Inter-Process Communication (IPC)

Kevin Webb

Swarthmore College

February 5, 2019

Today’s Goals

- Discuss general coordination between executing processes.
    - Why and how it happens.
- Describe common IPC design models.
- Evaluate common IPC mechanisms and their tradeoffs.
- For now: processes. Much of this is applicable to threads too.

Why is it important for processes to be able to communicate?

A. Performance

B. Modularity

C. Fault tolerance

D. Some other reason(s) (such as?)

E. More than one of these (which?)

Concurrency

- Single CPU: logical concurrency
- Multiple CPU cores (commonly 4-16): still WAY more processes
- With multiple cores (more hardware), performance is clearly important. There are other benefits too though!

If one CPU core can run a program at a rate of X, how quickly will the program run on two cores?

A. Slower than one core (<X)

B. The same speed (X)

C. Faster than one core, but not double (X-2X)

D. Twice as fast (2X)

E. More than twice as fast(>2X)

Non-Performance Benefits

- Modularity: divide a task among specialized processes
  - develop / debug / test independently
  - reusable processes for other tasks

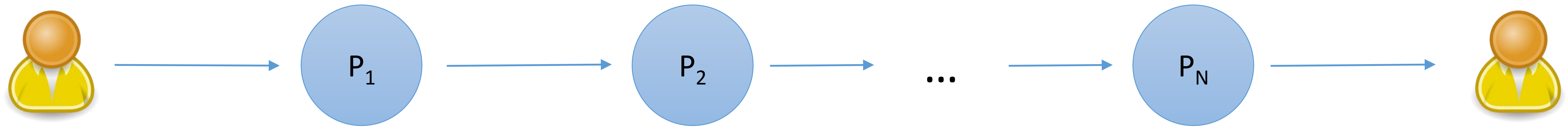
- Fault tolerance: if one process fails, user can interact with another (typically more distributed systems than OS…)

- I/O and blocking: if one process performs I/O and blocks, other(s) can keep executing.

Inter-Process Communication Models

- Common design patterns for coordinating processes (or threads).

1. Pipeline model:



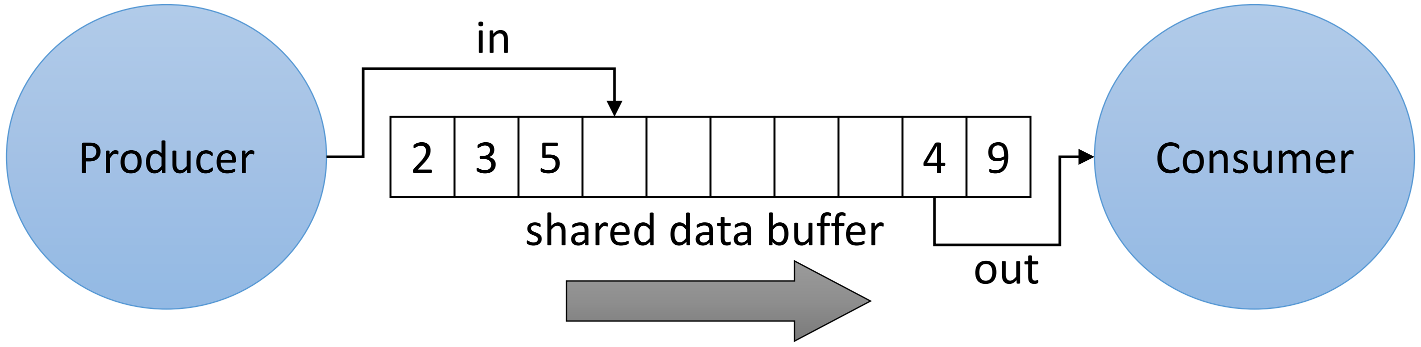
Example: `$ ls | sort | grep py`

"Unix philosophy": Do one thing, and do it well. (Modularity)
Result: lots of small utilities chained together at the command line.

Inter-Process Communication Models

* Common with threads too.

2. Producer / Consumer model:



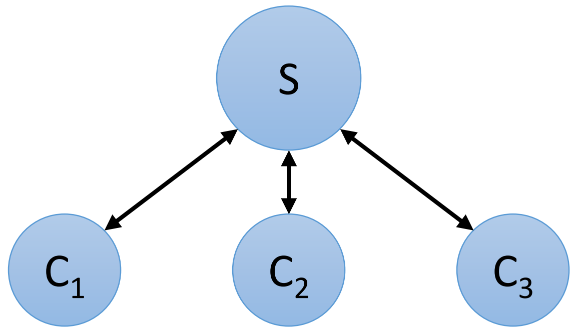
Example: media player

- Producer: read file from disk into buffer
- Consumer: decode file and send to output device

Each side can perform I/O and block independently of the other.

Inter-Process Communication Models

3. Client / Server model:



Example: Most Internet services

- Server: wait for clients to request service, satisfy requests when they do
- Client: connect to server, make request(s), disconnect when done

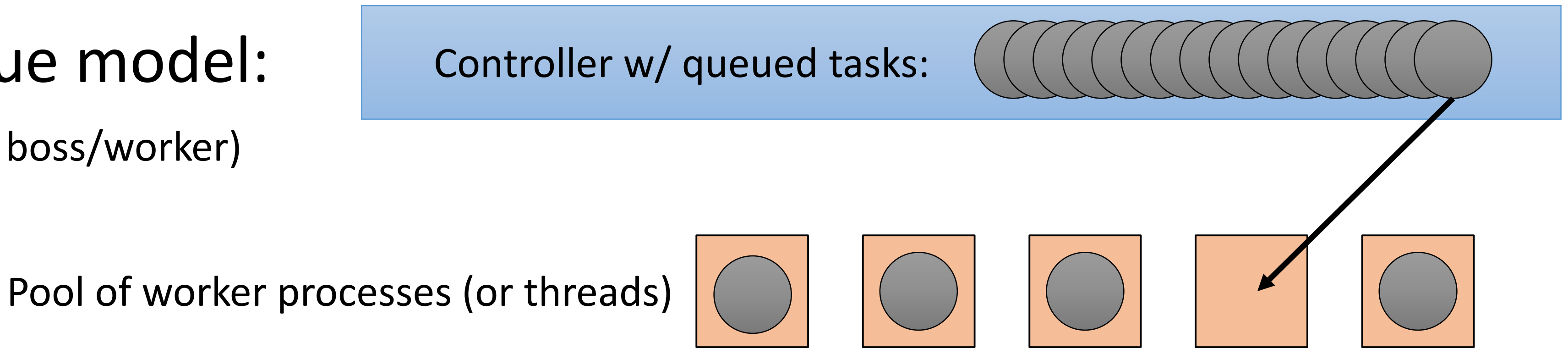
Each party (client and server) is specialized. (Modularity)
Server can do work for some clients while others are idle. (Independent I/O)

Inter-Process Communication Models

* Common with threads too.

4. Work Queue model:

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


Controller w/ queued tasks:
Pool of worker processes (or threads)

Examples:

- Folding@home: distributed biochemistry research
- Internet server (web, email, etc.): tasks are client requests

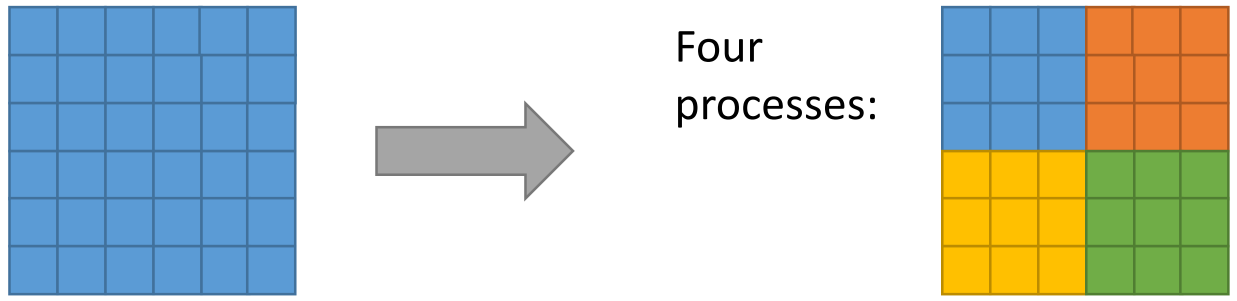
Each worker can perform I/O and block independently of the other.
Each worker can fail independently without stopping the system.

Inter-Process Communication Models

* Common with threads too.

5. Divide and Conquer model:

One process:



Four processes:

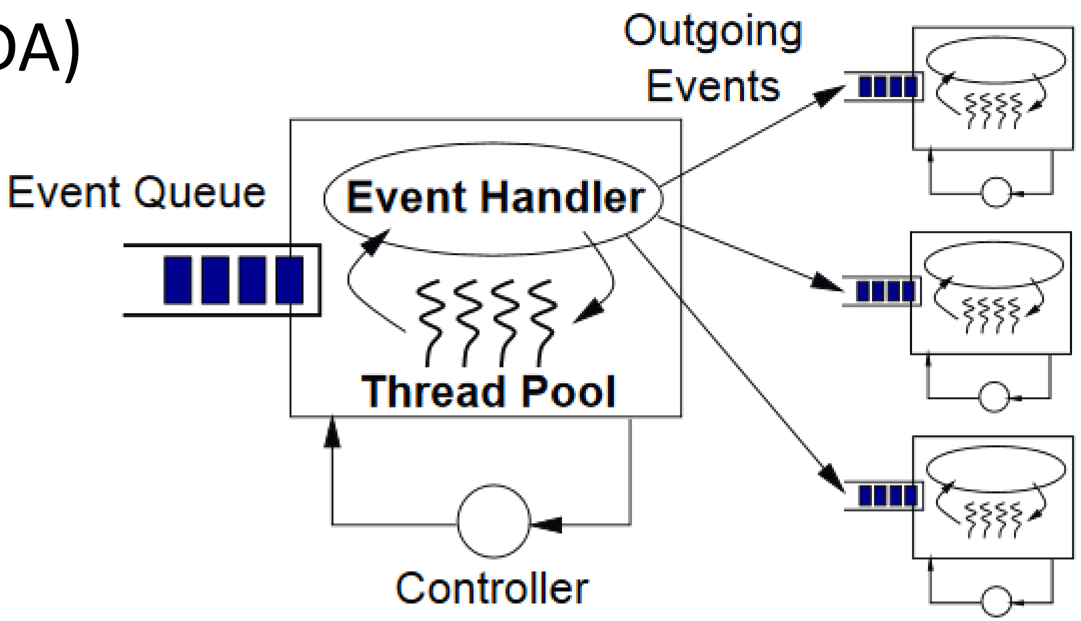
Example: Large scientific computing tasks

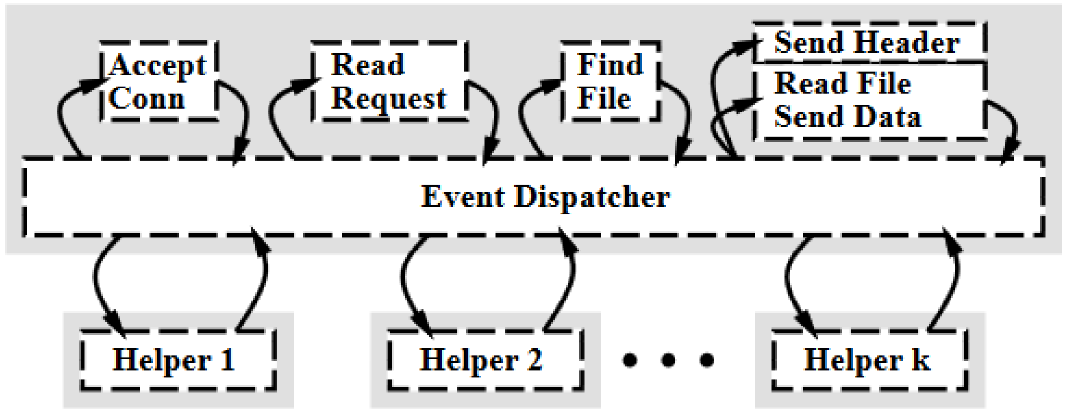
- Weather / earthquake / fluid dynamics simulations
- CS 31: Your parallel game of life lab…

Each piece can execute concurrently. (Performance)

Many Other Models!

- Peer-to-peer: processes communicate directly with peer processes
- Staged Event-Driven Architecture (SEDA)
- Asymmetric Multi-Process Event-Driven (AMPED)





Requirements

- Regardless of model, communication requires:

1. Data transfer: move data between processes

2. Synchronization: control execution order
    - arrange events to happen at the same time (or ensure that they don't)
    - Example: P_1 prints "ABC", P_2 prints "DEF" to shared terminal
        - Without synchronization: ABCDEF, ADBECF, ABDECF, ADEBCF, …
        - With synchronization (atomicity): ABCDEF or DEFABC

Communication Classes

Shared Memory

- OS maps the same physical memory frame(s) into the VAS of multiple processes.
- Interface: memory access (read or set variables in memory).
- Synch: explicit synchronization types (mutex, condition var, etc.)

Message Passing

- Processes ask OS to transfer data to/from other processes.
- Interface: make system calls to `send()` data to or `recv()` data from other process.
- Synch: implicit, based on ordering of `send()` & `recv()` calls.

Which feels more natural to you? Which do you think is most common? Why?

	More Natural	Most Common
A	Shared Memory	Shared Memory
B	Shared Memory	Message Passing
C	Message Passing	Shared Memory
D	Message Passing	Message Passing

Shared memory: OS maps the same physical memory frame(s) into the VAS of multiple processes.

Message passing: Processes ask OS to transfer data to/from other processes.

Going Forward…

- Present an IPC abstraction and how it works
- Whether it’s built using shared memory or message passing (or either)
- Where it’s used (e.g., single machine vs. over a network)
- When it’s used (e.g., example applications)

“Portable Operating System Interface”, also the P in Pthreads

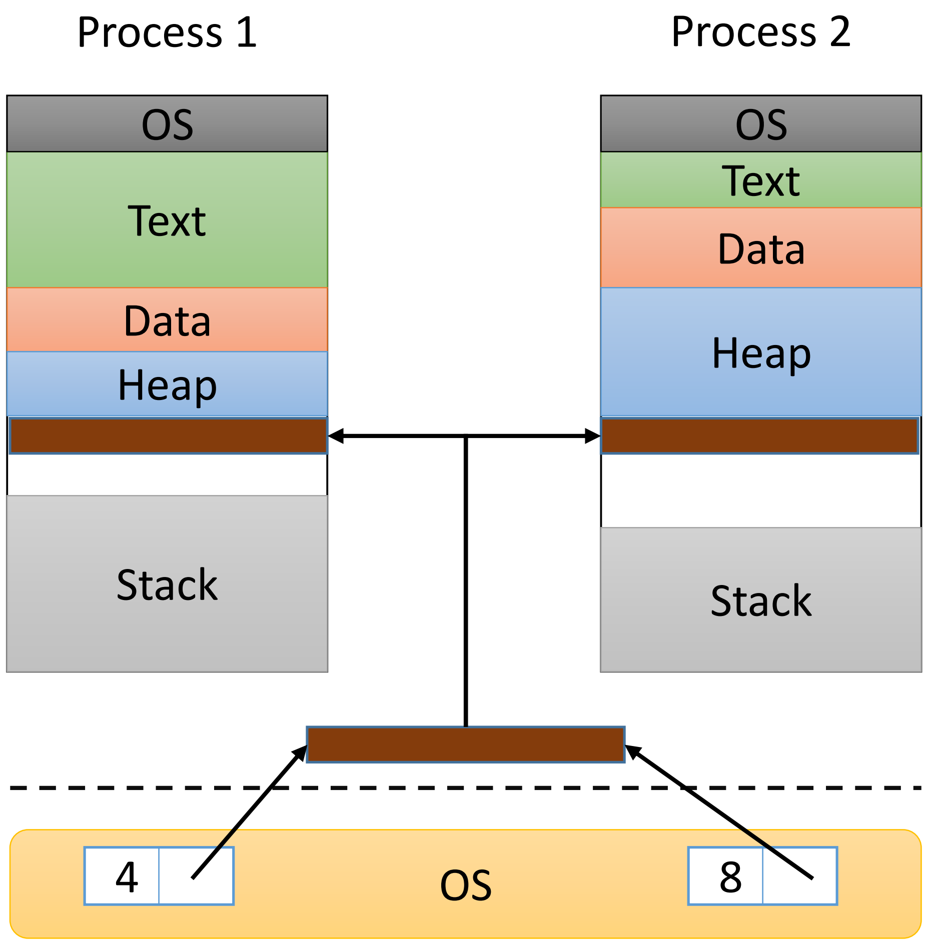
POSIX Shared Memory (similar alternative: SysV shared memory)

- Explicitly request a chunk of memory to be shared.

```
int fd = shm_open(name, …);
ftruncate(fd, 8192);
void *ptr = mmap(…, fd, …);
```

- Only works on shared hardware.
- I could find no app examples.

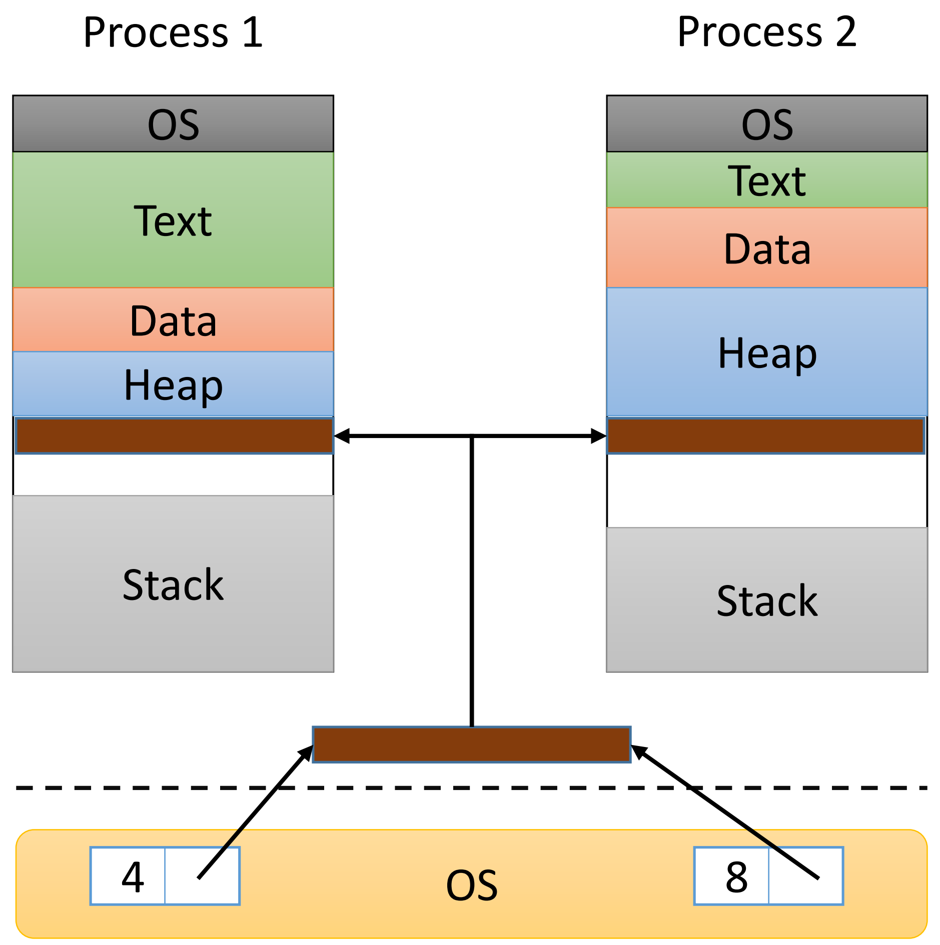
Apps that would have used this have largely switched to using threads.



“Portable Operating System Interface”, also the P in Pthreads

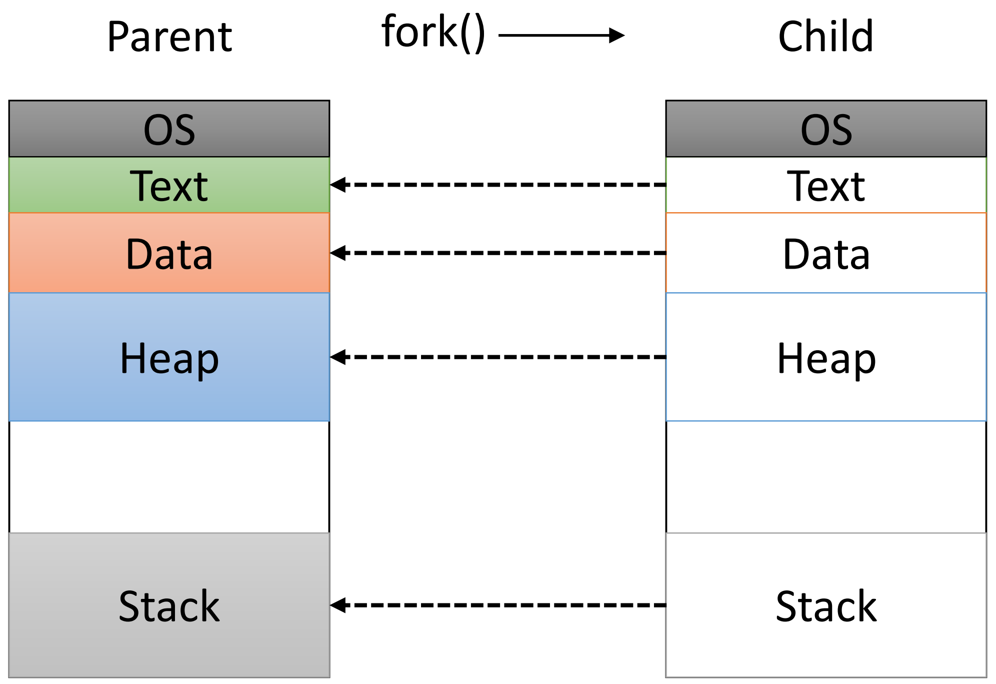
POSIX Shared Memory (similar alternative: SysV shared memory)

- Explicitly request a chunk of memory to be shared.
- Data transfer: read/write shared memory
- Synchronization: explicit variables (mutex locks, condition variables, barriers, etc.)



Implicit Shared Memory: fork()

- When new process is created, it shares a (read only) copy of its parent's memory.
- Copy-on-write (COW) – only make a private copy of memory when process attempts to write. (Why?)
- Only works on shared hardware.
- Used frequently (every process creation!), e.g., shell commands.

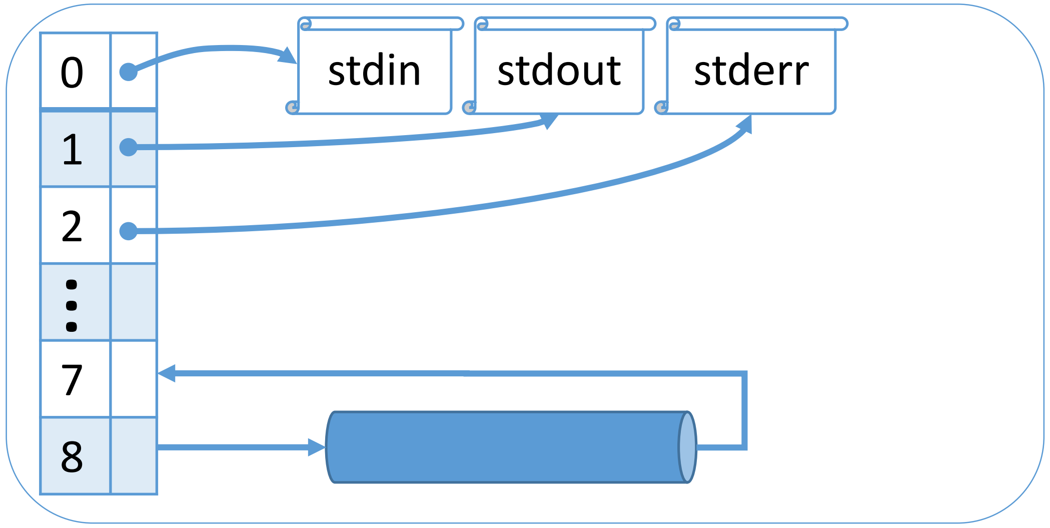


Usually isn't used for long-term communication, but it does cause shared memory.

Pipes

- Create two file descriptors – one for input, one for output.

```
int pipefds[2];
pipe(pipefds);
fork(); // Child inherits descriptors
// One process reads from pipefds[0]
// One process writes to pipefds[1]
```



Descriptor Table

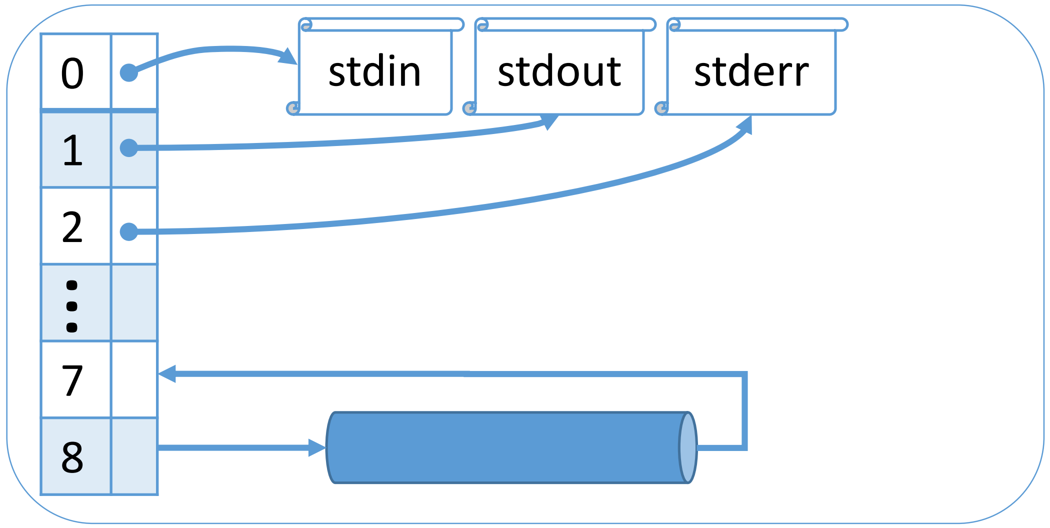
- Only works on shared hardware (relies on fork behavior)

Pipes and the Shell

- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`

"Take the output of ls and send it as the input to sort."

Note: ls writes to stdout, and sort reads from stdin.

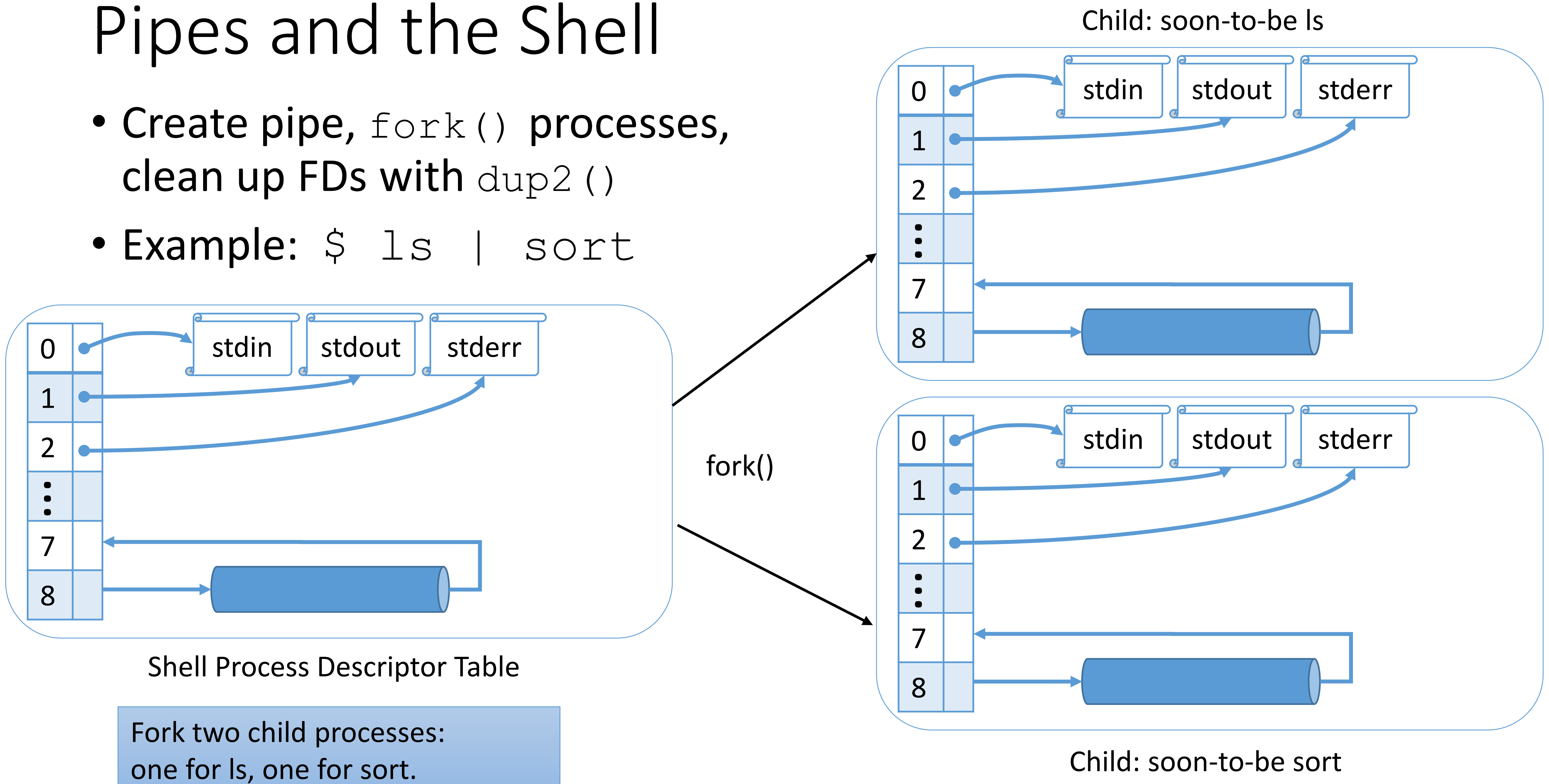


Shell Process Descriptor Table

Create a pipe with descriptors for reading (7) and writing (8).

Pipes and the Shell

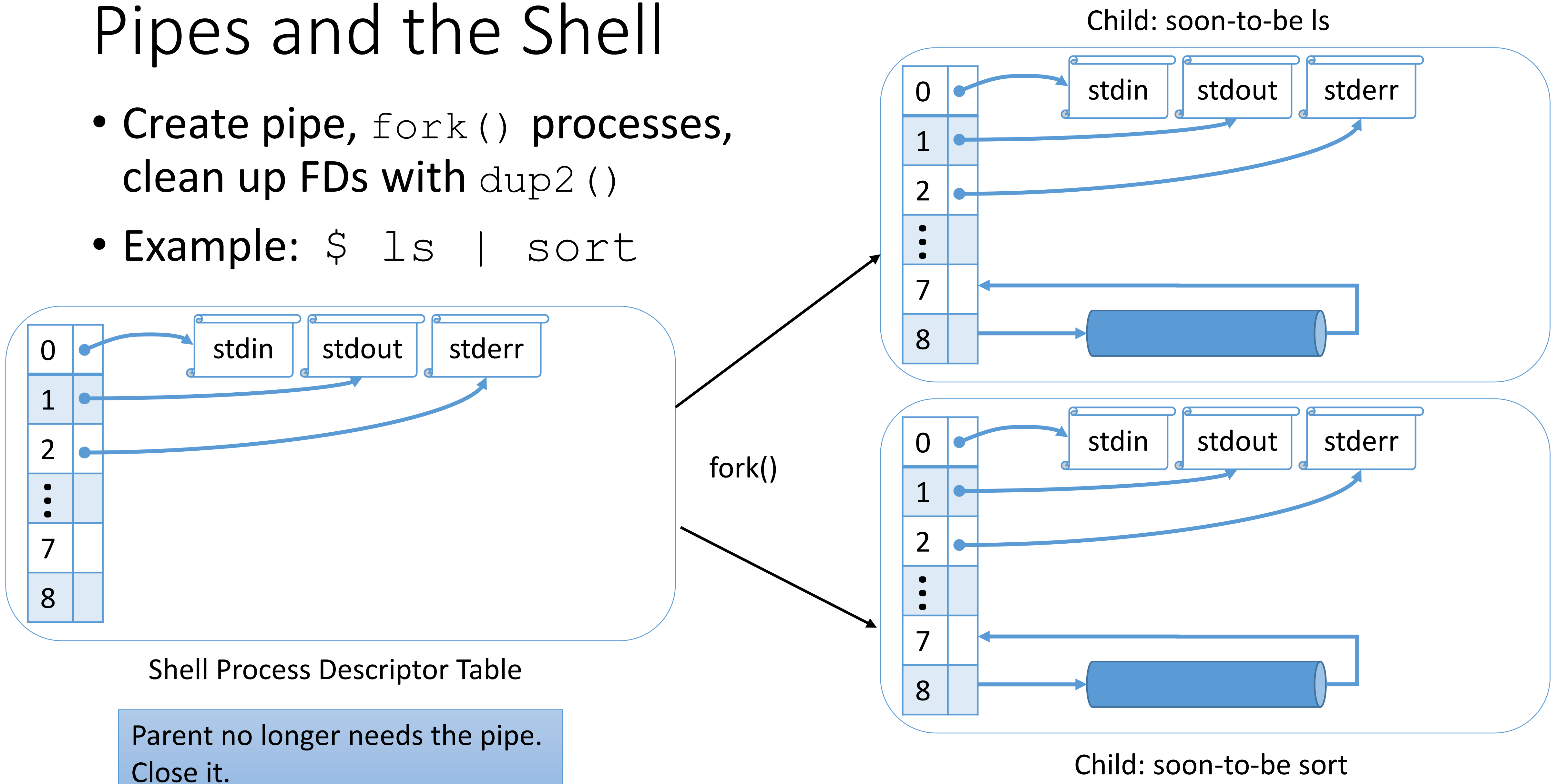
- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`



Fork two child processes:
one for ls, one for sort.

Pipes and the Shell

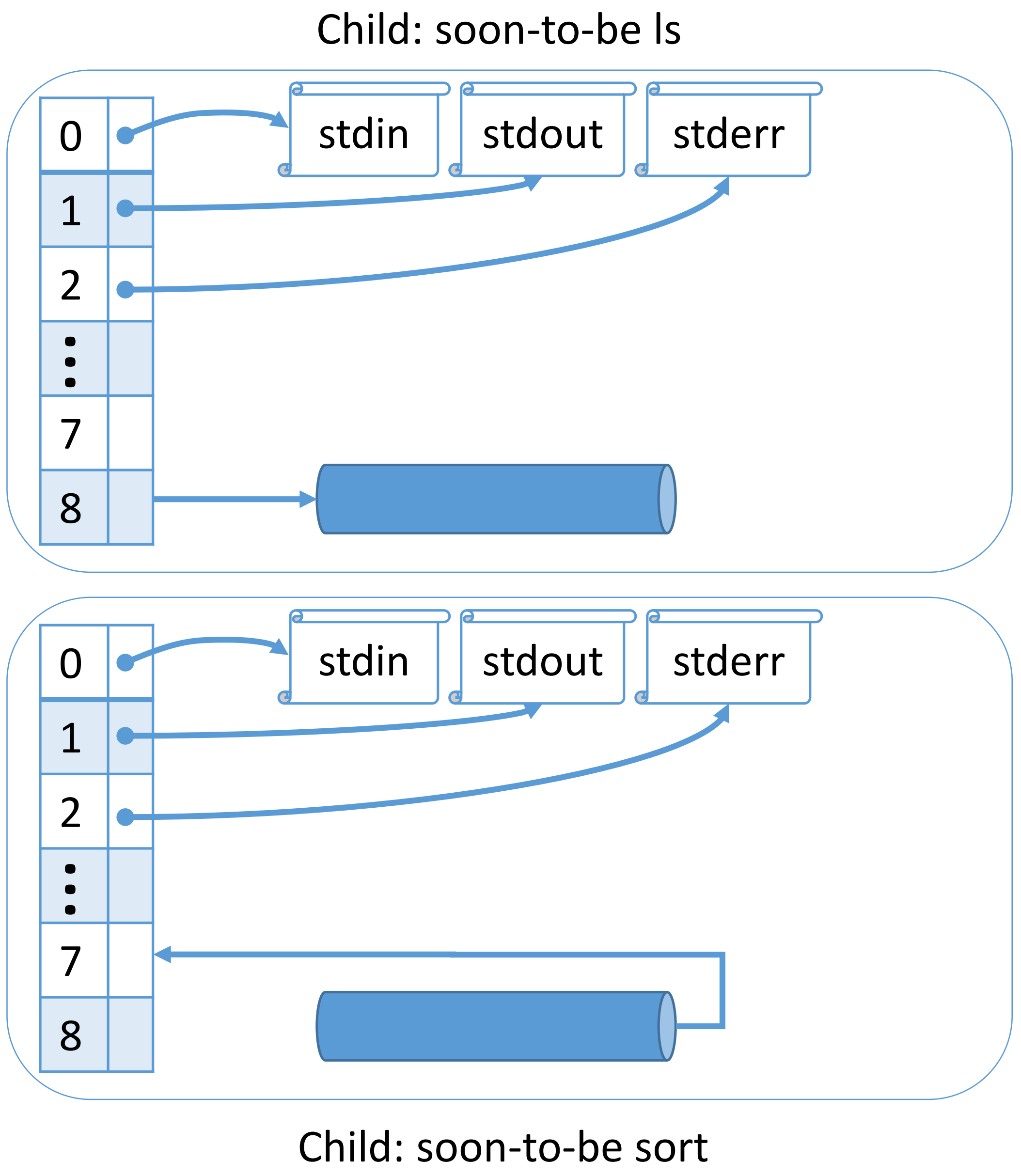
- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`



Parent no longer needs the pipe. Close it.

Pipes and the Shell

- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`



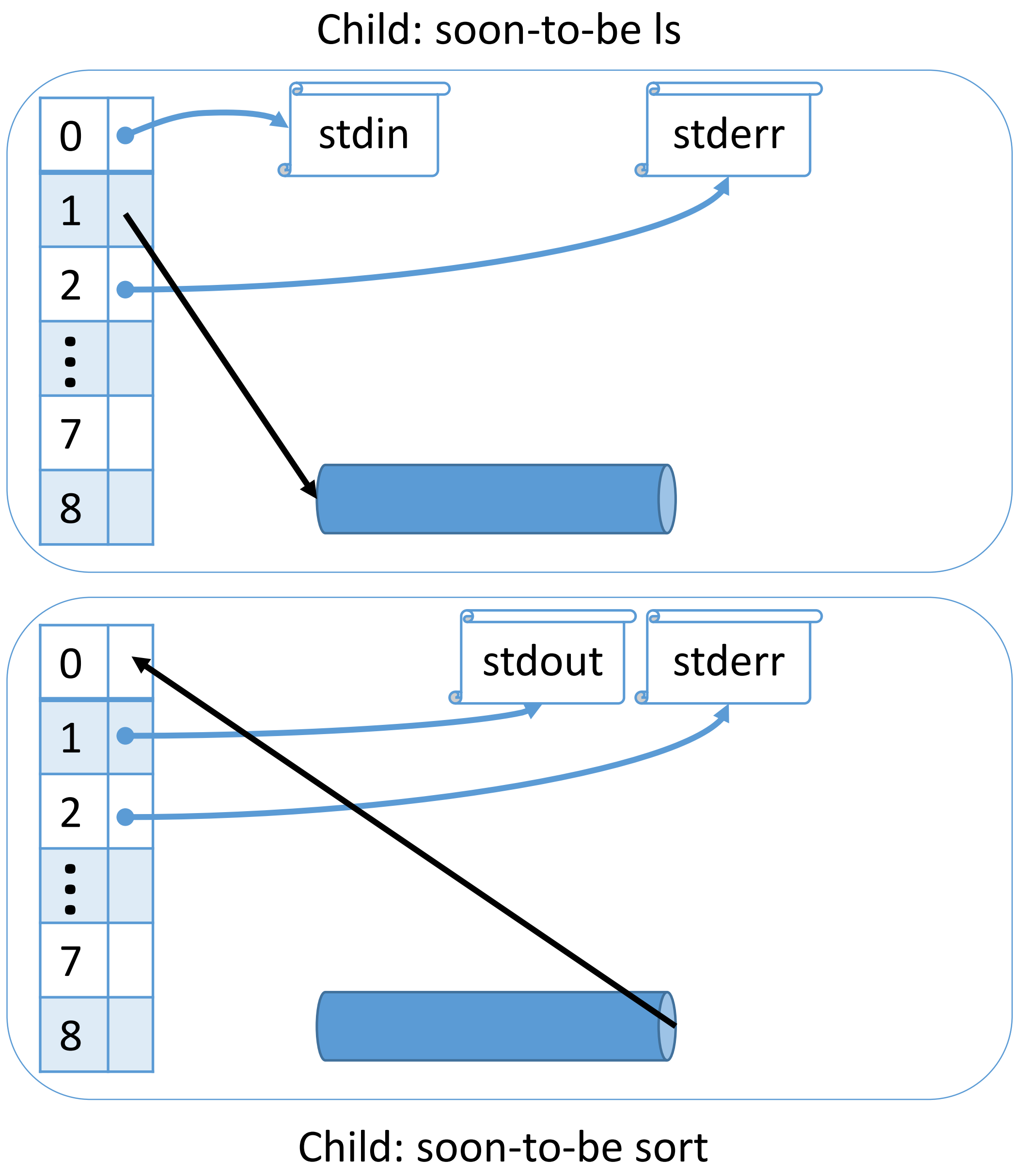
ls only needs to write, so it closes the read side. Sort needs only the read side.

Pipes and the Shell

- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`

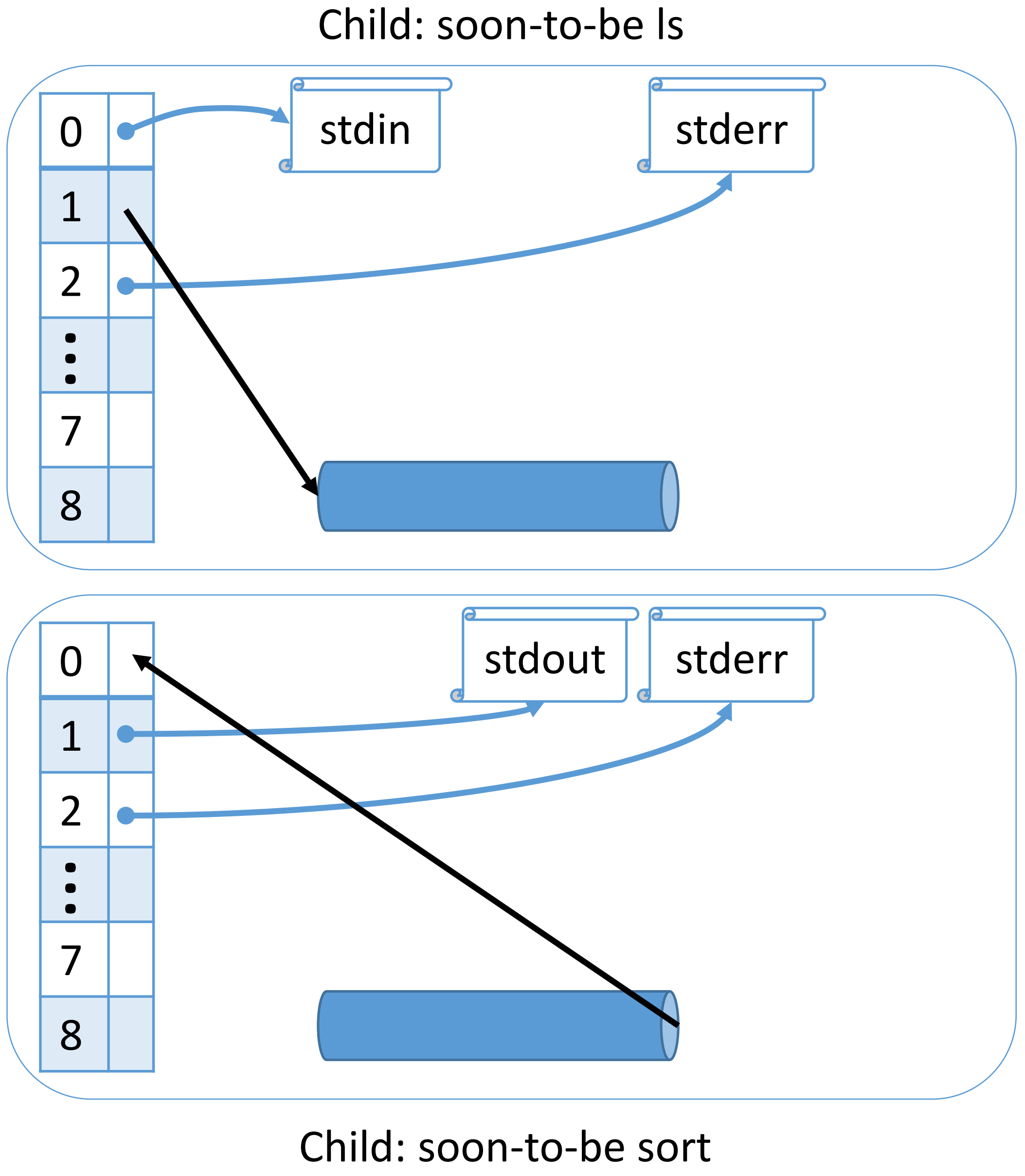
```
ls:   dup2(8, 1);

sort: dup2(7, 0);
```

Use dup2 to adjust FDs: close second FD and put first in its spot.



Pipes and the Shell

- Create pipe, `fork()` processes, clean up FDs with `dup2()`
- Example: `$ ls | sort`



Now, ls can printf or write to FD 1 just like it always does and output goes to pipe.

Likewise, sort can read from 0 like usual.

How would you implement pipes?

A. Shared memory (how? and why?)

B. Message passing (how? and why?)

Named Pipes (a.k.a. FIFOs)

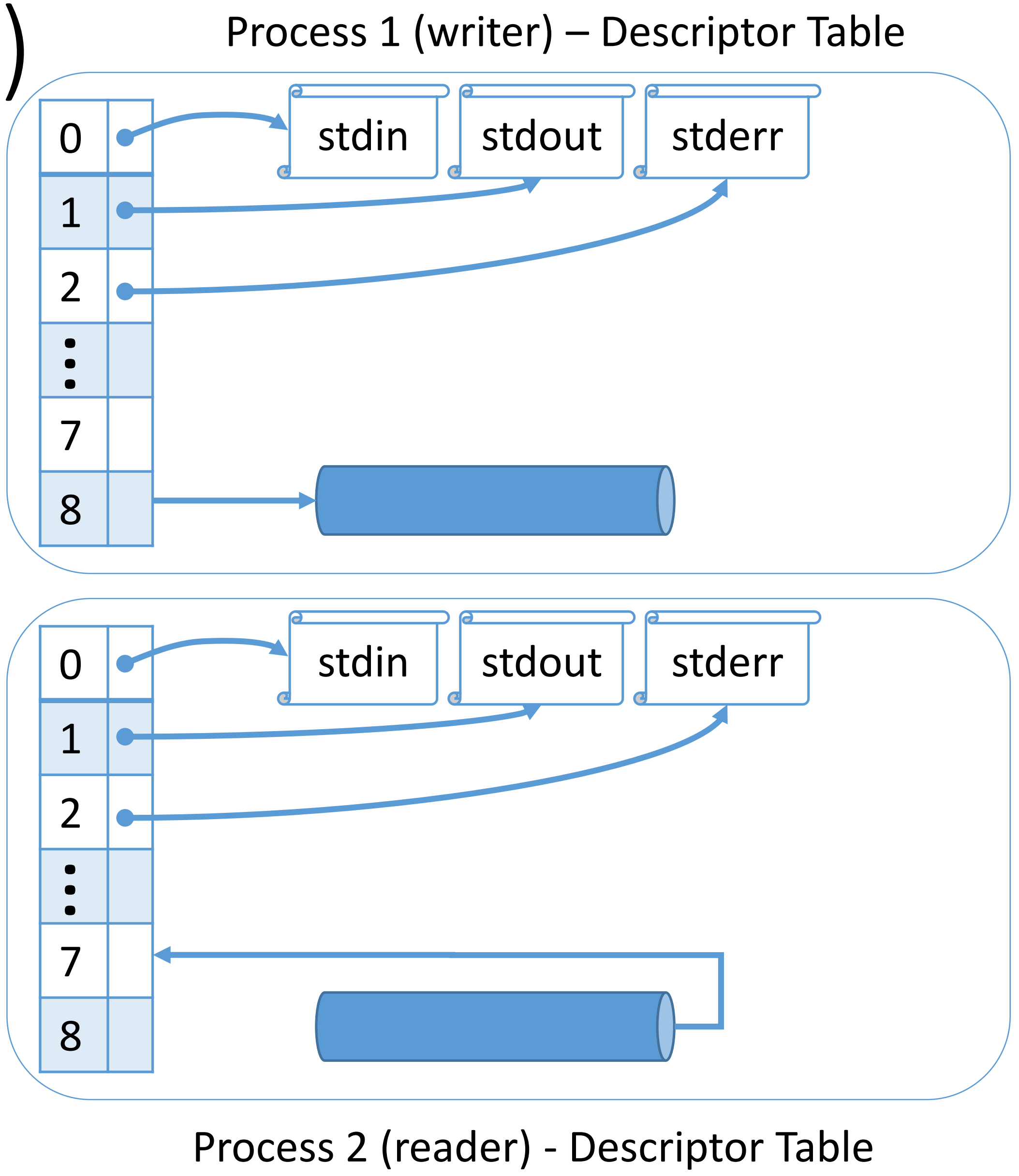
- Mechanism: same as pipe

- Does NOT rely on FD inheritance from process forking.

Represented in file system.

```
mkfifo(path, …);
open(path, …);
```

- Only works on shared hardware



Process 1 (writer) – Descriptor Table
0
1
2
⋮
7
8
stdin
stdout
stderr
0
1
2
⋮
7
8
stdin
stdout
stderr
Process 2 (reader) - Descriptor Table

Signals (UNIX)

- Coarse notifications: one integer
- Generates a “software interrupt” for target process
- Issued using `kill()` system call
- Only works on shared hardware

Signal Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)

Signals (UNIX)

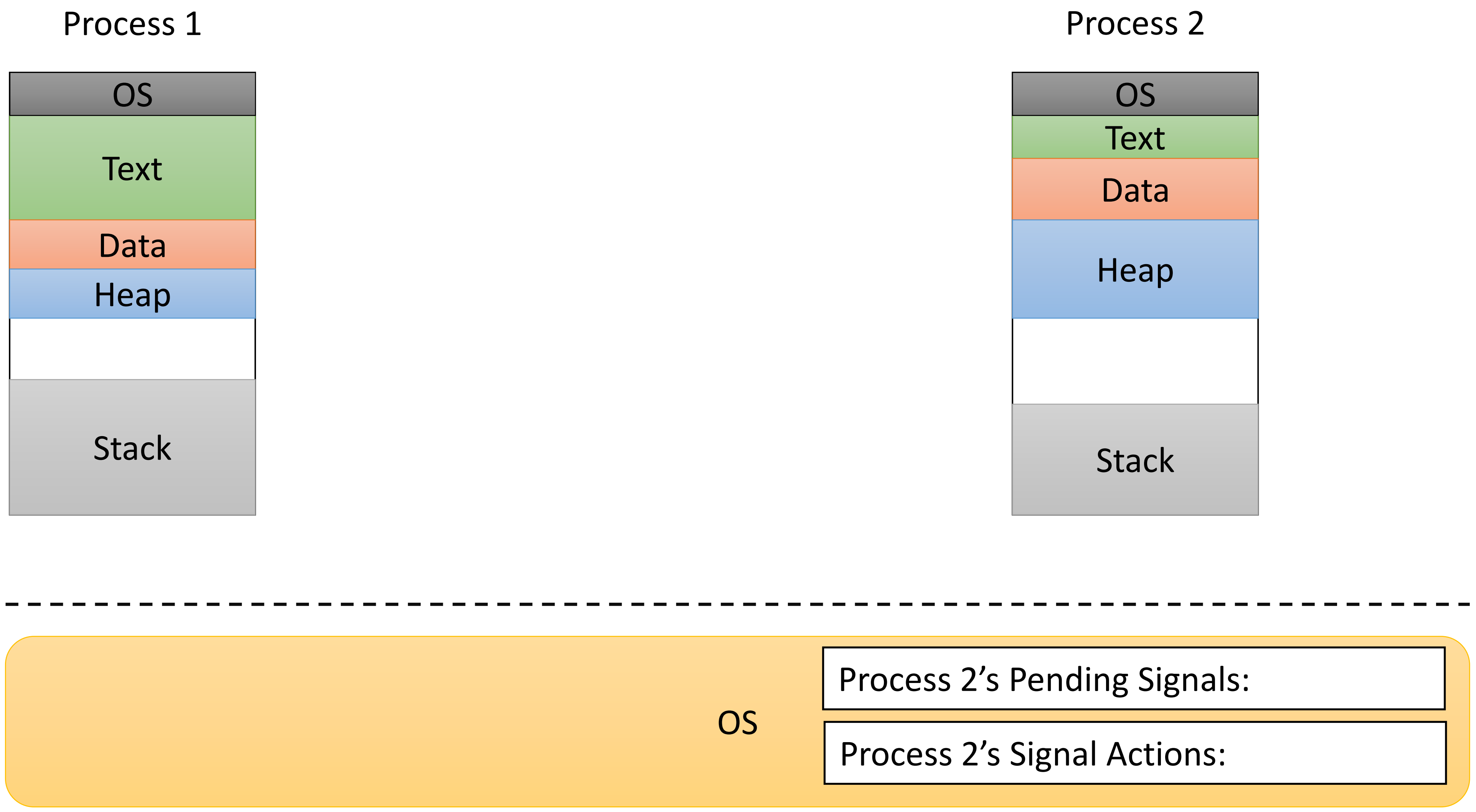
- OS keeps track of *pending signals* and *signal actions* for each process.

- Pending: signal has been "sent" to this process. (bit 0->1)
  - Signals are NOT queued. Each signal number is either pending or not.

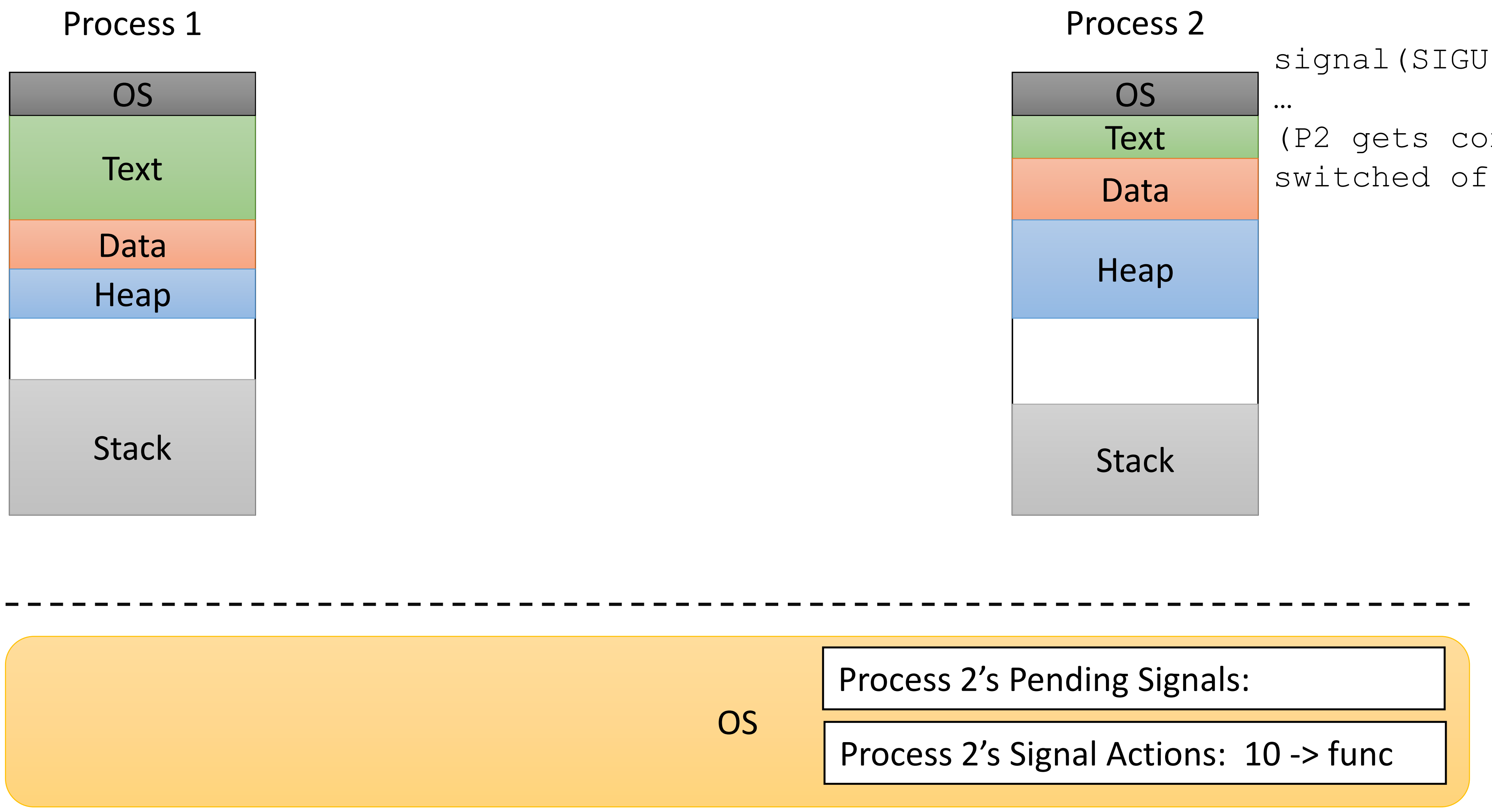
- Action: what to do in response. Each signal has a default, but you can override (most of) them. Choices: Default, Ignore, or Function

- <u>Kernel checks for, delivers pending signals when resuming a process.</u>

Signals (UNIX)



Signals (UNIX)

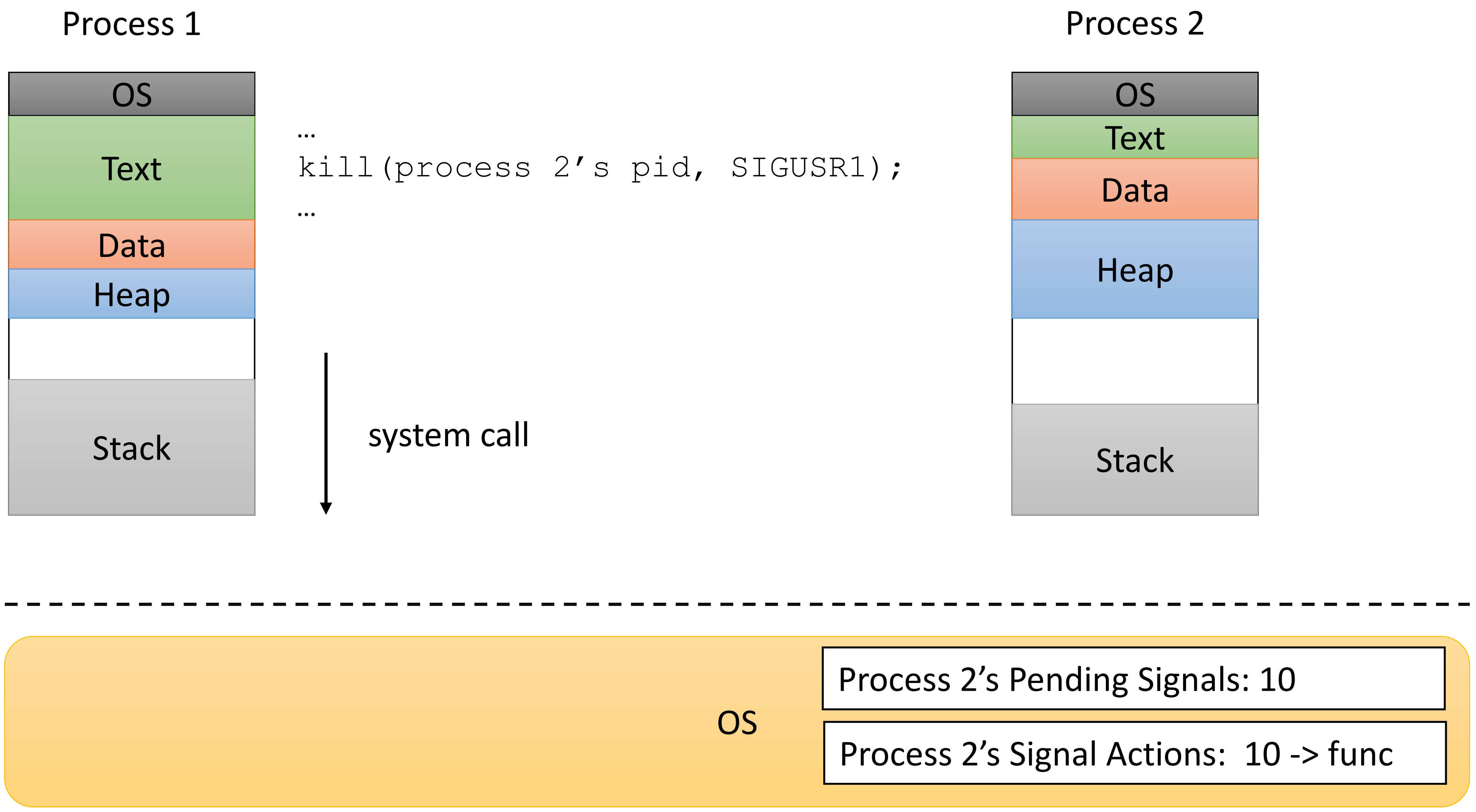


```
signal(SIGUSR1, func);
…
(P2 gets context
switched off CPU)
```

Process 2's Pending Signals:

Process 2's Signal Actions: 10 -> func

Signals (UNIX)

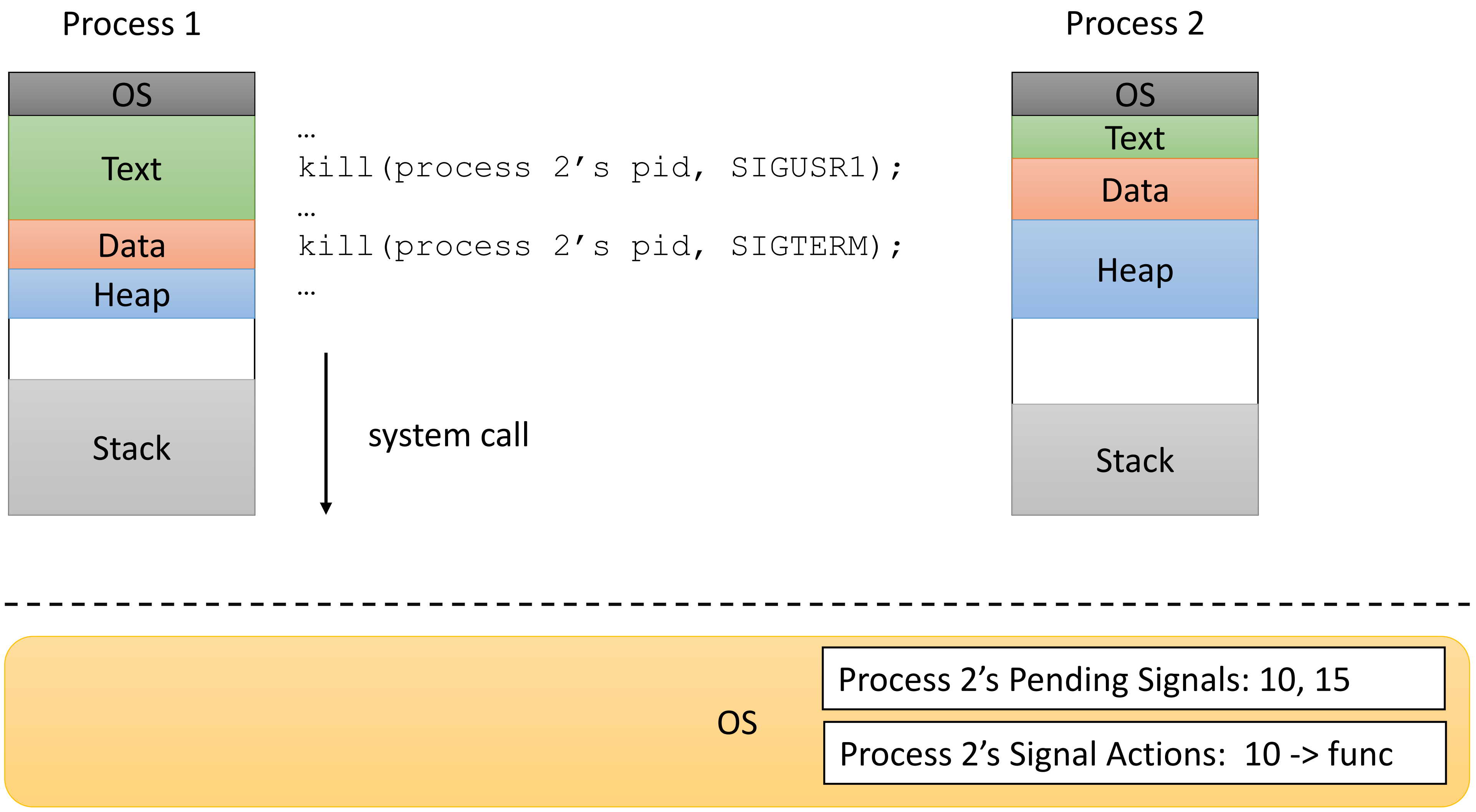


```
…
kill(process 2's pid, SIGUSR1);
…
```

Process 2's Pending Signals: 10

Process 2's Signal Actions: 10 -> func

Signals (UNIX)



```
…
kill(process 2’s pid, SIGUSR1);
…
kill(process 2’s pid, SIGTERM);
…
```

Process 2’s Pending Signals: 10, 15

Process 2’s Signal Actions: 10 -> func

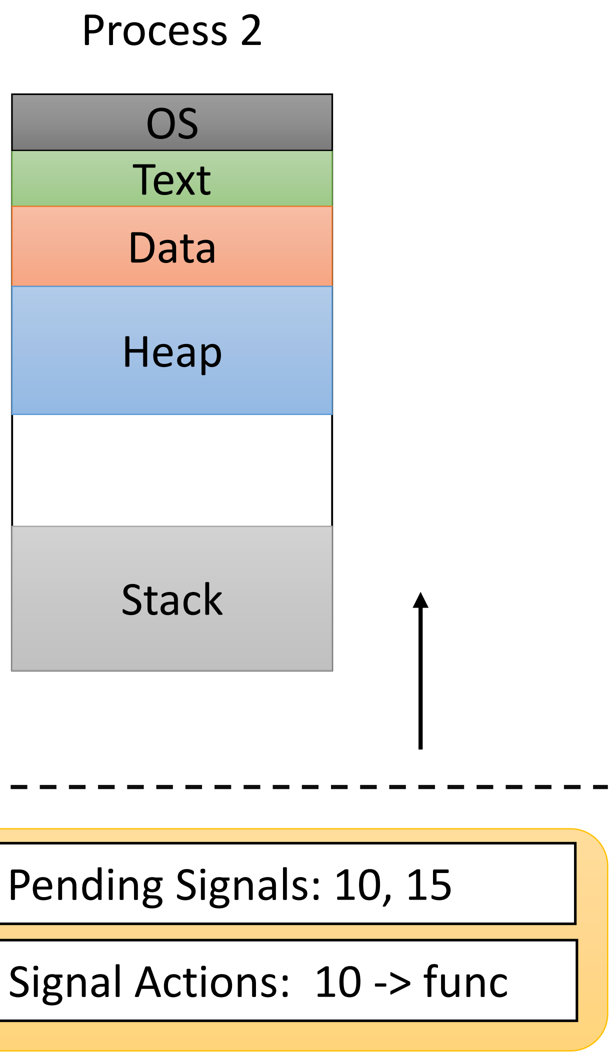
Signals (UNIX)

When process 2 resumes executing, deliver the signals.

(Don't count on any particular ordering.
For example: numerical order.)

For handling 10 (SIGUSR1), resume process by invoking the `func()` function.

For handling 15 (SIGTERM), use the default routine (terminate process).



Process 2's Pending Signals: 10, 15

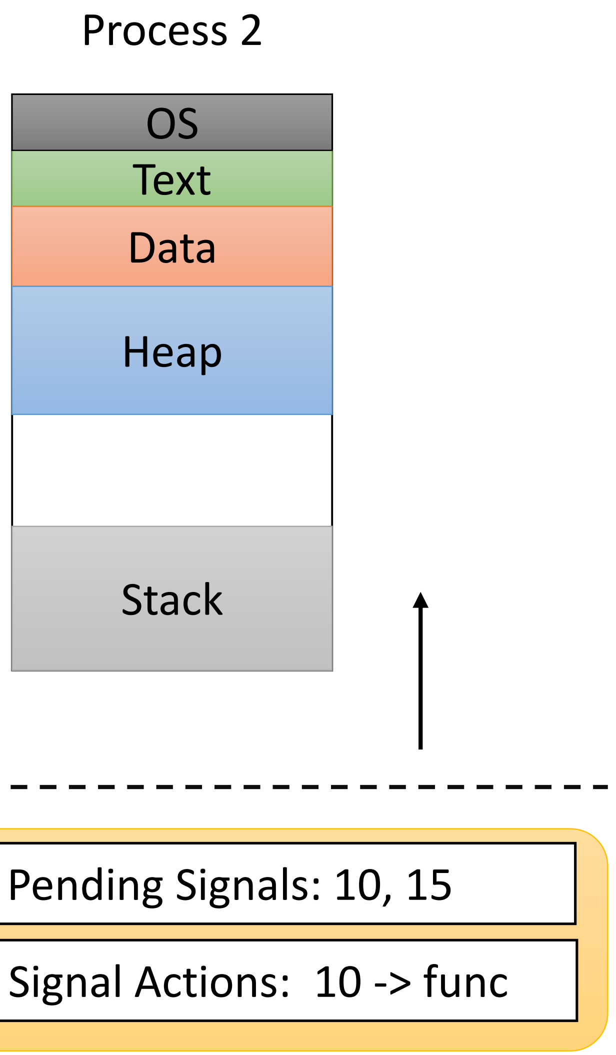
Process 2's Signal Actions: 10 -> func

Signals (UNIX)

- Data transfer: send one integer.

- Synchronization: OS marks signals as pending, delivers signals when it feels like it (when scheduler chooses target)

- Used for:
  - terminating processes
  - asking long-running processes to do maintenance
  - debugging
  - OS sending simple message to process (e.g., SIGCHLD, SIGPIPE)



Process 2's Pending Signals: 10, 15

Process 2's Signal Actions: 10 -> func

Going Forward…

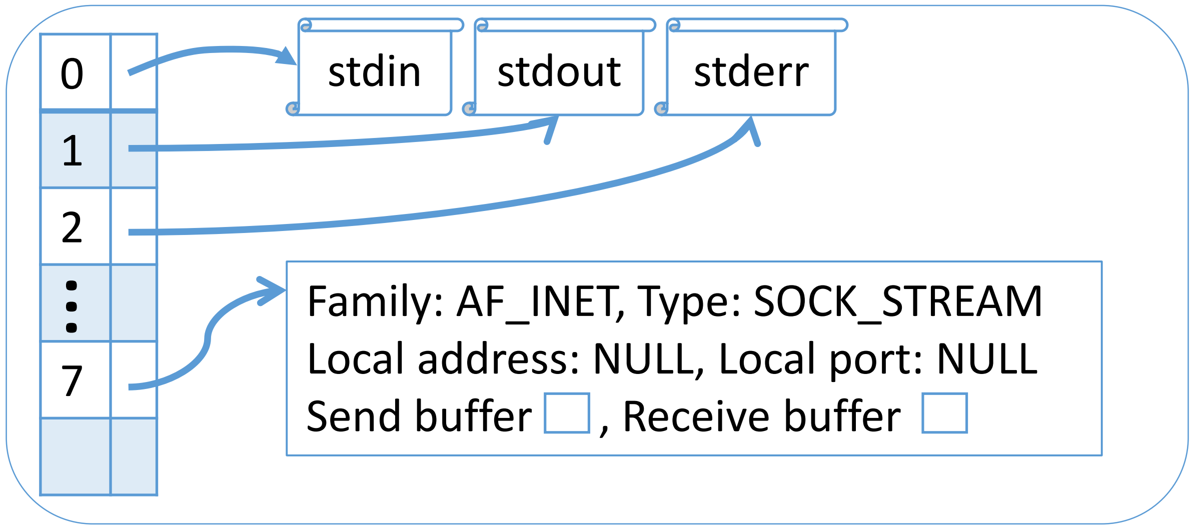
- Present an IPC abstraction and how it works
- Whether it’s built using shared memory or message passing (or either)
- Where it’s used (e.g., single machine vs. over a network)
- When it’s used (e.g., example applications)

Communicating Across Machines

- *Almost* always some form of message passing.
- OS Kernel buffers message data.
- Typically used over a network, but also works on a single machine.
    - client-server model VERY common

Sockets

- Abstraction: `socket()` descriptor
  - write data to one side of socket
  - read it at other
- Low-level data transfer interface
  - `send(fd, …)` and `recv(fd, …)`
- Most commonly (TCP):
  - one side `listen()`s for incoming connection
  - other side `connect()`s to listener

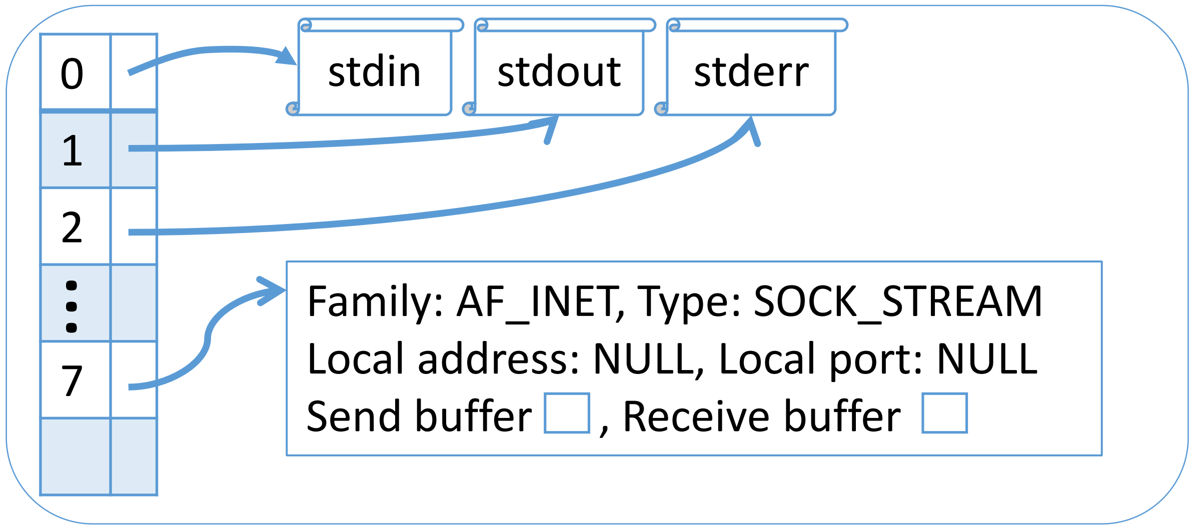

0
1
2
7
stdin
stdout
stderr
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer, Receive buffer

Descriptor Table

The subsequent cross-machine mechanisms we'll look at build upon sockets to provide higher-level abstractions.

Sockets

- Abstraction: `socket()` descriptor
  - write data to one side of socket
  - read it at other


0
1
2
⋮
7
stdin
stdout
stderr
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer , Receive buffer

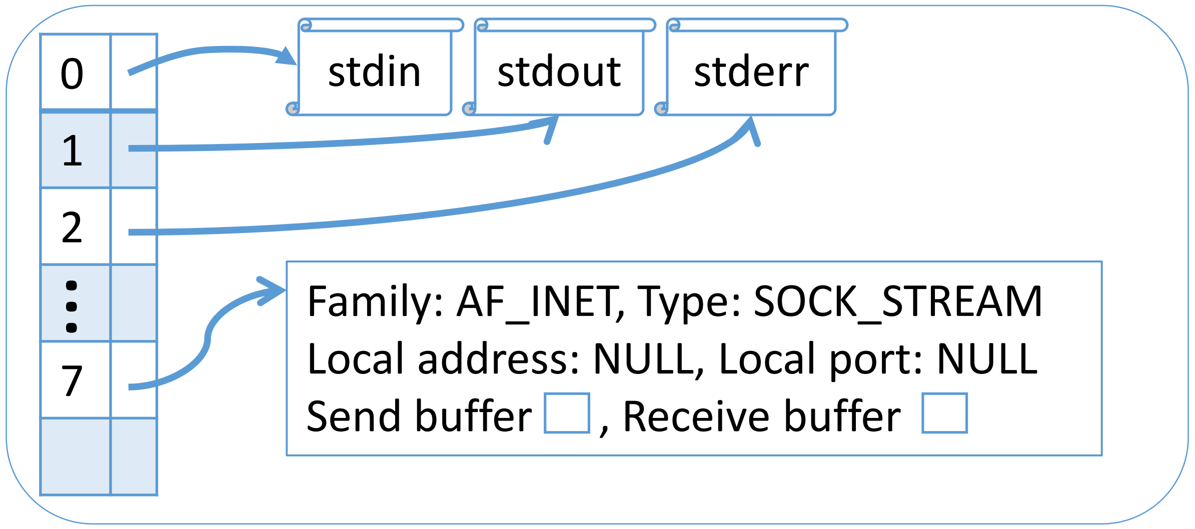
Descriptor Table

- Low-level data transfer interface
  - `send(fd, …)` and `recv(fd, …)`

- Data transfer: copy data to kernel, kernel will transmit (if necessary)
- Synchronization:?

Sockets

- Abstraction: `socket()` descriptor
  - write data to one side of socket
  - read it at other


0
1
2
⋮
7
stdin
stdout
stderr
Family: AF_INET, Type: SOCK_STREAM
Local address: NULL, Local port: NULL
Send buffer, Receive buffer
Descriptor Table

- Low-level data transfer interface
  - `send(fd, …)` and `recv(fd, …)`

- Data transfer: copy data to kernel, kernel will transmit (if necessary)
- Synchronization: kernel blocks process if operation can't be performed

“Message Passing Interface” (MPI)

- Typically used in scientific computing tasks / super computers.
- Each machine given a numeric “rank” ID by MPI library.
- Can send/recv like sockets by specifying rank of recipient.
- Can also broadcast, reduce values, add barriers, and many others

Details of data transfer and synchronization vary according to MPI implementation, available hardware, and configuration.

Remote Procedure Call (RPC)

- Works like a function call, but via message passing (e.g., over the network).
- Data transfer: name of function and parameters sent via socket.
- Synchronization: caller waits for server to return back to it.
- Used frequently:
  - NFS
  - Web services

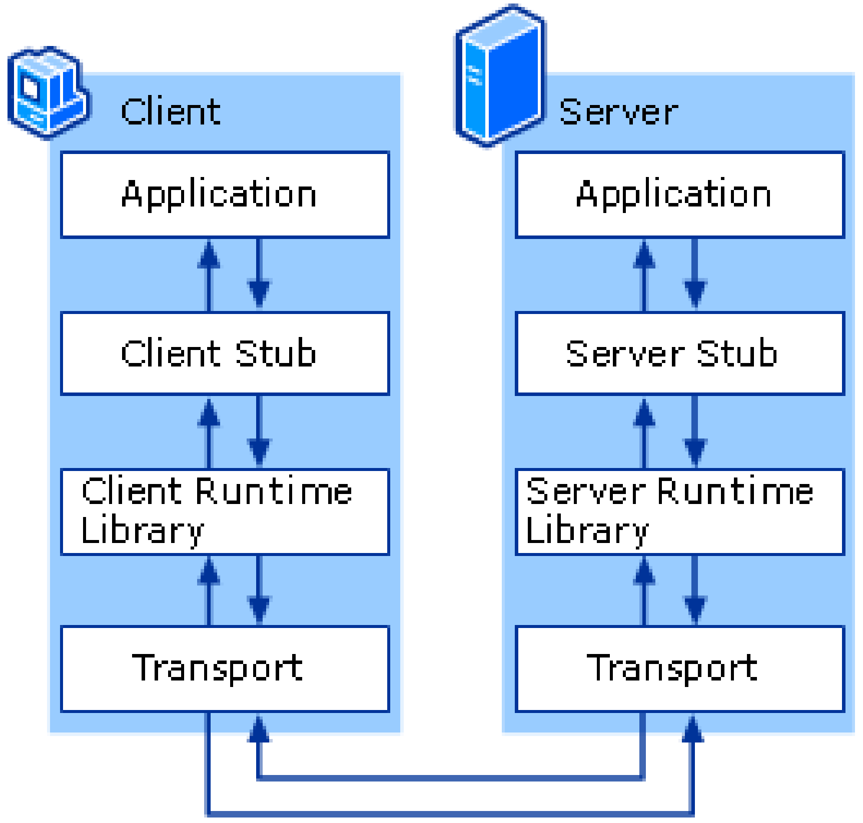


Image source:
https://technet.microsoft.com/en-us/library/cc738291(v=ws.10).aspx

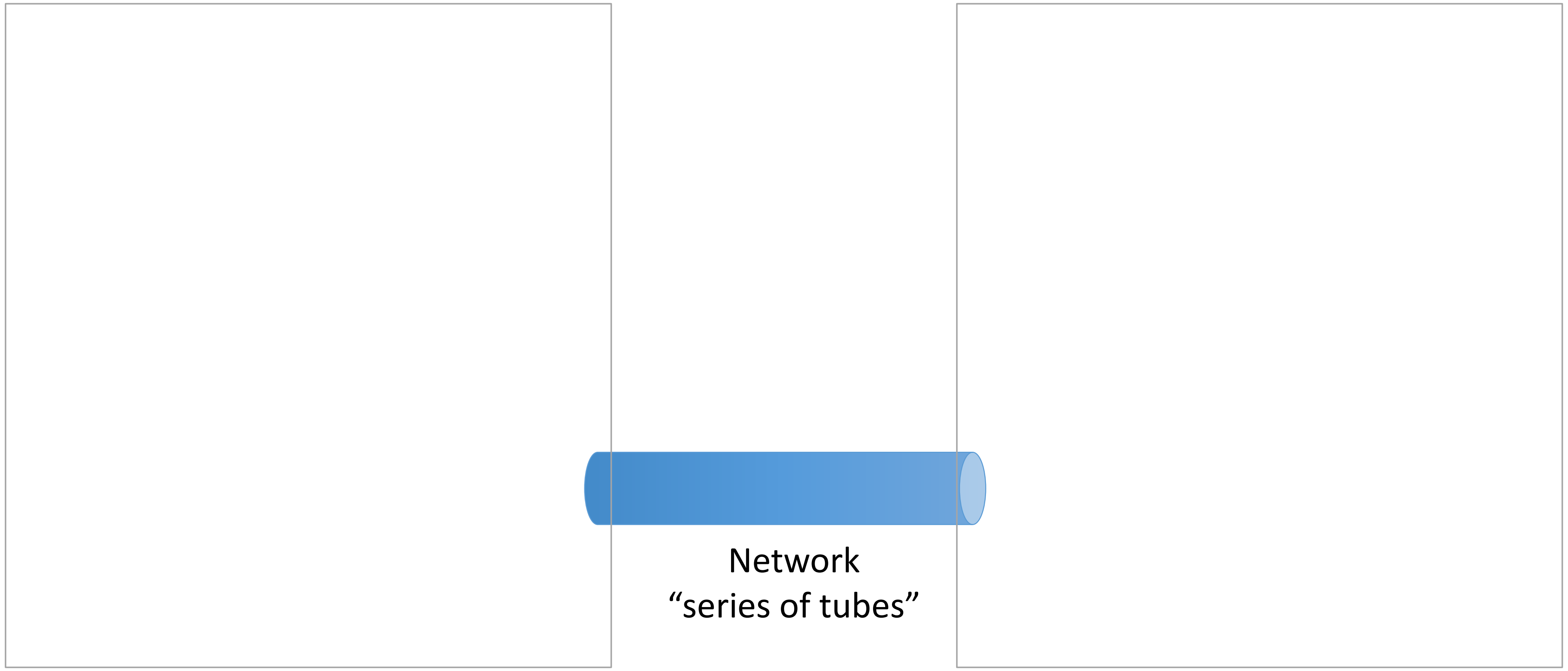
Message Passing: Buffering

- Applies to pipes, sockets, anything built on top of them, and message passing in general.

- Problem: Storing data that has been *sent* but not yet asked for by the receiver.

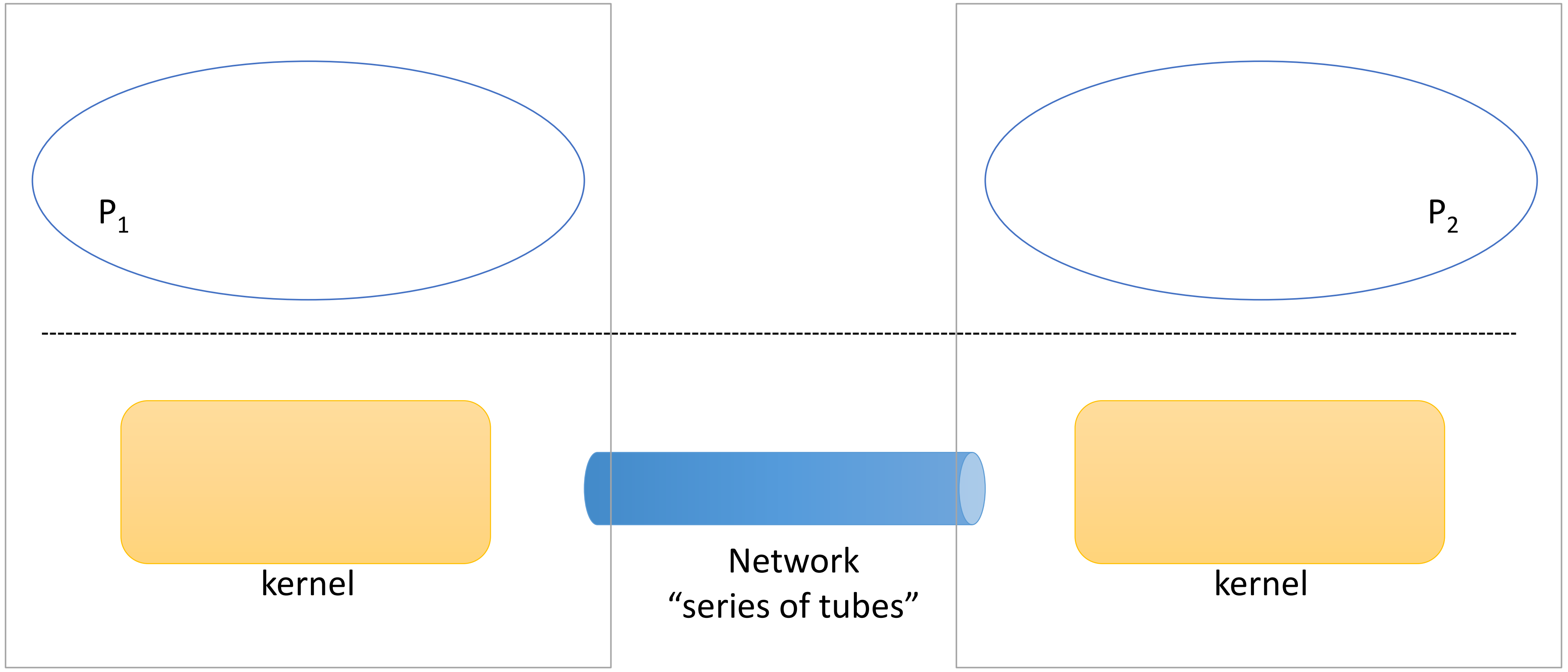
- Buffer status dictates synchronization:
  - buffer empty: read/recv will force process to wait (block)
  - buffer full: write/send will force process to wait (block)

Buffering



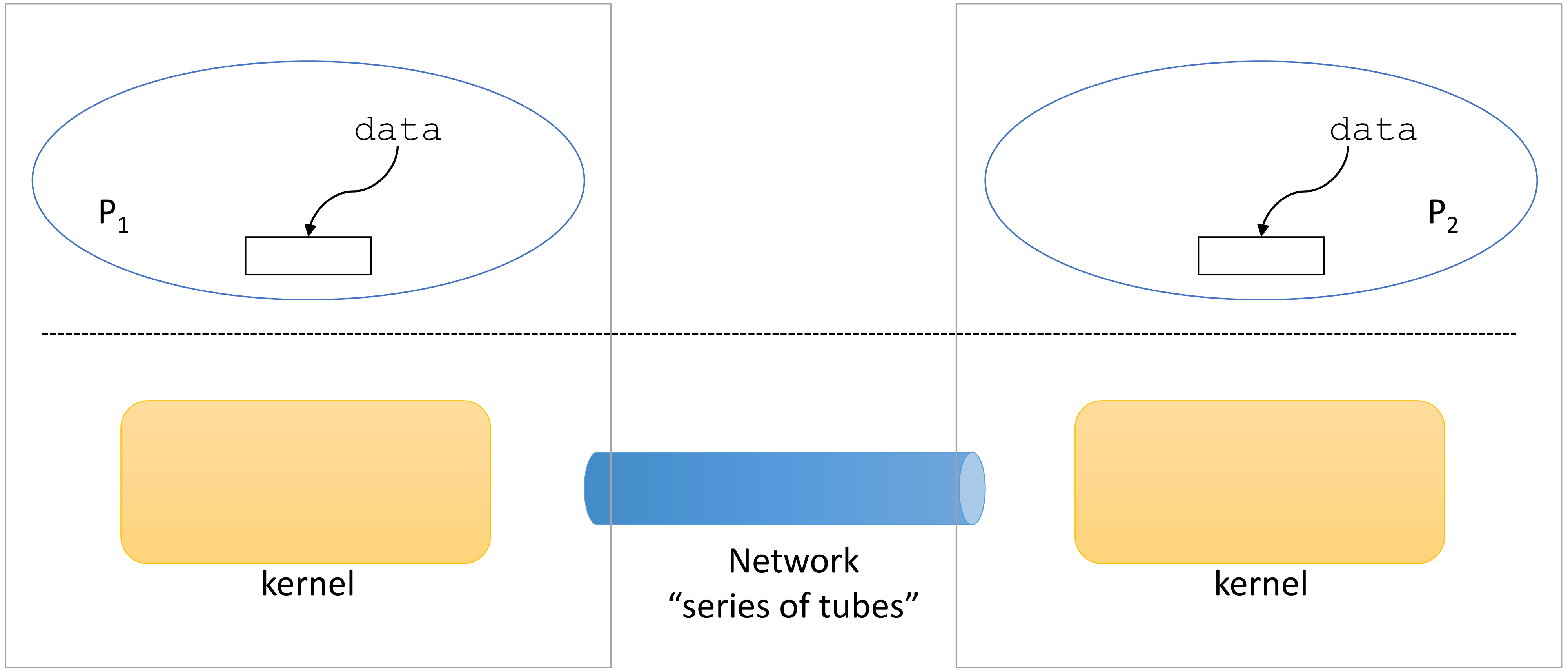
https://en.wikipedia.org/wiki/Series_of_tubes

Buffering



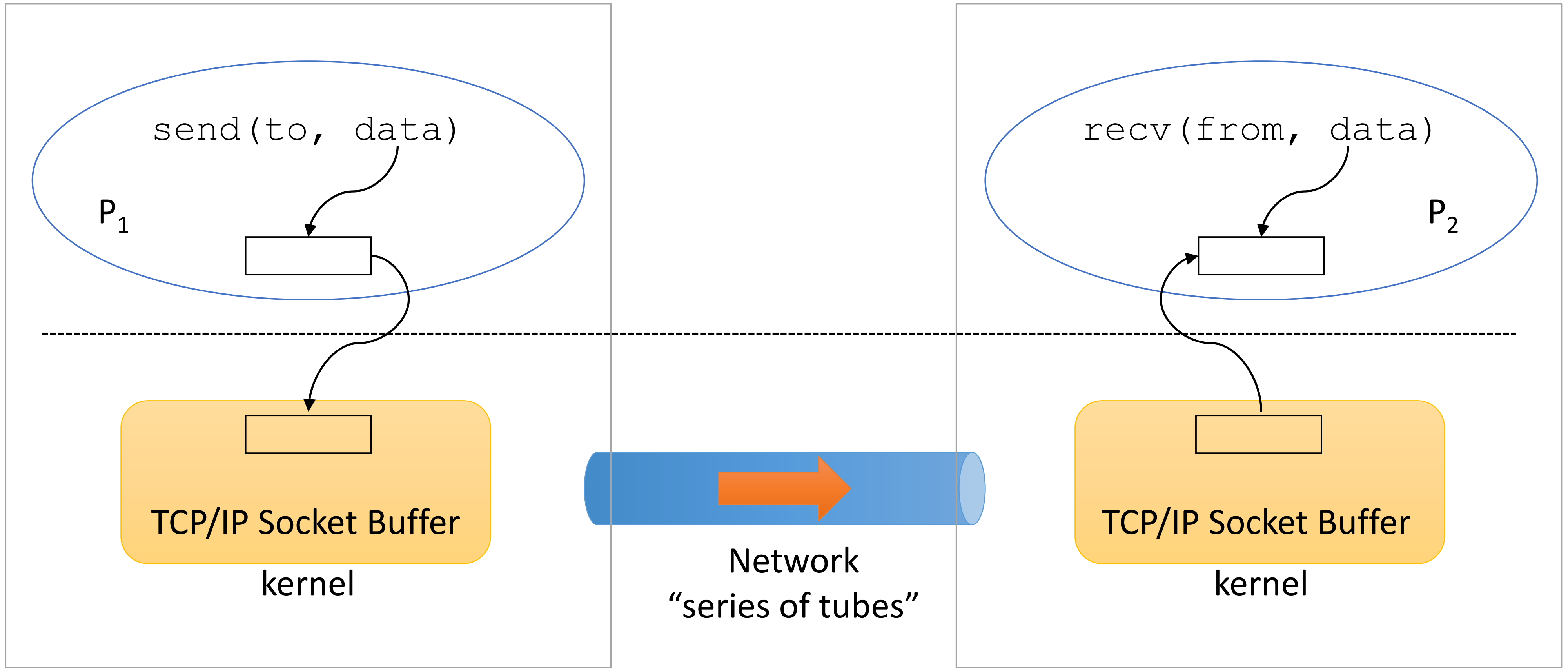
Buffering

P1 wants to send data to P2.



Buffering

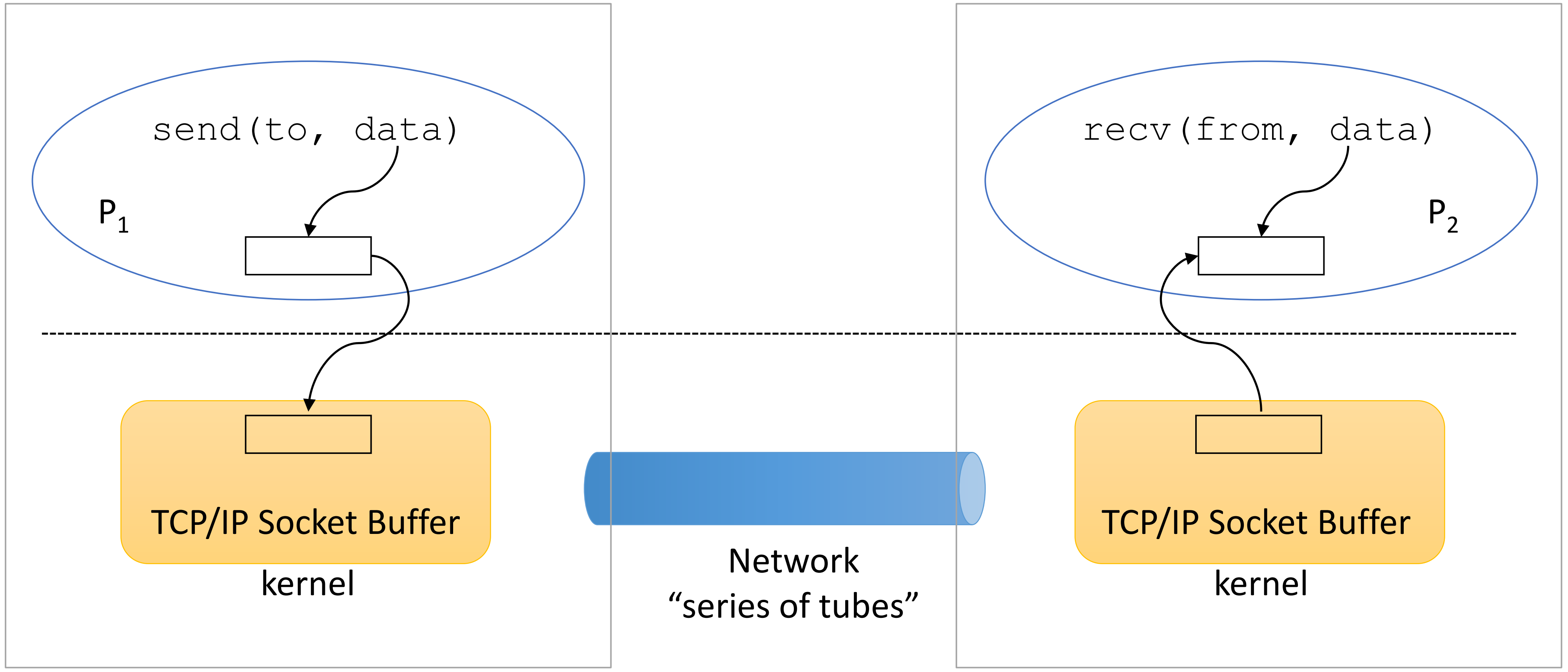
All send() and recv() do is copy data to/from the OS.



OS will format data and send it using the network device when it can.

Buffering

Kernel's buffers have finite storage space!



If sender fills buffer, OS will mark the process as blocked – can't be scheduled until space is free.

If the buffer is empty, OS will mark the receiver process as blocked – can't read data before it arrives!

This behavior is (commonly) the synchronization mechanism for message passing!

Note: Deadlock is still easily possible:

P_1 – recv()
P_2 – recv()

If sender fills buffer, OS will mark the process as blocked – can't be scheduled until space is free.

If the buffer is empty, OS will mark the receiver process as blocked – can't read data before it arrives!

Distributed Shared Memory (DSM)

- Multiple machines conspire to make it look like they have a giant pool of shared memory.
- OS intercepts memory accesses, sends to other nodes (if needed)
- Not much commercial success.

Summary

- Communication requires data transfer and synchronization.
- Common models for structuring the communication.
  - (e.g., producer/consumer, client/server, and others)
- Many abstractions for performing communication
  - Broad classifications: shared memory vs. message passing