

CHAPTER 3

# INTER-PROCESS COMMUNICATION (IPC)

gettyimages®

Daniel Grill



Independent vs. Cooperating Processes

Shared Memory Processors (SMP)

Message-Passing Processors (MPP)

Producer-Consumer Concept

## PROCESS OPERATIONS

OSs should be able to perform principal operations on processes such as:


- Create a process
- Destroy (kill) a process
- Suspend a process
- Resume a process
- Change the priority of a process
- Block a process
- Wake up a process
- Dispatch a process
- Enable a process to communicate with other processes: this is known as **Inter-Process Communication (IPC)**

## INDEPENDENT vs. COOPERATING PROCESSES

An **independent process** does not affect and is not affected by any other process executing in the system. In addition, it does not share data with any other process.

On the other hand, a **cooperating process** affects and is affected by other processes executing in the system. It also shares data with other processes.


Cooperating processes are needed for several reasons, such as:

- 
1. **Information Sharing**: several users may be interested in the same piece of information, such as a shared file.
  2. **Computation Speedup**: if we want a particular task to run faster, we must break it into independent subtasks; each of which will be executing in parallel with the others.
  3. **Modularity**: we may want to construct a system in a modular fashion, dividing the system into separate processes or threads (*to be discussed later*)
  4. **Convenience**: even a single user may work on many tasks at the same time. For example, you may be editing a file on Word, listening to music, and compile a Java program at the same time.

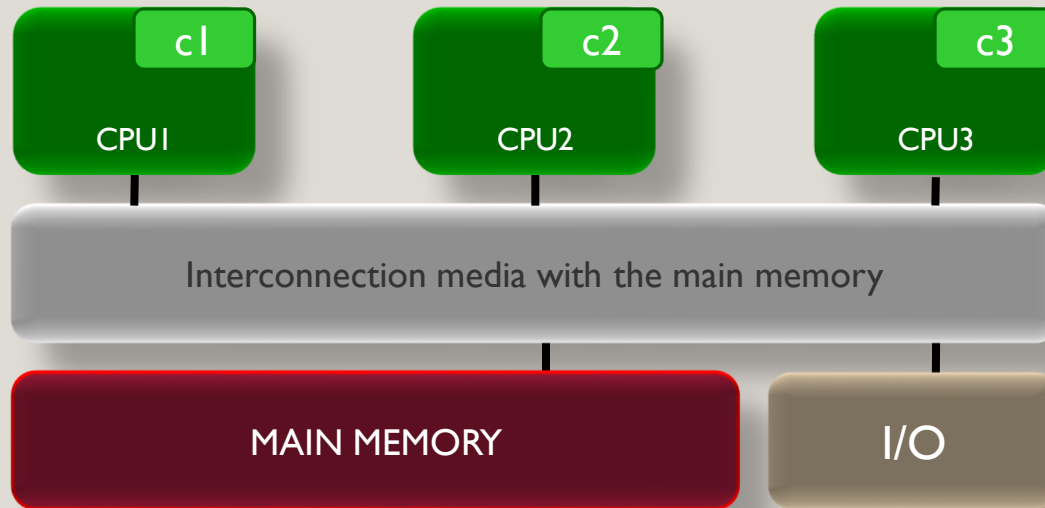
## INTER-PROCESS COMMUNICATION (I) – INTRODUCTION

Cooperating processes require an **inter-process communication** mechanism to allow them to exchange data and information.

There are two fundamental models of IPC:

- 
1. **Shared Memory**: A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
  2. **Message Passing**: Communication takes place by means of messages exchanged between the cooperating processes.

## INTER-PROCESS COMMUNICATION (2) – SHARED MEMORY (1)



When CPU1 needs to access a variable, it fetches it from the main memory.

Since the average access time to the main memory is relatively large, a cache is provided with each processor.

So a processor searches for the variable in its own cache. If not found, it is fetched from the main memory.

Processors don't search for the required variables in remote caches.

## INTER-PROCESS COMMUNICATION (3) – SHARED MEMORY – MEMORY ADDRESS SPACE

The shared-memory region resides in the address space of the process creating the **shared-memory segment**.

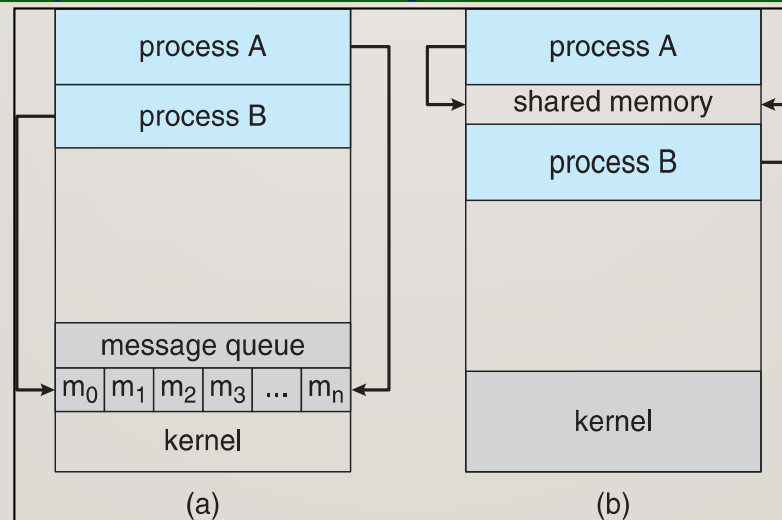
Recall that the OS prevents one process to access the address space of another process.

Shared memory environment requires that the communicating processes agree to remove this restriction.

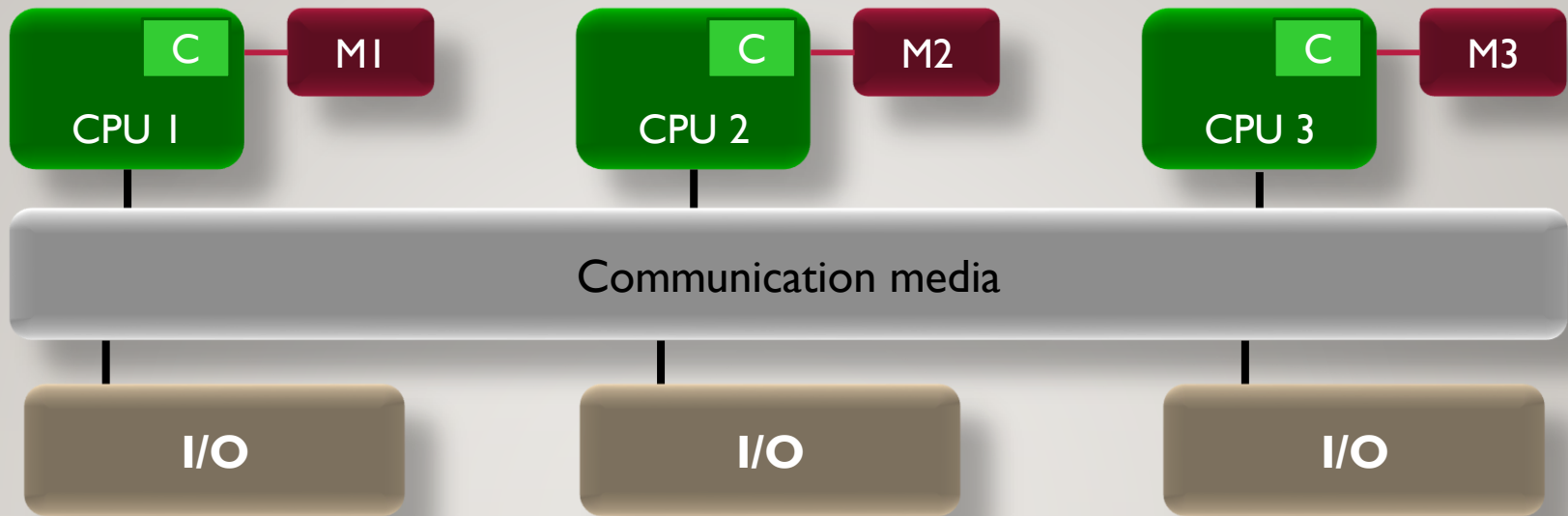
Each communicating process attaches the shared-memory segment to their own address space.

It is the responsibility of the processes, not the OS, to:

- ~ determine the form and the location of the data being shared
- ~ ensure that no two processes are writing to the same location simultaneously



## INTER-PROCESS COMMUNICATION (4) – MESSAGE PASSING



When CPU1 needs to access a variable, it searches for it in its local memory (M1).

If not found in M1, CPU1 broadcasts (sends to all connected CPUs) requiring for the value of the variable. The message is of the form:

`send (message)`

As CPU2 and CPU3 receive the message, they search for the variable in their local memories (M2 & M3 respectively). If found, they send the following message to CPU1:

`send (CPU1, message)`

Obviously, the access time to a local memory is less than that to a remote memory. The latter is less than the access time to the main memory.



## INTER-PROCESS COMMUNICATION (5) – SHARED MEMORY vs. MESSAGE PASSING

Shared Memory	Message Passing
Suitable for frequent data sharing.	Useful for exchanging smaller amounts of data.
Conflicts may occur because of data updates (cache coherence problem).	No conflicts need to be avoided.
Cache coherence problem makes it difficult to be implemented in distributed systems.	Easier to implement in distributed systems.
Faster than message passing; since all processes access the shared memory.	Slower because of the communicating media. In addition, each sent message should be acknowledged.
System calls are required once when establishing the shared memory region. Then no intervention is required from the kernel. All accesses are treated as routine memory accesses.	Another reason for being slower is that it is implemented using system calls, and thus requires the frequent intervention of the kernel.
Suffers from the cache coherence problem.	Provides better performance in distributed systems environment.

## PRODUCER-CONSUMER CONCEPT (I) – INTRODUCTION

The **producer-consumer** concept is applied by all cooperating processes.

A **producer process** outputs (produces) information that is to be used (consumed) by the **consumer process**.

### Examples:

→ A compiler (producer) produces an assembly code that is to be consumed by an assembler (consumer).

→ In a client-server paradigm, the server is the producer and the client is the consumer.

The KSU email server provides (produces) emails in your inbox. These are read (consumed) by the client which is the account owner (consumer).

A web server provides (produces) HTML files that are read (consumed) by the client; namely, the Internet user (consumer).

## PRODUCER-CONSUMER CONCEPT (2) – BUFFERS

Both producer and consumer should run concurrently (at the same time). In order to be able to do this, there should be a **buffer** that can be filled by the producer and emptied by the consumer.

There are two types of buffers:

→ The **unbounded buffer**: places no practical limits on the size of the buffer.

In such scenario:

The **producer** does not have to wait at all.

The **consumer** has to wait if the buffer is empty.

→ The **bounded buffer**: puts restrictions on the buffer size.

In such scenario:

The producer will have to wait if the buffer is full.

The consumer has to wait if the buffer is empty.

## PRODUCER-CONSUMER CONCEPT (3) – BOUNDED BUFFERS (I)

The following code segment declares the variables to be used to implement the producer-consumer concept using a bounded buffer:

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

The above variables are stored in region of memory shared by both the producer and consumer.

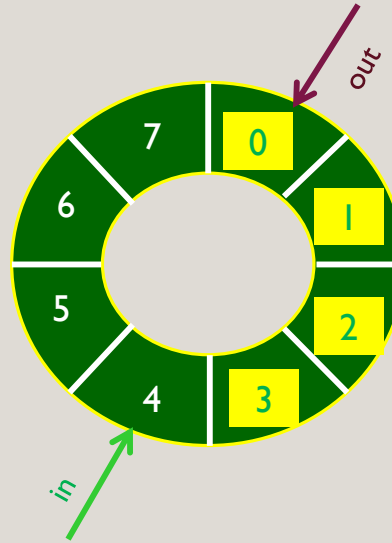
The shared buffer is implemented as a circular array using two pointers *in* and *out*.

The pointer *in* points to the next free position in the buffer.

The pointer *out* points to the first full position in the buffer.

## PRODUCER-CONSUMER CONCEPT (4) – BOUNDED BUFFERS (2)

Let us explore the following figure that represents a circular array with BUFFER\_SIZE = 8. They are indexed from 0 to 7.



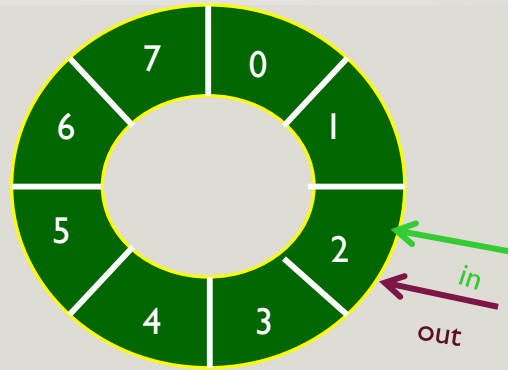
Assume we have 4 items in the buffer (denoted from 0 to 3).

The pointer *in* points to the next free position in the buffer.

The pointer *out* points to the first full position in the buffer.

**PRODUCER-CONSUMER CONCEPT (5) – BOUNDED BUFFERS (3) – EMPTY BUFFER**

Let us consider the case of an empty buffer:



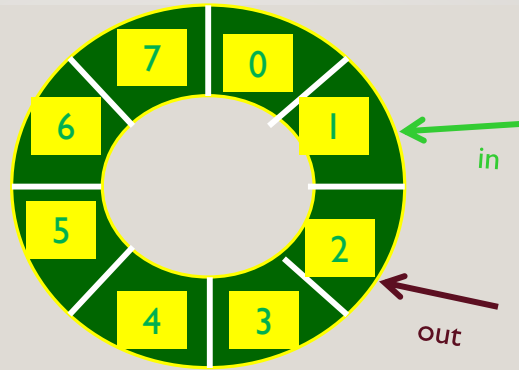
The pointer *in* points to the next free position in the buffer.

The pointer *out* points to the first full position in the buffer (no position is full in this case, but this the last position for *out*).

The buffer is empty when  $in = out$ .

**PRODUCER-CONSUMER CONCEPT (6) – BOUNDED BUFFERS (4) – FULL BUFFER**

Let us consider the case of a full buffer:



The pointer *in* points to the next free position in the buffer. No position is free in this case; but this the last position for *in*.

The pointer *out* points to the first full position in the buffer.

The buffer is full when  $((in + 1) \% BUFFER\_SIZE) = out$ .

**PRODUCER-CONSUMER CONCEPT (7) – BOUNDED BUFFERS (5) – PRODUCER CODE**

The following is code of the producer processor in a bounded buffer:

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out)
        /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Buffer is full

→ producer waits

An item is produced at “*in*” position

Update the position of the “*in*” pointer



**PRODUCER-CONSUMER CONCEPT (8) – BOUNDED BUFFERS (6) – CONSUMER CODE**

The following is the code of the consumer processor in a bounded buffer:

```
item next_consumed;
```

```
while (true) {
```

```
    while (in == out)
```

Buffer is empty

```
        ; /* do nothing */
```

→ consumer waits

```
    next_consumed = buffer[out];
```

An item is consumed

```
    out = (out + 1) % BUFFER_SIZE;
```

Update the position of the “out” pointer

```
    /* consume the item in next consumed */
```

```
}
```