Deadlock-02

Banker's Algorithm

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available***:* Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max***: n x m* matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation***:* n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need***:* n x m matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task

  Need [i,j] = *Max*[i,j] – *Allocation* [i,j]

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively. Initialize:

   Work = *Available*

   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1

2. Find an *i* such that both:

   (a) *Finish* [*i*] = *false*

   (b) $Need_i \leq Work$

   If no such *i* exists, go to step 4

3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish* [*i*] == true for all *i*, then the system is in a safe state

Example of Banker's Algorithm

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

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

Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Safety Algorithm Example

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3 (7,4,3 — 3,3,2)

Finish [0] is false and $Need_0$ > Work

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2 (1,2,2 — 3,3,2)

Finish [1] is false and $Need_1$ < Work

So P_1 must be kept in safe sequence

Step 3

Work = Work + $Allocation_1$ (3, 3, 2 — 2, 0, 0)

Work =	A	B	C
	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0 (6, 0, 0 — 5,3, 2)

Finish [2] is false and $Need_2$ > Work

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $Need_3$ < Work (0, 1, 1 — 5, 3, 2)

So P_3 must be kept in safe sequence

Step 3

Work = Work + $Allocation_3$ (5, 3, 2 — 2, 1, 1)

Work =	A	B	C
	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $Need_4$ < Work (4, 3, 1 — 7, 4, 3)

So P_4 must be kept in safe sequence

Step 3

Work = Work + $Allocation_4$ (7, 4, 3 — 0, 0, 2)

Work =	A	B	C
	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and Need < Work (7, 4, 3 — 7, 4, 5)

So P_0 must be kept in safe sequence

Step 3

Work = Work + $Allocation_0$ (7, 4, 5 — 0, 1 , 0)

Work =	A	B	C
	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $Need_2$ < Work (6, 0, 0 — 7, 5, 5)

So P_2 must be kept in safe sequence

Step 3

Work = Work + $Allocation_2$ (7, 5, 5 — 3, 0, 2)

Work =	A	B	C
	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \le i \le n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Resource-Request Algorithm for Process P_i

Request = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

- *If safe $\Rightarrow$ the resources are allocated to Pi*
- *If unsafe $\Rightarrow$ Pi must wait, and the old resource-allocation state is restored*

Example: P_1 Request (1,0,2)

A B C

$Request_1$ = 1, 0, 2

To decide whether the request is granted we use Resource Request algorithm

Step 1

1, 0, 2 < 1, 2, 2

$Request_1 < Need_1$

Step 2

1, 0, 2 < 3, 3, 2

$Request_1 < Available$

Step 3

Available = Available – $Request_1$

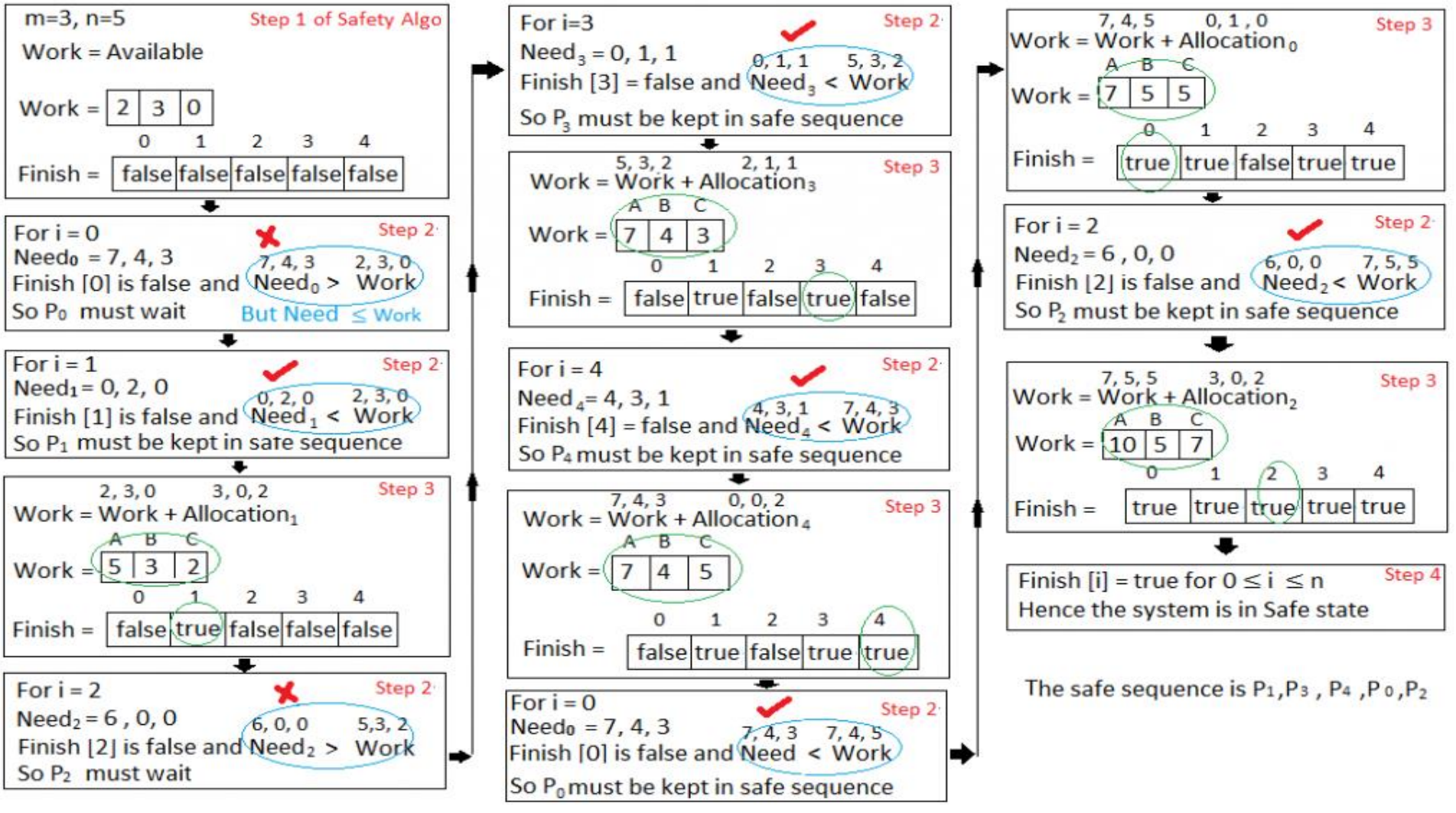
$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

Process	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Example: P_1 Request (1,0,2)(cont.)

- We must determine whether this new system state is safe. To do so, we again execute Safety algorithm on the above data structures.(Showed in next slide)



Hence the new system state is safe, so we can immediately grant the request for process $\mathbf{P_1}$.

Exercise

- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

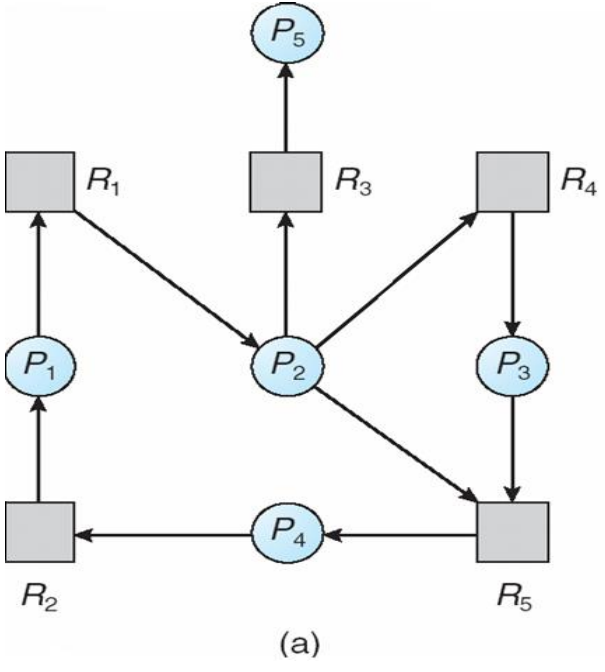
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

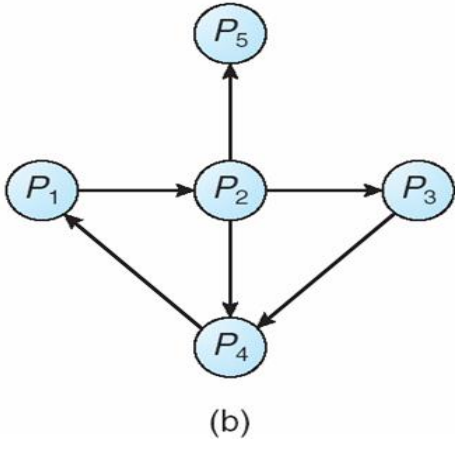
Single Instance of Each Resource Type

- Maintain *wait-for* graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- **Available***:* A vector of length *m* indicates the number of available resources of each type.
- **Allocation***:* An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
- **Request***:* An *n* x *m* matrix indicates the current request of each process. If *Request* $[i_j] = k$, then process P_i is requesting *k* more instances of resource type. R_j.

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, if $Allocation_i \neq 0$, then *Finish*[i] = false;otherwise, *Finish*[i] = *true*
2. Find an index *i* such that both:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$

   If no such *i* exists, go to step 4

Detection Algorithm (Cont.)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2
4. If *Finish*[*i*] == false, for some *i*, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Example of Detection Algorithm(Cont.)

1.In this, Work = [0, 0, 0] & Finish = [false, false, false, false, false]
2.i=0 is selected as both Finish[0] = false and [0, 0, 0]<=[0, 0, 0].
Work =[0, 0, 0]+[0, 1, 0] =>[0, 1, 0] & Finish = [true, false, false, false, false].
3.i=2 is selected as both Finish[2] = false and [0, 0, 0]<=[0, 1, 0].
Work =[0, 1, 0]+[3, 0, 3] =>[3, 1, 3] & Finish = [true, false, true, false, false].
4.i=1 is selected as both Finish[1] = false and [2, 0, 2]<=[3, 1, 3].
Work =[3, 1, 3]+[2, 0, 0] =>[5, 1, 3] & Finish = [true, true, true, false, false].
5.i=3 is selected as both Finish[3] = false and [1, 0, 0]<=[5, 1, 3].
Work =[5, 1, 3]+[2, 1, 1] =>[7, 2, 4] & Finish = [true, true, true, true, false].
6.i=4 is selected as both Finish[4] = false and [0, 0, 2]<=[7, 2, 4].
Work =[7, 2, 4]+[0, 0, 2] =>[7, 2, 6] & Finish = [true, true, true, true, true].
Sequence <P_0, P_2, P_3, P_1, P_4> will result in *Finish*[*i*] = true for all *i*
Since Finish is a vector of all true it means **there is no deadlock** in this example.

Example of Detection Algorithm(Cont.)

- P_2 requests an additional instance of type *C*

	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor

THANK YOU!