. I haven't looked at it yet
- F. I didn't submit / I prefer not to say

Administrivia

- ❖ HW4 due 1 week from Thursday (Dec. 9)
- ❖ Exercise 15 due Wednesday (Dec. 1) @ 10am
- ❖ Exercise 16 to be released Wednesday
 - Due Wednesday Dec. 8 @ 10 am
 - **The Last Exercise** 
- ❖ Final exam will be take home; similar style as midterm
 - On gradescope, will close at 4:20pm on Dec. 15
 - **Open for multiple days** prior to deadline

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`

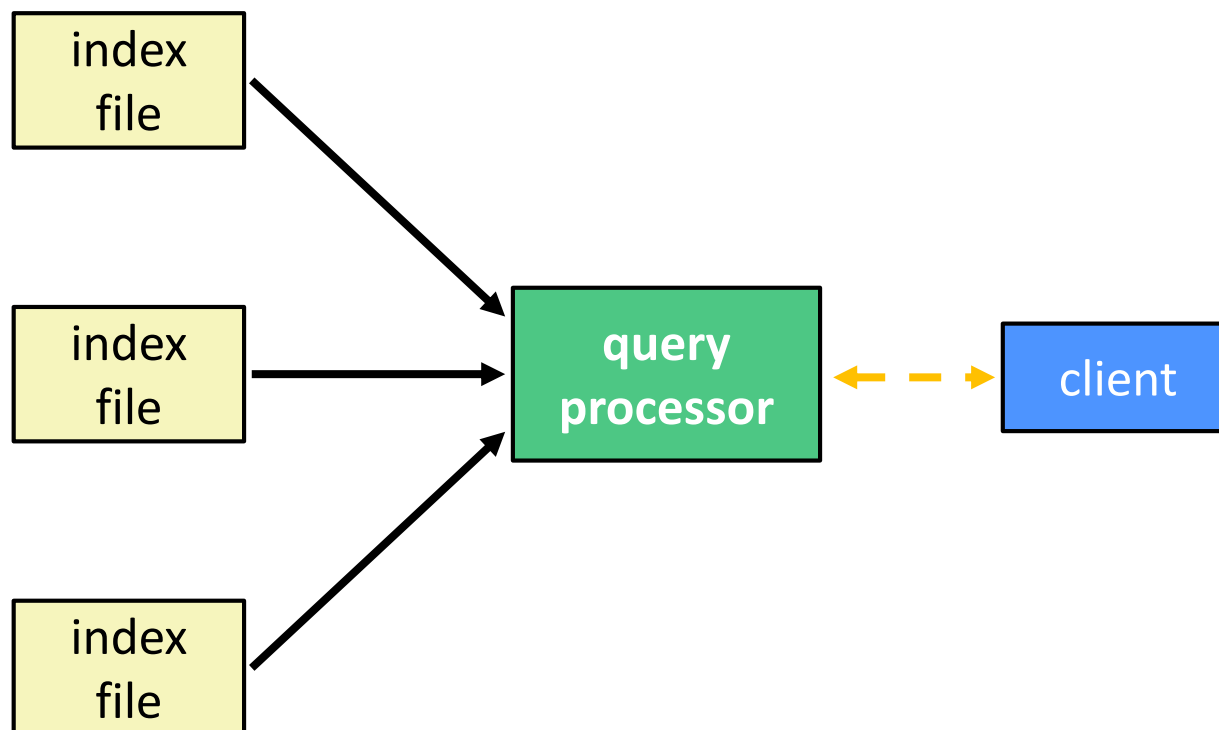
Lecture Outline

- ❖ **From Query Processing to a Search Server**
- ❖ Intro to Concurrency
- ❖ Threads and other 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
 - 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

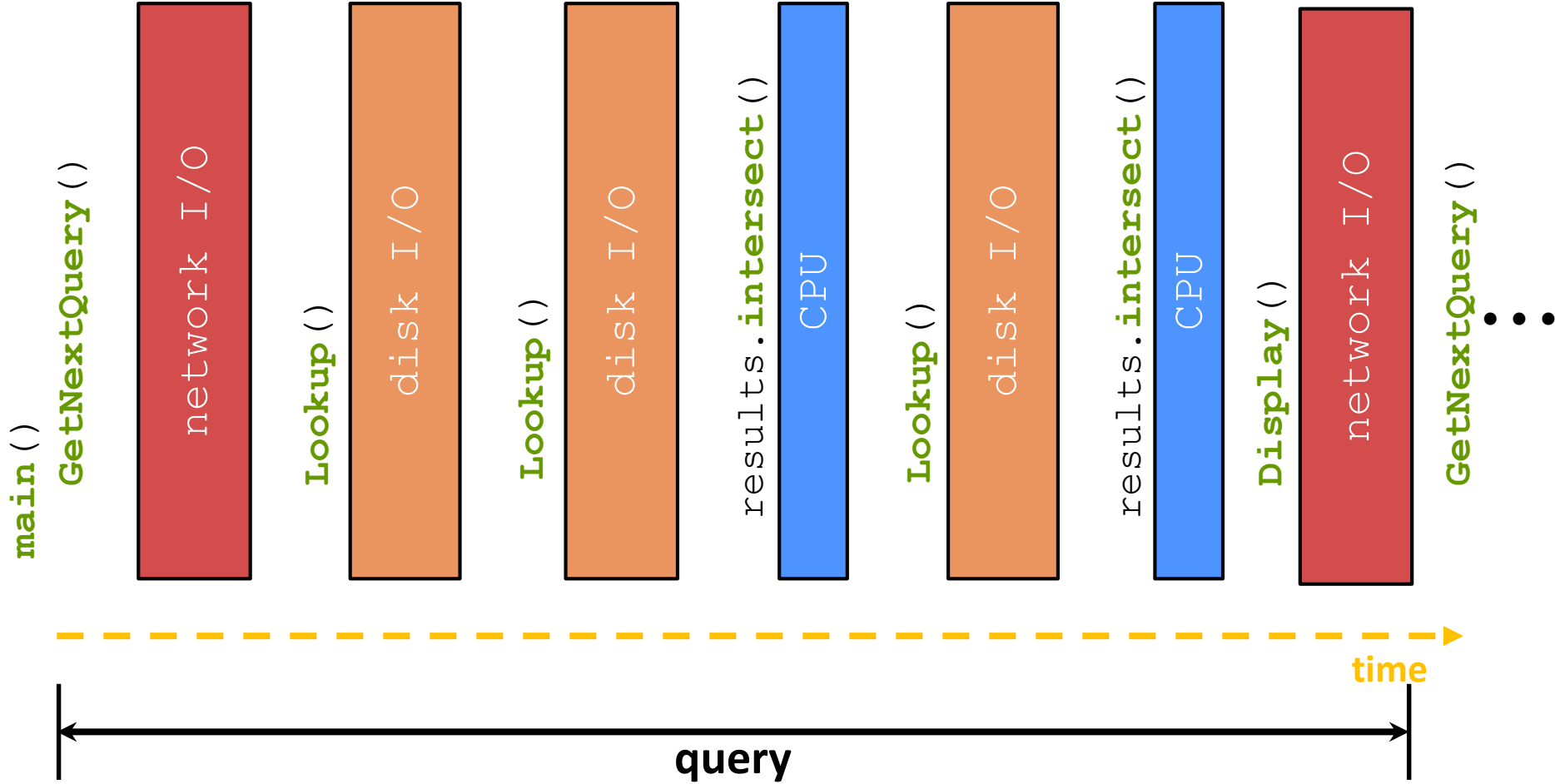


Search Engine (Pseudocode)

```
doclist Lookup(string word) {
    bucket = hash(word);
    hitlist = file.read(bucket); ← Disk I/O
    foreach hit in hitlist {
        doclist.append(file.read(hit)); ←
    }
    return doclist;
}

main() {
    SetupServerToReceiveConnections();
    while (1) {
        string query_words[] = GetNextQuery(); ← Network I/O
        results = Lookup(query_words[0]);
        foreach word in query[1..n] {
            results = results.intersect(Lookup(word));
        }
        Display(results); ← Network I/O
    }
}
```



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

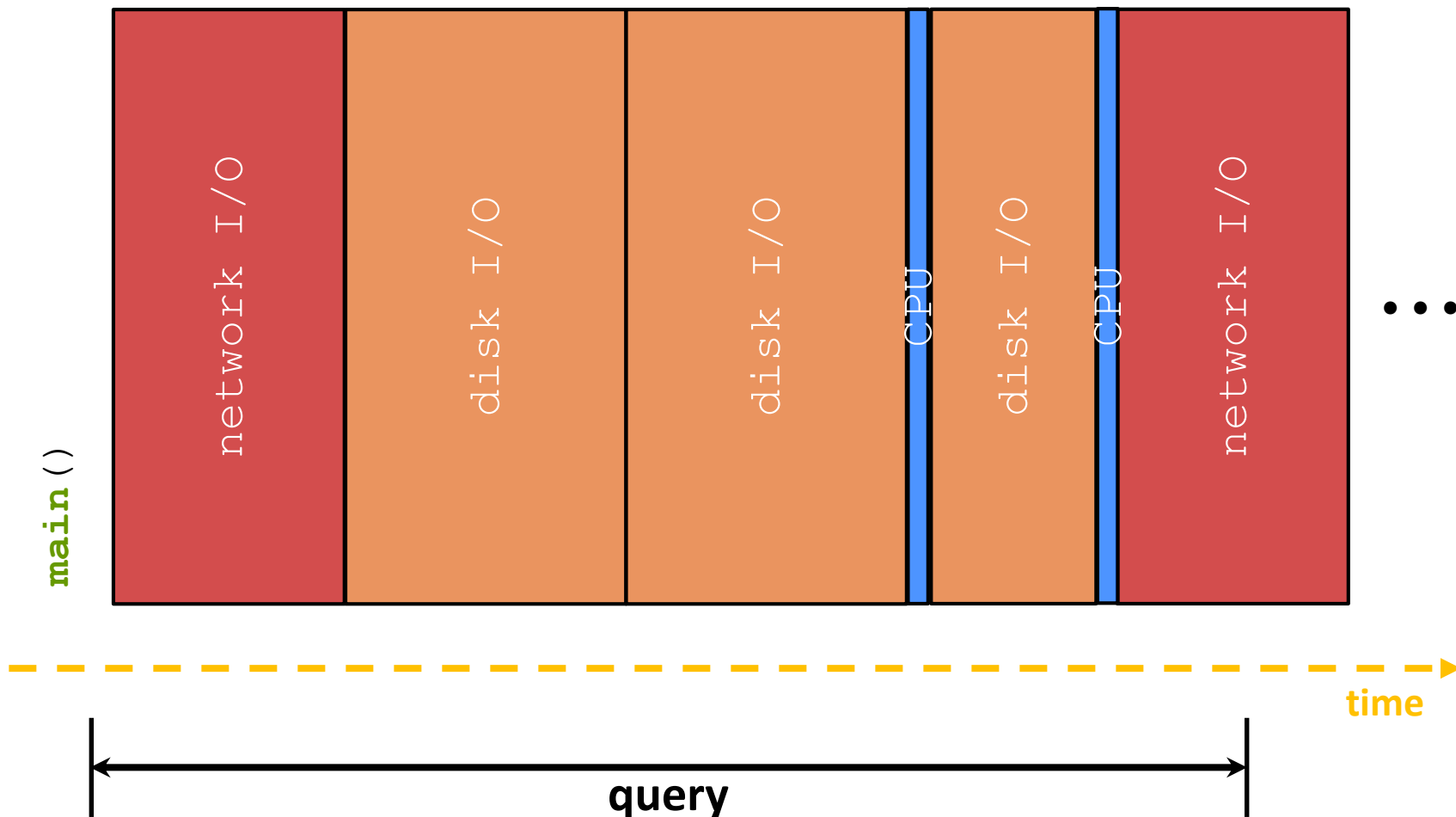


Execution Timeline: (Closer) To Scale

Model isn't perfect:

Technically also some cpu usage to setup I/O.

Network output also (probably) won't block program



Multiple (Single-Word) Queries

is the Query Number

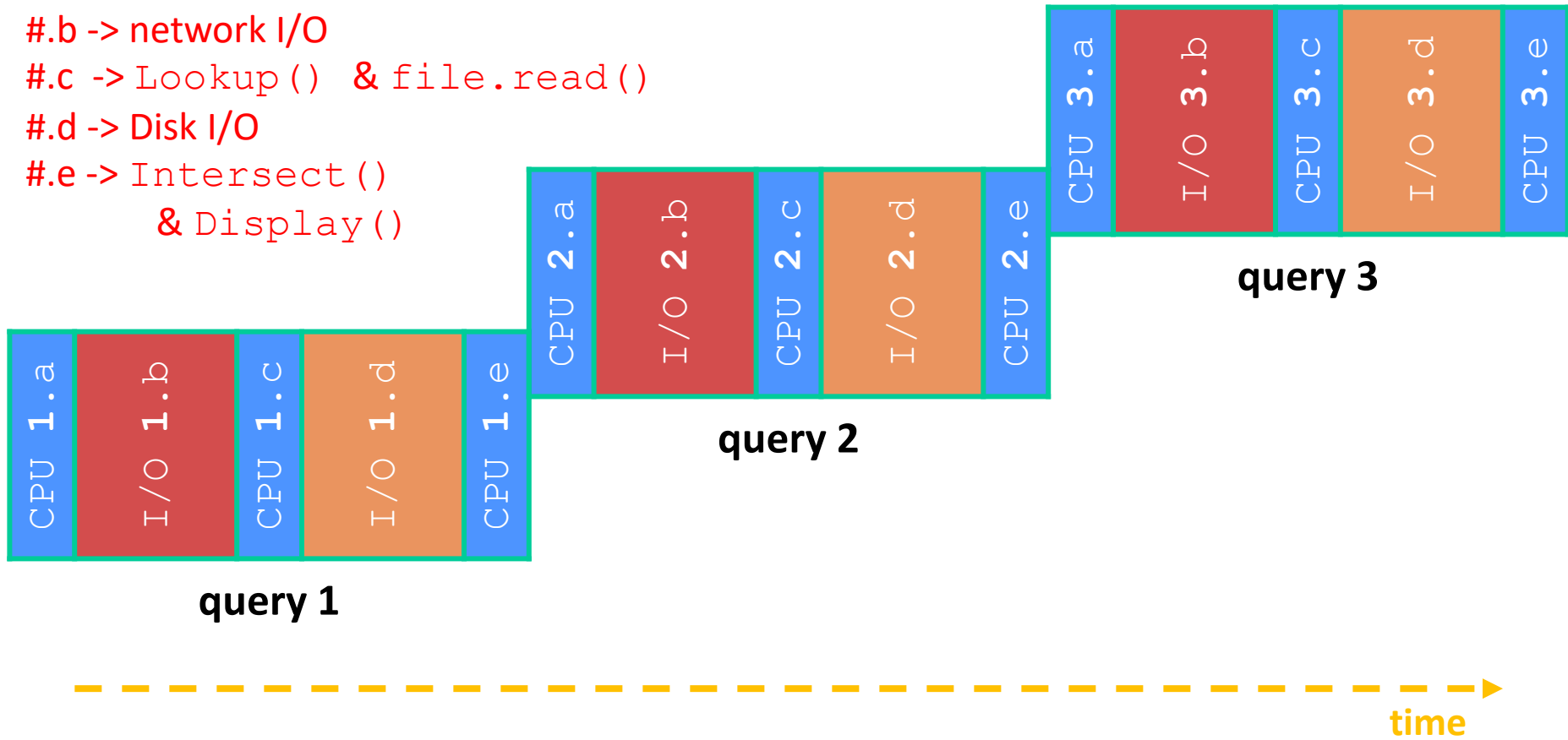
#.a -> GetNextQuery ()

#.b -> network I/O

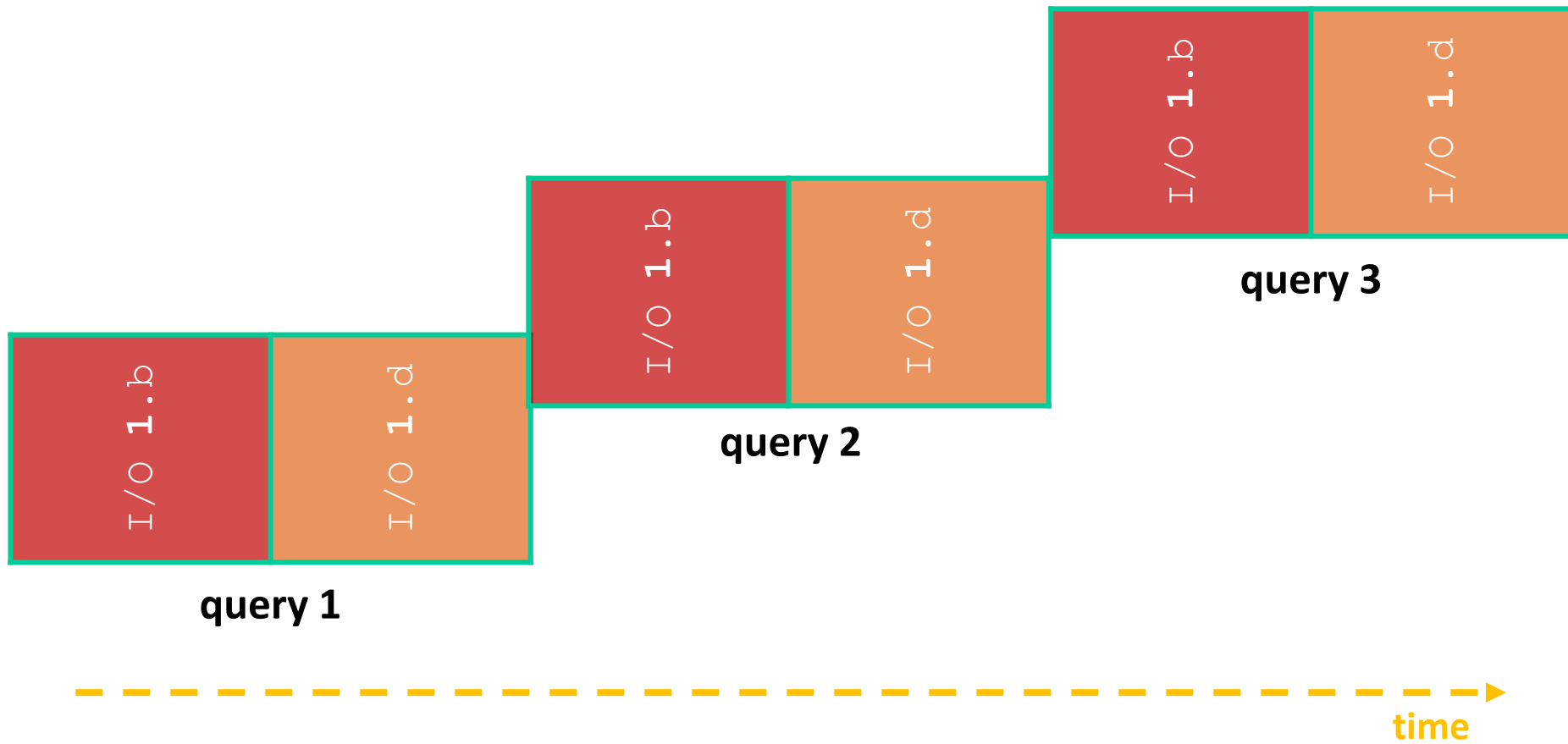
#.c -> Lookup () & file.read ()

#.d -> Disk I/O

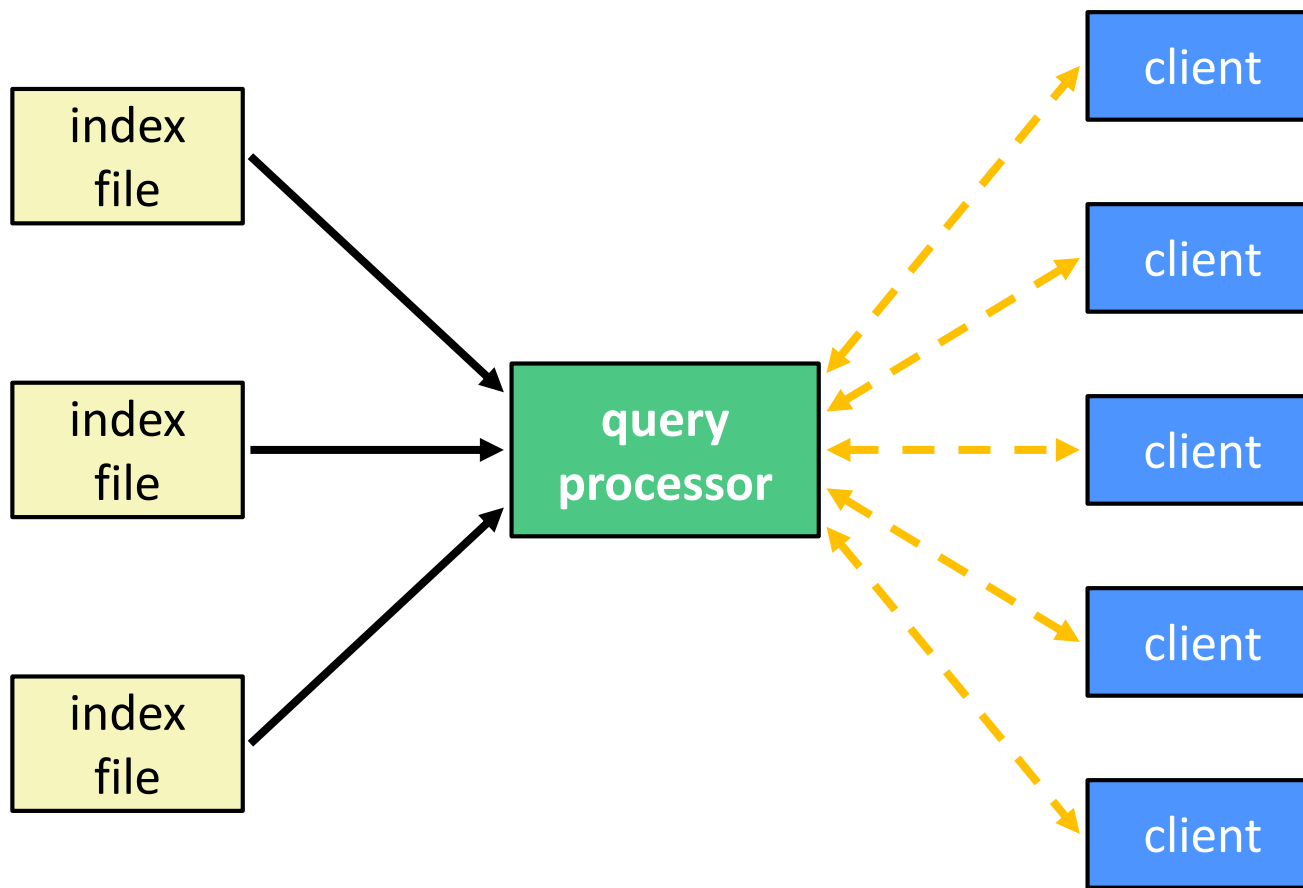
#.e -> Intersect ()
& Display ()



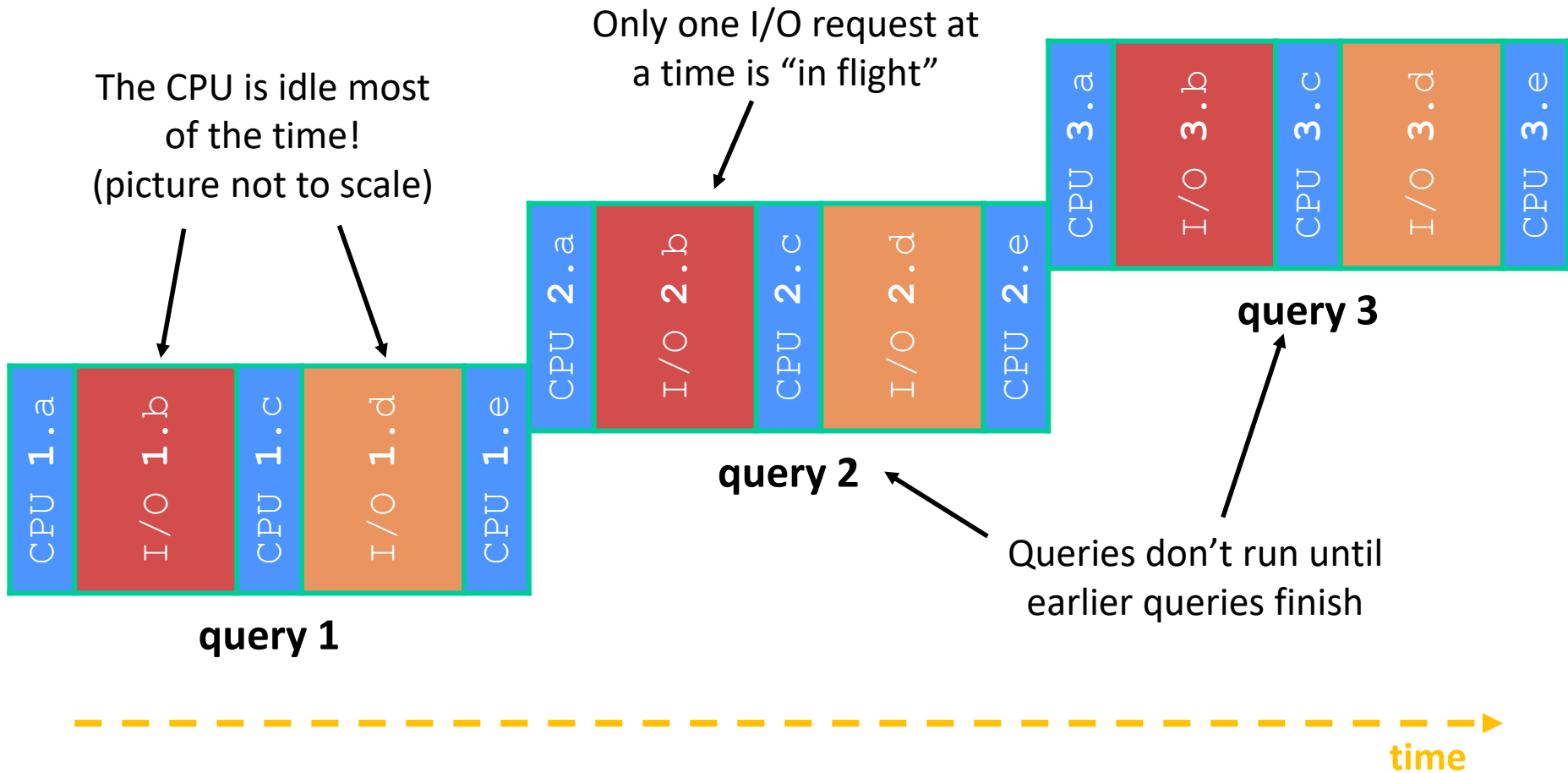
Multiple Queries: To Scale



Uh-Oh (1 of 2)



Uh-Oh (2 of 2)



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 I/O can be very, very slow (10 million times slower ...)
- ❖ At most one I/O operation is in flight 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
- ❖ **Intro to Concurrency**
- ❖ Threads and other concurrency methods

Concurrency

- ❖ Our search engine could run concurrently:
 - Example: Execute queries one at a time, but 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: Our web server could execute multiple *queries* at the same time
 - While one is waiting for I/O, another can be executing on the CPU
- ❖ Concurrency \neq parallelism
 - Concurrency is doing multiple tasks at a time
 - Parallelism is executing multiple CPU instructions *simultaneously*

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” uses blocking I/O; 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 “worker” I/O requests can be issued at once
- ❖ So what should we use for our “workers”?

Lecture Outline

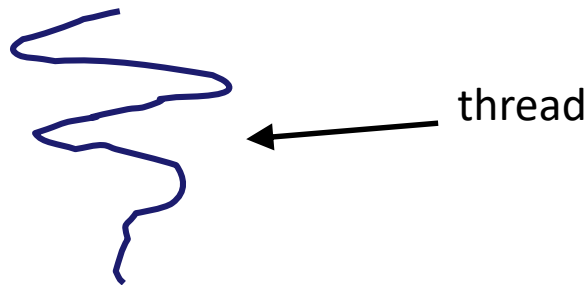
- ❖ From Query Processing to a Search Server
- ❖ Intro to Concurrency
- ❖ **Threads and other concurrency methods**

Review: Processes

- ❖ The components of a “process” are:
 - Resources such as file descriptors and sockets
 - An address space (page tables, etc.)
- ❖ Different Processes have independent components:
 - Most importantly: Isolated address spaces.
- ❖ An address space of a process can hold stack(s) that distinguish different “threads” of execution

Introducing Threads

- ❖ Separate the concept of a **process** from the “*thread of execution*”
 - Threads are contained within a process
 - Usually called a **thread**, this is a sequential execution stream within a process



- ❖ In most modern OS's:
 - Threads are the *unit of scheduling*.

Multi-threaded Search Engine (Pseudocode)

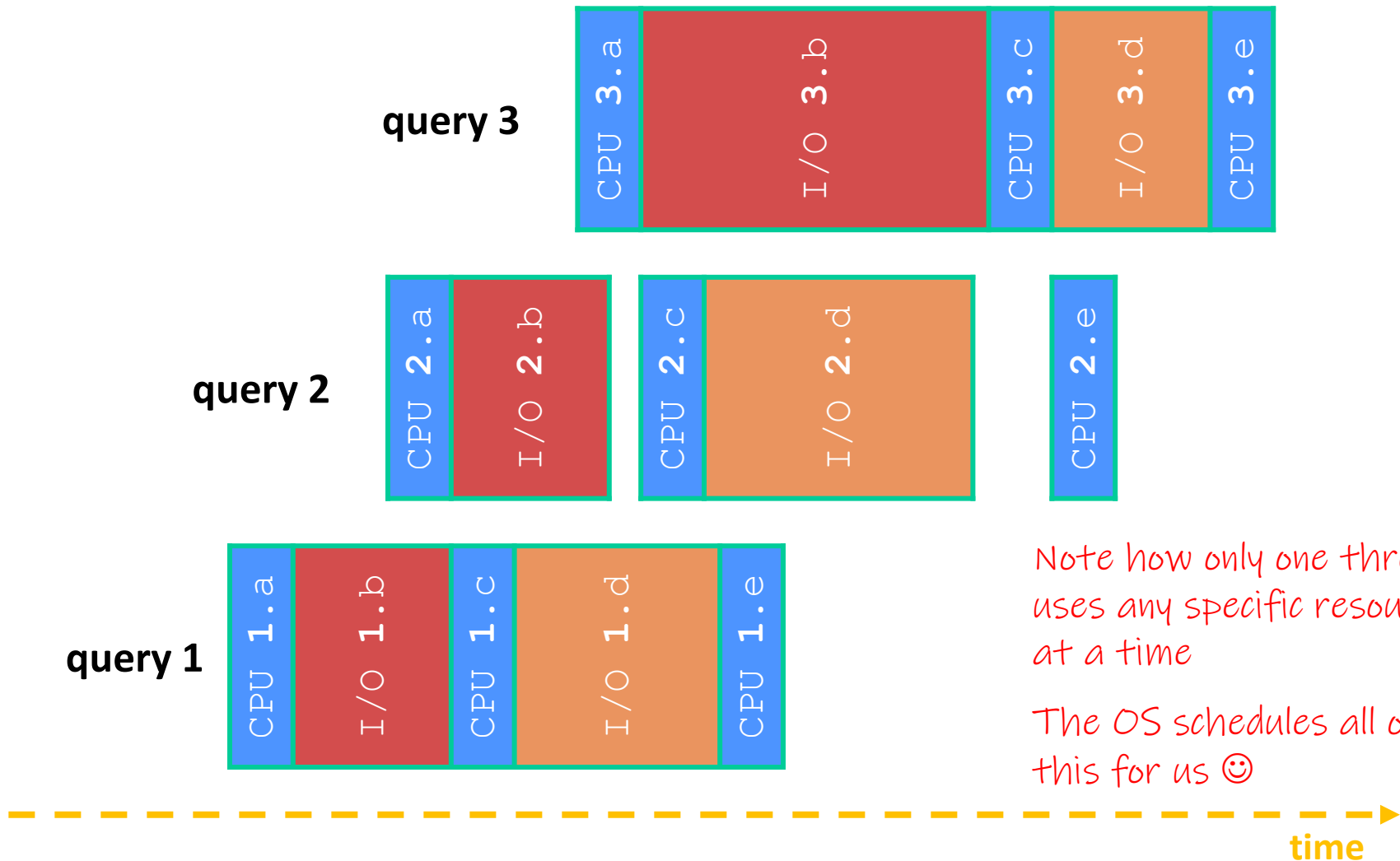
```
main() {  
  while (1) {  
    string query_words[] = GetNextQuery();  
    CreateThread(ProcessQuery(query_words));  
  }  
}
```

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

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

Multi-threaded Search Engine (Execution)

**Running with 1 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
- Threads can run in parallel if you have multiple CPUs/cores

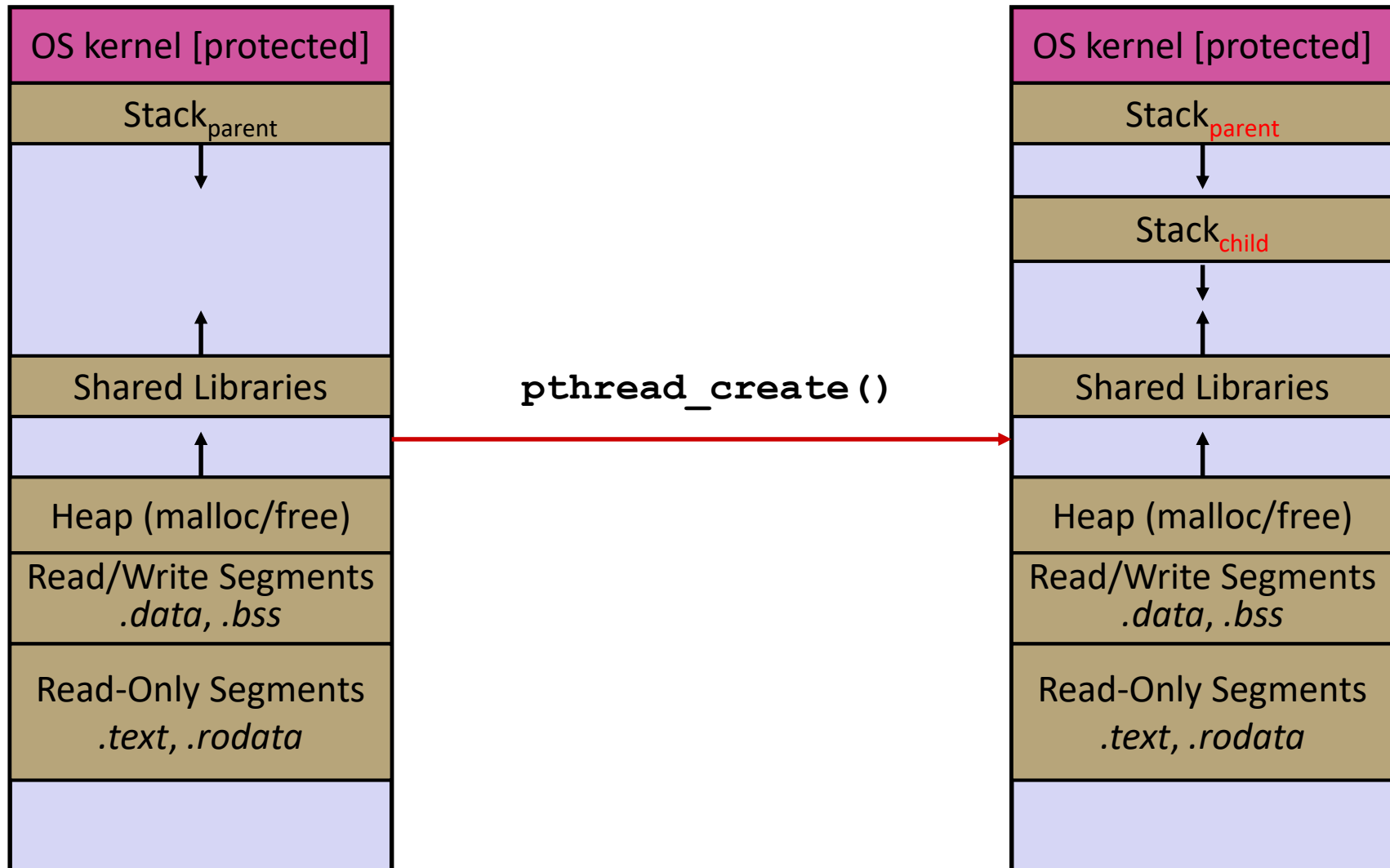
❖ Disadvantages:

- ❖ 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

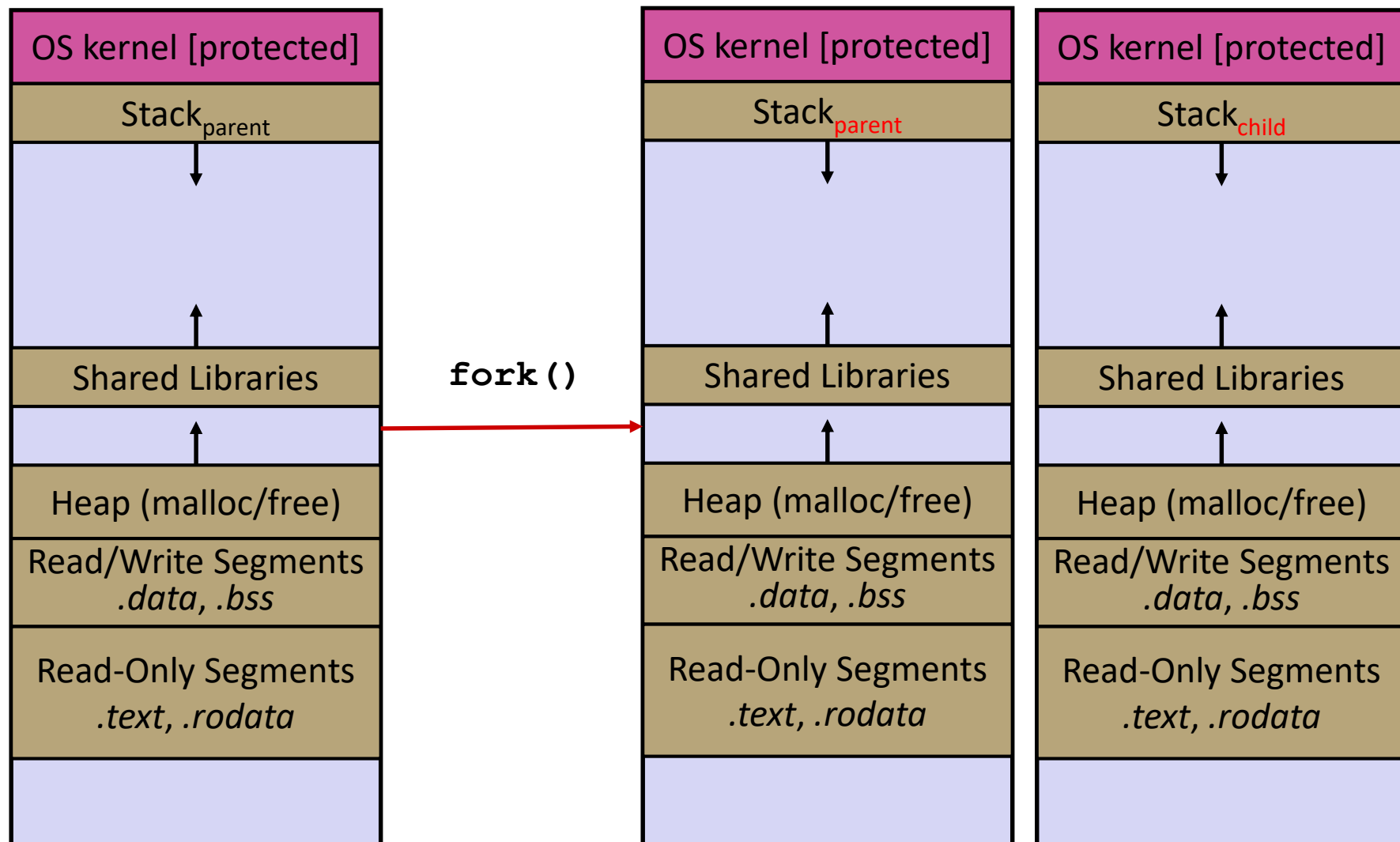
Threads vs. Processes

- ❖ In most modern OS's:
 - A Process has a unique: address space, OS resources, & security attributes
 - A Thread has a unique: stack, stack pointer, program counter, & registers
 - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes



Alternative: Processes

- ❖ What if we forked processes instead of threads?
- ❖ Advantages:
 - No shared memory between processes
 - No need for language support; OS provides “fork”
 - Processes are isolated. If one crashes, other processes keep going
- ❖ Disadvantages:
 - More overhead than threads during creation and context switching
 - Cannot easily share memory between processes – typically communicate through the file system

Alternate: Different I/O Handling

- ❖ Use **asynchronous** or **non-blocking** I/O
- ❖ Your program begins processing a query
 - When your program needs to read data to make further progress, it registers interest in the data with the OS and then switches to a different query
 - The OS handles the details of issuing the read on the disk, or waiting for data from the console (or other devices, like the network)
 - When data becomes available, the OS lets your program know
- ❖ Your program (almost never) blocks on I/O

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
 - 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
 - Program can choose to block while no file descriptors are ready

 **Poll Everywhere**pollev.com/cse333

- ❖ If I wanted to make a web browser, what concurrency model should I use within a single “tab”?
 - Note that a web browser may need to request many resources over the network and combine them together to load a page

- A. **Do it sequentially**
- B. **Use threads**
- C. **Use processes**
- D. **We're lost...**

 **Poll Everywhere**pollev.com/cse333

- ❖ If I wanted to make a web browser, what concurrency model should I use within a single “tab”?
 - Note that a web browser may need to request many resources over the network and combine them together to load a page

- A. Do it sequentially
- B. Use threads
- C. Use processes
- D. We're lost...

Concurrency will make more efficient use of time

We will need to share the data we request across “workers”

We want to be fast

Outline (next two lectures)

- ❖ We'll look at different `searchserver` implementations
 - Sequential
 - Concurrent via dispatching threads – `pthread_create()`
 - Concurrent via forking processes – `fork()`

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