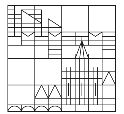
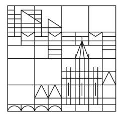
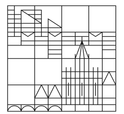


VI.
Deadlocks

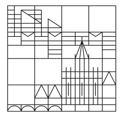


Intended Schedule

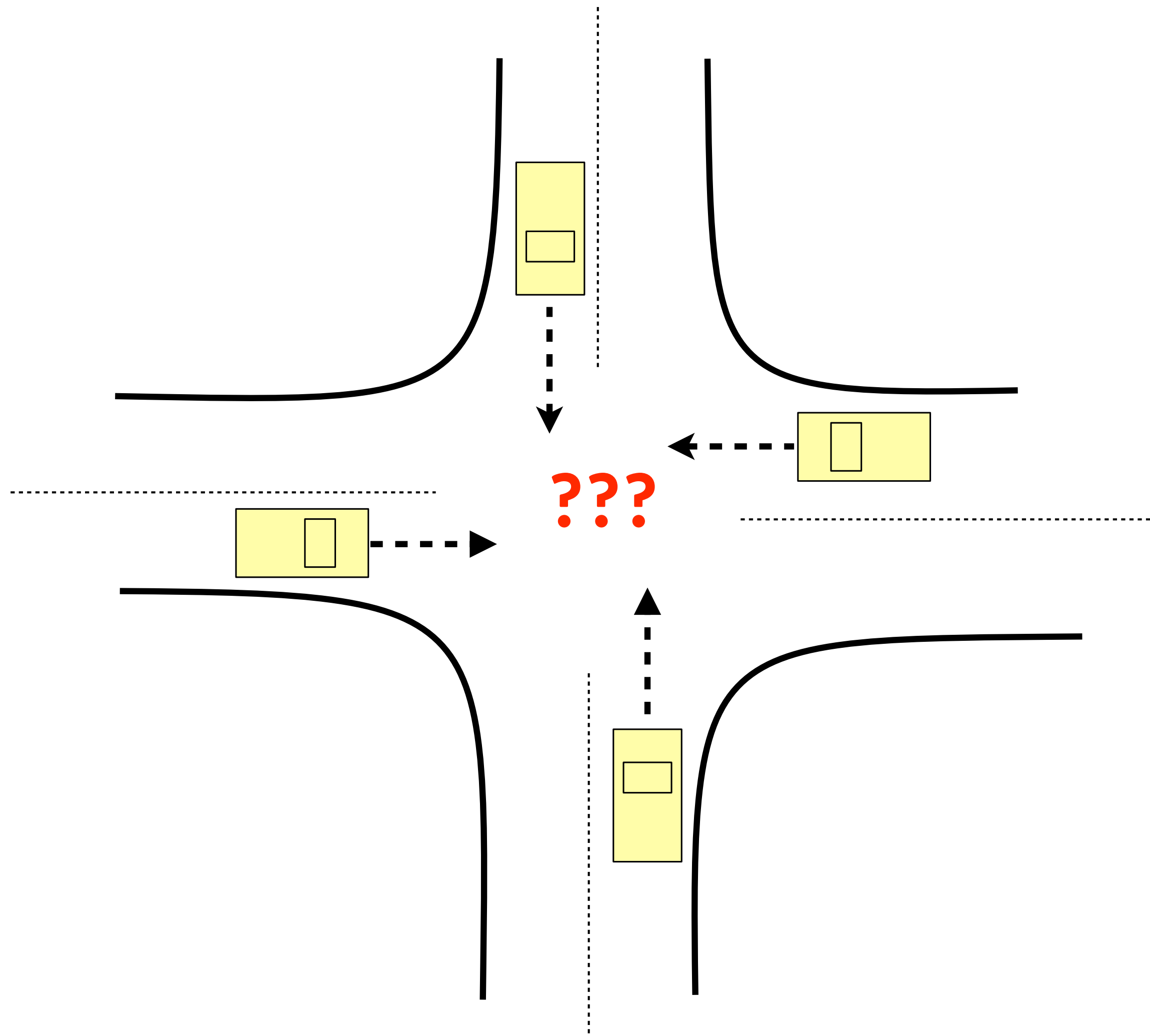
	Date	Lecture	Hand out	Submission
0	20.04.	Introduction to Operating Systems	Course registration	
1	27.04.	Systems Programming using C (File Subsystem)	1. Assignment	
2	04.05.	Systems Programming using C (Process Control)	2. Assignment	1. Assignment
3	11.05.	Process Scheduling	3. Assignment	2. Assignment
4	18.05.	Process Synchronization	4. Assignment	3. Assignment
5	25.05.	Inter Process Communication	5. Assignment	4. Assignment
	01.06.	Pfingstmontag	6. Assignment	5. Assignment
6	08.06.	Deadlocks	7. Assignment	6. Assignment
7	15.06.	Input / Output	8. Assignment	7. Assignment
8	22.06.	Memory Management	9. Assignment	8. Assignment
9	29.06.	Filesystems	10. Assignment	9. Assignment
10	06.07.	Special subject: Transactional Memory		10. Assignment
11	13.07.	Special subject: XQuery your Filesystem		
12	20.07.	Wrap up session		
✏	*27.07.*	First examination date		
	12.10.	Second examination date		

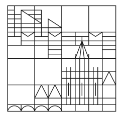


What is a Deadlock?



Deadlock Problem (I)

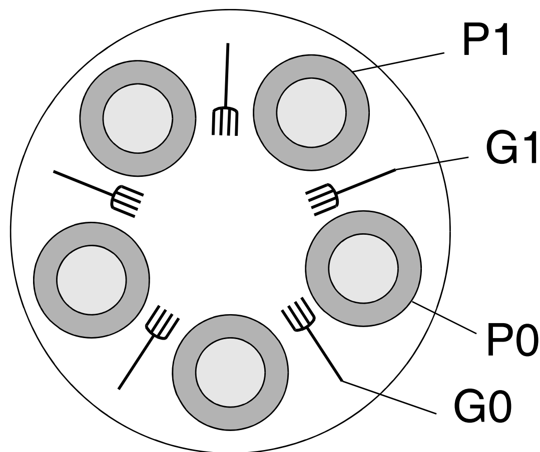




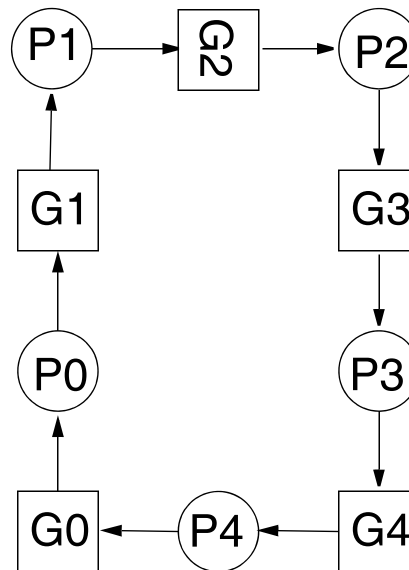
Deadlock Problem (2)

Dining Philosophers

Philosophen-Esstisch

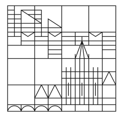


Deadlock-Situation



Aktionen eines Philosophen

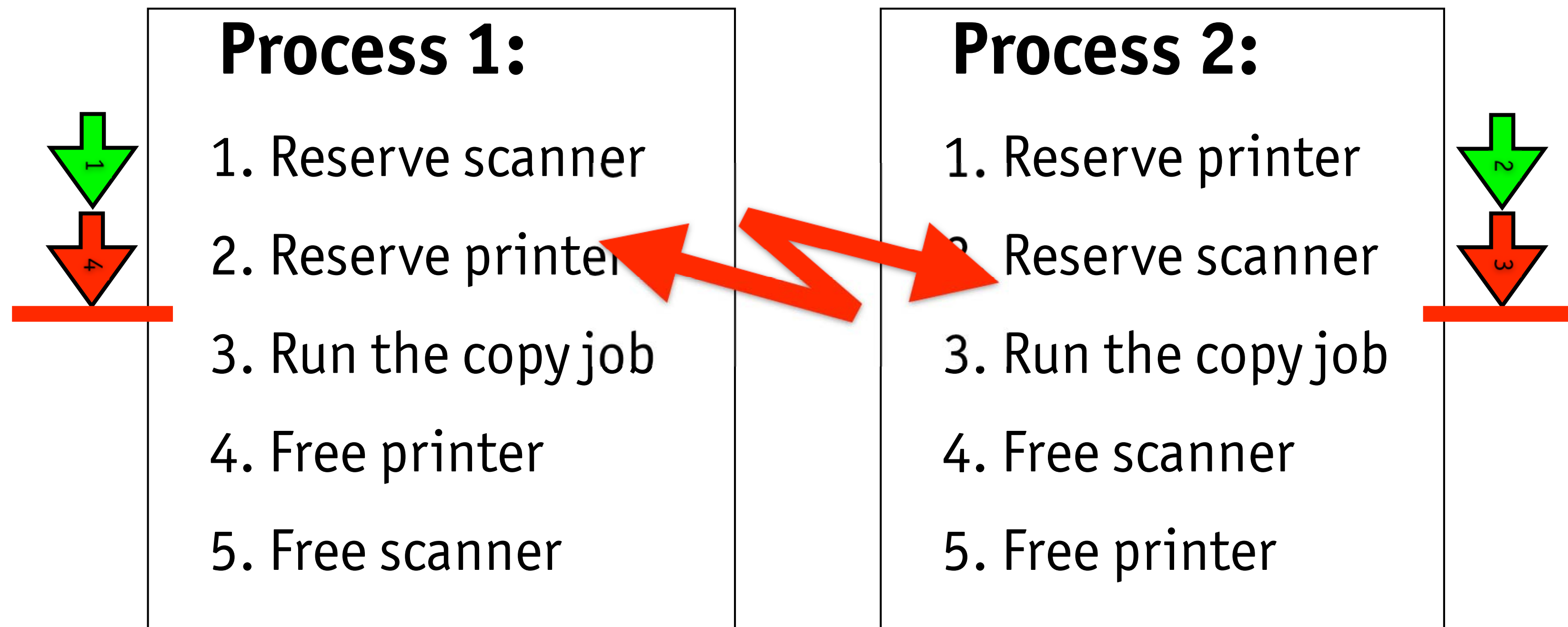
```
void philosoph(int n)
{
   while(1) {
      denken();
      Gabel_nehmen(n);
      Gabel_nehmen((n+1)%5);
      essen();
      Gabel_legen(n);
      Gabel_legen((n+1)%5);
   }
}
```

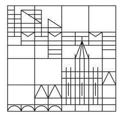


Deadlock Problem (3)

Two users (processes) in an OS want to run a copy job by scanning a page and sending the image to the printer.

They are using different software.





Deadlock Problem (4)

Flashback (Lecture 4 on Synchronization) ...

Deadlock and Starvation

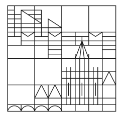
Bad design of synchronisation scheme:

- **Deadlock** – Processes mutually block each other:

```
P1 {
   ...
   wait(s1);
   wait(s2);
   ...
}
```

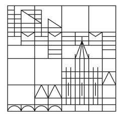
```
P2 {
   ...
   wait(s2);
   wait(s1);
   ...
}
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended



Resources (I)

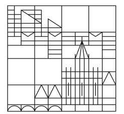
- **Resources** are the general abstraction of „objects“ for which the OS grants access (to processes).
  - devices (such as, printers, optical media drives, ...)
  - pieces of information (a file, a record in a database, an entry in a waiting queue, ...)
- The OS typically manages **many different** resources
  - some of which are available only **once**,
  - while others exist in many (equivalent) „**copies**“
- Resources are
  1. acquired
  2. used
  3. released



Resources (2)

- Different **kinds** of resources
  - *preemptable*
    (can be withdrawn from process before process frees those resources)
  - *non-preemptable*
    (cannot be withdrawn from process without harm)

- Model for the following discussion
  - resources: $R_1, \ldots, R_n$
  - each available in: $w_1, \ldots, w_n$ copies
  - processes: $P_1, \ldots, P_m$



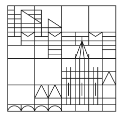
Definition: Deadlock

A set of processes is deadlocked, if each process in the set is waiting for an event that only another process in the set can cause.

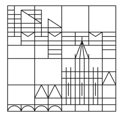
Characteristics (*all four are necessary simultaneously!*):

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

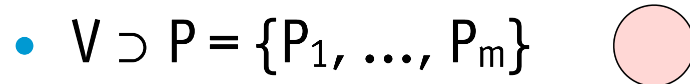
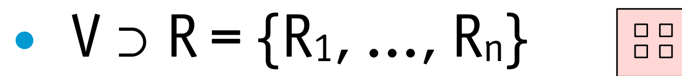
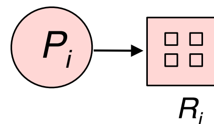
... each of those related to a policy a system may or may not employ.

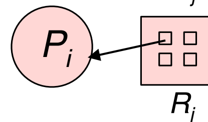


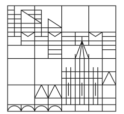
Dealing with Deadlocks



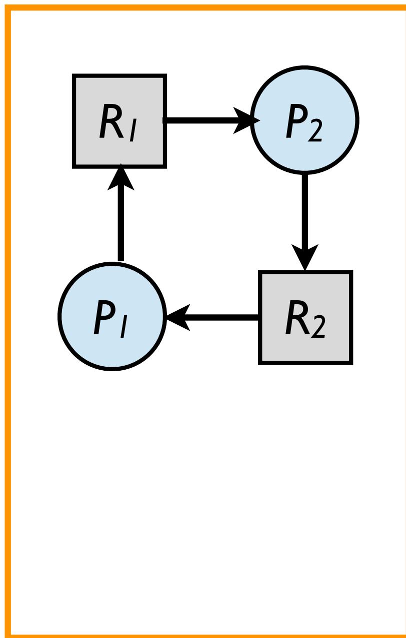
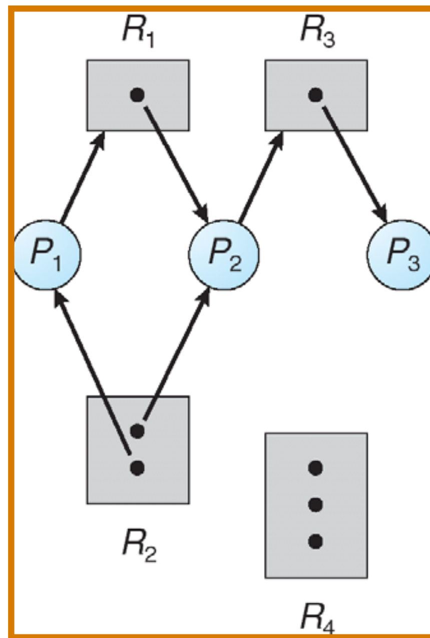
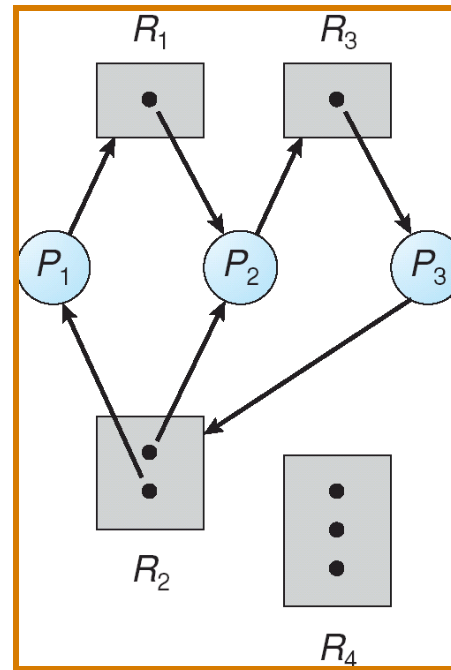
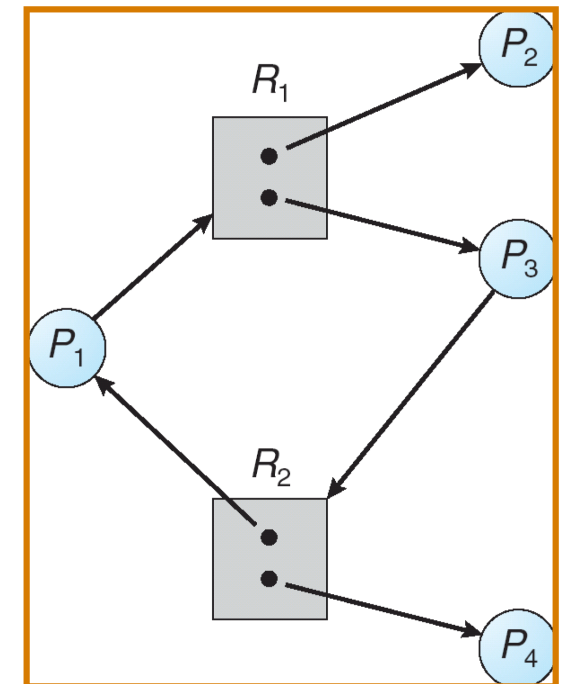
Resource Allocation Graph

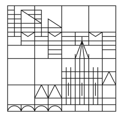
- Directed graph (V,E)
  - with two kinds of nodes V:
    - $V \supset P = \{P_1, \ldots, P_m\}$  [Processes]
    - $V \supset R = \{R_1, \ldots, R_n\}$  [Resources] [w/ multiple copies]
  - and two kinds of edges E:
    - $E \supset P \times R$  [Request edge]

    - $E \supset R \times P$  [Assignment edge]



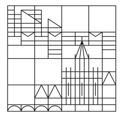
Examples: Resource Allocation Graph

Single copy	Multiple copies	Deadlock	Cycle, but no deadlock
			



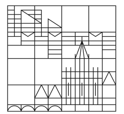
Observations

- No cycle in r.a. graph ⇒ no deadlock
- Cycle in r.a. graph and
  - only 1 copy per resource ⇒ deadlock
  - multiple copies per resource ⇒ possibly a deadlock

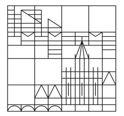


What to do about deadlocks...

- **„Ostrich algorithm“**
  - Ignore the problem [and hope, it won‘t ignore you!]
- **Detection & recovery**
  - Let deadlocks occur, detect them and take proper actions
- **Avoidance**
  - Dynamically take care that no deadlocks occur
- **Prevention**
  - Structurally negate one of the four characteristics

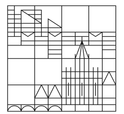


Ostrich Algorithm

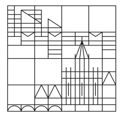


„Ignore" deadlocks

- … cannot fully ignore.
- In many situations, *timeouts* are sufficient:
  - Let processes run, potentially into deadlocks.
  - If no progress—or does not finish—within specified threshold: kill process [and optionally: restart]
- In quite a few cases, especially in networked environments, there is hardly any other choice, anyway.



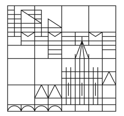
Deadlock Detection & Recovery



Deadlock Detection

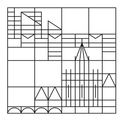
- Allow system to enter deadlock state
- Use detection algorithm
- Resolve situation to recover from deadlock

- Often distinguish two situations:
  1. Only 1 copy per resource
  2. Multiple copies per resource



Only 1 copy per resource

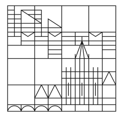
- Remember: cycle in r.a. graph ⇒ deadlock
- Typically generate simpler wait-for-graph from r.a. graph:
  - Nodes = processes
  - Edges = process 1 waits for process 2
- Periodically check wait-for-graph for cycles,
  if cycle ⇒ deadlock
  - [a lot of algorithms for cycle test are available, they need no more than $O(n^2)$ time, for n=#(nodes)]



Wait-for-graph

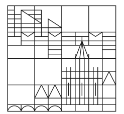
Resource-Allocation Graph

Corresponding wait-for graph



Multiple copies per resource

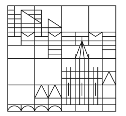
- *m* resources, *n* processes
- Vector $W^{(m)}$ of length *m* indicates # of existing resource copies per type
- Vector $A^{(m)}$ of length *m* indicates # of available (free) resources per type
- Current allocation matrix $C^{(n \times m)}$ holds # of resources of each type, currently held by each process
- Request matrix $R^{(n \times m)}$ holds # of resources of each type, currently requested by each process
- Invariant: $$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$



Worst-Case Algorithm

- *Worst-case:* assume each process holds all resources until end-of-process.
- **Idea:** iterate through all processes and find those, whose additional requests could be satisfied.
  - Assume they were run to completion, free all resources (i.e., add them to the available matrix *A*), and exit the system („mark" them as completed).
  - Proceed until no more such processes can be found.
  - Upon termination of the algorithm, any remaining unmarked processes are deadlocked.
- Notation: vector comparison

$$A \leq B \iff \forall i : 1 \leq i \leq m : A_i \leq B_i$$



Deadlock Detection Algorithm

1. Look for an unmarked process, P_i, for which $R[i] \leq A$.
2. If such a process, P_i, is found:
   - add ith row of C to A: $A \leftarrow A + C[i]$.
   - mark process P_i.
   - go back to step 1.
3. If no such process exists, terminate the algorithm.

All remaining unmarked processes, if any, are deadlocked.

- Even though non-deterministic (order of processes considered), the algorithm always returns the same results.

Example of Detection Algorithm

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

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

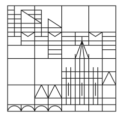
- Sequence <P_0, P_2, P_3, P_1, P_4> will result in *marked*[*i*] = true for all *i*.

Example (Cont.)

- P_2 requests an additional instance of type C.

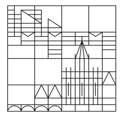
	Request R
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

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



When to run such algorithms?

- Every time a resource is requested
  - … very expensive
- Periodically, e.g., every *k* minutes
  - … still quite expensive, if deadlock situations are rare
- Only when CPU utilization drops below a certain threshold
  - … when massive deadlock situations occur, few processes remain ready to run

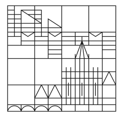


Recovering from a deadlock

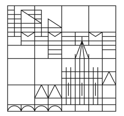
- In rare cases, it may be possible to preempt a resource.
  - Often requires manual intervention, thus very expensive
- Rolling back a process
  - When deadlocks are frequent, processes should be implemented in a checkpointed fashion, periodically writing state (& resource!) information to a file.
  - Processes might then be reset to an earlier checkpoint (one that does not include the deadlock-resource) and restarted from there after other processes involved in the deadlock have finished.
- Killing a process
  - Easiest, but most harmful way out.
  - Carefully select victim (need not be one involved directly in the deadlock).

Recovery from Deadlock: Victim Selection

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- In which order should we choose to abort?
  - Priority of the process.
  - How long process has computed, and how much longer to completion.
  - Resources the process has used.
  - Resources process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
- Avoid starvation: repetitive selection of same process as deadlock victim should be avoided (e.g., include # of restarts in cost function)



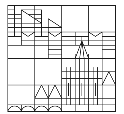
Deadlock Avoidance



Deadlock Avoidance

- The algorithm above tacitly assumes that all resources are requested at once. This is typically not the case in practice.
- With incremental resource requests, the system may try to grant only „*safe*“ allocations.
- **Question:** Is there an algorithm that can decide, dynamically, whether granting a resource request is *safe*? Is it possible to avoid deadlocks that way?

- *Answer:* Yes—but only, if enough information is available.
  - Typically, one assumes that the system knows the *maximum* resource requests for all resource types of all processes.



Safe States

- Basic idea: System tries to stay in „safe states“
- Example: 2 processes, 2 resources

Target: both processes finished

Progress of P_1

P_1 holds R_1

P_1 holds R_2

P_2 holds R_1

P_2 holds R_2

Progress of P_2

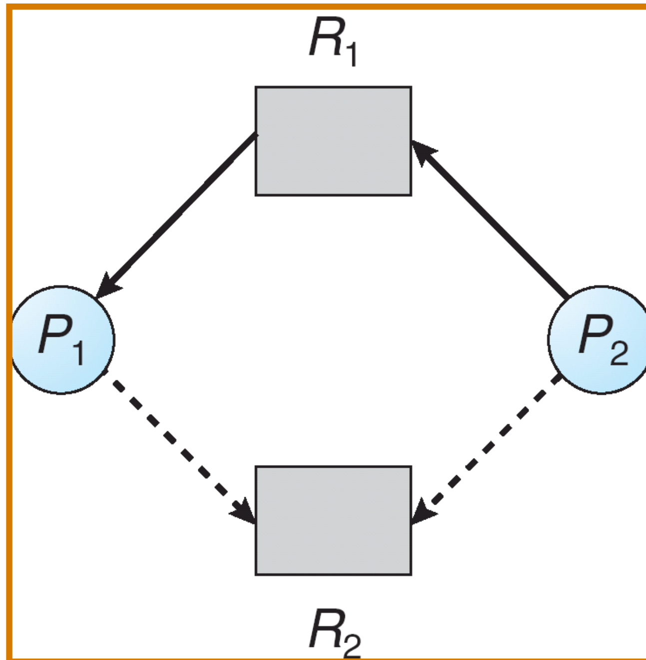
Avoid unsafe area

Single resource of each type

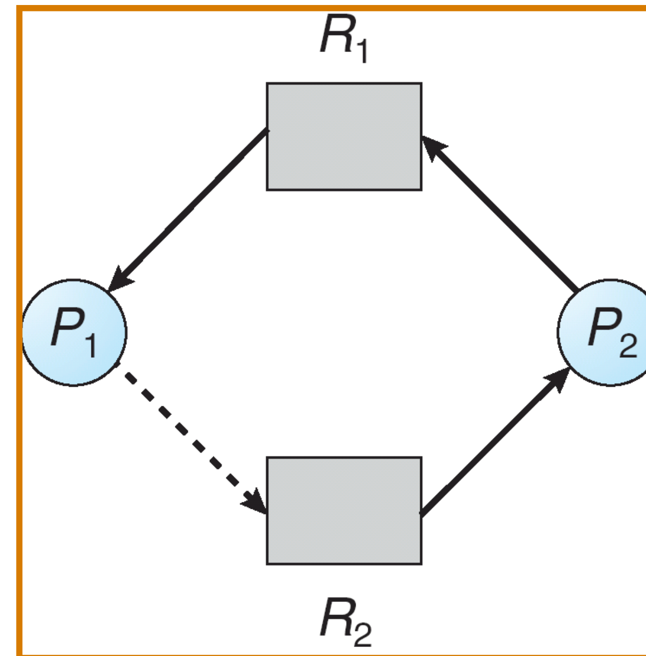
Use modified resource allocation graph method:

- *Claim edge* $P_i \rightarrow R_j$ indicates that process P_j *may* request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- Request edge converted to an assignment edge when the resource is allocated to the process.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed *a priori* in the system.

Resource-Allocation Graph

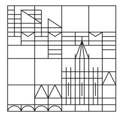


Safe State



Unsafe State

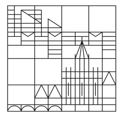
- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



Multiple copies: Banker's Algorithm

- Additional matrix $Max^{(n \times m)}$ holding maximum claim of resources of each type, possibly requested by each process.
  1. Look for an unmarked process, P_i, for which $Max[i]\text{-}R[i] \le A$.
  2. If such a process, P_i, is found:
     - add ith row of C to A: $A \leftarrow A + C[i]$.
     - mark process P_i as terminated.
     - go back to step 1.
  3. If no such process exists, terminate the algorithm.

If processes remain unmarked, there is a deadlock; additional requests of these processes could not be met.



Resource-Request Handling

- $R[i,j]=k$ means: process P_i requests k (more) instances of resource R_j.
- Algorithm:
  1. If $R[i] > Max[i]$,
     then signal an error: process exceeds its maximum claim
  2. If $R[i] > A$,
     then process must wait (not enough resources available)
  3. Assume the request were granted and tentatively compute:
     - $A \leftarrow A - R[i]$;
     - $C \leftarrow C + A$;
     - $Max[i] \leftarrow Max[i] - R[i]$;
     - If safe: actually allocate resources to P_i,
       else: P_i must wait, discard tentative changes of step 3.

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 C	Max	Available A
	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 (Cont.)

- The content of the matrix *Request R* is defined to be *Max – Allocation*.

	Request R
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

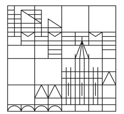
- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria.

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.

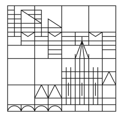
	Allocation C	*Request R*	*Available A*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement.
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?



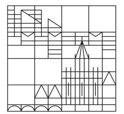
Deadlock Prevention

Universität Konstanz



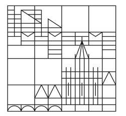
Deadlock Prevention

- Deadlock avoidance is essentially impossible in practice, since the OS cannot know the maximum resource claims of processes a priori!
- Potentially, we can prevent deadlocks by structurally negating one of the four characteristics of deadlock situations:
  - **Mutual exclusion.**
  - **Hold and wait.**
  - **No preemption.**
  - **Circular wait.**



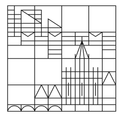
Preventing Deadlocks (I)

- Attack the **mutual exclusion** condition.
  - For example: spool printer output to disk file; only printer daemon requests actual printer resource, but no other resources.
  - Yet: spooling space may become subject to deadlock...
  - Generally: there are a lot of sharable resources, or those that can be „made“ sharable.
- Attack **hold and wait** condition.
  - Require processes to request all resources before start, or when no others are held.
  - Low resource utilization, danger of starvation.



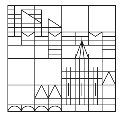
Preventing Deadlocks (2)

- Attacking the **no preemtpion** condition.
  - 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.
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Attacking the **circular wait** condition.
  - Impose a total order on all (copies of all) resources.
  - Require all processes to request resources in ascending order only.



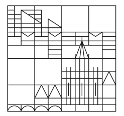
Preventing Deadlocks – Summary

Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away (temporarily)
Circular wait	Order resources numerically



Intended Schedule

	Date	Lecture	Hand out	Submission
0	20.04.	Introduction to Operating Systems	Course registration	
1	27.04.	Systems Programming using C (File Subsystem)	1. Assignment	
2	04.05.	Systems Programming using C (Process Control)	2. Assignment	1. Assignment
3	11.05.	Process Scheduling	3. Assignment	2. Assignment
4	18.05.	Process Synchronization	4. Assignment	3. Assignment
5	25.05.	Inter Process Communication	5. Assignment	4. Assignment
	01.06.	Pfingstmontag	6. Assignment	5. Assignment
6	08.06.	Deadlocks	7. Assignment	6. Assignment
7	15.06.	Input / Output	8. Assignment	7. Assignment
8	22.06.	Memory Management	9. Assignment	8. Assignment
9	29.06.	Filesystems	10. Assignment	9. Assignment
10	06.07.	Special subject: Transactional Memory		10. Assignment
11	13.07.	Special subject: XQuery your Filesystem		
12	20.07.	Wrap up session		
✏	*27.07.*	First examination date		
	12.10.	Second examination date		



VII.
Input/Output