Introduction to Concurrency

CSE 333 Spring 2019

Instructor: Justin Hsia

Teaching Assistants:

Aaron Johnston	Andrew Hu	Daniel Snitkovskiy
Forrest Timour	Kevin Bi	Kory Watson
Pat Kosakanchit	Renshu Gu	Tarkan Al-Kazily
Travis McGaha		

Administrivia

- hw4 due next Thursday (6/6)
  - Yes, can still use *one* late day on hw4
- Exercise 17 (last one!) released Monday, due Wednesday
  - Concurrency via `pthread`s
- Final is Wednesday (6/12), 12:30-2:20 pm, ARC 147
  - Review Session: Sunday (6/9), 4-6:30 pm, ECE 125
  - Reference sheet was passed out in section yesterday, also available on course website
  - *Two* double-sided, handwritten sheets of notes allowed
  - Topic list and past finals on Exams page on website

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`

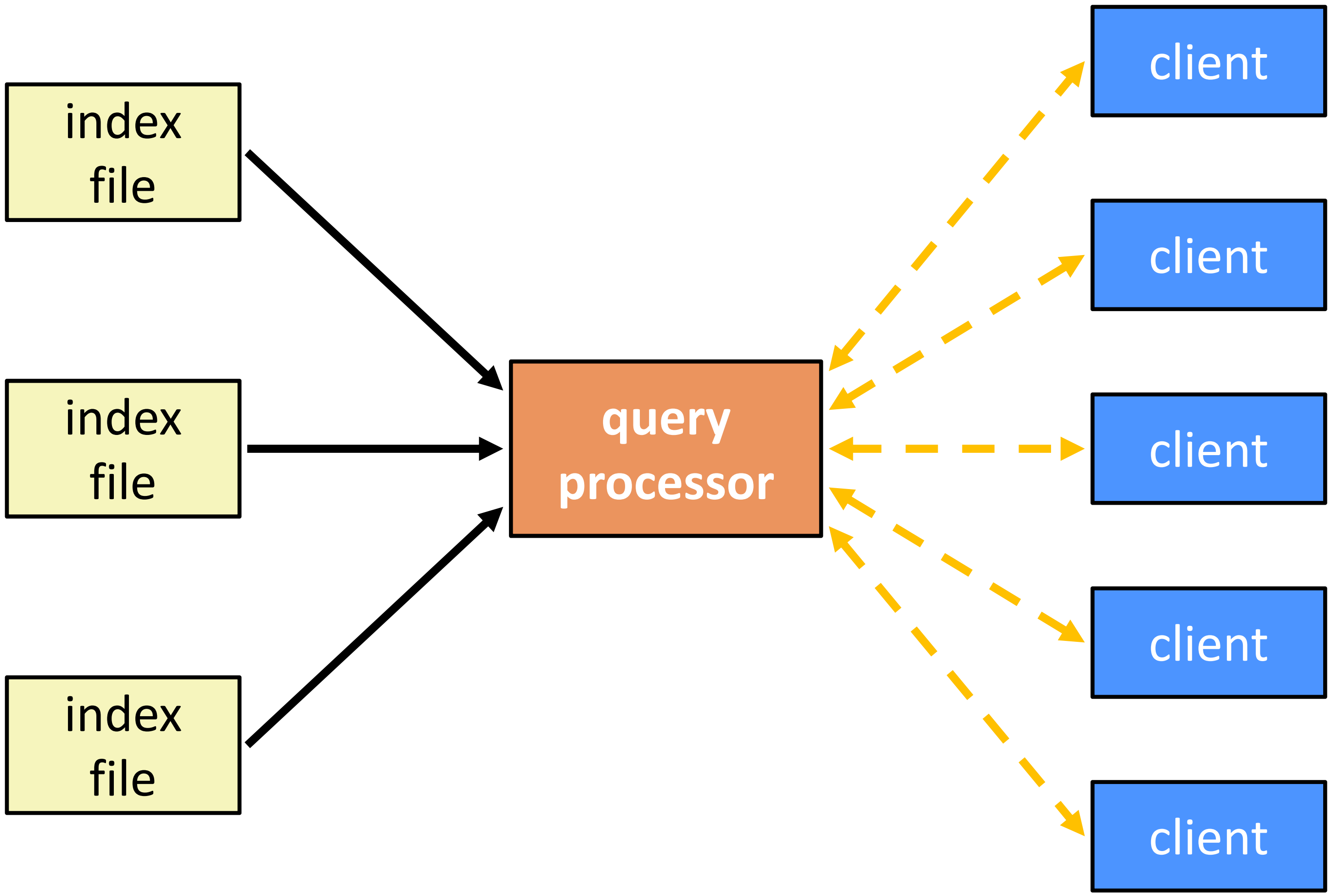
Outline

- Understanding Concurrency
  - Why is it useful
  - Why is it hard
- Concurrent Programming Styles
  - Threads vs. processes
  - Non-blocking I/O
- Search Server Revisited

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



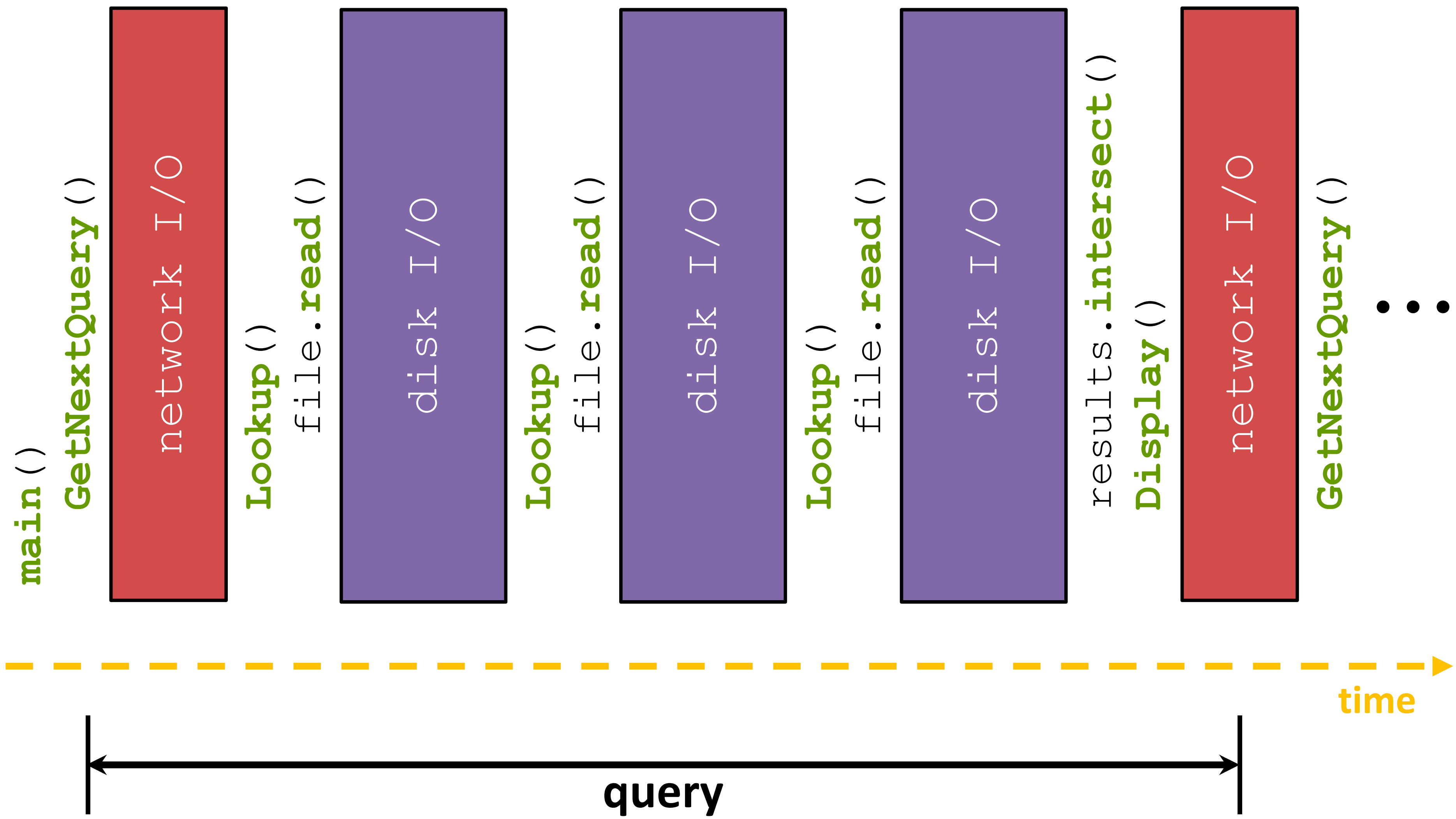
Sequential Implementation

- Pseudocode for sequential query processor:

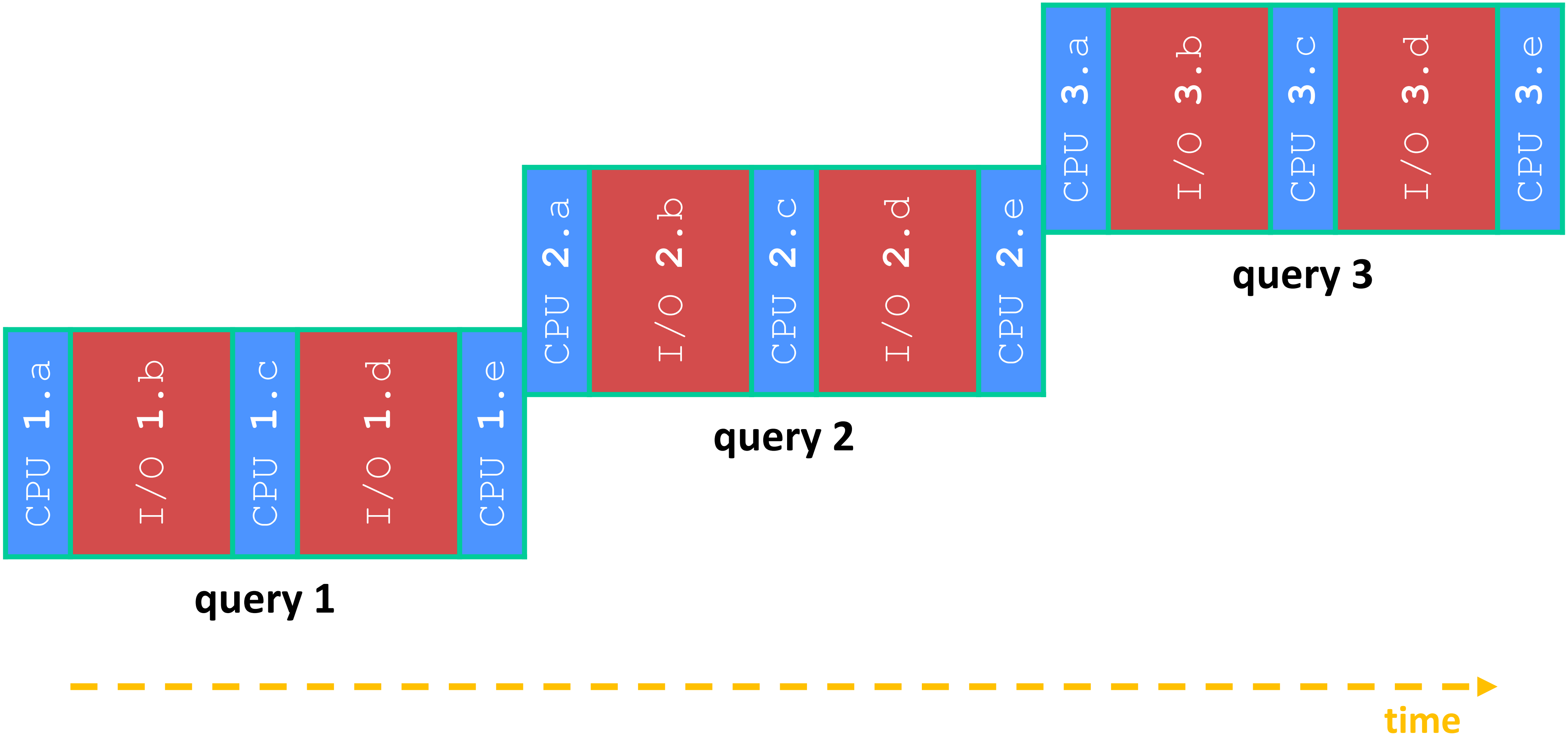
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist {
    doclist.append(file.read(hit));
  }
  return doclist;
}

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);
  }
}
```

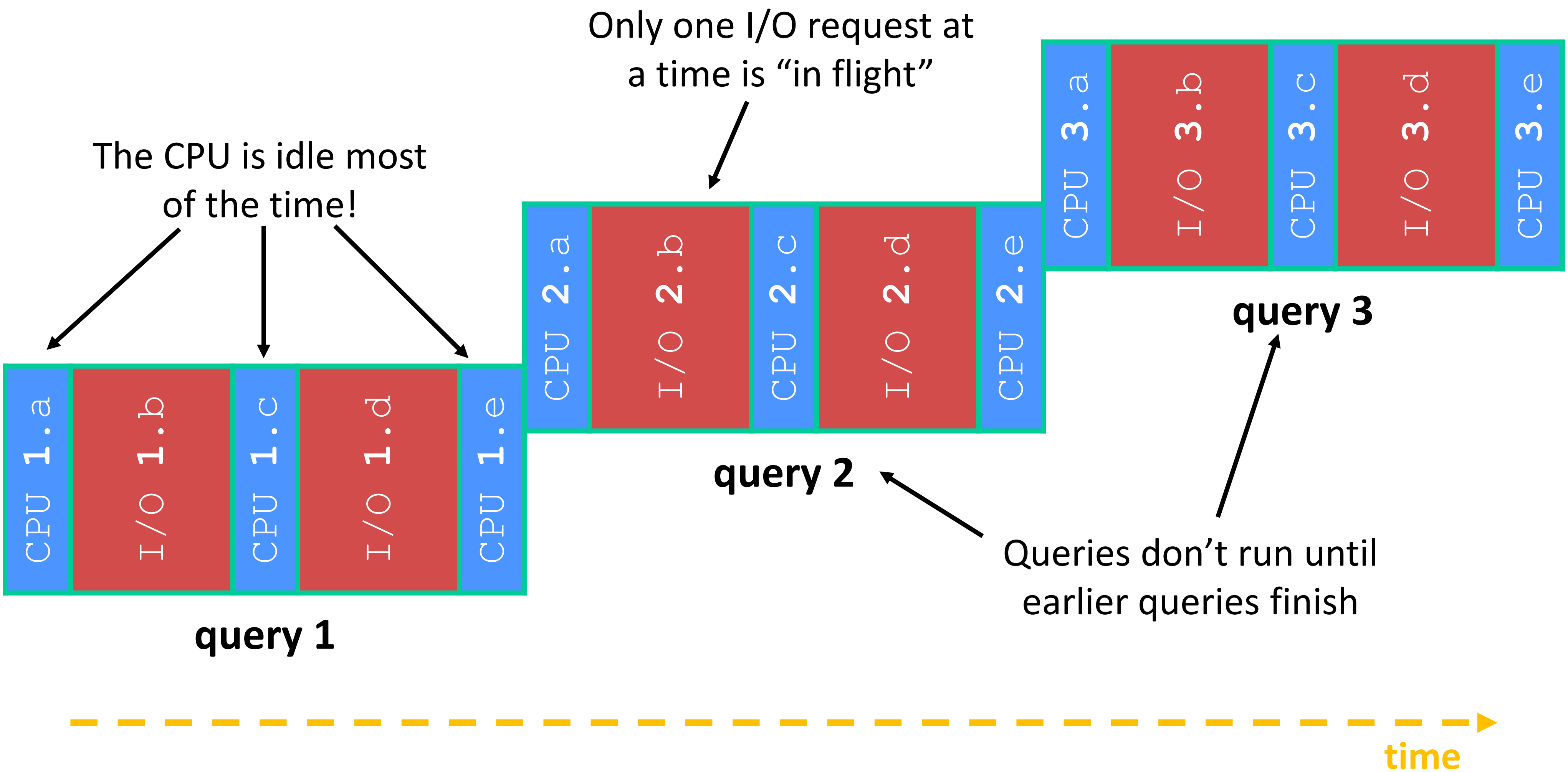
Sequential Execution Timeline



Sequential Queries – Simplified



Sequential Queries – Simplified



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

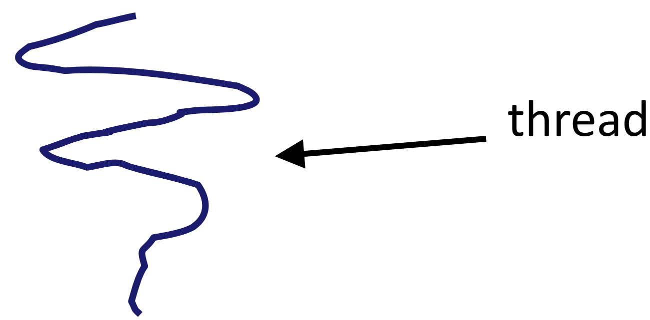
- A version of the program that executes multiple tasks simultaneously
  - Example: Our web server could execute multiple *queries* at the same time
    - While one is waiting for I/O, another can be executing on the CPU
  - Example: Execute queries one at a time, but issue *I/O requests* against different 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



- 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

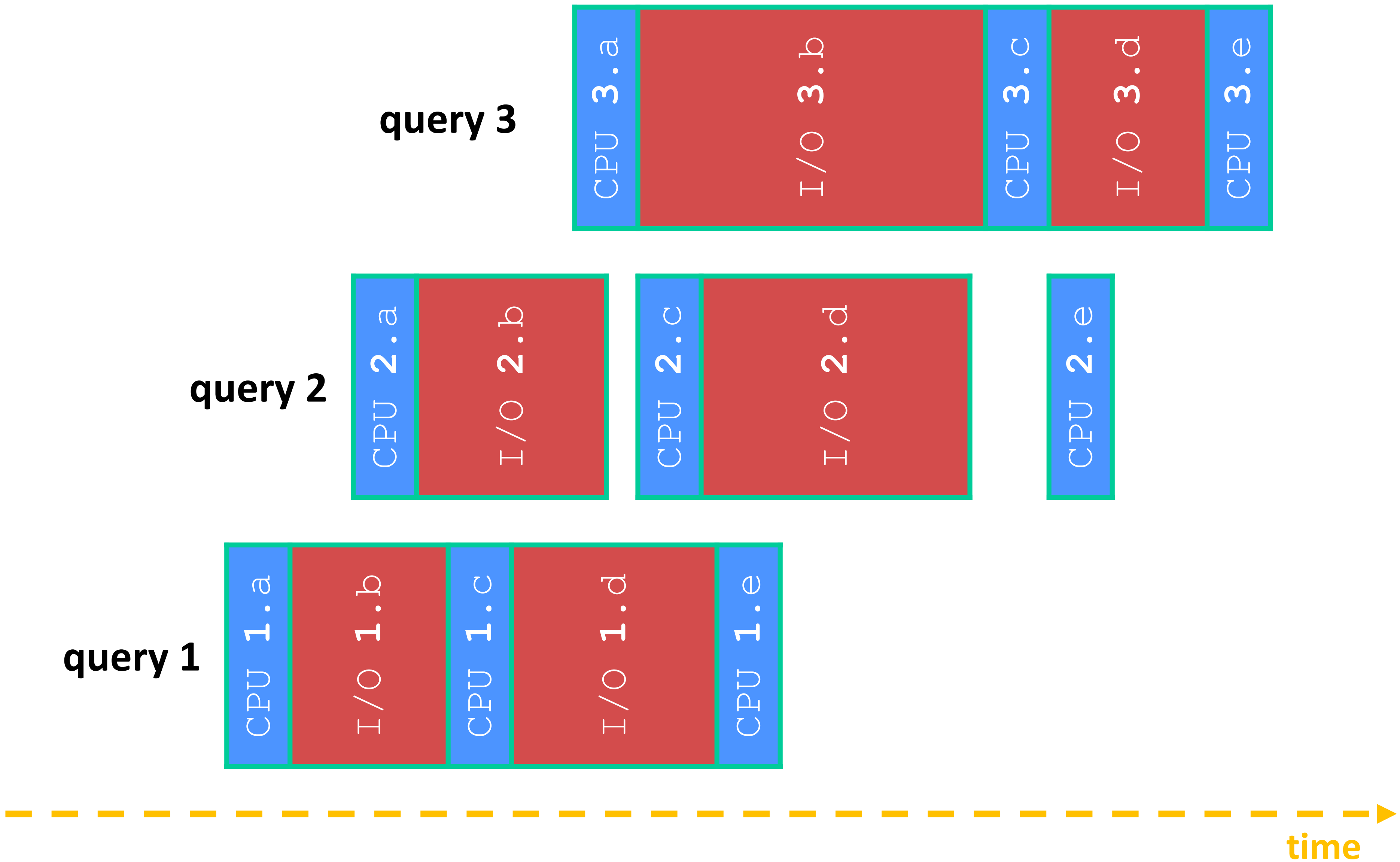
Multithreaded Pseudocode

```
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



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

Outline (next two lectures)

- We'll look at different `searchserver` implementations
  - Sequential
  - Concurrent via dispatching threads – `pthread_create()`
  - Concurrent via forking processes – `fork()`
  - Concurrent via non-blocking, event-driven I/O – `select()`
    - We won't get to this ☹

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

Sequential

- Pseudocode:

```
listen_fd = Listen(port);

while (1) {
  client_fd = accept(listen_fd);
  buf = read(client_fd);
  resp = ProcessQuery(buf);
  write(client_fd, resp);
  close(client_fd);
}
```

- See `searchserver_sequential/`

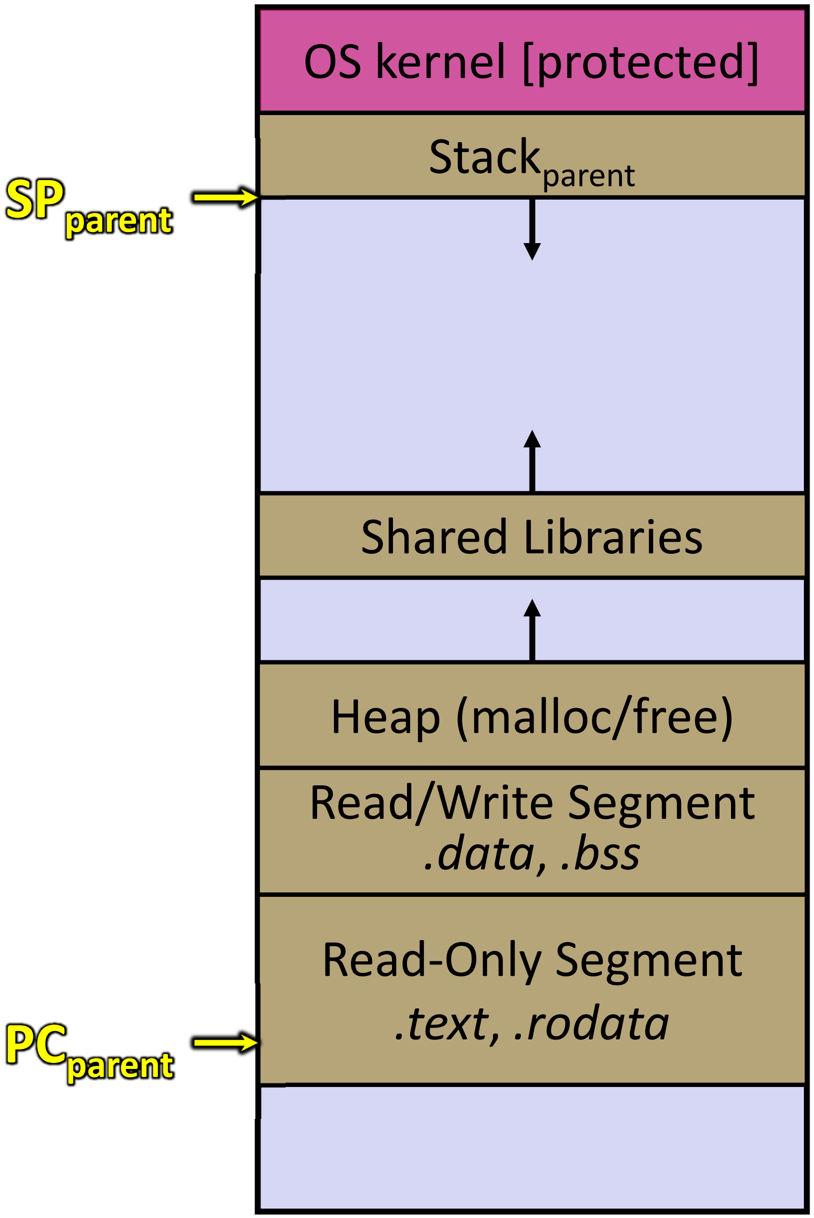
Why Sequential?

- 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)

Threads

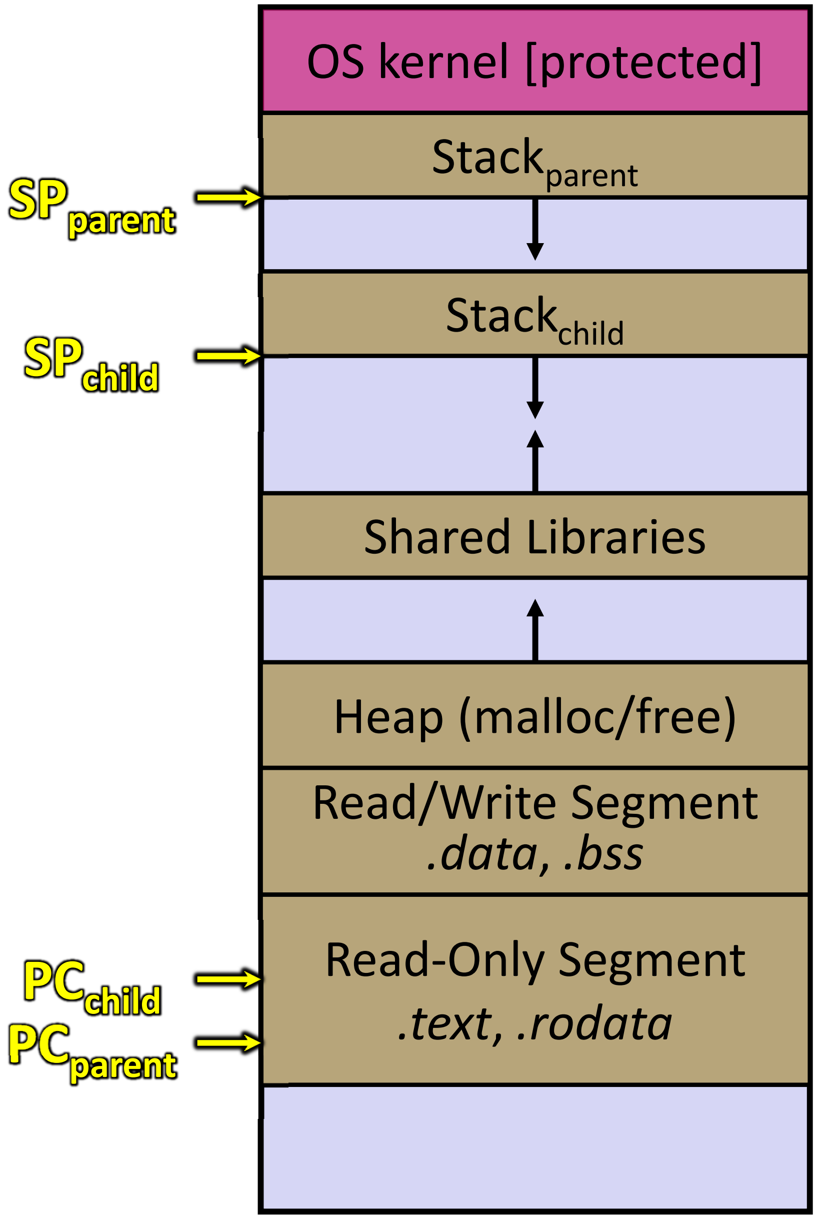
- Threads are like lightweight processes
  - They execute concurrently like processes
    - Multiple threads can run simultaneously on multiple CPUs/cores
  - Unlike processes, threads cohabitate the same address space
    - Threads within a process see the same heap and globals and can communicate with each other through variables and memory
      - But, they can interfere with each other – need synchronization for shared resources
    - Each thread has its own stack

Threads and Address Spaces



- Before creating a thread
  - One thread of execution running in the address space
    - One PC, stack, SP
  - That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Threads and Address Spaces



- After creating a thread
  - *Two* threads of execution running in the address space
    - Original thread (parent) and new thread (child)
    - New stack created for child thread
    - Child thread has its own *values* of the PC and SP
  - Both threads share the other segments (code, heap, globals)
    - They can cooperatively modify shared data

POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Part of the standard C/C++ libraries, declared in `pthread.h`
  - To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command

Creating and Terminating Threads

- ```
  int pthread_create(
            pthread_t* thread,
            const pthread_attr_t* attr,
            void* (*start_routine)(void*),
            void* arg);
  ```
  - Creates a new thread, whose identifier is place in `*thread`, with attributes `*attr` (`NULL` means default attributes)
  - Returns **0** on success and an error number on error (can check against error constants)
  - The new thread runs **`start_routine`**`(arg)`

- ```
  void pthread_exit(void* retval);
  ```
  - Equivalent of **`exit`**`(retval);` for a thread instead of a process
  - The thread will automatically exit once it returns from **`start_routine`**`()`