

Concurrency: Threads

CSE 333

Guest Lecturer: Audrey Seo

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatwani

Derek de Leuw

Katie Gilchrist

Administrivia

- ❖ Ex17 due Monday - last exercise! 🎉
- ❖ HW4 due Wednesday night
- ❖ Final exam in class on Friday (1 hour)
 - ❖ Updated topic list and old exams on course web now
 - Some old finals are 1-hour summer exams, some are 2-hour regular quarters – don't panic if you can't finish those in 1 hour
 - ❖ Review Q&A in sections next week
 - ❖ Extra points for coming to office hours next week!
 - Same rules as midterm, see Wednesdays slides for details

Administrivia

- ❖ More on HW4... (due Wednesday night next week!)
 - 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
- ❖ Course evaluations open today
 - <https://uw.iasystem.org/survey/311914>

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)
 - Be sure to check for data in the buffer – might be an http request (or part of one) already there left over from a previous read
- ❖ 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()`
- ❖ We **won't** look at:
 - *Concurrent via non-blocking, event-driven I/O:*
`select()`

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

Sequential

❖ Pseudocode:

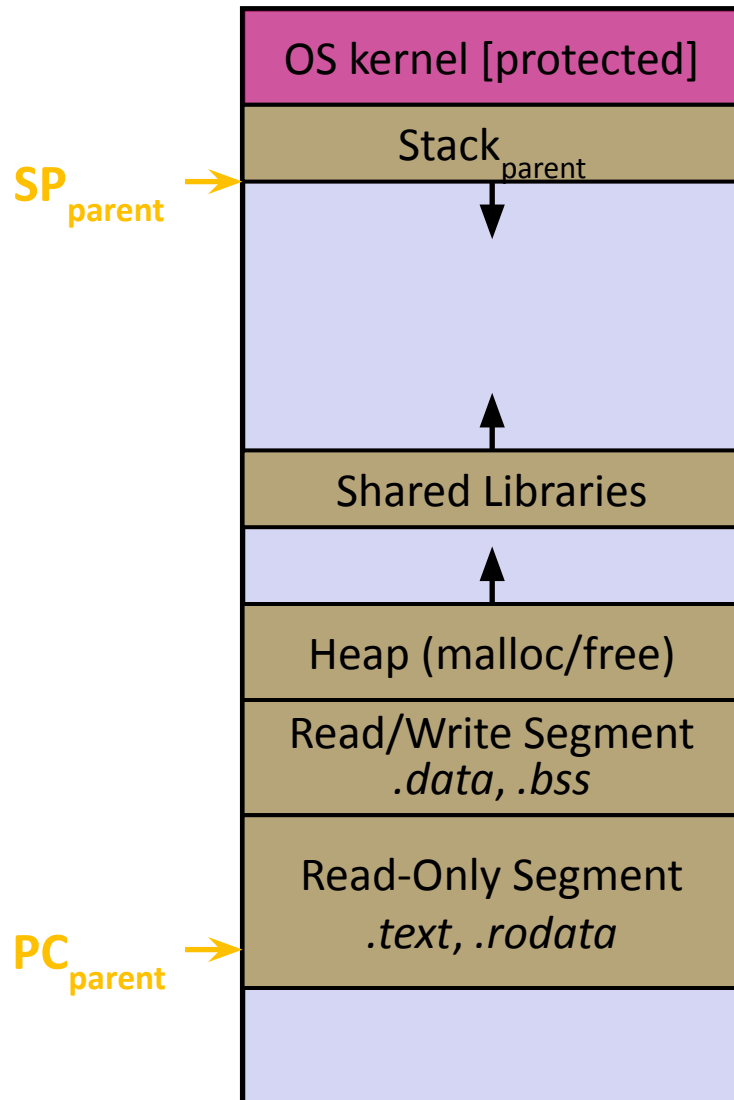
```
listen_fd = Listen(port);  
  
while (1) {  
    client_fd = accept(listen_fd);  
    buf = read(client_fd);  
    resp = ProcessQuery(buf);  
    write(client_fd, resp);  
    close(client_fd);  
}
```

❖ See [searchserver_sequential/](#) for more details

Threads

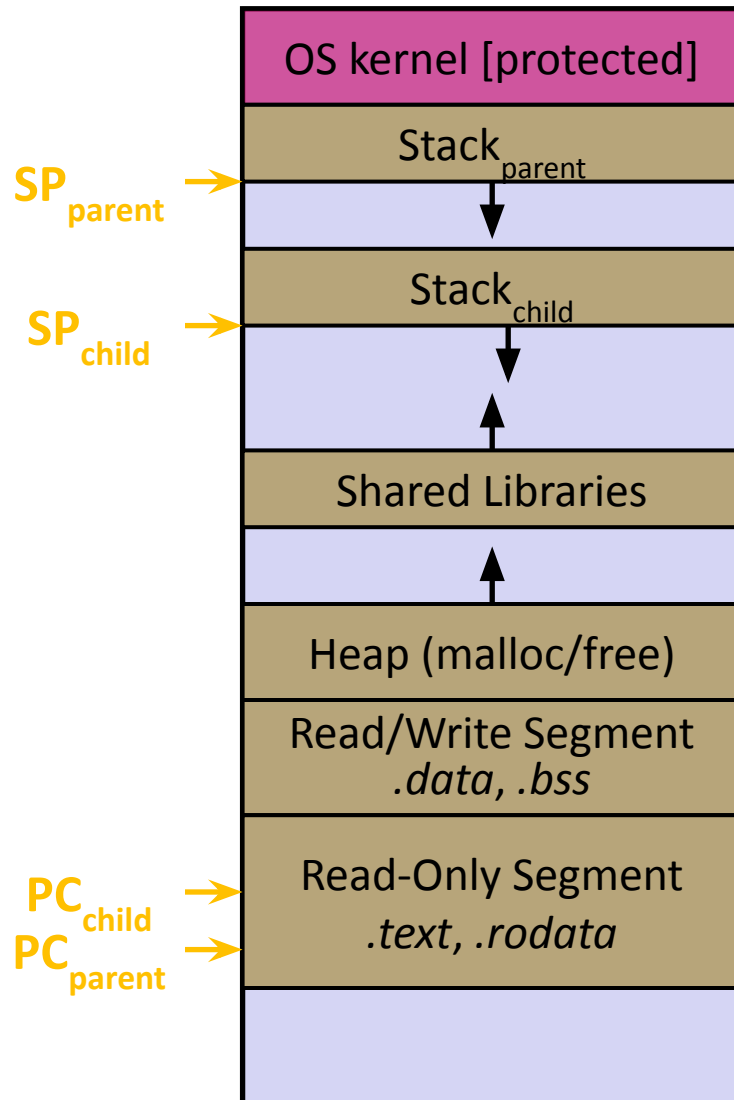
- ❖ Threads are like lightweight processes
 - They execute concurrently like processes
 - Multiple threads can run simultaneously on multiple CPUs/cores
 - Unlike processes, threads cohabit the same address space
 - Threads within a process see the same heap and globals and can communicate with each 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



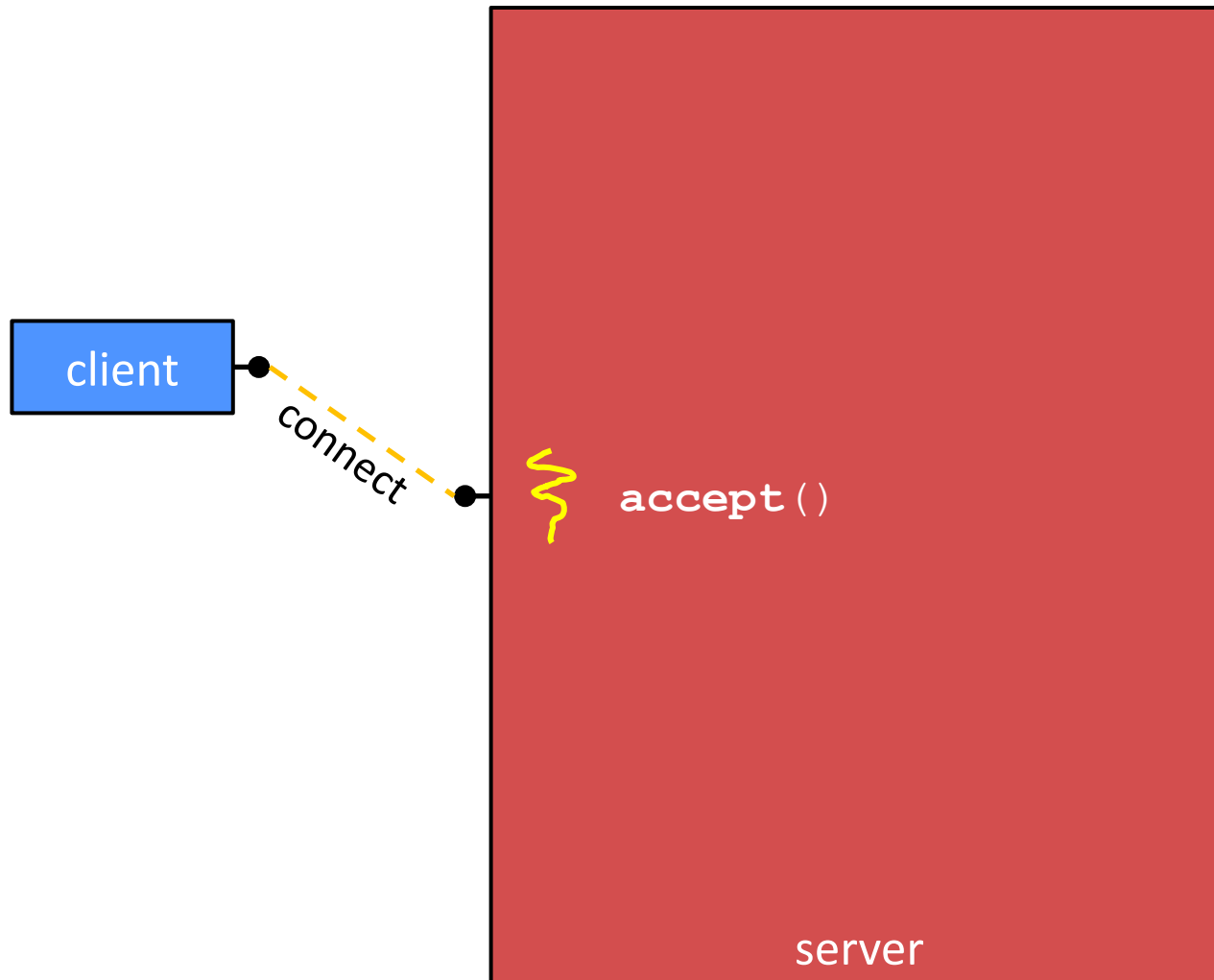
❖ After creating a thread

- *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

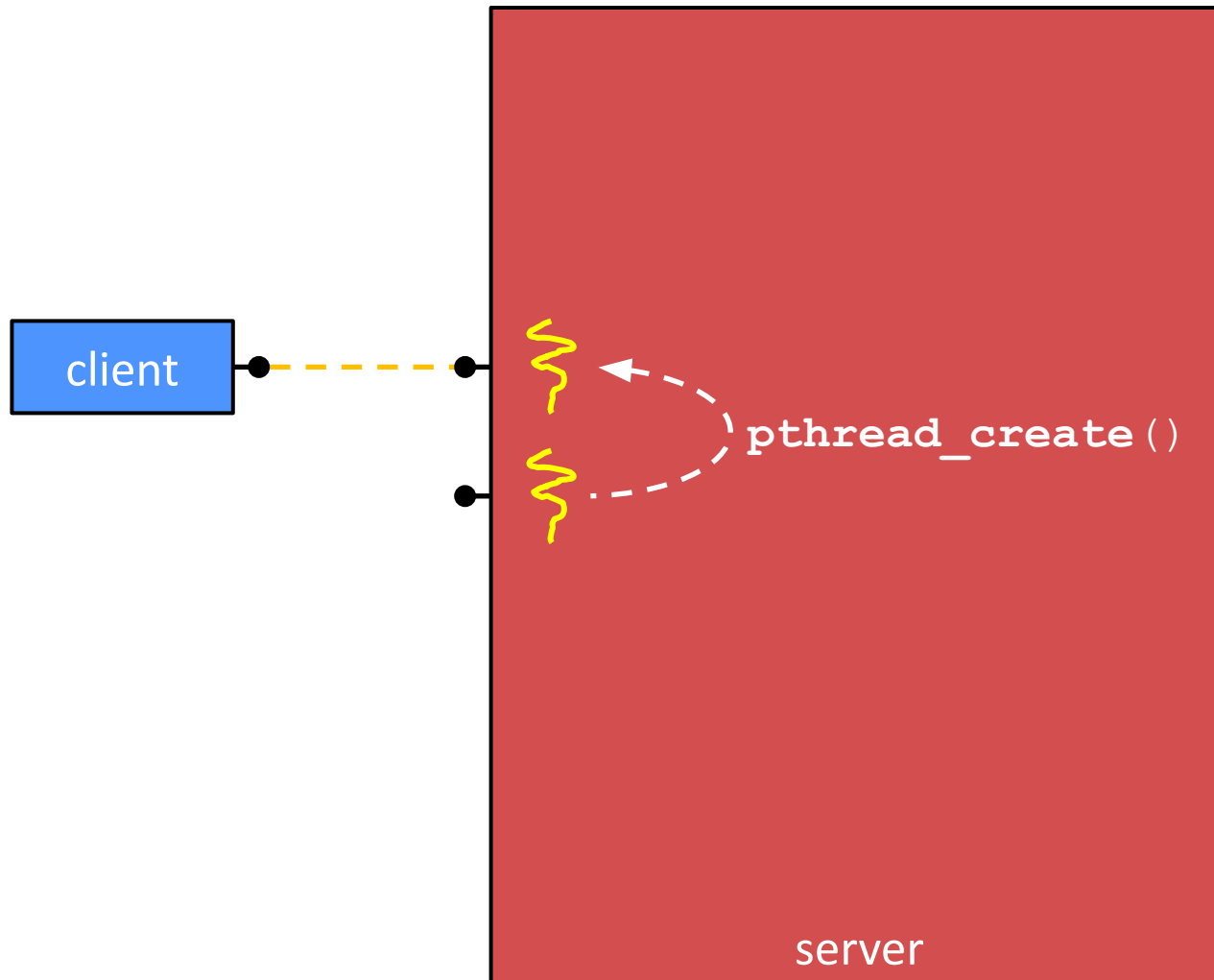
Multithreaded Server: Architecture

- ❖ A parent *thread* creates a new thread to handle each incoming connection
 - The child thread handles the new connection and subsequent I/O, then exits when the connection terminates

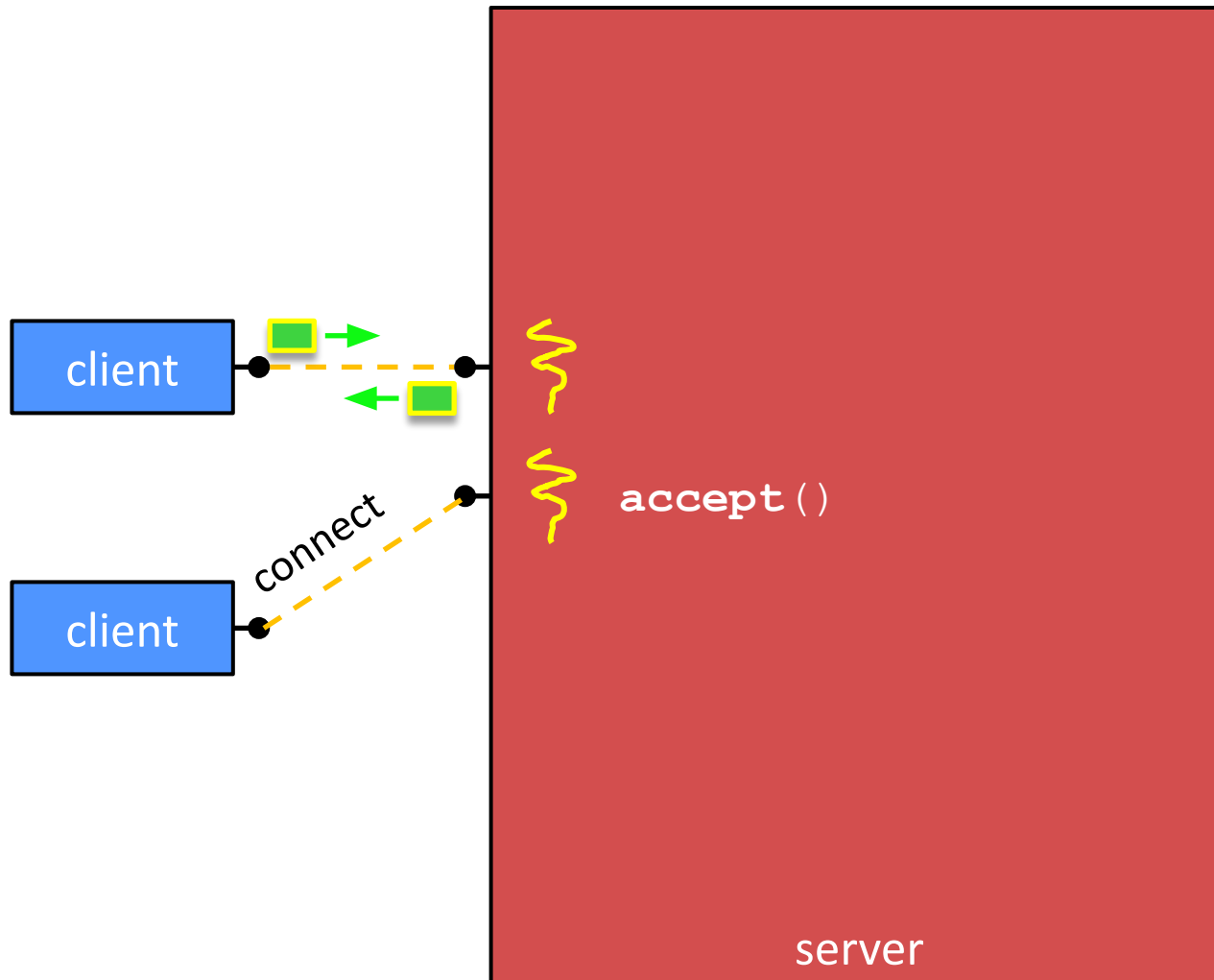
Multithreaded Server



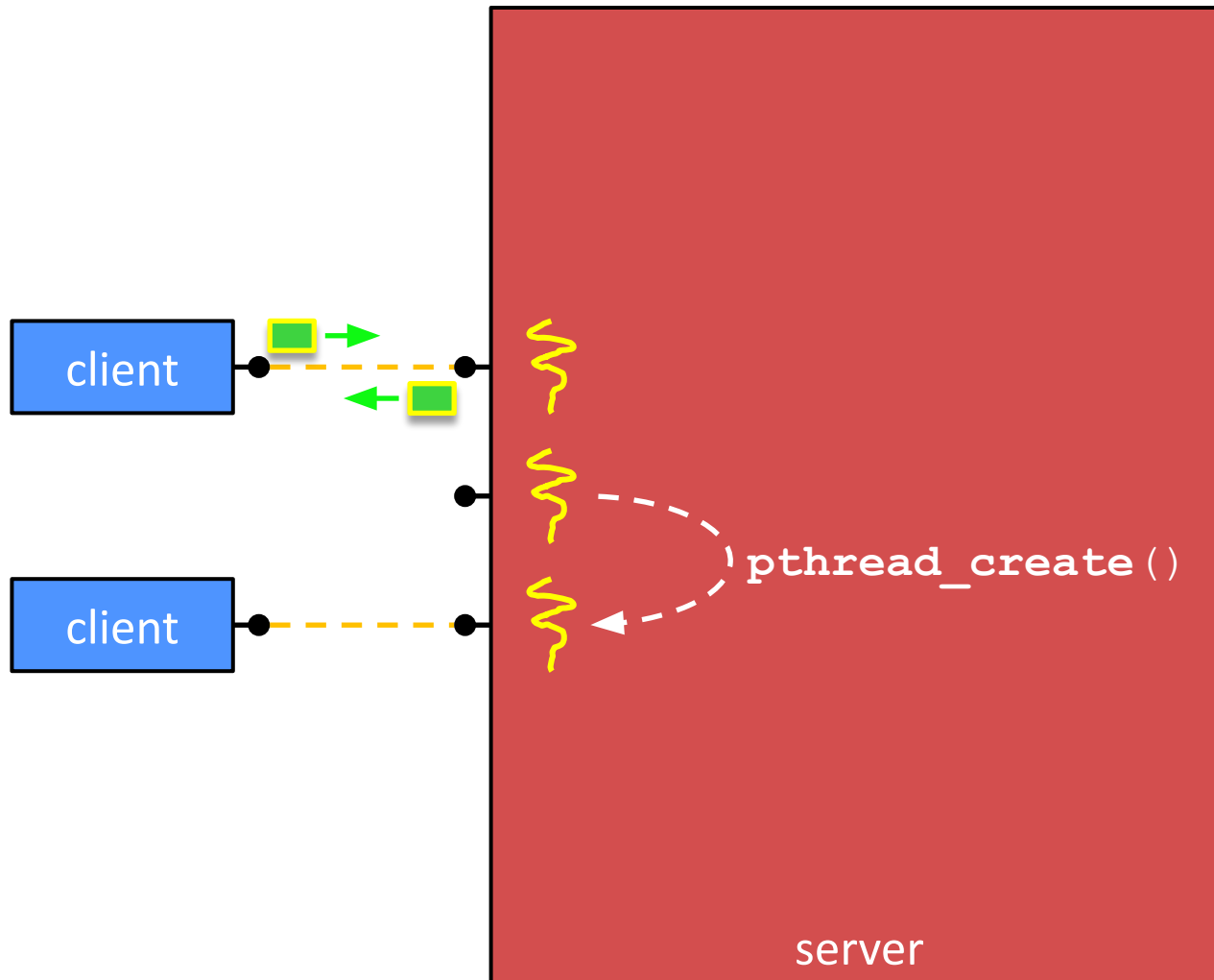
Multithreaded Server



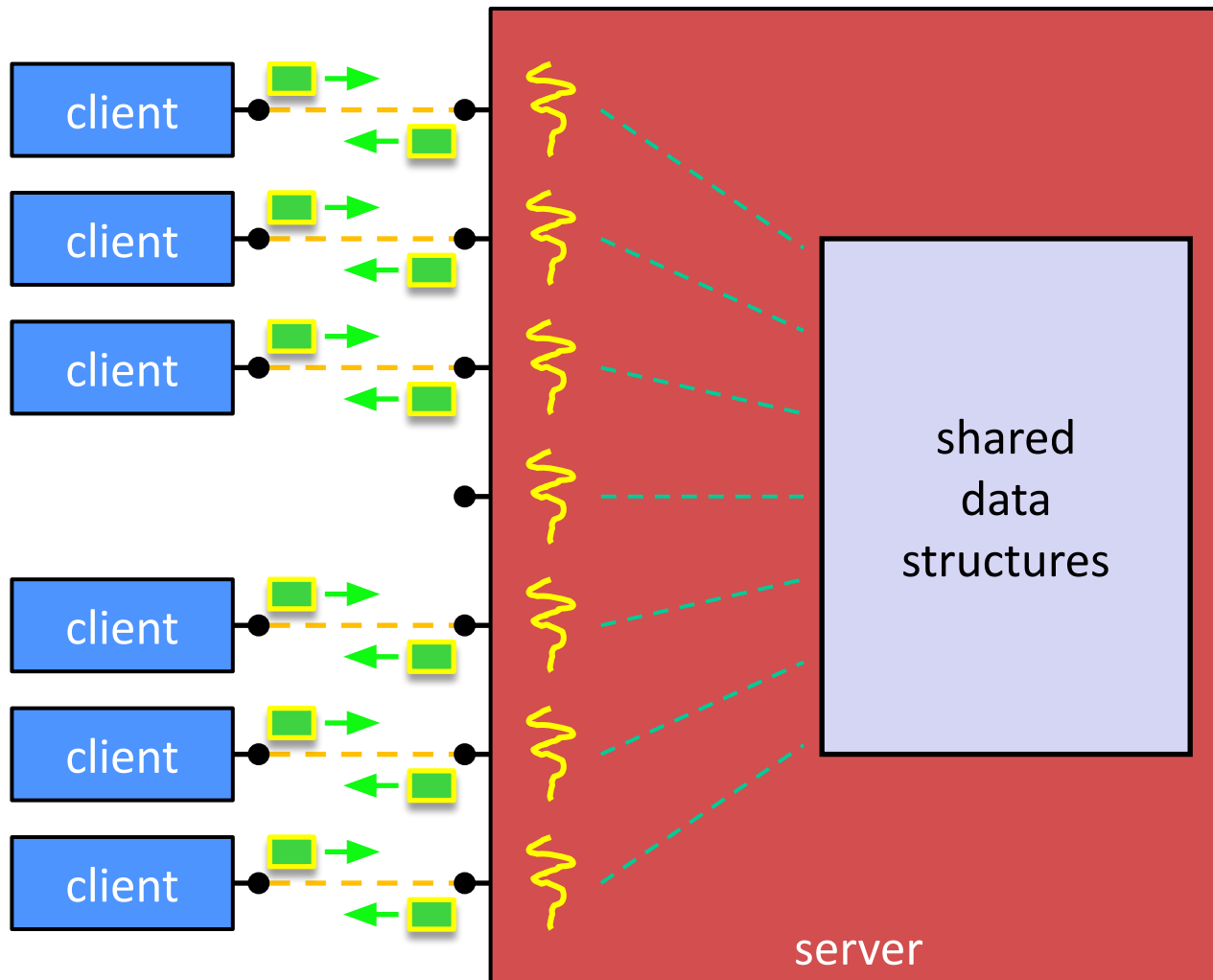
Multithreaded Server



Multithreaded Server



Multithreaded Server



POSIX Threads (pthreads)

- ❖ The POSIX APIs for dealing with threads
- ❖ Declared in `pthread.h`
 - Not part of the C/C++ language (unlike Java)
- ❖ To enable support for multithreading, must include `-pthread` flag when compiling and linking with `gcc` command

pthread Threads: Creation

For advanced usage;
always nullptr in this class.

❖

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

- Creates a new thread into *thread, with attributes *attr
- Returns a status code (0 or an error number)
- The new thread runs **start_routine** (arg)

❖

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

- Equivalent of **exit** (retval) for a thread instead of a process
- thread automatically exits when it returns from **start_routine** ()

pthread Threads: Afterwards



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

- Waits for thread to terminate
- Exit status of the terminated thread is placed in `**retval`



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

- Mark thread as detached ; will clean up its resources as soon as it terminates

pthread Example

See:

`thread_example.cc`

<https://courses.cs.washington.edu/courses/cse333/25su/lecture/22-threads-example>

Concurrent Server via Threads

❖ See `searchserver_threads/` for details

❖ 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?

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 (!milk) {  
    buy milk  
}
```

Too much milk!

- ❖ If you live with a roommate:



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

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)

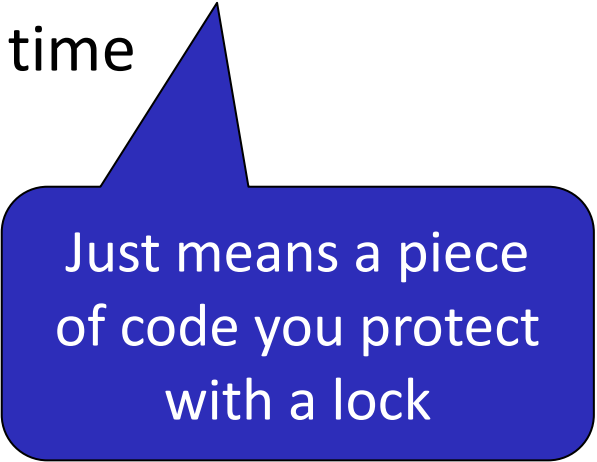
Synchronization

- ❖ It turns out, safe synchronization is impossible with the tools we've seen so far
- ❖ We need special support from the hardware for threads to interact safely
- ❖ The solution: **locks!**



Synchronization

- ❖ A **lock** combines the “check note” and “write note” operations into one **atomic** operation
 - **Atomic**: cannot be interleaved with another thread
- ❖ Use a **lock** to grant access to a *critical section* so that only one thread can operate there at a time




Just means a piece
of code you protect
with a lock

Lock Synchronization

- ❖ Two main operations on locks:
 - 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() ;  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()
```

Beware of Deadlocks

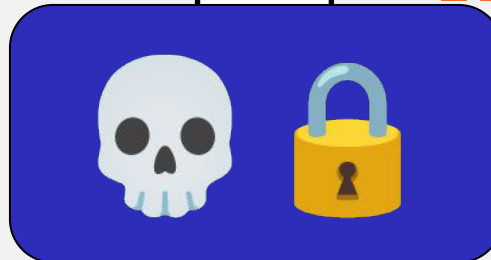
- ❖ What if our roommates want to go to the store only when there is more than one thing to get?



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



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



Synchronization

❖ Goals of synchronization:

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

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

pthread and Locks

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

- pthreads (`#include <pthread.h>`) defines datatype `pthread_mutex_t`

For advanced usage; always `nullptr` in this class.

- ❖

```
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

But I only want to read the data!

- ❖ Is a lock needed when reading shared data?
 - No if all threads *only* read the shared data
 - Yes if **any** thread could potentially write to the shared data!
- ❖ Why?
 - The C and C++ standards do not guarantee that writes of multi-byte data are indivisible when observed from other asynchronous threads
 - i.e., writing multiple bytes to memory might involve multiple updates to caches or backing stores
 - Which means a reading thread might be able to see the results of a partial, not-yet-finished update if it does not use locks

But I only am reading the data!

- ❖ Example: Suppose shared 32-bit int x is initially 0x0000FFFF
- ❖ Thread 1 properly updates x using locks:
 - acquire x_lock ;
 - $x = x + 1$;
 - release x_lock ;
- ❖ Thread 2 only reads x and outputs it without locking:
 - Might print 0x0000FFFF (old value)
 - Might print 0x00010000 (new value)
 - Might print 0x0001FFFF (partially updated value) !!!!!

But I only am reading the data!

❖ How to fix:

- Thread 2 must acquire `x_lock` before printing and release it afterwards

❖ In practice...

- On modern x86/arm/etc. processors this won't happen for things like aligned small ints that don't span cache boundaries, so you probably won't see the bug unless you're using larger data structures – but the C/C++ language does **not** guarantee this behavior! Use locks or atomics (see C/C++ refs for details) if there are any writers to a shared variable!!

C++11 Threads

- ❖ C++11 added threads and concurrency to its libraries
 - 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 our exercise

C++11 Threads

❖ C++11 threads headers:

- `<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

Don't forget!

- ❖ Ex17 due **Monday August 18th** - last exercise! 🎉
- ❖ HW4 due **Wednesday night August 20th**
- ❖ Final exam in class on **Friday August 22nd** (1 hour)
- ❖ Extra points for coming to office hours next week!
 - Same rules as midterm (see slides from 7/16)