EE5903 RTS
Chapter 6
Real-Time Process Synchronization

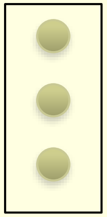
Bharadwaj Veeravalli

elebv@nus.edu.sg

Acknowledgments: Parts of the material are from the book, OS Concepts by Silberschatz, Galvin and Gagne, 9th Edition

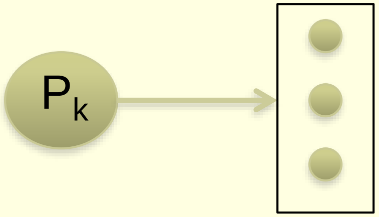
Deadlocks

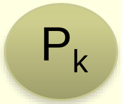
- Definition: A set of processes is said to be in deadlock when every process in the set waits for an event that can only be caused by another process in the set, which in turn is waiting for an event to occur!



Resource and its instances

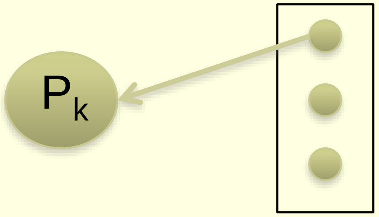
Process requests for a resource and waiting for that resource





Process k

An instance of a resource allocated to a process;

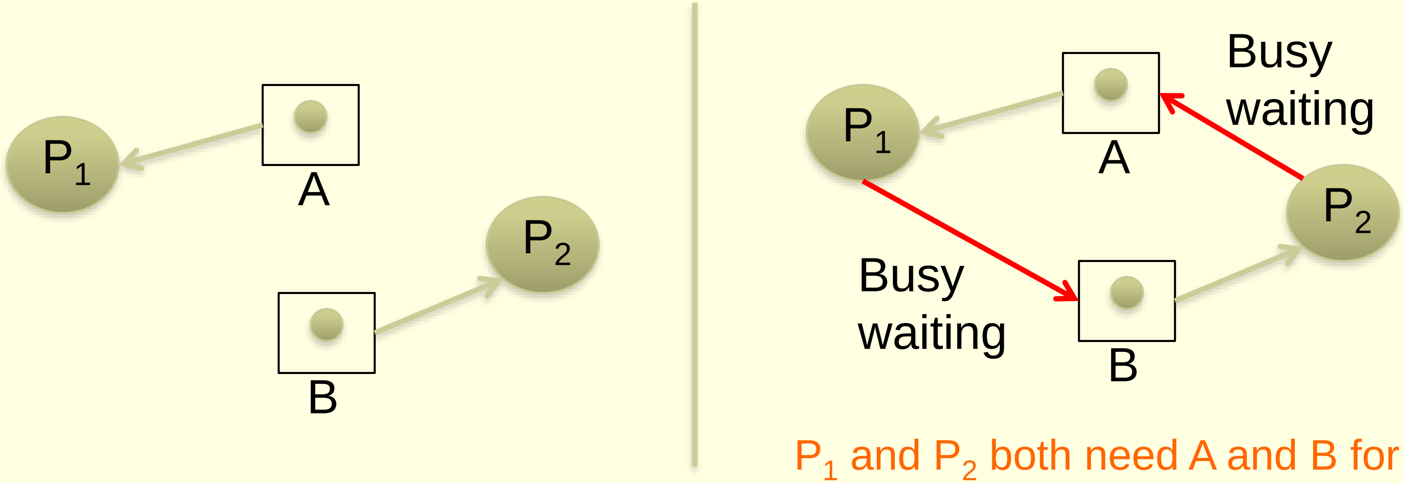


Deadlocks

- G = < V,E> - Graph with |V| vertices and |E| edges

V: Set of vertices (resources, processes);

E: Set of edges (requests, allocation)



P_1 and P_2 both need A and B for their execution;

OS must ensure such deadlocks do not occur; It should know the reasons under which such deadlocks would occur;

Deadlocks

- 4 Necessary Conditions for the occurrence of deadlocks [*Coffman (1971)*] - that must hold simultaneously for there to be a deadlock.

- Mutual Exclusion (The resources involved are non-shareable, that is only one process at a time can use the resource.)
- Hold & Wait (Process is holding some resource and it is waiting for other process to release a resource; needs to be avoided;)
- No preemption (Once allocated, a resource cannot be preempted; The process has to voluntarily release it!)
- Circular wait (Special case of #2)

Since these are necessary conditions, if we can avoid at least any of the conditions then we can ensure that the system is deadlock free.

Deadlocks – Prevention and Avoidance

- Prevention – *A system design concern!*

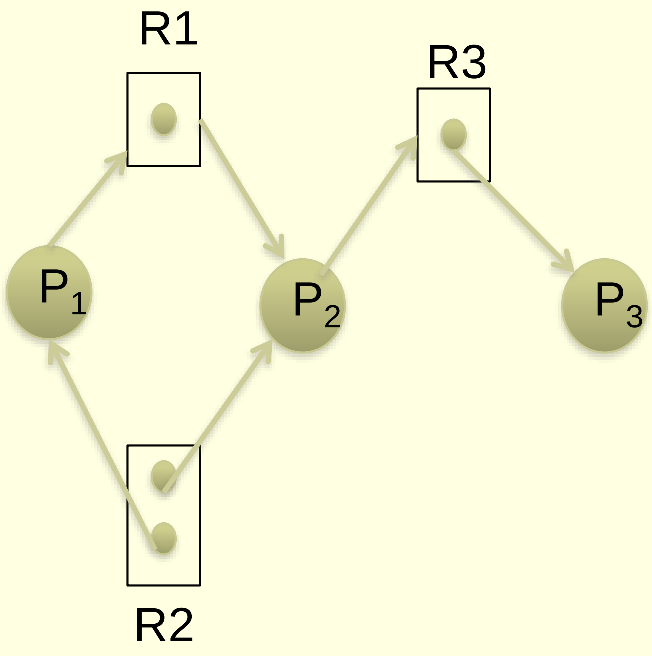
Key question - *How requests for resources can be made in the system and how they are handled?*

Objective is to ensure that at least one of the necessary conditions for deadlock can never hold.

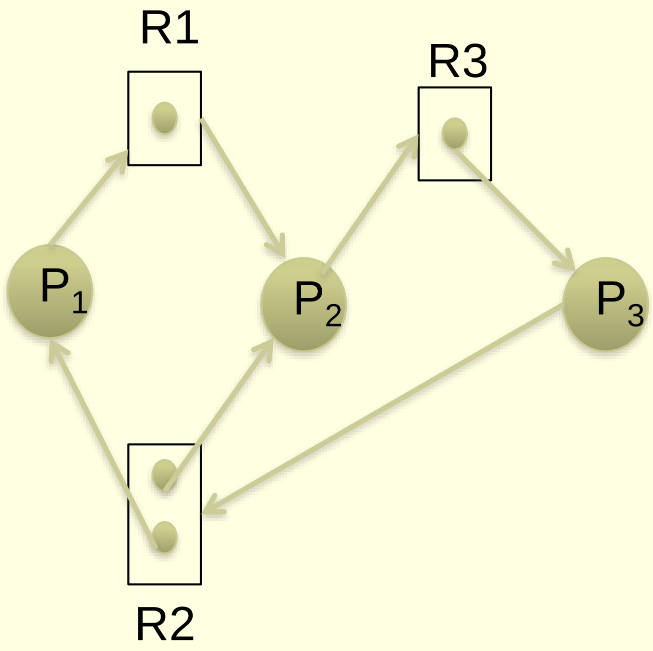
- **Avoidance** – We consider every request and decide whether it is safe to grant it at this point; Need more information from processes about their resource demands a priori! (*We can see this being used in Banker's algorithm later*)

Deadlocks – Resource Allocation Graphs

- Analyze the following situations (A) & (B):

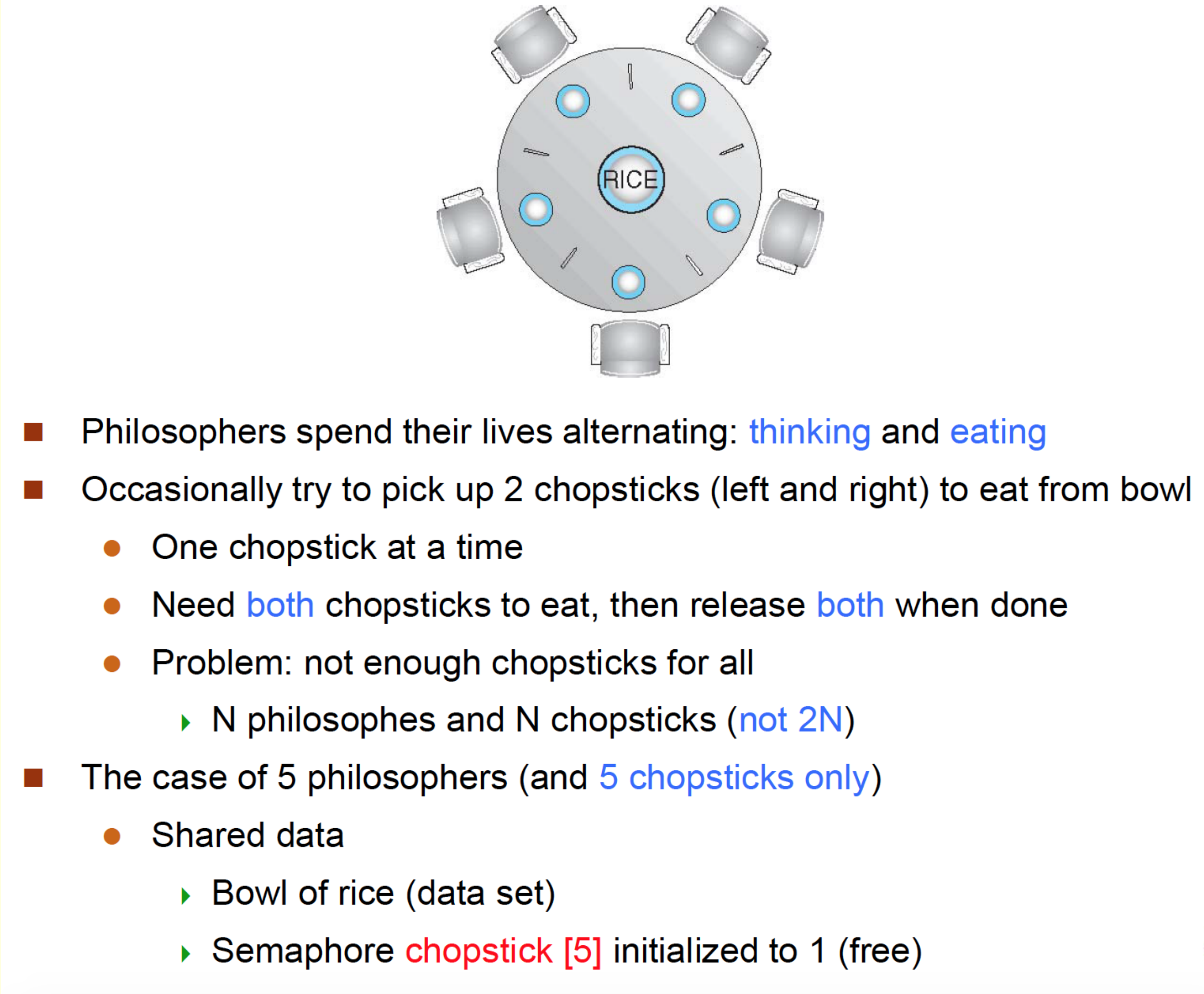


(A)



(B)

Dining Philosopher’s Problem



- Philosophers spend their lives alternating: thinking and eating
- Occasionally try to pick up 2 chopsticks (left and right) to eat from bowl
  - One chopstick at a time
  - Need both chopsticks to eat, then release both when done
  - Problem: not enough chopsticks for all
    - N philosophes and N chopsticks (not 2N)
- The case of 5 philosophers (and 5 chopsticks only)
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1 (free)

Pseudocode:
Given as a part of your solution set;

Follow the discussions in the lecture.

Deadlock Avoidance – Banker's Algorithm

Data structures, notations, and terminologies used:

n - # of processes; **m** - # of resource types (identical num of instances)

Available[i] = k; // **k** number of resource instances of **resource type i** are available; Dimension is **m** for this array;

Max[i][j] = q; // Matrix (**n x m**); **Proc i** needs a maximum of q number of instances of **resource type j**

Allocation[i][j] = p; // Matrix (**n x m**); p number of instances of resource j is allocated to Proc i

Need[i][j] = r; // Matrix (n x m); r num of instances of **Res type j** are needed for **Proc i** in the future; Can be deducted from the above two; This is the diff of Max and Allocation!

Deadlock Avoidance – Banker’s Algorithm

Algorithm execution: Whenever a request arrives this algorithm is executed and a request is presented in the form of a vector, as it involves different resource types;

Algorithm steps: **[Req_i** - Request by Proc I is a vector]

Note: X ≤ Y, means every component of X is ≤ Y.

```
(1) if Req_i ≤ Need[i] then Go To (2)
    else:  Error_msg;

(2) if Req_i ≤ Available[i] then Go To (3)
    else:  wait();
```

Now the system is in a state to meet the request; It has to decide on the allocation;

System pretends to execute the request (not actually committing to any allocation) and attempts to modify the data structures as follows.

Deadlock Avoidance – Banker’s Algorithm

(3) Available = Available - Req_i; // Available will be reduced by Req_i, if committed

Allocation[i] = Allocation[i] + Req_i; //Allocation gets modified by Req_i amount

Need[i] = Need[i] - Req_i; //So, Need has to be updated too

After temporarily modifying, we need to check if this is a “safe state” or not. That is, by doing this modification, we need to guarantee if all the processes can be successfully executed in some arbitrary sequence or not; We call such a state as ‘safe state’;

In order for a state to be safe, there must be a sequence of execution of processes in which they all are able to complete their execution but it is not necessary that this the best sequence; Such a sequence is called a ‘*safe sequence*’;

(4) Check if this new state is ‘Safe’, i.e., if a safe seq exists;

Call Safety_Check(Available) algorithm

Deadlock Avoidance – Banker’s Algorithm

Algorithm - Safety Check(Available):

```
(1) Initialization:  Work = Available;   Finish[i] = False, for all i;
```

Finish vector indicates if a process has finished execution or not; A False implies a process has not yet finished execution;

```
(2) Find a Proc P_i s.t: (Finish[i] == False && Need[i] <= Work)
    If no such Proc i exists, Go To (4)
    Else:
(3) Work = Work + Allocation[i]
    Finish[i] = True;
    Go To (2);

(4) If Finish[i] = True, for all i
        System = ‘Safe_mode’;
    Else: return;
```

Note that if such a process i exists then after it executes it releases all its resources and these get added to the Available list; Thus,

Conclusion:
If the state is Safe, then the physical allocation of resources is committed for that seq derived;

Deadlock Avoidance – Banker’s Algorithm

- Run Banker’s algorithm to determine a Safe Sequence for the following initial conditions given in the table.
- Resource instances: A – 10; B – 5; C – 7;
- Num of Processes: 5 (P0, P1,…,P4)

Max - Allocation

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3				7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2	10	5	7	6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

Start the sequence: P0 to P4

Final Solution:

Safety Seq
P1
P3
P4
P0
P2

Curr. Allocated: 7 2 5 (total of all individual components)

Solution

Available	Proc.	Need	Scheduled Proc
(3 3 2)	1	1 2 2	1
+(2 0 0) (P1 releases this!)			
(5 3 2)	3	0 1 1	3
+(2 1 1) (P3 releases this!)			
(7 4 3)	4	4 3 1	4
+(0 0 2) (P4 releases this!)			
(7 4 5)	0	7 4 3	0
+(0 1 0) (P4 releases this!)			
(7 5 5)	2	6 0 0	2

Generated Safe Sequence

Distributed Deadlock Detection & Recovery

- Distributed deadlock problem:
  - Distributed nodes each comprising different processes
  - Common/Shared resource pool;
  - Each process can independently place requests for any resource;

Two design approaches possible:

- Centralized Control
- Distributed Control (every node participates in resolving the deadlock)

Distributed Deadlock Detection & Recovery – Centralized Control Approach

- Processes are distributed over several machines

- Resources are also distributed; Each process does not know about the processes in other machines but it knows the resources available with other machines;

- Entire system **has a central coordinator (CC)** which maintains a global resource-allocation (GRA) graph;

- It is the job of every machine to update the status of resource requests and current resources that each of its processes owns to the CC. Then only CC can maintain the GRA graph;

Note: CC is also a process running in a particular machine; It is implemented in such a way that this CC process will never be a part of deadlock.

Distributed Deadlock Detection & Recovery – Centralized Control Approach

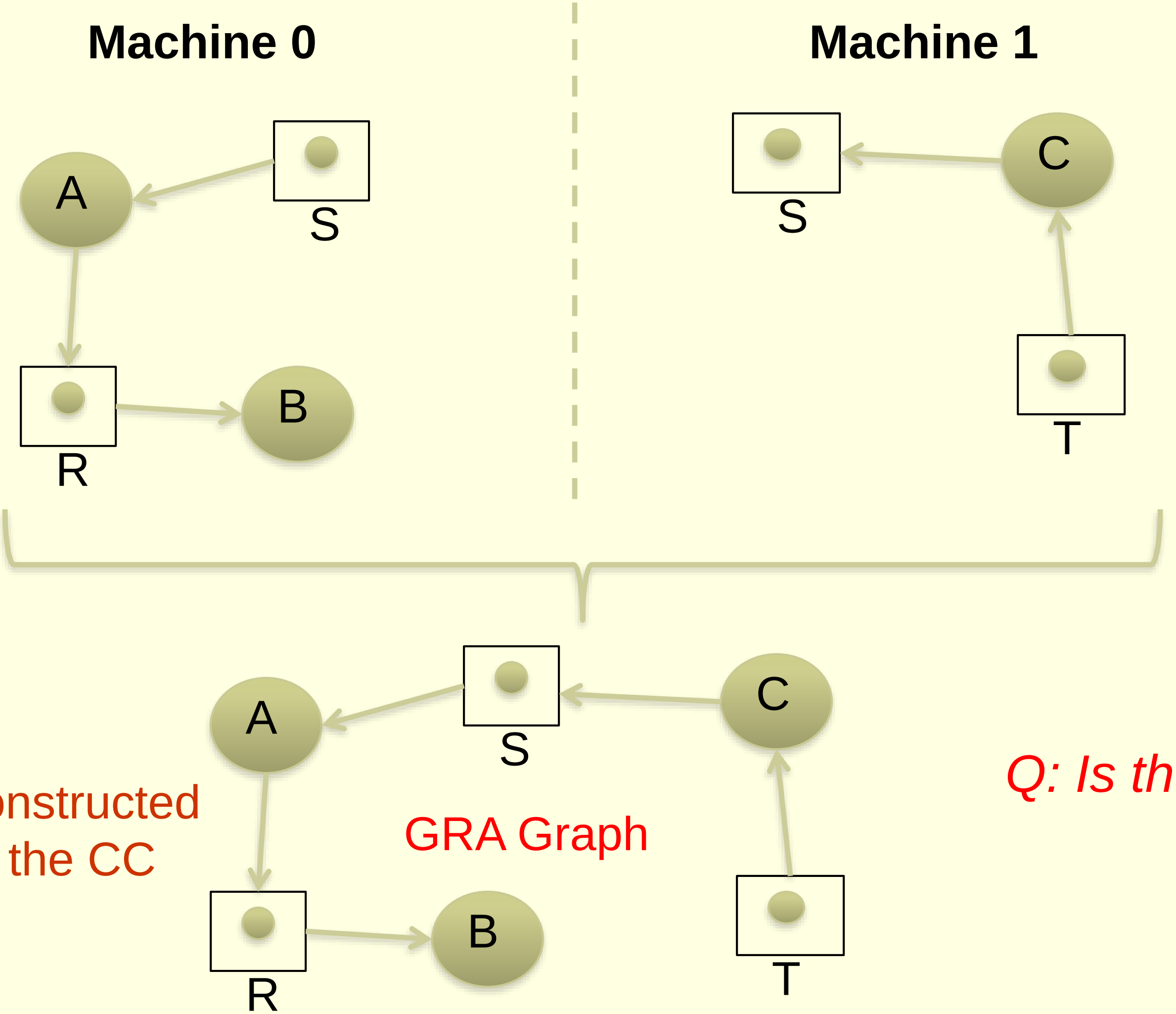
Updates to the GRA graph can be done in two ways:

(a) As soon as Possible (ASAP) Strategy: Whenever there is a change in a local resource allocation graph with a machine, it can update the CC and CC can reflect the change in the GRA graph;

(b) Periodic Update Strategy: Send only the updates, at pre-defined time epochs, since the previous updates. During this time if, say, a few edges are deleted in the local RA graph, then all these changes will be sent to CC and it can update in one shot;

Then the deadlock detection has to be done by the CC; Thus, when it finds a deadlock it can decide to kill one or more processes so that system can come out of deadlock.

Distributed Deadlock Detection & Recovery – Centralized Control Approach



Constructed by the CC

GRA Graph

Q: Is there a deadlock?

(c) Bharadwaj V 2020

Distributed Deadlock Detection & Recovery – Centralized Control Approach

- **Specific Scenario:** After releasing R, say, B puts a request for resource T; T is in the other machine. Let us see the sequence of events.

- When B releases R, the link R-B needs to be removed; This is communicated by a message (Msg1) by Machine 0 to CC to update the GRA graph;

- After deletion of the edge R-B in Machine 0, it puts a request to T; Machine 0 sends a request message to Machine 1;

- Then, Machine 1 sends this request to CC as a message Msg2) saying that it needs to add an edge B-T;
- These two are 2 independent messages communicated to CC;

Distributed Deadlock Detection & Recovery – Centralized Control Approach

What can go wrong here?

- Suppose Msg2 reaches CC first! Then CC will see a Cycle! Hence a deadlock scenario! CC will assume that this is a deadlock and may kill one of the process unnecessarily. This is referred to as a "False Deadlock".

How to avoid such False Deadlocks?

- The strategy is to use a **Global Clock**. Every machine has a global clock and whenever a msg needs to be sent from a machine to CC there will be a timestamp. Thus our messages will be: (Msg1,t1) and (Msg2,t2), where t2 > t1.

Distributed Deadlock Detection & Recovery – Centralized Control Approach

Thus when Msg2 reaches CC first **and** when it sees a deadlock, it broadcasts (O(n)!!) this condition to all machines saying that there is a deadlock and ask for any messages preceding this message. Thus, in our case, Machine 0 will resend the message and CC will find that edge R-B does not exist anymore, and hence, no deadlock. So, we can avoid by using global clocks.

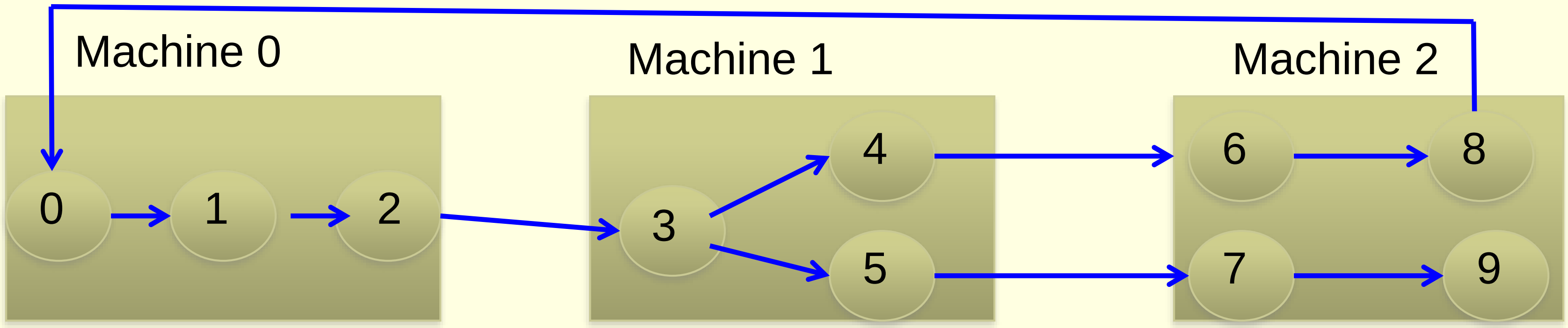
Again, what can go wrong here?

- What if CC itself dies?
- Clock skewing + correction procedures for clocks needed

Thus, this is not a fault-tolerant approach and also incurs overheads; Suits for a small-scale system;

Distributed Deadlock Detection & Recovery – Distributed Control Approach

Note: No global clock exists and no periodic updated shared knowledge;



Machine 0

Machine 1

Machine 2

Mach 0 maintains this graph

Mach 1 maintains this graph

Mach 2 maintains this graph

To draw a graph together with resources is a cumbersome process; Hence for such distributed approaches, for visualization purposes, we use a modified resource allocation graph. This is constructed as follows.

If a process **p** is waiting for a resource held by another process **q** we simply put a directed link between p and q and say that process p “waits for” q to release the resource.

Distributed Deadlock Detection & Recovery – Distributed Control Approach

Such a graph is referred to as a "***wait-for***" graph.

In the figure shown, consider P0, which is blocked. P0 will attempt to send a message in the following format:

< Source_id, Process_id that generated this message, Dest_id >

Thus, for P0, this message would be: <0,0,1>; Let us trace this message and how it percolates across the machines in the network;

Lowest chain: <0,0,1> - <1,0,2> - <2,0,3> - ... - <7,0,9>

Upper chain: <0,0,1> - <1,0,2> - <2,0,3> - ... - <6,0,8> - <8,0,0>

Distributed Deadlock Detection & Recovery – Distributed Control Approach

In the "Lower Chain" there is no deadlock, as there is no cycle formed;

In the "Upper Chain" machine 0 sent <0,0,1> and received <8,0,0>, from which we see that P0 can conclude that it is blocking or in a blocked state and hence, it can kill itself to avoid deadlock;

Consider P6: It could have generated a message simultaneously (with P0) as <6,6,8> and this message can circulate and come back to P6 as <4,6,6> and hence, P6 realizes as a blocking process and it can kill itself.

Thus, instead of killing one process two processes are getting killed unnecessarily; Killing one process is sufficient to avoid deadlock!

Key question: *Can we resolve this situation?*

Distributed Deadlock Detection & Recovery – Distributed Control Approach

To solve this issue, following is the protocol prescribed:

When a message traverses through a process, it appends its ID along with the message as follows:

Message from P0

P0 to P1: 0 <0,0,1>

P1 to P2: 1 0 <1,0,2>

...

P6 to P8: 6 4 3 2 1 0 <6,0,8>

P8 to P0: 8 6 4 3 2 1 0 <8,0,0>

Message from P6

P6 to P8: 6 <6,6,8>

P8 to P0: 8 6 <8,6,0>

...

P3 to P4: 3 2 1 0 8 6 <3,6,4>

P4 to P6: 4 3 2 1 0 8 6 <4,6,6>

Solution: Both P0 & P6 will now know they are blocking and hence both can request a higher ID process involved to kill itself– in this case it is P8;