WIKIBOOKS

Operating System Design/Concurrency/Deadlock

In computer science, **deadlock** refers to a specific condition when two or more processes are each waiting for another to release a resource, or more than two processes are waiting for resources in a circular chain (see *Necessary conditions*). Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software,* or *soft,* lock. Computers intended for the *time-sharing* and/or *real-time* markets are often equipped with a *hardware lock* (or *hard lock*) which guarantees *exclusive access* to processes, forcing serialization. Deadlocks are particularly troubling because there is no *general* solution to avoid (soft) deadlocks.

This situation may be understood by an analogy with two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can't be satisfied, so a deadlock occurs.

The telecommunications description of deadlock is a little stronger: deadlock occurs when none of the processes meet the condition to move to another state (as described in the process's finite state machine) *and* all the communication channels are empty. The second condition is often left out on other systems but is important in the telecommunication context and its systems.

Contents

Necessary conditions

There are four necessary conditions for a deadlock to occur, known as the *Coffman conditions* from their first description in a 1971 article by E. G. Coffman.

1. Mutual exclusion condition: a resource cannot be used by more than one process at a time
2. Hold and wait condition: processes already holding resources may request new resources
3. No preemption condition: only a process holding a resource may release it
4. Circular wait condition: two or more processes form a circular chain where each process waits for a resource that the next process in the chain holds

Deadlock can only occur in systems where all 4 conditions hold true.

Circular wait prevention

Circular wait prevention consists of allowing processes to wait for resources, but ensure that the waiting cannot be circular. One approach might be to assign a precedence to each resource and force processes to request resources in order of increasing precedence. That is to say that if a process holds some resources, and the highest precedence of these resources is *m*, then this process cannot request any resource with precedence smaller than *m*. This forces resource allocation to follow a particular and non-circular ordering, so circular wait cannot occur. Another approach is to allow holding only one resource per process; if a process requests another resource, it must first free the one it's currently holding (or hold-and-wait).

Examples

An example of a deadlock which may occur in database products is the following. Client applications using the database may require exclusive access to a table, and in order to gain exclusive access they ask for a *lock*. If one client application holds a lock on a table and attempts to obtain the lock on a second table that is already held by a second client application, this may lead to deadlock if the second application then attempts to obtain the lock that is held by the first application. (But this particular type of deadlock is easily prevented, e.g., by using an *all-or-none* resource allocation algorithm.)

Another example might be a text formatting program that accepts text sent to it to be processed and then returns the results, but does so only after receiving "enough" text to work on (e.g. 1KB). A text editor program is written that sends the formatter with some text and then waits for the results. In this case a deadlock may occur on the last block of text. Since the formatter may not have sufficient text for processing, it will suspend itself while waiting for the additional text, which will never arrive since the text editor has sent it all of the text it has. Meanwhile, the text editor is itself suspended waiting for the last output from the formatter. This type of deadlock is sometimes referred to as a *deadly embrace* (properly used only when only two applications are involved) or *starvation*. However, this situation, too, is easily prevented by having the text editor send a *forcing* message (eg. EOF) with its last (partial) block of text, which will *force* the formatter to return the last (partial) block after formatting, and not wait for additional text.

Nevertheless, since there is no *general* solution for deadlock prevention, each type of deadlock must be anticipated and specially prevented. But *general* algorithms can be implemented within the operating system so that if one or more applications becomes blocked, it will usually be terminated after (and, in the meantime, is allowed no other resources and may need to surrender those it already has, rolled back to a state prior to being obtained by the application).

Avoidance

Deadlock can be avoided if certain information about processes is available in advance of resource allocation. For every resource request, the system sees if granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states. In order for the system to be able to figure out whether the next state will be safe or unsafe, it must know in advance at any time the number and type of all resources in existence, available, and requested. One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance. However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.

Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a time stamp at process creation time. Smaller time stamps are older processes, while larger timestamps represent younger processes.

	Wait/Die	**Wound/Wait**
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

It is important to note that a process may be in unsafe state but would not result in a deadlock. The notion of safe/unsafe state only refers to the ability of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

Prevention

Deadlocks can be prevented by ensuring that at least one of the following four conditions occur:

- Removing the mutual exclusion condition means that no process may have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
- The "hold and wait" conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations); this advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens, are known as the all-or-none algorithms.)
- A "no preemption" (lockout) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a *priority* algorithm. (Note: Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead.) Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.
- The circular wait condition: Algorithms that avoid circular waits include "disable interrupts during critical sections" , and "use a hierarchy to determine a partial ordering of resources" (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra's solution.

Detection

Often neither deadlock avoidance nor deadlock prevention may be used. Instead deadlock detection and process restart are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS.

Detecting the possibility of a deadlock *before* it occurs is much more difficult and is, in fact, *generally* undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in *specific* environments, using *specific* means of locking resources, deadlock detection may be *decidable*. In the *general* case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

Distributed deadlock

Distributed deadlocks can occur in distributed systems when distributed transactions or concurrency control is being used. Distributed deadlocks can be detected either by constructing a global wait-for graph, from local wait-for graphs at a deadlock detector or by a distributed algorithm like edge chasing.

Phantom deadlocks are deadlocks that are detected in a distributed system.