Chapter 32: Concurrency Bugs

overview

Concurrency is a powerful tool for building high-performance applications, but it comes with its own set of challenges. When multiple threads execute concurrently, they can interact in unexpected ways, leading to a class of bugs known as **concurrency bugs**. These bugs are notoriously difficult to find and fix because they are often non-deterministic, meaning they don't appear every time the program is run.

This chapter explores the most common types of concurrency bugs and provides strategies for preventing, avoiding, and detecting them. We'll categorize these bugs into two main types: **non-deadlock bugs** and **deadlock bugs**.

> "Concurrency bugs have been the bane of many a programmer's existence. They are subtle, hard to reproduce, and can lead to catastrophic failures." - OSTEP

Non-Deadlock Bugs

Non-deadlock bugs are the most common type of concurrency bug. They arise from incorrect assumptions about the atomicity and ordering of operations across threads.

Atomicity-Violation Bugs

An **atomicity violation** occurs when a sequence of operations that should be executed as a single, atomic unit is interrupted by another thread. This interruption can leave the program in an inconsistent state.

Consider the following code snippet from a real-world application:

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

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

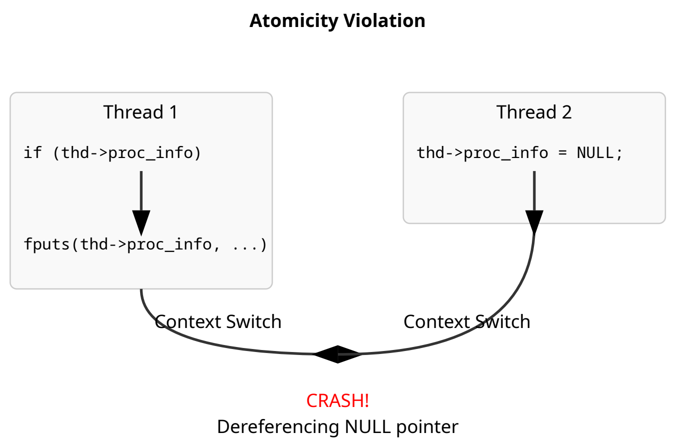
In this example, Thread 1 checks if `thd->proc_info` is not `NULL`. If the check passes, it proceeds to use the pointer. However, between the check and the use, Thread 2 can be scheduled, which sets `thd->proc_info` to `NULL`. When Thread 1 resumes, it will try to dereference a `NULL` pointer, leading to a crash.

This is a classic example of a **race condition**. To fix this, we need to ensure that the check and the use of the pointer are atomic. This can be achieved using a lock:

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

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

By wrapping the critical section in a lock, we ensure that no other thread can interfere with the check and use of `thd->proc_info`.



Order-Violation Bugs

An **order violation** occurs when the intended order of operations between two threads is not enforced. This often happens when one thread assumes that another thread has already completed a certain task.

Here's an example from the Mozilla codebase:

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

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

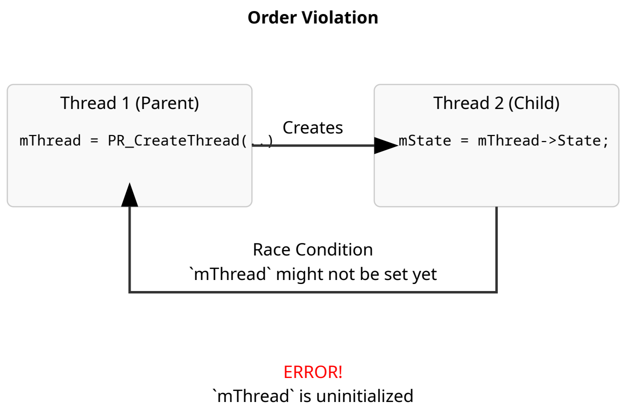
In this case, Thread 1 creates a new thread and assigns it to `mThread`. Thread 2, which is the newly created thread, then tries to access its own state through the `mThread` variable. The problem is that the assignment to `mThread` in Thread 1 might not have happened yet when Thread 2 starts executing. This is an order violation because the code assumes that the assignment to `mThread` will happen before the new thread starts running.

To fix this, we need to enforce the intended order. This can be done using condition variables:

```
// In Thread 1
pthread_mutex_lock(&lock);
mThread = PR_CreateThread(mMain, ...);
while (mThread->State == UNINITIALIZED) {
    pthread_cond_wait(&cond, &lock);
}
pthread_mutex_unlock(&lock);

// In Thread 2
pthread_mutex_lock(&lock);
mThread->State = INITIALIZED;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```

By using a condition variable, we can make Thread 1 wait until Thread 2 has finished its initialization.



Deadlock Bugs

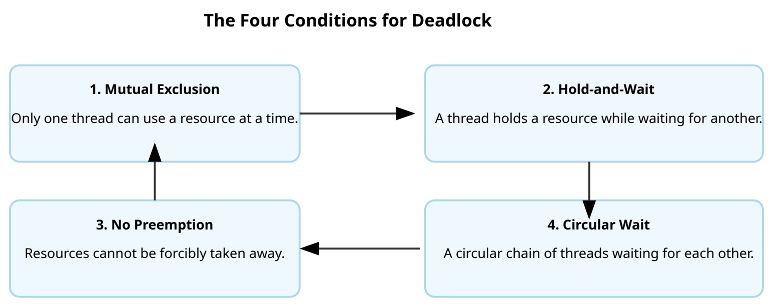
Deadlock is a situation where two or more threads are blocked forever, waiting for each other. Deadlocks are one of the most feared concurrency bugs because they can bring the entire system to a halt.

The Four Conditions for Deadlock

For a deadlock to occur, four conditions must hold simultaneously:

1. **Mutual Exclusion:** Threads claim exclusive control of resources that they require.
2. **Hold-and-Wait:** Threads hold resources allocated to them while waiting for additional resources.
3. **No Preemption:** Resources cannot be forcibly removed from threads that are holding them.
4. **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 any of these conditions is not met, deadlock cannot occur.


The Four Conditions for Deadlock
1. Mutual Exclusion
Only one thread can use a resource at a time.
2. Hold-and-Wait
A thread holds a resource while waiting for another.
3. No Preemption
Resources cannot be forcibly taken away.
4. Circular Wait
A circular chain of threads waiting for each other.

Deadlock Prevention

The most straightforward way to deal with deadlocks is to prevent them from happening in the first place. This can be done by breaking one of the four deadlock conditions.

Breaking Circular Wait A common strategy is to break the circular wait condition by enforcing a **lock order**. If all threads acquire locks in the same order, a circular wait is impossible.

For example, if we have two locks, `L1` and `L2`, we can enforce the rule that `L1` must always be acquired before `L2`.

```
// Thread 1
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);

// Thread 2
```

```
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
```

By enforcing this order, we prevent the deadlock scenario where Thread 1 holds `L1` and waits for `L2`, while Thread 2 holds `L2` and waits for `L1`.

Breaking Hold-and-Wait Another approach is to break the hold-and-wait condition by acquiring all locks at once, atomically.

```
pthread_mutex_lock(prevention_lock);
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
pthread_mutex_unlock(prevention_lock);
```

This approach can be effective, but it can also be complex to implement and may reduce concurrency.

Using `trylock` A more flexible approach is to use the `trylock` variant of the lock acquisition function. `trylock` will attempt to acquire the lock, but if it's already held, it will return immediately instead of blocking. This allows a thread to release the locks it holds and try again later, thus breaking the hold-and-wait condition.

```
while (1) {
    pthread_mutex_lock(L1);
    if (pthread_mutex_trylock(L2) == 0) {
        // Success
        break;
    } else {
        // Failure
        pthread_mutex_unlock(L1);
        // Try again later
    }
}
```

Deadlock Avoidance

Deadlock avoidance is a more dynamic approach where the system tries to avoid deadlocks by carefully scheduling threads. This approach requires some global knowledge of which locks each thread might grab during its execution.

For example, a scheduler could be designed to only run a set of threads that are known to be safe (i.e., they won't lead to a deadlock). However, this approach is often too conservative and can severely limit concurrency.

Deadlock Detection and Recovery

If prevention and avoidance are not feasible, another option is to detect deadlocks when they occur and then recover from them. This is a common approach in database systems.

A deadlock detector can be built to periodically check for cycles in the resource allocation graph. If a cycle is detected, the system can take action to break the deadlock, for example, by killing one of the threads involved in the deadlock.

Conclusion

Concurrency bugs are a serious problem in modern software development. This chapter has provided an overview of the most common types of concurrency bugs and the strategies for dealing with them.

Bug Type	Description	Prevention/Solution
Atomicity Violation	A sequence of operations that should be atomic is interrupted.	Use locks to protect critical sections.
Order Violation	The intended order of operations between threads is not enforced.	Use condition variables to enforce order.
Deadlock	Two or more threads are blocked forever, waiting for each other.	Prevention, avoidance, or detection and recovery.

By understanding the nature of these bugs and the tools available to combat them, you can write more robust and reliable concurrent programs.