CSE 374 Programming Concepts and Tools

Lecture 23 - Concurrency

This Week

Monday: C++ Smart Pointers

Wednesday: C++ Inheritance

Today: Concurrency

Today

Example: Building Search Engine

Why we need more than sequential code

Threads

Concurrency

Data Races

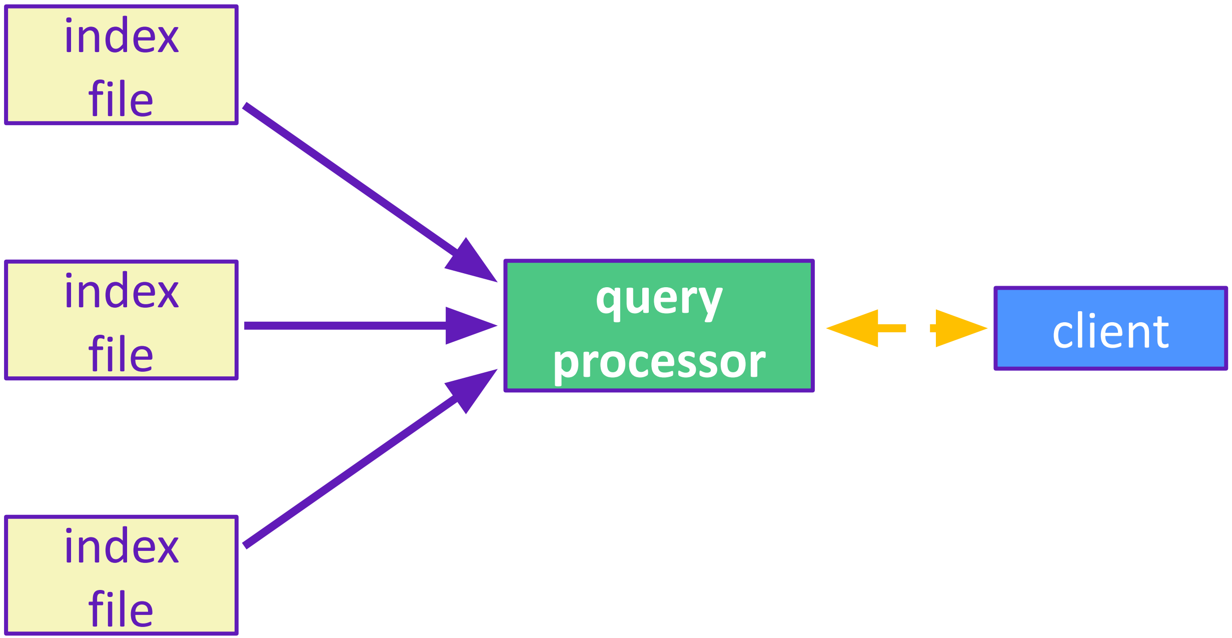
Honorable Mention: Thread Pools

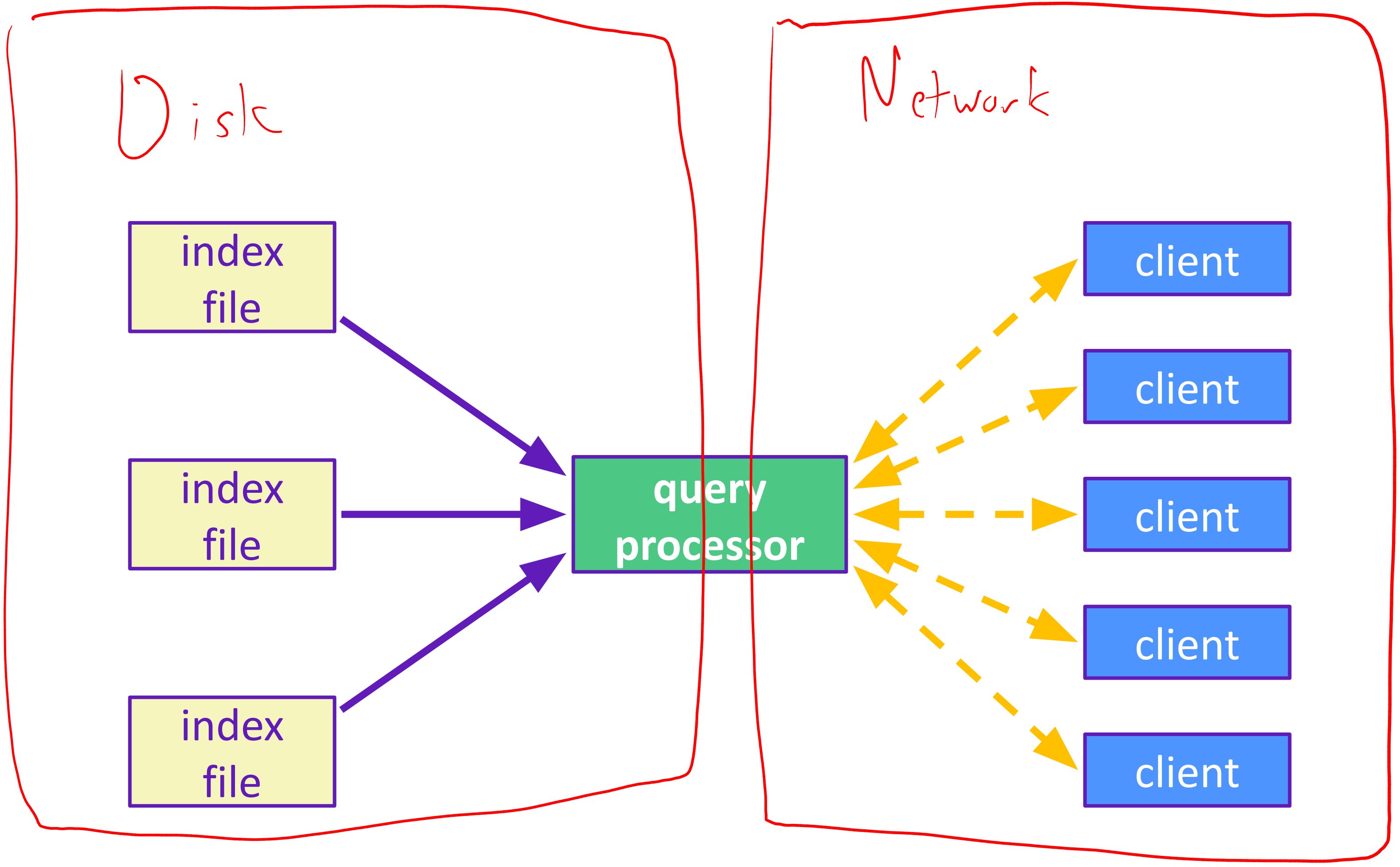
Building a Web Search Engine

We have:

- An index (similar to database)
  - A map from <*word*> to <*list of documents containing the word*>
  - This is probably *shared* 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




Disk
Network
index file
index file
index file
query processor
client
client
client
client
client

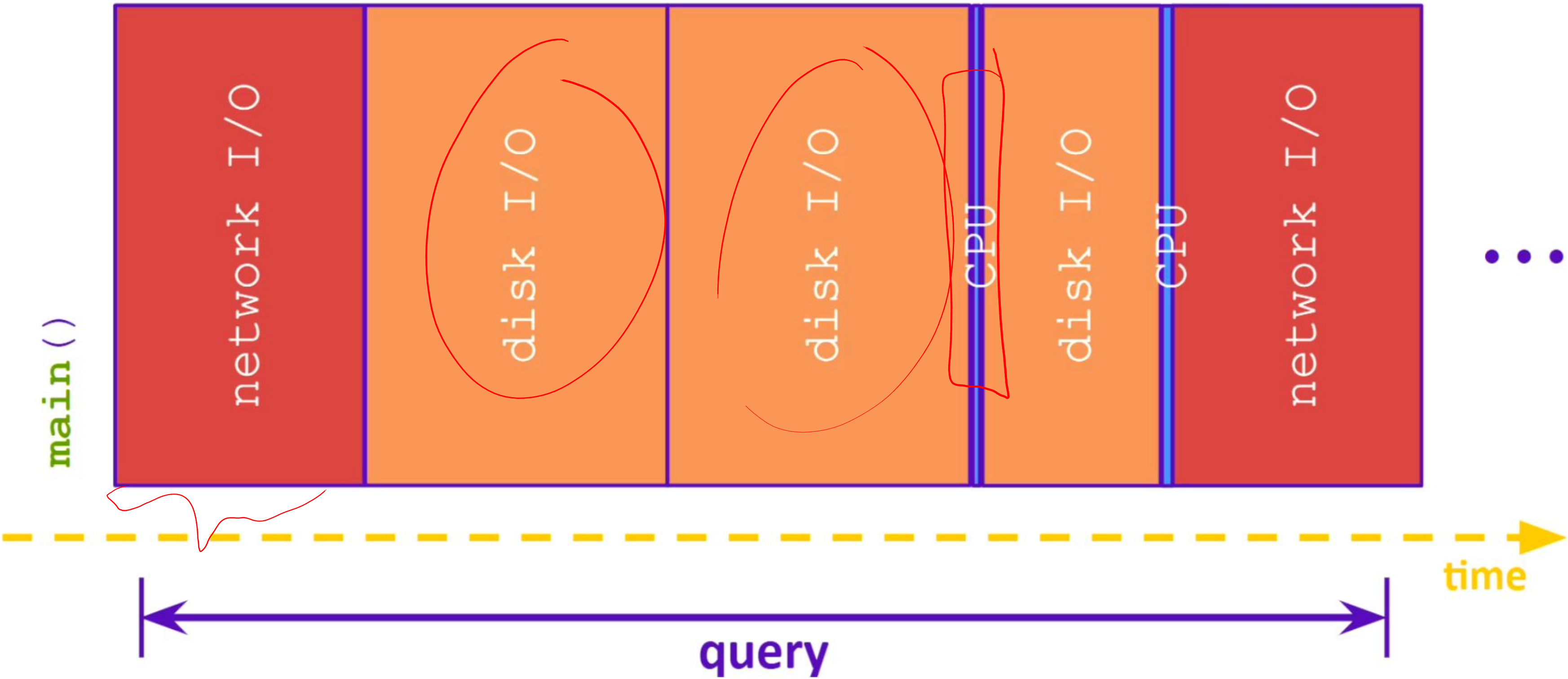
Sequential Code For Query Processing

Pseudocode:

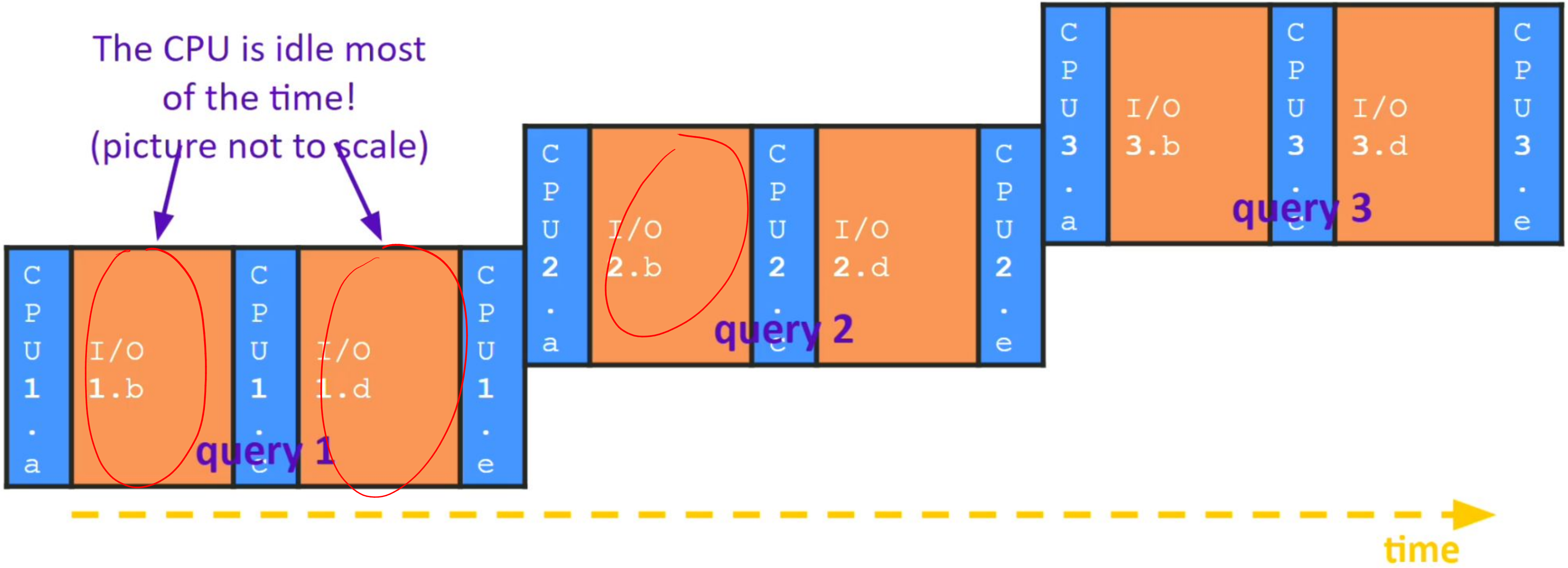
```
listen_fd = Listen(port);

while (1) {
  client_fd = accept(listen_fd);
  buf = read(client_fd);
  resp = ProcessQuery(buf);
  write(client_fd, resp);
  close(client_fd);
}
```

Execution Timeline: To Scale



Sequential Queries – Simplified



Sequential Is 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

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

Concurrency != Parallelism

- Concurrency is doing multiple tasks at a time
  - one person (or more) cooking multiple dishes at the same time
- Parallelism is executing multiple CPU instructions *simultaneously*
  - having *multiple* people (possibly cooking the same dish)

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

Threads

Review: Processes

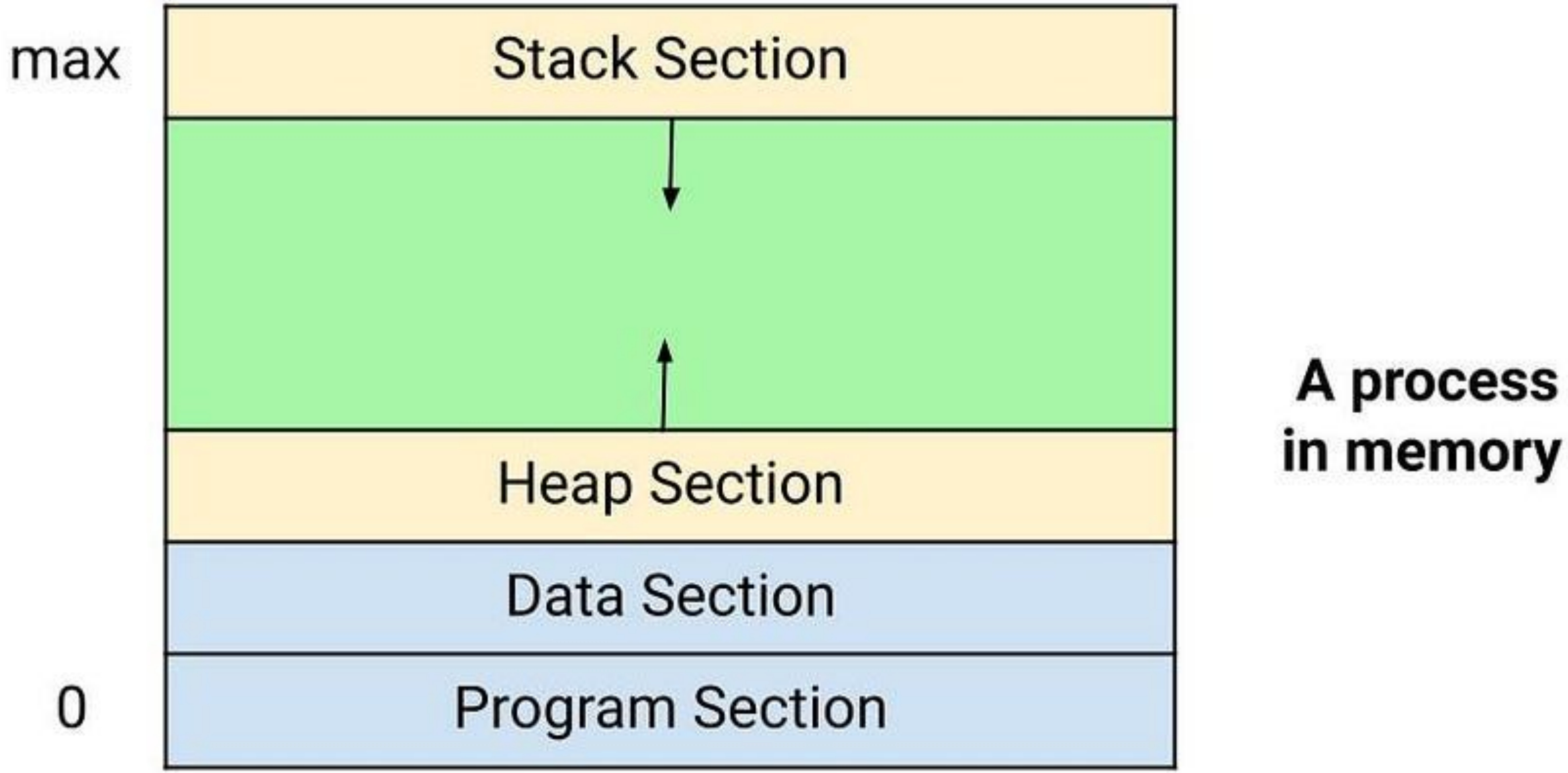
The components of a “process” are:

- Resources such as file descriptors and network sockets
- An address space

Different Processes have independent components:

- Most importantly: Isolated address spaces.
- Every process thinks it has access to the machine’s entire memory
- Memory Virtualization

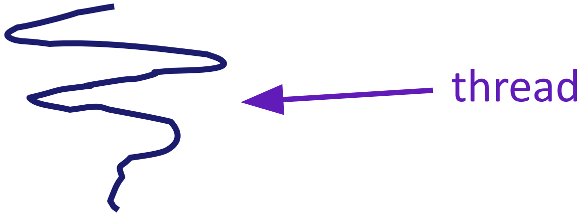
An address space of a process can hold stack(s) that distinguish different “threads” of execution.


max
0
Stack Section
Heap Section
Data Section
Program Section
A process
in memory

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
- Threads are contained within a process; **process > thread**



In most modern OS’s:

- Threads are the **unit of scheduling**

Once we have the idea of “threads” separate from “processes”, we arrive at a world where a process can have multiple threads.

Multi-threaded Search Engine (Pseudocode)

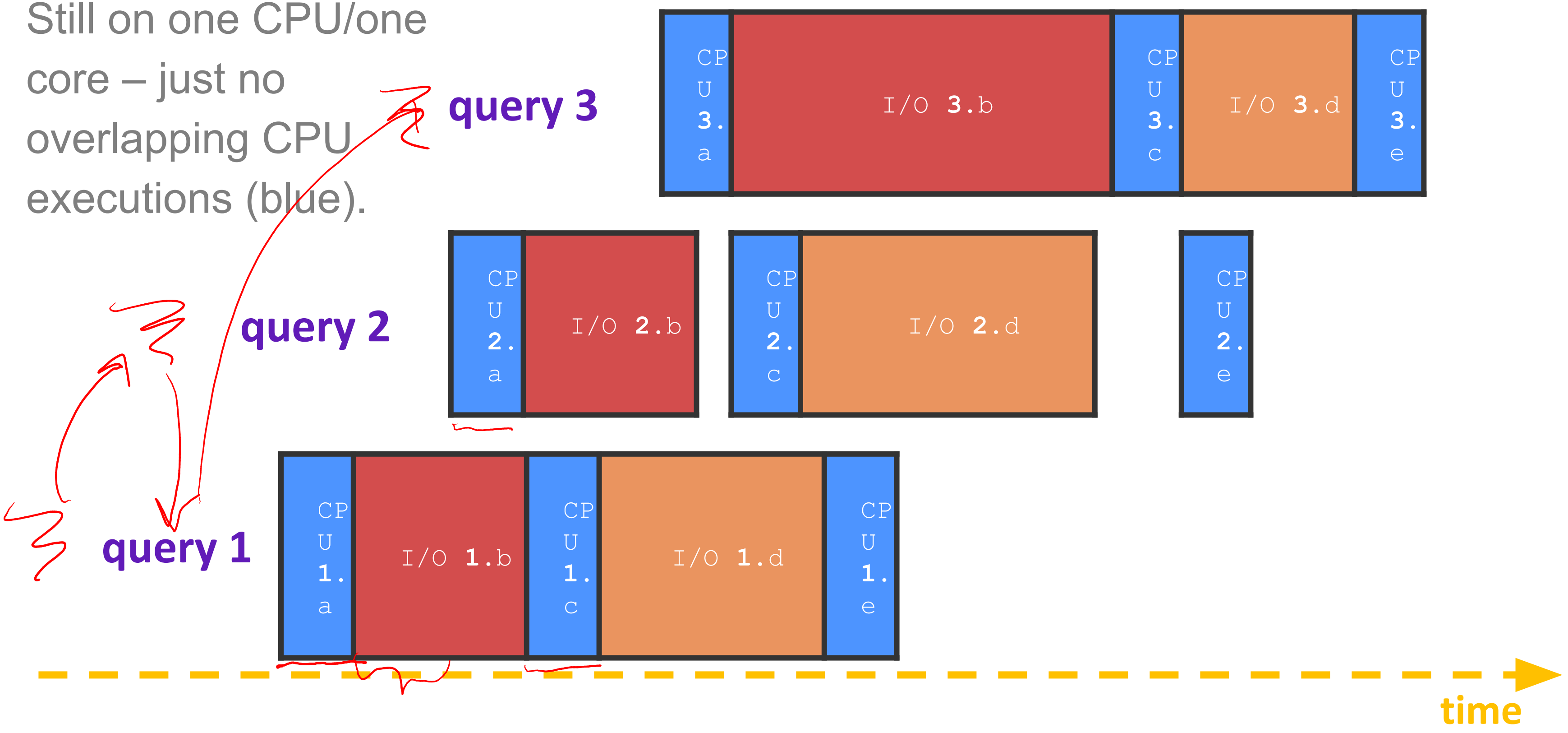
```
main() {
  while (1) {
    string query_words[] = GetNextQuery();
    CreateThread(ProcessQuery(query_words));
  }
}
```

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

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

Multi-threaded Search Engine (Execution)

Still on one CPU/one core – just no overlapping CPU executions (blue).



Why Threads?

Advantages:

- You (mostly) write sequential-looking code
- Threads can run in parallel if you have multiple CPUs/cores
    - AKA, on most modern computers
    - Definitely on Cancun :)

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:

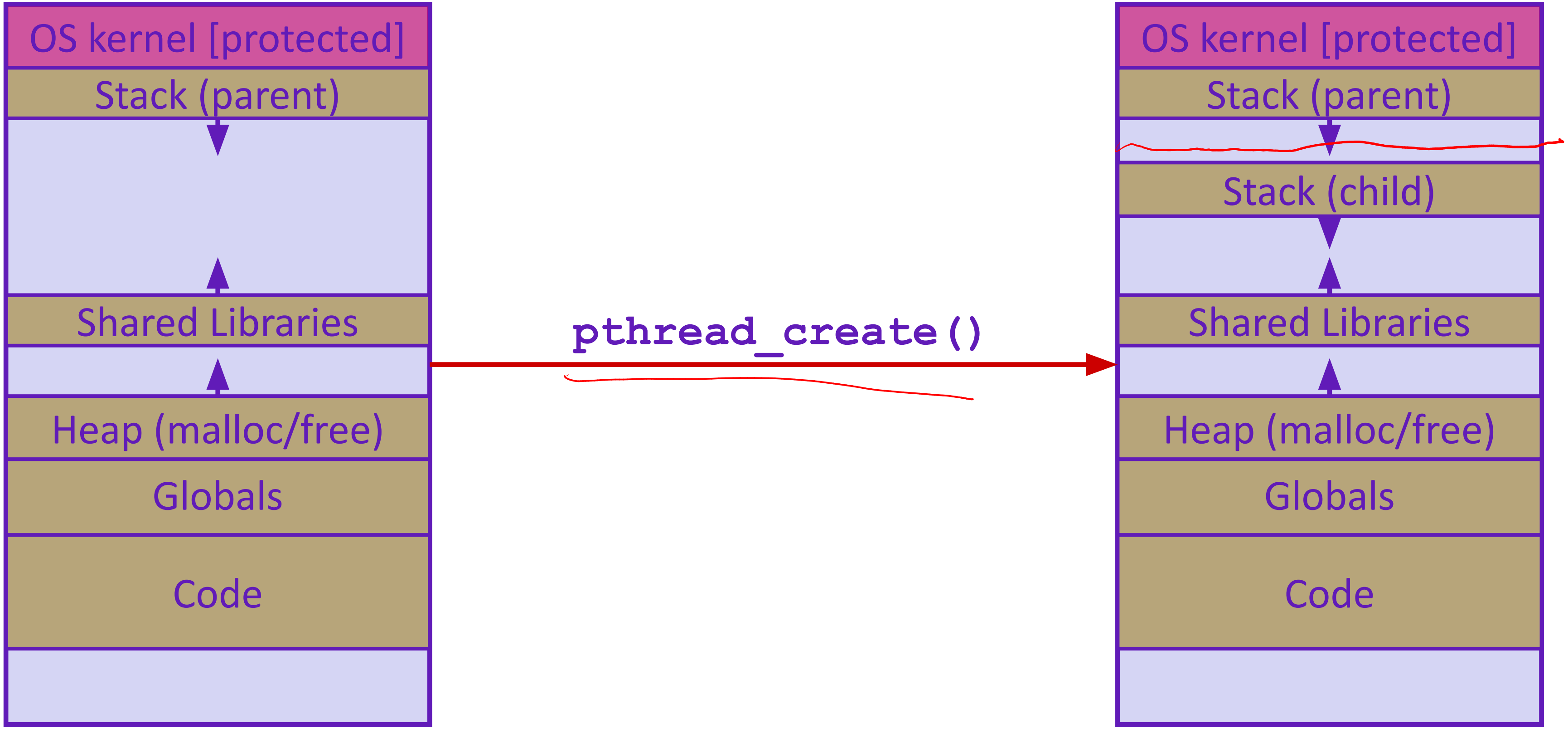
- A Process has a unique: address space, OS resources, and security attributes

- A Thread has a unique: stack, stack pointer, program counter, and registers

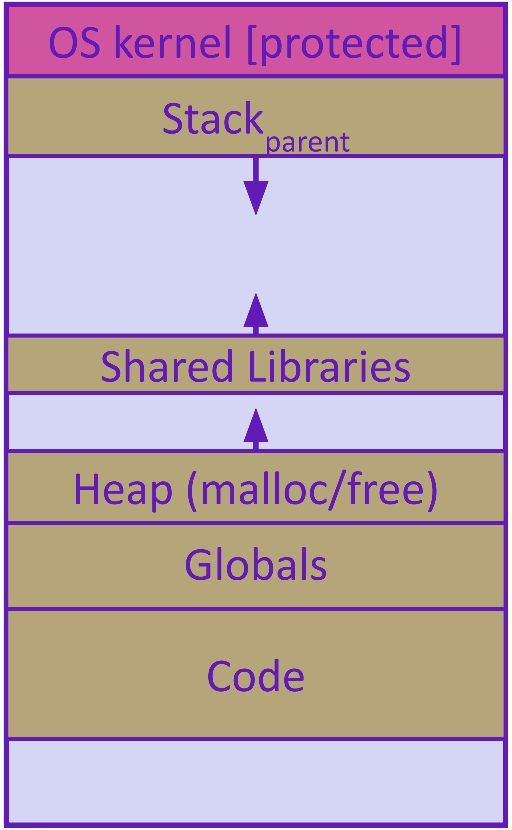
- Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Questions?

Threads vs. Processes



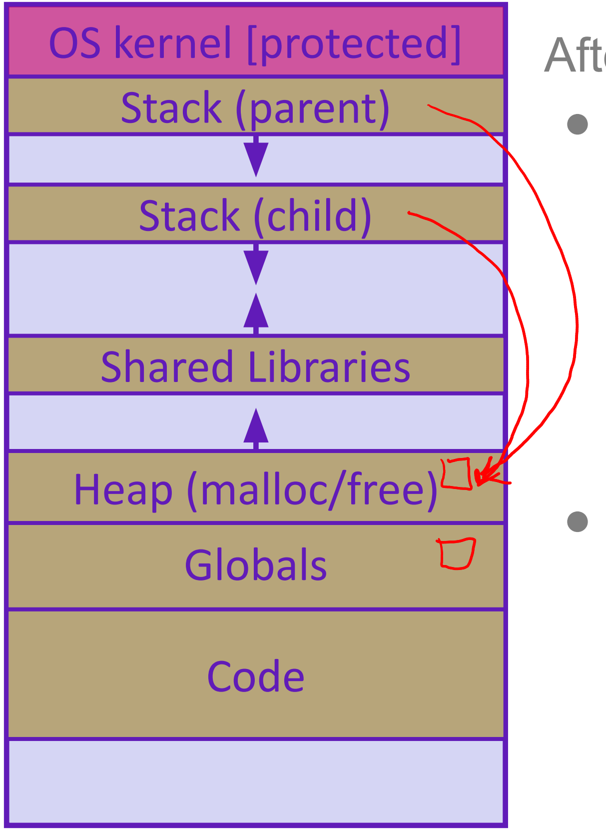
Single-Threaded Address Spaces



Before creating a thread

- One thread of execution running in the address space
    - One stack
- That main thread invokes a function to create a new thread
    - Typically `pthread_create()`

Multi-threaded Address Spaces



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 stack
- Both threads share the other segments (code, heap, globals)
  - They can cooperatively modify shared data

POSIX Threads (pthreads)

POSIX = Portable Operating System Interface

- A family of standards specified by IEEE for maintaining compatibility among operating systems.
- `ls`, `cd`, `grep`, `find`, etc. are all POSIX commands.

The POSIX APIs for dealing with threads

- Declared in `pthread.h`
  - Not part of the C/C++ language
- To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command
  - `gcc -g -Wall -std=c11 -pthread -o main main.c`

Creating and Terminating Threads

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

- Creates a new thread into `*thread`
- 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`**`()`

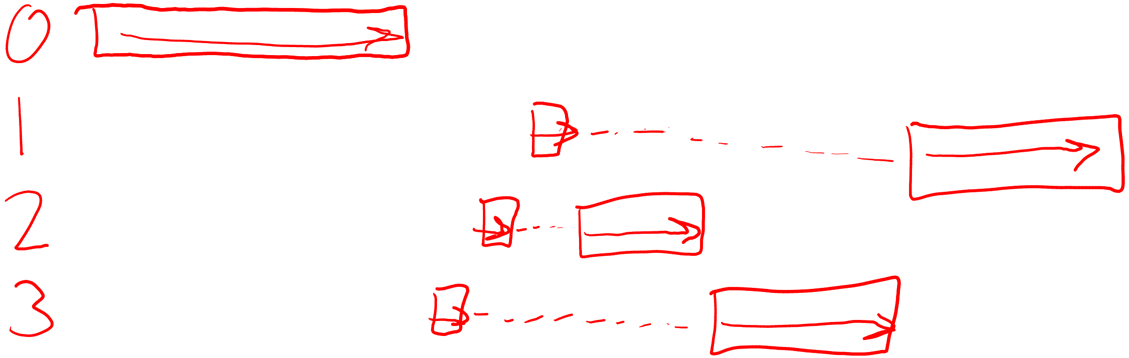
What To Do After Spawning Threads?

```
int pthread_join(pthread_t thread, void** retval);
```

- Waits for the thread specified by `thread` to terminate
- 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



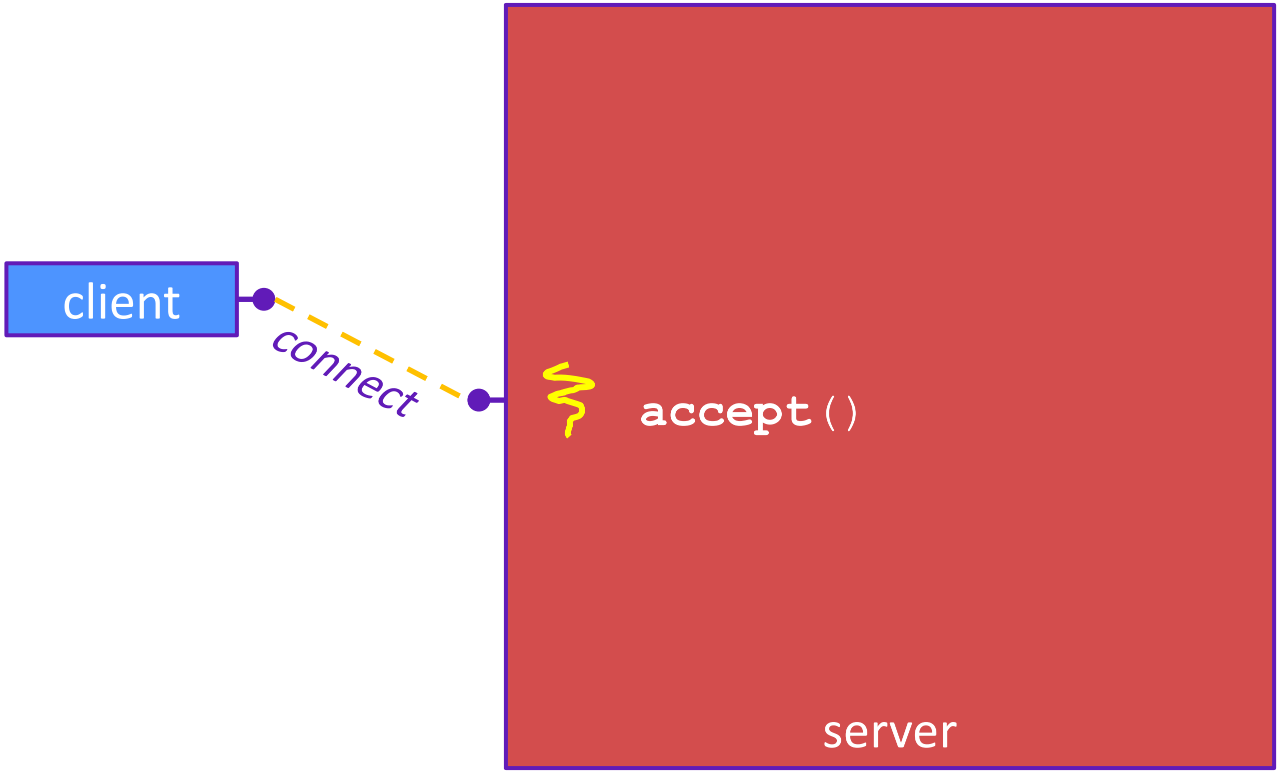
Demo: `cthreads`

Concurrent Server with Threads

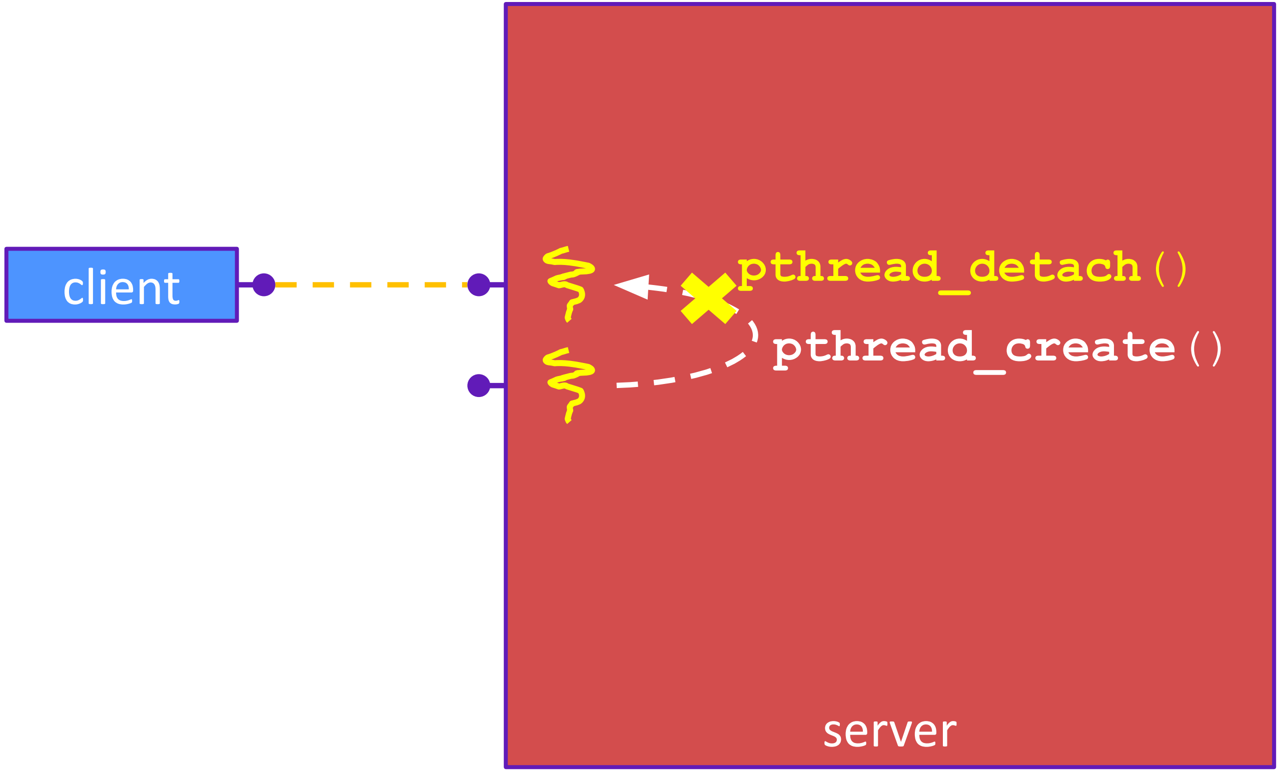
A single *process* handles all of the connections, but a parent *thread* dispatches (creates) a new thread to handle each connection

- The child thread handles the new connection and then exits when the connection terminates

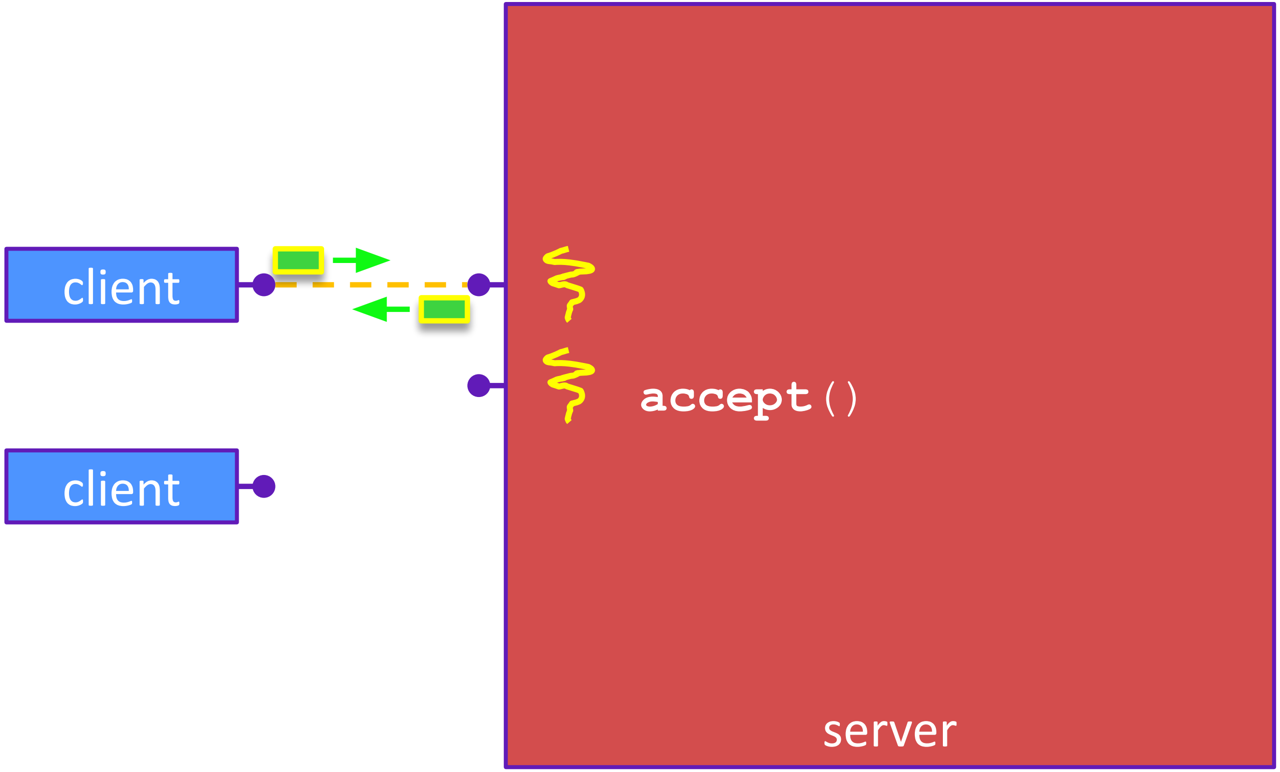
Multithreaded Server



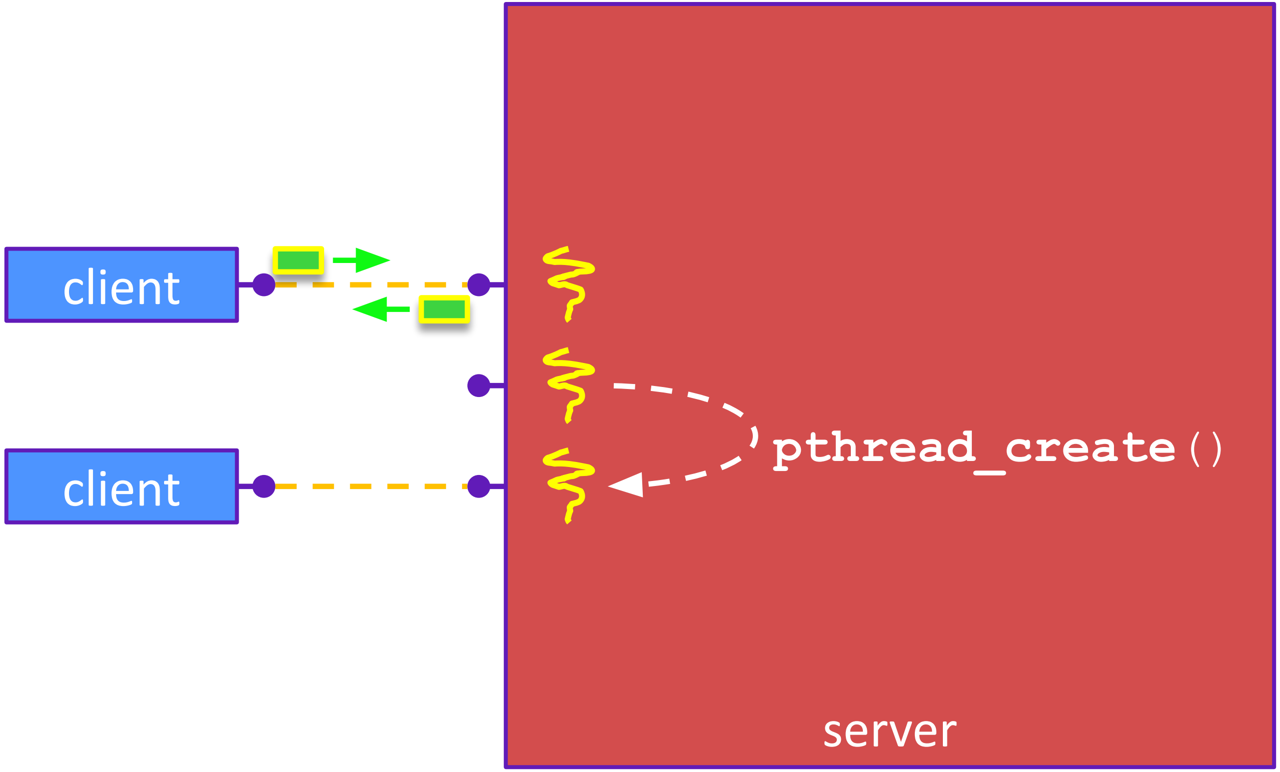
Multithreaded Server



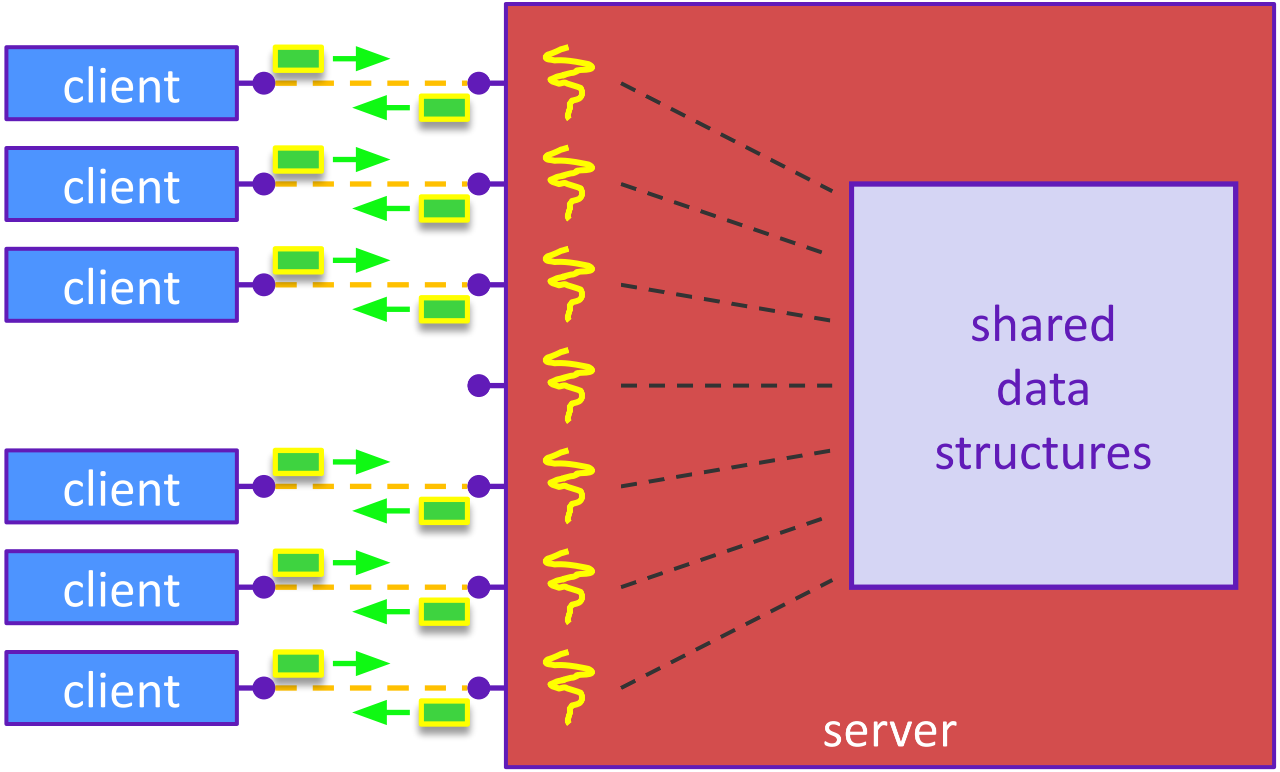
Multithreaded Server



Multithreaded Server



Multithreaded Server



Why Concurrent Threads?

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)
- Concurrent execution with good CPU and network utilization
    - Some overhead, but less than processes
- Shared-memory communication is possible

Disadvantages:

- Synchronization is complicated
- Shared fate within a process
    - One “rogue” thread can hurt you badly

Questions?

Data Races

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?)

```java
int x = 374;
... // code that doesn't change x
...
...
System.out.println(x);
...
...
x++;
...
...
System.out.println(x);
```

Data Race Example

If your fridge has no milk,
then go out and buy some more

- What could go wrong?

If you live alone:

If you live with a roommate:


```
if (!milk) {
  buy milk
}
```

Polling Time!

Please answer in the LP23 assignment on Gradescope!

Idea: leave a note!

Does this fix the problem?

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

- **No**, could end up with no milk
- **No**, could still buy multiple milk
- **Yes**, problem fixed
- **Yes**, but it defeats the purpose of having multiple threads

Poll Question Walkthrough

Alice	Bob
! note ! milk	
	! note ! milk Leave note Buy milk Remove note
Leave note Buy milk Remove note	

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

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!

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 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

❖ Pseudocode:

```
// non-critical code

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

// non-critical code
```

loop/idle if locked

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

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

```
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 this compare to sequential code?

- Likely *slower* – only 1 thread can increment at a time, but have to deal with checking the lock and switching between threads
- One possible fix: each thread increments a local variable and then adds its value (atomically) to the shared variable at the end

Demo: `total` and `total_locking`

C++11 Threads

C++11 added threads and concurrency to its libraries

- `<thread>` – thread objects
- `<mutex>` – locks to handle critical sections
- `<condition_variable>` – used to block objects until notified to resume
- `<atomic>` – indivisible, atomic operations
- `<future>` – asynchronous access to data
- These might be built on top of `<pthread.h>`, but also might not be

Definitely use in C++11 code if local conventions allow, but pthreads will be around for a long, long time

Questions?

Assignment Reminders

Exam 3 is on Friday, 10:50-12:50 in PCAR 291

- Use 26sp exam to study, disregard Misc. questions at the end
- Exam 3 grades will be released by Sunday night
- Regrade requests open until Monday night

HW 5 is due tonight, but late deadline has been extended to Sunday @ 11:59 PM

HW 6 (Extra Credit) can be submitted until Monday @ 11:59 PM

Bonus: Deadlock Game!