死锁问题

- 由于竞争资源或者通信关系，两个或更多线程在执行中出现，永远相互等待只能由其他进程引发的事件

死锁示例：单向通行桥梁

- 桥梁只能单向通行
- 桥的每个部分可视为一个资源
- 可能出现死锁
  - 对向行驶车辆在桥上相遇
  - 解决方法：一个方向的车辆倒退(资源抢占和回退)

死锁示例：单向通行桥梁

- 桥梁只能单向通行
- 桥的每个部分可视为一个资源
- 可能出现死锁
- 可能发生饥饿
  - 由于一个方向的持续车流，另一个方向的车辆无法通过桥梁

进程访问资源的流程

- 资源类型$R_1, R_2, \ldots, R_m$
  - CPU执行时间、内存空间、I/O设备等
- 每类资源R_i有W_i个实例
- 进程访问资源的流程
  - 请求/获取
    申请空闲资源
  - 使用/占用
    进程占用资源
  - 释放
    资源状态由占用变成空闲

资源分类

可重用资源（Reusable Resource）

- 资源不能被删除且在任何时刻只能有一个进程使用
- 进程释放资源后，其他进程可重用
- 可重用资源示例
  - 硬件：处理器、I / O通道、主和副存储器、设备等
  - 软件：文件、数据库和信号量等数据结构

资源分类

可重用资源（Reusable Resource）

- 资源不能被删除且在任何时刻只能有一个进程使用
- 进程释放资源后，其他进程可重用
- 可重用资源示例
- 可能出现死锁
  - 每个进程占用一部分资源并请求其它资源

资源分类

可重用资源（Reusable Resource）

- 资源不能被删除且在任何时刻只能有一个进程使用
- 进程释放资源后，其他进程可重用
- 可重用资源示例
- 可能出现死锁

消耗资源(Consumable resource)

- 资源创建和销毁
- 消耗资源示例
  - 在I/O缓冲区的中断、信号、消息等

资源分类

可重用资源（Reusable Resource）

- 资源不能被删除且在任何时刻只能有一个进程使用
- 进程释放资源后，其他进程可重用
- 可重用资源示例
- 可能出现死锁

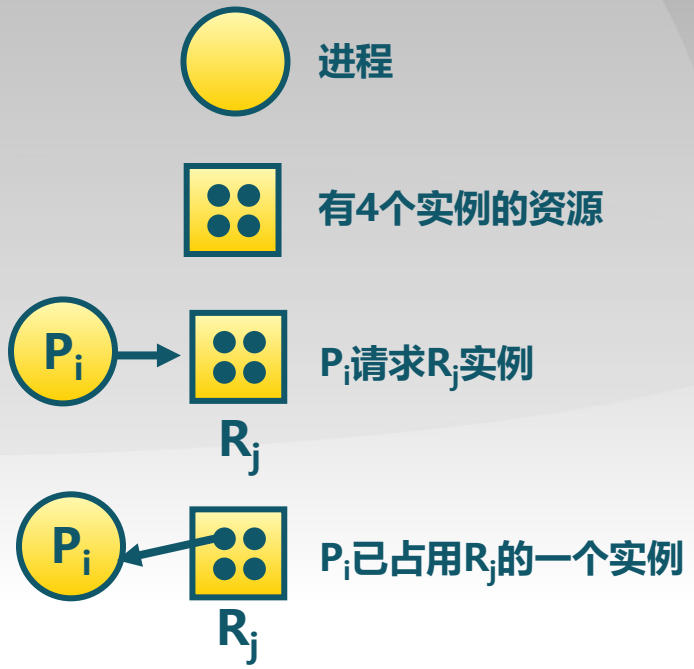
消耗资源(Consumable resource)

- 资源创建和销毁
- 消耗资源示例
- 可能出现死锁
  - 进程间相互等待接收对方的消息

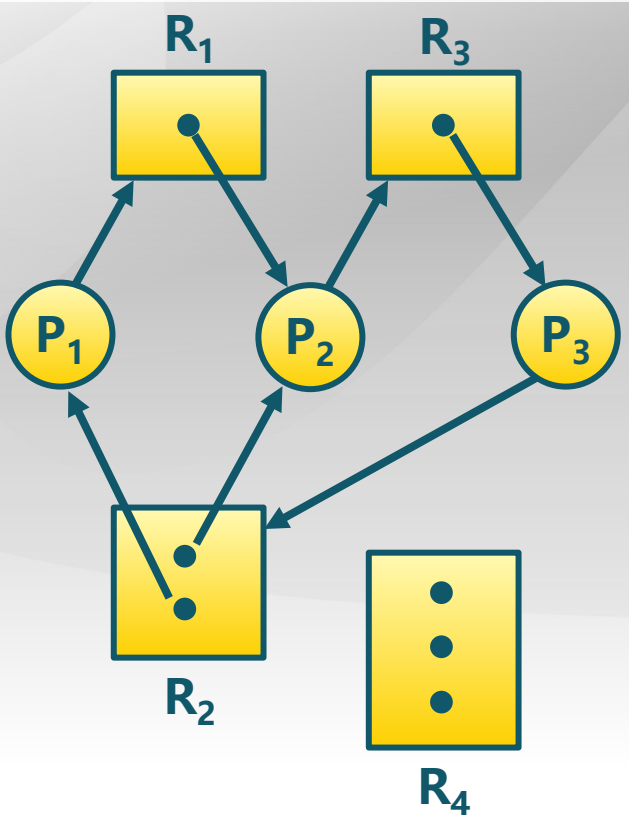
资源分配图

描述资源和进程间的分配和占用关系的有向图

- 两类顶点
  - 系统中的所有进程
    $P = \{P_1, P_2, \ldots, P_n\}$
  - 系统中的所有资源
    $R = \{R_1, R_2, \ldots, P_m\}$
- 两类有向边
  - 资源请求边
    进程P_i请求资源R_j：$P_i \rightarrow R_j$
  - 资源分配边
    资源R_j已分配给进程P_i：$R_j \rightarrow P_i$

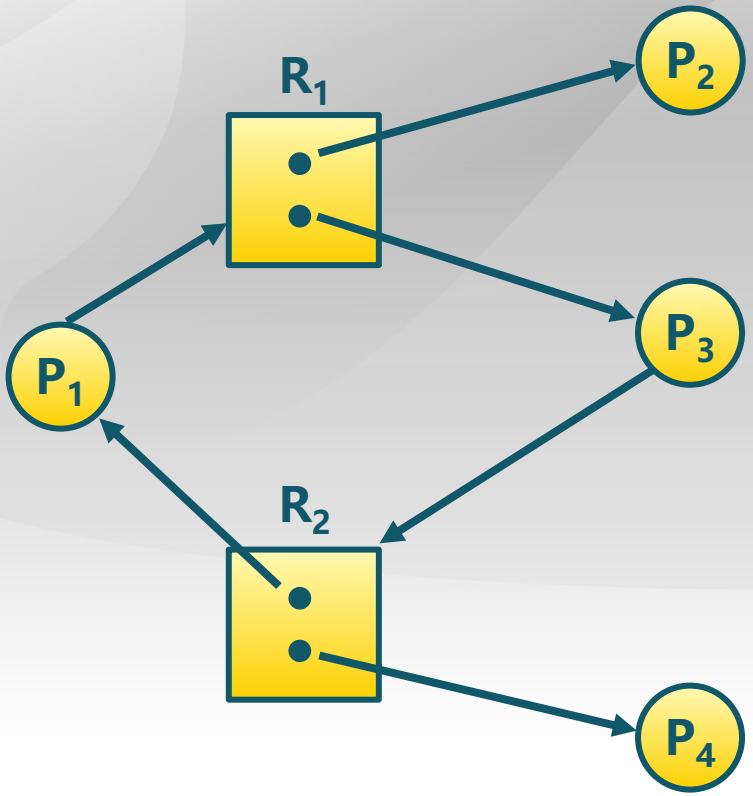


资源分配图示例



存在死锁

资源分配图示例



有循环等待，但没有锁死

出现死锁的必要条件

- 互斥
  - 任何时刻只能有一个进程使用一个资源实例

出现死锁的必要条件

- 互斥
- 持有并等待
  - 进程保持至少一个资源，并正在等待获取其他进程持有的资源

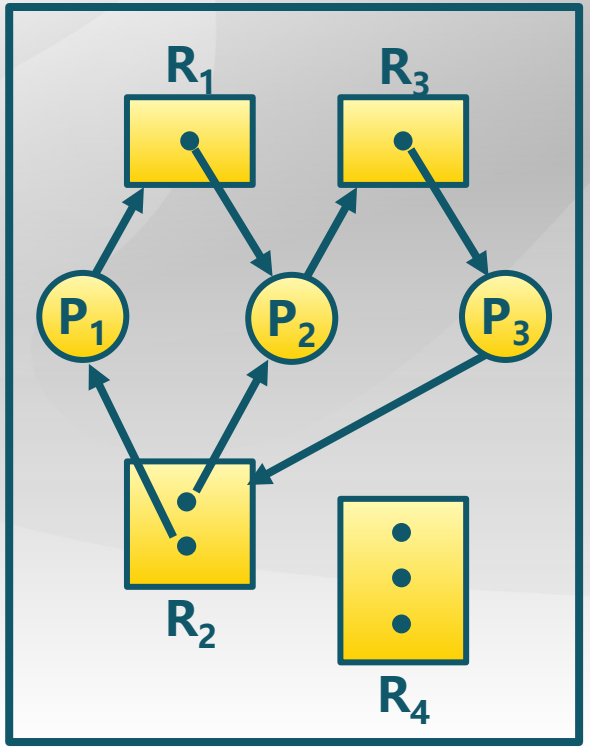
出现死锁的必要条件

- 互斥
- 持有并等待
- 非抢占
  - 资源只能在进程使用后自愿释放

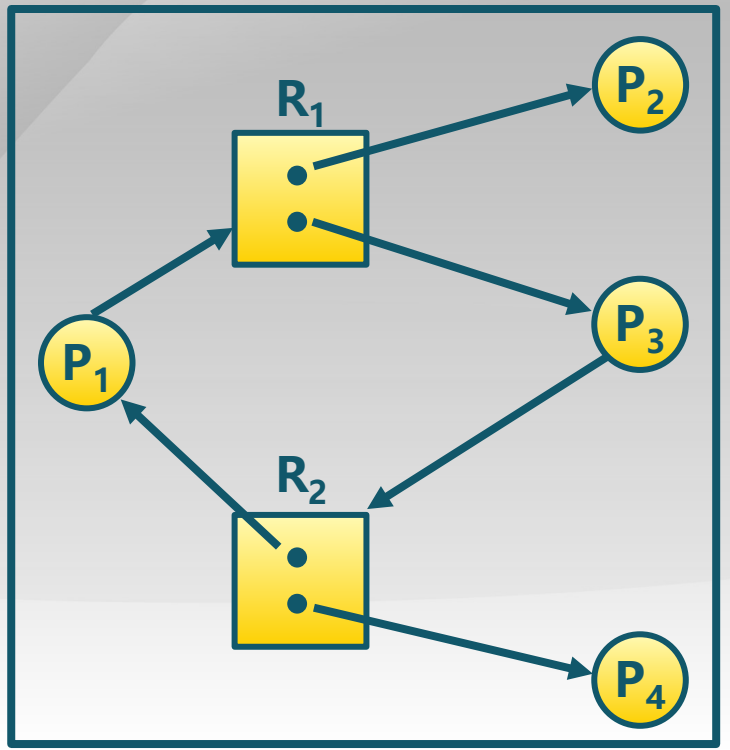
出现死锁的必要条件

- 互斥
- 持有并等待
- 非抢占
- 循环等待
  - 存在等待进程集合{P_0，P_1，…，P_N}，
    P_0正在等待P_1所占用的资源，
    P_1 正在等待P_2占用的资源，…，
    P_{N-1}在等待P_N所占用资源，
    P_N正在等待P_0所占用的资源

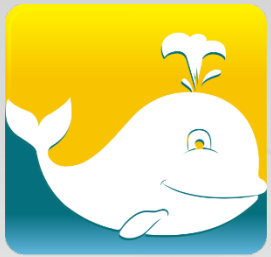
出现死锁的必要条件



死锁



没有死锁



操作系统

Operating Systems

死锁处理方法

- 死锁预防(Deadlock Prevention)
  - 确保系统永远不会进入死锁状态
- 死锁避免(Deadlock Avoidance)
  - 在使用前进行判断，只允许不会出现死锁的进程请求资源
- 死锁检测和恢复(Deadlock Detection & Recovery)
  - 在检测到运行系统进入死锁状态后，进行恢复
- 由应用进程处理死锁
  - 通常操作系统忽略死锁
    - 大多数操作系统（包括UNIX）的做法

死锁预防：限制申请方式

预防是采用某种策略，**限制**并发进程对资源的请求，使系统在任何时刻都**不满足死锁的必要条件**。

- 互斥
  - 把互斥的共享资源封装成可同时访问

死锁预防：限制申请方式

预防是采用某种策略，**限制**并发进程对资源的请求，使系统在任何时刻都**不满足死锁的必要条件**。

- 互斥
- 持有并等待
  - 进程请求资源时，要求它不持有任何其他资源
  - 仅允许进程在开始执行时，一次请求所有需要的资源
  - 资源利用率低

死锁预防：限制申请方式

预防是采用某种策略，**限制**并发进程对资源的请求，使系统在任何时刻都**不满足死锁的必要条件**。

- 互斥
- 持有并等待
- 非抢占
  - 如进程请求不能立即分配的资源，则释放已占有资源
  - 只在能够同时获得所有需要资源时，才执行分配操作

死锁预防：限制申请方式

预防是采用某种策略，**限制**并发进程对资源的请求，使系统在任何时刻都**不满足死锁的必要条件**。

- 互斥
- 持有并等待
- 非抢占
- 循环等待
  - 对资源排序，要求进程按顺序请求资源

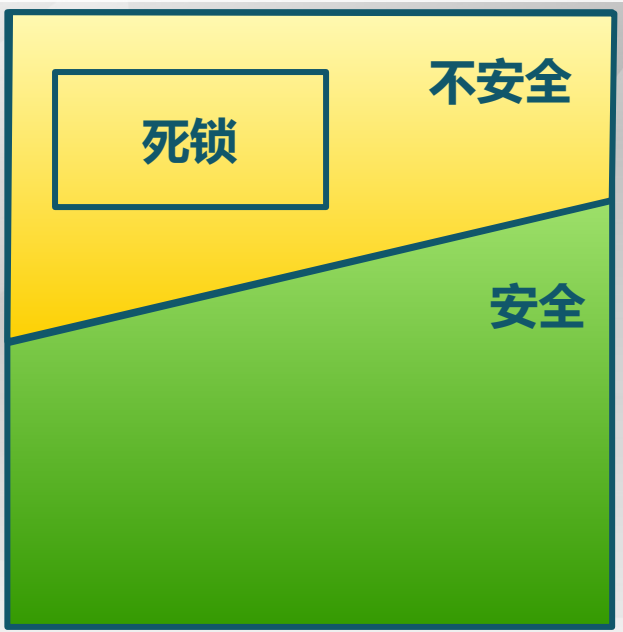
死锁避免

- 利用额外的先验信息，在分配资源时判断是否会出现死锁，只在不会死锁时分配资源
  - 要求进程声明需要资源的**最大数目**
  - 限定**提供**与**分配**的资源数量，确保满足进程的**最大**需求
  - **动态检查**的资源分配状态，确保不会出现环形等待

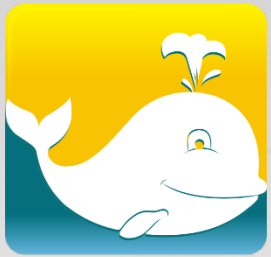
系统资源分配的安全状态

- 当进程请求资源时，系统判断分配后是否处于安全状态
- 系统处于安全状态
  - 针对所有已占用进程，存在安全序列
- 序列<P_1，P_2，…，P_N>是安全的
  - P_i要求的资源≤当前可用资源+所有P_j 持有资源 其中j<i
  - 如P_i的资源请求不能立即分配，则P_i等待所有 P_j（j<i）完成
  - P_i完成后，P_{i+1}可得到所需资源，执行并释放所分配的资源
  - 最终整个序列的所有P_i都能获得所需资源

安全状态与死锁的关系



- 系统处于安全状态，一定没有死锁
- 系统处于不安全状态，可能出现死锁
  - 避免死锁就是确保系统不会进入不安全状态



操作系统

Operating Systems

银行家算法（Banker's Algorithm）

- 银行家算法是一个避免死锁产生的算法。以银行借贷分配策略为基础，判断并保证系统处于安全状态
  - 客户在第一次申请贷款时，声明所需最大资金量，在满足所有贷款要求并完成项目时，及时归还
  - 在客户贷款数量不超过银行拥有的最大值时，银行家尽量满足客户需要
  - 类比
    - 银行家 ⟷ 操作系统
    - 资金 ⟷ 资源
    - 客户 ⟷ 申请资源的线程

银行家算法：数据结构

n = 线程数量, m = 资源类型数量

- **Max（总需求量）**：n×m矩阵
  线程T_i最多请求类型R_j的资源 Max[i,j] 个实例
- **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（未来需要量）**：n×m矩阵
  线程T_i 未来需要 Need[i, j] 个R_j资源实例

Need[i,j] = Max[i,j] – Allocation[i,j]

银行家算法：安全状态判断

1.Work 和Finish 分别是长度为m和n的向量初始化:
Work = Available //当前资源剩余空闲量
Finish[i] = false for i: 1,2, …, n. //线程i没结束

2.寻找线程T_i:
(a) Finish[i] = false //接下来找出Need比Work小的线程i
(b) Need[i]≤Work
没有找到满足条件的T_i，转4。

3.Work = Work + Allocation[i] //线程i的资源需求量小于当前剩余空闲资源量, 所以配置给它再回收
Finish[i] = true
转2.

4.如所有线程Ti满足Finish[i] == true，则系统处于安全状态 //所有线程的Finish为True, 表明系统处于安全状态

银行家算法

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:

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**最大要求**

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 :

生成一个需要判断状态是否安全的资源分配环境

$Available = Available - Request_i;$

$Allocation[i] = Allocation[i] + Request_i;$

$Need[i] = Need[i] - Request_i;$

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

银行家算法的安全状态判断示例

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

银行家算法的安全状态判断示例

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

银行家算法的安全状态判断示例

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

银行家算法的安全状态判断示例

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

银行家算法的安全状态判断示例2

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	5	1	1
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

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

当前可用资源向量V

银行家算法的安全状态判断示例2

线程T1请求R1和R3资源各1个实例

	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	2	0	1
T2	5	1	1
T3	2	1	1
T4	0	0	2

已分配资源矩阵 A

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

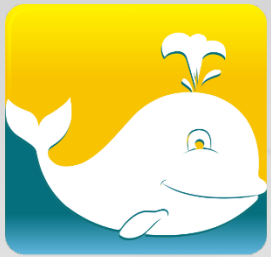
当前资源请求矩阵C-A

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

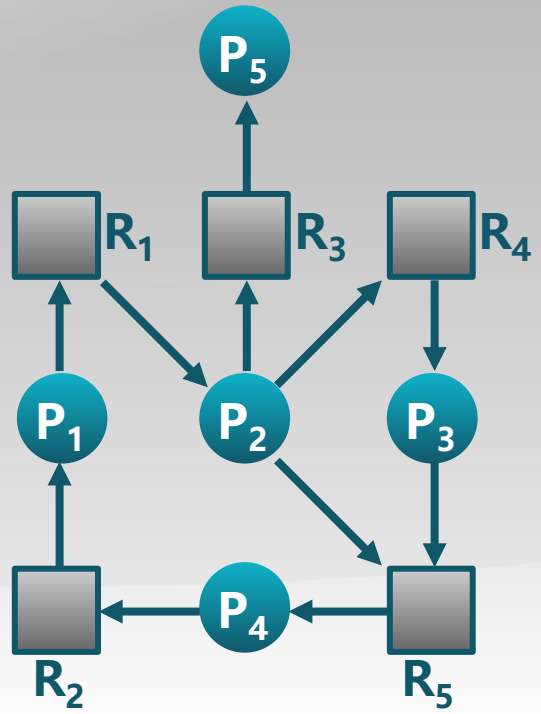


操作系统

Operating Systems

死锁检测

- 允许系统进入死锁状态
- 维护系统的资源分配图
- 定期调用死锁检测算法来搜索图中是否存在死锁
- 出现死锁时，用死锁恢复机制进行恢复



死锁检测算法：数据结构

- **Available**:长度为m的向量
  每种类型可用资源的数量
- **Allocation**:一个n×m矩阵
  当前分配给各个进程每种类型资源的数量
  进程P_i 拥有资源R_j的Allocation[i, j]个实例

死锁检测算法

1.Work 和Finish 分别是长度为m和n的向量初始化:

(a)Work = Available　　//work为当空闲资源量

(b)Allocation[i]> 0 时, Finish[i] = false;　　//finish为线程是否结束

否则，Finish[i] = true

2.寻找线程T_i满足:

(a)Finish[i] = false　　//线程没有结束的线程，且此线程将需要的资源量小于当前空闲资源量

(b)$Request_i \leq Work$

没有找到这样的i，转到4

3.Work = Work + Allocation[i]　　//把找到的线程拥有的资源释放回当前空闲资源中

Finish[i] = true

转到2

4.如某个Finish[i] == false，系统处于死锁状态　　//如果有Finish为false,表明系统处于死锁状态

算法需要$O(m \times n^2)$ 操作检测是否系统处于死锁状态

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2			
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0	7	2	4
T_4	0	0	2	0	0	2			

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	1	0
T_1	2	0	0	2	0	2	5	1	3
T_2	3	0	3	0	0	0	3	1	3
T_3	2	1	1	1	0	0	7	2	4
T_4	0	0	2	0	0	2	7	2	6

- 序列<P_0, P_2, P_1, P_3, P_4> 对于所有的i，都可满足Finish[i] = true

死锁检测示例

- 5个线程T_0 到 T_4；3种资源类型
  A (7个实例), B (2个实例), and C (6个实例)
- 在T_0时刻：

	已分配资源			资源请求			当前可用资源		
	A	B	C	A	B	C	A	B	C
T_0	0	1	0	0	0	0	0	0	0
T_1	2	0	0	2	0	1			
T_2	3	0	3	0	0	1			
T_3	2	1	1	1	0	0			
T_4	0	0	2	0	0	2			

可以通过回收进程P_0占用的资源，但资源不足以无法完成其他进程请求
死锁存在，包括进程P_1, P_2, P_3, P_4

死锁检测算法的使用

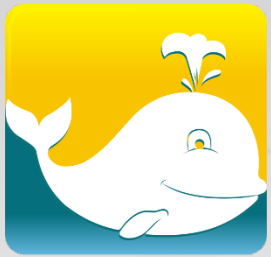
- 死锁检测的时间和周期选择依据
  - 死锁多久可能会发生
  - 多少进程需要被回滚
- 资源图可能有多个循环
  - 难于分辨“造成”死锁的关键进程

死锁恢复：进程终止

- 终止所有的死锁进程
- 一次只终止一个进程直到死锁消除
- 终止进程的顺序应该是
  - 进程的优先级
  - 进程已运行时间以及还需运行时间
  - 进程已占用资源
  - 进程完成需要的资源
  - 终止进程数目
  - 进程是交互还是批处理

死锁恢复：资源抢占

- 选择被抢占进程
  - 最小成本目标
- 进程回退
  - 返回到一些安全状态，重启进程到安全状态
- 可能出现饥饿
  - 同一进程可能一直被选作被抢占者



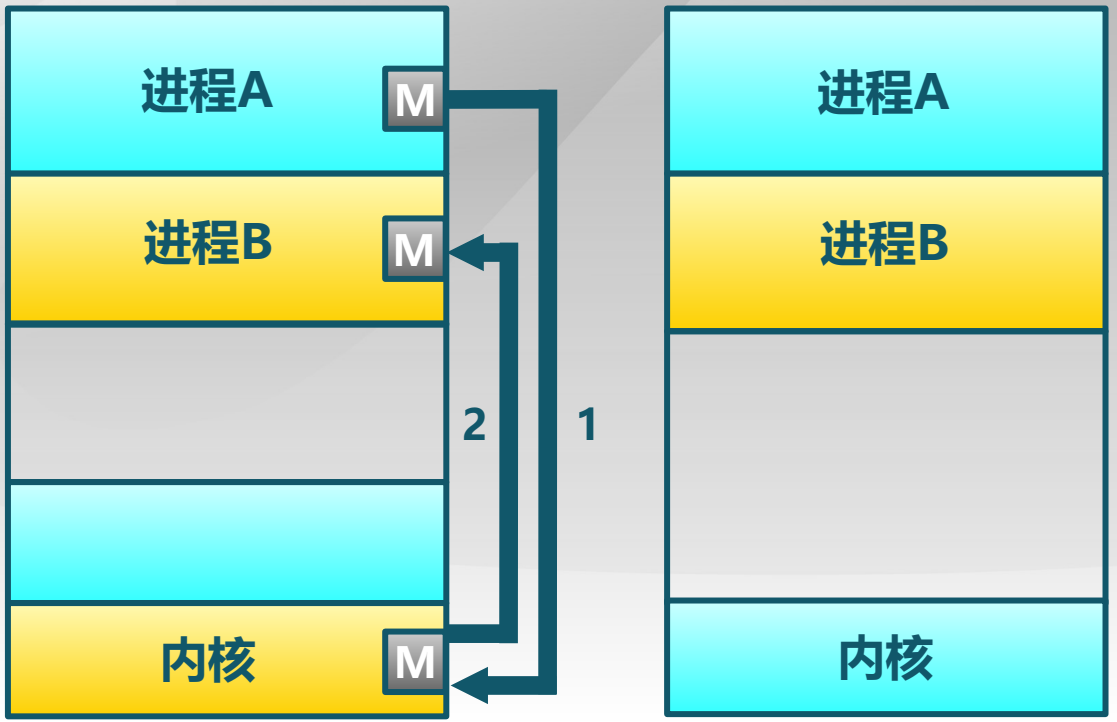
操作系统

Operating Systems

进程通信（IPC, Inter-Process Communication）

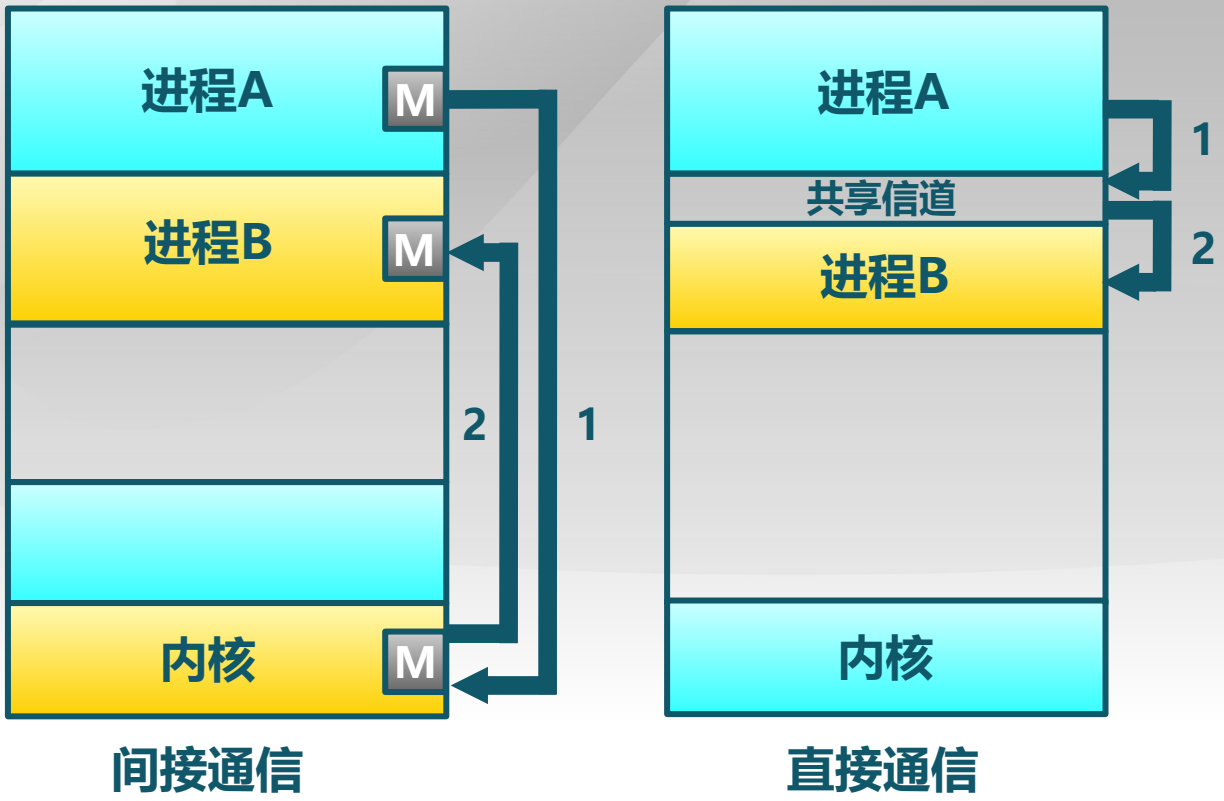
- 进程通信是进程进行通信和同步的机制
- IPC提供2个基本操作
  - 发送操作：send(message)
  - 接收操作：receive(message)
- 进程通信流程
  - 在通信进程间建立通信链路
  - 通过 send/receive交换消息
- 进程链路特征
  - 物理 (如，共享内存，硬件总线)
  - 逻辑 (如，逻辑属性)

通信方式



间接通信　　直接通信

通信方式



直接通信

- 进程必须正确的命名对方
  - send (P, message) – 发送信息到进程P
  - receive(Q, message) – 从进程 Q接受消息
- 通信链路的属性
  - 自动建立链路
  - 一条链路恰好对应一对通信进程
  - 每对进程之间只有一个链接存在
  - 链接可以是单向的，但通常为双向的

间接通信

- 通过操作系统维护的消息队列实现进程间的消息接收和发送
  - 每个消息队列都有一个唯一的标识
  - 只有共享了相同消息队列的进程，才能够通信
- 通信链路的属性
  - 只有共享了相同消息队列的进程，才建立连接
  - 连接可以是单向或双向
  - 消息队列可以与多个进程相关联
  - 每对进程可以共享多个消息队列

间接通信

- 通信流程
  - 创建一个新的消息队列
  - 通过消息队列发送和接收消息
  - 销毁消息队列
- 基本通信操作

  send(A, message) – 发送消息到队列A

  receive(A, message) – 从队列 A接受消息

阻塞与非阻塞通信

- 进程通信可划分为阻塞（同步）或非阻塞（异步）
- 阻塞通信
  - 阻塞发送
    发送者在发送消息后进入等待，直到接收者成功收到

阻塞与非阻塞通信

- 进程通信可划分为阻塞（同步）或非阻塞（异步）
- 阻塞通信
  - 阻塞发送
  - 阻塞接收
    接收者在请求接收消息后进入等待，直到成功收到一个消息

阻塞与非阻塞通信

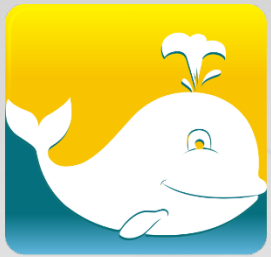
- 进程通信可划分为阻塞（同步）或非阻塞（异步）
- 阻塞通信
  - 阻塞发送
  - 阻塞接收
- 非阻塞通信
  - 非阻塞发送
    发送者在消息发送后，可立即进行其他操作

阻塞与非阻塞通信

- 进程通信可划分为阻塞（同步）或非阻塞（异步）
- 阻塞通信
  - 阻塞发送
  - 阻塞接收
- 非阻塞通信
  - 非阻塞发送
  - 非阻塞接收
    没有消息发送时，接收者在请求接收消息后，
    接收不到任何消息

通信链路缓冲

- 进程发送的消息在链路上可能有3种缓冲方式
  - 0 容量
    发送方必须等待接收方
  - 有限容量
    通信链路缓冲队列满时，发送方必须等待
  - 无限容量
    发送方不需要等待



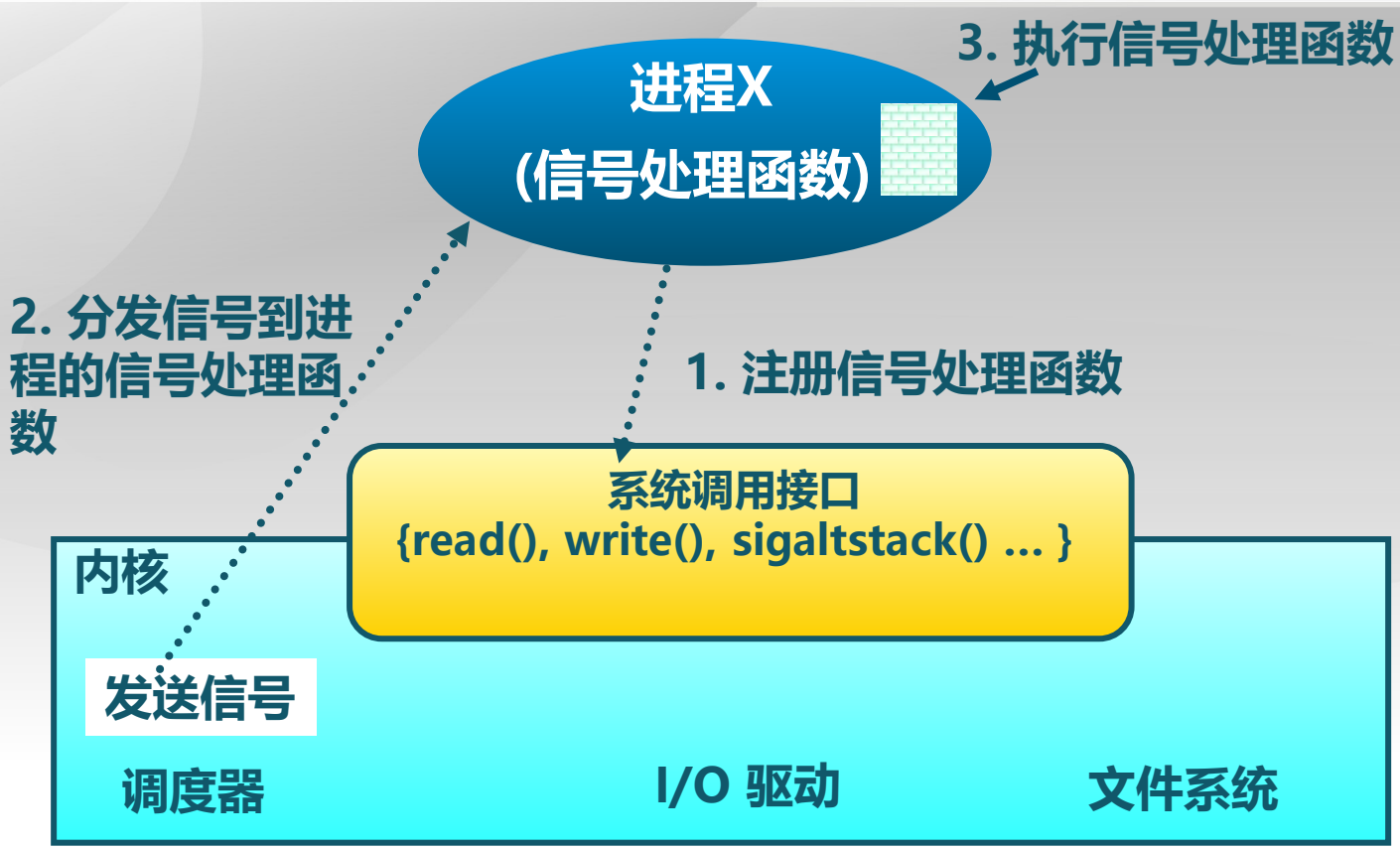
操作系统

Operating Systems

信号（Signal）

- 信号
  - 进程间的软件中断通知和处理机制
  - 如：SIGKILL, SIGSTOP, SIGCONT等
- 信号的接收处理
  - 捕获(catch)：执行进程指定的信号处理函数被调用
  - 忽略(Ignore)：执行操作系统指定的缺省处理
    - 例如：进程终止、进程挂起等
  - 屏蔽（Mask）：禁止进程接收和处理信号
    - 可能是暂时的(当处理同样类型的信号)
- 不足
  - 传送的信息量小，只有一个信号类型

信号的实现


3. 执行信号处理函数
进程X
(信号处理函数)
2. 分发信号到进程的信号处理函数
1. 注册信号处理函数
系统调用接口
{read(), write(), sigaltstack() ... }
内核
发送信号
调度器
I/O 驱动
文件系统

信号使用示例

```
#include <stdio.h>
#include <signal.h>

main()
{
signal(SIGINT, sigproc);     /* DEFAULT ACTION: term */
signal(SIGQUIT, quitproc);   /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

信号使用示例

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
signal(SIGINT, sigproc);    /* NOTE some versions of UNIX will reset
                             * signal to default after each call. So for
                             * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}




main()
{
signal(SIGINT, sigproc);     /* DEFAULT ACTION: term */
signal(SIGQUIT, quitproc);   /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

信号使用示例

```
#include <stdio.h>
#include <signal.h>
void sigproc()
{
signal(SIGINT, sigproc);    /* NOTE some versions of UNIX will reset
                             * signal to default after each call. So for
                             * portability reset signal each time */

    printf("you have pressed ctrl-c - disabled \n");
}

void quitproc()
{
    printf("ctrl-\\ pressed to quit\n");    /* this is "ctrl" & "\" */
    exit(0); /* normal exit status */
}

main()
{
signal(SIGINT, sigproc);     /* DEFAULT ACTION: term */
signal(SIGQUIT, quitproc);   /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\\ to quit\n");

    for(;;);
}
```

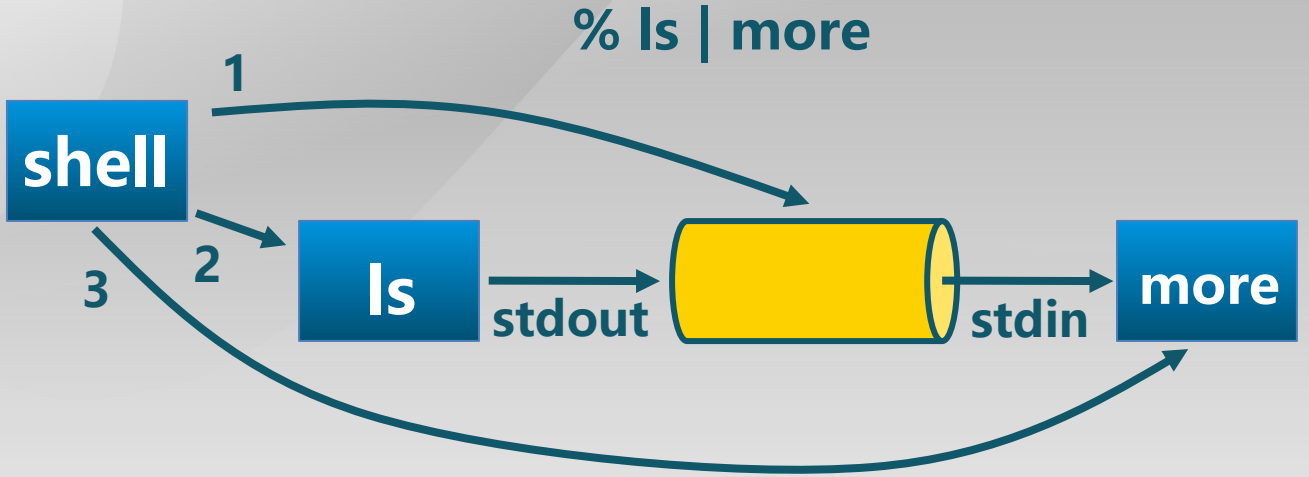
管道(pipe)

- 进程间基于内存文件的通信机制
  - 子进程从父进程继承文件描述符
  - 缺省文件描述符：0 stdin, 1 stdout, 2 stderr
- 进程不知道（或不关心！）的另一端
  - 可能从键盘、文件、程序读取
  - 可能写入到终端、文件、程序

与管道相关的系统调用

- 读管道：read(fd, buffer, nbytes)
  scanf()是基于它实现的
- 写管道： write(fd, buffer, nbytes)
  printf()是基于它实现的
- 创建管道：pipe(rgfd)
  rgfd是2个文件描述符组成的数组
  rgfd[0]是读文件描述符
  rgfd[1]是写文件描述符

管道示例

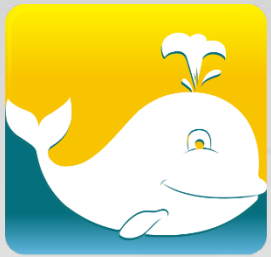


shell

创建管道

为ls创建一个进程，设置 stdout为 管道写端

为more 创建一个进程, 设置 stdin 为管道读端

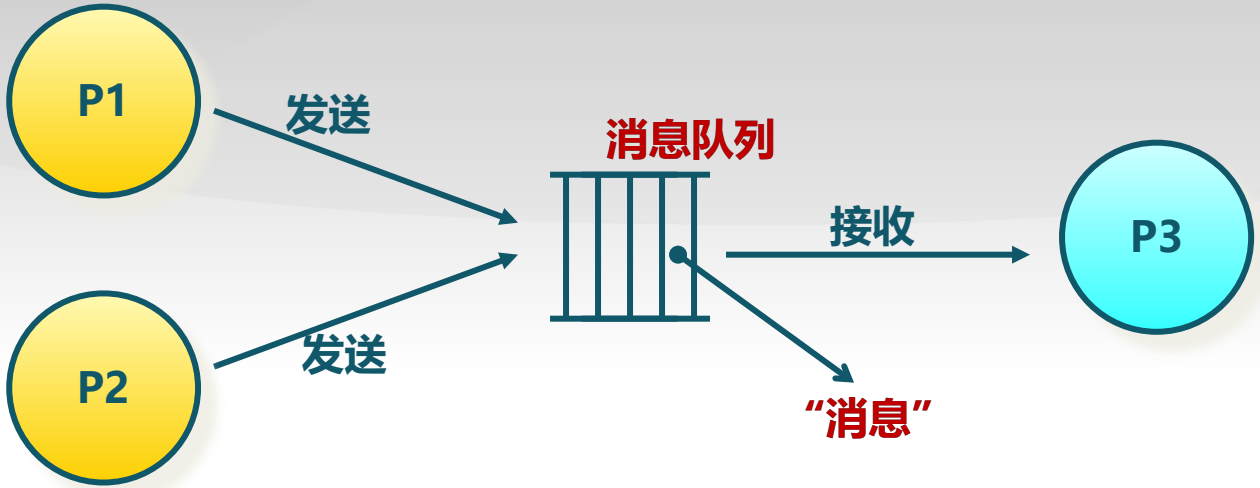


操作系统

Operating Systems

消息队列

- 消息队列是由操作系统维护的以字节序列为基本单位的间接通信机制
  - 每个消息(Message)是一个字节序列
  - 相同标识的消息组成按先进先出顺序组成一个消息队列（Message Queues）



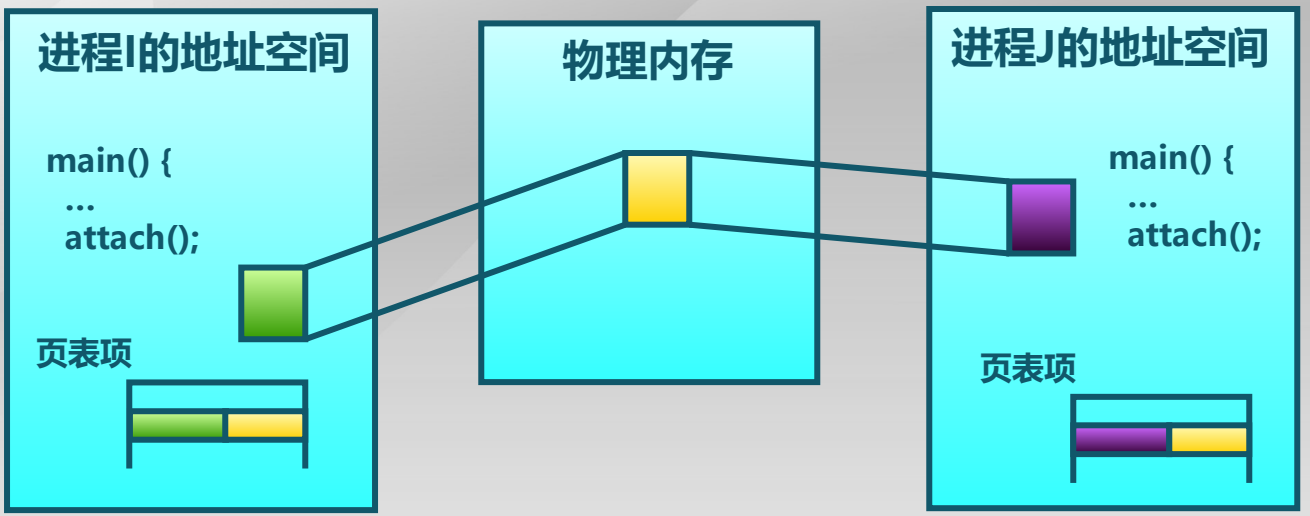
消息队列的系统调用

- msgget (key, flags)
  获取消息队列标识
- msgsnd (QID, buf, size, flags)
  发送消息
- msgrcv (QID, buf, size, type, flags)
  接收消息
- msgctl(...)
  消息队列控制

共享内存

- 共享内存是把同一个物理内存区域同时映射到多个进程的内存地址空间的通信机制
- 进程
  - 每个进程都有私有内存地址空间
  - 每个进程的内存地址空间需明确设置共享内存段
- 线程
  - 同一进程中的线程总是共享相同的内存地址空间
- 优点
  - 快速、方便地共享数据
- 不足
  - 必须用额外的同步机制来协调数据访问

共享内存的实现



- 最快的方法
- 一个进程写另外一个进程立即可见
- 没有系统调用干预
- 没有数据复制
- 不提供同步
  - 由程序员提供同步

共享内存系统调用

- shmget(key, size, flags)

  创建共享段
- shmat(shmid, *shmaddr, flags)

  把共享段映射到进程地址空间
- shmdt(*shmaddr)

  取消共享段到进程地址空间的映射
- shmctl(...)

  共享段控制
- 需要信号量等机制协调共享内存的访问冲突

第十五讲：死锁和并发错误检测

第 5 节：并发错误检测

向勇、陈渝

清华大学计算机系

xyong,yuchen@tsinghua.edu.cn

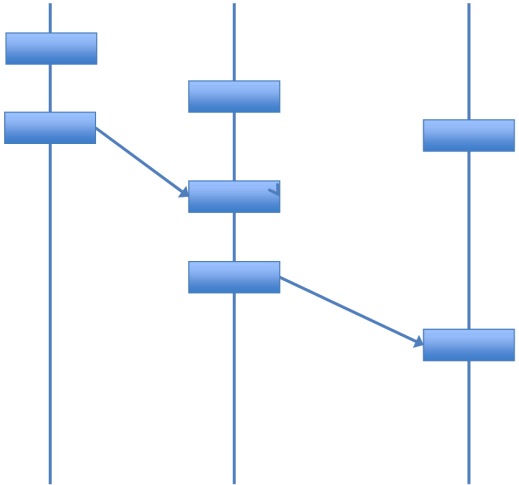
2020 年 5 月 5 日

提纲

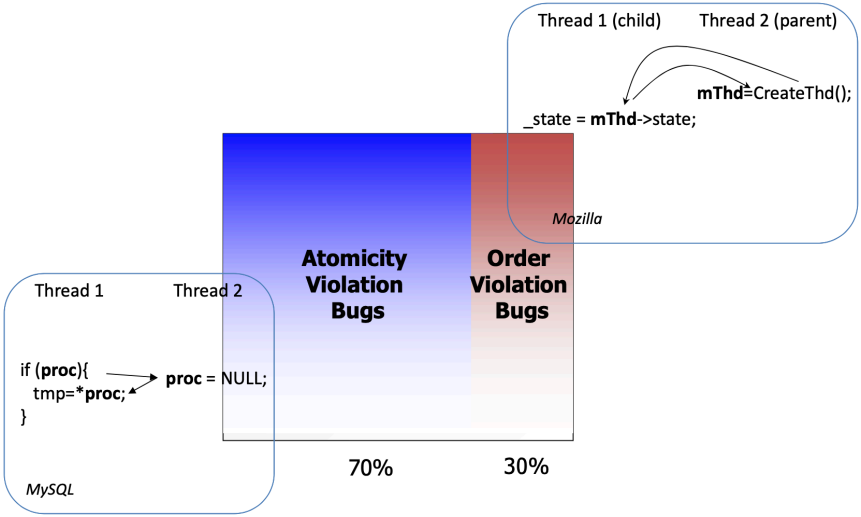
Ref: Shan Lu, Detecting and Fixing Concurrency Bugs, University of Chicago

Concurrency bug

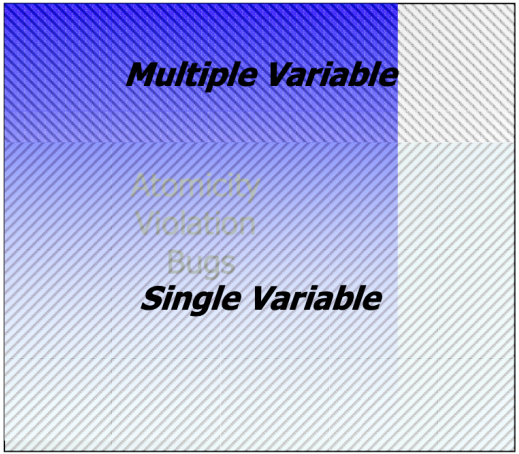
What ordering is guaranteed?



Concurrency bug: Voilation



Concurrency bug: Variable



Atomicity Violations

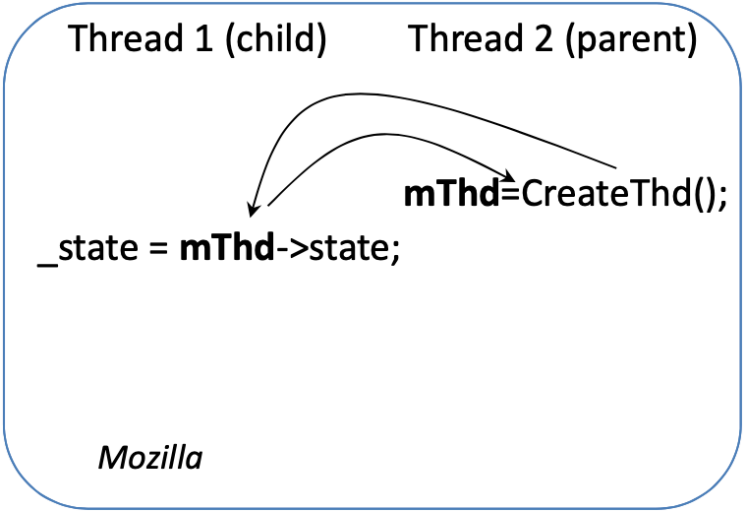
Thread 1	Thread 2

if (**proc**){
  tmp=***proc**;
}
	proc = NULL;

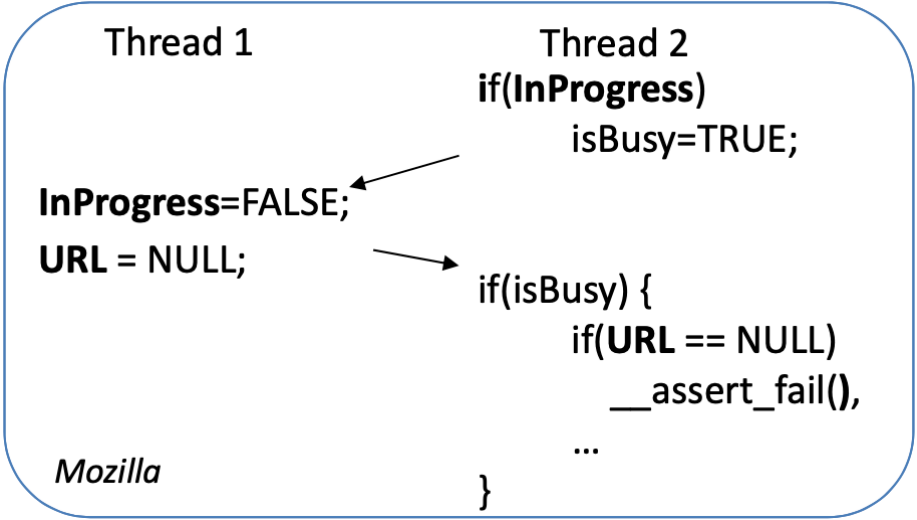
MySQL

Thread 1	Thread 2
while (!flag) {};	flag=TRUE;

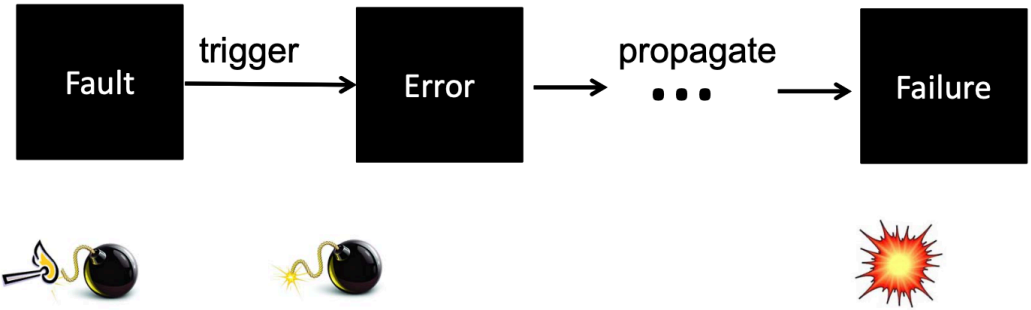
Order Violations



Multi-var Order Violations



The lifecycle of bugs



logical clock algorithm

Use logic time-stamps to find concurrent accesses

Thread 1	Thread 2	
	lock (L);	<0,1>
	ptr=NULL;	<0,2>
	unlock(L);	<0,3>

<1,0>**ptr** = malloc(10);
<2,3>lock (L);
<3,3>**ptr**[0]='a';
<4,3>unlock(L);

Lock-set algorithm

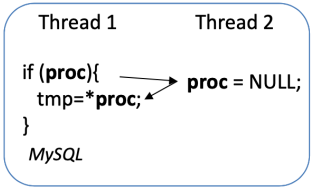
A common lock should protect all conflicting accesses to a shared variable

	Thread 1	Thread 2	
		lock (L);	
		ptr=NULL;	<L>
		unlock(L);	

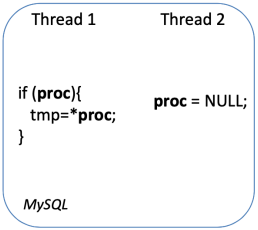
</>	**ptr** = malloc(10);	
	lock (L);	
<L>	**ptr**[0]='a';	
	unlock(L);	

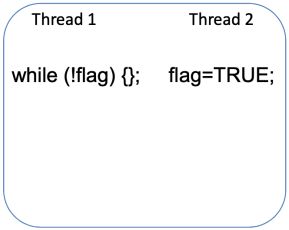
How to detect atomicity-violations?

Know which code region should maintain atomicity

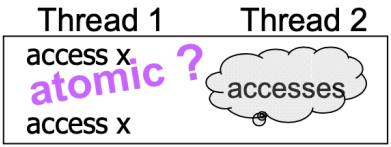


Judge whether a code region's atomicity is violated

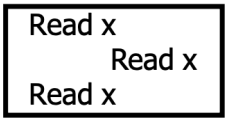


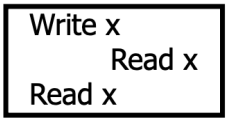


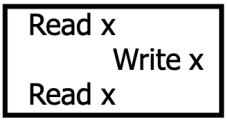
Atomicity violation = unserializable interleaving

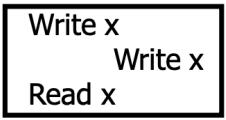


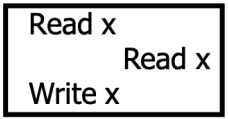
Totally 8 cases of interleaving

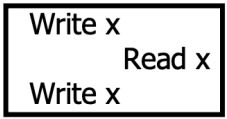


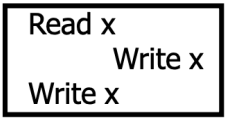


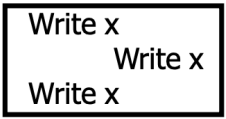




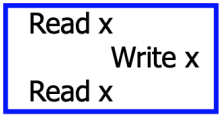


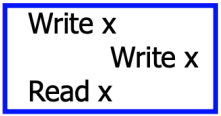


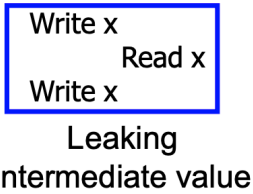


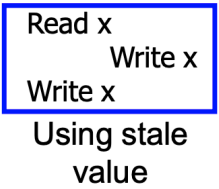


4 out of 8 cases are interleaving violations









Both hardware and software solutions exist

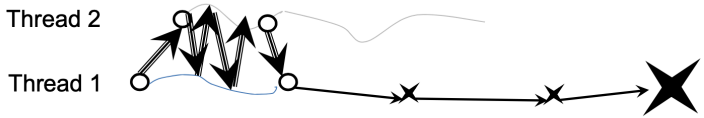
ConSeq & ConMem

If we cannot find a more accurate root-cause pattern, let's look at the effect patterns of concurrency bugs!

- ConMem
  - Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS' 10
- ConSeq
  - Detecting Concurrency Bugs through Sequential Errors, ASPLOS' 11

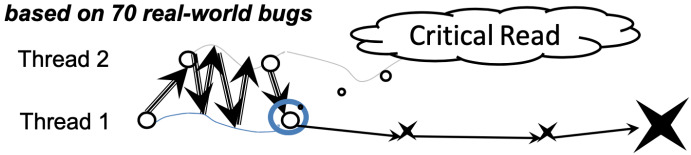
The lifecycle of concurrency bugs: Fault

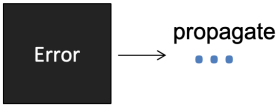
based on 70 real-world bugs



Data races
Atomicity violations
  single variable
  multiple variables
Order violations
…

The lifecycle of concurrency bugs: Error

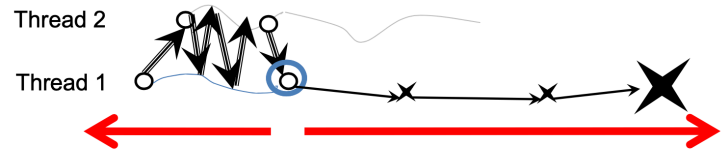




- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow
- Semantic errors

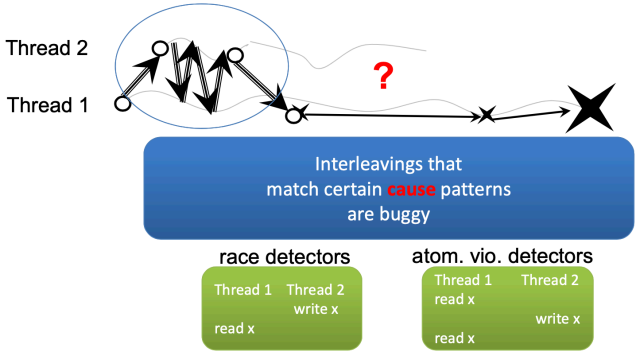
The lifecycle of concurrency bugs: Failure

based on 70 real-world bugs



- Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

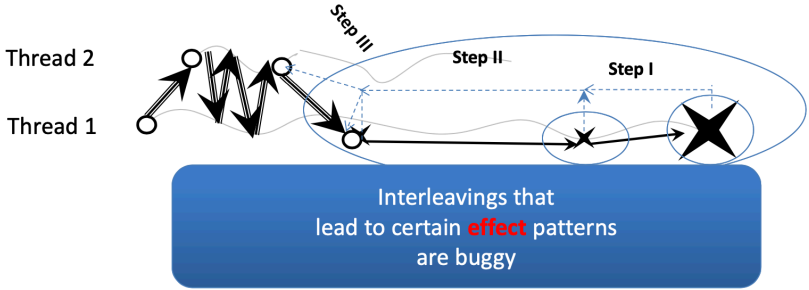
Cause-oriented approach



Limitations

- False positives
- False negatives

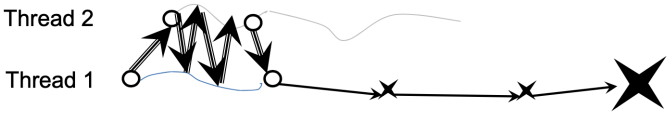
Effect-oriented approach

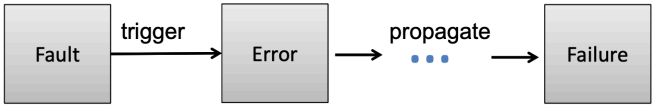


- Step 1: Statically identify potential failure/error site
- Step 2: Statically look for critical reads
- Step 3: Dynamically identify buggy interleaving

ConMem

Detecting Severe Concurrency Bugs through an Effect-Oriented Approach, ASPLOS' 10

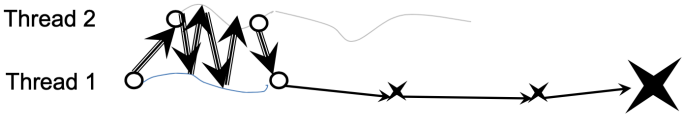


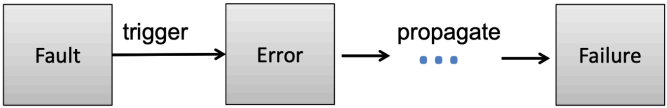


- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow
- Semantic errors

- Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages

ConSeq: Detecting Concurrency Bugs through Sequential Errors, ASPLOS'11





- Memory errors
  - NULL ptr
  - Dangling ptr
  - Uninitialized read
  - Buffer overflow
- Semantic errors

- Crash @ invalid memory
- Crash @ assertion
- Infinite loops
- Incorrect outputs
- Error messages