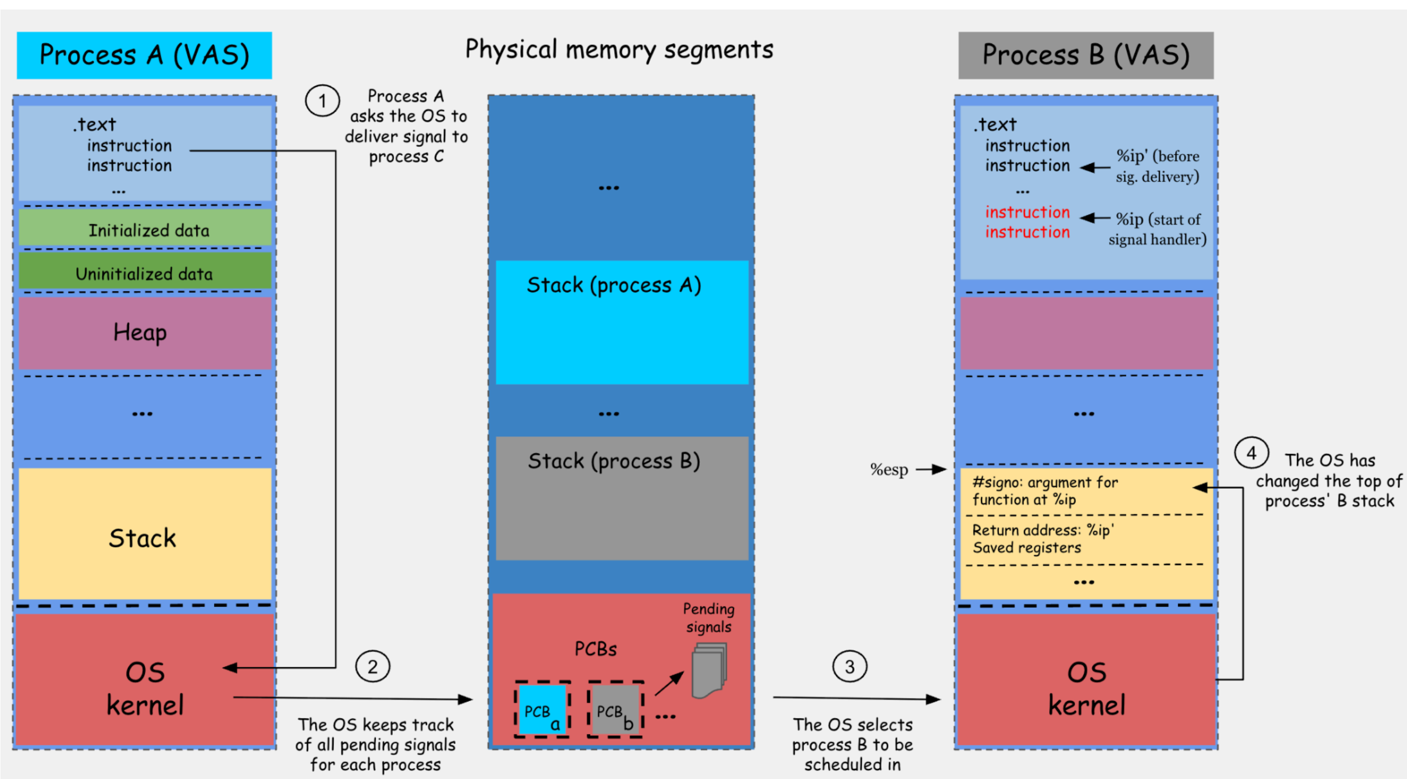
- **Q1: How do processes communicate with each other?**
  - We've invested a lot so far in the fact of life that processes are a domain of isolation.
  - How do we support communication between processes, if needed? That is, how do we cross the domain of isolation from one process to another?
  - Cooperation between processes requires communication, which is called Interprocess-Communication (**IPC**), and is mediated by the OS.
  - There are two core models for IPC: **Synchronous** (e.g., pipes) and **asynchronous** (e.g., signals) communications.

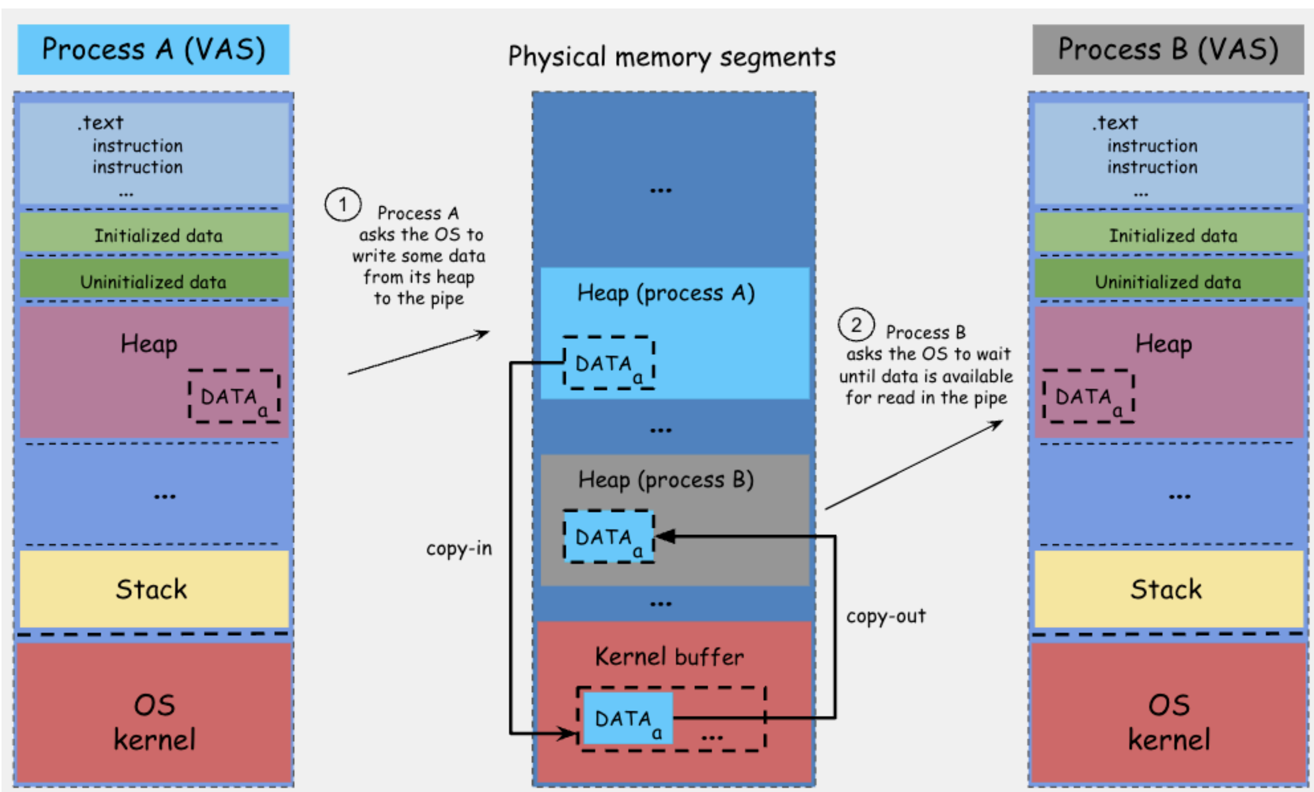
- **Q2: What are the most common asynchronous POSIX IPC mechanisms?**
  - Asynchronous communication means that the recipient (process) of a message is not necessarily waiting for a message to be delivered to it.
  - The OS needs to support an asynchronous IPC mechanism that will allow the delivery of messages in a manner that will not irreversibly corrupt the flow of executions of an "unprepared" recipient.
  - **POSIX signals** is the most common mechanism for asynchronous IPC.
    - The syscall **kill**(pid_t pid, int signo) is used to ask the kernel to deliver a signal from one process to another
    - A "signal" is a short message, identified by its signal number—typically, just a small integer.
    - There is a set of 32 predefined POSIX signals (e.g., 9: SIGKILL; 11: SIGSEGV; and others).

## Overview of delivering a POSIX signal



- **<u>Q3</u>: What are the most common synchronous POSIX IPC mechanisms?**
    - Synchronous communication means that the recipient (process) is waiting (blocked) for a message to be delivered to it.
    - Contrary to asynchronous IPC, the target recipient is, in a sense, to be assumed "prepared" for a message to be delivered to it.
    - **POSIX pipes (unnamed):**
        - Syscall to create a synchronous, unidirectional communication channel between two processes: pipe (int fds[2]).
        - Returns two file descriptors in the fds buffer.
            - fds[0]: The read end of the pipe.
            - fds[1]: The write end of the pipe.
        - Operations on pipes: read/ write/ close (akin to files, discussed later).
            - Read for data on fds[0] will block (wait) until data is written to fds[1], and will return 0 if no more data is available and fds[1] is closed.
            - Write on fds[1] will lead to a SIGPIPE signal delivered to the process trying to write on a pipe whose read end (fds[0]) is closed.
        - A problem with unnamed pipes is that the channel can only be established between descendant processes: the parent creates the fds[] buffer and invokes fork; the child "inherits" copied fds in its PCB. [Q: System-wide visible pipes? mkfifo; Q: Bidirectional channel? sockets.]

**<u>Overview of delivering a message via unnamed POSIX pipes</u>**



- **POSIX shared memory**
    - Pipe-based communication is mediated by the OS, and user programs have little responsibility, except for waiting for a message to be delivered.
    - But it is unnecessarily slow: two copies (in and out) are required on every transaction.
    - Processes can avoid this overhead, and ask the kernel to point them directly to a memory region that is shared with other processes.
    - syscall to create a shared memory segment: uid = shmget(key,…).
    - syscall to attach a shared memory segment to the address space of the calling process: shmat(int uid,...).
    - Zero unnecessary copies
        - New powers ⇒ New responsibilities
        - Synchronization ⇒ More on this drama later

<u>**Overview of delivering a message via POSIX shared memory**</u>