CS 43: Computer Networks

04: Sockets

September 14, 2026

Slides adapted from Kurose & Ross, Kevin Webb, Vasanta Chaganti

Announcements

- Grades for Quiz 1 will be out Tuesday
- Quiz 2 tomorrow, study guide posted

Reading Quiz
(graded for correctness)

Put devices away

Note: Reading quiz slides will have an orange border.

End of Reading Quiz!

Today: Sockets

 Why do you have to call send() in a loop?

 Why do you have to call recv() in a loop?

But first: some HTTP things I want to explain again…

From last time: Persistent connections improve HTTP/1.0 Performance

Non-persistent HTTP: response time

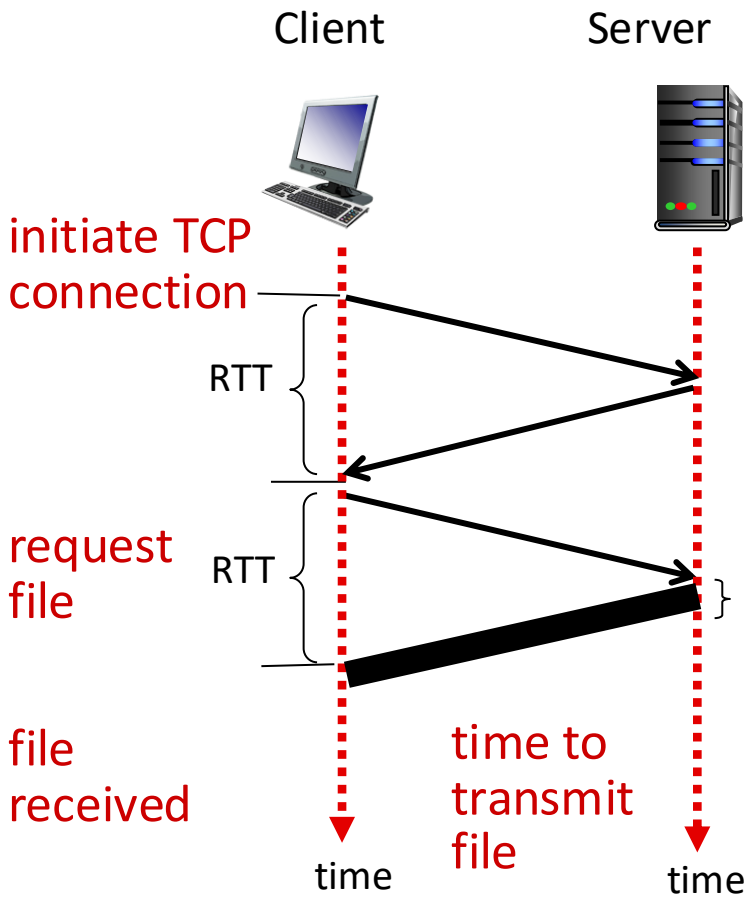
Round Trip Time (RTT): time for a small packet to travel from client to server and back

HTTP response time:

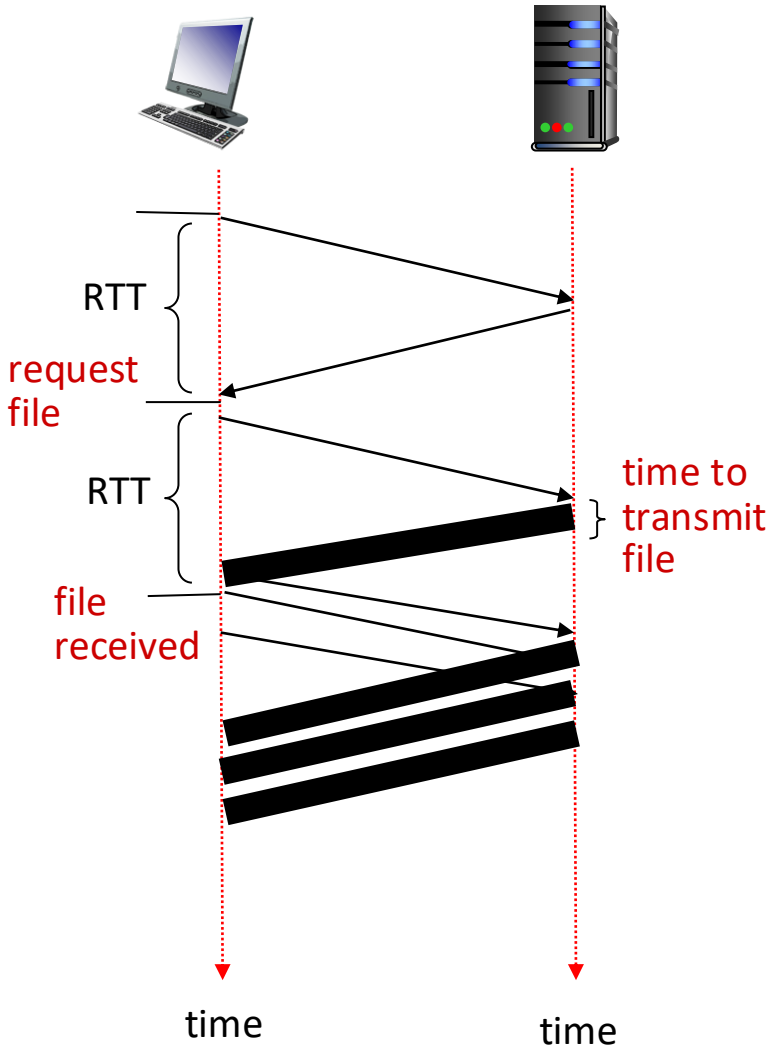
- 1-RTT to initiate TCP connection
- 1-RTT for HTTP request + first few bytes of HTTP response to return
- file transmission time
- non-persistent HTTP response time =

  2-RTT+ file transmission time

  For each object



Persistent Connection

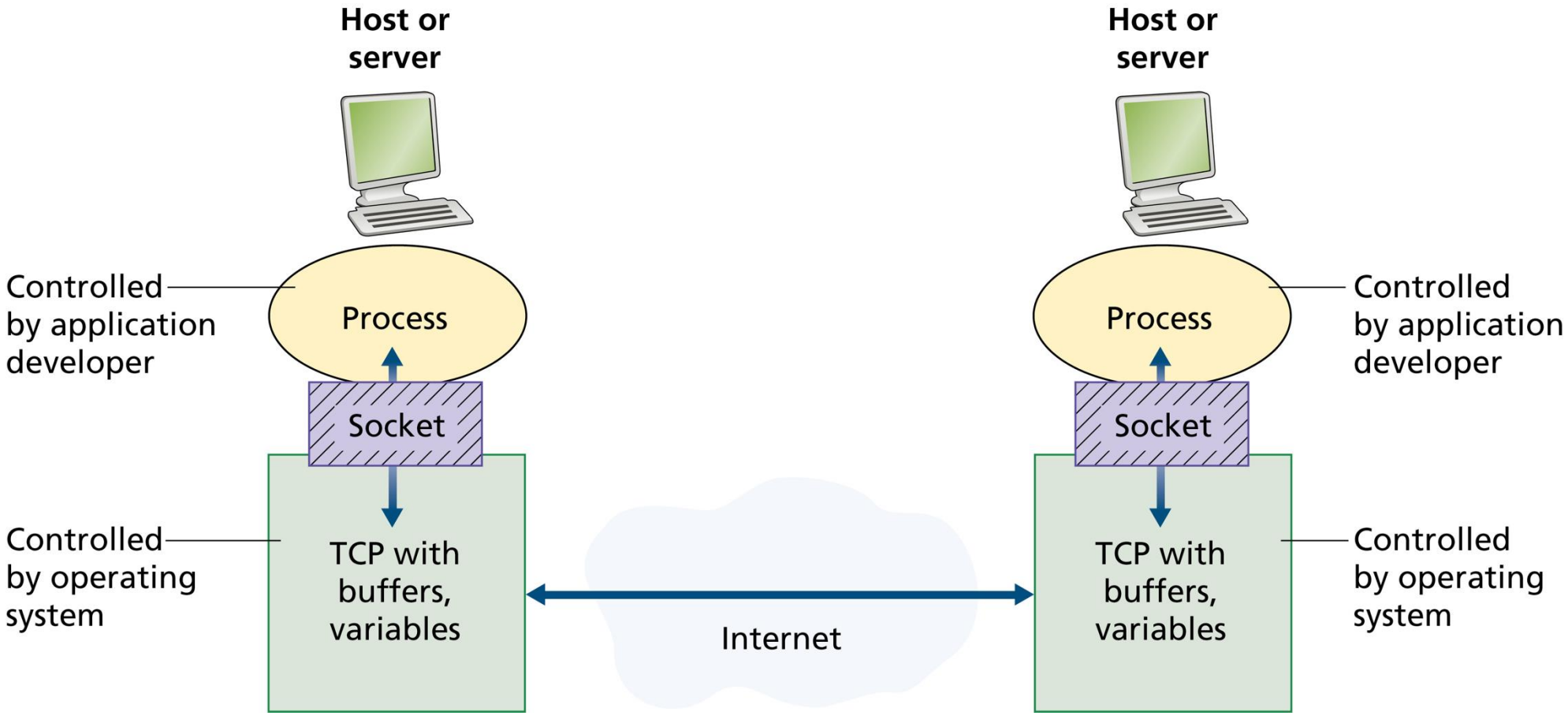


A web page consists of a base HTML file and 9 referenced objects, all on the same server. The client uses a single persistent connection and sends the nine requests back to back as soon as the base file arrives. R is the RTT, T is the transmission time of one object. Counting TCP connection setup and ignoring all other delays, what is the total time from the client's first action until the last object has arrived?

A. 2R + 10T
B. 3R + 10T
C. 11R + 10T
D. 20R + 10T
E. I don't know

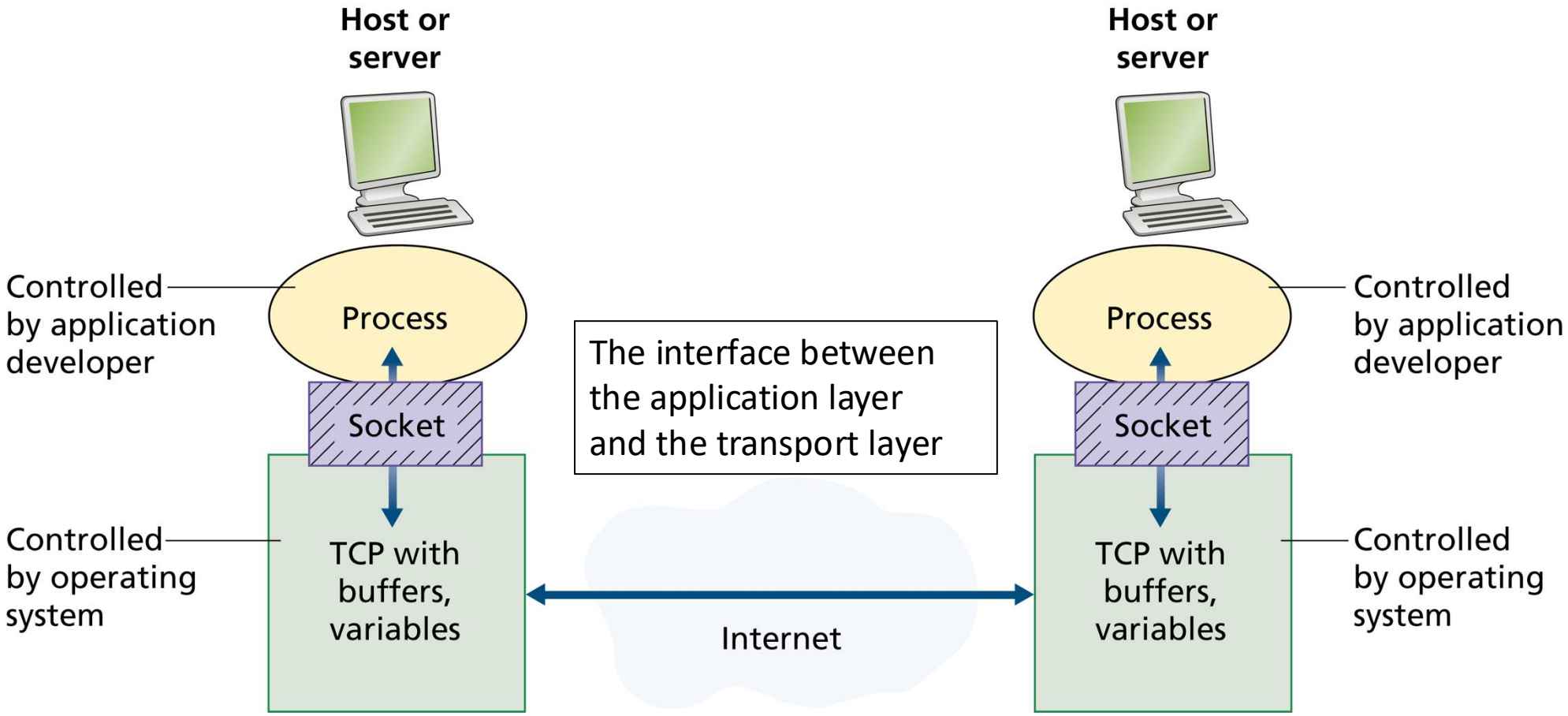
A socket is an abstraction through which applications may send and receive data in the same way as an open-file handle allows an application to read and write data to storage.

.



A socket is an abstraction through which applications may send and receive data in the same way as an open-file handle allows an application to read and write data to storage.

.



Why learn sockets in C?

- We learn how to manage all the low-level stuff like OS system calls and file descriptors that other languages will abstract away for us.
- Many network internals are written in C. For example: Linux TCP/IP stack is in C

What did we learn about process communication in CS 31?

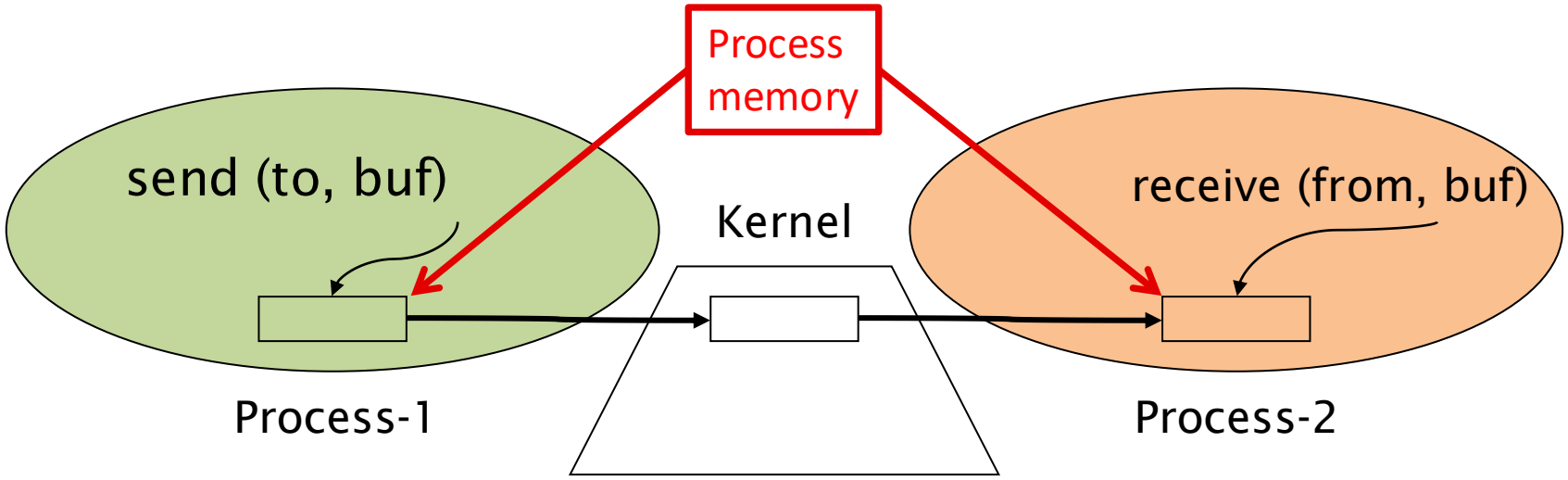
- Processes must communicate to cooperate
- Must have two mechanisms:
  - Data transfer
  - Synchronization
- On a single machine:
  - Signals
  - Threads (shared memory)

What did we learn about process communication in CS 31?

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

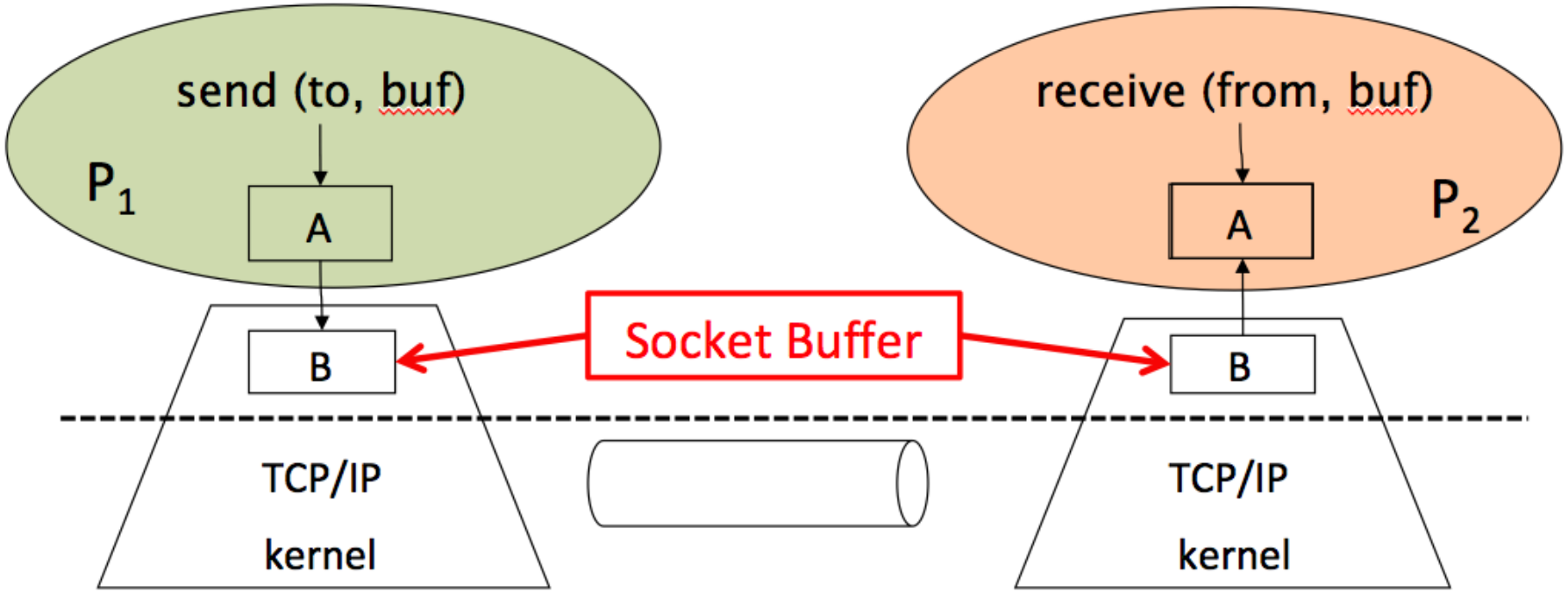
There's another way we didn't learn about!

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

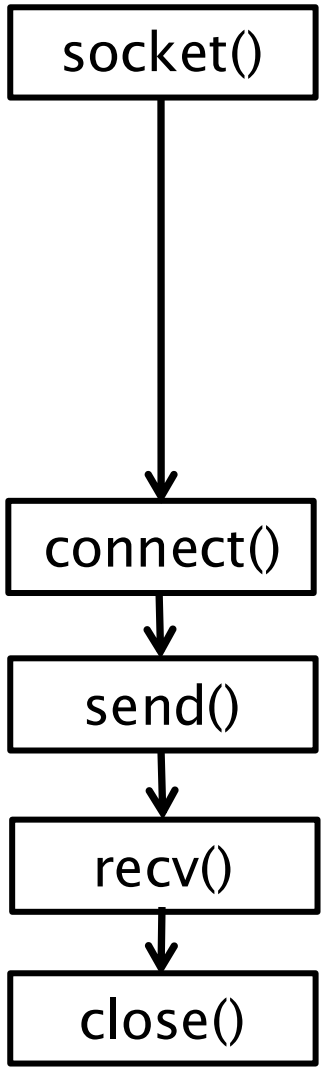
Message Passing (network)



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

TCP Socket Procedures: Client

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

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**().

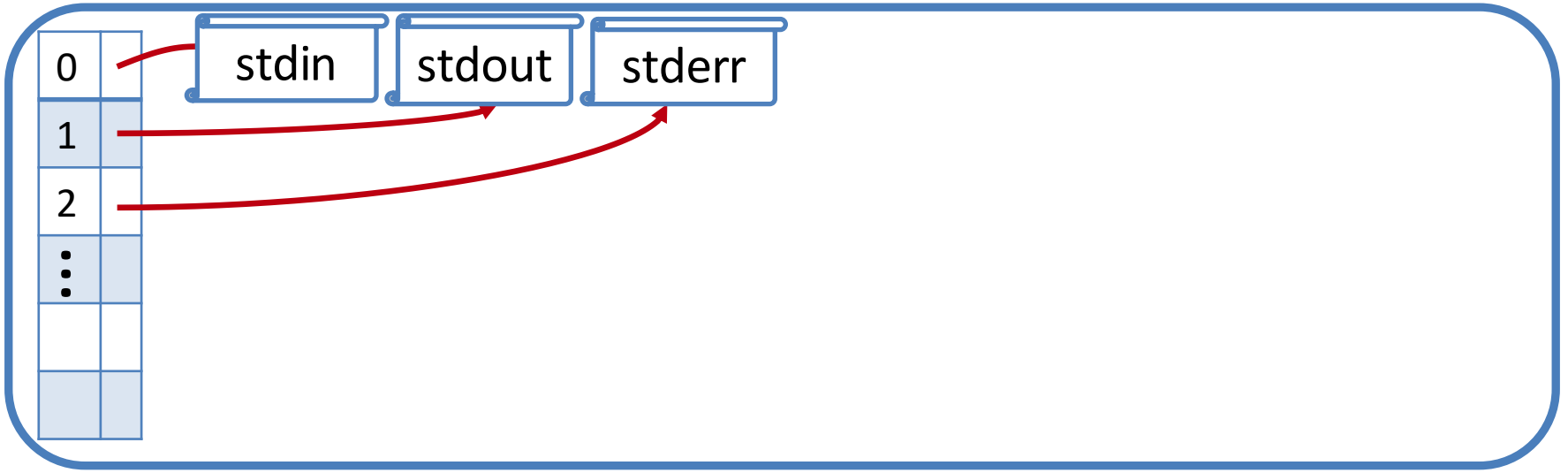
```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Descriptor Table

For each Process

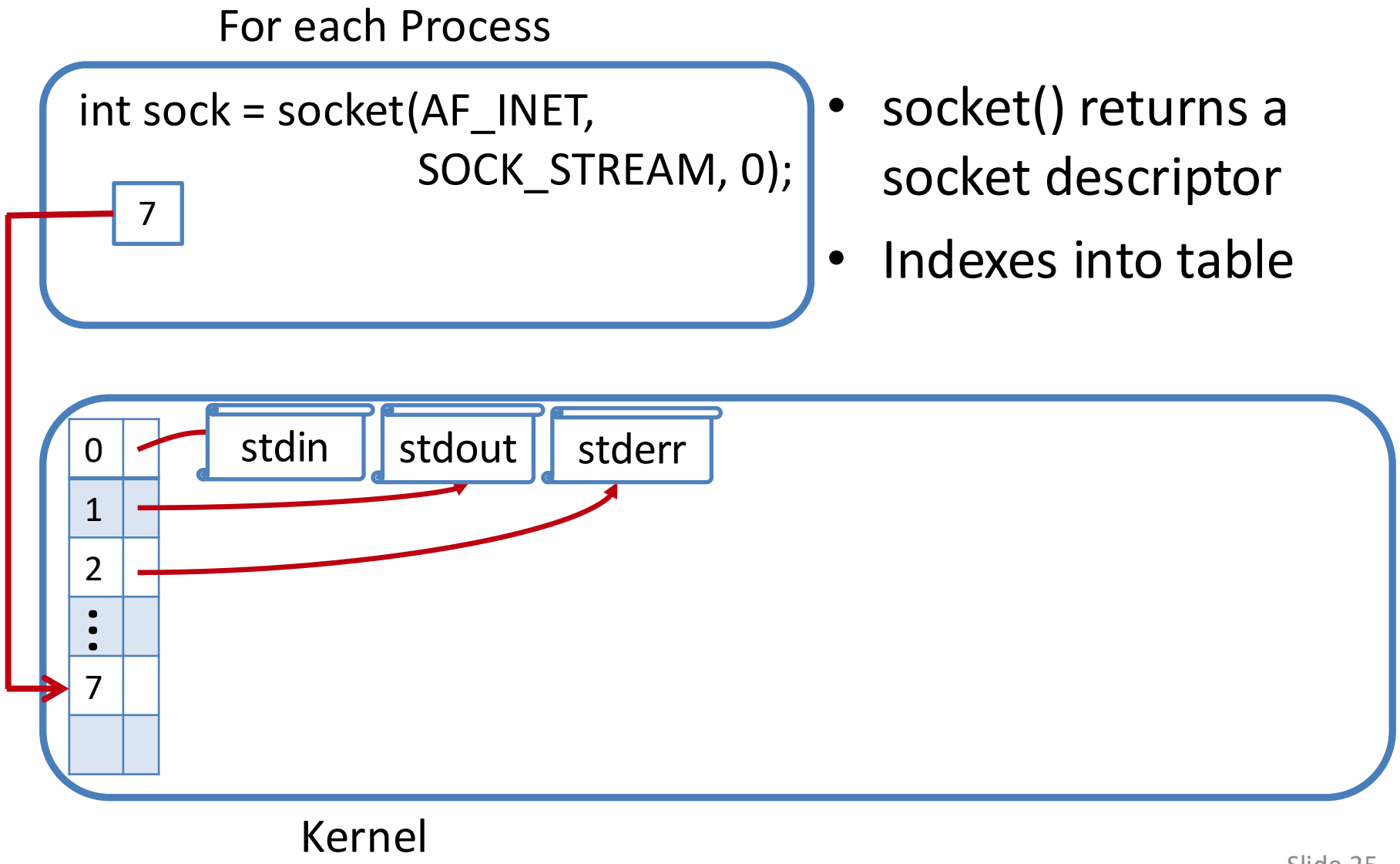
OS stores a table, per process, of descriptors

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



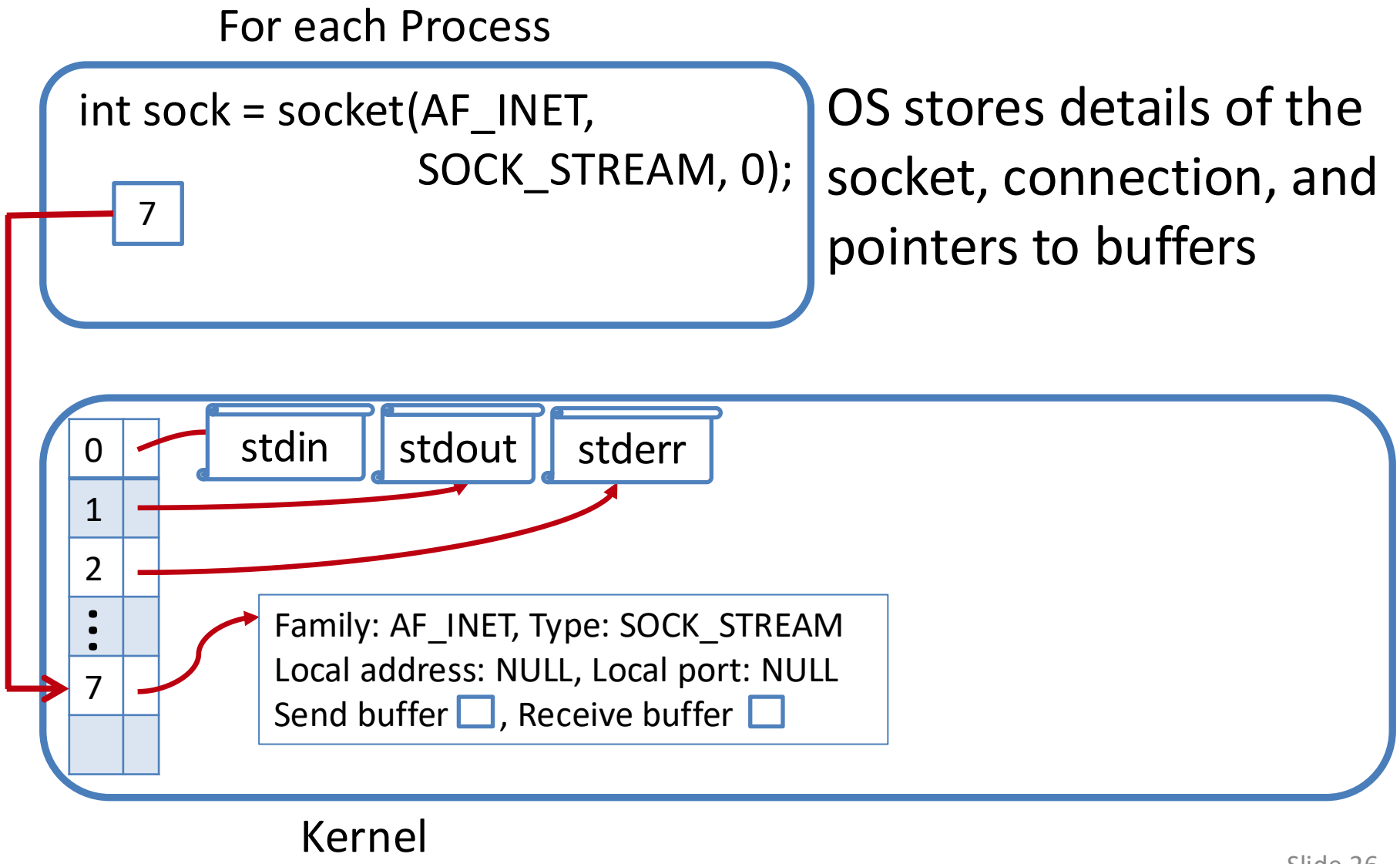
Kernel

socket()

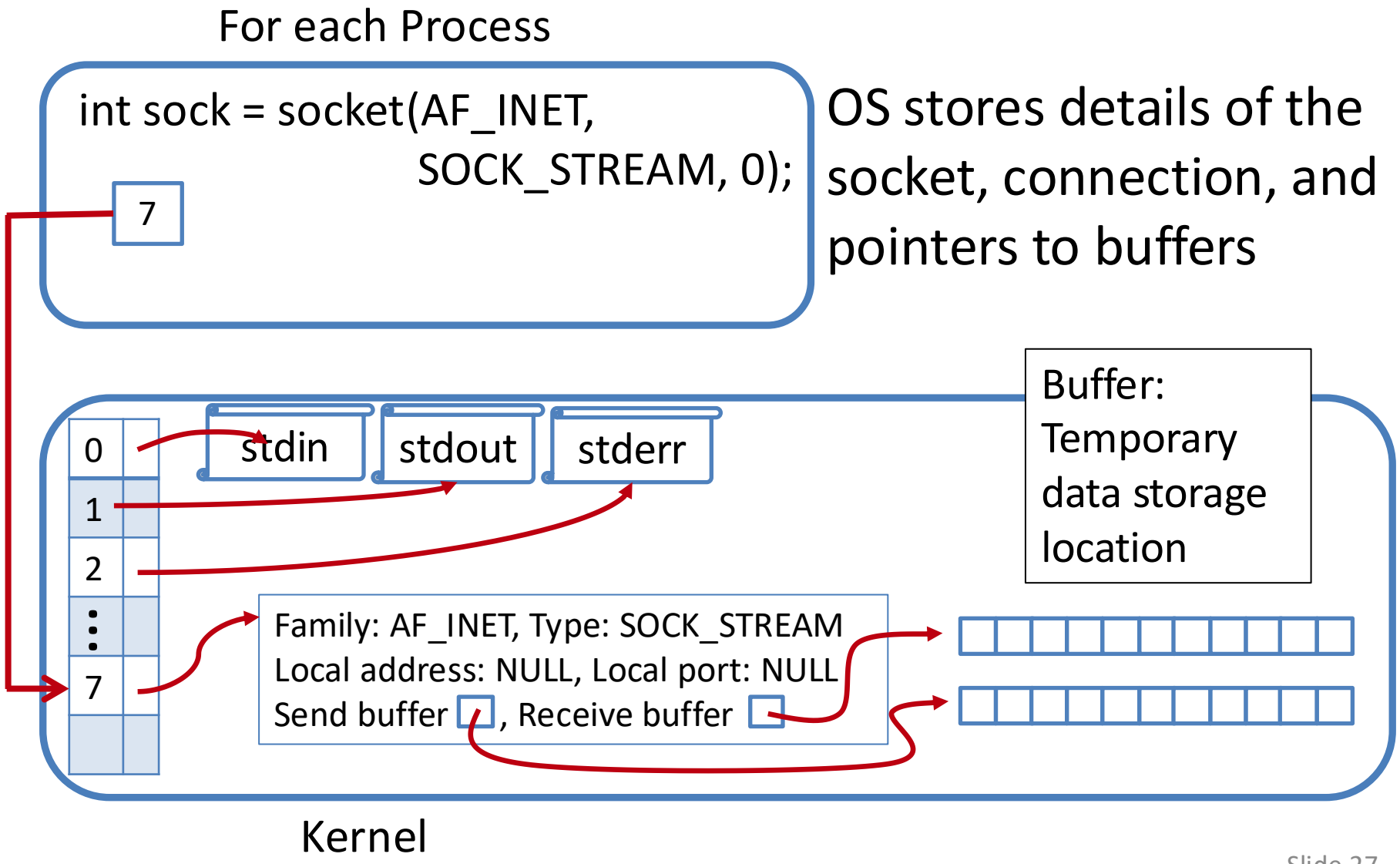


- socket() returns a socket descriptor
- Indexes into table

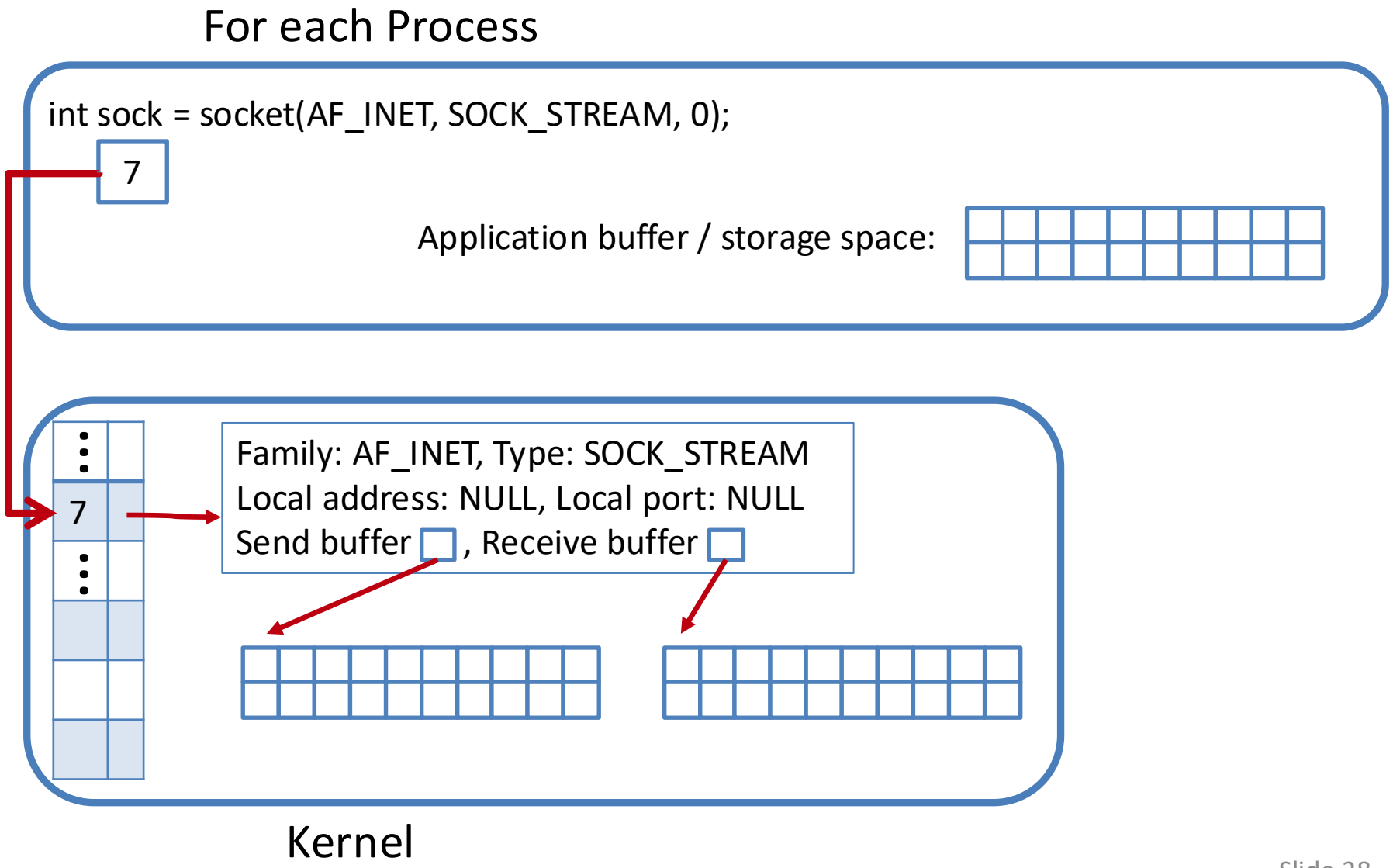
socket()


For each Process
int sock = socket(AF_INET,
SOCK_STREAM, 0);
7
OS stores details of the socket, connection, and pointers to buffers
0
1
2
7
stdin
stdout
stderr
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer
Kernel

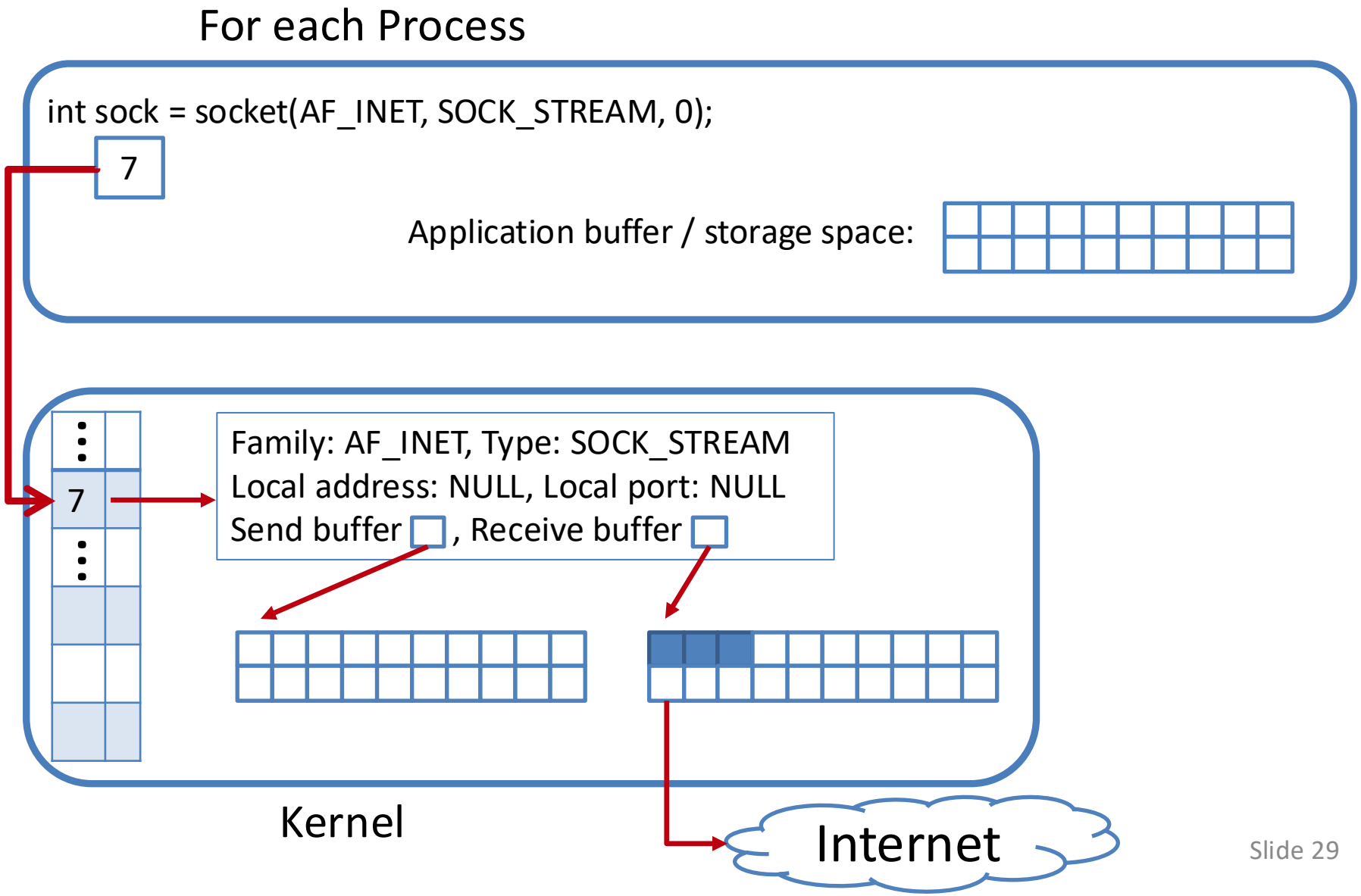
socket()


For each Process
int sock = socket(AF_INET,
SOCK_STREAM, 0);
7
OS stores details of the socket, connection, and pointers to buffers
Buffer: Temporary data storage location
0
1
2
7
stdin
stdout
stderr
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer
Kernel

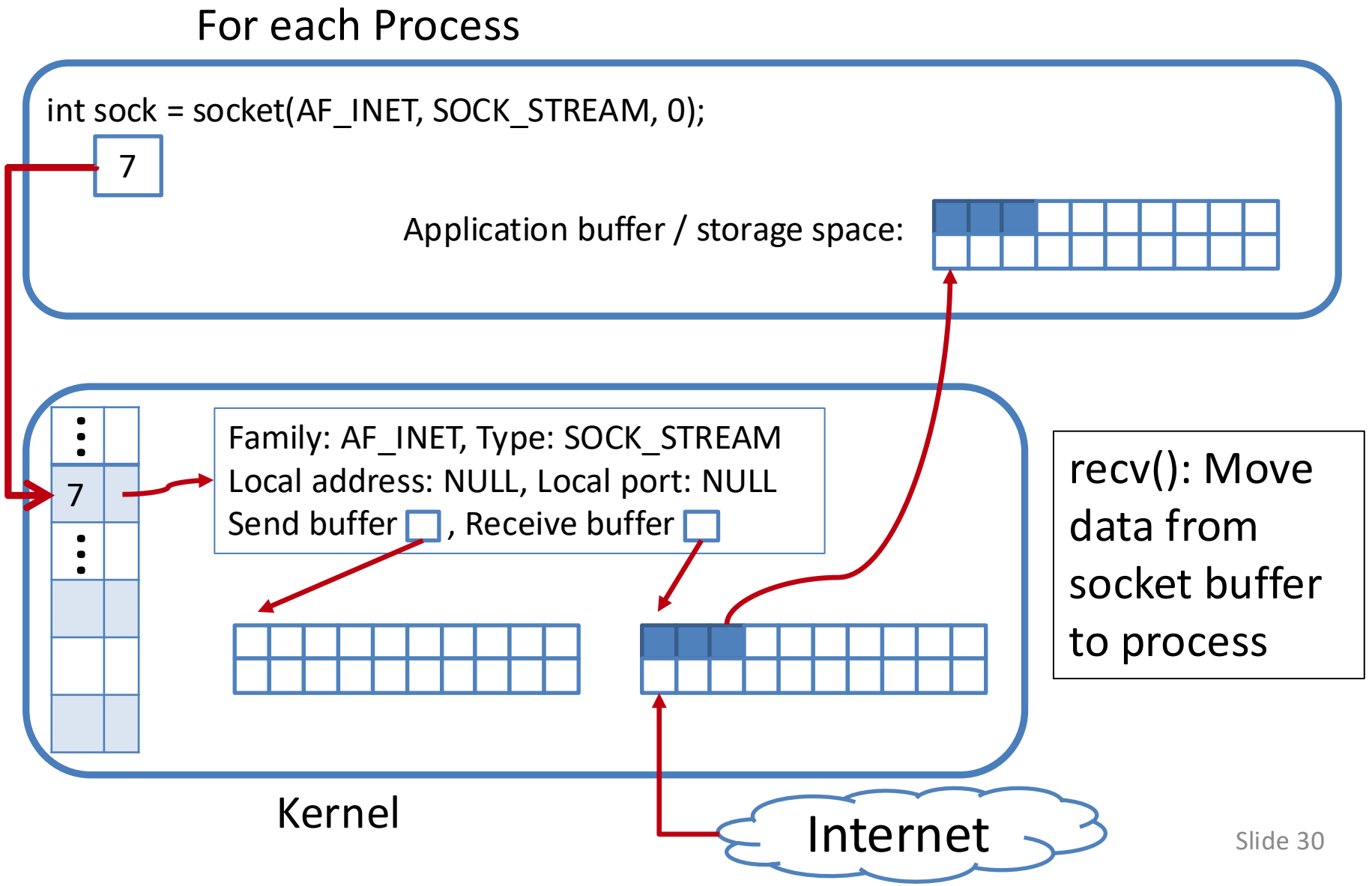
Socket Buffers



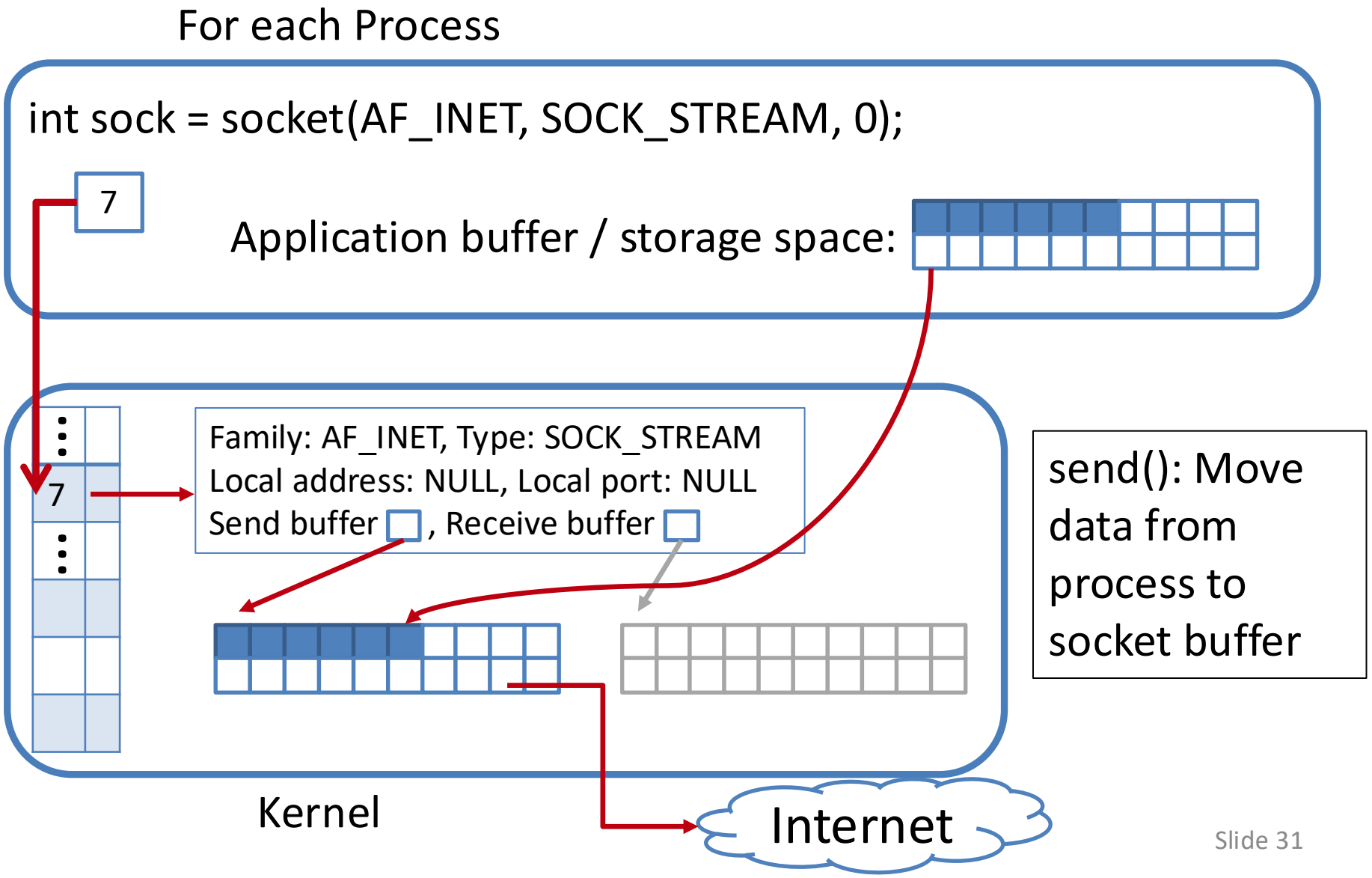
Socket Buffers



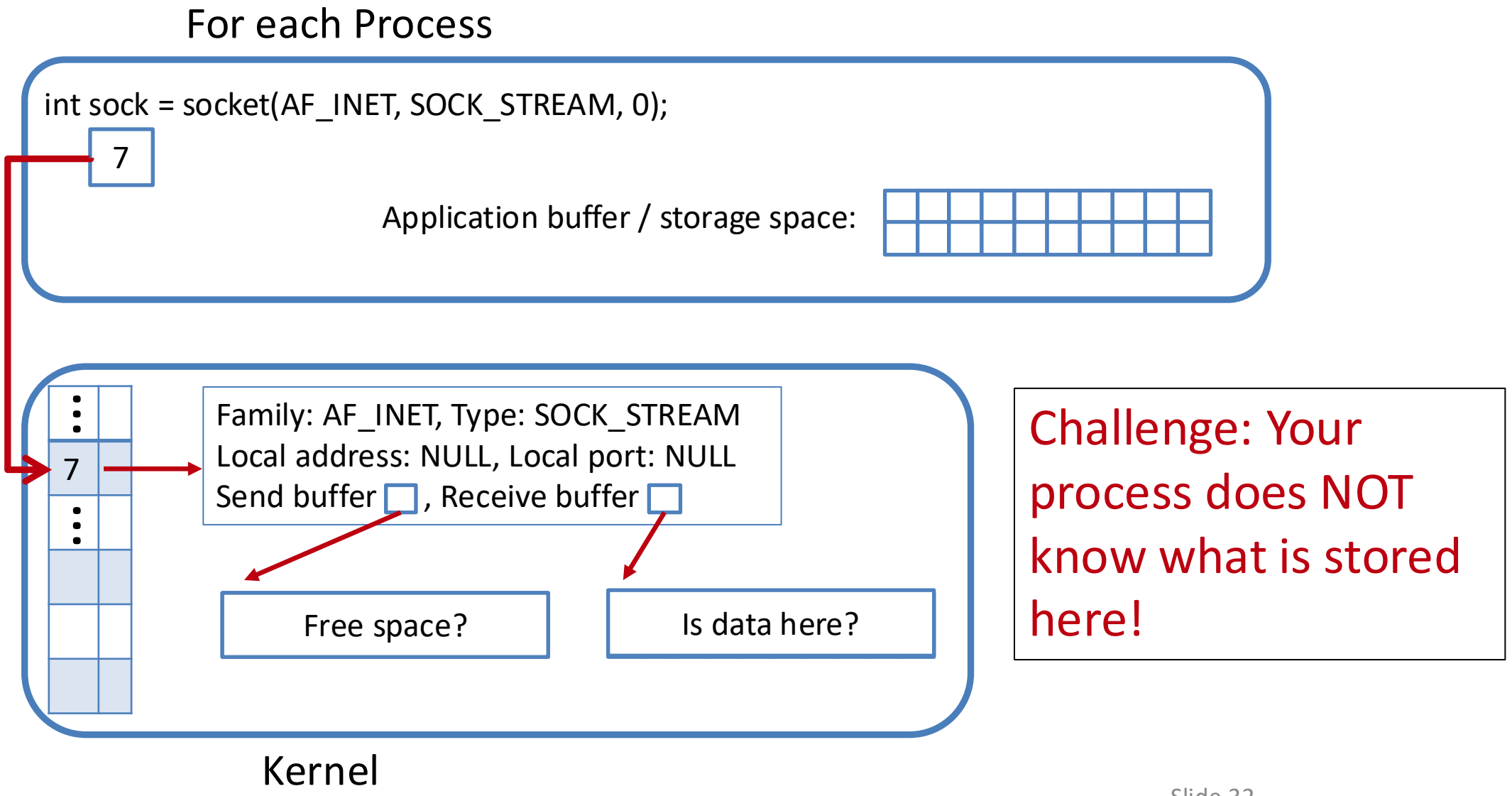
Socket Buffers



Socket Buffers



Socket Buffers



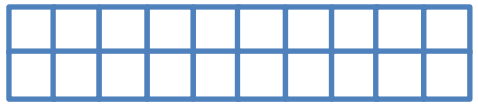
Challenge: Your process does NOT know what is stored here!

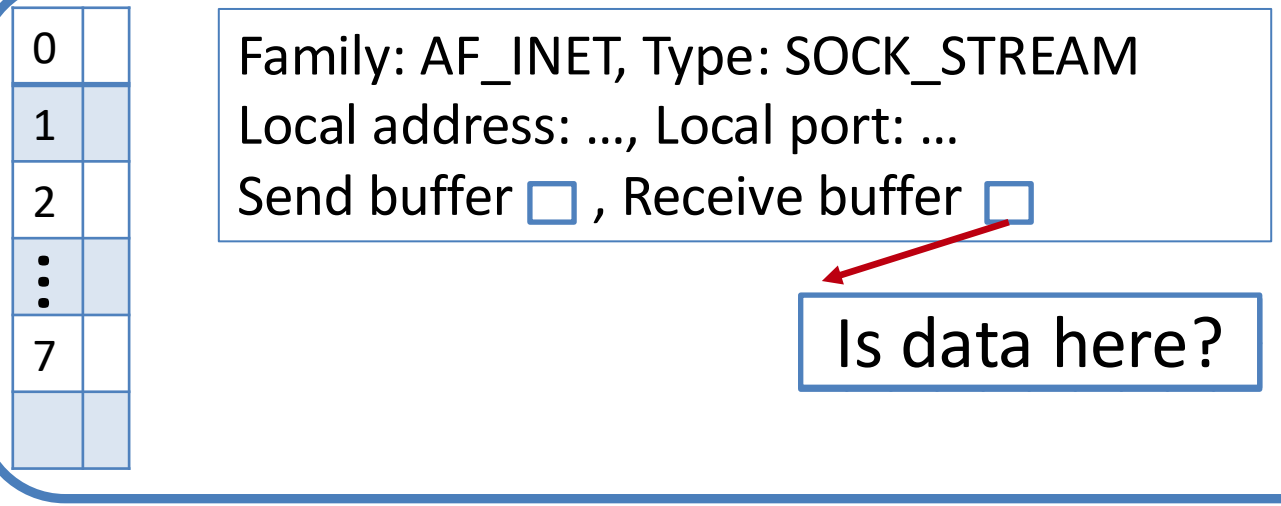
recv()

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 does the term “block” refer to in the context of socket programming?

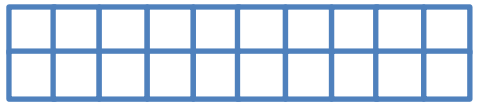
A. A form of encryption
B. A method of data transmission
C. A way of refusing unwanted connections
D. A function or process that waits (or “sleeps”) until it can proceed

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

For each Process

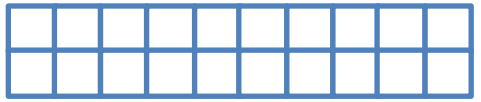
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

Socket buffer



Empty

100 bytes

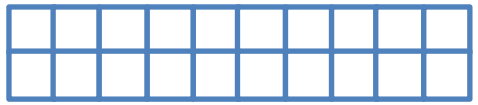
Kernel

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);
```

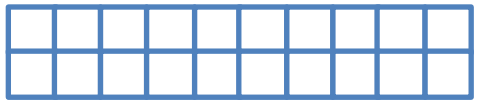
r_buf (size 200)



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	

Socket buffer



Empty

100 bytes

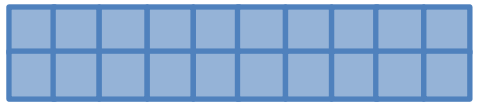
Kernel

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 send_val = send(sock, s_buf, 200, 0);
```

s_buf (size 200)



Two Scenarios:

Socket buffer

Full

100 bytes

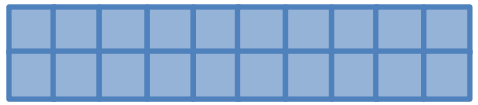
Kernel

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 send_val = send(sock, s_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

Sockets Worksheet

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.

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!