Common Concurrency Bugs and Deadlocks

What Types Of Bugs Exist?

- Focus on four major open-source applications
  - MySQL, Apache, Mozilla, OpenOffice

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

Bugs In Modern Applications

Non-Deadlock Bugs

- Make up a majority of concurrency bugs
- Two major types of non deadlock bugs
  - Atomicity violation
  - Order violation

Atomicity-Violation Bugs

- The desired **serializability** among multiple memory accesses is *violated*
  - Simple Example found in MySQL
    - Two different threads access the field `proc_info` in the `struct thd`

```
Thread1::
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}

Thread2::
thd->proc_info = NULL;
```

Atomicity-Violation Bugs

- **Solution**: Simply add locks around the shared-variable references

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
pthread_mutex_lock(&lock);
if(thd->proc_info){
    …
    fputs(thd->proc_info , …);
    …
}
pthread_mutex_unlock(&lock);

Thread2::
pthread_mutex_lock(&lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&lock);
```

Order-Violation Bugs

- The desired order between two memory accesses is flipped
  - i.e., **A** should always be executed before **B**, but the order is not enforced during execution
  - **Example**:
    - The code in Thread2 seems to assume that the variable `mThread` has already been *initialized* (and is not `NULL`)

```
Thread1::
void init(){
    mThread = PR_CreateThread(mMain, …);
}

Thread2::
void mMain(…){
    mState = mThread->State
}
```

Order-Violation Bugs

- **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);
    …
}

Thread2::
void mMain(…){
    …
```

Order-Violation Bugs

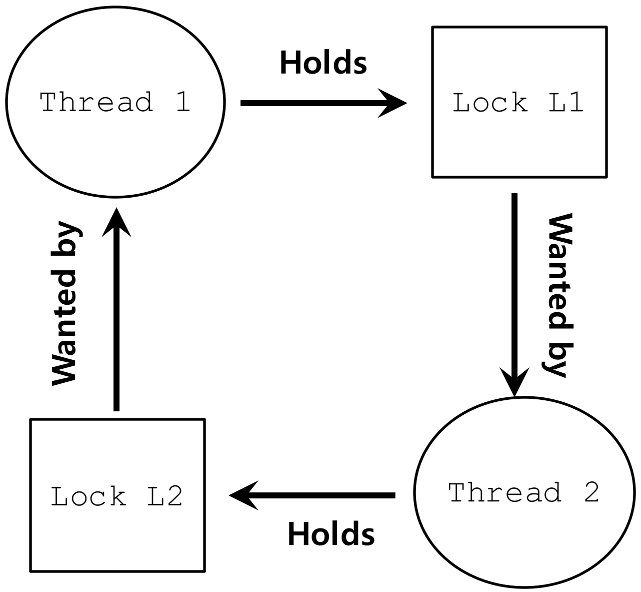
```
    // wait for the thread to be initialized …
    pthread_mutex_lock(&mtLock);
    while(mtInit == 0)
            pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

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

Deadlock Bugs

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

- The presence of a cycle
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`.
  - `Thread2` that holds lock `L2` is waiting for `L1` to be release.



Why Do Deadlocks Occur?

- Reason 1
  - In large code bases, complex dependencies arise between components

- Reason 2
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way
    - Such modularity *does not mesh* well with locking

Why Do Deadlocks Occur?

- **Example**: Java Vector class and the method `AddAll()`

```
Vector v1,v2;
v1.AddAll(v2);
```

- **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 (v1 then v2)
  - If some other thread `calls v2.AddAll(v1)` at nearly the same time → We have the potential for deadlock

Conditional for Deadlock

- Four conditions need to hold for a deadlock to occur

Condition	Description
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 more resources that are being requested by the next thread in the chain

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

Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies
- **Example**
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2

Prevention – Hold-and-wait

- Acquire all locks at once, atomically

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

  - This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition
  - **Problem**
    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time
    - Decrease *concurrency*

Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another
- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol
  - Grab the lock (if it is available)
  - Or, return -1: you should try again later

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
        unlock(L1);
        goto top;
    }
```

Prevention – No Preemption

- livelock
  - Both systems are running through the code sequence *over and over again*
  - Progress is not being made
  - Solution
    - Add **a random delay** before looping back and trying the entire thing over again

Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction
  - You can build data structures in a manner that *does not require* explicit locking

```
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
            *address = new;
            return 1; // success
    }
    return 0;
}
```

Prevention – Mutual Exclusion

- We now wanted to atomically increment a value by a certain amount

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

- Repeatedly tries to update the value to *the new amount* and uses the `compare-and-swap` to do so

- **No lock** is acquired
- **No deadlock** can arise
- **livelock** is still a possibility

Prevention – Mutual Exclusion

- **More complex example**: list insertion

```
void insert(int value){
    node_t * n = malloc(sizeof(node_t));
    assert( n != NULL );
    n->value = value ;
    n->next  = head;
    head     = n;
}
```

- If called by multiple threads at the "*same time*", this code has a race condition

Prevention – Mutual Exclusion

- **Solution**
  - 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 critical section
}
```

  - wait-free manner using the `compare-and-swap` instruction

```
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) == 0);
}
```

Deadlock Avoidance via Scheduling

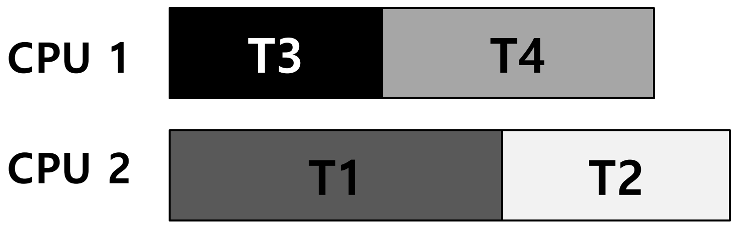
- In some scenarios deadlock avoidance is preferable
  - **Global knowledge** is required
    - Which locks various threads might grab during their execution
    - Subsequently schedules said threads in a way as to guarantee no deadlock can occur

Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads
  - Lock acquisition demands of the threads

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

  - A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise

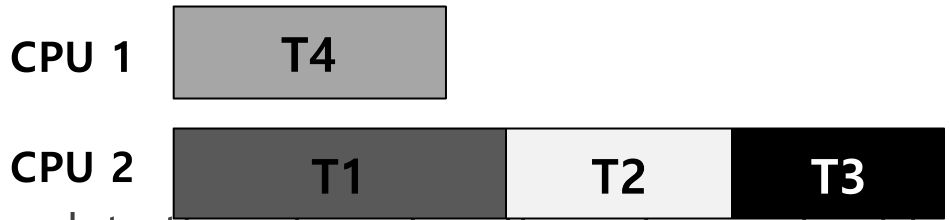


Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

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

  - A possible schedule that guarantees that *no deadlock* could ever occur



    – The total time to complete the jobs is lengthened considerably

Detect and Recover

- Allow deadlock to occasionally occur and then *take some action*
  - **Example**: if an OS froze, you would reboot it

- Many database systems employ *deadlock detection* and *recovery technique*
  - A deadlock detector **runs periodically**
  - Building a **resource graph** and checking it for cycles
  - In deadlock, the system **need to be restarted**