Intro to Threads

Computer Operating Systems, Spring 2025

Instructors: Joel Ramirez Travis McGaha

Head TAs: Ash Fujiyama Emily Shen Maya Huizar

TAs:

Ahmed Abdellah
Akash Kaukuntla
Alexander Cho
Alicia Sun
August Fu
Bo Sun
Connor Cummings
Eric Zou
Haoyun Qin
Jonathan Hong
Joy Liu
Khush Gupta
Kyrie Dowling
Rafael Sakamoto
Sarah Zhang
Susan Zhang
Vedansh Goenka
Vivi Li
Yousef AlRabiah
Yu Cao
Zihao Zhou

Poll Everywhere

pollev.com/cis5480

- How are you doing? How do you like processes?

Administrivia

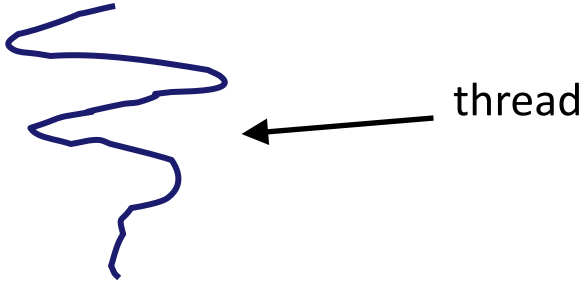
- Penn-shell is out (this shouldn't be news)!
  - ***Full thing is due (Fri, Feb 28) (1 more week!)***
  - ***Done in partners***
    - Everything was covered already that you would need…

Lecture Outline

- **Threads High Level**
- pthreads
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency

Introducing Threads

- Separate the concept of a process from the "*thread of execution*"
  - Threads are contained within a process
  - Usually called a thread, this is a sequential execution stream within a process

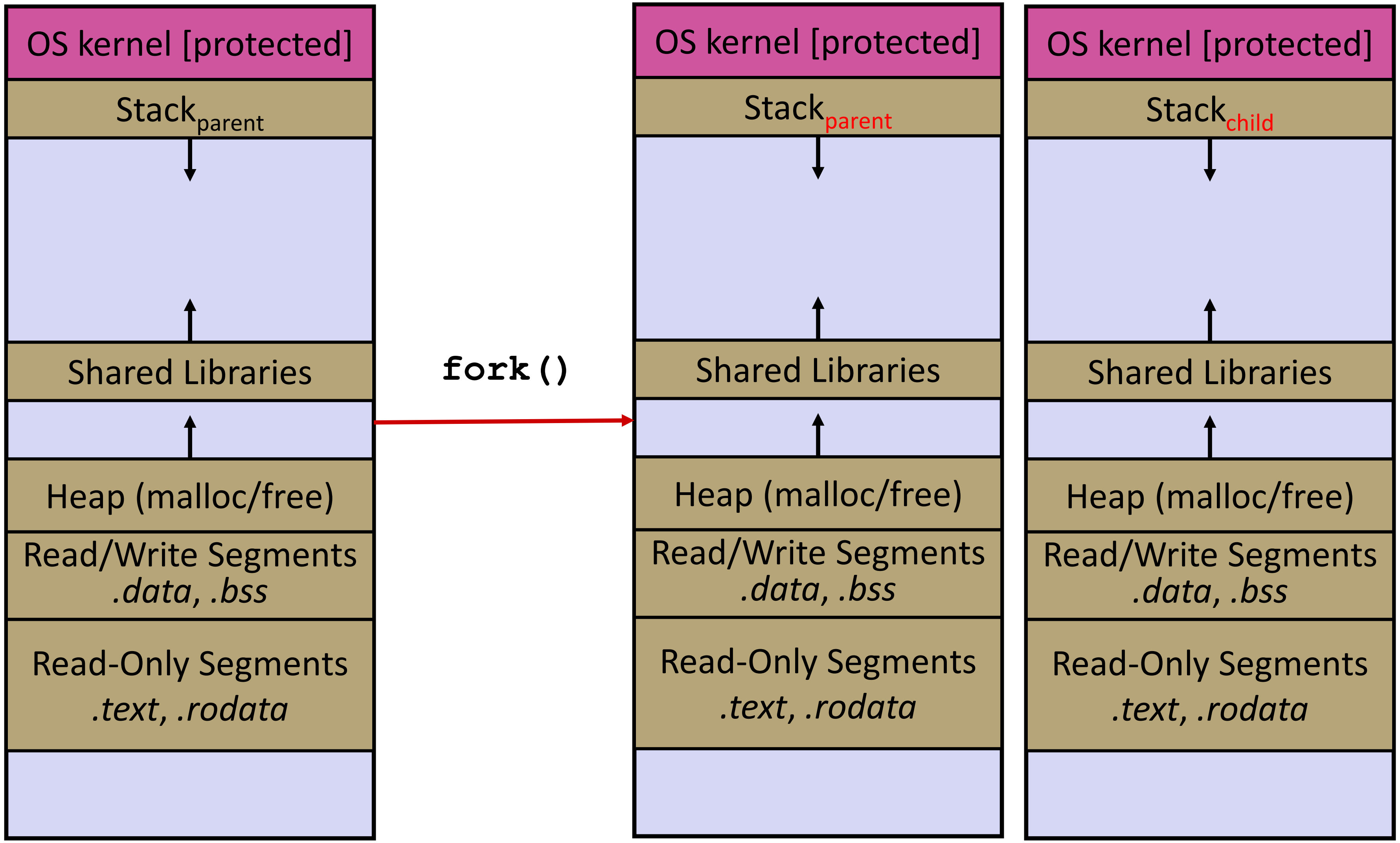


- In most modern OS's:
  - Threads are the *unit of scheduling.*

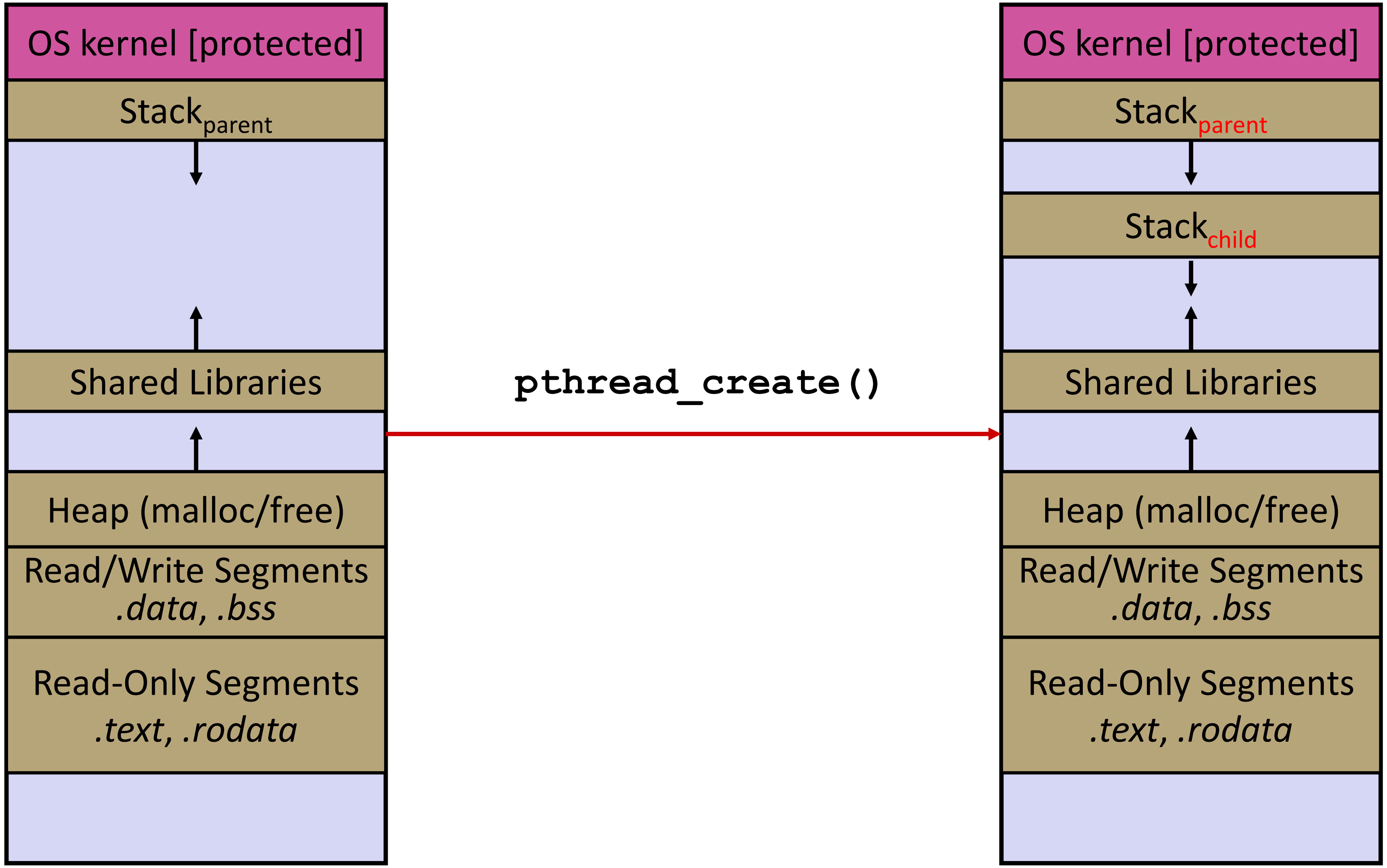
Threads vs. Processes

- 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

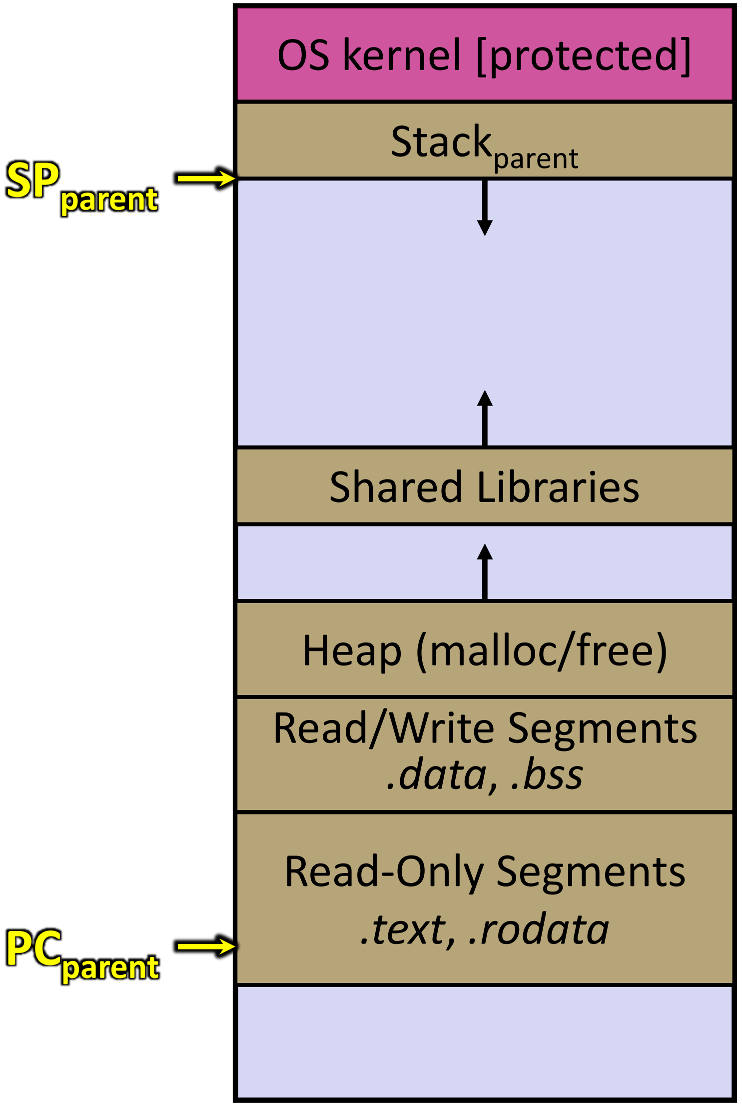


Threads

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

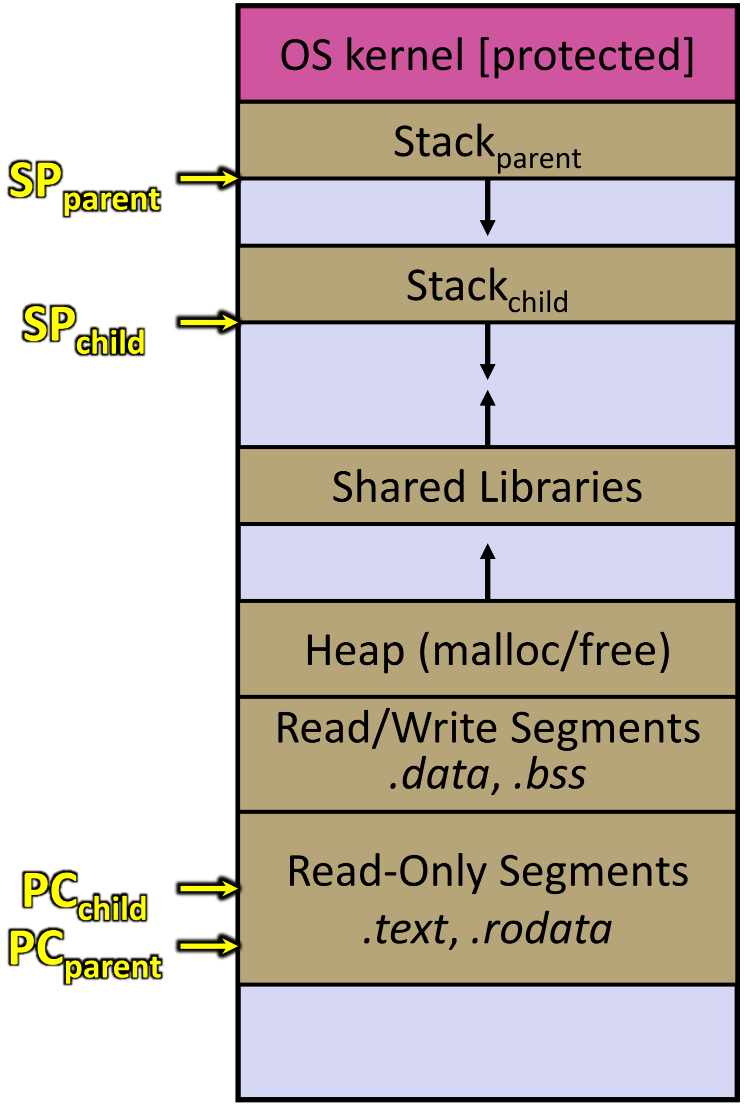
- Analogy: restaurant kitchen
  - Kitchen is process
  - Chefs are threads

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

Lecture Outline

- Threads High Level
- **pthreads**
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency

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

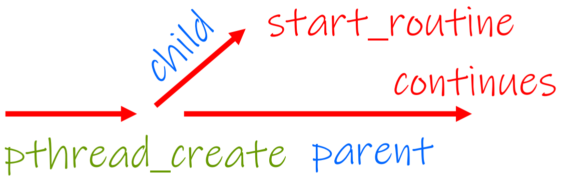
Output parameter.
Gives us a "thread_descriptor"

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

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

  Argument for the thread function

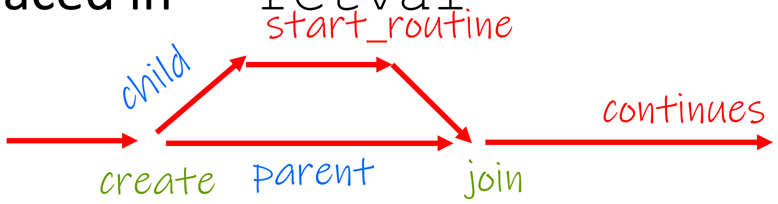
  - Creates a new thread into `*thread`, with attributes `*attr` (`NULL` means default attributes)
  - Returns **0** on success and an error number on error (can check against error constants)
  - The new thread runs **`start_routine`**`(arg)`



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



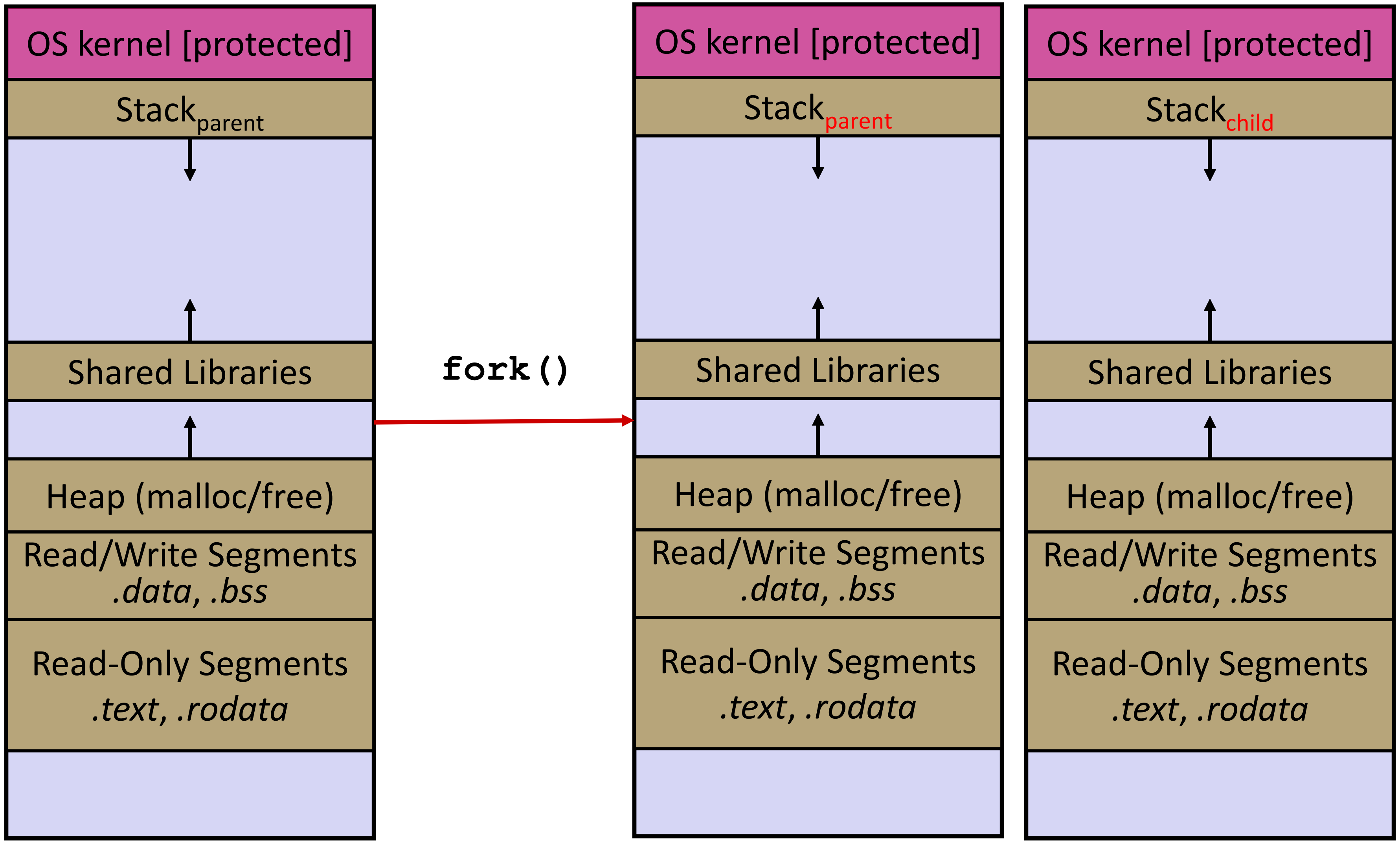
Thread Example

- See `cthreads.c`
    - How do you properly handle memory management?
        - Who allocates and deallocates memory?
        - How long do you want memory to stick around?
    - Threads execute in parallel

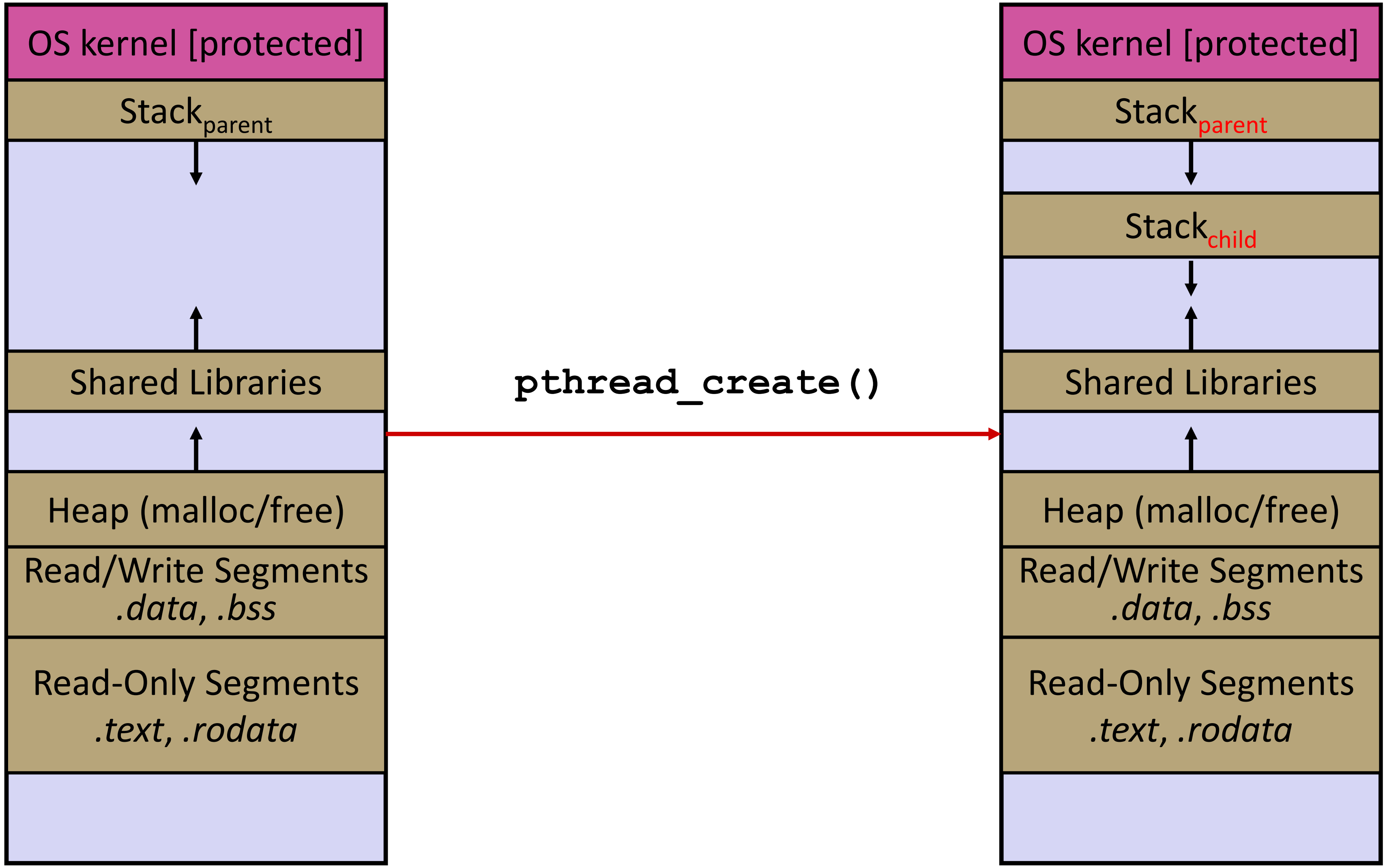
Lecture Outline

- Threads High Level
- pthreads
- **Processes vs threads**
- Thread Interleaving & Sequential Consistency
- Benefits of Concurrency

Threads vs. Processes



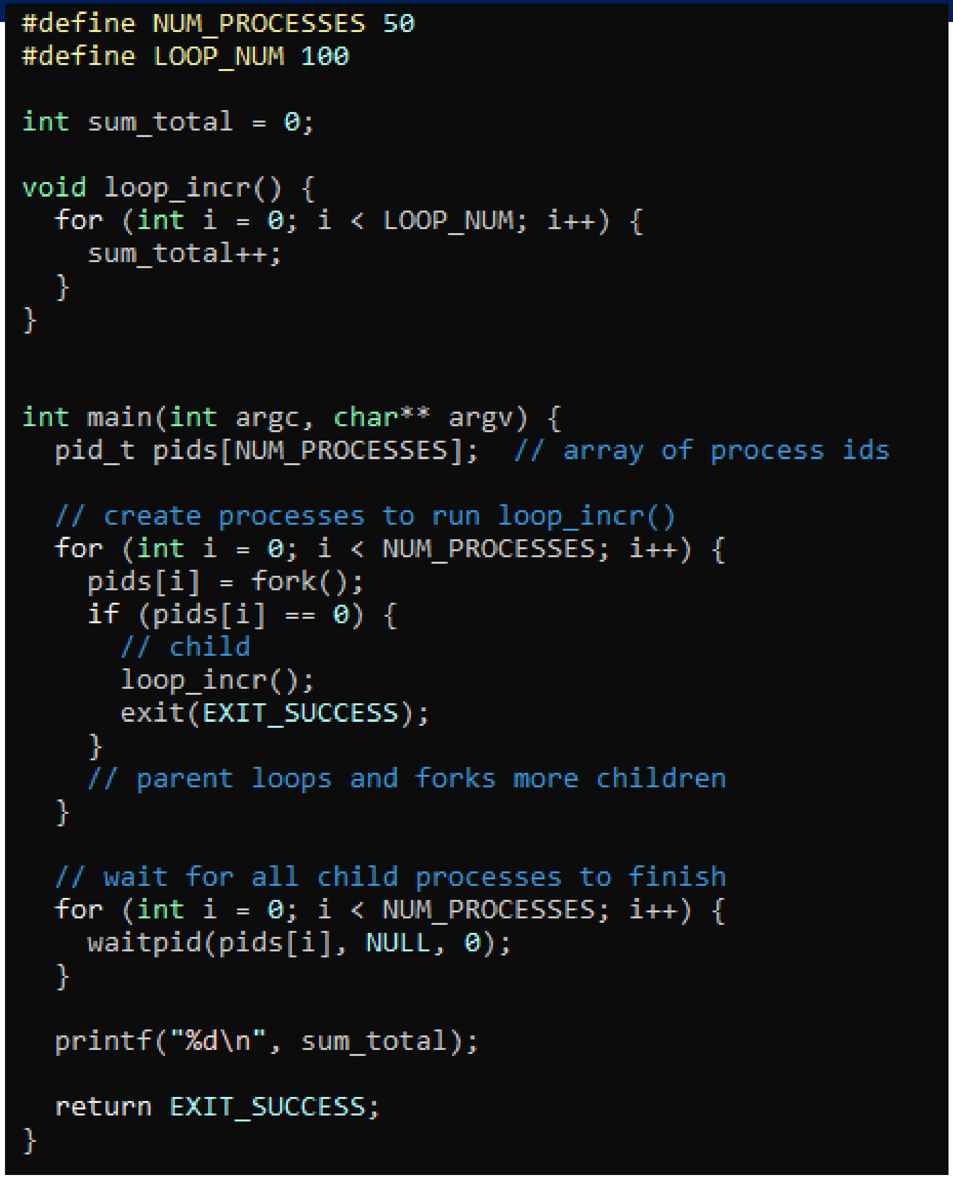
Threads vs. Processes



Poll Everywhere

pollev.com/tqm

- What does this print?

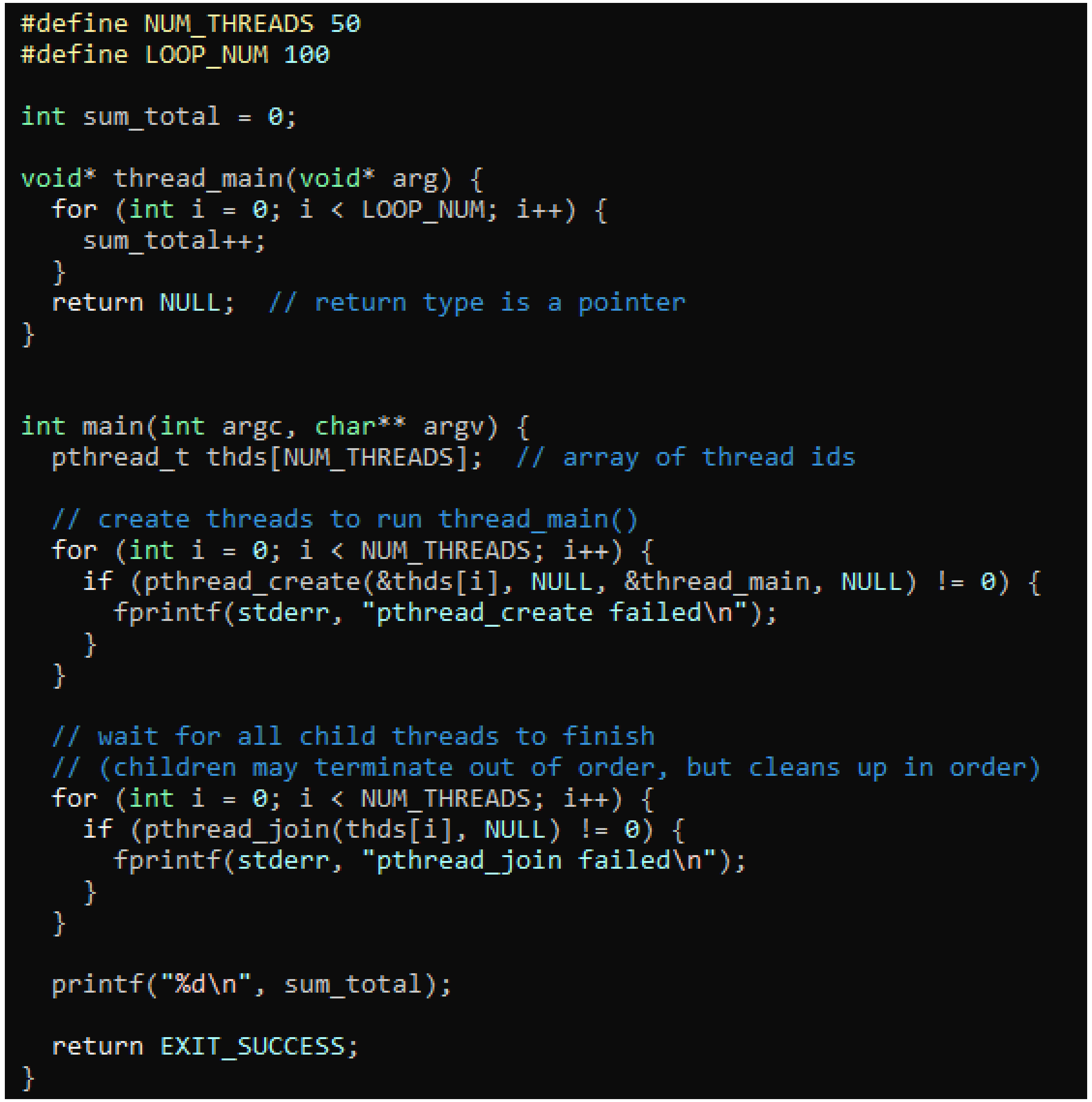


```
#define NUM_PROCESSES 50
#define LOOP_NUM 100

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;
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?



```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;
}
```

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 next week**

Process Isolation

- Process Isolation is a set of mechanisms implemented to protect processes from each other and protect the kernel from user processes.
    - Processes have separate address spaces
    - Processes have privilege levels to restrict access to resources
    - If one process crashes, others will keep running

- Inter-Process Communication (IPC) is limited, but possible
    - Pipes via pipe()
    - Sockets via socketpair()
    - Shared Memory via shm_open()

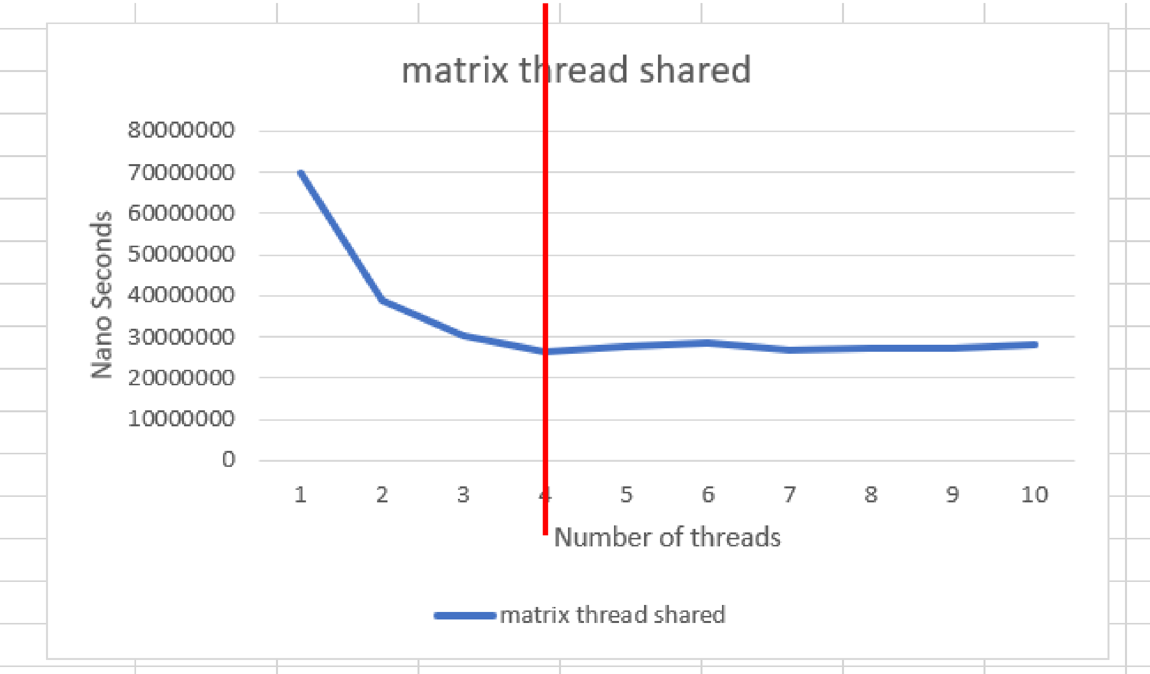
Parallelism

- You can gain performance by running things in parallel
  - Each thread can use another core

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix

Parallelism

- I have a 3800 x 3800 integer matrix, and I want to count the number of odd integers in the matrix
- I can speed this up by giving each thread a part of the matrix to check!
  - **Works with threads since they share memory**



Diminishing returns

After 4 threads, no gain in speed

why? Machine run on only has 4 cores

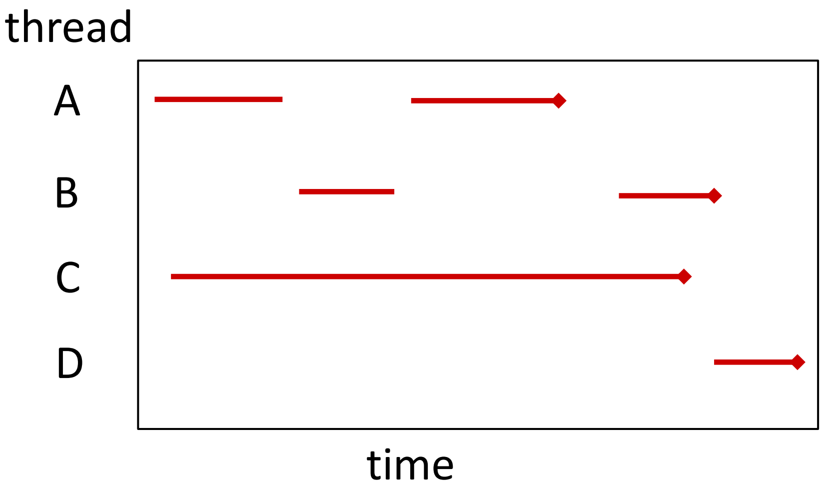
Parallelism vs Concurrency

- Two commonly used terms (often mistakenly used interchangeably).

- Concurrency: When there are one or more “tasks” that have overlapping lifetimes (between starting, running and terminating).
  - That these tasks are both running within the same **period**.

- Parallelism: when one or more “tasks” run at the same **instant** in time.

- Consider the lifetime of these threads. Which are concurrent with A? Which are parallel with A?



How fast is fork()?

- ~ 0.5 milliseconds per fork*
- ~ 0.05 milliseconds per thread creation*
  - 10x faster than fork()

- *Past measurements are not indicative of future performance – depends on hardware, OS, software versions, …
  - Processes are known to be even slower on Windows

Context Switching

- Processes are considered “more expensive” than threads. There is more overhead to enforce isolation

- Advantages:
    - No shared memory between processes
    - Processes are isolated. If one crashes, other processes keep going

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

Lecture Outline

- Threads High Level
- pthreads
- Processes vs threads
- **Thread Interleaving & Sequential Consistency**
- Benefits of Concurrency

Poll Everywhere

pollev.com/tqm

- What are all possible outputs of this program?

```
void* thrd_fn(void* arg) {
  int* ptr = (int*) arg;
  printf("%d\n", *ptr);
  return NULL;
}

int main() {
  pthread_t thd1;
  pthread_t thd2;
  int x = 1;
  pthread_create(&thd1, NULL, thrd_fn, &x);
  x = 2;
  pthread_create(&thd2, NULL, thrd_fn, &x);

  pthread_join(thd1, NULL);
  pthread_join(thd2, NULL);
}
```

Are these output possible?

```
----------------------------
1
2
----------------------------
2
2
----------------------------
1
1
----------------------------
2
1
```

Visualization

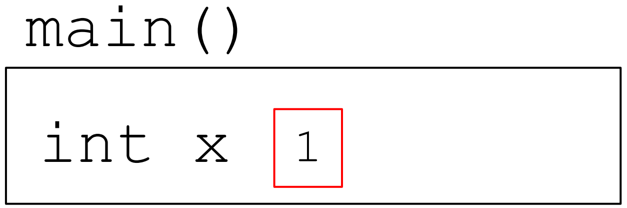
```
int main() {
  int x = 1;
  pthread_create(...);
  x = 2;
  pthread_create(...);

  pthread_join(...);
  pthread_join(...);
}
```

```
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

```
thrd_fn() {
  printf(*ptr);
  return NULL;
}
```

Visualization: Memory

- The variable x is shared across all threads.



```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Visualization: Memory

- The variable x is shared across all threads.

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

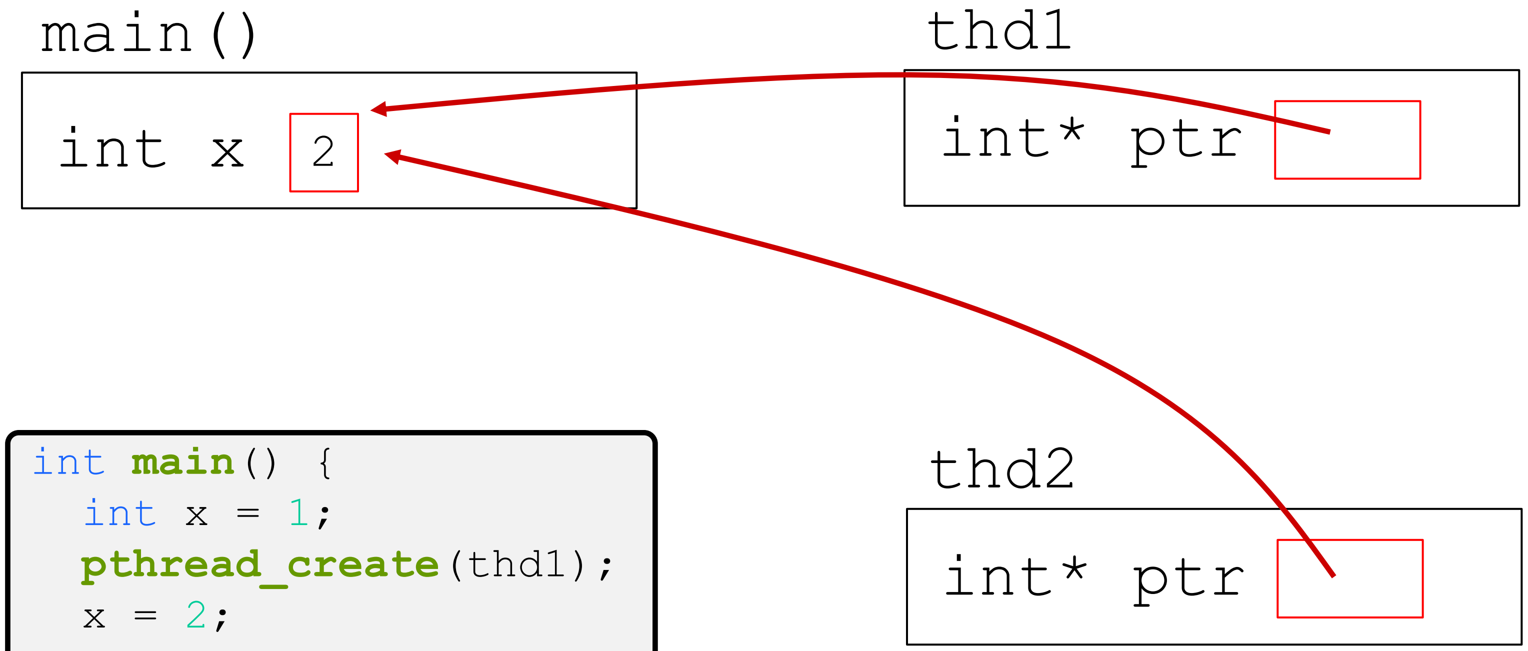
Visualization: Memory

- The variable x is shared across all threads.

```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Visualization: Memory

- The variable x is shared across all threads.



```
int main() {
  int x = 1;
  pthread_create(thd1);
  x = 2;
  pthread_create(thd2);

  pthread_join(thd1);
  pthread_join(thd2);
}
```

Sequential Consistency

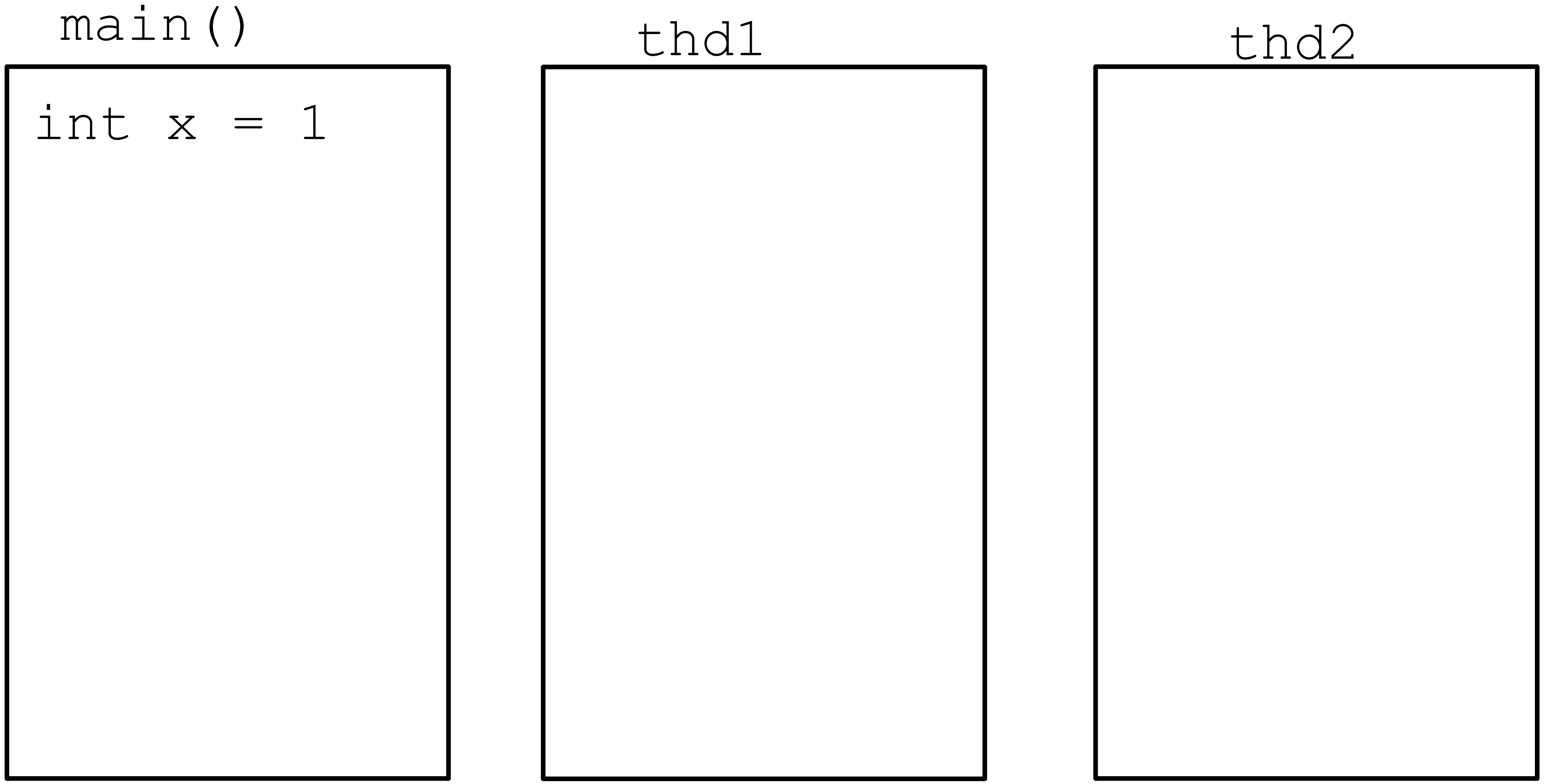
- Within a single thread, we assume* that there is sequential consistency. That the order of operations within a single thread are the same as the program order.

```
main()

int x = 1
create thd1
x = 2
create thd2
```

Within main(), x is set to 1 before thread 1 is created
then thread 1 is created
then x is set to 2
then thread 2 is created

Visualization: Ordering

- Threads run concurrently; we can’t be sure of the ordering of things across threads.



`main()`

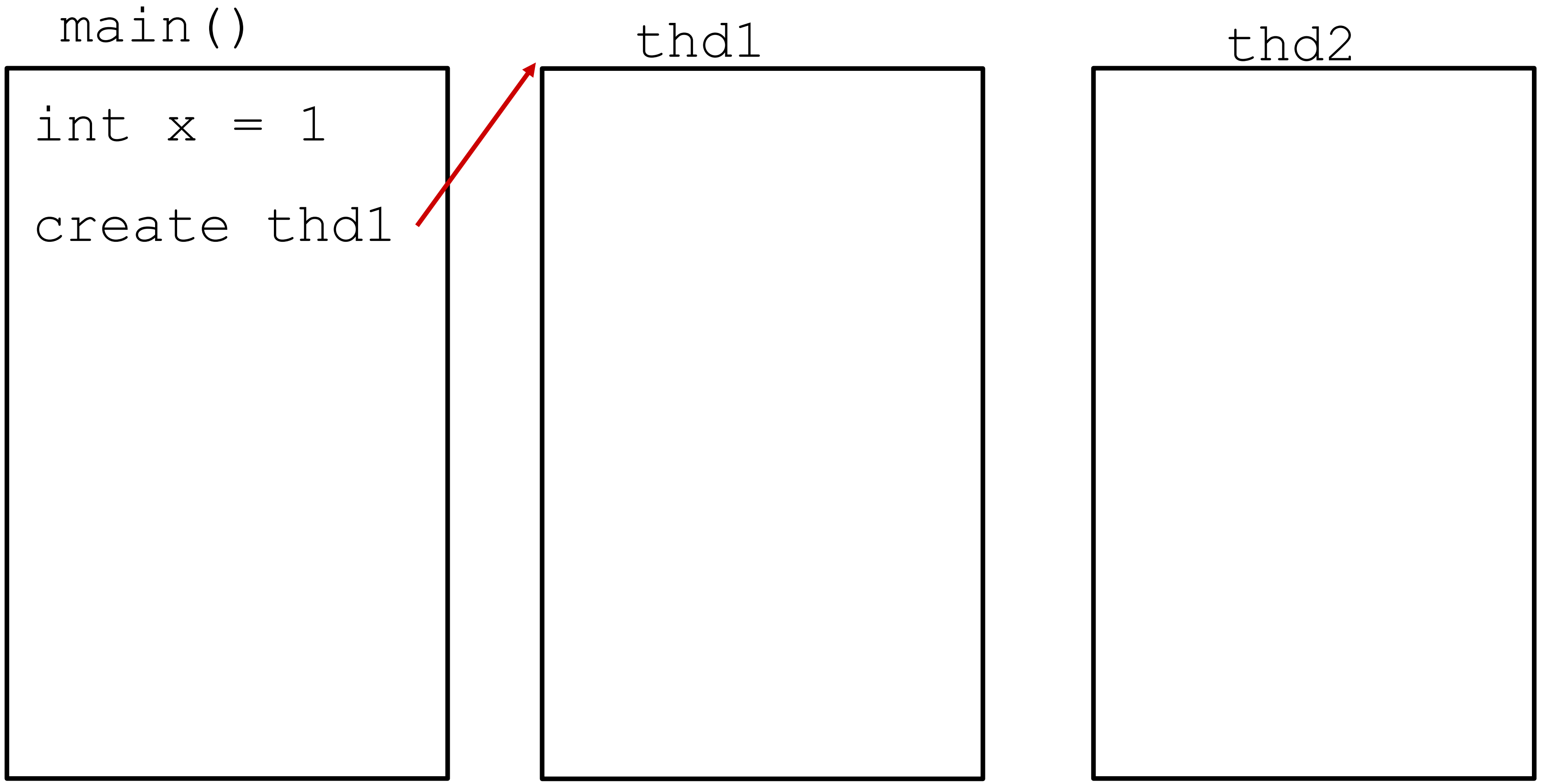
`int x = 1`

`thd1`

`thd2`

Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



main()

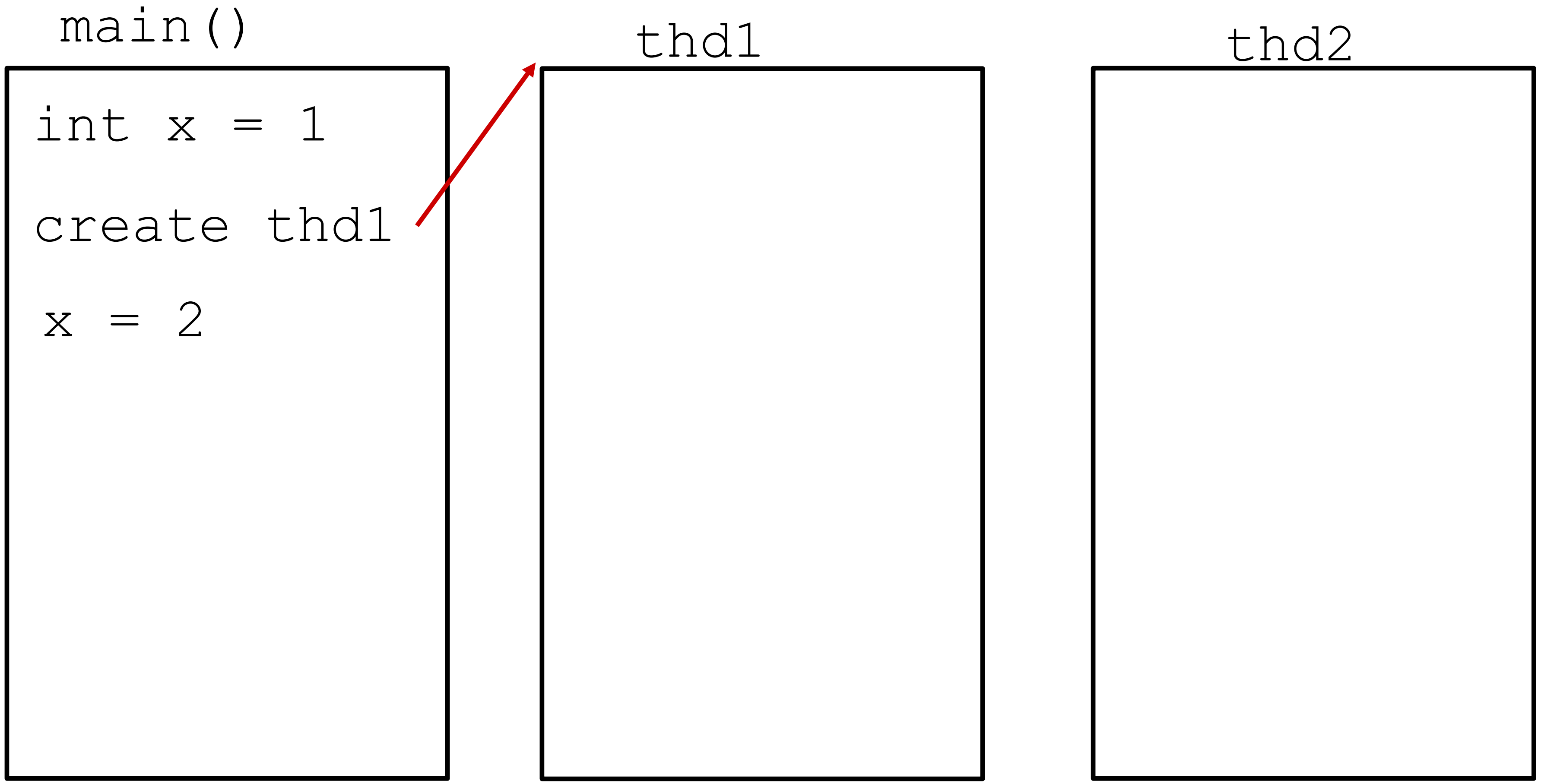
```
int x = 1

create thd1
```

thd1

thd2

Visualization: Ordering

- Threads run concurrently; we can't be sure of the ordering of things across threads.



main()

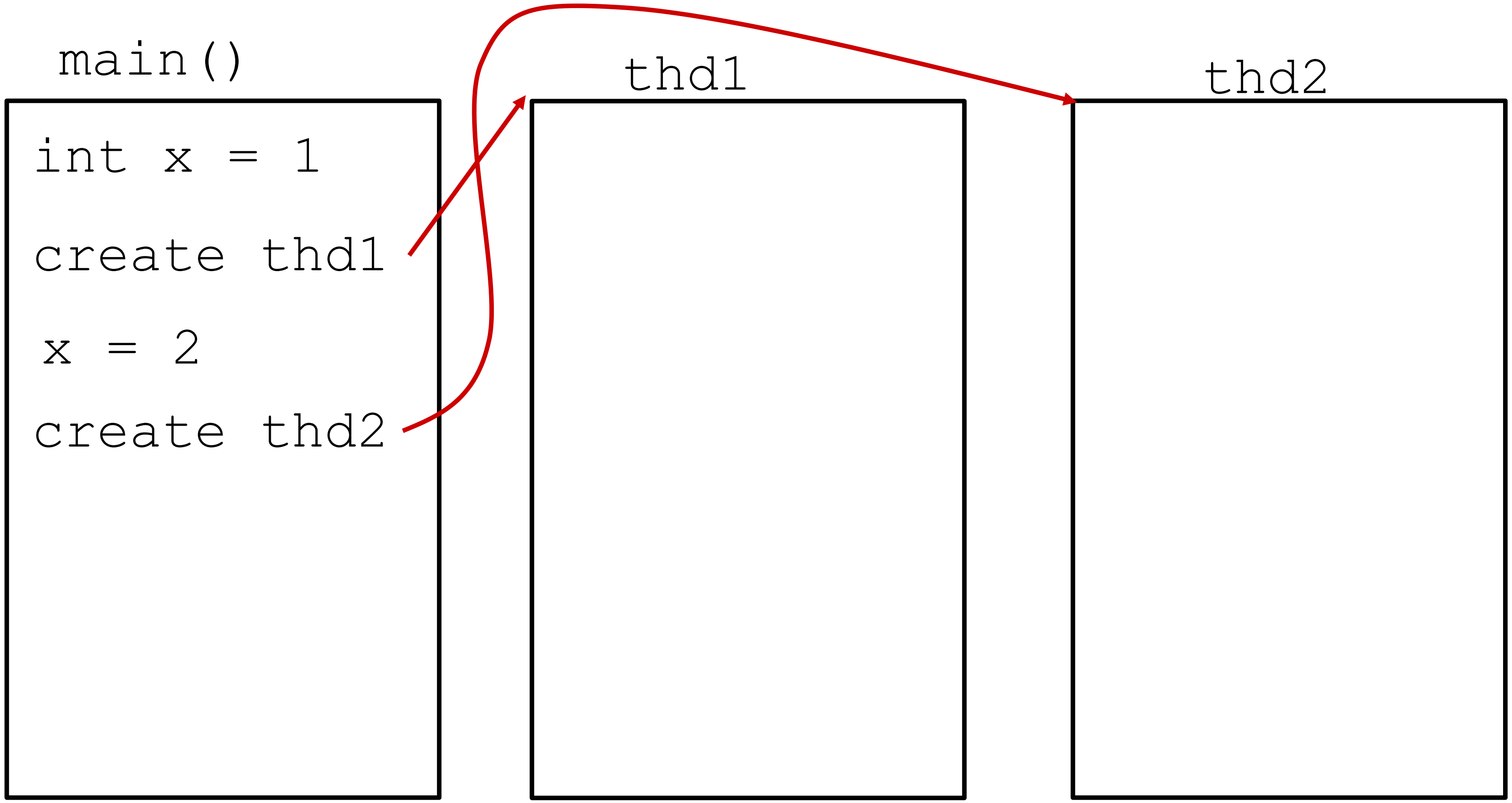
```
int x = 1

create thd1

x = 2
```

thd1

thd2

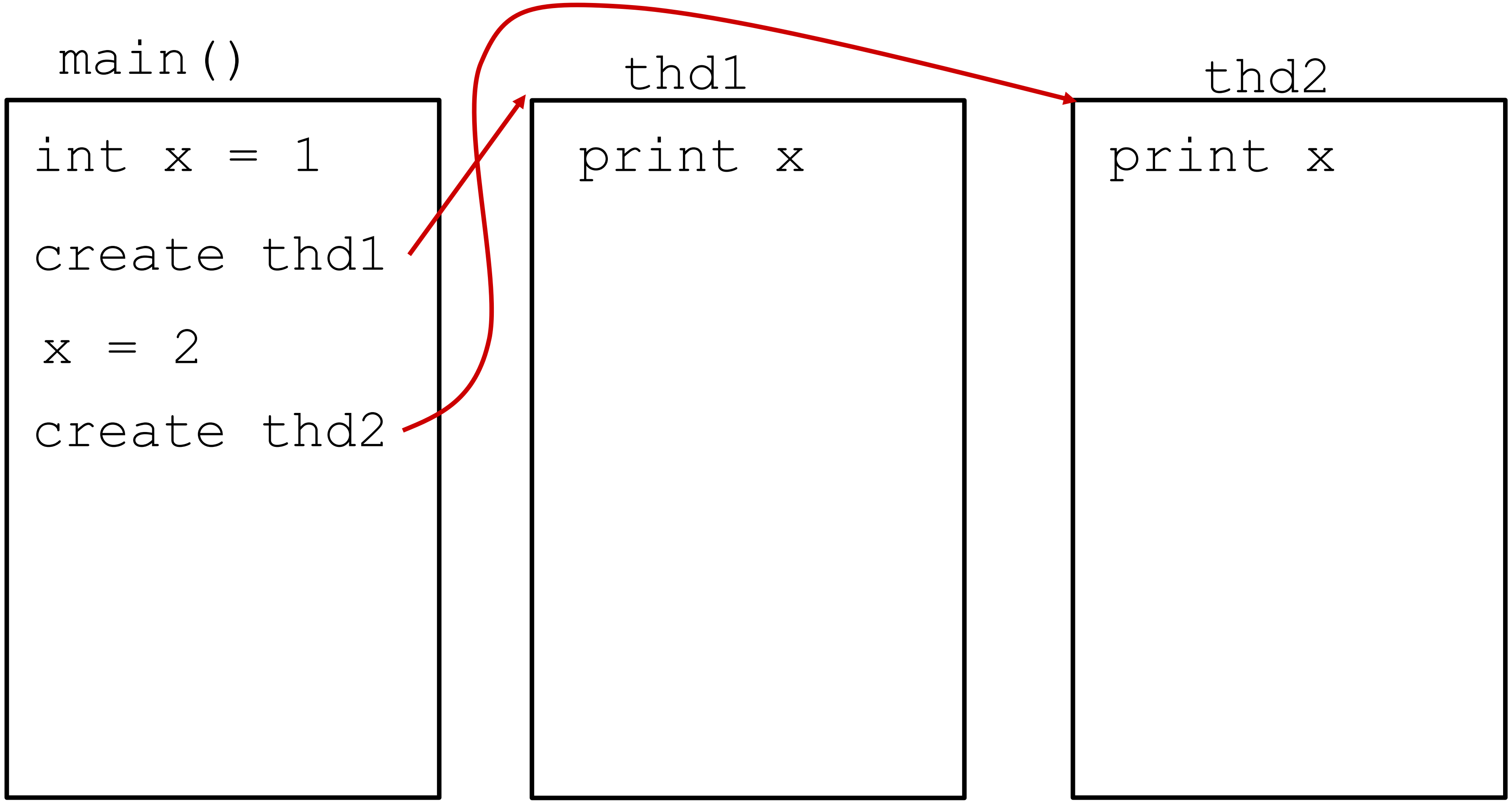
Visualization: Ordering

- Threads run concurrently; we can’t be sure of the ordering of things across threads.



Visualization: Ordering

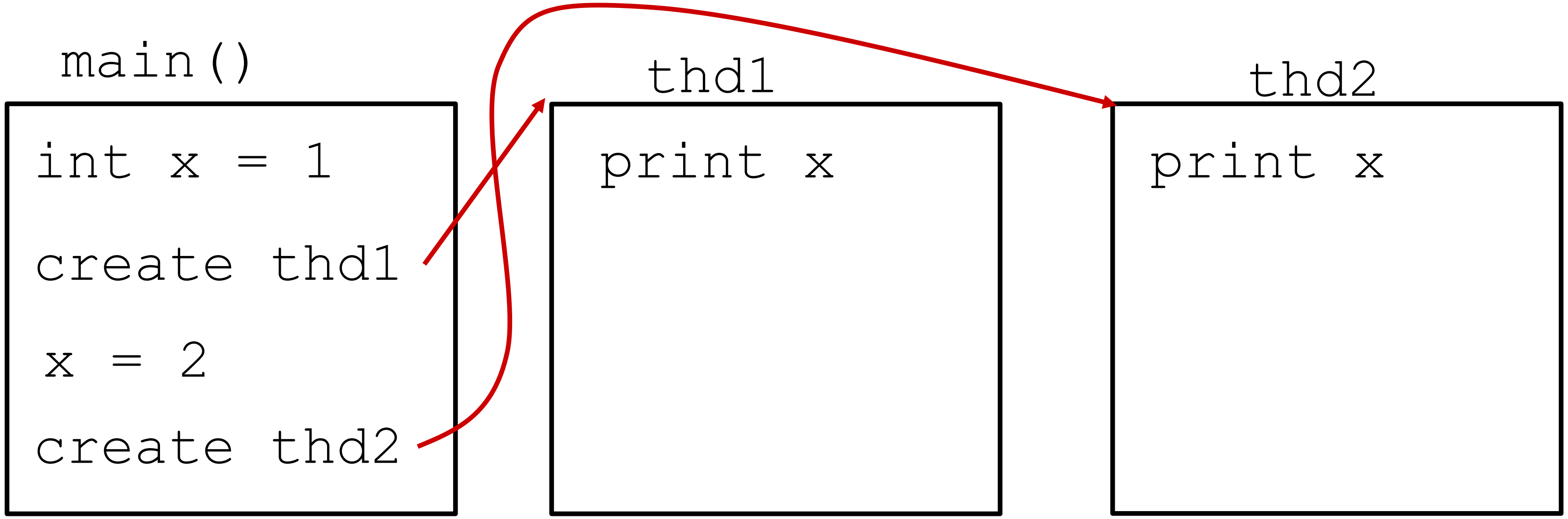
- Threads run concurrently; we can't be sure of the ordering of things across threads.



Visualization: Ordering

This is also why total.c malloc'd individual integers for each thread.
Though it could have also just made an array on the stack

- Threads run concurrently; we can't be sure of the ordering of things across threads.



main()

```
int x = 1
create thd1
x = 2
create thd2
```

thd1

```
print x
```

thd2

```
print x
```

We know that x is initialized to 1 before thd1 is created
We know that x is set to 2 and thd1 is created before thd2 is created

Anything else that we know? **No**. Beyond those statements, we do not know the ordering of main and the threads running.

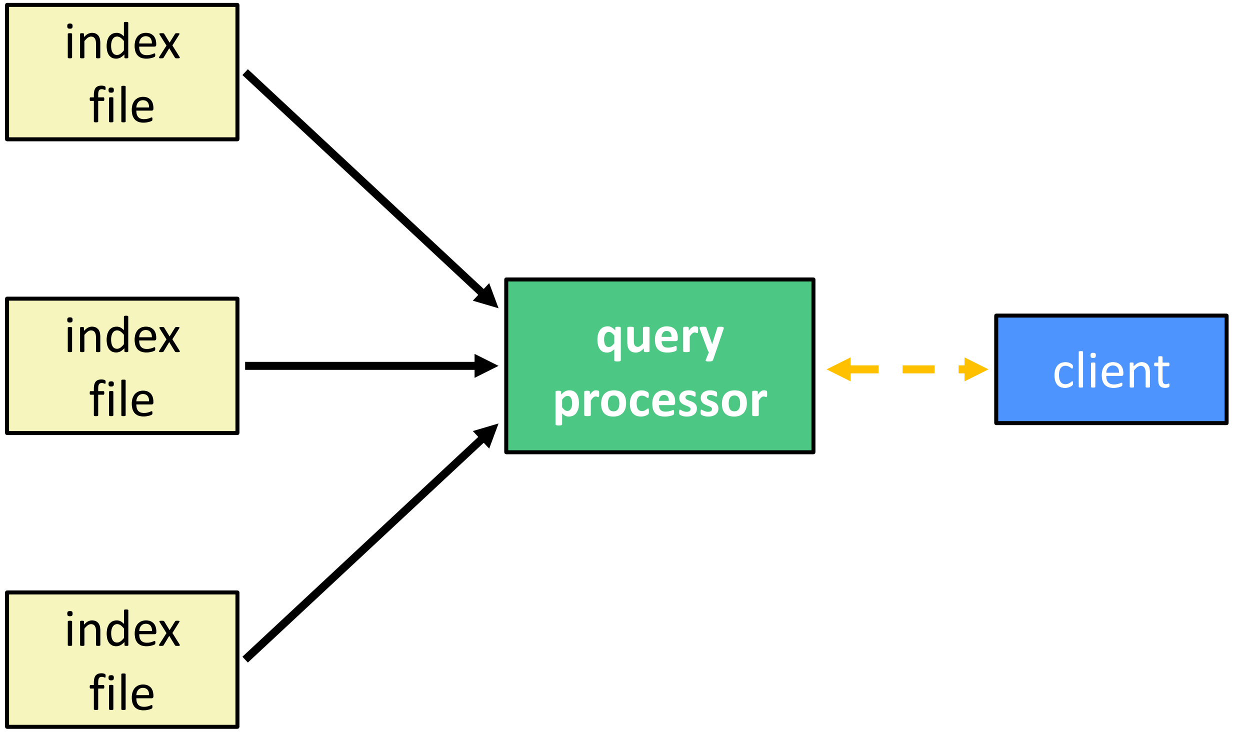
Lecture Outline

- Threads High Level
- pthreads
- Processes vs threads
- Thread Interleaving & Sequential Consistency
- **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

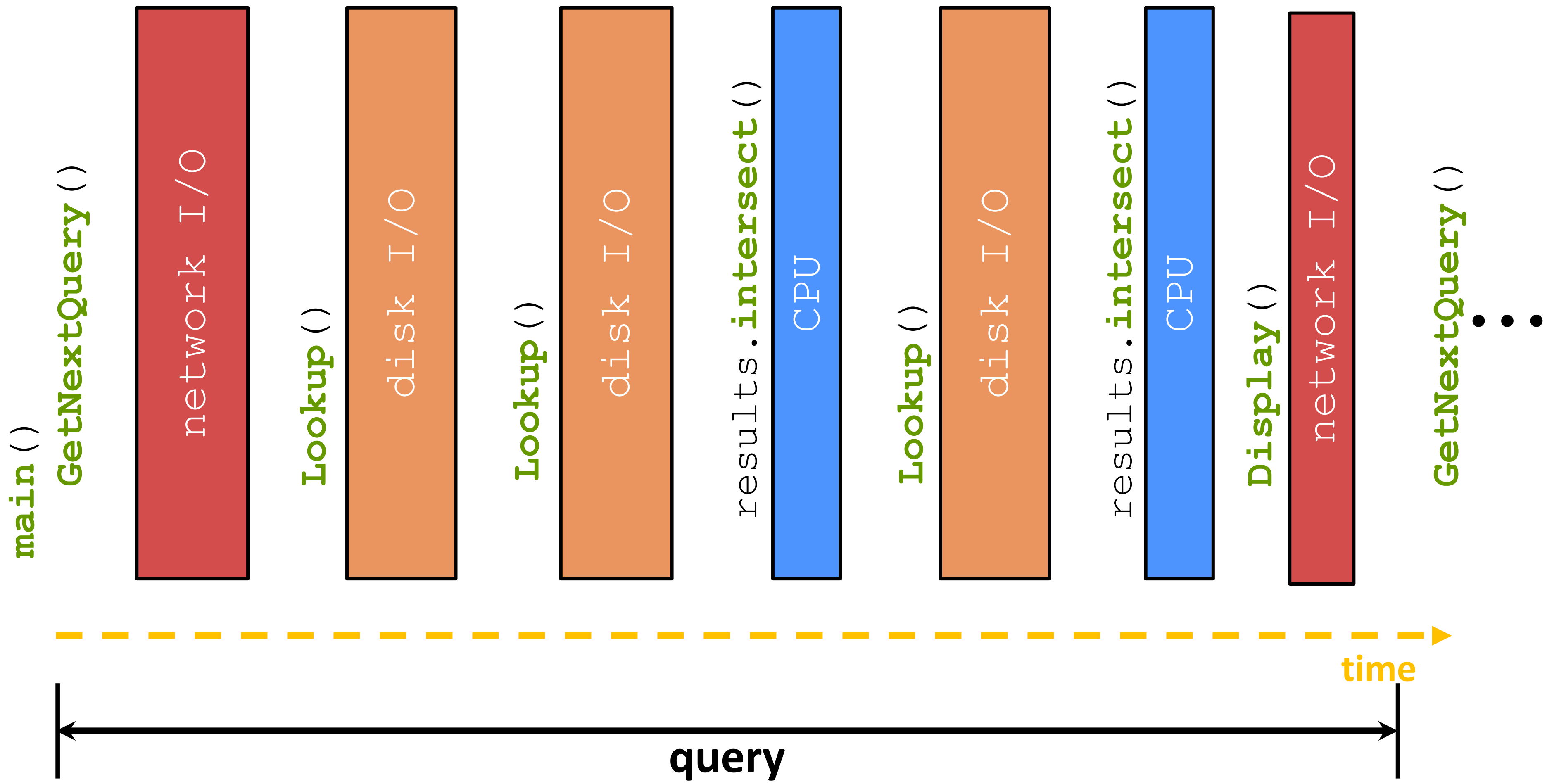
Search Engine Architecture



Search Engine (Pseudocode)

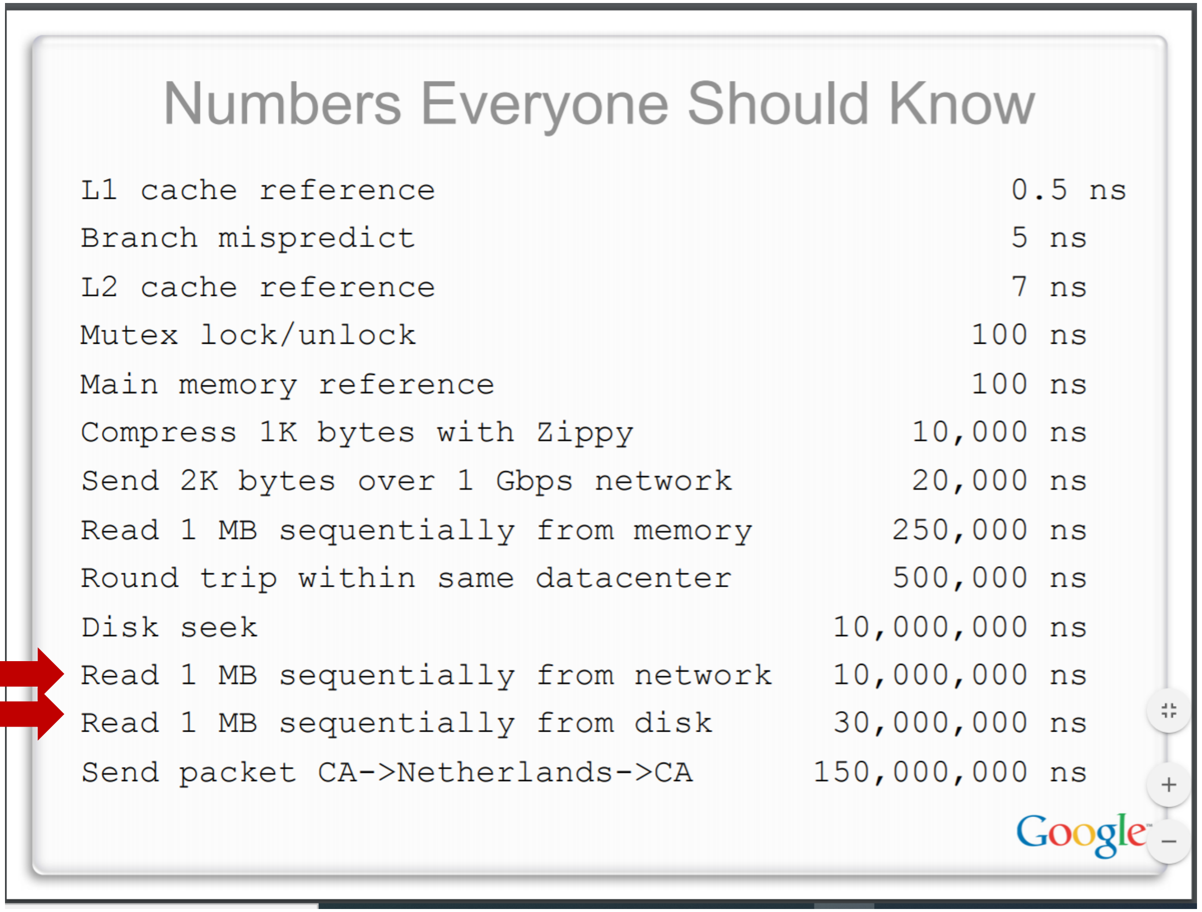
```
doclist Lookup(string word) {
  bucket = hash(word);
  hitlist = file.read(bucket);  ← Disk I/O
  foreach hit in hitlist {
    doclist.append(file.read(hit));
  }
  return doclist;
}

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

Execution Timeline: a Multi-Word Query



What About I/O-caused Latency?

- Jeff Dean’s “Numbers Everyone Should Know” (LADIS ‘09)

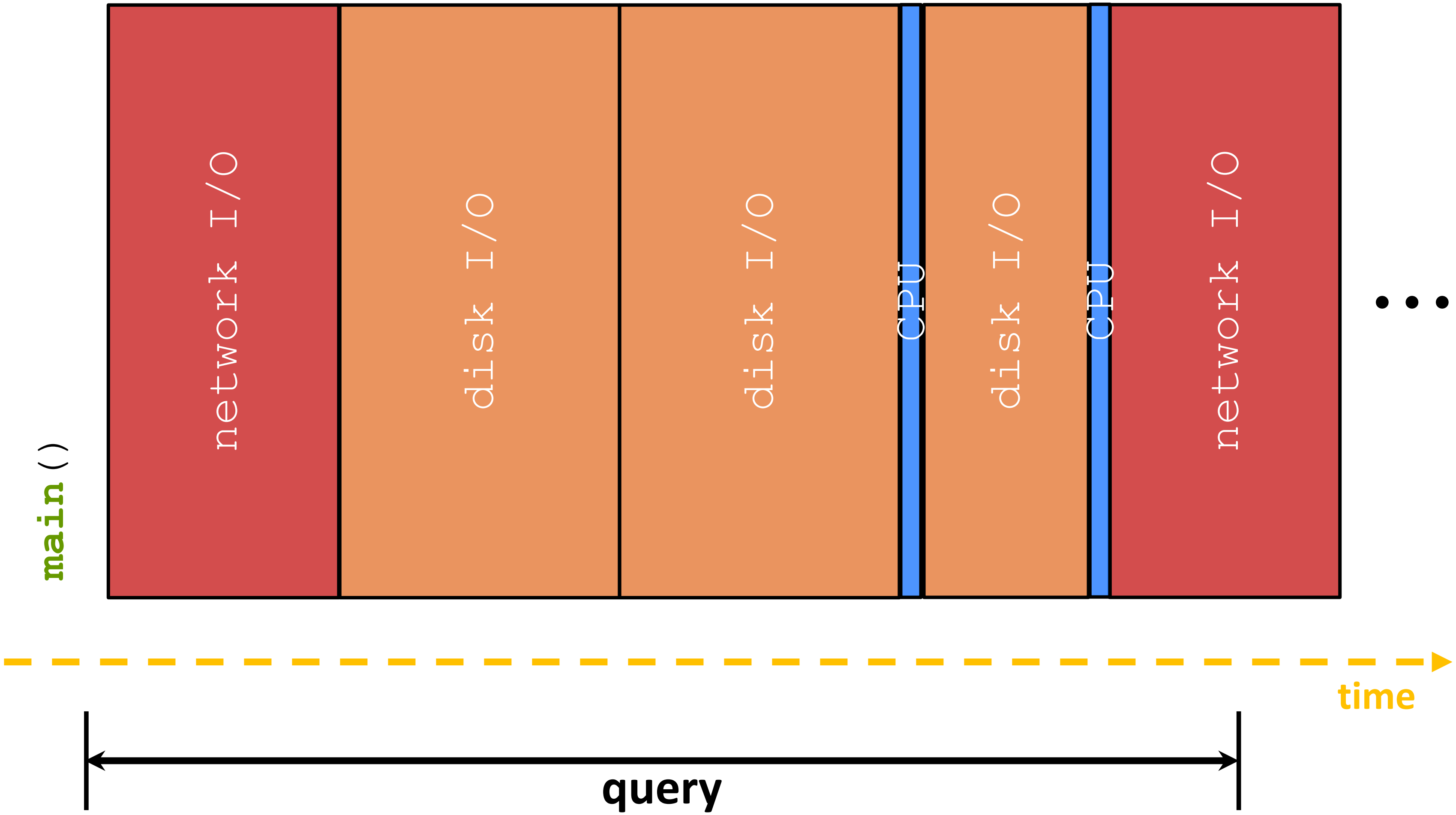


Numbers Everyone Should Know

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 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

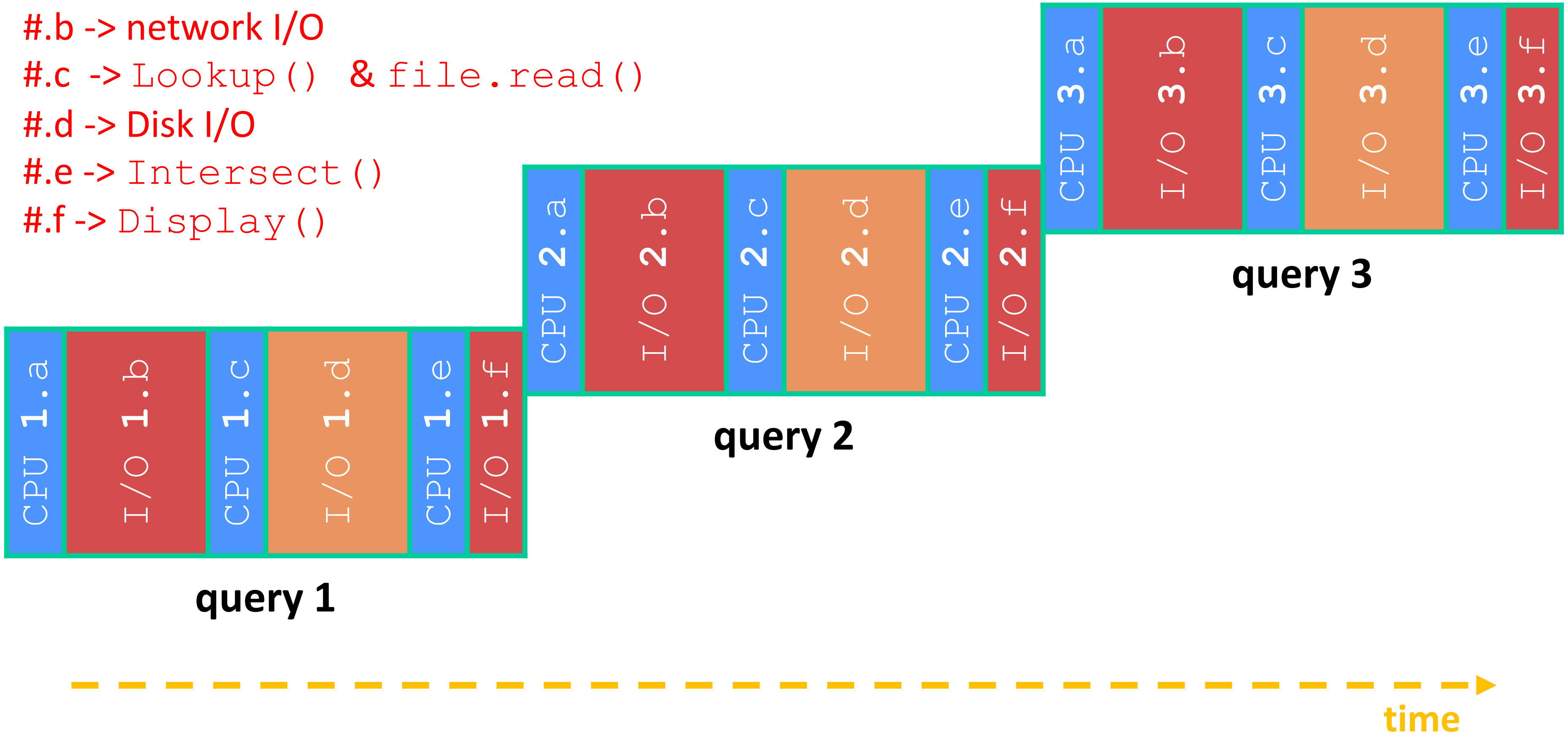
Execution Timeline: To Scale

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

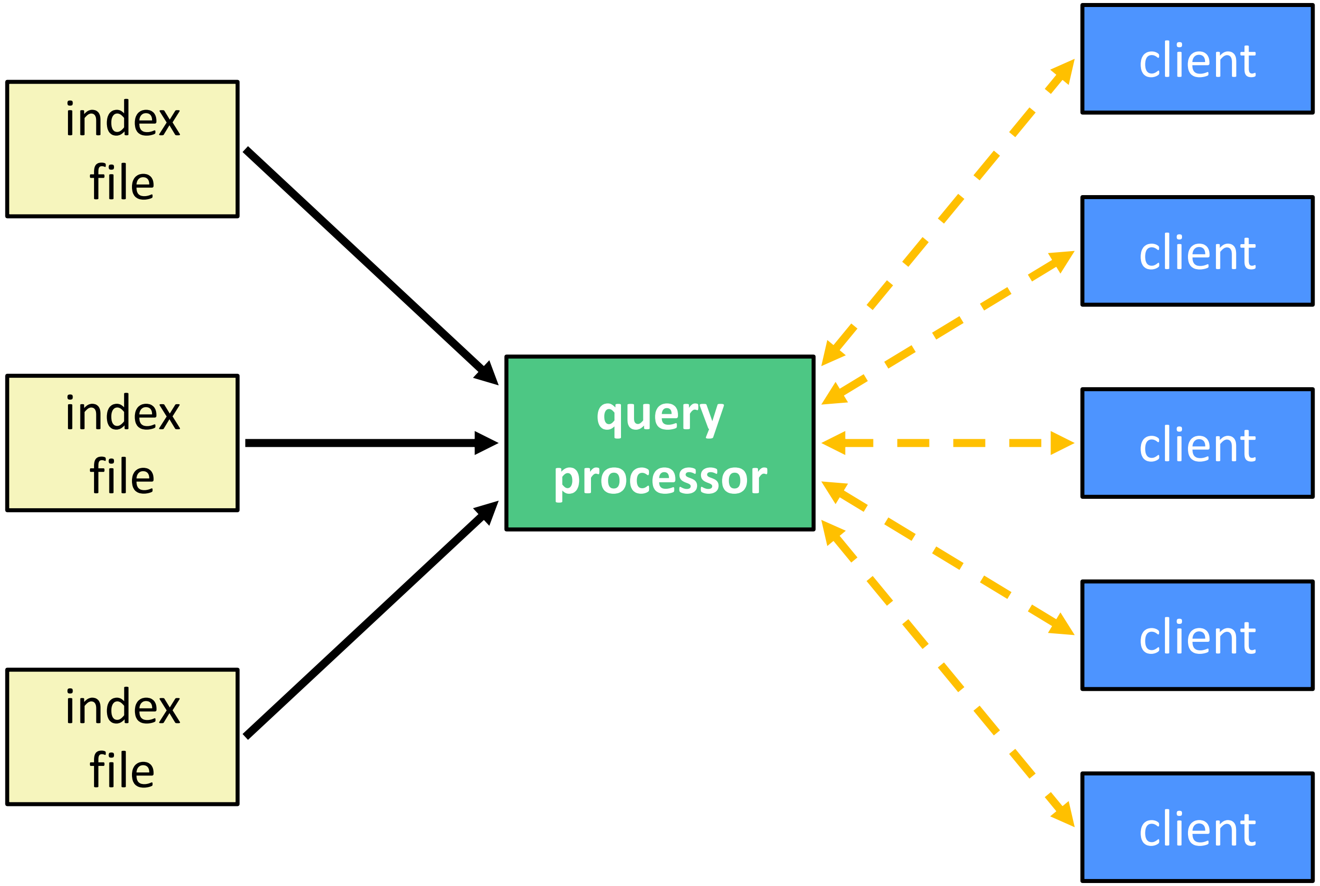


Multiple (Single-Word) Queries

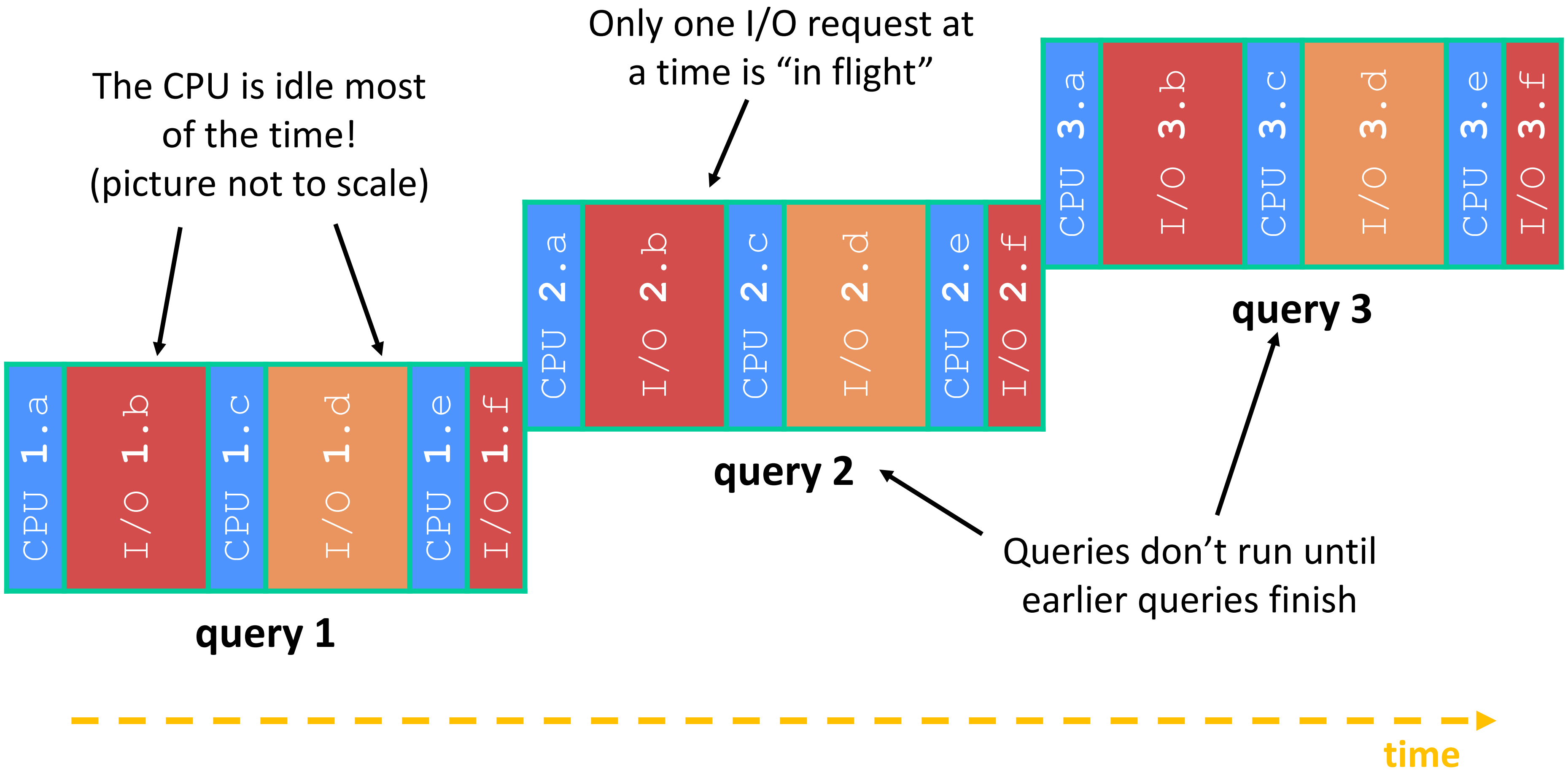
is the Query Number
#.a -> `GetNextQuery()`
#.b -> network I/O
#.c -> `Lookup()` & `file.read()`
#.d -> Disk I/O
#.e -> `Intersect()`
#.f -> `Display()`



Uh-Oh (1 of 2)


index file
index file
index file
query processor
client
client
client
client
client

Uh-Oh (2 of 2)



The CPU is idle most of the time!
(picture not to scale)

Only one I/O request at a time is "in flight"

Queries don't run until earlier queries finish

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 …)

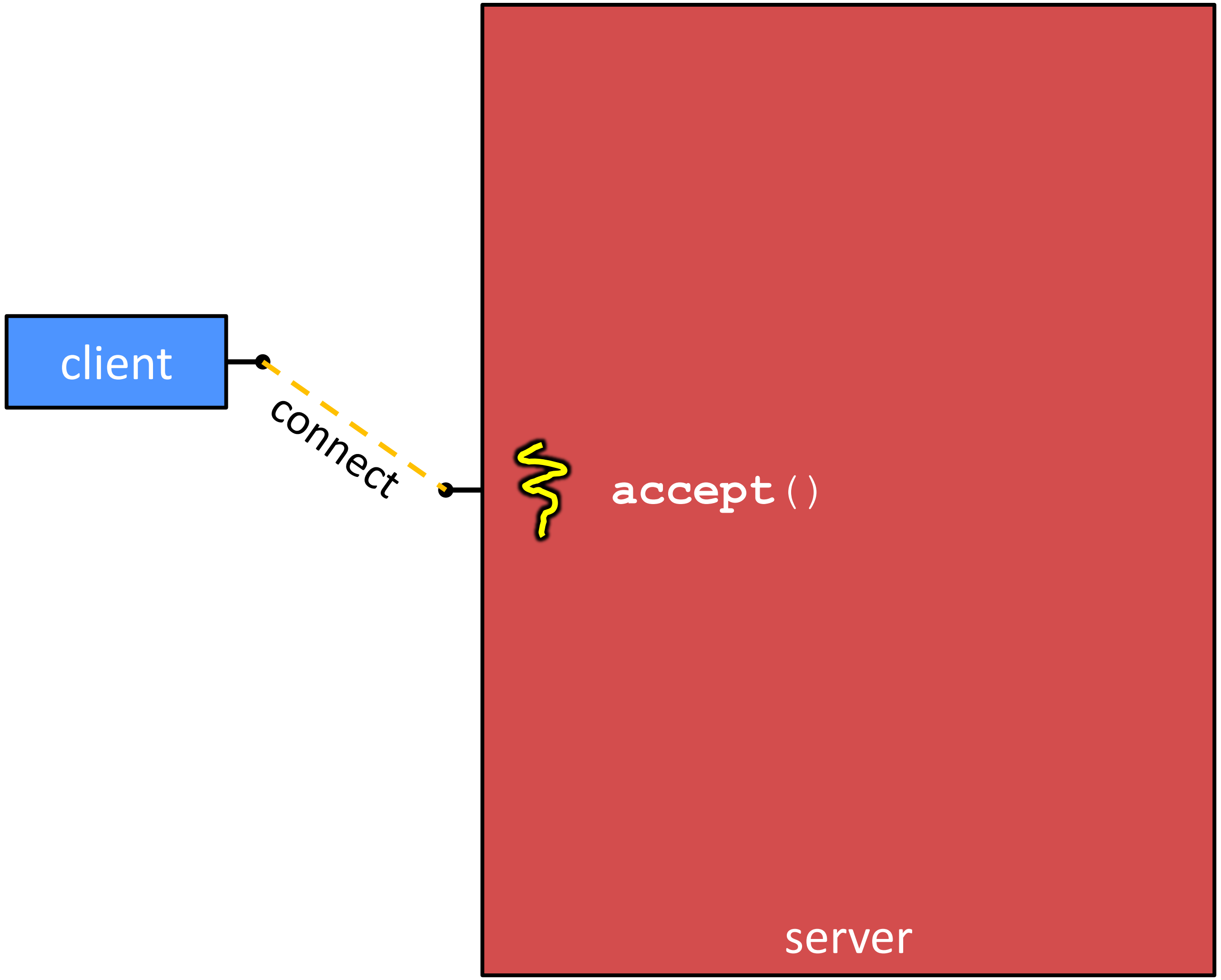
- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.

A Concurrent Implementation

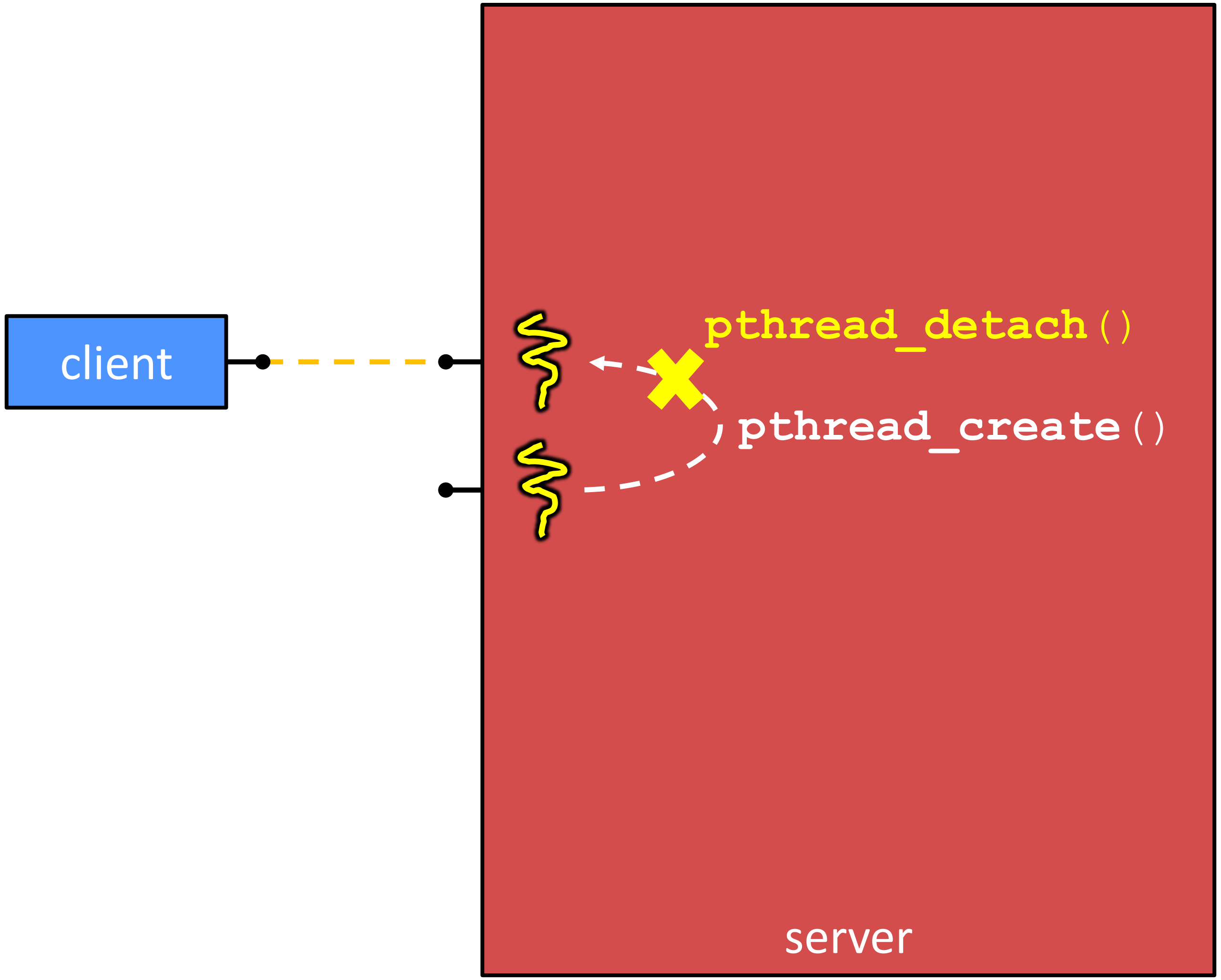
- Use multiple "workers"
  - As a query arrives, create a new "worker" to handle it
    - The "worker" reads the query from the network, issues read requests against files, assembles results and writes to the network
    - The "worker" uses blocking I/O; the "worker" alternates between consuming CPU 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
- So what should we use for our "workers"?

Threads!!!!

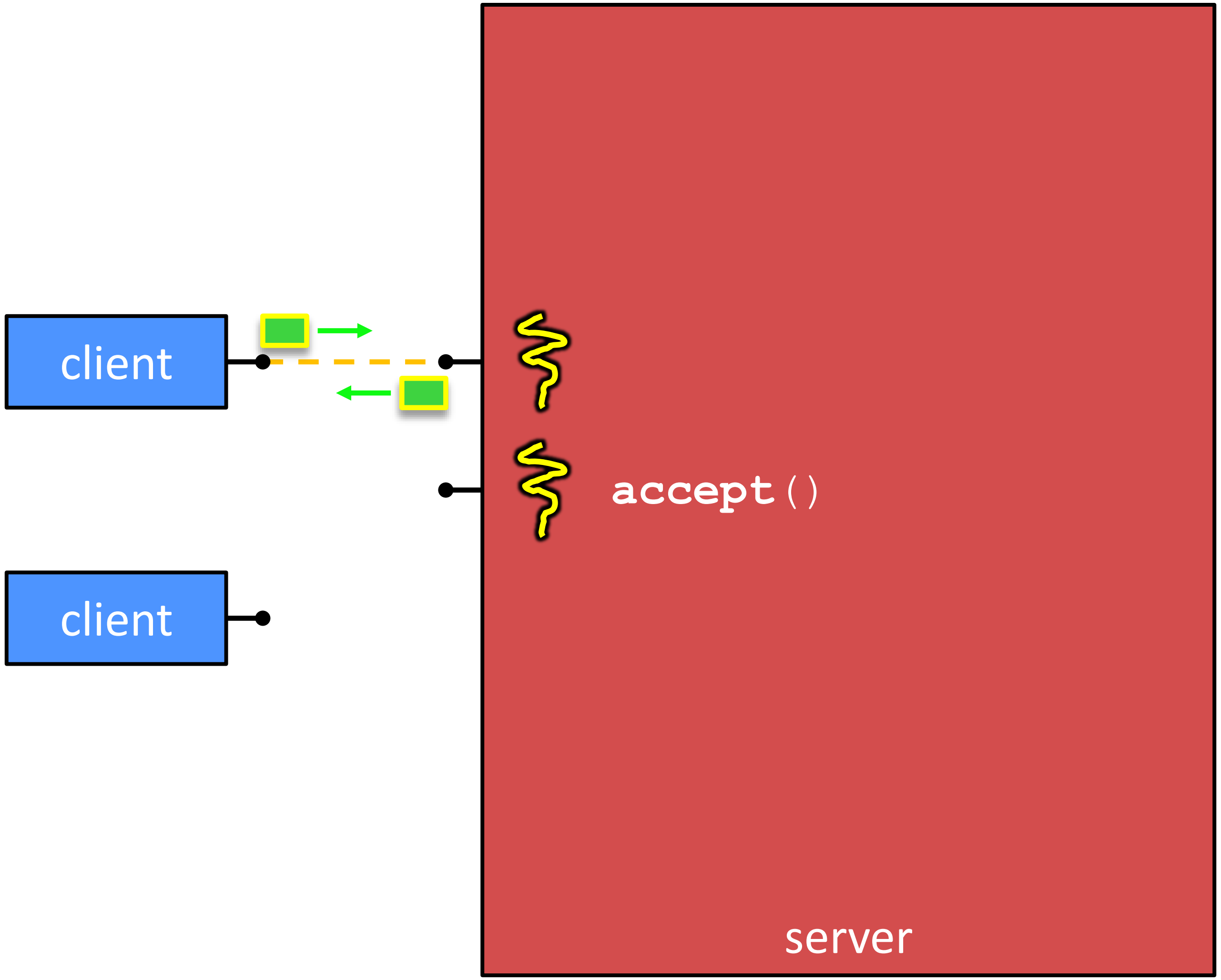
Multithreaded Server



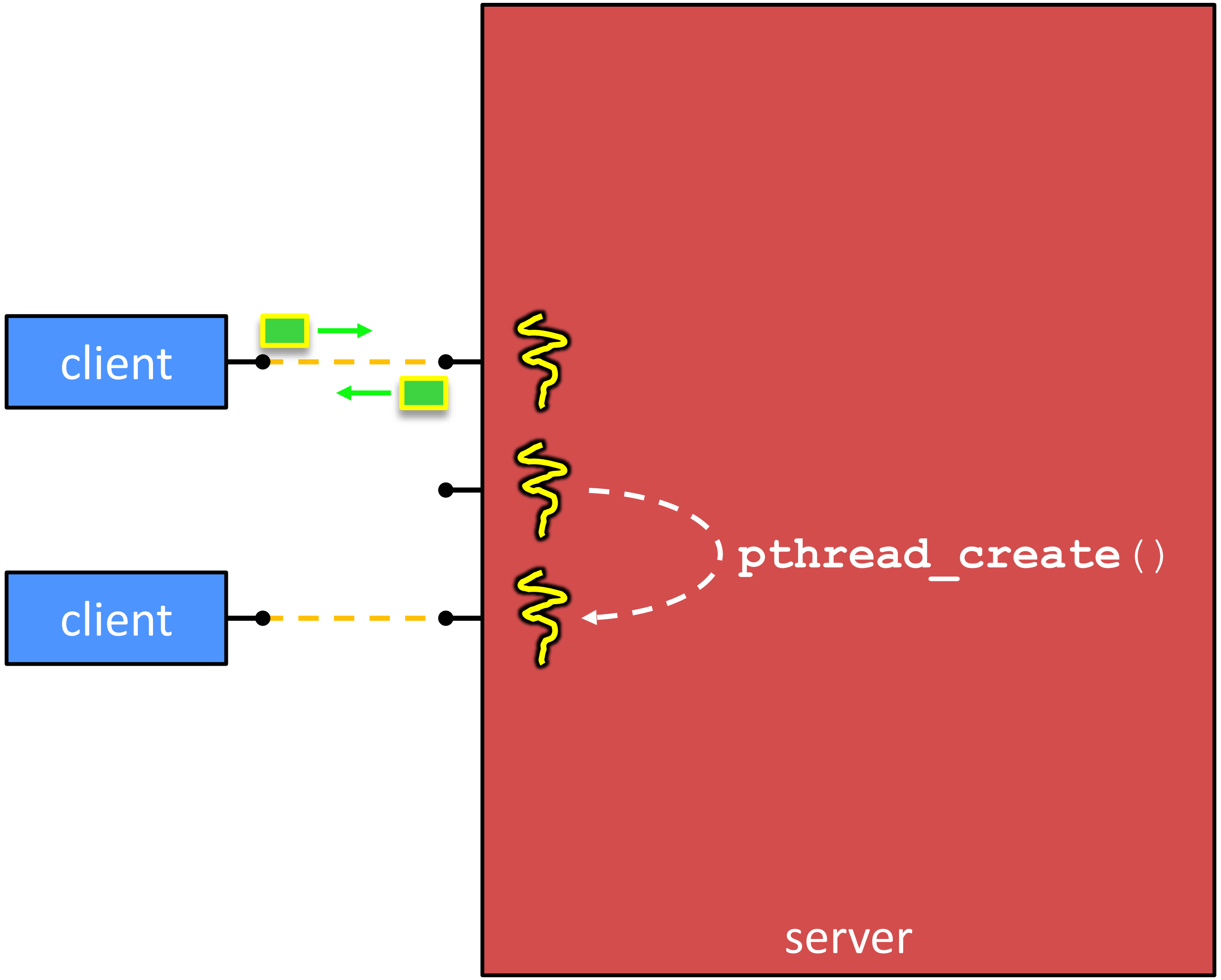
Multithreaded Server



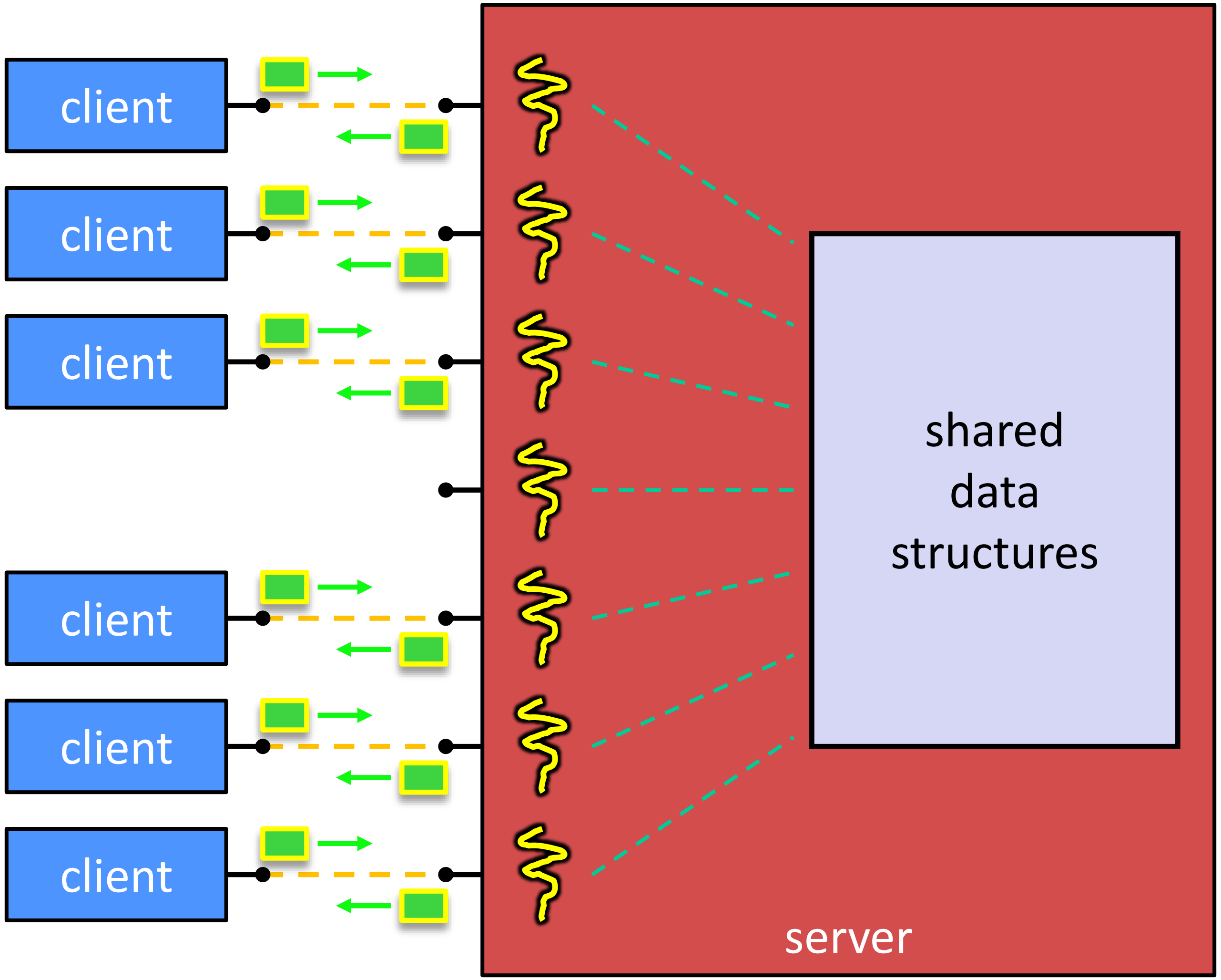
Multithreaded Server



Multithreaded Server

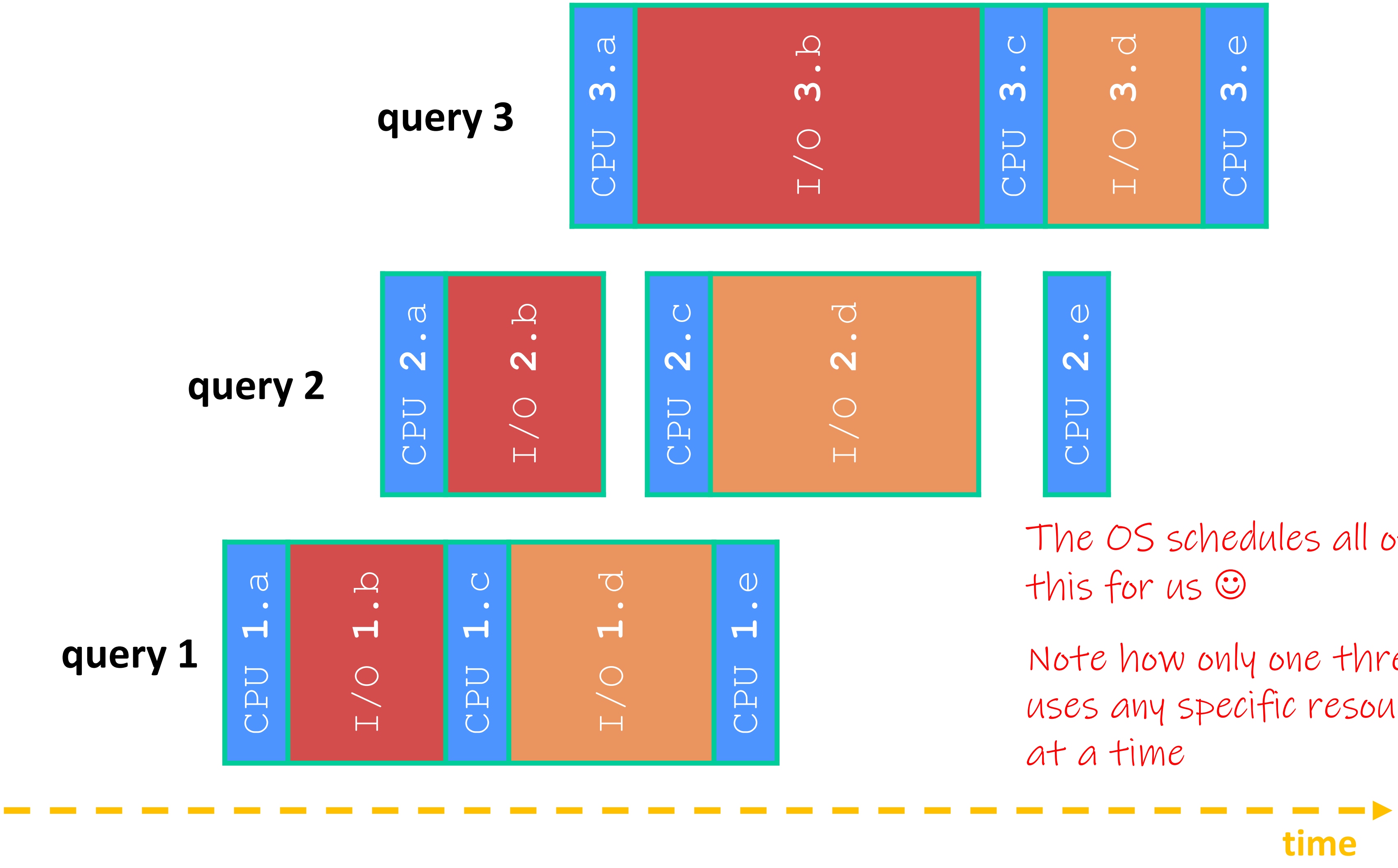


Multithreaded Server



Multi-threaded Search Engine (Execution)

*Running with 1 CPU



The OS schedules all of this for us ☺

Note how only one thread uses any specific resource at a time

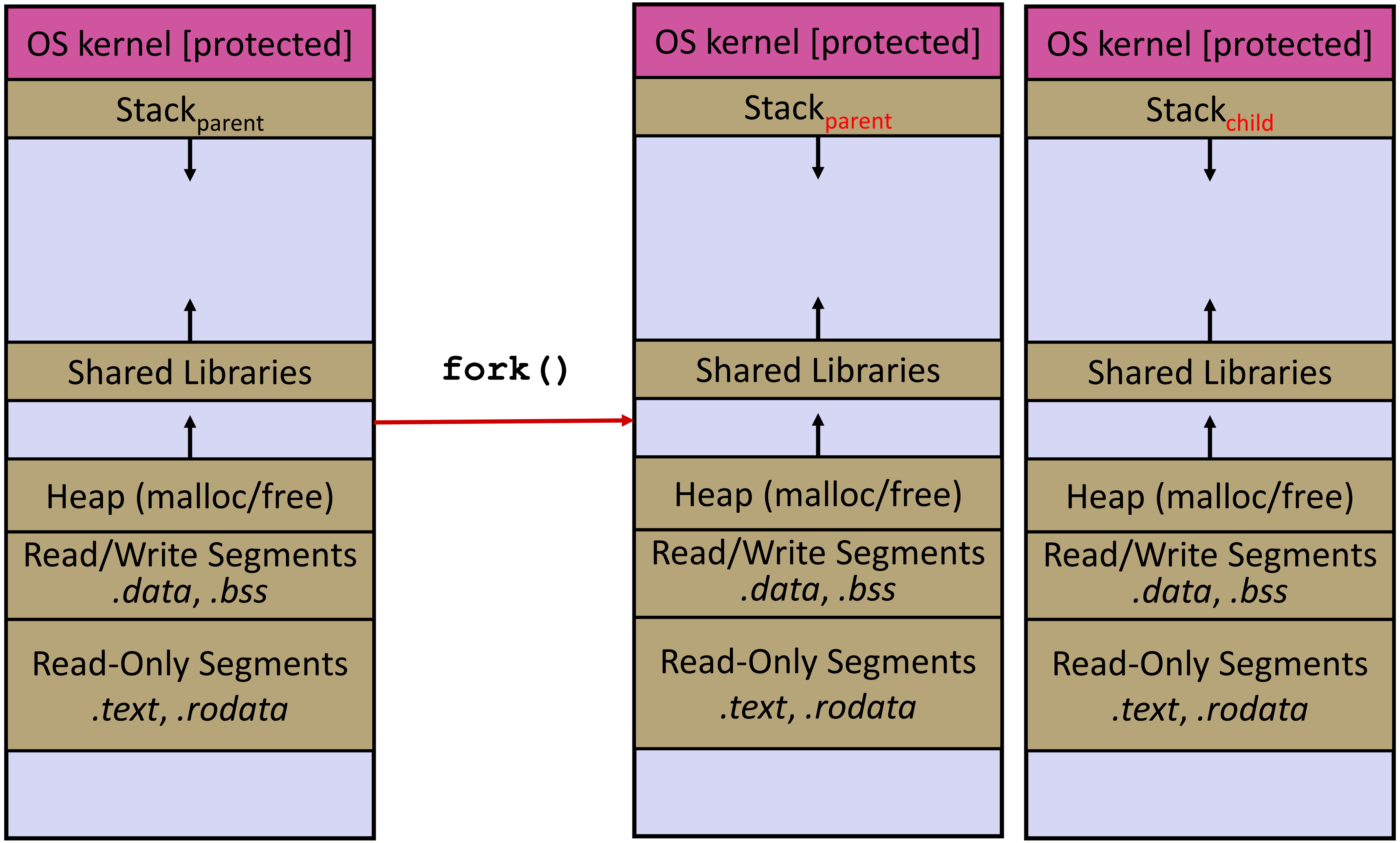
Why Threads?

- 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

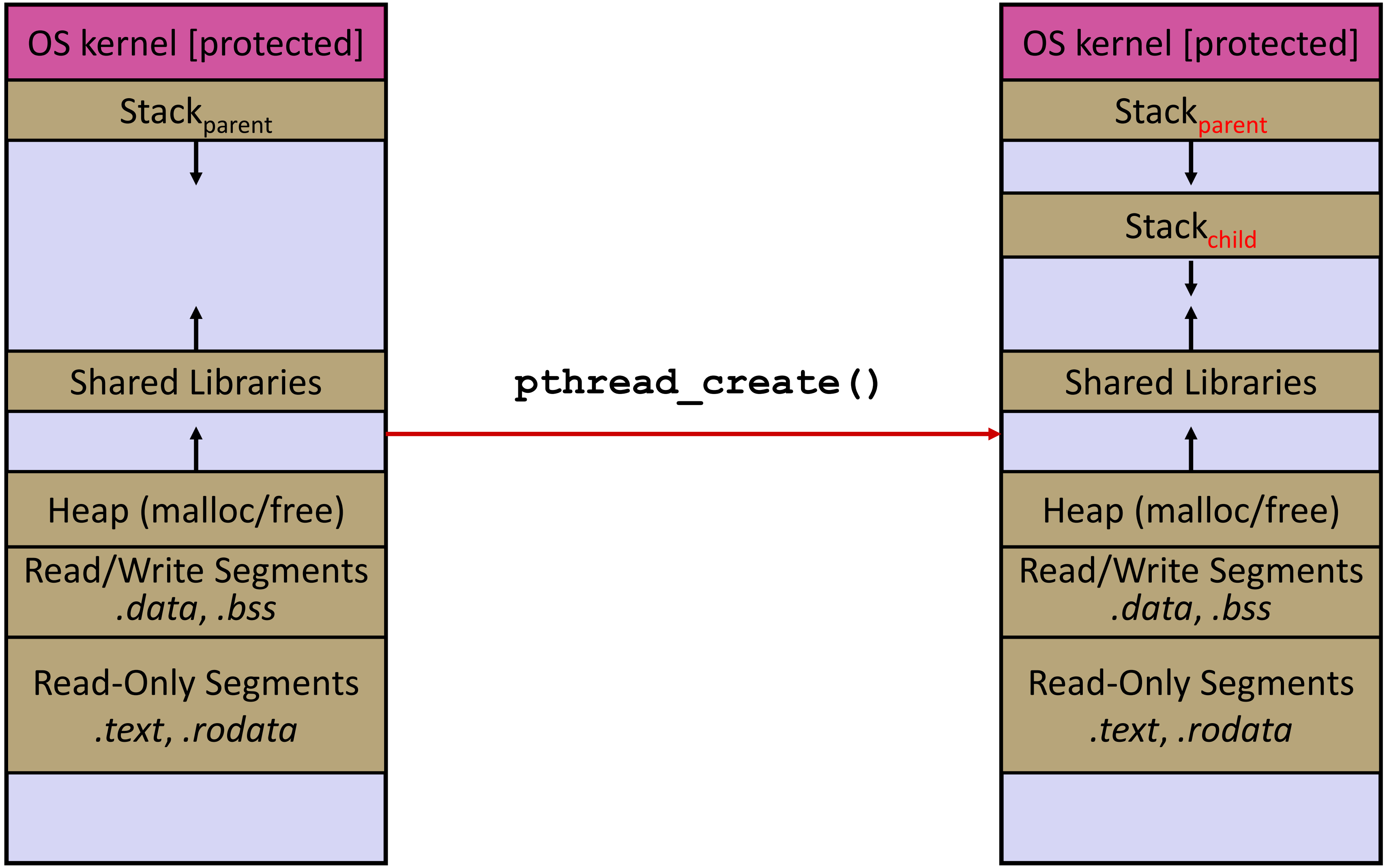
Threads vs. Processes

- 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!