CS 43: Computer Networks

Fall 2025, Week 2

Substitute Instructor: Kevin Webb

SWARTHMORE COLLEGE

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

Five-Layer Internet Model

Application: the application (e.g., the Web, Email)
Transport: end-to-end connections, reliability
Network: routing
Link (data-link): framing, error detection
Physical: 1's and 0's/bits across a medium (copper, the air, fiber)

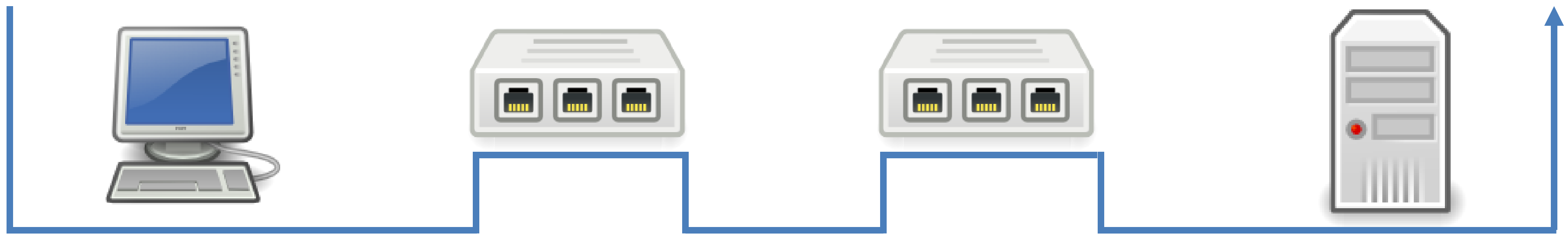
Networks have many concerns, such as reliability, error checking, naming and data ordering. Who/what should be responsible for addressing them? (Why? Which ones belong in which location?)

A. The network should take care of these for us.

B. The communicating hosts should handle these.

C. Some other entity should solve these problems.

The "End-to-End" Argument



- Don't provide a function at lower level of abstraction (layer) if you have to do it at higher layer anyway - *unless there is a very good performance reason to do so.*
- Examples: error control, quality of service
- Reference: Saltzer, Reed, Clark, "End-To-End Arguments in System Design," ACM Transactions on Computer Systems, Vol. 2 (4), 1984.

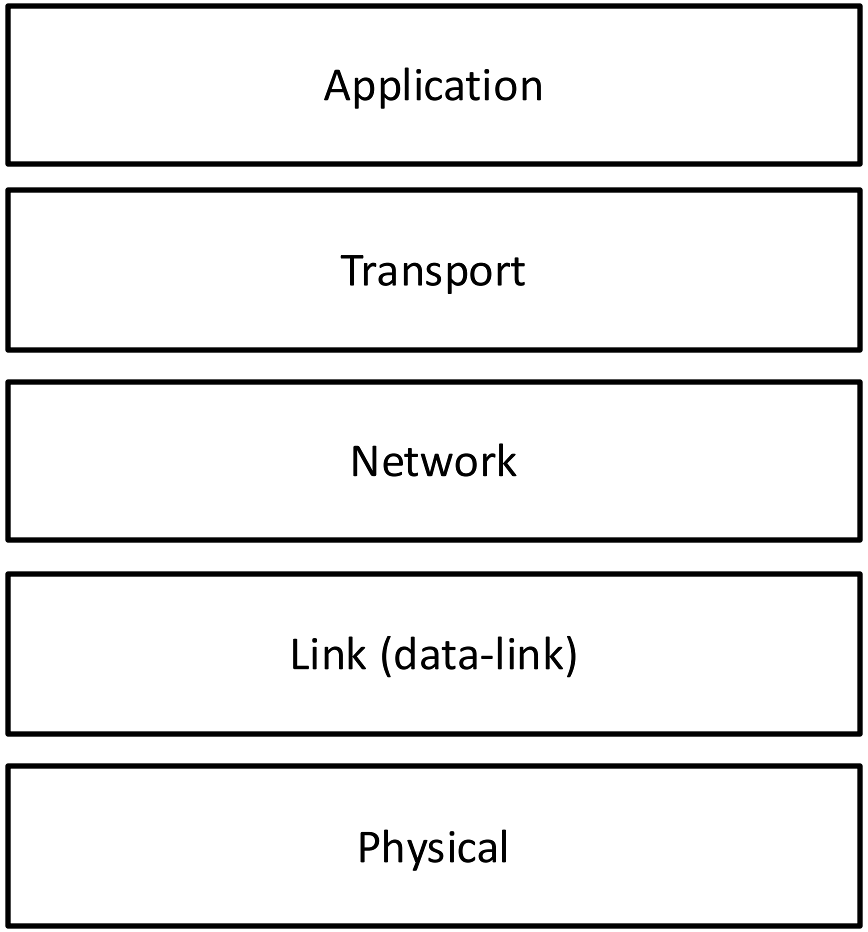
Which layers should routers participate in? (Getting data from host to host.) Why?

A. All of Them

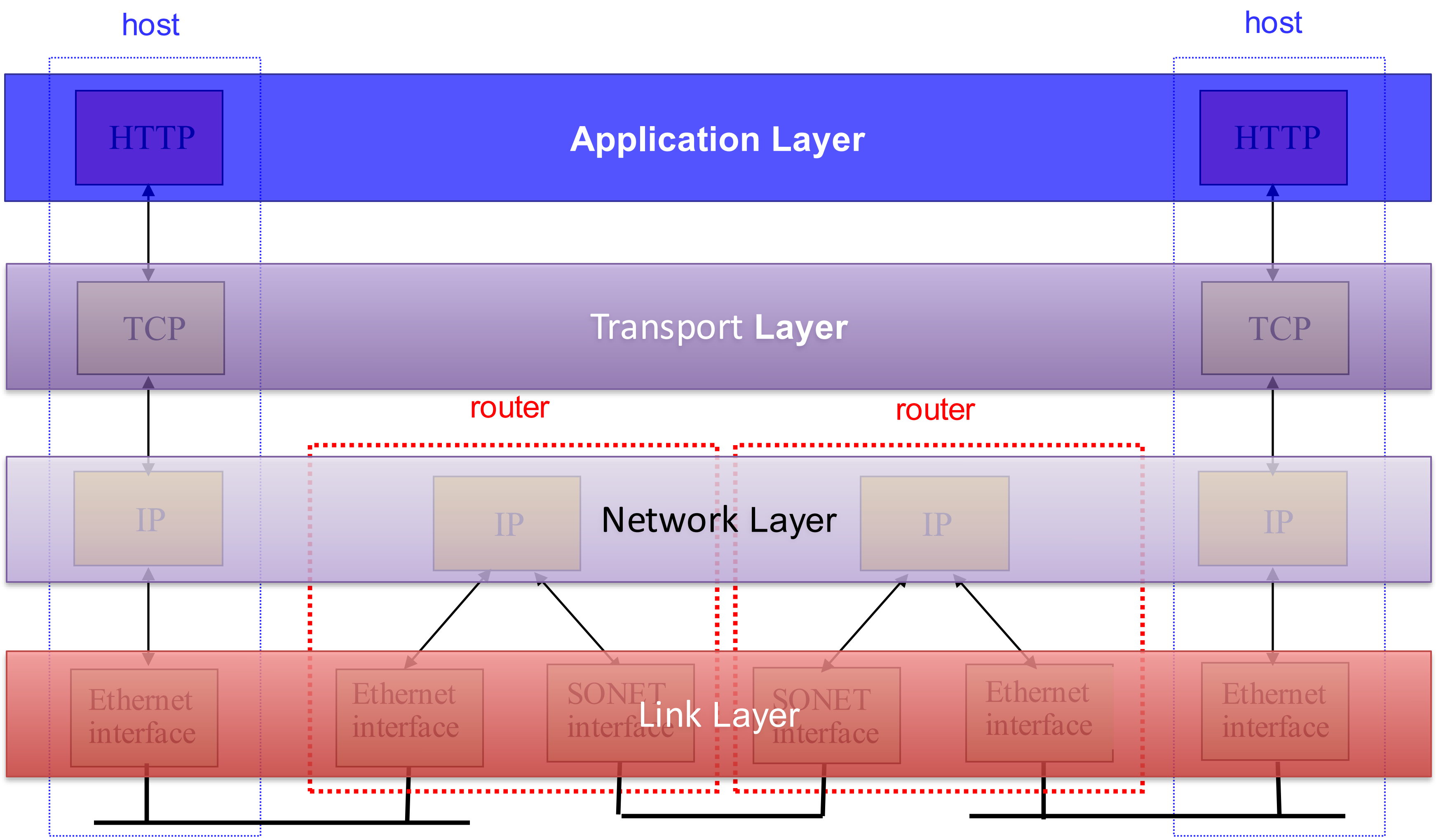
B. Transport through Physical

C. Network, Link and Physical

D. Link and Physical

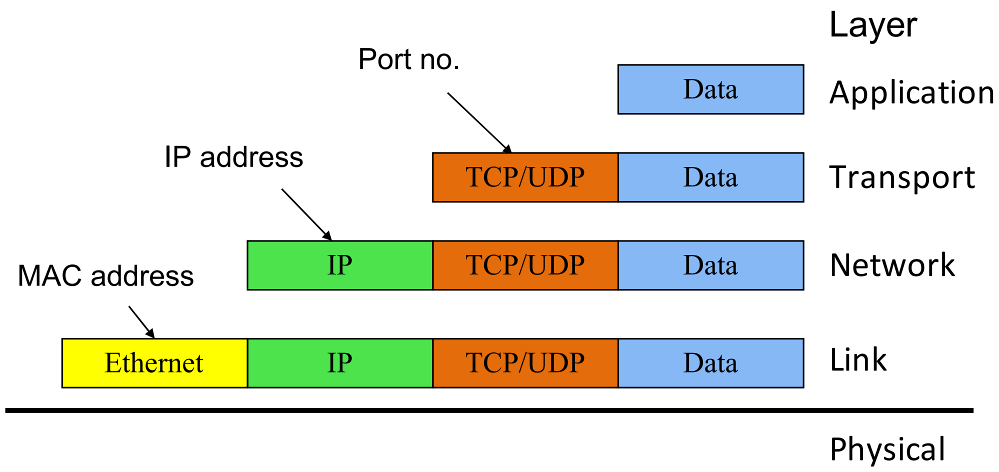


TCP/IP Protocol Stack



Application Layer
(HTTP, FTP, SMTP, Zoom)

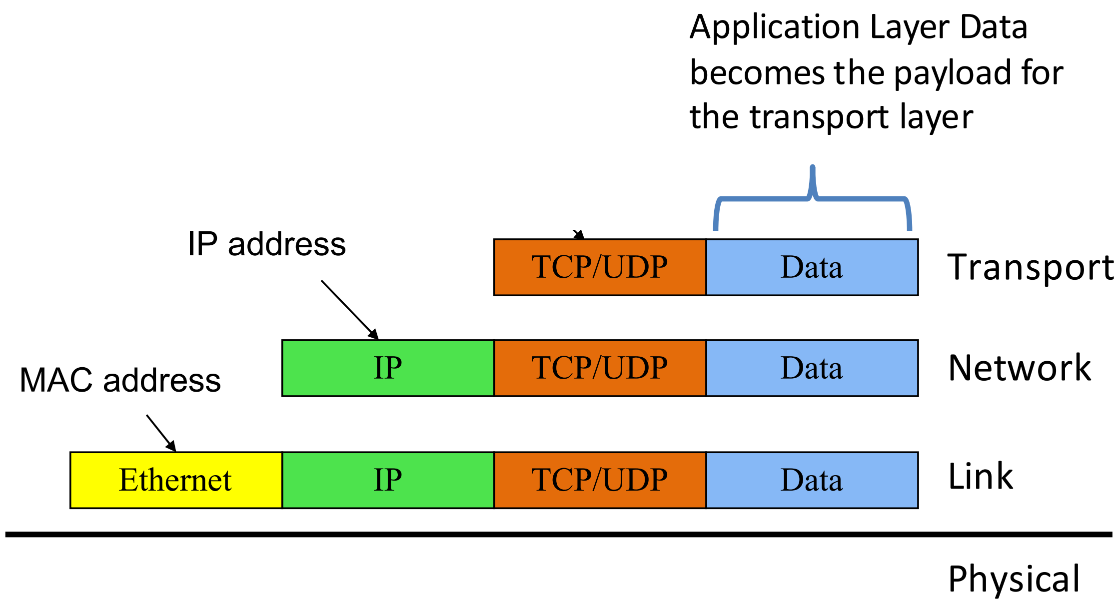
- Does whatever an application does!



Transport Layer (TCP, UDP)

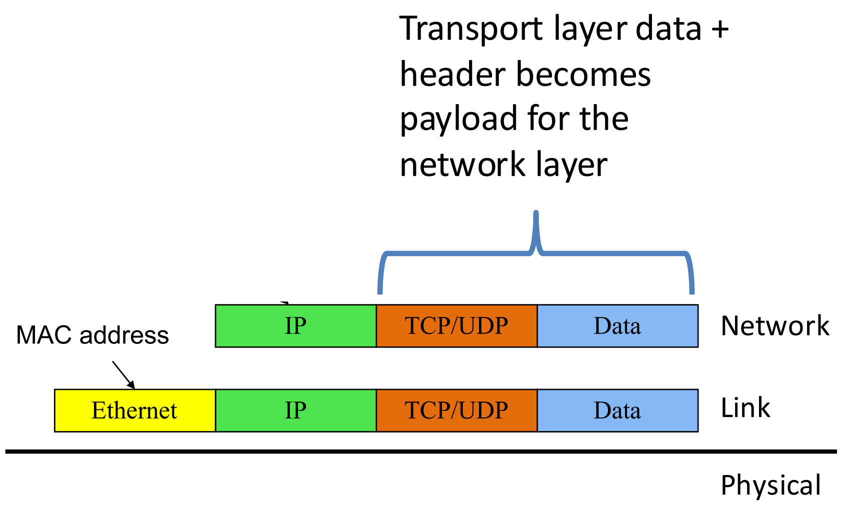
- Provides
  - Ordering
  - Error checking
  - Delivery guarantee
  - Congestion control
  - Flow control

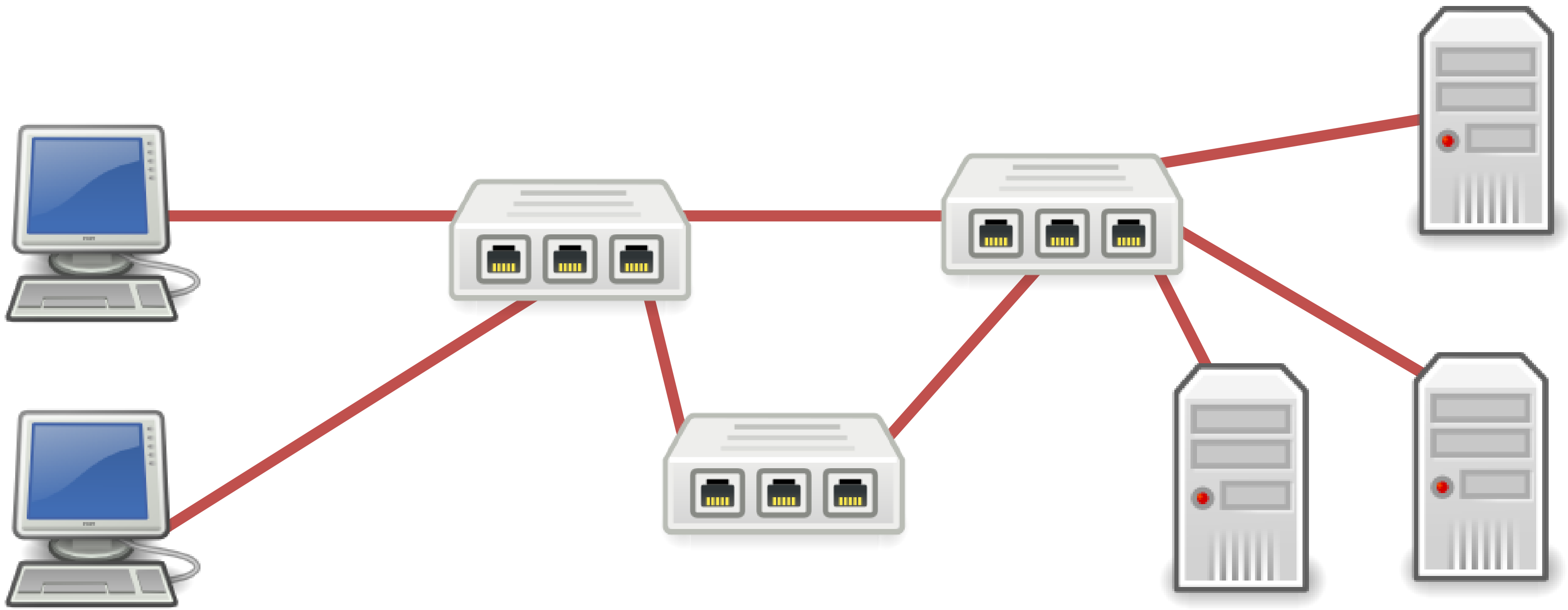
- Or doesn't!



Network Layer (IP)

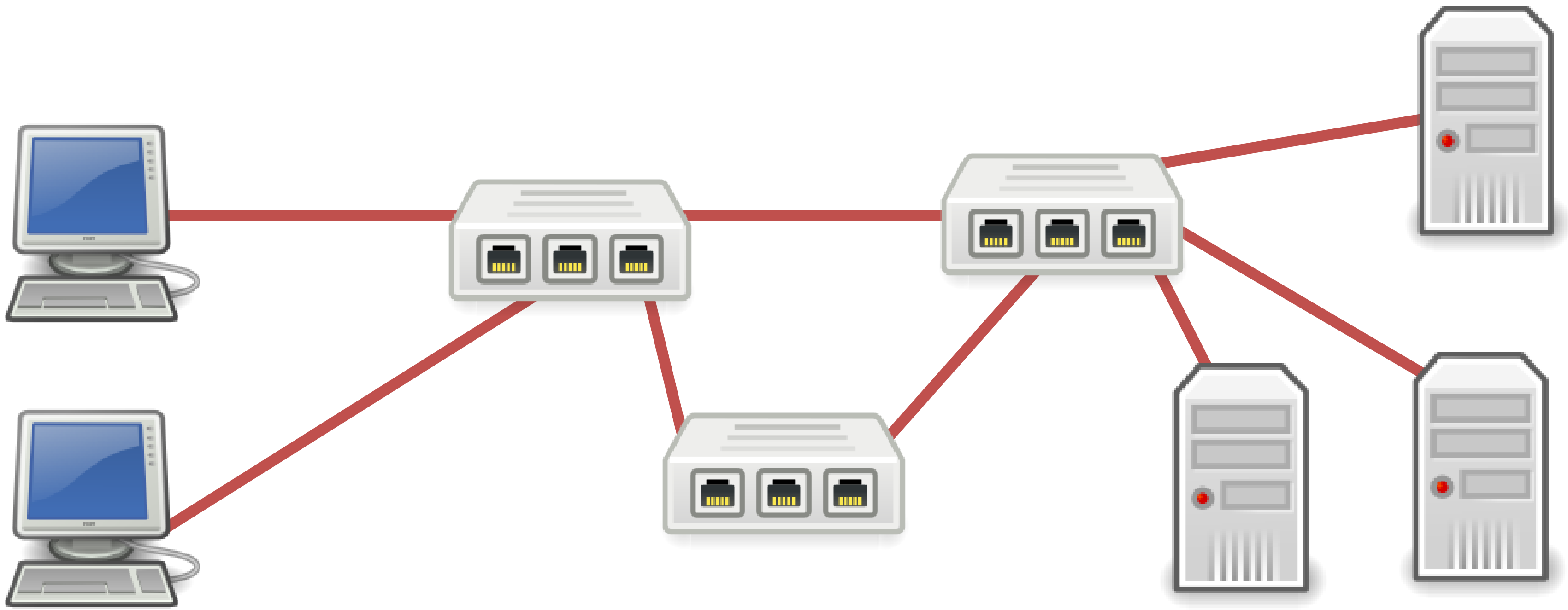
- **Routers**: choose paths through network





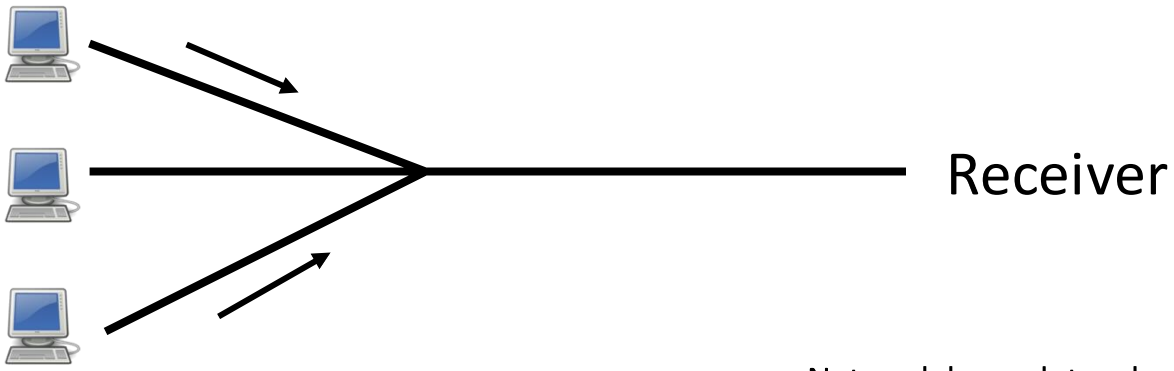
Network Layer (IP)

- **Routers**: chooses paths through network
  - *Circuit switching*: guaranteed channel for a session (Telephone system)
  - *Packet switching*: statistical multiplexing of independent pieces of data (Internet)

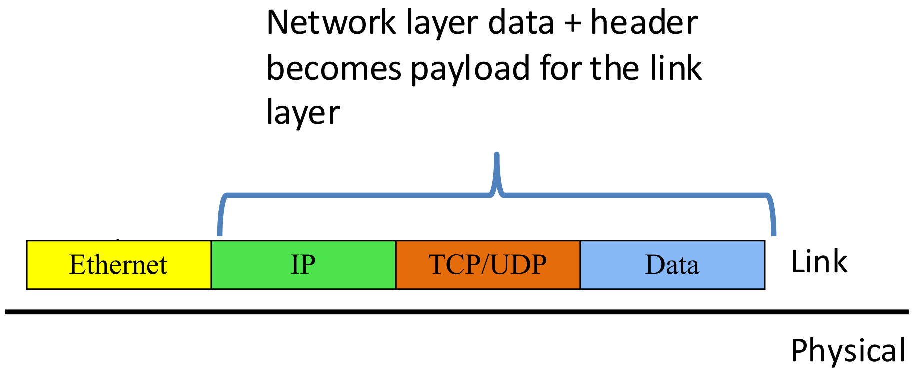


(Data) Link Layer (Ethernet, WiFi, DOCSIS)

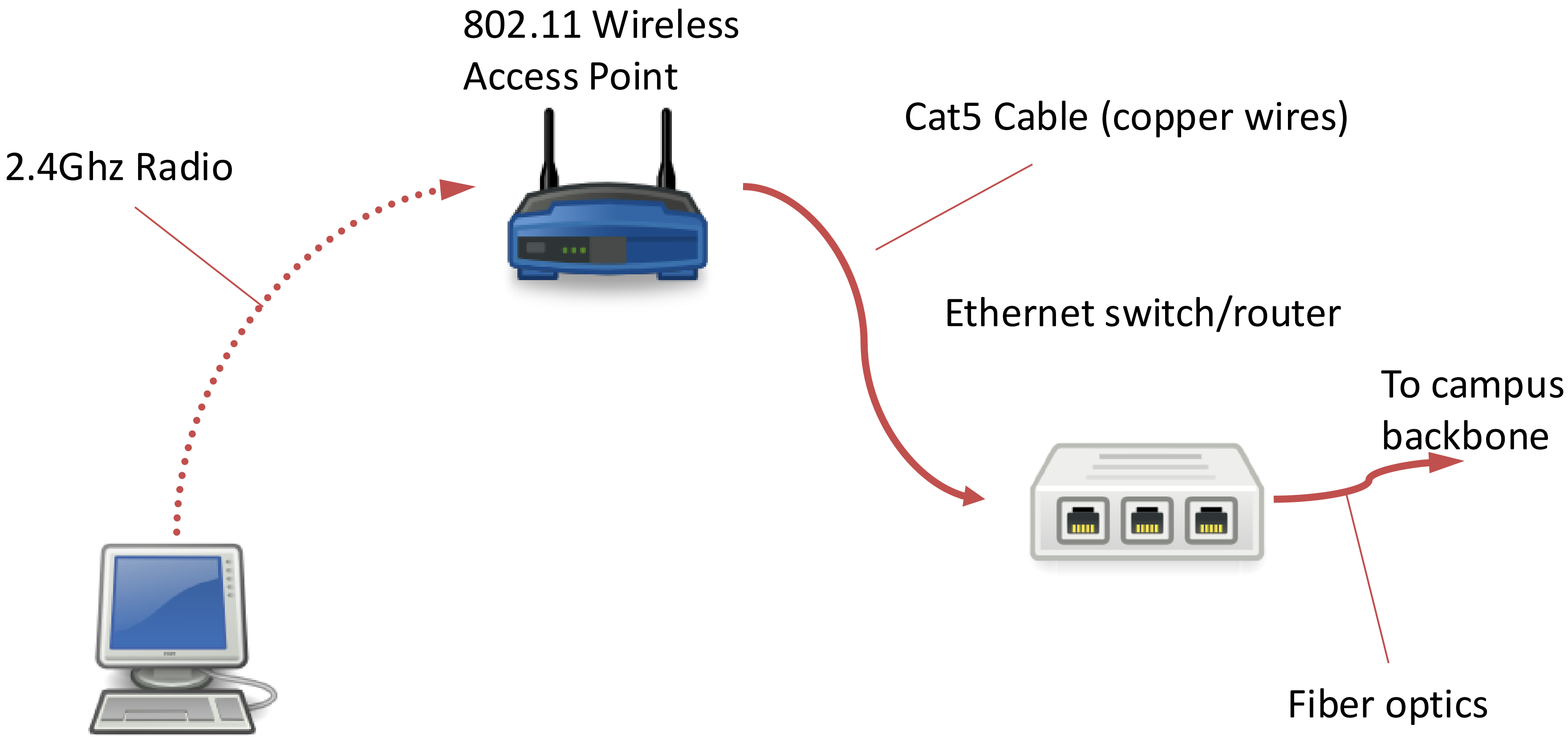
- Break message into chunks (frames) to send over physical medium
- Media access: can it send the frame now?



- Send frame, handle “collisions”



Physical layer
(Copper, Coax, Air, Fiber Optics)



Because of our layering abstractions, we can use any technology we want, at any layer (as long as it doesn't interfere with the other layers). (Why or why not?)

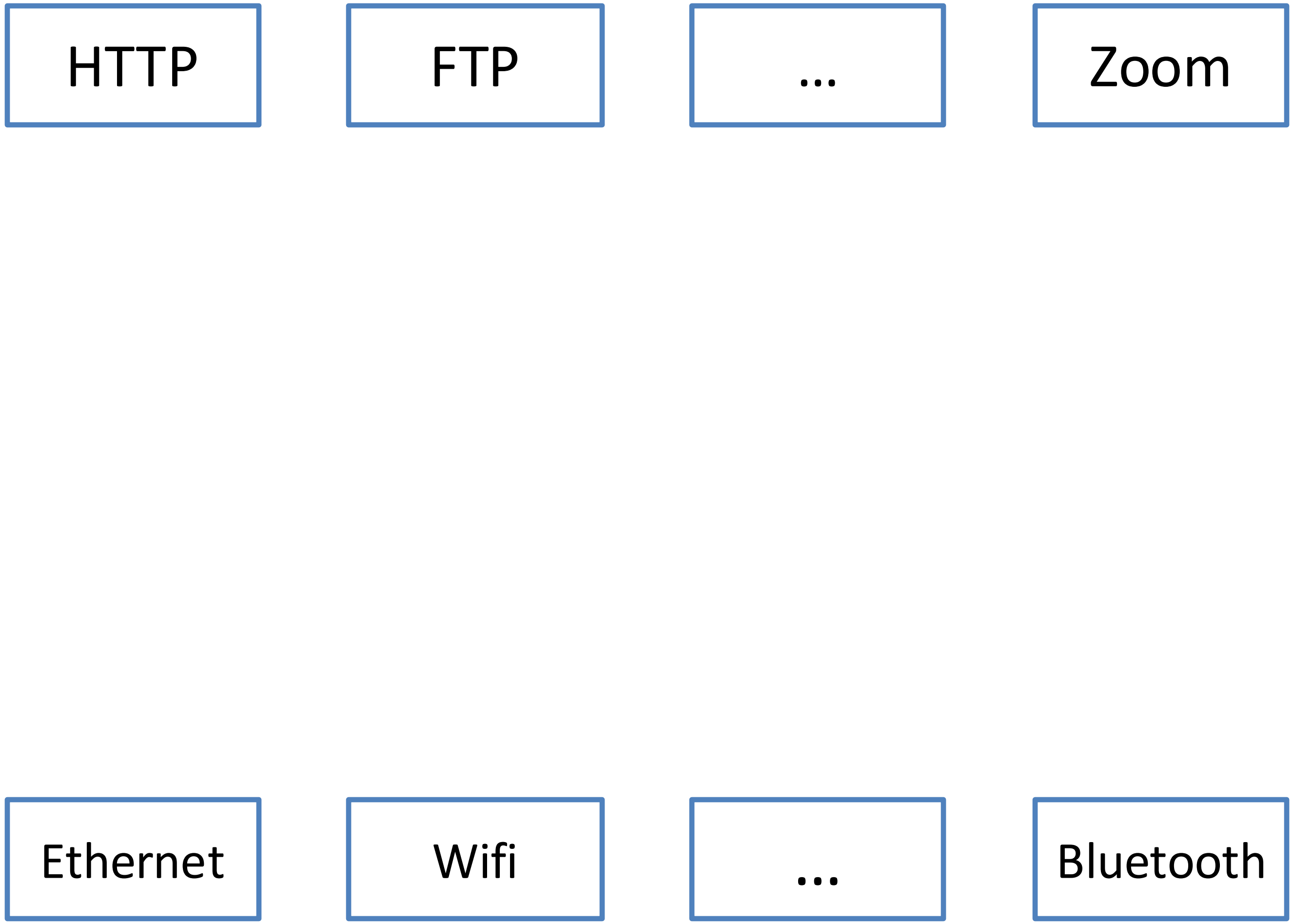
A. Always

B. Usually

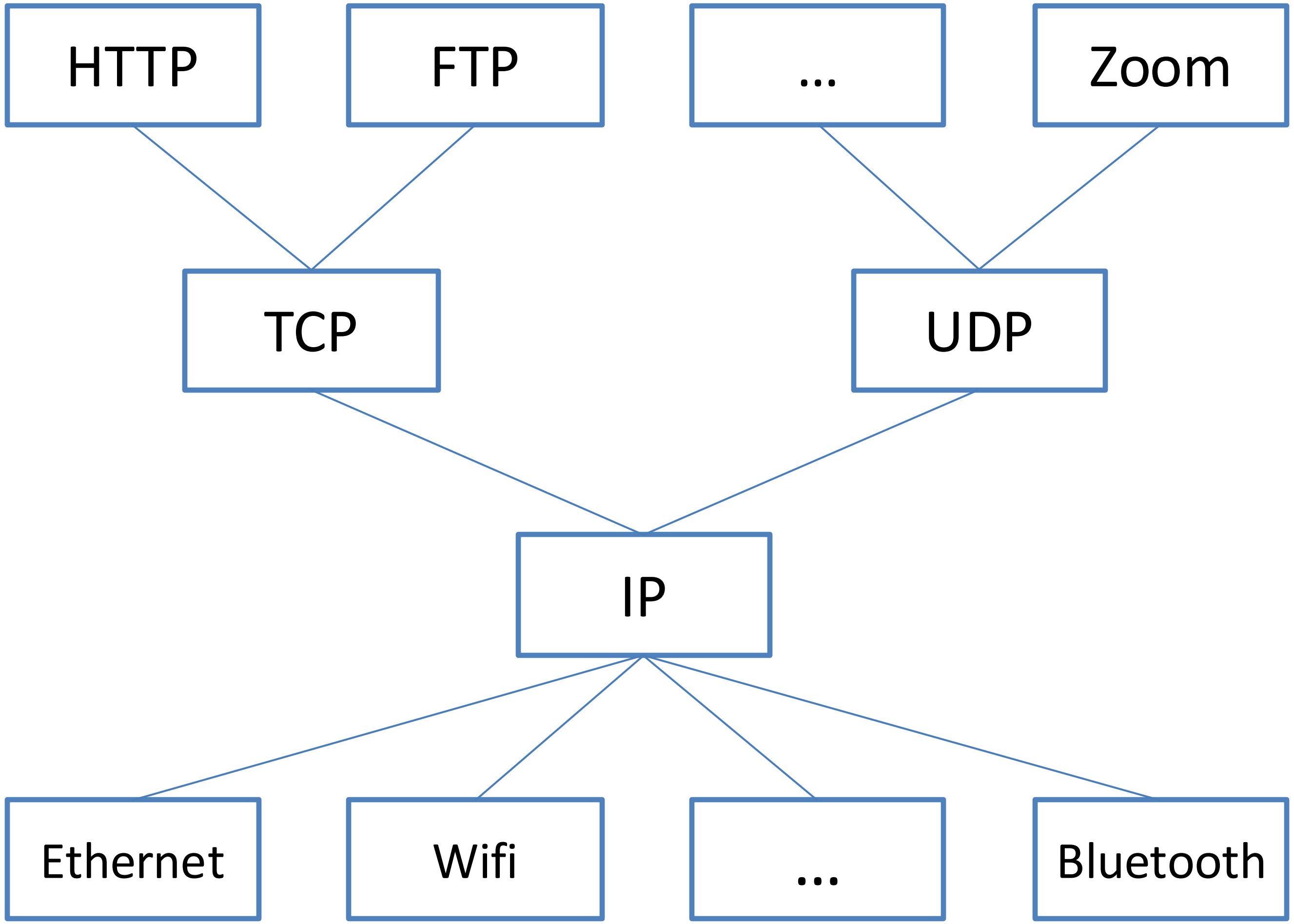
C. Sometimes

D. Never

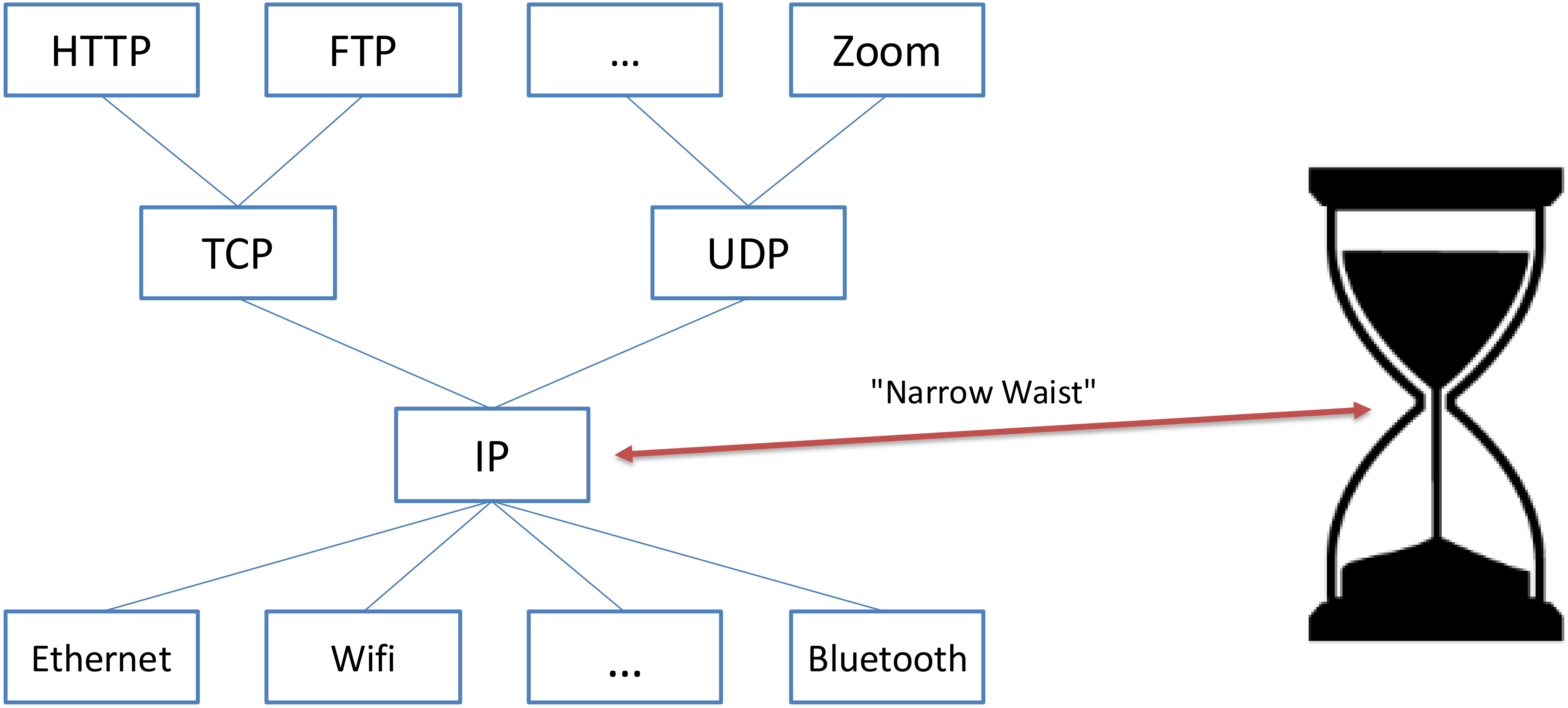
Internet Protocol Suite



Internet Protocol Suite

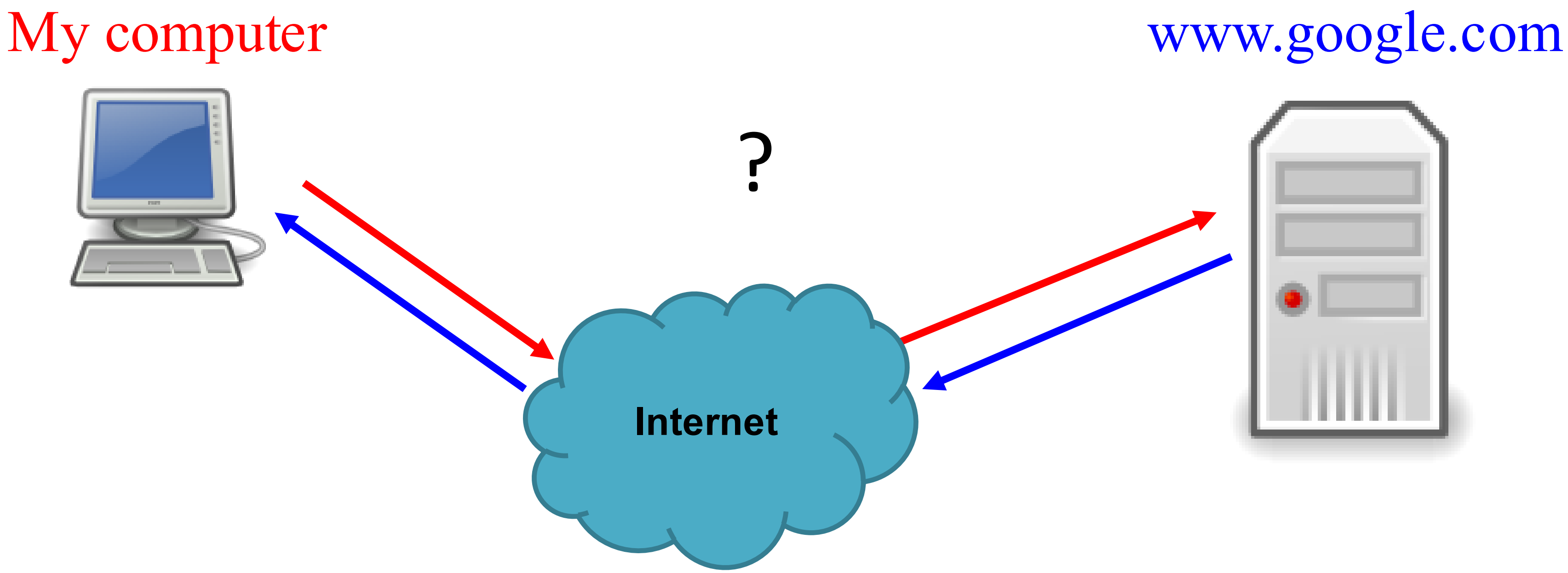


Internet Protocol Suite ("Hourglass model")



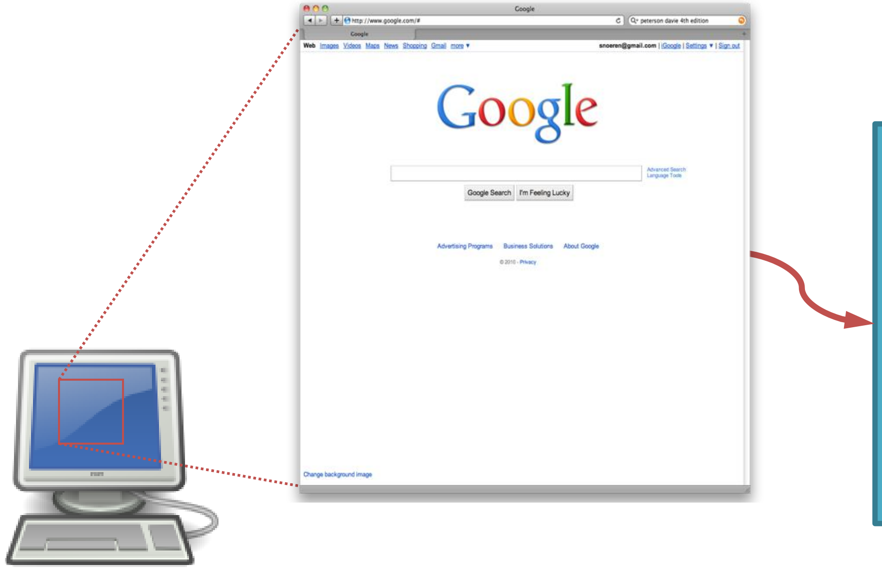
Putting this all together

- **ROUGHLY**, what happens when I click on a Web page from Swarthmore?



Web request (HTTP)

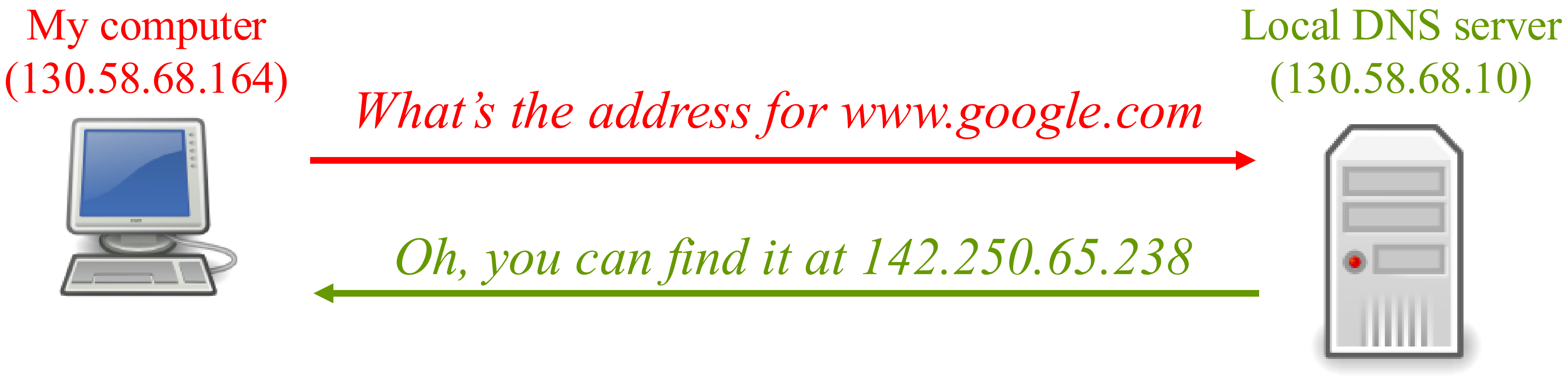
- Turn click into HTTP request




```
GET / HTTP/1.1
Host: www.google.com
…
```

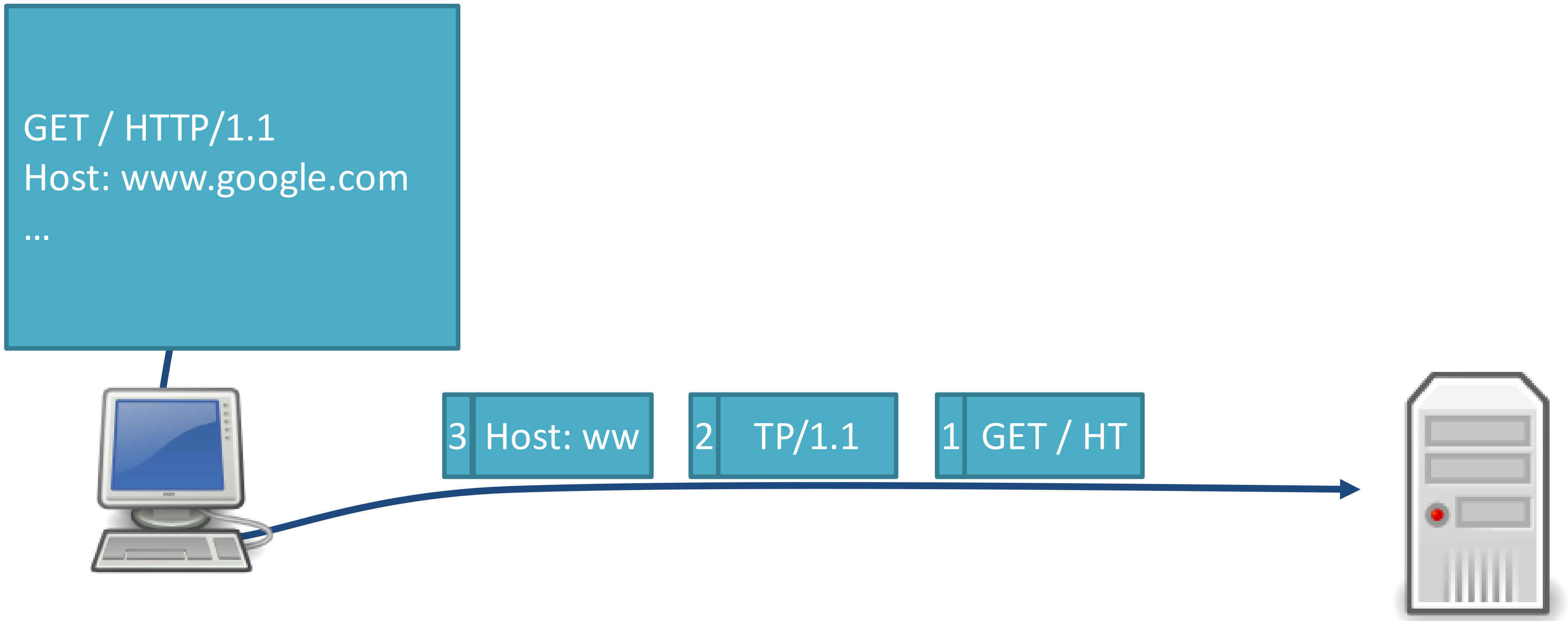
Name resolution (DNS)

- Where is www.google.com?



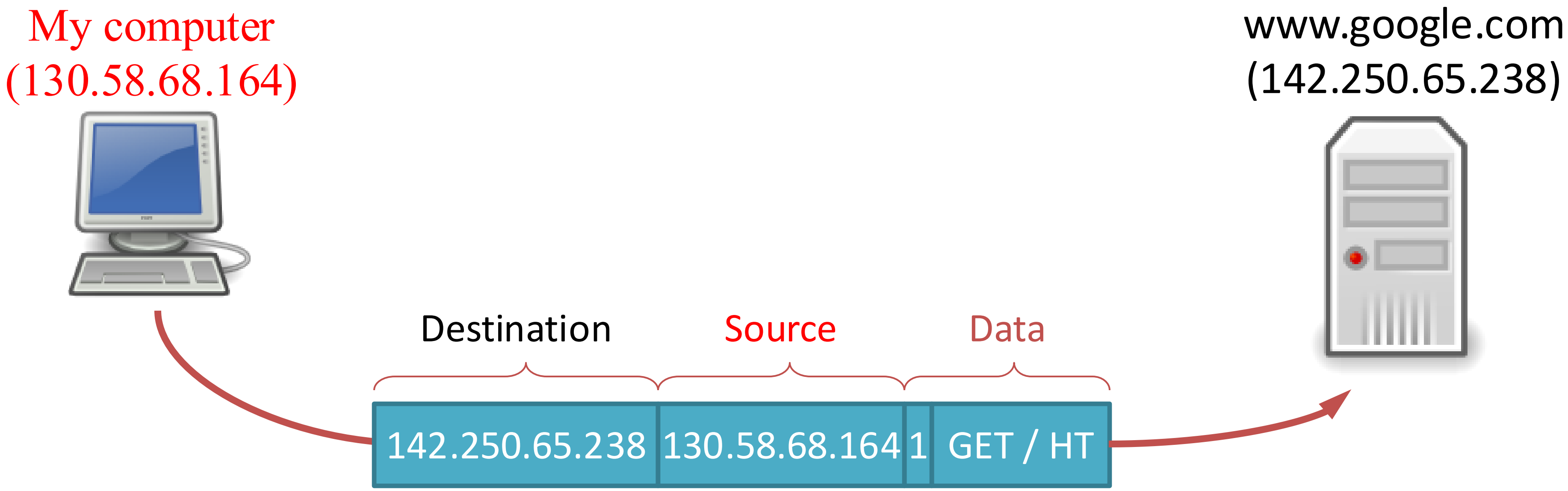
Transport (TCP)

- Break message into chunks (TCP segments)
- Should be delivered reliably & in-order



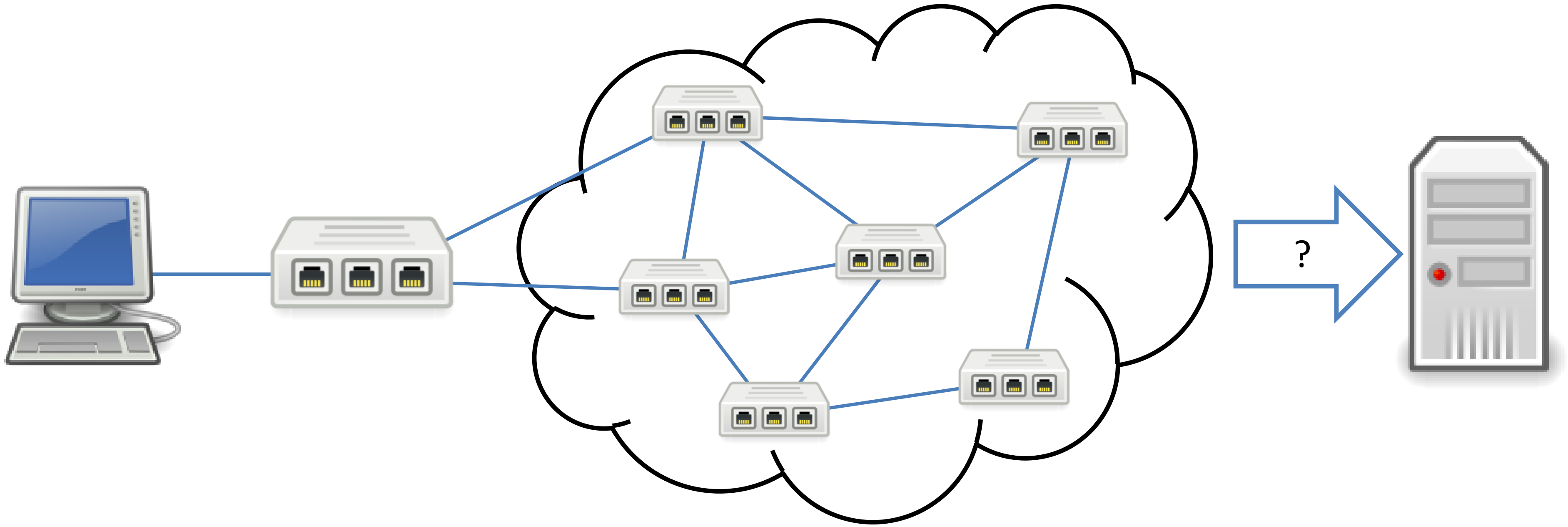
Global Network Addressing

- Add IP header, address each IP packet so it can traverse network and arrive at destination.



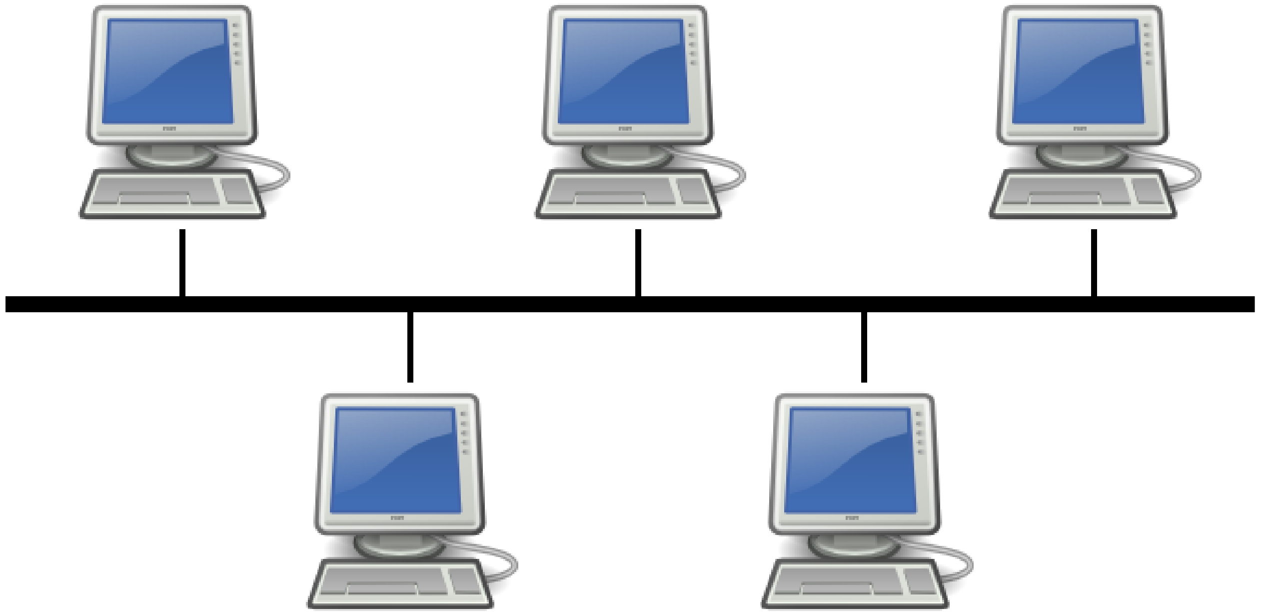
(IP) At Each Router

- Where do I send this to get it closer to Google?
- Which is the best route to take?



Link & Physical Layers

- Forward to the next node!
- Share the physical medium.
- Detect errors.



Summary

- Layers of abstraction divide up responsibility for network functionality
- End-to-end principle: do work at end hosts when possible
- Protocol governs message format and transfer procedure
- Messages encapsulated by protocol headers at each layer

Up Next: Prep For Lab 1

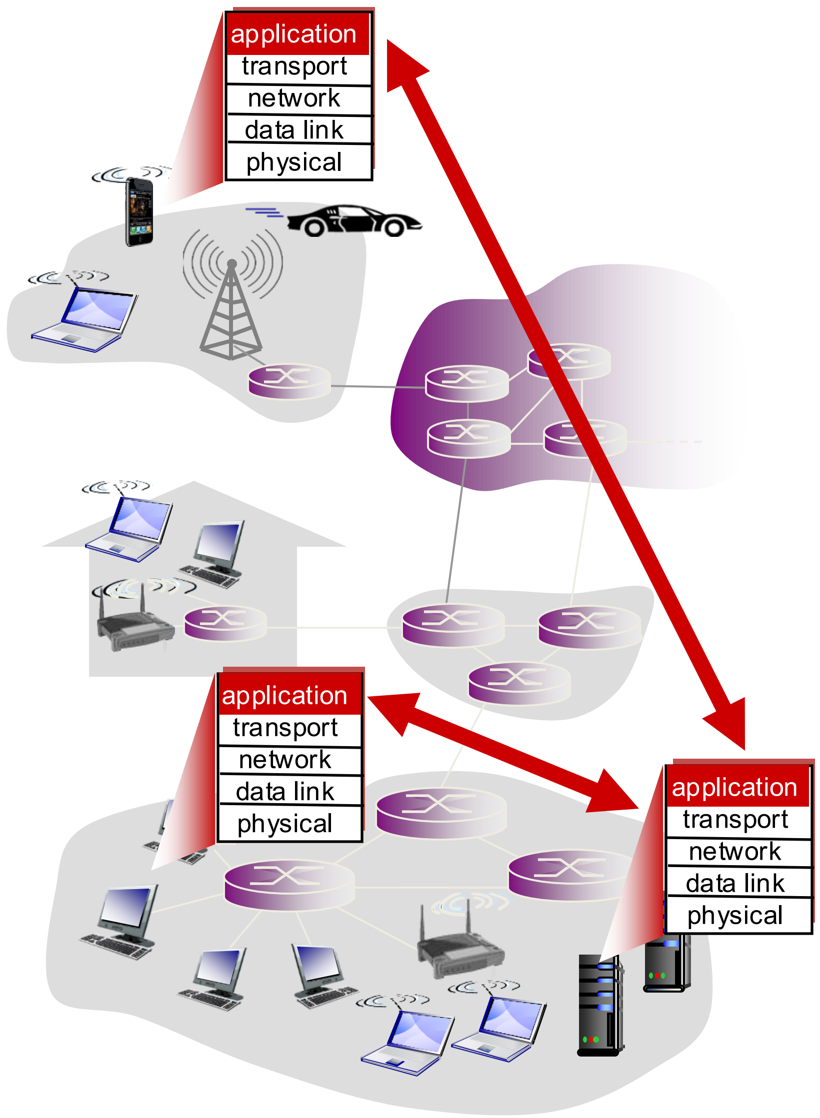
- You need to know a bit about HTTP
- You need to know a bit about sockets

- After we get these lab prerequisites out of the way, we’ll go into more depth about HTTP.

Creating a network app

write programs that:

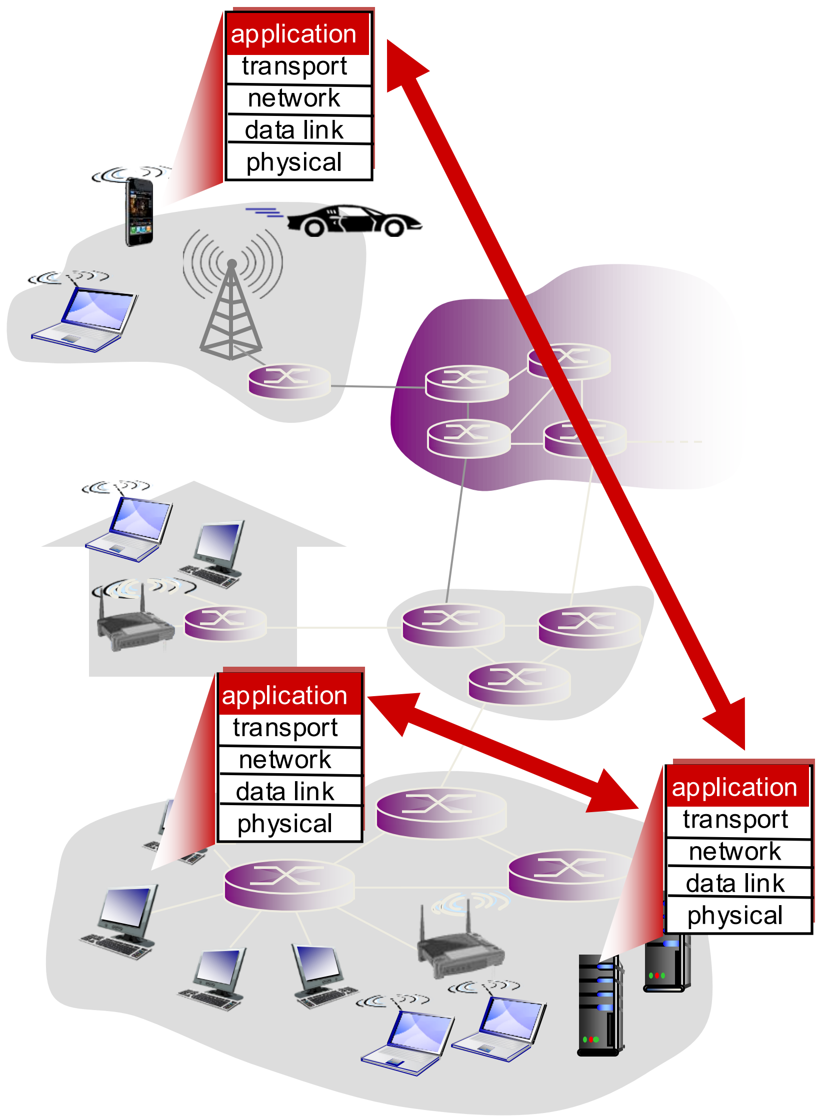
- run on (different) *end systems*
- communicate over network



Creating a network app

no need to write software for network-core devices!

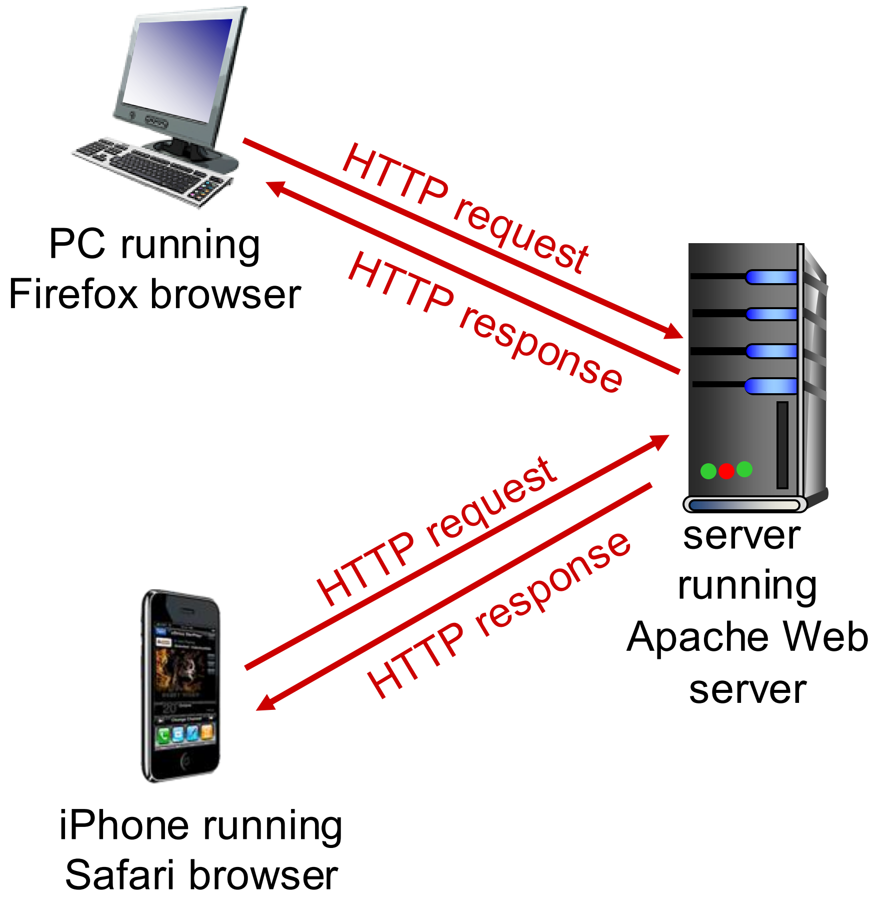
- network-core devices <u>do not run user applications</u>
- applications on end systems
  - rapid app development, propagation



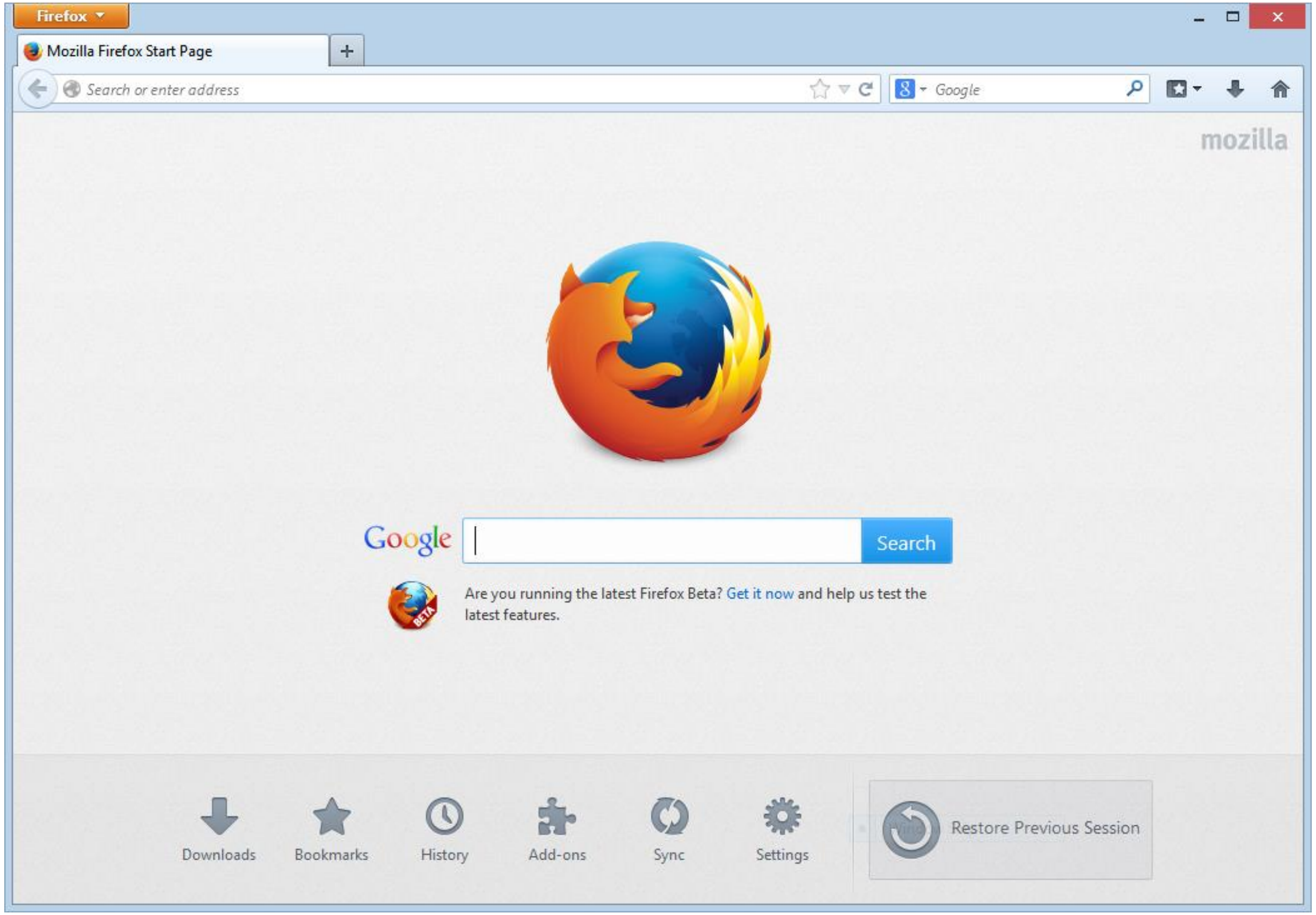
HTTP: HyperText Transfer Protocol

Client/Server model

- client: browser that uses HTTP to request, and receive Web objects.
- server: Web server that uses HTTP to respond with requested object.



What IS A Web Browser?



HTTP and the Web

- web page consists of objects
- object can be: an HTML file (index.html)

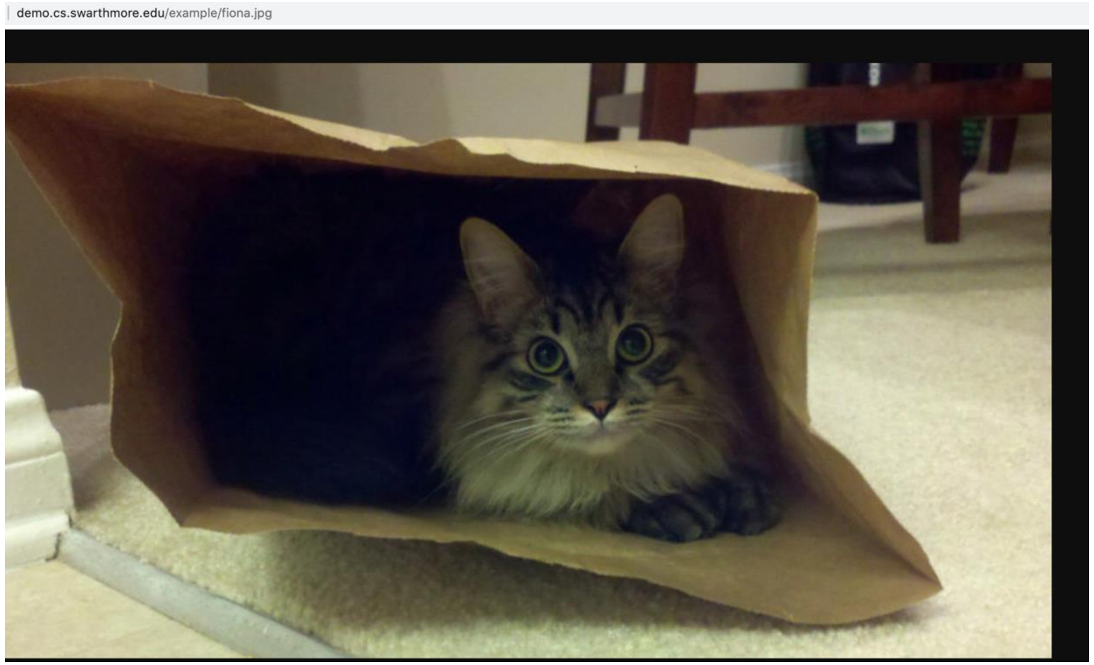
demo.cs.swarthmore.edu/index.html

This is the root page of the demo server. The interesting examples live in the /example directory. They are:

- /example/directory/: An example of a directory.
- /example/fiona.jpg: An example image (one of Kevin's cats).
- /example/hello.txt: A simple text file.
- /example/index.html: An HTML file serving as the default page for the /example directory.
- /example/pic.html: An HTML file that links to the cat picture.
- /example/pride_and_prejudice.pdf: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- /example/pride_and_prejudice.txt: A large text file containing Jane Austen's "Pride and Prejudice".

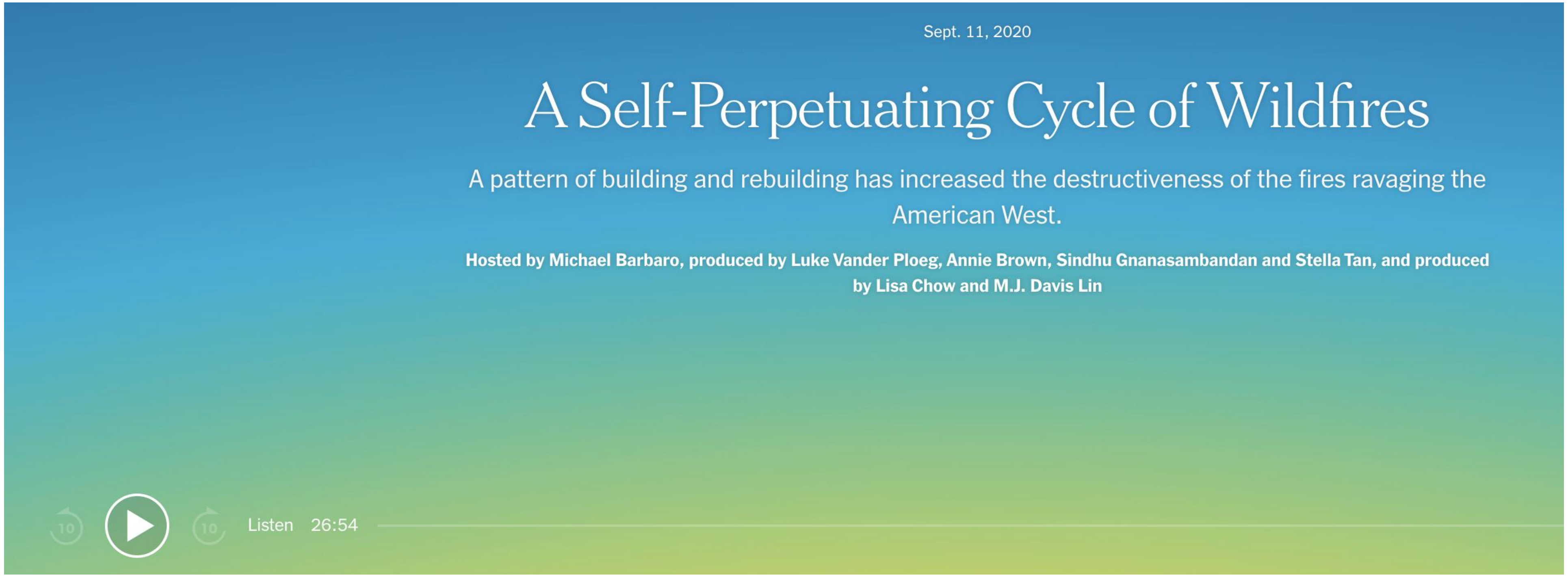
Web objects

- web page consists of objects
- object can be: JPEG image



Web objects

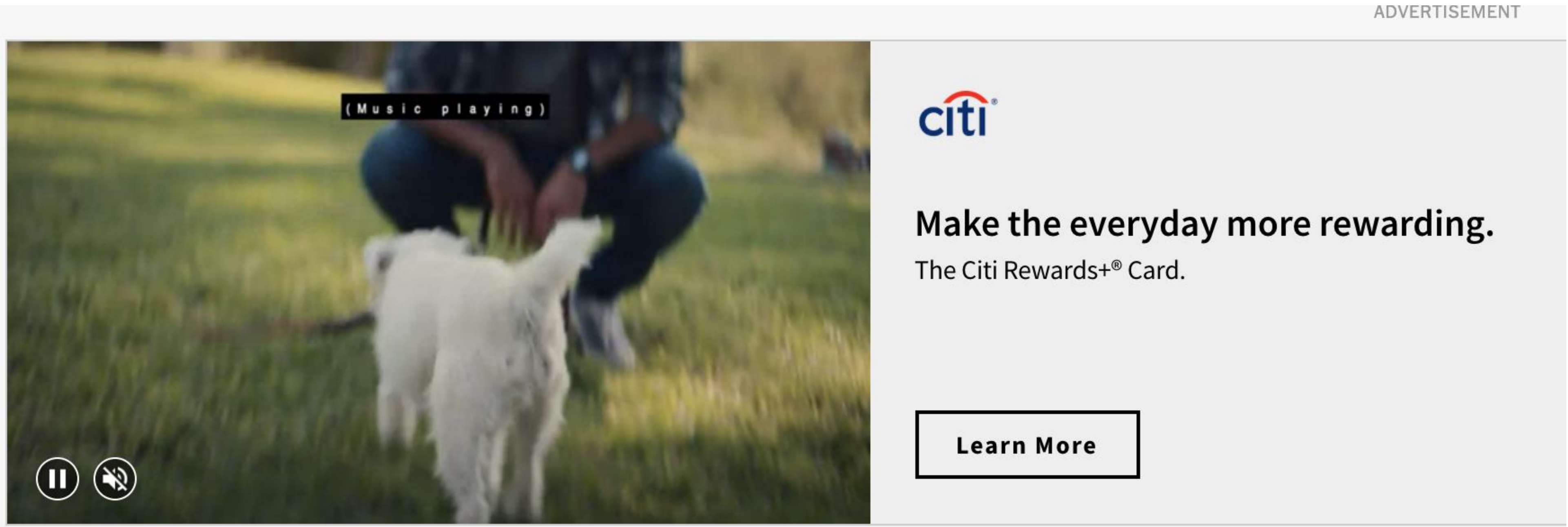
- web page consists of objects
- object can be: audio file



Courtesy: New York Times

Web objects

- web page consists of objects
- object can be: video, java applets, etc.



HTTP and the Web

- a web page consists of base HTML-file which includes several referenced objects
- each object is addressable by a URL, e.g.,

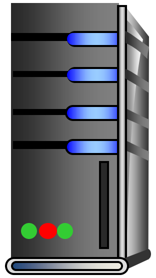
This is the root page of the demo server. The interesting examples live in the /example directory. They are:

- /example/directory/: An example of a directory.
- /example/fiona.jpg: An example image (one of Kevin's cats).
- /example/hello.txt: A simple text file.
- /example/index.html: An HTML file serving as the default page for the /example directory.
- /example/pic.html: An HTML file that links to the cat picture.
- /example/pride_and_prejudice.pdf: A large PDF (binary) file containing Jane Austen's "Pride and Prejudice".
- /example/pride_and_prejudice.txt: A large text file containing Jane Austen's "Pride and Prejudice".

`demo.cs.swarthmore.edu/example/pic.html`

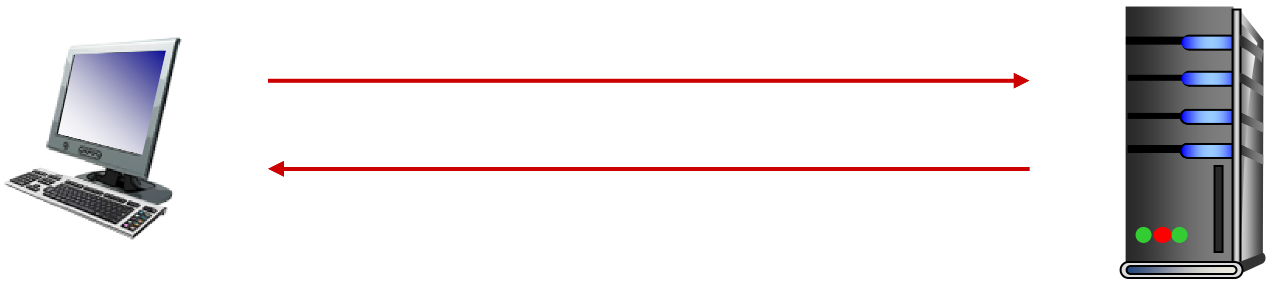
host name: `demo.cs.swarthmore.edu/` — path name: `example/pic.html`

HTTP Overview



1. User types in a URL.
   http://some.host.name.tld/directory/name/file.ext

HTTP Overview



2. Browser establishes connection with server.
   - Looks up “some.host.name.tld”
   - Calls connect()

HTTP Overview

3. Browser requests the corresponding data.

GET /directory/name/file.ext HTTP/1.0
Host: some.host.name.tld
[other optional fields, for example:]
User-agent: Mozilla/5.0 (Windows NT 6.1; WOW64)
Accept-language: en
[Blank line]

HTTP Overview

4. Server responds with the requested data.

HTTP/1.0 200 OK
Content-Type: text/html
Content-Length: 1299
Date: Sun, 01 Sep 2013 21:26:38 GMT
[Blank line]
(Data data data data…)

HTTP Overview

5. Browser renders the response, fetches any additional objects, and waits for server to close the connection.

HTTP Overvi

5. Browser renders the respo
additional objects, and waits
the connection.

```
<html>
  <head>
    <title>Page title!</title>
  </head>

  <body>
    <p>a paragraph of text</p>

    <img src="http://site/cat.jpg">
    <img src="http://site/dog.jpg">
    ...
  </body>
</html>
```

HTTP Overview

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser renders the response, fetches any additional objects, and waits for server to close the connection.

It’s a document retrieval system, where documents point to (link to) each other, forming a “web”.

HTTP Overview (Lab 1)

1. User types in a URL.
2. Browser establishes connection with server.
3. Browser requests the corresponding data.
4. Server responds with the requested data.
5. Browser ~~renders the response, fetches any additional objects, and~~ waits for server to close the connection.

It's a document retrieval system, where documents point to (link to) each other, forming a "web".

Trying out HTTP (client side) for yourself

1. Telnet to your favorite Web server:

```
telnet demo.cs.swarthmore.edu 80
```

Opens TCP connection to port 80 (default HTTP server port) at example server. Anything typed is sent to server on port 80 at demo.cs.swarthmore.edu

2. Type in a GET HTTP request:

```
GET / HTTP/1.0
Host: demo.cs.swarthmore.edu
(blank line)
```

By typing this in (hit enter twice), you send this minimal (but complete) GET request to the HTTP server.

3. Look at response message sent by HTTP server!

Example (live demo)

Example

```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET /example/hello.txt HTTP/1.0
Host: demo.cs.swarthmore.edu

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

Request

Response headers

Response body

(This is what you should be saving to file in lab 1.)

Note!

```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.sw
Escape character is '^
GET /example/hello.txt
Host: demo.cs.swarthmor

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

This server is intentionally NOT using encryption, to make it easier to work with for lab 1!

HTTPS (live demo)

- Telnet transfers unencrypted data ("clear text")
  - Great for learning
  - Not so great for real world security / privacy

- For a similar (interactive) command line experience with encryption:
  - `openssl s_client -crlf -connect server.name:443`

HTTP request message

- two types of HTTP messages: *request, response*
- HTTP request message:
  - ASCII (human-readable format)

carriage return character (CR)

line-feed character (LF)

request line (GET, POST, HEAD, etc. commands)

header lines

carriage return, line feed

```
GET /~kwebb/index.html HTTP/1.1\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

Why do we have these \r\n (CRLF) things all over the place?

```
GET /~kwebb/index.html HTTP/1.1\r\n
Host: web.cs.swarthmore.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

A. They’re generated when the user hits ‘enter’.
B. They signal the end of a field or section.
C. They’re important for some other reason.
D. They’re an unnecessary protocol artifact.

How else might we delineate messages?
(What are the good/bad properties of each of these ideas?)

A. There's not much else we can do.

B. Force all messages to be the same size.

C. Send the message size prior to the message.

D. Some other way (discuss).

HTTP is all text…

- Makes the protocol simple
  - Easy to delineate message (\r\n)
  - (Relatively) human-readable
  - No worries about encoding or formatting data
  - Variable length data
- Not the most efficient
  - Many protocols use binary fields
    - Sending “12345678” as a string is 8 bytes
    - As an integer, 12345678 needs only 4 bytes
  - The headers may come in any order
  - Requires string parsing / processing

HTTP is all text…

- The HTTP **PROTOCOL** is all text
  - That is, the messages that are required (request and response)
  - All headers are text

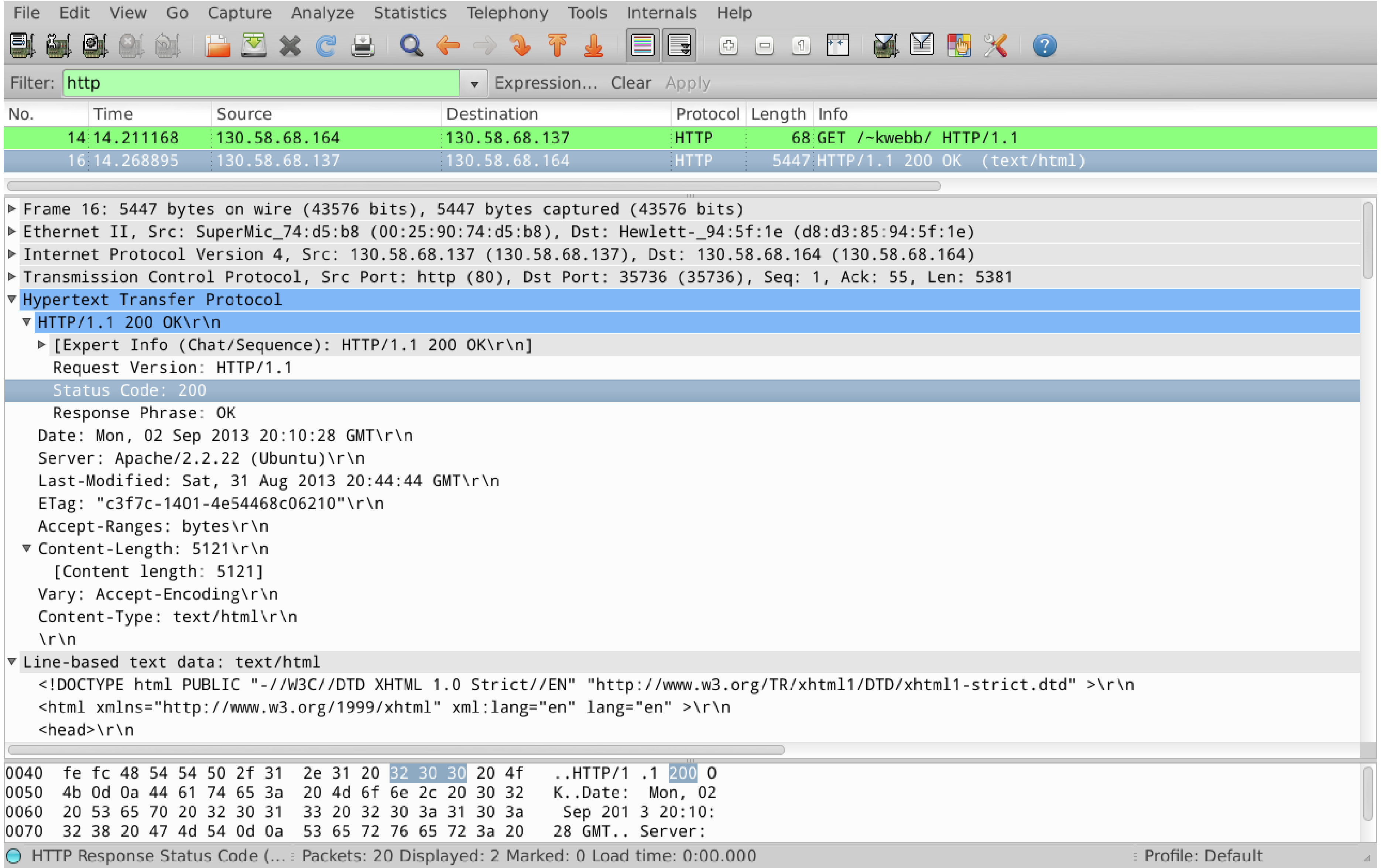
- The BODY of a message might **NOT** be text

- This distinction is critically important for lab 1!
  - Fine to use string functions on HTTP messages
  - You better not use string functions on body data

Visualizing HTTP: telnet

```
kwebb@sesame ~ $ telnet demo.cs.swarthmore.edu 80
Trying 130.58.68.26...
Connected to demo.cs.swarthmore.edu.
Escape character is '^]'.
GET /example/hello.txt HTTP/1.0
Host: demo.cs.swarthmore.edu

HTTP/1.0 200 OK
Content-Type: text/plain; charset=utf-8
ETag: "914817348"
Last-Modified: Mon, 24 Feb 2020 06:06:27 GMT
Content-Length: 40
Connection: close
Date: Thu, 20 Jan 2022 18:03:33 GMT
Server: lighttpd/1.4.59

Hello, you found the example text file!
```

Visualizing HTTP: wireshark



- There’s more to say about HTTP, but for lab 1, let’s talk a bit about sockets too…

What is a socket?

An inter-process communication (IPC) abstraction through which an application may send and receive data.

Behaves similarly to way an open file handle allows an application to read and write data to storage.

Recall Inter-process Communication (IPC)

- Processes must communicate to cooperate

- Must have two mechanisms:
  - Data transfer
  - Synchronization

Inter-process Communication (IPC)

- Operating systems provide several IPC mechanisms (Take CS 45)
  - files
  - shared memory (in several ways)
  - pipes
  - …
  - sockets

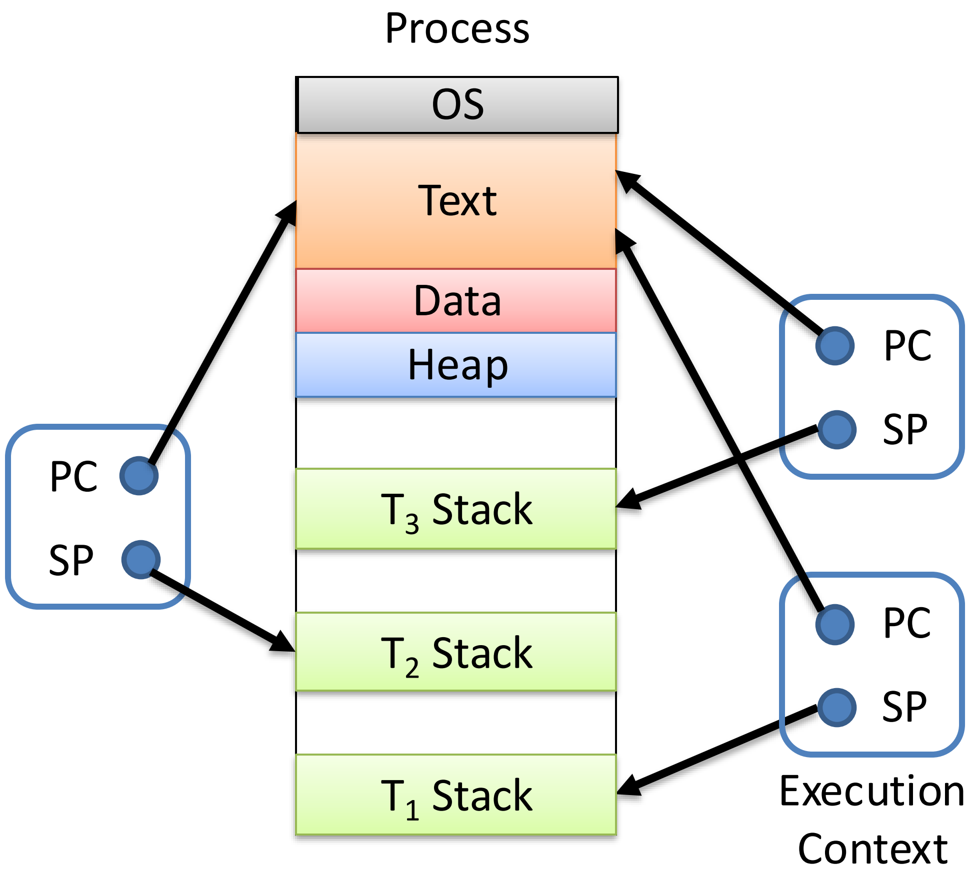
- Broadly, these fall into two categories:
  1. Shared memory
  2. Message passing

Only works on one computer (shared hardware).

Also, this is what you're most familiar with.

Thread Model (Shared Memory)

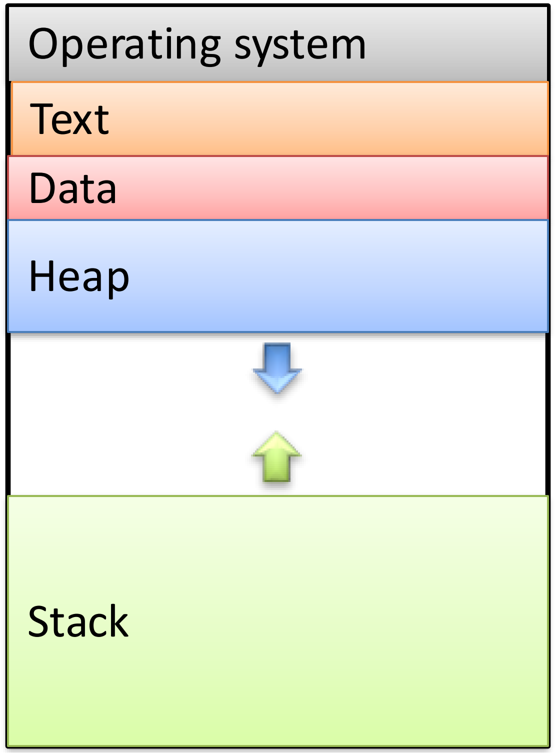
- Single process with multiple copies of execution resources.
- ONE shared virtual address space!
  - All process memory shared by every thread.
  - Threads coordinate by sharing variables (typically on heap)



Note: this is technically not IPC (there's only one process), but this is the most common form of shared memory today.

Message Passing IPC (Pipe)

P_1

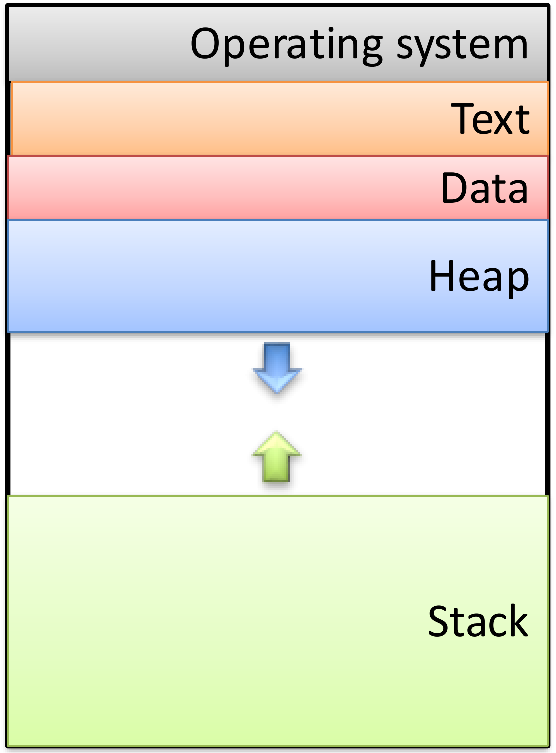


Let's say process P_1 wants to send data to process P_2.

They execute on the same hardware and share an operating system.

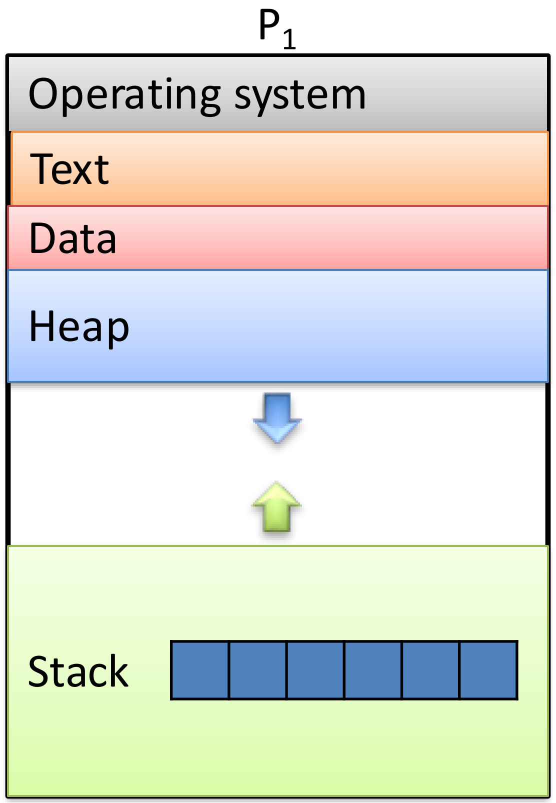
They do NOT directly share any memory.

P_2



OS kernel

Message Passing IPC (Pipe)

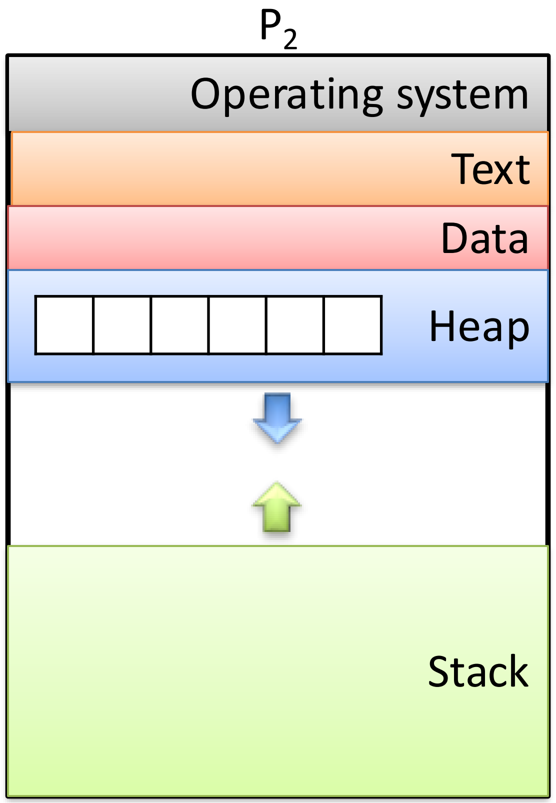


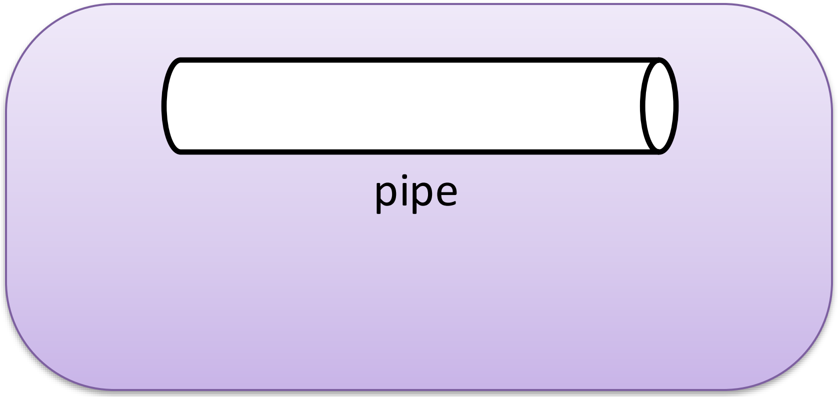
P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

data pointer: the start of data to copy

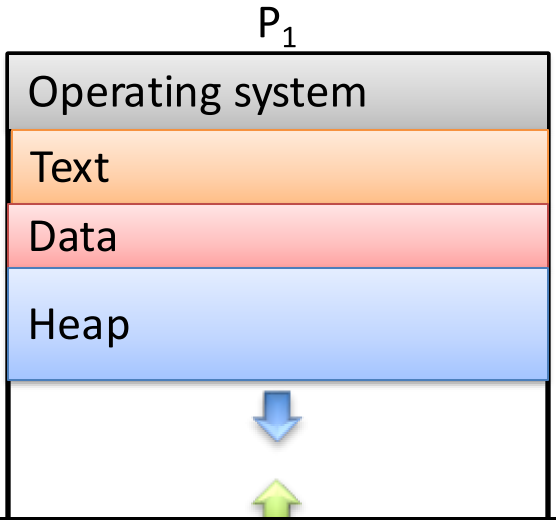
count: how many bytes to copy (at most)





OS kernel

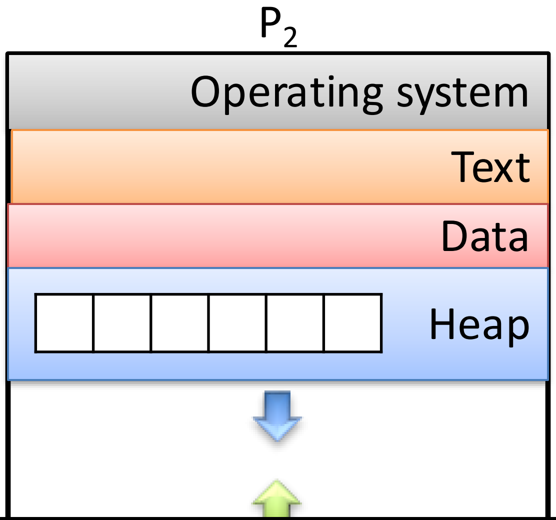
Message Passing IPC (Pipe)



P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

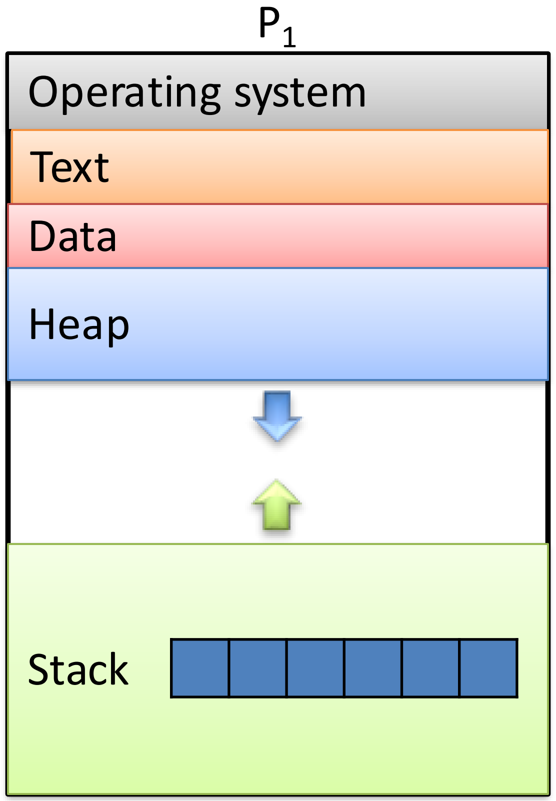
data pointer: the start of data to copy




```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes up to count bytes from the buffer starting at buf to
       the file descriptor referred to by the file descriptor fd.
```

OS kernel

Message Passing IPC (Pipe)

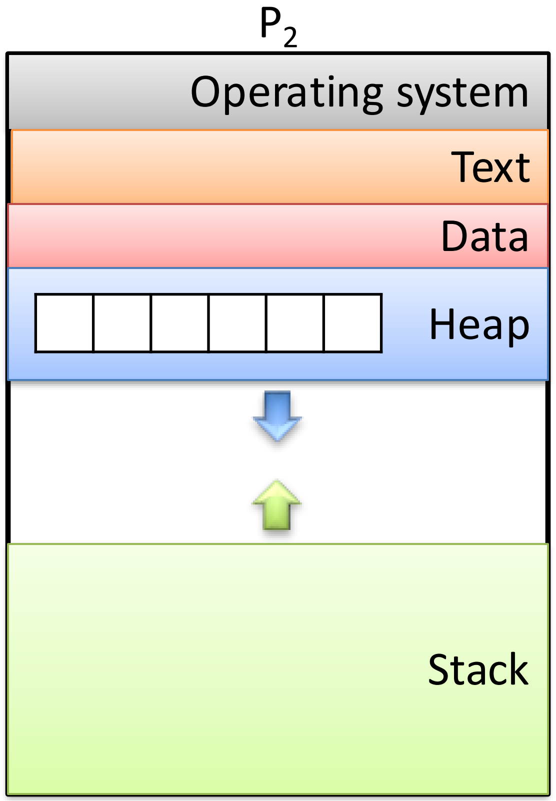


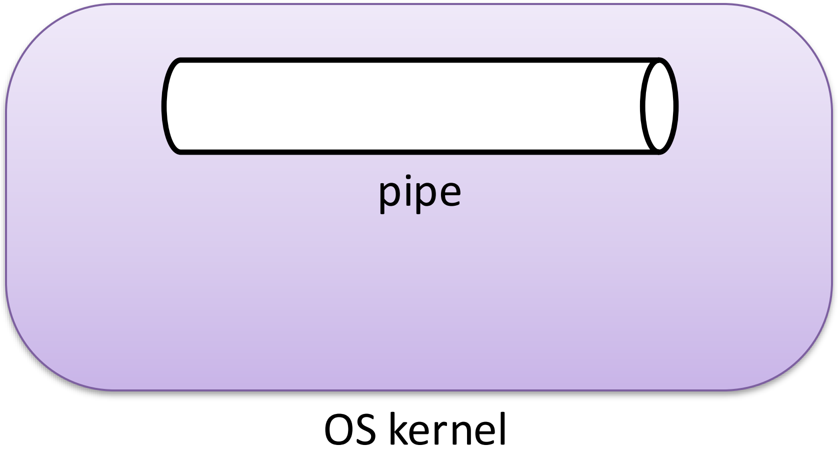
P_1 can send data into the pipe by calling:

```
write(fd, data pointer, count)
```

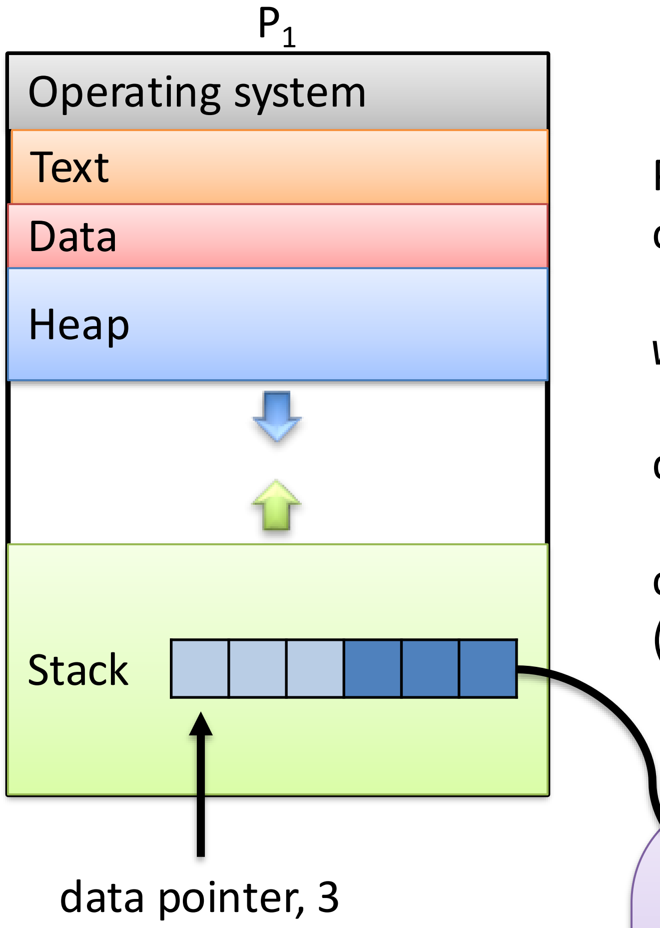
data pointer: the start of data to copy

count: how many bytes to copy





Message Passing IPC (Pipe)

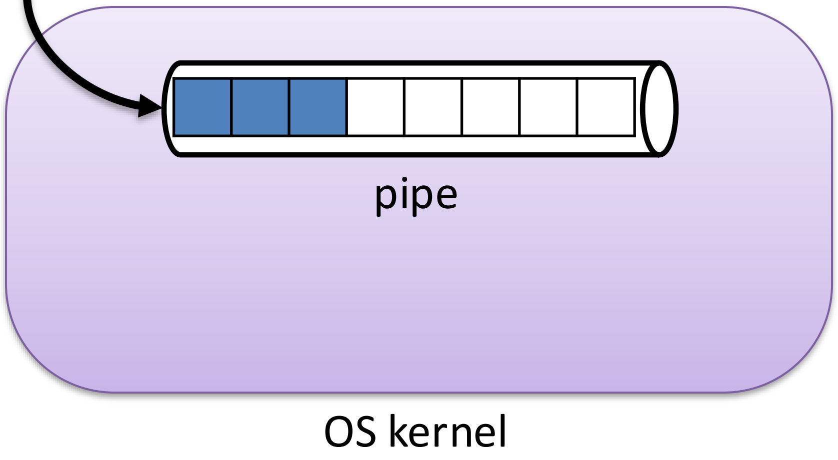


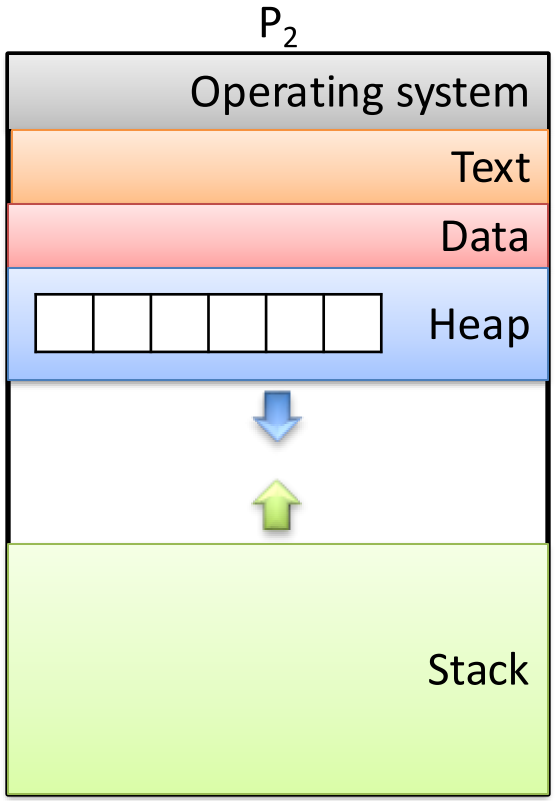
P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

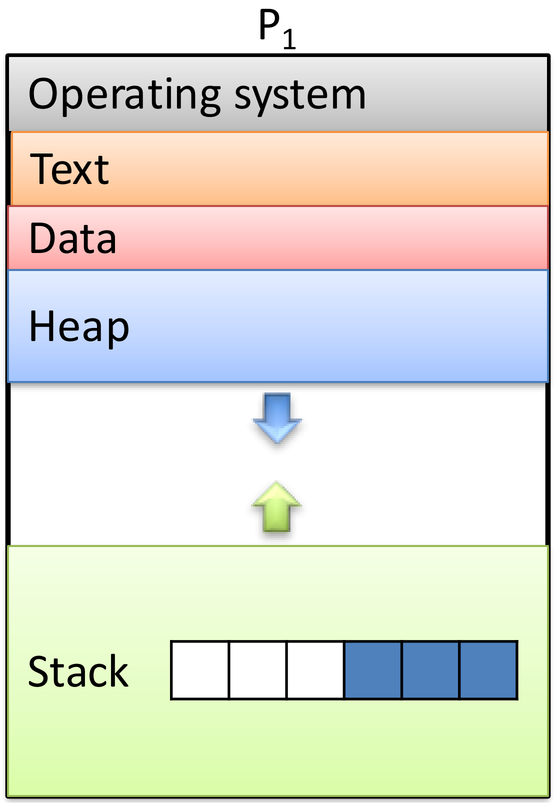
data pointer: the start of data to copy

count: how many bytes to copy (at most)





Message Passing IPC (Pipe)

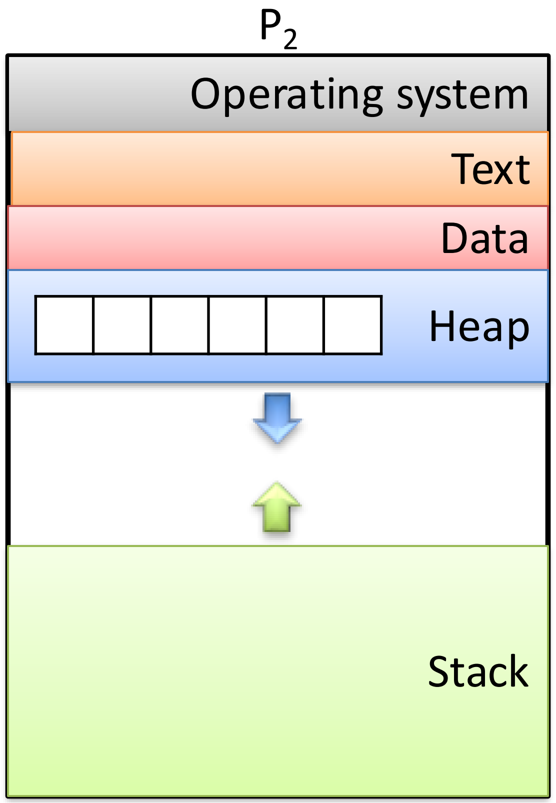


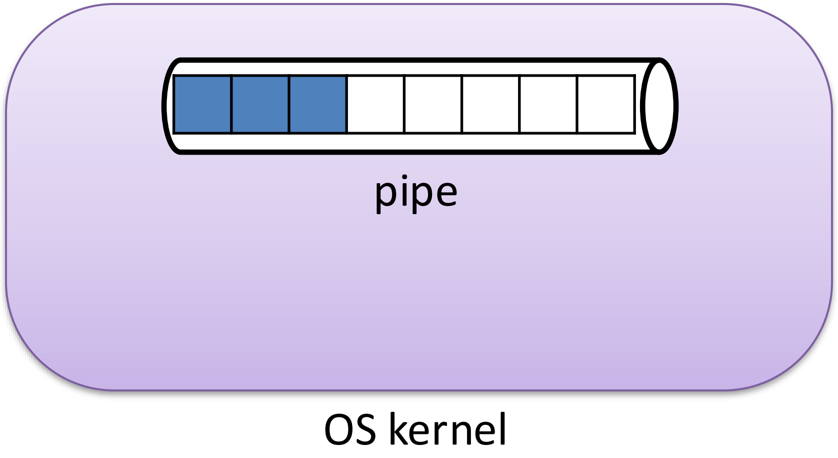
P_2 can receive data from the pipe by calling:

`read(…, data pointer, count)`

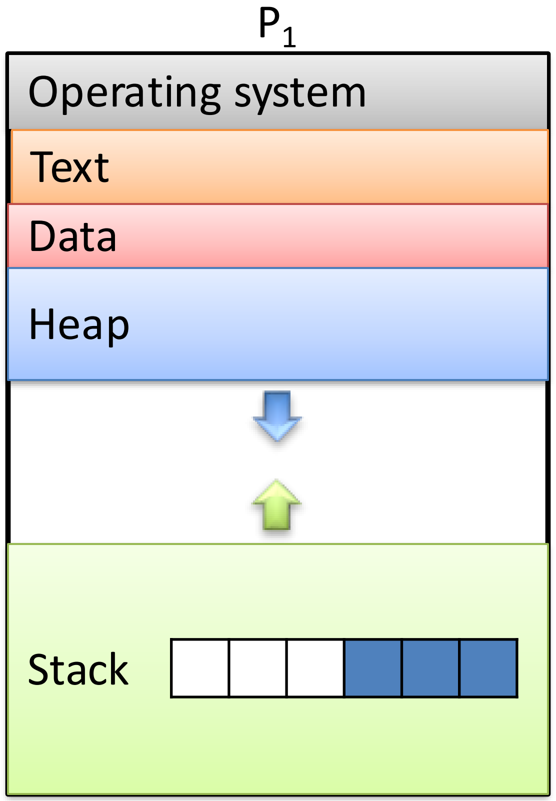
data pointer: the start of location to copy into

count: how many bytes to copy (at most)





Message Passing IPC (Pipe)

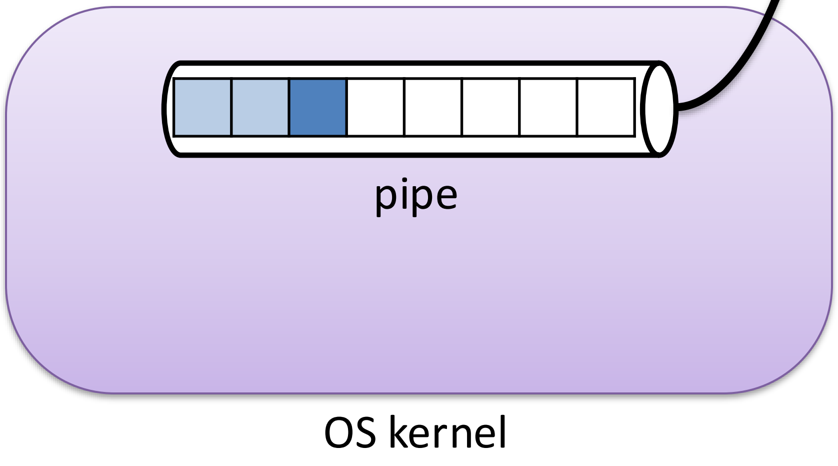


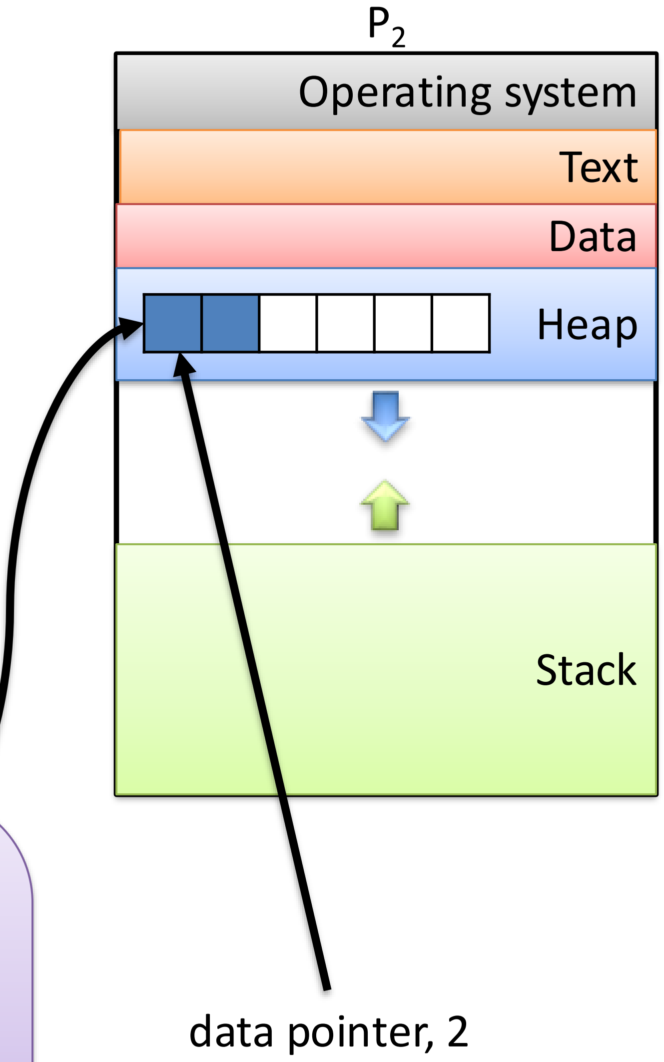
P_2 can receive data from the pipe by calling:

```
read(…, data pointer, count)
```

data pointer: the start of location to copy into

count: how many bytes to copy (at most)

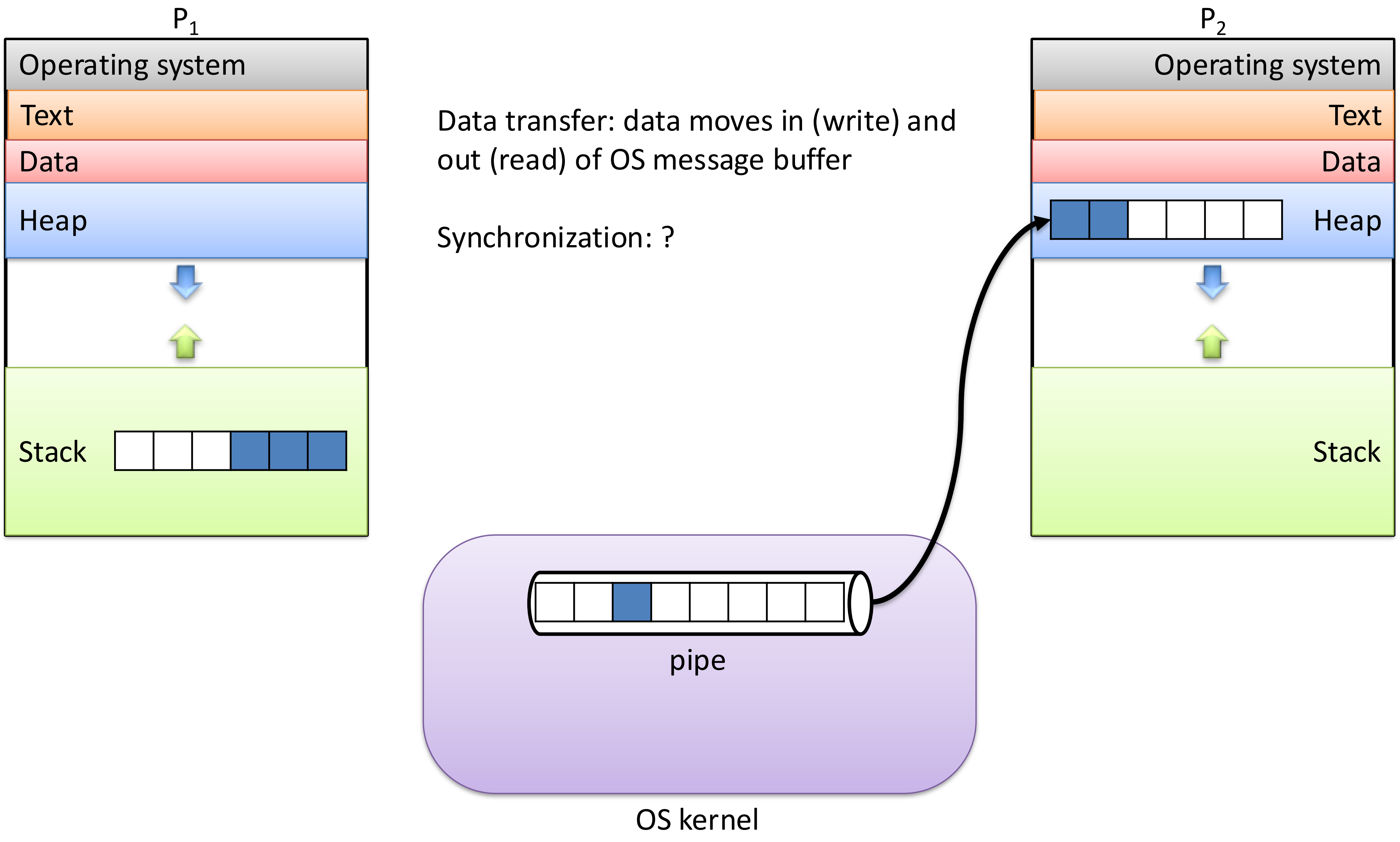




Message Passing IPC (Pipe)

Data transfer: data moves in (write) and out (read) of OS message buffer

Synchronization: ?



Where is the synchronization* in message passing IPC?

(*application synchronization)

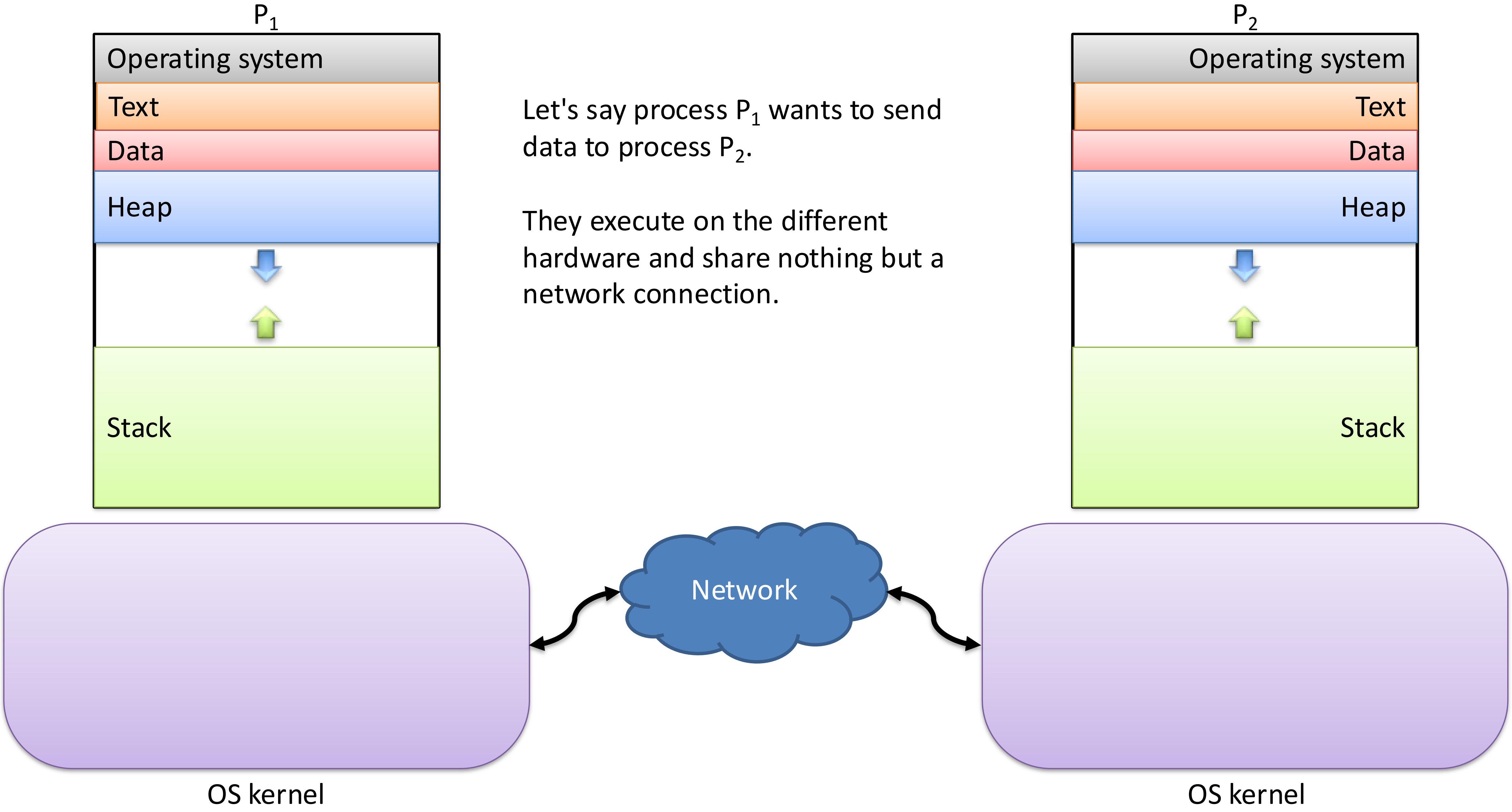
A. The OS adds synchronization.

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.

Message Passing IPC (Socket)



Let's say process P_1 wants to send data to process P_2.

They execute on the different hardware and share nothing but a network connection.

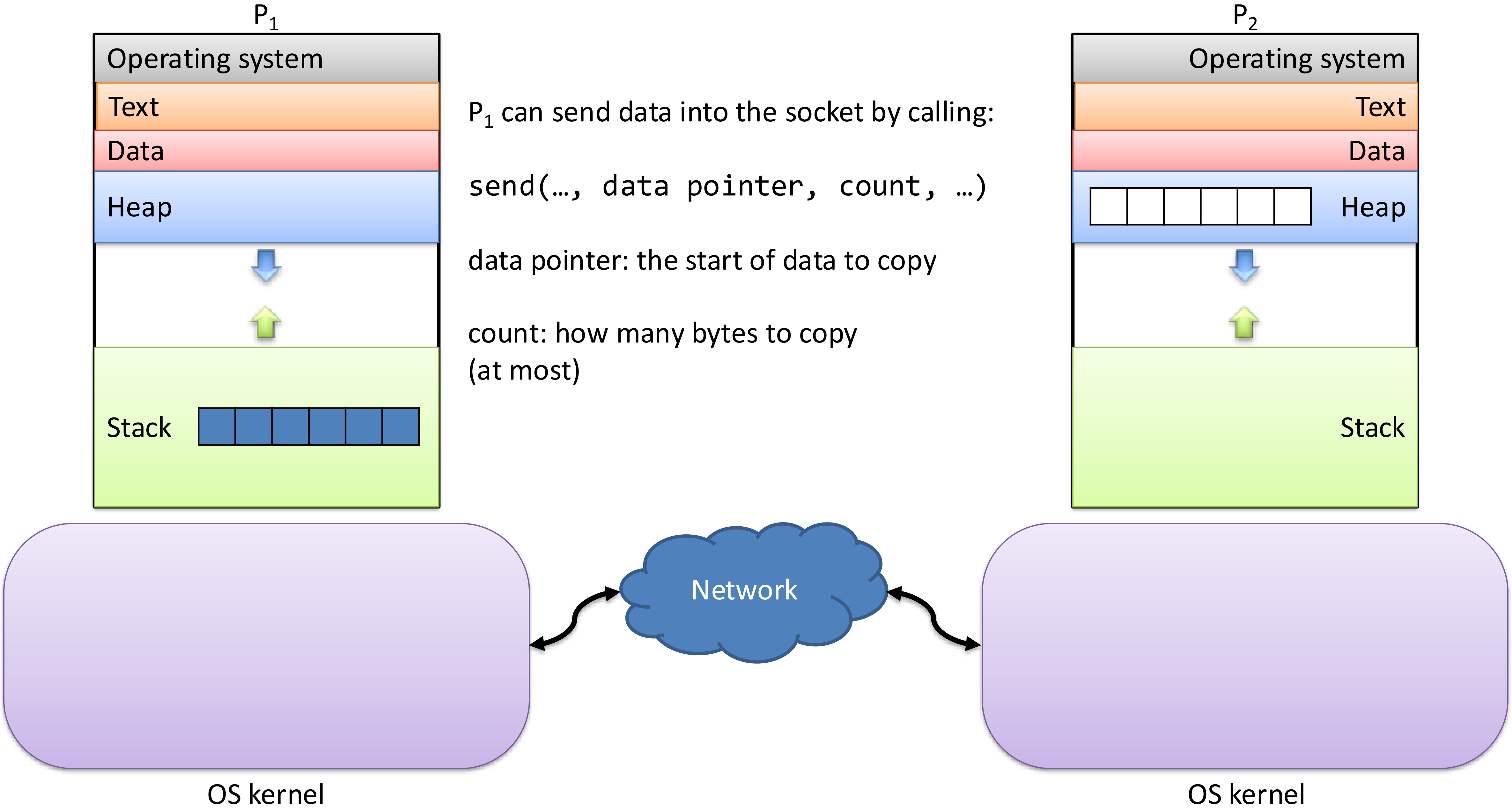
Message Passing IPC (Socket)

P_1 can send data into the socket by calling:

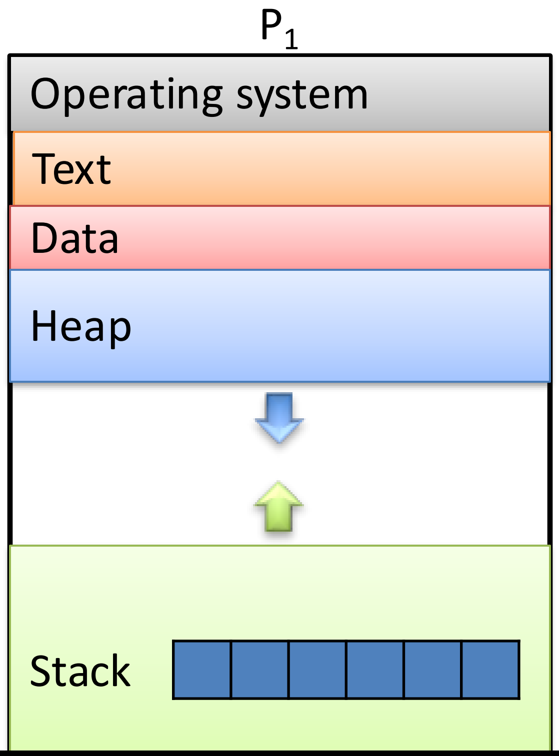
```
send(…, data pointer, count, …)
```

data pointer: the start of data to copy

count: how many bytes to copy
(at most)



Message Passing IPC (Socket)

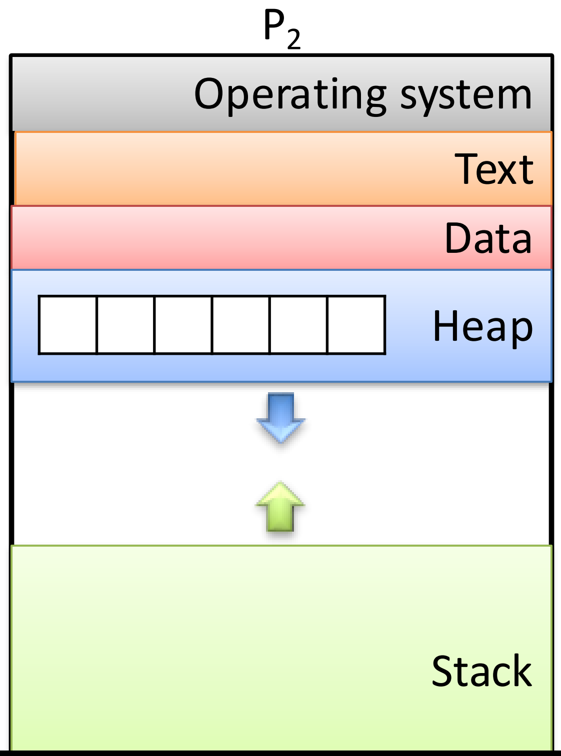


P_1 can send data into the socket by calling:

`send(…, data pointer, count, …)`

data pointer: the start of data to copy

count: how many bytes to copy (at most)



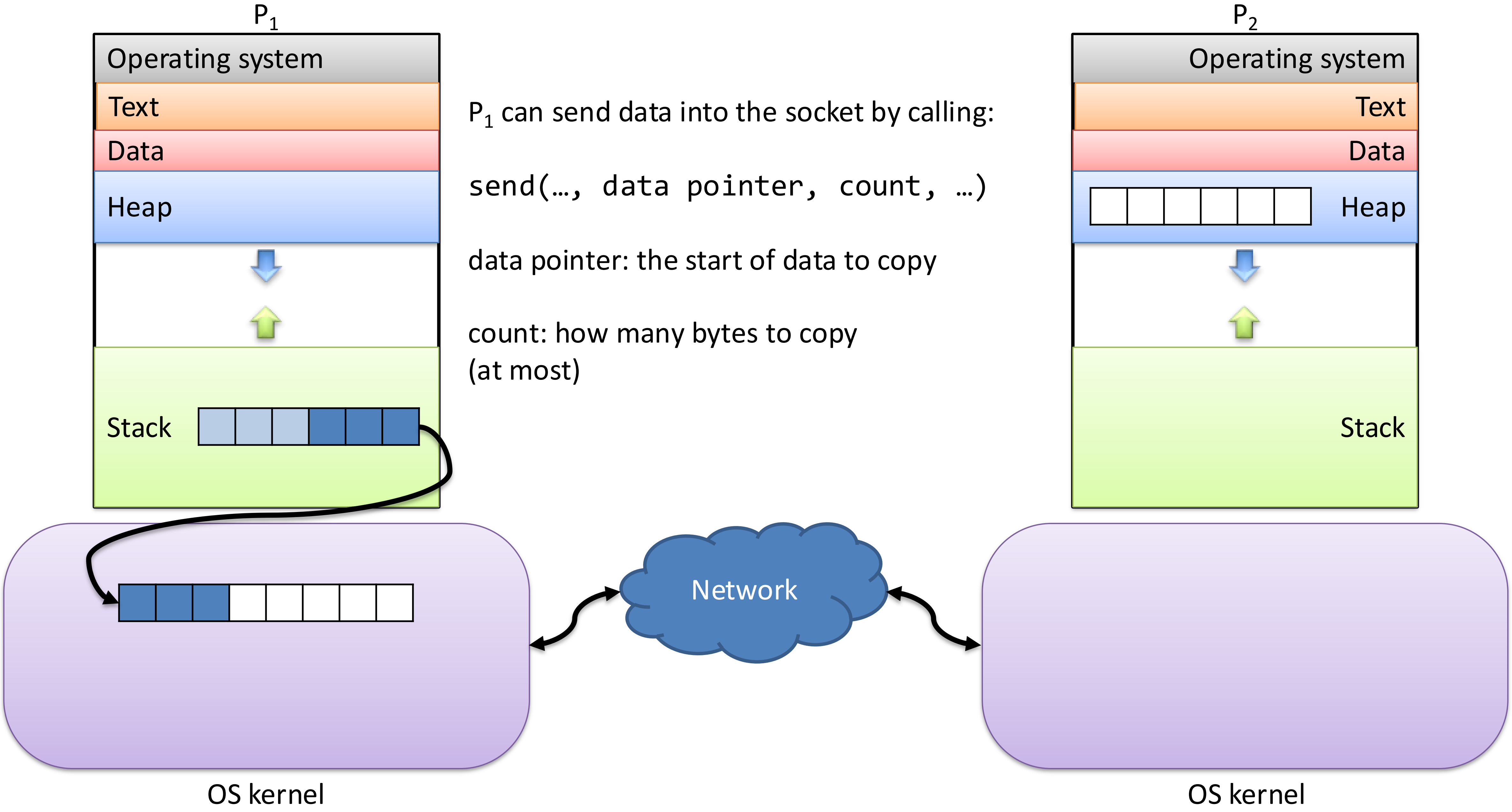

```
NAME
       send, sendto, sendmsg - send a message on a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Message Passing IPC (Socket)

```
NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write()  writes up to count bytes from the buffer starting at buf to
       the file referred to by the file descriptor fd.
```

```
NAME
       send, sendto, sendmsg - send a message on a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

Message Passing IPC (Socket)



P_1 can send data into the socket by calling:

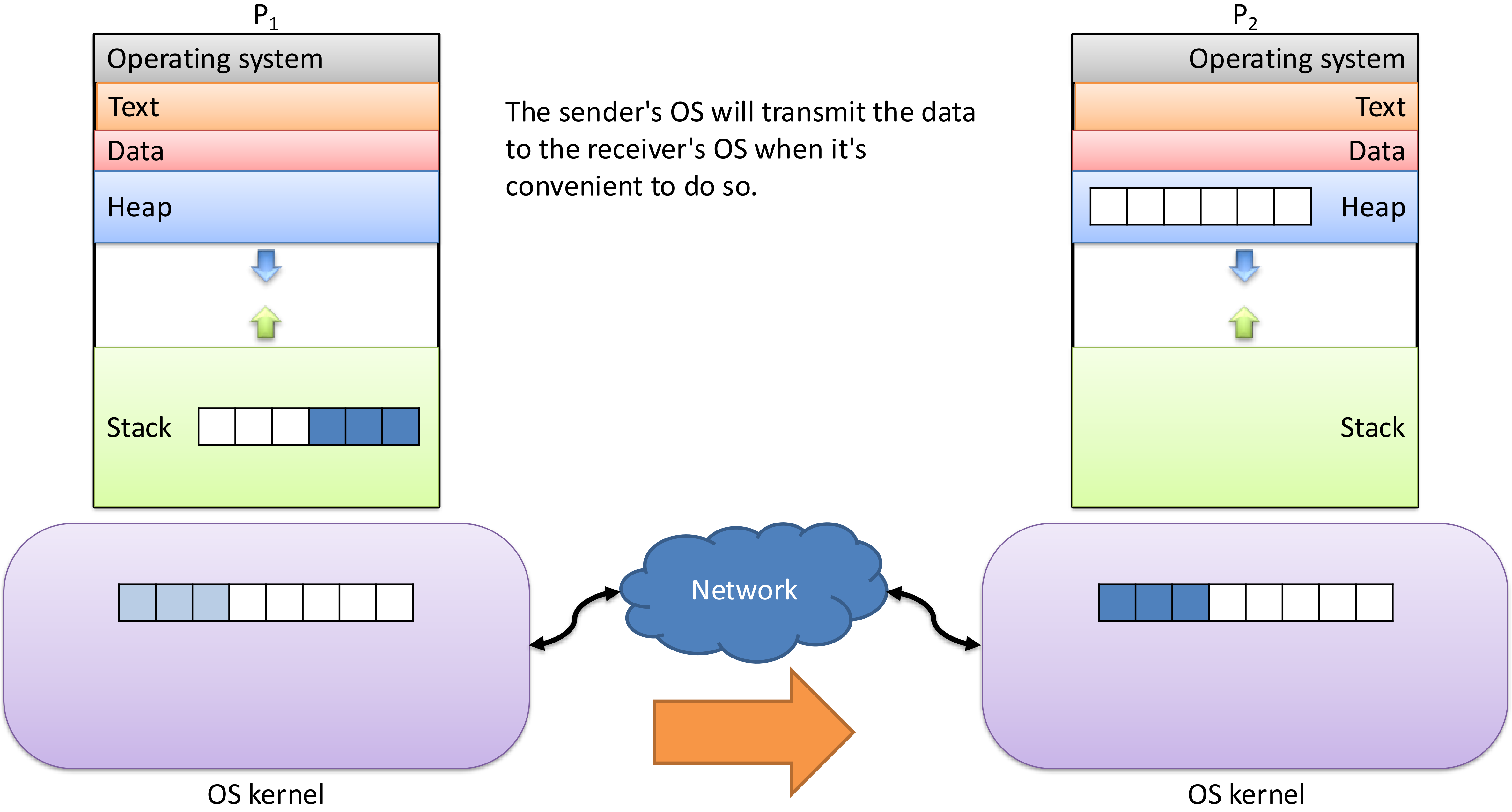
```
send(…, data pointer, count, …)
```

data pointer: the start of data to copy

count: how many bytes to copy (at most)

Message Passing IPC (Socket)

The sender's OS will transmit the data to the receiver's OS when it's convenient to do so.



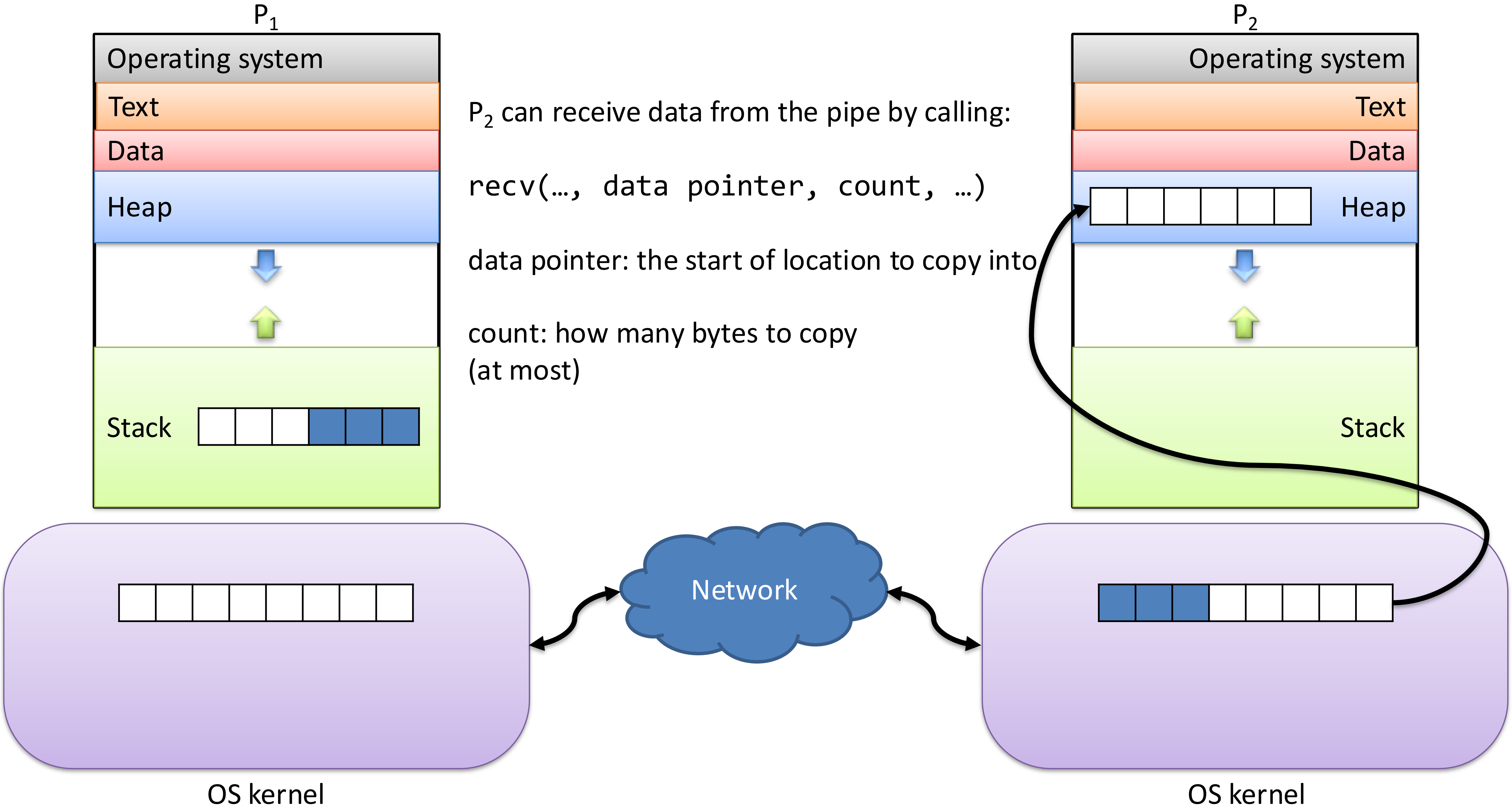
Message Passing IPC (Socket)

P_2 can receive data from the pipe by calling:

`recv(…, data pointer, count, …)`

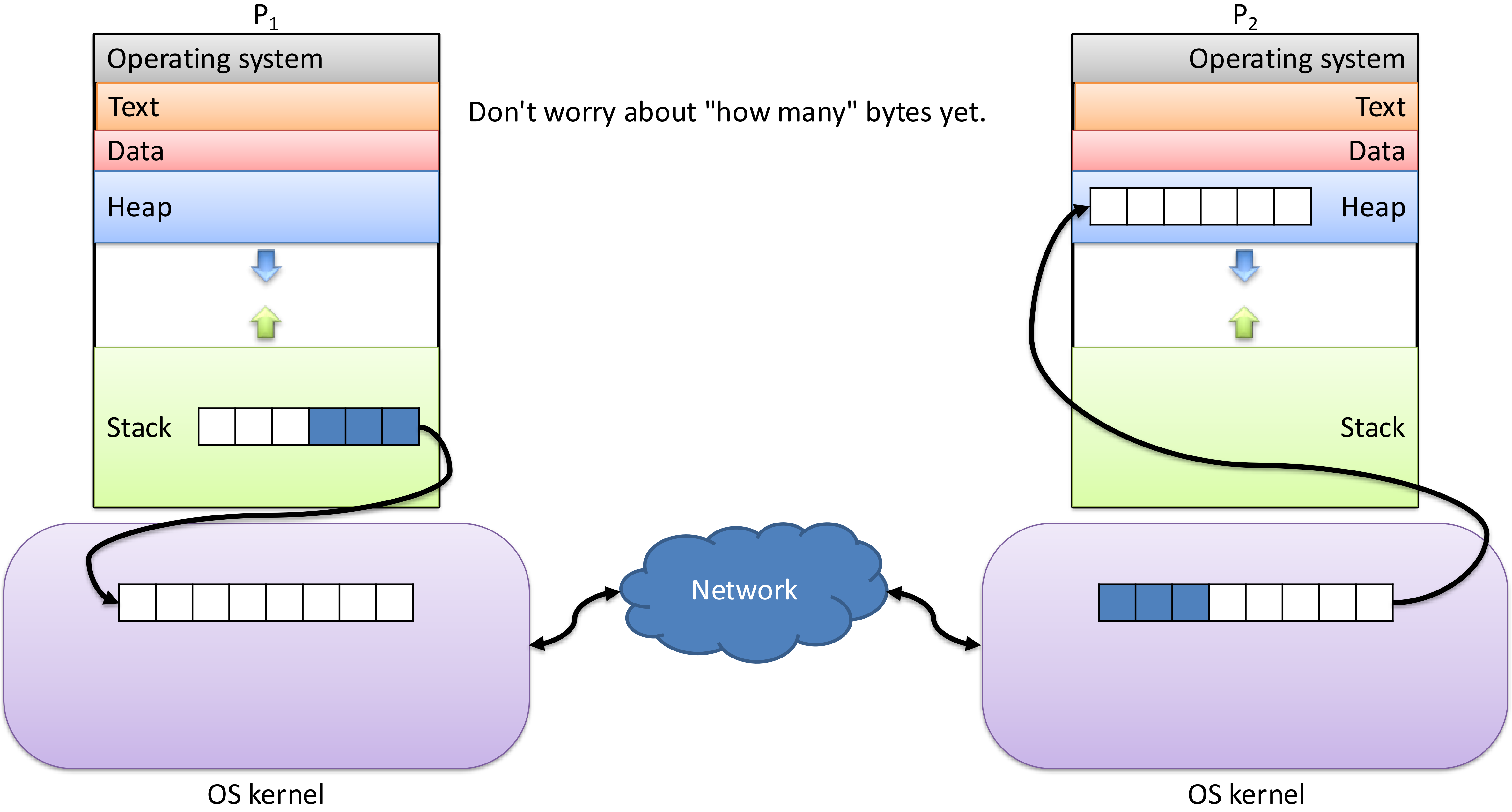
data pointer: the start of location to copy into

count: how many bytes to copy
(at most)

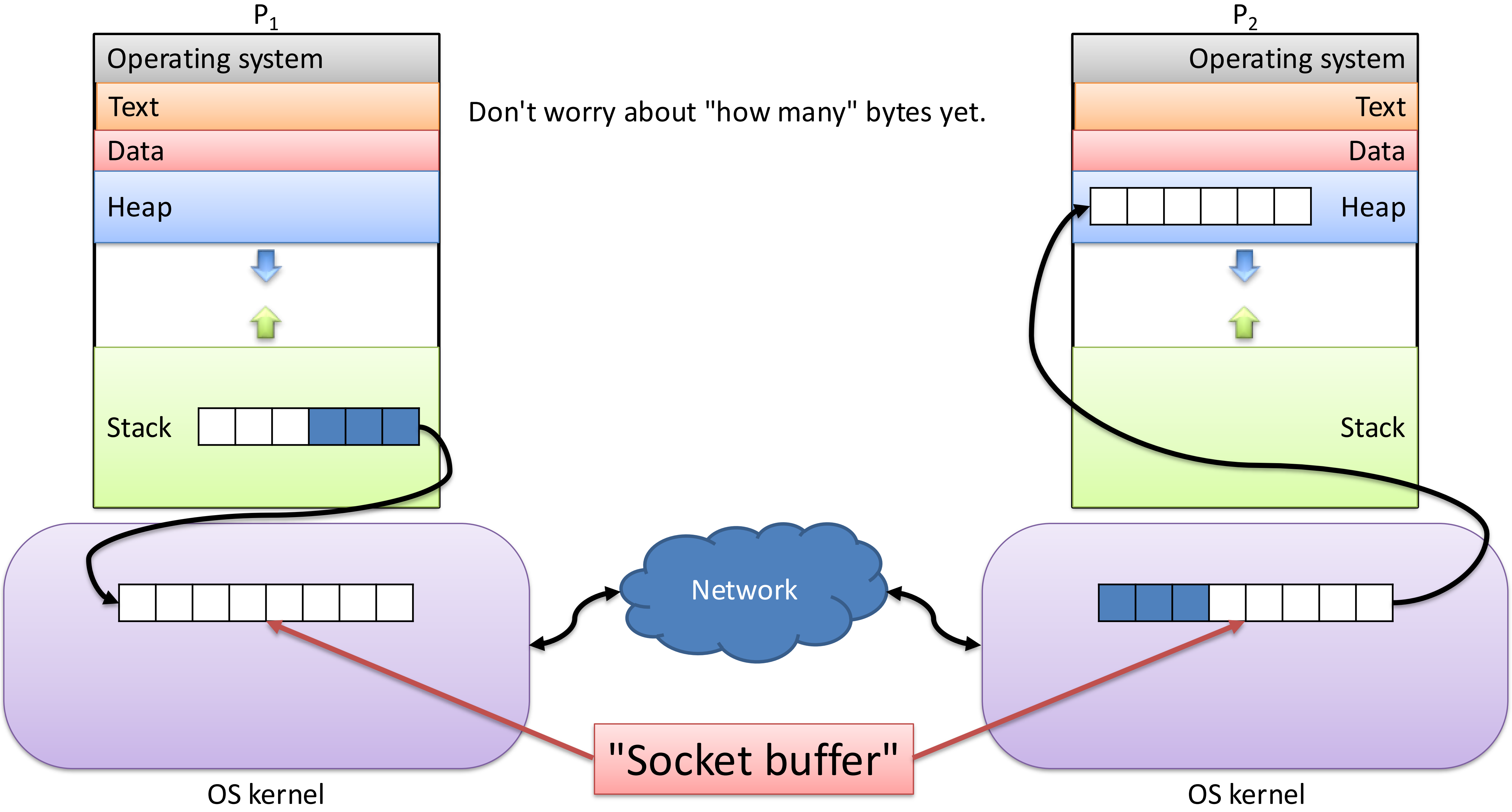


Questions about this model?

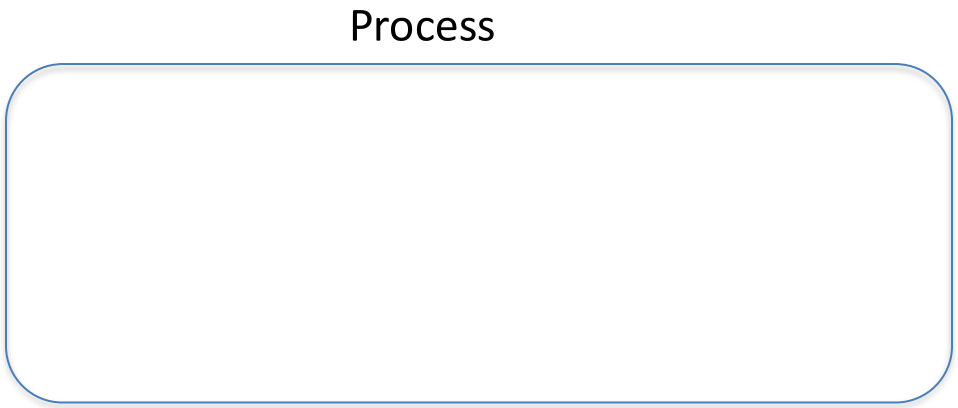
Don't worry about "how many" bytes yet.



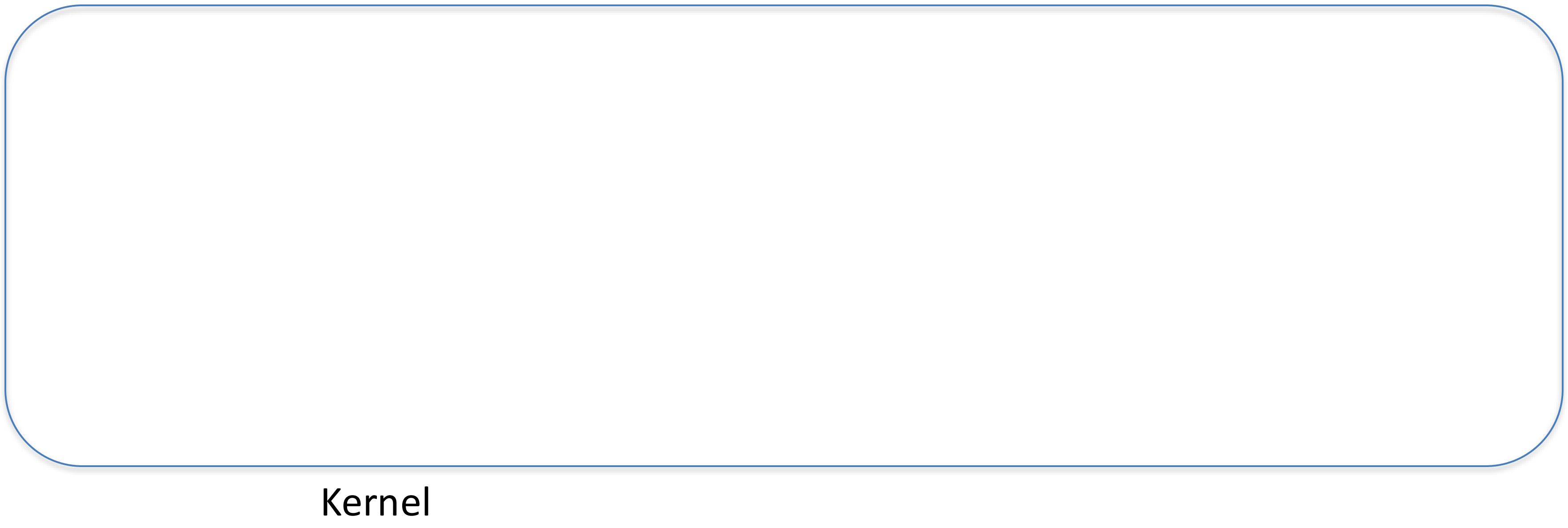
Questions about this model?



Descriptor Table



- OS stores a table, per process, of descriptors



Descriptors

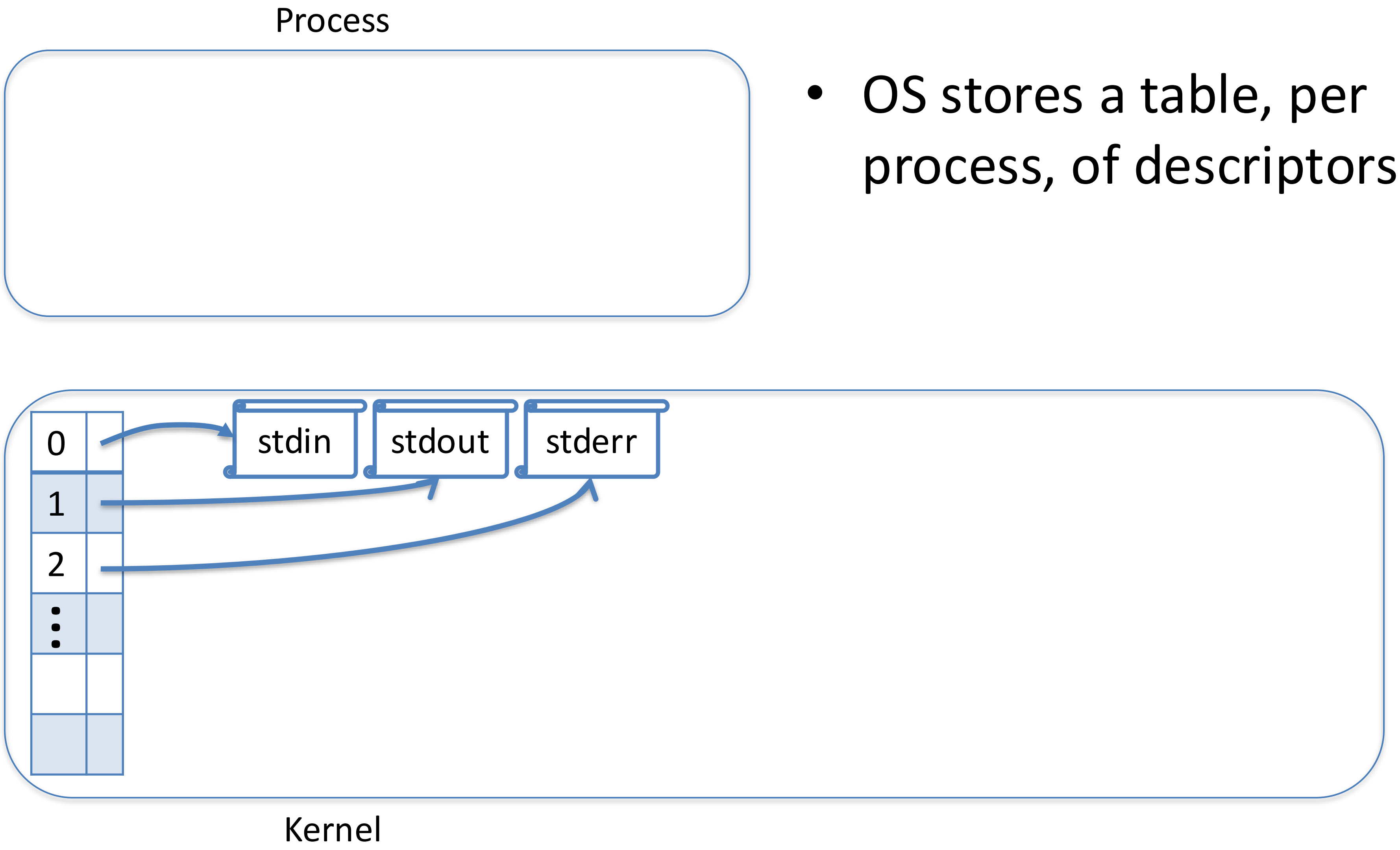
Where do descriptors come from?

What are they?

```
OPEN(2)                    Linux Programmer's Manual
          OPEN(2)

NAME
       open, openat, creat - open and possibly create a file

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>
       #include <fcntl.h>

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

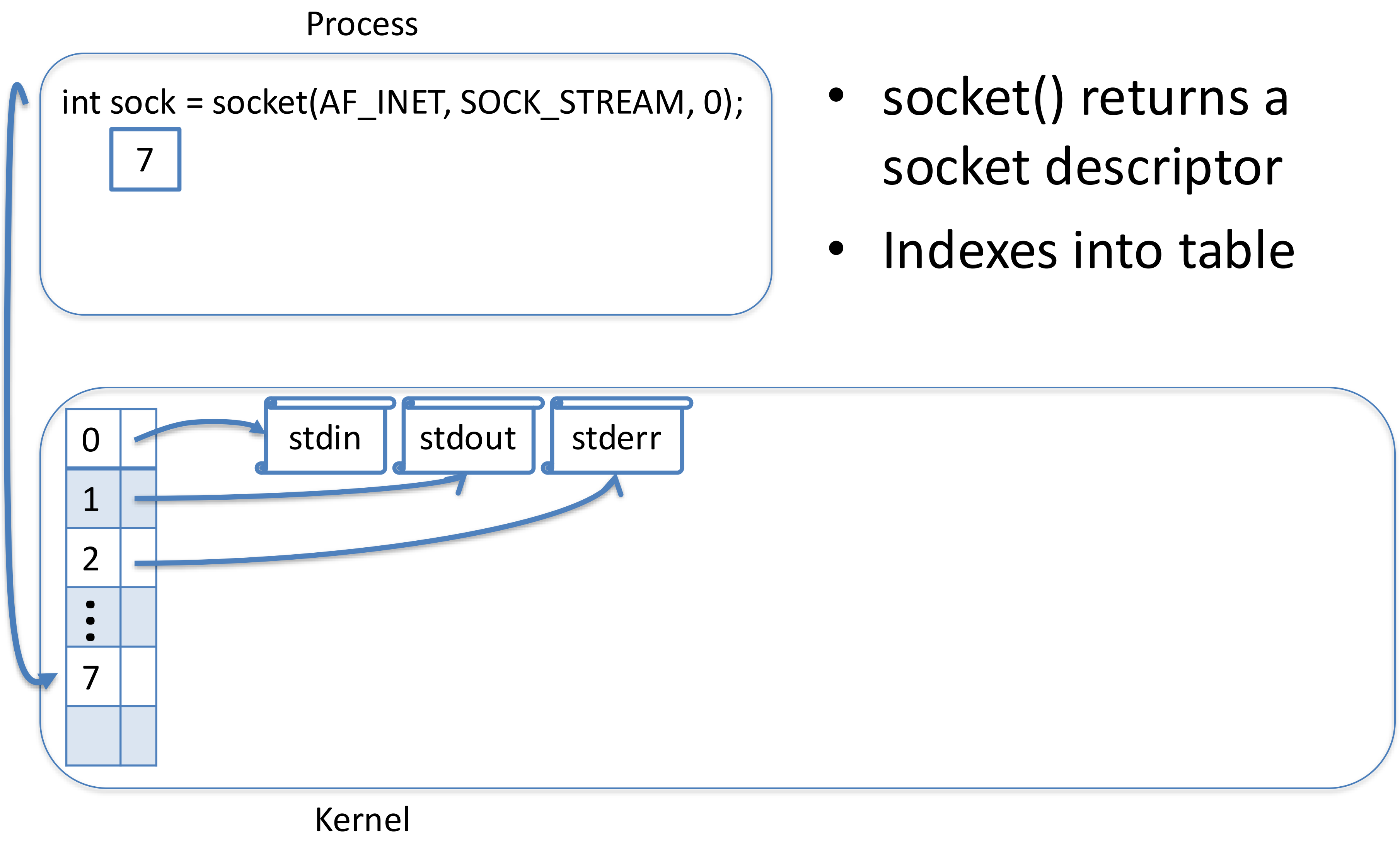
```
SOCKET(2)          Linux Programmer's Manual          SOCKET(2)

NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>          /* See NOTES */
       #include <sys/socket.h>

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

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

Descriptor Table



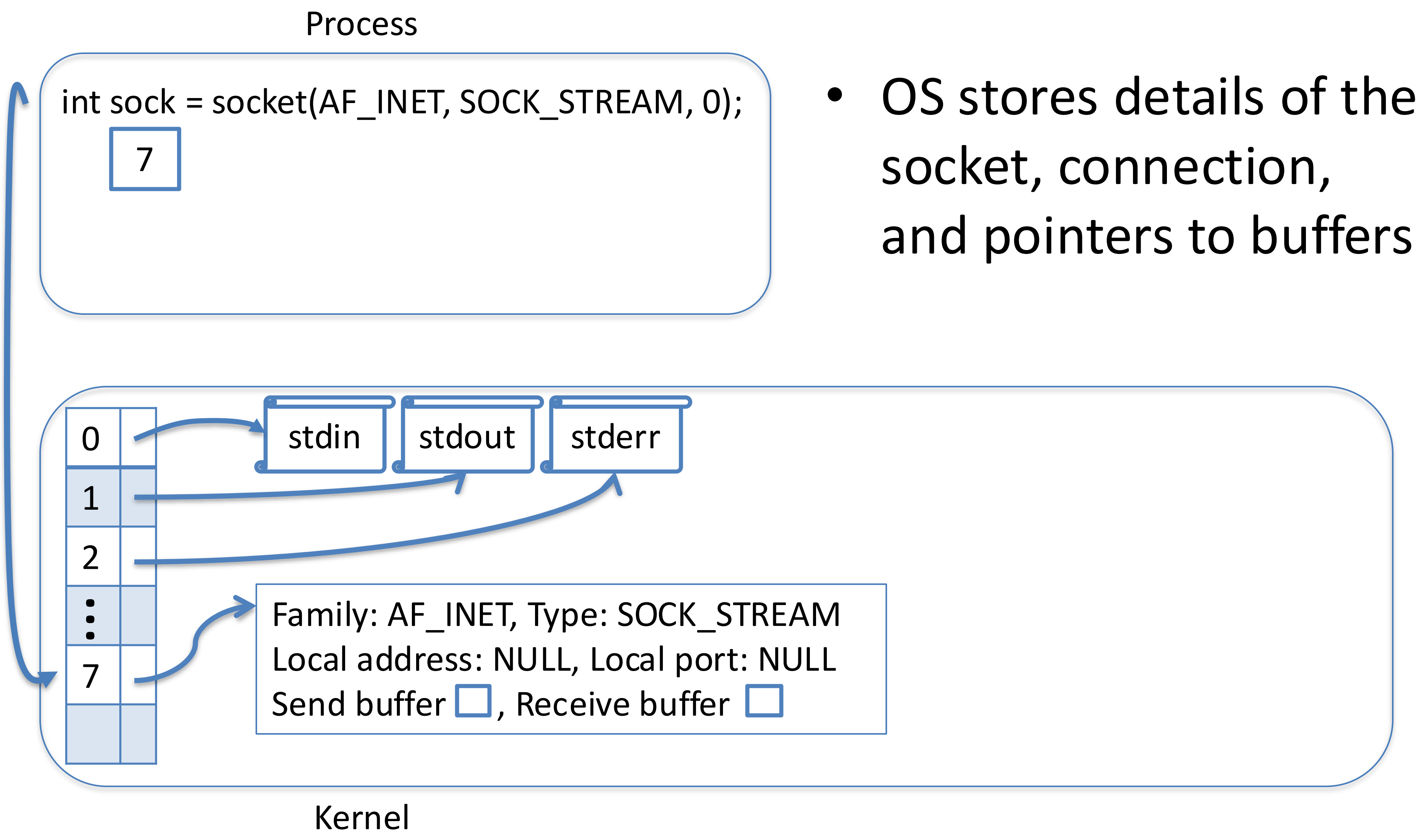
- OS stores a table, per process, of descriptors

socket()



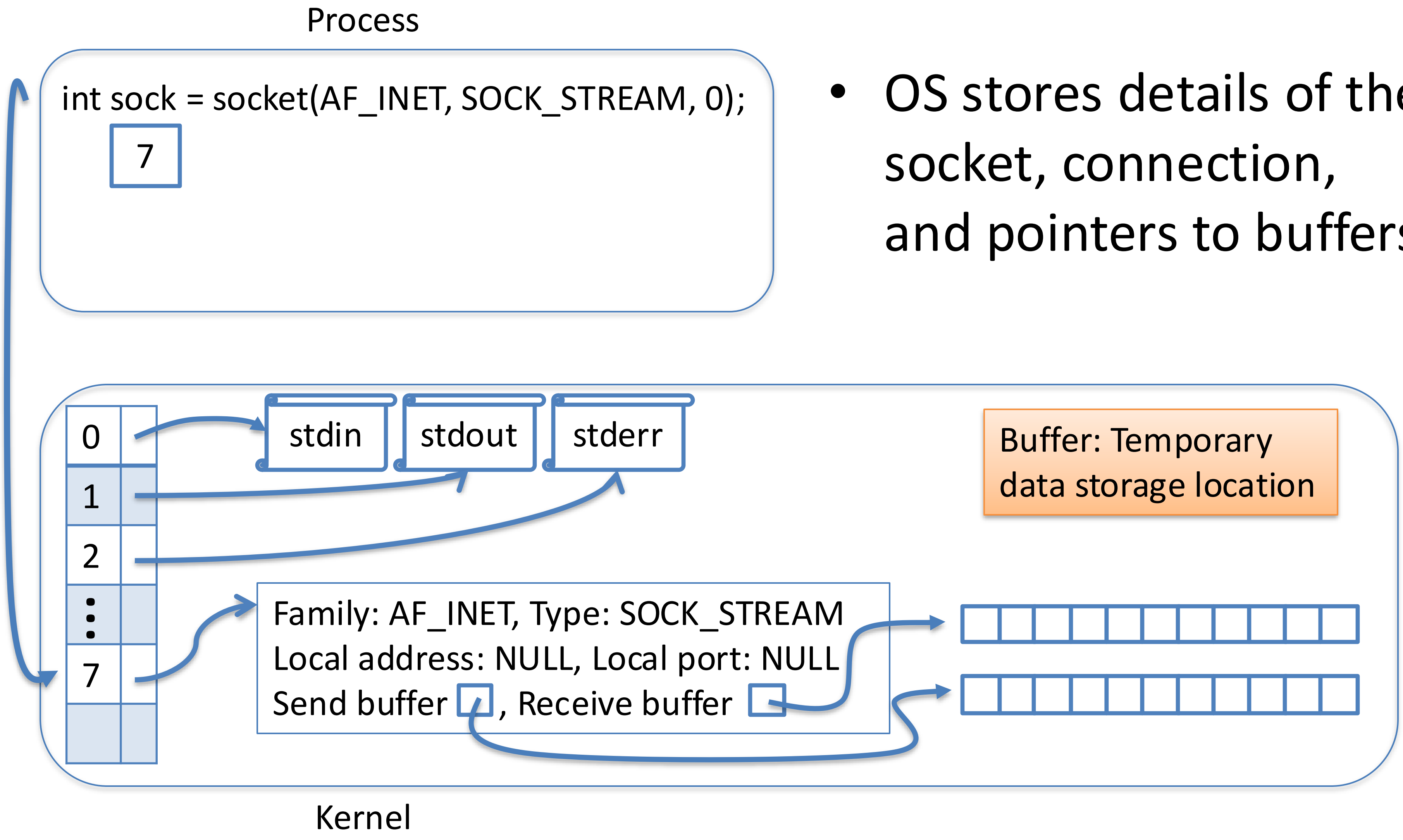
- socket() returns a socket descriptor
- Indexes into table

socket()



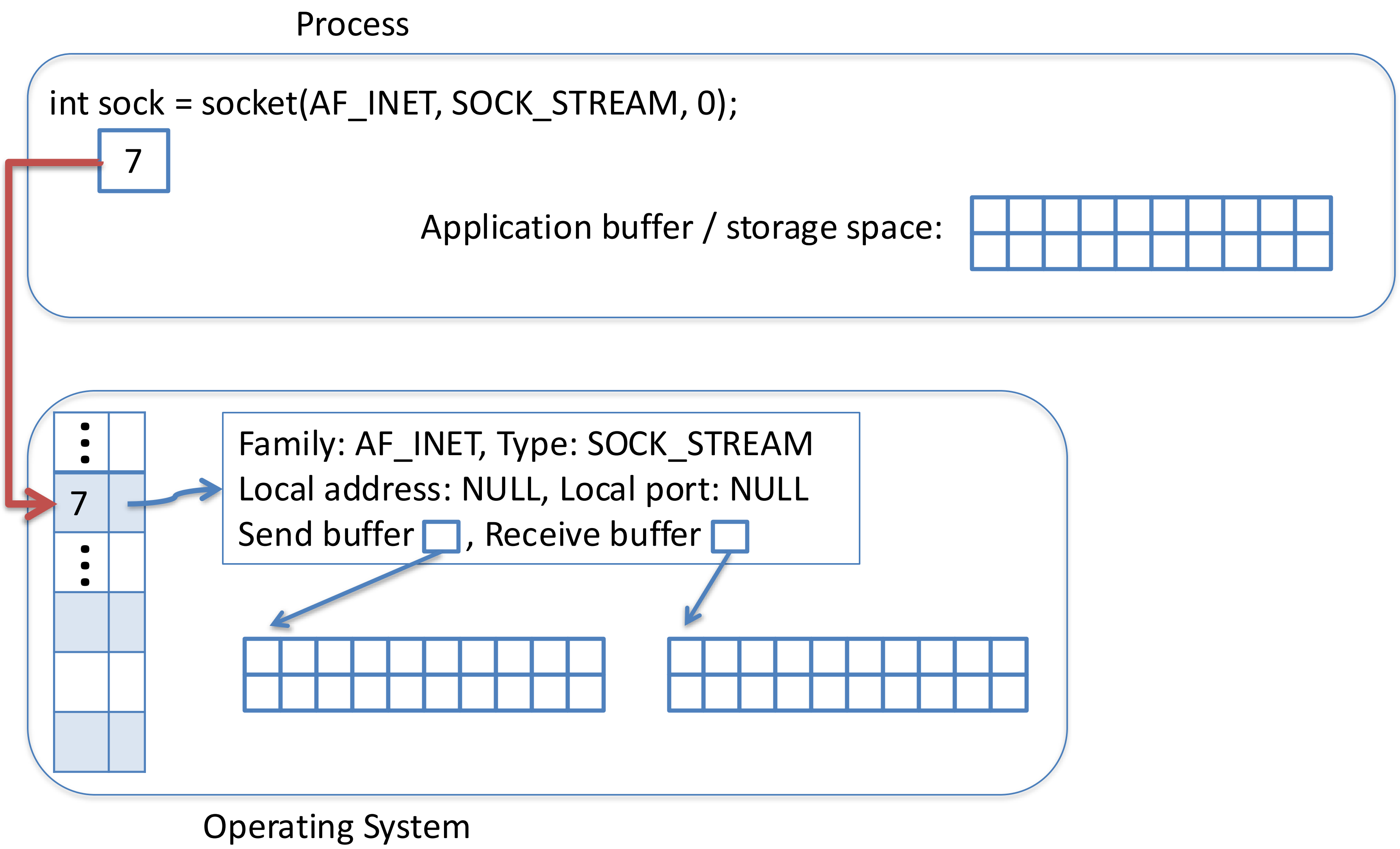
- OS stores details of the socket, connection, and pointers to buffers

socket()

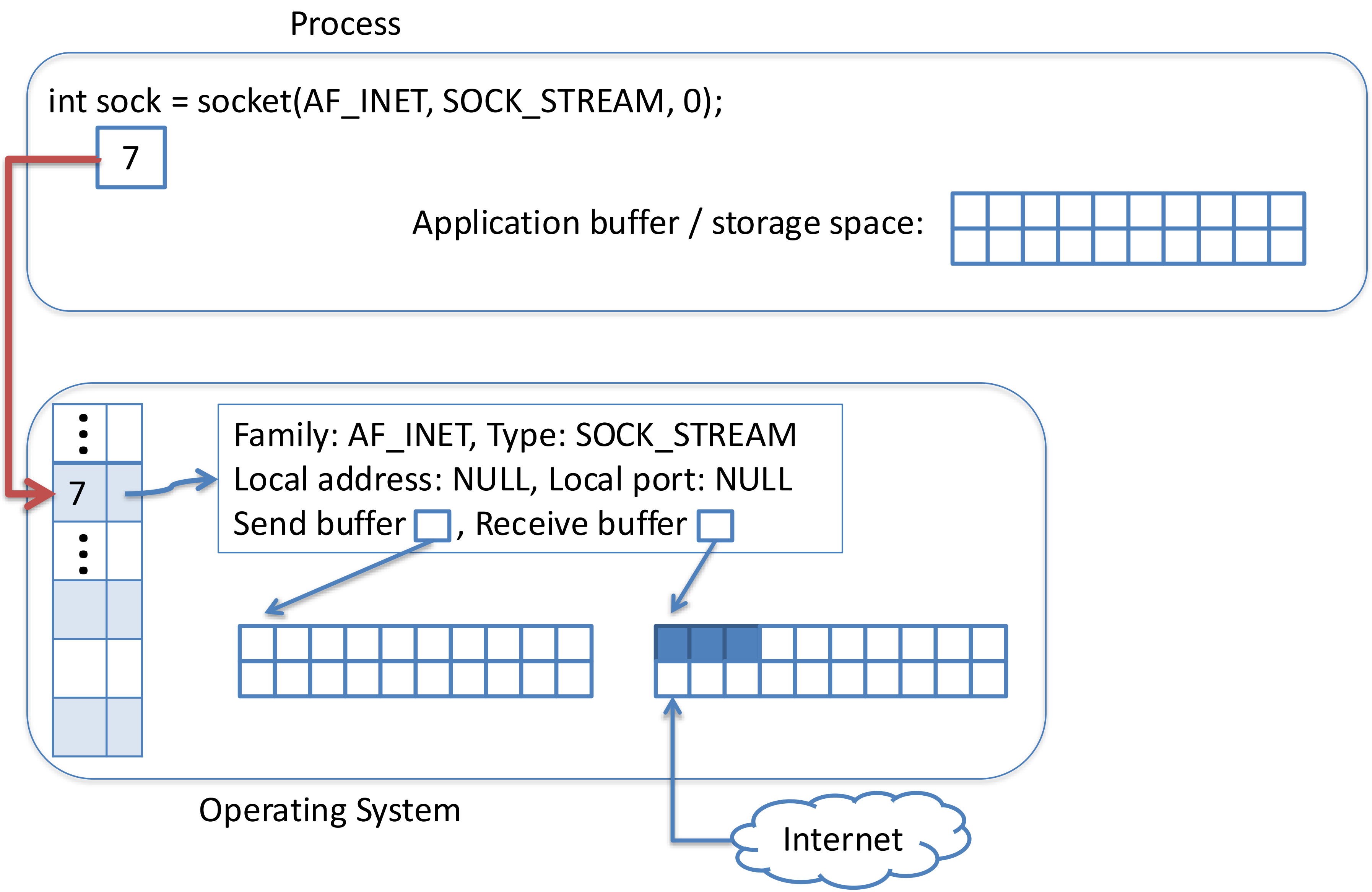


- OS stores details of the socket, connection, and pointers to buffers

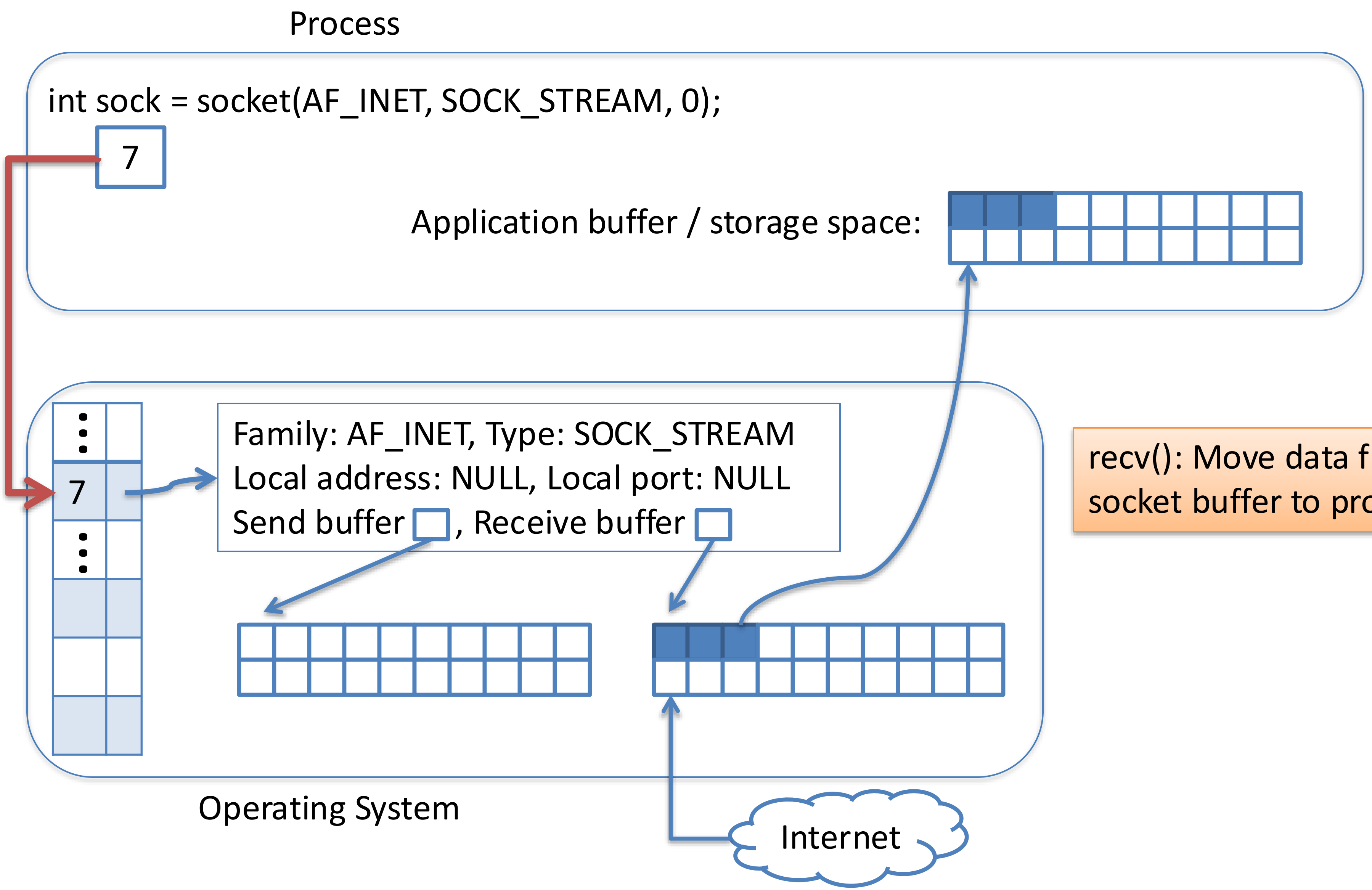
Socket Buffers



Socket Buffers

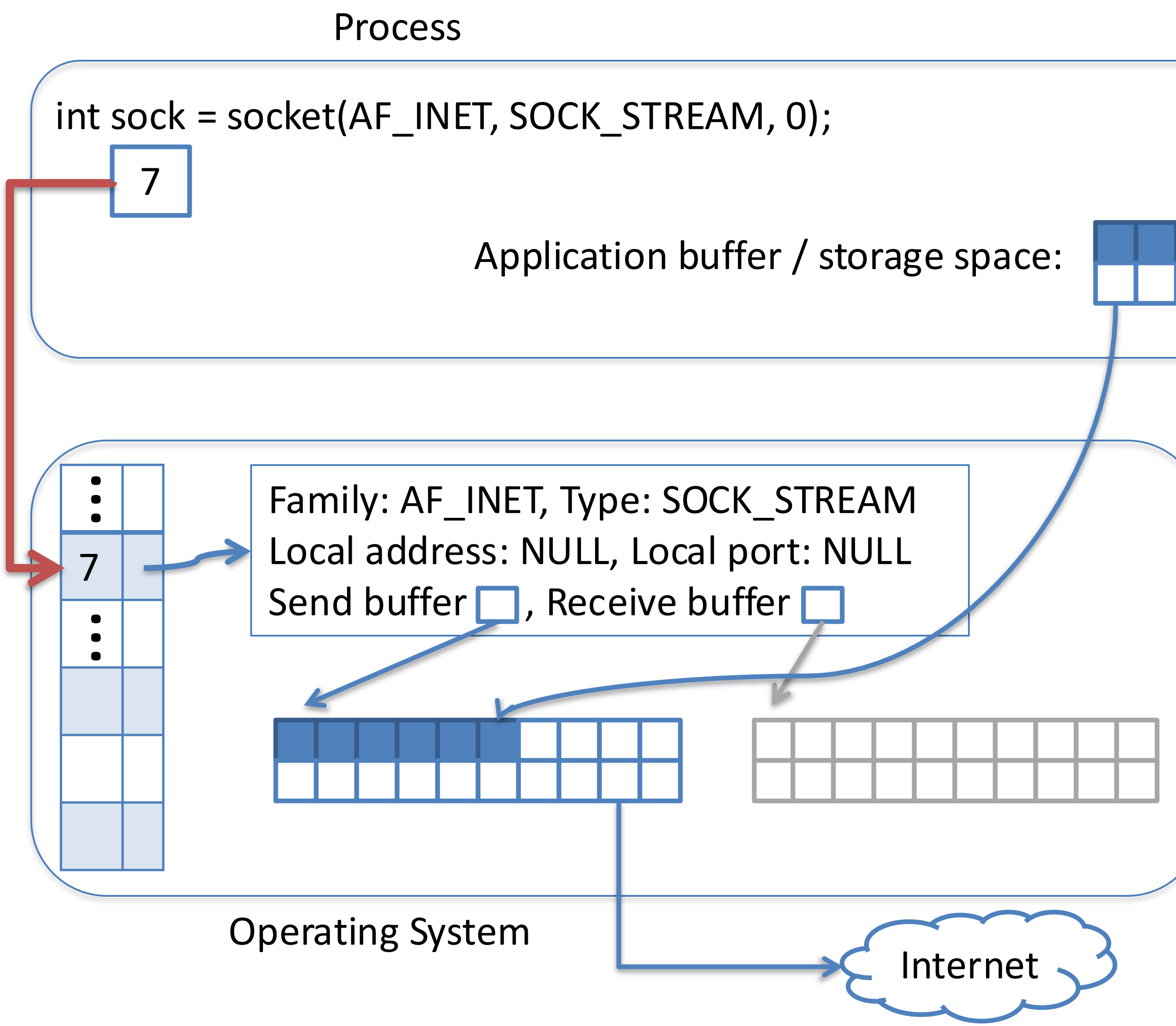


Socket Buffers



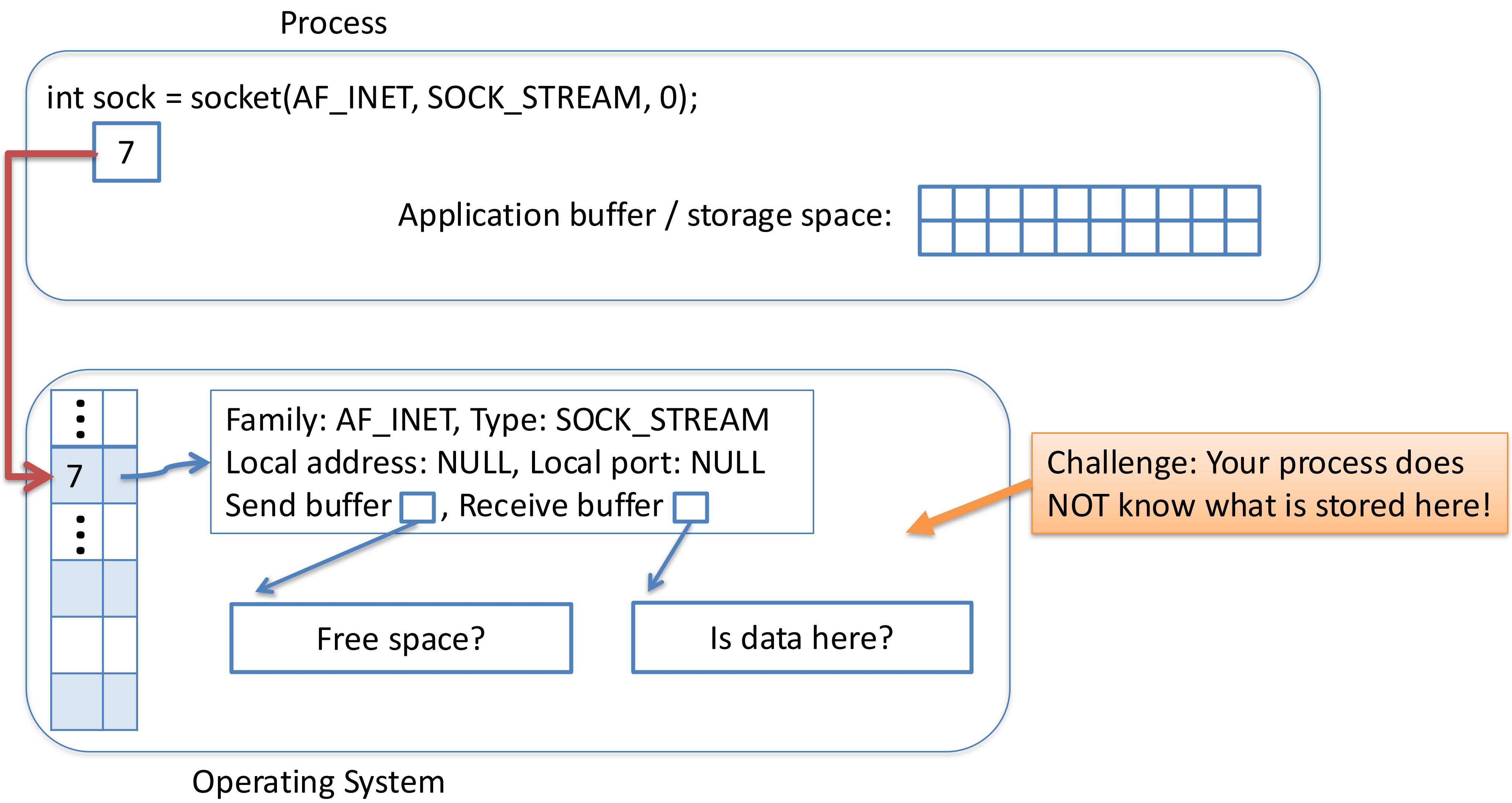
recv(): Move data from socket buffer to process.

Socket Buffers

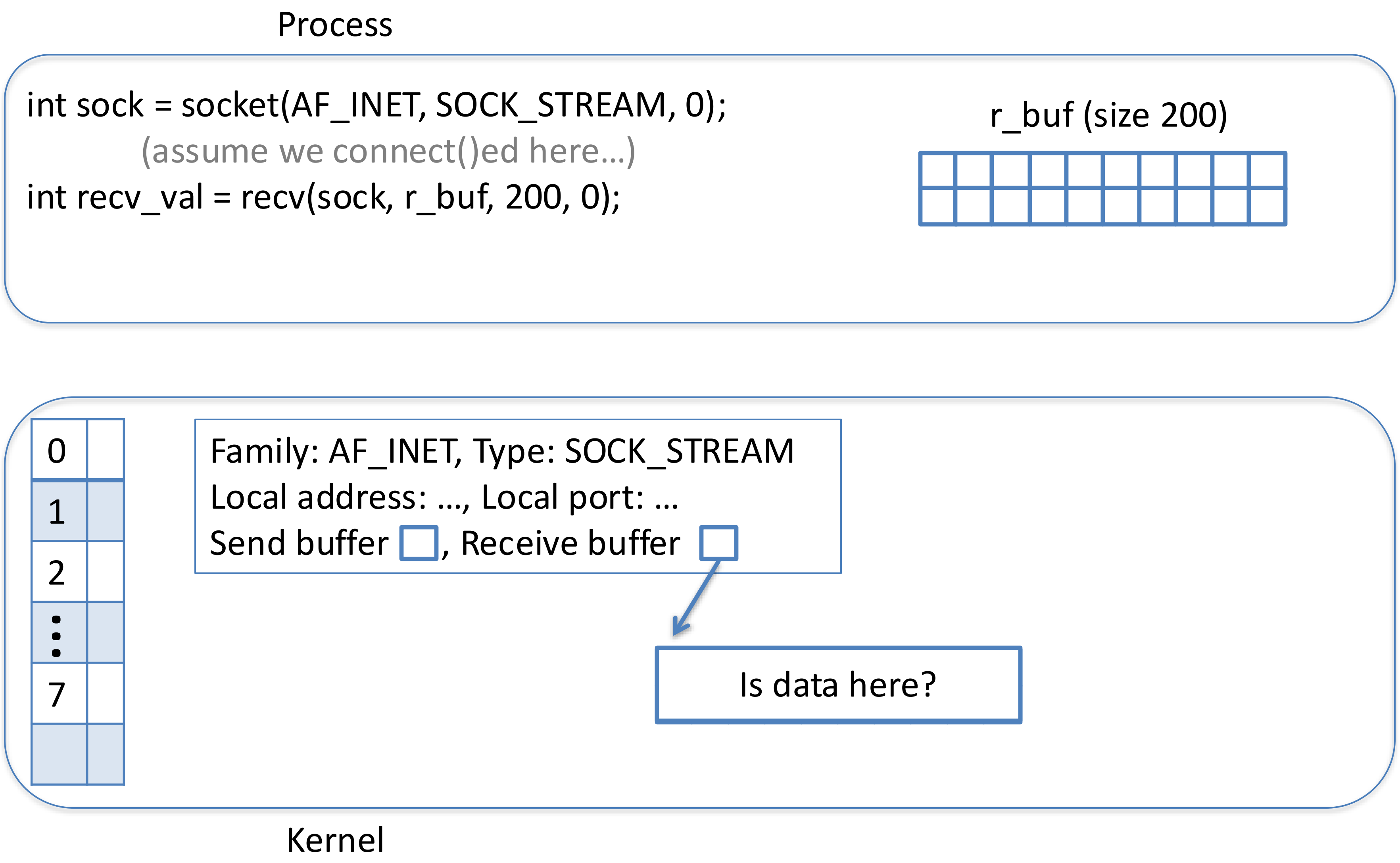


send(): Move data from process to socket buffer

Socket Buffers



recv()

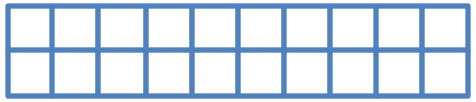

Process
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
r_buf (size 200)
0
1
2
7
Family: AF_INET, Type: SOCK_STREAM
Local address: …, Local port: …
Send buffer , Receive buffer
Is data here?
Kernel

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

Process

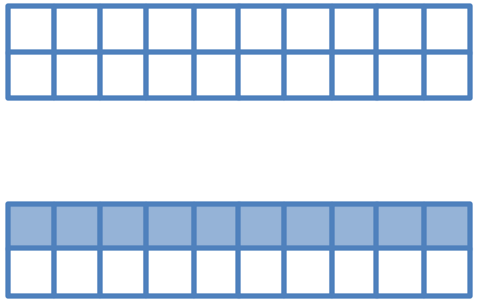
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

Socket buffer (receive)



Empty

100 bytes

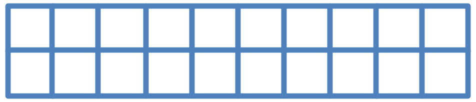
Kernel

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

Process

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

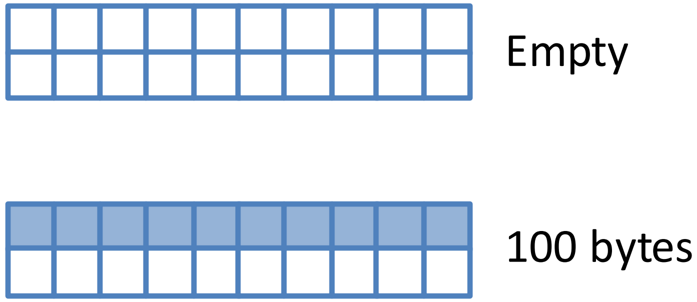
r_buf (size 200)



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 (receive)



Kernel

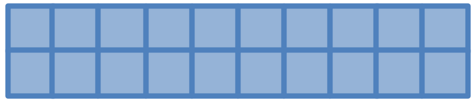
"Block" means pause the calling process.

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

Process

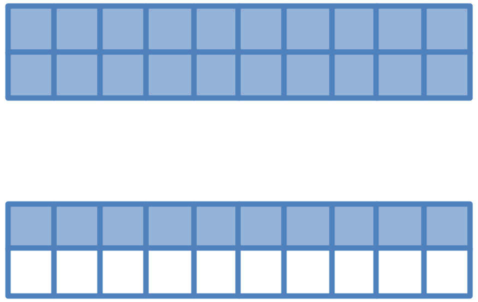
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int send_val = send(sock, s_buf, 200, 0);
```

s_buf (size 200)



Two Scenarios:

Socket buffer (send)



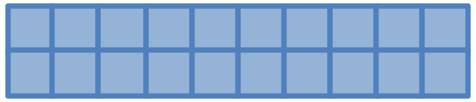
Kernel

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

Process

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
          (assume we connect()ed here…)
int send_val = send(sock, s_buf, 200, 0);
```

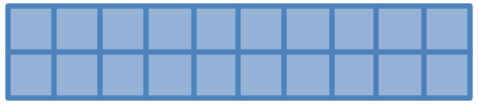
s_buf (size 200)



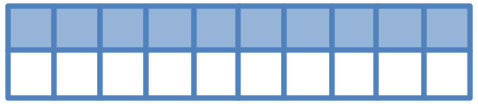
	Full	100 Bytes
A	**Return 0**	**Copy 100 bytes**
B	**Block**	**Copy 100 bytes**
C	**Return 0**	**Block**
D	**Block**	**Block**
E	**Something else**	

Two Scenarios:

Socket buffer (send)



Full



100 bytes

Kernel

Blocking Implications

- 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!*

- 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.

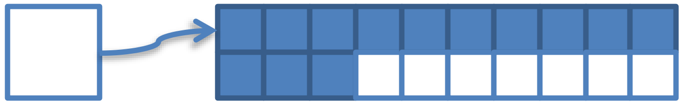
ALWAYS check send() return value!

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

Data sent: 0
Data to send: 130

send(sock, data, 130, 0);

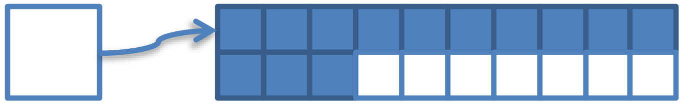
Data:



ALWAYS check send() return value!

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

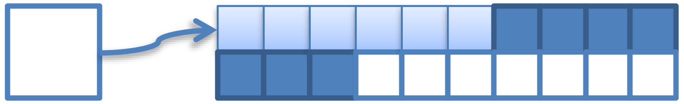
Data sent: 0
Data to send: 130

Data: 

60

send(sock, data, 130, 0);

Data sent: 60
Data to send: 130

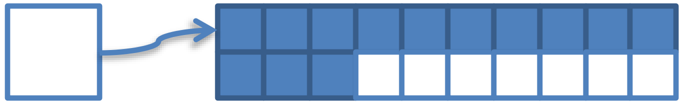
Data: 

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:

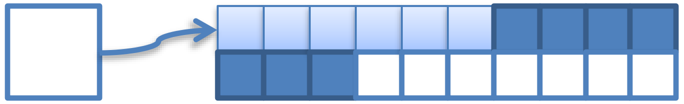


60

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

Data sent: 60
Data to send: 130

Data:



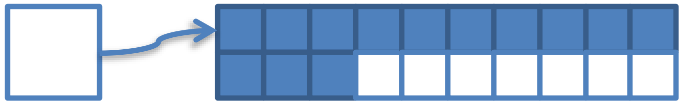
```
// Copy the 70 bytes starting from offset 60.
send(sock, data + 60, 130 - 60, 0);
```

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:

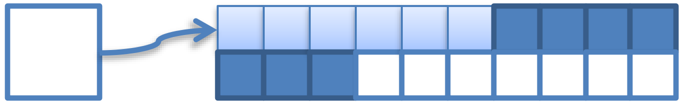


60

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!

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()

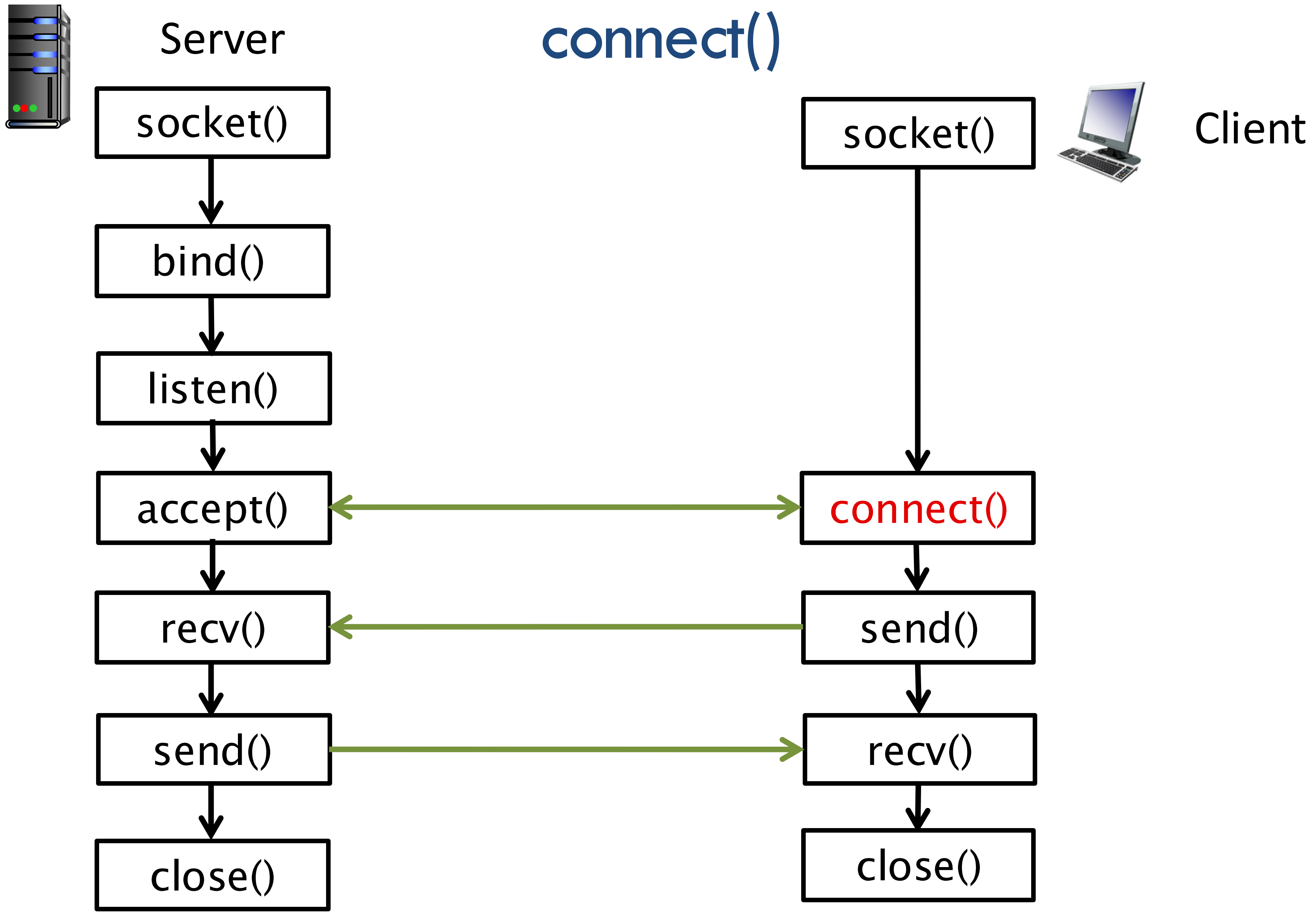
```
int socket(int domain, int type, int protocol)

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()
Client
socket()
connect()
send()
recv()
close()

connect()

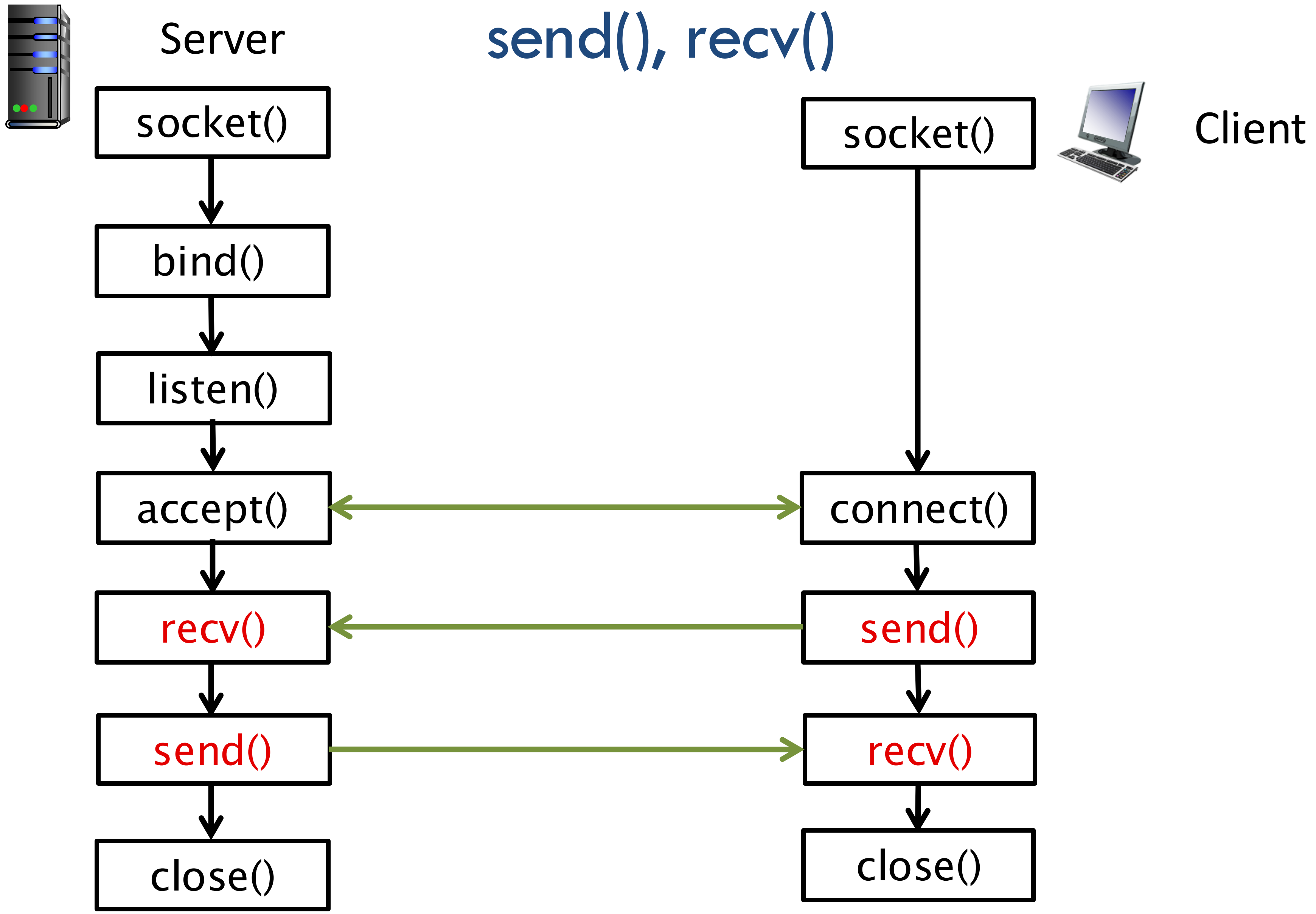
- Before you can communicate, a connection must be established.
- Client Initiates, Server waits.
- Once connect() returns, socket is connected and we can proceed with send(), recv()

```
int connect(int socket, const struct sockaddr
*foreign_address, socklen_t address_length)
```

connect()

```
int connect(int socket, const struct sockaddr
*foreign_address, socklen_t address_length)

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():

```
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.