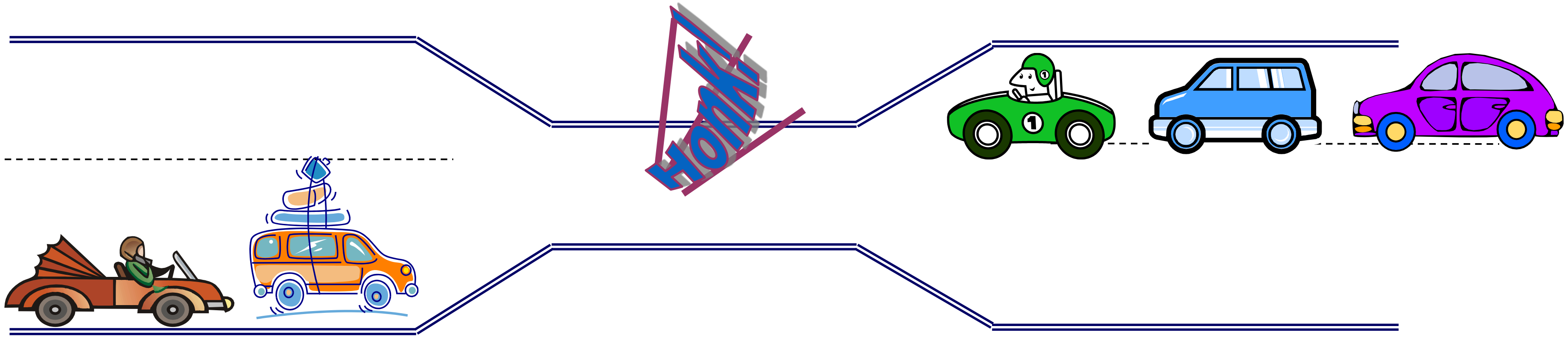
Deadlocks

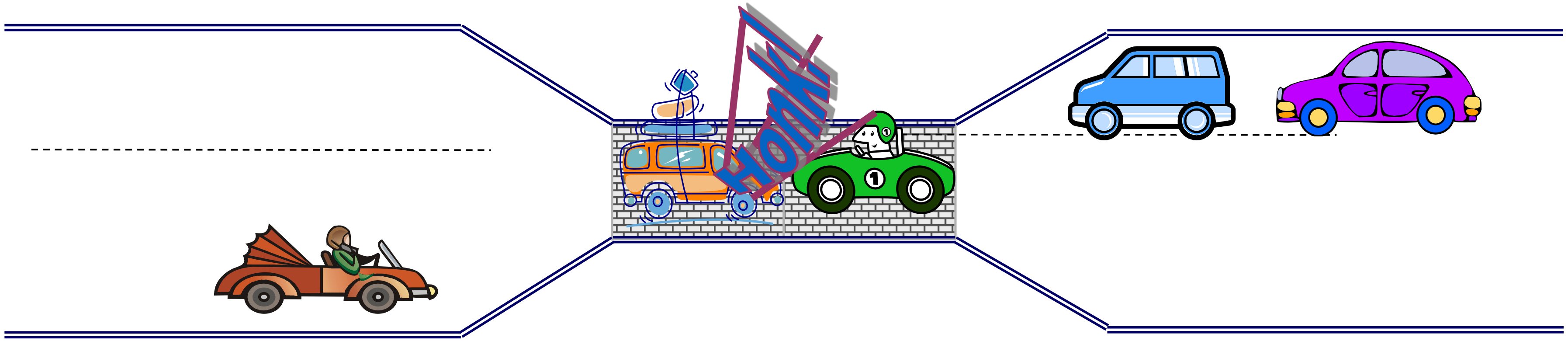

Honk!

Outline

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery From Deadlock

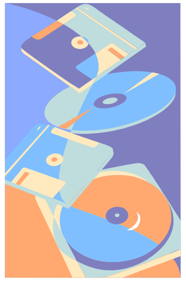
System Model

Bridge Crossing Example



Resources

- Resources – passive entities needed by processes to do their work
  - CPU time, disk space, memory
- Two types of resources:
  - Preemptable – can take it away
    - CPU, memory
  - Non-preemptable – must leave it with the process
    - Disk space, printer, CD-writer
    - Mutual exclusion – the right to enter a critical section



Resources

- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an operating system is to manage resources

Resources

- Resource types $\boldsymbol{R}_1$, $\boldsymbol{R}_2$, . . ., $\boldsymbol{R}_{\boldsymbol{m}}$
- Each resource type $\boldsymbol{R}_{\boldsymbol{i}}$ has $\boldsymbol{W}_{\boldsymbol{i}}$ instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

System Model

- The **Deadlock** Problem
  - A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Deadlocks occur with multiple resources

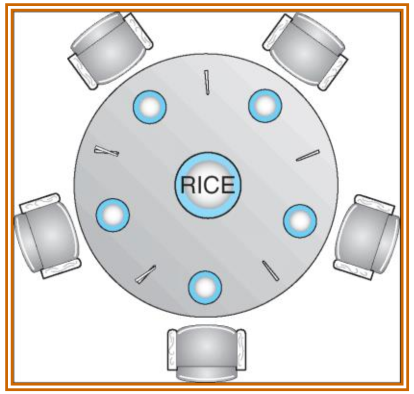
Deadlock

- Deadlock not always deterministic – Example 2 mutexes:
  - Deadlock won’t always happen with this code
    - Have to have exactly the right timing (“wrong” timing?)
    - So you release a piece of software, and you tested it, and there it is, controlling a nuclear power plant

```
Process A        Process B
x.P();           y.P();
y.P();           x.P();
y.V();           x.V();
x.V();           y.V();
```

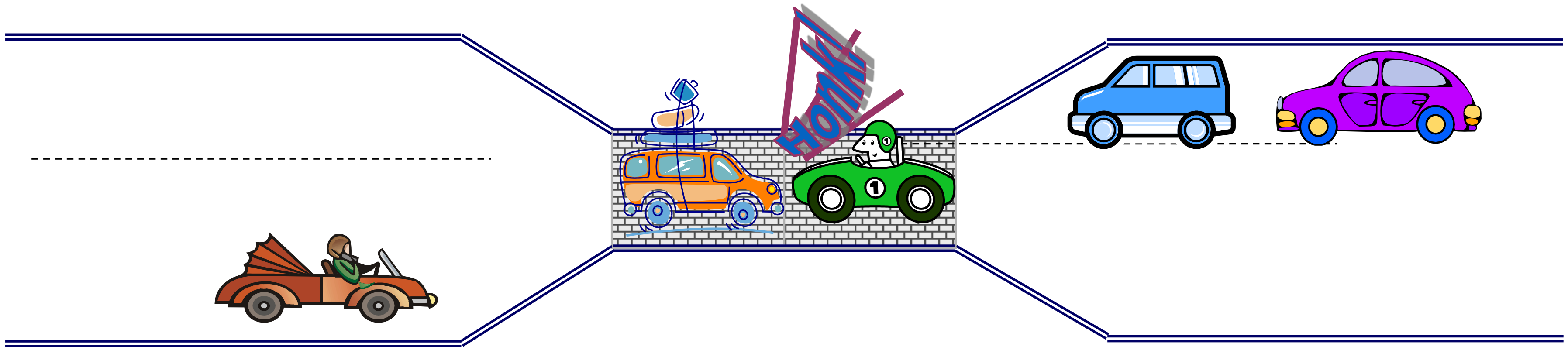
Dining-Philosophers Problem

- Five chopsticks/Five philosophers (really cheap restaurant)
  - Free-for all: Philosopher will grab any one they can
  - Need two chopsticks to eat
- What if all grab at same time?
  - **Deadlock**!



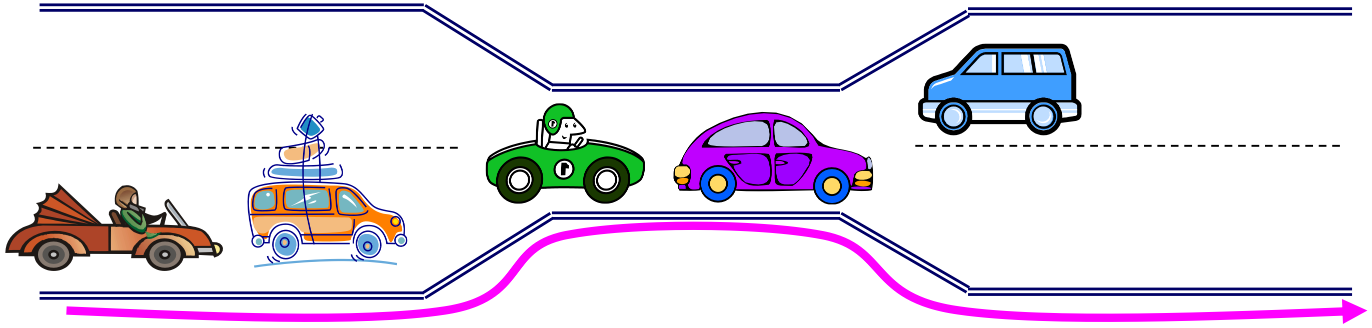
Bridge Crossing Example

- Each segment of road can be viewed as a resource
  - Car must own the segment under them
  - Must acquire segment that they are moving into
- For bridge: must acquire both halves
  - Traffic only in one direction at a time
  - Problem occurs when two cars in opposite directions on bridge: each acquires one segment and needs next



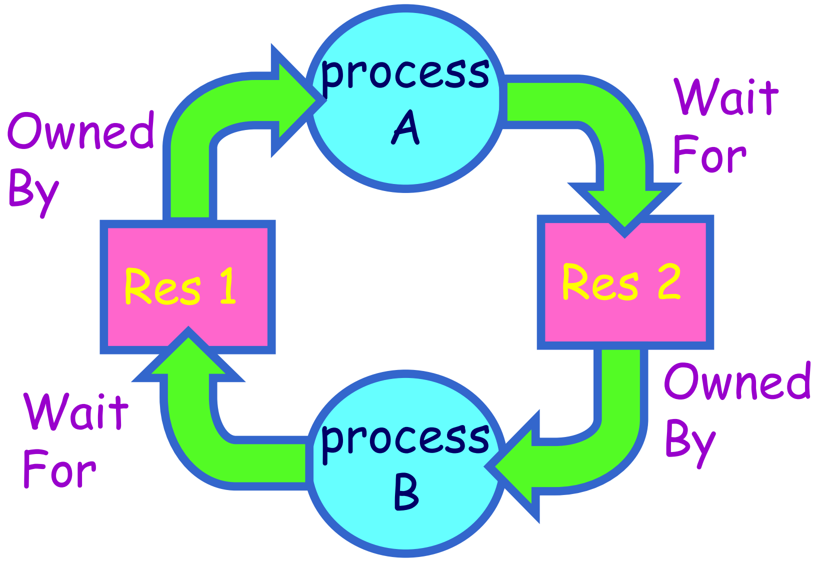
Bridge Crossing Example

- 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
- Starvation is possible
  - East-going traffic really fast $\Rightarrow$ no one goes west



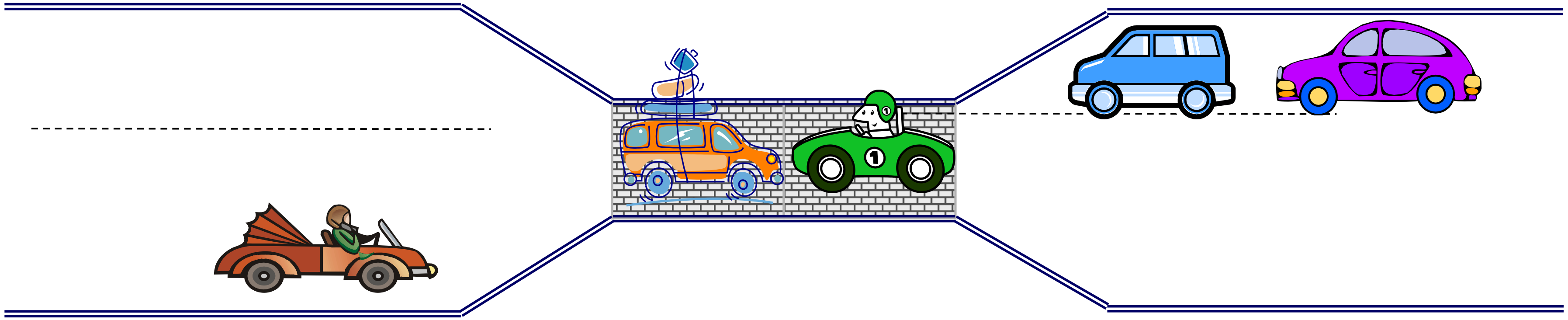
Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: process waits indefinitely
  - Deadlock: circular waiting for resources
  - Deadlock ⇒ Starvation but not vice versa



Deadlock Characterization

Deadlock Characterization



Four Requirements for Deadlock

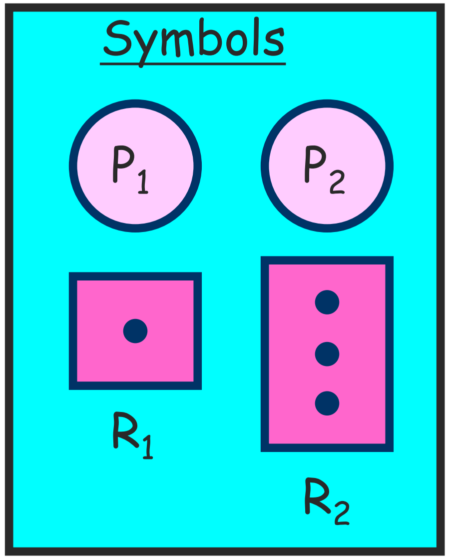
- **Mutual exclusion**
  - Only one process at a time can use a resource.
- **Hold and wait**
  - Process is holding at least one resource and is waiting to acquire additional resources held by other processes
- **No preemption**
  - Resources are released only voluntarily by the process holding the resource, after process is finished with it
- **Circular wait**
  - There exists a set $\{P_1, \ldots, P_n\}$ of waiting processes
    - P_1 is waiting for a resource that is held by P_2
    - P_2 is waiting for a resource that is held by P_3
    - …
    - P_n is waiting for a resource that is held by P_1

Four Requirements for Deadlock

- When deadlock occurs, all four conditions hold
- Not sufficient conditions
- Not completely independently
  - “Circular wait” implies “Hold and wait”

Resource-Allocation Graph

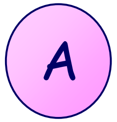
- **System Model**
  - A set of Processes $P_1, P_2, \ldots, P_n$
  - Resource types $R_1, R_2, \ldots, R_m$
    - CPU cycles, memory space, I/O devices
  - Each resource type R_i has W_i instances.
  - Each process utilizes a resource as follows:
    - `Request() / Use() / Release()`

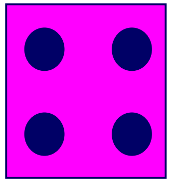
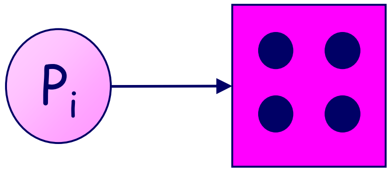


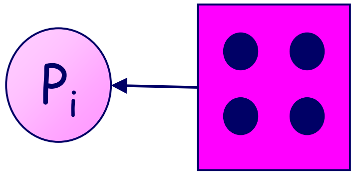
Resource-Allocation Graph

- A set of vertices $\boldsymbol{V}$ and a set of edges $\boldsymbol{E}$.
- $\boldsymbol{V}$ is partitioned into two types:
  - $\boldsymbol{P} = \{\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_{\boldsymbol{n}}\}$, the set consisting of all the processes in the system.
  - $\boldsymbol{R} = \{\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_{\boldsymbol{m}}\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $\boldsymbol{P}_{\boldsymbol{i}} \rightarrow \boldsymbol{R}_{\boldsymbol{j}}$
- Assignment edge – directed edge $\boldsymbol{R}_{\boldsymbol{j}} \rightarrow \boldsymbol{P}_{\boldsymbol{i}}$

Resource-Allocation Graph

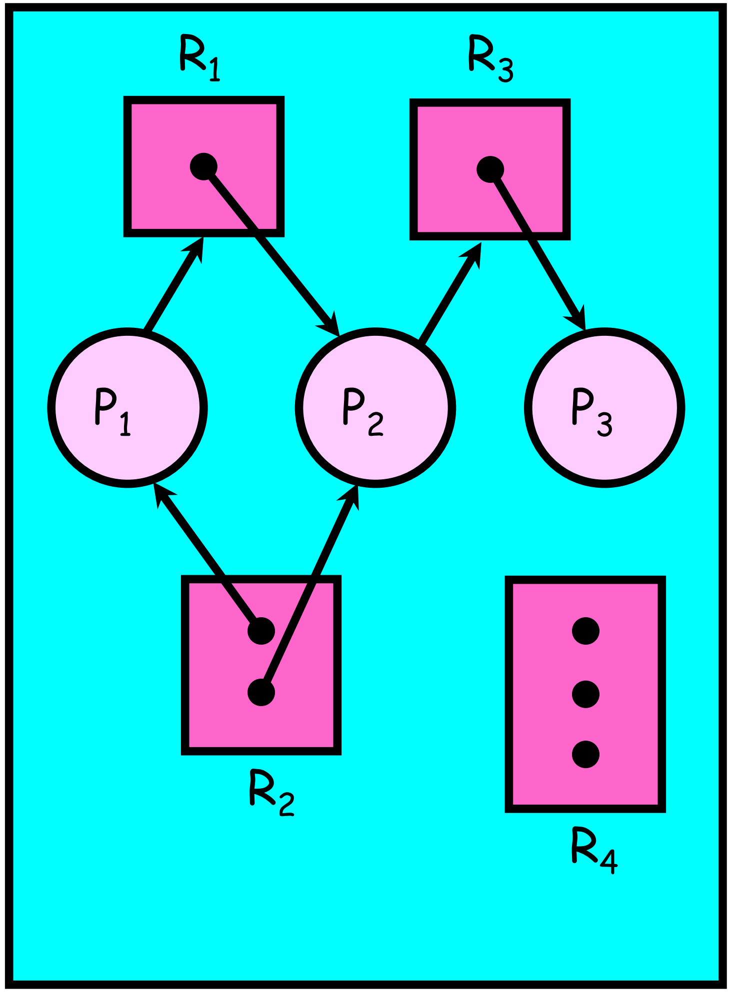
- Resource-Allocation Graph
  - Process 

  - Resource Type with 4 instances 
  - P_i requests instance of R_j 

  - P_i is holding an instance of R_j 

Resource Allocation Graph Examples

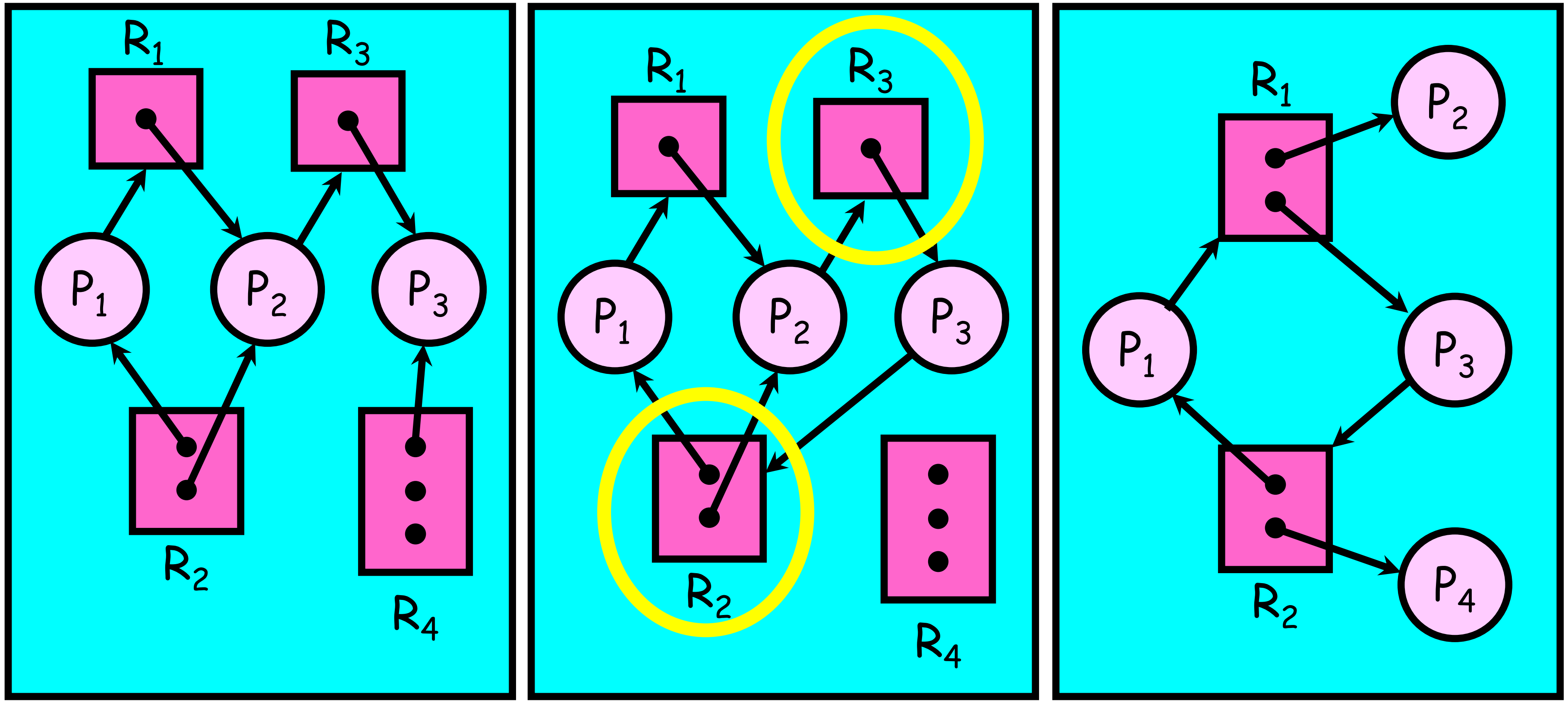
- $P = \{ P_1, P_2, P_3 \}$
- $R = \{ R_1, R_2, R_3, R_4 \}$
- $E = \{ P_1 \rightarrow R_1 , P_2 \rightarrow R_3 ,$
  $R_2 \rightarrow P_1, R_2 \rightarrow P_2 ,$
  $R_1 \rightarrow P_2, R_3 \rightarrow P_3 \}$



Simple Resource Allocation Graph

Resource Allocation Graph Examples

- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$



Simple Resource Allocation Graph

Allocation Graph With Deadlock

Allocation Graph With Cycle, but No Deadlock

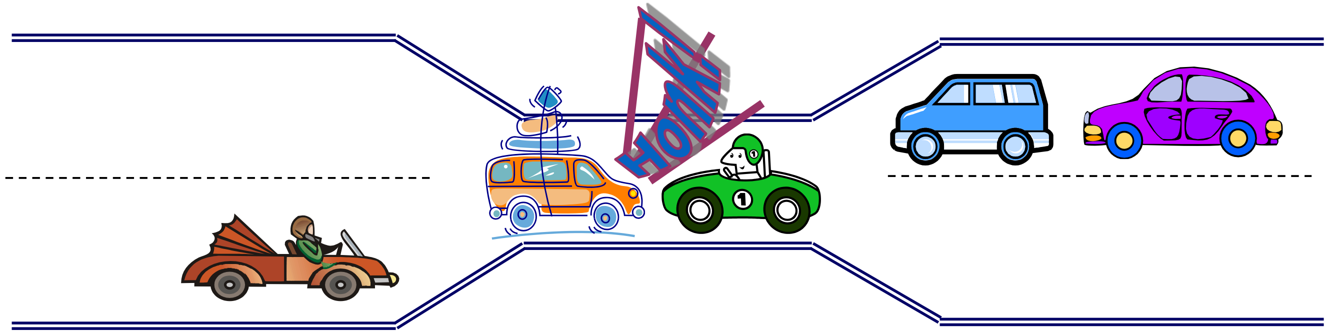
Deadlock Characterization

- Basic Facts
  - If graph contains no cycles ⇒ no deadlock
  - If graph contains a cycle ⇒
    - if only one instance per resource type, then deadlock
    - if several instances per resource type, possibility of deadlock
  - In general, deadlock ⇒ cycle
    - cycle ⇒ may deadlock
    - no cycle ⇒ no deadlock
    - no deadlock ⇒ may have cycle
  - cycle + each resource in the cycle has only an instance

Methods for Handling Deadlocks

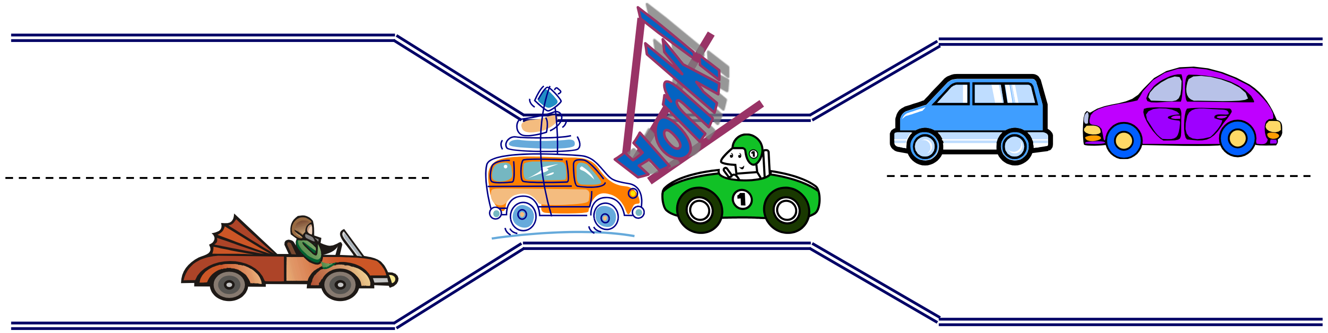
Methods for Handling Deadlocks

- Method 1
  - Left it unsolved



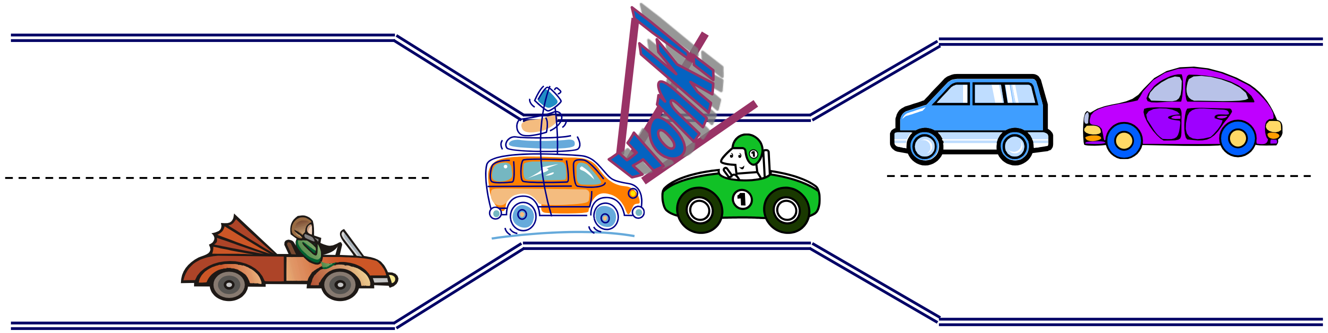
Methods for Handling Deadlocks

- Method 2
  - DELETE the two cars



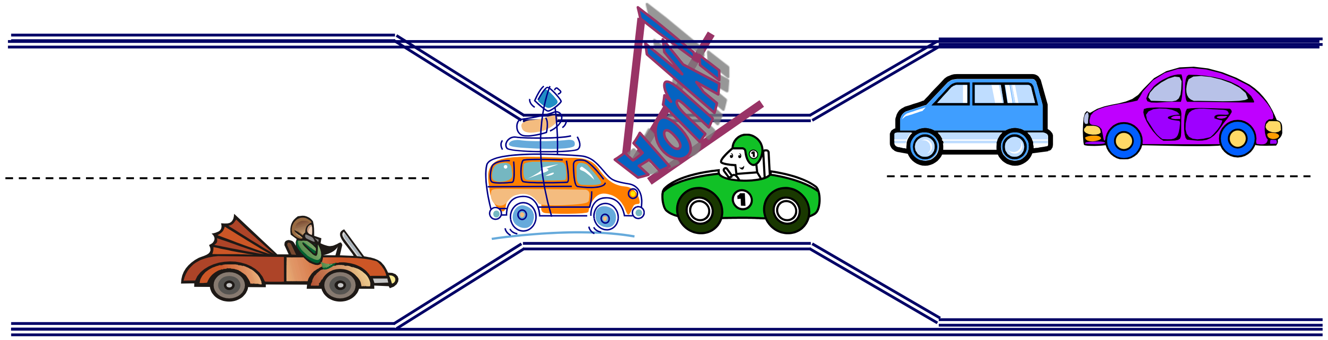
Methods for Handling Deadlocks

- Method 2
  - DELETE the two cars, or
  - Back the cars and restart



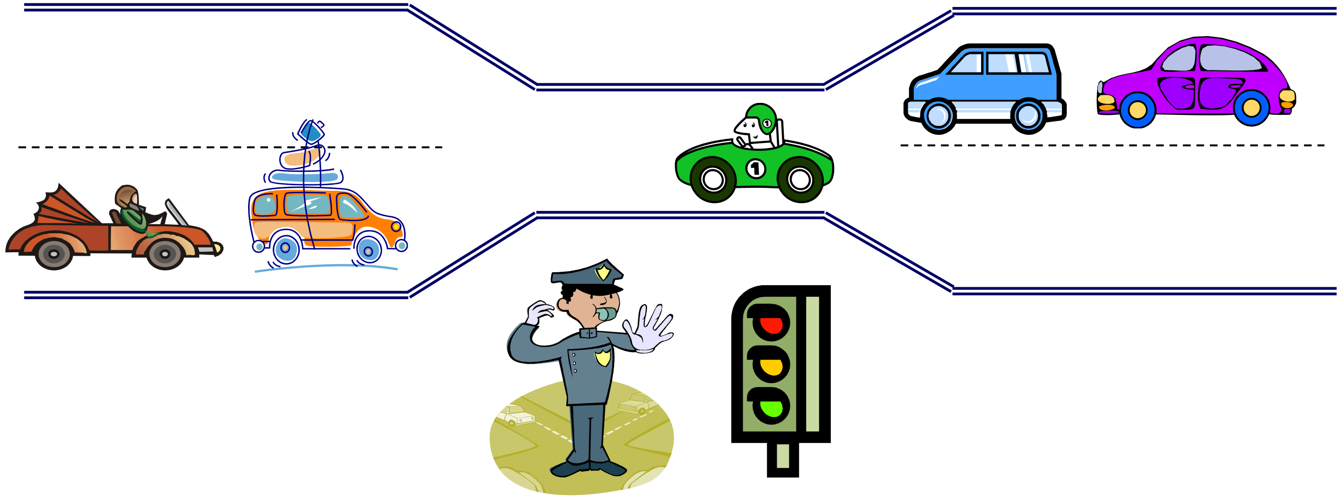
Methods for Handling Deadlocks

- Method 3
  - Widen the road



Methods for Handling Deadlocks

- Method 4
  - Ask for a police



Methods for Handling Deadlocks

- Method 1
  - Left it unsolved
- Method 2
  - DELETE the two cars, or
  - Back the cars and restart
- Method 3
  - Widen the road
- Method 4
  - Ask for a police

- Ignore the problem and pretend that deadlocks never occur in the system
- Allow system to enter deadlock and then recover
- Ensure that system will never enter a deadlock

Methods for Handling Deadlocks

- Ignore the problem and **pretend** that deadlocks never occur in the system
  - Used by most operating systems, including UNIX and Windows
- Ensure that the system will **never** enter a deadlock state
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
  - Deadlock detection
  - Deadlock recovery

Deadlock Prevention

Four requirements for Deadlock

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

- Deadlock $\Rightarrow$ Mutual exclusion $\wedge$ Hold and wait $\wedge$ No preemption $\wedge$ Circular wait
- Mutual exclusion $\vee$ Hold and wait $\vee$ No preemption $\vee$ Circular wait $\Rightarrow$ ~Deadlock

Deadlock Prevention ①

- **Mutual Exclusion** – not required for sharable resources; must hold for non-sharable resources.
  - Some resources are intrinsically non-sharable
  - Not very realistic

Deadlock Prevention ①

- Infinite resources
  - Examples:
    - Bay bridge with 12,000 lanes. Never wait!
  - Include **enough** resources so that **no** one ever runs out of resources.
  - Doesn’t have to be infinite, just large
  - Give **illusion** of infinite resources (e.g. virtual memory)

Deadlock Prevention ②

- **Hold and Wait**
  - must guarantee that whenever a process requests a resource, it **does not hold** any other resources.
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.
  - Low resource utilization; starvation possible

Deadlock Prevention ③

- **No Preemption**
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
  - Only apply to resources whose state can be easily saved/restored (such as CPU, registers, memory)

Deadlock Prevention ④

- **Circular Wait**
  1. impose a total ordering of all resource types
  2. require that each process requests resources in an increasing order of enumeration.

     When request R_k, should release all R_i, $i \geq k$.

  - Correctness (proof by contradiction)
- **Force all processes to request resources in a particular order preventing any cyclic use of resources**
  - Thus, preventing deadlock
  - Example (x.P, y.P, z.P, …)

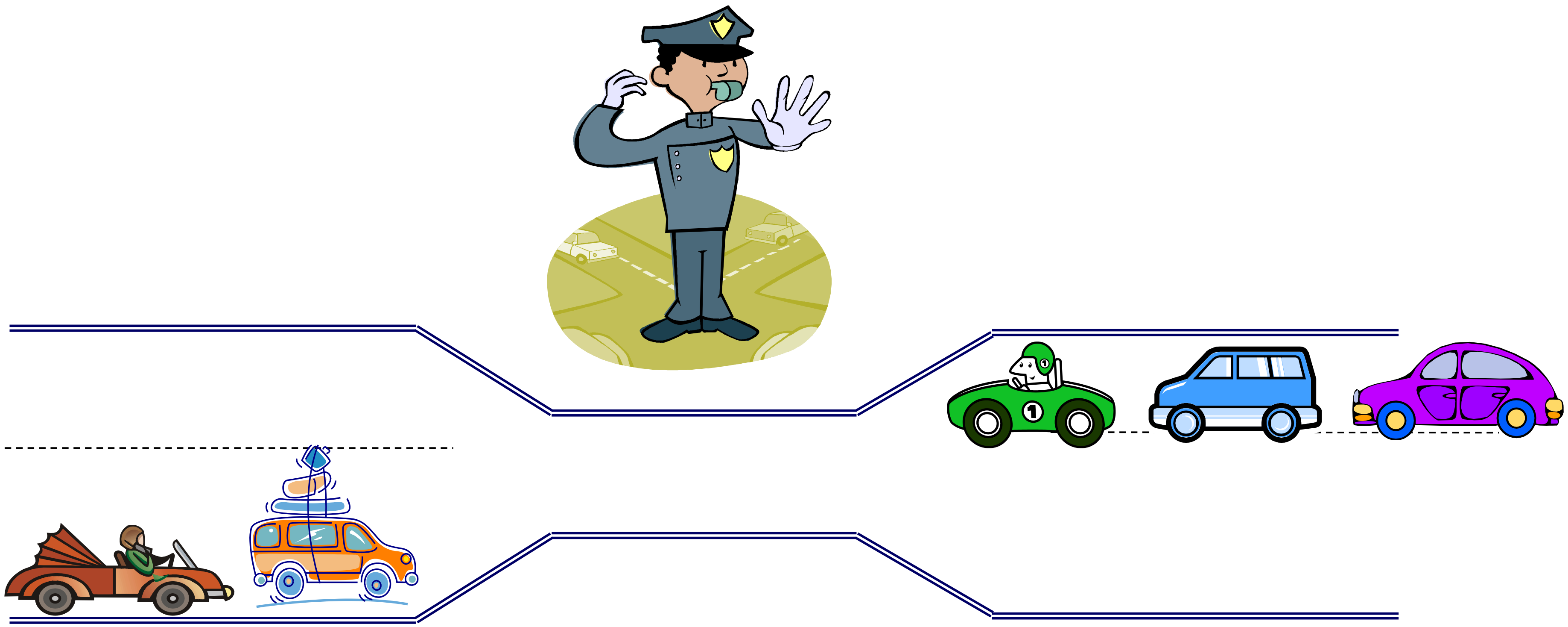
Deadlock Prevention

- Deadlock Prevention

  1. A set of methods for ensuring that at least one of the four necessary conditions cannot hold.
  2. These methods prevent deadlocks by constraining how requests for resources are made.

Deadlock Avoidance

Deadlock Avoidance



1 /4

3 /6

6 /9

Deadlock Avoidance

- Check for the given cases
- Require that the system has some additional a priori information available.

Deadlock Avoidance

- Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- The deadlock-avoidance algorithm **dynamically** examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation **state** is defined by the number of available and allocated resources, and the maximum demands of the processes.

Deadlock Avoidance

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a **safe state**.
- System is in **safe state** if there exists a **safe sequence** of all processes.

Deadlock Avoidance

- Sequence $<P_1, P_2, \ldots, P_n>$ is **safe** if for each P_i, the resources that P_i can still request, can be satisfied by currently available resources + resources held by all the P_j, with $j < i$.
- Formally, there is a **safe sequence** $<P_1, P_2, \ldots, P_n>$ such that for all $i = 1, 2, \ldots, n$,

$$Available + \Sigma_{1 \le j \le i}(Allocated_j) \ge MaxNeed_i$$

Deadlock Avoidance

北京交通大學

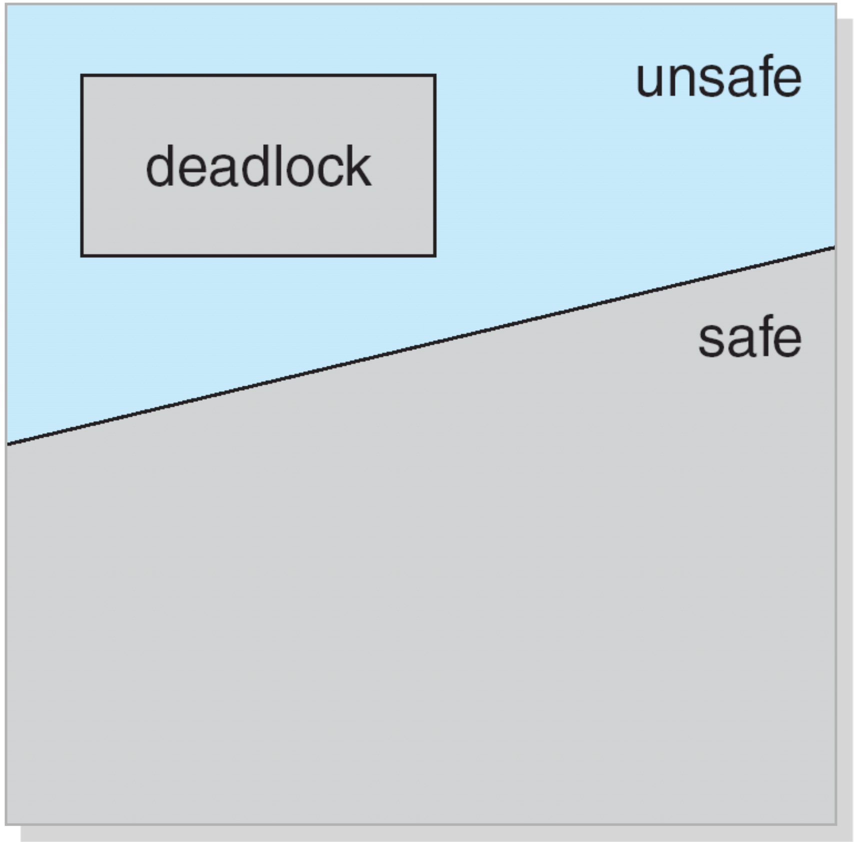
Available = 5		
	allocated	needed
P1	1	3
P2	2	6
P3	0	8
P4	5	1
P5	4	2
P6	7	3

Deadlock Avoidance

- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished (where $j < i$).
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Unsafe State

- No safe sequence exists
- **May** lead to a deadlock (not must)

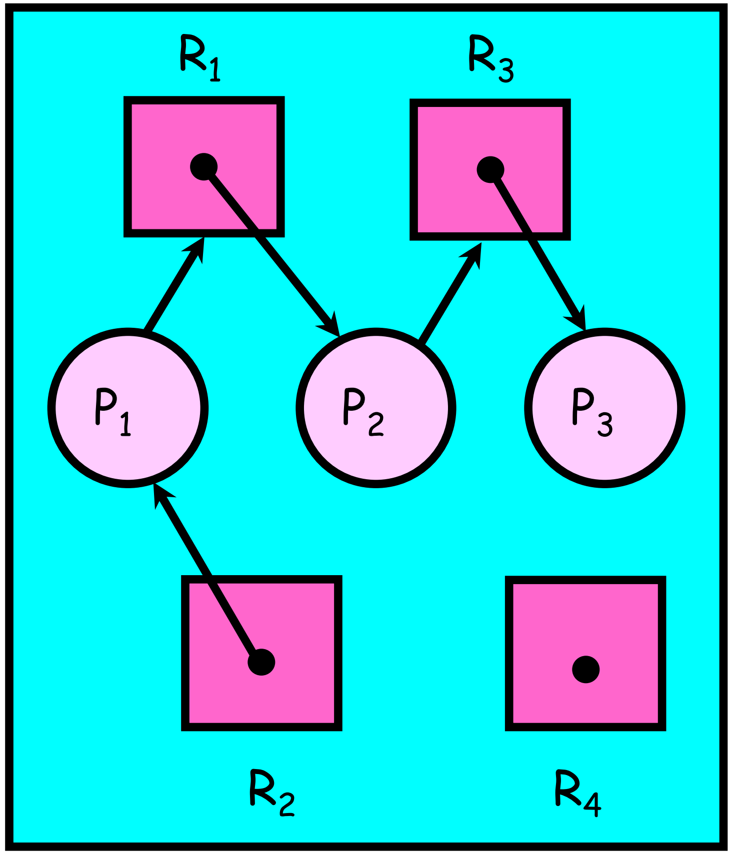


Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlocks
- Avoidance ⇒ ensure that a system will **never** enter an unsafe state

Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

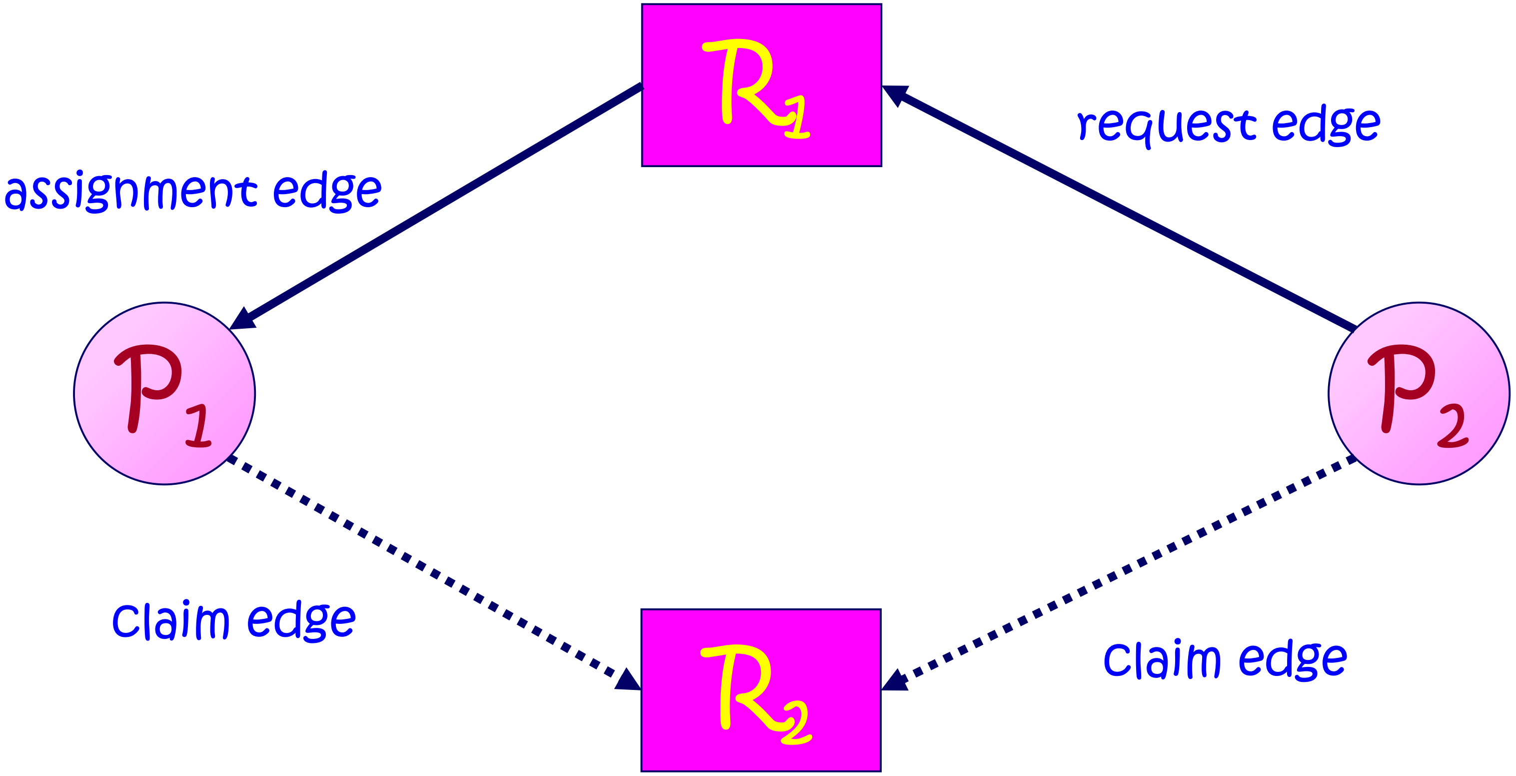


Resource Allocation Graph

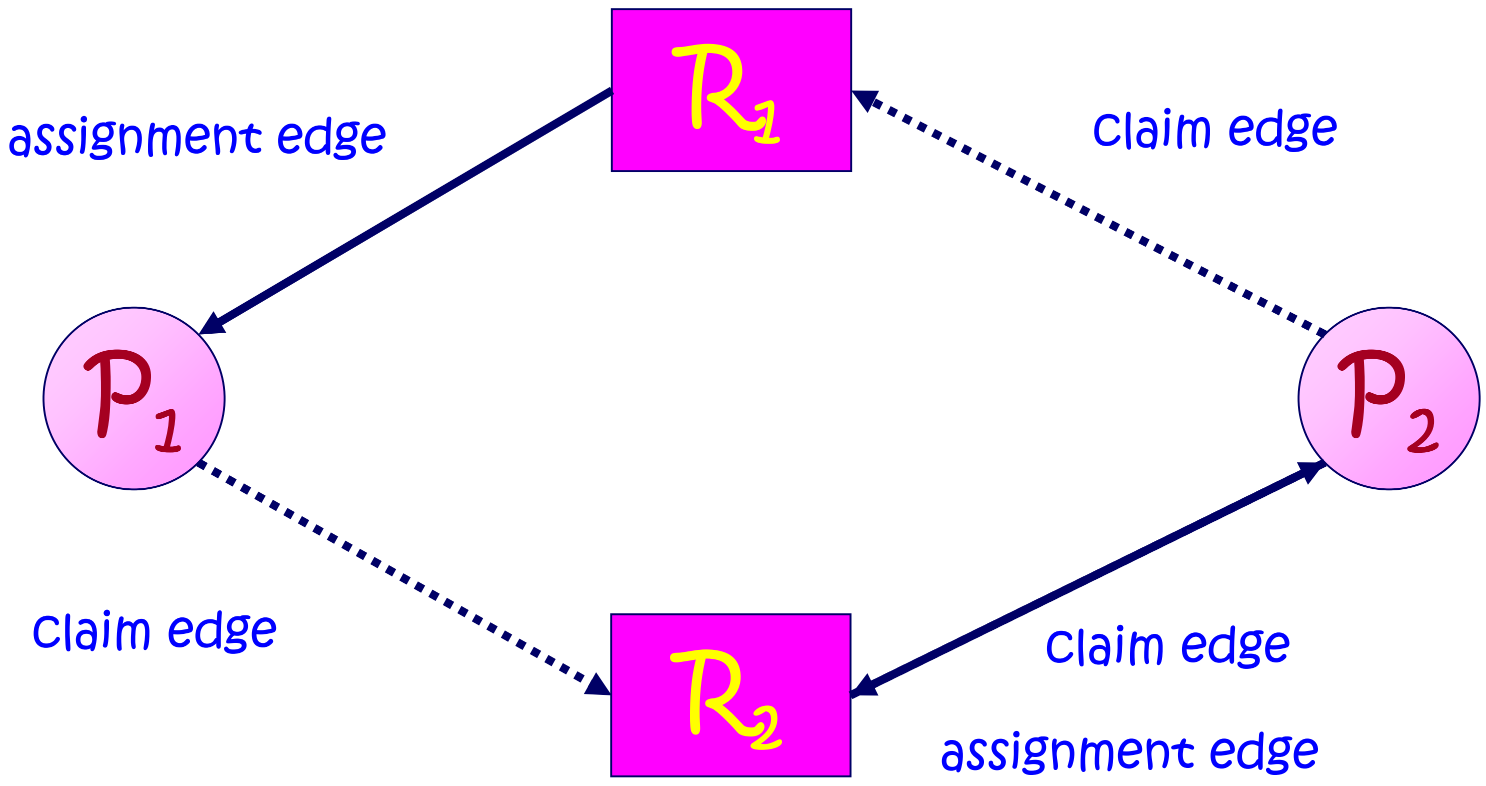
Resource-Allocation Graph Scheme

- Use a variant of resource-allocation graph
  - Claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j; represented by a dashed line.
  - Claim edge converts to request edge when a process requests a resource.
  - Request edge converted to assignment edge when the resource is allocated to the process.
  - When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed a priori in the system.
- Check for safety using a **cycle-detection algorithm**.

Resource-Allocation Graph Scheme



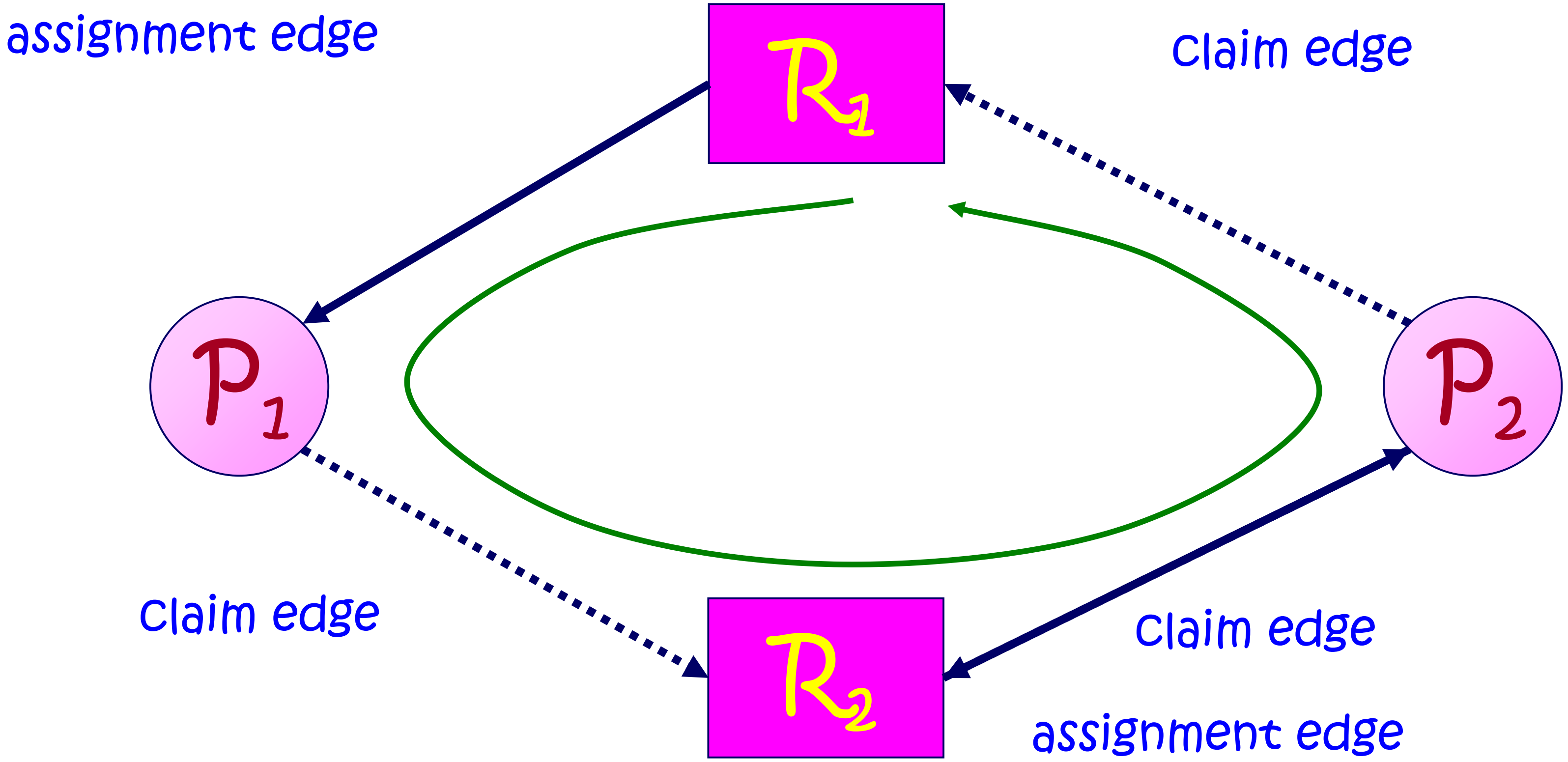
Unsafe State in Resource-Allocation Graph



Resource-Allocation Graph Algorithm

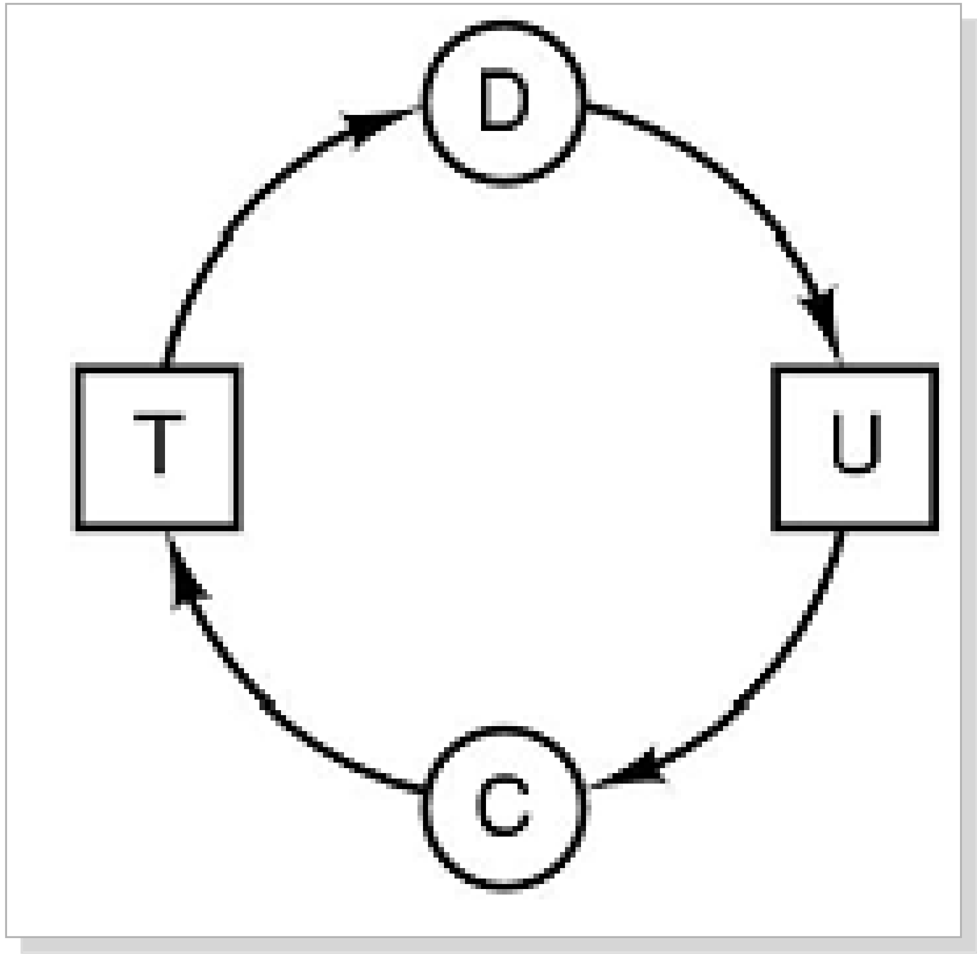
- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a **cycle** in the resource allocation graph

Unsafe State in Resource-Allocation Graph

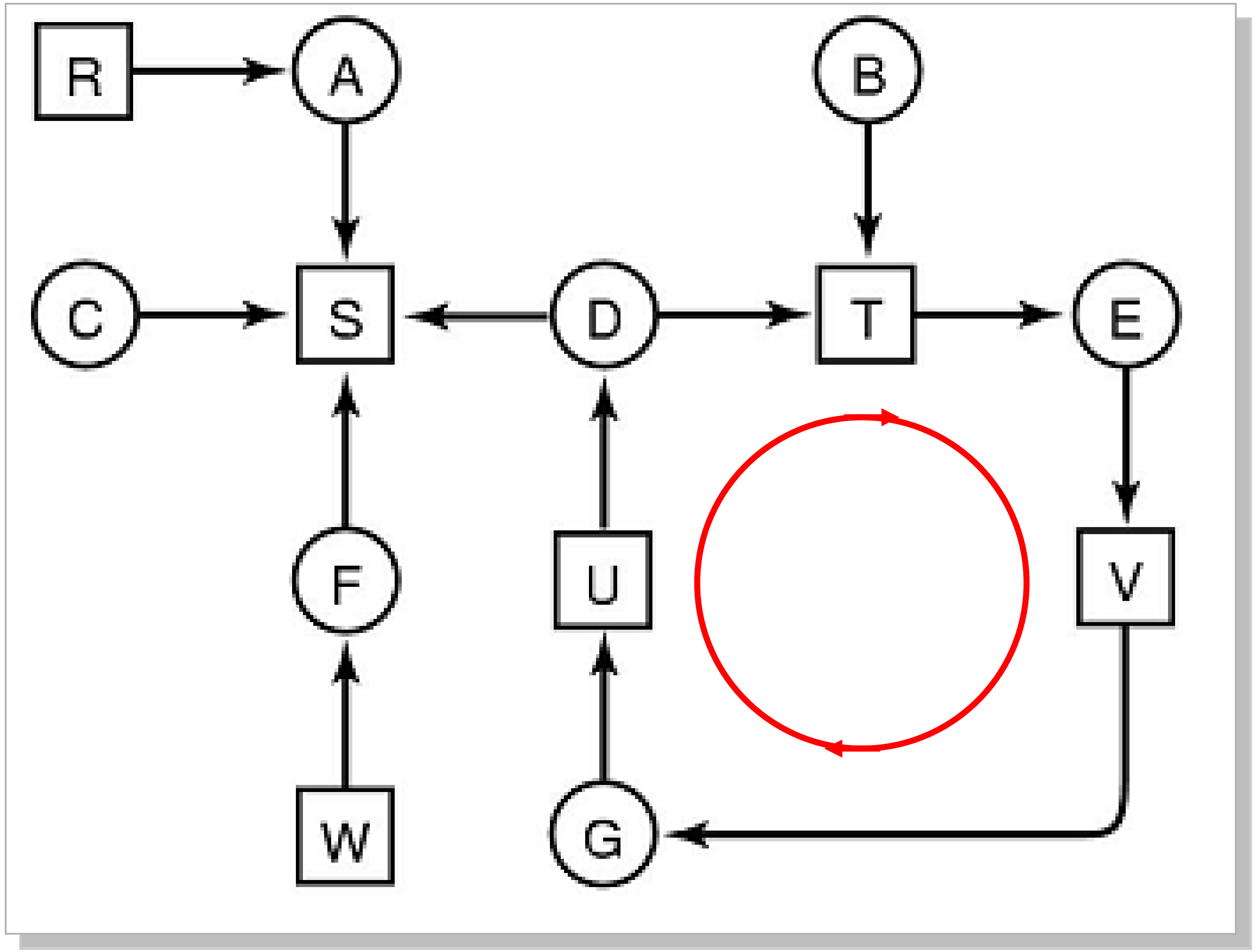


Cycle-Detection

- Only one of each type of resource ⇒ look for loops



Cycle-Detection



Avoidance algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

Banker's Algorithm

- Assumption
  - Each process must a priori claim maximum use
  - When a process requests a resource it may have to wait
  - When a process gets all its resources it must return them in a finite amount of time


1 /4
3 /6
6 /9

Banker's Algorithm for Avoiding Deadlock

- Toward right idea:
  - State maximum resource needs in advance
  - Allow particular process to proceed if:
    - (available resources - #requested) $\geq$ Max remaining that might be needed by any process

Banker's Algorithm for Avoiding Deadlock

- Banker's algorithm (less conservative):
  - Allocate resources dynamically
    - Evaluate each request and grant if some ordering of processes is still deadlock free afterward
    - Technique: pretend each request is granted, then run deadlock detection algorithm
    - Keeps system in a "SAFE" state

Data Structures for Banker's Algorithm

- Let
  - **n** = number of processes
  - **m** = number of resources types
  - **Available**: Vector of length **m**.

Available					
1	2	…	j	…	m

Available [j] = k —there are k instances of resource type R_j available.

Data Structures for Banker's Algorithm

- Let
  - **n** = number of processes
  - **m** = number of resources types
  - **Max**: **n × m** matrix.

Max					
1,1	…	…	1,j	…	1,m
…	…	…	…	…	…
i,1	…	…	i,j	…	i,m
…					
n					

If Max [i,j] = k — then process P_i may request at most k instances of resource type R_j.

Data Structures for Banker's Algorithm

- Let
  - **n** = number of processes
  - **m** = number of resources types
  - **Allocation**: **n** × **m** matrix

Allocation					
1,1	…	…	1,j	…	1,m
…	…	…	…	…	…
i,1	…	…	i,j	…	i,m
n					

If Allocation[i,j] = k — then P_i is currently allocated k instances of R_j.

Data Structures for Banker's Algorithm

- Let
  - **n** = number of processes
  - **m** = number of resources types
  - **Need**: **n × m** matrix

Need					
1,1	…	…	1,j	…	1,m
…	…	…	…	…	…
i,1	…	…	i,j	…	i,m

If Need[i,j] = k — then P_i may need k more instances of R_j to complete its task.

Need [i,j] = Max[i,j] – Allocation [i,j].

Data Structures for Banker's Algorithm

- Let $\mathbf{n}$ = number of processes, and $\mathbf{m}$ = number of resources types.
  - **Available**: Vector of length $\mathbf{m}$. If $\mathbf{available\ [j] = k}$, there are $\mathbf{k}$ instances of resource type $\mathbf{R_j}$ available.
  - **Max**: $\mathbf{n \times m}$ matrix. If $\mathbf{Max\ [i,j] = k}$, then process $\mathbf{P_i}$ may request at most $\mathbf{k}$ instances of resource type $\mathbf{R_j}$.
  - **Allocation**: $\mathbf{n \times m}$ matrix. If $\mathbf{Allocation[i,j] = k}$ then $\mathbf{P_i}$ is currently allocated $\mathbf{k}$ instances of $\mathbf{R_j}$.
  - **Need**: $\mathbf{n \times m}$ matrix. If $\mathbf{Need[i,j] = k}$, then $\mathbf{P_i}$ may need $\mathbf{k}$ more instances of $\mathbf{R_j}$ to complete its task.

$$Need\ [i,j] = Max[i,j] – Allocation\ [i,j].$$

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively. Initialize:

   (a) **Work = Available**

   (b) **Finish [i] = false** for **i** = 0, 1, …, **n**- 1.

2. Find an **i** such that both:

   (a) **Finish [i] = false**

   (b) $\mathbf{Need_i} \le \mathbf{Work}$

   If no such **i** exists, go to step 4.

3. $\mathbf{Work = Work + Allocation_i}$

   Finish[i] = true

   go to step 2.

4. If **Finish [i] == true** for all **i**, then the system is in a **safe** state.

Resource-Request Algorithm for Process P_i

- **$Request_i$** = request vector for process **P_i**.
- If **$Request_i [j] = k$** then process **P_i** wants **k** instances of resource type **R_j**.

1. If **$Request_i \leq Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If **$Request_i \leq Available$**, go to step 3. Otherwise **P_i** must wait, since resources are not available.
3. Pretend to allocate requested resources to **P_i** by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to **P_i**.
- If unsafe $\Rightarrow$ **P_i** must wait, and the old resource-allocation state is restored

Example of Banker's Algorithm

- Example
  - Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed.
  - Real systems, for example, would deal with much larger quantities of each resource.

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	5	1	0		False
P3	0	0	0	2		False
P4	0	7	5	0		False

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		False
P3	0	0	0	2		False
P4	0	7	5	0		False

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	4	0	6	5	6	1	5	2	0
P2	1	4	3	2	[illegible]	9	4	2				
P3	1	3	5	4	1	3	5	6				
P4	1	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	2		False
P4	0	7	5	0		False

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	4	0	6	5	6	1	5	2	0
P2	0	0	0	0	1	9	4	2				
P3	1	3	5	4	1	3	5	[illegible]				
P4	1	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	2		False
P4	0	7	5	0		False

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	2		False
P4	0	7	5	0		False

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	2		False
P4	0	7	5	0		False

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	4	0	6	5	6	2	9	5	2
P2	0	0	0	0	1	9	4	2	1	3	5	4
P3	0	0	0	0	1	3	5	6	3	12	10	6
P4	1	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	0		True
P4	0	7	5	0		False

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	4	0	6	5	6	3	12	10	6
P2	0	0	0	0	1	9	4	2				
P3	0	0	0	0	1	3	5	6				
P4	1	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	0		True
P4	0	7	5	0		False

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	1	4	0	6	5	6	4	12	10	6
P2	0	0	0	0	1	9	4	2				
P3	0	0	0	0	1	3	5	6				
P4	0	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	0	0	0		True
P3	0	0	0	0		True
P4	0	0	0	0		True

	Allocation				Max				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P1	0	0	0	0	0	6	5	6	4	12	11	10
P2	0	0	0	0	1	9	4	2				
P3	0	0	0	0	1	3	5	6				
P4	0	0	0	0	1	7	5	0				

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	0	0	0		True
P2	0	0	0	0		True
P3	0	0	0	0		True
P4	0	0	0	0		True

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	5	1	0		False
P3	0	0	0	2		False
P4	0	7	5	0		False

Safe State

Safety Algorithm

1. Let **Work** and **Finish** be vectors of length **m** and **n**, respectively. Initialize:

   (a) **Work = Available**

   (b) **Finish [i] = false** for **i** = 0, 1, …, **n**- 1.

2. Find an **i** such that both:

   (a) **Finish [i] = false**

   (b) $\mathbf{Need_i} \le \mathbf{Work}$

   If no such **i** exists, go to step 4.

3. $\mathbf{Work = Work + Allocation_i}$

   Finish[i] = true

   go to step 2.

4. If **Finish [i] == true** for all **i**, then the system is in a safe state.

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

	Need				FINISH
	A	B	C	D	
P1	0	6	4	2	False
P2	0	5	1	0	False
P3	0	0	0	2	False
P4	0	7	5	0	False

process 4 requests 4 unit of resource B.

There are enough resources

Assuming the request is granted, the new state would be:

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

	Need				=Max-Allocation	FINISH
	A	B	C	D		
P1	0	6	4	2		False
P2	0	5	1	0		False
P3	0	0	0	2		False
P4	0	3	5	0		False

Unsafe State

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

	Need				FINISH
	A	B	C	D	
P1	0	6	4	2	False
P2	0	5	1	0	False
P3	0	0	0	2	False
P4	0	7	5	0	False

process 1 requests 2 unit of resource B.

Resource-Request Algorithm for Process P_i

- **$Request_i$** = request vector for process **P_i**.
- If **$Request_i [j] = k$** then process **P_i** wants **k** instances of resource type **R_j**.

1. If **$Request_i \le Need_i$** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
2. If **$Request_i \le Available$**, go to step 3. Otherwise **P_i** must wait, since resources are not available.
3. Pretend to allocate requested resources to **P_i** by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

   - If safe $\Rightarrow$ the resources are allocated to **P_i**.
   - If unsafe $\Rightarrow$ **P_i** must wait, and the old resource-allocation state is restored

Banker's Algorithm

- The Banker's algorithm is run by the operating system whenever a process requests resources.
- The algorithm avoids deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

Banker's Algorithm

- For the Banker's algorithm to work, it needs to know three things:
  - How much of each resource each process could possibly request?
  - How much of each resource each process is currently holding?
  - How much of each resource the system has available?

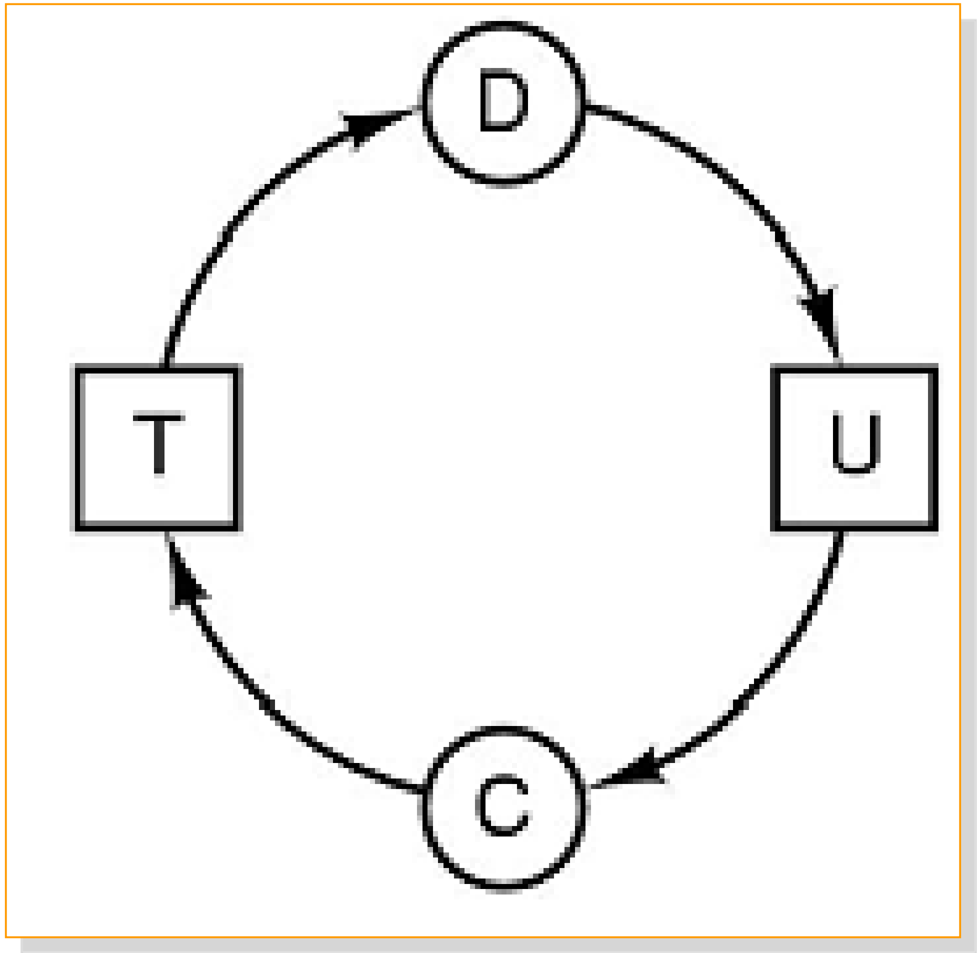
Deadlock Detection

Deadlock Detection

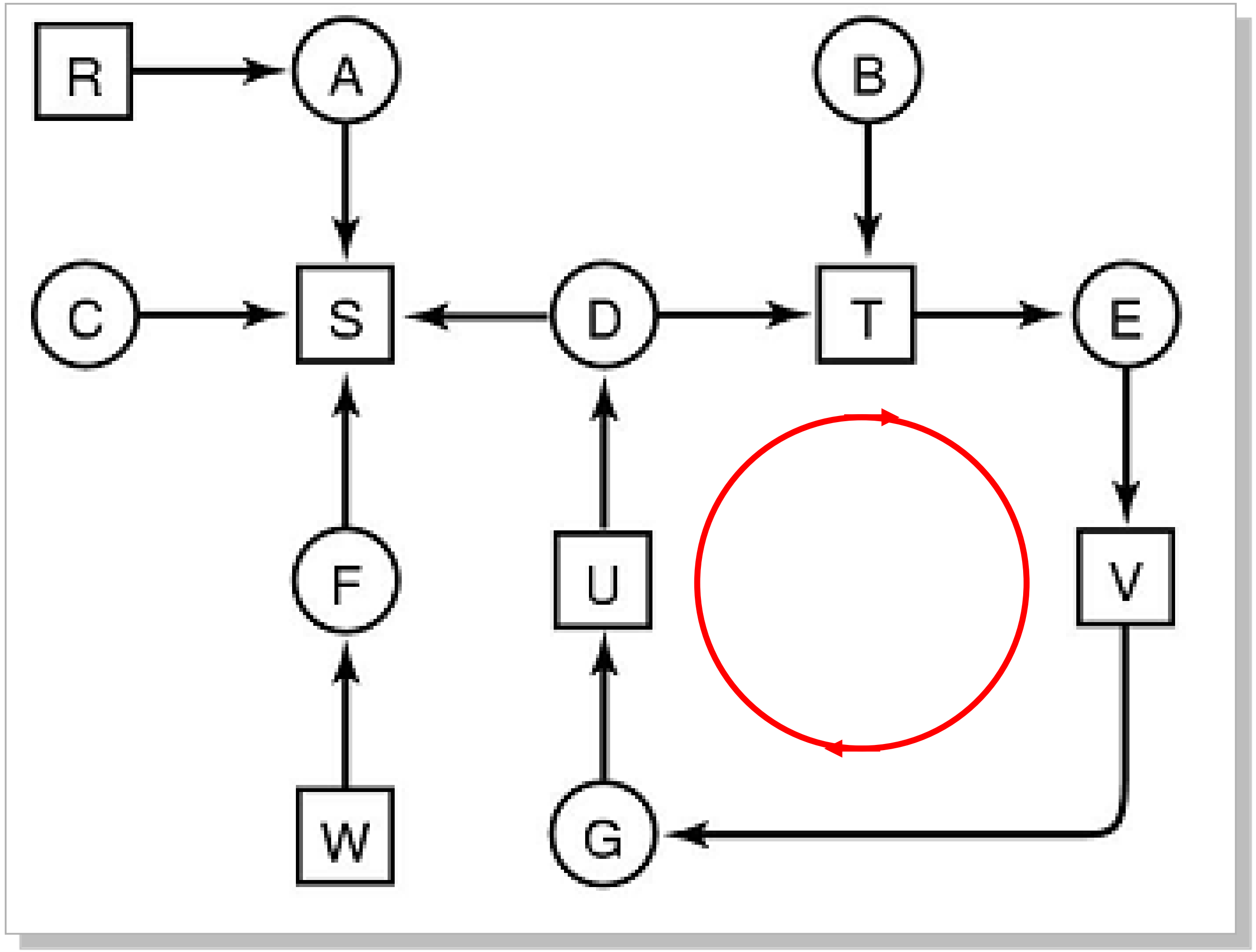
- Allow system to enter deadlock state
  - Detect whether deadlocks occur
  - If so, recover from the deadlock
- Detection algorithms
- Recovery scheme

Deadlock Detection Algorithm

- Single instance of each resource type ⇒ look for loops



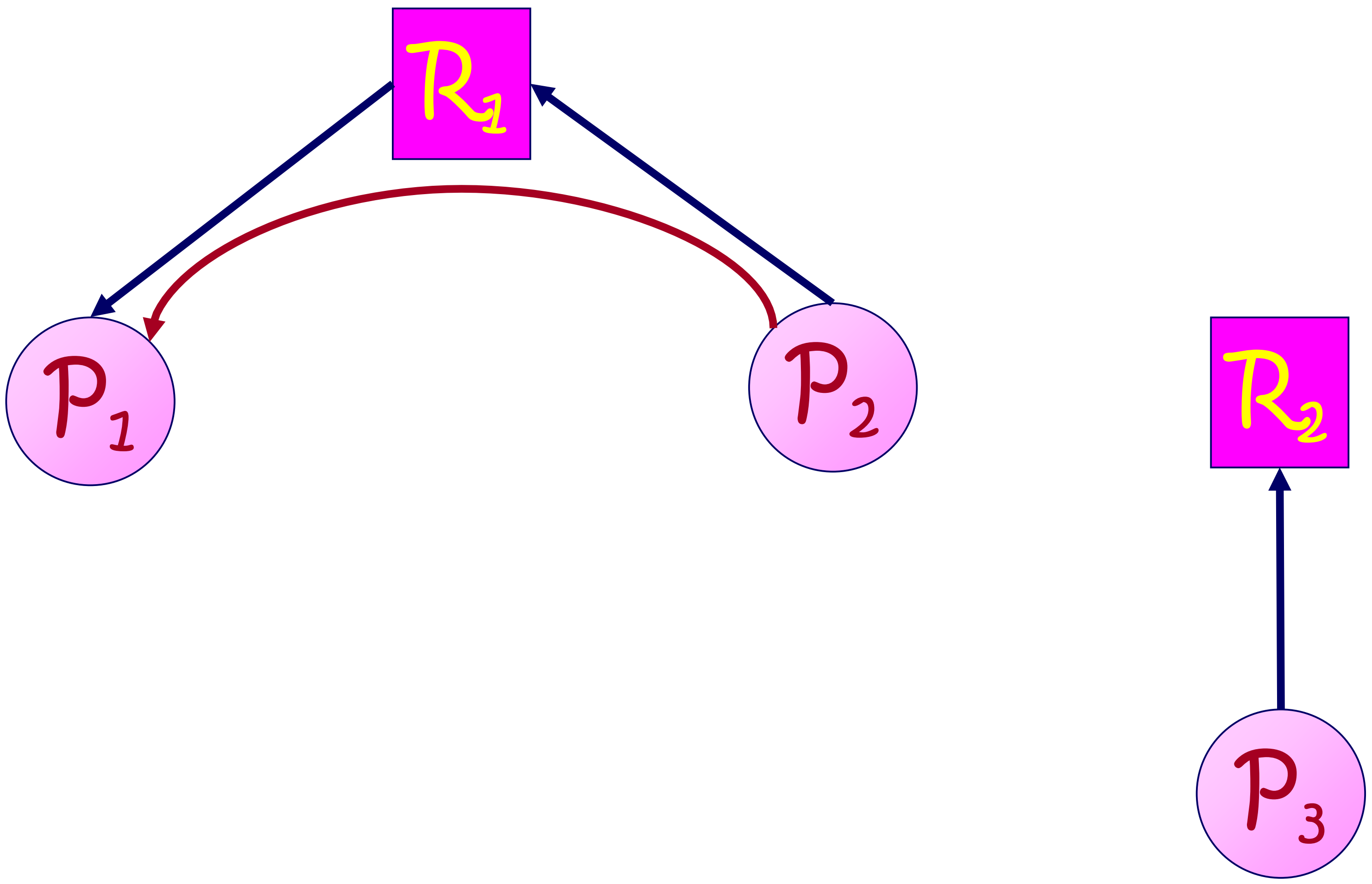
Single Instance of Each Resource Type



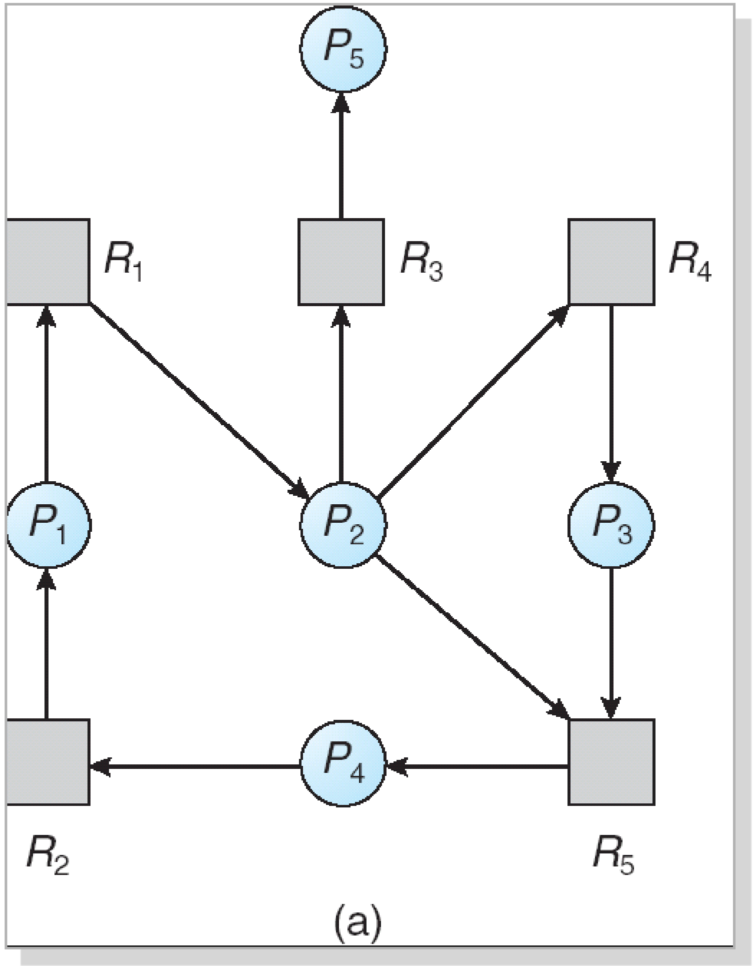
Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.

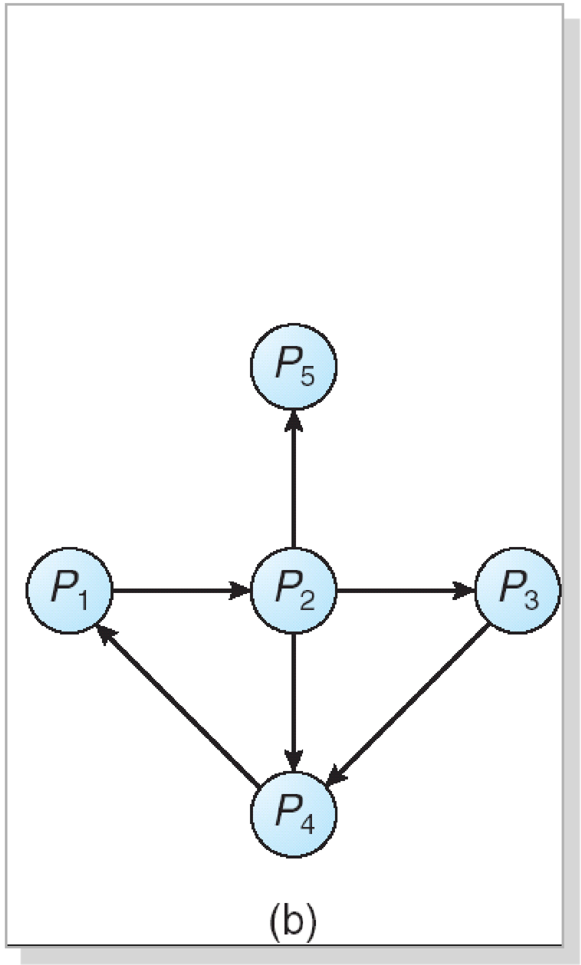
Wait-for Graph



Resource-Allocation Graph & Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Deadlock Detection Algorithm

- Single Instance of Each Resource Type

  ⇒ look for loops

- Several Instances of a Resource Type

  ⇒ ?

Several Instances of a Resource Type

- ***Available***: Vector of length $\boldsymbol{m}$ indicates the number of available resources of each type.
- ***Allocation***: $\boldsymbol{n} \times \boldsymbol{m}$ matrix defines the number of resources of each type currently allocated to each process.
- ***Request***: $\boldsymbol{n} \times \boldsymbol{m}$ matrix indicates the current request of each process. If $\boldsymbol{Request}[\,\boldsymbol{i}, \boldsymbol{j}\,] = \boldsymbol{k}$, then process $\boldsymbol{P_i}$ is requesting $\boldsymbol{k}$ more instances of resource type $\boldsymbol{R_j}$.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length $\boldsymbol{m}$ and $\boldsymbol{n}$, respectively. Initialize:
   - (a) $\boldsymbol{Work = Available}$
   - (b) For $\boldsymbol{i}$ = 1,2, …, $\boldsymbol{n}$, if $\boldsymbol{Allocation_i} \neq 0$, then $\boldsymbol{Finish[i] = false}$; otherwise, $\boldsymbol{Finish[i] = true}$.
2. Find an index $\boldsymbol{i}$ such that both:
   - (a) $\boldsymbol{Finish[i] == false}$
   - (b) $\boldsymbol{Request_i \leq Work}$
   - If no such $\boldsymbol{i}$ exists, go to step 4.
3. $\boldsymbol{Work = Work + Allocation_i}$

   $\boldsymbol{Finish[i] = true}$

   go to step 2.
4. If $\boldsymbol{Finish[i] == false}$, for some $\boldsymbol{i}$, $1 \leq \boldsymbol{i} \leq \boldsymbol{n}$, then the system is in deadlock state.

   Moreover, if $\boldsymbol{Finish[i] == false}$, then $\boldsymbol{P_i}$ is deadlocked.

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0							0	1	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0							3	1	3
P_1	2	0	0	2	0	2			
P_2									
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0							5	2	4
P_1	2	0	0	2	0	2			
P_2									
P_3									
P_4	0	0	2	0	0	2			

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:

		Allocation				Request				Available		
		A	B	C		A	B	C		A	B	C
P_0										5	2	6
P_1		2	0	0		2	0	2				
P_2												
P_3												
P_4												

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- Snapshot at time T_0:
- Sequence $<P_0, P_2, P_3, P_4, P_1>$ will result in Finish[i] = true for all *i*.

		Allocation				Request				Available		
		A	B	C		A	B	C		A	B	C
P_0										7	2	6
P_1												
P_2												
P_3												
P_4												

Example of Detection Algorithm

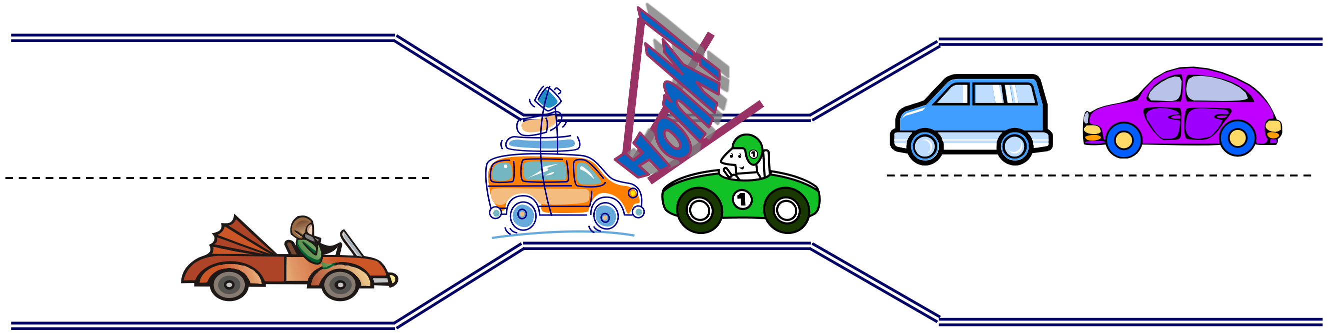
- P_2 requests an additional instance of type C.
- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	1			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Recovery From Deadlock

Methods for Handling Deadlock

- Method 1



Process Termination

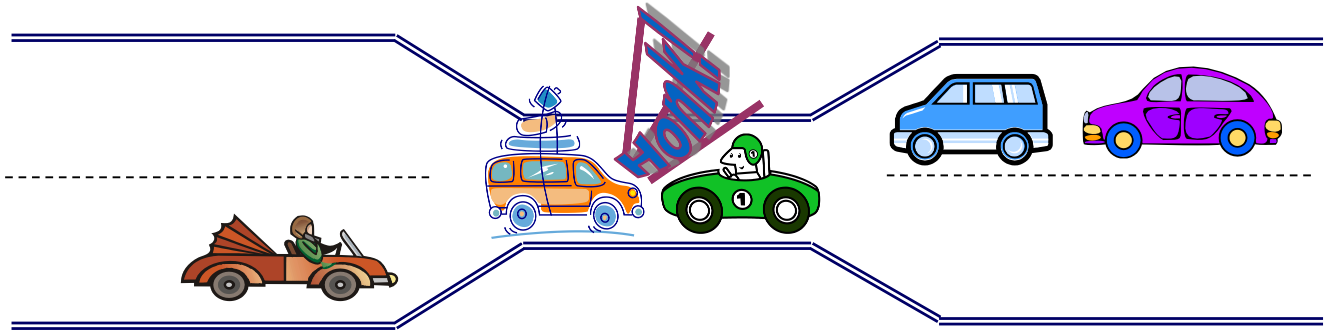
- Terminate process, force it to give up resources
  - In Bridge example, Godzilla picks up a car, hurls it into the river. Deadlock solved!
  - But, not always possible – killing a process holding a mutex leaves world inconsistent

Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Methods for Handling Deadlock

- Method 2



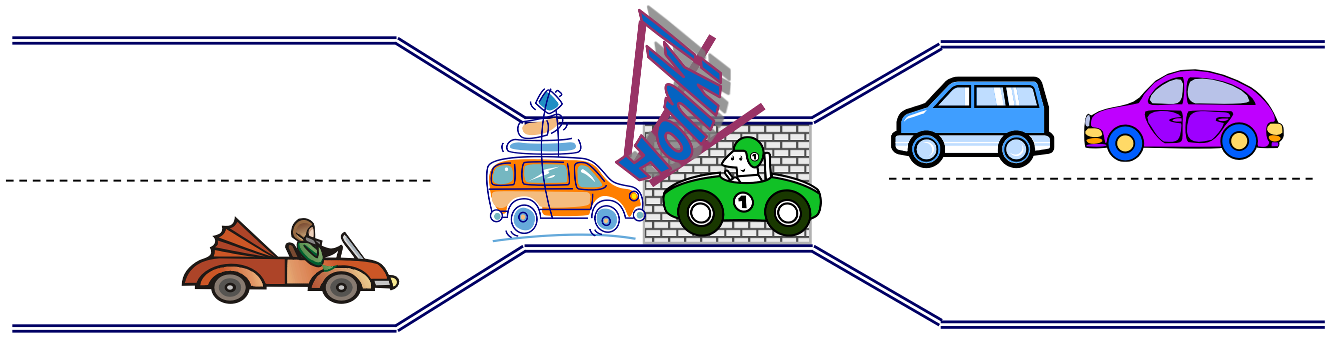
Roll Back

- Hit the rewind button on TiVo, pretend last few minutes never happened
- For bridge example, make one car roll backwards (may require others behind him)
- Return to some safe state, restart process for that state. (some resources are released)
  - totally rollback: simple but expensive
  - as far as necessary

    (OS should maintain "checkpoint" - a recording of the state of a process to allow rollback)
- Of course, if you restart in exactly the same way, may reenter deadlock once again

Methods for Handling Deadlock

- Method 3



Resource Preemption

- Preempt resources without killing off process
  - successively preempt some resources and give them to other processes until the deadlock cycle is broken.

Resource Preemption

- selecting a victim
  - How to minimize the cost?
    - what types and how many resources are held
    - how much time has run
    - how much time to end
    - ......
  - starvation
    - same process may always be picked as victim, include number of rollback in cost factor.
    - not always select the same process for preemption

What to do when detect deadlock?

- Terminate process, force it to give up resources

- Preempt resources without killing off process

- Roll back actions of deadlocked processes

- Many operating systems use other options

Summary

Summary

- **Starvation vs. Deadlock**
  - Starvation: processes waits indefinitely
  - Deadlock: circular waiting for resources
  - Deadlock $\Rightarrow$ Starvation, but not other way around
- **Four conditions for deadlocks**
  - Mutual exclusion
    - Only one process at a time can use a resource
  - Hold and wait
    - Process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption
    - Resources are released only voluntarily by the processes
  - Circular wait
    - $\exists$ set $\{T_1, \ldots, T_n\}$ of processes with a cyclic waiting pattern

Summary

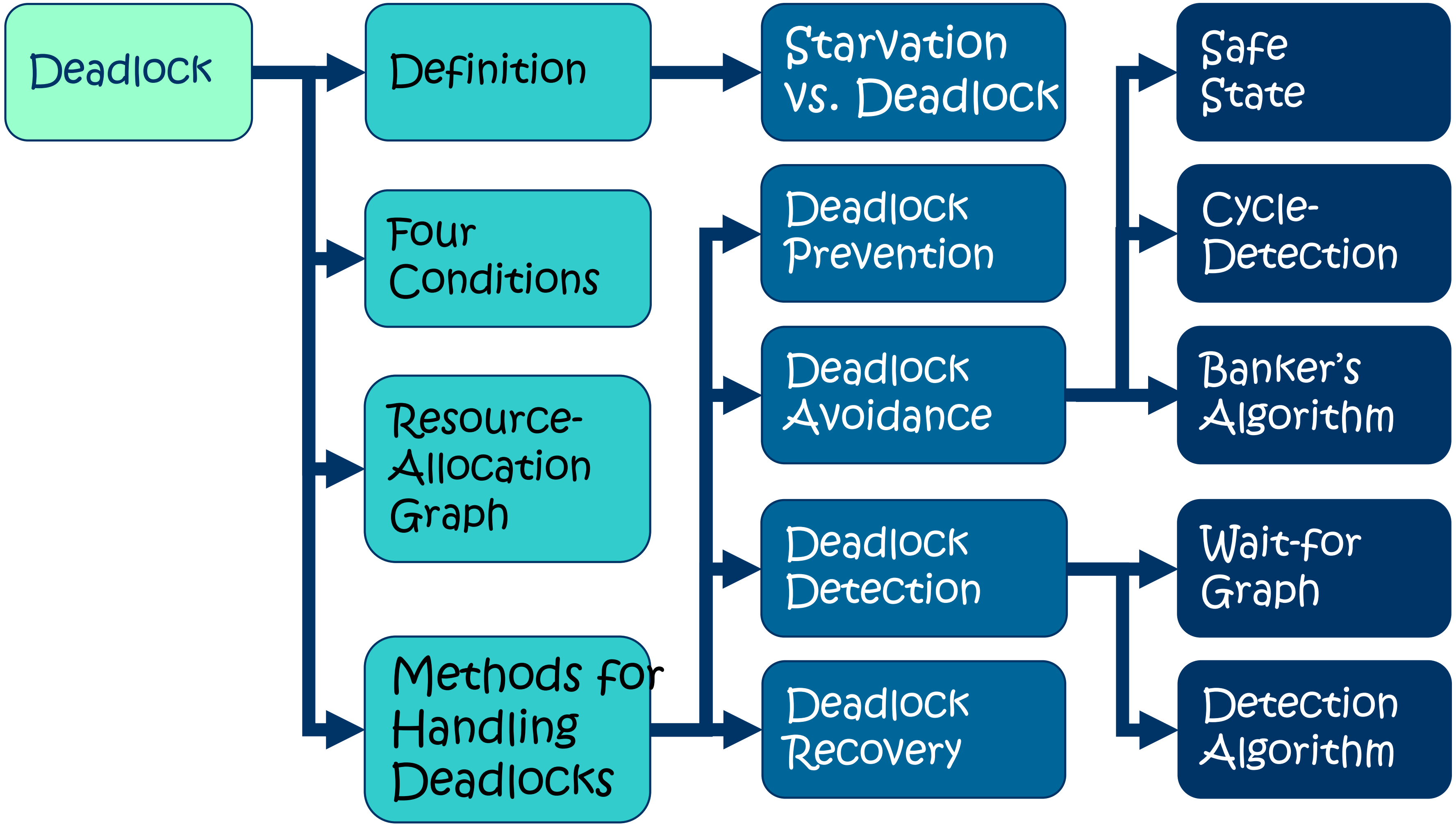
- Ensure that system will **never** enter a deadlock
  - deadlock prevention
  - deadlock avoidance
    - Banker's Algorithm
- Allow system to enter deadlock and then recover
  - deadlock detection
  - deadlock recovery
- Ignore the problem and **pretend** that deadlocks never occur in the system
  - Used by most operating systems, including UNIX and Windows

Summary

- Combine the three basic approaches
  - prevention
  - avoidance
  - detection

Allowing the use of the optimal approach for each of resources in the system.

Deadlock Handling



Thank you !

Q & A