背景

并发进程的正确性

- 独立进程
  - 不和其他进程共享资源或状态
  - 确定性=>输入状态决定结果
  - 可重现=>能够重现起始条件
  - 调度顺序不重要
- 并发进程
  - 在多个进程间有资源共享
  - 不确定性
  - 不可重现
- 并发进程的正确性
  - 执行过程是不确定性和不可重现的
  - 程序错误可能是间歇性发生的

进程并发执行的好处

- 进程需要与计算机中的其他进程和设备进行协作
- 好处 1：共享资源
  - 多个用户使用同一台计算机
  - 银行账号存款余额在多台 ATM 机操作
  - 机器人上的嵌入式系统协调手臂和手的动作
- 好处 2：加速
  - I/O 操作和 CPU 计算可以重叠（并行）
  - 程序可划分成多个模块放在多个处理器上并行执行
- 好处 3：模块化
  - 将大程序分解成小程序

    以编译为例，gcc 会调用 cpp，cc1，cc2，as，ld
  - 使系统易于复用和扩展

并发创建新进程时的标识分配

- 程序可以调用函数 fork() 来创建一个新的进程
  - 操作系统需要分配一个新的并且唯一的进程 ID
  - 在内核中，这个系统调用会运行

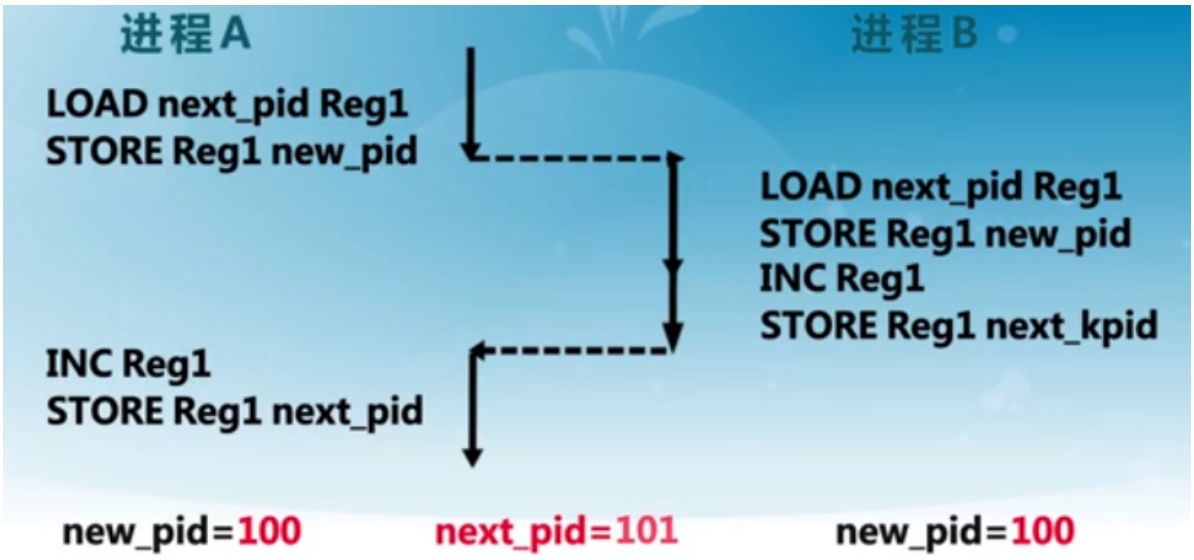
    ```
    new_pid = next_pid++;
    ```

  - 翻译成机器指令

    ```
    LOAD next_pid Reg1
    STORE Reg1 new_pid
    INC Reg1
    STORE Reg1 next_pid
    ```

- 两个进程并发执行时的预期结果（假定 next_pid = 100）
  - 一个进程得到的 ID 应该是 100
  - 另一个进程的 ID 应该是 101
  - next_pid 应该增加到 102

新进程分配标识中的可能错误



会出现并发错误

该图例说明在进程 A 执行到 STORE 的时候，进程 B 抢占资源也开始执行代码

导致二者访问到同一个 new_pid 变量，从而获得了相同的进程 ID

那么这就会导致两个不同进程得到相同的进程 ID 的错误

原子操作（Atomic Operation）

- 原子操作是指一次不存在任何中断或失败的操作
  - 要么操作成功执行
  - 或者操作没有执行
  - 不会出现部分执行的状态
- 操作系统需要利用同步机制在并发执行的同时，保证一些操作是原子操作

现实生活中的同步问题

- 操作系统和现实生活中的问题类比
  - 利用现实生活问题帮助理解操作系统同步问题
  - 同时注意，计算机与人的差异
- 例如：家庭采购协调

时 间	A	B
3:00	查看冰箱，没有面包了	
3:05	离开家去商店	
3:10	到达商店	查看冰箱，没有面包了
3:15	购买面包	离开家去商店
3:20	到家，把面包放进冰箱	到达商店
3:25		购买面包
3:30		到家，把面包放进冰箱

家庭采购协调问题分析

- 如何保证家庭采购协调的成功和高效
  - 有人去买

    需要采购时，有人去买面包
  - 最多只有一个人去买面包
- 可能的解决方法
  - 在冰箱上设置一个**锁和钥匙（lock&key）**
  - 去买面包之前锁住冰箱并且拿走钥匙
- 加锁导致的新问题
  - 冰箱中还有其他食品时，别人无法取到

方案一

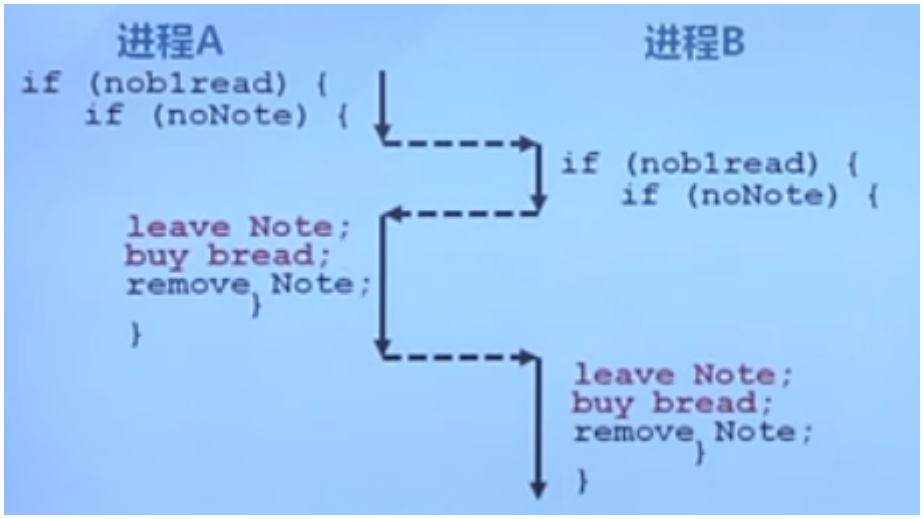
- 使用**便签**来避免购买太多面包
  - 购买之前留下一张便签
  - 买完后移除该便签
  - 别人看到便签时，就不去购买面包

```
if (nobread) {
    if (noNote) {
        leave Note;
        buy bread;
        remove Note;
    }
}
```

- 有效吗?

方案一分析

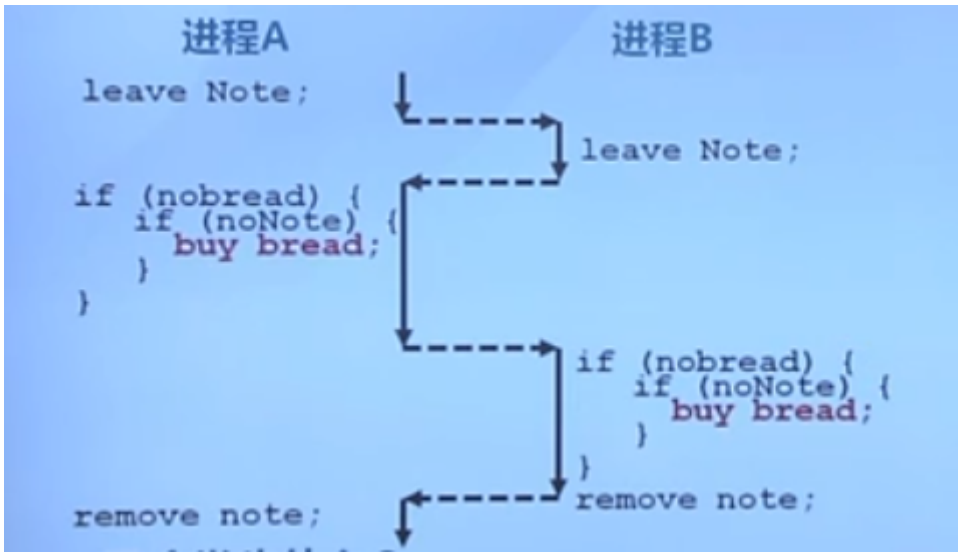
- 偶尔会购买太多面包
  - 检查面包和便签后贴便签前，有其他人检查面包和便签



- 解决方案只是间歇性地失败
  - 问题难以调试
  - 必须考虑调度器所做的事情

方案二

- 先留便签，后检查面包和便签

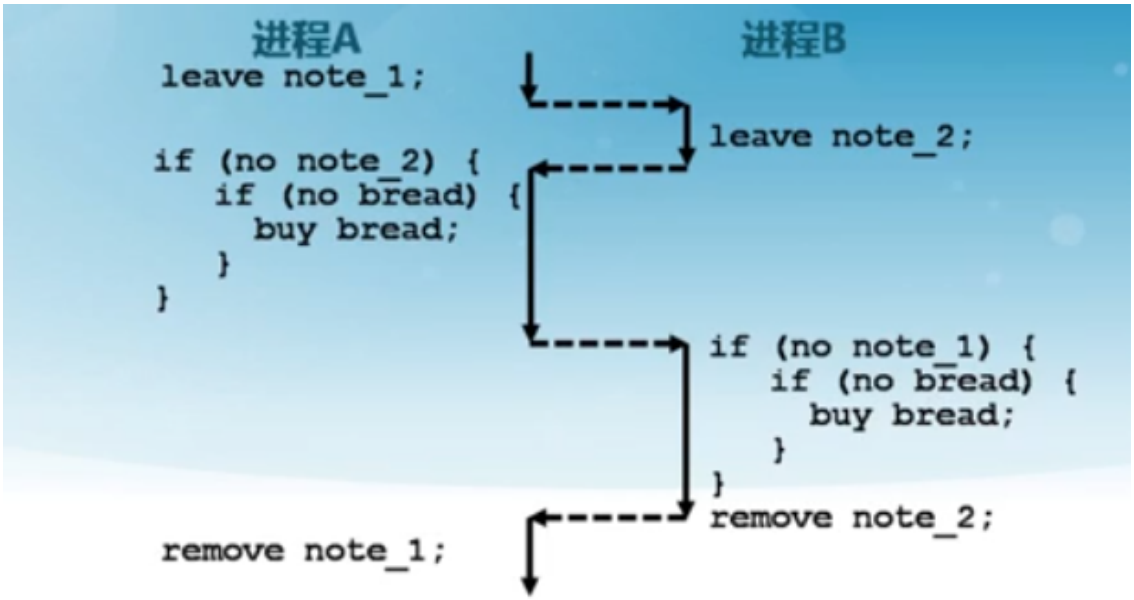


- 会发生什么?
  - 不会有人买面包

方案三

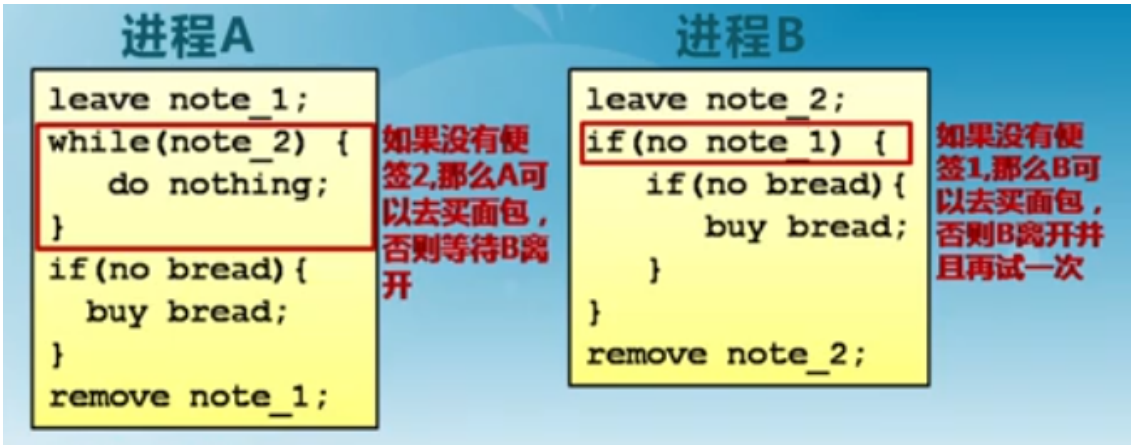
- 为便签增加标记，以区别不同人的便签

  现在可在检查之前留便签
- 会发生什么?
  - 可能导致没有人去买面包
    - 每个人都认为另外一个去买面包



方案四

- 两个人采用不同的处理流程



- 现在有效吗?
  - 枚举所有可能后，可以确认它是有效的
- 这种解决方案你满足?

方案四分析

- 它有效，但太复杂
  - 很难验证它的有效性
- A 和 B 的代码不同
  - 每个进程的代码也会略有不同
  - 如果进程更多，怎么办?
- 当 A 在等待时，它不能做其他事
  - 忙等待（busy-waiting）
- 有更好的方法吗?

方案五

- 利用两个原子操作实现一个锁（lock）
  - Lock.**Acquire**()
    - 在锁被释放前一直等待，然后获得锁
    - 如果两个线程都在等待同一个锁，并且同时发现锁被释放了，那么只有一个能够获得锁
  - Lock.**Release**()
    - 解锁并唤醒任何等待中的进程
- 基于原子锁的解决方法

```
breadlock.Acquire();  // 进入临界区
if (nobread) {
    buy bread;        // 临界区
}
breadlock.Release();  // 退出临界区
```

进程的交互关系：相互感知程度

相互感知的程度	交互关系	进程间的影响
相互不感知（完全不了解其它进程的存在）	独立	一个进程的操作对其他进程的结果无影响
间接感知（双方都与第三方交互，如共享资源）	通过共享进行协作	一个进程的结果依赖于共享资源的状态
直接感知（双方直接交互，如通信）	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息

- 互斥（mutual exclusion）
  - 一个进程占用资源，其它进程不能使用
- 死锁（deadlock）
  - 多个进程各占用部分资源，形成循环等待
- 饥饿（stravation）
  - 其他进程可能轮流占用资源，一个进程一直得不到资源