CS 43: Computer Networks Structure, Threading, and Blocking

Kevin Webb

Swarthmore College

February 3, 2022

Announcements

Agenda

- Under-the-hood look at networking system calls
  - Data buffering and blocking
- Processes, threads, and concurrency models
- Event-based, non-blocking I/O

Motivation: What is the goal of a network?

- Allow devices communicate with one another and coordinate their actions to work together.

(This was a slide from day 1)

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

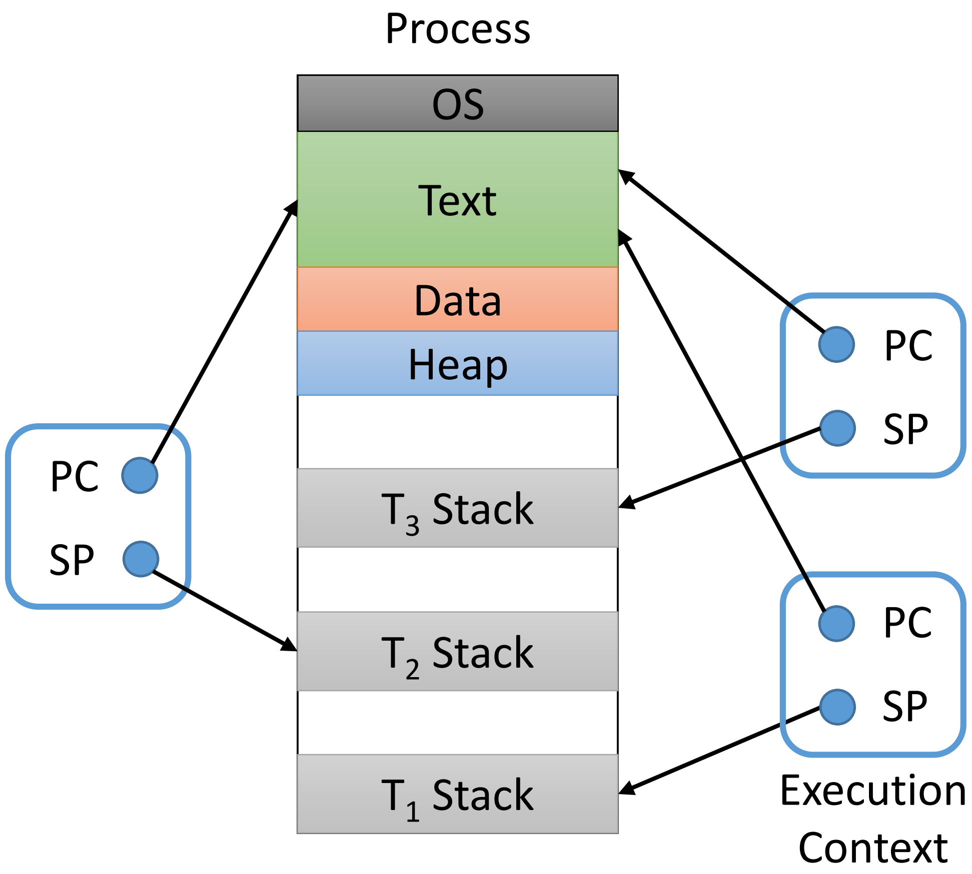
- Broadly, these fall into two categories:
  1. Shared memory
  2. Message passing

Only works on one computer (shared hardware).

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

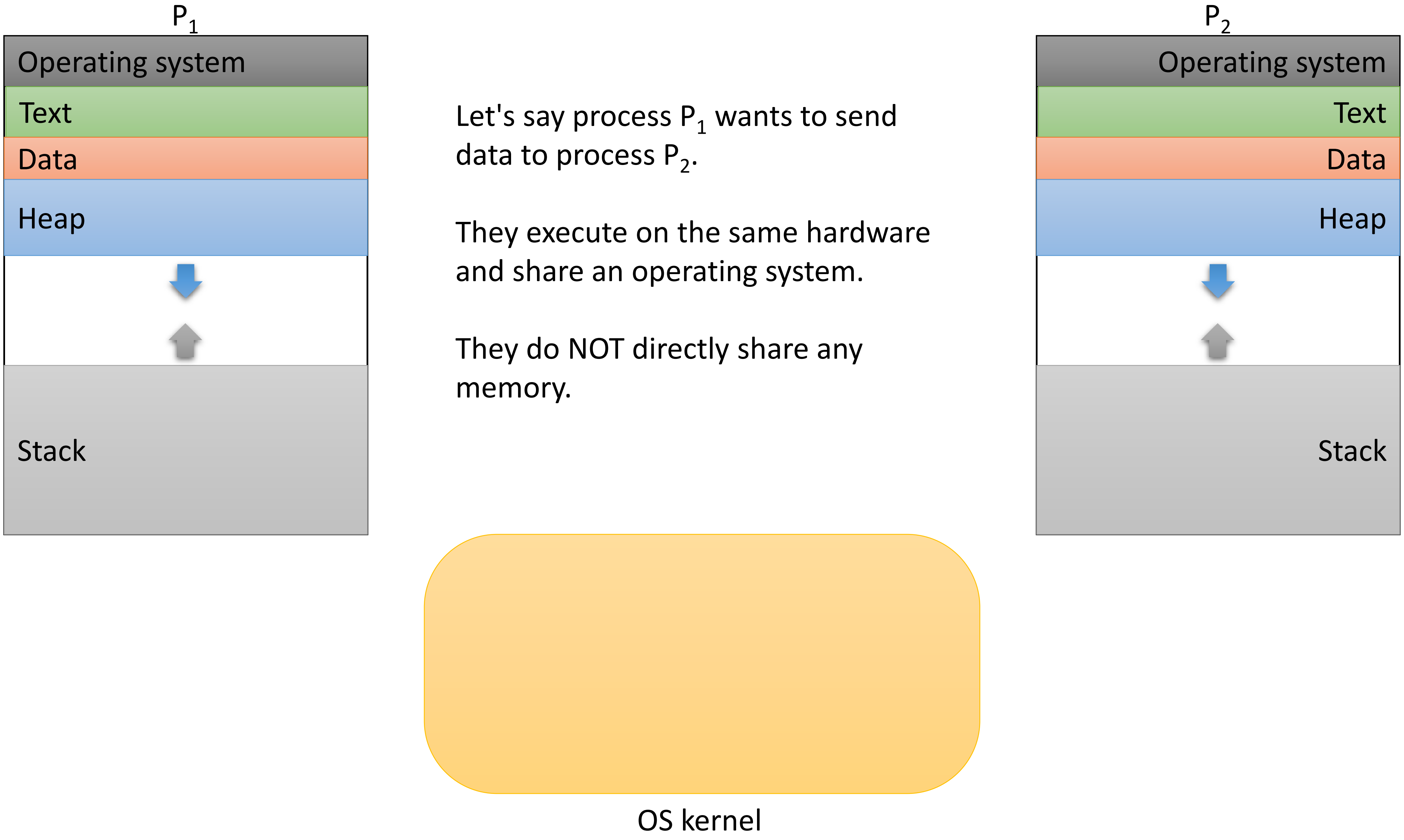
Thread Model (Shared Memory)

- 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)

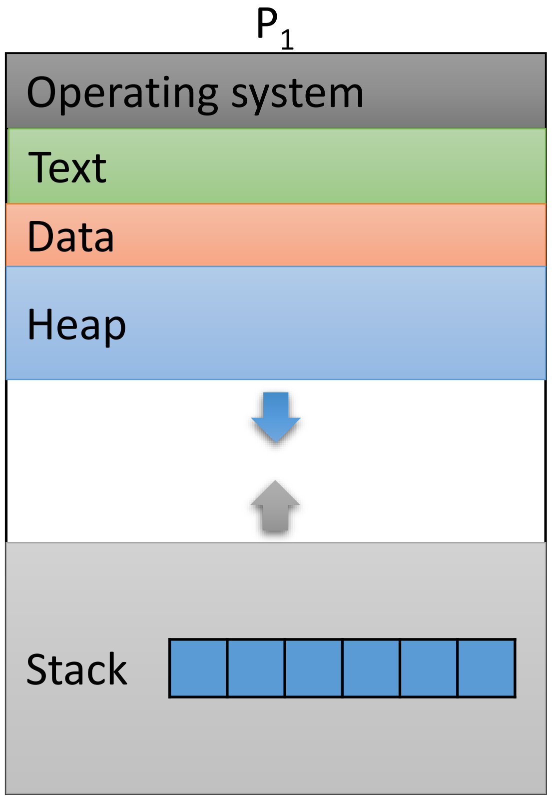


Let's say process P_1 wants to send data to process P_2.

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

They do NOT directly share any memory.

Message Passing IPC (Pipe)

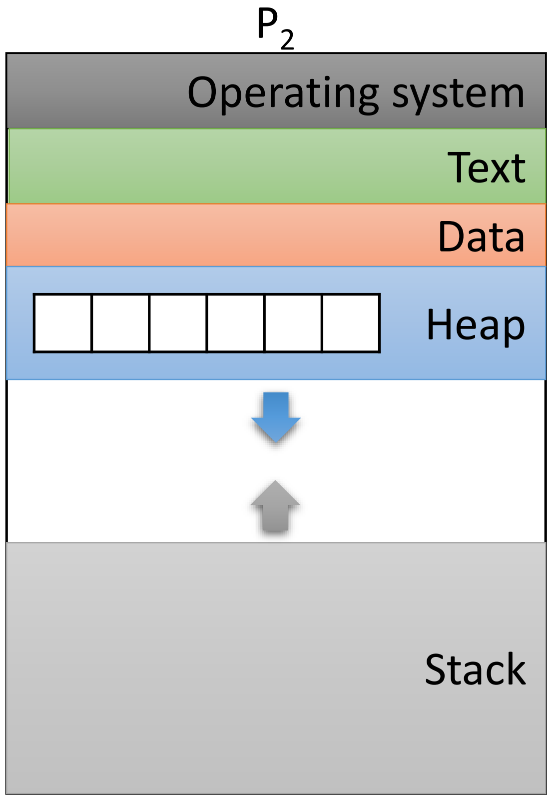


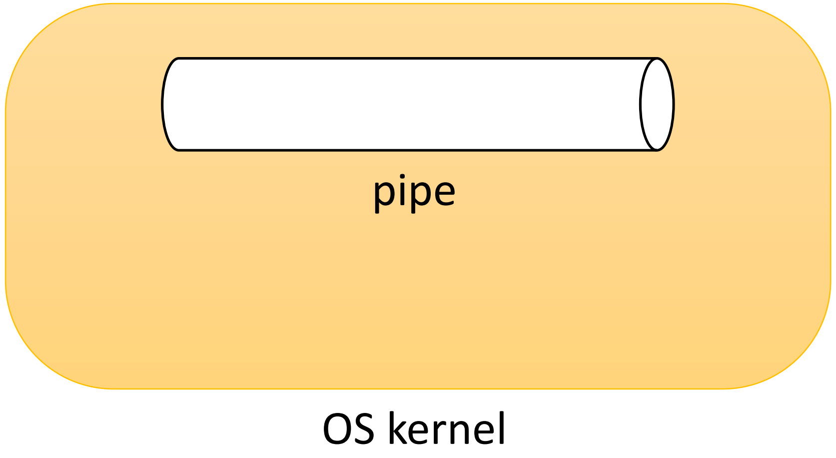
P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

data pointer: the start of data to copy

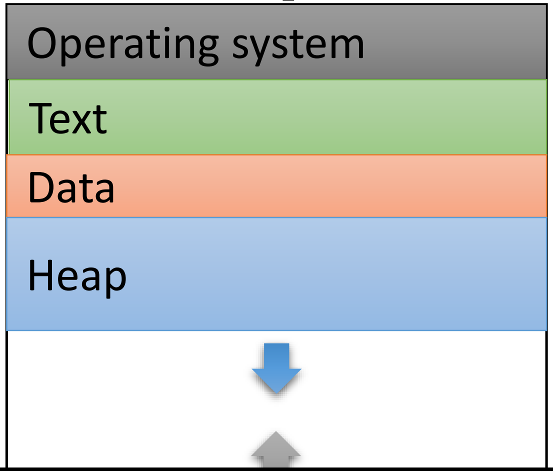
count: how many bytes to copy (at most)





Message Passing IPC (Pipe)

P_1

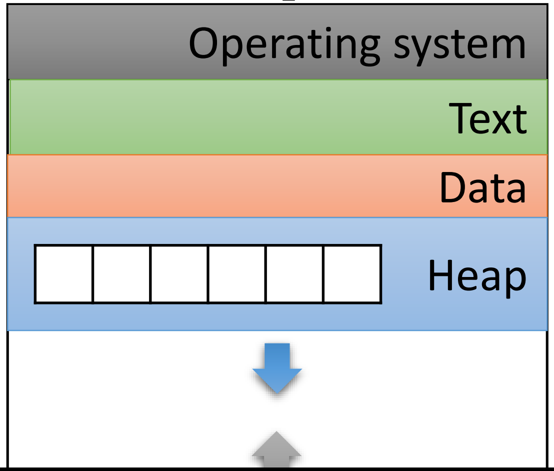


P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

data pointer: the start of data to copy

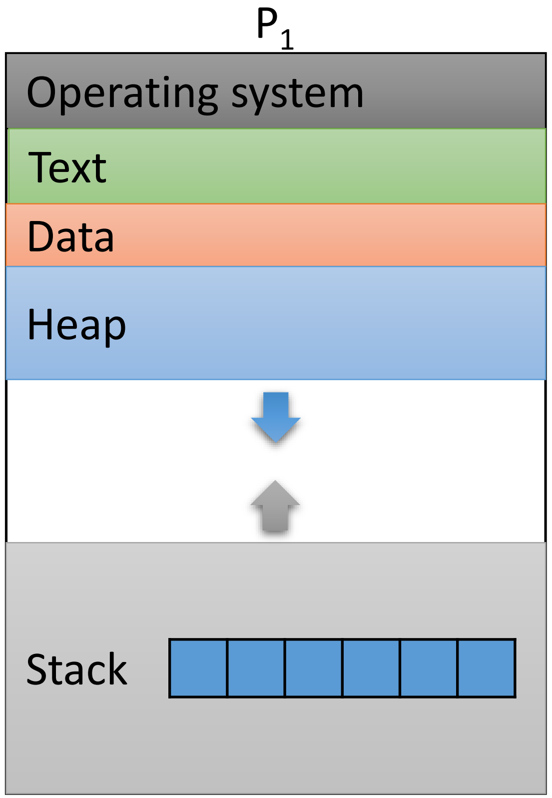
P_2




```
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.
```

OS kernel

Message Passing IPC (Pipe)

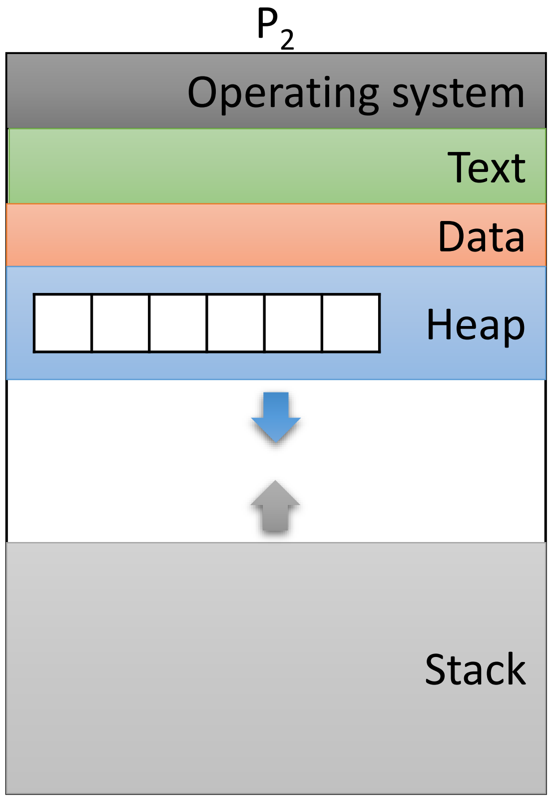


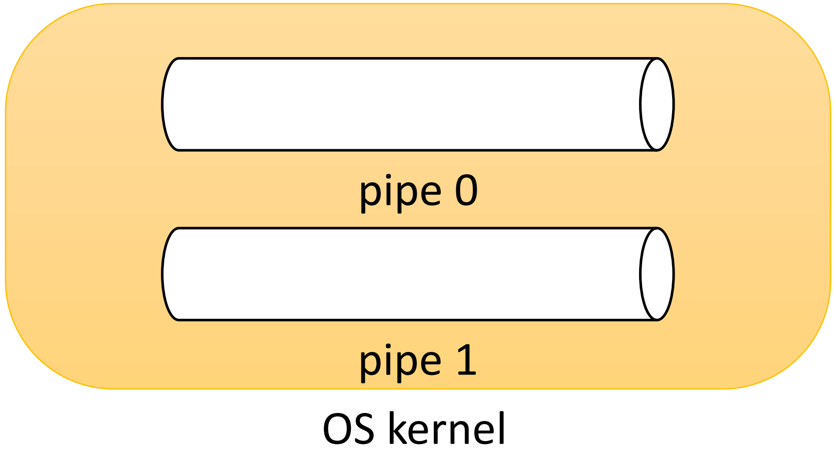
P_1 can send data into the pipe by calling:

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

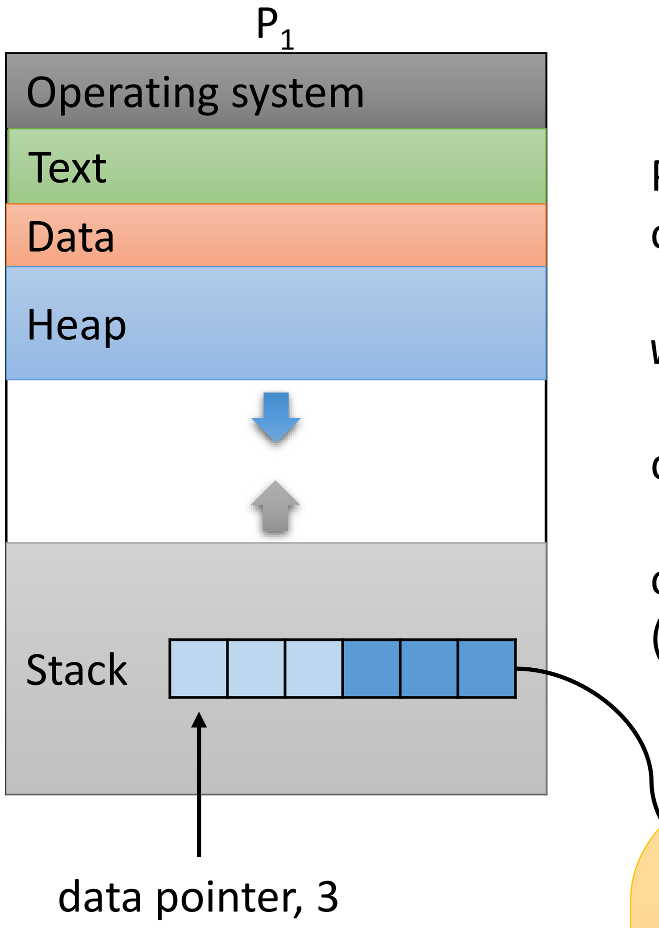
data pointer: the start of data to copy

count: how many bytes to copy





Message Passing IPC (Pipe)

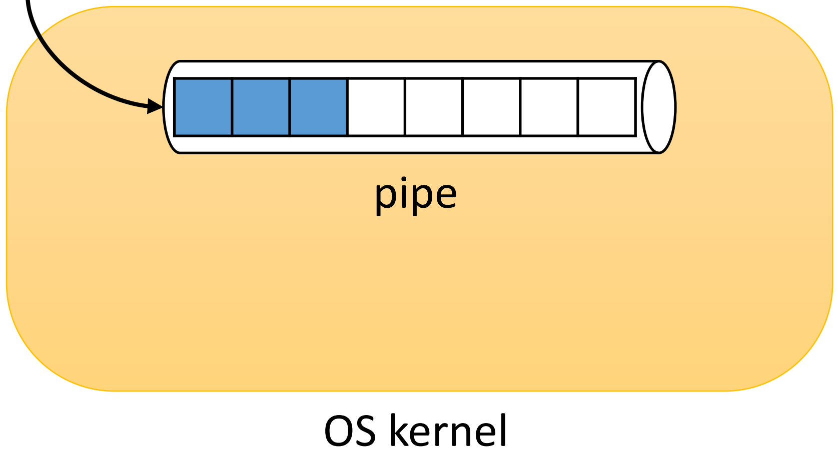


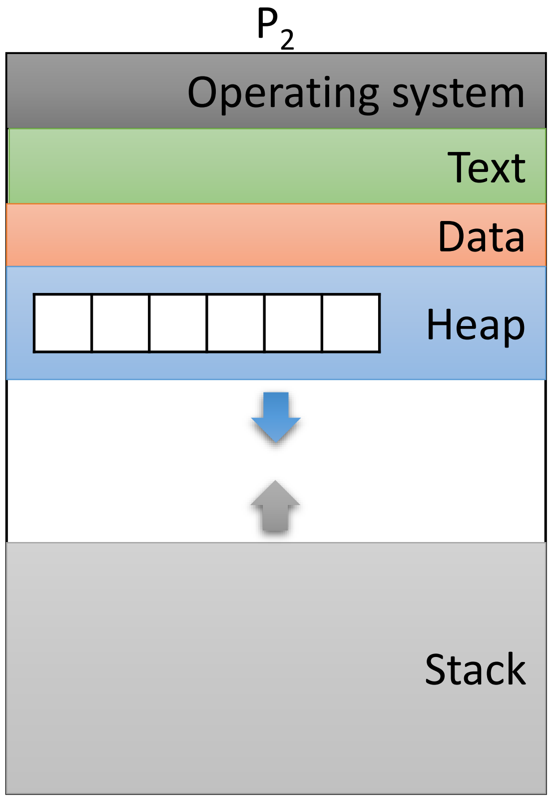
P_1 can send data into the pipe by calling:

`write(…, data pointer, count)`

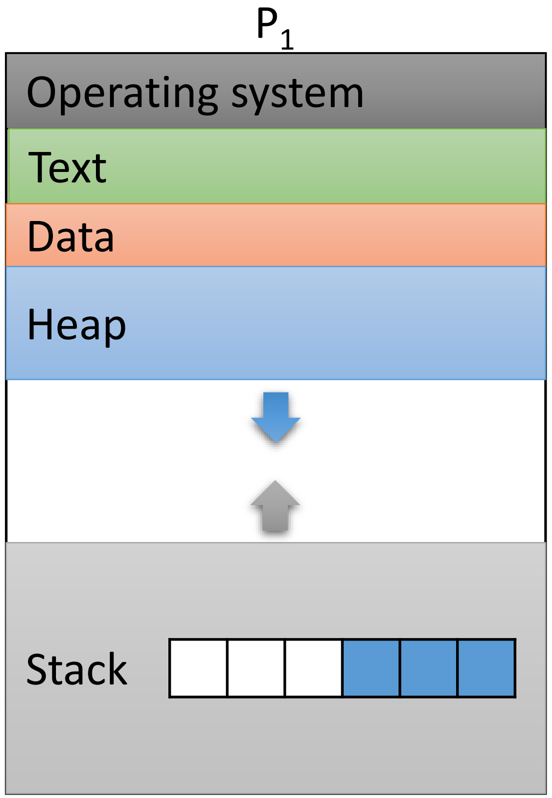
data pointer: the start of data to copy

count: how many bytes to copy (at most)





Message Passing IPC (Pipe)

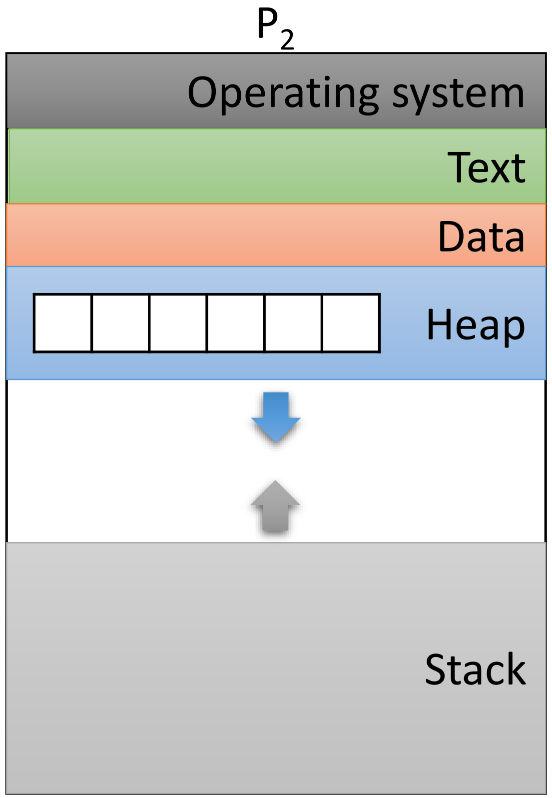


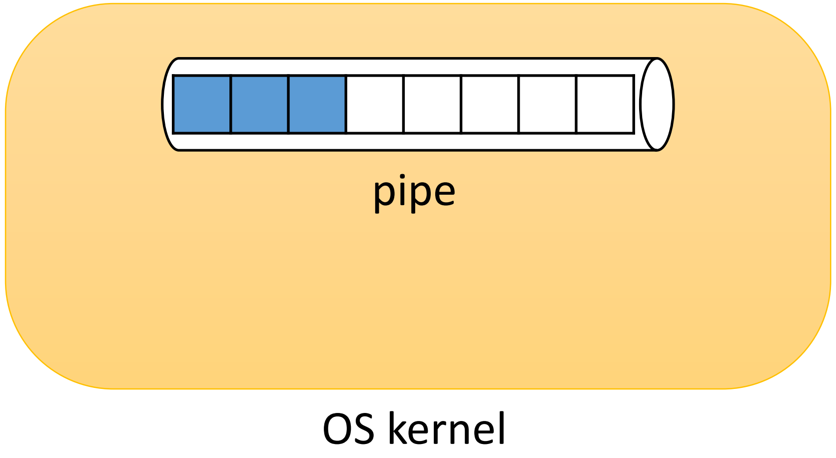
P_2 can receive data from the pipe by calling:

```
read(…, data pointer, count)
```

data pointer: the start of location to copy into

count: how many bytes to copy (at most)





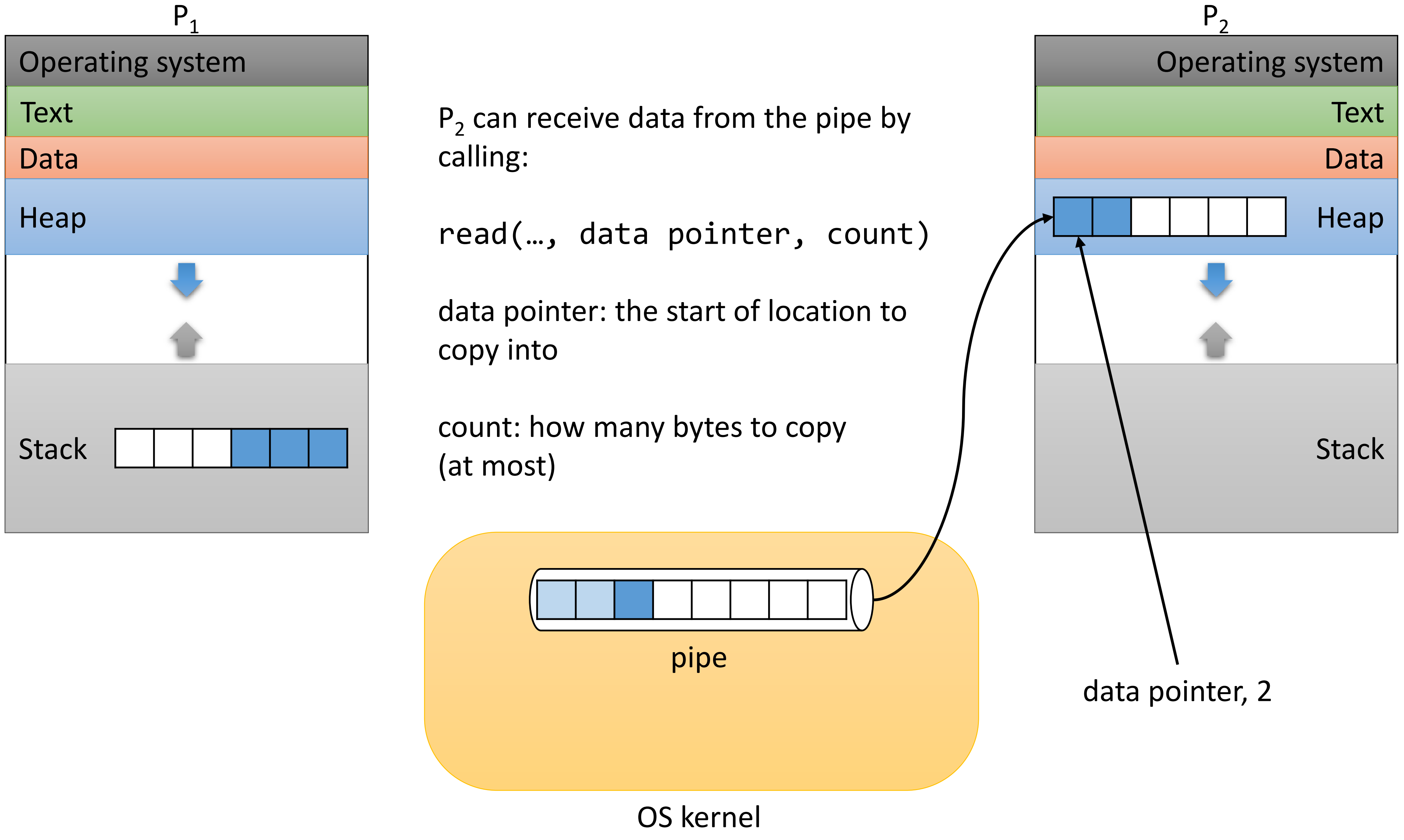
Message Passing IPC (Pipe)

P_2 can receive data from the pipe by calling:

```
read(…, data pointer, count)
```

data pointer: the start of location to copy into

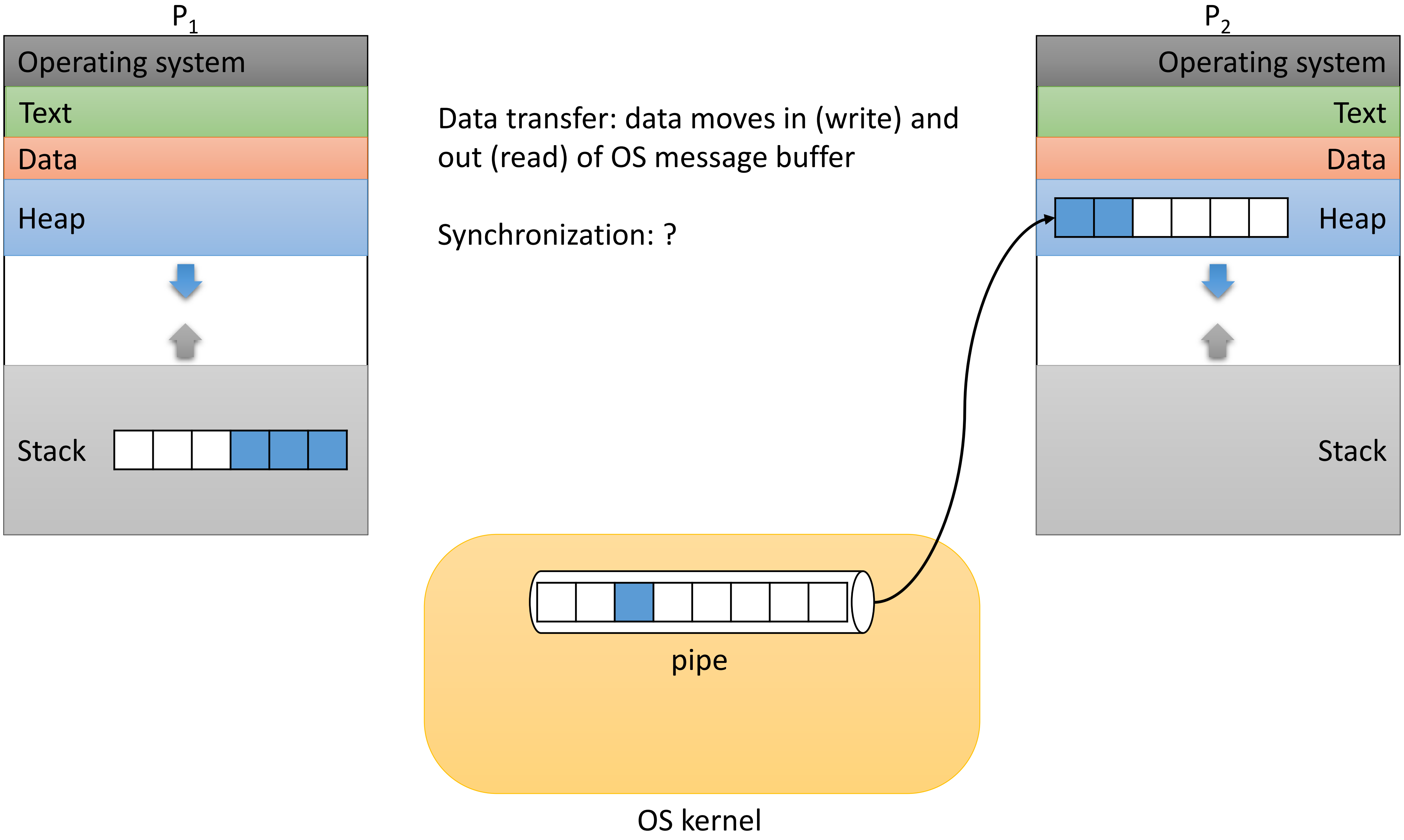
count: how many bytes to copy (at most)



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.

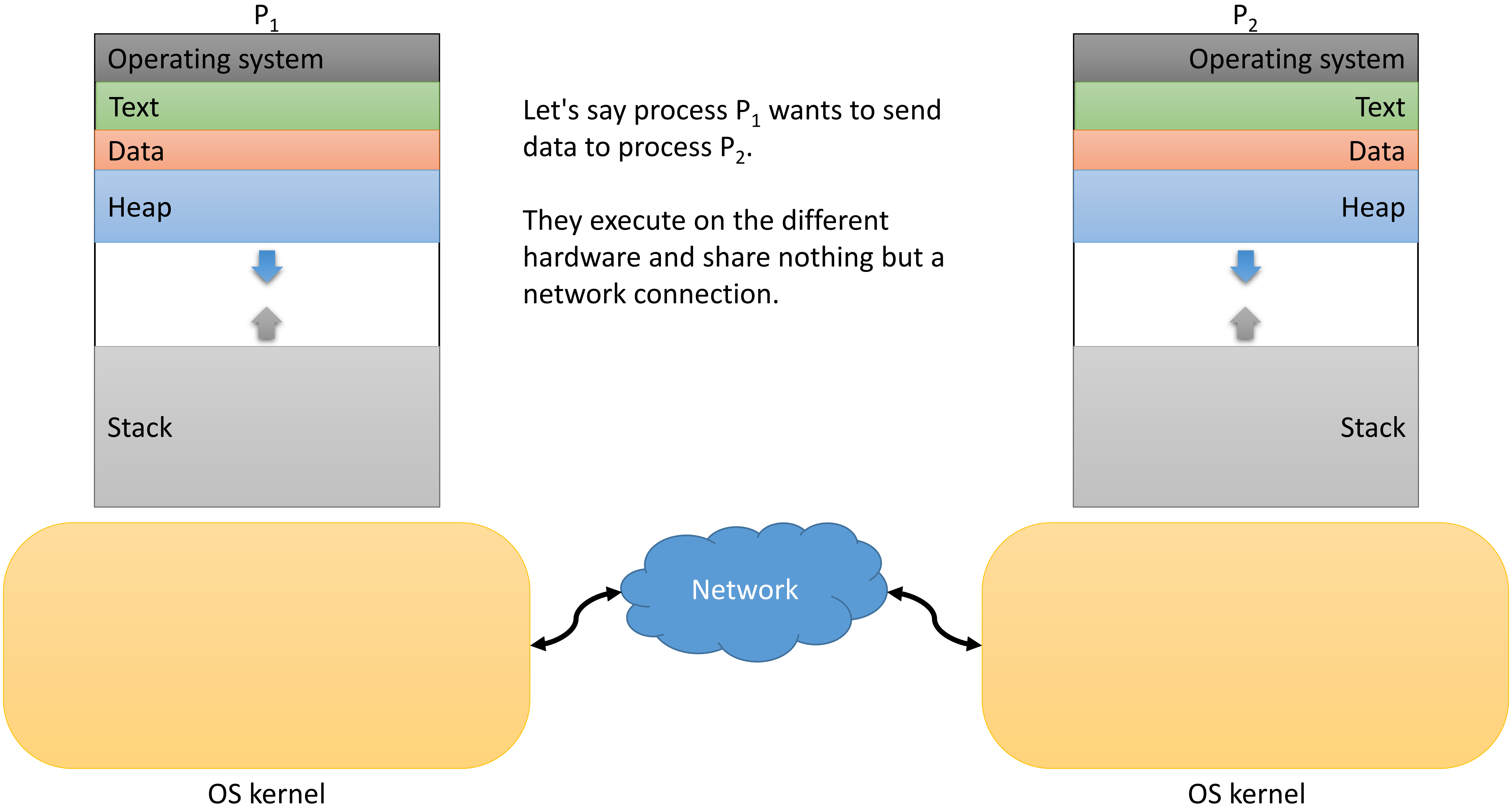
C. The communicating processes exchange synchronization messages (lock/unlock).

D. There is no synchronization mechanism.

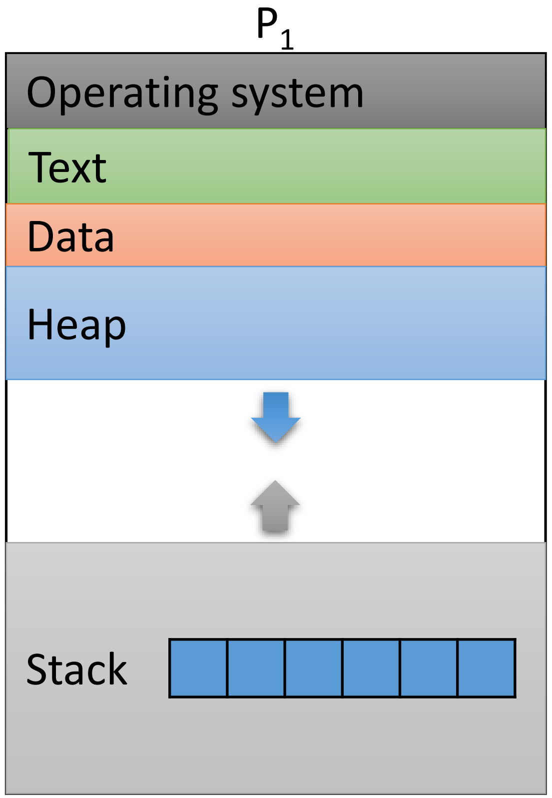
Message Passing IPC (Socket)

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)

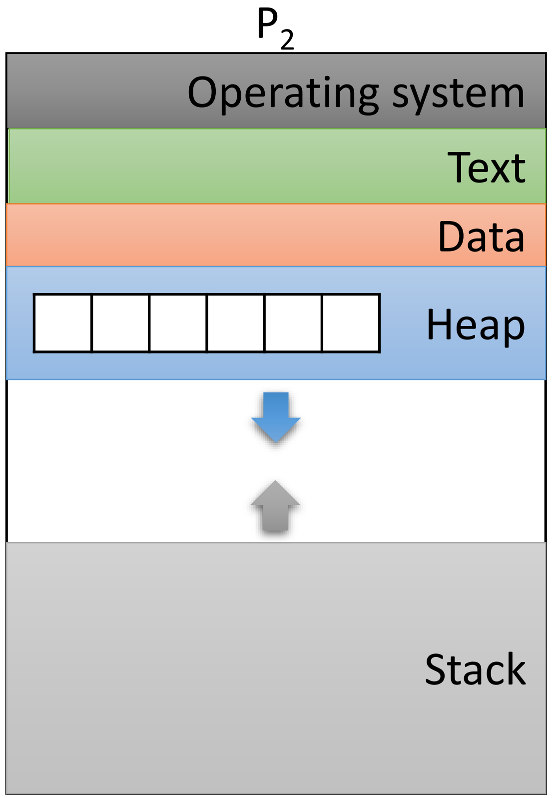


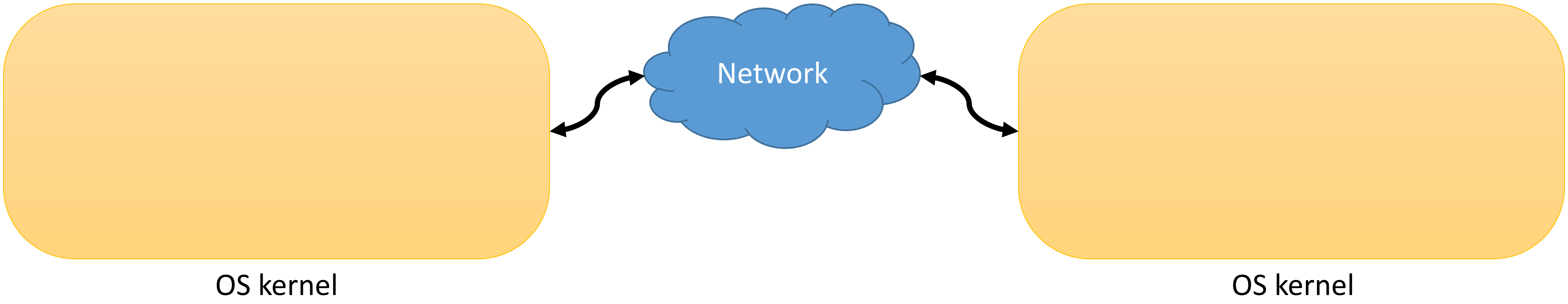
P_1 can send data into the socket by calling:

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

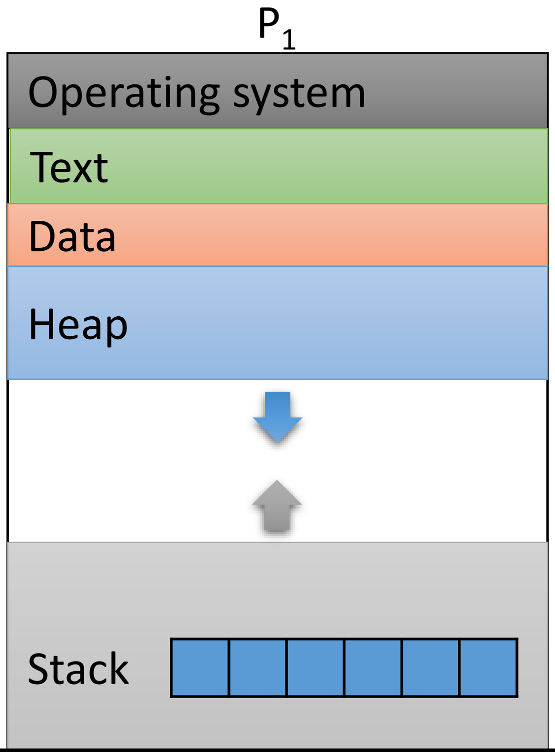
data pointer: the start of data to copy

count: how many bytes to copy
(at most)





Message Passing IPC (Socket)

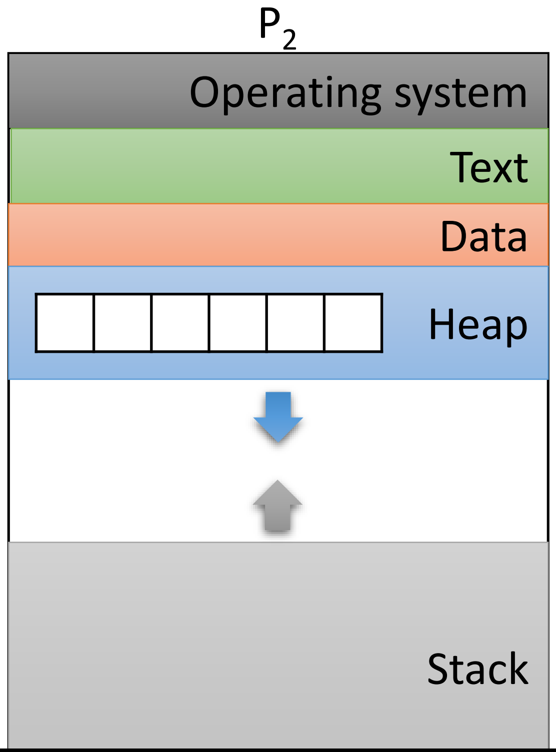


P_1 can send data into the socket by calling:

`send(…, data pointer, count, …)`

data pointer: the start of data to copy

count: how many bytes to copy
(at most)



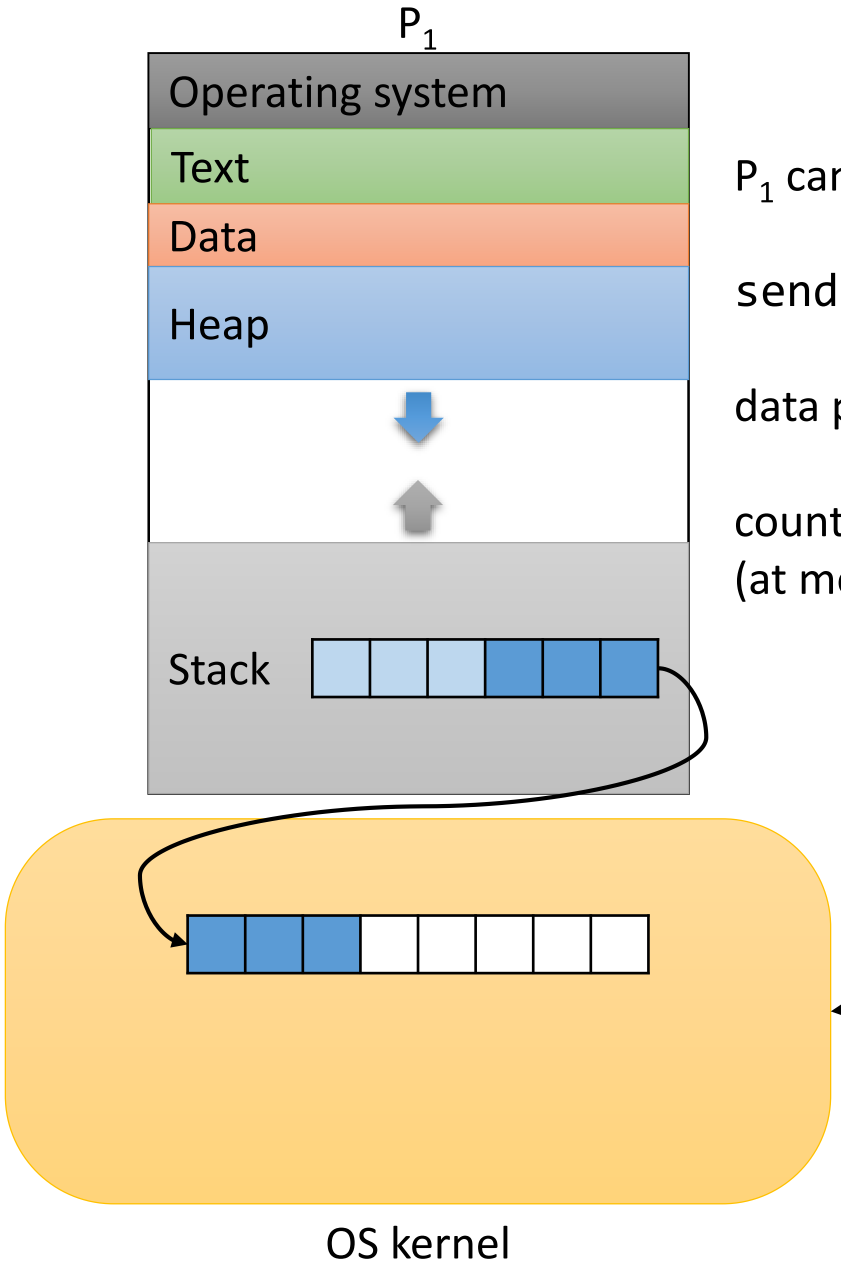

```
NAME
       send, sendto, sendmsg - send a message on a socket

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)

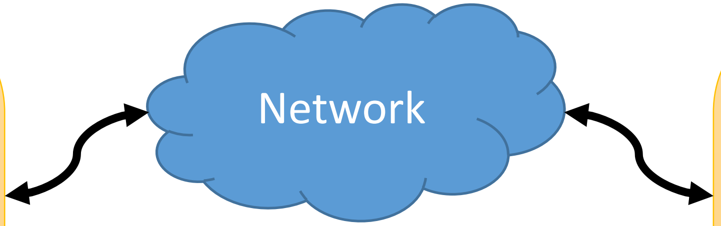


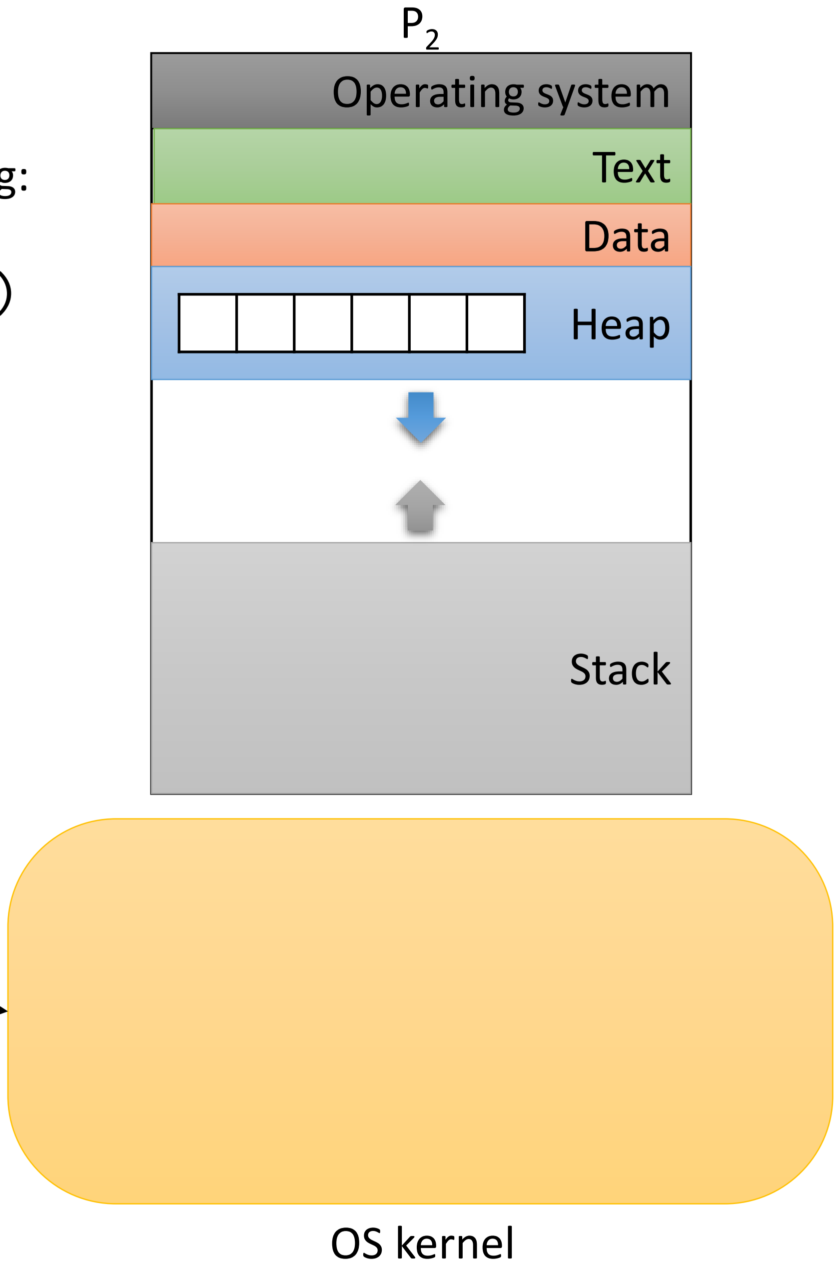
P_1 can send data into the socket by calling:

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

data pointer: the start of data to copy

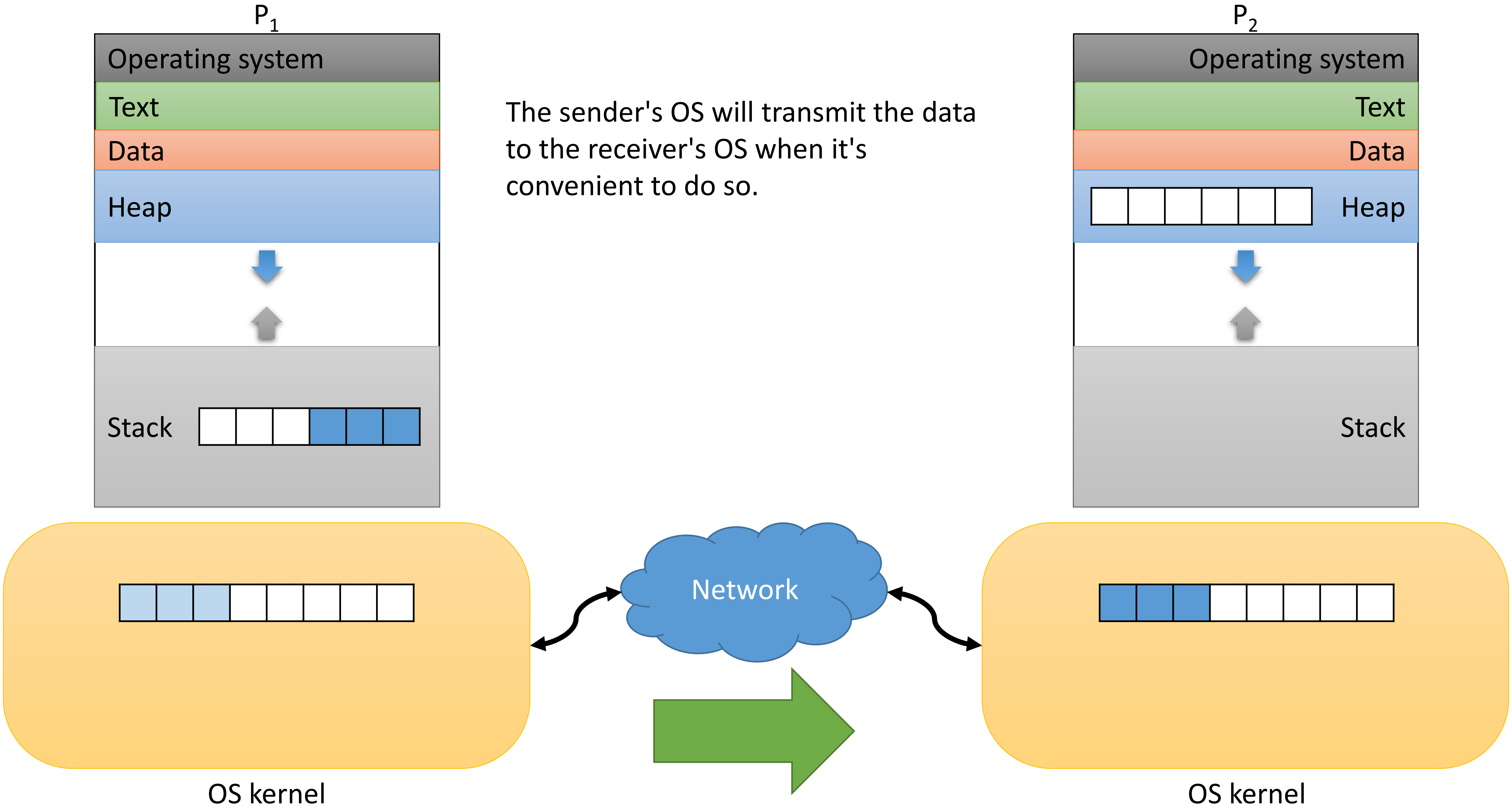
count: how many bytes to copy (at most)





Message Passing IPC (Socket)

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



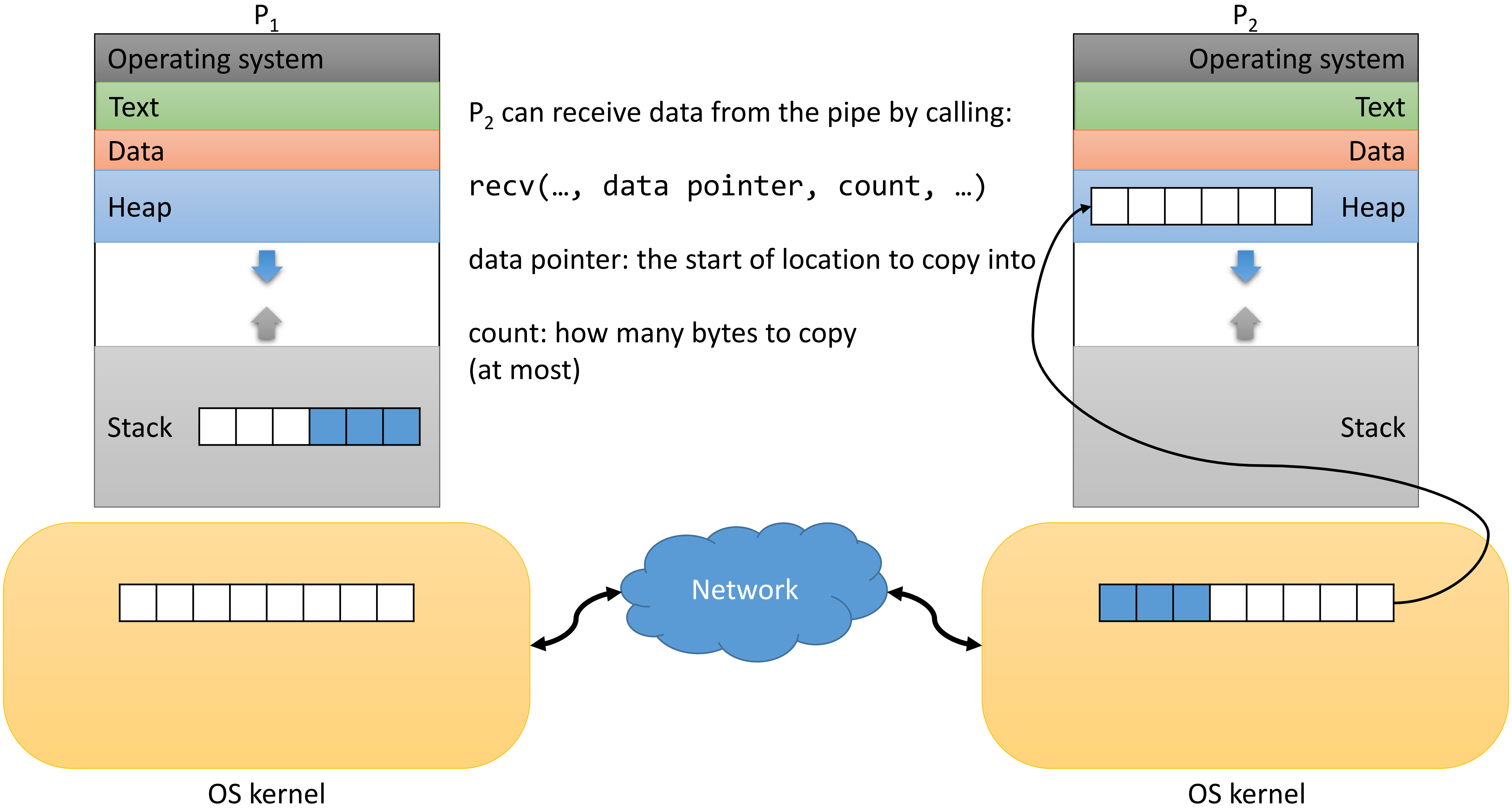
Message Passing IPC (Socket)

P_2 can receive data from the pipe by calling:

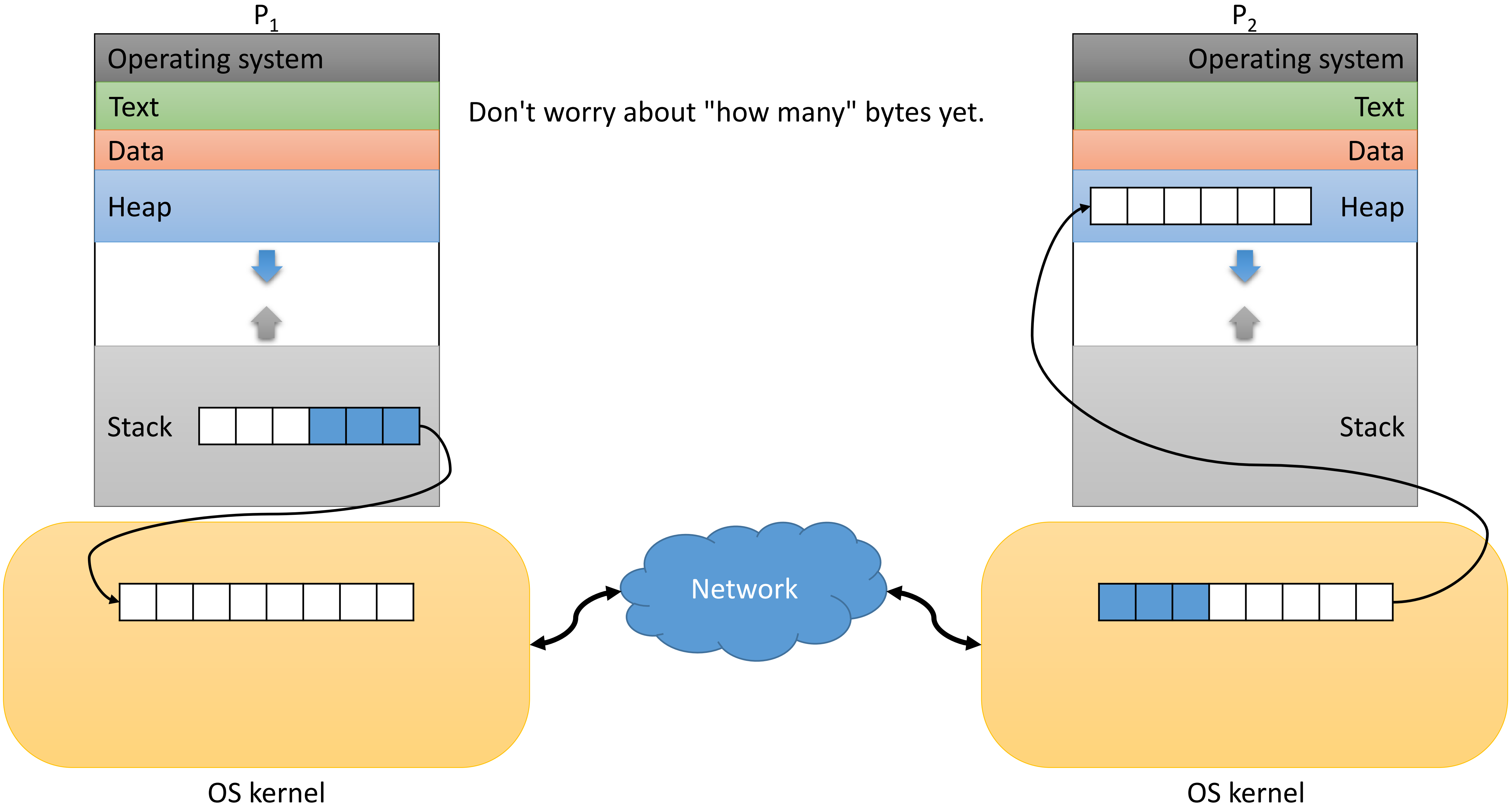
`recv(…, data pointer, count, …)`

data pointer: the start of location to copy into

count: how many bytes to copy
(at most)

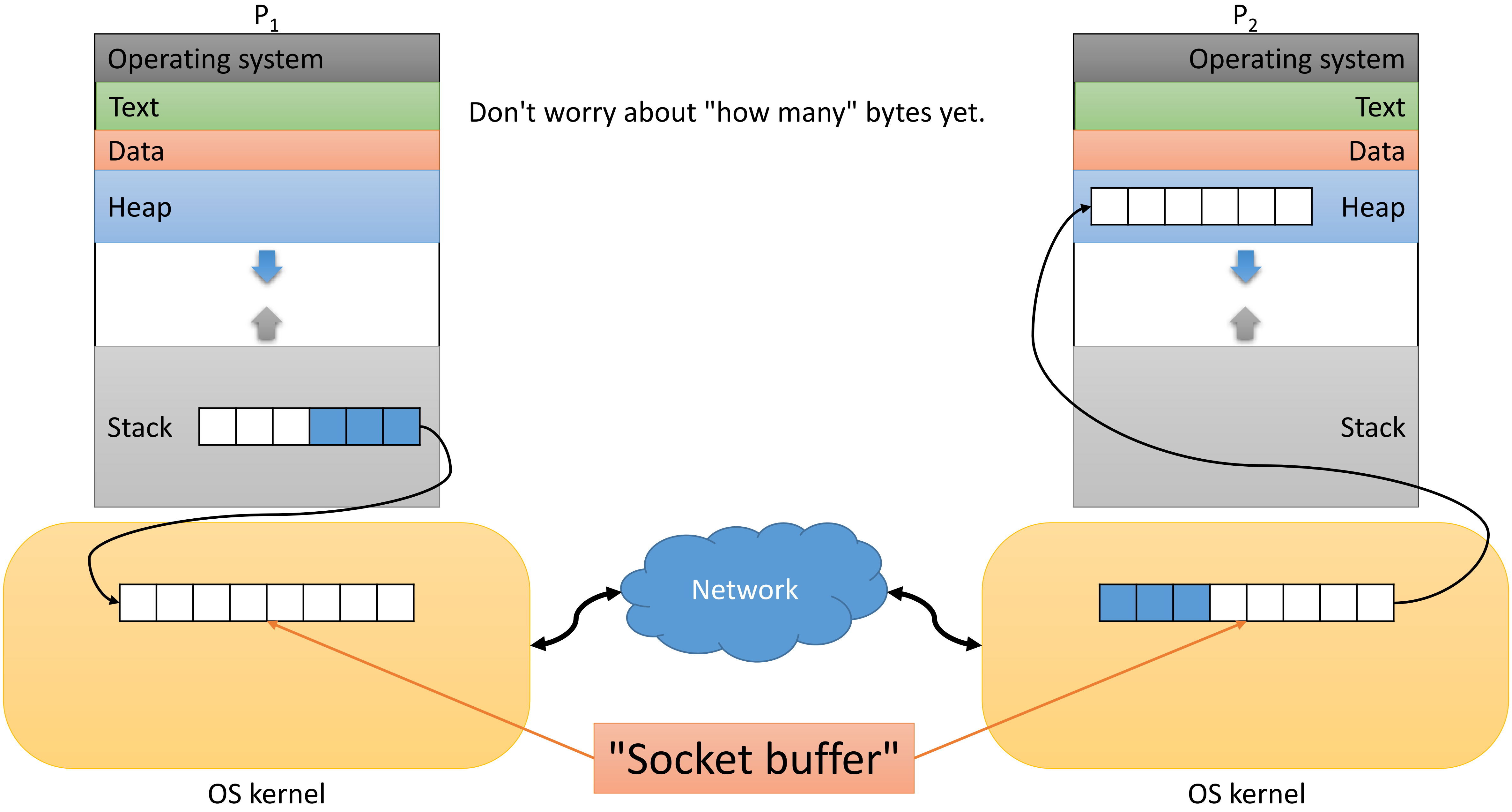


Questions about this model?

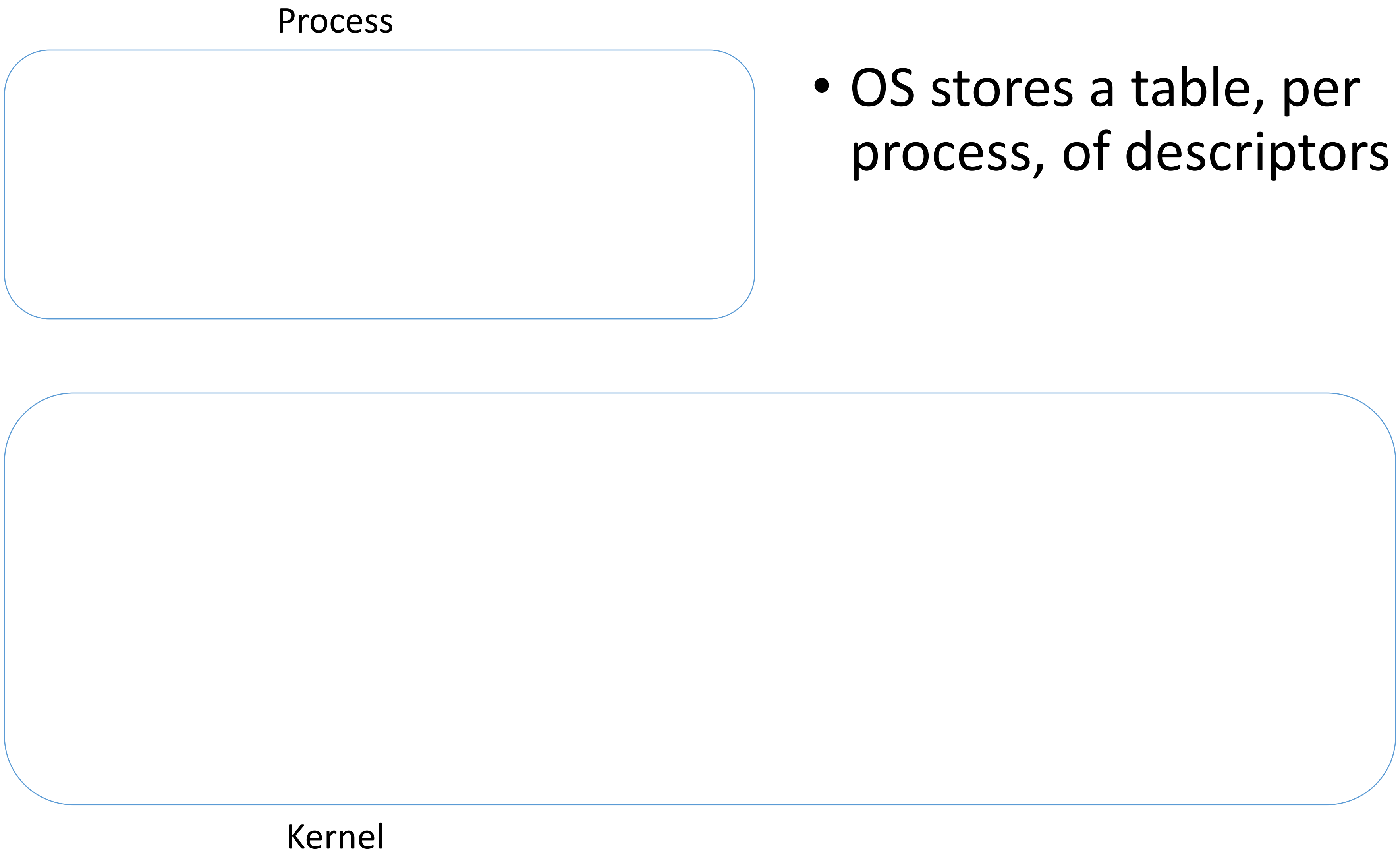


Don't worry about "how many" bytes yet.

Questions about this model?



Descriptor Table



- OS stores a table, per process, of descriptors

Descriptors

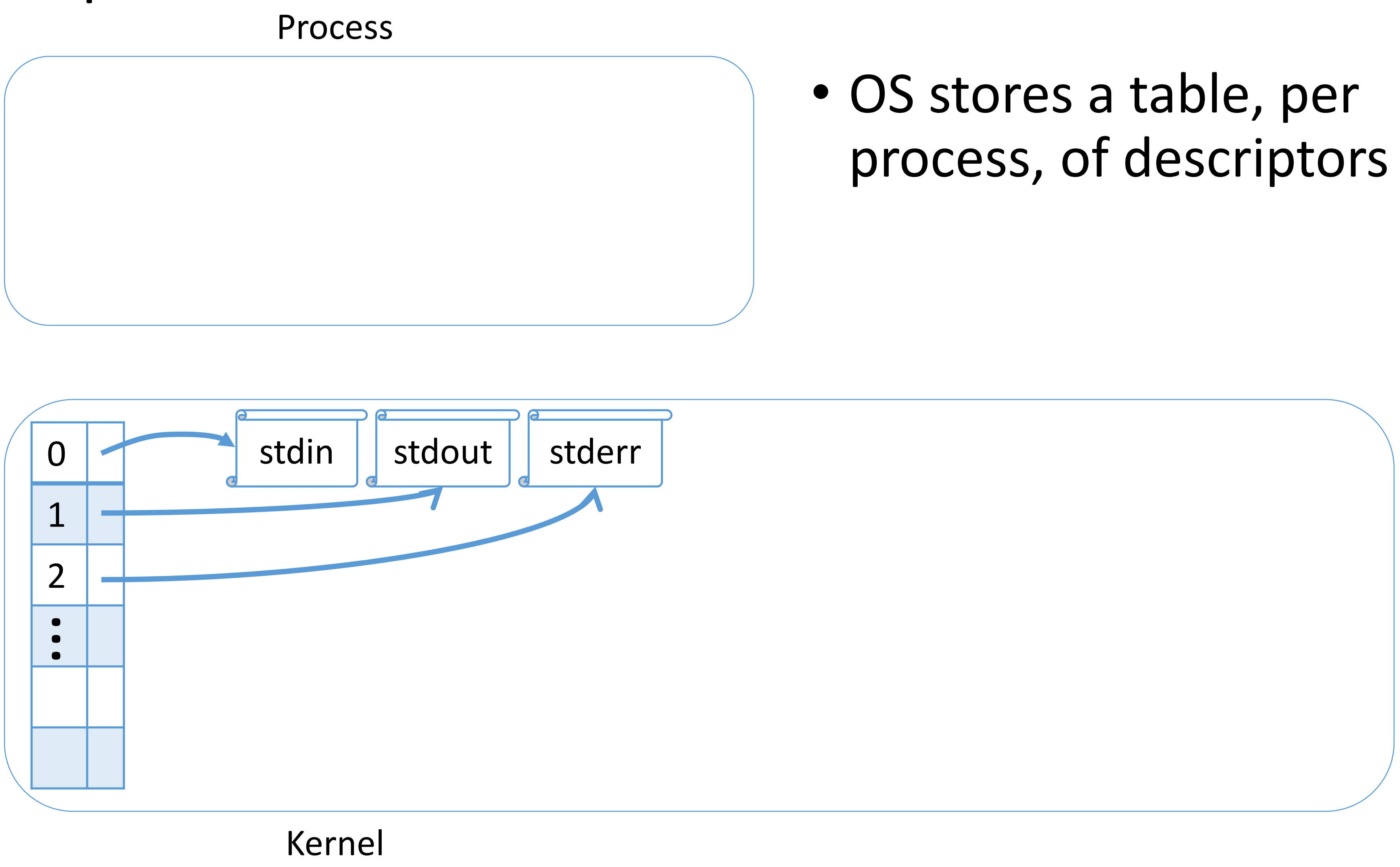
Where do descriptors come from?

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)
;
```

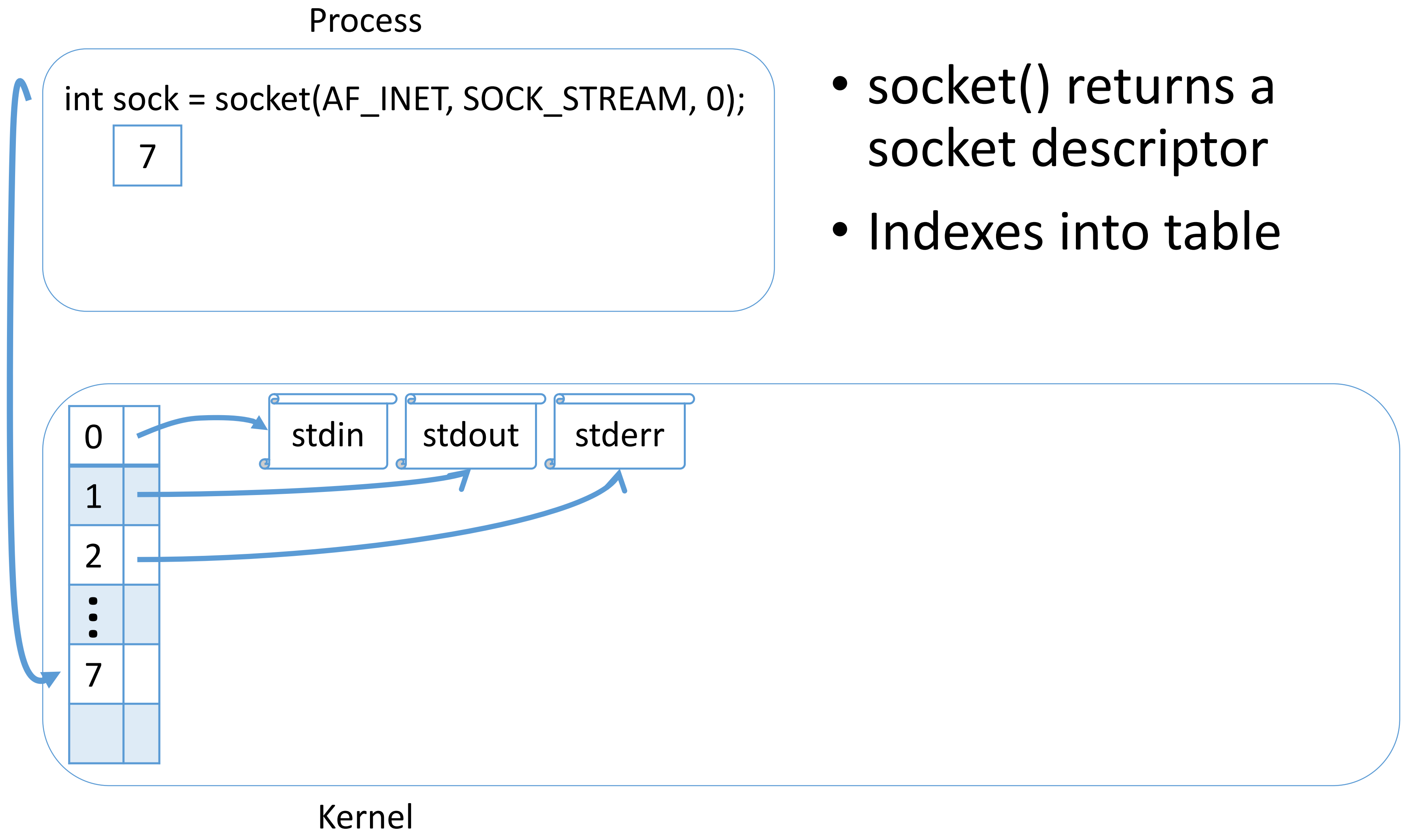
```
SOCKET(2)          Linux Programmer's Manual         SOCKET(2)

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



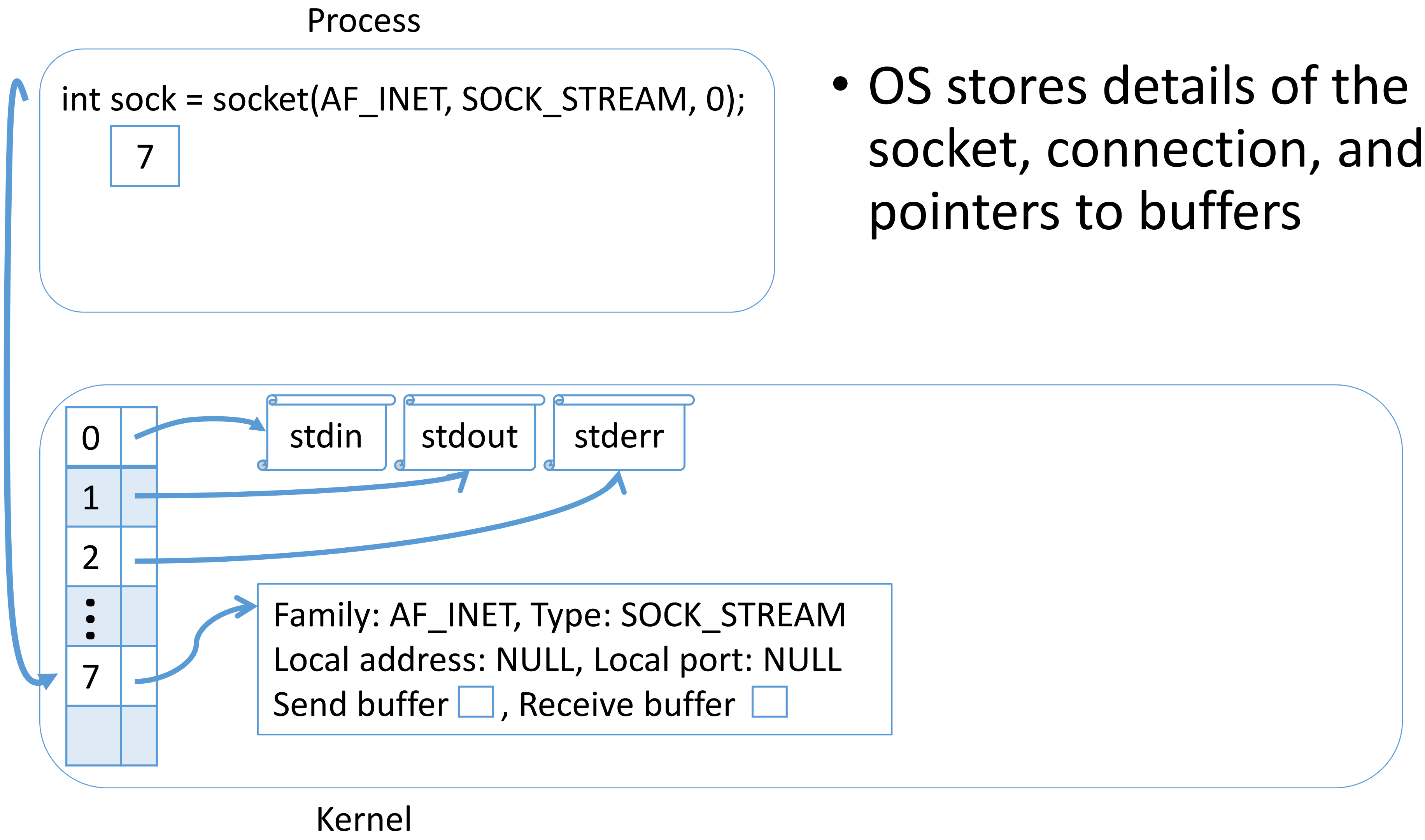
- OS stores a table, per process, of descriptors

socket()



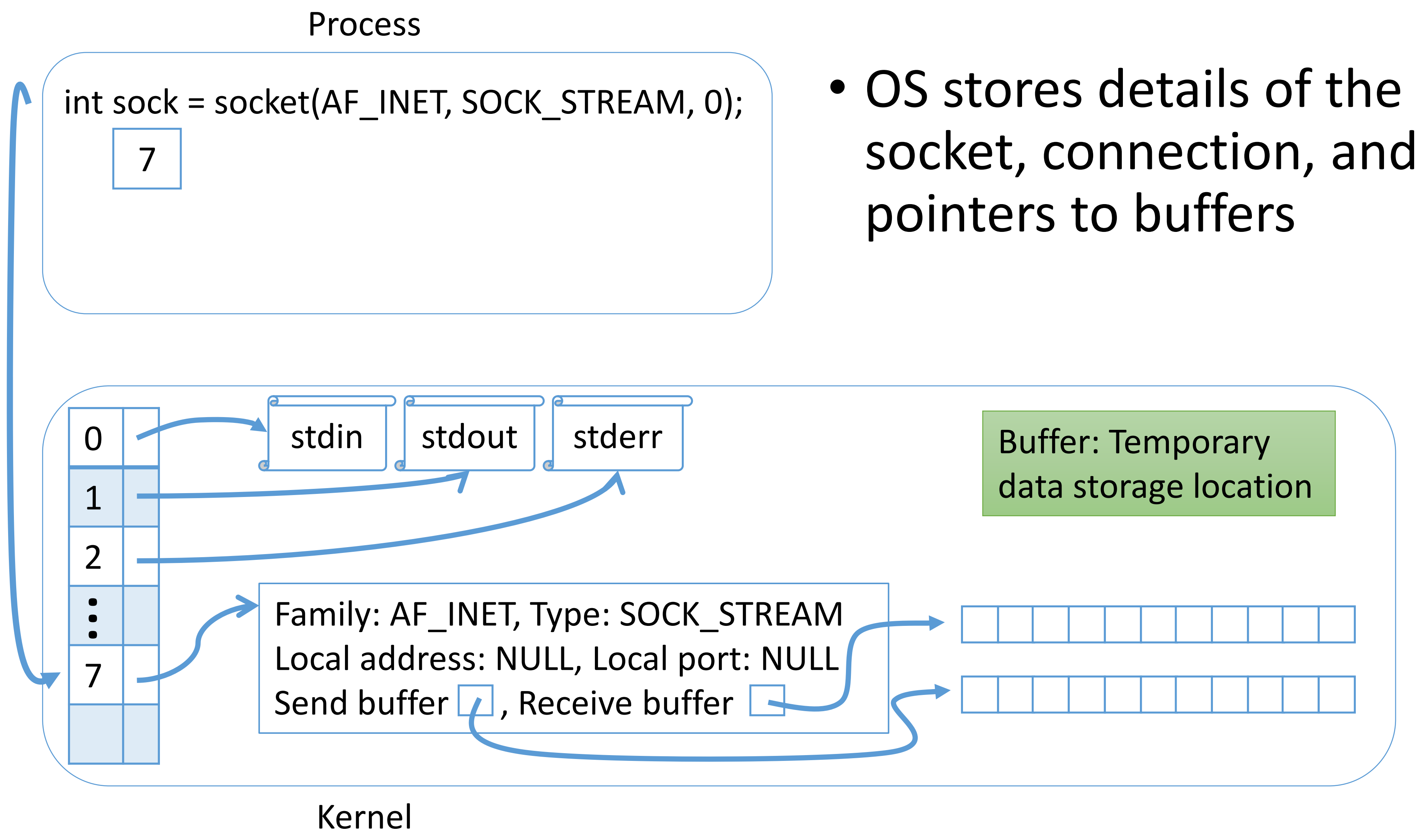
- socket() returns a socket descriptor
- Indexes into table

socket()



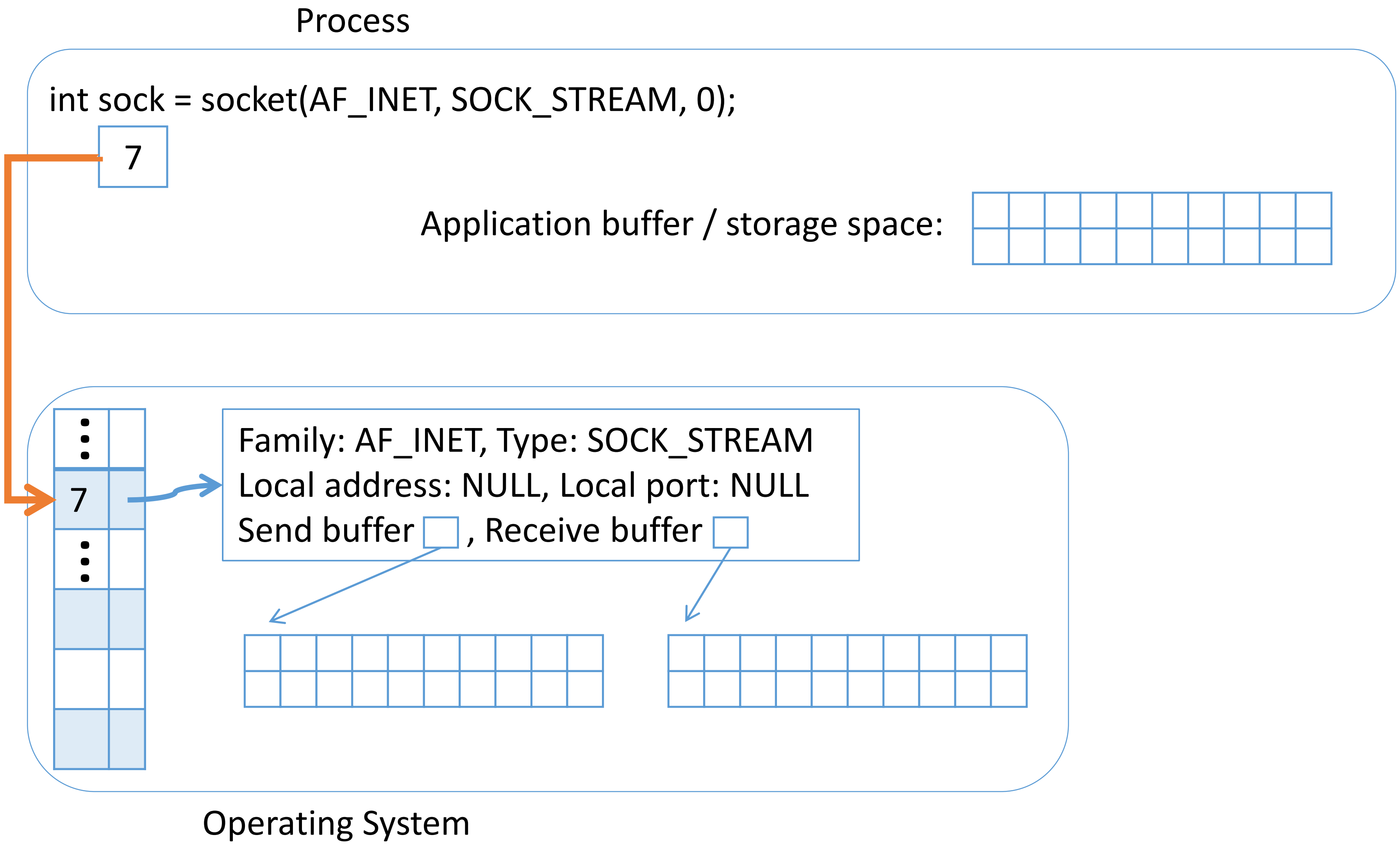
- OS stores details of the socket, connection, and pointers to buffers

socket()

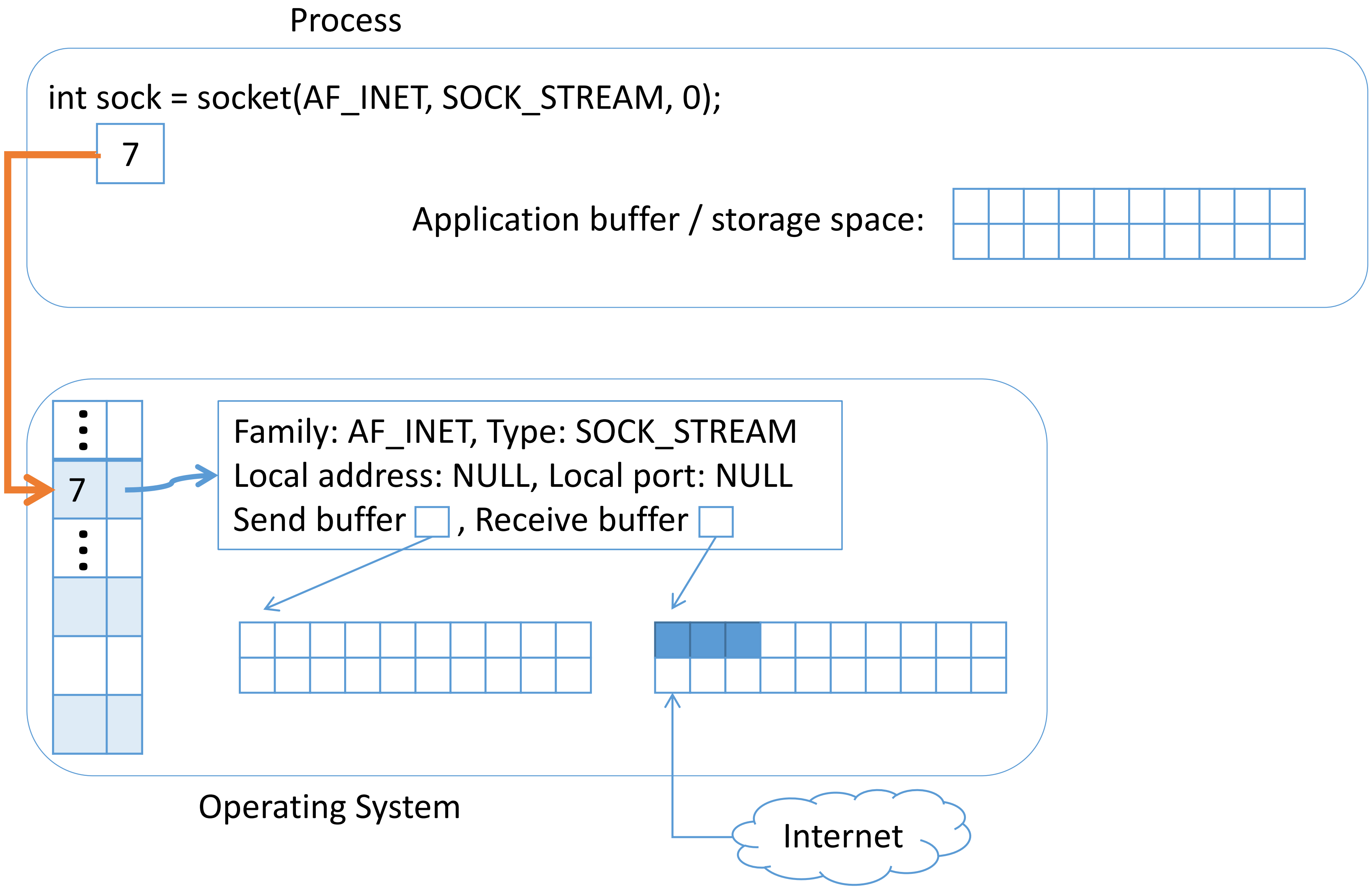


- OS stores details of the socket, connection, and pointers to buffers

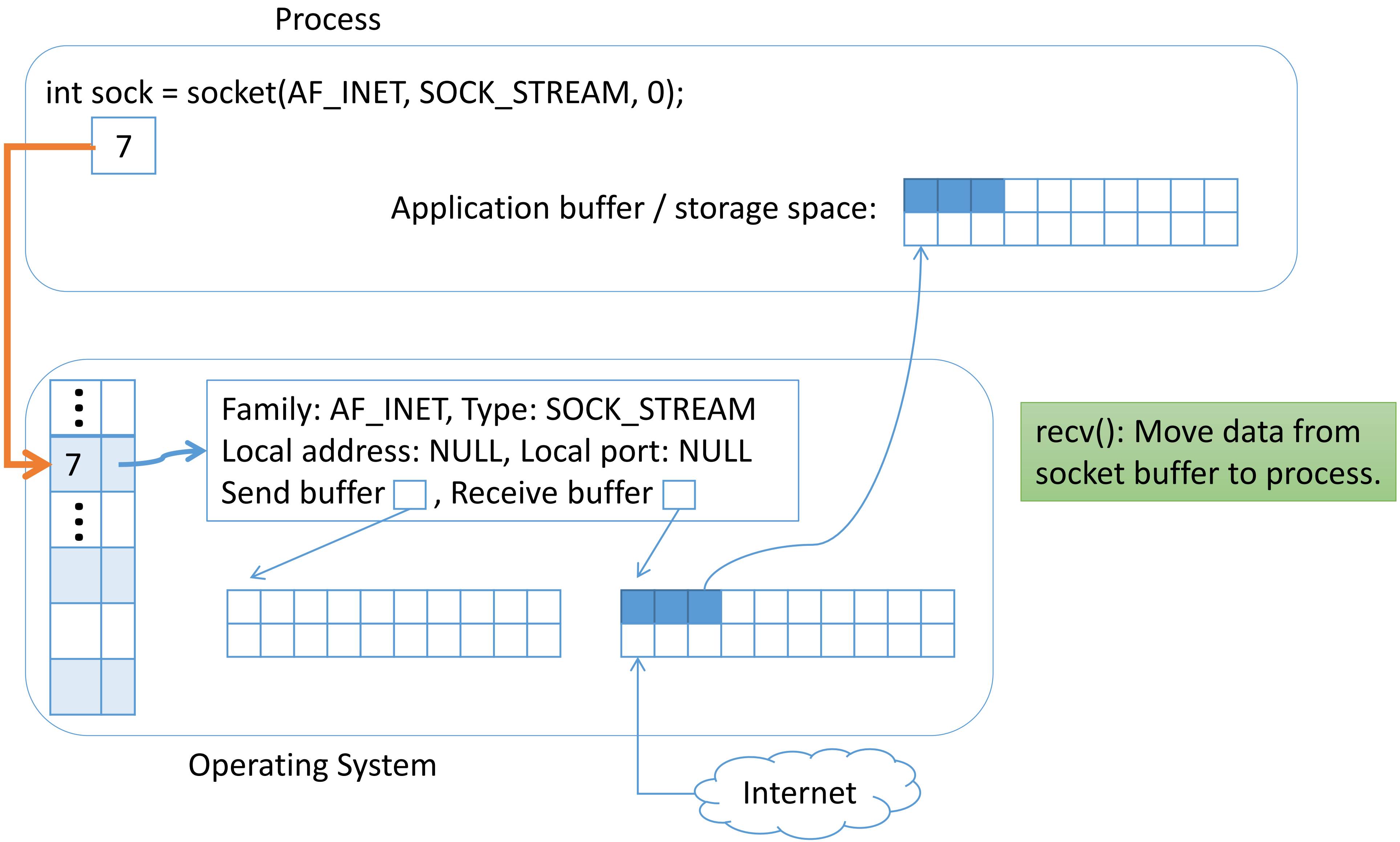
Socket Buffers



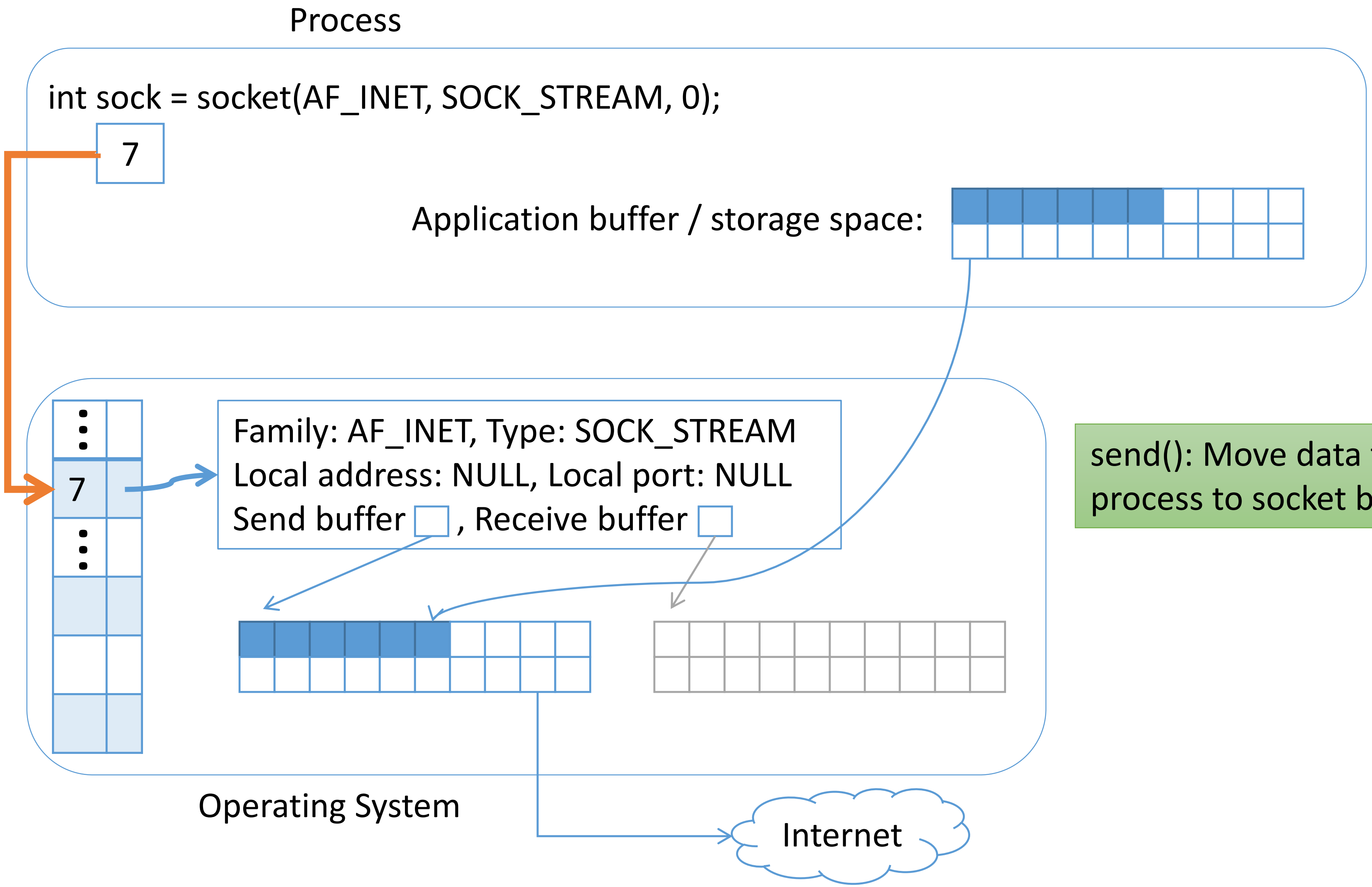
Socket Buffers



Socket Buffers

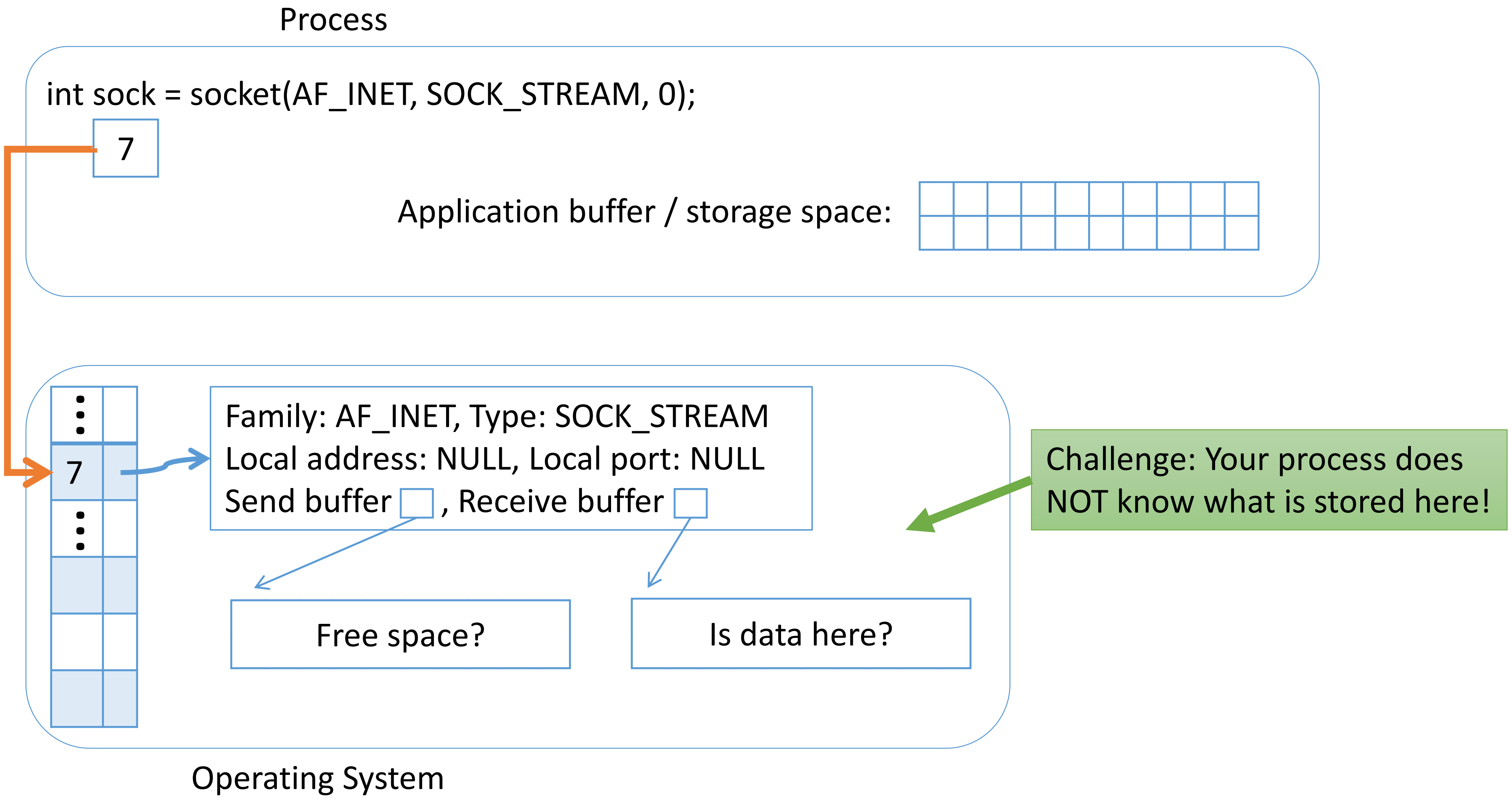


Socket Buffers

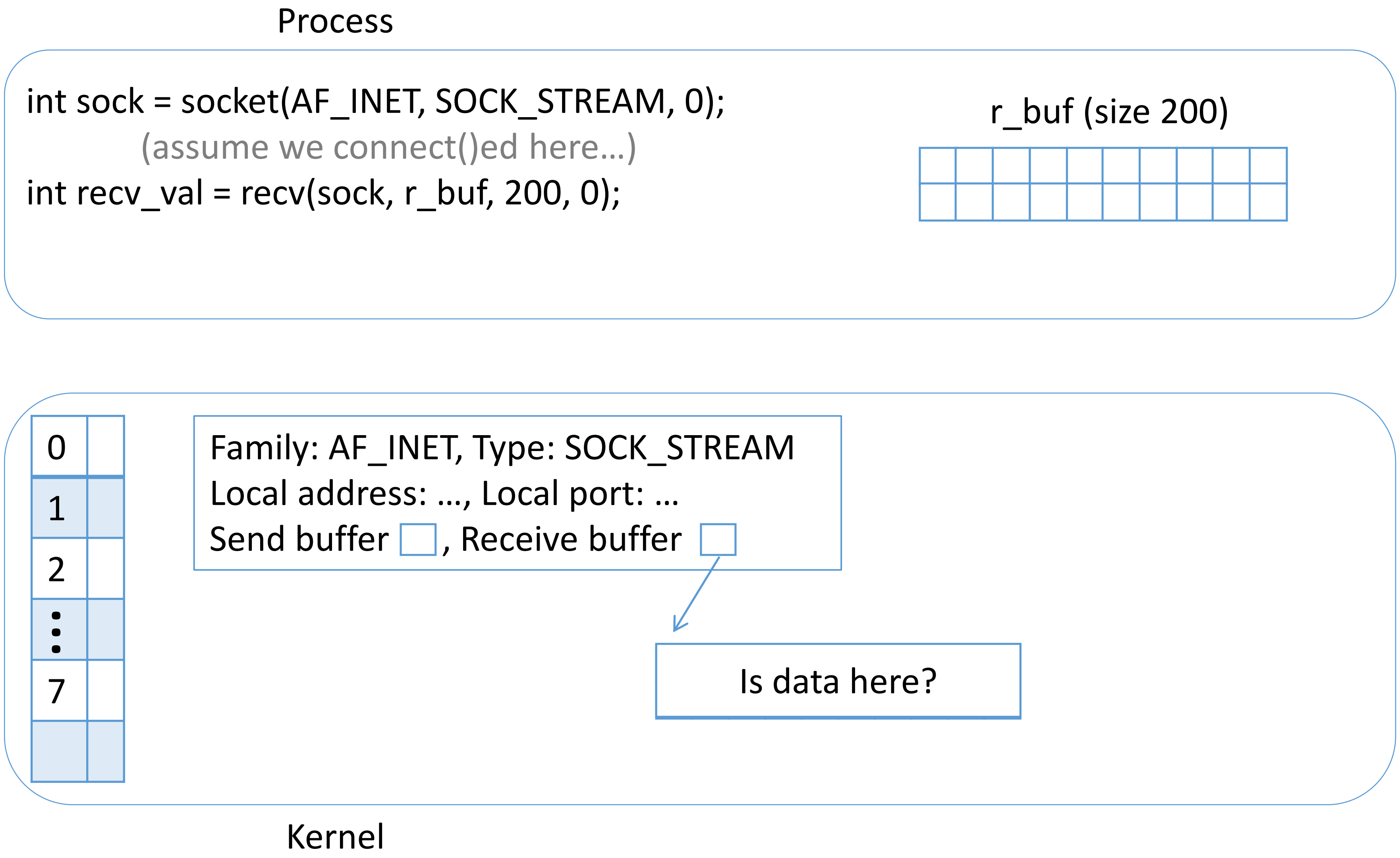


send(): Move data from process to socket buffer

Socket Buffers



recv()

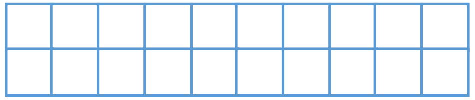

Process
int sock = socket(AF_INET, SOCK_STREAM, 0);
(assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
r_buf (size 200)
0
1
2
7
Family: AF_INET, Type: SOCK_STREAM
Local address: …, Local port: …
Send buffer , Receive buffer
Is data here?
Kernel

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

Process

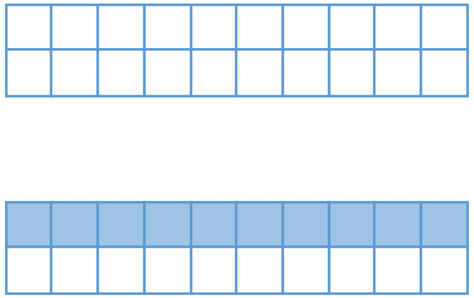
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
          (assume we connect()ed here…)
int recv_val = recv(sock, r_buf, 200, 0);
```

r_buf (size 200)



Two Scenarios:

Socket buffer (receive)



Empty

100 bytes

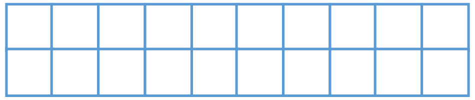
Kernel

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);
```

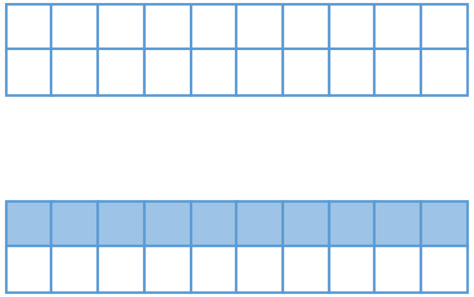
r_buf (size 200)



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)



Empty

100 bytes

Kernel

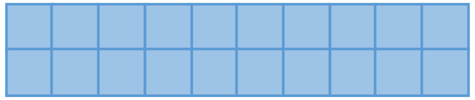
"Block" means pause the calling process.

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

Process

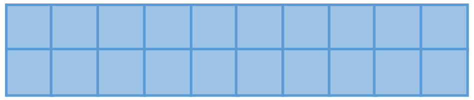
```
int sock = socket(AF_INET, SOCK_STREAM, 0);
        (assume we connect()ed here…)
int send_val = send(sock, s_buf, 200, 0);
```

s_buf (size 200)

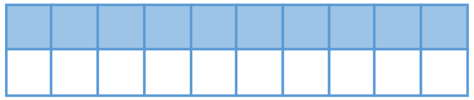


Two Scenarios:

Socket buffer (send)



Full



100 bytes

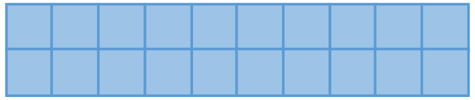
Kernel

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);
```

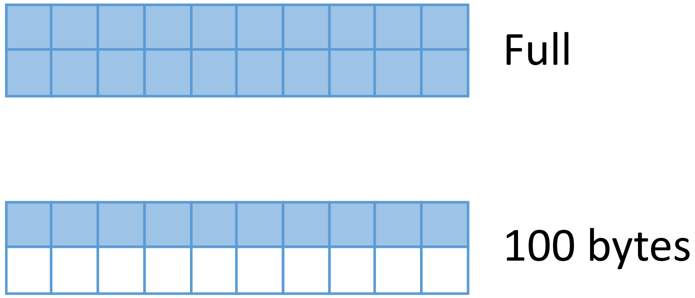
s_buf (size 200)



	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)



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.

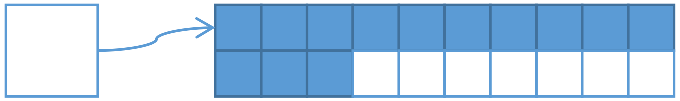
ALWAYS check send() return value!

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

Data sent: 0
Data to send: 130

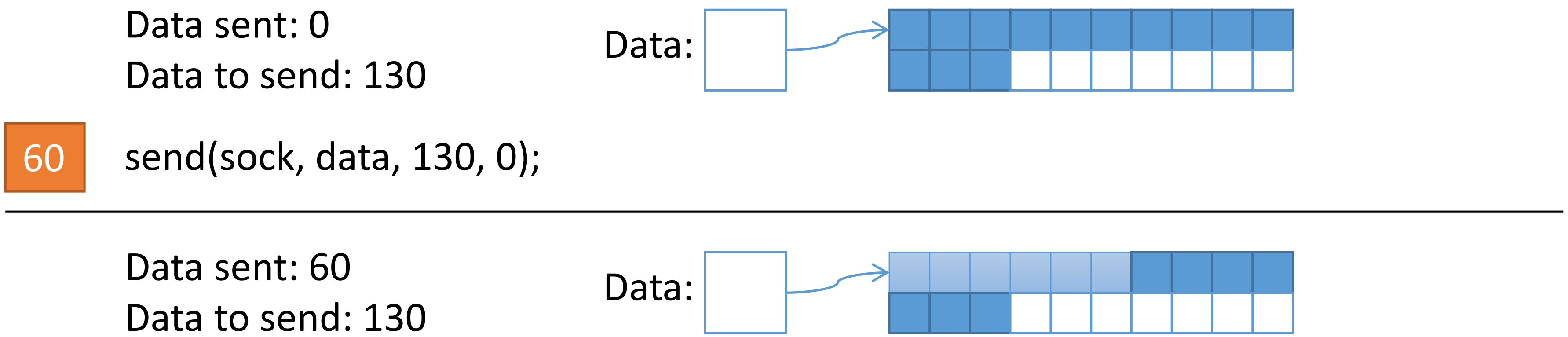
send(sock, data, 130, 0);

Data:



ALWAYS check send() return value!

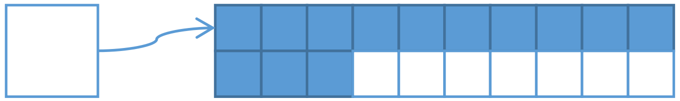
- When send() return value is less than the data size, **you are responsible for sending the rest**.



ALWAYS check send() return value!

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

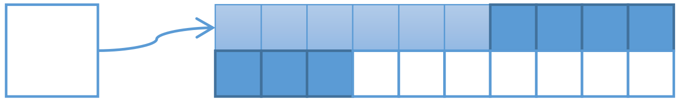
Data sent: 0
Data to send: 130

Data: 

60

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

Data sent: 60
Data to send: 130

Data: 

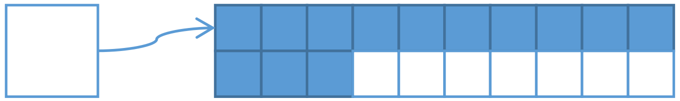
```
// Copy the 70 bytes starting from offset 60.
send(sock, data + 60, 130 - 60, 0);
```

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:

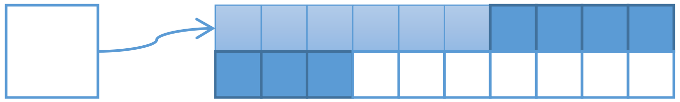


60

```
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

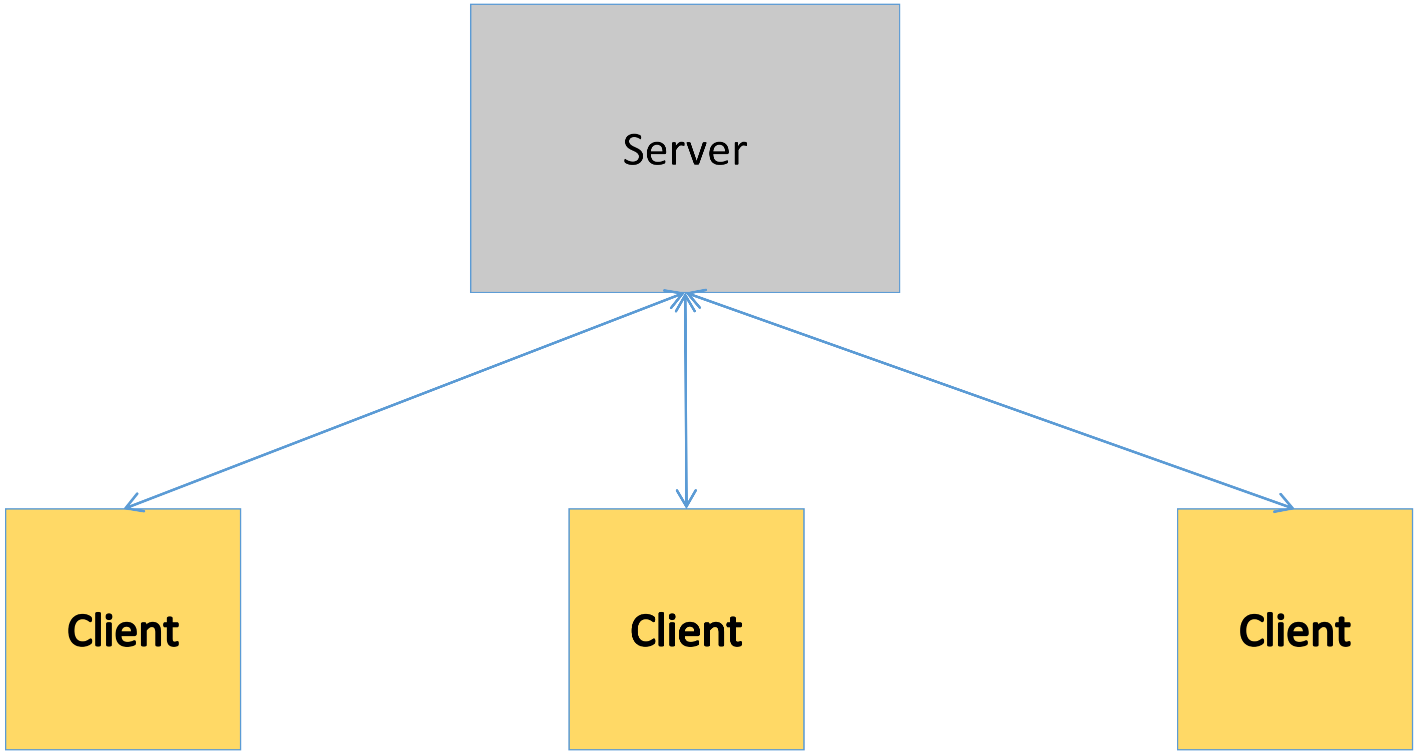
recv()

- 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!

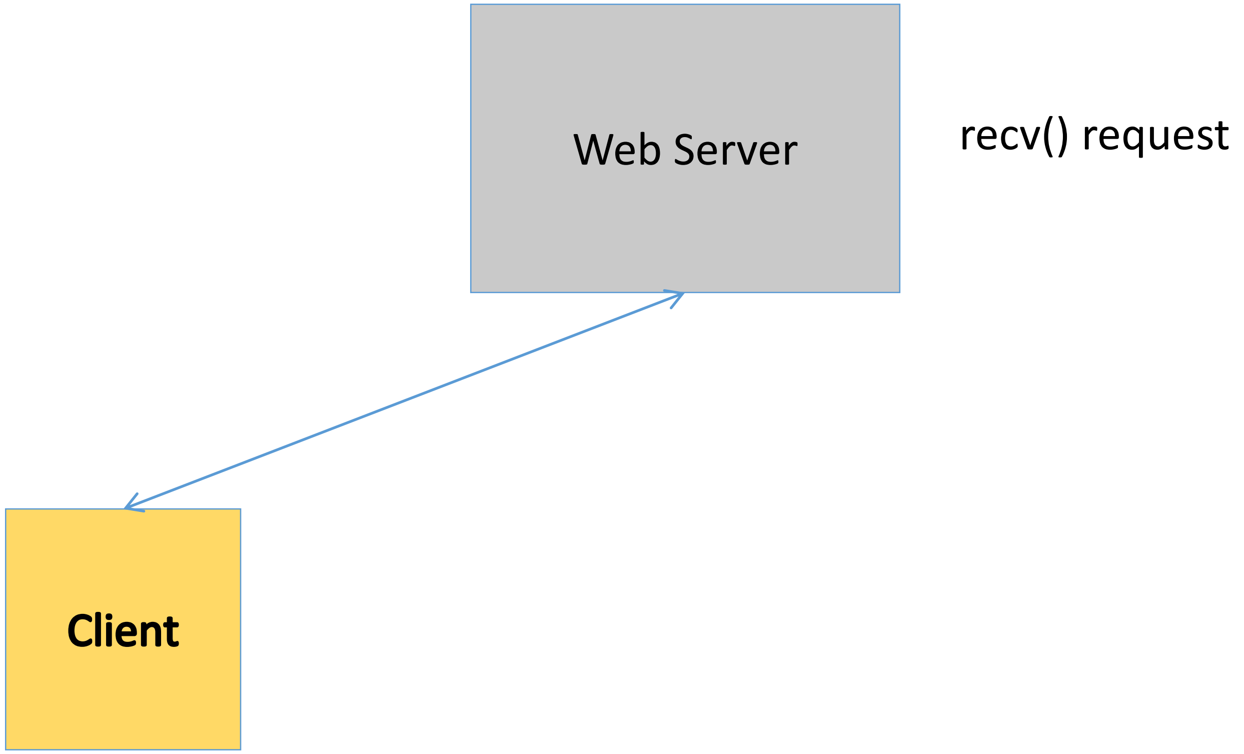
Concurrency

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



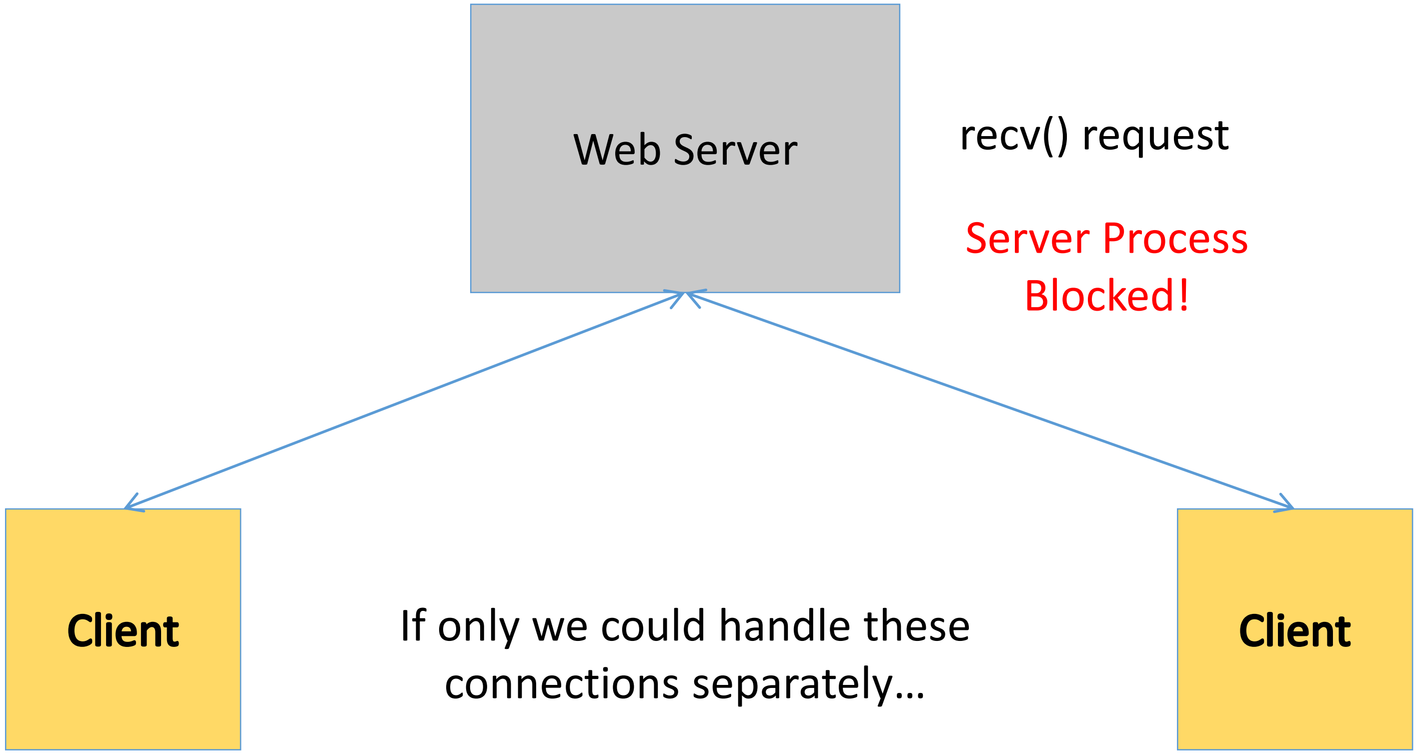
Without Concurrency

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



Without Concurrency

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



recv() request

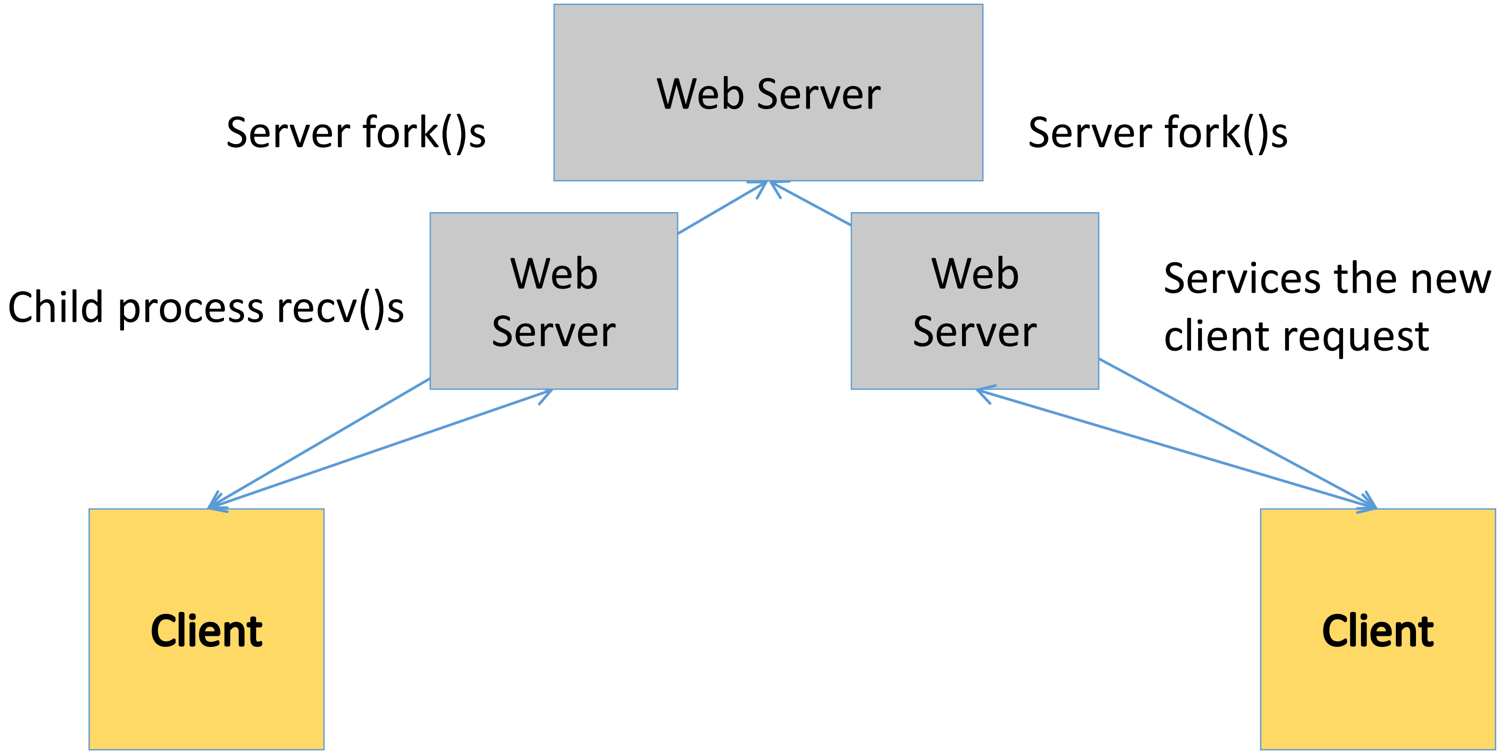
Server Process Blocked!

If only we could handle these connections separately…

Client taking its time…

Ready to send, but server still blocked on first client.

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

Proce[...]
(More de[...]

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

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

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

Whether processes or 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 the most critical?

A. Modular code/separation of concerns.

B. Multiple CPU/core parallelism.

C. I/O overlapping.

D. Some other benefit.

Whether processes or 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

- A socket can be put into "non blocking" mode
  - For a single call to send/recv, pass flag (MSG_DONTWAIT)
  - To apply to socket for all calls, use `fcntl` (file control)

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

(always check result – 0 on success)

Non-blocking I/O

- With O_NONBLOCK set on a socket (or MSG_DONTWAIT flag)
  - 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

How about…

```
server_socket = socket(), bind(), listen()
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?

```
server_socket = socket(), bind(), listen()
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
}
```

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.

Event-based Concurrency

- Rather than checking over and over, let the OS tell us when data can be read/written

- Create set of FDs 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).

`select()`

select()

- More interesting example in the select_tut man page.
- Beej's guide also has a good example.
- You'll use it in a future lab!

```
int main(void) {
    fd_set rfds;
    struct timeval tv;
    int retval;

   /* Watch stdin (fd 0) to see when it has input. */
    FD_ZERO(&rfds);
    FD_SET(0, &rfds);

   /* Wait up to five seconds. */
    tv.tv_sec = 5;
    tv.tv_usec = 0;

   retval = select(1, &rfds, NULL, NULL, &tv);
    /* Don't rely on the value of tv now! */

   if (retval == -1)
        perror("select()");
    else if (retval)
        printf("Data is available now.\n");
        /* FD_ISSET(0, &rfds) will be true. */
    else
        printf("No data within five seconds.\n");
}
```

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

Concurrency, so far…

Threads/Processes

- Create a new process/thread each time a new connection arrives

- One thread per connection

Event-based Concurrency

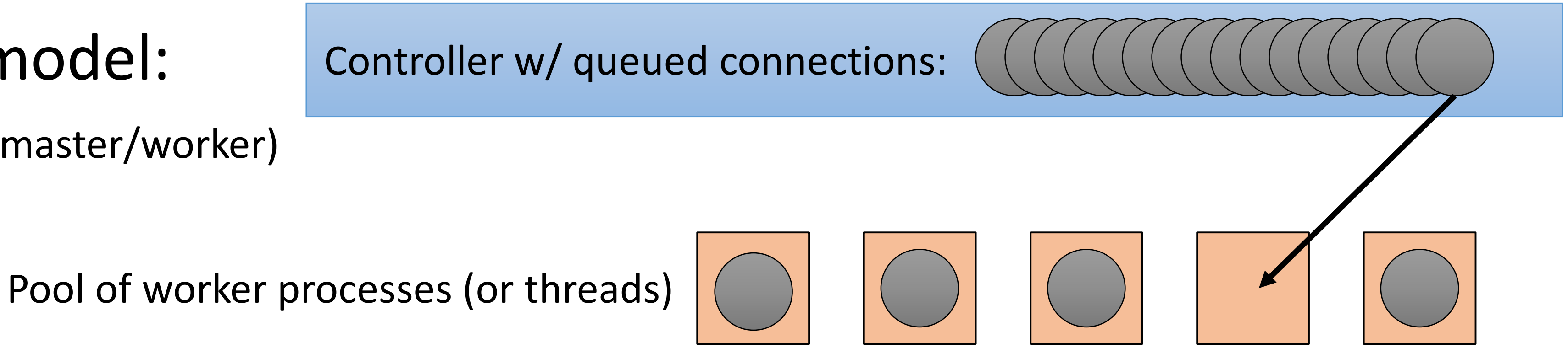
- Add sockets to descriptor set, use `select` to wait until one of them can do something

- One thread in total

Other Concurrency Patterns

Work Queue model:

(a.k.a boss/worker or master/worker)



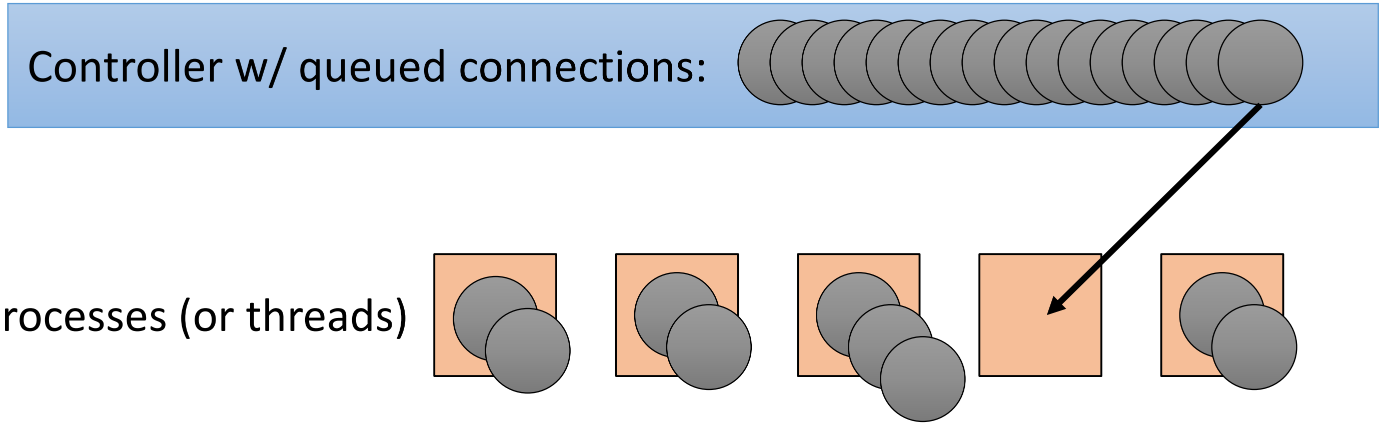
- Create many threads once and reuse them.

Each worker can perform I/O and block independently of the other.
Each worker can fail independently without stopping the system.

Other Concurrency Patterns

Work Queue model:

(a.k.a boss/worker or master/worker)



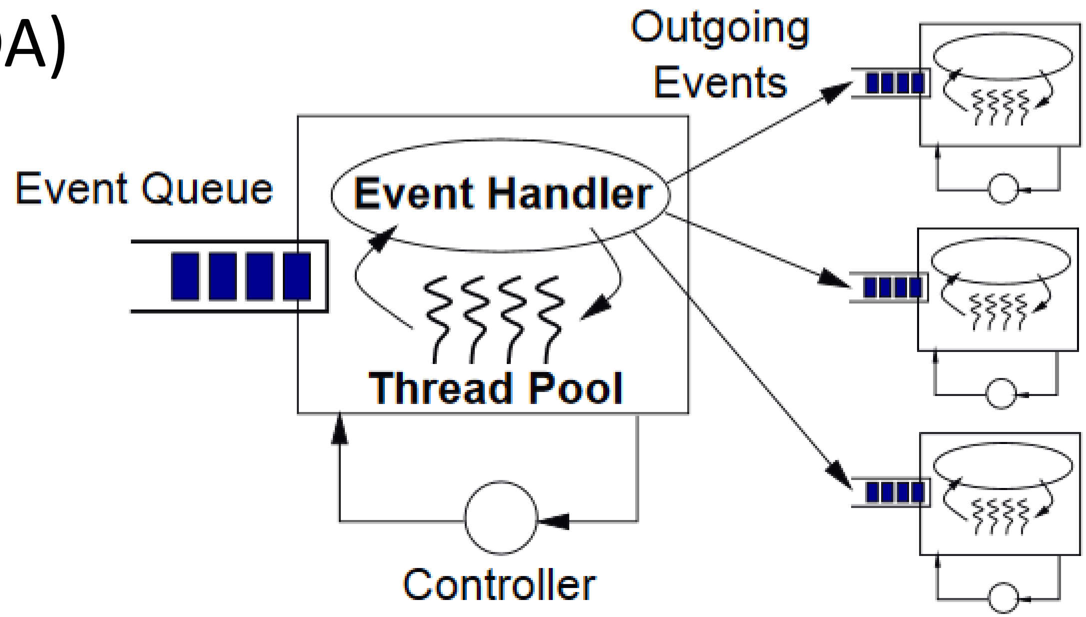
Pool of worker processes (or threads)

- More complex: each thread takes several connections and uses event-based concurrency to handle its subset

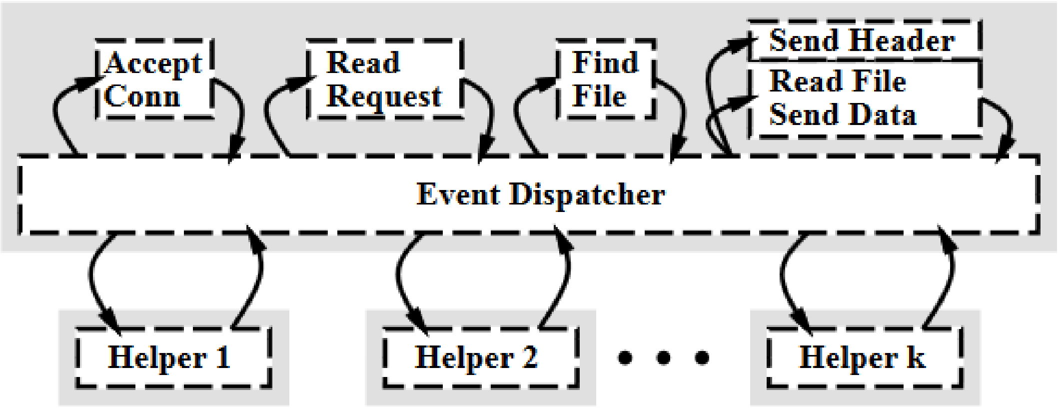
Each worker can perform I/O and block independently of the other.
Each worker can fail independently without stopping the system.

Many Other Models!

- Staged Event-Driven Architecture (SEDA)



- Asymmetric Multi-Process Event-Driven (AMPED)



Summary

- A network enables communication between processes
  - Many ways to structure communication, most require shared memory
  - For networks, we use sockets, which allows OS to buffer data
- OS manages socket buffers on behalf of processes
  - Asking for an operation that can't be performed will block the process
    - e.g., recv() from empty buffer or send() to full buffer
- Because blocking pauses the caller, must carefully structure apps