Chapter 32 Common Concurrency Problems

Now we look at common concurrency bugs:

- Deadlock-related bugs
- Non-deadlock bugs

32.2 Non-Deadlock Bugs

Two major types of non-deadlock bugs:

- Atomicity violation bugs
- Order violation bugs

Atomicity violation bugs

Example from MySQL

Consider this code:

```
type Thread struct {
        procInfo        *ProcInfo
}
```

Thread 1:

```
if thrd.procInfo != nil {
        …
        fputs(thrd.procInfo, …)
        …
}
```

Thread 2:

```
thrd.procInfo = nil
```

Q: What is the problem?

Thread 1 checks if procInfo != nil and calls fputs()
using procInfo, expecting it to be non-nil.

Thread 2 may run in-between and set procInfo to nil.

Formally: *The desired serializability (on after the other) among multiple memory accesses is violated.*

Code region is intended to be atomic, but not enforced during execution.

Solution: Add locks.

```
type Thread struct {
        mutex sync.Mutex
        procInfo        *ProcInfo
}
```

Thread 1:

```
mutex.Lock()
if thrd.procInfo != nil {
        …
        fputs(thrd.procInfo, …)
        …
}
mutex.Unlock()
```

Thread 2:

```
mutex.Lock()
thrd.procInfo = nil
mutex.Unlock()
```

Order-Violation Bugs

Implement example in Go.

Q: What is the problem?

Thread 1: Initialize mThread

Thread 2: Use mThread expecting it to already be initialized

Likely crash with a nil-pointer dereference failure.

Formally: *The desired order between two (groups of) memory accesses is flipped.*
(A should be executed before B, but the order is not enforced during execution…)

Solution: The fix is easy with condition variables.

Summary:
A large fraction of non-deadlock bugs are
atomicity or order violation bugs. (97 %)

32.3 Deadlock Bugs

Deadlock occurs when both conditions below hold:

- T1 hold L1, and is waiting for L2
- T2 hold L2, and is waiting for L1

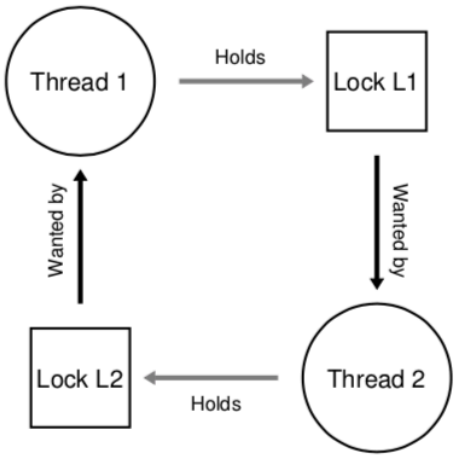


Figure 32.2: **The Deadlock Dependency Graph**

Example: two locks and two threads

Implement code snippet below in Go

Thread 1:	**Thread 2:**
Lock(L1)	Lock(L2)
Lock(L2)	Lock(L1)

Example above does not necessarily lead to deadlock. It can lead to a deadlock.

Q: What can be done to fix the deadlock?

A: Take lock in the same order.

Thread 1:	**Thread 2:**
Lock(L1)	Lock(L1)
Lock(L2)	Lock(L2)

Why do Deadlocks Occur?

One reason: In large code bases, complex dependencies arise between components.

Care must be taken to avoid deadlock:

- When we have naturally occurring circular dependencies.

Encapsulation (modularity)

- We are taught to hide details of implementations
  - To make our software easier to build in a modular way
- Such modularity does not always mesh well with locking.

Example modularity: Java Vector class: vec1.AddAll(vec2)

- Considered *thread safe*
  - Thread 1: vec1.AddAll(vec2)
  - Thread 2: vec2.AddAll(vec1)
  - Taking the lock for both vec1 and vec2
  - Potential deadlock

Conditions for Deadlock

Four conditions **must hold** for a deadlock to occur.

- **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 or more resources (e.g., locks) that are being requested by the next thread in the chain.

If any of these conditions are not met: *deadlock cannot occur*.

Technique: Prevention

- Prevent one of the above conditions from arising!!

Circular Wait:

To avoid circular waiting:

- Always acquire the locks in a **total ordering**, e.g., L1, then L2 and so on.

Total ordering may be difficult to achieve, especially in large systems with many locks.

- **partial ordering** can be useful to avoid deadlocks as well

Both total and partial ordering require carefully designed locking strategies

- Ordering is just a **convention**
- Slopply programmer can easily ignore locking protocol
  - Can cause deadlock

Hold-and-wait:

Acquire all locks at once atomically:

```
Lock(prevention)
Lock(L1)
Lock(L2)
…
// Critical section
…
Unlock(prevention)
```

Q: What is the problem with this approach?

- Requires knowing which locks must be held
- Decreases concurrency: because must hold locks for locks.

No Preemption

Release lock if other thread holds lock we want.

- TryLock(): grab lock L2 if available.
- Otherwise: we can release already held lock L1.

```
top:    L1.Lock()
        if !L2.TryLock() {
                L1.Unlock()
                goto top
        }
```

- Other thread could follow the same protocol: **deadlock free**

Q: What is the problem with this?

- Other thread could follow same protocol
  - Deadlock free
  - New problem arise: **livelock**
  - Two threads could repeatedly attempt this sequence, repeatedly failing to get both locks
  - No progress is made: hence the name livelock
  - Solution: add random delay before looping back and trying again
  - Could work for the Java vector example

Mutual Exclusion

Can we avoid the need for mutual exclusion at all?

Herlihy: we can design various data structures without locks at all.

Lock-free data structures: use powerful HW instructions

Recall TAS, CompareAndSwap() instruction:

```
func CompareAndSwap(address *int, expected, new int) bool {
        if *address == expected {
                *address = new
                return true             // success
        }
        return false                    // failure
}
```

From Go's sync/atomic package:

```
// CompareAndSwapInt64 executes the compare-and-swap operation for an int64 value.
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
```

Atomic increment with CAS:

```
func AtomicIncrement(value *int, amount int) {
        old := *value
        for !CompareAndSwap(value, old, old+amount) {
                old = *value
        }
}
```

Repeatedly tries to update value using CAS instruction.

- No deadlock can arise
- Livelock is still possible

From Go's sync/atomic package:

```
// AddInt64 atomically adds delta to *addr and returns the new value.
func AddInt64(addr *int64, delta int64) (new int64)
```

Linked list insert

```
func (l *List) Insert(key int) {
    newNode := &Node{key: key, next: l.head}
    l.head = newNode
}
```

When called by multiple threads, there is a race condition:

- Reading head and updating head

Linked list insert with lock

- Already saw that we can use locks

```
func (l *List) Insert(key int) {
    l.Lock()
    defer l.Unlock()
    // start of CS
    newNode := &Node{key: key, next: l.head}
    l.head = newNode
    // end of CS
}

// Can allocate before taking the lock
func (l *List) Insert(key int) {
    newNode := &Node{key: key}
    l.Lock()
    newNode.next = l.head
    l.head = newNode
    l.Unlock()
}
```

Linked list insert without lock

- We can perform insert in a lock-free manner using CAS

```
func (l *List) InsertCAS(key int) {
    newNode := &Node{key: key}
    // start of CS
    newNode.next = l.head
    for !CAS2(l.head, *newNode.next, *newNode) {
        newNode.next = l.head
    }
    // end of CS
}
```

- Tries to update next pointer to the current head
- Then tries to swap: newly-created node into position as the new head of list
  - If failure:
    - Because some other thread successfully swapped in another node as head
    - Try again
- Non-trivial to implement some of the other list operations in a lock-free manner.

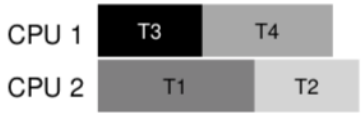
Technique: Deadlock Avoidance via Scheduling

Avoidance require *global knowledge* of which locks various threads might grab during an execution.
Schedule threads to guarantee no deadlock can occur.

Example: Two CPUs. Four threads, two locks.

Lock acquisition demands:

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



These lock demands allow flexible scheduling:

- T3 does grab L2, but cannot cause deadlock
- Since T3 only needs one lock, so cannot cause circular waiting

Lock acquisition demands:

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

Static schedule:

- Conservative approach:
  - T1, T2, and T3 run on same CPU
  - Takes longer to complete
  - Could have been possible to run these tasks concurrently.
  - But fear of deadlock prevents us from doing so.
  - Cost: performance.

Dijkstra's Banker's Algorithm is one example deadlock avoidance algorithm.

However, only useful in very limited environments (embedded systems) where system has full knowledge of

- The entire set of tasks (processes)
- The locks that they need

Deadlock avoidance via scheduling: not widely used technique.

Technique: Detect and Recover

If deadlocks are rare:

- Pragmatic approach: restart process/system

Deadlock detection:

- Run periodically
- Build resource graph
- Check graph for cycles
- If cycle (deadlock):
  - System/process restart

32.4 Summary

Concurrency and deadlock bugs:

- One of the most studied topics in computing

Best solution in practice:

- Be careful
- Develop lock acquisition order
- Prevent deadlock from occurring in the first place

Wait-free approaches

- Becoming more mainstream: used in libraries and critical systems
- Lack generality

Best approach: avoid locks if you can

- MapReduce (from Google)
- Functional programming style
  - No side-effects, no state that is updated