NOTES ON DEADLOCK - CS624

ERNESTO GOMEZ

Note. This document is not complete, it will be expanded.

Deadlock has generally been studied in the context of operating systems or databases, many of the proofs/ algorithms may not be directly applicable to message passing computational systems; however people still try to apply them.

[3] characterizes deadlock as a situation in which there is a subset \( \Sigma \) of states such that there is no transition out of \( \Sigma \), and there are no transitions in \( \Sigma \) which cause forward progress. Tanenbaum does not here characterize progress, for this see [4], chapter 4.

[1] cites four conditions that are required to hold simultaneously for deadlock to exist in a system:

1. **Mutual exclusion**: there exists at least one resource that can be used by only one process at a time, and must be released by that process before it can be used by another process (e.g. a message, a buffer, a communication channel, etc.).

2. **Hold and wait**: there must be a process that holds at least one resource and is waiting to acquire additional resources held by another process (e.g. a process is waiting to receive a message from a different process before sending its own message).

3. **No preemption**: a resource can only be released by the process holding it.

4. **Circular wait**: there must exist a set of processes \( \{p_0, p_1, \ldots, p_n\} \) such that 
   - \( p_0 \) is waiting for a resource held by \( p_1 \), \( p_1 \) is waiting for \( p_2 \), \ldots, \( p_{n-1} \) is waiting for \( p_n \) and \( p_1 \) is waiting for \( p_0 \).

Conditions 2 and 4 mean that deadlock a network property; it cannot be detected by examining the state of a single process. In an asynchronous network in particular, a process cannot infer that some other process will never release a resource from the fact that it has not done so even over an arbitrarily long time.

There is a tacit assumption in the above, that the resource being held is independent of the process holding it. This is not true for messaging systems where the resource is data generated by a process. In particular, we can get deadlock by the definition of [3] without condition 4, if some process \( p_1 \) does not execute a send which some other process \( p_2 \) is waiting for. In this case \( p_2 \) waits forever, but there is no cycle because \( p_1 \) can continue. (This kind of thing is sometimes called **livelock** - one or more processes are blocked while others continue. We consider this to be deadlock of a subset, which clearly falls under the definition of [3] and is implied in any case by condition 4 if the set of processes is not all processes).

[2] describes (page 159 and following) four common strategies for dealing with deadlock:

1. **Ostrich**: (ignore the problem)
2. **Detection**: (let deadlocks happen, detect them, try to recover)
3. Prevention: (stably make deadlock impossible)
4. Avoidance: (carefully allocate resources so deadlocks don’t happen)

Approach 1 relies on wishful thinking, and leads to the systems that appear to work most of the time and die when we really need them. [2] comments that most theoretical results on approach 2 are expensive, impractical or wrong, and often all three! (For example, the deadlock detection algorithm given in Lynch is based on depth-first search, which has exponential complexity. It also requires a global snapshot of all processes.)

One approach suggested in [2] involves having a process send a message to the resource it is waiting for, with the request that the message be passed on to any process that holds a resource that the receiving process is waiting for. If the message returns to the original sender, there is a circular dependence, therefore deadlock. Note that if the message does not return, we don’t know that there is a deadlock – as we explain above, deadlock can happen without circular dependence if there is dependence on a missing process, and in any case the message could take indefinite time to return in an asynchronous network.

A general approach rely on constructing a graph of dependences between processes in the network and detecting a cycle in the graph. Recovery generally requires killing one of the processes that holds a resource. One of our resources, however, is messages between processes. If we kill a process that is holding a message to break a circular dependence, we may lose data needed for a correct result. In a distributed computation, the failure of one process is frequently a sufficient condition to require aborting the entire computation!

(Except: transactions based systems such as databases where each transaction has independent existence and is designed to be atomic - that is, either it completes, or leaves the database in the same state as before the transaction was executed. Accomplishing this is generally expensive, see any reference on database systems).

We are left with strategies 3 and 4. Database and operating systems often attempt to prevent circular dependence by ensuring that all resources required by a transaction or process are acquired first, before the process is allowed to proceed. If the resources are not available, the process is aborted. This can be very time and resource intensive, assumes that resources do not depend on the presence of other processes (e.g. disks, terminals, data in a database, printers, cpu time, etc.). It is often necessary to have some arbitrary tiebreaking scheme in case two processes request the same resource simultaneously.

Strategy 4 is what we use with MPI and in general with message passing systems. We try to carefully write sends and receives so that they match, and so that we don’t get a circular dependence (e.g., p1 needs to send to p2, but can not do it until it receives a message from p2, but p2 is waiting for a message from p1 before it sends, so both wait forever). Being certain that no circular dependences can exist is difficult in practice, particularly in the presence of many processes with the possibility that a cycle will involve some large combination of them. It is also possible that such dependences will be sensitive to timing; that is, we may enter a circular dependence only if some specific processes are faster than some other specific processes but not otherwise. We could then get deadlock in some but not all runs, making it very difficult to debug the program.
REFERENCES