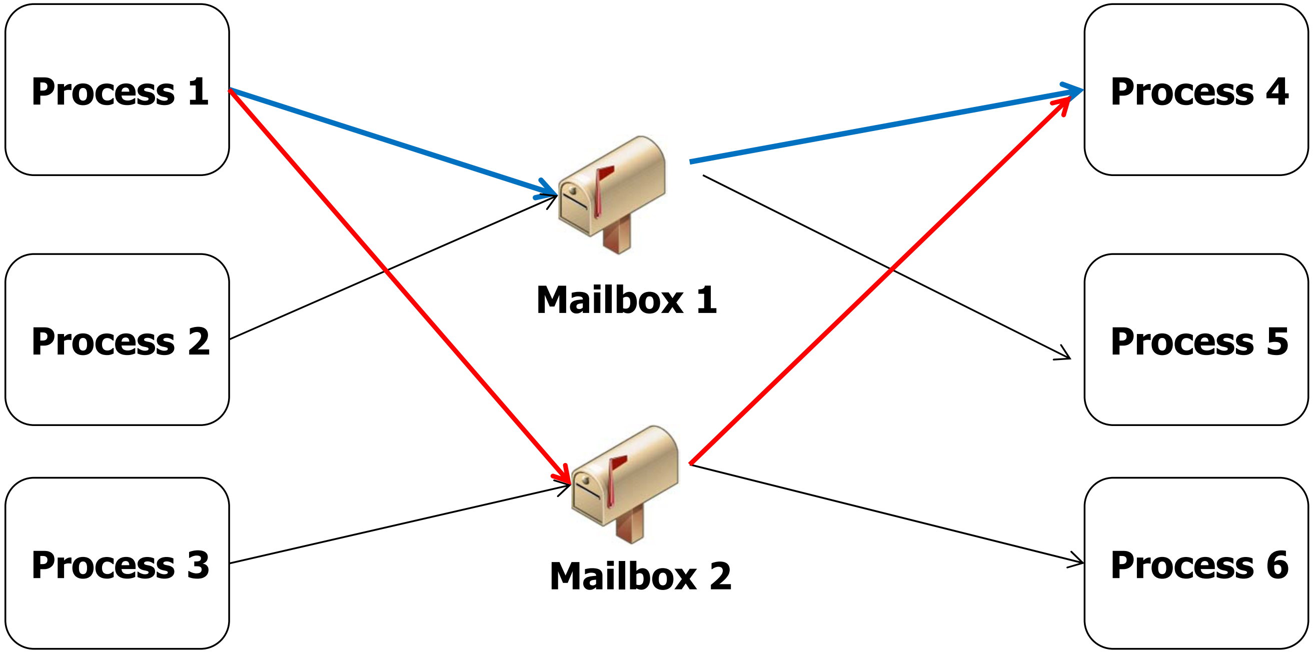
- Direct communication vs. Indirect communication
  - Direct communication
    - 1. Links are established automatically
    - 2. Between each pair there exists exactly one link
    - Shortcoming
      - Can we know the name in advance?
    - Example
      - Pipe mechanism
  - Indirect communication
    - 1. Link established only if processes share a common mailbox
    - 2. A link may be associated with many processes
    - 3. Each pair of processes may share several communication links
    - 4. Link may be unidirectional or bi-directional

- Implementation issues (indirect communication)
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can be there between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

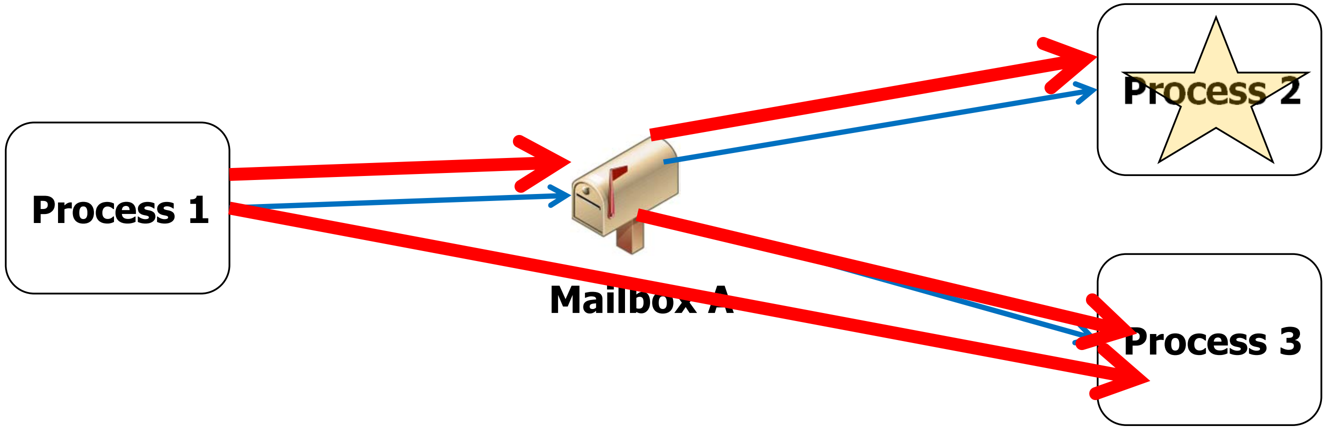


1. A link may be associated with many processes

2. Each pair of processes may share several communication links

- Indirect communication operations
  - 1. create a new mailbox
  - 2. send and receive messages through mailbox
  - 3. destroy a mailbox
  - Primitives are defined as:
    - **send**(*A, message*) – send a message to mailbox A
    - **receive**(*A, message*) – receive a message from mailbox A

- Mailbox sharing issues
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message (if P_2 , P_3 tries)?
- Solutions
  - Broadcasting
  - Allow only one process at a time to execute a receive operation
  - Allow the sending process to select receiver

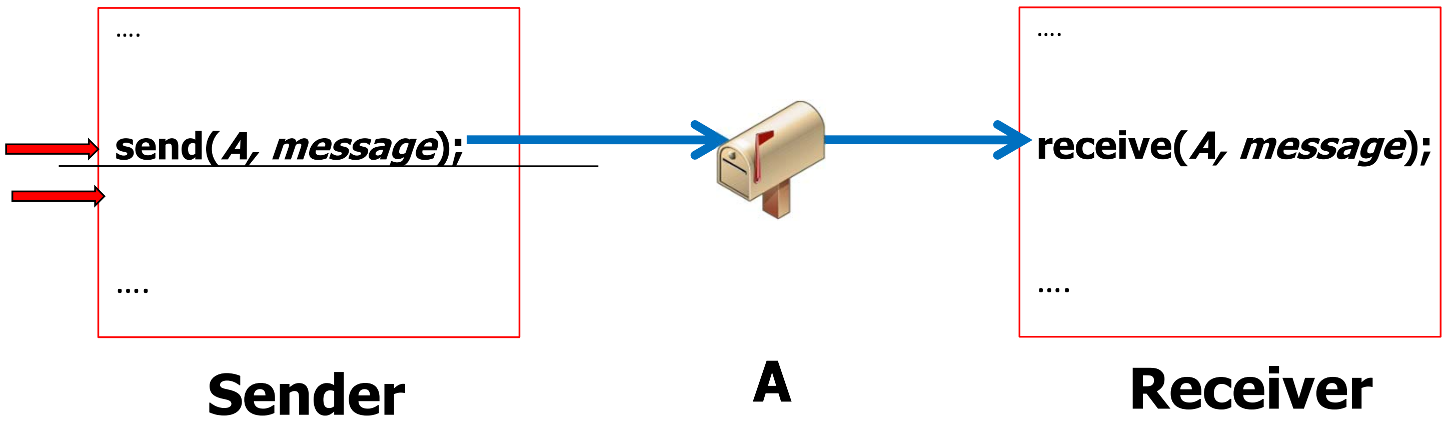


1. Broadcasting

2. Allow only one process at a time to receive a message

3. Allow the sending process to select the receiver

- Mailbox synchronization
  - Message passing may be either blocking or non-blocking
  - **Blocking** is considered **synchronous**
    - **Blocking send** has the sender block until the message is received by mailbox
    - **Blocking receive** has the receiver block until a message is available
  - **Non-blocking** is considered **asynchronous**
    - **Non-blocking** send has the sender send the message and continue
    - **Non-blocking** receive has the receiver receive a valid message or null



....

send(*A, message*);

....

Sender

A

....

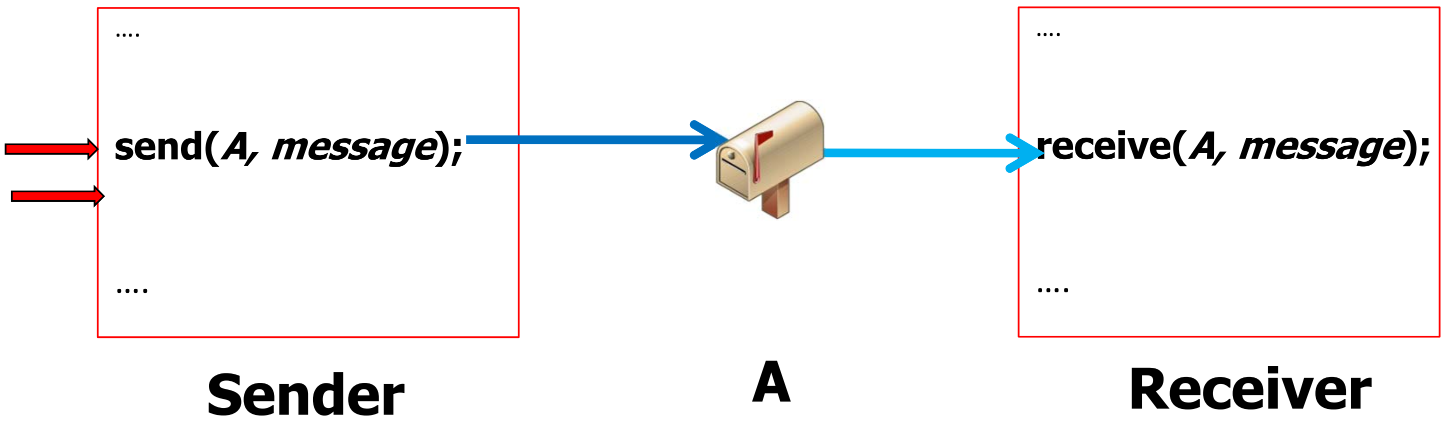
receive(*A, message*);

....

Receiver

Blocking send: The sender blocks until the message is received by mailbox or by the receiver

➡ **Program counter**



....

send(*A, message*);

....

Sender

A

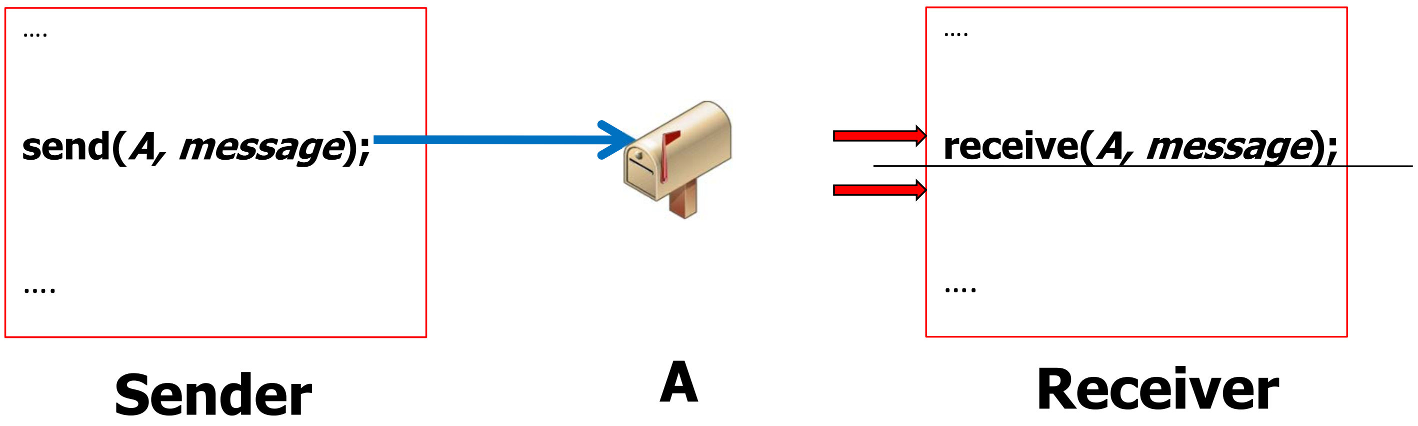
....

receive(*A, message*);

....

Receiver

NonBlocking send: The sender sends the message and continue its operation



....

send(*A, message*);

....

Sender

A

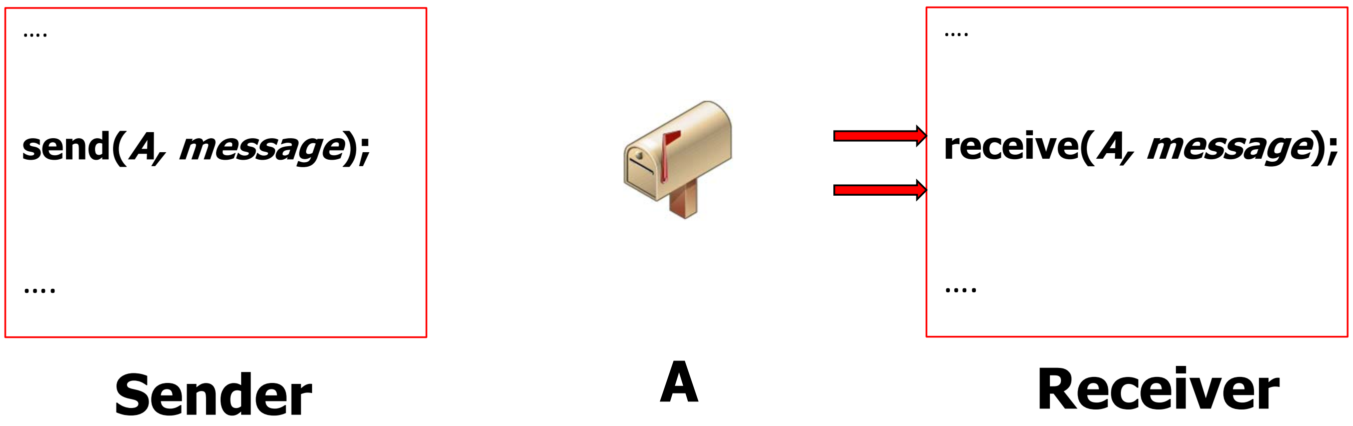
....

receive(*A, message*);

....

Receiver

Blocking receive: The receiver blocks until the message is available



….

send(*A, message*);

….

Sender

A

….

receive(*A, message*);

….

Receiver

Nonblocking receive: The receiver does not block until the message is received by mailbox

- Queue of messages attached to the link; implemented in one of three ways

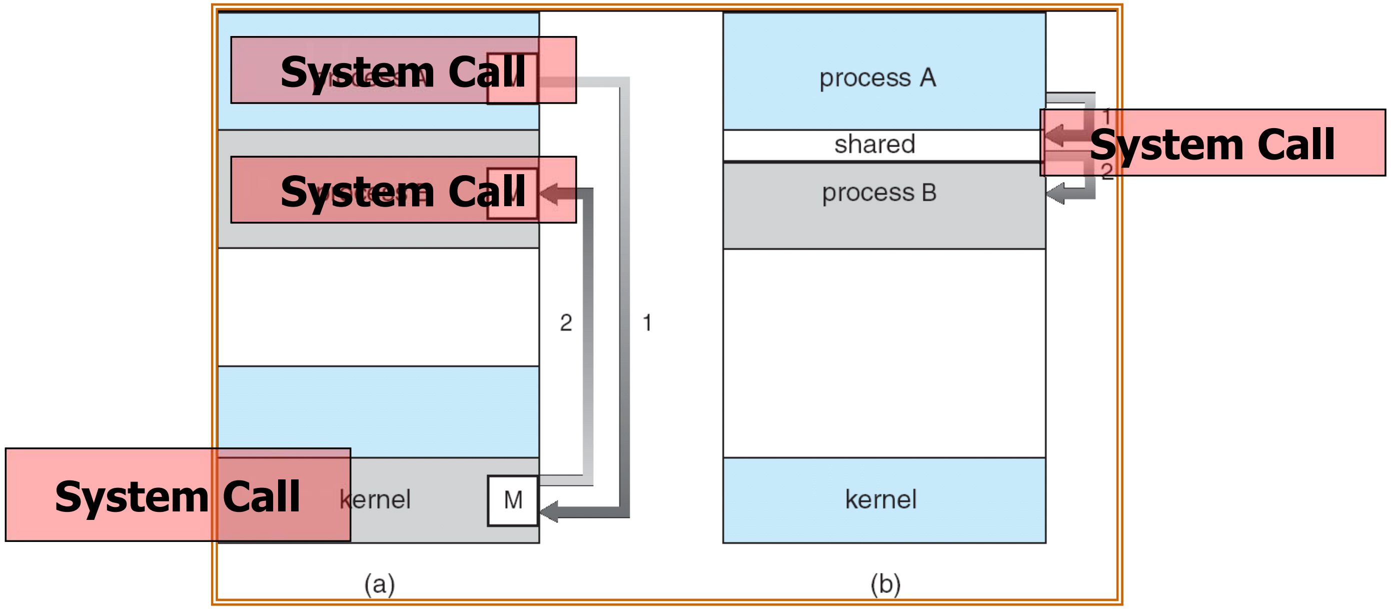
NO buffering

1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)

Automatic buffering

2. Bounded capacity – finite length of *n* messages
Sender must wait if link full
3. Unbounded capacity – infinite length
Sender never waits

- IPC comparison
  - 1. Message passing
  - 2. Shared memory



Message passing **Shared memory**

- Shared memory vs. message passing
  - Message passing
    - Useful for inter-computer communication
    - Useful for exchanging smaller amounts of data
    - Typically implemented using system calls so it is time-consuming
    - Easier for programming
  - Shared memory
    - Fast
      - System calls are required only to establish shared memory regions
      - It can be done at memory speeds
    - Some kind of protection mechanisms are needed