哈爾濱工業大學（深圳）

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

操作系统 (Operating System)

第五章：死锁

陈芳林　副教授

哈尔滨工业大学（深圳）

2024年秋

Email：chenfanglin@hit.edu.cn

本章的主要内容

- 死锁（Deadlock）的引入
- 死锁特征分析—产生死锁的四个必要条件
- 死锁处理方法

生活中的死锁

- 案例一：疫情期间，张三的手机摔坏了，因为急用，就想去益田假日买一部新的。这时候保安拦住了张三，由于深圳近期疫情，保安让张三出示健康码。张三说手机摔坏了，要进去买一部新手机才能出示健康码；保安说，最近疫情严重了，要出示健康码才能进去买手机。。。哇，多么完美的死锁！

- 案例二：春晚小品（开锁）

信号量产生死锁

```
1. volatile long cnt = 0; /* Counter */
2.  sem_t mutex[2]; /* Semaphore that pr
3.  int NITERS;

5.  int main()
6. {
7.      pthread_t tid[2];

9.       NITERS = atoi(argv[1]);

11.      Sem_init(&mutex[0], 0, 1);   /* mutex[0] = 1 */
12.      Sem_init(&mutex[1], 0, 1);   /* mutex[1] = 1 */
13.      Pthread_create(&tid[0], NULL, count, (void*) 0);
14.      Pthread_create(&tid[1], NULL, count, (void*) 1);
15.      Pthread_join(tid[0], NULL);
16.      Pthread_join(tid[1], NULL);
17.      printf("cnt=%d\n", cnt);
18.      exit(0);
19.}
```

deadlock.c

```
20.void *count(void *vargp)
21.{
22.   int i;
23.   int id = (int) vargp;
24.   for (i = 0; i < NITERS; i++) {
25.     P(&mutex[id]); P(&mutex[1-id]);
26.     cnt++;
27.     V(&mutex[id]); V(&mutex[1-id]);
28.   }
29.   return NULL;
30.}
```

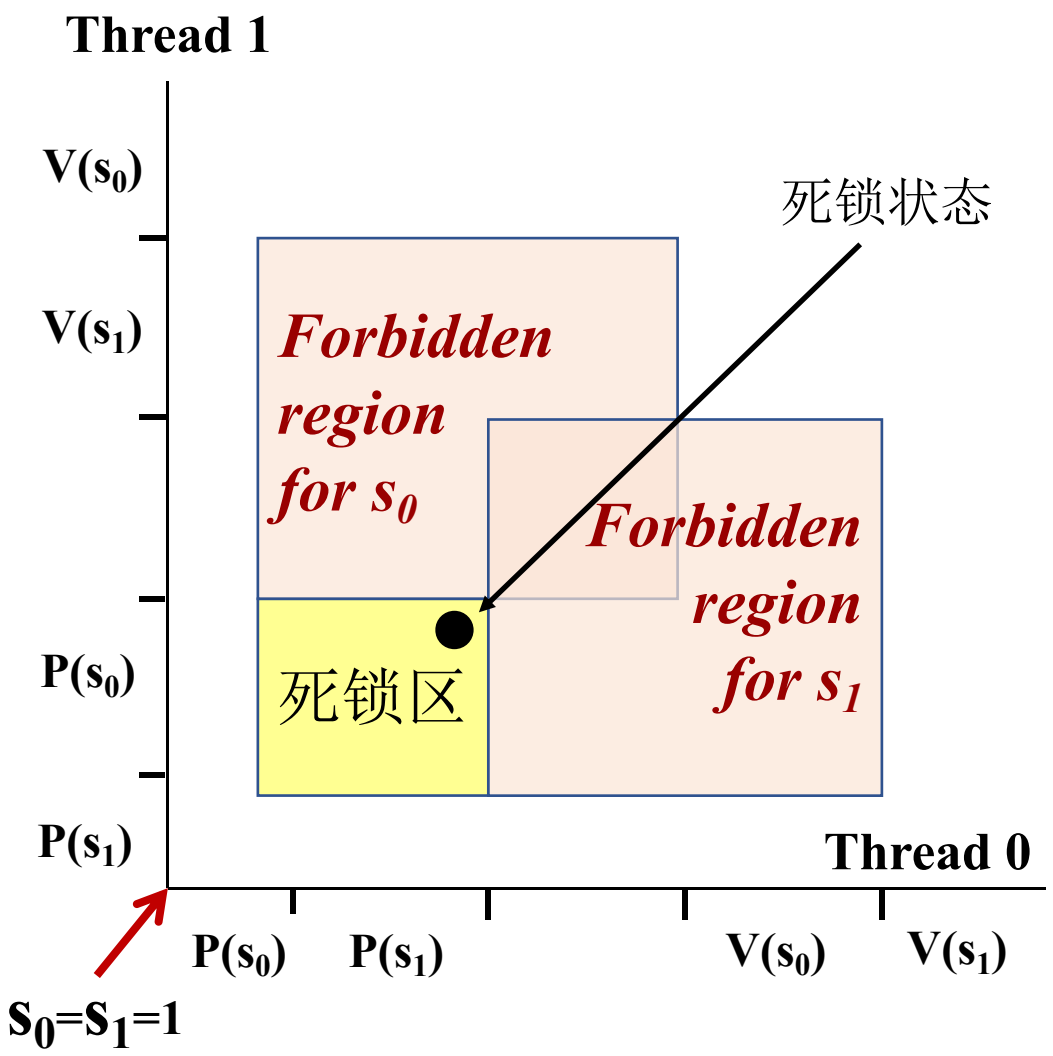
Tid[0]:	Tid[1]:
$P(s_0)$;	$P(s_1)$;
$P(s_1)$;	$P(s_0)$;
cnt++;	cnt++;
$V(s_0)$;	$V(s_1)$;
$V(s_1)$;	$V(s_0)$;

```
lpsh@ubuntu:~/Desktop/Files$ ./deadlock 1000
cnt=2000
lpsh@ubuntu:~/Desktop/Files$ ./deadlock 1000
```

正确执行

发生死锁

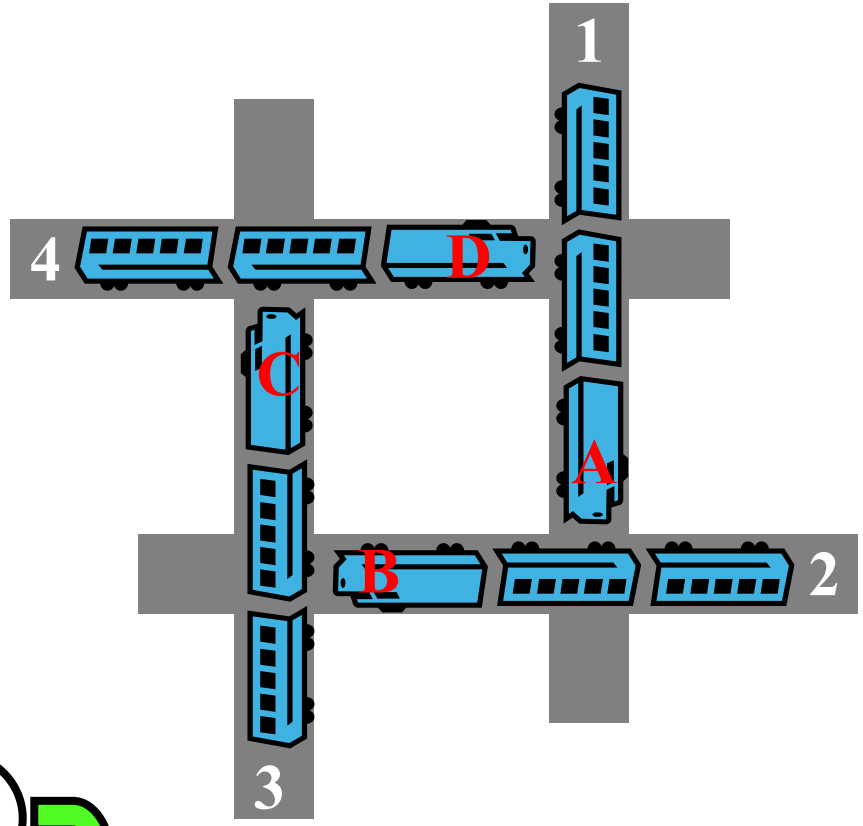
利用进度图可视分析死锁

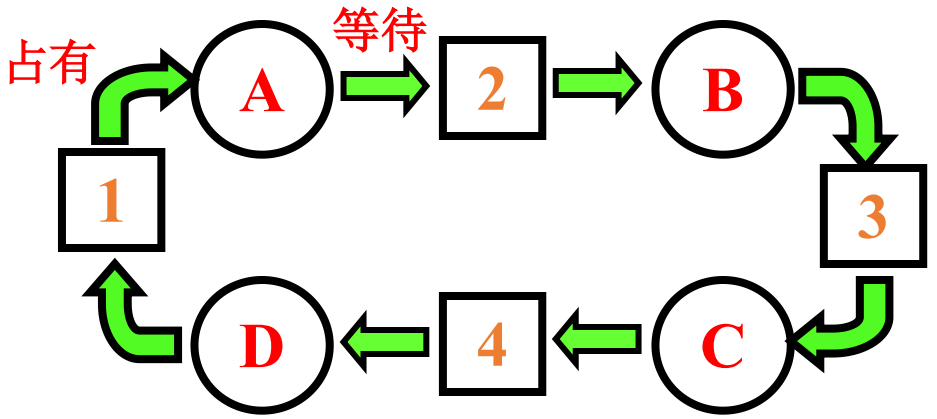


- 上锁带来了潜在的死锁 (deadlock)：即等待一个永远不会为真的条件。
- 任何进入死锁区(deadlock region) 的进程推进顺序都会最终进入死锁状态(deadlock state)，等待s_0或s_1变为非0。
- 其他进程推进顺序则可以避免进入死锁区。比较棘手的是，死锁经常是非确定的(nondeterministic)
- 不同的执行轨迹会造成不同的执行结果——死锁的产生与调度有关！

死锁现象

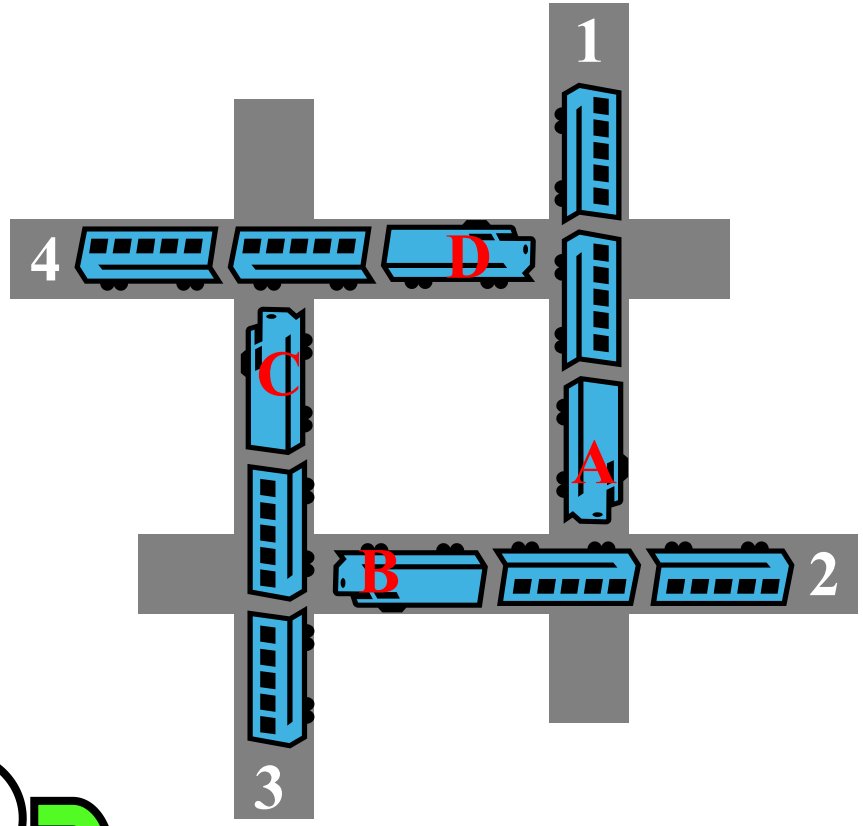
- 看一个实际的例子
- 竞争使用资源: 道路
- A占有道路1，又要请求道路2，B占有…
- 形成了无限等待

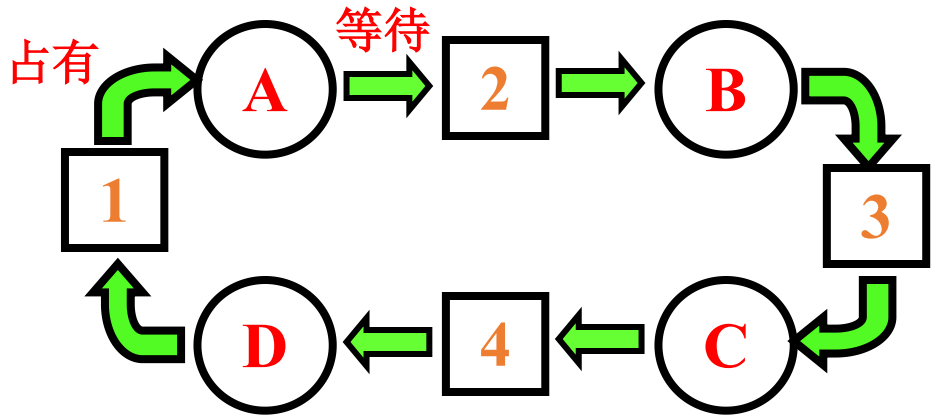




死锁特征

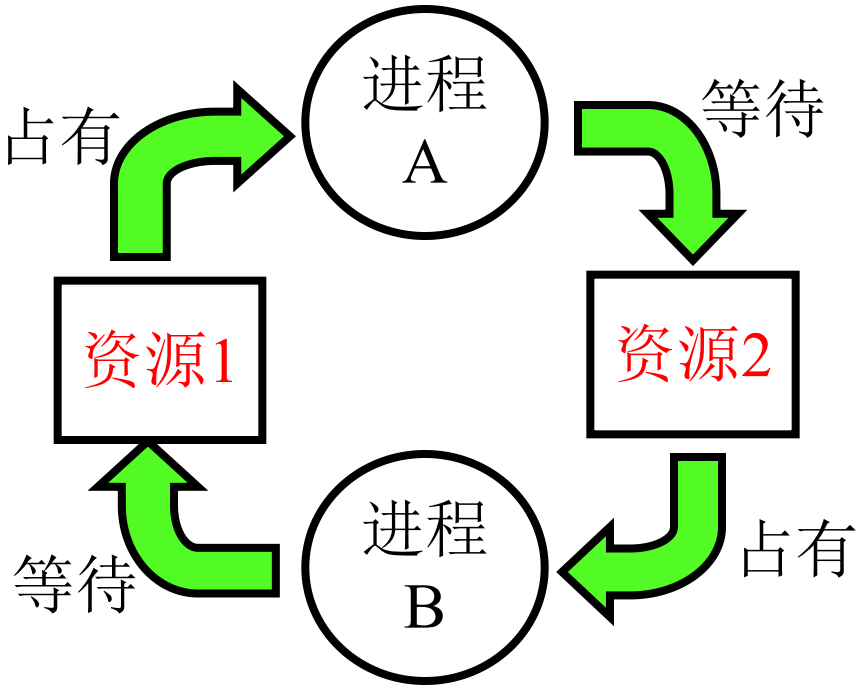
- 每条道路（资源）只能由一辆火车（进程）占有。
- 某条道路由一火车占用后其他火车不能抢占前者的道路。
- 火车持续持有所占有的道路，等待其他道路空闲。
- 火车之间相互等待。





死锁概念(Deadlock)

- 死锁: 多个进程（线程）因循环等待资源而造成无法执行的现象。
- 死锁会造成进程（线程）无法执行
- 死锁会造成系统资源的极大浪费(资源无法释放)



本章的主要内容

- 死锁（Deadlock）的引入
- 死锁特征分析—产生死锁的四个必要条件
- 死锁处理方法

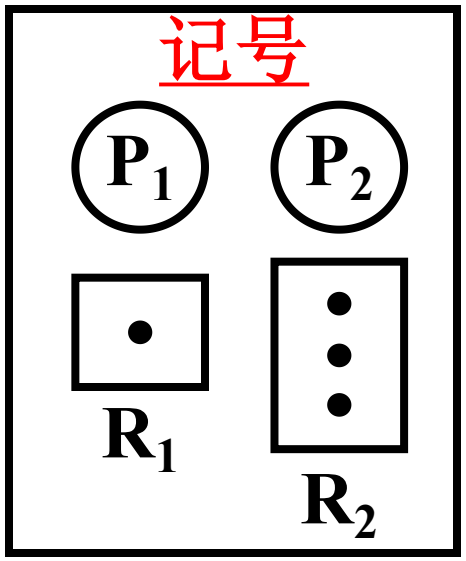
资源的分析

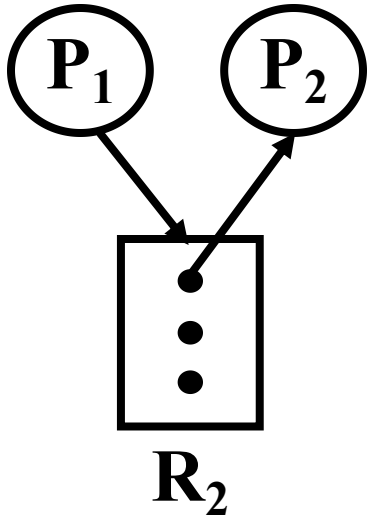
- 多个进程因等待资源才造成死锁
- 资源: 进程在完成其任务过程所需要的所有对象
  - CPU、内存、磁盘块、外设、文件、信号量 …
- 显然有些资源不会造成死锁，而有些会
  - 只读文件是不会造成进程等待的，也就不会死锁
  - 打印机一次只能让一个进程使用，可能会造成死锁
  - 显然，资源互斥访问是死锁的必要条件

称为互斥访问资源

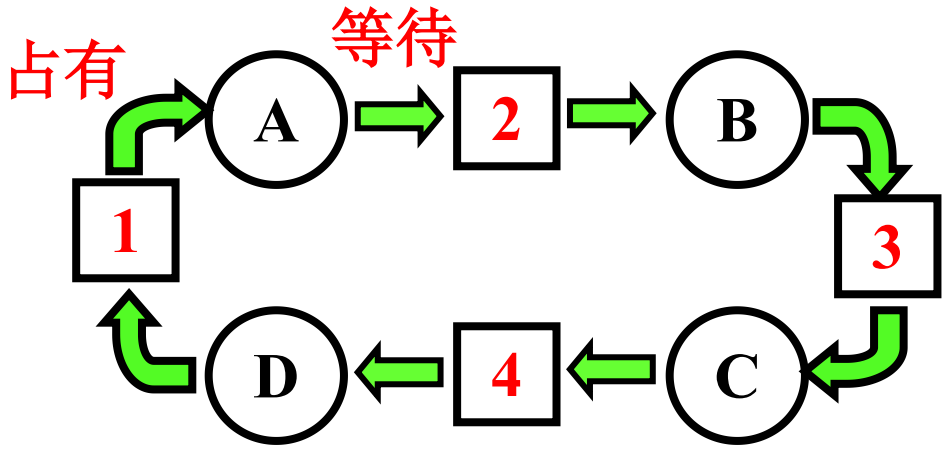
资源分配图

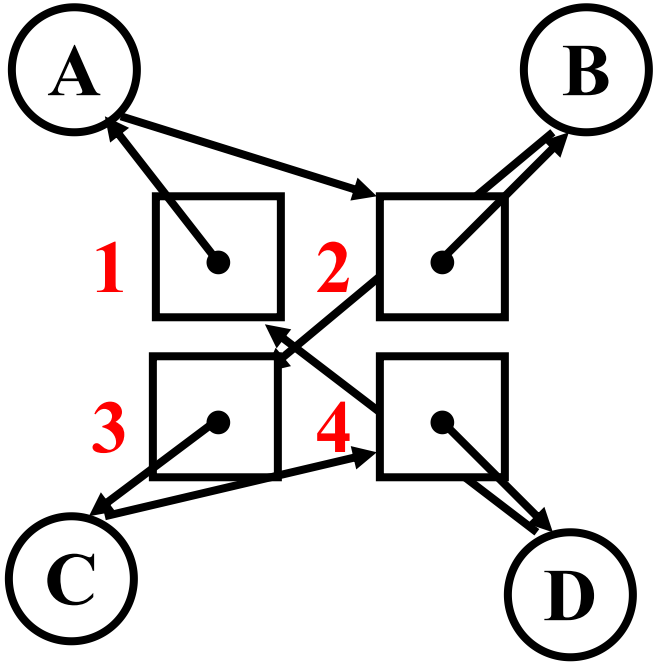
- 资源请求需要形成环路等待才死锁！如何描述这种等待关系？
- 资源分配图模型
  - 一个进程集合{$P_1, P_2, \ldots, P_n$}
  - 一资源类型集合{$R_1, R_2, \ldots, R_m$}
  - 资源类型R_i有W_i个实例
  - 资源请求边：有向边$P_i \rightarrow R_j$
  - 资源占有边：有向边$R_i \rightarrow P_k$





资源分配图实例



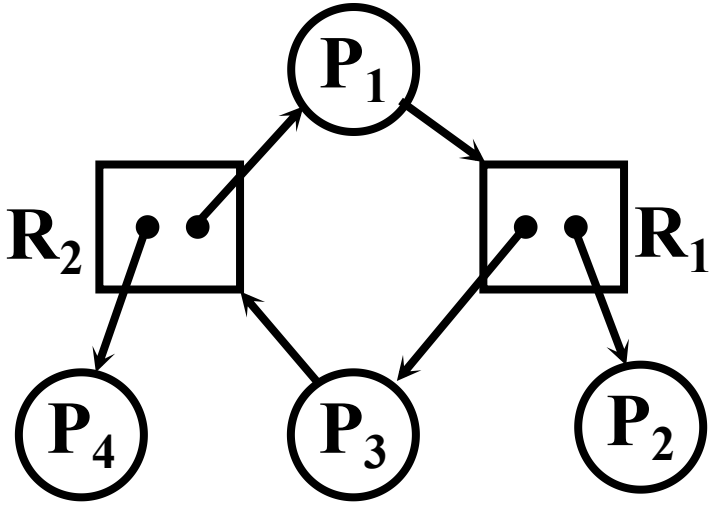


■ 存在环路:

1→A→2→B→3

→C→4→D→1

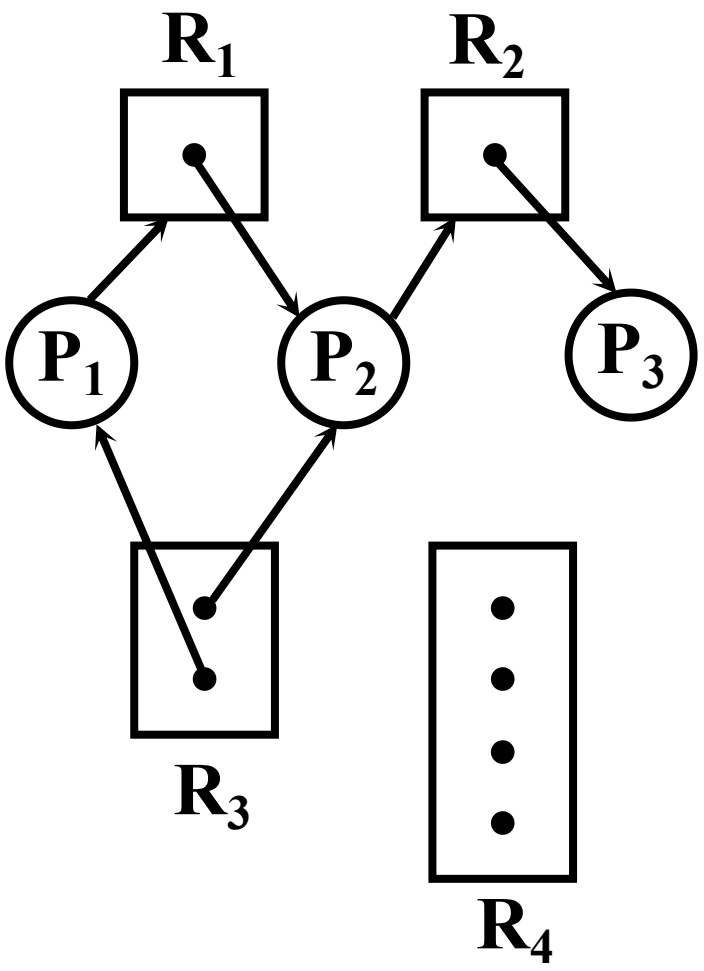
（产生死锁）



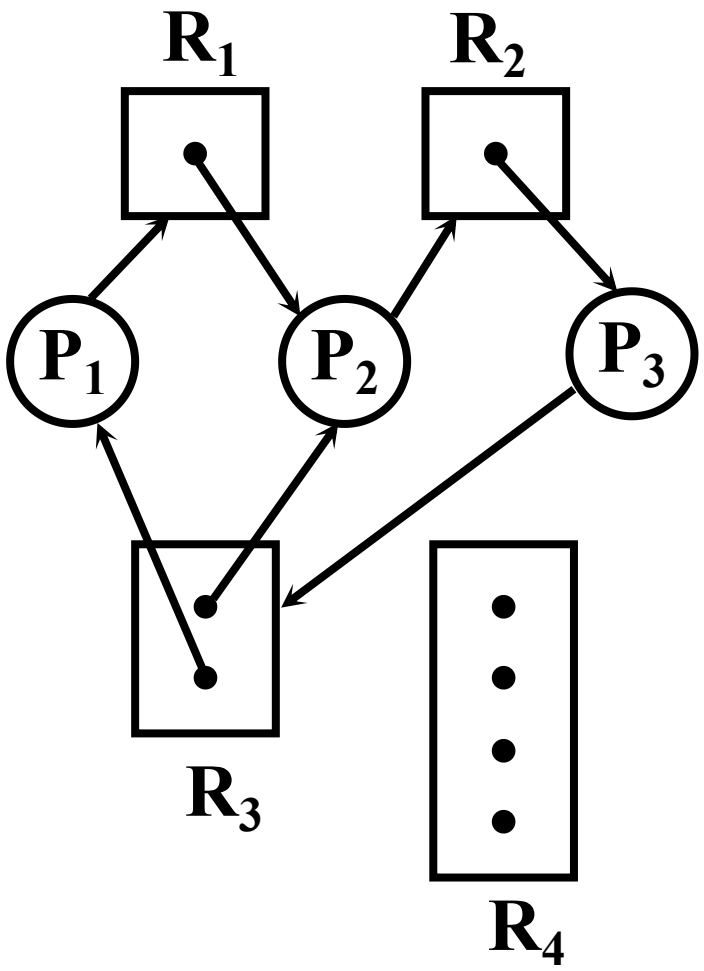
■ 存在环路: P1→R1→P3→R2→P1

（但并不死锁，仍可继续执行）

资源分配图实例



存在环路不一定死锁，
死锁一定存在环路

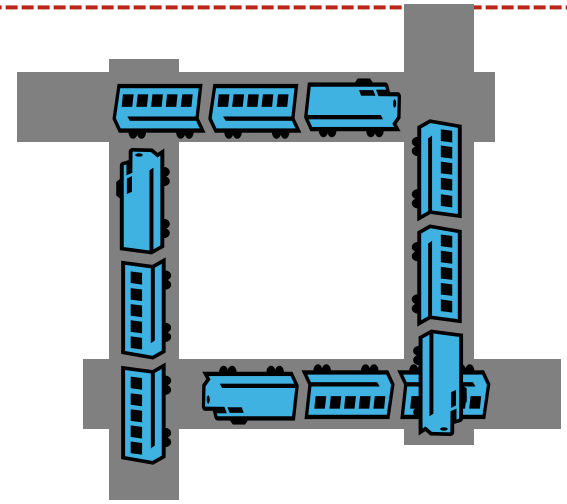


什么情况下存在环路就一定死锁？

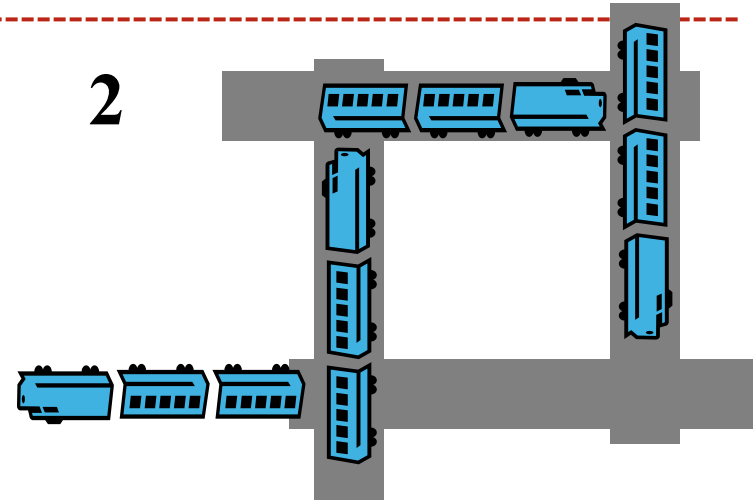
死锁的4个必要条件

- 1. 互斥使用(Mutual exclusion)
  - 至少有一个资源互斥使用

1

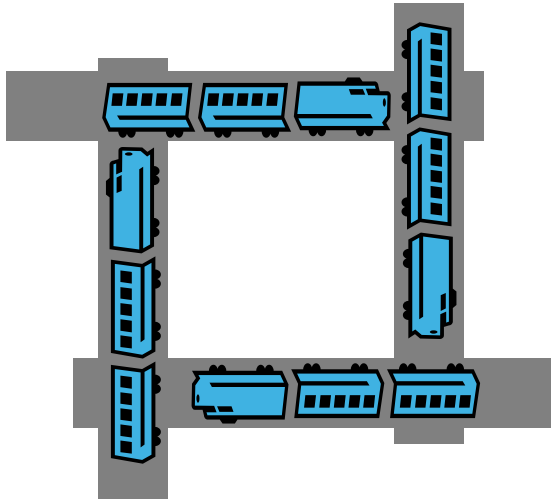


2

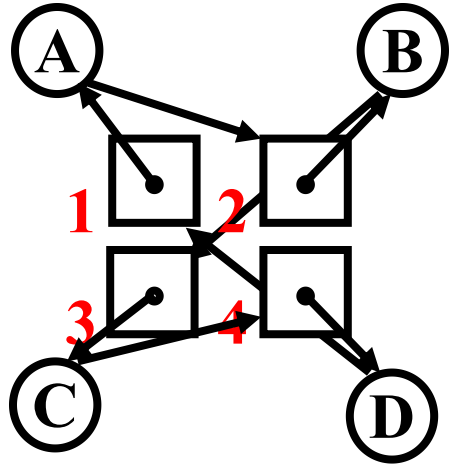


- 2. 不可抢占(No preemption)
  - 资源只能自愿放弃，如车开走以后
- 3.持有和等待(Hold and wait)
  - 进程占有资源时还申请资源

3



4



- 4. 循环等待(Circular wait)
  - 在资源分配图中存在一个环路

本章的主要内容

- 死锁（Deadlock）的引入
- 死锁特征分析—产生死锁的四个必要条件
- 死锁处理方法

死锁处理方法概述

- 1. 死锁预防：**规划出不安全的区域**
  - 破坏死锁的必要条件
- 2. 死锁避免：**摸石头过河**
  - 检测每个资源请求，如果造成死锁就拒绝
- 3. 死锁检测+恢复：**掉坑里，再爬出来**
  - 检测到死锁出现时，剥夺一些进程的资源
- 4. 死锁忽略：**躺平**
  - 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（即持有不用即可抢占）
- 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)

■ 破除持有和等待

- 必须保证在请求资源时，不持有任何其他资源
- 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: 要求编程时就需考虑，用户会觉得很别扭；可能释放某些资源(申请序号小的资源)，进程会无法执行。
- 在通用操作系统中使用不多，但在嵌入式操作系统中有使用。

■ 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

- 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

如何判断就成了问题的核心！

- 需要系统具有一些额外的先验信息。
  - 最简单和最有效的模式是要求每个进程声明它可能需要的每个类型资源的最大数目。
  - 资源的分配状态通过限定提供和分配的资源数量，不超过进程的最大需求。
  - 死锁避免算法动态检查资源的分配状态，以确保永远不会有一个环形等待状态。

死锁避免

- 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，…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$P_1, \ldots, P_n$
- 安全序列$P_1, \ldots, P_n$ 应该满足的性质：

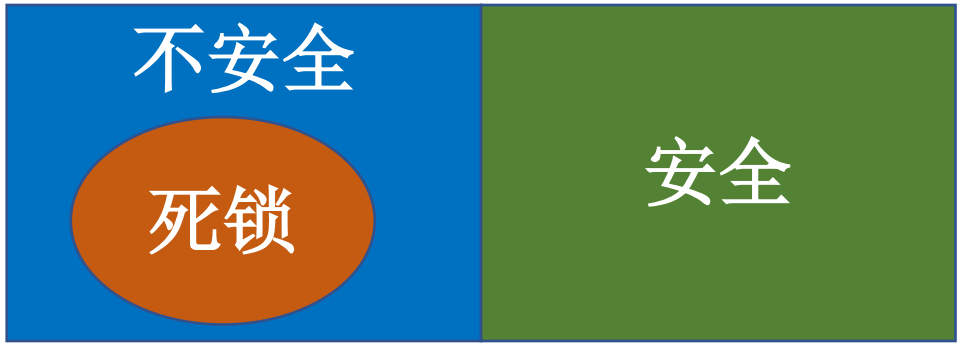
$P_i\ (1 \le i \le n)$ 需要资源 ≤ 剩余资源 + 分配给所有 $P_j\ (1 \le j \le i)$ 资源

- P_i要求资源能够由当前剩余资源 + 所有P_j持有的资源来满足。当P_{i-1}完成后，释放资源，P_i可以得到所需资源，执行。

不安全状态与死锁

- 如果系统处于安全状态 => 无死锁
- 如果系统处于不安全状态 => 可能死锁
- 避免死锁：确保系统永远不会进入不安全状态

如何检测不安全状态?



死锁避免之银行家算法

- 一个银行家：目前手里只有1亿
  - 第A个开发商：已贷款15亿，资金紧张还需3亿。
  - 第B个开发商：已贷款5亿，还需贷款1亿，运转良好能收回。
  - 第C个开发商：已贷款2亿，欲贷款18亿。
- 开发商B还钱，再借给A，则可以继续借给C
- 银行家当前可用的资金（Available）？可以利用的资金，即可用的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）

死锁避免之银行家算法

- 银行家算法（ Edsger Dijkstra）
  - 尝试寻找一个安全的执行时序，决定一个状态是否是安全的。
  - 不存在安全的序列，则状态是不安全的。

```
1.Banker();
2.int n,m; //系统中进程总数n和资源种类总数m
3.int Available[m]; //资源当前可用总量
4.int Allocation[n][m]; //当前给分配给每个进程的各种资源数量
5.int Need[n][m]; //当前每个进程还需分配的各种资源数量
6.int Work[m]; //当前可分配的资源，包括可收回的
7.bool Finish[n]; //进程是否结束
```

死锁避免之银行家算法

■ 安全状态判定（思路）：

- 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Pi（未完成但目前剩余资源可满足其需要，这样的进程是能够完成的）：

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

  如果没有这样的进程Pi，则跳转到第④步

- ③（若有则）Pi一定能完成，并归还其占用的资源，即：

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

- 当前状态:

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

安全序列是<P_1, P_3, P_4, P_0, P_2>

安全序列是唯一的吗？

死锁避免之银行家算法


```
1. bool Found;
2. Work = Available; Finish[1..n] = false;
3. while(true){
4.    Found = false; //是否为安全序列找到一个新进程
5.    for(i=1; i<=n; i++){
6.       if(Finish[i]==false && Need[i]<=Work){
7.          Work = Work + Allocation[i];
8.          Finish[i] = true;
9.          printf("%d->",i);//输出安全序列
10.         Found = true;
11.      }
12.   }  没有安全序列或已经找到
13.   if(Found==false) break;
14.}
15.for(i=1;i<=n;i++)
16.   if(Finish[i]==false)
17.      return "deadlock";
```

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

最好情形：安全状态就是$P_1 \rightarrow P_n$

最坏情形：$P_n \rightarrow P_1$

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，然后寻找安全序列，找到，分配成功，找不到，已分配资源收回。

```
1. extern Banker();
2. int Request[m]; /*进程Pi的资源申请*/
3. if(Request>Need[i])
4.    return “error”;
5. if(Request>Available)
6.    sleep();
7. Available = Available - Request;
8. Allocation[i] = Allocation[i] + Request;
9. Need[i] = Need[i] - Request; /*先将资源分配给Pi*/
10.if(Banker()==“deadlock”) /*调用银行家算法判定是否会死锁*/
11.    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)

■ P1申请资源(1,0,2)

	Allocation	Need	Available
	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	

	Allocation	Need	Available
	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	

序列<P_1, P_3, P_2, P_4, P_0>是安全的

此次申请允许

死锁避免之资源请求实例(2)

P0再申请(0,2,0)

	Allocation	Need	Available
	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	

	Allocation	Need	Available
	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	

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

```
1. bool Found;                          //对银行家算法进行改进
2. int Request[n][m];
3. Work = Available; Finish[1..n] = false;
4. if Allocation[i] != 0: Finish[i] = false;
5. else: Finish[i] = true;
6. while(true){
7.    Found = false; //是否为安全序列找到一个新进程
8.    for(i=1; i<=n; i++){
9.      if(Finish[i]==false && Request[i]<=Work){
10.          Work = Work + Allocation[i];
11.          Finish[i] = true;
12.          Found = true;
13.     }
14.   }
15.   if(Found==false)  break;
16.}
```

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

```
17.for(i=1;i<=n;i++) {
18.   if(Finish[i]==false) {
19.     deadlock = deadlock + {i};
20.     return “deadlock”;
21.   }
22.}
```

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？ …

鸵鸟算法（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(mn2)，效率太低
- 死锁检测+恢复?
  - 还要执行银行家算法O(mn2)，且恢复并不容易
- 鸵鸟算法: 对死锁不做任何处理……
  - 死锁出现时，手动干预——重新启动
  - 死锁出现不是确定的，避免死锁付出的代价毫无意义
  - 有趣的是大多数操作系统都用它，如UNIX和Windows

死锁总结

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

任何思想、概念、技术的主流都会随着时间而改变，操作系统尤为明显！

习题

■ 1.死锁的避免是根据（）采取措施实现的。

A.配置足够的系统资源　　B.使进程的推进顺序合理

C.破坏死锁的四个必要条件之一　　D.防止系统进入不安全状态

■ 答案：D

■ 解析：死锁避免是指在资源动态分配过程中用某些算法加以限制，防止系统进入不安全状态从而避免死锁的发生。选项B是避免死锁发生后的结果，而不是原理。

习题

- 2.死锁预防是保证系统不进入死锁状态的静态策略，其解决方法是破坏产生死锁的四个必要条件之一。下列方法中破坏了“循环等待”条件的是：

A.银行家算法　　B.一次性分配策略

C.剥夺资源法　　D.资源有序分配

- 答案：D

- 解析：资源有序分配策略可以限制循环等待条件的发生。选项A判断是否为不安全状态；选项B破坏了持有和等待条件；选项C破坏了不可抢占条件。

习题

- 3.某系统中有三个并发进程都需要四个同类资源，则该系统必然不会发生死锁的最少资源是（）

  A.9　　B.10　　C.11　　D.12

- 答案：B

- 解析：资源数为9时，存在每个进程占有三个资源，为死锁。资源数为10是，必然存在一个进程能拿到4个资源，可以顺利执行完其他进程。

习题

■ 4.某系统有R1、R2和R3共三种资源,在T0时刻P1、P2、P3和P4这四个进程对资源的占用和需求情况见下表,此时系统的可用资源向量为(2,1,2), 试问:

1) 用向量或矩阵表示系统中各种资源的总数和此刻各进程对各资源的需求数目。

进程	最大资源需求数			已分配资源数量		
	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0
P2	6	1	3	4	1	1
P3	3	1	4	2	1	1
P4	4	2	2	0	0	2

习题

- 2) 若此时进程P1和进程P2均发出资源请求向量Request(1,0,1)，为了保证系统的安全性，应如何分配资源给这两个进程?说明所采用策略的原因。
- 3）若2）中两个请求立即得到满足后，系统此刻是否处于死锁状态？

习题

解答

1）系统中资源总量为某时刻系统中可用资源量与各进程已分配资源量之和，即：

$$(2,1,2)+(1,0,0)+(4,1,1)+(2,1,1)+(0,0,2)=(9,3,6)$$

各进程对资源的需求量为各进程对资源的最大需求量与进程已分配资源量之差,即

$$\begin{bmatrix}3&2&2\\6&1&3\\3&1&4\\4&2&2\end{bmatrix}-\begin{bmatrix}1&0&0\\4&1&1\\2&1&1\\0&0&2\end{bmatrix}=\begin{bmatrix}2&2&2\\2&0&2\\1&0&3\\4&2&0\end{bmatrix}$$

习题

2）若此时P_1发出资源请求 $Request_1(1,0,1)$，则按银行家算法进行检查

$Request_1(1,0,1) \le Need_1(2, 2, 2)$

$Request_1(1,0,1) \le Available(2, 1, 2)$

试分配并修改相应数据结构，由此形成的进程P_1请求资源后的资源分配情况见右表。

进程	Allocation			Need			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	0	1	1	2	1	1	1	1
P2	4	1	1	2	0	2			
P3	2	1	1	1	0	3			
P4	0	0	2	4	2	0			

2）再利用安全性算法检查系统是否安全，可用资源 Available(1,1,1)已不能满足任何进程，系统进入不安全状态，此时系统不能将资源分配给进程P_1。

习题

2）若此时P_2发出资源请求 $Request_2(1,0,1)$，则按银行家算法进行检查

$Request_2(1,0,1) \le Need_2(2, 2, 2)$

$Request_2(1,0,1) \le Available(2, 1, 2)$

试分配并修改相应数据结构，由此形成的进程P_2请求资源后的资源分配情况见右表。

进程	Allocation			Need			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	1	0	0	2	2	2	1	1	1
P2	5	1	2	1	0	1			
P3	2	1	1	1	0	3			
P4	0	0	2	4	2	0			

习题

2）再利用安全性算法检查系统是否安全, 可得到如下表中所示的安全性检测情况。注意中各个进程对应的 Available向量表示在该进程释放资源之后更新的 Available向量。

从上表可以看出，此时存在一个安全序列{P2,P1,P3,P4}，故该状态是安全的，可以立即将P2申请的资源分配给它。

进程	work			Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P2	1	1	1	1	0	1	5	1	2	6	2	3
P1	6	2	3	2	2	2	1	0	0	7	2	3
P3	7	2	3	1	0	3	2	1	1	9	3	4
P4	9	3	4	4	2	0	0	0	2	9	3	6

3）若2）中的两个请求立即得到满足，则此刻系统并未立即进入死锁状态，因为这时所有的进程未提出新的资源申请，全部进程均未因资源申请没有得到满足而进入阻塞态。只有当进程提出资源申请且全部进程都进入阻塞态时，系统才处于死锁状态。

Hope you enjoyed the OS course!