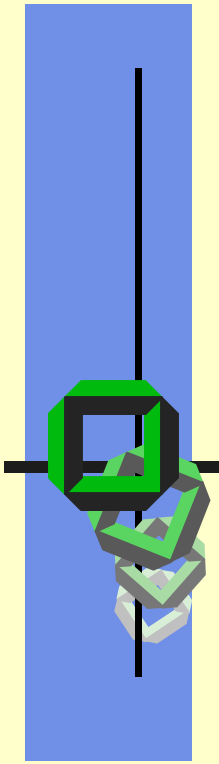
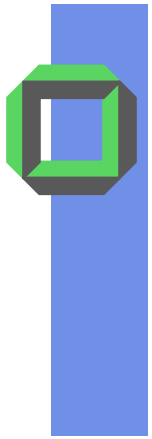


System Architecture

12 Deadlocks

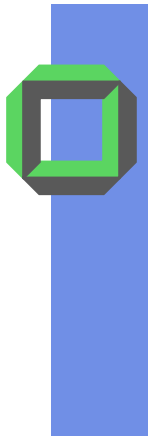
Resource Management
Deadlock Conditions
Dealing with potential Deadlocks

January 7 2009
Winter Term 2008/09
Gerd Liefländer



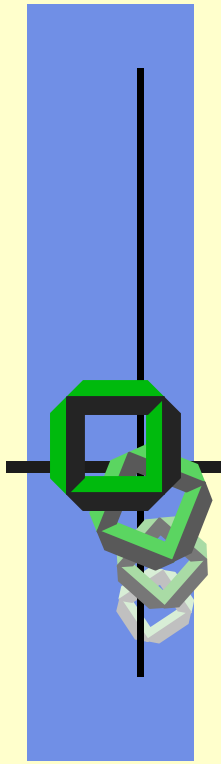
Literature

- Bacon, J.: Operating Systems (11, 18)
- Nehmer, J.: Grundlagen moderner BS (6, 7, 8)
- Silberschatz, A.: Operating System Concepts (7)
- Stallings, W.: Operating Systems, 6
- Tanenbaum, A.: Modern Operating Systems (3)

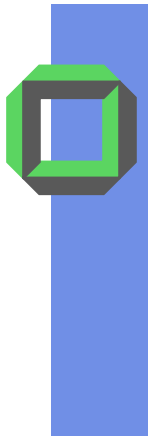


Agenda

- Motivation & Introduction
- Examples
- Resource Management
- Visualization of Deadlocks
- Necessary Deadlock Conditions
- Policies against Deadlocks
  - Ostrich Algorithm
  - Detection
  - Avoidance
  - Prevention
  - Approach in the practice
- Two-Phase Locking (next ST in the data base course)



Motivation & Introduction



Deadlocks? First View

Mutual blocking of a “set of threads” either competing for resources or interacting with each other
(via synchronization, cooperation or communication)

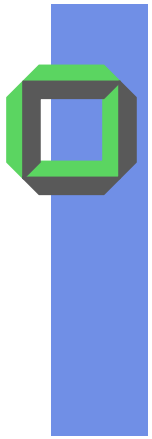
⇒ Conflicts by at least 2 threads/processes

State of the art::

No satisfying solution in all cases

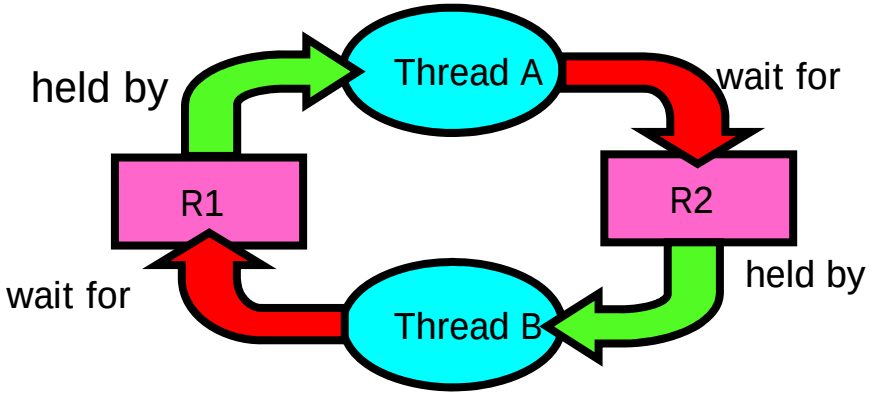
Some commodity OSes don‘t bother with deadlocks at all, e.g. the designer of SUN OS argued:

deadlocks will occur so scarcely

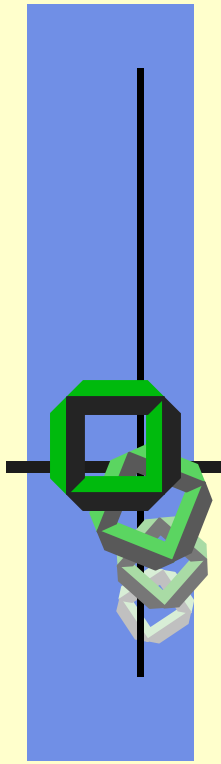


Starvation versus Deadlock

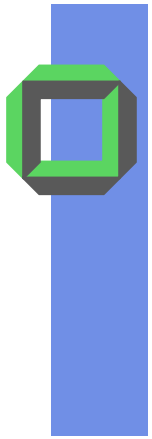
- Starvation: thread/process waits indefinitely
  - Example, low-priority thread waiting for resources constantly in use by high-priority threads
- Deadlock: circular waiting for resources
  - Thread A holds resource R1 and is waiting for R2
    Thread B holds resource R2 and is waiting for R1



- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn't have to)
  - Deadlock can't end without external intervention



Example System Deadlock



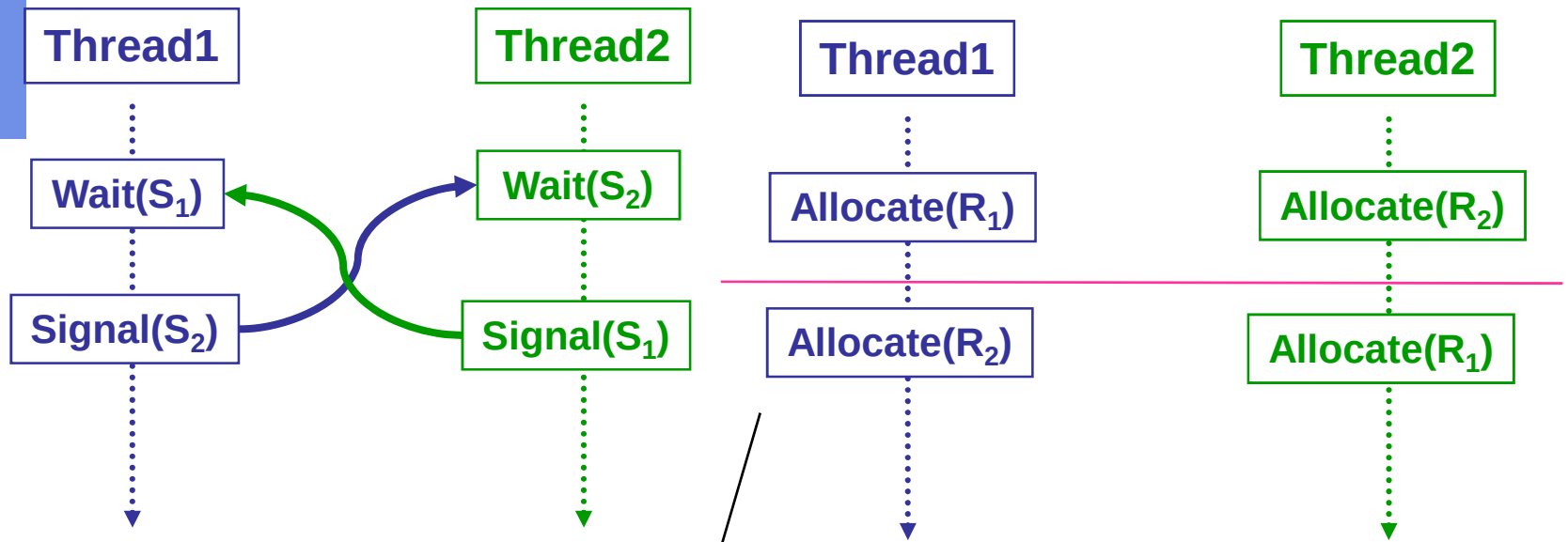
Example 1: Nested CS

```
semaphore s1 = 1, s2 = 1;

Thread T1 {                          Thread T2 {
…                                    …
p(s1); /* outer CS */                p(s2); /* outer CS */
  …                                    …
  p(s2); /* inner CS */                p(s1); /* inner CS */
  …                                    …
  …                                    …
  …                                    …
  v(s2);                               v(s1);
v(s1);                               v(s2);
}                                    }
```

Some authors: “Nested CSs are severe design errors”

Example 2: Signals & Resources

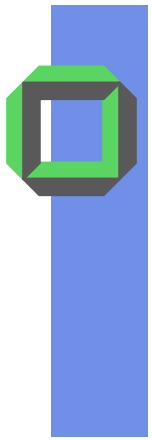


This is a program error

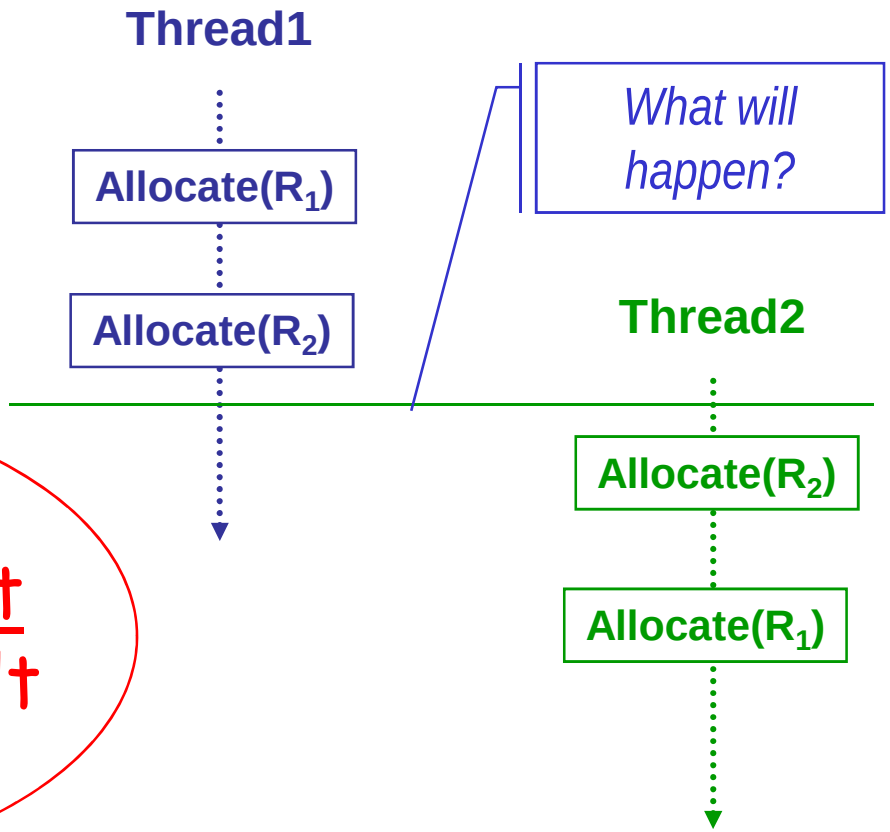
Is this also a program error?

Yes, this timing may lead to deadlock

However, if T1 and T2 belong to different applications?



Example 2

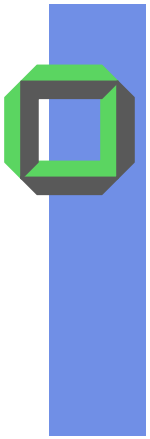


Nevertheless, that program is incorrect because it *can* result in a deadlock.

Is this also a program error?

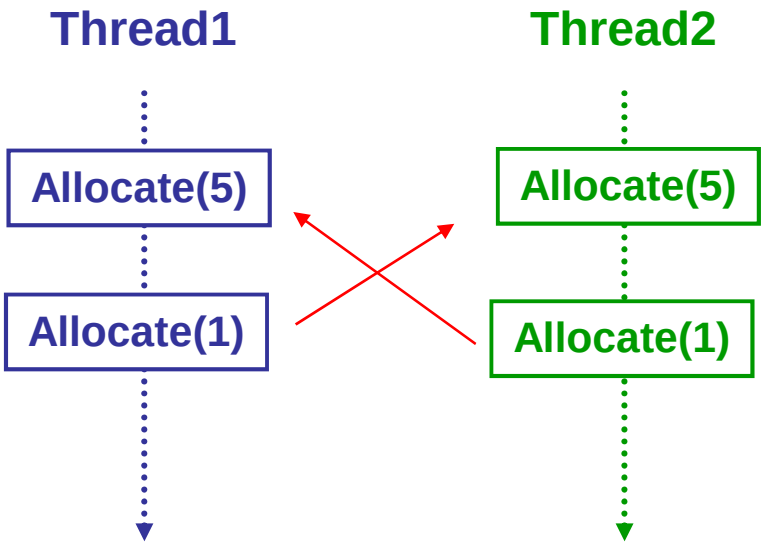
Can also run without a deadlock!

Deadlock analysis always requires “worst case” scenario.

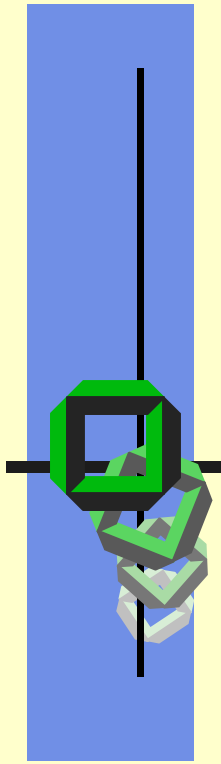


Example 3: Limited Resources

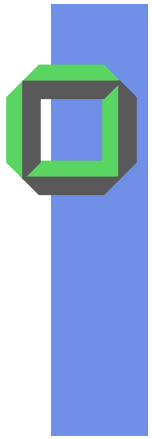
Suppose, system offers **10 memory frames**



Remark: Again, the system can result in an deadlock, but does not run into a deadlock in any case

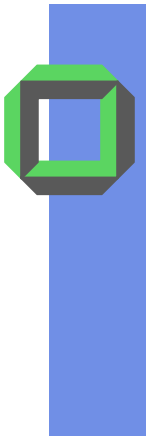


Resource Management



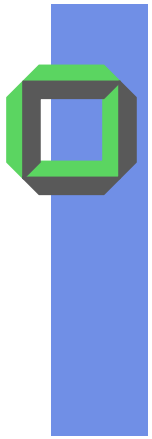
Resources Types

- Logical Resources
  - Lock, Mutex, Semaphore. Buffer
  - Container
  - File
  - Directory
  - ...
- Physical Resources
  - Memory (Cache, Main, Disk, ...)
  - Network
  - Printer
  - Display
  - Keyboard
  - ...

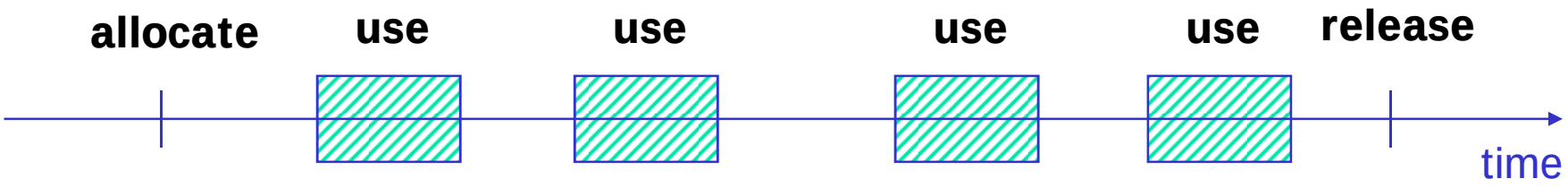


Resource Types

- Resources – passive entities needed by threads to do their work
  - CPU time, disk space, memory, ...
- Two types of resources:
  - Preemptible – can take it away
    - CPU, Embedded security chip
  - Non-preemptible – must leave it with the thread
    - Disk space, plotter, chunk of virtual address space
    - Mutual exclusion – the right to enter a critical section
- Resources may require exclusive access or may be sharable
  - Read-only files are typically sharable
  - Printers are not sharable during time of printing
- One of the major tasks of an OS is to **manage resources**



Utilization Protocol (1)



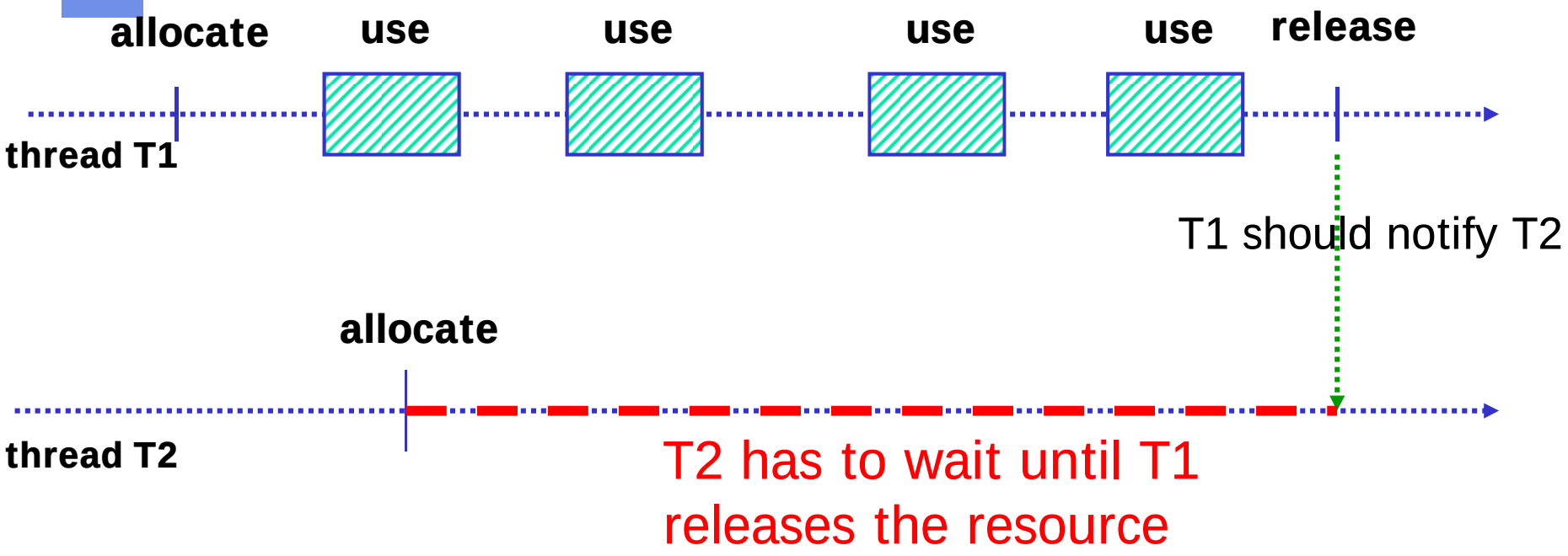
Remark:

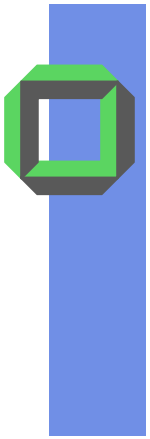
Utilization of an exclusive resource ~ handling a CS

i.e. **`allocate()`** ~ enter_CS(), e.g. **`p()`** and

`release()` ~ leaveCS(), e.g. **`v()`**

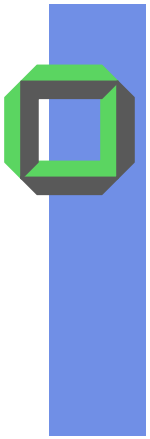
Utilization Protocol (2)





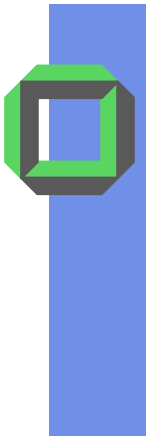
Resources

- Deadlocks can occur when ...
  - processes are granted exclusive access to devices (resources)
- non preemptible resources
  - will cause the process to fail if they are taken away
- preemptible resources
  - can be taken away from a previous resource holder with no side effects



Resource Usage

- Protocol required to use a resource
  1. **`allocate(R)`** resource //request for resource R
  2. **`use(R)`** resource (... several times)
  3. **`release(R)`** resource
- Caller must wait if request can not be fulfilled
- *How to wait?*
  - *In an active loop consuming CPU time?*
  - No, the requesting process should be blocked or
  - it gets the bad news, that the resource is currently held by someone else, so it can do something else in the mean time (trying request)



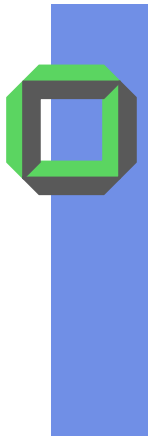
Standalone Resource

A resource manager is some type of a monitor with the following interface operations (methods):

1. **`allocate(r: resource)`**
   `release(r: resource)`

if resource r is of mutual exclusive resource type

Remark:
There are obvious similarities between the topics: resource management and critical sections



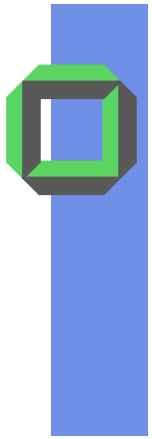
Contiguous Resource

2. **`allocate(r:resource, p:integer, a:address)`**
   `release(r:resource, p:integer, a:address)`

   if the resource can be allocated piece by piece, e.g. frames of RAM or block,

   `p` specifies the number of contiguous frames etc.,

   `a` specifies the address of the first entity.



Pool Resource

3. **allocate(r:resource, p:integer, a: address vector)**

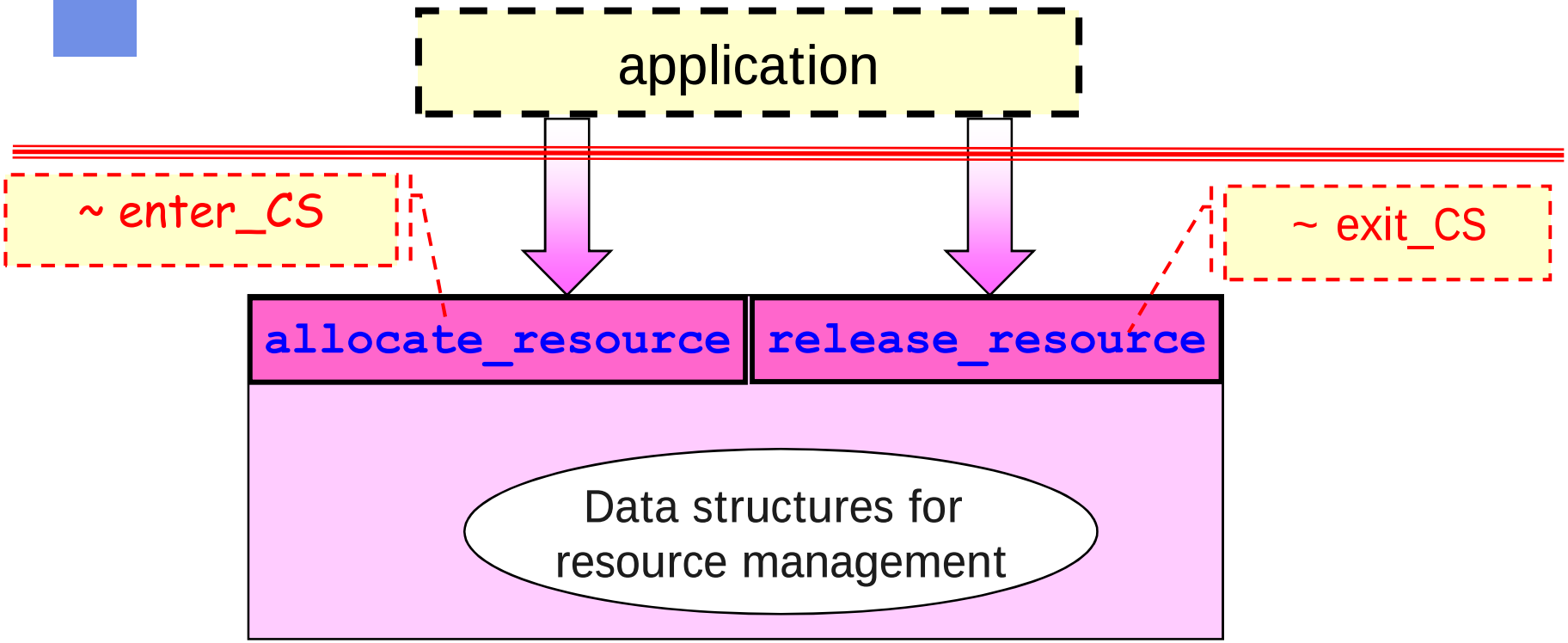
   release(r:resource, p:integer, a:address vector)

   if the resource can be allocated piece by piece,
   e.g. frames of main memory,

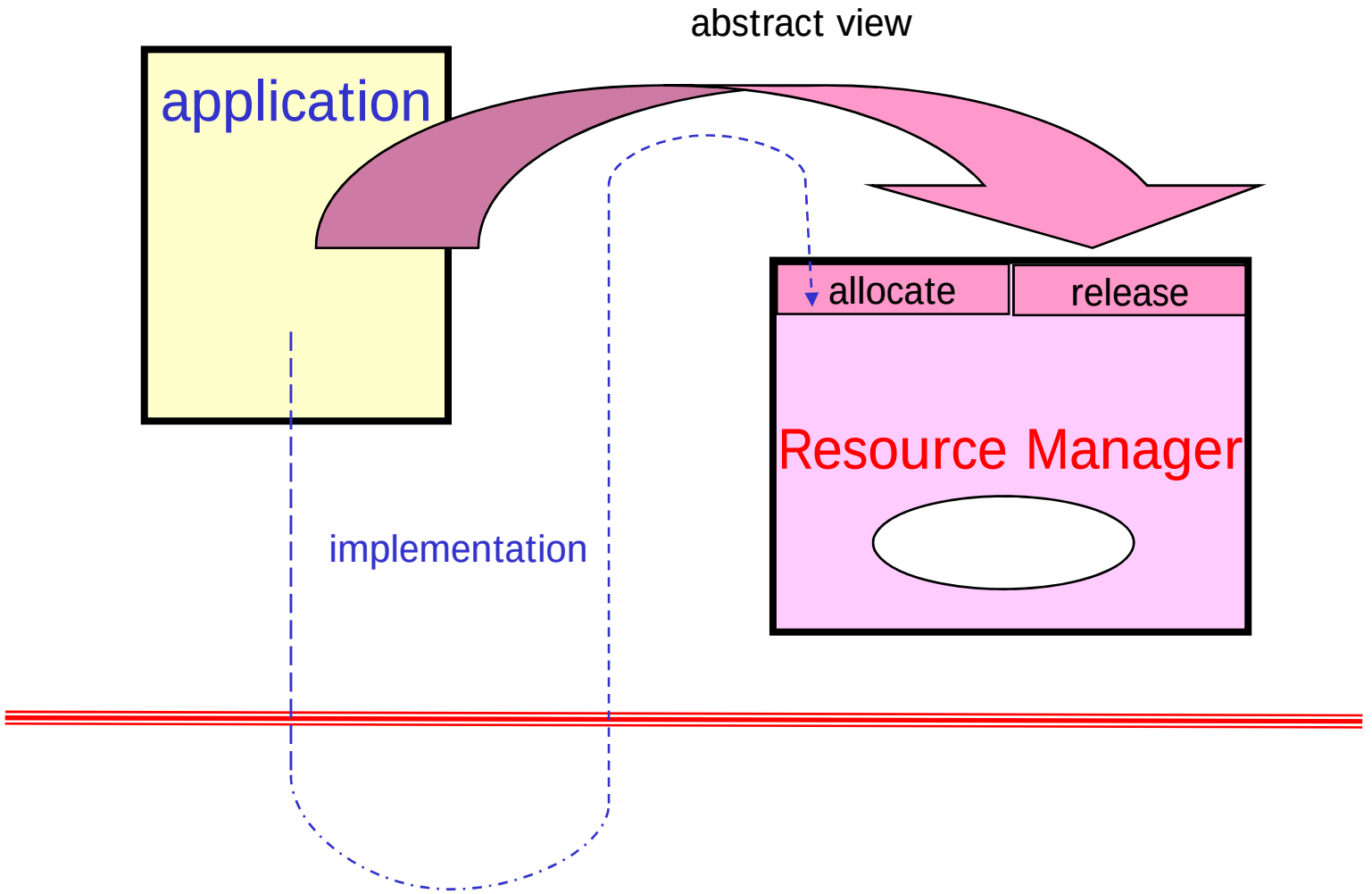
   p specifies the number of needed resources,

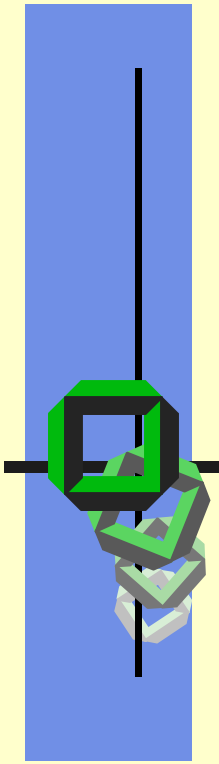
   a specifies the vector of start addresses (RIDs) of the p resources

Kernel based Resource Management



Microkernel based RM

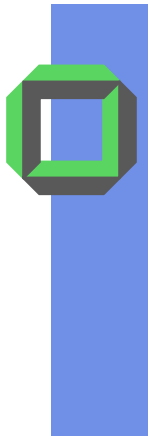




Deadlock & Deadlock Conditions

Necessary Conditions

Do not mix up with the four requirements of a valid solution for the critical section problem

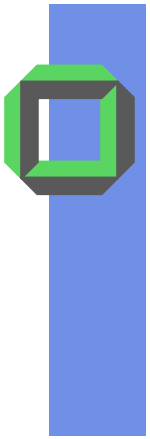


Formal Definition

Formal Definition:

A set S of processes (KLTs) is deadlocked if each process (KLT) in the set S is waiting for an awaking event that only another process (KLT) of the same set S can cause

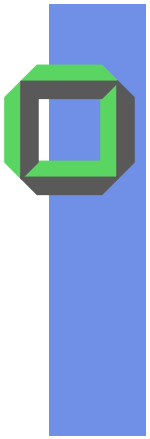
- Usually these “awaking events” , the blocked processes (KLTs) are waiting for, are
  - a **`release()`** notification of a held resource or
  - a **`v()`** operation when exiting a CS or ...
  - ...



Conditions for Resource Deadlock

1. Exclusiveness Only one process can use a resource
   - a resource is either assigned to 1 process or is available
2. Hold and Wait Allocating/releasing of individual resources occur at random (however, always allocate before release)
   - process holding resources can request additional resources
3. No Preemption ...from exclusive resources
   - previously granted resources cannot forcibly taken away
4. Circular Wait Circular dependency
   - must be a circular chain of at least 2 or more processes
   - each waiting for resource held by the next member in the chain

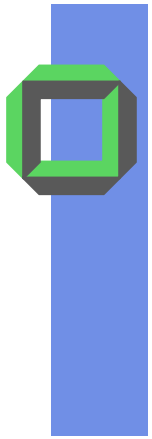
Note: These conditions are necessary, they are not sufficient



Exclusiveness

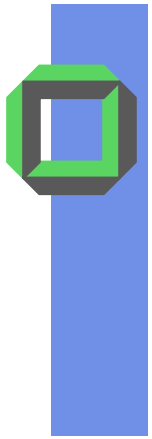
In the system at least one resource must be held in a non sharable mode, i.e. only a single KLT (process) at a time can use this resource

If another KLT (process) requests the resource, the requesting KLT (process) must be delayed (e.g. blocked) until that resource is released by the current resource holder



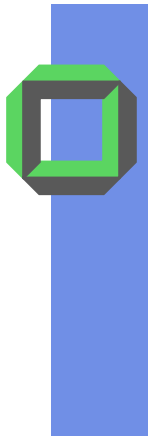
Hold and Wait

∃ KLT (process) holding at least one resource & waiting to acquire additional resources currently held by another KLT (process)



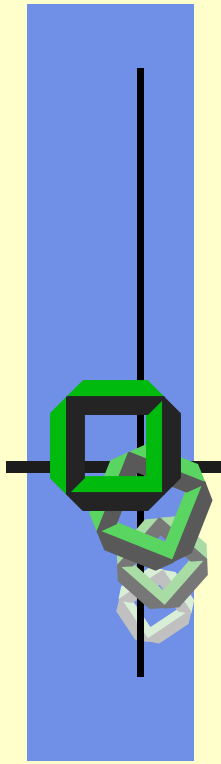
No Preemption

Resources cannot be preempted;
a resource can be released only voluntarily
by the KLT (process) currently holding it.

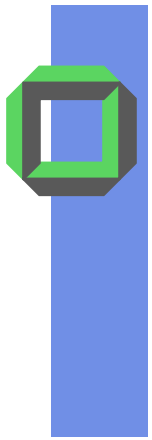


Circular Wait

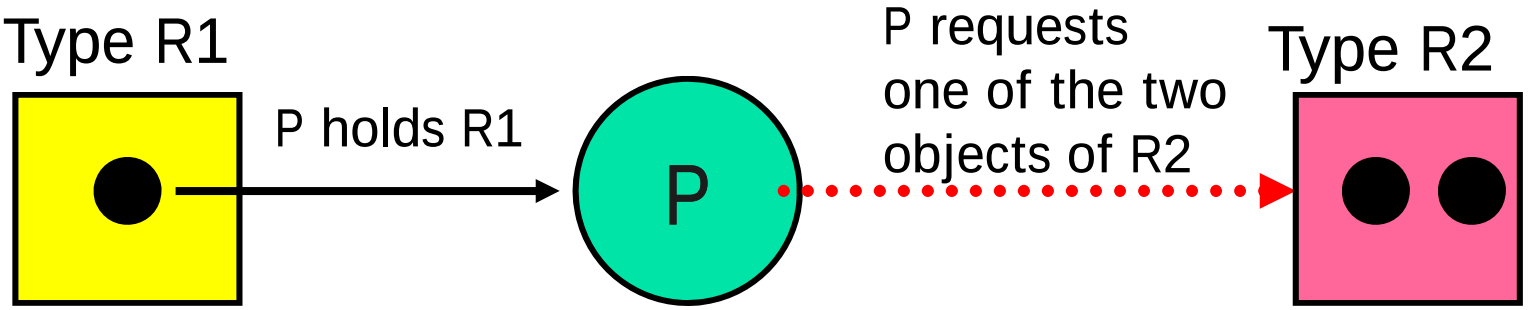
There must exist a closed chain of KLTs (processes) $\{T_1, T_2, \ldots T_k\}$, such that each KLT (process) T_i holds at least one exclusive resource needed by T_{i+1}, the next KLT (process) in this chain!



Visualization of Deadlocks

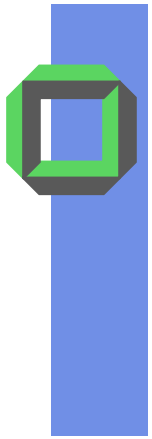


Resource Allocation Graph*

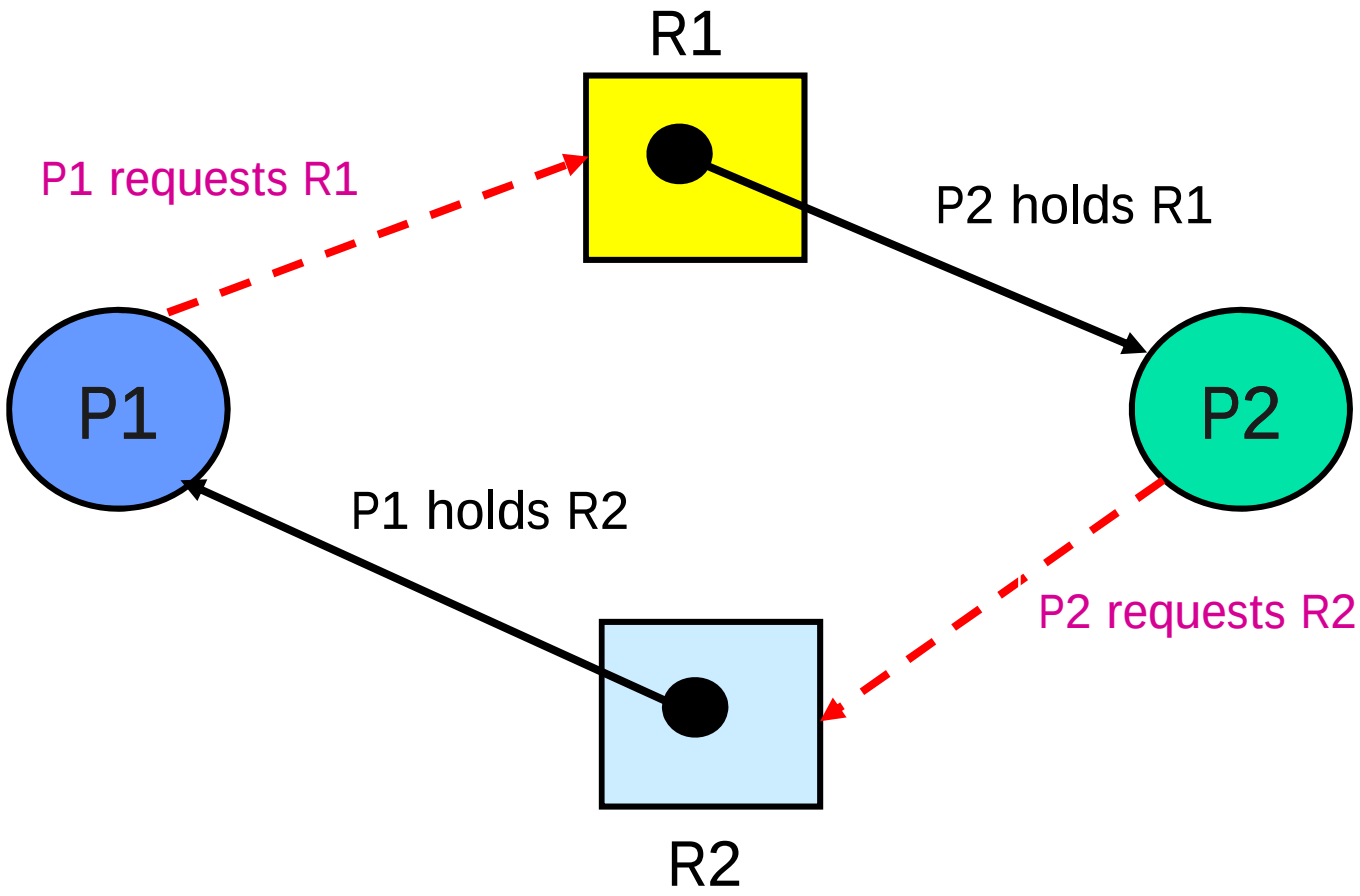


Remark: RAG = “means” to illustrate deadlocks — or situations without a deadlock

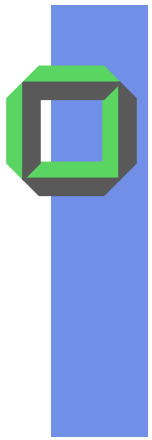
* Peterson, Silberschatz



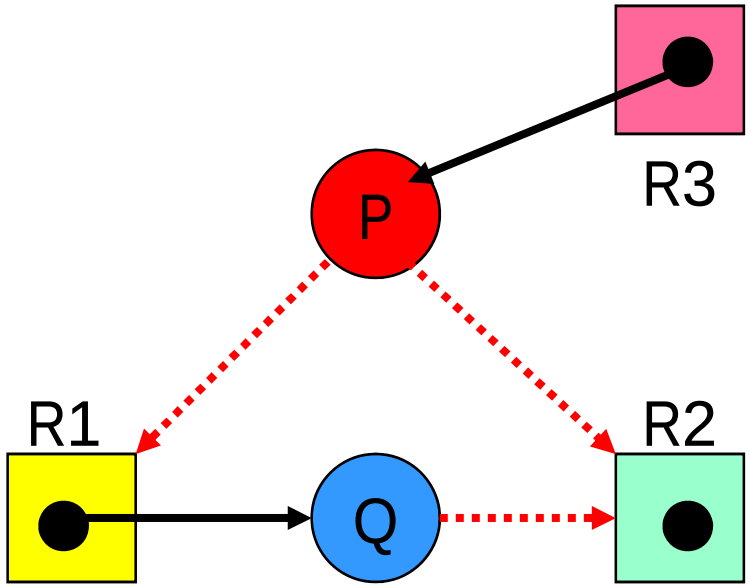
Circular Wait Condition



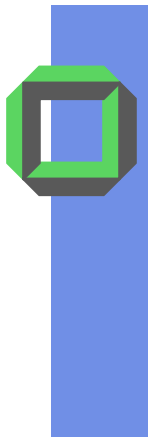
The circular wait condition is sufficient for the existence of a deadlock iff ∃ only one object per resource type (one exemplar systems)



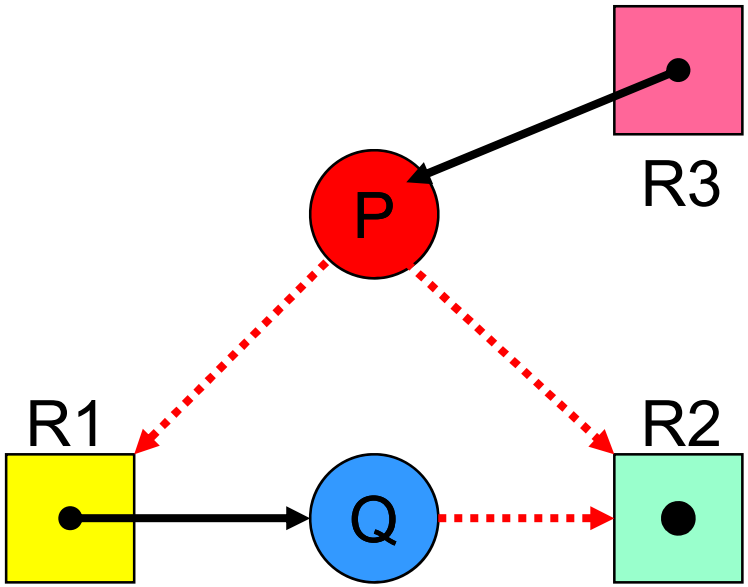
Specific Situations



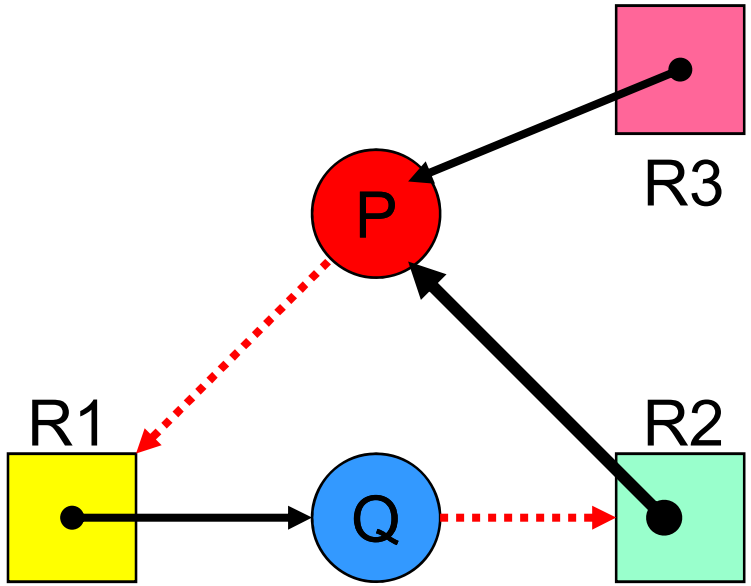
<u>Question:</u>
Can a deadlock occur due to requests from threads P or Q?



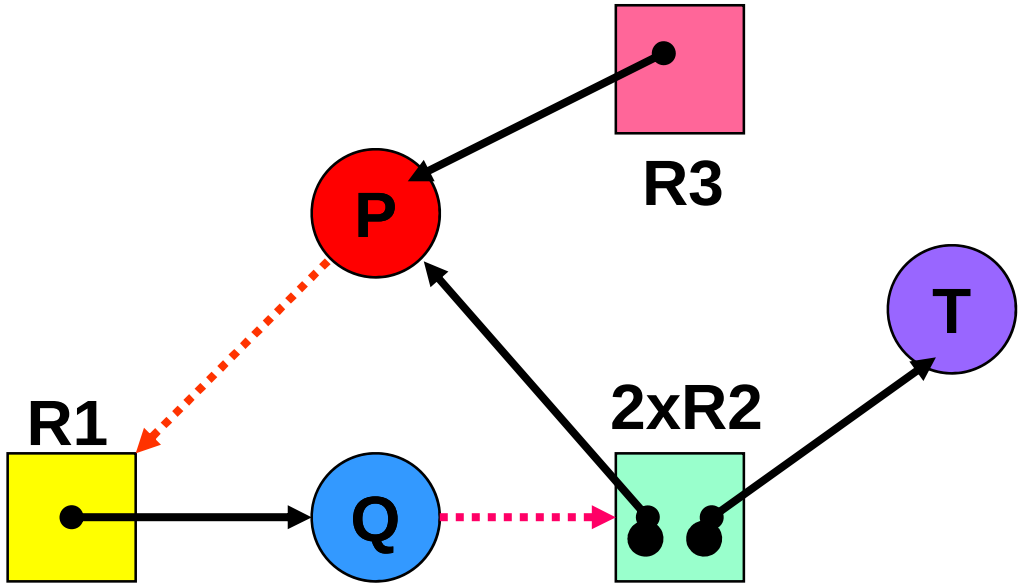
Specific Situations



Yes, if R2 is given to P, then an unsolvable circular wait exists.

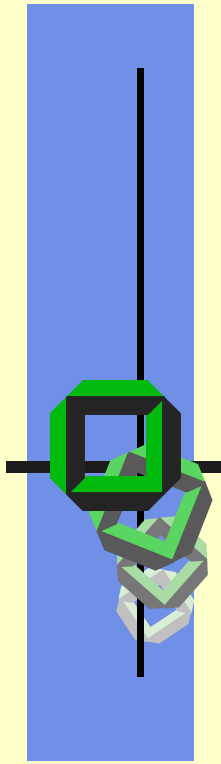


Specific Situations

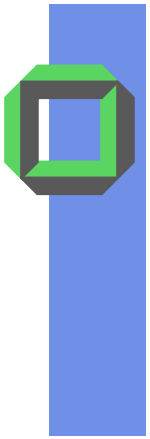


∃ a deadlock due to circle in resource allocation graph?
(Type R2 has 2 objects. One used by P, one by T)

Not necessarily, you can run T, then Q, and finally P.

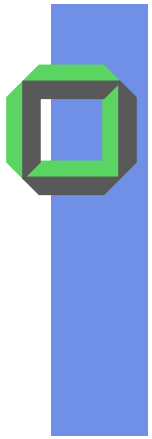


Dealing with Deadlocks



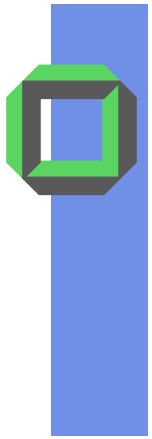
Dealing with Deadlocks

- Ignoring ~ ostrich algorithm
- Detection and Repair
  - Allow a deadlock, but detect it and recover from it
- Avoidance
  - System dynamically considers every request and decides whether it is safe to grant it at that time
  - Maximum requirements of each resource must be stated in advance by each process
- Prevention
  - Preventing deadlocks by constraining how requests for resources can be made in the system and how they are handled (system design)
  - Eliminate one of the four necessary deadlock conditions



Deadlock Detection

- The system may enter a deadlock state
- System needs a deadlock detection algorithm
  - Periodically, e.g. every t time units
  - *However, what is the optimal period length?*
    - If t too small, then huge overhead
    - If t too long, then bad resource usage
  - *Is there a better way to find the "usual suspects?*
- There are two algorithms
  - One resource instance per resource type
  - Multiple resource instances per resource type



Deadlock Detection Algorithm (1)

- One instance per resource type
- Maintain a resource-allocation graph:
  - search for cycles in this resource-allocation grap
  - *Complexity?*

Example: 7 processes A ... G, 6 resources R ...W

Current state: A holds R and requests S

B holds nothing and requests T

C holds nothing and requests S

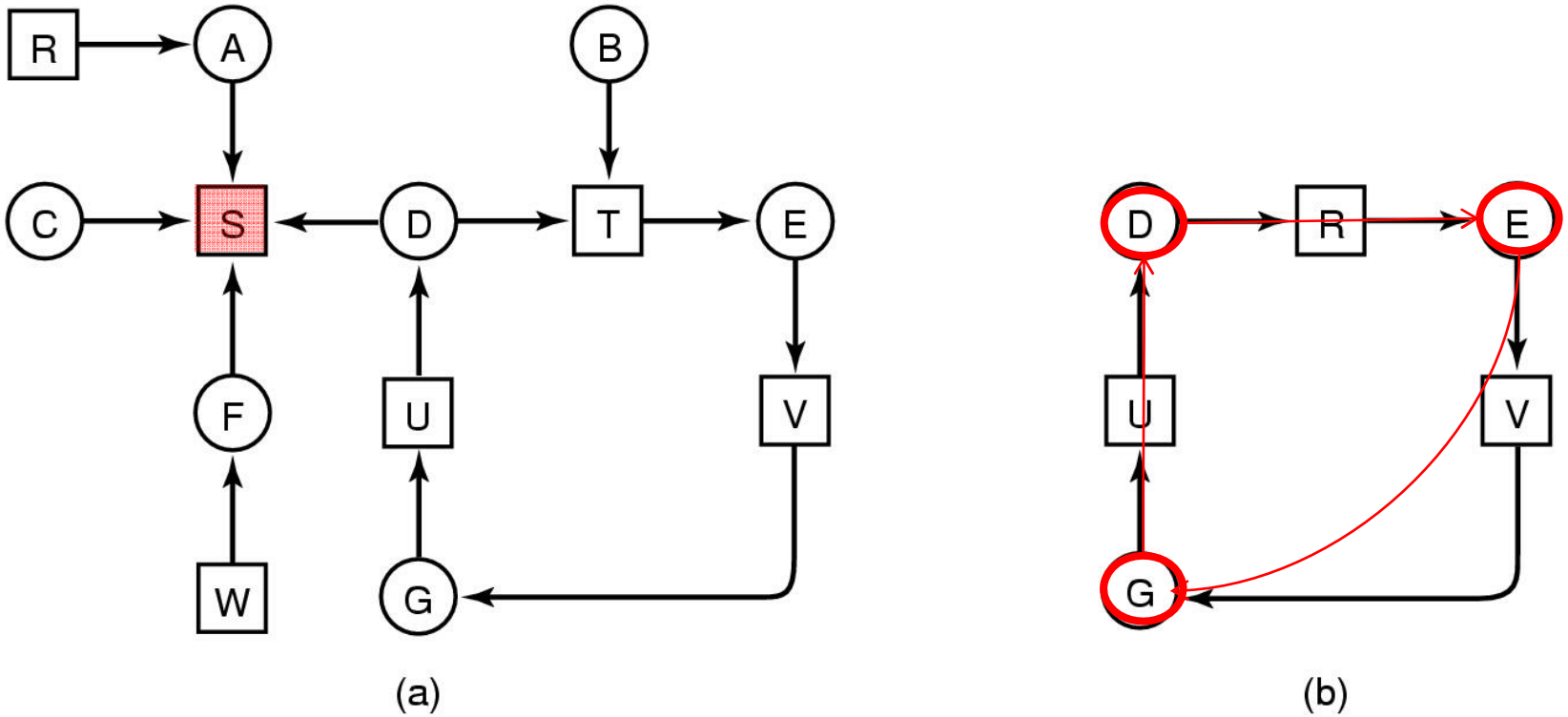
D holds U and requests S and T

E holds T and requests V

F holds W and requests S

G holds V and requests U

Deadlock Detection Algorithm (2)



(a) Example resource allocation graph

(b) Cycle found within the graph → deadlock

Detection N>1 Instances per R-Type

Needed data structures:

n processes $T_1, T_2, \ldots T_n$ and

m resource types $R_1, R_2, \ldots R_m$

R = $(r_1, r_2, \ldots r_m)$ total amount of each **r**esource type

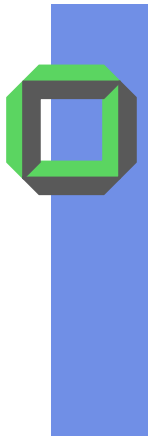
V = $(v_1, v_2, \ldots v_m)$ total amount of each a**v**ailable resource type

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ . & . & \ldots & . \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$$

Currently **allocated** resources of each process per resource type

$$\mathbf{CR} = \begin{pmatrix} CR_{11} & CR_{12} & \ldots & CR_{1m} \\ CR_{21} & CR_{22} & \ldots & CR_{2m} \\ . & . & \ldots & . \\ CR_{n1} & CR_{n2} & \ldots & CR_{nm} \end{pmatrix}$$

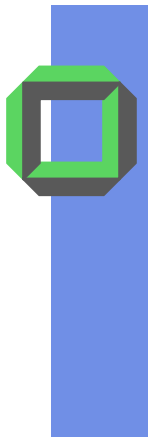
Currently requested resources of each process per resource type



Deadlock Detection Algorithm (3)

```
type state = record
       R,V: array[0..m-1] of integer     //current values

       A,CR: array[o,n-1,0…m-1]of integer
       T: {set of threads}
         end
procedure deadlock_detection
(var answer:state, DT:set of threads) {
answer := undefined              // initialization
DT := T                          // all threads deadlocked
while answer = undefined do
  if ∃T[i] ∈ DT: CR[i] - A[i] ≤ V        // comment 1
    then {
        DT := DT \ {T[i]}                // reduce DT
        V := V + A[i]                    // comment 2
       if DT = {} then answer := safe    // no deadlock
          }
    else answer := unsafe; //DT ={deadlocked processes}
od
}
```



Detection Algorithm (3)

Comment 1:

Look for a process Ti ∈DT with a current request vector which is smaller or equal the available vector per resource type

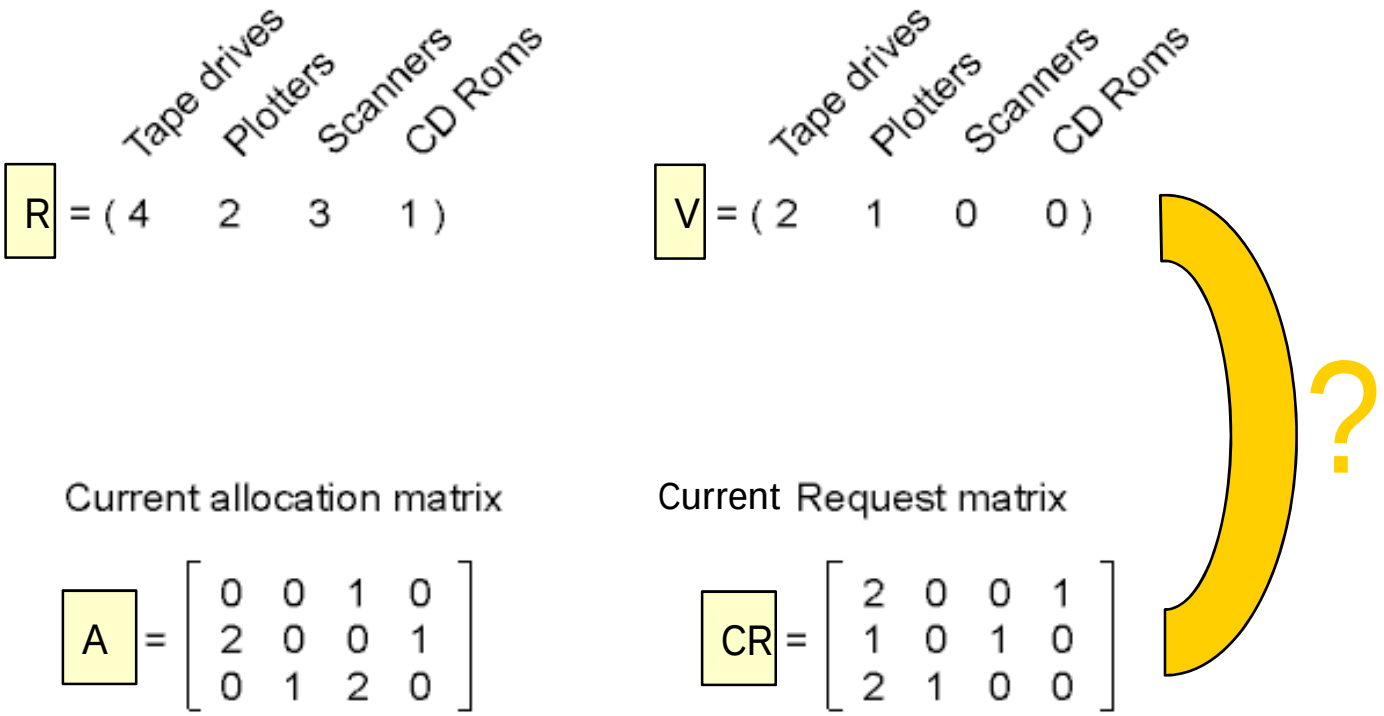
Comment 2:

If such a process exists, add its row Ai to the available vector (we just assume, that this process might execute, whether it will produce a deadlock in the future does not count)

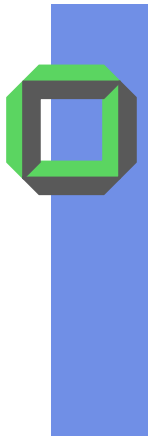
Conclusion:

If no such process exists or if there are some processes in DT left, terminate the algorithm and DT is exactly the set of processes that are currently deadlocked.

Example



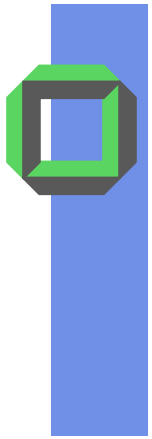
Is there at least one process that can potentially terminate with the available resources?
If you find such a process (i.e. T3), assume that T3 will execute for a while, it may even terminate. If it terminates, it will release additional resources, i.e. the vector v can be incremented by the corresponding row of matrix A, i.e. A3.
Now you try to find another process that can execute, under the assumption that T3 has already terminated etc. (see algorithm)



Deadlock Immunity

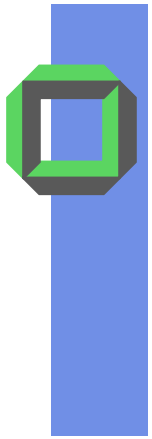
- Enabling systems to defend against deadlocks
  - Nice paper and talk on OSDI 2008 in San Diego
  - H. Jula, D. Tralamazza, C. Zamfir, G. Candea
  - Dependable Systems Laboratory, EPFL
- Basic idea:
  - Learn executions that lead to deadlock
  - Save fingerprints of encountered deadlock patterns into a persistent history
  - Avoid executions patterns that have led to deadlock in the past

http://dimmunix.epfl.ch/



Deadlock Recovery (1)

- Recovery via resource preemption
  - take a resource from some other process
  - depends on nature of the resource
- Recovery through rollback
  - checkpoint a process periodically
  - use this saved state
  - restart/resume process at checkpoint



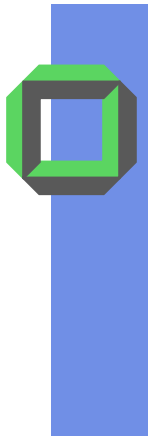
Recovery from Deadlock

What can athesystem do after it has detected a deadlock?

- Process termination
  - Abort all deadlocked processes:
    - Quite fast (as long as #deadlocked processes is not too big
    - That's simple, but a lot of process work has been wasted
  - Abort one deadlocked process at a time and check for deadlock again
    - More work to resolve a deadlock
    - Better in terms of process' work
    - *But in which order to abort the processes, i.e. which one to abort first?*
  - Resource preemption
    - *What is a good way to select a victim thread to be aborted?*
    - *How can we rollback and recover from preemption?*
    - *How can we protect from starvation?*

Recovery (2)

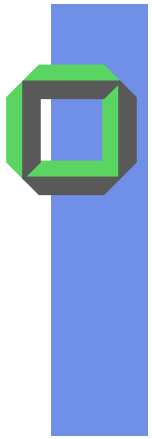
- Recovery through aborting processes
  - crude but simple way to break a deadlock
  - kill one of the processes in the deadlock cycle until the other processes get their resources
  - choose a process that can be rerun from the beginning



Deadlock Recovery (3)

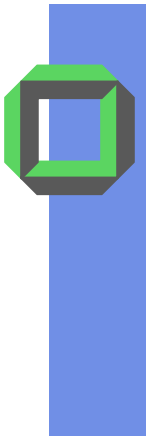
What possibilities for aborting processes?

- Some criteria for aborting processes:
  - Size of pending request
  - Amount of allocated resources
  - Priority
  - Application-/system-process
  - Expected duration of abortion
  - Accumulated execution time
  - Expected remaining execution time
  - ...



Deadlock Avoidance

- We must know the maximum requirements of each resource type per process before starting the process
- Two approaches:
  - Do not start a process if its maximum requirement might lead to deadlock.
  - Do not grant an incremental resource request if this allocation might lead to deadlock
  - Two algorithms:
    - One instance per resource type, the resource allocation graph algorithm
    - Multiple instances per resource type: Banker algorithm



Deadlock Avoidance (1)

Some formal description of the system

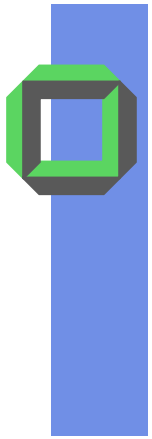
Given:

n processes T_1, T_2, ... T_n and

m resource types R_1, R_2, ... R_m

R = $(r_1, r_2, \ldots r_m)$ total amount of each **r**esource type

V = $(v_1, v_2, \ldots v_m)$ total amount of each a**v**ailable resource type, i.e. currently not allocated to one of the n processes



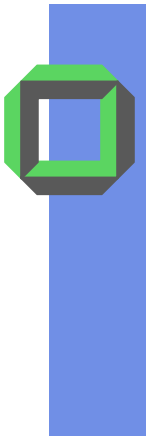
Deadlock Avoidance (3)

$$\mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ . & . & \dots & . \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{pmatrix}$$

Maximally **claimed** resources per resource type of each process

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ . & . & \dots & . \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$$

Currently **allocated** resources of each process per resource type



Invariants and Constraints

$$R_i = V_i + \sum_{k=1}^{n} A_{ki}$$

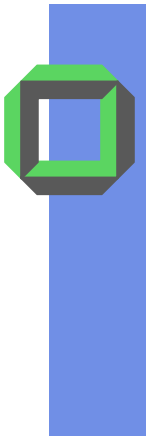
for all i: All resources of a resource type R_i are either available or allocated

$$C_{ki} \leq R_i, \text{ for all } k,i:$$

no process claims more than the total amount of resources in the system

$$A_{k,l} \leq C_{k,i}, \text{ for all } k,i:$$

no process tries to allocate more resources than initially claimed



Deadlock Avoidance: Deferred Start

Start a new process T_{n+1} iff

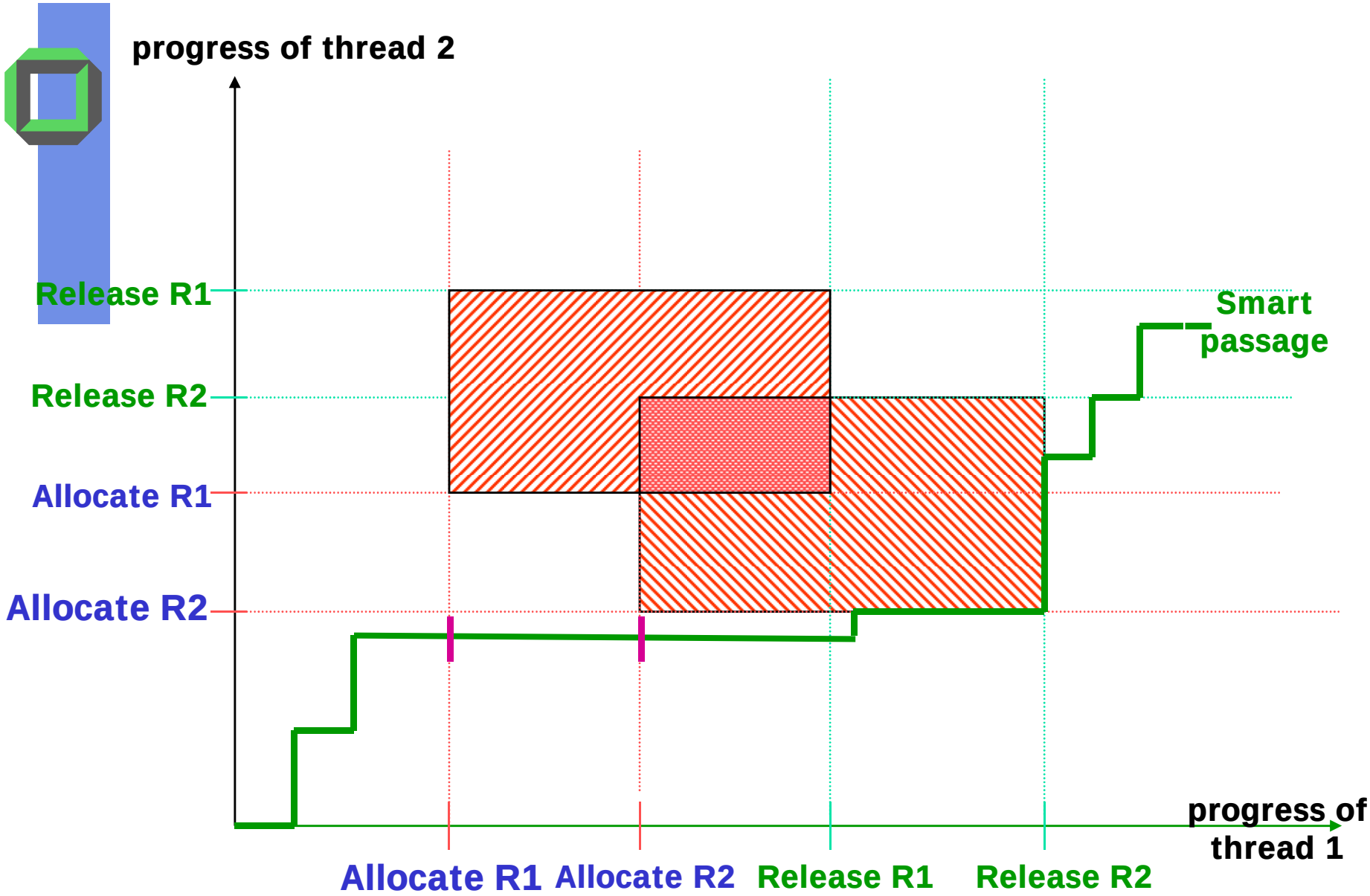
$$R_i \geq C_{n+1,i} + \sum_{k=1}^{n} C_{ki} \qquad \text{for all i}$$

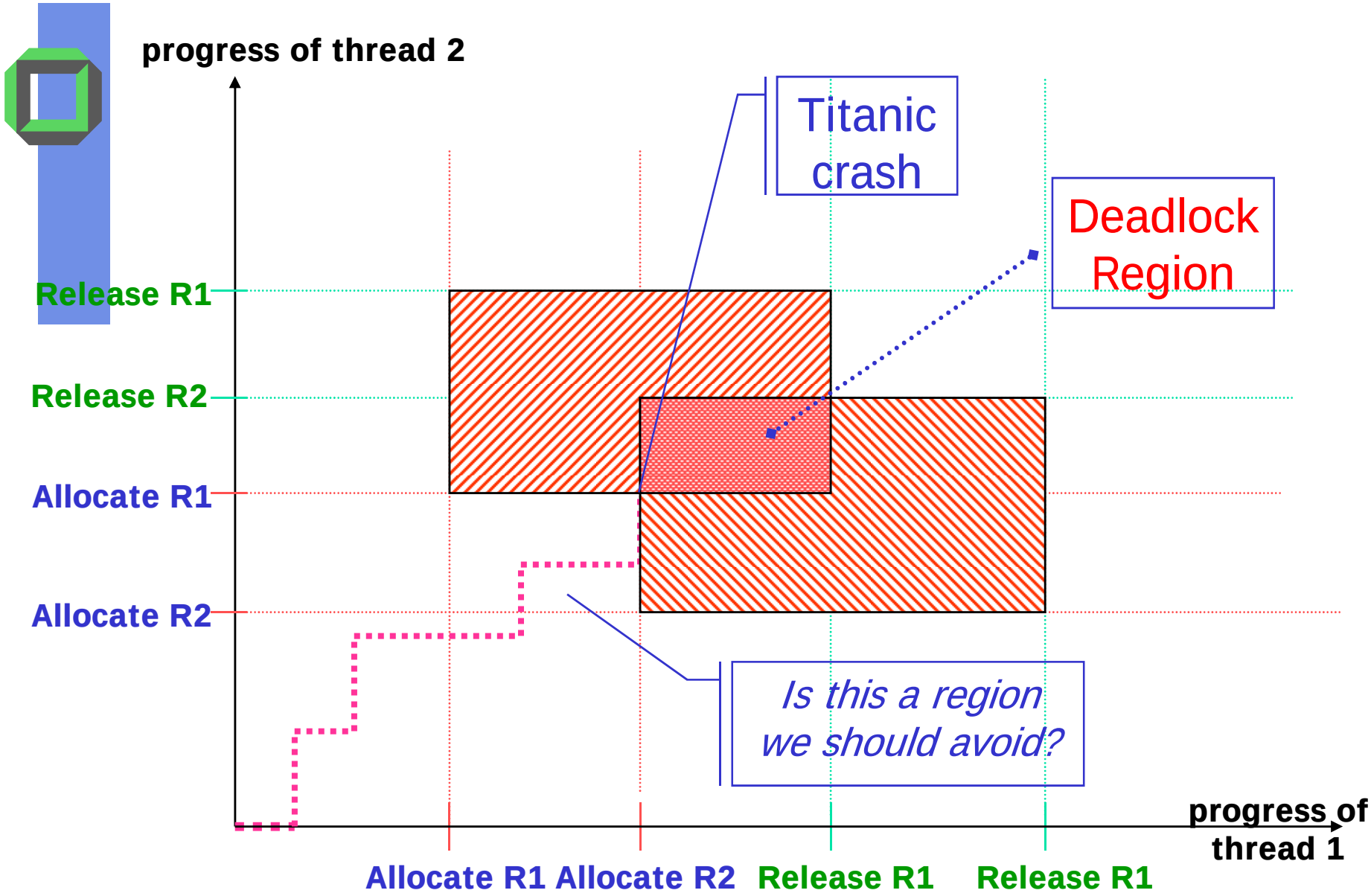
Policy is quite pessimistic assuming that all threads will request all their claimed resources at the same time.

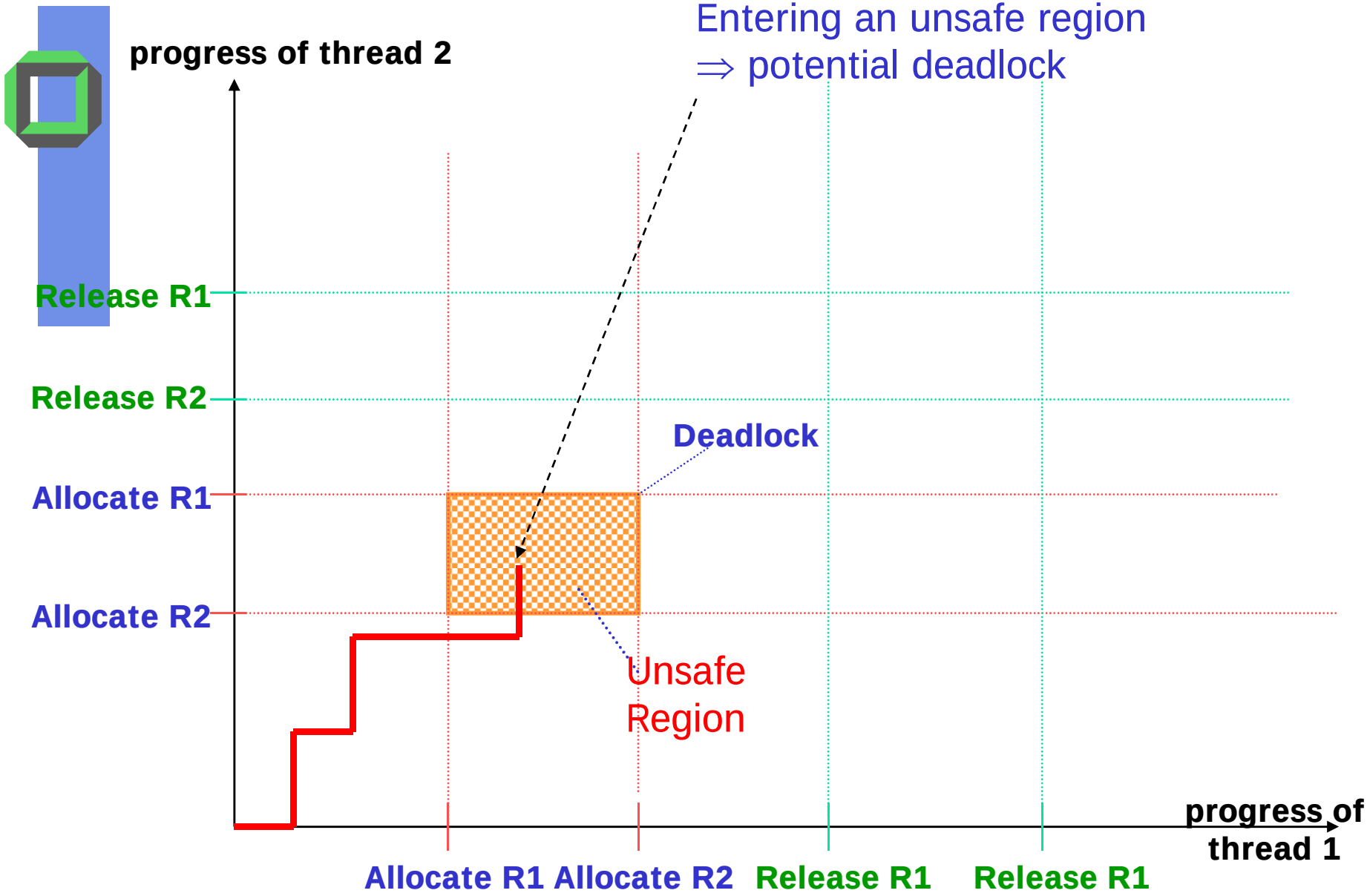
In practice, however, the following often holds:

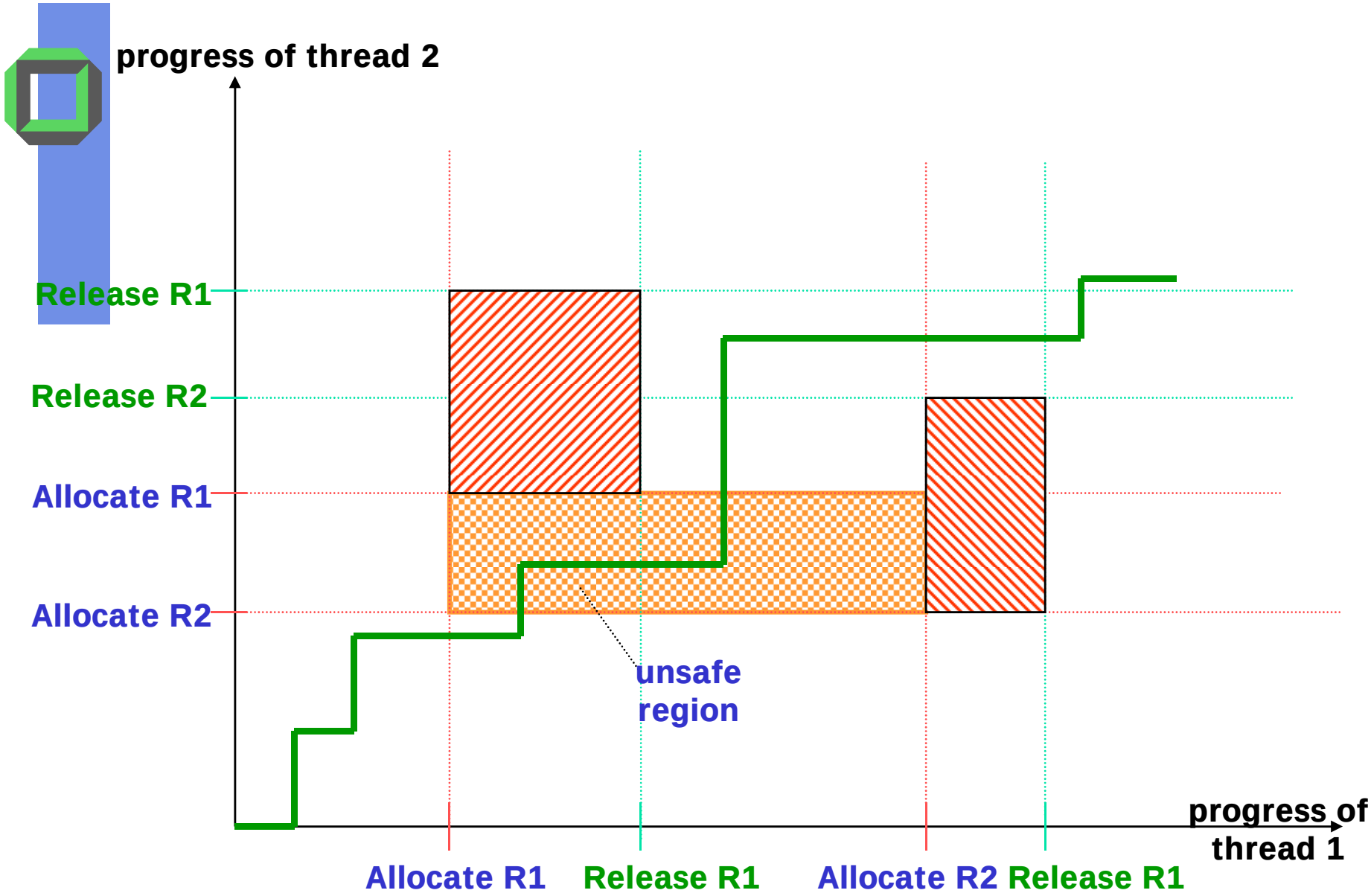
- Some of the claimed resources are never requested
- Few threads need all resources at the same time

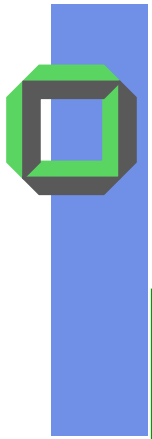
$\Rightarrow$ we need a better, i.e. more optimistic algorithm











Safe State

Definition: The state of a system of n threads is safe as long as there is at least one execution sequence allowing all threads to complete.

More formally:

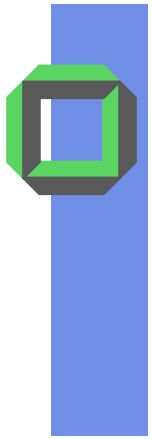
System state = safe $\Leftrightarrow$

$\exists$ permutation $<T_{k1}, T_{k2}, \ldots T_{kn}>$ within $\{T_1, T_2, \ldots, T_n\}$:

for all $i \in \{1,2, \ldots n\}$: $$C_{k_i} - A_{k_i} \leq V + \sum_{s=1}^{i-1} A_{k_s}$$

or

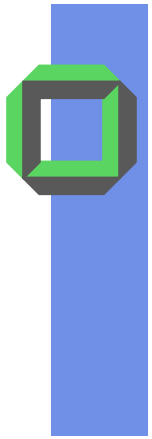
for all $i \in \{1,2, \ldots n\}$: $$C_{k_i} - A_{k_i} \leq R - \sum_{s=i}^{n} A_{k_s}$$



Properties of „Safe State“

- If a system is in safe state, there is no deadlock
- If system is deadlocked, it is in an unsafe state
- If a system is in unsafe state, there is a possibility for a deadlock
- Avoidance:

  Make sure the system will not enter an unsafe state

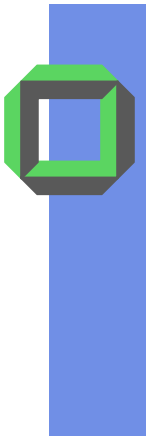


Banker Algorithm with $O(n^2m)$

```
type state = record
               R,V: array[0..m-1] of integer
               C,A: array[o,n-1,0…m-1]of integer
               T: {set of threads}
             end
procedure deadlock_avoidance
(var answer:state, DT:set of threads) {
answer := undefined          /* initialization */
DT := T                      /* all threads deadlocked */
while answer = undefined do
  if ∃T[i] ∈ DT: C[i] - A[i] ≤ V
    then {
        DT := DT \ {T[i]}
        V := V + A[i]
       if DT = {} then answer := safe
          }
    else answer := unsafe  /* n≥1 thread deadlocked */
od
}
```

Reduktion der Threadmenge T bzw. Aufbau der Sequence

$O(n^2m)$ with n = # of threads and m = # of resource types

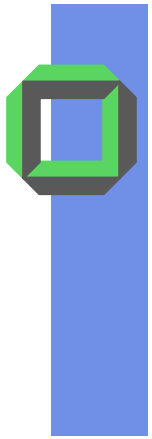


Deadlock Prevention (1)

Construct a system that prevents deadlocks, i.e. it has to guarantee that deadlocks can never arise. *How?*

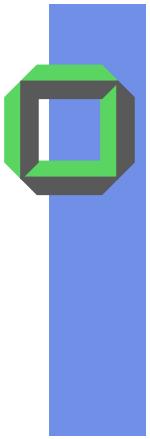
- Ensure that at least one of the necessary conditions for deadlocks can not occur

- Attacking the mutual-exclusion condition?
  - Some physical and logical resources require mutual exclusion, e.g. keyboard, stack etc.
  - We have to admit mutual exclusion
  - However, other resources like .....



Spooling or Multiplexing

- Some devices can be spooled
  - Applications do not access the printer directly, but a substitute resource, the application specific output file
  - The printer daemon is the only activity that is allowed to access the real printer
  - No resource conflicts with a “spooled” printer
- Other devices can be virtualized, e.g.
  - multiplexing the CPU

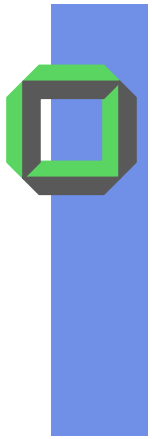


Deadlock Prevention (2)

- Attacking the hold-and-wait condition?
  - Requires that each thread requests all resources by one combined atomic request

    Consequence: Thread must wait for a long time until all requests can be granted at the same time

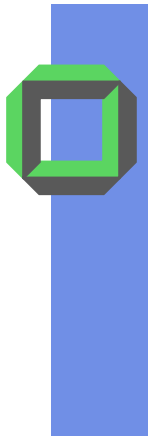
  - Disadvantages:
    - Thread has waited for a long time to get all its resources, even though it might not use all of them in the next run
    - Allocated resources can remain unused for a long period. Others threads could use them in the meantime.



Deadlock Prevention (3)

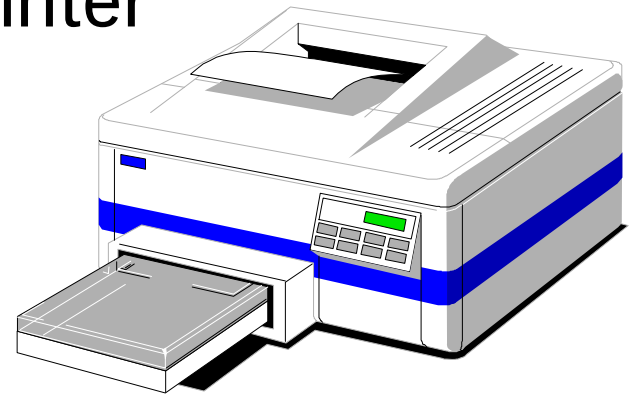
- Attacking the no-preemption condition?
  - Preemption: if a resource request is denied, the calling thread should release all resources that it already holds
  - Alternatively you could preempt the current resource holder to release its resources.
  - The state of the preempted resource has to be saved for later resumption

    $\Rightarrow$ It should be easy to save/restore the state of a resource
  - Resource preemption can lead to starvation

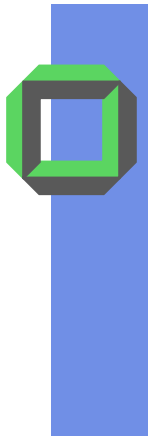


Preemptive Printer?

- This is not a viable option
- Consider a process given the printer
  - halfway through its job
  - now forcibly take away printer
  - !!??

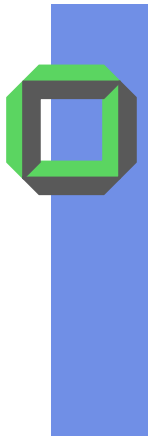


Hint: Try to find a solution for preempting a printer, even though Andy Tanenbaum does not like it



Deadlock Prevention (4)

- Attacking the no-circular-wait condition?
  - Define a strictly increasing linear ordering OR(RT) for resource types, e.g.
    - R1: tape drives: OR(R1) = 2
    - R2: disk drives: OR(R2) = 4
    - R3: printers: OR(R3) = 7
  - A process initially requests a number of instances of a resource type, say R_i
  - A single request must be issued to obtain several instances of the same resource type
  - After that, this process can request instances for resource types R_j if and only if $OR(R_j) > OR(R_i)$



Summary: Deadlock Prevention

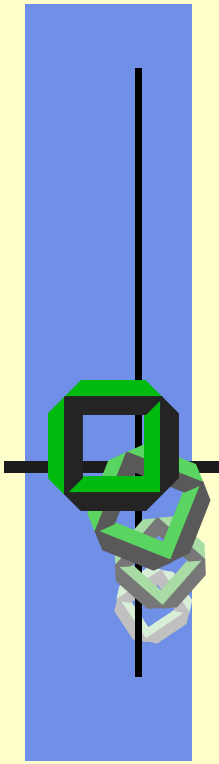
In general, this principle is either far too restrictive
(you only request resources in some predefined ordering)

or the usage of the resources is far too low
(allocating all resources at the start is quite wasteful)

or too much overhead is involved
(e.g. a preemption costs time and requires additional containers for saving and restoring the resource states)

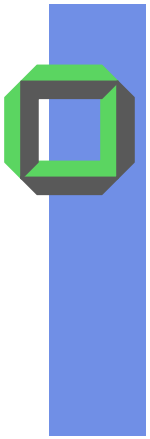
⇒

commercial systems do not use prevention



Approaches in the Practice

Additional problems

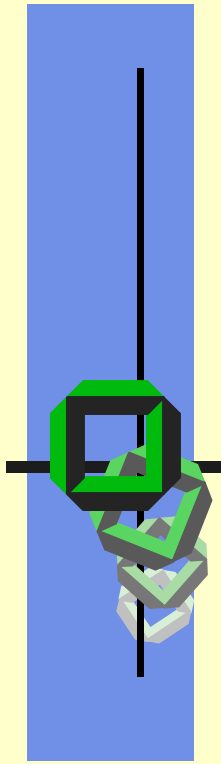


Combined Deadlock Policy

We can combine some previous approaches

in the following way:

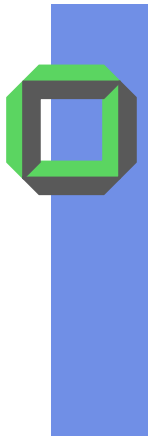
- Group resources into a number of different classes and order them, e.g.:
  - Swappable space (secondary memory)
  - Task resources (I/O devices, files...)
  - Main memory...
- Use prevention of circular wait to prevent deadlock between these resource classes
- Use the most appropriate approach against deadlocks within each class



Two Phase Locking & Non Resource Deadlocks

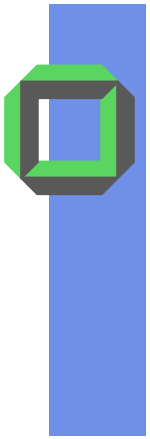
See related Course in ST 2009

„Kommunikation und Datenhaltung“



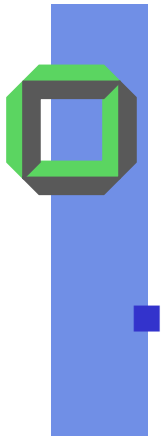
Two-Phase Locking

- Phase One
  - process tries to lock all records it needs, one at a time
  - if needed record found locked, start over
  - (no real work done in phase one)
- If phase one succeeds, it starts second phase,
  - performing updates
  - releasing locks
- Note similarity to requesting all resources at once
- Algorithm works where programmer can arrange
  - program can be stopped, restarted



Non Resource Deadlocks

- Possible for two processes to deadlock
  - each is waiting for the other to do some task
- Can happen with semaphores
  - each process required to do a `p()` on two semaphores (mutex and another)
  - if done in wrong order, deadlock still results
- Can happen with nested monitors
- An ordering on semaphores and monitors within one application can be useful, see nested critical sections



Starvation

- Algorithm to allocate a resource
  - may be to give to shortest job first
- Works great for multiple short jobs in a system
- May cause long job to be postponed indefinitely
  - even though not blocked
- Solution:
  - First-come, first-serve policy