中国科学院大学
University of Chinese Academy of Sciences

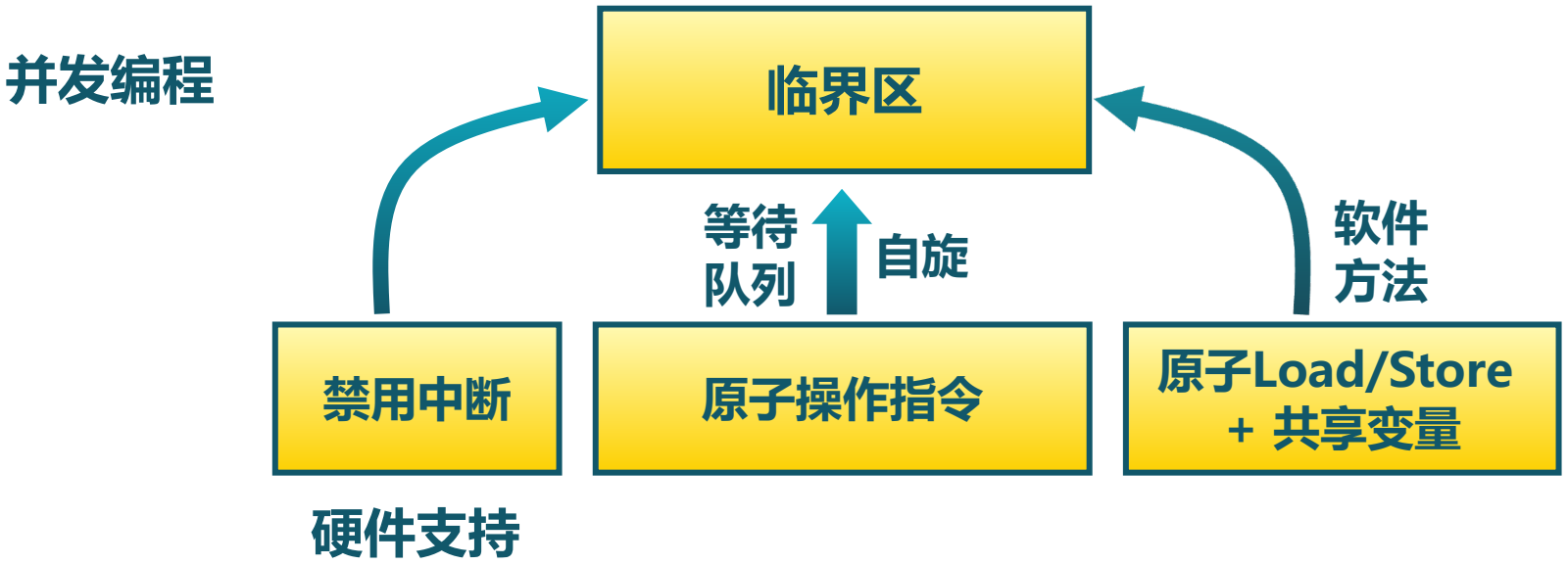
死锁

中国科学院大学计算机学院

2025-10-29

共享资源并发访问

- 锁机制
  - 锁获取：原子操作；一个线程获得锁后，其他线程忙等或阻塞
  - 锁释放：其他等锁线程中的一个线程可以获得锁
  - Sync01: 如何保证锁的原子性（进入临界区的原子性）



示例

- 潜在问题
  - 多个线程/进程同时请求多个共享资源
- 假设一个程序中有两个队列: q1和q2，每个队列操作时需要加锁。现有两个线程按如下逻辑访问队列
  - 线程1：先在q1里插入一项，再在q2里插入一项
  - 线程2：先在q1里删除一项，再在q2里删除一项

内容提要

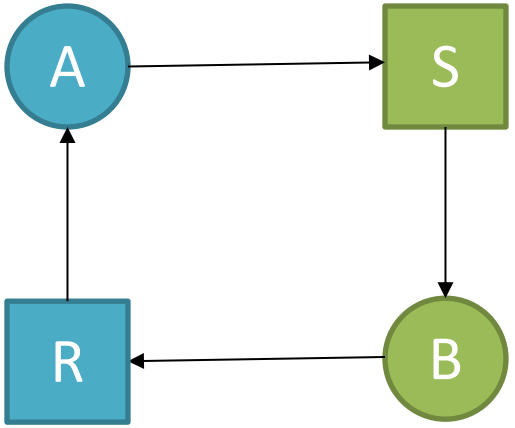
- 死锁的条件
- 处理死锁的策略

资源持有与请求

- 进程 A 持有资源 R

- 进程 B 请求资源 S

- 资源持有与请求形成图中的环路
  → **死锁**
- A 在持有 R 的时候请求 S，B 在持有 S 的时候请求 R

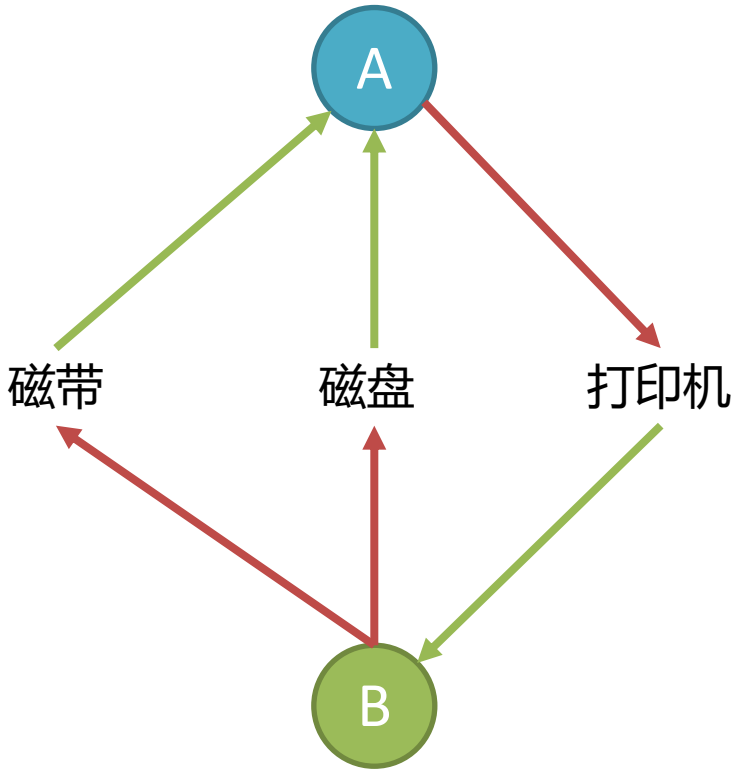


定义

- 死锁 (Deadlock)
  - 两个或两个以上进程/线程在执行过程中，因争夺资源而造成的一种相互等待的现象
- 影响
  - 发生死锁的进程/线程处于忙等或阻塞状态，无法执行
  - 占有的资源无法释放
  - 浪费系统资源，降低系统性能

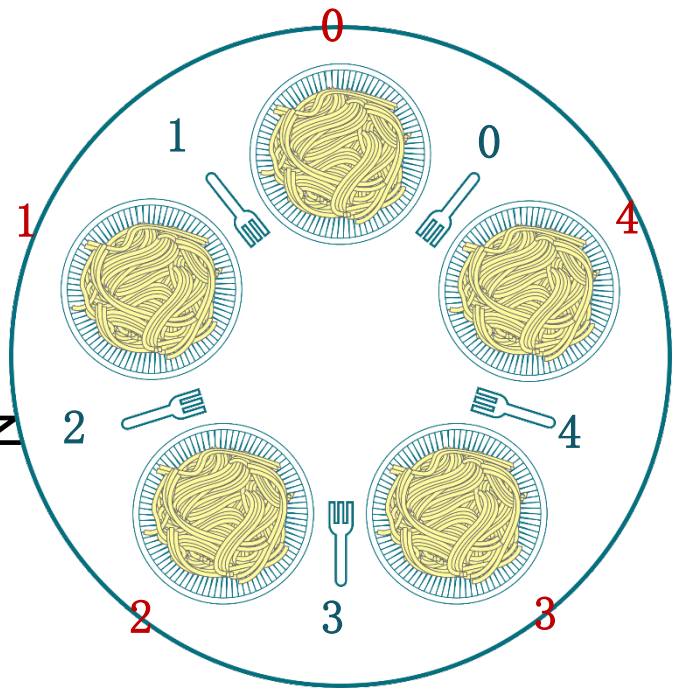
例子一

- 一个工具程序
  - 将一个文件从磁带复制到磁盘
  - 用打印机打印文件
- 资源
  - 磁带
  - 磁盘
  - 打印机
- 死锁
  - A 持有磁带和磁盘，请求打印机
  - B 持有打印机，请求磁带和磁盘



例子二

- 哲学家就餐
  - 5个哲学家围绕一张圆桌而坐
  - 桌子上放着5支叉子
  - 每两个哲学家之间放一支叉子
- 哲学家的动作包括思考和进餐
  - 进餐时需同时拿到左右两边的叉子
  - 思考时将两支叉子放回原处
- 问题
  - 每个哲学家同时进餐，拿起与其编号相同的叉子



死锁的必要条件

- 互斥
  - 某个资源在一段时间内只能由一个线/进程占有，其他线/进程无法访问
- 占有且等待
  - 一个线/进程占有资源，同时请求新资源
  - 新资源被其他线/进程占有，线/进程等待新资源被释放
- 不可抢占 (No Preemption)
  - 资源不可被夺走，只能由占有者主动释放
- 环路等待
  - 多个线/进程以环路的方式进行等待

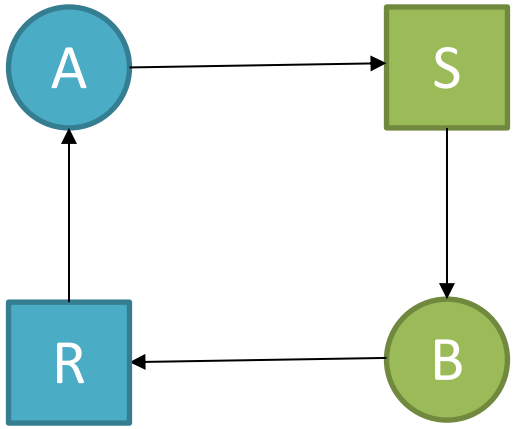
内容提要

- 死锁的条件
- 处理死锁的策略

彻底消除资源竞争？

- 如果让A 运行完毕后再运行B，就不会出现死锁

- 思考
  - 上述思想可以推广到所有多进程/线程竞争资源的场景么？



A与B死锁

策略

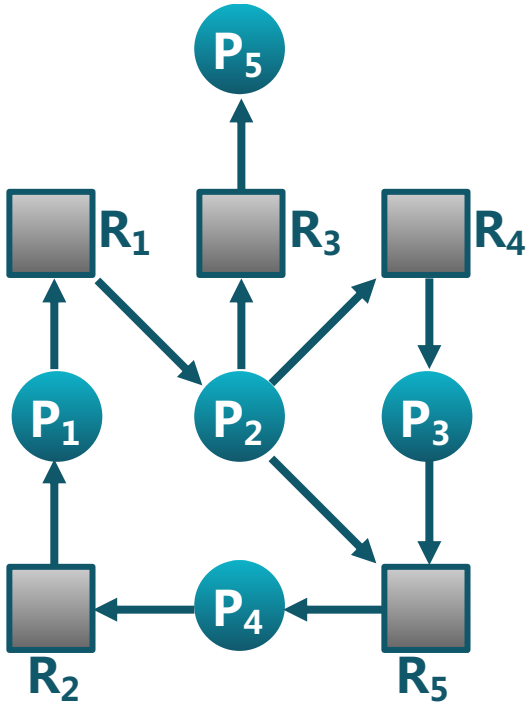
- 忽略死锁问题
  - 都是用户的错
- 检测并恢复 (Detection & Recovery)
  - 允许系统进入死锁状态
  - 事后修复问题
- 动态避免 (Avoidance)
  - 不限制进程申请资源
  - 小心地分配资源
- 静态预防 (Prevention)
  - 破坏四个必要条件中的一个
  - 可能限制进程申请资源

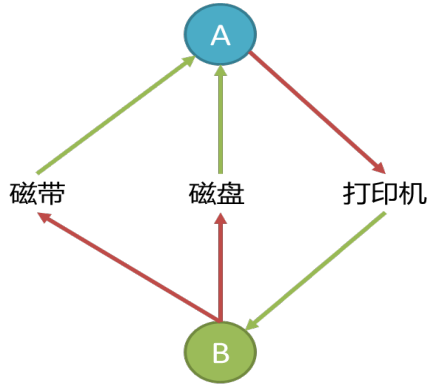
忽略死锁问题—鸵鸟算法

- 操作系统内核死锁（内核卡住）
  - 重启
- 应用程序死锁（程序卡住，“不响应”）
  - 方法一：杀死进程并重启
  - 方法二：给程序定期做快照（checkpoint），重启后，从上一个 checkpoint 重新开始执行，减少进程重启的开销

检测和恢复

- 检测
  - 扫描资源分配图，检测环路
- 恢复
  - 杀死进程/线程
    - 终止所有竞争资源的进/线程：代价大
    - 逐个终止进/线程直至死锁状态解除
  - 代价
    - 重做已经完成的计算
    - 回滚已执行的，常见于数据库场景
- 如何处理磁带-磁盘-打印机的例子？





避免

- 安全状态
  - 系统未发生死锁
  - 存在一个调度方案
    - 使得所有进程能够按照某一次序分配资源，依次运行完成
    - 即使所有进程同时请求最大资源

避免

- 安全状态判断
  - 1）初始化
    - 当前可用资源：Available；进程需求资源：Need
    - 进程已分配资源：Allocation；进程完成标记：Finish = true
  - 2）寻找一个线程Ti，满足以下条件
    - Need <= Available & Finish = false；如果都找不见，则执行4
  - 3）执行Ti，完成后，释放资源，更新如下后，继续执行2）
    - Available += Allocation
    - Finish = true
  - 4）所有进程 Finish = true，则系统安全；否则，系统不安全
  - 核心思想：寻找一个使系统安全的资源分配序列

例子：安全状态判断

总共：8

	Has	Max
P1	2	6
P2	2	3
P3	3	5

空闲：1

	Has	Max
P1	2	6
P2	3	3
P3	3	5

空闲：0

	Has	Max
P1	2	6
P2	0	0
P3	3	5

空闲：3

	Has	Max
P1	2	6
P2	0	0
P3	5	5

空闲：1

	Has	Max
P1	2	6
P2	0	0
P3	0	0

空闲：6

	Has	Max
P1	4	6
P2	1	3
P3	2	5

空闲：1

?

避免

- 银行家算法 (Banker's algorithm, Dijkstra 65)
  - 核心想法
    - 进行what-if analysis
    - 在分配资源前，假设按某个规则给资源做了分配，是否保证系统处于安全状态。若是，则分配
  - 单个资源
    - 每个线程有一个资源需求
    - 总的资源量可能不能满足所有的资源需求
    - 跟踪已分配的资源和仍然需要的资源
    - 每次线程请求资源时，系统分配前检查安全性
  - 多个资源
    - 两个矩阵：已分配和仍然需要

银行家算法：数据结构

- n = 线程数量, m = 资源类型数量
- i: 线程编号，j: 资源编号
- Available（剩余空闲量）：长度为m的向量
  - 当前有 Available[j] 个类型R_j的资源实例可用
- Allocation（已分配量）：n×m矩阵
  - 线程T_i 当前分配了 Allocation[i, j] 个R_j的实例
- Need（需要量）：n×m矩阵
  - 线程T_i 需要 Need[i, j] 个R_j资源实例
  - Need[i,j] = Max[i,j] – Allocation[i,j]
  - Max是线程所需的最大资源量

银行家算法描述

初始化: $Request_i$ 线程T_i的资源请求向量
$Request_i[j]$ 线程T_i请求资源R_j的实例

循环:依次处理线程T_i , i=0, 1, 2, …

1. 如果 $Request_i \le Need[i]$, 转到步骤2。
否则, 拒绝资源申请, 因为线程已经超过了其**最大需求量**

2. 如果 $Request_i \le Available$, 转到步骤3。
否则, T_i 必须**等待**, 因为资源不可用

3. 通过安全状态判断来确定是否分配资源给T_i :
执行what-if判断，进行如下更新计算

$Available = Available - Request_i;$
$Allocation[i] = Allocation[i] + Request_i;$
$Need[i] = Need[i] - Request_i;$

4. 调用安全状态判断

如果返回结果是**安全**，将资源分配给T_i
如果返回结果是**不安全**，系统会拒绝T_i的资源请求

安全状态判断示例

- 初始状态

初始状态

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	6	1	2
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	0	0	1
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
0	1	1

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T2

	R1	R2	R3
T1	3	2	2
T2	6	1	3
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	6	1	3
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	0	0	0
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
0	1	0

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T2

线程T2完成运行

	R1	R2	R3
T1	3	2	2
T2	0	0	0
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	1	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	2	2	2
T2	0	0	0
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
6	2	3

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T1

	R1	R2	R3
T1	3	2	2
T2	0	0	0
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	3	2	2
T2	0	0	0
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
4	0	1

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T1

线程T1完成运行

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	1	0	3
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
7	2	3

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T3

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	3	1	4
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	3	1	4
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
6	2	0

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T3

线程T3完成运行

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
9	3	4

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T4

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

系统资源向量R

R1	R2	R3
5	1	4

当前可用资源向量V

安全状态判断示例

- 可用资源分配给T4

线程T4完成运行

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

最大需求矩阵 C

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	4	2	2

已分配资源矩阵 A

	R1	R2	R3
T1	0	0	0
T2	0	0	0
T3	0	0	0
T4	0	0	0

当前资源请求矩阵C-A

R1	R2	R3
9	3	6

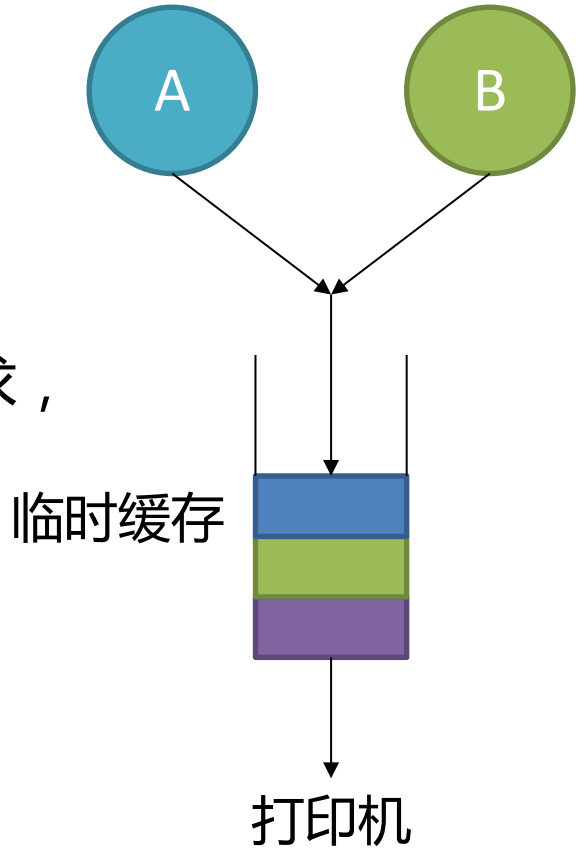
系统资源向量R

R1	R2	R3
5	1	4

当前可用资源向量V

预防：避免互斥

- 区分资源的读写需求，减少互斥的情况
  - 读-读不用互斥，写-读、写-写需要互斥
- 增加资源
  - 有些资源必须互斥访问，例如打印机、磁带等
  - 使用临时缓存（spooling）暂存资源请求，请求进程可以立即返回
  - 后台进程异步地发送资源请求
- Lock-free设计
  - 使用原子操作来修改数据，避免使用互斥锁
  - 实现无阻塞，修改不成功时则会重试、退出或执行其他操作，而不是等待
- 如何处理磁带-磁盘-打印机的例子？



预防：避免占有和等待

- 两阶段加锁 (Two-phase locking)

  阶段 I：

  – 试图对所有所需的资源进行加锁

  阶段 II：

  – 如果成功，使用资源，然后释放资源

  – 否则，释放所有的资源，并从头开始

- 如何处理磁带-磁盘-打印机的例子？

预防：允许抢占

- 使调度器了解资源分配情况
- 方法
  - 如果系统检测到某个进程长时间占有资源且未释放，则抢占该进程，并释放所有资源；释放的资源重新分配给等待的其他进程
- 减少抢占带来的开销
  - 将已完成工作（例如数据、状态等）复制到一个缓冲区，再释放资源
  - 进程再次执行时，可以复用已完成的工作
- 如何处理磁带-磁盘-打印机的例子？

预防：允许抢占

- 回顾：优先级反转问题
  - 高优先级进程所需资源被低优先级进程持有，等待低优先级执行
  - 中优先级进程优先执行，表现为优先级高于高优先级进程
  - 低优先级进程一直等不到执行，而高优先级进程则长时间等待
- 优先级继承
  - 高优先级进程由于等待资源被阻塞时，内核自动提升低优先级进程的优先级，让低优先级进程先执行，并释放资源

预防：避免环路等待

- 资源分级：对所有资源制定请求顺序
- 方法一
  - 对每个资源分配唯一的 id
  - 所有请求必须按 id 升序提出
- 方法二
  - 对每个资源分配唯一的 id
  - 进程不能请求比当前所占有资源编号低的资源
  - 占用高资源编号的进程需释放资源
- 如何处理磁带-磁盘-打印机的例子？

你最喜欢哪种策略？

- 忽略问题
  - 都是用户的错 :(
- 检测并恢复
  - 事后修复问题，代价大
- 动态避免
  - 小心地分配资源
- 预防 (破坏四个条件中的一个)
  - 避免互斥
  - 避免占有和等待
  - 允许抢占
  - 避免环路等待

权衡和应用

- 死锁处理
  - 处理死锁是应用开发者的工作
  - OS 提供打破应用程序死锁的机制：重启、死锁检测
  - OS提供减少死锁恢复开销的机制：快照
- 内核不应该出现死锁
  - 按序请求锁，避免环路等待
  - 死锁检测：lockdep模块
  - 使用细粒度锁，减少死锁机会

总结

- 死锁条件
  - 互斥
  - 占有和等待
  - 不可抢占
  - 环路等待
- 处理死锁的策略
  - 忽略
  - 检测与恢复
  - 动态检测和避免
  - 预防：消除四个必要条件中的一个