32. Common Concurrency Problems.

Operating System: Three Easy Pieces

Common Concurrency Problems

□ More recent work focuses on studying other types of common concurrency bugs.

  ◆ Take a brief look at some example concurrency problems found in real code bases.

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`.

```
1     Thread1::
2     if(thd->proc_info){
3         …
4         fputs(thd->proc_info , …);
5         …
6     }
7
8     Thread2::
9     thd->proc_info = NULL;
```

Atomicity-Violation Bugs (Cont.)

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

```
1    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3    Thread1::
4    pthread_mutex_lock(&lock);
5    if(thd->proc_info){
6         …
7         fputs(thd->proc_info , …);
8         …
9    }
10   pthread_mutex_unlock(&lock);
11
12   Thread2::
13   pthread_mutex_lock(&lock);
14   thd->proc_info = NULL;
15   pthread_mutex_unlock(&lock);
```

Order-Violation Bugs

□ The desired order between two memory accesses is <u>flipped</u>.

- ◆ 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`).

```
1    Thread1::
2    void init(){
3        mThread = PR_CreateThread(mMain, …);
4    }
5
6    Thread2::
7    void mMain(…){
8        mState = mThread->State
9    }
```

Order-Violation Bugs (Cont.)

- **Solution**: Enforce ordering using condition variables

```
1    pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2    pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
3    int mtInit = 0;
4
5    Thread 1::
6    void init(){
7        …
8        mThread = PR_CreateThread(mMain,…);
9
10       // signal that the thread has been created.
11       pthread_mutex_lock(&mtLock);
12       mtInit = 1;
13       pthread_cond_signal(&mtCond);
14       pthread_mutex_unlock(&mtLock);
15       …
16   }
17
18   Thread2::
19   void mMain(…){
20       …
```

```
21        // wait for the thread to be initialized …
22      pthread_mutex_lock(&mtLock);
23      while(mtInit == 0)
24              pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      …
29 }
```

Deadlock Bugs

```
Thread 1:          Thread 2:

lock(L1);          lock(L2);

lock(L2);          lock(L1);
```
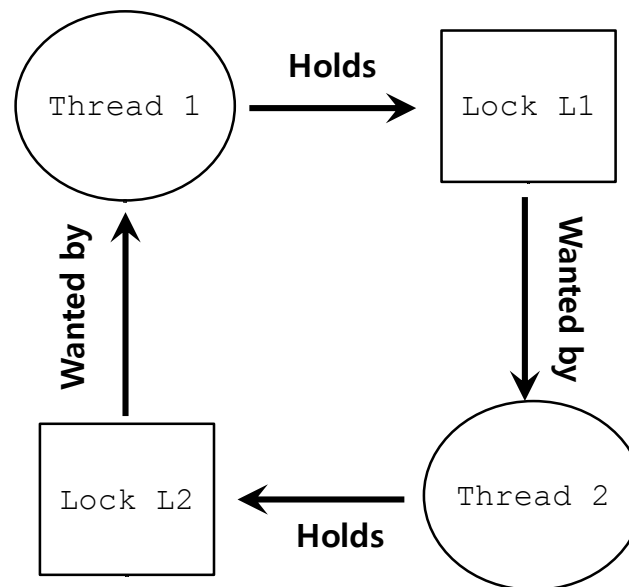
- ◆ The presence of a cycle

  - 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 <u>locking</u>.

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

```
1    Vector v1,v2;
2    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**.

❑ 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.

◻ Acquire all locks at once, atomically.

```
1     lock(prevention);
2     lock(L1);
3     lock(L2);
4     …
5     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*

- **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.

```
1   top:
2       lock(L1);
3       if( tryLock(L2) == -1 ){
4               unlock(L1);
5               goto top;
6       }
```

- livelock

    - Both systems are running through the code sequence *over and over again*.

    - <u>Progress is not being made</u>.

    - Solution:

        - Add **a random delay** before looping back and trying the entire thing over again.

- wait-free

  - Using powerful hardware instruction.

  - You can build data structures in a manner that *does not require* <u>explicit locking</u>.

```
1    int CompareAndSwap(int *address, int expected, int new){
2        if(*address == expected){
3                *address = new;
4                return 1; // success
5        }
6        return 0;
7    }
```

Prevention – Mutual Exclusion (Cont.)

□ We now wanted to atomically increment a value by a certain amount:

```
1    void AtomicIncrement(int *value, int amount){
2        do{
3                int old = *value;
4        }while( CompareAndSwap(value, old, old+amount)==0);
5    }
```

◆ 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.

□ **More complex example**: list insertion

```
1    void insert(int value){
2        node_t * n = malloc(sizeof(node_t));
3        assert( n != NULL );
4        n->value = value ;
5        n->next  = head;
6        head     = n;
7    }
```

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

□ **Solution**:

◆ Surrounding this code with a lock acquire and release.

```
1    void insert(int value){
2        node_t * n = malloc(sizeof(node_t));
3        assert( n != NULL );
4        n->value = value ;
5        lock(listlock); // begin critical section
6        n->next = head;
7        head     = n;
8        unlock(listlock) ;   //end critical section
9    }
```

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

```
1    void insert(int value) {
2        node_t *n = malloc(sizeof(node_t));
3        assert(n != NULL);
4        n->value = value;
5        do {
6                n->next = head;
7        } while (CompareAndSwap(&head, n->next, n));
8    }
```

Deadlock Avoidance via Scheduling

□ In some scenarios deadlock avoidance is preferable.

  ◆ **Global knowledge** is required:

    ○ Which locks various threads might grab during their execution.

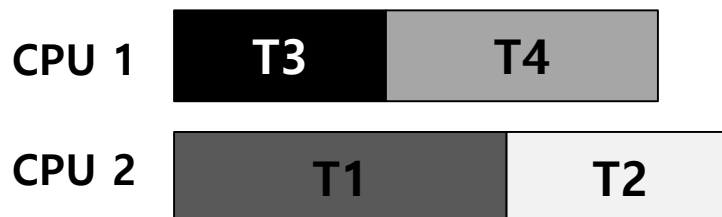   ○ Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

- 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 <u>T1 and T2 are not run at the same time</u>, no deadlock could ever arise.
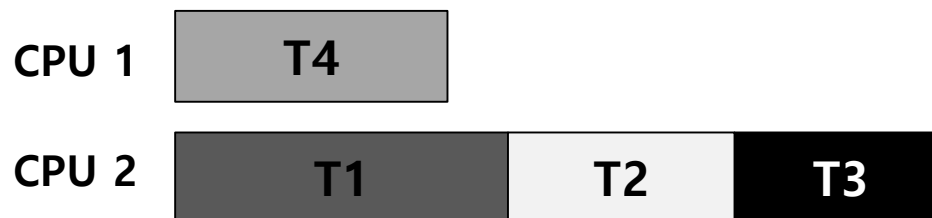
❑ More contention for the same resources

	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.

| CPU 1 | T4 |
| CPU 2 | T1 | T2 | T3 |

○ 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**.

□ Disclaimer: This lecture slide set was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea at University of Wisconsin.