

多处理器编程入门

多线程编程模型

多个共享内存的状态机

- C 语言状态机的多个线程
 - 共享所有全局变量
 - 独立的栈帧列表
- 汇编语言状态机的多个线程
 - 共享一个地址空间
 - 独立的寄存器 (SP 指向不同内存位置)

状态迁移

- 选择任意一个线程执行一步

多线程编程模型 (cont'd)

Mosaic 状态机

- “heap” 是共享内存
- `sys_sched` 主动随机切换线程
 - 单处理器系统：中断会引起切换
 - (这就是为什么死循环不能把机器卡死)
 - 多处理器系统：真正同时执行
 - 相当于无时无刻在切换

模拟多线程程序的行为

- 思考题：我们可以借助共享内存做什么？

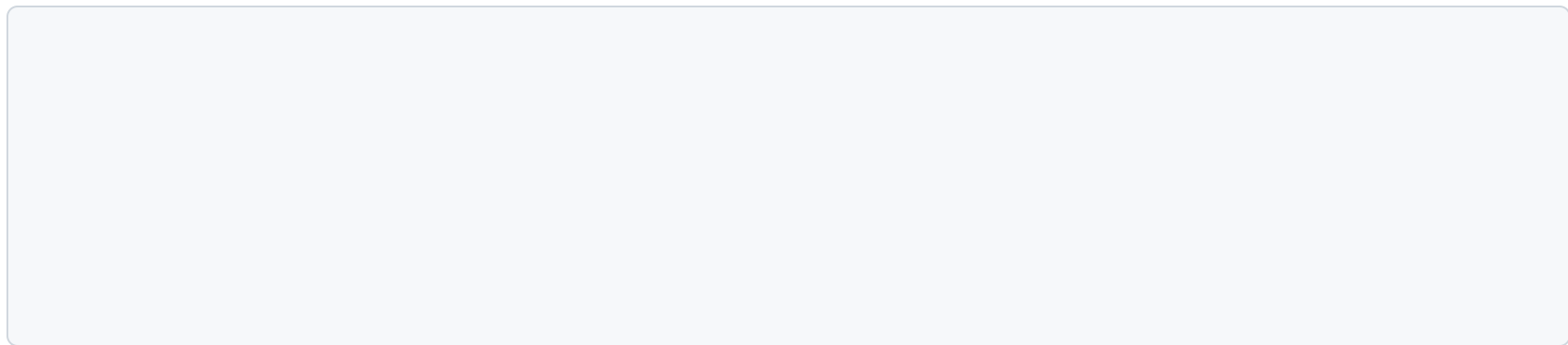
多处理器编程：入门

简化的线程 API (thread.h)

- `spawn(fn)`
 - 创建一个入口函数是 `fn` 的线程，并立即开始执行
 - `void fn(int tid) { ... }`
 - 参数 `tid` 从 1 开始编号
 - 行为： `sys_spawn(fn, tid)`
- `join()`
 - 等待所有运行线程的返回 (也可以不调用)
 - 行为： `while (done != T) sys_sched()`

多处理器编程入门，就这么简单

一个 API 搞定



实现多处理器的利用

- 操作系统会自动把线程放置在不同的处理器上
- CPU 使用率超过了 100%

关于线程的一些疑问 (和解答)

T_a 和 **T_b** 真的共享内存吗？

- 如何证明/否定这件事？

如何证明线程具有独立堆栈 (以及确定堆栈的范围)？

- 输出混乱，应该如何处理？

更多的“好问题” 和解决

- 创建线程使用的是哪个系统调用？
- 能不能用 gdb 调试？
 - 基本原则：**有需求，就能做到 (RTFM)**

放弃 (1)

反思：状态机模型的隐含假设

“世界上只有一个状态机”

- 没有其他任何人能“干涉”程序的状态
- 课堂上常用的简化方法
 - 假设一段程序执行没有系统调用
 - 可以直接简化为一个原子的状态迁移“计算”

放弃：状态迁移原子性的假设

共享内存推翻了“原子性”假设

- 任何时候，load 读到的值都可能是别的线程写入的
- 我们习以为常的简化会漏掉并发程序可能的行为
 - 如果你觉得你可能会犯错误，那别人也一定会的

一些我们见到过的例子

- 线程的交错执行 ABABAABB
- 消失的 1: A2B2A3A5B5B6
- 潘多拉的魔盒已经打开.....

并发 Bugs 来啦!

```
unsigned int balance = 100;

int T_alipay_withdraw(int amt) {
    if (balance >= amt) {
        balance -= amt;
        return SUCCESS;
    } else {
        return FAIL;
    }
}
```

两个线程并发支付 ¥100 会发生什么 (代码演示)

- 账户里会多出用不完的钱!
- Bug/漏洞不跟你开玩笑: Mt. Gox Hack 损失 650,000 BTC
 - 时值 ~\$28,000,000,000

真实的例子：Diablo I (1996)

在捡起物品的瞬间拿起 1 块钱.....



你发现 $1 + 1$ 都不会求了.....

计算 $1 + 1 + 1 + \dots + 1$

- 共计 $2n$ 个 1, 分 2 个线程计算

```
#define N 1000000000
long sum = 0;

void T_sum() { for (int i = 0; i < N; i++) sum++; }

int main() {
    create(T_sum);
    create(T_sum);
    join();
    printf("sum = %ld\n", sum);
}
```


- 会得到怎样的结果?

一些历史

正确实现并发 $1 + 1$ 比想象中困难得多

- 1960s, 大家争先在共享内存上实现原子性 (互斥)
- 但几乎所有的实现都是**错的**
 - 直到 **Dekker's Algorithm**, 还只能保证两个线程的互斥

感到脊背发凉?

- printf 还能在多线程程序里调用吗?
- 我们都知道 printf 是有缓冲区的 (为什么?)
 - 如果执行 `buf[pos++] = ch` 不就  了吗?
 - 消失的 1: `A2B2A3A5B5B6`

失去“原子执行”的终极后果

并发执行三个 `T_sum`，`sum` 的最小值是多少？

- 初始时 `sum = 0`；假设单行语句的执行是原子的

```
void T_sum() {  
    for (int i = 0; i < 3; i++) {  
        int t = load(sum);  
        t += 1;  
        store(sum, t);  
    }  
}
```

- GPT-4: 细微改变问题会导致各种错误回答
 - Claude 3 Opus 也不行，大模型没戏
 - ([Trace recovery is NP-Complete.](#))

答案到底是多少呢？

Model Checker: $\text{sum} = 2$

- 是的，不是 1
 - (因为有 `i` 的循环)
- 也不是 3
 - 虽然 $\text{sum} = 3$ 是很容易想到的
- 无论有多少 `T_sum`，都可以 $\text{sum} = 2$

GPT-4 的“直觉”哪怕对最“简单”的并发程序都不起效

开始理解“数学视角”的价值

对于并发，讲概念是不够的

- 事实可能不是你想的那样

甚至讲代码都是不够的

- 代码需要非常精巧的 workload 才能跑出那个 corner case

证明才是解决问题的方法

证明： \forall 线程调度方法，程序满足 XXX 性质。

- 我们现在甚至还没有趁手的并发程序证明工具！
- 对于课堂的例子，model checker 倒也够用了

放弃 (2)

反思：状态机模型的隐含假设 (再一次)

“世界上只有一个状态机”

- 没有其他任何人能“干涉”程序的状态
- 课堂上常用的简化方法
 - 假设一段程序执行没有系统调用
 - 可以直接简化为一个原子的状态迁移“计算”

编译器也做了同样的假设

- 编译器会试图优化状态迁移，改变执行流

放弃程序按顺序执行的假设

共享内存推翻了编译器的假设

- 但编译器依然会按照顺序执行优化代码
- 否则几乎任何涉及共享内存的代码都变得“不可优化”

程序的行为在并发编程下变得更难理解了

- “顺序程序”变得一点也不“顺序”了

例子：求和 (再次出现)

如果添加编译优化？

- -01 : 1000000000 🙌🙌
- -02 : 2000000000 🙌🙌🙌

另一个例子

“等另一个线程举起旗子，我再继续”？

- 且慢！
- 如果这是个顺序程序，编译器可以做什么优化？
 - (这甚至也是一个常见的并发 bug 模式)
 - “Ad hoc synchronization considered harmful”

控制执行顺序

方法 1：插入“不可优化”代码

- `asm volatile ("" ::: "memory");`
 - 告诉编译器其他线程可能写入内存

方法 2：标记变量 `load/store` 为不可优化

- 使用 `volatile` 修饰变量

以上都不是《操作系统》课推荐的方法

放弃 (3)

反思：状态机模型的隐含假设

状态迁移

- 选择一个线程，执行一条指令
- “顺序一致性” (sequential consistency)

单处理器多线程符合这个假设

- 处理器会保证指令“看起来”顺序完成
- 处理器也是编译器 (oops... 感觉不好的事情要发生了)
 - 预取状态机执行的若干步，然后像编译器一样优化
 - `Load(x); Store(y)`
 - $x \neq y \rightarrow$ 两条指令执行的先后顺序就无所谓
 - Load cache miss \rightarrow store 可以直接执行

放弃全局顺序存在的假设

共享内存推翻了“统一上帝视角”的存在性

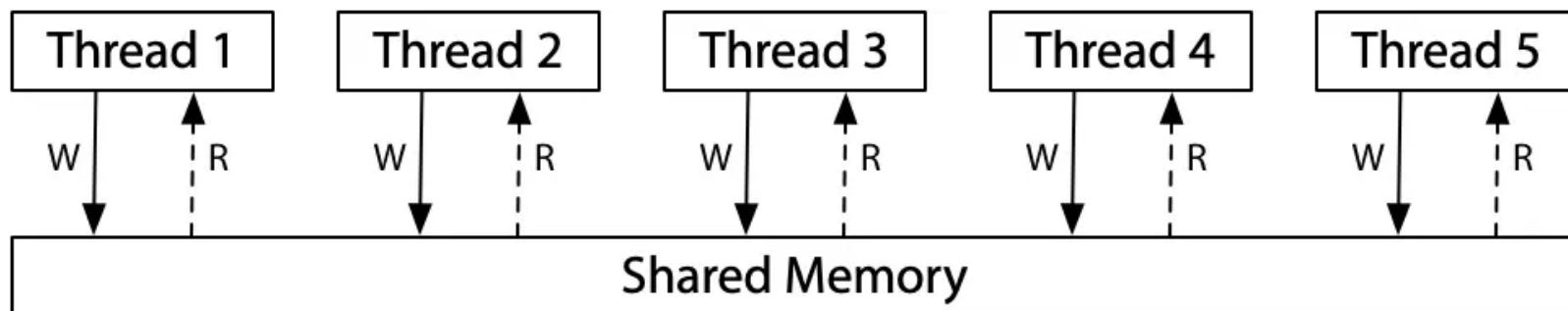
- 就像相对论中，时间顺序的相对性
 - A 和 B 没有因果关系，例子：Load(x) v.s. Store(y)
 - 两个观测者可以分别看到 $A \rightarrow B$ 或 $B \rightarrow A$
 - 观测的相对性使全局世界的行为“极难理解”

不同处理器可能看到不同的共享内存

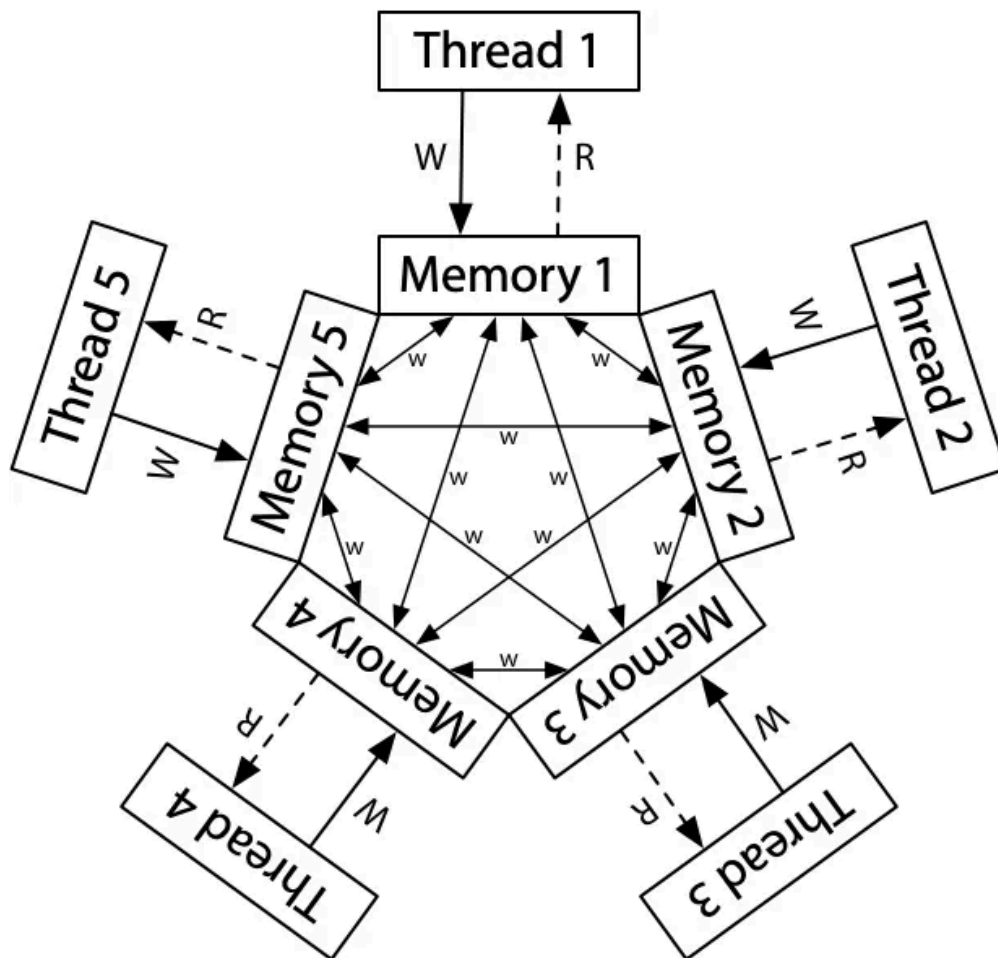
- “一个共享内存”只是个简化的幻觉
- Reading: [Memory Models](#) by Russ Cox

你以为的共享内存

模型是这么建的



实际的共享内存



“相对论效应”带来的后果

```
int x = 0, y = 0;

void T1() {
    x = 1; int t = y; // Store(x); Load(y)
    __sync_synchronize();
    printf("%d", t);
}

void T2() {
    y = 1; int t = x; // Store(y); Load(x)
    __sync_synchronize();
    printf("%d", t);
}
```

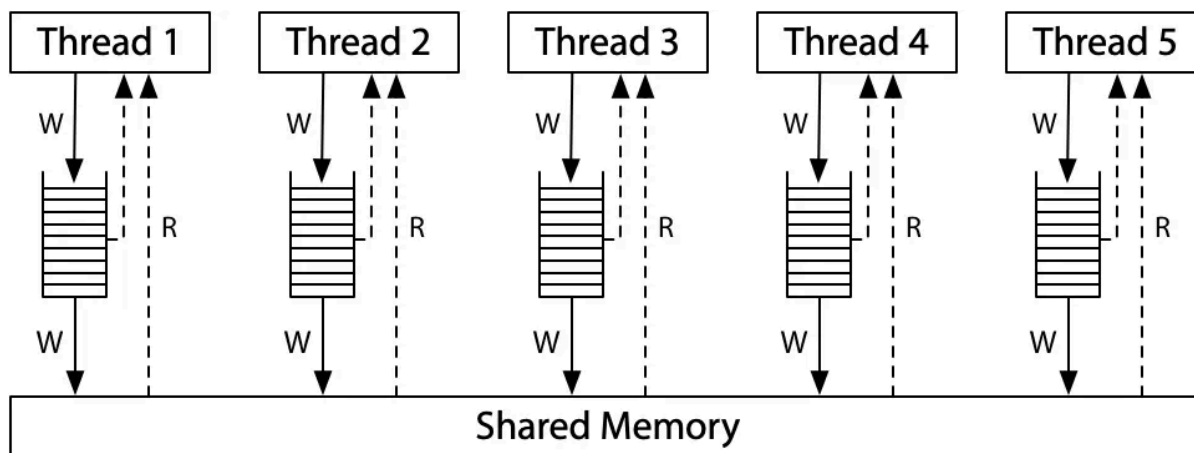
Model Checker: 01 10 11

- 实际: 00 (????)

“相对论效应”带来的后果 (cont'd)

CPU 设计者面临了难题

- 更强的内存模型 = 更糟糕的性能，但更容易编程
- x86：市面“最强”内存模型 (类比 ARM/RISC-V)



因此，在 ARM 上模拟 x86 是个世界性的难题

- [Apple cheated!](#) M1 可以把自己“配置”成 x86