**Chapter 32: Common Concurrency Problems**

One of the most common problems is **deadlock**.

**32.1 What Types Of Bugs Exist?**

Some common concurrency bugs in open source applications (MySQL, Apache, Mozilla and OpenOffice).

Graphical user interface, text

Description automatically generated

There were total of 105 bugs and 74 of them are non-deadlock bugs. The number of bugs also varies among applications.

**32.2 Non-Deadlock Bugs**

Non-deadlock bugs make up the majority of concurrency bugs. The two major types are **atomicity violation** bugs and **order violation** bugs.

**Atomicity-Violation Bugs**

Text, letter

Description automatically generated

In the above example, when thread 1 finished checking if the proc\_info is not NULL, thread 2 can interrupt and make it NULL. Thus, the fputs function would dereference NULL pointer, which will cause error.

**The definition of atomicity violation** is “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).”

A solution to such problem is to use locks around shared-variable references, ensuring that when a variable is accessed, it is done atomically.

Text

Description automatically generated

**Order-Violation Bugs**

Graphical user interface, text, application

Description automatically generated

The bug deals with the order of execution. For example, thread 2 assumed that mThread has been initialized. However, thread 1 might not have run and mThread has not been initiated. Thus, it will likely to cause a crash for dereferencing a NULL pointer.

**The formal definition of order-violation bugs** is “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)”.

TO fix this issue, we use condition variables and locks to enforce ordering.

Text

Description automatically generated

**Non-Deadlock Bugs: Summary**

A large fraction (97%) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations.

Unfortunately, not all bugs are as easily fixed 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.

**32.3 Deadlock Bugs**

Deadlock occurs when a thread is holding a lock and waiting for another one, but the other lock is never released.

Text

Description automatically generated with medium confidence

When thread 1 grabs lock L1 and then context switch occurs to thread 2. Thread 2 then grabs lock L2 and tries to acquire lock L1. In such scenario, both threads are waiting for each other, so no thread can proceed.

**Why Do Deadlocks Occur?**

One reason is that in large code bases, complex dependencies arise between components. 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 occur 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.

**Conditions for Deadlock**

Four conditions need to hold for a deadlock to occur:

* **Mutual exclusion**: Threads claim exclusive control of resources that they require.
* **Hold-and-wait**: threads hold resources allocated to them while waiting for additional resources.
* **No preemption**: resources 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 or more resources that are being requested by the next thread in the chain.

If one of the four conditions are not met, deadlock cannot occur.

**Prevention:**

**Circular Wait**

The most practical prevention technique is to write your locking code such that you never induce a circular wait. The most straightforward 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), you can prevent deadlock by always acquiring L1 before L2. Such strict ordering ensures that no cyclical wait arises.

**Partial ordering** can be a useful way to structure lock acquisition so as to avoid deadlock. An excellent real example of partial lock ordering can be seen in the memory mapping code in Linux.

Another thing we could do is to order the lock using lock address. By acquiring locks in either high-to-low or low-to-high address order, do something() can guarantee that it always acquires locks in the same order, regardless of which order they are passed in.

**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:

Text

Description automatically generated

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.

As before, encapsulation works against us: when calling a routine, this approach requires us to know exactly which locks must be held and to acquire them ahead of time. This technique also is likely to decrease 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, the routine pthread\_mutex\_trylock() either grabs the lock (if it is available) and returns success or returns an error code indicating the lock is held. We can try again later if we want to grab that lock.

Text, letter

Description automatically generated

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.

However, we have a new problem here: **livelock**. It is possible (though perhaps unlikely) that two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. The system runs through these sequences over and over again without any proceeding. The solution could be adding a random delay before looping back and try again.

There are some problems with this 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. However, in limited circumstances (e.g., the Java vector method mentioned earlier), this type of approach could work well.

You might also notice that this approach doesn’t really add preemption.

**Mutual Exclusion**

The final prevention technique would be to avoid the need for mutual exclusion at all (**lock-free** and **wait-free**). The approach is using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking.

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated with medium confidence

Instead of using a lock, we build an approach that repeatedly tries to update the value to the new amount and use compare-and-swap to do so. In this manner, no deadlock can arise.

Another example would be inserting element to the head in a list. Instead of using a lock on critical section to create new node and link together, we would use compare-and-swap as follows:

Graphical user interface, text, application

Description automatically generated

The insert above will try to insert at the head of the list. If fails, it will retry with a new head.

**Deadlock Avoidance via Scheduling**

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.

Consider the following example:

Text

Description automatically generated with medium confidence

Where “yes” indicates that a thread acquires a lock, “no” otherwise. A scheduler can schedule the thread as follows:

A picture containing diagram

Description automatically generated

This works because no deadlock could happen at any time.

Consider another example:

Text

Description automatically generated with medium confidence

Then, in this case, T1, T2 and T3 cannot be executed concurrently because it might cause deadlock. Thus, a valid schedule can be:

Chart, waterfall chart

Description automatically generated

However, this approach is 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.

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.