操作系统实验文档

Lab 13 - 同步 3 Synchronization 3

f2b6767c

Built by github-actions, at 2025-05-13 11:48:26+08:00

1. 每周实验

1.0.1 1.3 Week 13 - 同步 3 Synchronization 3

1.3.1 实验目的

1. 掌握并发 Bug 出现的原因和解决办法
2. 掌握死锁的必要条件
3. 掌握用户模式上锁的方式

1.3.2 并发 Bug

为了保护资源，我们需要进行上锁，但是上锁是个非常复杂且精细的操作，所以我们可能出现因为上锁的顺序不对导致 **死锁** 的情况，也可能出现因为上锁方式错了导致 **data race** 依然存在的情况。我们称它们为并发 Bug 。

DEADLOCK

死锁问题通常分为两种：AA-deadlock 和 ABBA-deadlock。

AA-deadlock

AA-deadlock 表示你在对一把已经获得了的锁重新上锁：

```
acquire(&lk);
assert(holding(&lk));
// ...

    // maybe in interrupt handler or other methods
    acquire(&lk);
    // <-- deadlock here.
```

尽管我们看起来不会犯这种错误，但是真实的系统往往是很复杂的，函数的控制流可能不会那么的显然。

想象你正在写函数 `C()`，已经存在函数 `B` 和函数 `A`，而 `B` 调用 `A` 时会在 `A` 中上锁和解锁。 而 `C` 可能也需要用到同一把锁，所以你会在 `C` 中也上锁解锁。而你发现你会需要复用 `A` 的功能，于是你在持有锁的情况下调用了 `A`。

```
void A() {
    acquire(&lk);
    // ...
    release(&lk);
}
```

```
void B() {
    // ...
    A();
    // ...
}

void C() {
    acquire(&lk);
    A();
    release(&lk);
}
```

当控制流从 `B` 进入 `A` 时是不会死锁的，而从 `C` 进入 `A` 时会死锁。

以及想象另一种情况，你 debug 找到了一个并发 bug；为了解决它，你在 `A` 中加入了上锁解锁的代码；你可能认为只有 `B` 会调用 `A`，而实际上存在 `C -> A` 这一条调用链，而这条调用链里面你在 `C` 中上了锁。

如何解决?

xv6 中，我们使用防御性编程来避免这种问题：

1. 在 `acquire` 上锁时，检查当前 CPU 是否持有着这把锁。如果是，则是 AA 型死锁，使用 `panic("already acquired by")` 中断内核执行。

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
void acquire(spinlock_t *lk)
{
    uint64 ra = r_ra();
    push_off(); // disable interrupts to avoid deadlock.
    if (holding(lk))
        panic("already acquired by %p, now %p", lk->where, ra);

    // ...
}
```

2. 对于一些常用的函数，我们在入口处 `assert(holding(&lk))` 来断言进入函数时我们持有这把锁。

```
// vm.c

pte_t *walk(struct mm *mm, uint64 va, int alloc) {
    assert(holding(&mm->lock));
}

int mm_mappages(struct vma *vma) {
    // ...

    assert(holding(&vma->owner->lock));
}
```

ABBA-deadlock

ABBA 型死锁则是：线程 1 以 `A -> B` 的顺序上锁，线程2 以 `B -> A` 的顺序上锁。 在某个最坏情况下，线程1和2分别得到了A和B，但是它们又各自在等待B和A，而这两把锁恰好在别人手上。

还有一些更加复杂的情况，如：

- 线程 1 上锁顺序：`A -> B`
- 线程 2: `B -> C`
- 线程 3: `C -> A`

我们可以总结出死锁产生的 **必要条件**，将锁（可以推广为“共享资源”）视为一个球：

1. Mutual Exclusion: 一个人拿到的完整的球，不可能两个人同时拿到一部分球。
2. Hold and wait：持有球的情况下等待额外的球
3. No-Preemption：不能抢别人手里的球
4. Circular wait：形成循环等待的关系

（推荐阅读 https://dl.acm.org/doi/10.1145/356586.356588）

> 论文原文： This deadlock situation has arisen only because all of the following general conditions were operative: 1) Tasks claim exclusive control of the resources they require ("mutual exclusion" condition). 2) Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition). 3) Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition). 4) A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

既然说这是 **必要条件**，那么我们只需要打破任何一个就可以避免死锁问题了。在大型系统中，显然最后一个是最容易打破的。

上锁顺序可以被视为一个有向边，而全局的上锁顺序则构成了一张图(dependency graph)，如果这个图上存在环，则表明有死锁问题。

如果避免环：我们可以给锁编号(Lock Ordering)，强制按照从小到大的顺序上锁。

此外，我们也可以动态检测死锁。

DATA RACE

关于数据竞争的并发 bug，主要是以下两种：

1. 上错了锁

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { spin_lock(&B); sum++; spin_unlock(&B); }
```

1. 忘记上锁

```
void T_1() { spin_lock(&A); sum++; spin_unlock(&A); }
void T_2() { sum++; }
```

1.3.3 用户模式下的锁

THE LOST WAKE-UP PROBLEM

在之前的课程中，我们介绍并发和同步的代码均是在内核模式中介绍的。那么，在用户模式下的互斥与同步是怎么实现的?

> **共享内存**
>
> 在本小章节的介绍都是基于共享内存的模型，在 Linux 上，它们可以使用 `mmap(2)` 和 `shm(2)` 创建。
>
> 在 xv6 上，我们的进程模型尚未支持共享内存，但是其核心思想是相同的。

> **用户模式和内核的区别**
>
> 当我们讨论并发时，用户模式（usermode）和内核模式（kernel）最显著的区别是：内核可以屏蔽中断，而用户程序不可以。

在用户模式下，我们依然可以使用原子指令来实现 `spinlock` 自旋锁。但是，对于 `sleeplock` 来说，事情会变得更加复杂。

`sleeplock` 要求在抢不到锁时睡眠进程，并在锁被释放后唤醒进程。这样的话，我们就需要系统调用来做“睡眠”和“唤醒”这两件事情，因为只有内核才能修改某个进程的状态。

```
void sleeplock_acquire(sleeplock_t *lk) {
retry:
    if (__sync_lock_test_and_set(&lk->locked, 1) == 0) {
        // we succeed in 0-to-1 transition, aka, we get the lock.
        return;
    } else {
        // we fail to get the lock, sleep myself.
        int mypid = getpid();
```

```
        // append myself to the waiter queue.
        acquire(&lk->waiters_lock);
        append_waiter(lk, mypid);
        release(&lk->waiters_lock);

        // <-- this process may be wakeup here.

        syscall_sleep();

        // <-- then we will never be woken-up.

        goto retry;
    }
}

void sleeplock_release(sleeplock_t *lk) {

    // A.
    __sync_lock_release(&lk->locked);

    // B.
    // find the next one to wakeup.
    acquire(&lk->waiters_lock);
    int nextone = pop_waiter(lk);
    release(&lk->waiters_lock);

    // C.
    syscall_wakeup(nextone);
}
```

然后，我们就会陷入和 Sync 1 课上讲的 "The Lost Wake-Up Problem" 一样的问题，`syscall_sleep` 可能会导致唤醒信号丢失。

最简单的正确方法即是将所有加锁和解锁的操作都放置到内核处理。但是，系统调用是一种相当耗时的操作，所以我们期望将 fast-path 留在用户态。对于 `sleeplock` 而言，fast-path 就是没有人争抢锁的情况下，只需要使用一条原子指令标记当前锁被占有。

FUTEX

> **Note**
>
> 请在 Linux 环境中执行 man 7 futex 和 man 2 futex 查看其相关文档

futex 是 Linux 内核提供的一个 syscall，用于实现 fast user-space mutexes，其定义如下：

```
long syscall(SYS_futex,
    uint32_t *uaddr, int futex_op, uint32_t val,
    const struct timespec *timeout,   /* or: uint32_t val2 */
    uint32_t *uaddr2, uint32_t val3);
```

它接受一个虚拟地址 uaddr 和用户的预期值 val，以及 futex 的操作 futex_op。 futex 会通过虚拟地址背后的物理地址来区分不同的 futex 对象。所以，对于跨进程的 `shm` 而言，只要它们各自的虚拟地址是同一个物理地址，那它们就可以通过 futex 实现同步。

最重要的两个op是：

1. `FUTEX_WAIT`：如果 `uaddr` 的值与 `val` 中一致，则陷入睡眠，它将等待一个 `FUTEX_WAKE` 操作将其唤醒。如果内核检测到 `uaddr` 的值与用户预期的 `val` 不一致（这说明用户调用 `futex` syscall 的背景发生了变化），那它会立刻从系统调用中返回 `EAGAIN`，而不陷入睡眠。（这就避免了 lost-wakeup 问题）

   内核也会使用原子指令对该虚拟地址背后的物理地址进行访问，确保原子性。

   This operation tests that the value at the futex word pointed to by the address `uaddr` still contains the expected value `val`, and if so, then sleeps waiting for a `FUTEX_WAKE` operation on the futex word.

   If the thread starts to sleep, it is considered a waiter on this futex word. **If the futex value does not match val, then the call fails immediately with the error EAGAIN.**

   The purpose of the comparison with the expected value is to prevent lost wake-ups. If another thread changed the value of the futex word after the calling thread decided to block based on the prior value, and if the other thread executed a `FUTEX_WAKE` operation after the value change and before this `FUTEX_WAIT` operation, then the calling thread will observe the value change and will not start to sleep.

2. `FUTEX_WAKE`： 唤醒在 `uaddr` 上等待的所有 waiter。

   This operation wakes at most `val` of the waiters that are waiting (e.g., inside `FUTEX_WAIT`) on the futex word at the addresslevellock 的。 `uaddr`. Most commonly, val is specified as either 1 (wake up a single waiter) or `INT_MAX` (wake up all waiters).

PTHREAD_MUTEX_T

`pthread_mutex_t` 是 libc 里面的 pthread 库提供的，它使用futex这个syscall来实现 **mutex** 的功能。

我们可以一探究竟 `pthread_mutex_t` 是如何实现的：

glibc 源代码

glibc 源代码非常复杂，它为了性能进行了高度优化。

这一部分代码位于 `nptl/lowlevellock.c` 以及 `sysdeps/nptl/lowlevellock.h`。

`pthread_mutex_t` 的基础功能是依赖于 `lll` lowlevellock

`pthread_mutex_t` 中有一个 `uint32_t`，表示锁的状态：0 表示未上锁，1 表示上锁了但是没有 waiter，`>1` 表示上锁但是可能存在 waiter。

```
void lll_lock(uint32_t* futex) {
    // try to make futex transit from 0 (UNLOCKED) to 1 (LOCKED).
    if (atomic_compare_and_exchange_bool_acq(futex, 1, 0)) {
        // if cmpxhg fails:

        // try to exchange 2 into `*futex`, return the original value.
        while (atomic_exchange_acquire (futex, 2) != 0) {

            // if old value is not `UNLOCKED`
            syscall_futex(futex, FUTEX_WAIT, 2); /* Wait if *futex == 2.  */

            // if syscall returns, *futex is not 2 or someone wakes me up.
        }

        // when we get there, old *futex is 0, now 2 (LOCKED).
    }
    // succeed, LOCKED (*futex is 1 or 2)
}

void lll_unlock(uint32_t* futex) {
    // exchange 0 UNLOCKED into futex
    int __oldval = atomic_exchange_release (futex, 0);

    if (__oldval > 1) {
        // wake up one waiter
        syscall_futex(futex, FUTEX_WAKE, 1);
    }
}
```

上锁步骤：

1. 首先尝试将 futex: `0->1`，如果失败了，则说明有其他线程占用着锁。
2. 随后，死循环地尝试将 futex: `0->2`，如果成功，则表明上锁成功。
3. 否则，在 `futex` 上通过系统调用 `futex_wait` 等待。当从 syscall 中退出时，重试第 2 步。

解锁步骤：

1. 将 0 写入 futex，如果旧值大于1，则使用 `futex` syscall 唤醒一个 waiter。

你可以试着证明这个锁永远不会 lost wakeup。注意，在用户模式下，每一步执行都是被中断、与其他函数执行步骤交错的。

例如以下例子描述了 T1 T2 竞争锁的流程图：

```
        T1                  T2
        |                   |
  lock (0->1, ok)           |
    (acquired)              |
        |              lock (0->1, fail)
unlock(store 0, see 1)      |
        |              lock (0->2, ok)
        |                   |
    (released)          (acquired)


        T1                  T2
        |                   |
  lock (0->1, ok)           |
    (acquired)         lock (0->1, fail)
        |              lock (0->2, fail)
        |                futex_wait()
unlock(store 0, see 2)      |
    futex_wake()        (woken-up)
        |              lock (0->2, ok)
    (released)              |
                          (acquired)


        T1                  T2
        |                   |
  lock (0->1, ok)           |
    (acquired)              |
        |              lock (0->1, fail)
        |              lock (0->2, fail)
unlock(store 0, see 2)      |
        |               futex_wait()
    futex_wake()     (EAGAIN, kernel sees 0)
        |                   |
    (released)         lock (0->2, ok)
                            |
                          (acquired)
```

你可以试着补全如下的图：

```
        T1                  T2                      T3
        |                   |                       |
  lock (0->1, ok)           |                 lock (0->1, fail)
    (acquired)              |                 lock (0->2, fail)
        |                   |                    futex_wait()
        |              lock (0->1, fail)            |
        |              lock (0->2, fail)            |
        |                   |                       |
unlock(store 0, see 2)      |                       |
        |               futex_wait()                |
    futex_wake()   (EAGAIN, kernel sees 0)          |
        |                   |                   (woken-up)
    (released)
```

1.3.4 Sync 1 & 2 Lab 练习解析

上课讲。

Built by github-actions, at 2025-05-13 11:48:26+08:00