Parallelism & Overhead Cost

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Administrivia

- HW06 posted before break
  - Due by end of day Friday due to fall break
- Midterm Details
  - In-class on Wed Oct 22nd
  - Posted
  - Review in lecture on Monday and recitation tomorrow
- HW07:
  - posted by end of week.
  - We don’t expect you to work on it till after midterm, but you can start whenever. Shouldn’t be too long (hopefully). if you want to get started, you can.

Lecture Outline

- **Ahmdal's Law**
- Parallelism vs Concurrency
- Sequential Consistency
- Overhead Cost

Amdahl's Law

- For most algorithms, there are parts that parallelize well and parts that don't. This causes adding threads to have diminishing returns
  - (even ignoring the overhead costs of creating & scheduling threads)

- Consider we have some parallel algorithm $T_1 = 1$
  - The 1 subscript indicates this is run on 1 thread
  - we define the work for the entire algorithm as 1

- We define S as being the part that can be parallelized
  - $T_1 = S + (1 - S)$ *// (1-S) is the sequential part*

Amdahl's Law

- For running on one thread:
  - $T_1 = (1 - S) + S$

- If we have P threads and perfect linear speedup on the parallelizable part, we get
  - $T_P = (1\text{-}S) + \frac{S}{P}$

- Speed up multiplier for P threads from sequential is:
  - $\frac{T_1}{T_p} = \frac{1}{1-S+\frac{S}{P}}$

Amdahl's Law

- Let's say that we have 100000 threads (P = 100000) and our algorithm is only 2/3 parallel? (s = 0.6666..)
  - $\frac{T_1}{T_p} = \frac{1}{1-0.6666+\frac{0.6666}{100000}} = 2.9999\ times\ faster\ than\ sequential$
- What if it is 90% parallel? (S = 0.9):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.9+\frac{0.9}{100000}} = 9.99\ times\ faster\ than\ sequential$
- What if it is 99% parallel? (S = 0.99):
  - $\frac{T_1}{T_p} = \frac{1}{1-0.99+\frac{0.99}{100000}} = 99.99\ times\ faster\ than\ sequential$

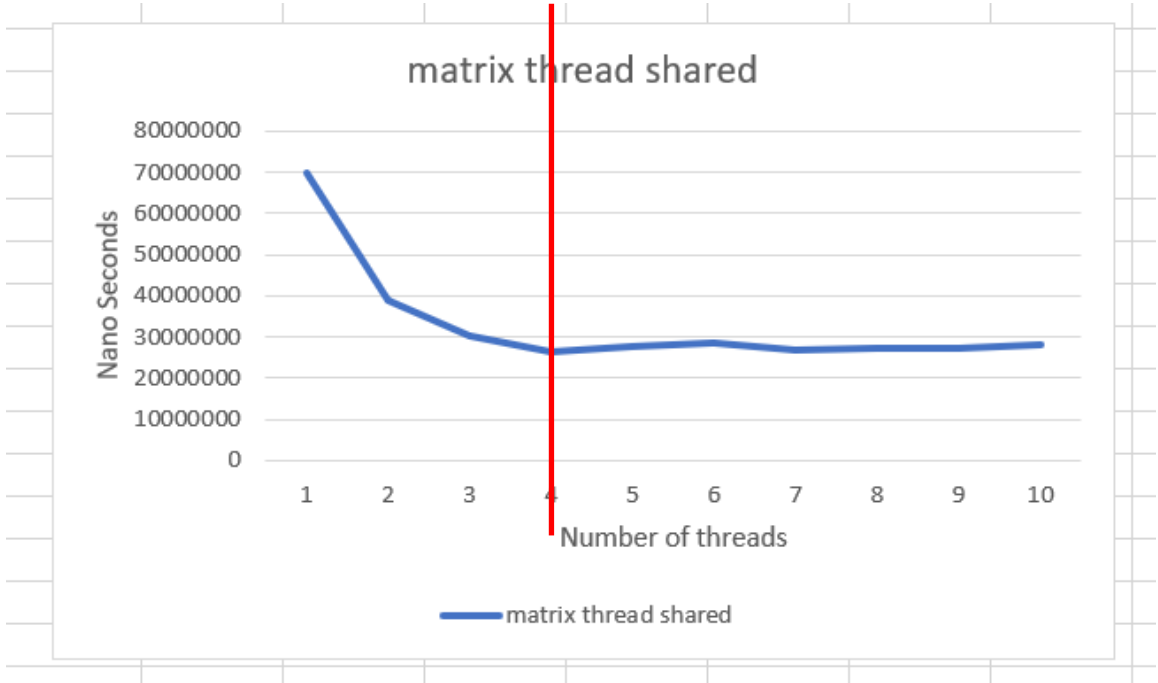
Parallelism

- You can gain performance by running things in parallel
  - Each thread can use another core
  - This makes it so that we can execute multiple threads at the same instant in time

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix

Parallelism

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix
- I can speed this up by giving each thread a part of the matrix to check!
  - **Works with threads since they share memory, harder to do with processes**



Diminishing returns

After 4 threads, no gain in speed

why? Machine run on only has 4 cores

Limitation: Hardware Threads

- These algorithms are limited by hardware.
- Number of Hardware Threads: The number of threads can genuinely run in parallel on hardware

- We may be able to create a huge number of threads, but only run a few (e.g. 4) in parallel at a time.

- Can see this information in with `lscpu` in bash
  - A computer can have some number of CPU sockets
  - Each CPU can have one or more cores
  - Each Core can run 1 or more threads

Limitations: Other Hardware

- This algorithm analysis assumes we are spending time purely in the CPU
- It doesn't account for threads blocking on I/O or other hardware.

- More on this in a second

Lecture Outline

- Ahmdal's Law
- **Parallelism vs Concurrency**
- Sequential Consistency
- Overhead Cost

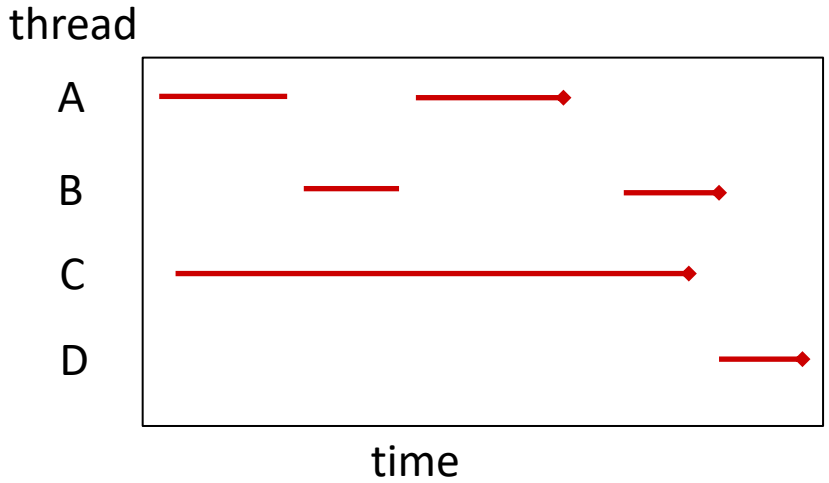
Parallelism vs Concurrency

- Two commonly used terms (often mistakenly used interchangeably).

- Concurrency: When there are one or more "tasks" that have overlapping lifetimes (between starting, running and terminating).
  - That these tasks are both running within the same **period**.

- Parallelism: when one or more "tasks" run at the same **instant** in time.

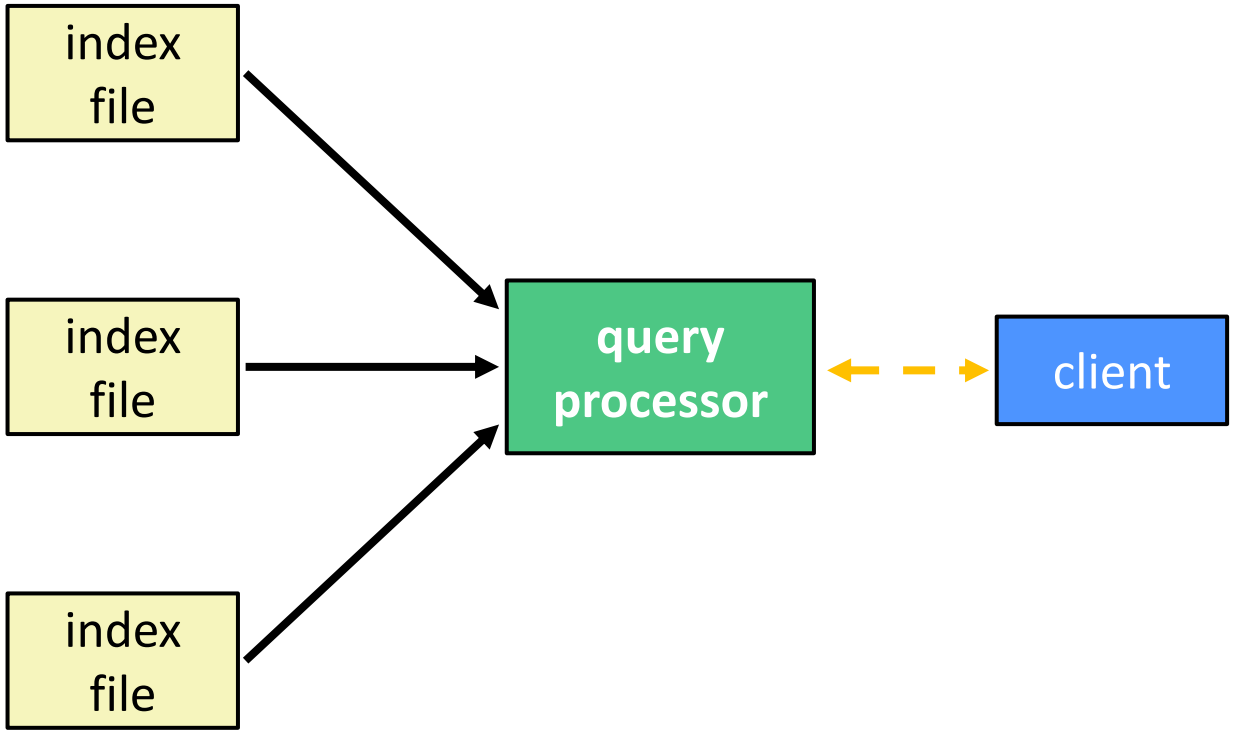
- Consider the lifetime of these threads. Which are concurrent with A? Which are parallel with A?



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



Discuss & Raise Hands

- This is pseudo code for what our multi threaded server does.
- When do you think our code reads from the network?
- When does it read from a file?
- Query size = 2
  each query “hits” once

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

Discuss & Raise Hands

- This is pseudo code for what our multi threaded server does.
- When do you think our code reads from the network?
- When does it read from a file?

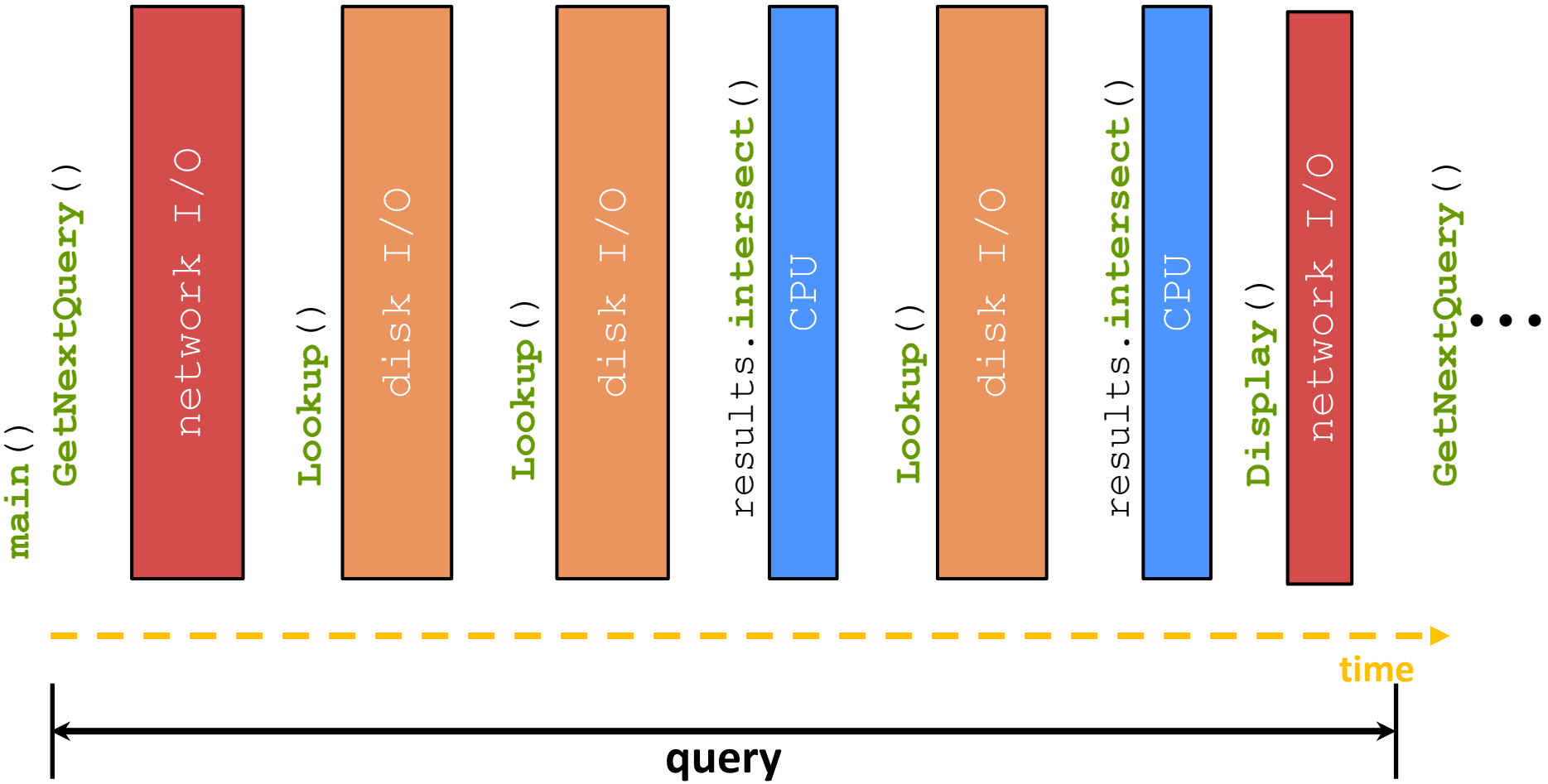
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);
  foreach hit in hitlist {
    doclist.append(file.read(hit));
  }
  return doclist;
}

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

Disk I/O

Network I/O

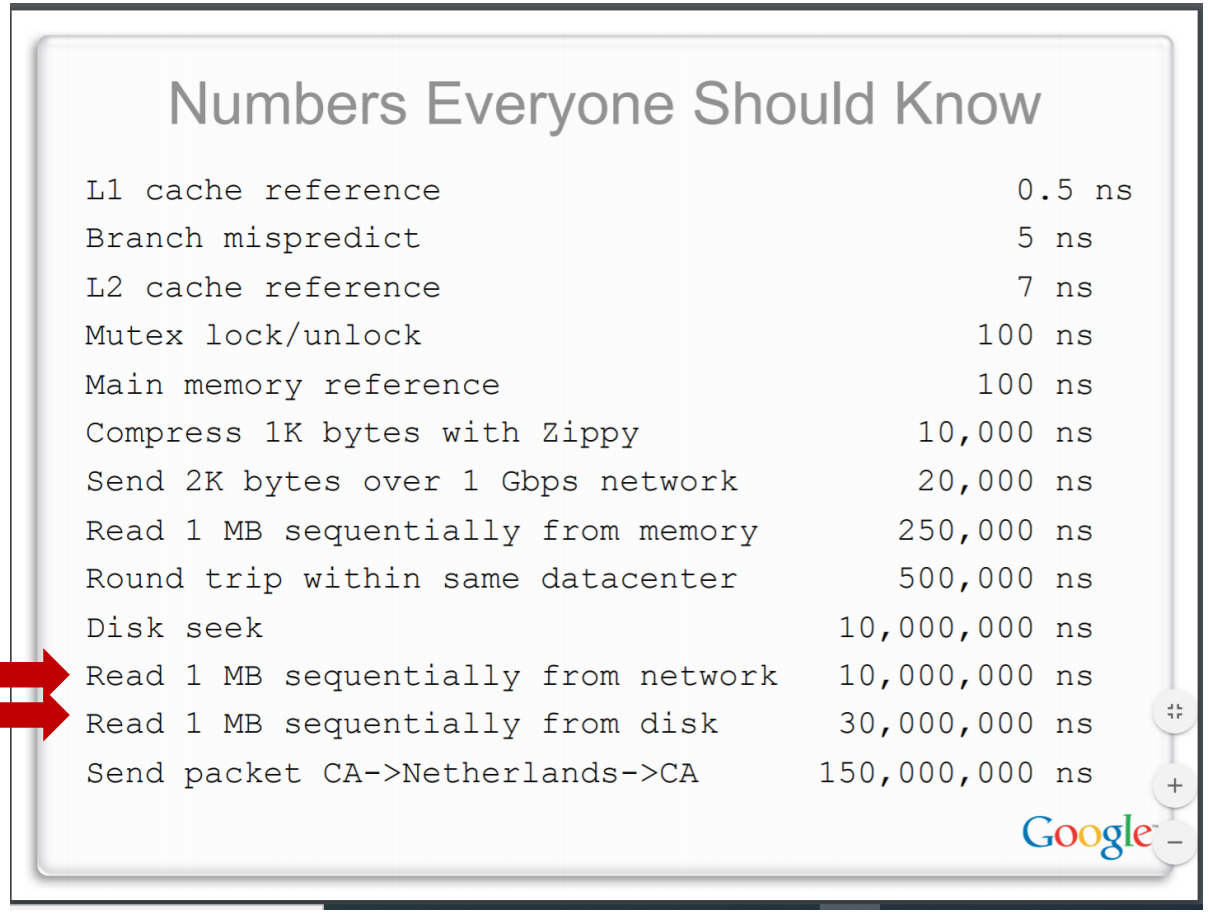
Network I/O

Execution Timeline: a Multi-Word Query



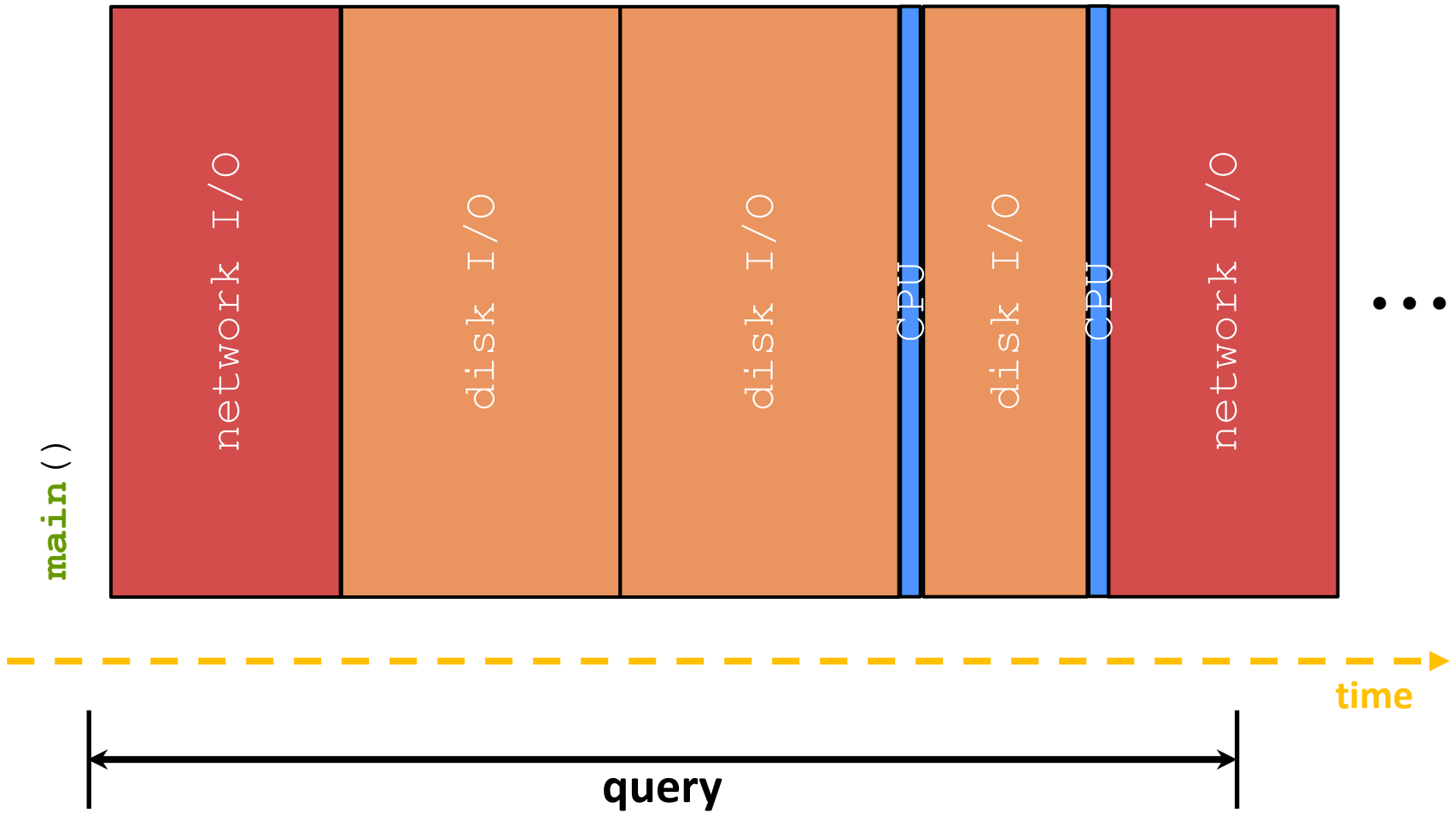
What About I/O-caused Latency?

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

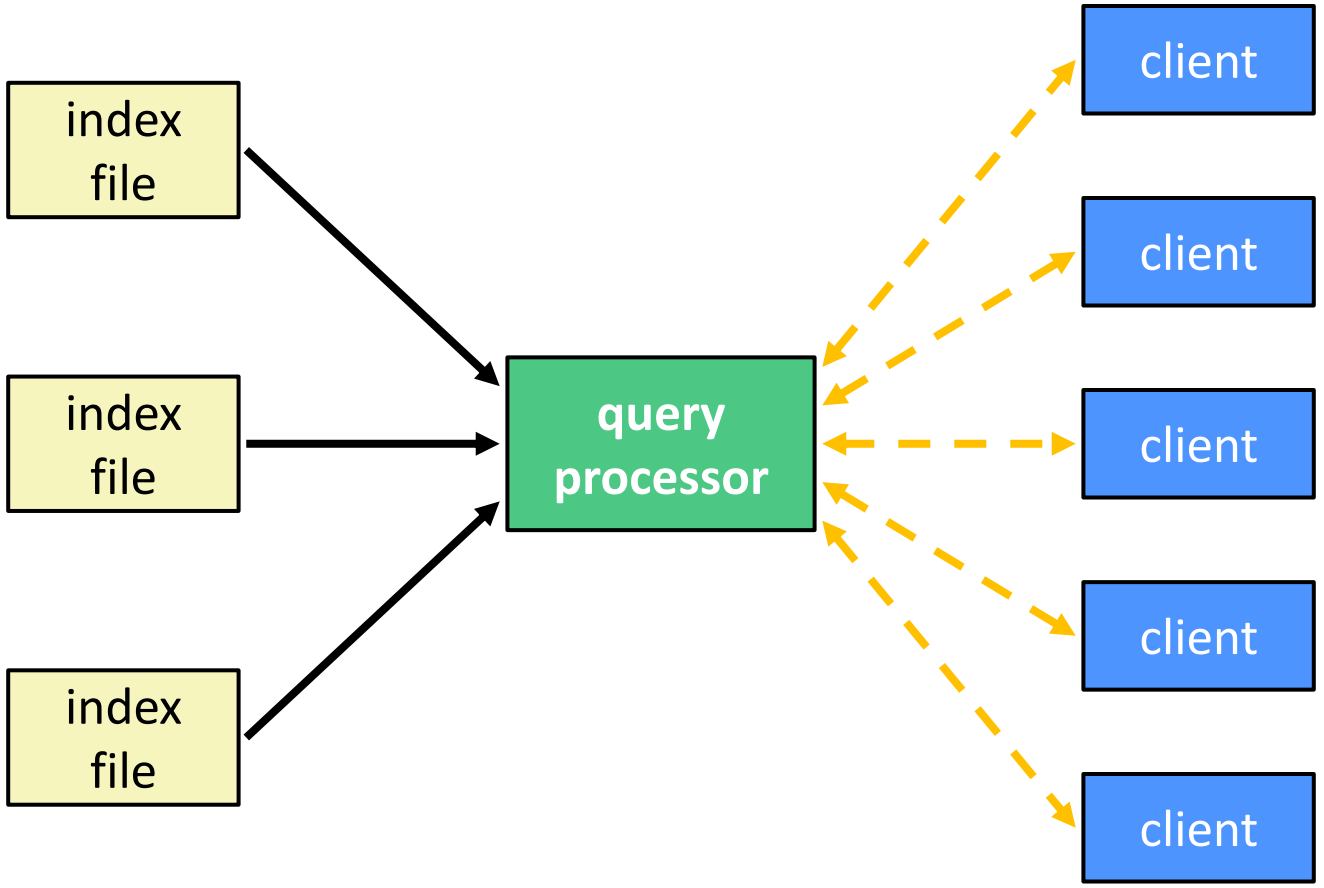


Execution Timeline: To Scale

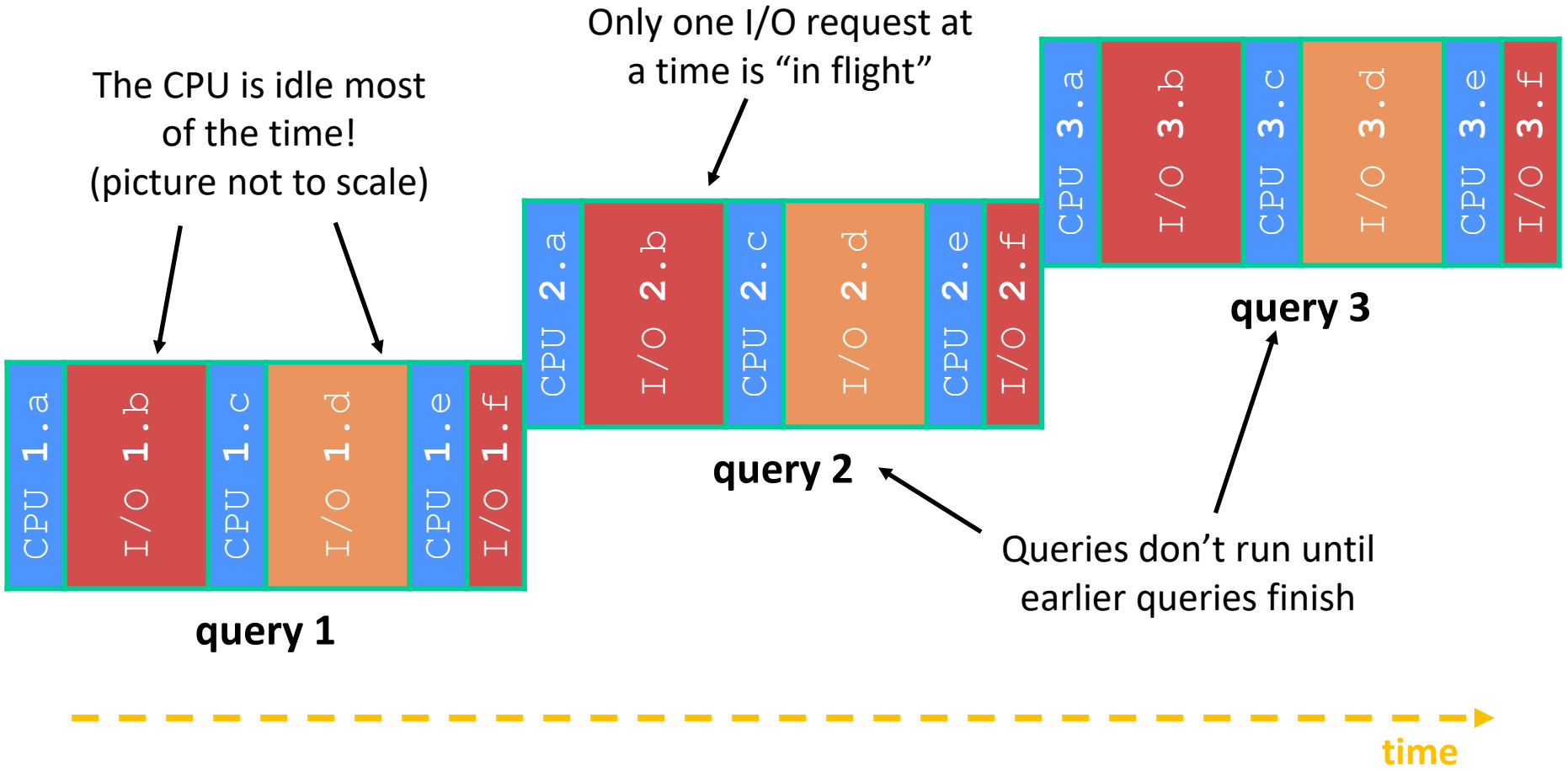
Model isn't perfect:
Technically also some cpu usage to setup I/O.
Network output also (probably) won't block program .....



Uh-Oh: Handling Multiple Clients



Uh-Oh: Handling Multiple Clients



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.

Concurrency Still Has Benefits

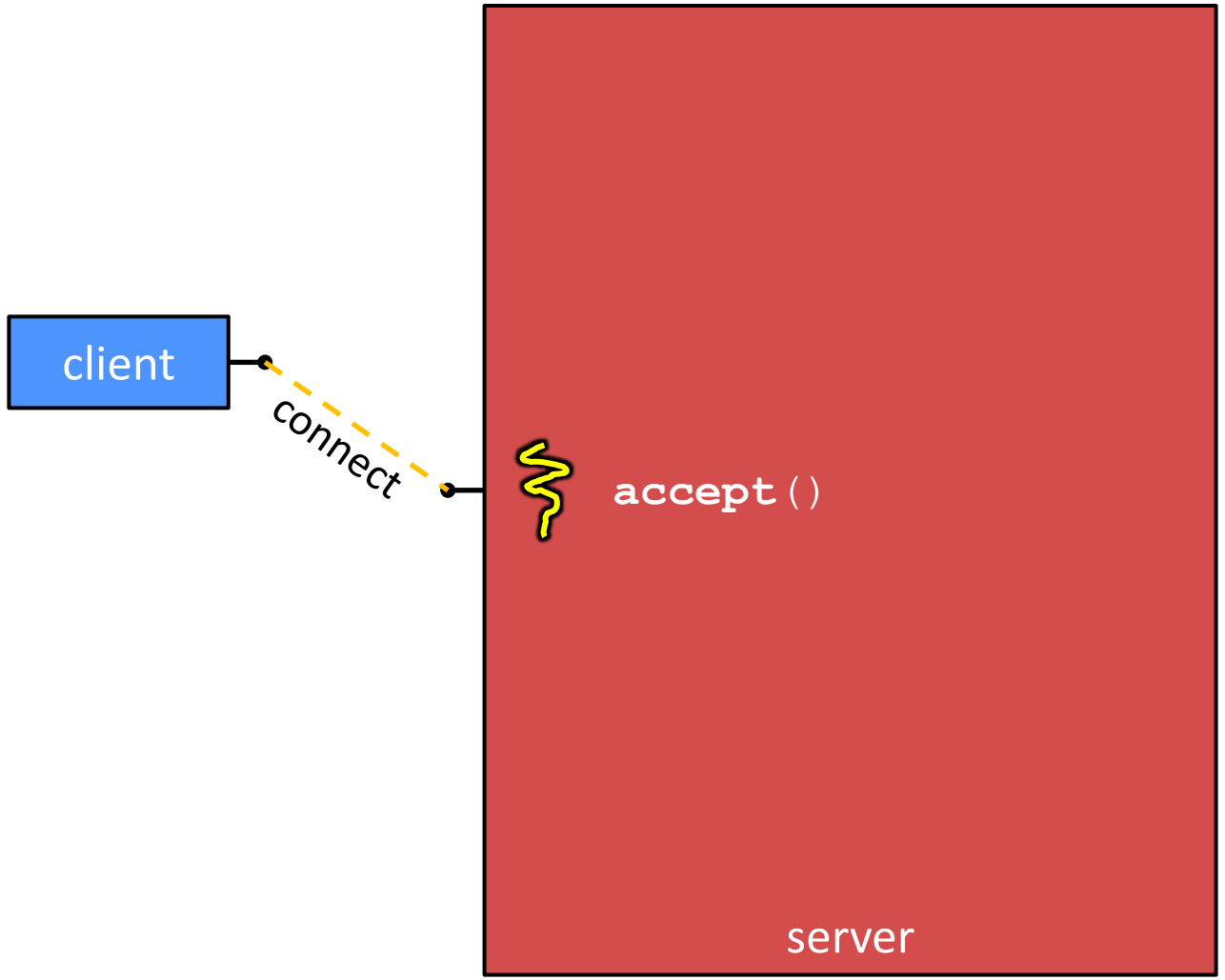
- Even if not running in parallel, there are still benefits to running code concurrently

- We focus on the CPU when we talk about programming, but there is a lot more to our machine than the CPU. But there are many time our CPU is waiting for something else
  - Waiting on results of I/O
    - Files
    - Pipes
    - Network, etc
  - Waiting on data to be loaded from memory
    - L2 cache, L3 cache, main memory, etc.

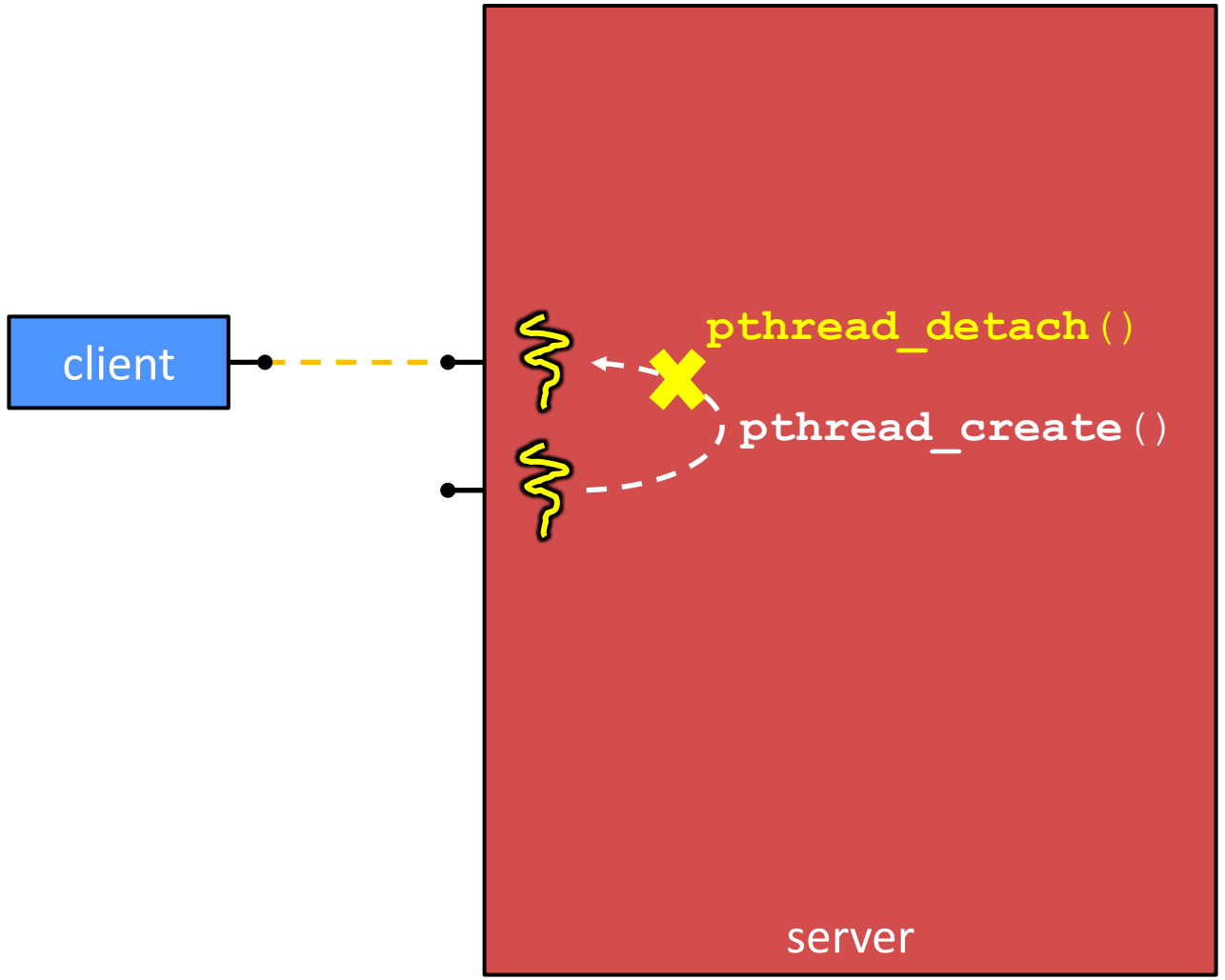
A Concurrent Implementation

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

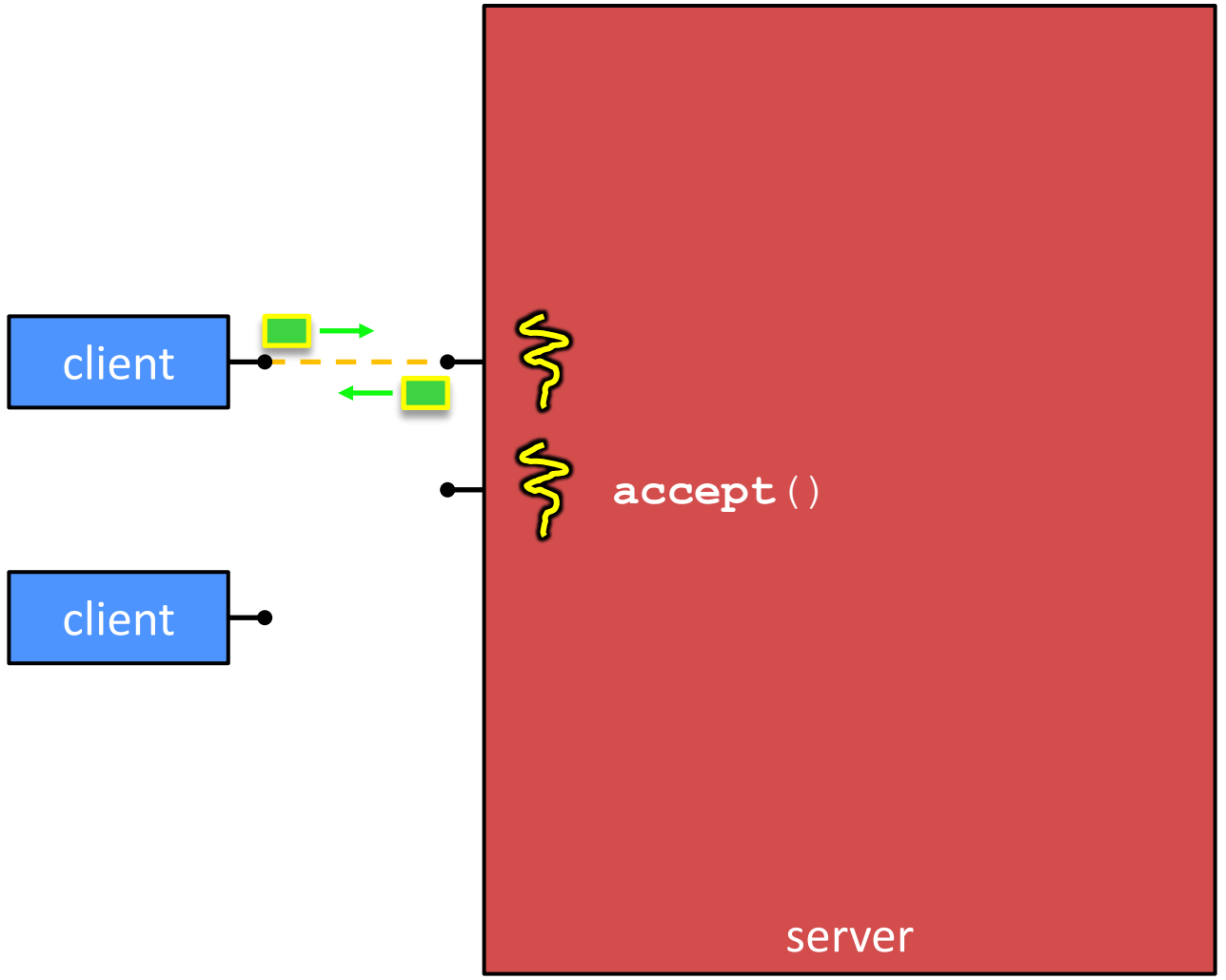
Multithreaded Server



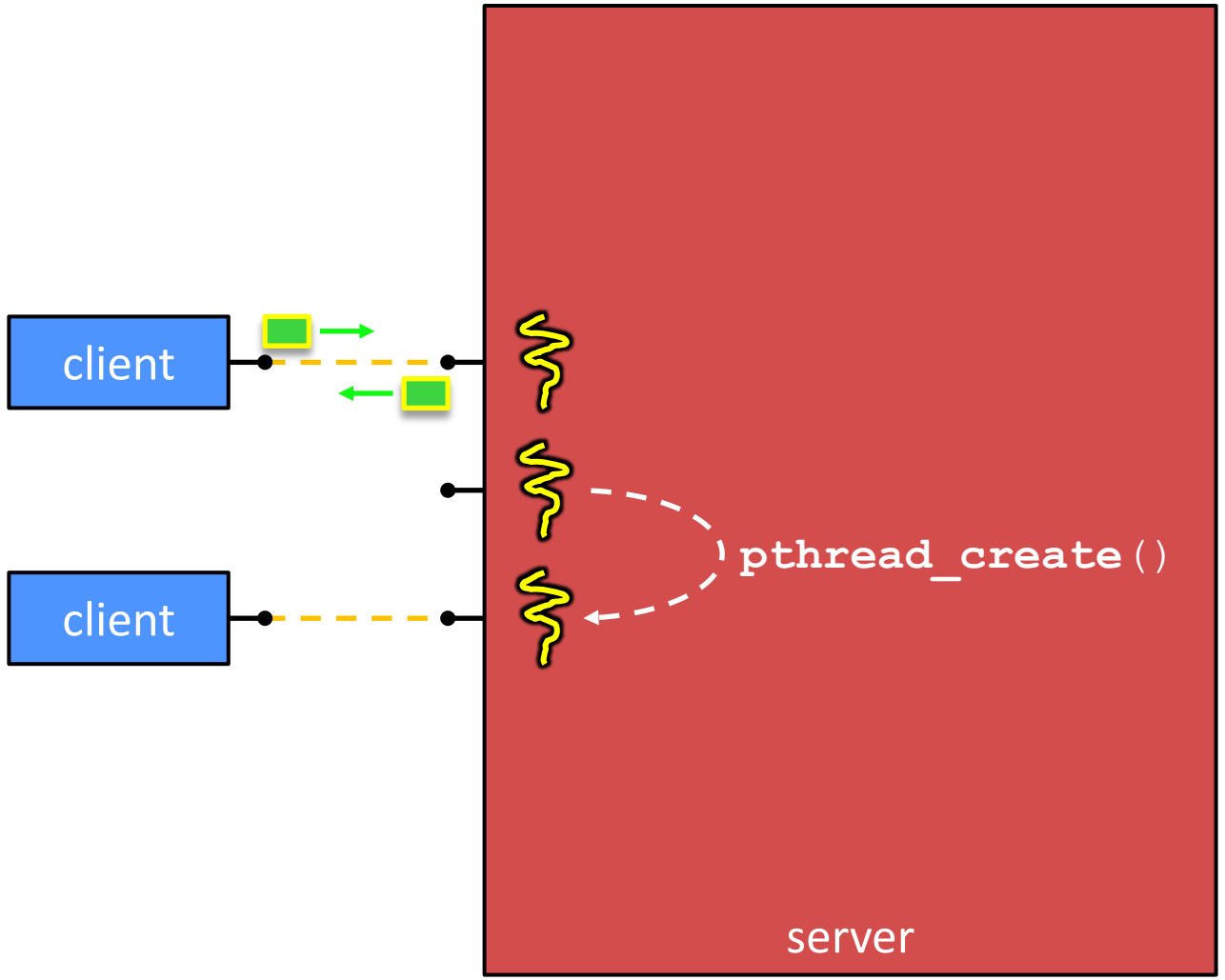
Multithreaded Server



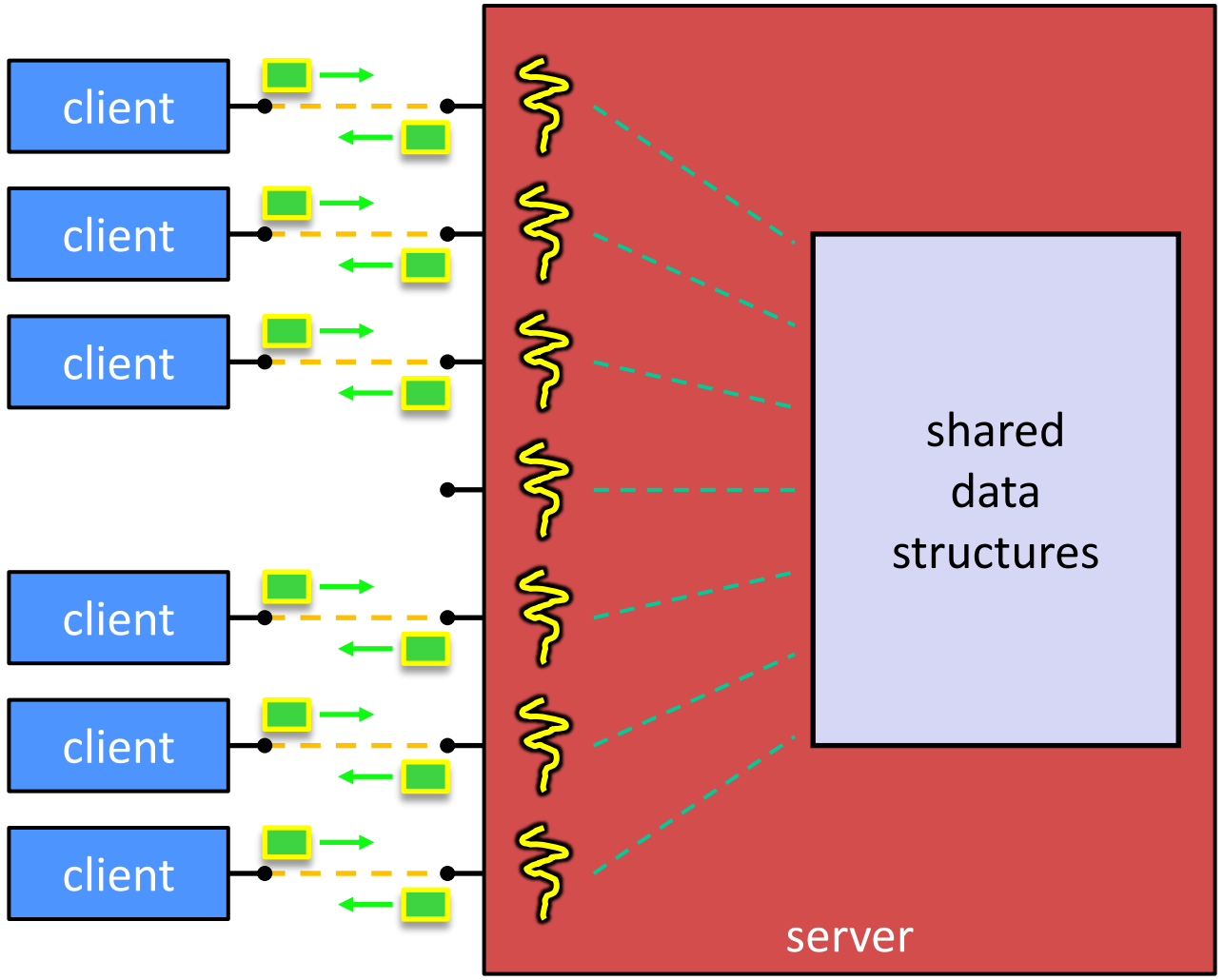
Multithreaded Server



Multithreaded Server



Multithreaded Server

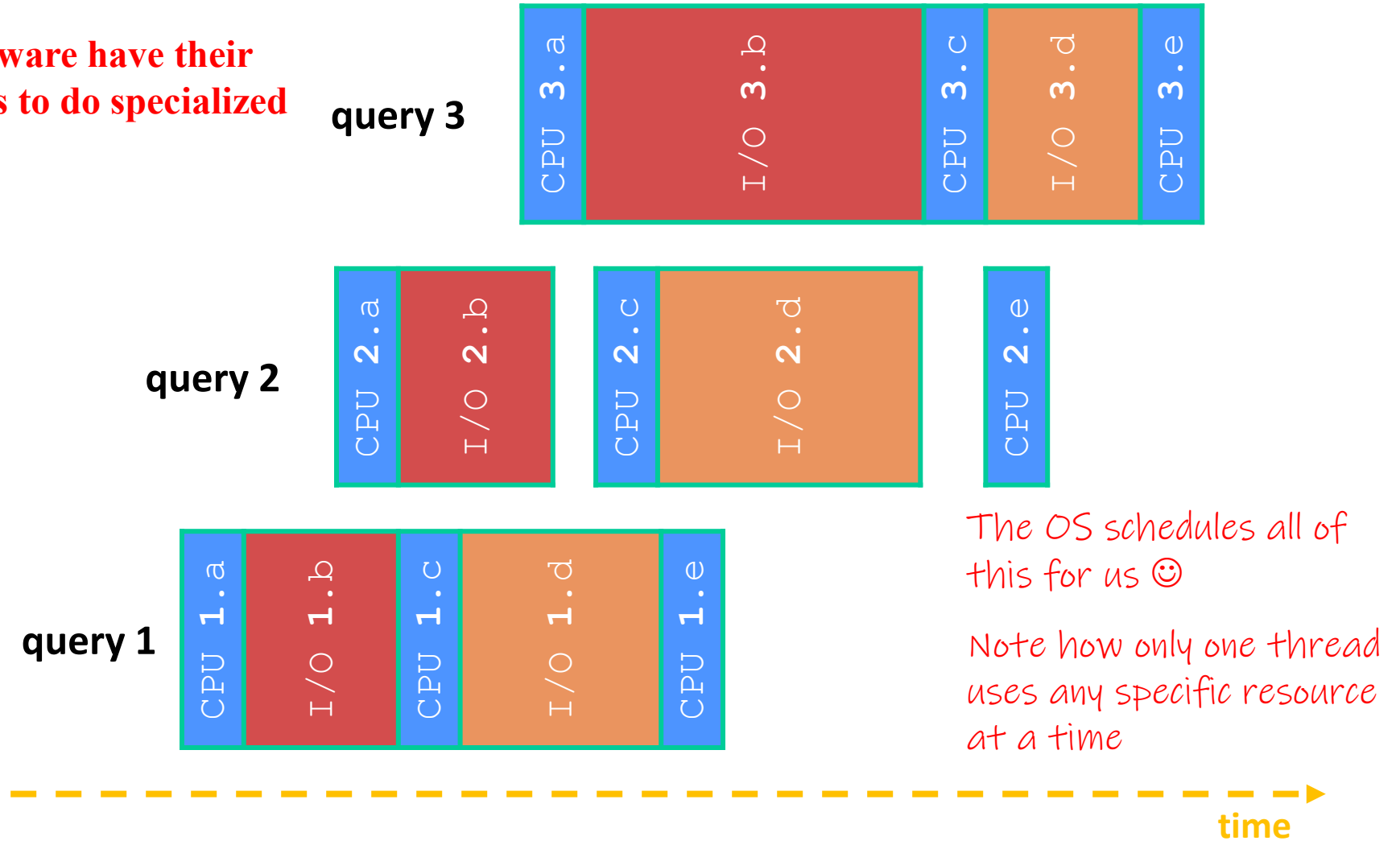


Multi-threaded Search Engine (Execution)

*Running with 1 CPU

The CPU is the Central Processing Unit

Other pieces of hardware have their own small processors to do specialized work.



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 on CPU if you have multiple CPUs/cores
  - Threads can run in "parallel" on different pieces of hardware

- Disadvantages:
  - If threads share data, you need locks or other synchronization
    - Very bug-prone and difficult to debug
  - Threads can introduce overhead
    - Thread Creation / Destruction, Lock contention, context switch overhead, and other issues
  - Need programming language support for threads
    - As long as you have a shell, you can fork a process

Lecture Outline

- Ahmdal's Law
- Parallelism vs Concurrency
- **Sequential Consistency**
- Overhead Cost

Discuss & Raise Hands

- Can this code ever print "AAAAAAA"?
- Assume that thd_main1 and thd_main2 are run by two separate threads.
- Assume no other threads use x or y

```
int x = 0;
int y = 0;

void* thd_main1(void* arg) {
  x = 3;
  y = 4;
}

void* thd_main2(void* arg) {
  if (y == 4) {
    if (x == 3) {
      cout << "We good :)\n";
    } else {
      cout << "AAAAAAAAAAA\n";
    }
  }
}
```

Sequential Consistency

- Do we know that **`t`** is set before **`g`** is set?

```
bool g = false;
int t = 0

void some_func(int arg) {
  t = arg;
  g = true;
}
```

Instruction & Memory Ordering

- Do we know that `t` is set before `g` is set?

NO

```
bool g = false;
int t = 0

void some_func(int arg) {
  t = arg;
  g = true;
}
```

The compiler may generate instructions that sets g first and then t
The Processor may execute these out of order or at the same time

Why? Optimizations on program performance

You can be guaranteed that `t` and `g` are set before `some_func` returns

Aside: Instruction & Memory Ordering

- The compiler may generate instructions with different ordering if it does not appear that it will affect the semantics of the function
  - Since `g = true;` is not affected by `t = arg;` then either one could execute first.

- The Processor may also execute these in a different order than what the compiler says

- Why? Optimizations on program performance
  - If you want to know more, look into “Out-of-Order Execution” and “Memory Order”

Aside: Memory Barriers

- How do we fix this?

- We can emit special instructions to the CPU and/or compiler to create a "memory barrier"
  - "all memory accesses before the barrier are guaranteed to happen before the memory accesses that come after the barrier"

  - A way to enforce an order in which memory accesses are ordered by the compiler and the CPU

  - This is done for us when we use a lock or use other methods of synchronizing threads.

Revisitng Sequential Consistency

- Here is the real definition of sequential consistency (from Wikipedia)
  - It is the property that:

    "… the result of any execution is the same **as if** the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

- The processor and compiler are allowed to "lie" to you.
  This is true for compiler optimizations outside of threads as well, but this is the place where they are most easily noticed.

Lecture Outline

- Ahmdal's Law
- Parallelism vs Concurrency
- Sequential Consistency
- **Overhead Cost**

Process Isolation

- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
  - Processes have separate address spaces
  - Processes have privilege levels to restrict access to resources
  - If one process crashes, others will keep running

- Inter-Process Communication (IPC) is limited, but possible
  - Pipes via pipe()
  - Sockets via socketpair()
  - Shared Memory via shm_open()

Processes vs Threads

- Processes are considered “more expensive” than threads. There is more overhead to enforce isolation
- Advantages:
  - No shared memory between processes
  - Processes are isolated. If one crashes, other processes keep going
  - Exec works without consuming other processes

- Disadvantages:
  - More overhead than threads during creation and context switching
    - Creation: creating new address space
    - Context switching: loading/unloading a new address space
  - Cannot easily share memory between processes – typically communicate through the file system

How fast is fork()?

- ~ 0.5 milliseconds per fork*
- ~ 0.05 milliseconds per thread creation*
  - ~10x faster than fork()

- Processes have more overhead since you must create a new** address space for the new process.

- *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …
  - Processes are known to be even slower on Windows
- **There are optimizations to minimize amount of copies made

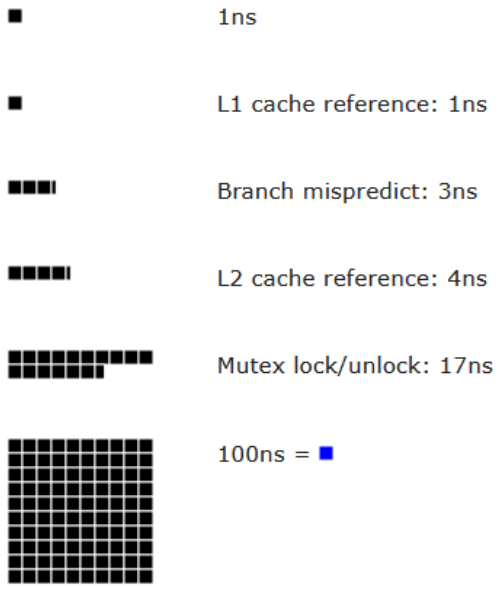
Thread Overhead

- Threads are sometimes called “Lightweight Processes”.
  - Processes came first, threads later. Threads were a lot cheaper to use than processes!
- Quicker to create and quicker to context switch between
- Communication can also be quicker. We don’t have to go through the file system, instead we can share memory!*

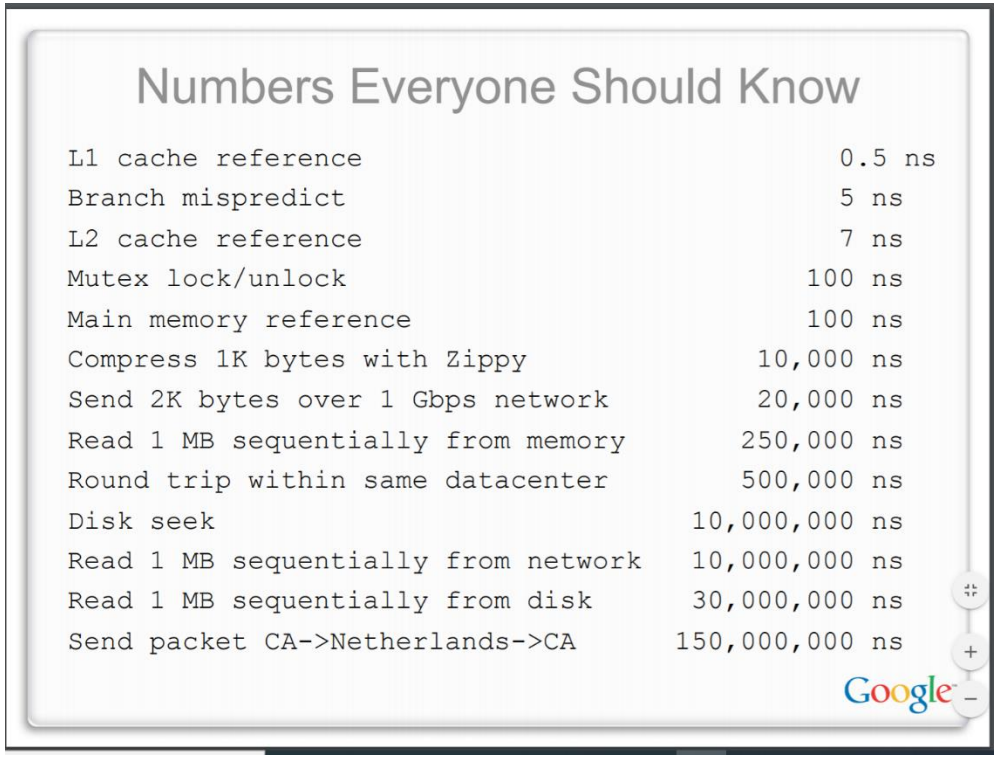
Lock Overhead

- Acquiring a lock is much slower than accessing memory

2020 numbers



2012 numbers



- *exact numbers here will vary, approximate ratio/order of magnitude is more useful.

What is the Critical Section?

- Generally, want to lock as little as possible
  - If we have a lock, other threads cannot hold it, so that makes parallelization worse!
  - Must be careful to lock *all* steps that must run uninterrupted (*i.e.* must run as an atomic unit), otherwise we get the wrong behaviour

```
for (int i = 0; i < LOOP_NUM; ++i) {
  pthread_mutex_lock(&lock);
  sum_total += 1;
  pthread_mutex_unlock(&lock);
}
```

- Also need to balance this with not acquiring and releasing the lock too often to avoid lock overhead ☹

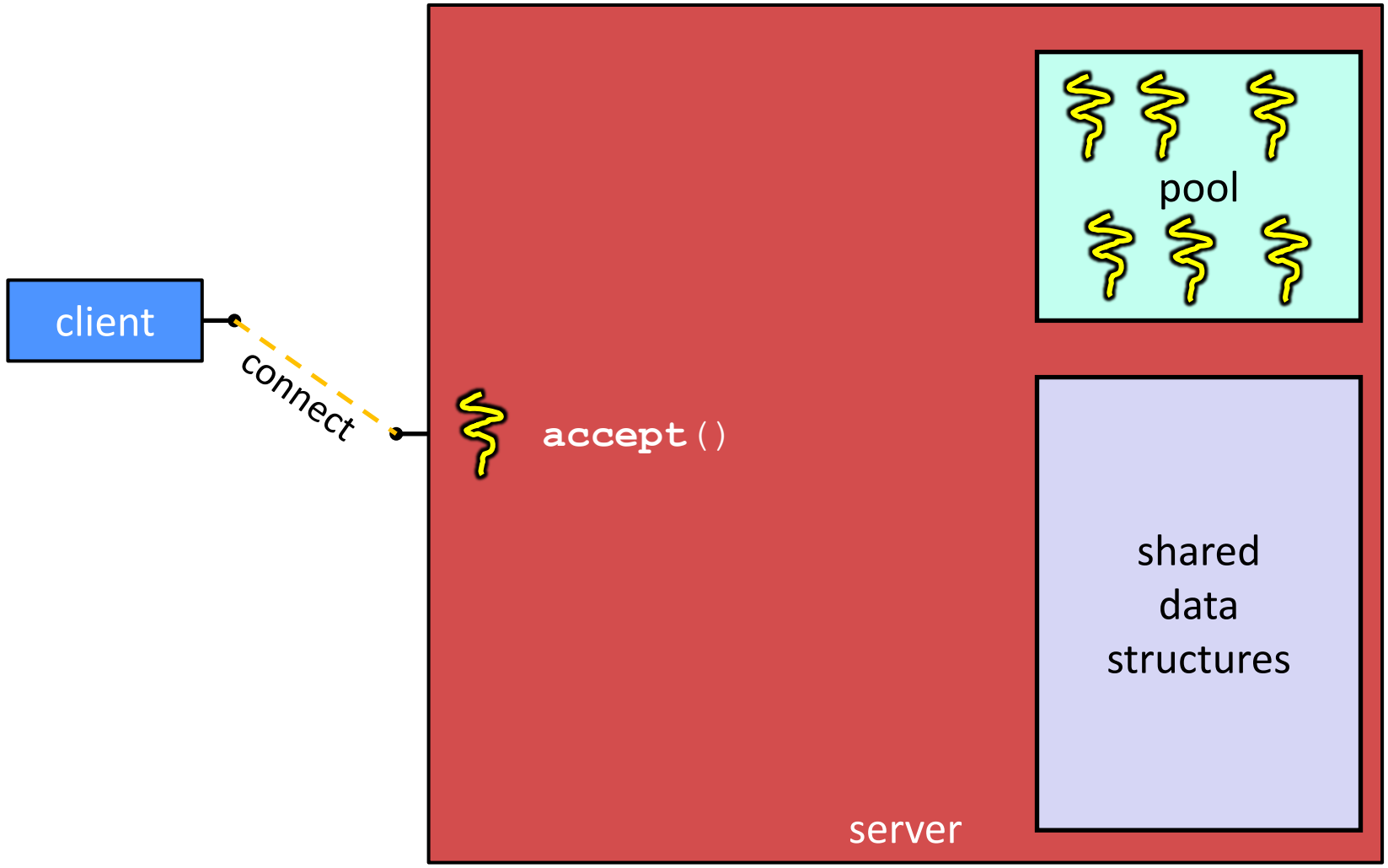
```
pthread_mutex_lock(&lock);
for (int i = 0; i < LOOP_NUM; ++i) {
  sum_total += 1;
}
pthread_mutex_unlock(&lock);
```

- Complex problem to balance!

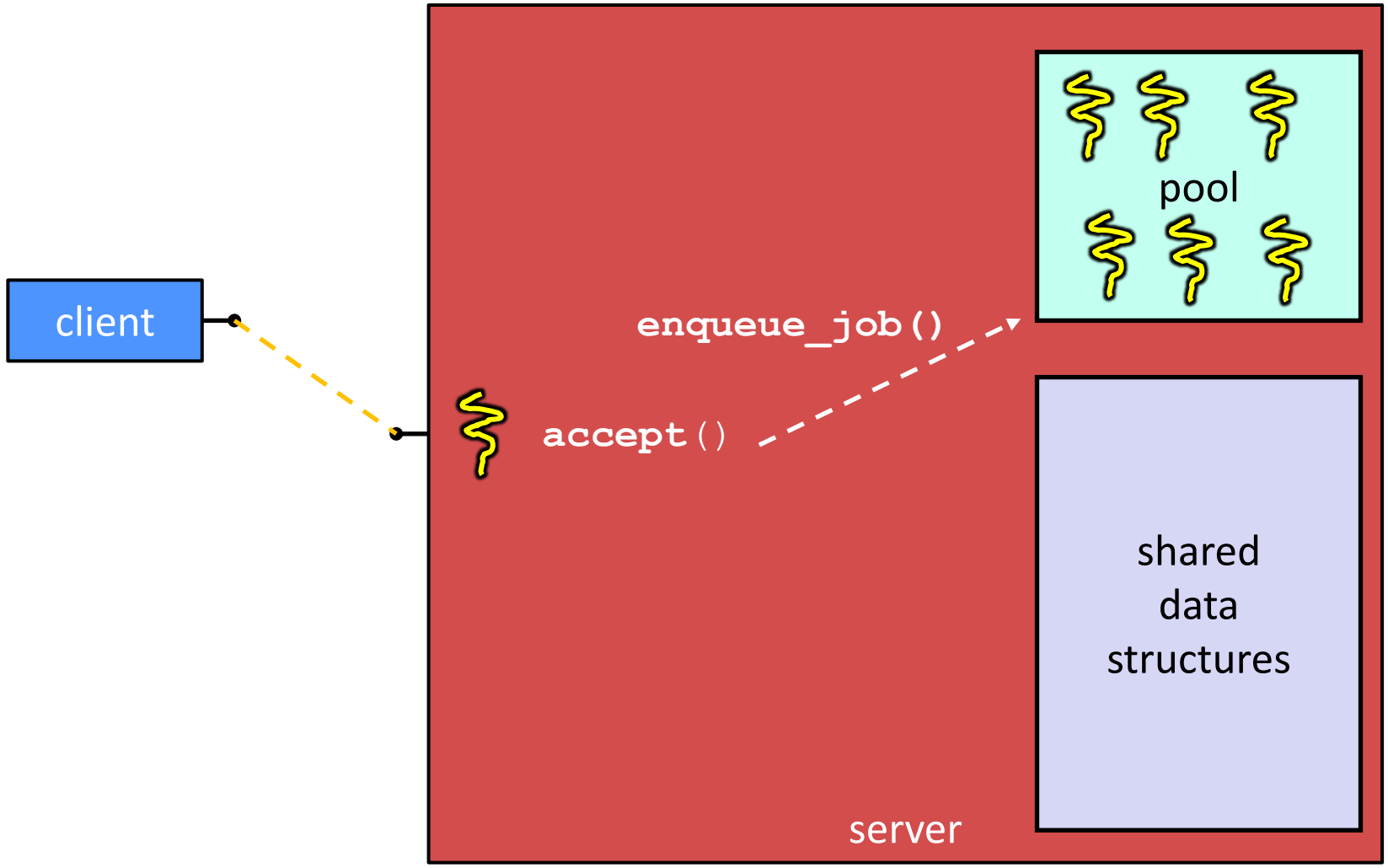
Limiting Overhead w/ Thread Pools

- Creation and destruction of threads can be expensive.
- What if we maintained a collection (a “pool”) of threads we could then reuse?
  - Often called a “worker-crew” model or “replicated workers” model
- Threads would wait for some task to be PRODUCED and then a thread would then go perform that task.
- You will have to implement one of these for the final project
  - More details next week on this

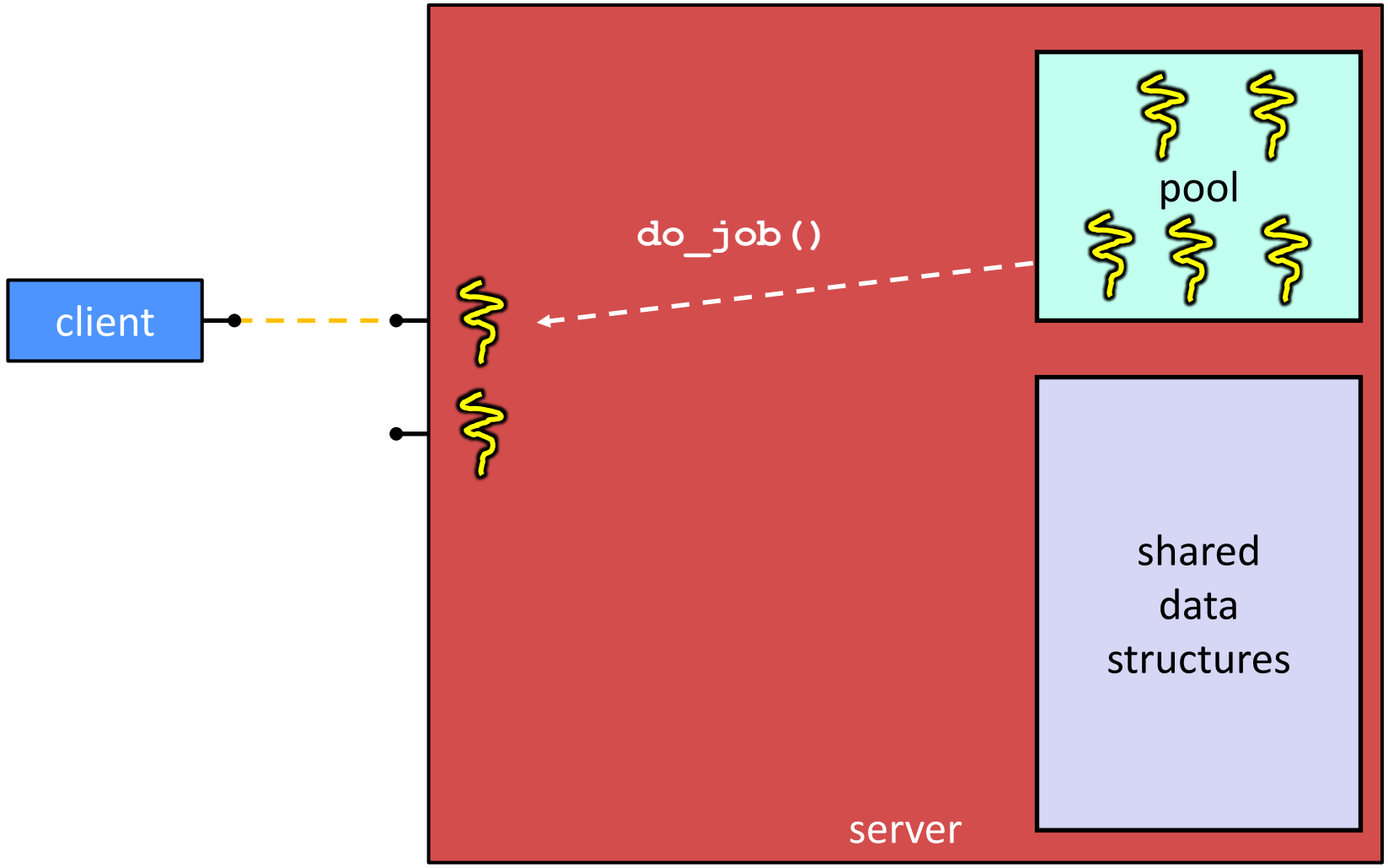
Multithreaded Server: Thread Pool



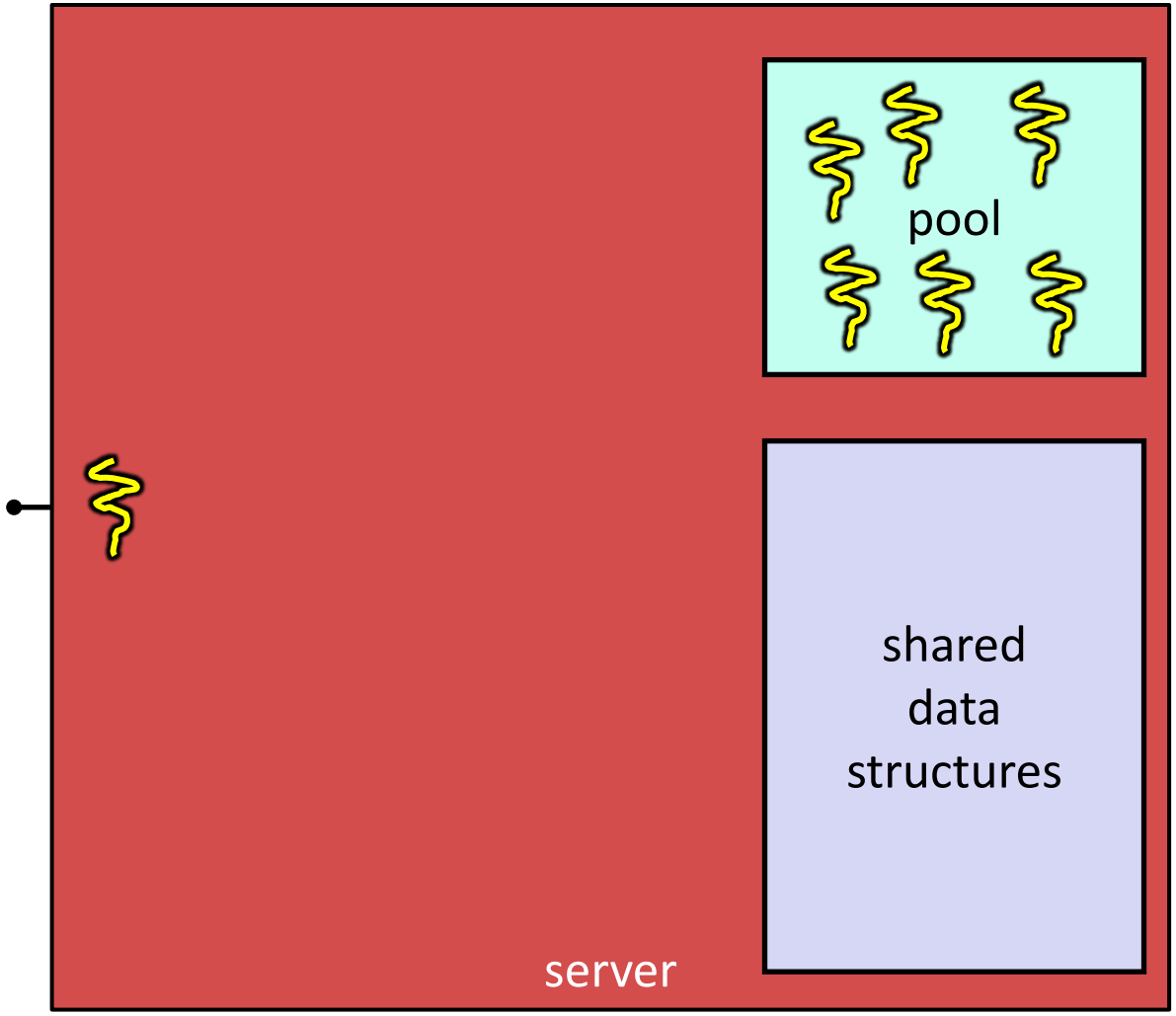
Multithreaded Server: Thread Pool



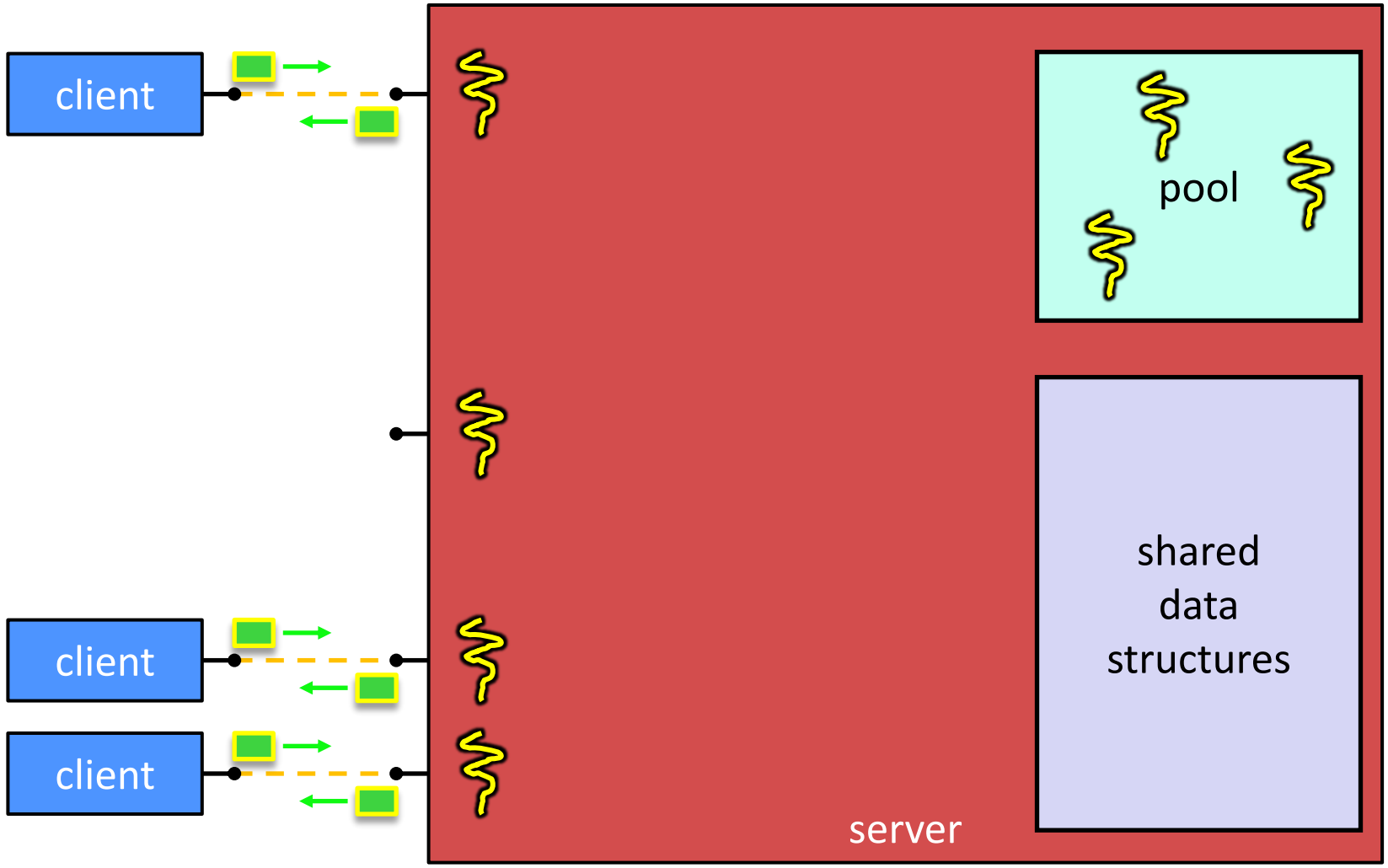
Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool



Ed Discussion

- There are at least 4 bad practices/mistakes done with locks in the following code. Find them.
  - Assume `g_lock` and `k_lock` have been initialized and will be cleaned up.
  - Assume that these functions will be called by multi-threaded code.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

Concurrency

- k++ could have a data race on it
- k_lock is uncessarily used around a+=b
- g_lock is used when k_lock should be used
- cin >> c does not need to be locked, could cause significant delays.

```
pthread_mutex_t g_lock, k_lock;
int g = 0, k = 0;

void fun1() {
  pthread_mutex_lock(&g_lock);
  g += 3;
  pthread_mutex_unlock(&g_lock);
  k++;
}

void fun2(int a, int b) {
  pthread_mutex_lock(&g_lock);
  g += a;
  pthread_mutex_unlock(&g_lock);
  pthread_mutex_lock(&k_lock);
  a += b;
  pthread_mutex_unlock(&k_lock);
}

void fun3() {
  int c;
  pthread_mutex_lock(&g_lock);
  cin >> c;  // have the user enter an int
  k += c;
  pthread_mutex_unlock(&g_lock);
}
```

That's all for now!

- Next time:
  - Exam Review
- After midterm
  - C++ threads, condition variables, atomics
  - Parallel Algorithms
- Hopefully you are doing well ☺