DSC204A: Scaleable Data Systems Spring 2025

11: IPC

Instructor: Umesh Bellur *Scribe: Mizuho Fukuda, Ashley Ho*

1 The Problem

How can programs talk to each other when they are in execution? In other words, the main goal is to address how programs can communicate and coordinate with each other while they are running. Each process normally has a private address space protected by the operating system, which creates challenges when important data needs to be shared between processes.

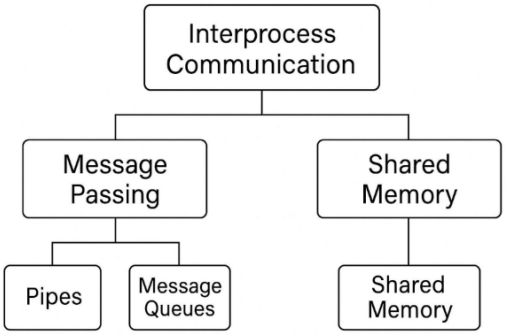
2 IPC

Inter-process communication (IPC) mechanisms are used to address this problem, including shared memory and message passing, and synchronization is necessary to ensure correct communication and avoid conflicts.

An effective IPC will ideally be:

- Fast
- Easy to use
- Well defined synchronization model
- Versatile
- Easy to implement
- Works remotely

IPC Options



There are two main types of IPC: shared memory and message passing. For shared memory, multiple processes can take advantage of a shared physical memory space. For message passing, one process encapsulates messages in a medium that can be received by other processes. Pipes and message queues are examples of the many ways message passing is performed on operating systems.

2.1 Sychronization

Both memory sharing and message passing require **synchronization** to ensure that multiple processes can safely share and access data without interfering with each other. Synchronization is important because processes often operate independently and at different speeds, which can lead to conflicts if they try to read or write shared information at the same time. By carefully managing when and how processes interact with shared resources, synchronization helps maintain correctness and stability during execution.

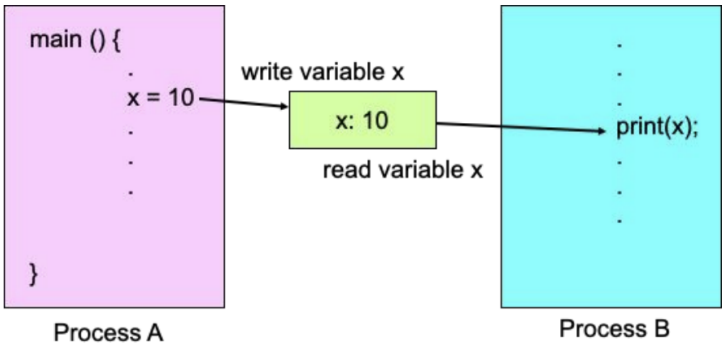
The goals of synchronization are as follows:

- Allowing a sender to indicate when data is transmitted.
- Allowing receiver to know when data is ready
- Allowing both to know when more IPC is possible

For example, when data moves from producer → message queue → consumer, the producer needs a way to indicate to the consumer to "go read the queue." In general, the processes needs to come to a common understanding about the current state.

3 Shared Memory IPC

Shared Memory IPC allows different processes to communicate by sharing a common piece of memory. This shared memory can be mapped either physically or virtually, depending on the system's architecture. Processes can read from and write to this memory space as a means of communication. However, since multiple processes may access the memory simultaneously, synchronization is crucial to prevent conflicts. To manage this, semaphores or locks are often employed within or alongside the shared memory, ensuring that only one process can modify the data at a time.



3.1 Reading and Writing from Shared Memory - Without Synchronization

The process of reading and writing from shared memory without synchronization mechanisms in place introduces risks, as multiple processes may access and attempt to modify the shared memory concurrently.

- Write
  1. Process A creates shared memory segment: "set aside a shared memory with this key and size."
  2. Attach Process A to the shared memory.
  3. Process A writes data to the shared memory.
  4. Allow time for Process B to read.

→ Problem: Process A has no way of knowing when Process B starts or finishes reading!

- Read

  1. Process A attaches to the shared memory segment.
  2. Process A accesses the shared memory.
  3. Map the shared memory segment into Process A's memory space.
  4. Process A tries to read from memory.
  5. If nothing there (i.e. Process B has not written yet), either:
     - Return blank.
     - Wait for a set amount of time or until something is there.

→ Problem: Process A has no way of knowing how long it needs to wait for process B to finish writing!

3.2 Reading and Writing from Shared Memory - With Synchronization

When reading and writing from shared memory with synchronization, mechanisms such as a **mutex** (lock variable) are utilized to ensure that only one process can access the shared memory at a time. The mutex ensures mutual exclusion by combining the reading and writing processes into one controlled sequence. More specifically, if one process holds the mutex, other processes cannot acquire it until the first process releases it. This prevents concurrent writing conflicts by controlling access to the shared resource.

The reading and writing process with a mutex is as follow:

1. Create shared memory segment.
2. **Create a mutex (lock)** for synchronization. This indicates when writing to the shared memory segment is available.
   - The mutex is protected by hardware. The hardware exports a set of instructions: test and set, which is atomic, meaning that the process of reading from the lock variable and flipping the switch occurs in one go.
3. Create two processes.
4. Start two processes.
5. Wait for both processes to finish.

But what if instead of a mutex, a boolean variable Y is used to indicate whether write access is available to the shared memory X? Suppose Y = True means that the memory is available for writing. When process A wants to write to X:

1. A checks if Y is True.

If Y = True:

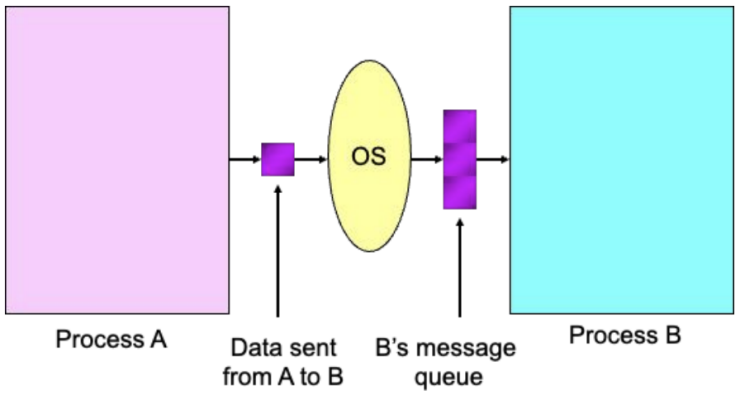
2. A sets Y to False to indicate that it is now using the memory
3. A writes to X

However, this approach has a serious problem: if process B also checks Y right after A checks it, but before A has the chance to set Y to False (between step 1 and 2), both processes may believe the memory is available and proceed to write at the same time. This happens because the checking and updating of Y are two separate actions, leading to incorrect behavior.

In conclusion, when process A wants to write to the shared memory, it checks the mutex, which is a special variable protected by hardware, to make sure that process B is not writing to it at the same time. Because the mutex is updated atomically, the lock state is always up to date when a process accesses it, ensuring that only one process can safely use the shared memory at a time.

4 Message Passing IPC

Message Passing IPC enables communication between processes by sending and receiving messages, rather than sharing memory directly. In this approach, data is passed from one process to another through message queues or pipes. Each process has its own private memory, and the operating system manages the exchange of information. This form of communication ensures that processes can interact without the need for shared memory but still requires careful handling to ensure proper synchronization and avoid simultaneous read-write conflicts.



In general, message passing IPC mechanisms work as follows:

1. The sender formats the data into a formal message, with an address for the receiver.

2. The operating system places this message into the receiver's message queue, which is a temporary storage area for messages waiting to be read by the receiver. The OS may also signal the receiver that a message is ready in the queue.

3. When the receiver is ready, it reads the message from the queue.

4. The sender may or may not block, depending on the implementation. In other words, the sender might wait (block) for confirmation that the receiver has processed the message, or it may continue its execution immediately without waiting.

4.1 Unix Pipes

Unix pipes are specific to Unix-based systems like Linux and macOS, and they do not work natively on Windows. In Unix, everything is treated as a file, including IPC mechanisms such as pipes. Each pipe has two file descriptors: one for reading and one for writing.

1. Create a pipe. This returns two file descriptors: one for reading and one for writing.
2. Fork the process:
   - Child only writes (close the read end)
   - Parent only reads (close the write end)
   - Child process writes some message to the file → parent process reads from the file after some time

This allows two processes to communicate by streaming data from one process into the pipe and reading it out from the other end, enabling simple and efficient inter-process communication.

4.2 Synchronous vs. Asynchronous Communication

For message passing IPCs, the synchronization can be handled one of two ways: synchronous (blocking) and asynchronous (non-blocking). In a blocking operation, a process will halt its execution and wait until the operation is fulfilled. For example, if a process tries to read and no data is available, it will block until data becomes available. Similarly, in a blocking write, the process will continue to block until there is space to write the data.

In contrast, non-blocking operations allow the process to continue executing without waiting for the operation to complete. If a non-blocking read finds no data, or a non-blocking write finds no space, the operation will return an error, allowing the process to handle the situation without blocking further execution.