Deadlocks

Operating Systems

Assoc. Prof. Milos Jovanovik, PhD

Objectives

- Resources – system model
- Necessary conditions for deadlocks
- Detection
  - Resource-allocation graphs
- Handling methods
  - Prevention
  - Avoidance
  - Deadlock detection and recovery
  - Do nothing

System Model: System Resources

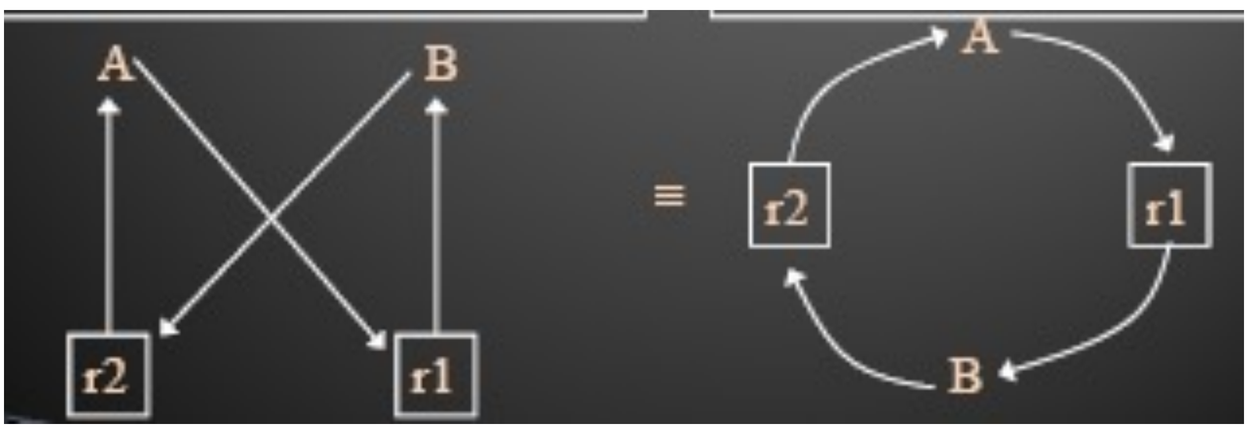
- The system consists of a finite number of resources, distributed among processes that compete to acquire them.
- There are several resource types, and each one has several number of identical instances.
  - Memory space, CPU cycles, files, I/O devices (e.g. printers) are different resource types;
  - If the system has 2 CPUs, then we say that the resource from type CPU has 2 instances. Similarly, the printer can have 5 instances;
- If the process requires a resource instance, then any instance should satisfy the requirement
  - If not, it cannot be regarded as the same resource;

Deadlock Problem

- A scenario in which 2 or more processes wait on each other endlessly.
- A set of processes is blocked when one of them is waiting on an event which can be triggered only by another process from the same set.
- Can happen when using mutual exclusion (working in critical sections).
- Example:
  - Process A has the printer, but now requests for a file;
  - Process B has the file, but now requests for the printer.

An Example with Databases

Process A	Process B
`lock(r2);`	`lock(r1);`
…	…
`lock(r1); // tries`	`lock(r2); // tries`



Deadlocks are Unwanted

Because:

- They "stop" the process from progressing
- They require intervention
- They reduce resource utilization
- All processes are waiting:
  - No process will be able to trigger an event that will awake some other process;
  - Usually the process waits on some resource to be released, which is acquired by some other blocked process;
- NONE of the processes:
  - Works
  - Can release a resource
  - Can be awaken
- All processes are waiting infinitely

Conditions for a Deadlock

- **Mutual Exclusion**: Only one process at a time can use a resource.
- **Hold and Wait**: A process holding at least one resource, is waiting to acquire additional resources held by other processes.
- **No Preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular Wait**: There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, …, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource that is held by P_0.

All conditions must be met simultaneously!

Solutions

- In general, there are 4 strategies to solve deadlocks:
  - **Prevention**: the concurrency in the OS is implemented in such a way that a deadlock is impossible
  - **Avoidance**: foresee the deadlock and avoid it
  - **Allow** a deadlock and react
  - **Do nothing** (mostly used)

Systems with multiple Processes and Resources

- In order to detect a deadlock in the system, we need a formalization which will allow us to represent the processes and the resources in the system

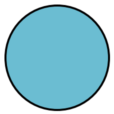
- The symplest way is to represent the system as a resource-allocation graph

Resource-Allocation Graph

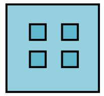
- A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Request edge: directed edge $P_i \rightarrow R_j$
- Assignment edge: directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph

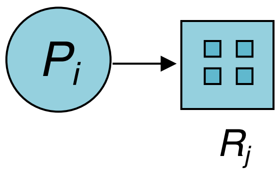
- Process



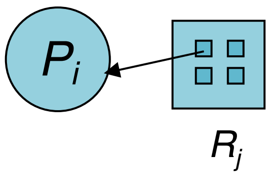
- Resource with 4 instances



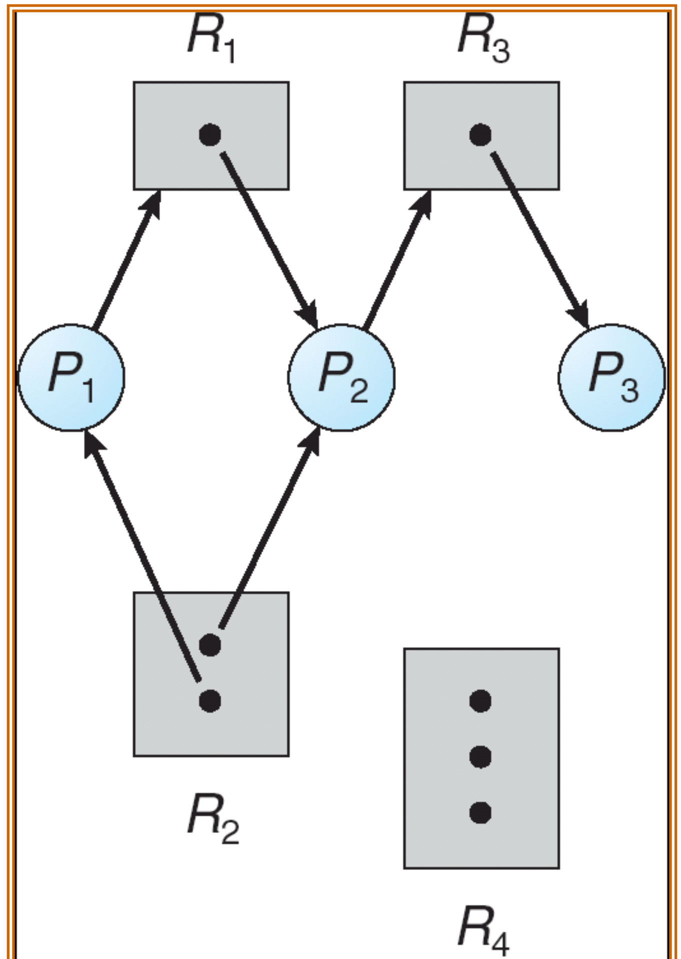
- P_i asks for one instance of R_j



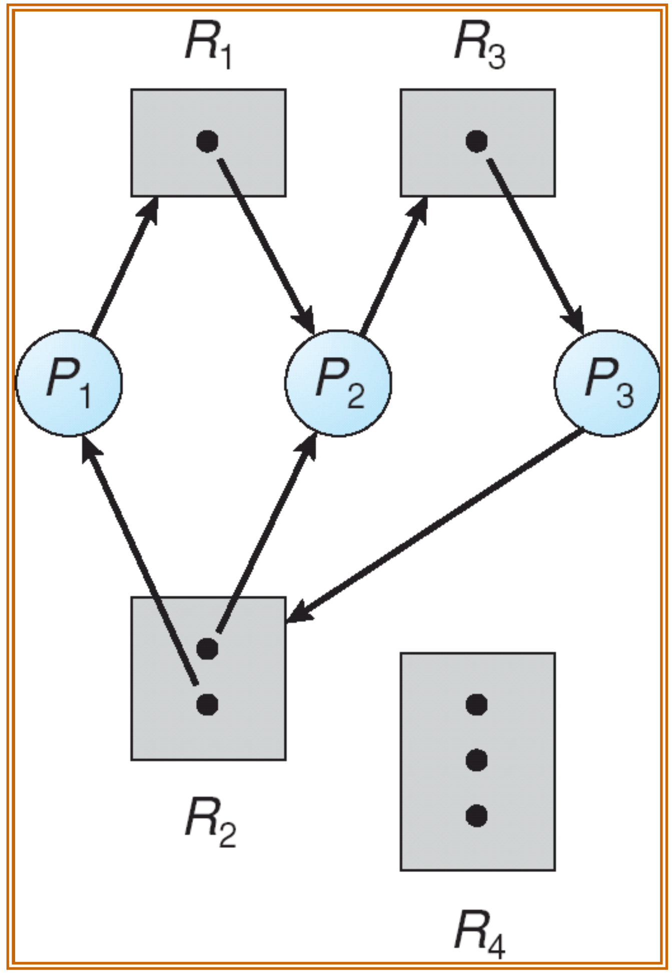
- P_i holds one instance of R_j



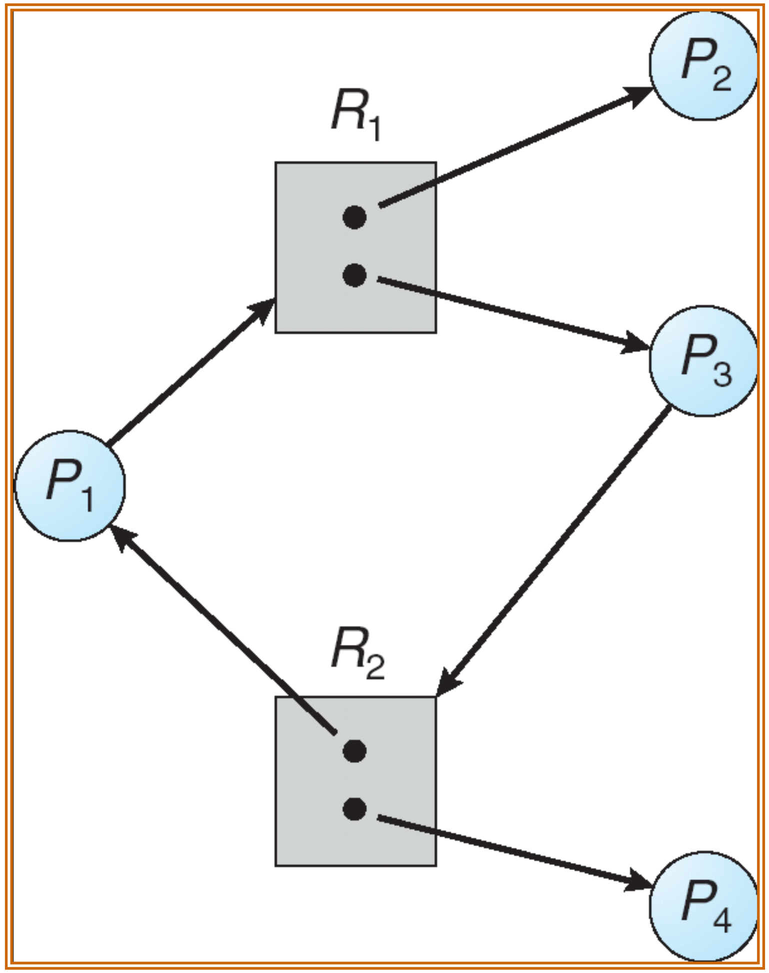
Resource-Allocation Graph: Example



Resource-Allocation Graph: Example with a Deadlock



Graph with a Cycle, but without a Deadlock



Basic Facts

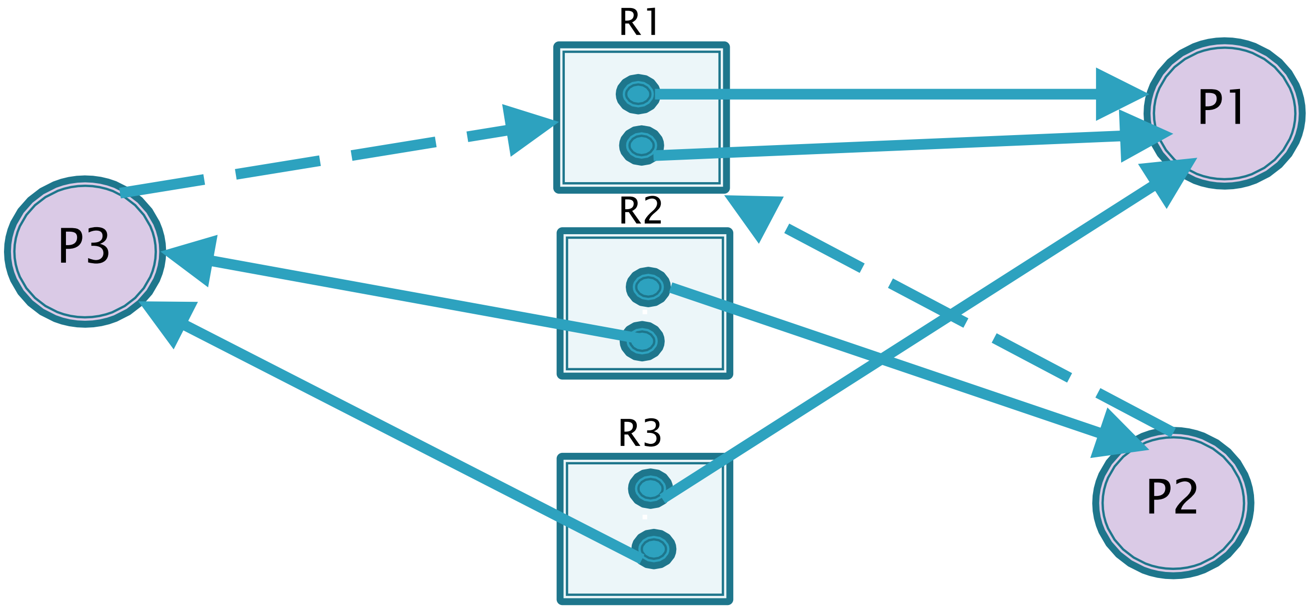
- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$
  - If there is only one instance per resource type, then there is a deadlock.
  - If there are several instances per resource type, there is a possibility of deadlock.

Example

- Assume that you have the following system:
  - There are three types (classes) of resources: R1, R2, R3.
  - There are two instances of each resource type.
  - There are three processes: P1, P2, P3.
- Some of the resource instances have already been assigned to processes:
  - Two instances of R1 are allocated to P1;
  - One instance of R2 is allocated to P2, and the other one to P3;
  - One instance of R3 is assigned to P1, and the other one to P3;
- Some processes have requested new instances:
  - P3 has requested one instance of R1;
  - P2 has reqursted one instance of R1, as well;

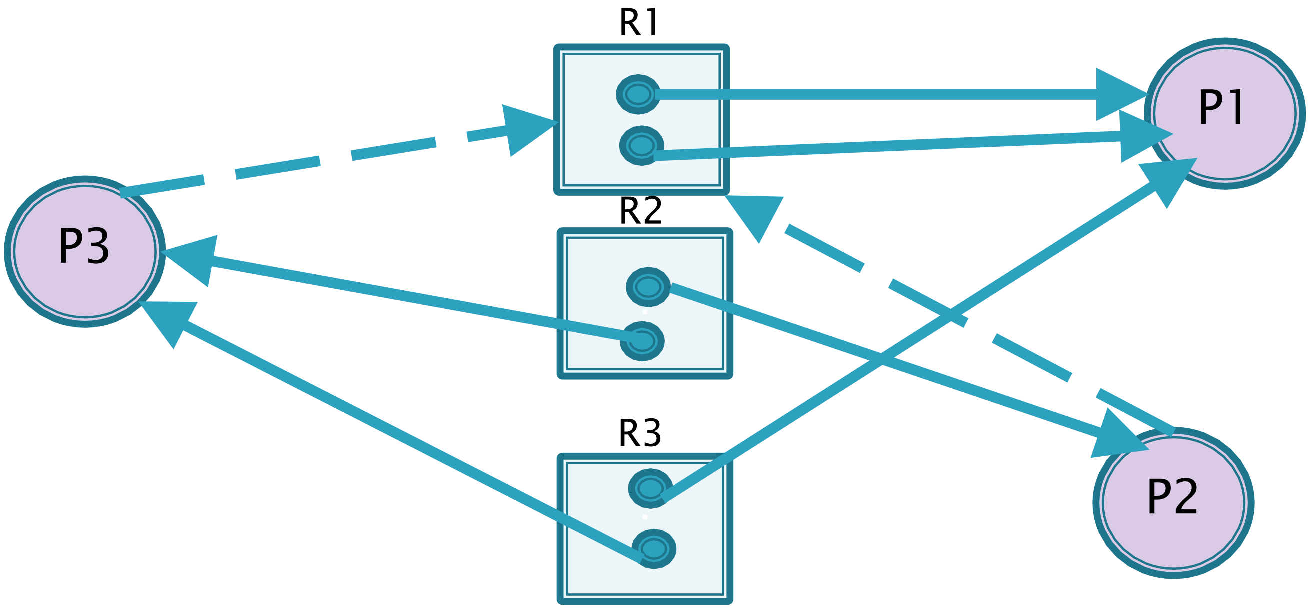
Example

- Which process can finish first?



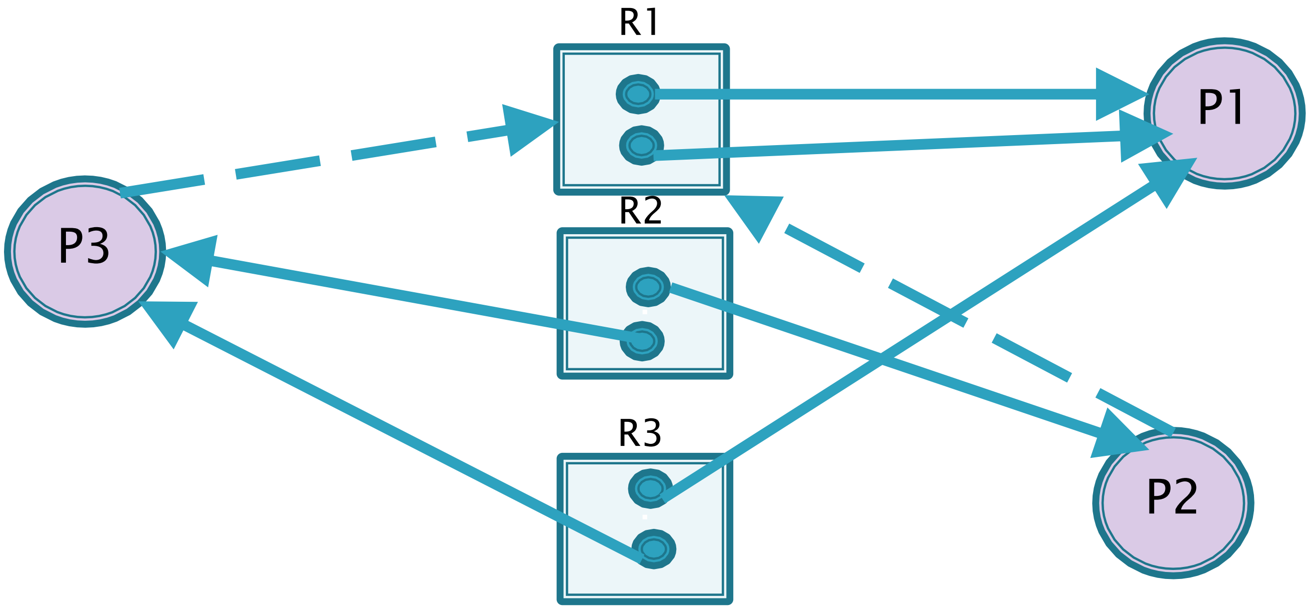
Example

- Which process can finish first?
  - P1 can finish first, it is in the "ready" state



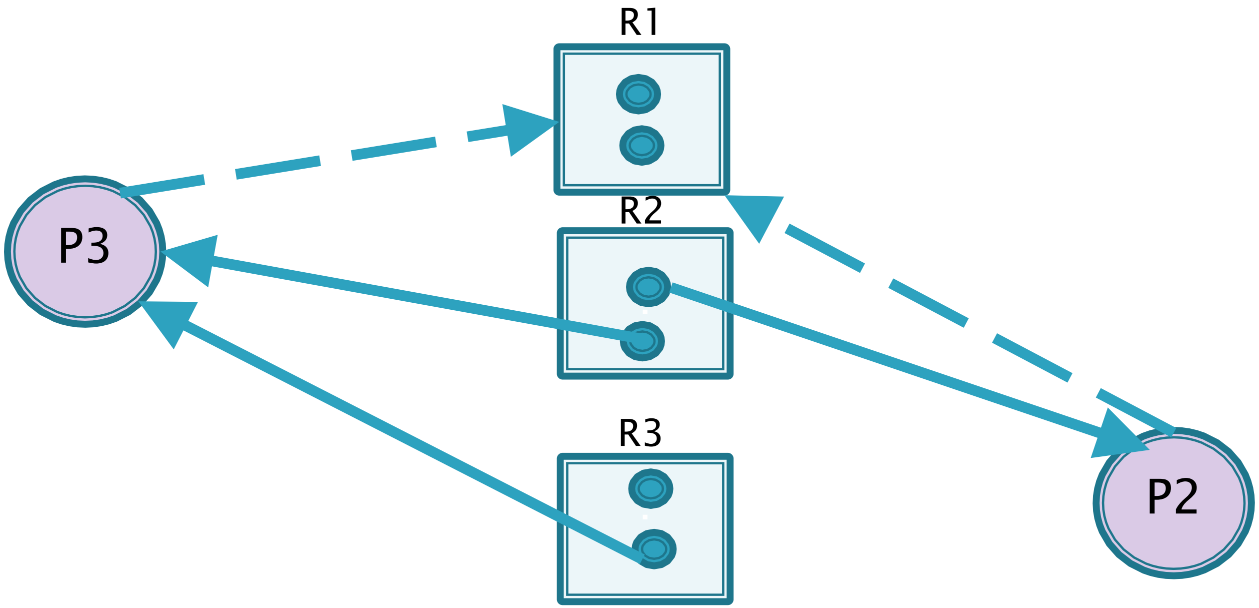
Example

- What about P2 and P3?
  - They are in the state of “blocked”



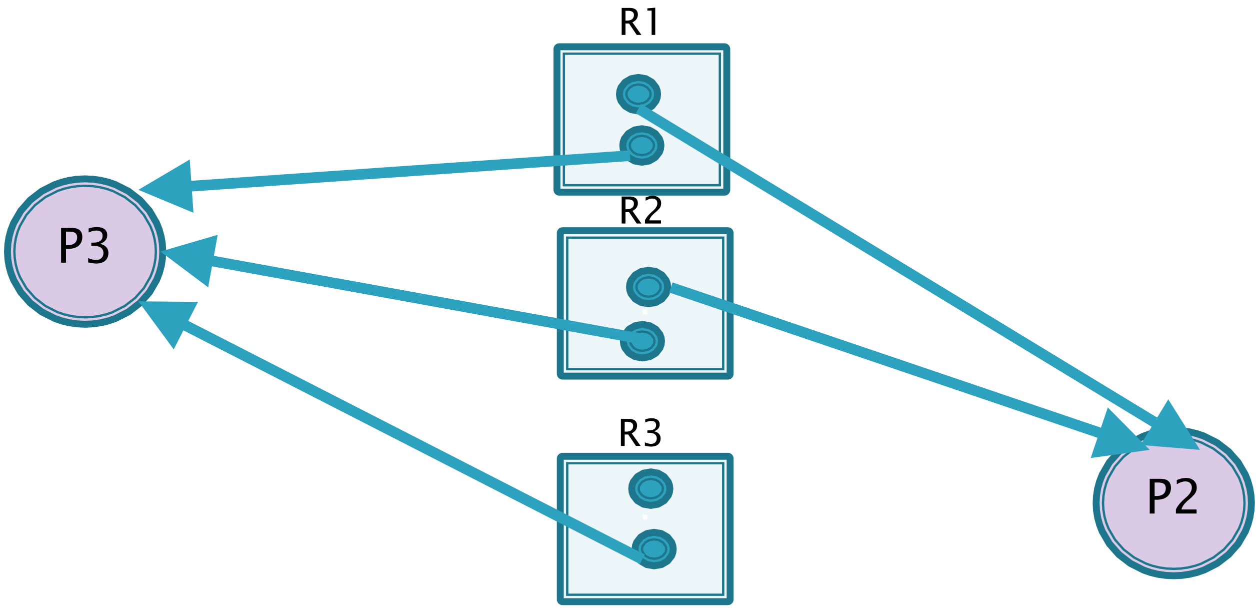
Example

- What happens after P1 finishes?
  - There are two R1 instances for P2 and P3, each



Example

- So, is the system in a deadlock?
  - No, because all processes can finish



Prevention

How to prevent Deadlocks

Conditions for Mutual Exclusion

- If we deny exclusive resource assignment to some process --> NO deadlocks...
- Not required for sharable resources; However, we must apply mutual exclusion for non-sharable resources.
- Some resources are intrinsically non-sharable.
- This is used in OS cores.

Conditions for Hold and Wait

- There must be a guarantee that whenever a process requests a resource, it does not hold any other resources.
  - Require process to request and get all resources before they begin their execution, or allow a process to request resources only when the process has none.
  - Drawbacks:
    - Low resource utilization;
    - Starvation is possible;

Conditions for 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.
- Preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Conditions for Preemption

- This is the worst condition for preventing deadlocks:
  - If a process has a printer assigned, and is in the middle of the job, taking away the printer just because the needed plotter is currently unavailable is a bad (if not an impossible) solution;

Conditions for Circular Wait

- Rule: Impose a total ordering of all resources.
- Require that each process requests resources in an increasing order of enumeration.
  - A process that holds a resource, cannot ask for a resource with a smaller number, until it frees its own resource
- Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock.
  - It is up to application developers to write programs that follow the ordering;
- Problems:
  - Inflexible solution;
  - Not applicable for bigger systems;

Deadlock Avoidance

How to avoid Deadlocks

Banker’s Algorithm

- Resources with multiple instances.
- 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.

Data Structures for the Banker's Algorithm

n = number of processes, and m = number of resources types.

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

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

Resource-Request Algorithm for Process P_i

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:

$$Available = Available - Request_i;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

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

Example of Banker's Algorithm

- 5 processes P_0 through P_4;
- 3 resource types
  - *A* (10 instances),
  - *B* (5 instances), and
  - *C* (7 instances).
- Snapshot at time T_0:

	Allocation	*Max*	*Available*
	A B C	*A B* C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example of Banker's Algorithm

- The content of *Need* is *Max – Allocation*.

	Need	*Available*
	A B C	*A B C*
P_0	7 4 3	3 3 2
P_1	1 2 2	
P_2	6 0 0	
P_3	0 1 1	
P_4	4 3 1	

- Safe squences:
  - <P1, P3, P4, P2, P0>
  - <P1, P4, P3, P0, P2>
  - ...

Example Resource-Request Algorithm

- P1 Request (1,0,2)
- Check that Request $_1 \leq$ Available
  - $(1,0,2) \leq (3,3,2) \Rightarrow$ true.

	Allocation	*Max*	*Available*
	A B C	*A B* C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example Resource-Request Algorithm

- State after allocating the requested resources:

	Allocation	*Need*	*Available*
	A B C	*A B* C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Banker's algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
  - Can request for (3,3,0) by P4 be granted? – not available
  - Can request for (0,2,0) by P0 be granted? – not safe

Usage of Deadlock Avoidance?

- Useful theoretically, but useless in practice
  - How will the system know the number of resources per process *in advance*?
  - The number of active processes changes dynamically, new users come and go!
  - Some resource can also be broken (physical error, driver error, etc.)!

- In practice, there is a very small number of systems that use the banker algorithm for deadlock avoidance!

Allow, Detect and React

React on a Deadlock

Allow, Detect and React

- Allow system to enter a deadlock state
- Analyse the situation (detection algorithm)
- Recover from deadlock
  - Pick a victim-process and kill it

- Questions:
  - How to detect when a deadlock is happening?
  - We need a formal algorithm: detecting cycles in oriented graph or a matrix algorithm;
- How to pick the victim?
  - To recognize the right process that makes the deadlock...

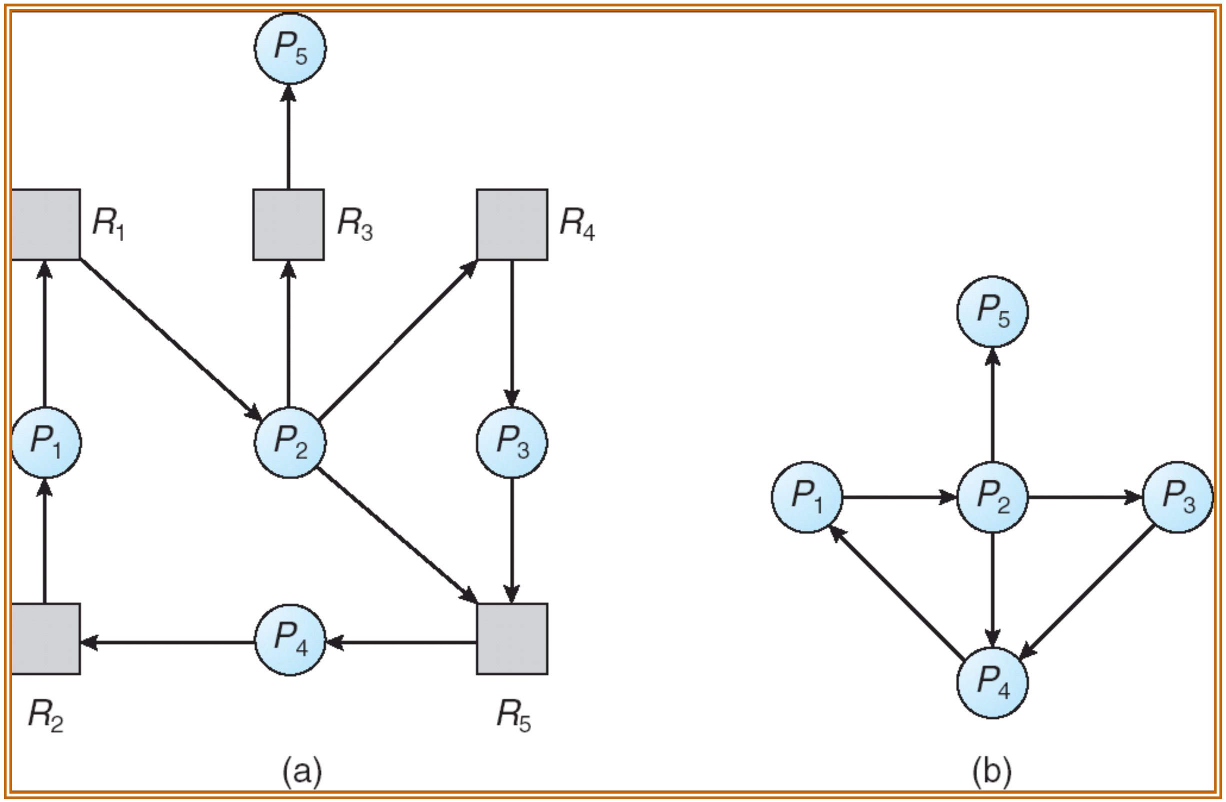
Single Instance of Each Resource Type

- Maintain a *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. If a cycle is there, there's a deadlock.

- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource-Allocation Graph and Wait-For Graph



Resource-Allocation Graph Corresponding Wait-For Graph

Do Nothing!

Don’t prevent, avoid or
proactively detect Deadlocks

Do not react!

- The simplest algorithm
- Ostrich algorithm (head in the sand)
  - How often there is a deadlock?
  - Mathematician vs Engineer!

Balance: Correctness vs. Comfort

- In each OS, the tables are finite and that leads to problems
  - Process and thread problems
  - Open files
  - Swap space
- It is probable, that each user prefers rare deadlocks over restrictions on resource use
  - **Balance between correctness and commodity**

Windows 11: Deadlocks

- **Resource Allocation Graph**: Windows 11 uses the RAG algorithm to detect deadlocks.
  - This algorithm represents the resource allocation relationship among processes as a directed graph, and checks for cycles in the graph;
  - If a cycle exists, it indicates a deadlock;
- **Timeout Mechanism**: Windows 11 employs a timeout mechanism to prevent deadlocks from occurring.
  - If a process is unable to acquire a resource within a specified time period, it releases all the resources it has acquired and goes into a wait state;
  - This ensures that a process does not hold resources indefinitely;
  - The timeout mechanism in Windows is implemented by setting a maximum wait time for processes to acquire resources and checking for cycles in the wait-for graph to detect deadlocks;

Windows 11: Deadlocks

- **Preemption**: Windows 11 uses preemption to resolve deadlocks.
  - If a process holds a resource and is unable to acquire another resource, the OS may preempt the resource and allocate it to another process;
  - This mechanism ensures that resources are used efficiently and prevents deadlocks from occurring;
- **Deadlock Detection and Recovery**: Windows 11 has built-in mechanisms to detect and recover from deadlocks.
  - When a deadlock is detected, the OS may terminate one or more processes involved in the deadlock to break the cycle and recover from the deadlock;
  - Random Process Termination, Priority-Based Process Termination, Resource-Usage-Based Process Termination, Deadlock Avoidance (is implemented to prevent the deadlocks - BY NOT GRANTING THE RESOURCE);

Linux: Deadlocks

- **Lock ordering**: The Linux kernel uses a strict lock ordering protocol to prevent deadlocks.
  - Lock ordering means that locks are always acquired in a specific order, and released in the reverse order;
  - This ensures that no process can acquire a lock that is held by another process, preventing circular dependencies and deadlocks;
- **Lockdep**: The Linux kernel includes a tool called "lockdep" that checks for potential deadlocks at runtime.
  - Lockdep analyses the lock dependencies in the kernel code and reports any potential circular dependencies;
  - This helps developers to identify and fix potential deadlocks before they occur;

Linux: Deadlocks

- **RCU** (**Read-Copy-Update**): RCU is a synchronization mechanism used in the Linux kernel to allow concurrent read access to shared data structures without the need for locks.
  - RCU uses a deferred deletion mechanism to remove data structures, ensuring that no process can access a deleted data structure;
- **Wait-for graph analysis**: The Linux kernel uses a wait-for graph to detect deadlocks.
  - The wait-for graph is a directed graph that represents the dependencies between processes and the resources they require;
  - When a process is waiting for a resource, it is added as a node to the wait-for graph;
  - If a process is waiting for another process that is also waiting for a resource, a cycle is detected, and a deadlock is assumed;
  - In this case, the kernel takes action to resolve the deadlock by either releasing the resources or terminating one of the processes involved;

Linux: Deadlocks

- **Priority inheritance**: Priority inheritance is a mechanism used in the Linux kernel to prevent priority inversion, which can lead to deadlocks.
  - Priority inheritance ensures that the priority of a low-priority process is temporarily increased to the priority of the highest-priority process that is waiting for a resource held by the low-priority process;
  - This prevents the high-priority process from being blocked and potentially causing a deadlock;

Conclusion

- Deadlocks are a potential problem for every system
- They can be avoided with safe states (there is a list of events that guarantees that all the processes will finish)
- The Banker algorithm avoids deadlocks by not allowing the request to force the system in an unsafe state
- The deadlock can be avoided with enumerating, so that each processes asks for resources in increasing order
- The starvation can be avoided with FCFS policy

Questions?