Threads Cont. Locks & Concurrency Benefits

Computer Operating Systems, Summer 2025

Instructors: Joel Ramirez Travis McGaha

TAs: Ash Fujiyama Maya Huizar Sid Sannapareddy

Poll Everywhere

pollev.com/tqm

- How are you enjoying the weather?

Administrivia

- PennOS:
  - Specifications and team sign-up have been posted
  - Done in groups of 4
  - Partner signup due by end of day **TODAY**
    - Those left unassigned will be randomly assigned tomorrow morning (Tuesday the 31st)
  - Lecture dedicated to PennOS in class **tomorrow**. Highly recommend you go. Highly recommend you read some of the spec today.
  - PennOS Due Dates for milestones to be updated tonight.

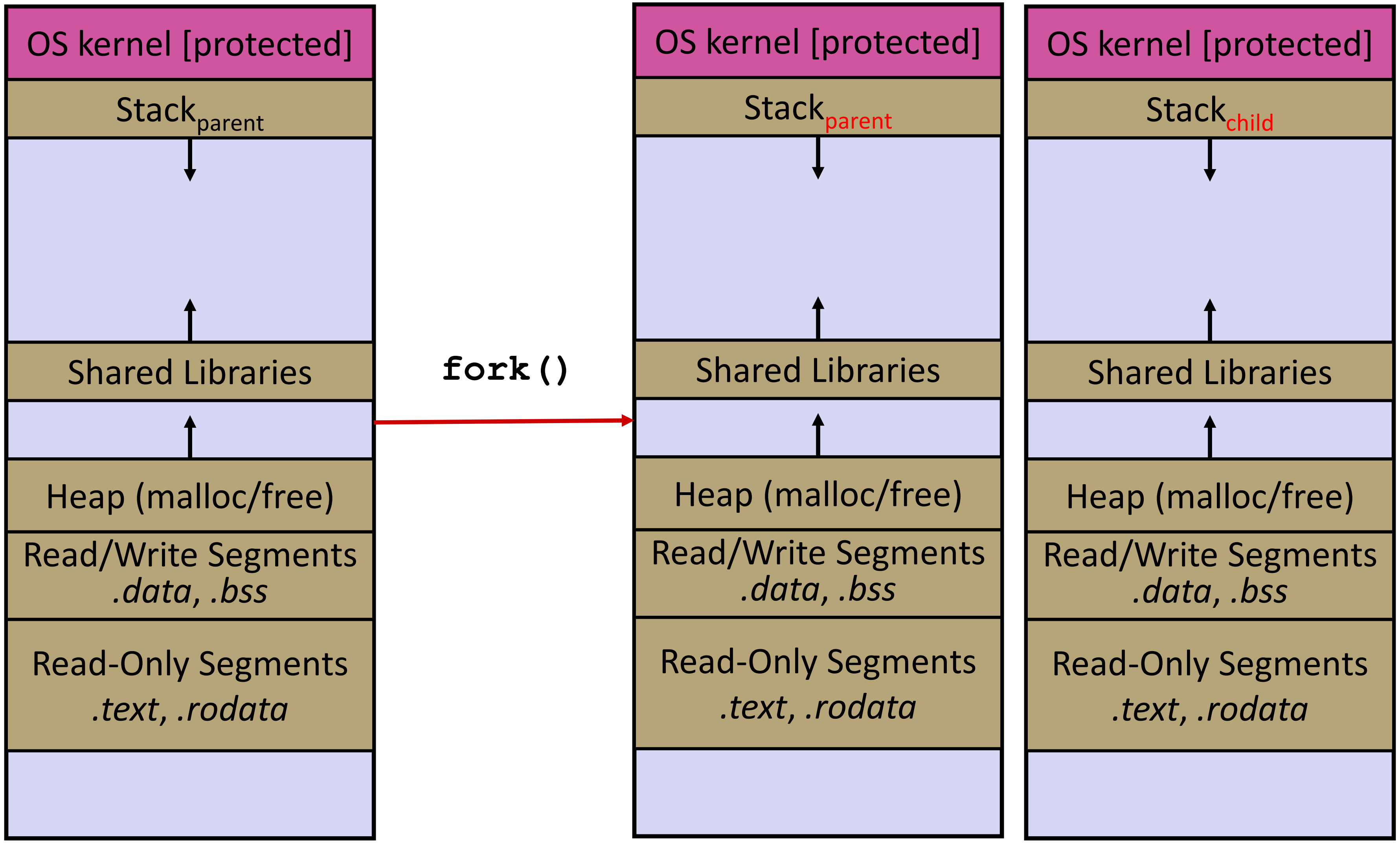
Lecture Outline

- **Threads Refresher**
- Mutex
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- Benefits of Concurrency

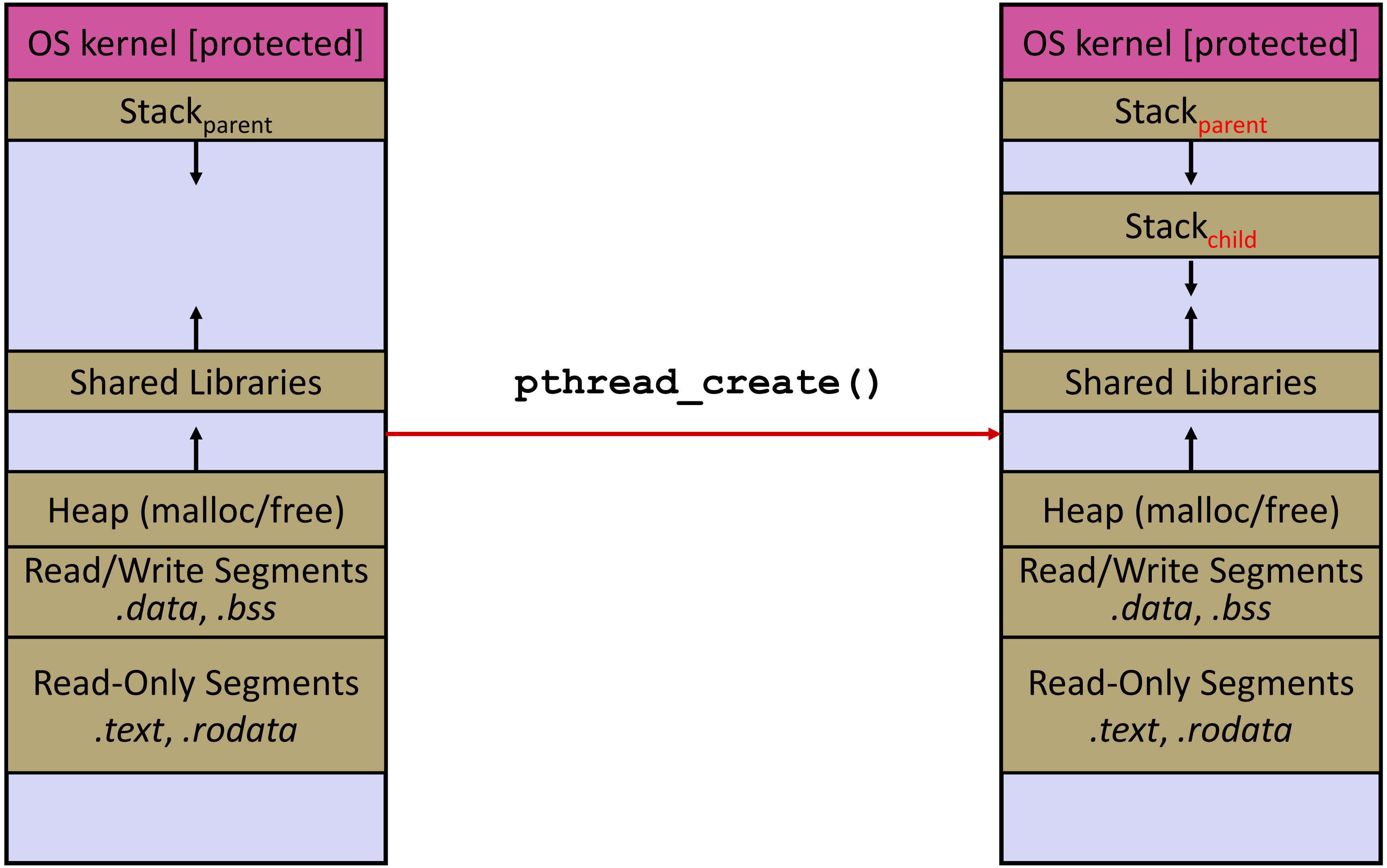
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources, & security attributes
  - A Thread has a unique: stack, stack pointer, program counter, & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

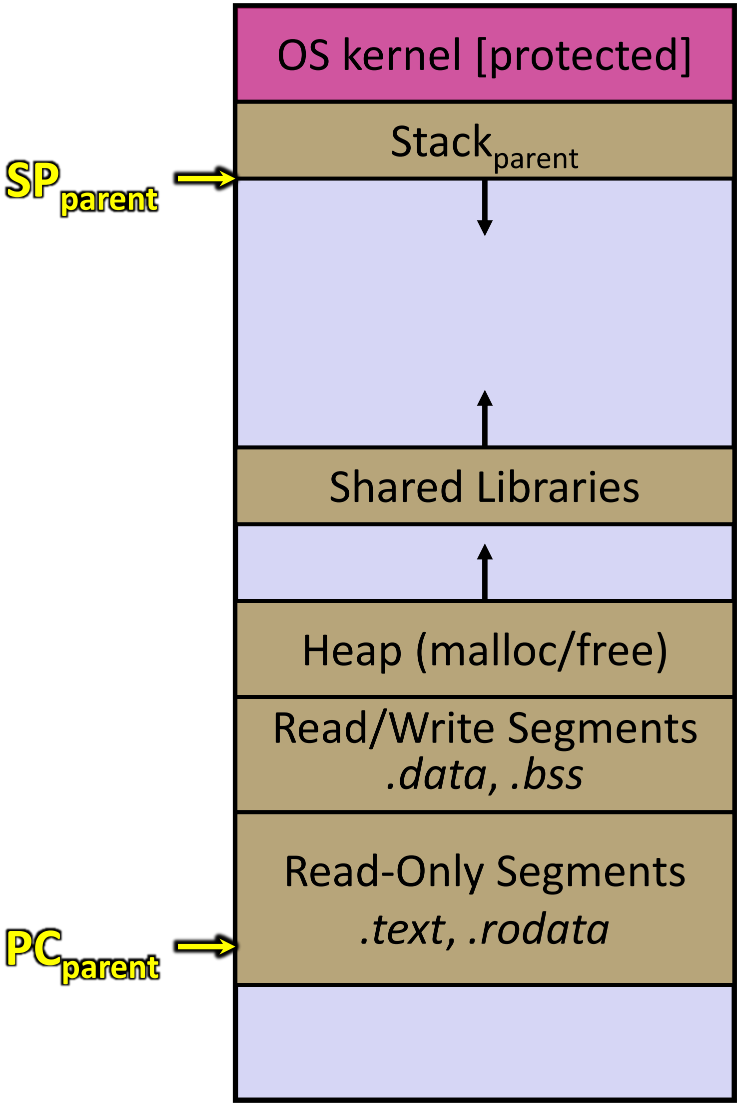
Threads vs. Processes



Threads vs. Processes

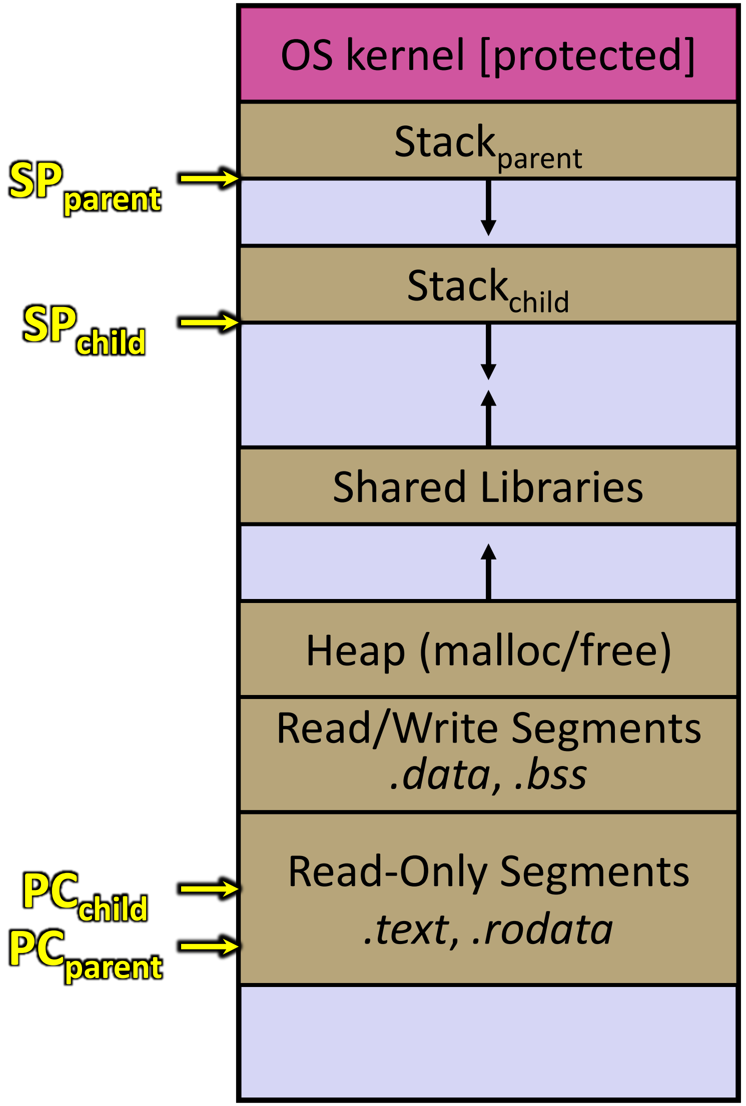


Single-Threaded Address Spaces



- 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()`

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

POSIX Threads (pthreads)

- 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 -pthread -o main main.c`
  - Implemented in C
    - Must deal with C programming practices and style

Creating and Terminating Threads

Output parameter.
Gives us a "thread_descriptor"

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

  Argument for the thread function

  Function pointer!
  Takes & returns void*
  to allow "generics" in C

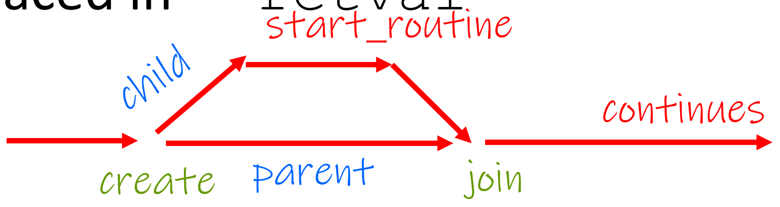
  - Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
  - Returns **0** on success and an error number on error (can check against error constants)
  - The new thread runs **`start_routine`**`(arg)`

pthread_create → child → start_routine
parent → continues
  ```

What To Do After Forking Threads?

- `int pthread_join(pthread_t thread, void** retval);`
  - Waits for the thread specified by `thread` to terminate
  - The thread equivalent of `waitpid()`
  - The exit status of the terminated thread is placed in `**retval`

Parent thread waits for child thread to exit, gets the child's return value, and child thread is cleaned up



Lecture Outline

- Threads Refresher
- **Mutex**
- Data Race vs Race Condition
- Is a mutex needed? (Peterson's)
- Benefits of Concurrency

Shared Resources

- Some resources are shared between threads and processes

- Thread Level:
  - Memory
  - Things shared by processes

- Process level
  - I/O devices
    - Files
    - terminal input/output
    - The network

Issues arise when we try to shared things

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? When did a thread get interrupted?)**

Data Race Example

- If your fridge has no milk, then go out and buy some more
  - What could go wrong?

```
if (!milk) {

  buy milk

}
```

- If you live alone:

- If you live with a roommate:

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

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

A. Yes, problem fixed

B. No, could end up with no milk

C. No, could still buy multiple milk

D. We're lost…

Poll Everywhere

pollev.com/tqm

- Idea: leave a note!
  - Does this fix the problem?

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

We can be interrupted between checking note and leaving note ☹

A. Yes, problem fixed
B. No, could end up with no milk
C. No, could still buy multiple milk
D. We're lost…

*There are other possible scenarios that result in multiple milks

you		roommate
Check note		
		Check note
Check milk		
Leave note		
		Check milk
		Leave note
		Buy milk
Buy milk		
	time	

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! ☠

Remember this?

- What does this print?

```
#define NUM_PROCESSES 50
#define LOOP_NUM 100

int sum_total = 0;

void loop_incr() {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
}


int main(int argc, char** argv) {
  pid_t pids[NUM_PROCESSES];  // array of process ids

  // create processes to run loop_incr()
  for (int i = 0; i < NUM_PROCESSES; i++) {
    pids[i] = fork();
    if (pids[i] == 0) {
      // child
      loop_incr();
      exit(EXIT_SUCCESS);
    }
    // parent loops and forks more children
  }

  // wait for all child processes to finish
  for (int i = 0; i < NUM_PROCESSES; i++) {
    waitpid(pids[i], NULL, 0);
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

Always prints 0, the global counter is not shared across processes, so the parent's global never changes

Remember this?

- What does this print?

```
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &thread_main, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

Usually 5000

Previous Demos:

- See `total.c` and `total_processes.c`
  - Threads share an address space, if one thread increments a global, it is seen by other threads
  - Processes have separate address spaces, incrementing a global in one process does not increment it for other processes

- NOTE: sharing data between threads is actually kinda unsafe if done wrong (we are doing it wrong in this example), **more on this NOW**

Increment Data Race

- What seems like a single operation is actually multiple operations in one. The increment looks something like this in assembly:

```
++sum_total
```

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

- What happens if we context switch to a different thread while executing these three instructions?

- **Reminder: Each thread has its own registers to work with. Each thread would have its own R0**

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD sum_total into R0
```

Thread 1

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 R0 = 0

```
LOAD sum_total into R0
```

Thread 1 R0 = 0

```
LOAD sum_total into R0
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 0

Thread 0 **R0 = 0**

```
LOAD   sum_total into R0
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 R0 = 0

```
LOAD  sum_total into R0
```

Thread 1 R0 = 1

```
LOAD  sum_total into R0
ADD   R0 R0 #1
STORE R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0



ADD    R0 R0 #1
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

Increment Data Race

- Consider that sum_total starts at 0 and two threads try to execute

```
++sum_total
```

sum_total = 1

Thread 0 **R0 = 1**

```
LOAD   sum_total into R0



ADD    R0 R0 #1
STORE  R0 into sum_total
```

Thread 1 **R0 = 1**

```
LOAD   sum_total into R0
ADD    R0 R0 #1
STORE  R0 into sum_total
```

- With this example, we could get 1 as an output instead of 2, even though we executed ++sum_total twice

Poll Everywhere

pollev.com/tqm

- What is the minimum value that could be printed?

```c
#define NUM_THREADS 50
#define LOOP_NUM 100

int sum_total = 0;

void* thread_main(void* arg) {
  for (int i = 0; i < LOOP_NUM; i++) {
    sum_total++;
  }
  return NULL;  // return type is a pointer
}


int main(int argc, char** argv) {
  pthread_t thds[NUM_THREADS];  // array of thread ids

  // create threads to run thread_main()
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_create(&thds[i], NULL, &thread_main, NULL) != 0) {
      fprintf(stderr, "pthread_create failed\n");
    }
  }

  // wait for all child threads to finish
  // (children may terminate out of order, but cleans up in order)
  for (int i = 0; i < NUM_THREADS; i++) {
    if (pthread_join(thds[i], NULL) != 0) {
      fprintf(stderr, "pthread_join failed\n");
    }
  }

  printf("%d\n", sum_total);

  return EXIT_SUCCESS;
}
```

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 eventually 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

- 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();   block
                  if locked
// critical section
lock.release();

// non-critical code
```

Lock API

- Locks are constructs that are provided by the operating system to help ensure synchronization
  - Often called a mutex or a semaphore

- Only one thread can acquire a lock at a time,
  No thread can acquire that lock until it has been released

- Has memory barriers built into it and usually uses TSL to ensure that acquiring the lock is atomic (more on TSL and memory barriers in a little bit)

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 Un-blocks when lock is acquired

```
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.c`
  - Data race between threads
- See `total_locking.c`
  - Adding a mutex fixes our data race
- How does `total_locking` compare to sequential code and to `total`?
  - Likely *slower* than both– only 1 thread can increment at a time, and must deal with checking the lock and switching between threads
  - One possible fix: each thread increments a local variable and then adds its value (once!) to the shared variable at the end
    - See `total_locking_better.c`

Lecture Outline

- Threads Refresher
- Mutex
- **Data Race vs Race Condition**
- Is a mutex needed? (Peterson's)
- Benefits of Concurrency

```
pthread_mutex_t lock;
bool print_ok = false;

void* thrd_fn1(void* arg) {
  pthread_mutex_lock(&lock);
  print_ok = true;
  pthread_mutex_unlock(&lock);
  return NULL;
}

void* thrd_fn2(void* arg) {
  pthread_mutex_lock(&lock);
  if (print_ok) {
    printf("print ok is true\n");
  } else {
    printf("print ok is false\n");
  }
  pthread_mutex_unlock(&lock);
  return NULL;
}

int main() {
  // assume main sets ups the threads & locks, etc.
```

pollev.com/tqm

- Does this code have a data race?
  - Can this program enter an “invalid” (unexpected or error) state from having concurrent memory accesses?
- Follow up: Does this code feel good?

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
- Data-races are a subset of race-conditions

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 produce things once the producer has produced them

- Need to make sure this communication has no data race or race condition

Lecture Outline

- Threads Refresher
- Mutex
- Data Race vs Race Condition
- **Is a mutex needed? (Peterson's)**
- Benefits of Concurrency

Poll Everywhere

pollev.com/tqm

- Lets try a more complicated software approach..
- We create two threads running thread_code, one with **`arg = 0`**, other thread has **`arg = 1`**
- Each thread tries to increment sum_total. Does this work?

```
int sum_total = 0;
bool flag[2] = {false, false};
int turn = 0

void thread_code(int arg) {
  int me = arg;

  flag[me] = true;
  turn = 1 - me;
  while((flag[1-me] == true) && (turn != me)) { }
  ++sum_total;
  flag[me] = false;
}
```

Check the index of the other thread

Peterson’s Algorithm

- What we just did was Peterson's algorithm
- Why does it work? (using an analogy)
  - Each thread first declares that they want to enter the critical section by setting their flag
  - Each thread then states (once) that the other should “go first”.
    - This is done by setting the turn variable to 1 – me
    - One of these assignments to the turn variable will happen last, that is the one that decides who goes first
  - One of the thread goes first (decided by the value of turn) and accesses the critical section, before saying it is done (by changing their flag to false)

Peterson's Algorithm

- What we just did was Peterson's algorithm
- Why does it work?
  - Case1:
    If P0 enters critical section, flag[0] = true, turn = 0. It enters the critical section successfully.
  - Case2:
    If P0 and P1 enter critical section, flag[0] and flag[1] = true

    Race condition on turn. Suppose P0 sets turn = 0 first. Final value is turn = 1. P0 will get to run first.

Explanation

TIME

	Thread 0	Thread 1	
	`flag[0] = true`		
		`flag[1] = true`	
RACE	`turn = 1` `turn = 1`	`turn = 0`	turn = ?
	// suppose `turn = 1` came after `turn = 0` // the `turn` variable is set to 1		turn = 1
	`while(flag[1] == true && turn != 0)`	`while(flag[0] == true && turn != 1)`	
		`++sum_total`	
		`flag[1] = false`	
	`++sum_total`		

Peterson's Assumptions

- Some operations are atomic:
  - Reading from the flag and turn variables cannot be interrupted
  - Writing to the flag and turn variables cannot be interrupted
  - E.g setting turn = 1 or 0 will set turn to 0 or 1, you can be interrupted before or after, but not "during" when turn may have some intermediate value that is not 0 or 1

- That the instructions are executed in the specific order laid out in the code

Atomicity

- Atomicity: An operation or set of operations on some data are *atomic* if the operation(s) are indivisible, that no other operation(s) on that same data can interrupt/interfere.

- Aside on terminology:
  - Often interchangeable with the term “Linearizability”
  - Atomic has a different (but similar-ish) meaning in the context of data bases and ACID.

Aside: Instruction & Memory Ordering

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

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

void some_func(int arg) {
  t = 5;
  g = true;
}
```

Aside: Instruction & Memory Ordering

NO

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

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

void some_func(int arg) {
  t = 5;
  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 = 5;` 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

- Or: just use a real method of synchronization (we will talk about more of these in Thursday's lecture)

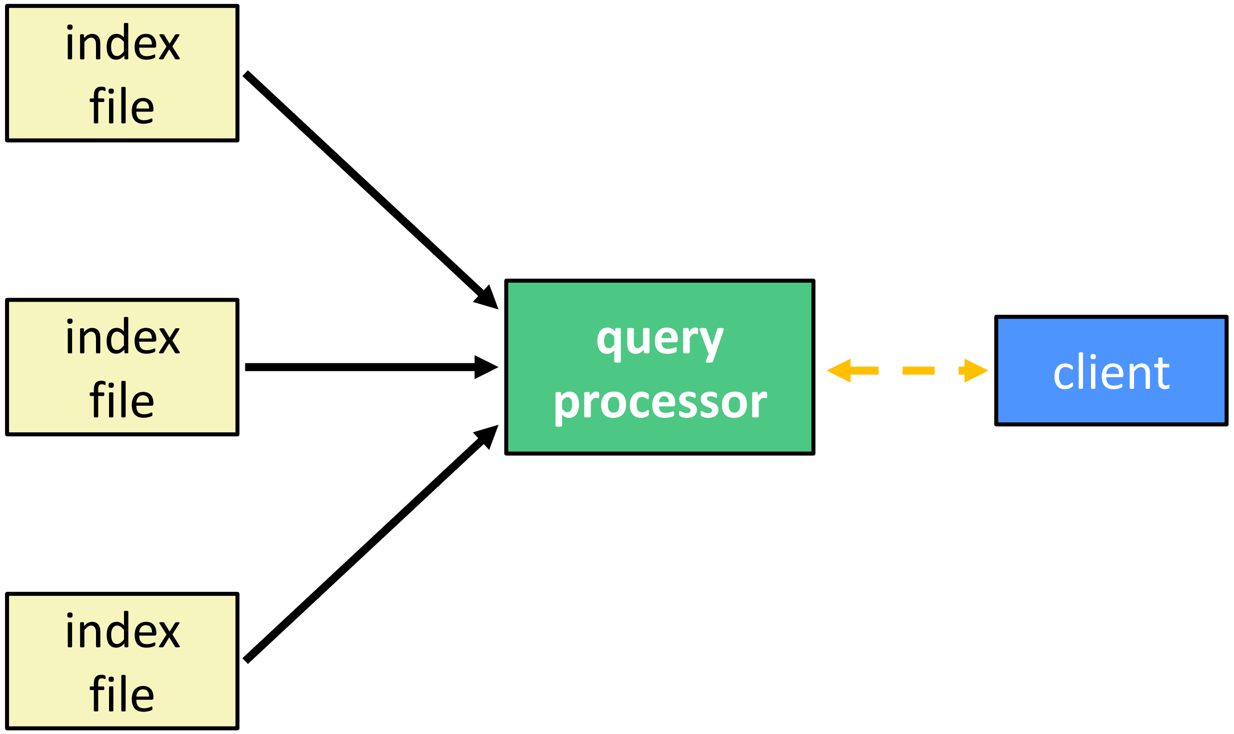
Lecture Outline

- Threads Refresher
- Mutex
- Data Race vs Race Condition
- Is a mutex needed? (Peterson’s)
- **Benefits of Concurrency**

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



Poll Everywhere

pollev.com/tqm

- This is pseudo code for what our single threaded server does.

- When do you think our code interacts with the network?
- How often 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);
  }
}
```

Poll Everywhere

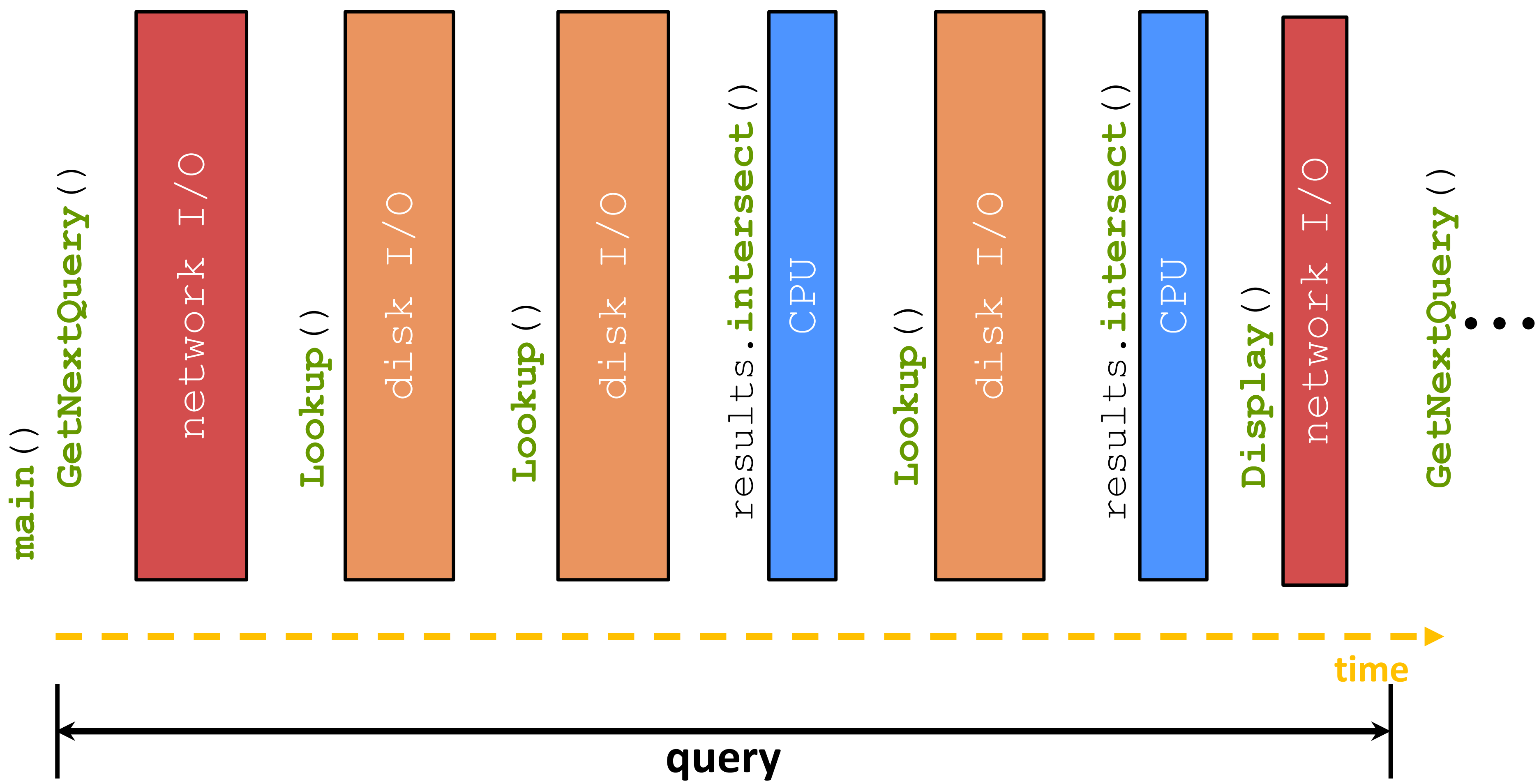
pollev.com/tqm

- This is pseudo code for what our single threaded server does.
- When do you think our code interacts with the network?
- How often does it read from a file?

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

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

Execution Timeline: a Multi-Word Query



Numbers Everyone Should Know

- There is a set of numbers that called “numbers everyone you should know”
- From Jeff Dean in 2009
- Numbers are out of date but the relative orders of magnitude are about the same

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 Zippy	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

Google

- More up to date numbers: https://colin-scott.github.io/personal_website/research/interactive_latency.html

What About I/O-caused Latency?

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

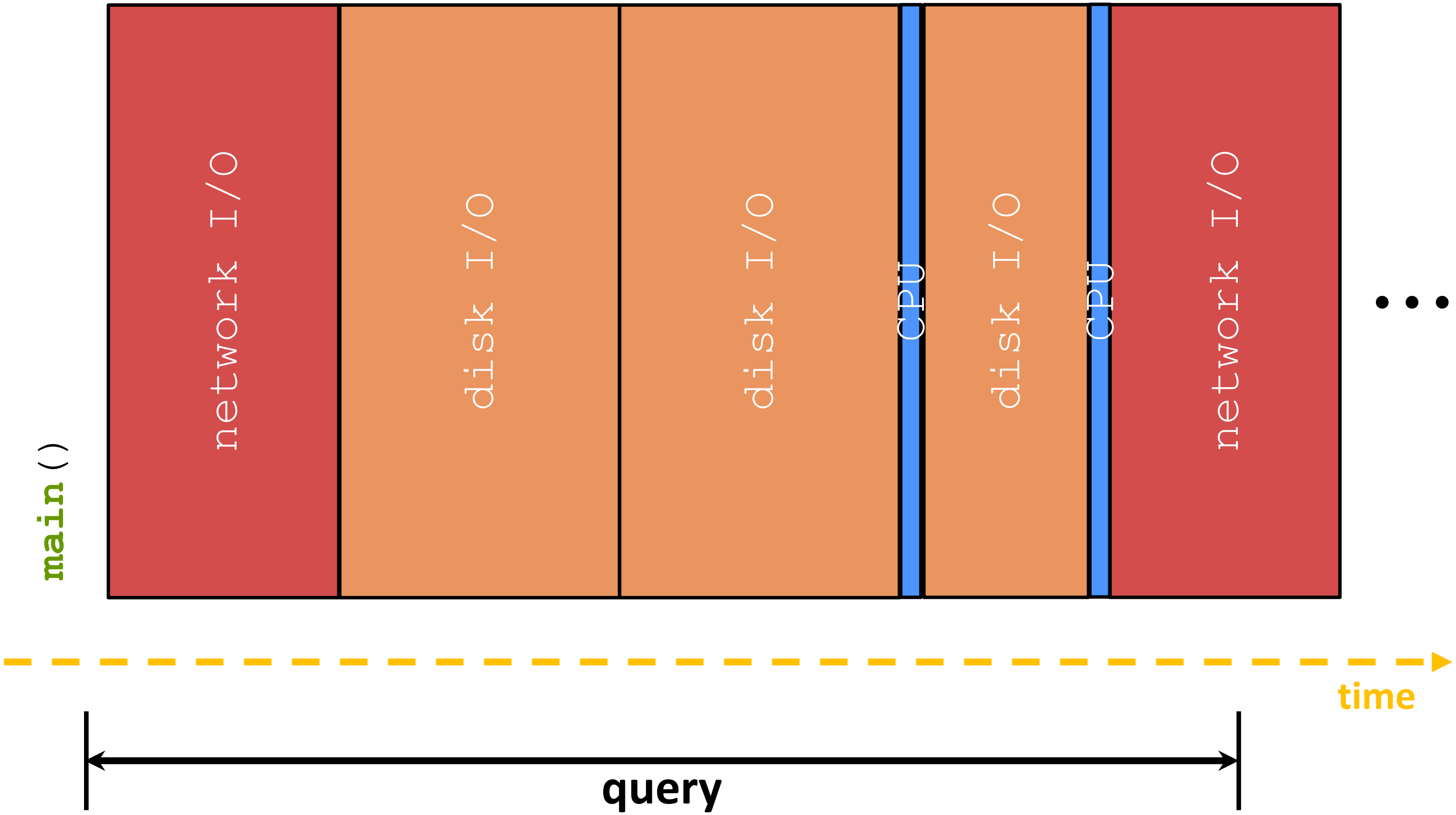
Numbers Everyone Should Know

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 Zippy	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

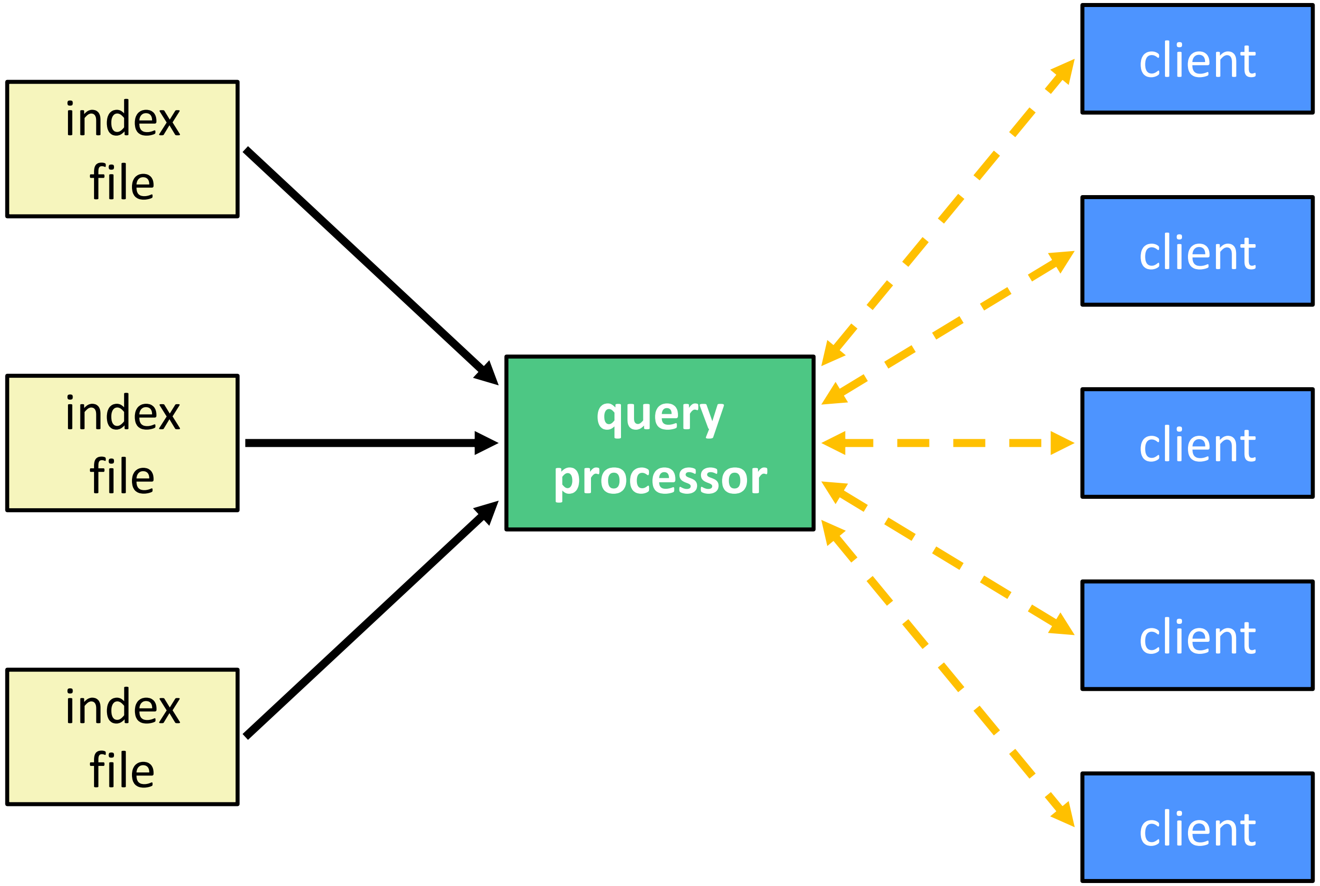
Does this look like efficient CPU Utilization?

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

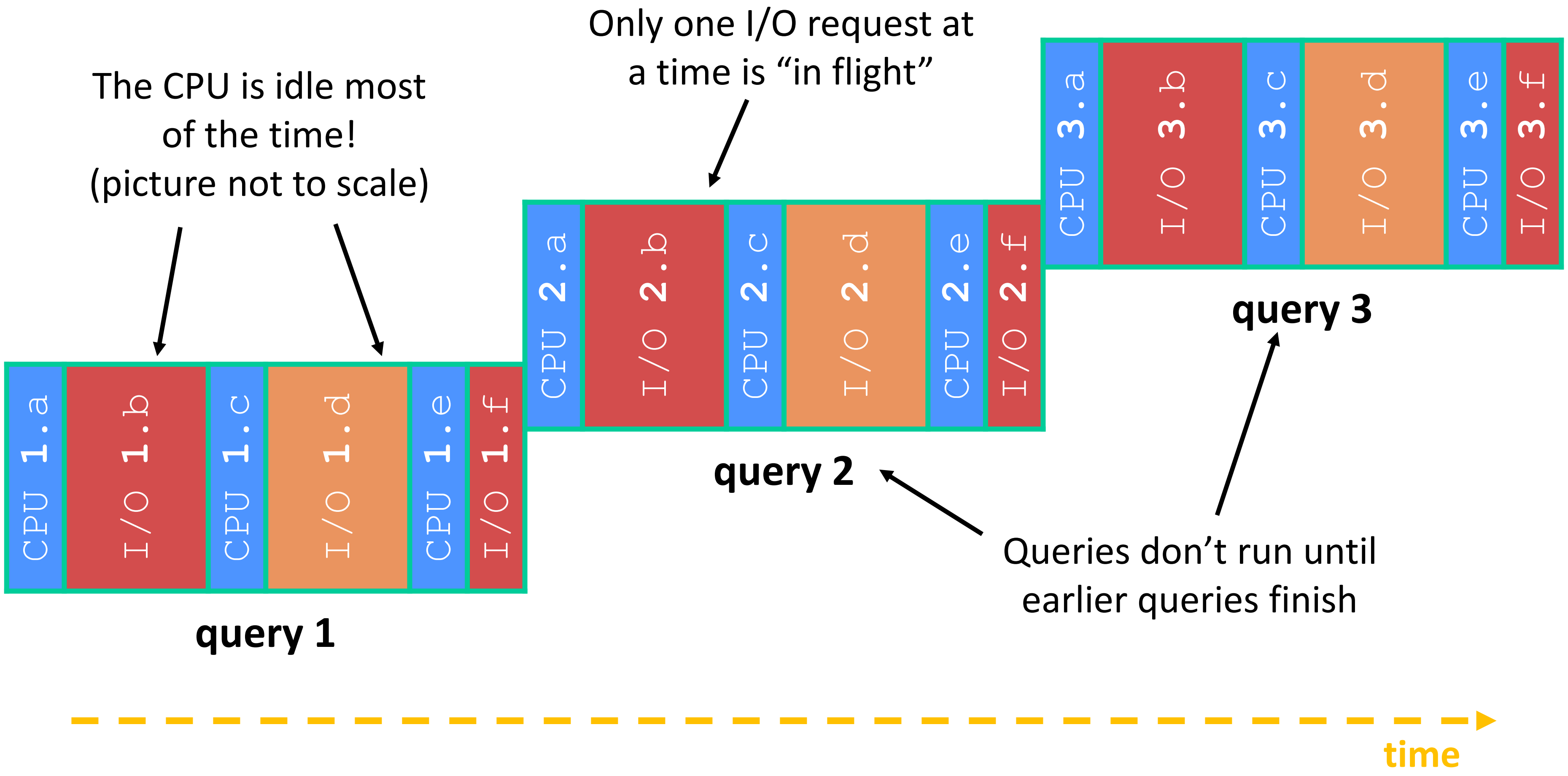


Uh-Oh: Handling Multiple Clients

What if we have multiple clients (and requests) happening at a time?
How are requests processed? One after the other….



Uh-Oh: Handling Multiple Clients



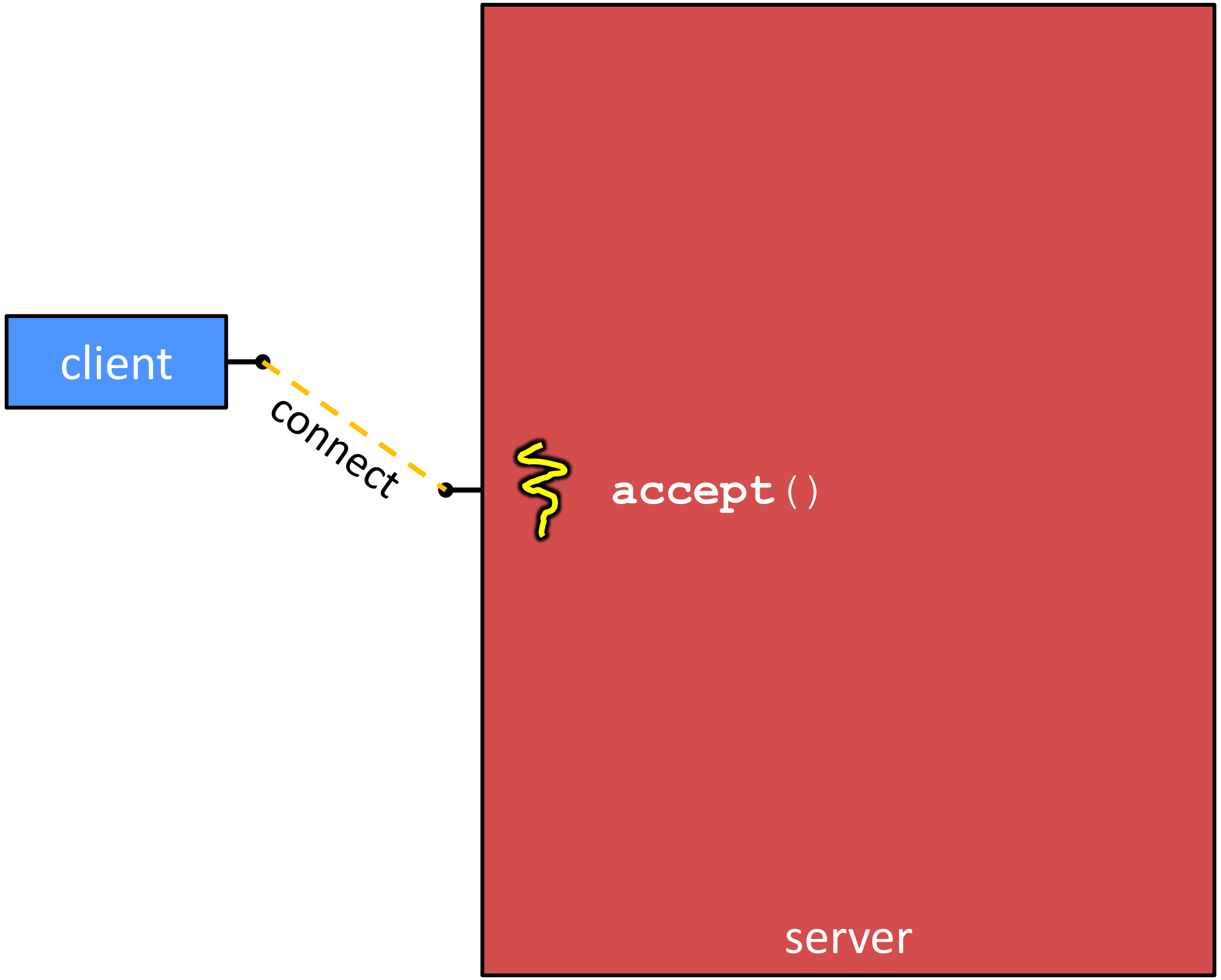
Sequential Can Be Inefficient

- 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.

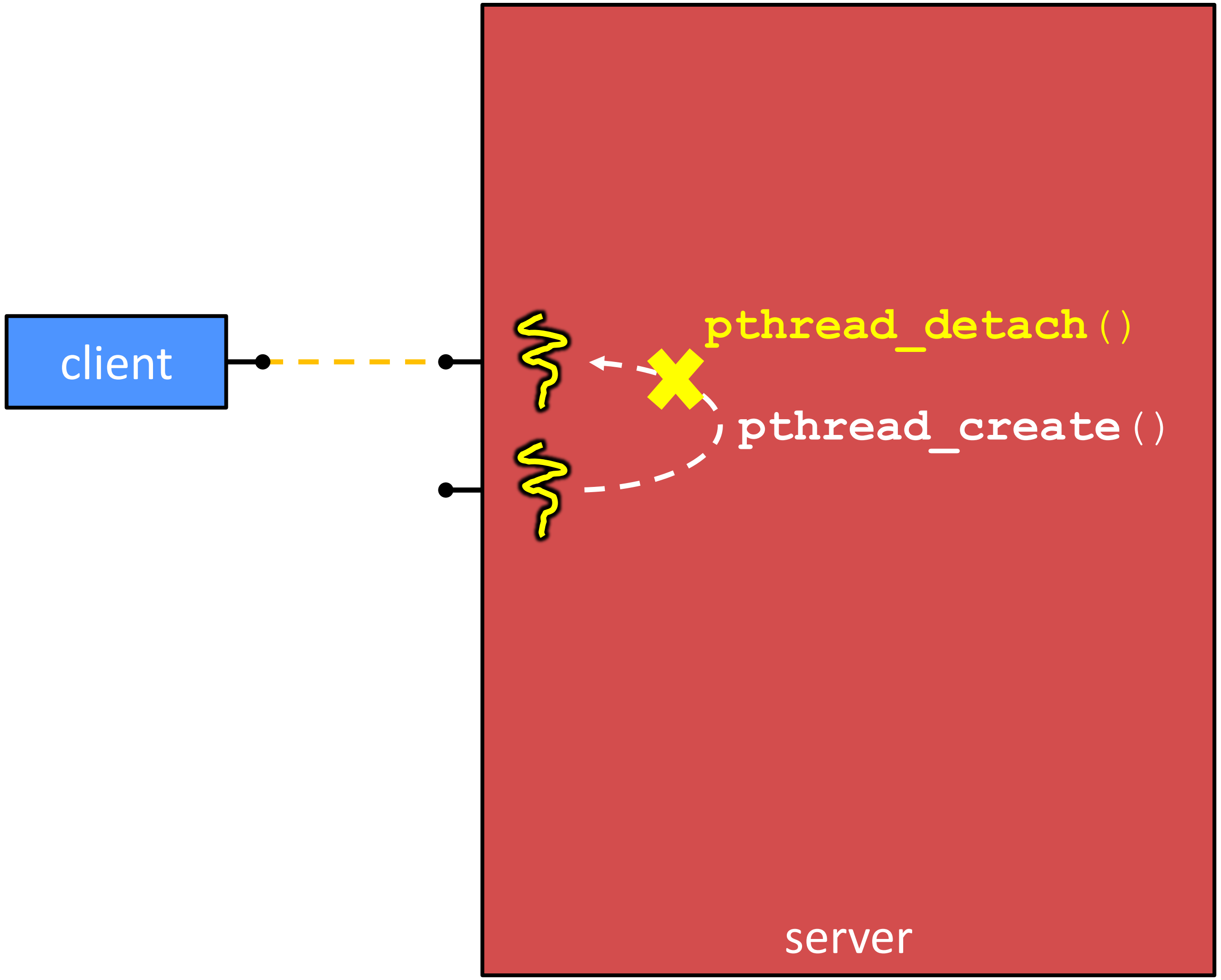
A Concurrent Implementation

- 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

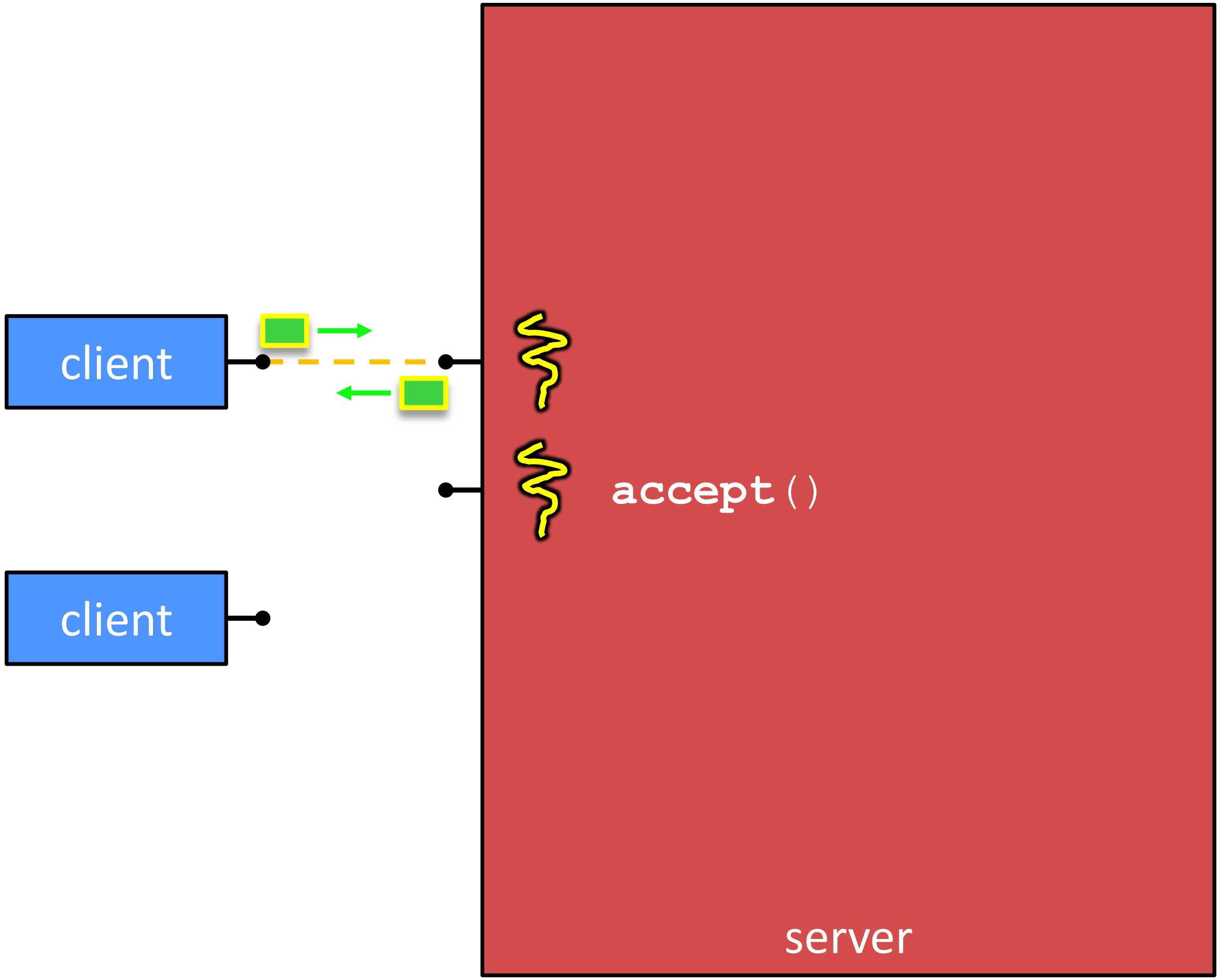
Multithreaded Server



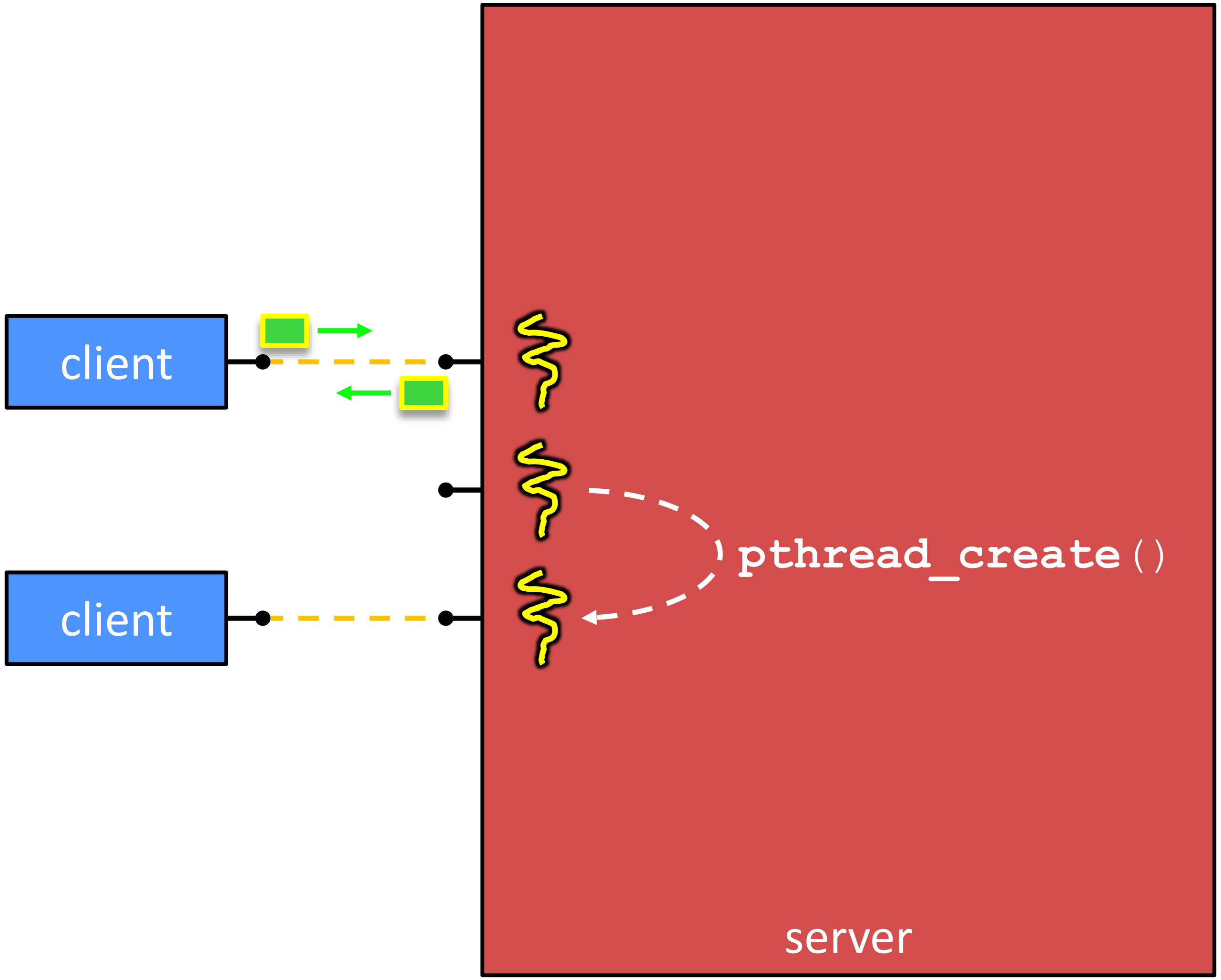
Multithreaded Server



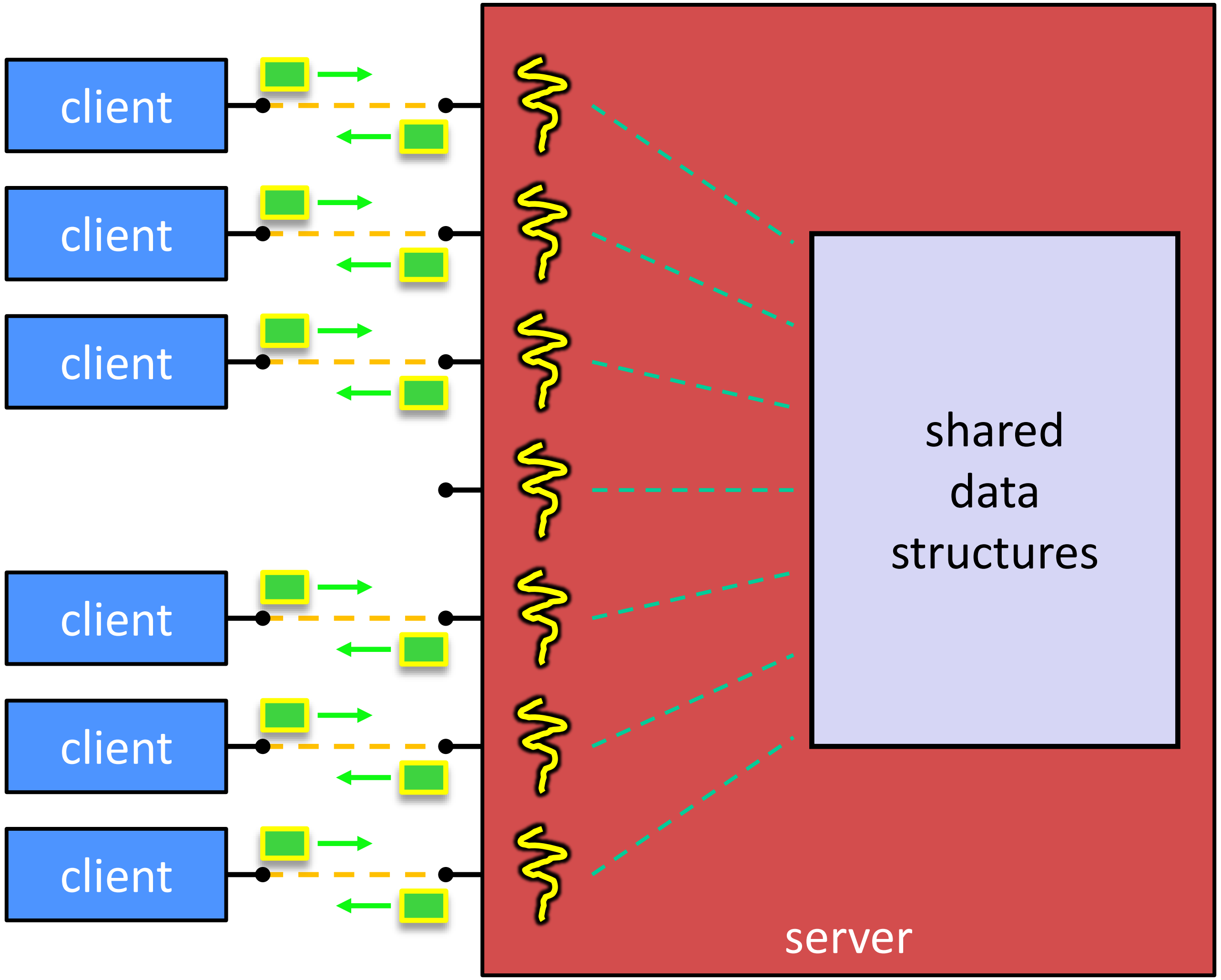
Multithreaded Server



Multithreaded Server



Multithreaded Server

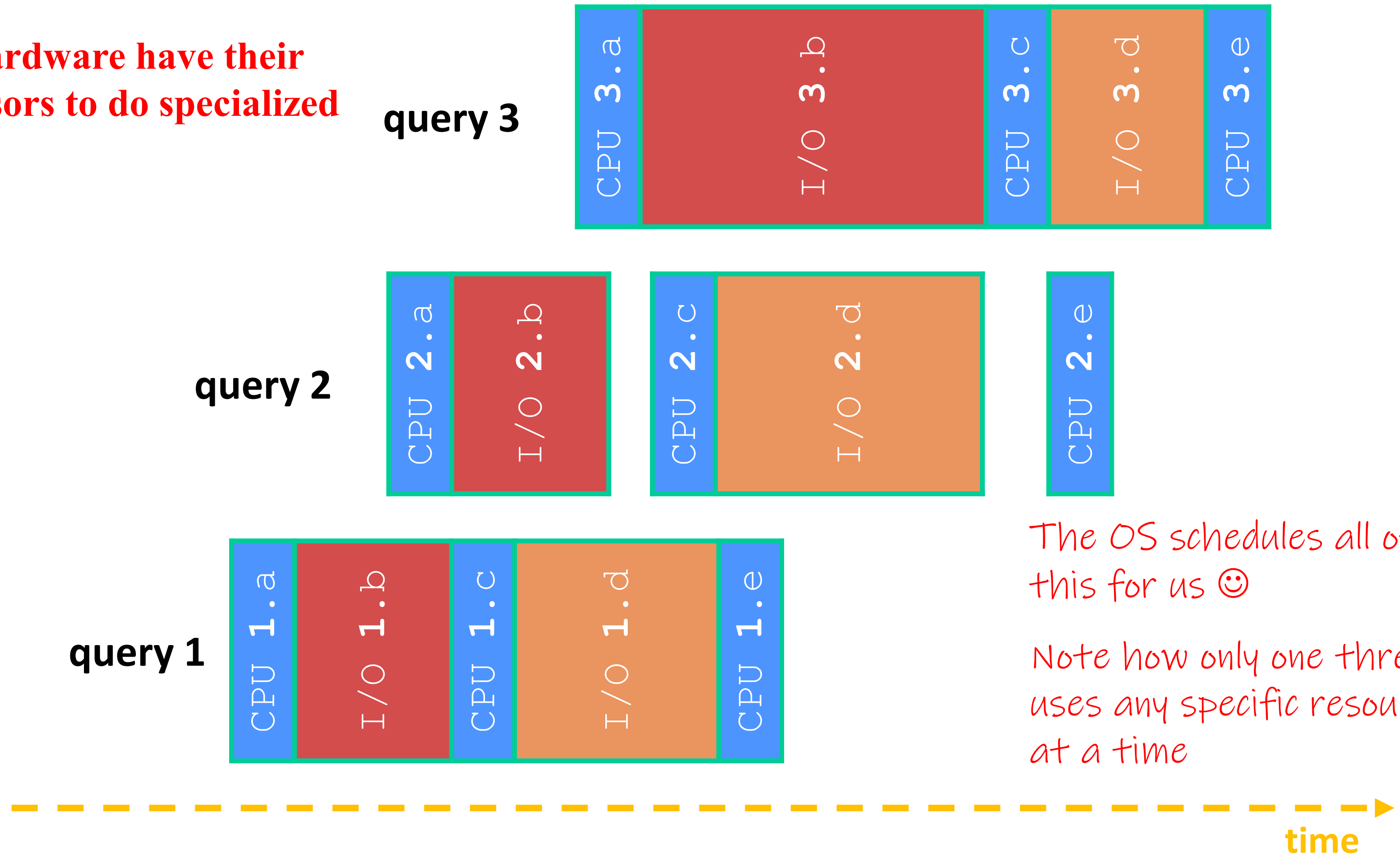


Multi-threaded Search Engine (Execution)

*Running with 1 CPU

The CPU is the <u>Central</u> 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?

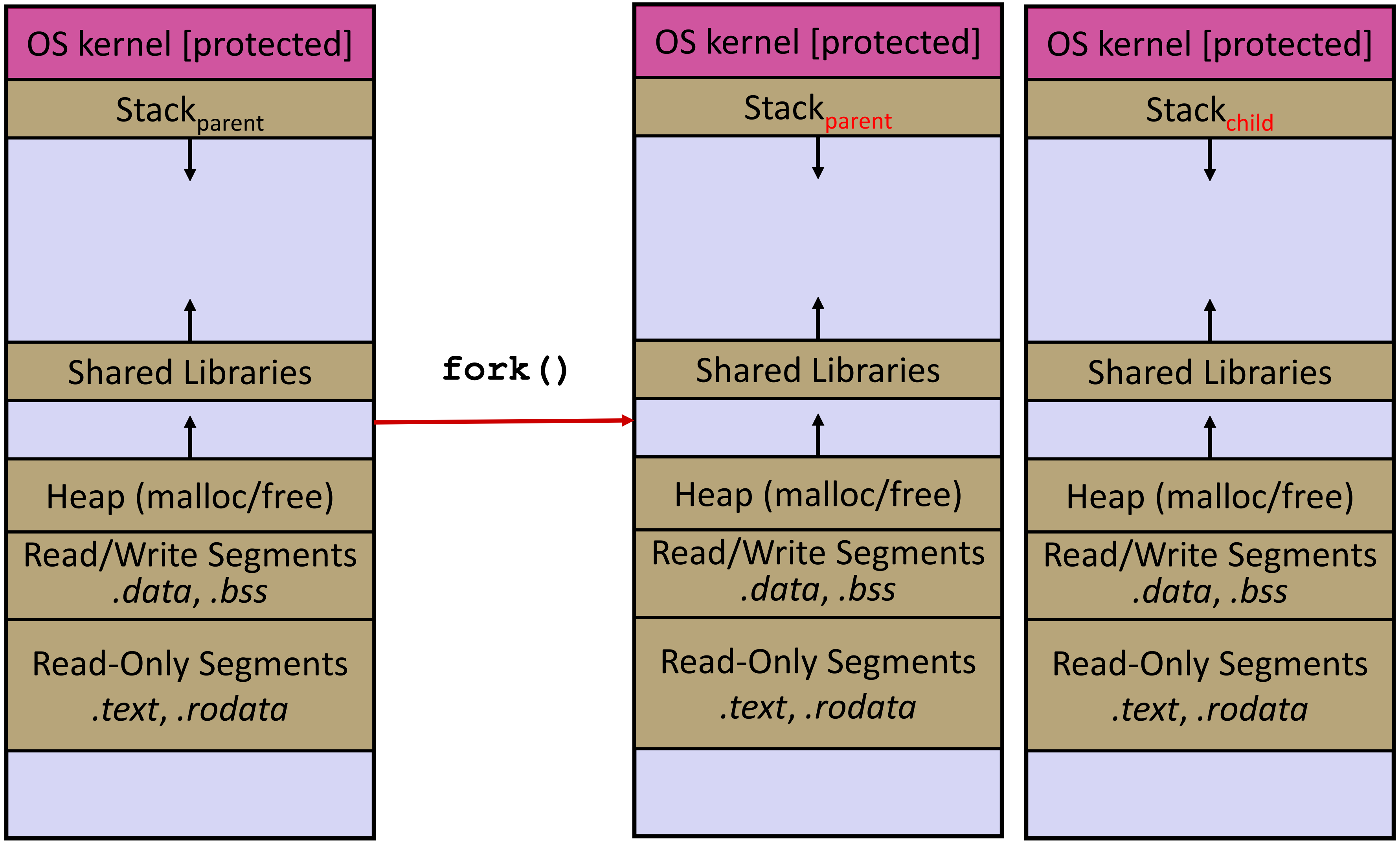
- Advantages:
  - You (mostly) write sequential-looking code
  - Threads can run in parallel if you have multiple CPUs/cores

- 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

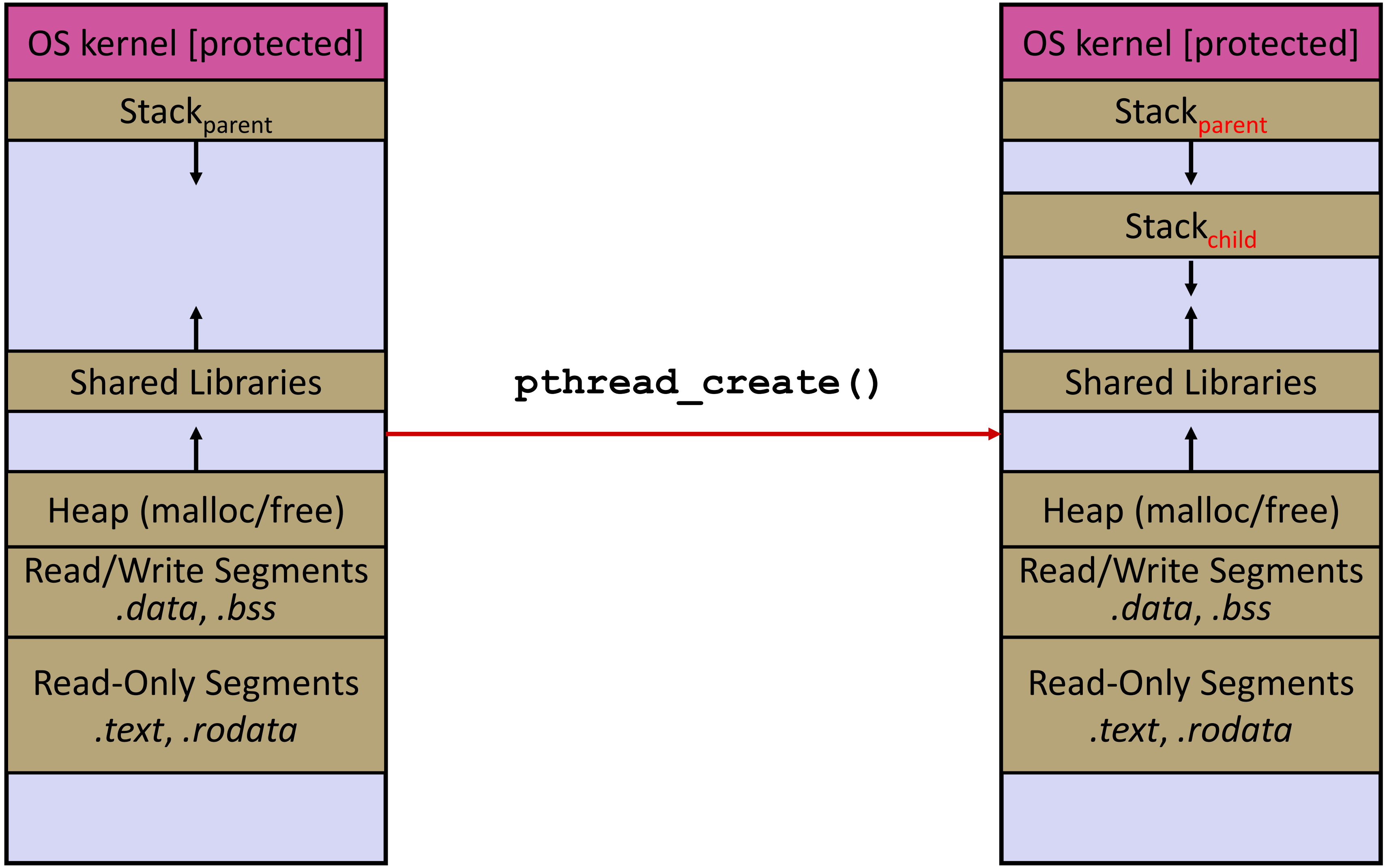
Threads vs. Processes

- In most modern OS's:
  - A Process has a unique: address space, OS resources,
    & security attributes
  - A Thread has a unique: stack, stack pointer, program counter,
    & registers
  - Threads are the *unit of scheduling* and processes are their *containers*; every process has at least one thread running in it

Threads vs. Processes



Threads vs. Processes



Alternative: Processes

- What if we forked processes instead of threads?

- Advantages:
  - No shared memory between processes
  - No need for language support; OS provides “fork”
  - Processes are isolated. If one crashes, other processes keep going

- Disadvantages:
  - More overhead than threads during creation and context switching (Context switching == switching between threads/processes)
  - Cannot easily share memory between processes – typically communicate through the file system

That's all!

- See you next time!