

Course
TÖL401G: Stýrikerfi /
Operating Systems
7. Deadlocks

Chapter Objectives

- Illustrate how deadlock can occur.
- Define the four necessary conditions that characterize deadlock.
- Detect a deadlock situation in a resource allocation graph.
- Detect a deadlock situation using the matrix-based deadlock detection algorithm
- Evaluate the four different approaches for handling deadlocks.
- Apply the Safety algorithm to obtain safe schedules (if they exist).
- Apply the Banker's algorithm for deadlock avoidance.
- Evaluate approaches for recovering from deadlock.

Contents

1. Introduction
2. Deadlock Characterisation
3. Methods for Handling Deadlocks
4. Ignoring Deadlocks
5. Deadlock Detection & Recovery from Deadlocks
6. Deadlock Avoidance
7. Deadlock Prevention
8. Summary

Note for users of the Silberschatz et al. book: for didactical reasons, we have re-arranged the order of sections 7.4 to 7.7

Helmut Neukirchen: Operating Systems/updated

I.Hjörleifsson ingolfuh@hi.is st. TG225

7.1 Introduction:

Classes of Resources

- **Sharable resource**: can be used by many at the same time.
- **Non-sharable resource**: can only be used by one process (or thread) at the same time. Can be further classified into:
 - **Preemptable resources**: Non-sharable resources used by a process that can be revoked from the process without having a permanently negative influence on the process.
 - E.g. processor: can be preempted by scheduler. Because registers are saved, context switch is no problem even though CPU is a non-shareable resource.
 - Main memory: processes can be swapped out from memory and swapped in again as long as contents is saved on/restored from hard disk.
 - **Non-preemptable resources**: Non-sharable resources used by a process where it will have a permanently negative influence if it would be revoked from the process.
 - E.g. CD burner: would end up with a burned CD where the first part of a song is from process A, the second part from process B.
 - Printer: would end up with a printed page where some characters are from process A, some characters are from process B.

Introduction:

Usage Pattern for Resources

- Deadlocks always involve **non-preemptable** resources.
 - (Multiple processes waiting on each other's resource; if it is non-preemptable, the processes may deadlock. More on conditions necessary for a deadlock: later slides...)
- As any non-sharable resource may only be used by one process at the same time (mutual exclusion), processes need to request their usage before using it.

Three step sequence:

1. **Request resource**,
If resource is available:
2. **Use resource for a finite amount of time**,
3. **Release resource!**

Three step sequence looks trivial, but the underlying assumption is really important: **if we give a process the requested resources, the process will finally give all resources back.**

Introduction: Multiple Instances For Each Type Of Resource

- Process may depend on **different types of resources** at the same time.
 - E.g. a CD cloning process may require a CD reader and a CD burner at the same time.
- For each type of resource, there may be **multiple instances** available.
 - E.g. two CD reader devices – then, any of the two devices would satisfy the request of a process for a CD reader.
- More formally:
 - Resource types R_1, R_2, \dots, R_m
 - Each resource type R_i has W_i instances.
 - E.g.:
Two CD readers, one CD writer:
 R_1 =CD reader, R_2 =CD writer,
 W_1 =2, W_2 =1.

Introduction:

Managing Access to Resources

- Resource may be **managed by the operating system**: requesting, using & releasing is performed by **system calls**.
 - E.g. a file in the file system:
 - Before using it, it's usage must be requested: "open" system call.
 - OS could block system call if file has already been locked by another process.
 - Then, it can be used: e.g. writing to it using a "write" system call
 - After usage, it must be released: "close" system call.
- Resource may be **managed by processes (threads)**: **processes themselves are** responsible for granting access to resource.
 - E.g. a shared data structure in shared memory:
 - Ensure mutual exclusion by guarding usage of data structure with, e.g., a semaphore (which lead to system calls that may block).

Introduction:

Example (1)

- Two types of resources (R_1 , R_2 with $W_1=1$, $W_2=1$) used by two processes P_A , P_B .
- Processes manage mutual exclusive access to resources using a semaphore for each type of resource:

```
semaphore resourceOne = resourceOne.init(1);  
semaphore resourceTwo = resourceTwo.init(1);
```

```
void process_A() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```

```
void process_B() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```


Introduction:

Example (2)

- Code is free from deadlocks. (No proof given.)
- Two sample schedules (single processor/single core):

1. schedule:

```
Process_A: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

```
Process_B: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

2. schedule:

```
Process_A: resourceOne.wait();
```

Time slice expired: context switch

```
Process_B: resourceOne.wait();
```

Process B is put to sleep: context switch

```
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

Process B is waked up again: context switch

```
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

Introduction:

Example (3)

- Same example, but two lines swapped in resource request of process B:

```
semaphore resourceOne = resourceOne.init(1);  
semaphore resourceTwo = resourceTwo.init(1);
```

```
void process_A() {  
    resourceOne.wait();  
    resourceTwo.wait();  
  
    use_both_Resources();  
  
    resourceTwo.signal();  
    resourceOne.signal();  
}
```

```
void process_B() {  
    resourceTwo.wait();  
    resourceOne.wait();  
  
    use_both_Resources();  
  
    resourceOne.signal();  
    resourceTwo.signal();  
}
```

Introduction:

Example (4)

- Deadlocks may occur. Two sample schedules (single proc./single core):

1. Schedule (no deadlock):

```
Process_A: resourceOne.wait();  
           resourceTwo.wait();  
           use_both_Resources();  
           resourceTwo.signal();  
           resourceOne.signal();
```

```
Process_B: resourceTwo.wait();  
           resourceOne.wait();  
           use_both_Resources();  
           resourceOne.signal();  
           resourceTwo.signal();
```

2. schedule:

```
Process_A: resourceOne.wait();
```

Time slice expired: context switch

```
Process_B: resourceTwo.wait();  
           resourceOne.wait();
```

Process B is put to sleep: context switch

```
           resourceTwo.wait();
```

Process A is put to sleep

Each process waits for the other:

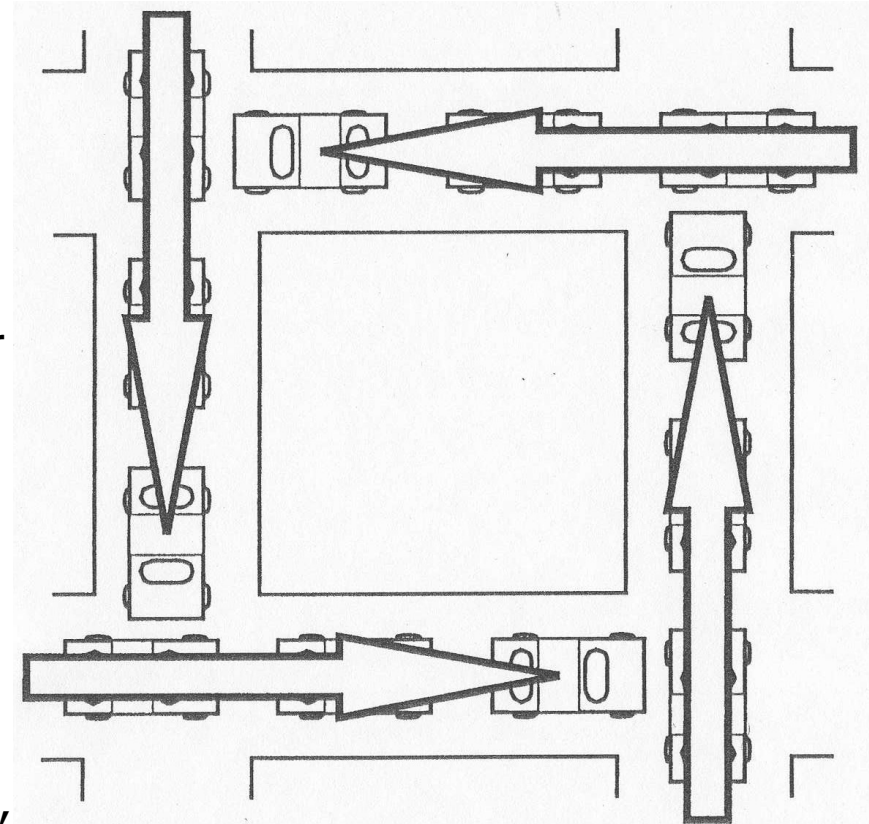
Process A waits for resourceTwo (however that will only be released by process B if it got resourceOne from process A).

Process B waits for resourceOne (however that will only be released by process A if it got resourceTwo from process B).

***** DEADLOCK *****

7.2 Deadlock Characterisation

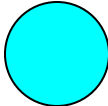

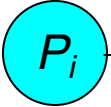
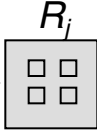
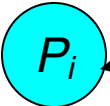
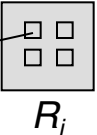
- Law passed Kansas legislature early 20th century (according to Silberschatz et al.):
 - “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone.”
- Informal definition of a deadlock:
 - „A set of processes is in a deadlock state, if each process from this set is waiting for an event that can only be triggered by another process from that set.”



Necessary Conditions For A Deadlock

- According to Coffman et al. (1971), four conditions must hold simultaneously for a deadlock state:
 - **Mutual exclusion:** only one process at a time can use a resource (non-sharable 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 $\{P_0, P_1, \dots, P_n\}$ 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 .
- (Still general assumption applies: if we give a process all the requested resources, the process will finally give all resources back.)

Resource-Allocation Graph

- For investigating deadlocks (and processes using resources) more formally, a **resource-allocation graph** can be used:
 - **Process node** represent a process: 
 - **Resource node** represents a resource type:  (4 instances of this type of resource are available)
 - **Request edge**:   (process P_i requests 1 resource of type R_j . If request can be fulfilled, edge becomes instantaneously an assignment edge.)
 - **Assignment edge**:   (1 instance of a resource type R_j is allocated to process P_i)

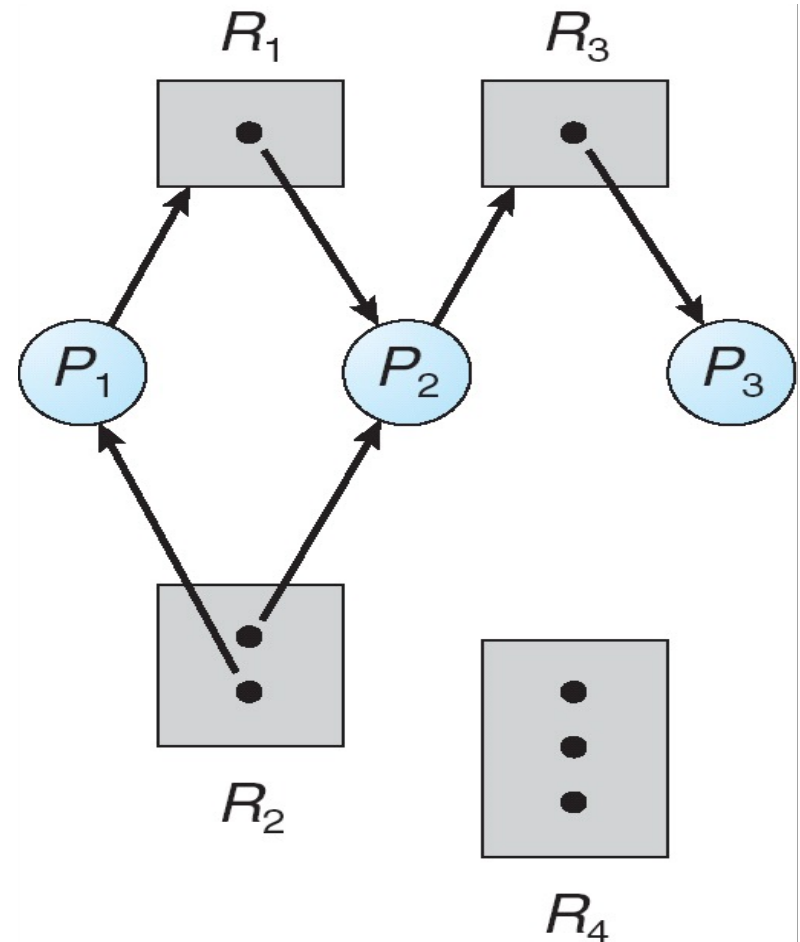
Resource-Allocation Graph: Example

- Resources:

- R_1 : 1 instance,
- R_2 : 2 instances,
- R_3 : 1 instance,
- R_4 : 3 instances.

- Processes:

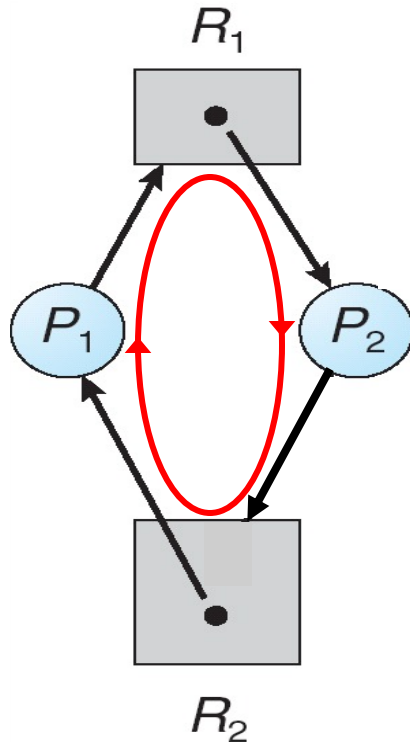
- P_1 : waits for one instance of R_1 , one instance of R_2 is allocated to it.
- P_2 : waits for one instance of R_3 , one instance of R_1 and one instance of R_2 is allocated to it.
- P_3 : one instance of R_3 is allocated to it.



Resource-Allocation Graphs & Deadlocks

- Resource-allocation graph can be used to analyse Circular wait (& Hold and wait) condition:
 - If directed graph contains **no cycles**
⇒ **no deadlock**.
 - If directed graph contains a cycle ⇒
 - if **only one instance per resource type**, then deadlock.
 - (In this case, a cycle is both a necessary and sufficient condition.)
 - if **several instances** per resource type, **possibility** of deadlock.
 - (In this case, a cycle is a necessary, but not a sufficient condition: further investigation required to decide finally.)

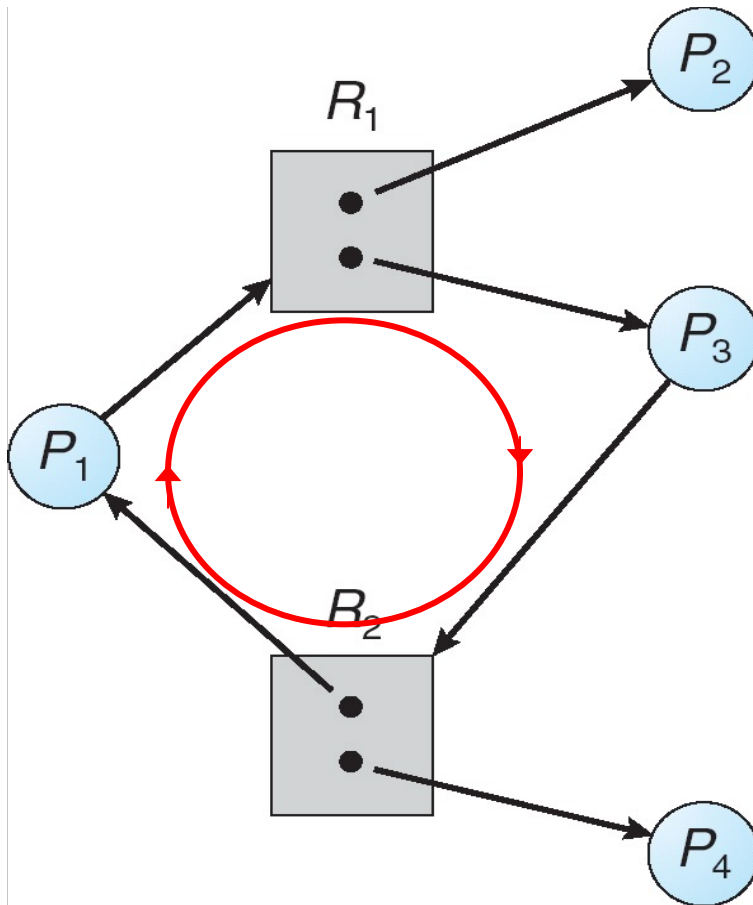
Resource-Allocation Graphs & Deadlocks: Examples (1)



- Cycle & only one instance per resource type
 \Rightarrow deadlock!
 - (P_1 holds R_2 and waits for R_1 that is held by P_2 .
 P_2 holds R_1 and waits for R_2 that is held by P_1 . I.e. cycle $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$.)

- Note: all 4 deadlock conditions are fulfilled:
 - In particular: circular wait, hold and wait.
 - (mutual exclusion, no preemption are assumed as given anyway.)

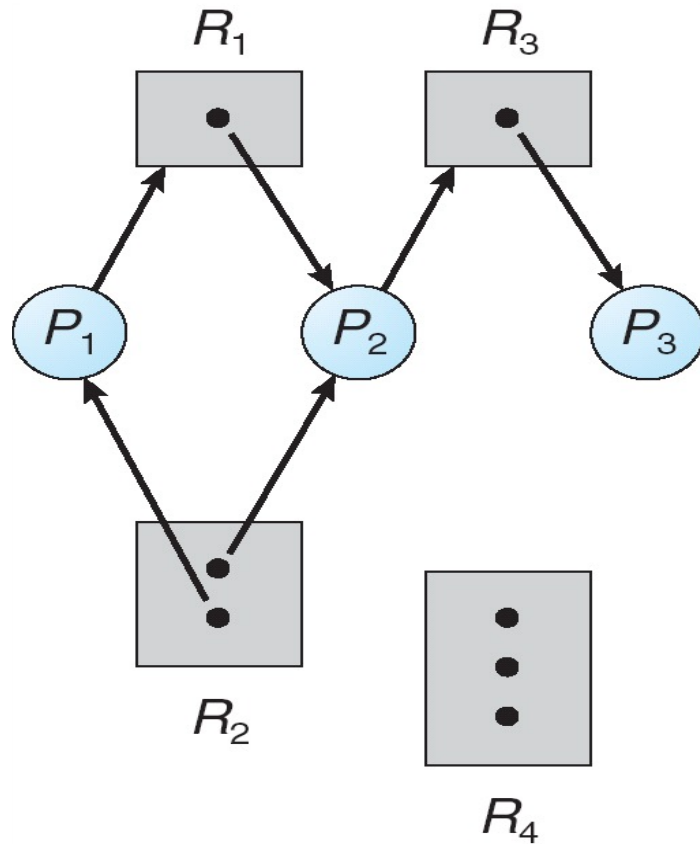
Resource-Allocation Graphs & Deadlocks: Examples (2)



■ Cycle & several instances per resource type \Rightarrow further analysis required:

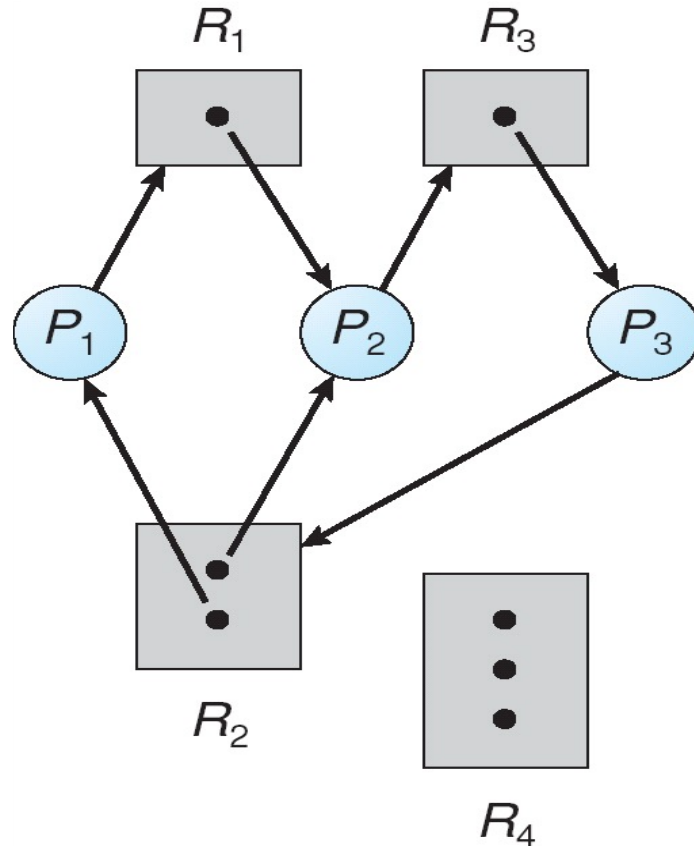
- Even though we have a cycle $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ we have **no deadlock**:
 - While P_1 and P_3 currently block each other, P_2 and P_4 are not blocked and thus still able to run. As they are able to run, they will eventually release their allocated resource, thus enabling also P_1 and P_3 to continue.

Resource-Allocation Graphs & Deadlocks: Examples (3)



- No cycle (note direction of edges!)
 \Rightarrow no deadlock!

Resource-Allocation Graphs & Deadlocks: Examples (4)



- Starting from example (3), P_3 requests R_2 .
- Cycle & several instances per resource type \Rightarrow further analysis required:
 - We have a small cycle $P_3 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3$. For this, one might be tempted to argue that maybe P_1 eventually releases the instance of R_2 it holds, thus enabling P_3 to continue. But as we assume hold-and-wait, P_1 does rather not release the instance of R_2 it holds as it is also waiting for R_1 . So already this is a deadlock. Even if this would not be a deadlock, having a closer look reveals that there is an even further, larger cycle into which also P_1 is involved:
 - Cycle $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$. In this cycle, all processes are waiting for a resource that is held by one of the other processes \Rightarrow **deadlock of P_1 , P_2 , and P_3**

7.3 Methods for Handling Deadlocks

- 4 different strategies to deal with deadlocks:
- **Ignore** the problem!
 - Not really a solution of the problem, but at least easy to implement. (→section 7.4)
- **Detect** deadlocks **and recover** from them.
 - Allow system to enter deadlock state, detect this and recover (→section 7.5).
- **Avoid** deadlocks by careful dynamic resource management.
 - Reject resource requests as soon as processes are endangered of creating a deadlock due to that resource request (→section 7.6).
- **Prevent** that deadlocks may occur at all.
 - Ensure that at least one of the four conditions necessary for a deadlock cannot hold (→section 7.7).
- Note: while the terms “avoidance” and “prevention” have a similar meaning, they refer to different solution strategies.

Handling deadlock: OS vs. applications

- All major operating system ignore deadlocks, i.e. deadlocks may occur concerning the resource managed by an OS (e.g. devices).
- While OSES are too generic resources to handle in a reasonable way concerning deadlocks, applications know their resources better.
- Example: **Database Management Systems** (DBMSes) have only one resource: data (e.g. the tables of an SQS database).
 - DBMS implementations use heavily **deadlock detection & recovery** to avoid deadlocks in concurrent atomic transactions.
 - Atomic transaction: either apply all changes or no changes, i.e. do not abort transaction with halfway applied changes.
 - Transactions may involve shared resources, i.e. same data accessed by different transactions in parallel: Deadlocks possible!
 - DBMS will detect deadlocks and then one (or more) transaction gets aborted (=none of their changes applied) while the others succeed (=all of their changes applied).

7.4 Ignoring Deadlocks

- Ignore the deadlock problem altogether.
 - Deadlocks may occur. If a deadlock occurs: so what?! – We have a multi-tasking operating system, thus we are still able to work with the remainder of the system that is not deadlocked. OK, maybe some resource (e.g. the printer) is blocked, but as long as no one wants to print, no one cares. At some point in time, hopefully someone restarts the system, thus resolving the deadlock.
- Advantage:
 - Easy to implement (nothing has to be implemented).
 - No management overhead for avoiding or detecting deadlocks.
- Disadvantage:
 - Processes are blocked in deadlock state: important data may not have been saved yet and thus get lost when process is killed or system is restarted.
- Reasonable approach
 - if deadlocks do not occur very often.
 - if costs of the other more advanced strategies are too high.
- All major operating system follow this approach: Unix, MS Windows (& Java).

7.5 Deadlock Detection & Recovery from Deadlock

- **Allow system to enter deadlock state:** processes may request resources as they like. This may lead to allocations of resources that result in a deadlock.
- In contrast to the “ignore deadlocks” approach, we want the **operating system to detect that a deadlock occurred**. If such a deadlock situation has been detected, the operating system should apply some recovery scheme to **remove the deadlock** (→later slides).
- **Deadlock detection algorithms:**
 - **If all resource types have only a single instance:** search for cycles in the resource allocation graph (slide 7-16). Complexity of graph algorithms for detecting cycles: $O((n+m)^2)$, where n = number of processes, m = number of different resource types, i.e. square of number of nodes in the graph.
 - **In case of several instances per resource type:** more advanced, matrix-based algorithm required (→next slides). Complexity: $O(m \cdot n^2)$, where n is the number of processes, m the number of different resource types.

Matrix-based Deadlock Detection Algorithm for Several Instances of a Resource Type (1)

- Deadlock detection algorithm uses vectors and matrices as data structures:

Existing resources vector **E**

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$(E_1, E_2, E_3, \dots, E_m)$$

Allocation matrix **C**

(describes how many instances of each resource are currently allocated (=“hold”) to each process):

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \dots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \dots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \dots & C_{nm} \end{bmatrix}$$

Row n describes current resource allocation of process n

Available resources vector **A**

(describes how many instances of each resource are currently available/free):

$$(A_1, A_2, A_3, \dots, A_m)$$

Request matrix **R**

(describes how many instances of each resource are currently requested (=“wait”) by each process):

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \dots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \dots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \dots & R_{nm} \end{bmatrix}$$

Row n describes which resources are currently requested by process n

Matrix-based Deadlock Detection Algorithm for Several Instances of a Resource Type (2)

- Algorithm:

1. Initialise the available resources vector A and the allocation matrix C according to the current state of the system and the request matrix R according to the current requests of the processes. Be prepared to be able to mark later-on processes as finished: thus, unmark now all processes.
2. Find an unmarked process P_i whose row i in the request matrix R is smaller (or equal) than the available resources vector A .
If no such process exists, go to step 4.
3. Add row i of the allocation matrix C to the available resources vector A . Mark process P_i as finished.
Go to step 2.
4. If there are unmarked processes: these processes are in a deadlock state.
If all processes are marked, system is deadlock free.

I.e. find a process for which it is possible to satisfy its current request.

I.e. pretend that we grant the request to that process and hence, finally, this process releases its resources making them available to the other processes.

Matrix-based Deadlock Detection Algorithm: Example (1)

- Example: Currently the system is in the state described by (E,) A, C (= "hold" part of "hold & wait") and R (= "wait" part of "hold & wait"):

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of
each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource
are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource
are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3

- Now, we want to know: is this a deadlock state or not?

⇒ Apply matrix-based deadlock detection algorithm! (Use this state as step 1.)

Matrix-based Deadlock Detection Algorithm: Example (2)

- First iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is smaller/equal than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1

P2

P3

Matrix-based Deadlock Detection Algorithm: Example (3)

- First iteration, step 3: Add row i of the allocation matrix C to the available resources vector A. Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

$$+ \Rightarrow \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3 ✓

Matrix-based Deadlock Detection Algorithm: Example (4)

- Second iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1

P2

P3 ✓

Matrix-based Deadlock Detection Algorithm: Example (5)

- Second iteration, step 3: Add row i of the allocation matrix C to the available resources vector A. Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 2 & 2 & 0 \end{pmatrix}$$

$$+ \Rightarrow \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2 ✓
P3 ✓

Matrix-based Deadlock Detection Algorithm: Example (6)

- Third iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 4 & 2 & 2 & 1 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2 ✓
P3 ✓

Matrix-based Deadlock Detection Algorithm: Example (7)

- Third iteration, step 3: Add row i of the allocation matrix C to the available resources vector A. Mark process P_i as finished.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \\ & \text{Hard disk} & \text{CD drive} & \text{Printer} & \text{Scanner} \end{matrix})$$

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = (\begin{matrix} 4 & 2 & 2 & 1 \\ + \Rightarrow & 4 & 2 & 3 & 1 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1 ✓

P2 ✓

P3 ✓

Matrix-based Deadlock Detection Algorithm: Example (8)

- Fourth iteration, step 2: Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A . If no such process exists, go to step 4.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1 ✓
P2 ✓
P3 ✓

- Fourth iteration, step 4: If there are unmarked processes: these processes are in a deadlock state. If all processes are marked, system is deadlock free. \Rightarrow no deadlock!

Matrix-based Deadlock Detection Algorithm: Example (9)

- Modified example:

Currently the system is in the state described by (E,) A, C and R:

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = (\begin{matrix} 4 & 2 & 3 & 1 \\ & \text{Hard disk} & \text{CD drive} & \text{Printer} & \text{Scanner} \end{matrix})$$

Available resources vector A

(describes how many instances of
each resource are currently available/free):

$$A = (\begin{matrix} 2 & 0 & 0 & 0 \end{matrix})$$

Allocation matrix C

(describes how many instances of each resource
are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Request matrix R

(describes how many instances of each resource
are currently requested by each process):

$$R = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3

- Now, step 2 (Find an unmarked process P_i whose row i in the request matrix R is \leq than the available resources vector A .) fails and we go to step 4. As no processes are marked, step 4 tells us that processes P1, P2, P3 are in a **deadlock state** (part of deadlock cycle)!

Detection Algorithm Usage

- Now, we have a nice algorithm to detect whether processes are in a deadlock state or not.
- When, and how often, to invoke a deadlock detection algorithm?
- Possibilities:
 - After each resource request that cannot be granted.
 - Disadvantage: computation overhead added to unsuccessful resource request operations.
 - Periodically.
 - Disadvantage: what is the appropriate period? Period too short: too much overhead. Period too long: deadlock remains undetected for some time during which the involved processes are blocked and even more processes might get involved into the deadlock.
 - When CPU utilisation is low.
 - Then, computation overhead does not harm. Furthermore, low CPU utilisation may be considered as a (necessary, but not sufficient) indicator of a deadlock: during a deadlock, processes are blocked (assuming no busy wait).

Recovery from Deadlock

- Once a deadlock detection algorithm has detected a deadlock, this situation has to be handled:
 - Either a human system operator is informed who has to deal with the deadlock **manually**.
 - Typically, the operator kills one of the deadlocked processes.
 - (This leads typically to a release of all resources held by that process, making these resources available for other processes that wait for them.)
 - Or **automatic recovery** is initiated by the system:
 - Either process termination (→next 2 slides)
 - or resource preemption. (→slide after the next 2 slides)

Process Termination (1)

- **Terminate** (& restart) at least one **process** that is part of the deadlock cycle.
 - “Brutal”, but simple approach.
 - When a process is terminated, the operating system releases typically all resources allocated to that process. Hence, other processes that were waiting for that resources may then use them.
 - However, this may leave resource in an inconsistent state.
⇒ Resource needs to be reset.
- Two alternative approaches of process termination:
 - **Abort all deadlocked processes.**
 - Definitely resolves the deadlock cycle. However, more processes than necessary may get killed.
 - **Abort one process at a time until the deadlock cycle is eliminated.**
 - As we have seen on slide 7-20, multiple deadlock cycles may exist at the same time. Hence, run deadlock detection algorithm after each killing of a process to decide whether this was sufficient to resolve the deadlock cycle.

Process Termination (2)

- When aborting one process at a time: **how to select process to abort?**
 - If deadlock detection algorithm runs frequently, it may be able to **identify one single process that closed the deadlock cycle**. This would be an obvious candidate.
 - Otherwise, we only know the set of deadlocked processes. Then selection could be based on many factors, e.g.
 - **Priority of the process** (scheduling priority):
 - abort process with lowest priority.
 - **How long has process computed and how many resources has it used?**
 - Process that has been used a lot of resources (incl. CPU time) is more likely to be finished soon, hence we should not abort that one.
 - **Is process interactive or batch?**
 - Aborting an interactive process is more annoying for the user than aborting a batch process that can be restarted without requiring user input.

Resource Preemption

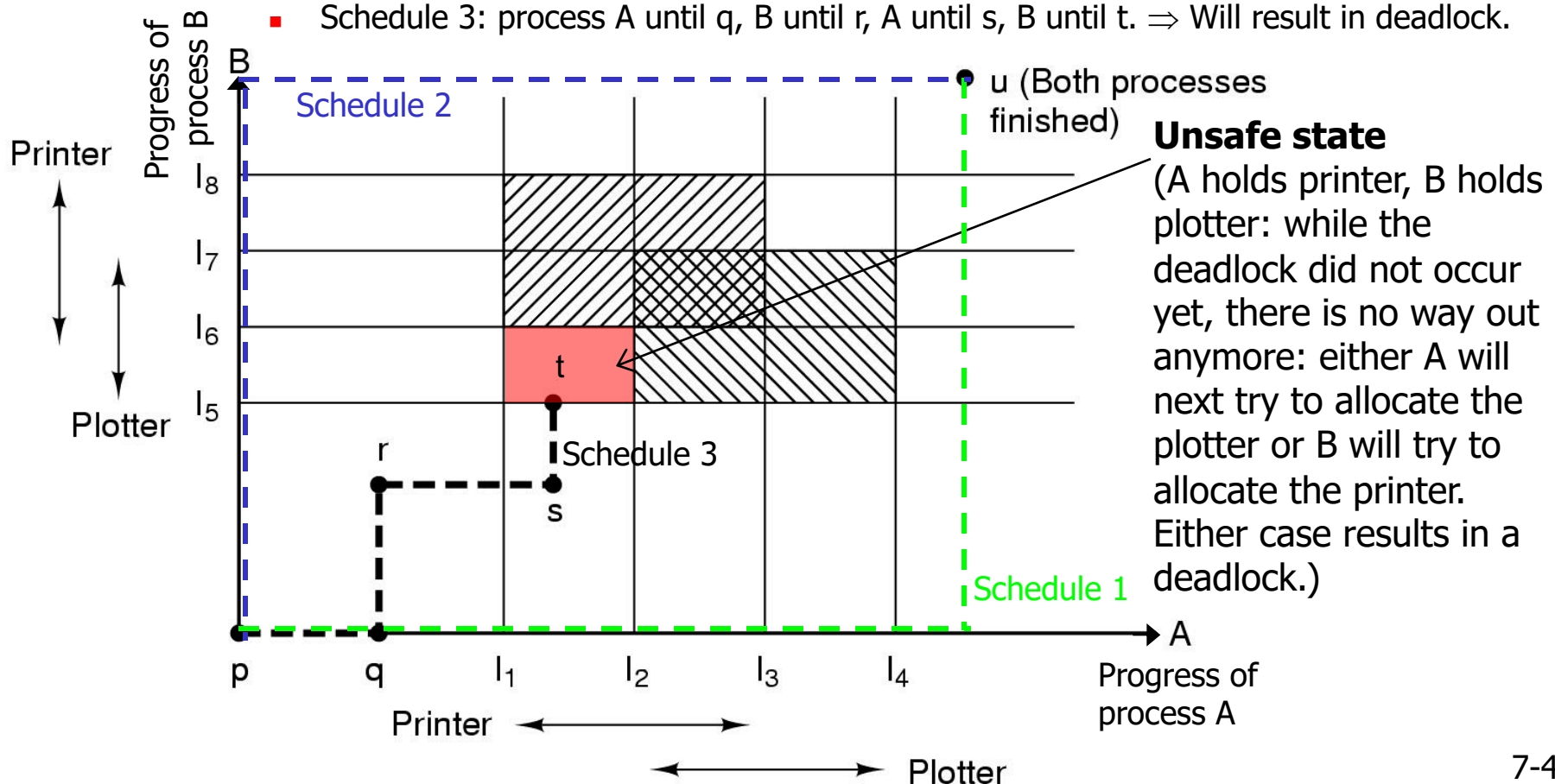
- Alternative to process termination: **Preempt resource from one process and give it to a process that is requesting it.**
 - Process and resource to be preempted need to be selected using some criteria (based on similar factors as on previous slide).
 - Resource preemption **not easy to implement**: process cannot simply continue execution after a resource has been preempted.
 - Solution: processes have to save periodically their current state ("checkpoint"). Then, a **rollback** can be performed back to a checkpoint where the preempted resource has not been requested yet.
 - Avoid **starvation**: it is likely that the preempted process tries to request the preempted resource again. Depending on the progression of the other processes this may result in a deadlock again. If the same factors for selecting a resource and process as last time are used, it is likely that the same resource gets preempted again from the same process ⇒ starvation.
 - Can be avoided by including number of preemptions into selection factors.

7.6 Deadlock Avoidance

- On slide 7-11, we have seen a schedule for two processes that leads to a deadlock and another schedule that leads to no deadlock for the same two processes.
 - If we were able to take only those schedules that lead to no deadlock, we would be able to avoid deadlocks.
- **Deadlock avoidance:** order processes and their resource requests (that potentially might lead to a deadlock) in a way that deadlocks do not occur (by using a clever schedule).
 - If processes announce their future resource requests in advance, the scheduler can take this into account and select processes to be executed based on this information to create schedules that avoid deadlocks.
 - I.e. as a result, a resource request of a process would not be granted, but the process would be blocked during resource request (**before** giving it the resource!) until another process that involves the same resources terminates.

Example (single core system): Trace of Three Different Schedules

- Process A will allocate printer at I1, allocate plotter at I2, release printer at I3, release plotter at I4.
- Process B will allocate plotter at I5, allocate printer at I6, release plotter at I7, release printer at I8.
- Schedule 1: first process A, then process B. \Rightarrow No deadlock.
- Schedule 2: first process B, then process A. \Rightarrow No deadlock.
- Schedule 3: process A until q, B until r, A until s, B until t. \Rightarrow Will result in deadlock.



Safe States and Unsafe States

- Definition **safe state**:

- A system is in a safe state if there exists an order of resource allocations (including release of resources once a process finishes) so that all requests of all processes can be satisfied.
 - Note: it is *not* necessary, that *all* different schedules fulfil this property. For a safe state, it is sufficient that at least *one* schedule exists that is able to fulfil all allocation requests.

- Definition **unsafe state**:

- A system is in an unsafe state if there exists no order of resource allocations (including release of resources once a process finishes) so that all requests of all processes can be satisfied.
 - Note: In an unsafe state, a system has not deadlocked yet, however all possible schedules will unavoidably lead to a deadlock state.
(Because processes will not go backwards.)

Safe States and Unsafe States: Examples (1)

- To be able to select schedules so that unsafe states are avoided, the **maximum amount of requested resources** of the processes **needs to be known in advance** (for each resource type).
- **Example:** Process A will request a maximum of 9 instances of a resource, B a maximum of 4, C a maximum 7 instances of the resource.
 - Processes need to use system calls to specify in advance the maximum number of a requested resource, e.g. amount of needed memory:
 - A: `max_memory(9)`; B: `max_memory(4)`; C: `max_memory(7)`;
 - Then, processes may use at any time system calls to allocate/release, e.g.:
 - A: `mem_alloc(3)`, `mem_alloc(6)`, `mem_release()`;
 - B: `mem_alloc(2)`, `mem_alloc(2)`, `mem_release()`;
 - C: `mem_alloc(2)`, `mem_alloc(5)`, `mem_release()`;
 - OS may now decide for each resource request whether to grant it immediately or rather block the calling process on scheduler level (and run another process instead) until a request would be safe (because another process releases some resources).

Safe States and Unsafe States: Examples (2)

- **Example:** 10 instances of the same resource.
 - Process A will request a maximum of 9 instances of the resource, B a of maximum of 4, C a maximum 7 instances of the resource.
 - Currently (step 1. shown below), 3 instances of the resource are free and A has 3, B has 2, C has 2 instances of the resource, i.e. in the worst case, A will request 6 further, B 2 further, C 5 further instances of the resource.
 - **A possible schedule that allows to satisfy all future request is:** grant request of B for 2 further instances (step 2a.) allowing B to finish and release its resources (step 2b.), then grant request of C for 5 further instances (step 3a.) allowing C to finish and release its resources (step 3b.), now A could request its 6 further instances of the resource. ⇒ **States 1.) to 3b.) are safe states!**

1.)	Has	Max	2a.)	Has	Max	2b.)	Has	Max	3a.)	Has	Max	3b.)	Has	Max
A	3	9	A	3	9	A	3	9	A	3	9	A	3	9
B	2	4	B	4	4	B	0	–	B	0	–	B	0	–
C	2	7	C	2	7	C	2	7	C	7	7	C	0	–
Free: 3			Free: 1			Free: 5			Free: 0			Free: 7		

Safe States and Unsafe States: Examples (3)

- Now, imagine the same initial situation (step 1.) as on the slide before, but where we would have chosen a schedule where we allowed process A to request first 1 further instance of the resource (step 2.).
 - From step 2.) on, the best possible schedule would be to grant request of B for 2 further instances (step 3a.) allowing B to finish and release its resources (step 3b.) However, now we would not be anymore able to satisfy a request from A for 5 further instance nor from C for 5 further instances because only 4 instances are left.
- ⇒ While **state 1.) is still a safe state**, the inappropriate decision to grant process A one further instance of the resource leads to **steps 2.) and beyond being unsafe states**.
- This would not have been happened, if we would have first scheduled process B and then process C as on the previous slide!

1.) Has Max

A	3	9
B	2	4
C	2	7

Free: 3

2.) Has Max

A	4	9
B	2	4
C	2	7

Free: 2

3a.) Has Max

A	4	9
B	4	4
C	2	7

Free: 0

3b.) Has Max

A	4	9
B	—	—
C	2	7

Free: 4

Safety Algorithm: Idea

- How to determine a schedule order that keeps processes in a safe state?
 - The example on slide 7-45 already demonstrates how to proceed:
 - If the currently available resources are sufficient to satisfy the maximum resource request of a process \Rightarrow give resources to that process and this process will eventually finish thus releasing even more resources that can then be used to satisfy further, even larger resource requests.

Safety Algorithm

- In fact, this idea is exactly the approach of the matrix-based deadlock detection algorithm from slides 7-25 to 7-35:
 - Just replace the Request matrix R by the Need matrix N as we do not consider current requests, but the maximum further needed instances of a resource:
 - Need matrix $N := \text{Maximum matrix } M - \text{Allocation matrix } C$,
 - where the Maximum matrix M describes how many instances of each resource type will at most be used by each process.
 - Then apply, matrix based deadlock detection algorithm.
 - Resulting order of process marking by that algorithm describes a schedule that is safe.
 - In unsafe state, if at least one process remains unmarked.

Safety Algorithm: Example (based on slide 7-27)

- Example: Currently the system is in the state described below, e.g.:
P1 holds 1 printer and may in future need up to 2 hard disks and 1 scanner.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of
each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource
are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Need matrix N

(describes how many instances of each resource
each process may need in addition):

$$N = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3

- Now, we want to know: is a safe schedule possible to satisfy all future requests of P1-P3 and, if yes, how does this safe schedule look like? \Rightarrow Apply algorithm from 7-26, however request matrix R is now called need matrix N. **Safe schedule exists, if all processes are marked: schedule is the order of marking** (for above example, safe schedule is P3,P2,P1).

Banker's Algorithm: Introduction

- The matrix-based safety algorithm yields only one possible order of a safe schedule of resource requests.
- However, in practise, processes may make resource requests in a different order and in different chunks.
- How to decide whether to grant a current request or whether to block a request until some other process has released further resources?
⇒ Banker's algorithm! (Dijkstra, 1965)
- (Name comes from the idea that a banker should only grant a request for a loan if enough money remains to satisfy further future requests of that client who might finally be able to pay back only if all loans are granted.)

Banker's Algorithm

- Having the matrix-based safety algorithm, the banker's algorithm is simple:
 - 1. Pretend that the currently pending request has been granted, i.e. update
 - Available resources vector A ,
 - Allocation matrix C , and
 - Need matrix N accordingly.
 - 2. Apply the matrix-based safety algorithm to decide if the resulting state is safe.
 - If yes: grant request (and keep A , C , N as modified in step 1 above.).
 - If no: reject request (i.e. put process to sleep without assigning resource to it even though resource would be available) and restore matrices A , C , N .
- Once resources are released by another process: re-run banker's algorithm to see if it would now be safe to grant the previously rejected request and wake up process that was put to sleep.

Banker's Algorithm: Example

- Example: Currently the system is in the state described below, e.g.:
P1 holds currently no resources, but P2 and P3 do.

Existing resources vector E

(describes how many instances of each resource exist
– not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of
each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 1 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource
are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Need matrix N

(describes how many instances of each resource
each process may need in addition):

$$N = \begin{pmatrix} 2 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3

- Now, let's assume P1 requests 1 printer (and may in future need in addition 2 hard disks and 1 scanner): The OS now, needs to decide whether to grant this request or not.
⇒ Apply steps of Banker's algorithm → next slide.

Banker's Algorithm: Example (based on slide 7-27)

- Step 1: Pretend that the currently pending request (=1 printer) has been granted, i.e. update Available resources vector A, Allocation matrix C, and Need matrix N accordingly.

Existing resources vector E

(describes how many instances of each resource exist – not really needed by algorithm, just for the record):

$$E = \begin{pmatrix} 4 & 2 & 3 & 1 \end{pmatrix}$$

Hard disk CD drive Printer Scanner

Available resources vector A

(describes how many instances of each resource are currently available/free):

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \end{pmatrix}$$

Allocation matrix C

(describes how many instances of each resource are currently allocated to each process):

$$C = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{pmatrix}$$

Need matrix N

(describes how many instances of each resource each process may need in addition):

$$N = \begin{pmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{pmatrix}$$

Processes:

P1
P2
P3

- Step 2: Apply the matrix-based safety algorithm (from 7-48) to decide if the resulting state is safe. If yes: grant request (and keep A, C, N as modified in step 1.). If no: reject request (i.e. put process to sleep without assigning resource to it even though resource would be available) and restore matrices A, C, N. (for above example: grant request!)

7.7 Deadlock Prevention

- By ensuring that one of the four conditions necessary for deadlocks (slide 7-13) does not hold, deadlocks can be prevented. Reminder of four conditions:
 - **Mutual exclusion:** only one process at a time can use a resource (non-sharable 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 $\{P_0, P_1, \dots, P_n\}$ 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 .

Deadlock Prevention:

Mutual Exclusion

- By definition, non-shareable resources must be accessed in a mutual exclusive way.
 - However, take care to **consider a resource only as non-sharable where necessary**: e.g. a read-only file can be shared.
 - **Spooling** may be used for some output devices (e.g. printers).
 - Instead of having to wait for a mutual exclusive access to a printer, send print job to a spooler system process that buffers print jobs.
 - Only spooler process accesses printer in an exclusive way (spooler process can be accessed in a shared manner) \Rightarrow deadlock prevented!
 - Spooling is not applicable for all resources. However, the underlying idea is good:
 - Access resource directly (=mutual exclusive) only where necessary. As often as possible, use some abstraction layer that is, e.g., able to re-order accesses.

Deadlock Prevention: Hold and Wait

- Prevent that processes keep resources while they are waiting for further resources.
- Possible approach:
 - Processes are required to request all needed resources at the beginning of their execution in one atomic step and are not allowed to request (in a hold & wait style) resources in-between.
 - Problem: in particular programs that are supposed to adapt dynamically to growing needs do not know in advance how many resources they need.
 - Solution: Processes are allowed to request resources also in-between, however, they are required to release all of their resources before requesting the new resources (again, the following request of the old and the new resources must be atomic).
 - Works nice as long as it is reasonable to release a resource during usage!

Deadlock Prevention: No Preemption

- In section 7.5 (Deadlock Detection and Recovery), we have seen that **preemption of resources is difficult** (in practise only possible if applications are designed to use checkpointing and rollback) and should therefore only be done as a last resort when a deadlock has already occurred.
 - Resource preemption as a method for deadlock prevention means to be able preempt resources at each resource request (if necessary).
 - “Polite” approach: do not preempt resources from processes that are busy using these resource, but only preempt resources from processes that are waiting anyway. E.g.:
 - Whenever a process has anyway to wait for some other resource, the resources of this waiting process are preempted.

Deadlock Prevention:

Circular Wait

- Circular wait can be prevented if all processes have to request resources in the same order:
 - Impose a total ordering on all resource types and require that each process requests resources only in an increasing order of enumeration.
 - Example: 1. Tape drive, 2. disk drive, 3. printer.
 - Processes that want to allocate a tape drive and a printer, always have to allocate them in the order: first tape drive, then printer.
 - While processes may still have to wait for a resource, **they will never wait it in a circular way!**
 - Problem: while it may be easy to agree on an ordering for all standard resources in a system, each process might use a different ordering for “exotic” custom resources that were not known when the ordering of the standard resources has been defined.

7.8 Summary

- Deadlocks arise whenever non-shareable (**mutual exclusive**) and **non-preemptable** resources are used in a **hold-and-wait** manner and processes are **waiting circular** for resources held by other processes.
 - Assumption: processes do not crash, give resources back after using them.
- Methods to deal with deadlocks:
 - **Ignore**: do not waste efforts with advanced methods.
 - **Detect & recover**: allow deadlocks to occur, try to recover from them once they have been detected (e.g. using **matrix-based detection algorithm**).
 - **Avoid**: only grant resource requests in a way that only safe orderings of requests occur (**Banker's algorithm**), i.e. put processes to sleep if they are about to make unsafe requests. May wake them later up again when resources are releases.
 - **Prevent**: break one of the four deadlock conditions.