Iter-Process Communication

UNIX IPC

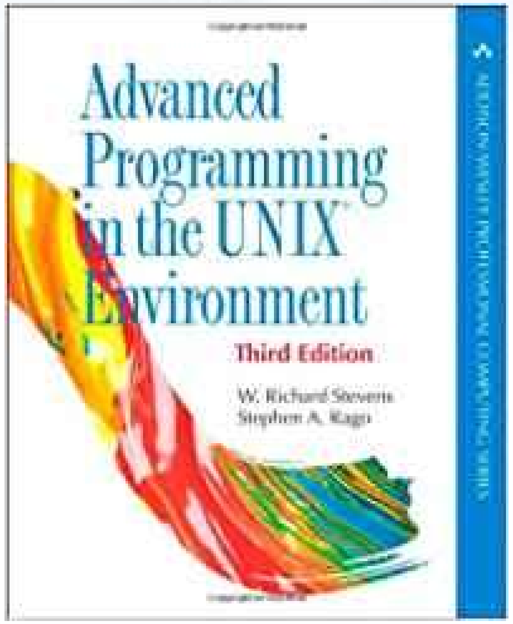
Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

Premises

- Where are we?

- u01-courseIntroduction
- u02-review
- u03-cppBasics
- u04-cppLibrary
- u05-multithreading
- u06-synchronization
- u07-advancedIO
- u08-IPC



- u08s03e
- u08s05e
- u08s01-introduction.pdf
- u08s02-FIFOs.pdf
- u08s03-messageQueues.pdf
- u08s04-sharedMemory.pdf
- u08s05-shockets.pdf

Introduction

Processes do not share their address space: Each process in an operating system has its own separate memory space. This isolation ensures that one process cannot directly access the memory of another process, which enhances security and stability.
Copy-on-Write (CoW): This is a memory management technique used to efficiently handle the duplication of memory. When a process is forked, the parent and child processes initially share the same memory pages. Only when one of the processes modifies a shared page is the page copied, ensuring that each process has its own version of the page. This reduces the overhead of duplicating memory unnecessarily.

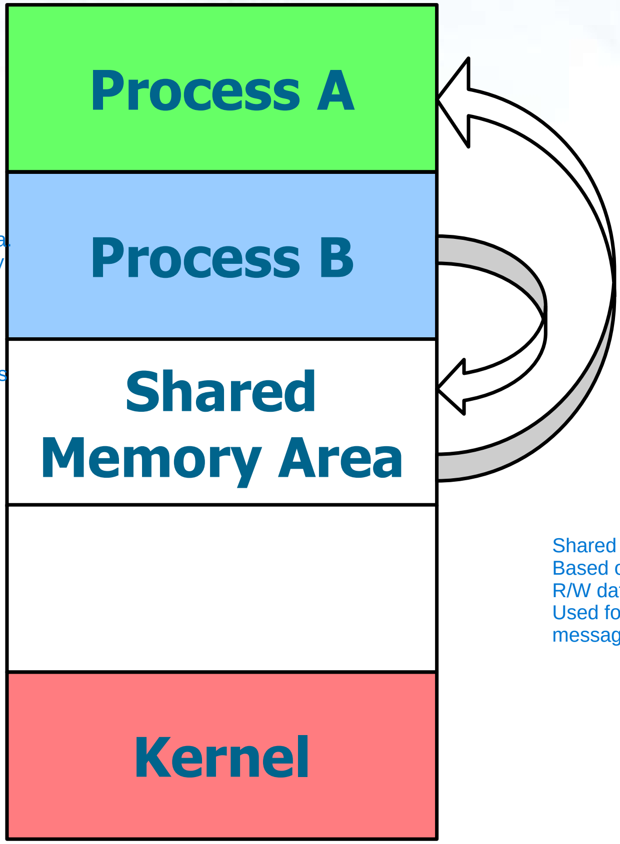
- Processes do not share their address space
  - Copy-on-Write duplicates the memory only when strictly required
- Information sharing among processes is referred to as **IPC** or **I**nter **P**rocess **C**ommunication
  - Processes may need to share information when running
    - On the same machine, **intra**-machine communication
    - On different machines, **inter**-machine communication
  - The communication models can be based on
    - Message exchange
    - Shared memory

Information sharing among processes: Since processes do not share their address space, they need mechanisms to communicate and share data. This is where IPC comes in.
IPC (Inter Process Communication): This refers to the methods and mechanisms that allow processes to exchange data and signals.

Message exchange: Processes communicate by sending and receiving messages. This can be done using various IPC mechanisms like pipes, message queues, sockets, etc.
Shared memory: A portion of memory is shared between processes. This allows processes to read and write to the same memory space, facilitating faster communication compared to message passing.

Communication models

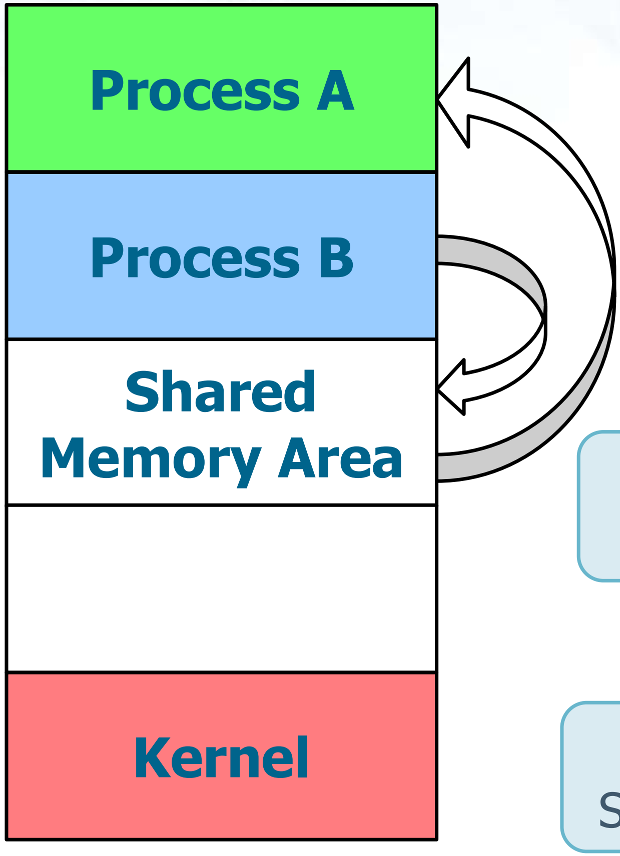
The diagram illustrates two processes, Process A and Process B, both accessing a shared memory area. This shared memory area is managed by the kernel, which ensures that both processes can read from and write to this common space.



- Shared memory
  - Based on sharing a memory area and R/W data in this area
  - Used for sharing a large amount of data

Shared Memory:
Based on sharing a memory area: This model allows multiple processes to access a common memory space.
R/W data in this area: Processes can read from and write to this shared memory area.
Used for sharing a large amount of data: Shared memory is efficient for transferring large volumes of data between processes because it avoids the overhead of message passing.

Communication models



- Shared memory
  - Most common methods
    - **File** sharing
      - Sharing the name or the file pointer or descriptor before fork/exec

Course of OS

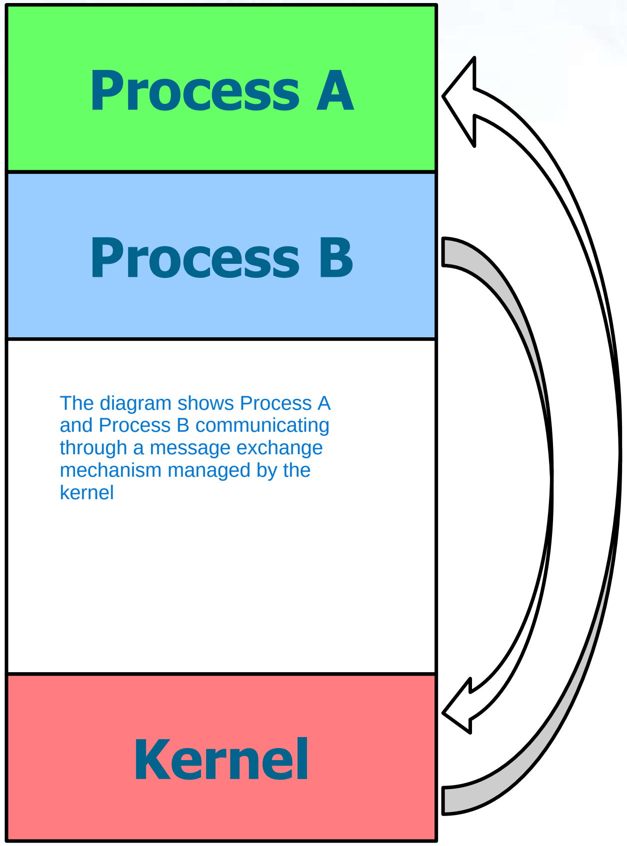
    - **File mapping**
      - A file is mapped into the address process space

Unit 07 Section 05

    - **Memory sharing**
      - A chunk of memory is mapped into the address process space

Unit 08 Section 04

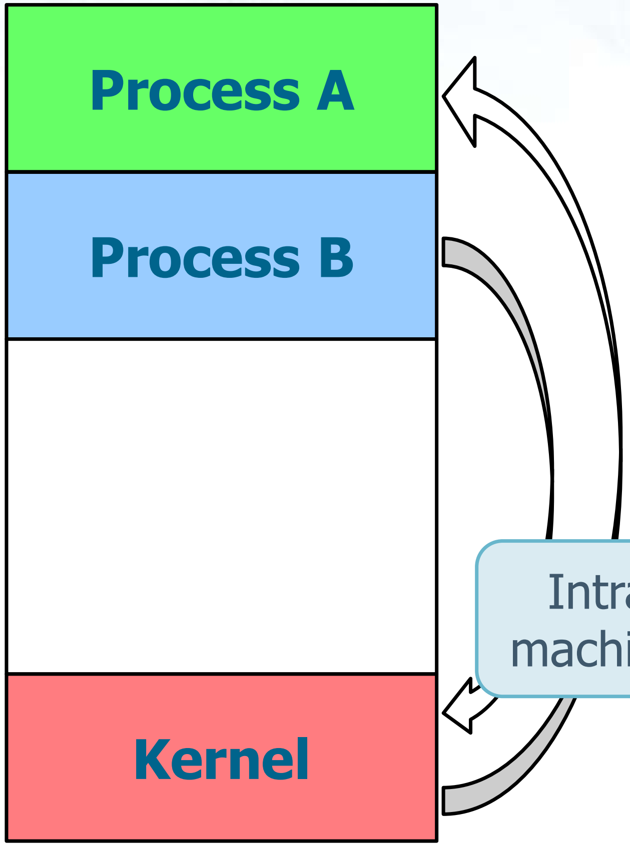
Communication models



- **Message exchange**
  - Communication takes place through the exchange of messages
  - Need to setup of a communication channel
  - Useful for exchanging limited amounts of data
  - Uses system calls
    - Require kernel intervention
    - Introduce overhead

Message Exchange:
Communication takes place through the exchange of messages: Processes communicate by sending and receiving messages.
Need to set up a communication channel: A communication channel must be established between the processes to facilitate message exchange.
Useful for exchanging limited amounts of data: This method is typically used for smaller data transfers.
Uses system calls: Message exchange relies on system calls, which:
Require kernel intervention: The kernel is involved in managing the communication.
Introduce overhead: System calls can add some performance overhead due to the context switching between user mode and kernel mode.

Communication models



- Message exchange
  - Most common methods
    - Pipes
      - Basic (older) strategy
    - FIFOs
      - Avoid a common ancestor between processes
    - Message queues
      - Allow structured data
    - Network sockets
      - Allow transfer (through the network) between processes running on different machines

Unit 08 Section 02

Unit 08 Section 03

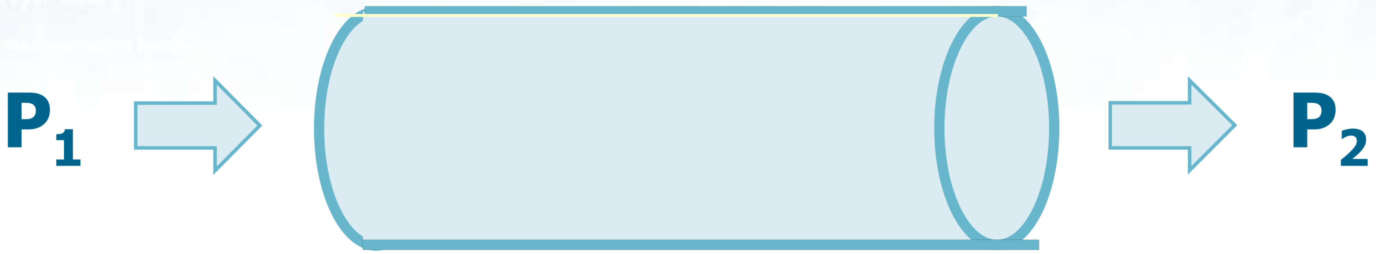
Unit 08 Section 05

Intra machine

Inter machine

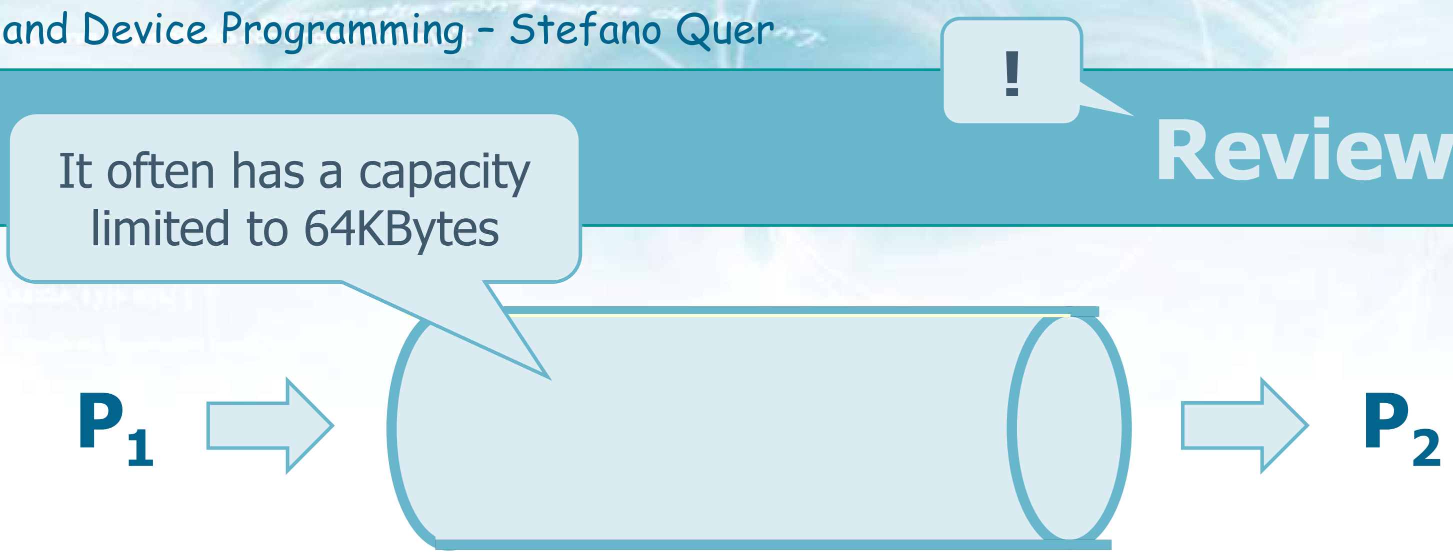
Review: Pipes

Original pipes (or unnamed pipes):
Common form of UNIX System IPC:
Pipes are widely used in UNIX systems for inter-process communication.
Pseudo-files linking two processes:
Pipes act as temporary files that link two processes, allowing data to flow between them.
Can be used only between processes that have a common ancestor:
The pipe must be created before forking the parent process: The parent process creates the pipe, and then forks child processes that inherit the pipe.
The pipe lasts only as long as the processes last: The lifespan of the pipe is tied to the lifespan of the processes using it



- Original pipes (or **unnamed** pipes)
  - Are a common form of UNIX System IPC
  - Are pseudo-files linking two processes
  - Can be used only between processes that have a **common ancestor**
    - The pipe must be created **before** forking the parent process
    - The pipe lasts only as long as the processes last

Review: Pipes



- Original pipes (or **unnamed** pipes)
  - Are **half duplex**
    - Data flows only in one direction
  - Can be seen as an **unstructured** sequential files
    - The communication channel is a FIFO queue
    - Structured data transfers require a communication protocol
  - Transfer only **limited quantity** of memory

! Review: Pipes

```
int file[2];
char cR cW;

if (pipe(file) != 0) { ... Error ... }
pid = fork ();
if (pid == -1) { ... Error ... }

if (pid == 0) {
  // Child
  close (file[1]);
  n = read (file[0], &cR, sizeof(char));
  ...
} else {
  // Parent
  close (file[0]);
  n = write (file[1], &cW, sizeof(char));
  ...
}
```

The direction of data flow depends on how the pipe is set up. Typically:
Parent writes, child reads: The parent process writes to the pipe, and the child process reads from it.
Child writes, parent reads: Alternatively, the child process can write to the pipe, and the parent process reads from it.
The key point is that data flows in one direction (half-duplex).

The child reads from the pipe

The parent writes into the pipe