Advanced Networking and Distributed Systems

Module 2: Scalable Servers and Network Performance

GW CSCI 3907/6907
Timothy Wood and Lucas Chaufournier

Outline

Weeks 1-3: Network Programming and Protocols

- Writing simple network programs is easy!
- Providing reliable services over a network is hard!

Weeks 4-5: Scalability and Performance

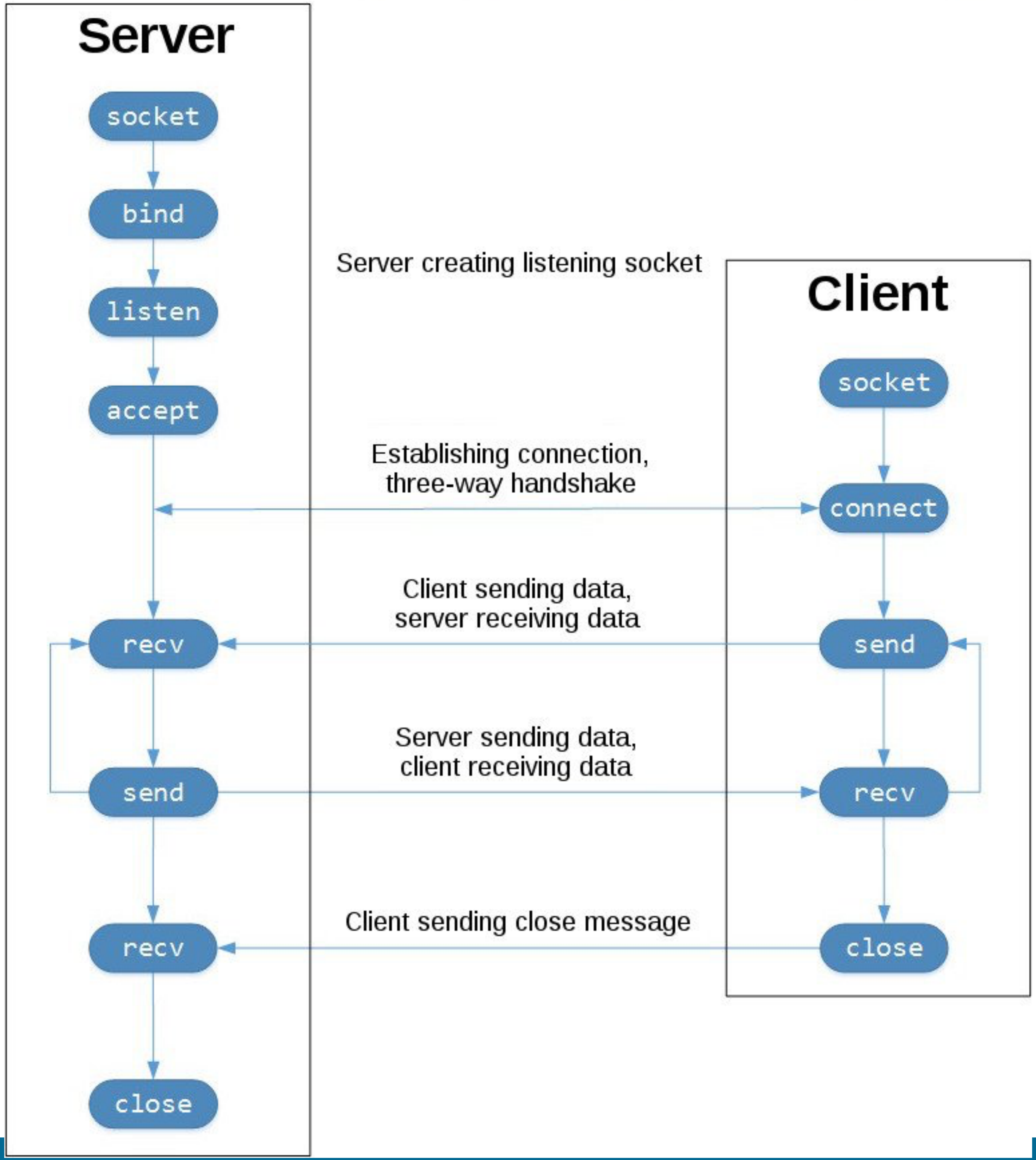
- How can we support many concurrent clients?
- What performance metrics matter for network services?

Weeks 6-7: Network Middleboxes

- How to deploy software *between* clients and servers?
- How to get the speed of HW and flexibility of SW?

Server Architecture

How many clients can this server handle at once?



Simplest Architecture

Server is a single thread

Network calls are blocking (**`recv`**, **`accept`**)

-> server can only handle one client at a time

What happens to other clients who try to connect?

Simplest Architecture

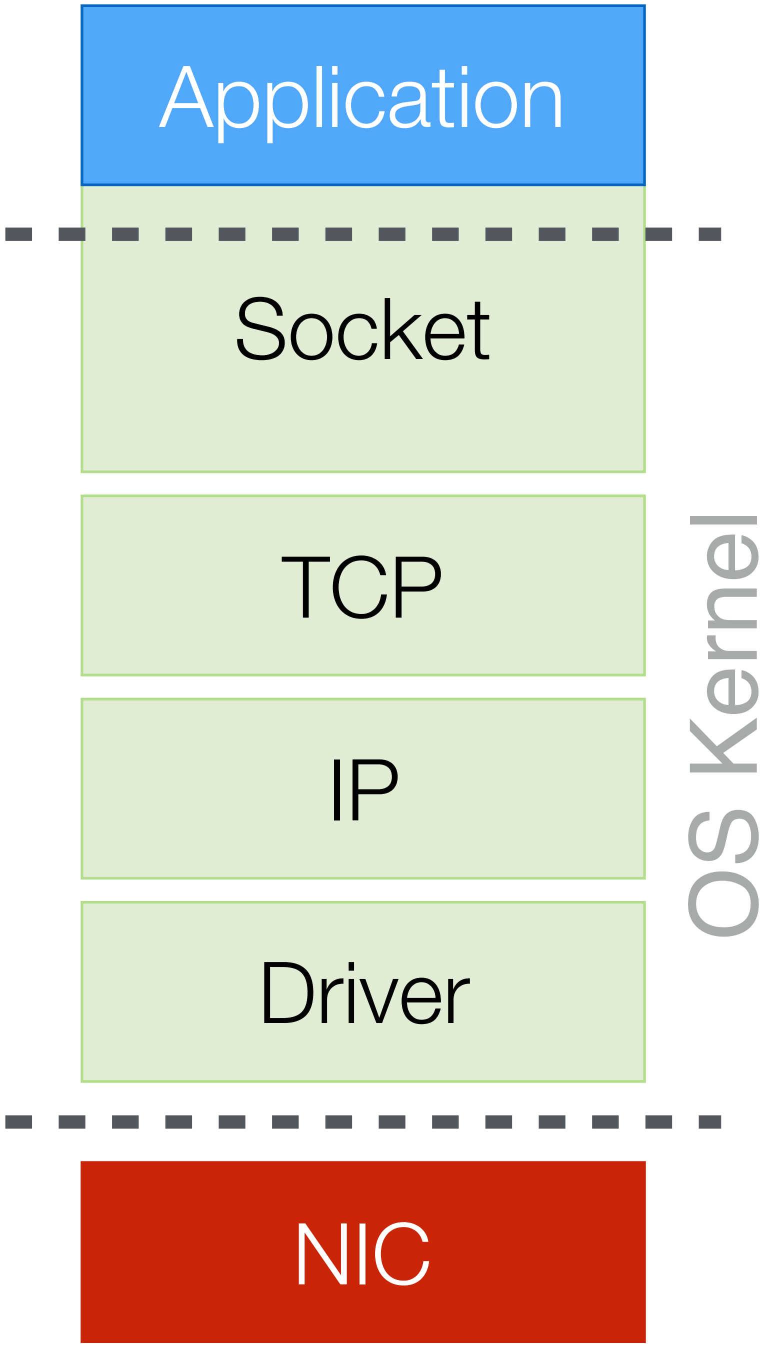
Server is a single thread

Network calls are blocking (**`recv`**, **`accept`**)

- Server can only handle one client at a time

What happens to other clients who try to connect?

- Incoming connections are buffered by the OS networking stack
- TCP Backlog parameter controls number of waiting connections
  - How do you think this works?



Threading!

Threading

Allows program to do multiple things at once

- Threads: execution context with its own stack and shared heap
- Processes: execution context with both stack and heap

How many threads or processes can we run?

Threading

Allows program to do multiple things at once

- Threads: execution context with its own stack and shared heap
- Processes: execution context with both stack and heap

How many threads or processes can we run?

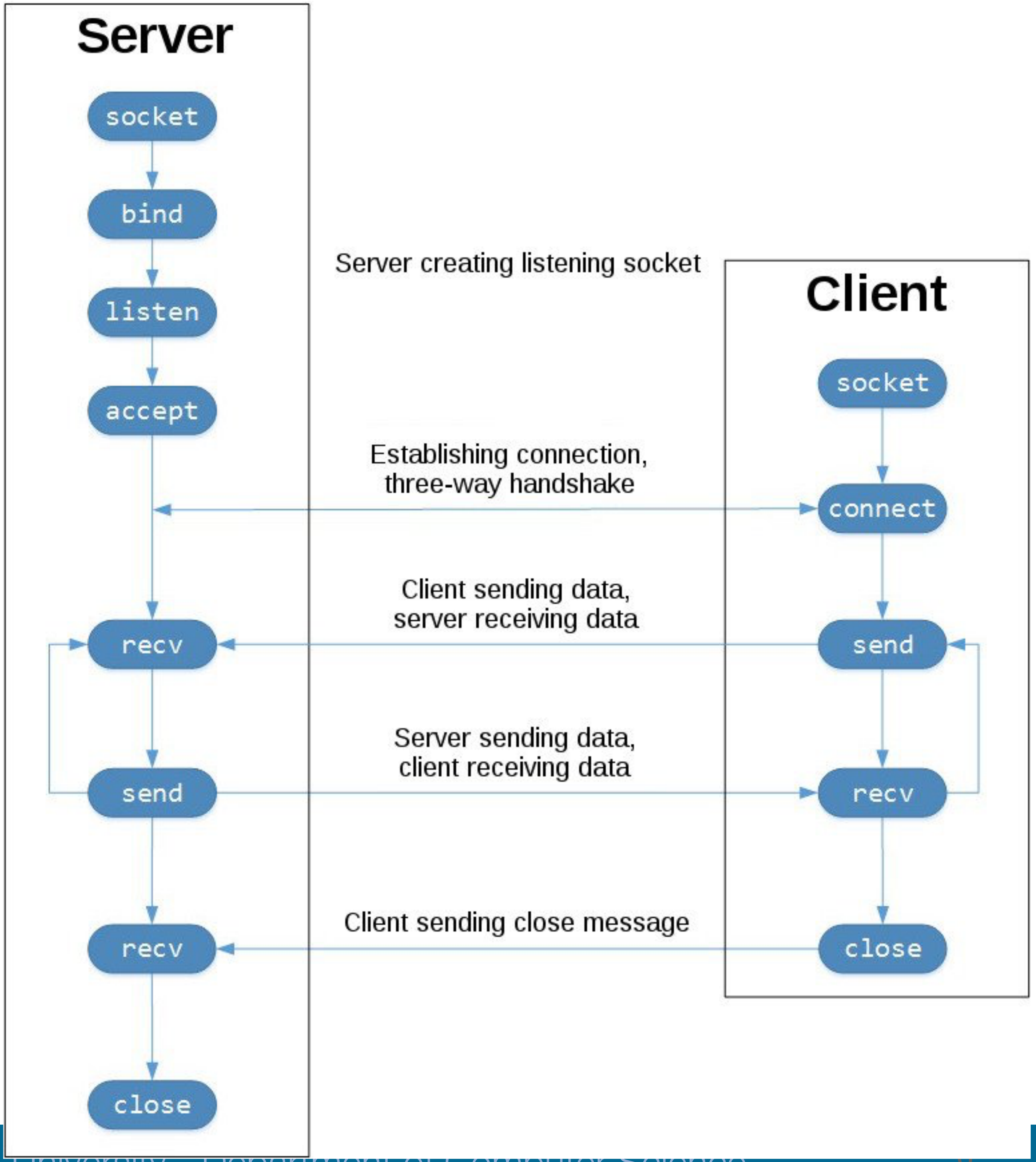
- Depends on available hardware and application type!

Concurrency is limited by…

- Number of CPU cores
- CPU vs IO intensiveness of application
- If CPU bound, then N cores can only run N threads at once
- If I/O bound, then may need >> N threads to keep N cores busy

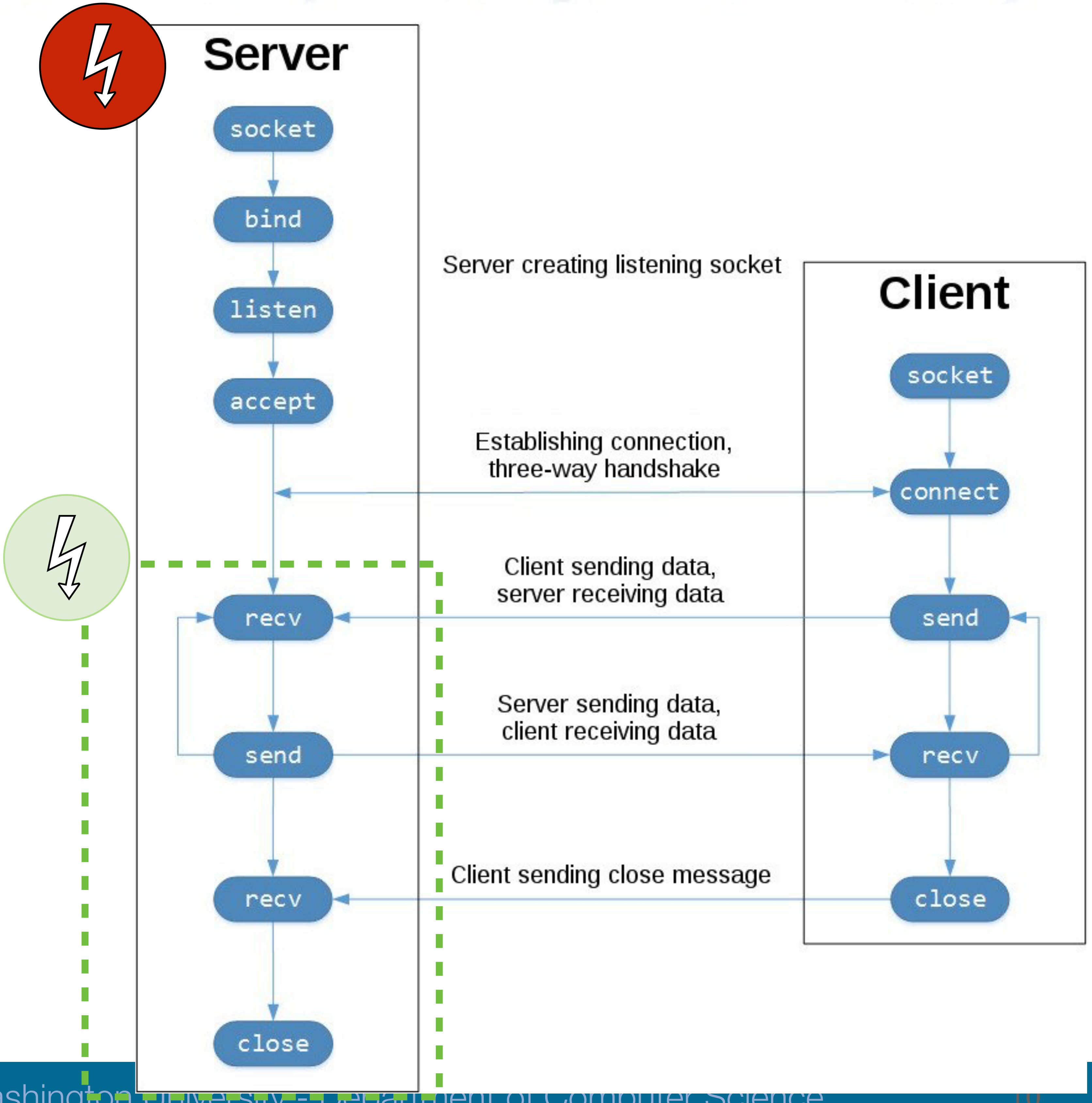
Thread Models

How can we use threads in our Server?


Server
socket
bind
listen
accept
recv
send
recv
close
Server creating listening socket
Establishing connection, three-way handshake
Client sending data, server receiving data
Server sending data, client receiving data
Client sending close message
Client
socket
connect
send
recv
close

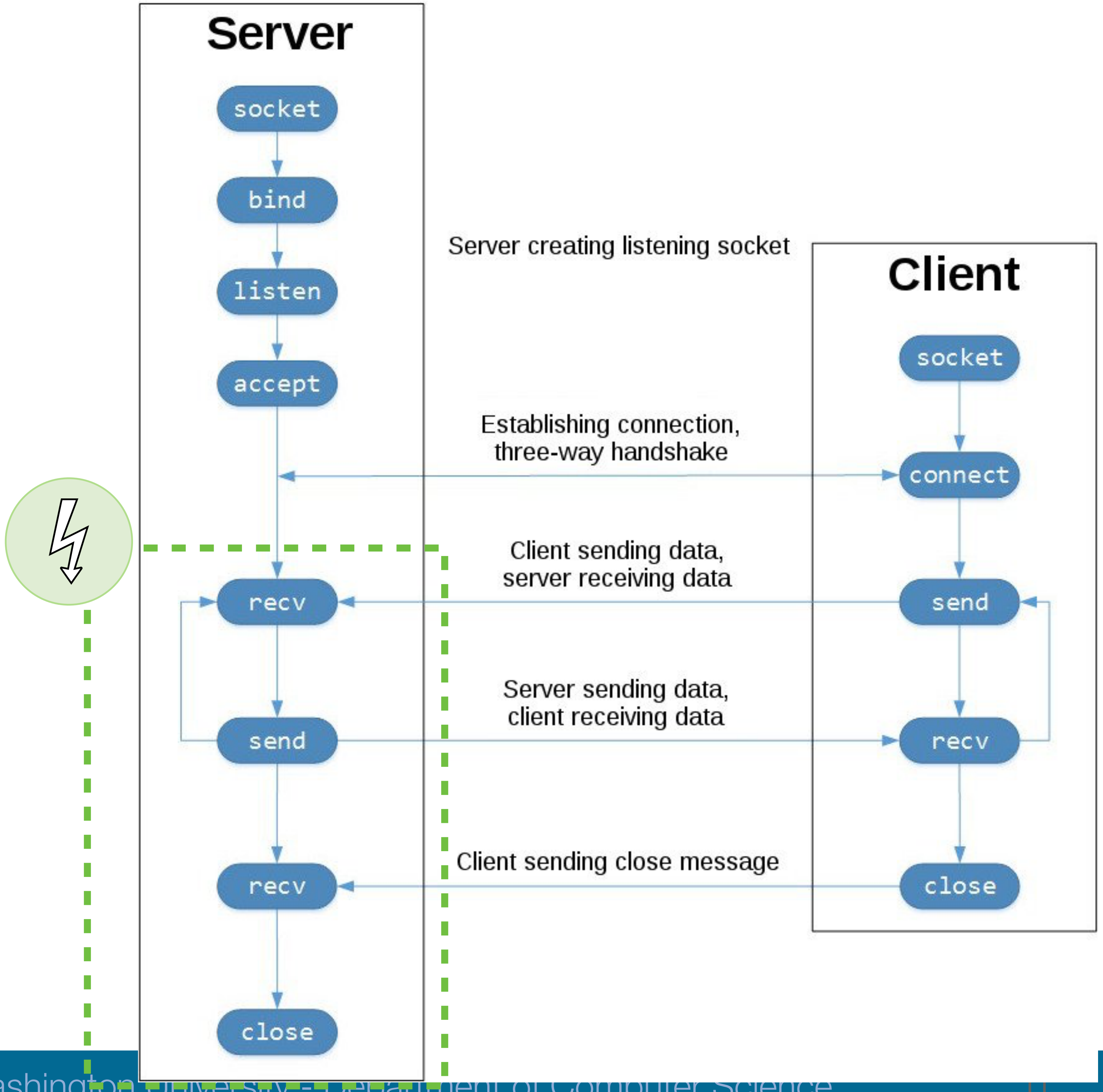
Thread Models

How can we use threads in our Server?



Thread Models

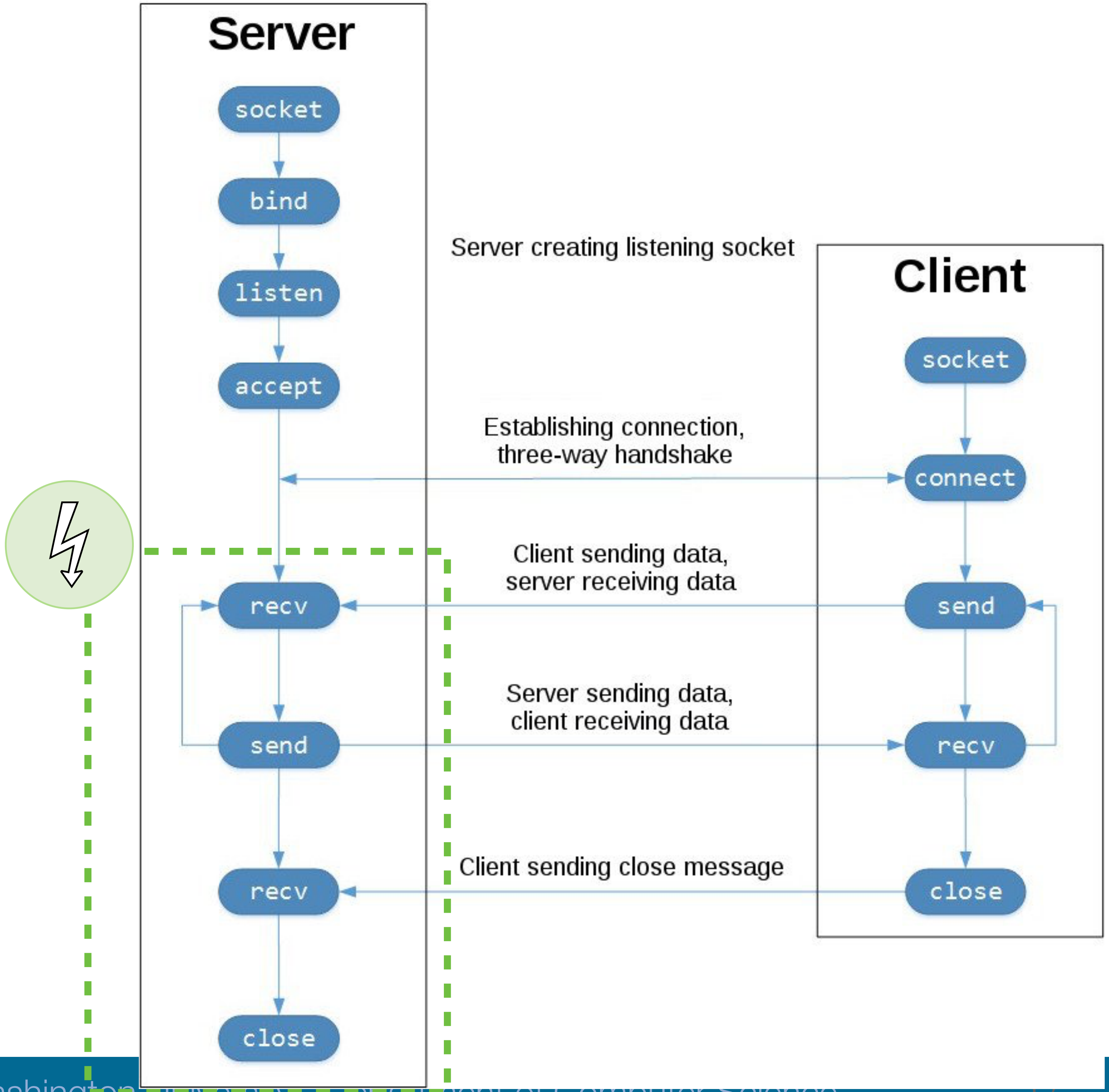
When to start threads?



Thread Models

When to start threads?

1. On every new request create a new thread
2. When program starts create a pool of threads


Server
socket
bind
listen
accept
recv
send
recv
close
Server creating listening socket
Establishing connection, three-way handshake
Client sending data, server receiving data
Server sending data, client receiving data
Client sending close message
Client
socket
connect
send
recv
close

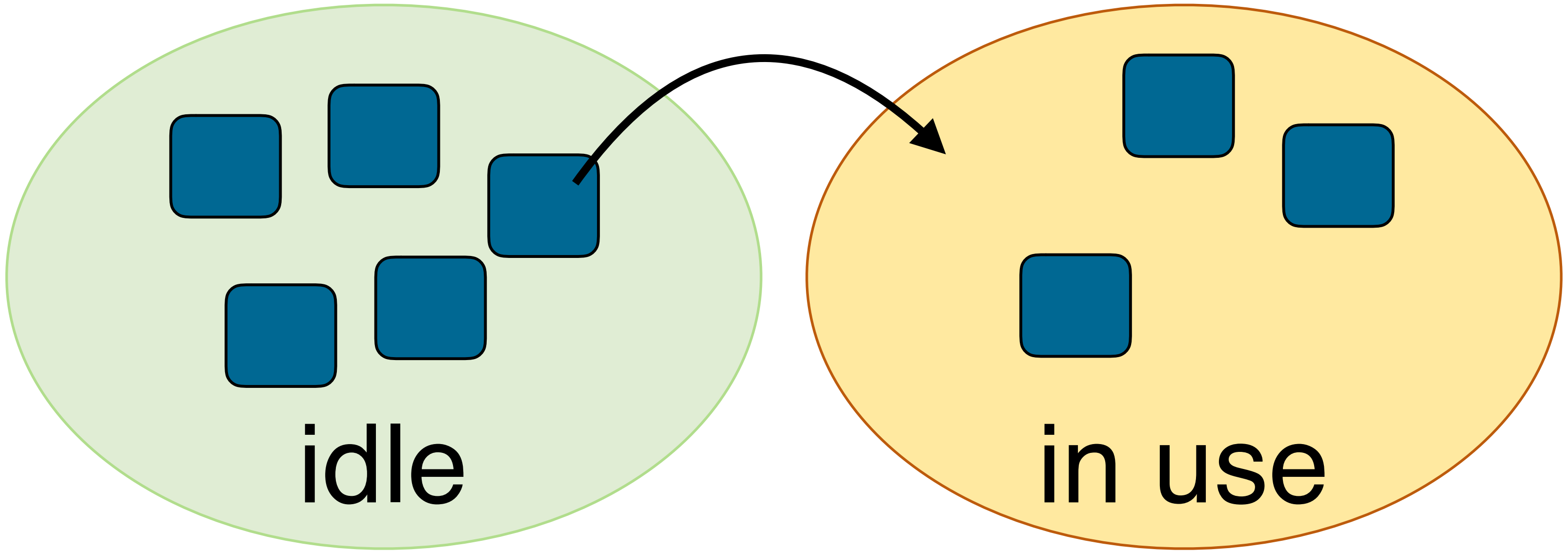
Object Pools

Common design pattern when you need to create and destroy lots of something

Create = malloc

Destroy = free

- Both of these may involve slow system calls
- Even worse if the thing you are creating is a thread!



Object pool just changes an object's state from **idle** to **in use** or back again

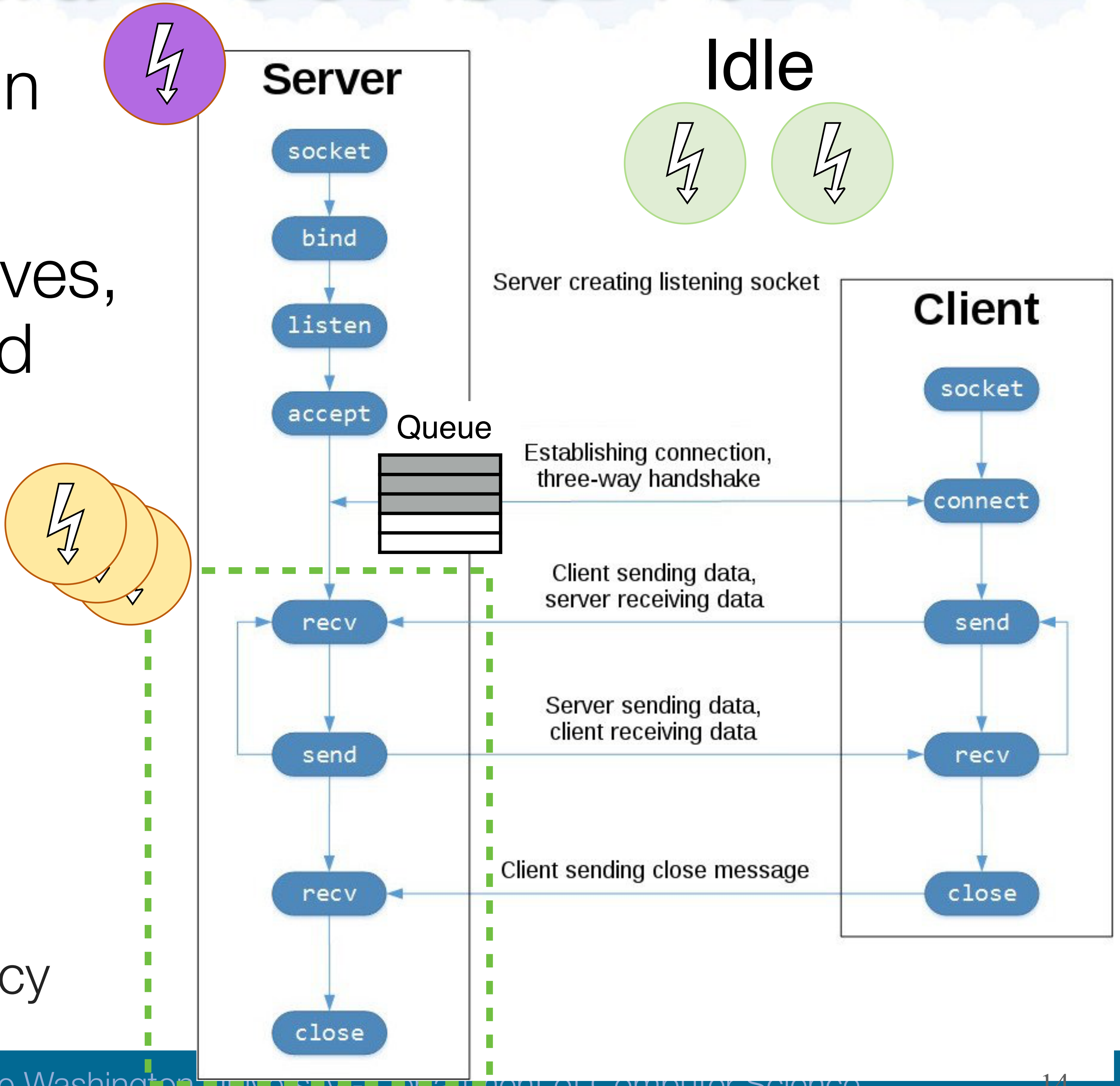
Thread Pool Server

Idle threads wait in pool

When a client arrives, alert an idle thread

How?

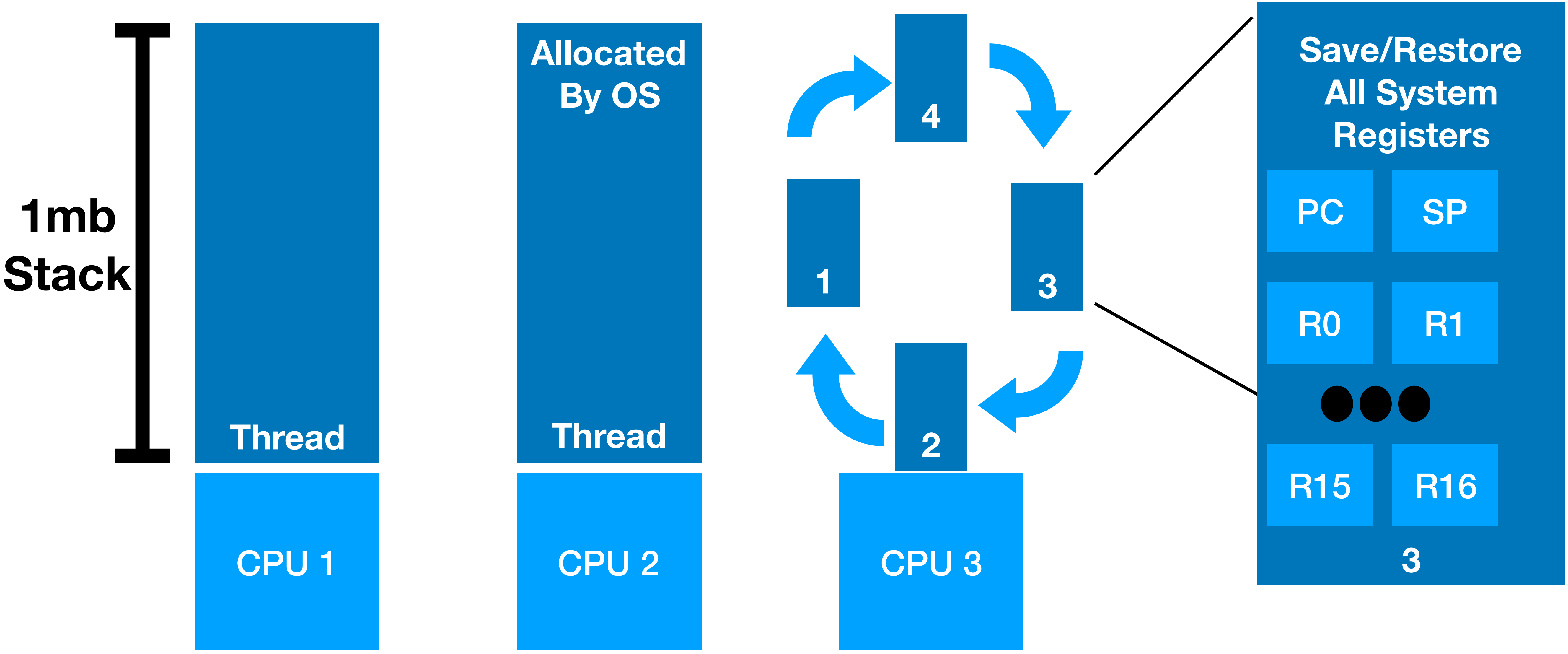
- Put new client into a queue
- Wake idle thread using condition variable
- Remove client from queue using locks for consistency


Idle
Server
socket
bind
listen
accept
Queue
recv
send
recv
close
Server creating listening socket
Establishing connection, three-way handshake
Client sending data, server receiving data
Server sending data, client receiving data
Client sending close message
Client
socket
connect
send
recv
close

Lightweight Threads

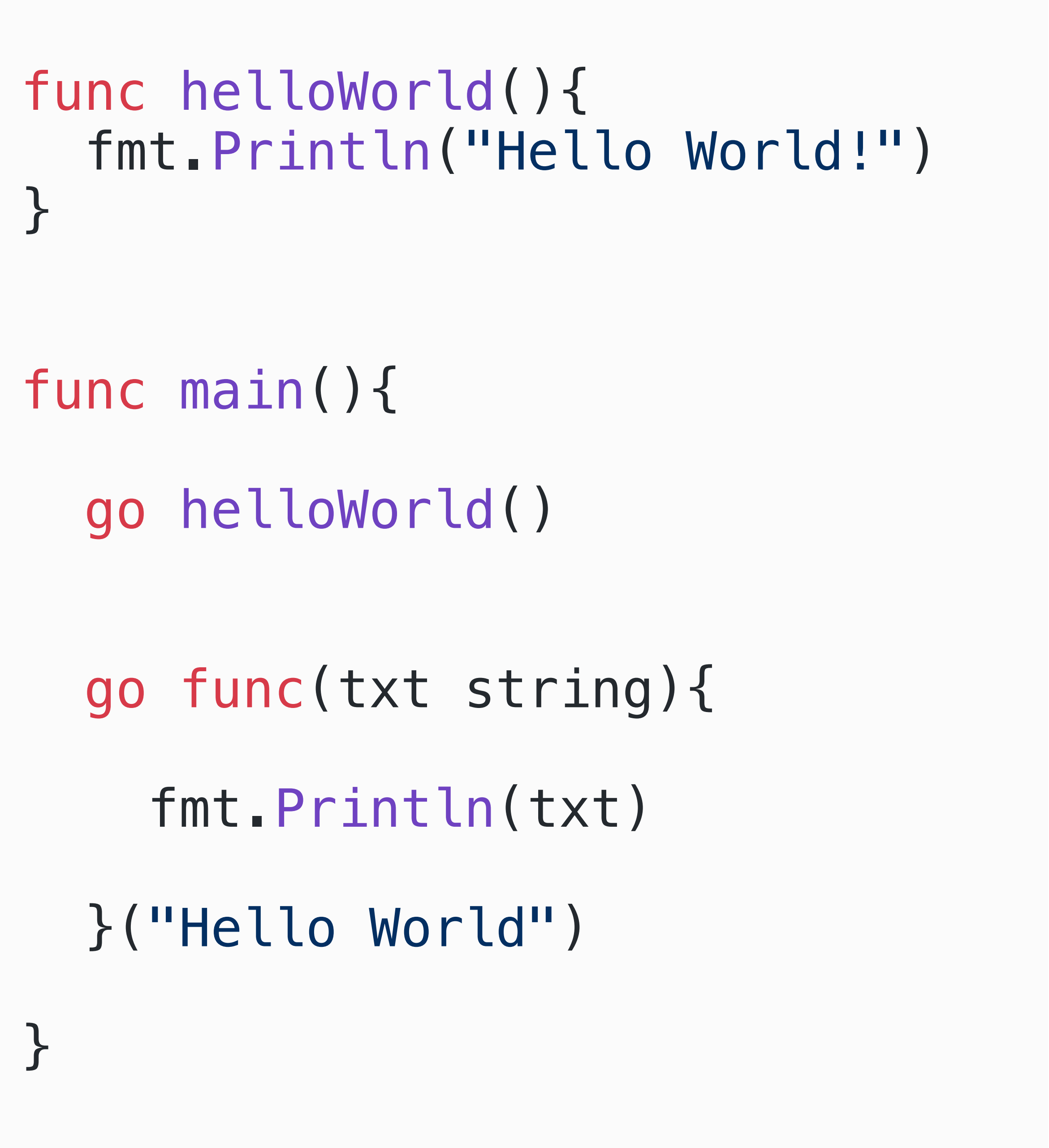
All about Go routines!

A threads primer



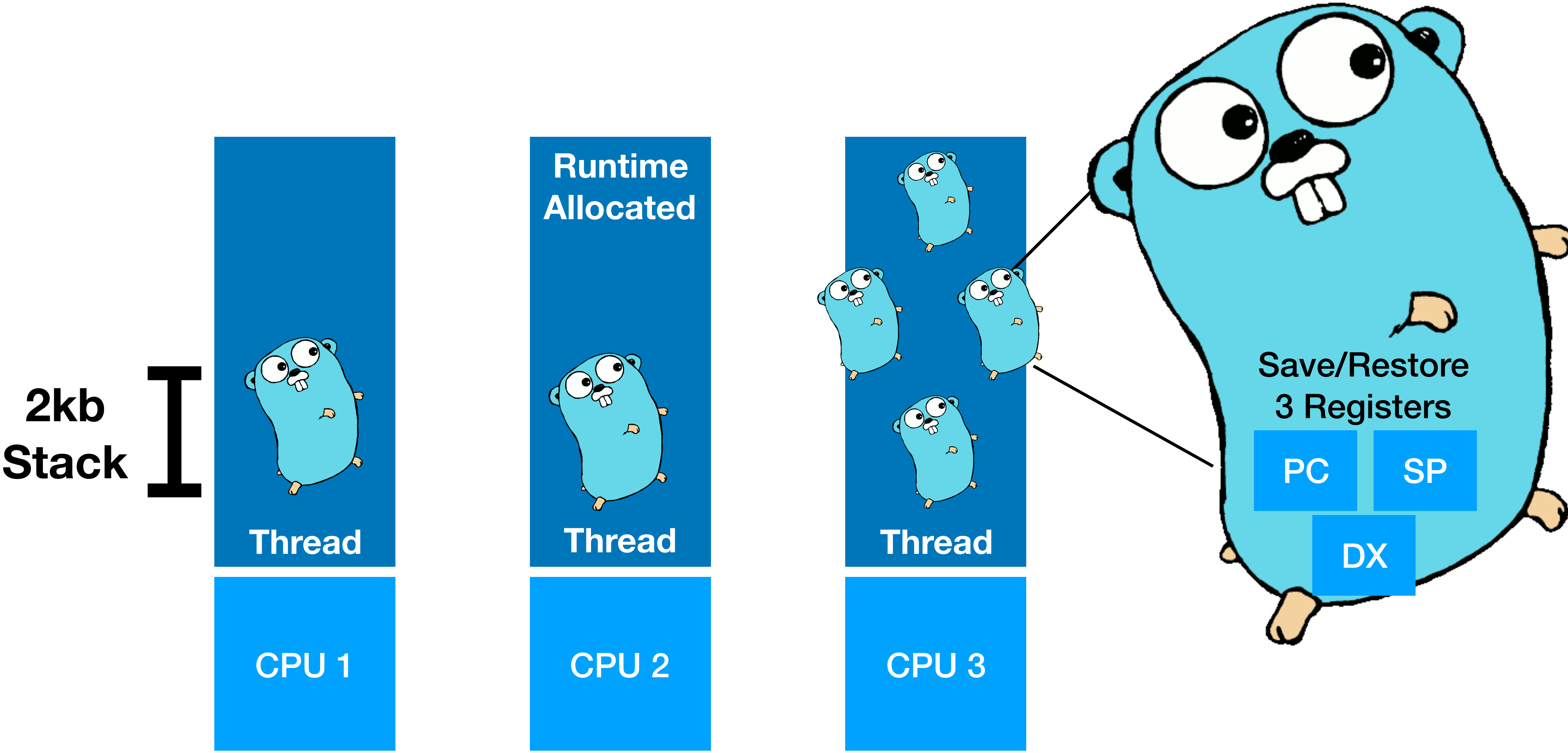
Go Routines

- Golang technique for concurrent programming.
- An abstraction on threading.
- Very lightweight and cheap!
- Allow programs to scale with ease

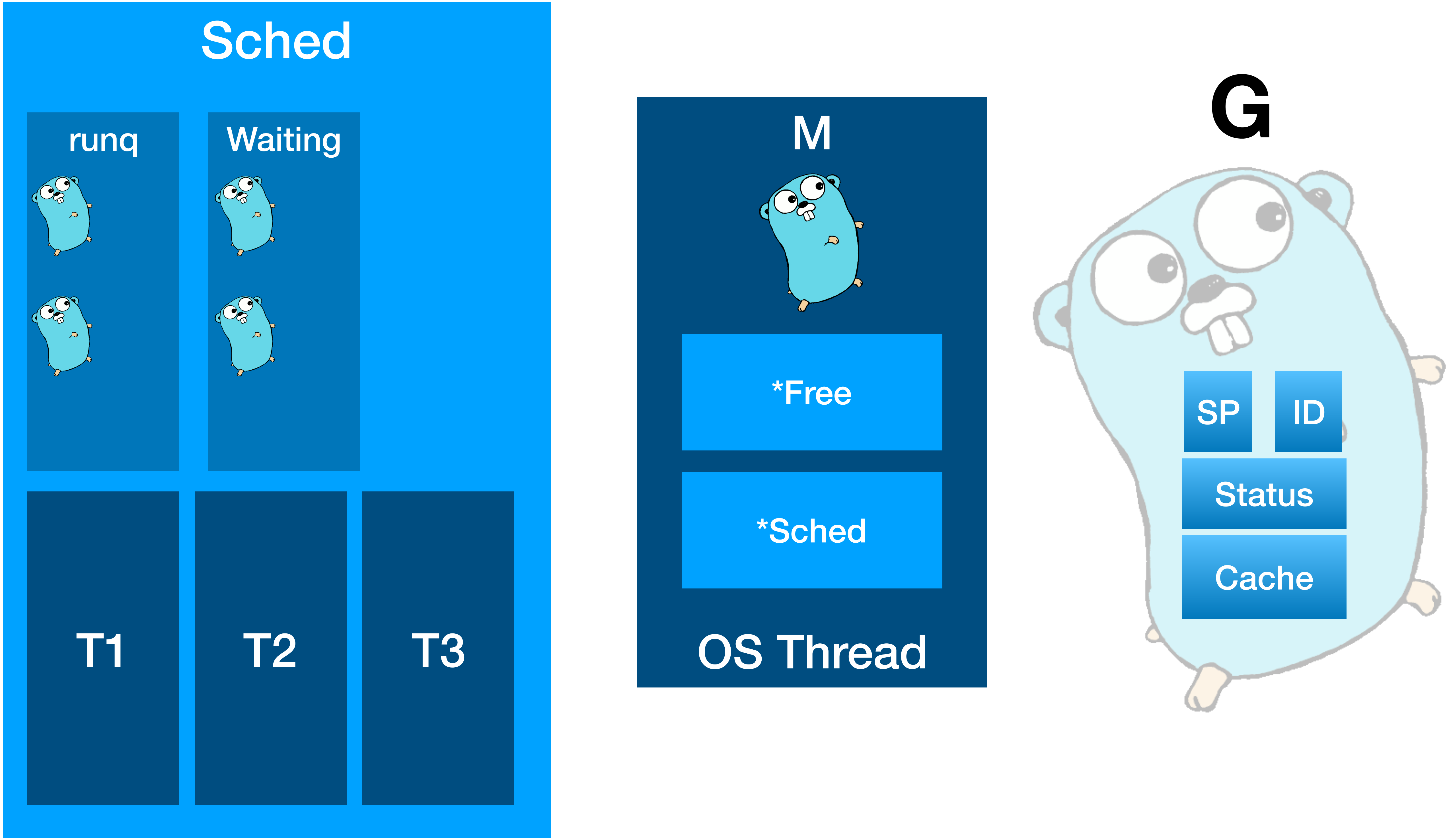


```
func helloWorld(){
  fmt.Println("Hello World!")
}


func main(){

  go helloWorld()


  go func(txt string){

    fmt.Println(txt)

  }("Hello World")

}
```

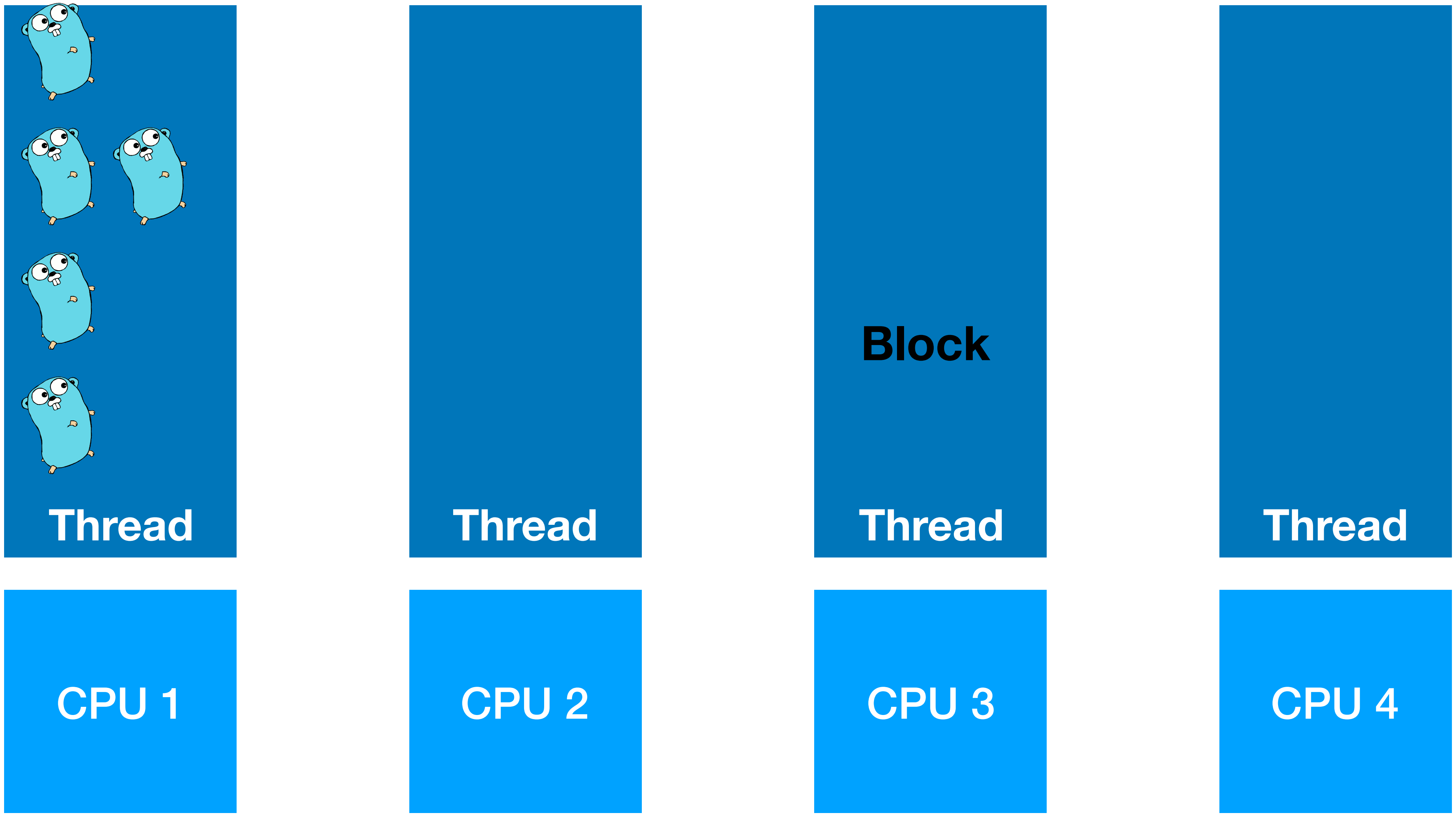
Go Routines


2kb
Stack
Thread
CPU 1
Runtime
Allocated
Thread
CPU 2
Thread
CPU 3
Save/Restore
3 Registers
PC
SP
DX

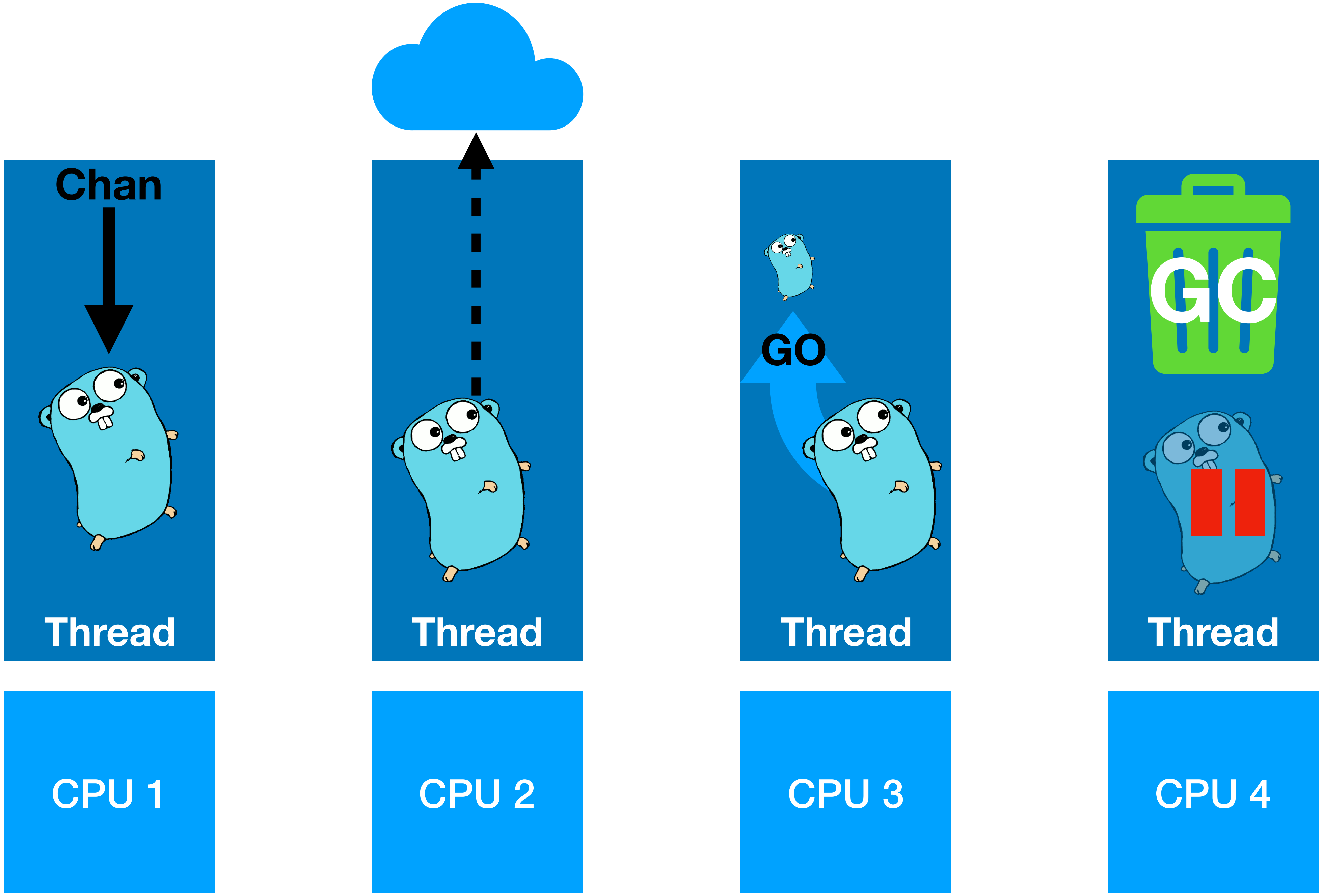
Under the Hood



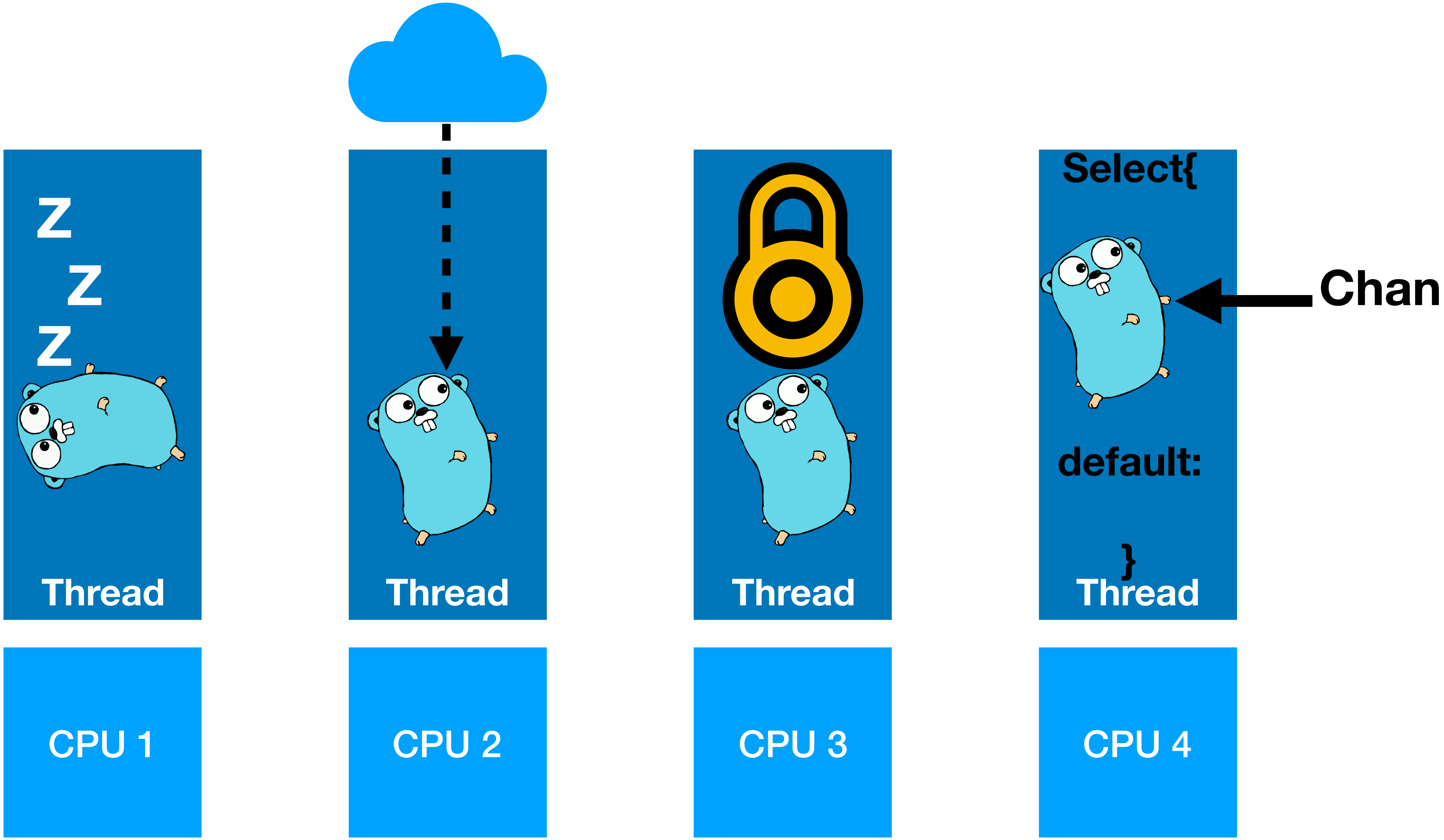
Go Routines



Blocking Go Routines



Non-Blocking Go Routines



More threads?

Is more threads always the answer?

Threads add context switch costs and consume system resources… is there another way?

Non-Blocking IO

Why wait?

Blocking Calls

We needed multiple threads because recv blocks

But is it really necessary to wait on recv?

- You already saw in RUDP project that we don't need to wait forever; we can just wait for a short time and then return

Blocking / Synchronous IO:

- Go to sleep if no data, get woken up when it arrives

Non-Blocking / Asynchronous IO:

- Check if there is data, do something else if no data, check again

Simple Non-Blocking

Sockets can be set to non-blocking mode

```
import socket
# Create a TCP/IP socket in non-blocking mode
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0)
```

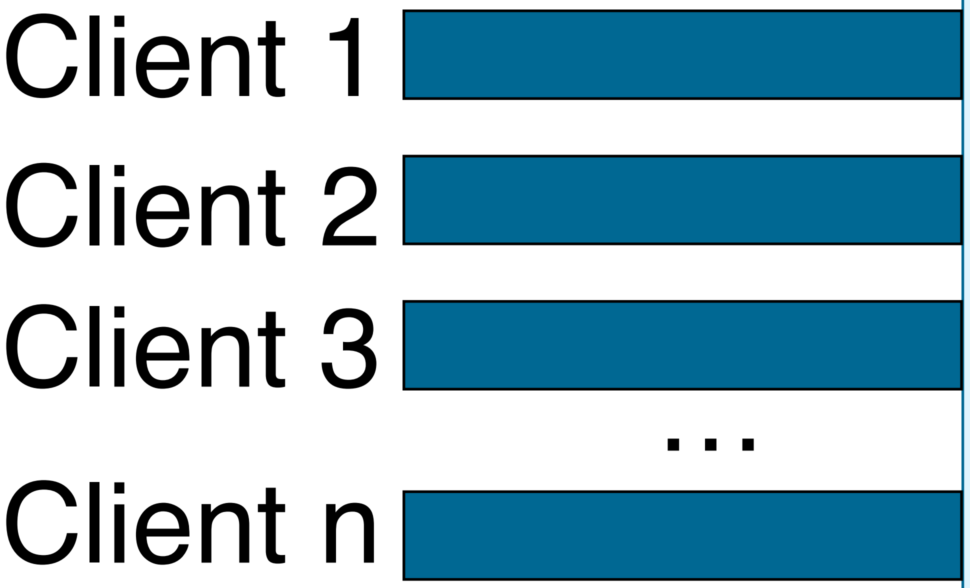
Then **recv** calls will not wait for data, just return error

```
while True:
  try:
    data = conn.recv(1024)
  except socket.error:
    print(“No data yet”)
```

Drawbacks of this approach?

Non-Blocking Server

What happens if we have many clients?



```
# Accept all clients…

for client in clients:
  try:
    data = client.recv(1024)
    process(data)
  except socket.error:
    print(“No data yet”)
```

Code is messy and inefficient if many clients!

Non-Blocking IO

We need a better way to know what data is ready!

`select` event polling

- Register a set of IO “file descriptors” you care about
- Sleeps until at least one of them has data -> won’t block!

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
             fd_set *errorfds, struct timeval *timeout);
```

- Assumes a Unix environment where files, sockets, and other types of IO are all mapped to a file interface

Select Example

```
import selectors
import socket
```

```
def accept(sock, mask):
  conn, addr = sock.accept()
  conn.setblocking(False)
  sel.register(conn,
     selectors.EVENT_READ, read)
```

```
def read(conn, mask):
  data = conn.recv(1000)
  if data:
    conn.send(data)
  else:
    sel.unregister(conn)
    conn.close()
```

```
sel = selectors.DefaultSelector()
sock = socket.socket()
sock.bind(('localhost', 1234))
sock.listen(100)
sock.setblocking(False)
sel.register(sock, selectors.EVENT_READ, accept)

while True:
    events = sel.select()
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

Non-blocking Variants

Languages, runtimes, and OS's typically have several ways to do non-blocking IO

select: system call for checking if things are ready

epoll / **kqueue**: app/OS interface for checking if things are ready (much more efficient than original select)

But now select can be viewed as an API, and might be implemented with something like epoll.

Event-Based Programming

Registering call backs for events can be a simpler programming model

- Simpler to write… maybe harder to debug!

Adds a layer of abstraction

- Event notification layer checks for events and decides what order to process them in. Why is this helpful/interesting?
- Could use multiple threads to process the events!

node.js

Web framework for javascript-based apps

Probably the most popular event based platform

Single threaded event based server!

- Faster and less resource intensive than many multi-threaded servers!

Other event based frameworks/languages:

- Erlang, Elixir, …

Assignment 2

Technical Writing

Being able to present ideas is just as important as being able to write code!

[ ] Write a blog post on a networking topic

- Must be long enough to be interesting
- You must write some code or run experiments
- Present useful information in an understandable way
- Present useful information in a visually appealing way

Ideas

Performance comparison of…

- Node.js vs Apache vs nginx vs …
- Thread pool vs new thread per request in language X
- http vs https vs http2

Tutorial on…

- how to use wireshark to analyze HTTP traces or solve a puzzle
- how to gather statistics of public wifi traffic (ethically)
- how to use go co-routines and how they work under the hood
- queueing theory 101 with example measurements
- how to use epoll / select / etc in language X
- everything that happens when you open a page in a browser
- python 2 vs python 3 networking code
- how to generate traffic to benchmark a web server

Inspiration

Julia Evans' blog and zines

- https://jvns.ca/