Operating systems

Lecture 5:
Deadlocks

System Model

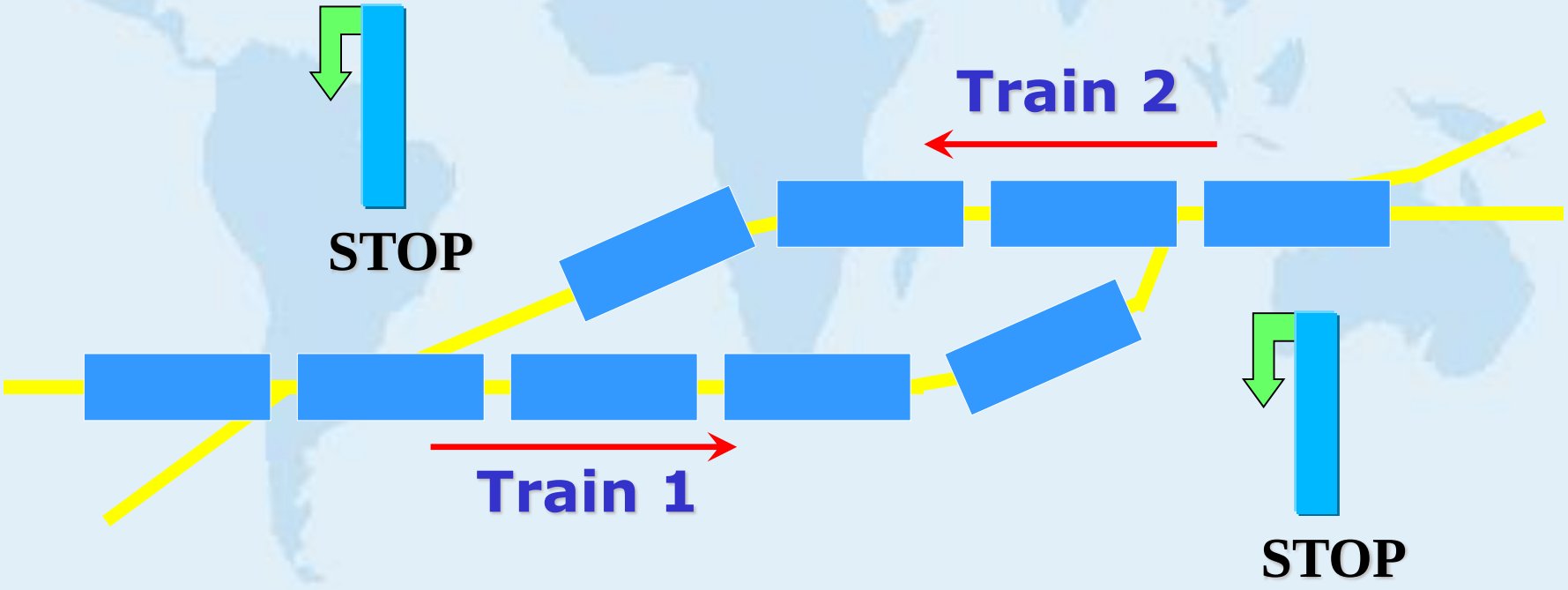
- Computer system consists of:
  - Number of **Processes** (programs in execution);
  - Finite number of **Resources** (CPU, memory space, files, I/O devices, etc.).
- Processes **compete** for resources.
  - If resource is not available – process waits.
  - **Deadlock** – when a process waits for resource held by another waiting process.

System Model

- A process use a resource in the following sequence:
  - **Request:** A process must request a resource before using it. If the resource is not available, the process enters waiting state.
  - **Use:** The process holds and operates on the resource.
  - **Release:** The process must release the resource after using it.
- The request and release are **System calls.**

Deadlock example

A deadlock state occurs when **two or more** processes are waiting for a event that can be caused **only** by one of the waiting processes.



Necessary Conditions

A **deadlock** situation can arise if the following four conditions hold **simultaneously** in a system:

1. **Mutual exclusion:**

   At least one resource must be held in a non-sharable mode, i.e. only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

Necessary Conditions

2. **Hold and wait:**

   There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.

3. **No preemption:**

   Resources cannot be preempted; that is, a resource can be released only **voluntary** by the process holding it, after that process has completed its task.

Necessary Conditions

4. **Circular wait:**

   There must exist a set $\{P_0, P_1, .., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2 ,…, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

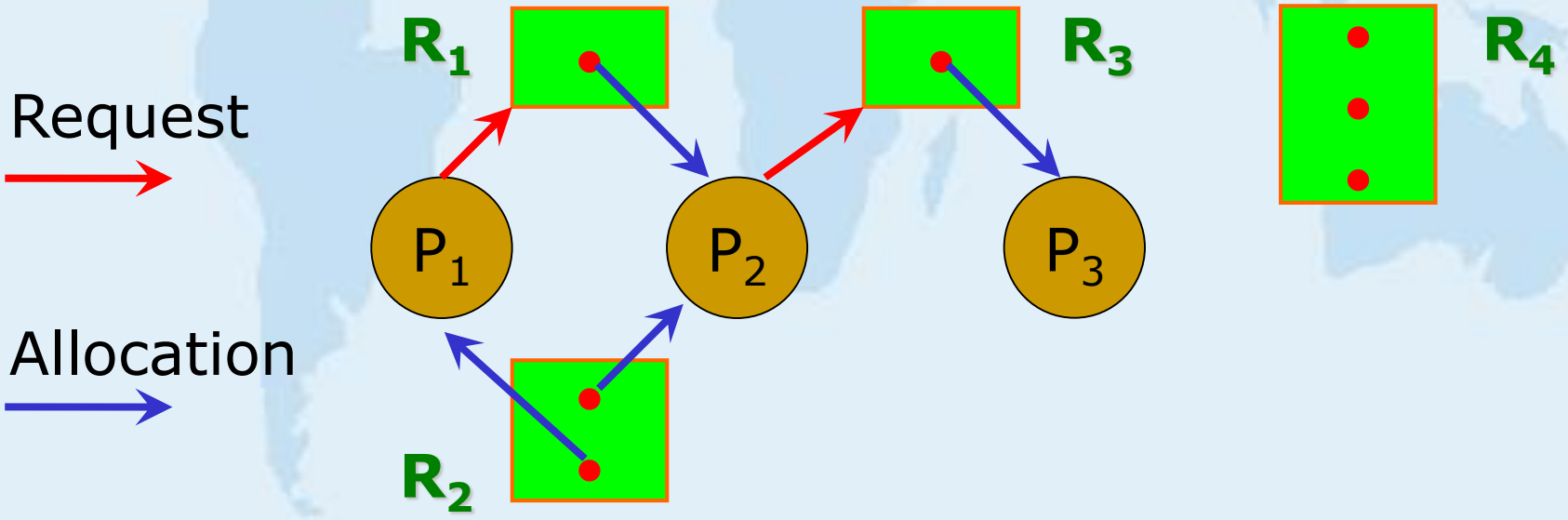
For a deadlock to occur, all the four conditions must hold.

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph, called a system **resource-allocation graph.**
- Nodes represent **processes** $P = \{P_1, P_2, \ldots, P_n\}$ and **resource types** $R = \{R_1, R_2, \ldots, R_m\}$.
- Directed edge from process P_i to resource R_j is called **request edge**.
- Directed edge from resource R_k to process P_l is called **assignment edge**.

Resource-Allocation Graph

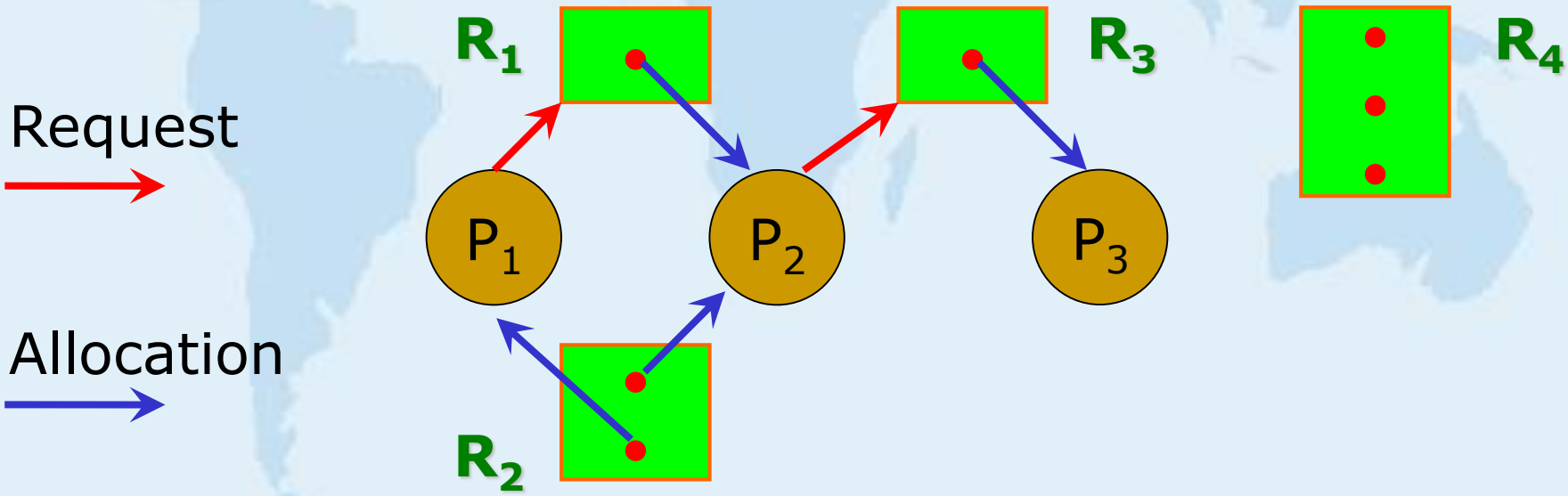
- Processes are denoted as **circles** and resources are denoted as **squares**.
- Each **instance** of a resource is denoted as a dot in the square.



Resource-Allocation Graph

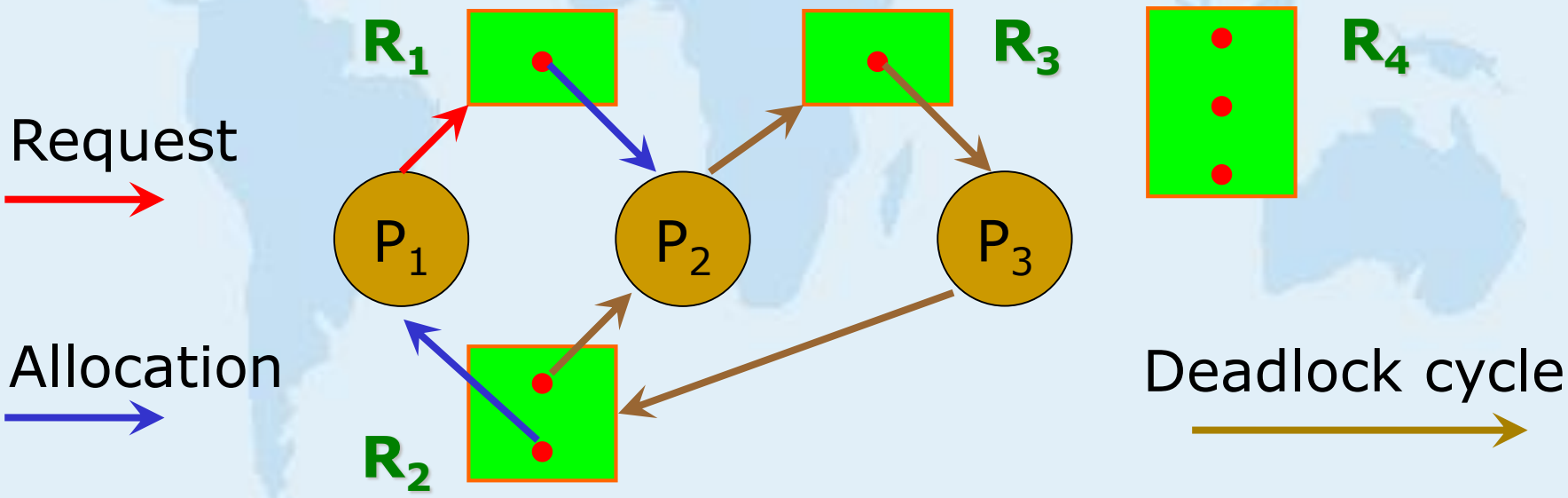
- **P** = {P_1, P_2, P_3} **R** = {R_1, R_2, R_3, R_4}

  E = {$P_1 \rightarrow R_1$, $P_2 \rightarrow R_3$, $R_1 \rightarrow P_2$, $R_2 \rightarrow P_2$, $R_2 \rightarrow P_1$, $R_3 \rightarrow P_3$}
- R_1 has 1 instance, R_2 has 2, R_3 has 1, and R_4 has 3.
- Process P_1 and P_2 are in **waiting** state.



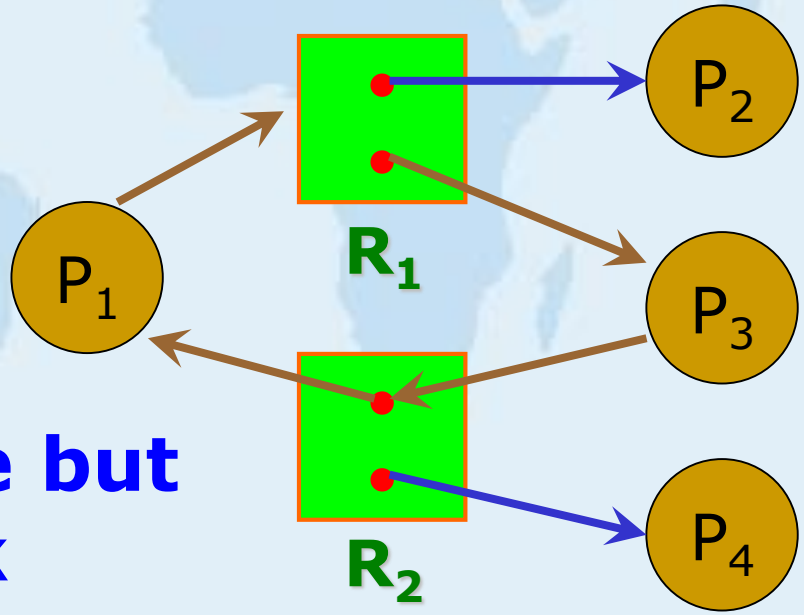
Resource-Allocation Graph

- If the graph has no cycle, no process is deadlocked. If the graph has a **cycle**, deadlock may exist. If each resource type involved in a cycle has only **one instance**, the cycle implies a deadlock:



Resource-Allocation Graph

- If each resource type involved in a cycle has **more** than one instance, the deadlock may or may not exist:



With a cycle but no deadlock

Deadlock Handling Methods

Principally, there are three methods for dealing with the deadlock problem:

1. We can use a **protocol** to ensure that the system will never enter a deadlock state.
2. We can allow the system to **enter** a deadlock state and then **recover**.
3. We can **ignore** the problem all together, and pretend that deadlocks never occur in the system. (UNIX and many other OS.)

Deadlock Handling Methods

- To ensure deadlocks do not occur, deadlock **prevention** or deadlock **avoidance** schemes can be used.

1. **Deadlock prevention** ensures that at least one of the four necessary conditions for deadlock **does not hold**.

2. **Deadlock avoidance** requires information on resources that a process will use to be known **in advance** .

Deadlock Prevention

It is necessary to **deny** one of the following:

1. **Mutual exclusion:** In general cannot be denied because some resources are intrinsically non-sharable (e.g. printer).
2. **Hold and wait:** Whenever process requests a resource, it should not hold any other resources.
   - Process requests and gets **all** the resources before it begins execution.
     - Leads to **lower** resource utilization and starvation.
   - Process requests resources **only** when it has none.
     - Waiting time **increases**, **low** process turnaround time.

Deadlock Prevention

3. **No preemption:** Allow preemption of all resources held by processes which are waiting for other resources.
4. **Circular wait:** Define a total ordering of all the resources and require that each process requests resources in an increasing order of enumeration.

   Assume $T(R_i)$ is the order of R_i. After a process requests R_i, it can request another R_j only if $T(R_j) > T(R_i)$.

Deadlock Prevention

Disadvantages of the Deadlock Prevention:

- **Restrains** how requests can be made.
- **Low** device utilization.
- **Reduced** system throughput.

Deadlock Avoidance

- Methods for deadlock avoidance require **additional** information about how resources are going to be requested.
- The simplest model requires that each process declare the **maximum number** of resources that it may need.
- A **resource-allocation state** is defined by the number of **available, allocated**, and **maximum requested resources**.

Deadlock Avoidance

- System is in **safe state** if there exists **safe sequence** of processes.
- A sequence of a processes $<P_1, P_2, \ldots, P_n>$ is a **safe sequence** for the current resource-allocation state if, for each P_i, the resources that P_i can still request can be satisfied by currently available resources plus the resources held by all the P_j , with $j < i$.
- If no safe sequence exists, system is in **unsafe state**.

Deadlock Avoidance

- If resources that P_i needs are not immediately available, then P_i can **wait** until all P_j have finished.
- When all P_j finish and release their resources, P_i can obtain all of its needed resources, complete task, release them, and terminate.
- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

Safe State - Example

(One resource class only)

- Total resources: **12**
- Unallocated: **2**

process	currently holds	max needed
A	4	6
B	4	11
C	2	7

- Safe sequence: **A,C,B**

Unsafe State – Ex.

(One resource class only)

- Total resources: **12**
- Unallocated: **2**

process	currently holds	max needed
A	4	6
B	4	11
C	2	9

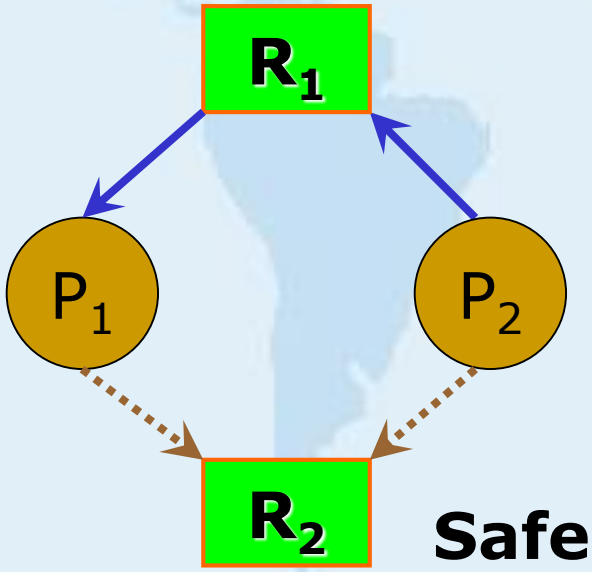
- **No** safe sequence exists!

Resource-Allocation Graph

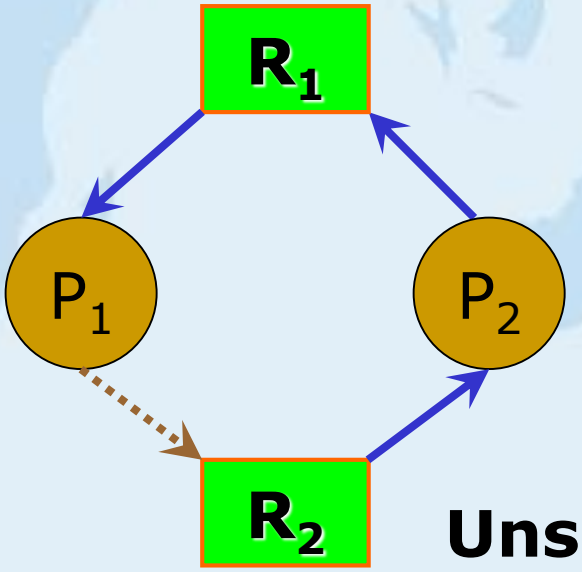
- Introduce new type of edge – **Claim edge.**
  - Claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j in the future.
  - Claim edge $P_i \rightarrow R_j$ is represented by a dashed line.
  - A claim edge $P_i \rightarrow R_j$ can be added to the graph only if **all** the edges associated to P_i are claim edges.
- When P_i requests R_j , the **claim edge** is converted to a **request edge**.

Resource-Allocation Graph

- **Deadlock** can be avoided if converting a request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ **does not** produce a cycle in the graph.



Safe state



Unsafe state

Deadlock Avoidance - Banker's Algorithm

- The resource-allocation graph algorithm is not applicable to a resource-allocation system with multiple instances for each resource type.
- For this case, we will use the **banker's algorithm**.
- This algorithm could be used in a banking system to ensure that the bank never allocates its available cash such that it can no longer satisfy the needs of all its customers.

Deadlock Avoidance - Banker's Algorithm

- When a new process enters the system, it must **declare** the maximum number of each resource type that it may need. This number may not exceed the total number of resources.
- When a user requests a set of resources, the system must **determine** whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must **wait** until some other process releases enough resources.

Deadlock Avoidance: Banker's Algorithm

- Let *n* be the number of processes in the system and *m* be the number of resource types. Define following:
- **Available**: *Avail [1…m]. Avail [j] = k* means k instances of R_j are available.
- **Max**: *Max [1..n,1..m]. Max [i, j] = k* means P_i requests at most *k* instances of R_j.
- **Allocation**: *Alloc [1..n,1..m]. Alloc [i, j] = k* means P_i holds k instances of R_j.
- **Need**: *Need [1..n,1..m]. Need [i, j] = k* means P_i may need *k* additional instances of R_j .

Deadlock Avoidance: Banker's Algorithm

- **Request**: *Req [1..n,1..m]. Req [i, j] = k* means P_i wants *k* instances of resource type R_j .
- Note that *Need [i, j] = Max [i, j] – Allo [i, j].*
- In addition:
  - For *X[1..n]* and *Y[1..n]*, *X ≤ Y* if *X[i] ≤ Y[i]*, *1 ≤ i ≤ n.*
  - *X < Y* if *X ≤ Y* and *X ≠ Y*.
- The *i*-th row *of Max*, *Alloc*, *Need* is denoted as Max_i , $Alloc_i$, $Need_i$, respectively.

Safe state checking algorithm

1. **Initialization**

   Work[1..m] = Avail [1..m]; *Finish [i] = false, 1 ≤ i ≤ n*.

2. **Find** *i* such that

   Finish [i] = false and $Need_i \leq Work$

   then go to step 3 else go to step 4;

3. **Execute**

   $Work = Work + Allo_i$; *Finish[i] = true*;

   go to step 2;

4. **Finish**

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

Resource-Request Algorithm

When a request for resources is made by process P_i :

1. **If** $Req_i \leq Need_i$ then go to step 2 else raise error (since the process has exceeded is maximum claim).
2. **If** $Req_i \leq Avail$ then go to step 3 else P_i must wait (since the resources are not available).
3. **Modify** state as follows:

   $Avail = Avail - Req_i$

   $Alloc_i = Alloc_i + Req_i$

   $Need_i = Need_i - Req_i$
4. **Check** if state is safe (call previous algorithm) and then allocate resources to P_i

Banker's Algorithm: An Example

- Five processes P_0, P_1, P_2, P_3, P_4 . Three types of resources **A** (10 instances), **B** (5 instances), **C** (7 instances). At time T_0 we have:

	Avail			Alloc			Max			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
	3	3	2									
P_0				0	1	0	7	5	3	7	4	3
P_1				2	0	0	3	2	2	1	2	2
P_2				3	0	2	9	0	2	6	0	0
P_3				2	1	1	2	2	2	0	1	1
P_4				0	0	2	4	3	3	4	3	1

Safe state sequence:
(by the algorithm)
P_1, P_3, P_4, P_2, P_0

Banker's Algorithm: An Example

- Assume P_1 gives additional request $Req_1 = (1,0,2)$. We first check $Req_1 \leq Avail$: $(1,0,2) \leq (3,3,2)$. The system enters following state:

	Avail			Alloc			Max			Need		
	A	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
	2	**3**	**0**									
P_0				0	1	0	7	5	3	7	4	3
P_1				**3**	**0**	**2**	3	2	2	**0**	**2**	**0**
P_2				3	0	2	9	0	2	6	0	0
P_3				2	1	1	2	2	2	0	1	1
P_4				0	0	2	4	3	3	4	3	1

Safe state sequence: (by the algorithm)
P_1, P_3, P_4, P_2, P_0

Banker's Algorithm: An Example

- By the safety algorithm, P_1, P_3, P_4, P_2, P_0 is a safe sequence and Req_1 **can** be granted.
- If P_4 requests (3,3,0) at this state, the request **cannot** be granted since the resources are not available.
- If P_0 requests (0,2,0), even though the resources are available, the granting will make the system **unsafe**.

Deadlock Detection

If a system **doesn't** have neither a deadlock-prevention nor a deadlock-avoidance, then a deadlock **may** occur. In this case, the system **must provide**:

- An algorithm that examines the state of the system to **determine** whether a deadlock has occurred.
- An algorithm to **recover** from a deadlock.

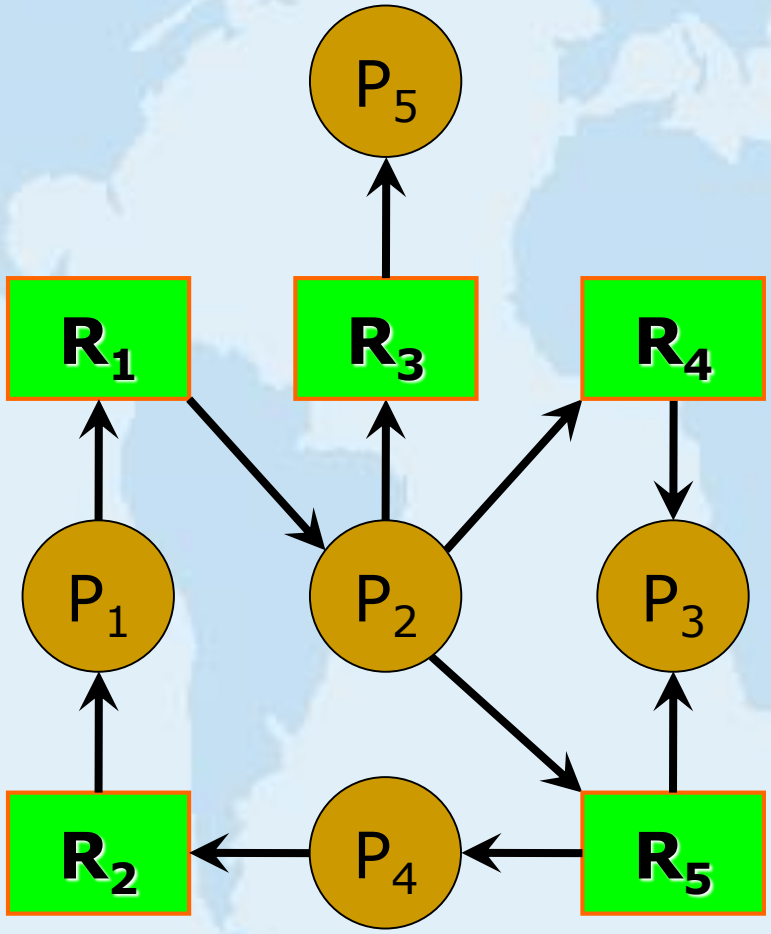
Deadlock Detection: Single Instance Case

When **all** resources have only a **single** instance:

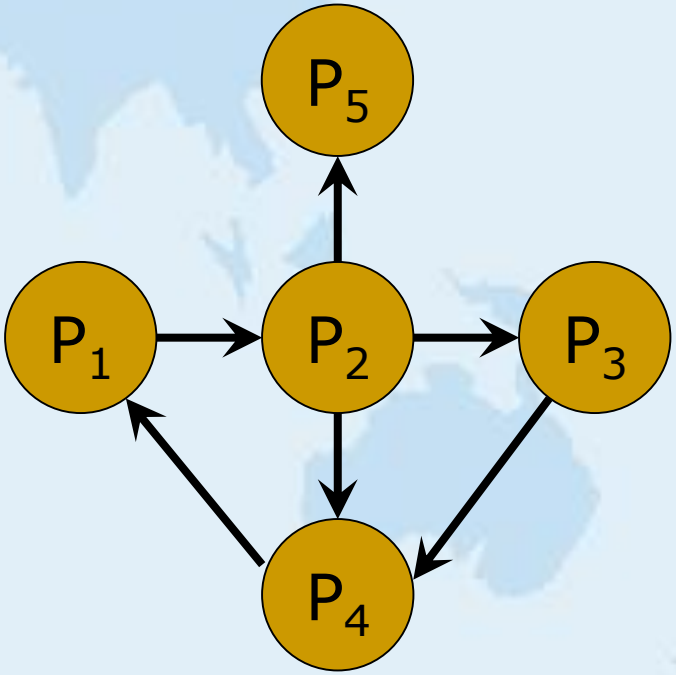
- Define a deadlock detection algorithm that uses a variant of the **resource-allocation** graph, called a **wait-for** graph.
- This graph is obtained from the resource-allocation graph by **removing** the resource nodes and collapsing the appropriate edges.

Deadlock Detection: Single Instance Case

Resource-allocation graph



Corresponding wait-for graph



Deadlock Detection: Single Instance Case

- A deadlock **exists** in the system if and only if the **wait-for** graph contains a cycle.
- To detect deadlocks, the system needs to maintain the **wait-for** graph and periodically to invoke an algorithm that **searches** for a cycle in the graph.
- An algorithm requires $O(n^2)$ operations, where n is the number of vertices in the graph.

Deadlock Recovery: Process Termination

To recover from deadlock we can **terminate**:

- **All deadlocked processes -** Simple method, high expense.
- **Abort one process at a time until the deadlock cycle eliminated -** Large overhead, since, after each process is aborted, **deadlock-detection** algorithm must be invoked to determine whether any process are still deadlocked.

Deadlock Recovery: Process Termination

To **choose** which process to abort, the following should be considered:

- The **priority** of the process.
- The **time** the process has executed and will be executed.
- How many and what type resources are **used.**
- Resources **needed** further.
- **Number** of processes needed to be aborted.
- Whether the process is **interactive** or **batch**.

Deadlock Recovery: Resource Preemption

To eliminate deadlocks we can use **resource preemption**:

- Successively **preempt** some resources from processes and **give** these resources to other processes until the deadlock cycle is broken.
- If preemption is required to deal with deadlocks, then some **issues** need to be addressed.

Deadlock Recovery: Resource Preemption

1. **Selecting a victim** - Which resources and which processes are to be preempted? Decide the order of preemption.
2. **Rollback** - If we preempt a resource from a process, we must roll back the process to some safe state, and restart it.
3. **Starvation** - How to prevent starvation? That is, how we guarantee that resources will not always be preempted from the same process?

Deadlock: Combined Approach

To handle deadlock situations **combinations** of the following approaches can be used:

- **Deadlock-prevention**,
- **Deadlock-avoidance**, and
- **Detection-recovery**.

Deadlock: Summary

- A deadlock state occurs when two or more processes are waiting **indefinitely** for an event that can be **caused** only by one of the waiting processes.

- Three methods for handling deadlocks:
  1) To ensure the system **never** enters a deadlock state.
  2) Allow the system to **enter** the deadlock state and then **recover**.
  3) **Ignore** the problem.

Deadlock: Summary

- **Deadlock-prevention** ensures one of the conditions never occur.
- **Deadlock-avoidance** requires a priori information about the requested resources of each process.
- **Detection and recovery** uses wait-for graphs for detection. Termination of processes and preemption of resources can be used in deadlock recovery.

That is all for today!