Lecture 9: Deadlock Pt.2

Adam Hawley

February 6, 2019

Contents

1 Resource-Request Algorithm for Process $\mathbf{P_i}$

Let `Request_i[...]` be the request vector for process $\mathbf{P_i}$.

`Request_i[j] = k` Process $\mathbf{P_i}$ wants **k** instances of resource type $\mathbf{R_j}$.

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 $\mathbf{P_i}$ must wait since resources are not available.
3. Pretend to allocate requested resources to $\mathbf{P_i}$ by modifying the state as follows:

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

- `If safe` ⇒ the resources are allocated to $\mathbf{P_i}$
- `If unsafe` ⇒ $\mathbf{P_i}$ must wait, and the old resource-allocation state is restored.

2 Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
  - Single Instance of a Resource Type
  - Multiple Instances of a Resource Type
- Recovery Scheme

2.1 Detection Algorithm for Single Instances of a Resource Type

Maintain a **wait-for** graph. In a **wait-for** graph, nodes are only processes, there are no resources.

- $\mathbf{P_i} \rightarrow \mathbf{P_j}$ if $\mathbf{P_i}$ is waiting for $\mathbf{P_j}$

To convert between a resource-allocation graph and a wait-for graph, just join processes where one is holding a resource that the other needs. Cycles in a wait-for diagram show deadlock.

Periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of $\mathbf{n^2}$ operations, where $\mathbf{n}$ is the number of vertices in a graph, this is why this algorithm is only invoked periodically.

2.2 Detection Algorithm for Multiple Instances of a Resource Type

Where n = number of processes and m = number of resource types:

Available Vector of length m. If `Available[j] == k`, then there are k instances of resource type $\mathbf{R_j}$ available.

Allocation n x m matrix. If `Allocation[i,j] == k`, then $\mathbf{P_i}$ is currently allocated k instances of $\mathbf{R_j}$.

Request n x m matrix that indicates the current request of each process. If `Request[i,j] == k`, then process $\mathbf{P_i}$ is requesting k additional instances of resource type $\mathbf{R_j}$.

See lecture 8 for details of algorithm.

2.3 Outlining the Detection Algorithm Stage

If a deadlock is detected, we must abort (rollback) some of the processes involved in the deadlock. Need to decide when and how often to invoke the deadlock detection algorthm which depends on the following:

- How often a deadlock is likely to occur?
- How many processes will need to be rolled back?
  - (One for each disjoint cycle)

If a 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.

3 Deadlock Recovery

3.1 Process Termination

- Abort all deadlocked processes
- Abort one process at at 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 the process interactive or batch?

3.2 Resource Preemption

Selecting a victim Minimise cost

Rollback Return to some safe state, restart process for that state

Starvation If the same process is always picked as a victim then it could suffer from starvation. To avoid this, we can include the number of rollbacks in cost factor.

4 Process Management - Conclusion

What we have covered:

- Processes, threads and their life cycle
- Scheduling
- Synchronising
- Deadlock
- Hardware Support

In the next section: memory management.