31

Deadlock

In this note we discuss one of the most basic problems of systems with complex locking protocols: **deadlock**. Deadlock occurs, for example, when a thread (say Thread 1) is holding a lock (L1) and waiting for another one (L2); unfortunately, the thread (Thread 2) that holds lock L2 is waiting for L1 to be released. Here is a code snippet that demonstrates such a potential deadlock:

```
Thread 1:                Thread 2:
  lock(L1);                lock(L2);
  lock(L2);                lock(L1);
```

Note that if this code runs, deadlock does not necessarily occur; rather, it may occur, if, for example, Thread 1 grabs lock L1 and then a context switch occurs to Thread 2. At that point, Thread 2 grabs L2, and tries to acquire L1. Thus we have a deadlock, as each thread is waiting for the other and neither can run. See Figure 31.1 for details; the presence of a **cycle** in the graph is indicative of the deadlock.

The figure should make clear the problem. How should programmers write code so as to handle deadlock in some way?

> CRUX: HOW TO DEAL WITH DEADLOCK
>
> How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

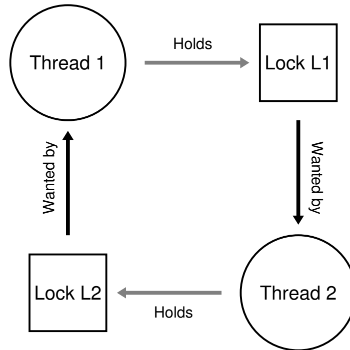


Figure 31.1: The Deadlock Dependency Graph

31.1 Why Do Deadlocks Occur?

As you may be noting already, simple deadlocks such as the one above seem readily avoidable. For example, if thread 1 and 2 both made sure to grab locks in the same order (which we will discuss further below), the deadlock would never arise. So why do deadlocks happen?

One reason is that in large code bases, complex dependencies exist between components. Take the OS, for example. The virtual memory system might need to access the file system in order to page in a block from disk; the file system might subsequently require a page of memory to read the block into and thus contact the virtual memory system. Thus, the design of locking strategies in large systems must be carefully done to avoid deadlock in the case of circular dependencies that may arise naturally in the code.

Another reason is due to the nature of **encapsulation**. As software developers, we are taught to hide details of implementations and thus make software easier to build in a modular way. Unfortunately, such modularity does not mesh well with locking. As Jula et al. point out [J+08], some seemingly innocuous interfaces almost invite you

to deadlock. For example, take the Java Vector class and the method `AddAll()`. This routine would be called as follows:

```
Vector v1, v2;
v1.AddAll(v2);
```

Internally, because the method needs to be multi-thread safe, locks for both the vector being added to (v1) and the parameter (v2) need to be acquired. The routine acquires said locks in some arbitrary order (say v1 then v2) in order to add the contents of v2 to v1. If some other thread calls `v2.AddAll(v1)` at nearly the same time, we have the potential for deadlock, all in a way that is quite hidden from the calling application.

31.2 Conditions for Deadlock

Four conditions need to hold for a deadlock to occur [C+71]:

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** There exists a circular chain of threads such that each thread holds one more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur. Thus, we first explore techniques to *prevent* deadlock; each of these strategies seeks to prevent one of the above conditions from arising and thus is one approach to handling the deadlock problem.

31.3 Prevention

Circular Wait

Probably the most practical prevention technique (and certainly one that is used in many systems today) is to write your locking code such that you never cause a circular wait to arise. The way to do that is to *provide a total ordering on lock acquisition*. For example, if there are only two locks in the system (L1 and L2), we can ensure deadlock does not occur by always making sure to acquire L1 before L2. Such strict ordering ensures that no cyclical wait can arise and hence no deadlock.

As you can imagine, this approach requires careful design of global locking strategies and must be done with great care. Further, it is just a convention, and a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock. Finally, it requires a deep understanding of the code base, and how various routines are called; just one mistake could result in the wrong ordering of lock acquisition, and hence deadlock.

Hold-and-wait

The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically. In practice, this could be achieved as follows:

```
lock(prevention);
lock(L1);
lock(L2);
...
unlock(prevention);
```

By first grabbing the lock `prevention`, this code guarantees that no untimely thread switch can occur in the midst of lock acquisition and thus deadlock can once again be avoided. Of course, it requires that any time any thread grabs a lock, it first acquires the global prevention lock. For example, if another thread was trying to grab locks L1 and L2 in a different order, it would be OK, because it would be holding the prevention lock while doing so.

Note that the solution is problematic for a number of reasons. As before, encapsulation works against us: this approach requires us to know when calling a routine exactly which locks must be held and to acquire them ahead of time. Further, the approach likely decreases concurrency as all locks must be acquired early on (at once) instead of when they are truly needed.

No Preemption

Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another. Many thread libraries provide a more flexible set of interfaces to help avoid this situation. Specifically, a `trylock()` routine will grab the lock (if it is available) or return -1 indicating that the lock is held right now and that you should try again later if you want to grab that lock.

Such an interface could be used as follows to build a deadlock-free, ordering-robust lock acquisition protocol:

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
  }
```

Note that another thread could follow the same protocol but grab the locks in the other order (L2 then L1) and the program would still be deadlock free. One new problem does arise, however: **livelock**. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. In this case, both systems are running through this code sequence over and over again (and thus it is not a deadlock), but progress is not being made, hence the name livelock. There are solutions to the livelock problem, too: for example, one could add a random delay before looping back and trying the entire thing over again, thus decreasing the odds of repeated interference among competing threads.

One final point about this solution: it skirts around the hard parts of using a trylock approach. The first problem that would likely exist again arises due to encapsulation: if one of these locks is buried in

some routine that is getting called, the jump back to the beginning becomes more complex to implement. If the code had acquired some resources (other than L1) along the way, it must make sure to carefully release them as well; for example, if after acquiring L1, the code had allocated some memory, it would have to release that memory upon failure to acquire L2, before jumping back to the top to try the entire sequence again. However, in limited circumstances (e.g., the Java vector method above), this type of approach could work well.

Mutual Exclusion

The final prevention technique would be to avoid the need for mutual exclusion at all. In general, we know this is difficult, because the code we wish to run does indeed have critical sections. So what can we do?

Herlihy had the idea that one could design various data structures to be **wait-free** [H91]. The idea here is simple: using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking.

As a simple example, let us assume we have a compare-and-swap instruction, which as you may recall is an atomic instruction provided by the hardware that does the following:

```
int CompareAndSwap(int *address, int expected, int new) {
  if (*address == expected) {
    *address = new;
    return 1; // success
  }
  return 0; // failure
}
```

Imagine we now wanted to atomically increment a value by a certain amount. We could do it as follows:

```
void AtomicIncrement(int *value, int amount) {
  do {
    int old = *value;
  } while (CompareAndSwap(value, old, old + amount) == 0);
}
```

Instead of acquiring a lock, doing the update, and then releasing it, we have instead built an approach that repeatedly tries to update the value to the new amount and uses the compare-and-swap to do

so. In this manner, no lock is acquired, and no deadlock can arise (though livelock is still a possibility).

Let us consider a slightly more complex example: list insertion. Here is code that inserts at the head of a list:

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  n->next  = head;
  head     = n;
}
```

This code performs a simple insertion, but if called by multiple threads at the "same time", has a race condition (see if you can figure out why). Of course, we could solve this by surrounding this code with a lock acquire and release:

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  lock(listlock);   // begin critical section
  n->next  = head;
  head     = n;
  unlock(listlock); // end of critical section
}
```

In this solution, we are using locks in the traditional manner[1]. Instead, let us try to perform this insertion in a wait-free manner simply using the compare-and-swap instruction. Here is one possible approach:

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  do {
    n->next = head;
  } while (CompareAndSwap(&head, n->next, n));
}
```

[1]The astute reader might be asking why we grabbed the lock so late, instead of right when entering the `insert()` routine; can you, astute reader, figure out why that is OK?

The code here updates the next pointer to point to the current head, and then tries to swap the newly-created node into position as the new head of the list. However, this will fail if some other thread successfully swapped in a new head in the meanwhile, causing this thread to retry again with the new head.

Of course, building a useful list requires more than just a list insert, and not surprisingly building a list that you can insert into, delete from, and perform lookups on in a wait-free manner is non-trivial. Read more of the rich literature on wait-free synchronization if you find this interesting.

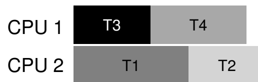
31.4 Avoidance via Scheduling

Instead of deadlock prevention, in some scenarios deadlock **avoidance** is preferable. Avoidance requires some global knowledge of which locks various threads might grab during their execution, and subsequently schedules said threads in a way as to guarantee no deadlock can occur.

For example, assume we have two processors and four threads which must be scheduled upon them. Assume further we know that Thread 1 (T1) grabs locks L1 and L2 (in some order, at some point during its execution), T2 grabs L1 and L2 as well, T3 grabs just L2, and T4 grabs no locks at all. In tabular form:

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

A smart scheduler could thus compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise. Here is one such schedule:



Note that it is OK for (T3 and T1) or (T3 and T2) to overlap. Even though T3 grabs lock L2, it can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

Let's look at one more example. In this one, there is more contention for the same resources (again, locks L1 and L2), as indicated by the following contention table:

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

In particular, threads T1, T2, and T3 all need to grab both locks L1 and L2 at some point during their execution. Here is a possible schedule that guarantees that no deadlock could ever occur:

As you can see, static scheduling leads to a conservative approach where T1, T2, and T3 are all run on the same processor, and thus the total time to complete the jobs is lengthened considerably. Though it may have been possible to run these tasks concurrently, the fear of deadlock prevents us from doing so, and the cost is performance.

One famous example of an approach like this is Dijkstra's Banker's Algorithm [D64], and many similar approaches have been described in the literature. Unfortunately, they are only useful in very limited environments, for example, in an embedded system where one has full knowledge of the entire set of tasks that must be run and the locks that they need. Further, such approaches can limit concurrency, as we saw in the second example above. Thus, avoidance of deadlock via scheduling is not a widely-used general-purpose solution.

31.5 Detect and Recover

One final general strategy is to allow deadlocks to occasionally occur, and then take some action once such a deadlock has been detected. For example, if an OS froze once a year, you would just reboot it and get happily (or grumpily) on with your work. If deadlocks are rare, such a non-solution is indeed quite pragmatic.

DESIGN TIP: TOM WEST'S LAW

Tom West, famous as the subject of the classic computer-industry book "Soul of a New Machine" [K81], says famously: "Not everything worth doing is worth doing well", which is a terrific engineering maxim. If a bad thing happens rarely, certainly one should not spend a great deal of effort to prevent it, particularly if the cost of the bad thing occurring is small.

Many database systems employ deadlock detection and recovery techniques. A deadlock detector runs periodically, building a resource graph and checking it for cycles. In the event of a cycle (deadlock), the system needs to be restarted. If more intricate repair of data structures is first required, a human being may be involved to ease the process.

31.6 Summary

We have briefly discussed deadlock: why it occurs, and what can be done about it. The problem is as old as concurrency itself, and many hundreds of papers have been written about the topic. The best solution in practice is to be careful, develop a lock acquisition total order, and thus prevent deadlock from occurring in the first place. Wait-free approaches also have promise, as some wait-free data structures are now finding their way into commonly-used libraries and critical systems, including Linux. However, their lack of generality and the complexity to develop a new wait-free data structure will likely limit the overall utility of this approach. Perhaps the best solution is to develop new concurrent programming models: in systems such as MapReduce (from Google) [GD02], programmers can describe certain types of parallel computations without any locks whatsoever. Locks are problematic by their very nature; thus, perhaps we should seek to avoid using them unless we truly have to.

References

[C+71] "System Deadlocks"
E.G. Coffman, M.J. Elphick, A. Shoshani
ACM Computing Surveys, 3:2, June 1971
The classic paper outlining the conditions for deadlock and how you might go about dealing with it. There are certainly some earlier papers on this topic; see the references within this paper for details.

[D64] "Een algorithme ter voorkoming van de dodelijke omarming"
Circulated privately, around 1964
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD108.PDF
Indeed, not only did Dijkstra come up with a number of solutions to the deadlock problem, he was the first to note its existence, at least in written form. However, he called it the "deadly embrace", which (thankfully) did not catch on.

[GD02] "MapReduce: Simplified Data Processing on Large Clusters"
Sanjay Ghemawhat and Jeff Dean
OSDI 2004
The MapReduce paper ushered in the era of large-scale data processing, and proposes a framework for performing such computations on clusters of generally unreliable machines.

[H91] "Wait-free Synchronization"
Maurice Herlihy
ACM TOPLAS, 13(1), pages 124-149, January 1991
Herlihy's work pioneers the ideas behind wait-free approaches to writing concurrent programs. These approaches tend to be complex and hard, often more difficult than using locks correctly, probably limiting their success in the real world.

[J+08] "Deadlock Immunity: Enabling Systems To Defend Against Deadlocks"
Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, George Candea
OSDI '08, San Diego, CA, December 2008
An excellent recent paper on deadlocks and how to avoid getting caught in the same ones over and over again in a particular system.

[K81] "Soul of a New Machine"
Tracy Kidder, 1980
A must-read for any systems builder or engineer, detailing the early days of how a team inside Data General (DG), led by Tom West, worked to produce a "new machine." Kidder's other book are also excellent, in particular, "Mountains beyond Mountains". Or maybe you don't agree with me, comma?