# Processes

- What is a process?
- Structure of a process.
- Process state.
- Schedulers and Scheduling Queues.
- Context switching.
- Process creation and termination.
- Inter Process Communication.

Dr. Manmath N. Sahoo
Dept. of CSE, NIT Rourkela

# Process

- Process is a program that has initiated its execution.
- A program is a **passive** entity; whereas a process is an **active** entity.

# Process structure

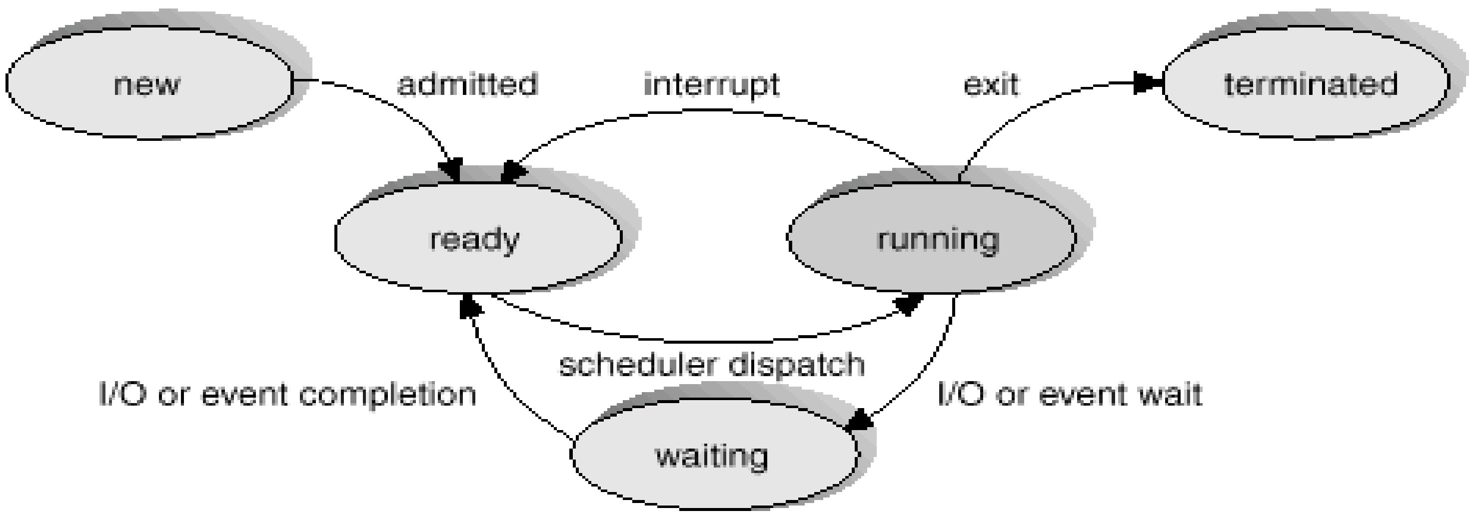
A process includes:

- program section - contains a copy of the machine code instructions
- user data section - to hold variable data values
- system data section - to hold
  - process context info when process interrupted - program counter, processor registers, etc and
  - system information about resources allocated to process, etc.
- user and system data sections obviously may be different from one run of a process to another

# Process State

As a process executes, it changes state.

- **new**: The process is created by OS
- **ready**: The process is waiting to be assigned to a processor.
- **running**: Instructions are being executed on processor
- **waiting**: The process is waiting for some event to occur
- **terminated**: The process has finished execution.



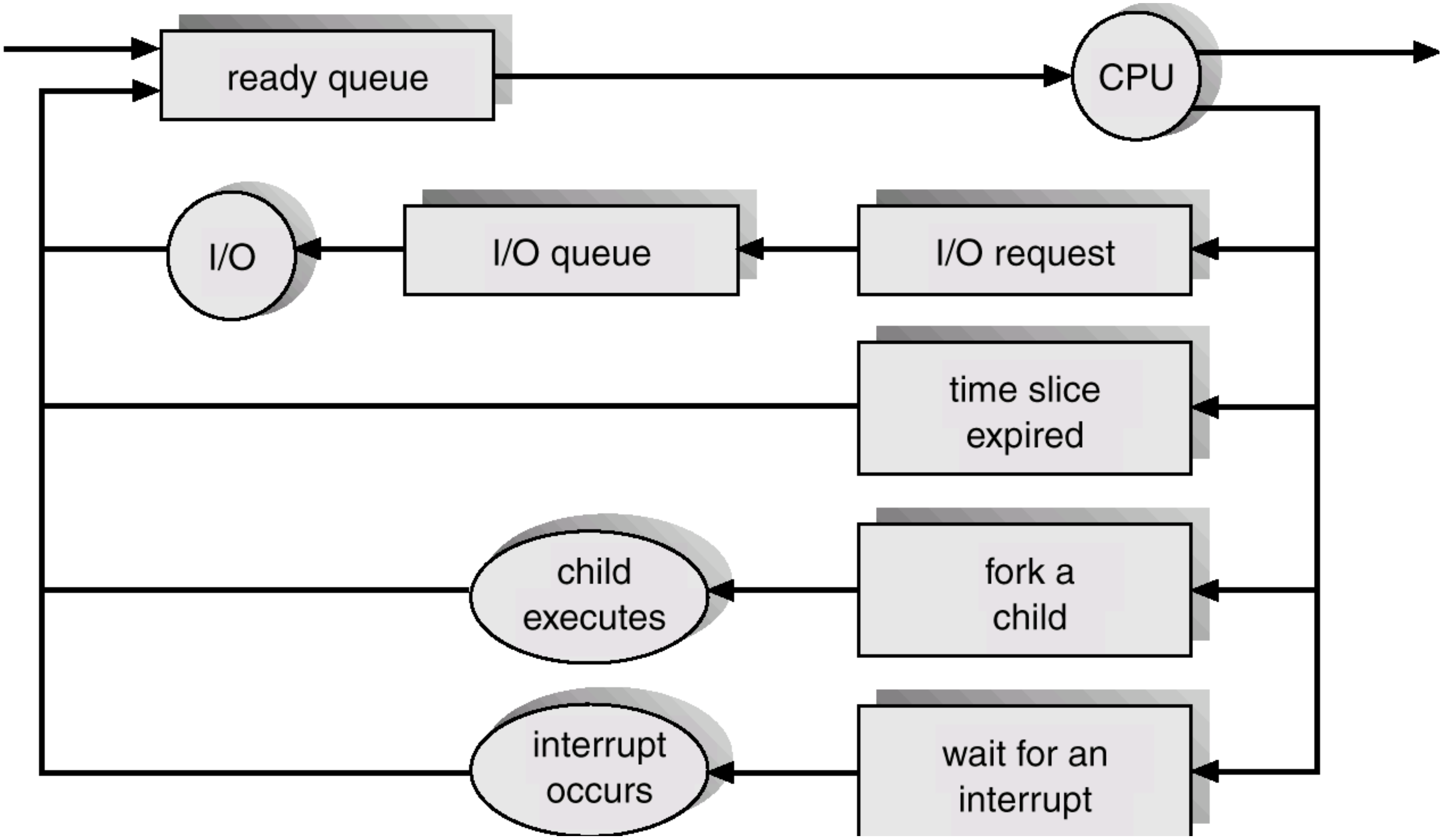
# Process Control Block (PCB)

- PCB - data structure that contains Information associated with each process:
  - Process ID
  - Process State
  - Program Counter
  - Register Contents
  - Starting address of the process
  - Size of the process
  - Pointer to Child Process
  - Pointer to Next PCB (Sibling Pointer)
  - Resource Pointer

# Process Scheduling Queues

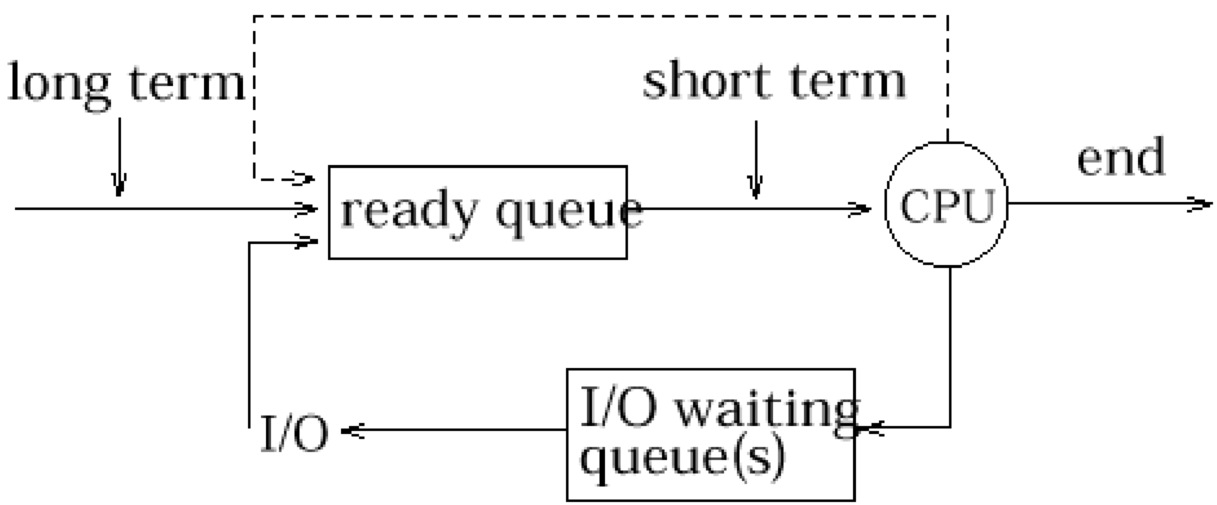
- **Job queue** – Set of all processes just created by LTS (new state processes).
- **Ready queue** – set of all processes that are in main memory, ready to execute (ready state processes).
- **Device queues** – set of processes waiting to use an I/O device (waiting state processes).

# Queueing Diagram



# Schedulers

- **Long-term scheduler (LTS)** (or job scheduler) – selects which processes should be brought into the ready queue.
- **Short-term scheduler (STS)** (or CPU scheduler) – selects which process should be executed next on CPU.



# Schedulers

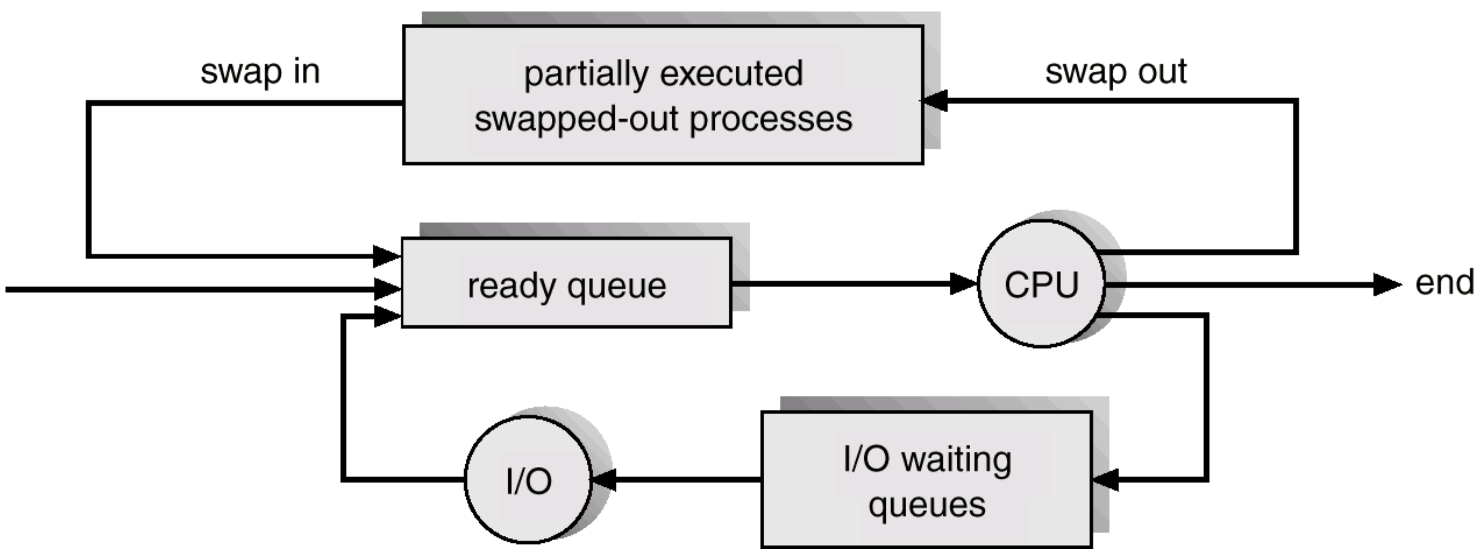
- Short-term scheduler is invoked very frequently (milliseconds) -> (must be fast).
- Long-term scheduler is invoked very infrequently (seconds, minutes) -> (may be slow).
- The long-term scheduler controls the degree of multiprogramming - this is the number of processes in the system that can be scheduled onto the CPU I.e. number of processes competing for CPU as a resource.

# Schedulers

- Long-term schedule should choose a good process mix; otherwise it may lead to either low CPU utilization (if I/O bound processes are selected only) or low Device utilization (if CPU bound processes are selected only).
  - I/O-bound process – have small amount of computation before it needs to do some I/O; many I/O requests -Typical interactive user programs.
  - CPU-bound process – large amount of computation before it needs to do some I/O; few I/O requests - Programs that require sustained periods of calculation e.g. modelling applications

# Schedulers

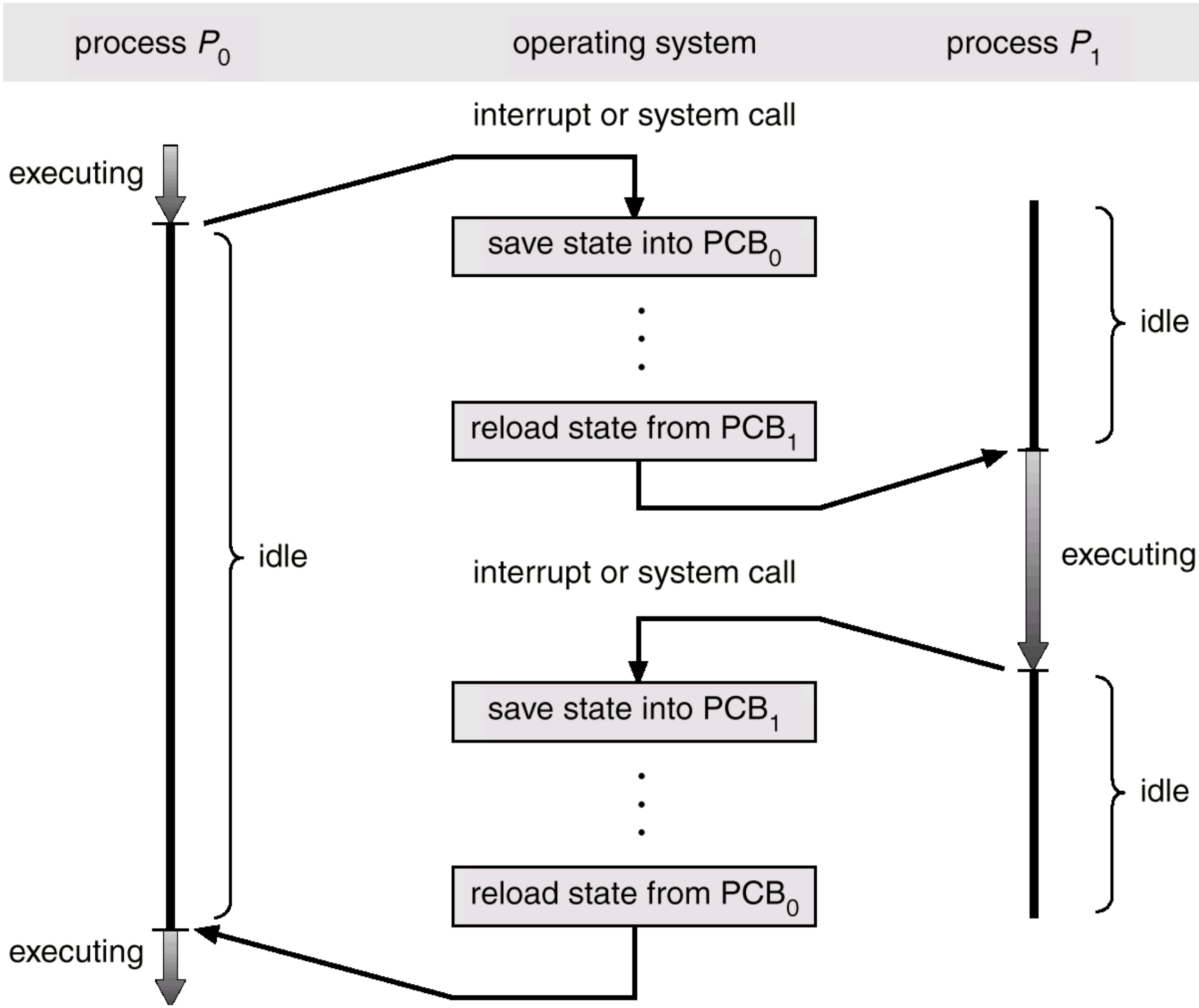
- Some OS have a light LTS; and a **Medium Term Scheduler (MTS)**.
- If degree of multiprogramming becomes high then total context switch overhead will be more.
- MTS can temporarily swap some processes out to disk and bring them back later.



# Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.
- Context-switch time is overhead; the system does no useful work while switching.

# CPU switched between processes



# Process Creation

- When a process is created it is created by another process known as the parent process - new process created is the child process of the parent. New process may in turn create other processes, forming a family tree of processes.
- There must be a first process (**init** process) - this is process that is created at bootup of the OS and is part of OS.

# Process Creation (Cont.)

- Resource sharing between parent and child processes - options include:
  - Parent and children share all resources, or
  - Children share subset of parent's resources, or
  - Parent and child share no resources.
- Execution options:
  - Parent and children execute concurrently, or
  - Parent waits until children terminate.

# Process Termination

- Process executes last statement and asks the operating system to delete it (**exit** system call). Then
  - Return data from child to parent.
  - Process' resources are deallocated by operating system.
- Parent may terminate execution of children processes (**abort**). Reasons for this could be:
  - Task assigned to child is no longer required.
  - Parent is exiting and O/S does not allow children to continue if their parent terminates – **cascaded termination**.

# Cooperating Processes

- Process is **independent** if it cannot legally affect or be affected by the execution of another process.
- Process is **cooperating** if it can legally affect or be affected by the execution of another process.

# Why Cooperating Processes?

- Processes may need to share data
  - More than one process reading/writing the same data (a shared file, a database record,…)
  - Output of one process being used by another
- Ordering executions of multiple processes may be needed to ensure correctness
  - Process X should not do something before process Y does something etc.
  - Need mechanisms to pass control signals between processes
- Modularity - dividing system functions into separate processes that then talk to each other
- Computation speed-up - but only if >1 processor

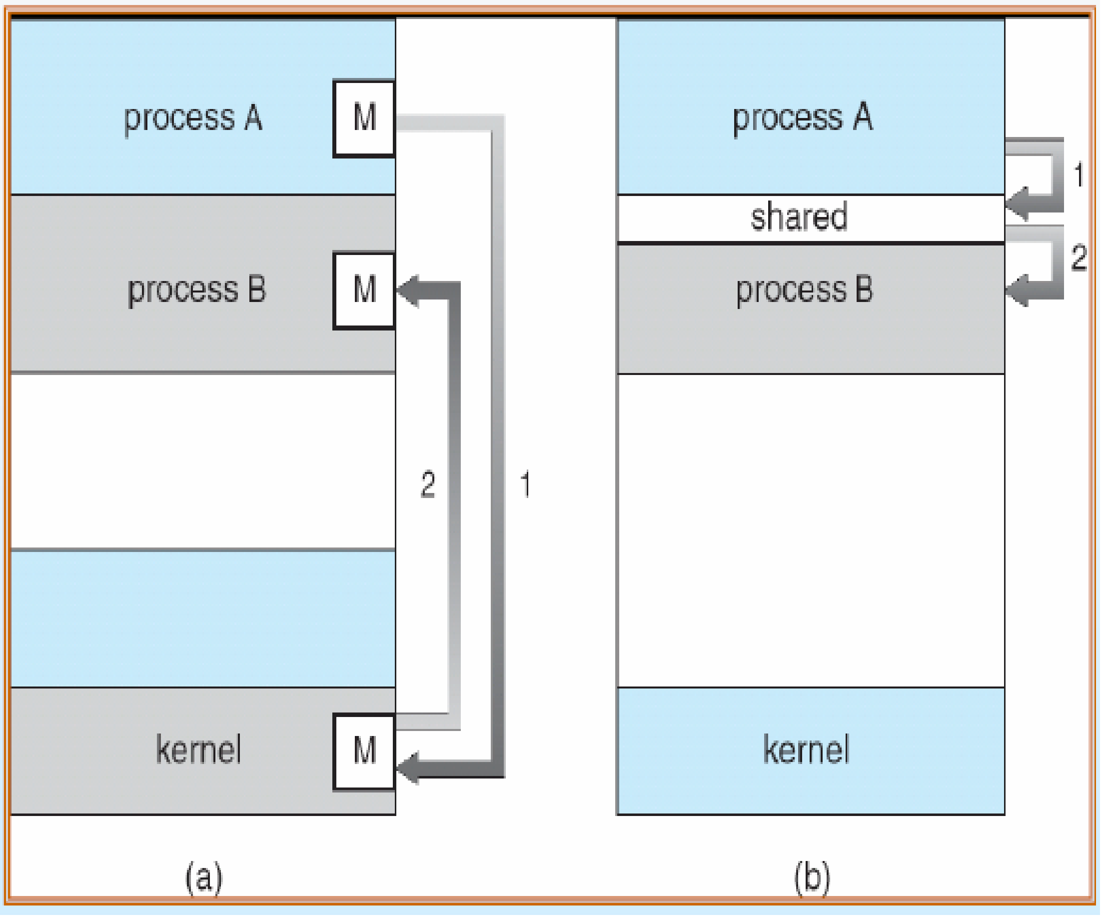
# Inter-Process Communication (IPC)

Fundamental types of IPC

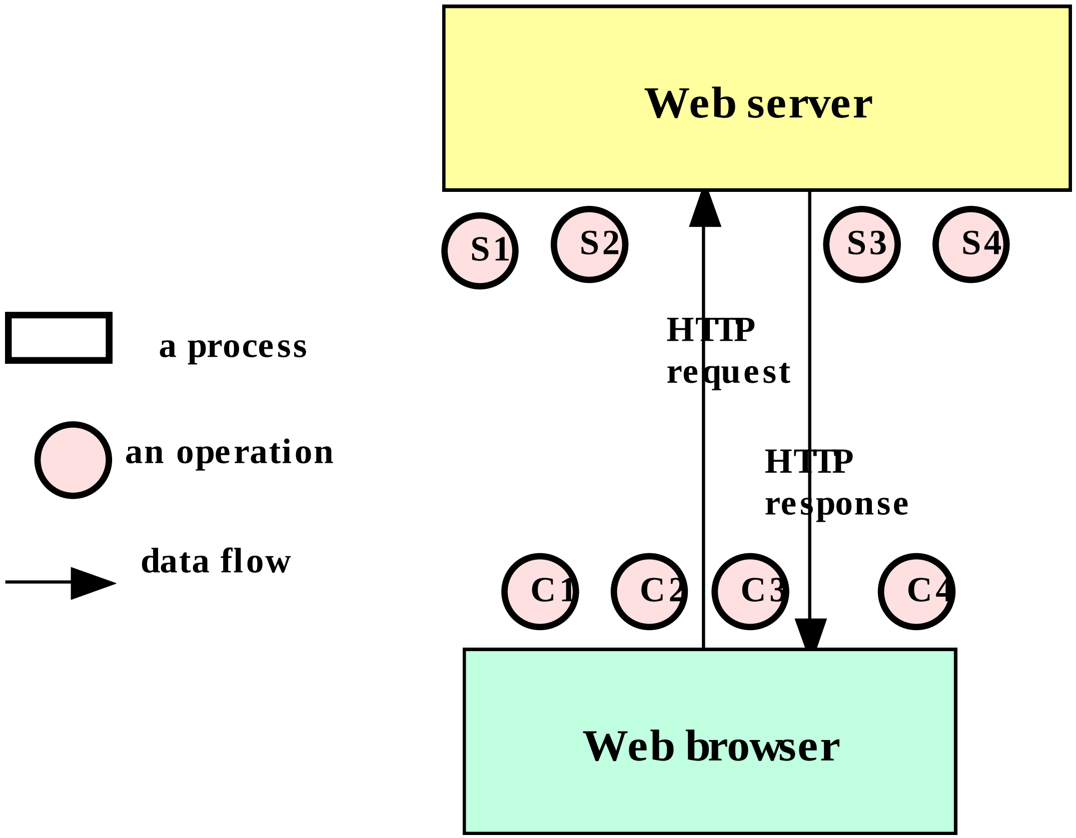
- Message passing
  - P and Q exchange messages
- Shared memory
  - P writes into a shared location, Q reads from it and vice-versa

# Inter-Process Communication (IPC)

(a) Message Passing (b) Shared Memory



# Message Passing System




operations:
S1: accept connection
S2: receive (request)
S3: send (response)
S3: disconnect
C1: make connection
C2: send (request)
C3: receive (response)
C4: disconnect

# Message Passing System

- Direct vs. Indirect
- Symmetric vs. Asymmetric
- Synchronous vs. Asynchronous

# Message Passing System: Direct

**Primitives:**

- **Connect (sender address, receiver address)**, for connection-oriented communication.
- **Send ( [receiver], message)** e.g., Send(P, msg)
- **Receive ( [sender], message storage object)** e.g., Receive(Q,msg)
- **Disconnect (connection identifier),** for connection-oriented communication.

# Message Passing System: Direct

- **Symmetric** – Both the sender and receiver have to explicitly name each other.
- **Asymmetric** – Only the sender has to name the receiver, but the receiver doesn't need to name the sender.
  - Facilitates one-to-many communication.
  - Send(P,msg); - send message to P.
  - Receive(id,msg); - receive message from any process. "**id**" will be set to the name of the process with which communication has taken place.

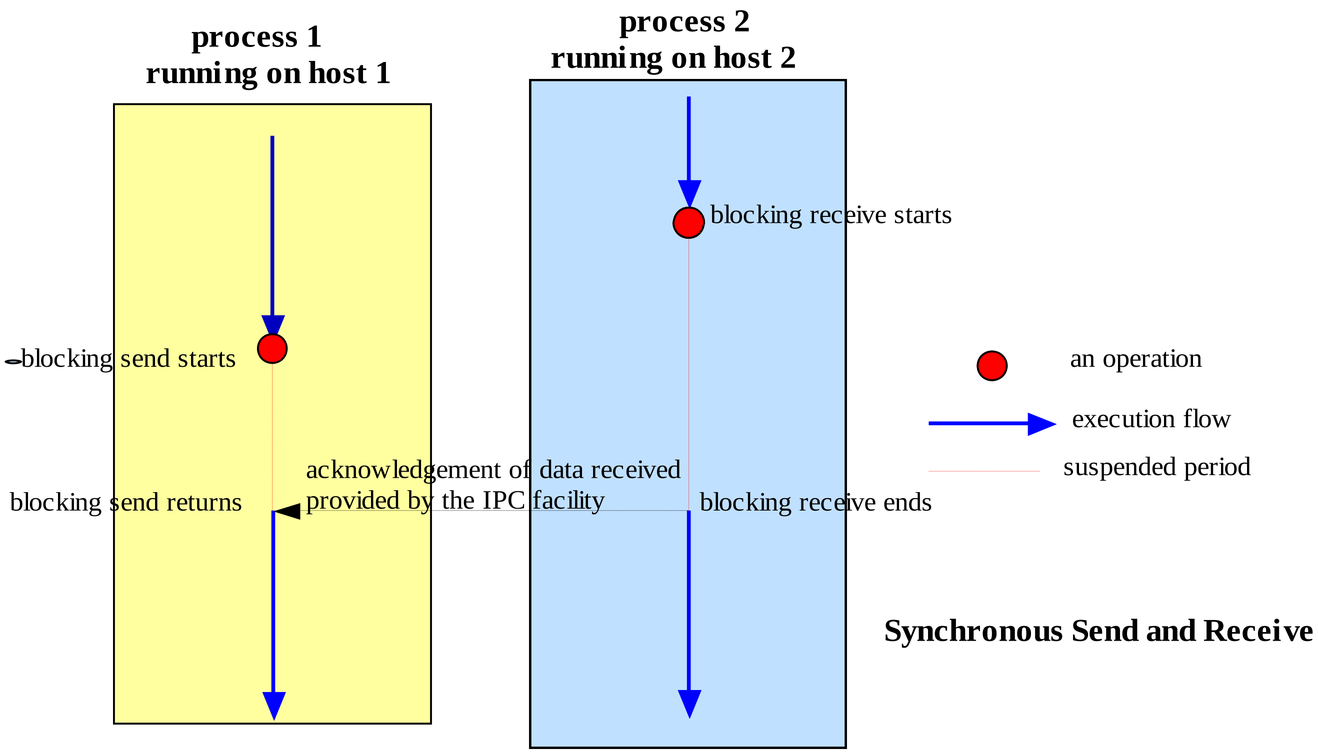
# Message Passing System: Indirect

- The messages are sent and received to/from **mailboxes**.
- Each mailbox has a unique identifier.
- Two processes can communicate with each other iff they have a shared mailbox.

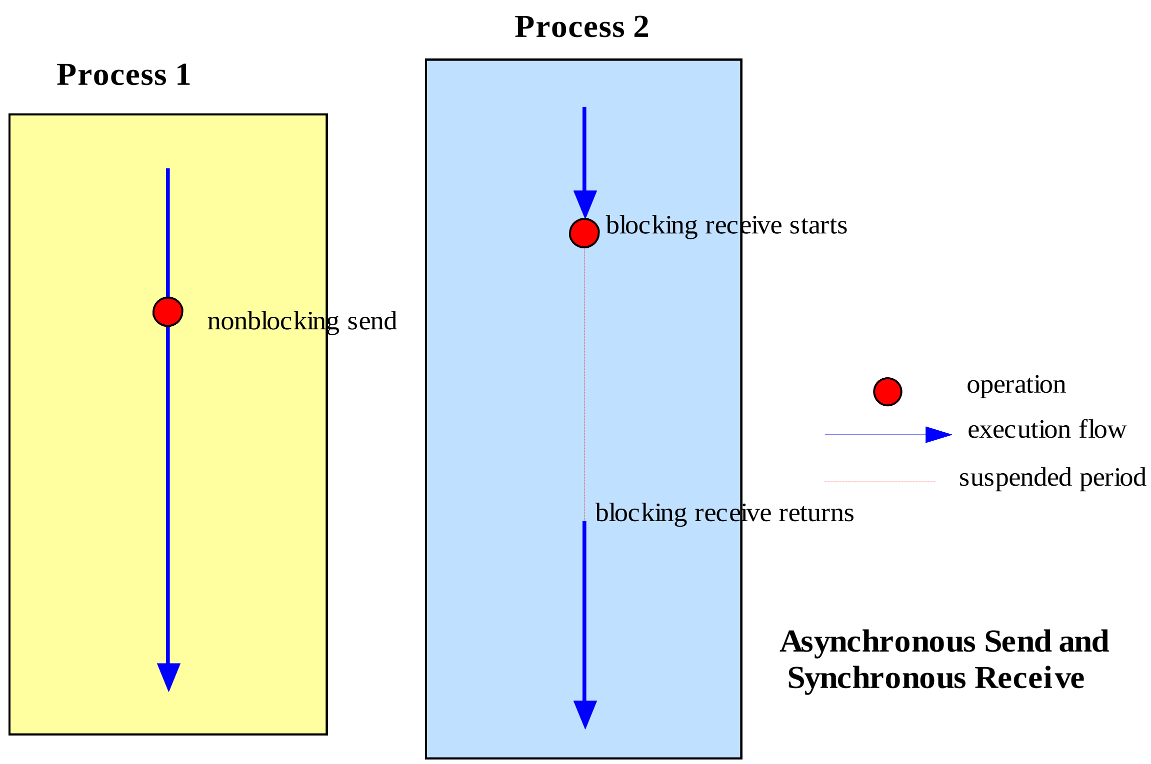
# Message Passing System: Synchronous Vs. Asynchronous

- Synchronous – After initiating a send/receive operation, the process waits for the acknowledgement (the process is blocked)
- Asynchronous – After initiating a send/receive operation, the process continues its execution

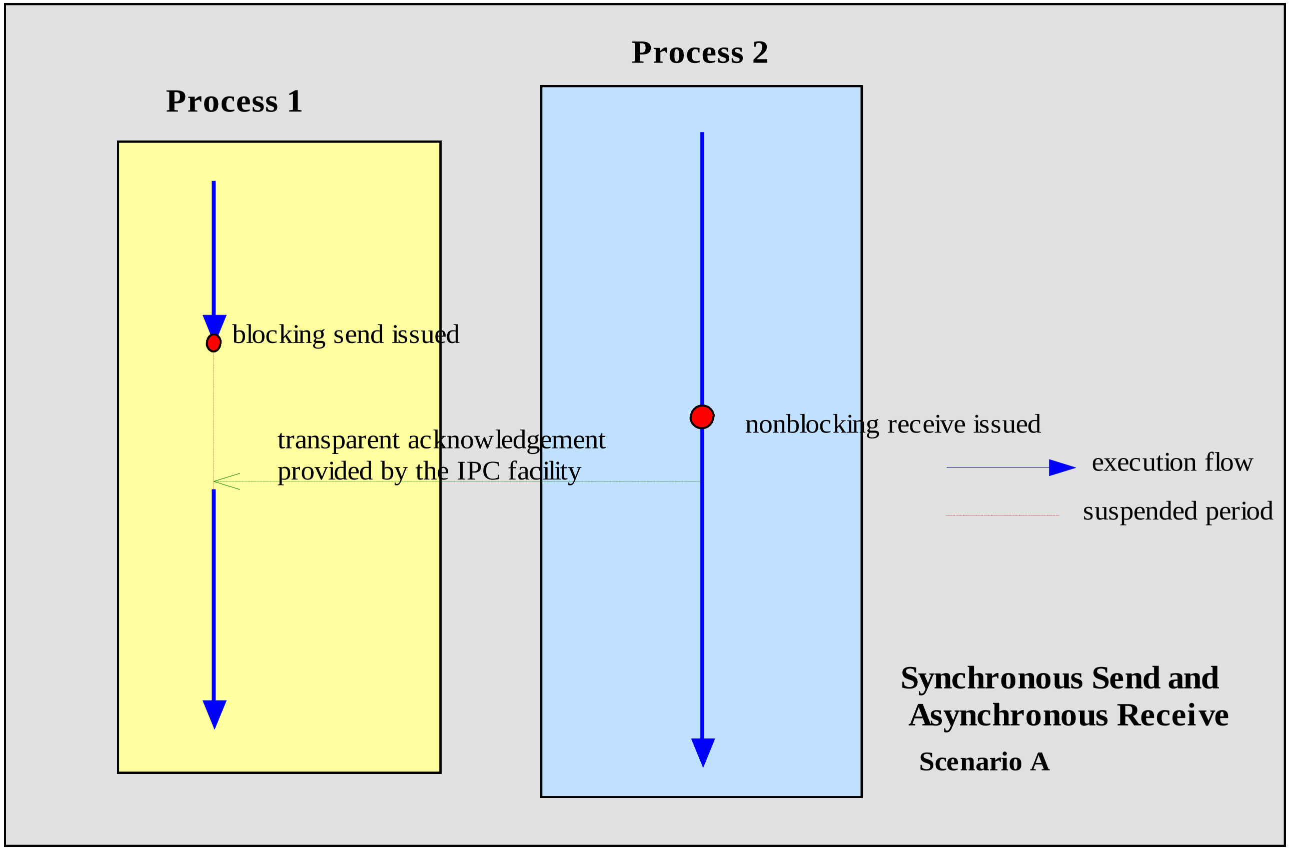
# Synchronous send and receive



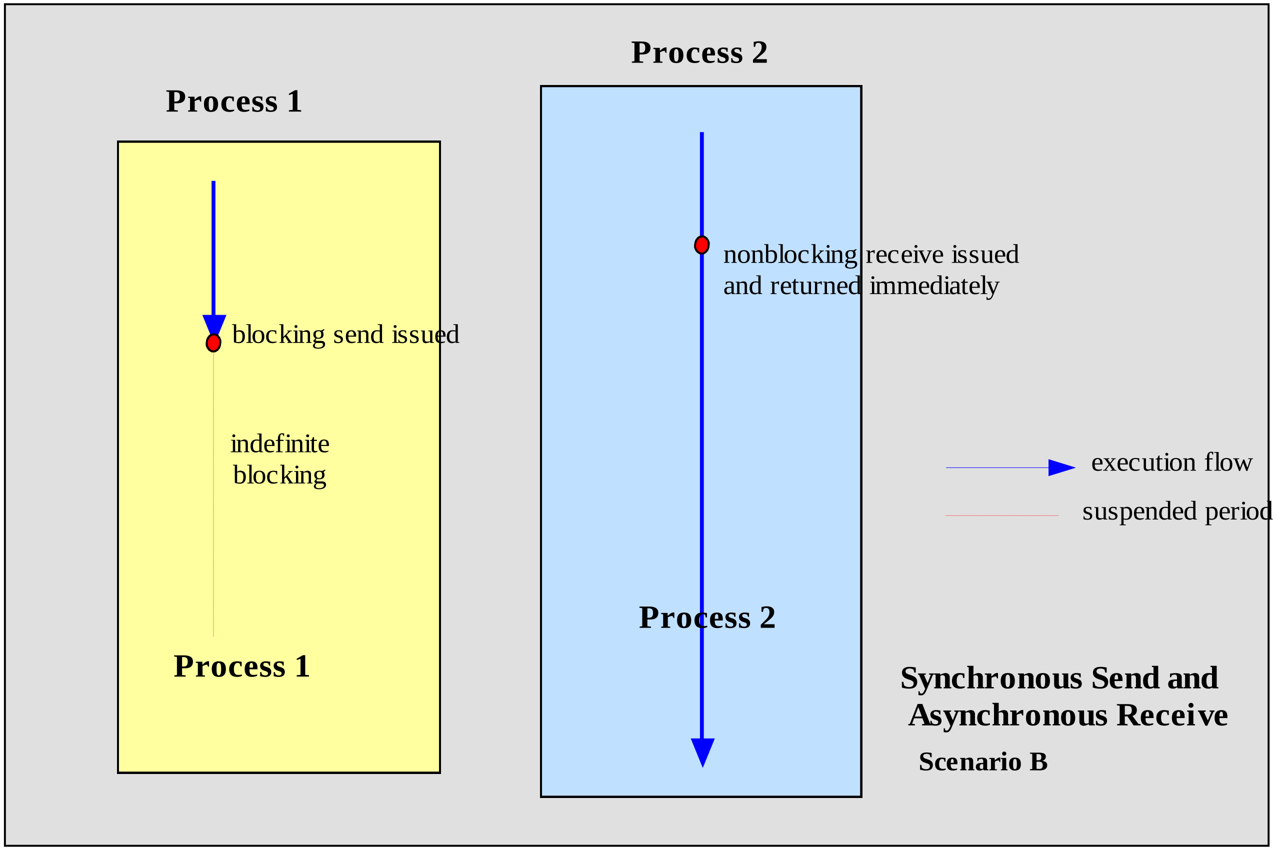
# Asynchronous send and synchronous receive



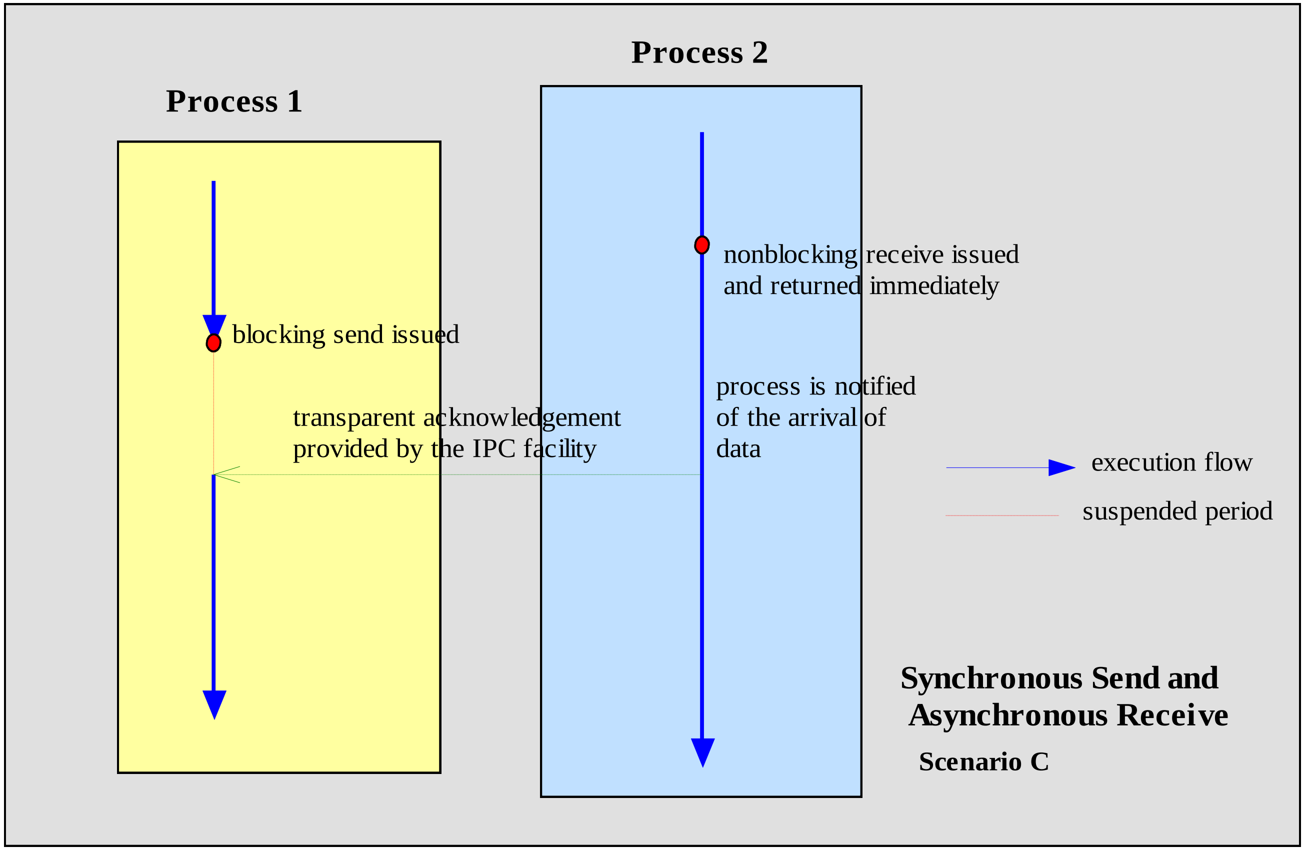
# Synchronous send and Async. Receive - A



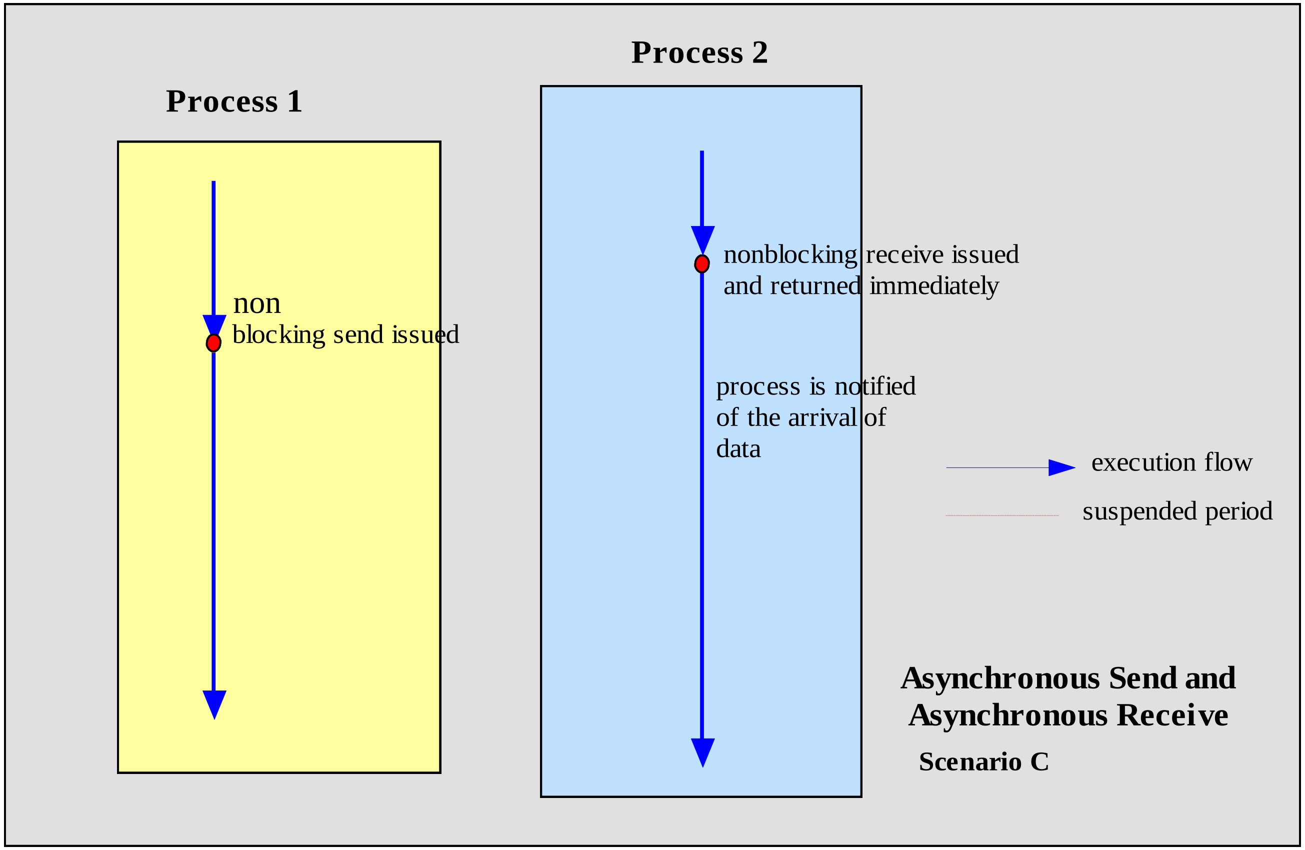
# Synchronous send and Async. Receive - B



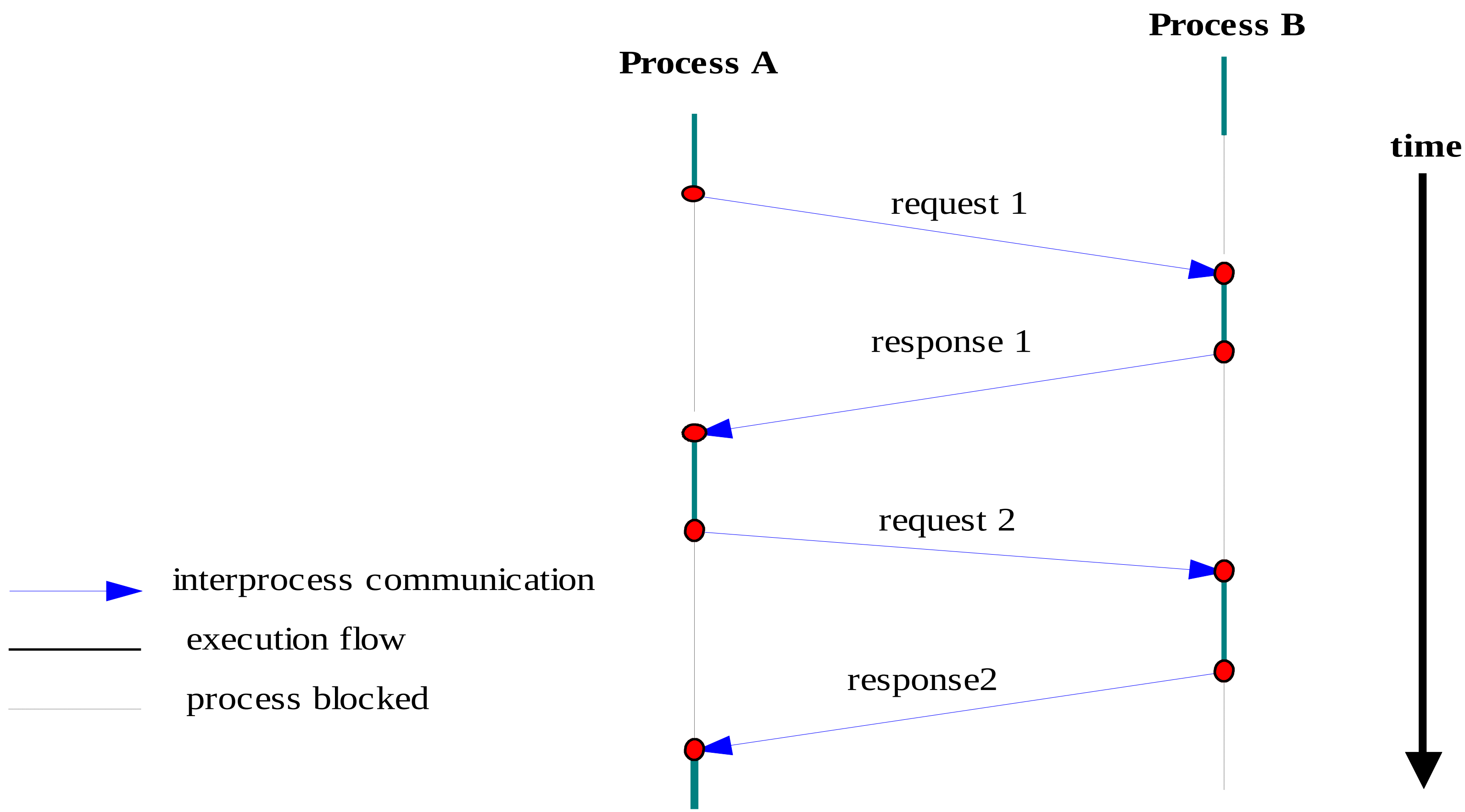
# Synchronous send and Async. Receive - C



# Asynchronous send and Asynchronous receive - C



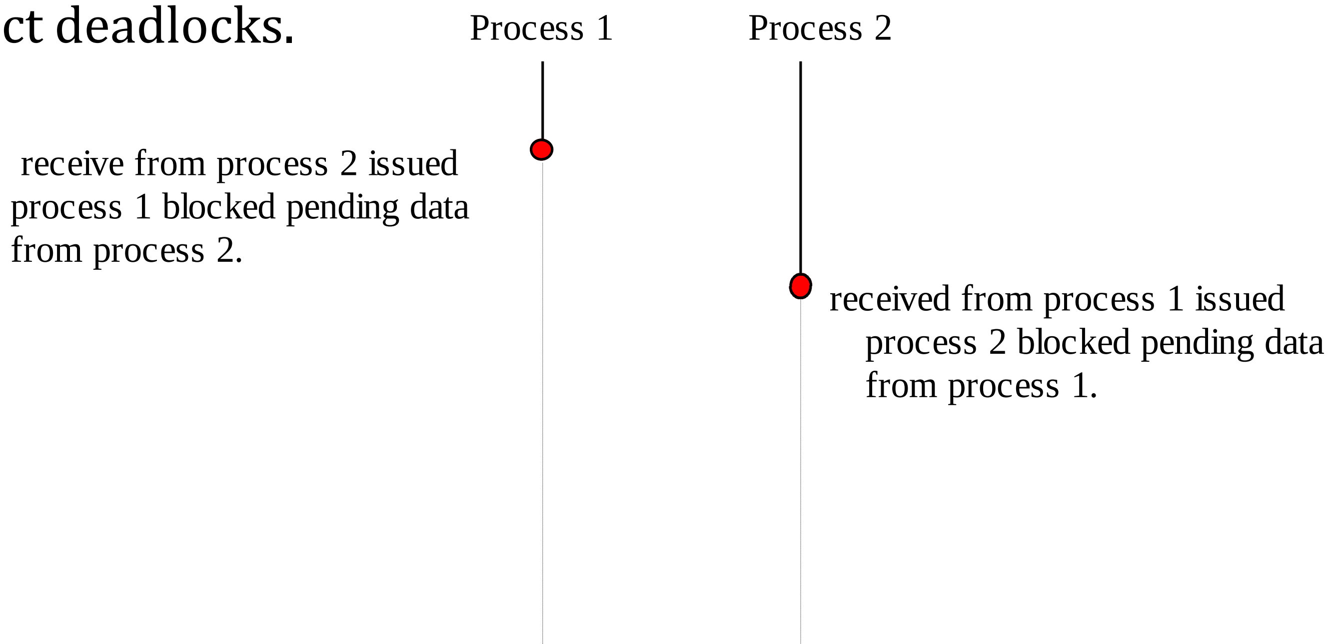
# Event diagram




**Event diagram for a protocol**

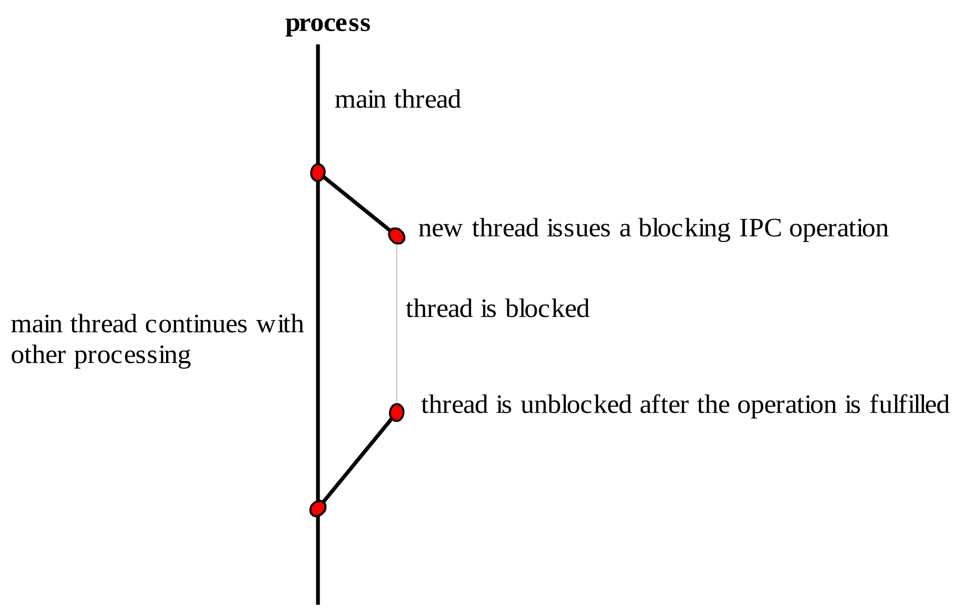
# Blocking, deadlock, and timeouts

- Blocking operations issued in the wrong sequence can cause deadlocks.
- Deadlocks should be avoided. Alternatively, timeout can be used to detect deadlocks.



# Using threads for asynchronous IPC

- If only blocking operation is provided for send and/or receive, then it is the programmer's responsibility to using child processes or threads if asynchronous operations are desired.



# Buffering during message passing

- Zero capacity
  - No waiting queue/buffer
  - Sender must wait for receiver, after generating one item
- Bounded capacity
  - Buffer has a fixed length N.
- Unbounded capacity
  - No practical limit on the size of the buffer

# Process: Exception Conditions

- Process Termination
  - Sender Terminates: If the receiver has initiated a blocking receive then the OS either terminates the receiver or notifies it that Sender has terminated.
  - Receiver Terminates: If the sender has initiated a blocking send then the OS either terminates the Sender or notifies it that Receiver has terminated.
- Lost Messages
  - The OS is responsible for detecting this event and for resending the message.
  - The Sender is responsible for detecting this event and for resending it.
  - The OS detects the event and notifies the Sender about it.
- Scrambled Messages
  - Due to noise in the channel
  - Handled in the similar ways as lost messages.