Introduction to Concurrency
CSE 333 Winter 2019

Instructor: Hal Perkins

Teaching Assistants:

Alexey Beall	Renshu Gu	Harshita Neti
David Porter	Forrest Timour	Soumya Vasisht
Yifan Xu	Sujie Zhou	

Administrivia

- ❖ Sections tomorrow: `pthread` tutorial/demo
  - ▪ pthread exercise posted after sections, due Monday morning
  - ▪ Much more about concurrency in this and next several lectures
    - • But will not repeat section material

- ❖ hw4 due next Thursday night
  - ▪ Yes, can still use up to 2 late days on hw4 (*if* you haven't used them up already – check!)

- ❖ Gates Center Linux workstation networking ok now once workstation rebooted (only needs to be done once)

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 need:
  - 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

Web Search Architecture

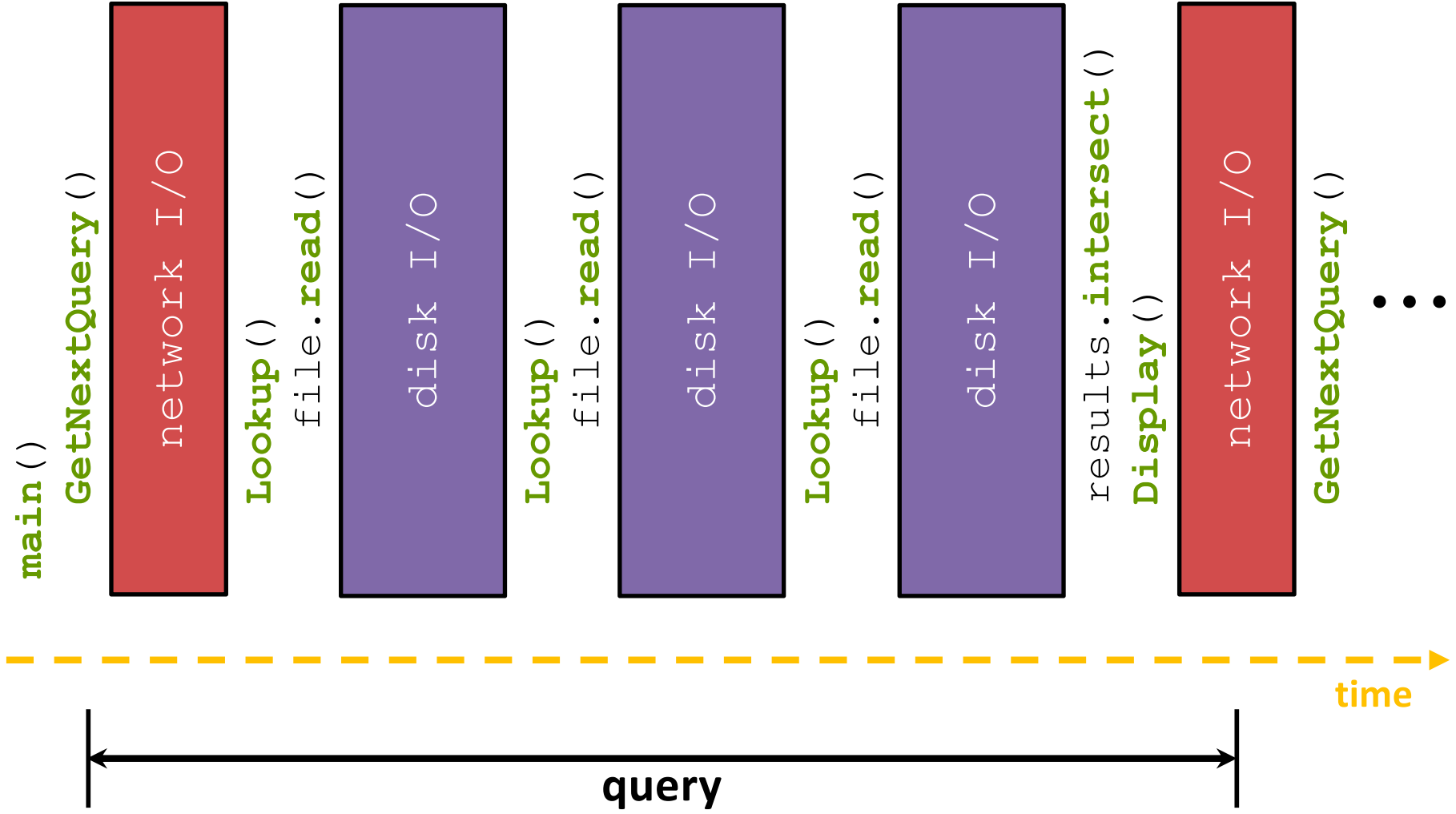| index file		query processor		client
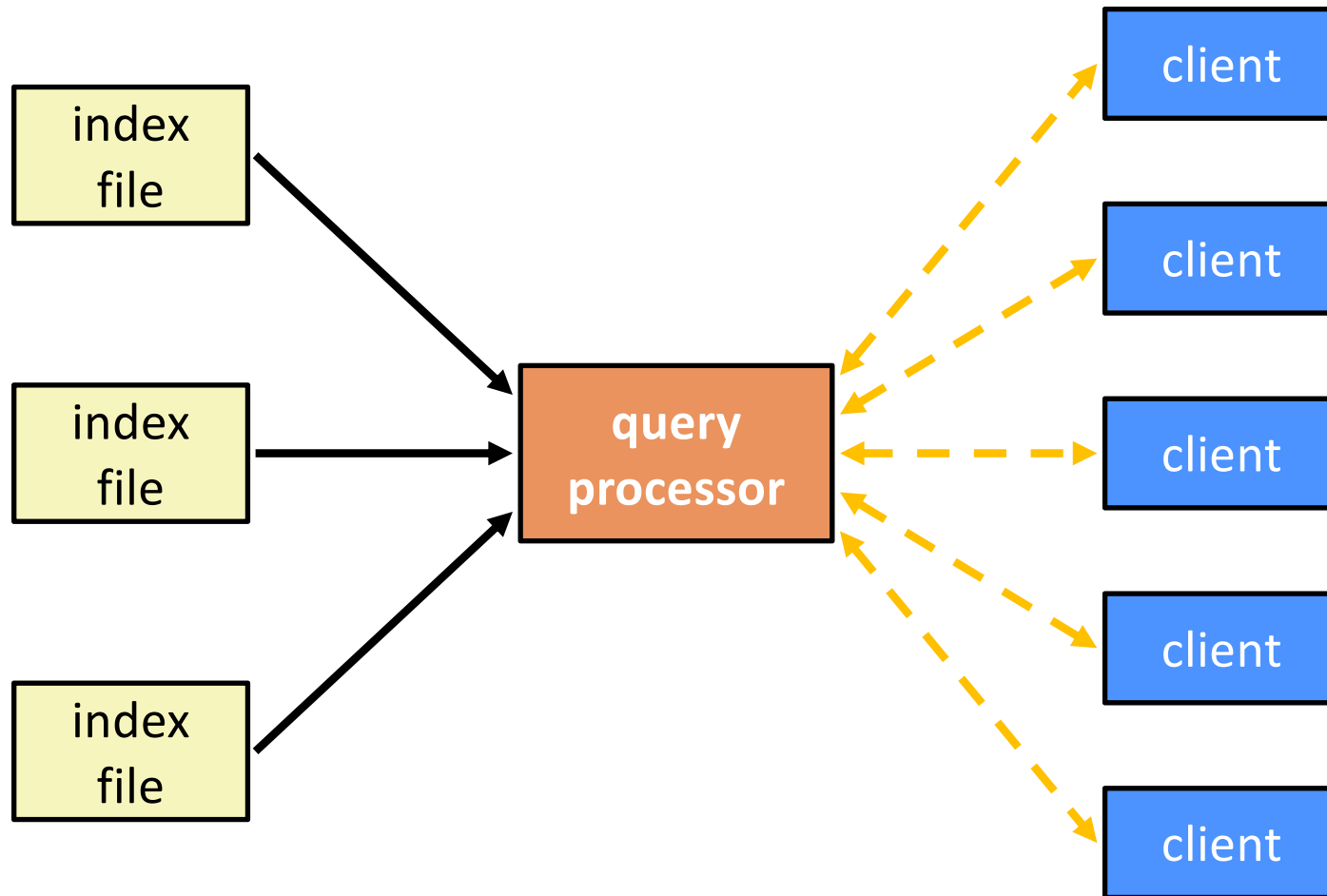
Sequential Implementation

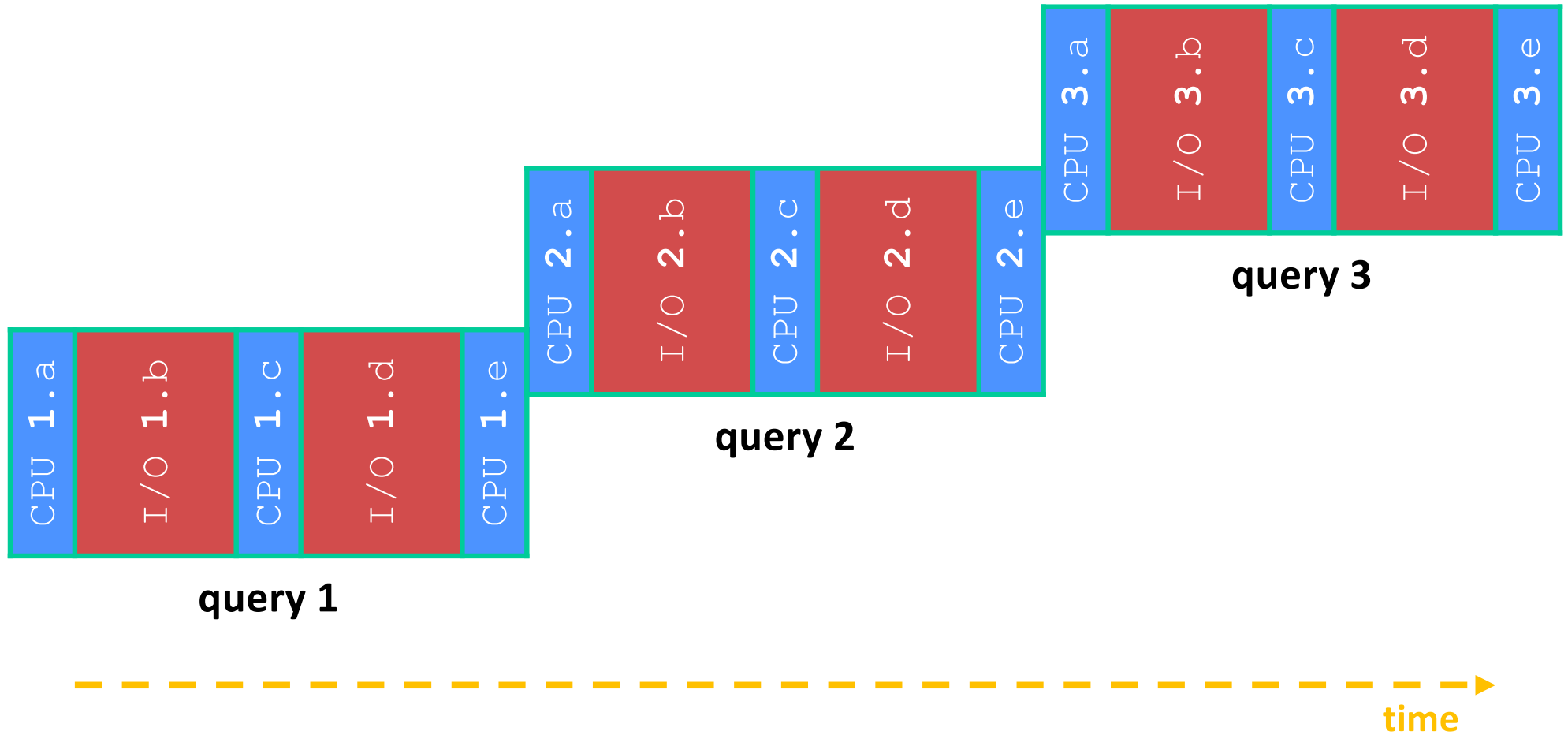❖ Pseudocode for sequential query processor:

```
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);
  }
}
```
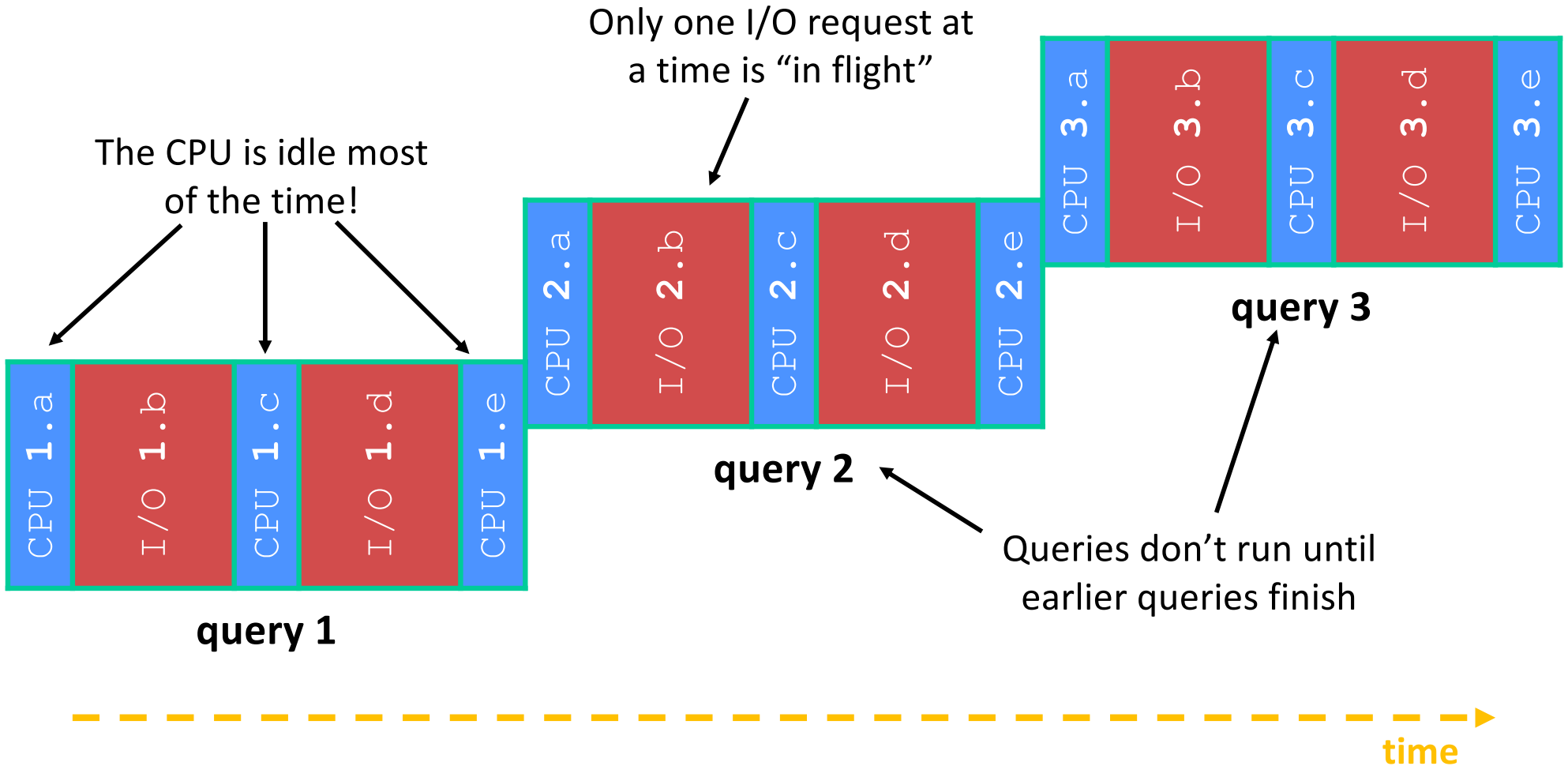
Sequential Execution Timeline

Sequential Queries – Simplified

query 1

query 2

query 3

time

Sequential Queries – Simplified

Only one I/O request at
a time is "in flight"

The CPU is idle most
of the time!

| CPU 3.a | I/O 3.b | CPU 3.c | I/O 3.d | CPU 3.e |

query 3

| CPU 2.a | I/O 2.b | CPU 2.c | I/O 2.d | CPU 2.e |

query 2

| CPU 1.a | I/O 1.b | CPU 1.c | I/O 1.d | CPU 1.e |

query 1

Queries don't run until
earlier queries finish

time

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

- ❖ 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.

Concurrency

❖ 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 executing multiple CPU instructions simultaneously

A Concurrent 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

Introducing Threads

* ❖ Separate the concept of a process from an individual "*thread of control*"

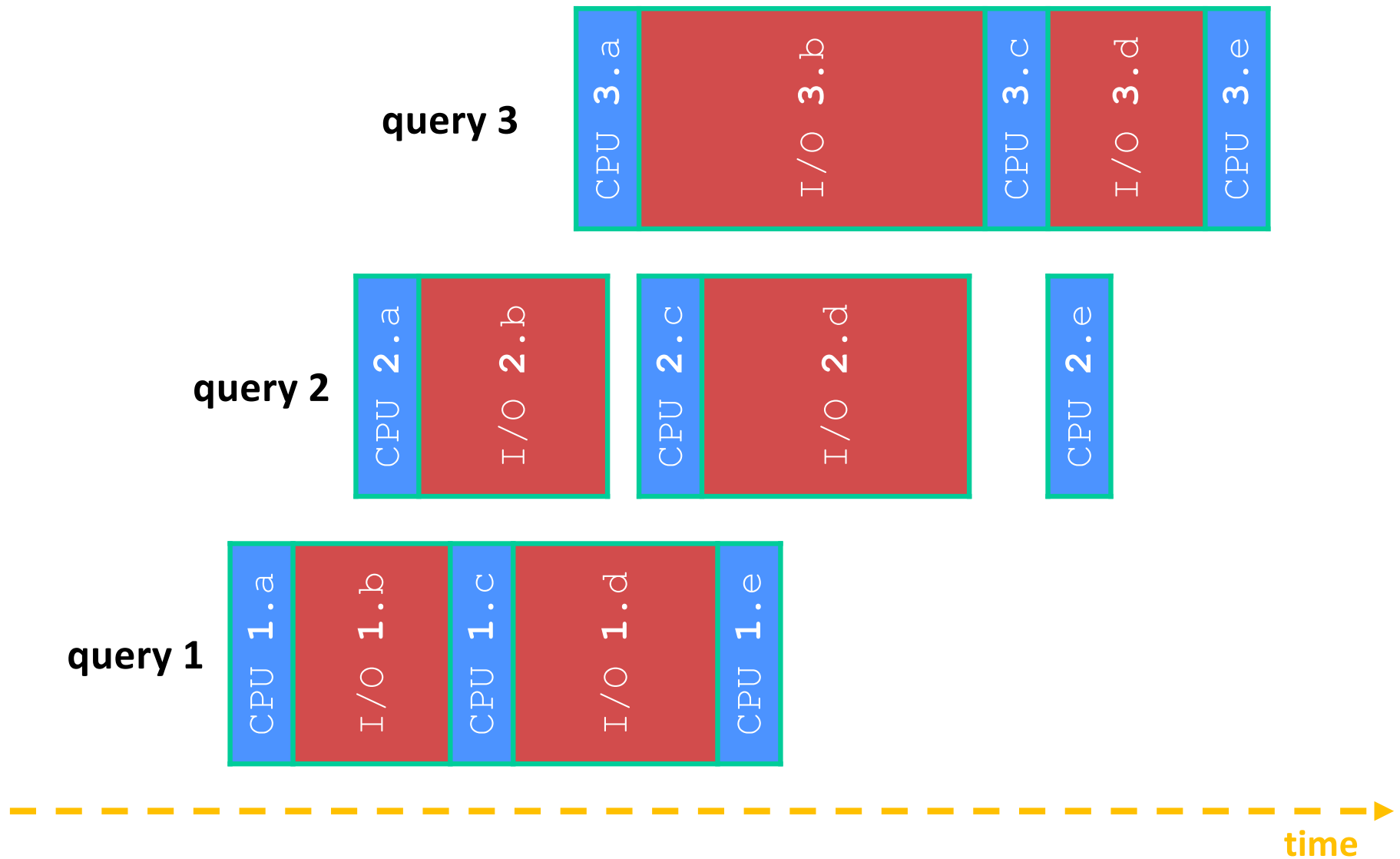   * ▪ Usually called a thread (or a *lightweight process*), this is a sequential execution stream within a process
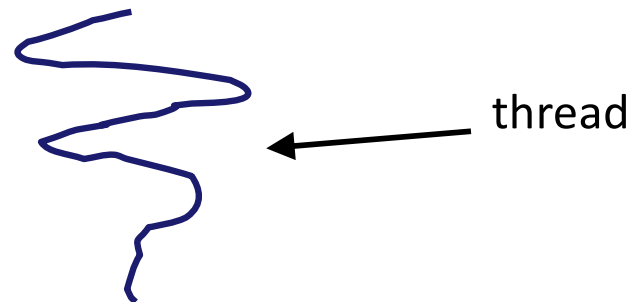
thread

* ❖ 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

Multithreaded Pseudocode

```
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);
}
```

Multithreaded Queries – Simplified

query 3

| CPU 3.a | I/O 3.b | CPU 3.c | I/O 3.d | CPU 3.e |

query 2

| CPU 2.a | I/O 2.b | | CPU 2.c | I/O 2.d | | CPU 2.e |

query 1

| CPU 1.a | I/O 1.b | CPU 1.c | I/O 1.d | CPU 1.e |

time →

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

Alternative: Processes

- ❖ What if we forked processes instead of threads?

- ❖ Advantages:
  - No shared memory between processes
  - No need for language support; OS provides "fork"

- ❖ Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share memory between processes – typically communicate through the file system

Alternate: Asynchronous I/O

❖ 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

Event-Driven Programming

❖ Your program is structured as an *event-loop*

```
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:
      ...
  }
}

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

Asynchronous, Event-Driven
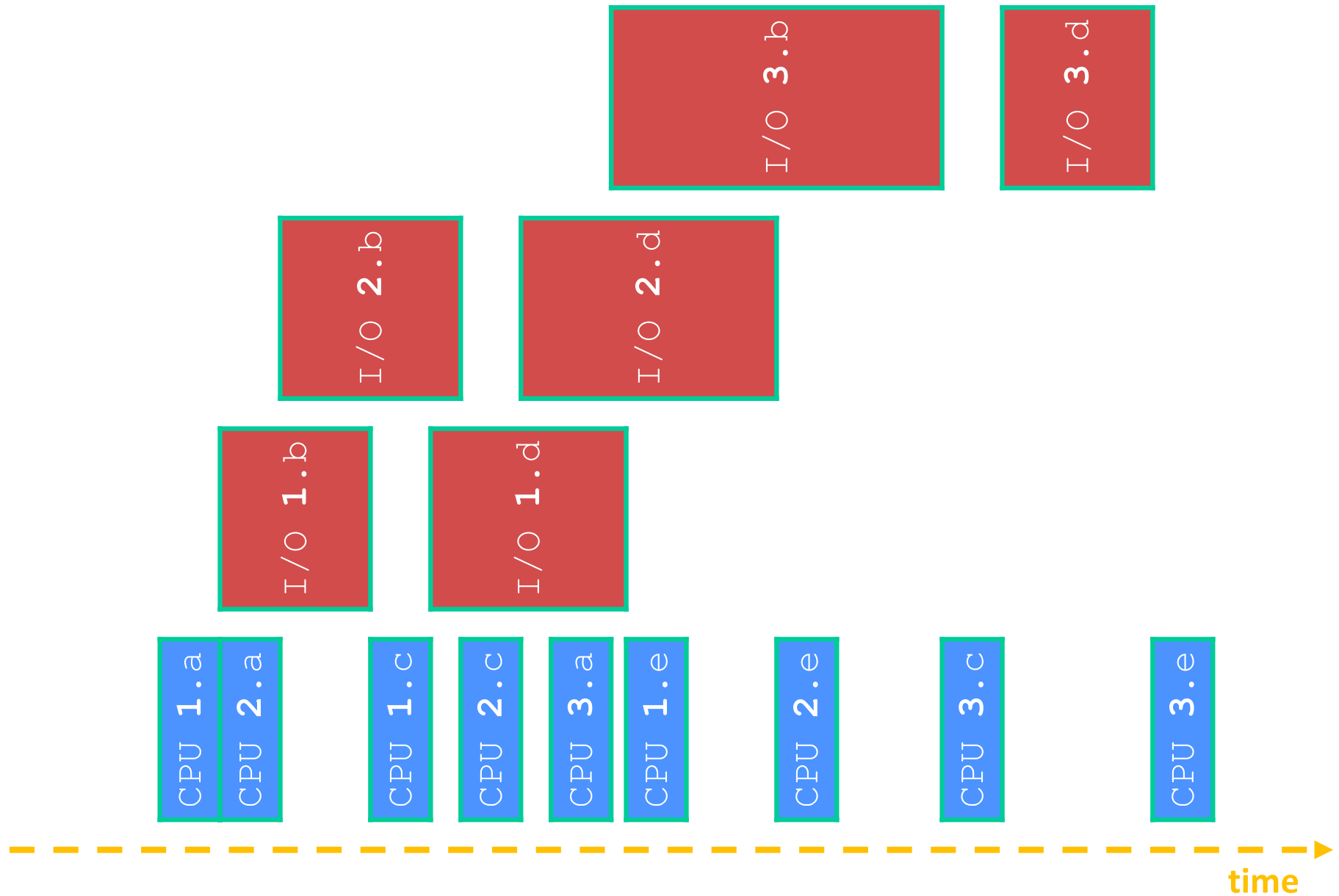
Non-blocking vs. Asynchronous

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

Non-blocking vs. Asynchronous

❖ Asynchronous I/O (disk)

▪ Program tells the OS to being 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:
  - ▪ 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