19. Common Concurrency Problems

동기화 문제를 어떻게 해결할 것인가?

- 현대의 application에는 동기화와 관련하여 다양한 bug가 존재

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

▼ Non-deadlock bugs

▼ atomicity-violation bugs

atomic하게 실행해야 하는 것을 지키지 x

⇒ 동시에 memory access하는 것은 막아야 함

(code 영역은 atomic하게 여겨지지만 실행 도중에는 지켜지지 않음)

- example(MySQL)

```
Thread 1:
if (thd->proc_info) {
  ... ① null 값이 아님을 확인 후 if문 진입
  fputs(thd->proc_info, ...); ④ null pointer 참조 error ⇒ 비정상 종료
  ...                          ↳ pointer가 null 값
Thread 2: ③ null 값으로 만들어줌
thd->proc_info = NULL;
```

② context switch 발생

⑤ context switch 발생

- solution → mutex 활용

```
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);
```

② 번에서 context switch 하더라도 이미 lock을 acquire했기에 thread 2는 lock을 얻지 X

▼ Order-violation bugs

memory 접근 순서를 지켜야 하는데 지키지 않음

ex) A→B 순서여야 하는데 B→A 순서로 진행됨.

- example(Mozila)

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

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

mThread = PR_CreateThread → thread 생성 함수
thread의 여러 정보를 구조체로 가짐
mState = mThread->State; → ③ 아직 ① 실행이 끝나지 않아서 mThread return 값이 들어오지 X → null-pointer 참조 error

- solution → condition variable 사용 (myThread 생성 완료 조건이 만족되어야 다음 실행)

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t mtCond = PTHREAD_COND_INITIALIZER;
int mtInit = 0; → state variable

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) ↓ thread 1의 signal 올 때까지 wait (mtInit == 1 까지)
  //mtInit이 1로 변경될 때까지 wait
    pthread_cond_wait(&mtCond, &mtLock);
  pthread_mutex_unlock(&mtLock);
  mState = mThread->State;
  ...
}
```

항상 valid

▼ Deadlock bugs

서로 남이 가지고 있는 것을 기다릴 때 발생

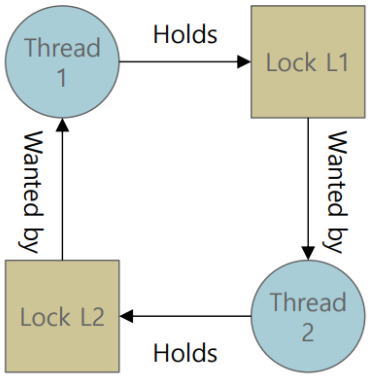
→ **circular dependencies**

- Circular dependencies

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

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

↳서로 남이 가지고 있는 것을 기다림



Resource-allocation graph

▼ why do deadlock occur?

- code base가 커지면 dependency가 복잡해져서 사람이 파악하기 어려움
  - example(virtual memory system, vms) ⇒ 서로서로 접근 가능해야 함.
    - vms : disk로부터 한 block의 page를 위해 file system에 접근할 필요가 있음
      - VMS → FS ⇒ page fault. FS에 어떤 정보를 넘겨줘야 함. 또한, 그 정보가 FS에 있음.
    - file system : block을 읽기 위해 memory의 page를 꽤 자주 필요로 하기에 vms에 접근할 수 있어야 함
- nature of encapsulation (캡슐화)
  - example(java vector class)
    - v1에 추가된 vector와 acquire될 필요가 있는 v2 모두 찾을 때 발생

```
Vector v1, v2;

Thread 1:
v1.AddAll(v2);

Thread 2:
v2.AddAll(v1);
```

circular dependency

↳서로서로 자원 꺼내오려고 함.

▼ Conditions for Deadlock

4개 중에 하나만 만족이 되어도 deadlock 발생 x

① Mutual exclusion

② Hold-and-wait

- thread가 lock을 얻은 채로 또 다른 mutex lock을 받으려 할 때

③ No preemption
(ex. lock)
- 자원을 고정적으로 가지고 있는 thread → 강제로 제거할 수 없음 ⇒ lock을 강제로 뺏어올 수 X

④ Circular wait
- circular chain → 서로 서로 lock과 thread가 하나의 자원을 필요로 함

▼ Deadlock Prevention

1. circular wait
   - total ordering on lock acquisition ⇒ lock을 얻는 순서를 정의
     - 항상 순서대로 lock을 얻도록!
     - partial ordering ↓ lock 순서를 정의해놓음 ⇒ 이 순서대로 항상 lock
       - ex) lock ordering in mm/filemap.c

```
->i_mmap_rwsem              (truncate_pagecache)
  ->private_lock            (__free_pte->__set_page_dirty_buffers)
    ->swap_lock             (exclusive_swap_page, others)
      ->i_pages lock

->i_mutex
  ->i_mmap_rwsem            (truncate->unmap_mapping_range)

->mmap_sem
  ->i_mmap_rwsem
    ->page_table_lock or pte_lock   (various, mainly in memory.c)
      ->i_pages lock        (arch-dependent flush_dcache_mmap_lock)
```

2. Hold-and-wait
   - 한 번에 자동적으로 모든 lock을 얻음으로써 막을 수 있음

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

필요한 lock을 한 번에 얻음 → critical section이 너무 커질 수 있음 ☆

3. No preemption
   - *pthread_mutex_trylock()* ⇒ 스스로 쥐고 있는 lock을 놓도록 하는 operation
     - 유효한 lock을 얻은 뒤 return success
     - ***or*** lock을 이미 얻었다면 return error code

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

→ 돌아가기 전에 얻은 Lock 모두 release

Code 실행은 되는데 상태는 제자리

- **Livelock** : 여러 thread가 모두 lock을 얻지 못 한 채로 계속 돌아감
  - 해결책 : looping back 이전에 random delay 추가
- 만약 code가 가지고 있던 자원이 있다면 top으로 돌아가기 전에 모두 release해야 함

4. Mutual exclusion

- **Lock-free** approaches : race condition은 만들지 않으면서 lock 사용하지 x

Keyword code

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

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

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

→ thread safe

- insert ⇒ mutex_lock 사용 ver

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

- insert ⇒ compare&swap 사용

```
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);
}
```

같아지면 while문 빠져나옴
⇒ Busy Waiting

▼ Deadlock Avoidance

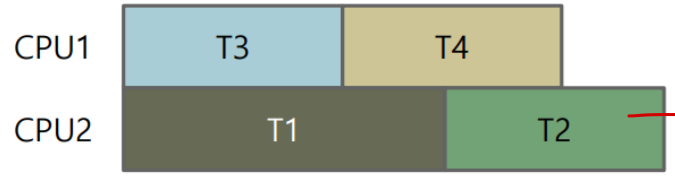
- via scheduling → scheduling approach

	T1	T2	T3	T4
L1 Lock1	Yes	Yes	No	No
L2 Lock2	Yes	Yes	Yes	No

↳ Lock을 아예 안 거는 놈.
↳ 별걱다 완화됨.

It is OK for (T3 and T1) or (T3 and T2) to overlap (deadlock 없이 실행)

overlapping되어도 괜찮은지 실행 순서나 여러 조건들을 고려해야 함 ⇒ 복잡하고 어려움!!



L1, L2 모두 필요하니 별걱정 들어가지 않도록

↳ embedded system에서 사용하는 메커니

▼ Detect and Recover

restart?,..