Deadlocks

Andrés Ponce

December 2, 2020

Deadlocks result from the situation when multiple threads want access to a resource that is being held by another thread waiting to access that same resource.

System Model

When a thread makes a request to use a specific resource, it typically follows the steps

1. **Request** The thread makes a request to access the specific resource. If access cannot be granted, then the thread must wait until that resource becomes available.
2. **Use** The thread can make use of the specific resource, given that for example, it can access its critical section.
3. **Release** Control of the resource is returned.

Operating systems don't provide any protection against deadlocks, and instead the responsibility lies on the application programmer to avoid deadlock.

Deadlock in Multithreaded Applications

Deadlock is hard to predict because it can only occur in certain conditions. When there are two threads `thread_one` and `thread_two`, and each want to access two mutex locks, if `thread_one` can get hold of the first lock and second thread gets hold of the second lock, deadlock would occur. To continue execution, the first thread would be waiting for the second lock, and the second thread is waiting for access to the first thread .

Since the instructions might be scheduled differently by the CPU, it might not be easy to rpedict if these two sequences would cause deadlock.

Livelock is similar to deadlock in that it prevents threads from executing. In this scenario, one thread constantly attempts an action that fails, and thus is unable to make any progress.

Deadlock Characterization

If the following four conditions simultaneously hold on a system, then deadlock will occur:

- **Mutual Exclusion** This means that there is a resource r such that only one thread can access r at the same time. If r is in use, the other thread must wait until the resource has been delayed.
- **Hold And Wait**: A thread has to be occupying at least one resource and wants to access one more.
- **No preemption**: If a thread t occupies resource r, then only t can releaes r, it cannot be forced by the operating system to release any resource it occupies.
- **Circular Wait**: There is a set of threads $[T_1, T_2, ..., T_n]$ such that T_1 is waiting for T_2's resource and so on. T_n is waiting for the resource held by T_0.

Again, all these conditions have to be true simultaneously for deadlock to occur.

If we use a resource allocation graph, where the resources are mapped to the threads that are using them, we can model deadlocks by finding cycles in this graph.

If we use a resource allocation graph, where the resources are mapped to the threads that are using them, we can model deadlocks by finding cycles in this graph. If the resource only has one instance, then deadlock occurs. If there is more than one instance, the other thread occupying the other instance might release it and thus break the cycle.

Methods for Handling Deadlocks

There are three general ways of avoiding deadlock:

- Ignore the deadlock and pretend they never happen in the system. [1]
- We can try to prevent or avoid deadlocks so that they never happen.
- We can let the program enter into a deadlocked state and then recover.

[1] Don't try this approach for your personal issues!

Deadlock Prevention

To prevent possible deadlocks, we can do so by targeting the deadlock properties we had seen previously. The **mutual exclusion** principle seems to be very unflexible, since some resources are not very flexible.

The **hold and wait** principle states that a thread will be using a resource before requesting another resource. We could have the thread request all the resources that it's going to need at the start, but this might be very inefficient. Or we might make a thread ask for a resource only if it has none. However, none of these approaches are all that efficient, since utilization could never be that high.

If we want to stop the **no-preemption** principle, we could preempt any thread that asks for a resource that can't be provided at the given time, and delay its execution until that resource becomes available. However, this approach probably could not be applied to variables such as semaphores and mutex locks, since they have to be empted to be check-able?

The last alternative involves avoiding the **circular wait** condition. The algorithm involves assigning the set of resources $R = \{r_1, r_2, ..., r_n\}$ a total ordering by having a function F that assigns an integer to the ordering of the resources. Then, when the thread makes the request, we can only grant it if the thread has released all resources with a higher value integer. These two conditions guarantee that there will be no circular wait.

Deadlock Avoidance

Safe State

Another way of preventing deadlock is to have the programs first declare what resources they will be using. If we know the resources that thread T_i will need,and in what order, then we can check the graph and make sure that no deadlock state will be reached by granting such resources.

A system is in a **safe state** if it can allocate all the resources to the competing threads and noe reach a deadlock. For all T_i, its resources can either be granted from the currently unoccupies resources, or they can be obtained from T_j with $j < i$.

Resource-Allocation Graph Algorithm

In this approach, we use a similar type of graph to the resource allocation graph. Here, each edge from a resource R_i to a thread T_i represents an assignment, and an edge from a thread to a resource represents a request. A dashed line represents that a thread might request a resource in the future.

When a thread is about to start making requests, we need to know it a-priori and we need to add them to the graph. The reason for this is that we constantly need to check for cycles, in which case we cannot assign the requested resource.

Banker's Algorithm

The gist of this algorithm is that a banker that gives out cash can do so in such a way that he will not be left without any cash to give out. When a new thread comes in to the system, it must first declare how many resources it will need of each type. Then, we need to make sure that granting such resources will still leave us in a safe state, otherwise we cannot grant that.

We need some data structures:

- **Available** This is a list of the available resources. If `Available[i] = k`, then there are k instances of that resource.
- **Max** This is the maximum amount of resources of type R_j that thread T_i is expected to make. An nxm matrix keeps track of how many resources T_i makes. matrix$[i][j]$ is the number of resources R_j that T_i makes.
- **Allocation**: How many resources of a type R_j are allocated to thread T_i.
- **Need**: This nxm matrix will keeps track of how many remaining items of a certain type are needed. This is just the difference of $Max[i][j] - Allocation[i][j]$.

Safety Algorithm

The previous approaches all rely on the safe state approach, so we need to know whether or not the system is in a safe state. The **Safety Algorithm** goes as follows:

1. Initialize $Work$ and $Finish$ to vectors of length m and n, respectively.
2. Check if there is an unfinished thread which needs less resources than are availale (can be scheduled).
3. Add $Allocation_i$ to $Work$.
4. If all $finish[i] == True \forall i$, then we are in a safe state.

Resource-Request Algorithm

In this algorithm, we have to check whether the required resources are available, subtract the needed resources from the available, and then update the $Available, Allocation, Need$ matrices.

Deadlock Detection

The previous lesson involved finding deadlocks and taking action to prevent it. We can also not take care of them, and present algorithms to just detect whether they have had. We can also draw a graph of which thread is waiting for which other thread that has a resource. Periodically, we also need to check whether there is a cycle in that graph.

When there are several isntances of a resource type, we cannot rely on detecting cycles, since the other instance of a resource might become available and thus break the cycle if two threads are waiting for the same resource.

For multiple instances of a single resource, the algorithm used is still similar to the safety algorithm, and we use some of the data structures used by the banker's algorithm. We check if the resources are available for the threads that are still not done with their workload.

Thus, we can regularly schedule the algorithm to run, on predefined intervals. Then

Recovery From Deadlock

Once we detect that there is deadlock, we have one of two options, both of which involve a lot of overhead.

- **Abandon all deadlocked processes**: This approach will get rid of the deadlock, but also of processes that potentially have been computing for long and will have to recompute later.
- **Remove a process until the deadlock is broken**: This approach carries some significant overhead since the deadlock-detection algorithm would have to be run every time we remove a process to see if the system remains deadlocked. [2]

[2] Remember the deadlock-detection algorithms rely on finding cycles in a graph, which would take $O(n^2)$ since we have to check a loop of length n for all n nodes.

Clearly, we are going to have to preempt a process, so how do we choose which process to preempt? There are some things we should consider:

1. **Selecting the Victim**: Which resource do we select as the victim? We should consider how many resources it is currently using and how much resources it has used so far.
2. **Rollback**: When we choose a victim process, we need to roll back its execution to a safe state. The process can't continue with normal execution until the resource becomes available, so we have to roll it back to some previous state. However, determining what is a safe state requires us to keep more information for each process, we might just terminate that process entirely.
3. **Starvation**: How can we guarantee that the victim process isn't constantly chosen as the victim process and deprived of resources?

The answer might be to include the number of rollbacks the current process has experienced and factor that into our decision.