Common Concurrency Problems

Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Common Concurrency Problems

- Much of the early work focused on deadlock
  - Prevention
  - Avoidance
  - Detect and Recover
- More recent work focuses on studying other types of common concurrency bugs (i.e., non-dead lock bugs)

Non-Deadlock Bugs

- Make up a majority of concurrency bugs.

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

Lu et al. (2008), Analysis of concurrent bugs in open source programs

- Two major types of non deadlock bugs:
  - Atomicity violation (원자성 위반)
  - Order violation (순서 위반)

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;
```

The first thread checks if the value is non-NULL and then prints its value

The second thread sets it to 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 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`).

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

Thread 1 must be executed before Thread 2

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
```

Order-Violation Bugs (Cont.)

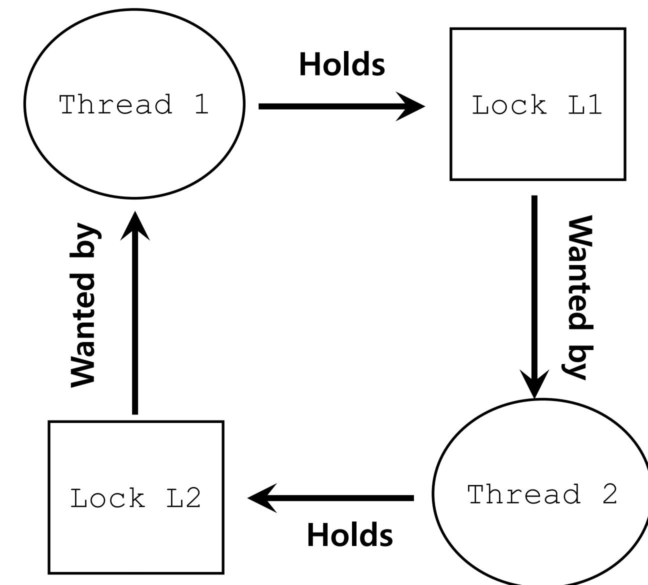
```
18  Thread2::
19  void mMain(…){
21      // wait for the thread to be initialized …
22      pthread_mutex_lock(&mtLock);
23      while(mtInit == 0)
24              pthread_cond_wait(&mtCond, &mtLock);
25      pthread_mutex_unlock(&mtLock);
26
27      mState = mThread->State;
28      …
29  }
```

Deadlock Bugs

- What is the Deadlock?
  - A situation where multiple threads are waiting on each other's resources and are stuck in an infinite wait

```
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.

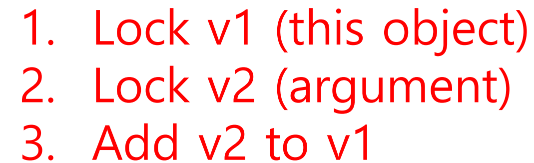
- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way.
    - Such modularity *does not mesh* well with locking.
    - 즉, encapsulated function가 어떤 순서로 lock을 잡는지 알기 어렵다.

Why Do Deadlocks Occur? (Cont.)

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

```
1   vector v1,v2;
2  // thread 1
3  v1.AddAll(v2);

4  // thread 2
5  v2.AddAll (v1) ;
```



1. Lock v1 (this object)
2. Lock v2 (argument)
3. Add v2 to v1

- **Locks** for both the vector being added to (`v1`) and the parameter (`v2`) *need to be acquired.*
  - 각 Vector 객체마다 자기 lock을 가짐!
- Lock V1→V2 (Thread 1) and lock V2→V1 (Thread 2)
  - The deadlock may occur!

Conditions 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. (자원은 한번에 하나의 thread만 사용할 수 있다.)
Hold-and-wait	Threads hold resources allocated to them while waiting for additional resources (이미 자원을 점유한 스레드가, 다른 자원을 기다린다.)
No preemption	Resources cannot be forcibly removed from threads that are holding them. (자원은 외부에서 강제로 회수할 수 없다.)
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. (Thread들 사이에 자원을 기다리는 Cycle이 존재한다.)

- If any of these four conditions are not met, **deadlock cannot occur**.
  (데드락이 발생하기 위해서는 4가지 조건이 동시에 성립)

Prevention – Circular Wait

- Provide a total ordering on lock acquisition (Lock을 요청하는 순서를 정함)
  - 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.

```
if (m1 > m2) {
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
} else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
}
```

락 주소값 기준으로 고정된 순서

Page 9의 예제:

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

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

Prevention – Hold-and-wait

- 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 (*Circular wait* and *Hold-and-wait*):**
  - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
  - Decrease *concurrency (내가 원하는 지점, 즉 Critical section 보다 훨씬 일찍 락이 걸림)*

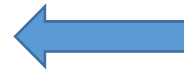
```
1   vector v1,v2;
2  // thread 1
3  v1.AddAll(v2);
4  // thread 2
5  v2.AddAll (v1) ;
```

E.g., AddAll 에 lock이 숨겨져 있음 (Encapsulation)

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.

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

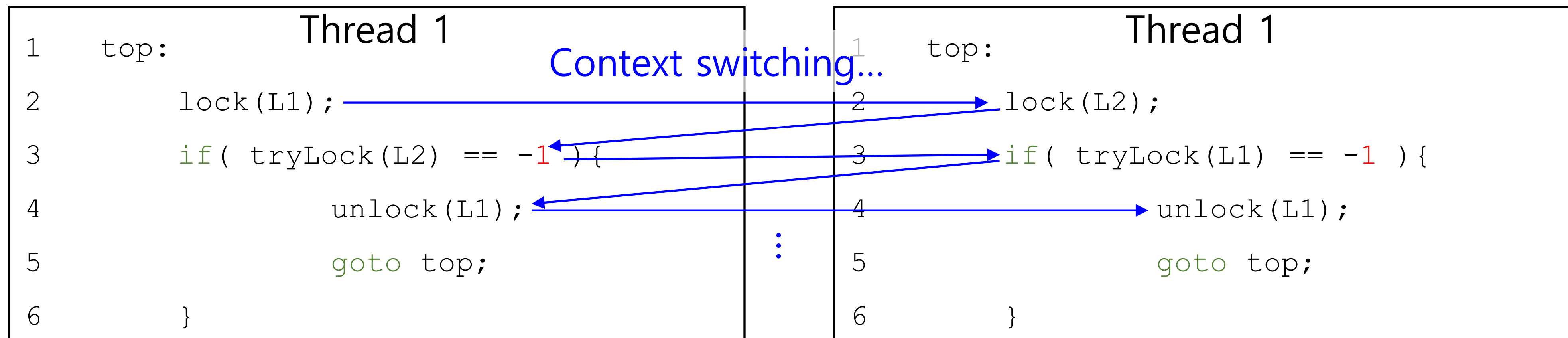


Page 9의 예제:

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

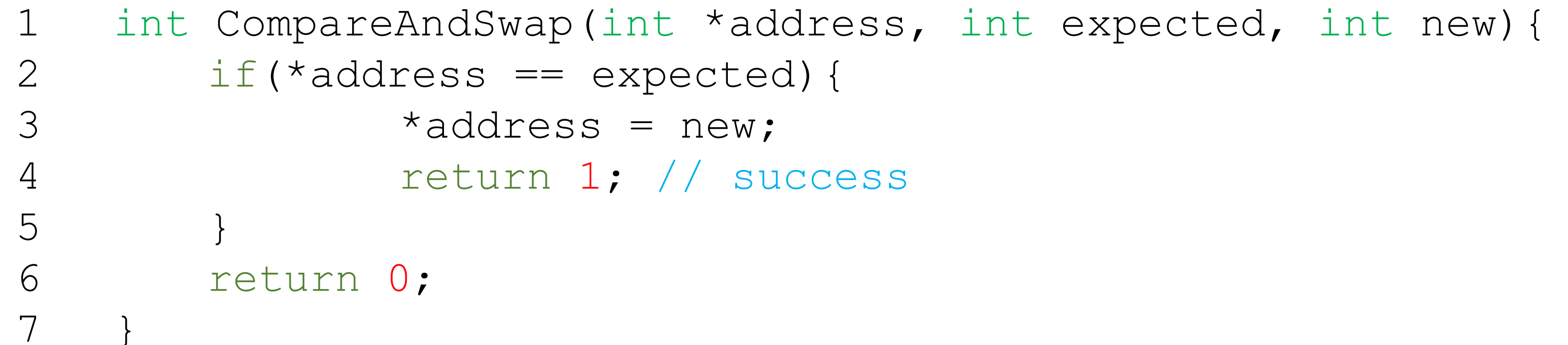
Prevention – No Preemption (Cont.)

- **Problem**: livelock
  - Both systems are running through the code sequence *over and over again* (여러 thread가 동작을 하지만, 진전이 없는 상황)
  - Progress is not being made (타이밍이 절묘하게 맞으면…).
  - 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 (*Compare과 Swap (insert)가 atomic하게 이루어짐*)



```
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   }
```

19_Locks_v2.pdf, page 14

Compare-And-Swap (SPARC)

- Test whether the value at the address(`ptr`) is equal to `expected`.
  - *If so*, update the memory location pointed to by `ptr` with the `new` value.
  - *In either case*, return the actual value at that memory location.

```
1   int CompareAndSwap(int *ptr, int expected, int new) {
2       int actual = *ptr;
3       if (actual == expected)
4               *ptr = new;
5       return actual;
6   }
```

Compare-and-Swap hardware atomic instruction (C-style)

```
1   void lock(lock_t *lock) {
2       while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3               ; // spin
4   }
```

Spin lock with compare-and-swap

14

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.

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.

실패하면, 다시 반복
(여러 스레드가 동시에 실행된다면?)

Prevention – Mutual Exclusion (Cont.)

- **Solution**:
  - Surrounding this code with a lock acquire and release.

기존 방법

```
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

wait-free

```
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)==0);
8    }
```

Deadlock Avoidance via Scheduling

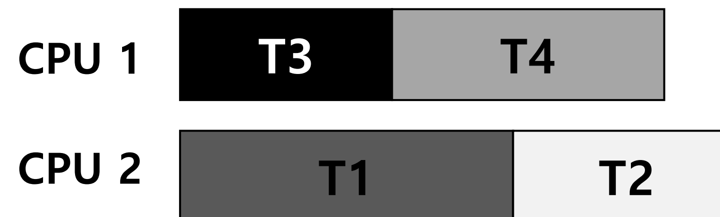
- **Deadlock Avoidance**
  - Get the information about the locks various threads might grab during their execution.
  - schedule the threads in a way <u>to guarantee</u> no deadlock can occur.
- In some scenarios, deadlock avoidance is preferable.
- Problem: Global knowledge is required.

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 <u>lengthened considerably</u>.

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. (cycle이 있으면 deadlock, page9)
  - In deadlock, the system **need to be restarted**.

Summary

- Non-deadlocked bugs
  - Atomicity violation
  - Ordering violation
- Deadlock
  - Enforcing the lock order
  - Lockless mechanism