# IN1805 I – Operating System Concepten

Hoofdstuk 7: Deadlocks

# Deadlocks (1)

- Definitie: Een groep processen verkeert in een deadlock toestand als ieder proces wacht op een event dat alleen veroorzaakt kan worden door een ander proces in die groep.
- Deadlocks kunnen makkelijk optreden bij het aanvragen van resources
- resourcetoekenning moet daarom voorzichtig gebeuren.
- Maar: te voorzichtig is ook niet goed, resources zijn dan onderbezet.

# Deadlocks (2)

Model

- Resources ingedeeld in typen
- Er kunnen meerdere *instances* van een type zijn
- Processen moeten resources aanvragen voor zij ze kunnen gebruiken, en vrijgeven als zij klaar zijn.
- Een proces vraagt in de regel een willekeurige instance van een bepaald type
- Als geen instance van het gevraagde type aanwezig is, moet een proces wachten (blocked)

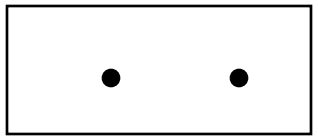
# Deadlocks (3)

Noodzakelijke voorwaarden voor een Deadlock:

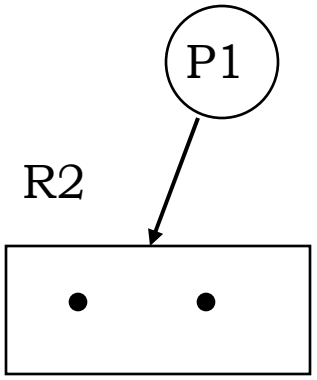
1. Wederzijdse uitsluiting
2. Hold en Wait
3. Geen Preemption
4. Circulaire Wait

Weergave d.m.v. resource-allocation graphs

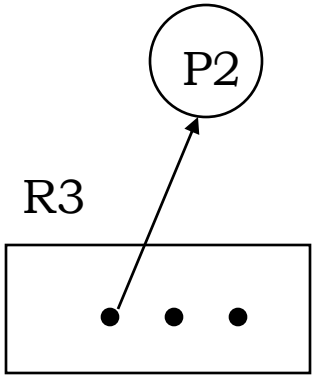
# Deadlocks (3A)
# resource allocation graph (1)




*Resource R1 met 2 instances*

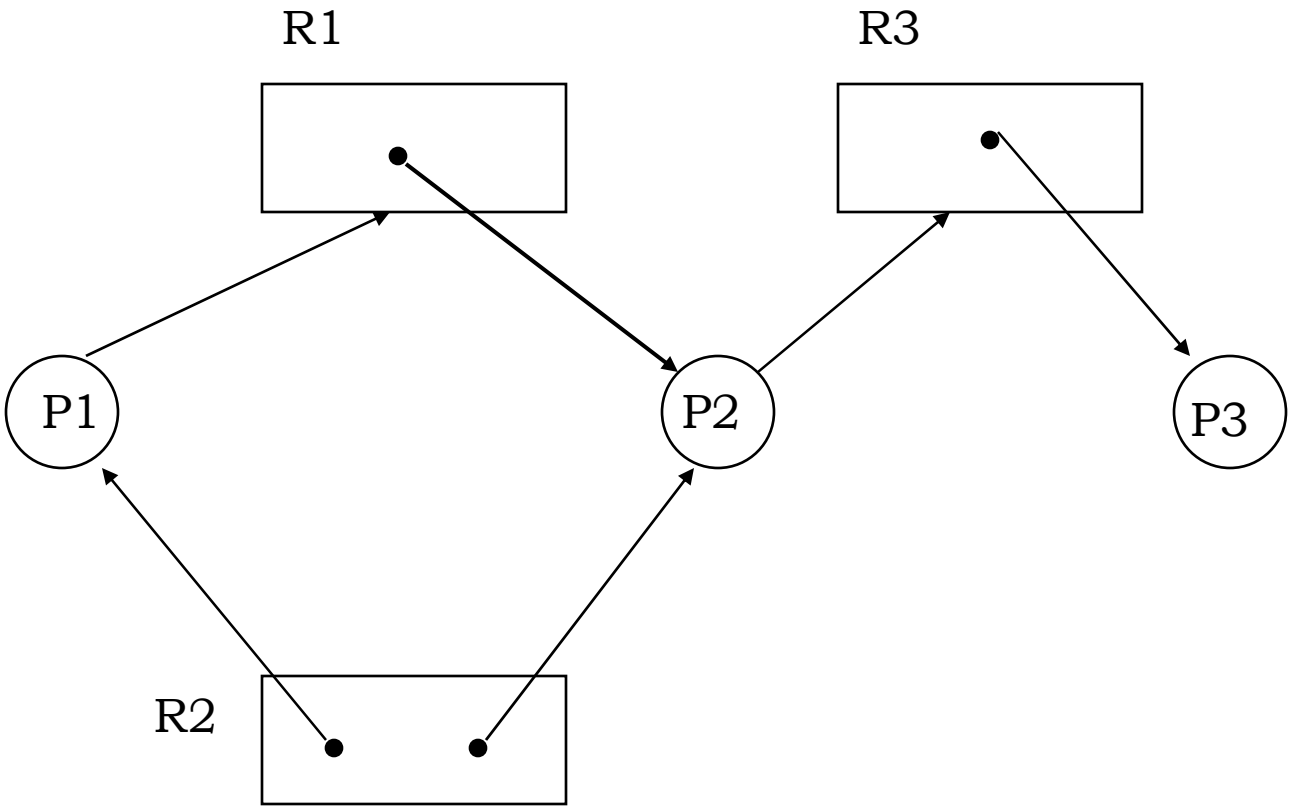



*P1 vraagt een instance van R2*



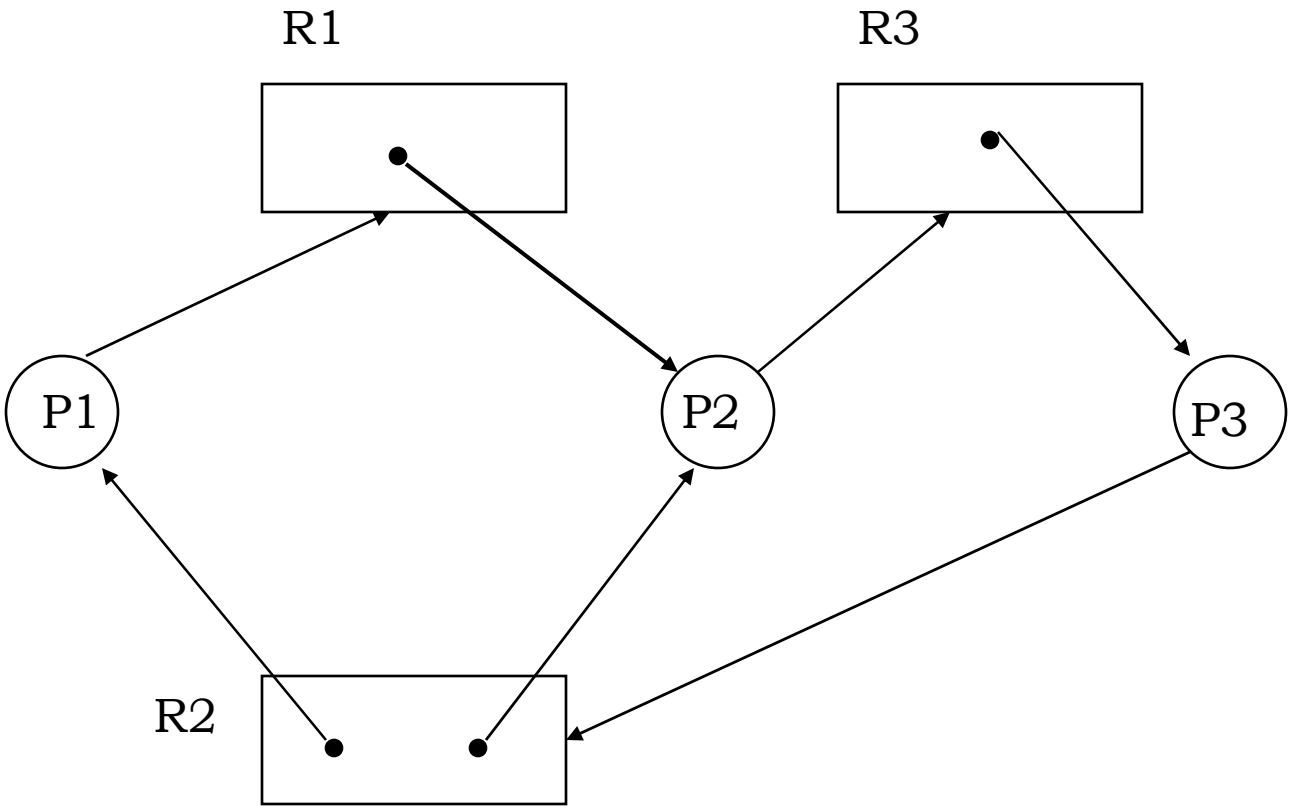

P2 heeft een instance van R3 toegewezen gekregen

# Deadlocks (3B): resource allocation graph(2)



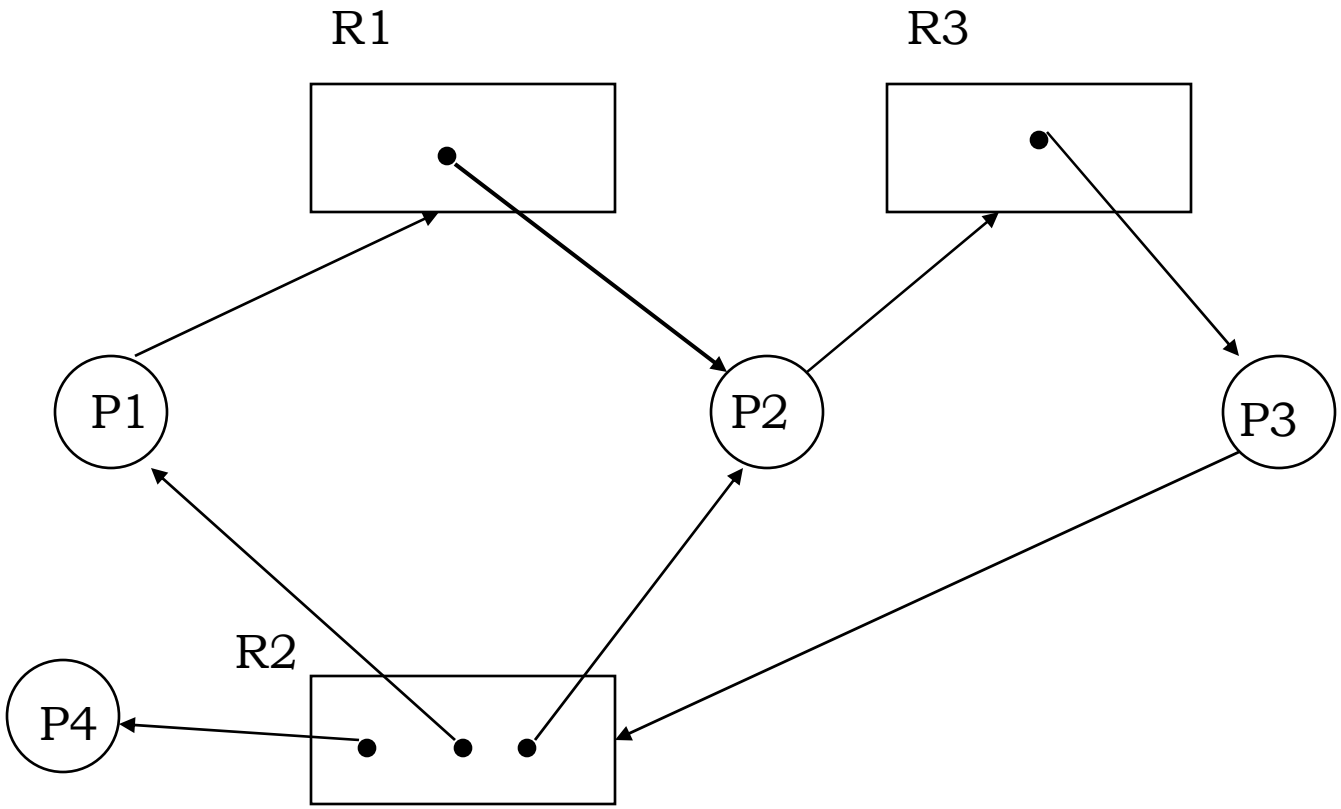

*geen deadlock, want geen cycle*

# Deadlocks (3C): resource allocation graph(3)




*wel deadlock, want cycles kunnen niet beëindigd worden*

# Deadlocks (3D): resource allocation graph(4)




*geen deadlock, want cycles kunnen beëindigd worden*

# Deadlocks (4)

Mogelijke strategie m.b.t. deadlocks

1. Zorg dat ze niet optreden
   - preventie (prevention)
   - ontwijking (avoidance)
2. Laat deadlocks toe,  onderneem actie wanneer ze optreden.
3. Doe alsof deadlocks niet bestaan.

# Deadlocks
# preventie (1)

Preventie: Elimineer 1 van de 4 noodzakelijke voorwaarden.

1. Wederzijdse uitsluiting wegnemen

   door zoveel mogelijk resources sharable te maken

2. Hold en Wait onmogelijk maken

   a. Een proces moet alle resources die het nodig heeft in 1 keer aanvragen (pre-allocation)

   b. Een proces mag alleen resources vragen als hij niets meer heeft

   nadelen van a en b:

   – lage resource benutting
   – mogelijkheid van starvation

# Deadlocks
# preventie (2)

preventie (vervolg)

3. No-preemption ongedaan maken

   door resources van wachtende processen af te pakken.

   (kan alleen als de resources daarvoor geschikt zijn , b.v. memory)

4. Circulaire Wait onmogelijk maken

   Door resource typen te nummeren

   Een proces mag alleen resourcetypen aanvragen met

   een nummer dat hoger is dan hij al heeft.

# deadlocks
# ontwijken (1)

Ontwijking

- Condities voor een deadlock zijn niet bij voorbaat uitgeschakeld.
- Door voorkennis m.b.t de door de processen te vragen resources (b.v. maximum aantal per type), kan een systeem deadlock ontwijken.

# Deadlocks
# Ontwijken (2)

- Een toestand is *veilig (safe)* als er een *veilige volgorde (safe sequence)* bestaat.
- Een veilige volgorde van processen is een reeks van processen waarvoor het volgende geldt:
  - Het eerste proces kan gegarandeerd eindigen, zelfs als hij zijn maximale aantal resources van ieder type vraagt.
  - Proces $P_j$ kan zeker eindigen als alle processen $P_i$ met $i < j$ geeindigd zijn (en hun resources hebben vrijgegeven)
- Basisregel voor het ontwijken van deadlock: *Toekenning van een resource mag alleen als daardoor de toestand veilig blijft.*

# Oefening

- Gegeven een resource met 12 instances en 3 processen die in de volgende toestand verkeren:

| | Max | Current |
|---|---|---|
| P1 | 10 | 5 |
| P2 | 4 | 2 |
| P3 | 9 | 2 |

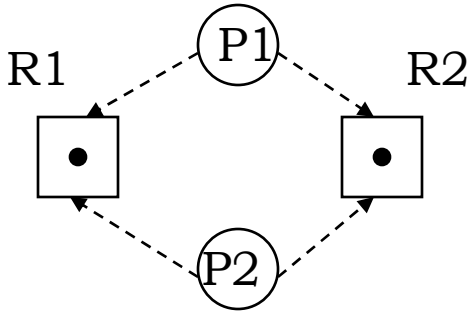
- Is dit een veilige (safe) toestand?

# deadlocks
# ontwijken (3)

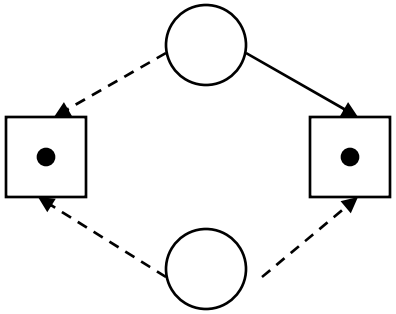
a. resource allocation graph algorithm

- voorwaarde: van ieder resource type is maar één instance aanwezig.
- een proces moet vooraf aangeven welke resources hij nodig zal hebben (*claim)*
- *claim*-takken toevoegen aan resource-allocation graph
- bij aanvraag voor resource , claim-tak wijzigen in request-tak, bij toekenning in assignment -tak
- als hierdoor een gesloten kring (*cycle*) zou ontstaan, de aanvraag **niet** toewijzen.
- cycle-detectie algoritme nodig. Complexiteit: $O(n^2)$

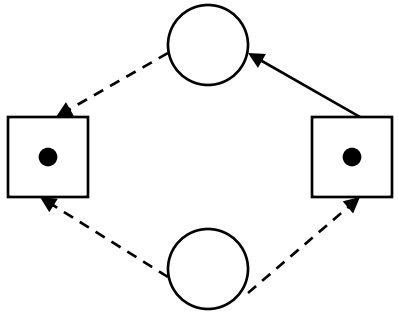
# deadlocks ontwijken (4)



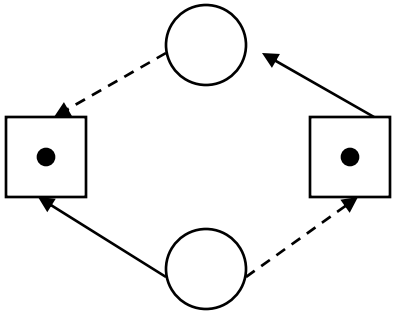

a. P1 en P2 claimen beide R1 en R2 (vooraf)



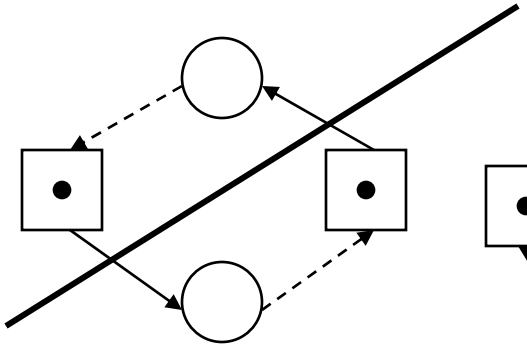
b. P1 vraagt R2



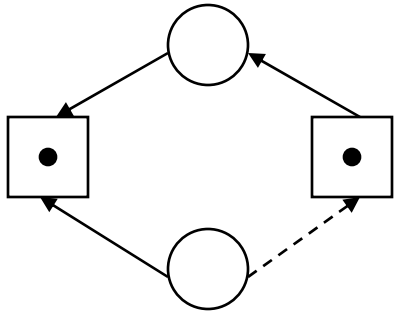
c. P1 krijgt R2, mag want er ontstaat geen circulair pad



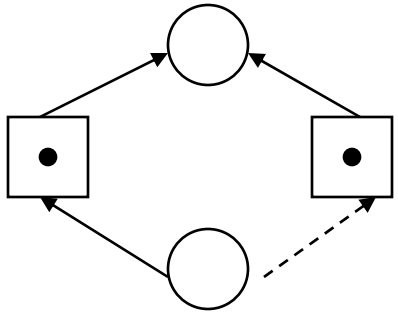
d. P2 vraagt R1



e. P2 krijgt R1 niet, want circulair pad



f. P1 vraagt R1



g.P1 krijgt R1, want geen circulair pad

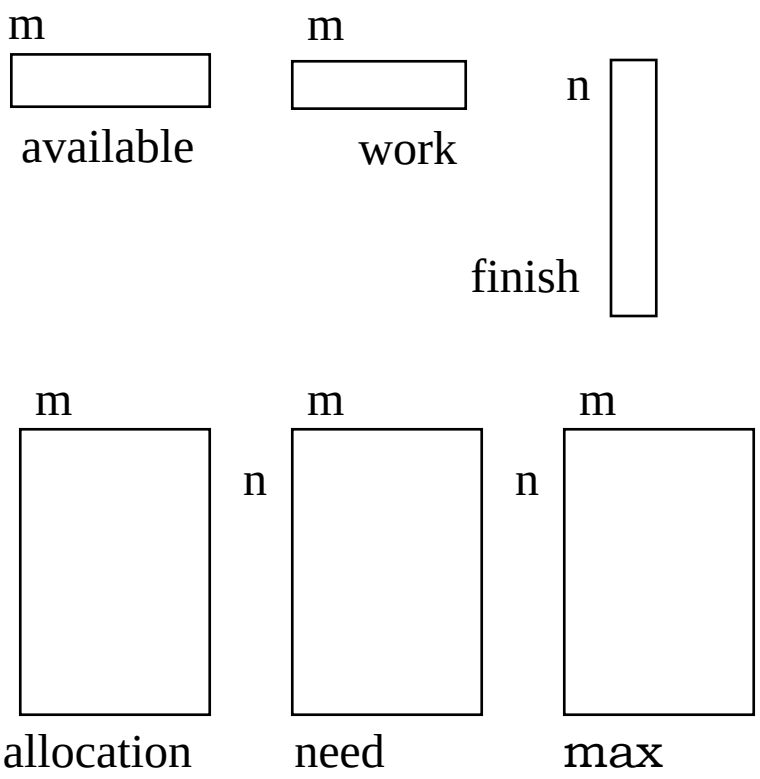
PAUZE

# deadlocks
# ontwijken (4)

b. Bankiers algoritme
   - ook geschikt als van resourcetypen meerdere instances bestaan
   - proces geeft van te voren op hoeveel instances van ieder type hij maximaal nodig zal hebben (*claim)*
   - als aanvraag binnen de claim past en het gevraagde aantal instances is aanwezig,
     - doe alsof gevraagde resources worden toegewezen
     - kijk of de dan ontstane situatie nog veilig is (d.w.z. is er op basis van de **claims** nog een veilige volgorde)

# Deadlocks ontwijken (5)

m = aantal resourcetypen
n = aantal processen

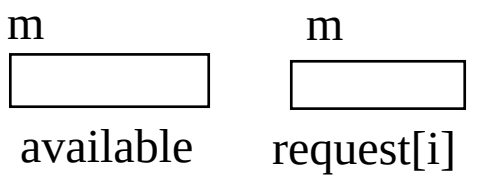


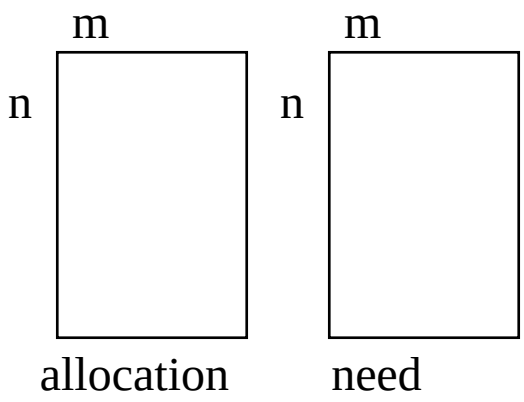

**Safety check**

1. initieel : work =available
   finish = false voor alle i
2. zoek i, zodat finish[i] = false
   en need $_i$ <= work
   als deze niet bestaat goto 4
3. work := work + allocation $_i$
   finish[i] := true
   goto 2
4. als finish[i]= true voor alle i
   toestand safe , anders toestand unsafe

# Deadlocks ontwijken (5)

m = aantal resourcetypen
n = aantal processen




**Resource-request**

1. als request[i] > need[i]
   geef foutmelding
2. als request[i] > available
   WAIT
3. pretend allocation
   available := available – request[i]
   allocation[i] := allocation[i] + request[i]
   need[i] := need[i] – request[i]
4. als unsafe state
   maak allocatie ongedaan en WAIT

# Example of Banker's Algorithm

- 5 processes $P_0$ through $P_4$;
  3 resource types: $A$ (10 instances), $B$ (5), and $C$ (7).
- Snapshot at time $T_0$:

| | *Allocation* | *Max* | *Need* | *Available* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | 1 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

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

# Exercise:

- Can we honor the following requests without running the risk of ending up in a deadlock?
  - P1 Request (1,0,2)
  - P4 Request (3,3,0)
  - P0 Request (0,2,0)

# deadlocks
# detectie (1)

a. Door cycle in de Wait-for-Graph (WFG)
   (als voor ieder resource type één instance)

b. Als meerdere instances van resource typen mogelijk zijn:
   deadlockvrije volgorde verifiëren op basis van Bankiers algoritme, maar dan op grond van bestaande **requests** ( i.p.v. op claims)

# deadlocks
# detectie (2)

- wanneer deadlock detectie algoritme aanroepen?

  afhankelijk van

  – frequentie van optreden van deadlocks
  – aantal betrokken processen
- in principe kan deadlock optreden iedere keer wanneer een request niet onmiddellijk kan worden gehonoreeerd. In al deze gevallen meteen testen niet verstandig i.v.m. overhead.

# deadlocks
# recovery

Op 2 wijzen uit te voeren:

- handmatig (operator of gebruiker)
- automatisch

mogelijke acties:

- proces termination
  - probleem: welk proces (of processen) ?
- resource preemption:
  - problemen:
    - keuze van slachtoffer
    - roll-back (hoe ver ?)
    - starvation