Living with Deadlock!

Deadlock Recovery

Prevention, and Avoidance

The Banker's Algorithm

From detection to recovery

It should be apparent by now that deadlocks are a matter of fact and we will have to deal with them one way or another.

Once a deadlock has been detected, a recovery mechanism must be initiated.

There are two approaches to recovering from deadlock:

1. Process Termination
2. Resource Preemption

Recovery is expensive!! Why?

It is not just the time and resources spend to discover a deadlock and to cleanly remove them.

We must keep in mind that the processes that will terminate have already consumed resources.

The state of resources may have been changed and will need to be rolled back (how?).

During the execution of a now deadlocked process, other processes may have executed that changed the state of resources.
→ these processes may have to be un-done!

Process Termination

- We could simply terminate all processes that are involved in the deadlock → prohibitively expensive.

- The removal of processes is generally done incrementally, i.e., one by one until the deadlock is eliminated.

- When deciding the order of termination, we should consider:
  - The priority of the process
  - The cost restarting the process
  - The current process state

The best termination sequence is highly dependent on the particular system and its application.

Regardless of the degree of rollback, the ideal sequence is one that minimizes the cost of re-executing the set of rolled back processes.

The hardest problem to solve is that of process interleaving of processes. In general, execution sequences of processes are NOT indempotent i.e., they cannot be repeated (restarted) without side effects.

Resource Preemption

Resource Preemption takes away a contested resource from one or more processes, thereby enabling others to execute.

This attempt may leave all the processes unblocked and the deadlock is resolved.

1. Direct Preemption: The system temporarily deallocates the resource and allows other processes to use it.

→ only few resources can be handled this way transparently

2. Process Rollback: The system indirectly preempts resources by rolling back processes.

→ The system must maintain checkpoints for each process. Each checkpoint represents a snapshot of the process history.

→ Advantage: With check pointing, we achieve implicit fault tolerance; i.e., a crash does not wipe out all the work as we can go back to the last checkpoint.

→ However, rollback may involve multiple processes, resulting in a cascading rollback which is very expensive.

→ Consider (again) the problem of interleaving processes…. (Discuss!!)

Deadlock Avoidance

Rather than allowing deadlock to occur and having to recover from it, deadlock avoidance take a more conservative approach.

At runtime, the system checks and enforces rules thus avoiding deadlock from ever occurring.

Of course, we are adding additional complexity to the system as the avoidance algorithm has to be executed for each resource request.

NOTE: Deadlock avoidance is defined for reusable resources!!

The Basic Idea:

- Delay resource acquisition for requests that may cause deadlock.
- Use additional information about the future requests a process may make. ANTICIPATE !!
- Since processes never know exactly what future requests are to be made, a upper bound on their total resource need is used.

→ this is referred to as the MAXIMUM CLAIM

Maximum Claim

The maximum resource claim of a process is a conservative projection into the future.

The maximum resource claim can be represented by an extended version of the general resource graph (GRG).

Extend the GRG:

- Processes P and Resources R
- Request Edges (p_i, r_j) [solid]
- Assignment Edges (r_j, p_i) [solid]
- Potential Request (p_i, r_j) [dashed]

The Maximum Claim Graph (MCG) has the following properties:

- It shows all possible resource requests – current and future.
- The number of edges between processes and resources remains the same throughout the execution of the processes.
- Upon making a resource request, a potential request edge is changes to an actual request edge to reflect the request.
- The avoidance algorithm performs graph reduction on the MCG.

The Banker's Algorithm

The MCG is used for dynamic deadlock avoidance.

If the resulting MCG is completely reducible, then the worst of all possible cases of resource requests will not lead to a deadlock.

→The system can allow the request to be granted ➔ the system is safe

→If the graph is not completely reducible the system will disallow the acquisition.

Summary of the Algorithm:

1. When a new process enters the system, it declares the max # of instances of each resource type needed. (of course, this cannot exceed the number of available units)
2. When a process requests resources, the system projects the future state by changing the request edge to a tentative assignment edge.
3. The system then attempts to reduce the MCG (i.e., check for deadlock.
   - If safe → grant the request
   - If not safe → defer granting the request.

Example

Consider the following example:

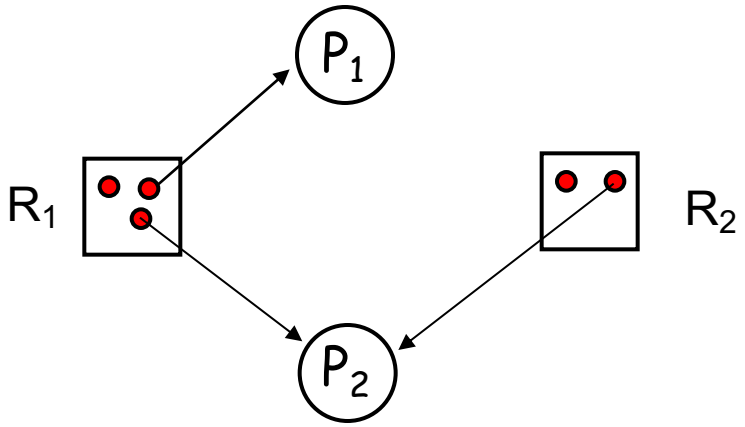
- 2 processes, p_1 and p_2
- 2 resource types, R_1 and R_2
  - R_1 has 3 units available
  - R_2 has 2 units available
- The maximum claim matrix *C* is given as:

Resources

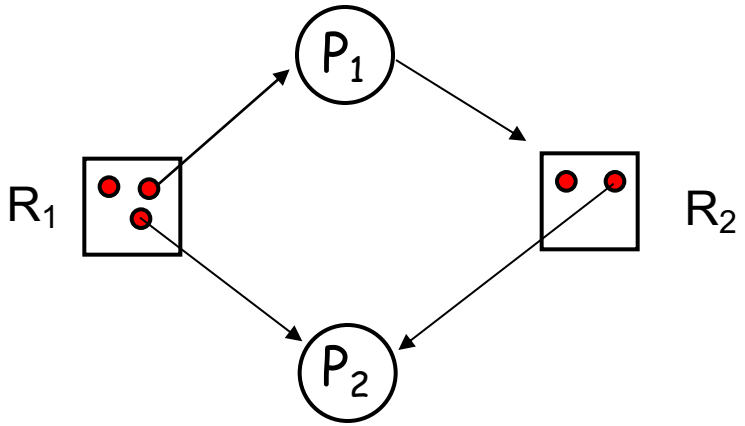
$$C = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$$

Processes

Initial System State:

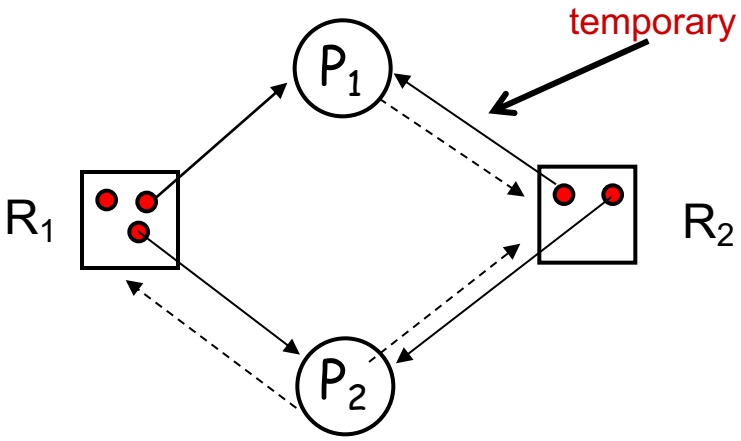


Request by p_1:



example cont'd

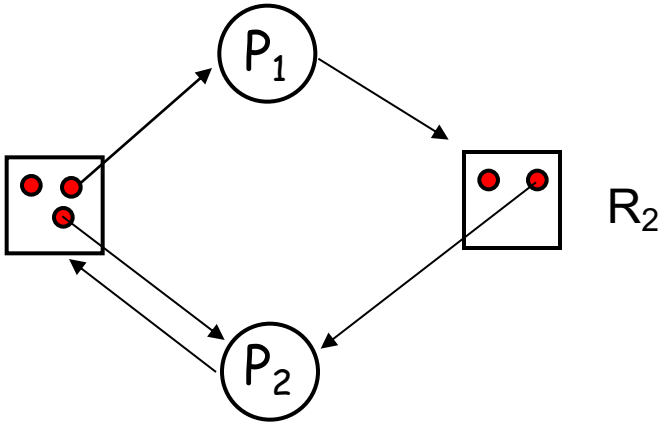
Max. Claim Limited Graph



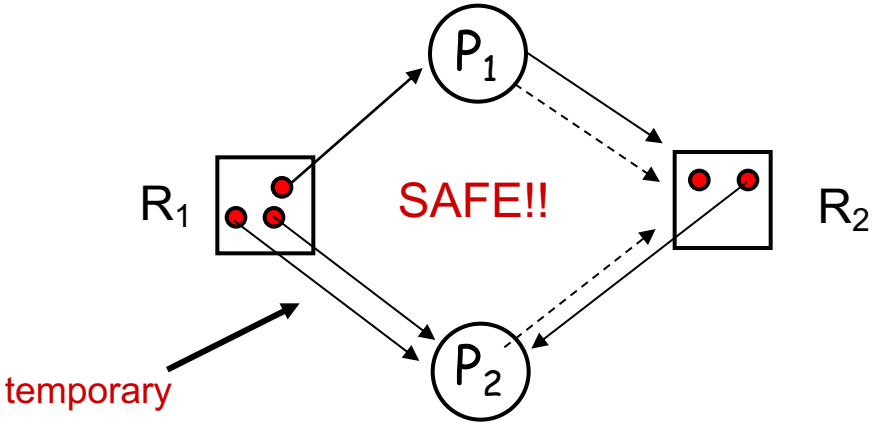
This maximum claim graph cannot be reduced if p_1's request is granted!

➔ defer granting of p_1's request

Request for R_1 by p_2



Request for R_1 by p_2 can be safely granted



...last words

Deadlock Avoidance in general and the Banker's algorithm in particular are using simple matrix operations.

The Banker's algorithm may define the following matrices and vectors:

- The maximum claim matrix *C*, representing the max resource demands by each process.
- The vector of available resources, Avail:: $[R_1, R_2, \ldots R_k]$
- The allocation matrix, Alloc, representing the current resource allocation to the processes.
- some implementations use a Need matrix to represent which resources are still needed by each process:

  Need = *C* – Alloc

Processes by request more than one unit of a particular resource type; the Banker's algorithm will work!!

Deadlock Prevention

Dynamic deadlock avoidance relies on runtime checks to assure that the system never enters an unsafe state.

Deadlock prevention is accomplished by designing the system such that deadlock will never occur.

All processes must follow strict rules for requesting and releasing resources.

Recall the 3 necessary conditions for deadlock:

1. Mutual Exclusion
2. Hold & Wait
3. Circular Wait

Eliminating any one of the 3 necessary conditions will prevent deadlock from ever occurring.

- The elimination of mutual exclusion is generally not possible!

  Why not?

- Hold & Wait can be eliminated by requiring every process to request all resources it will ever need at the same time (i.e., at startup)

  ➔ poor resource utilization.

…more prevention

- A (slightly) more flexible ay to eliminate Hold & Wait is to force processes to release all resources held before requesting additional ones.
- ➔ none of these approaches are really feasible in reality!
- The approach to eliminate circular wait requires processes to request resources in order.
- To implement ordered resources request, all resources are assigned a sequence number by which they can completely ordered,
- When a process is requesting resources, it must order the requests such that only resources with seq # higher than those currently being held are requested and/or acquired.
- How does this prevent circular wait?
- Discuss !!
- The final word on Deadlocks:

➔ Deadlocks are hard to deal with but they are a matter of fact!