操作系统A

Principles of Operating System

北京大学计算机科学技术系 陈向群

Department of computer science and Technology, Peking University

2020 Autumn

本章要求掌握的概念

死锁	活锁	饥饿
死锁预防	死锁避免	死锁检测与解除
资源有序分配法	银行家算法	安全状态
资源分配图	哲学家就餐问题	……

大纲

- 基本概念
- 资源分配图
- 死锁解决方案
  - 死锁预防
  - 死锁避免
  - 死锁检测与解除
- 哲学家就餐问题

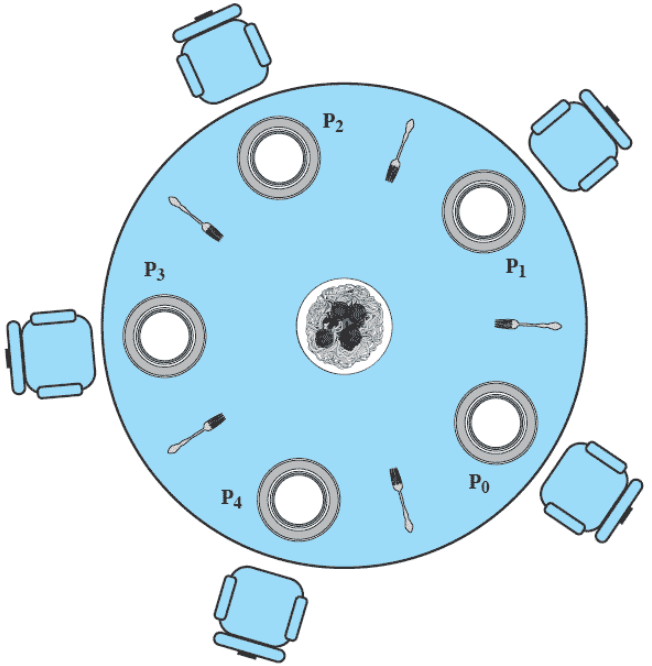
死锁　活锁　饥饿……

死锁的基本概念

经典的哲学家就餐问题

问题描述：

- 有五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只筷子
- 每个哲学家的行为是思考，感到饥饿，然后吃通心粉
- 为了吃通心粉，每个哲学家必须拿到两只筷子，并且每个人只能直接从自己的左边或右边去取筷子(筷子的互斥使用、不能出现死锁和饥饿)

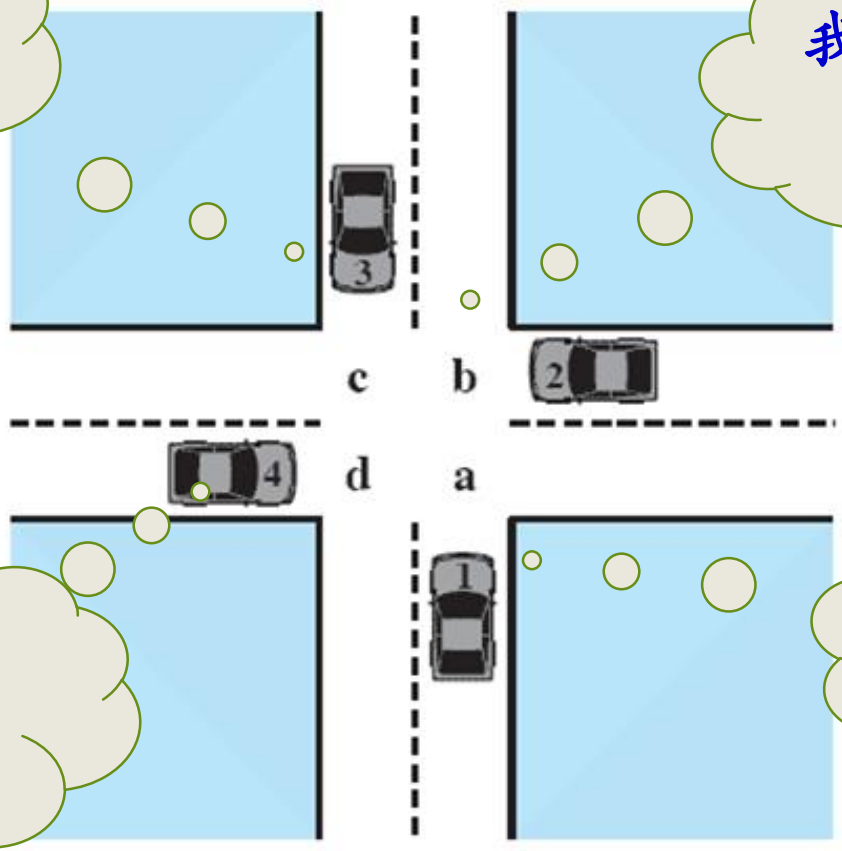


问题模型：
应用程序中并发线程执行时，协调处理共享资源

哲学家就餐问题第一种解决方案

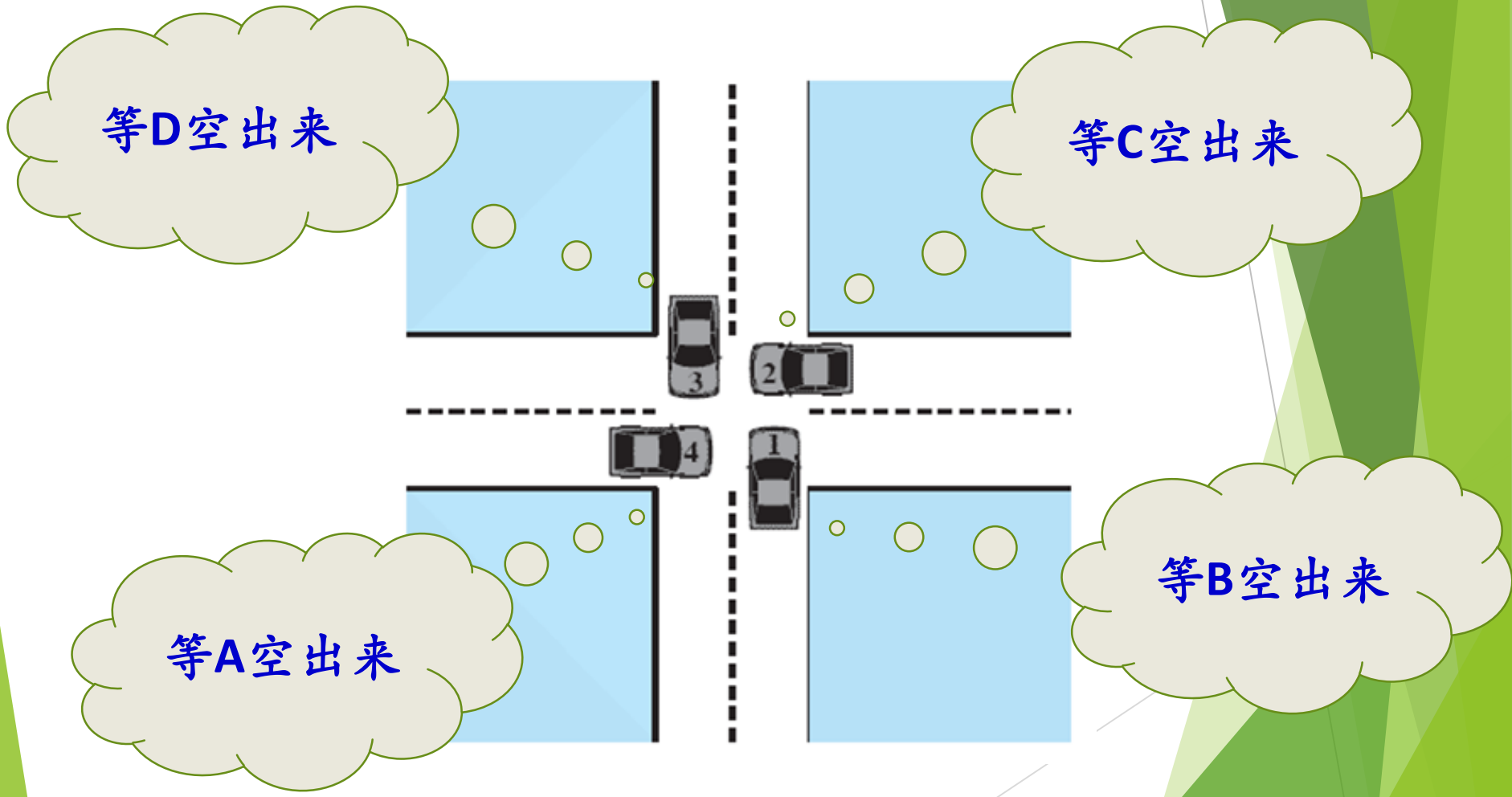
```
semaphore fork [5] = {1};   /* 筷子 */
int i;
void philosopher (int i)
{
      while (true) {
      think();
      P (fork[i]);
      P (fork [(i+1) mod 5]);
      eat();
      V (fork [(i+1) mod 5]);
      V (fork[i]);
}
}
void main()
{
     parbegin (philosopher (0), philosopher (1),philosopher (2),
philosopher (3),philosopher (4));   /* 五个哲学家并发执行 */
}
```

日常生活中的死锁现象（1）



William Stallings

日常生活中的死锁现象（2）



William Stallings

死锁的基本概念

死锁的定义

一组进程中，每个进程都无限等待被该组进程中另一进程所占有的资源，因而永远无法得到的资源，这种现象称为进程死锁，这一组进程就称为死锁进程

如果死锁发生，会浪费大量系统资源，甚至导致系统崩溃

- 参与死锁的所有进程都在等待资源
- 参与死锁的进程是当前系统中所有进程的子集

为什么会出现死锁？

资源数量有限、锁和信号量错误使用

资源的使用方式：

“申请--分配--使用--释放”模式

可重用资源：可以被多个进程多次使用

- 可抢占资源与不可抢占资源
- 处理器、I/O部件、内存、文件、数据库、信号量

可消耗资源：

只可使用一次的可以创建和销毁的资源

- 信号、中断、消息

活锁和饥饿的区别

活锁

- 加锁
- 轮询
- 没有进展也没有阻塞

饥饿

资源分配策略决定

Peterson算法

```
void process_A(void) {
        enter_region(&resource_1);
        enter_region(&resource_2);
        use_both_resources( );
        leave_region(&resource_2);
        leave_region(&resource_1);
}

 void process_B(void) {
        enter_region(&resource_2);
        enter_region(&resource_1);
        use_both_resources( );
        leave_region(&resource_1);
        leave_region(&resource_2);
 }
```

产生死锁的必要条件

- 互斥使用(资源独占)

  一个资源每次只能给一个进程使用

- 占有且等待(请求和保持，部分分配)

  一个进程在申请新的资源的同时保持对原有资源的占有

- 不可抢占(不可剥夺)

  资源申请者不能强行的从资源占有者手中夺取资源，资源只能由占有者自愿释放

- 循环等待

  存在一个进程等待队列 $\{P_1, P_2, \ldots, Pn\}$，其中P_1等待P_2占有的资源，P_2等待P_3占有的资源，…，Pn等待P_1占有的资源，形成一个进程等待环路

Resource Allocation Graph……

资源分配图（RAG）

资源分配图(RAG)

用有向图描述系统资源和进程的状态

二元组G=（V，E）

V：结点集合，分为P(进程)，R(资源)两部分

$P = \{P_1, P_2, \dots, P_n\}$

$R = \{R_1, R_2, \dots, R_m\}$

E：有向边的集合，其元素为有序二元组

(P_i, R_j) 或 (R_j, P_i)

资源分配图

系统由若干类资源构成，一类资源称为一个资源类；
每个资源类中包含若干个同种资源，称为资源实例

资源类：用方框表示

资源实例：用方框中的黑圆点表示

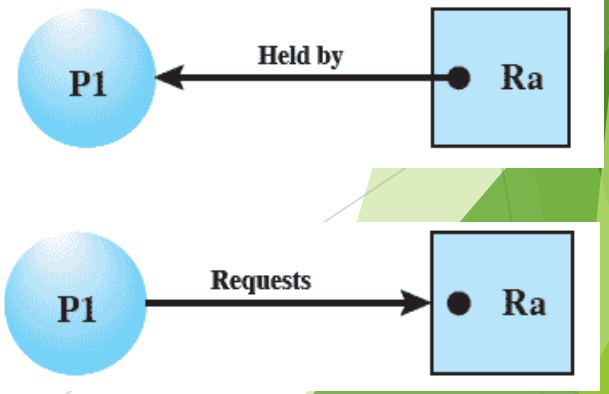
进程：用圆圈中加进程名表示

分配边：

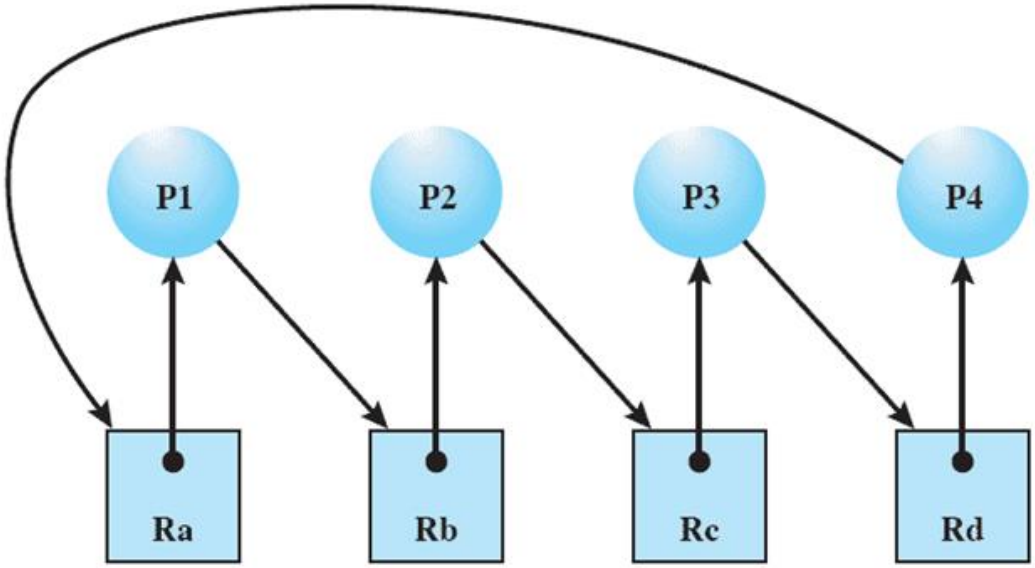
资源实例→进程的一条有向边

申请边：

进程→资源类的一条有向边



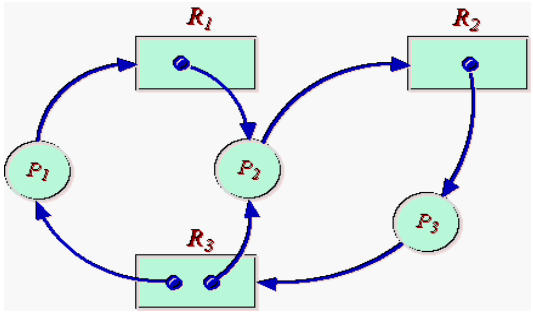
例子：十字路口



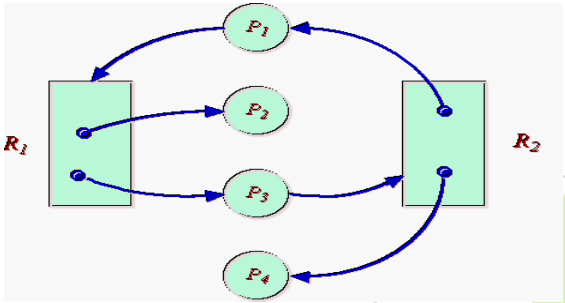
死锁定理

死锁定理

- 如果资源分配图中没有环路，则系统中没有死锁，如果图中存在环路则系统中可能存在死锁
- 如果每个资源类中只包含一个资源实例，则环路是死锁存在的充分必要条件



有环有死锁



有环无死锁

资源分配图化简

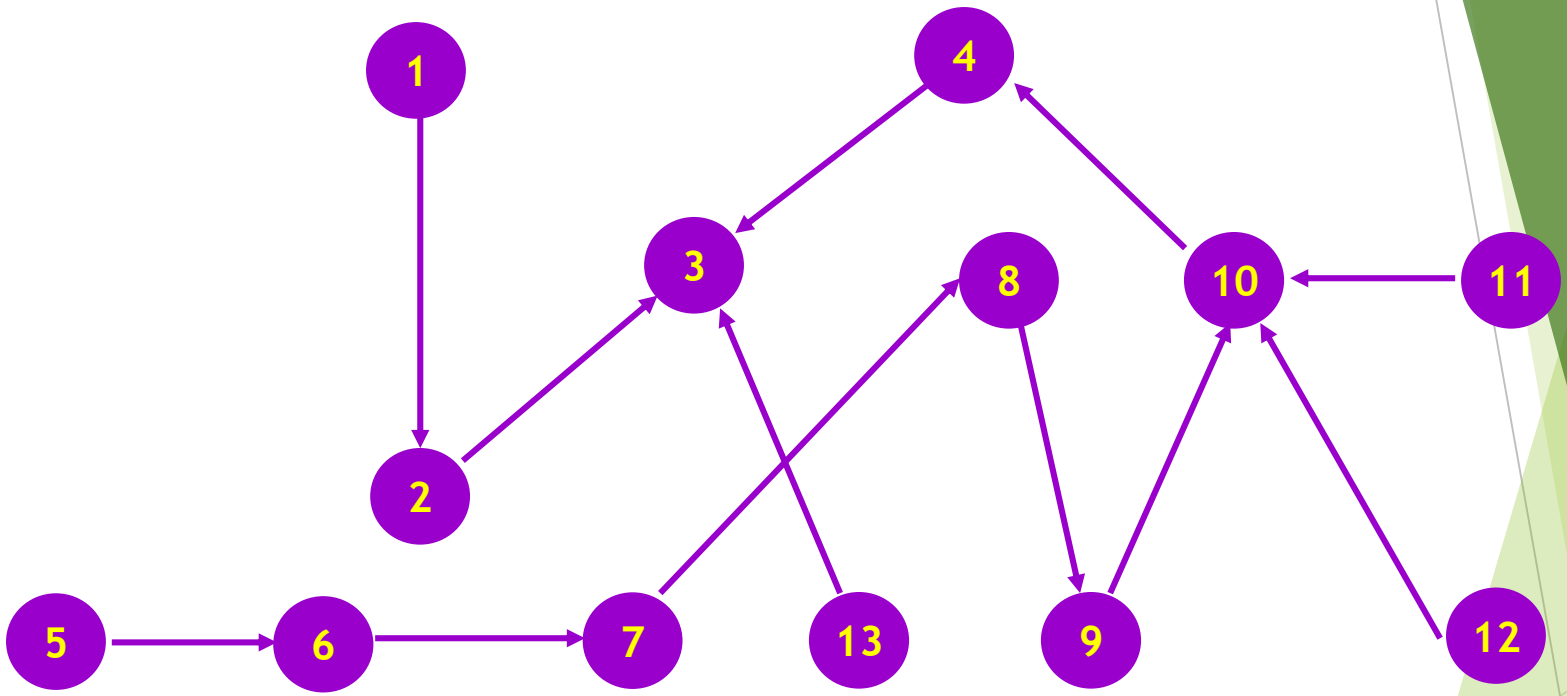
化简步骤：

1）找一个非孤立点进程结点且只有分配边，去掉分配边，将其变为孤立结点

2）再把相应的资源分配给一个等待该资源的进程，即将某进程的申请边变为分配边

重复1）、2）

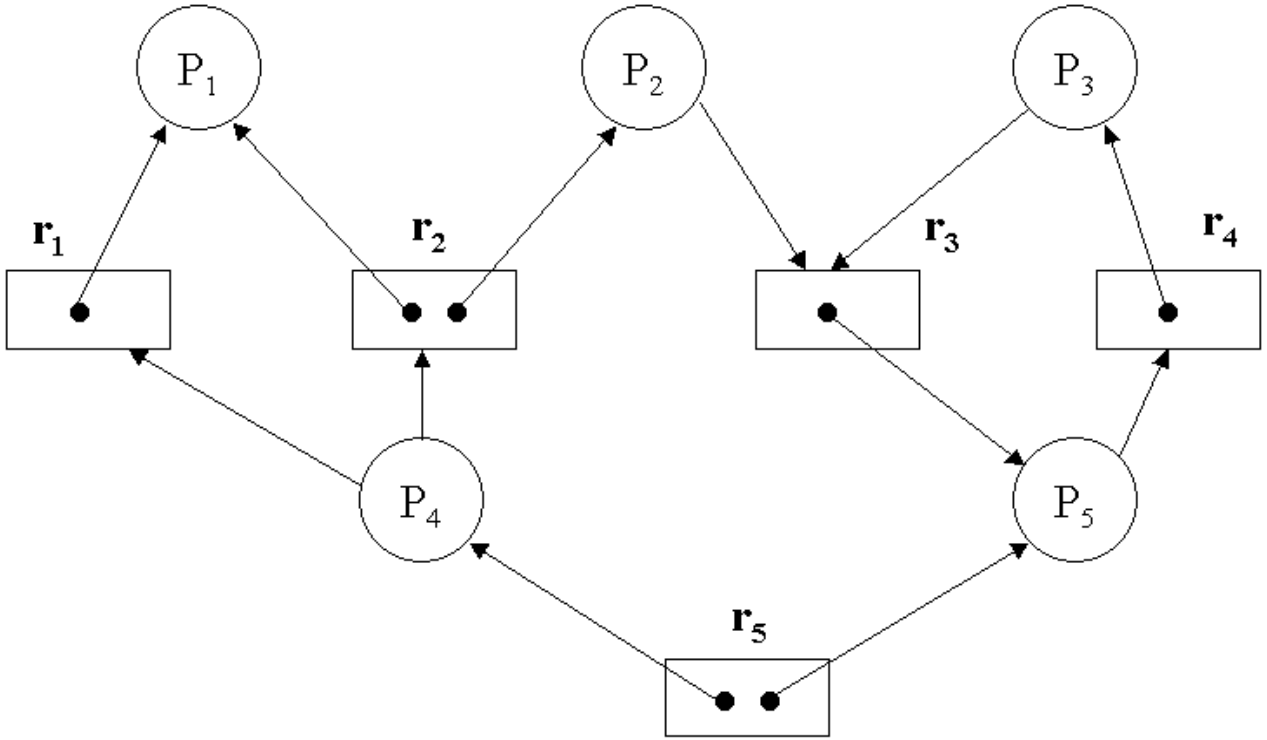
资源分配图化简的例子



这个图可以被完全化简，所以没有死锁

当然，以后情况可能发生变化

课堂练习



死锁预防 死锁避免 死锁检测与解除.....

解决死锁

解决死锁的方法

- ✓ 不考虑此问题（鸵鸟算法）
- ✓ 不让死锁发生
  - 死锁预防

    静态策略：设计合适的资源分配算法，不让死锁发生
  - 死锁避免

    动态策略：以不让死锁发生为目标，跟踪并评估资源分配过程，根据评估结果决策是否分配
- ✓ 让死锁发生
  - 死锁检测与解除

1. 死锁预防

定义：

在设计系统时，通过确定资源分配算法，排除发生死锁的可能性

具体的做法是防止产生死锁的四个条件中任何一个发生

死锁预防（1/4）

破坏“互斥使用/资源独占”条件

资源转换技术：把独占资源变为共享资源

SPOOLing技术的引入

解决不允许任何进程直接占有打印机的问题

设计一个“精灵deamon”进程/线程负责管理打印机，进程需要打印时，将请求发给该daemon，由它完成打印任务

死锁预防（2/4）

破坏“占有且等待”条件

实现方案1：要求每个进程在运行前必须一次性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

问题：资源利用率低；“饥饿”现象

实现方案2：在允许进程动态申请资源前提下规定，一个进程在申请新的资源不能立即得到满足而变为等待状态之前，必须释放已占有的全部资源，若需要再重新申请

死锁预防（3/4）

破坏“不可抢占”条件

实现方案：虚拟化资源

当一个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(两个进程优先级不同)

局限性：适用于状态易于保存和恢复的资源

CPU、内存

死锁预防（4/4）

破坏“循环等待”条件

通过定义资源类型的线性顺序实现

实施方案：资源有序分配法

把系统中所有资源编号，进程在申请资源时必须严格按资源编号的递增次序进行，否则操作系统不予分配

实现时要考虑什么问题呢?

例子：解决哲学家就餐问题

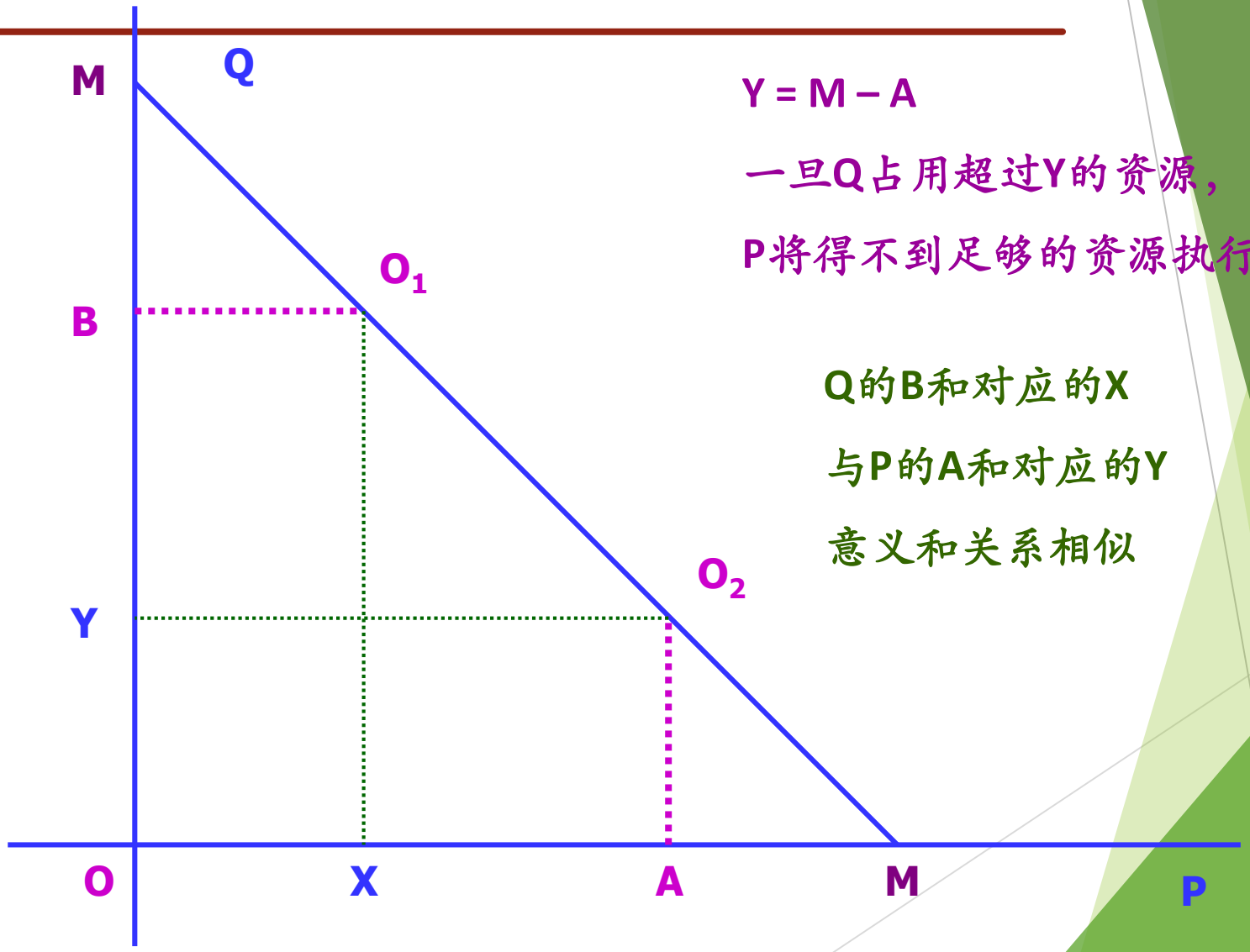
采用资源有序分配法会产生死锁吗？

有资源1，2，3，…，10

P_1：	P_2：	P_3： …… P_n：
申请1	申请1	
申请3	申请2	
申请9	申请5	
…	…	

2. 死锁避免

A为P总资源需求量



Y = M – A

一旦Q占用超过Y的资源，

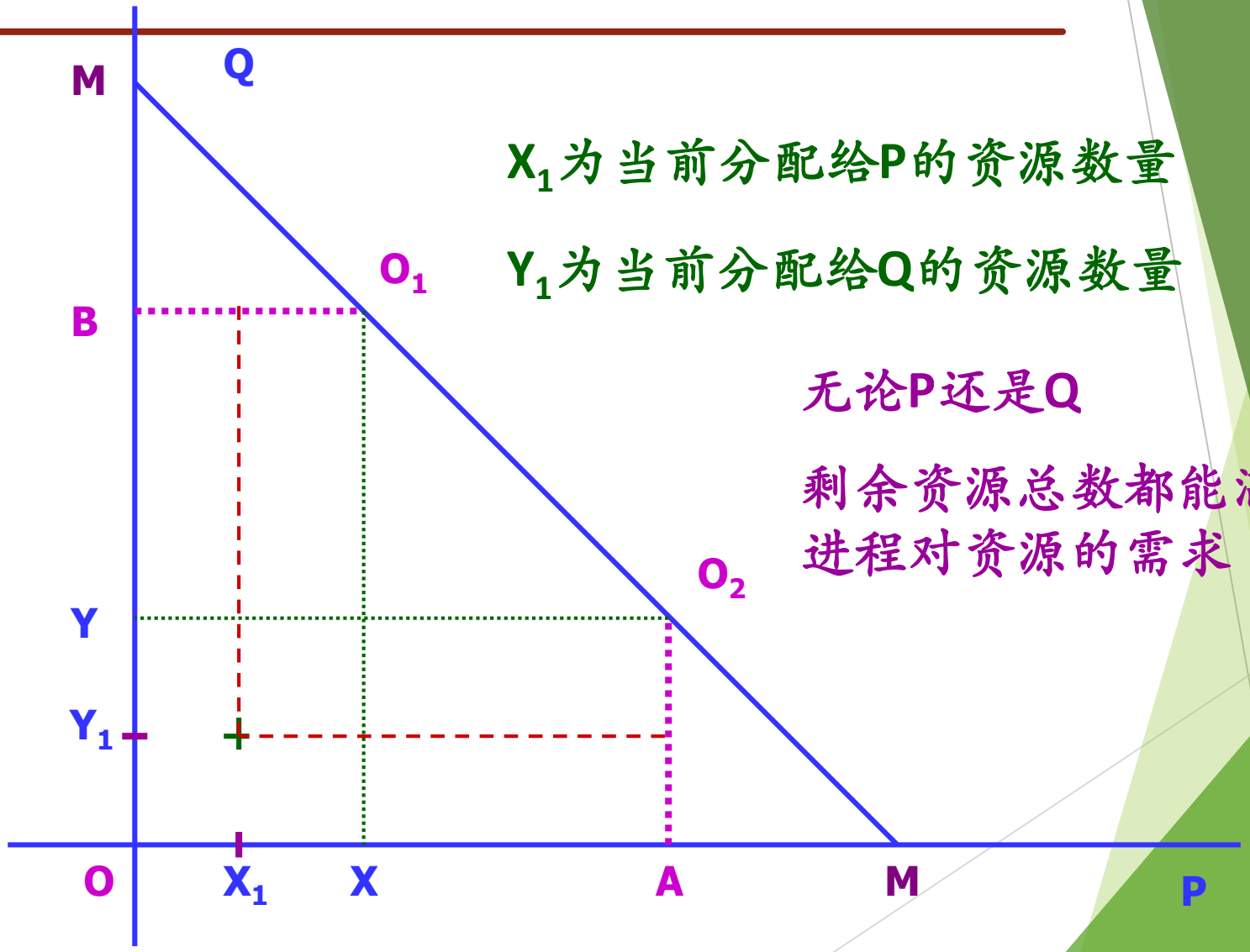
P将得不到足够的资源执行

Q的B和对应的X

与P的A和对应的Y

意义和关系相似

死锁避免



X_1为当前分配给P的资源数量

Y_1为当前分配给Q的资源数量

无论P还是Q

剩余资源总数都能满足进程对资源的需求

死锁避免

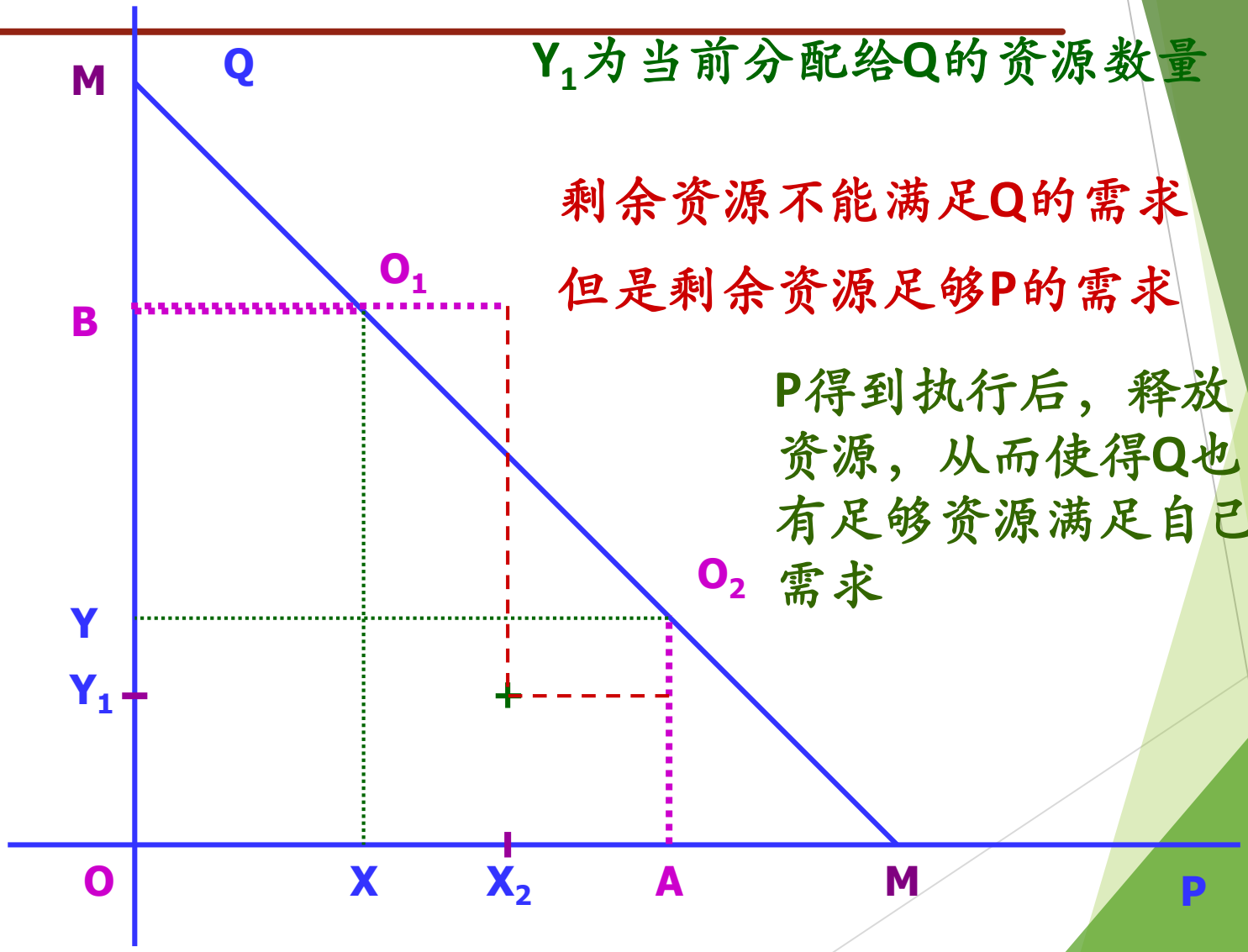
X_2为当前分配给P的资源数量

Y_1为当前分配给Q的资源数量

剩余资源不能满足Q的需求

但是剩余资源足够P的需求

P得到执行后，释放资源，从而使得Q也有足够资源满足自己需求



死锁避免

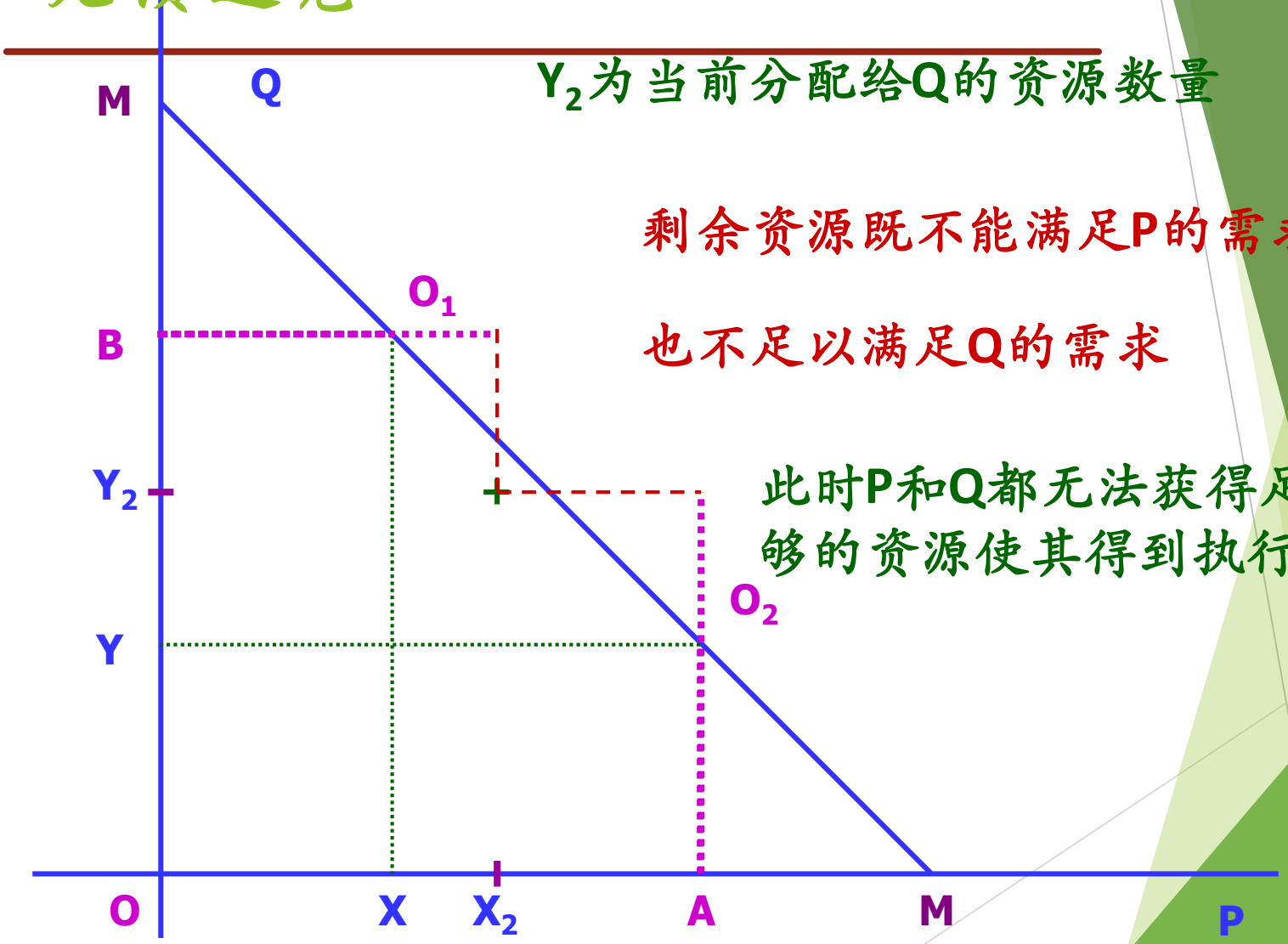
X_2为当前分配给P的资源数量

Y_2为当前分配给Q的资源数量

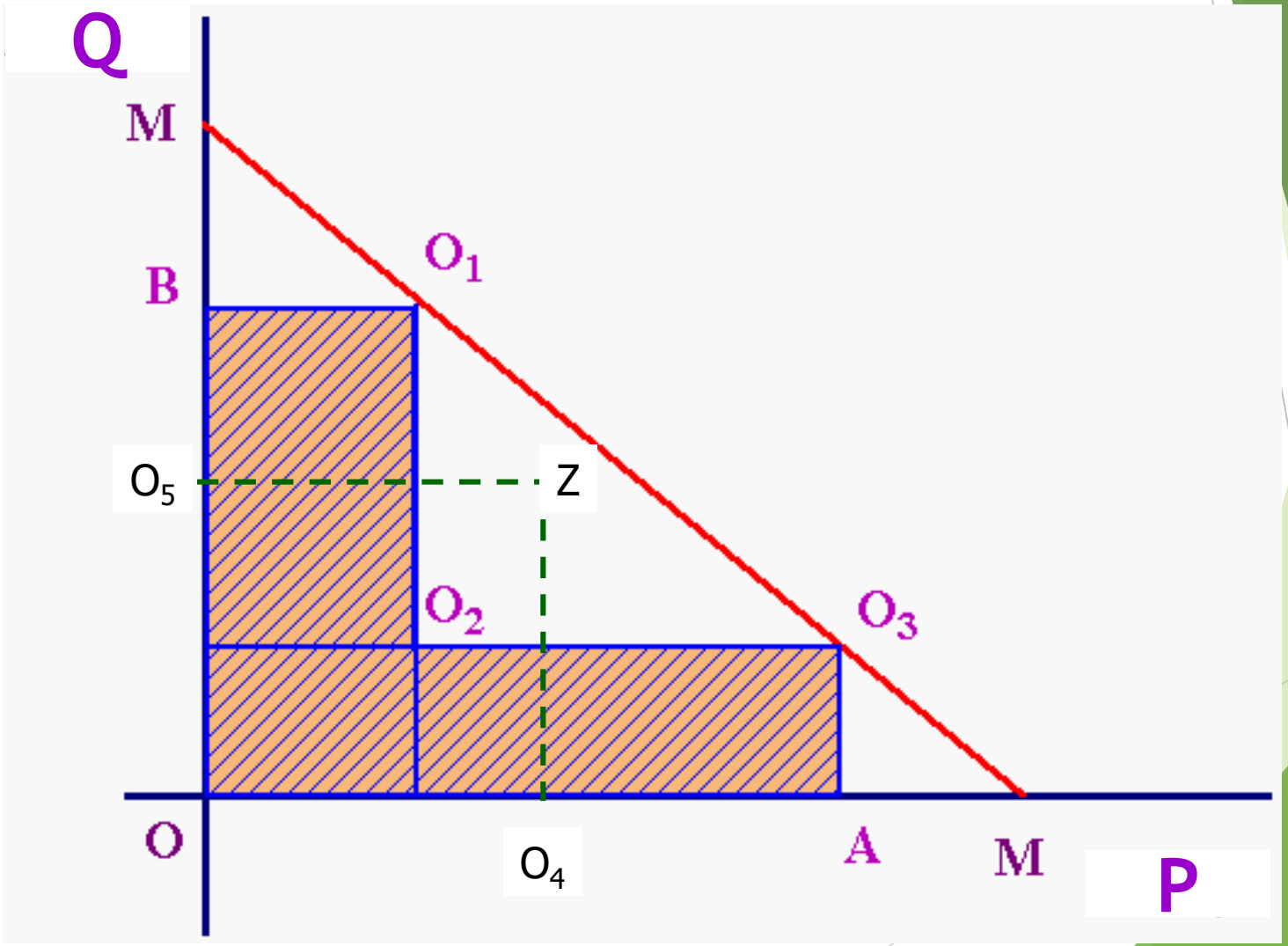
剩余资源既不能满足P的需求

也不足以满足Q的需求

此时P和Q都无法获得足够的资源使其得到执行



小结



死锁避免定义

定义：

在系统运行过程中，对进程发出的每一个系统能够满足的资源申请进行动态检查，并根据检查结果决定是否分配资源，若分配后系统发生死锁或可能发生死锁，则不予分配，否则予以分配

安全状态：

如果存在一个由系统中所有进程构成的安全序列P_1，…，P_n，则系统处于安全状态

安全序列

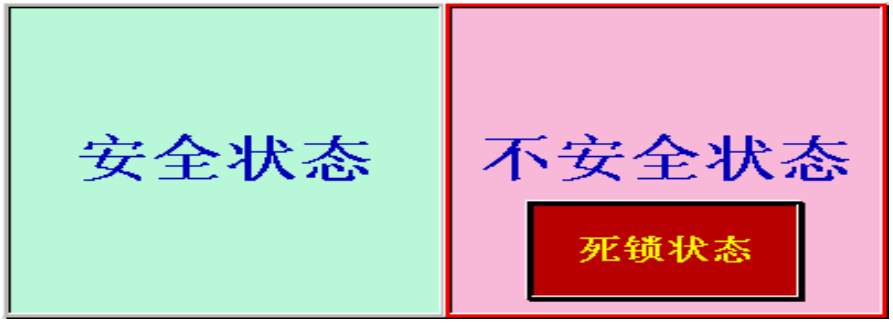
一个进程序列{P_1，…，P_n}是安全的，如果对于每一个进程P_i($1 \leqslant i \leqslant n$），它以后尚需要的资源量不超过系统当前剩余资源量与所有进程P_j ($j < i$)当前占有资源量之和，系统处于安全状态

安全状态一定没有死锁发生

安全状态与不安全状态

不安全状态：不存在一个安全序列

不安全状态一定导致死锁



银行家算法（Banker’s Algorithm）

Dijkstra提出(1965)

仿照银行发放贷款时采取的控制方式而设计的一种死锁避免算法

基本思想：？

银行家算法（Banker's Algorithm）

系统具有的特征：

1、在固定数量的进程中共享数量固定的资源

2、每个进程预先指定完成工作所需的最大资源数量

3、进程不能申请比系统中可用资源总数还多的资源

4、进程等待资源的时间是有限的

5、如果系统满足了进程对资源的最大需求，那么进程应该在有限的时间内使用资源，然后归还给系统

银行家算法

n：系统中进程的总数

m：资源类总数

Available: ARRAY[1..m] of integer;

Max: ARRAY[1..n,1..m] of integer;

Allocation: ARRAY[1..n,1..m] of integer;

Need: ARRAY[1..n,1..m] of integer;

Request: ARRAY[1..n,1..m] of integer;

简记符号：

Available

Max[i]

Allocation[i]

Need[i]

Request[i]

银行家算法

当进程pi提出资源申请时，系统执行下列步骤：

（1）若Request[i]≤Need[i]，转（2）；

否则错误返回；

（2）若Request[i]≤Available,

转（3）；否则进程等待；

（3）假设系统分配了资源，则有：

Available = Available - Request[i];

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

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

若系统新状态是安全的，则分配完成

若系统新状态是不安全的，则恢复原来状态，进程等待

银行家算法

为进行安全性检查，定义数据结构：

Work: ARRAY[1..m] of integer;

Finish: ARRAY[1..n] of Boolean;

安全性检查的步骤：

(1) Work = Available;

Finish = false;

(2) 寻找满足条件的i：

a.Finish[i]==false;

b.Need[i]≤Work;

如果不存在，则转(4)

(3) Work = Work + Allocation[i];

Finish[i] = true;

转(2)

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

银行家算法应用1

	目前占有量	最大需求量	尚需要量
P1	1	4	3
P2	4	6	2
P3	5	8	3
系统剩余量		2	

银行家算法应用2

	已分配的资源			最大需求量		
	A	B	C	A	B	C
P_1	0	1	0	7	5	3
P_2	2	0	0	3	2	2
P_3	3	0	2	9	0	2
P_4	2	1	1	2	2	2
P_5	0	0	2	4	3	3

剩余资源 A B C

3 3 2

问题:此状态是否为安全状态，如果是，则找出安全序列

在此基础上

P_2 申请（1，0，2）能否分配？为什么？

P_5 申请（3，3，0）能否分配？为什么？

P_1 申请（0，2，0）能否分配？为什么？

3. 死锁的检测与解除

死锁检测：

允许死锁发生，操作系统不断监视系统进展情况，判断死锁是否发生

一旦死锁发生则采取专门的措施，解除死锁并以最小的代价恢复操作系统运行

检测时机：

- 当进程由于资源请求不满足而等待时检测死锁（其缺点是系统的开销大）
- 定时检测
- 系统资源利用率下降时检测死锁

一个简单的死锁检测算法

* 每个进程和资源指定唯一编号
* 设置一张资源分配表

  记录各进程与其占用资源之间的关系
* 设置一张进程等待表

  记录各进程与要申请资源之间的关系

资源分配表		进程等待表	
r1	p2	p1	r1
r2	p5	p2	r3
r3	p4	p4	r4
r4	p1		
…	…	…	…

死锁的解除

重要的是以最小的代价恢复系统的运行

方法如下：

选择原则

1）撤消所有死锁进程

2）进程回退（Roll back）再启动

3）按照某种原则逐一撤消死锁进程，直到…

4）按照某种原则逐一抢占资源(资源被抢占的进程必须回退到之前的对应状态)，直到…

哲学家就餐问题

哲学家就餐问题讨论

- 何时发生死锁?
- 怎样从死锁中恢复?
- 怎样避免死锁的发生?
- 如何预防死锁? √

为防止死锁发生可采取的措施

- 仅当一个哲学家左右两边的筷子都可用时，才允许他拿筷子
- 给所有哲学家编号，奇数号的哲学家必须首先拿左边的筷子，偶数号的哲学家则反之
- 最多允许4个哲学家同时坐在桌子周围
- ……

哲学家就餐问题第二种解决方案

使用管程解决哲学家就餐问题

```
void philosopher[k=0 to 4]
/* the five philosopher clients */
{
while (true) {
      <think>;
      get_forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release_forks(k);   /* client releases forks via the monitor */
}
}
```

哲学家就餐问题第二种解决方案

```
monitor dining_controller;
cond ForkReady[5];
boolean fork[5] = {true};
void get_forks(int pid)
{
int left = pid;
int right = (++pid) % 5;
/*grant the left fork*/
if (!fork(left)
    cwait(ForkReady[left]);
 /* queue on condition variable */
fork(left) = false;
/*grant the right fork*/
if (!fork(right)
    cwait(ForkReady(right);
/* queue on condition variable */
fork(right) = false:
}
```

```
void release_forks(int pid)
{
int left = pid;
int right = (++pid) % 5;
/*release the left fork*/
if (empty(ForkReady[left])
/*no one is waiting for this fork */
    fork(left) = true;
else /* awaken a process waiting on this
fork */
    csignal(ForkReady[left]);
/*release the right fork*/
if (empty(ForkReady[right])
/*no one is waiting for this fork */
    fork(right) = true;
else /* awaken a process waiting on this
fork */
    csignal(ForkReady[right]);
}
```

哲学家就餐问题第三种解决方案(1)

```
#define  N  5
#define  THINKING  0
#define  HUNGRY    1
#define  EATING    2
#typedef  int semaphore;
int state[N];
semaphore mutex=1;
semaphore s[N];
```

为了避免死锁，把哲学家分为三种状态，思考，饥饿，进食，并且一次拿到两只筷子，否则不拿

哲学家就餐问题第三种解决方案(2)

```
void test(int i)
{
    if (state[ i ] == HUNGRY)
       && (state [(i-1) % 5] != EATING)
       && (state [(i+1) % 5] != EATING)
       {
          state[ i ] = EATING;
          V(&s[ i ]);
        }
 }
```

哲学家就餐问题第三种解决方案(3)

```
void  philosopher (int i)
{   while (true)
    {
     思考;
     P(&mutex);
     state[i] = HUNGRY;
     test(i);
     V(&mutex);
     P(&s[i]);
     拿左筷子;
     拿右筷子;
     进食;
     放左筷子;
     放右筷子;
     P(&mutex)
     state[ i ] = THINKING;
     test([i-1] % 5);
     test([i+1] % 5);
     V(&mutex);
    }
}
state[ i ] = THINKING
s[ i ] = 0
```

哲学家就餐问题第四种解决方案

```
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
        while (true) {
        think();
        P (room);
        P (fork[i]);
        P (fork [(i+1) mod 5]);
        eat();
        V (fork [(i+1) mod 5]);
        V (fork[i]);
        V (room);
}
}
void main()
{
        parbegin ( philosopher (0), philosopher (1),philosopher (2),
philosopher(3),philosopher (4) );
}
```

课堂讨论

- 画出5个进程陷入死锁的所有非同构模型

思考题（现代操作系统6.28）

计算机系学生想到了如下的一个消除死锁的方法。当某一进程请求一个资源时，规定一个时间限。如果进程由于得不到需要的资源而阻塞，计时器开始运行。当超过时间限时，进程会被释放掉，并且允许该进程重新执行。

如果你是教师，你会给这样的学生多少分？为什么？

Andrew S.Tanenbaum

重点小结

- 死锁、活锁和饥饿
- 资源分配图
- 死锁预防
  - 死锁产生的四个必要条件
  - 资源有序分配法
  - 虚设备技术
- 死锁避免
  - 银行家算法
- 死锁检测与解除
- 哲学家就餐问题

Thanks

The End