# CS 43: Computer Networks

## 04: Socket Programming

September 12, 2024

SWARTHMORE COLLEGE

Slides adapted from Kurose & Ross, Kevin Webb

# Reading Quiz

# Announcements

- TA for the course: Marcus Wright
  - Office Hours: 2 – 4pm in Overflow.
- Regarding missed classes/labs
  - three free misses on classes
  - lab attendance is mandatory

# Midterm Scheduling: Monday Oct 21st 7 – 8.30 PM

Can you make this time?

A. Yes
B. No

# Client-Server communication

- Client:
  - initiates communication
  - must know the address and port of the server
  - active socket
- Server:
  - passively waits for and responds to clients
  - passive socket

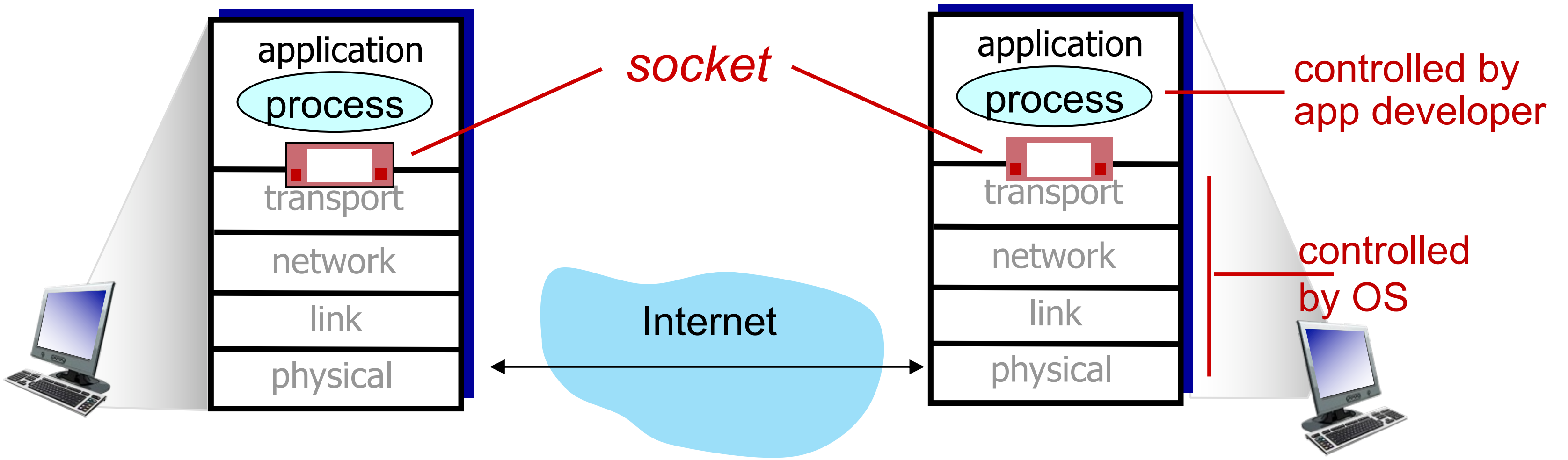
# What is a socket?

An abstraction through which an application may send and receive data,

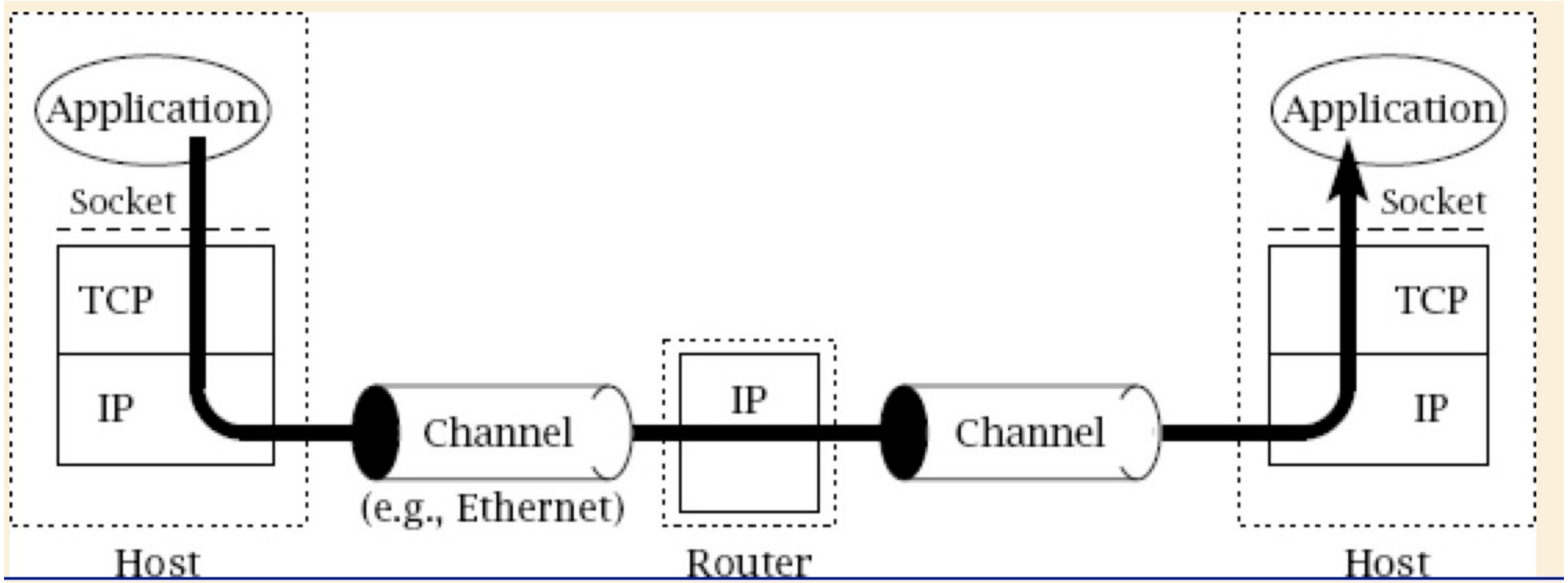
in the same way as a open-file handle or file pointer allows an application to read and write data to storage.

# Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
  - two sockets involved: one on each side

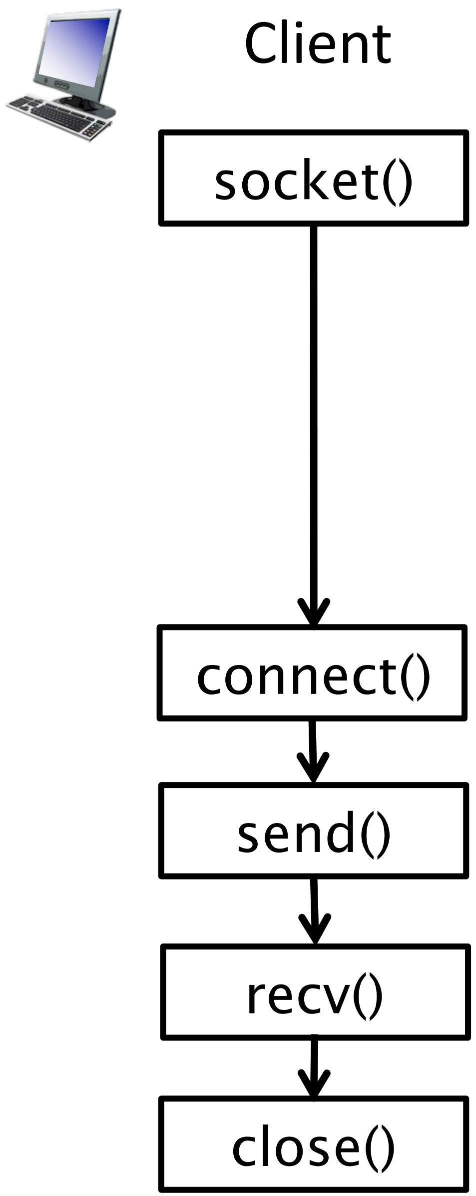


# Socket Programming




Adapted from: Donahoo, Michael J., and Kenneth L. Calvert. TCP/IP sockets in C: practical guide for programmers. Morgan Kaufmann, 2009.

# TCP Socket Procedures: Client




create a new communication endpoint

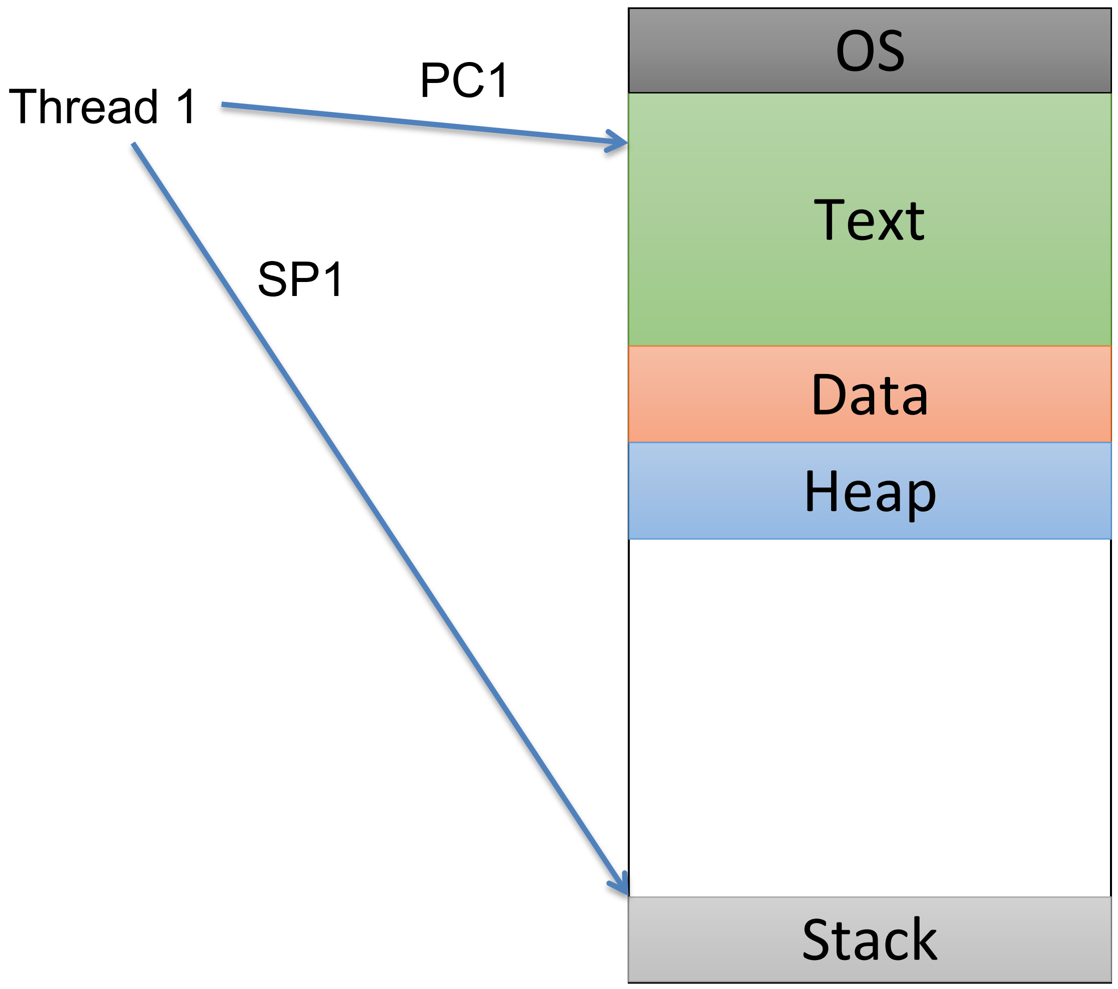
actively attempt to establish a connection

send some data over a connection

receive some data over a connection

release the connection

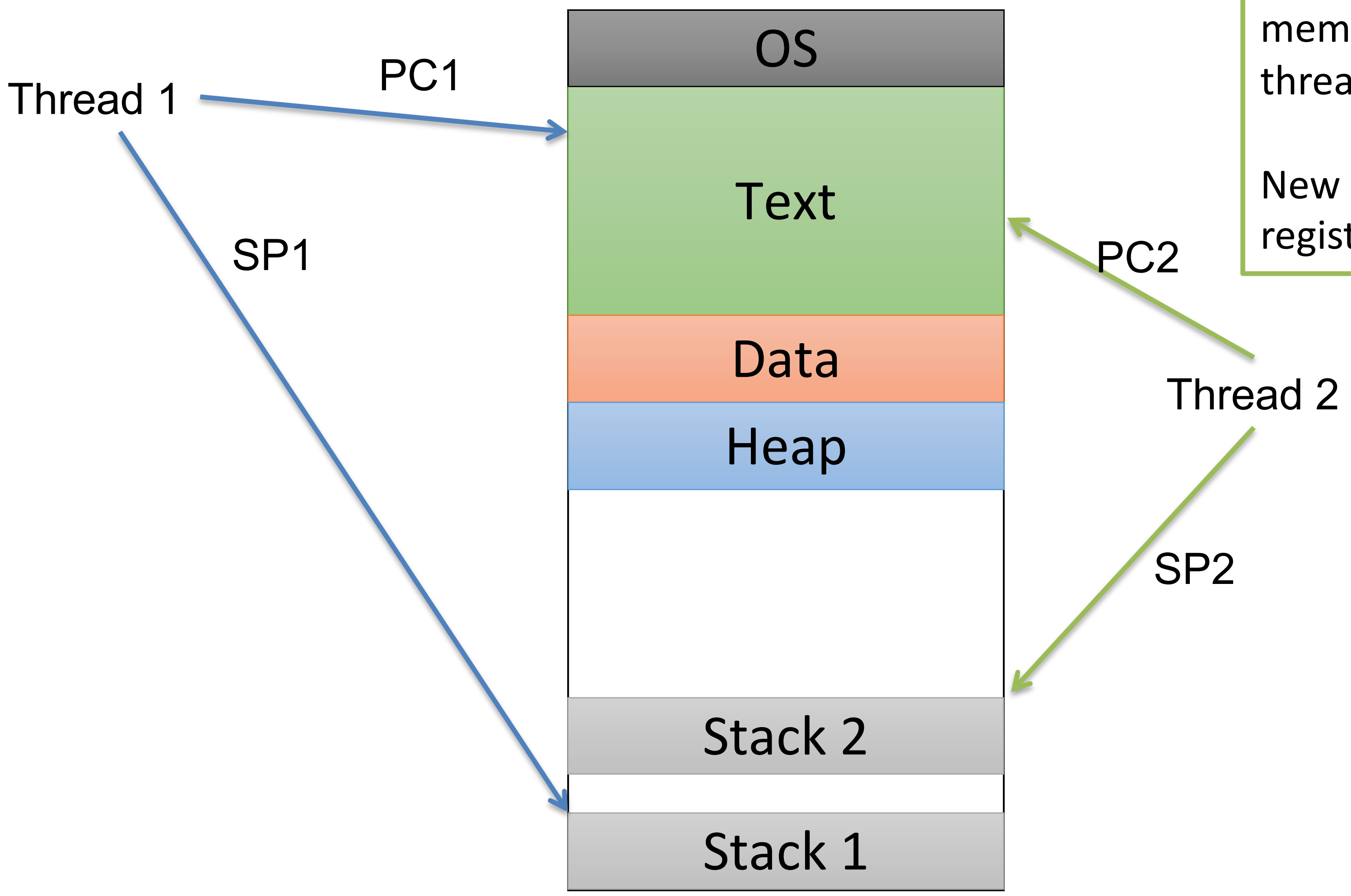
# Threads




This is the picture we've been using all along:

A process with a single thread, which has execution state (registers) and a stack.

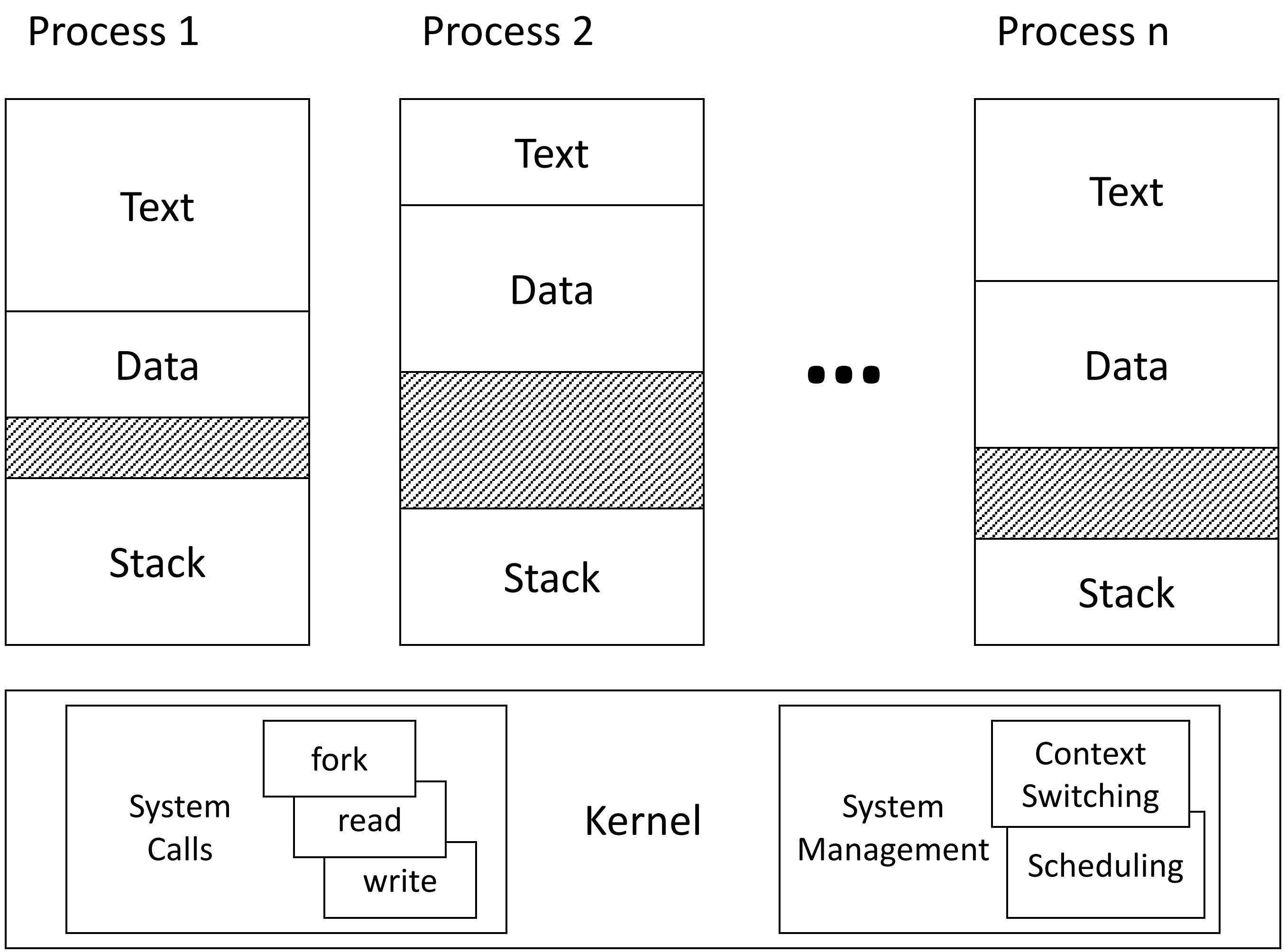
# Threads




We can add a thread to the process. New threads share all memory (VAS) with other threads.

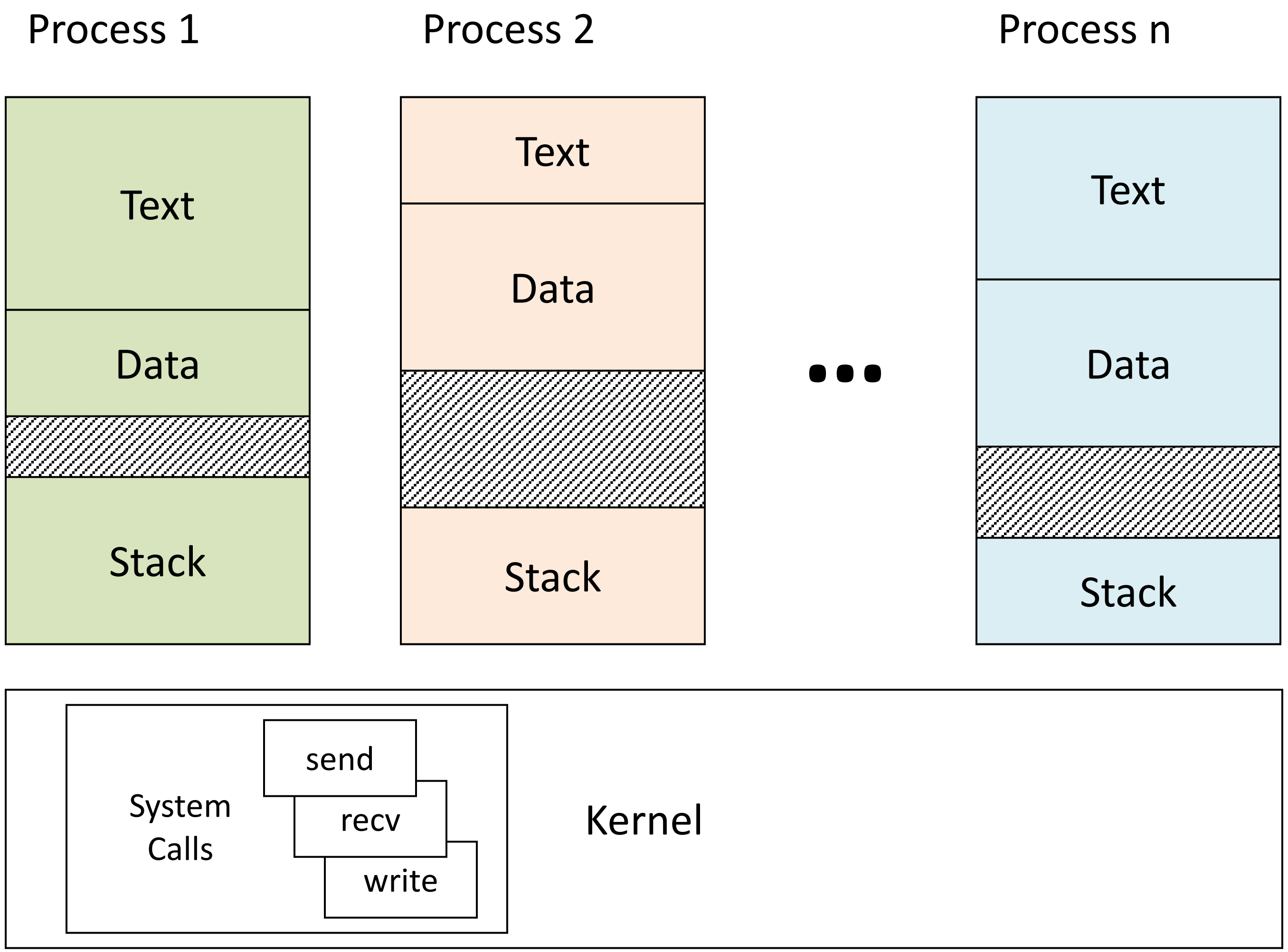
New thread gets private registers, local stack.

# Recall: Processes

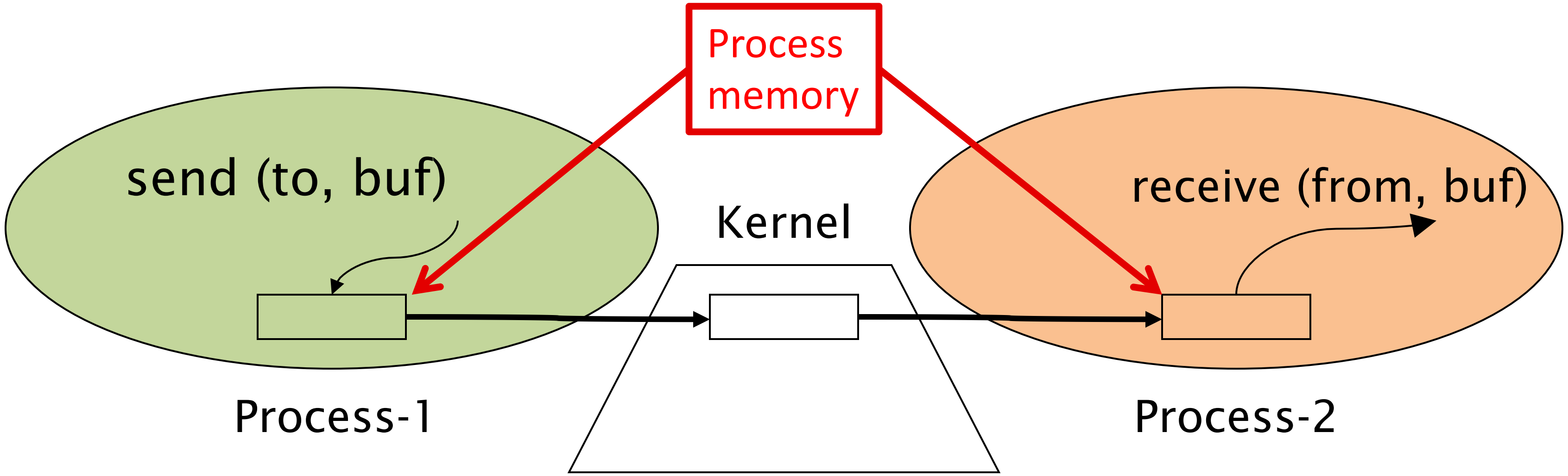


# Inter-process Communication (IPC)

- Processes must communicate to cooperate
- Must have two mechanisms:
  - Data transfer
  - Synchronization
- On a single machine:
  - ~~Threads (shared memory)~~
  - Message passing



# Interprocess Communication (local model)




- Operating system mechanism for inter-process communication
  - send (destination, message_buffer)
  - receive (source, message_buffer)
- Data transfer: in to and out of kernel message buffers
- Synchronization

# Interprocess Communication (non-local)

- Processes must communicate to cooperate

- Must have two mechanisms:
  - Data transfer
  - Synchronization

- Across a network:
  - Message passing

# Message Passing (network)

- Same synchronization
- Data transfer
  - Copy to/from OS socket buffer
  - Extra step across network: hidden from applications

# Descriptor Table

For each Process

OS stores a table, per process, of descriptors

Kernel

# Descriptors

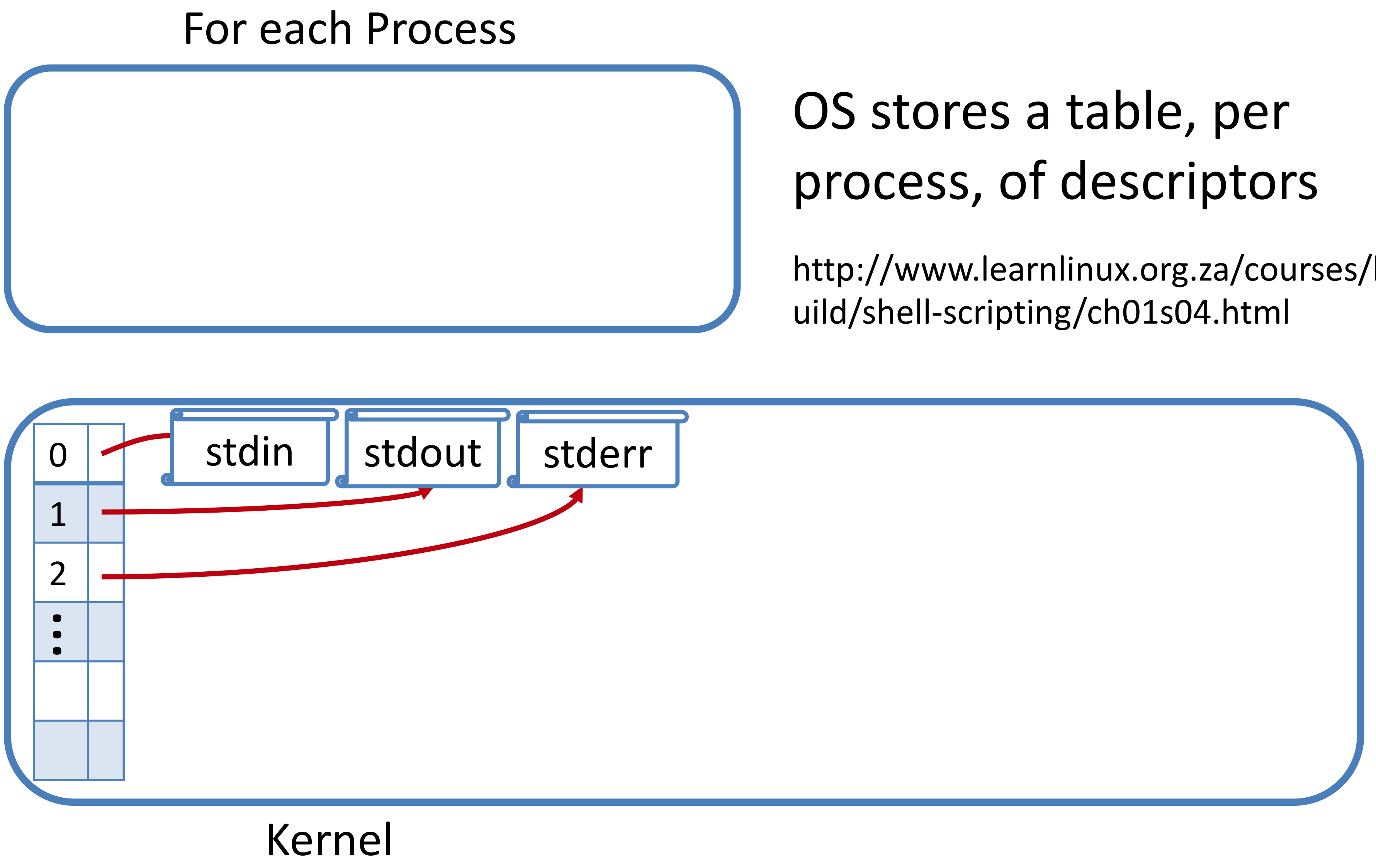
```
SOCKET(2)                    BSD System Calls Manual                    SOCKET(2)

NAME
     socket -- create an endpoint for communication

SYNOPSIS
     #include <sys/socket.h>

     int
     socket(int domain, int type, int protocol);

DESCRIPTION
     socket() creates an endpoint for communication and returns a descriptor.
```

**DESCRIPTION** top

```
       The open() system call opens the file specified by pathname.  If the
       specified file does not exist, it may optionally (if O_CREAT is
       specified in flags) be created by open().
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

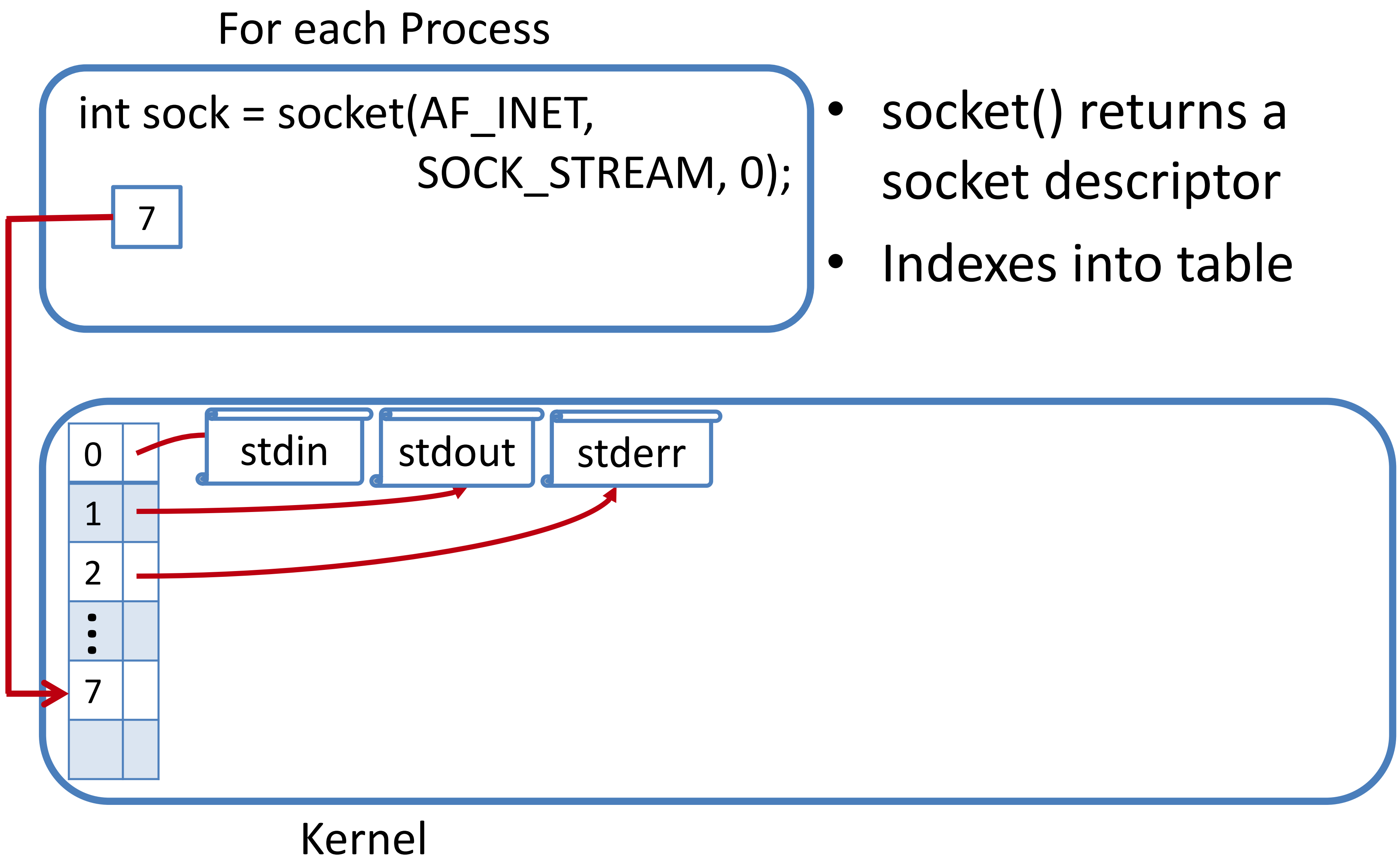
# Descriptor Table




OS stores a table, per process, of descriptors

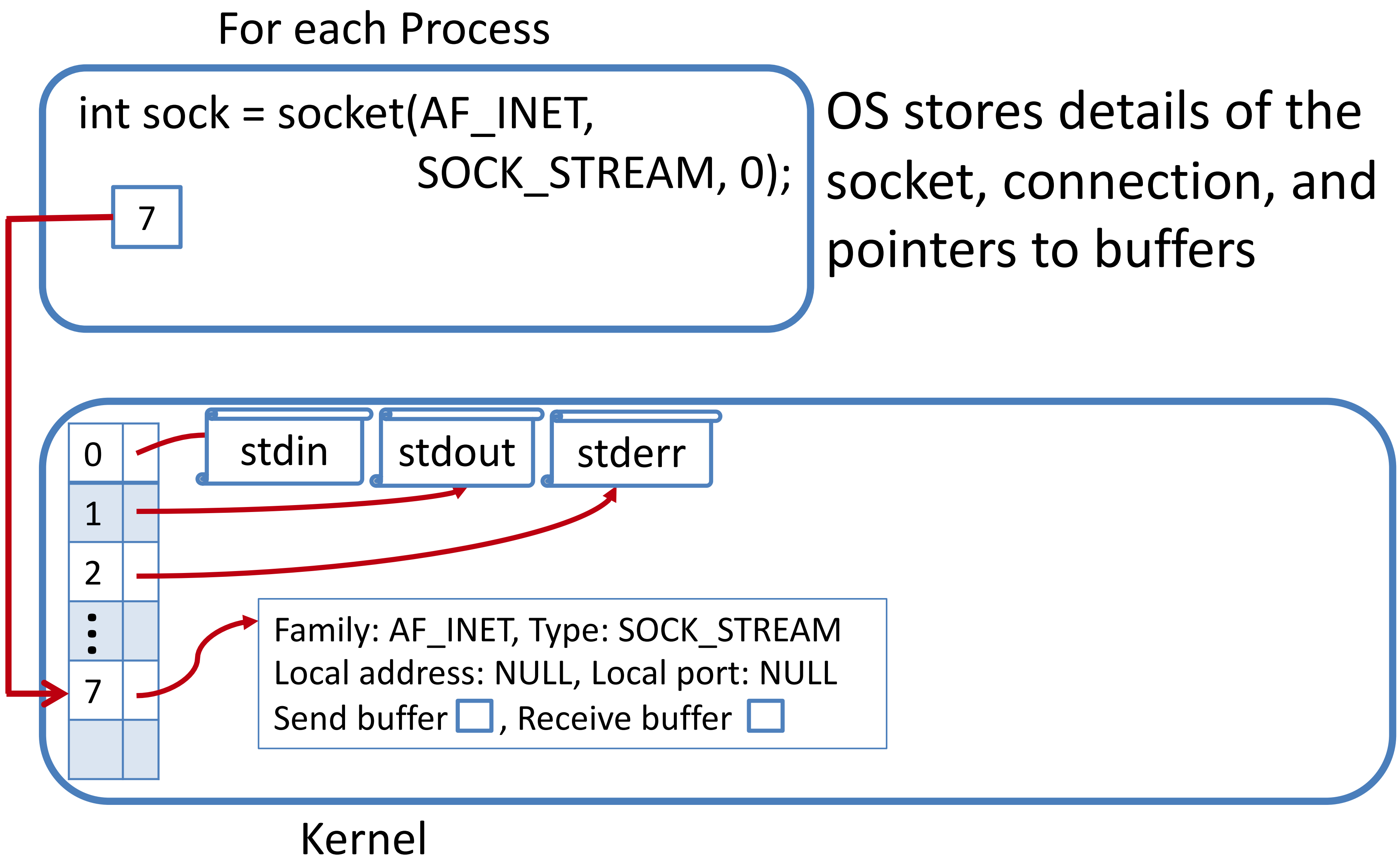
http://www.learnlinux.org.za/courses/build/shell-scripting/ch01s04.html

# socket()



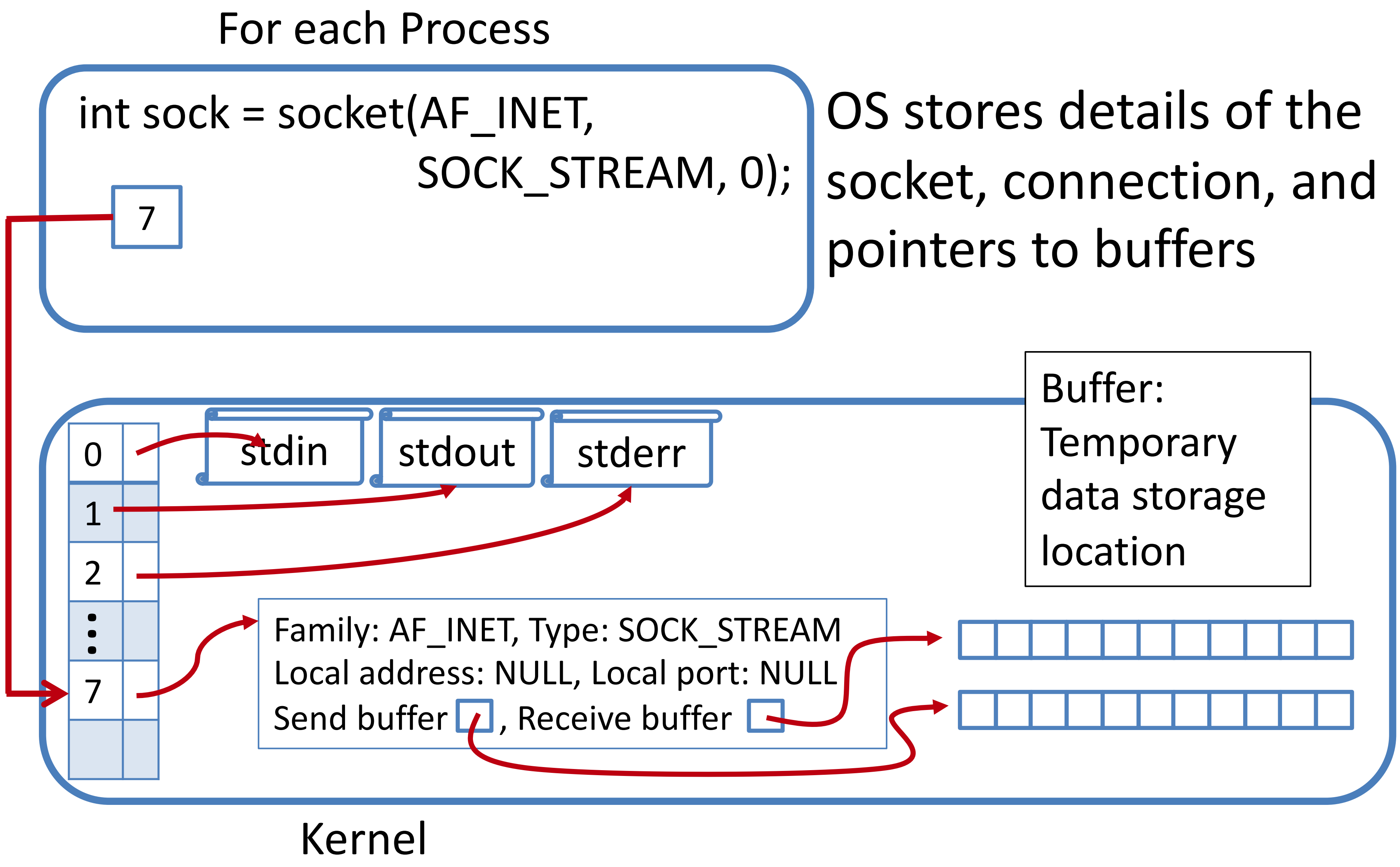

- socket() returns a socket descriptor
- Indexes into table

# socket()



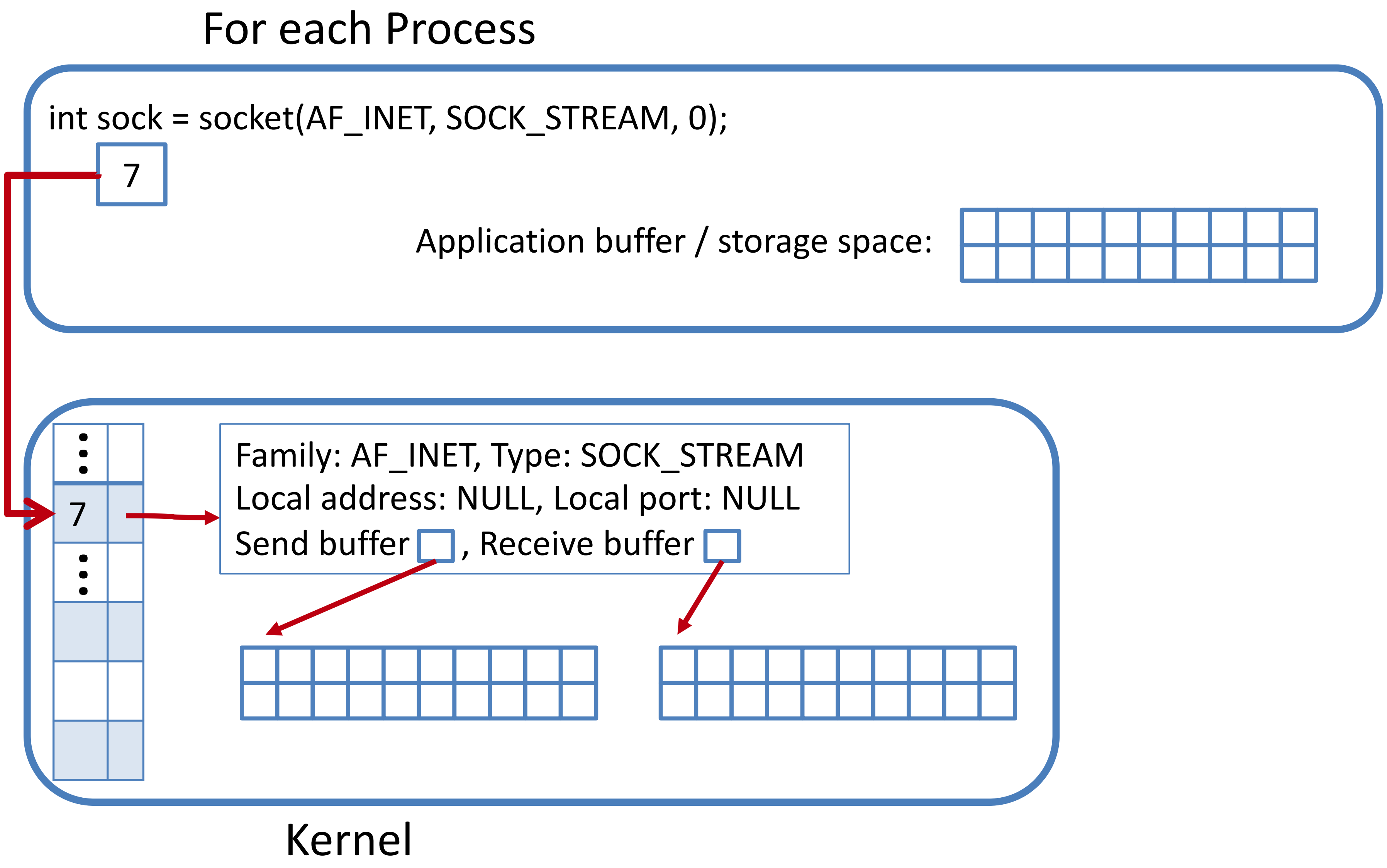

OS stores details of the socket, connection, and pointers to buffers

# socket()

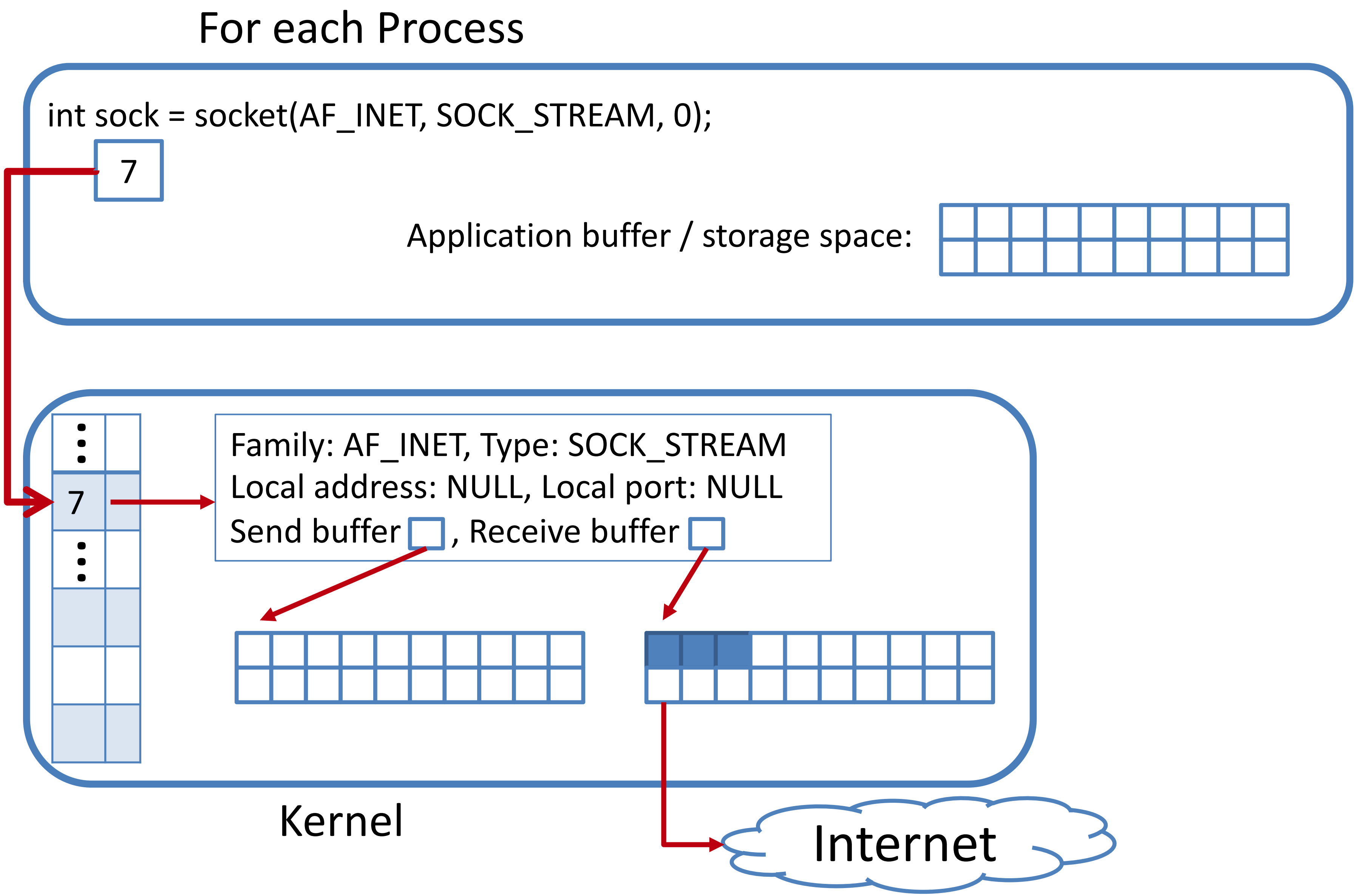



OS stores details of the socket, connection, and pointers to buffers

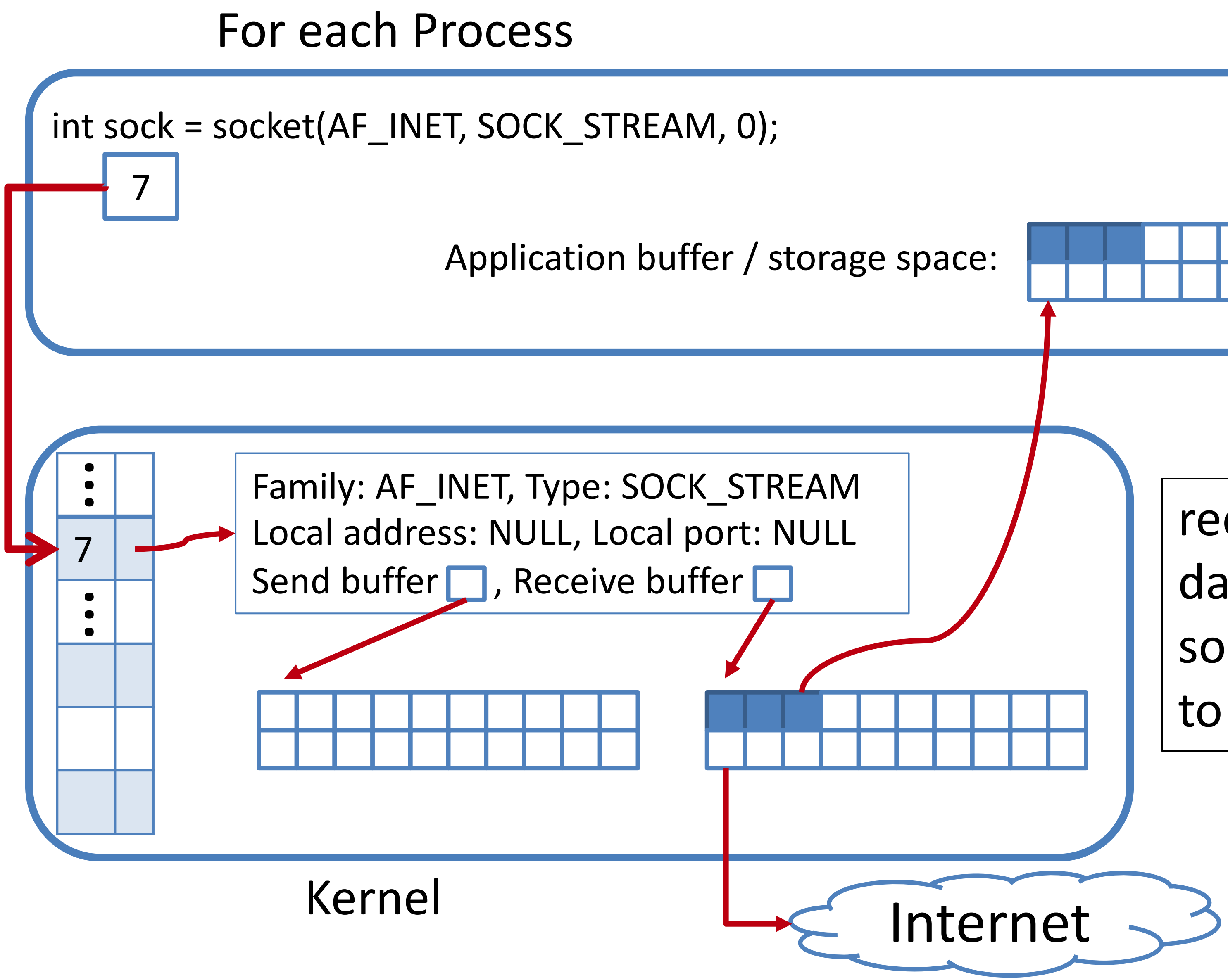
# Socket Buffers



# Socket Buffers

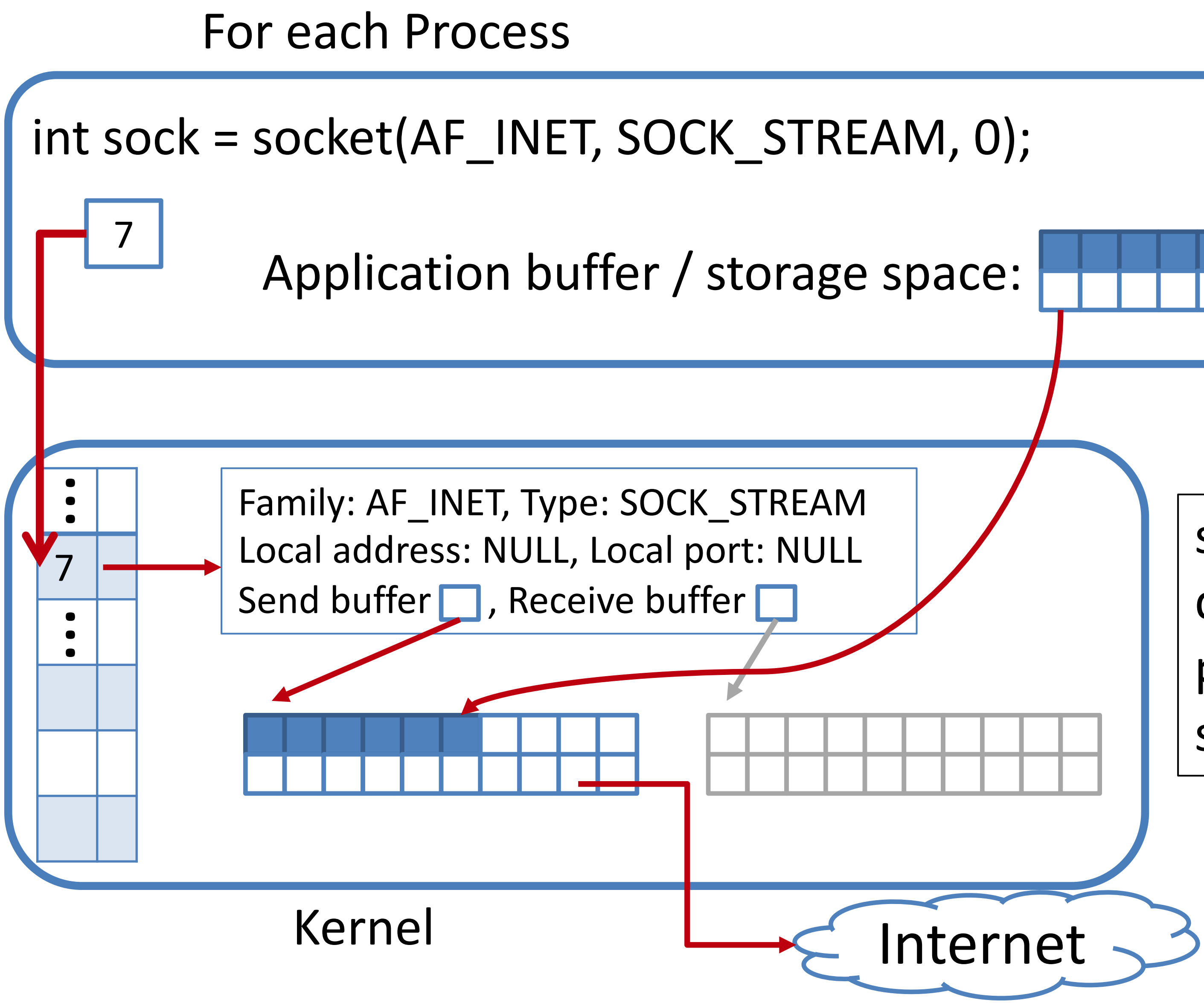


# Socket Buffers



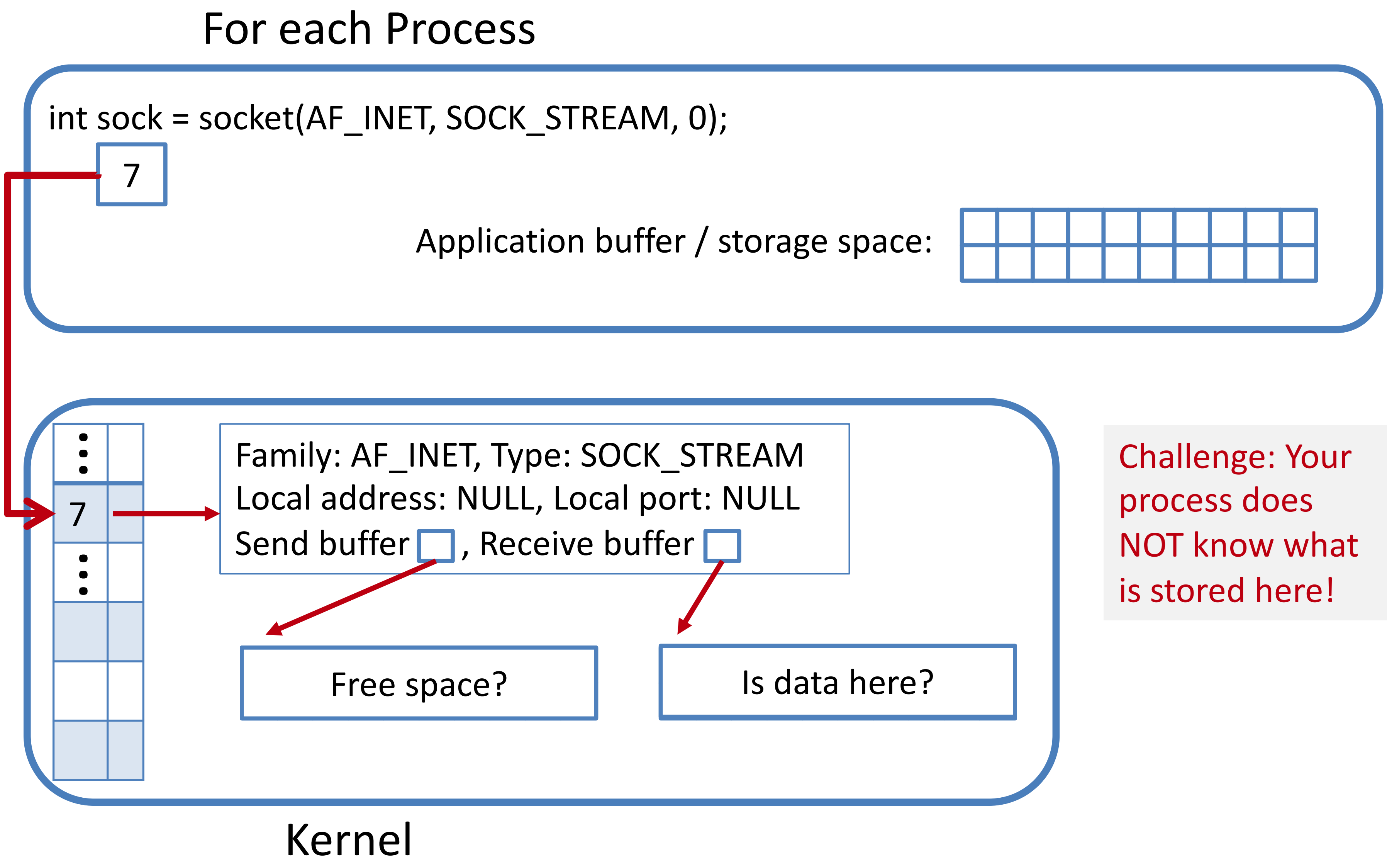

recv(): Move data from socket buffer to process

# Socket Buffers




send(): Move data from process to socket buffer

# Socket Buffers



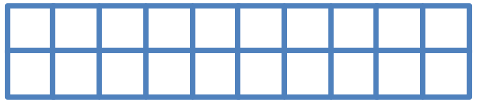

Challenge: Your process does NOT know what is stored here!

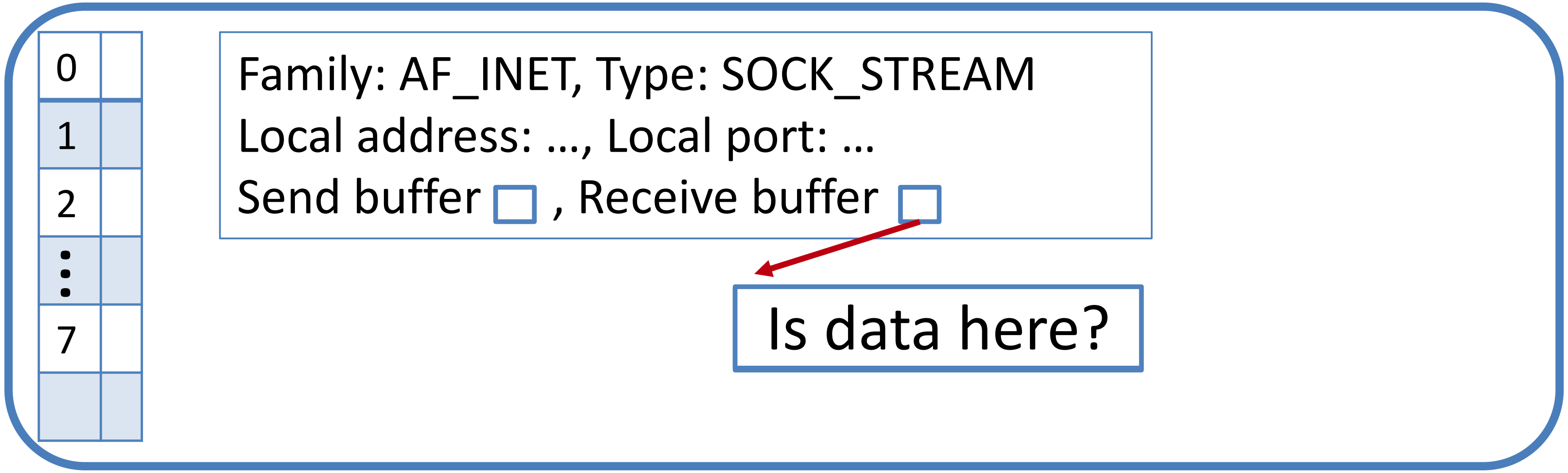
# recv()

For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we issued a connect() here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)





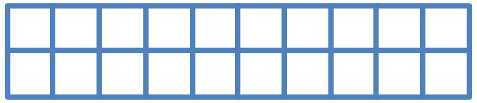

Kernel

# What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

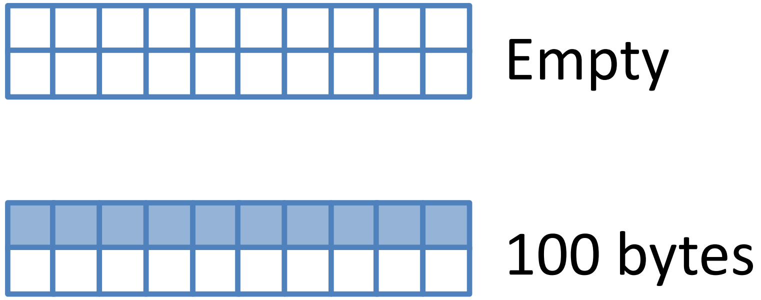
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here...)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

Socket buffer



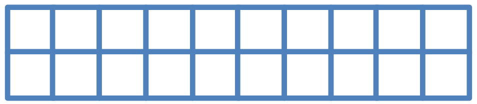

Kernel

# What should we do if the receive socket buffer is empty? If it has 100 bytes?

For each Process

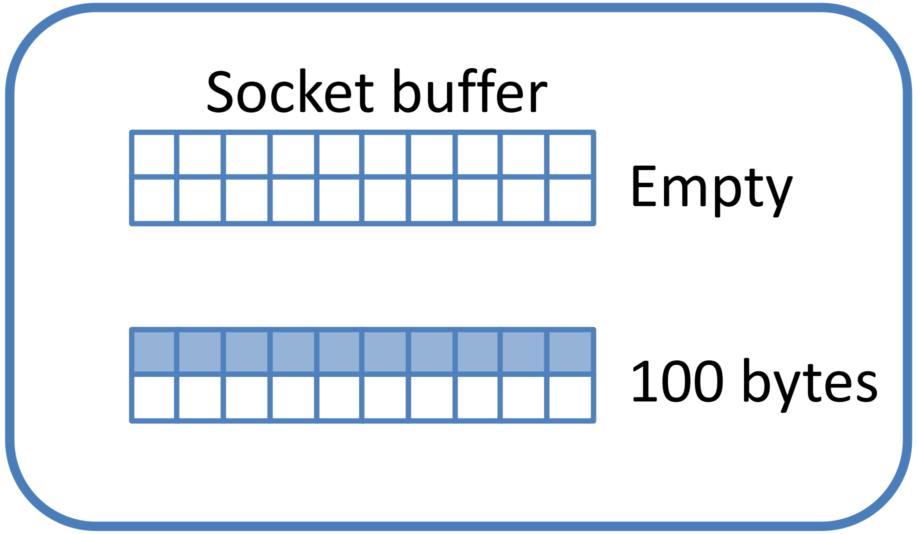
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here...)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

| | Empty | 100 Bytes |
|---|---|---|
| **A** | **Block** | **Block** |
| **B** | **Block** | **Copy 100 bytes** |
| **C** | **Copy 0 bytes** | **Block** |
| **D** | **Copy 0 bytes** | **Copy 100 bytes** |
| **E** | **Something else** | |



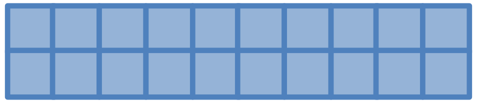

Kernel

# What should we do if the send socket buffer is full? If it has 100 bytes?

For each Process

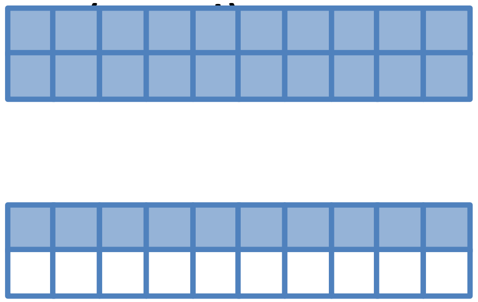
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

s_buf (size 200)



Two Scenarios:

Socket buffer



Full

100 bytes

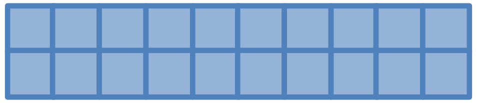
Kernel

# What should we do if the send socket buffer is full?  If it has 100 bytes?

For each Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

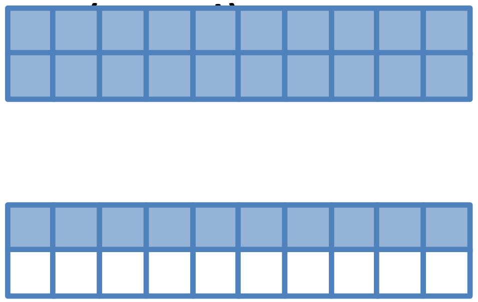
s_buf (size 200)



Two Scenarios:

| | Full | 100 Bytes |
|---|---|---|
| **A** | **Return 0** | **Copy 100 bytes** |
| **B** | **Block** | **Copy 100 bytes** |
| **C** | **Return 0** | **Block** |
| **D** | **Block** | **Block** |
| **E** | **Something else** | |

Socket buffer



Full

100 bytes

Kernel

# Blocking Implications

recv()

- Do not assume that you will recv() all of the bytes that you ask for.
- Do not assume that you are done receiving.
- Always receive in a loop!*

send()

- Do not assume that you will send() all of the data you ask the kernel to copy.
- Keep track of where you are in the data you want to send.
- Always send in a loop!*

* Unless you're dealing with a single byte, which is rare.

# ALWAYS check send()/recv() return values!

When recv() returns a non-zero number of bytes always call recv() again until:

- the server closes the socket,
- or you've received all the bytes you expect.

# ALWAYS check send()/recv() return values!

When recv() returns a non-zero number of bytes always call recv() again until:

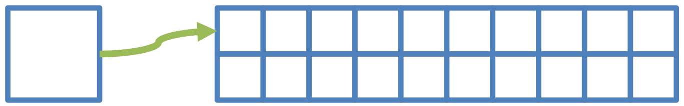
- In the case of your web client: keep receiving until the server closes the socket.

# ALWAYS check send()/recv() return values!

- E.g.: Let's assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

Data:

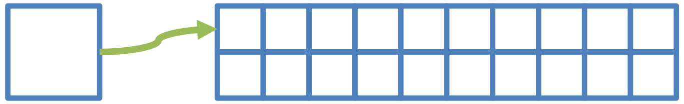


# ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

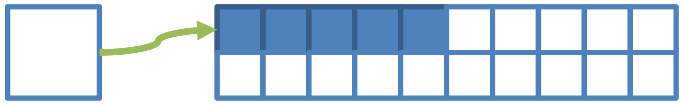
Data:



---

Data received = 50
Remaining buffer size = 150

Data:

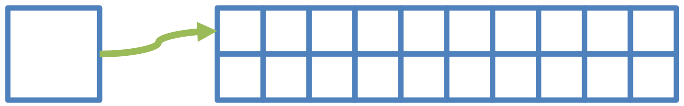


# ALWAYS check send()/recv() return values!

- E.g.: Let’s assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown
recv(sock, data, 200, 0);

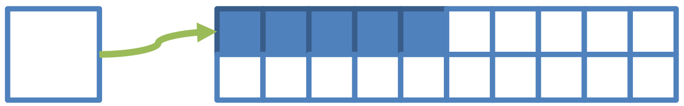
Data:



---

Data received = 50
Remaining buffer size = 150

Data:



// Receive remaining bytes from offset of 50

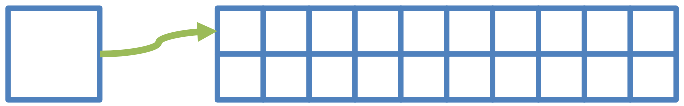
# ALWAYS check send()/recv() return values!

- E.g.: Let's assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown

recv(sock, data, 200, 0);

Data:

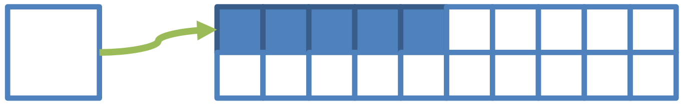


---

Data received = 50

Remaining buffer size = 150

Data:



// Receive remaining bytes from offset of 50

recv(sock, data + 50, 200 – 50, 0)

Data received = ?

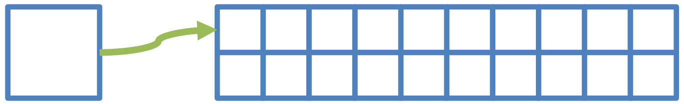
# ALWAYS check send()/recv() return values!

- E.g.: Let's assume we have a 200 byte data buffer and we want to receive data from a server.

Data size to receive = unknown

recv(sock, data, 200, 0);

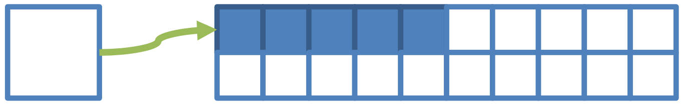
Data:



Data received = 50

Remaining buffer size = 150

Data:



// Receive remaining bytes from offset of 50

recv(sock, data + 50, 200 – 50, 0)

Data received = ?

Repeat until server closes the socket. (return value = 0)

# Blocking Summary

**send()**

- Blocks when socket buffer for sending is full
- Returns less than requested size when buffer cannot hold full size

**recv()**

- Blocks when socket buffer for receiving is empty
- Returns less than requested size when buffer has less than full size

Always check the return value!

# Create a TCP socket: socket()

int socket(int domain, int type, int protocol)

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
```

- domain: communication domain of the socket: generic interface.
- type of socket: reliable vs. best-effort
- end-to-end protocol: TCP for a stream socket -
  - 0: default E2E for specified protocol family and type.

# Create a TCP socket: socket()

int socket(int domain, int type, int protocol)

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
/* AF_INET: Communicate with IPv4 Address Family (AF),
  SOCK_STREAM: Stream-based protocol
  int sock: returns an integer-valued socket descriptor or handle
*/
  if(sock < 0) { // If socket() fails, it returns -1
     perror("socket");
     exit(1);
  }
```

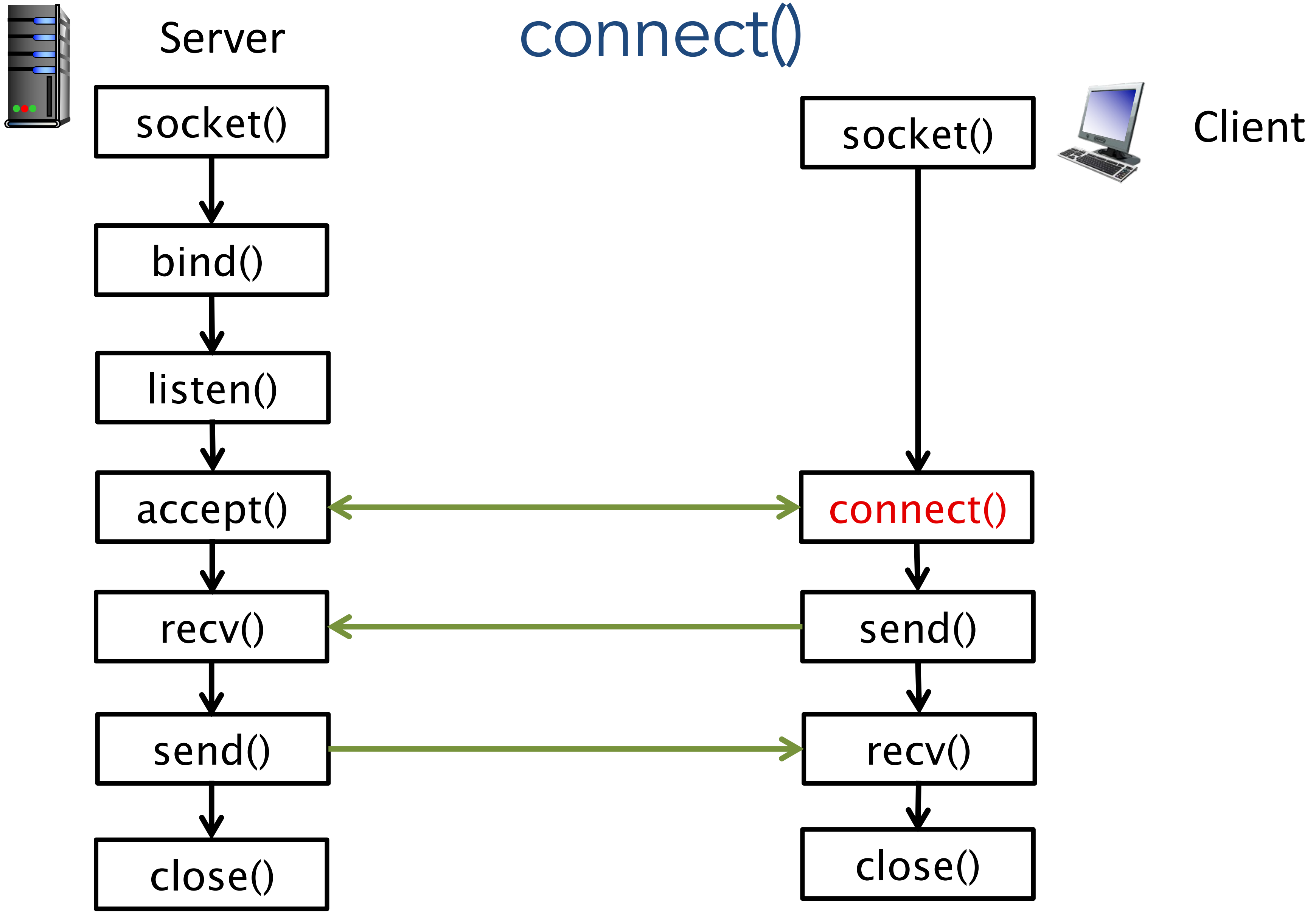
# Close a socket: close()

int close(int socket)

```
if (close(sock)) {
        perror("close");
        exit(1);
   }
/* int socket: int socket descriptor is passed to close()*/
```

- Close operation similar to closing a file.
- initiate actions to shut down communication
- deallocate resources associated with the socket
- cannot send(), recv() after you close the socket.

# connect()


Server
socket()
bind()
listen()
accept()
recv()
send()
close()
socket()
Client
connect()
send()
recv()
close()

# connect()

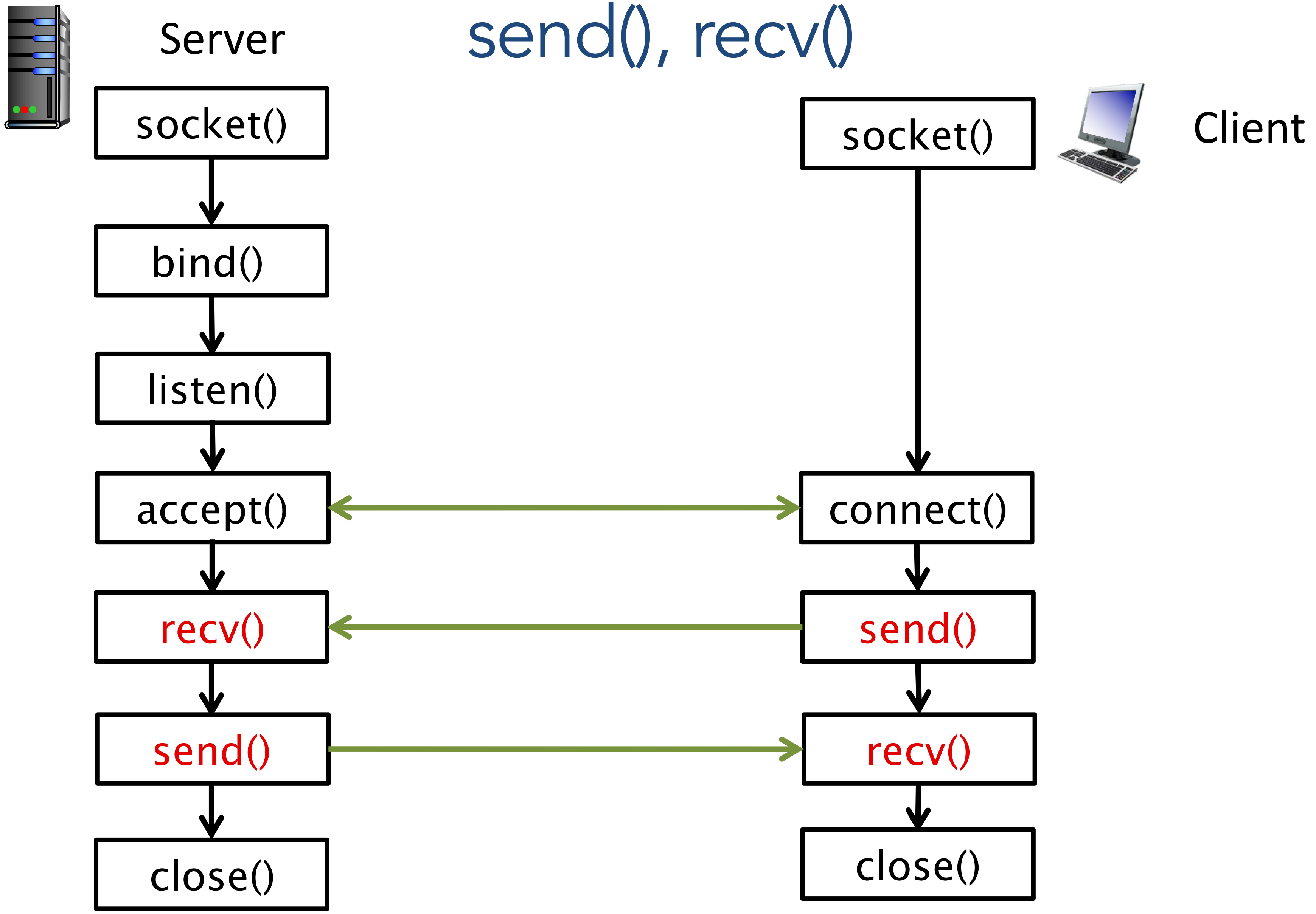
- Before you can communicate, a connection must be established.
- Client Initiates, Server waits.
- Once connect() returns, socket is connected and we can proceed with send(), recv()

```
int connect(int socket,
            const struct sockaddr *foreign Address, socklen_t addressLength)
```

# connect()

```
int connect(int socket,
        const struct sockaddr *foreign Address,             socklen_t addressLength)
struct sockaddr_in addr;
int res = connect(sock, (struct sockaddr*)&addr, sizeof(addr));
/* int socket: socket descriptor
   foreignAddress: pointer to sockaddr_in containing Internet address, port of server.
   addressLength: length of address structure
*/
```

# send(), recv()


Server
socket()
bind()
listen()
accept()
recv()
send()
close()
Client
socket()
connect()
send()
recv()
close()

# send(), recv()

Socket is connected when:

- client calls connect()
- connected socket is returned by accept() on server

ssize_t send(int socket, const void *msg, msgLength, int flags)
ssize_t recv (int socket, void *rcvBuffer, size_t bufferLength, int flags)

/* int socket: socket descriptor
return: # bytes sent/received or -1 for failure.

# send()

send():

- by default send: blocks until data is sent

ssize_t send(int socket, const void *msg, msgLength, int flags)

/* int socket: socket descriptor

send(): msg: sequence of bytes to be sent

send(): mesgLength: # bytes to send

# send(), recv()

recv():

ssize_t recv (int socket, void *rcvBuffer, size_t bufferLength, int flags)

int recv_count = recv(sock, buf, 255, 0);

/* int socket: socket descriptor

void *rcvBuffer: generally a char array

size_t bufferLength: length of buffer: max # bytes that can be received at once.

flags: setting flag to zero specifies default behavior.

Place all send() and recv() calls in a loop, until you are left with no more bytes to send or receive. One call to send()/recv(), irrespective of the buffer does not necessarily mean all your data will be received at once.

# Request Method Types ("verbs")

HTTP/1.0 (1996):

- GET:
  - Requests page.
- POST:
  - Uploads user response to a form.
- HEAD:
  - asks server to leave requested object out of response

HTTP/1.1 (1997 & 1999):

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field
- TRACE, OPTIONS, CONNECT, PATCH
- Persistent connections

# Uploading form input

GET (in-URL) method:

- uses GET method
- input is uploaded in URL field of request line:

www.somesite.com/animalsearch?monkeys&banana

POST method:

- web page often includes form input
- input is uploaded to server in request entity body

# GET vs. POST

GET can be used for idempotent requests

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

# GET vs. POST

GET can be used for idempotent requests

- Idempotence: an operation can be applied multiple times without changing the result (the final state is the same)

Q: How many of the following operations are idempotent?

I. Incrementing a variable
II. Assigning a value to a variable
III. Allocating Memory
IV. Compiling a program

A. None of them
B. One of them
C. Two of them
D. Three of them
E. All of them

# GET vs. POST

GET can be used for idempotent requests.

- Idempotence:  an operation can be applied multiple times without changing the result (the final state is the same)

# GET vs. POST

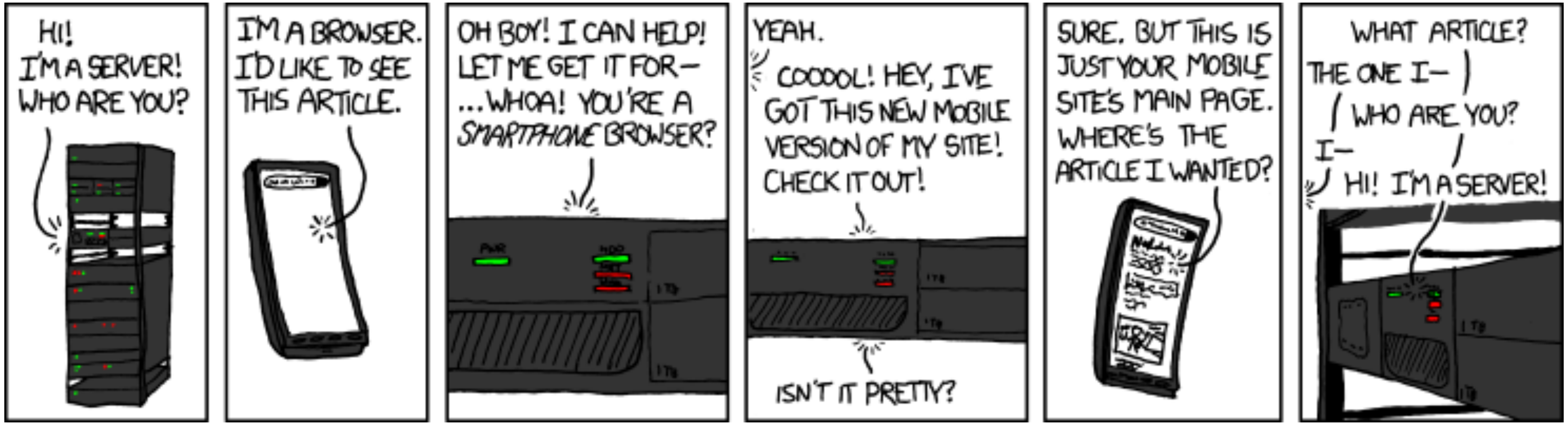
POST should be when:

- A request changes the state of the server or DB
- Sending a request twice would be harmful: (Some) browsers warn about sending multiple post requests
- Users are inputting non-ASCII characters
- Input may be very large
  - You want to hide how the form works/user input

# When might you use GET vs. POST?

| | GET | POST |
|---|---|---|
| A. | Forum post | Search terms, Pizza order |
| B. | Search terms, Pizza order | Forum post |
| C. | Search terms | Forum post, Pizza order |
| D. | Forum post, Search terms, Pizza Order | |
| E. | | Forum post, Search terms, Pizza Order |

# State(less)




(XKCD #869, "Server Attention Span")

# HTTP State

Does the HTTP protocol, allow for a server to keep track of every client?

A. Yes, it’s required to
B. No, it would not scale
C. That’s against privacy rules!
D. Something else

# State(less)

- Original web: simple document retrieval
- Maintain State? Server is not required to keep state between connections

  ...often it might want to though
- Authentication: Client is not required to identify itself
  - server might refuse to talk otherwise though

# User-server state: cookies

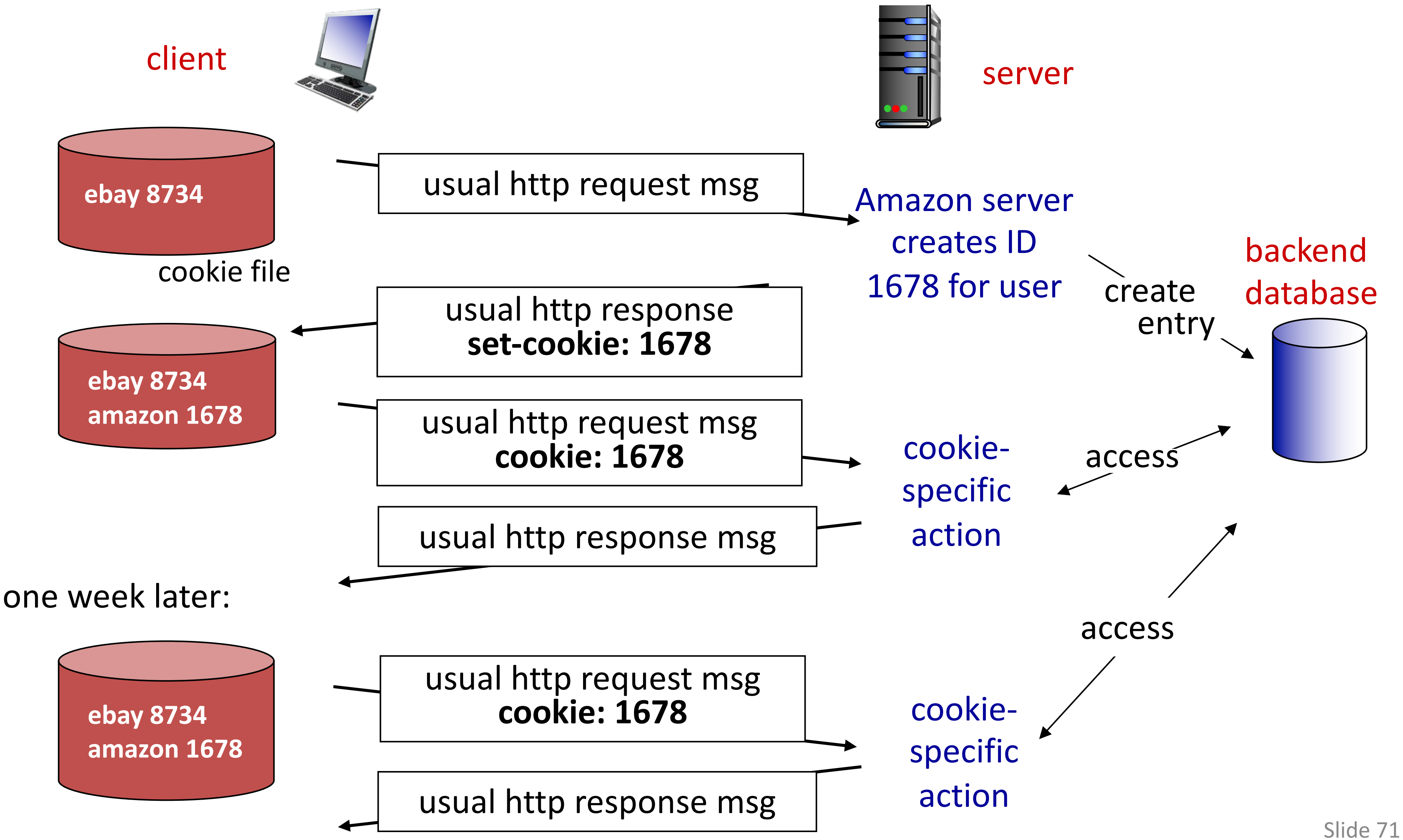
What cookies can be used for:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

How to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

# Cookies: keeping "state" (cont.)



# User-server state: cookies

Many web sites use cookies

Four components:

1) cookie header line of HTTP response message

2) cookie header line in next HTTP request message

3) cookie file kept on user's host, managed by user's browser

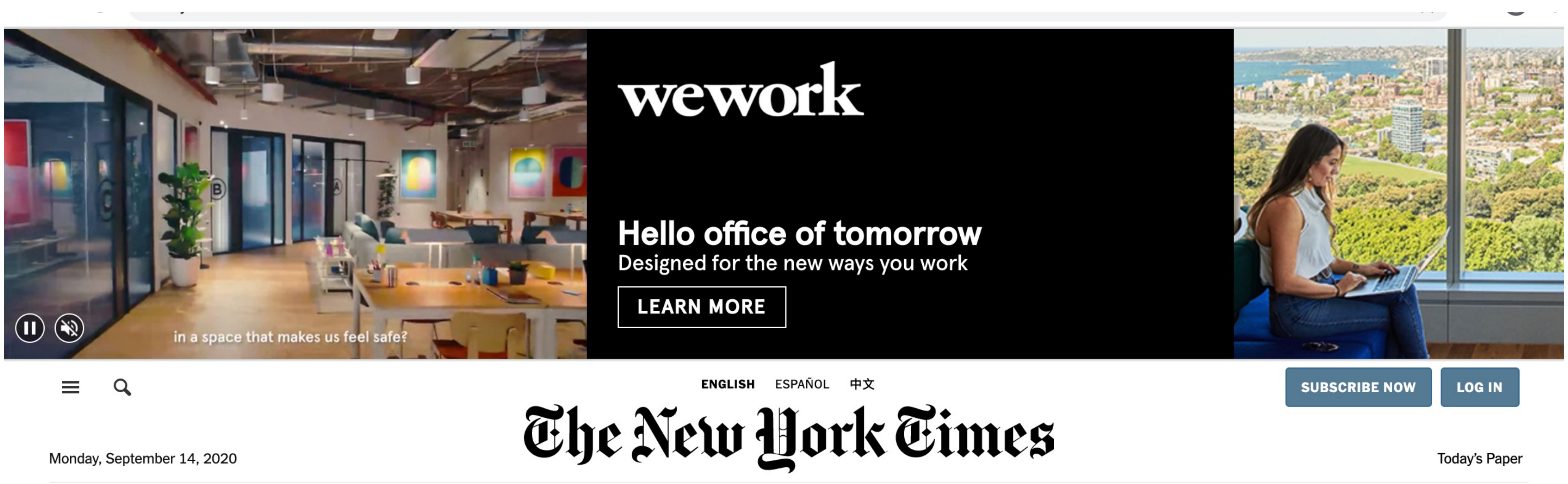
4) back-end database at Web site

# Cookies and Privacy

Cookies permit sites to learn a lot about you

supply name and e-mail to sites (and more!)

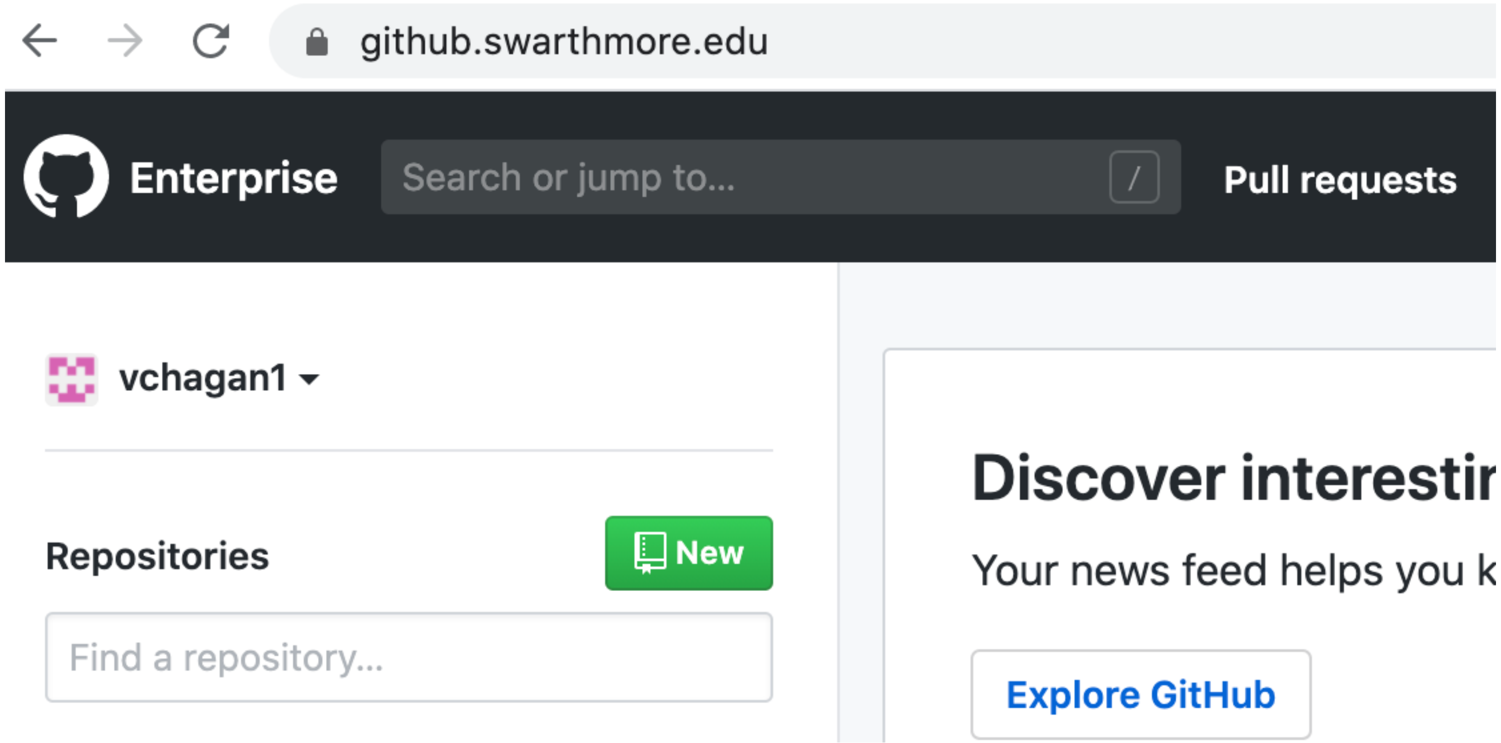
third-party cookies (ad networks) follow you across multiple sites.



# Cookies and Privacy

Cookies permit sites to learn a lot about you

You could turn them off …but good luck doing anything on the internet!



# HTTP connections

**Non-persistent HTTP**

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects requires multiple connections

**Persistent HTTP**

- multiple objects can be sent over single TCP connection between client, server

object: image, script, stylesheet, etc.

# Non-persistent HTTP

suppose user enters URL: contains references to 10 jpeg images

1a. HTTP client initiates TCP connection to HTTP server

1b. HTTP server "accepts" connection, notifying client

2. HTTP client sends HTTP *request message: URL*

3. HTTP server:
   - receives request
   - forms *response message* containing requested *index.html*
   - sends message

4. HTTP server closes TCP connection.

5. HTTP client receives response:
   - index.html
   - finds 10 referenced jpeg objects

time

6. Steps 1-5 repeated for each of 10 jpeg objects

# Pseudocode Example

non-persistent HTTP

```
for object on web page:
    connect to server
    request object
    receive object
    close connection
```

persistent HTTP

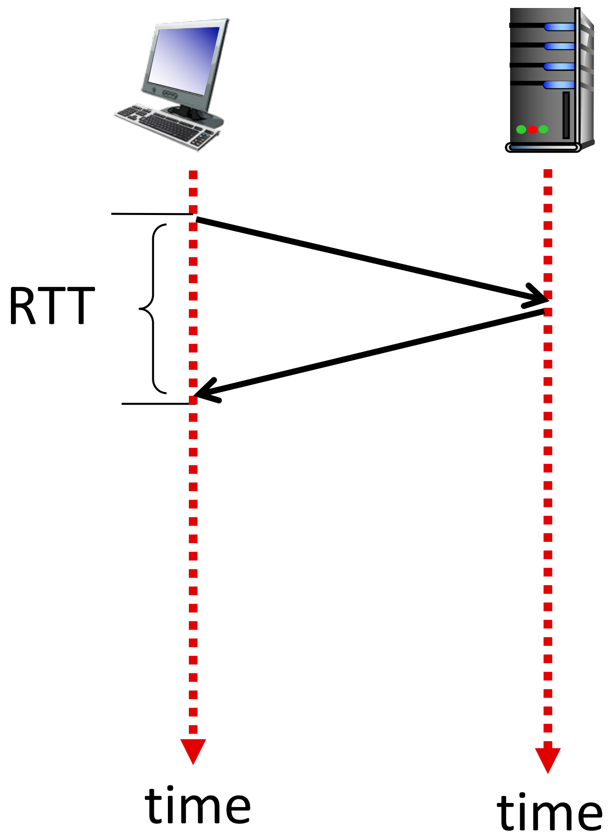
```
connect to server
for object on web page:
    request object
    receive object
close connection
```

# Round Trip Time

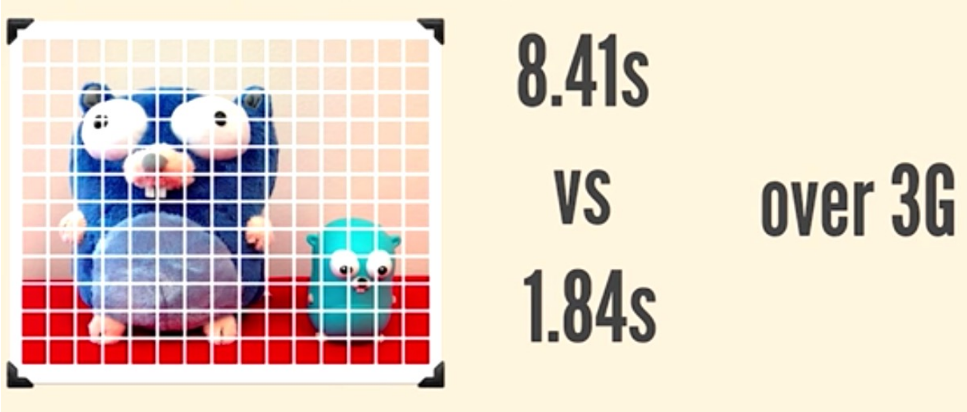
Round Trip Time (RTT):

- time for a small packet to travel from client to server and response to come back.

- Connection establishment (via TCP) requires one RTT.



# HTTP 1.x vs HTTP 2.0



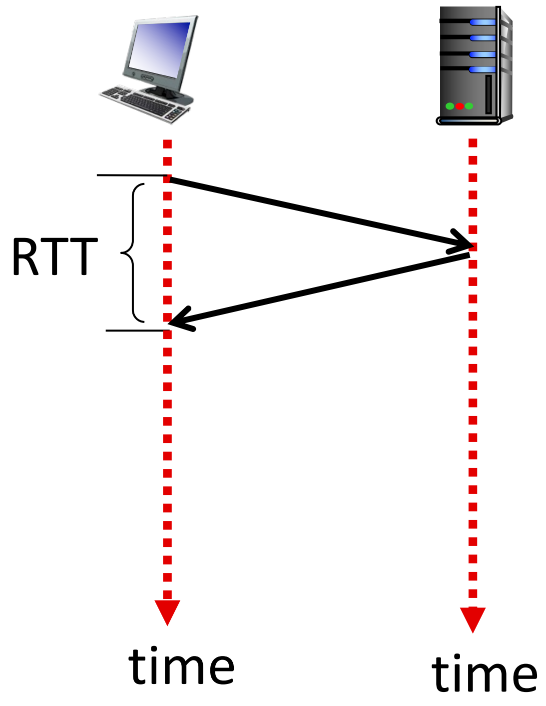

Courtesy: HTTP/2 101 Chrome Dev Summit 2015

Learn more: https://http2.github.io/

- SPDY: protocol to speed up the web: Basis for HTTP 2.0
- Request pipelining
- Compress header metadata

# Non-Persistent HTTP Connections can download a website with several objects in…

A. One RTT + (File transfer time per object)
B. (One RTT + File transfer time) per object
C. Two RTTs
D. Two RTTs + (File transfer time per object)
E. (Two RTTS + File transfer time) per object



# Non-persistent HTTP: response time

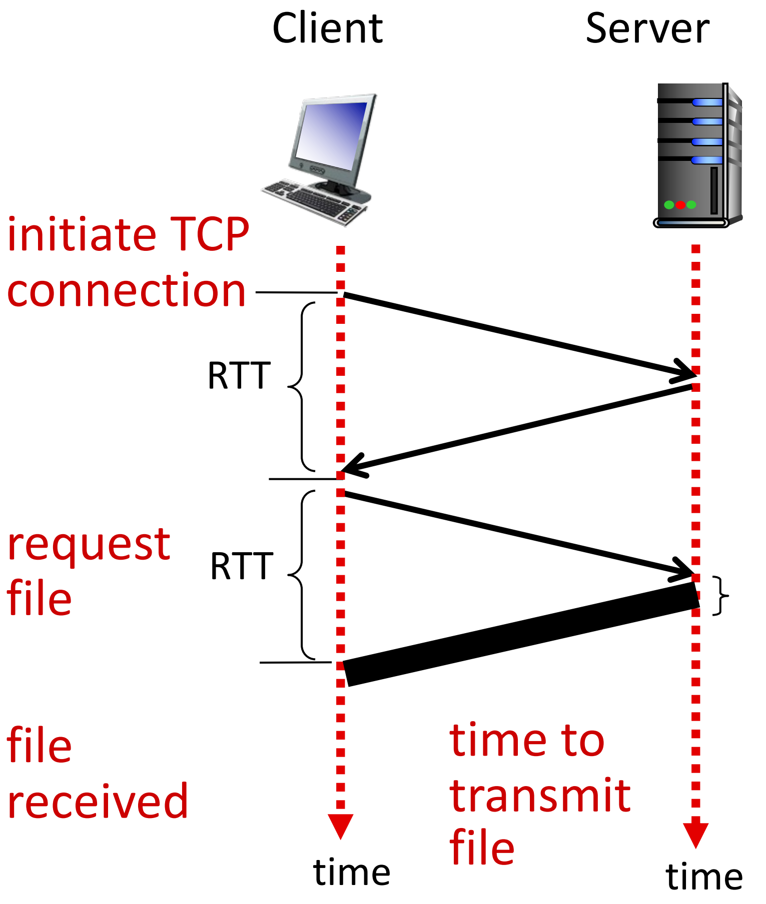
Round Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time:

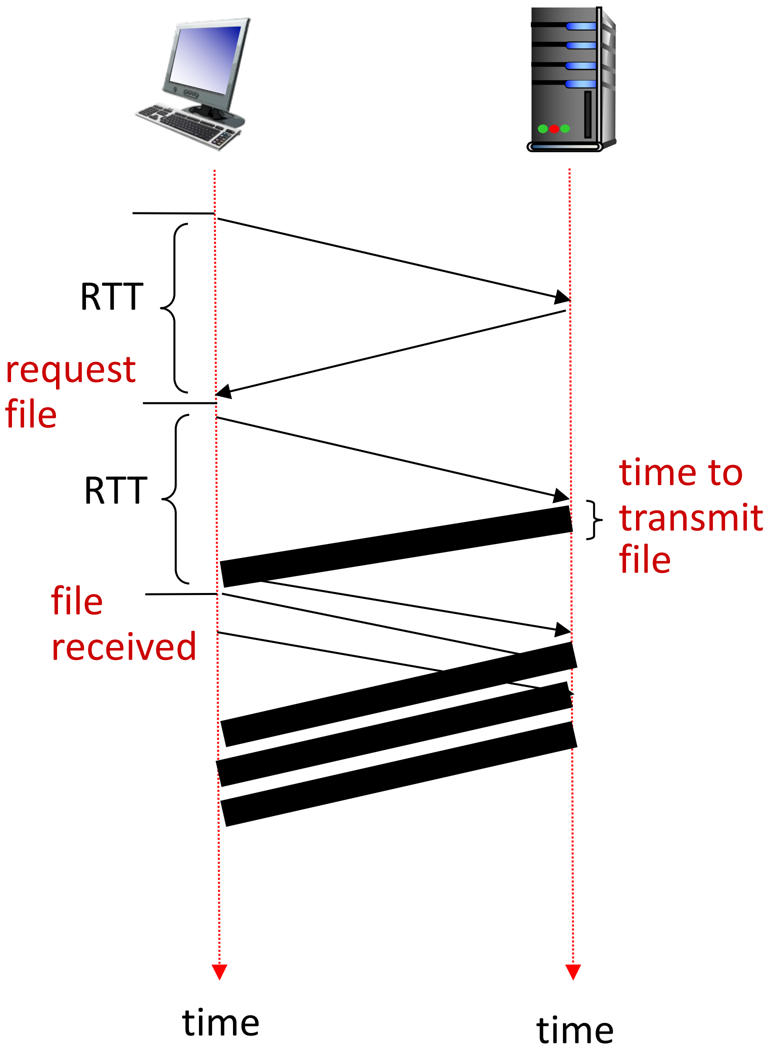
- 1-RTT to initiate TCP connection
- 1-RTT for HTTP request + first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2-RTT+ file transmission time

  <u>For each object</u>



# Persistent Connection



# Persistent HTTP

**Non-persistent HTTP issues:**

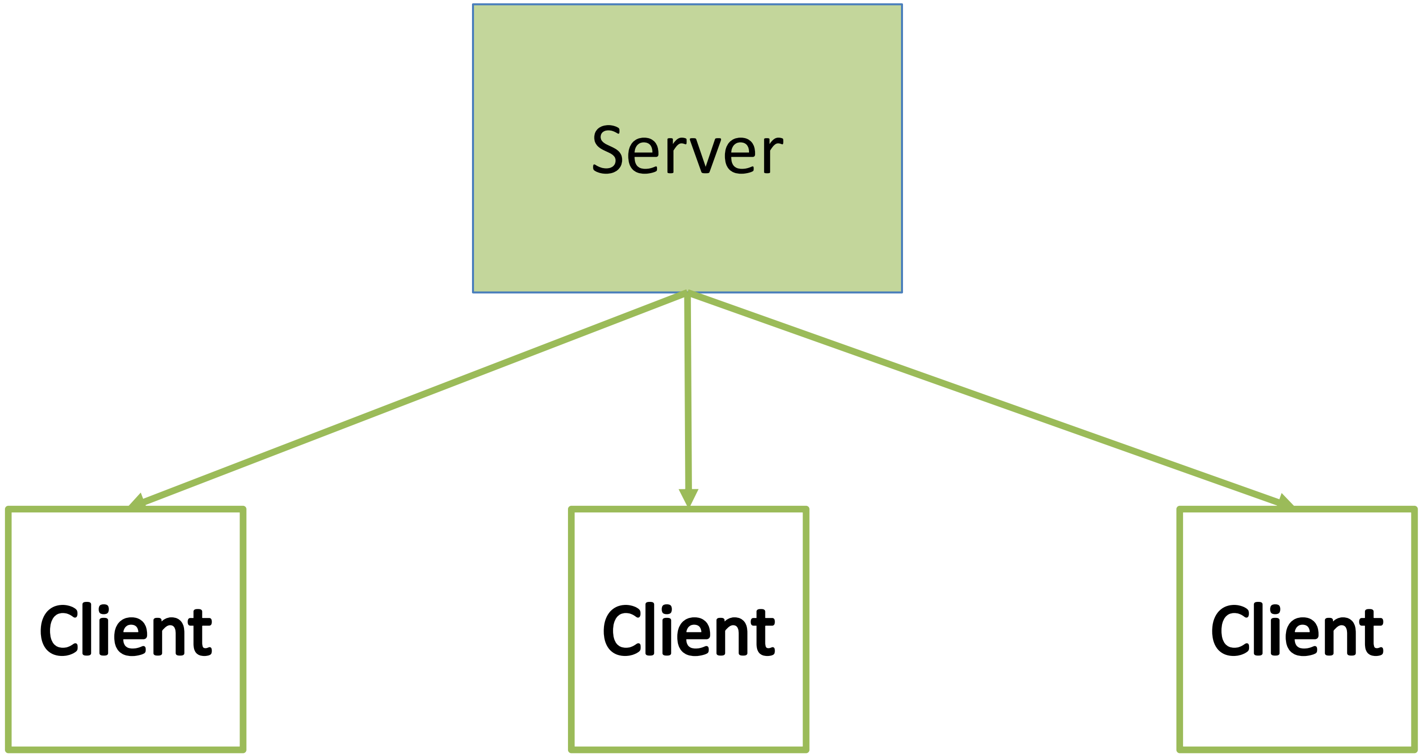
- requires 2 RTTs per object
- OS overhead for each TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

**Persistent HTTP:**

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

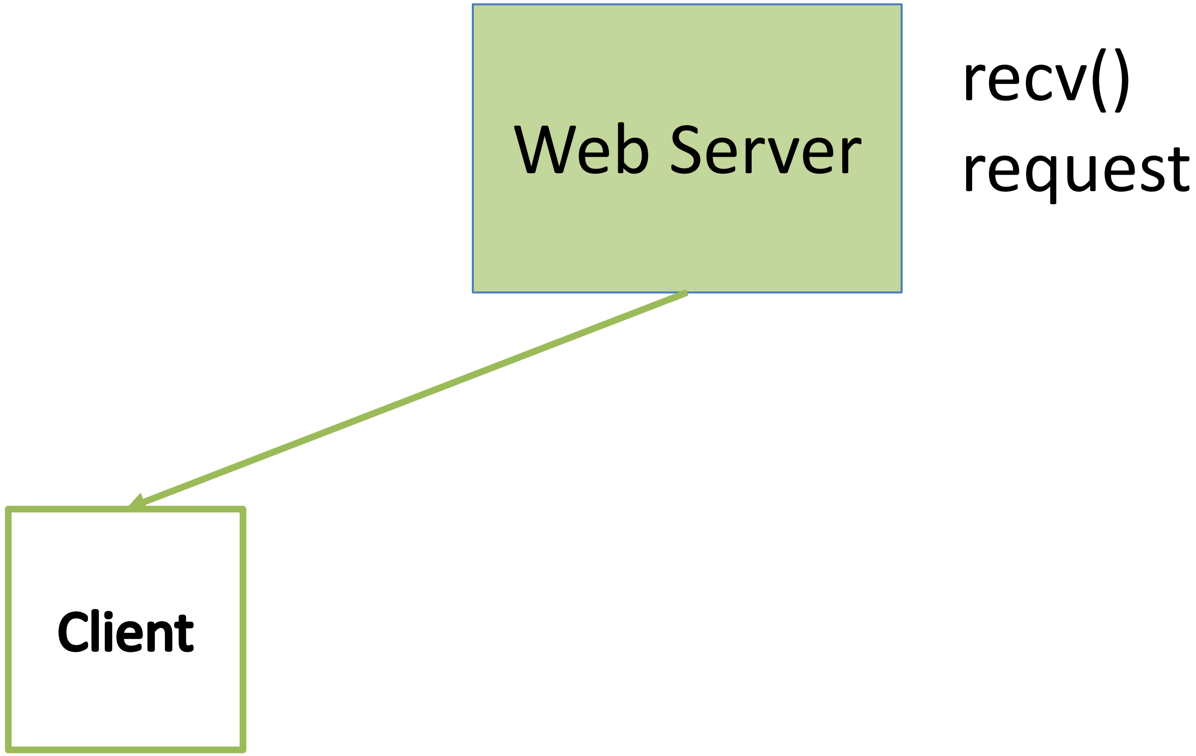
# Concurrency

- Think you're the only one talking to that server?



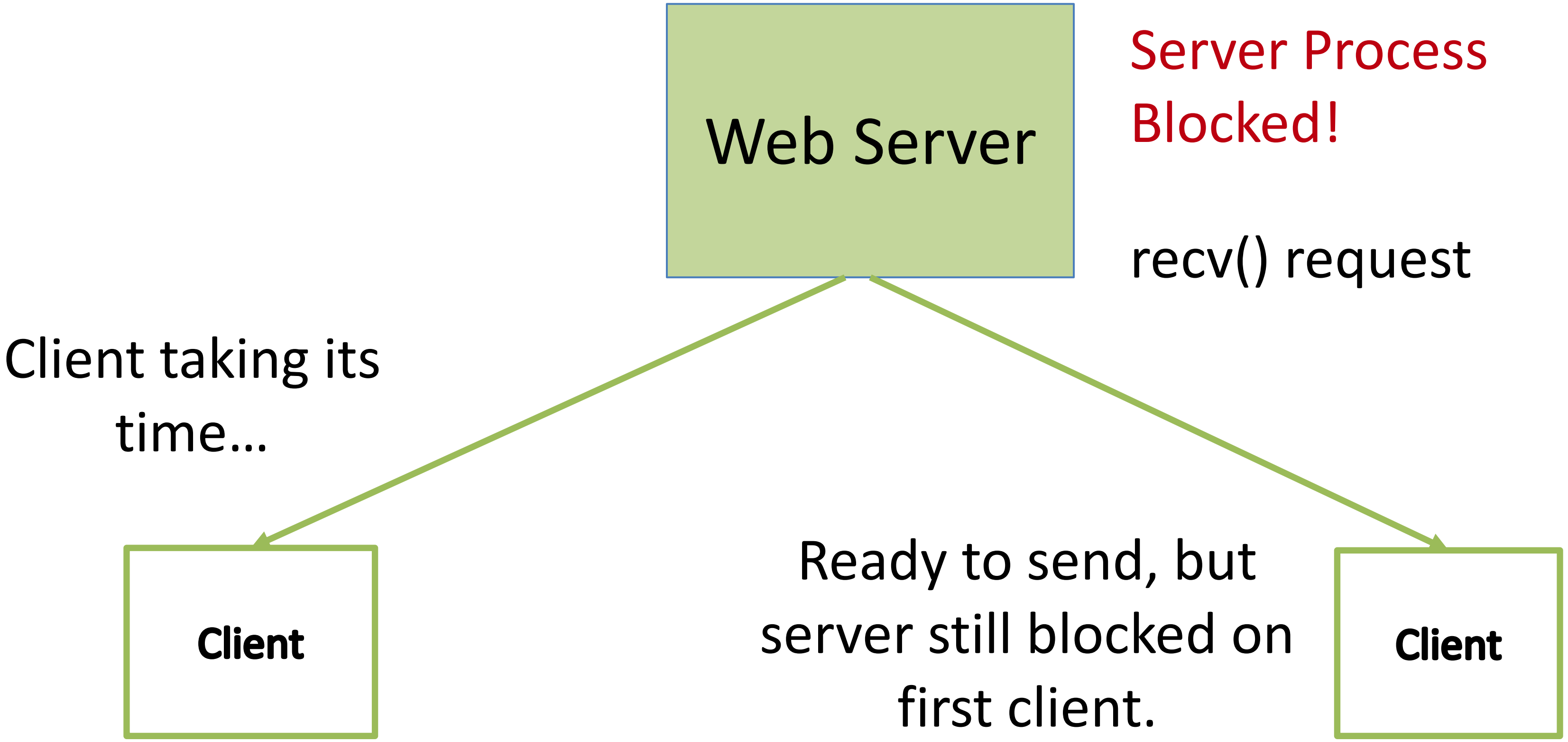
# Without Concurrency

- Think you're the only one talking to that server?



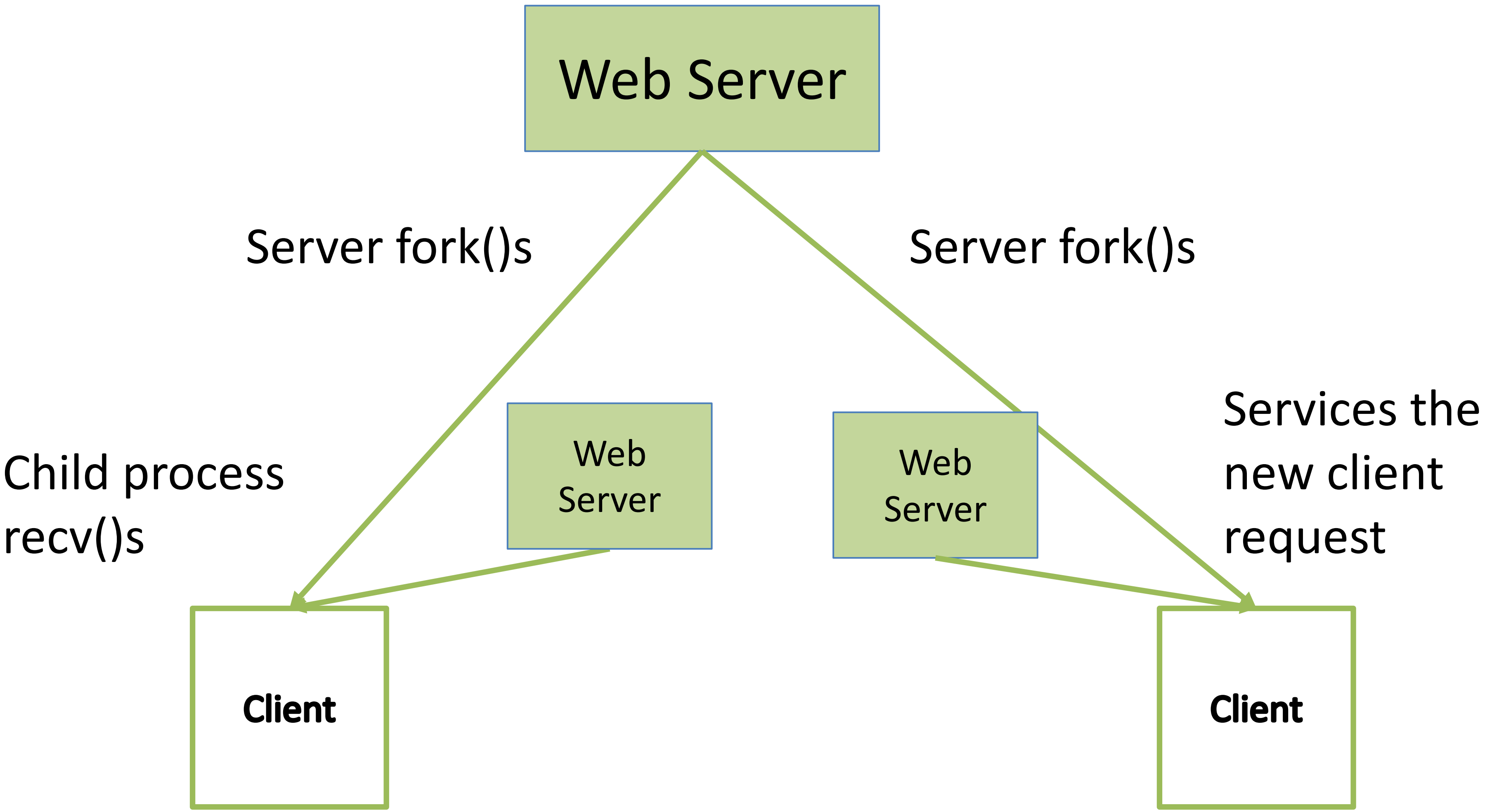
# Without Concurrency

- Think you’re the only one talking to that server?



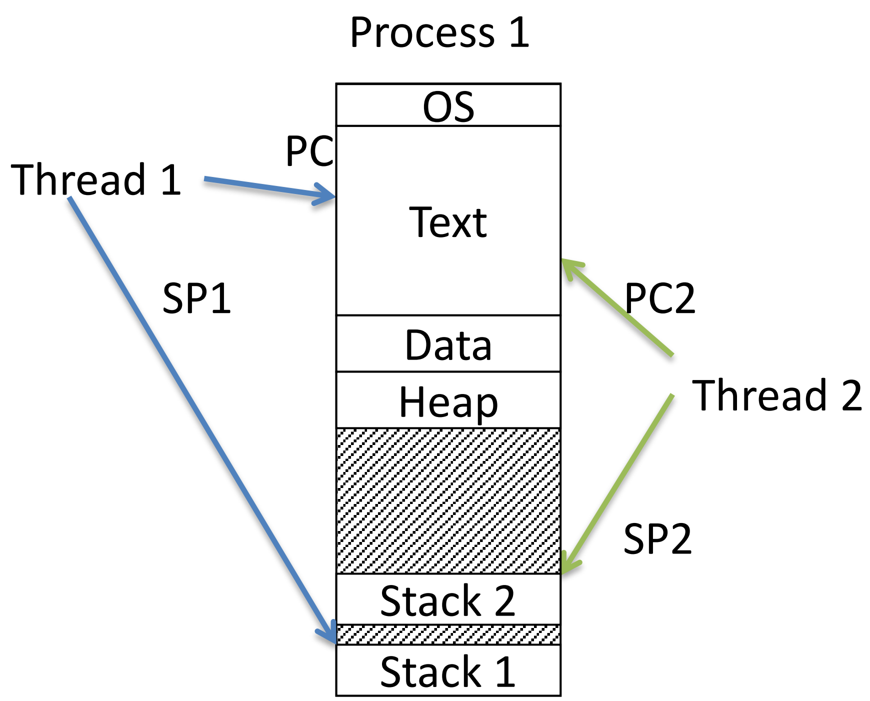

If only we could handle these connections separately…

# Multiple processes

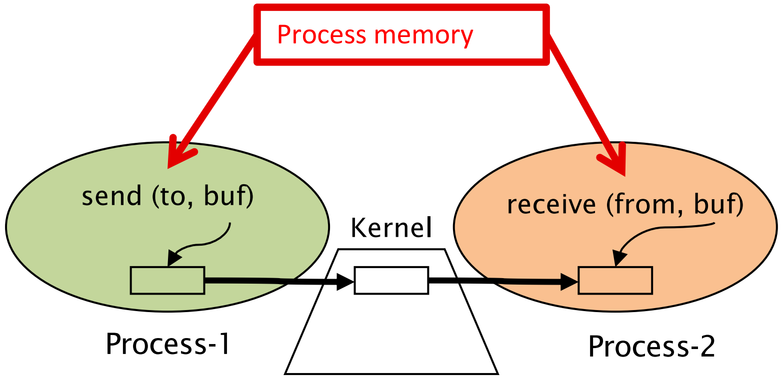


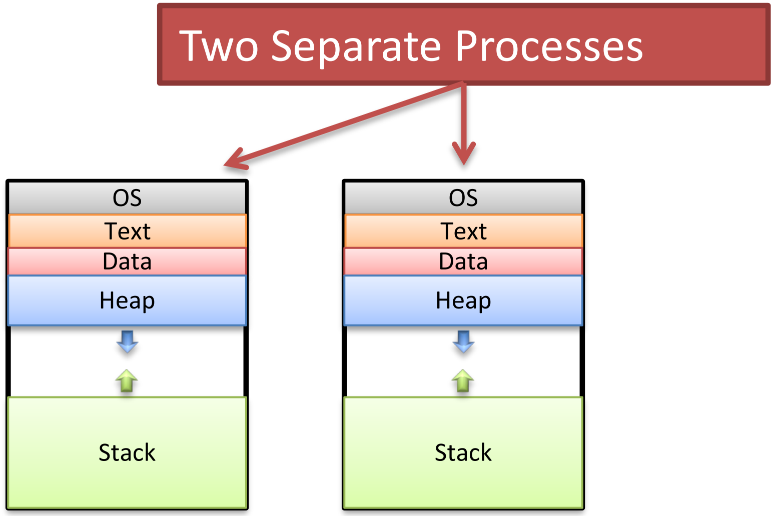
# Concurrent Web-servers with multiple threads/processes

- Threads (shared memory)




- Message Passing (locally)



# Processes/Threads vs. Parent
# (More details in an OS class…)

## Spawned Process

- Inherits descriptor table
- Does not share memory
  - New memory address space
- Scheduled independently
  - Separate execution context
  - Can block independently

## Spawned Thread

- Shares descriptor table
- Shares memory
  - Uses parent's address space
- Scheduled independently
  - Separate execution context
  - Can block independently

# Processes/Threads vs. Parent
# (More details in an OS class…)

### Spawned Process

- Inherits descriptor table
- Does not share memory
  - New memory address space
- Scheduled independently

### Spawned Thread

- Shares descriptor table
- Shares memory
  - Uses parent's address space
- Scheduled independently

Often, we don't need the extra isolation of a separate address space. Faster to skip creating it and share with parent – threading.

# Which benefit is most critical?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.

# Both processes and threads:

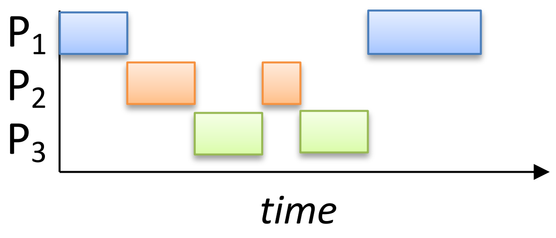
Several benefits

- Modularizes code: one piece accepts connections, another services them
- Each can be scheduled on a separate CPU
- Blocking I/O can be overlapped

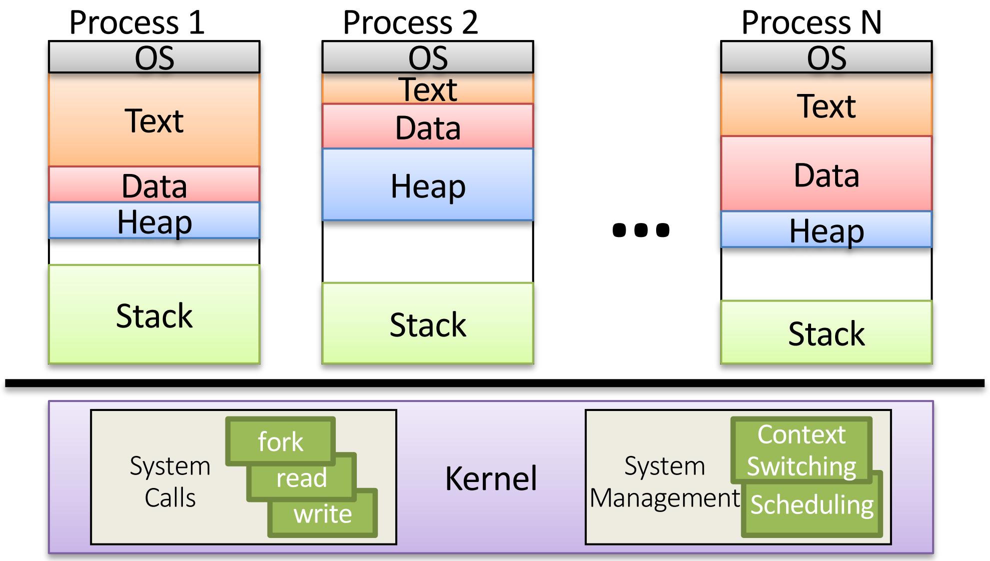
# Both processes and threads

## Still not maximum efficiency…

- Creating/destroying threads takes time
- Requires memory to store thread execution state
- Lots of context switching overhead




CPU: Time
Single core




Context Switching