Concurrency: Intro and Threads

CSE 333 Autumn 2019

Instructor: Hannah C. Tang

Teaching Assistants:

Dao Yi	Farrell Fileas	Lukas Joswiak
Nathan Lipiarski	Renshu Gu	Travis McGaha
Yibo Cao	Yifan Bai	Yifan Xu

Poll Everywhere

pollev.com/cse333

About how long did Exercise 16 take?

A. 0-1 Hours
B. 1-2 Hours
C. 2-3 Hours
D. 3-4 Hours
E. 4+ Hours
F. I prefer not to say

Administrivia

- HW4 due two Thursdays from now (12/05)
  - You can use at most ONE late day

- Short week next week:
  - Wed lecture cancelled (but OH available in AND 223 at 11:30)
  - 🦃 Fri holiday 🦃

Lecture Outline

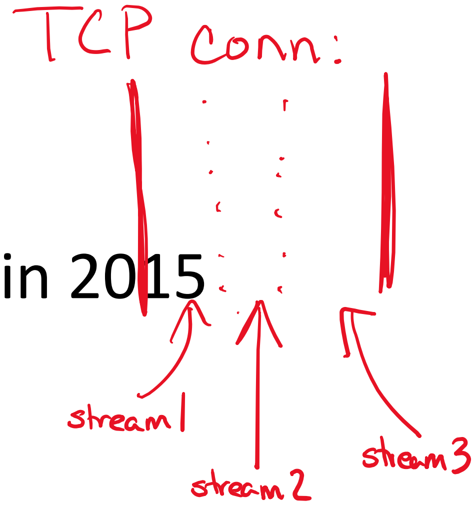
- **HTTP/2 Review**
- From Query Processing to a Search Server
- Intro to Concurrency
- Threads
- Search Server with pthreads

HTTP/1.1 Feature: Persistent connections

- Establishing a TCP connection is costly
  - Multiple network round trips to set up the TCP connection
  - TCP has a feature called "slow start"; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks

- A web page consists of multiple objects and a client probably visits several pages on the same server
  - Bad idea: separate TCP connection for each object
  - Better idea: single TCP connection, multiple requests

HTTP/2 (2 of 3)



- Based on Google SPDY (2010) ; standardized in 2015
- Features:
  - Same core request/response model (GET, POST, OK, …)
  - Binary protocol
    - Easier parsing by machines (harder for humans)
    - Sizes in headers, not discovered as requests are processed
    - Headers compressed and deduplicated by default!
  - Multiple data steams multiplexed on single TCP connection
    - Fixes “head-of-line blocking”
    - With priorities on the streams!
  - Server push and more…

Lecture Outline

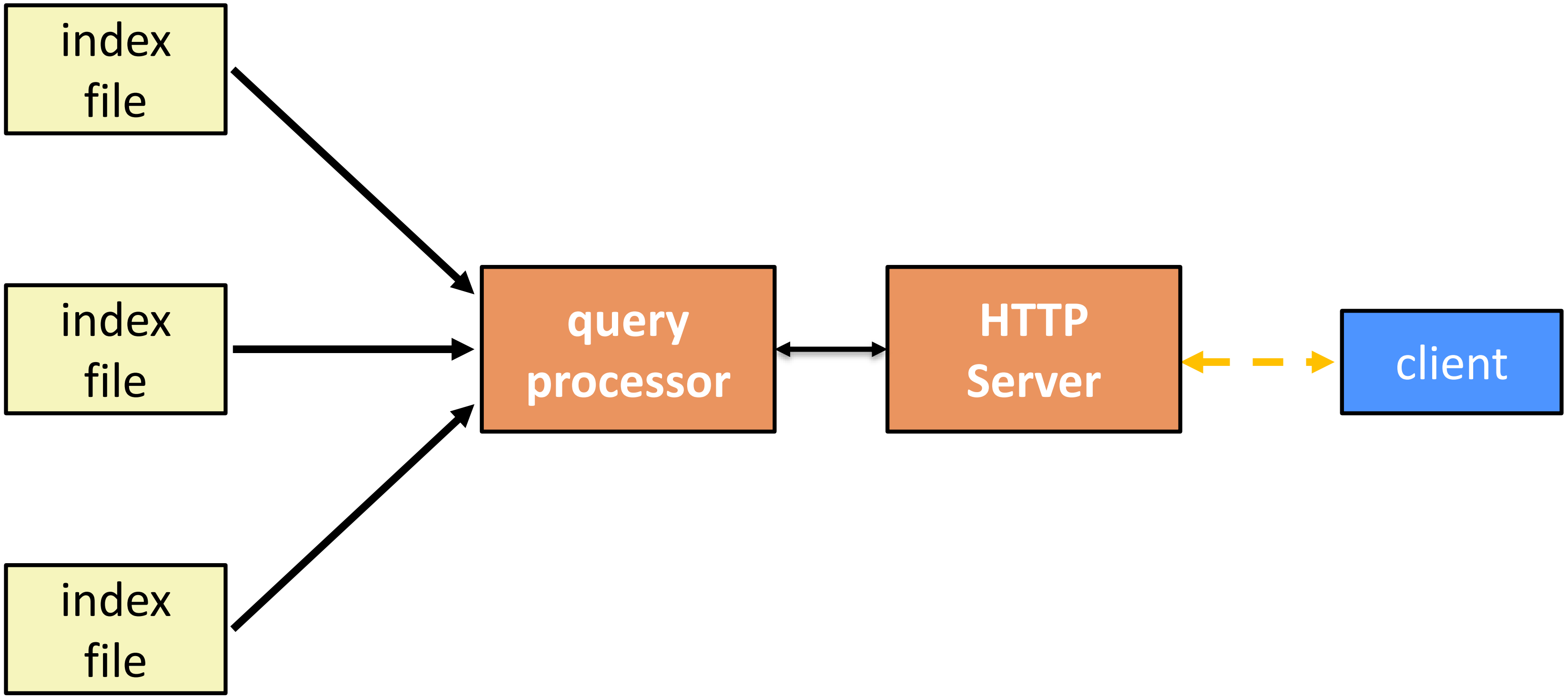
- HTTP/2 Review
- **From Query Processing to a Search Server**
- Intro to Concurrency
- Threads
- Search Server with pthreads

Building a Web Search Engine

- We have:
  - 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

Search Engine Architecture



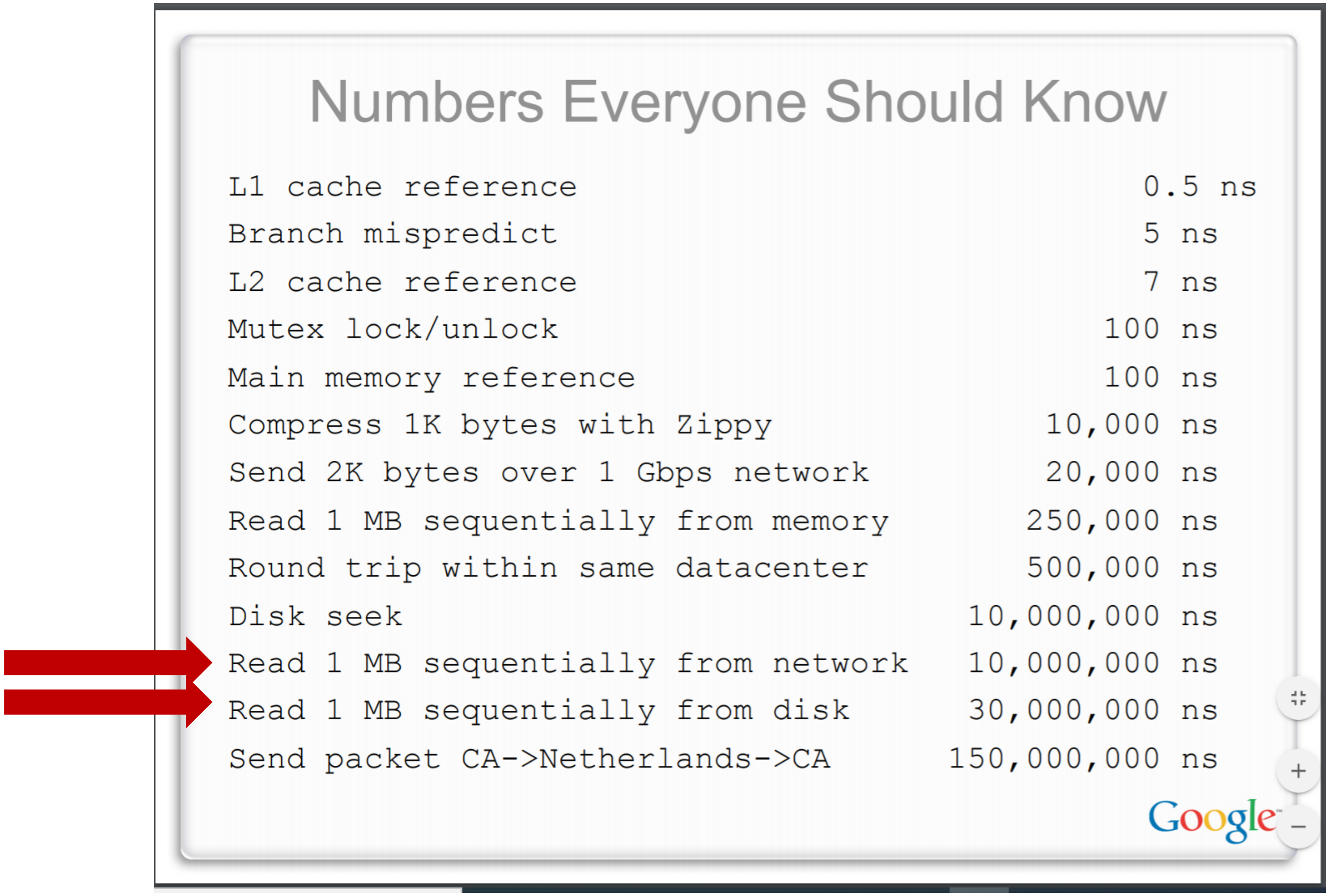
Search Engine (Pseudocode)

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

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

server read/write loop

What About I/O-caused Latency?

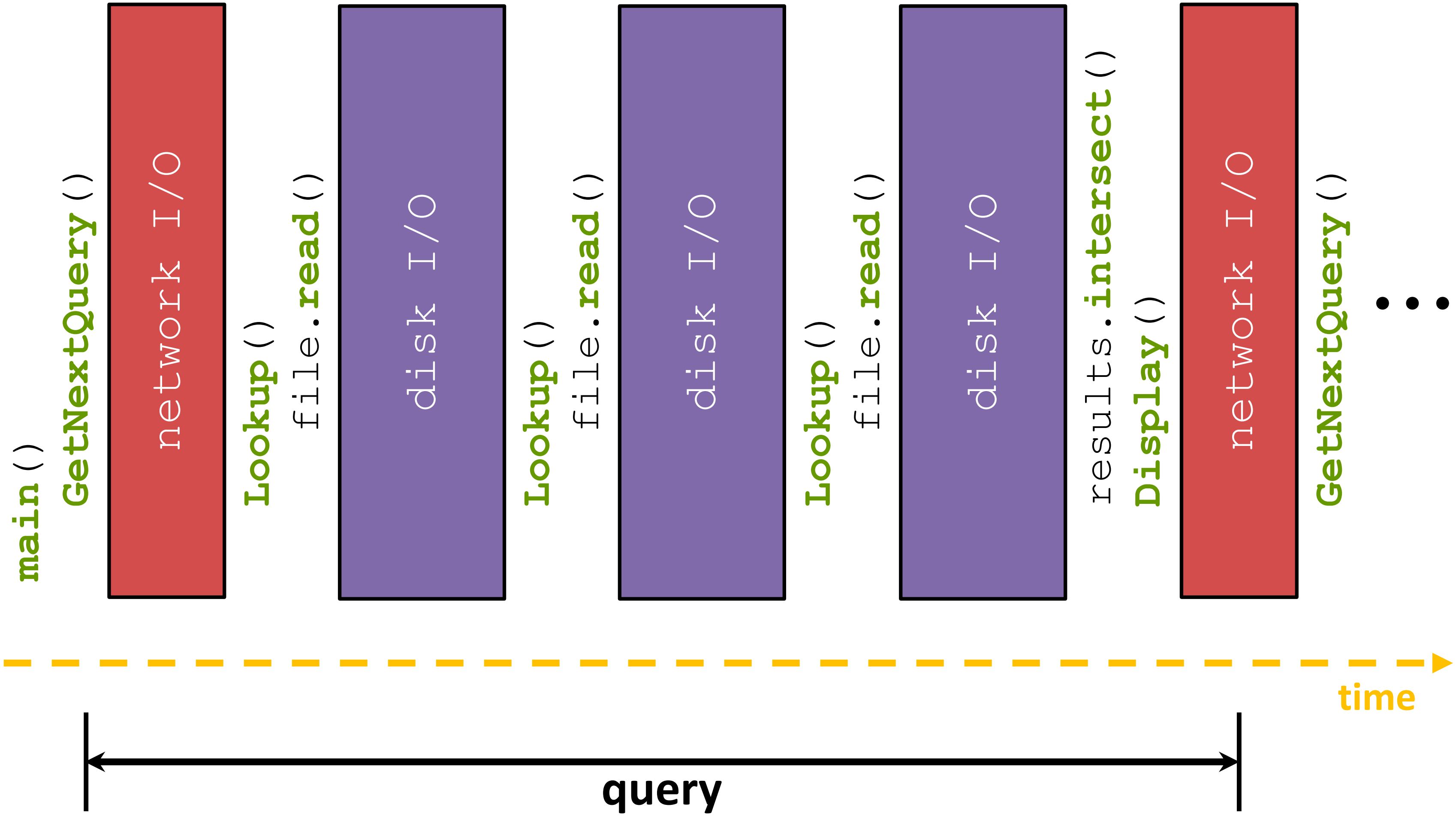
- Jeff Dean's "Numbers Everyone Should Know" (LADIS '09)



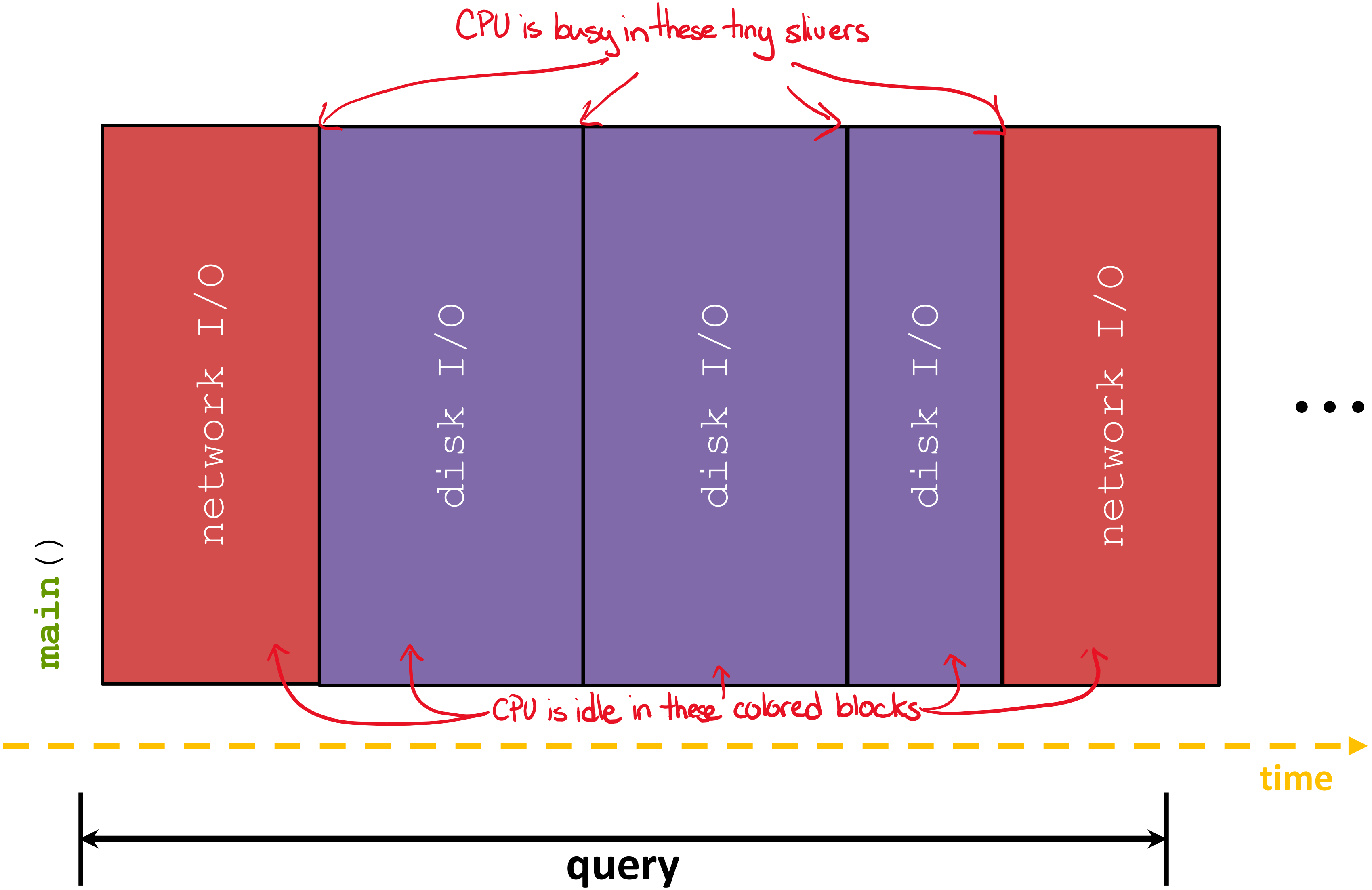
Numbers Everyone Should Know

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

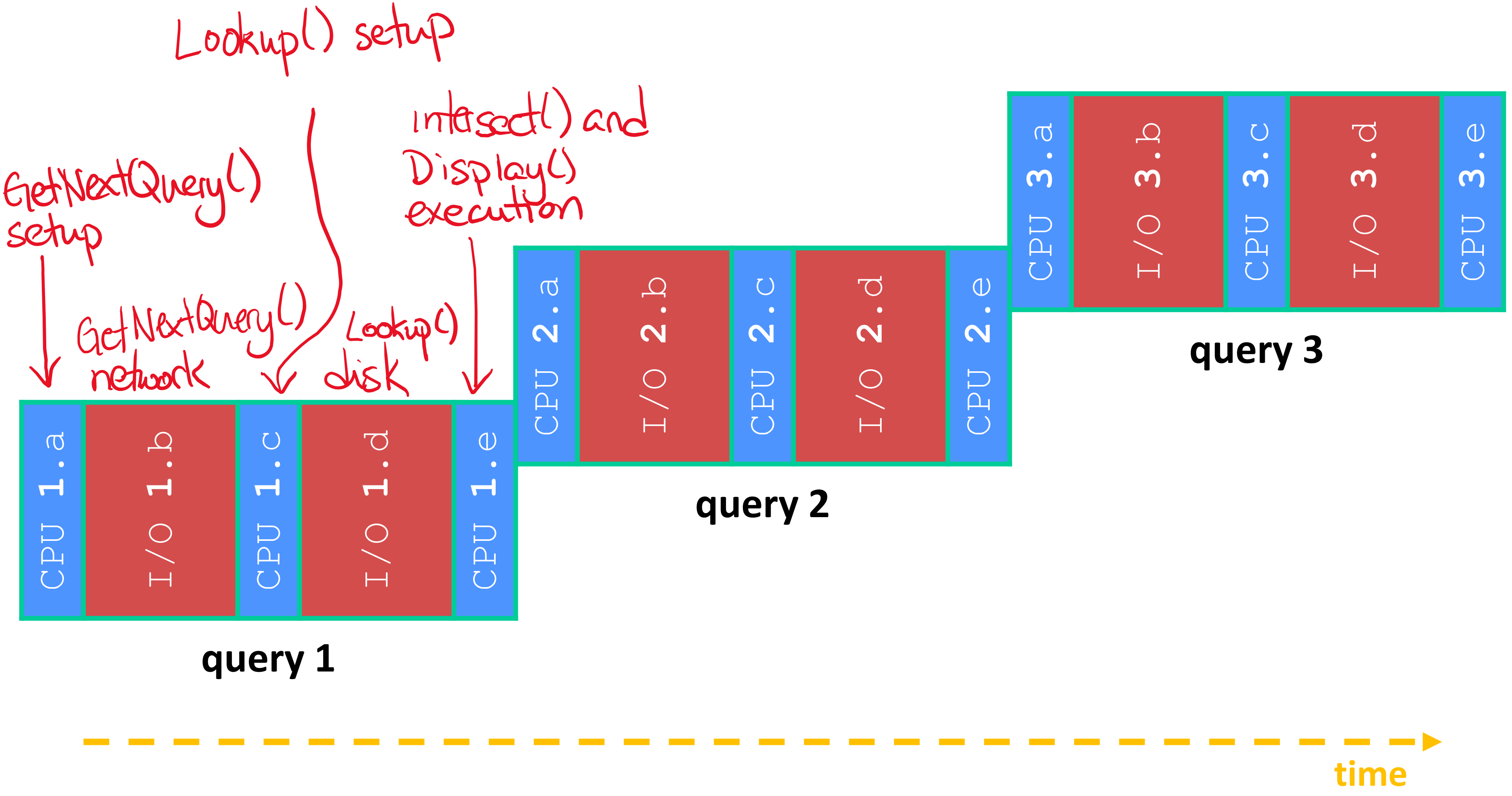
Execution Timeline: One multi-word query



Execution Timeline: To Scale



Multiple (Single-Word) Queries



Lookup() setup

GetNextQuery() setup

GetNextQuery() network

intersect() and Display() execution

Lookup() disk

CPU 1.a

I/O 1.b

CPU 1.c

I/O 1.d

CPU 1.e

query 1

CPU 2.a

I/O 2.b

CPU 2.c

I/O 2.d

CPU 2.e

query 2

CPU 3.a

I/O 3.b

CPU 3.c

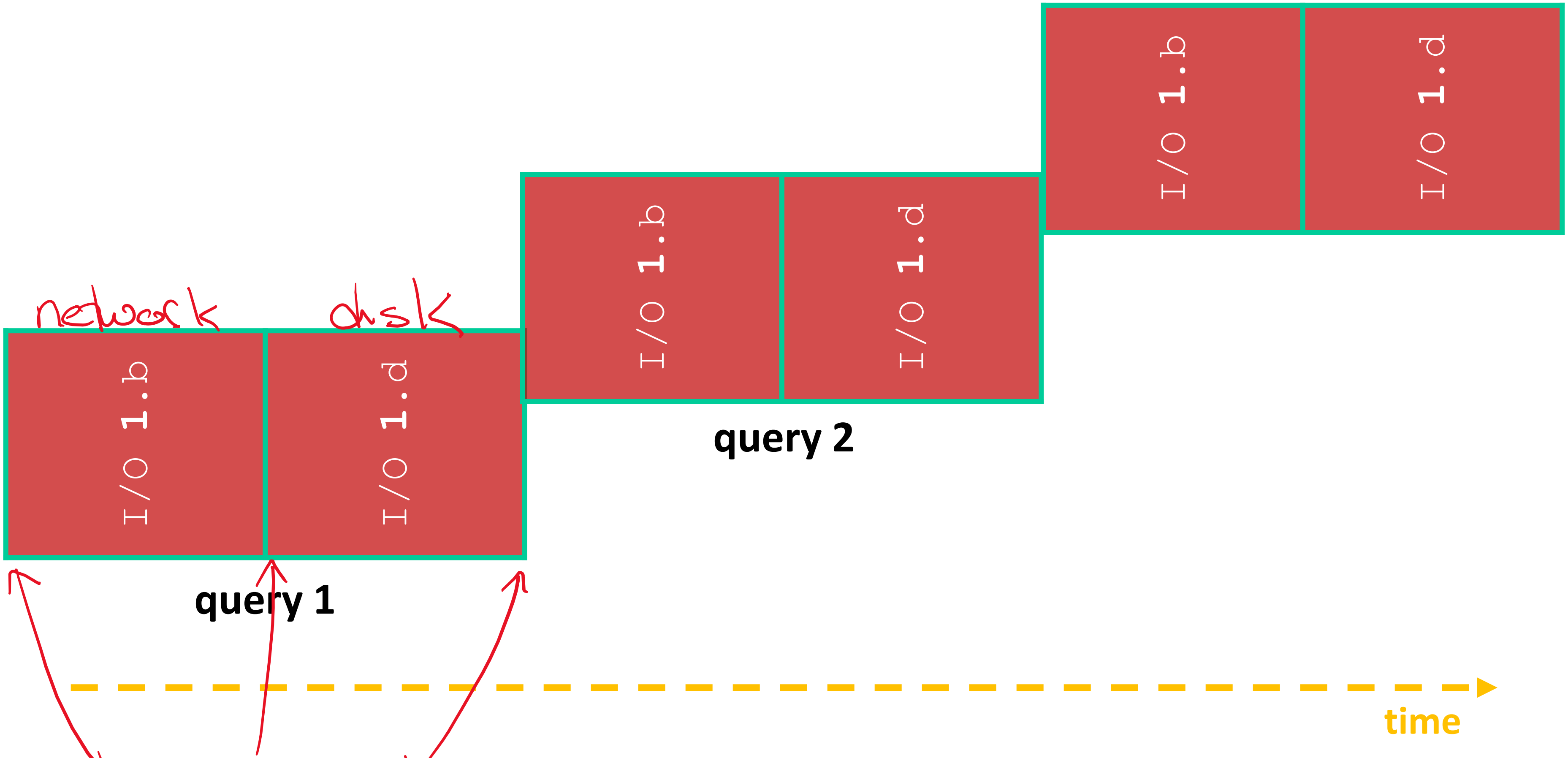
I/O 3.d

CPU 3.e

query 3

time

Multiple Queries: To Scale



network

disk

`I/O 1.b`

`I/O 1.d`

query 1

`I/O 1.b`

`I/O 1.d`

query 2

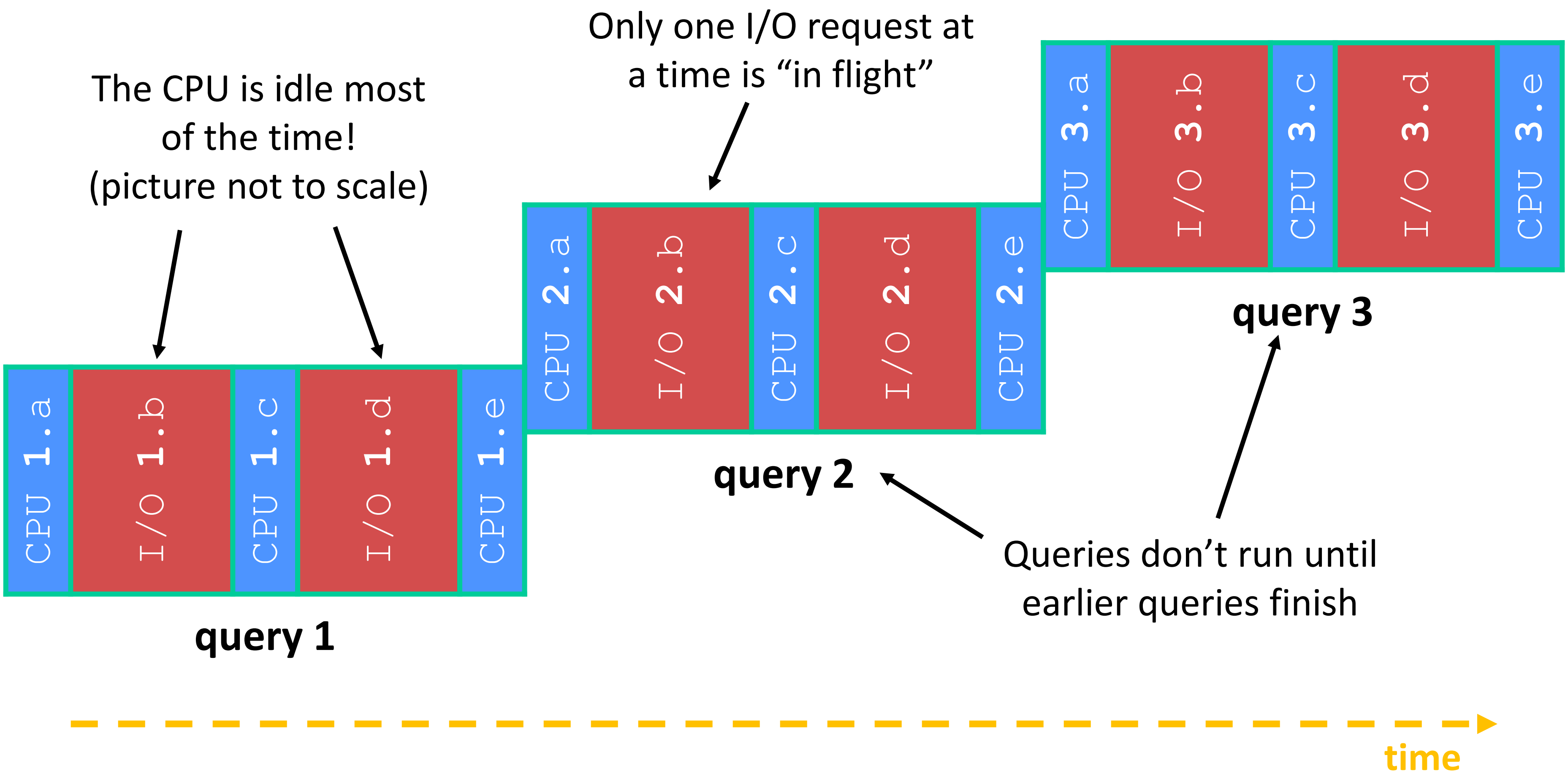
`I/O 1.b`

`I/O 1.d`

time

CPU is busy in these
tiny slivers

Uh-Oh (1 of 2)

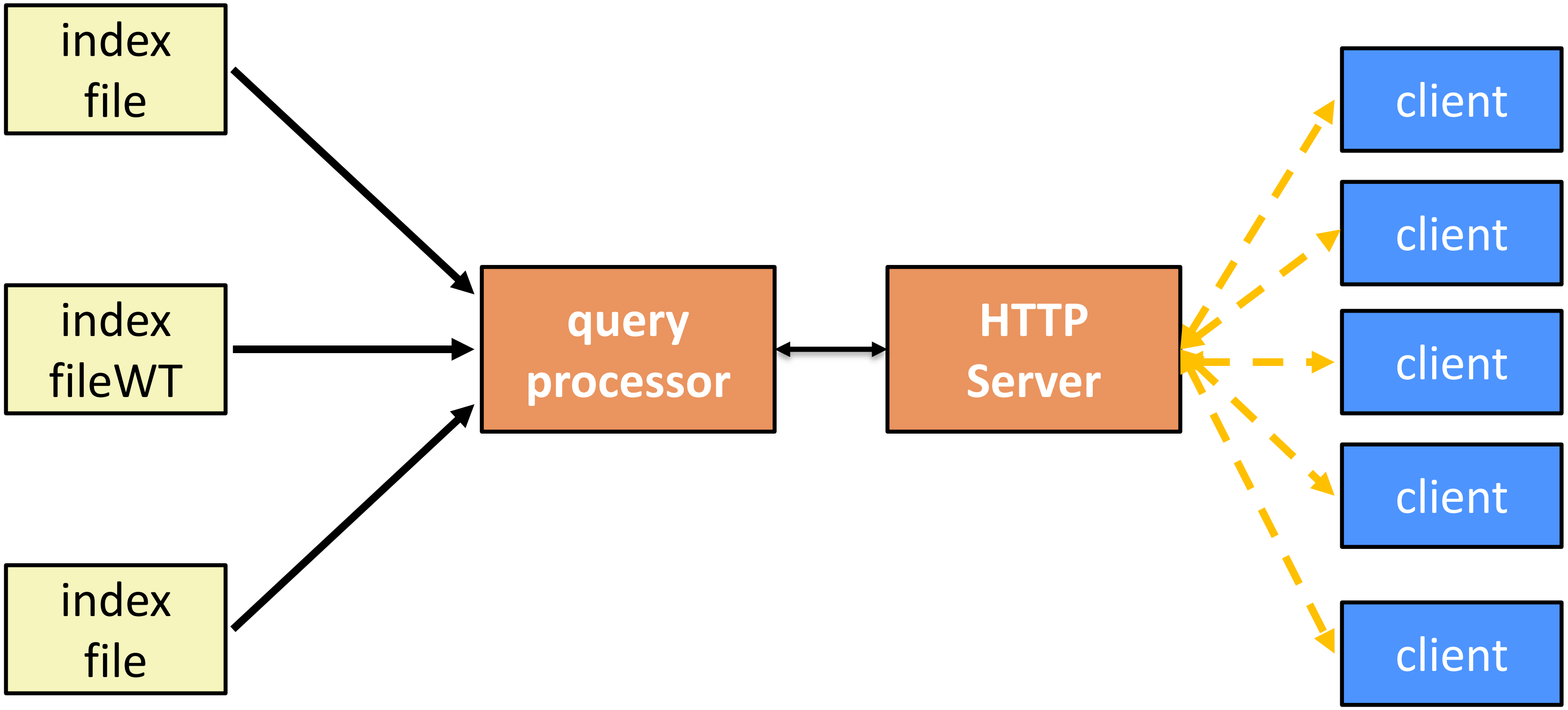


The CPU is idle most of the time!
(picture not to scale)

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

Queries don’t run until earlier queries finish

Uh-Oh (2 of 2)


index file
index fileWT
index file
query processor
HTTP Server
client
client
client
client
client

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

- HTTP/2 Review
- From Query Processing to a Search Server
- **Intro to Concurrency**
- Threads
- Search Server with pthreads

Concurrency

- Concurrency != parallelism
  - Concurrency is doing multiple tasks at a time
  - Parallelism is executing multiple CPU instructions *simultaneously*

- 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

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 “workers’” 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**
- Search Server with pthreads

Review: Processes

- To implement a "process", the operating system gives us:
  - Resources such as file handles and sockets
  - Call stack + registers to support (eg, PC, SP)
  - Virtual memory (page tables, TLBs, etc …)

- If we want concurrency, what is the "minimal set" we need to execute a single line of code?

"Worker" 1

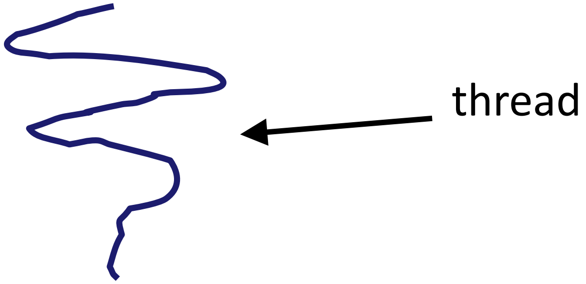
```
bucket = hash(word);
hitlist = file.read(bucket);
```

"Worker" 2

```
foreach hit in hitlist {
  doclist.append(file.read(hit));
}
```

Introducing Threads

- Separate the concept of a process from the "*thread of execution*"
  - Usually called a thread, this is a sequential execution stream within a process



- In most modern OS's:
  - Process: address space, OS resources, security 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 were formerly called "lightweight processes"
  - They execute concurrently like processes
    - OS's often treat them, not processes, as the unit of scheduling
    - Parallelism for free! If you have multiple CPUs/cores, can run them simultaneously
  - 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

change virtual mem, registers, PC, SP, etc

- What does the OS do when you switch processes?
  - How does that differ from switching threads?

Multi-threaded Search Engine (Pseudocode)

server read/write loop

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

query processing happens in a thread

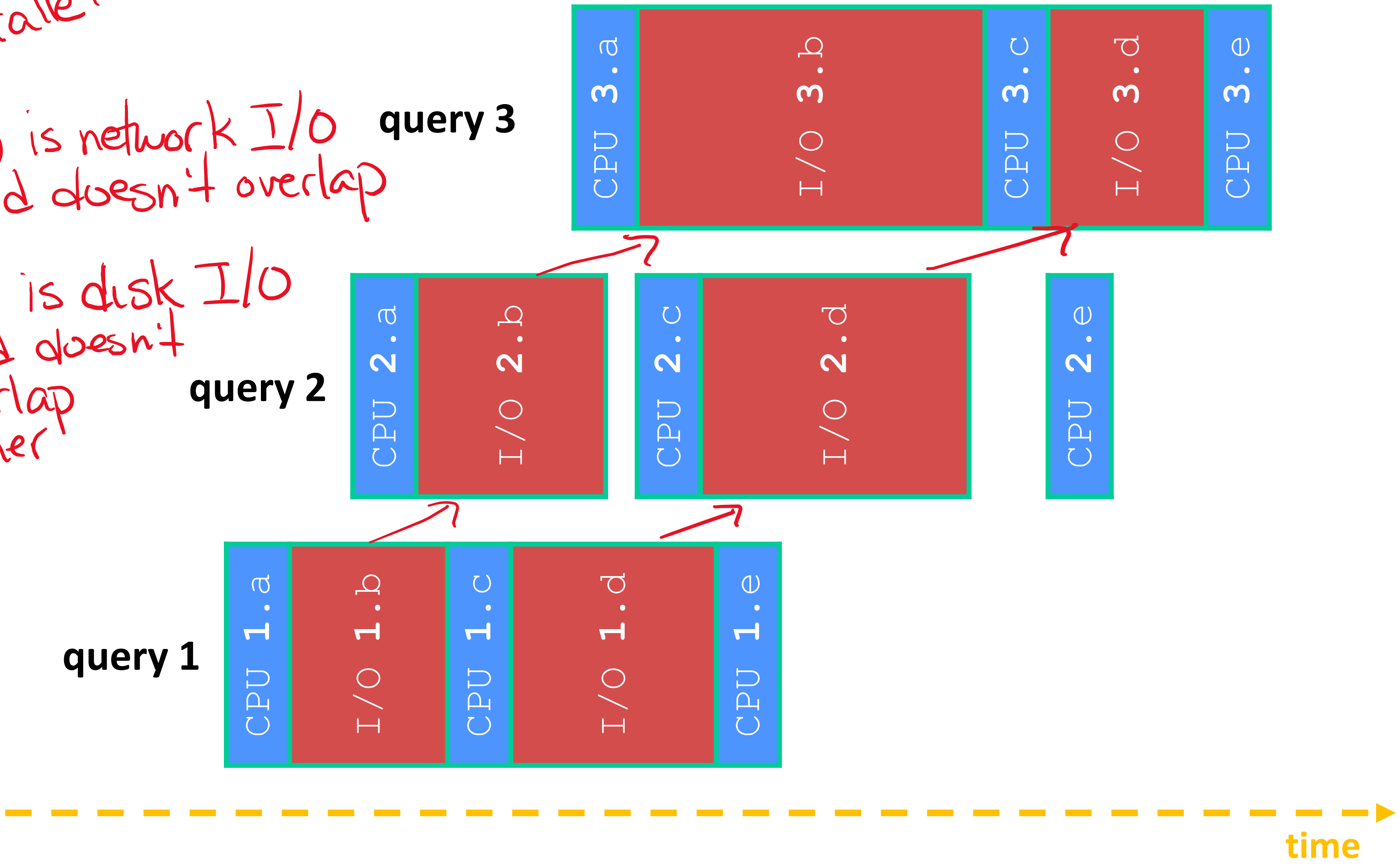
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist
    doclist.append(file.read(hit));
  return doclist;
}

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

Multi-threaded Search Engine (Execution)

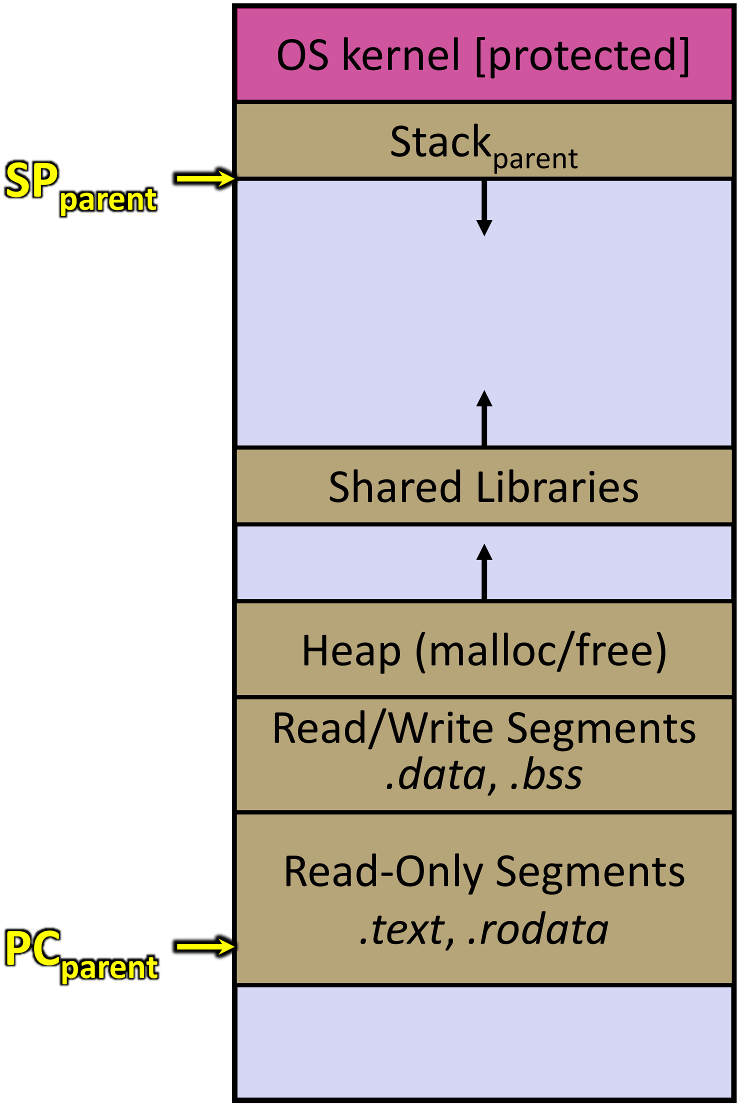
still no parallelism!

(b) is network I/O and doesn't overlap

(d) is disk I/O and doesn't overlap either

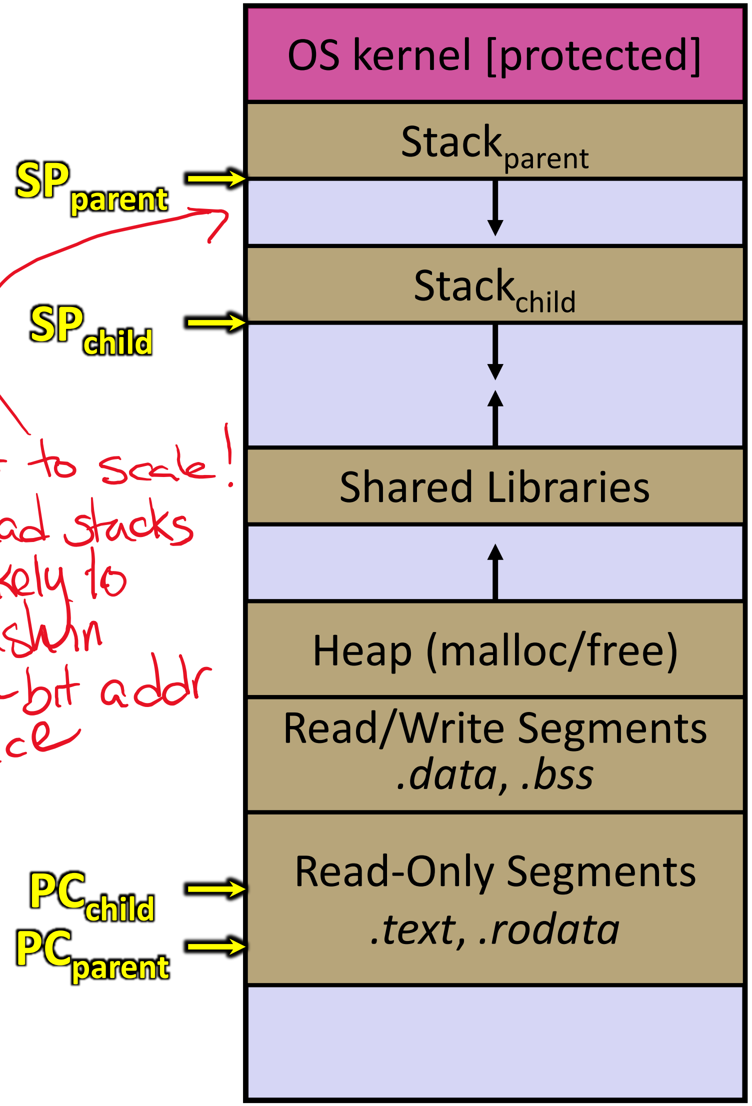


Single-Threaded Address Spaces



- 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()`

Multi-threaded Address Spaces



not to scale!
thread stacks unlikely to crash in 64-bit addr space

- 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

Lecture Outline

- From Query Processing to a Search Server
- Intro to Concurrency
- Threads
- **Search Server with pthreads**

POSIX Threads (pthreads)

- The POSIX APIs for dealing with threads
  - Declared in `pthread.h`
    - Not part of the C/C++ language (cf. Java)
  - 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);
  ```

  initialized by create

  - Creates a new thread into `*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)`

  frequently a struct containing the "real" arguments

-
  ```
  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`**`()`

What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid`**`()`
  - The exit status of the terminated thread is placed in `**retval`

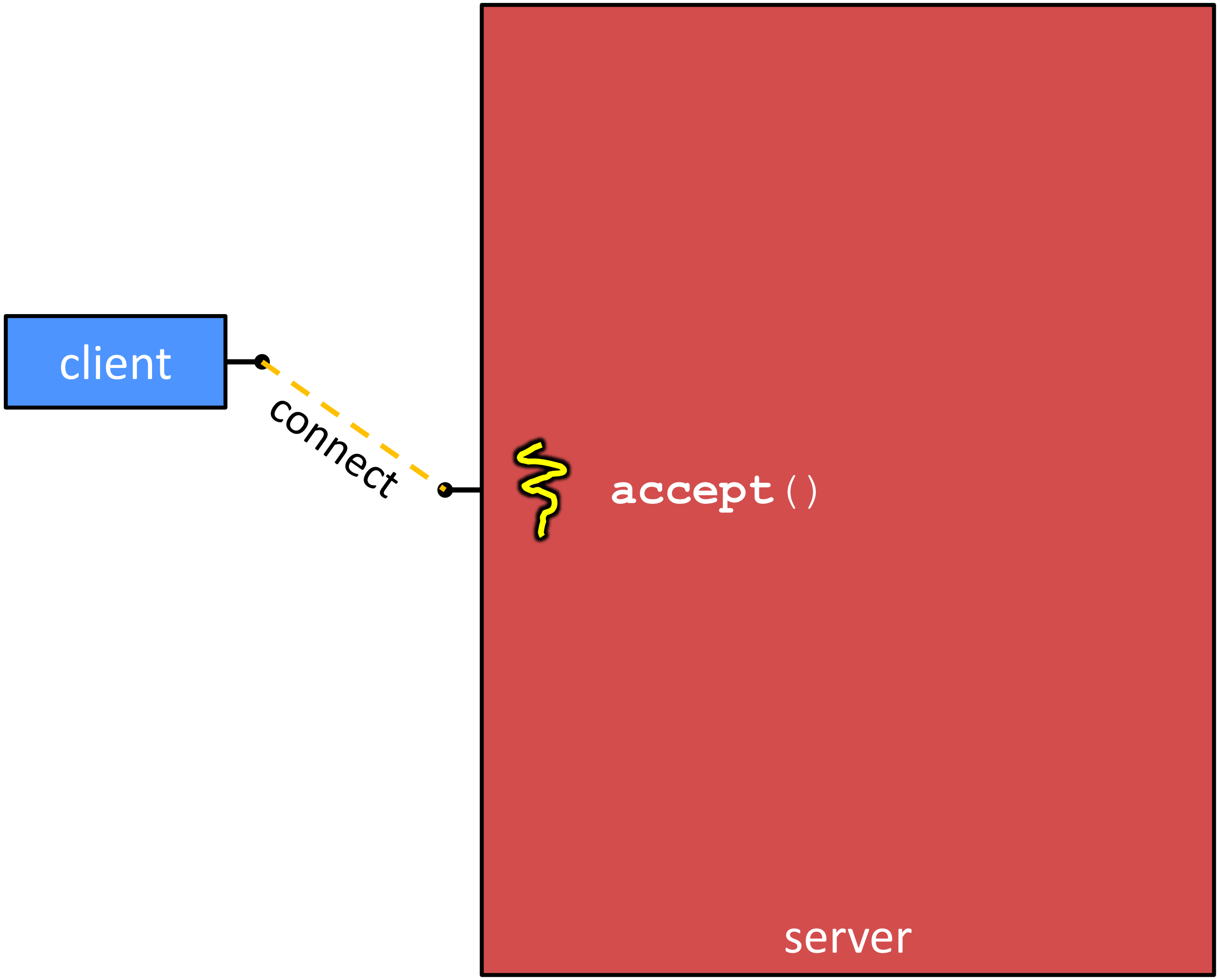
- `int pthread_detach(pthread_t thread);`
  - Mark thread specified by `thread` as detached – it will clean up its resources as soon as it terminates

Multi-threaded Search Engine: Architecture

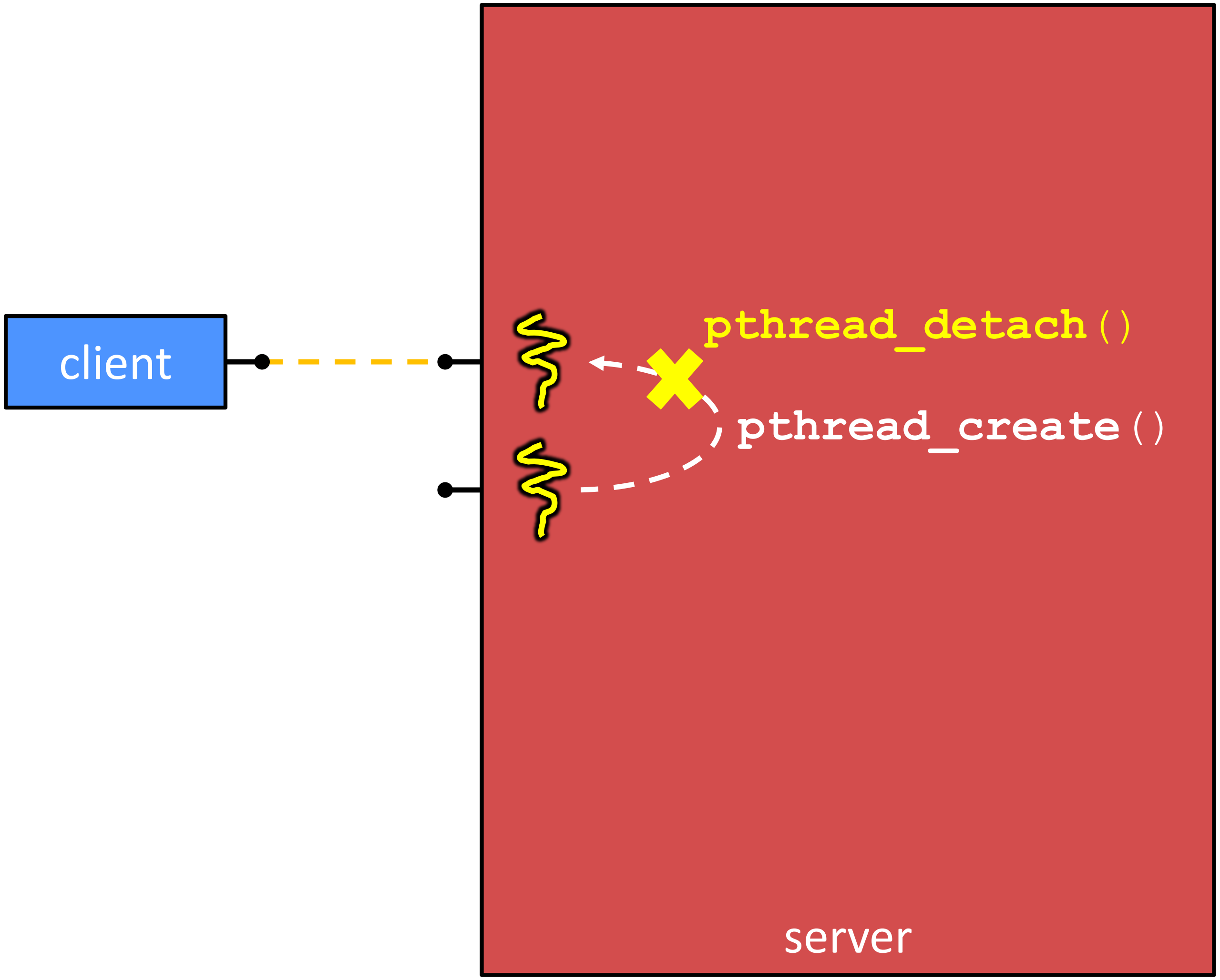
- A single *process* handles all of the connections, but a parent *thread* dispatches (creates) a new thread to handle each connection
  - The child thread handles the new connection and subsequent I/O, then exits when the connection terminates

- See `searchserver_threads/` for code if curious

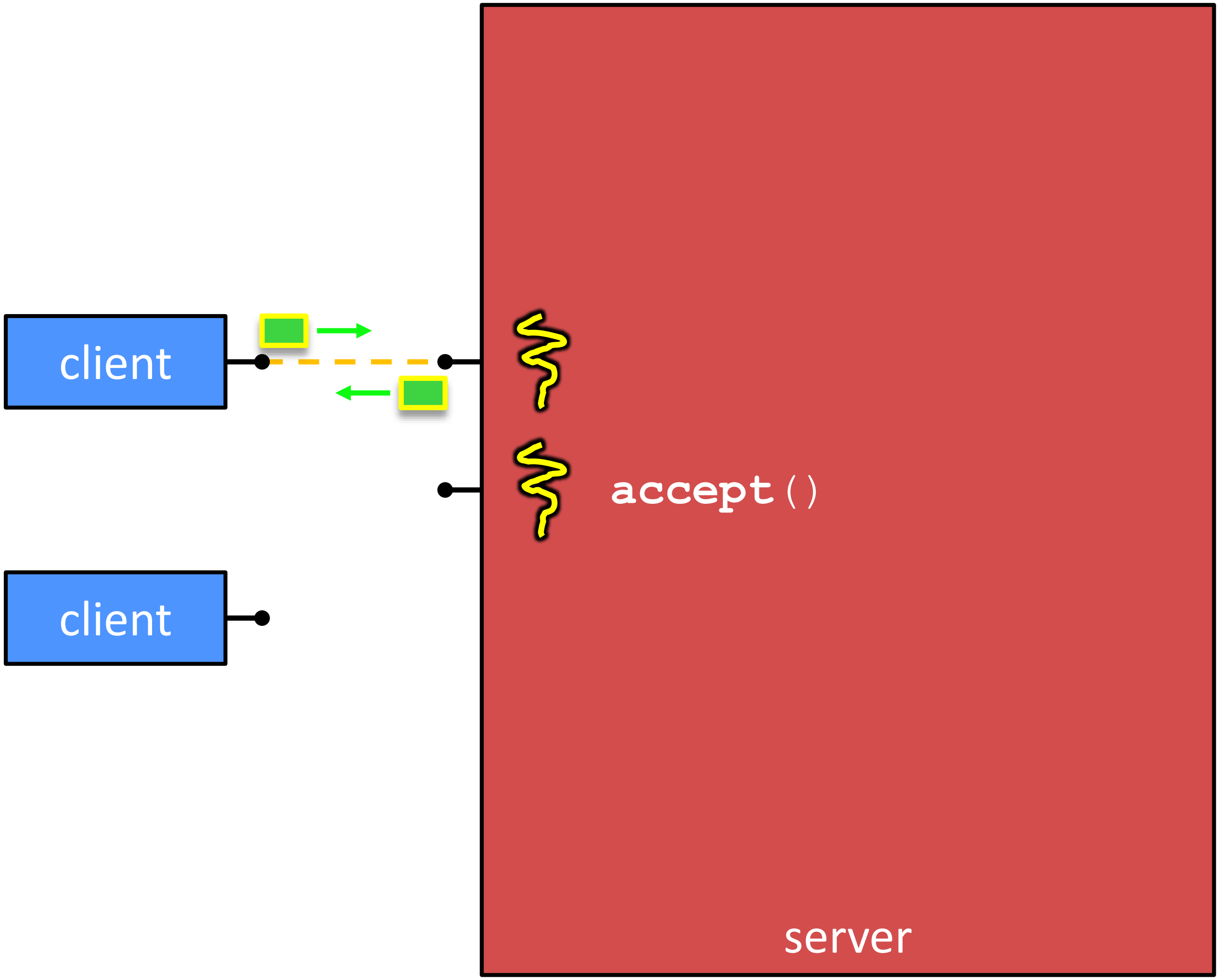
Multi-threaded Search Engine: Request Flow



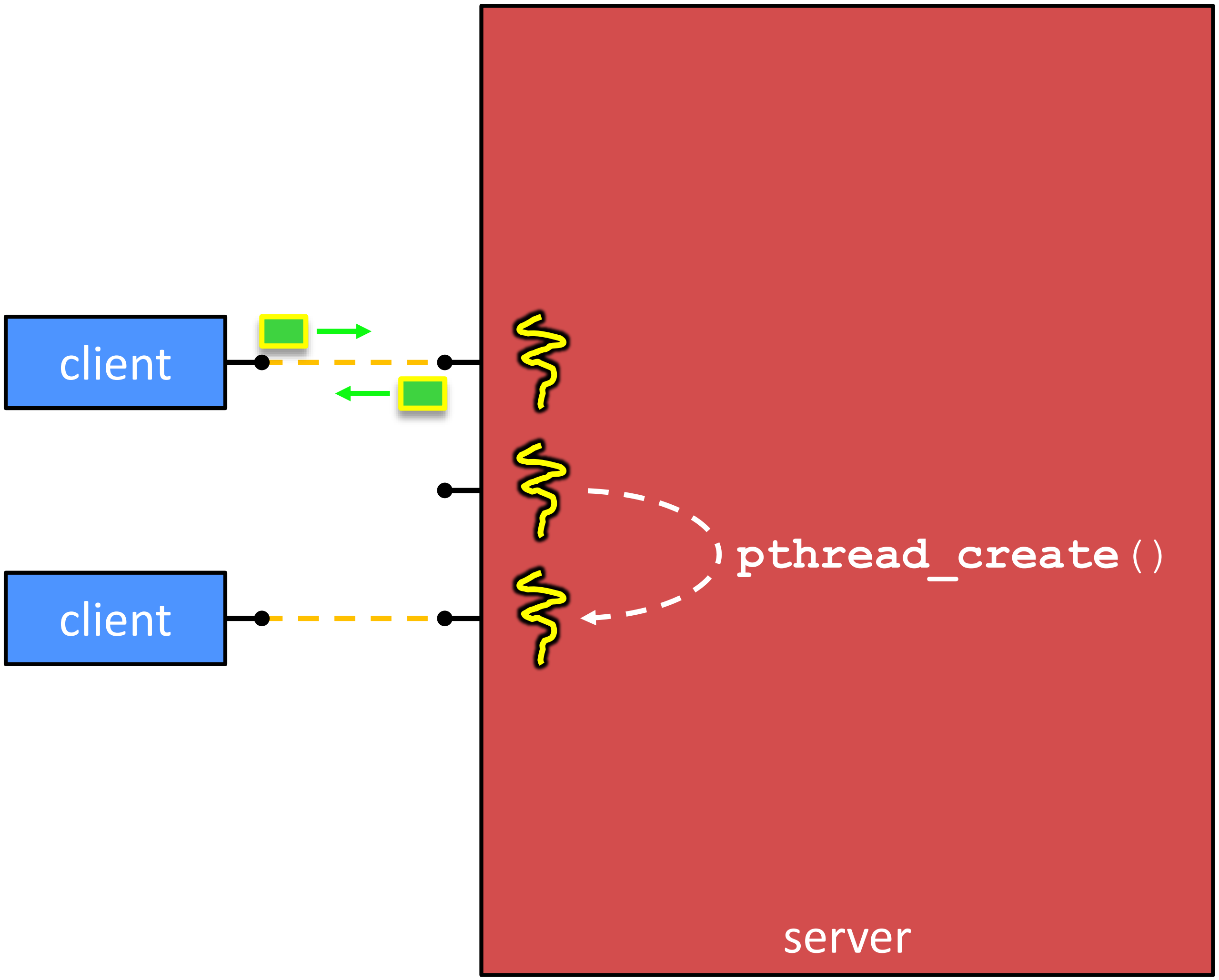
Multi-threaded Search Engine: Request Flow



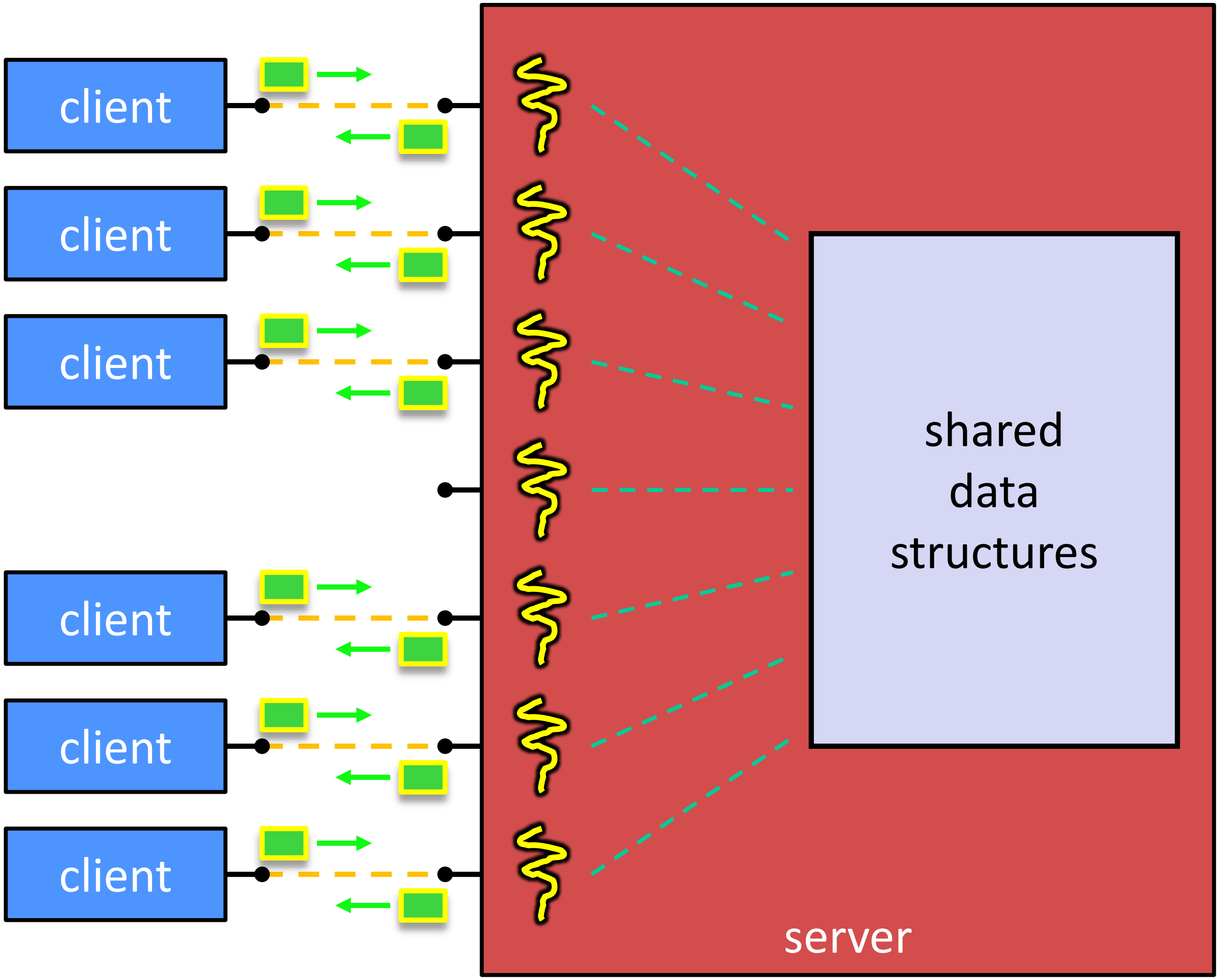
Multi-threaded Search Engine: Request Flow



Multi-threaded Search Engine: Request Flow



Multi-threaded Search Engine: Request Flow



pthread Examples

- `pthread.c:` pthreads in C
- `pthread.cc:` Same, but in C++
- `searchserver_threads:` Non-trivial example

- Things to keep in mind while reading:
  - More instructions per thread = higher likelihood of interleaving
  - How do you handle memory management?
    - Who allocates and deallocates memory?
    - Can two threads call `new` at the same time?
  - When calling **`pthread_create`**`()`, `start_routine` points to a function that takes only one argument (a `void*`)
    - To pass complex arguments into the thread, create a struct to bundle the necessary data

Why Threads? (1 of 2)

- Advantages:
  - Almost as simple to code as sequential
    - In fact, most of the code is identical! (but a bit more complicated to dispatch a thread)
  - Threads can run in parallel if you have multiple CPUs/cores
  - Concurrent execution with good CPU and network utilization
    - Some overhead, but less than processes
  - Shared-memory communication is possible

Why Threads? (2 of 2)

- Disadvantages:
  - Need language and OS support for threads
  - If threads share data, you need locks or other synchronization
    - See next lecture: Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - See next lecture: Lock contention, context switch overhead, CPU thrashing, and other issues
    - Also cognitive overhead for future programmers!
  - Threads within the same process have a “shared fate”
    - Eg, shared file-handle limits, no crash isolation, etc.

Why Sequential?

- Advantages:
  - Simple to write, maintain, debug
  - The default, supported everywhere

- Disadvantages:
  - Depending on application, poor performance
    - One slow client will cause *all* others to block
    - Poor utilization of resources (CPU, network, disk)