Operating Systems 2019
Deadlocks

October 16, 2019 —

Motivation

- some resources can be used by a single process at a point in time
- exclusive access necessary
- often: exclusive access to multiple resources required
- sufficiently unfortunate order: deadlock possible
  - not only with operating systems

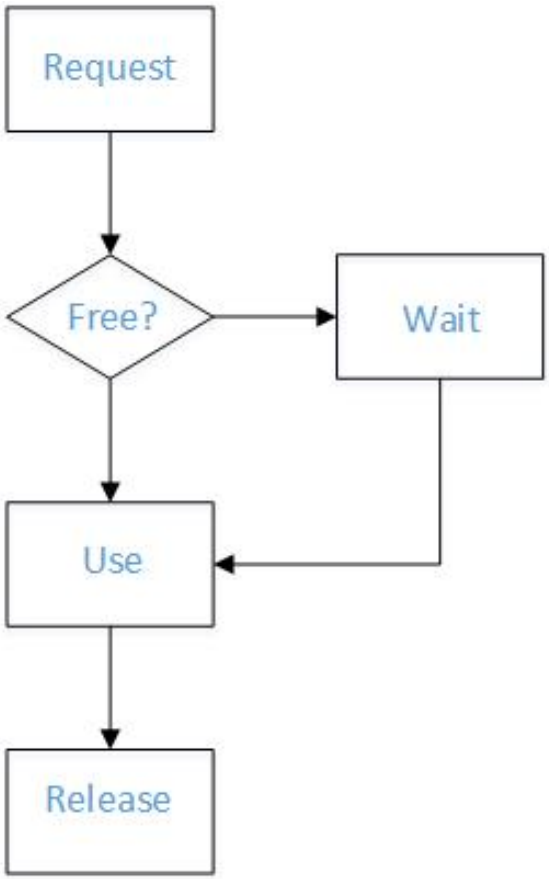
Resources

- Exclusive access required for a deadlock to be possible
- Resource: anything that might be used by a process
- two classes of resources:
  - preemtable: can be taken away without potential side-effect
    - Example: memory
  - non-preemptable: cannot be taken away without potential side-effect
    - Example: semaphore

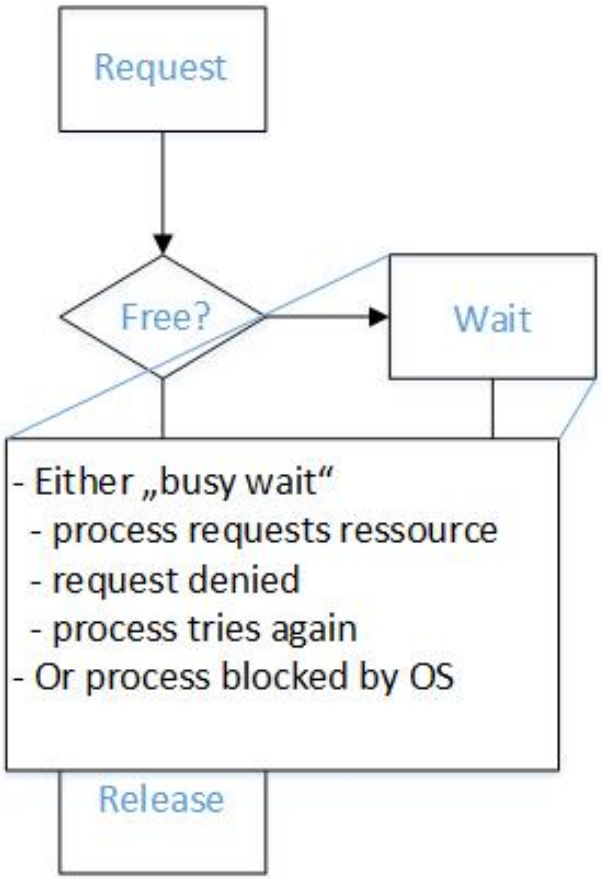
Non-Preemptable Resources

- Deadlocks involve non-preemptable Resources only
- potential deadlock involving preemtable Resources can be solved by reallocation

Deadlock-”Work-flow”



Deadlock-”Work-flow”



Request

- How to request a resource?
- depends on system and resource
  - special requests (like semaphores)
  - use files
  - up to the user

Up to the user?

- General solution: a semaphore or mutex per resource

```
wait(Resource_1);
use_Resource();
signal(Resource_1);
```

Process requires 2 Resources

```
wait(Resource_1);
wait(Resource_2);
use_Resource();
signal(Resource_2);
signal(Resource_1);
```

Multiple processes

```
1  wait(Resource_1);
2  wait(Resource_2);
3  use_Resource();
4  signal(Resource_2);
5  signal(Resource_1);
```

```
1  wait(Resource_1);
2  wait(Resource_2);
3  use_Resource();
4  signal(Resource_2);
5  signal(Resource_1);
```

Multiple processes

```
1  wait(Resource_1);
2  wait(Resource_2);
3  use_Resource();
4  signal(Resource_2);
5  signal(Resource_1);
```

```
1  wait(Resource_2);
2  wait(Resource_1);
3  use_Resource();
4  signal(Resource_2);
5  signal(Resource_1);
```

Deadlock-Examples

Java JDK 1.6 deadlocks:

- Vector: Concurrently call v_1.addAll(v_2) and v_2.addAll(v_1)
- Hashtable: With h_1 a member of h_2 and h_2 a member of h_1, concurrently call h_1.equals(foo) and h_2.equals(bar)
- StringBuffer: With StringBuffers s_1 and s_2, concurrently call s_1.append(s_2) and s_2.append(s_1)

From: Jula et.al., Deadlock Immunity: Enabling Systems To Defend Against Deadlocks

Deadlock

Formal definition

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.

Assumptions: processes, threads - both may be deadlocked. Number of threads, types of resources relevant.

Conditions for deadlocks (1)

Mutual Exclusion condition

Each resource is either currently assigned to exactly one process or is available.

Conditions for deadlocks (2)

Hold-and-wait condition

Processes currently holding resources that were granted earlier can request new resources

Conditions for deadlocks (3)

No-preemption condition

Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them

Conditions for deadlocks (4)

Circular wait condition

There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain

Conditions for deadlocks

All four conditions must be present for a deadlock to occur

- Mutual Exclusion condition
- Hold-and-wait condition
- No-preemption condition
- Circular wait condition

Strategies to deal with deadlocks

- Ignore it (maybe it ignores us too...)
- Detection and Recovery
- Avoidance
- Prevention

Ignoring - the “Ostrich Algorithm”

Mathematical Approach

We MUST prevent deadlocks!

Engineering Approach

- How often does the problem occur?
- How expensive is it to solve?
- Let’s do a cost-benefit analysis!

Ignoring as Best Practice

- Unix, Windows: the problem is ignored
- Cost to prevent deadlocks too high
- Prevention may not be possible at all
- Even detection is too expensive
- Weigh “comfort” versus “correctness”

Example

- Resources in OS are limited
- limited number of processes or open files at any time
- assume: all active process need to do another fork or open one more file
- None are available –> deadlock!
- Now how likely is that?

Detection and Recovery

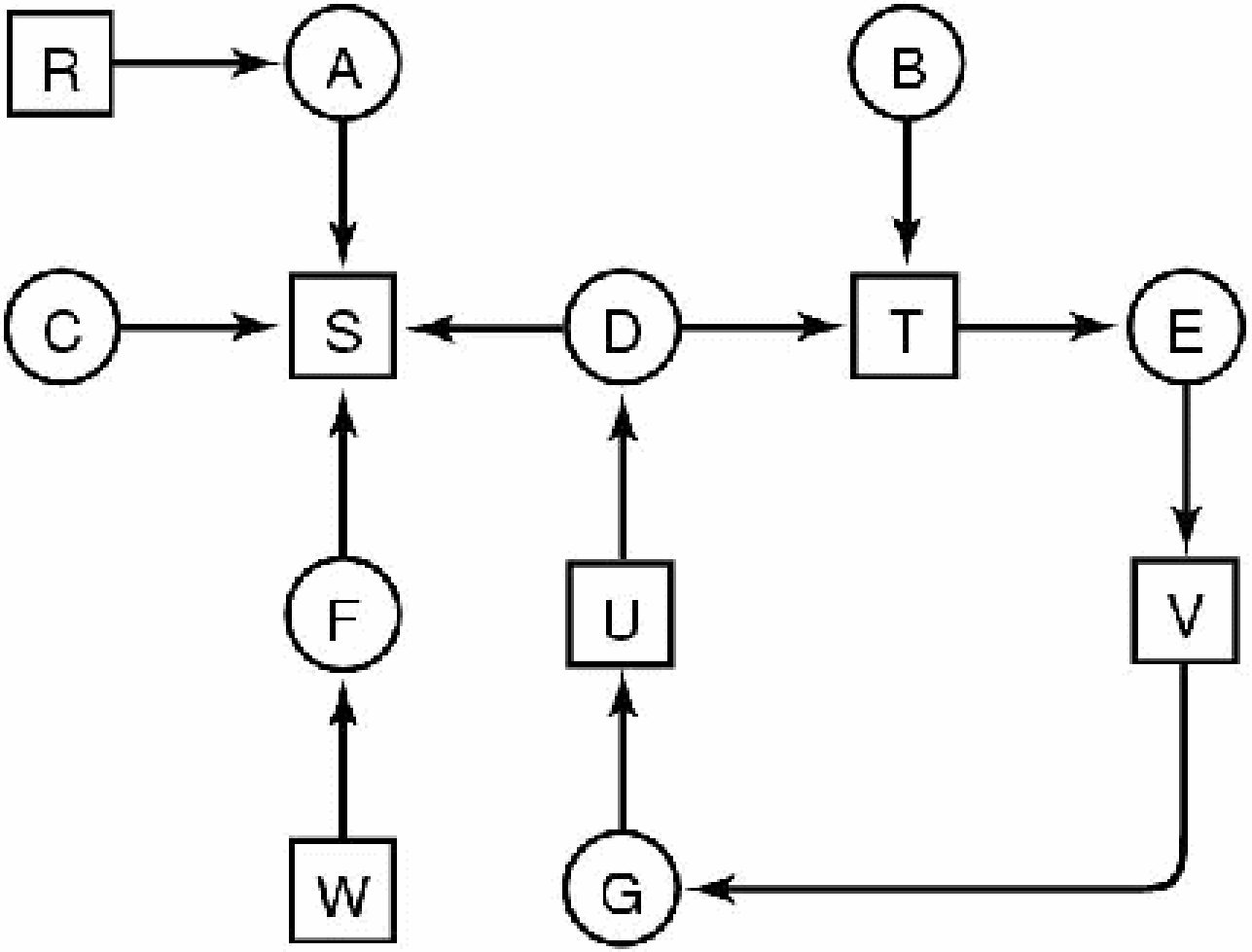
- Don’t prevent occurrence
- try to detect occurrence and deal with it when it happens
- how can we do that?
- e.g.: “draw” resource-graphs and detect circles

Detection

- Example: is the following system deadlocked?

```
1 Process A holds R and wants S
2 Process B holds nothing but wants T
3 Process C holds nothing but wants S
4 Process D holds U but wants S and T
5 Process E holds T but wants V
6 Process F holds W but wants S
7 Process G holds V but wants U
```

Detection



Detection

- easy - visually
- but there is an algorithm too
- many algorithms for detecting cycles in directed graphs

Simple Algorithm

Depth-first search in a tree

- take each node as the root of a tree
- do a depth-first search
- if we ever come back to a node we have already been to: cycle found
- when we have visited all arcs from a node: backtrack one level up
- back to start: no deadlock found
- need to try for all nodes as roots

not quite optimal

Detection

- When do we check for deadlock?
  - each request? (earliest detection, expensive)
  - every x minutes?
  - nothing else to do (or low CPU workload)?

Recovery

- Deadlock detected. What now?
  - Preemption
  - Rollback
  - Kill Processes

Recovery/Preemption

- Take resource away from process
- may be possible with some resources
- side-effects?
- difficult to impossible
- manual intervention may be required

Rollback

- Assume deadlocks are likely
- set checkpoints
  - save state of processes, including
    - memory
    - registers
    - resources assigned
    - ...

Rollback (2)

- When deadlock occurs:
  - select process
  - set back to a checkpoint when he did not have a resource assigned that is involved in the deadlock

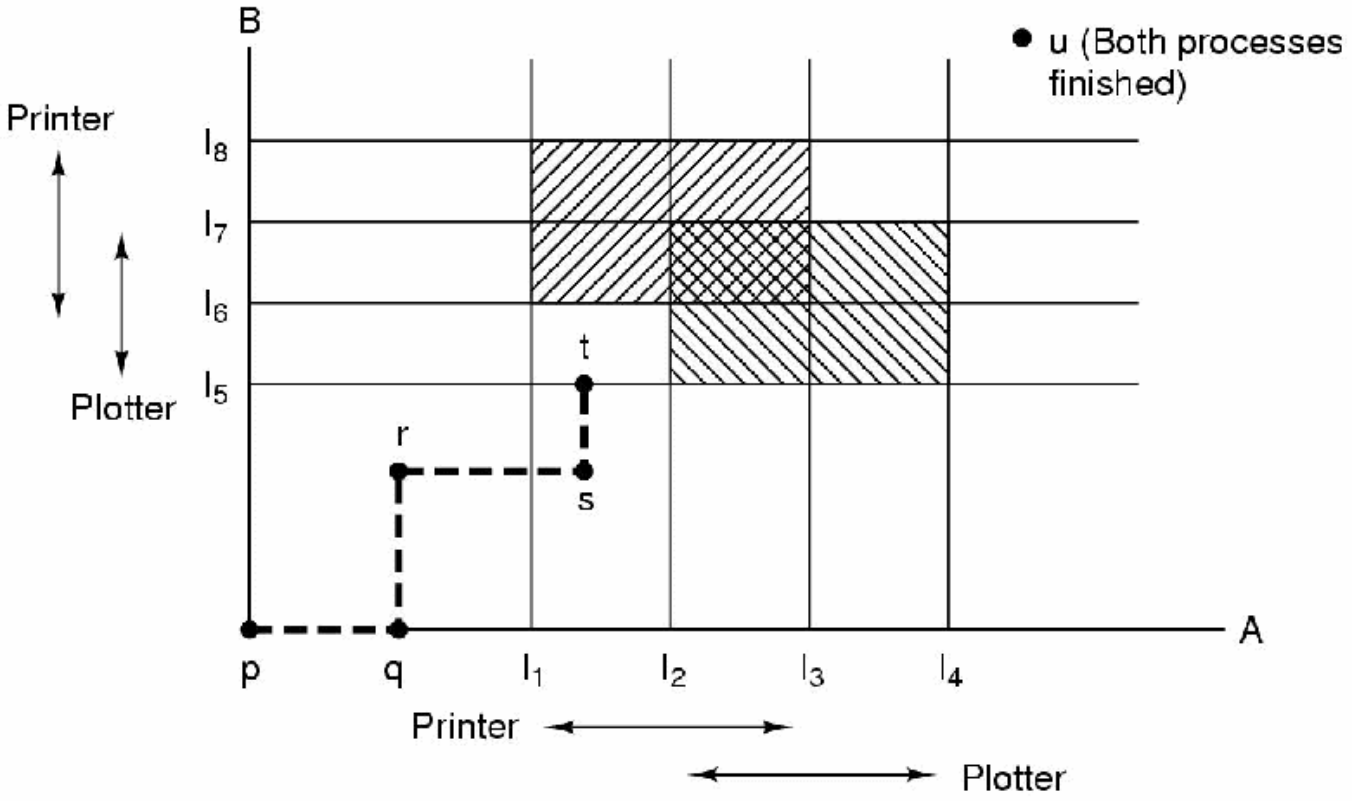
Kill Processes

- Simple and effective
- just kill a process involved in deadlock
- if this resolves deadlock: fine
- if not: kill one more
- best to kill one which can easily start again (like a compiler)
- killing processes that e.g. changed databases not a great idea

Avoidance

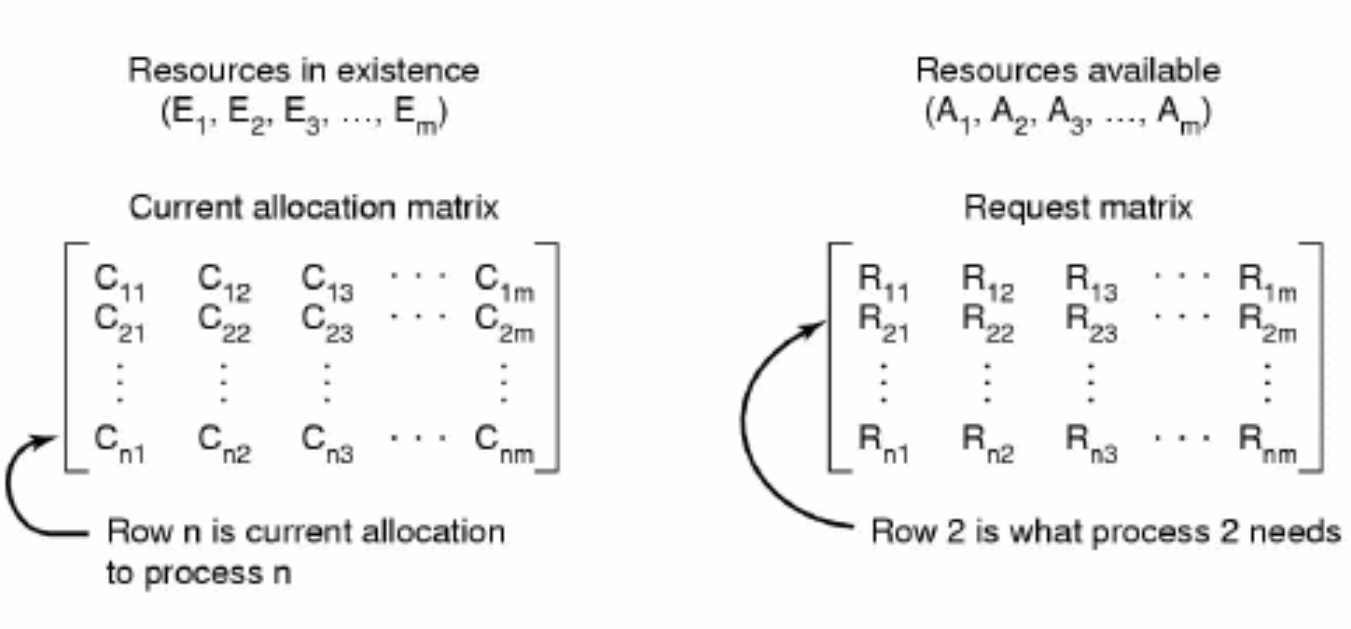
- Processes ask for resources on at a time
- avoidance would check if resource can be assigned safely and assign resource only when safe
- is there an algorithm that can do this?
- Yes, if certain information is available in advance

Resource Trajectories



Safe state

- We have to know some things about the resources and the processes:



Safe State

- state consists of
  - E: existing resources
  - A: available resources
  - C: current allocation
  - R: max. resources needed
- A state is safe if there is some scheduling order in which every process can run to completion even if all of them request their maximum

Safe state example

We have three processes ABC which need 4,5 and 5 pages at most

	Has	Max
A	0	4
B	0	5
C	0	5

We have 8 pageframes available and they aquire frames in the following order:
ABCABCABC

Safe state example

Frames requested: A

	Has	Max
A	1	4
B	0	5
C	0	5

Frames free: 7

Safe state example

Frames requested: AB

	Has	Max
A	1	4
B	1	5
C	0	5

Frames free: 6

Safe state example

Frames requested: ABC

	Has	Max
A	1	4
B	1	5
C	1	5

Frames free: 5

Safe state example

Frames requested: ABCA

	Has	Max
A	2	4
B	1	5
C	1	5

Frames free: 4

Safe state example

Frames requested: ABCAB

	Has	Max
A	2	4
B	2	5
C	1	5

Frames free: 3

Safe state example

Frames requested: ABCABC

	Has	Max
A	2	4
B	2	5
C	2	5

Frames free: 2

Safe state example

Frames requested: ABCABCA

	Has	Max
A	3	4
B	2	5
C	2	5

Frames free: 1

Safe state example

Frames requested: ABCABCAB

	Has	Max
A	3	4
B	3	5
C	2	5

Frames free: 0

Safe state example

Frames requested: ABCABCAB

	Has	Max
A	3	4
B	3	5
C	2	5

Frames free: 0 - DEADLOCK!

Safe state example

Lets go back....

Frames requested: ABCABCA

	Has	Max
A	3	4
B	2	5
C	2	5

Frames free: 1

is this state safe?

Safe state example

Frames requested: ABCABCA

	Has	Max
A	3	4
B	2	5
C	2	5

Frames free: 1

is this state safe? - it is safe *iff* we can find a sequence where all processes terminate.

Safe state example

Frames requested: ABCABCAA

	Has	Max
A	4	4
B	2	5
C	2	5

Frames free: 0

A gets “last” frame and terminates

Safe state example

Frames requested: ABCABCAA

	Has	Max
B	2	5
C	2	5

Frames free: 1

Now B can get all he needs and terminate

Safe state example

Frames requested: ABCABCAABBB

	Has	Max
B	5	5
C	2	5

Frames free: 1

Now B can get all he needs and terminate

Safe state example

Frames requested: ABCABCAABBB

	Has	Max
C	2	5

Frames free: 6

Now C can get all he needs and terminate

Safe state example

Frames requested: ABCABCA

	Has	Max
A	3	4
B	2	5
C	2	5

Frames free: 1

So this state is safe. If B requests a frame, we check whether the state would be safe when granting the request.

Safe state example

Frames requested: ABCABCA

	Has	Max
A	3	4´
B	3	5
C	2	5

simulated

Frames free: 0

We will find: this state is not safe. We will not grant the request.

Safe state example

Frames requested: ABCABCA

	Has	Max
A	3	4
B	2	5
C	2	5

blocked

Frames free: 1

B is blocked. Similar check for request by C

Safe state example

	Has	Max	
A	3	4	
B	2	5	blocked
C	2	5	blocked

- ∃ sequence that guarantees completion: safe state
- unsafe state: no deadlock (and <may> never happen)
- Bankers-algorithm (Dijkstra 1965)

Bankers algorithm

```
1  class ResourceMgr {
2          private:
3                  Lock: lock;
4                  CV: cv;
5                  int r;           // number of resources
6                  int t;           // number of threads
7                  int avail[];     // number if instances of
                       resources available
8                  int max[][];     // max of resources needed
                       by threat
9                  int alloc[][];   // # of resources allocated
                        by threat
10 }
```

Bankers algorithm

```
1  ResourceMgr::Request(int resouceID, int threadID) {
2          lock.Acquire();
3          assert(isSafe());
4          while ( ! wouldBeSafe(resourceID,threadID) ){
5                  cv.Wait(&lock);
6          }
7          alloc[resourceID][threadID]++;
8          avail[resourceID]--;
9          assert(isSafe());
10         lock.Release();
11 }
```

Bankers algorithm

```
1  ResourceMgr::wouldBeSafe(int resouceID, int threadID) {
2          bool result = false;
3          alloc[resourceID][threadID]++;
4          avail[resourceID]--;
5          if (isSafe()) {
6                  result=true;
7          }
8          alloc[resourceID][threadID]--;
9          avail[resourceID]++;
10 }
```

Bankers algorithm

```
1  ResourceMgr::isSafe() {
2          int j;
3          int toBeAvail[] = copy avail[];
4          int need[][] = max[][] - alloc[][];
5          bool finish[]=[false,false,false, ...];
6          while(true) {
```

Bankers algorithm

```
1         while(true) {
2               j = any thread that finished[j]=false && forall
                    i: need[i][j]<=toBeAvail[i];
3               if (no j exists) {
4                     if(forall j: finished[j]==true) {
5                           return true;
6                     } else {
7                           return false;
8                     }
9               } else {
10                    finish[j]=true;
11                    forall i: toBeAvail[i] += alloc[i][j];
12              }
13        }
14  }
```

Deadlock Immunity

- Ideas from EPFL 2008
- a property by which programs, once afflicted by a given deadlock, develop resistance against future occurrences of similar deadlocks
- Dimmunix Tool
  - automatically captures deadlock signatures (stacks of threads involved in deadlock)
  - subsequently avoids entering the same pattern.
  - Tested for e.g. Java (monitors and locks) and POSIX threads (using mutex)

Prevention

- Avoidance rarely practical
- Recovery after detection difficult
- What can we do?
- Prevent - by excluding one of the requirements

Mutual Exclusion

- no mutual exclusion - no deadlock
- since mutual exclusion is a requirement, this is practically impossible
- guideline:
  - avoid assigning a resource unless absolutely necessary
  - try to make sure that as few processes as possible may actually claim the resource

Hold and Wait

- Prevent processes that hold resources form waiting for more resources
- E.g. Require all processes to request all resources before starting execution
- Drawbacks:
  - processes don’t necessarily know that
  - very defensive tactics - and very very bad for effective resource utilization

Hold-and-wait

- Alternative:
  - release all resources first whenever acquiring a new one
  - then try to get all of them again

No preemption

- Very difficult. Rarely possible.

Circular Wait

- Easy way: allow only one resource to be held. Not very practical though.
- Better way: Provide global numbering of all resources.
- Processes can request resources, but only in numerical order.
- No cycle can exist.
- Difficult to find a satisfactory numbering scheme. What to do if resources are dynamic.

Two-Phase-Locking

- Avoidance and Prevention not very promising in the general case
- for specific applications excellent algorithms are known
- one example: database systems
  - frequently need locks on several records
  - then update all of them
- multiple processes: real danger of a deadlock

Two-Phase-Locking

- Phase 1: try to get locks for all records
- successful:
  - Phase 2:update records and release locks
- unsuccessful:
  - release locks and start again with Phase 1

Starvation

- Closely related to deadlocks
- policies decide who gets which resource when
- may lead to the situation that some process never gets service even if they are not deadlocked
- can e.g. be avoided by a first-come-first-served basis