

第7章 死锁

孙承杰

E-mail: sunchengjie@hit.edu.cn

哈工大计算学部人工智能教研室

2023年秋季学期

主要内容

1. 死锁的概念
2. 死锁特征分析
 - 产生死锁的4个必要条件
3. 死锁处理方法
 - (1) 死锁预防
 - (2) 死锁避免
 - (3) 死锁检测
 - (4) 死锁恢复

生产者 – 消费者的信号量解法



■ 不合理的信号量使用会导致...

```
Producer(item) {  
    mutex.P();  
    emptyBuffers.P();  
    Enqueue(item);  
    mutex.V();  
    fullBuffers.V();  
}
```

```
Consumer() {  
    mutex.P();  
    fullBuffers.P();  
    item = Dequeue();  
    mutex.V();  
    emptyBuffers.V();  
}
```

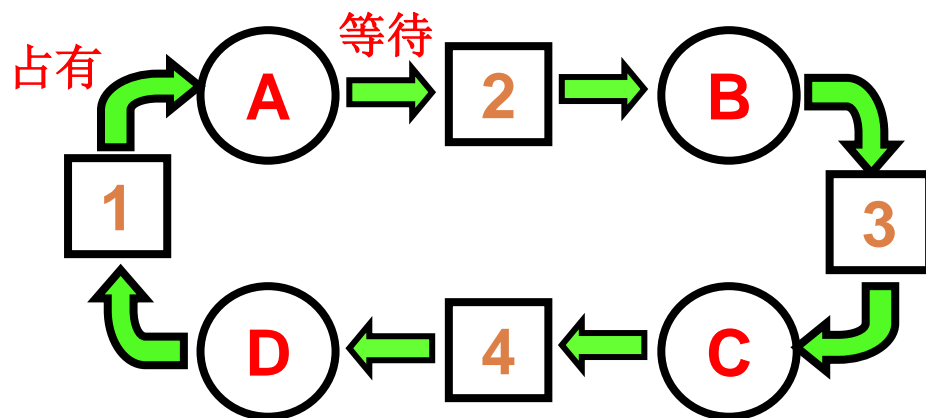
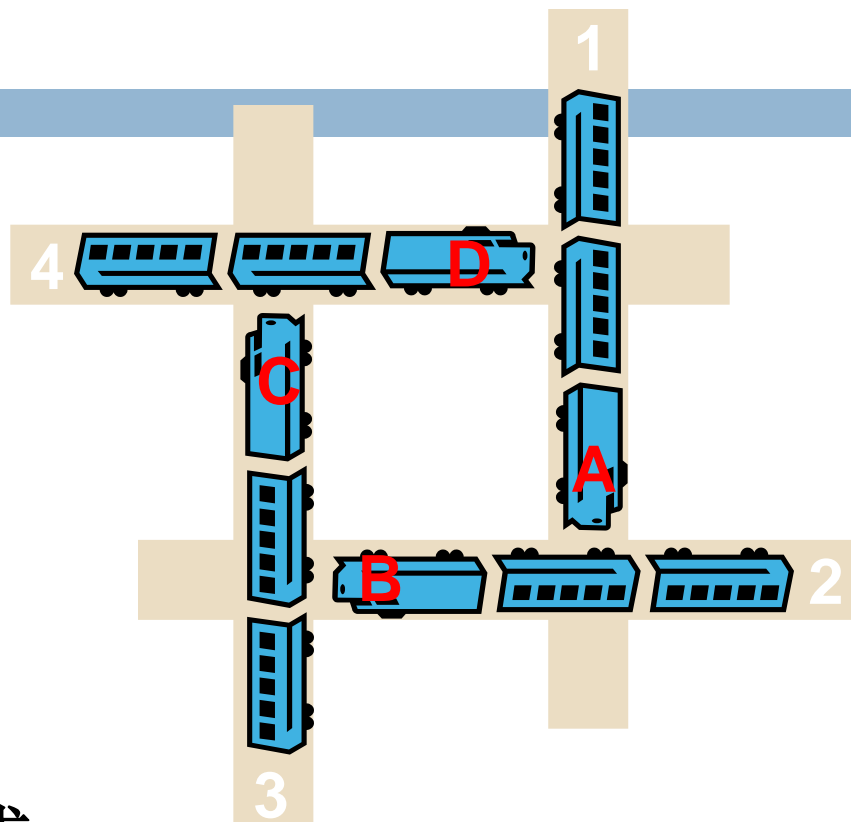
- 生产者占有mutex信号量后又阻塞在emptyBuffers信号量上。而消费者阻塞在mutex上，不能唤醒生产者.....最后谁也没法执行.....

死锁现象

■ 看一个实际的例子

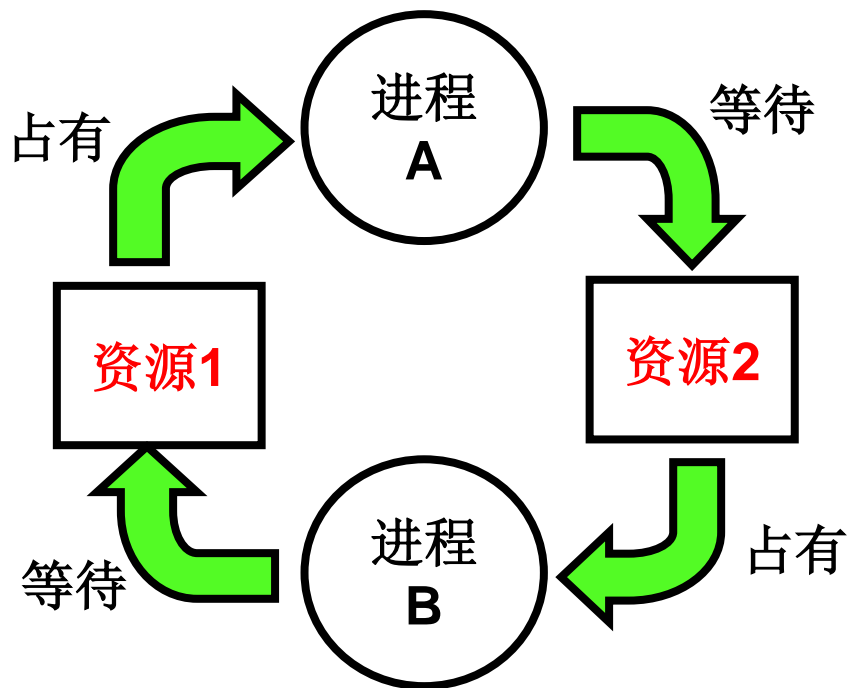
■ 现在分析这个例子

- 竞争使用资源: 道路
- **A**占有道路**1**，又要请求道路**2**，**B**占有...
- 形成了无限等待



死锁概念(Deadlock)

- **死锁：多个进程因循环等待资源而造成无法执行的现象。**



- 死锁会造成进程无法执行
- 死锁会造成系统资源的极大浪费(资源无法释放、资源无法有效利用)

死锁概念(Deadlock)

- **死锁:** 多个进程因循环等待资源而造成无法执行的现象。

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

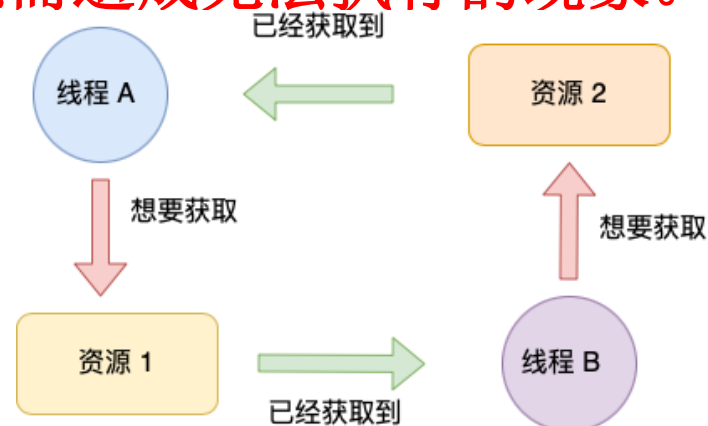
pthread_mutex_t mutex_A = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex_B = PTHREAD_MUTEX_INITIALIZER;

//线程函数 A
void *threadA_proc(void *data)
{
    printf("thread A waiting get ResourceA \n");
    pthread_mutex_lock(&mutex_A);
    printf("thread A got ResourceA \n");

    sleep(1);

    printf("thread A waiting get ResourceB \n");
    pthread_mutex_lock(&mutex_B);
    printf("thread A got ResourceB \n");

    pthread_mutex_unlock(&mutex_B);
    pthread_mutex_unlock(&mutex_A);
    return (void *)0;
}
```



```
//线程函数 B
void *threadB_proc(void *data)
{
    printf("thread B waiting get ResourceB \n");
    pthread_mutex_lock(&mutex_B);
    printf("thread B got ResourceB \n");

    sleep(1);

    printf("thread B waiting get ResourceA \n");
    pthread_mutex_lock(&mutex_A);
    printf("thread B got ResourceA \n");

    pthread_mutex_unlock(&mutex_A);
    pthread_mutex_unlock(&mutex_B);
    return (void *)0;
}
```

死锁概念(Deadlock)

- **死锁:** 多个进程因循环等待资源而造成无法执行的现象。

```
[root@localhost deadloak_prac]# gcc -o deadlock -lpthread deadlock.c
[root@localhost deadloak_prac]# ./deadlock
thread A waiting get ResourceA
thread A got ResourceA
thread B waiting get ResourceB
thread B got ResourceB
thread A waiting get ResourceB
thread B waiting get ResourceA
```

https://blog.csdn.net/www_dong

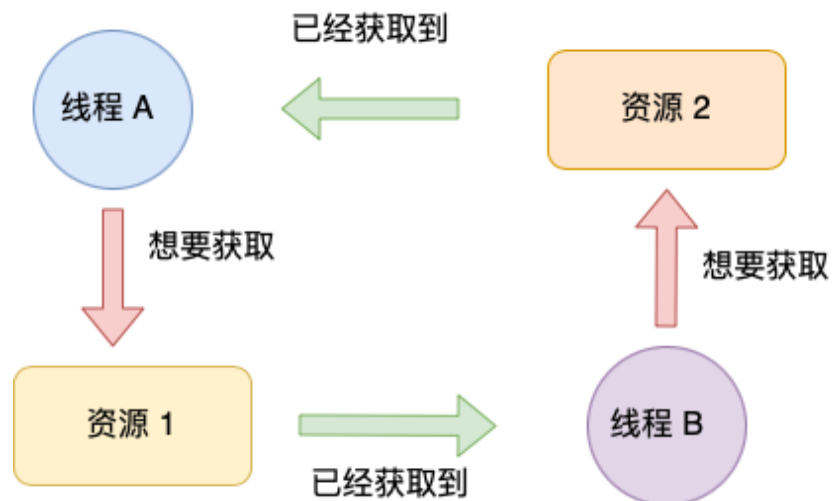
```
int main()
{
    pthread_t tidA, tidB;

    //创建两个线程
    pthread_create(&tidA, NULL, threadA_proc, NULL);
    pthread_create(&tidB, NULL, threadB_proc, NULL);

    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);

    printf("exit\n");

    return 0;
}
```





**为什么会产生死锁问题？
我们来分析一下！**

资源的分析

- 多个进程因等待**资源**才造成死锁
 - **资源**: 进程在完成其任务过程所需要的所有对象
 - CPU、内存、磁盘块、外设、文件、信号量 ...
 - 显然有些资源不会造成死锁，而有些会
 - 只读文件是不会造成进程等待的，也就不会死锁
 - 打印机一次只能让一个进程使用，就会造成死锁
- 称为互斥访问资源
- 显然，资源互斥访问是死锁的必要条件

死锁并不总是发生

■ 一简单实例

进程 A

x.P();

y.P();

y.V();

x.V();

进程 B

y.P();

x.P();

x.V();

y.V();

- 考虑序列: **A:x.P(), B:y.P(), B:x.P(), A:y.P()...**

形成循环等待, 出现死锁

- 考虑序列: **A:x.P(), A:y.P(), B:y.P()...**

- 死锁与调度有关, 是非确定的!

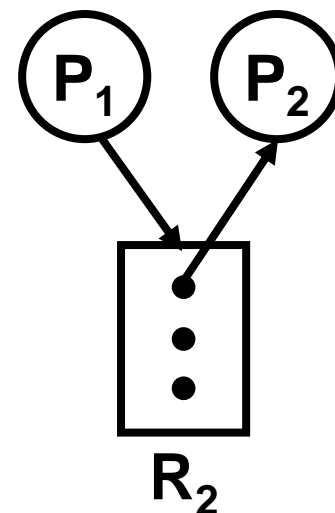
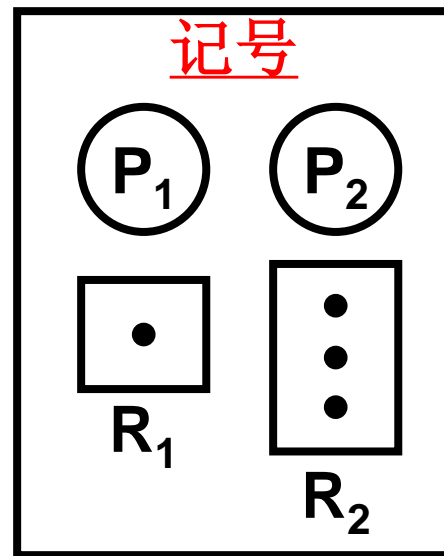
并不形成死锁

- 资源请求需要形成环路等待才死锁, 如何描述这种等待关系?

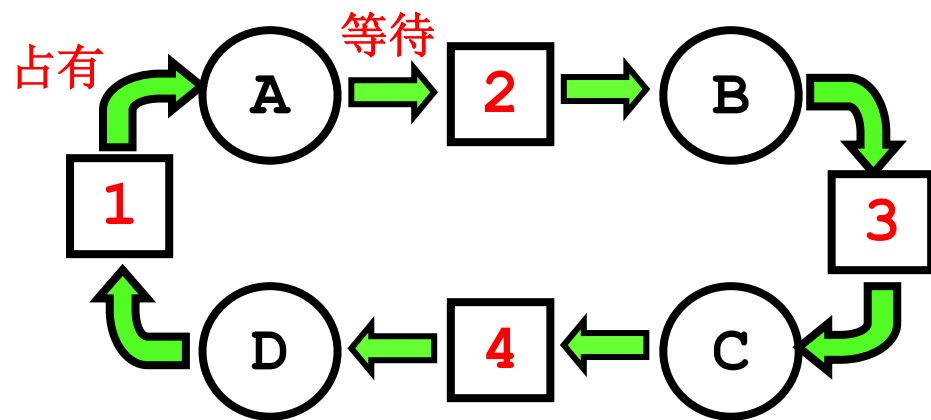
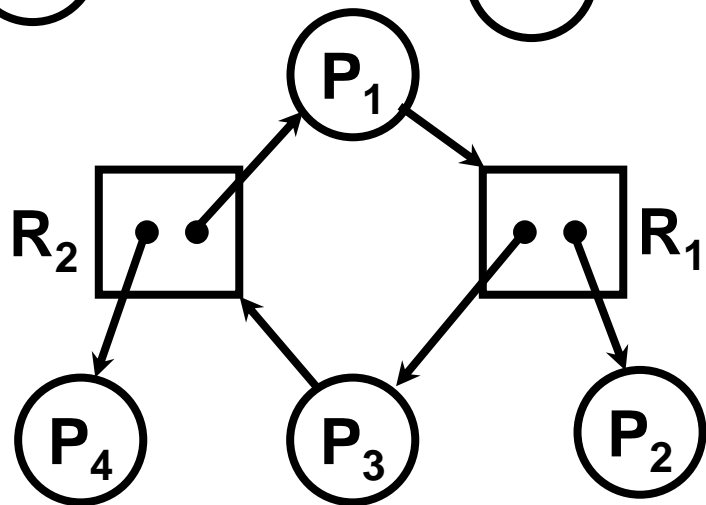
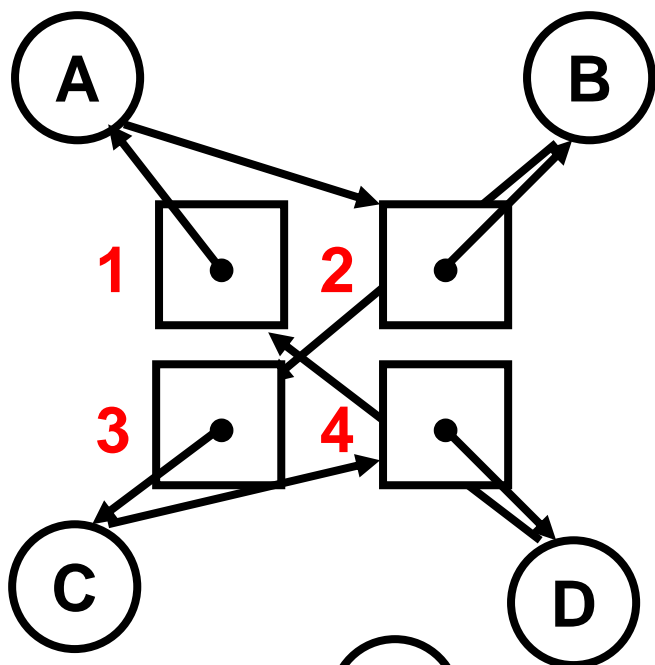
资源分配图

■ 资源分配图模型

- 一个进程集合 $\{P_1, P_2, \dots, P_n\}$
- 一资源类型集合 $\{R_1, R_2, \dots, R_m\}$
- 资源类型 R_i 有 W_i 个实例
- 资源请求边：有向边 $P_i \rightarrow R_j$
- 资源分配边：有向边 $R_i \rightarrow P_k$



资源分配图实例



■ 存在环路:

1 → A → 2 → B → 3 → C → 4 → D → 1

■ 产生死锁

■ 存在环路:

$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

■ 但并不死锁，仍可继续执行

死锁的4个必要条件

■ 互斥使用(Mutual exclusion)

- 至少有一个资源互斥使用

■ 不可抢占(No preemption)

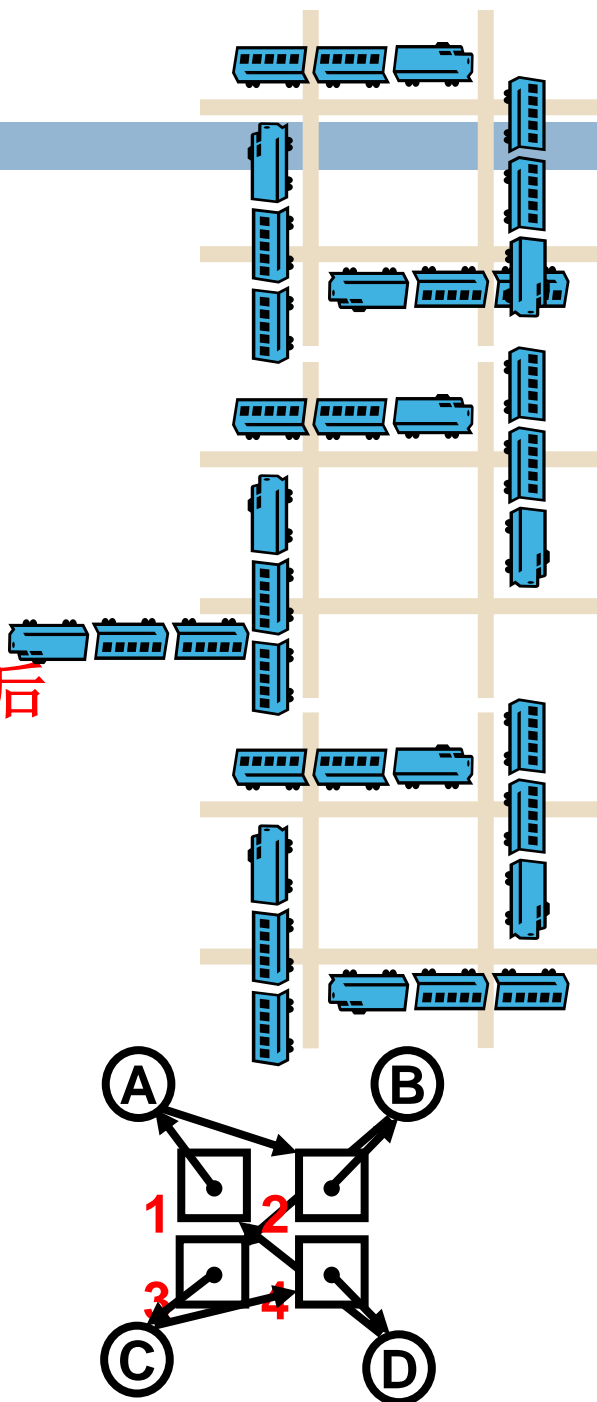
- 资源只能自愿放弃，如车开走以后

■ 请求和保持(Hold and wait)

- 进程必须占有资源，再去申请

■ 循环等待(Circular wait)

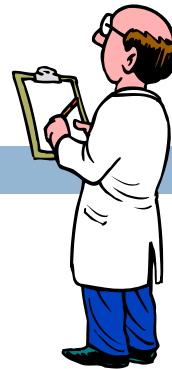
- 在资源分配图中存在一个环路





如何消除死锁？ 有什么方法？

死锁处理方法概述



- **死锁预防** “no smoking”，预防火灾
 - 破坏死锁的必要条件
- **死锁避免** 检测到煤气超标时，自动切断电源
 - 检测每个资源请求，如果造成死锁就拒绝
- **死锁检测+恢复** 发现火灾时，立刻拿起灭火器
 - 检测到死锁出现，剥夺进程的资源或结束进程
- **死锁忽略** 小火不会带来什么损失
 - 就好像没有出现死锁一样

死锁预防: 破除死锁的必要条件之(1)(2)

■ 破坏互斥使用

- 资源的固有特性，通常无法破除，如打印机

■ 破除不可抢占

- 如果一个进程占有资源并申请另一个不能立即分配的资源，那么已分配资源就可被抢占。
- 如果申请的资源得到满足，则抢占其他资源一次性分配给该进程
- 只对状态能保存和恢复的资源(如CPU，内存空间)有效，对打印机等外设不适用

实例：两个进程使用串口，都要读串口，数据不同不可恢复。

死锁预防: 破除死锁的必要条件之(3)

■ 破除请求和保持

- 在进程执行前，**一次性申请所有需要的资源**
- 缺点1: 需要预知未来，编程困难
- 缺点2: 许多资源分配后很长时间后才使用，资源利用率低

死锁预防: 破除死锁的必要条件之(4)

■ 破除循环等待

- 对资源类型进行排序，资源申请必须按序进行
- 例如：所有的进程必须先申请磁盘驱动，再申请打印机，再....，如同日常交通中的单行道
- 缺点：编程时就需考虑；可能先需要释放某些资源(申请序号小的资源)，进程可能会无法执行
P1:1,2,3 ;
P2:2,3 ; P3:3,4 (4被长时间使用，不释放3，则P1\P2无法执行)

- 总之，破除死锁的必要条件会引入不合理因素，实际中很少使用。

死锁避免

不死锁就成了问题的核心!

- 思想: 判断此次请求**是否造成死锁**

若会造成死锁, 则拒绝该请求

- **安全状态定义:** 如果系统中的所有进程存在一个可完成的执行序列 P_1, \dots, P_n , 则称系统处于安全状态

都能执行完成当然就不死锁

- **安全序列:** 上面的执行序列 P_1, \dots, P_n

如何找到这样的序列?

死锁避免之银行家算法

一个银行家：目前手里只有1亿，但是已经贷出很多钱

开发商A：已贷款15亿，资金紧张还需3亿。

开发商B：已贷款5亿，还需贷款1亿，运转良好能收回。

开发商C：已贷款2亿，欲贷款18亿

会不会出现楼盘烂尾？

开发商B还钱，再借给A，则可以继续借给C

银行家当前手里现金（Available）；

银行家可以利用的资金，即手里现金加上能收回的共有多少（work）；

各个开发商已贷款——已分配的资金（Allocation）；

各个开发商还需要贷款（need）

钱就是资源，开发商就是进程，银行家的决策就是调度

死锁避免之银行家算法

- 生成安全序列 P_1, \dots, P_n 应该满足的条件:

$P_i (1 \leq i \leq n)$ 需要资源 \leq 剩余资源 + 分配给 $P_j (1 \leq j < i)$ 资源

Banker()

int n, m; //系统中进程总数n和资源种类总数m

int Available[1..m]; //资源当前可用总量

int Allocation[1..n, 1..m];

//当前分配给每个进程的各种资源数量

int Need[1..n, 1..m];

//当前每个进程还需分配的各种资源数量

int Work[1..m]; //当前可分配的资源, 包括可收回的

bool Finish[1..n]; //进程是否能放入安全序列-结束

死锁避免之银行家算法

■ 安全状态判定（思路），寻找安全序列

将对n个进程集合进行多次扫描，构建出P1-Pn的安全序列

①初始化设定：

Work = Available （动态记录当前可（收回）分配资源）

Finish[i]=false （设定所有进程均未完成）

②查找这样的进程 P_i （未完成但目前可用资源可满足其需要，这样的进程是能够完成的）：

a) Finish[i] = false b) Need[i] ≤ Work

如果没有这样的进程 P_i ，则跳转到第④步

③（若有则） P_i 一定能完成，并归还其占用的资源，即：

a) Finish[i] = true b) Work = Work + Allocation[i]

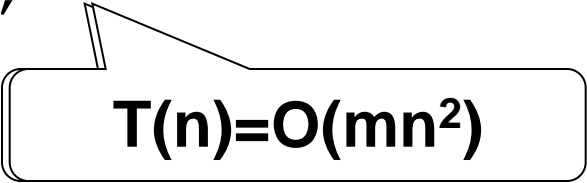
GOTO 第②步，继续查找

④如果所有进程 P_i 都是能完成的，即Finish[i]=ture

则系统处于安全状态，否则系统处于不安全状态

死锁避免之银行家算法

```
Boolean Found;  
Work = Available; Finish[1..n] = false;  
while(true) {  
    Found = false; //是否为安全序列 找到一个新进程  
    for(i=1; i<=n; i++){  
        if(Finish[i]==false && Need[i]<=Work) {  
            Work = Work + Allocation[i];  
            Finish[i] = true;  
            Found = true;  
        }  
    } 没有安全序列或已经找到  
    if(Found==false) break;  
}  
for(i=1; i<=n; i++){  
    if(Finish[i]==false) return "deadlock";  
}
```


$$T(n)=O(mn^2)$$

最好情形：安全状态就是p1-pn
最坏情形：pn-p1

死锁避免之银行家算法实例

■ 当前状态:

		<u>Allocation</u>	<u>Need</u>	<u>Available</u>
		A B C	A B C	A B C
	Work=[3 3 2]	P0 0 1 0	7 4 3	3 3 2
P_1	Work=[5 3 2]	P1 2 0 0	1 2 2	
P_3	Work=[7 4 3]	P2 3 0 2	6 0 0	
P_4	Work=[7 4 5]	P3 2 1 1	0 1 1	
P_0	Work=[7 5 5]	P4 0 0 2	4 3 1	
P_2	Work=[10 5 7]			

■ 安全序列是 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

■ 安全序列是唯一的吗?

死锁避免之资源请求算法

思想：可用的资源可以满足某个进程的资源请求，则分配；然后用银行家算法寻找安全序列，找到分配成功，找不到已分配资源收回。

```
extern Banker();  
int Request[1..m]; /*进程 $P_i$ 的资源申请*/  
if (Request > Need[i]) return "error";  
if (Request > Available) sleep();  
Available = Available - Request;  
Allocation[i] = Allocation[i] + Request;  
Need[i] = Need[i] - Request;  
    /*先将资源分配给 $P_i$ */  
if (Banker() == "deadlock")  
    /*调用银行家算法判定是否会死锁*/  
    拒绝Request; /*若算法判定deadlock则  
        拒绝请求，资源回滚*/
```



死锁避免之资源请求实例(1)

■ P_1 申请资源(1,0,2)

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

分配后

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

分配前

- 序列 $\langle P_1, P_3, P_2, P_4, P_0 \rangle$ 是安全的
- 此次申请允许

死锁避免之资源请求实例(2)

■ P_0 再申请(0,2,0)

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	3	0	7	2	3	2	1	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

分配后

	<u>Allocation</u>			<u>Need</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

分配前

- 进程 P_0 , P_1 , P_2 , P_3 , P_4 一个也没法执行, 死锁进程组
- 此次申请被拒绝

银行家算法讨论：

- 每个进程进入系统时必须告知所需资源的最大数量
对应用程序员要求高
- 安全序列寻找算法（安全状态判定算法）计算时间
复杂度为 $O(mn^2)$ ，过于复杂
- 若每次资源请求都要调用银行家算法，耗时过大，
系统效率降低
- 采用此算法，存在情况：当前有资源可用，尽管可能很快就会释放，由于会使整体进程处于不安全状态，而不被分配，致使资源利用率大大降低

死锁检测+恢复: 死锁检测

- 基本原因: 每次申请都执行 $O(mn^2)$, 效率低
- 对策: 只要可用资源足够, 则分配, **发现问题再处理**
- 定时检测或者当发现资源利用率低时检测

```
Finish[1..n] = false;  
  
if (Allocation[i] == 0) Finish[i]=true;  
if (available < x) ... // 执行Banker算法  
for (i=1; i<=n; i++)  
    if (Finish[i]==false)  
        deadlock = deadlock + {i};
```

对于无分配资源的进程, 不论其是否获得请求资源, 则认为其是完成的, 之后调用一次Banker算法, 就能找出所有死锁进程

死锁检测+恢复: 死锁恢复

- 终止进程 选谁终止?
 - 优先级? 占用资源多的? ...
- 剥夺资源 进程需要回滚 (rollback)
 - 回滚点的选取? 如何回滚? ...

死锁忽略

■ 死锁预防？

- 引入太多不合理因素...

■ 死锁避免？

- 每次申请都执行银行家算法 $O(mn^2)$ ，效率太低

■ 死锁检测+恢复？

- 观测进程状态，定期检测，恢复较为复杂

■ 死锁忽略：对死锁不在意，重启OS

- 大多数操作系统都用它，如windows\linux\UNIX
- 死锁出现不是确定的，可以用重启来处理死锁

鸵鸟策略：出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价，还不如不做处理，OS中这种置之不理的策略称之为鸵鸟策略，是平衡性能和复杂性而选择的一种方法。

死锁总结

- 进程竞争资源 \Rightarrow 有可能形成循环竞争 \Rightarrow 死锁
- 死锁需要处理 \Rightarrow 死锁分析 \Rightarrow 死锁的必要条件
- 死锁处理 \Rightarrow 预防、避免、检测+恢复、忽略
- 死锁预防: 破除必要条件 \Rightarrow 引入了不合理因素
- 死锁避免: 用银行家算法找安全序列 \Rightarrow 效率太低
- 死锁检测恢复: 银行家算法找死锁进程组并恢复 \Rightarrow 实现较难
- 死锁忽略: 就好像没有死锁 \Rightarrow 现在用的最多