# Introduction to Concurrency

CSE 333

**Instructor:** Alex Sanchez-Stern

**Teaching Assistants:**
Audrey Seo
Deeksha Vatwani
Derek de Leuw
Katie Gilchrist

# Administrivia

- Ex16 due this morning
- Ex17 due **Monday, August 18th** - last exercise! 🎉
  - On pthreads
  - Will be posted after sections tomorrow, because…
- Sections tomorrow: `pthread` tutorial

- HW4 due a week from today (**Wednesday, August 20th**)
- Final a week from Friday (**Friday, August 22nd**)

# 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 final question
  - Make sure the TA writes down your name

# Administrivia

- Want to know your grade so far? Email me
  - Percentages only, grade points aren't computed yet

- Guest lecturer on Friday: Audrey Seo

# Lecture Outline

- Concurrency
  - **Why is it useful**
  - Concurrency with threads
  - Concurrency with processes
  - Concurrency with events

# Building a Web Search Engine

- We've already built:
  - An on-disk index
    - A map from <*word*> to <*list of documents containing the word*>
  - 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're building:
  - Something that reads HTTP requests from the network and returns results

**Disk**

**CPU**

**Network**

# Sequential Implementation

- Pseudocode for sequential query processor:

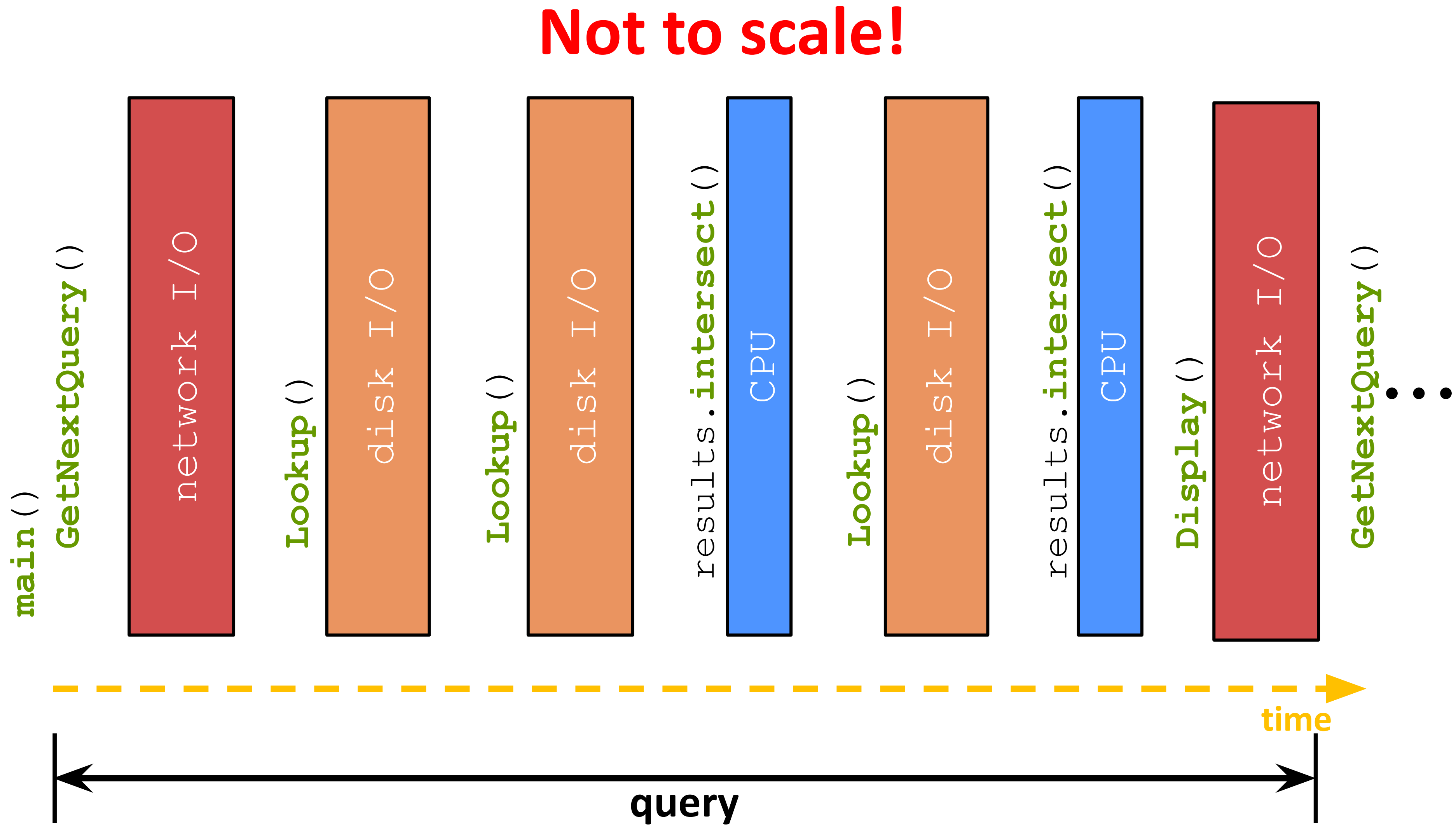
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist {
    doclist.append(file.read(hit));
  }
  return doclist;
}

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);
  }
}
```

Network

Disk

CPU

# Execution Timeline: a Multi-Word Query



# What About I/O-caused Latency?

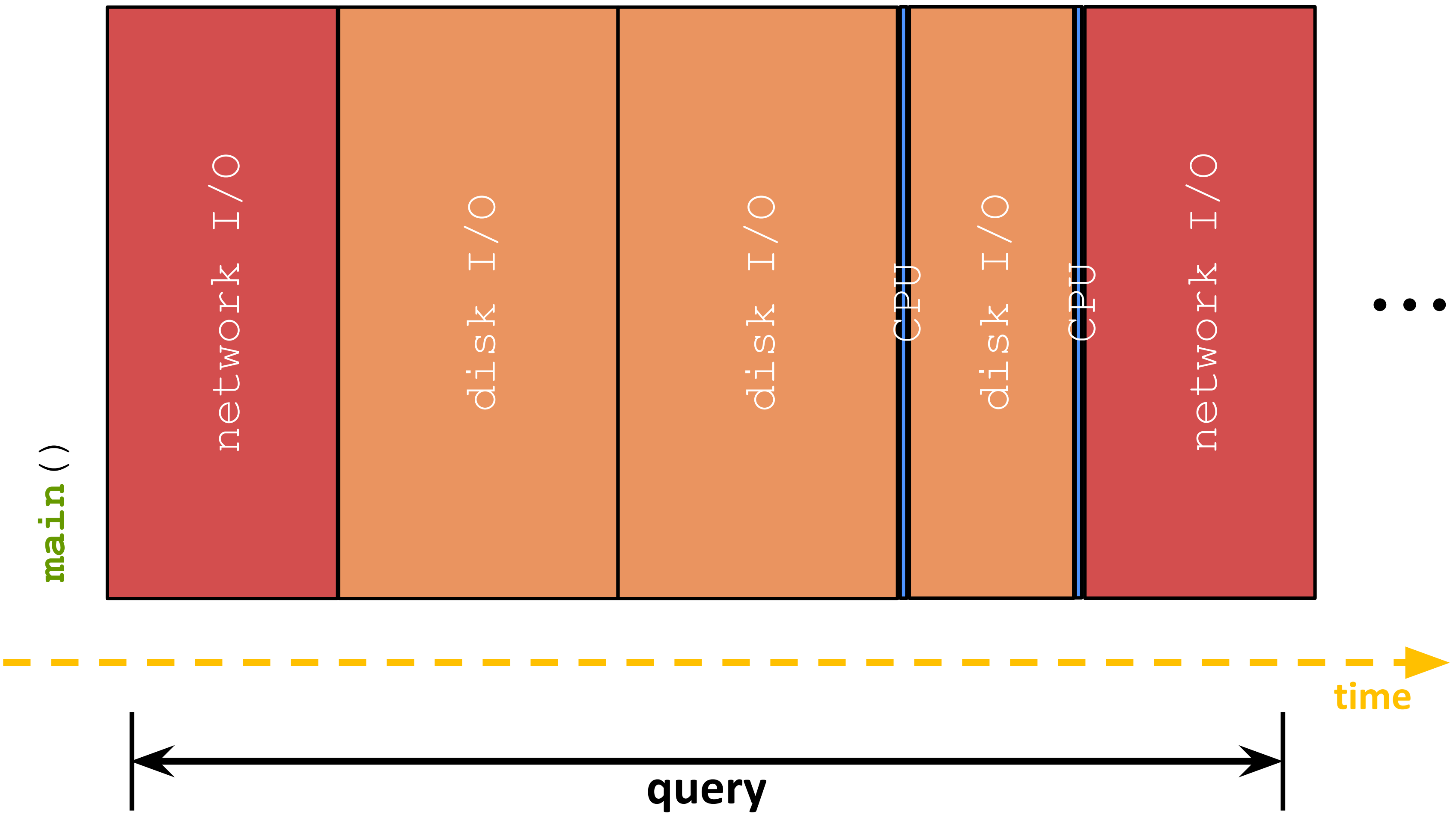
- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)

**Numbers Everyone Should Know**

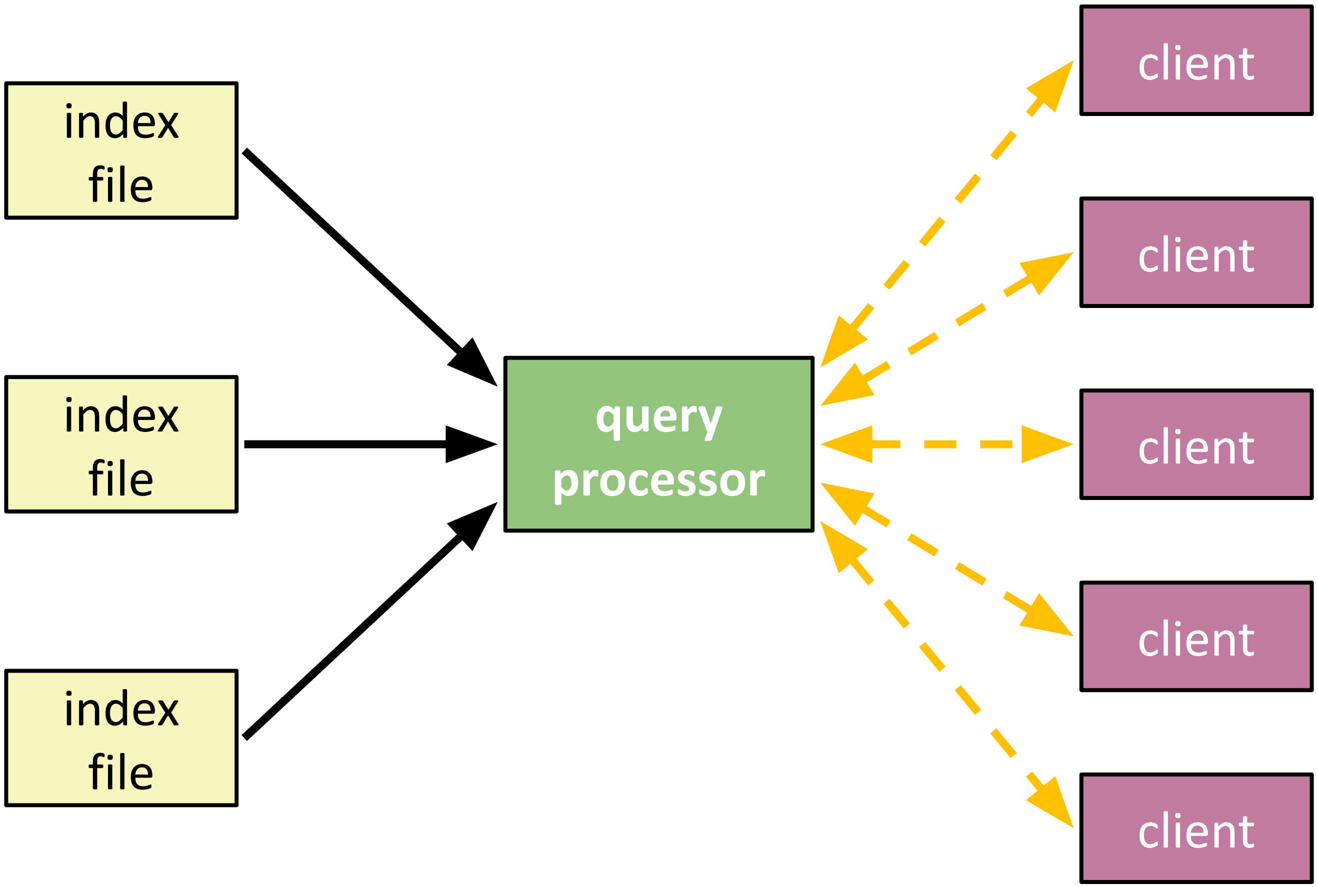
| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 100 ns |
| Main memory reference | 100 ns |
| Compress 1K bytes with Zippy | 10,000 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 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 |

Google

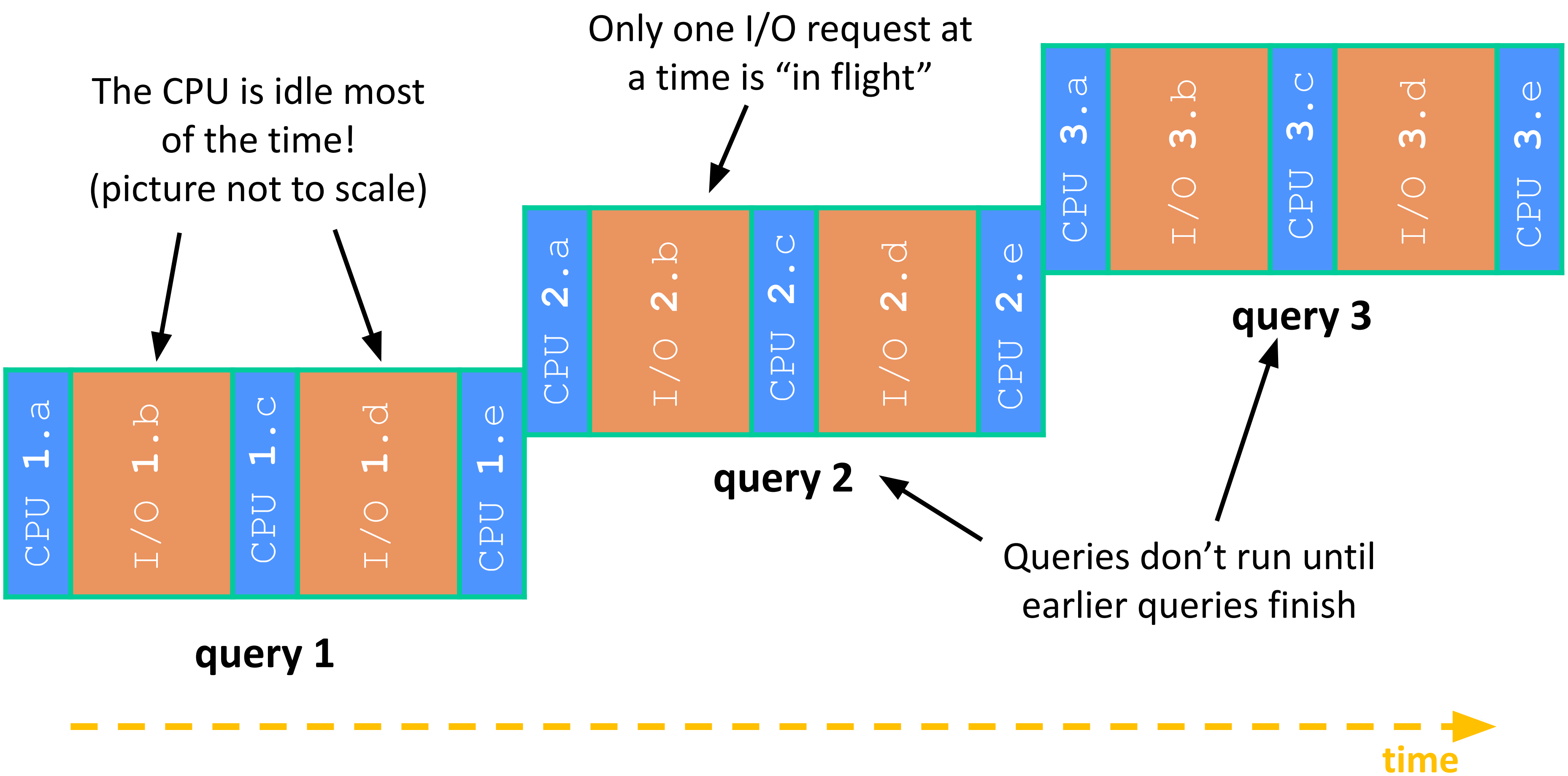
# Execution Timeline: (Closer) To Scale



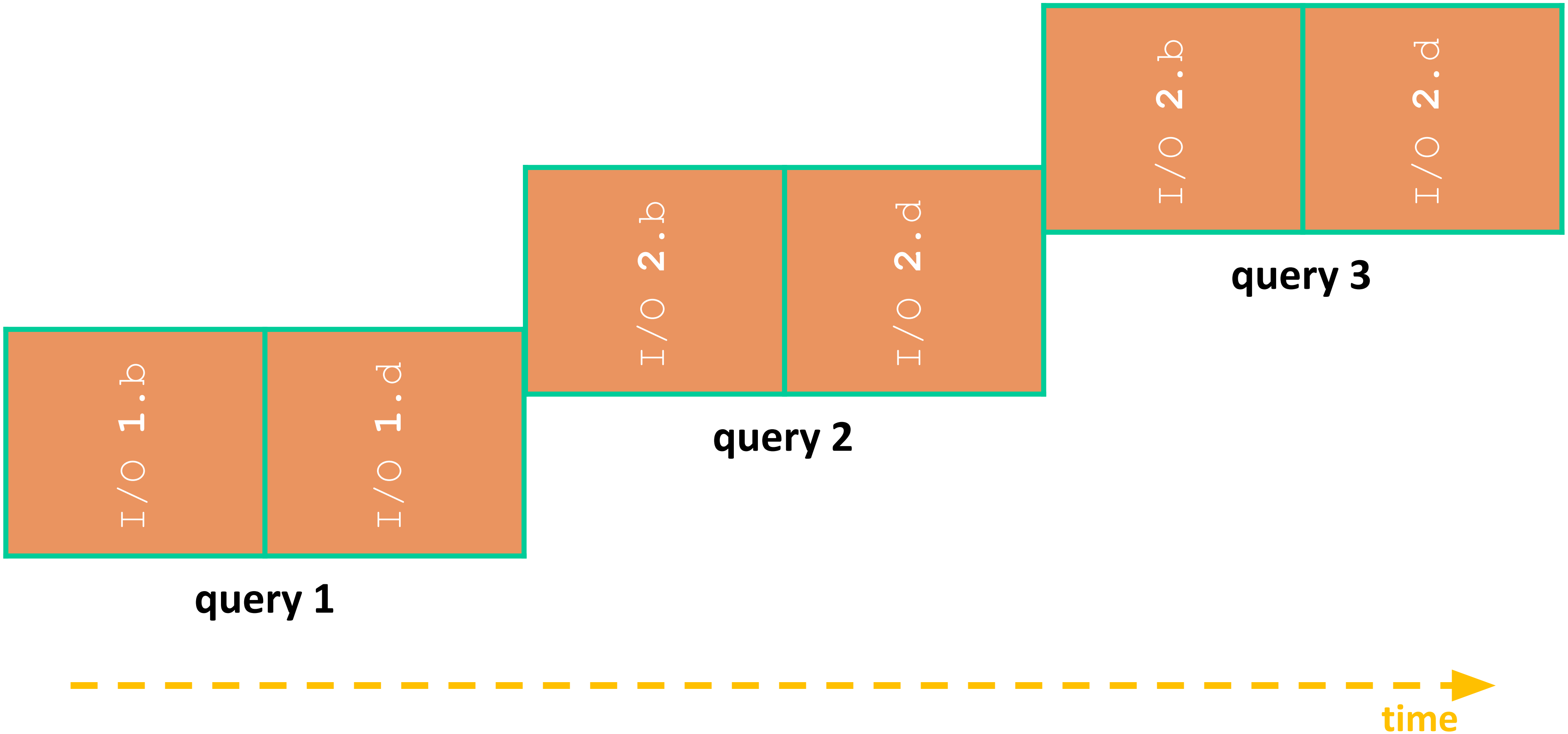
# Web Search Architecture



# Sequential Queries – Simplified



# Sequential Queries: To Scale



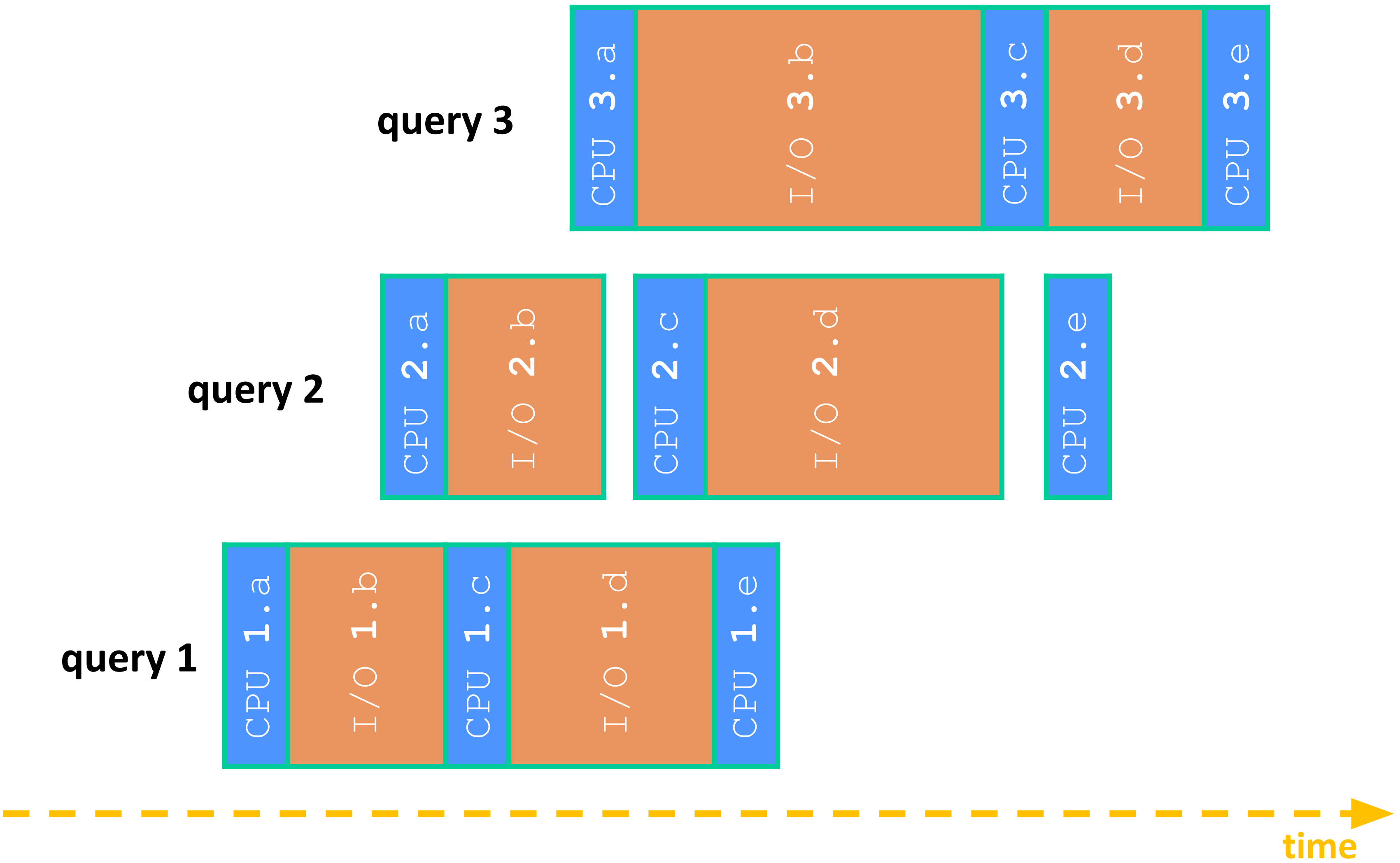
# Sequential Can Be Inefficient

- 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: 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 != parallelism
  - Parallelism is when multiple CPUs work simultaneously on 1 job

# Concurrent Queries – Simplified



# A Concurrent Webserver Implementation

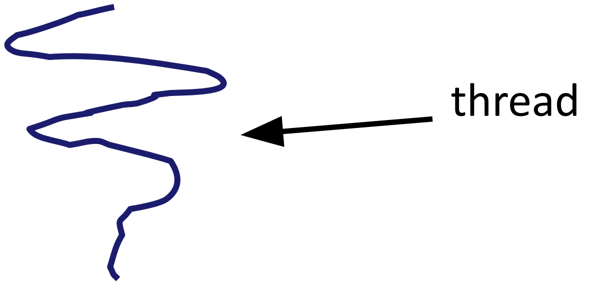
- Use multiple workers
  - As a query arrives, create a new worker to handle it
    - The worker reads the query from the console, issues read requests against files, assembles results and writes to the console
    - The worker uses blocking I/O; the worker alternates between consuming CPU cycles and blocking on I/O
  - The OS context switches between workers
    - While one is blocked on I/O, another can use the CPU
    - Multiple workers' 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.

# Lecture Outline

- Concurrency
  - Why is it useful
  - **Concurrency with threads**
  - Concurrency with processes
  - Concurrency with events

# 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

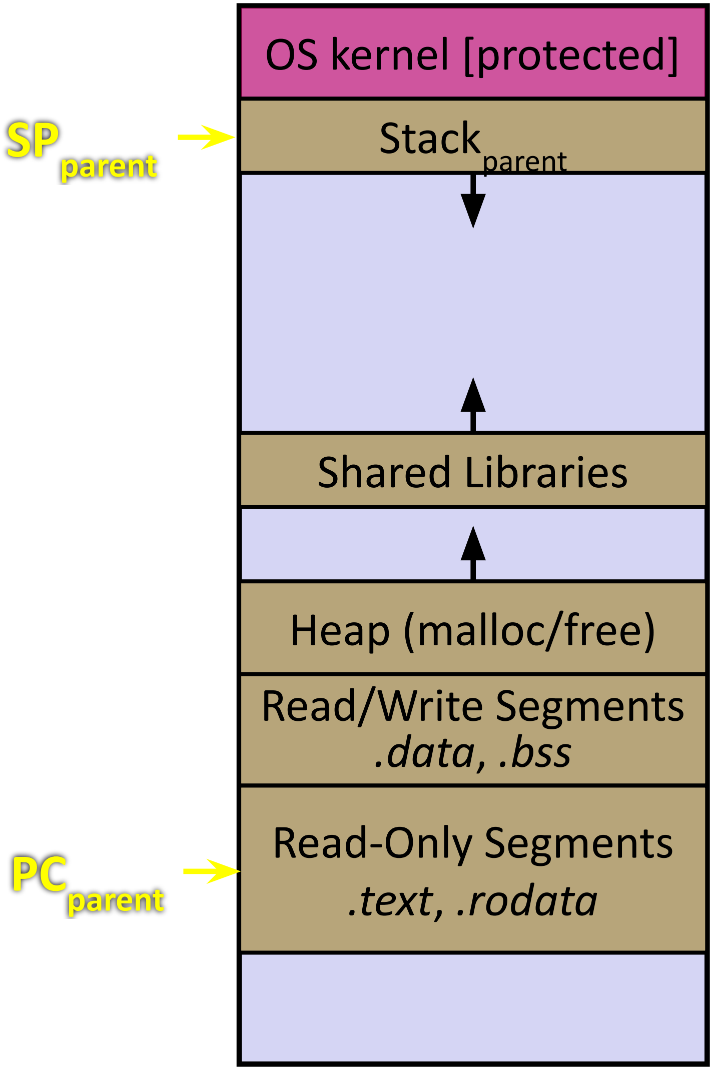



- In most modern OS's:
  - Process: address space, OS resources/process attributes
  - Thread: stack, stack pointer, program counter, other registers

# Threads

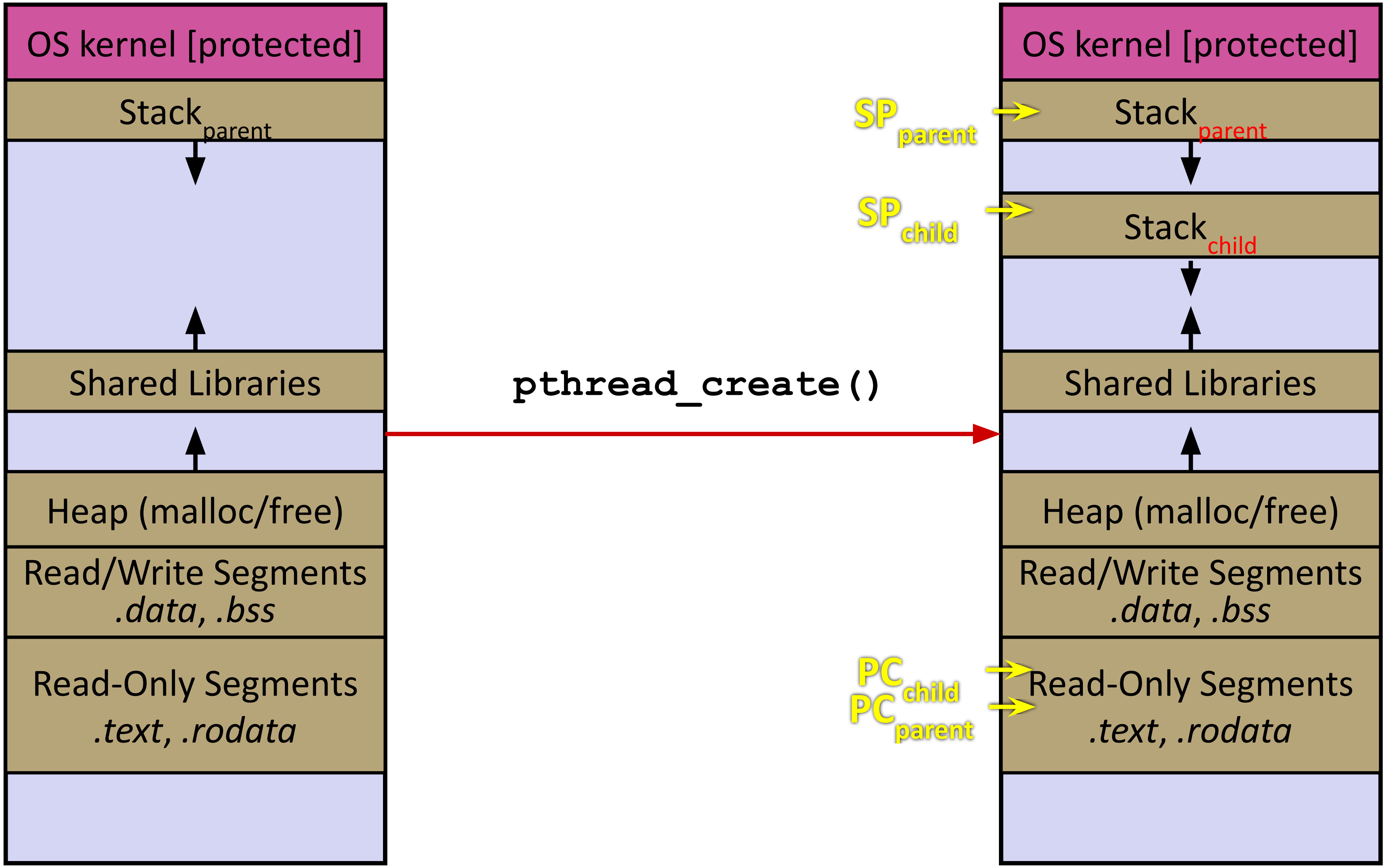
- Threads execute concurrently like processes
  - OS's often treat them, not processes, as the unit of scheduling
- Every process has at least one thread
- Each thread has its own stack and saved registers
- Unlike processes, threads share memory
  - All threads access the memory of the process
    -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

# Threads in the Address Space




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

# Threads in the Address Space



# 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;
}

void ProcessQuery() {
  results = Lookup(query_words[0]);
  foreach word in query[1..n]
    results = results.intersect(Lookup(word));
  Display(results);
}
```

# Threads vs Processes

- 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

# Lecture Outline

- Concurrency
  - Why is it useful
  - Concurrency with threads
  - **Concurrency with processes**
  - Concurrency with events

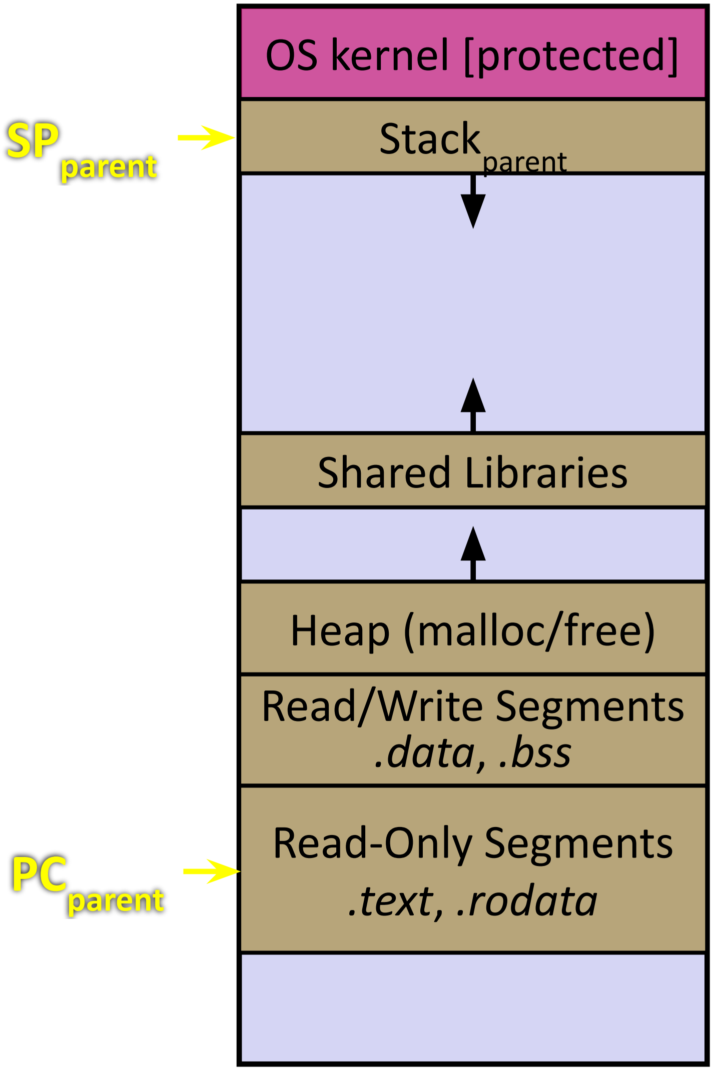
# Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No locks or other synchronization needed
  - No need for language support; OS provides "fork"

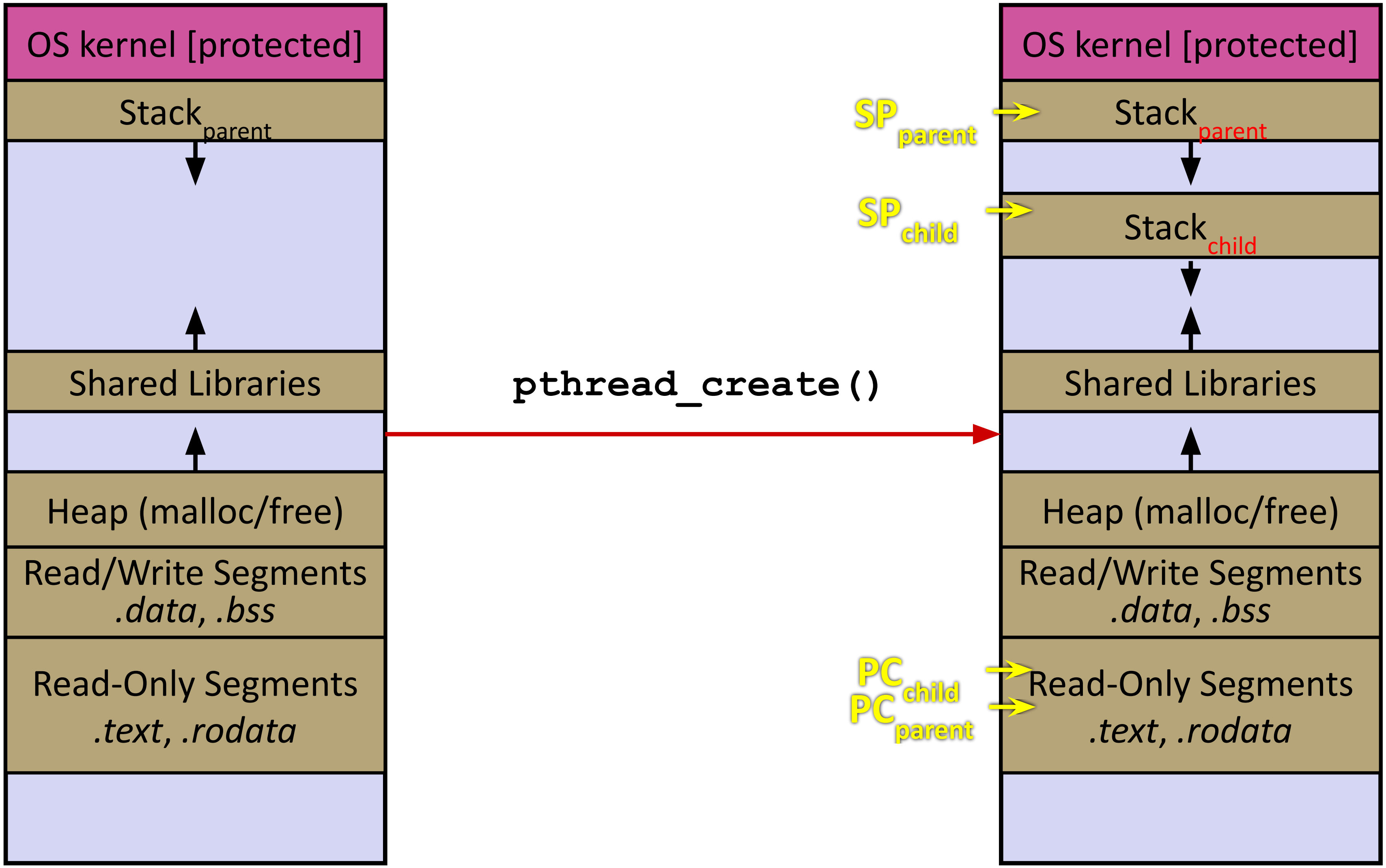
- Disadvantages:
  - More overhead than threads during creation and context switching
  - Cannot easily share data between processes – typically communicate through the file system

# Threads vs. Processes

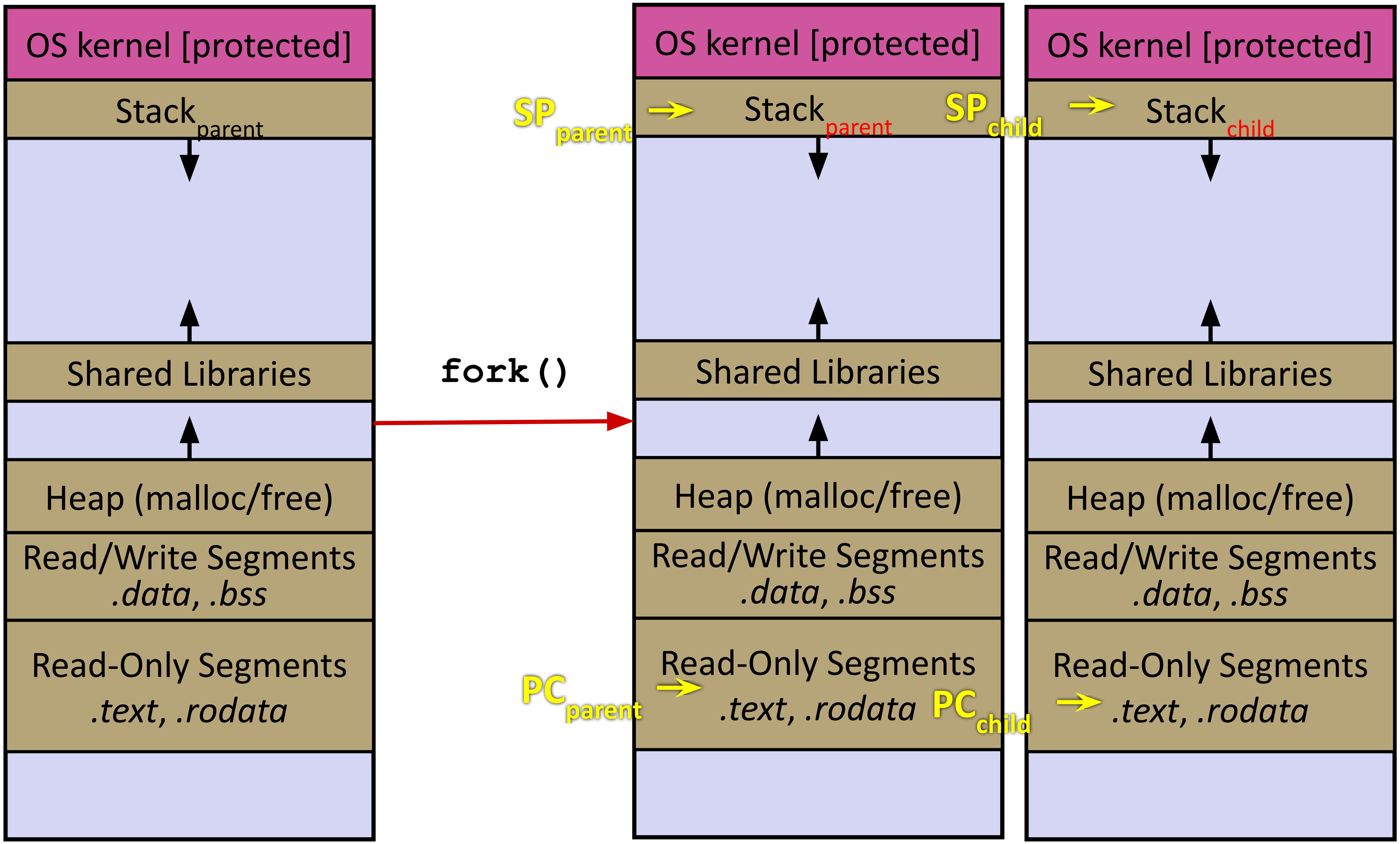



- Before any concurrency
  - One thread of execution running in one address space
  - One PC, stack, SP

# Threads vs. Processes



# Threads vs. Processes



# Lecture Outline

- Concurrency
  - Why is it useful
  - Concurrency with threads
  - Concurrency with processes
  - **Concurrency with events**

# 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

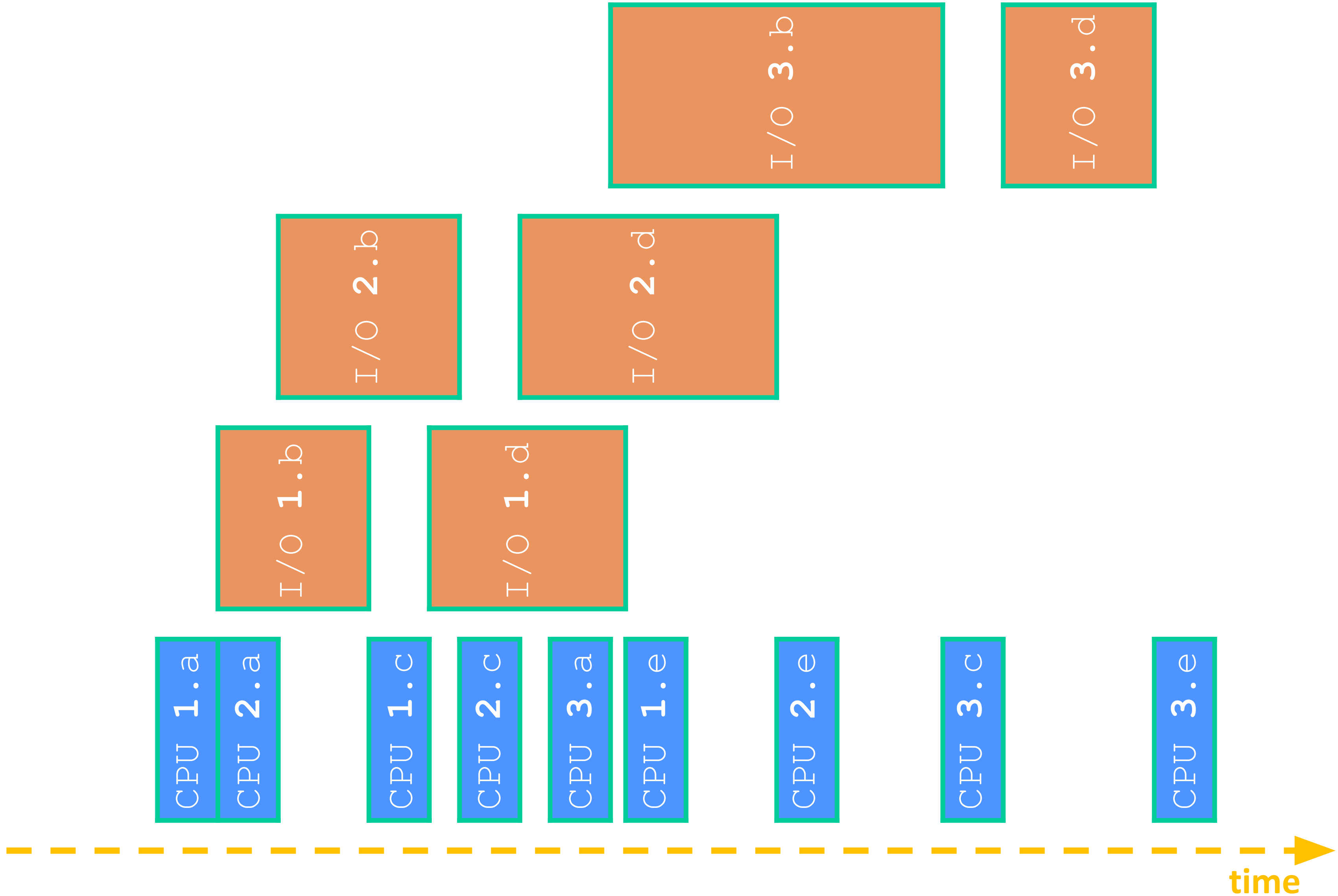
- Your program (almost never) blocks on I/O

# 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

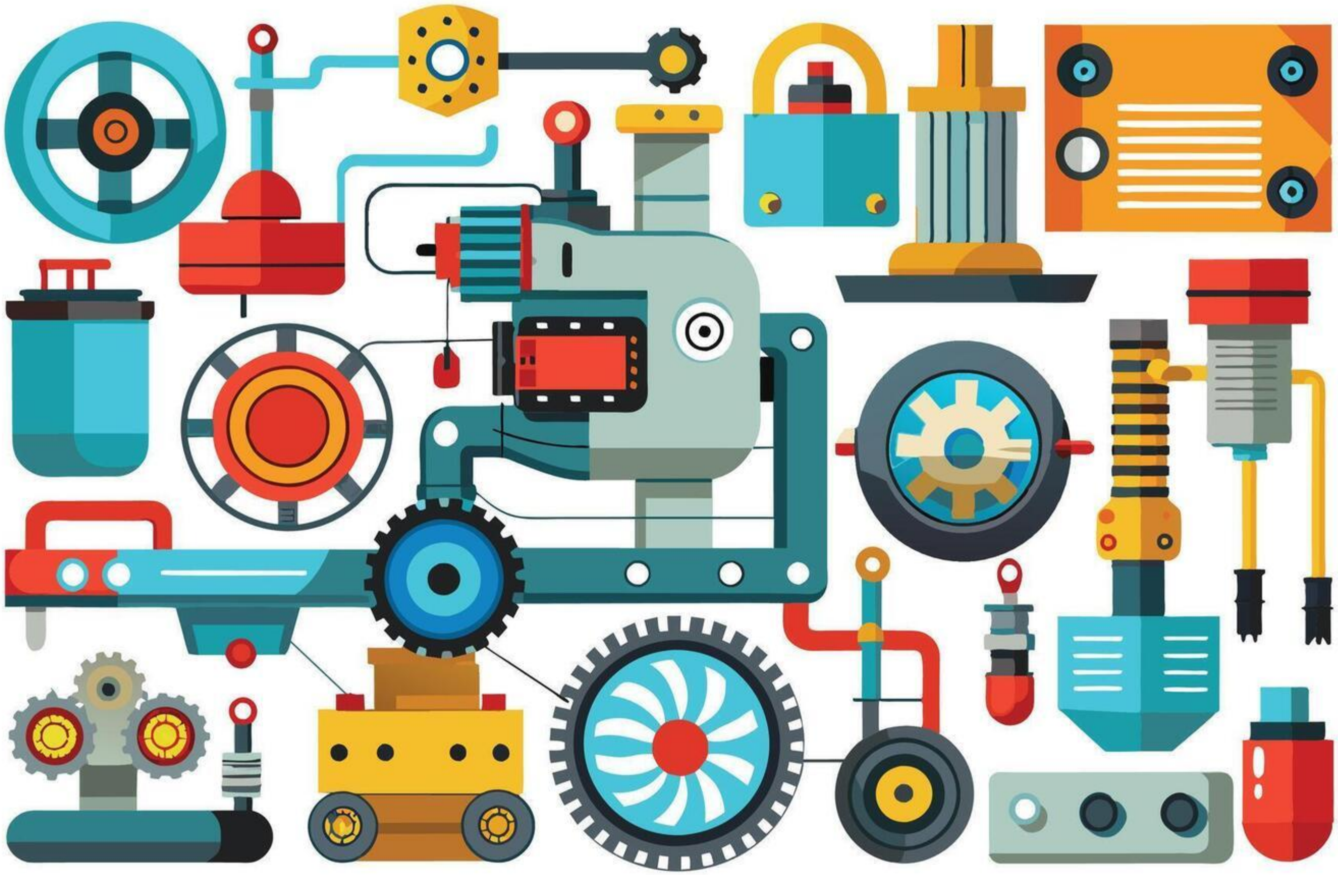
- Reading from the network can truly *block* your program
  - Remote computer may wait arbitrarily long before sending data

- 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/writeable
    - 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 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 ☹

# Event-Driven Web Server



# 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
  - Even faster than threads

- 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
- We won't actually cover Event-driven concurrency further in this class

# Don't Forget

- Ex17 due **Monday, August 18th** - last exercise! 🎉

- Sections tomorrow: `pthread` tutorial

- HW4 due a week from today (**Wednesday, August 20th**)
- Final a week from Friday (**Friday, August 22nd**)