CS 43: Computer Networks

04: Sockets, Concurrency and Non-blocking I/O

Sep 12, 2019

SWARTHMORE COLLEGE

Reading Quiz

Last class

- HTTP
  - GET vs. POST
  - response messages
- Cookies
- HTTP Performance
  - Persistence vs. Non-persistence

Today

- Under-the-hood look at system calls
  - Data buffering and blocking
- Inter-process communication
- Processes, threads and blocking

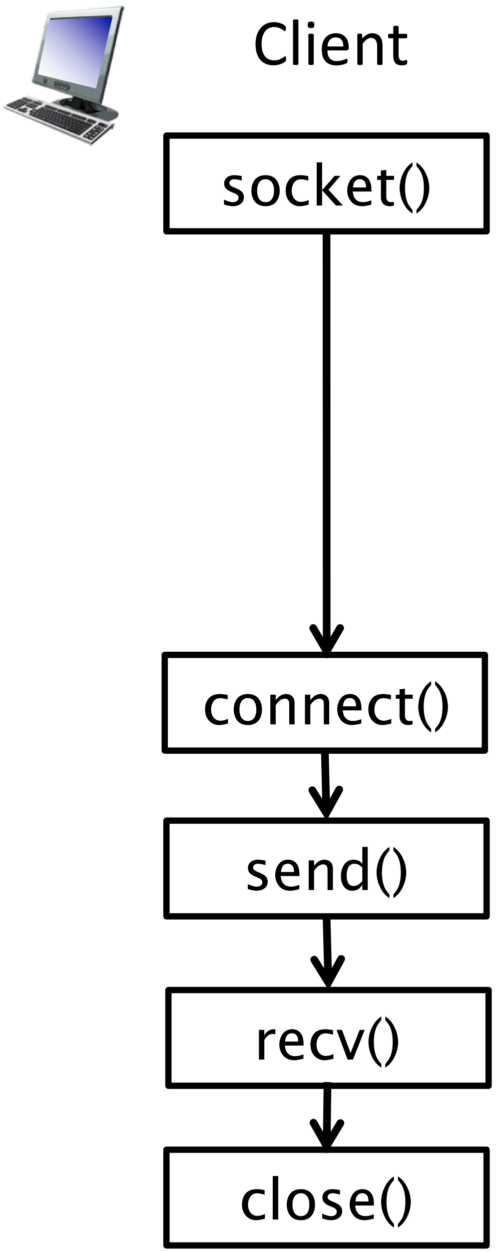
Client-Server communication

- 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 allows an application to read and write data to stable storage.

TCP Socket Procedures: Client



create a new communication endpoint

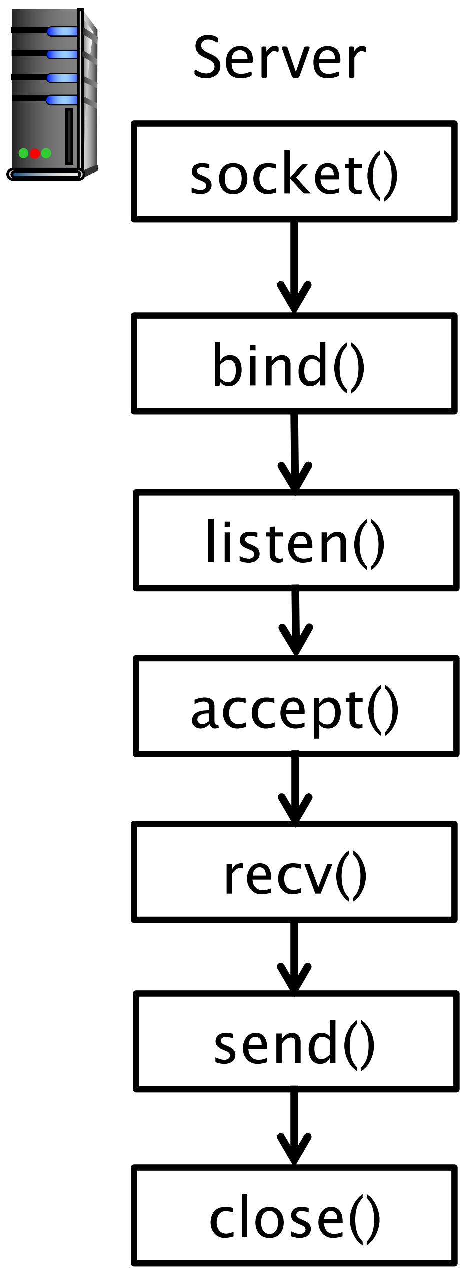
actively attempt to establish a connection

receive some data over a connection

send some data over a connection

release the connection

TCP Socket Procedures: Server



create a new communication endpoint

attach a local address to a socket

announce willingness to accept connections

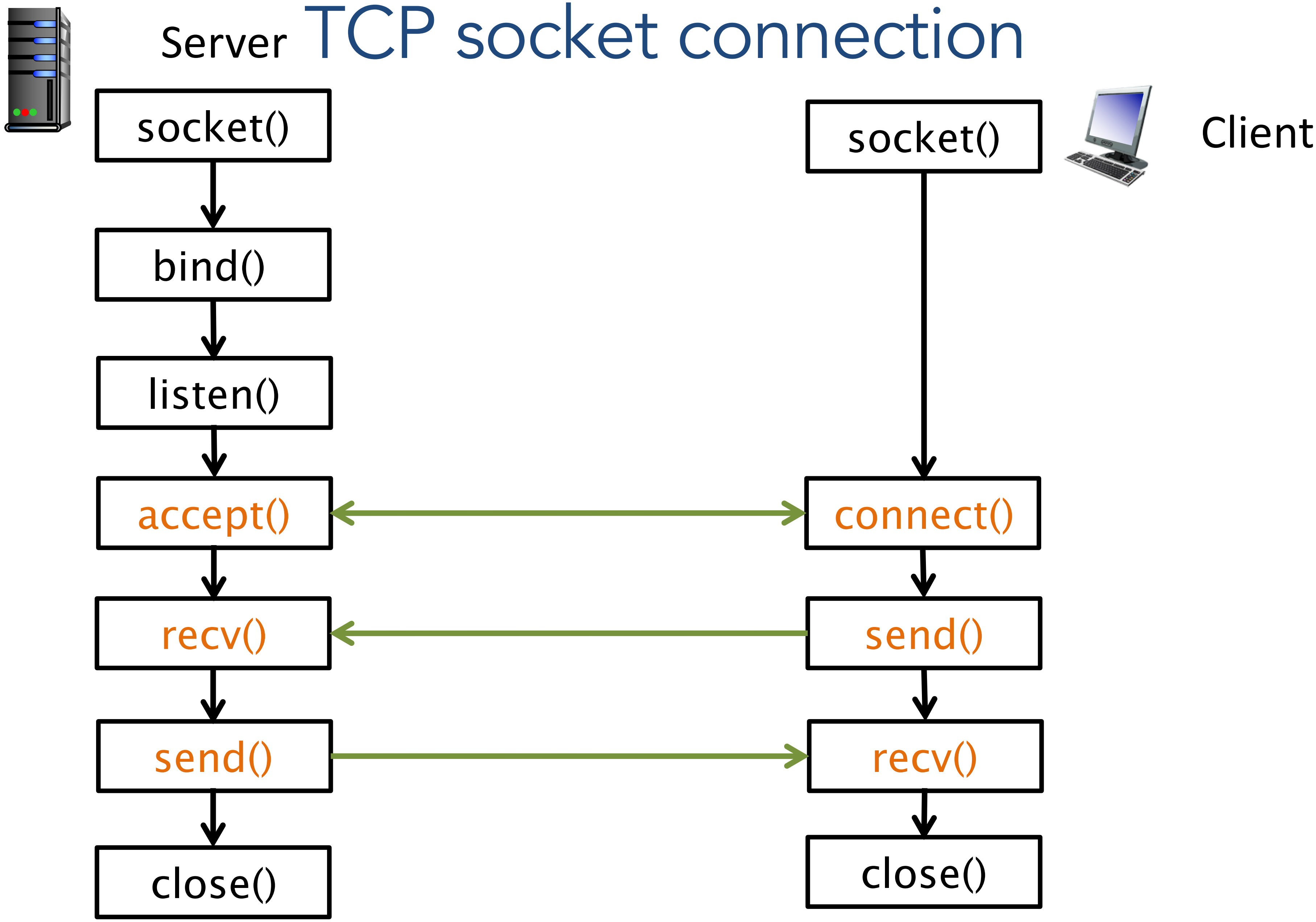
block caller until a connection request arrives

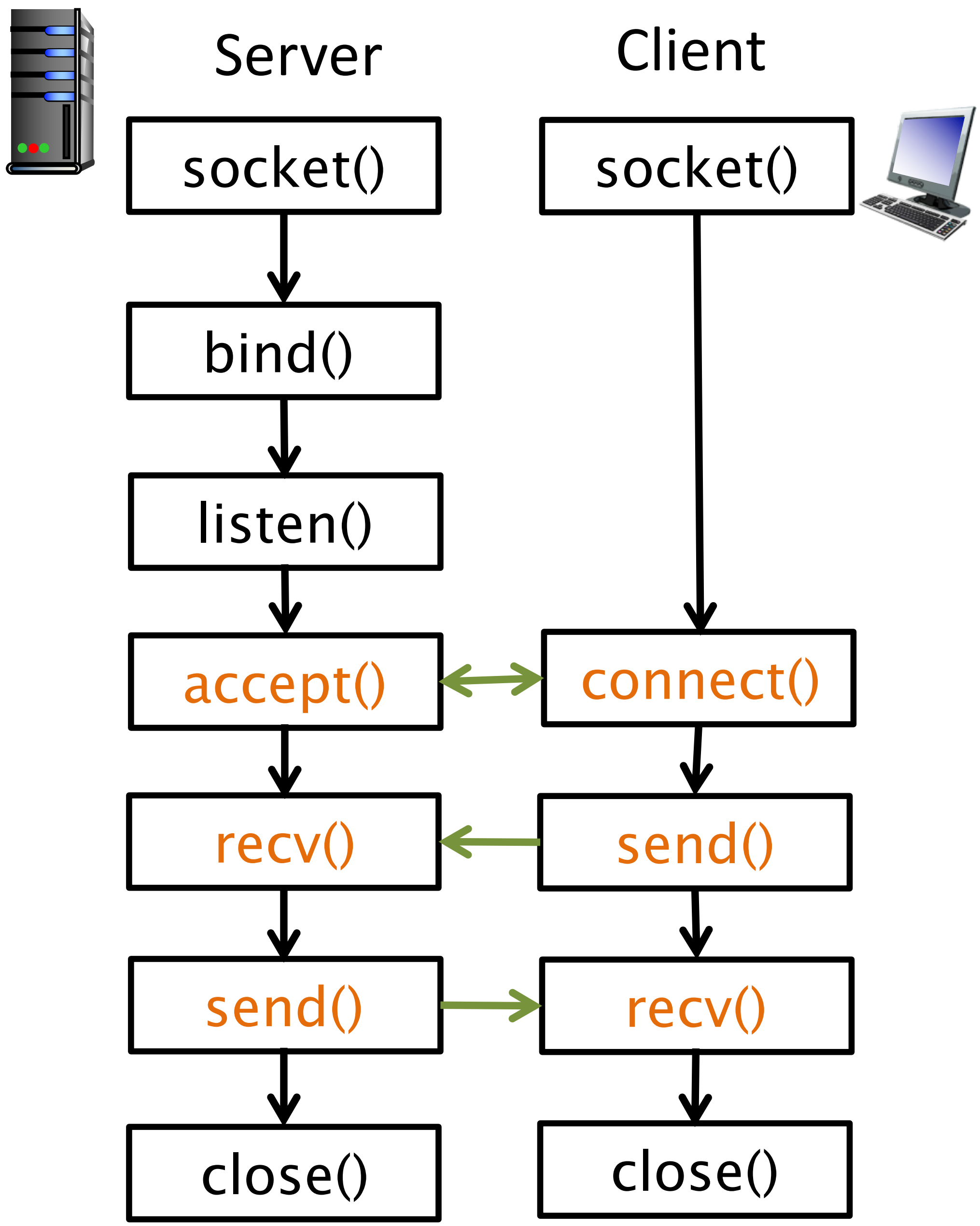
receive some data over a connection

send some data over a connection

release the connection

TCP socket connection

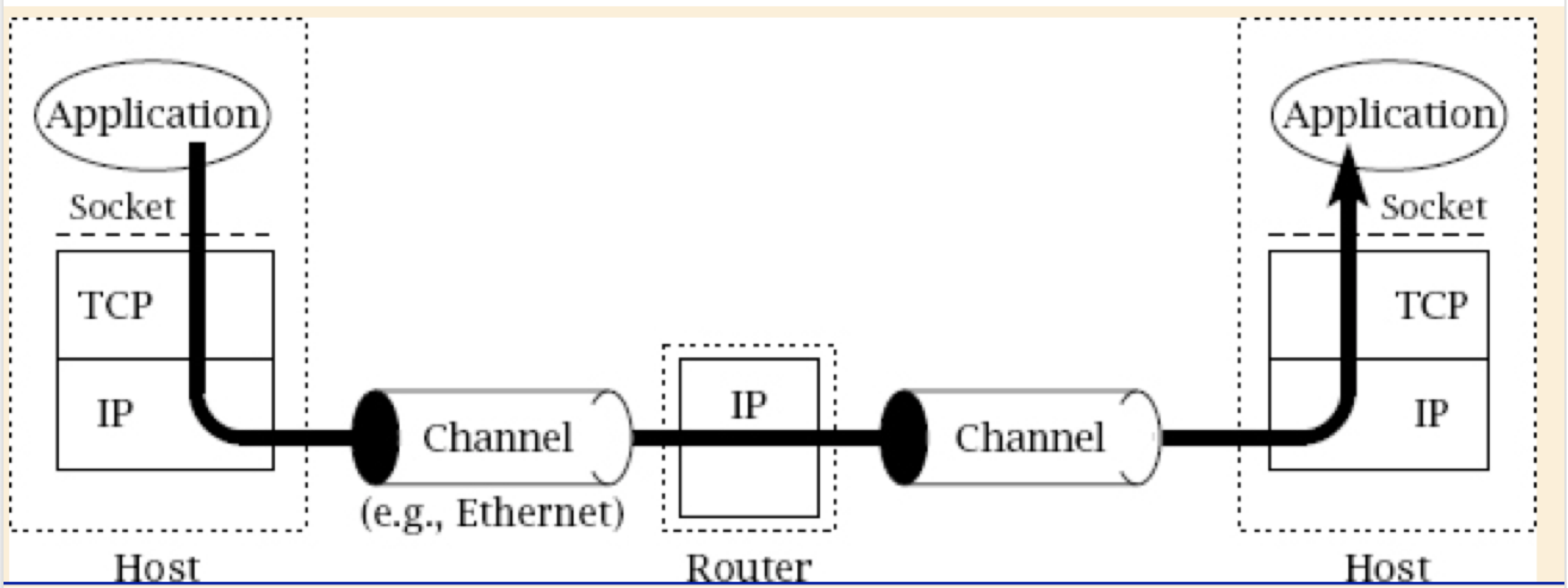




If the client sends a GET request to the server using send() but forgets to send the last /r/n which of the following can happen?

A. Server, Client both recv()
B. Server send()s, Client recv()s
C. Server recv()s, Client send()s
D. Some other combination

What happens when we call send and receive?



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

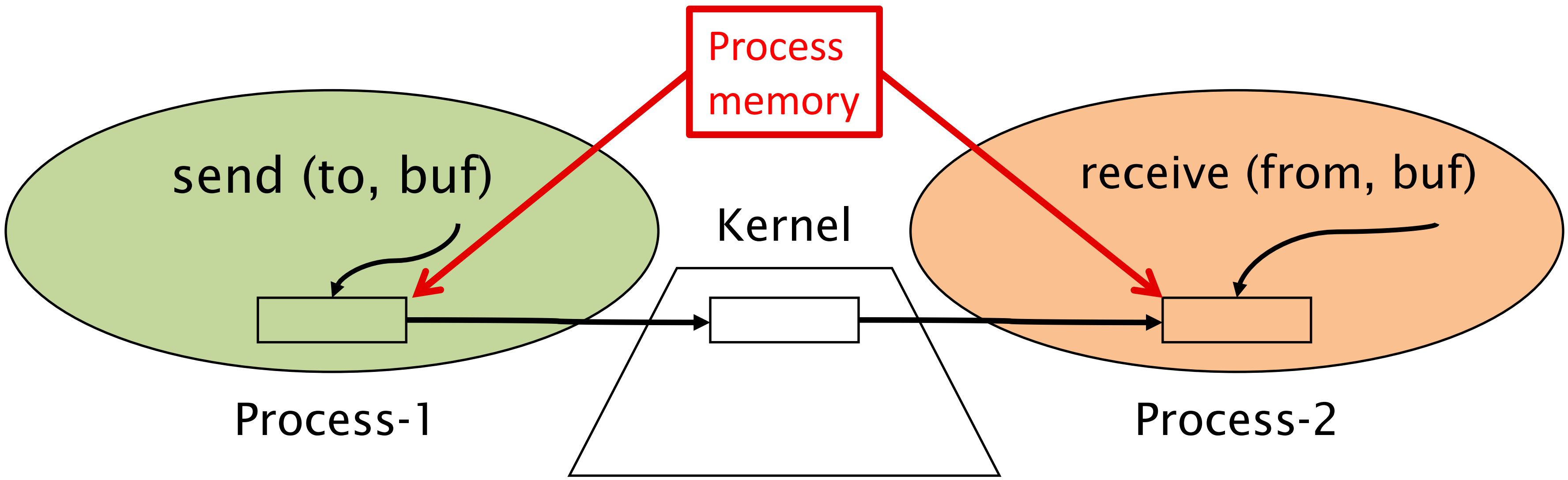
Recall: Inter-process communication

But first, let's go over inter-process communication within one machine

Recall 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

Message Passing (local)



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

Where is the synchronization in message passing Inter-process Communication?

A. The OS adds synchronization.

B. Synchronization is determined by the order of sends and receives.

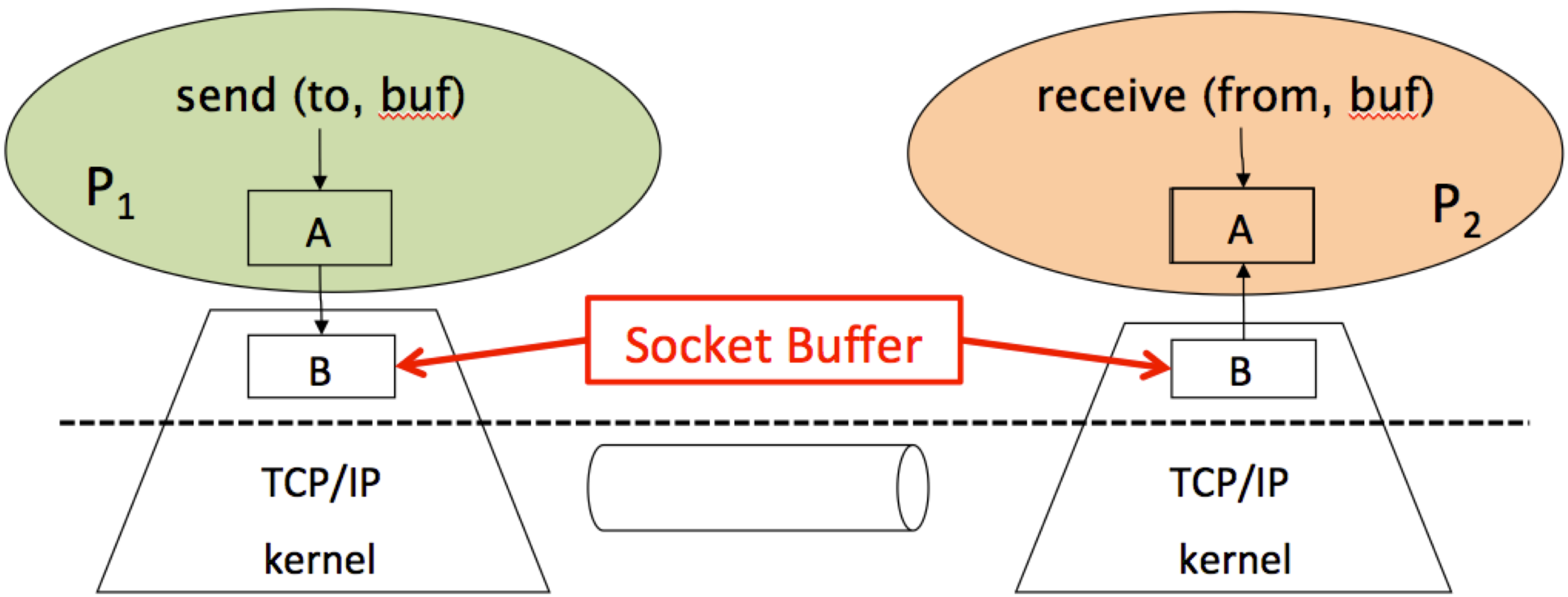
C. The communicating processes exchange synchronization messages (lock/unlock).

D. There is no synchronization mechanism.

Interprocess Communication (non-local)

- Processes must communicate to cooperate
- Must have two mechanisms:
  - Data transfer
  - Synchronization
- Across a network:
  - Threads (shared memory) <u>NOT AN OPTION</u>!
  - 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**().

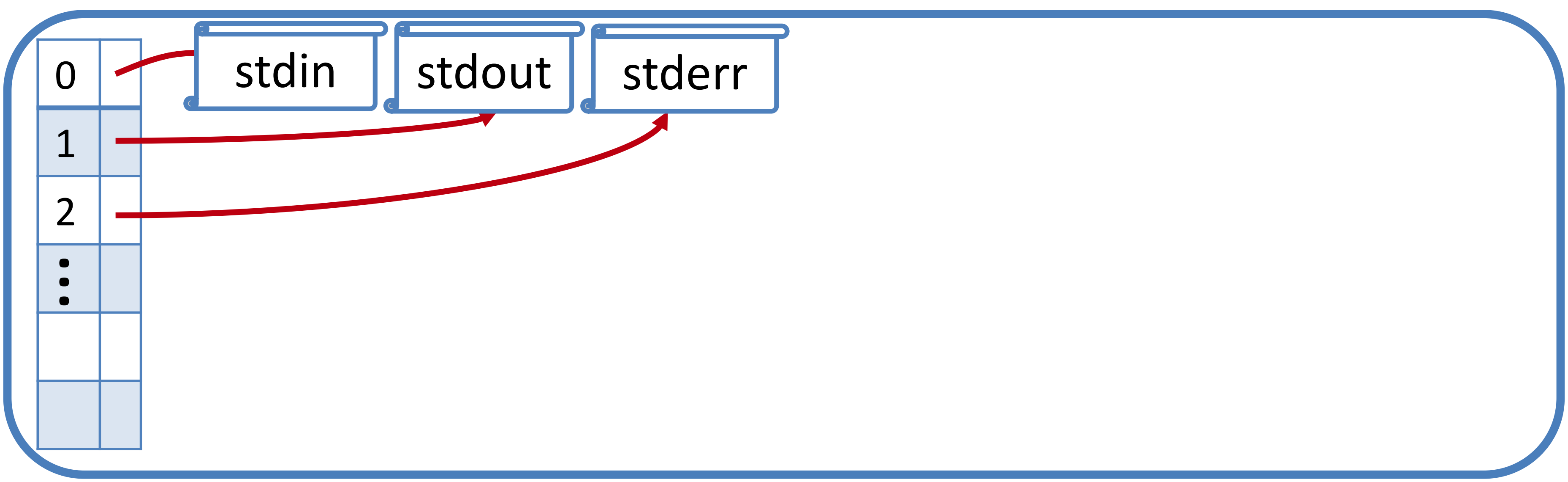
```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Descriptor Table

For each Process

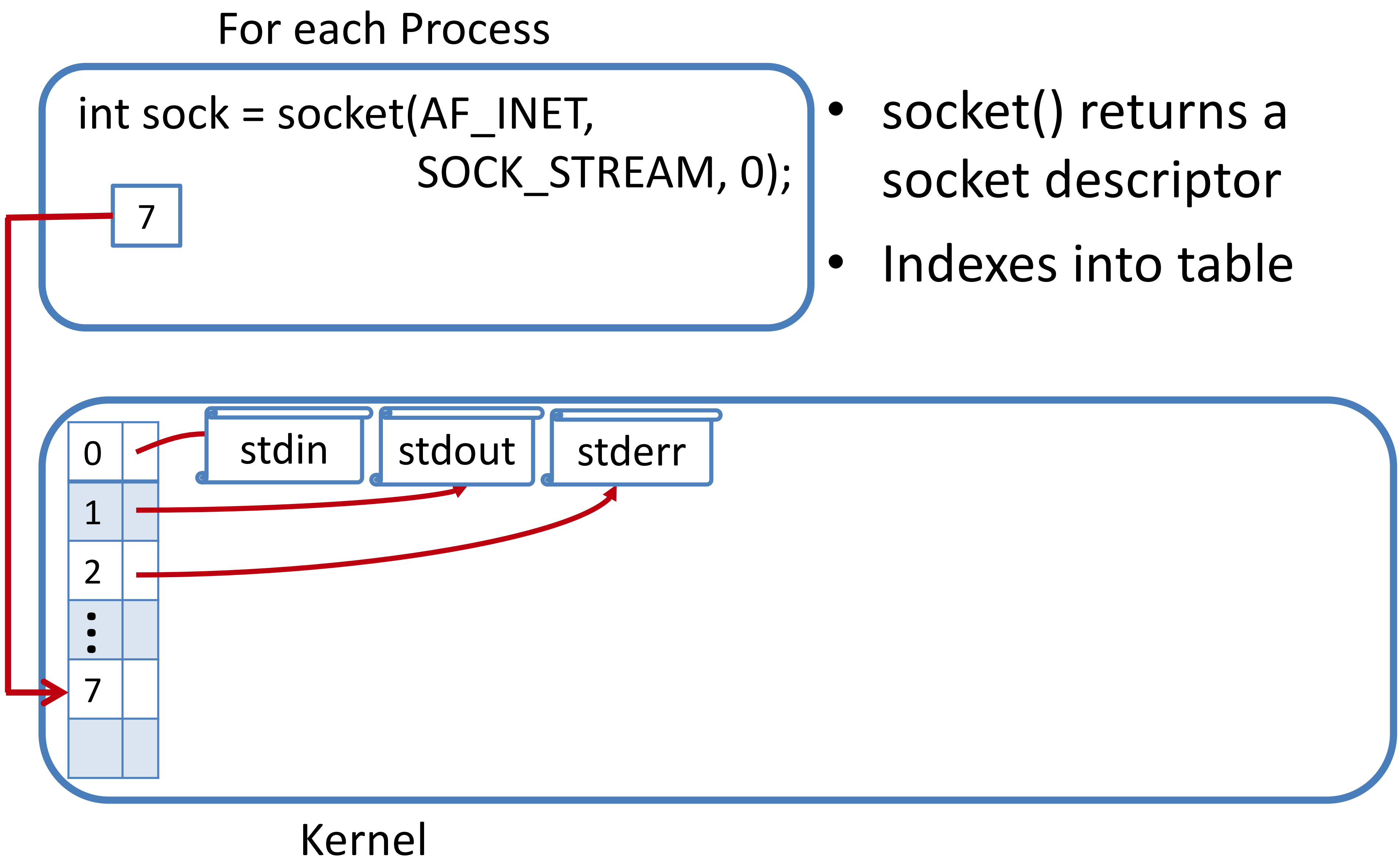
OS stores a table, per process, of descriptors

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



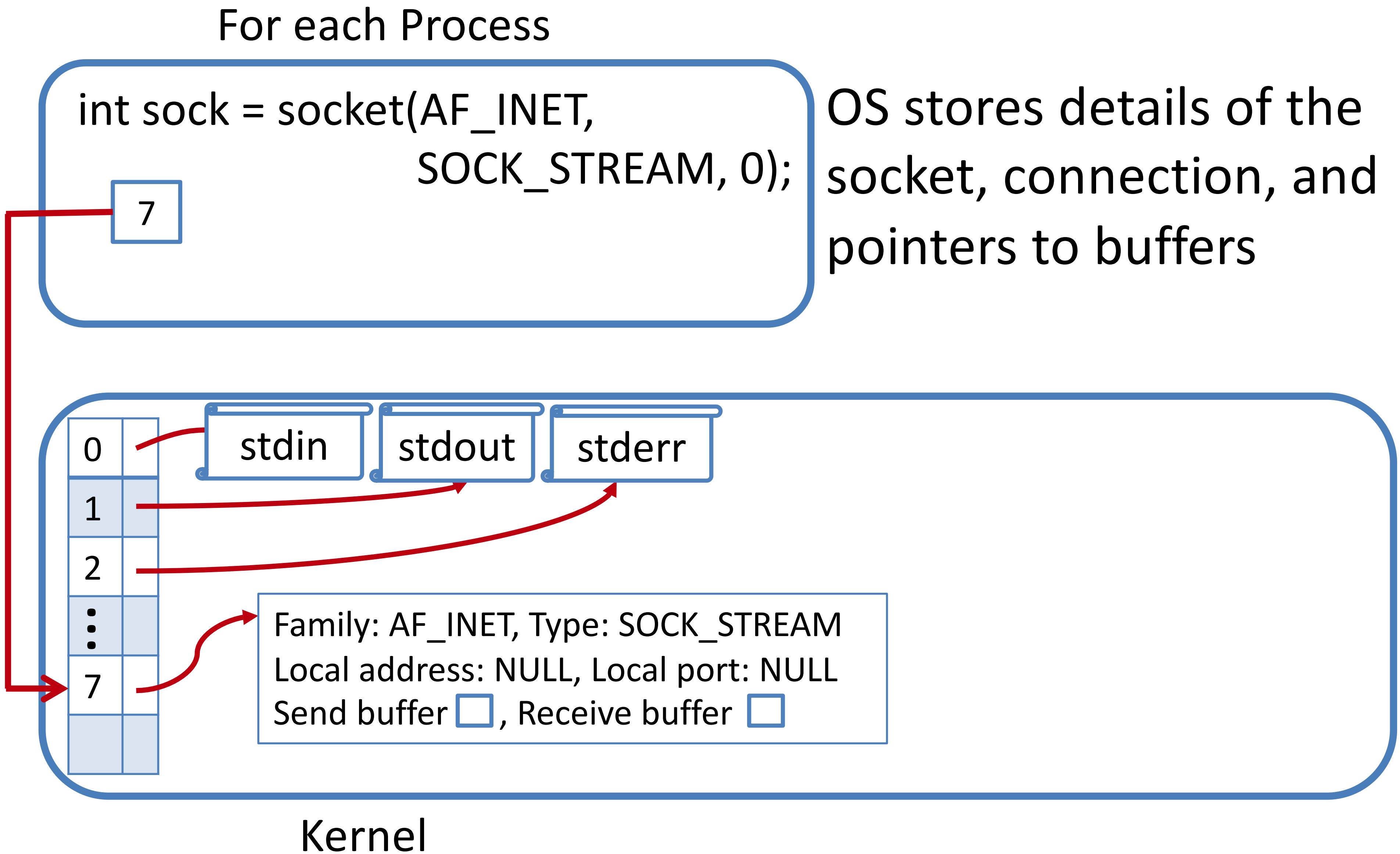
Kernel

socket()



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

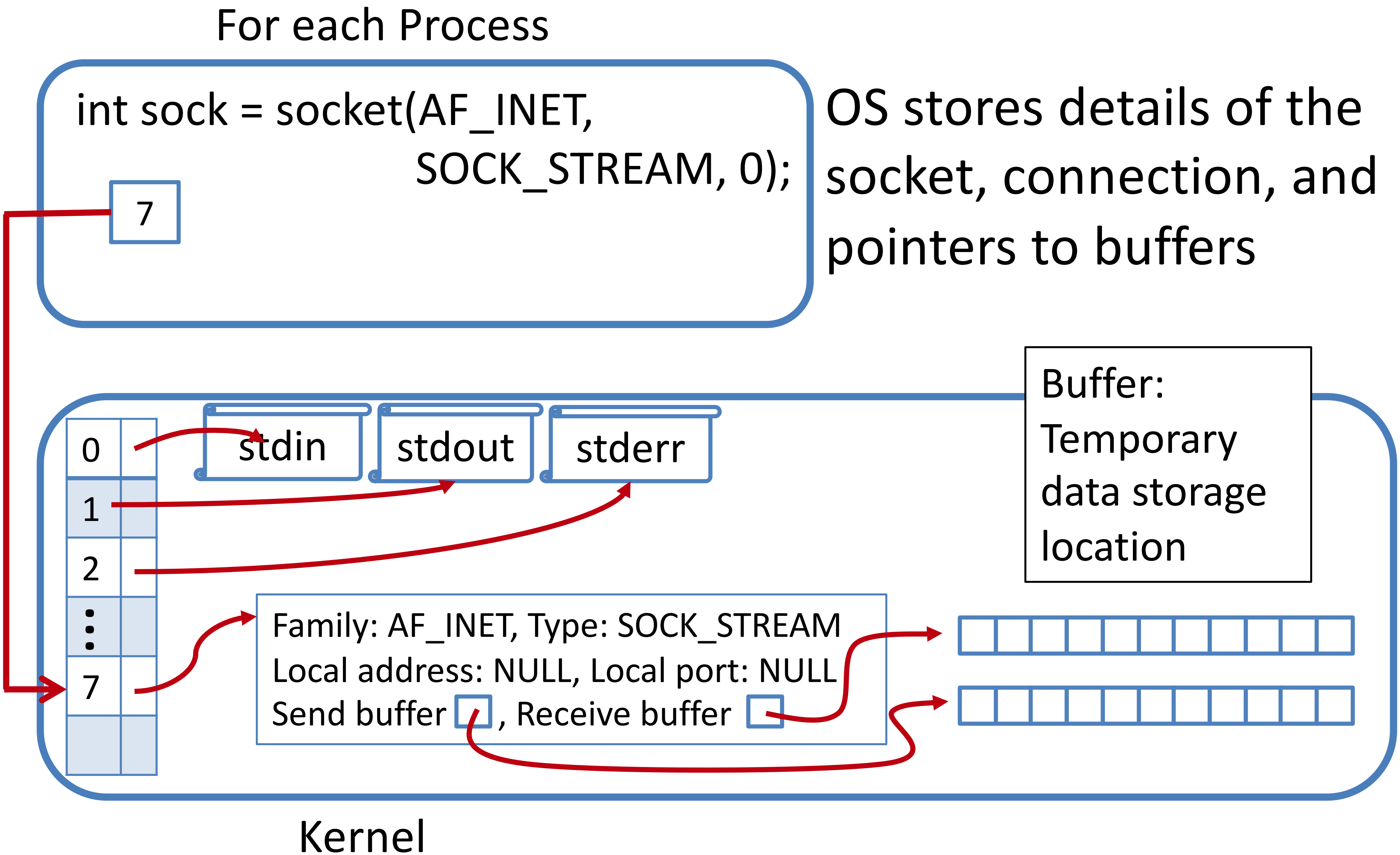
socket()



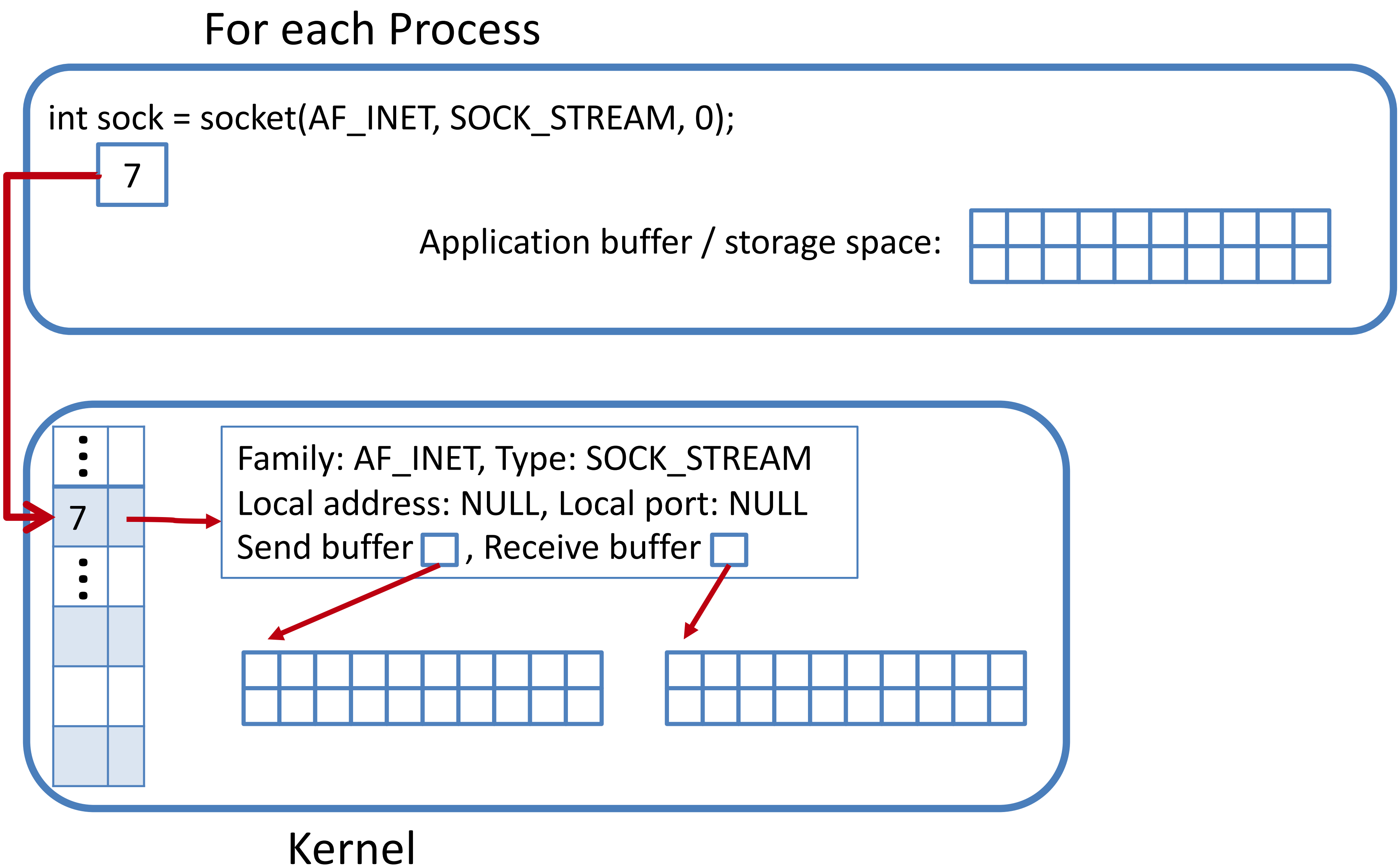
OS stores details of the socket, connection, and pointers to buffers

socket()

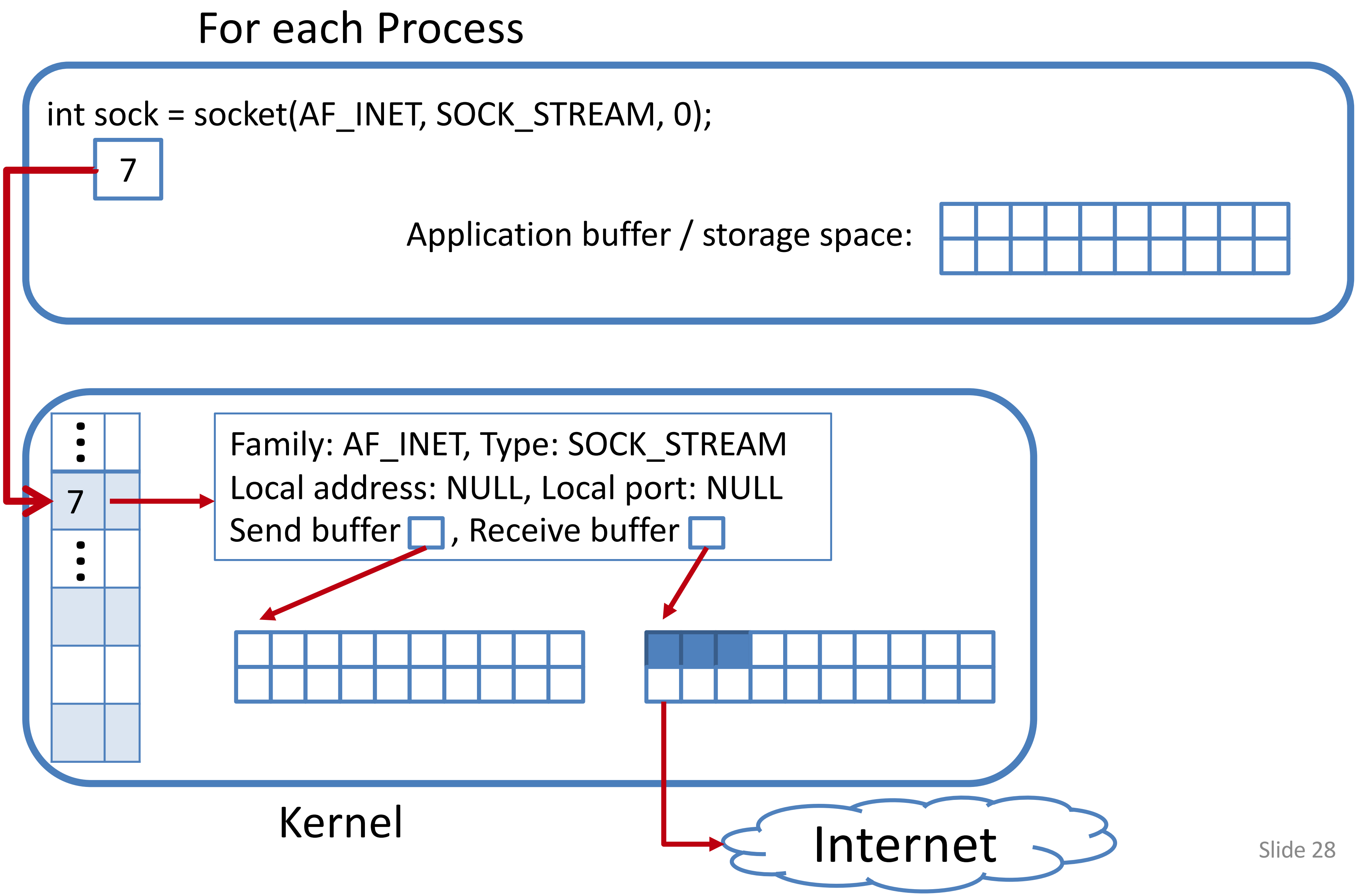
OS stores details of the socket, connection, and pointers to buffers



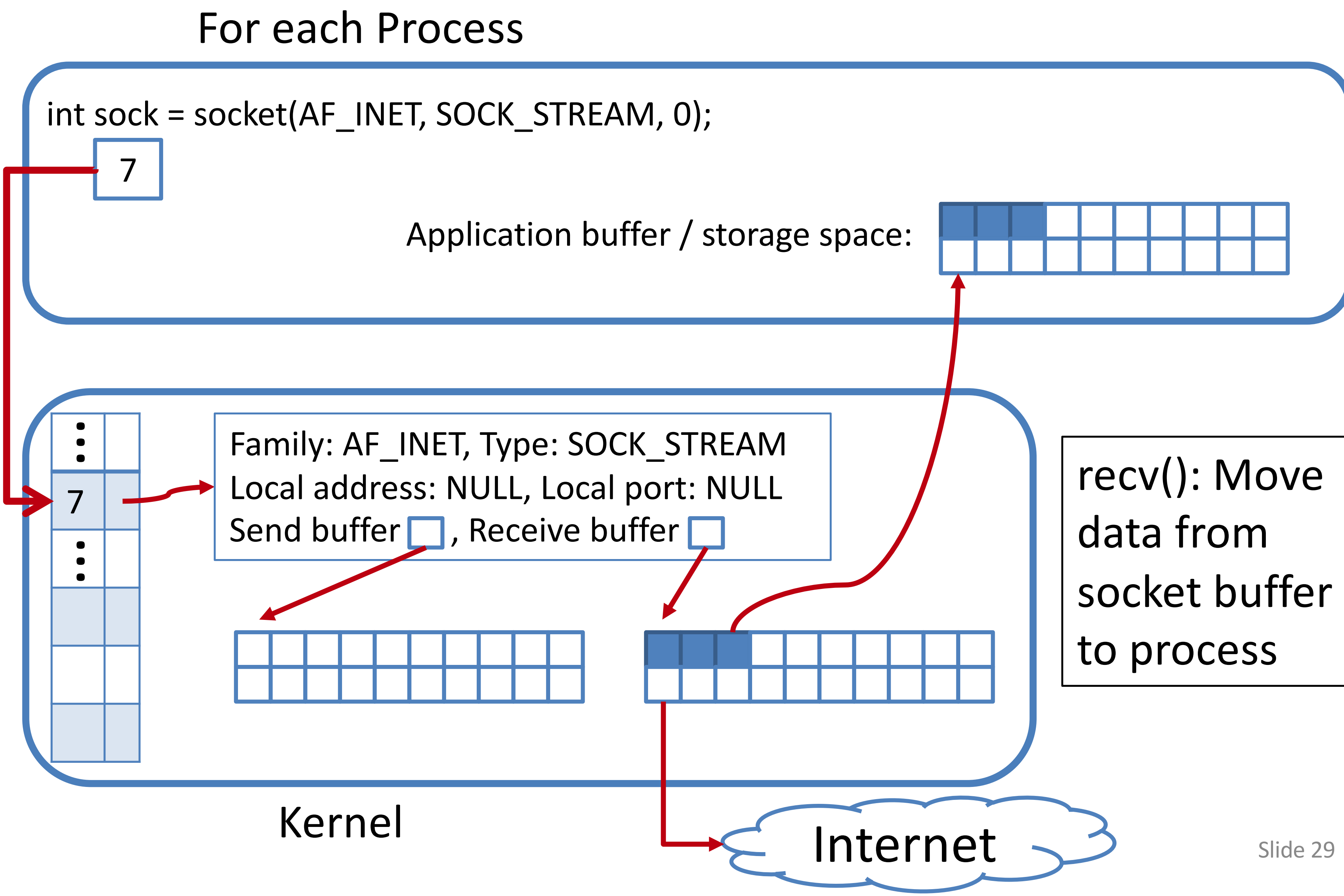
Socket Buffers



Socket Buffers

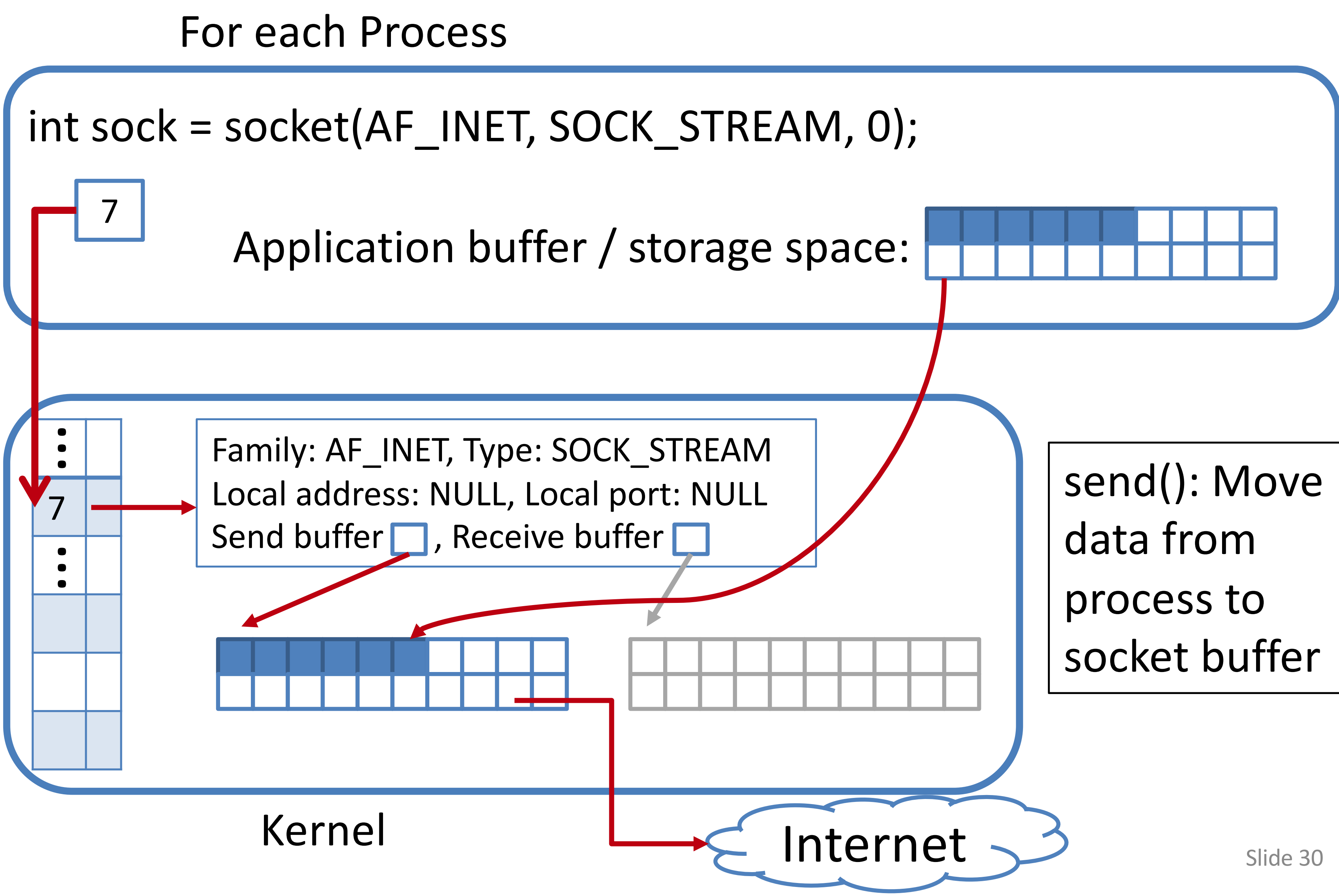


Socket Buffers



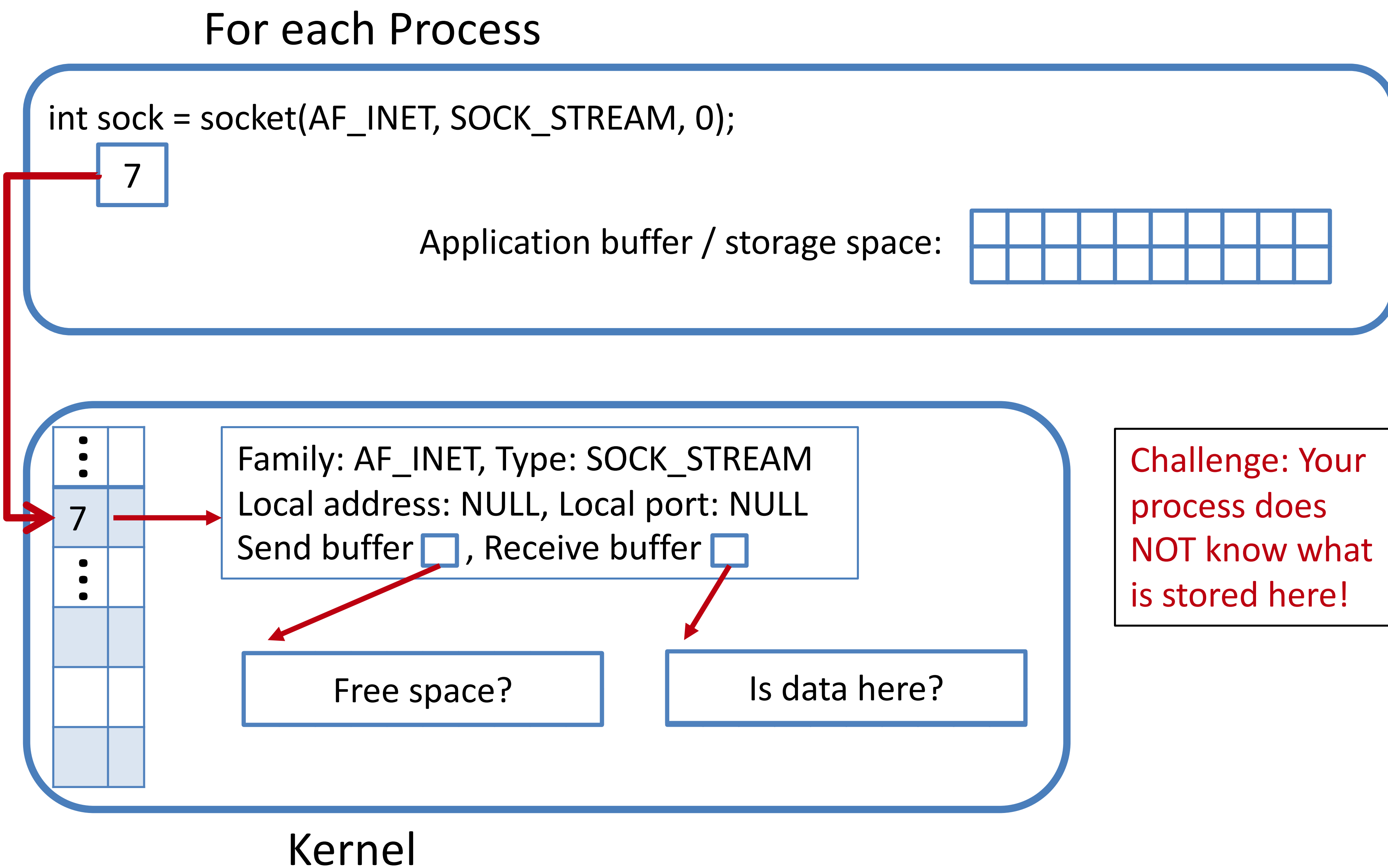
recv(): Move data from socket buffer to process

Socket Buffers



send(): Move data from process to socket buffer

Socket Buffers



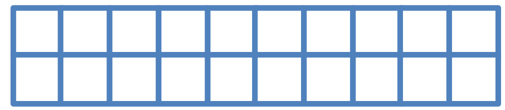
Challenge: Your process does NOT know what is stored here!

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)



0	
1	
2	
⋮	
7	

Family: AF_INET, Type: SOCK_STREAM

Local address: …, Local port: …

Send buffer ☐ , Receive buffer ☐

Is data here?

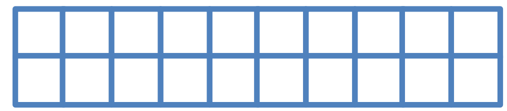
Kernel

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

For each Process

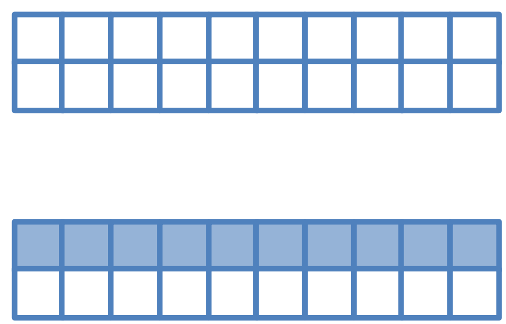
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

Socket buffer



Empty

100 bytes

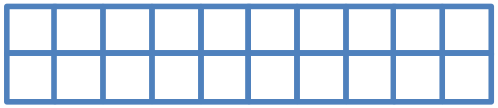
Kernel

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

For each Process

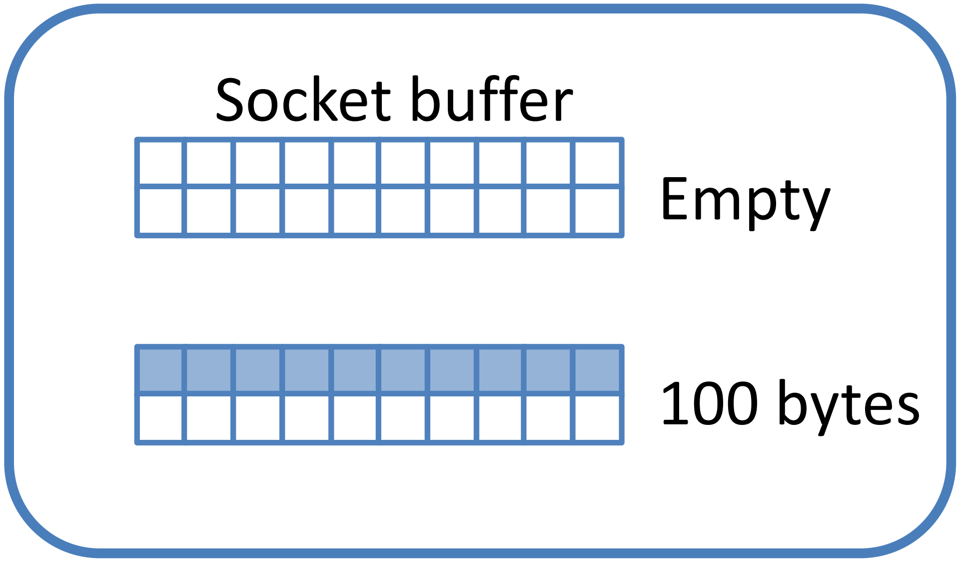
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

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?

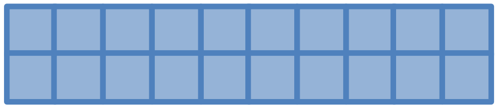
For each Process

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

(assume we connect()ed here…)

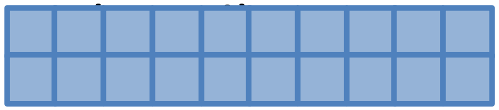
int recv_val = recv(sock, r_buf, 200, 0);

s_buf (size 200)

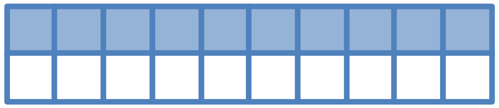


Two Scenarios:

Socket buffer



Full



100 bytes

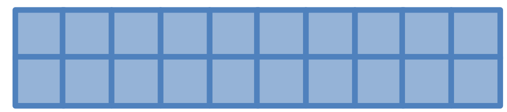
Kernel

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);
```

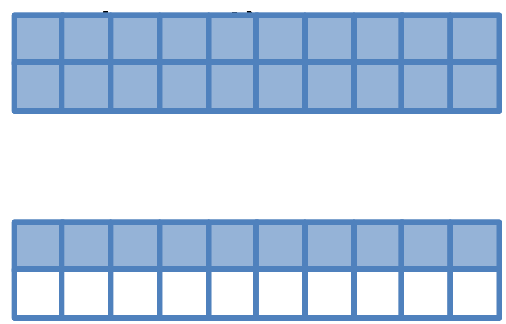
s_buf (size 200)



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

send()

- 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!*

recv()

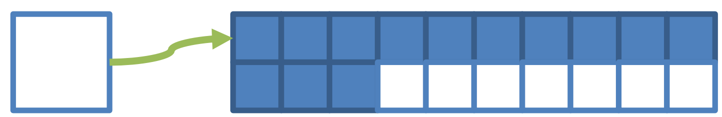
- Do not assume that you will send() all of the data you ask the kernel to copy.
- Keep track of where you are in the data you want to send.
- Always send in a loop!*

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

ALWAYS check send() return value!

- When send() return value is less than the data size, **you are responsible for sending the rest**.

Data sent: 0
Data to send: 130

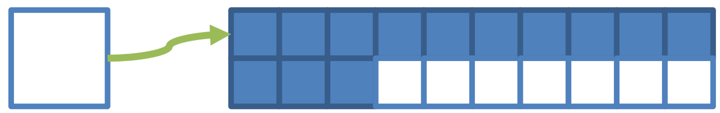
Data: 

send(sock, data, 130, 0);

ALWAYS check send() return value!

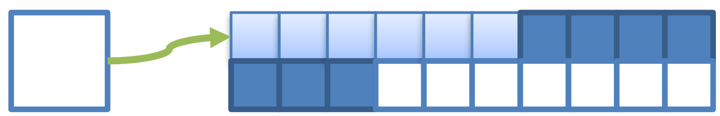
- When send() return value is less than the data size, **you are responsible for sending the rest.**

Data sent: 0
Data to send: 130

Data: 

60 send(sock, data, 130, 0);

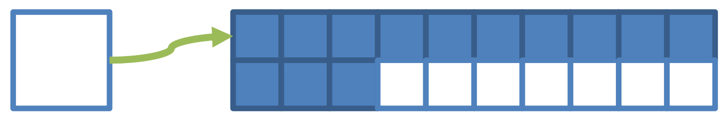
Data sent: 60
Data to send: 130

Data: 

ALWAYS check send() return value!

- When send() return value is less than the data size, **you are responsible for sending the rest.**

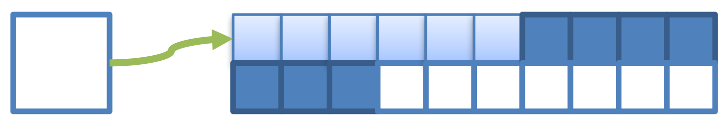
Data sent: 0
Data to send: 130

Data: 

60

send(sock, data, 130, 0);

Data sent: 60
Data to send: 130

Data: 

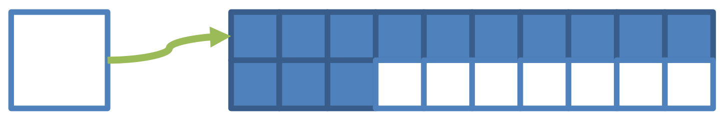
// Copy the 70 bytes starting from offset 60.
send(sock, data + 60, 130 - 60, 0);

ALWAYS check send() return value!

- When send() return value is less than the data size, **you are responsible for sending the rest.**

Data sent: 0

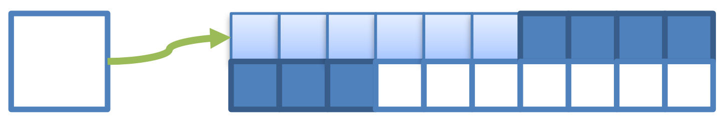
Data to send: 130

Data: 

60 send(sock, data, 130, 0);

Data sent: 60

Data to send: 130

Data: 

? // Copy the 70 bytes starting from offset 60.

send(sock, data + 60, 130 - 60, 0);

Repeat until all bytes are sent. (data_sent == data_to_send)…

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!

Blocking Summary

send()

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

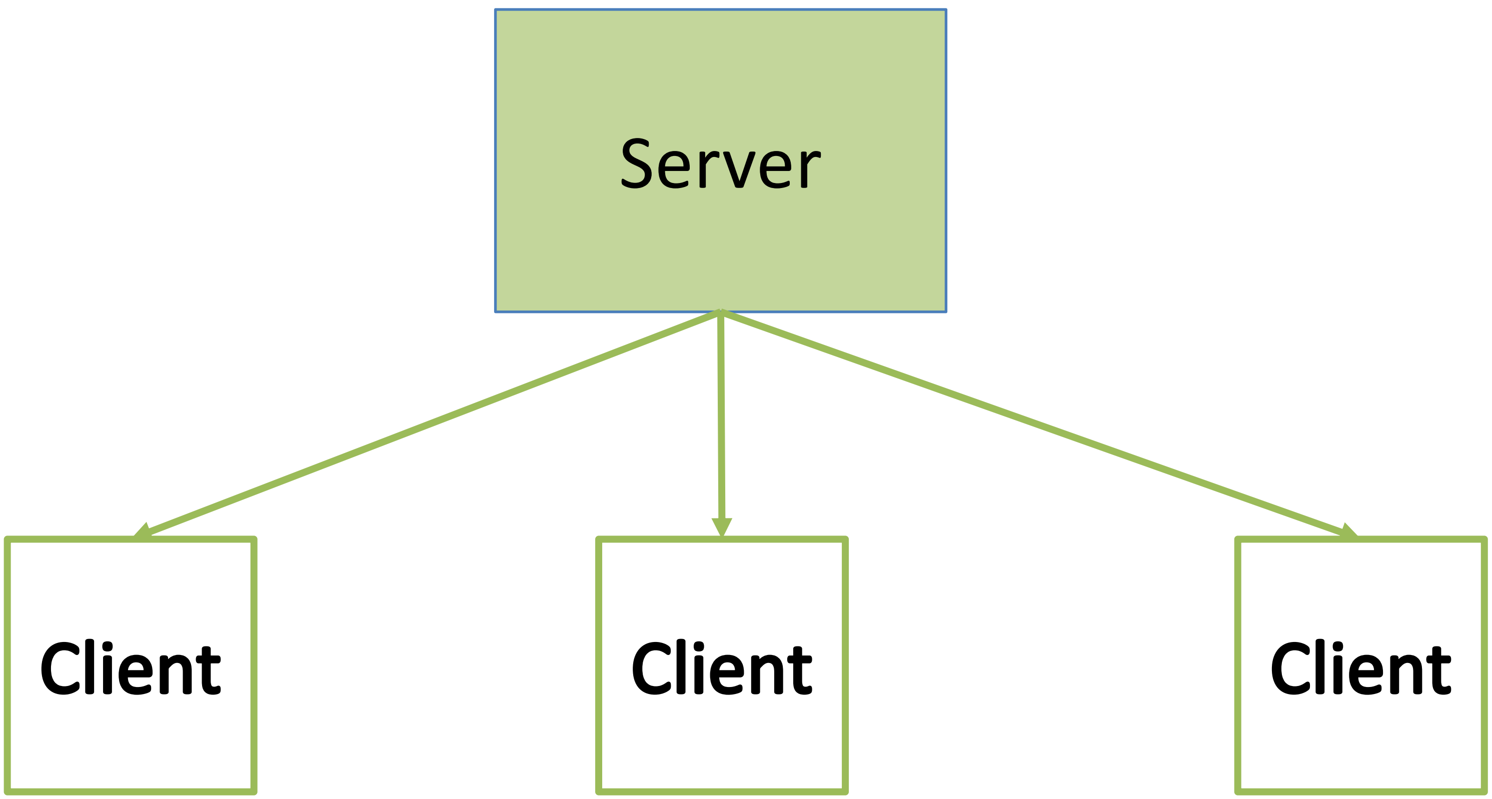
recv()

- 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!

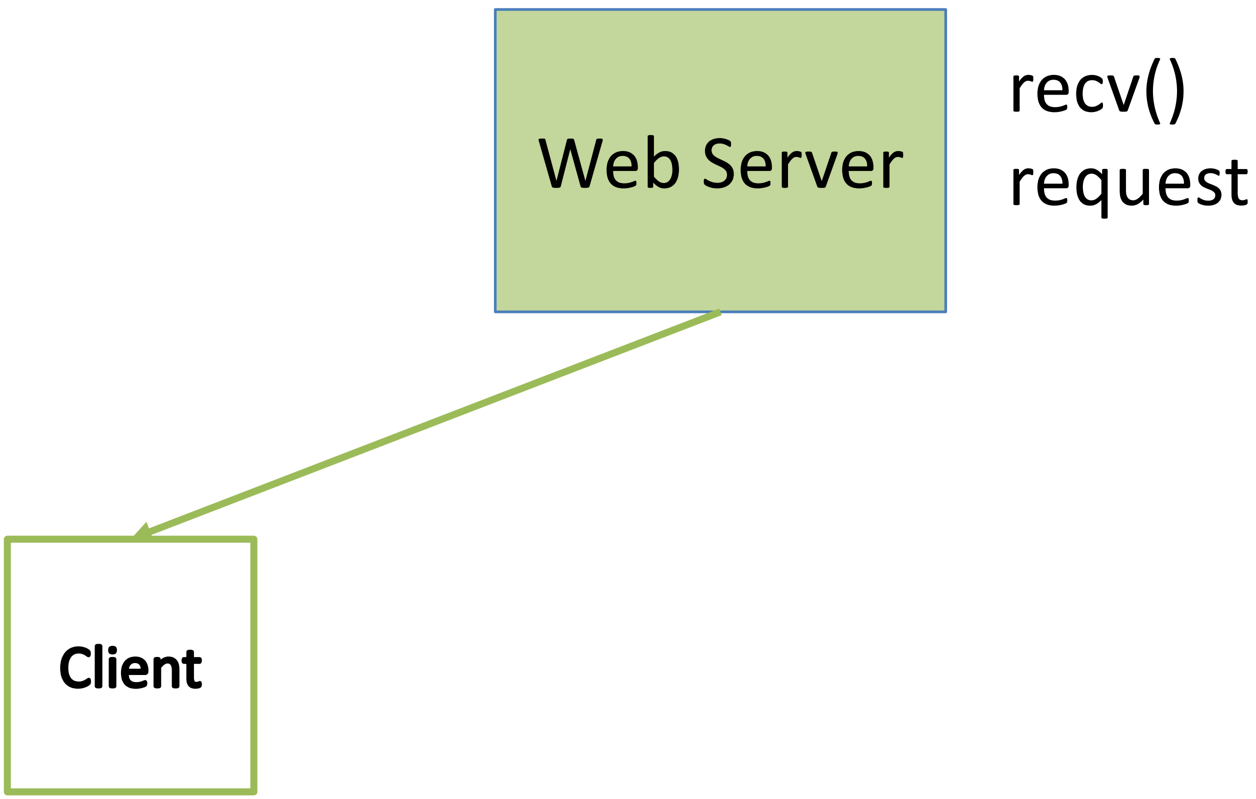
Concurrency

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



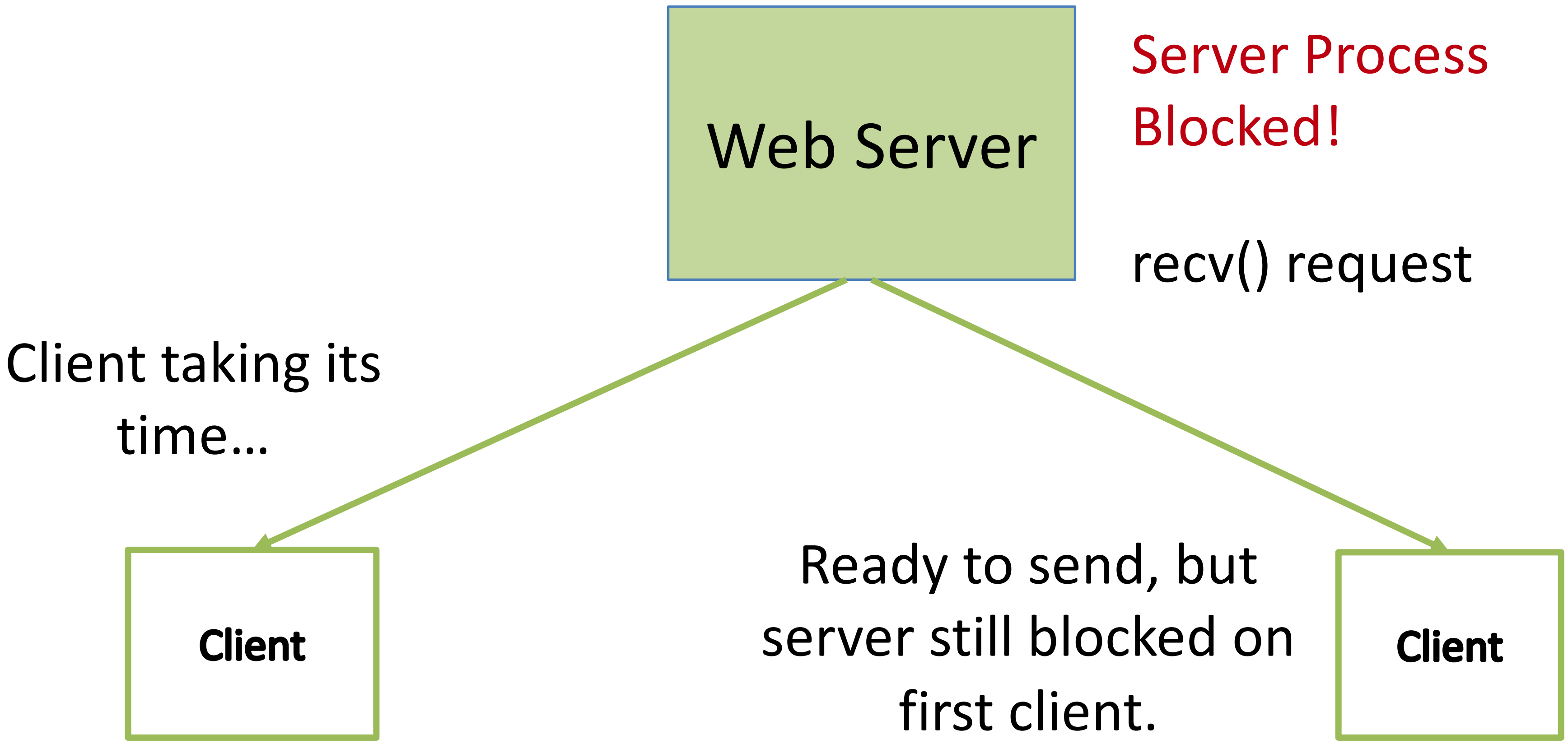
Without Concurrency

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



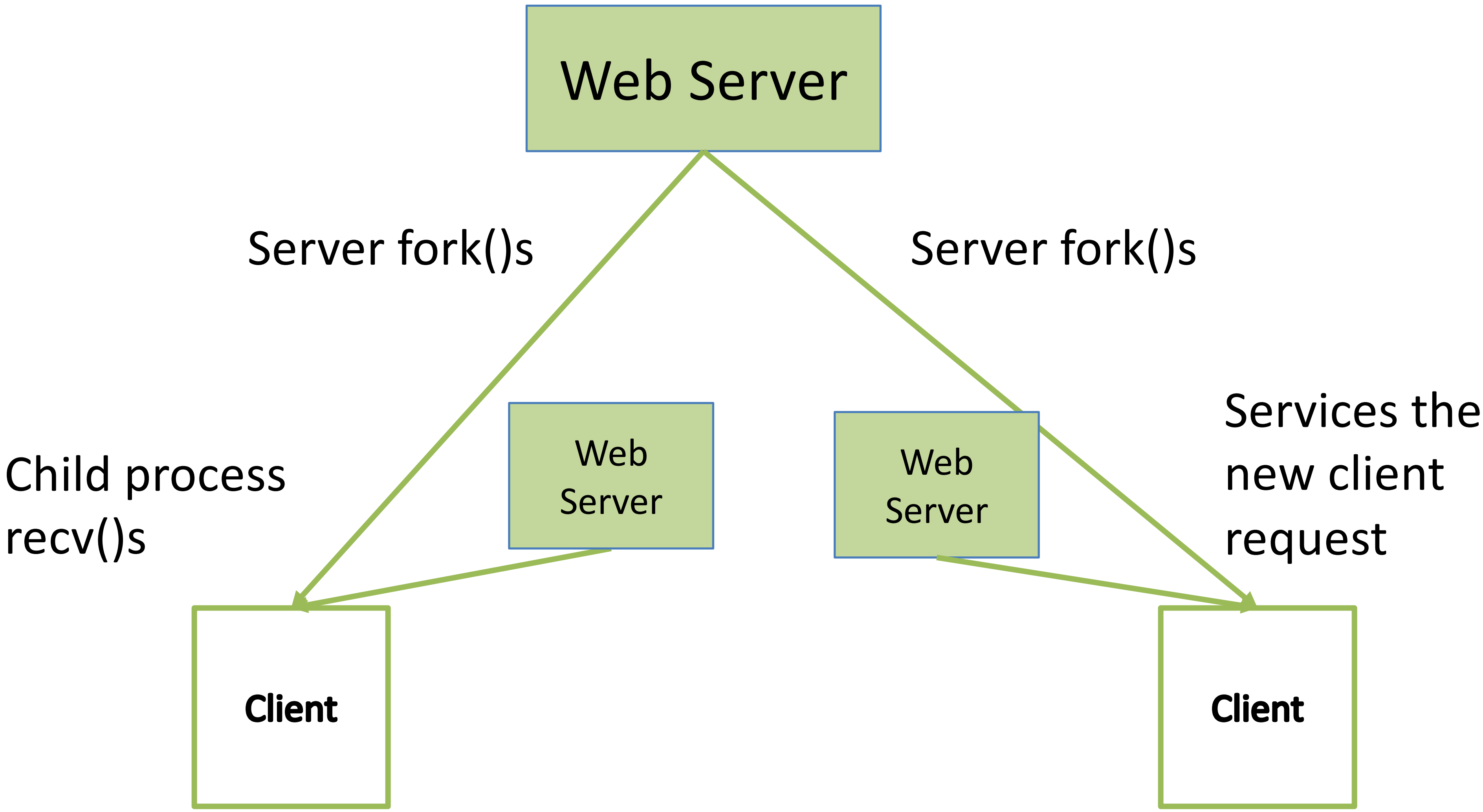
Without Concurrency

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



If only we could handle these connections separately…

Multiple processes



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
  - 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

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

Threads & Sharing

- Global variables and static objects are shared
  - Stored in the static data segment, accessible by any thread
- Dynamic objects and other heap objects are shared
  - Allocated from heap with malloc/free or new/delete
- Local variables are not shared
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack

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

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
- Still not maximum efficiency…
  - Creating/destroying threads still takes time
  - Requires memory to store thread execution state
  - Lots of context switching overhead

Non-blocking I/O

- One operation: add a flag to send/recv
- Permanently, for socket: fcntl() – “file control”
  - Allows setting options on file/socket descriptors

```
int sock, result, flags = 0;
sock = socket(AF_INET, SOCK_STREAM, 0);
result = fcntl(sock, F_SETFL, flags|O_NONBLOCK)
```

check result – 0 on success

Non-blocking I/O

- With O_NONBLOCK set on a socket
  – No operations will block!

- On recv(), if socket buffer is empty:
  – returns -1, errno is EAGAIN or EWOULDBLOCK

- On send(), if socket buffer is full:
  – returns -1, errno is EAGAIN or EWOULDBLOCK

Will this work?

```
server_socket = socket(), bind(), listen() //non-blocking
connections = []
while (1)
  new_connection = accept(server_socket)
  if new_connection != -1,
     add it to connections
  for connection in connections:
     recv(connection, …)  // Try to receive
     send(connection, …) // Try to send, if needed
```

Will this work?

A. Yes, this will work.
B. No, this will execute too slowly.
C. No, this will use too many resources.
D. No, this will still block.

```
server_socket = socket(), bind(), listen() //non-blocking
connections = []
while (1)
  new_connection = accept(server_socket)
  if new_connection != -1,
    add it to connections
  for connection in connections:
    recv(connection, …)  // Try to receive
    send(connection, …) // Try to send, if needed
```

Event-based concurrency: select()

- Rather than checking over and over, let the OS tell us when data can be read/written
- Create set of file/socket descriptors we want to read and write
- Tell system to block until at least one of those is ready for us to use. The OS worries about selecting which one(s).

Event-based concurrency

- Rather than checking over and over, let the OS tell us when data can be read/written
- Tell system to block until at least one of those is ready for us to use. The OS worries about selecting which one(s).
- Only one process/thread (or one per core)
  - No time wasted on context switching
  - No memory overhead for many processes/threads

Next class

- Network and Distributed Systems
- Client-Server Architecture
- Peer-to-Peer Architecture