

BMCS3003 Distributed Systems and Parallel Computing

L01 - Introduction to Distributed Systems

Presented by

Assoc Prof Ts Dr Tew Yiqi
May 2023



Table of contents

01

**What is a
Distributed System ?**

02

**What is a
Real-time System ?**

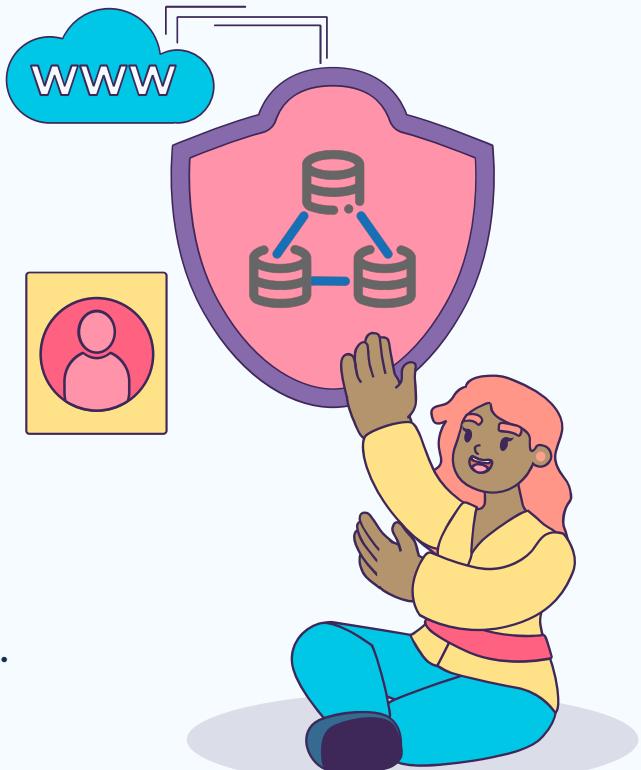
03

Operating System

01

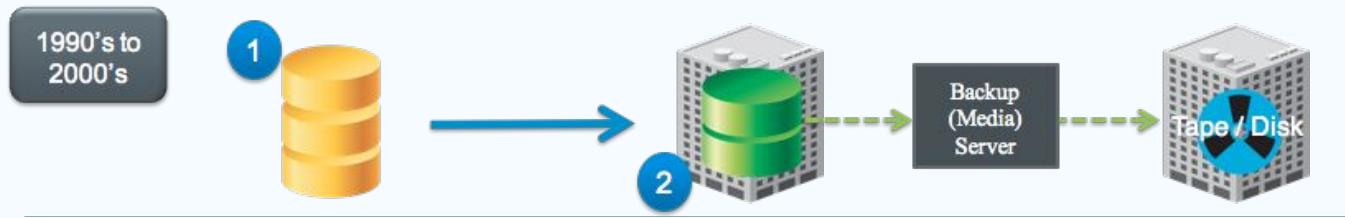
Distributed Systems

A computing environment in which various components are spread across multiple computers.



Tradisional vs Distributed

Tradisional databases



Distributed databases



Source: insidebigdata.com

Basic Concept

- A distributed computing system is one in which the computation is distributed / spread across multiple processing entities.
- Various aspects of the system can be distributed:



Database / data



Operating System



File System



Business Logic



Authentication



Workload

Resources allocation / load sharing

Benefits



Scalability

Continuously evolve in order to support the growing amount of work.



Reliability

Keep delivering its services even when one or several of its software / hardware components fail



Performance

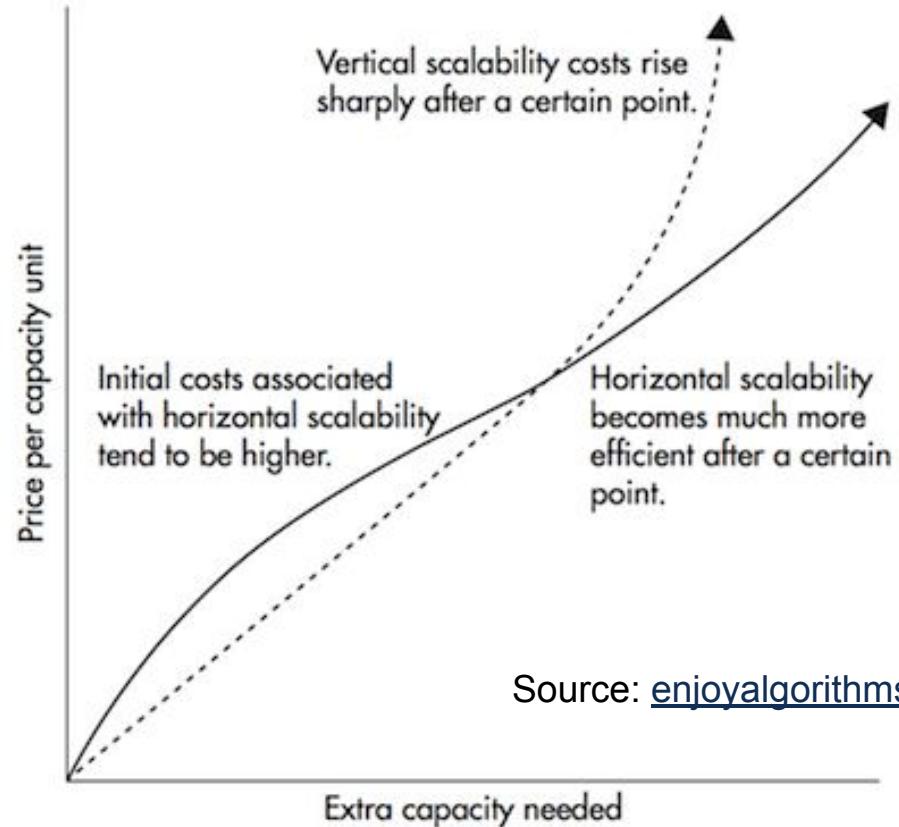
Availability to interact and coordinate actions to appear to the end-user as a single system



Geographical

Distributed processing and resources based on application specific requirements and user locality.

Benefits - Scalability

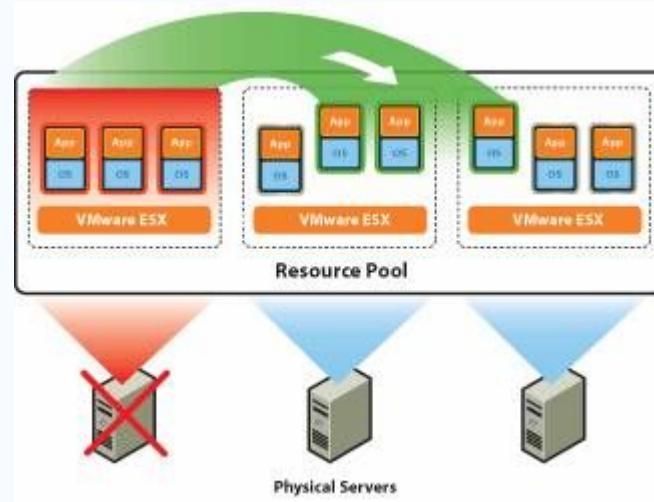


Challenges when building Distributed Systems



- Avoid single point of failure.
- Replication.
- Availability and performance.
- Resource naming, addressing and location of resources.
- Binding (mapping between parts of the system).

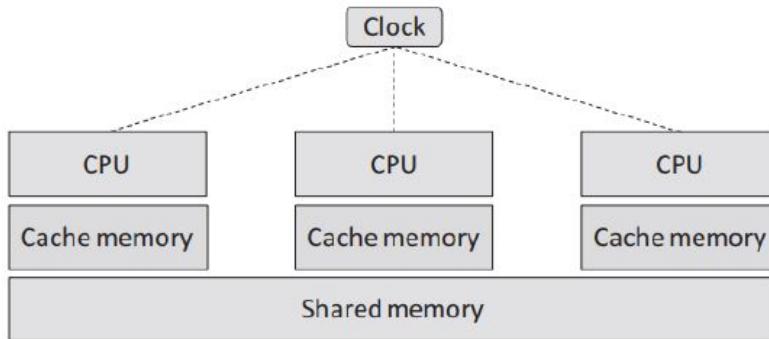
Benefits - Fault Tolerance



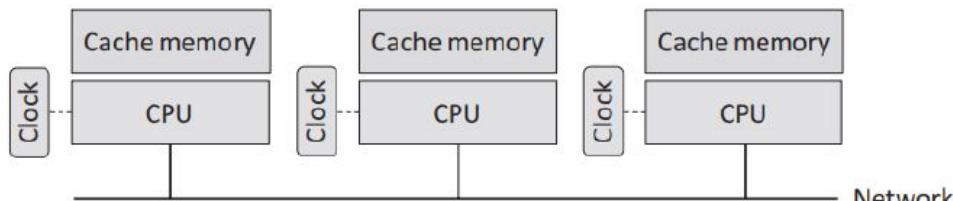
Case Study: Facebook's Maelstrom system

Tightly and loosely coupled hardware architectures

Recall the syllabus in
Computer Organisation and
Architecture /
Computer System Architecture /
Introduction to Computer System



(a) Tightly-coupled processors with private cache memories and shared main memory



(b) Loosely-coupled processors use the network to communicate

Tightly-coupled systems

The processor units are physically part of the same computer.

Processors are connected:

By a high-speed blackplane bus, or are on the same “motherboard”,
or in the same integrated circuit (Chip)



Specialised hardware:

- Fixed architecture (number of processors)
- Expensive
- Multi-core (typically 2 or 4 CPUs in PCs at present)
- Large scale (of order of 64 processors and greater) not common
(also referred to as parallel processing systems)

Shared clock - synchronisation is possible.

Shared memory - fast / reliable inter-processor communication.



Activity

What is the output of the program ?

```
#include "stdafx.h"
#include <iostream>
#include <omp.h>
using namespace std;

int main()
{
#pragma omp parallel
{
    cout << "Hello World\n";
}
return 0;
```

Loosely-coupled systems

The processor units are within separated computers.

The computers are connected by a network technology:

General purpose hardware:

- Cheap
- Abundant



Tightly-coupled systems (Challenges)

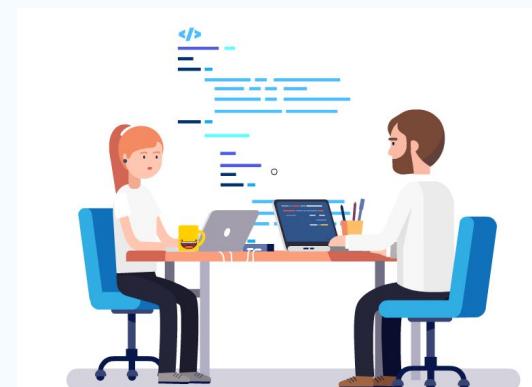
- Each computer has its own clock:
 - absolute synchronisation NOT possible.
 - 'loose' synchronisation is necessary.
- Computers have separate memory not suitable for inter-processor communication.
- The individual computers are Autonomous need some overall guidance.
- The individual computers are Heterogeneous different memory size, disk size, processor speed, hardware platform, operating system etc.



Motivation for Loosely-coupled Distributed Systems (1/2)

The Interest in distributed systems has grown because of:

- The need to share large amounts of data
- The availability of cheap workstations
- The need to share expensive peripherals
- The availability of cheap high speed networks



Motivation for Loosely-coupled Distributed Systems (2/2)

- The need for local control but overall access
- The need to communicate and interact
- The need for flexibility of growth
- The need to provide users with facilities with realistic response times.

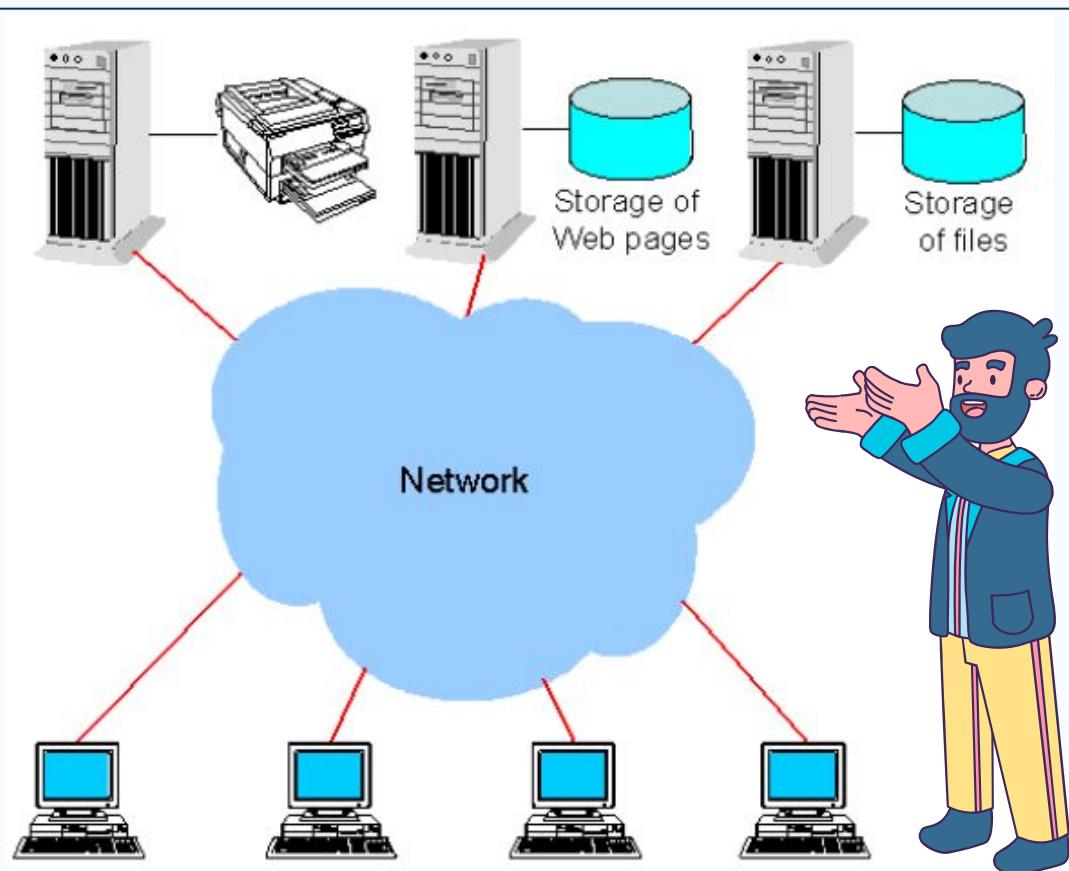
More example on Loosely vs Tightly: embedded.com

Distribution System Architecture

The Workstation and
“Server” model

**A server is a process,
not a computer !**

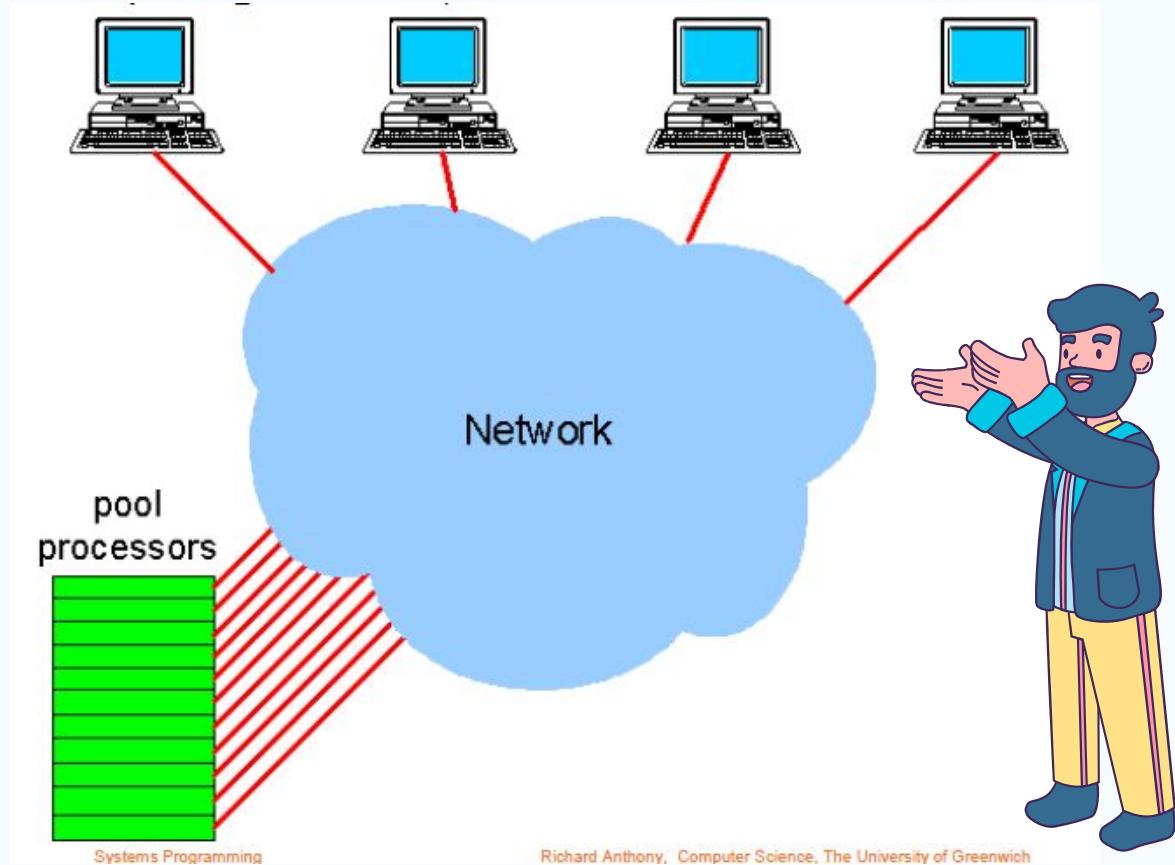
Powerful computers host
services (file service,
database, web service etc.)



Distribution System Architecture

The Processor-Pool model

"Grid Computing" is based on this model, but tends to be larger scale and can be across multiple organisations



Transparency (1/5)

- Distributed systems present numerous **challenges** to the developer, such as
- Where is the process ? Can it be moved ?
- Where is the data / resource ? Can it (data) be moved?
- Providing robustness, and dealing with failures
- Ensuring consistency
- Building scalable systems
(communication efficiency, interaction model).



Transparency (2/5)

- Transparency means ***hiding*** the details of distribution
- The goal is to ***reduce the burden on developers*** so that they can focus their efforts on the ‘business logic’ of the application and not have to deal with all the vast array of technical issues arising because of distribution.

Transparency (3/5)

Access transparency

- Local and remote objects may be accessed with the same operations

Location transparency

- Objects can be accessed without knowledge of their location

Concurrency transparency

- Concurrent processes can use shared objects without interference.

Transparency (4/5)

Replication transparency

- Multiple copies of objects can be created without any effect of the replication seen by applications that use the objects.

Failure transparency

- Faults are concealed such that applications can continue without knowledge that a fault has occurred.

Migration transparency

- (For data objects) Objects can be moved without affecting the operation of applications that use those objects.
- (For processes) Processes can be moved without affecting their operations or results.

Transparency (5/5)

Performance transparency

- The performance of systems should degrade gracefully as the load on the system increases.

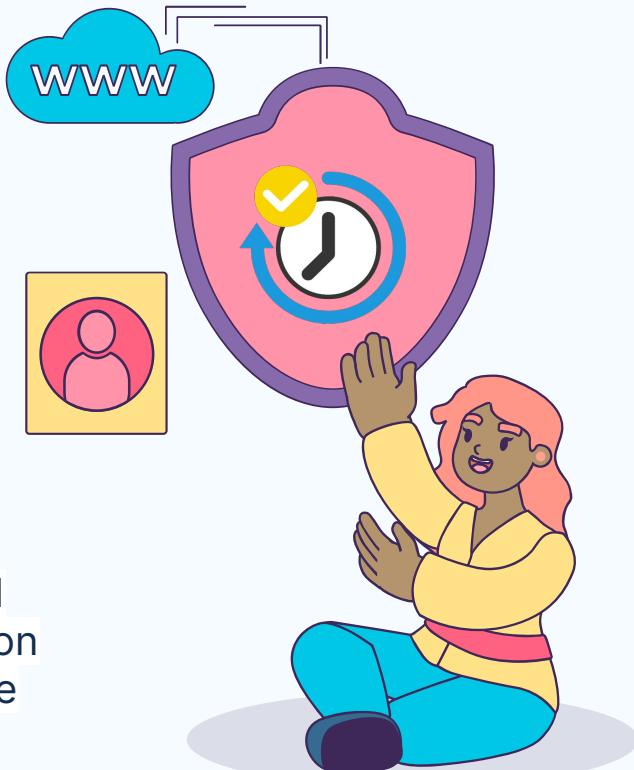
Scaling transparency

- It should be possible to scale up an application, service or system without changing the system structure or algorithms.

02

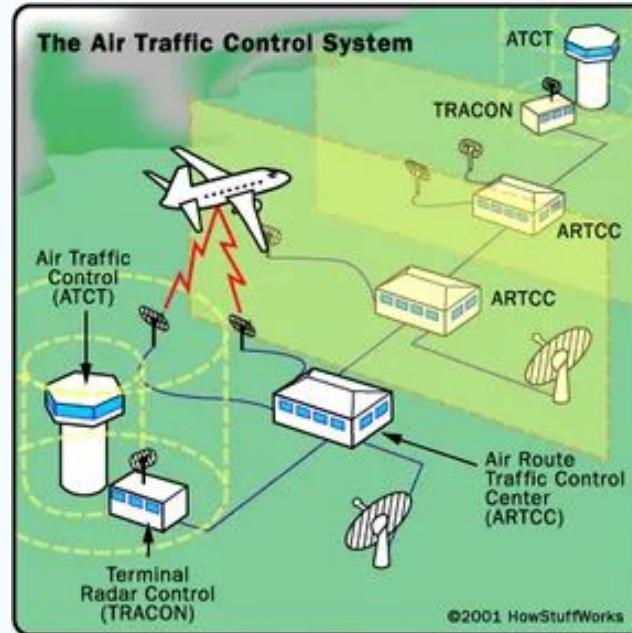
Real-time System

Any information processing system with hardware and software components that perform real-time application functions and can respond to events within predictable and specific time constraints



Real-time System

A real time system consists of a controlling system (computer) and a controlled system (environment)



Source: howstuffworks.com

Real-time System

A real time system consists of a controlling system (computer) and a controlled system (environment)



Source: uber.com , AVS streetscape

Typical Feature of Real-time system

- Time critical
- Made up of concurrent processes (Tasks)
- Share resources (e.g. processor) and communicate with each other
- Reliability and fault tolerance are essential
- Perform a certain specific job
- Car, CD player, phone, camera etc

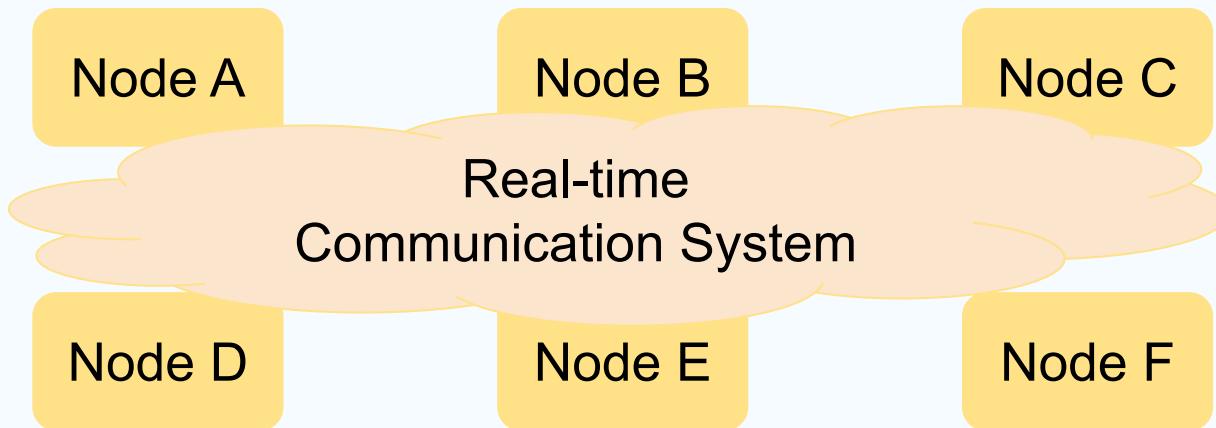
Distributed Real-time system

- Real-time systems very often are implemented as distributed systems. Some reasons:
 - Fault tolerance
 - Certain processing of data has to be performed at the location of the sensors and actuators.
 - Performance issues

Distributed Real-time system

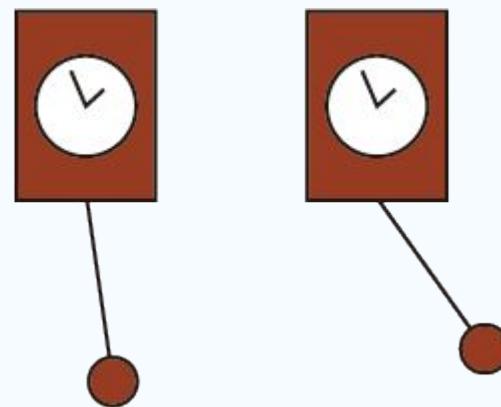
- If the real time computer system is distributed, it consists of a set of (computer) nodes interconnected by a real time communication network.

Source: seas.upenn.edu



Specific Issues Concerning Distributed Real-time system

- Clock synchronisation
- Real-time communication

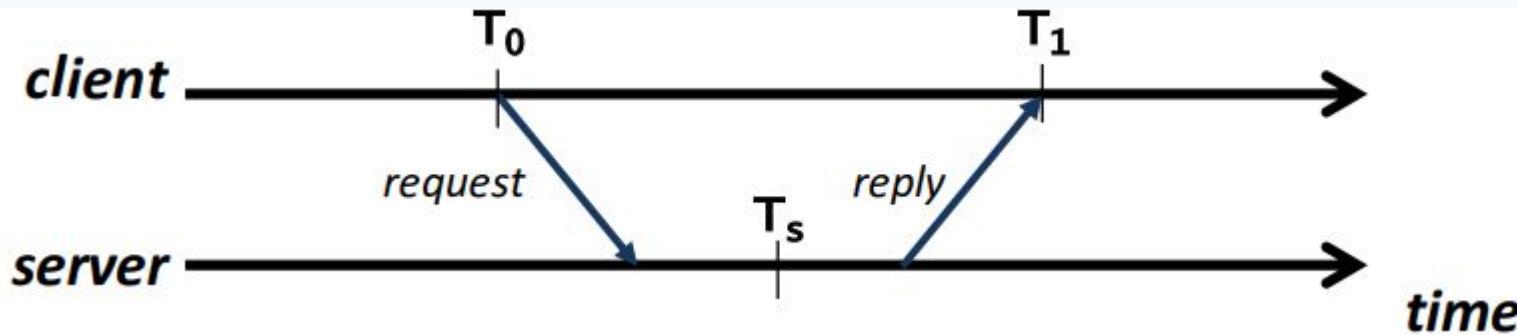


Understanding the clock synchronisation

- Started to look at time in distributed systems
 - Coordinating actions between processes
- Physical clocks 'tick' based on physical processes (e.g. oscillations in quartz crystals, atomic transitions)
 - Imperfect, so gain/ lose time over time
 - wrt nominal perfect 'reference' clock (such as UTC)
- The process of gaining/ losing time is **clock drift**
- The difference between two clocks is called **clock skew**
- **Clock synchronization** aims to minimize clock skew between two (or a set of) different clocks.

Clock synchronisation: Cristian's Algorithm (1989)

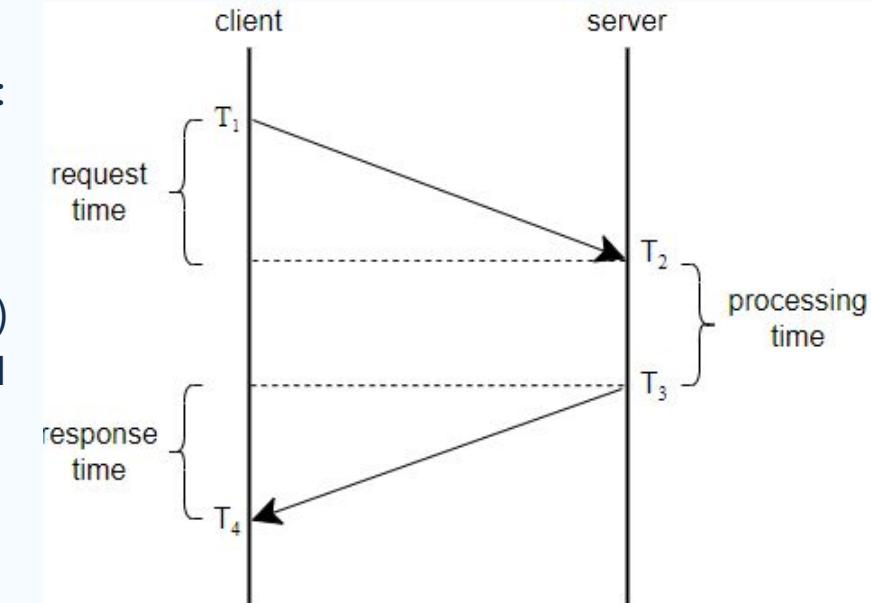
- Uses a time server that is synchronized to Coordinated Universal Time (UTC)



Clock synchronisation: Cristian's Algorithm (1989)

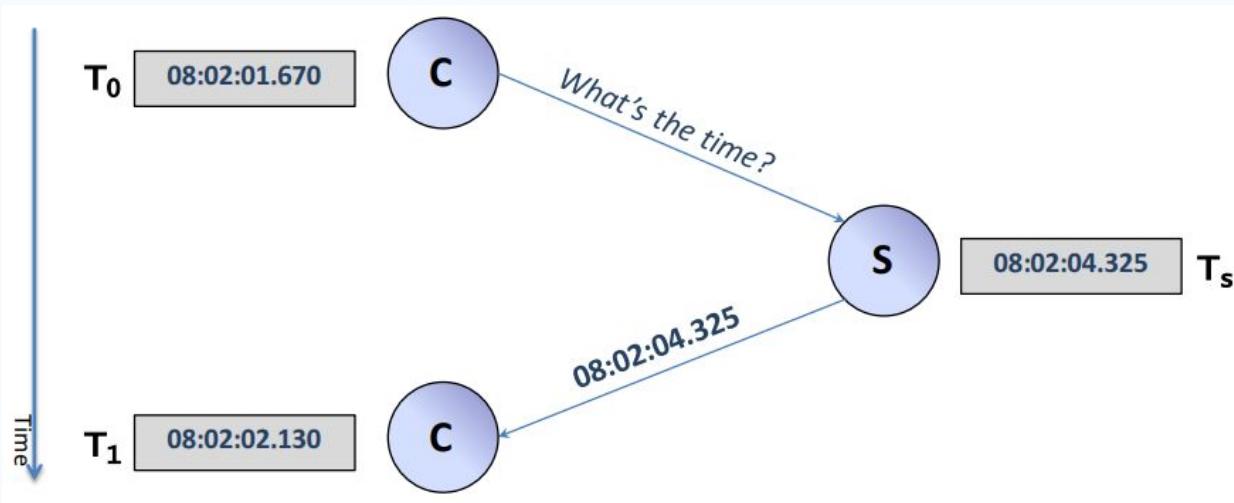
Attempt to compensate for network delays:

- Remember local time just before sending: T_1
- Server gets request, and puts T_s into response (processing time, $T_s = T_3 - T_2$)
- When client receives reply, notes local time: T_4
- Correct time is then approximately $(T_s + (T_4 - T_1)) / 2$



* assumes symmetric behaviour...

Clock synchronisation: Cristian's Algorithm (1989) Example:



$$\text{Gain} = 08:02:02.130 - 08:02:04.555 = 2.425\text{s}$$

Round Trip Time (RTT)
= 460ms, so one way
delay is [approx]
230ms.

Estimate correct time
as $(08:02:04.325 +$
230ms)
= 08:02:04.555

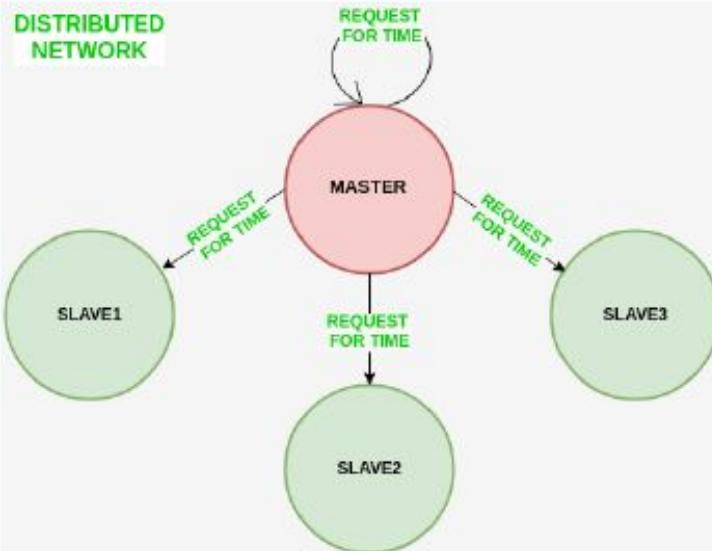
Client gradually
adjusts local clock to
gain 2.425 seconds

Clock synchronisation: Cristian's Algorithm (1989)

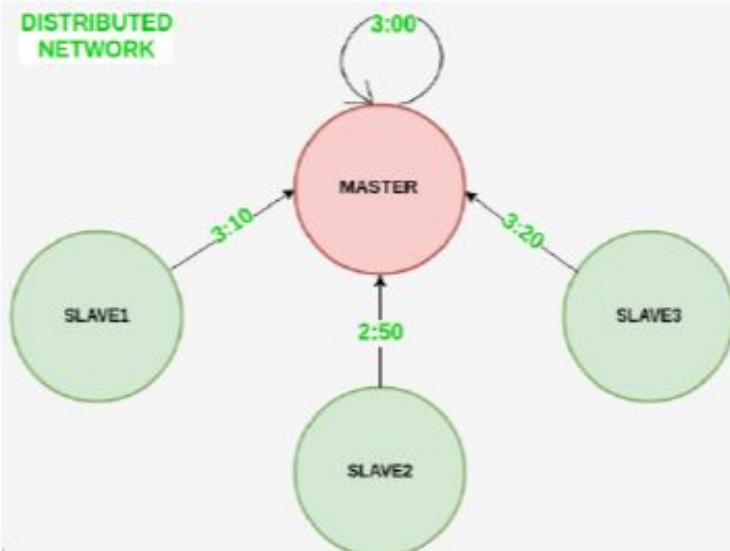
Problem

1. Network delays are time varying
2. Network delays can be different in each direction, even on the same link
3. The processing delay on each computer is also time varying

Clock synchronisation: Berkeley Algorithm (1989)

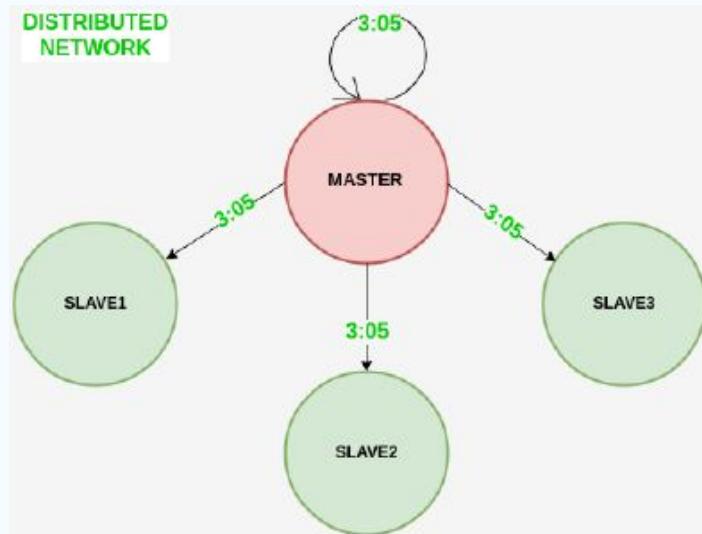


1) The master sends request to slave nodes.



2) The slave nodes send back time given by their system clock

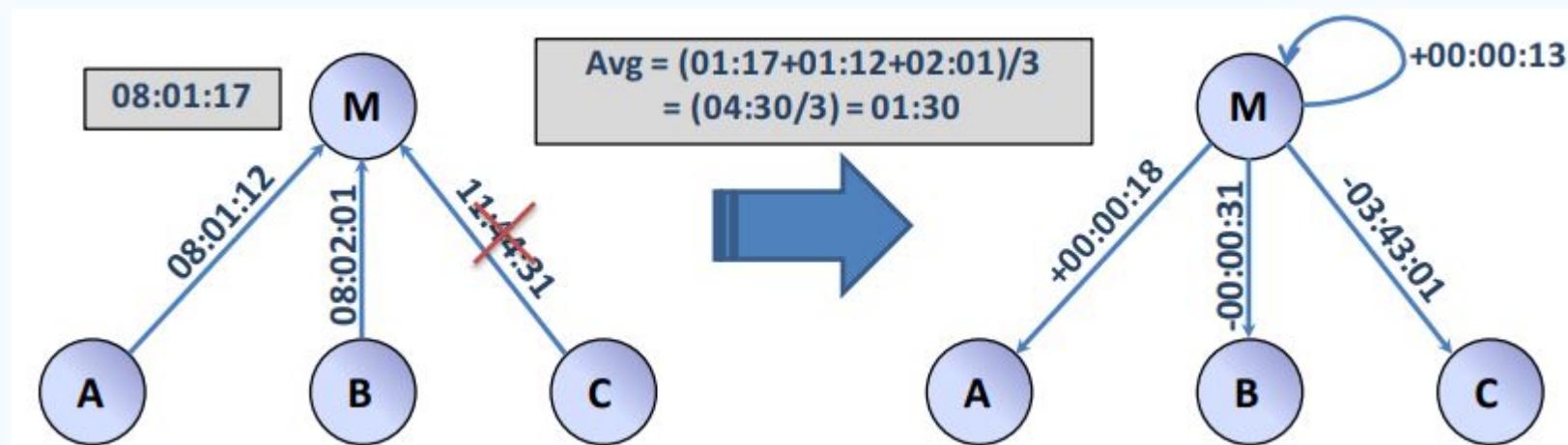
Clock synchronisation: Berkeley Algorithm (1989)



3) Broadcasting synchronised time to whole network

Clock synchronisation: Berkeley Algorithm (1989) Example

Master computes average (including itself, but ignoring outliers), and sends an adjustment to each machine.



Clock synchronisation: Berkeley Algorithm (1989)

Pseudocode

```
# receiving time from all slave nodes
repeat_for_all_slaves : time_at_slave_node =
receive_time_at_slave ()

#calculating time difference time_difference =
time_at_master_node time_at_slave_node

#average time difference calculation
average_time_difference = all_time_differences ) /
number_of_slaves
synchronized_time = current_master_time +
average_time_difference

#broadcasting synchronized to whole network
broadcast_time_to_all_slaves synchronized_time
```

More info : cl.cam.ac.uk

03

Operating System

manages all of the software and hardware on the computers.



What does an OS do ?

Process / Thread

Management

Scheduling / Communication /
Synchronization

**Storage
Management**

**Protection and
Security**

Memory

Management

**File Systems
Management**

Networking

Types of OS with distributed processing features

Network Operating Systems

Microsoft Windows Server,
UNIX, Linux, Mac OS X.

Distributed Operating System

Solaris, Micros, Mach.

Difference between the two types

Autonomy
System Image



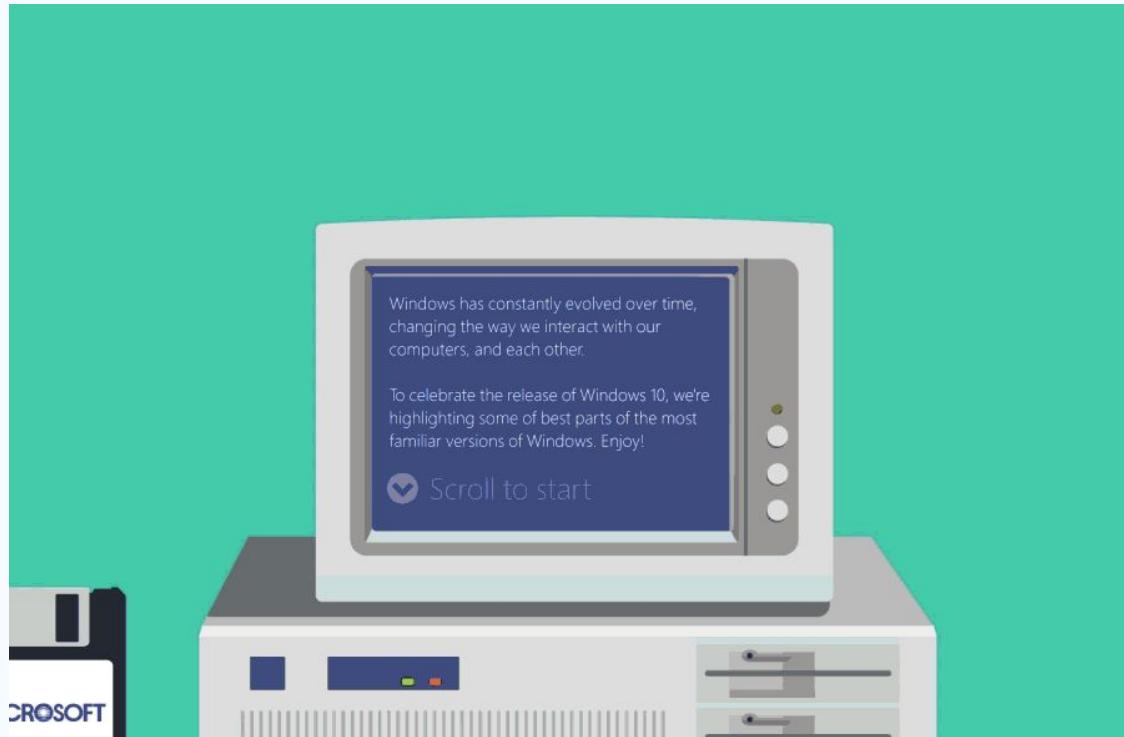
Fault Tolerance Capability

Reference: [tutorialpoints](#)

Difference between types of DOS

Item	Distributed OS		Network OS
	Multiproc.	Multicomp.	
Degree of transparency	Very High	High	Low
Same OS on all nodes	Yes	Yes	No
Number of copies of OS	1	N	N
Basis for communication	Shared memory	Messages	Files
Resource management	Global, central	Global, distributed	Per node
Scalability	No	Moderately	Yes
Openness	Closed	Closed	Open

Enjoy the chronology



Thank
you





BMCS3003 Distributed Systems and Parallel Computing

L02 - Inter-process Communication

Presented by

Assoc Prof Ts Dr Tew Yiqi
May 2023



Table of contents

01

**Client-server,
Peer-to-peer**

02

Socket and Pipes

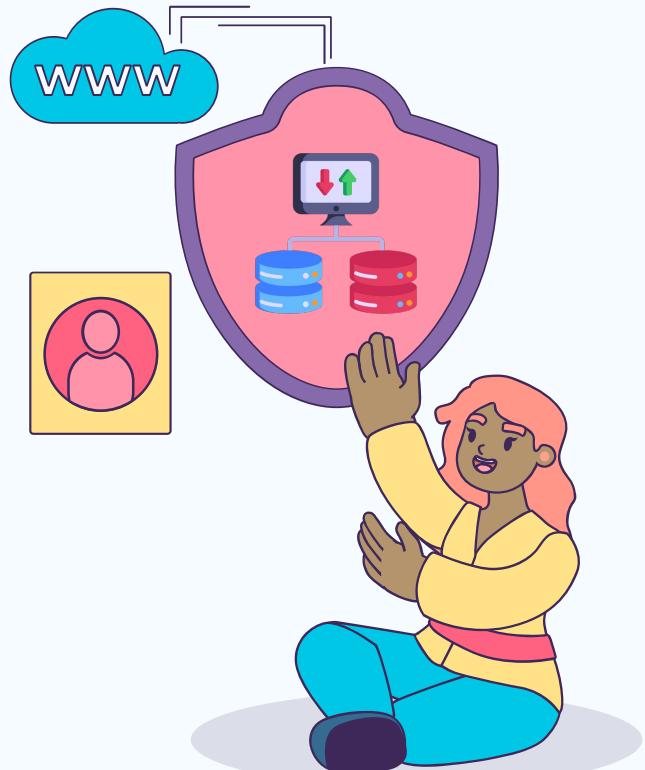
03

**Communication
Issues**

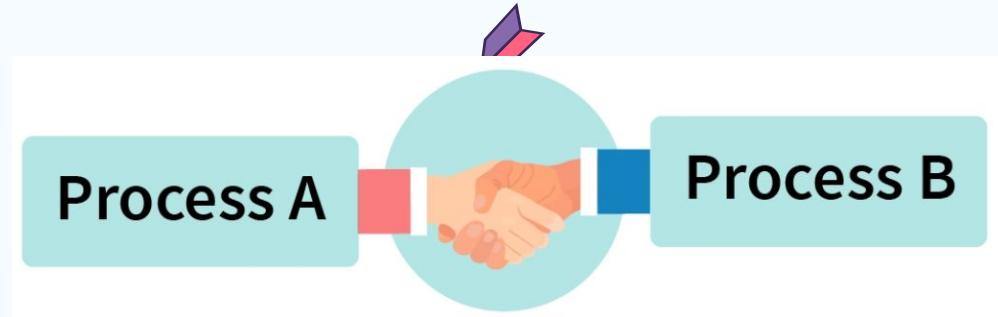
01

Client-server, Peer-to-peer

A relationship in which one program requests a service or resource from another program.



Inter-process communication in an operating system



Processes executing concurrently in the operating system may be either **independent processes** or **cooperating processes**

Independent processes:

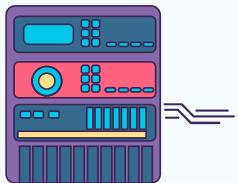
They cannot affect or be affected by the other processes executing in the system

Cooperating processes:

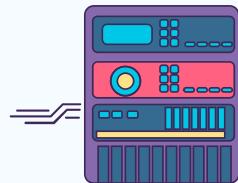
They can affect or be affected by the other processes executing in the system

Independent Processes

Process 1



Process 2

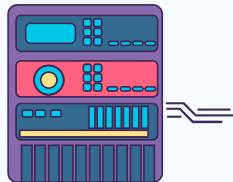


Run concurrently

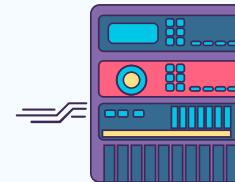
Does not share resources

Cooperating Processes

Process 1



Process 2



Advantages of process cooperation

Information sharing - several users may access to the same file.

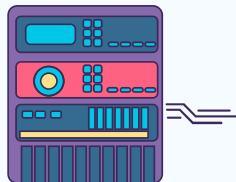
Computation speedup - a task to be divided into several concurrent subtasks.

Modularity - system functions to be divided into separate processes or threads.

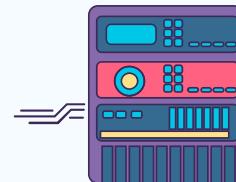
Convenience - a user may perform several tasks at the same time eg editing, compiling or printing.

Inter-process communication (IPC)

Process 1



Process 2

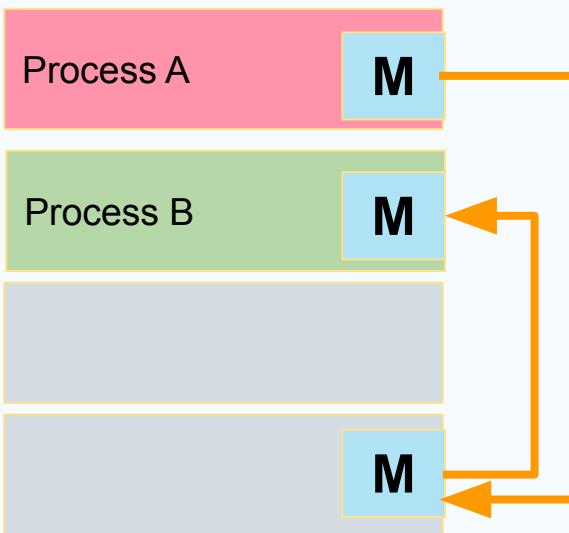


Cooperating processes require an inter process communication(IPC) mechanism that will allow them to exchange data and information.

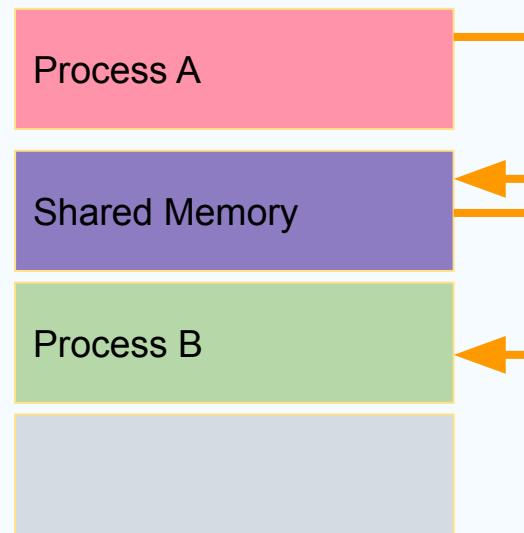
There are 2 fundamental models:

- a. Message passing communication model**
- b. Shared memory communication model**

Communication Models



Message Passing



Shared Memory

Inter-process communication (IPC)

Message passing

- Implemented by using system calls, time consuming task of kernel intervention

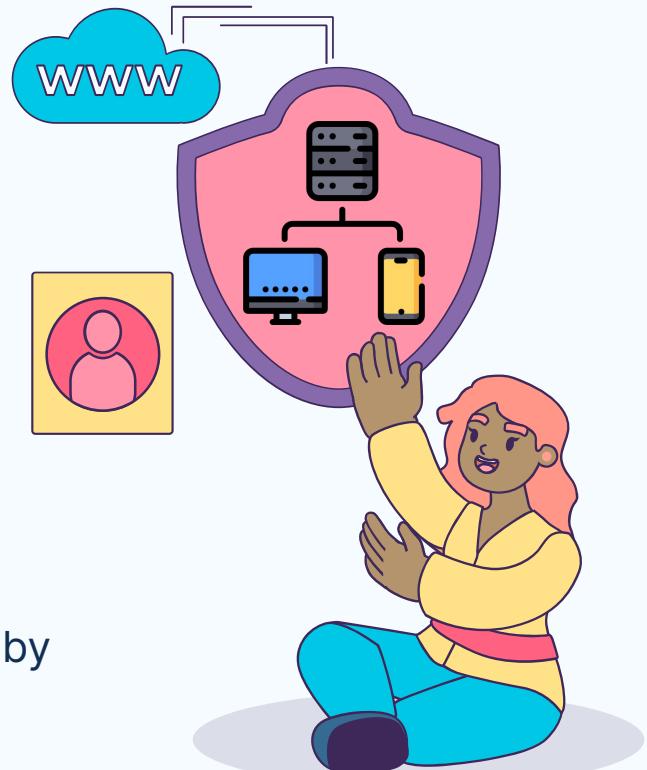
Shared memory

- Faster, kernel is not required
- Message exchange by reading & writing data to the shared region
- Must ensure that they are not writing to the same location simultaneously

01a

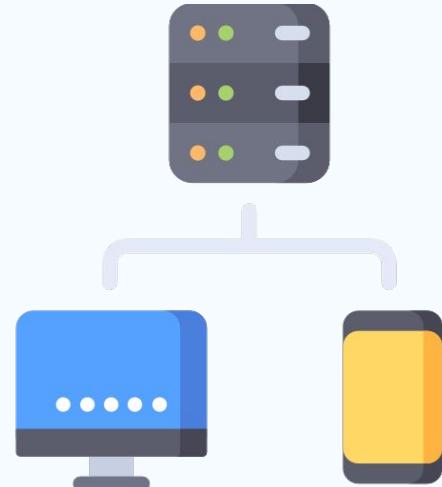
Client-server

Used to distinguish computing by Computers from the monolithic, centralised computing model used by mainframes

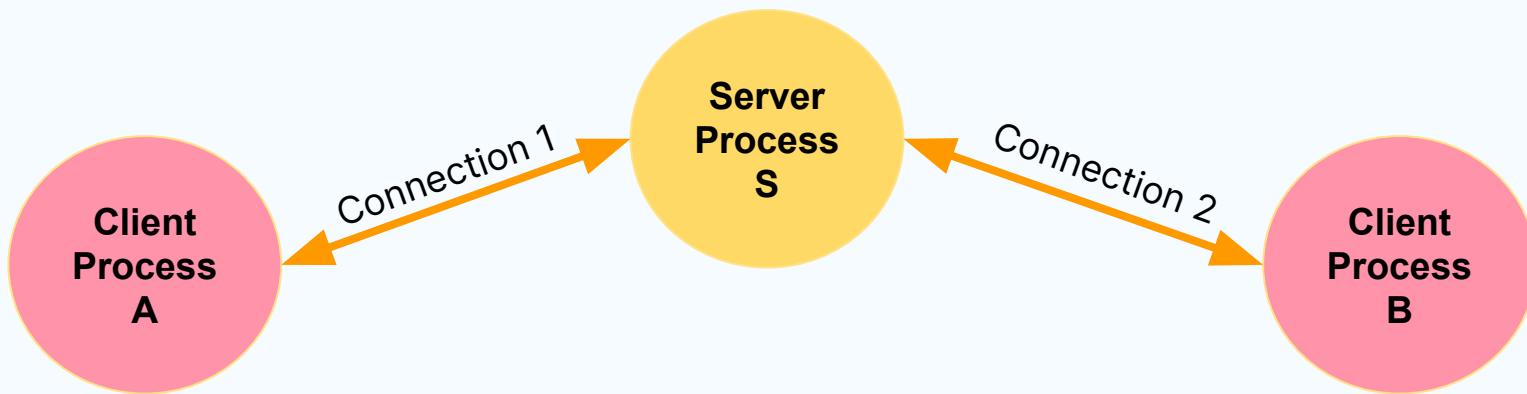


Client - Server

- Perhaps the most well known model of distributed computing
- The application logic is split into two parts:
 - The ‘Server’ is a **process** that provides some sort of computation service.
 - The ‘Client’ is a **process** that makes service requests to the Server.
- The Client is usually associated with human users (it acts as their agent).
- The Server is usually hosted on a dedicated computer that is designed specially
- (e.g. it might have larger memory, faster CPU etc to ensure it can handle requests at a high speed and with low delay).



Client - Server model 1

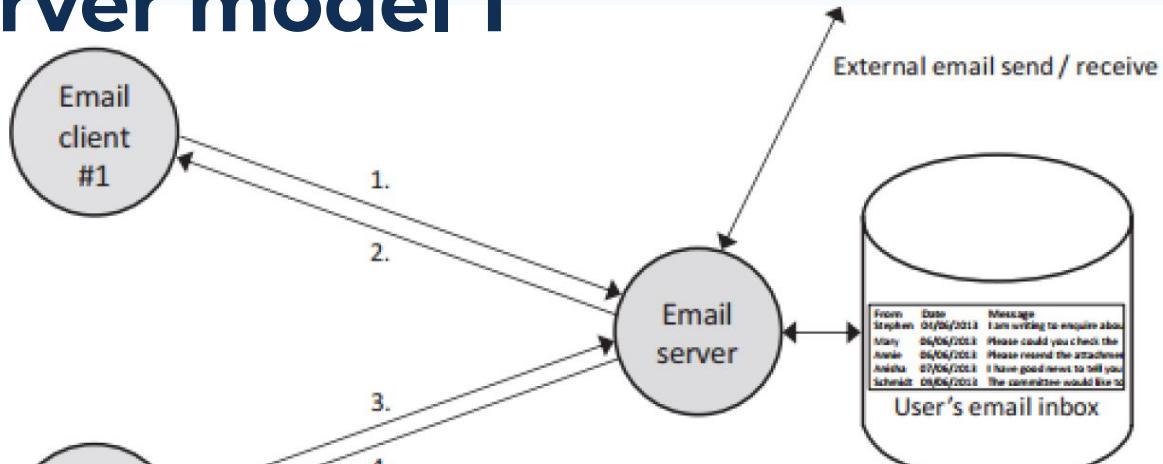


Connections are private between one client and one server.
One server may allow many clients to be connected at one time.
Clients usually initiate communication (as and when service is needed).
All communication is via the server (clients do not communicate directly).

Client - Server model 1

The user uses a desktop computer to check for any new email messages. This computer has a certain email client installed

(a)



B. Later, the same user uses a mobile device such as a tablet to compose and send an email message. The tablet has a different email client installed

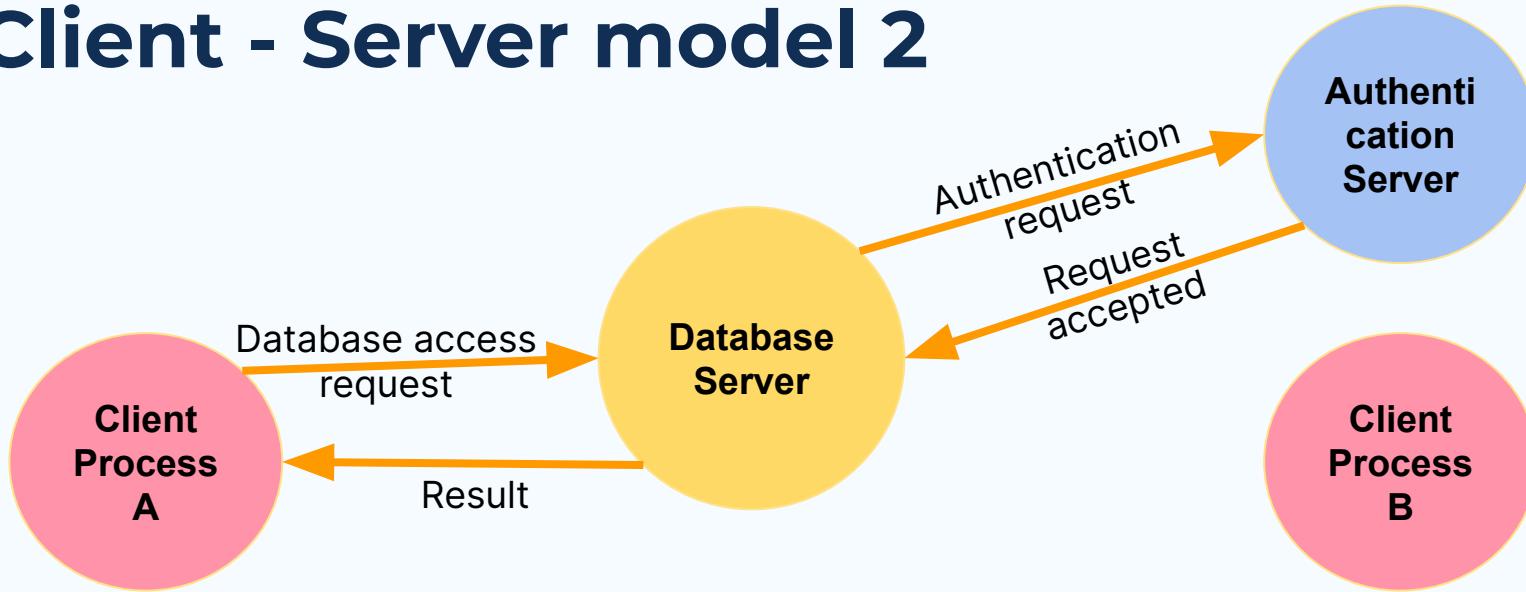
(b)



Key

1. User views email (client requests inbox contents from server)
2. Server sends reply containing email messages from inbox
3. User creates email and 'sends' it (client passes the email message to the server, which actually does the sending)
4. Server sends confirmation that email was sent (some clients may display it and others may not)

Client - Server model 2

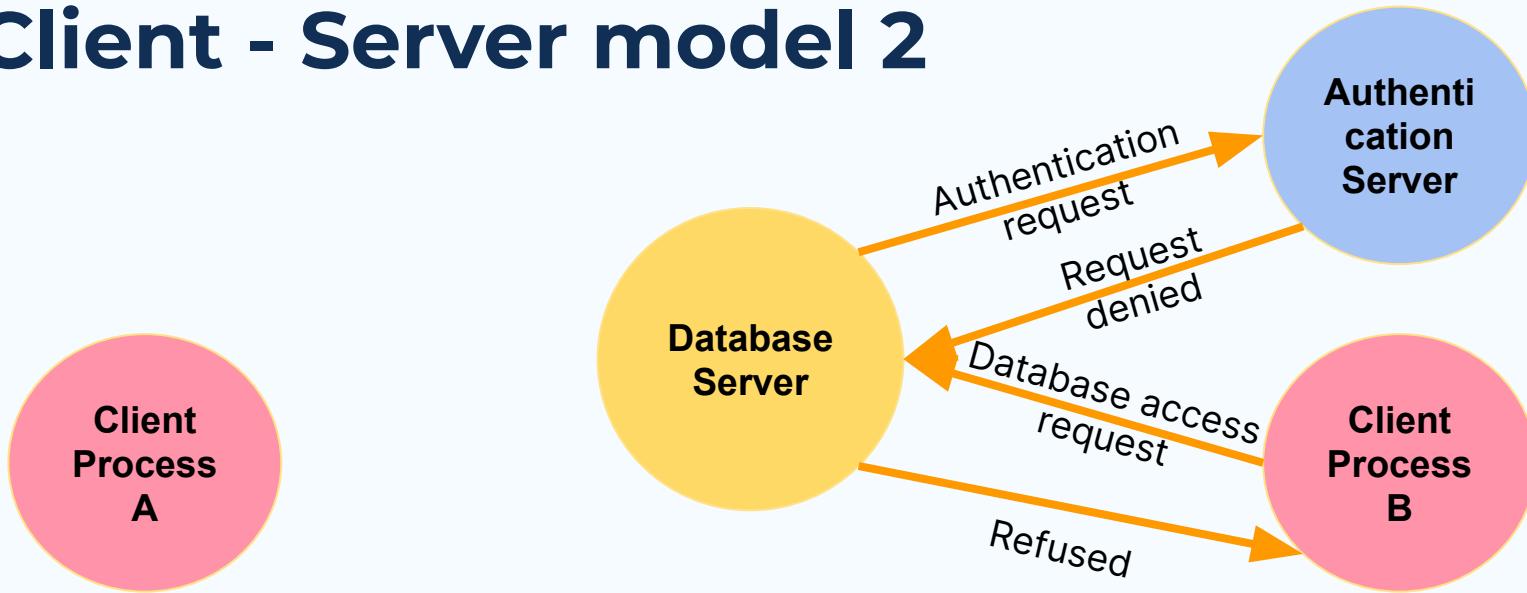


Client - Server is quite a flexible model, can operate at several levels.

Consider a system comprising:

- Database server (holds the database itself and the access / update logic),
- Database clients (user local interfaces to access the database),
- Authentication service (holds information to validate / authenticate users).

Client - Server model 2

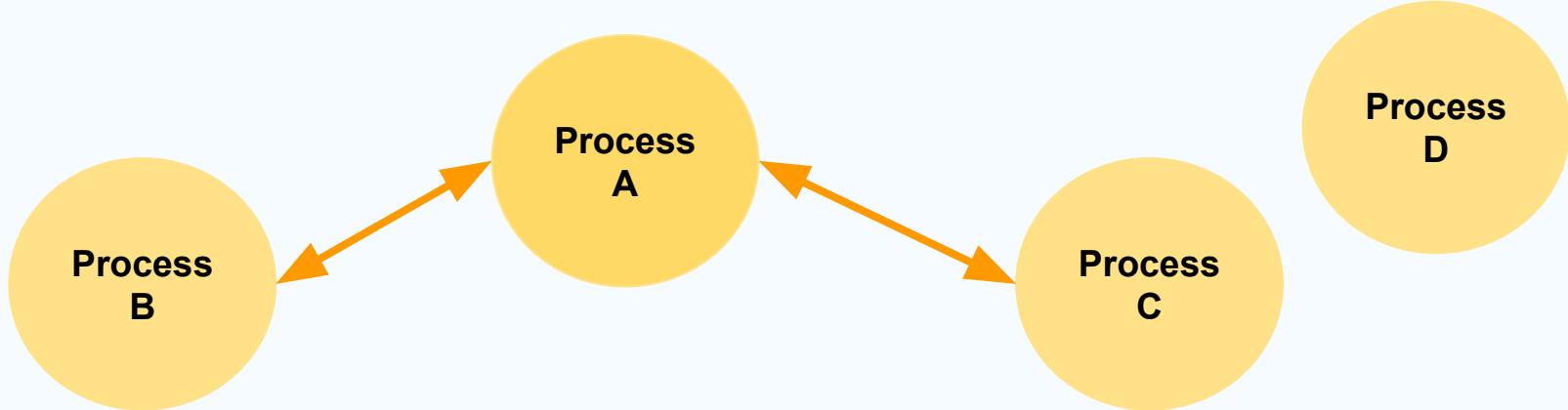


Client - Server is quite a flexible model, can operate at several levels.

Consider a system comprising:

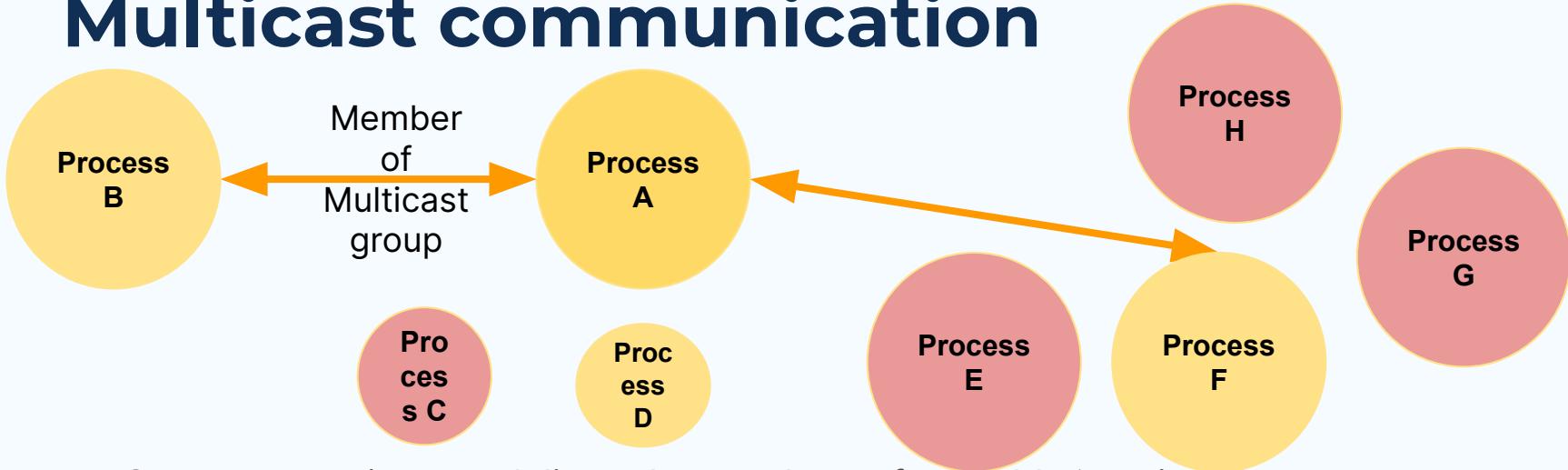
- Database server (holds the database itself and the access / update logic),
- Database clients (user local interfaces to access the database),
- Authentication service (holds information to validate / authenticate users).

Broadcast communication



- One message is sent, delivered to **all** available 'receivers'.
- Sender does not need to know how many, or identities of, receivers.
- Insecure (any process on the **appropriate port** can hear the message).
- Limited to LAN scope (routers block broadcasts).
- Inefficient (interrupts at all receivers even if not interested in the data).

Multicast communication



- One message is sent, delivered to a subset of available 'receivers'.
- Sender may not need to know how many, or identities of, receivers (depends on implementation).
- Insecure (any process on the appropriate port can hear the message).
- Usually limited to LAN scope (routers block some application multicasts, but routers use multicast when sharing routing information amongst themselves).
- Inefficient (interrupts at all receivers even if not interested in the data).

01b

Peer-to-peer

When two or more PCs are connected and share resources without going through a separate server computer.

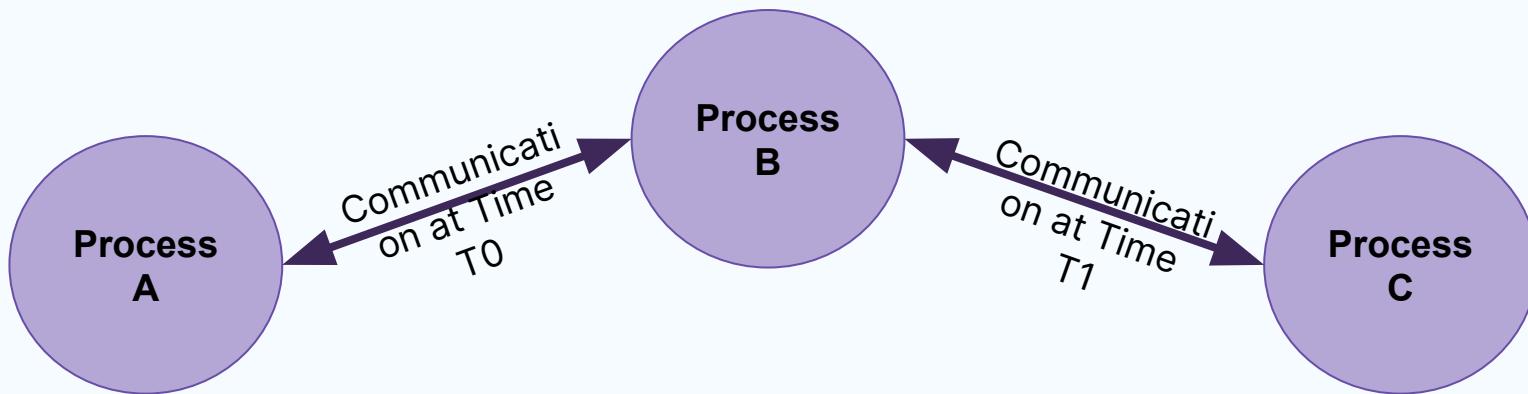


Peer-to-peer



- Peers operate with 'equal standing' each part (peer) of the application has the same basic function.
- Peers communicate to achieve their function (e.g. games, messaging, filesharing).
- Tends to be used in limited scope applications (typically single function) but where connection to remote 'others' needs to be simple and flexible.
- Either side may offer services to the other side.

Peer-to-peer model

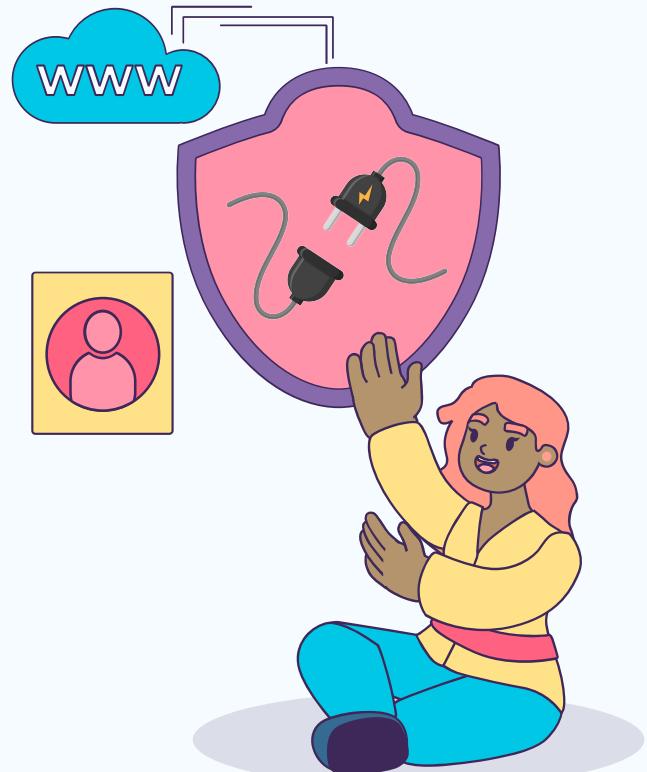


- Connectivity is ad hoc (i.e. it can be spontaneous, unplanned, unstructured).
- Peers can interact with others in any order, at any time.
- Well-suited to mobile applications on mobile devices (some games).
- Some applications (including some 'sensor network' applications) rely on 'promiscuous' connectivity of peers to pass information across a system.

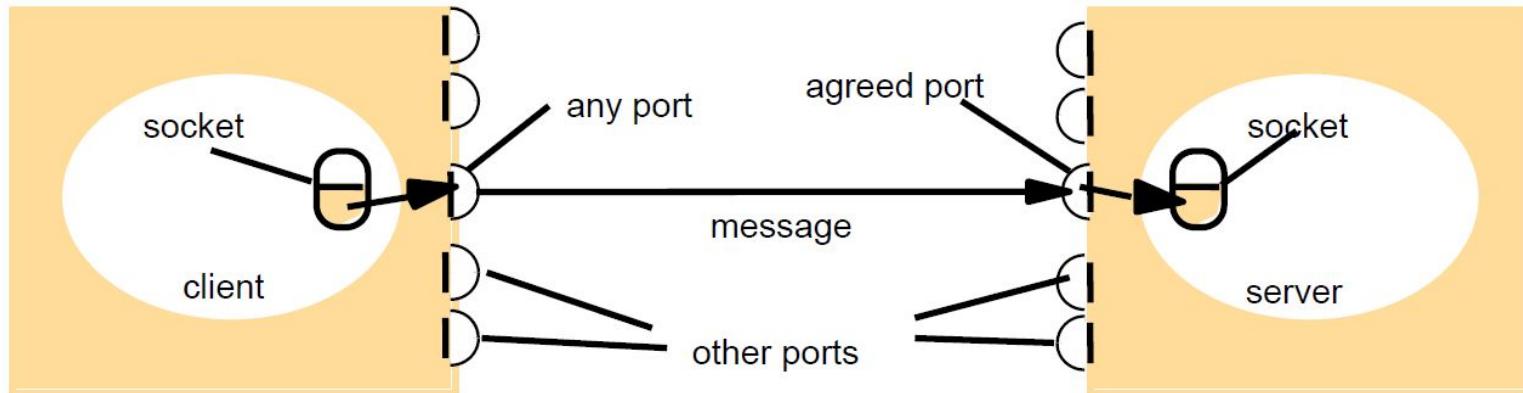
02

Sockets and Pipes

User Datagram Protocol (UDP)
Transmission Control Protocol (TCP)



Sockets and ports



Internet address = 138.37.94.248

Internet address = 138.37.88.249

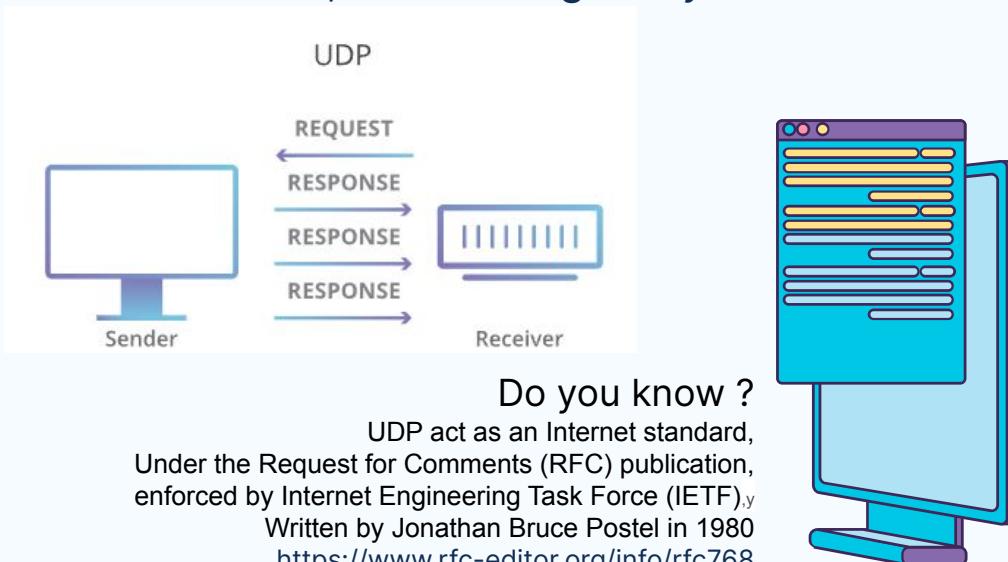
- A library of primitives to support transport layer communication with UDP and TCP.
- The most flexible of all techniques (because it is 'low level').
- Requires the application developer to deal with the communication aspects.

User Datagram Protocol (UDP)

Transmitted from a sending process to a receiving process without acknowledgement or retries. If a failure occurs, the message may not arrive.

Characteristics:

- Connectionless
- Unreliable
- Lightweight
- Point to point (Uni cast)
- Broadcast



UDP use cases

For service that is liable to occasional omission failures

Example:

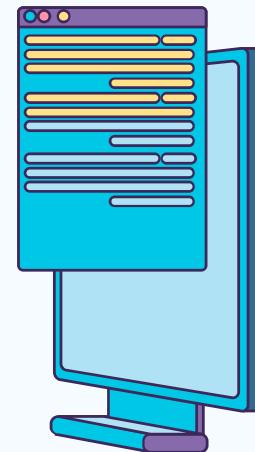
- Domain Name System
- Voice over IP (VOIP)

Activity:

Build the UDP model and send “Hello World” from client
to server.

Question to ponder:

How to generate automatic reply from server to client ?



UDP - server : header

In CPP (under Microsoft Visual Studio, Microsoft Windows OS)

- To include the default Winsock library, for creating a new socket under Windows OS platform.

```
#include <iostream>
#include <WS2tcpip.h>

// Include the Winsock library (lib) file
#pragma comment (lib, "ws2_32.lib")

// Saves us from typing std::cout << etc. etc. etc.
using namespace std;
```

UDP - server : Winsock initialisation

In CPP (under Microsoft Visual Studio, Microsoft Windows OS)

- Set up a Winsock with defined WSADATA object, and version of WinSock to be utilised.

```
// Structure to store the WinSock version. This is filled in
// on the call to WSAStartup()
WSADATA data;

// To start WinSock, the required version must be passed to
// WSAStartup(). This server is going to use WinSock version
// 2 so I create a word that will store 2 and 2 in hex i.e.
// 0x0202
WORD version = MAKEWORD(2, 2);

// Start WinSock
int wsOk = WSAStartup(version, &data);
if (wsOk != 0)
{
    // Not ok! Get out quickly
    cout << "Can't start Winsock! " << wsOk;
    return;
}
```

UDP - server : Socket Creation & Binding

In CPP (under Microsoft Visual Studio, Microsoft Windows OS)

- A socket requests server ip address and port.

```
// Create a socket, notice that it is a user datagram socket (UDP)
SOCKET in = socket(AF_INET, SOCK_DGRAM, 0);

// Create a server hint structure for the server
sockaddr_in serverHint;
serverHint.sin_addr.S_un.S_addr = ADDR_ANY; // Us any IP address available on the machine
serverHint.sin_family = AF_INET; // Address format is IPv4
serverHint.sin_port = htons(54000); // Convert from little to big endian

// Try and bind the socket to the IP and port
if (bind(in, (sockaddr*)&serverHint, sizeof(serverHint)) == SOCKET_ERROR)
{
    cout << "Can't bind socket! " << WSAGetLastError() << endl;
    return;
}
```

UDP - server : Main Loop Setup & Entry

- Initialize buffer type and size for sending and receiving data.
- Refer to the UDPserver.cpp file to see how to use Send and Recv function.

UDP - server : Close socket

In CPP (under Microsoft Visual Studio, Microsoft Windows OS)

- The defined WSADATA object, will be clean up before ending the programme.

```
// Close socket  
closesocket(in);  
  
// Shutdown winsock  
WSACleanup();
```

UDP - client : Send message

- Same as server for initialising and ending the socket network configurations.
- Send the message based on the data size.

```
// Create a hint structure for the server
sockaddr_in server;
server.sin_family = AF_INET; // AF_INET = IPv4 addresses
server.sin_port = htons(54000); // Little to big endian conversion
inet_pton(AF_INET, "127.0.0.1", &server.sin_addr); // Convert from string to byte array
                                                    // Socket creation, note that the socket type is datagram
SOCKET out = socket(AF_INET, SOCK_DGRAM, 0);

// Write out to that socket
string s(argv[1]);
int sendOk = sendto(out, s.c_str(), s.size() + 1, 0, (sockaddr*)&server, sizeof(server));

if (sendOk == SOCKET_ERROR)
{
    cout << "That didn't work! " << WSAGetLastError() << endl;
}

// Close the socket
closesocket(out);

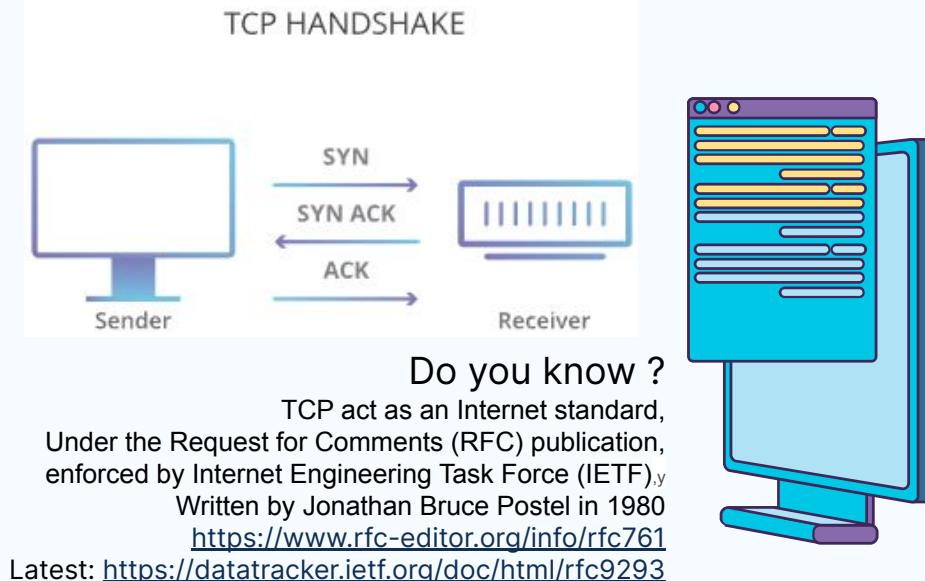
// Close down Winsock
WSACleanup();
```

Transmission Control Protocol (TCP)

Stream communication assumes that when a pair of processes are establishing a connection, one of them plays the client role and the other plays the server role , but thereafter they could be peers

Characteristics:

- Connection oriented
- Reliable
- Relatively high overheads
(larger header, acks ,
more processing overhead)
- Point to point



TCP use cases

HTTP : The Hypertext Transfer Protocol

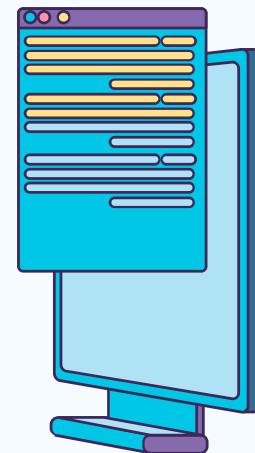
FTP : The File Transfer Protocol

Telnet : Telnet provides access by means of a terminal session to a remote computer

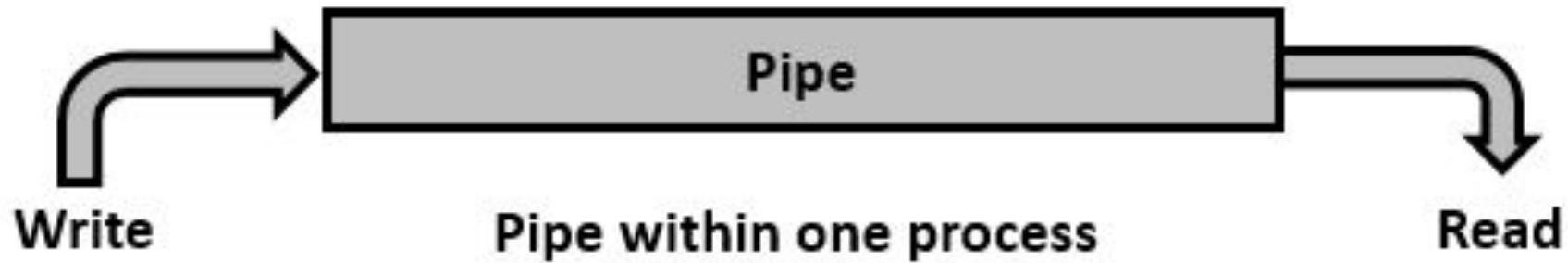
SMTP : The Simple Mail Transfer Protocol

Activity:

Demonstration on TCP



Pipes



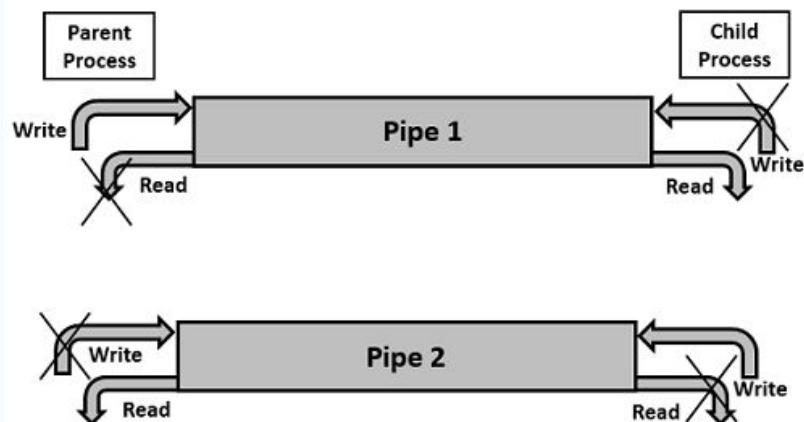
- Unidirectional
- Standard producer consumer mechanism
- Parent-child relationship between communicating processes
- Once communication is over and processes terminated, the ordinary pipe ceases to exist

Pipes properties (with example)

- More powerful with assigned pipe's name
- Do not require parent child relationship
- Once a named pipe is created, multiple non related processes can communicate over it
- Must be explicitly deleted

Example:

Pipe communication with two pipes
tutorialspoint.com



03

Communication Issue

Message Passing, Remote Procedure Call,
Remote Method Invocation, Middleware.



Message Passing

- This is a simple unstructured form of communication in which messages are passed from a sender to a receiver.
- Conceptually this is very similar to datagram communication provided by UDP.

Message Passing Interface (MPI)

- A specific implementation of message passing.
- Very popular in parallel processing applications.
- Has a large number of enhancements (beyond for example UDP datagrams), added specifically to support the synchronisation aspects of communication in parallel applications.

Message Passing Model (Abstraction)

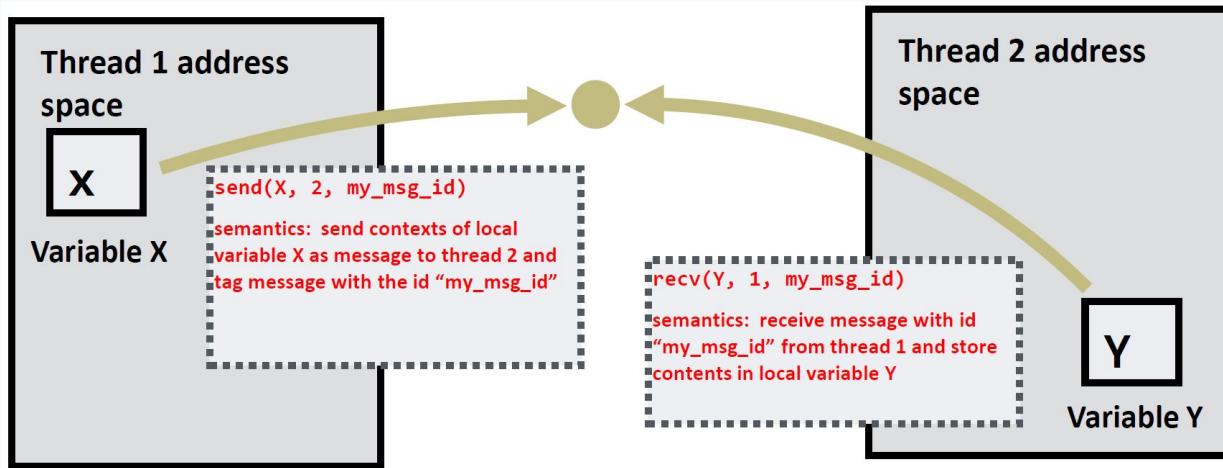
Threads operate within their own private address spaces

Threads communicate by sending/receiving messages

send : specifies recipient, buffer to be transmitted, and optional message identifier ("tag")

receive : sender, specifies buffer to store data, and optional message identifier

Sending messages is the only way to exchange data between threads 1 and 2



Message Passing Systems

Popular software library: **MPI (message passing interface)**

Hardware need not implement system wide loads and stores to execute message passing programs (need only be able to communicate messages)

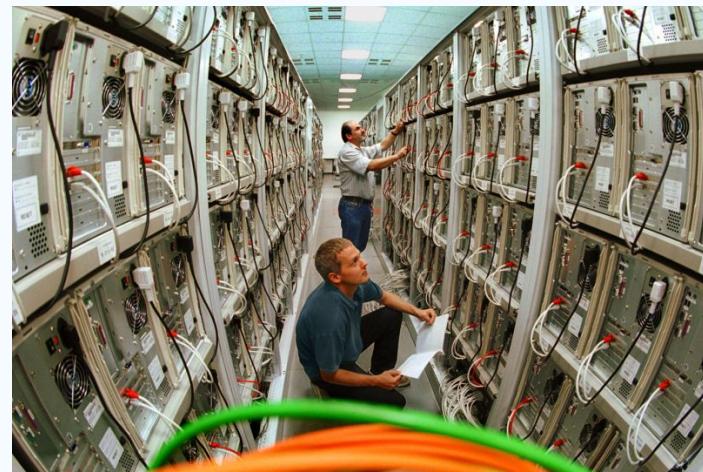
- Can connect commodity systems together to form large parallel machine (message passing is a programming model for clusters).

Source:

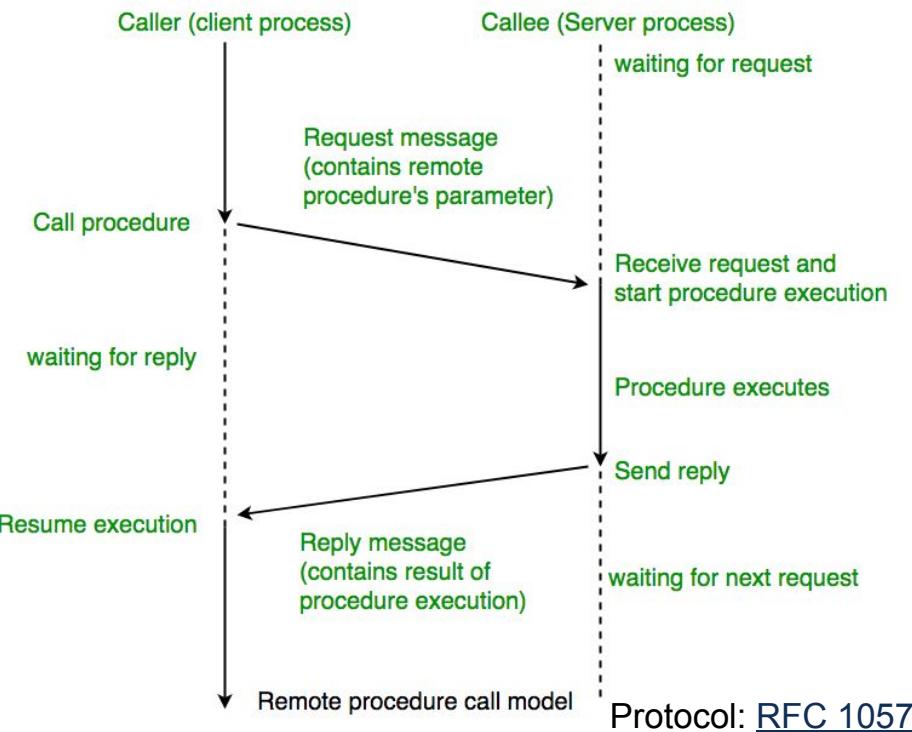
Technicians working on a cluster consisting of many computers working together by sending messages over a network.

MEGWARE Computer GmbH - <http://www.megware.com>

Chemnitzer Linux Cluster (CLIC) an der Technischen Universität Chemnitz

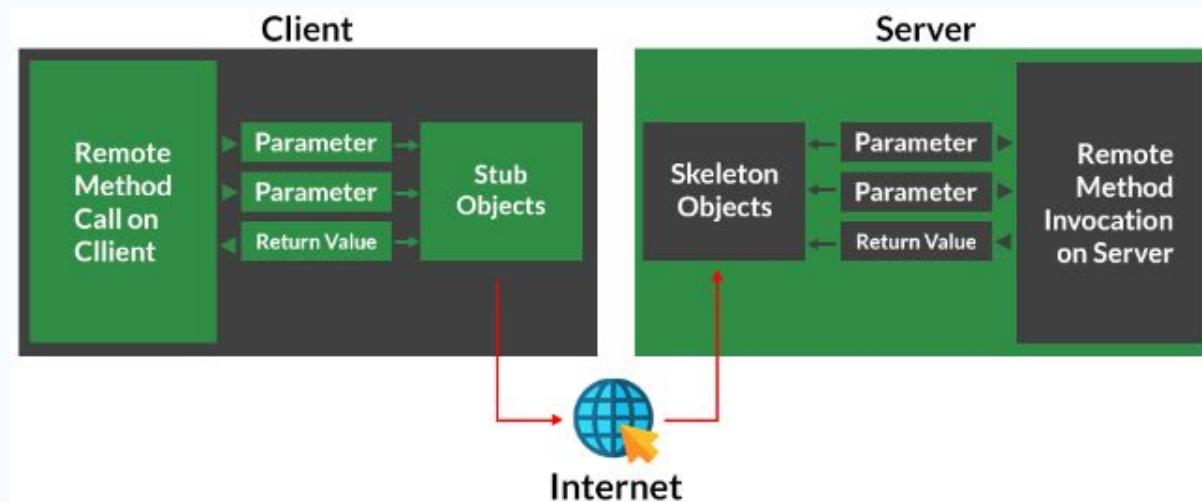


Remote Procedure Call (RPC)



More Example : [IBM](#)
 Exercise : [Dave Marshall](#)

Remote Method Invocation (RMI)



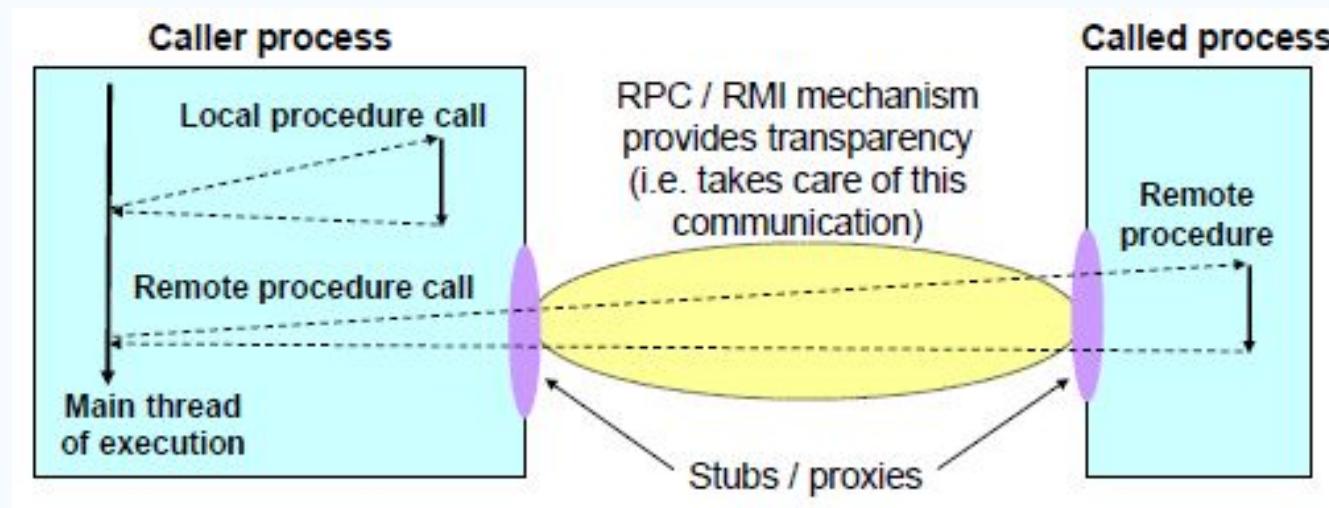
The object oriented version of RPC.

For object oriented languages such as (java, C++).

One side makes a method call to a method within the other process.

More Example : [RMI in Java](#)
Exercise : [Java Card](#)

RPC / RMI model

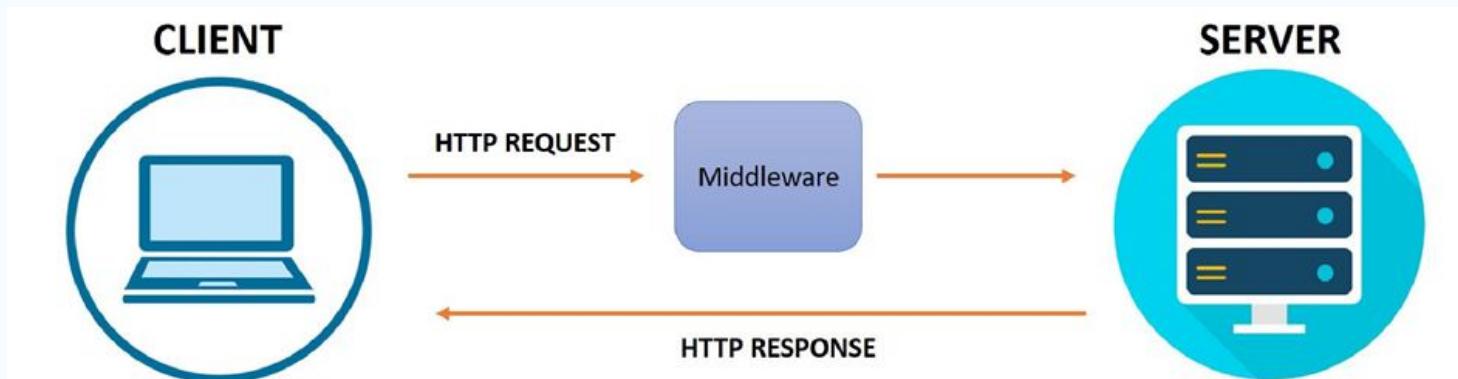


A process can call a procedure (or invoke a method) in another process

To the programmer this is made to look as much like a local procedure call as possible.

The RPC / RMI mechanisms provide local 'stub' mechanisms which act as communication proxies, so all communication appears to be 'process local'.

Middleware



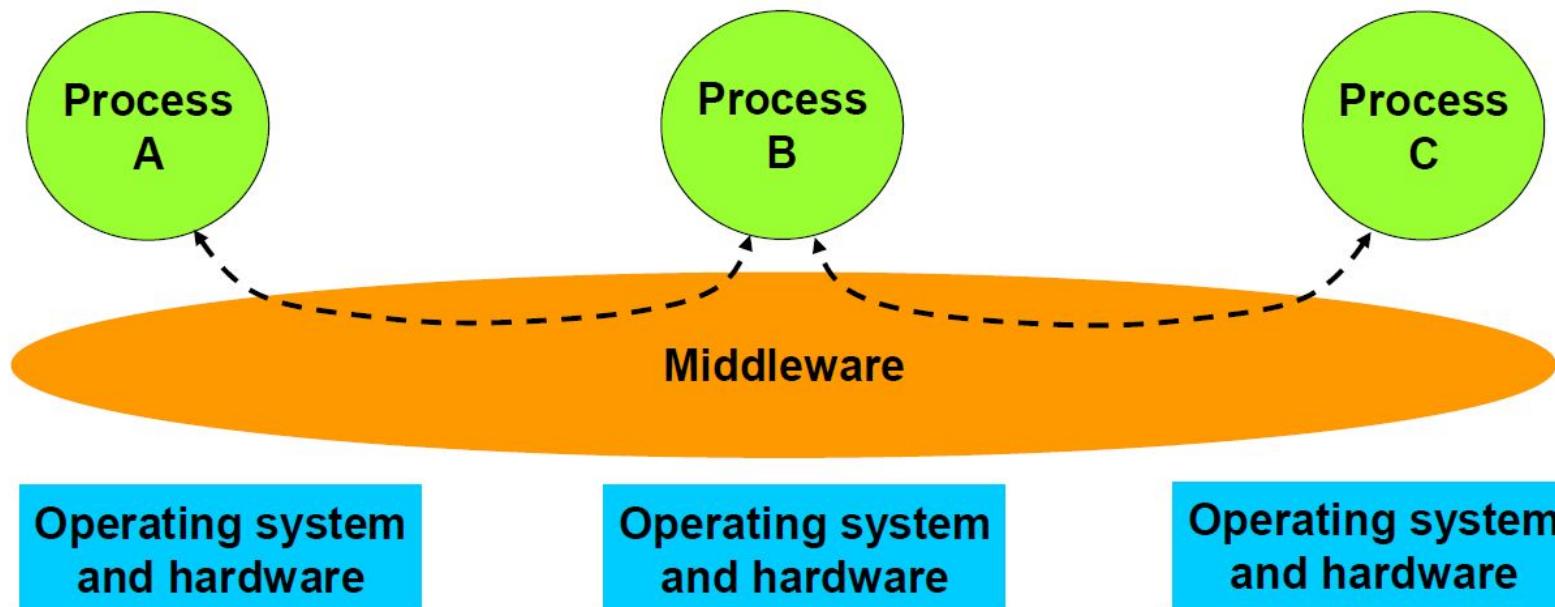
A software 'layer' that sits between processes and the network.

Can comprise part of the operating system and some special services.

Services include location and remote invocation services.

The underlying communication is usually sockets.

Middleware Concept



More Description: [Amit Saha](#)
Exercise : [Middleware Web Apps with Go, Js, Python](#)

Middleware

Application level communication resembles RMI / RPC but with greater transparency:

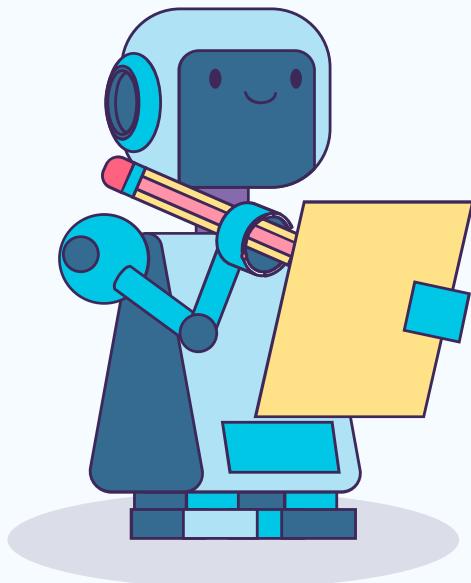
- Application developer does not need to be concerned with the communication mechanism, OR the location of processes.
- Supports relocation of objects transparent to application.

As shown in the conceptual diagram, processes communicate via the middleware layer.

The middleware hides details of location (a process need not know where the other process is).

The middleware itself is implemented with sockets or RMI

The middleware consists of processes and services on each computer, but acts like a continuous 'layer' or 'channel' that is spread across all computers.



Thank you

CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)



BMCS3003 Distributed Systems and Parallel Computing

L03 - Memory Management

Presented by

Assoc Prof Ts Dr Tew Yiqi
May 2023

Table of contents

01

**Centralised Memory
Management**

02

**Simple, Shared,
Distributed Shared
Memory**

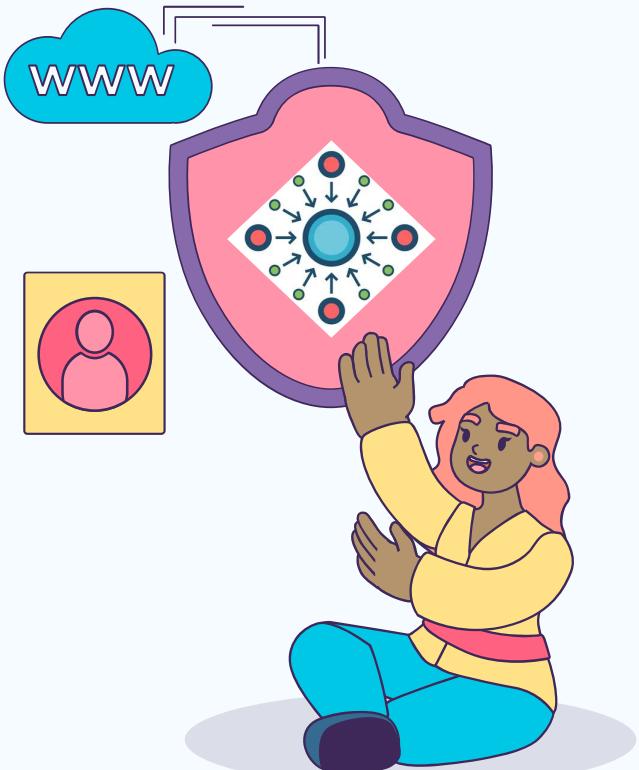
03

Memory Migration

01

Centralised Memory Management

An architecture in which memory allocations are grouped based on their type, lifetime, or other requirements.



As computing tasks get larger and larger, may need to enlist more computers to the job.



Tasks feeder



As computing tasks get larger and larger, may need to enlist more computers to the job.

Bigger:

more memory and storage

Faster:

each processor is faster

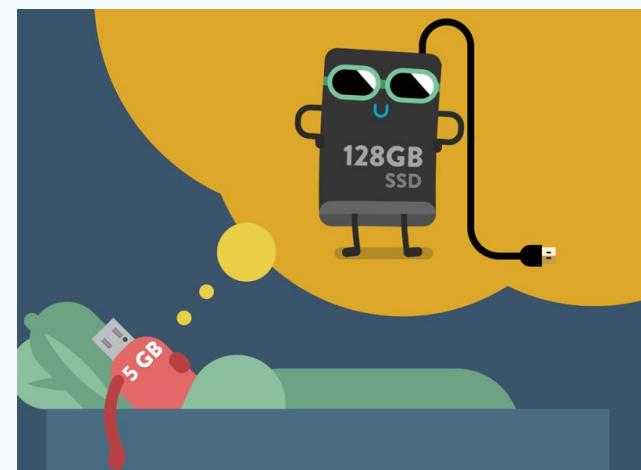
More tasks:

do many computations simultaneously

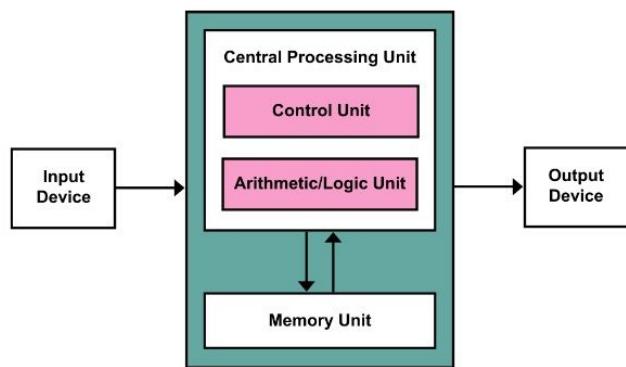
Review of Centralised Memory Management

Virtual memory

- Extending the size of available memory beyond its physical size of RAM
- Paging versus segmentation
- Internal versus external fragmentation
- Segment placement algorithms
- Page replacement algorithms
 - page faults and thrashing



Simple Memory Model



In Simple Memory Model access times for all processors are equal.

Requires strict control of degree of multi-programming.

Often does not use virtual memory or caching because of overhead.

Shared Memory Model

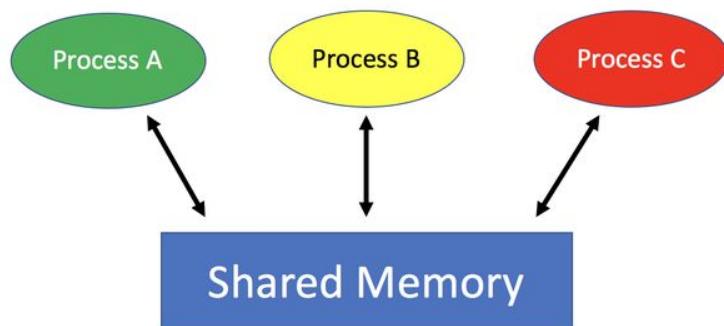
Used for inter-process communication.

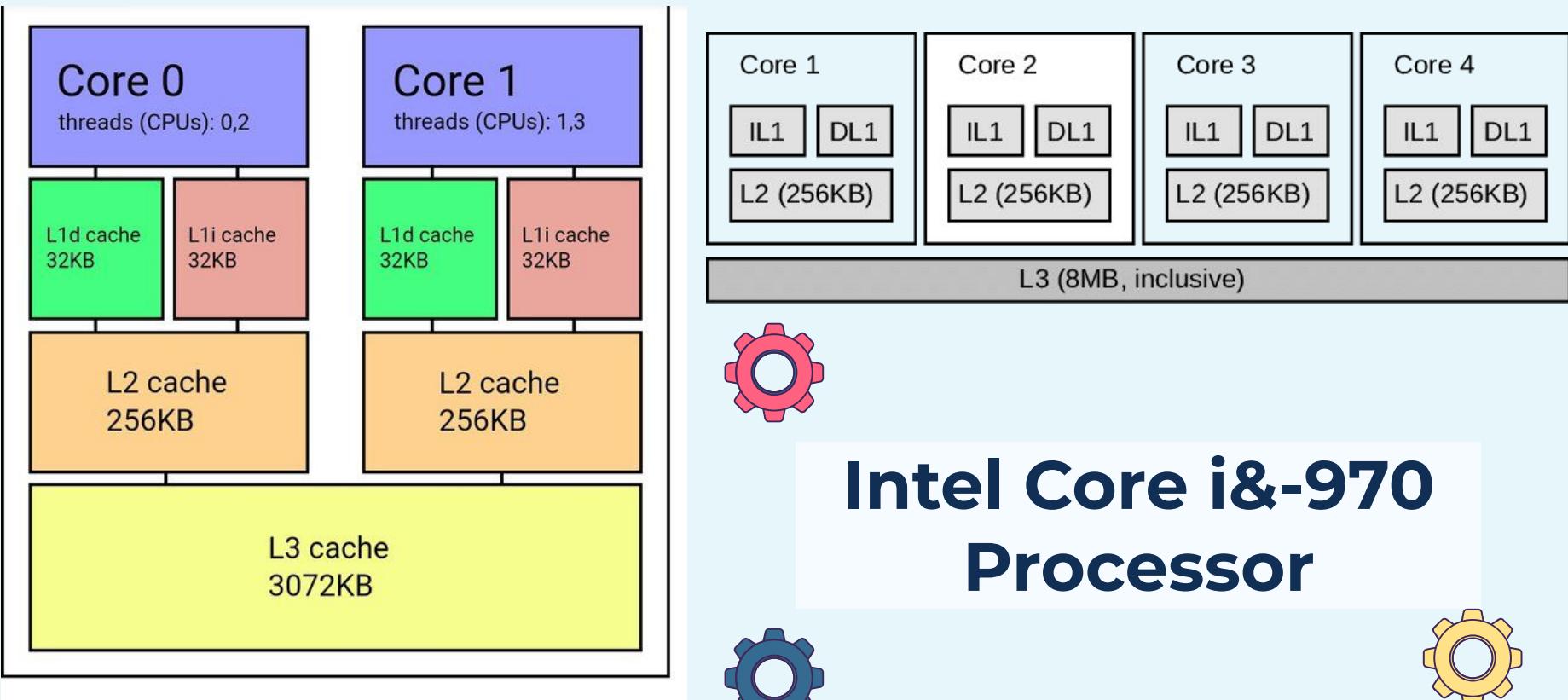
Multiple processes share memory locations.

May include physical RAMs, local cache, and secondary storage.

Memory access takes place via common bus thus a possibility of a **bus contention**.

Example: OpenMP





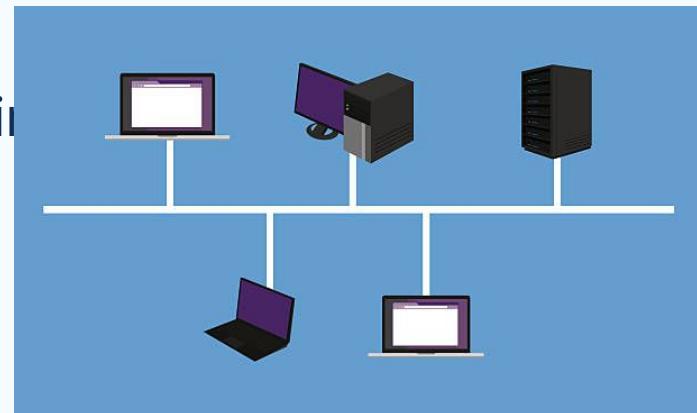
Intel Core i&-970 Processor

Bus Contention

Bus contention occurs when the demand for bus access is excessive.

Bus contention may cause a bottleneck in a shared memory system.

Larger multiprocessor systems (>32 CPUs) cannot use a single bus to interconnect CPUs to memory modules, because bus contention becomes unmanageable.

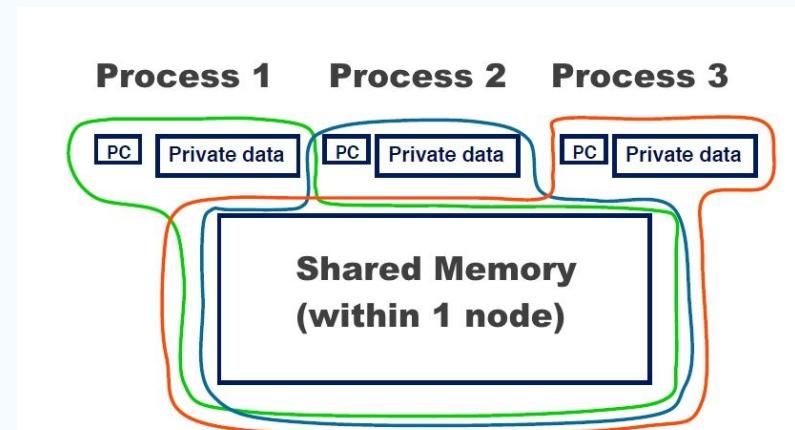


Shared Memory Performance

Performance is an important issue in a *Shared Memory System*.

Important issues:

- scalability ability to accommodate growth without sacrificing performance
- real time needs
 - overlap of communication and computation
 - prefetching of data
- non local memory references are expensive (up to 10:1 ratio)

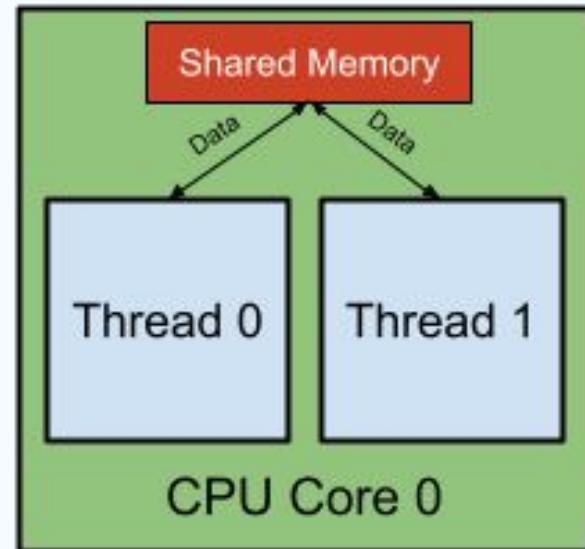


More reference: [OpenMP MIMD](#)

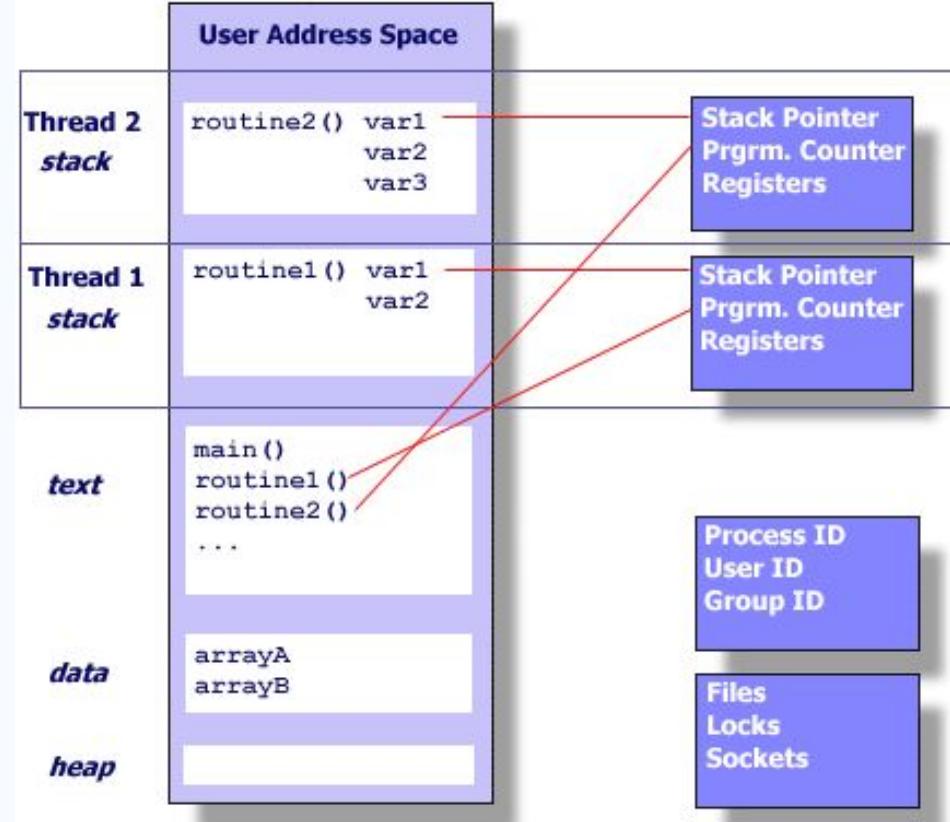
Shared Memory Performance

Large symmetric multiprocessor systems offered more compute resources to solve large computationally intense problems.

- Threading is the most popular shared memory programming technique
- Advantage:
 - makes it easy for a developer to divide up work, tasks, and data.
- Disadvantage:
 - data races
- E.g., OpenMP



Threads are lightweight process and share process state among multiple threads. This greatly reduces the cost of switching contexts



OpenMP is an acronym for **Open Multi-Processing**.

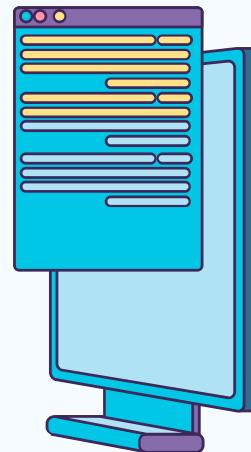
OpenMP is a directive-based Application

Programming Interface (API) for developing parallel programs on shared memory architectures. The OpenMP standard is maintained by the OpenMP ARB, a corporation whose board of directors includes representatives from many major computer hardware and software vendors.

"The OpenMP ARB (Architecture Review Boards) mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable."



Source: [Cornell University](#)



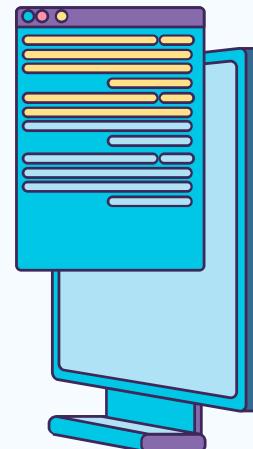
```
// A pragma defines beginning of a  
parallel region and specifies iterations  
of the following for loop should be  
spread across openmp threads and is the  
extent of the parallel region
```

```
#pragma omp parallel for  
for (i =0; i < n; i++)  
{  
    . . . ;  
    computations to be completed ;  
    . . . ;  
}
```

Official Documentation
[OpenMP API](#)

More Lessons:
[Lawrence Livermore](#)
[National Laboratory](#)

[Jaka's Corner](#)



```
#include <unistd.h>
#include <stdlib.h>
#include <omp.h>
#include <stdio.h>
#define THREADS 4
#define N 16
int main ( ) {
    int i;
    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for (i = 0; i < N; i++) {
        /* wait for i seconds */
        sleep(i);
        printf("Thread %d has completed iteration %d.\n",
            omp_get_thread_num( ), i);
    }
    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

What is this code doing ?

What do the OpenMP semantics specify ?

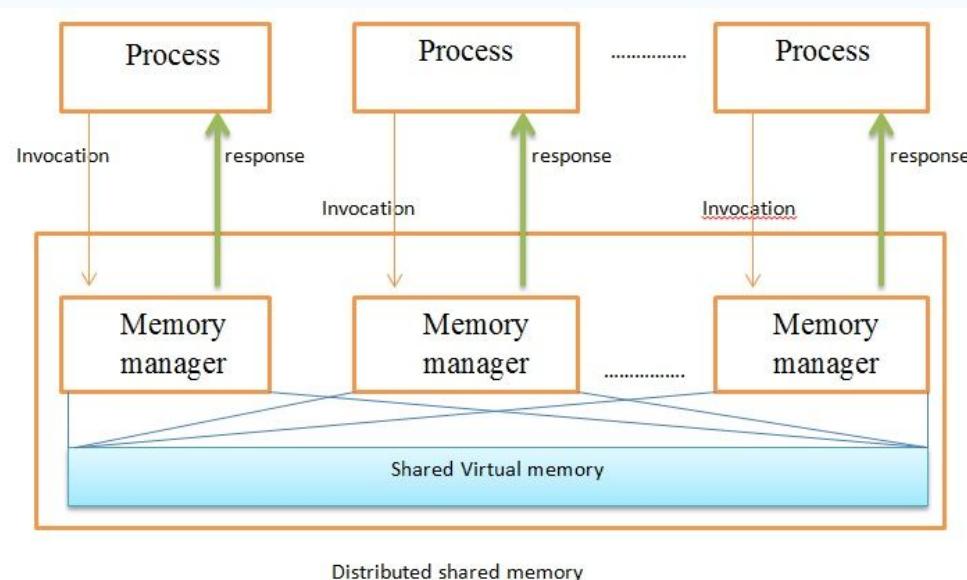
How might you accomplish this ?

Under the hood:

1. Scheduling
2. Work (in parallel)
3. Reduction
4. Barrier ...



Distributed Shared Memory (DSM)



Source: [Mehrnazhian](#)

- Concept introduced in 1989.
- DSM presents a logical shared memory for multi-computer systems.
- DSM maintains communication and data consistency for applications.
- Usually portions of local memory are mapped onto DSM.
- DSM maintains a directory service for all data residing in the system.

Distributed Shared Memory Management

DSM Management involves two main decisions:

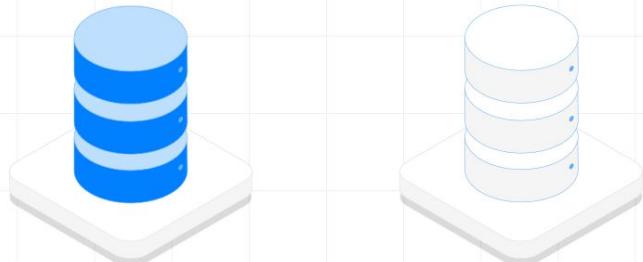
- How to distribute shared data ?
- How many readers and writers should be allowed for a shared data segment?



Distributing Shared Data

Replication

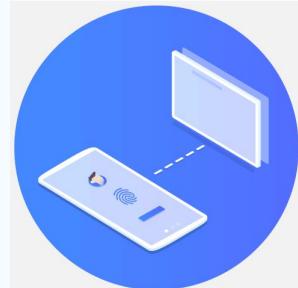
maintaining multiple copies of shared data at various locations



Migration

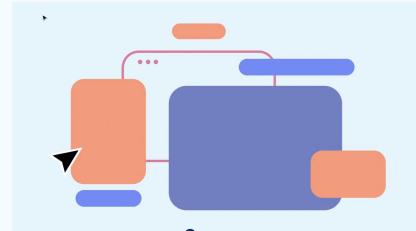
moving the copy of shared data to various locations
(only one copy of shared data is allowed)





DSM Performance

Issue to be considered



Thrashing

How to avoid thrashing,
i.e. the situation where
data constantly travels
between various locations?

Data Location

Which is the best original site
for shared data location?



Block Size Selection

What is the best size for sharable
data block?
(page size, packet size,
segment size)

Implementation Location

Where should be DSM
implemented?
(hardware or software or both)

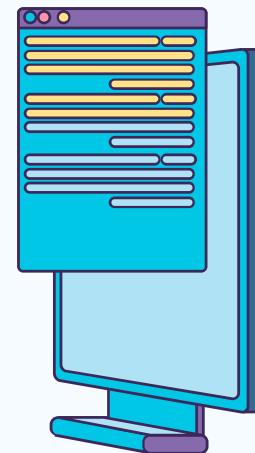
Distributed Memory Programming

Explicitly packaged data and sent it to another system in the cluster.



Message Passing Interface (MPI)

- Explicitly handle the decomposition of the problem across the cluster as well as make sure messages were sent and received in the proper order.



What is MPI ?



- Message passing is a programming model for coordination processes on these computers
- Each process has its own data
- Data is exchanged by sending and receiving messages
- Programmers use these tools to build multi-computer computation

Official MPI community:
[MPI Forum](#)

Exercises:
[UPM](#)

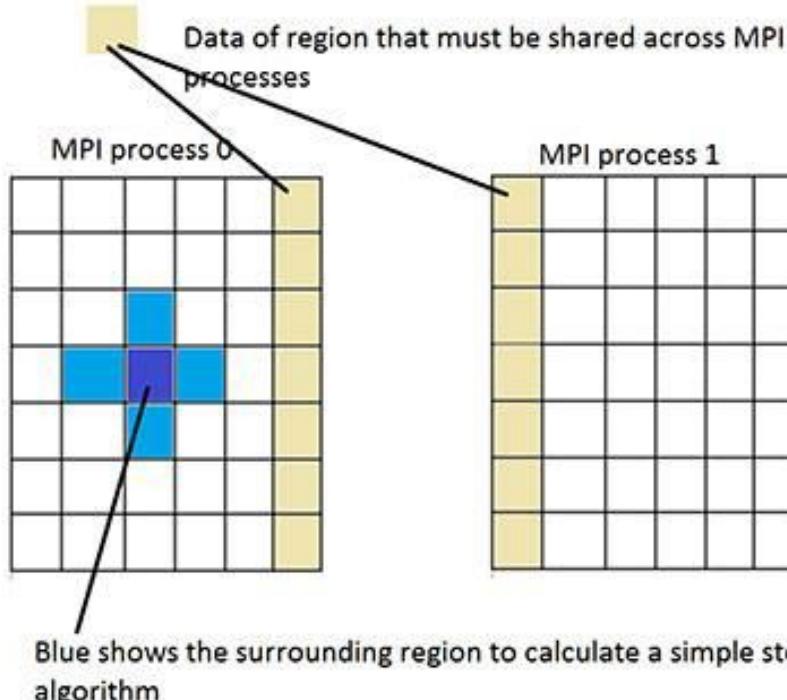
[LLNL](#)

[Rabernat \(Python\)](#)

[MPI Tutorial \(AWS\)](#)

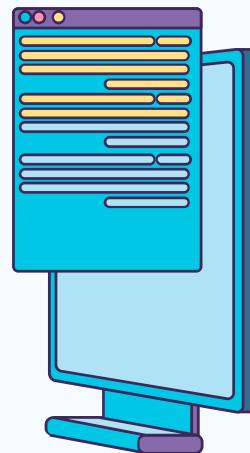
[Microsoft MPI](#)





C - calls for MPI process 0 to send and receive border data with MPI process 1

```
CALL MPI_ISEND(SBUF, scount, MPI_REAL, 1, stag, MPI_COMM_WORLD, ireq,ierr)  
CALL MPI_IRECV(RBUF,rcount,MPI_REAL,0,rtag,MPI_COMM_WORLD, rreq,irerr)
```



What is MPI ?



- MPI is a standard library for message passing.
- Ubiquitous in high performance technical computing.
- Nearly every big academic or commercial simulation or data analysis running on multiple nodes uses MPI directly or indirectly.

Official MPI community:
[MPI Forum](#)

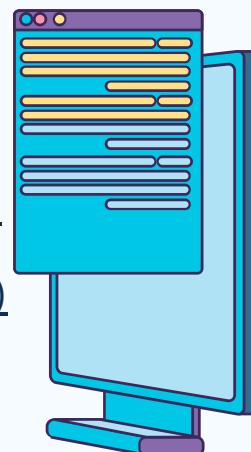
Exercises:
[UPM](#)

[LLNL](#)

[Rabernat \(Python\)](#)

[MPI Tutorial \(AWS\)](#)

[Microsoft MPI](#)



Why bother with MPI ?

Don't we have network libraries?

- Optimized for performance
- Will take advantage of fastest transport found:
 - Shared memory (Within a computer)
 - Fast cluster interconnects (Infiniband , Myrinet ..)between computers (nodes)
 - TCP/IP if all else fails

Official MPI community:
[MPI Forum](#)

Exercises:
[UPM](#)

[LLNL](#)

[Rabernat \(Python\)](#)

[MPI Tutorial \(AWS\)](#)

[Microsoft MPI](#)



Why bother with MPI ?

Don't we have network libraries?

- Optimized for performance.
- Will take advantage of fastest transport found.
- Enforces certain guarantees:
 - Reliable messages
 - In order arrival of messages

Official MPI community:
[MPI Forum](#)

Exercises:
[UPM](#)

[LLNL](#)

[Rabernat \(Python\)](#)

[MPI Tutorial \(AWS\)](#)

[Microsoft MPI](#)



Why bother with MPI ?

Don't we have network libraries?

- Optimized for performance.
- Will take advantage of fastest transport found.
- Enforces certain guarantees.
- Designed for multi-node technical computing:
 - Completely standard: Available everywhere
 - Has specialized routines for collective operations

Official MPI community:
[MPI Forum](#)

Exercises:
[UPM](#)

[LLNL](#)

[Rabernat \(Python\)](#)

[MPI Tutorial \(AWS\)](#)

[Microsoft MPI](#)



MPI basic functions

Set up / teardown

MPI_INIT
MPI_FINALIZE

More Examples:

PDC-support

By Jarno Rantaharju et al.

Who am I?

MPI_COMM_SIZE
MPI_COMM_RANK

Message Passing

MPI_SEND
MPI_RECV

Universiti of
Minnesota

By David Porter &
Mark Nelson

Princeton Plasma
Physics Lab

By Stephane Ethier

Question

**So is MPI the way
to do all distributed
computing ?**



A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
Int main (int argc, char *argv[]){
    Int rank, size ;
    MPI_Init(NULL NULL);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    MPI_Comm_size(MPI_COMM_WORLD , &size);
    printf("Hello! I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```



- **MPI_COMM_WORLD** indicates the set of all processes
- All MPI functions return error code
- Update values by passing pointers as arguments

A Simple MPI Program

```
#include "mpi.h"
#include <stdio.h>
Int main (int argc, char *argv[]){
    Int rank, size ;
    MPI_Init(NULL NULL);
    MPI_Comm_rank(MPI_COMM_WORLD , &rank);
    MPI_Comm_size(MPI_COMM_WORLD , &size);
    printf("Hello! I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

- **Compile and Run with cmd:**

```
mpi exec np 2 ./hello
```



```
Hello! I am 0 of 2
Hello! I am 1 of 2
```

What just happened ?

- mpiexec launched 4 processes on your computer
- Each process ran the program hostname
- Each ran independently
- Can do the same thing with 8, 27, or 93 processes
- For performance reasons, usually use as many processes as there are processors



... but Painful !

OpenMP

- Add pragmas to existing program
- Compiler + runtime system arrange for parallel execution
- Rely on shared memory for communication

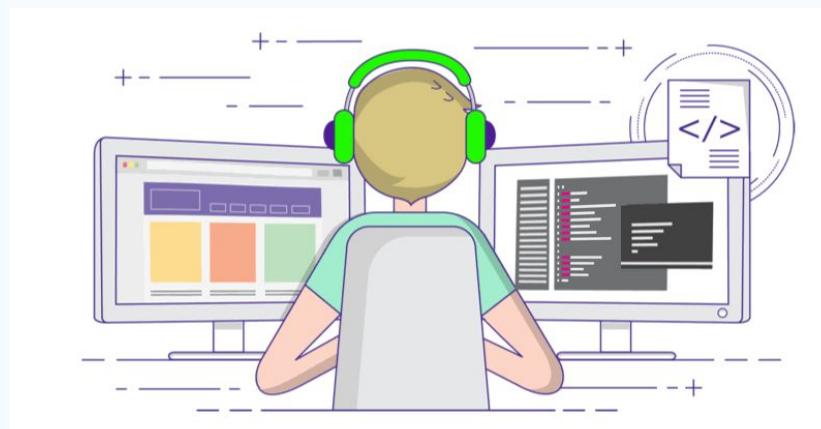
MPI

- Must rewrite program to describe how single process should operate on its data and communicate with other processes.
- Explicit data movement: programmer must say exactly what data goes where and when.
- Advantage: Can operate on systems that don't have shared memory.

MPI - Process Identification

When running with P processes:

- Size : P
Total number of processes
- Rank: Number between 0 and P 1
Identity of individual process
- Library Functions
`MPI_Comm_size`
`MPI_Comm_rank`



Comparing Frameworks



MPI

Fixed processes number created
All execute code starting with main
Isolated address spaces

Multiple Processes



FORK

Processes created during program execution
Replicate address space upon creation,
but then isolated



pthread_create

Threads created during program
execution
Shared address space

Multiple Threads



OpenMP

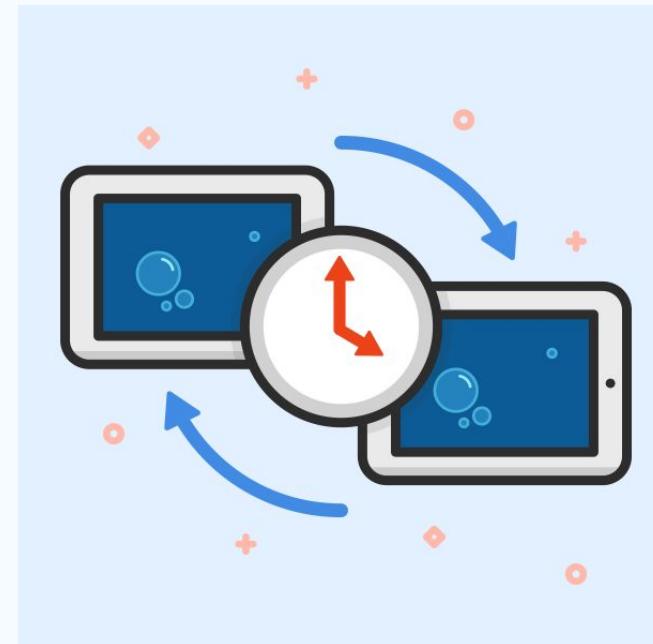
Set of threads created at beginning of program
Recruited to execute tasks spawned by
#pragma omp parallel
Shared address space

Synchronous Sending and Receiving

send(): call returns when sender receives acknowledgement that message data resides in address space of receiver.

recv(): call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender

Potential for deadlock if all processes attempt to send and then receive



Synchronous Sending and Receiving

Sender:

Call **SEND(foo)**

Copy data from buffer 'foo' in sender's address
space into network buffer

Send message

Receiver:

Call **RECV(bar)**

Receive message
Copy data into buffer 'bar' in receiver's
address space

Send ack

RECV() returns

Receive ack

SEND() returns

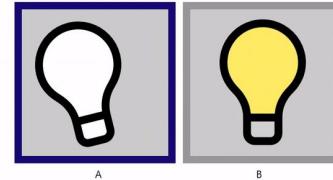
Asynchronous Sending and Receiving

send(): call returns immediately

- Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with process execution
- Calling thread can perform other work while waiting for message to be sent.

recv(): posts intent to receive in the future, returns Immediately

- Use checksend(), checkrecv() to determine actual status of send/receipt
- Calling thread can perform other work while waiting for message to be received



Asynchronous Sending and Receiving

Sender:

Call SEND(foo)

SEND returns handle h1

Copy data from 'foo' into network buffer

Send message

Call CHECKSEND(h1) // if message sent, now safe for thread to modify 'foo'

Receiver:

Call RECV(bar)

RECV(bar) returns handle h2

Receive message

Messaging library copies data into 'bar'

Call CHECKRECV(h2)

// if received, now safe for thread

// to access 'bar'

RED TEXT = executes concurrently with application thread

MPI Send / Receive Operation



Synchronous

`MPI_Send`

`MPI_Recv`



Asynchronous

`MPI_Irecv`

`MPI_Wait`

Memory Migration

Memory Migration requires two fundamental decisions:

- When in the migration process will we migrate memory?
- How much memory needs to be migrated?



Vector Add Example

Example

Steps to parallelize Vector Add with GPU

1. Make the variables accessible to both CPU and GPU
2. Launch configuration of the kernel function
3. Parallelize the kernel function to run on GPU

More Examples

Example

Matrix Transpose

<https://www.hpc.cineca.it/content/exercise-15>

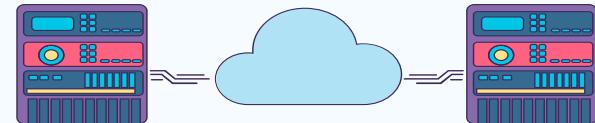
Matrix Multiplication

<https://www.hpc.cineca.it/content/exercise-16>

2D Laplace Equation

https://www.mcs.anl.gov/research/projects/mpi/tutorial/mpie_xmpl/src/jacobi/C/main.html

Summary of Chapter



- Shared address space

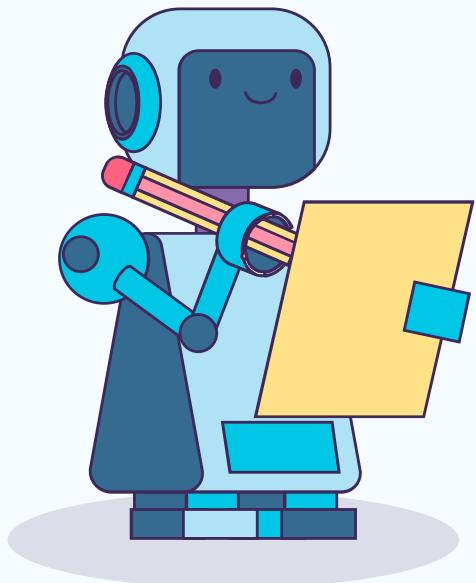
- > Communication is unstructured, implicit in loads and stores.
- > Natural way of programming, but can shoot yourself in the foot easily.
- > Program might be correct, but not perform well.

- Message passing

- > Structure all communication as messages.
- > Often harder to get first correct program than shared address space.
- > Structure often helpful in getting to first correct, scalable program.

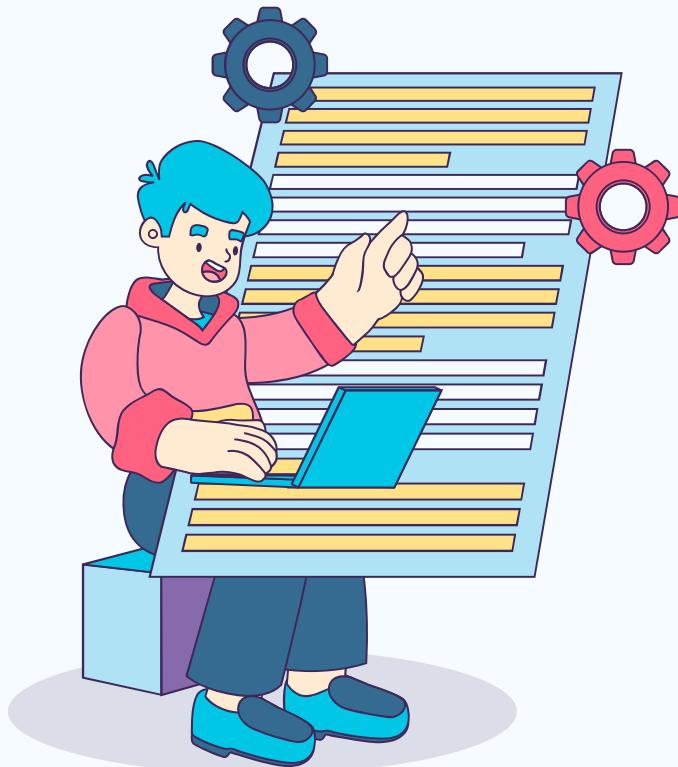
- Data parallel

- > Structure computation as a big “map” over a collection.
- > Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map (goal: preserve independent processing of iterations).
- > Modern embodiments encourage, but don’t enforce, this structure.



Thank you

CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)



BMCS3003 Distributed Systems and Parallel Computing

L04 - Concurrency Control (Part 1)

Presented by

Assoc Prof Ts Dr Tew Yiqi
May 2023



Table of contents

01

Mutual Exclusion &
Critical Region

02

Locks

03

Semaphores

01

Mutual Exclusion & Critical Region

A program object that blocks multiple users from accessing the same shared variable data at the same time.



Synchronization

Parallel processing (multi-processing) – 2 or more processors operate in unison.

- 2 or more CPUs are executing instructions simultaneously.
- Each CPU can have a process in RUNNING state at same time.
- Processor Manager has to coordinate activity of each processor and synchronize interaction among CPUs.

Synchronization is critical to a system's success because many things can go wrong in a multiprocessing system.



Example - Application of concurrent processing

Compute $A = 3 * B * C + 4 / (D+E) ** (F-G)$



COBEGIN

T1 = 3 * B

T2 = D+E

T3 = F-G

COEND

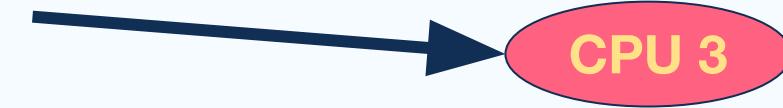
COBEGIN

T4 = T1 * C

T5 = T2 ** T3

COEND

A = T4+4/T5



Example - Application of concurrent processing

```
DO i = 1,3
```

```
    A(i) = B(i) + C(i)
```

```
ENDDO
```



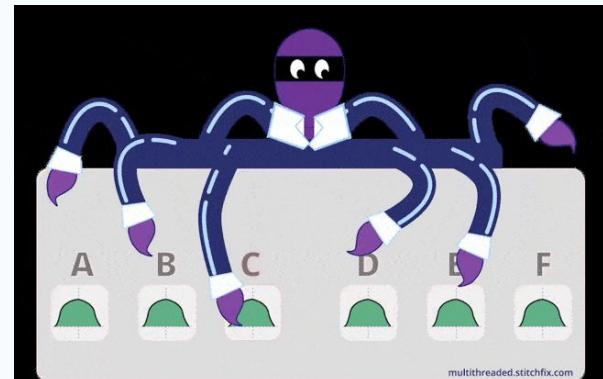
Processor 1 perform : $A(1) = B(1) + C(1)$

Processor 2 perform : $A(2) = B(2) + C(2)$

Processor 3 perform : $A(3) = B(3) + C(3)$

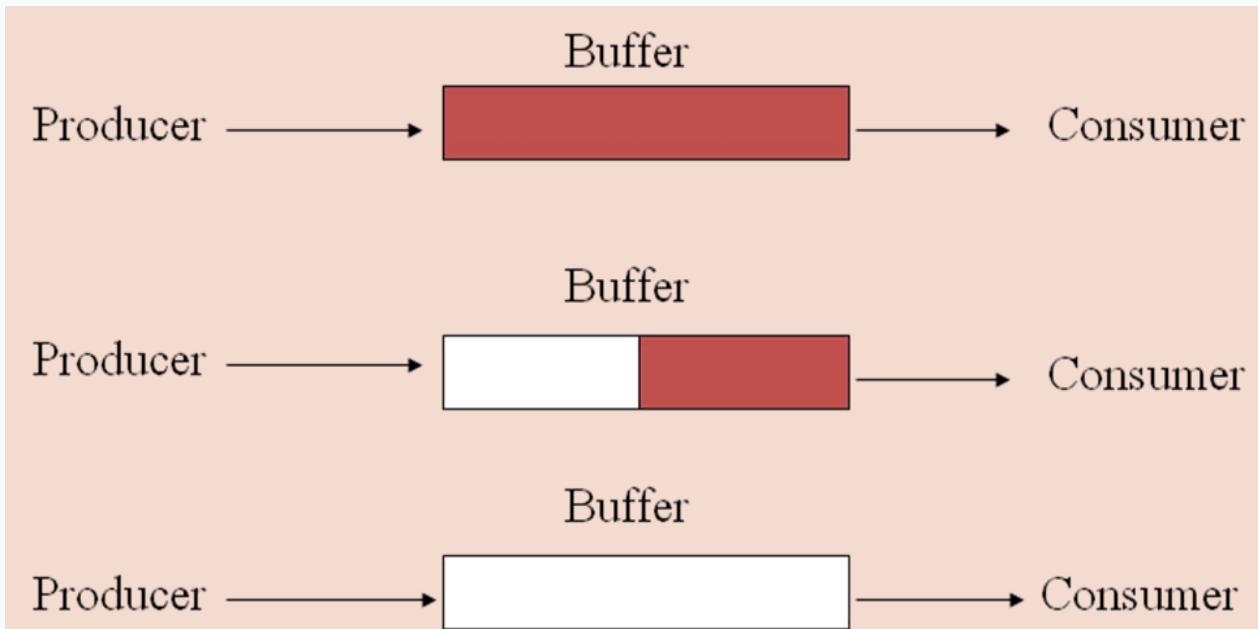
Classical Synchronisation Problems

- Common element in all synchronization schemes is to allow a process to finish work on a **critical region** of program before other processes have access to it.
 - Applicable to both multiprocessors and to 2 or more processes in a single-processor (time-shared) processing system.
 - Called a critical region because its execution must be handled as **ONE UNIT**.



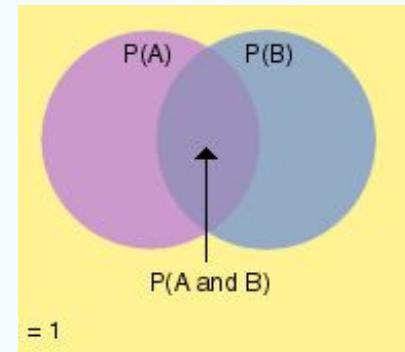
Producers and Customers

One Process Produces Some Data That
Another Process Consumes Later



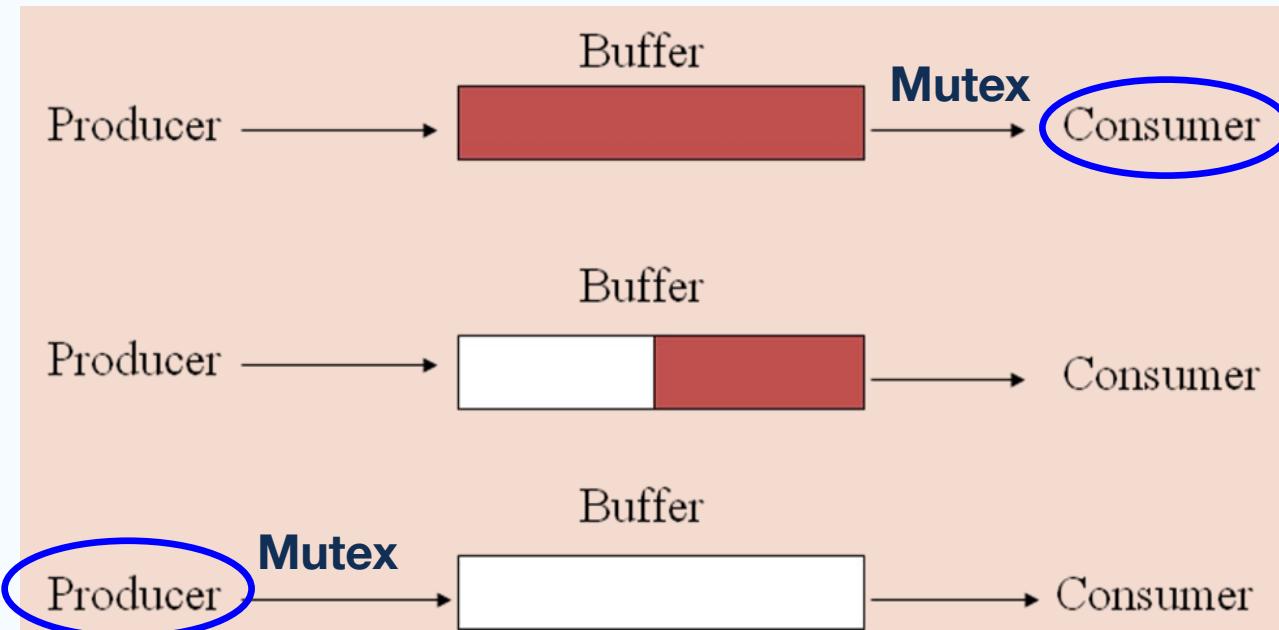
Mutual Exclusion (Mutex)

- Because buffer holds finite amount of data, synchronization process must delay producer from generating more data when buffer is full.
- Delay consumer from retrieving data when buffer is empty.
- This task can be implemented by 3 semaphores (signals):
 - Indicate number of full positions in buffer.
 - Indicate number of empty positions in buffer.
 - Mutex, will ensure mutual exclusion between processes (A **mutex** is a program object that is created so that multiple program thread can take turns sharing the same resource)



Producers and Customers

One Process Produces Some Data That
Another Process Consumes Later

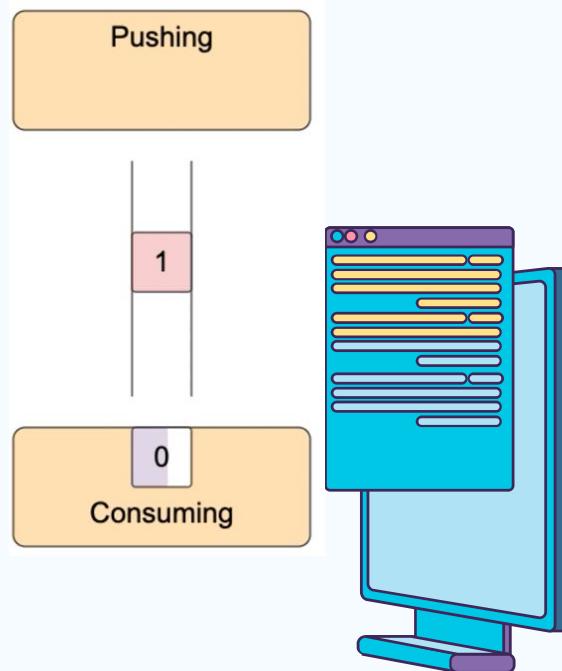


Producers and Customers

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

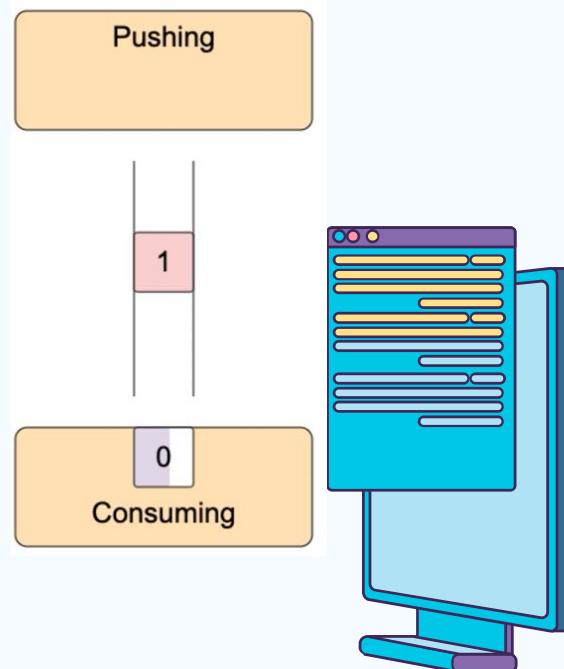
Producers

```
while (true)
{
    -- Produce an item and put in nextProduced
    while (count == BUFFER_SIZE)
        ; // do nothing
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```



Customers

```
while (true)
{
    while (count == 0)
        ; // do nothing
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    -- Consume the item in nextConsumed
}
```

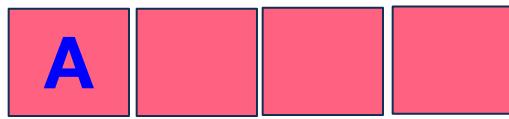


count

0

in
↓

Buffer



0 1 2 3

out
↑

Producer

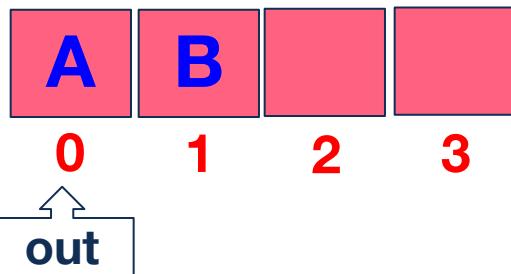
```
while (true)      {  
    /* produce an item and put in nextProduced  
     *  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

1

in
↓

Buffer



Producer

```
while (true)      {  
    /* produce an item and put in nextProduced  
     *  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

2

in

Buffer



0 1 2 3

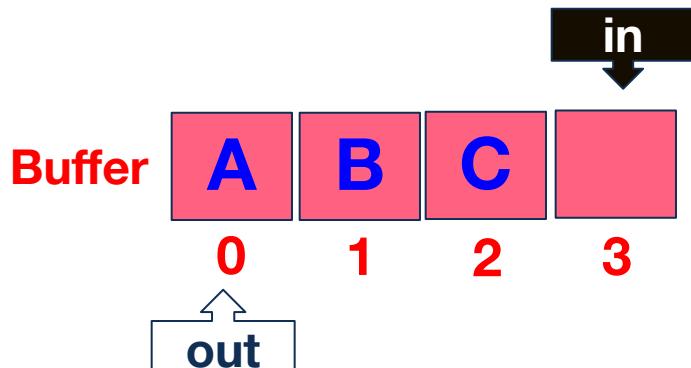
out

Producer

```
while (true)      {  
    /* produce an item and put in nextProduced  
     *  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

3

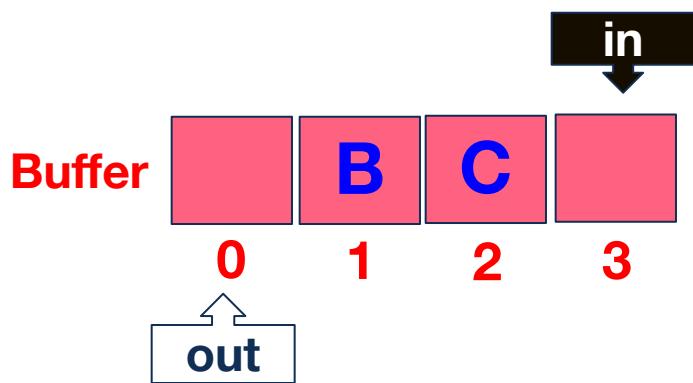


Producer

```
while (true)      {  
    /* produce an item and put in nextProduced  
     *  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

3



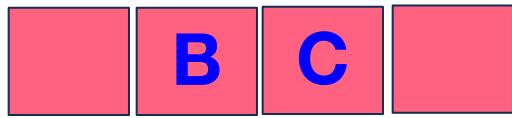
Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

count

2

Buffer



0 1 2 3

out

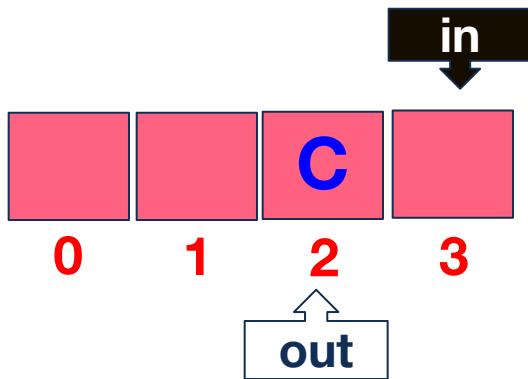
Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

count

1

Buffer



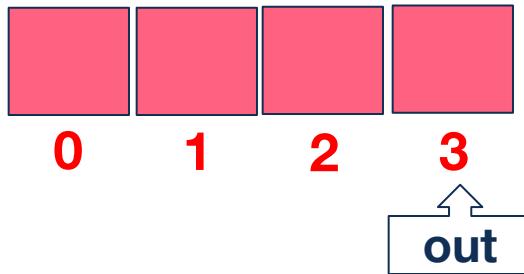
Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

count

0

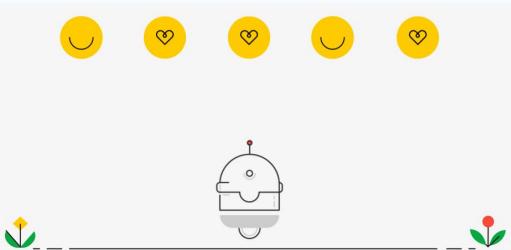
Buffer



Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in nextConsumed */  
}
```

a. Producer-Consumer Race Condition



➤ `count++` is actually implemented as
`register1 = count`
`register1 = register1 + 1`
`count = register1`

➤ `count--` is actually implemented as
`register2 = count`
`register2 = register2 - 1`
`count = register2`

Consider this execution interleaving with “`count = 5`” initially:

- t0: producer execute `register1 = count`
- t1: producer execute `register1 = register1 + 1`
- t2: consumer execute `register2 = count`
- t3: consumer execute `register2 = register2 - 1`
- t4: producer execute `count = register1`
- t5: consumer execute `count = register2`

{ <code>register1 = 5</code> }
{ <code>register1 = 6</code> }
{ <code>register2 = 5</code> }
{ <code>register2 = 4</code> }
{ <code>count = 6</code> }
{ <code>count = 4</code> }



Illustration

Count

5



Producer

```
register1 = count  
register1 = register1 + 1  
count     = register1
```

← t0

Consumer

```
register2 = count  
register2 = register2 - 1  
count     = register2
```

Illustration

Count

5



Producer

```
register1 = count  
register1 = register1 + 1 ← t1  
count    = register1
```

Consumer

```
register2 = count  
register2 = register2 - 1  
count    = register2
```

Illustration

Count

5



Producer

```
register1 = count
```

```
register1 = register1 + 1
```

```
count = register1
```

t2 → register2 = count

```
register2 = register2 - 1
```

```
count = register2
```

Illustration

Count

5



Producer

```
register1 = count  
register1 = register1 + 1  
count     = register1
```

Consumer

```
register2 = count  
register2 = register2 - 1  
count     = register2
```

t3

→

Illustration

Count

6



Producer

```
register1 = count  
register1 = register1 + 1  
count     = register1
```

← t4

Consumer

```
register2 = count  
register2 = register2 - 1  
count     = register2
```

Illustration

Count

4



Producer

```
register1 = count
```

```
register1 = register1 + 1
```

```
count = register1
```

t5 → count

Consumer

```
register2 = count
```

```
register2 = register2 - 1
```

```
= register2
```

Elaboration

- If the producer and consumer run concurrently the value of the variable counter may be **4** or **6** (the value of counter is 6 when we reversed the order of the statements at t4 and t5)
- Incorrect state occurs because we allowed both processes to manipulate the variable **counter** concurrently.
- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which access take place is called a **Race Condition**

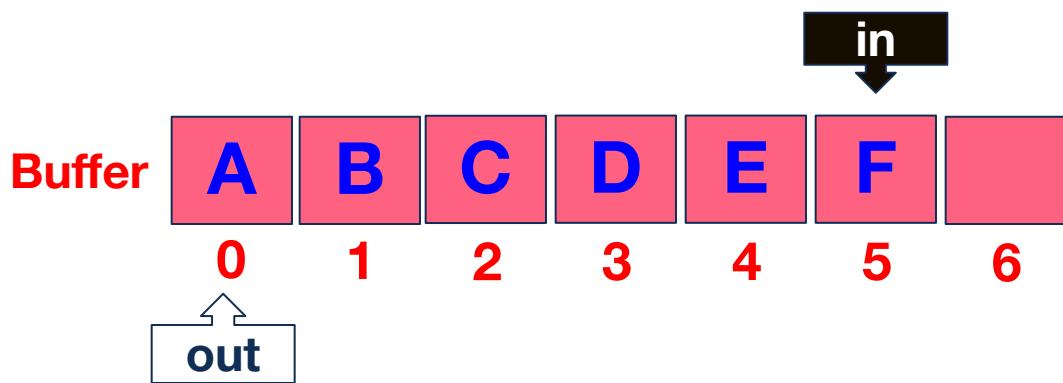
Elaboration

- In fact, the producer program has to finish its execution before the consumer program to print or display the result, so the counter value suppose is 5. Which is generated correctly if producer and consumer execute separately.



count

5



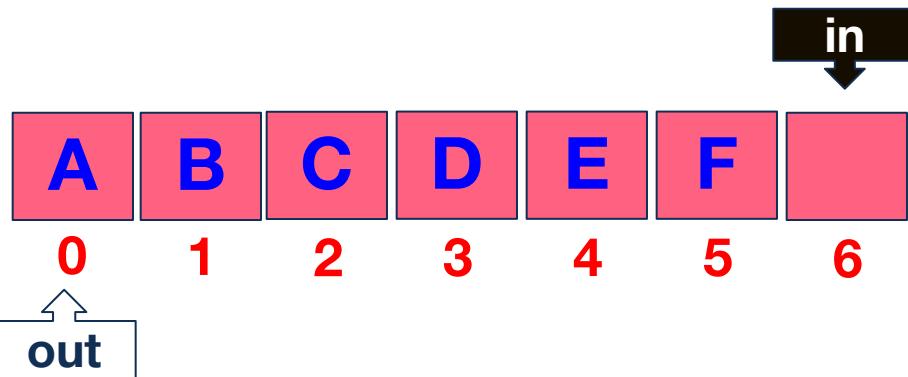
Producer

```
while (true)      {  
    /* produce an item and put in  
    nextProduced  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

6

Buffer



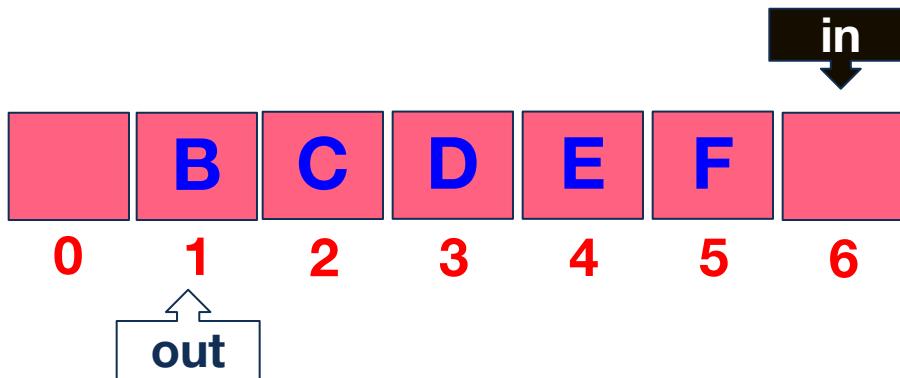
Producer

```
while (true)      {  
    /* produce an item and put in  
     * nextProduced  
    */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

count

5

Buffer



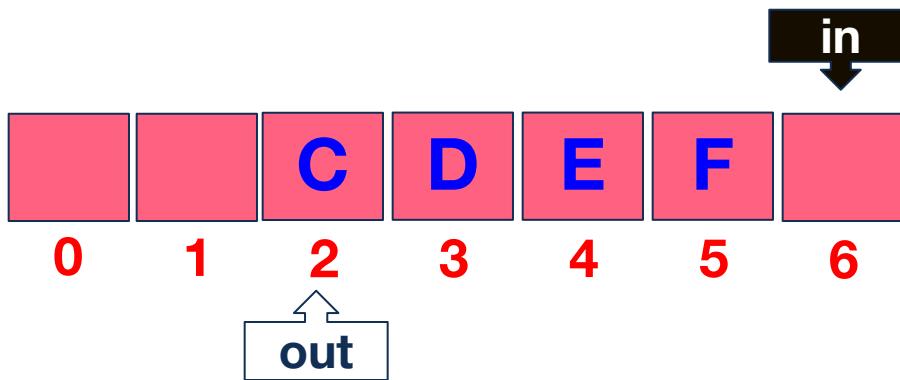
Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in  
     * nextConsumed  
    */  
}
```

count

4

Buffer



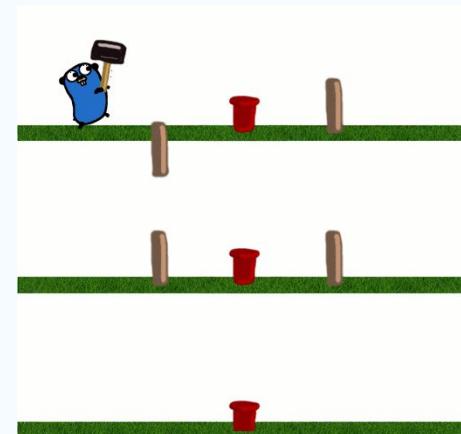
Consumer

```
while (true)      {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* consume the item in  
    nextConsumed  
}
```

b. Print Spooler Race Condition

- A process wants to print a file, it enter the file name in **spooler** directory.
- Another process, the printer **daemon***, periodically checks to see if there are any files to be printed, then prints them and removes their names from the directory.
- Imagine that our spooler directory has very large number of slots, numbered 0,1,2,3,..., each one capable of holding a file name.
- There are two shared variables **out** and **in**.
 - **out** which points to the next file to be printed.
 - **in** is point to next free slot in directory.
- Process A and Process B decide they want to queue a file for printing.

* A **daemon** is a program that waits for another program to ask it to do something.



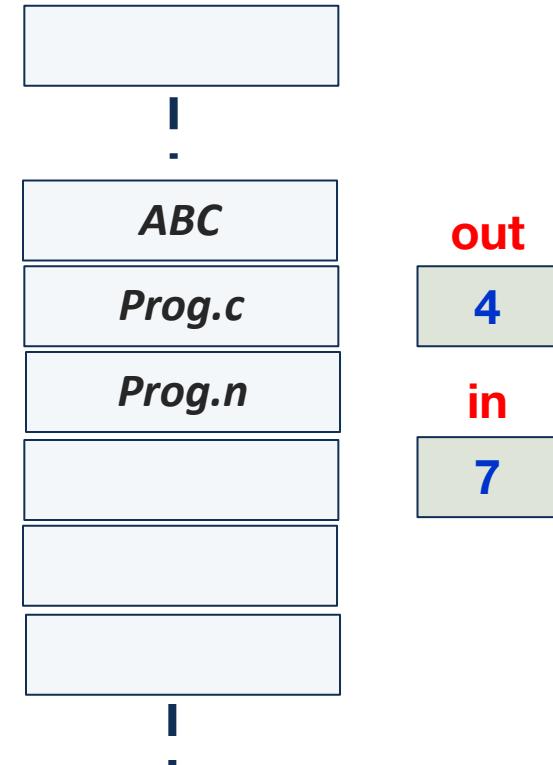
Process A

```
read in  
store into A_NextFreeSlot  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



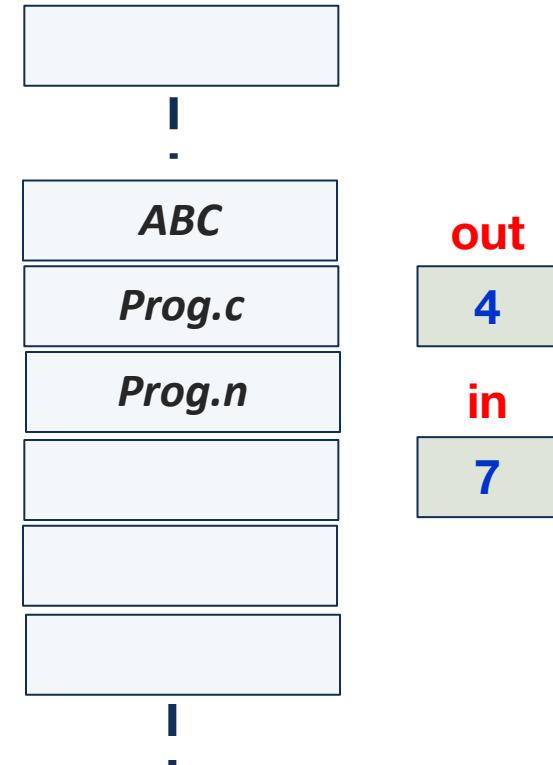
Process A

```
read in  
store into A_NextFreeSlot7 ←  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



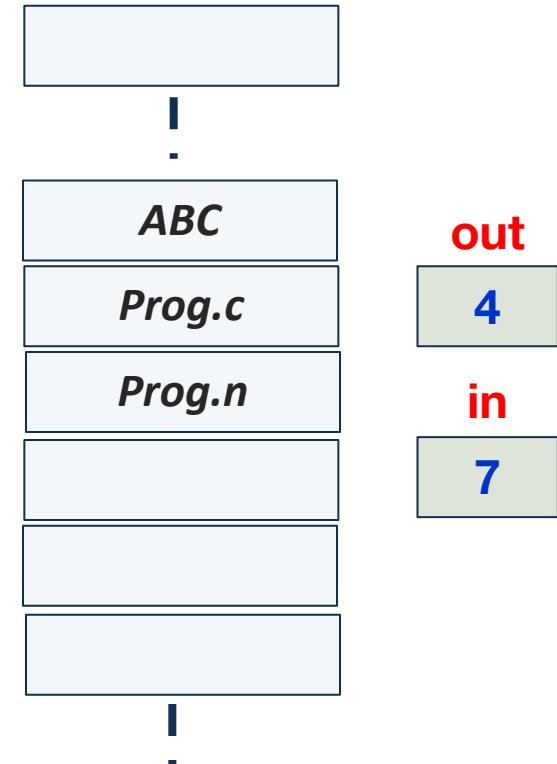
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



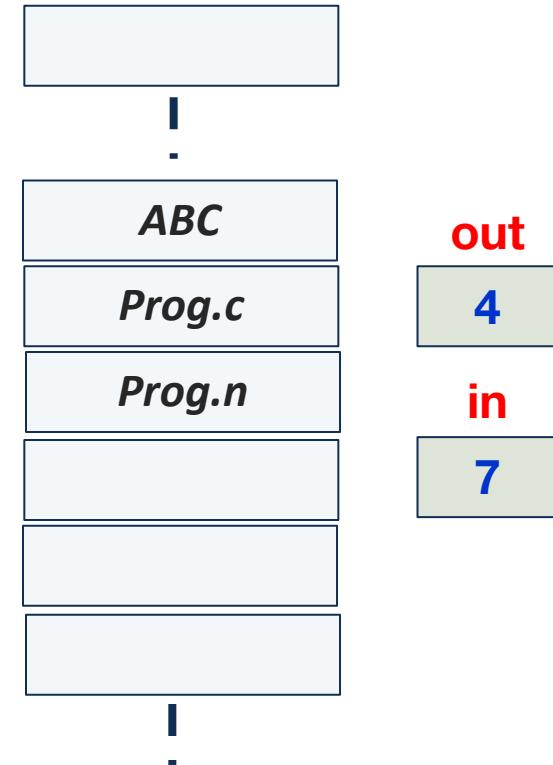
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



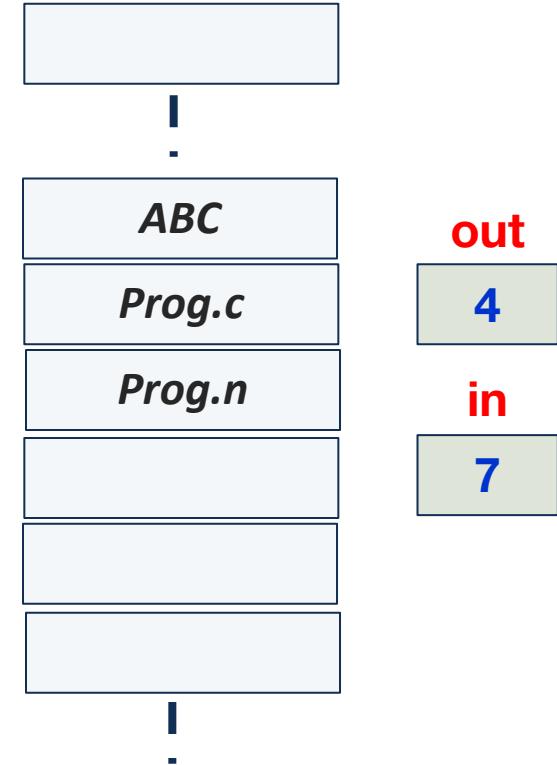
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot7 ←  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



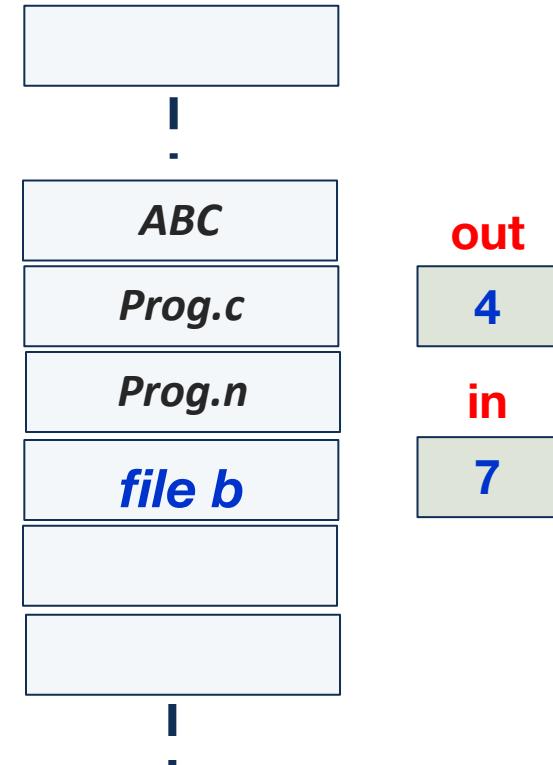
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot7  
add file b into the buffer ←  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



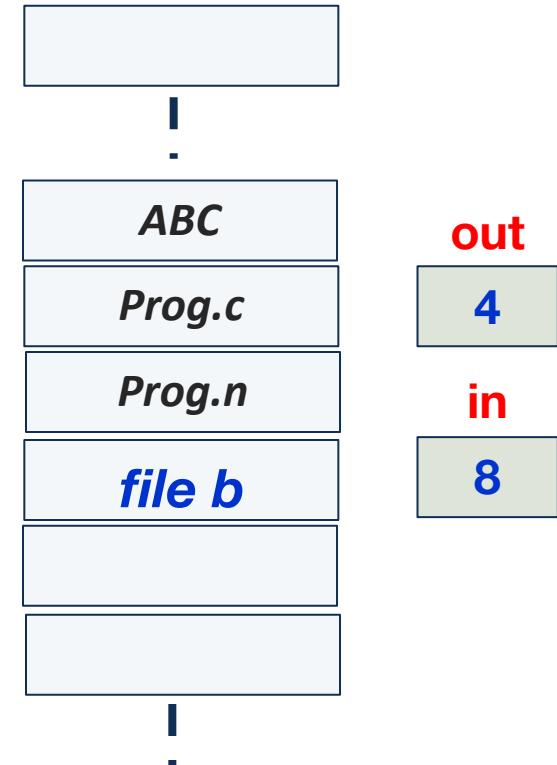
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



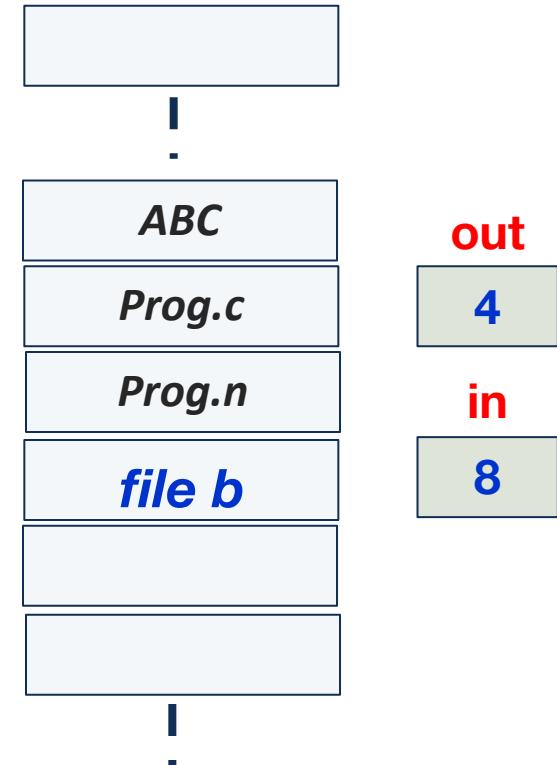
Process A

```
read in  
store into A_NextFreeSlot7  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

Spooler Directory/ buffer



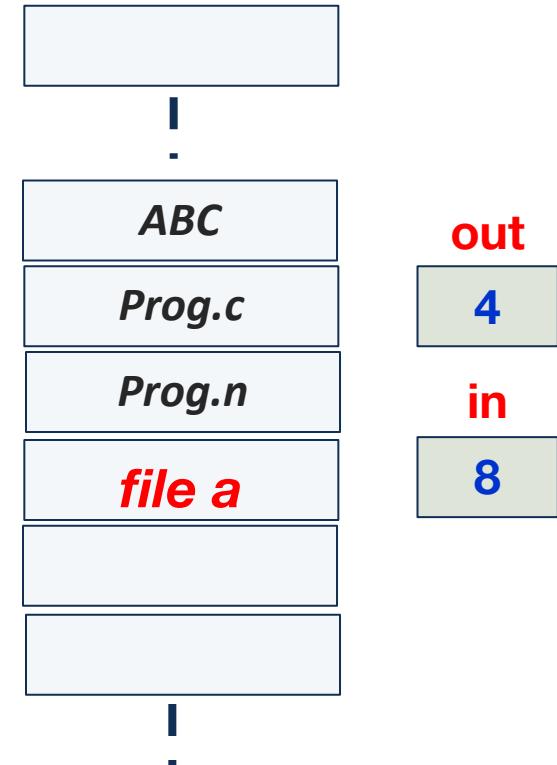
Process A

read in
store into A_NextFreeSlot⁷
add *file a* into the buffer ←
 $\text{in} = \text{A_NextFreeSlot} + 1$

Process B

read in
store into B_NextFreeSlot
add *file b* into the buffer
 $\text{in} = \text{B_NextFreeSlot} + 1$

Spooler Directory/ buffer



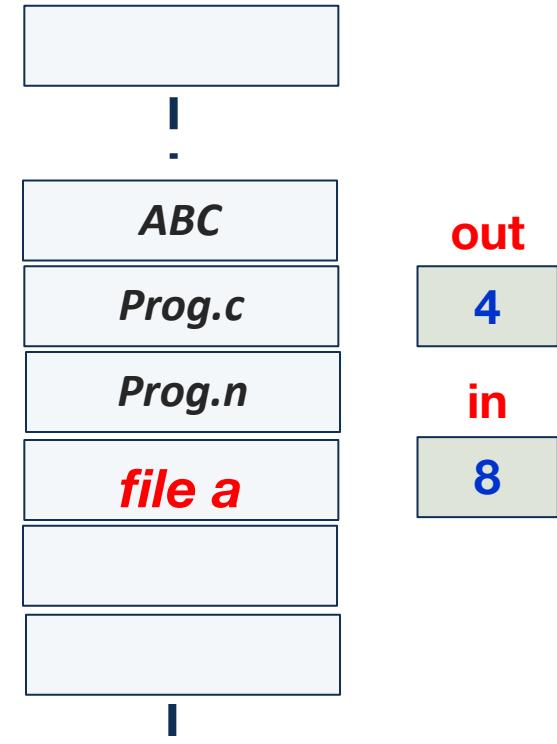
Process A

```
read in  
store into A_NextFreeSlot  
add file a into the buffer  
in = A_NextFreeSlot + 1
```

Process B

```
read in  
store into B_NextFreeSlot  
add file b into the buffer  
in = B_NextFreeSlot + 1
```

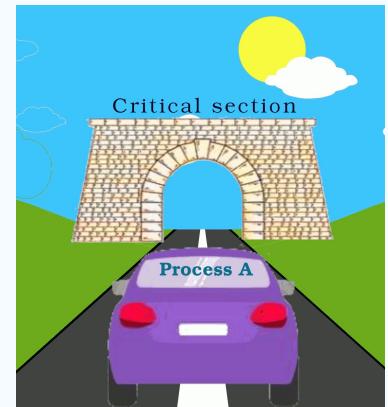
Spooler Directory/ buffer



Process B will never receive any output

The Critical Section Problem

- Each process has a segment of code called a **critical section**, in which the process may be changing common variables, updating a table, writing a file and so on.
- The important feature of the system is have to ensure that no other processes are executing its critical section when one particular process is executing its critical section.
- Processes cooperation protocol, whereby each process must request permission to enter its critical section. There is the entry section, then followed by exit section after that do the remaining code in the remainder section.



The Critical Section Problem

Example:

Shared data:

```
int Balance;
```

Process A:

.....

enter section

```
Balance = Balance - 200;
```

exit section

remainder section

Where is the Critical-section ?

Process B:

.....

entry section

```
Balance = Balance - 100;
```

exit section

remainder section

The Critical Section Problem

Example:

Shared data:

```
int Balance;
```

Process A:

.....

enter section

Balance = Balance - 200;

exit section

remainder section

Process B:

.....

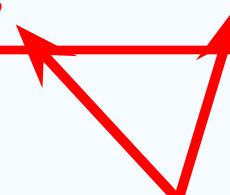
entry section

Balance = Balance - 100;

exit section

remainder section

Critical-section



Synchronization solutions

Solution to Critical-Section requirements

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. **Progress** - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely (unclear reasons).
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
 - Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes

Peterson's Algorithm

In Process synchronization

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted (privileged instructions).
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!

Check the resources on C : geeksforgeeks.org

Critical Section

```
do {
```

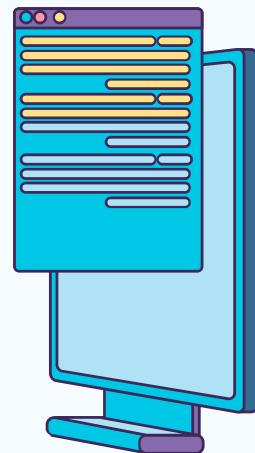
```
    Entry section
```

```
    Critical section
```

```
    Exit section
```

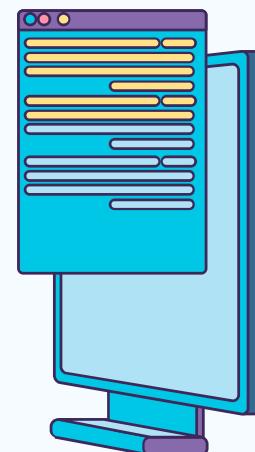
```
    Remainder section
```

```
} while (TRUE);
```



Algorithm for Process P_i

```
do {  
    flag[i] = TRUE;           Entry section  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
  
    flag[i] = FALSE;          Exit section  
    Remainder section  
} while (TRUE);
```



Algorithm for Process P_i

flag	FALSE	FALSE
-------------	-------	-------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_0

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	TRUE	FALSE
-------------	------	-------

```
do {                                do {  
    flag[i] = TRUE;                flag[i] = TRUE;  
    turn = j;                      turn = j;  
    while (flag[j] && turn == j);  while (flag[j] && turn == j);  
    Critical section               Critical section  
    flag[i] = FALSE;              flag[i] = FALSE;  
    Remainder section            Remainder section  
} while (TRUE);                    } while (TRUE);
```

P_o

P_1



Algorithm for Process P_i

flag	TRUE	FALSE
-------------	------	-------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_o

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_1



Algorithm for Process P_i

flag	TRUE	TRUE
-------------	------	------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_o

```
do {  
    flag[i] = TRUE; ←  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag

TRUE

TRUE

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_o

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_1

Algorithm for Process P_i

flag	TRUE	TRUE
------	------	------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_0

P_1

Algorithm for Process P_i

flag	FALSE	TRUE
-------------	-------	------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```



P_0

P_1

Algorithm for Process P_i

flag	FALSE	TRUE
-------------	-------	------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_o

P_1

Algorithm for Process P_i

flag	FALSE	TRUE
-------------	-------	------

```
do {                                do {  
    flag[i] = TRUE;                  flag[i] = TRUE;  
    turn = j;                        turn = j;  
    while (flag[j] && turn == j);    while (flag[j] && turn == j);  
    Critical section                 Critical section  
    flag[i] = FALSE;                flag[i] = FALSE;  
    Remainder section               Remainder section  
} while (TRUE);                      } while (TRUE);
```

 P_0 P_1

Algorithm for Process P_i

flag	FALSE	FALSE
-------------	-------	-------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_o

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```



P_1

Algorithm for Process P_i

flag	FALSE	FALSE
-------------	-------	-------

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

P_0

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
    Critical section  
    flag[i] = FALSE;  
    Remainder section  
} while (TRUE);
```

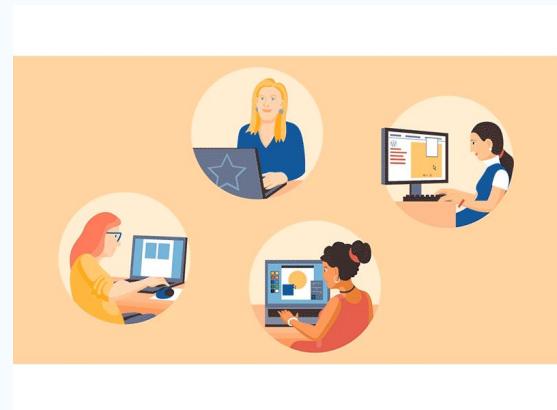
P_1



Synchronization Hardware (1/3)

Disable Interrupt

- The critical-section problem could be solved simply in a uniprocessor environment, if we could prevent interrupts from occurring while a shared variable was being modified.
- The current sequence of instructions would be allowed to execute in order without preemption.
- This is often the approach taken by non preemptive kernel.



Synchronization Hardware (2/3)

Disable Interrupt

- a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.
- Unfortunately, this solution is not as feasible in a multiprocessors environment, because disabling interrupts on a multiprocessors can be time consuming **as the message is passed to all the processors.**



Synchronization Hardware (3/3)

Disable Interrupt

why not disable interrupts?

- This message passing delays entry into each critical-section and the system efficiency decreases.
- Furthermore, disabling interrupts affect only the CPU that executed the disable instruction, where the other processor will continue running and can access the shared memory.



02

Locks (Spin Locks)

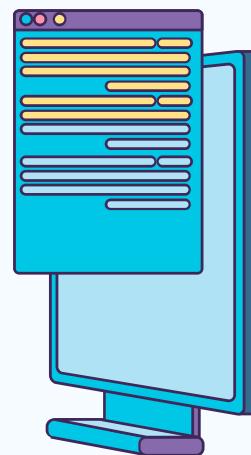
A synchronization primitive that enforces limits on access to a resource when there are many threads of executions.



Solution to Critical-section Problem Using Locks

Many systems provide hardware support for critical section code:

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```



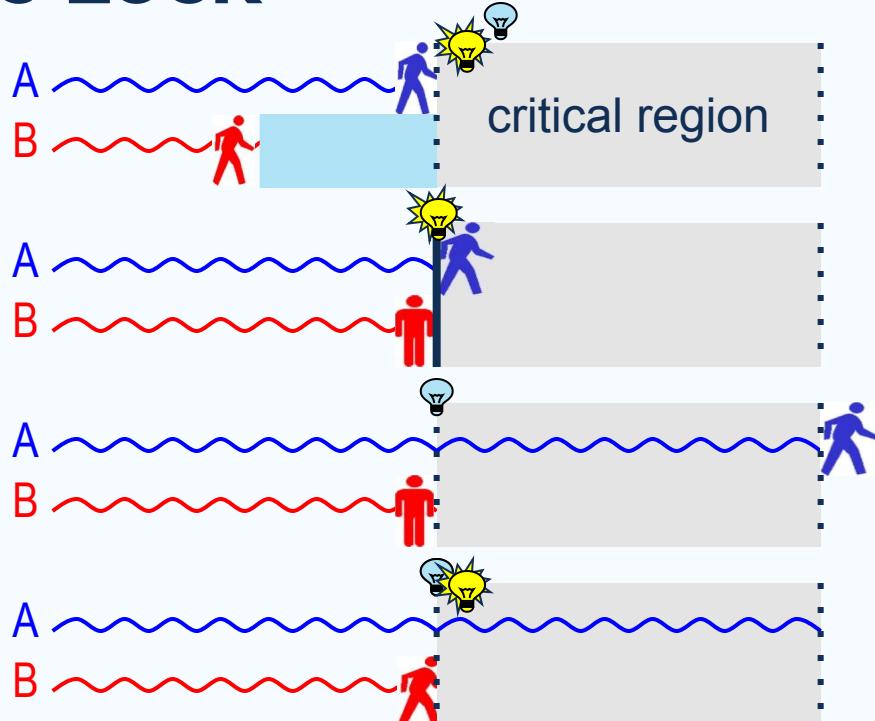
Lock

- A single shared variable (lock) initially 0.
- When the process wants to enter its critical-section, it first tests the lock. **If the lock is 0**, the process set it to 1 and enter the critical-section.
- If the lock is already 1, the process just wait until it becomes 0.
- Disadvantages: Some threads/processes have to wait until a lock is released. If one of the threads holding a lock dies, stalls, blocks, or enters an infinite loop, other threads waiting for the lock may wait forever.



Atomic Lock

1. **thread A reaches CR and finds the lock at 0 and sets it in one shot, then enters**
2. **even if B comes right behind A, it will find that the lock is already at 1**
3. **thread A exits CR, then resets lock to 0**
4. **thread B finds the lock at 0 and sets it to 1 in one shot, just before entering CR**



03

Semaphore

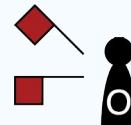
A variable or abstract data type used to control access to a common resource by multiple threads and avoid critical section problems.



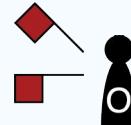
Semaphore

- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via **two indivisible (atomic) operations**

```
wait (S) {  
    while S ≤ 0;  
    // no-op  
    S--;  
}
```

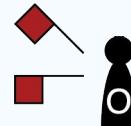


```
signal (S) {  
    S++;  
}
```



Semaphore as General Synchronization Tool

- Provide mutual exclusion

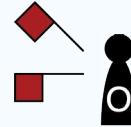


```
Semaphore S; //initialized to 1
```

```
wait (S);
```

Critical Section

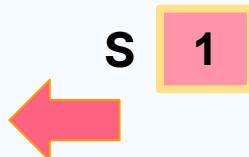
```
signal (S);
```



Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```



```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
signal (S) {  
    S++;  
}
```

Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

S 0

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

```
signal (S) {  
    S++;  
}
```

Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

S 0

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

```
signal (S) {  
    S++;  
}
```

Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

S 1

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```



```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

```
signal (S) {  
    S++;  
}
```

Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

S 1

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
signal (S) {  
    S++;  
}
```



Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

S 0

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
signal (S) {  
    S++;  
}
```



Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

S 0

```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100; ←  
signal(S);  
remainder sessions
```

```
signal (S) {  
    S++;  
}
```

Semaphore with two individual atomic operation

```
int balance, S;  
S = 1;  
// Process A  
...  
wait(S);  
balance = balance - 200;  
signal(S);  
remainder sessions
```

```
wait (S) {  
    while S ≤ 0 ; // no-op  
    S--;  
}
```

S 1

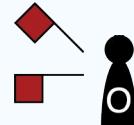
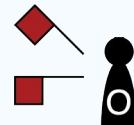
```
int balance, S;  
S = 1;  
// Process B  
...  
wait(S);  
balance = balance - 100;  
signal(S);  
remainder sessions
```

```
signal (S) {  
    S++;  
}
```



Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
 - Could now have busy waiting in critical section implementation
 - But implementation code is short
 - **Little busy waiting** if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.



Semaphore Bounded-Buffer

N buffers, each can hold one item

**Semaphore
Mutex**

Initialise to the
value 1

**Semaphore
Full**

Initialise to the
value 0

**Semaphore
Empty**

Initialise to the
value N

Semaphore in Producer-Consumer

Structure Process of Producer

```
do {  
    // produce an item  
    wait (empty);  
    wait (mutex);  
  
    // add item to buffer  
  
    signal (mutex);  
    signal (full);  
} while (true);
```

Structure Process of Consumer

```
do {  
    wait (full);  
    wait (mutex);  
    // remove item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
} while (true);
```

mutex

empty

full

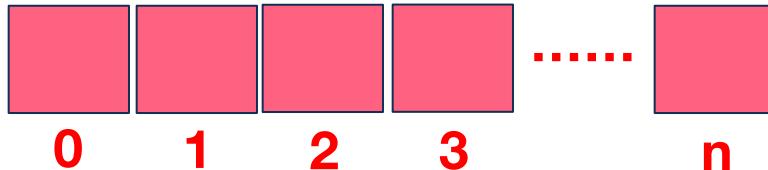
1

n

0

Next P

Buffer



0

1

2

3

n

Next C

Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

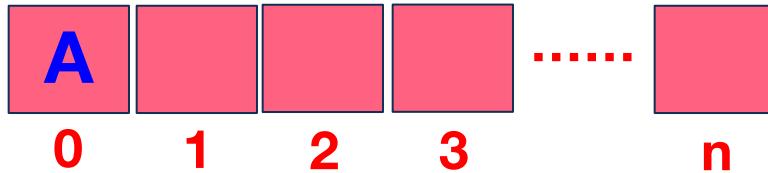
1

n

0

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

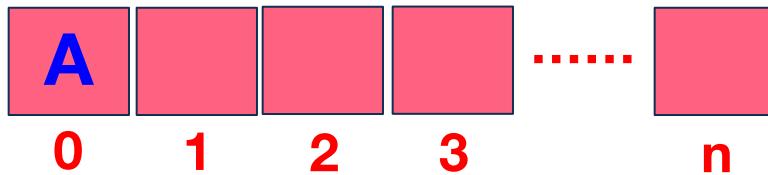
1

n-1

0

Next P
↓

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

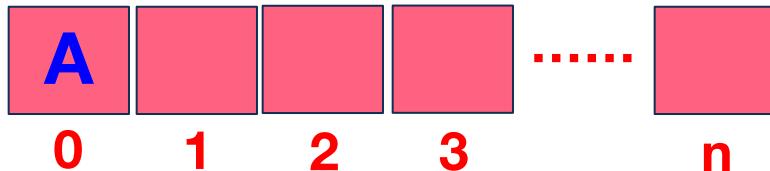
0

n-1

0

Next P
↓

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

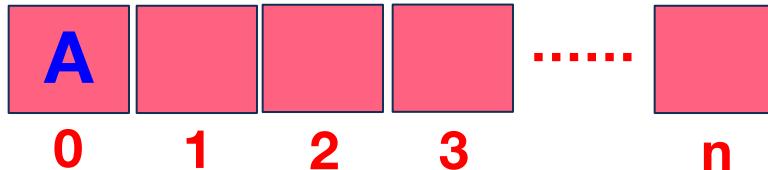
0

n-1

0

Next
P

Buffer



Next
C

Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

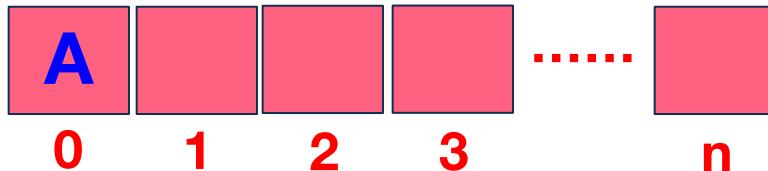
1

n-1

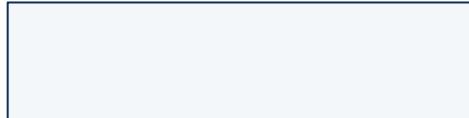
0

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    Wait(empty);  
    Wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

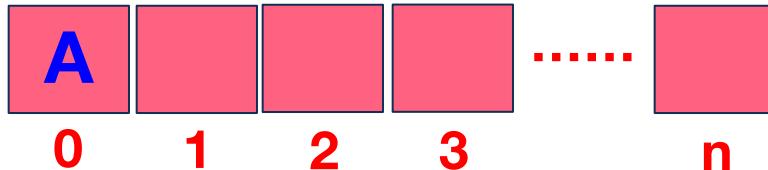
1

n-1

1

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

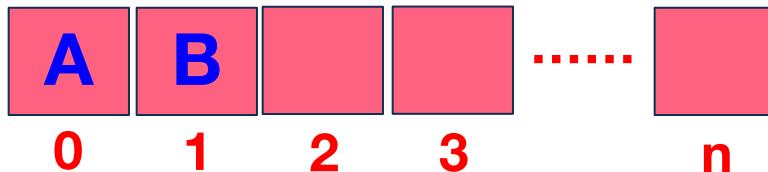
1

n-1

1

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

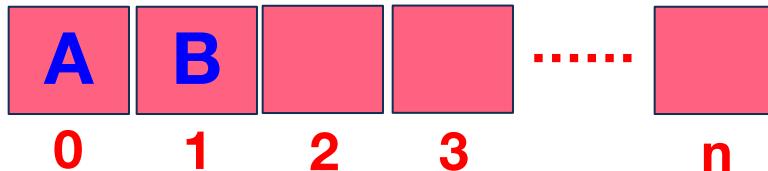
1

n-2

1

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

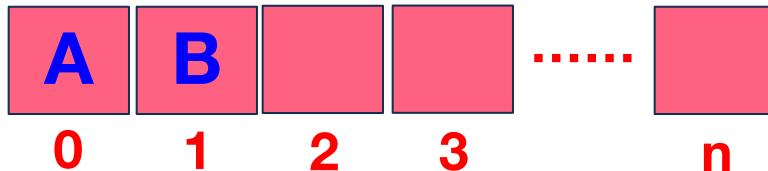
0

n-2

1

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

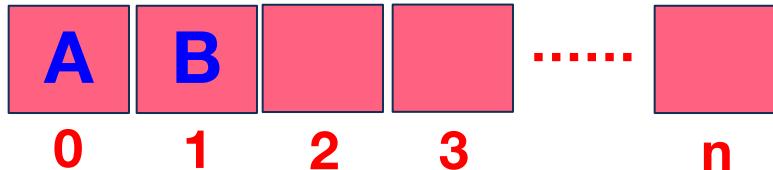
0

n-2

1

Next
P

Buffer



Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

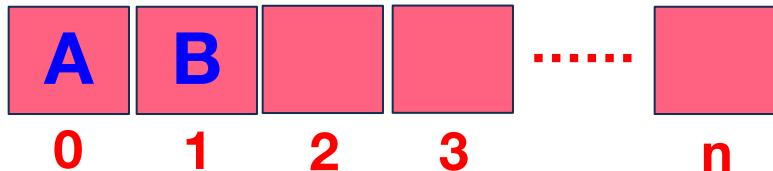
1

n-2

1

Next
P

Buffer



Display



Producer

```
do{
    //produce an item in nextp
    wait(empty);
    wait(mutex);
    //add nextp to buffer
    signal(mutex);
    signal(full);
}(true)
```

Consumer

```
do{
    wait(full);
    wait(mutex);
    //remove item from buffer to nextc
    signal(mutex);
    signal(empty);
    //consume the item in nextc
}(true)
```

mutex

empty

full

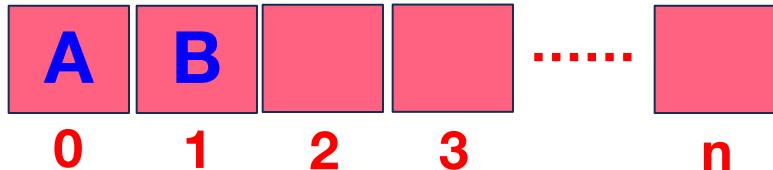
1

n-2

2

Next
P

Buffer



0 1 2 3

n

Next
C

Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

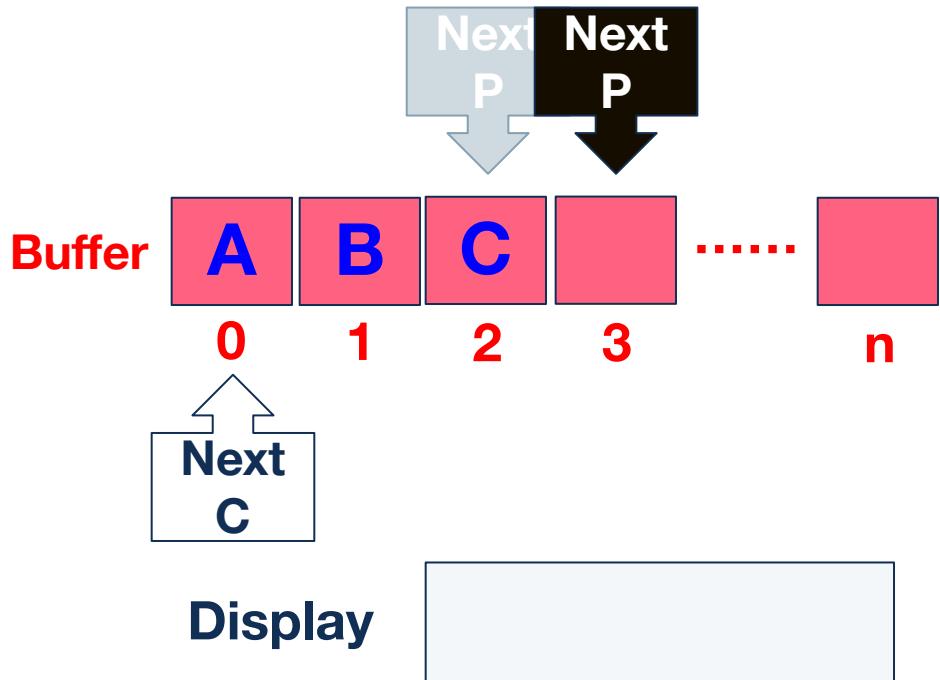
Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    wait(full);  
    wait(mutex);  
    //remove item from buffer to nextc  
    signal(mutex);  
    signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

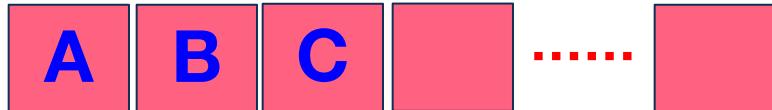
1

n-3

2

Next
P

Buffer



0 1 2 3

n

Next
C

Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    → Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

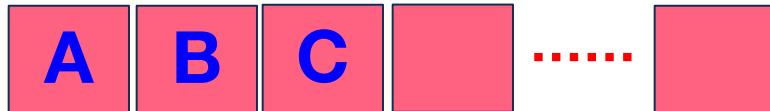
0

n-3

2

Next
P

Buffer



Next
C

Display



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

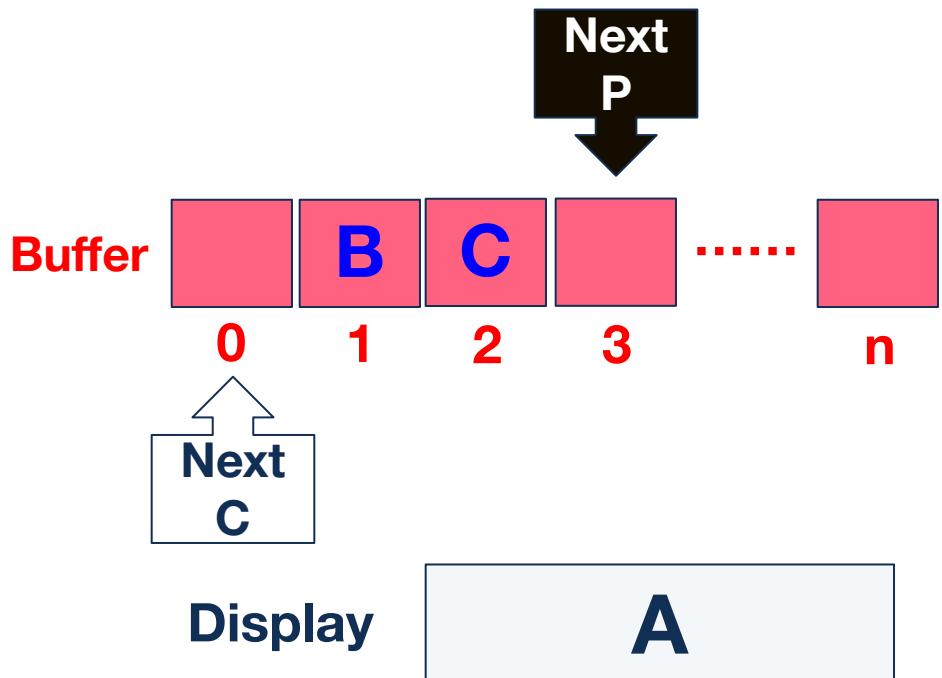
Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

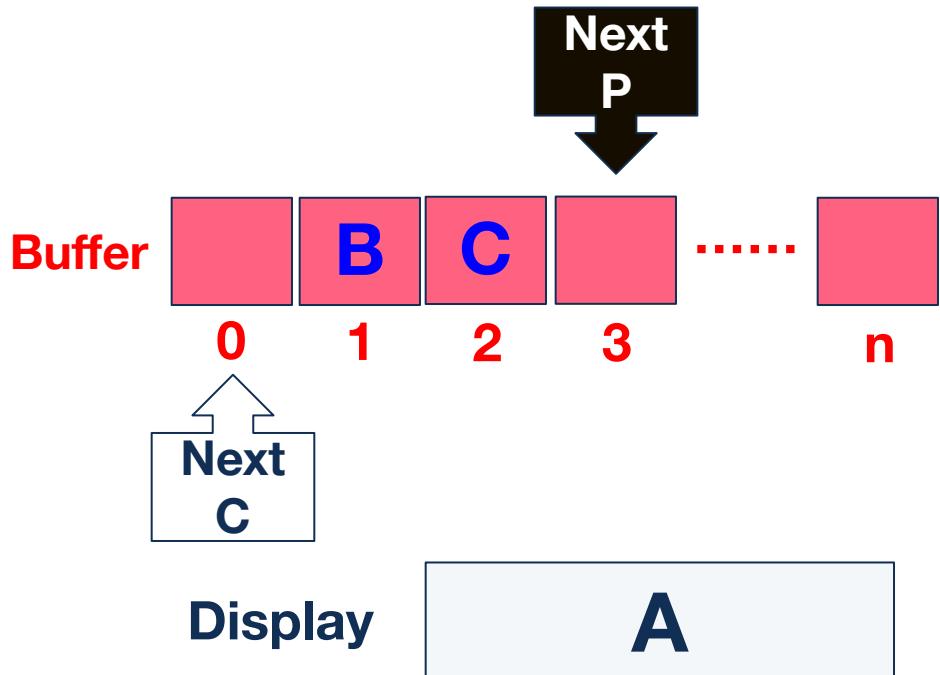
Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

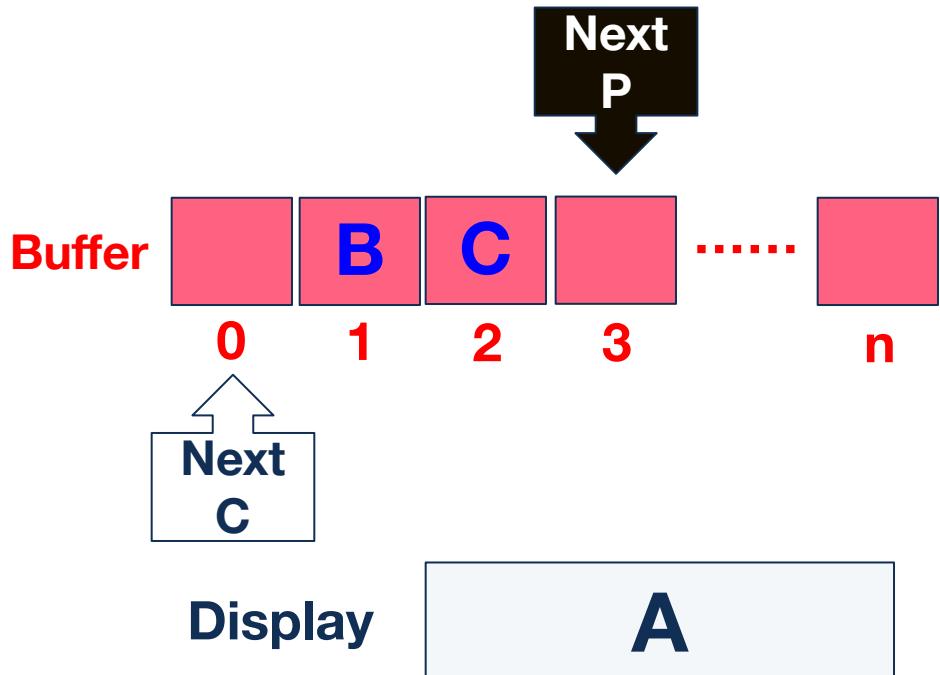
Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

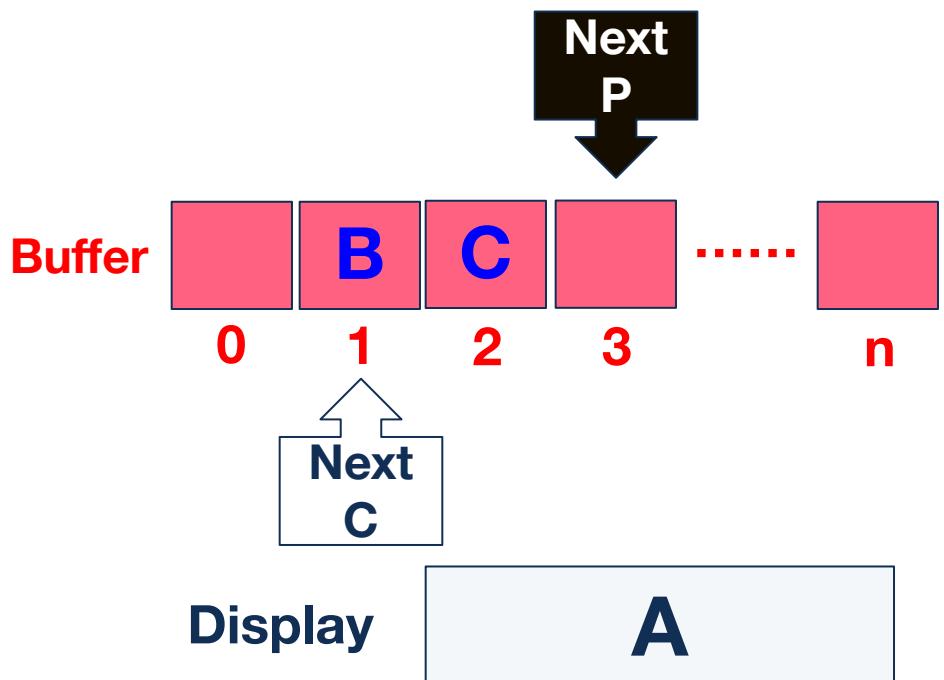
Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    → Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

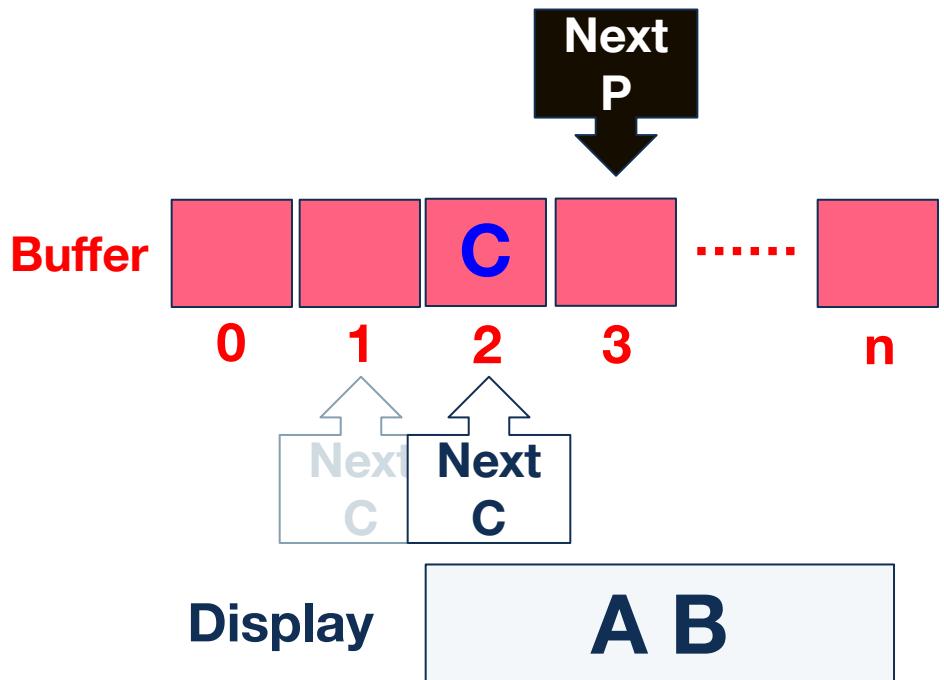
Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full



Producer

```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

mutex

empty

full

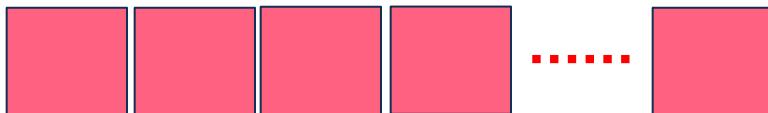
1

n

0

Next
P

Buffer



0 1 2 3 n

Next
C Next
C

Display

A B C

Producer

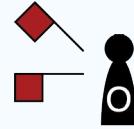
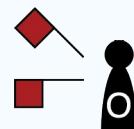
```
do{  
    //produce an item in nextp  
    wait(empty);  
    wait(mutex);  
    //add nextp to buffer  
    signal(Mutex);  
    signal(full);  
}(true)
```

Consumer

```
do{  
    Wait(full);  
    Wait(mutex);  
    //remove item from buffer to nextc  
    Signal(mutex);  
    Signal(empty);  
    //consume the item in nextc  
}(true)
```

Semaphore Implementation With No Busy Waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - Block()** – place the process invoking the operation on the appropriate waiting queue.
 - Wakeup()** – remove one of processes in the waiting queue and place it in the ready queue.



Semaphore implementation

Implementation of Wait()

```
wait (S){  
    value--;  
  
    if (value < 0) {  
        block();}  
  
    -- add this process to  
    waiting queue  
}
```

Implementation of Signal()

```
Signal (S){  
    value++;  
  
    if (value ≤ 0) {  
        wakeup(P); }  
  
    -- remove a process P from  
    the waiting queue  
}
```

Mutex
value

1

Mutex list

Process B

Start
Wait(Mutex);
CriticalSection;
Signal(Mutex);
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection;
Signal(Mutex);
Do remainder

Process A

Start
Wait(Mutex);
CriticalSection;
Signal(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

1

Mutex list

Process B

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

Process C

Start

Wait(Mutex);

Section

Mutex);

Do remainder

Process A

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

0

Mutex list

Process B

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

Process C

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

Process A

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

0

Mutex list

Process B

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

Process C

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

Process A

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

0

Mutex list

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex);
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

-1

Mutex list

Process B

Start
→ Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Process C

Start
→ Signal(Mutex);
CriticalSection
Wait(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

-1

Mutex list

Process B

Blocked

Process B

Start
→ Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Process C

Start
→ Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Process A

Start
→ Wait(Mutex);
CriticalSection
Signal(Mutex);

Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

Mutex
value

-1

Mutex list

Process B

Blocked

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex),
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

-2

Mutex list

Process B

Blocked

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex);
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

Mutex
value

-2

Mutex list

Process B, Process C

Blocked

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Blocked

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex);
Do remainder

```
wait (S){  
    value--;  
    if (value < 0) {  
        //add this to waiting  
        queue  
        block();  
    } }
```

Semaphore implementation with No Busy Waiting

Mutex
value

-1

Mutex list

Process B, Process C

Blocked

Process B

Start

Wait(Mutex);

Critical Sec

Signal(Mut

Do remain

Blocked

Process C

Start

Wait(Mutex);

Critical Section

Signal(Mut

Do remainder

Process A

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

Mutex
value

-1

Mutex list

Process B, Process C

Blocked

Blocked

Process B

Start

Wait(Mutex);

Critical Sec

Signal(Mut

Do remain

Process A

Start

Wait(Mutex);

Critical Section

Signal(Mutex);

Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

0

Mutex list

Process C

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Blocked

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

0

Mutex list

Process C

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Blocked

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Signal(Mutex);
Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

Mutex
value

1

Mutex list

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

ooo

Mutex
value

1

Mutex list

Process B

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

Process C

Start
Wait(Mutex);
CriticalSection
Signal(Mutex);
Do remainder

```
signal (S){  
    value++;  
    if (value <= 0) {  
        //remove a process P  
        //from the waiting queue  
        wakeup(P);  
    }  
}
```

Semaphore implementation with No Busy Waiting

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
wait (S){
    value--;
    if (value < 0) {
        //add this process to
        //waiting queue
        block();
    }
}
```

	S	Q
value	1	1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

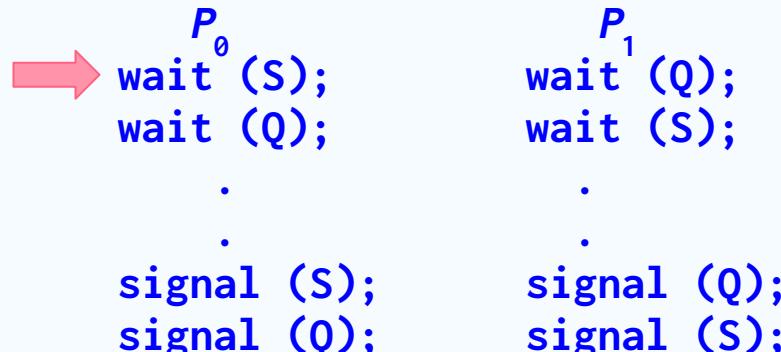
The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
wait (S){
    value--;
    if (value < 0) {
        //add this process to
        //waiting queue
        block();
    }
}
```

	S	Q
value	0	1



Deadlock and Starvation

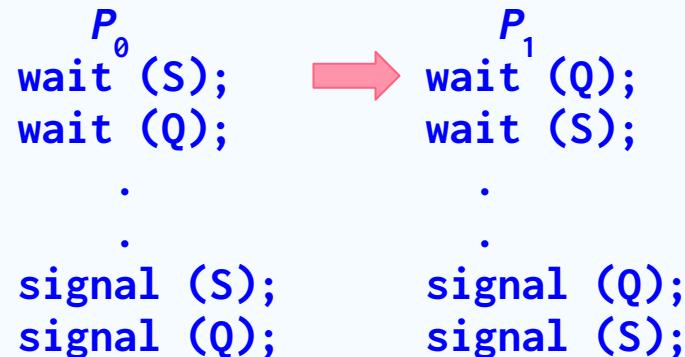
The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1.

```
wait (S){  
    value--;  
  
    if (value < 0) {  
        //add this process to  
        waiting queue  
  
        block();  
    }  
}
```

	S	Q
value	0	0



Deadlock and Starvation

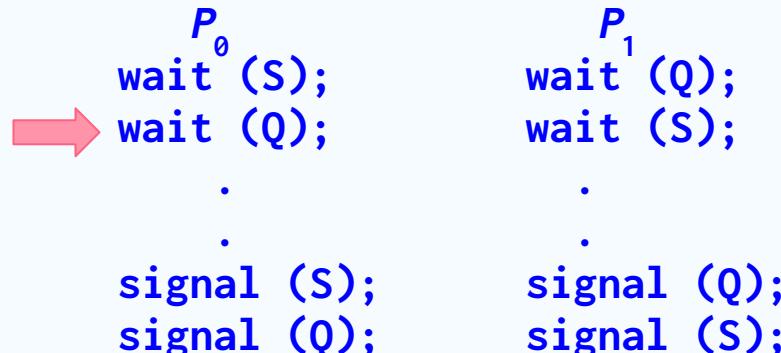
The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
wait (S){
    value--;
    if (value < 0) {
        //add this process to
        //waiting queue
    }
    block();
}
```

	S	Q
value	0	-1



Deadlock and Starvation

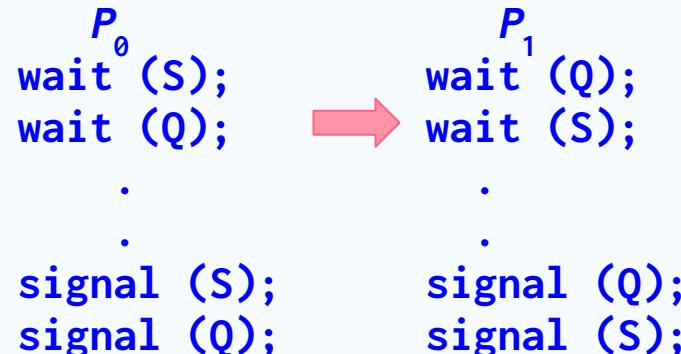
The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1.

```
wait (S){  
    value--;  
  
    if (value < 0) {  
        //add this process to  
        //waiting queue  
  
        block();  
    }  
}
```

	S	Q
value	-1	-1



Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
signal (S){
    value++;
    if (value <= 0) {
        //remove a process P
        from the waiting queue
    }
    wakeup(P);
}
```

	S	Q
value	-1	-1

P_0	wait (S);	P_1
	wait (Q);	wait (S);
.	.	.
.	.	.
	signal (S);	signal (Q);
	signal (Q);	signal (S);



Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

	S	Q
value	1	1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

	S	Q
value	0	1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0 wait (S); wait (Q); . signal (S); signal (Q);	P_1 wait (Q); wait (S); . signal (Q); signal (S);
--	--

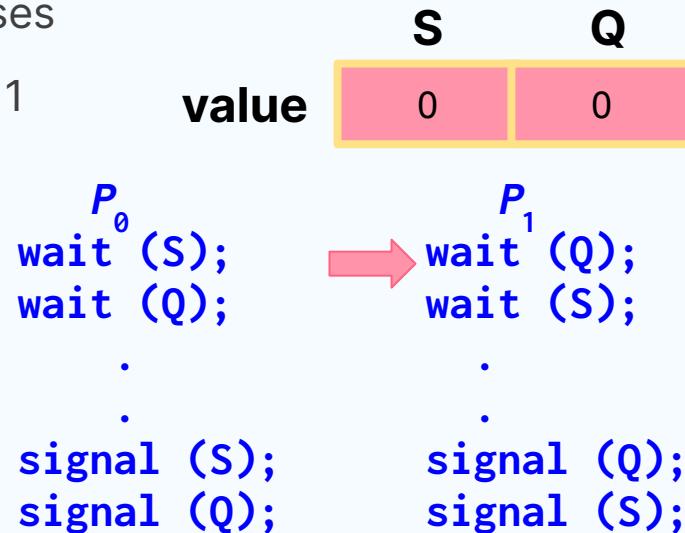
Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```



Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

	S	Q
value	0	0

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```

P_0 wait (S); wait (Q); . . signal (S); signal (Q);	P_1 wait (Q); wait (S); . . signal (Q); signal (S);
---	---

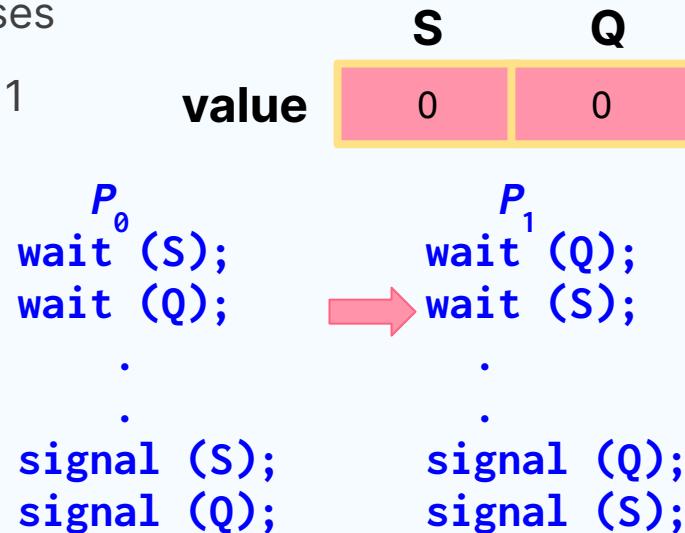
Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

```
wait (S) {
    while S ≤ 0 ; // no-op
    S--;
}
```



Deadlock and Starvation

The implementation of a semaphore with a **waiting queue** may result in a situation where two or more processes are waiting for each other.

Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let **S** and **Q** be two semaphores initialized to 1

	S	Q
value	0	0

```
signal (S) {
    S++;
}
```

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);

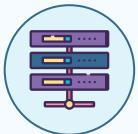
.	.
.	.

signal (S);	signal (Q);
signal (Q);	signal (S);



Explanation

Suppose that P0 executes `wait(S)` and then P1 executes `wait(Q)`.



When P0 executes `wait(Q)`, it must wait until P1 executes `signal(Q)`.

Similarly, when P1 executes `wait(S)`, it must wait until P0 executes `signal(S)`.



Since these `signal()` operations cannot be executed, P0 and P1 are deadlock.

Problems with semaphore

Incorrect use of semaphore operations:

01 signal (mutex) wait (mutex)

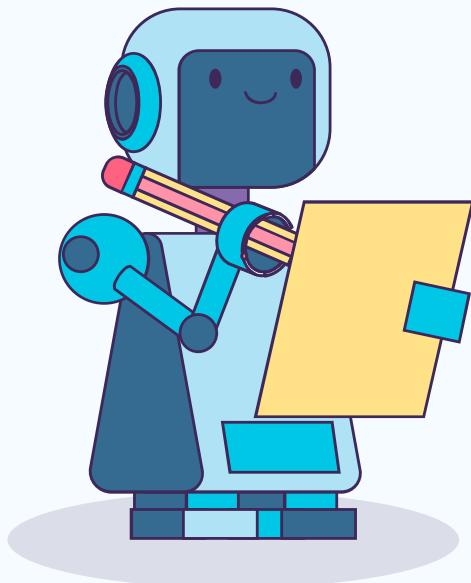
Several processes may execute in the CS

02 wait (mutex) ... wait (mutex)

Deadlock will occur

03 Omitting of wait (mutex) or/and signal (mutex)

Several processes may execute in the CS



Thank you

CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)



BMCS3003

Distributed Systems

and Parallel Processing

L05 - Concurrency Control (Part 2)

Presented by

Assoc Prof Ts Dr Tew Yiqi
May 2023



Table of contents

01

Monitors

02

Token-passing
Mutual Exclusion

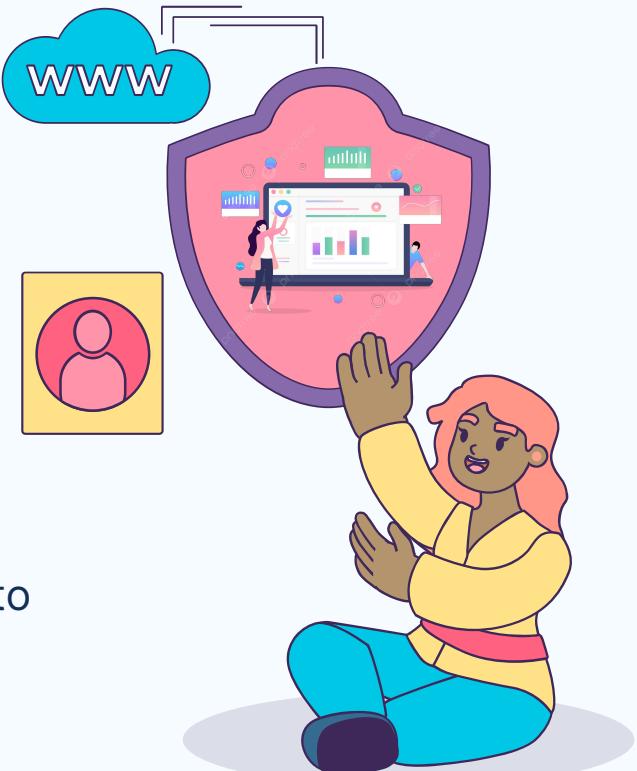
03

Deadlocks

01

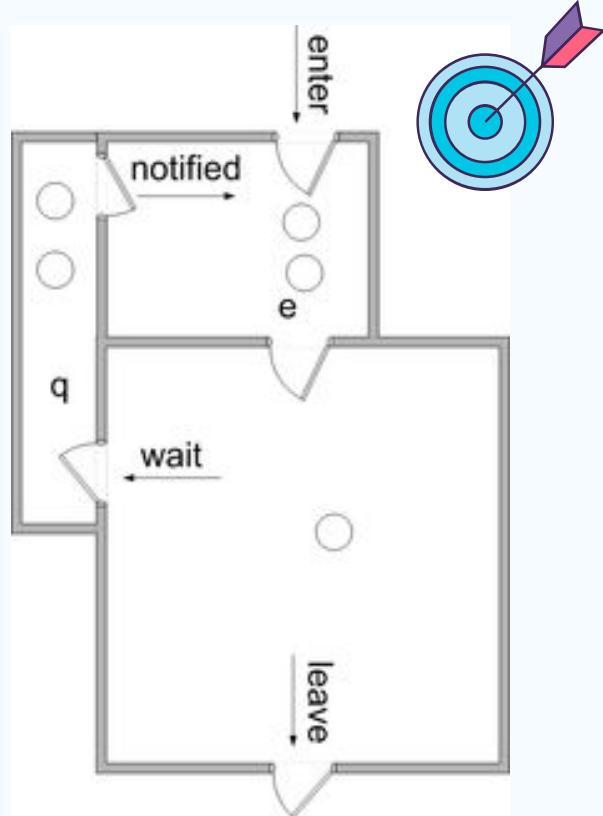
Monitors

An abstract data type that enables only a process to use a shared resource at a time.



Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control.
- Implemented in a number of programming languages.
 - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data



Monitor Characteristics



Local data variables are accessible only by the monitor's procedures and not by any external procedure

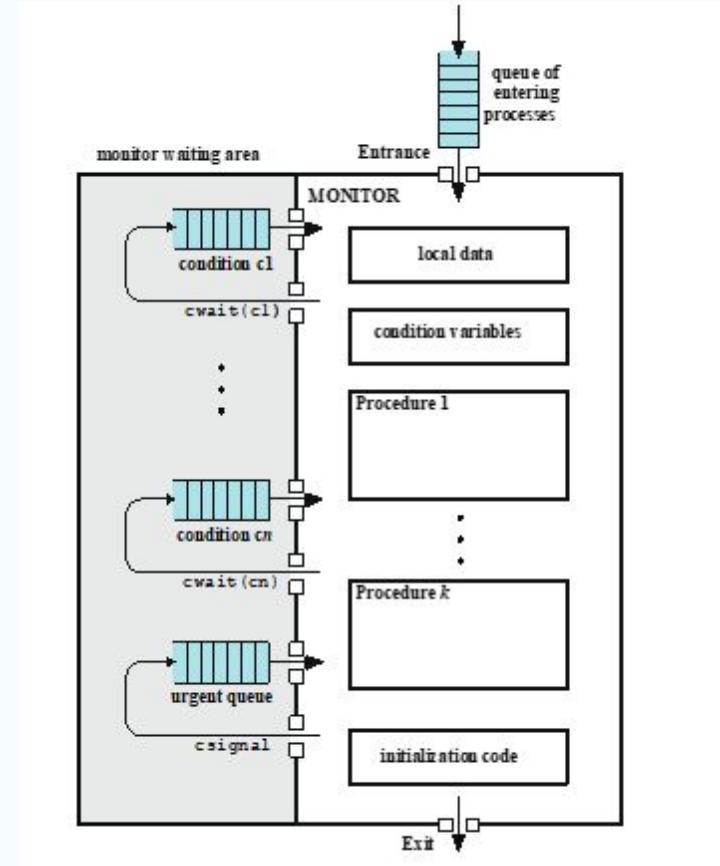


Process enters monitor by invoking one of its procedures



Only one process may be executing in the monitor at a time

Example of Monitor



Synchronisation

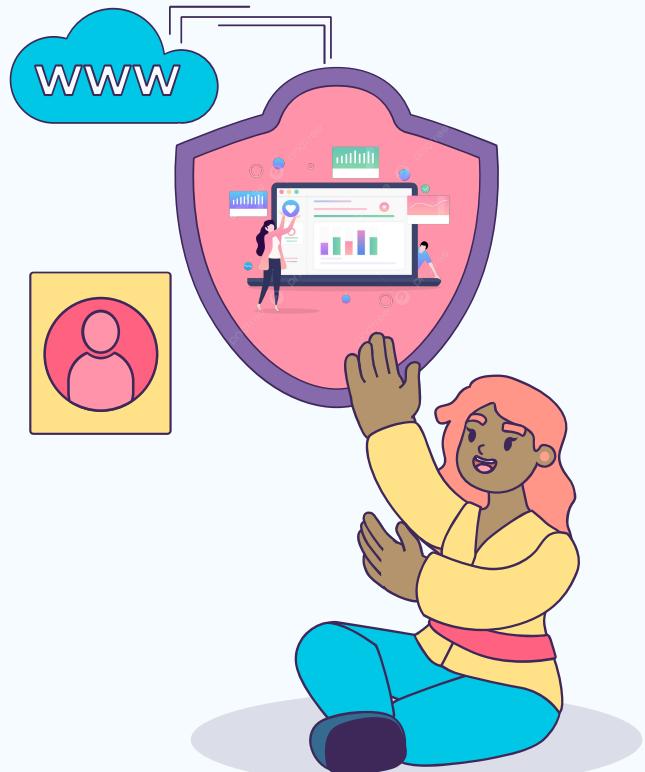
- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
 - Condition variables are operated on by two functions:
 - cwait(c): suspend execution of the calling process on condition c
 - csignal(c): resume execution of some process blocked after a cwait on the same condition



02

Token Passing

An algorithm for Mutual Exclusion



Token Passing

Algorithm for Mutual Exclusion

Suzuki-Kasami algorithm [\[Paper\]](#)

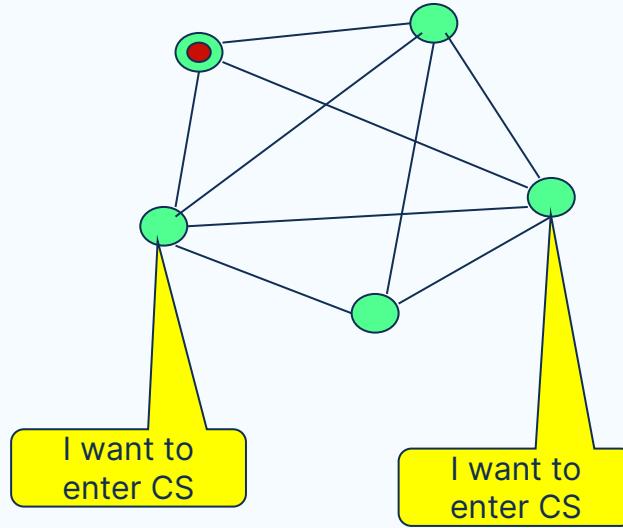


Ichiro Suzuki



Tadao Kasami

Resources: [Geeksforgeeks](#)



- Completely connected network of processes.
- There is **one token** in the network. The holder of the token has the permission to enter Critical Section (**CS**).
- Any other process trying to enter CS must acquire that token. Thus the token will move from one process to another based on demand.

Suzuki-Kasami algorithm [Paper]

Process i broadcasts (i, num)

Sequence number
of the request

Each process maintains

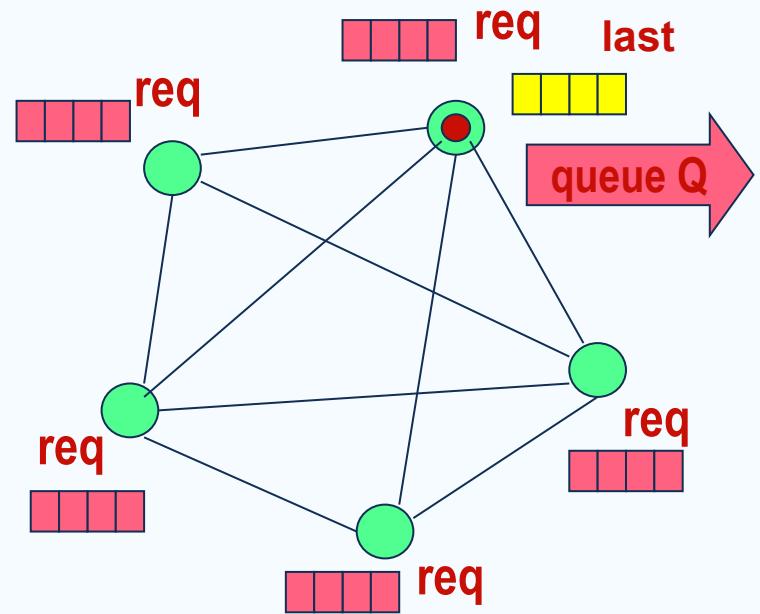
- an array **req**: $\text{req}[j]$ denotes the sequence number of the request from process j

(Some requests will be stale soon)

Additionally, the holder of the token maintains

- an array **last**: $\text{last}[j]$ denotes the sequence number of *the latest visit* to CS from for process j .

- **a queue Q** of waiting processes



Suzuki-Kasami algorithm [Paper]

When a process **i** receives a request (k, num) from process **k**, it sets $\text{req}[k]$ to $\max(\text{req}[k], \text{num})$.

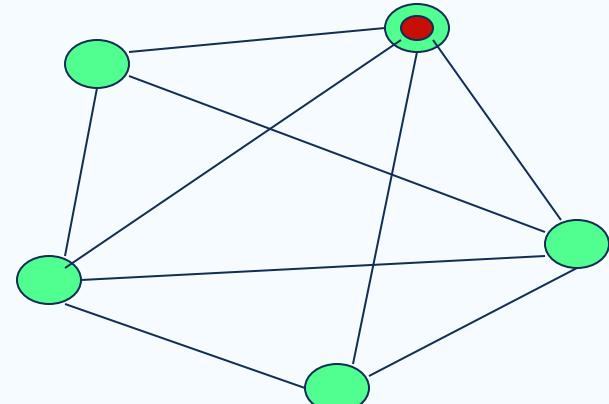
The holder of the token

- Completes its CS
- Sets $\text{last}[i] :=$ its own **num**
- Updates **Q** by retaining each process **k** only if
 $1 + \text{last}[k] = \text{req}[k]$

(This guarantees the freshness of the request)

- Sends the token to the **head of Q**, along with the array **last** and the **tail of Q**

In fact, **token** $\equiv (Q, \text{last})$



Req: array[0..n-1] of integer
Last: array [0..n-1] of integer

Suzuki-Kasami algorithm [Paper]

{Program of process j}

Initially,

$\forall i: \text{req}[i] = \text{last}[i] = 0$

* Entry protocol *

$\text{req}[j] := \text{req}[j] + 1$

Send (j, $\text{req}[j]$) to all

Wait until token (Q, last) arrives

Critical Section

* Exit protocol *

$\text{last}[j] := \text{req}[j]$

$\forall k \neq j: k \notin Q \wedge \text{req}[k] = \text{last}[k] + 1 \rightarrow$ append k to Q;

if Q is not empty \rightarrow send (tail-of-Q, last) to head-of-Q **fi**

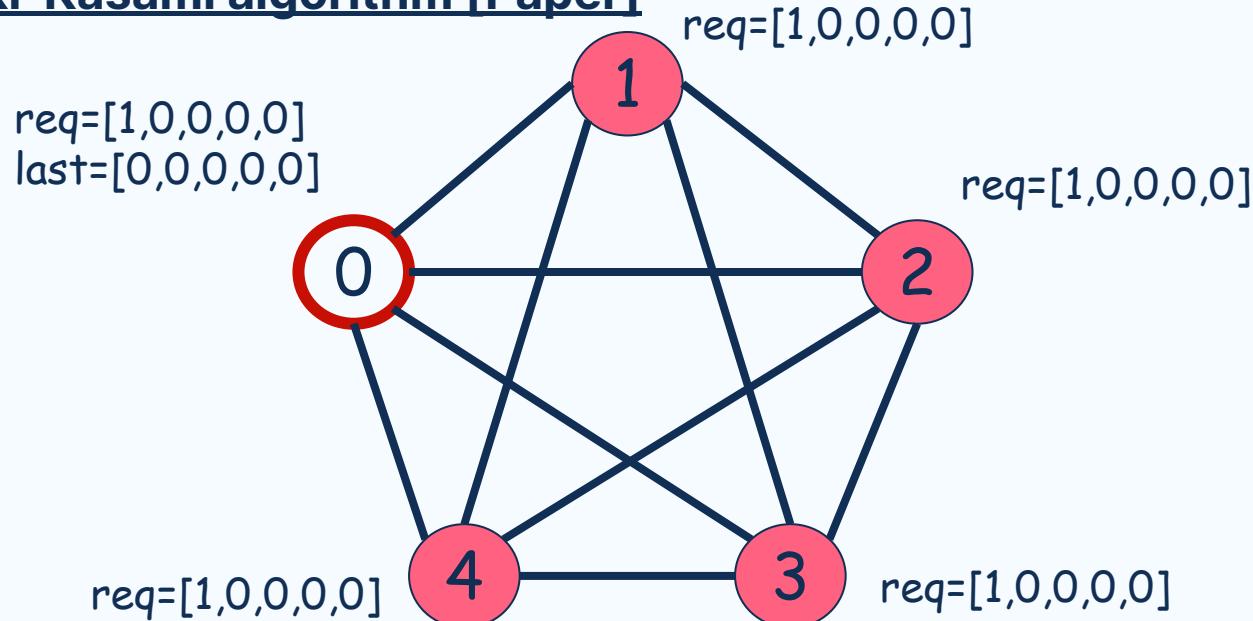
* Upon receiving a request (k, num)

*

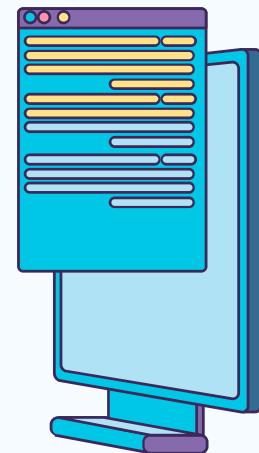
$\text{req}[k] := \max(\text{req}[k], \text{num})$

ooo

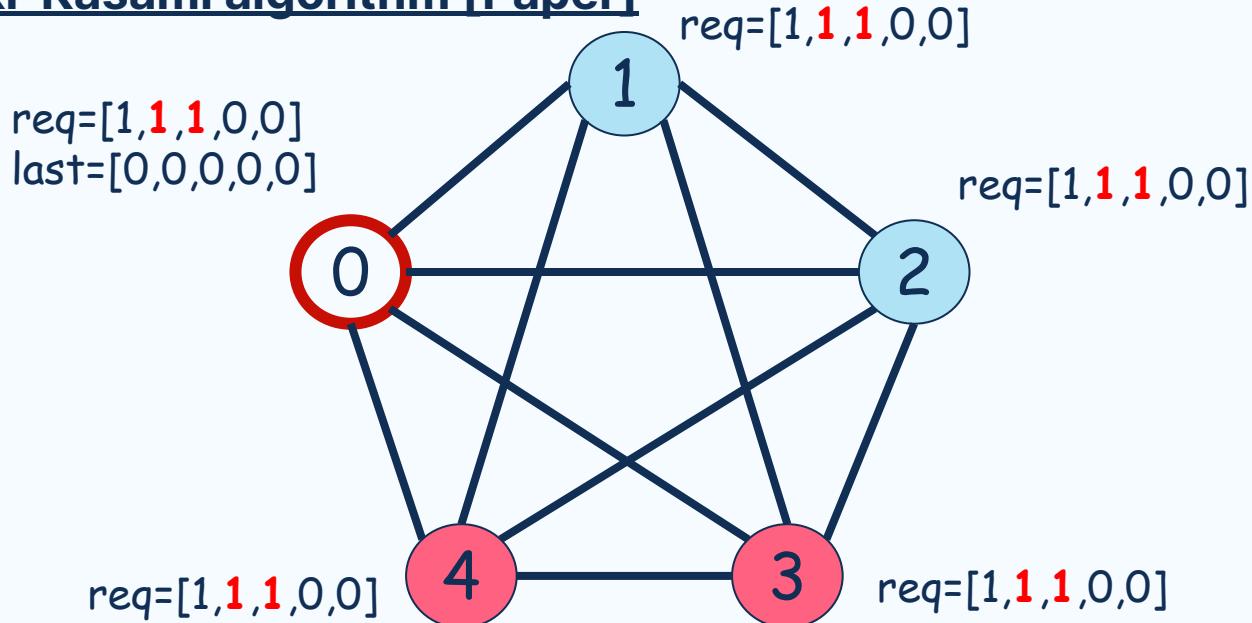
Suzuki-Kasami algorithm [Paper]



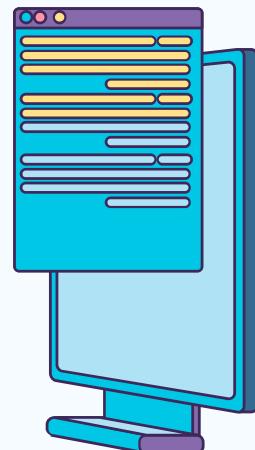
In the initial state:
process 0 has sent a request to all,
and grabbed the token.



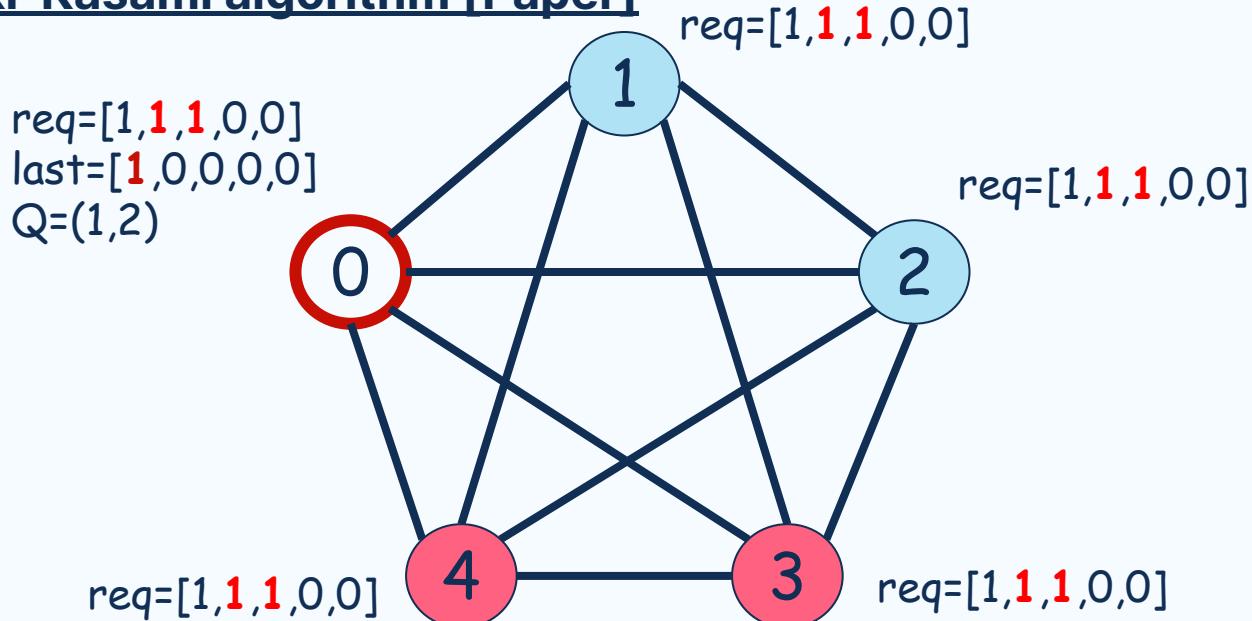
Suzuki-Kasami algorithm [Paper]



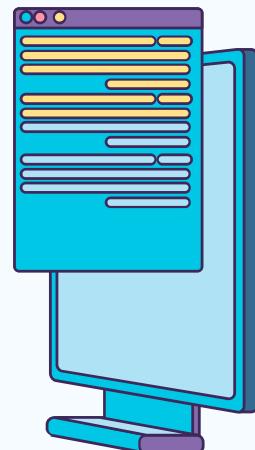
1 & 2 send the request to enter Critical Section



Suzuki-Kasami algorithm [Paper]

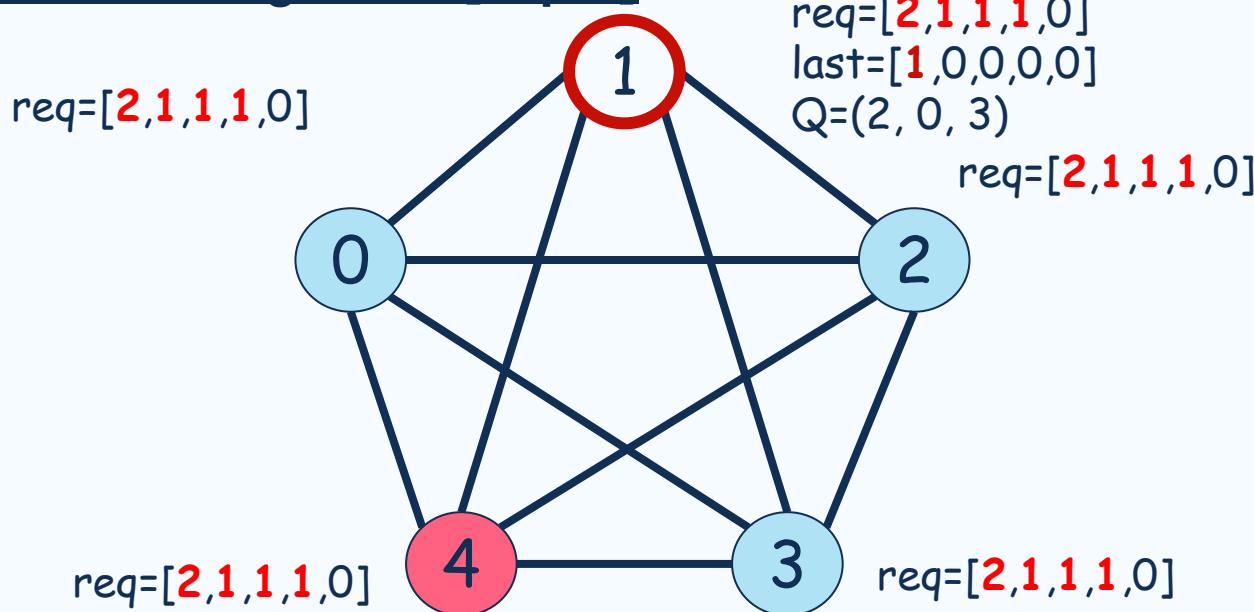


0 prepares to quit Critical Section



ooo

Suzuki-Kasami algorithm [Paper]



req=[2,1,1,1,0]

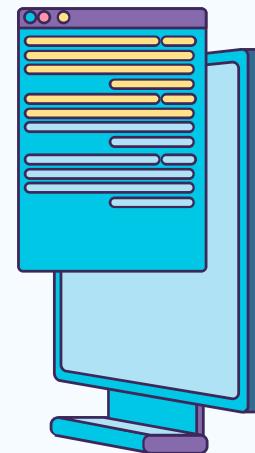
last=[1,0,0,0,0]

Q=(2, 0, 3)

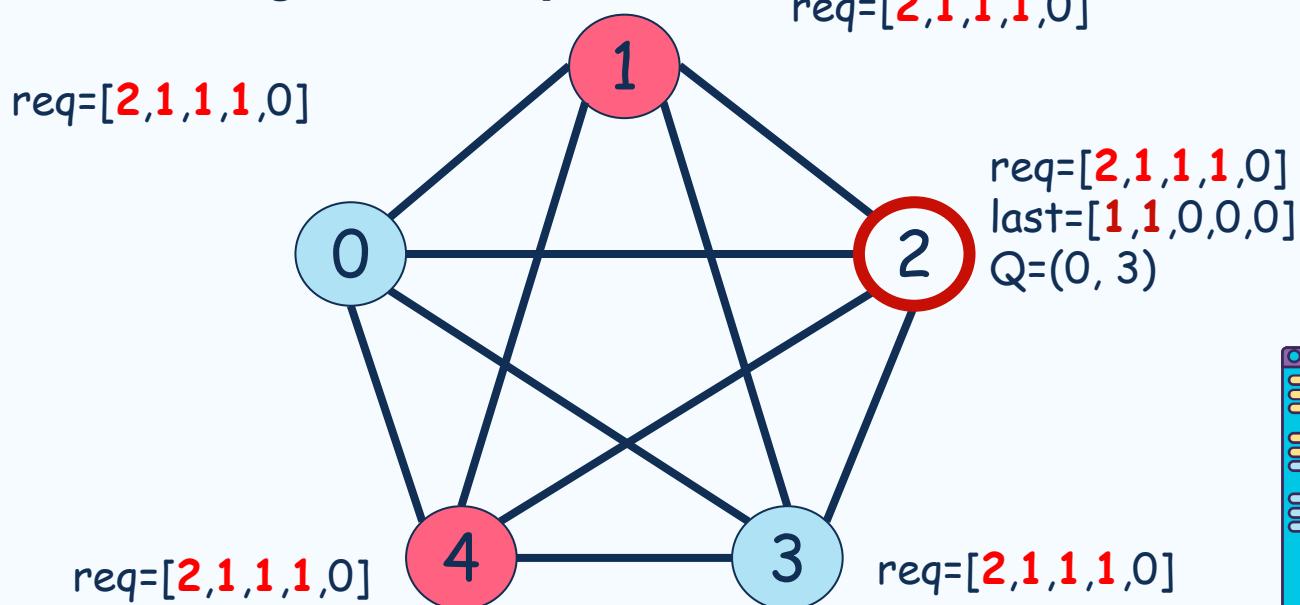
req=[2,1,1,1,0]

1 enters CS

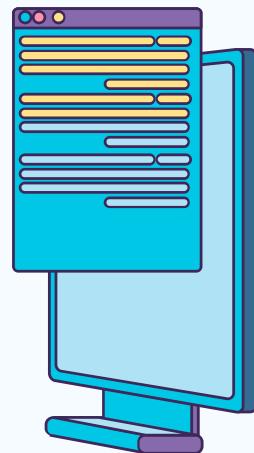
0 and 3 send requests



Suzuki-Kasami algorithm [Paper]

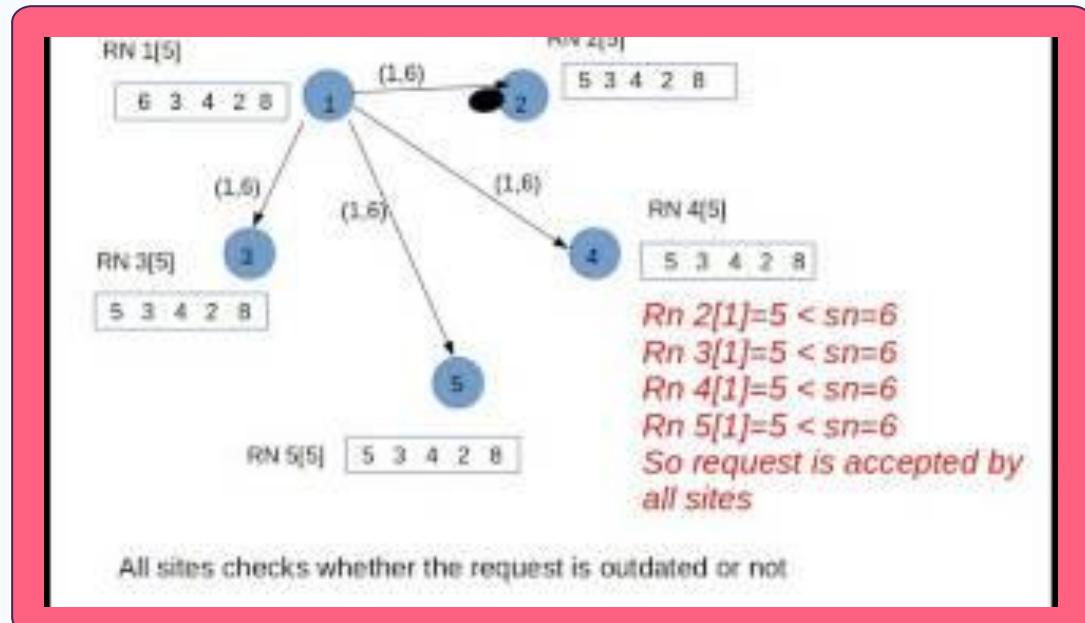


1 prepares to quit Critical Section, sends token to 2
2 enter to CS



Suzuki-Kasami Algorithm

by
Shivani Srivarshini



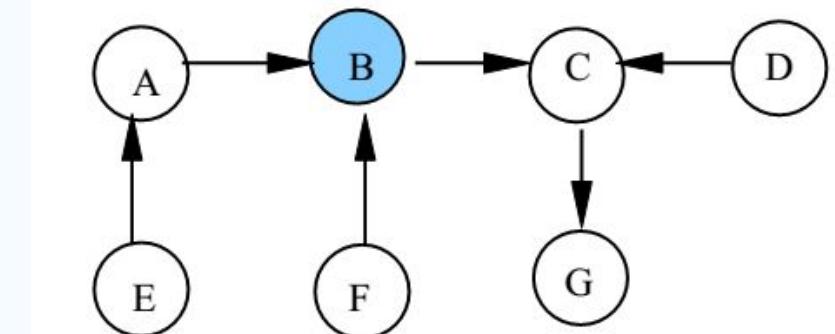
Token Passing Algorithm for Mutual Exclusion

Raymond's Tree algorithm [[Paper](#)]



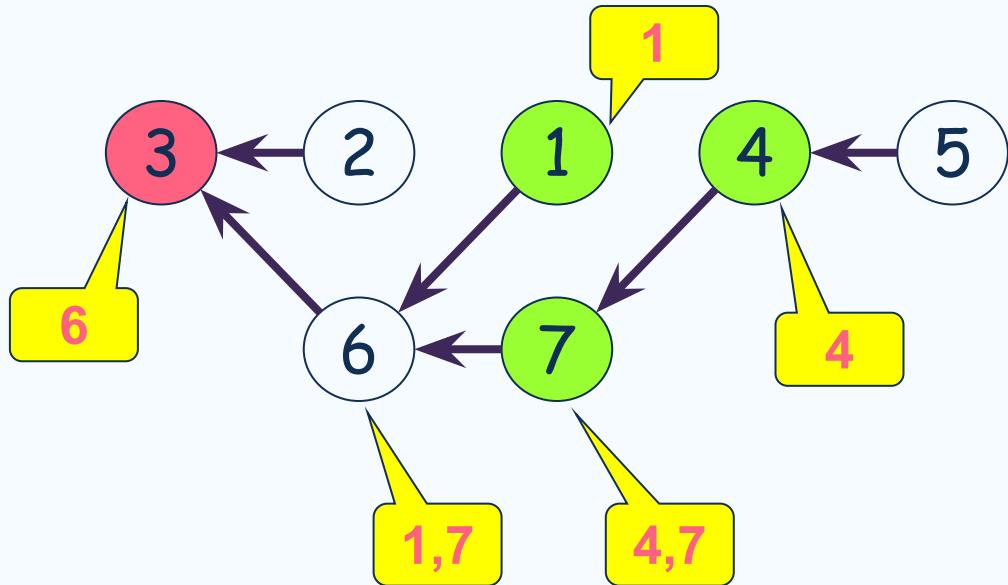
Kerry Raymond

Resources: [Geeksforgeeks](#)

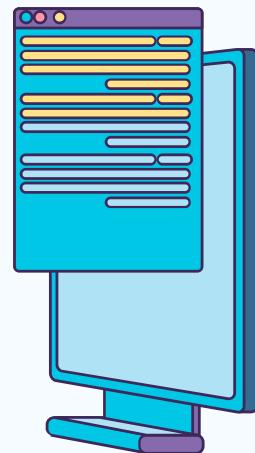


- **Raymond's tree based algorithm** is lock based algorithm for mutual exclusion in a distributed system in which a site is allowed to enter the critical section if it has the token.
- All sites are arranged as a directed tree such that the edges of the tree are assigned direction towards the site that holds the token.
- Site which holds the token is also called root of the tree.

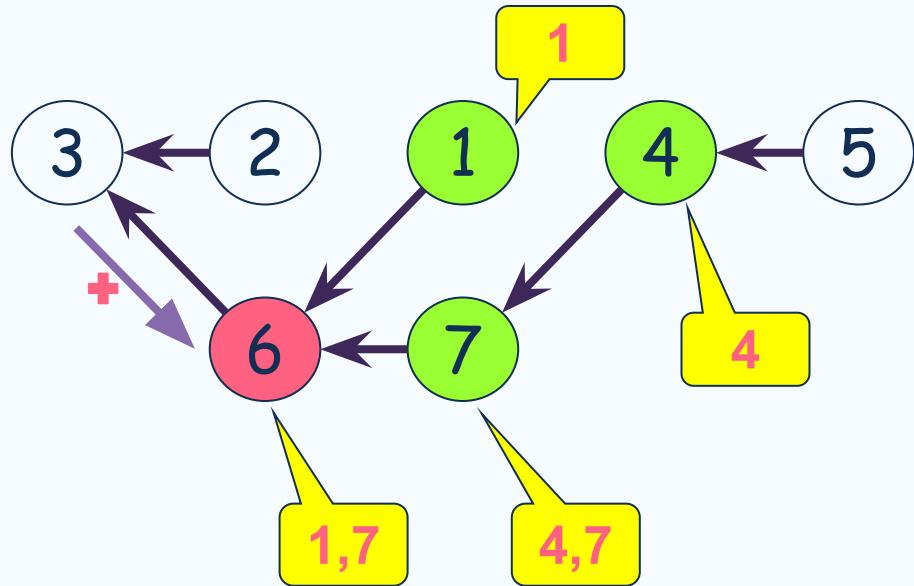
Raymond's Tree algorithm [Paper]



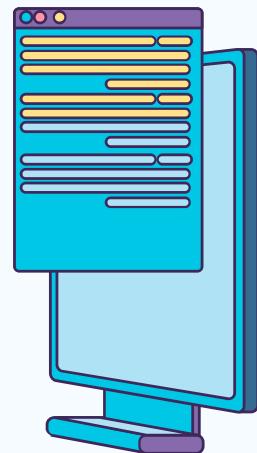
1, 4, 7 want to enter their CS



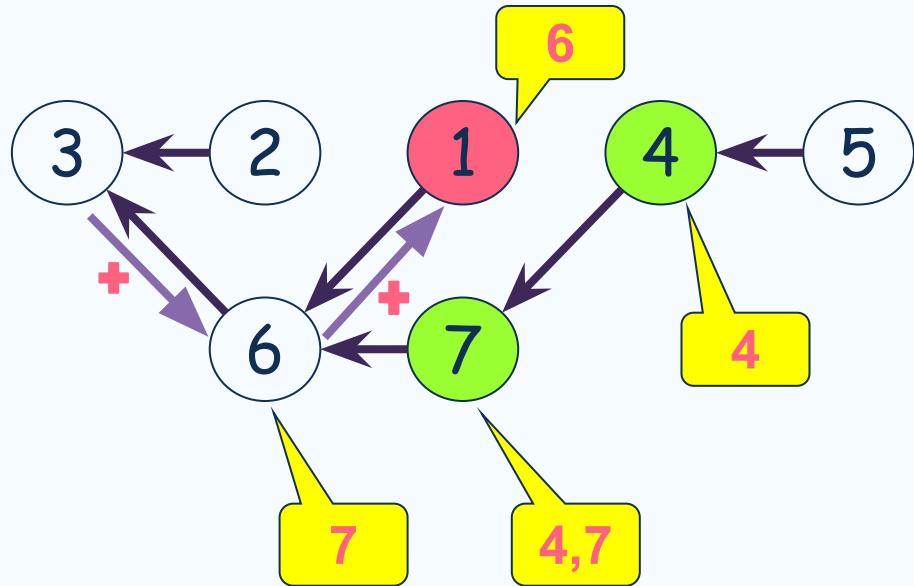
Raymond's Tree algorithm [Paper]



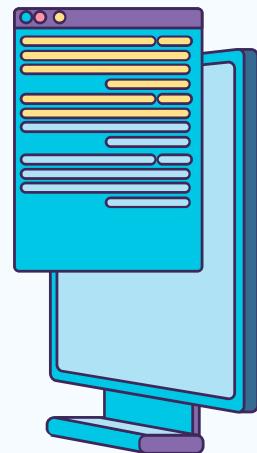
3 sends the token to 6



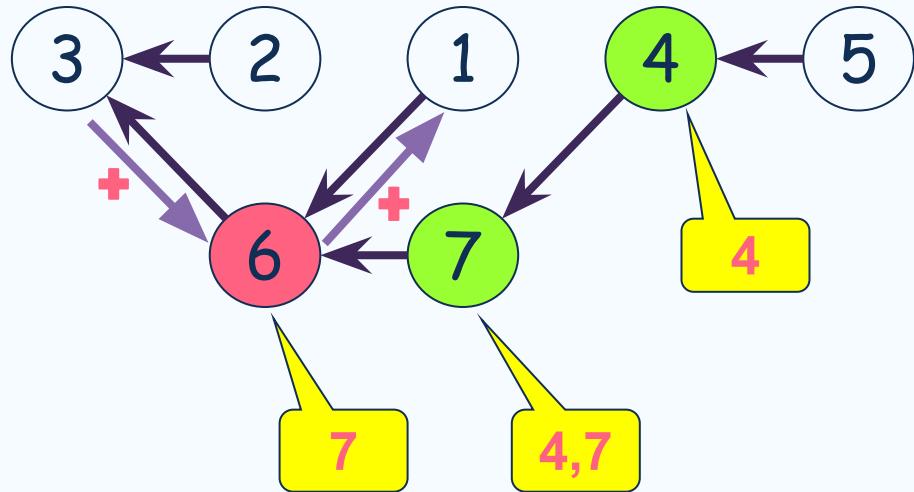
Raymond's Tree algorithm [Paper]



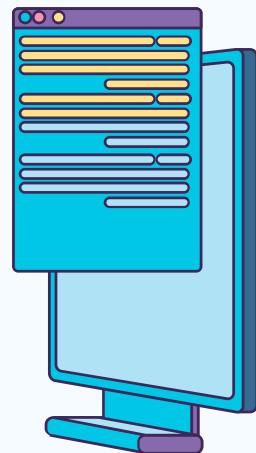
6 forwards the token to 1



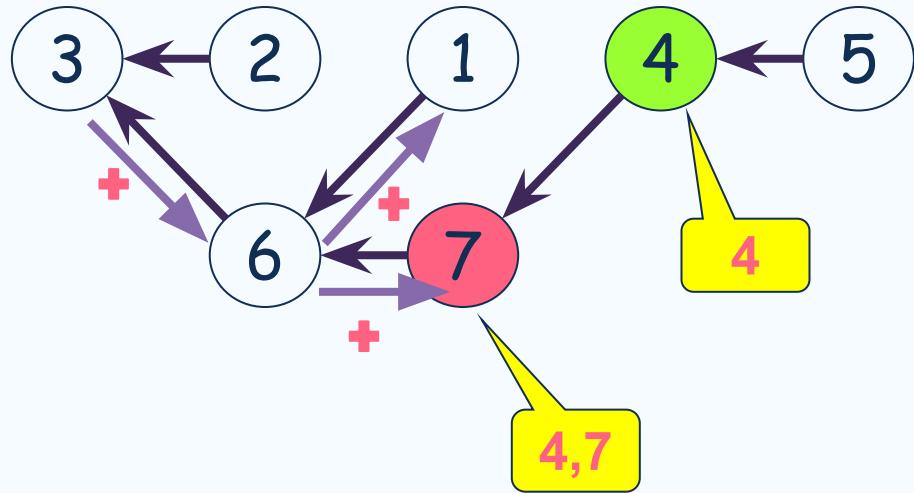
Raymond's Tree algorithm [Paper]



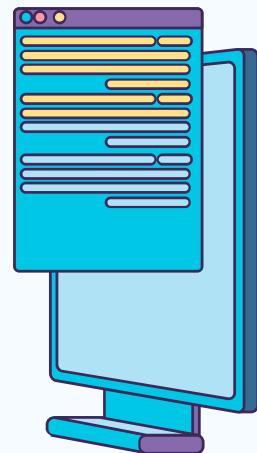
6 requests the token from 1



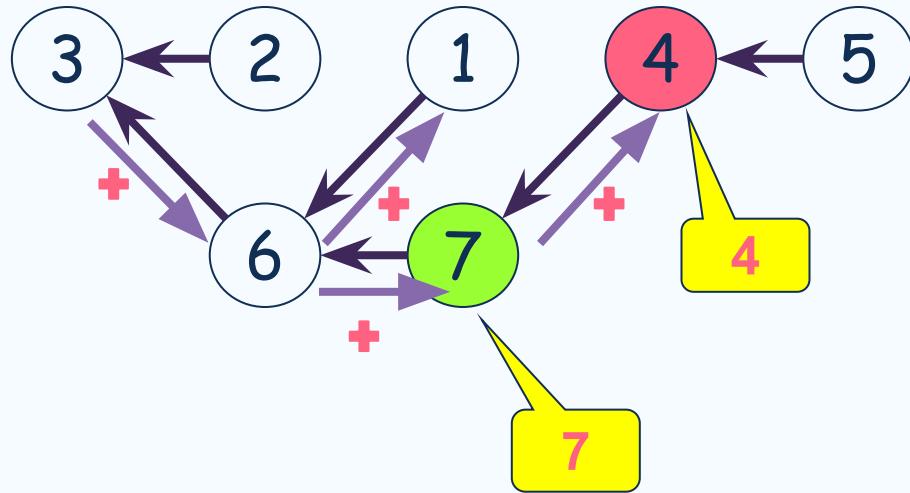
Raymond's Tree algorithm [Paper]



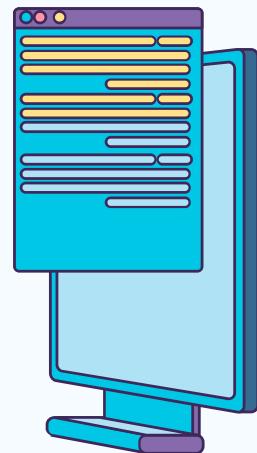
7 requests the token from 6



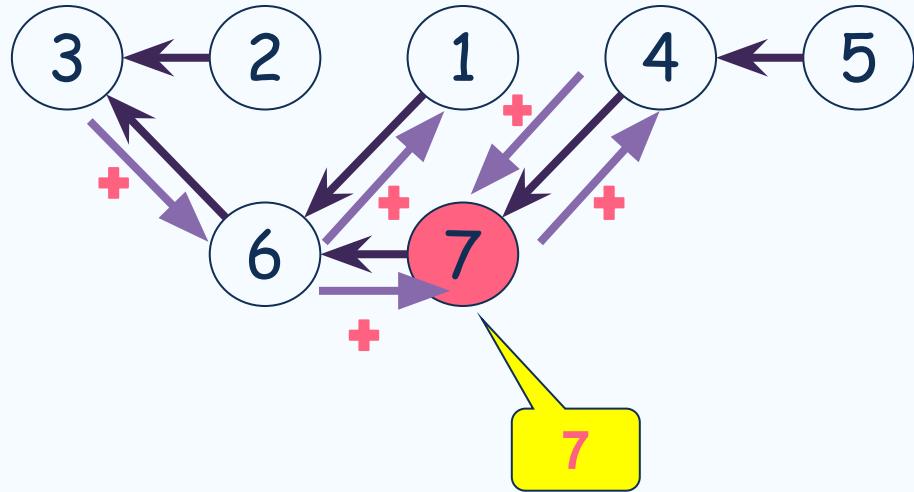
Raymond's Tree algorithm [Paper]



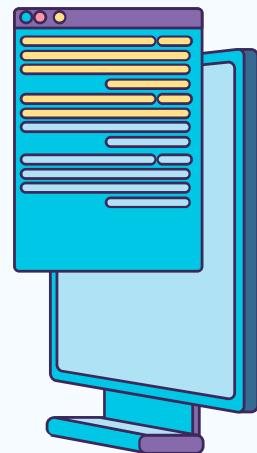
4 requests the token from 7



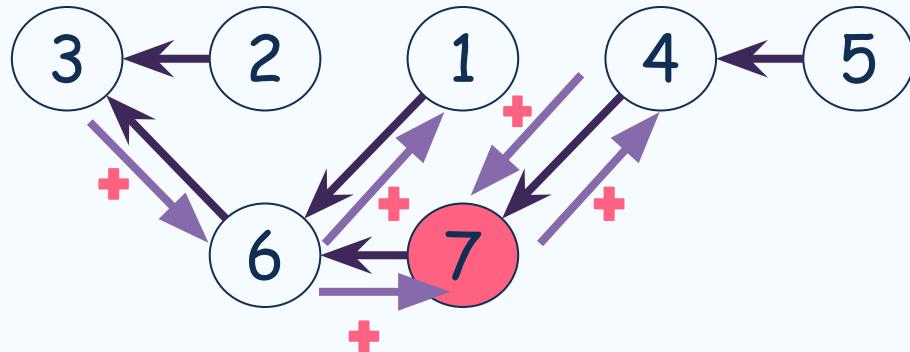
Raymond's Tree algorithm [Paper]



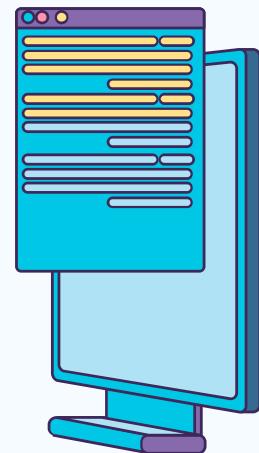
7 requests the token from 4



Raymond's Tree algorithm [Paper]



Done



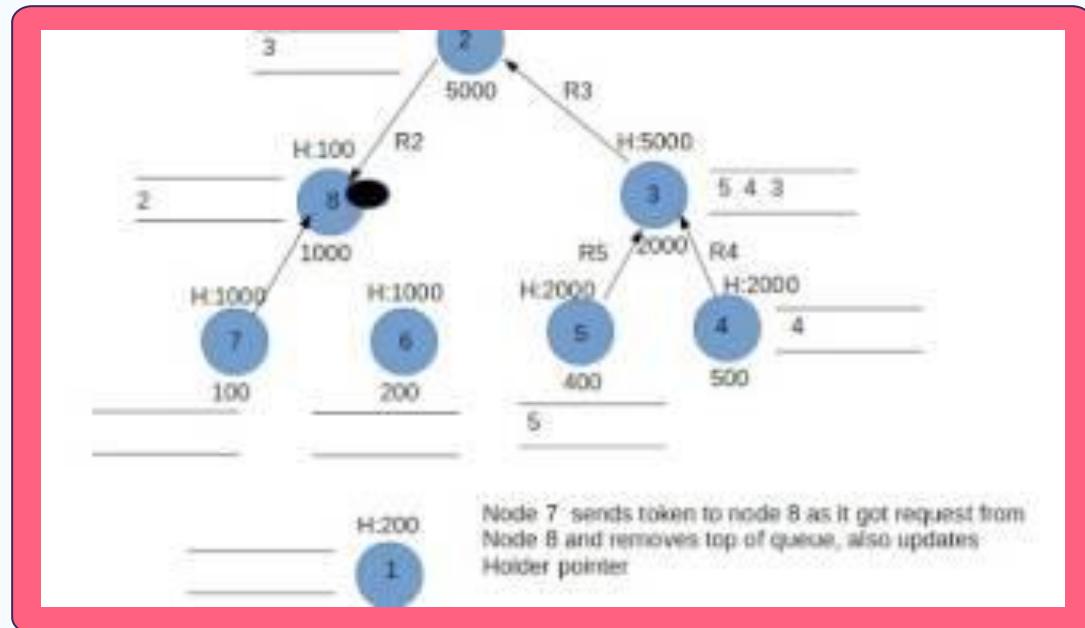
Raymond's Tree algorithm [Paper]



- The message complexity is $O(\text{diameter})$ of the tree.
- Extensive empirical measurements show that the average diameter of **randomly chosen** trees of **size n** is $O(\log n)$.
- Therefore, the authors claim that the average message complexity is $O(\log n)$

Raymond's Tree Algorithm

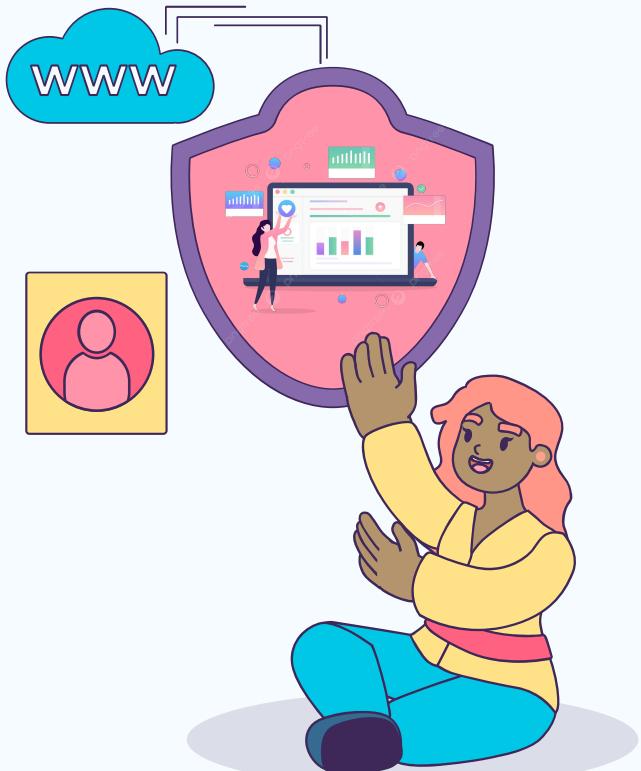
by
Shivani Srivarshini



03

Deadlock

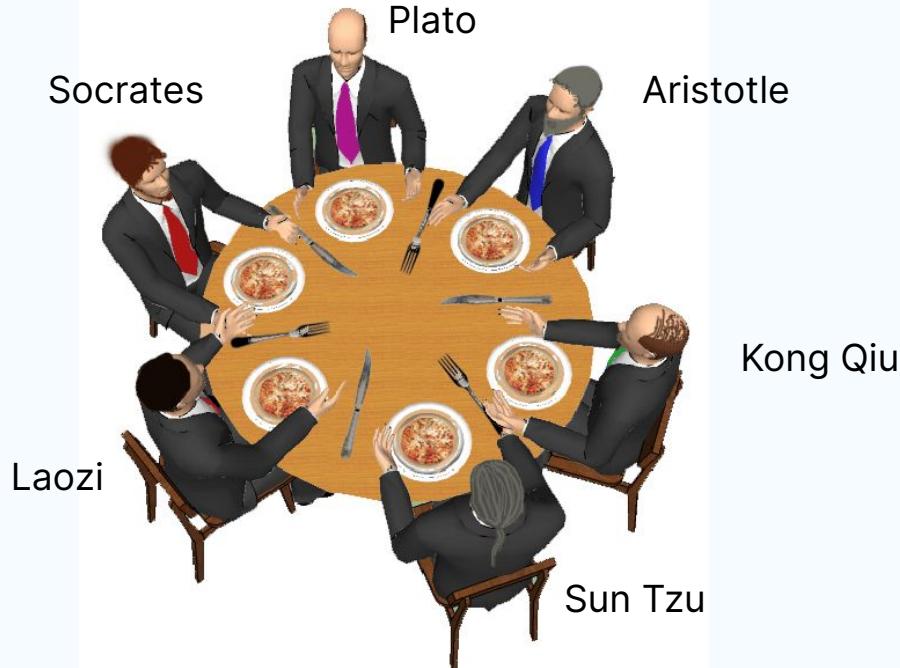
An algorithm for Mutual Exclusion



Dining - Philosophers Problem

Philosophers need both forks to eat and only one philosopher can use a fork at a time.

How to solve?



Dining-Philosophers Problem

We need a

Waiter !

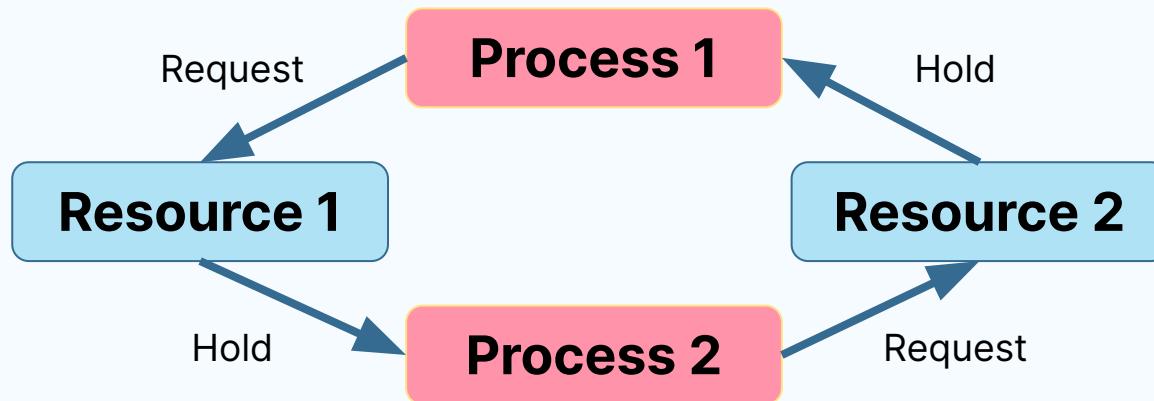
who represents a lock, that the philosophers need exclusive access to before picking up either of their fork.



Resource: [Wikipedia](#)

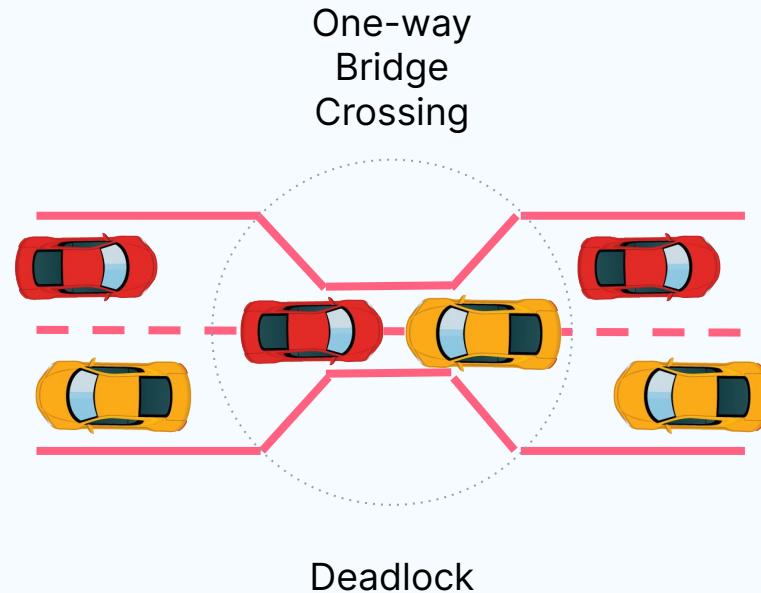
Deadlock Problems and Conditions

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.



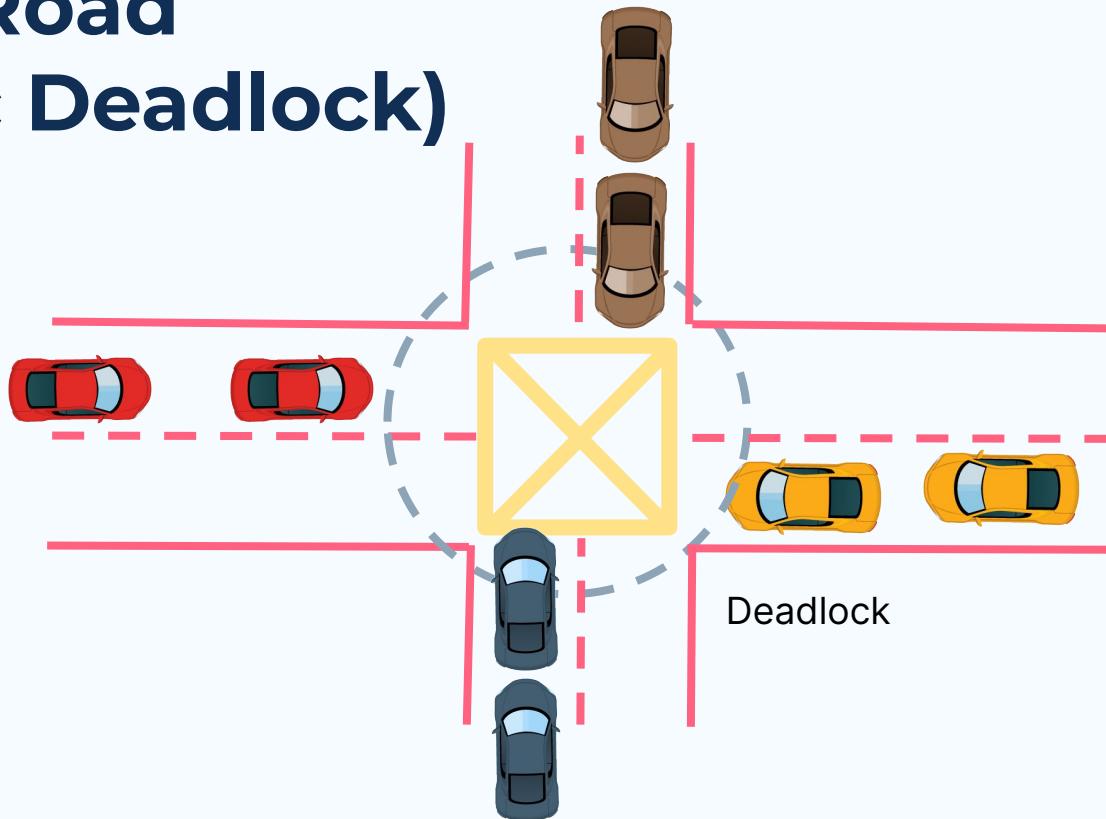
Bridge Crossing

- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

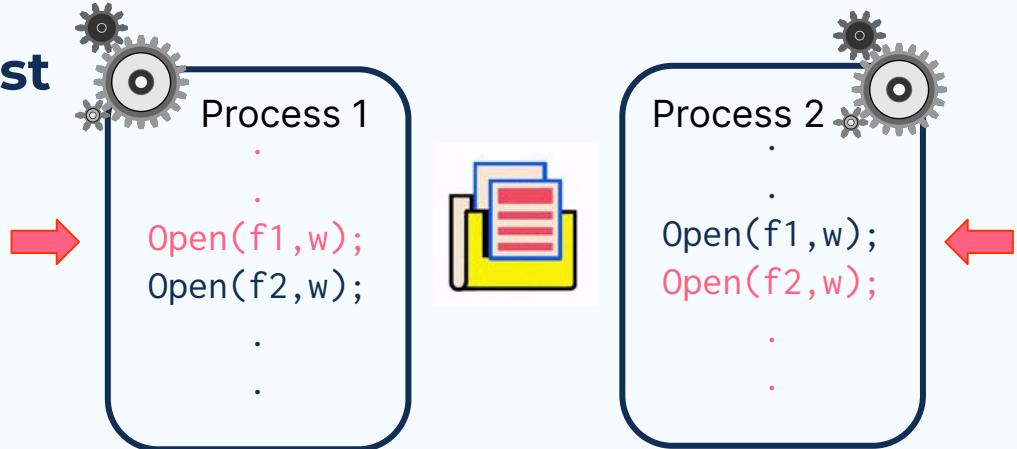




Cross Road (Traffic Deadlock)



Case 1: Deadlocks on File Request

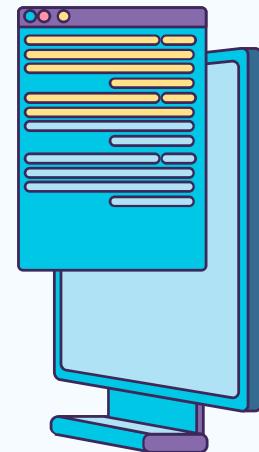
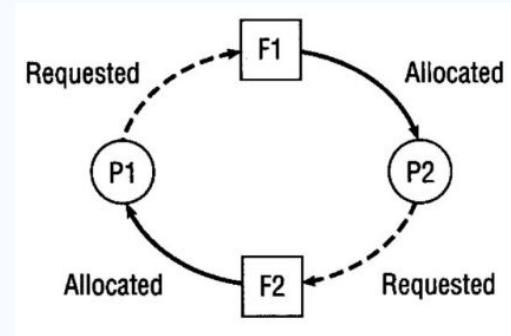


Each holding one of the files open while trying to open other. Since this will never be close, the processes will wait.

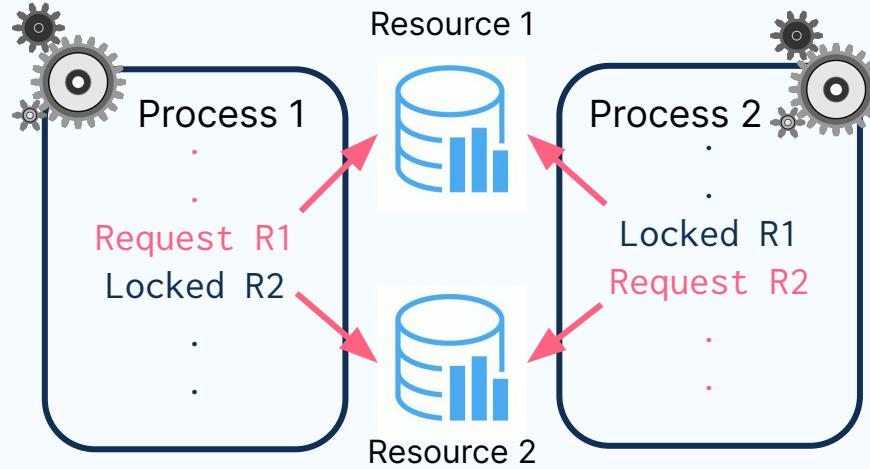
Case 1: Deadlocks on File Request

Occurs if jobs are allowed to request and hold files for the duration of their execution:

- P1 has access to F1 but requires F2 also
- P2 has access to F2 but requires F1 also
- Deadlock remains until a program is withdrawn or forcibly removed and its file is released.
- Any other programs that require F1 and F2 are put on hold as long as this situation continues



Case 2 : Deadlocks in Databases



Occurs if two processes access and lock records in a database:

- P1 accesses R1 and locks it.
- P2 accesses R2 and locks it.
- P1 requests R2, which is locked by P2.
- P2 requests R1, which is locked by P1.

Case 2 : Deadlocks in Databases

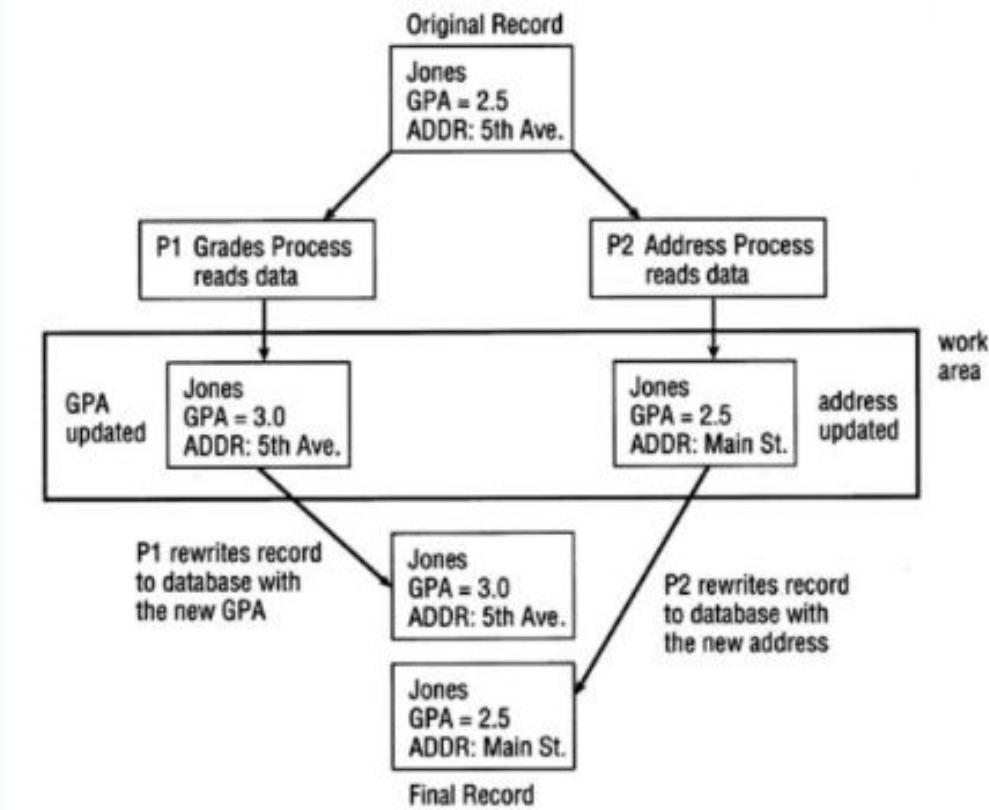
Locking:

A technique through which the user locks out all other user while working with the database.

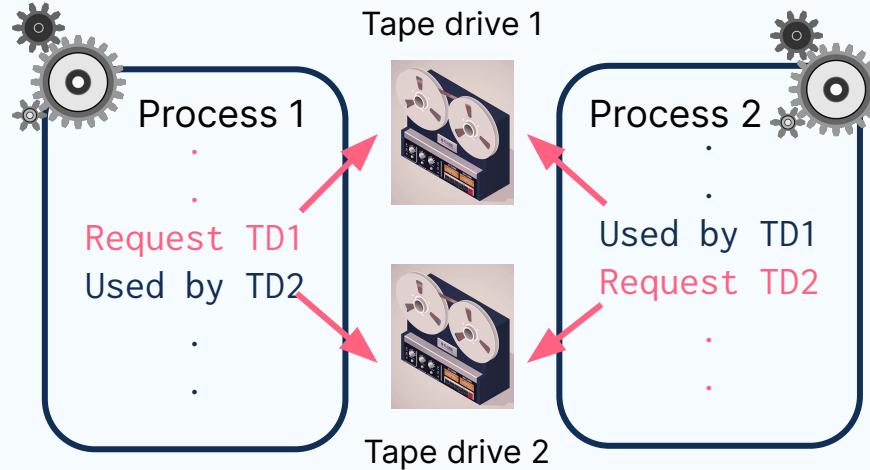
Race between processes:

A condition resulting when locking is not used.

- Causes incorrect final version of data
- Depends on order in which each process executes it.



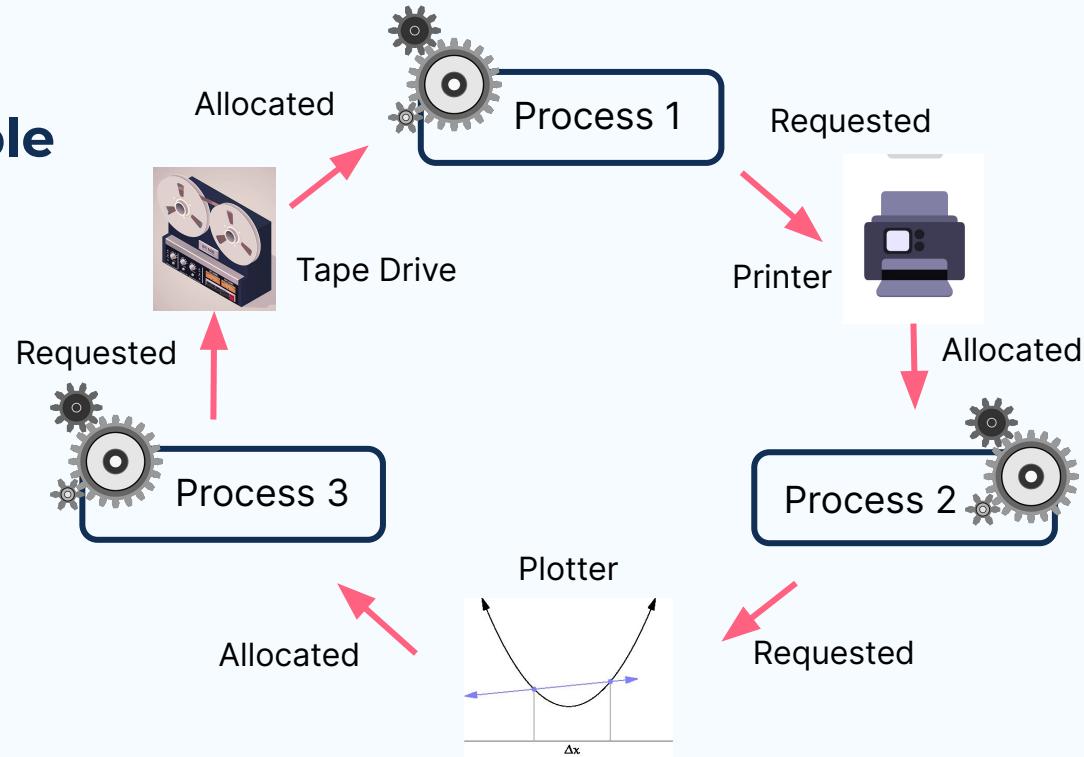
Case 3 : Deadlocks in Dedicated Device Allocation



Occurs when there is a limited number of dedicated devices:

- P1 requests tape drive 1 and gets it.
- P2 requests tape drive 2 and gets it.
- P1 requests tape drive 2 but is blocked.
- P2 requests tape drive 1 but is blocked.

Case 4 : Deadlocks in Multiple Device Allocation



Case 5 : Deadlocks in Spooling



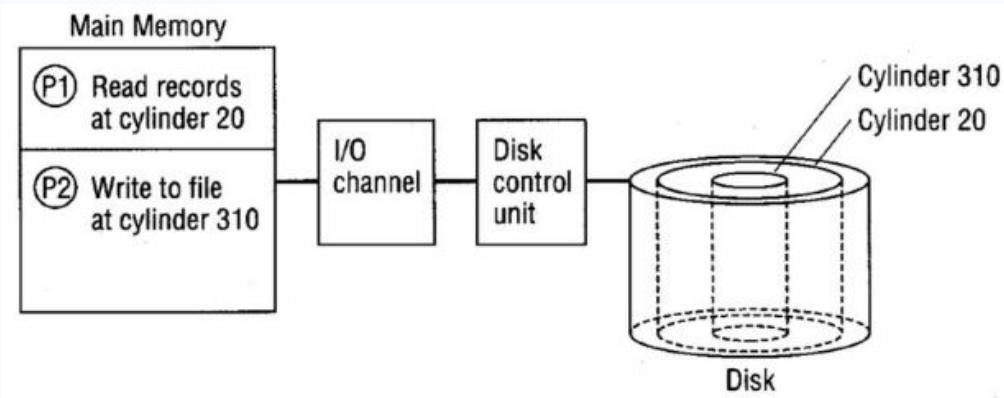
- **Virtual Devices:** Most systems have transformed dedicated devices such as a printer into a sharable device by installing a high-speed device, a disk, between it and the CPU.
- **Spooling:** Disk accepts output from several users and acts as a temporary storage area for all output until printer is ready to accept it.
- **Deadlock in spooling:** If printer needs all of a job's output before it will begin printing, but spooling system fills available disk space with only partially completed output, then a deadlock can occur.

Case 6 : Deadlocks in Disk Sharing

Occurs when competing processes send conflicting commands to access a disk.



- Two processes are each waiting for an I/O request to be filled: Cylinder 20, 310.
- Neither can be satisfied because the device puts each request on hold when it tried to fulfill the other processes.
- Without controls to regulate use of disk drive, **competing processes could send conflicting commands** and deadlock the system.

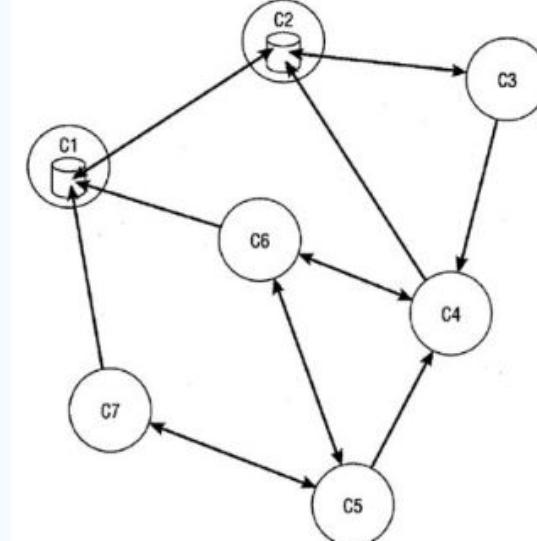


Case 7 : Deadlocks in a Network

Occurs when the network doesn't have protocols to control the flow of messages through the network.



- Consider seven computers on a network, each on different nodes.
- Direction of the arrows indicates the flow of messages.
- Deadlock occurs if all the available buffer space fills.



Understanding

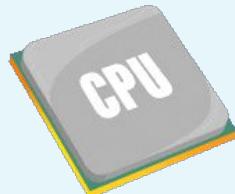
Physical



Tape Drive



Printer



CPU



Memory

Logical

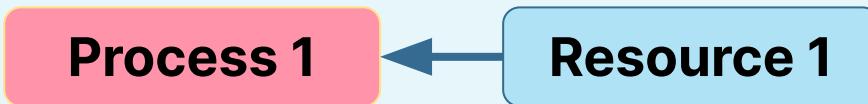


Files

Four Condition for Deadlocks

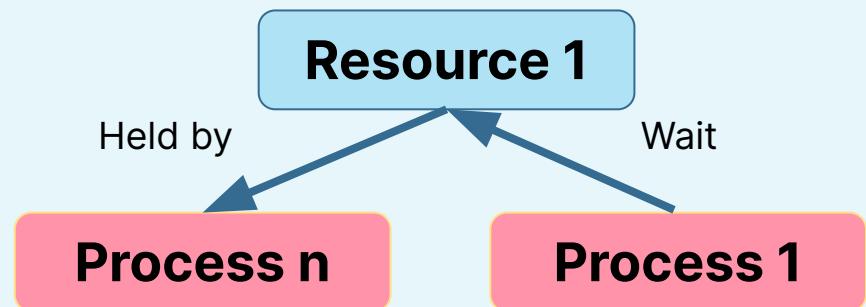
1) Mutual Exclusion

A resource is held in a non-sharable mode (one process use resource at a time. If another process requests, it has to wait)



2) Hold and Wait

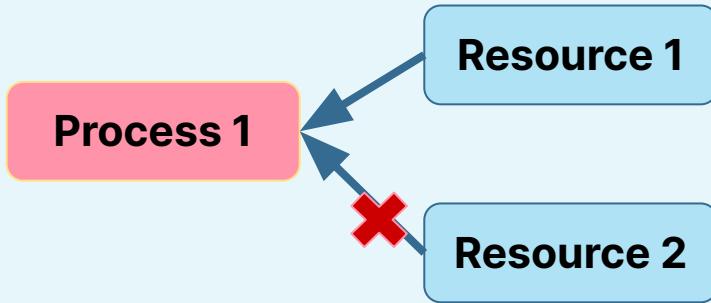
A process holds a resource & waits to acquire another resource that is used by another process.



Four Condition for Deadlocks

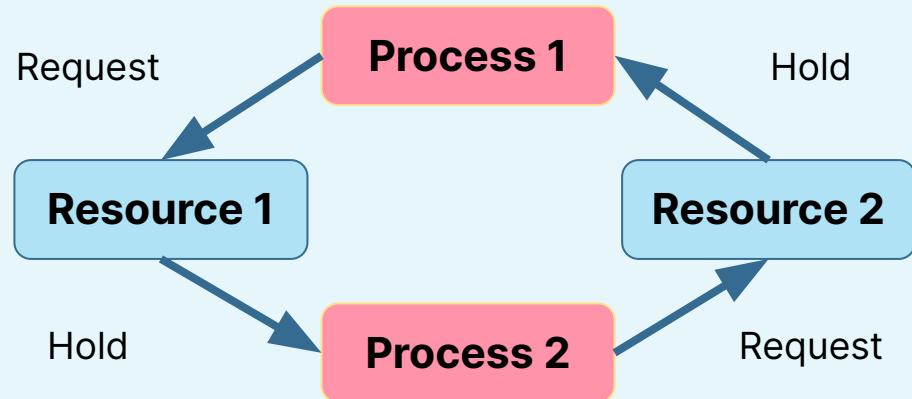
3) No Preemption

Resources cannot be pre-empted; can only be released after the process finished using it.



4) Circular Wait

P1 is waiting for R1 which is held by P2, P2 is waiting for R2 which is held by P1



Modeling Deadlocks Using Directed Graphs (Holt, 1972)



Requesting
 P_i is waiting for R_j



Holding
 P_i is holding R_j

- * Deadlock may (or may not) exists if a cycle is found; If no cycle, no deadlock

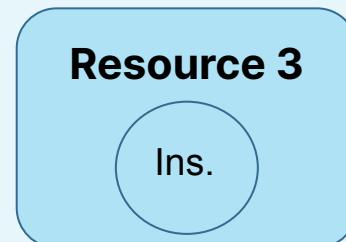
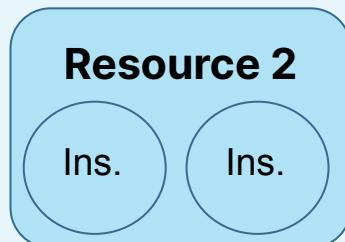
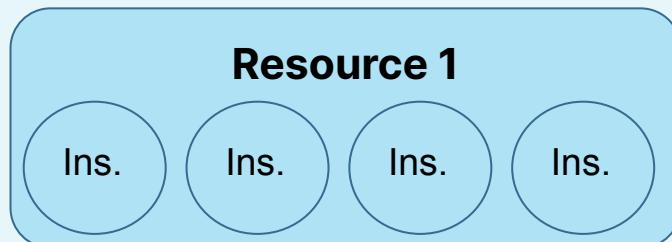
- Resource types R_1, R_2, \dots, R_m
- eg : *CPU cycles, memory space, I/O devices*

Each resource type R_i has W_i instances.

R_1 has 4 instances eg 4 printers in a system

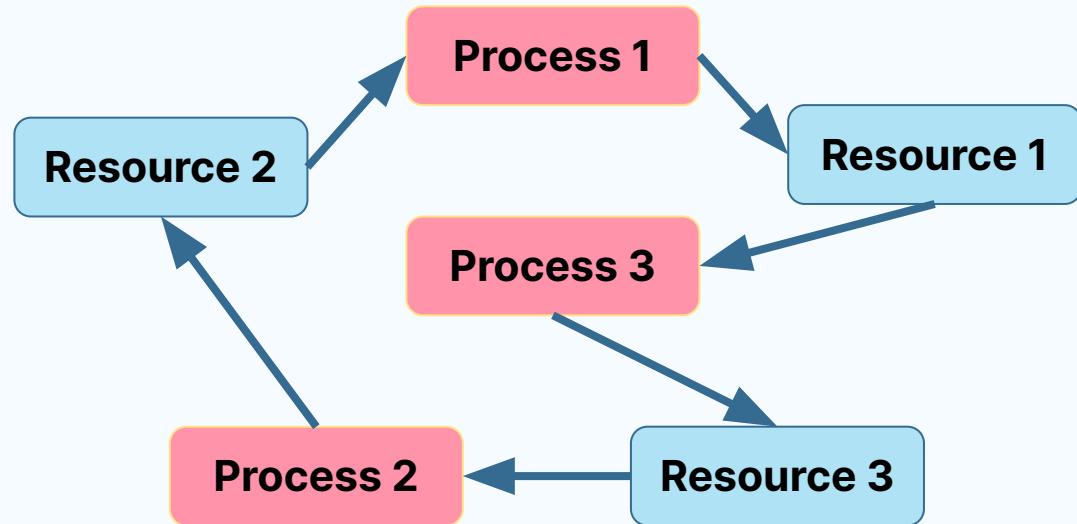
R_2 has 2 instances eg 2 disk drive in a system

R_3 has 1 instance eg 1 CPU in a system



Check your understanding

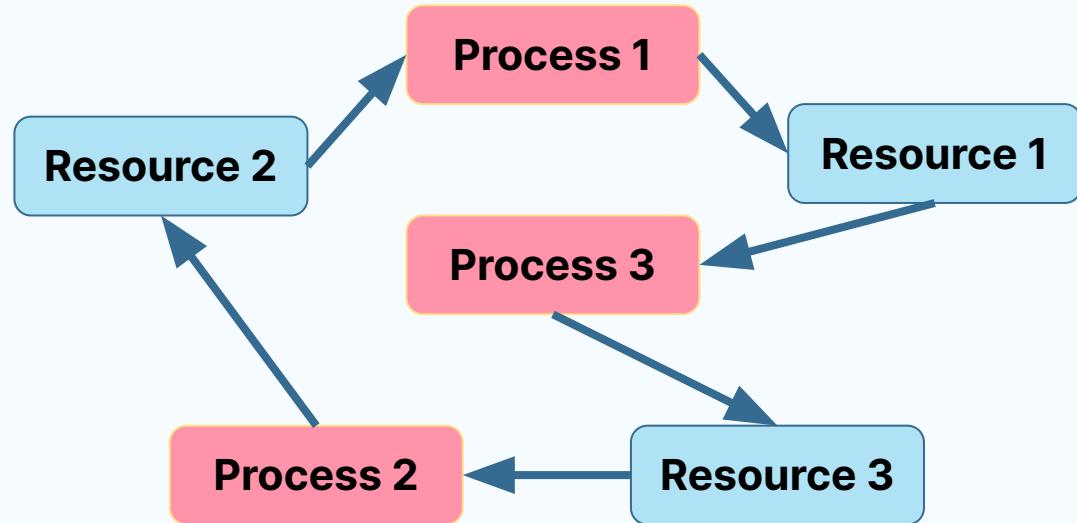
Is this a Deadlock ?



Resource-Allocation Graph 1

Check your understanding

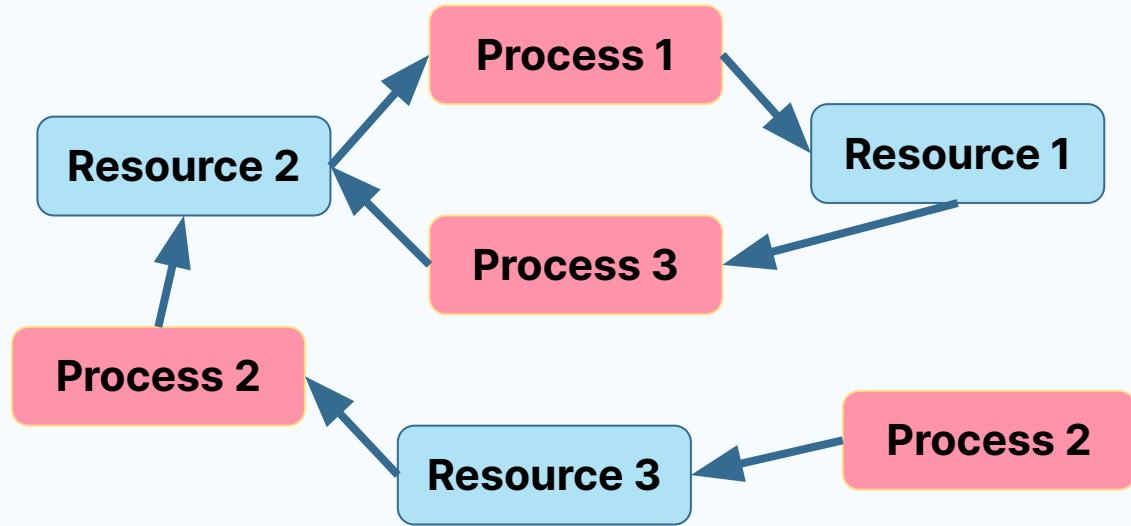
Is this a Deadlock ?



Resource-Allocation Graph 1

Check your understanding

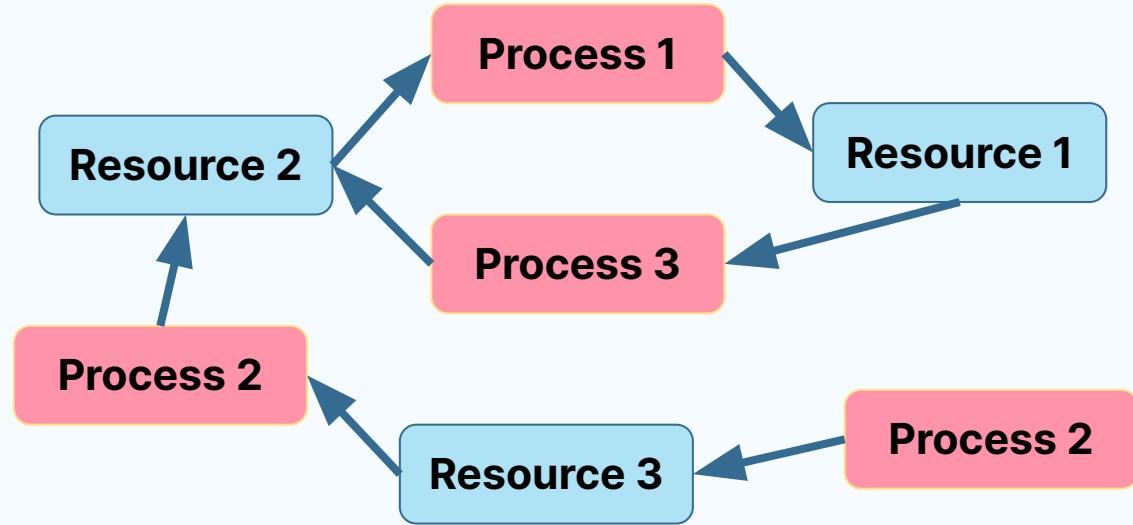
Is this a Deadlock ?



Resource-Allocation Graph 2

Check your understanding

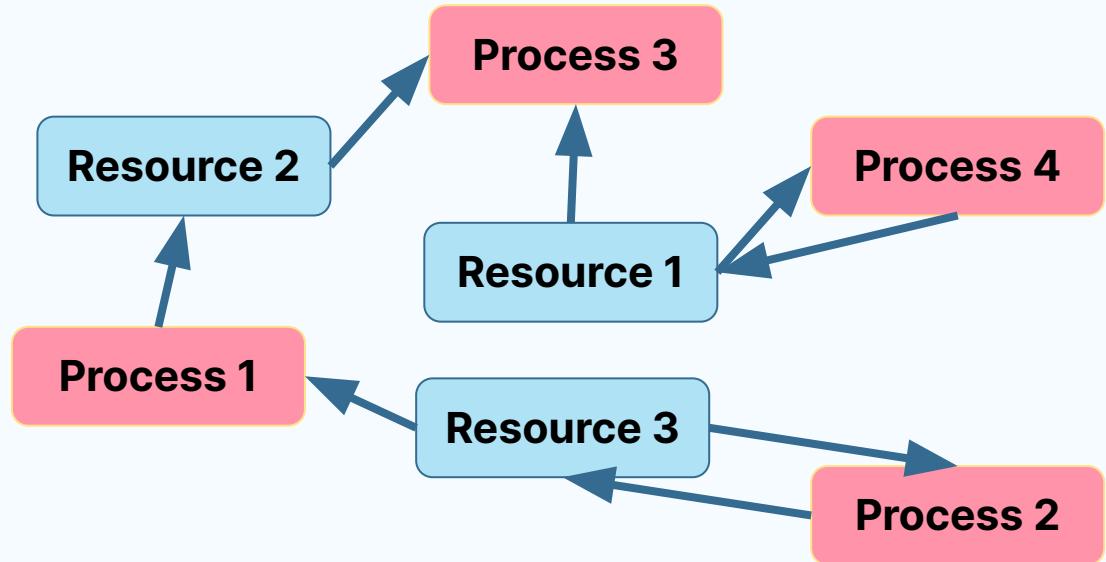
Is this a Deadlock ?



Resource-Allocation Graph 2

Check your understanding

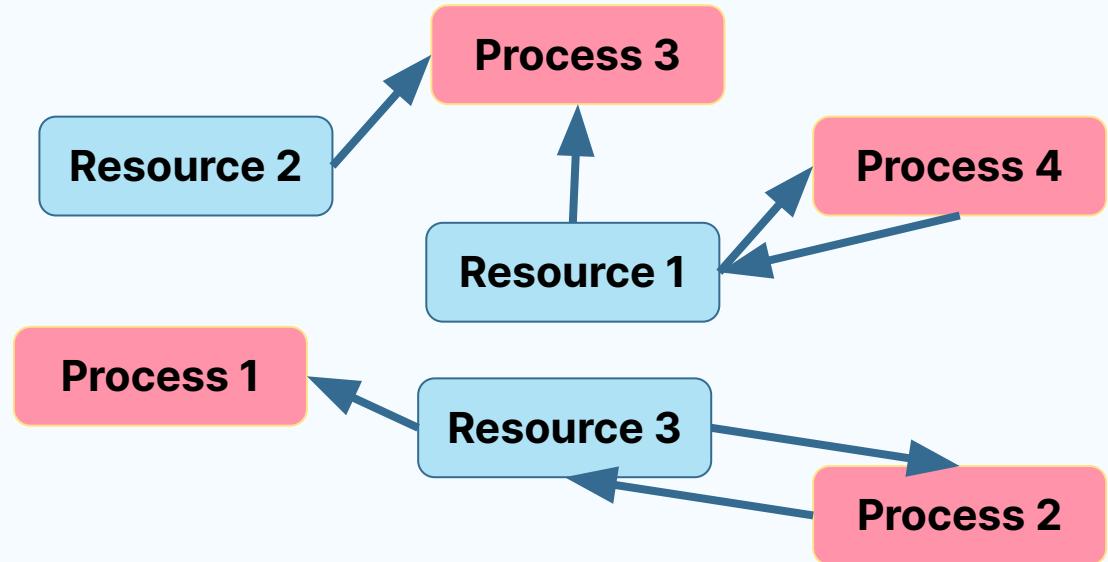
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

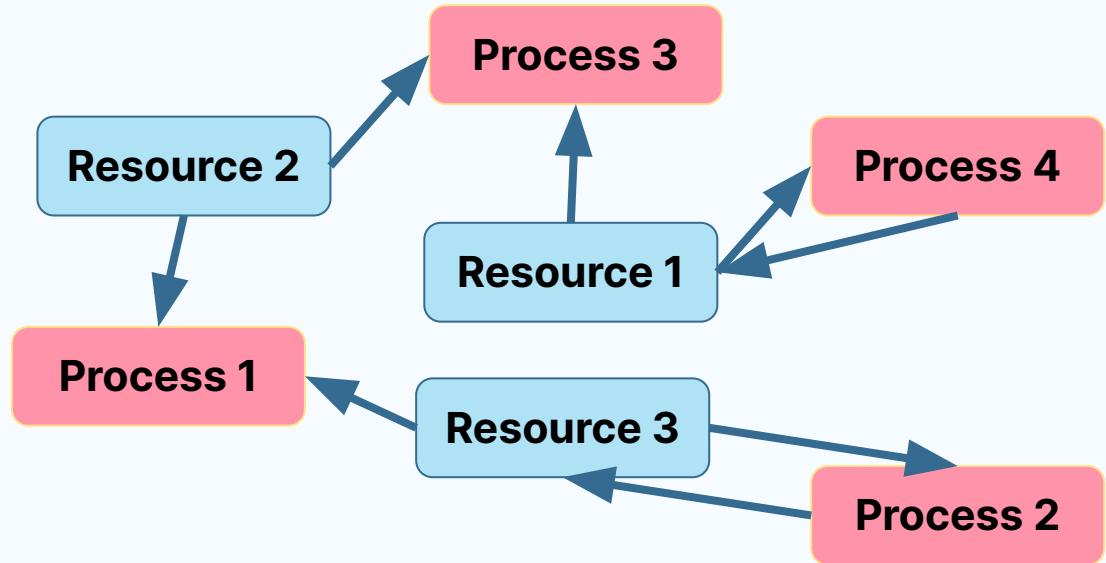
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

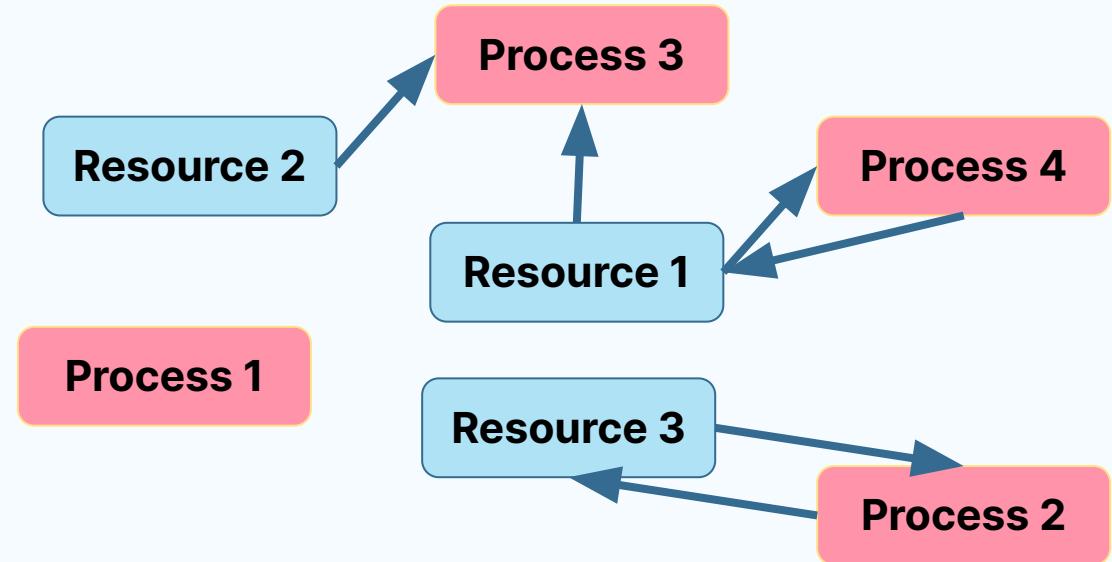
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

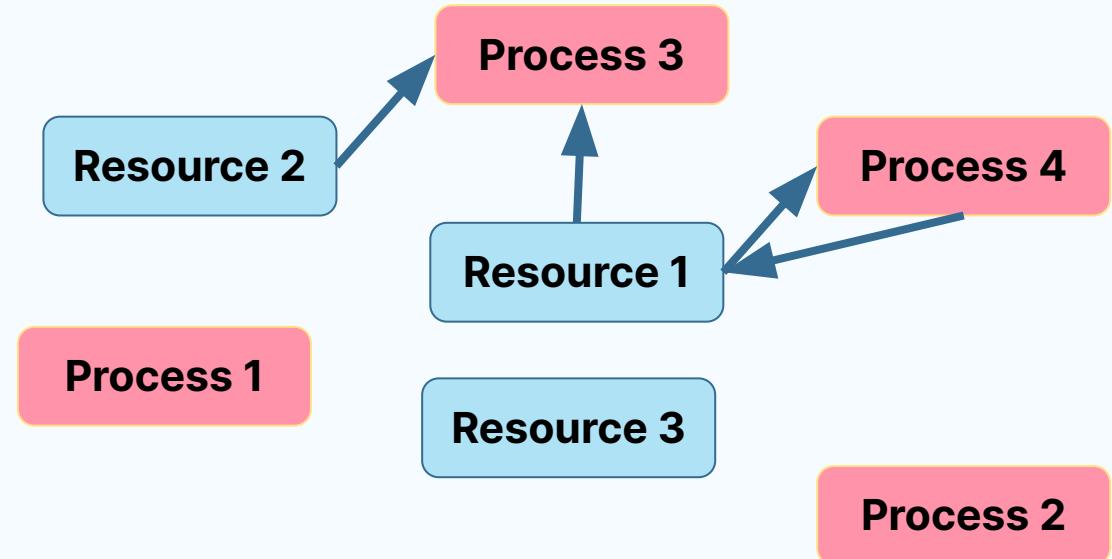
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

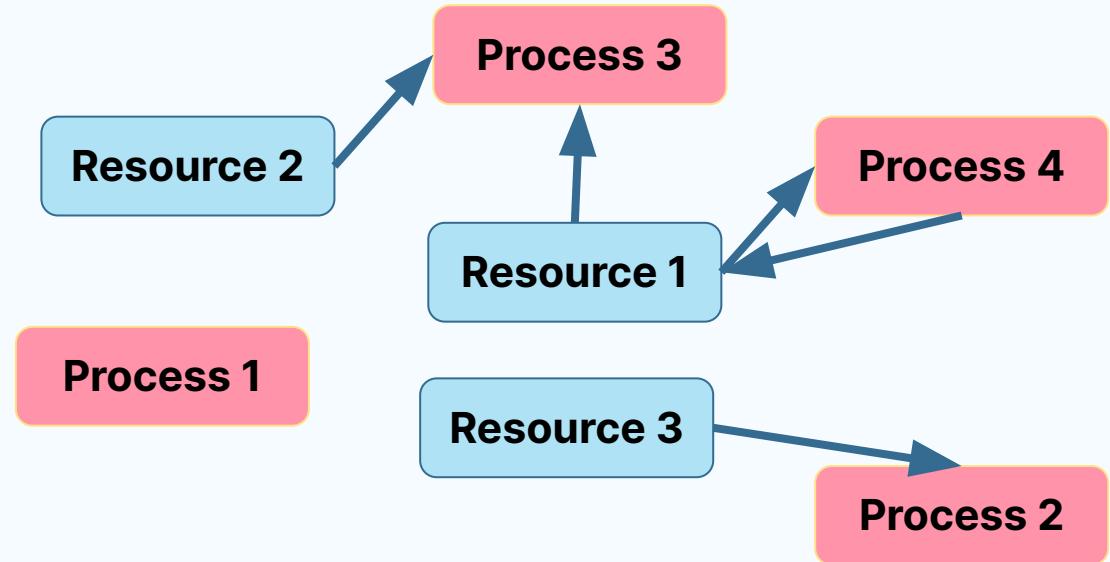
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

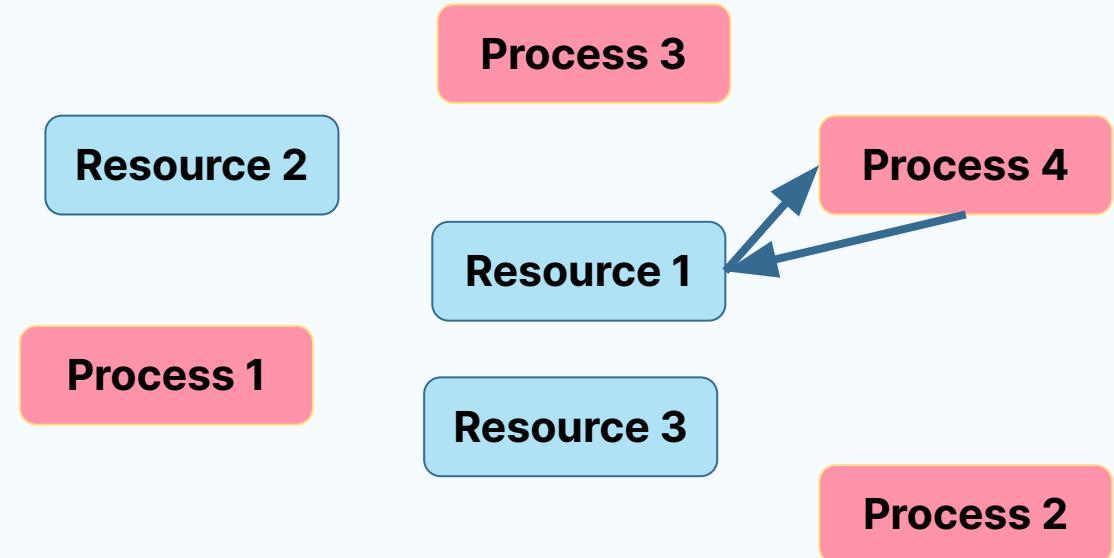
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

Is this a Deadlock ?



Resource 2

Process 3

Process 1

Resource 1

Process 4

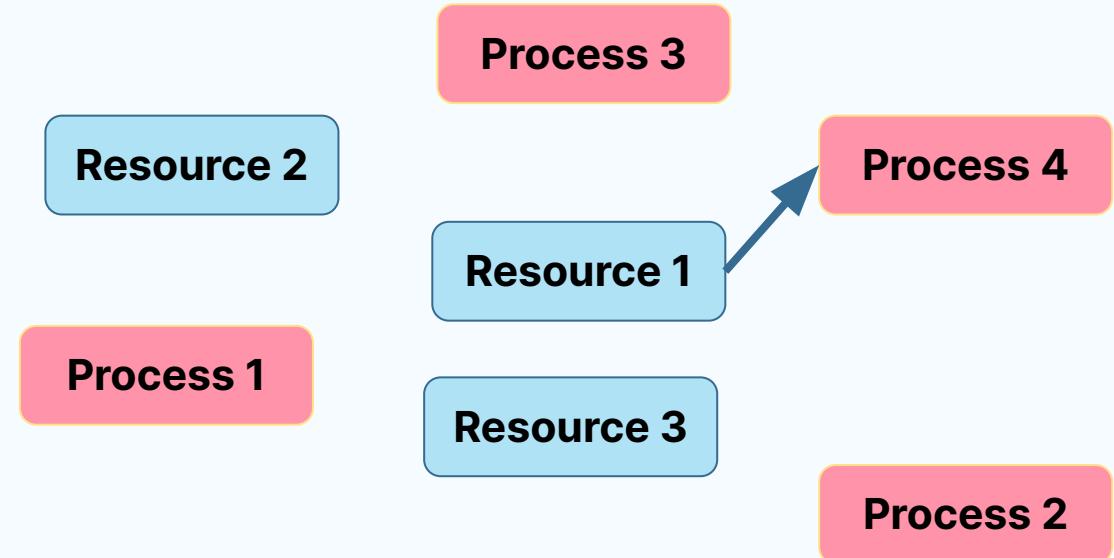
Resource 3

Process 2

Resource-Allocation Graph 3

Check your understanding

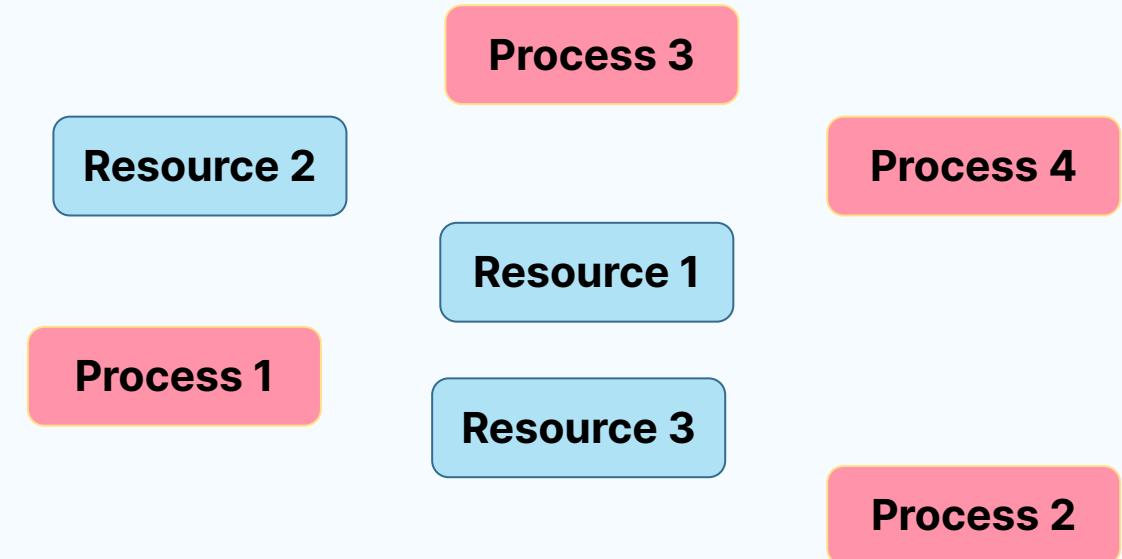
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

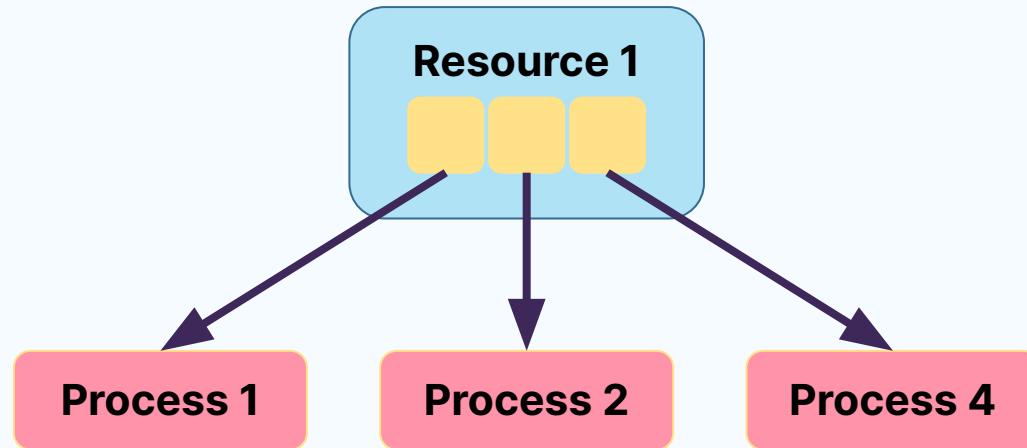
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

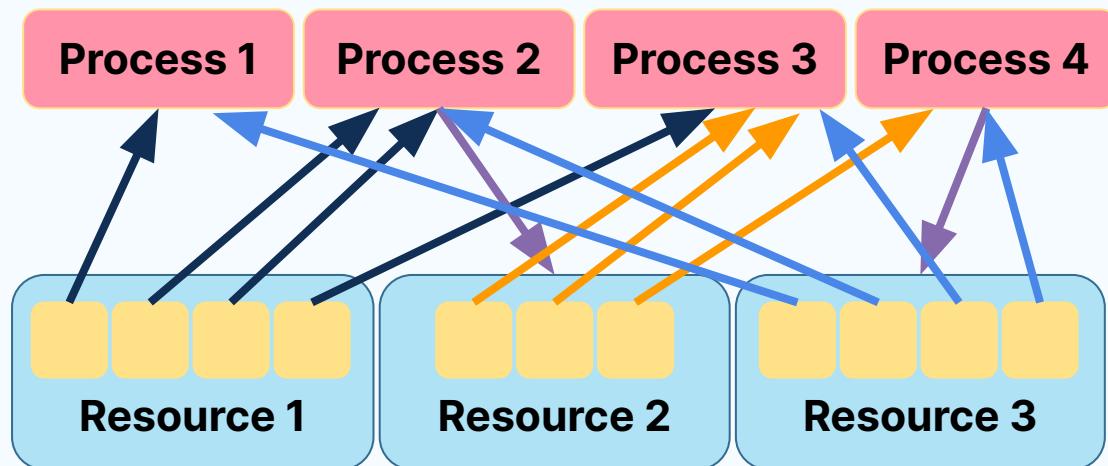
Is this a Deadlock ?



Resource-Allocation Graph 4

Check your understanding

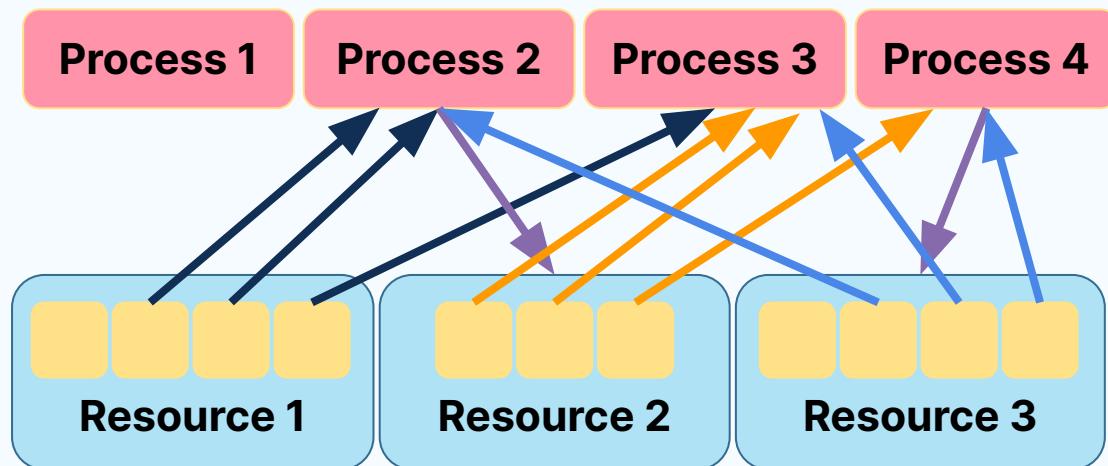
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

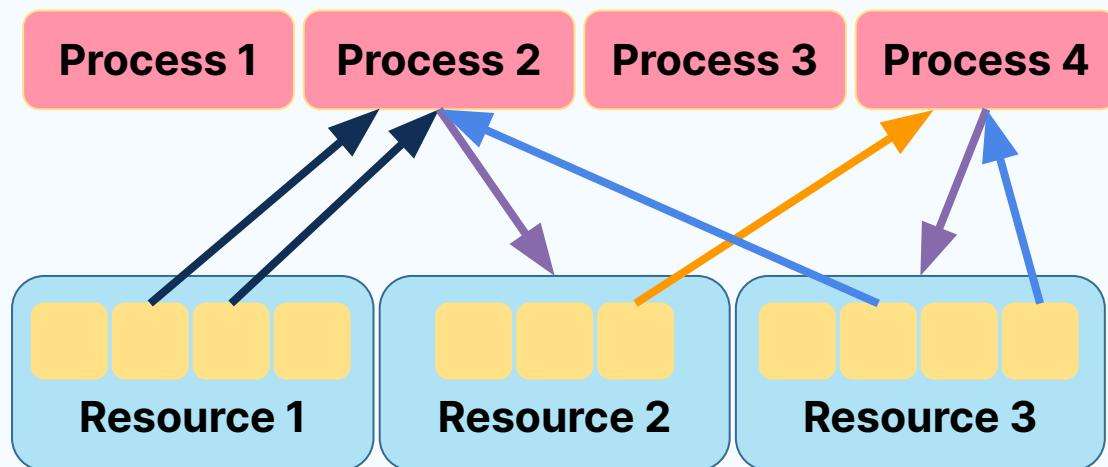
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

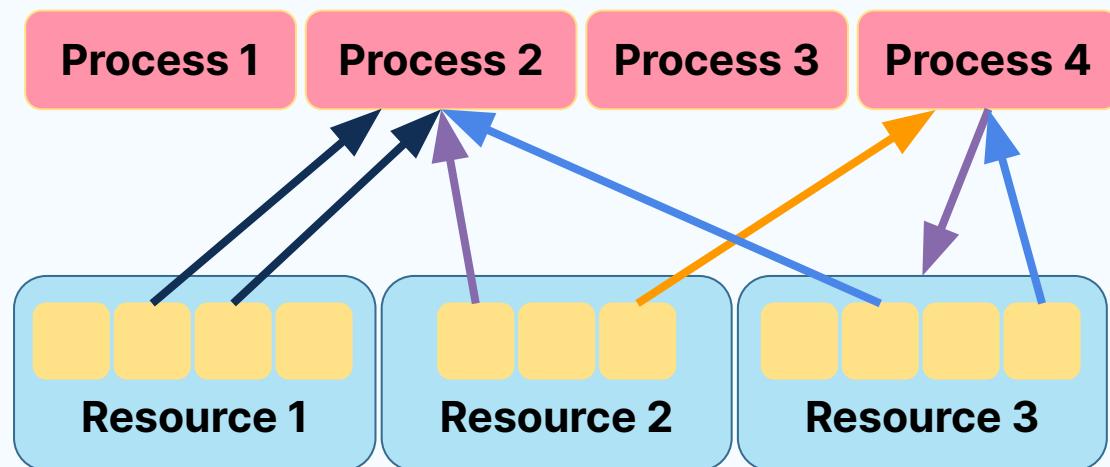
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

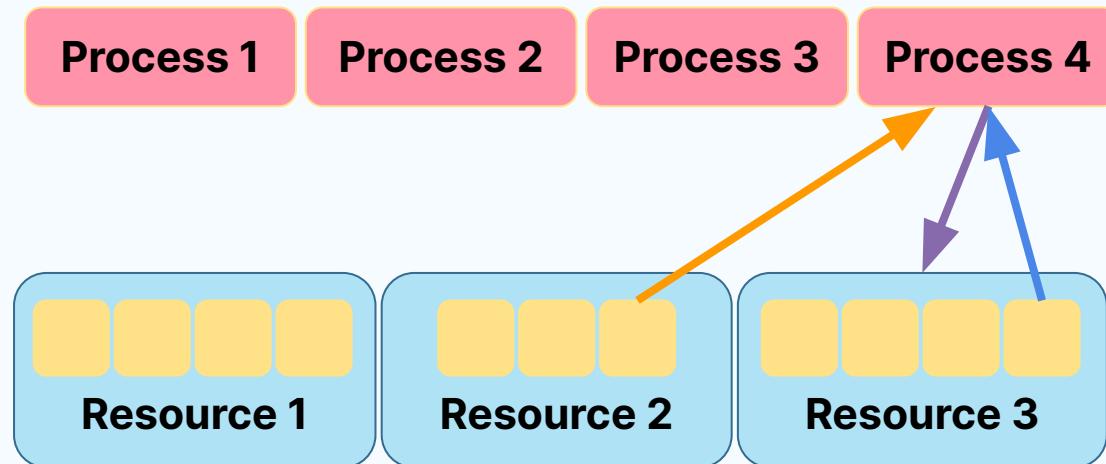
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

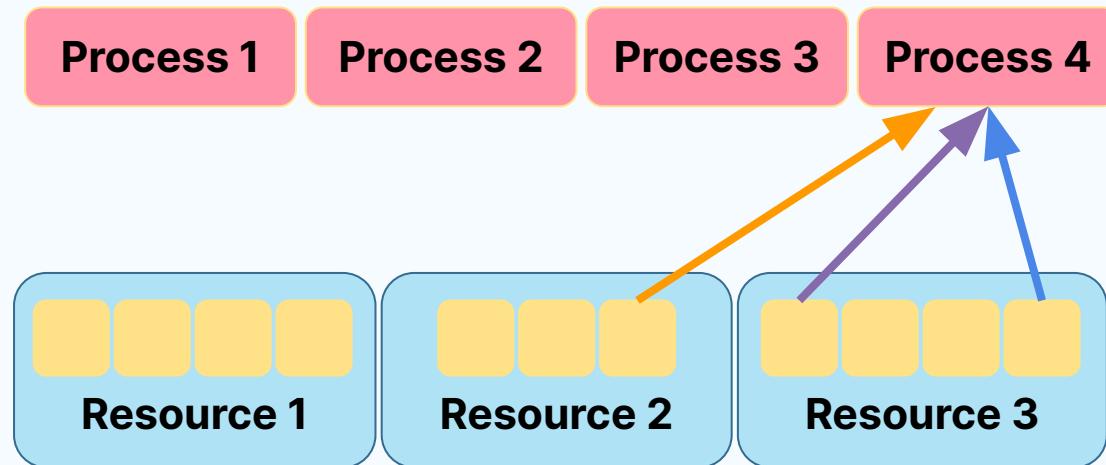
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

Is this a Deadlock ?



Check your understanding

Is this a Deadlock ?



Process 1

Process 2

Process 3

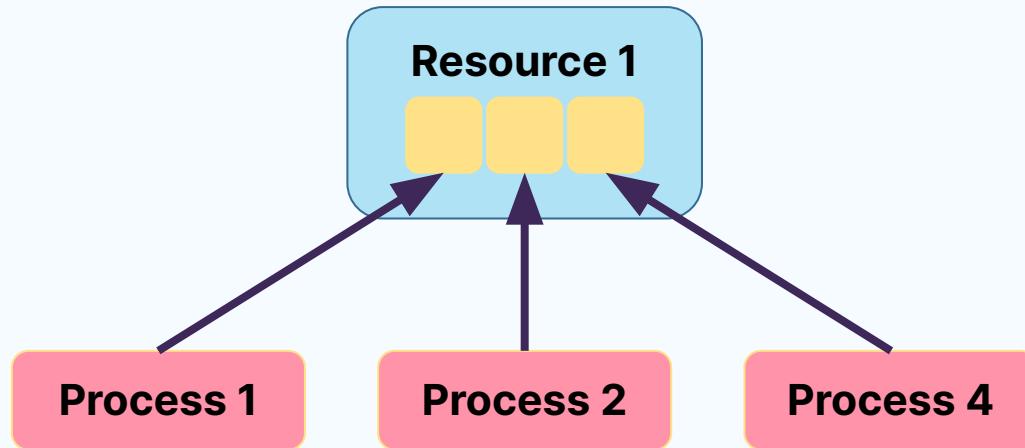
Process 4



Resource-Allocation Graph 5

Check your understanding

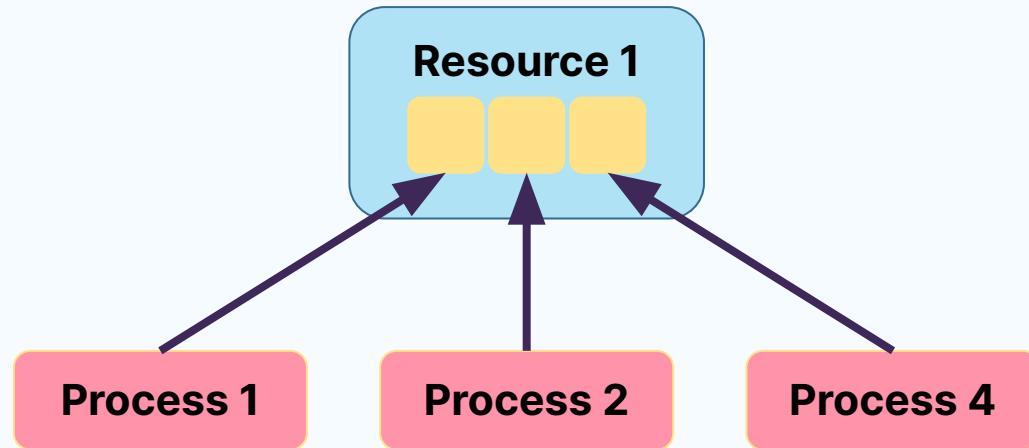
Is this a Deadlock ?



Resource-Allocation Graph 6

Check your understanding

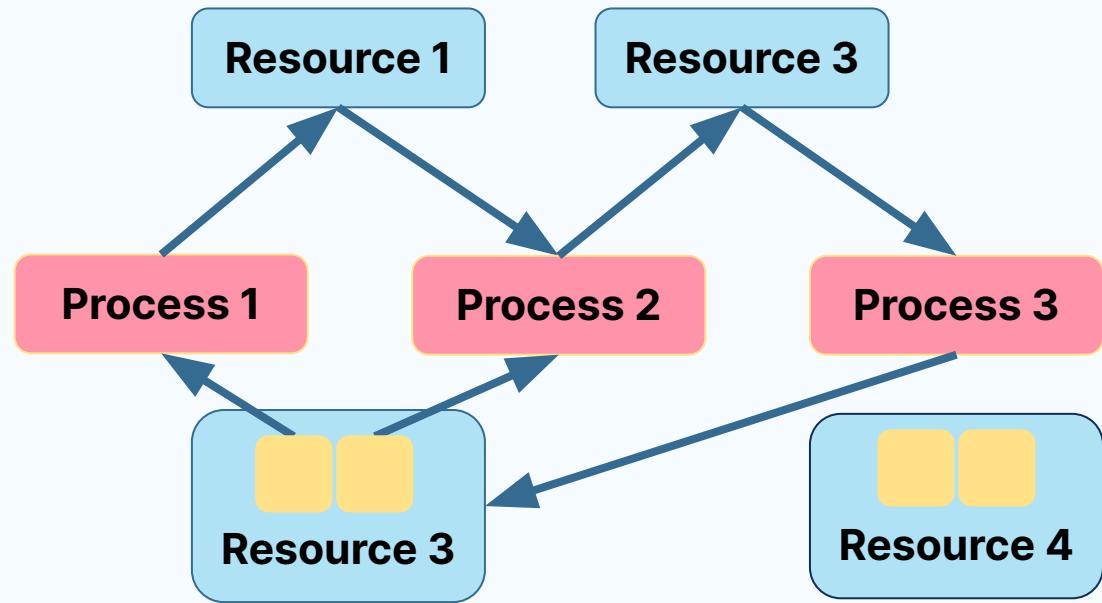
Is this a Deadlock ?



Resource-Allocation Graph 6

Check your understanding

Is this a Deadlock ?



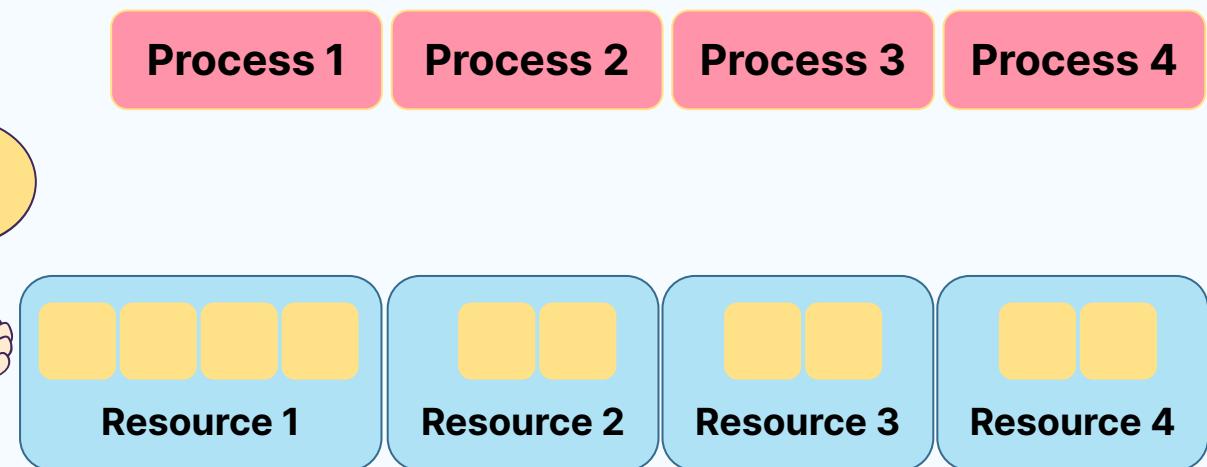
Resource-Allocation Graph 7

Consider a system of 4 processes, $P = \{P1, P2, P3, P4\}$ and 4 resources types, $R = \{R1, R2, R3, R4\}$. Assume all the resources are non-sharable and the numbers of instances for each resource type are 4, 2, 2 and 2 respectively. Given the process states as follows:

- $P1$ holds all instances of $R1$, an instance of $R3$ and an instance of $R4$ and requests an instance of $R2$
- $P2$ holds an instance of $R2$ and requests an instance of $R4$
- $P3$ holds an instance of $R2$ and requests an instance of $R3$
- $P4$ holds an instance of $R3$ and $R4$

Check your understanding

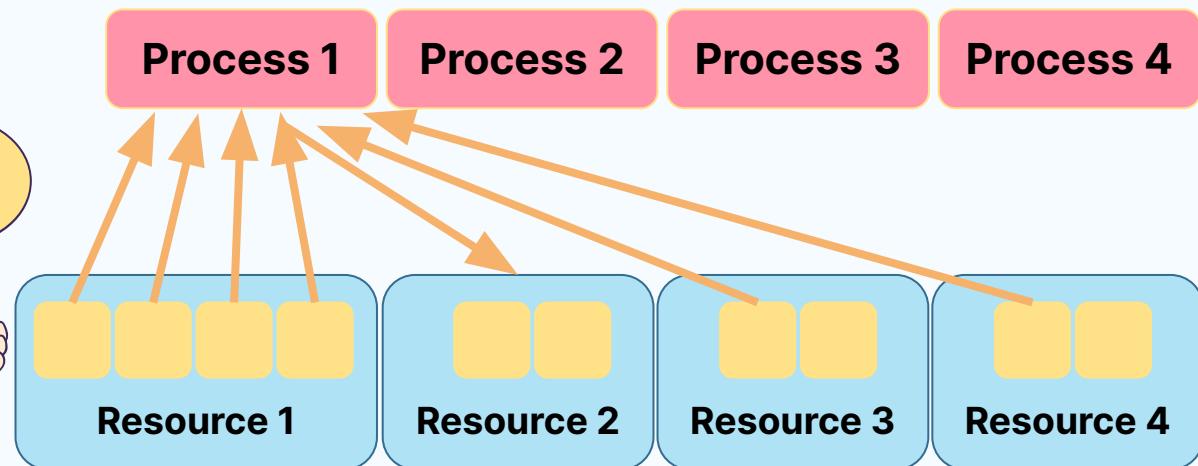
P1 holds all instances of R1, an instance of R3 and an instance of R4 and requests an instance of R2.



Resource-Allocation Graph 8

Check your understanding

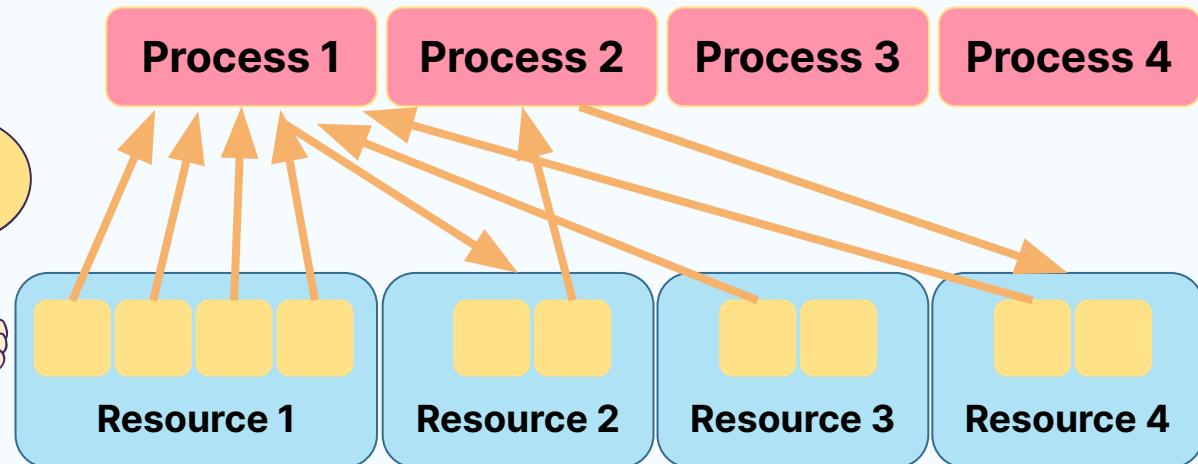
P1 holds all instances of R1, an instance of R3 and an instance of R4 and requests an instance of R2.



Resource-Allocation Graph 8

Check your understanding

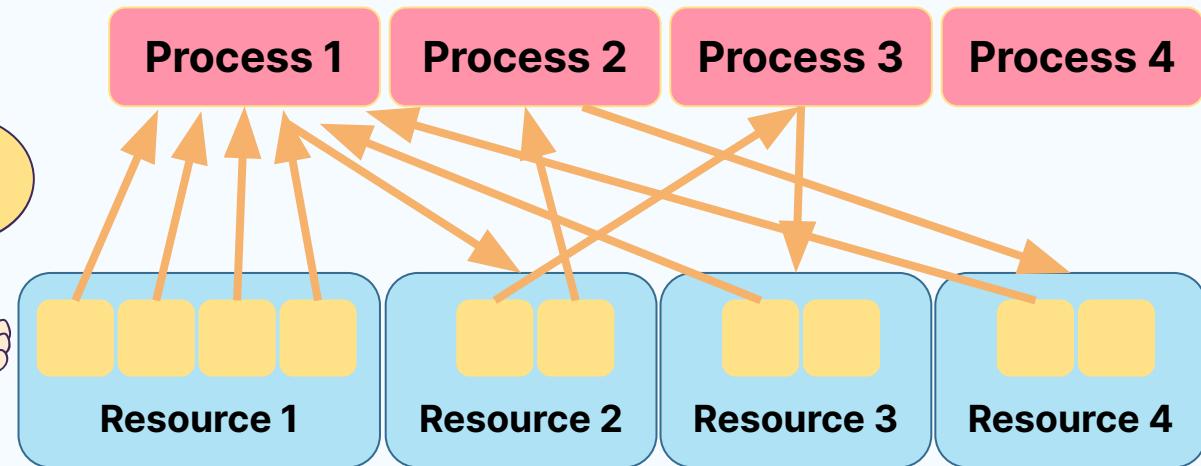
P2 holds an instance of R2
and requests an instance
of R4



Resource-Allocation Graph 8

Check your understanding

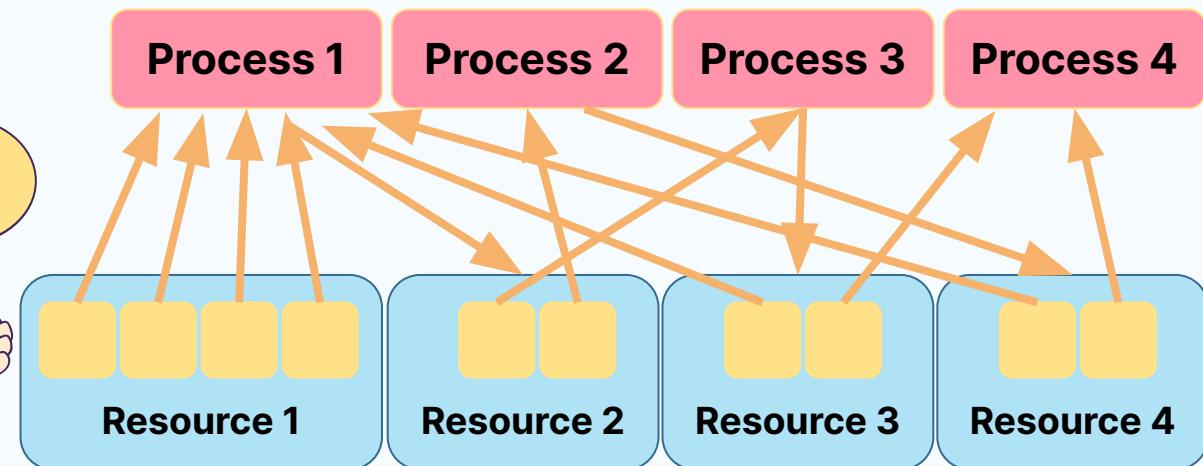
P3 holds an instance of R2
and requests an instance
of R3



Resource-Allocation Graph 8

Check your understanding

P4 holds an instant of R3 and R4.



Resource-Allocation Graph 8

Deadlock handling mechanisms

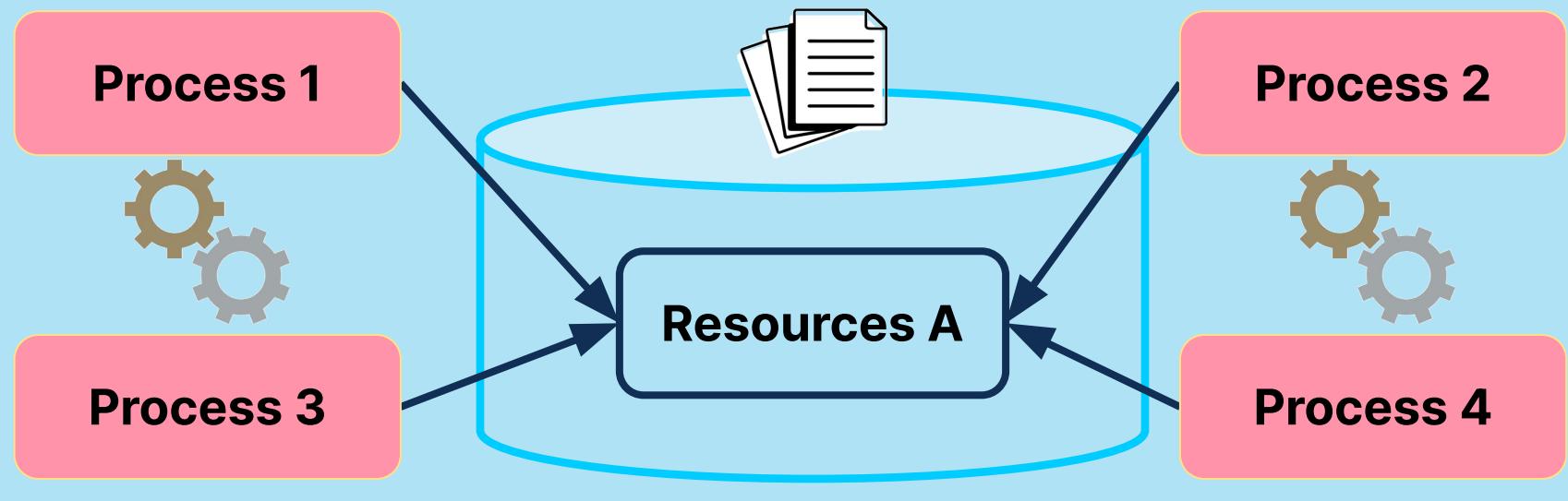
Prevent one of the four conditions from occurring.

Avoid the deadlock if it becomes probable.

Detect the deadlock when it occurs and recover from it gracefully.

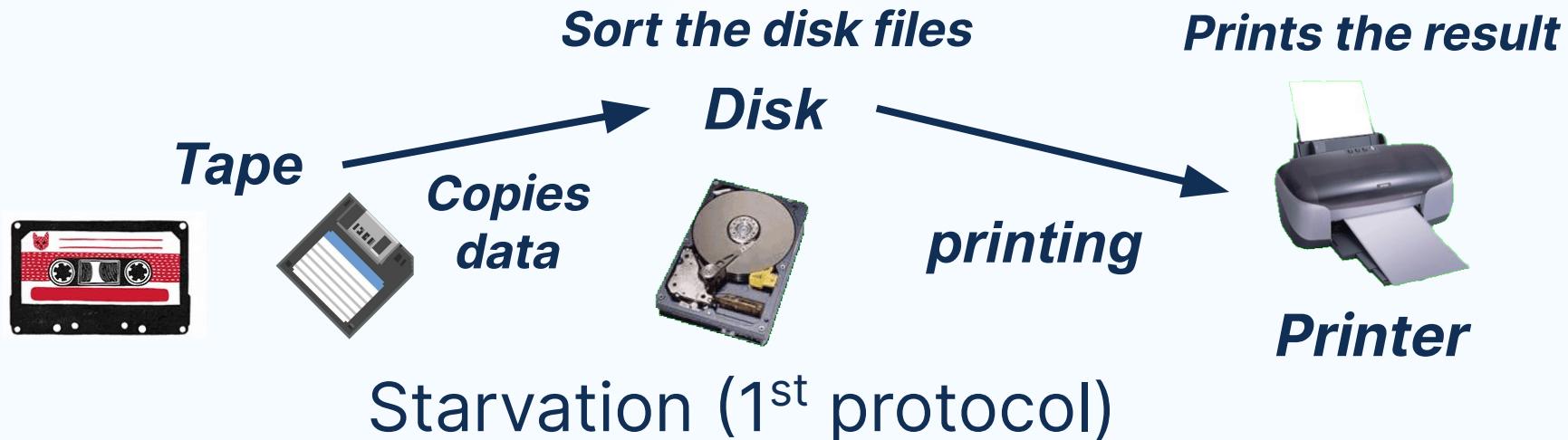
- 1) Mutual Exclusion
- 2) Hold and Wait
- 3) No Preemption
- 4) Circular Wait

Mutual Exclusion – must apply for non-sharable resources; not required for mutually exclusive access. E.g., Read-only files.



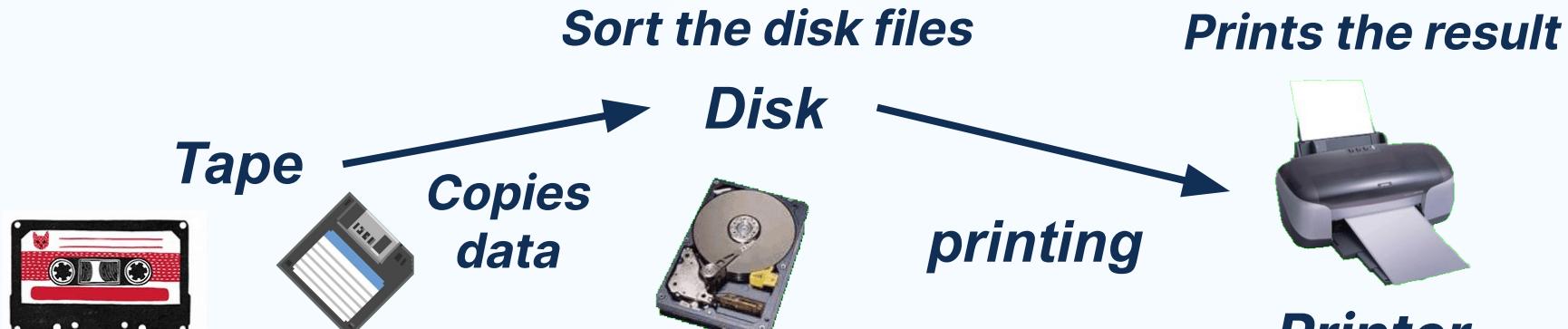
Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources. (2 protocols)

1st protocol: Holding all resources until the end of execution



Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources. (2 protocols)

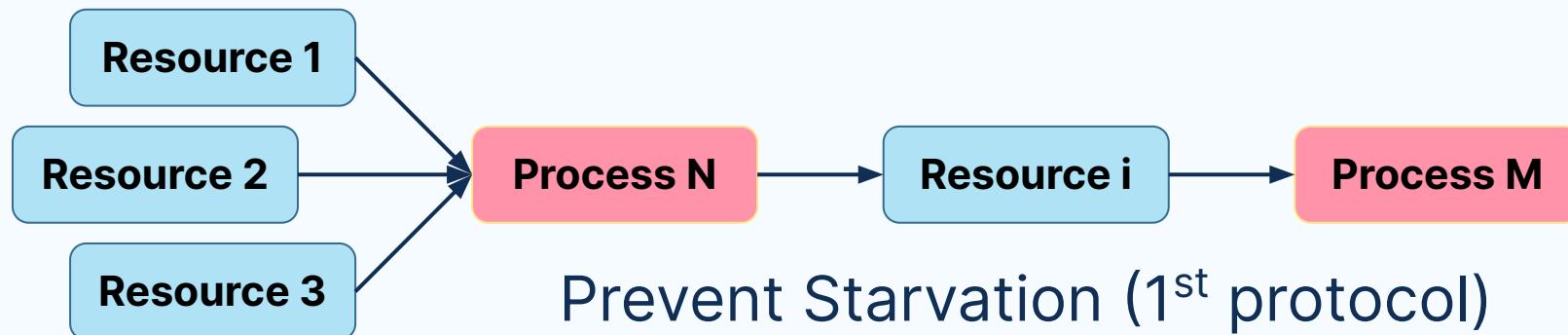
2nd protocol: After using a resource must release and then request again for another execution.



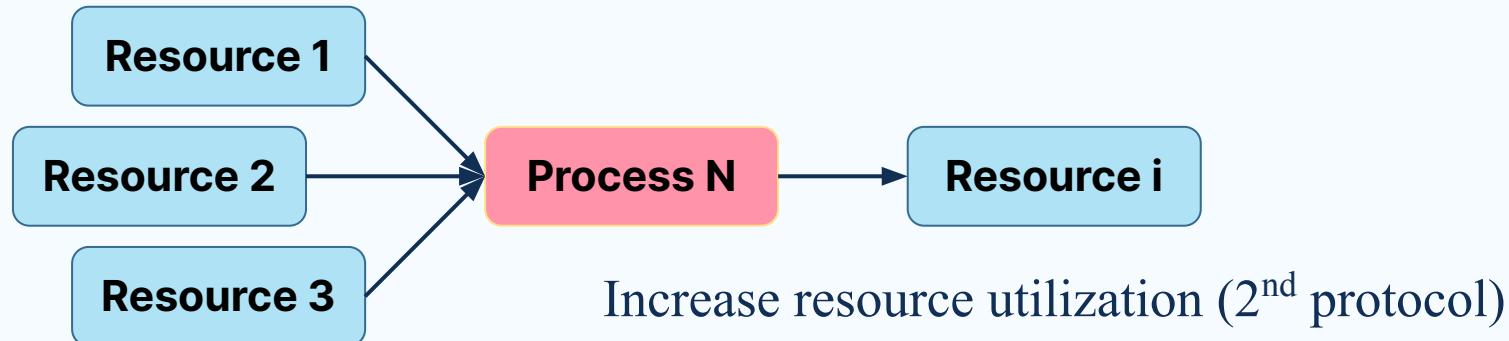
Low resource utilization (2nd protocol)

No Preemption – Could be bypassed by allowing OS to deallocate resources from jobs.

(resources being held are preempted then given to another process which is waiting) preempted resources will then put into the list for the process is waiting. (Round Robin environment)



No Preemption – If a process that is holding some resources requests another resource that can be immediately allocated to it, then all resources currently being held will not be released.



Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration. Forces each job to request its resources in ascending order.



$$\begin{aligned} F(\text{Printer}) &= 1 \\ F(\text{disk drive}) &= 2 \\ F(\text{tape drive}) &= 3 \end{aligned}$$



ooo

$$F(\text{tape drive}) = 1$$

$$F(\text{disk drive}) = 5$$

$$F(\text{printer}) = 12$$



$$F(R_i) = 1$$

Resource I

$$F(R_j) = 5$$

Resource J

Process N

A process Pn
using tape
drive(1) then
disk drive(5)

$F(\text{tape drive}) = 1$

$F(\text{disk drive}) = 5$

$F(\text{printer}) = 12$

or has released any
resource R_i
if $F(R_i) \geq F(R_j)$

$$F(R_i) = 12$$

Resource I

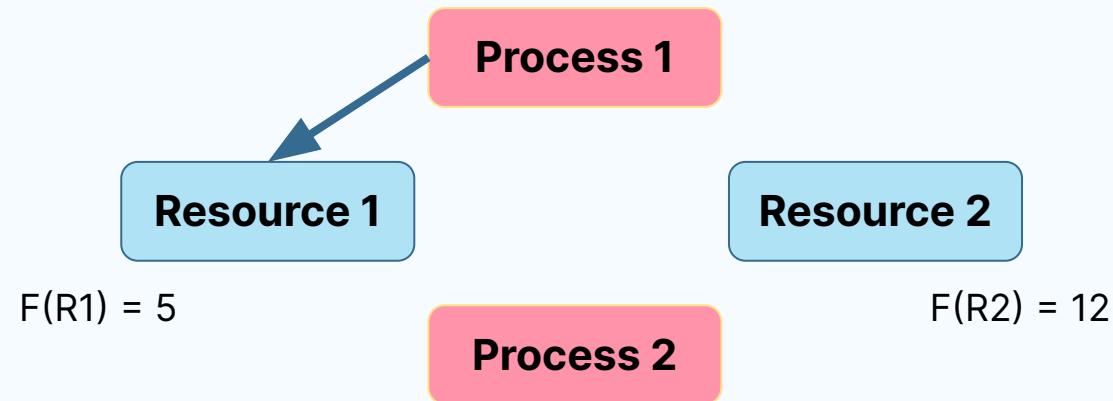
$$F(R_j) = 5$$

Resource J

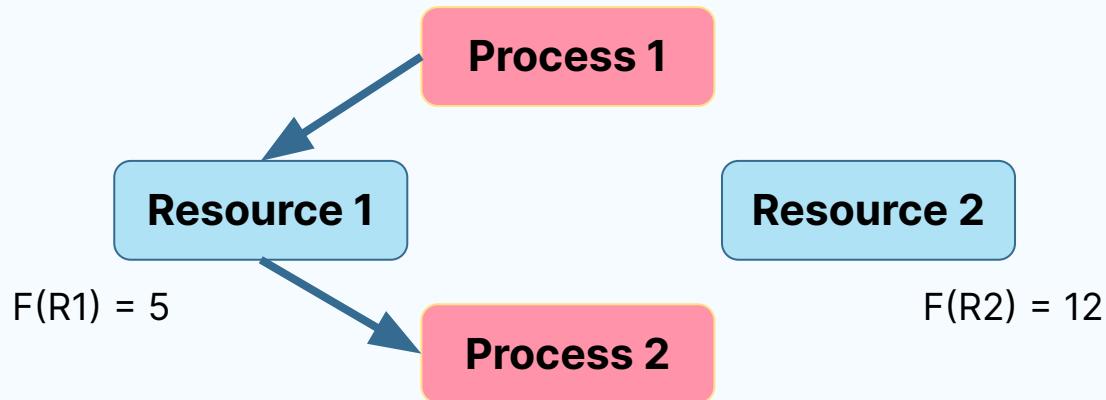
Process N

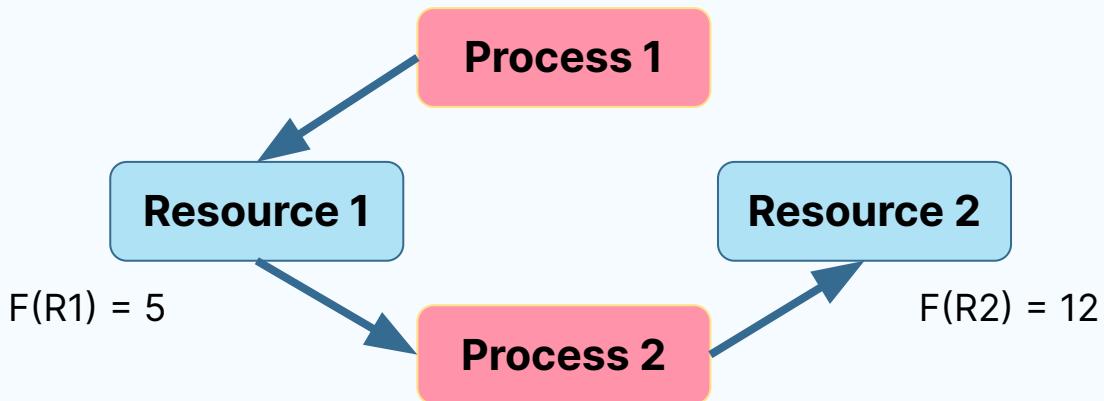
Pn holding
printer(12) at the
same time
requesting tape
drive(1)

○○○



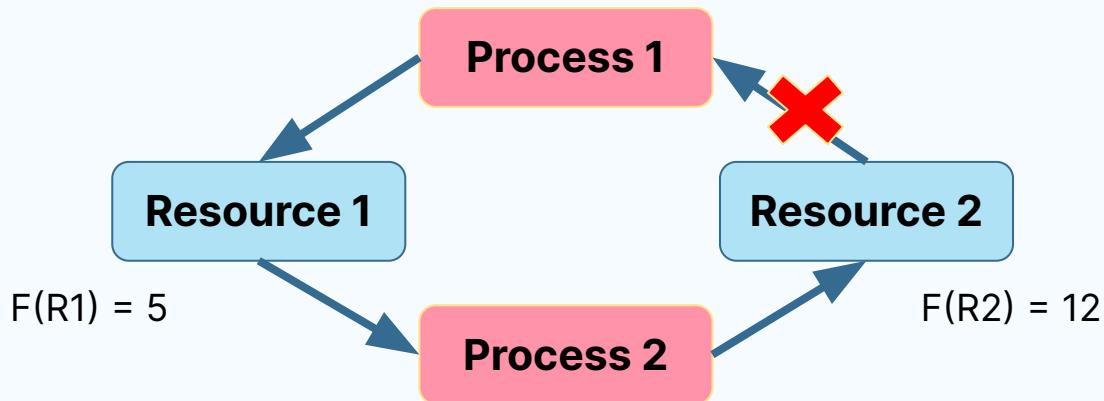
○○○



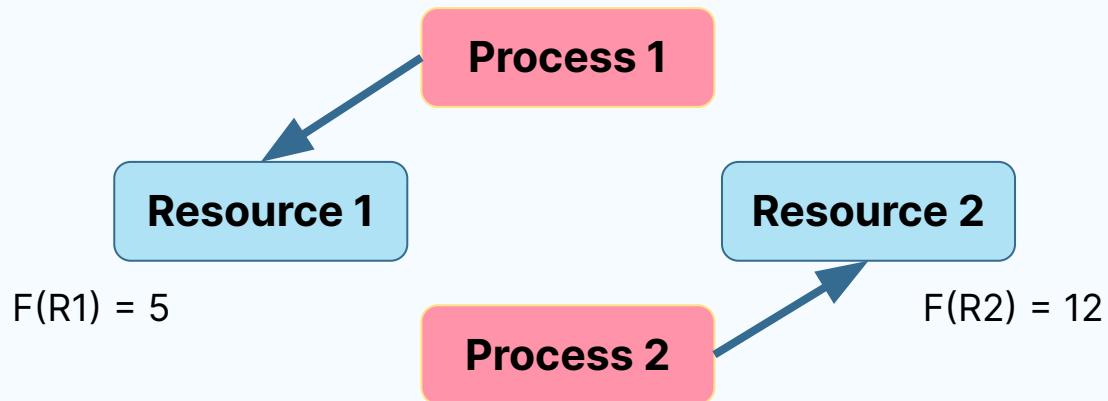


○○○

$$F(R2) \geq F(R1)$$



○○○



Deadlock Avoidance

- Even if the OS is unable to remove 1 condition for deadlock, it can avoid one if the system knows ahead of time the sequence of requests associated with each of the active processes.
- Dijkstra's Bankers Algorithm (1965) used to regulate resources allocation to avoid deadlock.

Safe State

if there exists a safe sequence of all processes where they can all get the resources needed.

Unsafe State

doesn't necessarily lead to deadlock, but it does indicate that system is an excellent candidate for one.

Safe State

Customer	Loan amount	Maximum credit	Remaining credit
C1	0	4,000	4,000
C2	2,000	5,000	3,000
C3	4,000	8,000	4,000

Total loaned: \$6,000

Total capital fund: \$10,000



Unsafe State

Customer	Loan amount	Maximum credit	Remaining credit
C1	2,000	4,000	2,000
C2	3,000	5,000	2,000
C3	4,000	8,000	4,000

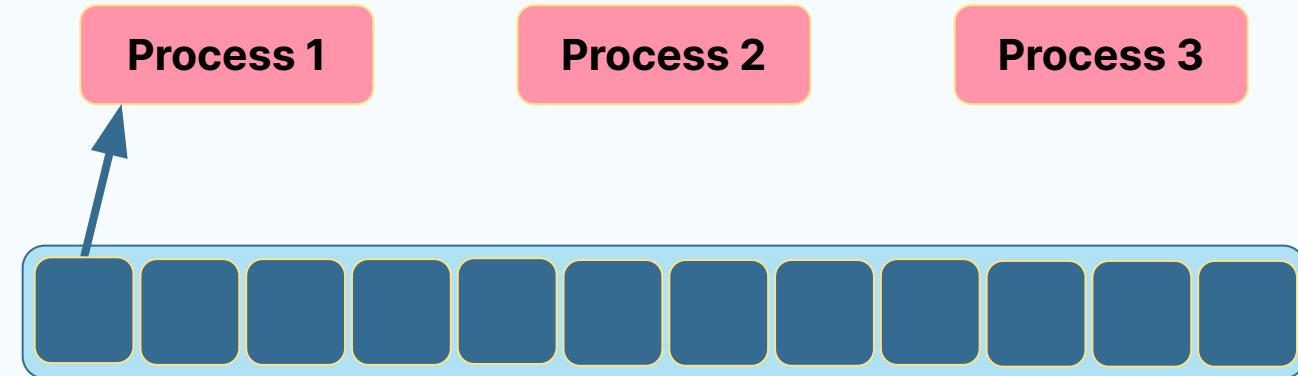
Total loaned: \$9,000

Total capital fund: \$10,000



At t0

Deadlock Avoidance



	Max Needs	Current Needs
--	-----------	---------------

Process 1	10	5
Process 2	4	2
Process 3	9	7

ooo



At t0



Deadlock Avoidance

Allocation 5

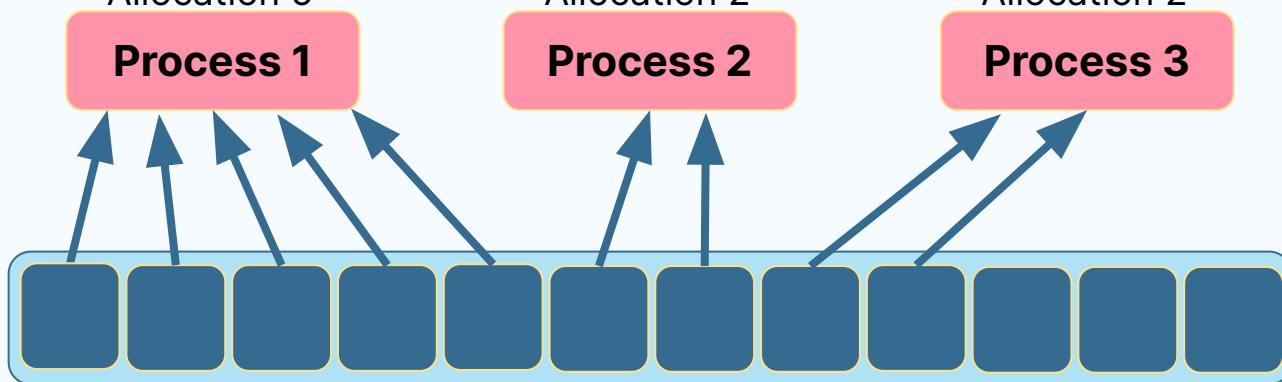
Process 1

Allocation 2

Process 2

Allocation 2

Process 3



Max Needs

Process 1

10

Current Needs

5

Process 2

4

2

Process 3

9

7

ooo



At t0



Deadlock Avoidance

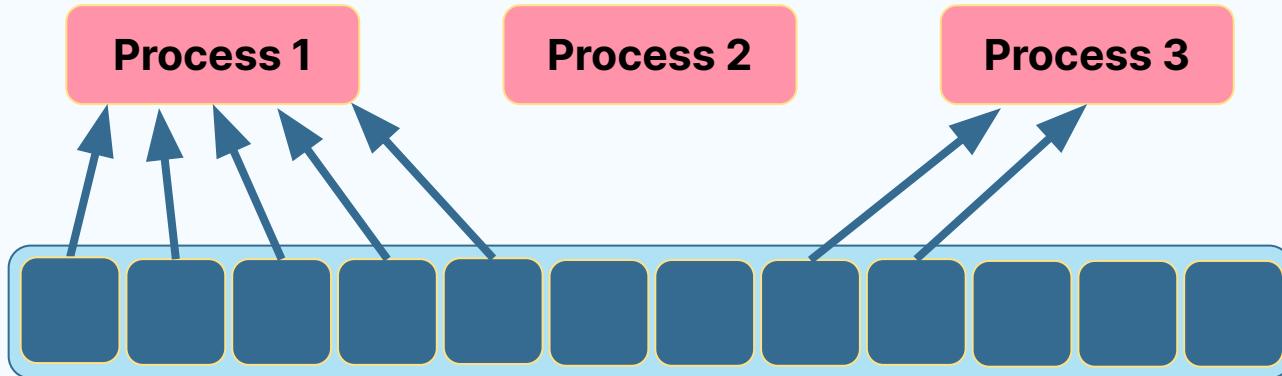
Allocation 5

Process 1

Process 2

Allocation 2

Process 3



Max Needs

Process 1

10

Current Needs

5

Process 2

4

2

Process 3

9

7

ooo



At t0

Deadlock Avoidance

Process 1

Process 2

Allocation 2

Process 3



Max Needs

Current Needs

Process 1 10

5

Process 2 4

2

Process 3 9

7

ooo



At t0

Deadlock Avoidance

Process 1

Process 2

Process 3



	Max Needs	Current Needs	< P2, P1, P3 >
Process 1	10	5	
Process 2	4	2	Satisfy the
Process 3	9	7	safety condition

ooo



At t1



Deadlock Avoidance

Allocation 5

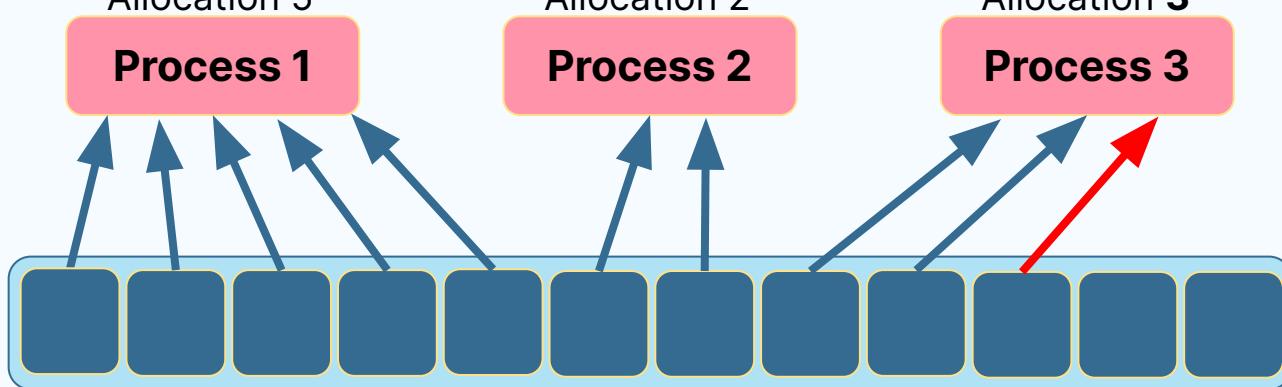
Process 1

Allocation 2

Process 2

Allocation 3

Process 3



Max Needs

Process 1

10

Current Needs

5

Process 2

4

2

Process 3

9

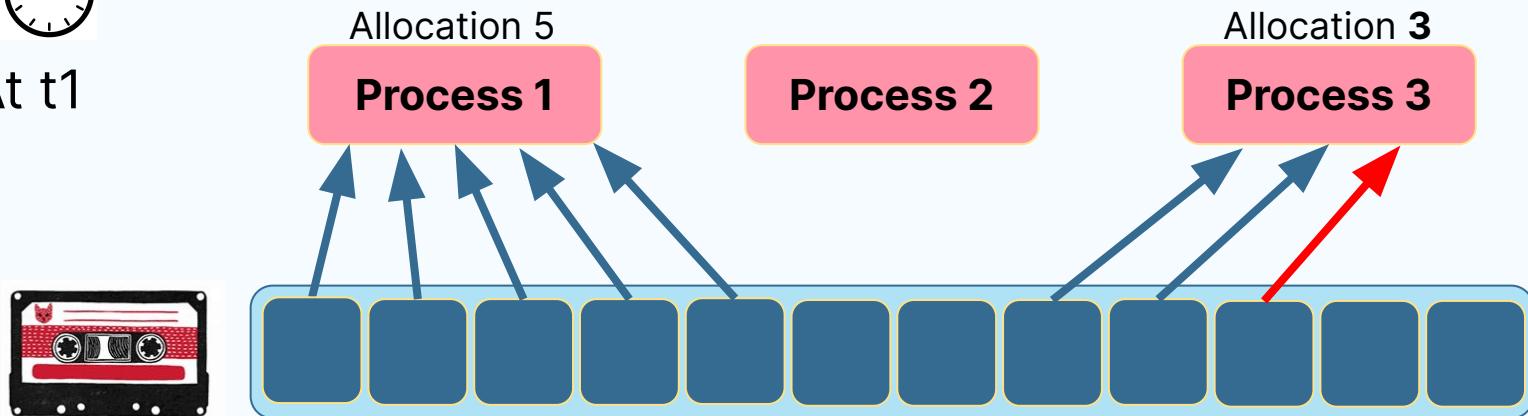
7

ooo



At t1

Deadlock Avoidance



	Max Needs	Current Needs	< P2,	>
Process 1	10	5		
Process 2	4	2		
Process 3	9	7	the unsafe condition	

Banker's Algorithm

Consider a system with 5 processes P0 through P4 and 3 resources types A, B and C.

Resources type A has 10 instances, resource B has 5 and resource C has 7. Suppose that, at time t0, the following snapshot of the system has been taken:

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3				3	3	2
P1	2	0	0	3	2	2						
P2	3	0	2	9	0	2						
P3	2	1	1	2	2	2						
P4	0	0	2	4	3	3						



Process Flow:

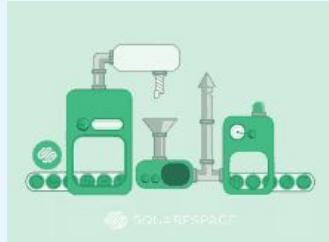
P1 → P3 → P4 → P2 → P0

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			



The content of the matrix Need is defined to be Max - Allocation.

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2	5	3	2
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			



Process Flow:

P1 → P3 → P4 → P2 → P0

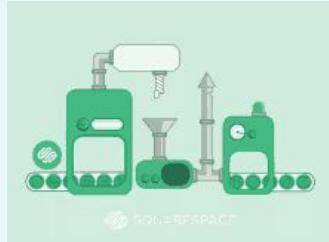
Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2	5	3	2
P2	3	0	2	9	0	2	6	0	0	7	4	3
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			



Process Flow:

P1 → P3 → P4 → P2 → P0

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2	5	3	2
P2	3	0	2	9	0	2	6	0	0	7	4	3
P3	2	1	1	2	2	2	0	1	1	7	4	5
P4	0	0	2	4	3	3	4	3	1			



Process Flow:

P1 → P3 → P4 → P2 → P0

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2	5	3	2
P2	3	0	2	9	0	2	6	0	0	7	4	3
P3	2	1	1	2	2	2	0	1	1	7	4	5
P4	0	0	2	4	3	3	4	3	1	10	4	7



Process Flow:

P1 → P3 → P4 → P2 → P0

Pn	Allocation			Max			Needs			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2	5	3	2
P2	3	0	2	9	0	2	6	0	0	7	4	3
P3	2	1	1	2	2	2	0	1	1	7	4	5
P4	0	0	2	4	3	3	4	3	1	10	4	7
										10	5	7



Process Flow:

P1 → P3 → P4 → P2 → P0

Resources type A has 10 instances, resource B has 5 and resource C has 7.

Recovery Algorithms

- Terminate every job that's active in system and restart them from beginning.
- Terminate only the jobs involved in deadlock and ask their users to resubmit them.
- Terminate jobs involved in deadlock one at a time, checking to see if deadlock is eliminated after each removal, until it has been resolved.



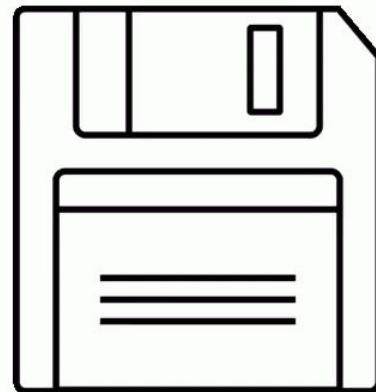
Recovery Algorithms

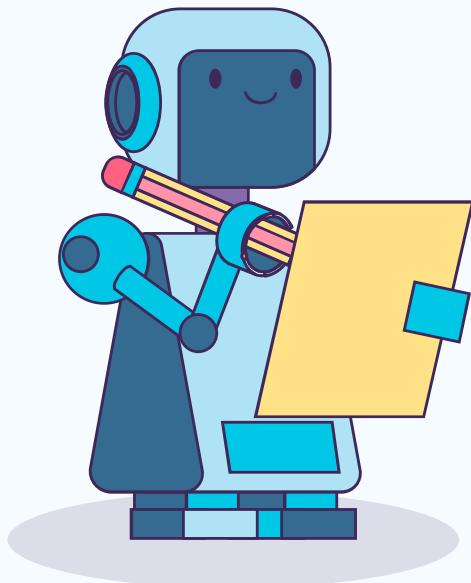
- Have **job keep record** (snapshot) of its progress so it can be interrupted and then continued without starting again from the beginning of its execution.
- Select a **non-deadlocked** job, preempt resources it's holding, and allocate them to a deadlocked process so it can resume execution, thus breaking the deadlock
- Stop new jobs from entering system, which allows non-deadlocked jobs to run to completion so they'll release their resources (no victim).



Starvation

- Result of conservative allocation of resources where a single job is prevented from execution because it's kept waiting for resources that never become available.
- Avoid starvation via algorithm designed to detect starving jobs which tracks how long each job has been waiting for resources (**aging**).





Thank you

CREDITS: This presentation template was created by [Slidesgo](#), and includes icons by [Flaticon](#), and infographics & images by [Freepik](#)