Common Concurrency Problems

dead-lock.., 이런 문제 들

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

What Types of Bugs Exist

- **Researchers have spent a great deal of time and effort looking into concurrency bugs over many years**
  - What types of concurrency bugs manifest in complex, concurrent programs?
  - A study focuses on four major and important open-source applications

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

  - There were 105 total bugs, most of which were not deadlock (74); remaining 31 were deadlock bugs

- **We now dive into these different classes of bugs (non-deadlock, deadlock) a bit more deeply**

Non-Deadlock Bugs: Atomicity Violation

→ 원자적으로 실행 X

- **Non-deadlock bugs make up a majority of concurrency bugs**
  - Two major types: atomicity violation bugs (원자성 위반) and order violation bugs
- **Consider an example that exposes atomicity violation bug**
  - Two different threads access the field **thd->proc_info**
  - T_1 performs **if** and it is not **NULL** and then interrupted → T_2 sets it **NULL** → T_1 **fputs()** crash

T1 T2.

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

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

crash. interrupt.

```
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);
```

→ 한 순간에 한 스레드만 접근할 수 있도록 제한.

1.
2.
3.

순차적 violation

  - The formal definition of an atomicity violation is that the desired serializability among multiple memory accesses is violated
- **The solution is to simply add locks around the shared-variable references**

→ 간단히 lock를 걸어주면 된다.

Non-Deadlock Bugs: Order Violation Bugs

condition-variable. 순서 보장.

- **Consider a simple example that exposes the order violation**
  - T_2 seems to assume that **`mThread`** has already been initialized
  - T_2 runs immediately once created, **`mThread`** will not be set when it is accessed within **`mMain()`** in T_2 → crash

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

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

parent

child

순차적으로!

```
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;
    ...
}
```

state variable + condition variable

pthread_cond_t mtCond = ...

lock

  - The formal definition of an order violation is that the desired order between two (groups of) memory accesses is flipped
- **The fix to this type of bug is generally to enforce ordering**
  - Using condition variables is an easy and robust way to add the synchronization

Deadlock Bugs

→ 해결하기가 되게 까다로움.

" 프로그래밍을 잘못 짬 → 알고리즘 문제. "

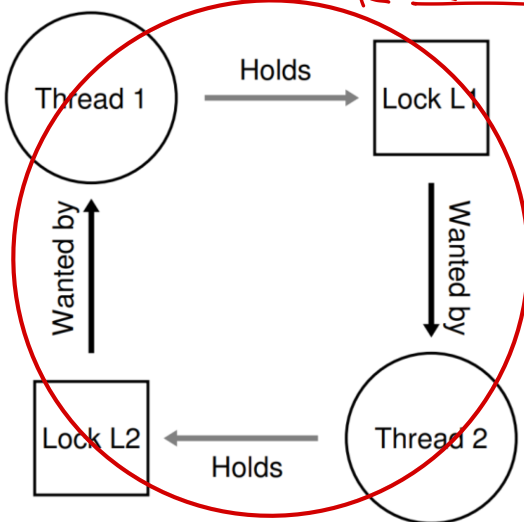
- **A classic problem that arises in many concurrent systems with complex locking protocols is known as deadlock**
  - Deadlock occurs, for example, when T_1 holds a lock L1 and waiting for another one L2; unfortunately, T_2 that holds lock L2 is waiting for L1 to be released

→ 특정 context-switch

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

잡을 수가 없음 (deadlock)

The presence of a cycle in the graph is indicative of the deadlock



하이젠 버그.

- **Why do deadlocks occur?**
  - One reason is that In large code bases, complex dependencies arise between components

    복잡한 종속성.

    → 나혼자 프로그램을 짜는 것이 아니라 여러명이 여러명

  - Another reason is due to the nature of encapsulation, which hides the details of implementation

    캡슐화.

    ↳ abstract 개념은 좋지만 안 보이는 게 많음.

Conditions for Deadlock

- **Four conditions need to hold for a deadlock to occur**

Condition	Description
① **Mutual exclusion** → 없으면 dead-lock이 발생할 이유가 없다	Threads claim exclusive control of resources that they require (e.g. a thread grabs a lock)
② **Hold-and-wait** → 리소스를 잡고. 필요로 하는 상황	Threads hold resources allocated to them (e.g. locks that they already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
③ **No preemption** → 리소스를 preemptive하게 강제로 끌어낼 수가 없음.	Resources (e.g., locks) cannot be forcibly removed from threads that are holding them
④ **Circular wait** → 계속 chaining	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

이중 하나라도 만족하지 않으면 dead-lock이 발생하지 않음

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

- **We first explore techniques to prevent deadlock**
  - The prevention is one approach to handling the deadlock problem

Prevention: Circular Wait & Hold-and-Wait

예방

- **The most straightforward way to prevent the circular wait is to provide a total ordering on lock acquisition**

  순서를 지정해 줌.

  - If there are only two locks in the system (L1 and L2), you can prevent deadlock by always acquiring L1 before L2 수많은 lock을 일일이 순서를 지정해 주기 어려움
  - Such strict ordering ensures that no cyclical wait arises; hence, no deadlock
  - Total lock ordering may be difficult to achieve; thus, a partial ordering can be a useful way to structure lock acquisition so as to avoid deadlock 부분적으로 라도 지정해 주자.

- **The hold-and-wait requirement for deadlock can be avoided by acquiring all locks at once, atomically**
  - By first grabbing the lock **`prevention`**, it guarantees that no thread switch can occur in the midst of lock acquisition and thus deadlock can be avoided

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

L1 lock, L2 lock을 잡는 걸 Atomic하게 하기 위해 또다른 lock을 준다.
= prevention.

  - Note that the solution is problematic for a number of reasons; encapsulation works against us (know the detail locks), likely to decrease concurrency

Prevention: No Preemption

(얻지 못하는 lock을 계속 얻으려 하는 것이 아니라 나중에 다시 시도!)

- **Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble**
  - Many thread libraries provide a more flexible set of interfaces to help avoid this situation, such as **`pthread_mutex_trylock()`**

잡혀 있나 확인

L1 L2

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

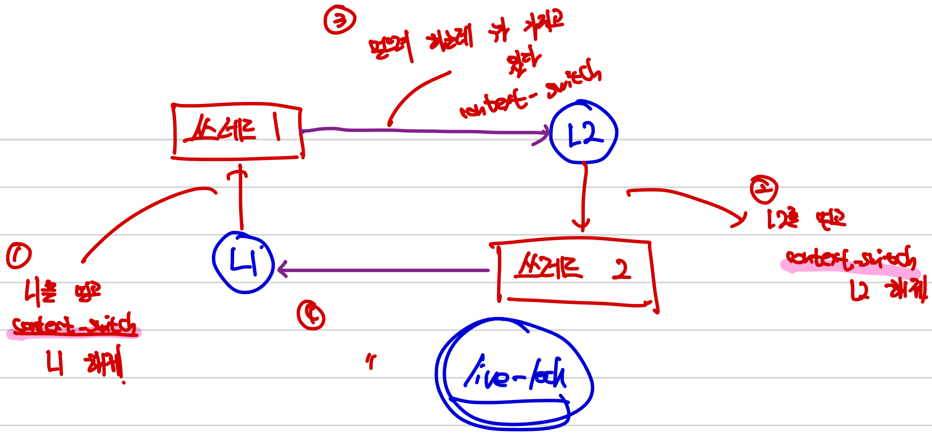
```
top:
  pthread_mutex_lock(L2);
  if (pthread_mutex_trylock(L1) != 0) {
    pthread_mutex_unlock(L2);
    goto top;
  }
```

L2가 잡혀 있으면 L1는 풀어버리자

  - Note that another thread grabbing the locks in other order (L2→L1) still works

- **One new problem does arise, however: livelock**
  - It is possible (though perhaps unlikely) that two threads can both be repeatedly attempting this sequence and repeatedly failing to acquire both locks →동시에 계속 lock 가지려는 상태일 경우
  - This is not deadlock because both threads keep running; but no progress →데드락은 아님.
  - One solution is that one could add a random delay before looping back and trying the entire thing over again

랜덤으로 delay를 준다. = 간섭 가능성을 줄임



③
context-switch
쓰레드 1
L2
L1
쓰레드 2
①
L1을 얻고
context-switch
L1 해제!
②
L2를 얻고
context-switch
L2 해제
live-lock

Prevention: Mutual Exclusion

mutex 자체를 없애버림

- **The final prevention technique would be to avoid the need for mutual exclusion at all**
  - The idea behind these lock-free (and related wait-free) approaches here is simple: using powerful hardware instructions

    atomic instruction
  - This code repeatedly tries to update the value by using `compare-and-swap()`

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

→ 업데이트 하고 return 1 (성공)

→ return 0 (실패)

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

lock
counter++
unlock

→ Atomic하게 진행된다.

→ 전의 값이 바뀌지 않았으면 업데이트.

- **Let's consider a more complex example: list insertion**
  - To avoid a race condition, a lock can be used for the critical section
  - Instead, this insertion can be done lock-free using **`compare-and-swap()`**

→ 좋지만 제한적인 작업만 처리하기 때문에 좋은 방법은 아니다.

new node → node1 → node2

①③ + 원자적으로 수행되어야 함

n->next = head

```
void insert(int value) {
  node_t *n = malloc(sizeof(node_t));
  assert(n != NULL);
  n->value = value;
  pthread_mutex_lock(listlock);     // begin critical section
  n->next  = head;
  head     = n;
  pthread_mutex_unlock(listlock);   // end critical section
}
```

```
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);
}
```

old-value

→ lock 자체를 없애 버렸음 (HW CompareAndSwap을 사용함으로써

Deadlock Avoidance via Scheduling

→ 조건 자체를 피한다. ← 예방하는 것이 아니다.

- **Instead of deadlock prevention, in some scenarios deadlock avoidance is preferable**

  = 서로 같은. 공유자원을 사용하는 스레드끼리 같은 CPU가 실행하도록 스케줄링 하는 방식

  – Avoidance requires to know which locks threads grab during their execution, and subsequently schedules said threads to guarantee no deadlock can occur

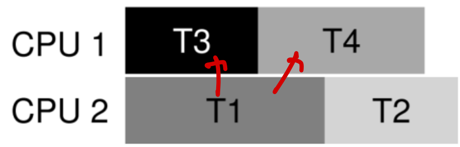
  – Consider two processors and four threads with global knowledge for lock

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

  → 모든 스레드와 무슨 공유 자원을 사용하는지 알아야 함.

  교착상태가 발생할 일이 없음

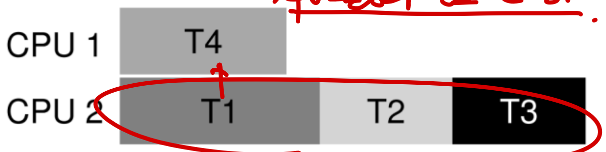
	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no



  2) T1, T2, and T3 all need to grab both locks L1 and L2 at some point during their execution → T1, T2, and T3 are all run on the same processor

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

→ CPU 활용도가 별로 안 좋게됨.



→ exclusive 하게

Thus, the total time to complete the jobs is lengthened considerably

좀 전체적 지연되고 → dead-lock이 절대 발생하지 않는다.