

Introduction to Concurrency

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Justin Tysdal

Sayuj Shahi

Nicholas Batchelder

Leanna Mi Nguyen

Administrivia

- ❖ Ex16 due this morning
- ❖ Ex17 due Monday - last exercise! 🎉
- ❖ Sections tomorrow: `pthread` tutorial
 - Ex17 is on pthreads, so go to section!
 - Exercise will be posted at 3:00pm tomorrow, after sections are done
- ❖ HW4 due a week from today
 - How's it going? Networking code??

Administrivia

- ❖ Extra final points for coming to office hours next week
 - +5 points on the final (out of 100), but can't go above 100 total
 - Must go to an existing, in-person office hours and bring a problem set to work on; either from the extra-problems in the slides, or an old midterm question
 - Make sure the TA writes down your name

Outline

- ❖ Understanding Concurrency
 - Why is it useful
 - Why is it hard

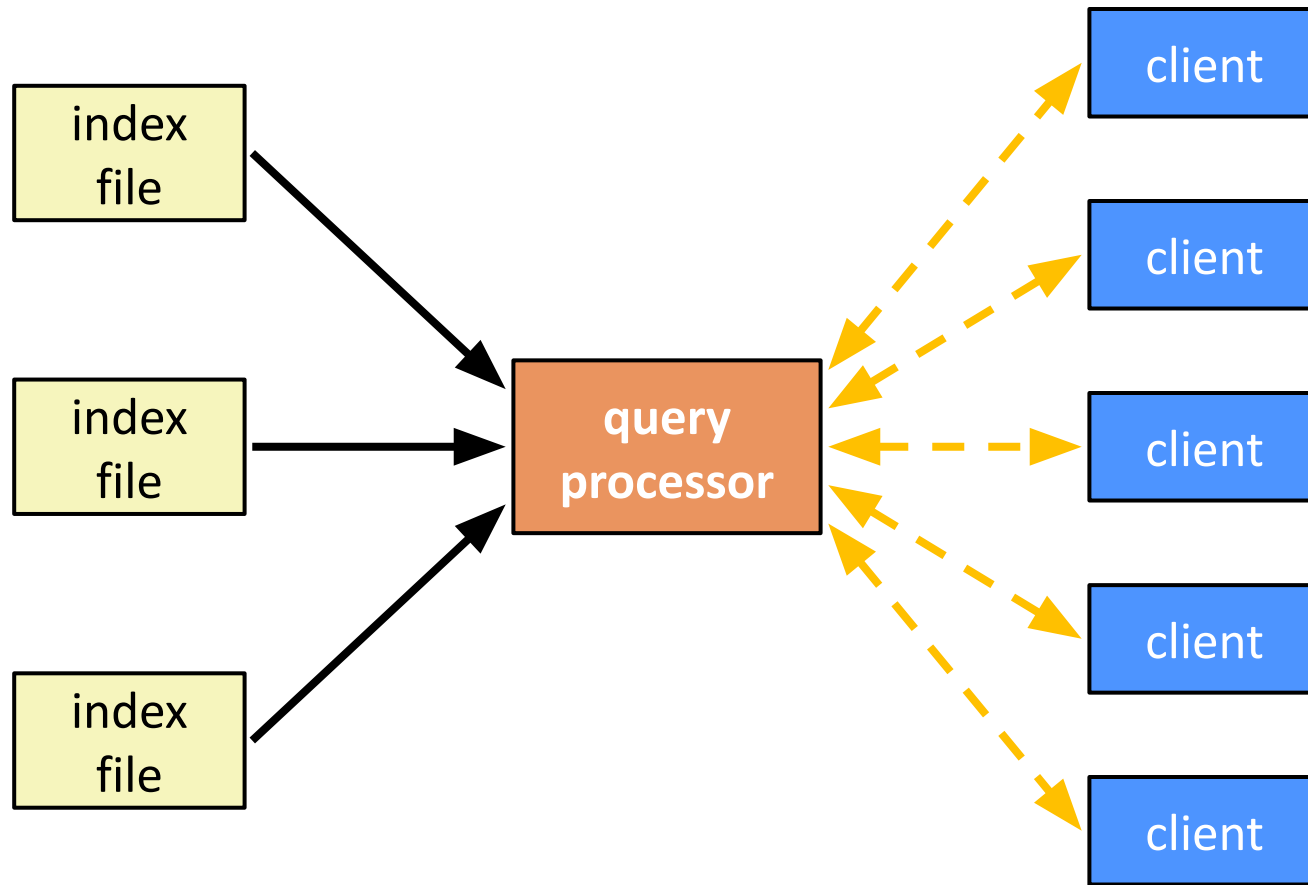
- ❖ Concurrent Programming Styles
 - Threads vs. processes
 - Asynchronous or non-blocking I/O
 - “Event-driven programming”

Building a Web Search Engine

- ❖ We've already built:
 - A web index
 - 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

- ❖ We need:
 - Something that turns HTTP requests into well-formed queries

Web Search Architecture



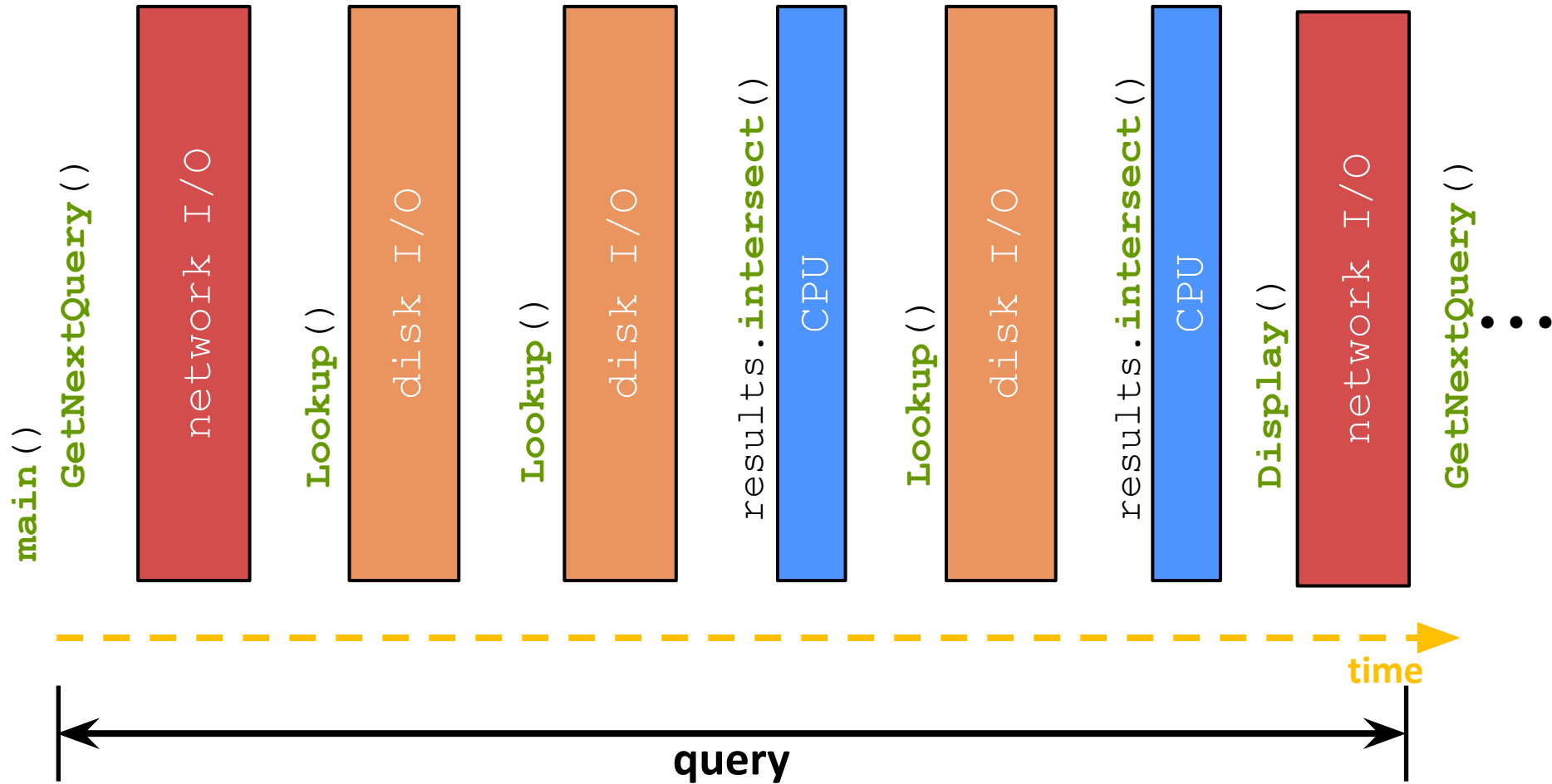
Sequential Implementation

- ❖ Pseudocode for sequential query processor:

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

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



Execution Timeline: a Multi-Word Query



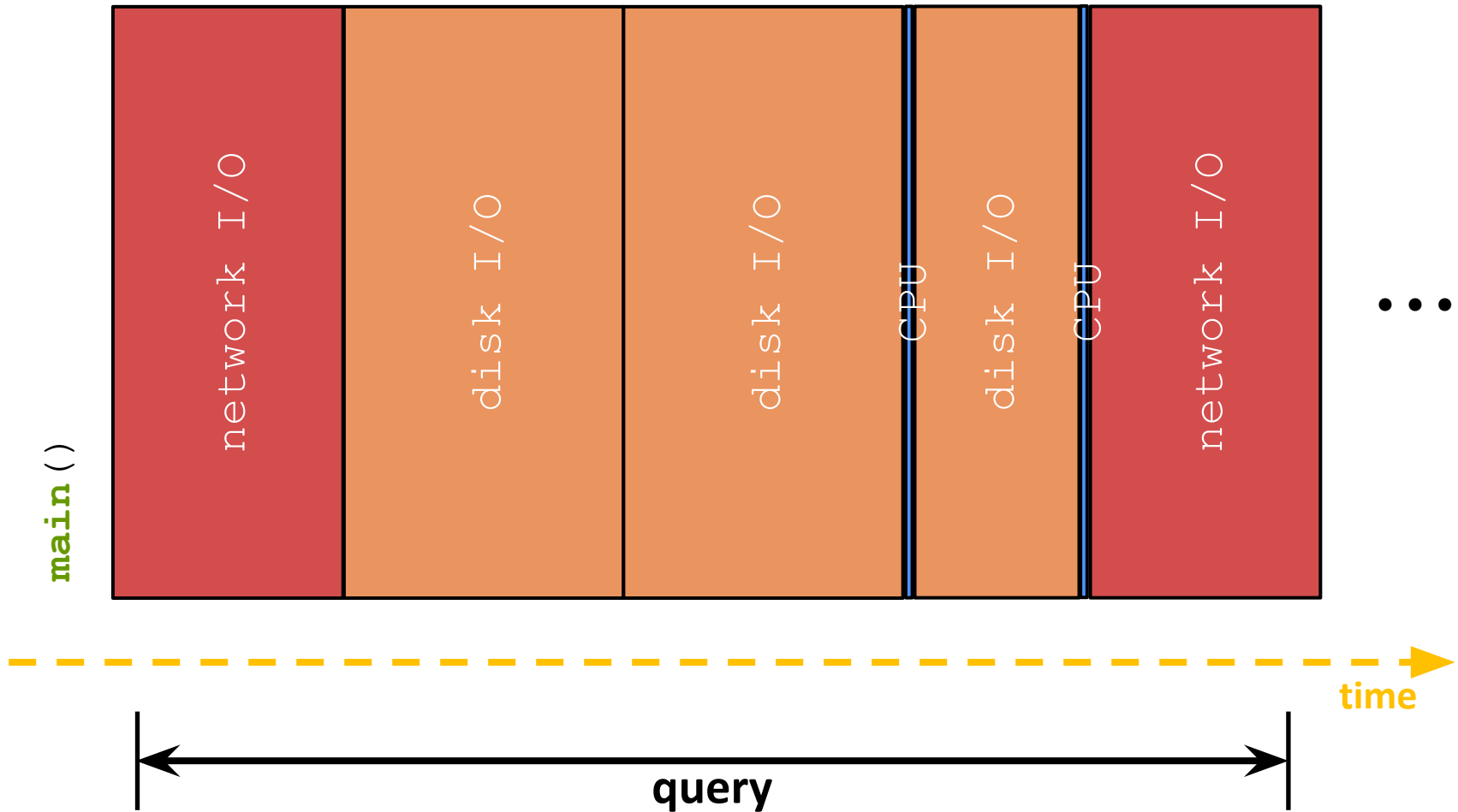
What About I/O-caused Latency?

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

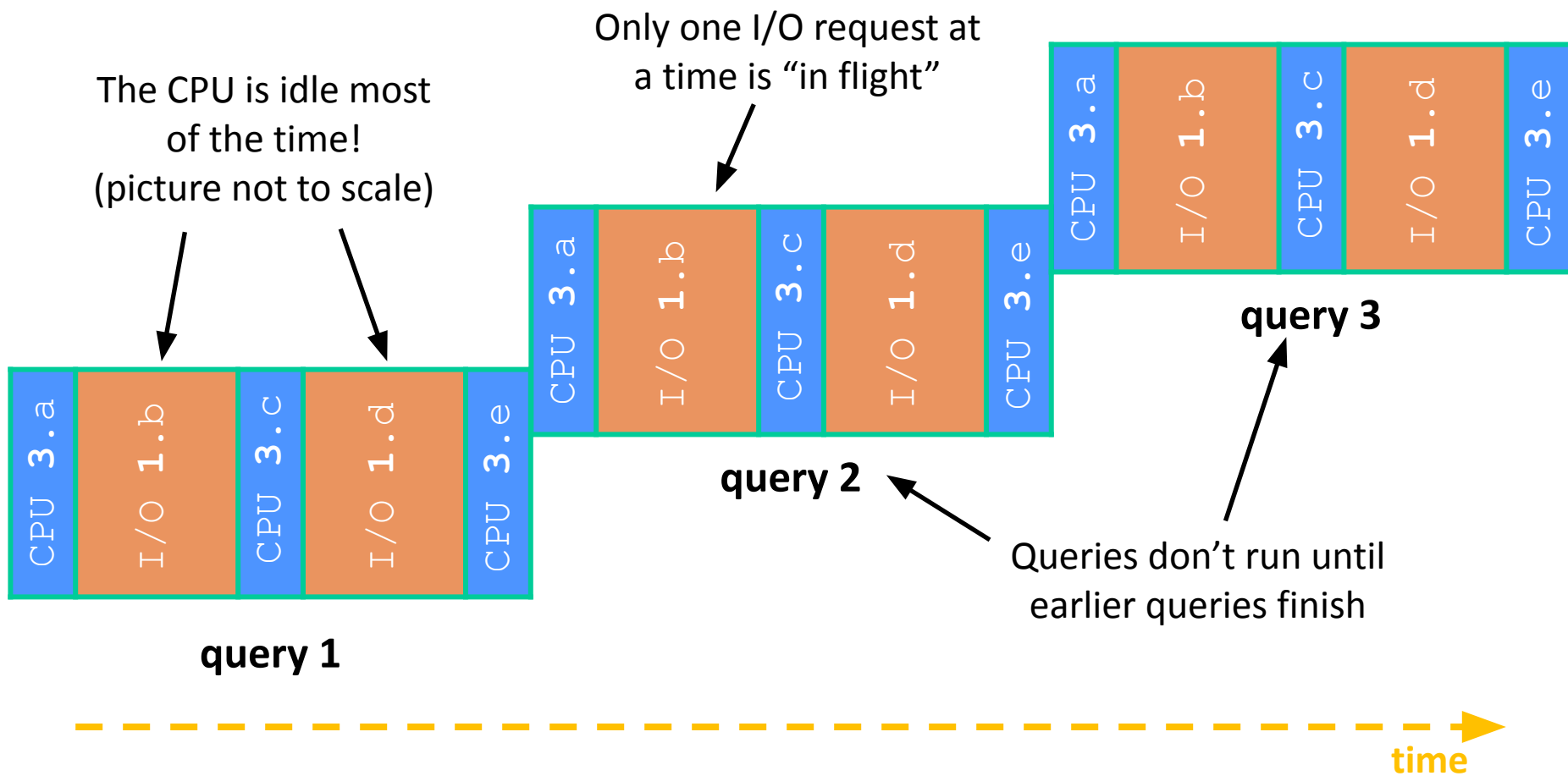
Numbers Everyone Should Know	
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within 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



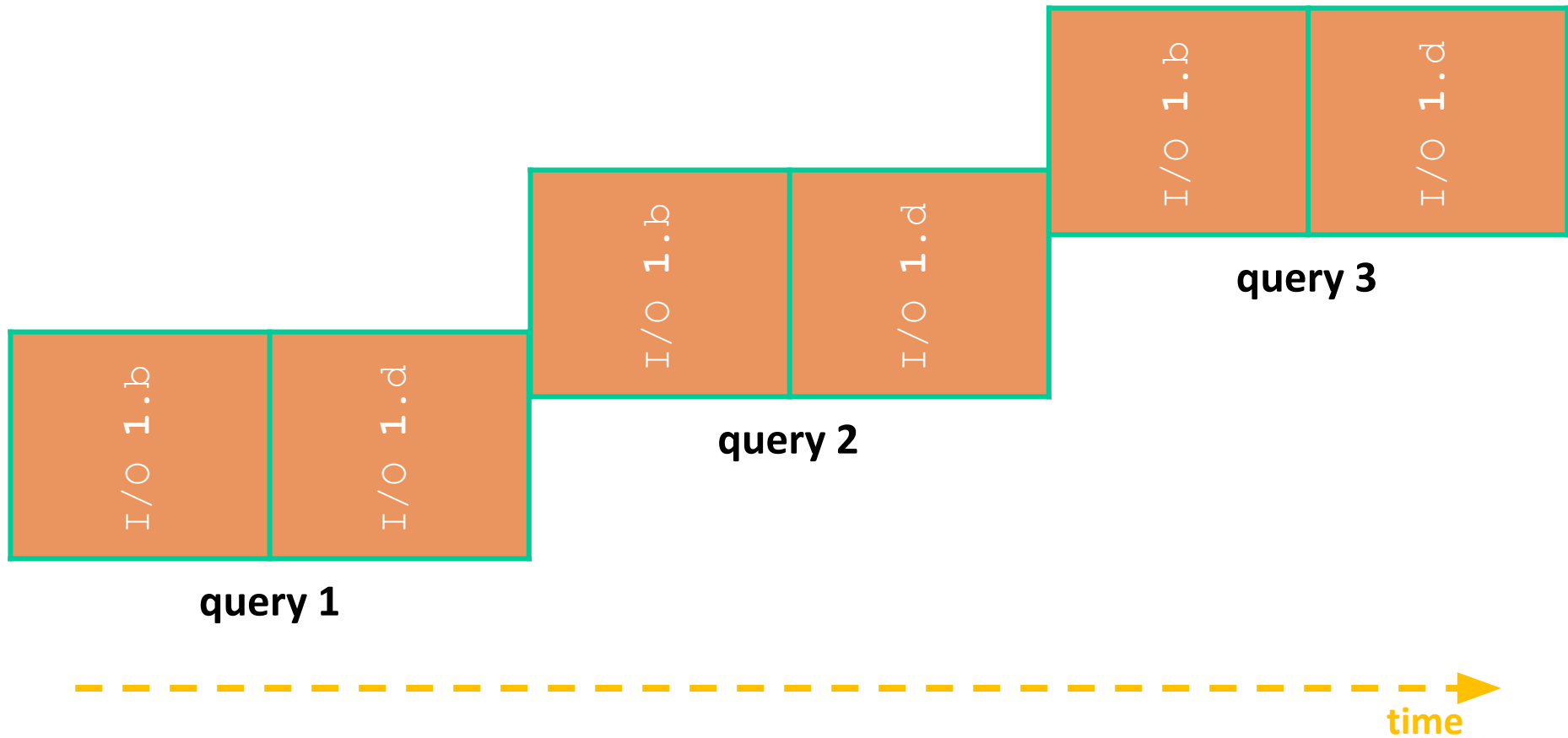
Execution Timeline: (Closer) To Scale



Sequential Queries – Simplified



Sequential Queries: To Scale



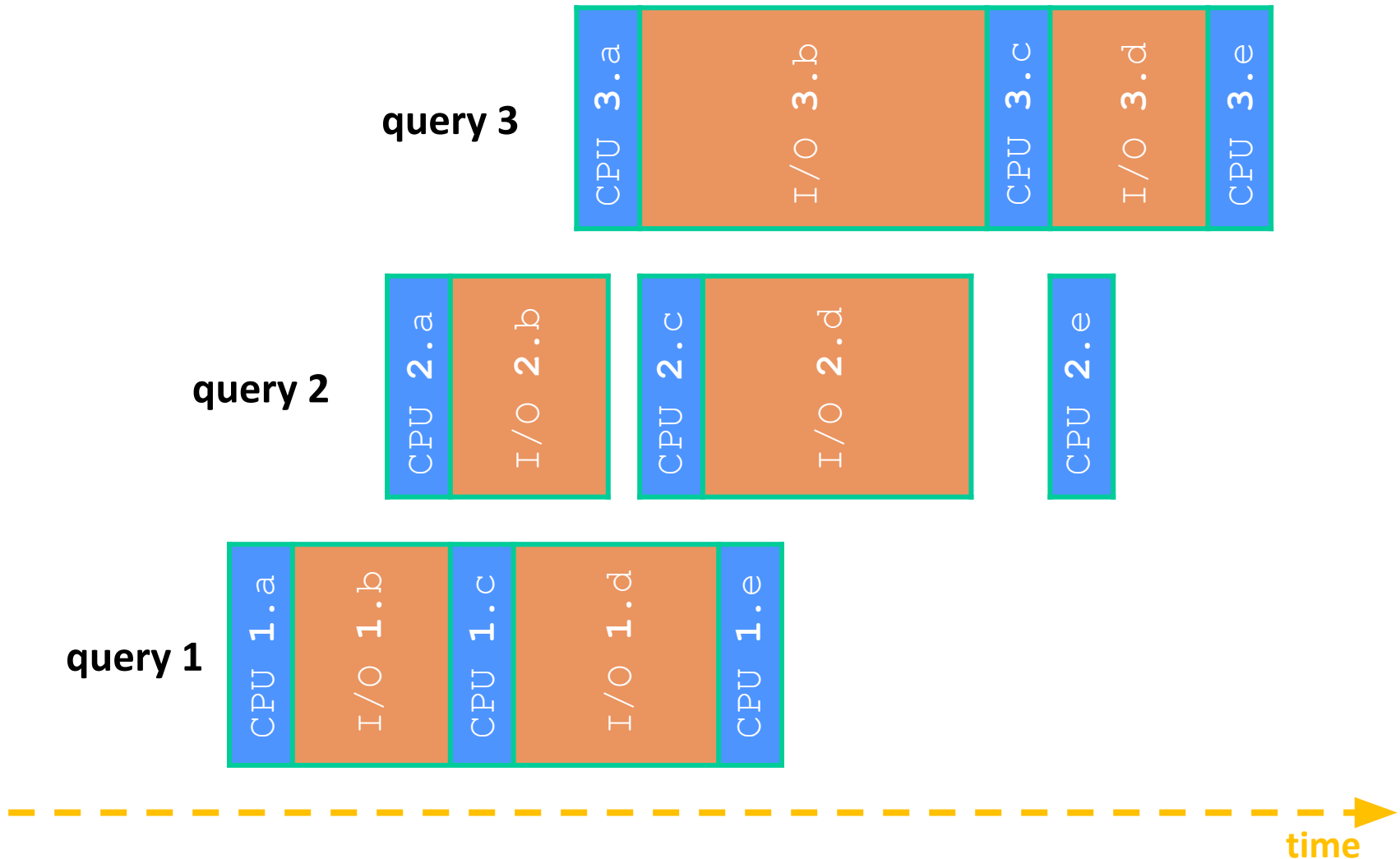
Sequential Can Be Inefficient

- ❖ Only one query is being processed at a time
 - All other queries queue up behind the first one
- ❖ The CPU is idle most of the time
 - It is *blocked* waiting for I/O to complete
 - Disk I/O can be very, very slow
- ❖ I/O operations from different queries shouldn't need to wait for each other
 - Often will be accessing different storage devices
 - Network card is idle most of the time during communication

“Concurrent Program”

- ❖ A version of the program that executes multiple tasks simultaneously
 - 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
 - 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
- ❖ Concurrency != parallelism
 - Parallelism is when multiple CPUs work simultaneously on 1 job

Concurrent Queries – Simplified

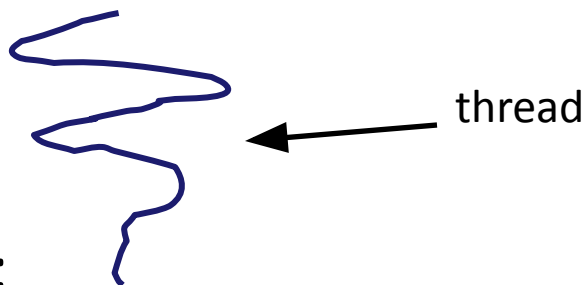


A Concurrent Webserver Implementation

- ❖ Use multiple threads or processes
 - As a query arrives, fork a new thread (or process) to handle it
 - The thread reads the query from the console, issues read requests against files, assembles results and writes to the console
 - The thread uses blocking I/O; the thread alternates between consuming CPU cycles and blocking on I/O
 - The OS context switches between threads/processes
 - While one is blocked on I/O, another can use the CPU
 - Multiple threads' I/O requests can be issued at once
 - When the machine has multiple cores, they can be working at the same time to serve different requests.

Introducing Threads

- ❖ Separate the concept of a **process** from an individual “*thread of control*”
 - Usually called a **thread** (or a *lightweight process*), this is a sequential execution stream within a process

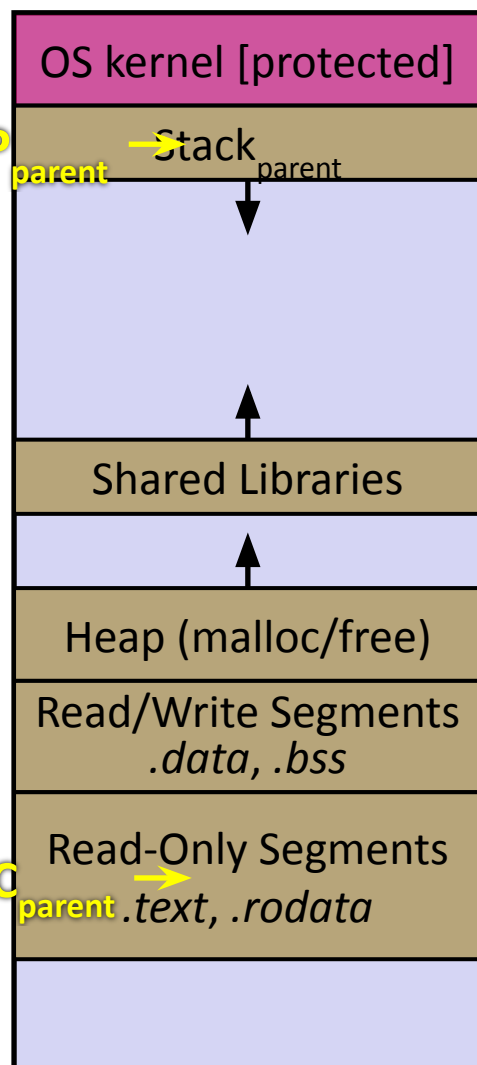


- ❖ In most modern OS's:
 - Process: address space, OS resources/process attributes
 - Thread: stack, stack pointer, program counter, registers
 - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads

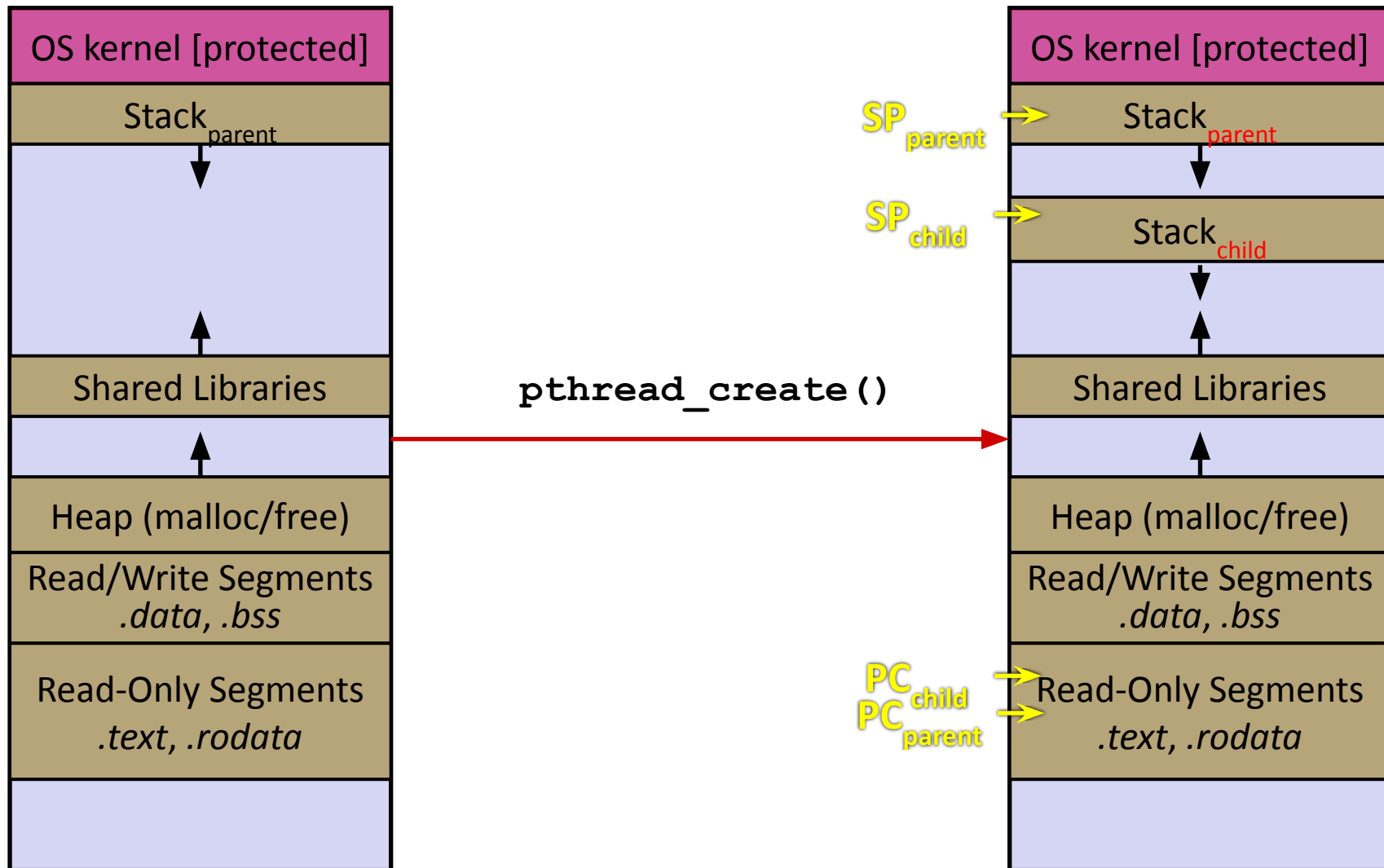
- ❖ Threads execute concurrently like processes
 - OS's often treat them, not processes, as the unit of scheduling
- ❖ Each thread has its own stack and saved registers
- ❖ Unlike processes, threads share memory
 - All threads access the memory of the process
 - Threads can communicate with each other through data
 - But they can also interfere with each other

Threads vs. Processes

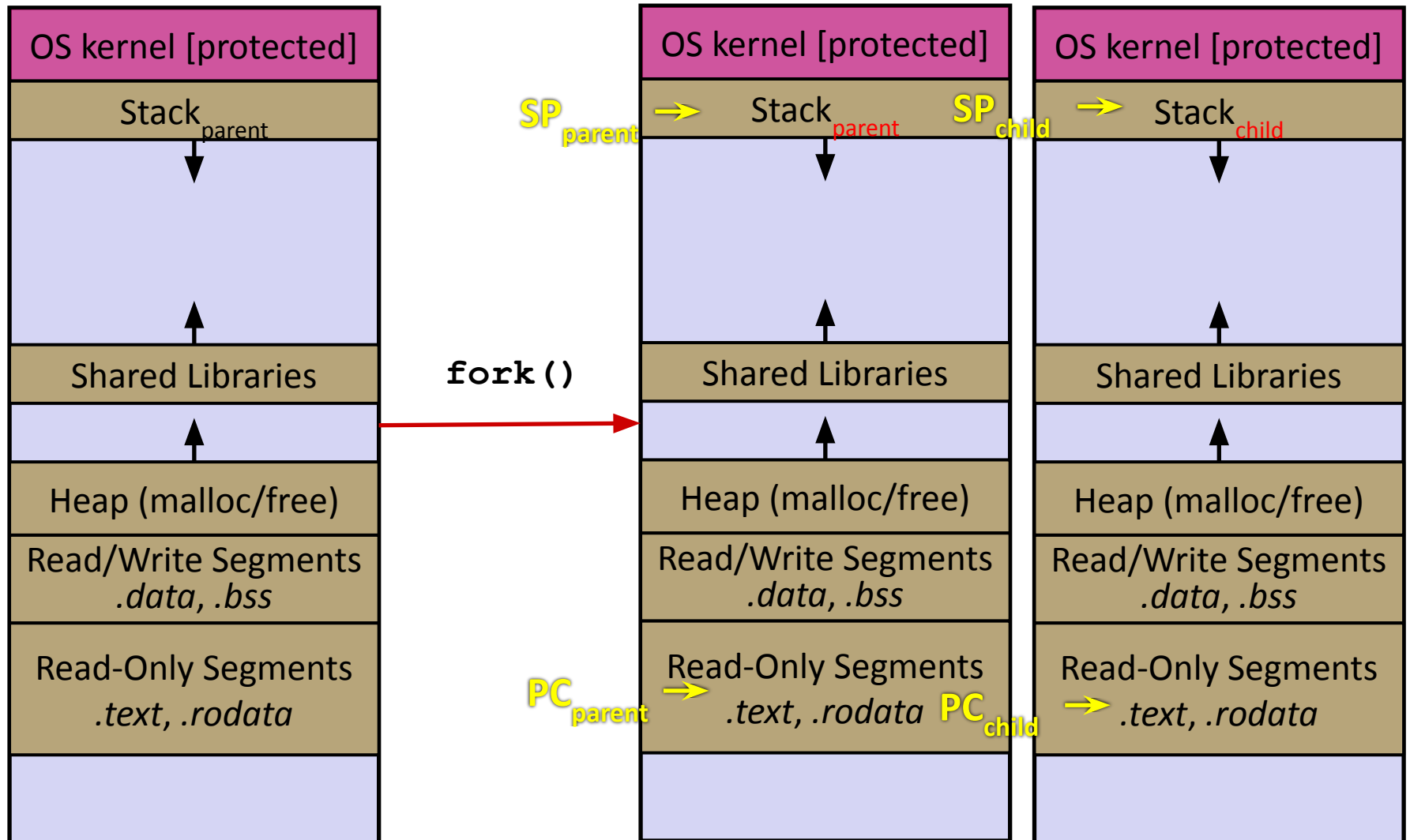


- ❖ Before any concurrency
 - One thread of execution running in one address space
 - One PC, stack, SP

Threads vs. Processes



Threads vs. Processes



Multithreaded Pseudocode

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

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

Why Threads?

❖ Advantages:

- Threads can share data structures without serialization or OS intervention
- Switching threads is faster than switching processes

❖ Disadvantages:

- Need language support for threads
- If threads share data, you need **locks** or other **synchronization**
 - Very bug-prone and difficult to debug
 - Introduces performance overhead

Alternative: Processes

- ❖ What if we forked processes instead of threads?
- ❖ Advantages:
 - No locks or other synchronization needed
 - No need for language support; OS provides “fork”
- ❖ Disadvantages:
 - More overhead than threads during creation and context switching
 - Cannot easily share data between processes – typically communicate through the file system

Alternate: Asynchronous I/O

- ❖ Use **asynchronous** or **non-blocking** I/O
 - When your program needs to read data, 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, and it switches back to the query that needed the data
- ❖ Your program (almost never) blocks on I/O

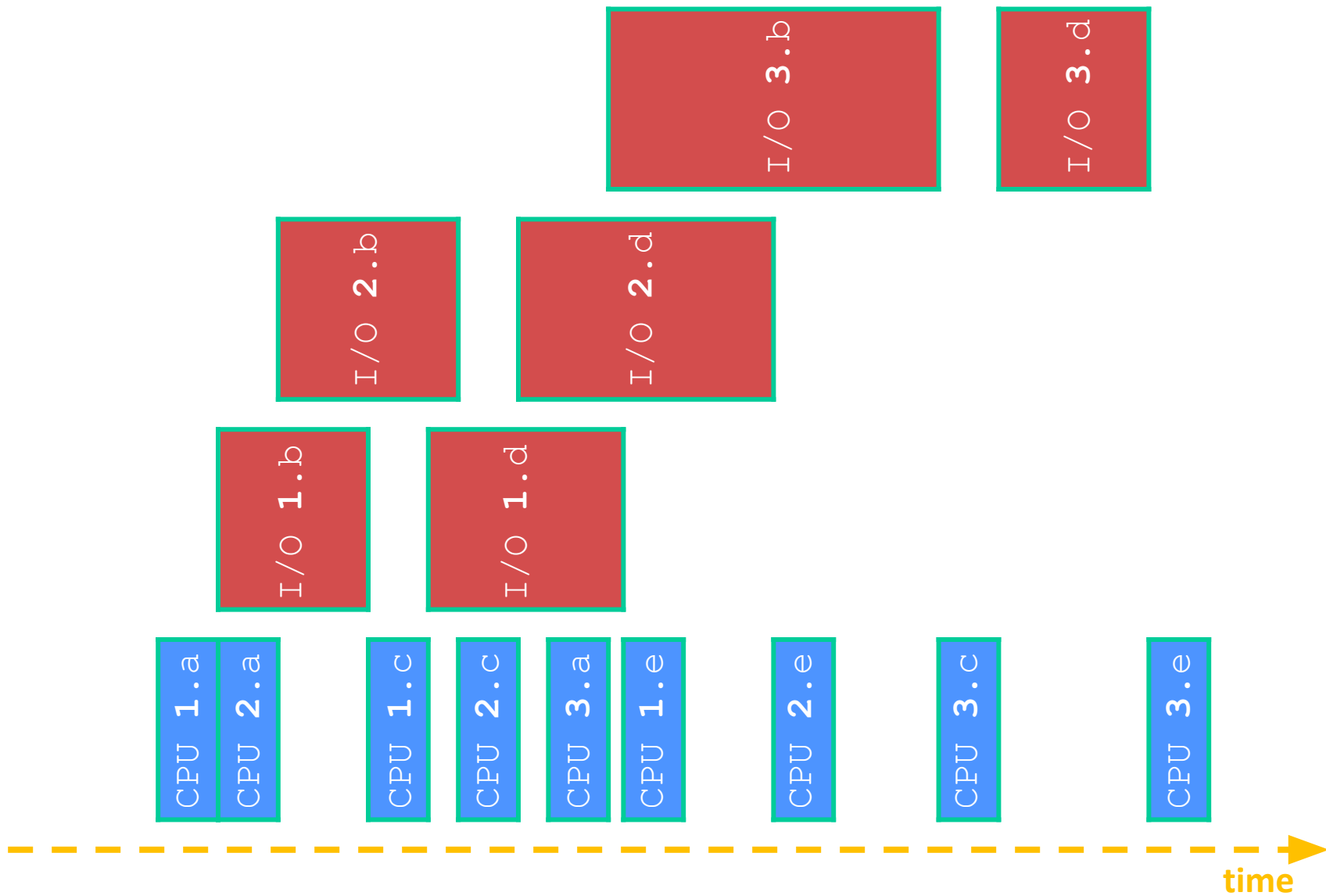
Event-Driven Programming

- ❖ Your program is structured as an *event-loop & state machine*

```
int main() {
    while (1) {
        event = OS.GetNextEvent();
        task = lookup(event);
        dispatch(task, event);
    }
}

void dispatch(task, event) {
    switch (task.state) {
        case READING_FROM_CONSOLE:
            query_words = event.data;
            async_read(index, query_words[0]);
            task.state = READING_FROM_INDEX;
            return;
        case READING_FROM_INDEX:
            ...
    }
}
```

Asynchronous, Event-Driven



Non-blocking vs. Asynchronous

- ❖ Mean the same thing in some contexts
- ❖ In other contexts:
 - Non-blocking I/O: start the operation, then periodically check with the OS to see if it's done
 - Asynchronous I/O: start the operation, the OS/runtime will send a function call or event when it's done

Non-blocking vs. Asynchronous

- ❖ 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
 - Essentially allows your program to “schedule” itself

Non-blocking vs. Asynchronous

- ❖ Asynchronous I/O (disk)
 - Program tells the OS to begin reading/writing
 - The “begin_read” or “begin_write” returns immediately
 - When the I/O completes, OS delivers an event to the program
- ❖ According to the Linux specification, the disk never blocks your program (just delays it)
 - Asynchronous I/O is primarily used to hide disk latency
 - Asynchronous I/O system calls are messy and complicated 😞

Why Events?

❖ Advantages:

- Interleaving is at predictable places
- Don't have to worry about locks and race conditions
- For some kinds of programs, especially GUIs, leads to a very simple and intuitive program structure
 - One event handler for each UI event

❖ Disadvantages:

- Can lead to very complex structure for programs that do lots of disk and 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

One Way to Think About It

- ❖ Threaded code:
 - Each thread executes its task sequentially, and per-task state is naturally stored in the thread's stack
 - OS and thread scheduler switch between threads for you
- ❖ Event-driven code:
 - *You* are the scheduler
 - You have to bundle up task state into continuations (data structures describing what-to-do-next); tasks do not have their own stacks

Don't Forget

- ❖ Ex17 due Monday - last exercise! 🎉
- ❖ Sections tomorrow: `pthread` tutorial
 - Ex17 is on pthreads, so go to section!
 - Exercise will be posted at 3:00pm tomorrow, after sections are done
- ❖ HW4 due a week from today