31

Common Concurrency Problems

Researchers have spent a great deal of time and effort looking into concurrency bugs over many years. Much of the early work focused on **deadlock**, a topic which we've touched on in the past chapters but will now dive into deeply [C+71]. More recent work focuses on studying other types of common concurrency bugs (i.e., non-deadlock bugs). In this chapter, we take a brief look at some example concurrency problems found in real code bases, to better understand what problems to look out for. And thus our problem:

> CRUX: HOW TO HANDLE COMMON CONCURRENCY BUGS
>
> Concurrency bugs tend to come in a variety of common patterns. Knowing which ones to look out for is the first step to writing more robust, correct concurrent code.

31.1 What Types Of Bugs Exist?

The first, and most obvious, question is this: what types of concurrency bugs manifest in complex, concurrent programs? This question is difficult to answer in general, but fortunately, some others have done the work for us. Specifically, we rely upon a study by Lu et al. [L+08], which analyzes a number of popular concurrent applications in great detail to understand what types of bugs arise in practice.

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Table 31.1: Bugs In Modern Applications

The study focuses on four major and important open-source applications: MySQL (a popular database management system), Apache (a well-known web server), Mozilla (the famous web browser), and OpenOffice (a free version of the MS Office suite). In the study, the authors look through the concurrency bugs that have been found and fixed in each of these code bases, turning the developers' work into a quantitative bug analysis.

Table 31.1 shows a summary of the bugs Lu and her colleagues studied. From the table, you can see that there were 105 total bugs, most of which were not deadlock (74); the remaining 31 were deadlock bugs. Further, you can see that the number of bugs studied from each application; while OpenOffice only had 8 total concurrency bugs, Mozilla had nearly 60.

We now dive into these different classes of bugs (non-deadlock, deadlock) a bit more deeply. For the first class of non-deadlock bugs, we use examples from the study to drive our discussion. For the second class of deadlock bugs, we discuss the long line of work that has been done in either preventing, avoiding, or handling deadlock.

31.2 Non-Deadlock Bugs

Non-deadlock bugs make up a majority of concurrency bugs, according to Lu's study. But what types of bugs are these? How do they arise? How can we fix them? We now discuss the two major types of non-deadlock bugs found by Lu et al.: **atomicity violation** bugs and **order violation** bugs.

Atomicity-Violation Bugs

The first type of problem encountered is referred to as an **atomicity violation**. Here is a simple example, found in MySQL. Before reading the explanation, try figuring out what the bug is. Do it!

```
Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}

Thread 2::
thd->proc_info = NULL;
```

In the example, two different threads access the field `proc_info` in the structure `thd`. The first thread checks if the value is non-NULL and then prints its value; the second thread sets it to NULL. Clearly, if the first thread performs the check but then is interrupted before the call to `fputs`, the second thread could run in-between, thus setting the pointer to NULL; when the first thread resumes, it will crash, as a NULL pointer will be dereferenced by `fputs`.

The more formal definition of an atomicity violation, according to Lu et al, is this: "The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution)." In our example above, the code has an *atomicity assumption* (in Lu's words) about the check for non-NULL of `proc_info` and the usage of `proc_info` in the `fputs()` call; when assumption is broken, the code will not work as desired.

Finding a fix for this type of problem is often (but not always) straightforward. Can you think of how to fix the code above?

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&lock);
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...
}
pthread_mutex_unlock(&lock);

Thread 2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

In this solution, we simply add locks around the shared-variable references, ensuring that when either thread accesses the `proc_info` field, it has a lock held. Of course (not shown), any other code that accesses the structure should also acquire this lock before doing so.

Order-Violation Bugs

Another common type of non-deadlock bug found by Lu et al. is known as an **order violation**. Here is another simple example; once again, see if you can figure out why the code below has a bug in it!

```
Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);
   ...
}

Thread 2::
void mMain(...) {
    ...
    mState = mThread->State;
    ...
}
```

As you probably figured out, the code in Thread 2 seems to assume that the variable `mThread` has already been initialized (and is not NULL); however, if Thread 1 does not happen to run first, we are out of luck, and Thread 2 will likely crash with a NULL pointer dereference (assuming that the value of `mThread` is initially NULL; if not, even strager things could happen as arbitrary memory locations are read through the dereference in Thread 2).

The more formal definition of an order violation is this: "The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)." [L+08]

The fix to this type of bug is generally to enforce ordering. As we discussed in detail previously, using **condition variables** is an easy and robust way to add this style of synchronization into modern code bases. In the example above, we could thus rewrite the code as follows.

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
int mtInit             = 0;

Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);

   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    ...
}
```

In this fixed-up code sequence, we have added a lock (`mtLock`) and corresponding condition variable (`mtCond`), as well as a state variable (`mtInit`). When the initialization code runs, it sets the state of `mtInit` to 1 and signals that it has done so. If Thread 2 had run before this point, it will be waiting for this signal and corresponding state change; if it runs later, it will check the state and see that the initialization has already occurred (i.e., `mtInit` is set to 1), and thus continue as is proper. When ordering matters between threads, condition variables (or semaphores) can come to the rescue.

Non-Deadlock Bugs: Summary

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations. Thus, by carefully thinking about these types of bug patterns, programmers can likely do a better job of avoiding them. Moreover, as more automated code-checking tools develop, they should likely focus on these two types of bugs as they constitute such a large fraction of non-deadlock bugs found in deployment.

Unfortunately, not all bugs are as easily fixable as the examples we looked at above. Some require a deeper understanding of what the program is doing, or a larger amount of code or data structure reorganization to fix. Read Lu et al.'s excellent (and very readable) paper for more details.

31.3 Deadlock Bugs

Beyond the concurrency bugs mentioned above, a classic problem that arises in many concurrent systems with complex locking protocols is known as **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?

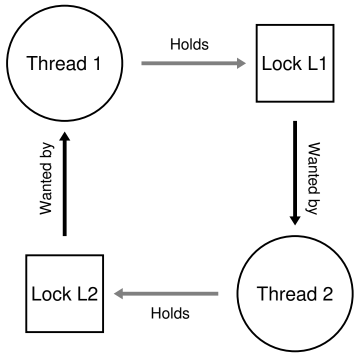


Figure 31.1: The Deadlock Dependency Graph

Why Do Deadlocks Occur?

As you may be thinking, 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, 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.

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.

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));
}
```

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.

[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?

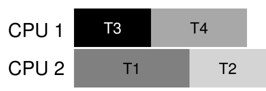
Deadlock Avoidance via Scheduling

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:



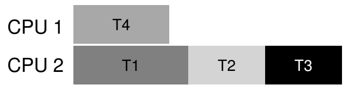
Note that it is OK for (T3 and T1) or (T3 and T2) to overlap. Even though T3 grabs lock L2, it can never cause a deadlock by running concurrently with other threads because it only grabs one lock.

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.

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.

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.

TIP: DON'T ALWAYS DO IT PERFECTLY (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.

31.4 Summary

In this chapter, we have studied the types of bugs that occur in concurrent programs. The first type, non-deadlock bugs, are surprisingly common, but often are easier to fix. They include atomicity violations, in which a sequence of instructions that should have been executed together was not, and order violations, in which the needed order between two threads was not enforced.

We have also 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 must.

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?

[L+08] "Learning from Mistakes – A Comprehensive Study on
Real World Concurrency Bug Characteristics"
Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou
ASPLOS '08, March 2008, Seattle, Washington
The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at YY Zhou's or Shan Lu's web pages for many more interesting papers on bugs.