并发 Bug 的应对

我们在编写并发程序时，难免会遇到死锁、数据竞争、原子性/顺序违反等类型的并发 bugs。即便我们知道它们的定义和触发条件，直接在编程时消灭它们依然是十分困难的。

以数据竞争为例，它的定义貌似简单：两个线程同时访问同一内存地址，并且至少有一个是写。但“访问内存”则可能出其不意——例如 ret 指令和栈上数据的修改产生的数据竞争。

那么，我们应该如何应对这些并发 bugs？

目录

如何应对死锁

回顾：死锁产生的必要条件

System deadlocks (1971)：死锁产生的四个必要条件

- 用“资源”来描述
  - 状态机视角：就是“当前状态下持有的锁（校园卡/球）”

---

1. Mutual-exclusion - 一张校园卡只能被一个人拥有
2. Wait-for - 一个人等其他校园卡时，不会释放已有的校园卡
3. No-preemption - 不能抢夺他人的校园卡
4. Circular-chain - 形成校园卡的循环等待关系

应对死锁：死锁产生的必要条件（cont'd）

站着说话不腰疼的教科书：

- “理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。所以，在系统设计、进程调度等方面注意如何不让这四个必要条件成立，如何确定资源的合理分配算法，避免进程永久占据系统资源。此外，也要防止进程在处于等待状态的情况下占用资源。因此，对资源的分配要给予合理的规划。”

不能称为是一个合理的 argument

- 对于玩具系统/模型
  - 我们可以直接证明系统是 deadlock-free 的
- 对于真正的复杂系统
  - Bullshit

如何在实际系统中避免死锁?

四个条件中最容易达成的

- 避免循环等待

Lock ordering

- 任意时刻系统中的锁都是有限的
- 严格按照固定的顺序获得所有锁（Lock Ordering），就可以消灭循环等待
  - “在任意时刻获得“最靠后”锁的线程总是可以继续执行”
- 例子：修复哲学家吃饭问题

```
1 #include "thread.h"
```

```c
#include "thread-sync.h"

#define N 5

sem_t avail[N];

void Tphilosopher(int id) {
  int lhs = (id + N - 1) % N;
  int rhs = id % N;

  // Enforce lock ordering
  if (lhs > rhs) {
    int tmp = lhs;
    lhs = rhs;
    rhs = tmp;
  }

  while (1) {
    P(&avail[lhs]);
    printf("+ %d by T%d\n", lhs, id);
    P(&avail[rhs]);
    printf("+ %d by T%d\n", rhs, id);

    // Eat

    printf("- %d by T%d\n", lhs, id);
    printf("- %d by T%d\n", rhs, id);
    V(&avail[lhs]);
    V(&avail[rhs]);
  }
}

int main() {
  for (int i = 0; i < N; i++) {
    SEM_INIT(&avail[i], 1);
  }
  for (int i = 0; i < N; i++) {
    create(Tphilosopher);
  }
}
```

Lock Ordering: 应用 (Linux Kernel: rmap.c)

```
/*
 * Lock ordering in mm:
 *
 * inode->i_mutex       (while writing or truncating, not reading or faulting)
 *   mm->mmap_lock
 *     page->flags PG_locked (lock_page)   * (see huegtlbfs below)
 *       hugetlbfs_i_mmap_rwsem_key (in huge_pmd_share)
 *         mapping->i_mmap_rwsem
 *           hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
 *           anon_vma->rwsem
 *             mm->page_table_lock or pte_lock
 *               swap_lock (in swap_duplicate, swap_info_get)
 *                 mmlist_lock (in mmput, drain_mmlist and others)
 *                 mapping->private_lock (in __set_page_dirty_buffers)
 *                   lock_page_memcg move_lock (in __set_page_dirty_buffers)
 *                     i_pages lock (widely used)
 *                       lruvec->lru_lock (in lock_page_lruvec_irq)
 *                 inode->i_lock (in set_page_dirty's __mark_inode_dirty)
 *                 bdi.wb->list_lock (in set_page_dirty's __mark_inode_dirty)
 *                   sb_lock (within inode_lock in fs/fs-writeback.c)
 *                   i_pages lock (widely used, in set_page_dirty,
 *                             in arch-dependent flush_dcache_mmap_lock,
 *                             within bdi.wb->list_lock in __sync_single_inode)
 *
 * anon_vma->rwsem,mapping->i_mutex      (memory_failure, collect_procs_anon)
 *   ->tasklist_lock
 *     pte map lock
 *
 * * hugetlbfs PageHuge() pages take locks in this order:
 *         mapping->i_mmap_rwsem
 *           hugetlb_fault_mutex (hugetlbfs specific page fault mutex)
 *             page->flags PG_locked (lock_page)
 */
```

Emmm……

Textbooks will tell you that if you always lock in the same order, you will never get this kind of deadlock. *Practice will tell you that this approach doesn't scale*: when I create a new lock, I don't understand enough of the kernel to figure out where in the 5000 lock hierarchy it will fit.

The best locks are encapsulated: they *never get exposed in headers*, and are *never held around calls to non-trivial functions outside the same file*. You can read through this code and see that it will never deadlock, because it never tries to grab another lock while it has that one. People using your code don't even need to know you are using a lock.

— *Unreliable Guide to Locking* by Rusty Russell

- 最终犯错的还是人

即便是最容易发现、最容易预防的死锁类 bug，在实际的复杂系统中，想要使程序员能够正确遵守编程规范，也是十分困难的。因此，对于复杂的系统，我们必须总是假设程序员会花式犯错，最终才能得到高质量的系统。

Bug 的本质和防御性编程

回顾：调试理论

> 程序 = 物理世界过程在信息世界中的投影

Bug = 违反程序员对“物理世界”的假设和约束

- Bug 违反了程序的 specification
  - 该发生的必须发生
  - 不该发生的不能发生
- Fault → Error → Failure

编程语言与 Bugs

编译器/编程语言

- 只管“翻译”代码，不管和实际需求（规约）是否匹配
  - “山寨支付宝”中的余额 `balance`
    - 正常人看到 0 → 18446744073709551516 都认为“这件事不对”（“balance”自带 no-underflow 的含义）

怎么才能编写出“正确”（符合 specification）的程序?

- 证明：Annotation verifier (Dafny), Refinement types
- 推测：Specification mining (Daikon)
- 构造：Program sketching
- 编程语言的历史和未来
  - 机器语言 → 汇编语言 → 高级语言 → 自然编程语言

回到现实

今天（被迫）的解决方法

- 虽然不太愿意承认，但始终假设自己的代码是错的
- （因为机器永远是对的）

然后呢?

- 首先，做好测试
- 检查哪里错了
- 再检查哪里错了
- 再再检查哪里错了

- “防御性编程”
- 把任何你认为可能“不对”的情况都检查一遍

防御性编程：实践

> 把程序需要满足的条件用 assert 表达出来。

及早检查、及早报告、及早修复

- Peterson 算法中的临界区计数器
  - `assert(nest == 1);`
- 二叉树的旋转
  - `assert(p->parent->left == p || p->parent->right == p);`
- AA-Deadlock 的检查
  - `if (holding(&lk)) panic();`
  - xv6 spinlock 实现示例

防御性编程和规约给我们的启发

你知道很多变量的含义

```
#define CHECK_INT(x, cond) \
  ({ panic_on(!((x) cond), "int check fail: " #x " " #cond); })
#define CHECK_HEAP(ptr) \
  ({ panic_on(!IN_RANGE((ptr), heap)); })
```

变量有“typed annotation”

- `CHECK_INT(waitlist->count, >= 0);`
- `CHECK_INT(pid, < MAX_PROCS);`
- `CHECK_HEAP(ctx->rip); CHECK_HEAP(ctx->cr3);`
- 变量含义改变 → 发生奇怪问题（overflow, memory error, ...）
  - 不要小看这些检查，它们在底层编程（M2, L1, ...）时非常常见
  - 在虚拟机神秘重启/卡住/...前发出警报

```
#include "thread.h"
#include "thread-sync.h"

struct cpu {
  int ncli;
};

struct spinlock {
  const char *name;
  int locked;
```

```
  struct cpu *cpu;
};

__thread struct cpu lcpu;

struct cpu *mycpu() {
  return &lcpu;
}

#define panic(...) \
do { \
    fprintf(stderr, "Panic %s:%d ", __FILE__, __LINE__); \
    fprintf(stderr, __VA_ARGS__); \
    fprintf(stderr, "\n"); \
    abort(); \
   } while (0) \

 void
 initlock(struct spinlock *lk, char *name)
 {
   lk->name = name;
   lk->locked = 0;
   lk->cpu = 0;
 }

 // Pushcli/popcli are like cli/sti except that they are matched:
 // it takes two popcli to undo two pushcli.  Also, if interrupts
 // are off, then pushcli, popcli leaves them off.

void
pushcli(void)
{
   // removes CPU-dependent code
   // eflags = readeflags();
   // cli();
   // if(mycpu()->ncli == 0)
   //   mycpu()->intena = eflags & FL_IF;
   mycpu()->ncli += 1;
}

void
popcli(void)
{
   // removes CPU-dependent code
   //if(readeflags()&FL_IF)
   //  panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
     panic("popcli");
   // if(mycpu()->ncli == 0 && mycpu()->intena)
   //  sti();
}

```

```
63  // Check whether this cpu is holding the lock.
64  int
65  holding(struct spinlock *lock)
66  {
67     int r;
68     pushcli();
69     r = lock->locked && lock->cpu == mycpu();
70     popcli();
71     return r;
72  }
73
74  // Acquire the lock.
75  // Loops (spins) until the lock is acquired.
76  // Holding a lock for a long time may cause
77  // other CPUs to waste time spinning to acquire it.
78  void
79  acquire(struct spinlock *lk)
80   {
81     pushcli(); // disable interrupts to avoid deadlock.
82    if(holding(lk))
83      panic("acquire");
84
85     // The xchg is atomic.
86    while(atomic_xchg(&lk->locked, 1) != 0)
87      ;
88
89    // Tell the C compiler and the processor to not move loads or stores
90    // past this point, to ensure that the critical section's memory
91    // references happen after the lock is acquired.
92    __sync_synchronize();
93
94    // Record info about lock acquisition for debugging.
95    lk->cpu = mycpu();
96  }
97
98  // Release the lock.
99  void
100 release(struct spinlock *lk)
101 {
102   if(!holding(lk))
103     panic("release");
104
105   lk->cpu = 0;
106
107   // Tell the C compiler and the processor to not move loads or stores
108   // past this point, to ensure that all the stores in the critical
109   // section are visible to other cores before the lock is released.
110   // Both the C compiler and the hardware may re-order loads and
111   // stores; __sync_synchronize() tells them both not to.
112   __sync_synchronize();
113
114   // Release the lock, equivalent to lk->locked = 0.
```

```
115   // This code can't use a C assignment, since it might
116   // not be atomic. A real OS would use C atomics here.
117   asm volatile("movl $0, %0" : "+m" (lk->locked) : );
118
119   popcli();
120 }
121
122 struct spinlock lk;
123
124 #define N 10000000
125
126 long sum = 0;
127
128 void Tworker(int tid) {
129   lcpu = (struct cpu) { .ncli = 0 };
130   for (int i = 0; i < N; i++) {
131     acquire(&lk);
132     sum++;
133     release(&lk);
134   }
135 }
136
137 int main() {
138   initlock(&lk, "spinlock");
139   for (int i = 0; i < 2; i++) {
140     create(Tworker);
141   }
142   join();
143   printf("sum = %ld\n", sum);
144 }
```

防御性编程对大型系统来说是至关重要的。如果没有适当的 assertions，调试代码会变得非常艰难。

自动运行时检查

动态程序分析

通用（固定）bug 模式的自动检查

- ABBA 型死锁
- 数据竞争
- 带符号整数溢出（undefined behavior）
- Use after free
- ……

动态程序分析：状态机执行历史的一个函数 $f(\tau)$

- 付出程序执行变慢的代价
- 找到更多 bugs

Lockdep：运行时 Lock Ordering 检查

Lockdep 规约（Specification）

- 为每一个锁确定唯一的“allocation site”
  - assert：同一个 allocation site 的锁存在全局唯一的上锁顺序

检查方法：printf

- 记录所有观察到的上锁顺序，例如 $[x,y,z] \Rightarrow x \to y, x \to z, y \to z$
- 检查是否存在 $x \nrightarrow y \wedge y \nrightarrow x$
  - 我们有一个“山寨版”的例子

Lockdep 的实现

- Since Linux Kernel 2.6.17, also in OpenHarmony!

ThreadSanitizer：运行时的数据竞争检查

并发程序的执行 trace

- 内存读写指令（load/store）
- 同步函数调用
- Happens-before：program order 和 release acquire 的传递闭包

对于发生在不同线程且至少有一个是写的 x, y 检查 $x<y \vee y<x$

- 实现：Lamport's Vector Clock

- Time, clocks, and the ordering of events in a distributed system

更多的 Sanitizers

现代复杂软件系统必备的支撑工具

- AddressSanitizer (asan);(paper)

  : 非法内存访问

  - Buffer (heap/stack/global) overflow, use-after-free, use-after-return, double-free, ...
  - Linux Kernel 也有 kasan
- ThreadSanitizer (tsan): 数据竞争
- MemorySanitizer (msan): 未初始化的读取
- UBSanitizer(ubsan): undefined behavior
  - Misaligned pointer, signed integer overflow, ...
  - Kernel 会带着 `-fwrapv` 编译
- IntelliSanitize
  - 等你开发

Take-away Messages

Bugs（包括并发 bugs）一直以来困扰着所有软件工程的实践者。这是因为到目前，仍然没有经济、可靠的手段能帮助我们检查实现的程序逻辑（状态机描述，一个纯粹的数学对象）是否满足我们设计时的种种预期。因此，为了实现“更正确”的软件，我们把对程序的预期表达在程序中（race-free, lock ordering, ...），而不是让程序在自然状态下悄悄进入有问题的状态，就是我们目前解决程序调试问题的折中办法。

各类 sanitizer 给我们带来的启发是：如果我们能清楚地追溯到问题产生的本源，我们就总是能找到好的应对方法——甚至是我们可以构造低配版的“近似” sanitizer，它们在暗中帮助你实现 fail-fast 的程序，从而减轻你调试问题的负担。

课后习题/编程作业

1. 阅读材料

教科书 Operating Systems: Three Easy Pieces:

- 第 32 章 - Concurrency Bugs

2. 编程实践

在实验中尽可能地增加运行时检查和压力测试，以帮助你诊断实验中可能发生的问题。