BMCS3003 Distributed Systems and Parallel Processing

L05 - Concurrency Control (Part 2)

Presented by

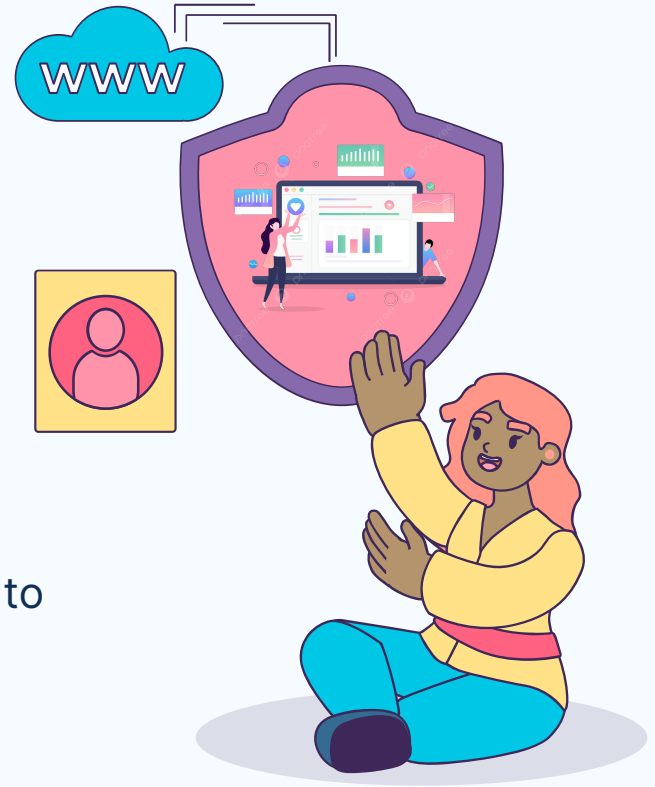
Assoc Prof Ts Dr Tew Yiqi
May 2023

Table of contents

01

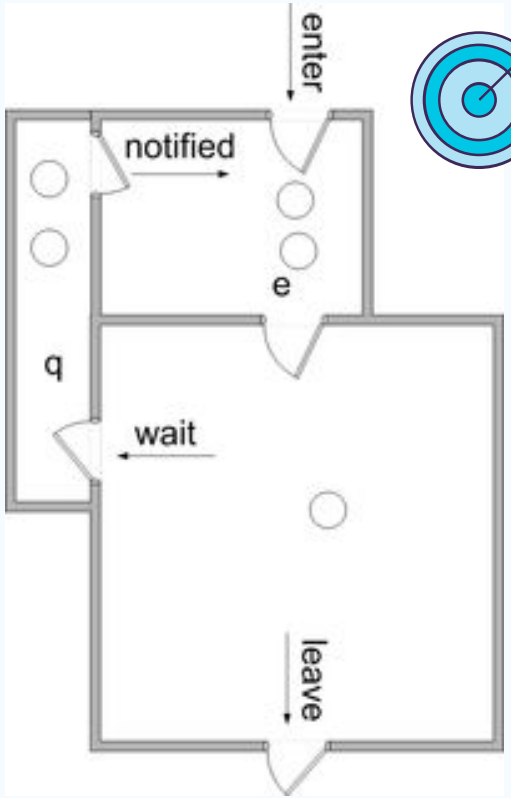
Monitors

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



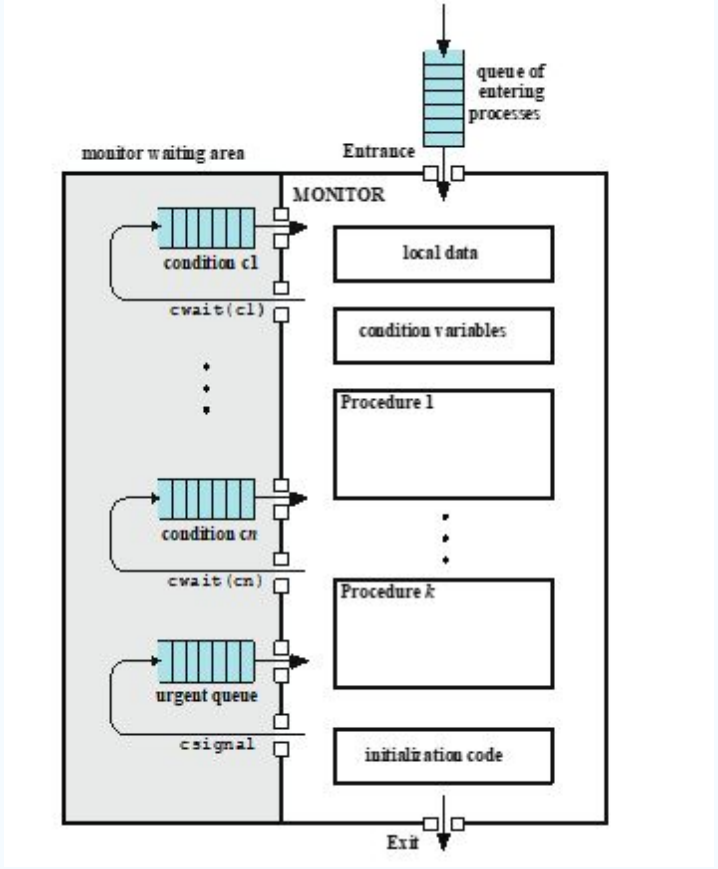
Monitor Characteristics

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

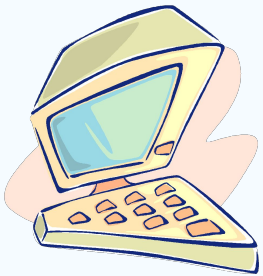
Only one process may be executing in the monitor at a time

Example of Monitor



Synchronisation

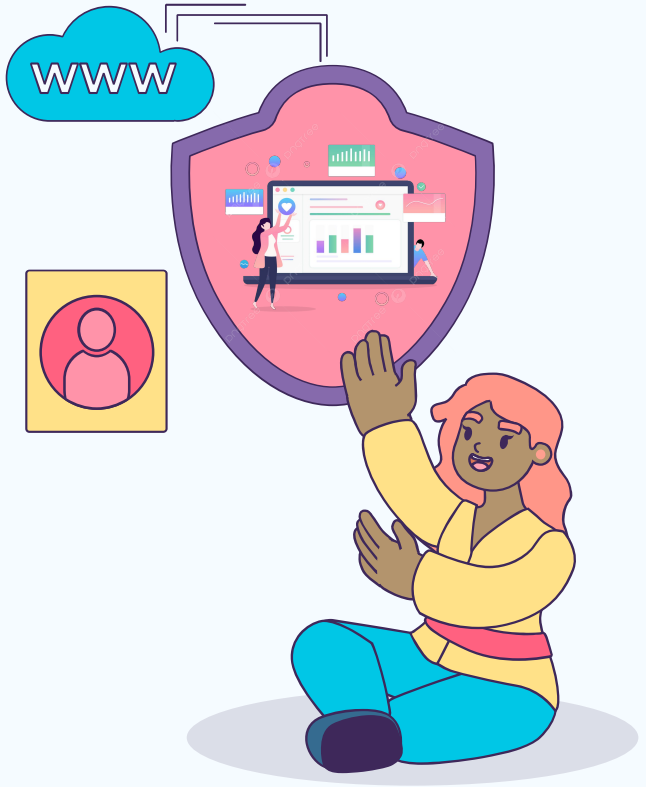
- Achieved by the use of **condition variables** that are contained within the monitor and accessible only within the monitor
  - Condition variables are operated on by two functions:
    - cwait(c): suspend execution of the calling process on condition c
    - csignal(c): resume execution of some process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



02

Token Passing

An algorithm for Mutual Exclusion



Token Passing

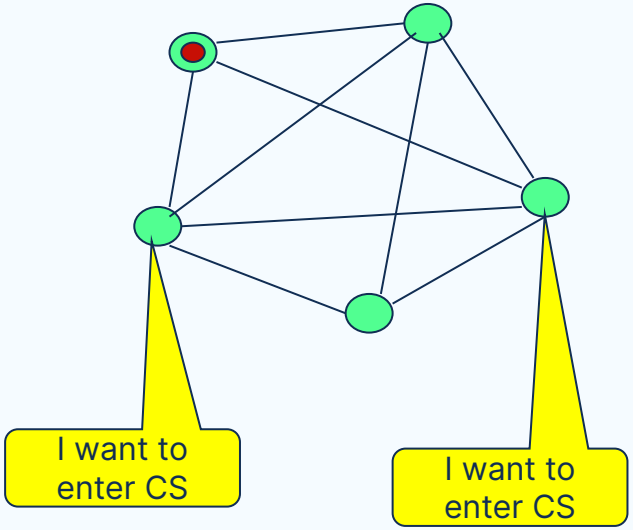
Algorithm for Mutual Exclusion

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.

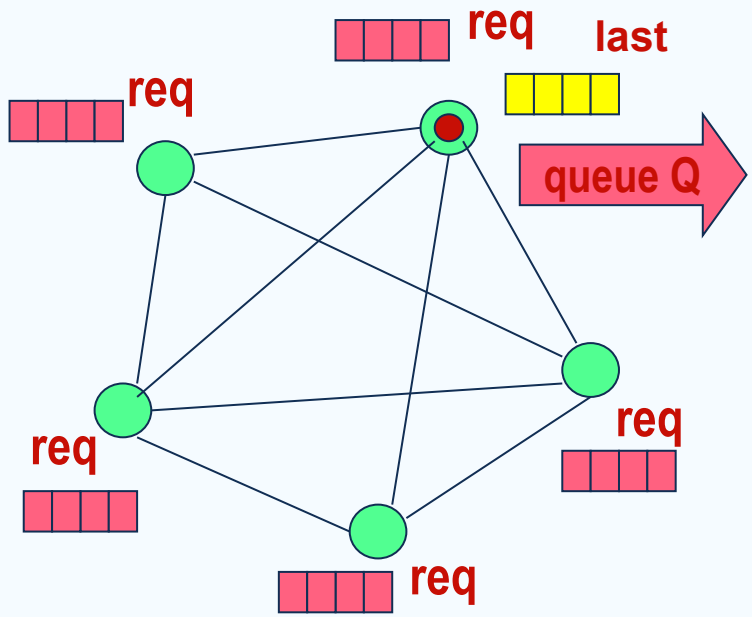
Suzuki-Kasami algorithm [Paper]

Process i broadcasts **(i, num)**

Sequence number of the request

Each process maintains
-an array **req**: **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**: **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 **req[k] to max(req[k], num).**

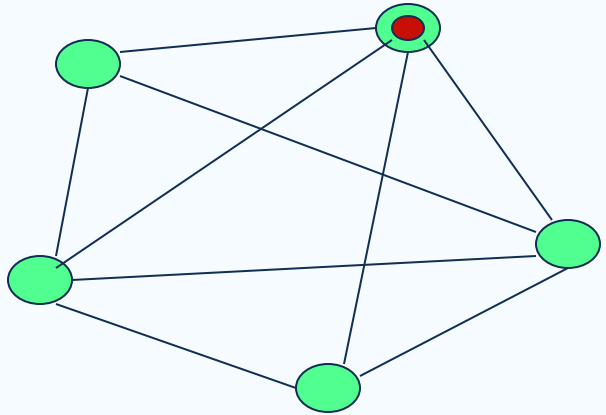
The holder of the token

--Completes its CS
--Sets **last[i]:=** its own **num**
--Updates **Q** by retaining each process **k** only if **1+ last[k] = 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 ≡ (Q, last)**



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

Suzuki-Kasami algorithm [Paper]

{Program of process j}

Initially,
∀i: req[i] = last[i] = 0

*** Entry protocol ***

req[j] := req[j] + 1
Send (j, req[j]) to all
Wait until token (Q, last) arrives

Critical Section

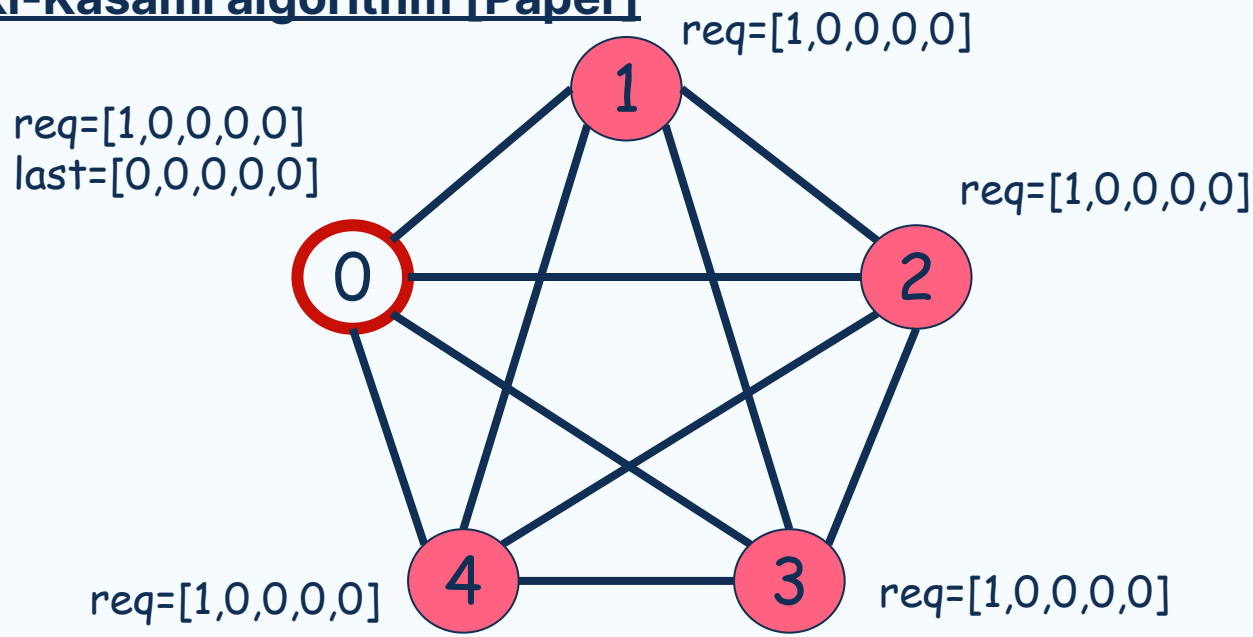
*** Exit protocol ***

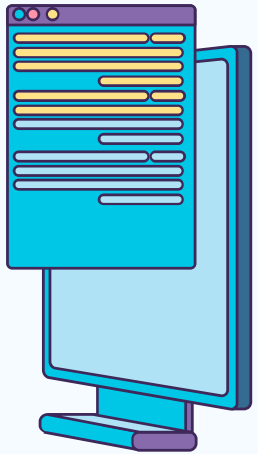
last[j] := req[j]
∀k ≠ j: k ∉ Q ∧ req[k]
= last[k] + 1 -> append k to Q;
if Q is not empty -> send
(tail-of-Q, last) to head-of-Q **fi**

*** Upon receiving a request (k, num) ***

req[k] := max(req[k], num)

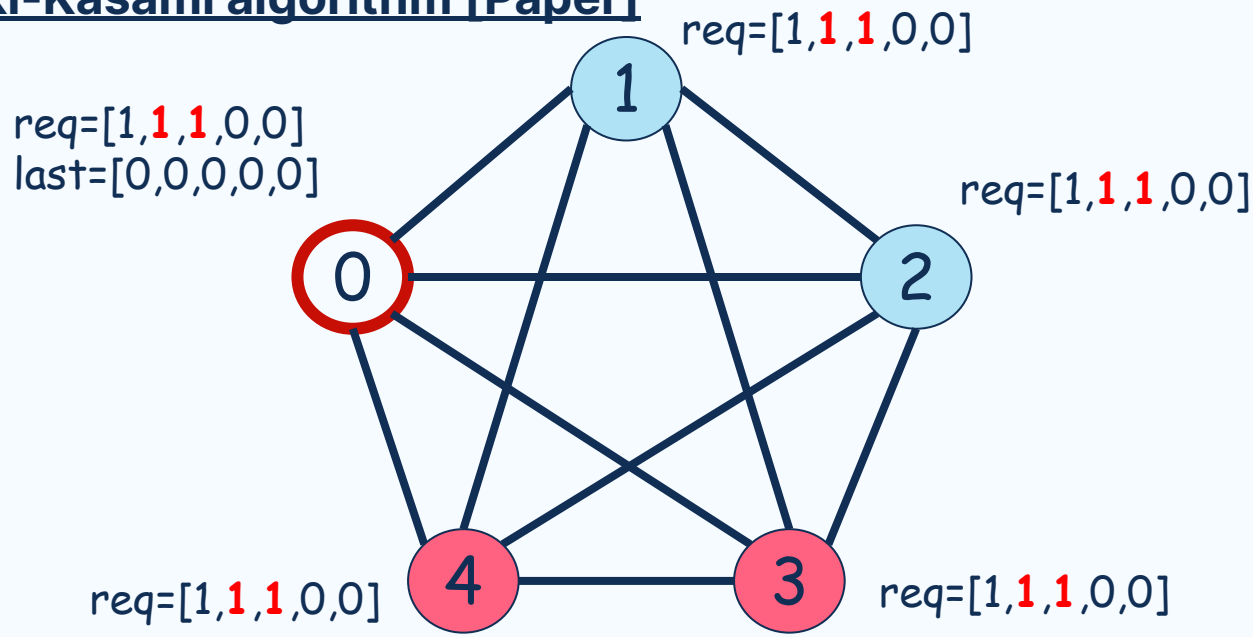
Suzuki-Kasami algorithm [Paper]

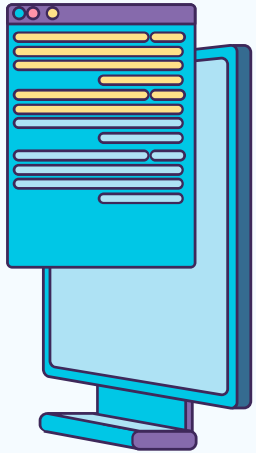




In the initial state:
process 0 has sent a request to all,
and grabbed the token.

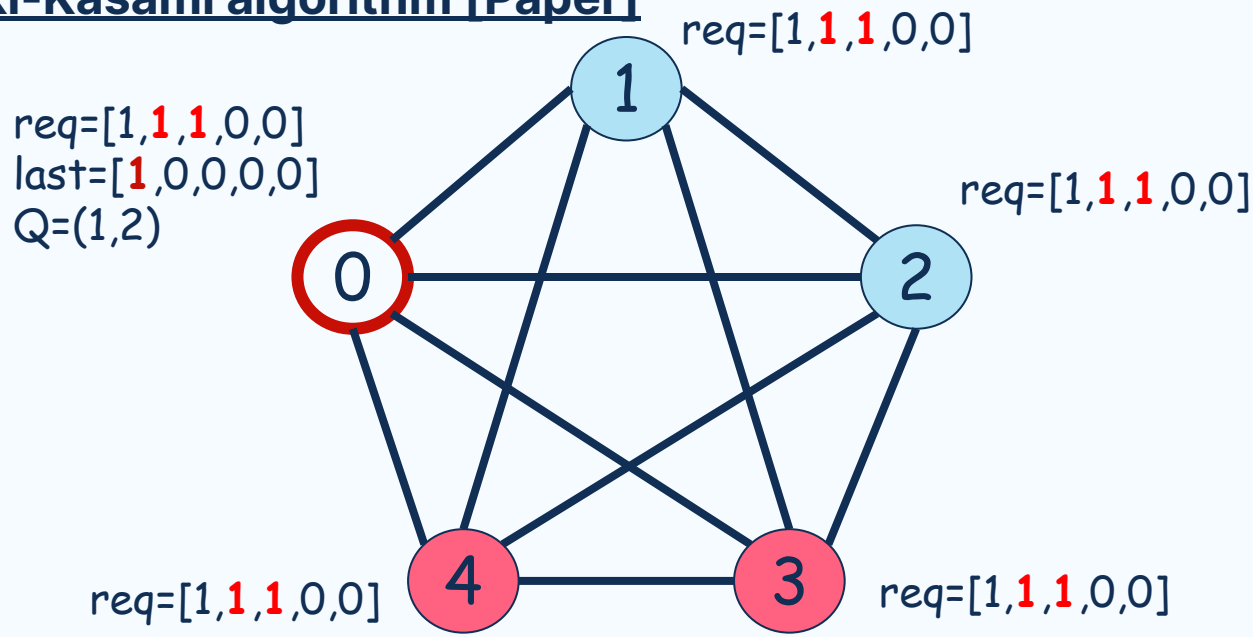
Suzuki-Kasami algorithm [Paper]

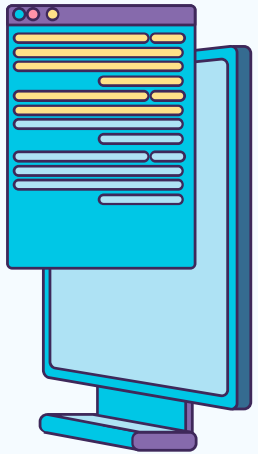




1 & 2 send the request to enter Critical Section

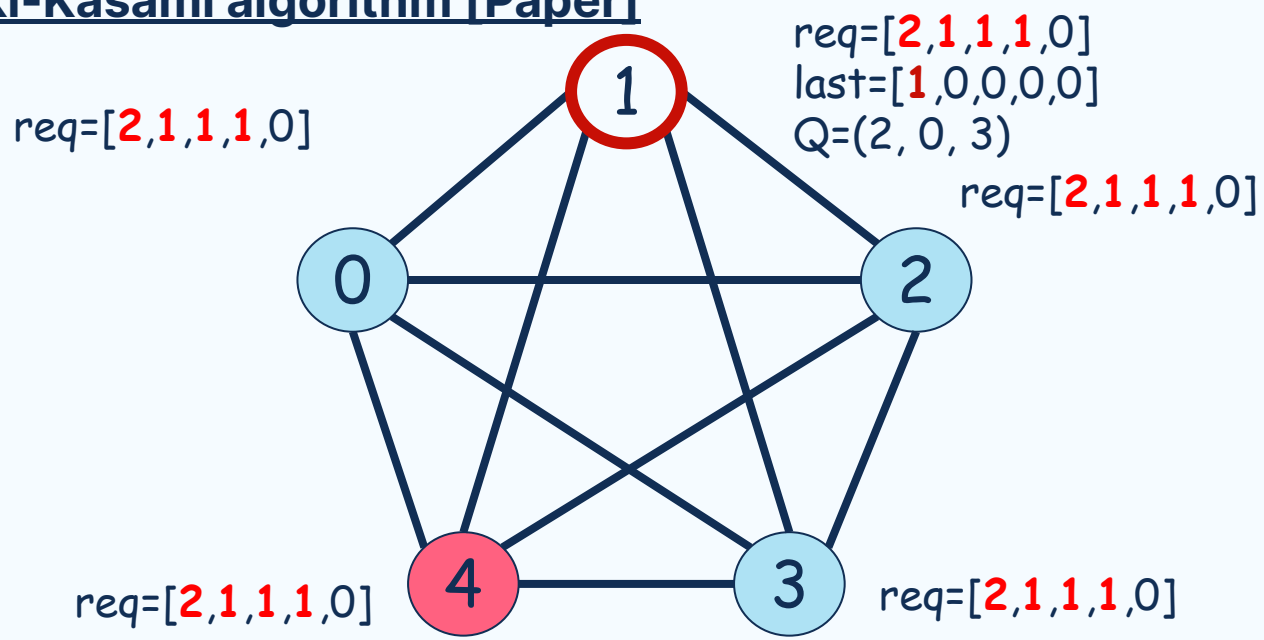
Suzuki-Kasami algorithm [Paper]

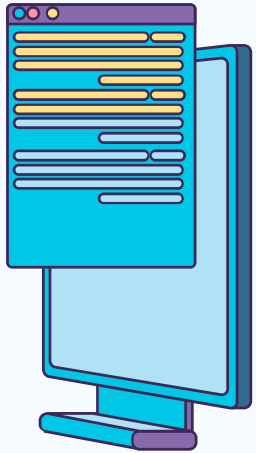




0 prepares to quit Critical Section

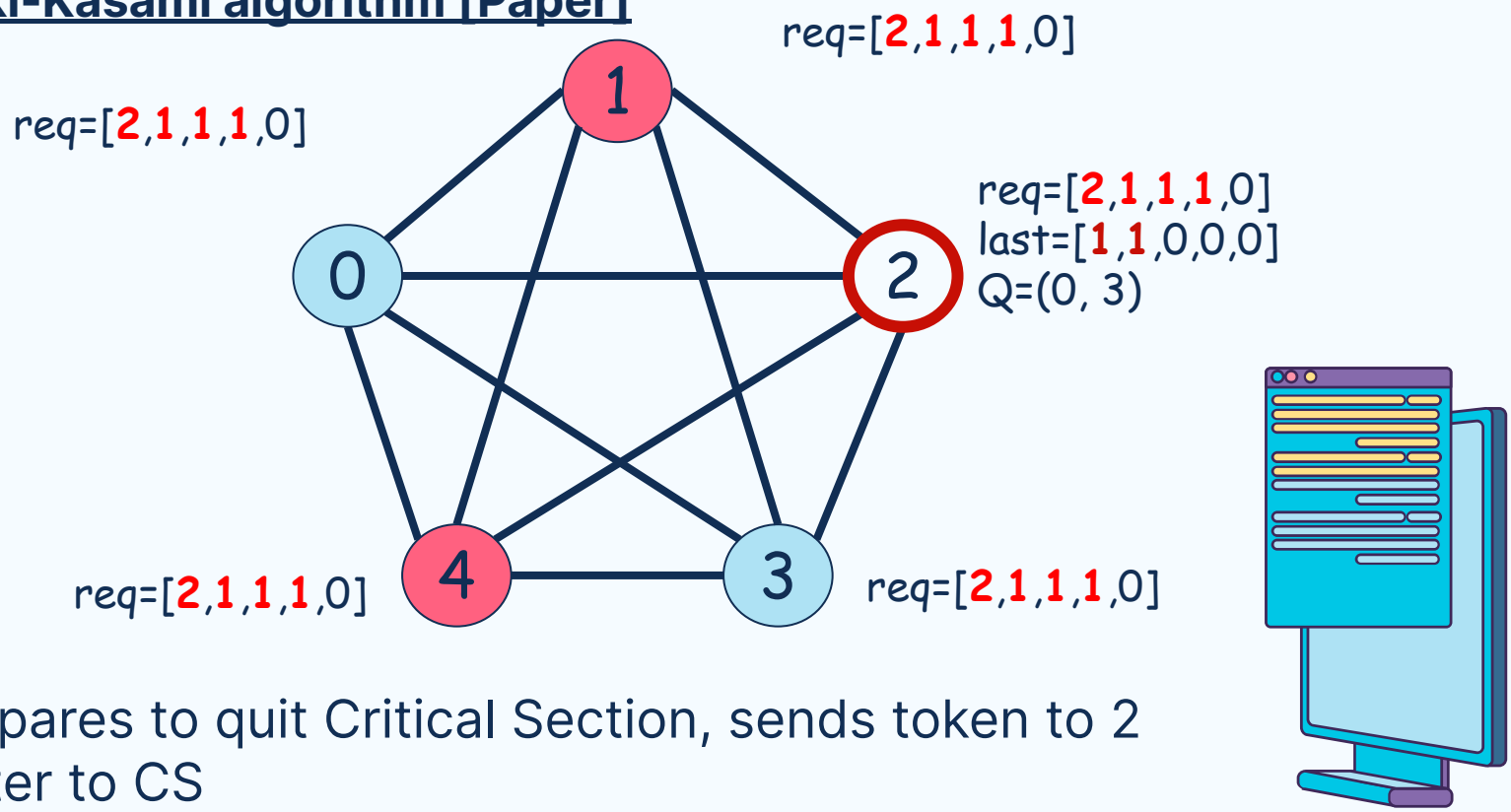
Suzuki-Kasami algorithm [Paper]





1 enters CS
0 and 3 send requests

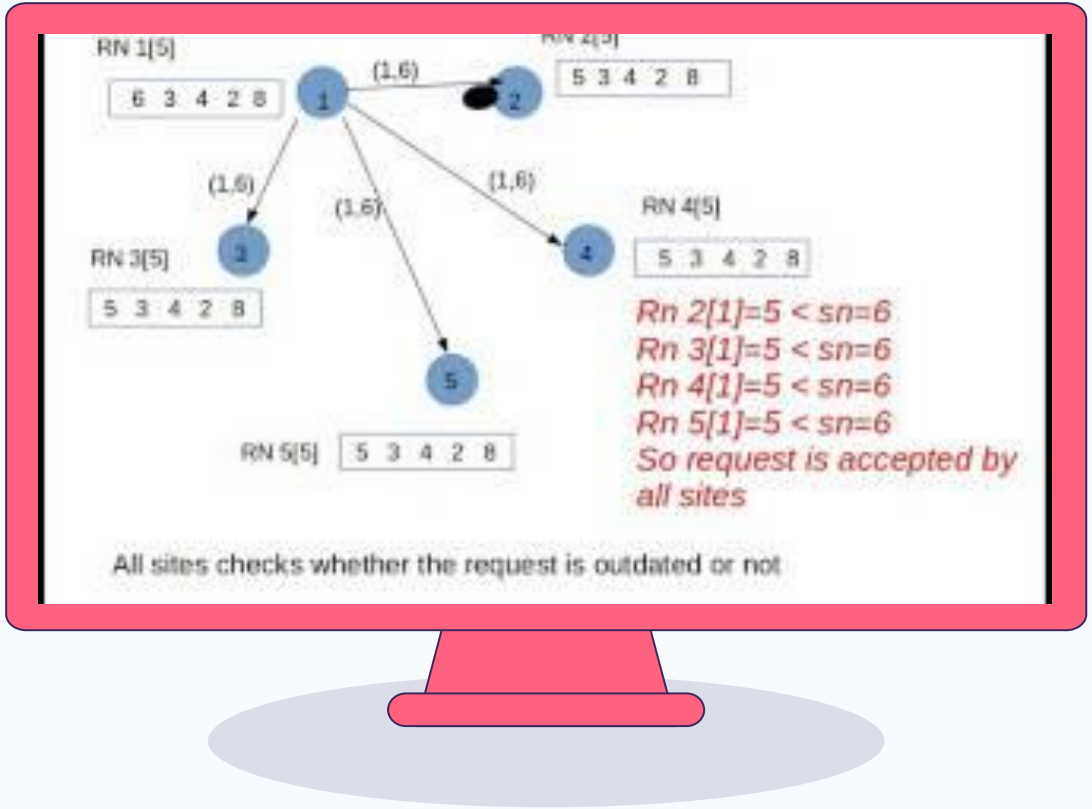
Suzuki-Kasami algorithm [Paper]



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

Suzuki-Kasami Algorithm

by
Shivani Srivarshini



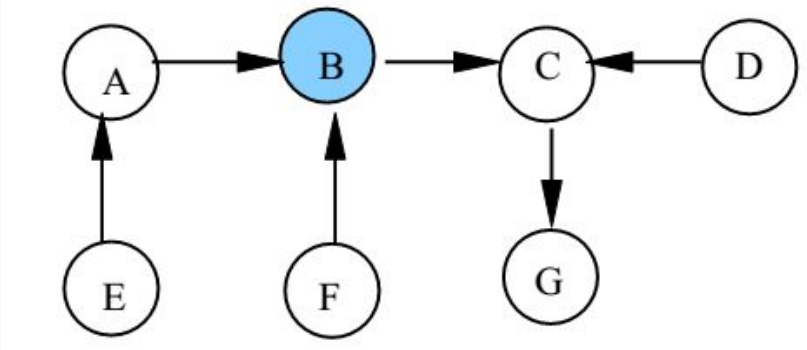
Token Passing

Algorithm for Mutual Exclusion

Raymond's Tree algorithm [Paper]

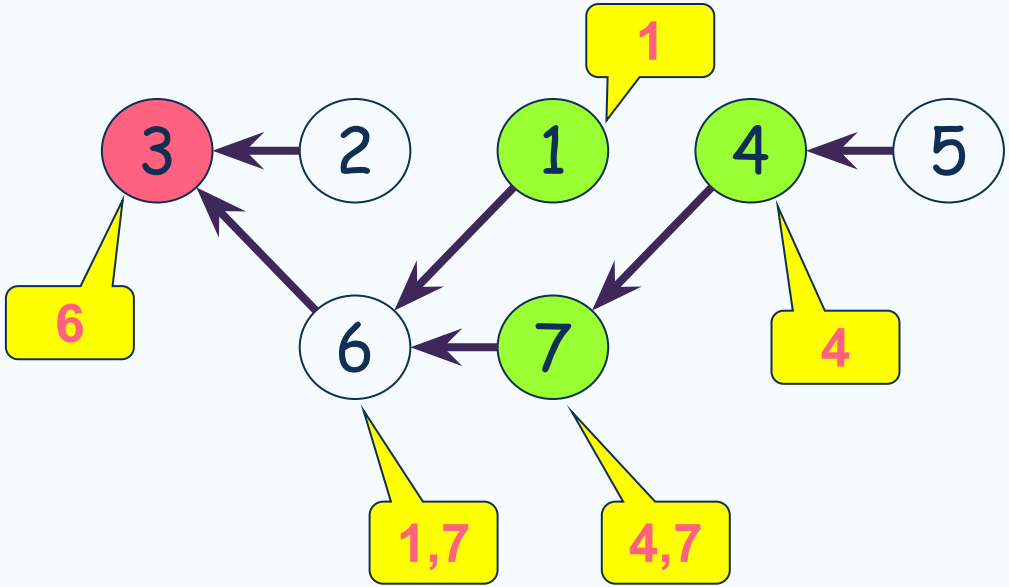
Kerry Raymond

Resources: Geeksforgeeks

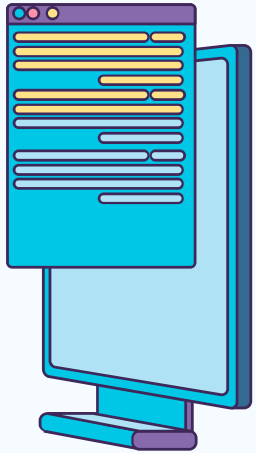


- **Raymond's tree based algorithm** is lock based algorithm for mutual exclusion in a distributed system in which a site is allowed to enter the critical section if it has the token.
- All sites are arranged as a directed tree such that the edges of the tree are assigned direction towards the site that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

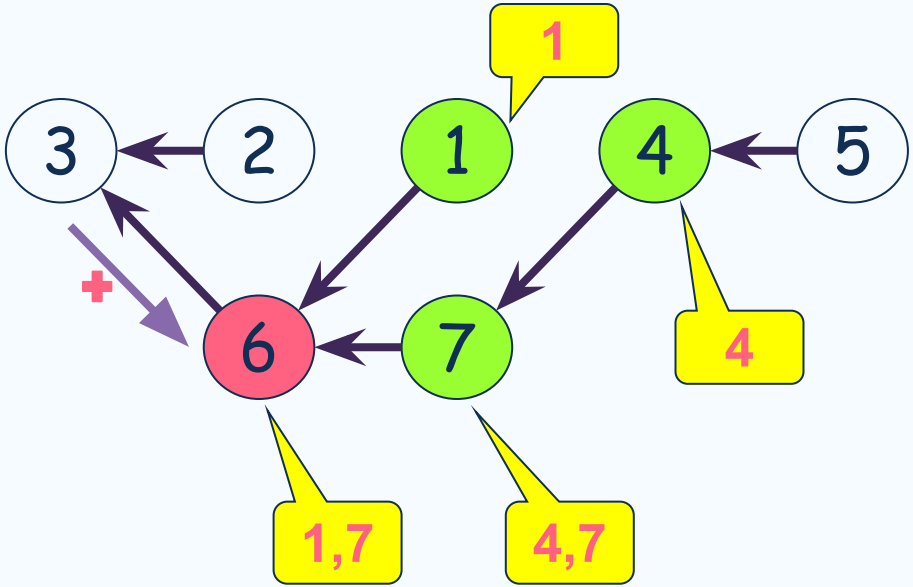
Raymond's Tree algorithm [Paper]



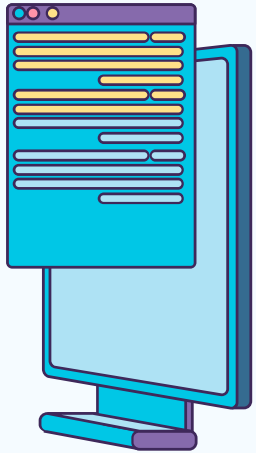
1, 4, 7 want to enter their CS



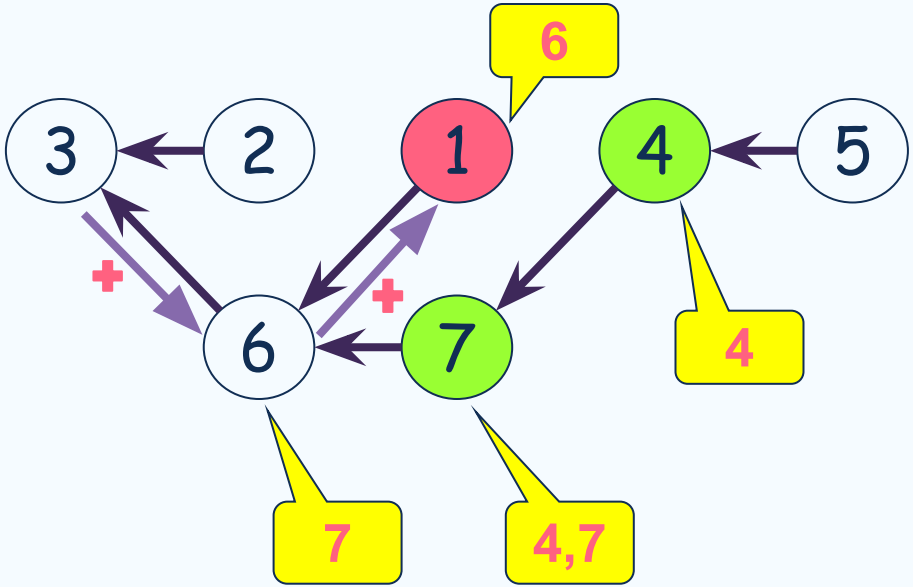
Raymond's Tree algorithm [Paper]



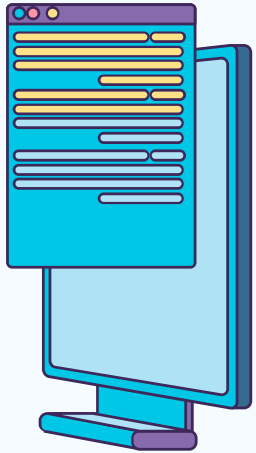
3 sends the token to 6



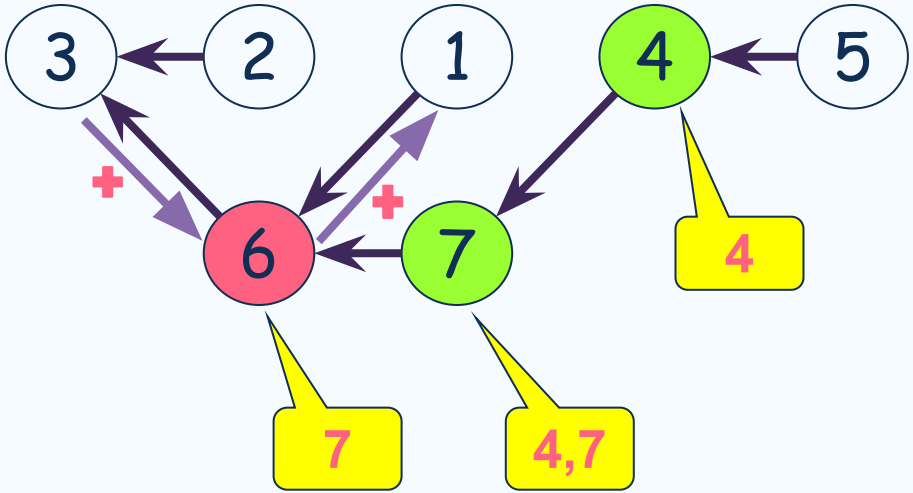
Raymond's Tree algorithm [Paper]



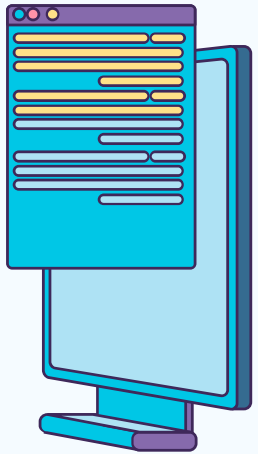
6 forwards the token to 1



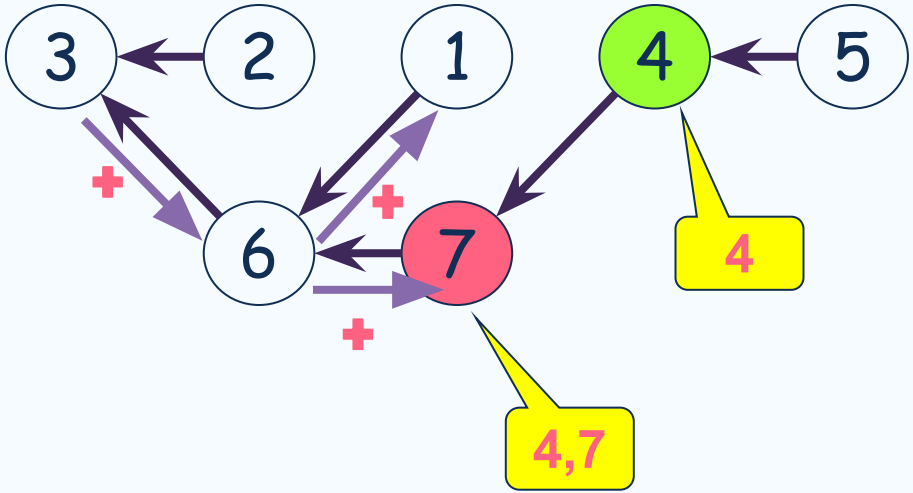
Raymond's Tree algorithm [Paper]



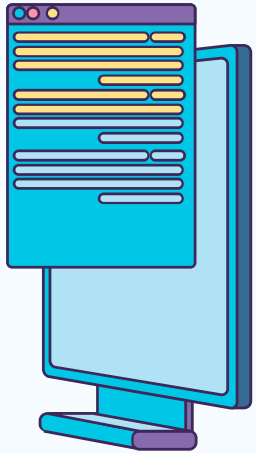
6 requests the token from 1



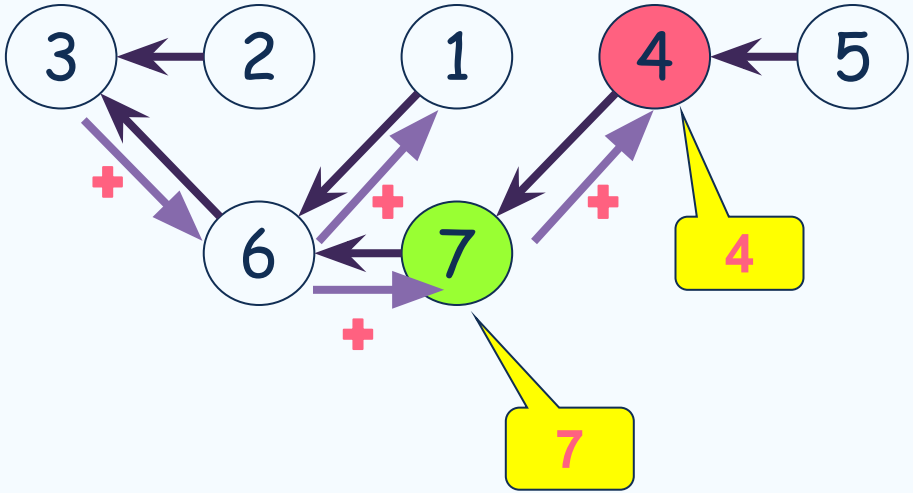
Raymond's Tree algorithm [Paper]



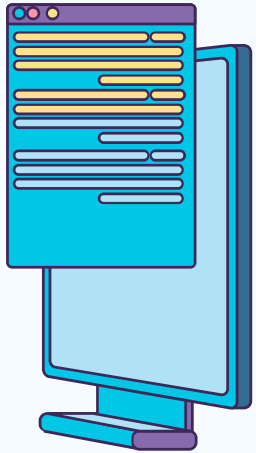
7 requests the token from 6



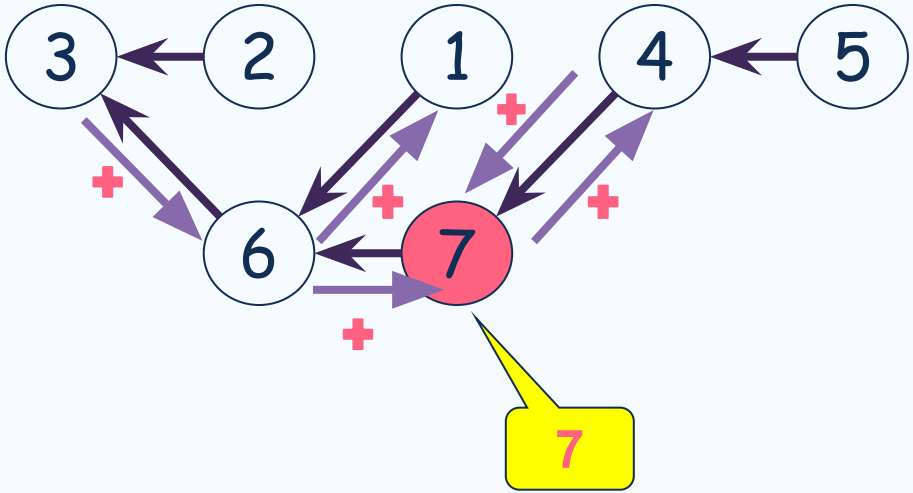
Raymond's Tree algorithm [Paper]

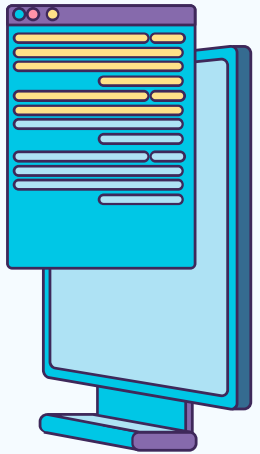


4 requests the token from 7



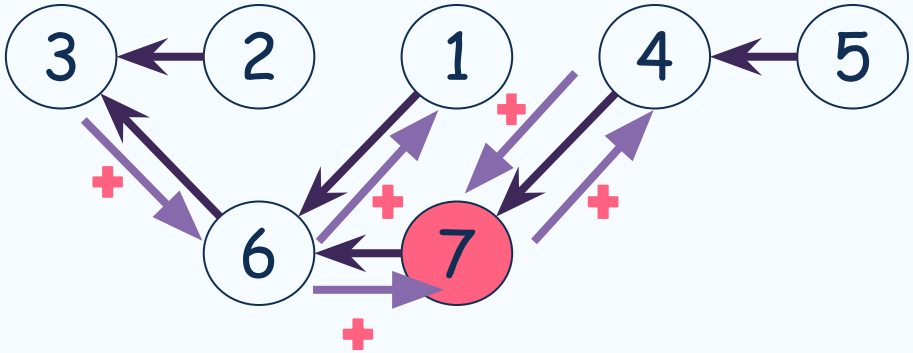
Raymond's Tree algorithm [Paper]



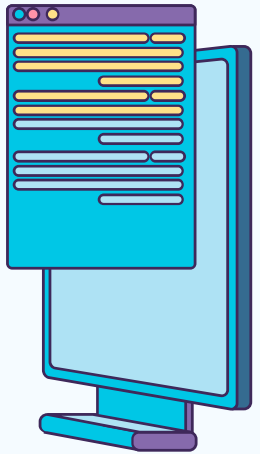


7 requests the token from 4

Raymond's Tree algorithm [Paper]



Done

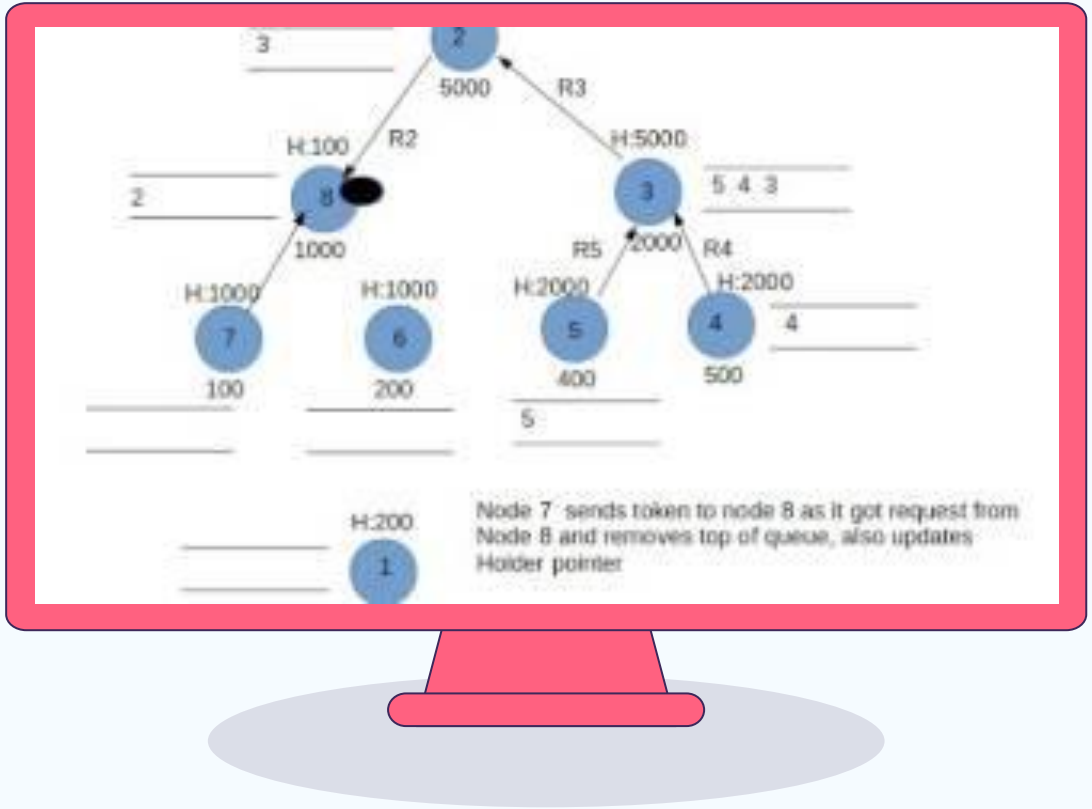


Raymond's Tree algorithm [Paper]

- The message complexity is O(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

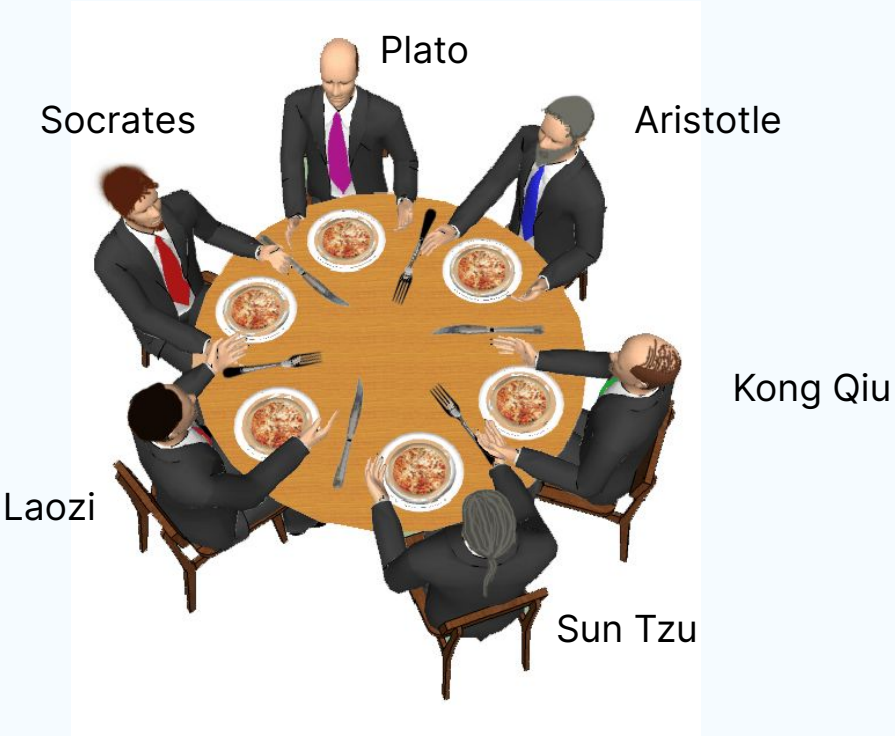
Deadlock

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

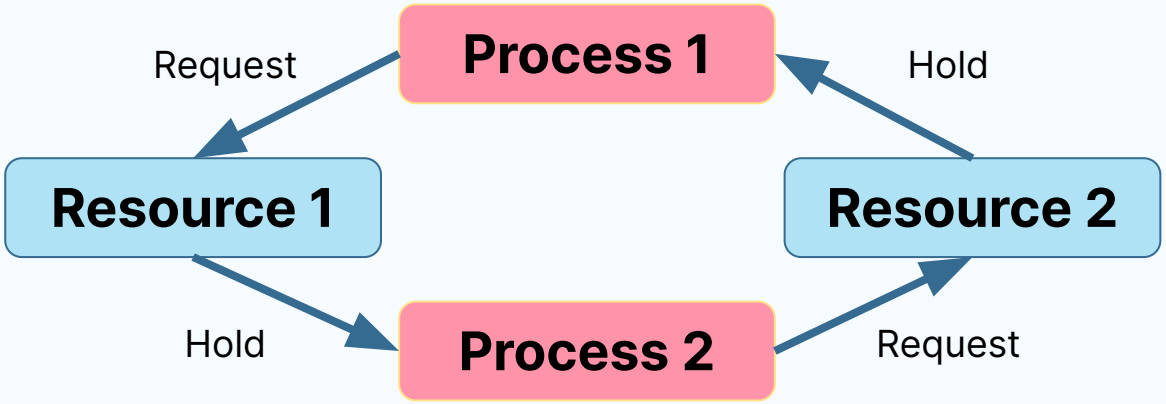
Waiter !

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

Resource: Wikipedia

Deadlock Problems and Conditions

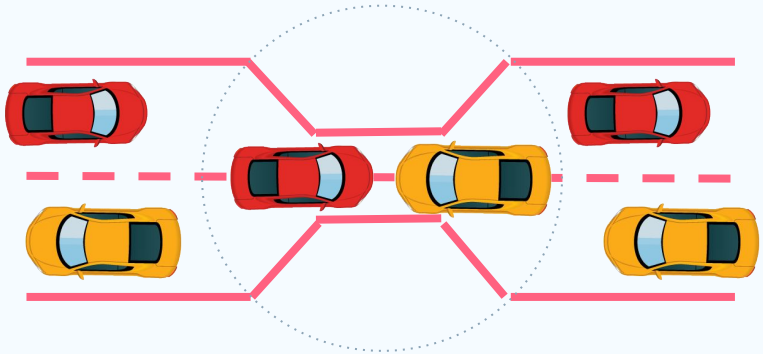
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.



Bridge Crossing

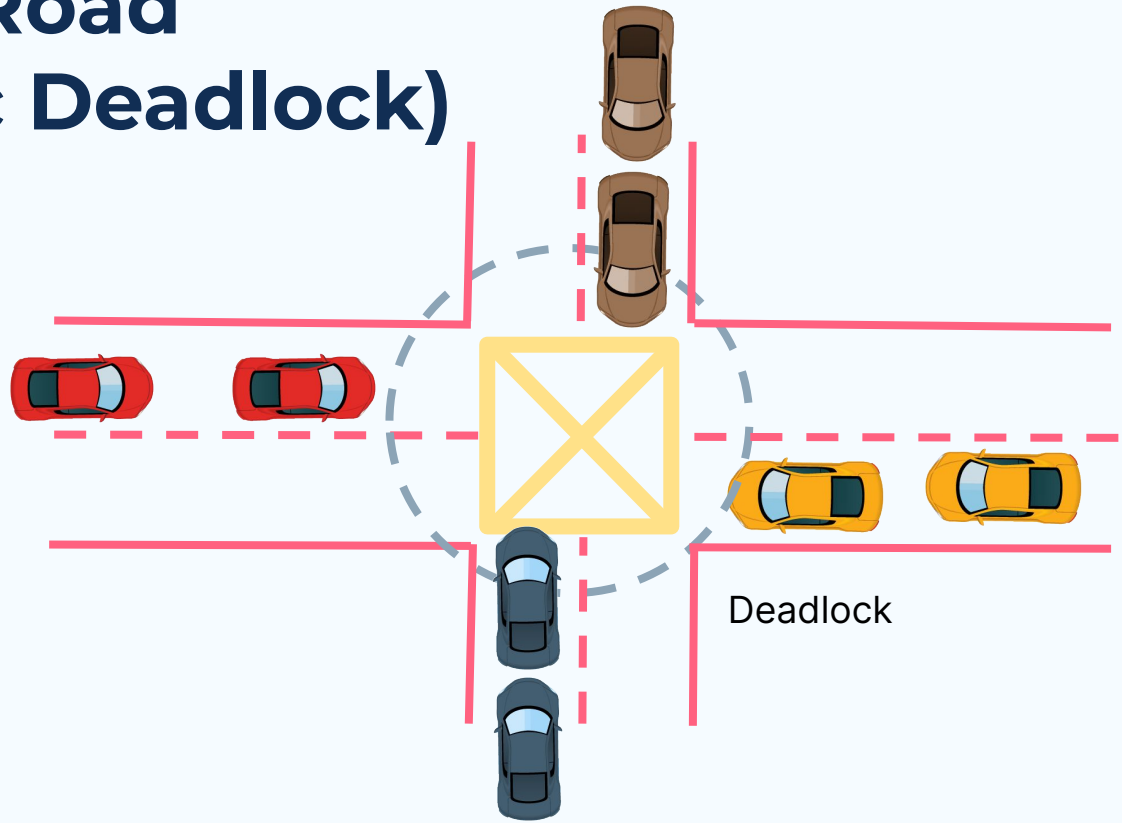
- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

One-way Bridge Crossing

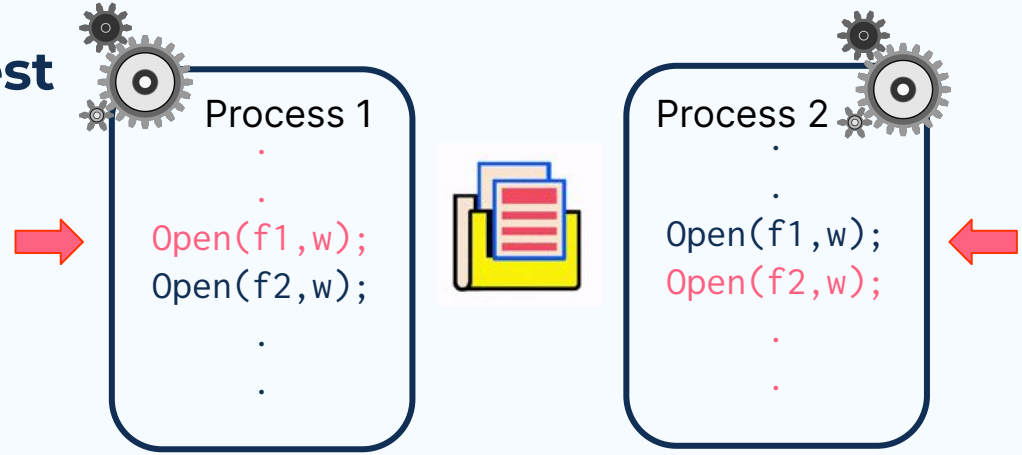


Deadlock

Cross Road (Traffic Deadlock)



Case 1 :
Deadlocks on File Request

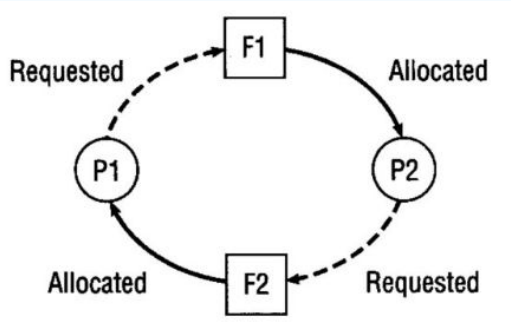


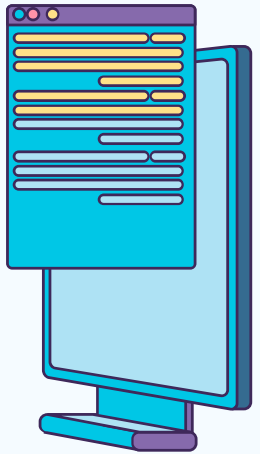
Each holding one of the files open while trying to open other. Since this will never be close, the processes will wait.

Case 1 :
Deadlocks on File Request

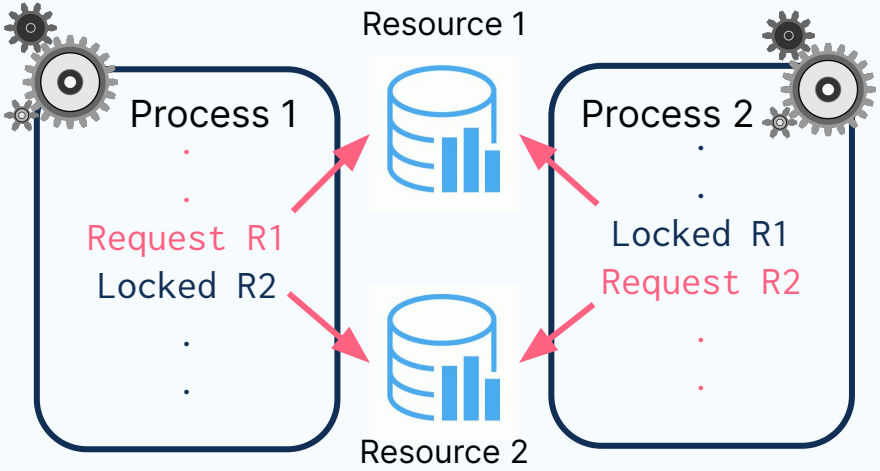
Occurs if jobs are allowed to request and hold files for the duration of their execution:

- 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 ois 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:

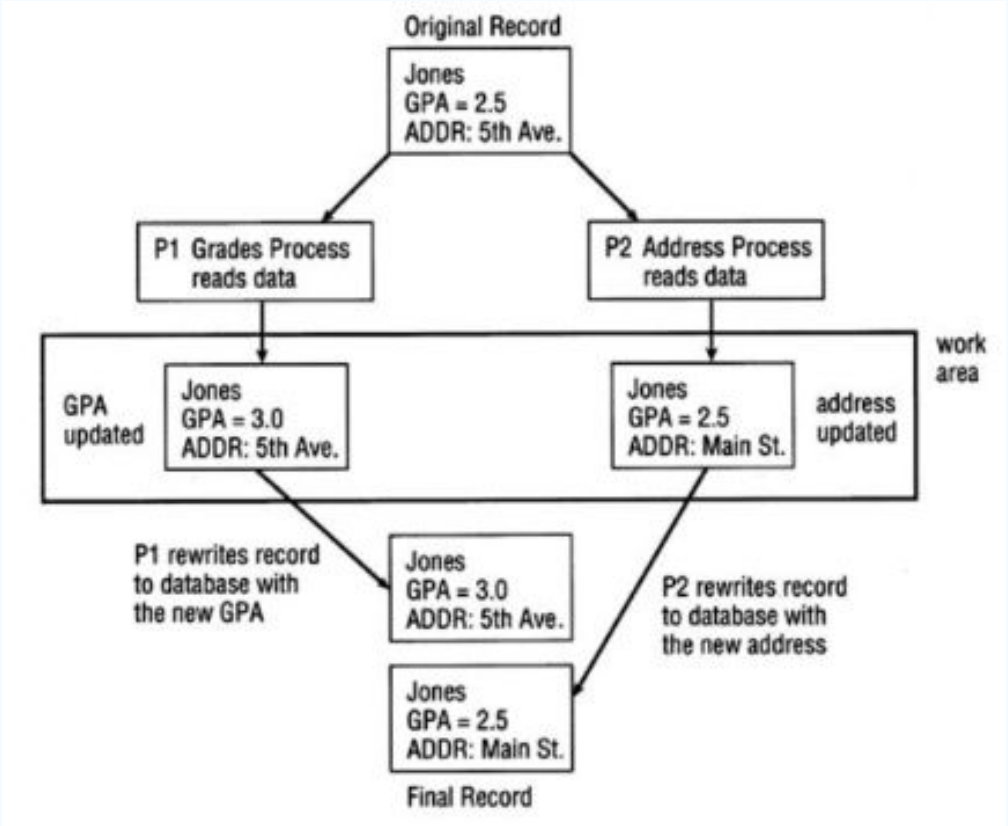
- P1 accesses R1 and locks it.
- P2 accesses R2 and locks it.
- P1 requests R2, which is locked by P2.
- P2 requests R1, which is locked by P1.

Case 2 :
Deadlocks in Databases

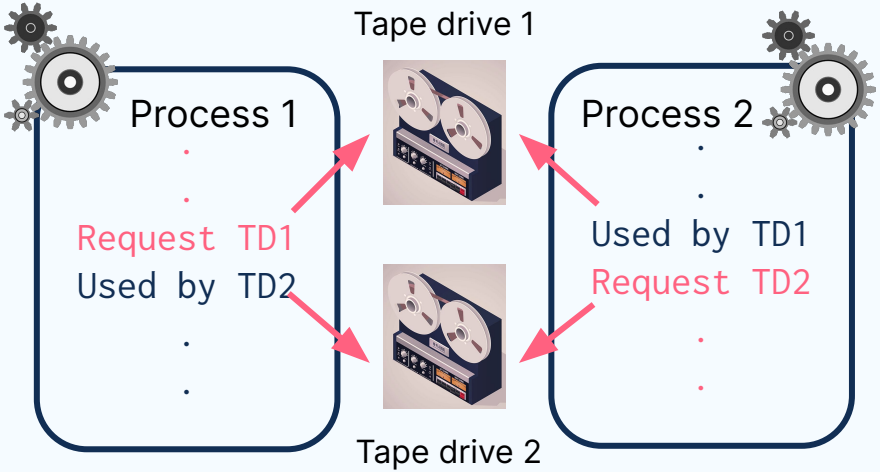
Locking:
A technique through which the user locks out all other user while working with the database.

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.



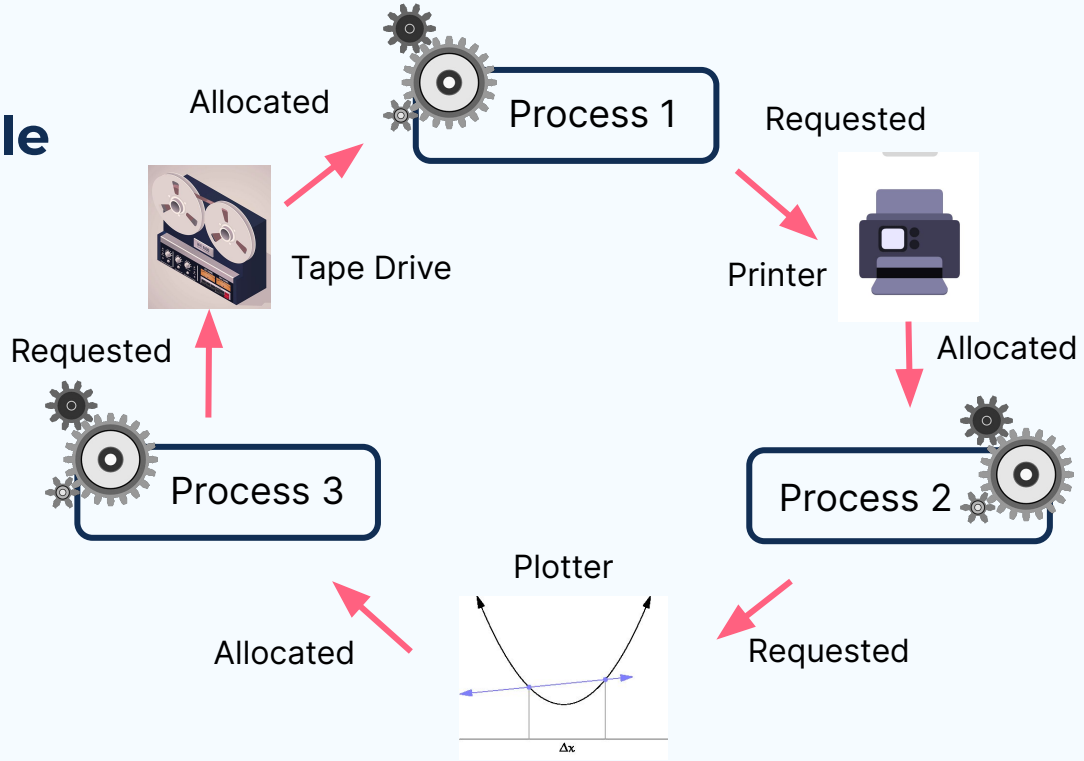
Case 3 : Deadlocks in Dedicated Device Allocation



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



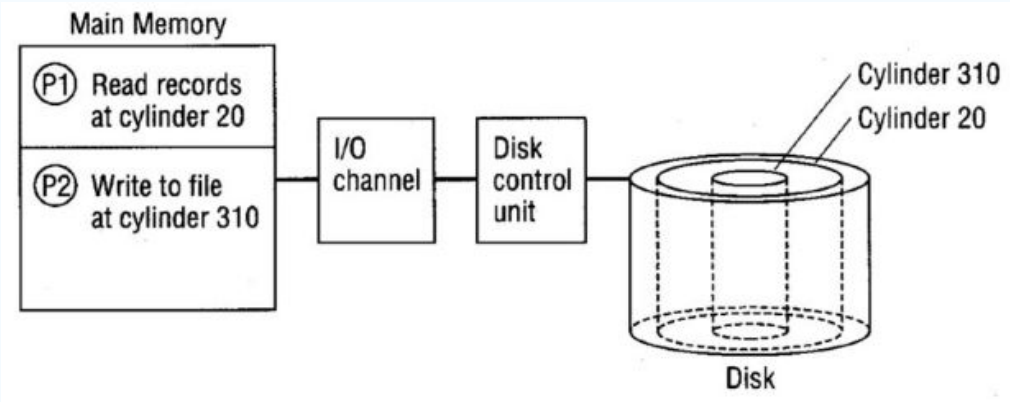
Case 5 :
Deadlocks in Spooling

- **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.

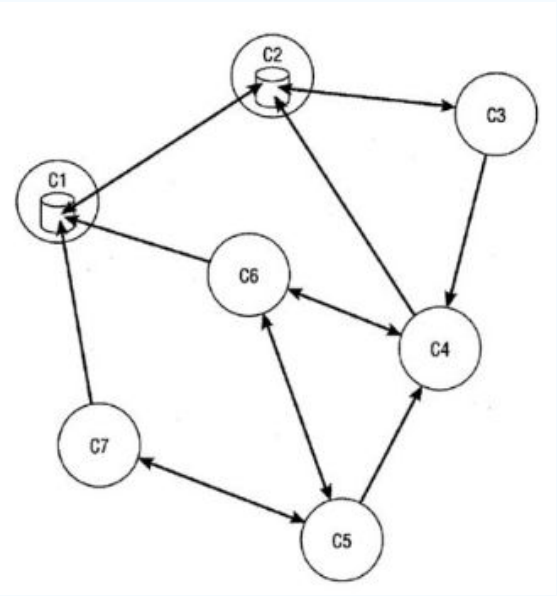
- Two processes are each waiting for an I/O request to be filled: Cylinder 20, 310.
- Neither can be satisfied because the device puts each request on hold when it tried to filfill the other processes.
- Without controls to regulate use of disk drive, competing processes could send conflicting commands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

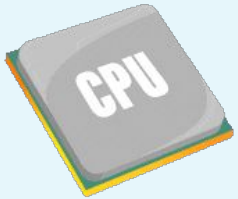


Understanding

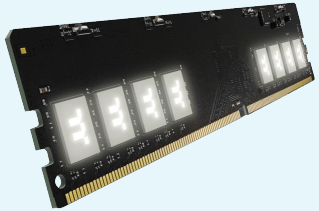
Physical

Tape Drive

Printer



CPU



Memory

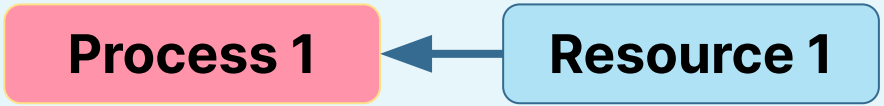
Logical

Files

Four Condition for Deadlocks

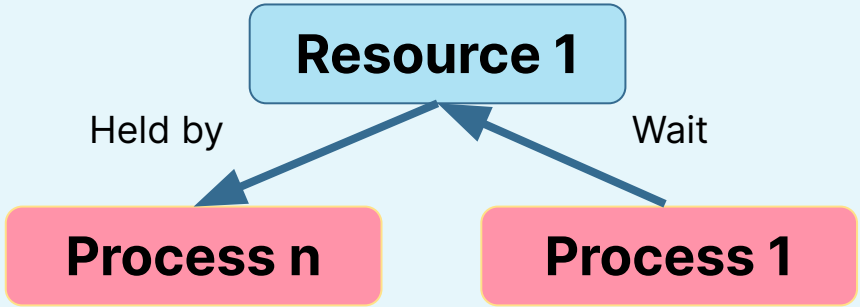
1) Mutual Exclusion

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

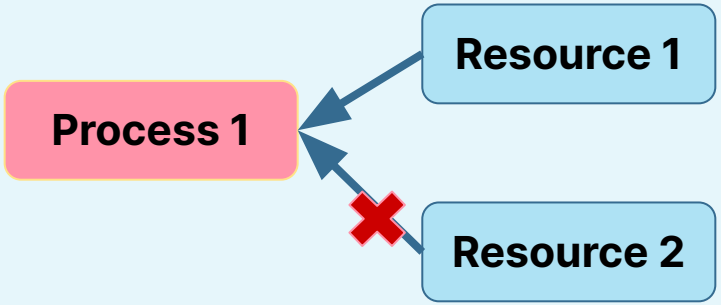
A process holds a resource & waits to acquire another resource that is used by another process.



Four Condition for Deadlocks

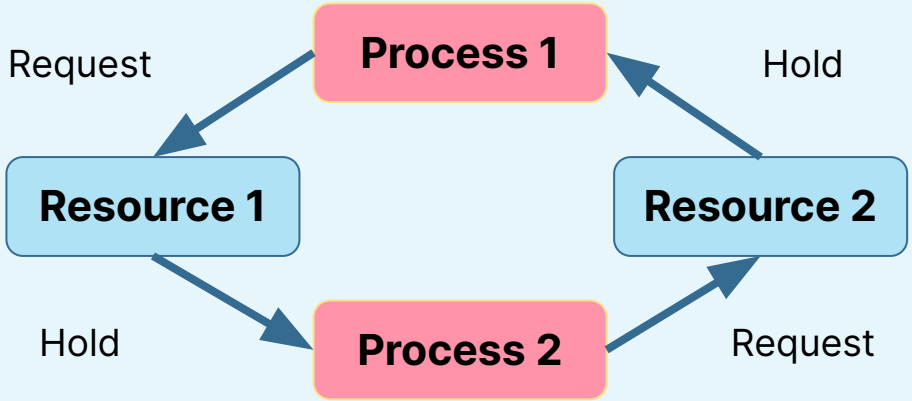
3) No Preemption

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

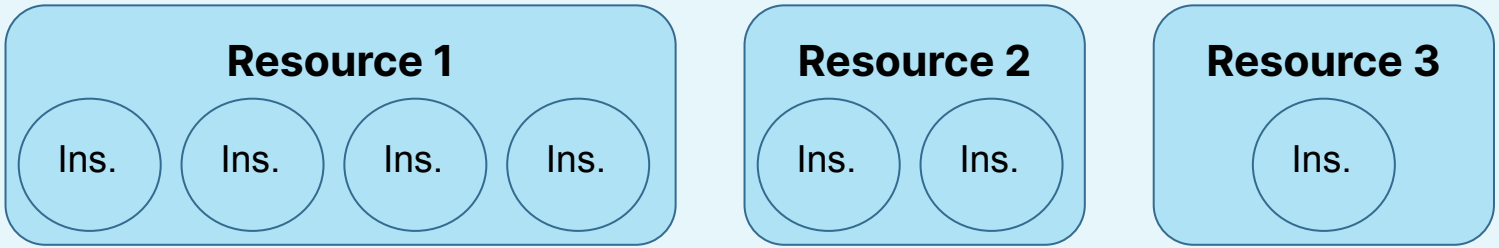
Holding

P_i is holding R_j

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

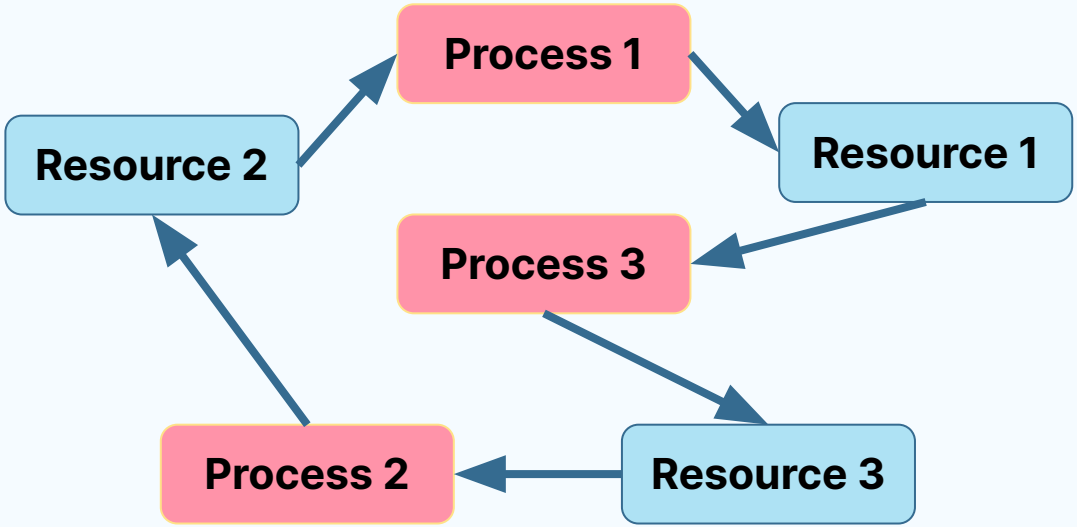
- Resource types $R_1, R_2, \ldots, R_m$
- *eg : CPU cycles, memory space, I/O devices*

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

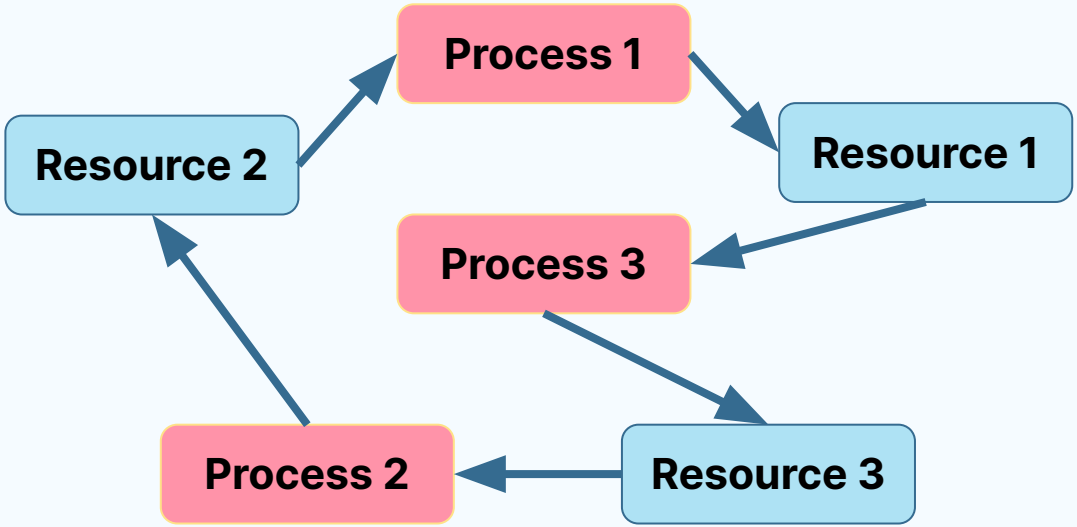
Is this a Deadlock ?



Resource-Allocation Graph 1

Check your understanding

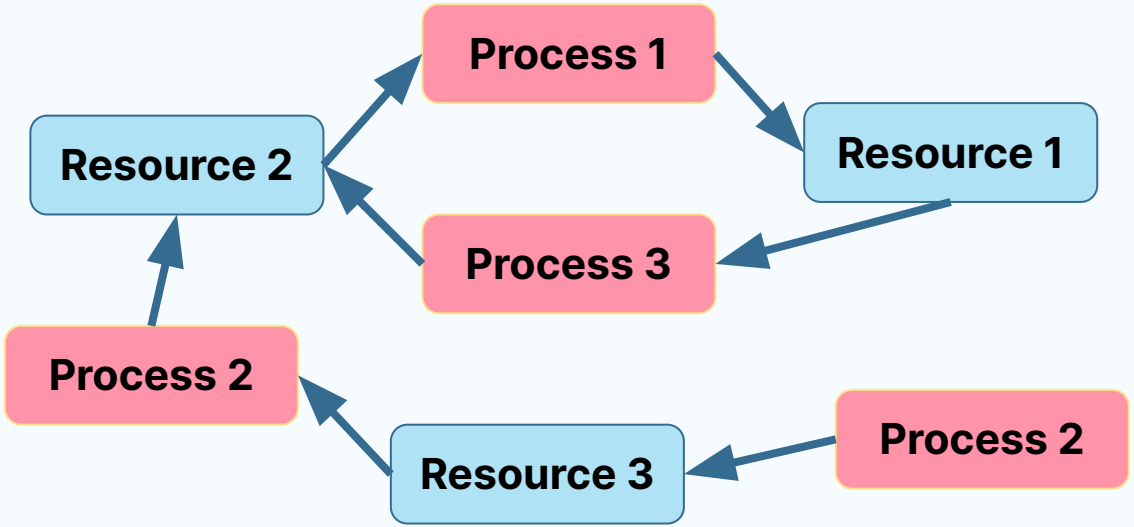
Is this a Deadlock ?



Resource-Allocation Graph 1

Check your understanding

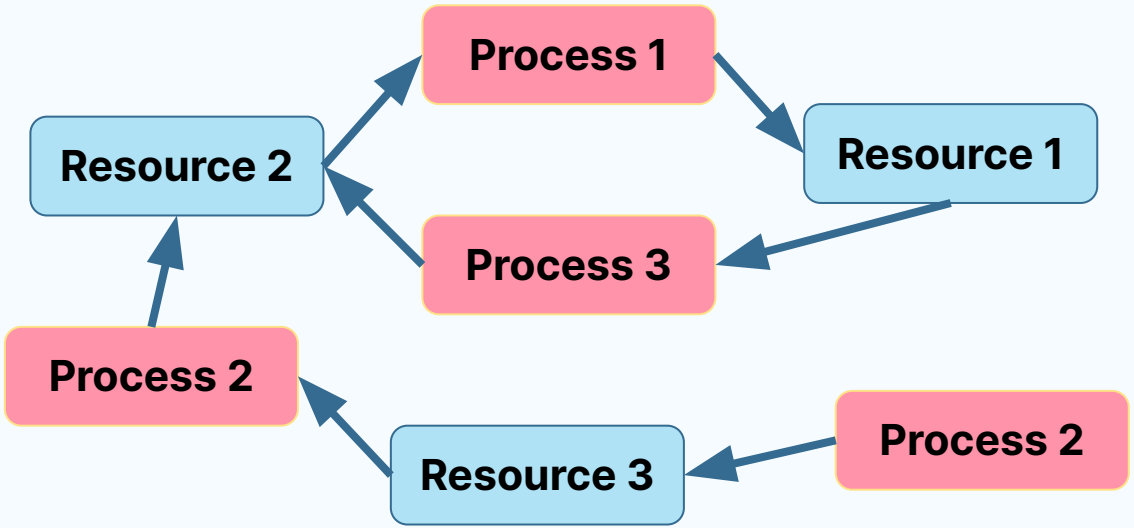
Is this a Deadlock ?



Resource-Allocation Graph 2

Check your understanding

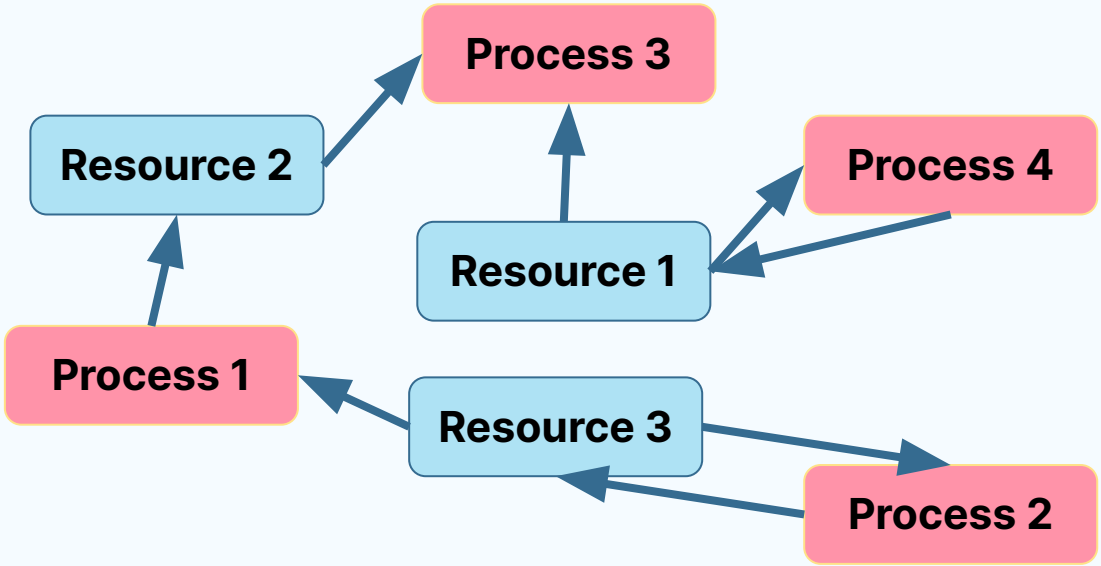
Is this a Deadlock ?



Resource-Allocation Graph 2

Check your understanding

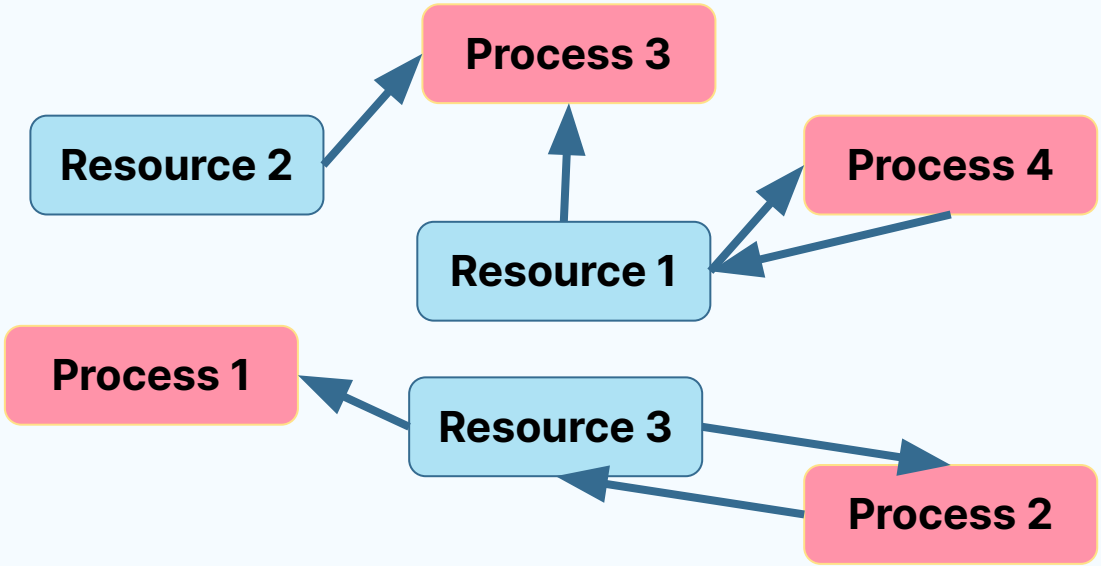
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

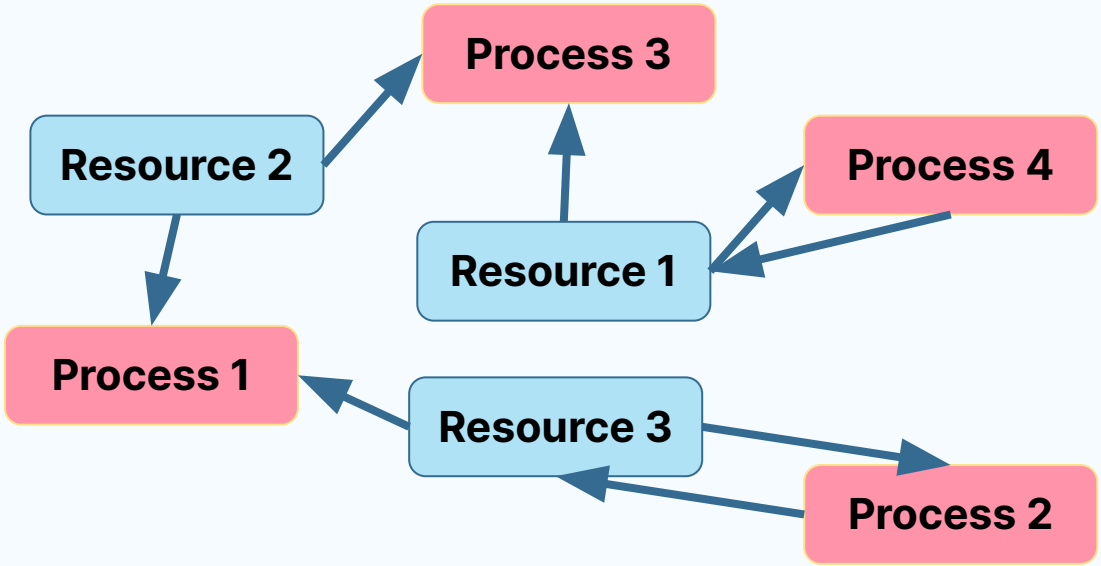
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

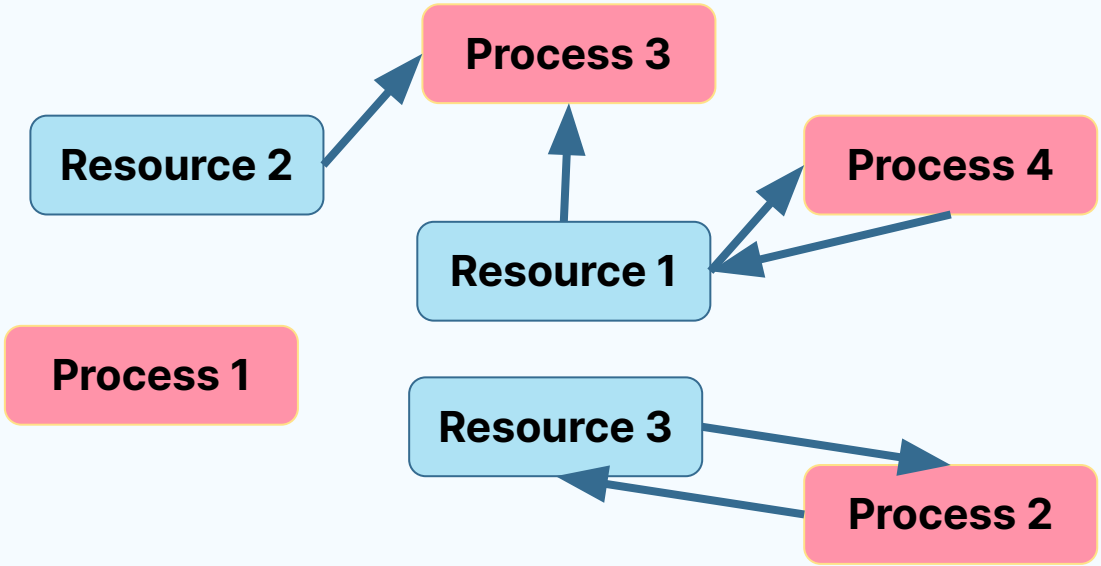
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

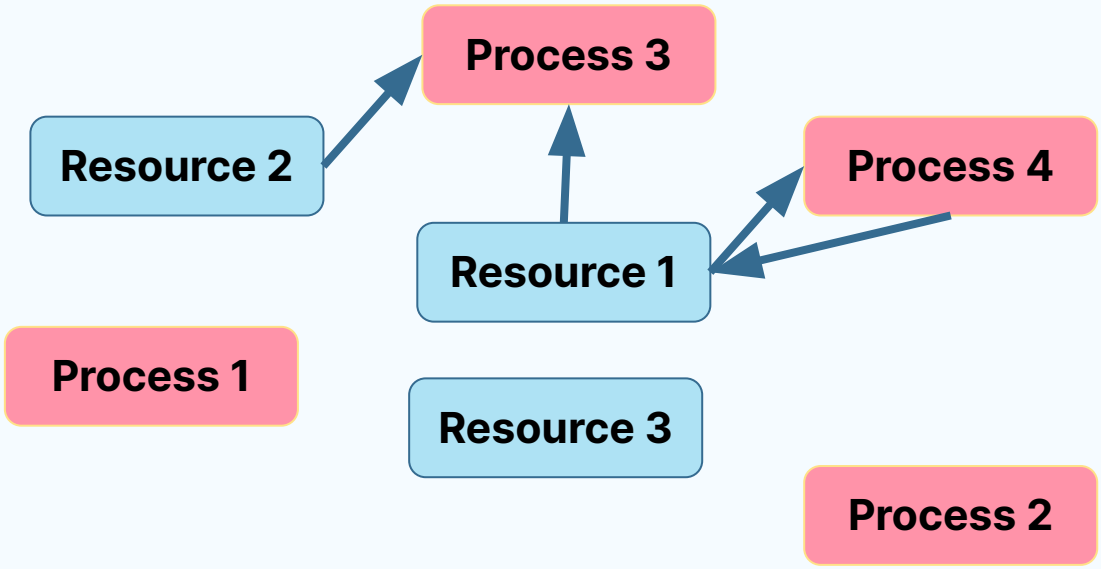
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

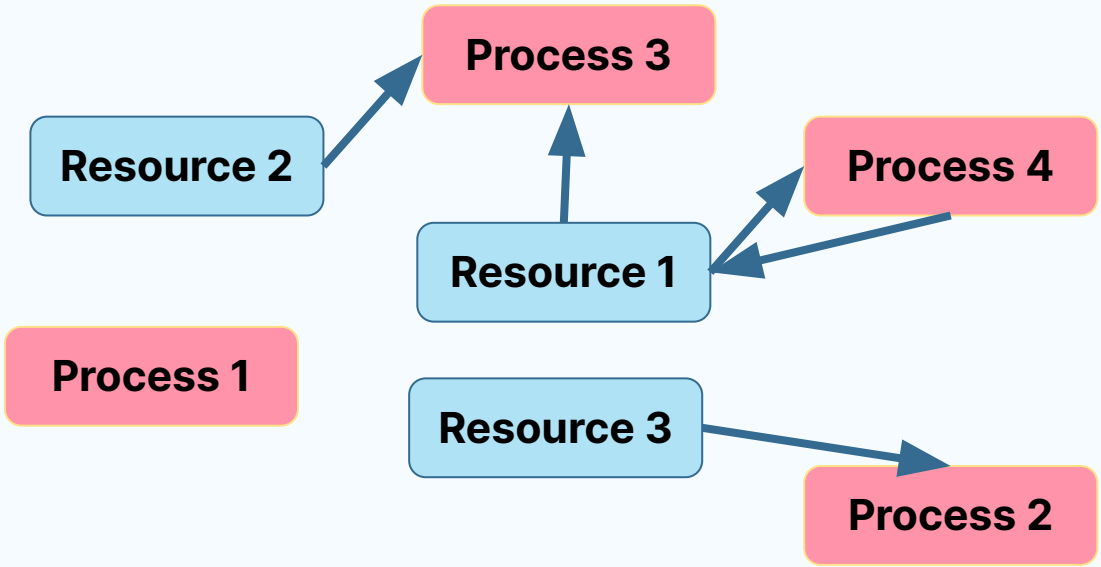
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

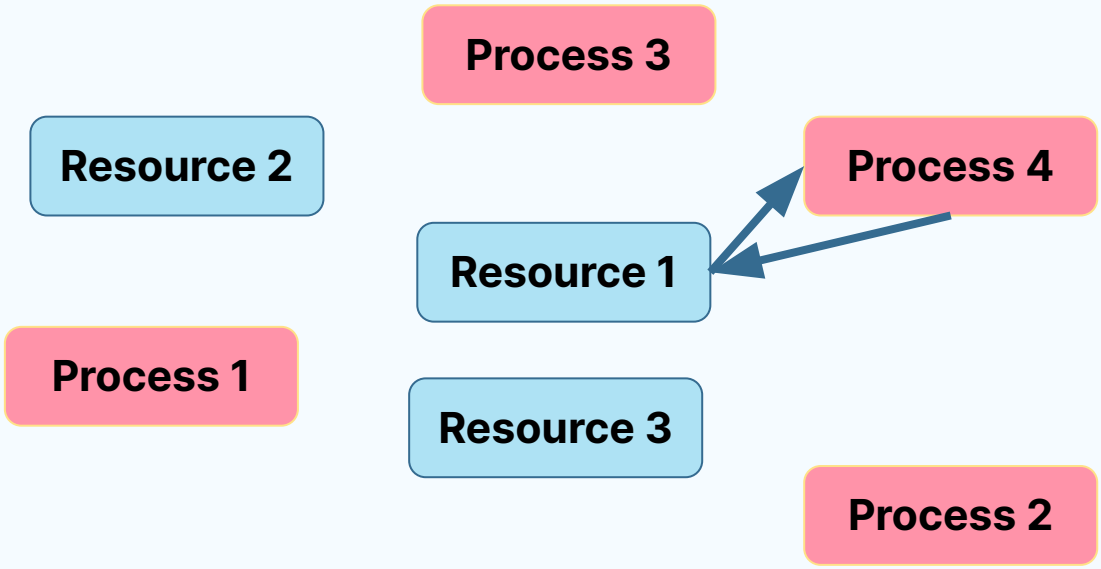
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

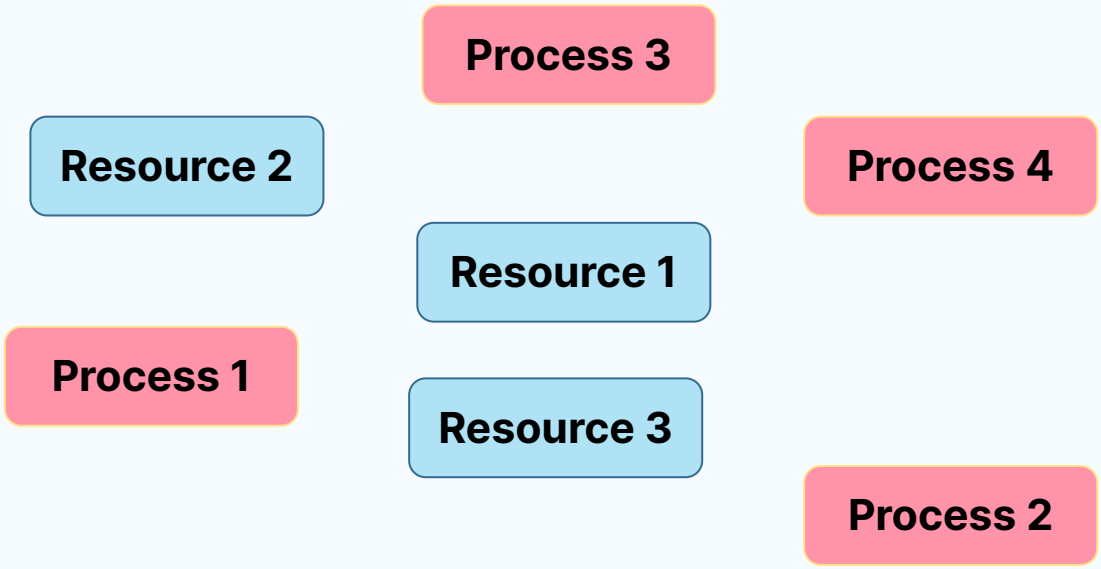
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

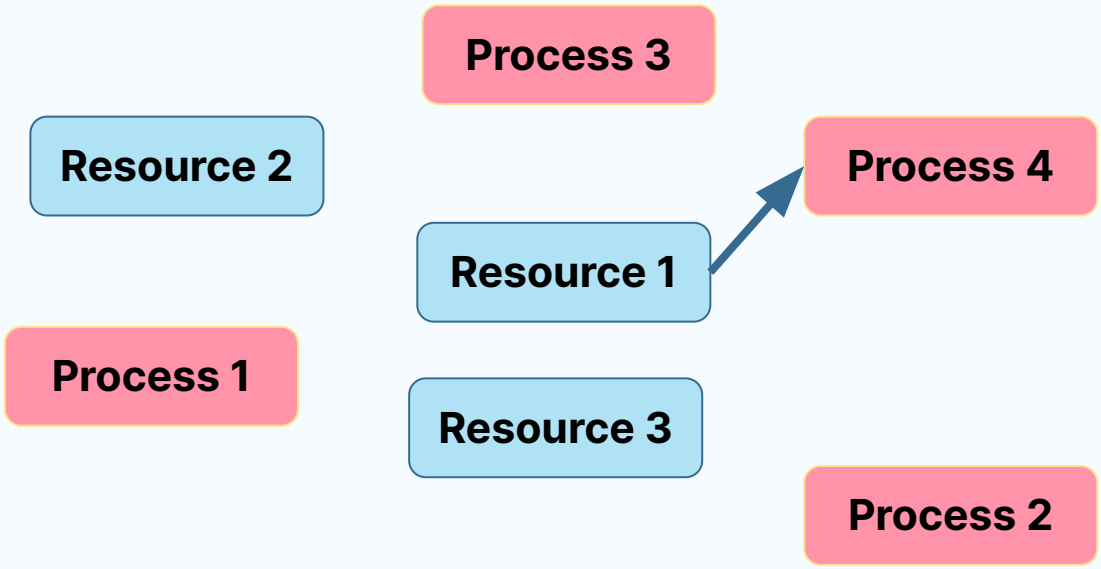
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

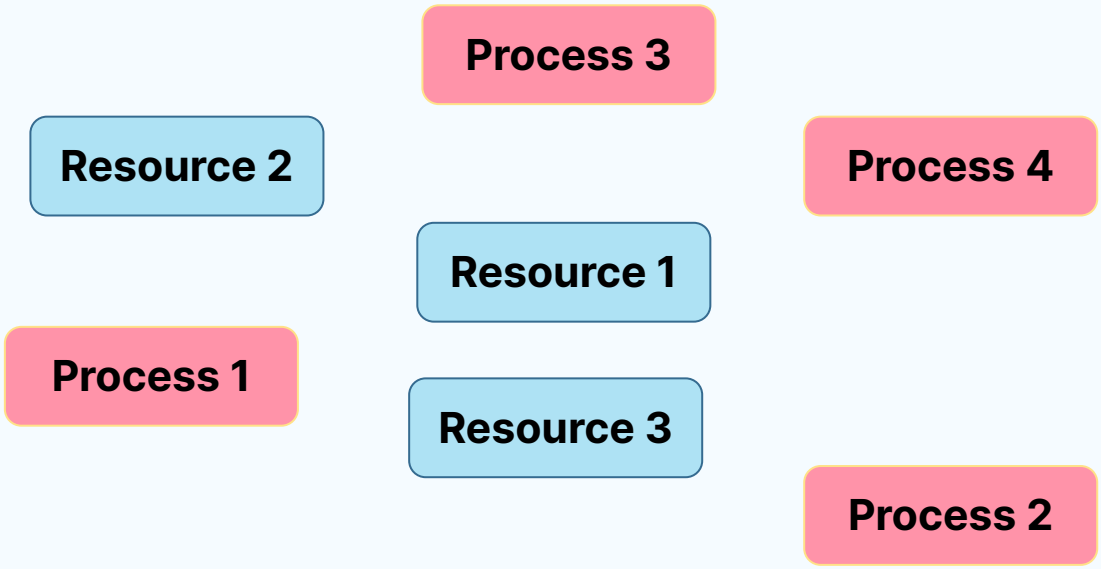
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

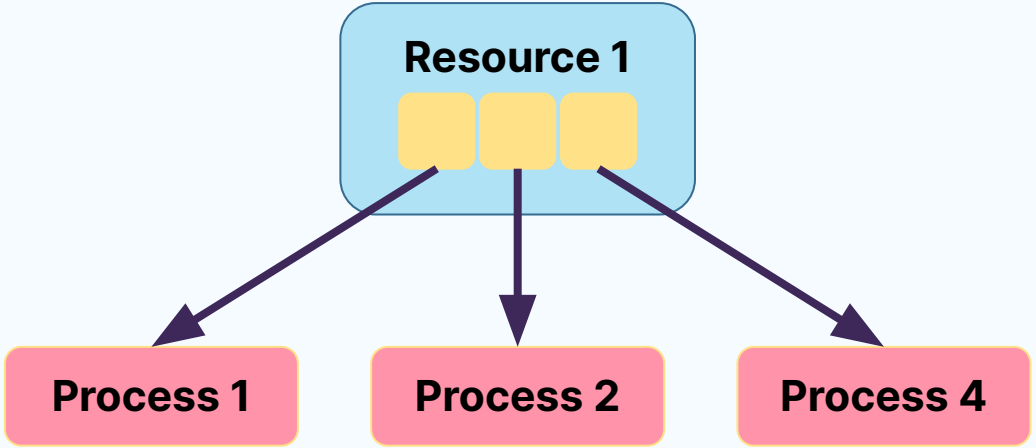
Is this a Deadlock ?



Resource-Allocation Graph 3

Check your understanding

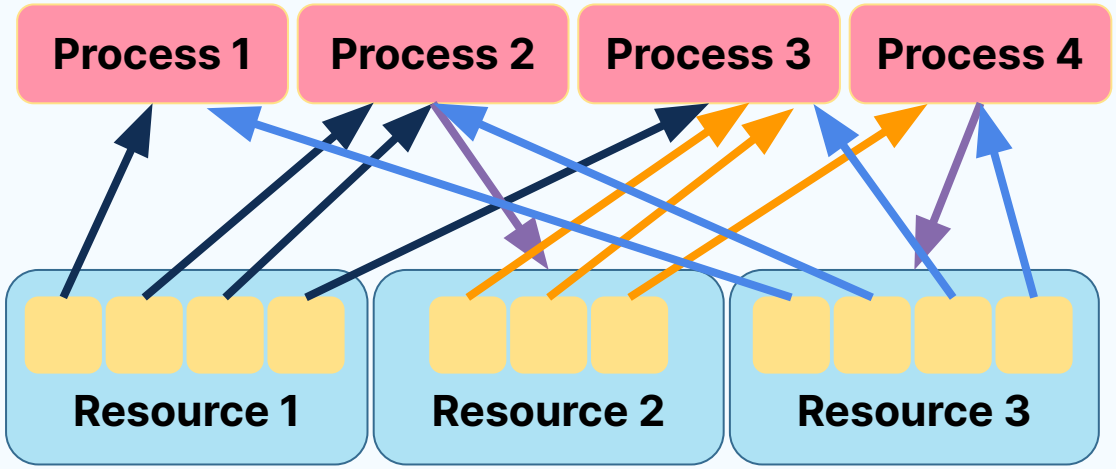
Is this a Deadlock ?



Resource-Allocation Graph 4

Check your understanding

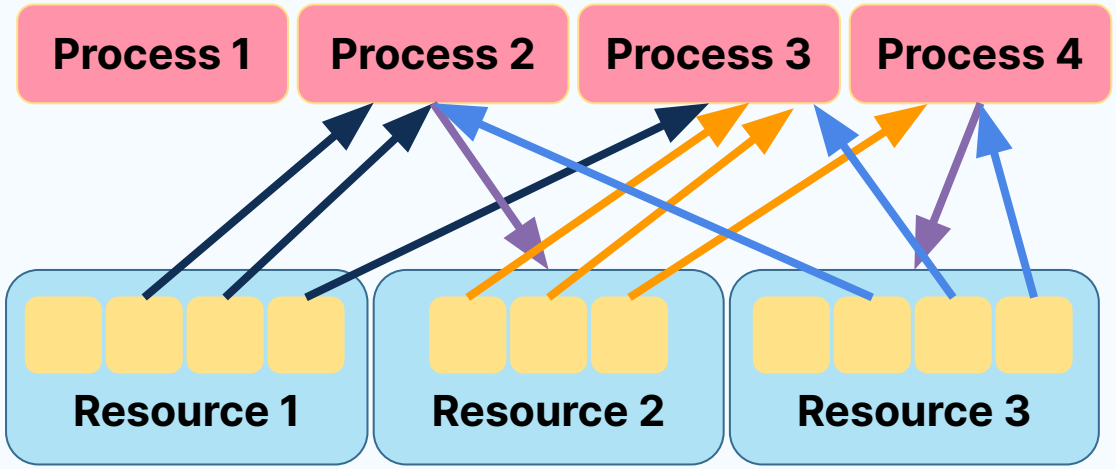
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

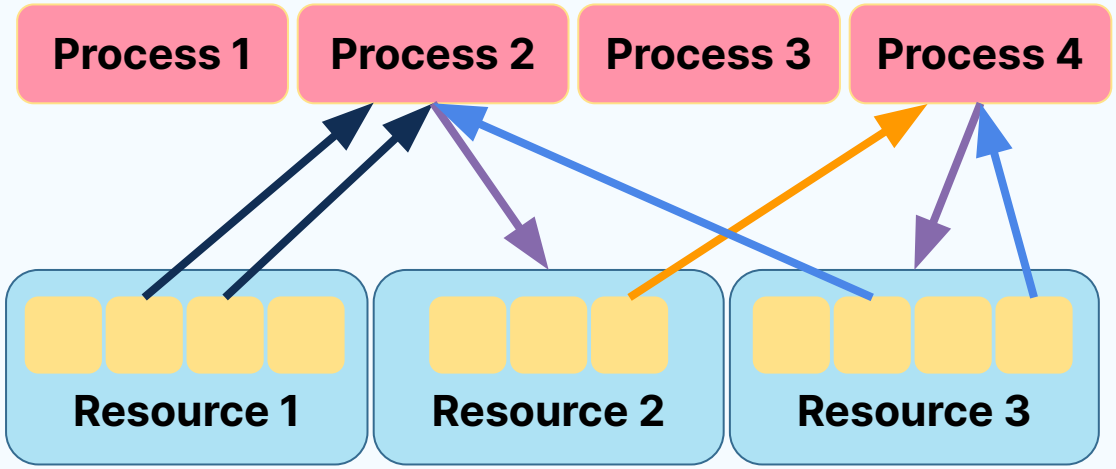
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

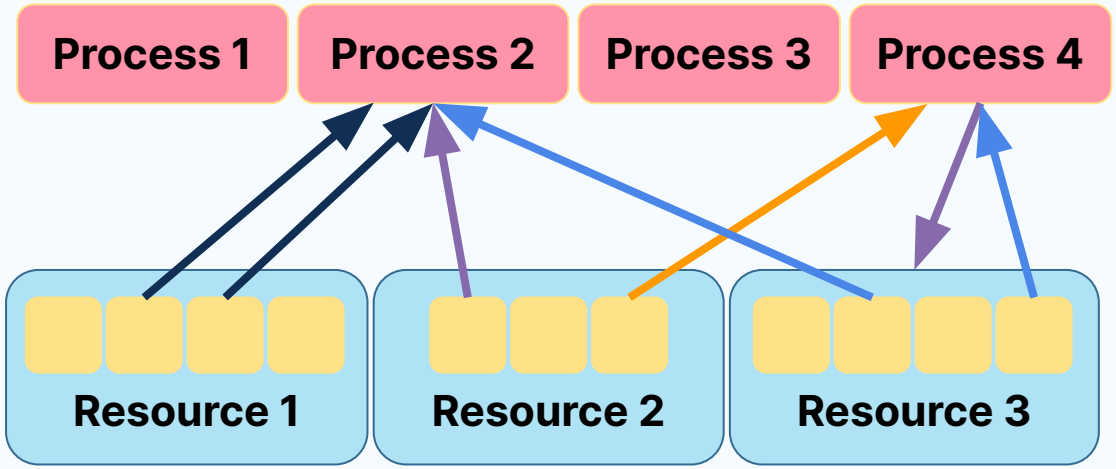
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

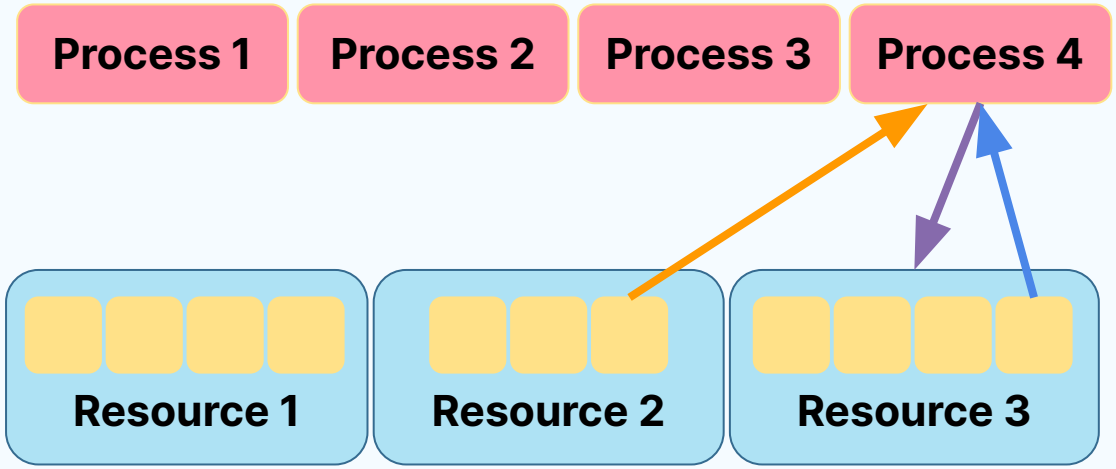
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

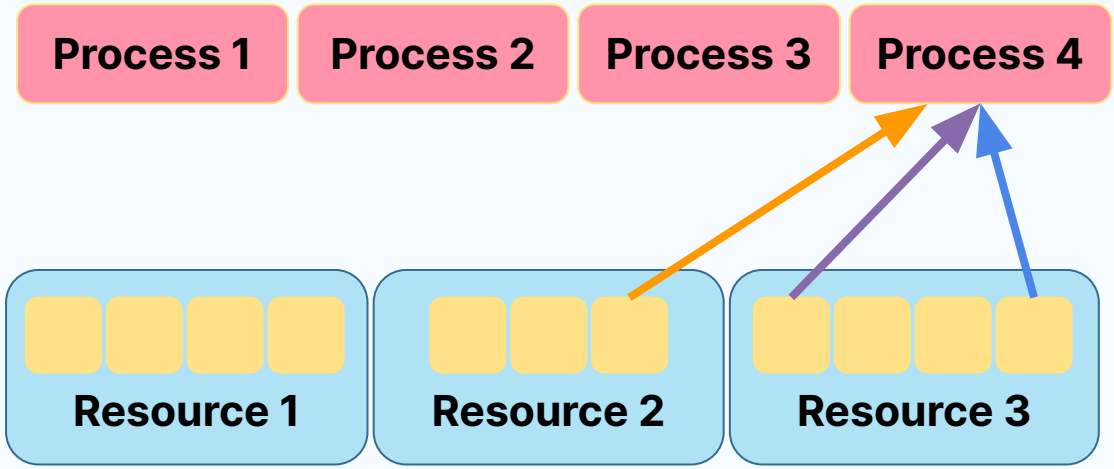
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

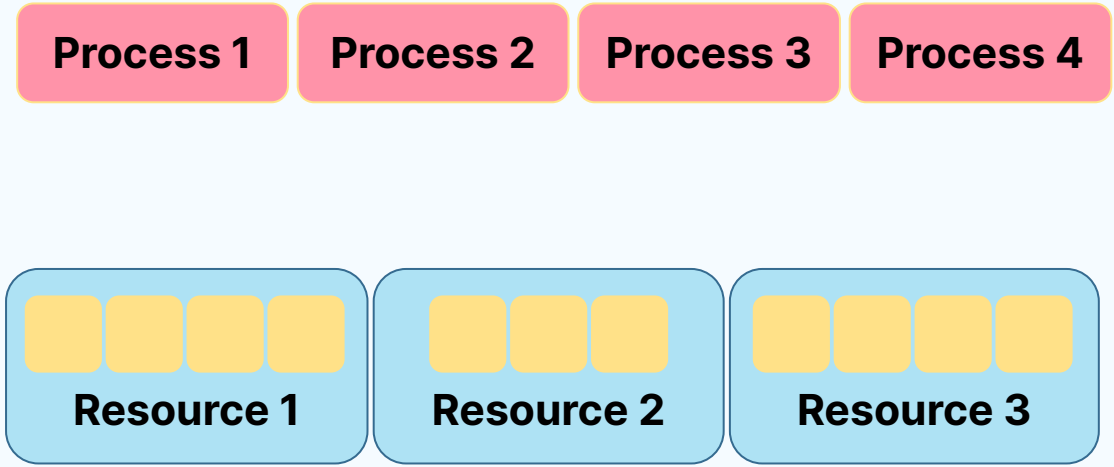
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

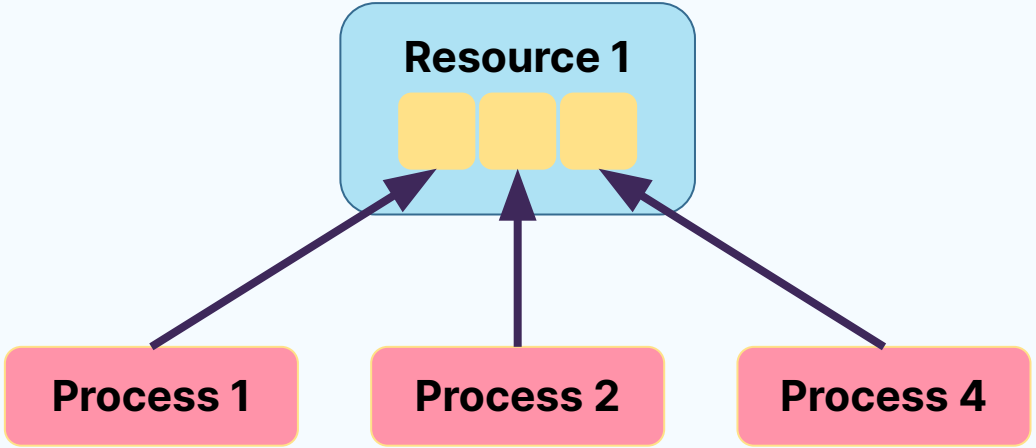
Is this a Deadlock ?



Resource-Allocation Graph 5

Check your understanding

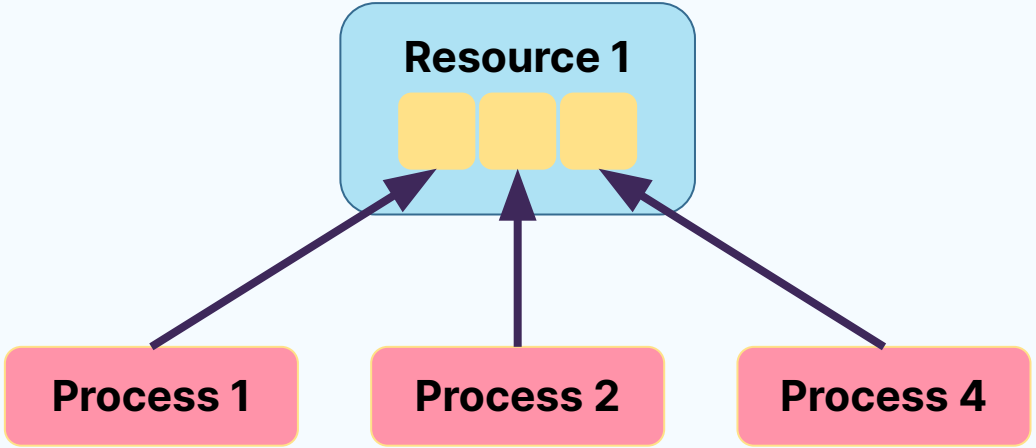
Is this a Deadlock ?



Resource-Allocation Graph 6

Check your understanding

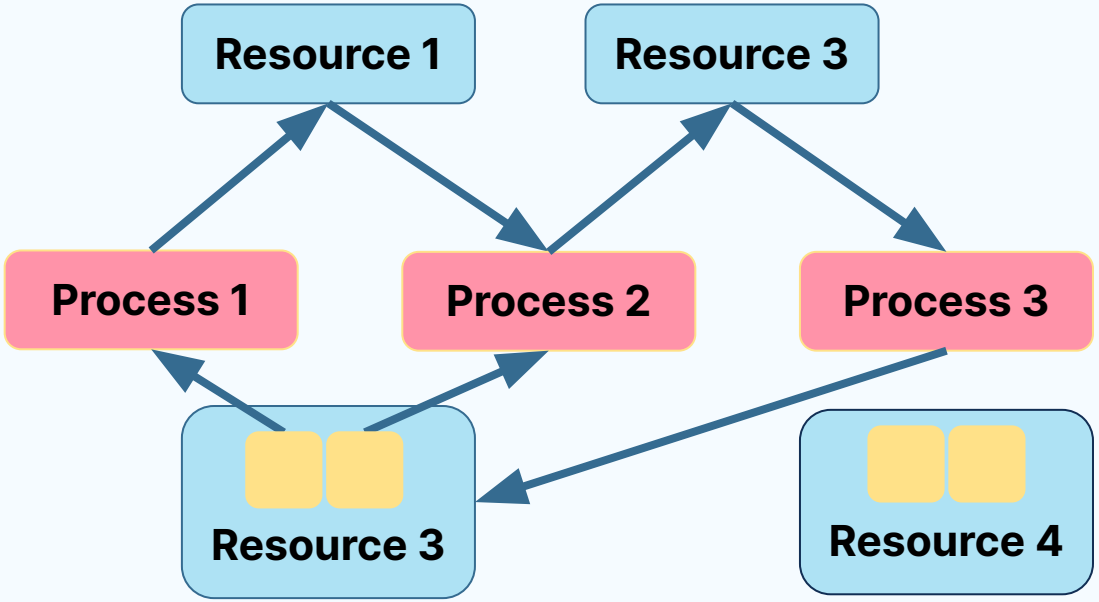
Is this a Deadlock ?



Resource-Allocation Graph 6

Check your understanding

Is this a Deadlock ?



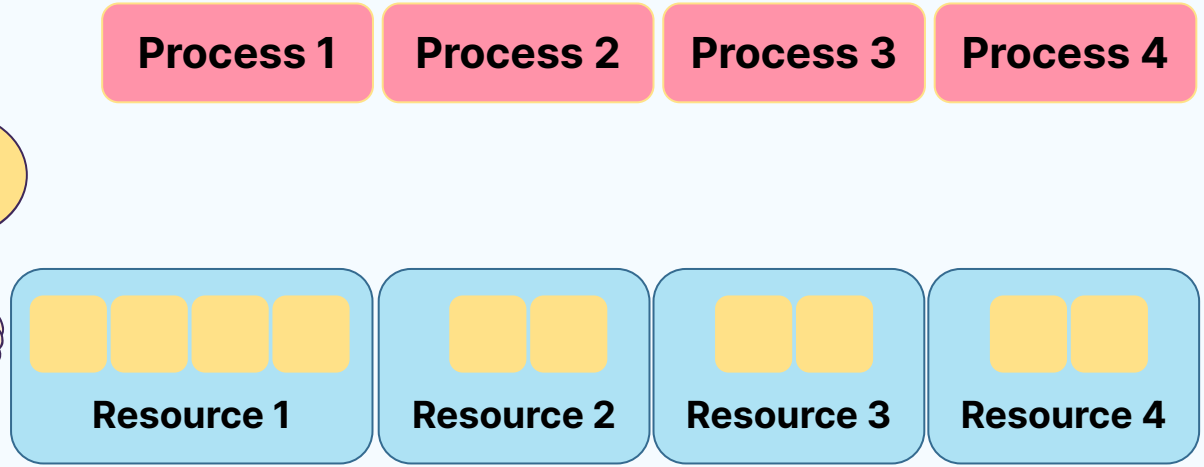
Resource-Allocation Graph 7

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

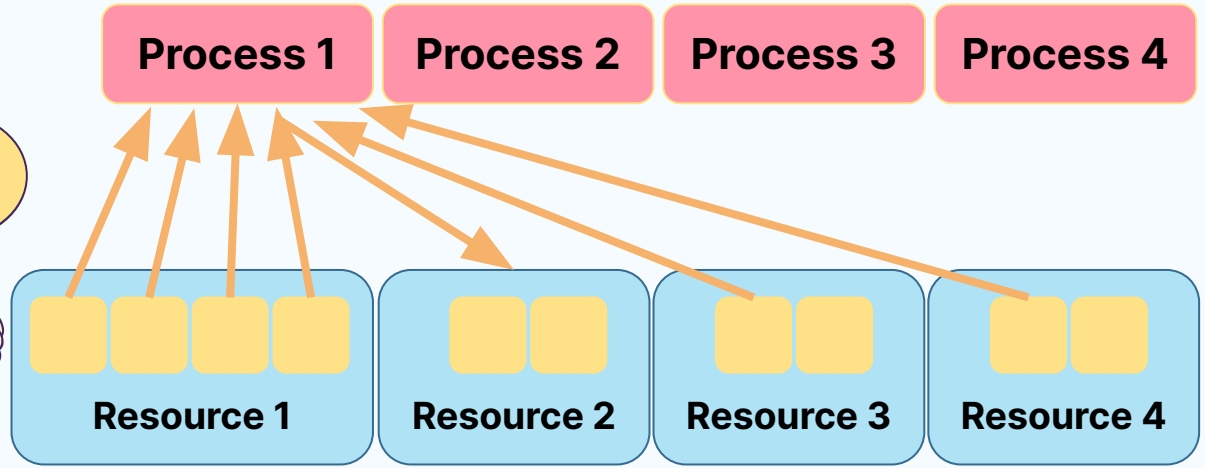
P1 holds all instances of R1, an instance of R3 and an instance of R4 and requests an instance of R2.



Resource-Allocation Graph 8

Check your understanding

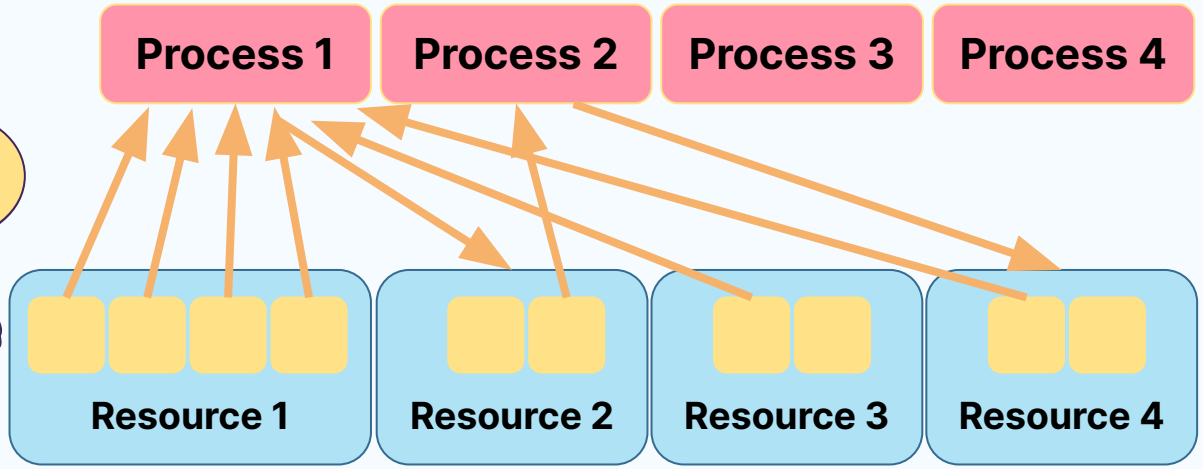
P1 holds all instances of R1, an instance of R3 and an instance of R4 and requests an instance of R2.



Resource-Allocation Graph 8

Check your understanding

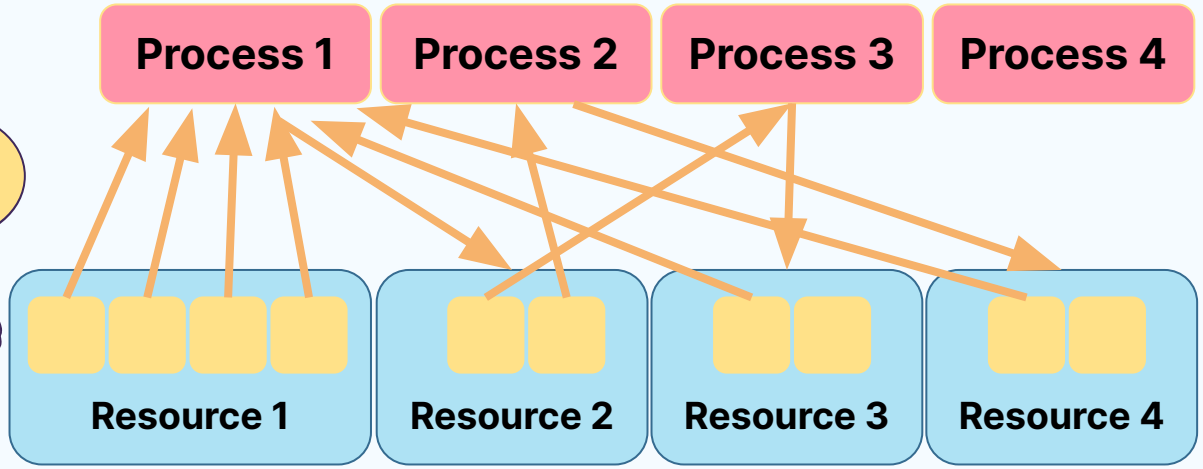
P2 holds an instance of R2 and requests an instance of R4



Resource-Allocation Graph 8

Check your understanding

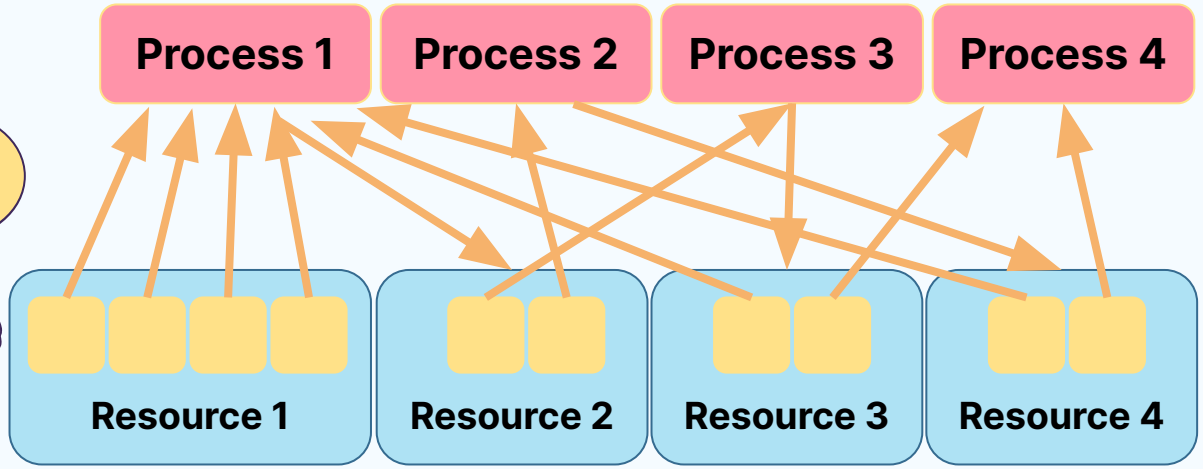
P3 holds an instance of R2 and requests an instance of R3



Resource-Allocation Graph 8

Check your understanding

P4 holds an instant of R3 and R4.



Resource-Allocation Graph 8

Deadlock handling mechanisms

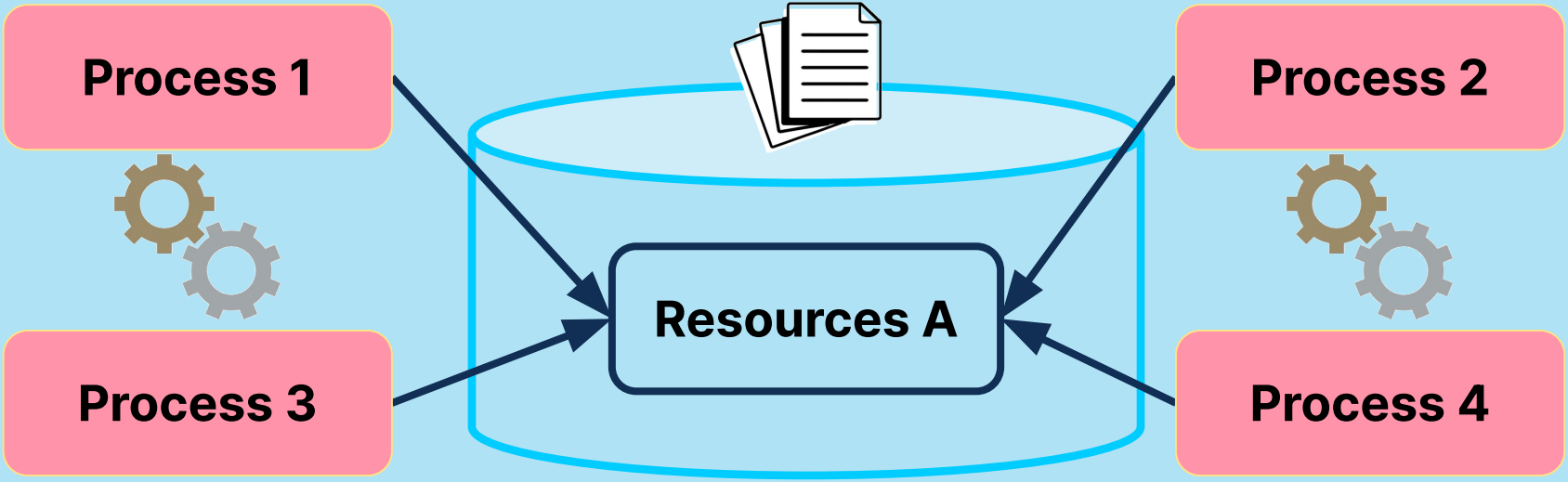
Prevent one of the four conditions from occurring.

Avoid the deadlock if it becomes probable.

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

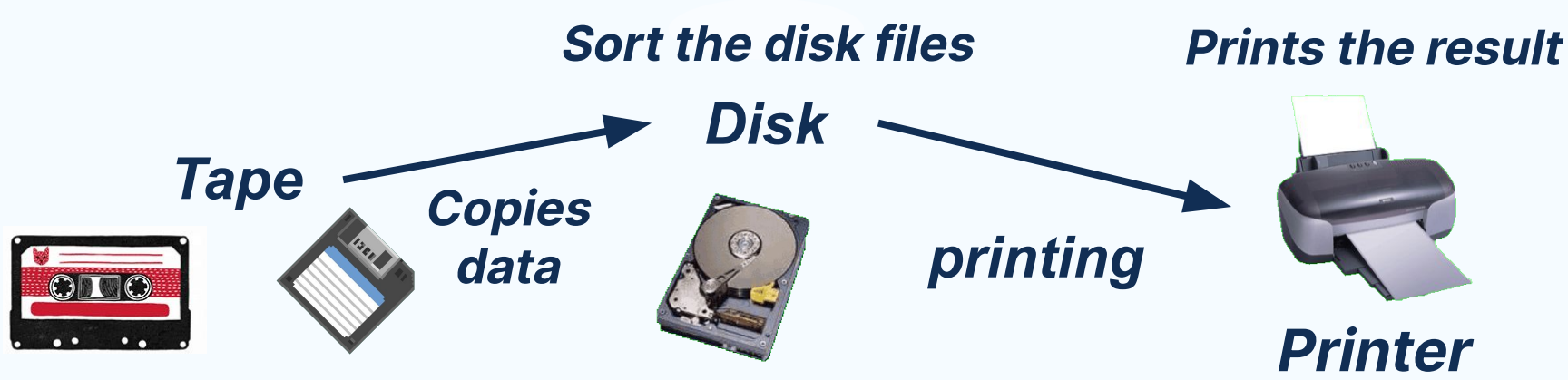
1) Mutual Exclusion
2) Hold and Wait
3) No Preemption
4) Circular Wait

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)

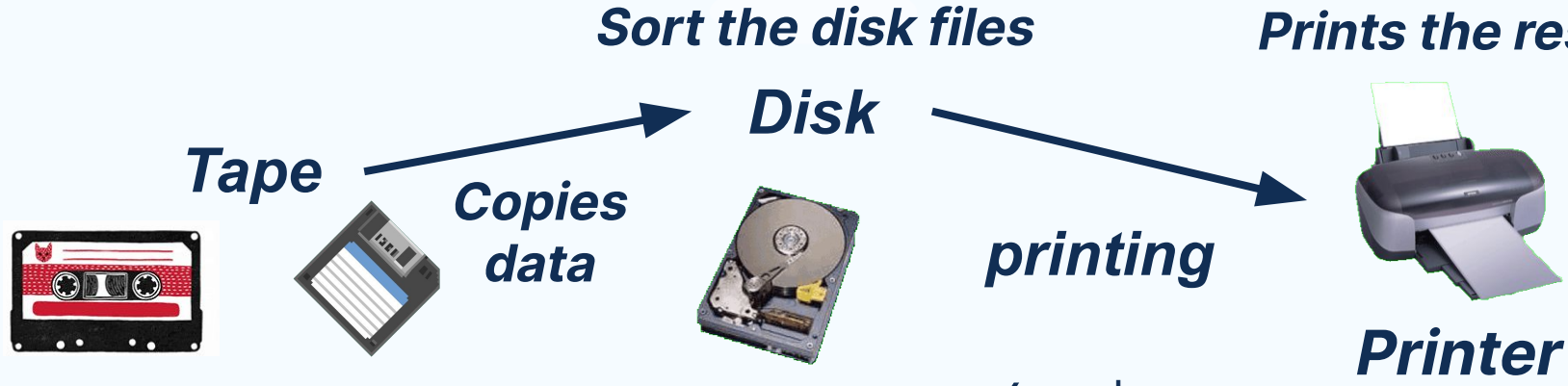
1st protocol: Holding all resources until the end of execution



Starvation (1st protocol)

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

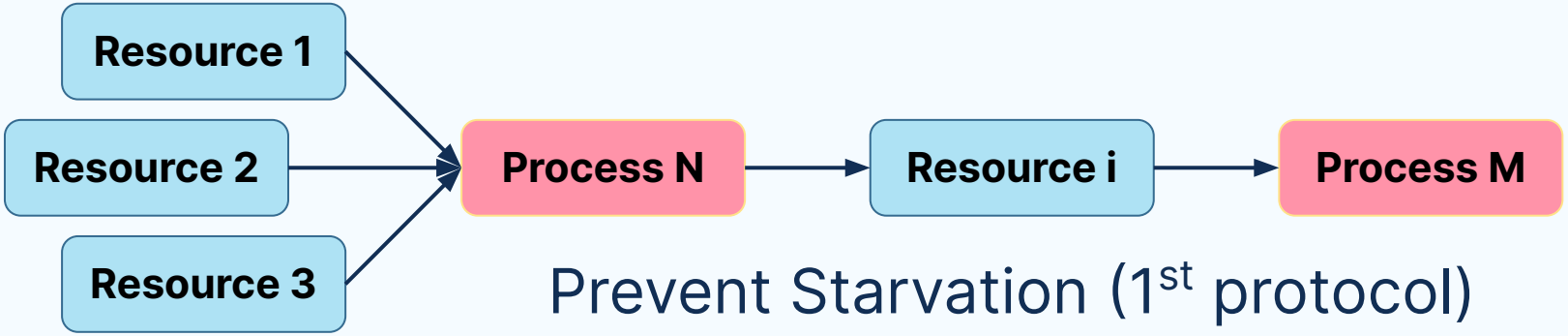
2nd protocol: After using a resource must release and then request again for another execution.



Low resource utilization (2nd protocol)

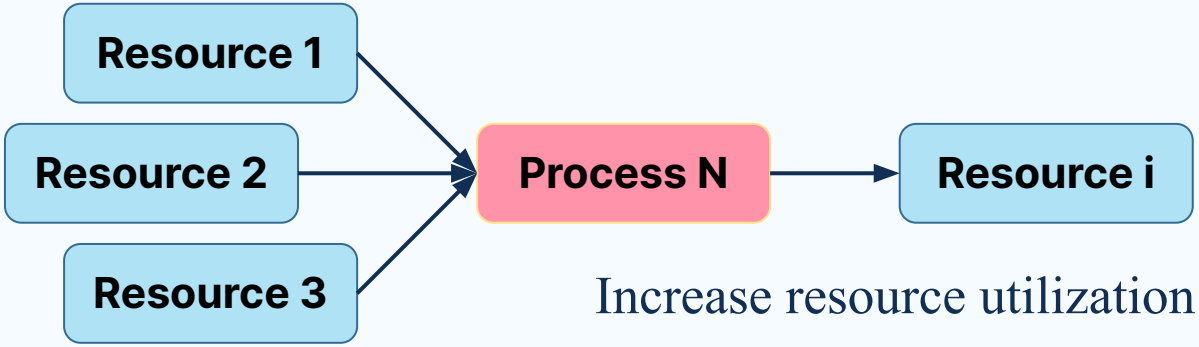
No Preemption – Could be bypassed by allowing OS to deallocate resources from jobs.

(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)



Prevent Starvation (1st protocol)

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.



Increase resource utilization (2nd protocol)

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.

F (Printer) = 1

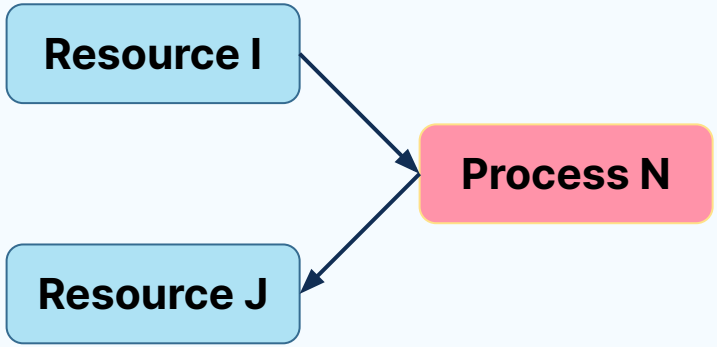
F (disk drive) = 2

F (tape drive) = 3

F (tape drive) = 1
F (disk drive) = 5
F (printer) =12

F(Ri) = 1

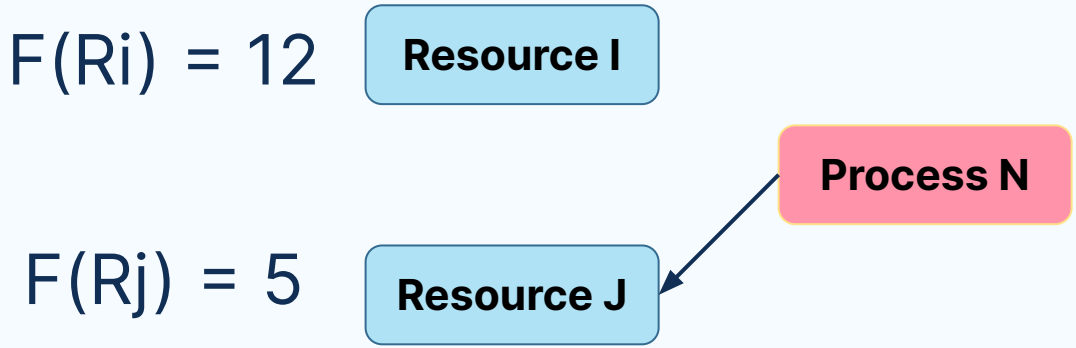
F(Rj) = 5



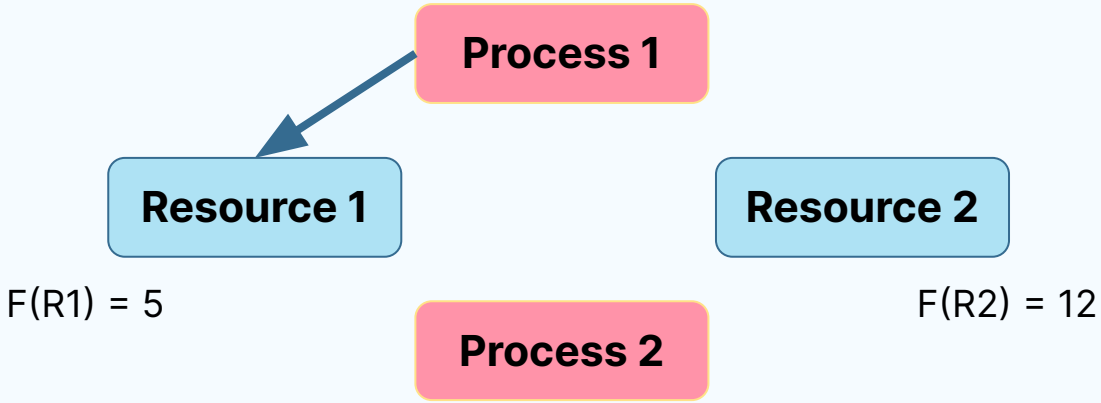
A process Pn using tape drive(1) then disk drive(5)

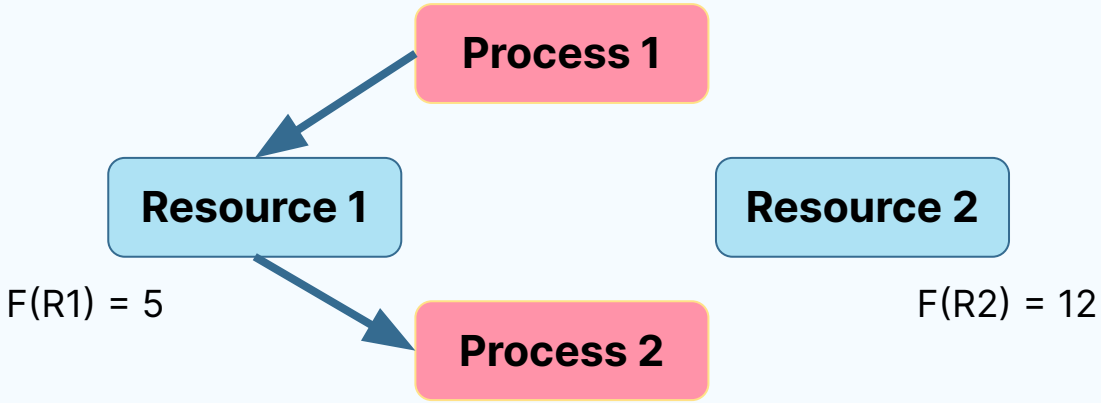
F (tape drive) = 1
F (disk drive) = 5
F (printer) =12

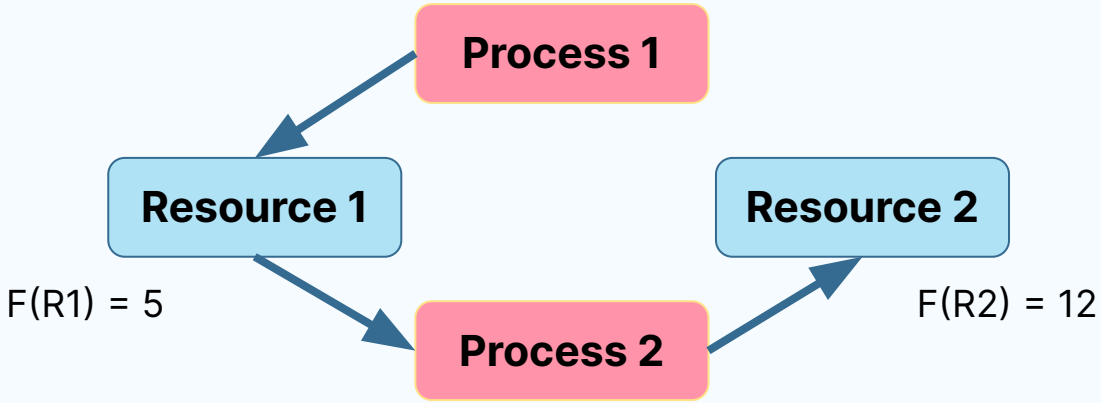
or has released any resource Ri
if F(Ri) >= F(Rj)



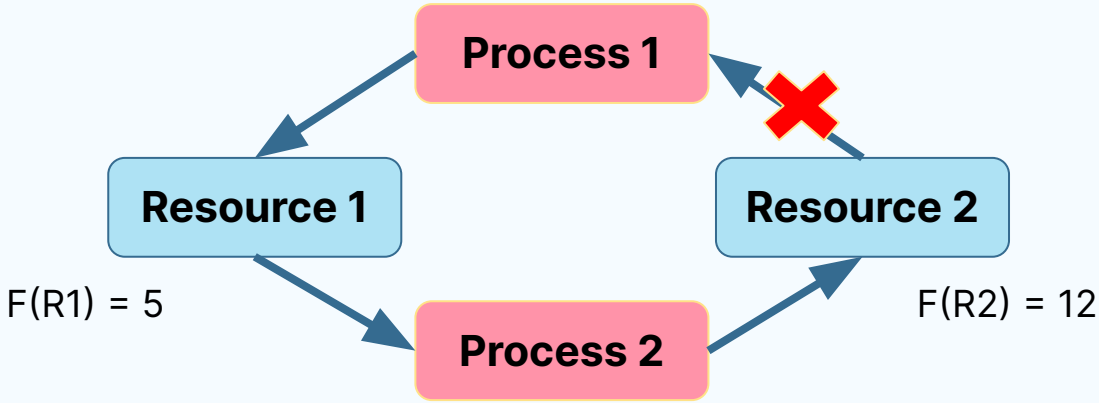
Pn holding printer(12) at the same time requesting tape drive(1)

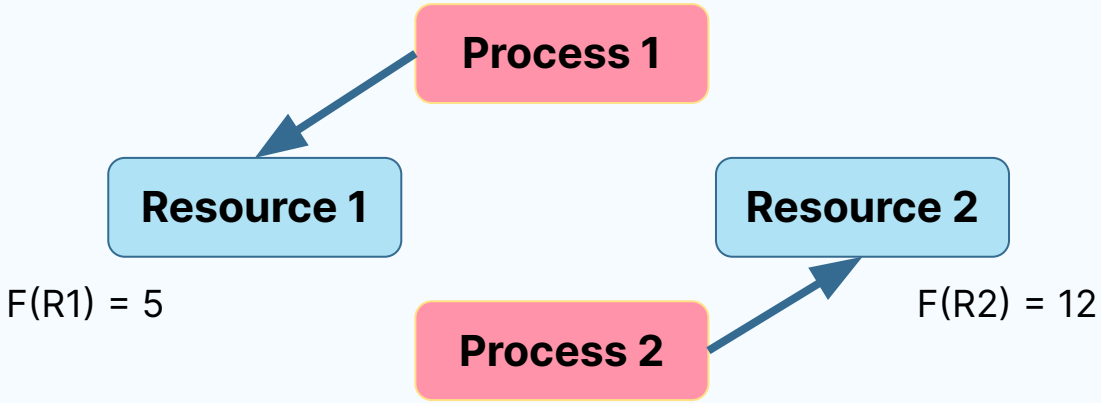

Process 1
Resource 1
Resource 2
F(R1) = 5
F(R2) = 12
Process 2


Process 1
Resource 1
Resource 2
F(R1) = 5
F(R2) = 12
Process 2


Process 1
Resource 1
Resource 2
F(R1) = 5
Process 2
F(R2) = 12

F(R2) >= F(R1)




Process 1
Resource 1
Resource 2
F(R1) = 5
Process 2
F(R2) = 12

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

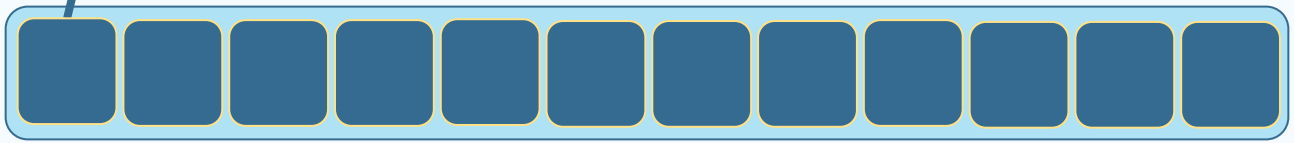
Total loaned: $9,000

Total capital fund: $10,000

Deadlock Avoidance

At t0

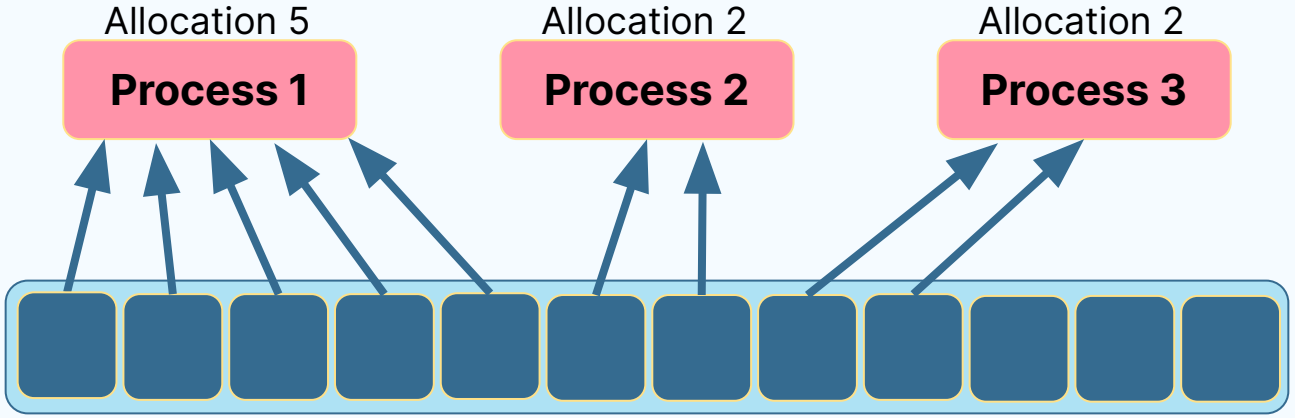
Process 1 | Process 2 | Process 3



	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

Deadlock Avoidance

At t0



	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

Deadlock Avoidance

At t0

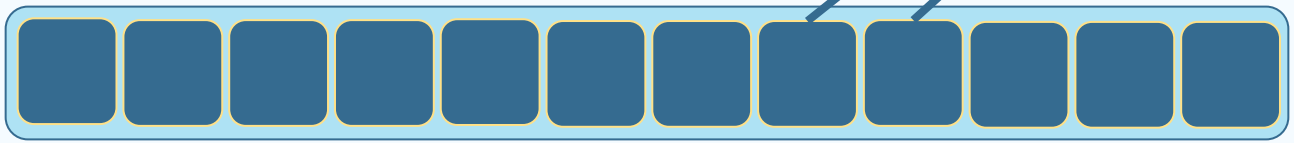
	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

Deadlock Avoidance

At t0

Allocation 2

Process 1 **Process 2** **Process 3**

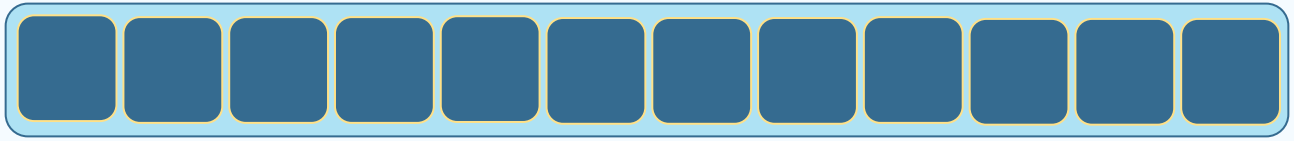


	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

Deadlock Avoidance

At t0

Process 1 **Process 2** **Process 3**



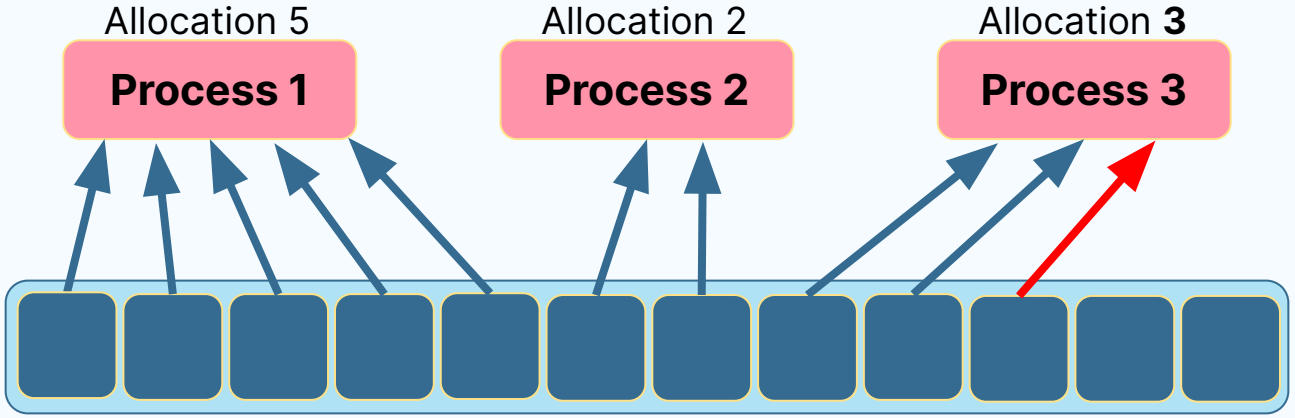
	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

< P2, P1, P3 >

Satisfy the safety condition

Deadlock Avoidance

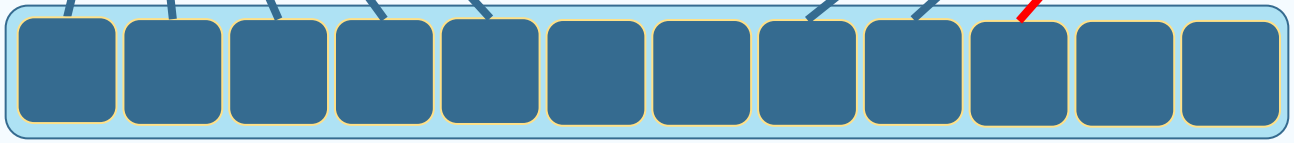
At t1



	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

Deadlock Avoidance

At t1



	Max Needs	Current Needs
Process 1	10	5
Process 2	4	2
Process 3	9	7

< P2, >

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:

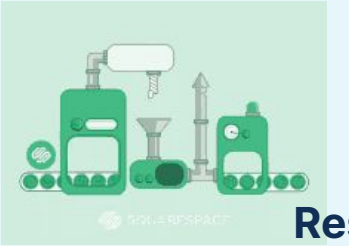
P1 → P3 → **P4** → P2 → P0

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

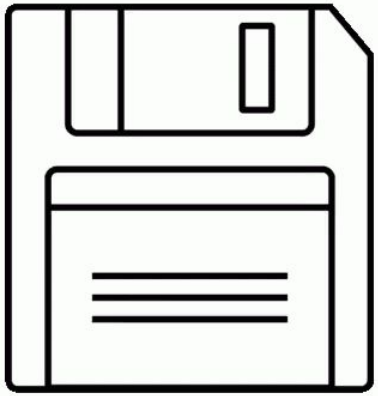
- Terminate every job that's active in system and restart them from beginning.
- Terminate only the jobs involved in deadlock and ask their users to resubmit them.
- Terminate jobs involved in deadlock one at a time, checking to see if deadlock is eliminated after each removal, until it has been resolved.

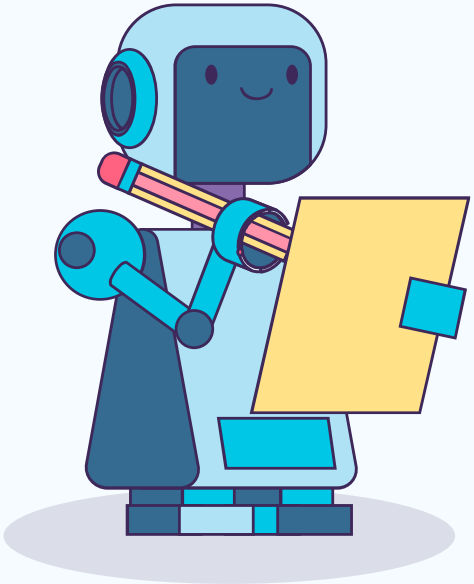
Recovery Algorithms

- 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**