# Software Infrastructure
# Open Question 1st Unit

**Luís Cruz**[1]

[1]Computer Science – CESAR School
Avenue, Cais do Apolo, 77, Recife - PE, 50030-22

lhcc@cesar.school

***Abstract.*** *This paper will describe the different Interprocess Communication (IPC) mechanisms found in the three major modern operating systems: Windows, MacOS and Linux. For each of them, three mechanisms will be presented and detailed. Finally, they will be compared, both within the same operating system and to other mechanisms from other operating systems.*
***Keywords —*** *Interprocess Communication, Operating Systems*

## 1. Introduction

Interprocess communication is what we call the concept of two or more processes sharing data or sending messages to one another. It can happen via a number of different mechanisms that are implemented over different models of communication, such as *shared memory* and *message passing*. (SILBERSCHATZ, 2018)

## 2. Linux

Linux is a Unix-based operating system, and it prefers the word *task* when referring to processes and threads, but to even out the information presented in this paper, processes will be referred to as *processes* normally.

### 2.1. Unnamed Pipes

Pipes create a unidirectional channel between two processes, a *write* end and a *read* end. When pipe() is called, the system calls two child processes, *write()* and *read()*, linking both the write end process' file descriptor and read end process' file descriptor to a Virtual File System (VFS) memory buffer as if they were writing and reading from the same file in disk, without actually having a mount point. The buffer uses a FIFO (first in, first out) structure, so if the read child process tries to read bytes from the pipe without having available data, the kernel blocks it until there are bytes to be read. (UNIX. . . , 2020) and (A. . . , 2019)

### 2.2. Named Pipes

Commonly known as FIFOs, named pipes are similar to unnamed pipes in implementation. The difference is that a FIFO, created with the system call *mkfifo()* creates a special file and mounts it in the file system. Different from ordinary files, this special file must be necessarily opened on two ends, *write* and *read*. On the *read* end it is opened with the O_RDONLY flag, which the kernel uses to block other operations other than reading, and

the *write* end uses the O_WRONLY flag, which the kernel uses to refuse operations other than writing. The kernel also blocks simultaneous reading and writing operations since these must be synchronized to avoid data corruption. (MKFIFO3, 2021) and (PIPE7, 2021)

### 2.3. Signals

Linux signals are a message passing mechanism created to send direct messages to processes usually when a program needs to be terminated, killed or halted, with system calls such as *kill(), killpg(), raise()*. Each signal has an integer identification, a name and a default action. There are two stages in a signal transmission lifespan: signal generation and *signal delivery*. During *signal generation* the kernel updates a data structure in the destination process' memory space to indicate that a signal has been sent, usually containing the signal's name, integer identification and default action. Next comes *signal delivery*, in which the kernel forces the destination process to react by interfering with the process' execution. The process can choose to ignore the signal, except for specific cases such as SIGKILL, which must be handled immediately. (BOVET, 2005)

## 3. Windows

Windows strongly relies on handles, they are a data structure representing the system resource in use by the process and contains addresses and means to identify the resource. (HANDLES..., 2022)

### 3.1. Named Pipes

Windows also has its own implementation of named pipes. A named pipe must be created by a server process, which is the process that will write to the pipe. When a named pipe is created, it receives an identification name and it's given a certain buffer size, then returns a handle to which the server process connects. The server process must now wait for the client process to call and connect to the pipe, which also returns a handle. Both processes must be aware that the communication will take place. When the named pipe is created, the kernel creates the buffer in a nonpaged pool, which is a reserved space in primary memory for the Windows kernel. The availability of nonpaged pools also limits the instances of pipe buffers that can be created by processes. When a write operation occurs, the operating system first tries to compare the memory availability to the process' given write quota, if the remaining quota is enough to fulfill the request, then it's completed immediately, or else the system tries to expand the buffer to accommodate the remaining quota, and if that's not possible, then the write process is blocked until enough data is read by the client process in order to release the additional buffer quota. When both processes close their handles, the buffer is cleared and memory is returned to the kernel. (PIPE..., 2021) and (CREATENAMEDPIPEA..., 2022)

### 3.2. Mailslots

Mailslots provide mostly one-way communication, usually from one process to many processes. A mailslot server is a process that creates a mailslot, and a mailslot client is a process that writes to the server's mailslot. The mailslot will save the messages until the server has read them, but the messages can only be up to 400 bytes in size. The mailslot content is saved on a pseudofile that resides in primary memory, and standard

file procedures are used to access it, and it is local to the process that created it. New messages are always appended to the mailslot. Just like pipes, the server process creating a mailslot receives a handle for the mailslot. The writing process inputs data into the mailslot like it's a regular file. The process does not know the pseudofile is not a regular file, only a space in primary memory, so it interprets writing as if it's to a regular file. The reading (server) process also does not know the pseudofile is not a regular file, so it reads until there are no messages left and then closes the handle. When both processes close their handles, the operating system knows it's supposed to delete the mailslot, and the communication is finished. (USING. . . , 2021) and (CREATING. . . , 2021)

### 3.3. File Mapping

File mapping has the processes treat the contents of a file as a memory space in their own addressed space. Both reading and writing processes receive a pointer to the file in their own address space that they use to perform operations. Given that a regular file is not created or managed by the kernel, the system creates a file mapping object to be able to control I/O operations and use synchronization methods such as a semaphore, to prevent data corruption from unsynchronized reading and writing, and the file mapping object is actually where the pointers in the process' memory is pointing to.All operations performed are actually cached to the file mapping object to improve performance, and the operating system is responsible for committing changes made on the mapping object to the file on disk. (FILE. . . , 2021)

## 4. Mac OS

The Mach kernel is a microkernel present within Mac OS mostly responsible for interprocess communication. That means the Mach kernel is not enough on its own to suffice an entire operating system, but does a good job when paired to the Mac OS kernel. (DUTT, 2021)

### 4.1. Mach IPC

A Mach Port is the Mach's abstraction to system resources such as threads, memory objects and processes. A Mach Port is protected and managed by the kernel. The most basic of operations are for sending and receiving messages on a queue of limited length in kernel space. To access a port, a process must have the *port right*, which is managed by the kernel. Mach IPC is integrated with the system's virtual memory subsystem, and when a process dequeues a message, it is copied into its memory address space, which means the message could be as large as a process' memory address space. If the queue is full or the receiving process cannot hold any more messages into its address space, both sending and receiving processes are blocked by the kernel until the receiving process is able to read more messages and free the queue space. (LEVIN, 2013) and (SINGH, 2006)

### 4.2. Signals

Mac OS also provides Unix-style signals, built on top of the Mach exception handling. Just like Linux, Mac OS signal implementation also has two fundamental phases: *signal generation* and *signal delivery*. Signal generation happens when there is an event that warrants a signal, and delivery happens when the signal is carried out to the destination process. A signal is mostly used to kill or terminate a process. (LEVIN, 2013) and (SINGH, 2006)

### 4.3. Named Pipes

Pipes, or FIFOs, as presented by many other Unix based operating systems, are also present in Mac OS. If a FIFO is opened for reading, the system will block *open()* if there are no writers. The same works the other way, meaning that there must be a reading and a writing process linked to the same FIFO. The Mac OS FIFOs also use a special type of file with physical presence in the file system. The processes connect to the file via a Unix domain local stream socket. (LEVIN, 2013) and (SINGH, 2006)

## 5. Discussion

In order to compare the mechanisms before presented, I must emphasize that the similarities between pipes and signals on Linux and MacOS operating systems are too close, if not exactly the same, because of that they will be treated as the same. First, let's compare pipes from the Unix-based systems (Linux and MacOS) to Windows systems. The way the Linux and MacOS kernels connect the two processes to the pipe is via a file descriptor, which "tricks" the processes into interpreting the pipe as a regular file, with regular read and write capabilities, except in this case the pipe is a kernel memory buffer in the virtual file system and not mounted in disk. The Windows implementation, on the other hand, does not try to hide the fact that the buffer is not a real file. Instead it gives each process a handle, implemented specifically to represent a pipe's data structure and resource capabilities, so instead of pretending to open a real file, windows is straightforward to its process and whatever processes want to use pipes must have a specified implementation of it. The closest a Windows system gets to a Unix pipe is the Mailslot mechanism. It creates a virtual file in primary memory with no mount point, and it's used just like a pipe for unidirectional communication. The primary difference is where the virtual file resides, which is not in kernel memory space, instead it's allocated in user memory space, and the messages queued can only have up to 400 bytes in size. When both read and write handles are closed, the mailslot is closed too. The most unique mechanism I found during the production of this paper was File Mapping in Windows. No other operating system presents such a concept: an object for mapping parts of a file and sharing the mapped object between two processes. The closest implementation we can find is to have two processes opening the same file in disk, but this is inefficient and unsafe, due to disk I/O time and synchronization issues. File mapping, on the other hand, by selecting a small section of the file to use in the mapping object, cuts down the time cost to perform operations to disk, because it only writes changes made to the object when the handles are closed. It is also safer on a synchronization standpoint, because the mapping object applies synchronization mechanisms on its implementation.

## 6. Conclusion

The production of this paper presented me many challenges. First and foremost, there was time. This type of work requires a lot of time, and even though I believe it is entirely possible to do a great job in the given time, I also believe that the person doing a great job must be completely focused on this, or at least have no other obligations in their academic or professional life. Next comes the challenge of finding information, especially about MacOS. Windows manages to simplify this situation by explaining kernel procedures within their own guides, but MacOS is a whole other problem, since the only source of information is a few books, which I mostly believe to be guessing and speculation. If you

check the explanation for unnamed pipes in (Mac OS X Internals A Systems Approach), it is mostly similar if not sometimes exactly the same as the open explanation for Linux unnamed pipes. I know they're both Unix-based so this might technically be right, but just comes to show how difficult it is to come by actual original knowledge regarding Apple software. Last comes the challenge of just writing a paper, for someone who never did it before, comes a learning curve. Not just speaking for myself but many of my colleagues have been encountering the same challenge, that we are now being presented with the opportunity to overcome.

## References


A guide to inter-process communication in Linux. 2019. https://opensource.com/sites/default/files/gated-content/inter-process_communication_in_linux.pdf. Accessed: 2022-08-19.

BOVET, D. P. **Understanding the Linux Kernel**. 3rd. ed. [S.l.]: O'Reilly, 2005.

CREATENAMEDPIPEA fucntion. 2022. https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-createnamedpipea. Accessed: 2022-08-23.

CREATING a Mailslot. 2021. https://docs.microsoft.com/en-us/windows/win32/ipc/creating-a-mailslot. Accessed: 2022-08-22.

DUTT, H. **Interprocess Communication with MacOS: Apple IPC Methods**. 1st. ed. [S.l.]: Apress, 2021.

FILE Mapping. 2021. https://docs.microsoft.com/en-us/windows/win32/memory/file-mapping. Accessed: 2022-08-22.

HANDLES and Objects. 2022. https://docs.microsoft.com/en-us/windows/win32/sysinfo/handles-and-objects. Accessed: 2022-08-21.

LEVIN, J. **Mac OS X and iOS Internals**. 1st. ed. [S.l.]: John Wiley  Sons, Inc., 2013.

MKFIFO3. 2021. https://man7.org/linux/man-pages/man3/mkfifo.3.html. Accessed: 2022-08-19.

PIPE Functions. 2021. https://docs.microsoft.com/en-us/windows/win32/ipc/pipe-functions. Accessed: 2022-08-23.

PIPE7. 2021. https://man7.org/linux/man-pages/man7/pipe.7.html. Accessed: 2022-08-19.

SILBERSCHATZ, A. **Operating System Concepts**. 10th. ed. [S.l.]: Laurie Rosatone, 2018.

SINGH, A. **Mac OS X Internals: A Systems Approach**. 1st. ed. [S.l.]: Addison Wesley Professional, 2006.

UNIX pipe implementation. 2020. https://toroid.org/unix-pipe-implementation. Accessed: 2022-08-19.

USING a Mailslot for IPC. 2021. https://docs.microsoft.com/en-us/windows/win32/ipc/interprocess-communications#using-a-mailslot-for-ipc. Accessed: 2022-08-22.