

A hand is shown typing on a keyboard, with glowing lines representing data flow or network connections. The text "message passing model" is overlaid on the image.

message passing model

# PRODUCER-CONSUMER PROBLEM



## Two semaphores

empty (i.v. 1)

full (i.v. 0)

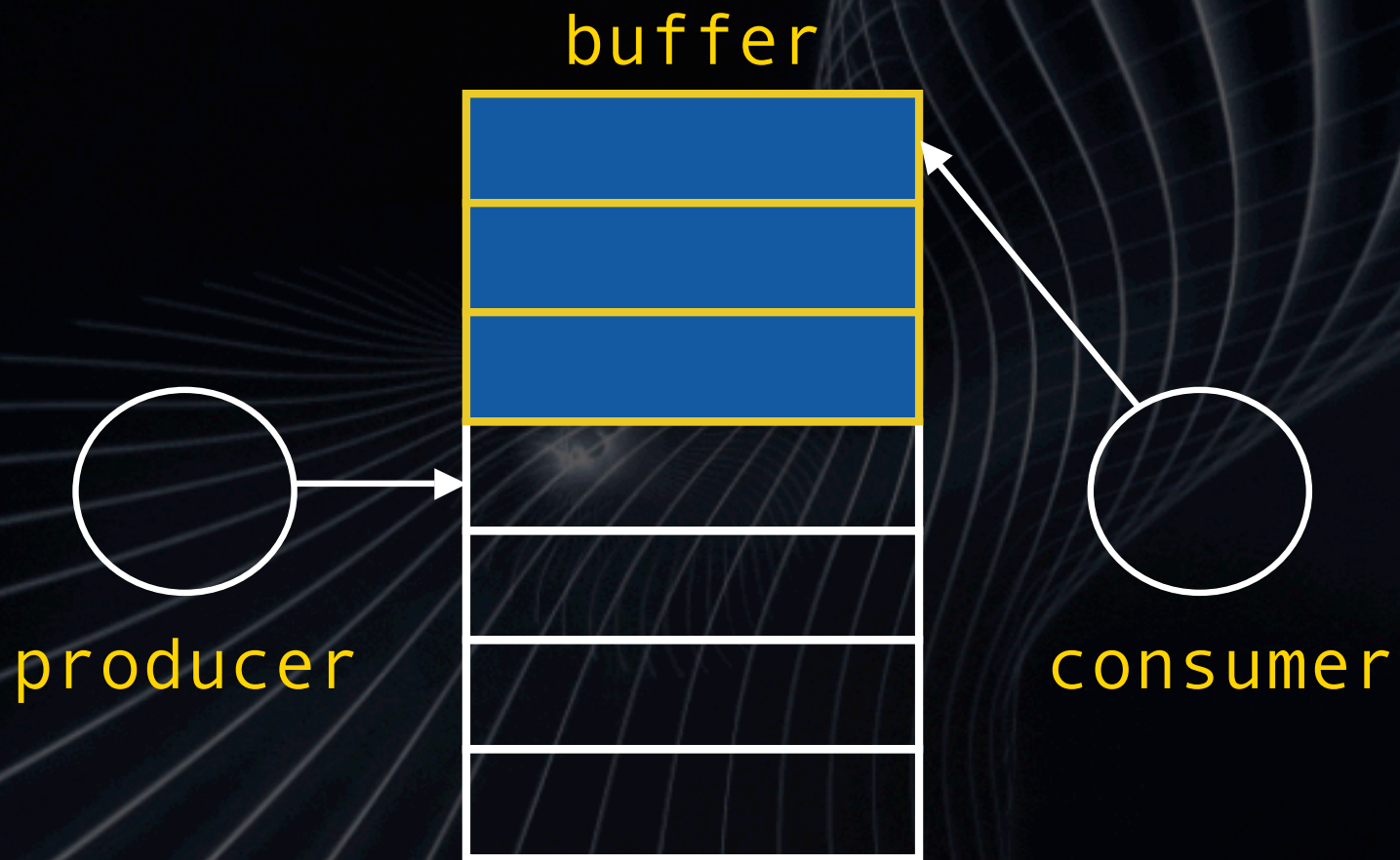


```
void producer(void)
```

```
{  
  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>;  
    }  
}
```



## 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>;
```

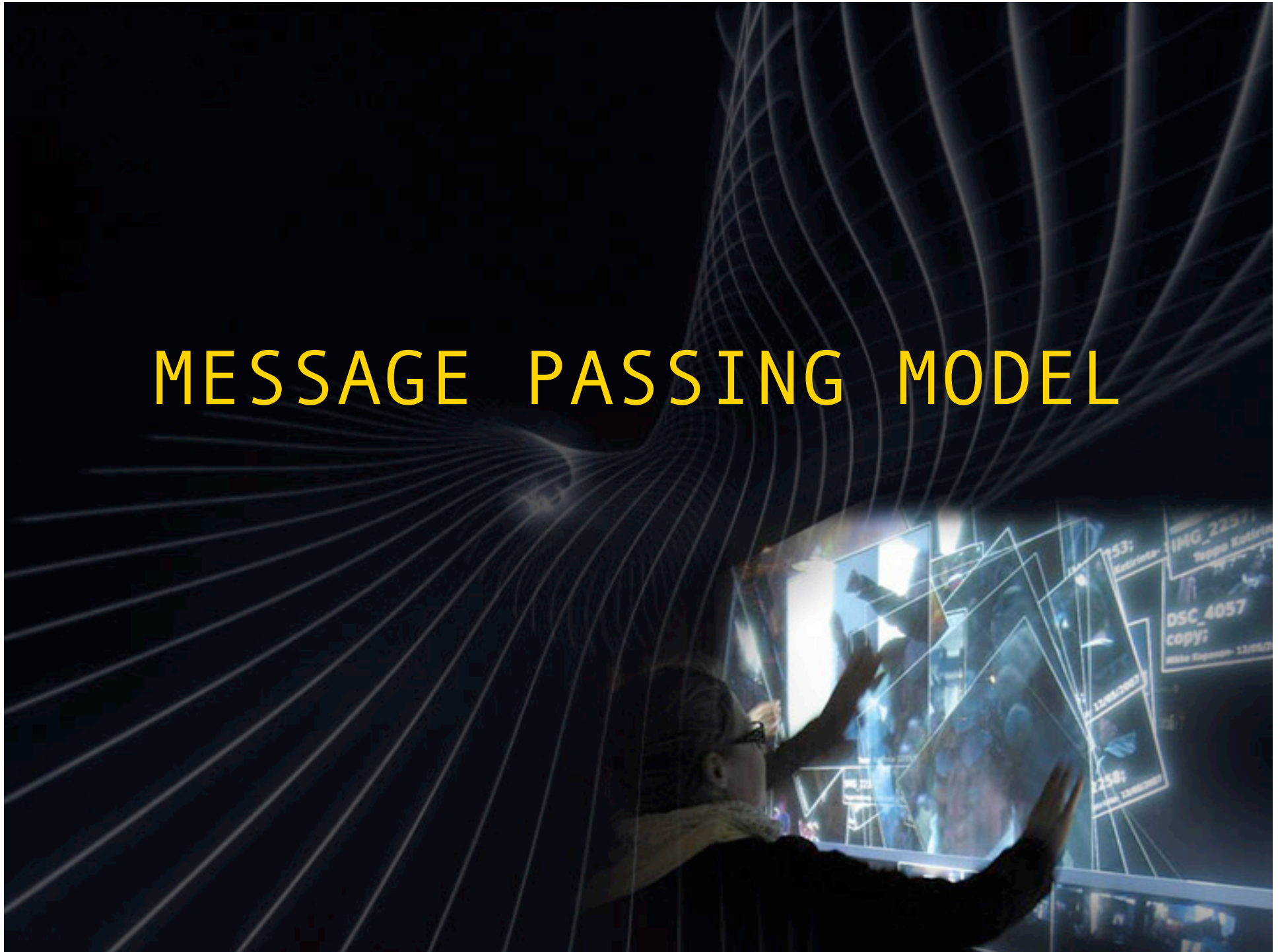
```
V(mutex);
```

```
V(empty);
```

```
<consume the message>;}
```

```
}
```

# MESSAGE PASSING MODEL



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**

```
total memory = 4096000, 4092000 used, 4000 free, 4092000 loaded,
total paging = 4096 + 3476 private, 2620 shared,
1420 active, 2780 inactive, 18880 used, 320 free,
1000000000 pages, 201480000 pageouts
```

PPID	TIME	#IN	#OUT	PROCESS	SENT	RECV	DEST	VSZ
1	0:00.00	1	0	71	30K	272K	180K	3432
1	0:00.00	1	14	19	256K	62K	80K	768
1	0:00.00	1	0	55	24K	24K	70K	752
1	0:01.04	1	21	29	2832K	18K	252K	768
1	0:00.01	2	73	73	956K	236K	514K	2448
1	0:00.01	1	14	19	24K	62K	80K	768
1	0:00.01	1	0	55	24K	24K	70K	752
1	0:00.13	4	53	30	64K	222K	194K	872
1	0:03.04	7	189	185	424K	118	170	4192
1	0:00.02	22	282	768	668	318	910	5072
1	0:00.09	15	311	585	458	250	758	5184
1	0:00.00	14	204	724	1438	180	1748	4272
1	0:00.05	5	36	182	200K	312K	414K	1808
1	0:00.00	14	143	250	856K	844K	220	4488
1	0:00.01	3	24	25	28K	272K	194K	768
1	0:00.01	1	19	21	4K	104K	24K	768
1	0:00.00	2	81	58	16K	14K	128K	832
1	0:00.00	2	25	28	6K	17K	272K	768
1	0:00.00	20	352	919	1818	458	1858	13184
1	0:00.00	1	58	83	184K	227K	94K	3528
1	0:00.00	1	49	42	124K	2112K	80K	2464
1	0:00.00	2	99	81	124K	104K	120K	1168
1	0:00.00	6	238	215	1272K	9452K	234K	4008

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



## Design issues of message systems:

→ Addressing

→ Synchronization

# Addressing

## → Direct addressing

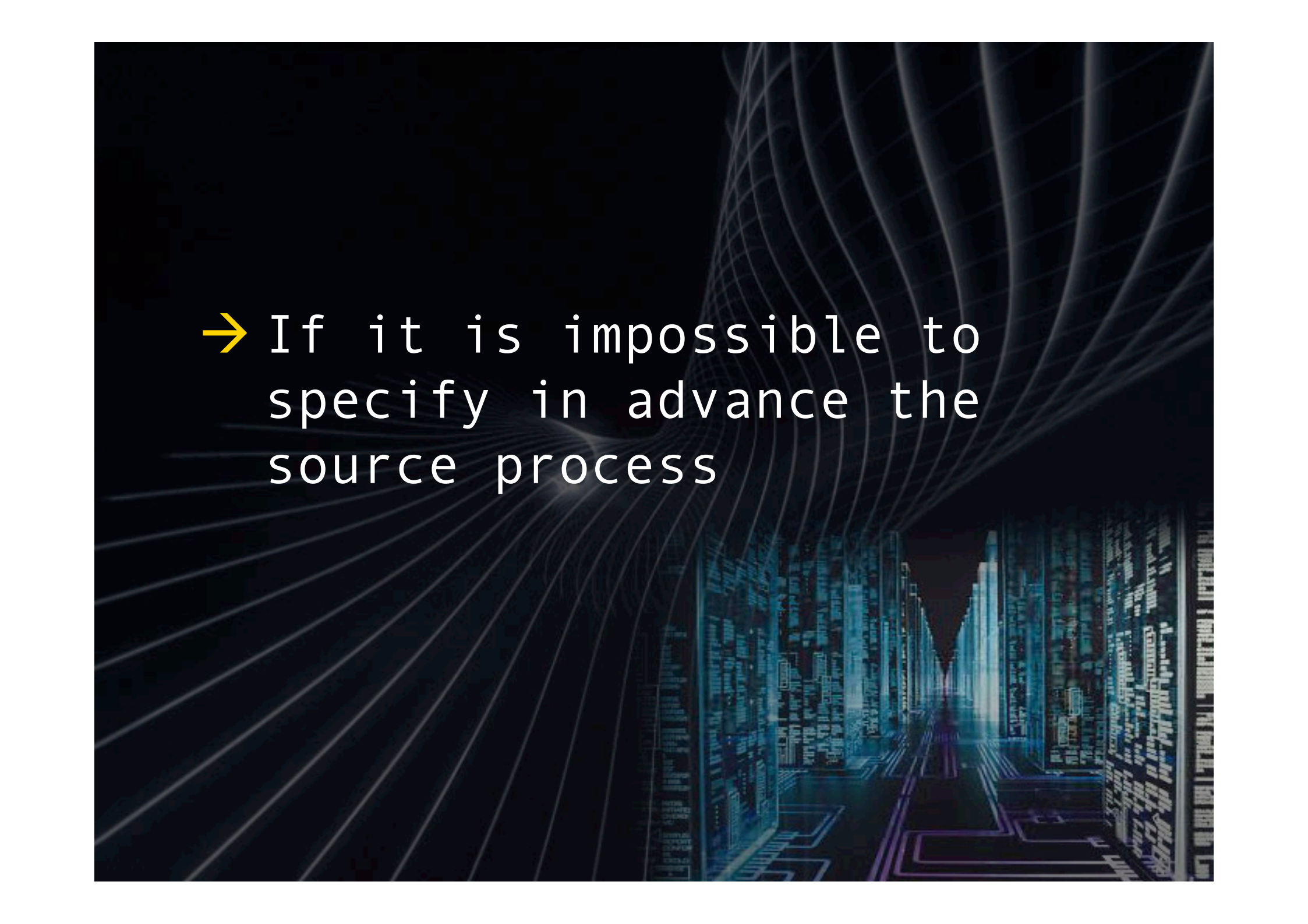
The send primitive includes a specific identifier for the destination process

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

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

→ The process **explicitly**  
design a sending process:  
receive (P1, message)



A futuristic digital tunnel with glowing blue and purple lines and data streams. The perspective is from the center of the tunnel, looking down a long, narrow path that recedes into the distance. The walls and ceiling are composed of numerous thin, curved lines that create a sense of depth and movement. The floor is also composed of similar lines, with some glowing purple and blue patterns. The overall atmosphere is one of high-tech, digital connectivity.

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

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

## Indirect addressing

Messages are not directly sent from senders to receivers





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



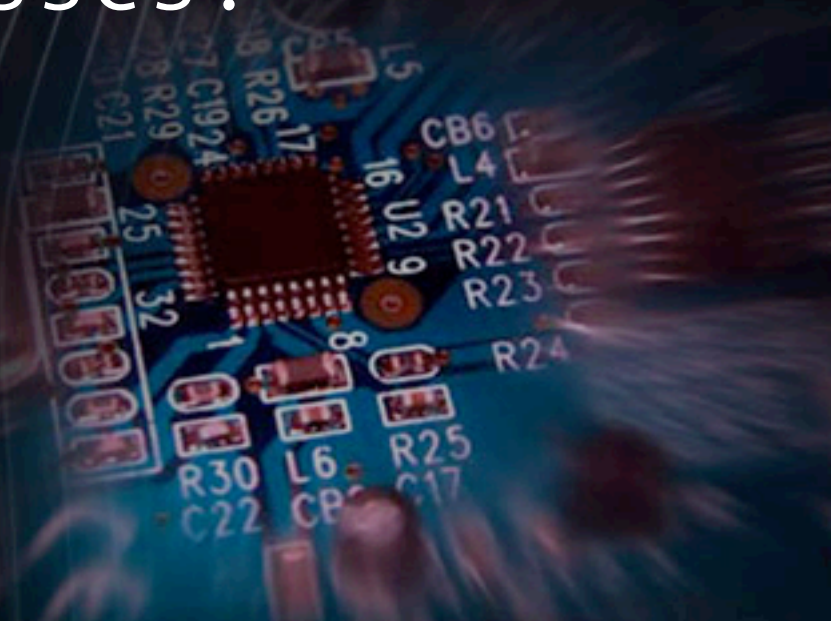
The relationship between  
senders and receivers can  
be:

one-to-one

many-to-one

many-to-many

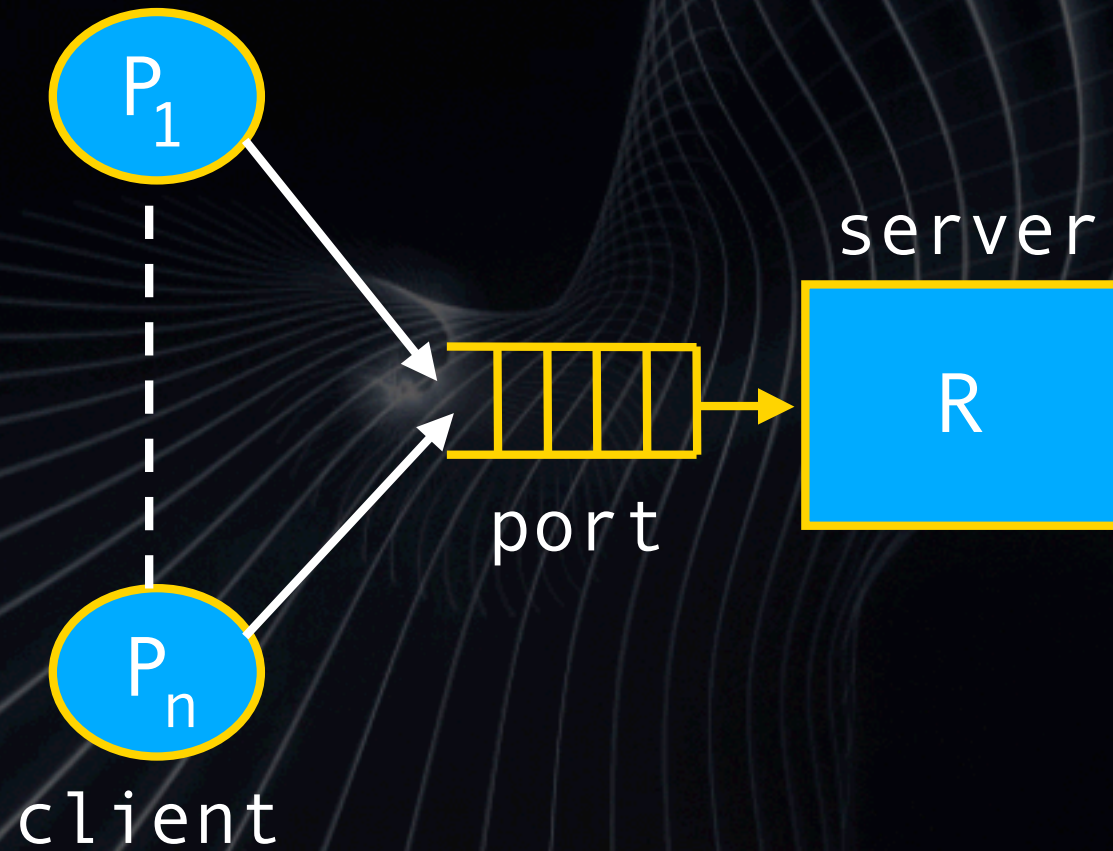
A **one-to-one** relationship allows private communication to be set up between a couple of processes.



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

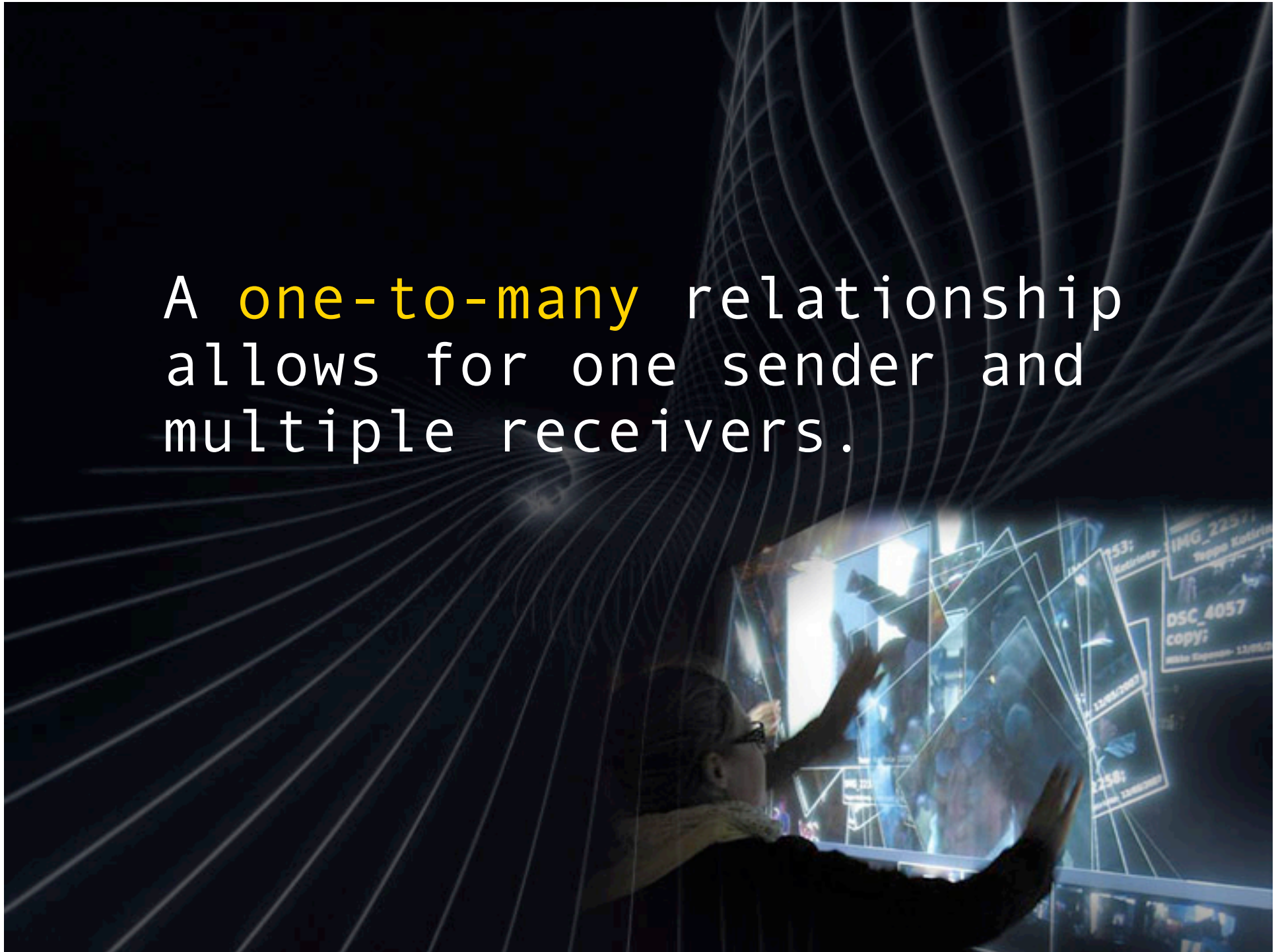
The mailbox is often referred to as a **port**

# 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 receiver processes.

# Message

header

body

message type

destination ID

source ID

message length

control information

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 receive primitive.



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



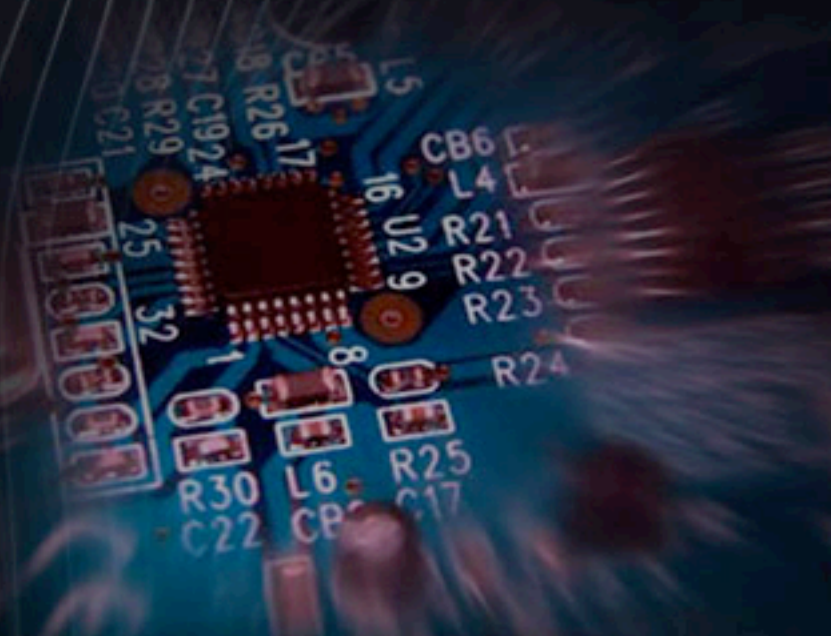
**Blocking send:** the sending process is blocked until the message is received either by the receiving process or by the mailbox.






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

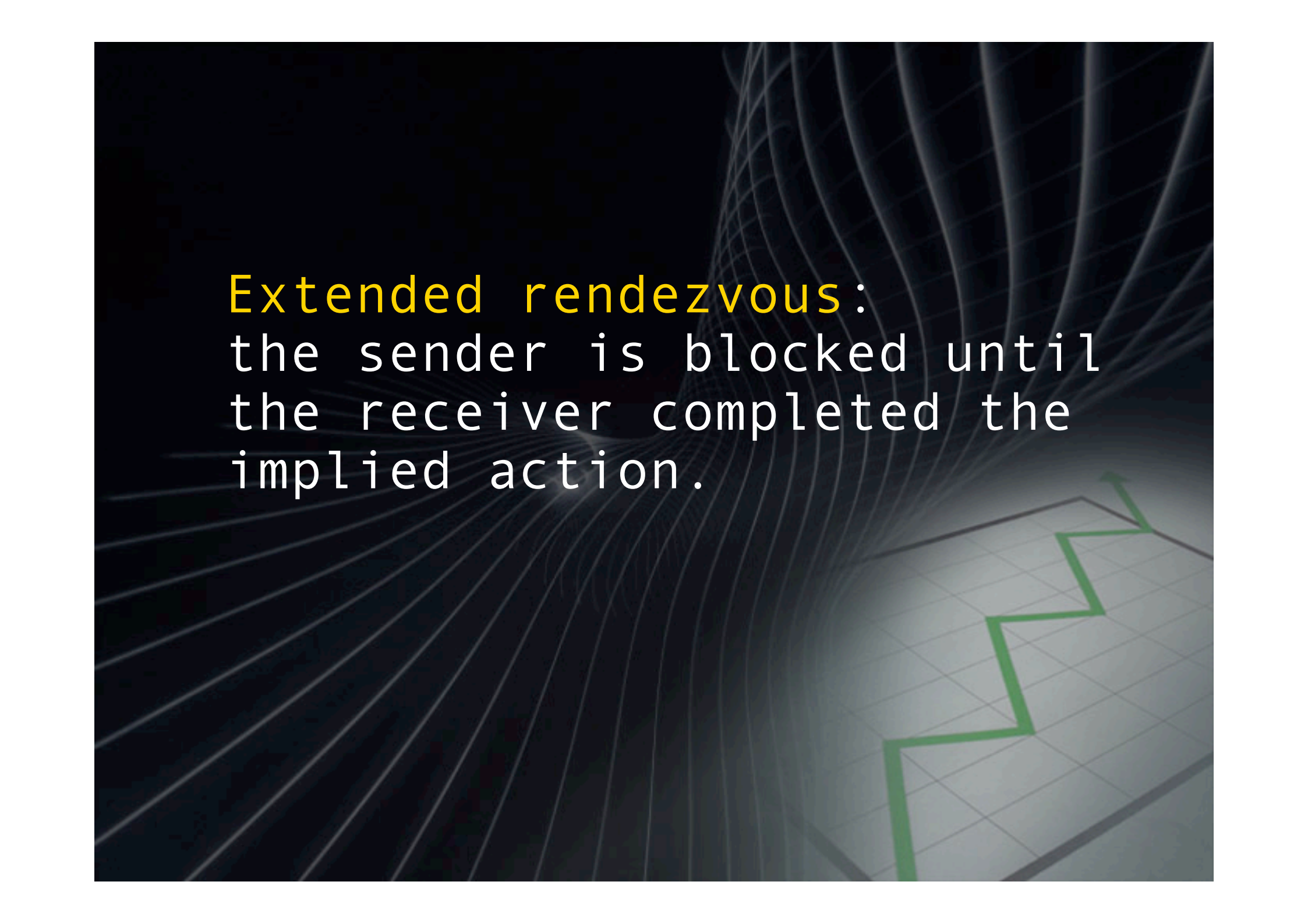
**Blocking receive:** the receiver blocks until a message is available.



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



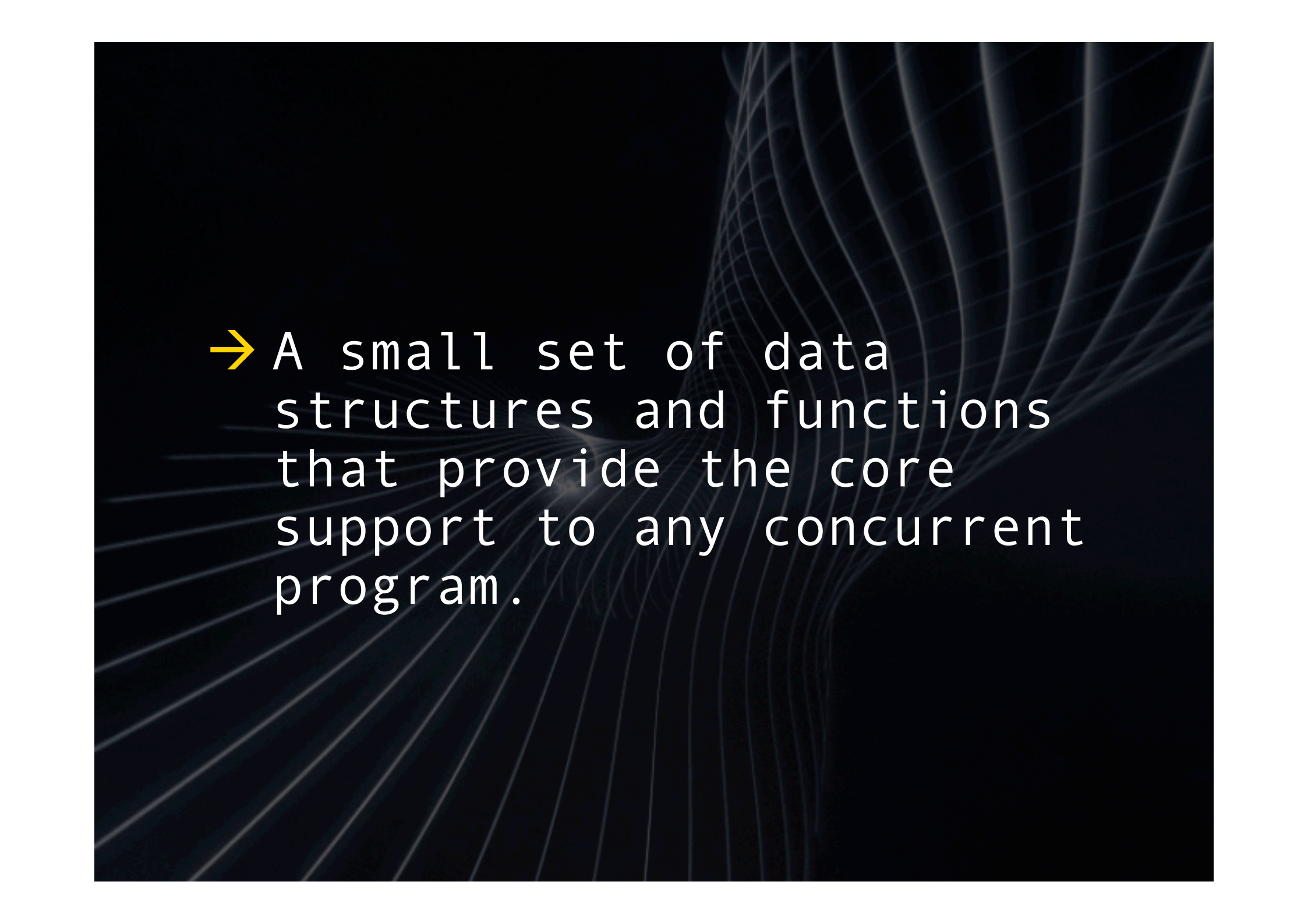
**Rendez vous** between the sender and the receiver:  
both send and receive are blocked by the operation

The background features a dark, wavy grid pattern that creates a sense of depth and movement. In the lower right corner, there is a light-colored grid with a green line graph that trends upwards, symbolizing progress or data analysis.

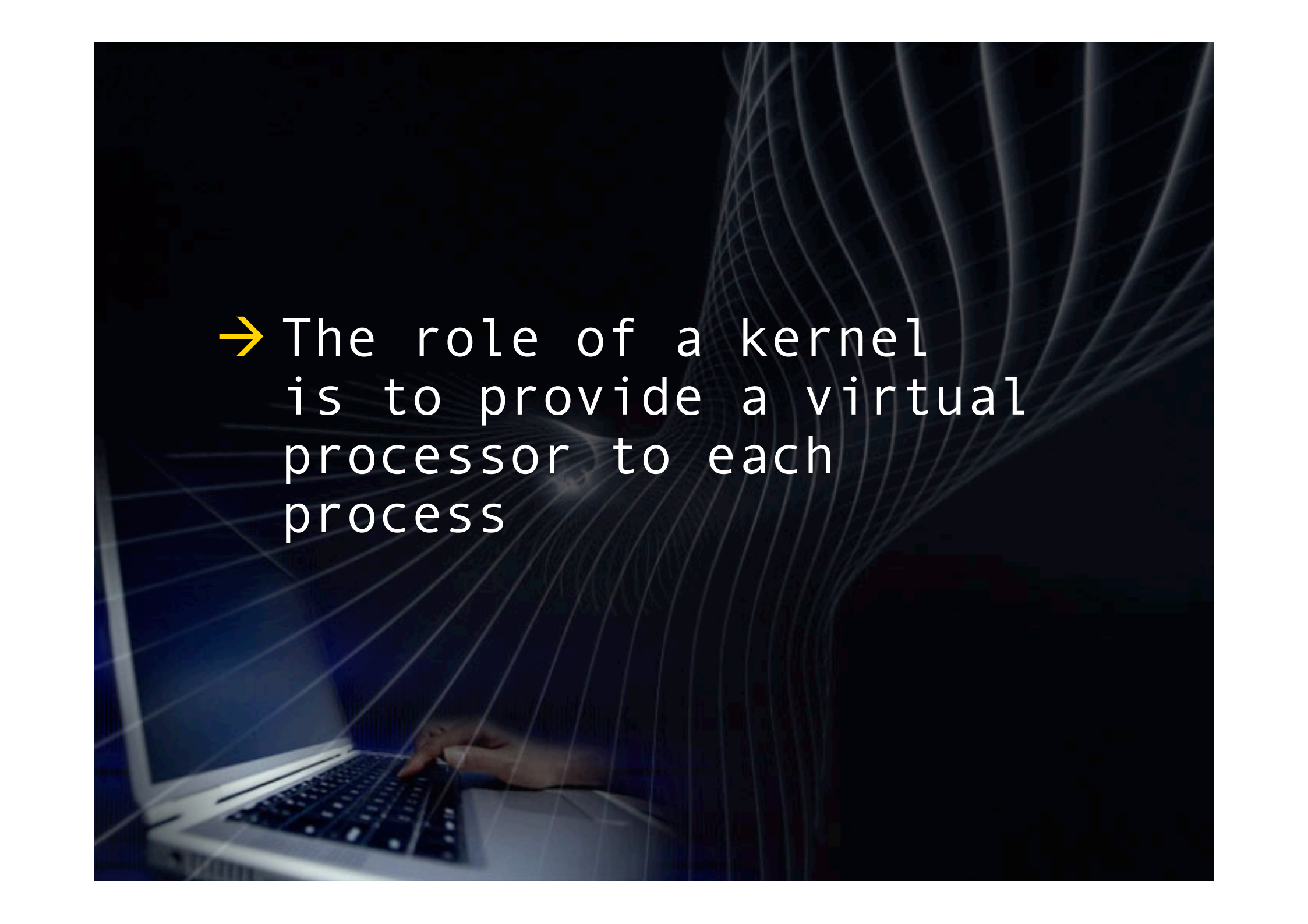
**Extended rendezvous:**  
the sender is blocked until  
the receiver completed the  
implied action.

The background of the slide features a close-up of a hand typing on a keyboard, bathed in a soft blue light. Overlaid on this are numerous thin, white, curved lines that create a sense of motion and digital connectivity, resembling a stylized network or data flow.

# 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

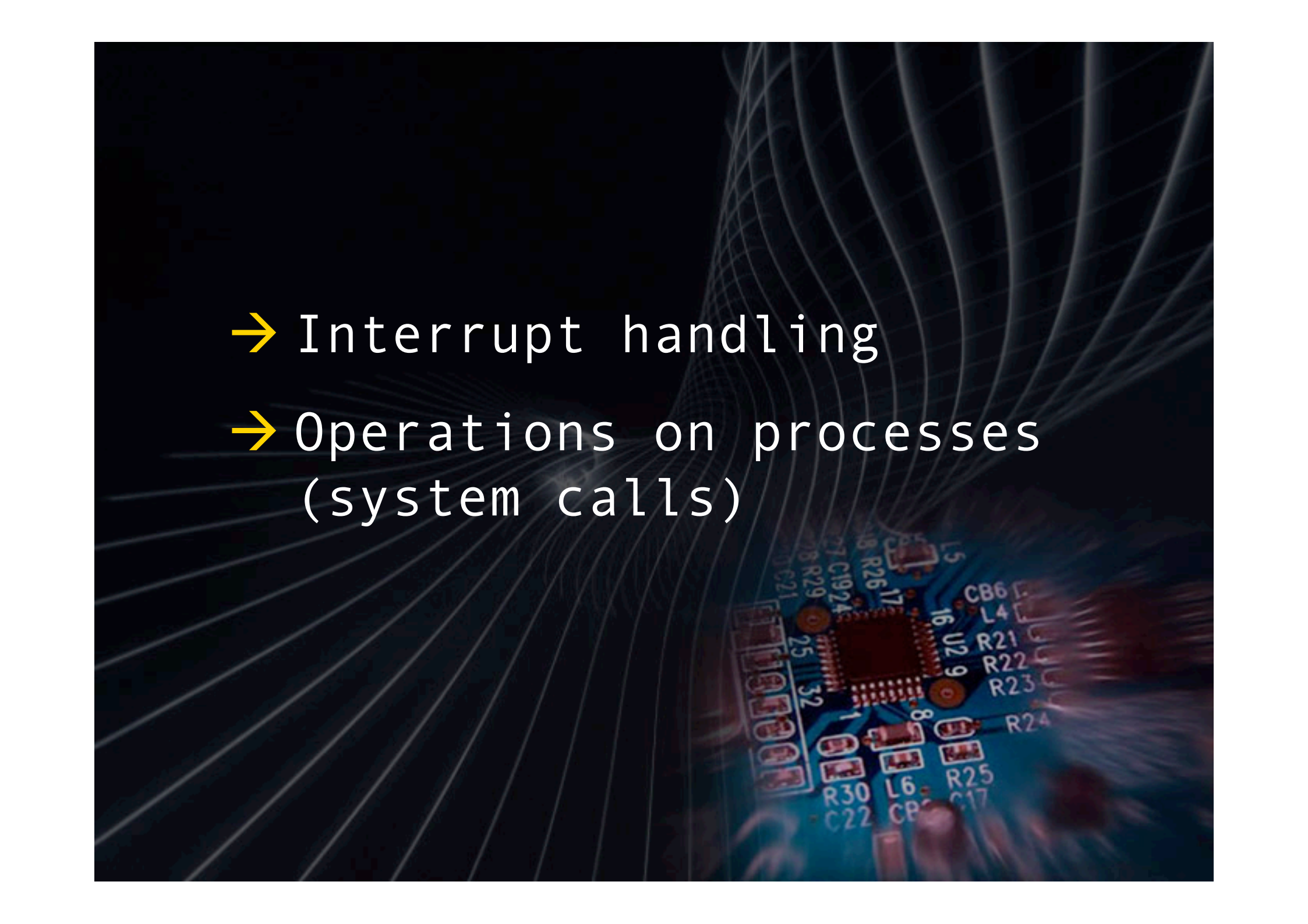
→ semaphores and blocked-process-queues

## 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

A hand is shown typing on a laptop keyboard. The background is dark blue with white, wavy, abstract lines that create a sense of motion or data flow. The text is overlaid on the image.

→ Internal interrupts or traps triggered by the executing process

## Example

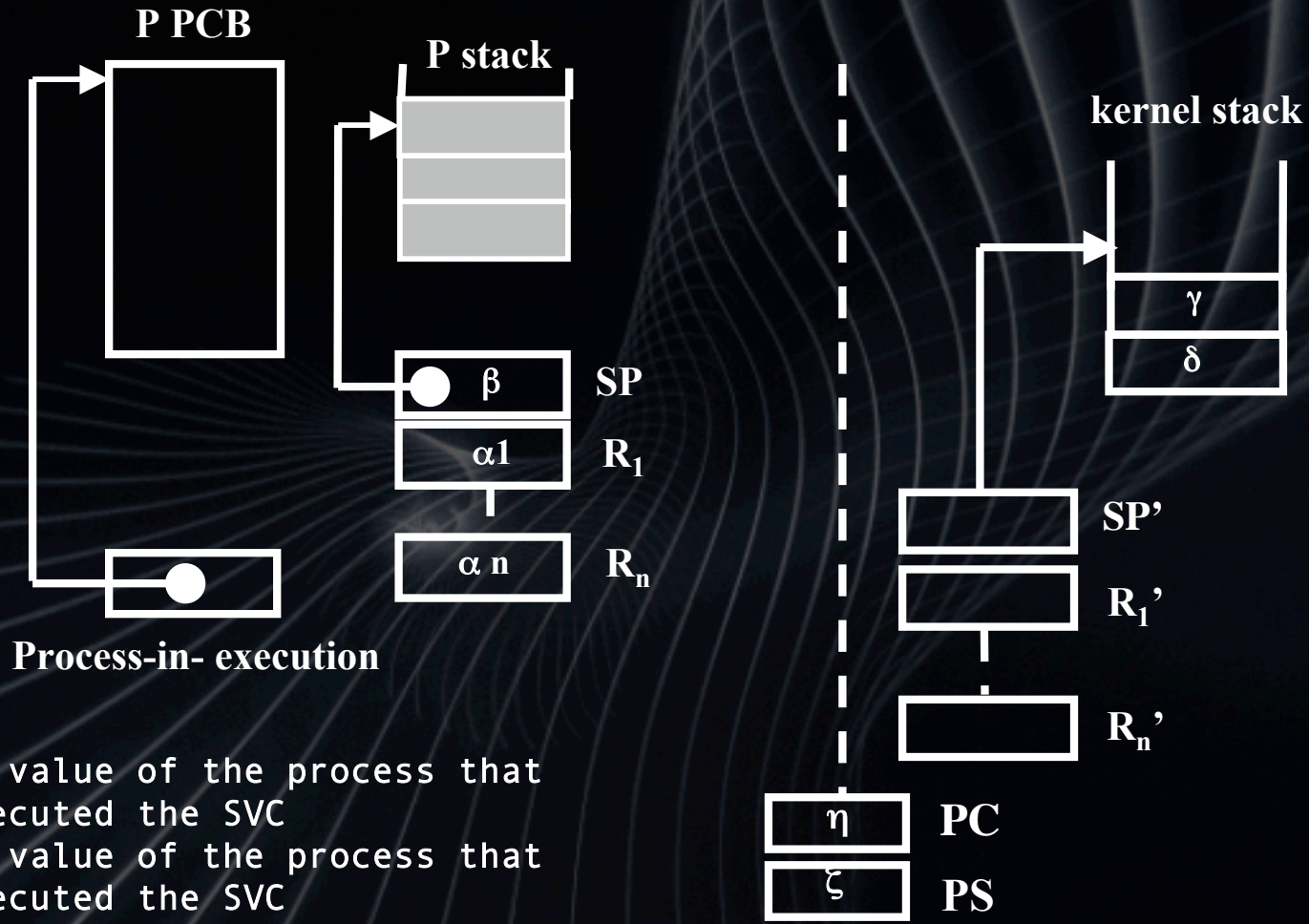
→ A **stack** is associated to any process



## Example

→ Double set of general registers  $R_1, R_2, \dots, R_n$  and  $R'_1, R'_2, \dots, R'_n$  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
- $\epsilon$  interrupt handler address
- z kernel PSW

