

Concurrency: Threads

CSE 333 Spring 2022

Instructor: Hal Perkins

Teaching Assistants:

Esau Abraham

Nour Ayad

Ramya Challa

Cleo Chen

Sanjana Chintalapati

Dylan Hartono

Kenzie Mihardja

Brenden Page

Aakash bin Srazali

Justin Tysdal

Julia Wang

Timmy Yang

Administrivia (1)

- ❖ Last exercise ex17 out now, due ~~Monday~~ Wednesday
 - Concurrency using pthreads
- ❖ No class Monday – Memorial Day holiday
 - No office hours this Sunday night – will be Monday night instead

Administrivia (2)

- ❖ hw4 due Thursday night next week
 - <panic>If you haven't started yet</panic>
 - Usual late days (max 2) available if you have any left
 - Mime types (in server query replies): hw4 server only needs to have ones that match the files that it will actually send (including pictures)
 - Remember – don't modify Makefiles or header files
 - Please be careful about inappropriate copying of solution code from others or found on the web. (Let's not have problems this late in the quarter)

Some Common hw4 Bugs

- ❖ Your server works, but is really, really slow
 - Check the 2nd argument to the `QueryProcessor` constructor
- ❖ Funny things happen after the first request
 - Make sure you're not destroying the `HTTPConnection` object too early (*e.g.* falling out of scope in a while loop)
- ❖ Server crashes on a blank request
 - Make sure that you handle the case that `read()` (or `WrappedRead()`) returns `0`

Previously...

- ❖ We implemented a search server but it was sequential
 - Processes requests one at a time regardless of client delays
 - Terrible performance, resource utilization

- ❖ Servers should be concurrent
 - Different ways to process multiple queries simultaneously:
 - Issue multiple I/O requests simultaneously
 - Overlap the I/O of one request with computation of another
 - Utilize multiple CPUs or cores
 - Mix and match as desired

Outline (next two lectures)

- ❖ We'll look at different `searchserver` implementations
 - Sequential
 - Concurrent via dispatching threads – `pthread_create()`
 - Concurrent via forking processes – `fork()`
 - Concurrent via non-blocking, event-driven I/O – `select()`
 - We won't get to this 😞

- ❖ Reference: *Computer Systems: A Programmer's Perspective*, Chapter 12 (CSE 351 book)

Sequential

❖ 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);  
}
```

❖ See [searchserver_sequential/](#)

Whither Sequential?

❖ Advantages:

- Super(?) simple to build/write

❖ Disadvantages:

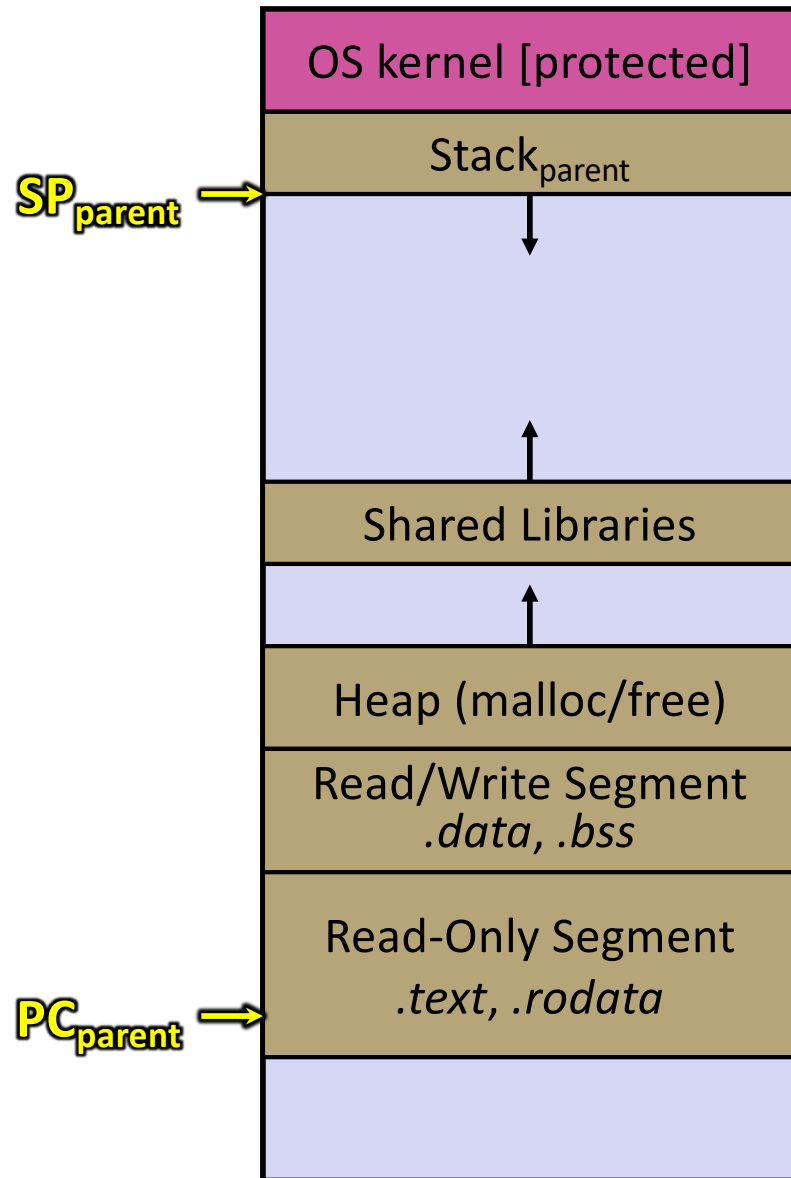
- Incredibly poor performance
 - One slow client will cause *all* others to block
 - Poor utilization of resources (CPU, network, disk)



Threads

- ❖ Threads are like lightweight processes
 - They execute concurrently like processes
 - Multiple threads can run simultaneously on multiple CPUs/cores
 - Unlike processes, threads cohabit 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

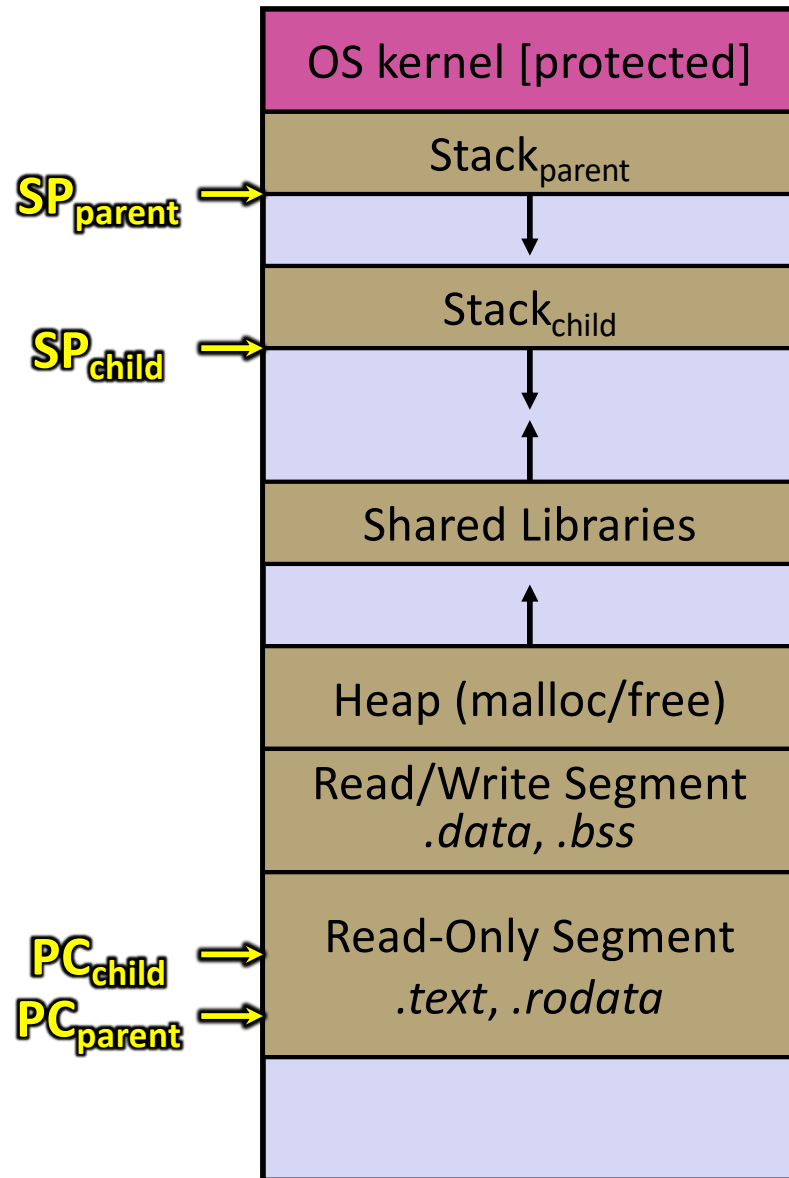
Threads and Address Spaces



❖ 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()`

Threads and 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 PC, SP
- Both threads share the other segments (code, heap, globals)
 - They can cooperatively modify shared data

pthread Threads

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

```
❖ int pthread_detach (pthread_t thread);
```

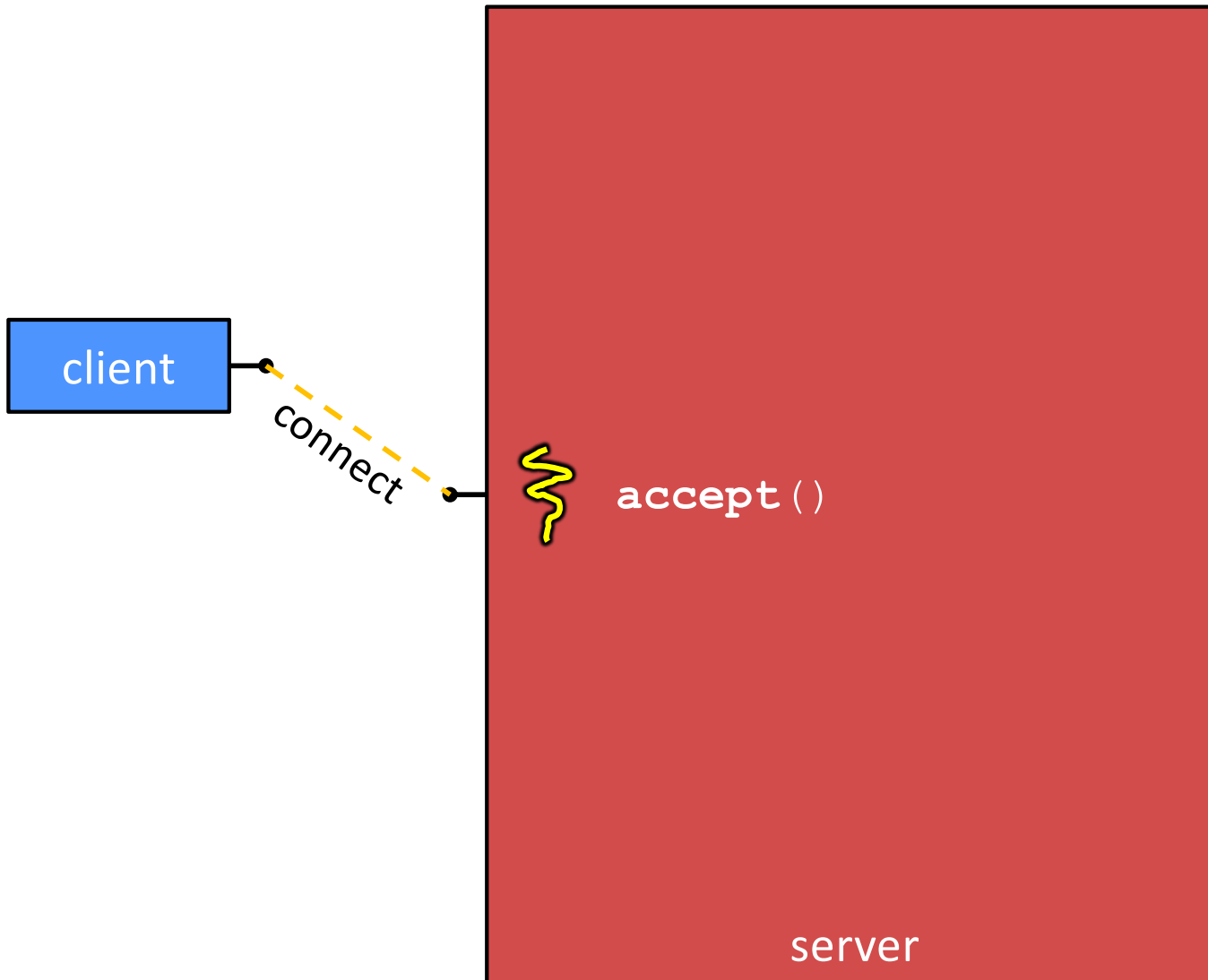
```
❖ int pthread_join (pthread_t thread,
    void** retval);
```

```
❖ See thread_example.cc
```

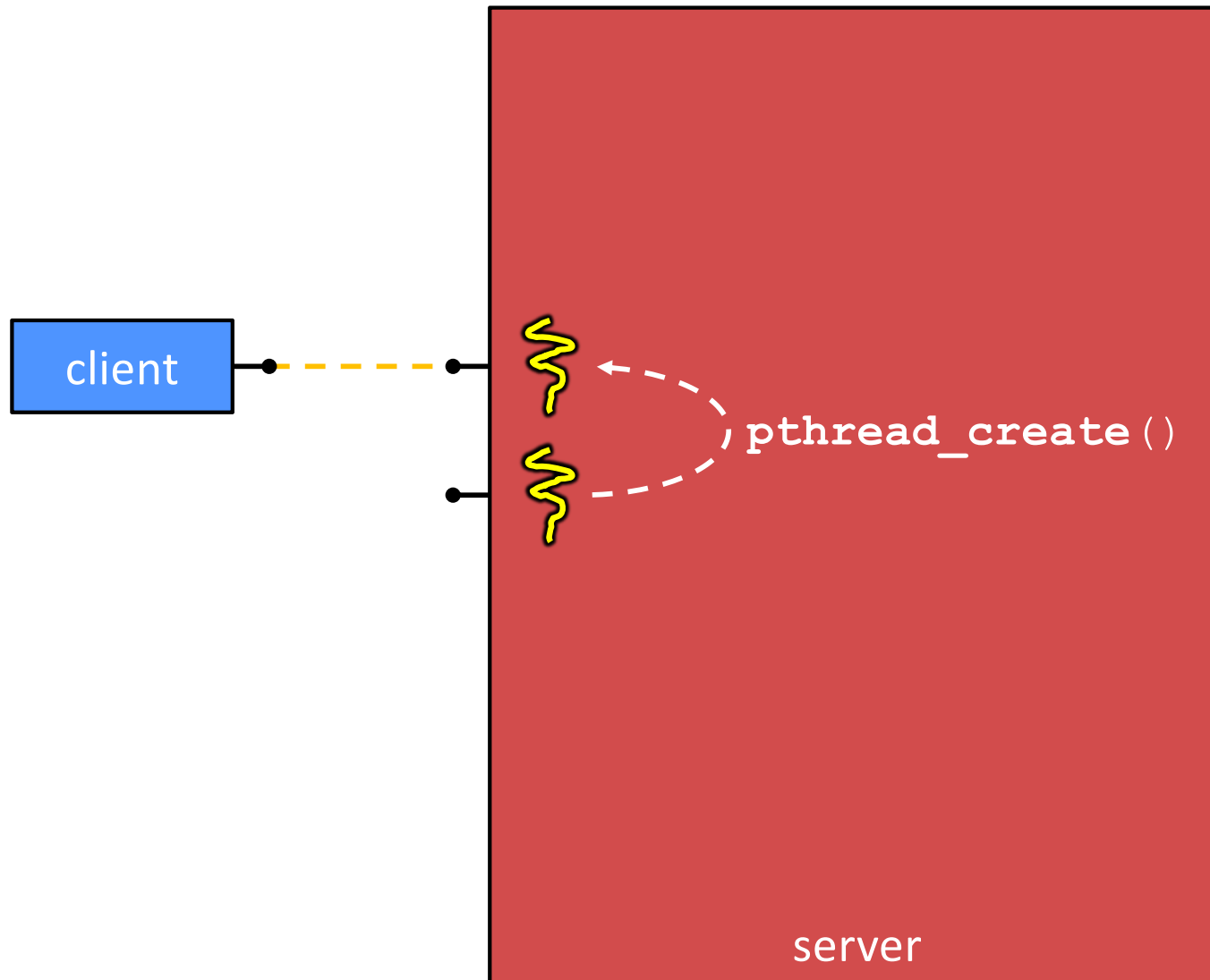
Concurrent Server with Threads

- ❖ 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

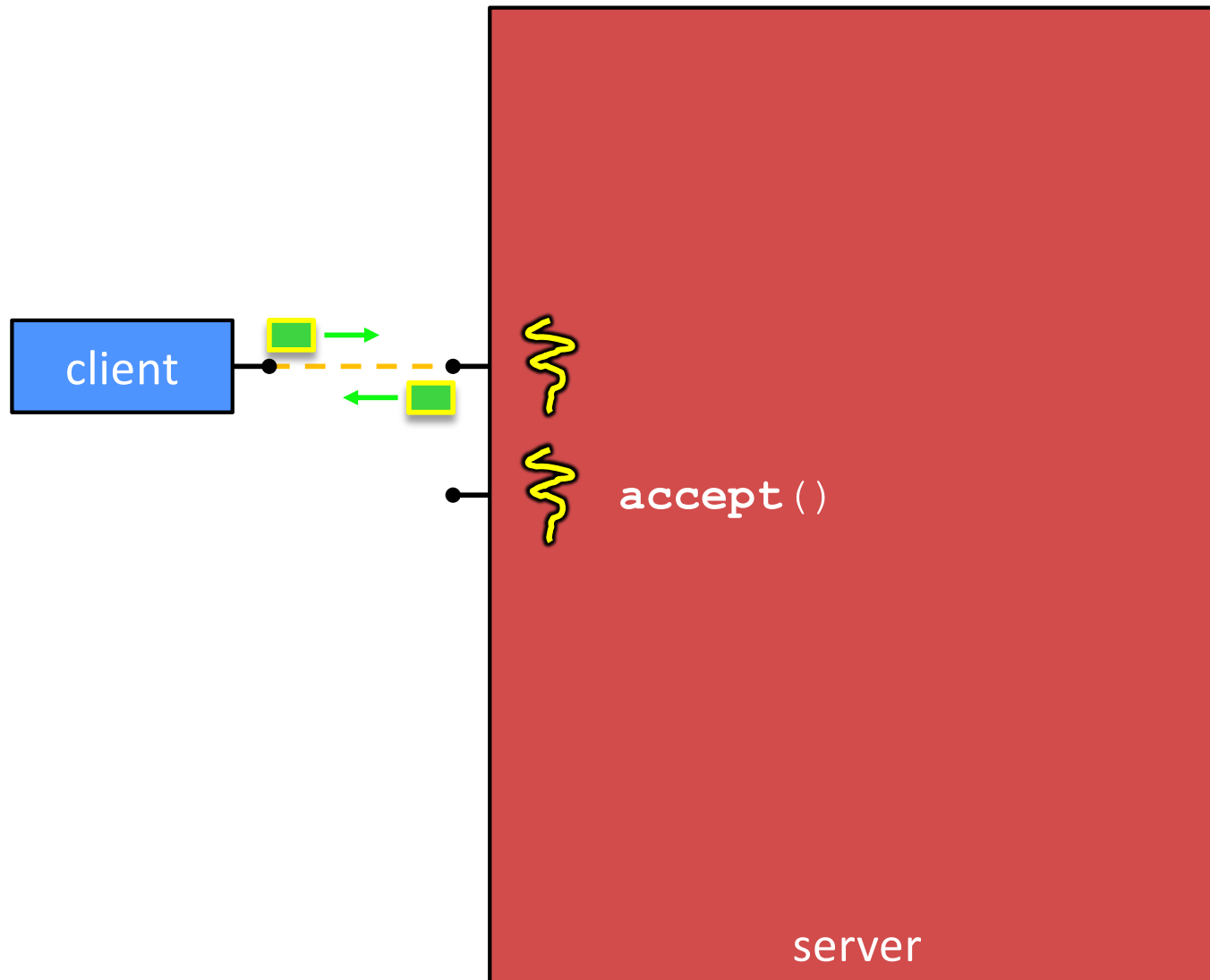
Multithreaded Server



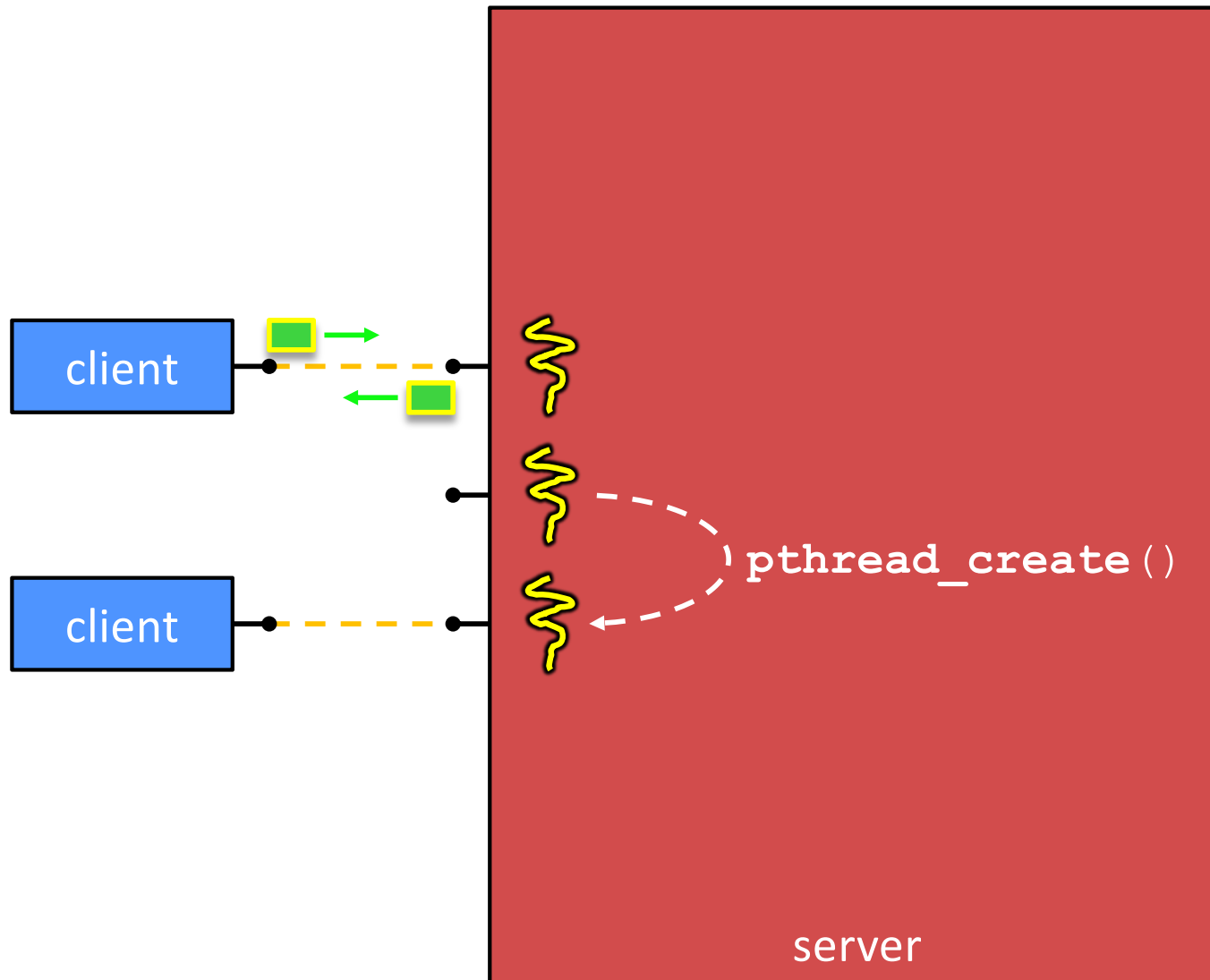
Multithreaded Server



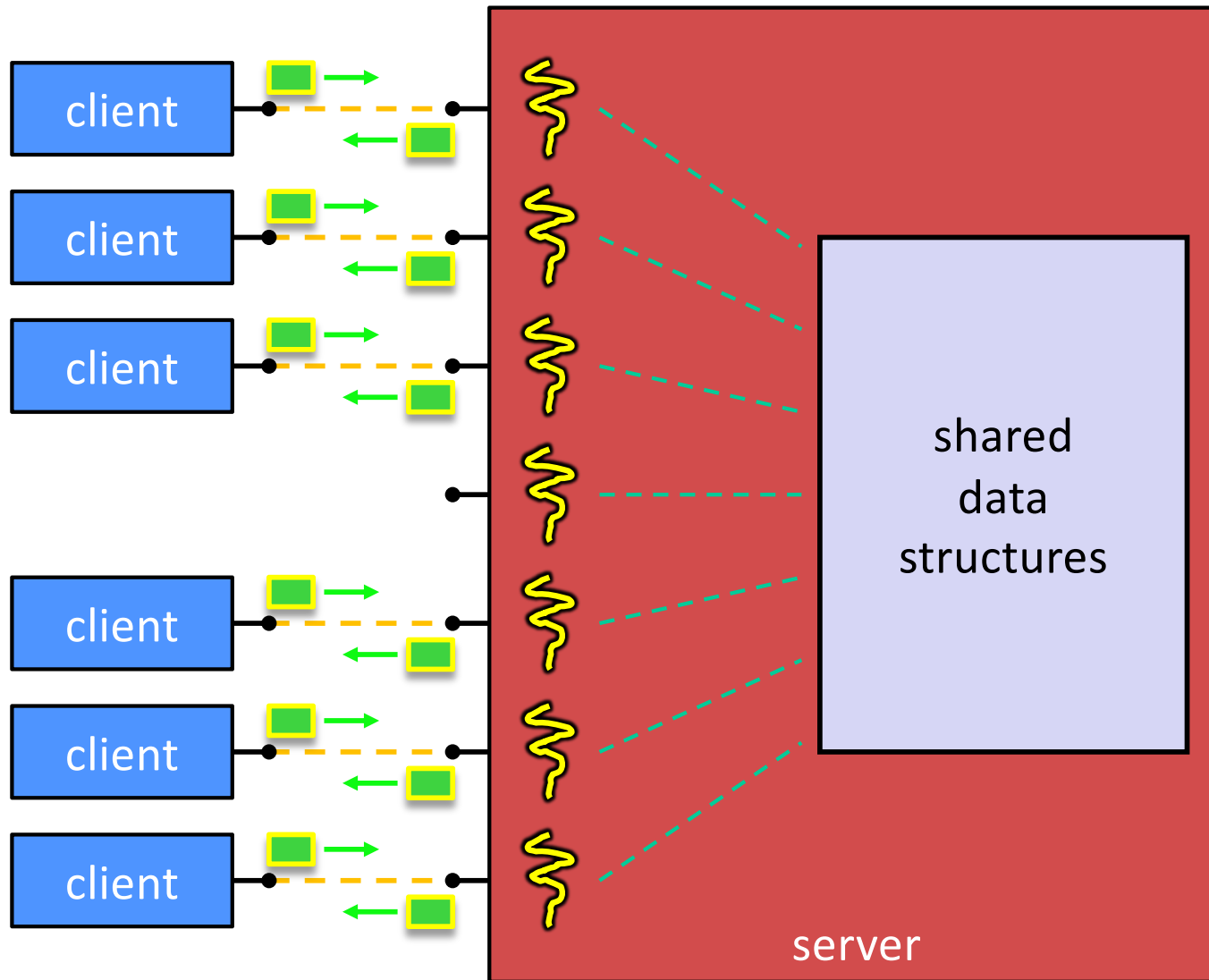
Multithreaded Server



Multithreaded Server



Multithreaded Server



Concurrent Server via Threads

- ❖ See `searchserver_threads/`

- ❖ **Notes:**
 - 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
 - How do you properly handle memory management?
 - Who allocates and deallocates memory?
 - How long do you want memory to stick around?

Whither 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

Threads and Data Races

- ❖ What happens if two threads try to mutate the same data structure?
 - They might interfere in painful, non-obvious ways, depending on the specifics of the data structure
- ❖ Example: two threads try to push an item onto the head of a linked list at the same time
 - Could get “correct” answer
 - Could get different ordering of items
 - Could break the data structure! ☠
 - Likely *will* get different results each time you run the program – a debugging nightmare

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  
}
```

Data Race Example

- ❖ Idea: leave a note!
 - Does this fix the problem?
- A. **Yes, problem fixed**
- B. **No, could end up with no milk**
- C. **No, could still buy multiple milk**
- D. **We're lost...**

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

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 (see CSE 451)
- ❖ **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:

- ❖ 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

```
// non-critical code
lock.acquire() ;  loop/idle if locked
// critical section
lock.release() ;
// non-critical code
```

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()
```

pthread and Locks

❖ Another term for a lock is a **mutex** (“mutual exclusion”)

- pthreads (`#include <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

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
 - Use pthreads in current exercise