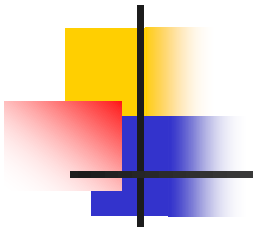


第9章 死锁

9.1 死锁的基本概念

9.2 死锁的检测与恢复

9.3 死锁避免

9.4 死锁预防

9.1 死锁的基本概念

一、什么是死锁（Deadlock）？

一个进程集合中的每个进程都在等待只能由该集合中的其它进程才能引发的事件，这种状态称作死锁。

一组竞争系统资源的进程由于相互等待而出现“永久”阻塞。

9.1 死锁的基本概念

为什么会出现死锁？

竞争资源，而资源有限。

例如，2个进程A、B，都需要资源R1、R2

若A：拥有R1，申请R2

若B：拥有R2，申请R1

如何？

9.1 死锁的基本概念

二、什么情况下会产生死锁？

4个必要条件：coffman(1971)年提出

1）互斥条件

每个资源要么被分配给了1个进程，要么空闲

2）占有及等待（部分分配）条件

已经得到了资源的进程要申请新的资源

3）不可剥夺条件

已经分配给一个进程的资源不能被剥夺，只能由占有者显式释放

4）环路等待条件

存在由2个或多个进程组成的一条环路，

该环路中的每个进程都在等待相邻进程占有的资源

9.1 死锁的基本概念

三、如何处理死锁？

1、由OS处理

- ✓ 检测死锁并恢复
- ✓ 分配资源时避免死锁
- ✓ 假装没看见（鸵鸟策略）：多数OS对待死锁的策略

2、由应用程序本身预防死锁

9.2 死锁的检测与恢复

一、死锁的检测方法

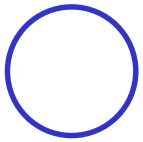
1. 每种资源只有1个

借助于**资源分配图**

方法：构造资源分配图，若存在环，则环中进程死锁。

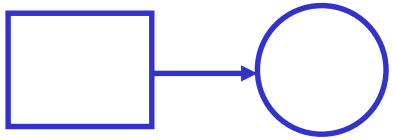
死锁的检测方法

资源分配图：

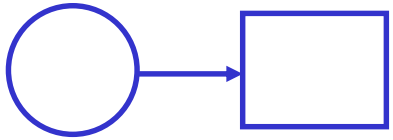


：进程

：资源



：进程已占有资源



：进程申请资源，处于等待状态

死锁的检测方法

【例】设系统中有进程7个：A~G，资源6种：R~W，每种资源只有1个。

资源占有及申请情况如下：

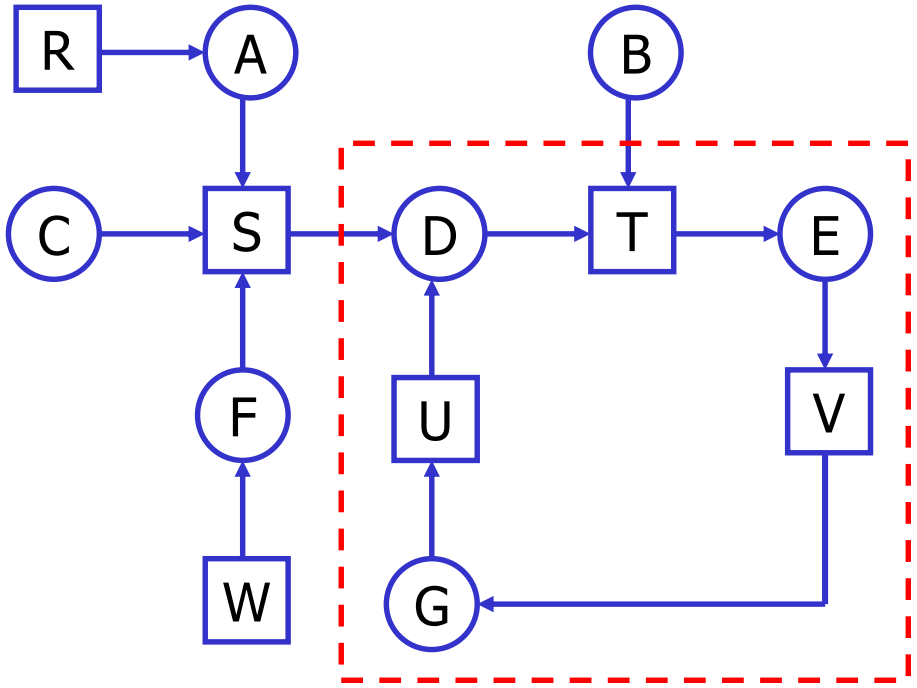
是否存在死锁？

若存在，涉及哪些进程？

进程	占有资源	申请资源
A	R	S
B		T
C		S
D	U	S, T
E	T	V
F	W	S
G	V	U

9.2 死锁的检测与恢复

构造资源分配图如下：



存在环。可以看出，

D、E、G死锁。

死锁的检测方法

2. 每种资源有多个

设有M种资源，N个进程

1）数据结构

E[M]：总资源数；E[i]：资源i的个数

A[M]：当前可用资源数；A[i]：资源i的可用数

C[N][M]：当前分配矩阵；C[i][j]：进程i对资源j的占有数

第i行是进程i当前占有的资源数

R[N][M]：申请矩阵；R[i][j]：进程i对资源j的申请数

第i行是进程i申请的资源数

F[N]：进程标记；F[i]取0/1，为1表示进程i能够执行

已分配资源数 + 可用资源数 = 总资源数

死锁的检测方法

2）算法

① 置F[0]~F[N]为0;

② 寻找一个满足下列条件的进程i：

F[i] == 0 且

R[i] <= A，即R[i][j] <= a[j], a<=j<=M //进程i申请的资源可满足

③ 若找不到这样的进程，则算法终止;

④ A = A + C[i];

F[i] = 1;

转②继续;

算法结束后，所有F[i]==0的进程(i)都会死锁。

该算法的思想：查找一个进程，使可用资源能满足该进程的请求；设想让该进程执行直到完成，再释放它占有的资源。

死锁的检测方法

【例】3个进程，4种资源。已知

总资源数E = (4, 2, 3, 1)；可用资源数A = (2, 1, 0, 0)

当前分配矩阵C = $\begin{pmatrix} 0, 0, 1, 0 \\ 2, 0, 0, 1 \\ 0, 1, 2, 0 \end{pmatrix}$　　申请矩阵R = $\begin{pmatrix} 2, 0, 0, 1 \\ 1, 0, 1, 0 \\ 2, 1, 0, 0 \end{pmatrix}$

检测步骤：

1）进程3可执行，释放其拥有的资源后，A=(2, 2, 2, 0)

2）进程2可执行，使 A = (4, 2, 2, 1)。

不存在死锁。

死锁的检测方法

如果分配矩阵C如下，其它不变，会死锁吗？

$$\text{当前分配矩阵C} = \begin{pmatrix} 0, 0, 2, 0 \\ 2, 0, 1, 1 \\ 0, 1, 0, 0 \end{pmatrix}$$

9.2 死锁的检测与恢复

二、何时检测死锁？

1）每当有资源请求时；

2）周期性检测；

3）每当CPU的使用率降到某一阈值时。

死锁检测会占用大量的CPU时间

9.2 死锁的检测与恢复

三、如何从死锁中恢复？

1）剥夺法恢复

将某一资源从一个进程抢占过来给另一个进程使用

不能影响原进程的最终执行结果

取决于资源的特性

2）回退法恢复

从各进程最近的检查点（check point）逐次重新启动

3）杀死进程来恢复

最好杀死可重复执行、不会产生副作用的进程

9.3 死锁避免

在保证安全的条件下分配资源。

在给每个进程分配资源前，先进行检查：

如果满足该资源请求肯定不会导致死锁，则予以分配；否则，拒绝。

9.3 死锁避免

一、安全状态与不安全状态

安全状态：

若在某一时刻，系统中存在进程序列<P1, P2, ..., Pn>，如果按此序列依次为每个进程Pi分配资源，使每个进程均可顺利完成。则称此时系统的状态为安全状态，称这样的一个进程序列<P1, P2, ..., Pn>为**安全序列**。

安全序列的实质是：对于每一个进程$P_i(1 \le i \le n)$，它以后尚需要的资源量不超过系统当前剩余资源量与所有进程P_j ($j < i$)当前占有资源量之和

若在某一时刻，系统中不存在一个安全序列，则称系统处于**不安全状态**。

9.3 死锁避免

【例】假定系统有三个进程P_1、P_2、P_3，共有12台磁带机。进程P_1总共要求10台磁带机，P_2和P_3分别要求4台和9台。设在T_0时刻，进程P_1、P_2和P_3已经获得5台、2台和2台，还有3台空闲没有分配。

进程	最大需求	已分配
P_1	10	5
P_2	4	2
P_3	9	2

T_0时刻系统是安全的。这时存在一个安全序列<P_2，P_1，P_3>

9.3 死锁避免

说明：

（1）系统在某一时刻的安全序列可能不唯一，但这不影响对系统安全状态的判断。

（2）**处于安全状态一定可以避免死锁，而系统处于不安全状态则仅仅可能进入死锁状态。**

9.3 死锁避免

二、银行家算法

银行家算法是最有代表性的避免死锁算法

是Dijkstra 1965年提出的。

基本思想：一个小城镇的银行家，他向一群客户分别承诺了一定金额的贷款，而他知道不可能所有客户同时都需要最大的贷款额。在这里，我们可将客户比作进程，银行家比作操作系统。银行家算法就是对每一个客户的请求进行检查，检查如果满足它是否会引起不安全状态。如果是，则不满足该请求；否则，便满足。

银行家算法

银行家算法的实质：

设法保证系统动态分配资源后不进入不安全状态，以避免可能产生的死锁。

每当进程提出资源请求且系统的资源能够满足该请求时，系统将判断如果满足此次资源请求，系统状态是否安全；

如果判断结果为安全，则给该进程分配资源，否则不分配资源，申请资源的进程将阻塞。

银行家算法

【例】已知4个进程：A、B、C、D，总资源数 = 10，进程对资源的最大需求量分别为6、5、4、7。

进程	已分配	最大需求
A	0	6
B	0	5
C	0	4
D	0	7

剩余：10

安全

安全序列：随意

进程	已分配	最大需求
A	1	6
B	1	5
C	2	4
D	4	7

剩余：2

安全

安全序列：C, B, A, D

进程	已分配	最大需求
A	1	6
B	2	5
C	2	4
D	4	7

剩余：1

不安全

银行家算法

银行家算法中所用的主要数据结构：

(1)可用资源向量Available

记录资源的当前可用数量。如果Available[j]==k，表示现有R_j类资源k个。

(2) 最大需求矩阵Max

每个进程对各类资源的最大需求量

(3)分配矩阵Allocation

已分配给每个进程的各类资源的数量

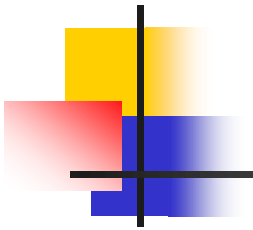
(4)需求矩阵Need

进程对各类资源尚需要的数量

Need = Max - Allocation

(5)请求向量Request

记录某个进程当前对各类资源的申请量



银行家算法

当Pi发出资源请求后，系统按下列步骤进行检查：

(1)如果Request > Need[i]，则出错;

(2)如果Request > Available，则Pi阻塞;

(3)假设把要求的资源分配给进程Pi，则有

Available = Available – Request;

Allocation[i] = Allocation[i] + Request;

Need[i] = Need[i] - Request;

检查此次资源分配后，系统是否处于安全状态。若安全，正式将资源分配给进程Pi，以完成本次分配；否则，恢复原来的资源分配状态，让进程Pi等待。

银行家算法

为进行安全性检查，定义数据结构：

① 工作向量Work

表示系统可提供给进程的各类资源的数量

开始时，Work = Available

② 向量Finish

标记进程是否可运行结束。开始时先令Finish[i] = false；当有足够的资源分配给进程时，令Finish[i] = true

银行家算法

安全性检查的步骤：

(1) Work = Available;

对于每个i，置Finish[i] = false;

(2) 寻找满足条件的i：

Finish[i] = false，且Need[i]≤Work;

如果不存在，则转(4);

(3) 假设把资源分配给进程Pi，Pi可以运行到结束并释放资源

Work = Work + Allocation[i];

Finish[i] = true;

转(2);

(4) 若对所有i，Finish[i]都等于true，则系统处于安全状态，否则处于不安全状态。

银行家算法

银行家算法虽然可以避免死锁，但**缺乏实用价值**，

因为**需要事先知道每个进程对每种资源的最大需求**。

很少有进程能够在运行前就知道其所需资源的最大值，而且进程数也不是固定的，往往在不断地变化（如新用户登录或退出），况且原本可用的资源也可能突然间变成不可用（如磁带机可能坏掉）。因此，在实际中，如果有，也只有极少的系统使用银行家算法来避免死锁。

9.4 死锁预防

在系统（用户程序）设计时考虑死锁问题

基本做法：

使产生死锁的四个必要条件至少有一个不成立

9.4 死锁预防

一、破坏互斥条件

互斥是由资源的固有特性决定的，是不能改变的。

可利用的思想：

对资源统一管理，如SPOOLing技术。

9.4 死锁预防

二、破坏不可剥夺条件

当进程申请的资源被其他进程占用时，可以通过操作系统抢占这一资源。

当资源状态很容易保存并恢复时，这种方法是实用的。

举个剥夺的例子：

换入/换出（剥夺内存）

进程切换（剥夺CPU）

9.4 死锁预防

三、破坏占有及等待条件

基本思想：

原子性地获得所需全部资源。

2种方法：

（1）每个进程开始执行前一次性地申请它所要求的所有资源，且仅当该进程所要资源均可满足时才给予一次性分配

（2）当进程申请资源时，先暂时释放其当前占有的所有资源，然后再试图一次申请获得所需的全部资源

9.4 死锁预防

四、破坏环路等待条件

采用资源有序分配法：

把系统中所有资源编号

每个进程只能按编号的递增次序申请资源

存在的问题：有时不容易给出合适的资源编号，特别是资源数量庞大时

破坏环路等待条件

例如：1，2，3，…，10