26. Concurrency Bugs

Non-Deadlock Bugs

common, easier to fix. 97% of non-deadlock bugs are either atomicity or order violations

Atomicity-Violation Bugs

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

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

Figure 32.2: **Atomicity Violation (`atomicity.c`)**

: 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)

Solution : lock

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

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

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

Figure 32.3: **Atomicity Violation Fixed (`atomicity_fixed.c`)**

Order Violation Bugs

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

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

Figure 32.4: **Ordering Bug (`ordering.c`)**

: The desired order between groups of memory access is flipped
there is an onder to keep, but it is not enforced during execution.

Solution : enforce ordering (using condition variables)

```
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;
    ...
}
```

Figure 32.5: **Fixing The Ordering Violation (`ordering_fixed.c`)**

Deadlock Bugs

occurs
when a thread is holding a lock(L1) and waiting for another one(L2),
and another thread is holding a lock(L2) and waiting for another one(L1)

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

Figure 32.6: **Simple Deadlock (`deadlock.c`)**

Why Do Deadlocks Occurs?

1. Complex dependencies arise between components
   : 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.
2. nature of encapsulation
   : hide details of implementations. → some seemingly innocuous interfaces almost invite you to deadlock

Conditions for Deadlock

- 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 requrested by the next thread in the chain.

If any of these four conditions are not met, deadlock cannot occur.

Prevention : Circular Wait

the most practical prevention technique

Total Ordering

- provide a total ordering on lock acquisition
- strict ordering ensures that no cyclical wait arises → no deadlock
- in more complex systems, multiple locks
  => total lock ordering may be difficult to achieve (& unnecessary)
  - Why is unnecessary?
    1. Many locks are independent, so global order is irrelevant.
    2. Strict ordering can hurt performance.
    3. There are alternative, more flexible deadlock-avoidance strategies.

Partial Ordering

in linux's memory mapping code, the comment at the top of the source code reveals 10 different groups of lock acquisition orders (ex. "i_mutex before i_mmap_rwsem")

Ordering require careful design of locking strategies and must be constructed with great care.
Ordering is just a convention
: a sloppy programmer can easily ignore the locking protocol and potentially cause deadlock.
Lock ordering requires a deep understanding of the code base, and how various routines are called.

Enforce Lock Ordering By Lock Address

Let's say function do_something acquire 2 locks.
what if one thread call do_something(L1,L2) while another thread call do_something(L2,L1)
=> DeadLock → can use the address of each lock as a way of ordering lock acquisition to avoid deadlock

```
if (m1 > m2) { // grab in high-to-low address order
  pthread_mutex_lock(m1);
  pthread_mutex_lock(m2);
} else {
  pthread_mutex_lock(m2);
  pthread_mutex_lock(m1);
}
// Code assumes that m1 != m2 (not the same lock)
```

Prevention : Hold-and-wait

can be avoided by acquiring all locks at once, atomically (like a lock for locks)

```
pthread_mutex_lock(prevention);   // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

problematic

- encapsulation
  : we need to know exactly which locks must be held, to acquire them ahead of time
- likely to decrease concurrency
  : all locks must be acquired early on at once instead of when they are truly needed.

Prevention : No Preemption

the routine “pthread_mutex_trylock()

- grabs the lock if it is available & return success
- return error code indicating the lock is held

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

problem : livelock

two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. / but progress never made

→ could add a random delay before looping back and trying the entire thing over again

→ decreasing odds of repeated interference among competing threads.

It skirts around the hard parts of using a trylock approach.

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

must make sure to carefully release if the code had acquired some resources along the way when failed.

Prevention : Mutual Exclusion

difficult. The code we wish to run does indeed have critical section

could design various data structures without locks at atll : lock-free

using powerful hardware instructions, you can build data structures in a manner that does not require explicit locking

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

Deadlock Avoidance via Scheduling

Avoidance
requires 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

static scheduling leads to a conservative approach
→ 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.

Only useful in very limited environments
embedded system whre one has full knowledge of the entire set of tasks that must be run and the locks that they need.

can limit concurrency
=> Avoidance of deadlock via scheduling is not a widely-used general-purpose solution

Detect and Recover

Allow deadlocks to occasionally occur, take action after that.
If deadlock is super rare, just rebooting is a good choice.

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.

Don’t Always Do It Perfectly

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.
One will always have to decide which aspects of a system to build well and which to put aside for another day.
The hard part is knowing which to do when

Locks are problematic by their very nature.
Perhaps we should seek to avoid using them unless we truly must.