CS 162: Operating Systems and Systems Programming

Lecture 24: Lang. Support for Concurrency, Intro to Distributed Systems

August 6, 2019

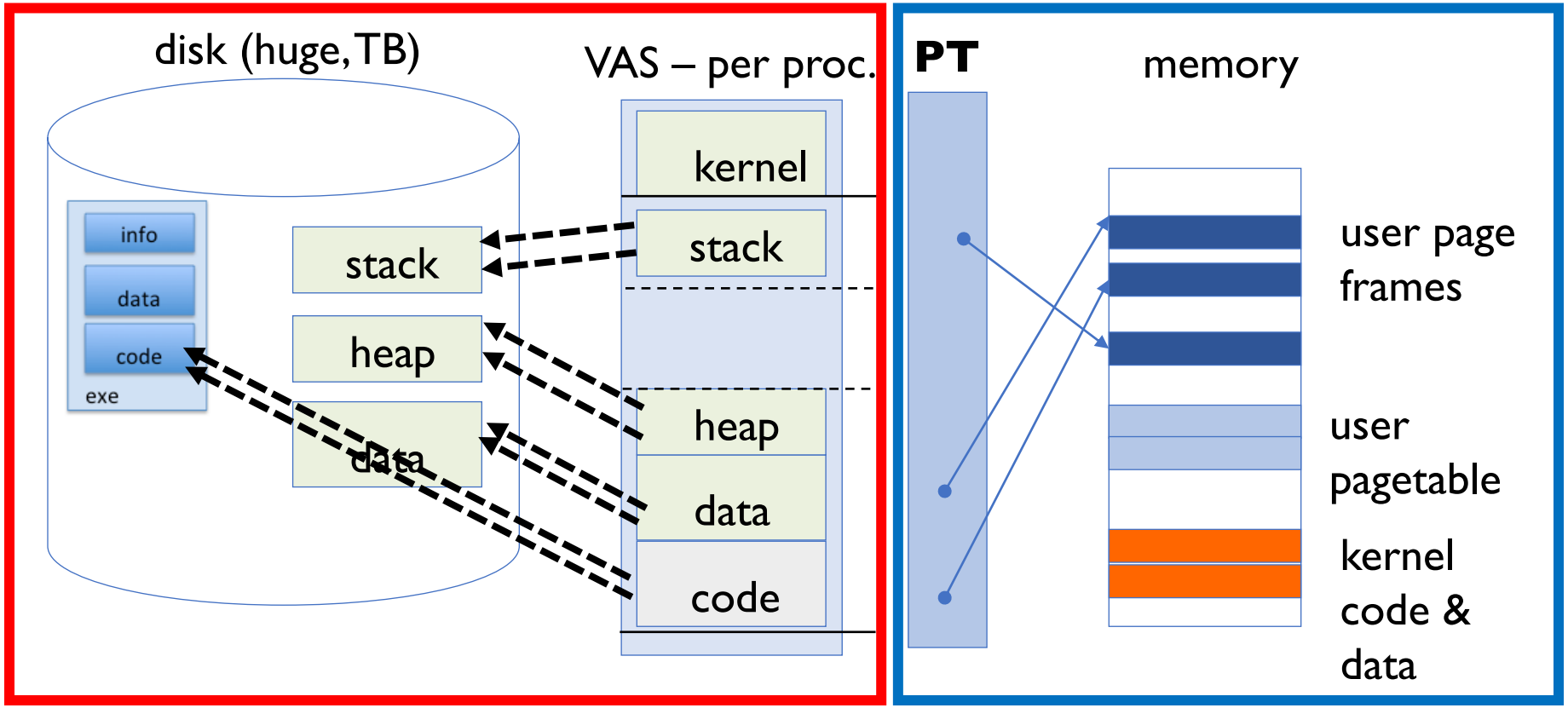
Instructor: Jack Kolb

https://cs162.eecs.berkeley.edu

Logistics

- Proj 3 Due on August 12
  - Milestone coming up on Tuesday
- HW3 Due on August 13
  - Last Tuesday of the class
- Course Evaluations at https://course-evaluations.berkeley.edu/

Executing a Program: Create Address Space



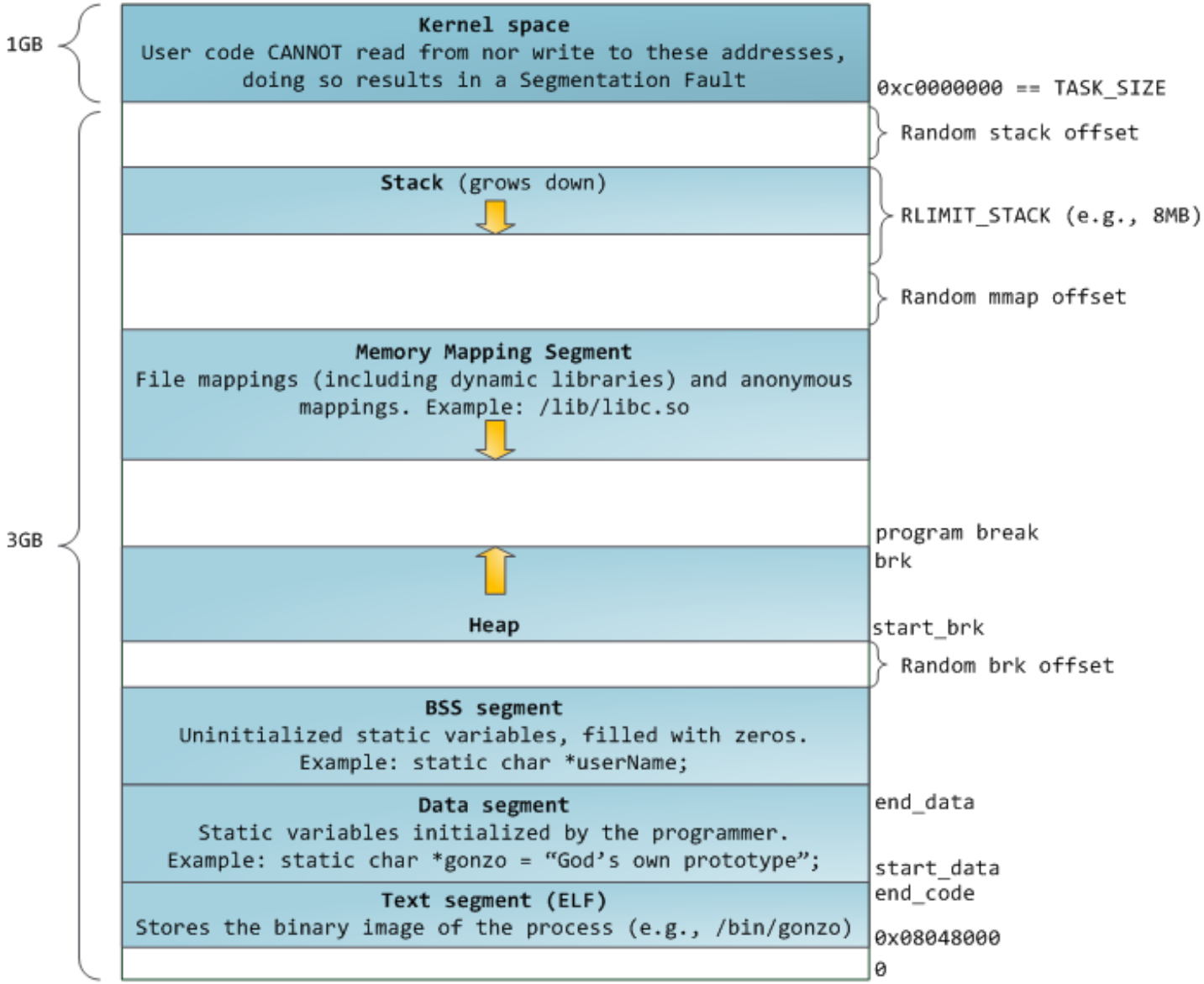
Backing Store **Cache**

Recall: Memory-Mapped IO

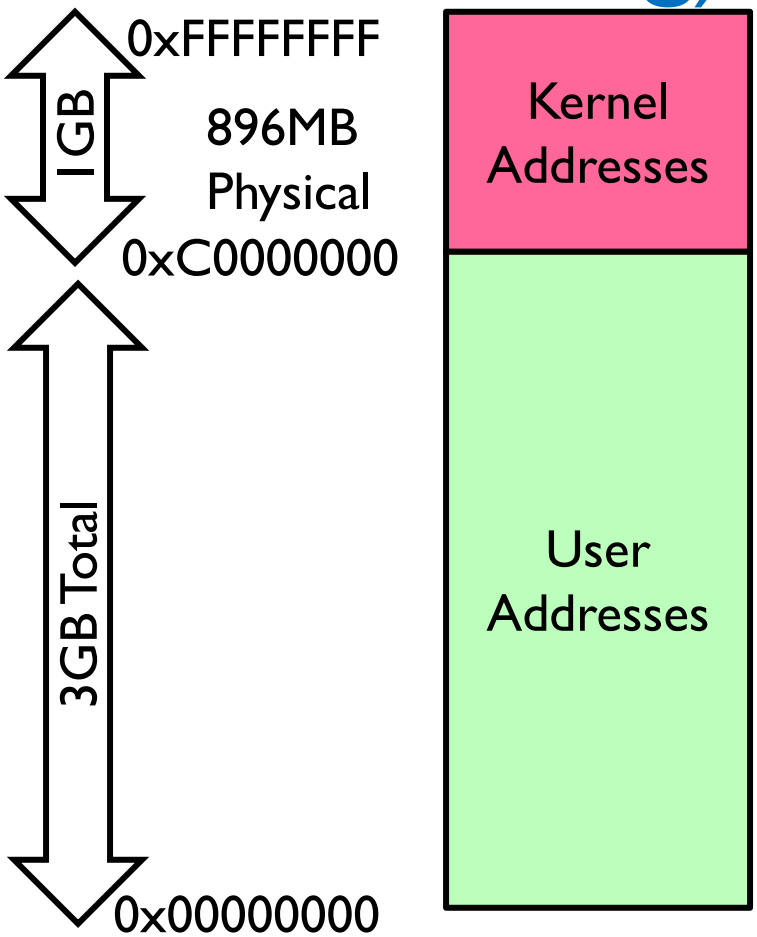
- IO so far: Explicit transfer between buffers in process address space to regions of a file
- Overhead: multiple copies in memory, syscalls

- Alternative: Map file directly into an empty region of a process address space
  - Implicitly page in file when we read it
  - Write to file and eventually page it out
- Executable file is treated this way when we execute a process!

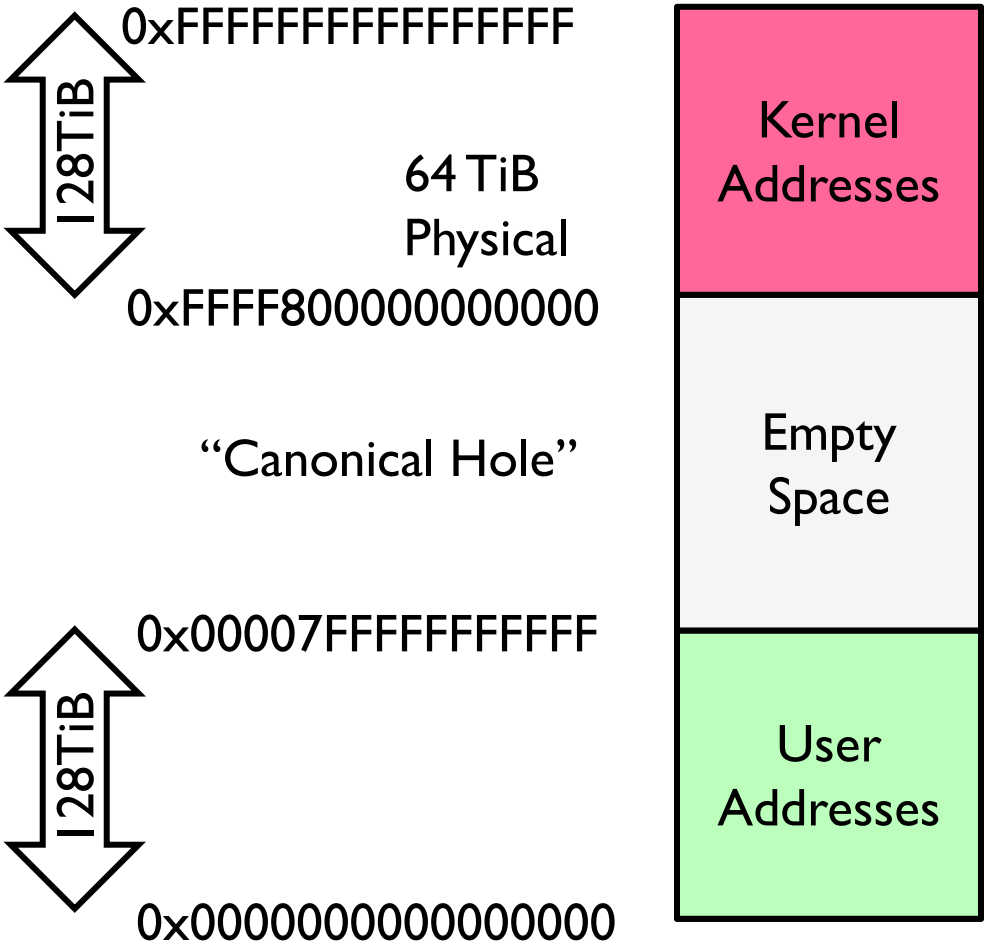
32-bit x86 Linux Memory Layout



Linux Virtual memory map (pre-Meltdown bug)



32-Bit Virtual Address Space



64-Bit Virtual Address Space

Recall: Interprocess Communication

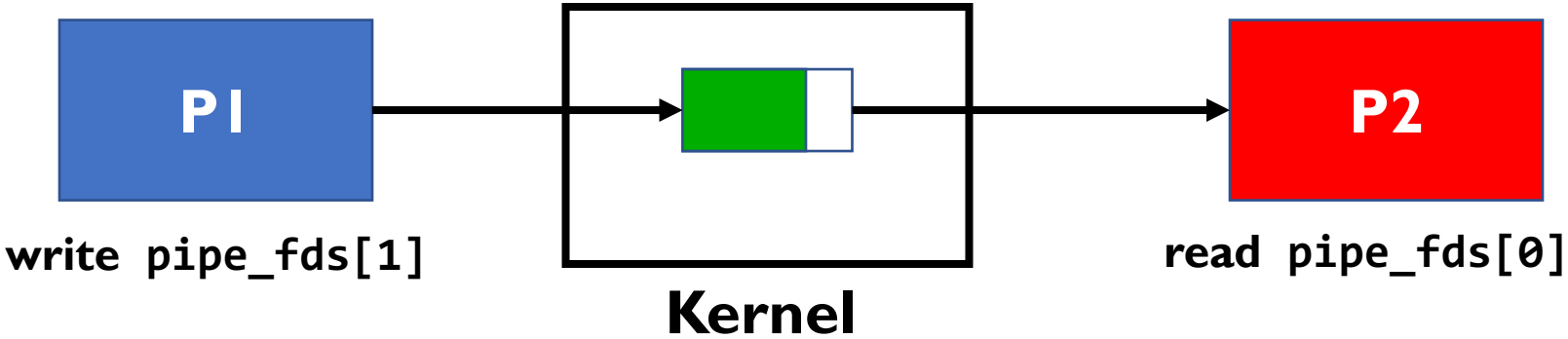
- Allow two (or more) processes to exchange information with each other

- Why use this approach rather than multithreading?
- Keep most of the benefits of process isolation
- Expose processes to each other only through a *carefully structured* interface
  - Ex: Google Chrome Design

IPC Example: Pipes

- Use classic file-related syscalls: **`read`**, **`write`**
- But avoid the overhead of actually interacting with kernel IO subsystem
- Instead: writes/reads manipulate a buffer of memory maintained by the kernel

Using Pipes



- Remember producer-consumer problem?
- Pipe's buffer has maximum size
  - Write to full pipe blocks until space available
  - Read from empty pipe blocks until data available
- Read from pipe with no writers returns 0
- Write to pipe with no readers prompts `SIGPIPE`

UNIX Domain Sockets

- Open a socket connection with a **local** process
- Use familiar `write`/`read` calls to communicate
- But **don't** incur usual **overhead** of networking
- **Optimized** for processes on same machine

Using Unix Domain Sockets

- Still need same sequence of syscalls: **`socket`**, **`bind`**, **`listen`**, **`accept`** to act as a server

- But socket address now corresponds to an object in local machine's filesystem
  - Specify path on `bind`
- Why this approach?
  - Filesystem gives us a *namespace*: any process can specify path on `connect` (doesn't need to be a child of server)
  - Filesystem enforces *permissions*

C Concurrency and Synch.

- Standard approach: use **`pthreads`**, protect access to shared data structures
- *Shared Memory Paradigm*
- One pitfall: consistently unlocking a mutex

```
int Rtn() {
  lock.acquire();
  …
  if (error) {
    lock.release();
    return errCode;
  }
  …
  lock.release();
  return OK;
}
```

Other Languages and Threading

- Many other mainstream languages also focus on threads and shared memory

- But offer useful libraries and built-in features to make our lives easier
  - Thread management libraries
  - Thread pools
  - Safer lock management
  - Objects as monitors

C++ Lock Guards

```
#include <mutex>
int global_i = 0;
std::mutex global_mutex;

void safe_increment() {
  std::lock_guard<std::mutex> lock(global_mutex);
  …
  global_i++;
  // Mutex released when 'lock' goes out of scope
}
```

Python `with` Keyword

- More versatile than we'll show here (can be used to close files, database connections, etc.)

```
lock = threading.Lock()
…
with lock: # Automatically calls acquire()
  some_var += 1
  …
# release() called however we leave block
```

Java Support for Synchronization

```
class Account {
  private int balance;


  public Account (int initialBalance) {
    balance = initialBalance;
  }
  public synchronized int getBalance() {
    return balance;
  }
  public synchronized void deposit(int amount) {
    balance += amount;
  }
}
```

- Every Java object has an associated lock for synchronization:
  - Lock is acquired on entry and released on exit from synchronized method
  - Lock is properly released if exception occurs inside synchronized method

Java Support for Synchronization

- Along with a lock, every object has a **single** condition variable associated with it
- To wait inside a synchronized method:
  - `void wait();`
  - `void wait(long timeout);`
- To signal while in a synchronized method:
  - `void notify();`
  - `void notifyAll();`

Go Programming Language

- "Goroutines": Lightweight, user-level threads
- Channels: Named message queues for communication among threads
  - Given a *type* (send and recv instances)
- Key Idea: Prefer *message passing* over *shared memory*

Go Programming Language

Why this approach?

- Efficiency of a shared address space
- Tolerates many threads in one program

- Passing data through channels: no need for explicit synchronization
  - Sender *passes ownership* to receiver

Why are we using Go for HW3?

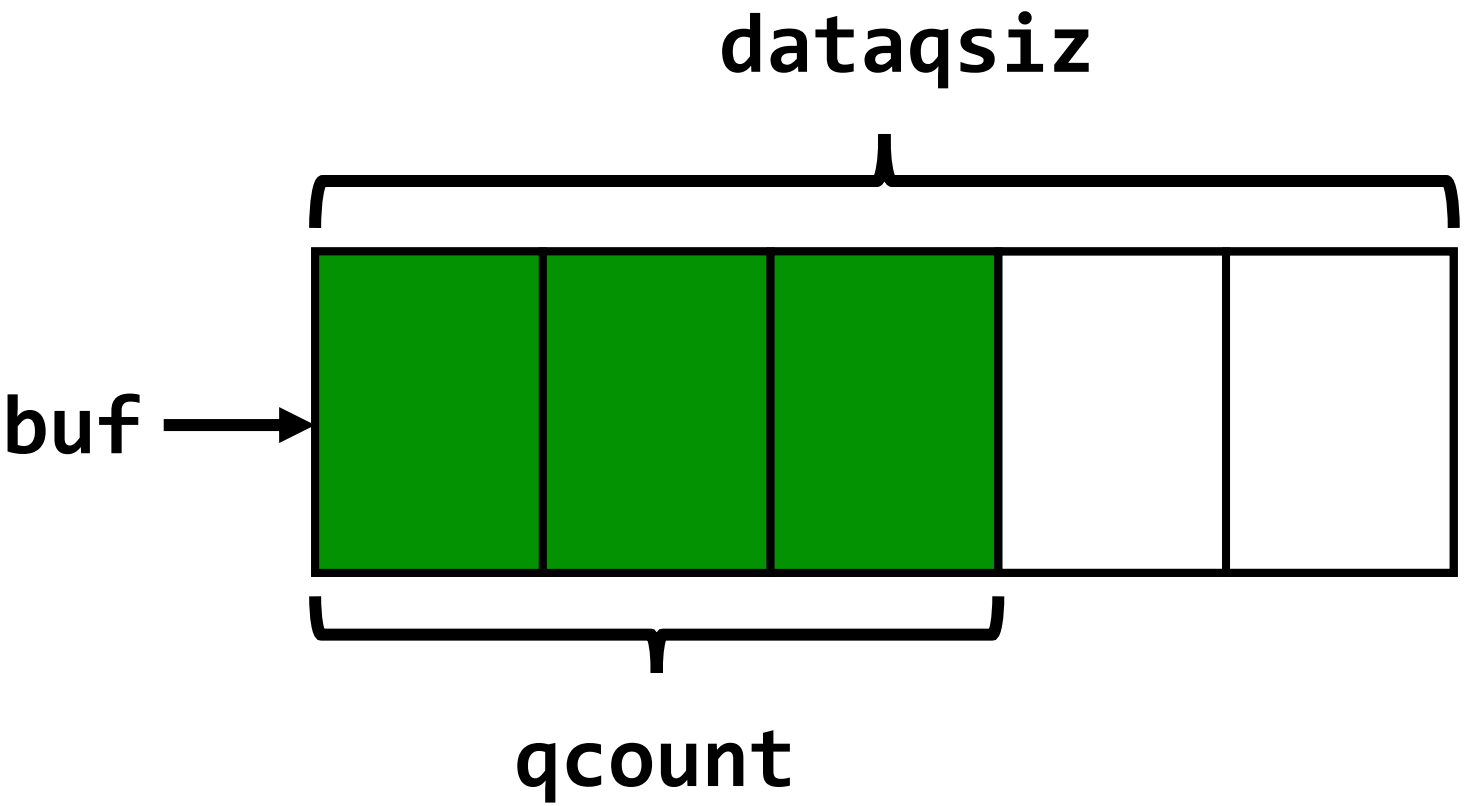
- Becoming a major tool for modern systems programming (e.g., networked systems)
  - Garbage Collection (unlike C)
  - Compiles to native machine code (like C)
  - Versatile libraries (arguably unlike C)
  - Simple feature set (like C)

- Language abstractions influence how you think about and solve problems

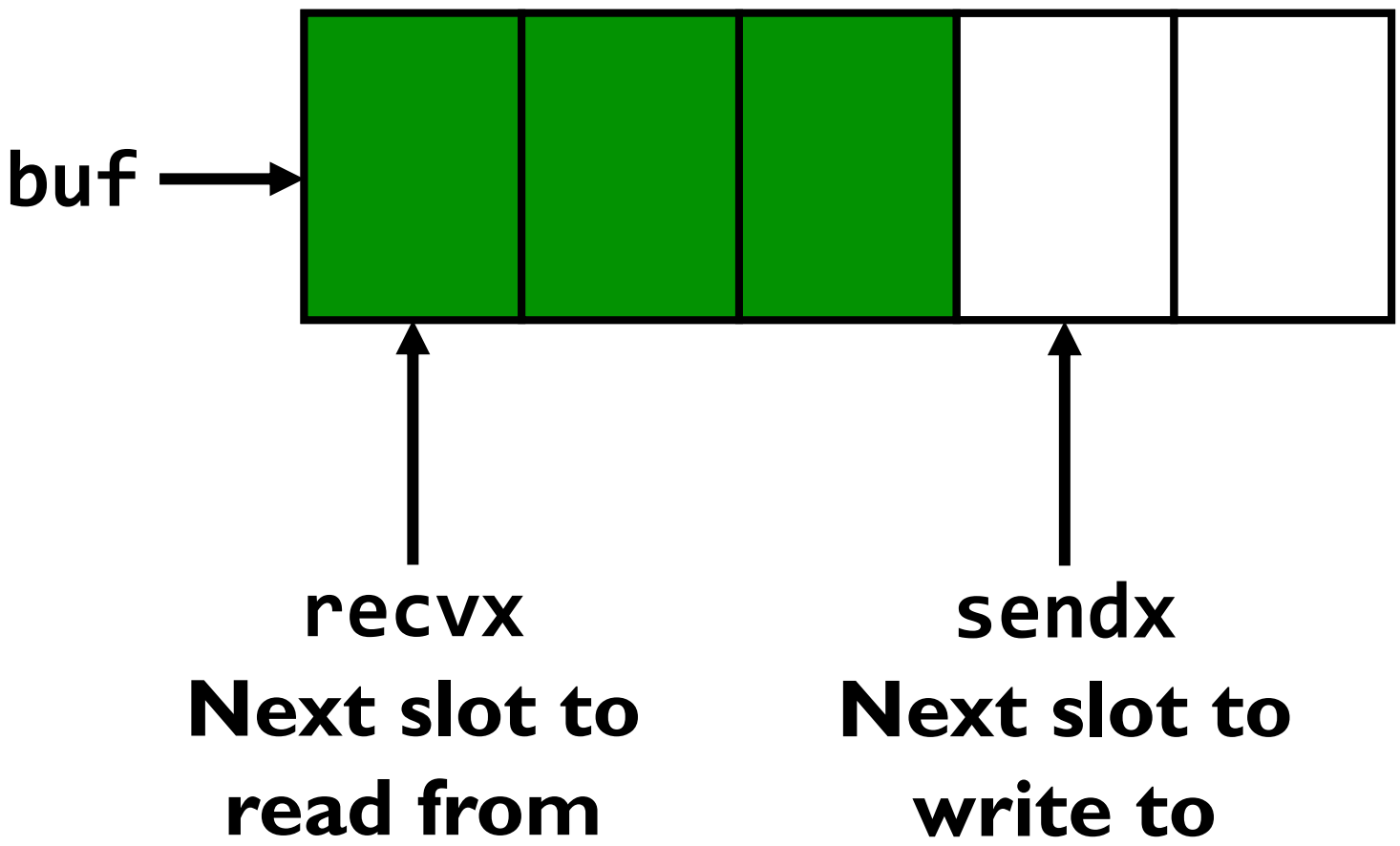
Go Channels

```
type hchan struct {
  qcount   uint // total data in the queue
  dataqsiz uint // size of the circular queue
  buf      unsafe.Pointer // array of dataqsiz elements
  elemsize uint16
  closed   uint32
  elemtype *_type // element type
  sendx    uint   // send index
  recvx    uint   // receive index
  recvq    waitq  // list of recv waiters
  sendq    waitq  // list of send waiters
  lock mutex
}
```

Go Channels



Go Channels



Go Channels

- Rules much like for pipes
- Synchronization handled for us
  - Use atomic ops or acquire channel's `mutex`
- Send: If `recvq` non-empty, pass value to first waiter and wake it up
  - Otherwise, append element to buffer at `sendx`
- Send when full buffer (`qcount == dataqsize`)
  - Put thread on `sendq`
  - Schedule another thread

Go Channels

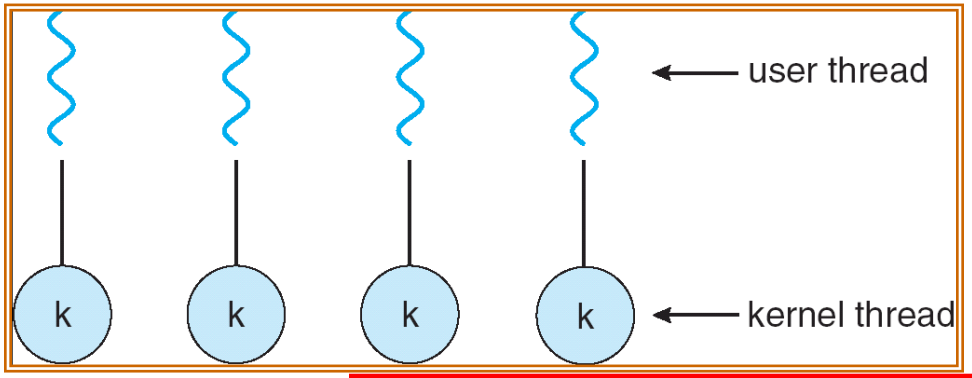
- Rules much like for pipes
- Synchronization handled for us
  - Use atomic ops or acquire channel's `mutex`
- Recv: If buffer non-empty, read from `recvx`
  - If thread on `sendq`, append its element to buffer, remove it from queue, mark as ready again
- Recv when empty buffer (`qcount == 0`)
  - Put thread on `recvq`
  - Schedule another thread

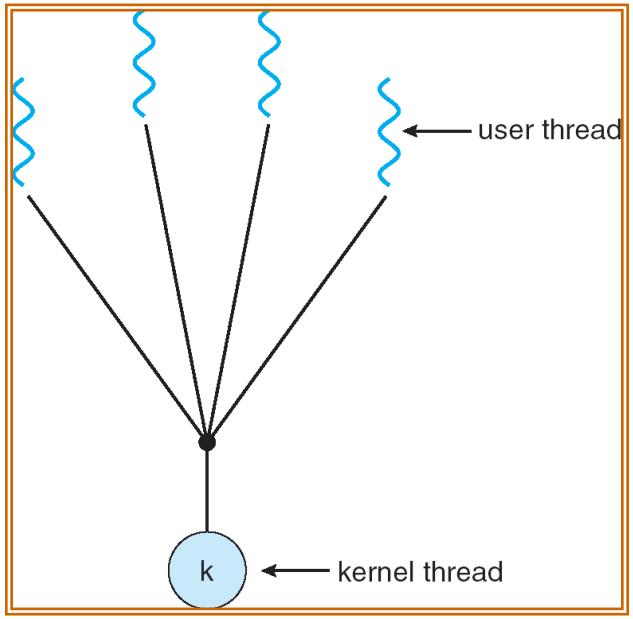
Go Channels

- Closing a channel much like closing a pipe
  - All waiting readers woken up, "receive" zero value
  - All waiting writers woken up, panic (~exception) occurs in each
- What if we give a channel a buffer length of 0?
  - *Synchronous channel*
  - Send blocks until another thread receives

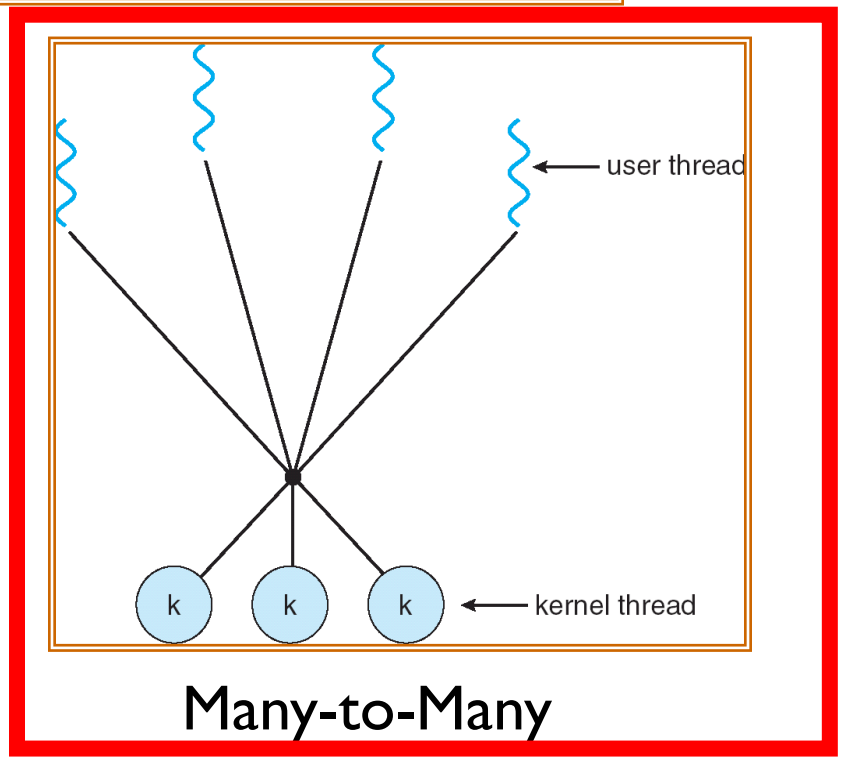
Remember this Slide?

Simple One-to-One Threading Model





Many-to-One

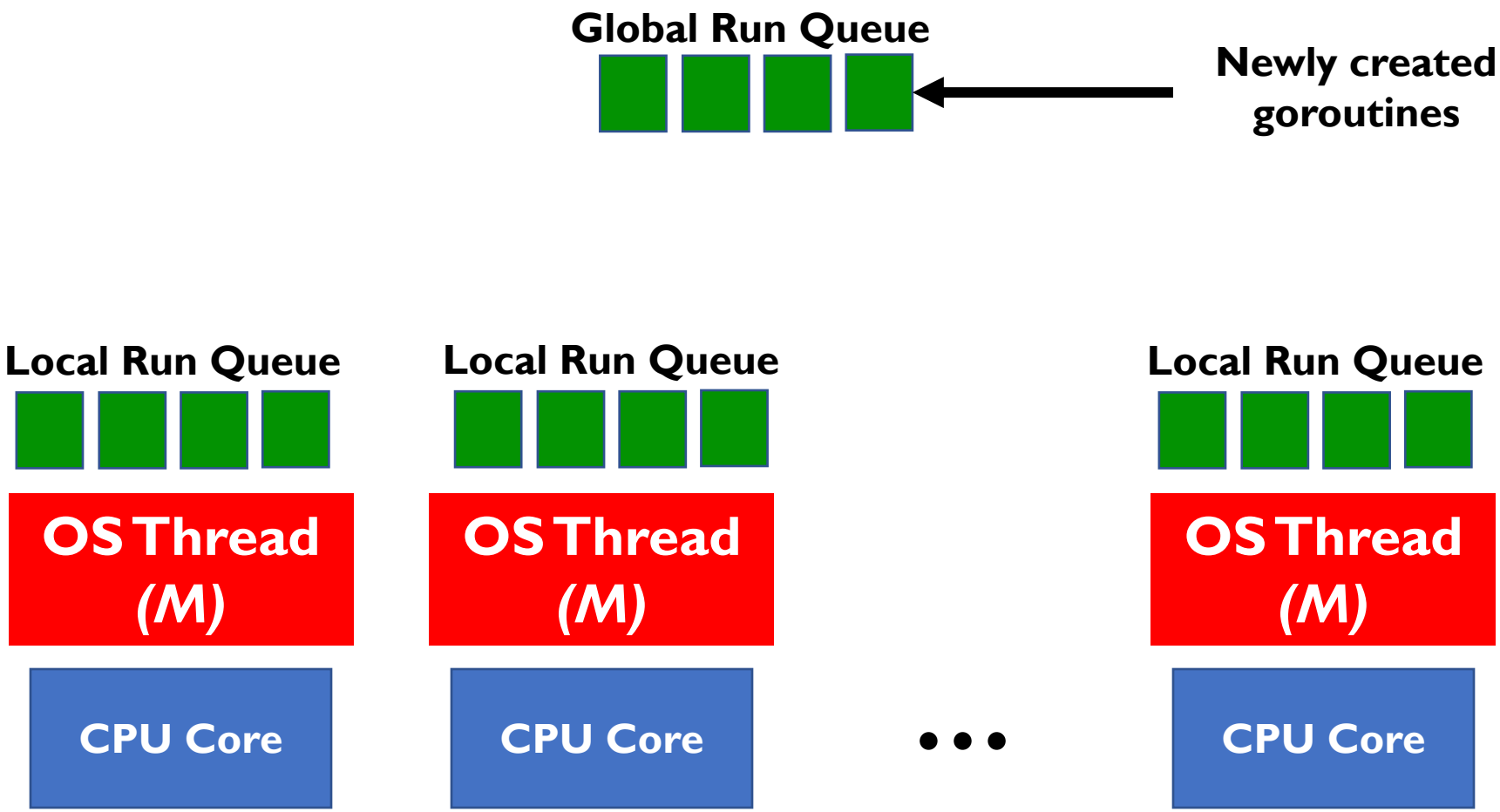


Many-to-Many

User-Mode Threads: Problems

- One user-level thread blocks on syscall: all user-level threads relying on same kernel thread also block
  - Kernel cannot intelligently schedule threads it doesn't know about
- Multiple Cores?
- No pre-emption: User-level thread must explicitly yield CPU to allow someone else to run

Go User-Level Thread Scheduler



Go User-Level Thread Scheduler

Why this approach?

- 1 OS (kernel-supported) thread per CPU core: allows go program to achieve *parallelism* not just *concurrency*
  - Fewer OS threads? Not utilizing all CPUs
  - More OS threads? No additional benefit
    - We'll see one exception to this involving syscalls

- Keep goroutine on same OS thread: *affinity*, nice for caching and performance

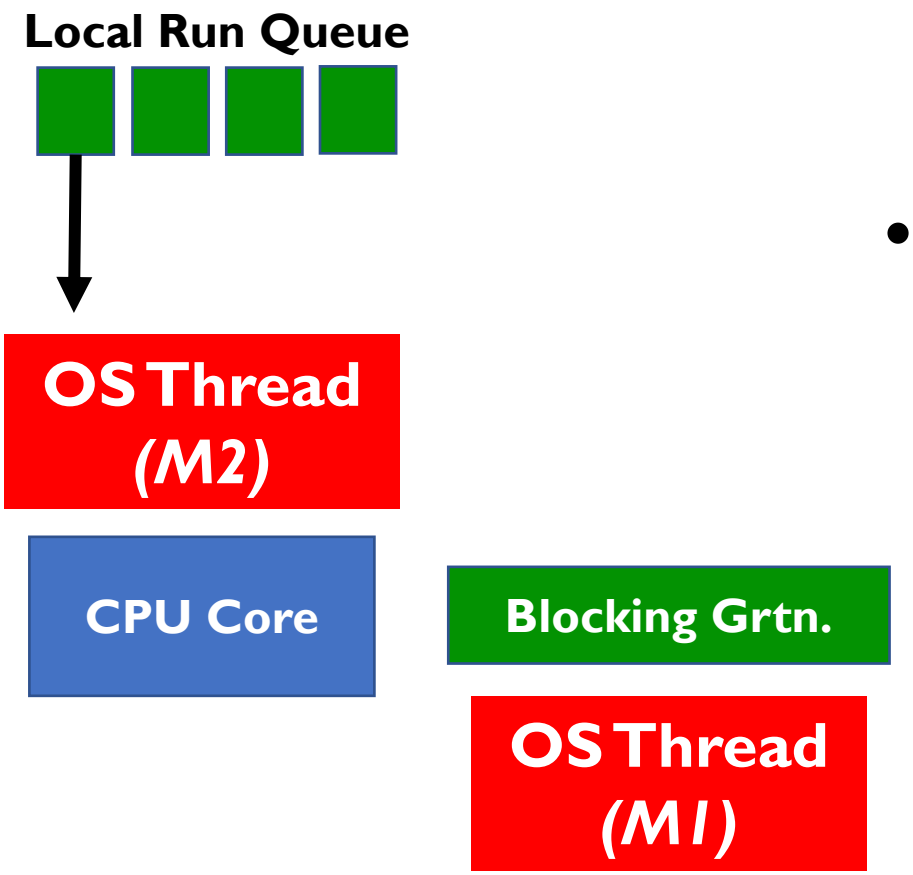
Cooperative Scheduling

- No pre-emption => goroutines must yield to allow another thread to run
- Programmer does **not** need to do this explicitly
- Go runtime injects yields at safe points in execution
  - Sending/receiving with a channel
  - Acquiring a mutex
  - Making a function call
- But your code can still tie up the scheduler in "tight loops" (Go's developers are working on this…)

Dealing with Syscalls

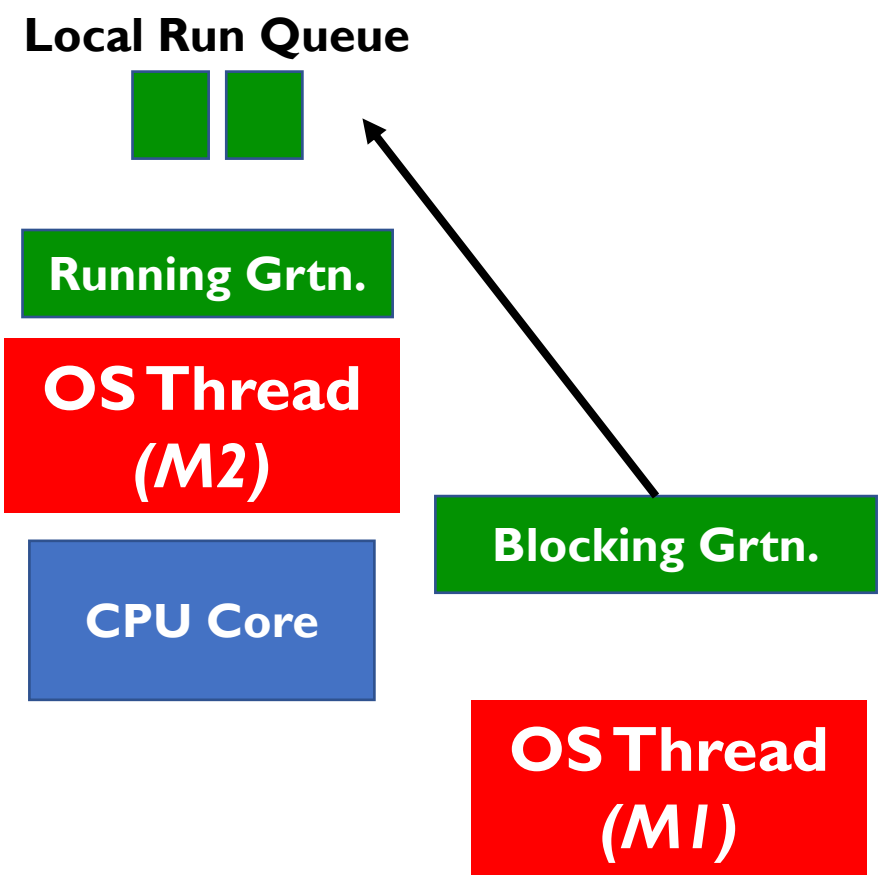
- What if a goroutine wants to make a blocking syscall?
- Example: File I/O

Dealing with Syscalls



- While syscall is blocking, allocate new OS thread (M2)
- M1 is blocked by kernel, M2 lets us continue using CPU

Dealing with Syscalls



- Syscall completes: Put invoking goroutine back on queue
- Keep *M1* around in a spare pool
- Swap it with *M2* upon next syscall, no need to pay thread creation cost

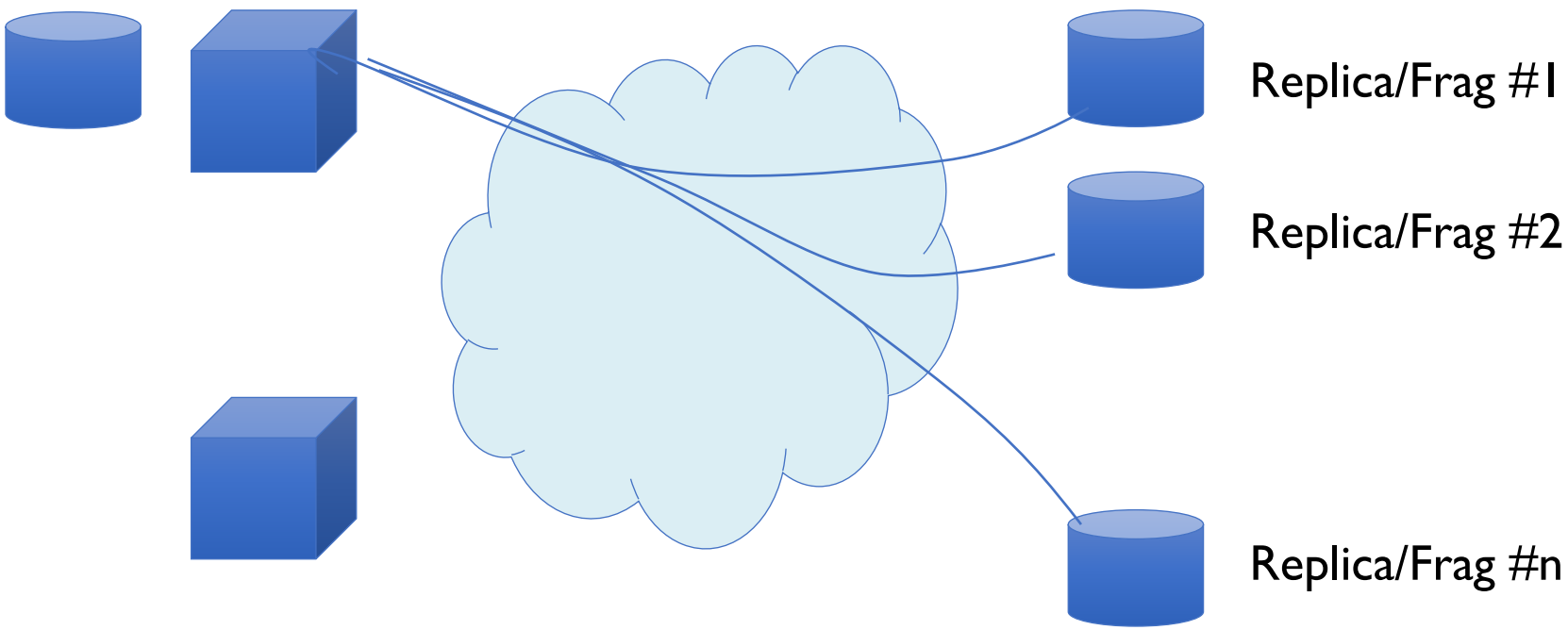
Break

Important "ilities"

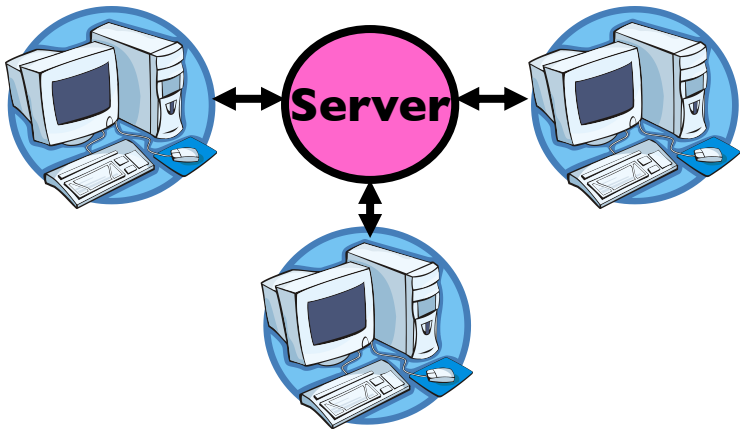
- Availability: probability that the system can accept and process requests
- Durability: the ability of a system to recover data despite faults
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)

One Approach: Geographic Replication

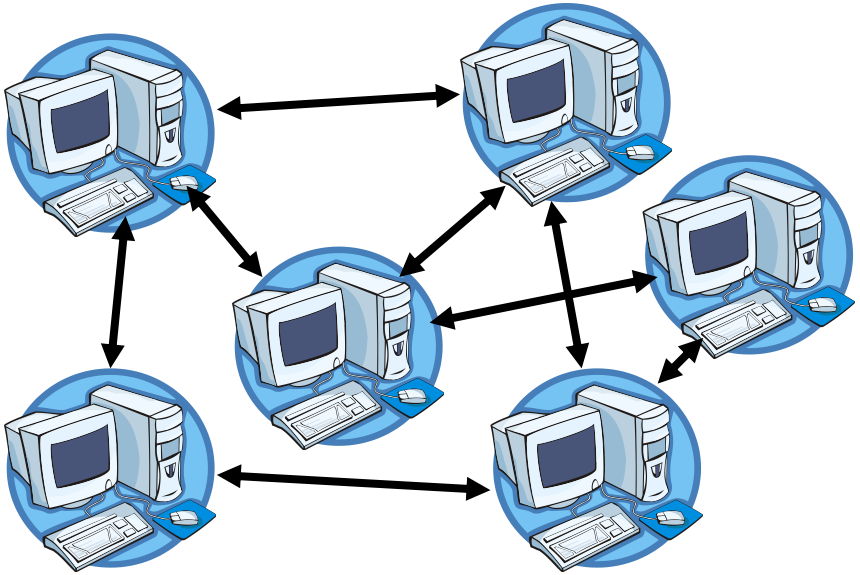
- Highly durable: Hard to destroy all copies
- Highly available for reads: Just talk to any copy
- What about for writes? Need every copy online to update all together?



Centralized vs Distributed



Client/Server Model



Peer-to-Peer Model

- **Centralized System:** Major functions performed on one physical computer
- **Distributed System:** Physically separate computers working together to perform a single task

Parallel vs Distributed

- Distributed: different machines responsible for different parts of task
  - Usually no centralized state
  - Usually about different responsibilities or redundancy
- Parallel: different parts of same task performed on different machines
  - Usually about performance

Distributed: Why?

- Simple, **cheaper** components
- Easy to add capability **incrementally**
- Let multiple users cooperate (maybe)
  - Physical components owned by different users
  - Enable **collaboration** between diverse users

The Promise of Dist. Systems

- **Availability:** One machine goes down, overall system stays up
- **Durability:** One machine loses data, but *system* does not lose anything
- **Security:** Easier to secure each component of the system individually?

Distributed: Worst-Case Reality

- **Availability:** Failure in one machine brings down entire system
- **Durability:** Any machine can lose your data
- **Security:** More components means more points of attack

Distributed Systems Goal

- **Transparency:** Hide "distributed-ness" from any external observer, make system simpler
- **Types**
  - Location: Location of resources is invisible
  - Migration: Resources can move without user knowing
  - Replication: Invisible extra copies of resources (for reliability, performance)
  - Parallelism: Job split into multiple pieces, but looks like a single task
  - Fault Tolerance: Components fail without users knowing

Challenge of Coordination

- Components communicate over the network
  - Send messages between machines

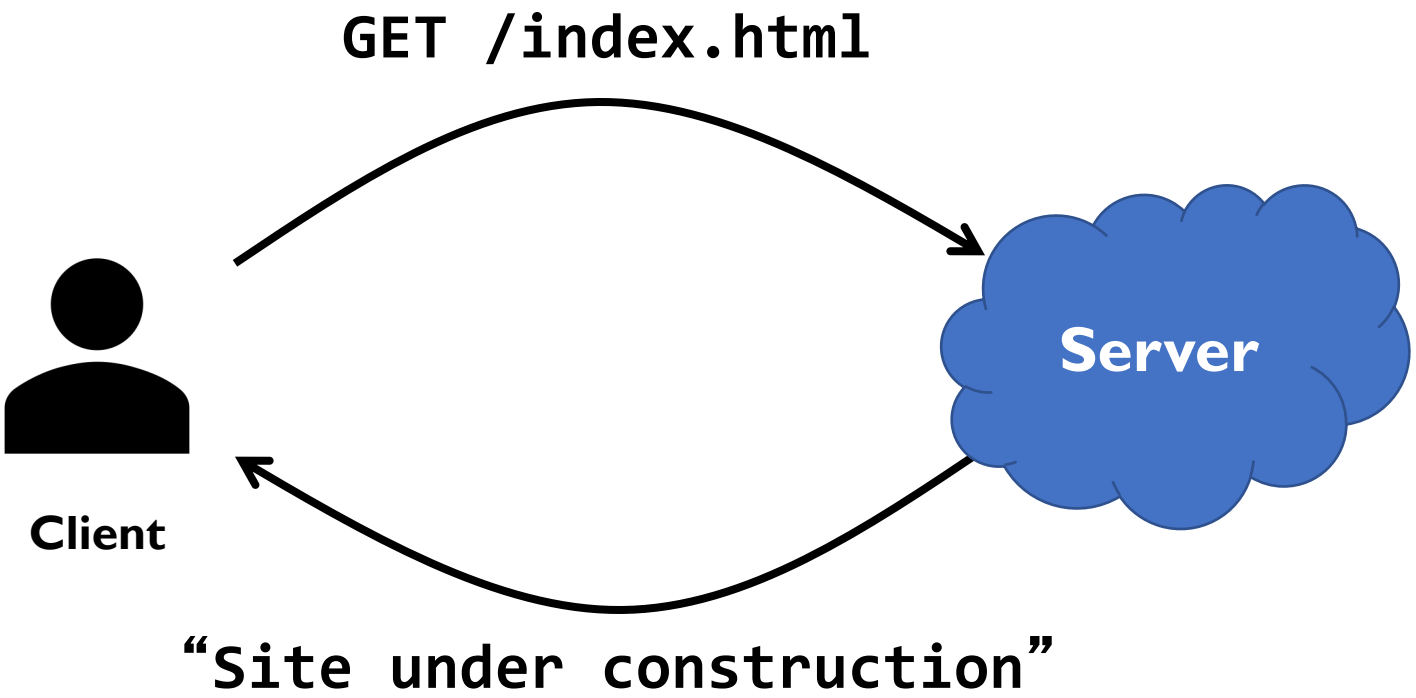
- Need to use messages to **agree on system state**
  - This issue does not exist in a centralized system

Recall: What is a Protocol?

- **An agreement on how to communicate**
  - **Syntax:** Format, order messages are sent and received
  - **Semantics:** Meaning of each message
- Described formally by a state machine
- A distributed system is embodied by a protocol

Clients and Servers

- Client program
  - Running on end host
  - Requests service
  - E.g., Web browser
- Server program
  - Running on end host
  - Provides service
  - E.g., Web server



Client-Server Communication

- Client "sometimes on"
  - Initiates a request to the server when interested
  - E.g., Web browser on your laptop or cell phone
  - Doesn't communicate directly with other clients
  - Needs to know the server's address
- Server is "always on"
  - Services requests from many client hosts
  - E.g., Web server for the *www.berkeley.edu*
  - Doesn't initiate contact with the clients
  - Needs a fixed, well-known address

Peer-to-Peer Communication

- No always-on server at the center of it all
  - Hosts may come and go, change addresses
  - Hosts may have a different address for each interaction with the system
- Example: Peer-to-peer file sharing (BitTorrent)
  - Any host can request files, send files, search for files
  - Scalability by harnessing millions of peers
  - Each peer acting as both client and server

Summary

- C++, Python, Java all offer support for more convenient, robust synchronization primitives
- Go: Exchange messages instead of sharing mem.
  - Channels: thread-safe, named queues
  - Goroutines: Cleverly managed user-level threads
- Distributed Systems built from many separate, communicating components
  - Many new challenges, e.g. agreeing on state
  - Client-server vs. peer-to-peer models