

# System with Single-Unit resources

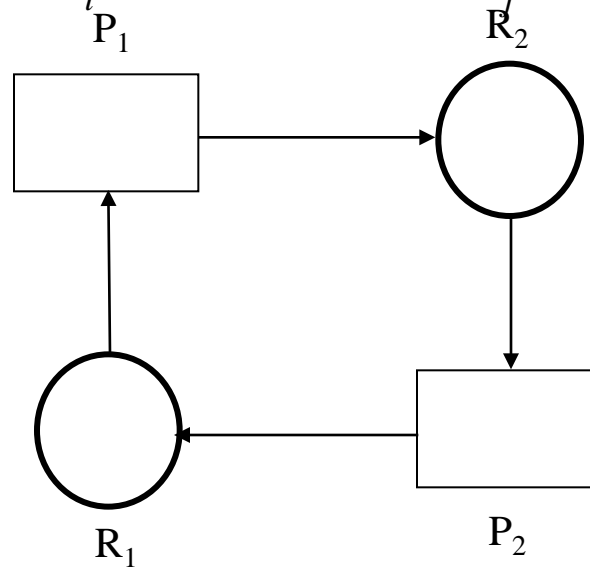
- Th. 3.6: “If there is only a single unit of every resource, then a *cycle* in an expedient resource graph is a **necessary and sufficient** condition for a deadlock”
- In this particular case, deadlock presence is equivalent to a graph feature - a cycle
  - cycle search complexity is  $O(n^2)$  for a  $n$ -node graph

# Systems with single unit requests

- The General Resource Graph of such systems can have only a single outgoing edge from a process node
- Th. 3.5: “An expedient grg represents a deadlock if and only if it contains a knot”

# Systems with only consumable resources

- Definition: a *claim-limited graph* is the grg of a particular state, in which:
  - each resource has zero available units
  - there is a request edge  $(P_i, R_j)$  iff  $P_i$  is a consumer of  $R_j$
- Example:



# Claim-limited graph

- Used to characterize the system, not just a single state of the system
- Worst case situation of a system
  - all resources exhausted
  - every consumer is requesting one unit of all the resources it could possibly request
- Claim-limited graph is a conservative analysis

# Systems with only consumable resources (2)

- Th. 3.4: “A consumable resource only system is deadlock-free if its claim-limited graph is completely reducible”
- Practical consequence:
  - deadlock absence is equivalent to a certain graph property:  
claim-limited graph is CR  $\Rightarrow$  state is deadlock free
  - cannot say: “if graph not CR then system is not deadlock free”!

# Table of deadlock theorems

	<b>Deadlock freedom conditions</b>	<b>Deadlock presence conditions</b>
<b>General case</b>	$CR \Rightarrow DLF$	$DL \Rightarrow Cy$ $Kn, Exp \Rightarrow DL$
<b>Reusable resources</b>	$CR \Leftrightarrow DLF$	
<b>Reusable + single unit resources</b>		$Cy, Exp \Leftrightarrow DL$
<b>Single unit requests</b>		$Kn, Exp \Leftrightarrow DL$
<b>Consumable resources</b>	Claim-limited Graph $CR \Rightarrow DLF$	

Note: CR = "GRG is Completely Reducible" -- DLF = "State is Deadlock Free" -- DL = "State is deadlocked"  
 Cy = "GRG contains a Cycle" -- Kn = "GRG contains a Knot" -- Exp = "GRG is Expedient"

# How to use the graph reduction method

- The GRG and the reduction method can be used for the automatic detection of deadlock
  - More or less powerful depending on the features of the system => available theorem
- Other strategies can be used depending on the design constraints
  - Deadlock prevention or avoidance have different tradeoffs than the deadlock detection approach

# Pros and Cons of different strategies

- **Deadlock detection  $\Rightarrow$  graph reduction method**
  - Pros: maximum concurrency and resource utilization
  - Cons: roll-back, cycle checking overhead even if not deadlocked
- **Deadlock prevention  $\Rightarrow$  four necessary conditions**
  - Pros: suitable where roll-back is impossible or expensive
  - Cons: inefficient due to preemption or low resource utilization
- **Deadlock avoidance  $\Rightarrow$  banker's algorithm**
  - Pros: no roll-back of processes required
  - Cons: safe state checking overhead, safe state is overly conservative estimate, future resource requirements must be known a priori

# Banker's algorithm

- Definitions for a system with  $n$  processors and  $m$  resources:

$$\text{Max - Avail matrix } A = (a_1 \quad a_2 \quad \dots \quad a_m)$$

$$\text{Current Avail matrix } D = (d_1 \quad d_2 \quad \dots \quad d_m) = A - \sum_{k=1}^n C_k$$

$$\text{Max - Claim matrix } B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix}$$

$$\text{Current Need matrix } E = \begin{pmatrix} e_{11} & e_{12} & \dots & e_{1m} \\ e_{21} & e_{22} & \dots & e_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ e_{n1} & e_{n2} & \dots & e_{nm} \end{pmatrix} = B - C = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

$$\text{Current Allocation matrix } C = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nm} \end{pmatrix} = \begin{pmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{pmatrix}$$

$$\text{Request vector } F_i = (f_{i1} \quad f_{i2} \quad \dots \quad f_{im})$$

# Algorithm steps

- Step 1 - tentative accept of request
  - $D := D - F$  // update Current Avail matrix  $D$
  - $C_i := C_i + F_i$  // update Current Alloc vector  $C_i$
  - $E_i := E_i - F_i$  // update Current Need matrix  $E_i$
- Step 2 - safe-state checking test
  - see next slide in which
    - freemoney =  $D$
    - loan[i] =  $C_i$
    - need[i] =  $E_i$
- Step 3 - if test positive definitive accept of request, otherwise roll back the updates of step 1

# Banker at work: safe-state checking

```
While (last_iteration_successful)
  last_iteration_successful = false ;
  for i = 1 to N do
    if (finishdoubtful[i] AND need[i] ≤ freemoney) then // need = claim - loan, how
                                                         // much process i still needs

      finishdoubtful[i] = false ;                       // process can finish because
      last_iteration_successful = true                   // need ≤ free resources

      free money = free money + loan[i] ;               // process is able to finish thus
                                                         // it will repay the loan back

    end if
  end for
End while

if (free money == capital) then safe!
  else not safe! ;
```

# Banker's algorithm example (1)

- System with 3 processors and 3 resources, and with the following matrices:

Max - Avail matrix  $A = (2 \quad 4 \quad 3)$

Max - Claim matrix  $B = \begin{pmatrix} 1 & 2 & 2 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$

Current Allocation matrix  $C = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$

Current Avail matrix  $D = (0 \quad 1 \quad 1) = A - \sum_{k=1}^n C_k$

Current Need matrix  $E = \begin{pmatrix} 0 & 0 & 2 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = B - C$

- Request  $F_1 = (0 \ 0 \ 1)$  should be granted?

# Banker's algorithm example (2)

- Suppose the request is granted, the resulting state would be the following:

$$\text{Current Allocation matrix } C = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

$$\text{Current Avail matrix } D = (0 \quad 1 \quad 0) = A - \sum_{k=1}^n C_k$$

$$\text{Current Need matrix } E = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix} = B - C$$

# Banker's algorithm example (3)

- The new state is safe
  - $P_3$  can complete (because  $E_3 \leq D$ ) and thus can return (1 0 1) to the pool of available resources
  - $D$  becomes (1 1 1); the outstanding needs of both  $P_1$  and  $P_2$  can be satisfied