Lecture 8: Deadlock

Adam Hawley

February 5, 2019

Contents

1 Introduction

Systems consist of resources:

- Resource types R_1, R_2, ..., R_m
- CPU cycles, memory space, I/O devices

Each resource type R_i has W_i instances. Each process utilises a resource using the following actions:

- Request
- Use
- Release

A set of processes is deadlocked if each process in the set is waiting for an action that only another process in the set can cause.

Note: A resource is considered **preemptable** if it can be taken away from the process owning it (at any time) with no ill effects.

2 Conditions for Deadlock

Deadlock can arise if the following four conditions hold simultaneously:

1. **Mutual Exclusion**: Only one process at a time can use a resource.
2. **Hold and Wait**: A process holding at least one resource is waiting to acquire additional resources held by other processes.
3. **No Preemption**: A resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular Wait**: Circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

If one of these conditions is absent, no deadlock is possible.

3 Resource Allocation Graphs

In resource allocation graphs we have nodes to represent processes as well as nodes to represent resources. See lecture slides for examples.

4 Methods for Handling Deadlocks

Ensuring that the system will never enter a deadlock state is usually done using one of two methods:

- **Deadlock Prevention** (4.1)
- **Deadlock Avoidance** (4.2)

Or we can allow the system to enter a deadlock state and then recover afterwards using **Deadlock Recovery**. The final method is to ignore the problem and pretend that deadlocks never occur in the system (used by most OSs, including UNIX, Linux & Windows).

4.1 Deadlock Prevention

Ensure that at least one of the necessary conditions for deadlocks does not hold. This can be accomplished by restraining the ways a request can be made:

Mutual Exclusion Avoid mutually-exclusive access to resources. This is impractical as most systems have inherently non-sharable resources that cannot be accessed simultaneously by various processes.

Hold and Wait Must guarantee that whenever a process requests a resource, it does not hold any other resources. This requires processes to request and be allocated all its resources only when the process has none allocated to it. This may result in low resource utilisation and possibly starvation.

Therefore, both of these are impractical to try and prevent.

No Preemption While also impractical, resource preemption may be imposed as follows:

- If a process that is holding some resources requests another resource than cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Circular Wait Impose a total ordering of all types.

- Require that each process requests resources in an increasing order of enumeration.

4.2 Deadlock Avoidance

Deadlock avoidance is when you ensure that the system will never enter a deadlock state. This requires that the system has some additional **a priori** information available on possible requests.

The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need. Deadlock-aoidance algorithms dynamixally examie the resource-allocation state to ensure that there cam never be a circular-wait condition. Resource-allocation

state is defined by the number of available and allocated resources, and the maximum demands of the processes.

4.2.1 Safe States

Safe states are the foundation of every deadlock-avoidance algorithm. The system is said to be in a **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of all of the processes in the systems such that for each P_i the resources that P_i can still request can be satisfied by currently available resources and resources held by all the P_j, with $j<i$.

That is:

1. If a P_i resource needs are not immediately available, then P_i can wait until all processes P_j ($j<i$) have finished executing.
2. When they have finished executing they release all their resources and then P_i can obtain the needed resources, execute, return allocated resources and terminate.
3. When P_i terminates, $P_{(i+1)}$ can obtain its needed resources and so on.

- If a system is in a **safe** state $\Rightarrow$ no deadlocks
- If a system is in an **unsafe** state $\Rightarrow$ possibility of deadlocks

4.2.2 Deadlock Avoidance Algorithms

If we have a single instance of a resource type; we can use a variant of the resource-allocation graph. (1) Or if we have multiple instances of a resource type: we can use the *banker's algorithm* (2)

1. Resource-Allocation Graph Algorithm Single instance of a resource type. Each process must a priori claim maximum resource use. Use a variant of the resource-allocation graph with claim edges. **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i **may** request resource R_j, represented by a dashed line. Claim edge converts to request edge when a process requests a resource. Request edge converted to an assignment edge when tthe resource is allocated the process. When a resource is released by a process, assignment edge reconverts to a claim edge. Resources must be a claimed a priori in the system. A cycle in the graph implies that the system is in an unsafe state.

2. Banker's Algorithm Multiple instances of a resource type. (Created by Edsger Dijkstra) Each process must a priori claim maximum use. When a process requests a resource it may have to wait. When a process gets all of its resources it must return them in a finite amount of time. The algorithm uses the analogy of an interest-free bank where:

   - A customer establishes a line of credit.
   - Borrows money in chunks that together never exceed the total line of credit.
   - Once it reaches the maximum, the customer must pay back in a finite amount of time.

   To implement the Banker's Algorithm, we use the following data structures 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 R_j available.

   Max nm matrix

   - If `Max[i,j] = k`, then process P_i may request at most k instances of resource type R_j.

   Allocation nm matrix:

   - If `Allocation[i,j] = k`, then P_i is currently allocated k instances of R_j.

   Need nm matrix:

   - If `Need[i,j] = k`, then P_i may need at most k more instances of R_j to complete its task

   Therefore: `Need[i,j] = Max[i,j] - Allocation[i,j]`

   The **safety algorithm** is part of the Banker's Algorithm. It involves the following:

   (a) Let **Work** and **Finish** be vectors of length m and n, respectively. Initialise:

   - `Work = Available`
   - `Finish[i] = false for i = 0, 1, ..., n-1`

   (b) Find an **i** such that both:

- `Finish[i] = false`
- `Need_i <= Work`

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

(c) `Work = Work + Allocation_i` `Finish[i] = true` Go to step 2.

(d) If `Finish[i] == true` for all **i**, then the system is in a safe state. Otherwise it is in an unsafe state.