Threads

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevin Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Upcoming Due Dates

- HW1 (FileReaders) Due Tomorrow
  - Get started if you haven’t already!!!!
  - Should have everything you need to complete the assignment

- HW2 (Threads)
  - To be released soon HW1

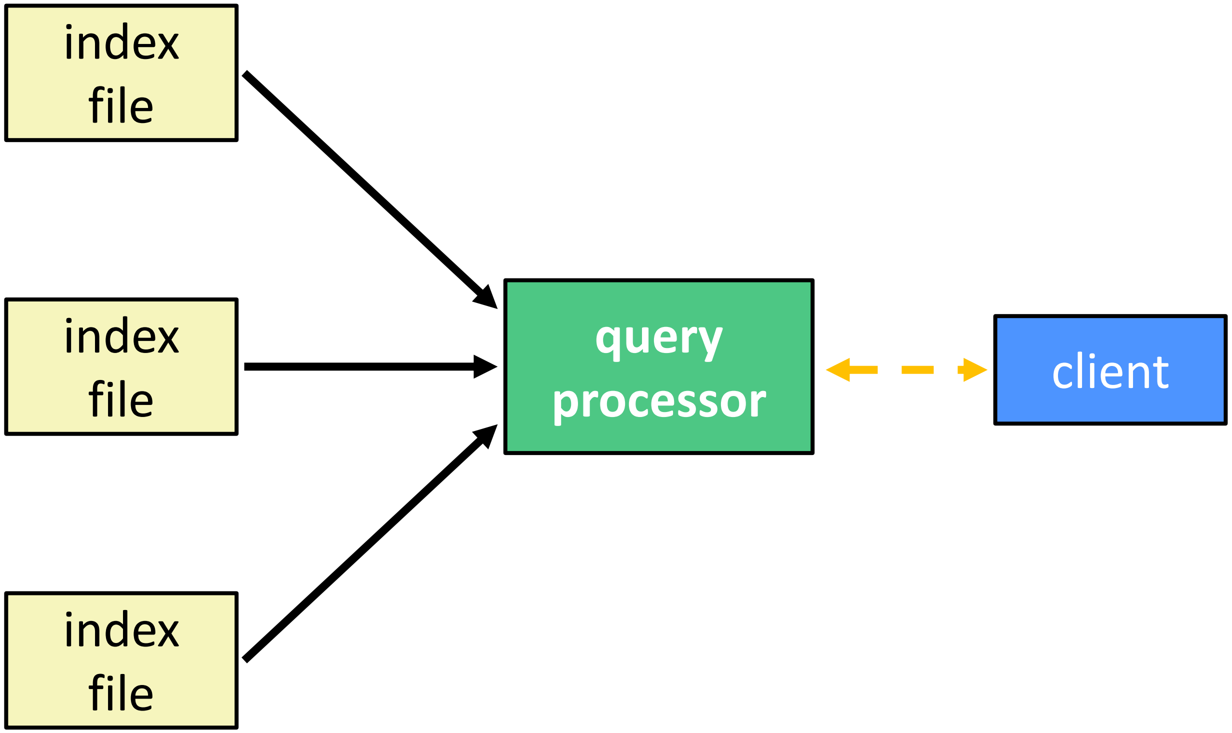
Lecture Outline

- **Why threads?**
- `pthreads` review
- Shared resources & data races
- Locks & mutexes

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

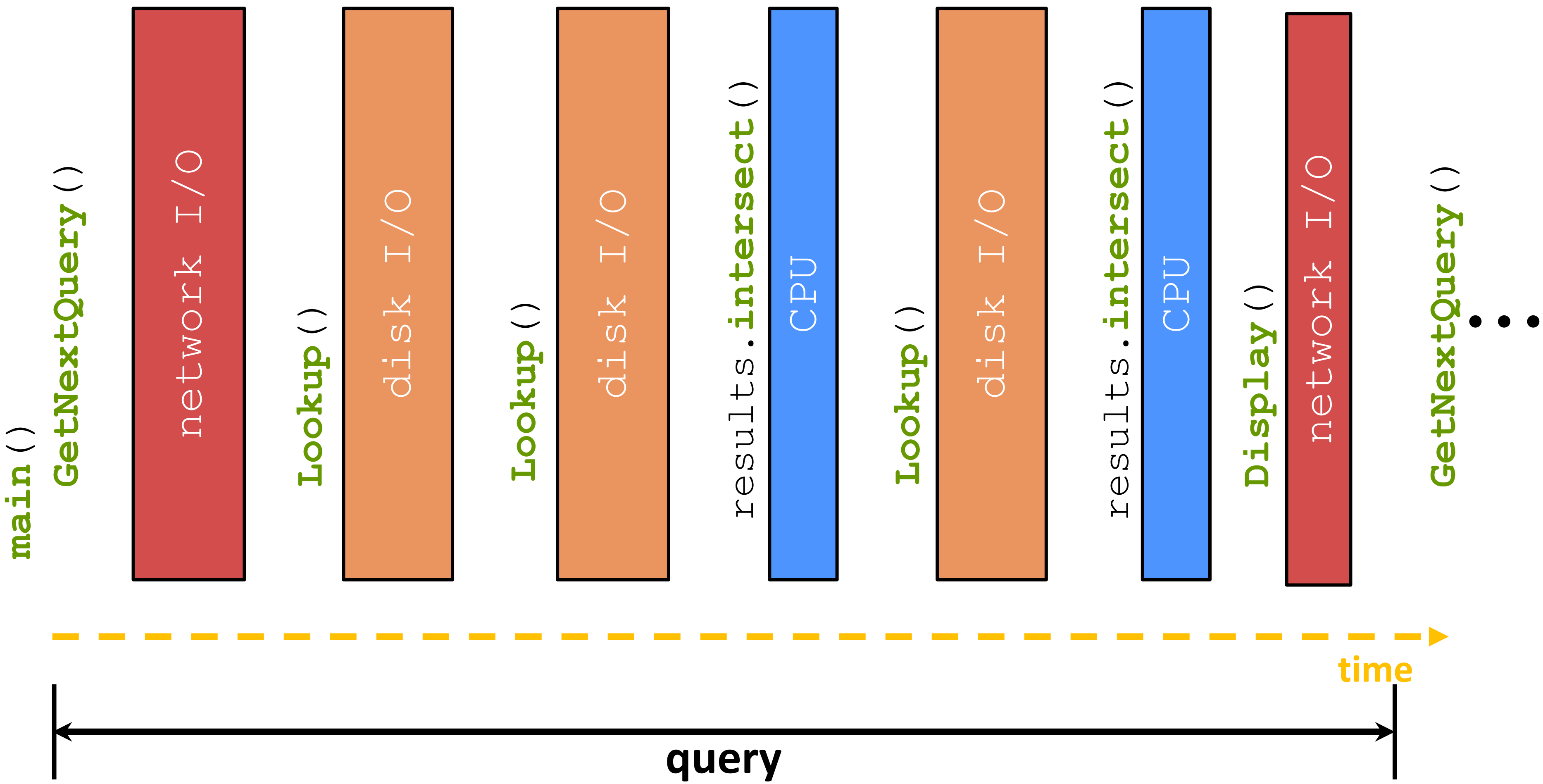
Search Engine Architecture



Search Engine (Pseudocode)

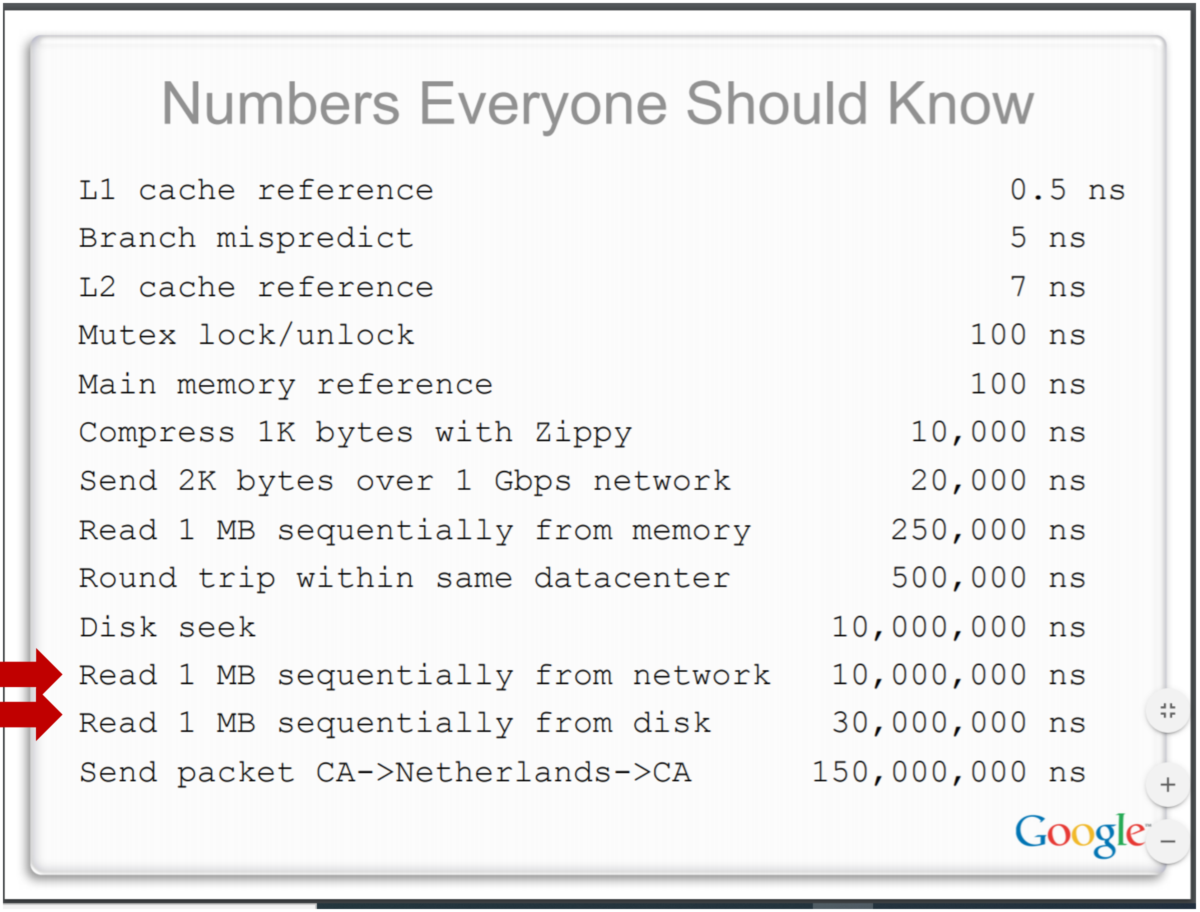
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket); ←Disk I/O
  foreach hit in hitlist {
    doclist.append(file.read(hit));
  }
  return doclist;
}

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

Execution Timeline: a Multi-Word Query



What About I/O-caused Latency?

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

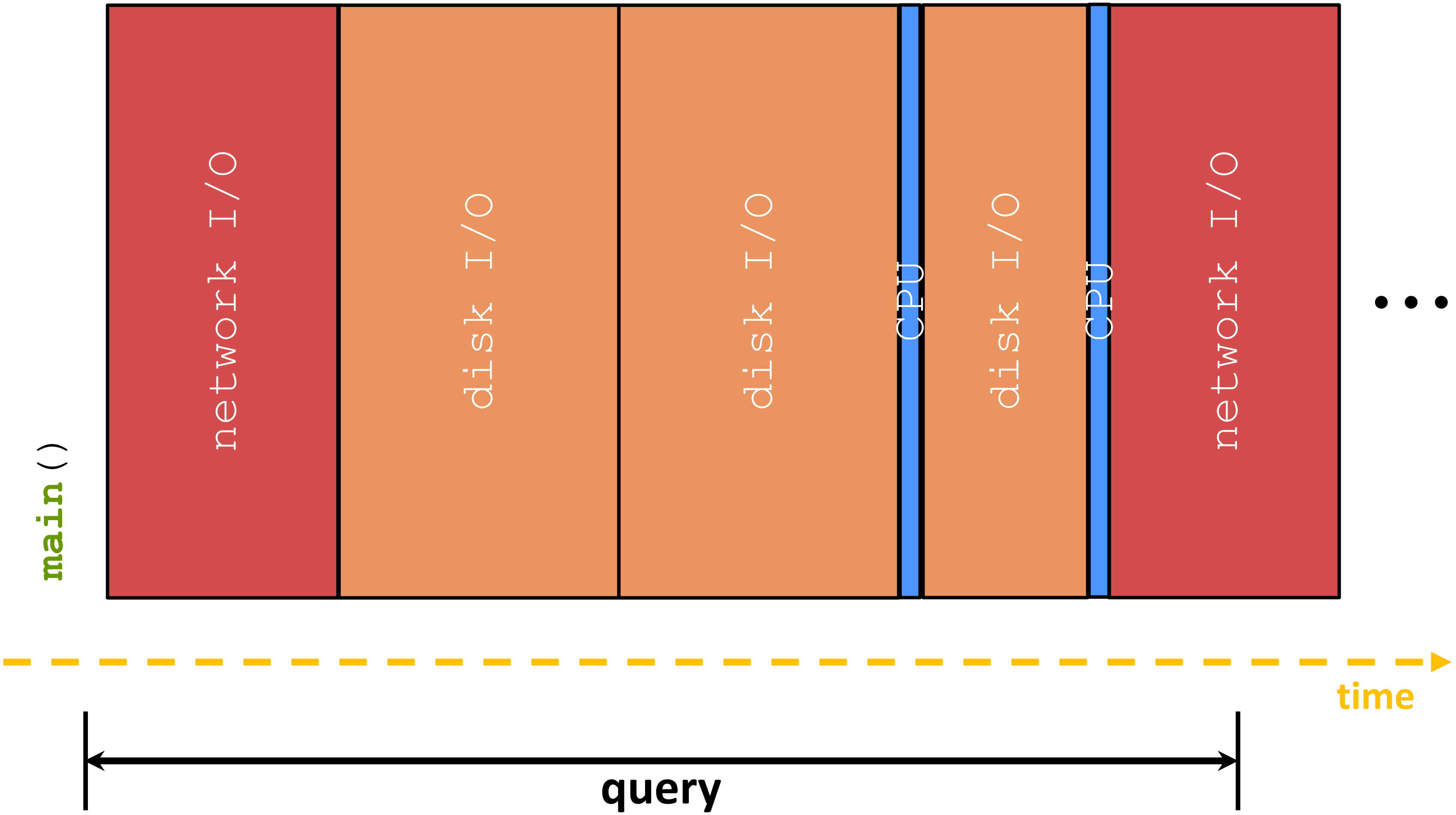


Numbers Everyone Should Know

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: To Scale

Model isn't perfect:
Technically also some cpu usage to setup I/O.
Network output also (probably) won't block program .....



Multiple (Single-Word) Queries

# is the Query Number

#.a -> `GetNextQuery()`

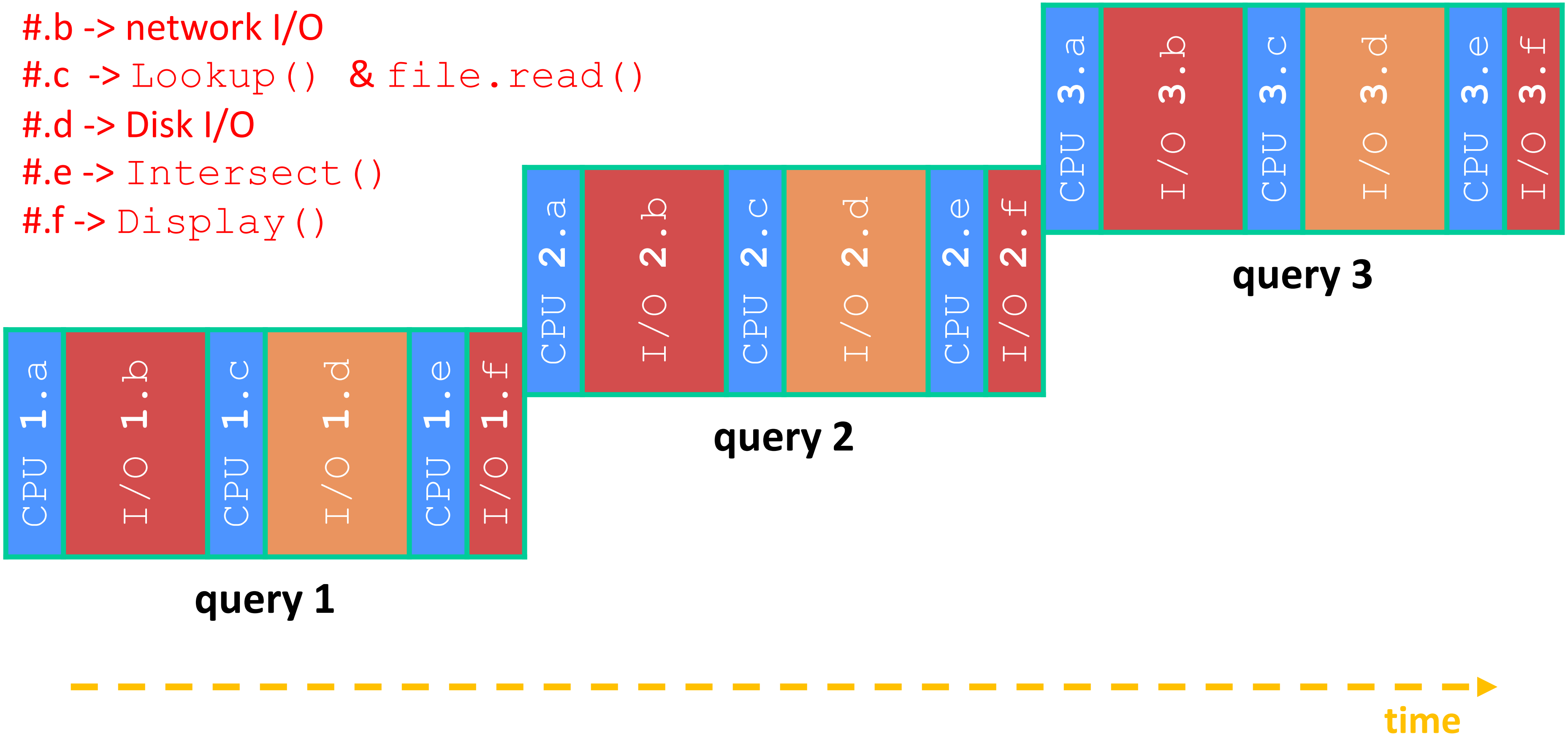
#.b -> network I/O

#.c -> `Lookup()` & `file.read()`

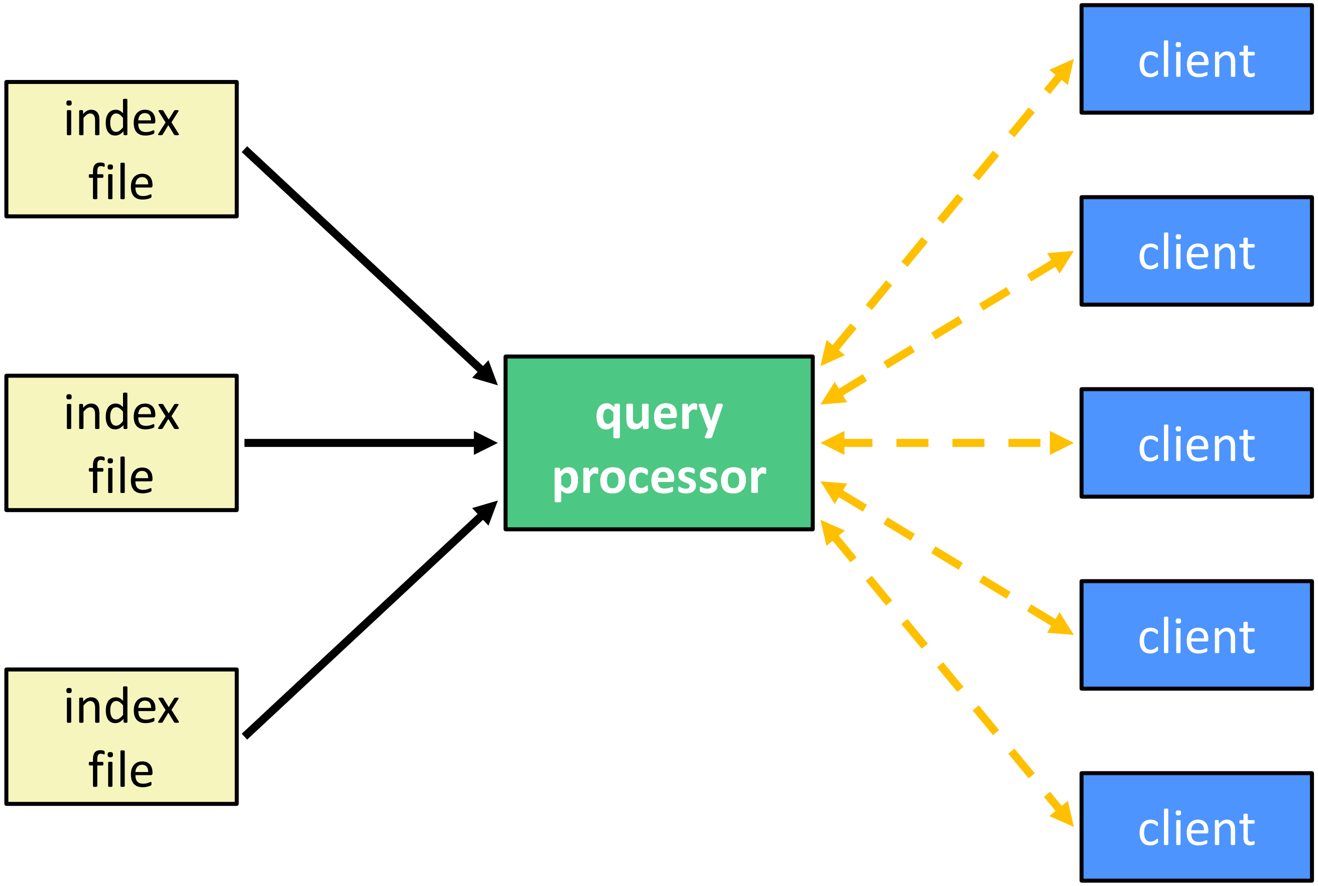
#.d -> Disk I/O

#.e -> `Intersect()`

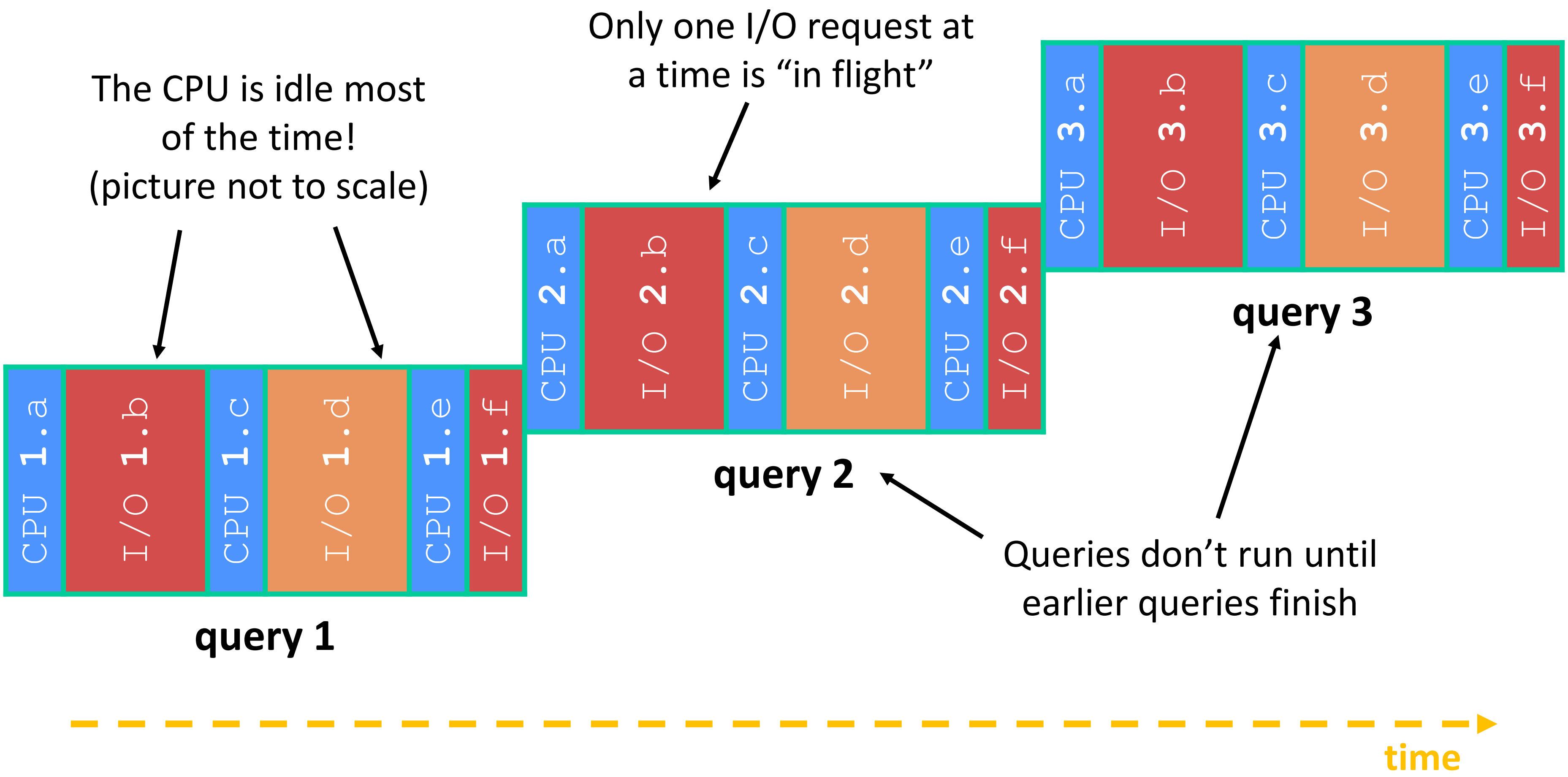
#.f -> `Display()`



Uh-Oh (1 of 2)


index file
index file
index file
query processor
client
client
client
client
client

Uh-Oh (2 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"
CPU 1.a
I/O 1.b
CPU 1.c
I/O 1.d
CPU 1.e
I/O 1.f
query 1
CPU 2.a
I/O 2.b
CPU 2.c
I/O 2.d
CPU 2.e
I/O 2.f
query 2
CPU 3.a
I/O 3.b
CPU 3.c
I/O 3.d
CPU 3.e
I/O 3.f
query 3
Queries don't run until earlier queries finish
time

Sequential Can Be Inefficient

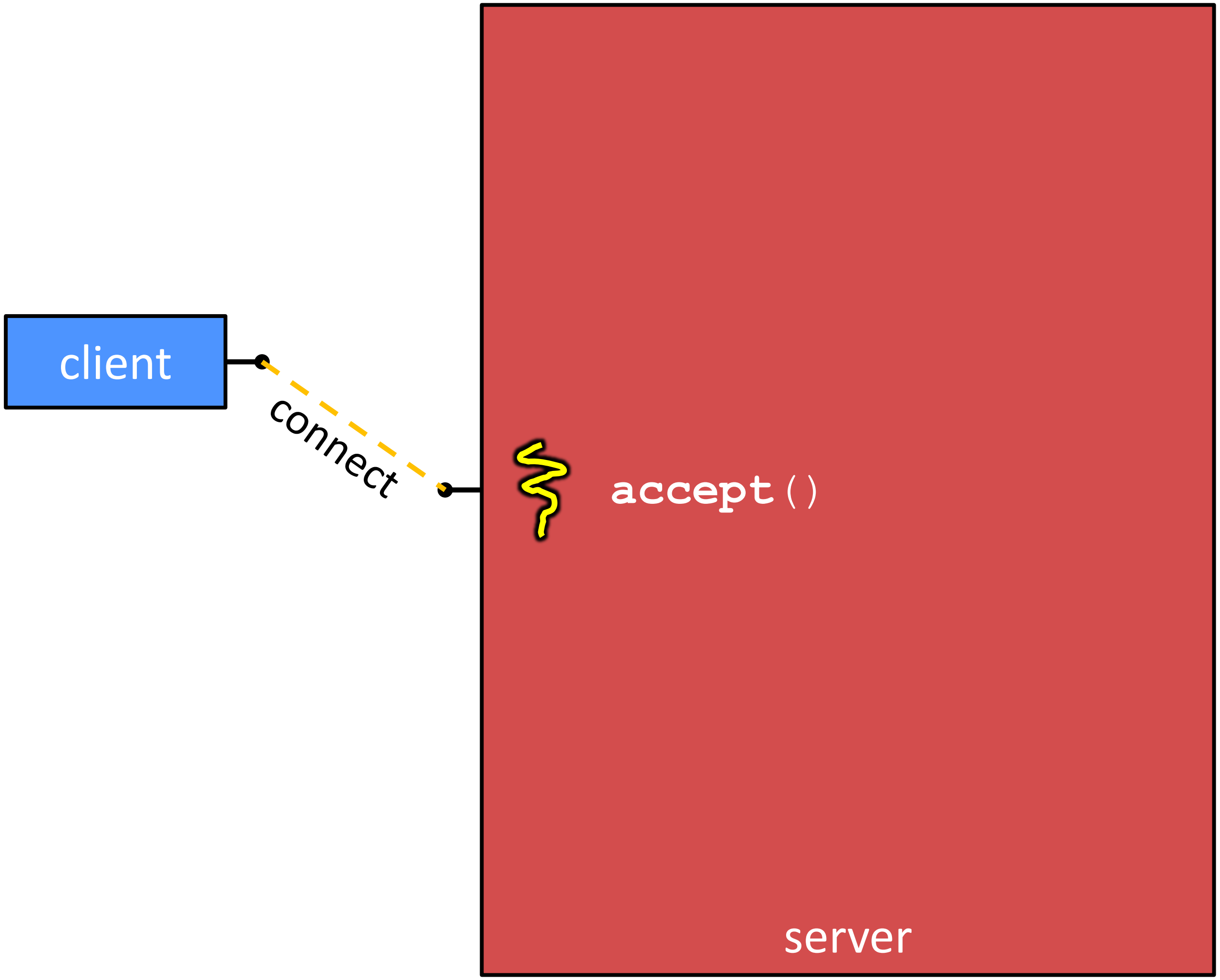
- Only one query is being processed at a time
  - All other queries queue up behind the first one
  - And clients queue up behind the queries …
- Even while processing one query, the CPU is idle the vast majority of the time
  - It is *blocked* waiting for I/O to complete
    - Disk I/O can be very, 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.

A Concurrent Implementation

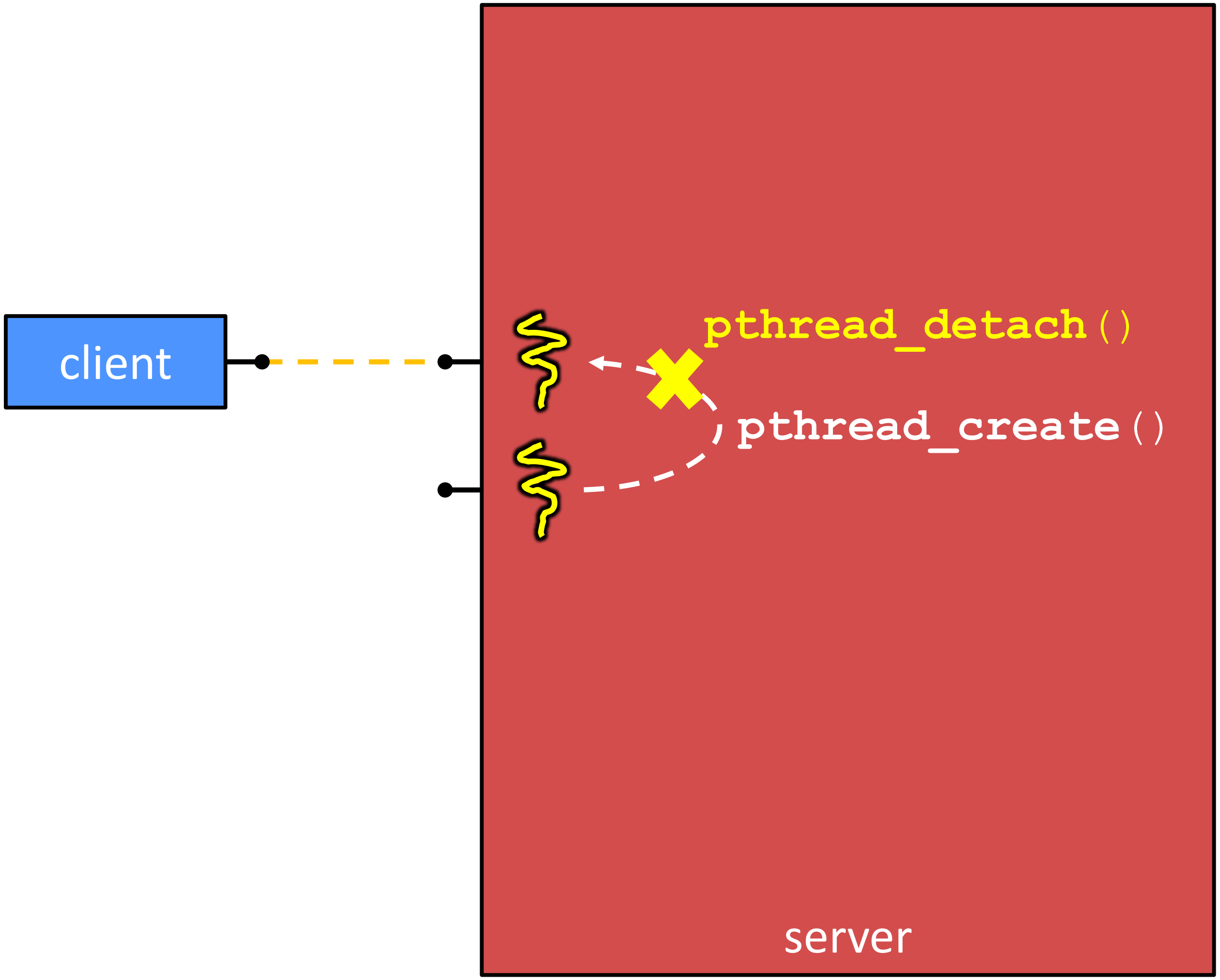
- Use multiple “workers”
  - As a query arrives, create a new “worker” to handle it
    - The “worker” reads the query from the network, issues read requests against files, assembles results and writes to the network
    - The “worker” uses blocking I/O; the “worker” alternates between consuming CPU 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”?

Threads!!!!

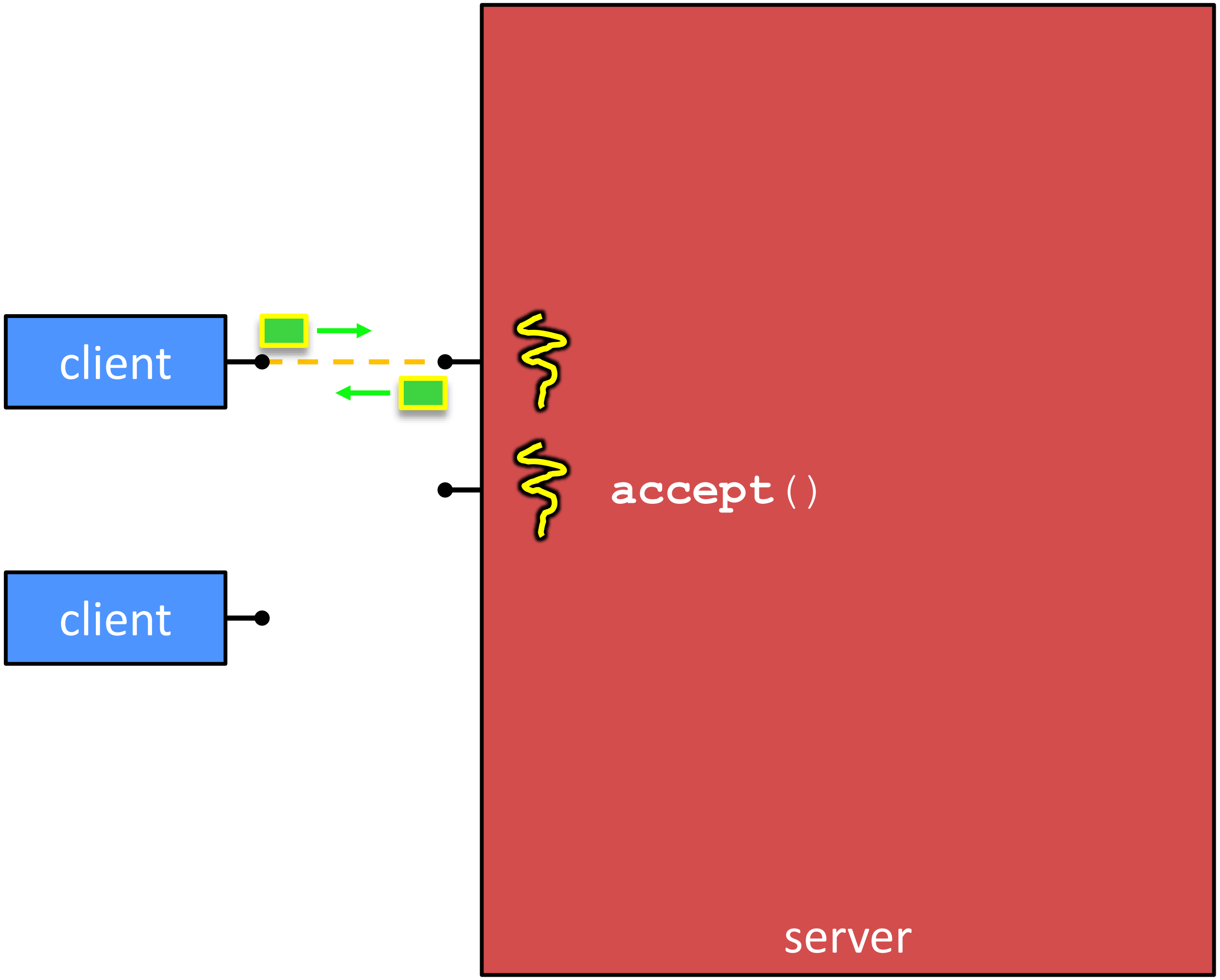
Multithreaded Server



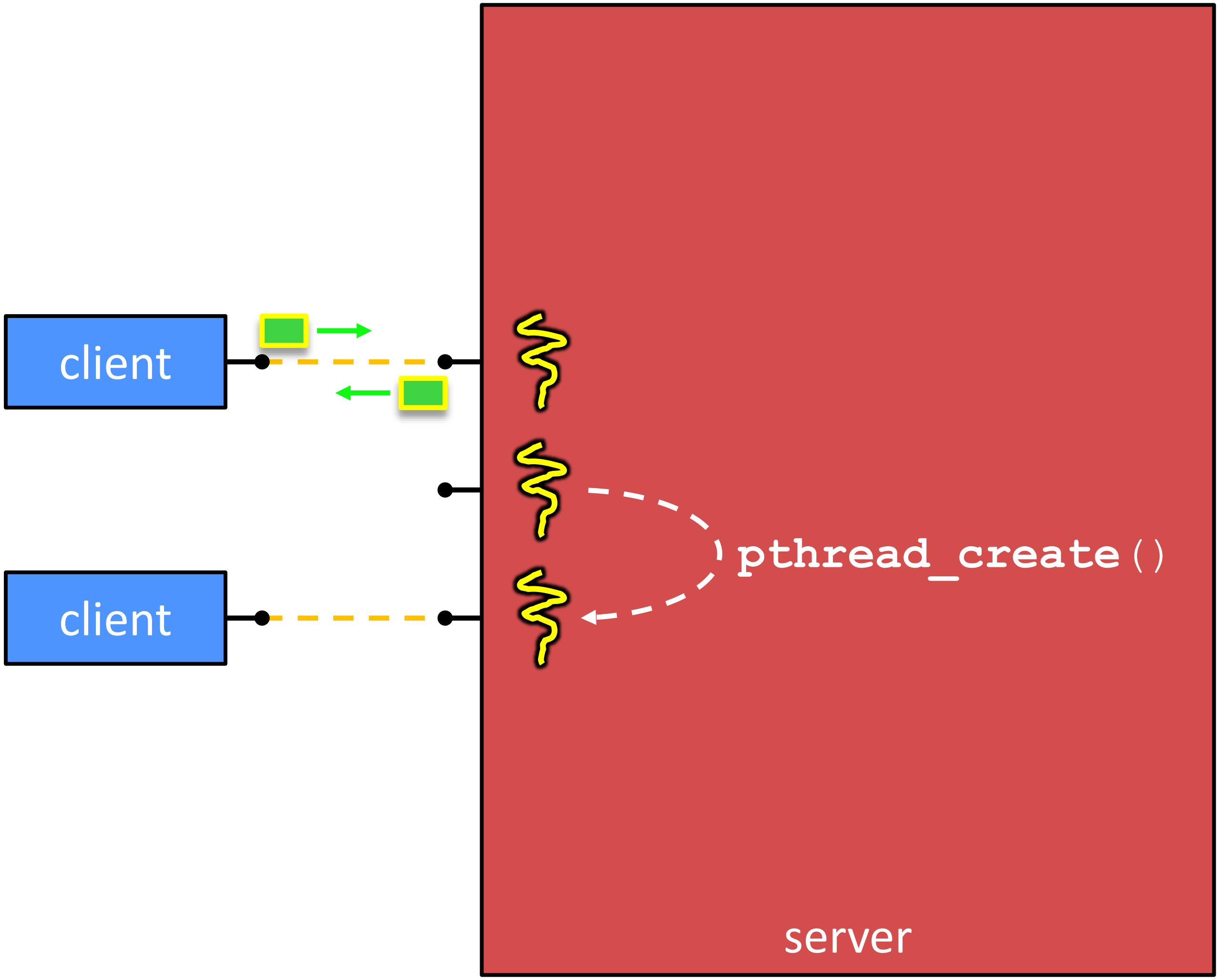
Multithreaded Server



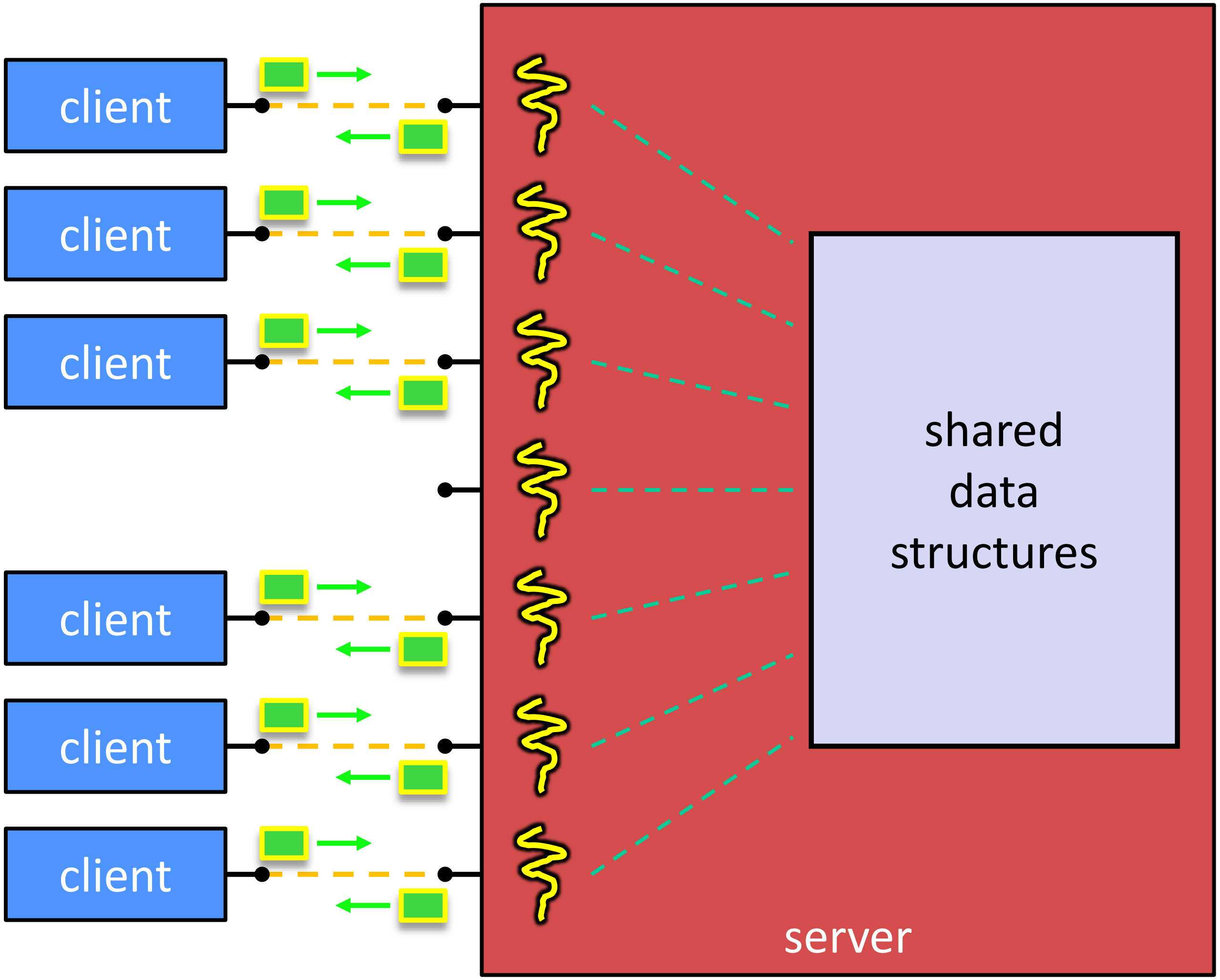
Multithreaded Server



Multithreaded Server

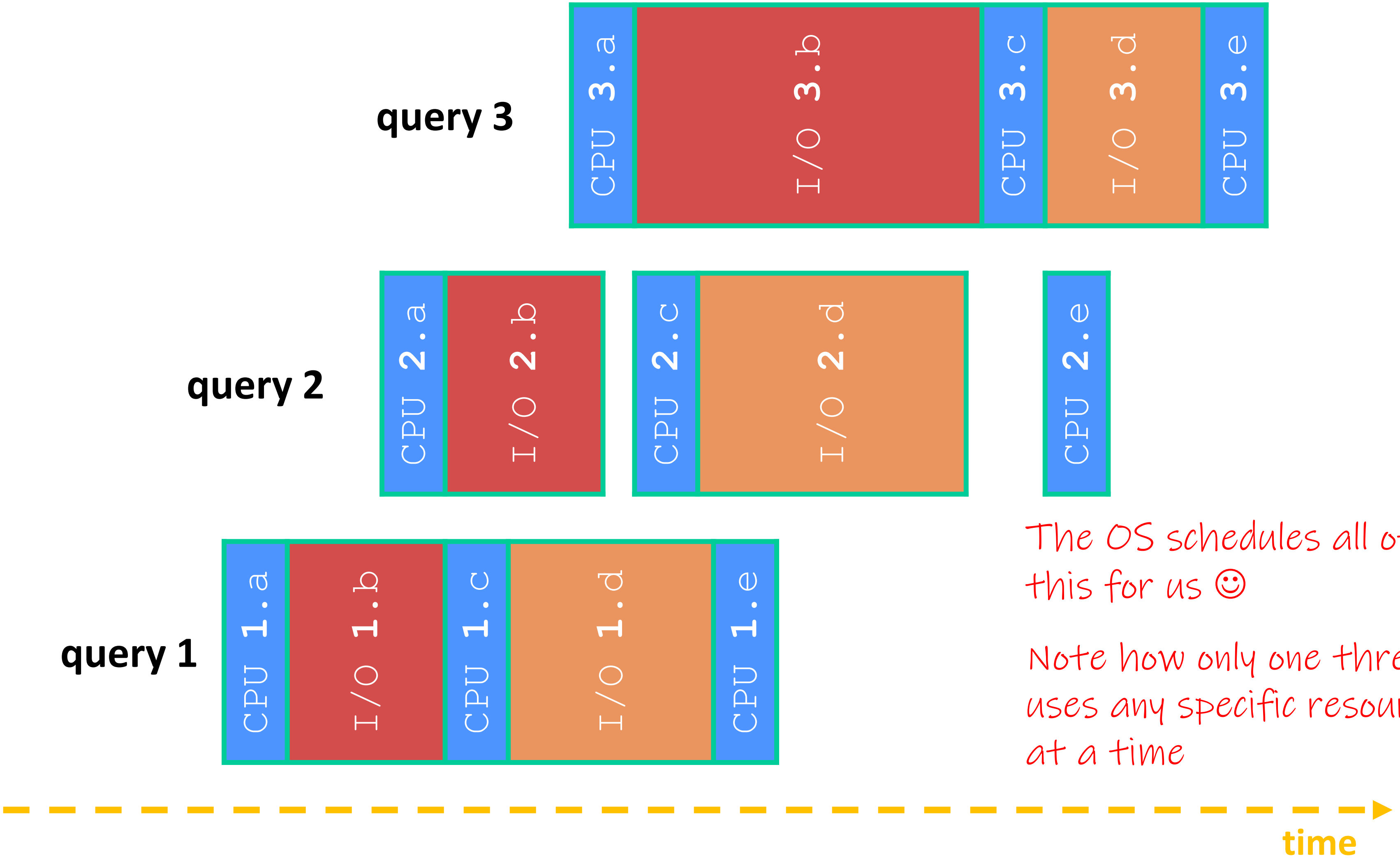


Multithreaded Server



Multi-threaded Search Engine (Execution)

*Running with 1 CPU



The OS schedules all of this for us ☺

Note how only one thread uses any specific resource at a time

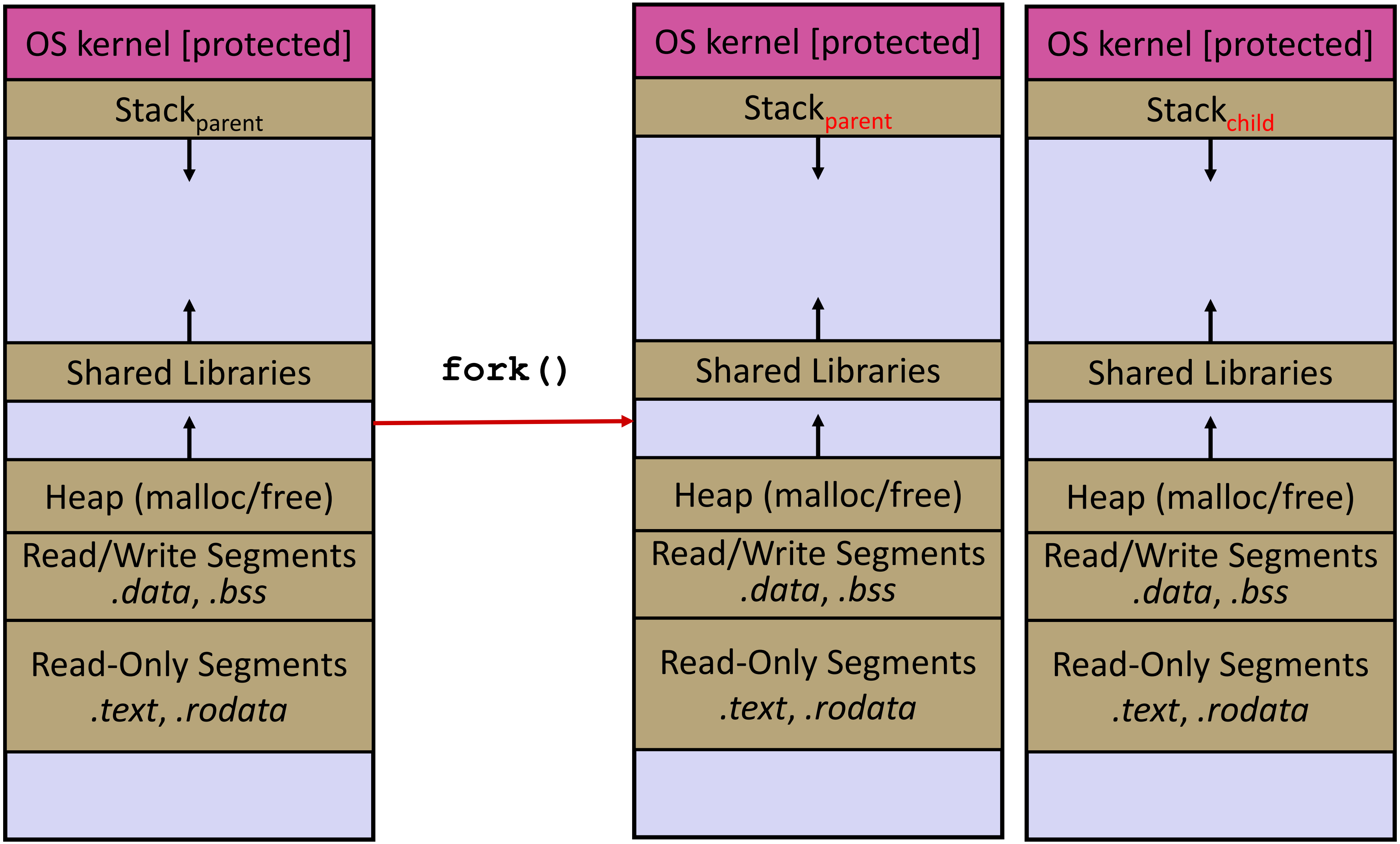
Why Threads?

- 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

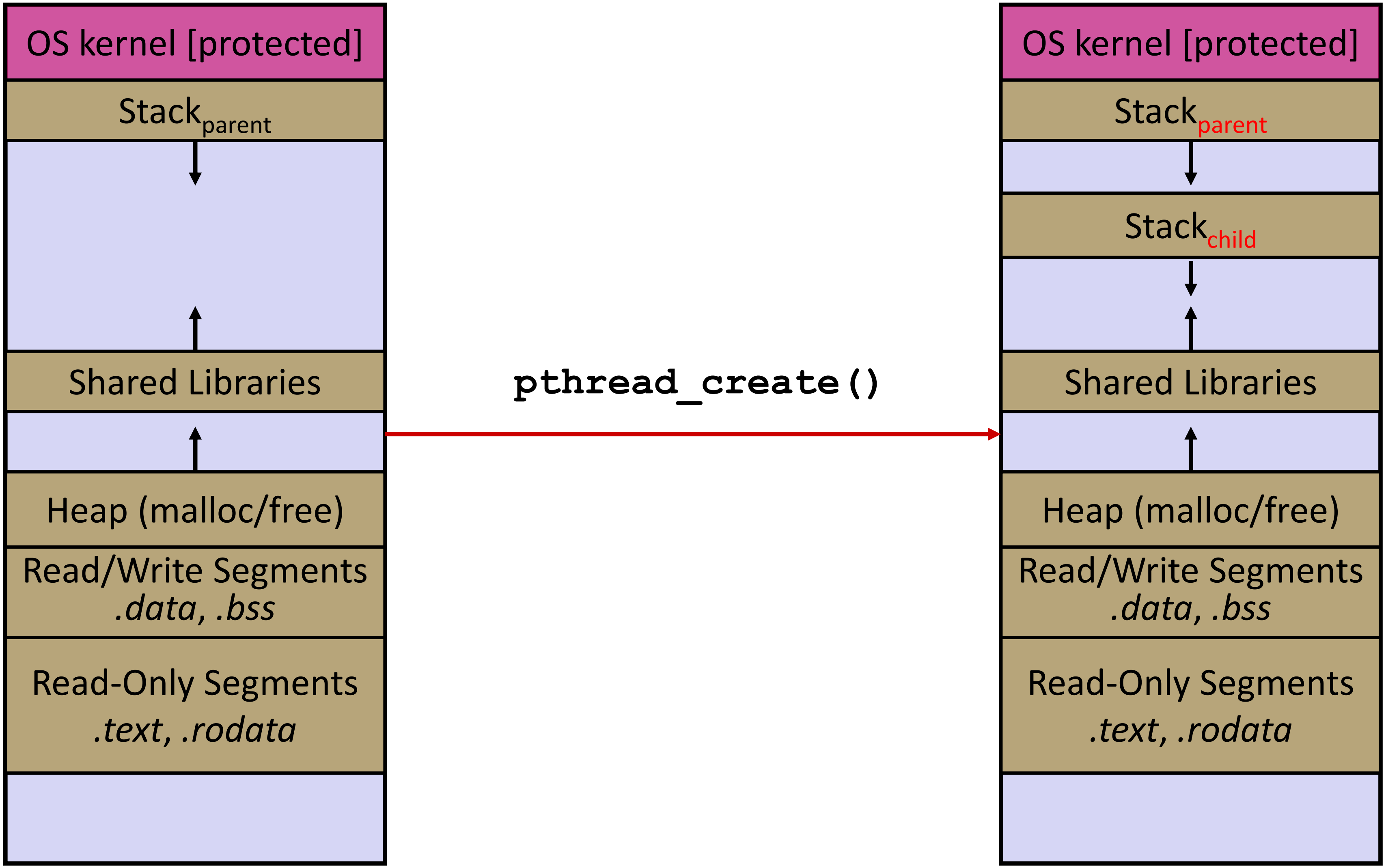
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources,
    & security attributes
  - A Thread has a unique: 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 vs. Processes



Threads vs. Processes



Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides "fork"
  - Processes are isolated. If one crashes, other processes keep going

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

Poll Everywhere

pollev.com/tqm

- If I wanted to make a web browser, what concurrency model should I use?
  - Note that a web browser may need to request many resources over the network and combine them together to load a page

A. **Do it sequentially**

B. **Use threads**

C. **Use processes**

D. **We're lost…**

Poll Everywhere

pollev.com/tqm

- If I wanted to make a web browser, what concurrency model should I use?
  - Note that a web browser may need to request many resources over the network and combine them together to load a page

A. Do it sequentially

[✓ B.] Use threads

C. Use processes

D. We're lost…

Concurrency will make more efficient use of time

We will need to share the data we request across "workers"

We want to be fast

Lecture Outline

- Why threads?
- **`pthreads` review**
- Shared resources & data races
- Locks & mutexes

Creating and Terminating Threads

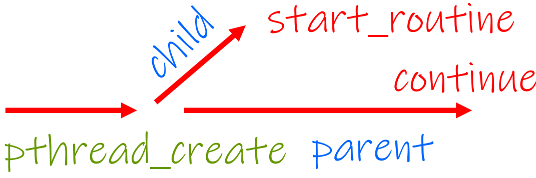
Output parameter.
Gives us a "thread_descriptor"

```
int pthread_create(
          pthread_t* thread,
          const pthread_attr_t* attr,
          void* (*start_routine)(void*),
          void* arg);
```

Function pointer!
Takes & returns void*
to allow "generics" in C

Argument for the thread function

- 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)`



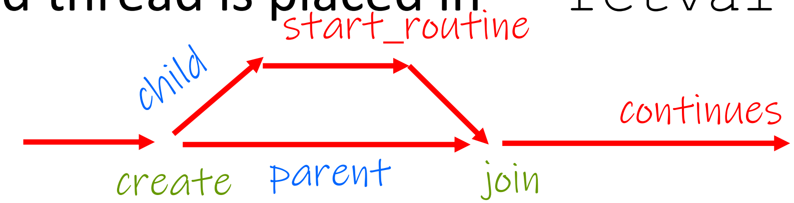
```
void pthread_exit(void* retval);
```

- 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?

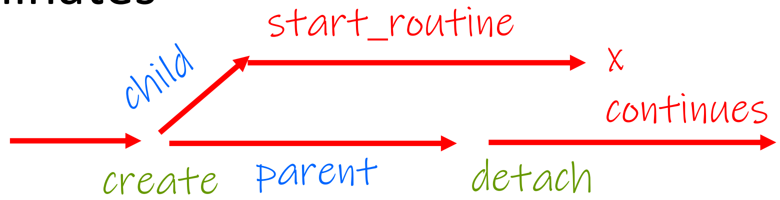
- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of **`waitpid`**`()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



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

Detach a thread.
Thread is cleaned up when it is finished



Thread Examples

- See `cthreads.c`
  - How do you properly handle memory management?
    - Who allocates and deallocates memory?
    - How long do you want memory to stick around?
- See `exit_thread.cc`
  - Do we need to join every thread we create?
- See `ccthreads.cc`
  - Rewriting cthreads.c, but in C++

Lecture Outline

- Why threads?
- `pthreads` review
- **Shared resources & data races**
- Locks & mutexes

Shared Resources

- Some resources are shared between threads and processes

- Thread Level:
  - Memory
  - Things shared by processes

- Process level
  - I/O devices
    - Files
    - terminal input/output
    - The network

Issues arise when we try to shared things

Data Races

- Two memory accesses form a data race if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance (which thread ran first?)

Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {

  buy milk

}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We’re lost…

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

We can be interrupted between checking note and leaving note ☹

```
if (!note) {
  if (!milk) {
    leave note
    buy milk
    remove note
  }
}
```

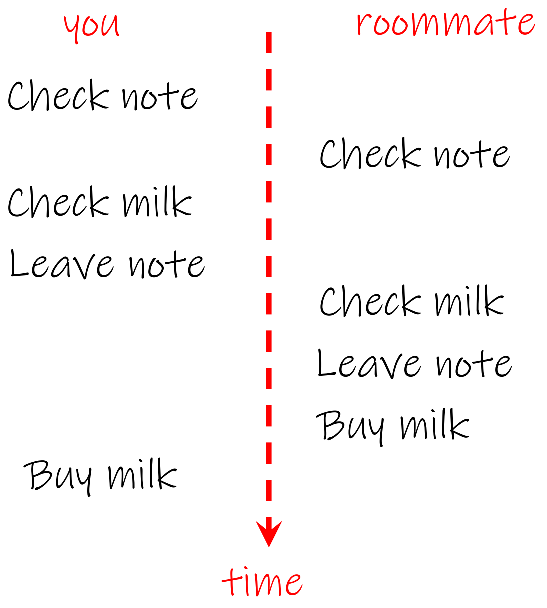
A. Yes, problem fixed

B. No, could end up with no milk

[✓ C.] No, could still buy multiple milk

D. We're lost…

*There are other possible scenarios that result in multiple milks



you	roommate
Check note	
	Check note
Check milk	
Leave note	
	Check milk
	Leave note
	Buy milk
Buy milk	

Threads and Data Races

- Data races might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- Example: two threads try to read from and write to the same shared memory location
  - Could get "correct" answer
  - Could accidentally read old value
  - One thread's work could get "lost"
- Example: two threads try to push an item onto the head of the linked list at the same time
  - Could get "correct" answer
  - Could get different ordering of items
  - Could break the data structure! ☠

Lecture Outline

- Why threads?
- `pthreads` review
- Shared resources & data races
- **Locks & mutexes**

Synchronization

- Synchronization is the act of preventing two (or more) concurrently running threads from interfering with each other when operating on shared data
  - Need some mechanism to coordinate the threads
    - “Let me go first, then you can go”
  - Many different coordination mechanisms have been invented

- Goals of synchronization:
  - Liveness – ability to execute in a timely manner (informally, “something good eventually happens”)
  - Safety – avoid unintended interactions with shared data structures (informally, “nothing bad happens”)

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time
  - Executed in an uninterruptible (*i.e.* atomic) manner

- Lock Acquire
  - Wait until the lock is free, then take it

- Lock Release
  - Release the lock
  - If other threads are waiting, wake exactly one up to pass lock to

- Pseudocode:

```
// non-critical code

lock.acquire();
// critical section
lock.release();

// non-critical code
```

loop/idle if locked

Milk Example – What is the Critical Section?

- What if we use a lock on the refrigerator?
  - Probably overkill – what if roommate wanted to get eggs?
- For performance reasons, only put what is necessary in the critical section
  - Only lock the milk
  - But lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit)

```
fridge.lock()
if (!milk) {
  buy milk
}
fridge.unlock()
```

```
milk_lock.lock()
if (!milk) {
  buy milk
}
milk_lock.unlock()
```

pthreads and Locks

- Another term for a lock is a mutex (“mutual exclusion”)
  - `pthread.h` defines datatype `pthread_mutex_t`

-
  ```
  int pthread_mutex_init(pthread_mutex_t* mutex,
                         const pthread_mutexattr_t* attr);
  ```
  - Initializes a mutex with specified attributes

-
  ```
  int pthread_mutex_lock(pthread_mutex_t* mutex);
  ```
  - Acquire the lock – blocks if already locked Un-blocks when lock is acquired

-
  ```
  int pthread_mutex_unlock(pthread_mutex_t* mutex);
  ```
  - Releases the lock

-
  ```
  int pthread_mutex_destroy(pthread_mutex_t* mutex);
  ```
  - “Uninitializes” a mutex – clean up when done

pthread Mutex Examples

- See `total.cc`
  - Data race between threads
- See `total_locking.cc`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.cc`