Handling Deadlock

Handling Deadlock

Issues:

- Reusable vs. consumable resources
- Resource vs. communication deadlock
- AND vs OR deadlock
- Wait-For graphs (WFG)
- Prevention, Avoidance, Detection??
- Resolution??

Deadlock Detection Strategies

Requirements:

- No undetected deadlocks
- No “false” or “phantom” deadlocks
  - Detecting a deadlock even when there is none present in the system

Strategies:

- Centralized
- Distributed
  - Path-pushing vs. Edge-chasing algorithms
- Hierarchical

A Simple Centralized Algorithm (Ho-Ramamoorthy)

- Each node has a status table, contains status (resources locked and resources waited on) of all processes at that node
- A central site periodically collects the status table from all nodes, constructs the WFG and checks for cycles
- If no cycle detected, no deadlock
- If cycle detected, status from all nodes requested again and WFG constructed using ONLY information common both times. If the same cycle is detected again, deadlock is declared.
- Does NOT work!! Why??
- Can you make it work with additional information?

Chandy-Misra-Haas Algorithm for AND Deadlocks

- Distributed control
- An "Edge-Chasing" algorithm
- Uses a special probe message of the form (i, j, k) where:
  - p_i : process originally initiating deadlock detection
  - p_j : current sender
  - p_k : destination/receiver
- A process p_i is dependant on another process p_j if there exists a path from p_i to p_j in the WFG
- If p_i and p_j are in the same node, p_i is locally dependent on p_j

- Main Idea:
  - A blocked process p_i initiates detection by sending probes to all processes p_k at another node on which it is dependent (directly or indirectly)
  - A process receiving a probe (i, j, k) forwards it to all processes it is waiting for after changing the j and k fields appropriately, i remains unchanged.
  - Thus the probe message travels across the edges of the WFG; if it comes back to the initiator, WFG has a cycle and we have a deadlock.
- Each process maintains an array *$dependent_i$*: *$dependent_i(j)$* is true if P_i knows that P_j is dependent on it. (initially set to false for all i & j).

The Algorithm

Sending the probe (from $P_{i)}$:

if P_i is locally dependent on itself then deadlock.

else for all P_j and P_k such that

(a) P_i is locally dependent upon P_j, and

(b) P_j is waiting on P_k, and

(c) P_j and P_k are on different sites, send probe(i,j,k) to the home site of P_k.

Receiving the probe (i, j, k) at P_k :

if (d) P_k is blocked, and

(e) *dependent*$_k$*(i)* is false, and

(f) P_k has not replied to all requests of P_j,

then begin

dependent$_k$(i) := true;

if k = i then P_i is deadlocked

else ...

Receiving the probe (contd.):

…….

else for all P_m and P_n such that

(a') P_k is locally dependent upon P_m, and

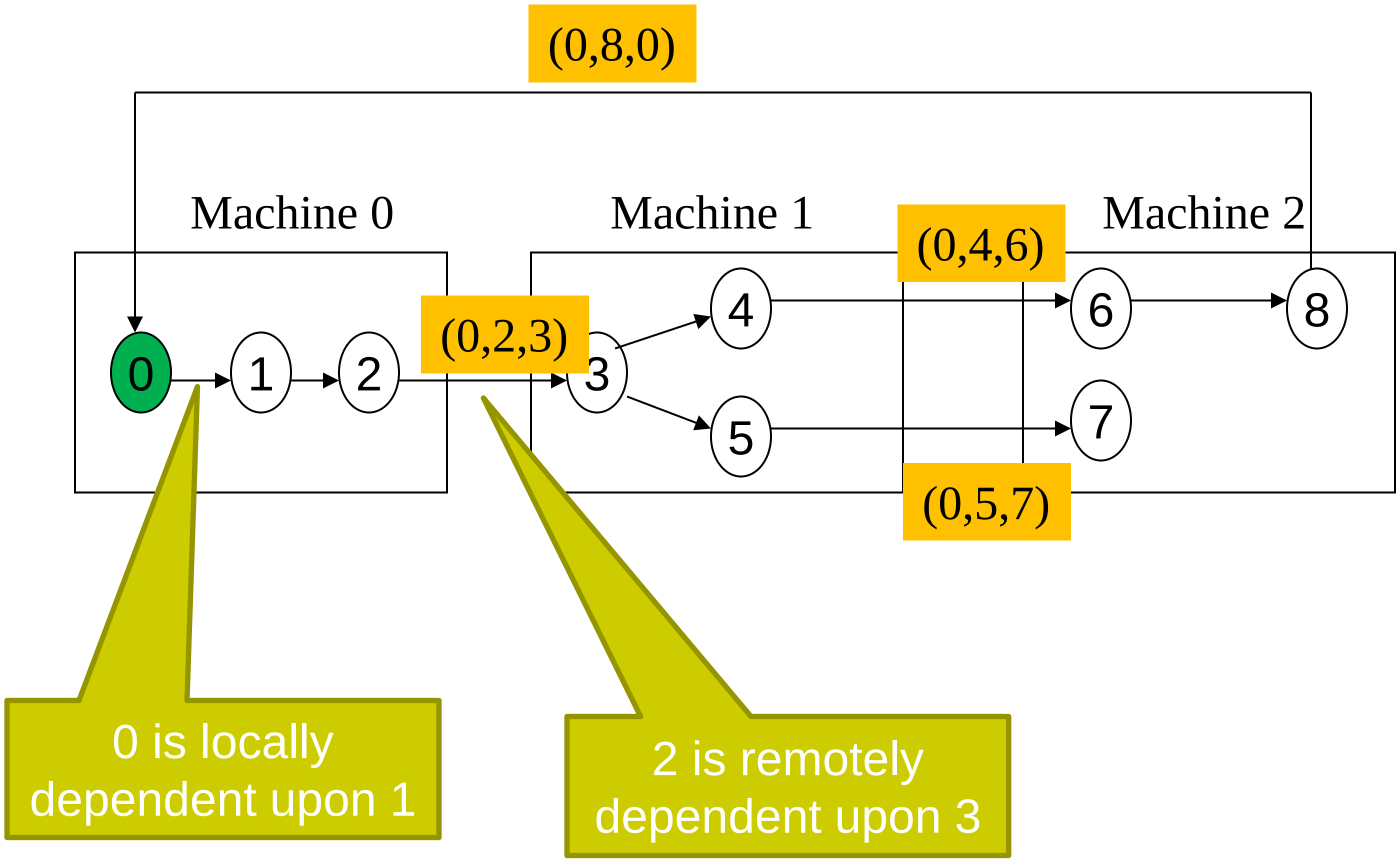
(b') P_m is waiting on P_n, and

(c') P_m and P_n are on different sites,

send probe(i,m,n) to the home site of P_n.

end.

Example


(0,8,0)
Machine 0
Machine 1
Machine 2
(0,4,6)
(0,2,3)
0
1
2
3
4
5
6
7
8
(0,5,7)
0 is locally dependent upon 1
2 is remotely dependent upon 3