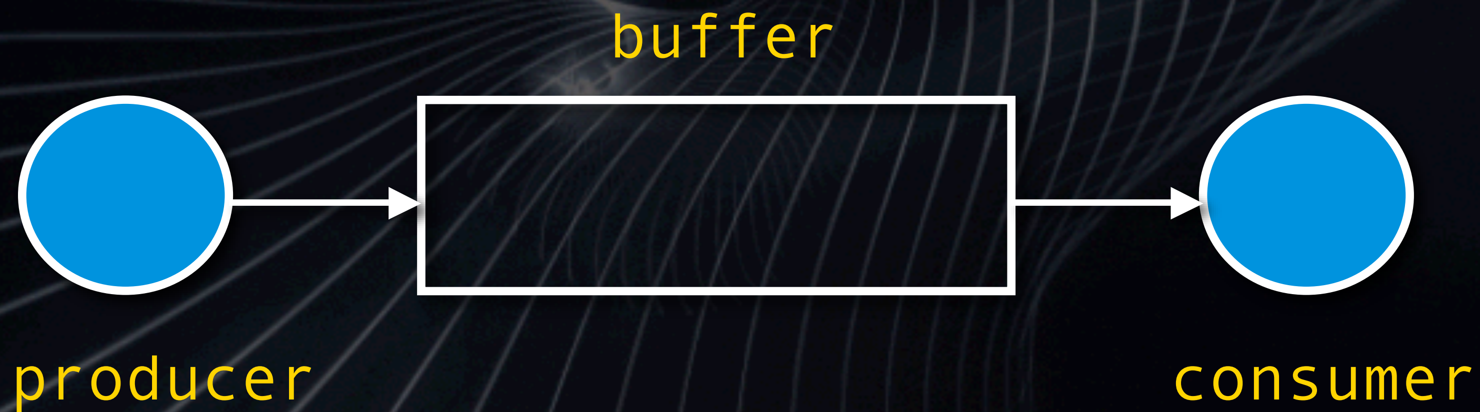
message passing model

# PRODUCER-CONSUMER PROBLEM



## Two semaphores

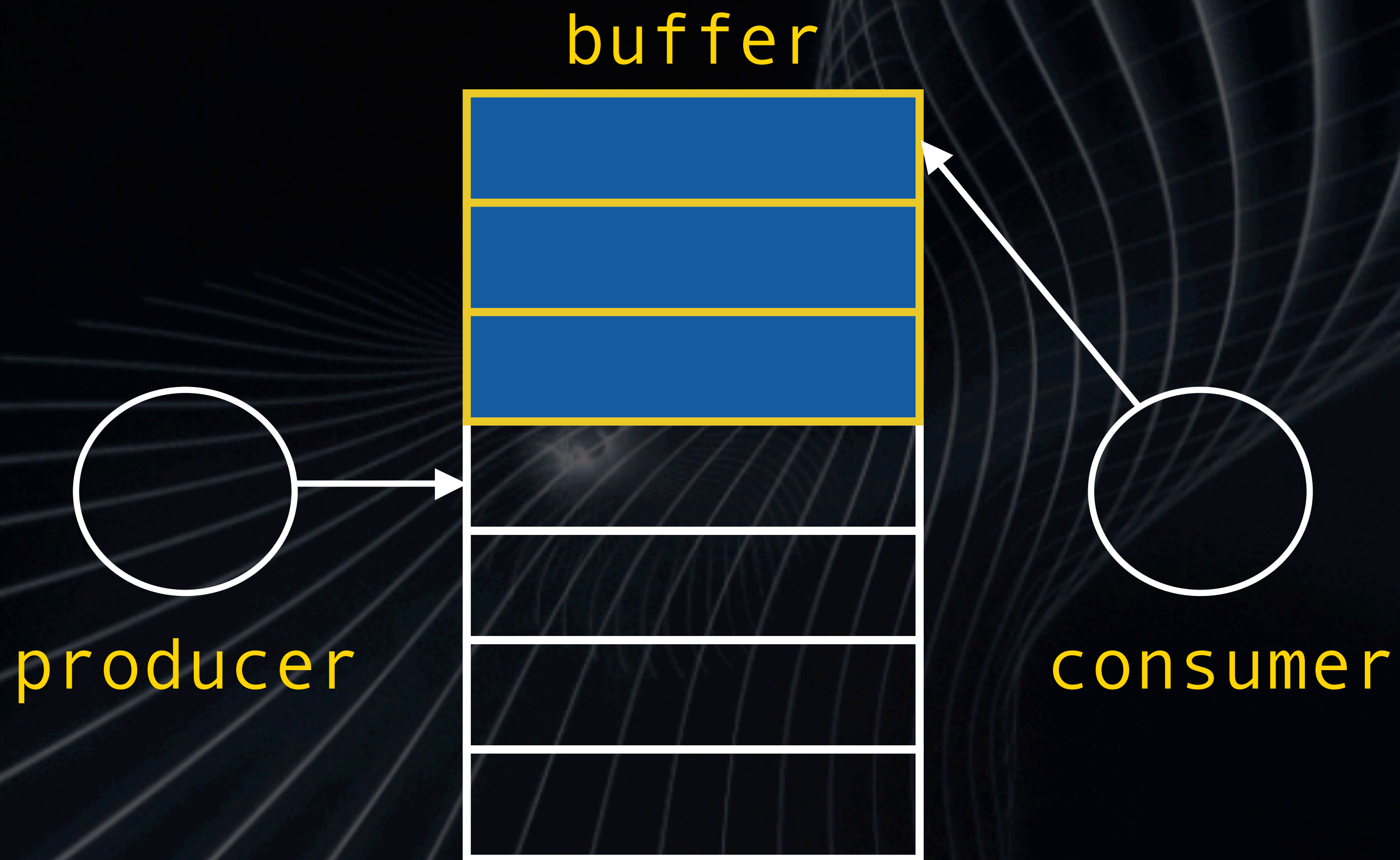
empty (i.v. 1)

full (i.v. 0)

```
void producer(void)
{
while (TRUE) {
        <generate message to put in the
buffer >;
        P (empty);
        <put new message in the buffer>;
        V (full);
        }
}
```

```
void consumer(void)

{
while (TRUE) {
        P (full);
        <take one message from the buffer> ;
        V(empty);
        <consume the message>;
          }
}
```


buffer
producer
consumer

## Three semaphores

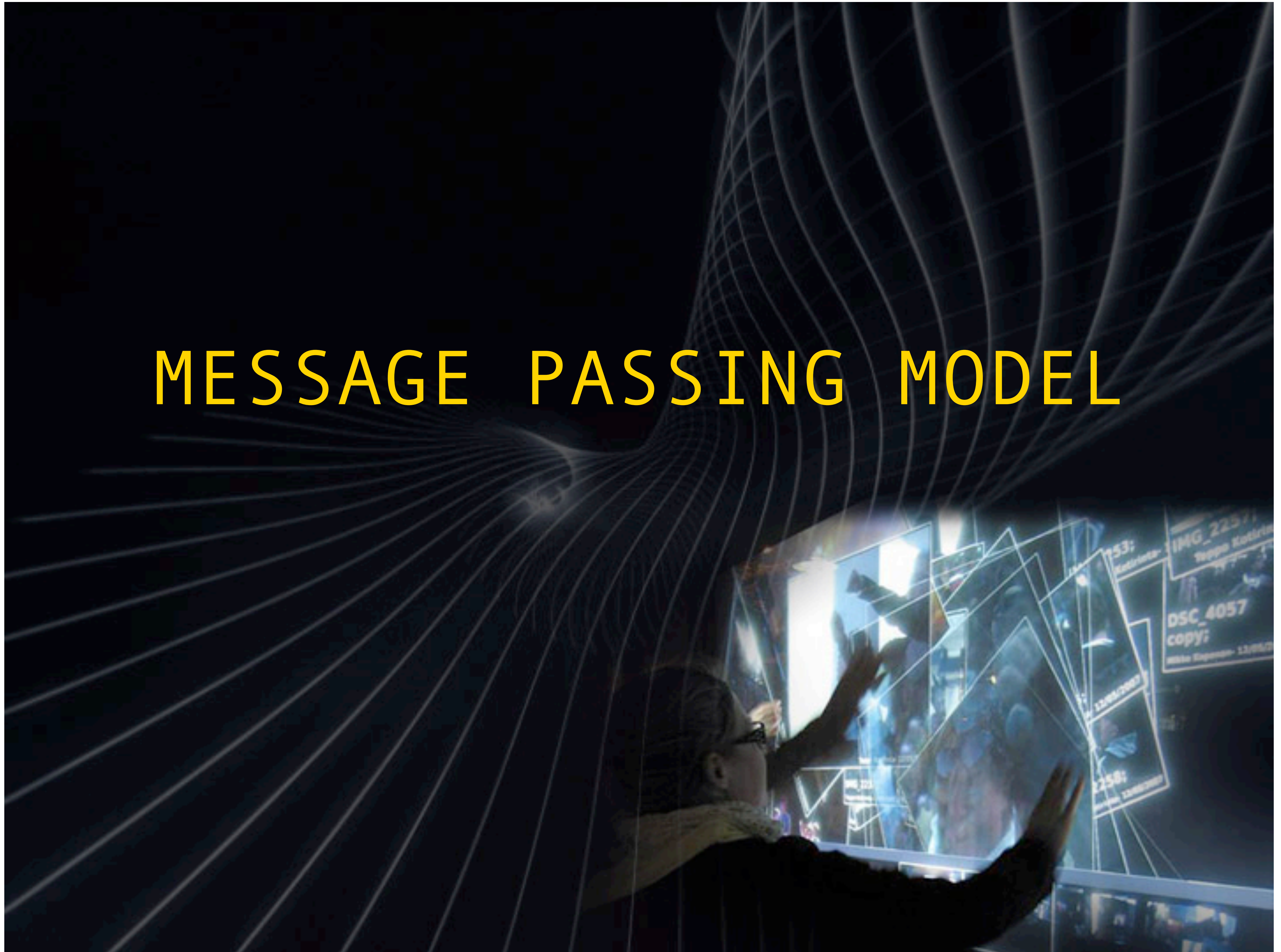
empty (i.v. N)

full (i.v. 0)

mutex (i.v. 1)

```
void producer (void)
{ while (TRUE)
{< generate one message to be put into
 the buffer >;
P (empty);
P (mutex);
<put the new message in the buffer>;
V (mutex);
V(full);}
     }
```

```
void consumer(void)
{while (TRUE)
{P (full);
P(mutex);
<take one message from the buffer>;
V(mutex);
V(empty);
<consume the message>;}
}
```


MESSAGE PASSING MODEL
DSC_4057
copy;

The functions of message passing are normally provided in the form of a couple of primitives:

- → send (destination, message)
- → receive (source,message)

→ One process sends information in the form of a message to another process designated by a destination

→ A process receives information by executing the receive primitive, to obtain the source of the sending process and the message

→Addressing

→Synchronization

## Addressing

→Direct addressing

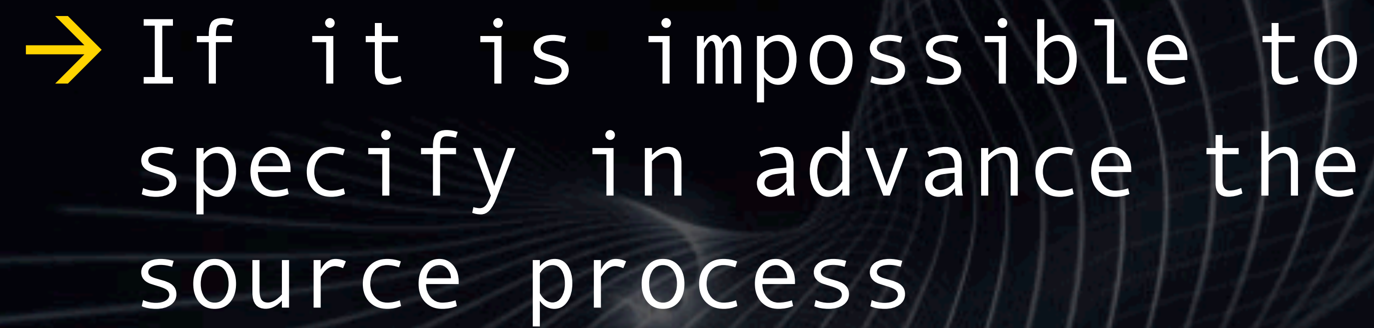
The send primitive includes a specific identifier for the destination process

```
send (P2, message)
```

The receive can be handled in one of the two ways:

→ The process explicitly design a sending process:

```
receive (P1, message)
```


→ If it is impossible to specify in advance the source process

→ implicit addressing: the source parameter specifies a value yelded when the receive operation has been performed to indicate the sender process.

Indirect addressing
Messages are not directly sent from senders to receivers
DSC_4057
copy;

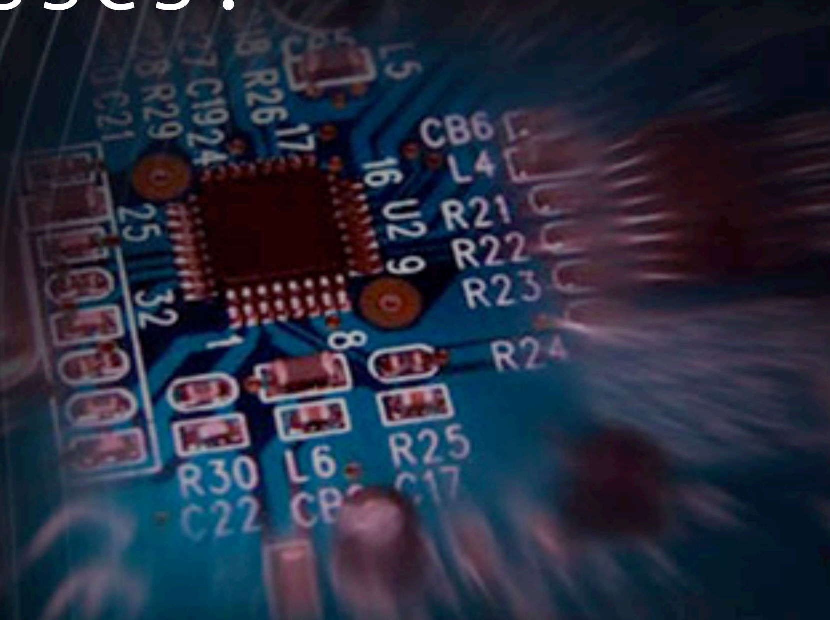
Messages are sent to intermedia shared data structures (mailbox) that can temporaly hold messages

one-to-one

many-to-one

many-to-many

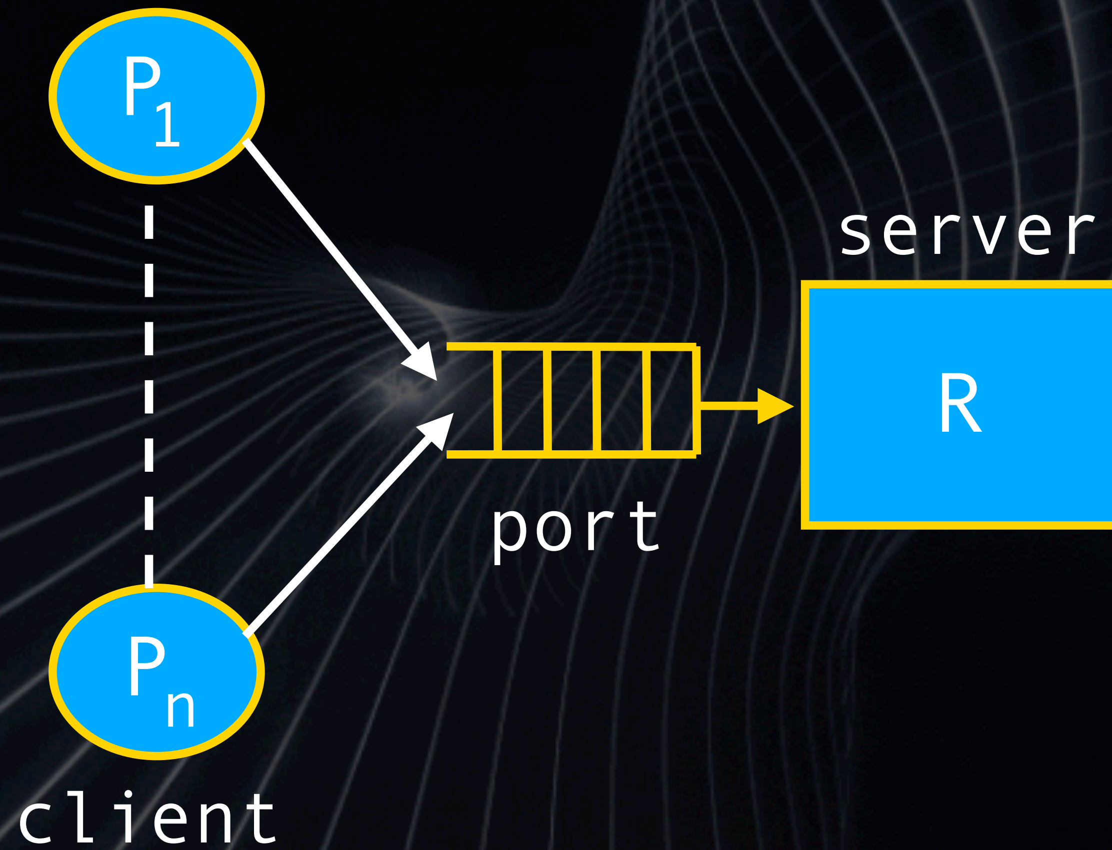
A one-to-one relationship allows private communication to be set up betweeen a couple of processes.

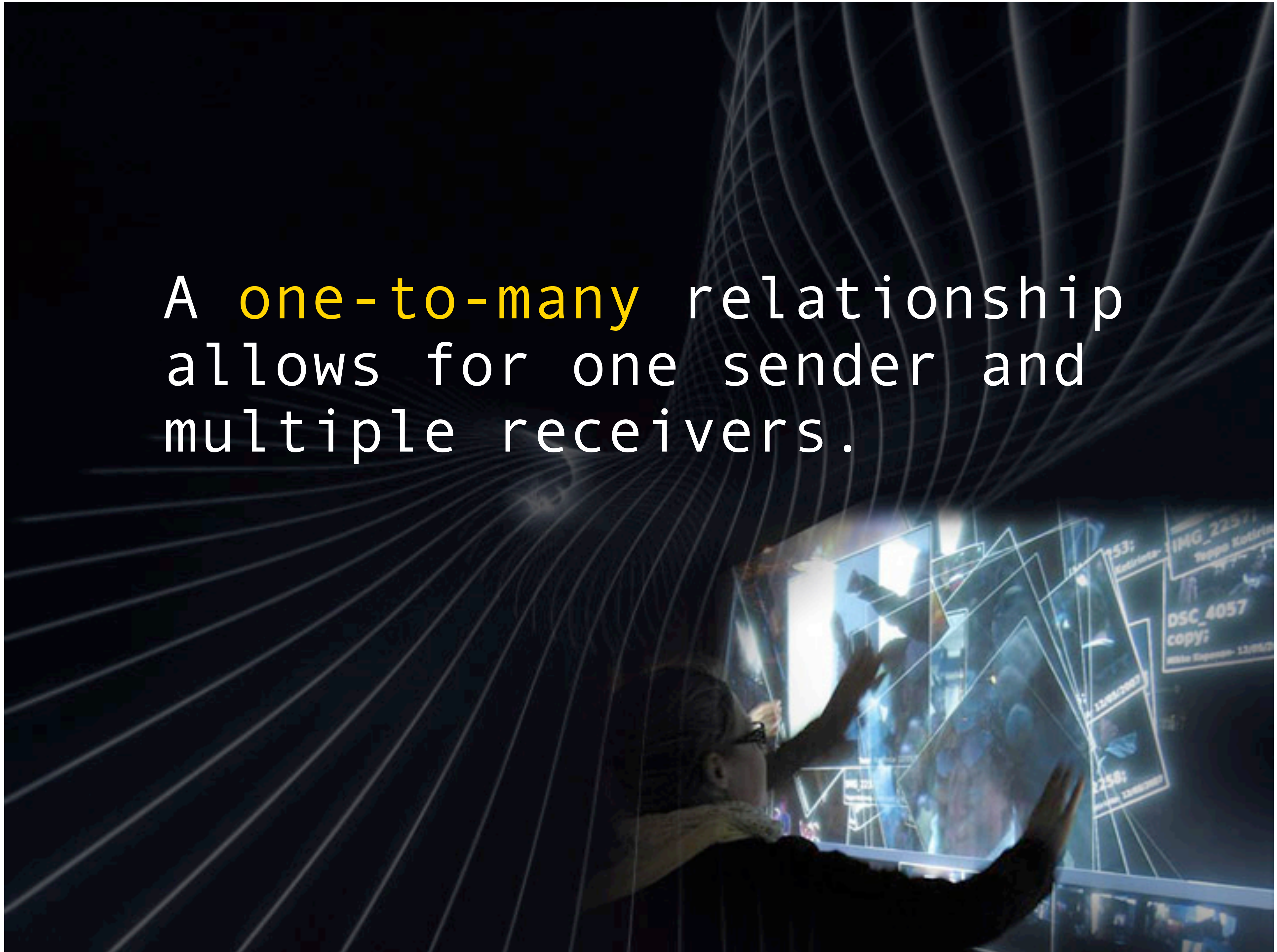


A many-to-one relationship is useful for client-server interaction.

The mailbox is often referred to as a port

# client/server model




A one-to-many relationship
allows for one sender and
multiple receivers.

It is useful for applications where messages are to be broadcasted to groups of reveiver processes.

# Message

header

- message type
- destination ID
- source ID
- message length
- control information

body

- message contents

## SYNCHRONIZATION

The communication of a message between two processes implies some level of synchronization

## SYNCHRONIZATION

The receiver cannot receive a message until it has been sent by another process

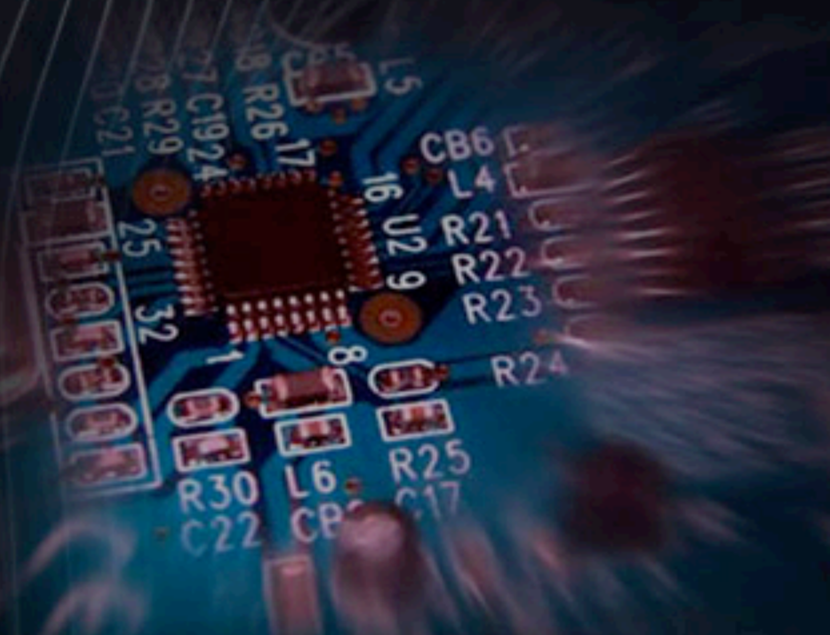
In addition, we need to specify what happens to a process after a send or receice primitive.

Message passing may be either **blocking or nonblocking** (synchronous or asynchronous)

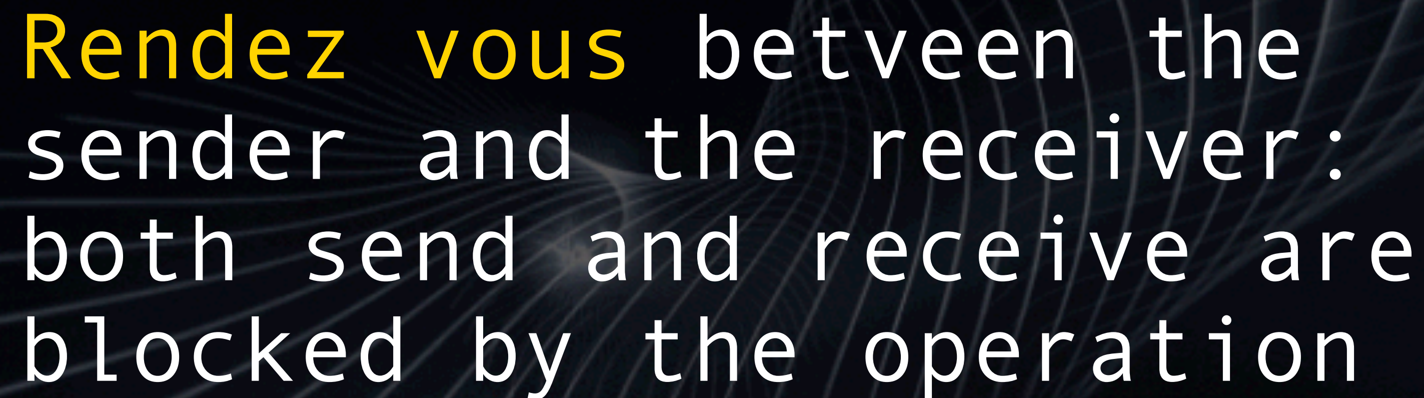
**Blocking send**: the sending process is blocked until the message is received either by the receiving process or by the mailbox.

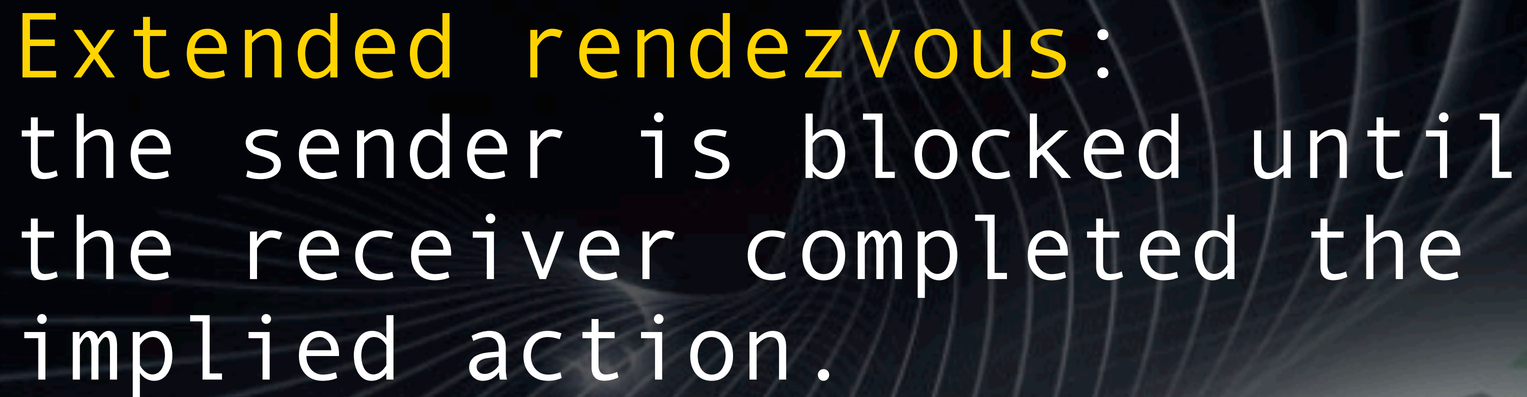
Nonblocking send: the
sending process sends the
message and immediately
resumes operation.

**Blocking receive**: the receiver blocks until a message is available.

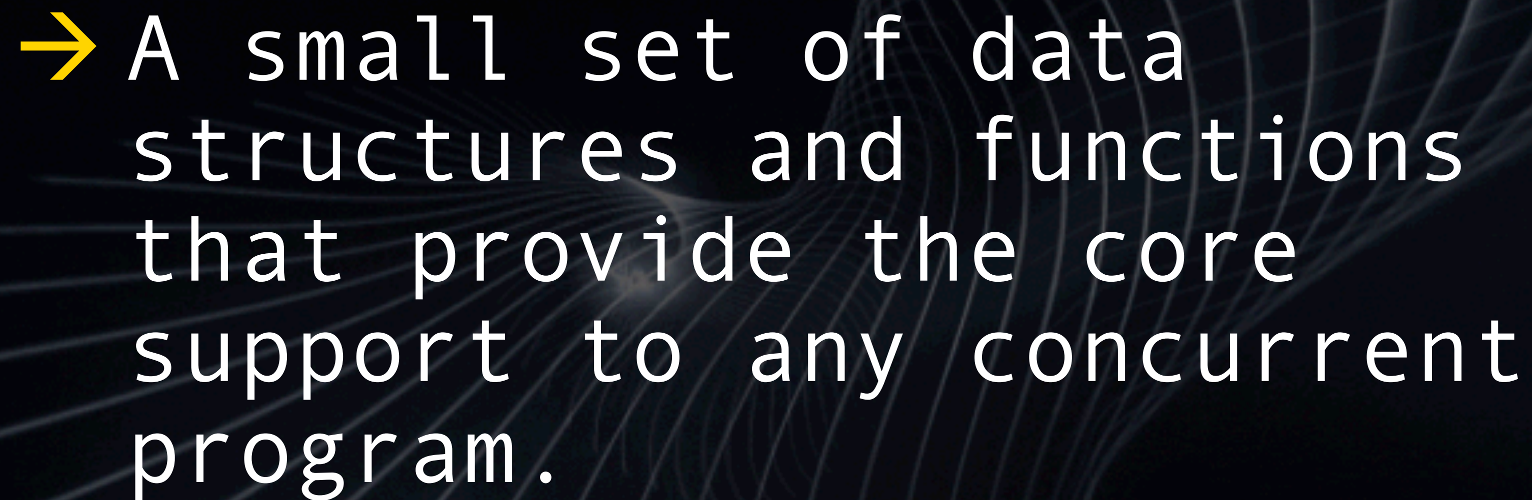


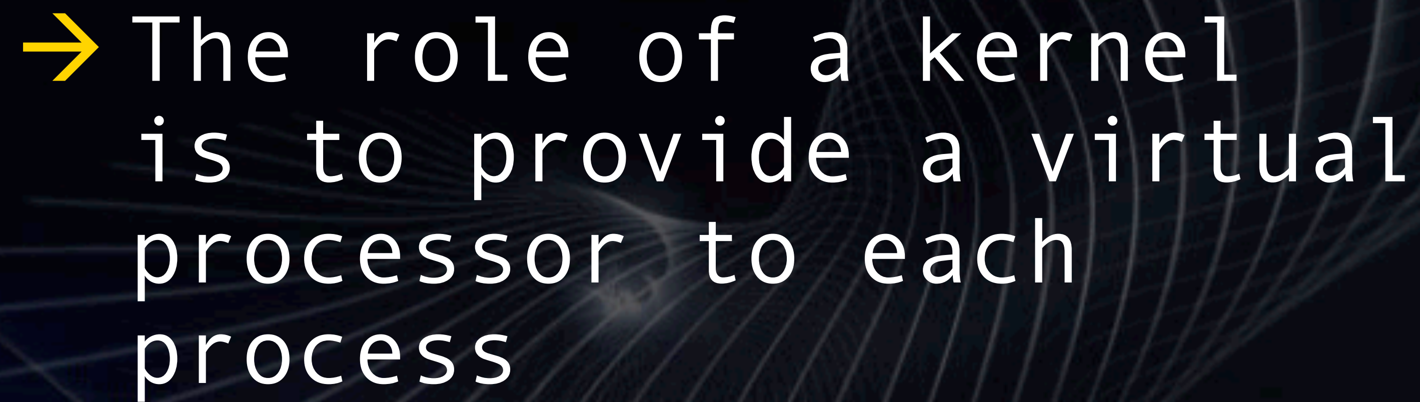
**Nonblocking receive:**
if there is no waiting message, the process continues executing, abandoning the attempt to receive.


Rendez vous betveen the
sender and the receiver:
both send and receive are
blocked by the operation


Extended rendezvous:
the sender is blocked until
the receiver completed the
implied action.

KERNEL OF A
PROCESS SYSTEM


→ A small set of data structures and functions that provide the core support to any concurrent program.


→ The role of a kernel
is to provide a virtual
processor to each
process

# Kernel data structures

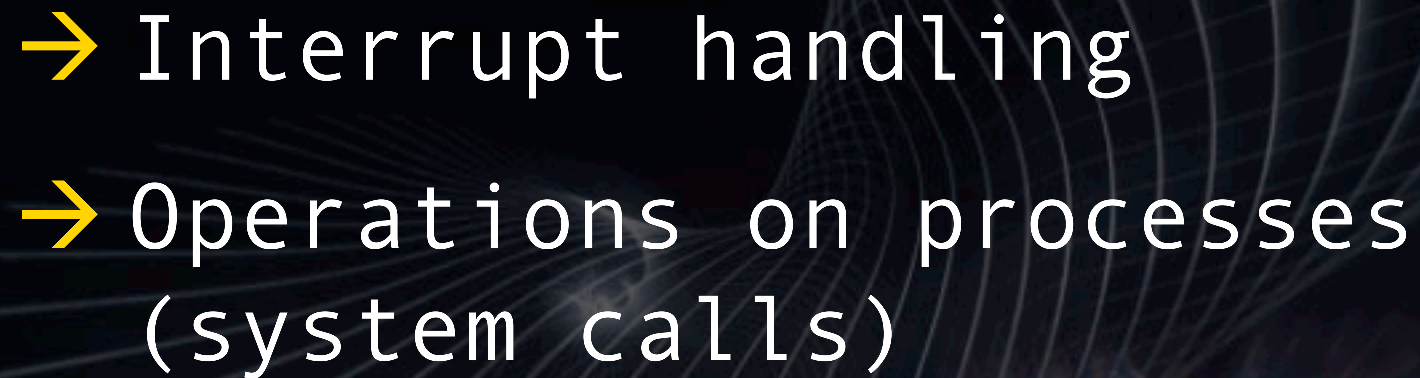
- → PCBs
- → process_in_execution
- → ready-process-queues
- → semaphores and blocked-process-queues

## Kernel functions

→Context switch management: save and restore the PCBs of the processes.

## Kernel functions

→ Decision of the ready process to which assign the CPU.


→ Interrupt handling
→ Operations on processes (system calls)

- → The kernel starts executing when an interrupt occurs
- → External interrupts from peripheral devices

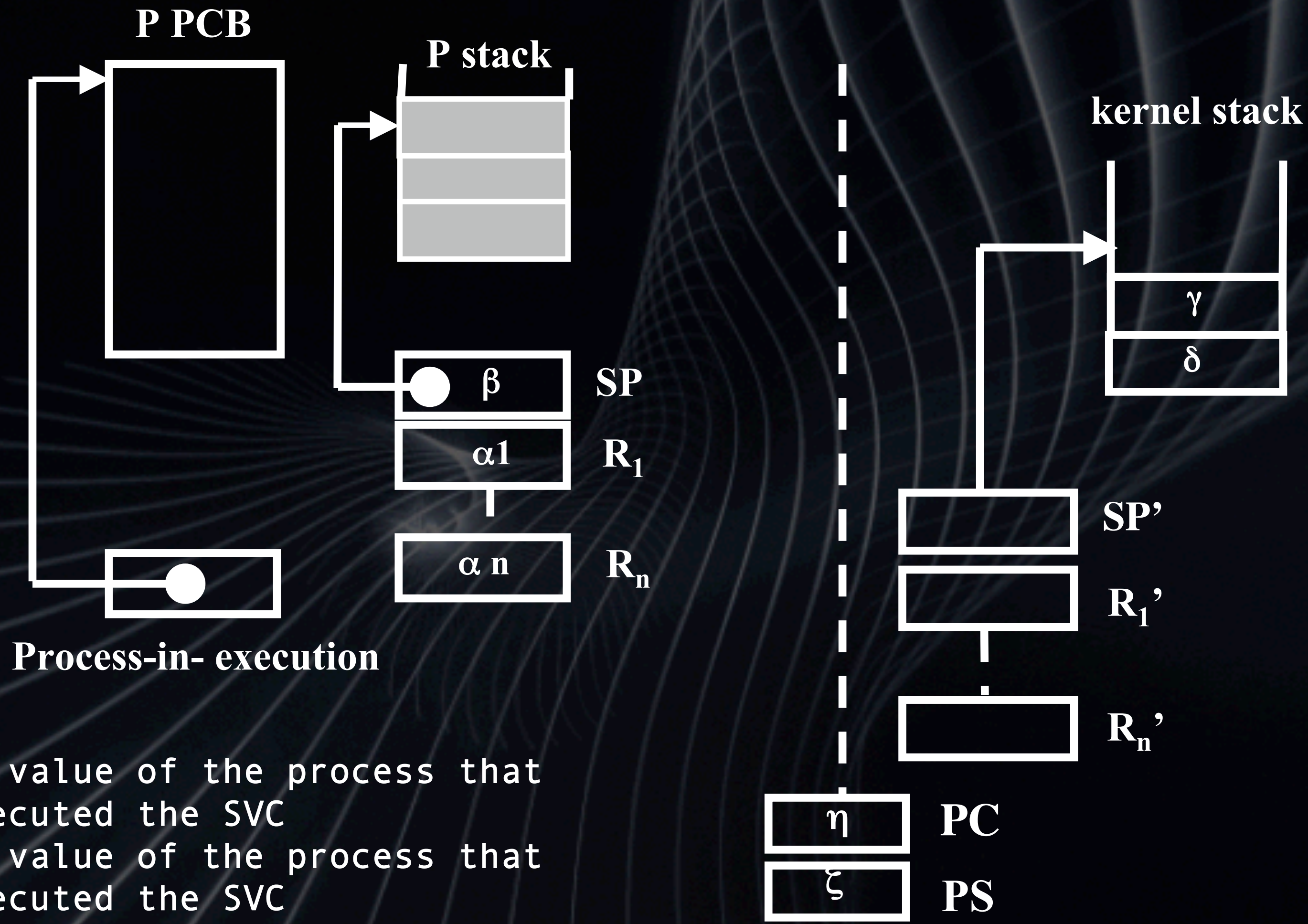
→ Internal interrupts or traps triggered by the executing process

## Example

→ A stack is associated to any process

## Example

→ Double set of general registers R1, R2, …, Rn and R’1, R’2, …, R’n and two registers SP and SP’ (user and kernel)

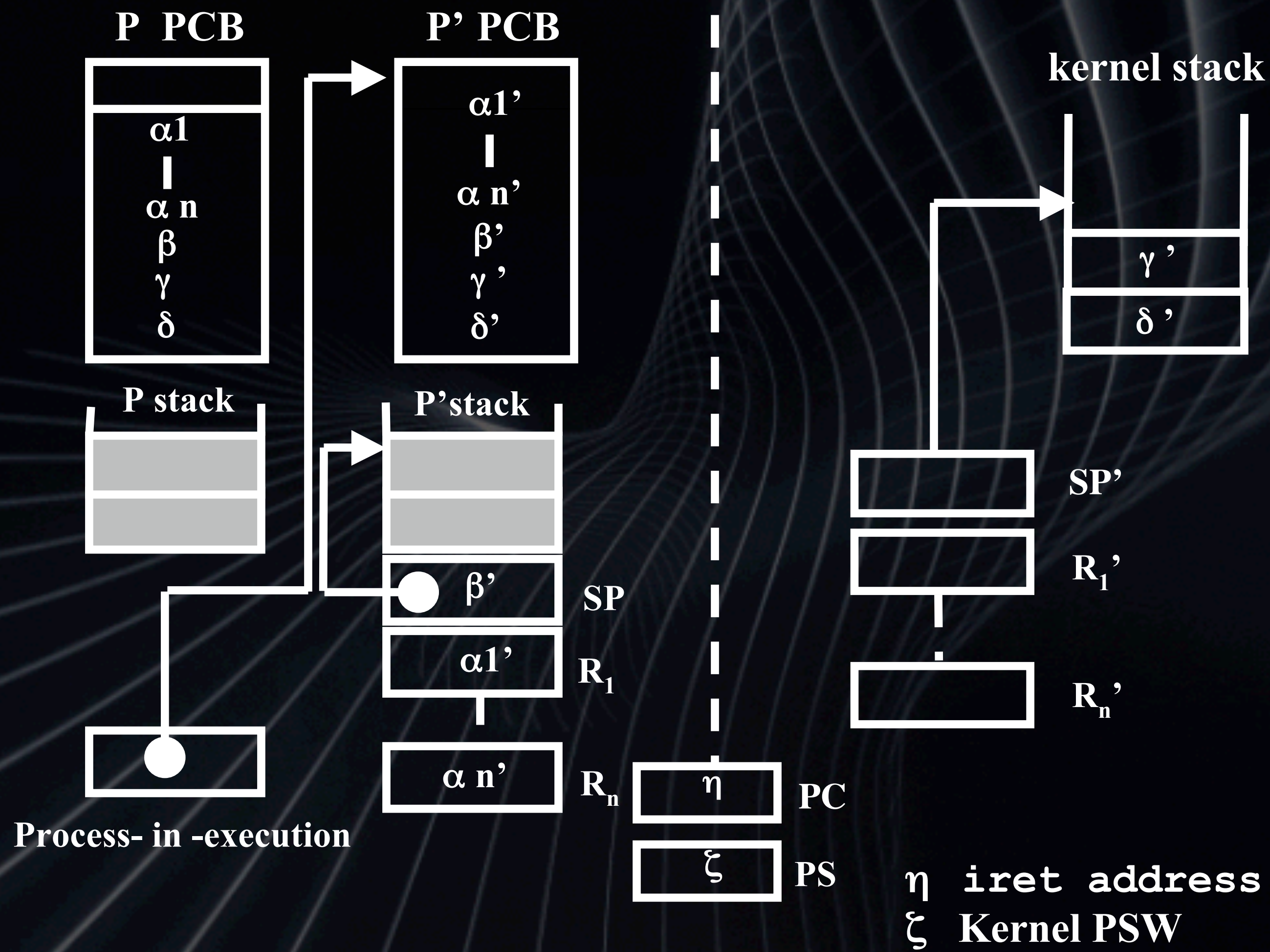



g PC value of the process that executed the SVC

d PS value of the process that executed the SVC

ε interrupt handler address

z kernel PSW


P PCB
α1
α n
β
γ
δ
P' PCB
α1'
α n'
β'
γ '
δ'
kernel stack
γ '
δ '
P stack
P'stack
β'
SP
α1'
$R_1$
α n'
$R_n$
SP'
$R_1$'
$R_n$'
η
PC
ζ
PS
Process- in -execution
η iret address
ζ Kernel PSW