# Deadlock

# The Deadlock Problem

- Definition
  - A set of blocked processes each holding a resource and waiting to acquire a resource held by another process
  - None of the processes can proceed or back-off (release resources it owns)
- Example
  - semaphores *A* and *B*, initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| wait (A); | wait(B) |
| wait (B); | wait(A) |

# Deadlock Conditions

- Deadlock can arise if four conditions hold simultaneously
  - **Mutual exclusion:** only one process at a time can use a resource instance
  - **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 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 $P_0$.

# Methods for Handling Deadlocks

- **Ignore** the problem and pretend that deadlocks would never occur.
- **Prevent** the system from entering a deadlock state.
- Allow the system to enter a deadlock state and then **detect/recover**.

# The IGNORE Approach

- Pretend there is no problem
  - Unfortunately they can occur
  - Reasonable if
    - Deadlocks occur very rarely and cost of prevention is high
- Do your typical OSes take this approach?
- It is a trade off between
  - Overhead
  - Correctness

# The PREVENT Approach

- Restrain the ways requests can be made to break one of the four necessary conditions for deadlocks
- Attacking the mutual exclusion condition:
  - Some devices (such as printer) can be spooled
    - only the printer daemon uses printer resource
    - thus deadlock for printer eliminated
  - Not all devices can be spooled

# The PREVENT Approach

- Attacking the Hold and Wait Condition:
  - Require processes to request all resources before starting
- Problems
  - may not know required resources at start of run
  - also ties up resources other processes could be using
- Variation:
  - before a process requests for a new resource, it must give up all resources and then request all resources needed

# The PREVENT Approach

- Attacking the No Preemption Condition:
  - When a process holding some resources and waiting for others, its resources may be preempted to be used by others
- Problems
  - Many resources may not allow preemption; i.e., preemption will cause process to fail

# The PREVENT Approach

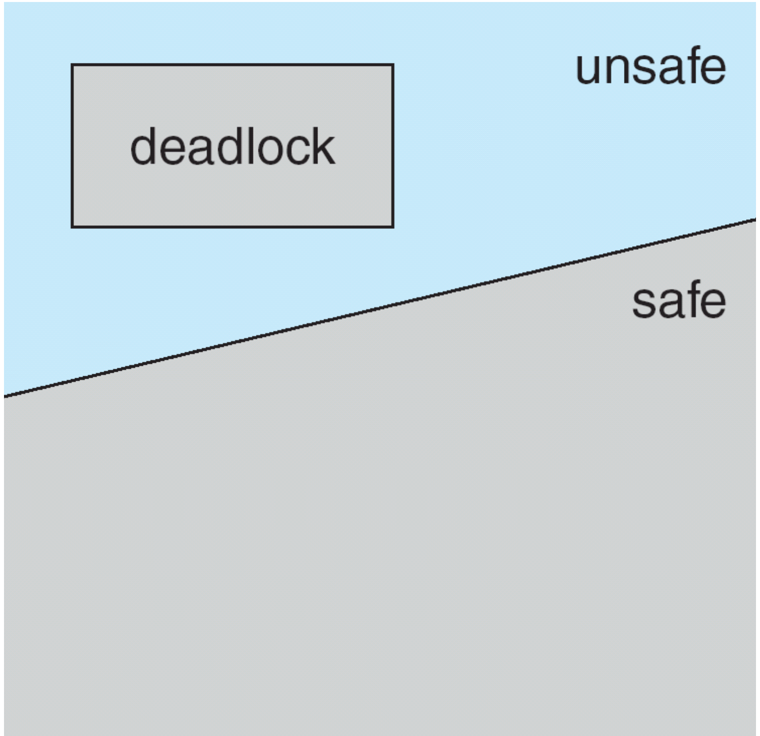
- Attacking the Circular Wait Condition:
  - Impose a total order of all resource types; and require that all processes request resources in the same order

# Deadlock Avoidance

- When a process requests available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all processes, such that
  - 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 $j < i$
  - That is:
    - If $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_j$ have finished
    - 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

# Deadlock Avoidance

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.

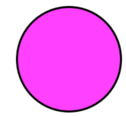


# Resource-Allocation Graph

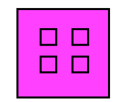
- A set of vertices $V$ and a set of edges $E$.
  - V is partitioned into two types:
    - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
    - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
  - E is partitioned into two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

# Resource-Allocation Graph

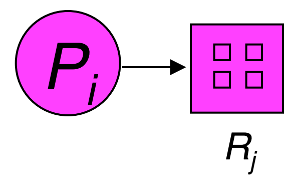
- Process



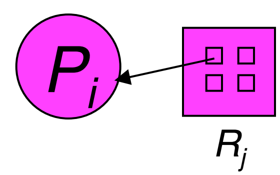
- Resource type with 4 instances



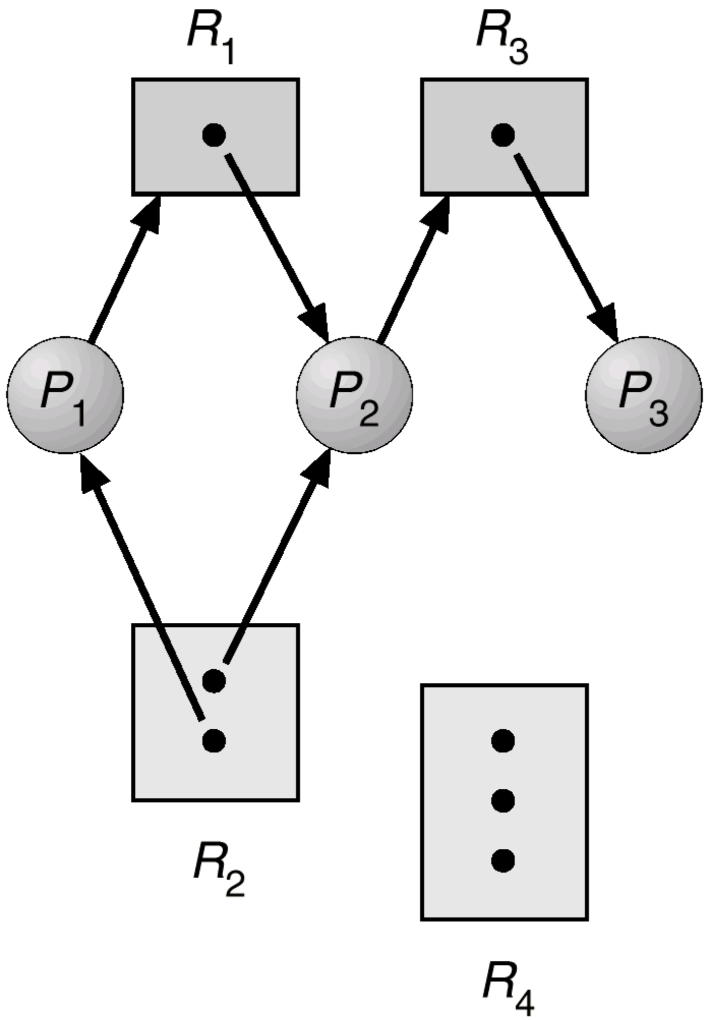
- $P_i$ requests instance of $R_j$




- $P_i$ is holding an instance of $R_j$

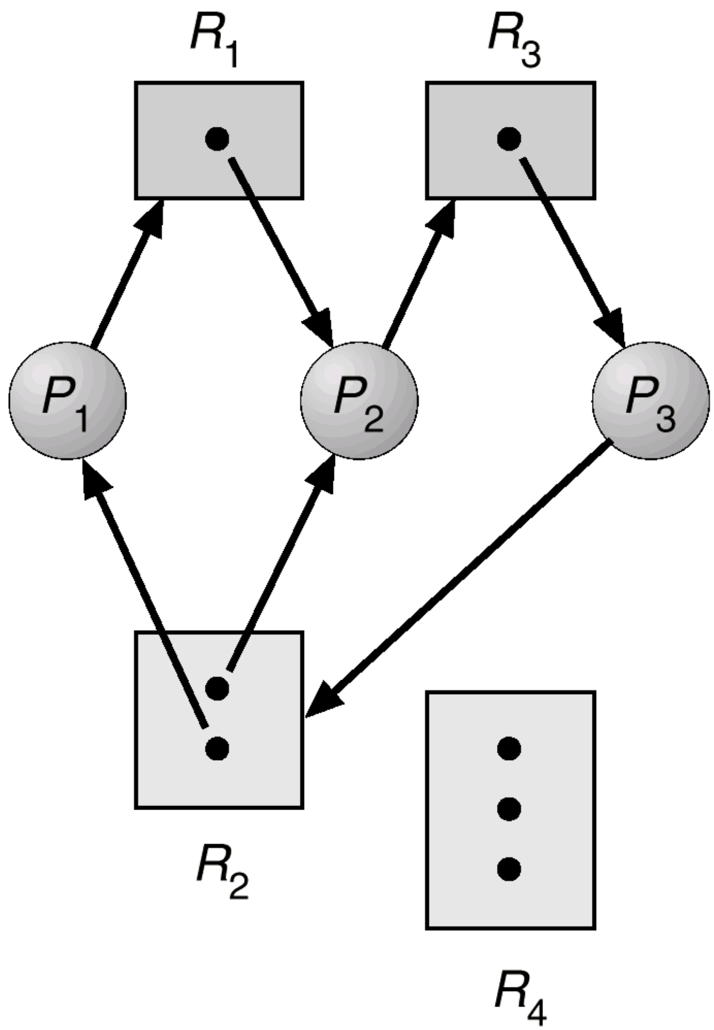


# Resource Allocation Example 1



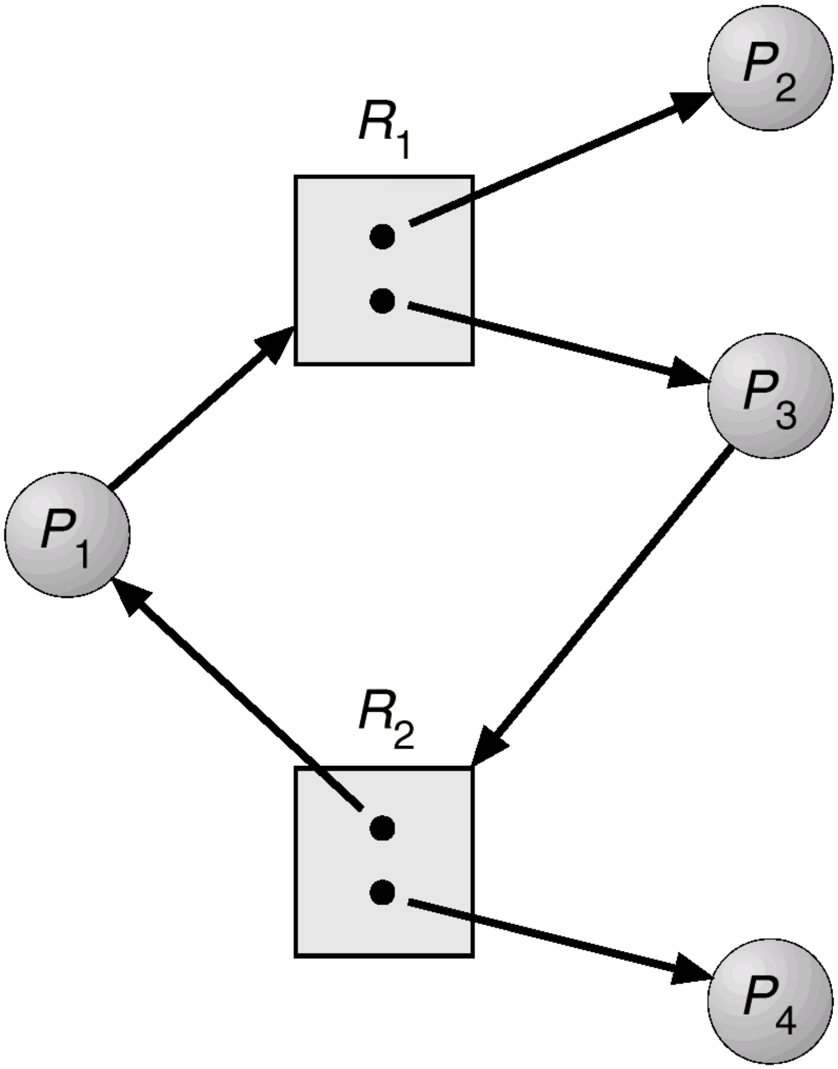

- Is there deadlock?

# Resource Allocation Graph Example 2




- Is there a deadlock?

# Resource Allocation Graph Example 3




- Is there a deadlock?

# Basic Facts

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

# 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

# Banker’s Algorithm

- Each process must a priori claim the maximum set of resources that might be needed in its execution.
- Safety check
  - Repeat
    - pick any process that can finish with existing available resources; finish it and release all its resources
    - until no such process exists
  - all finished → safe; otherwise → unsafe.

# Data Structure for the Banker's Algorithm

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

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

# Safety Algorithm

- Step 1: Let *Work* and *Finish* be vectors of length $m$ and $n$, respectively. Initialize:
  - *Work* = *Available*
  - *Finish* [$i$] = *false* for $i$ = 0, 1, …, $n$- 1
- Step 2: Find any $i$ such that both (If no, Step 4)
  - *Finish* [$i$] = *false*
  - $Need_i \leq Work$
- Step 3. $Work = Work + Allocation_i$
  - *Finish*[$i$] = *true*
  - Step 2
- Step 4. If *Finish* [$i$] == true for all $i$, then the system is in a safe state

## Resource-Request Algorithm for Process $P_i$

- Process $P_i$ wants $k$ instances of $R_j$ ($Request_i[j] = k$)
  - Step 1: If $Request_i \leq Need_i$ , go to step 2
    - Otherwise, raise error condition, since process has exceeded its maximum claim
  - Step 2: If $Request_i \leq Available$, go to step 3
    - Otherwise $P_i$ must wait, since resources are not available
  - Step 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 ⇒ the resources are allocated to Pi*
- *If unsafe ⇒ 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 | Need |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 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 |

**Question:**

Is this a safe state?

**Question:**

Can request for (1,0,2) by $P_1$ be granted?

# 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* | *Need* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 | 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 |

**Question:**

Can request for (3,3,0) by $P_4$ be granted?

**Question:**

Can request for (0,2,0) by $P_0$ be granted?

# Methods for Handling Deadlocks

- **Ignore** the problem and pretend that deadlocks would never occur.
- **Prevent** the system from entering a deadlock state.
- Allow the system to enter a deadlock state and then **detect/recover**.

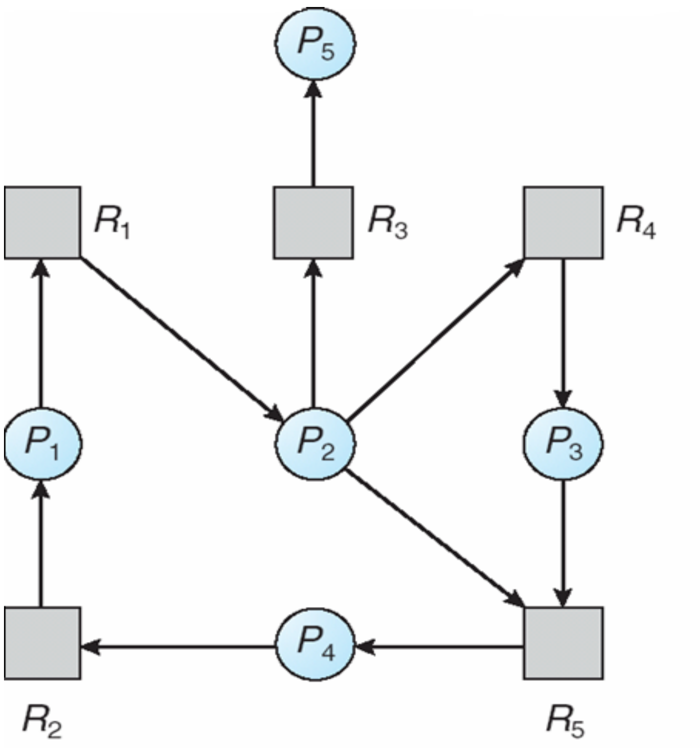
# Single Instance of Each Resource Type

- 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. If there is a cycle, there exists a deadlock

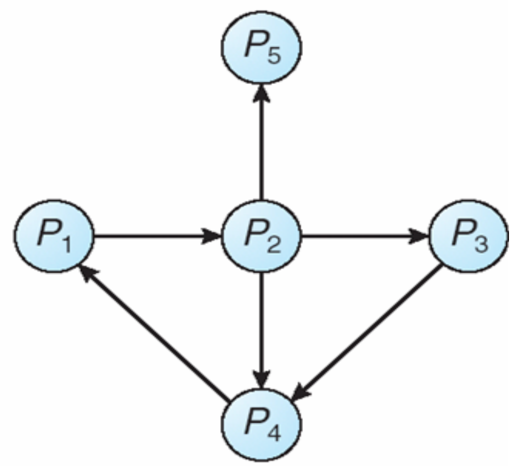
- An algorithm to detect a cycle in a graph requires an order of $n^2$ operations, where $n$ is the number of vertices in the graph

# Single Instance of Each Resource Type




(a)

Resource-Allocation Graph




(b)

Corresponding wait-for graph

# Additional Issues

- When there are several instances of a resource type
  - cycle detection in wait-for graph is not sufficient.
- Deadlock detection is very similar to the safety check in the Banker's algorithm

# Recovery from Deadlock

- Recovery through preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process state periodically
  - rollback a process to its checkpoint state if it is found deadlocked
- Recovery through killing processes
  - kill one+ of the processes in the deadlock cycle
  - the other processes get its resources
  - In which order should we choose process to kill?