




## 第三章 处理机调度与死锁

- 3.1 处理机调度概述
- 3.2 调度算法
- 3.3 实时调度
- 3.4 死锁概述
- 3.5 预防死锁
- 3.6 避免死锁
- 3.7 死锁的检测与解除

韩明峰

QQ 2022446359





## 3.1 处理机调度概述

### 3.1.1 处理机调度的层次

#### 1. 高级调度(High Level Scheduling)

又称作业调度，根据算法将外存的作业调入内存，为它们创建进程、分配资源并放入就绪队列。

在某些系统中，新创建的进程最初处于换出状态。此时，它会进入到供中级调度使用的队列中等待调度。

主要用于多道批处理系统，不用于分时和实时系统。



## 2. 低级调度 (Low Level Scheduling)

又称进程调度，调度对象是进程。主要功能是根据算法，决定就绪队列中的哪个进程应获得处理机，恢复选中进程的处理机现场信息并运行。

多道批处理系统、分时和实时系统都需要进程调度。

### 3. 中级调度 (Intermediate Scheduling)

又称**内存调度**。为了使内存中同时存放的进程数目不至于太多，有时就需要把某些进程从内存中移到外存上，以减少多道程序的数目。

当以后内存有足够的空闲空间时，再将合适的进程重新换入内存，等待进程调度。

主要目的是为了提高内存利用率和系统吞吐量。



### 3.1.2 作业与作业调度

在多道批处理系统中，作业是用户提交给系统的一项相对独立的工作。

操作员把用户提交的作业通过相应的输入设备输入到磁盘存储器，并保存在一个**后备作业队列**中，再由作业调度程序将其从外存调入内存。

作业调度的主要任务：

- (1) 接纳多少作业
- (2) 接纳哪些作业
  - 1) 先来先服务调度算法
  - 2) 短作业优先的调度算法
  - 3) 优先级调度算法



### 3.1.3 进程调度


#### 1. 进程调度的时机

(1) 进程主动放弃处理机时。

1) 正在执行的进程执行完毕。

2) 正在执行的进程等待其他进程或系统发生的事件，或等待其他进程释放资源。

3) 正在执行的进程发出自愿放弃处理机的系统调用。



(2) 为支持可抢占的进程调度方式，有新进程就绪时。

1) 当中断处理程序处理完中断，如I/O中断引起某个阻塞进程变成就绪状态时，应该申请重新调度。

2) 当进程释放独占资源或发生某个事件，引起其他等待该资源或事件的进程从阻塞状态进入就绪状态时，应该申请重新调度。

3) 其它任何原因引起有进程从其它状态变成就绪状态，如进程被中级调度选中时。





(3) 为支持可抢占的进程调度方式，没有新进程就绪时，以下情况也可进行调度。

1) 当时钟中断发生, 时钟中断处理程序调用有关时间片的处理程序，发现正运行进程时间片到。

2) 在按进程优先级进行调度的操作系统中，任何原因引起进程的优先级发生变化时，应请求重新调度。

如进程通过系统调用自愿改变优先级；或者系统处理时钟中断时，根据各进程等待处理机的时间长短而调整进程的优先级。







## 2. 进程调度方式

### (1) 非抢占方式 (Nonpreemptive Mode)

在采用这种调度方式时，一旦把处理机分配给某进程后，就一直让它运行下去，直至该**进程完成**，或等待某事件而被**阻塞**时，才把处理机分配给其它进程。

### (2) 抢占方式 (Preemptive Mode)

调度程序根据某种原则，去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。

### 3. 进程切换

当请求调度的事件发生后，才会运行进程调度程序；  
当选中了新的就绪进程后，才会进行进程间的切换。

进程切换包括：

(1) 保存当前进程的现场信息，包括PC、PSW、通用寄存器和堆栈指针；

(2) 恢复被调度进程的现场信息，包括堆栈指针、通用寄存器，用被调度进程的页表指针更新页表基址寄存器，使快表中的数据项无效，恢复PC、PSW，从而运行新的进程。



### 3.1.4 处理机调度算法的目标

#### 1. 批处理系统中处理机调度算法的目标

##### (1) 平均周转时间短。

周转时间，指从作业被提交给系统开始，到作业完成为止的这段时间间隔。包括四部分时间：①作业在外存后备队上等待调度的时间；②进程在就绪队列上等待进程调度的时间；③进程在CPU上执行的时间；④进程等待I/O操作完成的时间。可把平均周转时间描述为：

$$T = \frac{1}{n} \left[ \sum_{i=1}^n T_i \right]$$



(2) 系统吞吐量高。吞吐量是指在单位时间内系统所完成的作业数。

(3) 处理机利用率高。



## 2. 分时系统中处理机调度算法的目标

(1) 保证响应时间快。响应时间，从用户通过键盘提交一个请求开始，直到屏幕上显示出处理结果为止的一段时间间隔。

(2) 保证均衡性。系统响应时间的快慢应与用户所请求的复杂性相适应。

## 3. 实时系统中处理机调度算法的目标

**截止时间**的保证。

**截止时间：**某任务必须**开始执行**的最迟时间，或必须**完成**的最迟时间。



## 3.2 调度算法

### 3.2.1 先来先服务 (first-come first-served, FCFS) 调度算法

最简单的调度算法，可以用于作业调度和进程调度。

当在**作业调度**中采用该算法时，系统将从**后备队列**中选择最先进入该队列的作业，将它们调入内存；

当在**进程调度**采用FCFS算法时，从**就绪的进程队列**选择最先进入的进程。



### 3.2.2 短作业优先(short job first, SJF)的调度算法

#### 1. 短作业优先算法

SJF算法是以作业的长短来计算优先级，作业越短，其优先级越高。SJF算法可以用于作业调度和进程调度。

#### 2. 短作业优先算法的缺点

- 1) 必须预知作业的运行时间。
- 2) 对长作业非常不利，长作业的周转时间会明显地增长。
- 3) 在采用SJF算法时，人-机无法实现交互。
- 4) 该调度算法完全未考虑作业的**紧迫程度**，故不能保证紧迫性作业能得到及时处理。





### 3.2.3 优先级调度算法

基于进程的紧迫程度，由外部赋予进程相应的优先级，根据该优先级进行调度。

可用于作业调度和进程调度。

#### 1. 优先级调度算法的类型

把处理机分配给就绪队列中优先级最高的进程。这时，又可进一步把该算法分成如下两种。

- (1) 非抢占式优先级调度算法。
- (2) 抢占式优先级调度算法。



## 2. 优先级的类型



### (1) 静态优先级

静态优先级是在创建进程时确定的，在进程的整个运行期间保持不变。确定进程优先级大小的依据有如下三个：

- 1) 进程类型。系统进程的优先级高于一般用户进程。
- 2) 进程对资源的需求。对资源要求少的优先级高。
- 3) 紧迫程度高的进程优先级高。

### (2) 动态优先级

动态优先级是指在创建进程之初，先赋予其一个优先级，然后其值随进程的推进或等待时间的增加而改变，以便获得更好的调度性能。





### 3.2.4 轮转调度算法

#### 1. 轮转法的基本原理

(1) 在轮转 (RR) 法中，系统将所有的就绪进程按FCFS策略排成一个就绪队列。

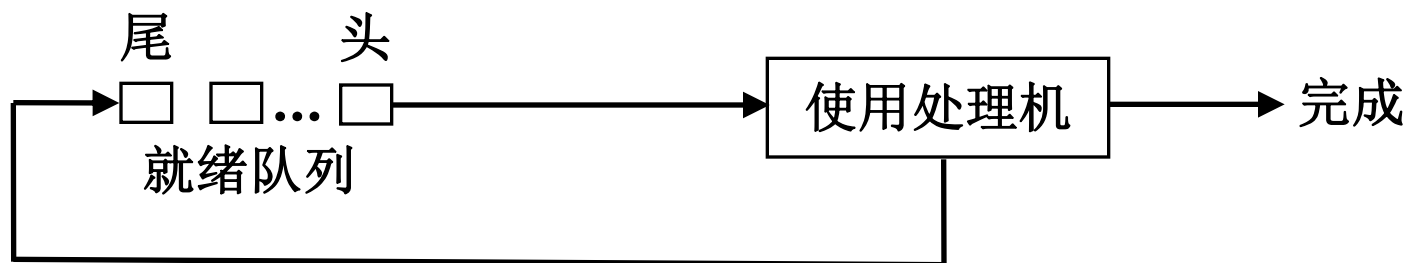
(2) 把CPU分配给队首进程，并令其执行一个时间片。当它运行完毕后，又把处理机分配给就绪队列中新的队首进程，也让它执行一个时间片。

(3) 保证就绪队列中的所有进程在确定的时间段内，都能获得一个时间片的处理机时间。

## 2. 进程调度时机

(1) 若一个时间片未用完，正在运行的进程便已经完成，则立即激活调度程序。

(2) 在一个时间片用完时，计时器中断处理程序被激活。如果进程尚未运行完毕，就把他送往就绪队列的尾部。



时间片用完，抢占

时间片轮转调度算法示意图

### 3.2.5 多级队列调度算法

该算法将系统中的进程就绪队列从一个拆分为若干个，将不同类型或性质的进程**固定**分配在不同的就绪队列。

- (1) 不同的就绪队列采用不同的调度算法；
- (2) 队列之间通常采用固定优先级抢占调度，也可以划分时间片。



如系统有前台进程队列和后台进程队列：

- (1) 前台队列采用时间片轮转调度算法；
- (2) 后台队列采用先来先服务调度算法；
- (3) 前台队列的优先级高于后台队列。



### 3.2.6 多级反馈队列(multileved feedback queue)调度算法

(1) 设置多个就绪队列，并为每个队列赋予不同的优先级和不同的时间片。优先级越高，时间片越小。



(2) 每个队列都采用FCFS算法。

①当新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则等待调度。

②当轮到该进程执行时，如它能在该时间片内完成，便可撤离系统。否则，即它在一个时间片结束时尚未完成，调度程序将其转入第二队列的末尾等待调度；

③如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，依此类推。

④当进程最后被降到第 $n$ 队列后，在第 $n$ 队列中便采取按RR方式运行。





### (3) 按队列优先级调度。

仅当第 $1 \sim (i-1)$ 所有队列均空时，才会调度第 $i$ 队列中的进程运行。

如果处理机正在第 $i$ 队列中为某进程服务时又有新进程进入任一优先级较高的队列，此时须立即把正在运行的进程放回到第 $i$ 队列的末尾，而把处理机分配给新到的高优先级进程。

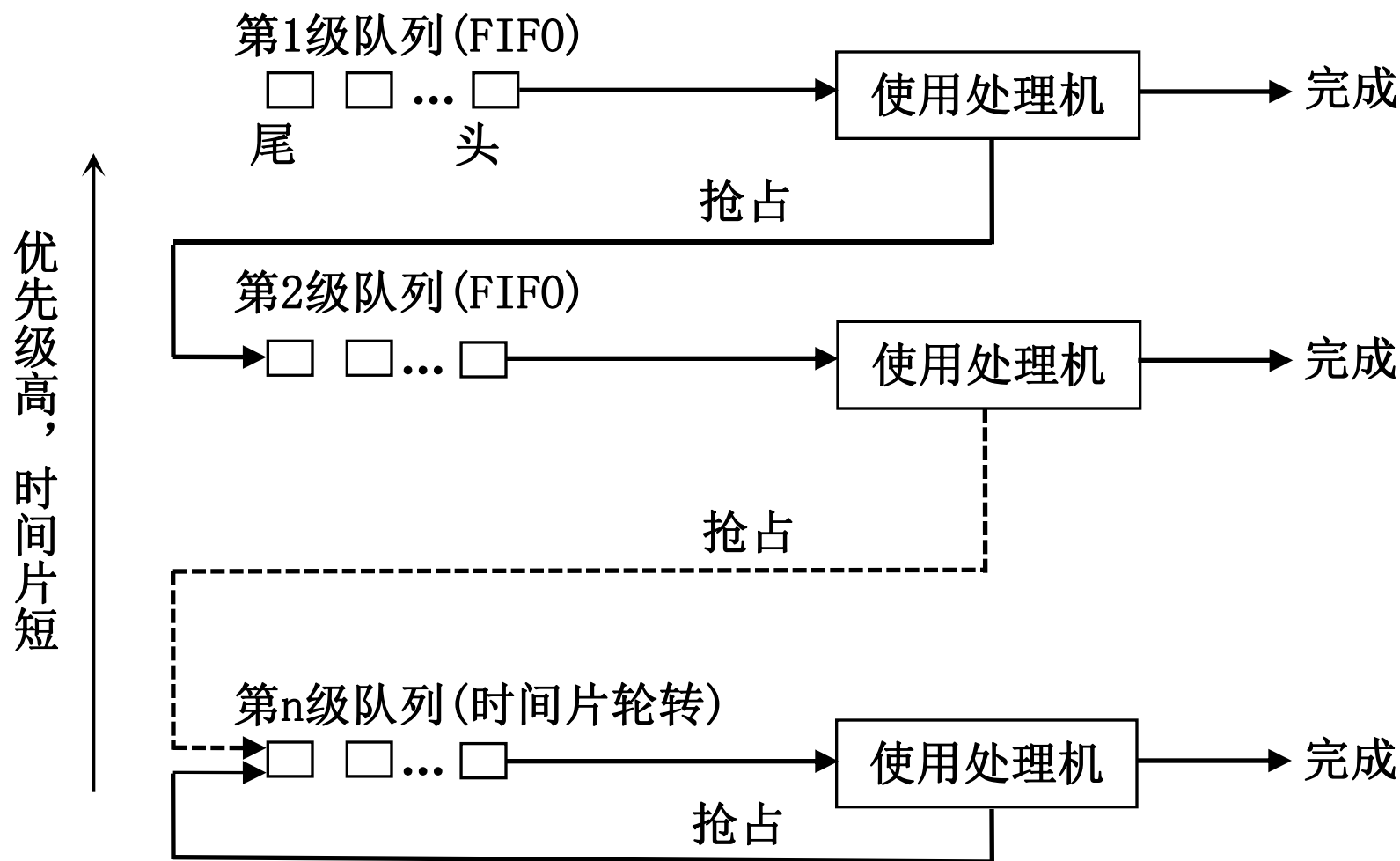


图3-4 多级反馈队列调度算法



### 3.2.7 基于公平原则的调度算法

#### 1. 保证进程公平的调度算法

保证调度算法向用户所做出的保证并不是优先运行，而是明确的性能保证，该算法可以做到调度的公平性。

一种比较容易实现的性能保证是处理机分配的公平性。如果在系统中有 $n$ 个相同类型的进程同时运行，为公平起见，须保证每个进程都获得相同的处理机时间 $1/n$ 。

#### 2. 保证用户公平的调度算法

在该调度算法中，调度的公平性主要是针对用户而言，使所有用户（每用户进程数不同）能获得相同的处理机时间，或所要求的时间比例。



## 3.3 实时调度

### 3.3.1 实现实时调度的基本条件

#### 1. 系统处理能力强

假定系统中有 $m$ 个周期性的硬实时任务HRT，它们的处理时间可表示为 $C_i$ ，周期时间表示为 $P_i$ ，则在单处理机情况下，必须满足下面的限制条件系统才是可调度的：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$



提高系统处理能力的途径有二：

(1) 采用单处理机系统，但须增强其处理能力，以显著地减少对每一个任务的处理时间；

(2) 采用多处理机系统。假定系统中的处理机数为N，则应将上述的限制条件改为：

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq N$$



## 2. 采用基于优先级的抢占式调度机制

### 3. 中断延迟和调度延迟最小化

(1) 中断延迟指从中断发生到中断服务程序开始执行的时间。应使禁止中断的时间尽量短。

(2) 调度延迟指从进程就绪到进程运行的时间。应使禁止调度的时间尽量短。

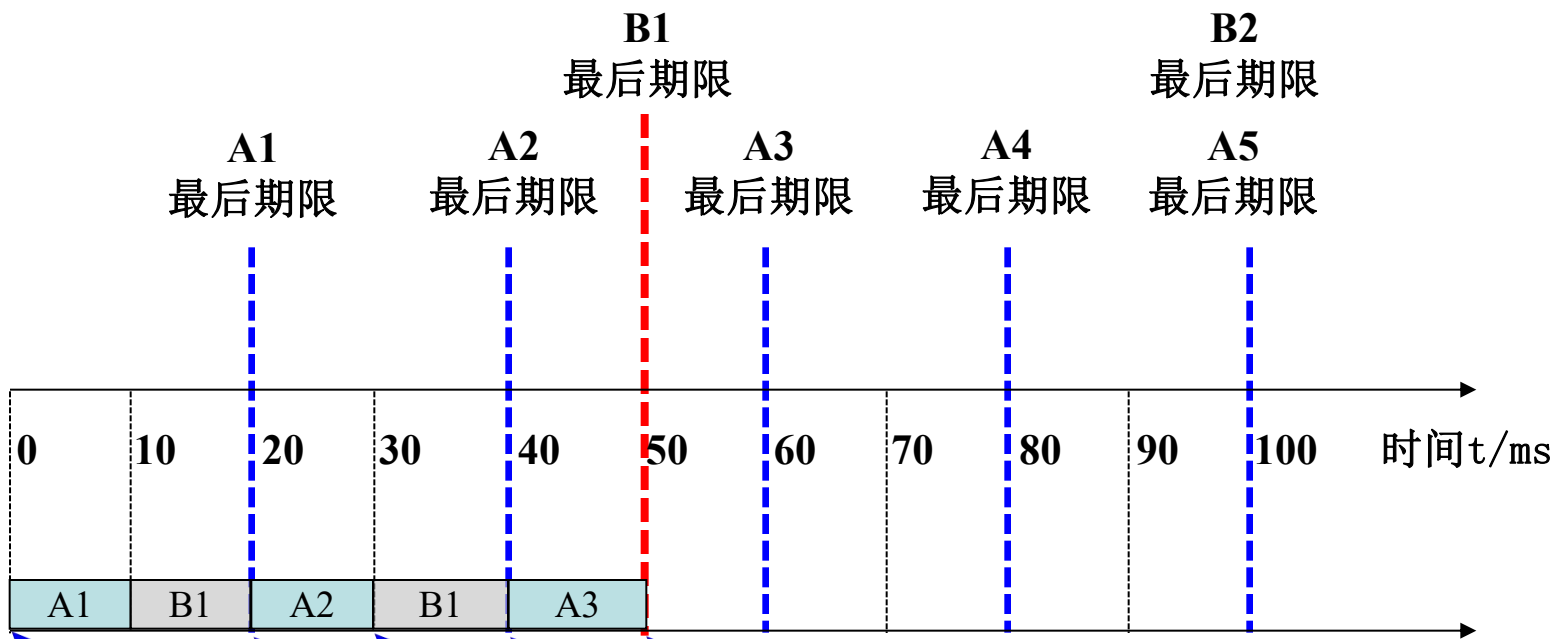
### 3.3.2 最早完成截止时间优先算法

图3-7示出了将**最早完成截止时间优先**用于周期实时任务之例。在该例中有两个周期任务，任务A和任务B的周期时间分别为20 ms和50 ms，每个周期的处理时间分别为10 ms和25 ms。

**注意：**此时

$$\sum_{i=1}^m \frac{C_i}{P_i} = 1$$





$t=0\text{ms}$ 时，  
行，在A1  
度B1以

$t=20$

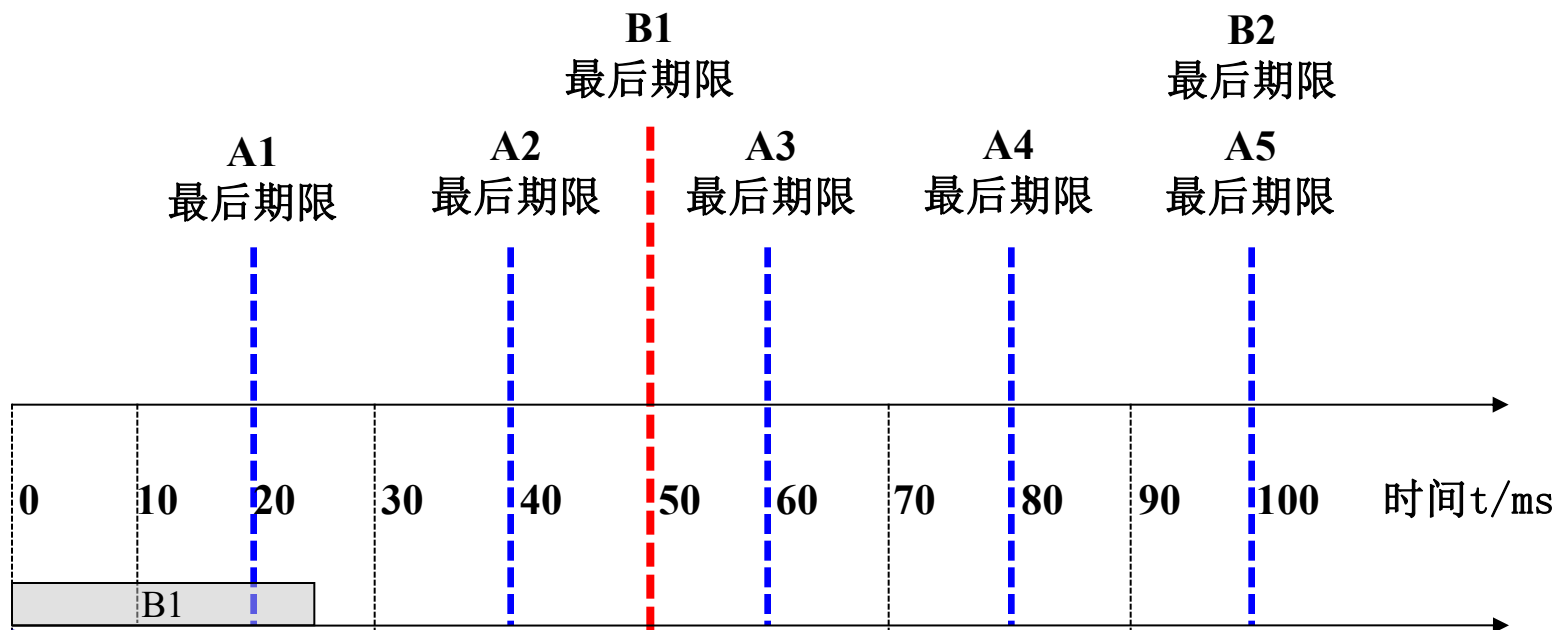
在 $t=$

在 $t=4$

在 $t=50\text{ms}$ 时，虽然A3  
完成，但B1已错过了  
它的最后期限

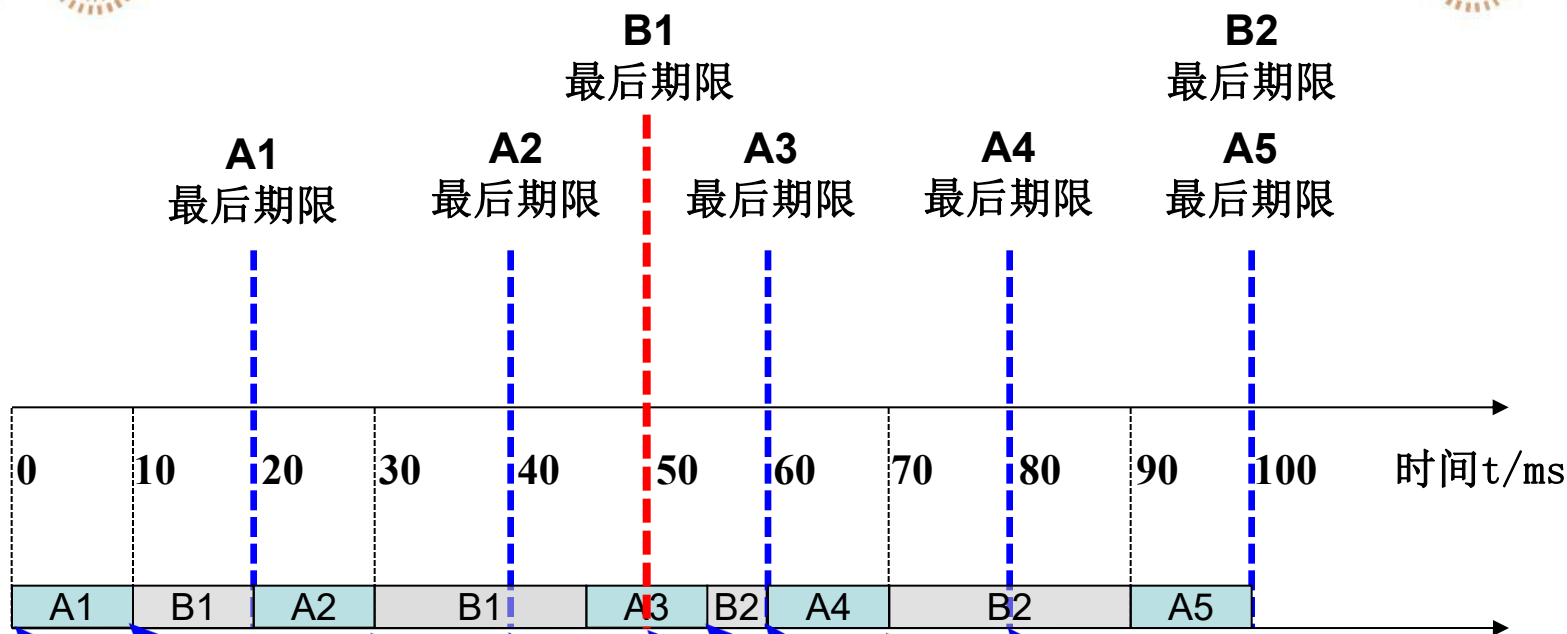
这说明利用通常的固定优先级调度已经失败！

图3-7 (1) 固定优先级调度图示 (A有较高优先级)



在 $t=0ms$ 时，先调度B1执行，A1已错过最后期限

图3-7 (2) 固定优先级调度图示 (B有较高优先级)



在  $t=10\text{ms}$  时，A1 到达，调度 A1 执行

在  $t=30\text{ms}$  时，A2 到达，调度 A2 执行，B1 中断而调

在  $t=45\text{ms}$  时，A3 到达，由于 A3 的截止时间比 B2 早，所以 A3 继续执行

在  $t=50\text{ms}$  时，B2 到达，由于 B2 的截止时间比 A3 的截止时间晚，所以 A3 继续执行

在  $t=100\text{ms}$  时，B2 和 A5 均完成，满足调度要求

图3-7 (3) 使用最早完成截止时间优先算法图示



### 3.3.3 优先级倒置(priority inversion problem)

#### 1. 优先级倒置的形成

即高优先级进程(或线程)被低优先级进程(或线程)阻塞, 又因中优先级进程的存在而延长了高优先级进程被阻塞的时间。

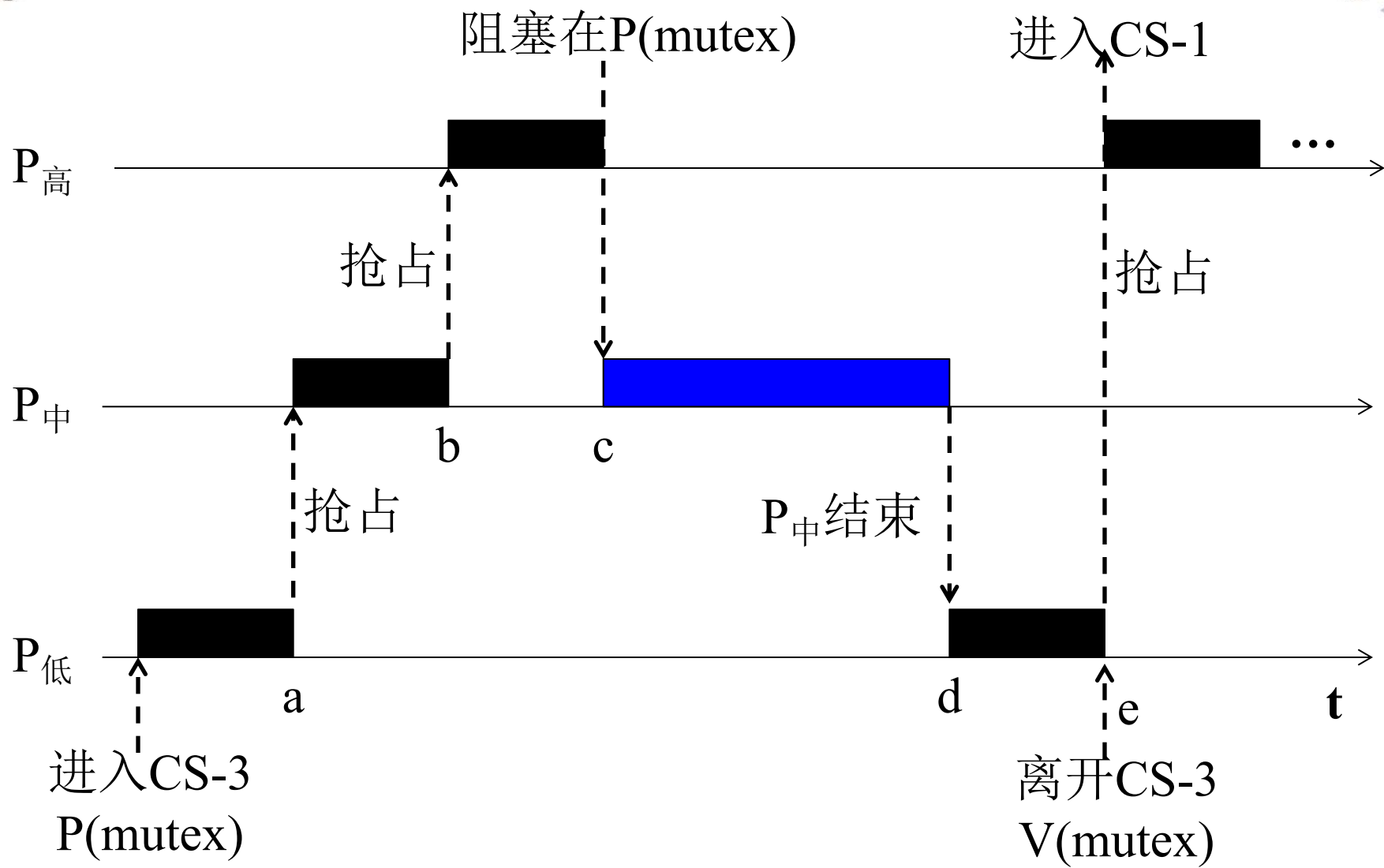


图3-10 优先级倒置示意图



## 2. 优先级倒置的解决方法

### (1) 动态优先级继承方法。

当高优先级任务在资源上阻塞时，占用资源的低优先级任务继承高优先级任务的优先级；

当低优先级任务释放资源时，恢复其原先的优先级。

### (2) 优先级置顶方法。

访问资源的任务被设置为比使用该资源的任务中最高优先级高一级的优先级；

当任务释放资源时，恢复其原先的优先级。

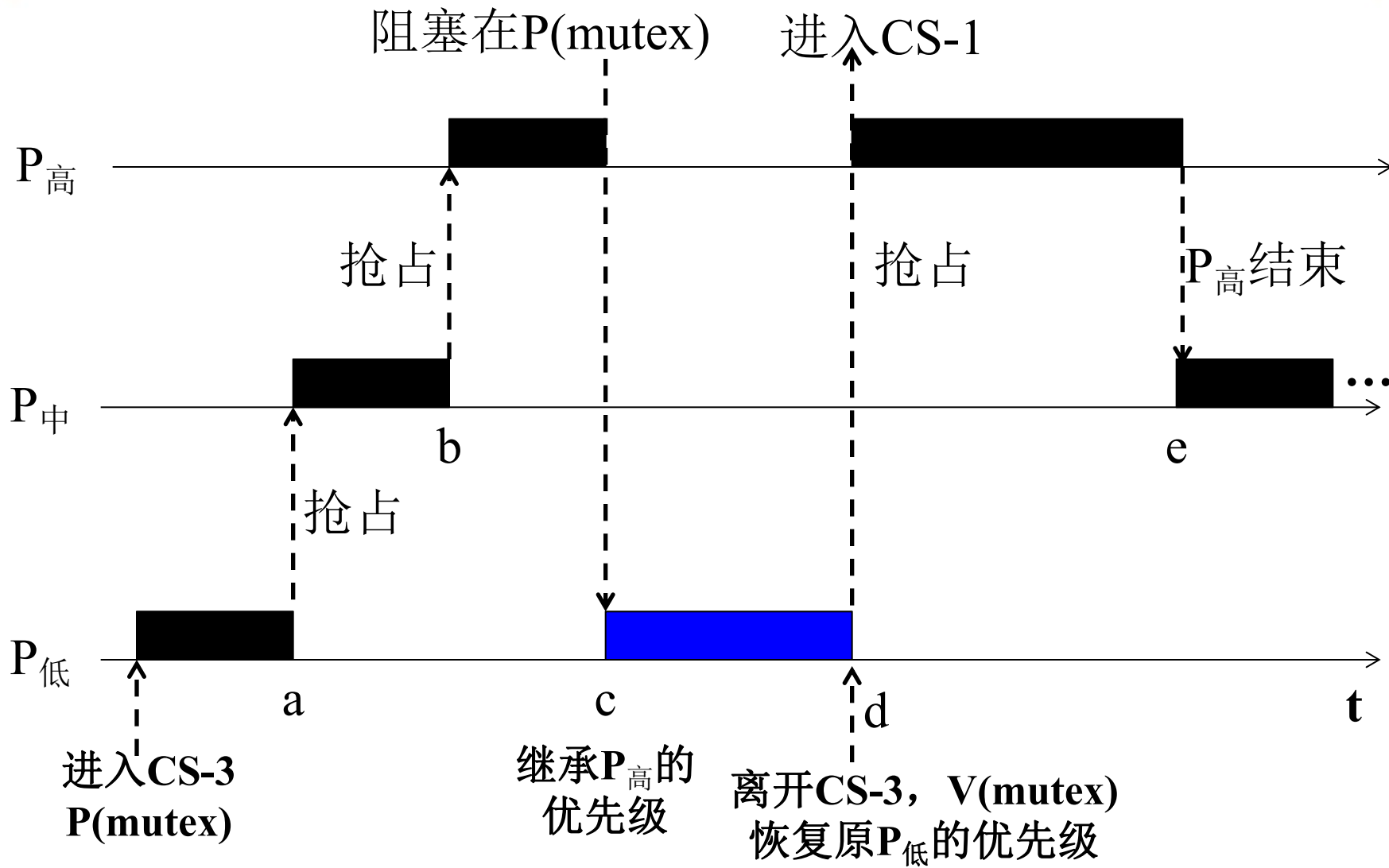
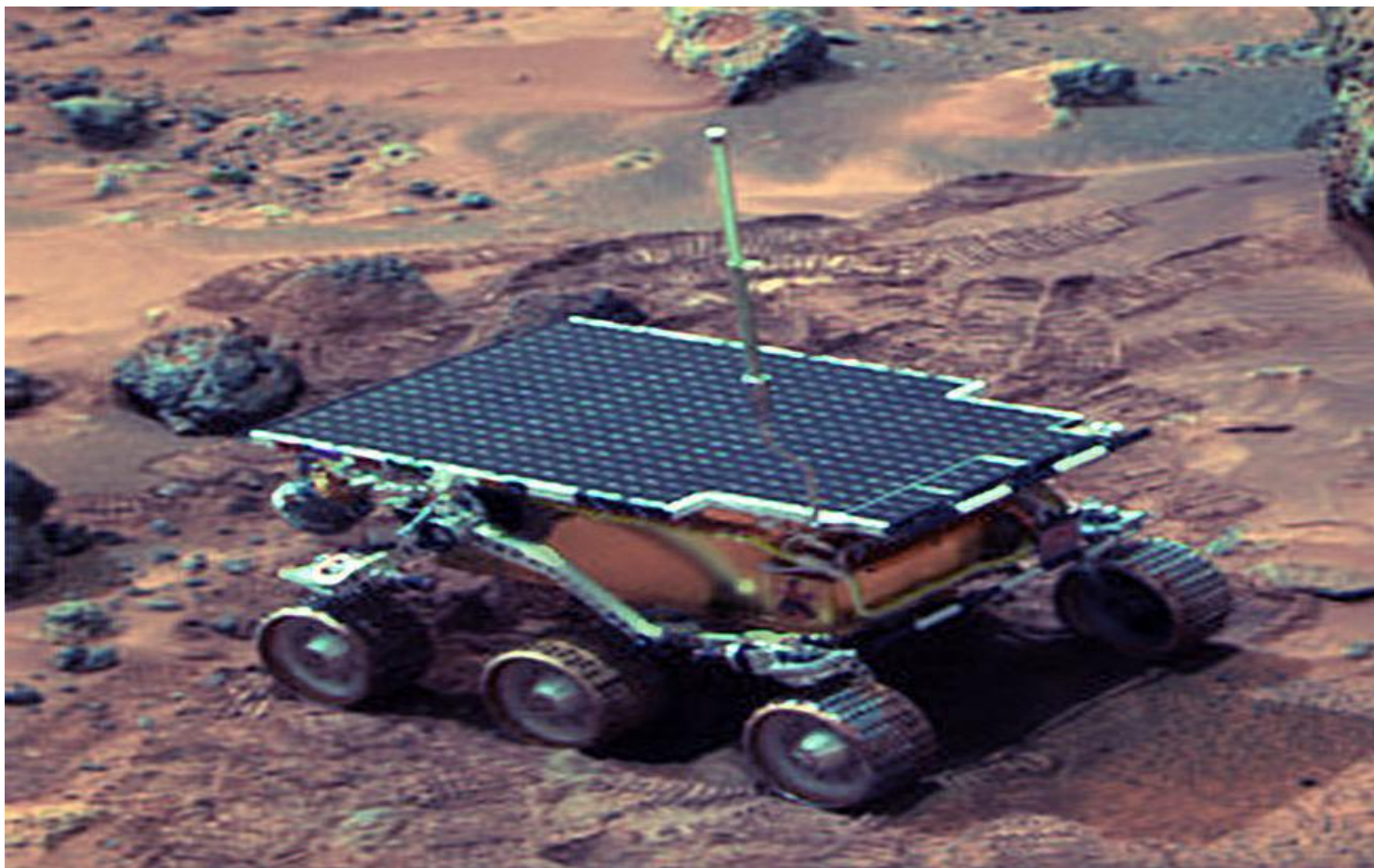




图3-11 采用了动态优先级继承方法的运行情况



### 3. 优先级倒置问题在“Sojourner”火星车上的发现



采用实时嵌入式操作系统VxWorks的“Sojourner”火星车



(1) 气象数据采集任务作为一个低优先级线程运行，并且使用互斥信号量来访问共享存储区；

(2) 一个高优先级的共享存储区管理任务，也以互斥信号量来访问共享存储区；

(3) 一个长期运行的、具有比气象任务高、但是比共享存储区管理任务低的通信任务，阻止了共享存储区管理任务的运行。

(4) 不久，一个看门狗定时器注意到共享存储区管理任务已经很长时间没有被执行了，一定是出了什么问题，所以强制系统复位，信息和图像传送就被一系列的复位所中断。



## 3.4 死锁概述

### 3.4.1 资源问题

#### 1. 排它性资源

如打印机，文件等。

#### 2. 可抢占性资源和不可抢占性资源

(1) 可抢占性资源，如CPU和主存。

(2) 不可抢占性资源，如打印机等。

### 3.4.2 计算机系统死锁

#### 1. 竞争排它性且不可抢占性资源引起死锁

如两个进程P1和P2准备写两个文件F1和F2。

P1

...

①Open (F1, w);

③Open (F2, w);

P2

...

②Open (F2, w);

④Open (F1, w);

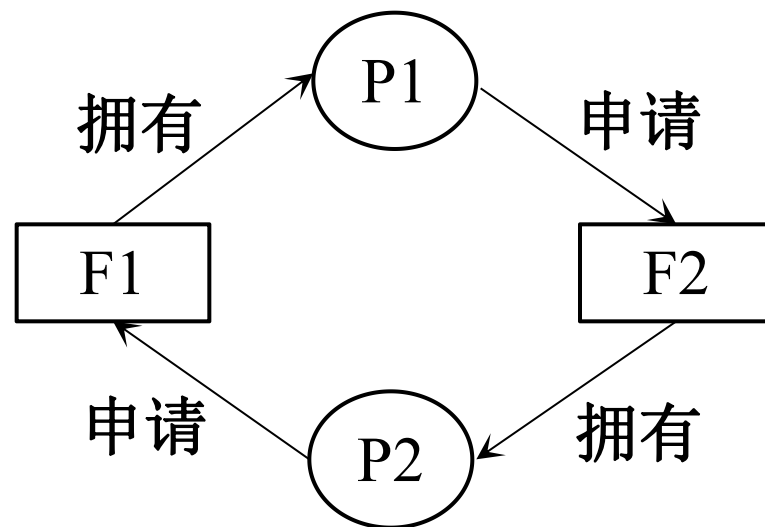


图3-12 共享文件时的死锁情况

## 2. 通信死锁

如在利用消息通信机制进行通信时所形成的死锁情况。

正常

$P_1$ :	... send( $P_2$ , $m_1$ );	receive( $P_3$ , $m_3$ );	...
$P_2$ :	... send( $P_3$ , $m_2$ );	receive( $P_1$ , $m_1$ );	...
$P_3$ :	... send( $P_1$ , $m_3$ );	receive( $P_2$ , $m_2$ );	...

死锁

$P_1$ :	...receive( $P_3$ , $m_3$ );	send( $P_2$ , $m_1$ );	...
$P_2$ :	...receive( $P_1$ , $m_1$ );	send( $P_3$ , $m_2$ );	...
$P_3$ :	...receive( $P_2$ , $m_2$ );	send( $P_1$ , $m_3$ );	...







### 3.4.3 死锁的定义、必要条件和处理方法

#### 1. 死锁的定义

如果一个进程集合中的每个进程都在等待只能由该进程集合中的其它进程才能引发的事件，那么该进程集合就是死锁的。

我们主要讨论排它性、不可抢占的**资源死锁**。



## 2. 产生（资源）死锁的必要条件

产生死锁必须同时具备下面四个必要条件，只要其中任一个条件不成立，死锁就不会发生：

- (1) 互斥条件。
- (2) 占有（已获得资源）并等待（新的资源）条件。
- (3) 不可抢占条件。
- (4) 循环等待条件。





### 3. 处理死锁的方法

目前处理死锁的方法可归结为四种：

- (1) 预防死锁。
- (2) 避免死锁。
- (3) 检测死锁。
- (4) 解除死锁。

## 3.5 预 防 死 锁

预防死锁的方法是通过破坏产生死锁的四个必要条件中的一个或几个，以避免发生死锁。

由于互斥条件是非共享设备所必须的，不仅不能改变，还应加以保证，因此主要是破坏产生死锁的后三个条件。



### 3.5.1 破坏“占有并等待”条件

#### 1. 第一种协议

所有进程在开始运行之前，必须一次性地申请其在整个运行过程中所需的全部资源。

#### 2. 第二种协议

先申请初期所需资源，用后释放，再申请新的所需资源。

### 3.5.1 破坏“占有并等待”条件

举例：进程先将数据从磁带复制到磁盘文件上，然后对磁盘文件进行排序，最后把结果打印出来。

两种协议分配资源的差异：

第一种协议：

一次性地申请磁带，磁盘文件，打印机。

第二种协议：

- (1) 先申请磁带，磁盘文件，用后释放；
- (2) 再申请磁盘文件，打印机。



### 3.5.2 破坏“不可抢占”条件

当一个进程处于等待资源的状态时，如果其他进程申请其已经拥有的资源时，那么该进程的部分资源可以被抢占。只有当一个进程分配到申请的资源，并且恢复在等待时被抢占的资源，它才能重新执行。

只有在资源状态可以很容易地保存和恢复的情况下，这种方法才是实用的，如处理机和主存。



### 3.5.3 破坏“循环等待”条件

对系统所有资源类型进行线性排序，并赋予不同的序号。所有进程对资源的请求必须严格按照资源序号递增的次序提出。同类资源必须一起请求。

例如：一个进程在开始时，可以请求某类资源 $R_i$ 的单元。以后，当且仅当 $j > i$ 时，进程才可以请求资源 $R_j$ 的单元。

**难点：**在采用这种策略时，应如何来规定每种资源的序号十分重要。

## 3.6 避免死锁

避免死锁同样是属于事先预防的策略，但并不是事先采取某种限制措施，破坏产生死锁的必要条件，而是在资源动态分配过程中，防止系统进入不安全状态，以避免发生死锁。





### 3.6.1 系统安全状态

#### 1. 安全状态

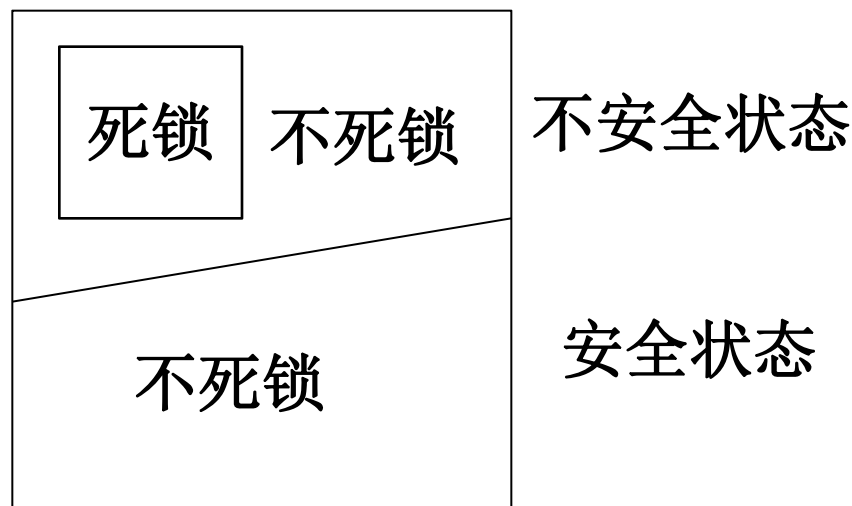
安全状态指系统能按某种进程顺序( $P_1, P_2, \dots, P_n$ )来为每个进程 $P_i$ 分配其所需资源,直至满足每个进程对资源的最大需求,使每个进程都可顺利地完成。

如果系统无法找到这样一个安全序列,则称系统处于不安全状态。

## 1. 安全状态

**结论：**当系统处于安全状态时，可避免发生死锁；反之，当系统处于不安全状态时，则**可能**进入到死锁状态。

**注意：**我们假设了一种最坏的情形：进程一次性申请全部剩余资源，并且在退出以前都会保持所有已获取的资源。



安全、不安全和死锁的状态空间图示

## 2. 安全状态之例

假定系统中三个进程 $P_1$ 、 $P_2$ 和 $P_3$ ，共有12台磁带机，某时刻分配情况如下表所示：

进程	最大需求	已分配	还需要	可用
<b>P1</b>	<b>10</b>	<b>5</b>	<b>5</b>	<b>3</b>
<b>P2</b>	<b>4</b>	<b>2</b>	<b>2</b>	
<b>P3</b>	<b>9</b>	<b>2</b>	<b>7</b>	

在 $T_0$ 时刻，存在一个安全序列 $\langle P_2, P_1, P_3 \rangle$ ，因此是安全的。

### 3. 由安全状态向不安全状态的转换

若在  $T_0$  时刻以后  $P_3$  又请求 1 个资源，系统将资源分配给  $P_3$ ，则进入不安全状态。

进程	最大需求	已分配	还需要	可用
P1	10	5	5	2
P2	4	2	2	
P3	9	3	6	

### 3.6.2 利用银行家算法避免死锁

在避免死锁方法中允许进程动态地申请资源，但系统在进行资源分配之前，应先计算此次分配资源的安全性，若分配不会导致系统进入不安全状态，则分配，否则等待。

#### 1. 银行家算法中的数据结构

- (1) 可利用资源向量Available。
- (2) 最大需求矩阵Max。
- (3) 分配矩阵Allocation。
- (4) 需求矩阵Need ( $\text{Max} - \text{Allocation}$ )。

## 2. 安全性算法

系统所执行的安全性算法可描述如下：

(1) 设置两个向量：

① 工作向量Work，它表示系统可提供给进程继续运行所需的各类资源数目，它含有m个元素，在执行安全算法开始时， $Work = Available$ ；

② Finish：它表示系统是否有足够的资源分配给进程，使之运行完成。开始时先做 $Finish[i] = false$ ；当有足够资源分配给进程时，再令 $Finish[i] = true$ 。



(2) 从进程集合中找到一个能满足下述条件的进程:

①  $\text{Finish}[i] = \text{false};$

②  $\text{Need}[i] \leq \text{Work};$

若找到, 执行步骤(3), 否则, 执行步骤(4)。

(3) 当进程 $P_i$ 获得资源后, 可顺利执行, 直至完成, 并释放出分配给它的资源, 故应执行:

$\text{Work}[j] = \text{Work}[j] + \text{Allocation}[i, j];$

$\text{Finish}[i] = \text{true};$

go to (2);

(4) 如果所有进程的 $\text{Finish}[i] = \text{true}$ 都满足, 则表示系统处于安全状态; 否则, 系统处于不安全状态。




### 3. 资源请求算法

设 $\text{Request}_i$ 是进程 $P_i$ 的请求向量，如果 $\text{Request}_i[j]=K$ ，表示进程 $P_i$ 需要 $K$ 个 $R_j$ 类型的资源。当 $P_i$ 发出资源请求后，系统按下述步骤进行检查：

(1) 如果 $\text{Request}_i[j] \leq \text{Need}[i, j]$ ，便转向步骤(2)；否则认为出错，因为它所需要的资源数已超过它所宣布的最大值。

(2) 如果 $\text{Request}_i[j] \leq \text{Available}[j]$ ，便转向步骤(3)；否则，表示尚无足够资源， $P_i$ 须等待。



(3) 系统**试探着**把资源分配给进程 $P_i$ ，并修改下面数据结构中的数值：

$$\text{Available}[j] = \text{Available}[j] - \text{Request}_i[j];$$

$$\text{Allocation}[i, j] = \text{Allocation}[i, j] + \text{Request}_i[j];$$

$$\text{Need}[i, j] = \text{Need}[i, j] - \text{Request}_i[j];$$

(4) 系统执行**安全性算法**，检查此次资源分配后系统是否处于安全状态。

若安全，才正式将资源分配给进程 $P_i$ ，以完成本次分配；否则，将本次的试探分配作废，恢复原来的资源分配状态，让进程 $P_i$ 等待。

#### 4. 银行家算法之例

假定系统中有五个进程  $\{P_0, P_1, P_2, P_3, P_4\}$  和三类资源  $\{A, B, C\}$ ，各种资源的数量分别为10、5、7，在 $T_0$ 时刻的资源分配情况如图3-15所示。



资源 情况 进程	Max			Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
$P_0$	7	5	3	0	1	0	7	4	3	3	3	2
$P_1$	3	2	2	2	0	0	1	2	2			
$P_2$	9	0	2	3	0	2	6	0	0			
$P_3$	2	2	2	2	1	1	0	1	1			
$P_4$	4	3	3	0	0	2	4	3	1			

图3-15  $T_0$ 时刻的资源分配表

(1)  $T_0$ 时刻的安全性：利用安全性算法对 $T_0$ 时刻的资源分配情况进行分析(如图3-16所示)可知，在 $T_0$ 时刻存在着一个安全序列  $\{P_1, P_3, P_4, P_2, P_0\}$ ，故系统是安全的。

资源情况 进程	Work			Need			Allocation			Work+=Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
$P_1$	3	3	2	1	2	2	2	0	0	5	3	2	true
$P_3$	5	3	2	0	1	1	2	1	1	7	4	3	true
$P_4$	7	4	3	4	3	1	0	0	2	7	4	5	true
$P_2$	7	4	5	6	0	0	3	0	2	10	4	7	true
$P_0$	10	4	7	7	4	3	0	1	0	10	5	7	true

图3-16  $T_0$ 时刻的安全序列



(2)  $P_1$ 请求资源:  $P_1$ 发出请求向量 $\text{Request}_1(1, 0, 2)$ , 系统按银行家算法进行检查:



- ①  $\text{Request}_1(1, 0, 2) \leq \text{Need}_1(1, 2, 2)$ ;
- ②  $\text{Request}_1(1, 0, 2) \leq \text{Available}_1(3, 3, 2)$ ;
- ③ 系统先假定可为 $P_1$ 分配资源, 并修改 $\text{Available}$ ,  $\text{Allocation}_1$ 和 $\text{Need}_1$ ;
- ④ 再利用安全性算法检查此时系统是否安全。如图3-17所示, 因此可立即把资源分配给 $P_1$ 。

**注意:** 因为在(1)中已求出安全序列 $\{P_1, P_3, P_4, P_2, P_0\}$ , 因此(2)只需判断①和②满足即可。

资源情况 进程	Work			Need			Allocation			Work+=Allocation			Finish
	A	B	C	A	B	C	A	B	C	A	B	C	
P <sub>1</sub>	2	3	0	0	2	0	3	0	2	5	3	2	true
P <sub>3</sub>	5	3	2	0	1	1	2	1	1	7	4	3	true
P <sub>4</sub>	7	4	3	4	3	1	0	0	2	7	4	5	true
P <sub>2</sub>	7	4	5	6	0	0	3	0	2	10	4	7	true
P <sub>0</sub>	10	4	7	7	4	3	0	1	0	10	5	7	true

预分配 (1, 0, 2)

图3-17 P<sub>1</sub>申请资源时的安全性检查





(3)  $P_4$ 请求资源:  $P_4$ 发出请求向量 $\text{Request}_4(3, 3, 0)$ ,  
系统按银行家算法进行检查:

①  $\text{Request}_4(3, 3, 0) \leq \text{Need}_4(4, 3, 1)$ ;

②  $\text{Request}_4(3, 3, 0) > \text{Available}(2, 3, 0)$  (已分配给 $P_1$ 后的 $\text{Available}$ )。

让 $P_4$ 等待。





(4)  $P_0$ 请求资源:  $P_0$ 发出请求向量 $\text{Request}_0(0, 2, 0)$ , 系统按银行家算法进行检查:

①  $\text{Request}_0(0, 2, 0) \leq \text{Need}_0(7, 4, 3)$ ;

②  $\text{Request}_0(0, 2, 0) \leq \text{Available}(2, 3, 0)$  (已分配给 $P_1$ 后的 $\text{Available}$ )。

③ 系统先假定可为 $P_0$ 分配资源, 并修改有关数据, 如图3-18所示。

预分配 (0, 2, 0)

分配 (1, 0, 2), 预  
分配 (0, 2, 0)

资源情况	Allocation			Need			Available		
进程	A	B	C	A	B	C	A	B	C
P <sub>0</sub>	0	3	0	7	2	3	2	1	0
P <sub>1</sub>	3	0	2	0	2	0			
P <sub>2</sub>	3	0	2	6	0	0			
P <sub>3</sub>	2	1	1	0	1	1			
P <sub>4</sub>	0	0	2	4	3	1			

分配  
(1, 0, 2)

图3-18 为P<sub>0</sub>分配资源后的有关资源数据

④进行安全性检查：可用资源Available(2, 1, 0)已不能满足任何进程的需要，故系统进入不安全状态，此时系统不分配资源。



## 5. 为什么一般不采用银行家算法？

- (1) 很少有进程在运行前就能知道所需的最大资源数。
- (2) 系统的进程数也不是固定的。

## 3.7 死锁的检测与解除

如果在系统中，既不采取死锁预防措施，也未配有死锁避免算法，系统很可能会发生死锁。在这种情况下，系统应当提供两个算法：

① 死锁检测算法。该方法用于检测系统状态，以确定系统中是否发生了死锁。

② 死锁解除算法。当认定系统中已发生了死锁，利用该算法可将系统从死锁状态中解脱出来。

### 3.7.1 死锁的检测

#### 1. 死锁检测算法

死锁检测中的数据结构类似于银行家算法中的数据结构：

(1) 可利用资源向量Available，它表示了m类资源中每一类资源的可用数目。如果 $Allocation_i=0$ ，则 $Finish[i]=true$ ，否则 $Finish[i]=false$ 。

(2) 找这样的i，同时满足

①  $Finish[i]=false$ ;

②  $Request_i \leq Work$ 的进程，找不到则转（4）。

(3)  $Work = Work + Allocation_i$ ，  
 $Finish[i]=true$ 。

转到（2）。

(4) 如果有 $Finish[i]=false$ ，则系统死锁。而且如果 $Finish[i]=false$ ，则 $P_i$ 死锁。



**注意：**死锁检测算法使用 $\text{Request}_i \leq \text{Work}$ ，而不是安全性算法中的 $\text{Need}[i] \leq \text{Work}$ 。

**分析：**“在死锁检测算法（3）中，我们乐观地认为 $P_i$ 不再需要更多资源以完成任务，它将返回已分配的所有资源。

如果我们的假设不正确，随后会发生死锁，这可以在下一次死锁检测算法运行时被检测到。”



### 3.7.2 死锁的解除

终止进程：

- (1) 终止所有死锁进程。
- (2) 逐个终止进程，直到消除死锁循环为止。