Threads & Scheduling

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Project 1 is out now
  - Project is due 11:59 pm on Wed, Oct 11 **(TOMORROW)**
    late deadline 11:59 pm on Sun, Oct 15
- For project 1 full submission, please do a group submission on gradescope (one of you submits but you add your partner to the submission)
- Recitation Today after lecture:
  - Some cool stuff ☺ and then Open Office Hours Afterwards
- Travis has Office hours 4:30 to 6:30
  - And will host more office hours tomorrow night

Administrivia

- Midterm is coming soon (1 week + 2 days from now!)
  - Meyerson B1 7:00 pm to 9:00pm Thursday 10/19
  - If you can't make the time, please send me an email **ASAP**
- Midterm Policies posted on the course website. Please read through them.
  - You are allowed 1 page of notes 8.5 x 11 double sided notes
  - Clobber policy: can show growth by doing better on the second midterm
- Recitation next week and lectures next week will contain midterm review
  - Tuesday lecture will warp up scheduling, not only review

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

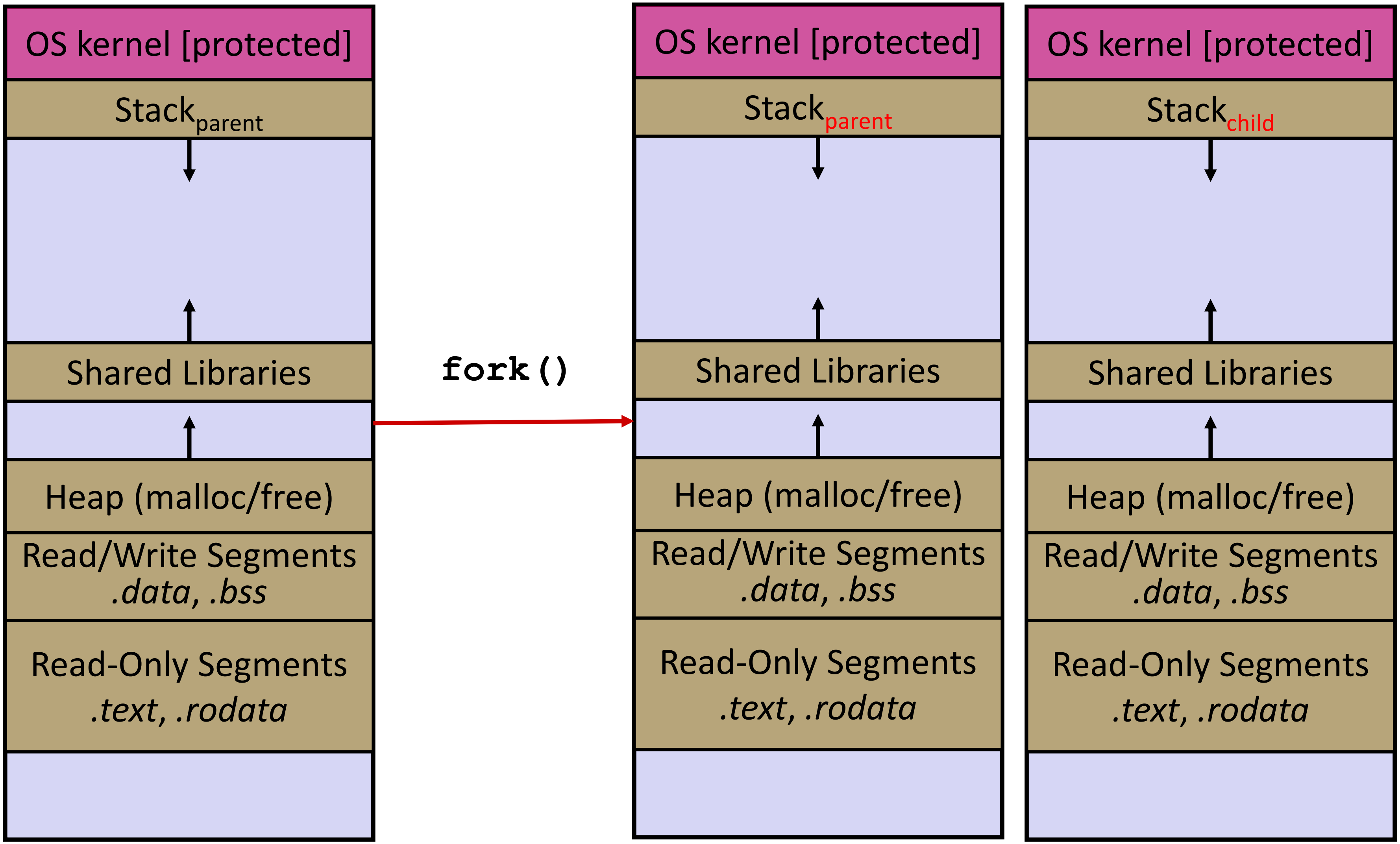
Lecture Outline

- **Threads vs processes**
- Threads & Blocking
- User Level Threads vs Kernel Level Threads
  - `ucontext`
- Scheduling

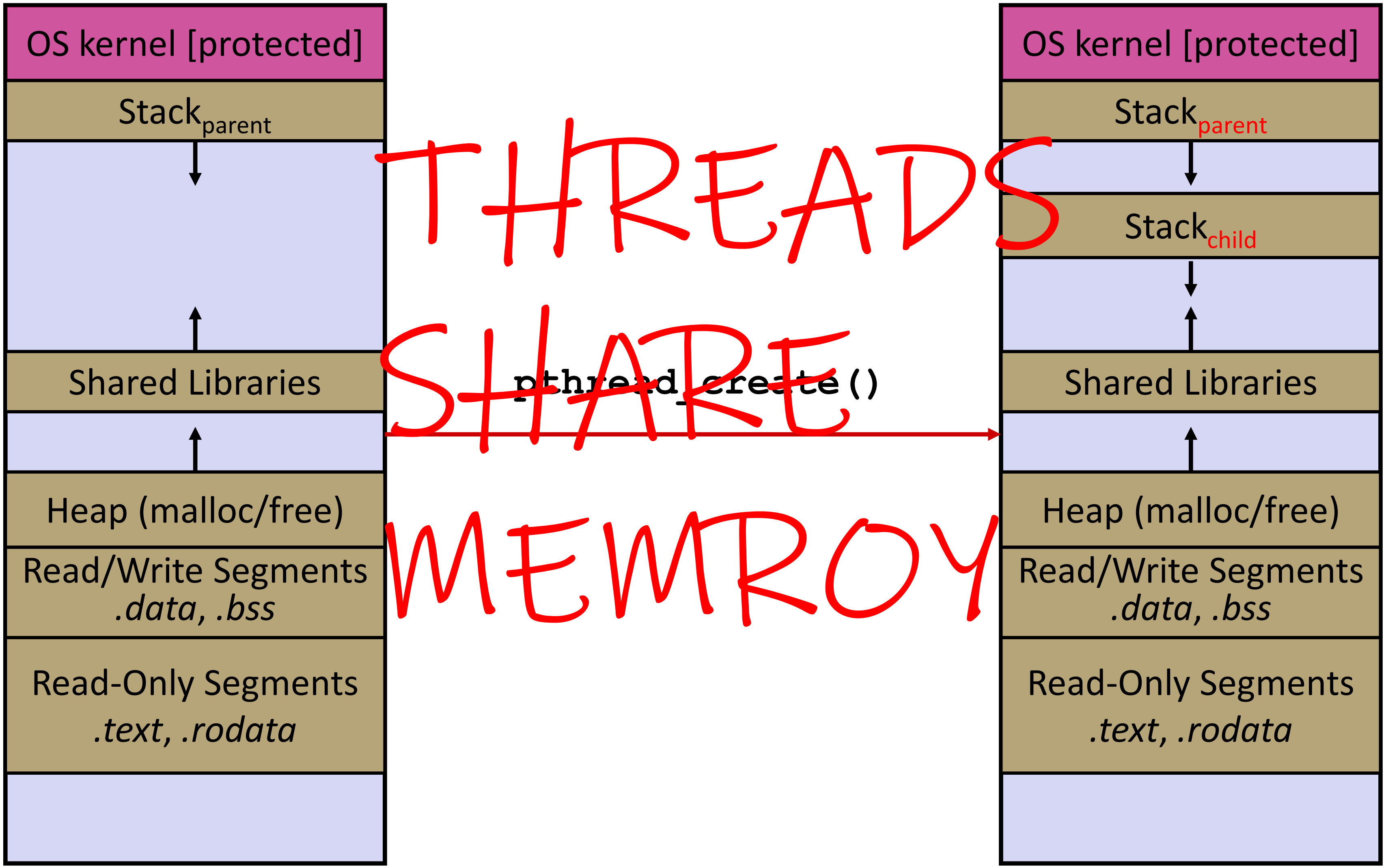
Threads vs. Processes

- 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



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()

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

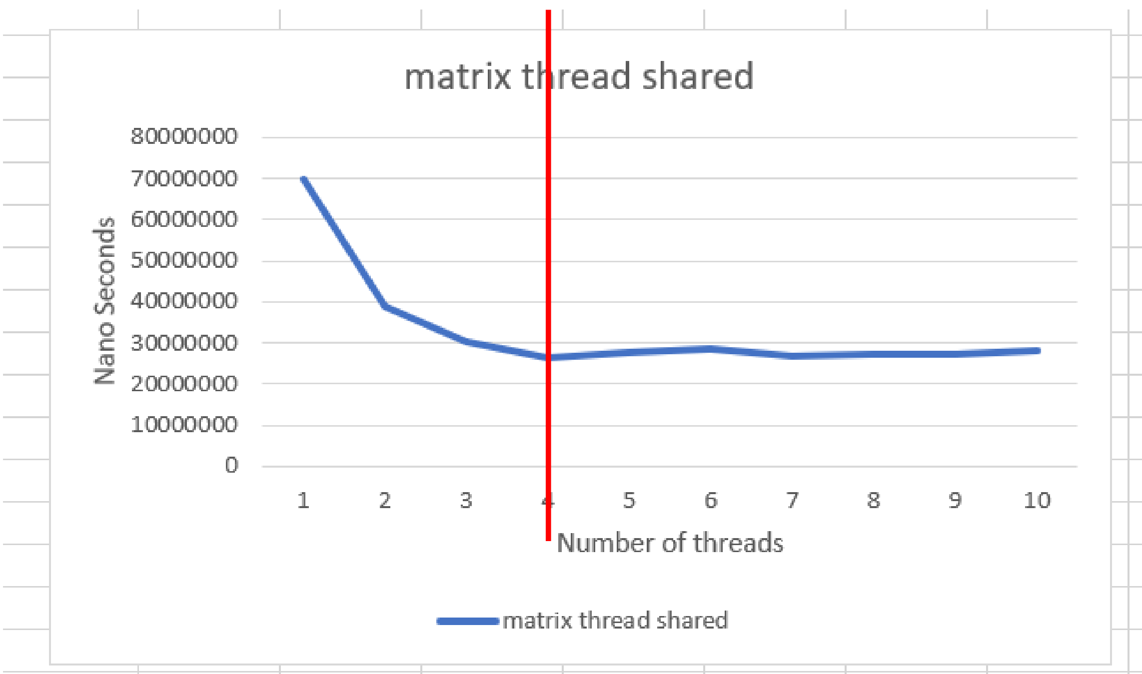
Parallelism

- 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

Other programs running, that may use the cores

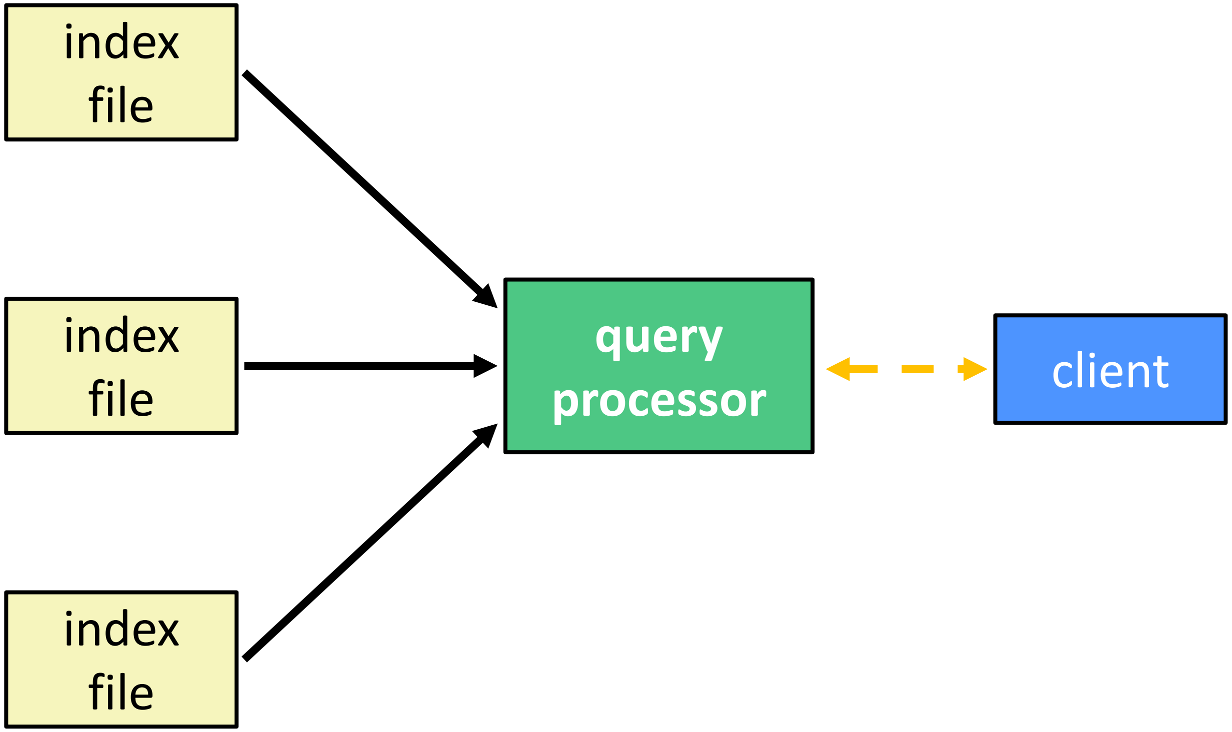
Lecture Outline

- Threads vs processes
- **Threads & Blocking**
- User Level Threads vs Kernel Level Threads
  - `ucontext`
- Scheduling

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

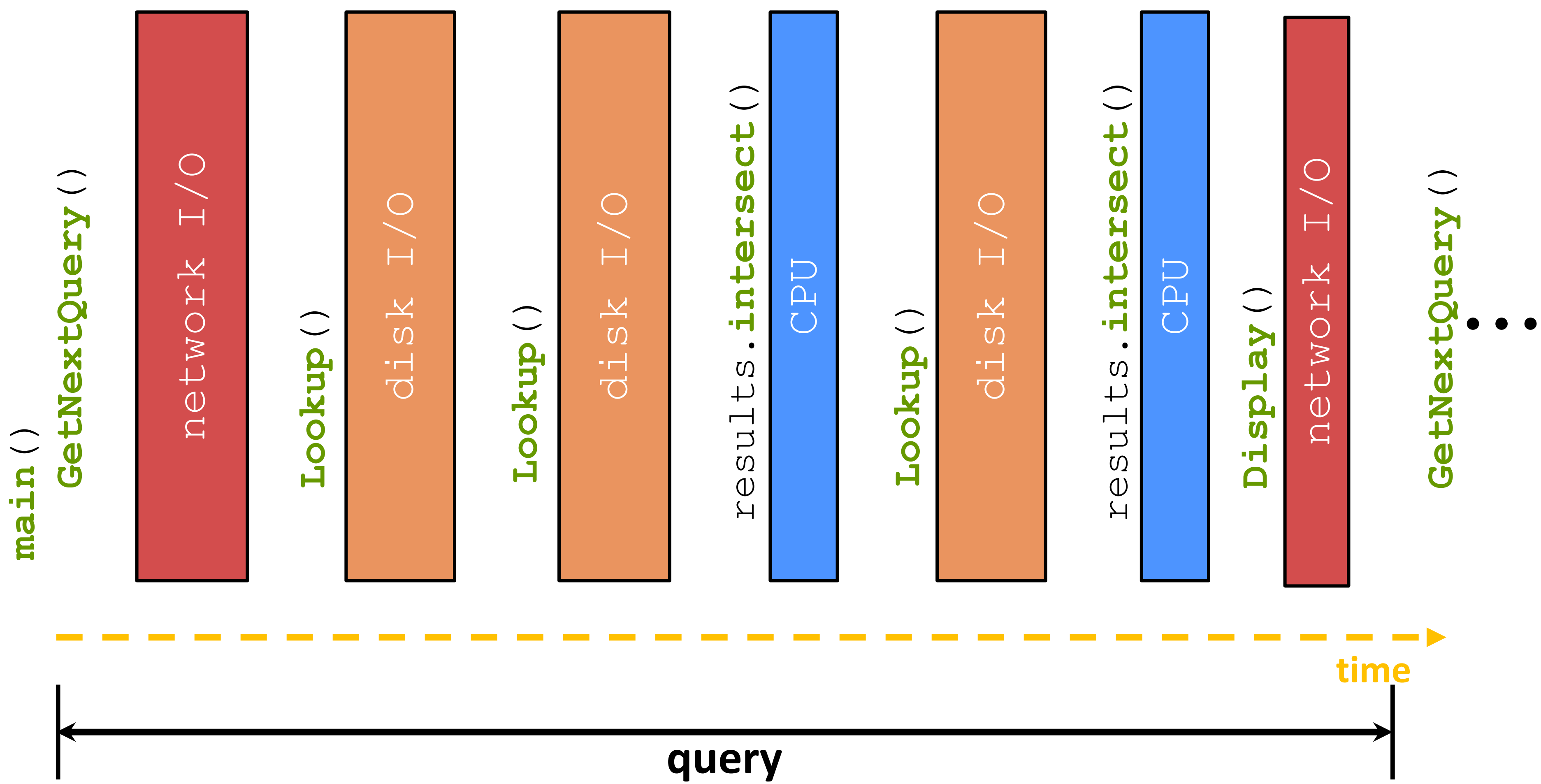


Search Engine (Pseudocode)

```
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)

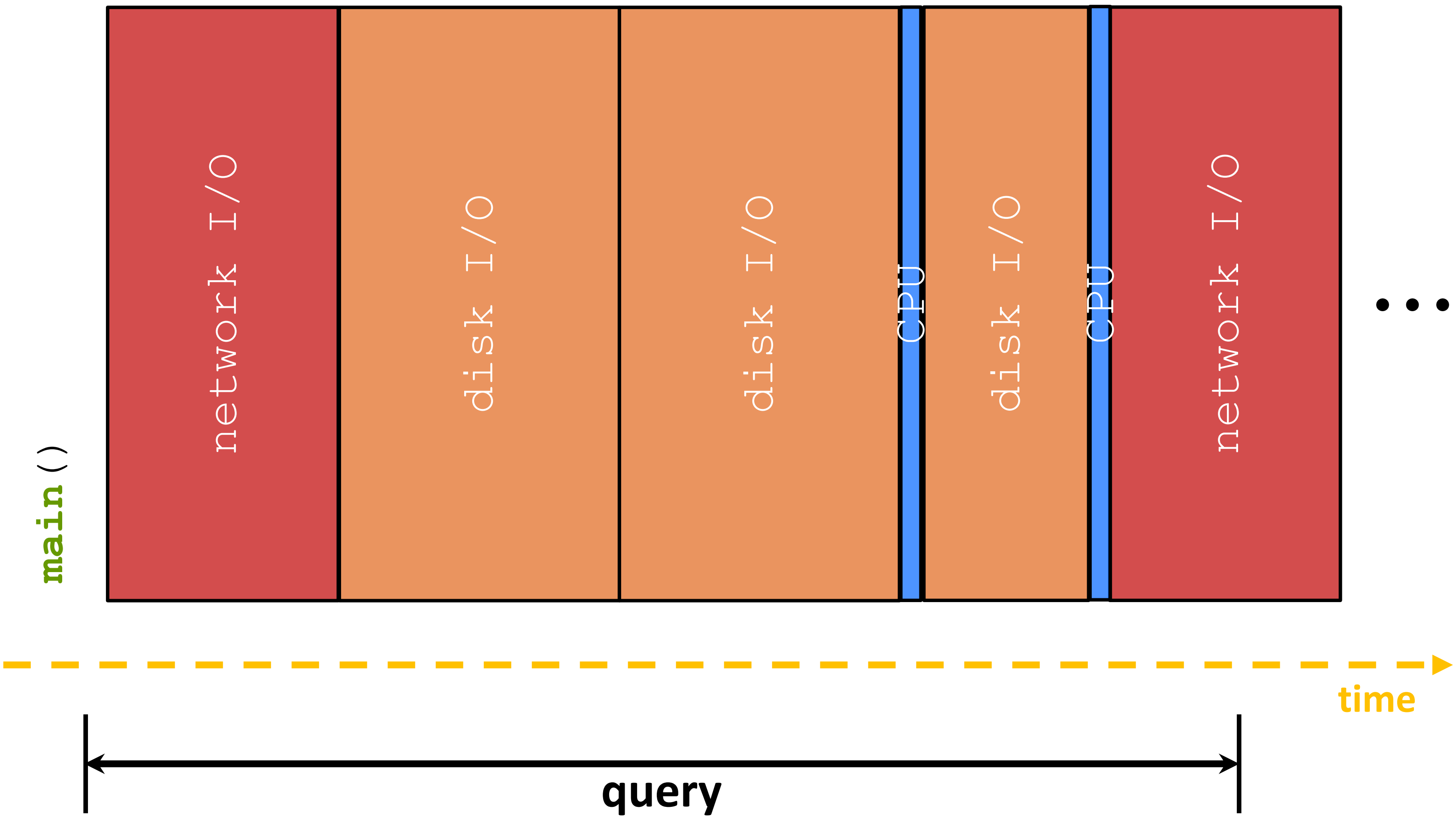
Numbers Everyone Should Know

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

Google

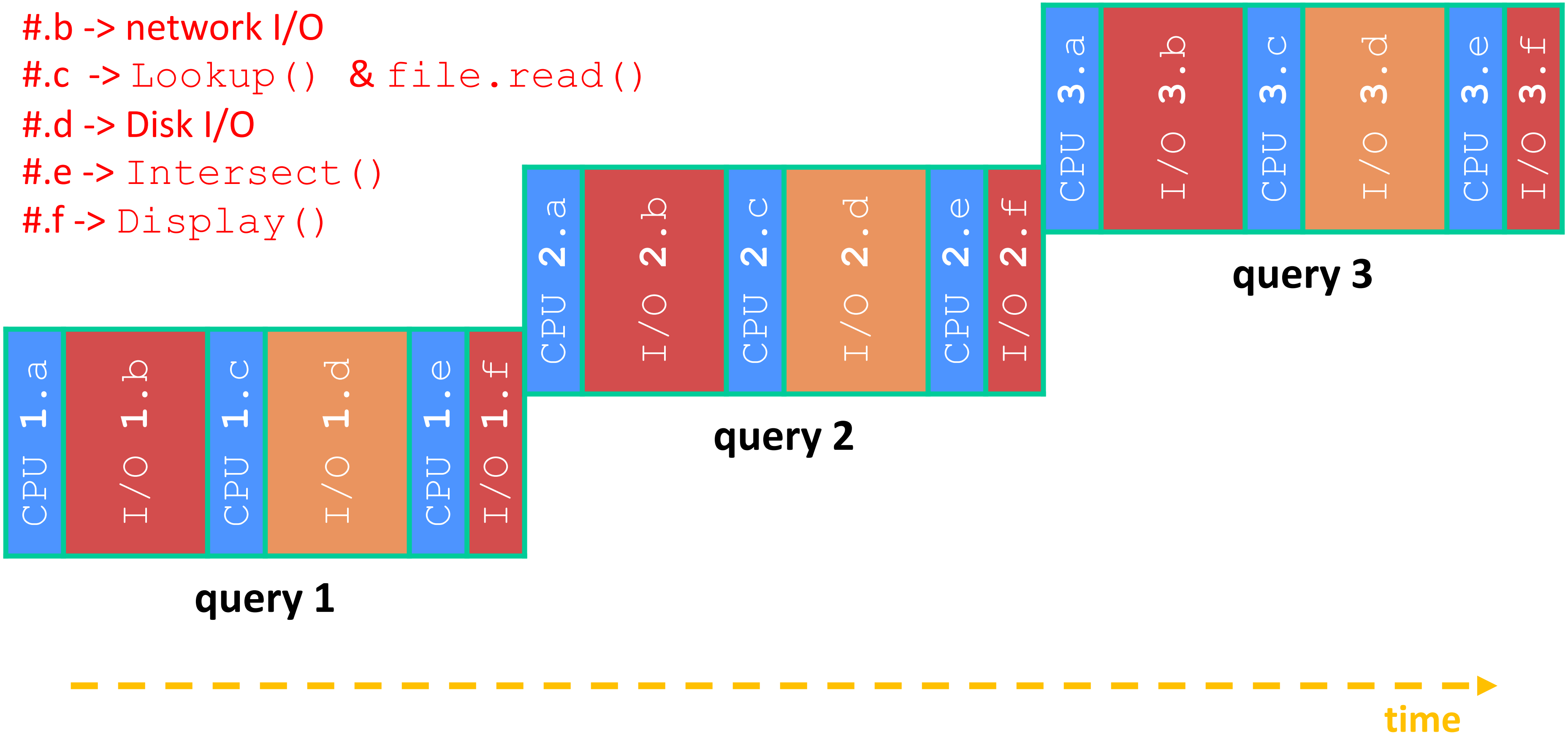
Execution Timeline: To Scale

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

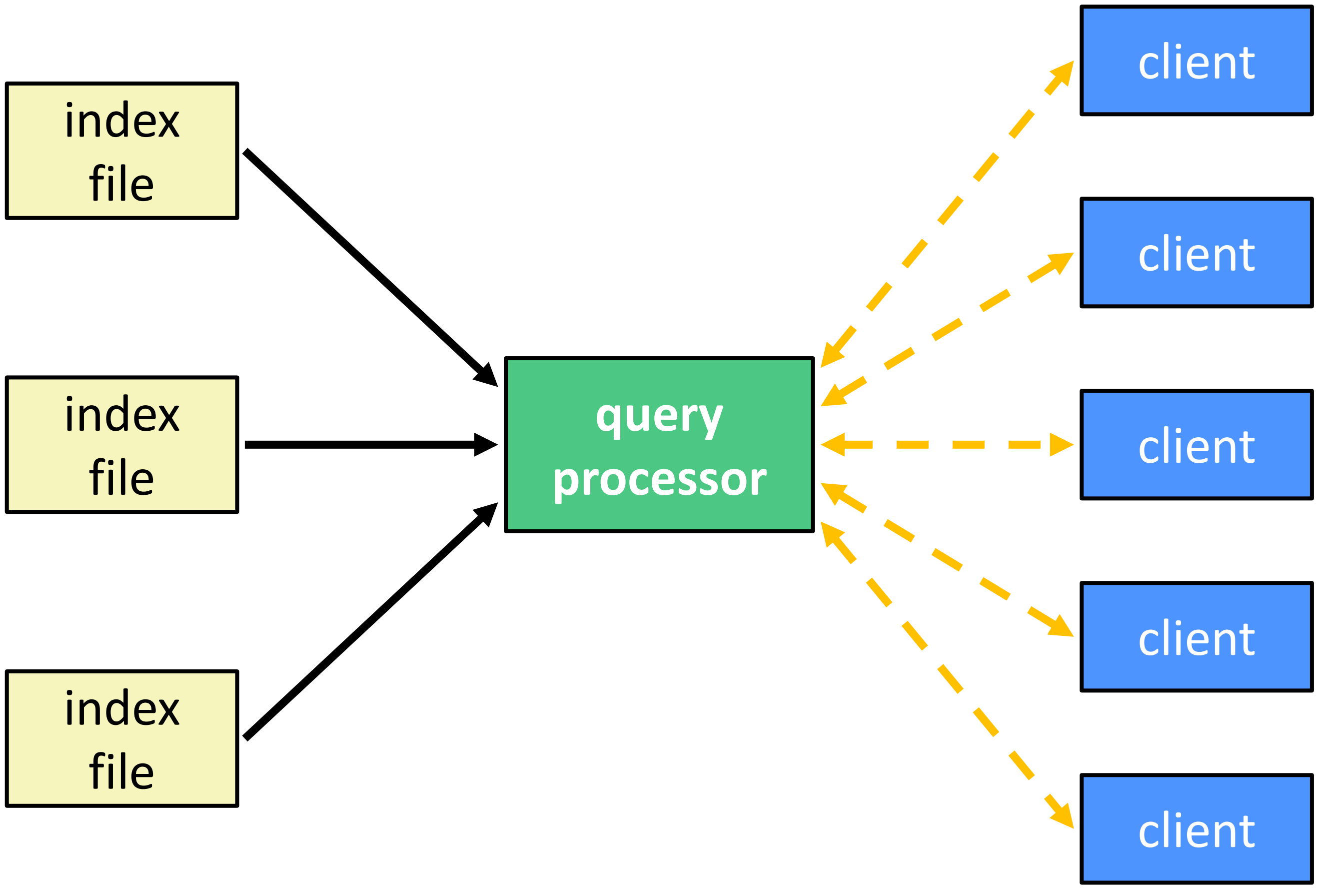


Multiple (Single-Word) Queries

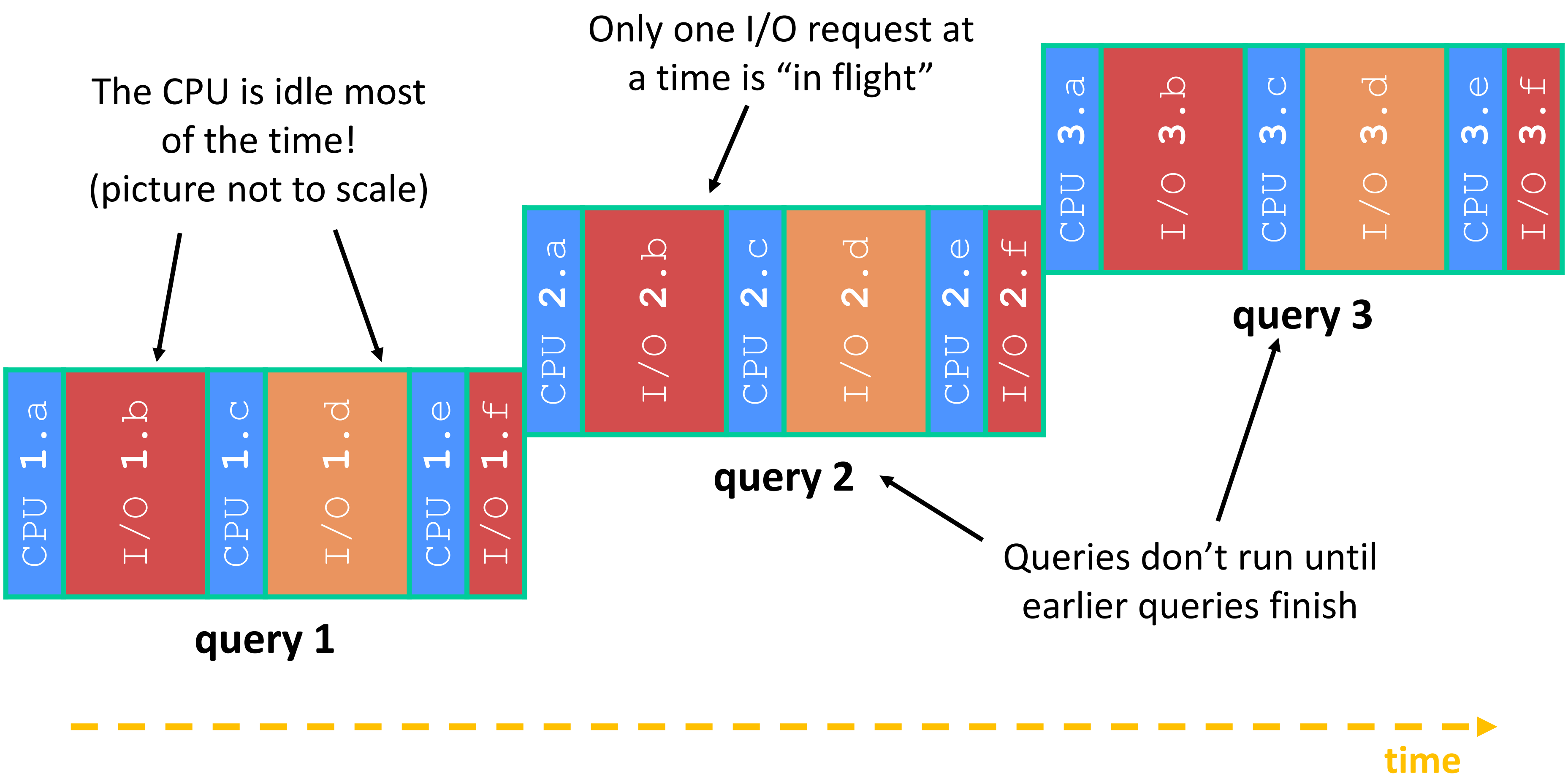
is the Query Number
#.a -> `GetNextQuery()`
#.b -> network I/O
#.c -> `Lookup()` & `file.read()`
#.d -> Disk I/O
#.e -> `Intersect()`
#.f -> `Display()`



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"
CPU 1.a
I/O 1.b
CPU 1.c
I/O 1.d
CPU 1.e
I/O 1.f
query 1
CPU 2.a
I/O 2.b
CPU 2.c
I/O 2.d
CPU 2.e
I/O 2.f
query 2
CPU 3.a
I/O 3.b
CPU 3.c
I/O 3.d
CPU 3.e
I/O 3.f
query 3
Queries don't run until
earlier queries finish
time

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 …)

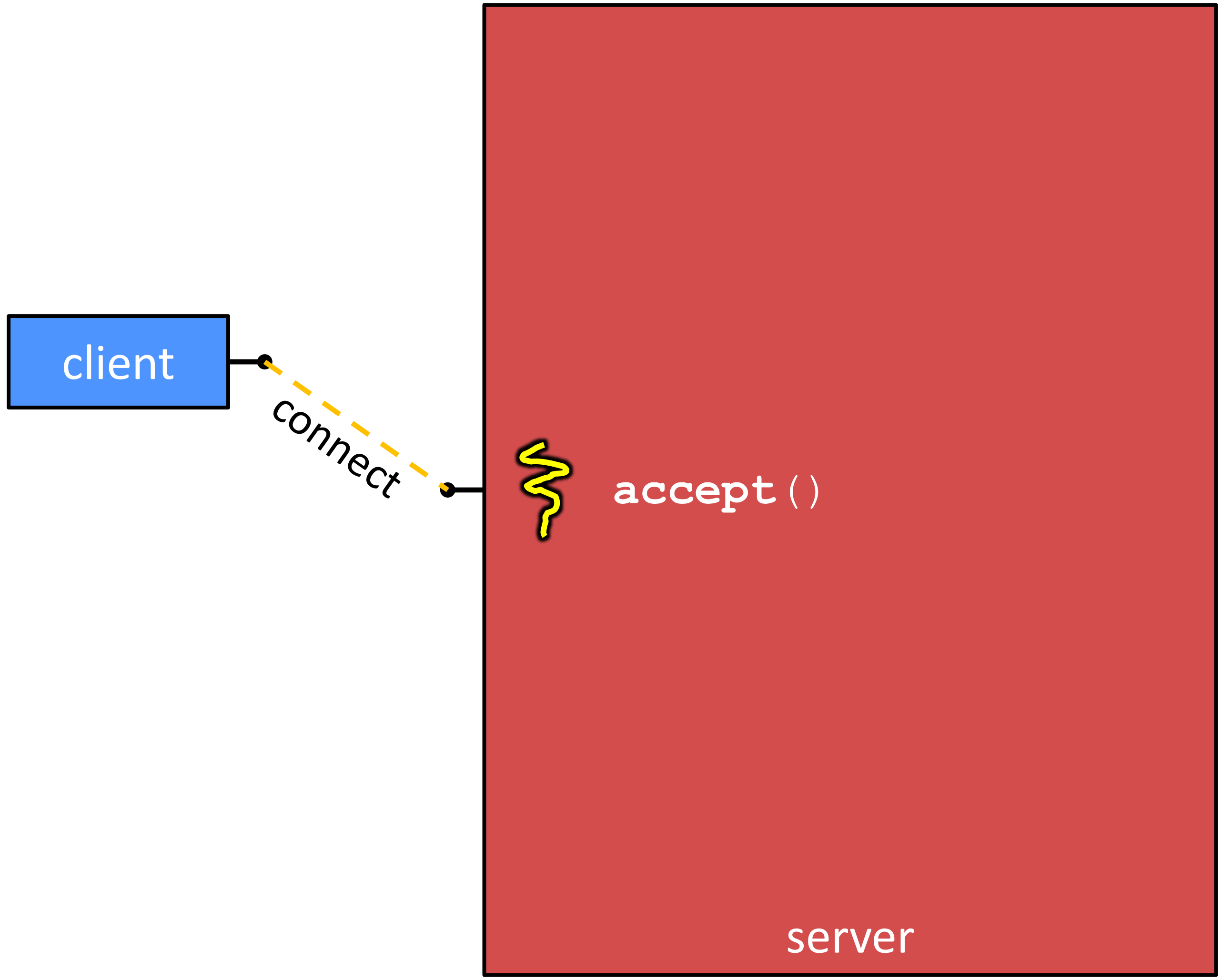
- At most one I/O operation is in flight at a time
  - Missed opportunities to speed I/O up
    - Separate devices in parallel, better scheduling of a single device, etc.

A Concurrent Implementation

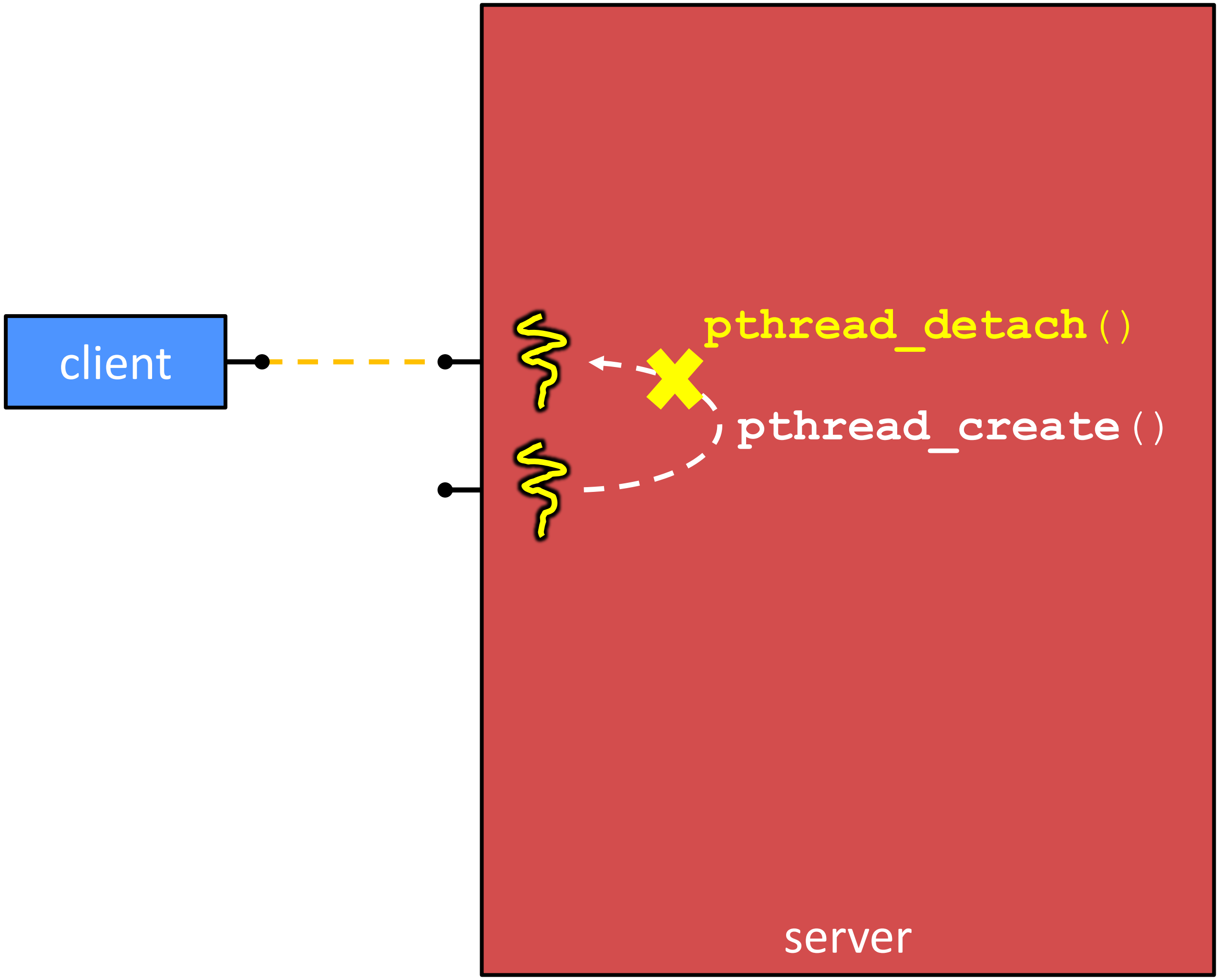
- Use multiple "workers"
  - As a query arrives, create a new "worker" to handle it
    - The "worker" reads the query from the network, issues read requests against files, assembles results and writes to the network
    - The "worker" uses blocking I/O; the "worker" alternates between consuming CPU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

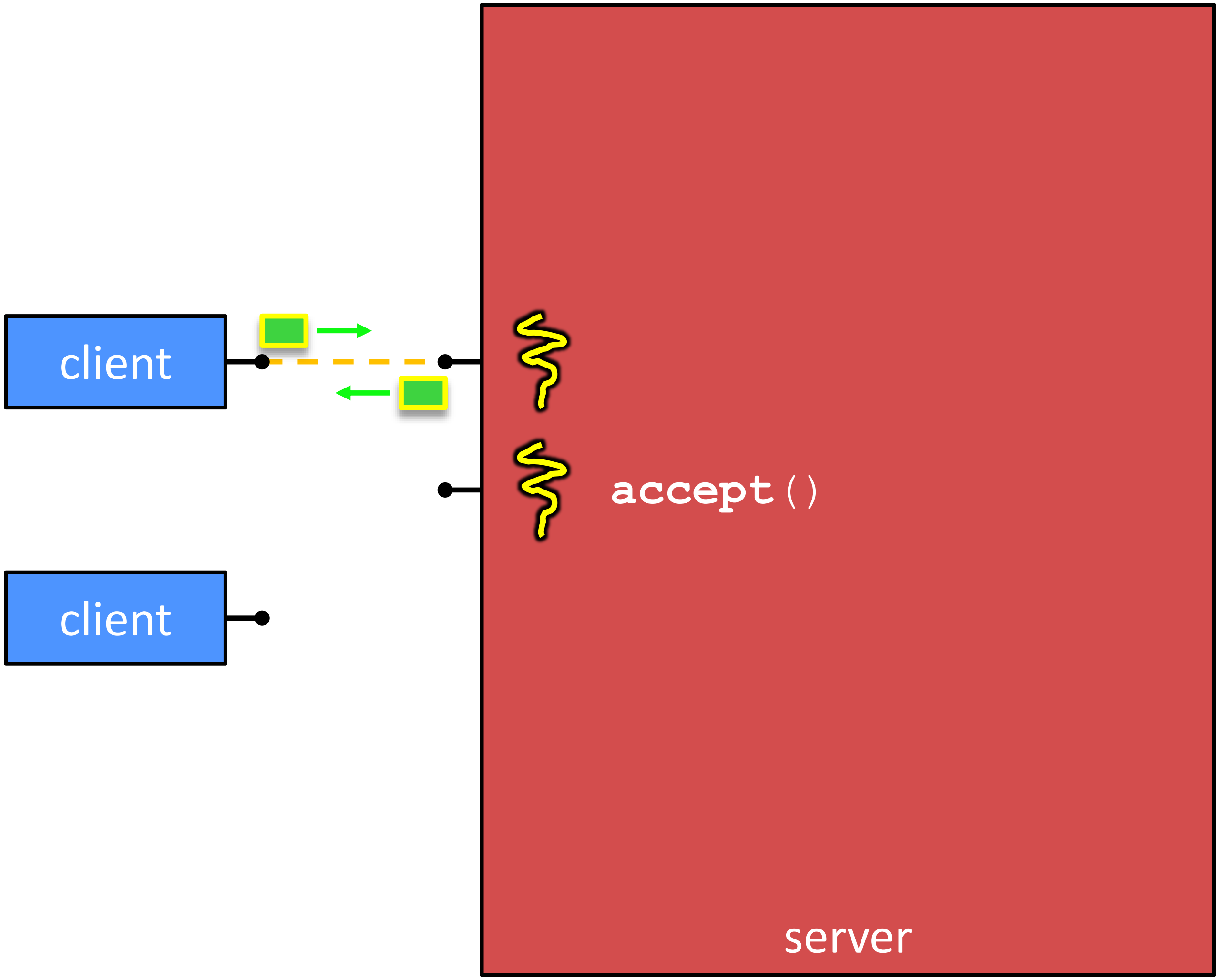
Multithreaded Server



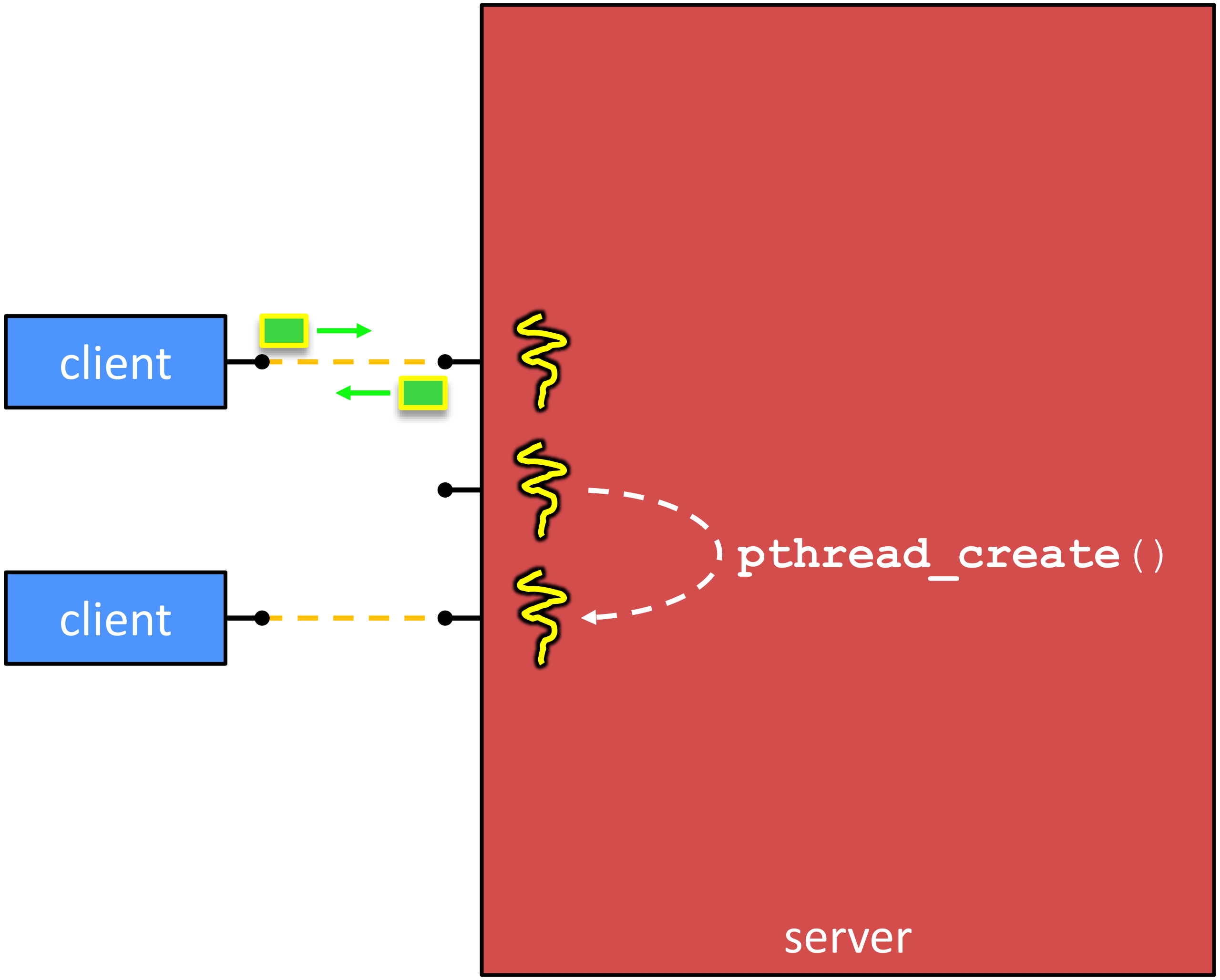
Multithreaded Server



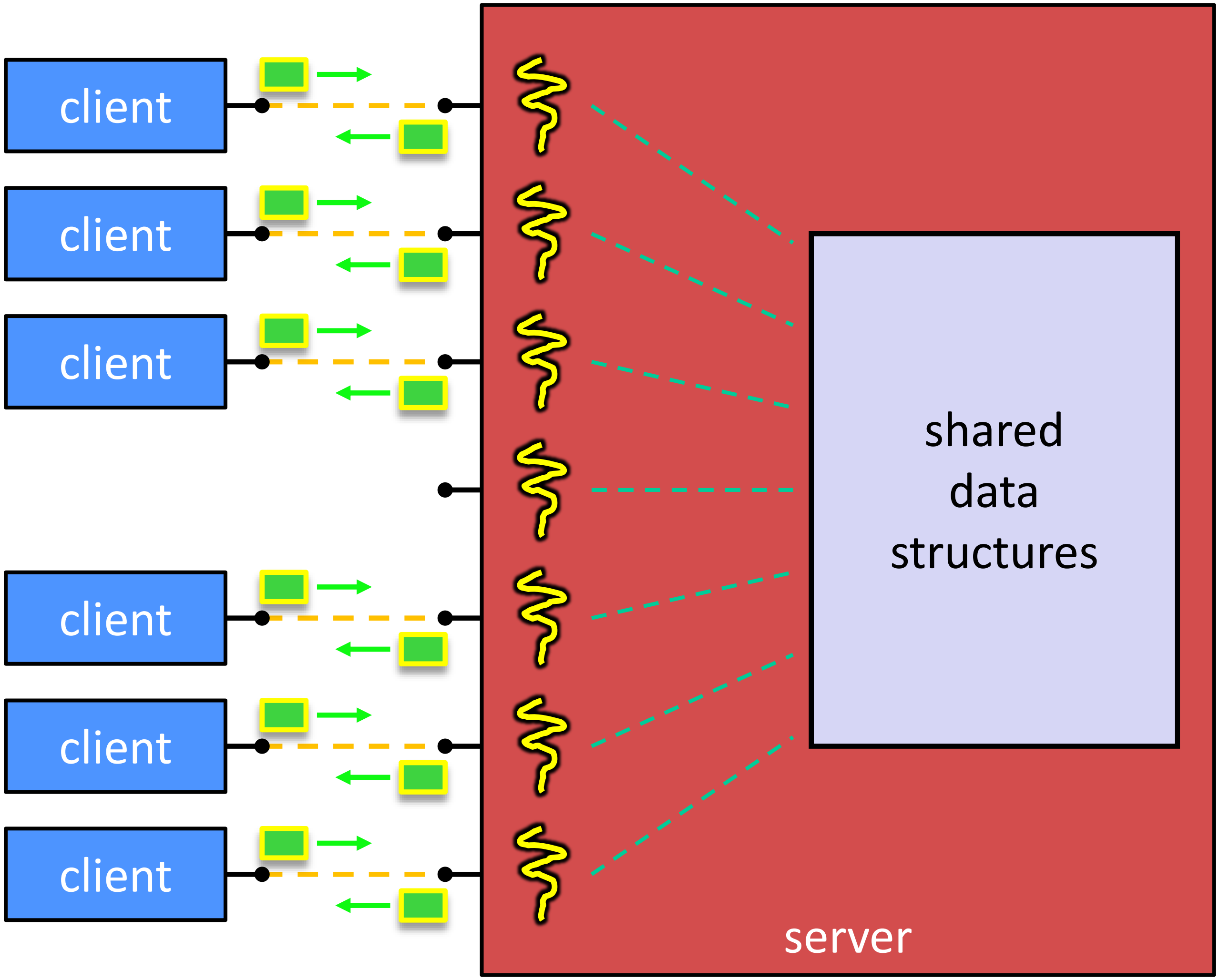
Multithreaded Server



Multithreaded Server

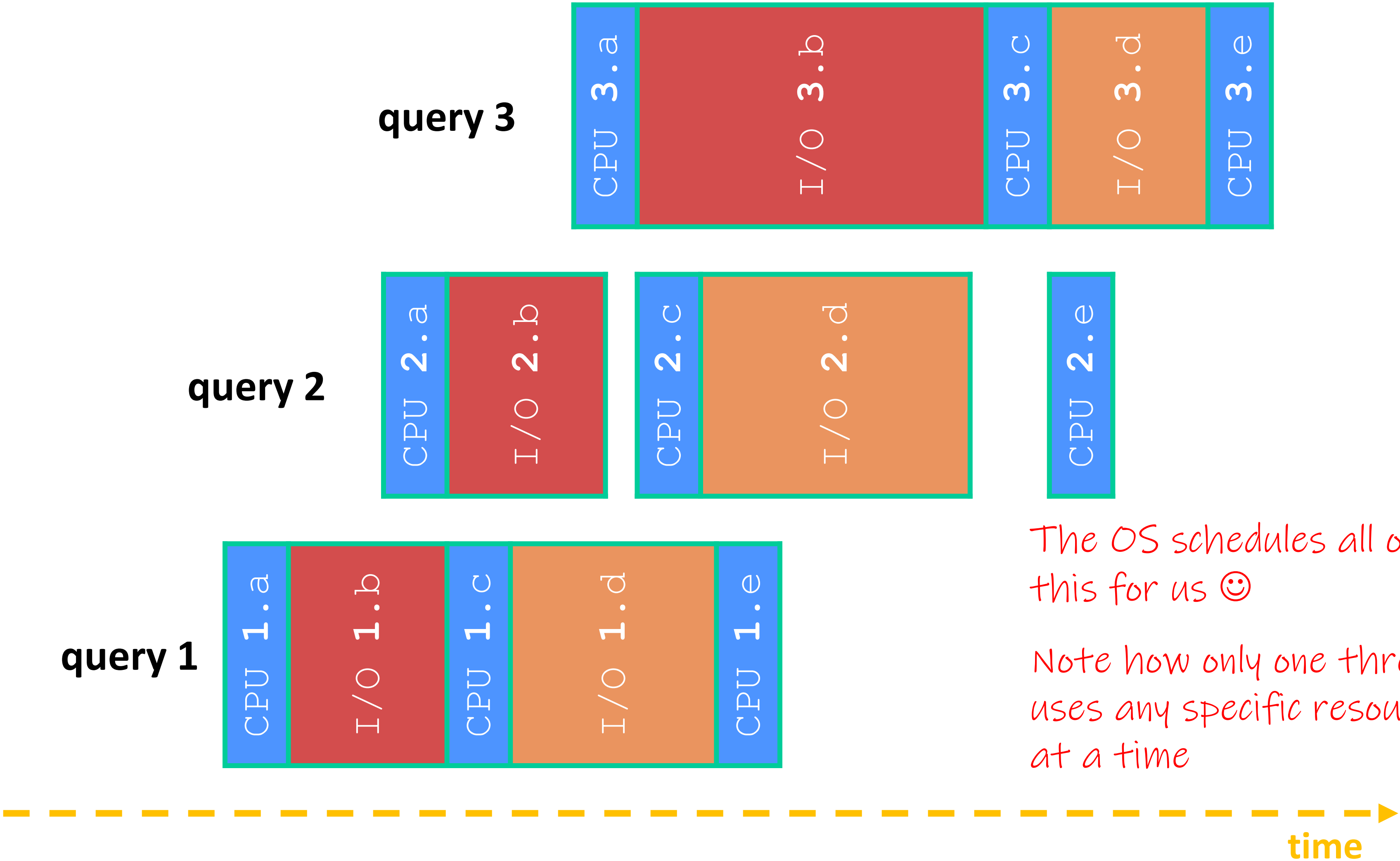


Multithreaded Server



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

MORE ON THE DISADVANTAGES LATER IN THE SEMESTER

Lecture Outline

- Threads vs processes
- Threads & Blocking
- **User Level Threads vs Kernel Level Threads**
  - **`ucontext`**
- Scheduling

Kernel Level Threads

- When the pthread library creates a new thread, it registers the new thread with the kernel
  - The new thread is stored similar to how a new process gets a new PCB
- The kernel knows about the new thread and schedules the thread for us
- Despite the name, the thread still runs in user space
- This is the default for pretty much every language

User Level Threads

- Instead of having the kernel manage threads and schedule them, we instead have the user program do this?
  - There is still a single OS thread, you can think of it as being "shared" among user level threads.
- In languages with a runtime (like Java), the runtime environment can switch between threads for us
- In C, you must switch between threads manually if you want to manage them in user land
  - Or use some user level threading library
  - You will sort of be implementing PennOS using user-level threads

Threading Models

- The “kernel level threads” approach can be called 1:1
  - For each thread we create, it is backed by the operating system, is run & scheduled by the operating system, and can be run in parallel

- The “User level threads” approach can be called N:1
  - The kernel sees the process as containing a singular thread that is scheduled and run as normal.
  - The program decides which user level thread is the one running and when to swap to another user level thread
    - This all happens while the kernel is scheduling the “1 thread” as if it is any other thread

Hybrid Threading

- Can instead have a model that is M:N
  - Create M user level threads that share N threads of execution maintained by the operating system
  - Not too common
  - Rather complex to implement yourself
  - Neat Idea ☺

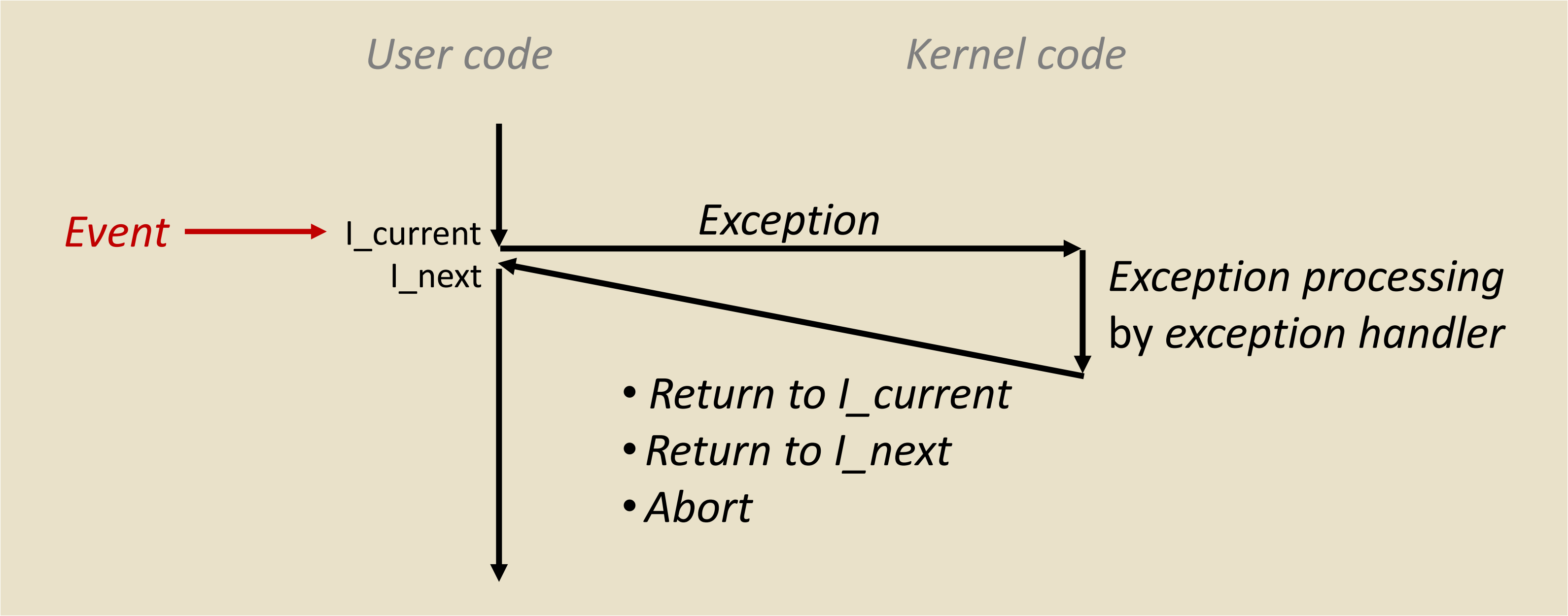
Pros & Cons of user level threads

- Pros
  - Less Operating System Overhead
  - Can customize scheduler more easily
  - If a system did not support multi threading, you can do this

- Cons
  - If a thread blocks on I/O or page fault, all user level threads
  - If you need to make sure threads share time, hard to do this without pre-emption through the kernel or some time-based signal

Interrupts

- An *Interrupt* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C



Two ways of switching between “Threads”

- There are two main ways we switch between threads.
- The most common way is to be “pre-empted”, to be interrupted, which then switches threads on the interruption
- An alternative is “cooperative”, a thread willingly gives up execution to someone else.
  - ucontext does something like this, but in PennOS we will emulate pre-emption

ucontext

```
typedef struct ucontext_t {
  struct uctonext_t *uc_link;
  sigset_t           uc_sigmask;
  stack_t            uc_stack;
  // other machine specific stuff
} ucotnext_t;
```

- Stores information about an execution context.
  - uc_sigmask stores the signal mask of the context
  - uc_stack points to the stack used by that context
  - uc_link points to the context that will be resumed when the context represented by the struct returns. NULL if we just want the process to exit.
  - Stores some other information that is machine & architecture specific. E.g. registers and their values

Getcontext & setcontext

- ```c
  int getcontext(ucontext * ucp);
  ```
  - Initializes the ucontext_t struct pointed at by ucp to have the currently active context.
  - Specifically, the context of what the calling thread would look like right after getcontext returns

- ```c
  int setcontext(const context * ucp);
  ```
  - Sets the current executing context to the one specified by ucp
  - Does not return on success, sorta like exec

Poll Everywhere

pollev.com/tqm

- What does this code do?

```
#include <ucontext.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  ucontext_t context;
  getcontext(&context);

  printf("hello\n");

  setcontext(&context);

  printf("goodbye\n");

  exit(EXIT_SUCCESS);
}
```

Getcontext & setcontext

- ```
  int makecontext(ucontext * ucp, void (*func)(),
                          int argc, ...);
  ```
  - Modifies a context (which you got from getcontext)
    - Will now call the function specified by func when context is run
  - Need to allocate a new stack for the context beforehand
  - can set new signal mask and/or uc_link

  More on this in the PennOS Demo

- ```
  int swapcontext(ucontext_t *oucp, const context *ucp);
  ```
  - Like setcontext, but stores the context of the caller into oucp

Poll Everywhere

pollev.com/tqm

- What does this code do?

```c
#define STACKSIZE 4096

void funky_function() {
  printf("HOWDY :)\n");
}

int main(int argc, char * argv[]){
  ucontext_t uc;
  getcontext(&uc);

  void * stack;
  stack = malloc(STACKSIZE);

  uc.uc_stack.ss_sp = stack;
  uc.uc_stack.ss_size = STACKSIZE;
  uc.uc_stack.ss_flags = 0;

  ucontext_t ouc;
  uc.uc_link = &ouc;

  sigemptyset(&(uc.uc_sigmask));

  makecontext(&uc, funky_function, 0);

  if (swapcontext(&ouc, &uc) != 0) {
    perror("swapcontext");
  }

  printf("Well, how did I get here?\n");

  return EXIT_FAILURE;

}
```

Lecture Outline

- Threads vs processes
- Threads & Blocking
- User Level Threads vs Kernel Level Threads
  - `ucontext`
- **Scheduling**

OS as the Scheduler

- The scheduler is code that is part of the kernel (OS)
- The scheduler runs when a thread:
  - starts ("arrives to be scheduled"),
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time
- It is responsible for scheduling threads
  - Choosing which one to run
  - Deciding how long to run it

Scheduler Terminology

- The scheduler has a scheduling algorithm to decide what runs next.

- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That "something" will eventually happen
  - Throughput: amount of work completed over an interval of time
  - Wait time: Average time a "task" is "alive" but not running
  - Turnaround time: time between task being ready and completing
  - Response time: time it takes between task being ready and when it can take user input
  - Etc…

Goals

- The scheduler will have various things to prioritize
- Some examples:
- Minimizing wait time
  - Get threads started as soon as possible
- Minimizing latency
  - Quick response times and task completions are preferred
- Maximizing throughput
  - Do as much work as possible per unit of time
- Maximizing fairness
  - Make sure every thread can execute fairly

- These goals depend on the system and can conflict

Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized
- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures
- Different tasks have different priorities
  - Higher priority tasks should finish first

Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

First Come First Serve (FCFS)

- Idea: Whenever a thread is ready, schedule it to run until it is finished (or blocks).
- Maintain a queue of ready threads
  - Threads go to the back of the queue when it arrives or becomes unblocked
  - The thread at the front of the queue is the next to run

Example of FCFS

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

- Example workload with three “jobs”:
  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:

```
|   Job 1                          |   Job 2   |   Job 3   |
 0                                 24          27          30
```

- Total waiting time: 0 + 24 + 27 = 51
- Average waiting time: 51/3 = 17
- Total turnaround time: 24 + 27 + 30 = 81
- Average turnaround time: 81/3 = 27

Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **First Come First Serve**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various lengths…

Shortest Job First (SJF)

- Idea: variation on FCFS, but have the tasks with the smallest CPU-time requirement run first
  - Arriving jobs are instead put into the queue depending on their run time, shorter jobs being towards the front
  - Scheduler selects the shortest job (1st in queue) and runs till completion

Example of SJF

1 CPU
Job 2 arrives slightly after job 1.
Job 3 arrives slightly after job 2

- Same example workload with three "jobs":
  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:

```
| Job 2 | Job 3  |    Job 1                   |
0       3        6                            30
```

- Total waiting time: 6 + 0 + 3 = 9
- Average waiting time: 3
- Total turnaround time: 30 + 3 + 6 = 39
- Average turnaround time: 39/3 = 13

Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Shortest Job First**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various lengths…

Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

Round Robin

- Sort of a preemptive version of FCFS
  - Whenever a thread is ready, add it to the end of the queue.
  - Run whatever job is at the front of the queue

- BUT only led it run for a fixed amount of time (quantum).
  - If it finishes before the time is up, schedule another thread to run
  - If time is up, then send the running thread back to the end of the queue.

Example of Round Robin

- Same example workload:

  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
- RR schedule with time quantum=2:

```
|Job 1|Job 2|Job 3|Job 1|Jo2|Jo3|Job 1| …     |Job 1|
 0     2     4     6     8   9   10    12,14…       30
```

- Total waiting time: (0 + 4 + 2) + (2 + 4) + (4 + 3) = 19
  - Counting time spent waiting between each "turn" a job has with the CPU
- Average waiting time: 19/3 (~6.33)
- Total turnaround time: 30 + 9 + 10 = 49
- Average turnaround time: 49/3 (~16.33)

Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Round Robin**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing Latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context Switch

- Imagine we have 1 core, and tasks of various lengths…

More

- More next lecture