

Condition Variables & Thread Wrapup

Computer Systems Programming, Spring 2025

Instructor: Travis McGaha

Teaching Assistants:

Andrew Lukashchuk

Ashwin Alaparthi

Lobi Zhao

Angie Cao

Austin Lin

Pearl Liu

Aniket Ghorpade

Hassan Rizwan

Perrie Quek

pollev.com/tqm

❖ Meow

Administrivia

❖ HW08 – Threads Image Blurring

- Posted 😊
- Due Friday 4/04 at midnight, leaving open till Sunday night tho
- AG posted ~~soon~~ **NOW** (I encountered issues but they should be resolved ~~soon~~ **NOW**)
- Some hints gone over in Recitation this week

❖ Next Check-in posted soon-ish

Lecture Outline

- ❖ **Producer / Consumer**
- ❖ **Spinning & Condition Variables**
- ❖ Benefits of Concurrency
- ❖ Parallel Analysis

pollev.com/tqm

```
pthread_mutex_t lock;
string data;

void* produce(void* arg) {
    int fd = open(some_file, O_RDONLY);
    char buf[1024];
    ssize_t res = read(fd, buf, 1023);
    buf[res] = '\0';

    pthread_mutex_lock(&lock);
    data = string(buf);
    pthread_mutex_unlock(&lock);

    pthread_exit(NULL);
}
```

```
void* consume(void* arg) {
    pthread_mutex_lock(&lock);
    if (!data.empty()) {
        print(data);
    }
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}
```

Does this code have a data race?

- Assume that there is one thread running produce() and another thread running consume()
 - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
 - Assume lock initialized and funcs don’t fail
- ❖ Any issues with this code?

Race Condition vs Data Race

- ❖ Data-Race: when there are concurrent accesses to a shared resource, with at least one write, that can cause the shared resource to enter an invalid or “unexpected” state.
- ❖ Race-Condition: Where the program has different behaviour depending on the ordering of concurrent threads. This can happen even if all accesses to shared resources are “atomic” or “locked”
- ❖ The previous example has no data-race, but it does have a race condition

Thread Communication

- ❖ Sometimes threads may need to communicate with each other to know when they can perform operations
- ❖ Example: Producer and consumer threads
 - One thread creates tasks/data
 - One thread consumes the produced tasks/data to perform some operation
 - The consumer thread can only consume things once the producer has produced them
- ❖ Need to make sure this communication has no data race or race condition

Producer & Consumer Problem

- ❖ Common design pattern in concurrent programming.
 - There are at least two threads, at least one producer and at least one consumer.
 - The producer threads create some data that is then added to a shared data structure
 - Consumers will process and remove data from the shared data structure
- ❖ We need to make sure that the threads play nice

Aside: C++ deque

- ❖ I am using a c++ **deque** for this example so that we don't have to write our own data structure. This is not legal C
- ❖ Deque is a double ended queue, you can push to the front or back and pop from the front or back

```
// global deque of integers  
// will be initialized to be empty  
deque<int> dq {};  
  
int main() {  
    dq.push_back(3);           // adds 3  
    int val = dq.at(0);        // access index 0  
    dq.pop_front()             // delete first element  
    printf("%d\n", val);       // should print 3  
}
```

Producer Consumer Example

- ❖ Does this work?
- ❖ Assume that two threads are created, one assigned to each function

```
deque<int> dq {};  
  
void* producer_thread(void* arg) {  
    while (true) {  
        dq.push_back(long_computation());  
    }  
}  
  
void* consumer_thread(void* arg) {  
    while (true) {  
        while (dq.size() == 0) {  
            // do nothing  
        }  
        int val = dq.at(0);  
        dq.pop_front();  
        do_something(val);  
    }  
}
```

Poll Everywhere

pollev.com/tqm

- ❖ How do we use mutex to fix this? To make sure that the threads access dq safely.
 - You are only allowed to add calls to `pthread_mutex_lock` and `pthread_mutex_unlock`
 - Can add other mutexes if needed
- ❖ Similar code: `no_sync.cpp`

```
deque<int> dq {};
pthread_mutex_t dq_lock;

void* producer_thread(void* arg) {
    while (true) {
        dq.push_back(long_computation());
    }
}

void* consumer_thread(void* arg) {
    while (true) {
        while (dq.size() == 0) {
            // do nothing
        }
        int val = dq.at(0);
        dq.pop_front();
        do_something(val);
    }
}
```

Any issue?

- ❖ The code is correct, but do we notice anything wrong with this code?
- ❖ Maybe a common inefficiency that I have told you about several times before (just in other contexts?)
- ❖ The consumer code “busy waits” when there is nothing for it to consume.
 - It is particularly bad if we have multiple consumers, the locks make the busy waiting of the consumers sequential and use more CPU resources.

Thread Communication: Naïve Solution

- ❖ Consider the example where a thread must wait to be notified before it can print something out and terminate
- ❖ Possible solution: “Spinning”
 - Infinitely loop until the producer thread notifies that the consumer thread can print
- ❖ See `spinning.cpp`
 - The thread in the loop uses A LOT of cpu just checking until the value is safe
 - Use `top` to see CPU util
- ❖ Alternative: Condition variables

Condition Variables

- ❖ Variables that allow for a thread to wait until they are notified to resume
- ❖ Avoids waiting clock cycles “spinning”
- ❖ Done in the context of mutual exclusion
 - a thread must already have a lock, which it will temporarily release while waiting
 - Once notified, the thread will re-acquire a lock and resume execution

pthread and condition variables

❖ pthread.h defines datatype `pthread_cond_t`

❖

```
int pthread_cond_init(pthread_cond_t* cond,  
                      const pthread_condattr_t* attr);
```

- Initializes a condition variable with specified attributes

❖

```
int pthread_cond_destroy(pthread_cond_t* cond);
```

- “Uninitializes” a condition variable – clean up when done

pthread_s and condition variables

❖ pthread.h defines datatype `pthread_cond_t`

❖ `int pthread_cond_wait(pthread_cond_t* cond,
pthread_mutex_t* mutex);`

- Atomically releases the mutex and blocks on the condition variable. Once unblocked (by one of the functions below), function will return and calling thread will have the mutex locked

❖ `int pthread_cond_signal(pthread_cond_t* cond);`

- Unblock at least one of the threads on the specified condition

❖ `int pthread_cond_broadcast(pthread_cond_t* cond);`

- Unblock all threads blocked on the specified condition

pthread_cond_t Internal Pseudo-Code

- ❖ Here is some pseudo code to help understand condition variables

```
int pthread_cond_wait(pthread_cond_t* cond, pthread_mutex_t* mutex) {  
    pthread_mutex_unlock(&lock);  
    sleep_on_cond(cond); // sleeps till cond wakes them up  
    pthread_mutex_lock(&lock);  
    return 0;  
}
```

```
int pthread_cond_signal(pthread_cond_t* cond) {  
    wakeup_a_thread(cond); // wake's up a thread sleeping on the cond  
    return 0;  
}
```

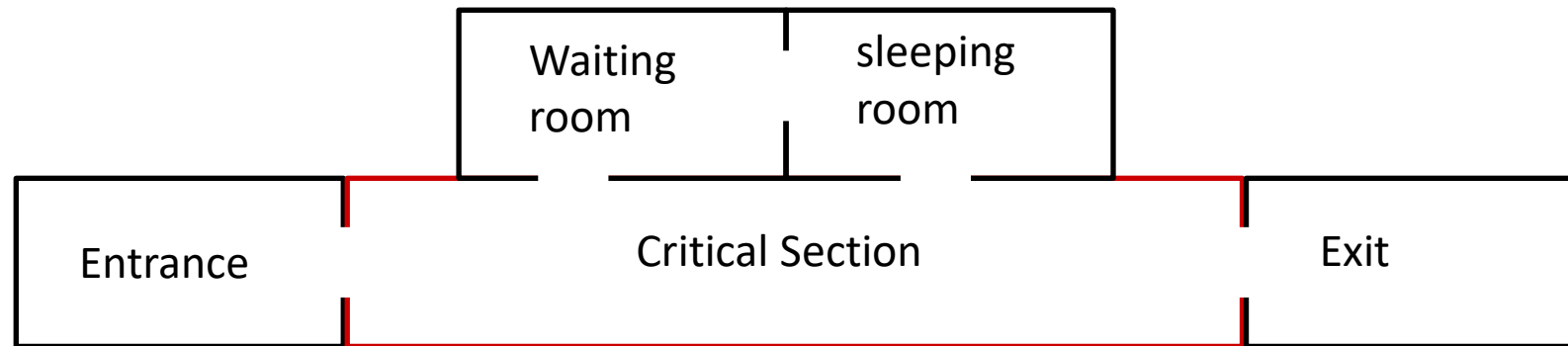
```
int pthread_cond_broadcast(pthread_cond_t* cond) {  
    for (thread_sleeping : cond->asleep) { // wake's up all threads  
        wakeup(thread_sleeping);  
    }  
    return 0;  
}
```

Demo: `cond.cpp`

- ❖ See `cond.cpp`
 - Changes our spinning code to use a condition variable properly
 - No issues with cpu utilization!

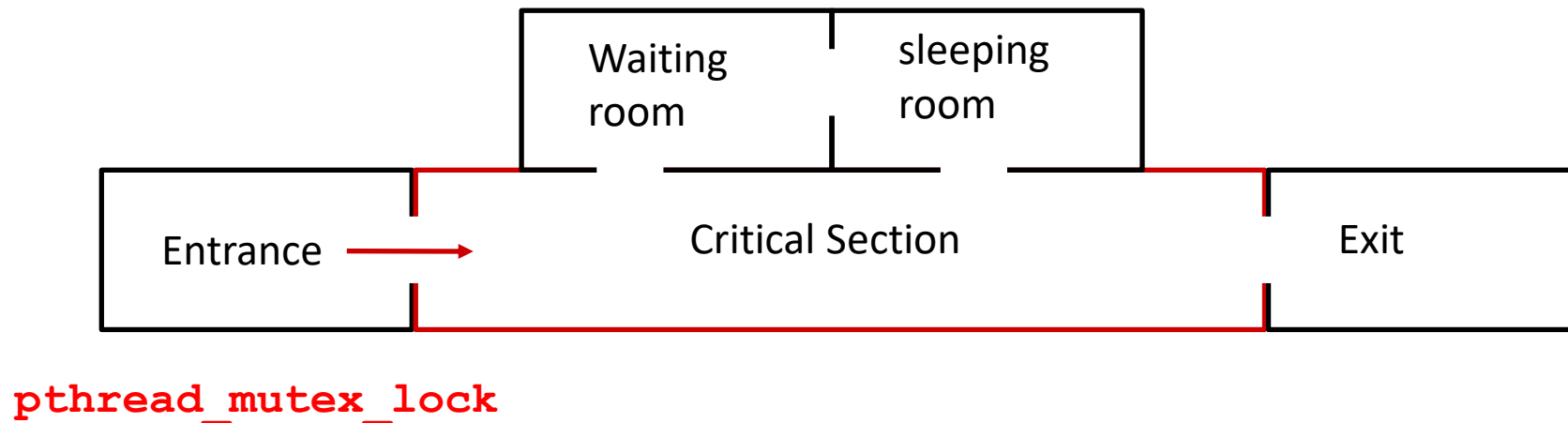
Condition Variable & Mutex Visualization

- ❖ This is to visualize how we are using condition variables in this example



Condition Variable & Mutex Visualization

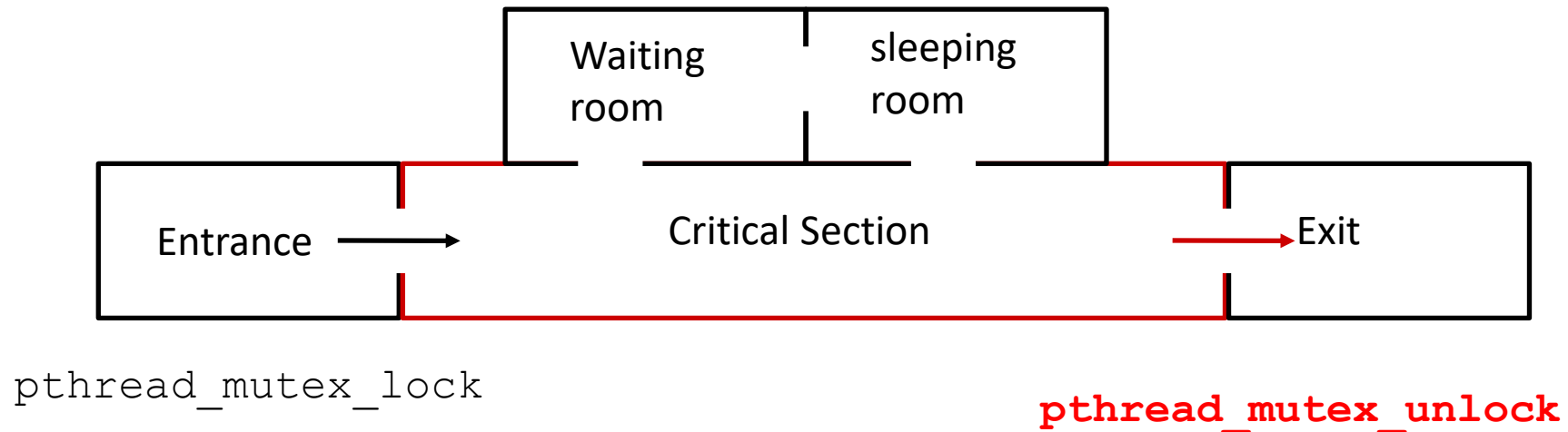
- ❖ This is to visualize how we are using condition variables in this example



A thread enters the critical section by acquiring a lock

Condition Variable & Mutex Visualization

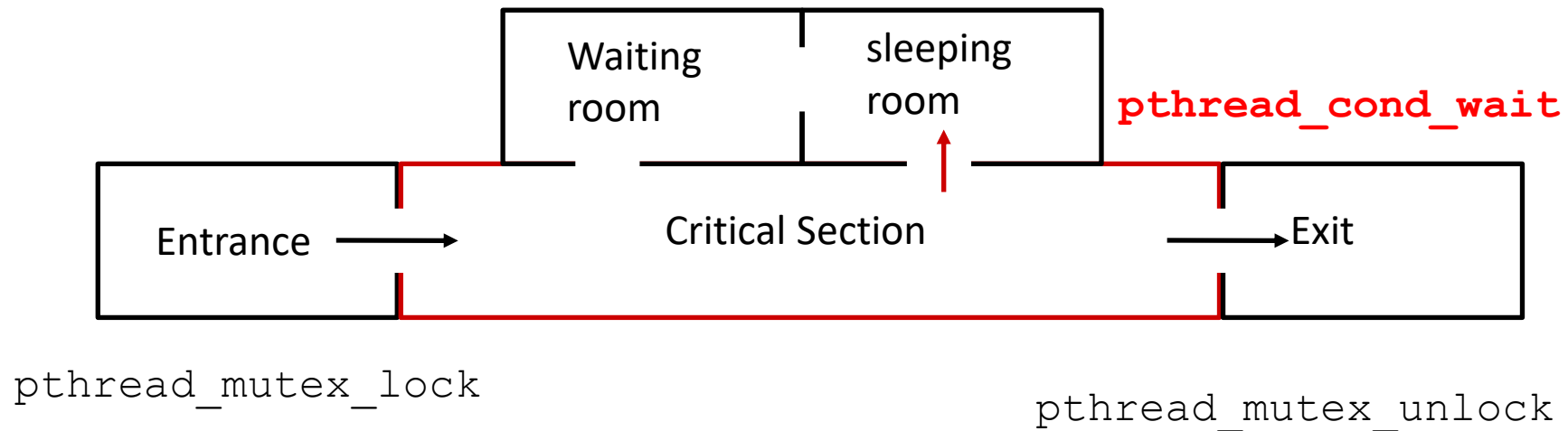
- ❖ This is to visualize how we are using condition variables in this example



A thread can exit the critical section by acquiring a lock

Condition Variable & Mutex Visualization

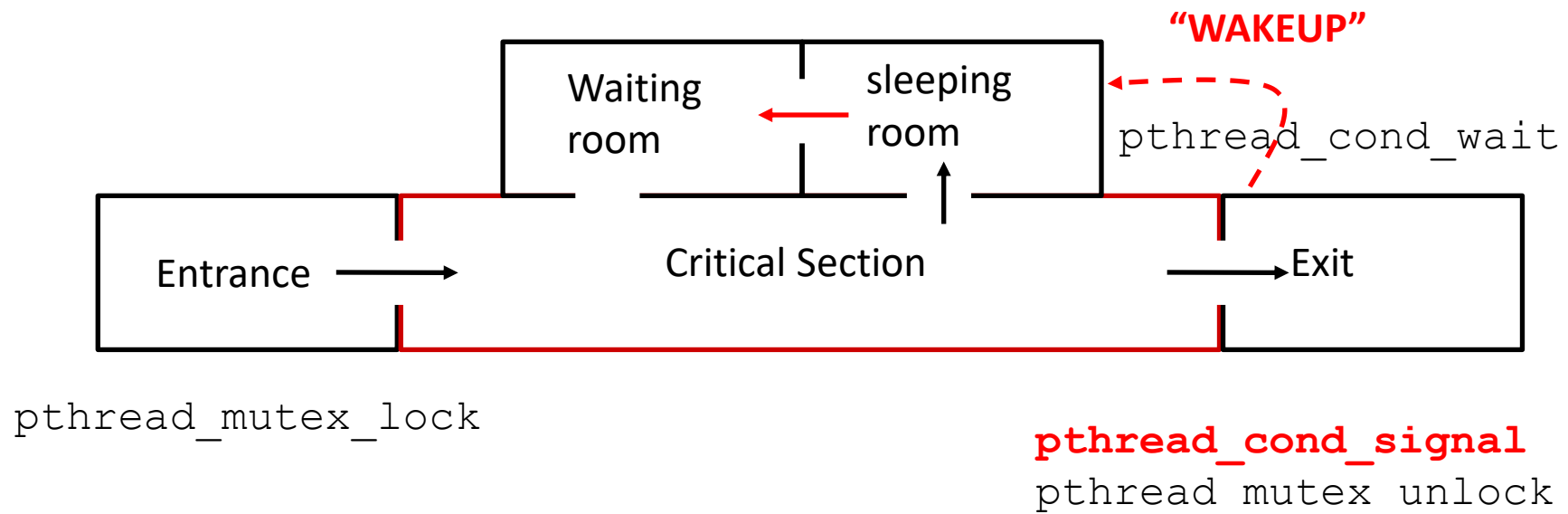
- ❖ This is to visualize how we are using condition variables in this example



If a thread can't complete its action, or must wait for some change in state, it can "go to sleep" until someone wakes it up later.
It will release the lock implicitly when it goes to sleep

Condition Variable & Mutex Visualization

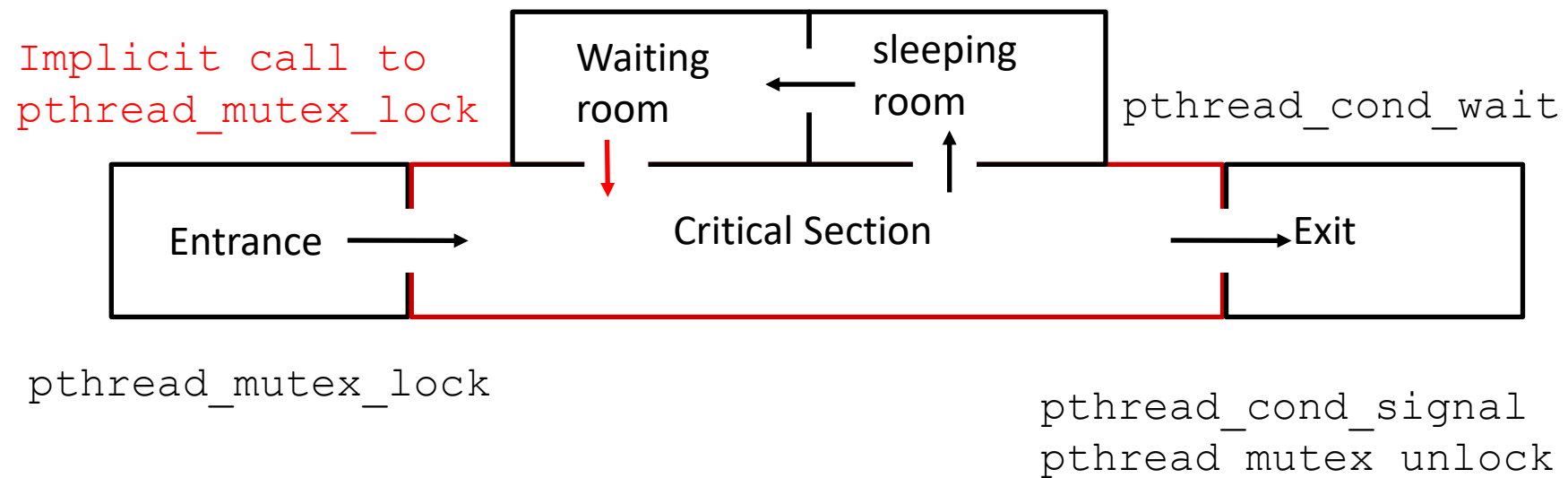
- ❖ This is to visualize how we are using condition variables in this example



When a thread modifies state and then leaves the critical section, it can also call `pthread_cond_signal` to wake up threads sleeping on that condition variable

Condition Variable & Mutex Visualization

- ❖ This is to visualize how we are using condition variables in this example



One or more sleeping threads wake up and attempt to acquire the lock.
Like a normal call to `pthread_mutex_lock` the thread will block until it can acquire the lock

Lecture Outline

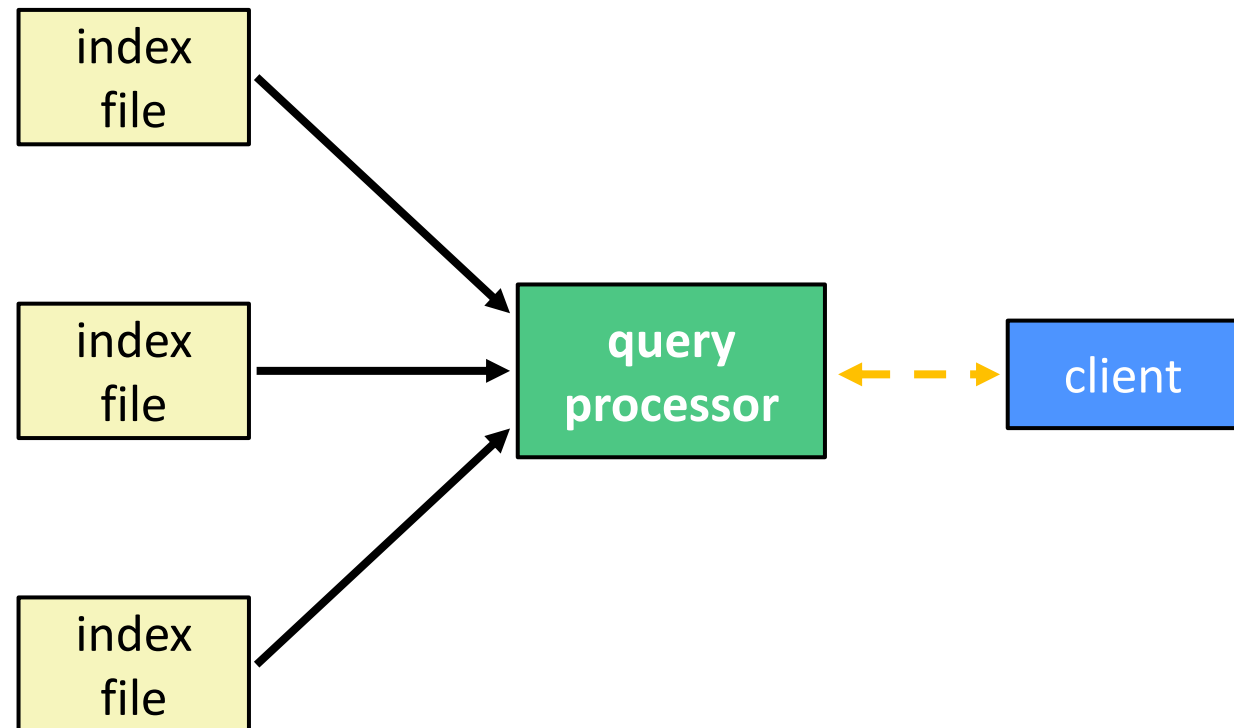
- ❖ Producer / Consumer
- ❖ Spinning & Condition Variables
- ❖ **Benefits of Concurrency**
- ❖ Parallel Analysis

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



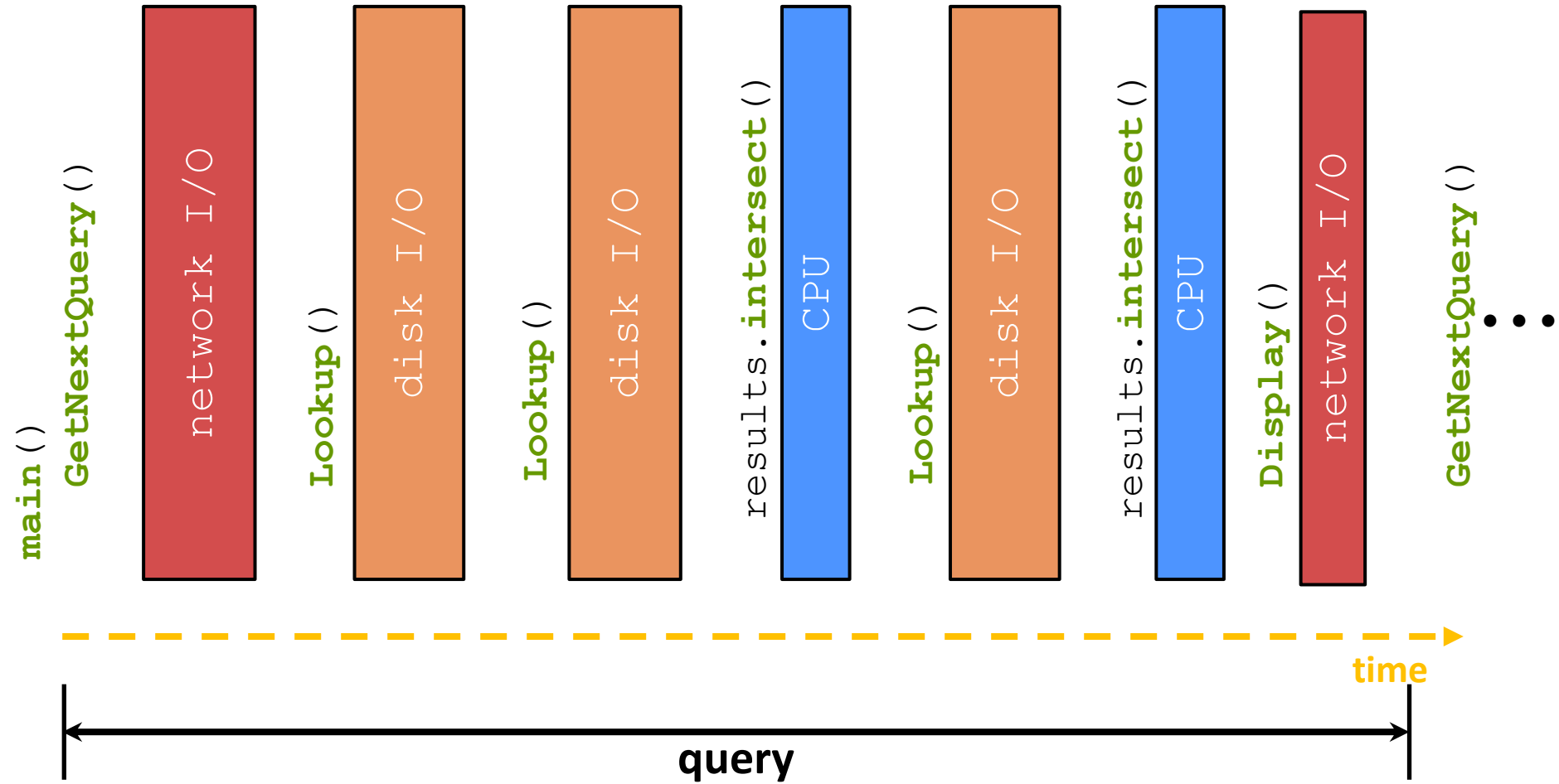
pollev.com/tqm

- ❖ 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);
    }
}
```

Execution Timeline: a Multi-Word Query



What About I/O-caused Latency?

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

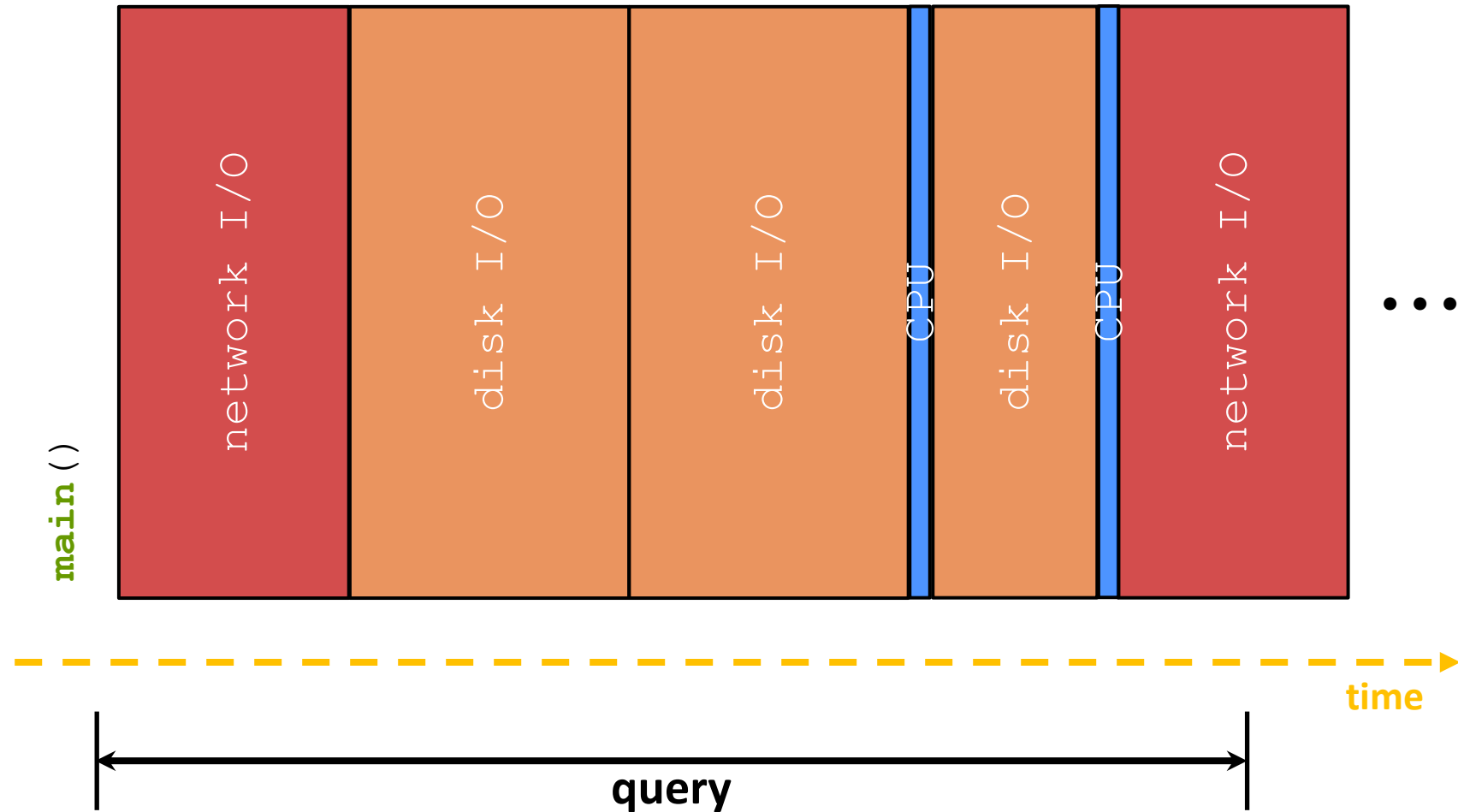
Numbers Everyone Should Know	
L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	100 ns
Main memory reference	100 ns
Compress 1K bytes with Zip	10,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within 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

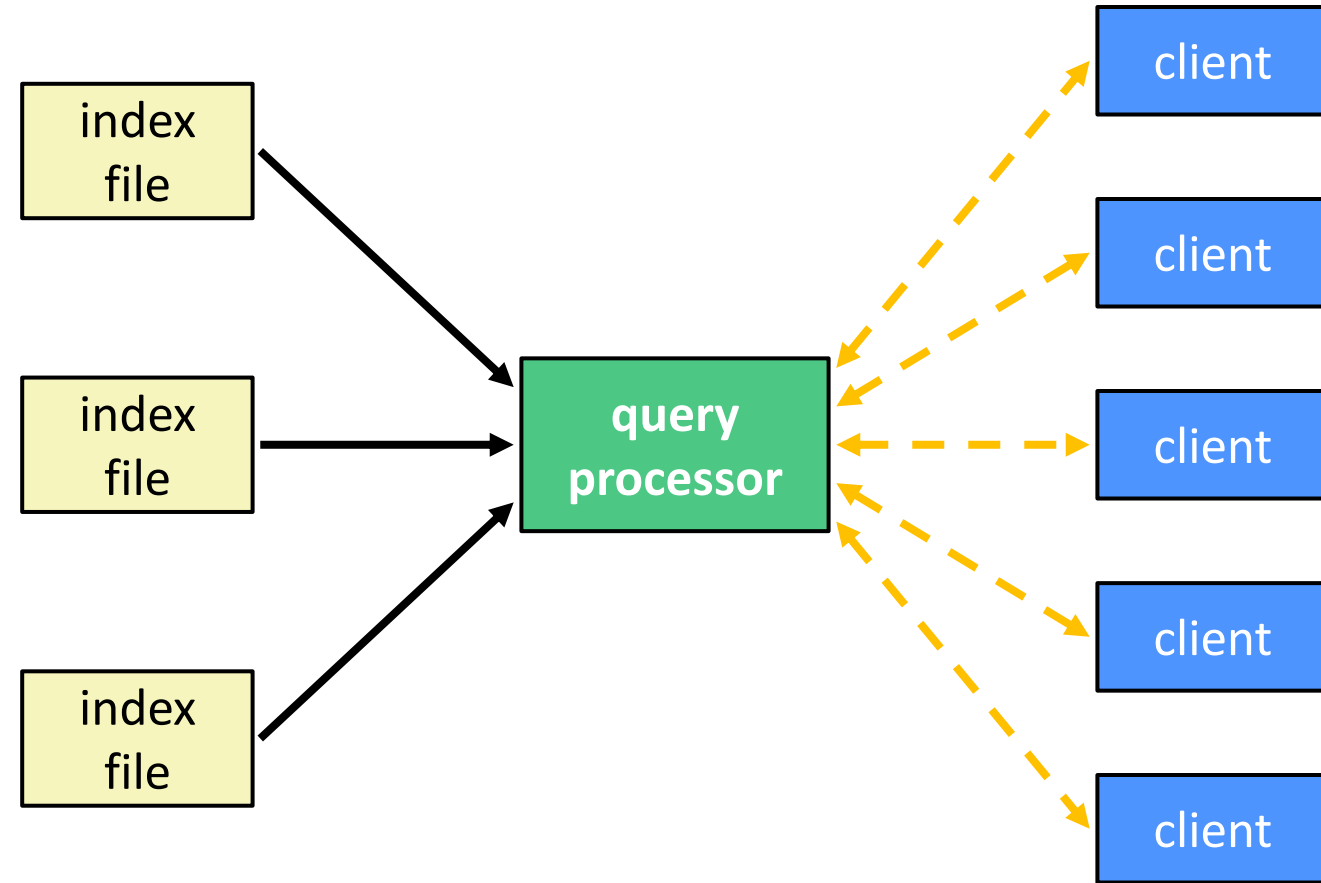
Model isn't perfect:

Technically also some cpu usage to setup I/O.

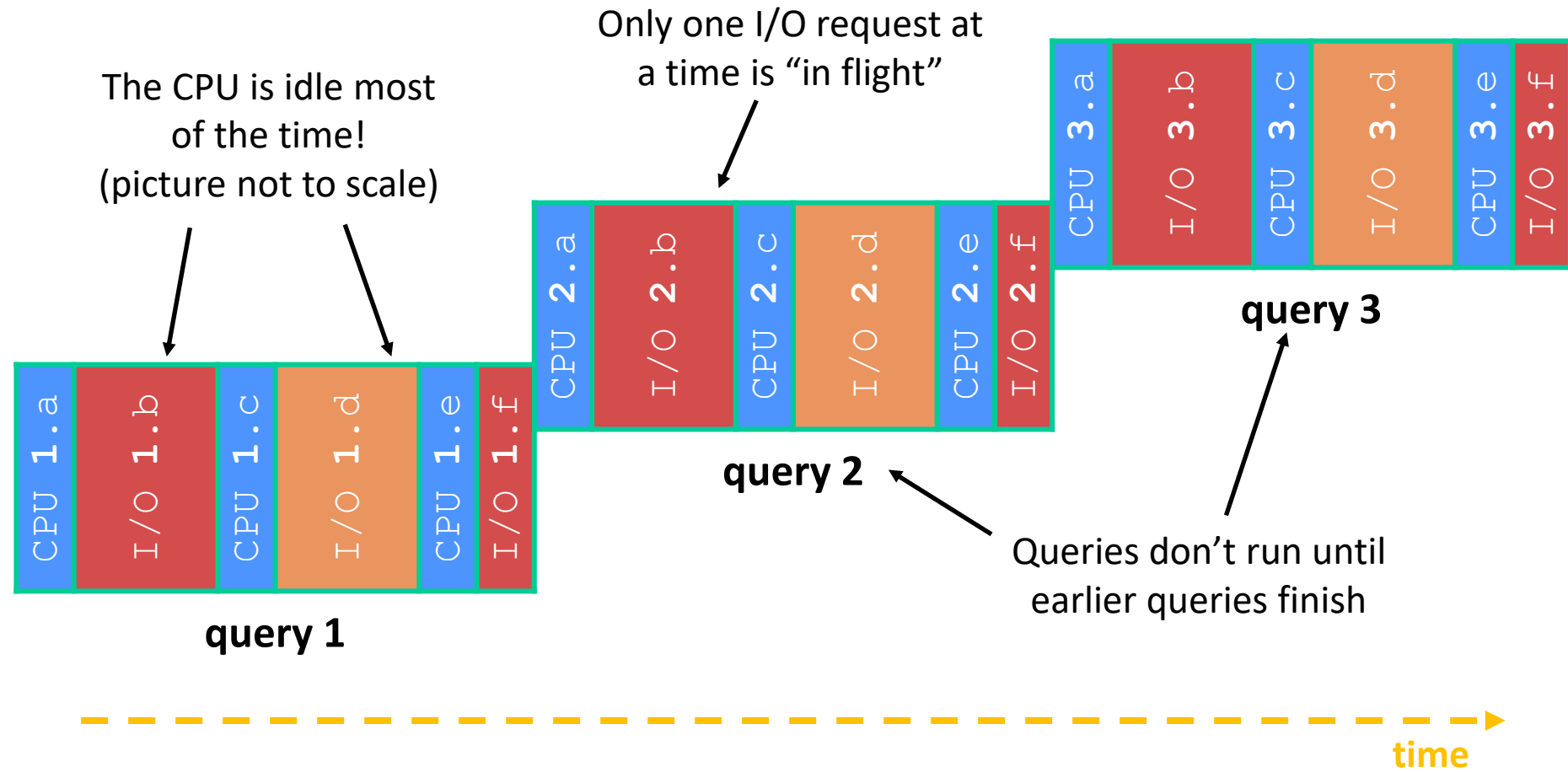
Network output also (probably) won't block program



Uh-Oh: Handling Multiple Clients



Uh-Oh: Handling Multiple Clients



Sequential Can Be Inefficient

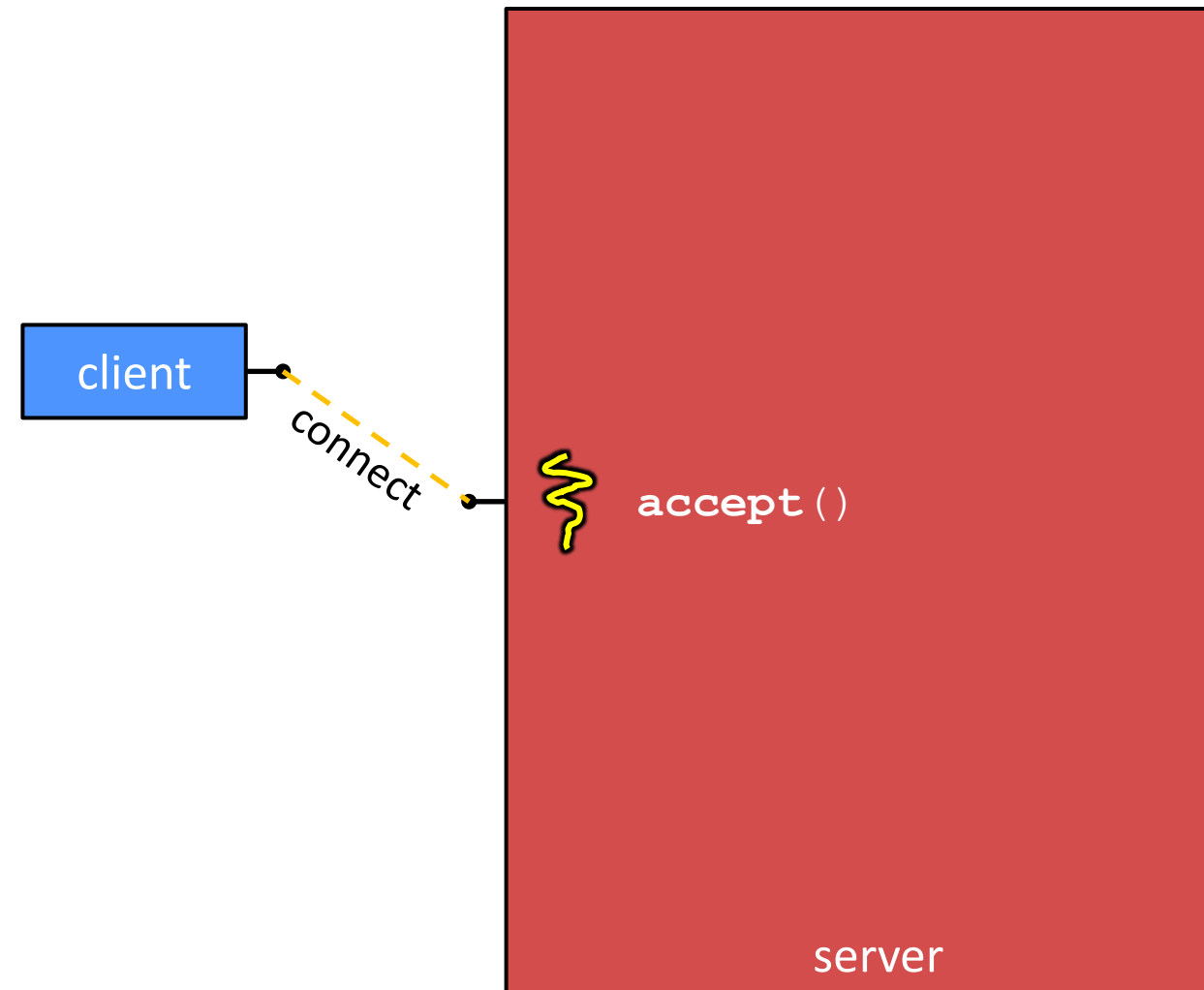
- ❖ Only one query is being processed at a time
 - All other queries queue up behind the first one
 - And clients queue up behind the queries ...
- ❖ Even while processing one query, the CPU is idle the vast majority of the time
 - It is *blocked* waiting for I/O to complete
 - Disk I/O can be very,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

A Concurrent Implementation

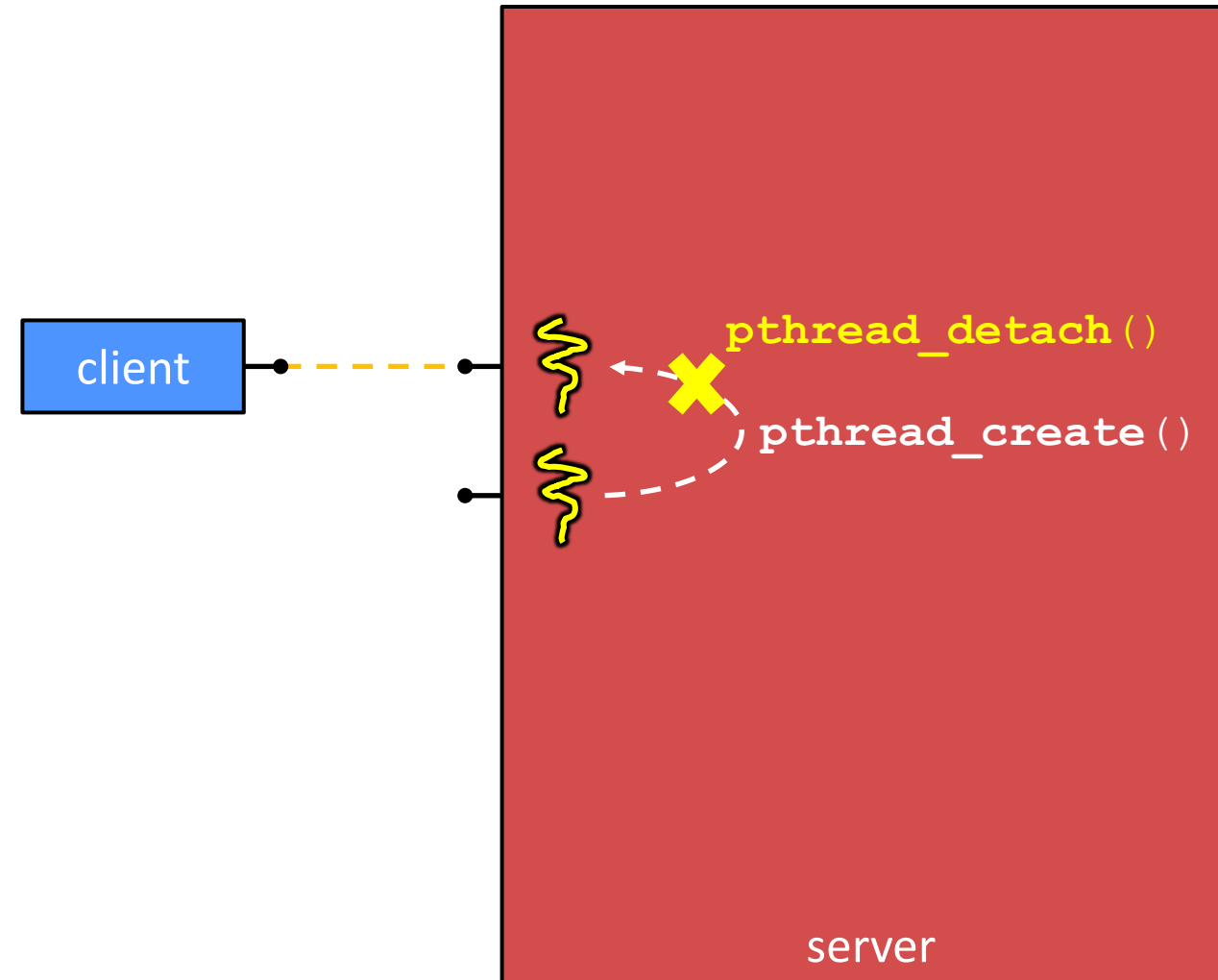
❖ Use multiple threads

- 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

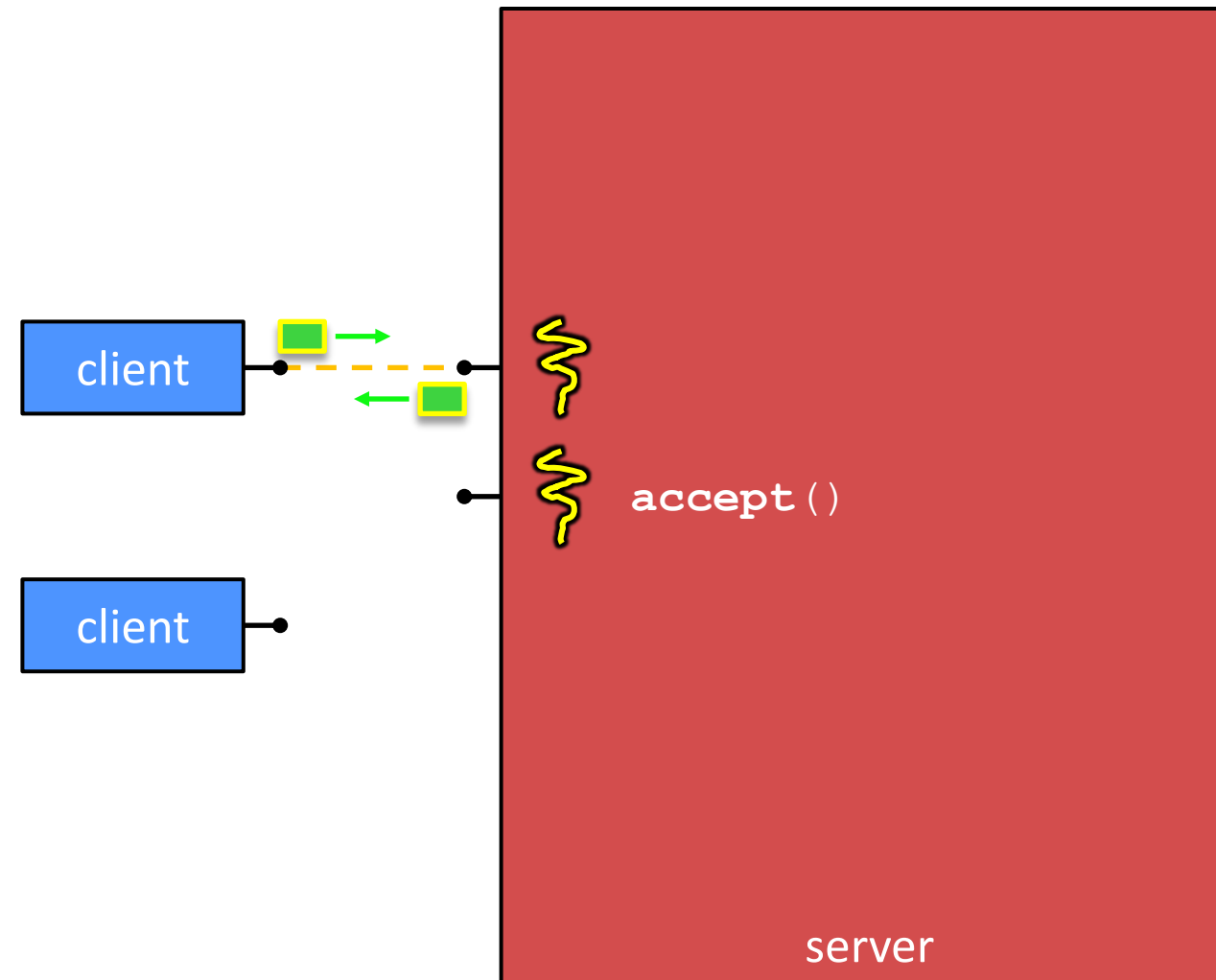
Multithreaded Server



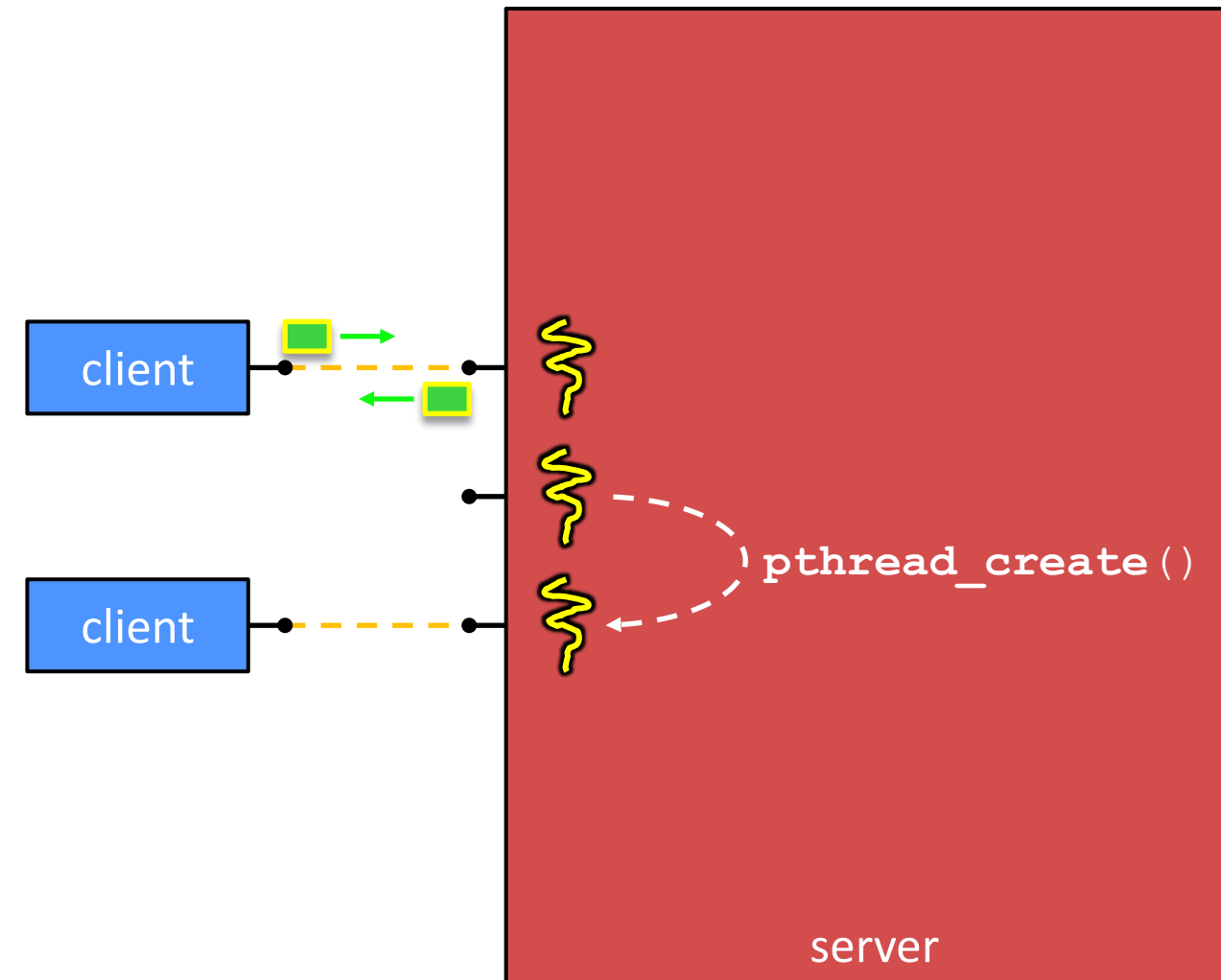
Multithreaded Server



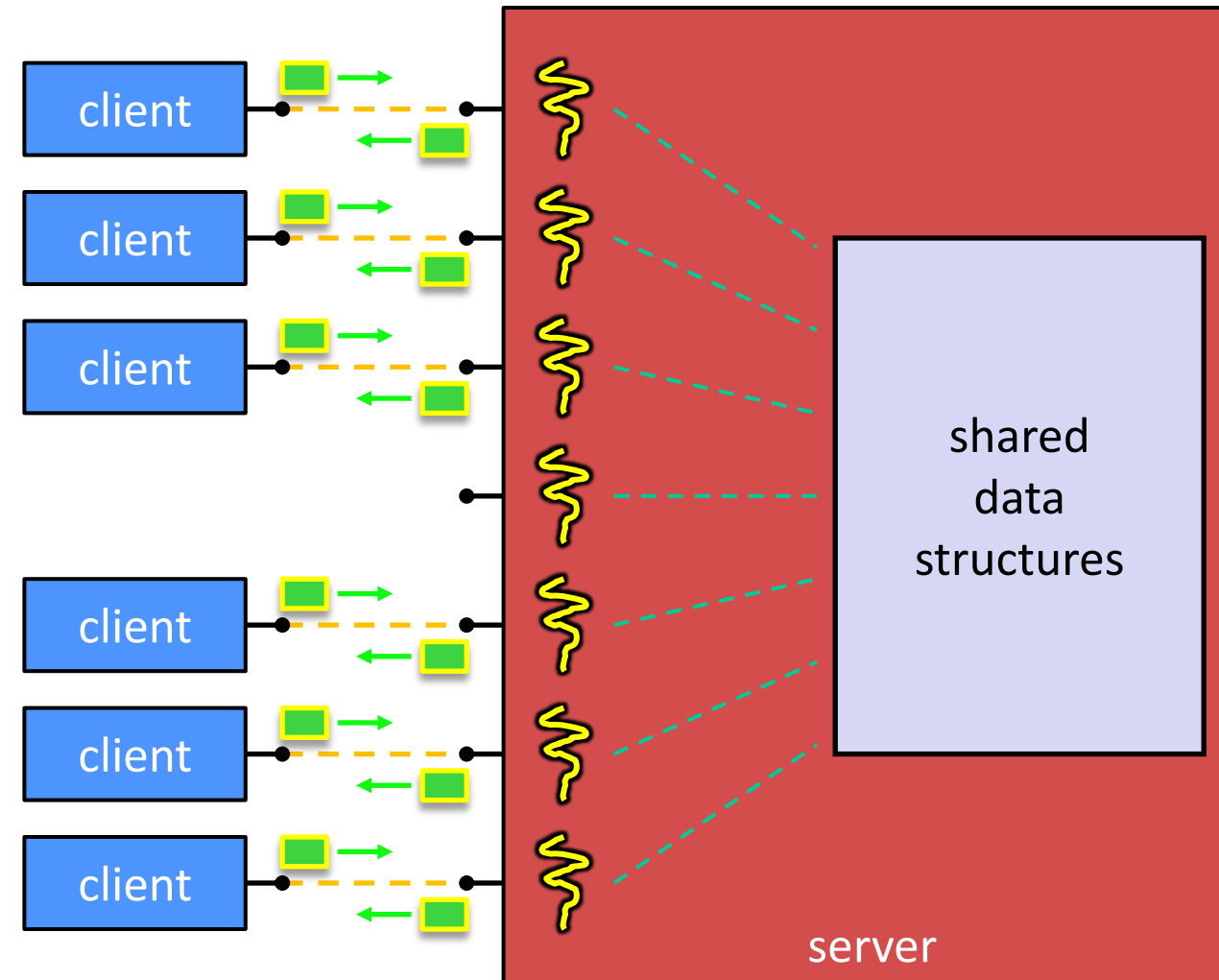
Multithreaded Server



Multithreaded Server



Multithreaded Server

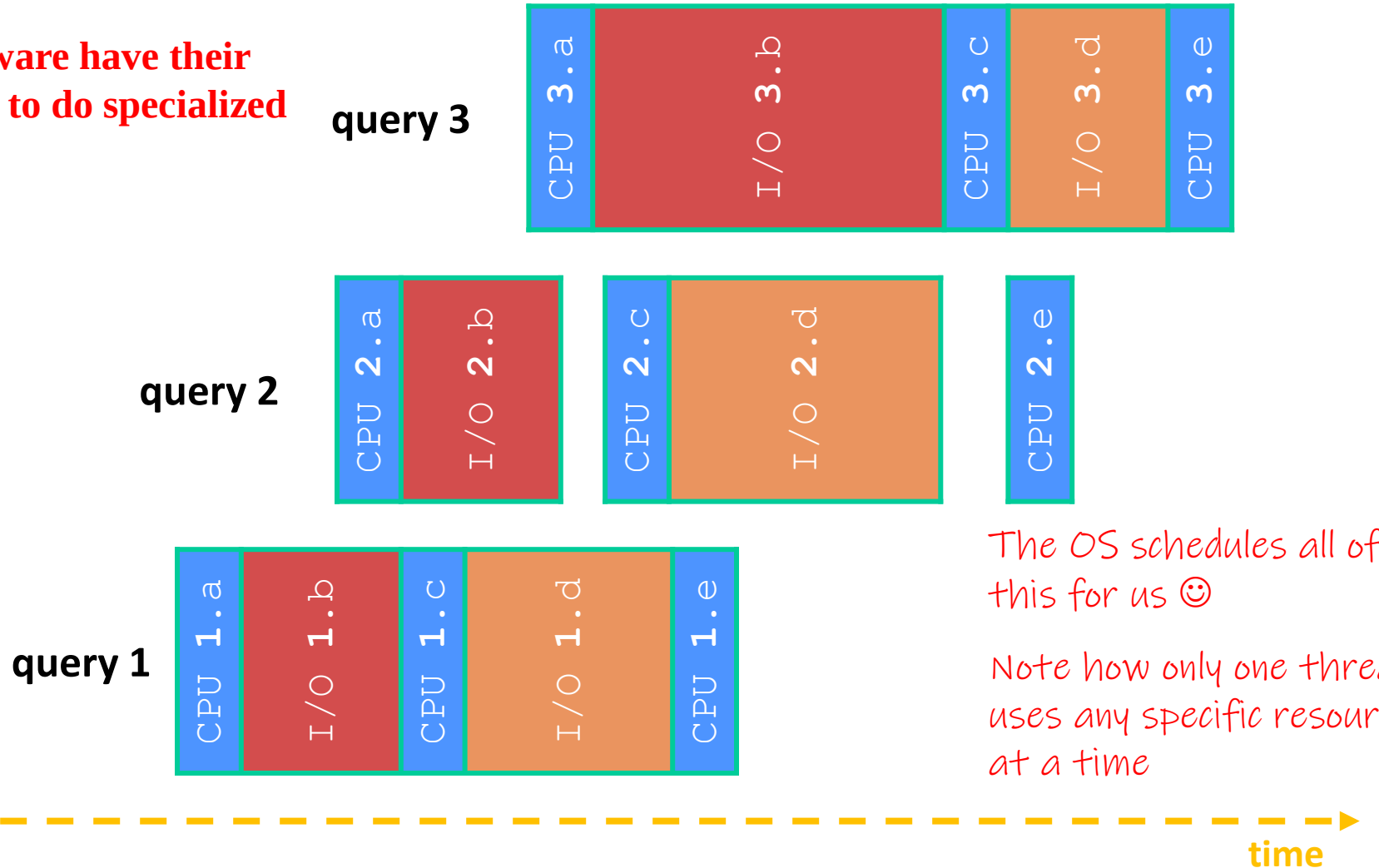


Multi-threaded Search Engine (Execution)

The CPU is the Central Processing Unit

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

*Running with 1 CPU



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

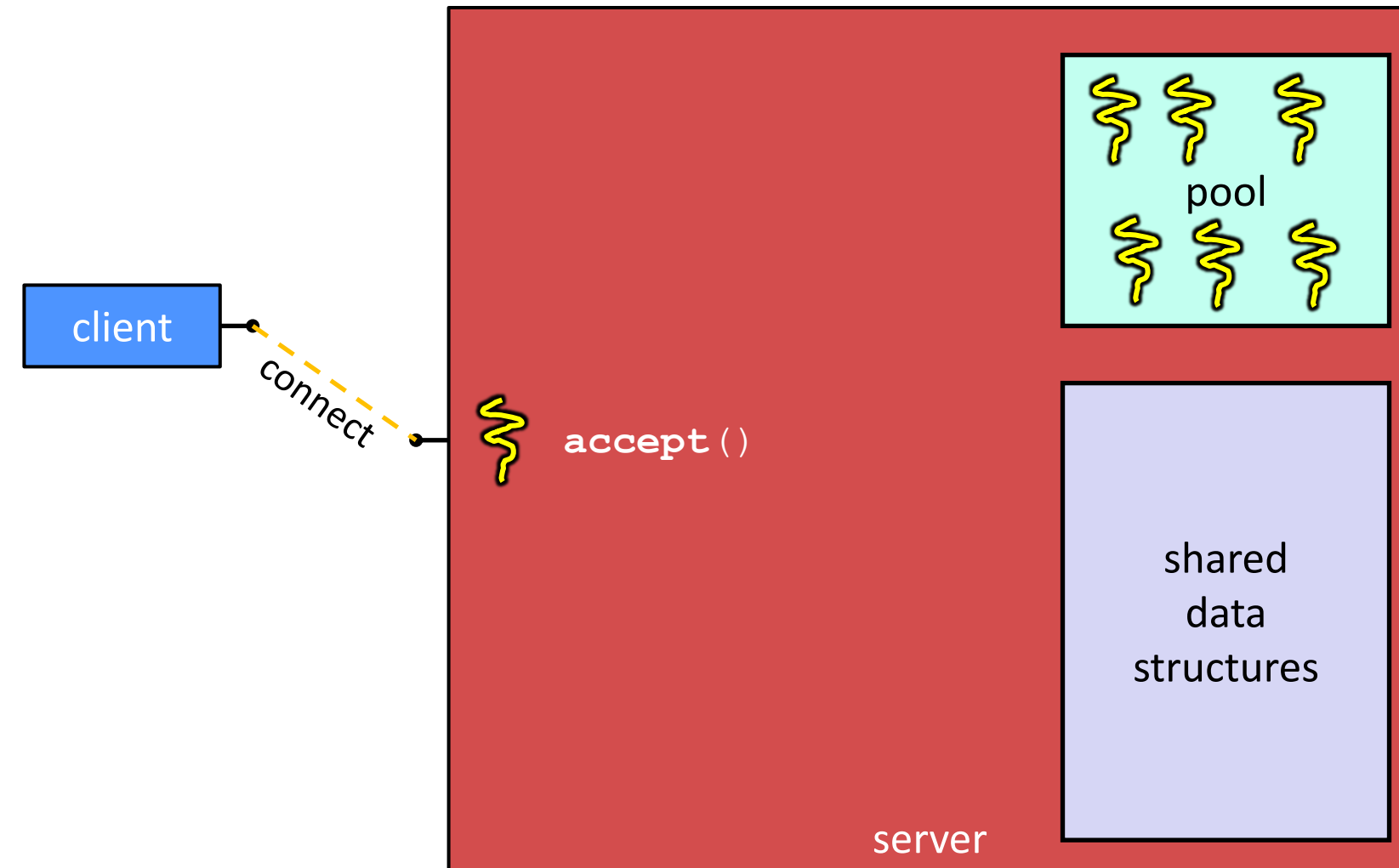
❖ Disadvantages:

- ❖ ■ 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

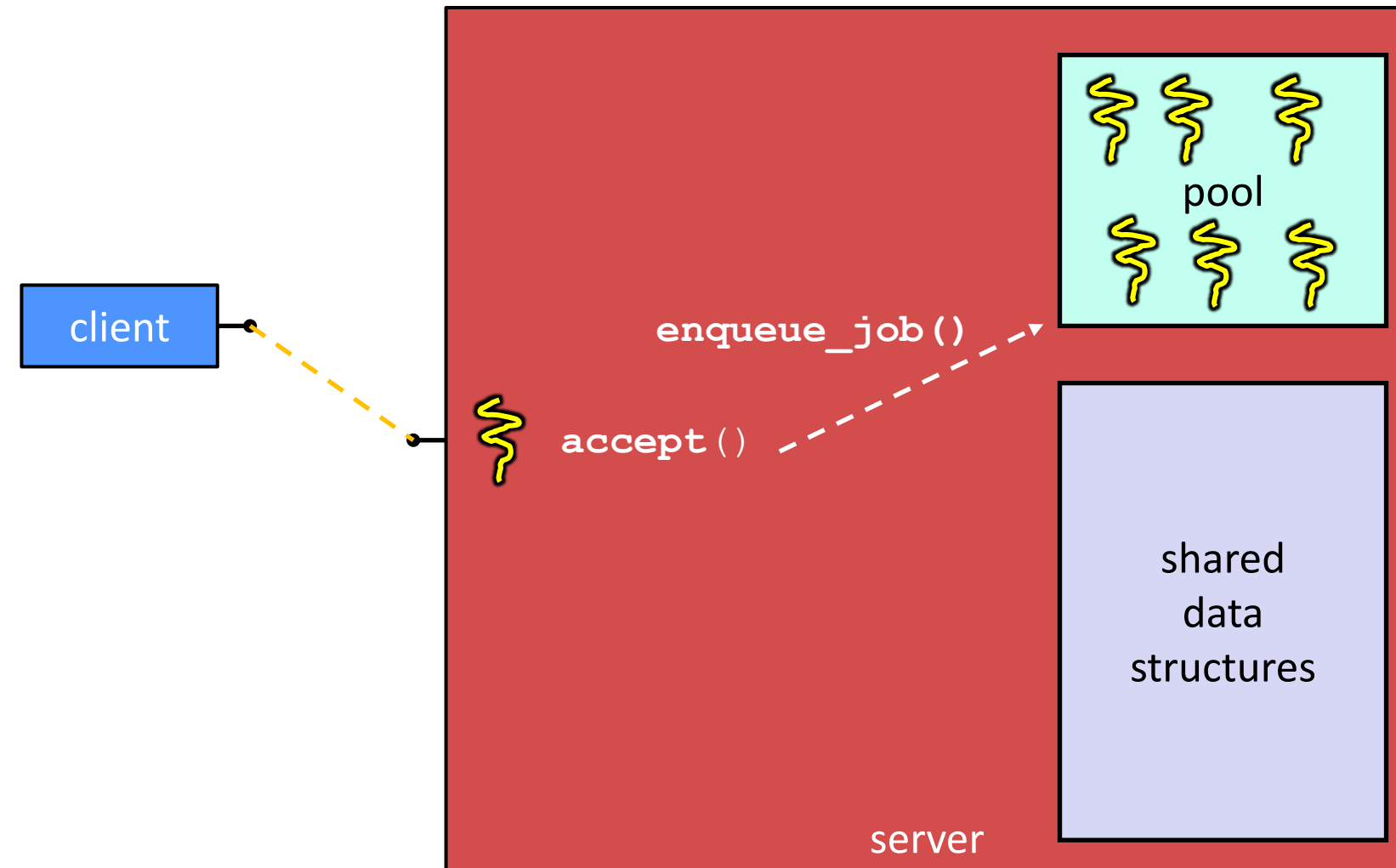
Limiting Overhead w/ Thread Pools

- ❖ 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

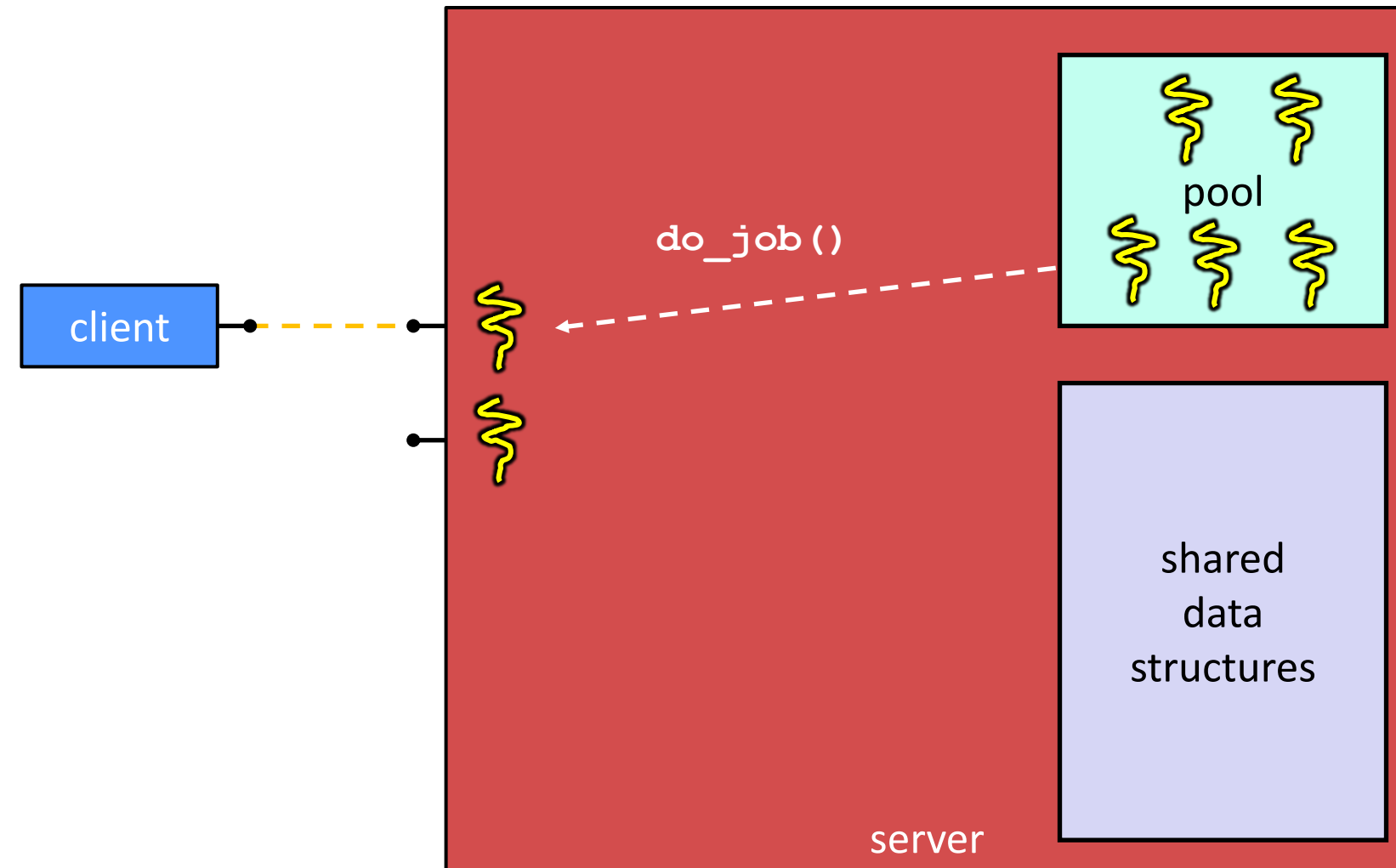
Multithreaded Server: Thread Pool



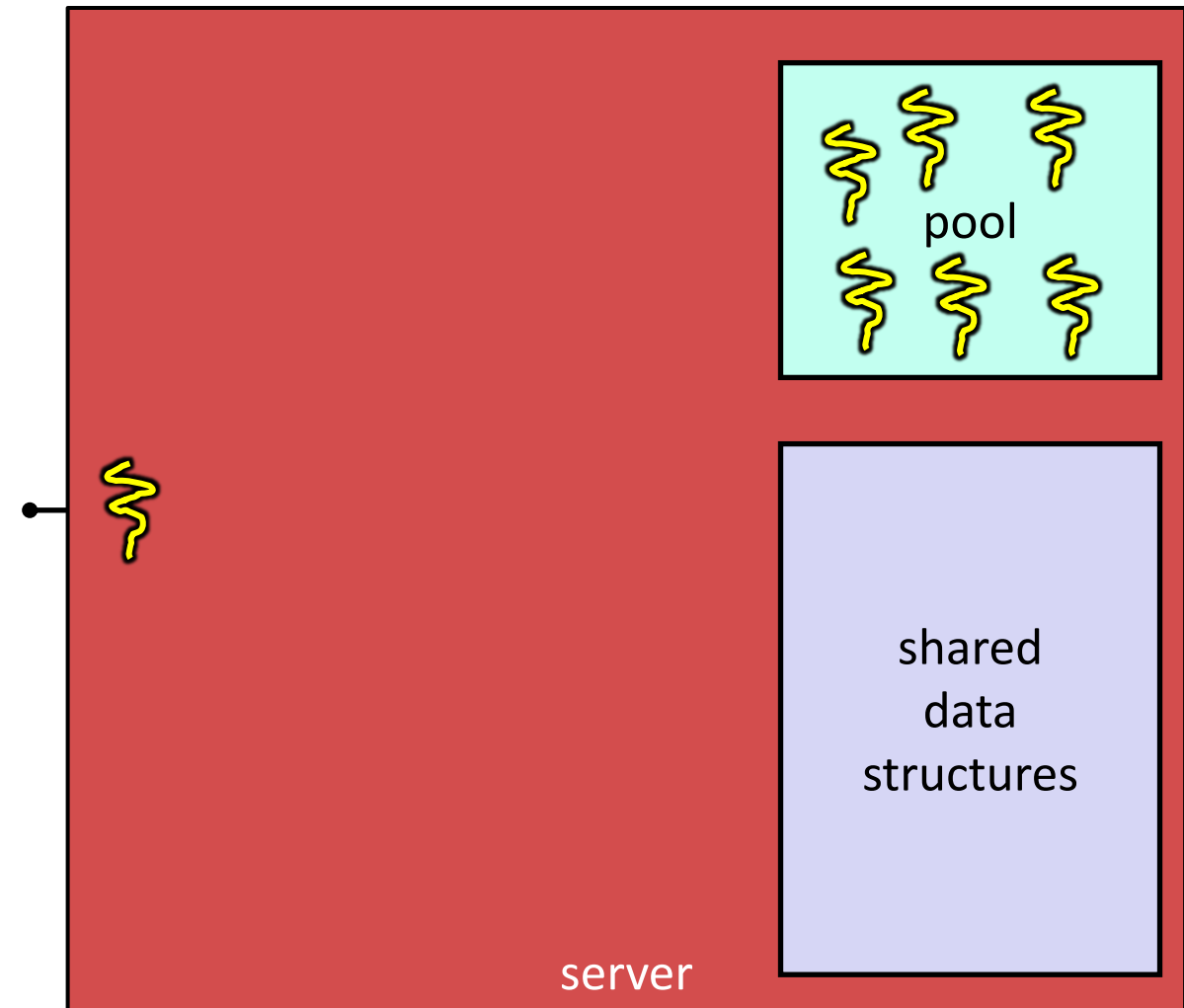
Multithreaded Server: Thread Pool



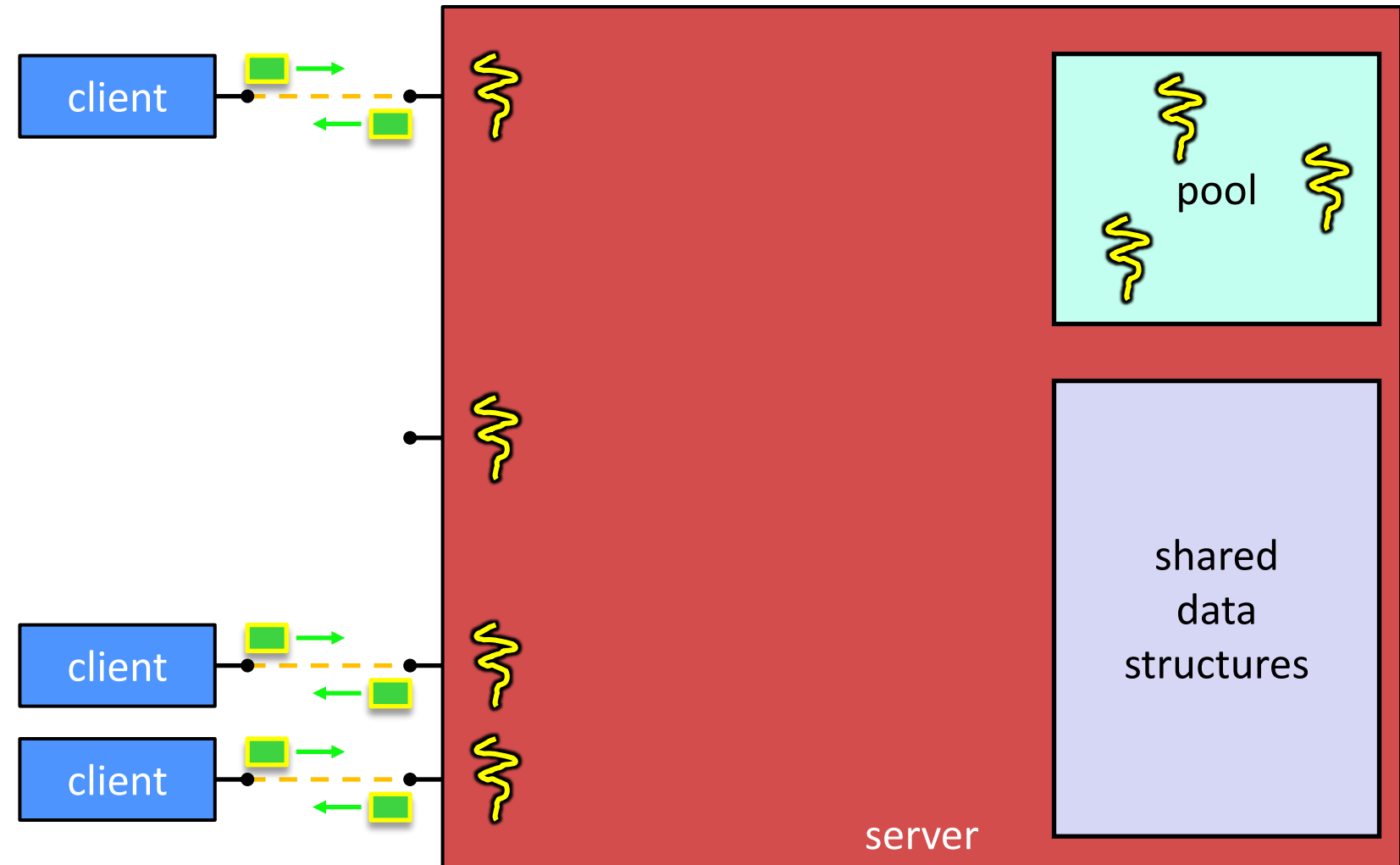
Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool



Multithreaded Server: Thread Pool



Lecture Outline

- ❖ Producer / Consumer
- ❖ Spinning & Condition Variables
- ❖ Benefits of Concurrency
- ❖ **Parallel Analysis**

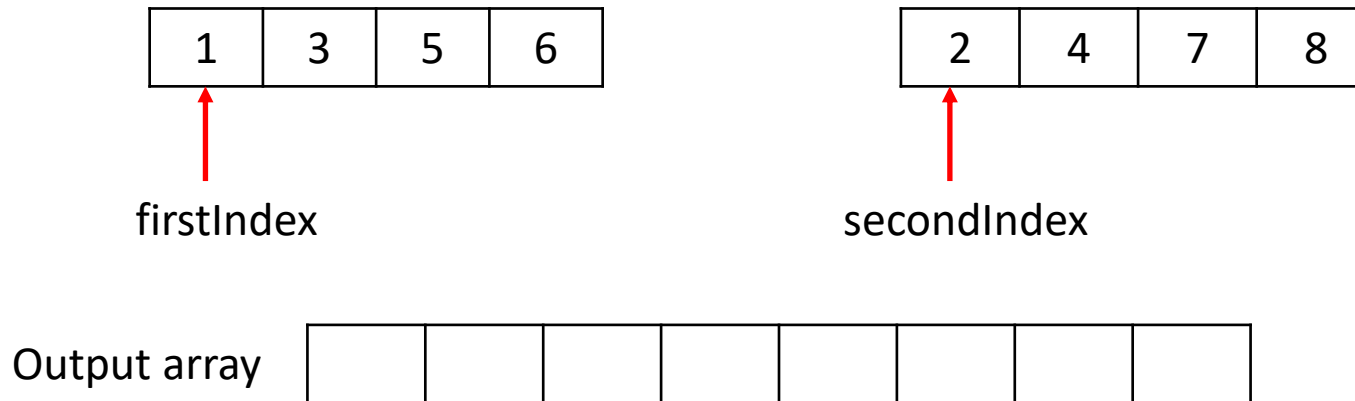
Parallel Algorithms

- ❖ One interesting applications of threads is for faster algorithms
- ❖ Common Example: Merge sort

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:



Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

firstIndex

2	4	7	8
---	---	---	---

secondIndex

Output array

1							
---	--	--	--	--	--	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

firstIndex

2	4	7	8
---	---	---	---

secondIndex

Output array

1	2						
---	---	--	--	--	--	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

firstIndex

2	4	7	8
---	---	---	---

secondIndex

Output array

1	2	3					
---	---	---	--	--	--	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

firstIndex

2	4	7	8
---	---	---	---

secondIndex

Output array

1	2	3	4				
---	---	---	---	--	--	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

firstIndex

2	4	7	8
---	---	---	---

secondIndex

Output array

1	2	3	4	5			
---	---	---	---	---	--	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

2	4	7	8
---	---	---	---

↑
firstIndex

↑
secondIndex

Output array

1	2	3	4	5	6		
---	---	---	---	---	---	--	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

2	4	7	8
---	---	---	---

↑
firstIndex

↑
secondIndex

Output array

1	2	3	4	5	6	7	
---	---	---	---	---	---	---	--

Merge Sort: Core Ideas

- ❖ It is easier to sort small arrays than big arrays
- ❖ It is quicker to merge two sorted arrays than sort an unsorted array

- Consider the two sorted arrays:

1	3	5	6
---	---	---	---

2	4	7	8
---	---	---	---

↑
firstIndex

↑
secondIndex

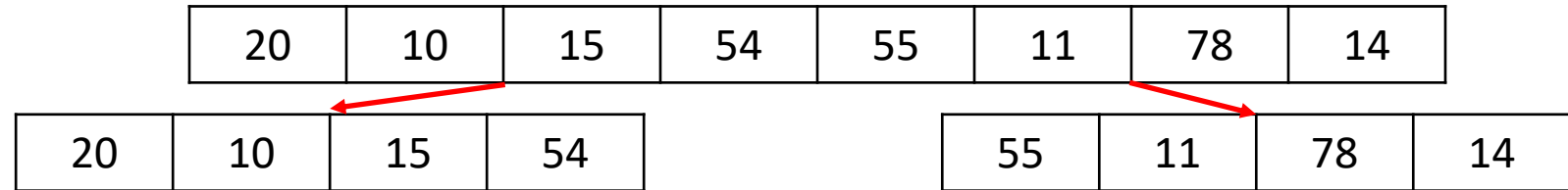
Output array

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

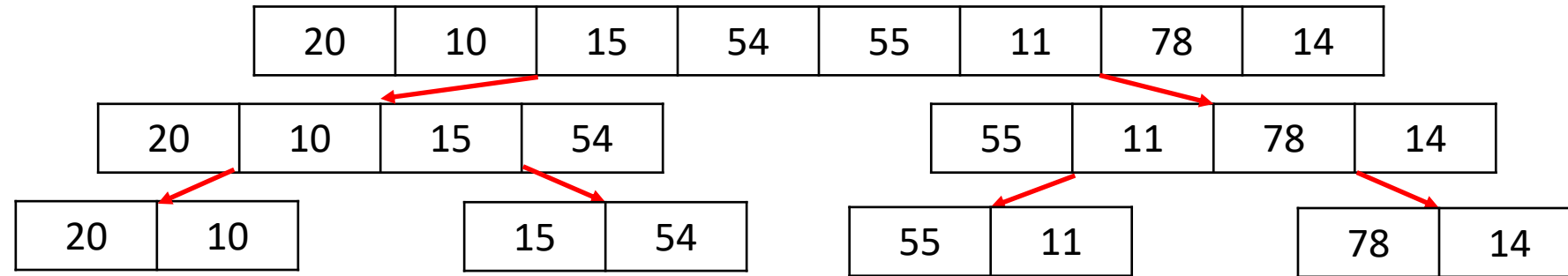
Merge Sort: High Level Example

20	10	15	54	55	11	78	14
----	----	----	----	----	----	----	----

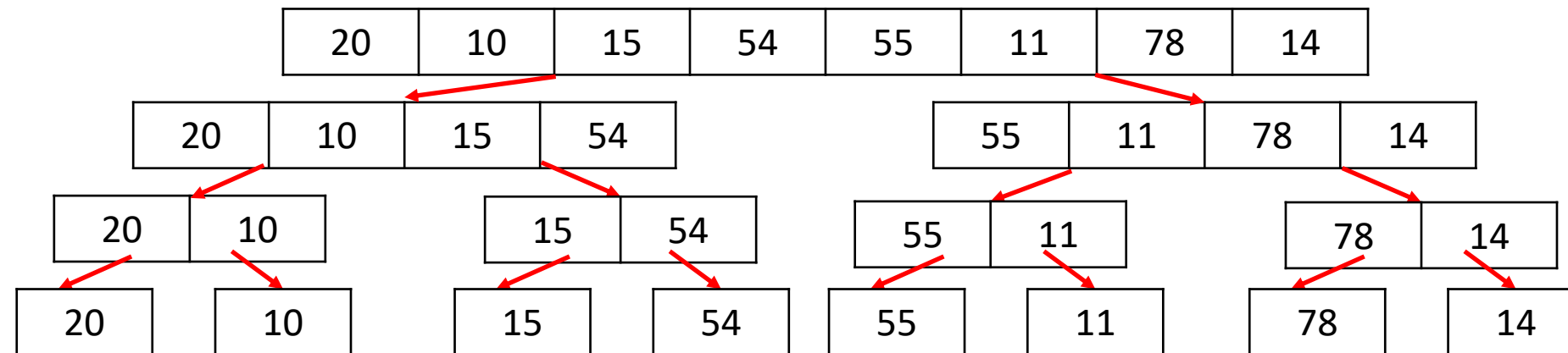
Merge Sort: High Level Example



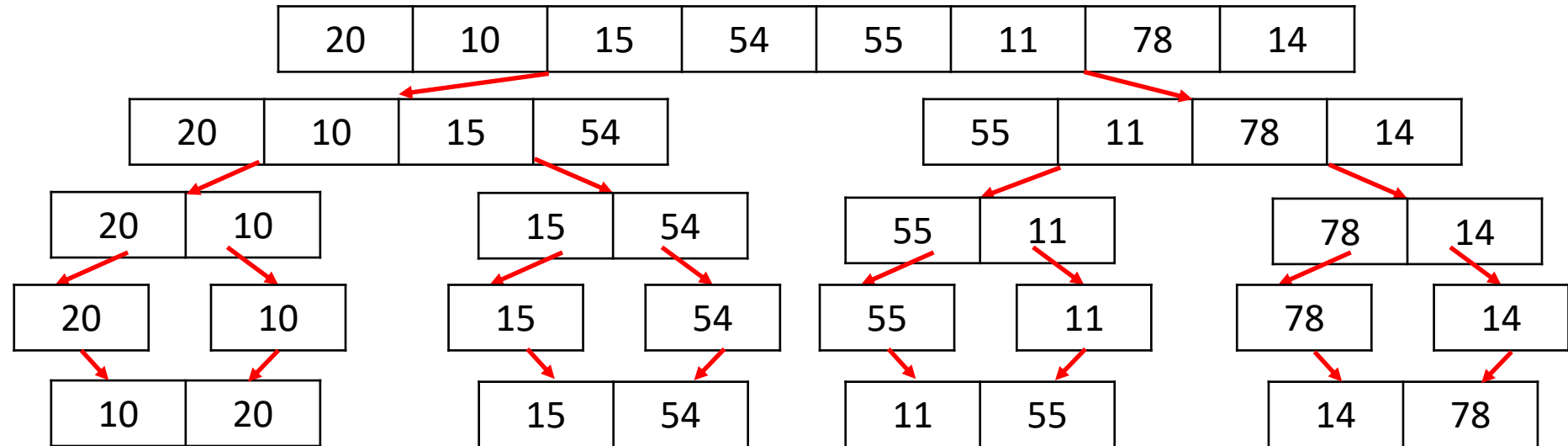
Merge Sort: High Level Example



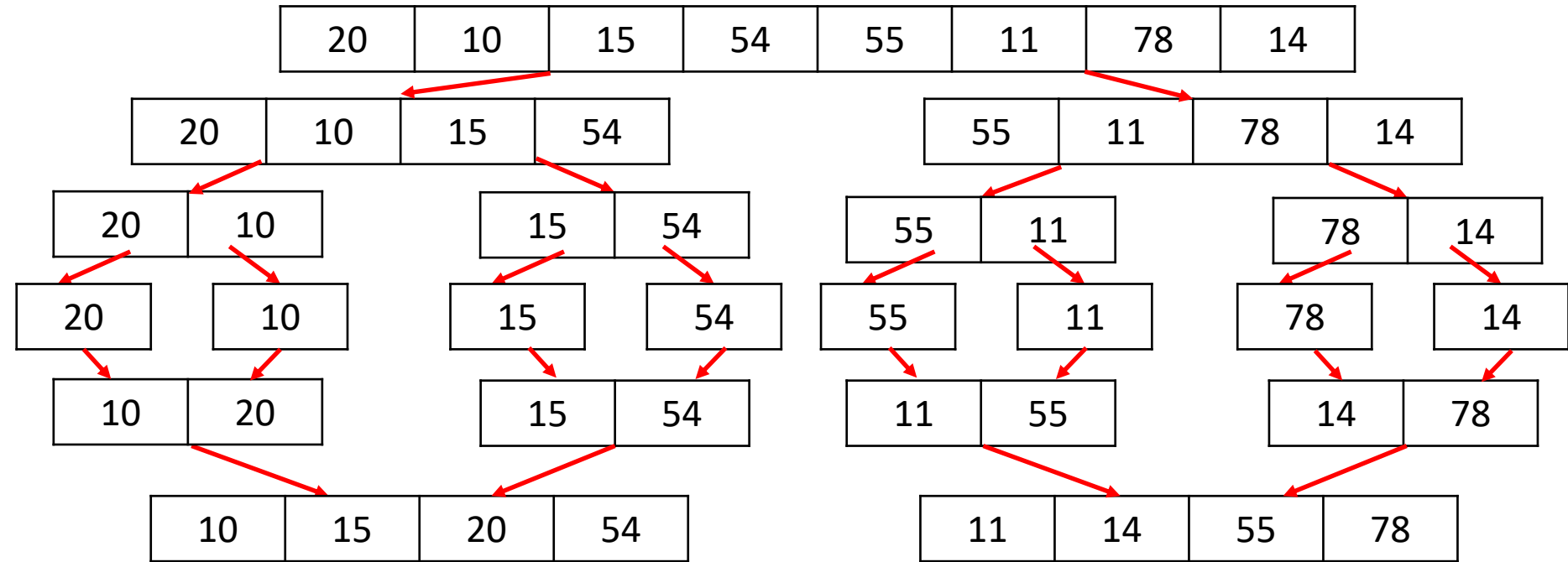
Merge Sort: High Level Example



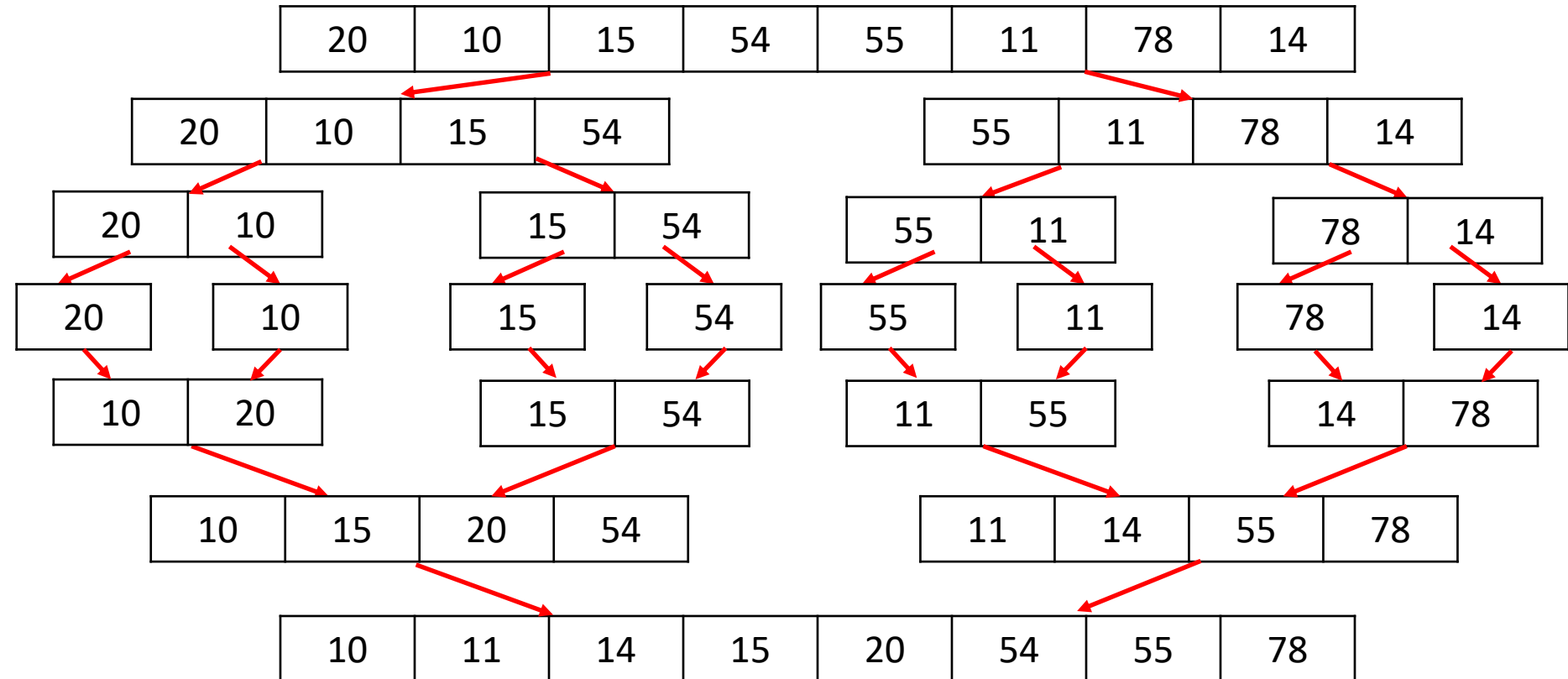
Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort: High Level Example



Merge Sort Algorithmic Analysis

- ❖ Algorithmic analysis of merge sort gets us to $O(n * \log(n))$ runtime.

```
void merge_sort(int[] arr, int lo, int hi) {  
    // lo high start at 0 and arr.length respectively  
    int mid = (lo + hi) / 2;  
    merge_sort(arr, lo, mid); // sort the bottom half  
    merge_sort(arr, mid, hi); // sort the upper half  
  
    // combine the upper and lower half into one sorted  
    // array containing all eles  
    merge(arr[lo : mid], arr[mid : hi]);  
}
```

- ❖ We recurse $\log_2(N)$ times, each recursive “layer” does $O(N)$ work

Merge Sort Algorithmic Analysis

- ❖ We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {  
    // lo high start at 0 and arr.length respectively  
    int mid = (lo + hi) / 2;  
  
    // sort bottom half in parallel  
    pthread_create(&merge_sort(arr, lo, mid));  
    merge_sort(arr, mid, hi); // sort the upper half  
  
    pthread_join(); // join the thread that did bottom half  
  
    // combine the upper and lower half into one sorted  
    // array containing all eles  
    merge(arr[lo : mid], arr[mid : hi]);  
}
```

- Now we are sorting both halves of the array in parallel!

❖ We can use threads to speed this up:

```
void merge_sort(int[] arr, int lo, int hi) {  
    // lo high start at 0 and arr.length respectively  
    int mid = (lo + hi) / 2;  
  
    // sort bottom half in parallel  
    pthread_create(&merge_sort(arr, lo, mid));  
    merge_sort(arr, mid, hi); // sort the upper half  
  
    pthread_join(); // join the thread that did bottom half  
  
    // combine the upper and lower half into one sorted  
    // array containing all eles  
    merge(arr[lo : mid], arr[mid : hi]);  
}
```

- Now we are sorting both halves of the array in parallel!
- How long does this take to run?
- How much work is being done?

Parallel Algos:

Will not test you on this

- ❖ We can define $T(n)$ to be the running time of our algorithm
- ❖ We can split up our work between two parts, the part done sequentially, and the part done in parallel
 - $T(n) = \text{sequential_part} + \text{parallel_part}$
 - $T(n) = O(n) \text{ *merging* } + T(n/2) \text{ *sort half the array* }$
 - This is a recursive definition
- ❖ If we start recurring...
 - $T(n) = O(n) + O(n/2) + T(n/4)$
 - $T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)$

Parallel Algos:

Will not test you on this

- ❖ If we start recurring...
 - $T(n) = O(n) + O(n/2) + T(n/4)$
 - $T(n) = O(n) + O(n/2) + O(n/4) + T(n/8)$
 - ...
 - Eventually we stop, there is a limit to the length of the array.
And we can say an array of size 1 is already sorted, so $T(1) = O(1)$
- ❖ This approximates to $T(n) = \sim 2 * O(n) = O(n)$
 - This parallel merge sort is $O(n)$, but there are further optimizations that can be done to reach $\sim O(\log(n))$
- ❖ There is a lot more to parallel algo analysis than just this, I am just giving you a sneak peek

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

- $T_p = (1 - S) + \frac{S}{P}$

- ❖ 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? ($s = 0.6666..$)

- $\frac{T_1}{T_p} = \frac{1}{1 - 0.6666 + \frac{0.6666}{100000}} = 2.9999 \text{ times faster than sequential}$

- ❖ What if it is 90% parallel? ($S = 0.9$):

- $\frac{T_1}{T_p} = \frac{1}{1 - 0.9 + \frac{0.9}{100000}} = 9.99 \text{ times faster than sequential}$

- ❖ What if it is 99% parallel? ($S = 0.99$):

- $\frac{T_1}{T_p} = \frac{1}{1 - 0.99 + \frac{0.99}{100000}} = 99.99 \text{ times faster than sequential}$

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.

Next Lecture

❖ Intro to networking 😊