Chapter 9
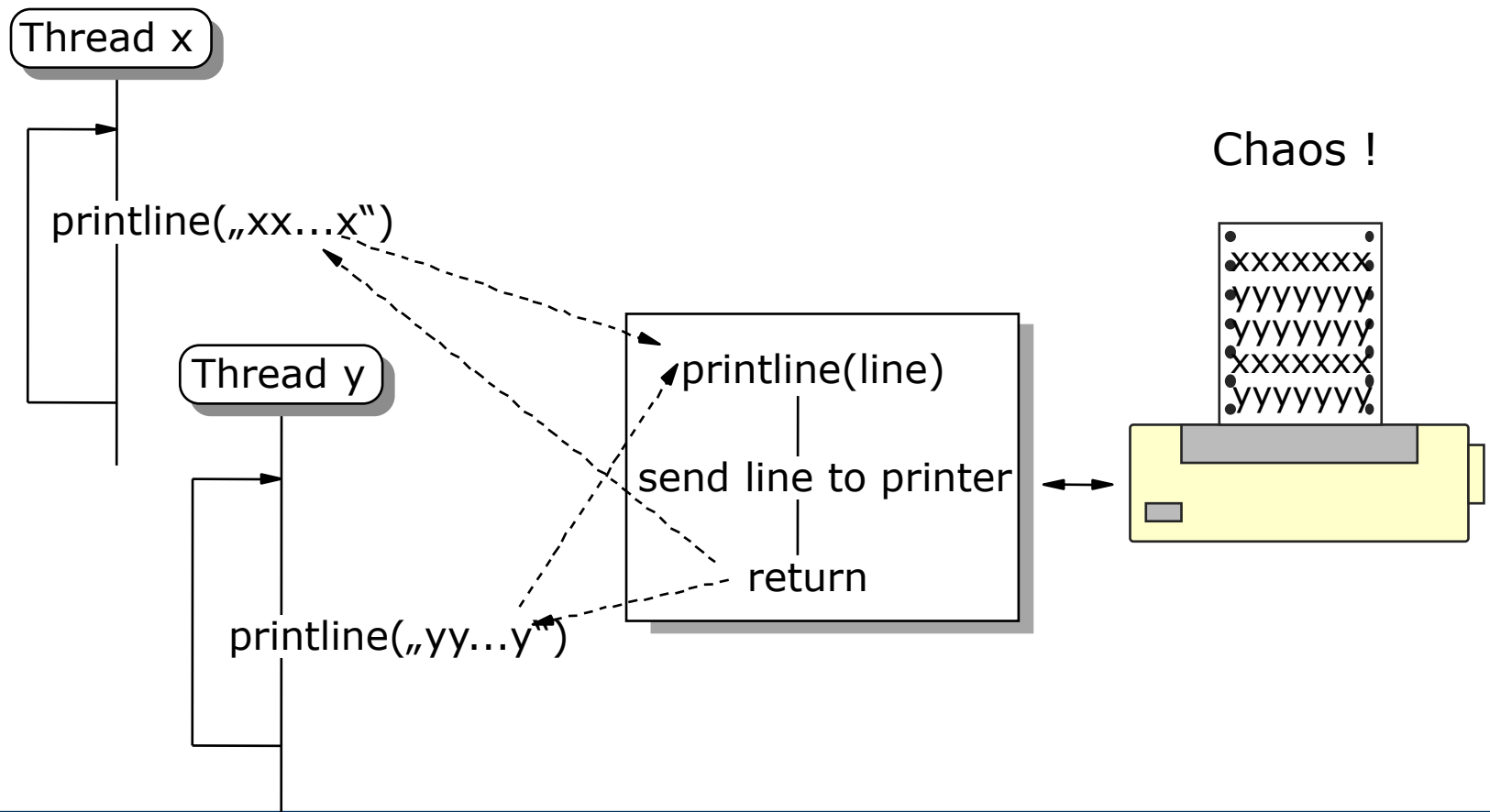
Resource Management

9.1 Introduction and Overview

- Every entity needed by a thread to run can be called a *resource.*
- Resources can lead to problems if they are limited, or if they can used exclusively only.
- There are different kinds of resources and therefore different approaches to manage them.

- Example 1:
  - A thread needs the program code to be executed accessible in main memory.
  - The program code is a resource of the thread.
  - Other threads may execute the same program code. These threads can access the code as well. There is no need to manage the access to the resource.

- Example 2:
  - Threads need memory to store some data.
  - Memory is limited and should be assigned exclusively. Therefore the memory must be managed.
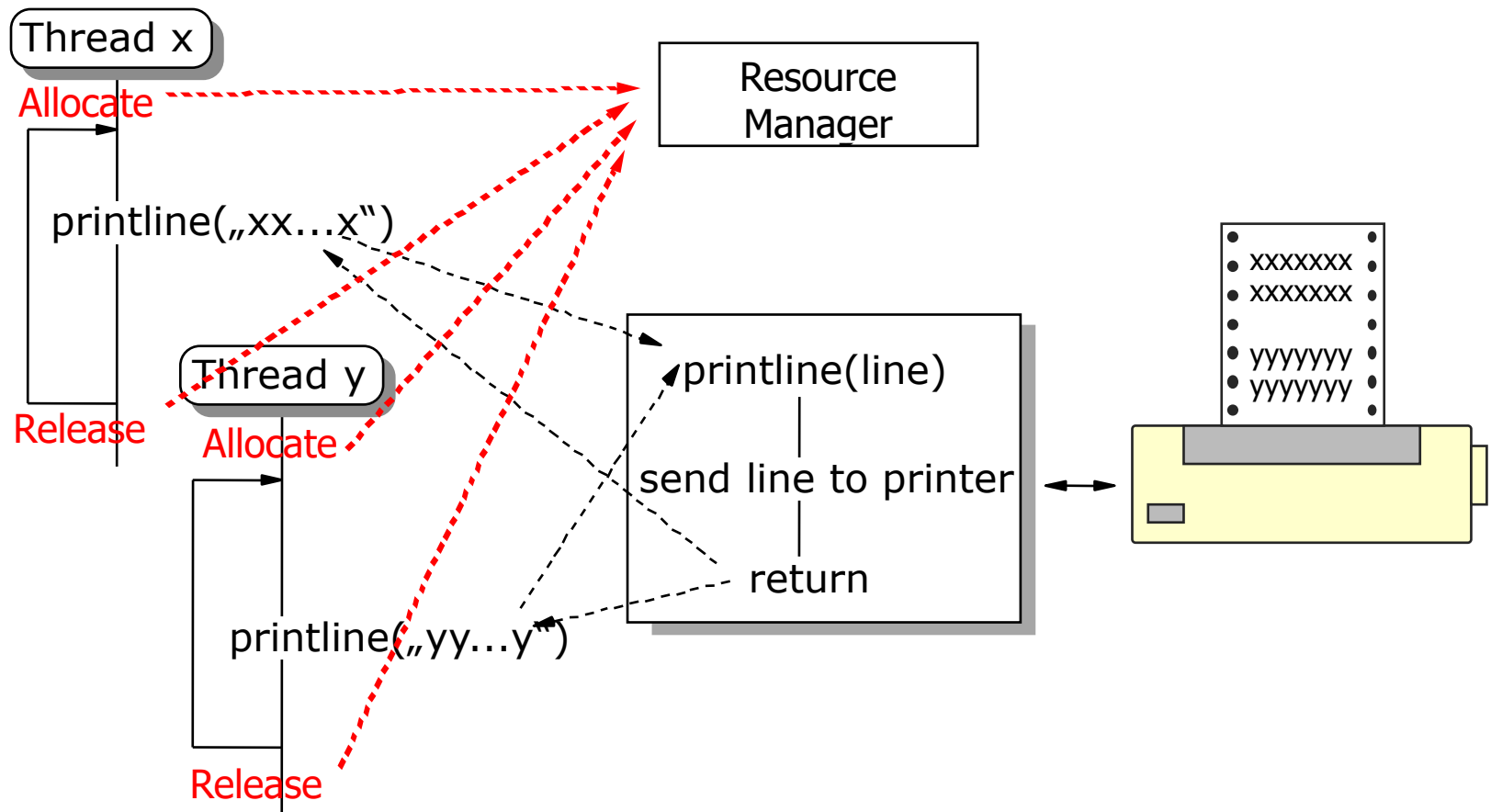
Uncoordinated usage

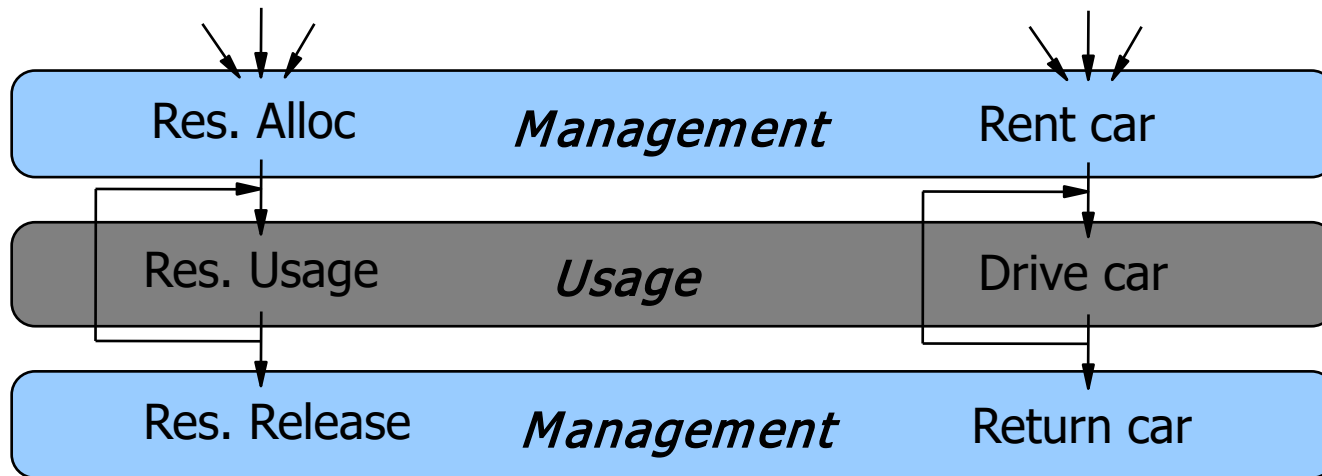- Uncoordinated usage of a resource may lead to undesirable effects.

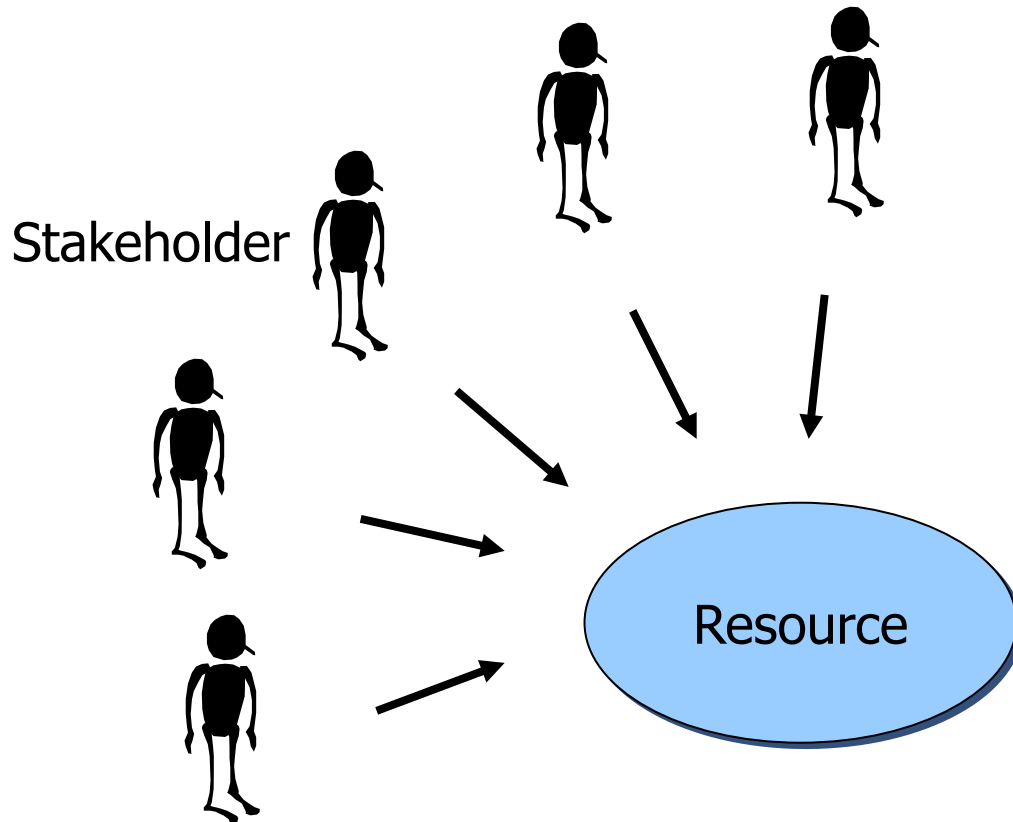# Coordination by Resource Manager

- Using a resource manager for instance the exclusive access can be ensured.

- Separation of usage and management!
- Wrapping usage by management operations!

What is resource management?

Stakeholder

Resource

Stakeholder, e.g.
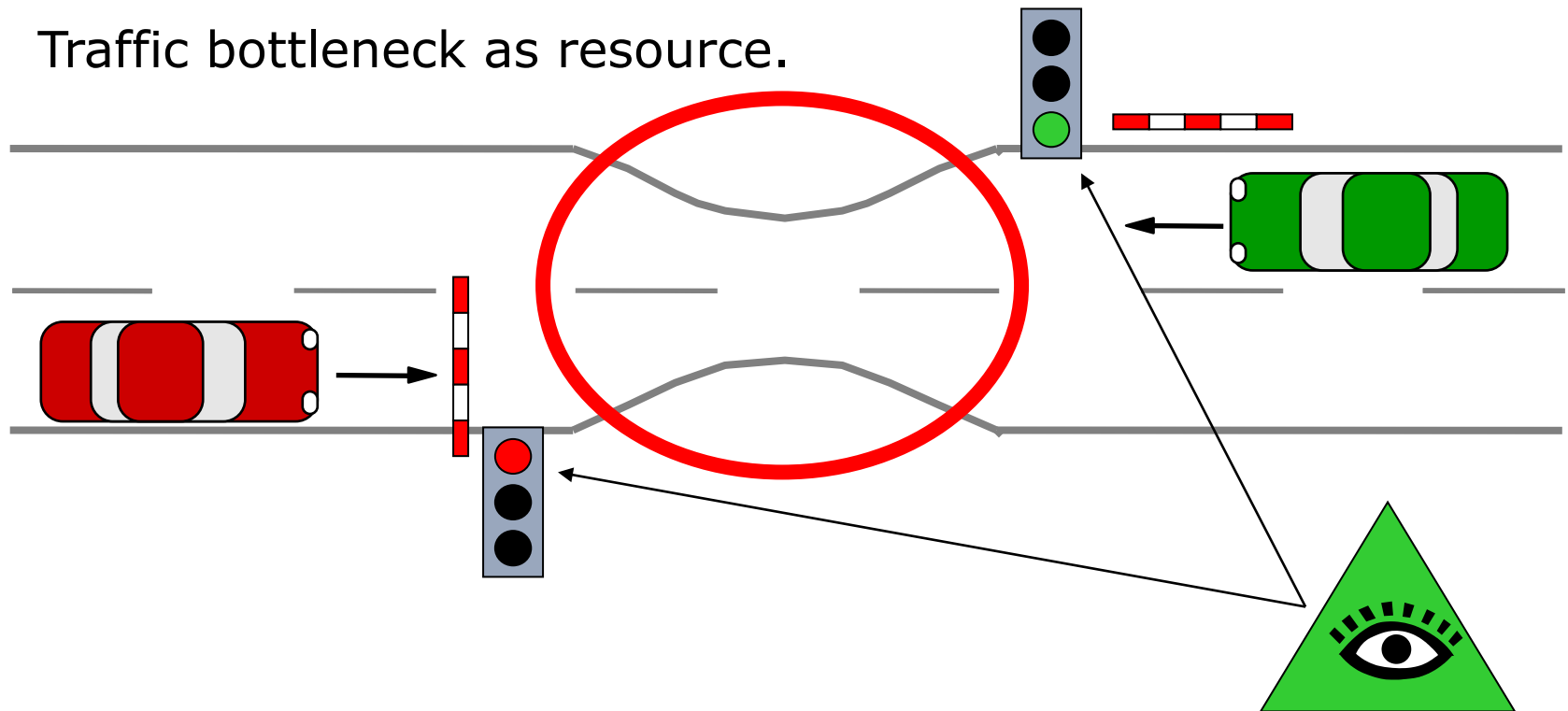 User
  Process
  Thread
  CPU
  Network interface

Resources, e.g.
  CPU
  Memory
  Bandwidth

  File
  Signal
  Message
  Name
  Colour

- Limited amount of resources
- Exclusive usage
- Management is reasonable

Resource management in everyday life

Traffic bottleneck as resource.

- Solution 1:

  Central authority decides (resource management).
  Traffic lights, gates

Solution 1: Resource Management

- Usage of a resource only after **allocation**.
- Allocation before usage **enforced** by intermediate instance.

- Examples
  - 2-phase locking for transaction based systems
    (Access to data as a resource managed exclusively by the scheduler)
  - (Main) memory management
    (Access to allocated segments only)
  - Monitor
    (Call of entry procedure only after release of monitor)
  - Printer
   (Access to printer only via driver as resource manager)

Resource management in everyday life
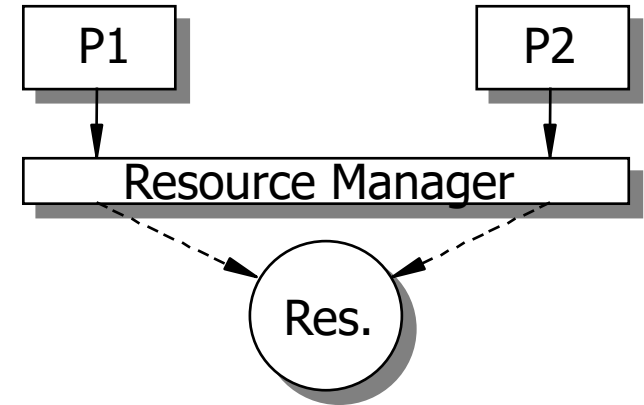
Traffic bottleneck as resource.
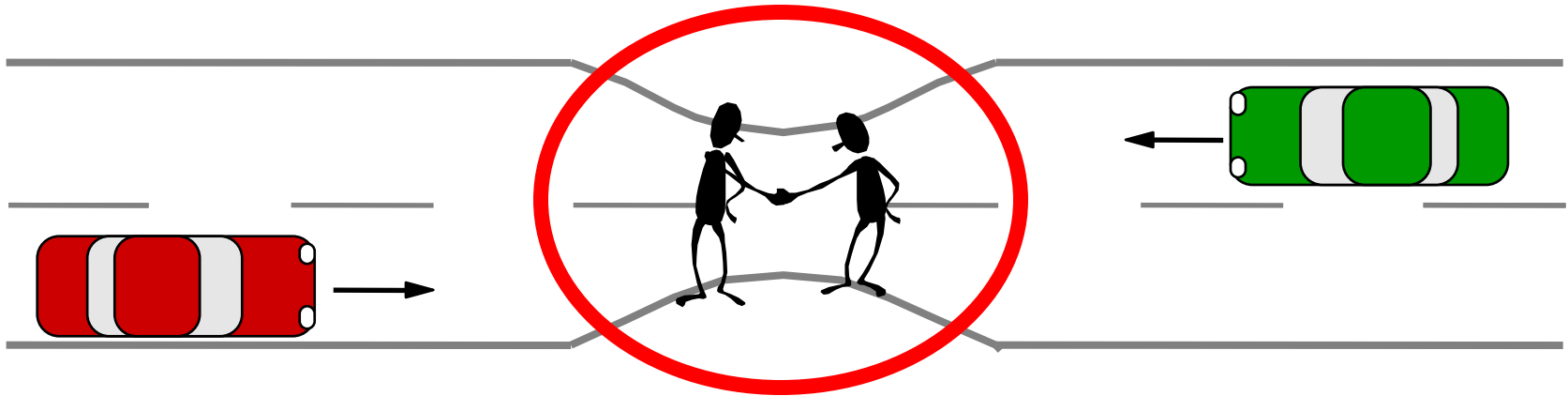
- Solution 2:

Agreement between all parties (rules, negotiation, protocol)

Traffic sign, hand signal, flash light

Solution 2: Agreement

- **Applicants agree about access to resource (protocol).**

- **Critical section**

  Involved processes agree to implement mutual exclusion by usage of lock.

- **Decentralized Bus Arbitration**

  Access to bus as shared communication media must be managed. Agreement about special protocol to coordinate bus arbitration in case of requesting component.

- **Distributed systems – global serialization**

  Nodes apply for global serialization by broadcast. Coordination of access sequence based on logical time.

Resource management in everyday life
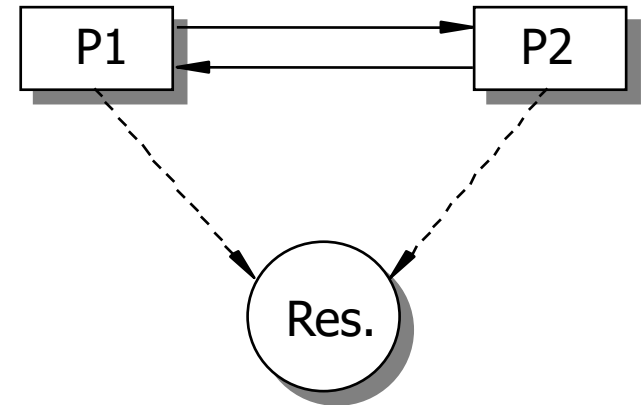
Traffic bottleneck as resource.

- Solution 3:

  No action (uncoordinated usage)

  Risk of collision

# Solution 3: Uncoordinated usage

- Without coordination **collisions** are possible.

- Collisions have to dissolve properly.
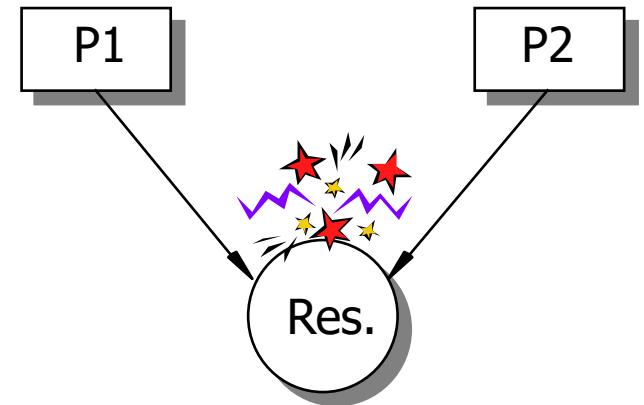
- The effort to handle collisions may be smaller than to avoid collision permanently.

- Can be used in case:
  - Collision is unlikely or seldom and
  - the "damage" done by collision can be fixed afterwards.

- Optimistic synchronization of transactions (Validation)
  - Transaction don't implement locks, but perform access
  - Access is recorded (Log)
  - At the end of transaction (Commit) check for collision (access by different processes) is performed (Validation)
  - Abort and roll-back in case of collision
- LAN (CSMA/CD)
  - Listen to the medium
  - Sending if medium is free, otherwise wait until it's free
  - Listening while sending
  - Packets are destroyed by other packets on the medium
  - In case of collision sender have to wait and send again
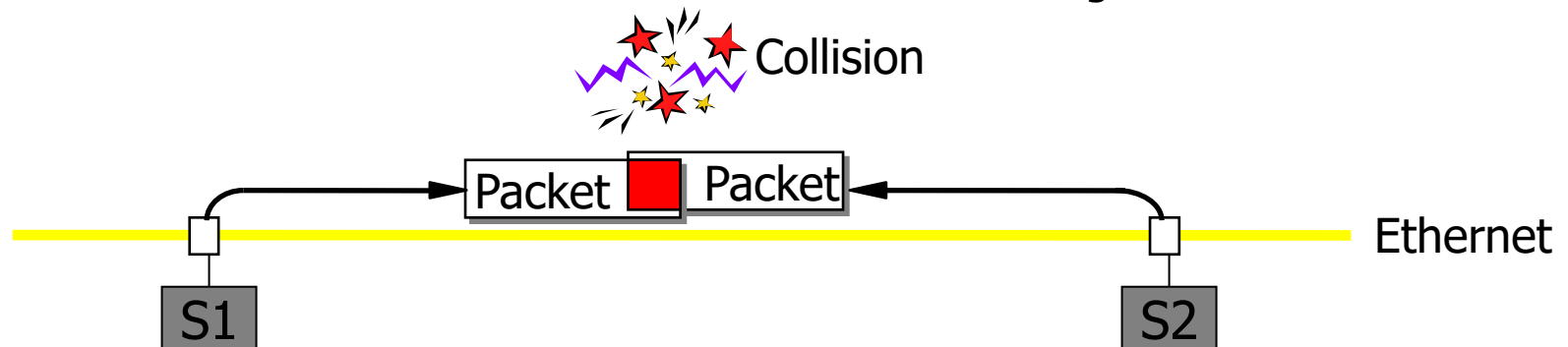
Classification

- Forms
  - **real / logical / virtual**

- *Real resources* are physically existent.

  Real resources are base for virtual and logical resources.

  Examples: Main memory, Disc drive, Processor

- In order to offer more or higher capacity of real resource a *virtual resource* is built.

  Usage is often intermittent, i.e. virtual resource is mapped to the real resource only for short time (Multiplexing).

  Examples: Virtual memory, Virtual processor

- *Logical resources* extend virtual or real resource in order to provide "higher" level of service via comfortable interface or with interface with enhanced functionality.

  Logical resource is kind of abstraction of the real one.

  Examples: File, Window

Classification

- Persistence
  - Reusable

    Resources usually are released after usage and can be used by other processes.
  - Consumable

    Some logical resources are consumed by usage and are not usable afterwards.

    Examples: Signals, Messages, Times stamps
- Capacity
  - Limited

    Amount of usage of the resource has to be managed (allocation/release).
  - Unlimited

    No management of amount of usage needed; management of access only.

Classification

- Acquisition
    - Process requests usage of resource for itself.
    - Some different instance requests allocation of resource for the process (request for memory for forked process).

- Implementation of resource management
    - As procedure

        without parallel execution to the requesting process
    - As process

        with parallel execution to the requesting process

- Resource

 Term for all entities needed by a process to run.

- Resource management

  All tasks before and after usage of a resource needed to implement a correct execution.
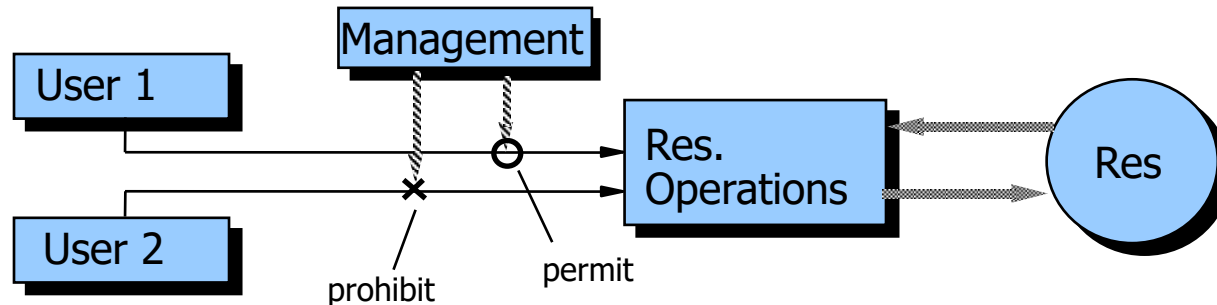
- Goals to resource management
  - Correct execution
  - No deadlock
  - No starvation
  - High level of parallel execution
  - High level of resource usage
- In real live leads to complex optimization problems.
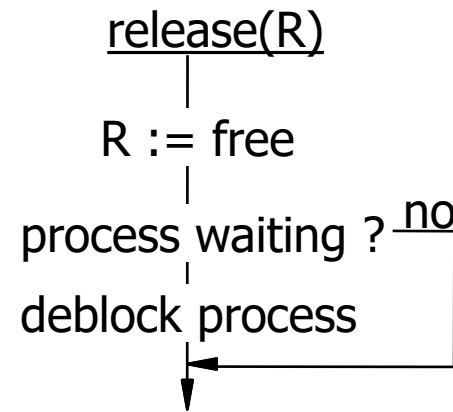
Example: Parallel computer

- Parallel programs run on a given amount of processors or compute nodes for a specific time span: allocate (num_processors, processing_time).
- Requests can be seen as a rectangle (number of processors x time).
- So the search for the optimal utilization is a Bin Packing Problem (Knapsack problem).
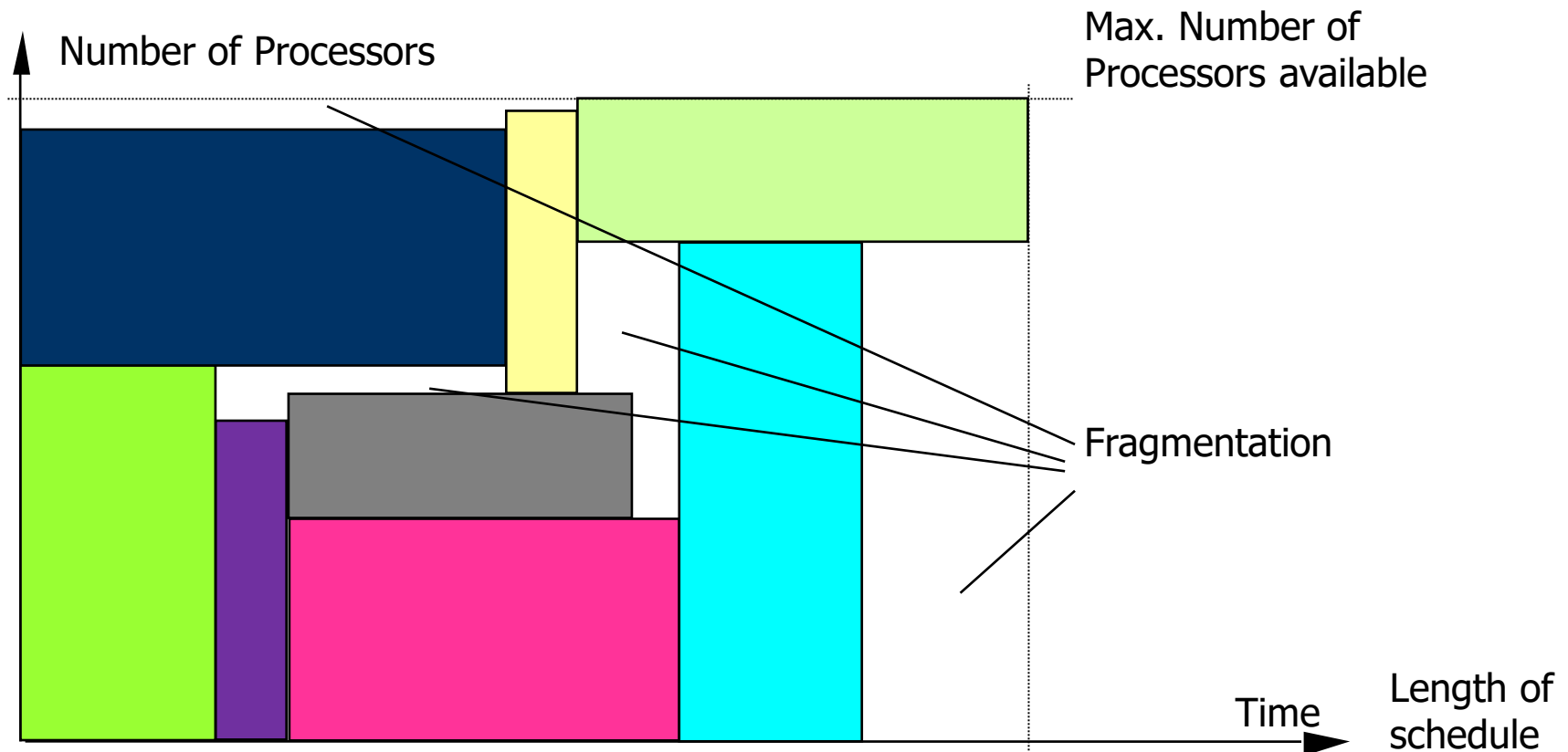
9.2 Central resource management
9.2.1 One-exemplar resources

- Usage of a one-exemplar resource can be seen as critical section.

- So resource management is like handling a coordination problem.

- Operations of resource management *allocate* and *release* do have same structure as operation for coordination at a critical section with *lock* and *unlock*.

allocate(R)

R = free? —yes

block

R := occupied

release(R)

R := free

process waiting ? —no

deblock process

Processes

„waiting"

„has allocated"

allocated

free

Single Resources

Data for resource management
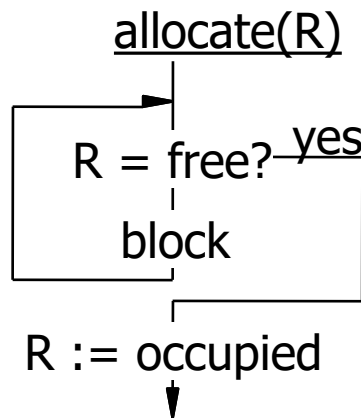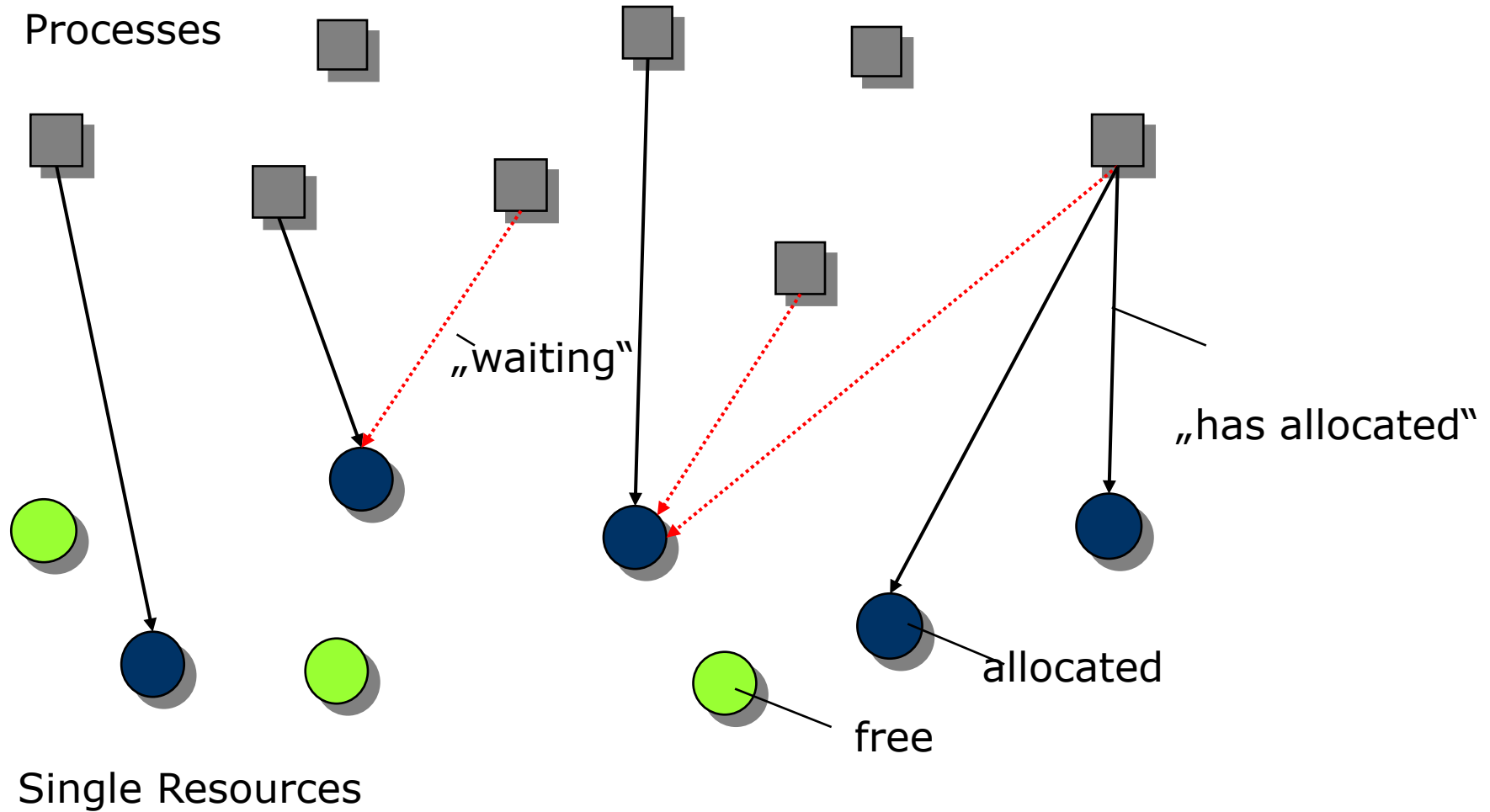
- Required data
  - State of allocation (free, allocated)
  - Waiting processes (blocked at allocation request)

- Additional data (optional)
  - Allocating processes (current owner)
  - Number of allocations
  - Average length of allocation time
  - Degree of utilization
  - Start of current allocation
  - ...

- Data needed to implement strategies for assignment or revocation.

Example of implementation

```
module single_unit_resource;
      export ALLOCATE_R, RELEASE_R;
      import BLOCK, DEBLOCK;

      var single_unit_R =
         record
            STATE: (free,occupied) = free;
            WP: queue of process = empty
         end;

      procedure ALLOCATE_R(R: single_unit_R);
         begin
         while R.STATE = occupied do
            BLOCK(R.WP);
         R.STATE := occupied
         end;

      procedure RELEASE_R(R: single_unit_R);
         begin
         R.STATE := free;
         if R.WP /= empty then
            DEBLOCK(R.WP);
         end;
   end single_unit_resource.
```

Freie Universität Berlin

- Amount of identical exemplars (e.g. drives for storage media)
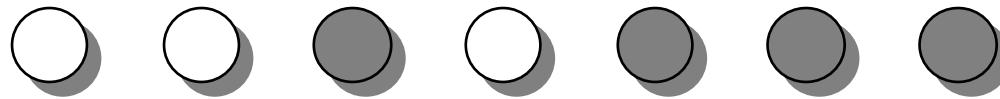
- Assumption: Single allocation
  - ALLOCATE_S (ID)
    - ID – Name or number of the allocated exemplar (return value)
  - RELEASE_S (ID)
    - ID – Name or number of the allocated exemplar (input value)

  - In this simple case a Semaphore initialized with amount of exemplars available would be sufficient.

- Divisible resources
  - Memory (one-dimensional divisible resource)

0 1 2 3   v-1
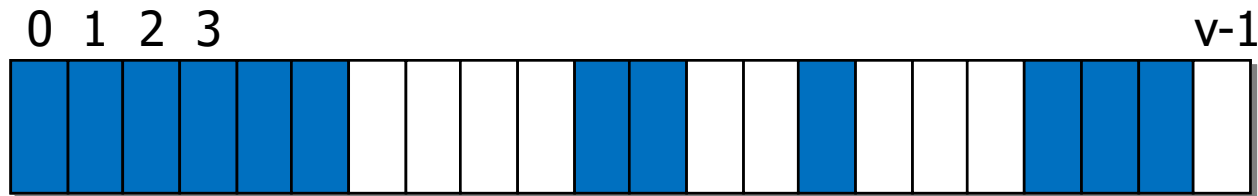
  - Processors (in Parallel Computers; two-dimensional divisible resource)

(0,0)   (0,v-1)

(v-1,0)   (v-1,v-1)

Interface
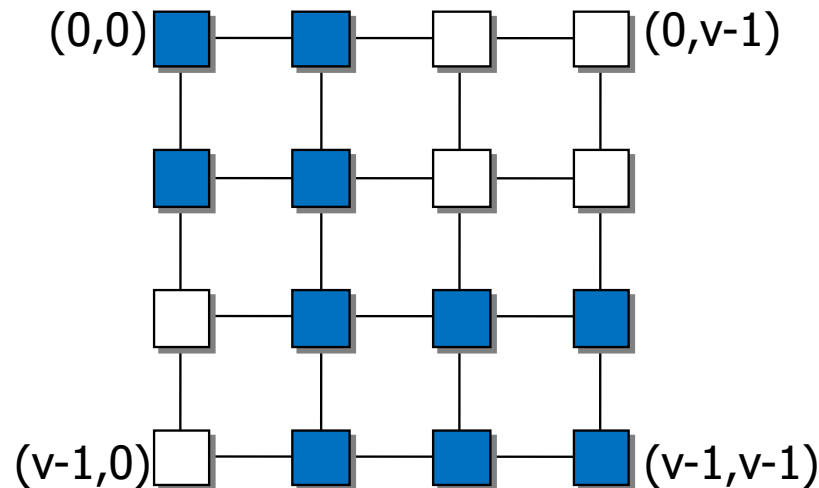
- Usually these divisible resources are allocated contiguously.

- In this case the area or amount of resources is determined by start address and number of parts exactly.

- ALLOCATE_R (START, NUM)
  - START – Index to the begin of allocated area (return value)
  - NUM – Number of units requested (input value)

- RELEASE_R (START, NUM)
  - Release an area starting with START and with length of NUM units

Representation of allocation

- Status of allocation is stored using a data structure.
- Simple case: Bit list

```
  0   1   2   3                                    v-1
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ 1 │ 1 │ 1 │ 1 │ 1 │ 1 │ 0 │ 0 │ 0 │ 0 │ 1 │ 1 │ 0 │ 0 │ 1 │ 0 │ 0 │ 0 │ 1 │ 1 │ 1 │ 0 │
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
```

    used   free
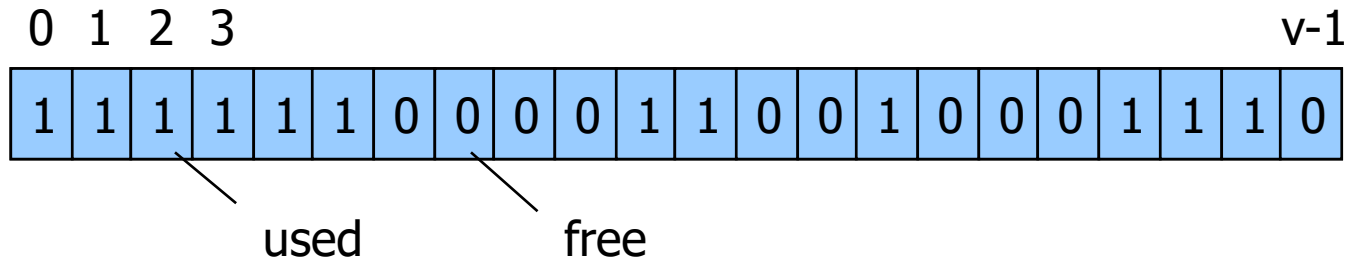
- Operations to change the status of allocation:
  - FREE (NUM)
    Check if amount of NUM units are available
  - SET_OCCUPIED (START, NUM)
    Set given NUM bits from START to "used"
  - SET_FREE (START, NUM)
    Set given NUM bits from START to "free"

Structure of management for a divisible resource

ALLOCATE _R(START,NUM)

FREE(NUM) — Y

BLOCK(WP)

SET_OCCUPIED(START,NUM)

Ⓡ

RELEASE _R(START,NUM)

SET_FREE (START,NUM)

WP = empty — Y

DEBLOCK(WP)

Ⓡ

```
module Multi_unit_resource;
export ALLOCATE_R, RELEASE_R;
    import BLOCK; DEBLOCK,
            FREE, SET_OCCUPIED, SET_FREE;
    var multi_unit_R =
        record
          STATE: array[0..v-1] of (free,occupied) = all free;
          WP: queue of process = empty
        end;

    procedure ALLOCATE_R(R: multi_unit_R;
                            START: address; NUM: int);
        begin
        while ¬ FREE(R,NUM) do BLOCK(R.WP);
        SET_OCCUPIED(R,START,NUM)
        end;

    procedure RELEASE_R(R: multi_unit_R; START: address; NUM: int);
        begin
        SET_FREE(R,START,NUM);
        while R.WP /= empty do DEBLOCK(R.WP);
        end;
end Multi_unit_resource.
```
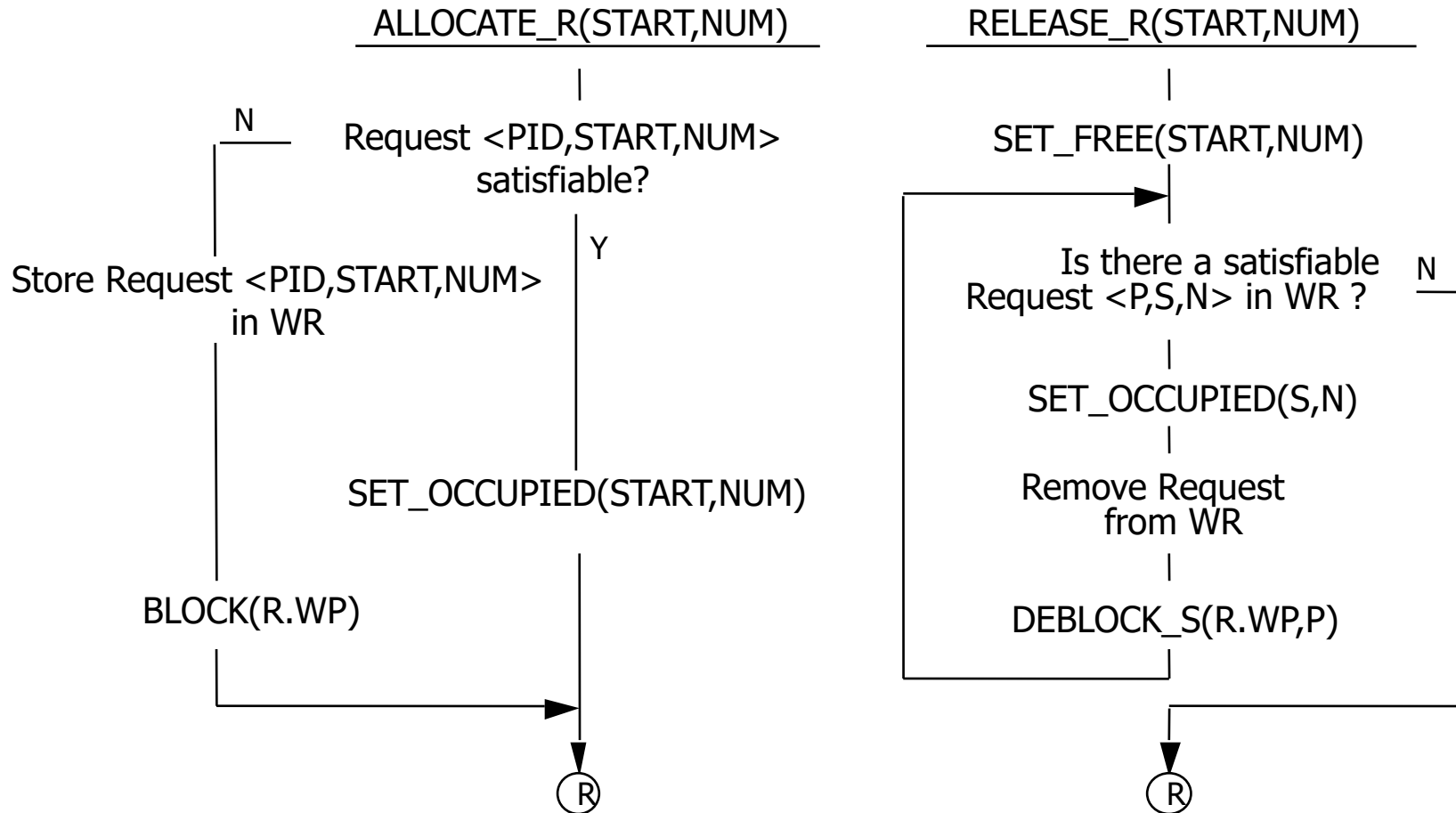
Expansion

- With many processes waiting the process with feasible request should be chosen.

- Therefore the request have to be stored at the resource manager (waiting requests).

- If requesting processes are sorted by size of request (amount of units), allocation can be combined with release.

- As the allocation is done be the releasing process it's called foreign-allocation (in contrast to self-allocation).

- There should be no additional check for resources within the ALLOCATE procedure.

Management of multi-exemplar resources

ALLOCATE_R(START,NUM)

Request <PID,START,NUM> satisfiable?

N

Store Request <PID,START,NUM> in WR

Y

SET_OCCUPIED(START,NUM)

BLOCK(R.WP)

R

RELEASE_R(START,NUM)

SET_FREE(START,NUM)

Is there a satisfiable Request <P,S,N> in WR ?

N

SET_OCCUPIED(S,N)

Remove Request from WR

DEBLOCK_S(R.WP,P)

R

- The implemented DEBLOCK operation is enhanced by a suffix _S (for select) as a specific process – other than the first one – is to be deblocked.

Example of implementation

```
module multi_unit_resource;
    export ALLOCATE_R, RELEASE_R;
    import BLOCK; DEBLOCK_S, INSERT, REMOVE, FIRST, ELEM, FREE,
        SET_OCCUPIED, SET_FREE;
    var multi_unit_r =
      record
          STATE: array[0..v-1] of (free,occupied) = all free;
          WP: list of process = empty;
          WR: list of record    PROC: process;
                                 START: address;
                                 NUM: int
                                 end = empty
        end;
    procedure ALLOCATE_R(R: multi_unit_r; START: address; NUM: int);
        var P:process; S:address;  N:int;
        begin
            if FREE(R, NUM)
            then SET_OCCUPIED(START,NUM)
            else begin    INSERT(R.WR,<P_RUN,START,NUM>);
                          BLOCK(R.WP)
                 end
        end;
    procedure RELEASE_R(R: multi_unit_r; START: address; NUM: int);
        var P:process; A:address; L:int;
        begin
        SET_FREE(R,START,NUM);
        while ∃ELEM ∈ R.WR: FREE(ELEM(R.WR).NUM) do
            begin   REMOVE(R.WR,ELEM(R.WR),<P,S,N>);
                          SET_OCCUPIED(S,N);
                          DEBLOCK_S(R.WP,P)
            end end;
    end multi_unit_resource.
```

- Debating foreign-allocation introduced strategy differed from FCFS principle.

- Strategies can be used to
  - get high utilization of the resource
  - treat the requesting processes fair.

- Given
  - $n_f(t)$ number of free units of resource at time t
  - $n(i)$ number of units requested by process i
  - $W(t)$ amount of waiting processes (requests) at time t
    W(t) can be implemented as queue; new requesting processes are inserted at the end.

Selection strategies

- FIFO (First-In-First-Out) or FCFS (First-Come-First-Served)
    - If $n(i) \leq n_f(t)$ for the request $n(i)$ units are allocated, otherwise nothing no action is performed.
    - Disadvantages:
        - Low utilization in case of large first request.
        - Because of the first large request smaller – satisfying – request are not served.
- First-Fit-Request
    - Searching the queue from head to find a satisfying request with $n(i) \leq n_f(t)$.
- Best-Fit-Request
    - Queue is searched until end and the request fitting to $$\min_{j \in W(t) \wedge n(j) \leq n_f(t)} \left\{ n_f(t) - n(j) \right\}$$ is chosen.
    - The request minimizing the amount of remaining free units is chosen.
    - Disadvantages of First-Fit- and Best-Fit-Request:
        - Processes with large requests may be penalized (starvation).

- **Iterative procedure**
  - After release of a large amount of units more than one request may be satisfied.
  - So the strategies may be applied until no more request can satisfied.
- **Windowing**
  - To reduce effort the search may be limited by a window of the size L and therefore only the first *L* positions of the queue are investigated.

Window

Request   Allocation

A5 | A2 | A6 | A3

Queue
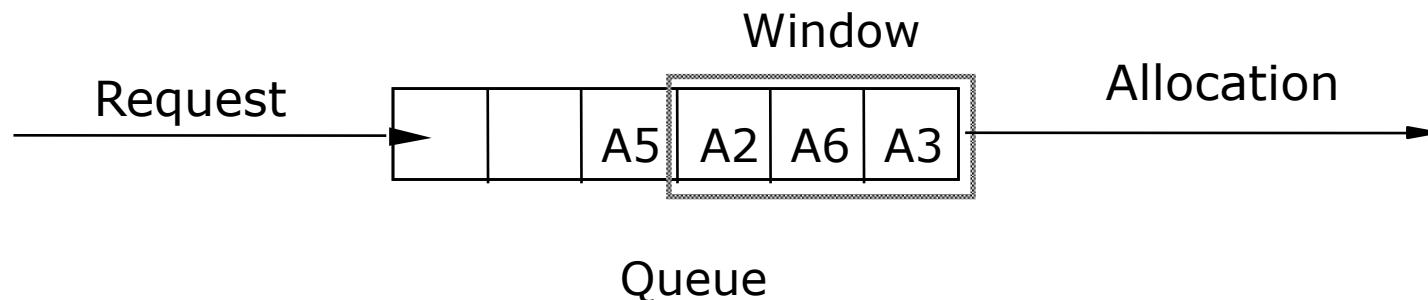
Solution for the problem of starvation

- Using a dynamic size of the window the problem of starvation in case of large requests (for First-Fit- or Best-Fit-Request) can be solved.

- Given L_{max} as maximal window size (initial value).

- The size of the window is adapted at every (successful) allocation:

$$L := \begin{cases} L-1, \text{ if } L>1 \text{ and request at head of queue is ignored} \\ L_{\max}, \text{ otherwise} \end{cases}$$

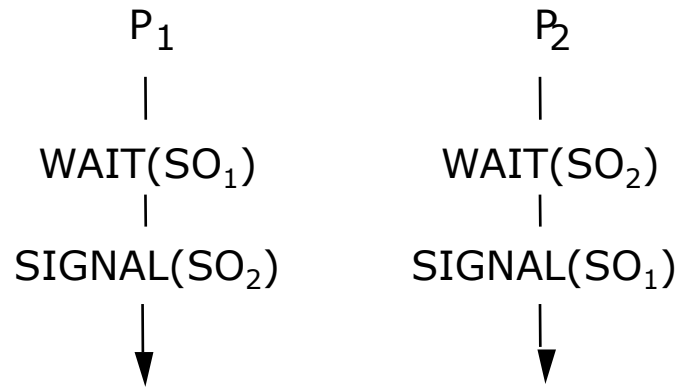- Therefore after max. L-1 omissions the first request is served (windows size == 1).

Window size L=1

Request → | | A8 | A5 | A2 | A6 | A3 | → Allocation

Queue
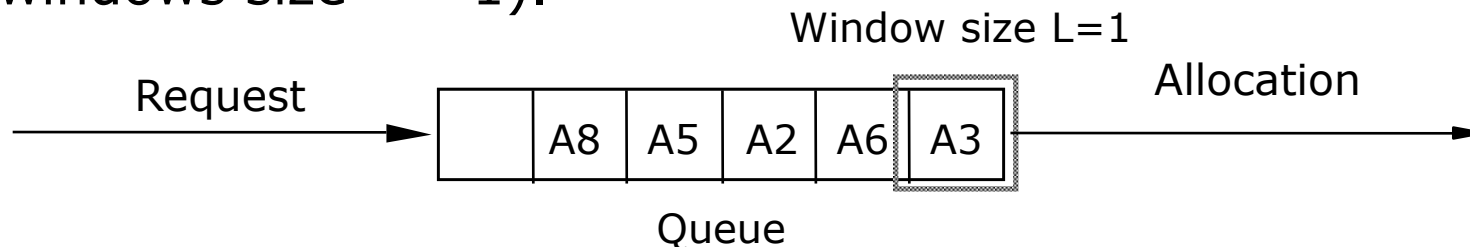
- With L converges to 1 the Best-Fit- and First-Fit-Request converge to FCFS.

- Examples:

Signaling | Communication

P_1 P_2

|

WAIT(SO_1)   WAIT(SO_2)

|

SIGNAL(SO_2)   SIGNAL(SO_1)

P_1 P_2

|

RECEIVE_S(CO_1)   RECEIVE_S(CO_2)

|

SEND_A(CO_2)   SEND_A(CO_1)

Locks

P_1 P_2

|

LOCK(LO_1)   LOCK(LO_2)

|

LOCK(LO_2)   LOCK(LO_1)
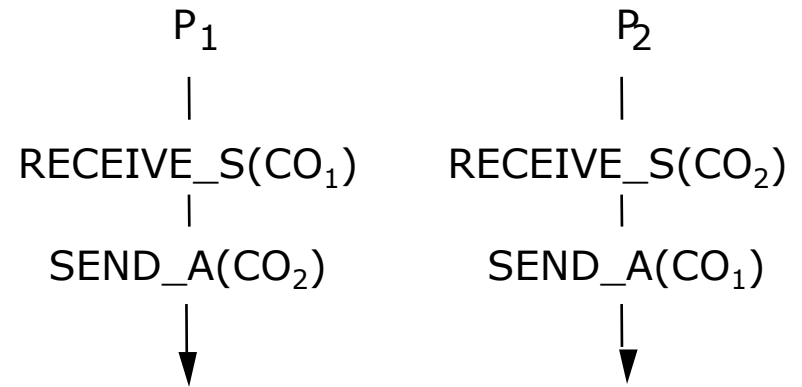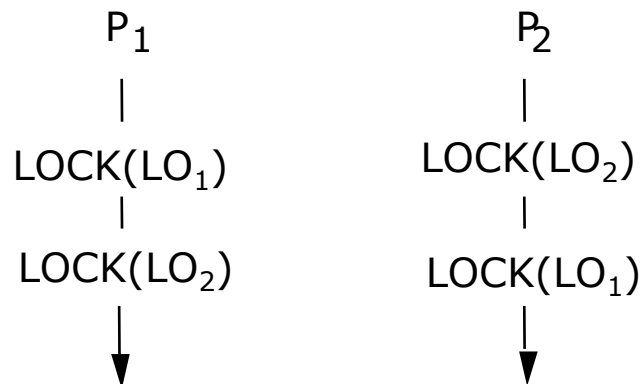
- ## Management of one-exemplar resource

P_1 P_2

ALLOCATE(R1)   ALLOCATE(R2)

ALLOCATE(R2)   ALLOCATE(R1)
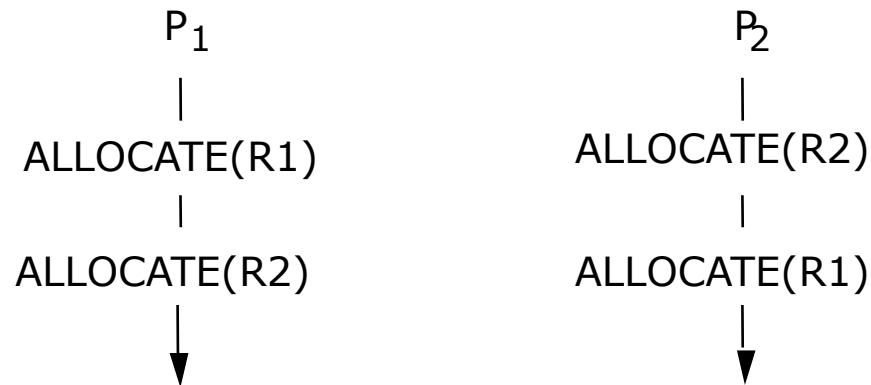
- ## Management of multi-exemplar resource with 10 units

P_1 P_2

ALLOCATE_R(5)   ALLOCATE_R(5)

ALLOCATE_R(1)   ALLOCATE_R(1)
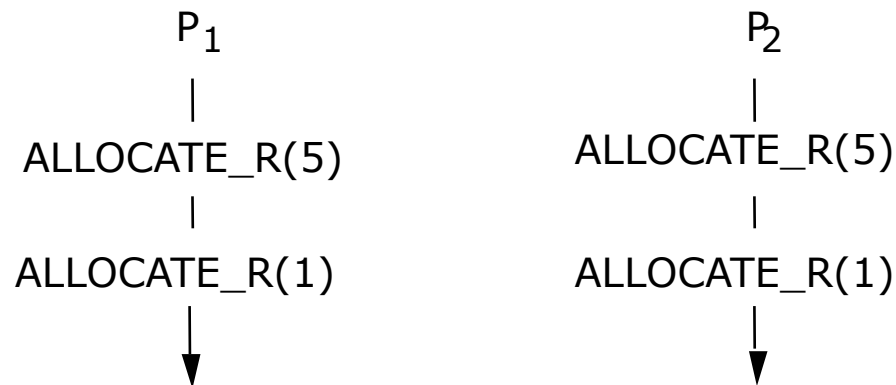
- More than one requesting processes
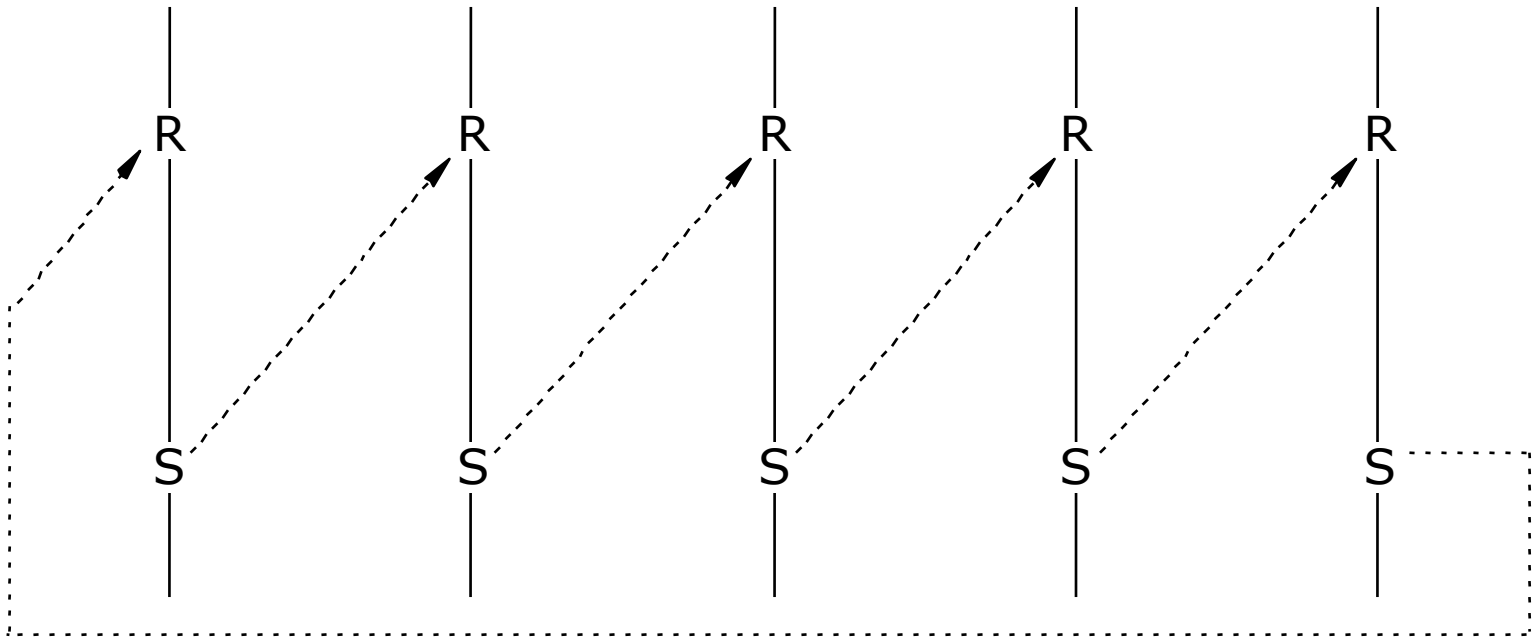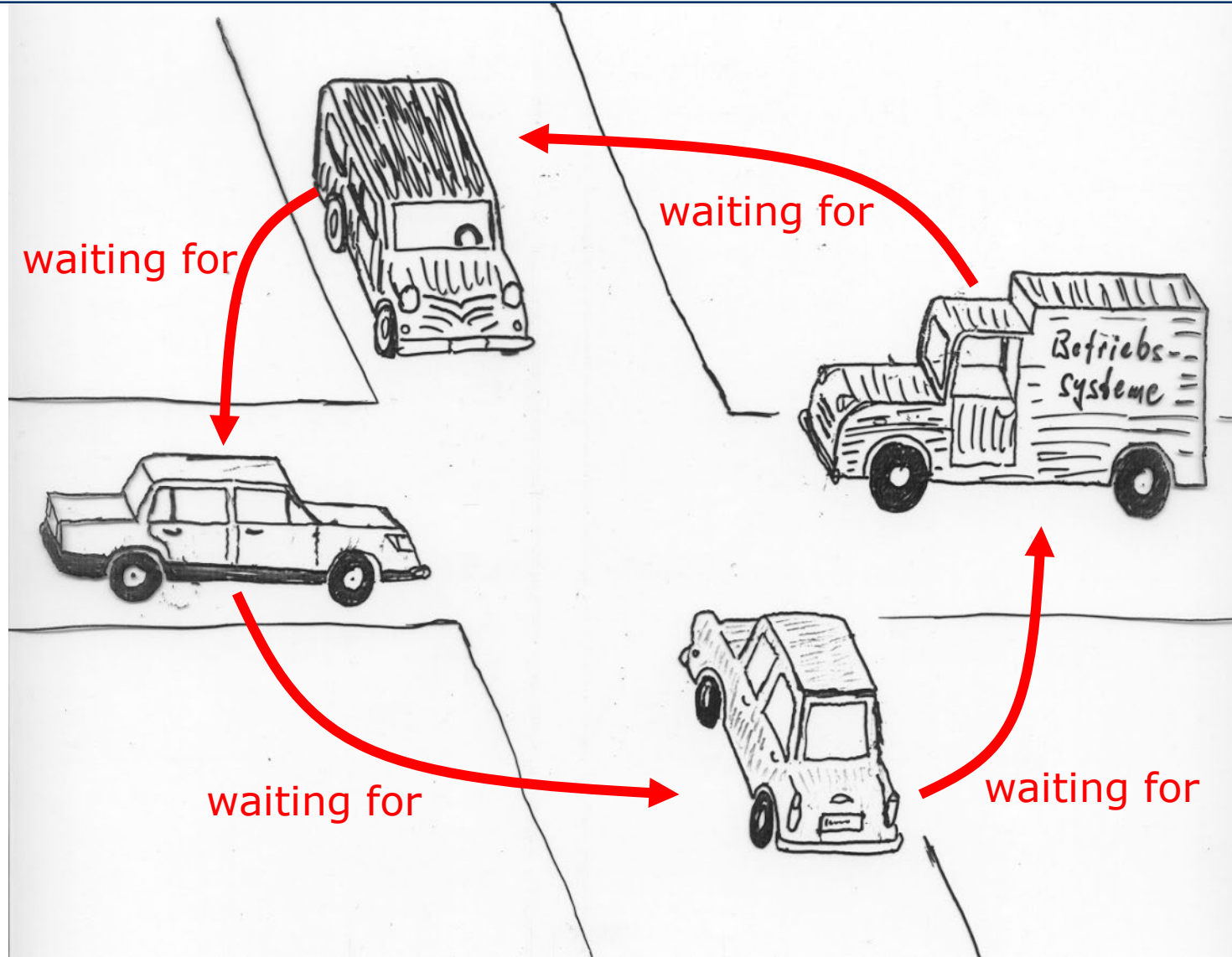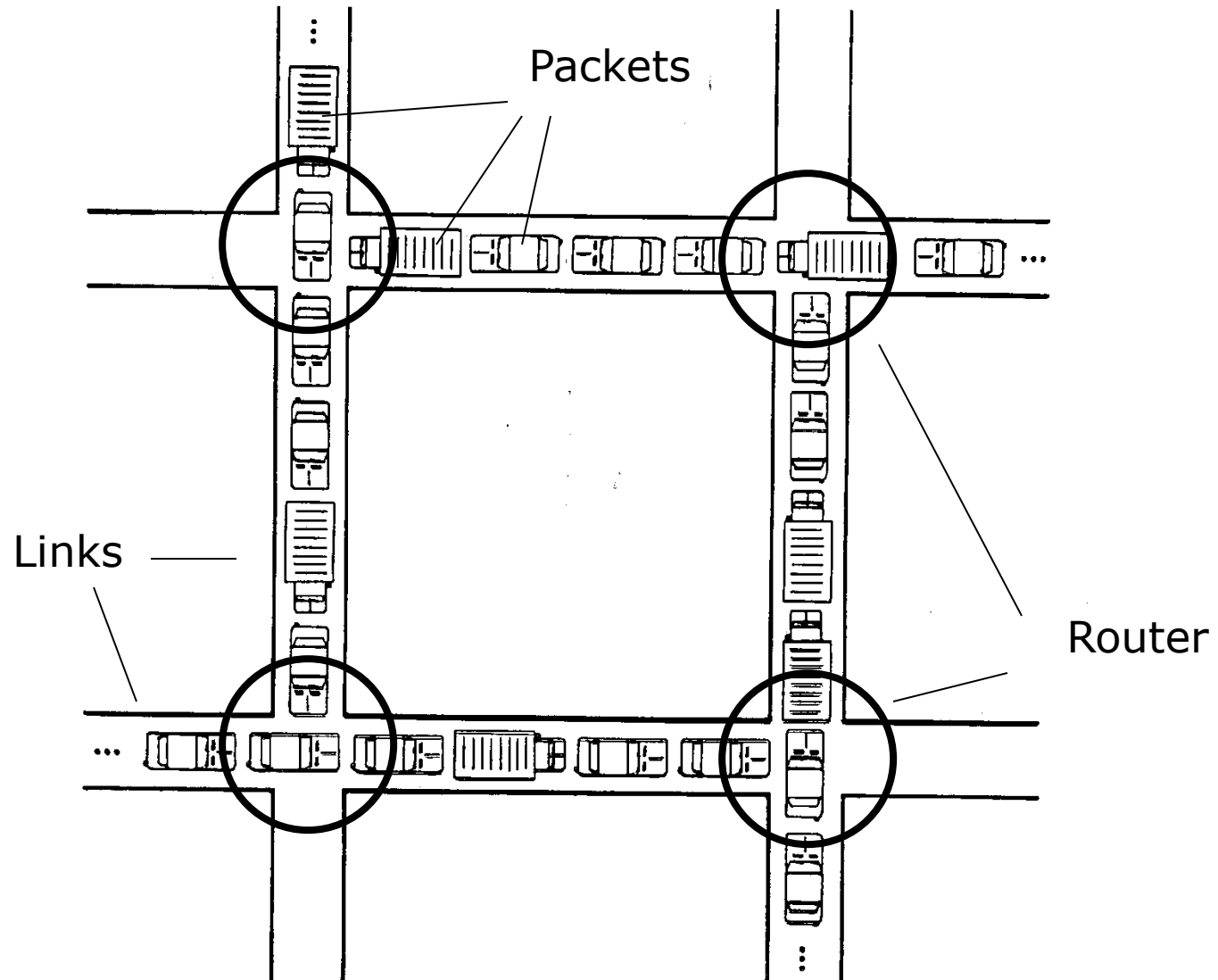
- Directed graph with processes as nodes and edges show the relation if a process waits for a resource allocated by another process.

- A cycle in graph indicates a deadlock.

with 6 processes

with 2 processes
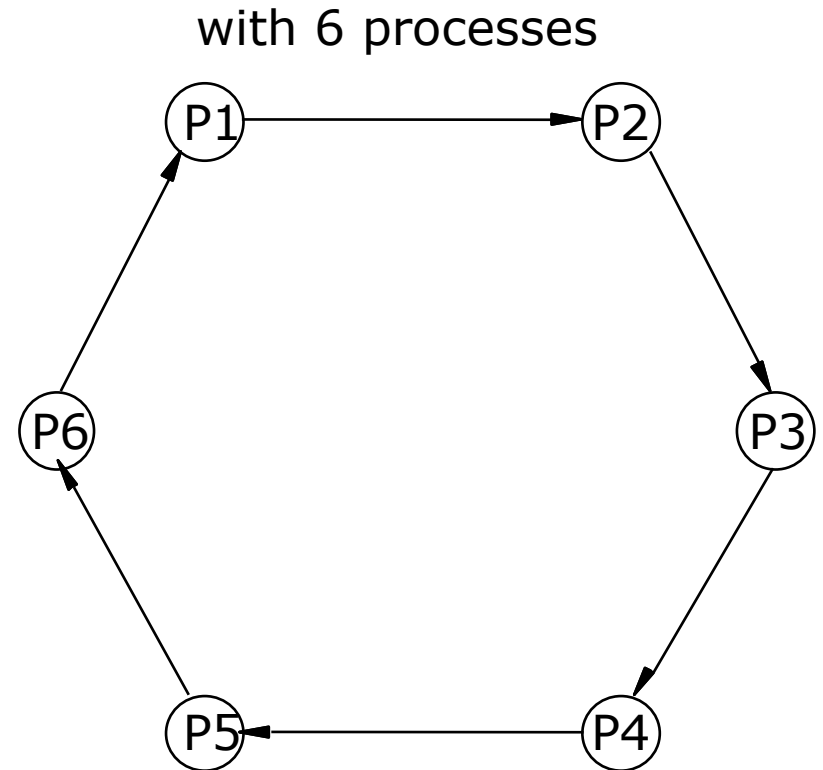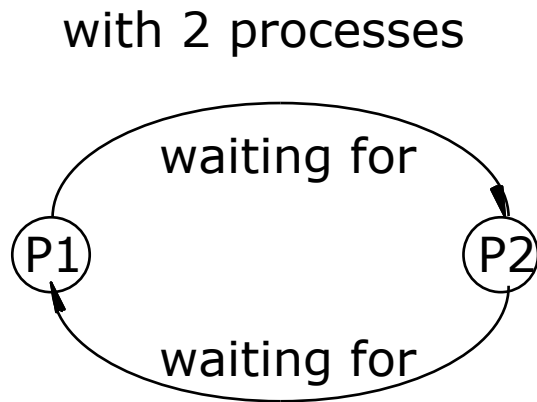
Deadlock

- Deadlocks can occur in different situations.
- In context of resource management these three **requirements are necessary** for a deadlock:

  1. Resources are used exclusive.
  2. Processes hold allocation of a resource and try to allocate another.
 3. There is no preemption.

  With these requirements fulfilled the following constraint may occur and together with the first requirements this is **sufficient** for a deadlock:

  4. There is a cycle in the wait-for graph.

- To deal with a deadlock different counteractions have to be provided:
  - Prevention
  - Avoidance
  - Detection
  - Resolution, Recovery

- Prevention stands for a methodic procedure handle resource assignment restrict in a way no deadlock can occur.

- Pre-claiming
  - All resources needed by a process are requested (and allocated) at start time.

waiting for

Pi
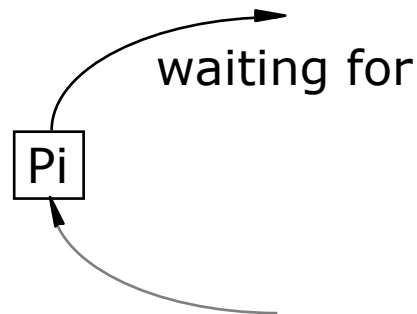
cannot occur as process holds no resources

  - In dynamic systems the overall demand is difficult to predict.
  - Uneconomical approach as resources are occupied longer than needed.

- **Overall release at request**

$$\text{Allocate (Res}_1, \text{Res}_2)$$

$$\text{Release (Res}_1, \text{Res}_2)$$
$$\text{Allocate (Res}_3)$$

eventual forced release

$$\text{Release (Res}_3)$$
$$\text{Allocate (Res}_1, \text{Res}_2)$$
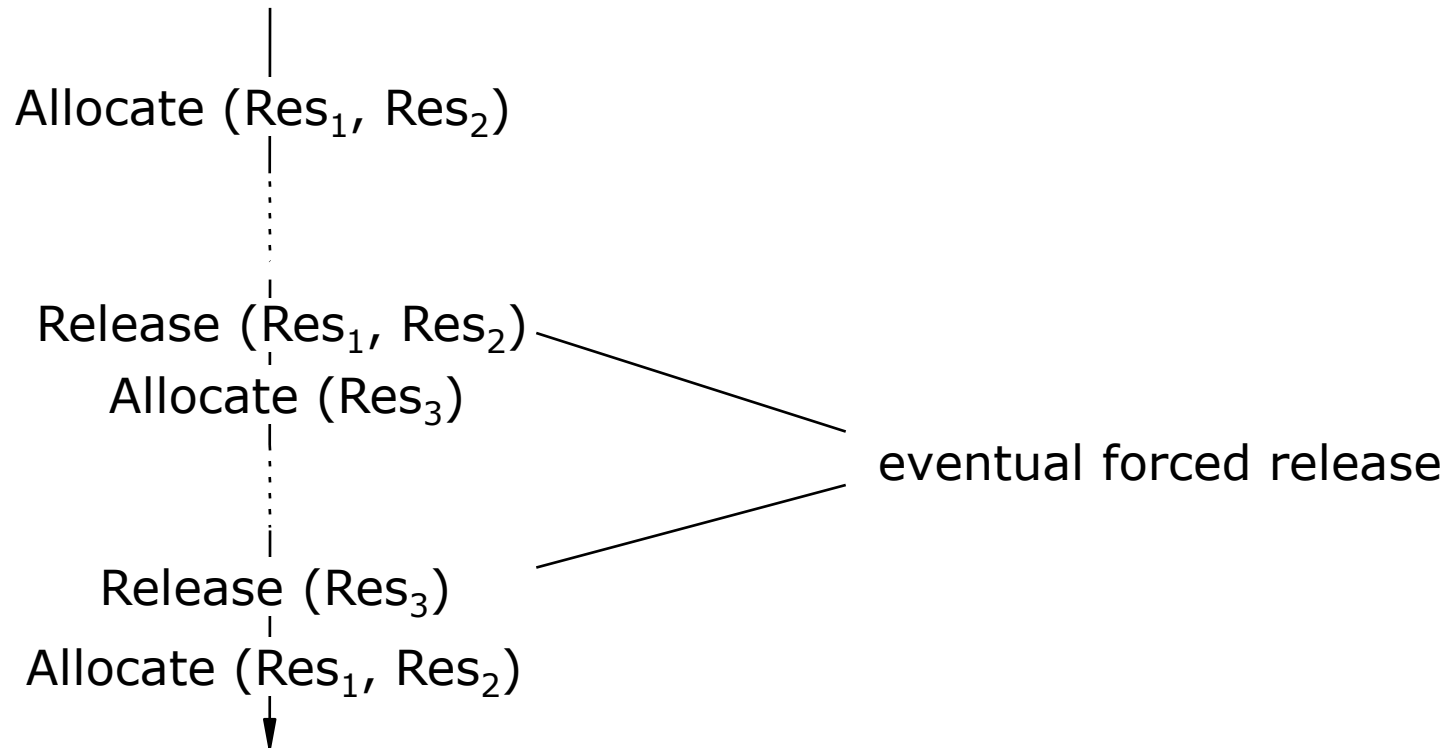
- As the process does not possess any resource at allocation the cycle in wait-for graph is prevented, too.

Deadlock prevention

- **Allocation by (given) order**
  - Resource are sorted (Res_1, Res_2, Res_3, …).
  - Resource allocation is performed in order only.

<div>

P1

Allocate (Res_1)

Allocate (Res_3)

Allocate (Res_4)

P2

Allocate (Res_2)
Allocate (Res_3)

Allocate (Res_5)

</div>
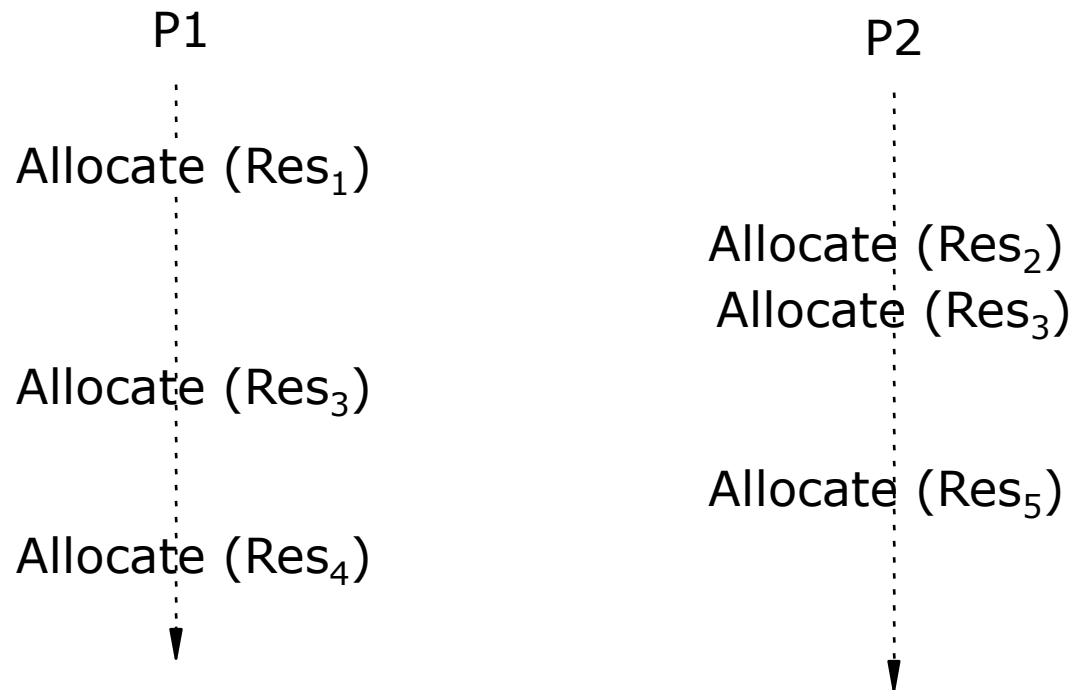
  - With this approach cycles in the wait-for graph are prevented successfully.

- P Set of processes, $|P| = m$
- Rs Set of resource types, $|Rs| = n$
- $\vec{v} := (v_1, v_2, ..., v_n)$ Available resources
- Requests: Assignments:

$$A := \begin{pmatrix} a_{11} & ... & a_{1n} \\ : & & : \\ a_{m1} & ... & a_{mn} \end{pmatrix} \qquad B := \begin{pmatrix} b_{11} & ... & b_{1n} \\ : & & : \\ b_{m1} & ... & b_{mn} \end{pmatrix}$$

- Overall request (maximum of all requests):

$$G := \begin{pmatrix} g_{11} & ... & g_{1n} \\ : & & : \\ g_{m1} & ... & g_{mn} \end{pmatrix}$$

- Quintuple $(P, Rs, \vec{v}, B, A)$ represents current resource allocation.

1. $\forall j \in \{1,...,n\}: \sum_{i=1}^{m} b_{ij} \leq v_j$ no more than available resources can be allocated.

2. $\forall i \in \{1,...,m\} \ \forall j \in \{1,...,n\}:$ $\begin{aligned} a_{ij} + b_{ij} &\leq v_j \\ g_{ij} &\leq v \end{aligned}$

   Requests can only contain amount of resource available.

3. Requesting processes are blocked until allocation.

4. Only non-blocked processes can request another resource (see 3.).

- **Free resources**

$$\vec{f} := (f_1, f_2, \ldots, f_n)$$

with

$$f_j := v_j - \sum_{i=1}^{m} b_{ij} \qquad \text{free res. = (existing) res. – allocated res.}$$

- **Remaining requests**

$$R := \begin{pmatrix} r_{11} & \ldots & r_{1n} \\ : & & : \\ r_{m1} & \ldots & r_{mn} \end{pmatrix}$$

with

$$r_{ij} := g_{ij} - b_{ij} \qquad \text{remaining req. = overall req. – allocated res.}$$

Notation

- Row vector instead of matrix

  - $\vec{a_i} := (a_{i1}, a_{i2}, \ldots, a_{in})$ request of process i

  - $\vec{b_i} := (b_{i1}, b_{i2}, \ldots, b_{in})$ allocated resources of process i

  - $\vec{g_i} := (g_{i1}, g_{i2}, \ldots, g_{in})$ overall request of process i

  - $\vec{r_i} := (r_{i1}, r_{i2}, \ldots, r_{in})$ remaining request of process i

- Relational operators

  - $\vec{x} \leq \vec{y} \Leftrightarrow \forall k : x_k \leq y_k$

  - $\vec{x} \nleq \vec{y} \Leftrightarrow \exists k : x_k > y_k$

- Process P_i is blocked if $\vec{a}_i \not\leq \vec{v} - \sum_{k=1}^{m} \vec{b}_k = \vec{f}$, the current request cannot be satisfied.

- Set of processes $P = \{P_1, P_2, ..., P_m\}$ is in a deadlock if $\exists I \subseteq \{1,2,...,m\} : \forall k \in I : \vec{a}_k \not\leq \vec{v} - \sum_{k \in I} \vec{b}_k$, there is a subset of processes that requests could no be satisfied by the amount of resource not allocated by the processes of P.

- The subset I is also called to be in a deadlock.

- Example:

$$I = \{1,2\}, \; n = 2 \qquad \vec{v} = (4,4) \qquad \vec{b}_1 = (2,3) \quad \vec{b}_2 = (1,1) \quad \vec{a}_1 = (0,1) \qquad \vec{a}_2 = (2,0)$$

$$\vec{v} - \sum_{k \in I} \vec{b}_k = (4,4) - (2,3) - (1,1) = (1,0)$$

$$\vec{a}_1 = (0,1) \not\leq (1,0) \qquad \qquad \vec{a}_2 = (2,0) \not\leq (1,0)$$

- With resource graph the request and allocation status can be formally described.
- P is set of processes, Rs is set of resource types.
- A resource graph is a directed graph *(V, E)* with
  $V=P \cup Rs$ and
  - *(p, r)* $\in E \Leftrightarrow$ Process *p* make a request for a unit of resource type *r*
  - *(r, p)* $\in E \Leftrightarrow$ Process *p* holds allocation of a unit of resource type *r*

- Resource graph is bipartite regarding *P* (cycles) and resources *Rs* (rectangles), so there are edges between *P* and *Rs* only.

- The number of units provided by a resource type is depicted as node weight (points within the rectangle) and determines the max. number of units that can be allocated by processes.
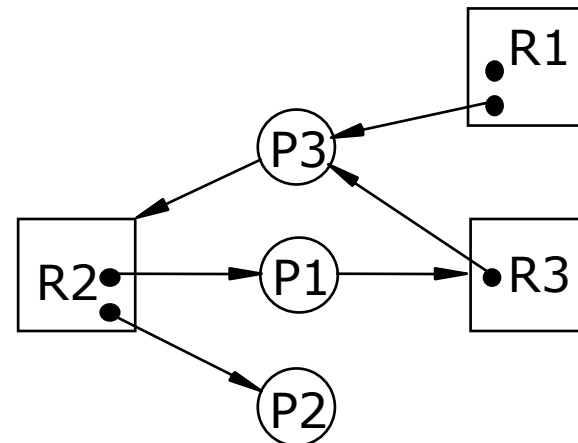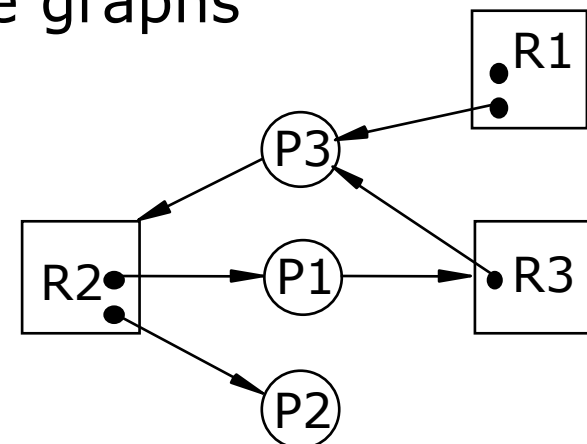
Resource graph
Properties and Operations

- Resource graph depict a specific status of system, shows the current resource status.

- Comparable to the wait-for graph a cycle points to a deadlock condition.

- But in difference to the wait-for graph with a cycle in the resource graph there is no necessity to have a deadlock, but it is necessary to have such a cycle to get one.

- Every operation of resource management (request, allocation, release) imply a transformation of the graph (add or remove edge).

- A process only can perform such a operation if it's not blocked.

- In case all of the remaining request can be fulfilled the process may finish successfully (termination).

- With termination of a process all allocated resources are released.

- A process *p* can reduce the resource graph by removing all of the edges for allocation if it's not isolated or blocked.

- A resource graph can be reduced completely if there is an order of reductions (processes reducing the graph by releasing the resources) so that all edges will be removed at the end.

- Theorem of deadlocks for resource graphs
  - There is a deadlock if the resource graph cannot be reduced completely.

Example: a)
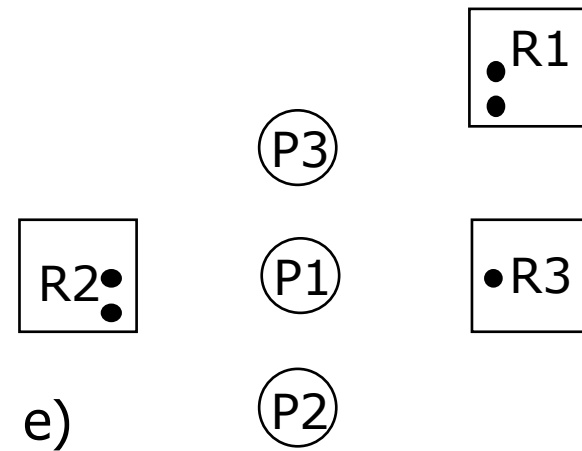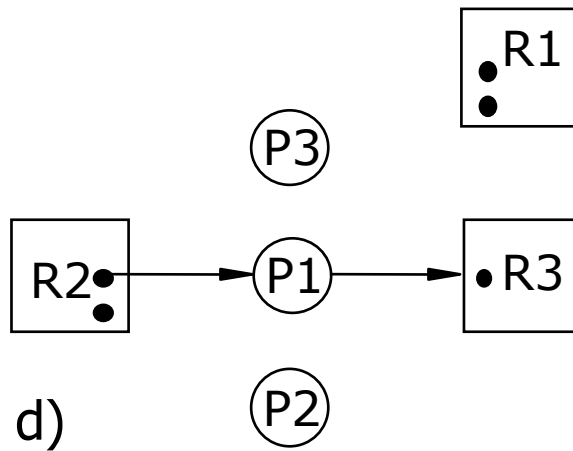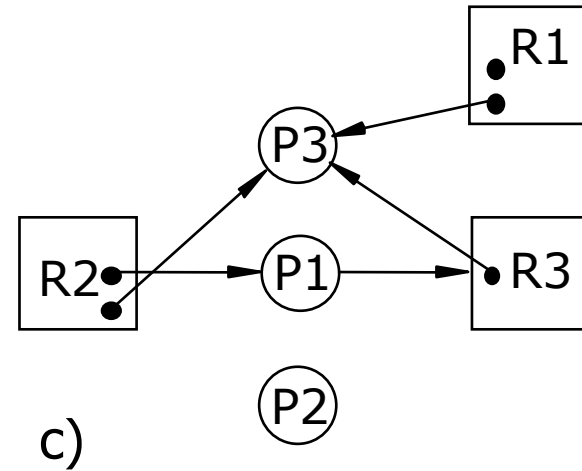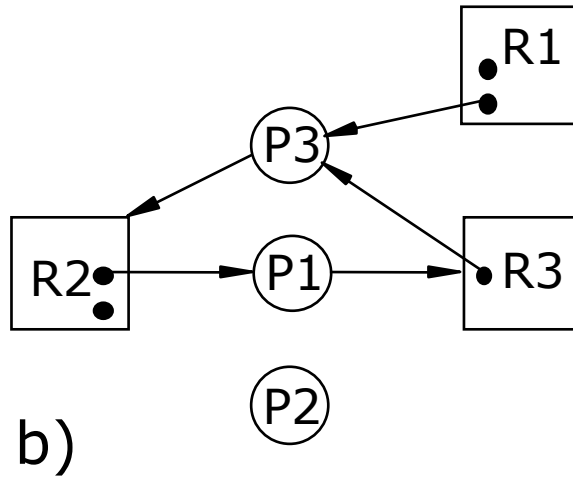
- The definition of a deadlock describes a current system situation.

- To *avoid* such a deadlock the remaining requests of the processes have to be known.

- In worst case all of the remaining requests occur at the same time.

- If all of these remaining requests can be satisfied the situation is *safe*.

- Otherwise the situation is *unsafe*.

- The system status is unsafe if there is a subset of processes that have remaining requests that cannot be satisfied with the resource currently available.

- There might be some requests that can be satisfied, but in worst case of orders of allocation and release a deadlock may occur.
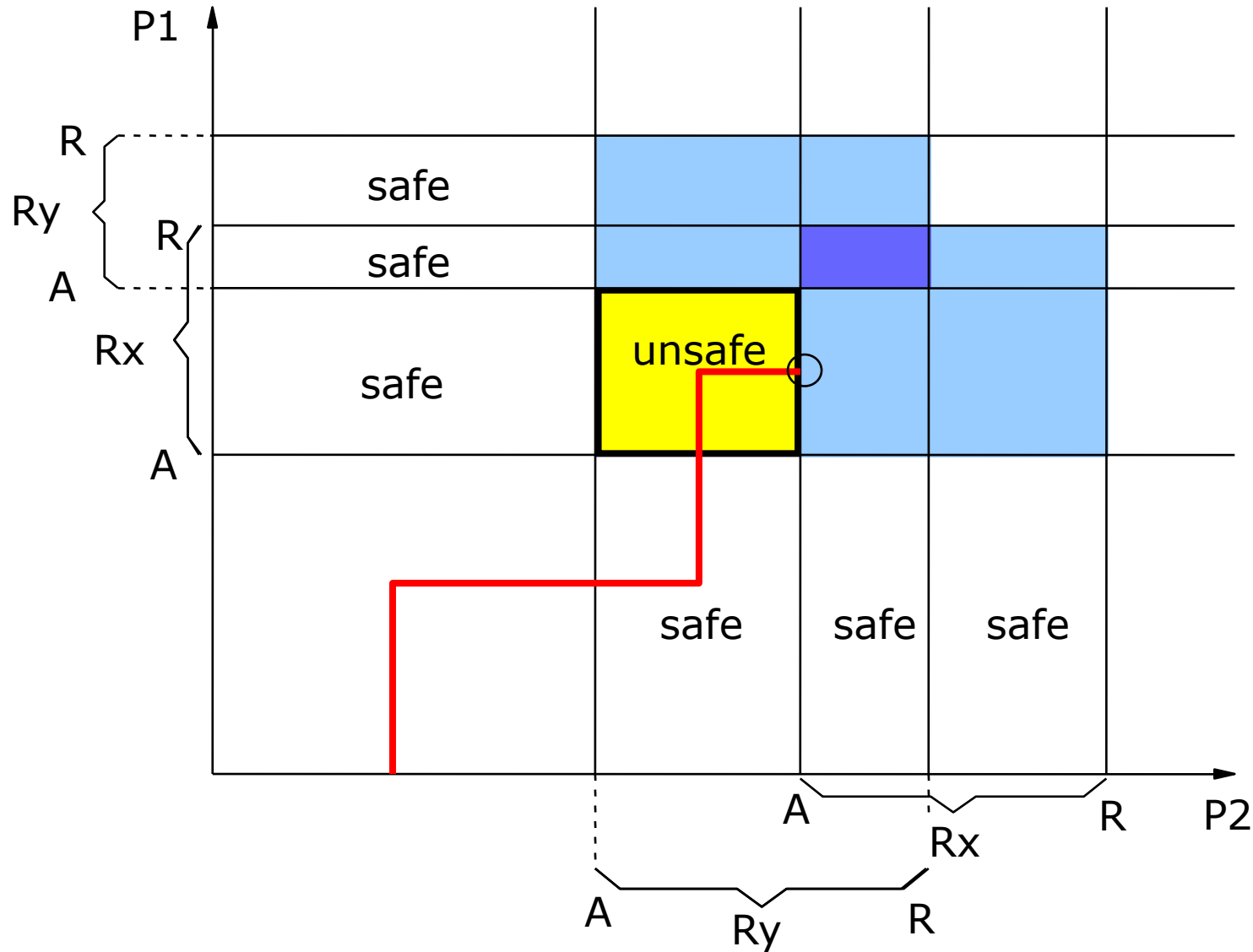
Deadlock avoidance

- Deadlock avoidance implement the resource management in such a way no unsafe situation can occur. So requests are satisfied only if it leads to a safe situation.

- A set of processes $P = \{P_1, P_2, \ldots, P_m\}$ is called safe if

    $$\exists \text{Permutation } P_{k_1}, P_{k_2}, \ldots, P_{k_m} \text{ with}$$
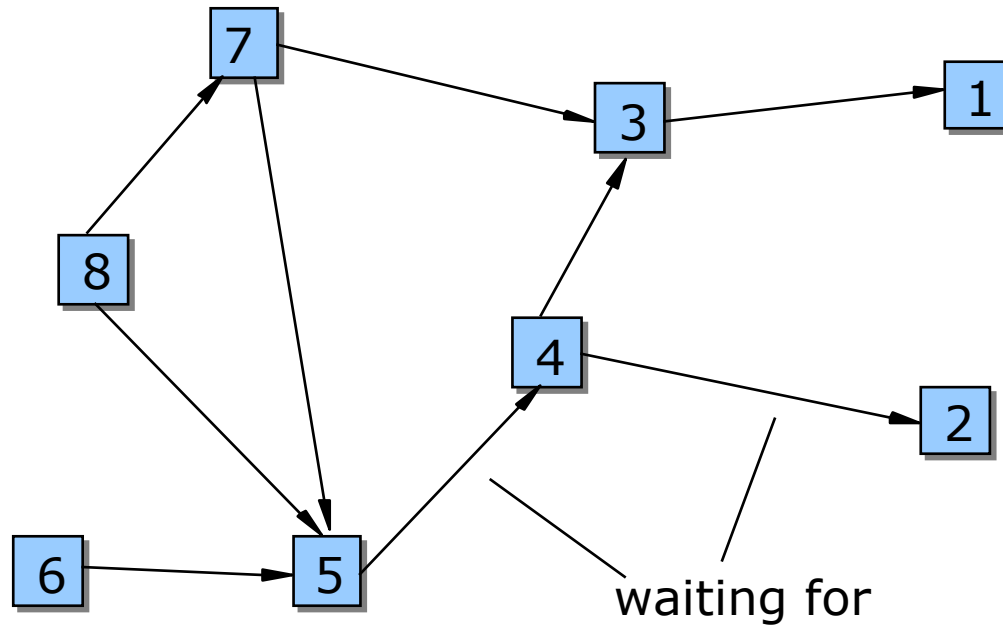
    $$\forall r \in \{1,2,\ldots,m\}: \vec{r}_{k_r} \leq \vec{f} + \sum_{s=1}^{r-1} \vec{b}_{k_s}$$

    or

    $$\forall r \in \{1,2,\ldots,m\}: \vec{r}_{k_r} \leq \vec{v} - \sum_{s=r}^{m} \vec{b}_{k_s}$$

    i.e. there is a order of process termination so the remaining request of the other processes can be satisfied.

waiting for

Deadlock avoidance

- Check for every request if the allocation would lead to an unsafe situation.

- If case the request would lead to an unsafe situation postpone request.

- In order to perform the check for the current situation the **Banker's Algorithm** can be used:

```
procedure deadlock_avoidance (P: set of processes, v: vector,
    r:matrix,b:matrix, var answer: state, DP: set of processes)
var f: vector;
begin
    answer := undefined;
    DP := P;
```
$$f: = v - \sum_{i=1}^{m} b_i;$$
```
    while answer = undefined do
        if ∃ Pᵢ ∈ DP: rᵢ ≤ f
                then begin
                        DP := DP − {Pᵢ};
                        f := f + bᵢ  ;
                        if DP = ∅ then answer:= safe;
                    end
                else answer := unsafe;
end;
```

Complexity and Example

- Complexity of the Banker's Algorithm
  - Selection of a process (pass loop):   O (m n)
  - with max. m processes: O (m² n)
- Example
  - Given system with m=4 processes (i=1,..,4) and n=2 type of resources (j=1,2). Current status:

Allocation   Overall requests   Free resources

$$B = \begin{pmatrix} 0 & 4 \\ 1 & 0 \\ 3 & 0 \\ 5 & 4 \end{pmatrix} \qquad G = \begin{pmatrix} 4 & 6 \\ 2 & 4 \\ 3 & 1 \\ 10 & 6 \end{pmatrix} \qquad f = \begin{pmatrix} 3 & 2 \end{pmatrix}$$

So the Remaining requests can be calculated as R and the overall available resources v:

$$R = \begin{pmatrix} 4 & 2 \\ 1 & 4 \\ 0 & 1 \\ 5 & 2 \end{pmatrix} \qquad v = \begin{pmatrix} 12 & 10 \end{pmatrix}$$

- Following the algorithm:

  1. P3 can terminate successfully: $r_3 = (0 \quad 1) \leq f = (3 \quad 2)$
     $f := f + b_3 = (3 \quad 2) + (3 \quad 0) = (6 \quad 2)$
  2. P1 can terminate successfully: $r_1 = (4 \quad 2) \leq f = (6 \quad 2)$
     $f := f + b_1 = (6 \quad 2) + (0 \quad 4) = (6 \quad 6)$
  3. P2 can terminate successfully: $r_2 = (1 \quad 4) \leq f = (6 \quad 6)$
     $f := f + b_2 = (6 \quad 6) + (1 \quad 0) = (7 \quad 6)$
  4. P4 can terminate successfully: $r_4 = (5 \quad 2) \leq f = (7 \quad 6)$
     $f := f + b_4 = (7 \quad 6) + (5 \quad 4) = (12 \quad 10) = v$

- As there is a order to terminate the processes successfully (P3, P1, P2, P4) the situation is *safe*!
  - (P3, P1, P4, P2 is also possible.)

- In case there is no knowledge available about the remaining requests Deadlock avoidance cannot be performed.

- So deadlocks may occur.

- At least the occurrence of a deadlock has to be *detected*.

- This can be done by searching for all of the processes that are not involved in a deadlock situation.

- The Processes P_1, P_2, ..., P_m are not involved in a deadlock situation if $\exists \text{Permutation}\, P_{k_1}, P_{k_2}, .., P_{k_m}$ with

$$\forall r \in \{1,2,...,m\} : \vec{a}_{k_r} \leq \vec{f} + \sum_{s=1}^{r-1} \vec{b}_{k_s}$$

or

$$\forall r \in \{1,2,...,m\} : \vec{a}_{k_r} \leq \vec{v} - \sum_{s=r}^{m} \vec{b}_{k_s}$$

there is a order of termination of the processes in such kind all request can be satisfied by free resources or by resources released earlier.

Deadlock detection

- The approaches to avoid a deadlock enforcing *safe* situations only and the deadlock *detection* are nearly the same.

- In case of deadlock avoidance a permutation is searched for that can be executed if all of the *remaining requests* occur at the same time.

- For deadlock detection the same operations are performed for the current request.

- So we can use the Banker's Algorithm for deadlock detection too.

- We do have to replace the *remaining requests* with the *current requests*.

- The algorithm has to consider all of the processes.

- The Banker's Algorithm for deadlock detection can run at:
  - every new request
  - periodically
  - as part of the Idle process
  - in case of suspicion (maybe manually)

Freie Universität Berlin

```
procedure deadlock_detection(P: set of processes, v: vector,
    a:matrix, b:matrix, var answer: state, DP: set of
    processes)
var f: vector;
begin
    answer := undefined;
    DP := P;
```

$$f: = v - \sum_{i=1}^{m} b_i \ ;$$

```
    while answer = undefined do
        if ∃ Pᵢ ∈ DP: aᵢ ≤ f
                then begin
                        DP := DP - {Pᵢ};
                        f := f + bᵢ   ;
                        if DP = ∅ then answer:= no_deadlock;
                    end
        else answer := deadlock;
end;
```

Special case: One-exemplar resource

- In case of a one-exemplar resource: a_{ij}, $b_{ij} \in \{0,1\}$.
- Relationship at the wait-for graph can be determined:

  Process i is waiting for process $j \Leftrightarrow w_{ij} = 1 \Leftrightarrow$

  $\exists k: a_{ik} \times b_{jk} = 1$
- The matrix $W = (w_{ij})$ can be seen as adjacency matrix of the wait-for graph.

- Proposition
  - There is a deadlock in case there is a cycle within the wait-for graph.

- Deadlock detection is reduced cycle detection (depth-first search with complexity $O(|edges| + |nodes|)$).

9.4.6 Deadlock resolution

- Every deadlock resolution focuses on cutting the cycle in the wait-for graph.

- In case it is impossible to (controlled) withdraw the resource the abort of the process is necessary.

- Then there is the question which process is to be aborted.

- Criteria:
  - Size of request
  - Amount of resources allocated
  - Urgency
  - User- vs. system process
  - Effort of abort
  - Wasted work
  - Remaining answer time

Further References

- Stallings,W.: *Operating Systems*, Prentice Hall, 2001, Chapter 6
- Tanenbaum, A.: *Moderne Betriebssysteme*, 2.Aufl., Hanser, 1995, Kapitel 6
- Bacon,J.: *Concurrent Systems*, Addison Wesley, 1997 Chapter 17
- Nehmer,J.; Sturm,P.: *Systemsoftware*, dpunkt-Verlag, 2001, Kapitel 8
- Holt, R.: *Some Deadlock Properties of Computer Systems.* Computer Surveys, Sept. 1972
- Isloor,S.; Marsland, T.: *The Deadlock Problem: An Overview.* Computer, Sept. 1980