Distributed Deadlock

Paul Krzyzanowski
Rutgers University

November 2017

1 Introduction

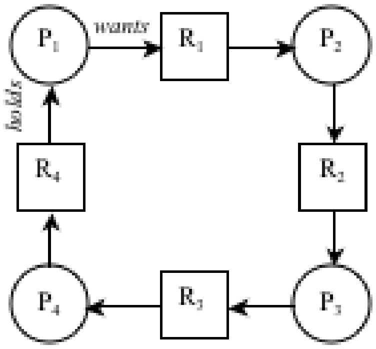

Figure 1: Figure 1. Deadlock

A **deadlock** is a condition in a system where a set of processes (or threads) have requests for resources that can never be satisfied. Essentially, a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, Coffman[1] defined four conditions have to be met for a deadlock to occur in a system:

1. **Mutual exclusion** A resource can be held by at most one process.
2. **Hold and wait** Processes that already hold resources can wait for another resource.
3. **Non-preemption** A resource, once granted, cannot be taken away.
4. **Circular wait** Two or more processes are waiting for resources held by one of the other processes.

[1] http://people.cs.umass.edu/∼mcorner/courses/691J/papers/TS/coffman_deadlocks/coffman_deadlocks.pdf

A directed graph model used to record the resource allocation state of a system. This state consists of n processes, P1 … Pn, and m resources, R1 … $m. In such a graph:

P1 R1 means that resource R1 is allocated to process P1.

P1 R1 means that resource R1 is requested by process P1.

Deadlock is present when the graph has a directed cycles. An example is shown in Figure 1. Such a graph is called a **Wait-For Graph** (**WFG**).

2 Deadlock in distributed systems

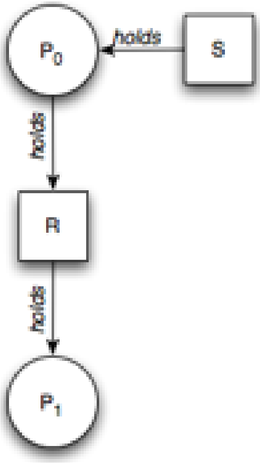

Figure 2: Figure 2. Resource graph on A

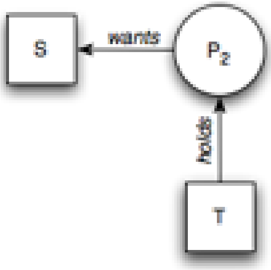

Figure 3: Figure 3. Resource graph on B

The same conditions for deadlock in uniprocessors apply to distributed systems. Unfortunately, as in many other aspects of distributed systems, they are harder to detect, avoid, and prevent. Four strategies can be used to handle deadlock:

1. **ignorance**: ignore the problem; assume that a deadlock will never occur. This is a surprisingly common approach.
2. **detection**: let a deadlock occur, detect it, and then deal with it by aborting and later restarting a process that causes deadlock.

3. **prevention**: make a deadlock impossible by granting requests so that one of the necessary conditions for deadlock does not hold.

4. **avoidance**: choose resource allocation carefully so that deadlock will not occur. Resource requests can be honored as long as the system remains in a safe (non-deadlock) state after resources are allocated.

The last of these, deadlock avoidance through resource allocation is difficult and requires the ability to predict precisely the resources that will be needed and the times that they will be needed. This is difficult and not practical in real systems. The first of these is trivially simple but, of course, ineffective for actually doing anything about deadlock conditions. We will focus on the middle two approaches.

In a conventional system, the operating system is the component that is responsible for resource allocation and is the ideal entity to detect deadlock. Deadlock can be resolved by killing a process. This, of course, is not a good thing for the process. However, if processes are transactional in nature, then aborting the transaction is an anticipated operation. Transactions are designed to withstand being aborted and, as such, it is perfectly reasonable to abort one or more transactions to break a deadlock. The transaction can be restarted later at a time when, we hope, it will not create another deadlock.

3 Centralized deadlock detection

Centralized deadlock detection attempts to imitate the nondistributed algorithm through a central coordinator. Each machine is responsible for maintaining a resource graph for its processes and resources. A central coordinator maintains the resource utilization graph for the entire system: the **Global Wait-For Graph**. This graph is the union of the individual Wait-For Graphs. If the coordinator detects a cycle in the global wait-for graph, it aborts one process to break the deadlock.

In the non-distributed case, all the information on resource usage lives on one system and the graph may be constructed on that system. In the distributed case, the individual subgraphs have to be propagated to a central coordinator. A message can be sent each time an arc is added or deleted. If optimization is needed, a list of added or deleted arcs can be sent periodically to reduce the overall number of messages sent.

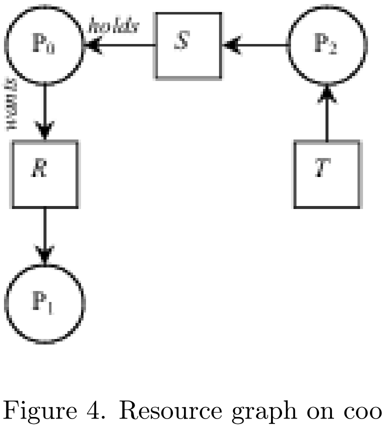

Figure 4: Figure 4. Resource graph on coordinator

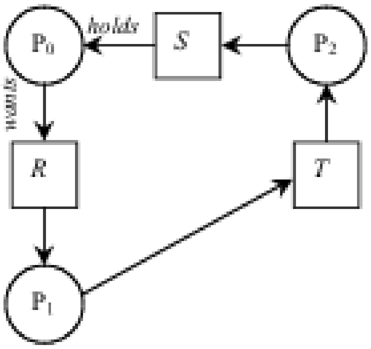

Figure 5: Figure 5. False deadlock

Here is an example (from Tanenbaum[2]). Suppose machine A has a process P0, which holds the resource S and wants resource R, which is held by P1. The local graph on A is shown in Figure 2. Another machine, machine B, has a process P2, which is holding resource T and wants resource S. Its local graph is shown in Figure 3. Both of these machines send their graphs to the central coordinator, which maintains the union (Figure 4).

All is well. There are no cycles and hence no deadlock. Now two events occur. Process P1 releases resource R and asks machine B for resource T. Two messages are sent to the coordinator:

```
message 1 (from machine A): ‘‘releasing R’’
message 2 (from machine B): ‘‘waiting for T’’
```

This should cause no problems (no deadlock). However, if message 2 arrives first, the coordinator would then construct the graph in Figure 5 and detect a deadlock. Such a condition is known as **false deadlock**. A way to fix this is to use Lamport's algorithm to impose global time ordering on all machines. Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any release messages. Each machine will then respond with either a release message or a negative acknowledgement to acknowledge receipt of the message.

4 Distributed deadlock detection

An algorithm for detecting deadlocks in a distributed system was proposed by Chandy, Misra, and Haas in 1983. Processes request resources from the current holder of that resource. Some processes may wait for resources, which may be held either locally or remotely. Cross-machine arcs make looking for cycles, and hence detecting deadlock, difficult. This algorithm avoids the problem of constructing a Global WFG.

The **Chandy-Misra-Haas algorithm** works this way: when a process has to wait for a resource, a **probe** message is sent to the process holding that resource. The probe message contains three components: the process ID that blocked, the process ID that is sending the

[2] http://www.amazon.com/dp/0132392275/pkorg

request, and the destination. Initially, the first two components will be the same. When a process receives the probe: if the process itself is waiting on a resource, it updates the sending and destination fields of the message and forwards it to the resource holder. If it is waiting on multiple resources, a message is sent to each process holding the resources. This process continues as long as processes are waiting for resources. If the originator gets a message and sees its own process number in the blocked field of the message, it knows that a cycle has been taken and deadlock exists. In this case, some process (transaction) will have to die. The sender may choose to commit suicide and abort itself or an election algorithm may be used to determine an alternate victim (e.g., youngest process, oldest process, . . .).

5 Distributed deadlock prevention

An alternative to detecting deadlocks is to design a system so that deadlock is impossible. We examined the four conditions for deadlock. If we can deny at least one of these conditions then we will not have deadlock.

Mutual exclusion To deny this means that we will allow a resource to be held (used) by more than one process at a time. If a resource can be shared then there is no need for mutual exclusion and deadlock cannot occur. Too often, however, a process requires mutual exclusion for a resource because the resource is some object that will be modified by the process.

Hold and wait Denying this means that processes that hold resources cannot wait for another resource. This typically implies that a process should grab all of its resources at once. This is not practical either since we cannot always predict what resources a process will need throughout its execution.

Non-preemption A resource, once granted, cannot be taken away. In transactional systems, allowing preemption means that a transaction can come in and modify data (the resource) that is being used by another transaction. This differs from mutual exclusion since the access is not concurrent but the same problem arises of having multiple transactions modify the same resource. We can support this with optimistic concurrency control algorithms that will check for out-of-order modifications at commit time and roll back (abort) if there are potential inconsistencies.

Circular wait Avoiding circular wait means that we ensure that a cycle of waiting on resources does not occur. We can do this by enforcing an ordering on granting resources and aborting transactions or denying requests if an ordering cannot be granted.

One way of avoiding circular wait is to obtain a globally-unique timestamp (e.g., Lamport total ordering) for every transaction so that no two transactions get the same timestamp. When one process is about to block waiting for a resource that another process is using, check which of the two processes has a younger timestamp and give priority to the older process.

If a younger process is using the resource, then the older process (that wants the resource) waits. If an older process is holding the resource, the younger process (that wants the resource) aborts itself. This forces the resource utilization graph to be directed from older to younger processes, making cycles impossible. This algorithm is known as the **wait-die algorithm**.

An alternative, but similar, method by which resource request cycles may be avoided is to have an old process abort (kill) the younger process that holds a resource. If a younger process wants a resource that an older one is using, then it waits until the older process is done. In this case, the graph flows from young to old and cycles are again impossible. This variant is called the **wound-wait algorithm**.

6 References

- Andrew S. Tanenbaum and Maarten Van Steen, Distributed Systems: Principles and Paradigms, Second Edition. Prentice Hall, October 2006.
- K Mani Chandy, Jayadev Misra, and Laura M. Haas, Distributed Deadlock Detection, ACM Transactions on Computer Systems, Vol. 1, No. 2, May 1983, Pages 144–156.
- E. G. Coffman, Jr; M. J. Elphick; A. Shoshani, System Deadlocks, Computing Surveys, Vol. 3, No. 2, June 1971.
- Edgar Knapp, Deadlock Detection in Distributed Databases, ACM Computing Surveys, Vol. 19, No. 4, December 1987

This document is modified from an original version published on October 23, 2012.