第25 - 26讲 进程死锁及解决方法

§2.7 Concurrency Deadlock And Starvation

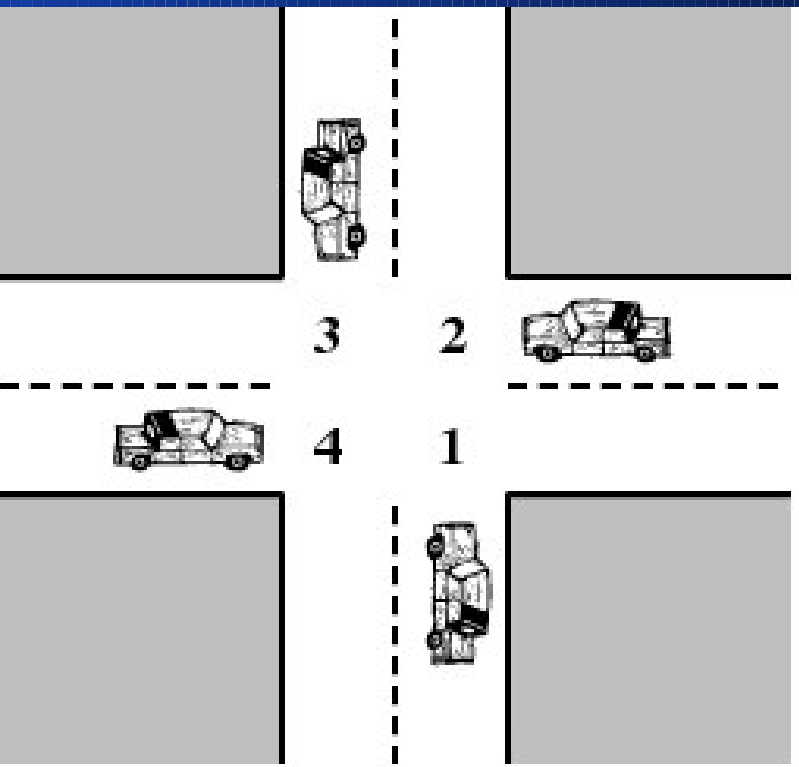
电子科技大学
University of Electronic Science and Technology of China

内容

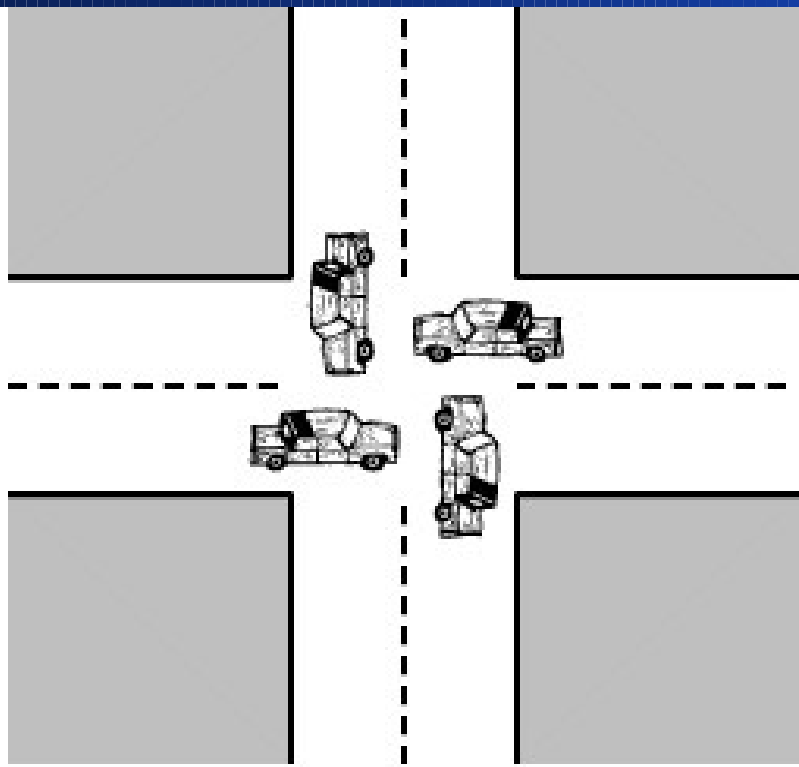
- 产生死锁与饥饿的原因
- 解决死锁的方法
- 死锁／同步的经典问题：哲学家进餐问题

Deadlock

- Permanent blocking of a set of processes that either compete for system resources or communicate with each other.
- No efficient solution.
- Involve conflicting needs for resources by two or more processes.



(a) Deadlock possible



(b) Deadlock

Figure 6.1 Illustration of Deadlock

Eg. Process P and Q compete two resources, Their general forms are:

Process P	Process Q
…	…
Get A	Get B
…	…
Get B	Get A
…	…
Release A	Release B
…	…
Release B	Release A
…	…

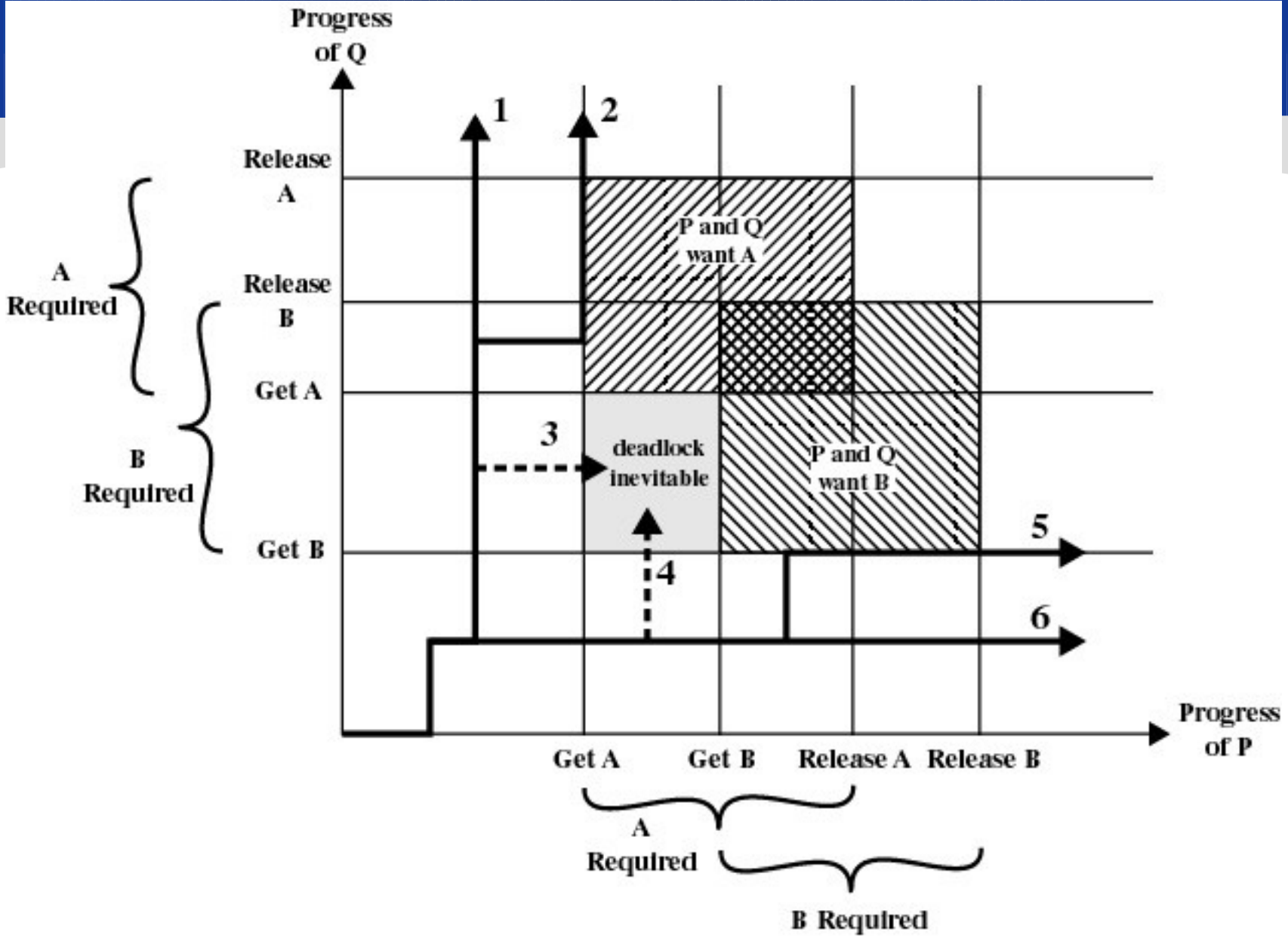


Figure 6.2 Example of Deadlock [BACO98]

死锁与进程的推进顺序有关。若修改 P 的代码，则不会产生死锁

```
Process  P
 …
Get A
 …
 Release A
 …
 Get B
…
Release B
…
```

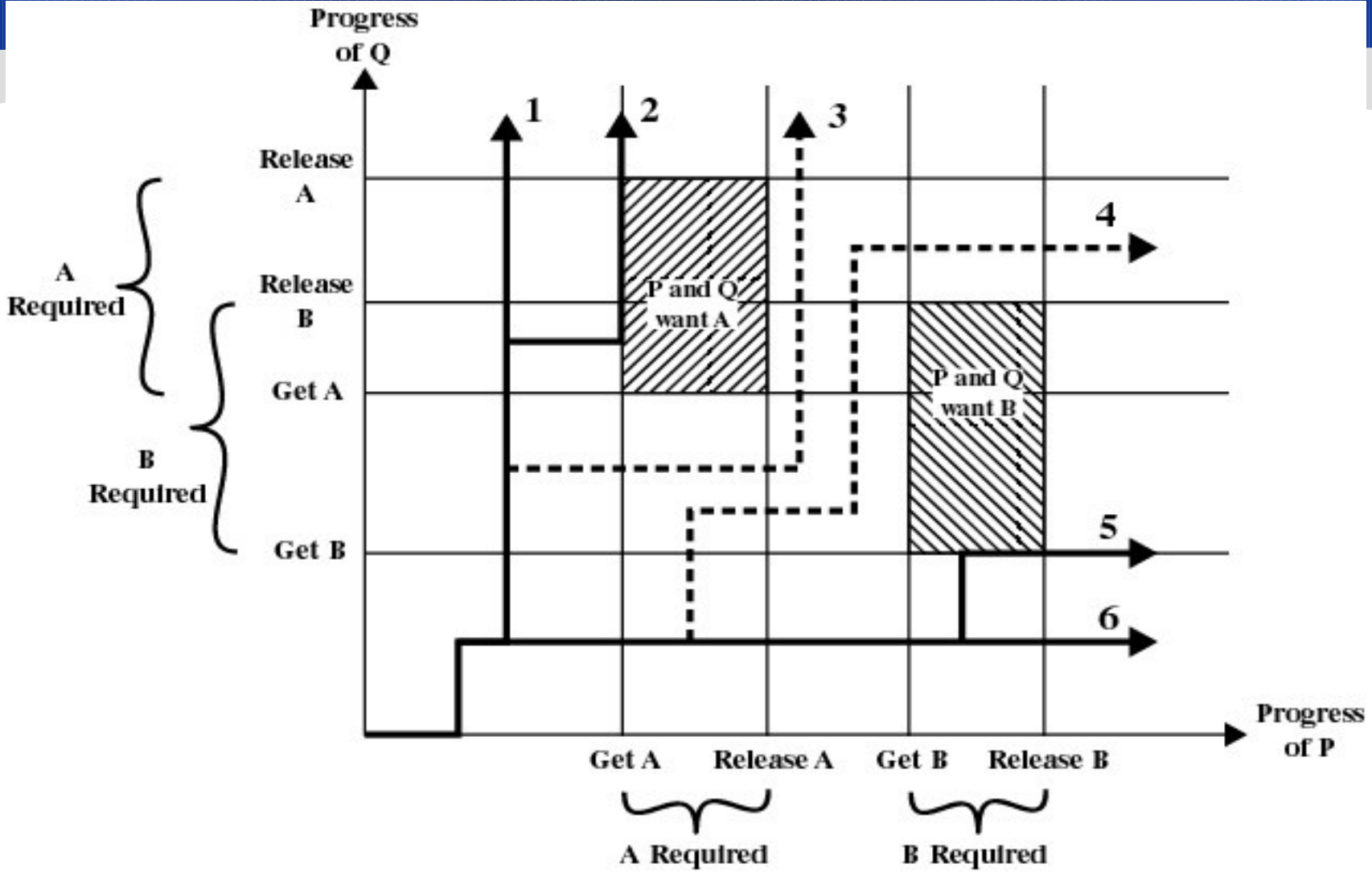


Figure 6.3 Example of No Deadlock [BACO98]

Reusable Resources (可重用资源)

- Used by one process at a time and not depleted (耗尽)by that use.
- Processes obtain resources that they later release for reuse by other processes.
- Processors, I/O channels, main and secondary memory, files, databases, and semaphores.
- Deadlock occurs if each process holds one resource and requests the other.

Example of Deadlock

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

Process Q

Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Another Example of Deadlock

- Space is available for allocation of 200K bytes, and the following sequence of events occur.

P1	P2
. . .	. . .
Request 80K bytes;	Request 70K bytes;
. . .	. . .
Request 60K bytes;	Request 80K bytes;

- Deadlock occurs if both processes progress to their second request.

Consumable Resources（可消耗资源）

- Created (produced) and destroyed (consumed) by a process.
- Interrupts, signals, messages, and information in I/O buffers.

Example of Deadlock

- Deadlock occurs if receive is blocking

```
P1
...
Receive(P2);
...
Send(P2, M1);
```

```
P2
...
Receive(P1);
...
Send(P1, M2);
```

- 此类死锁是由于设计失误造成的，很难发现，且潜伏期较长

Conditions for Deadlock

- **Mutual exclusion(互斥)**
  - **only one process may use a resource at a time.**
- **Hold-and-wait(保持并等待)**
  - **A process may hold allocated resources while awaiting assignment of other resources.**

Conditions for Deadlock

- **No preemption(不剥夺)**
  - No resource can be forcibly removed from a process holding it.
- **Circular wait(环路等待)**
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

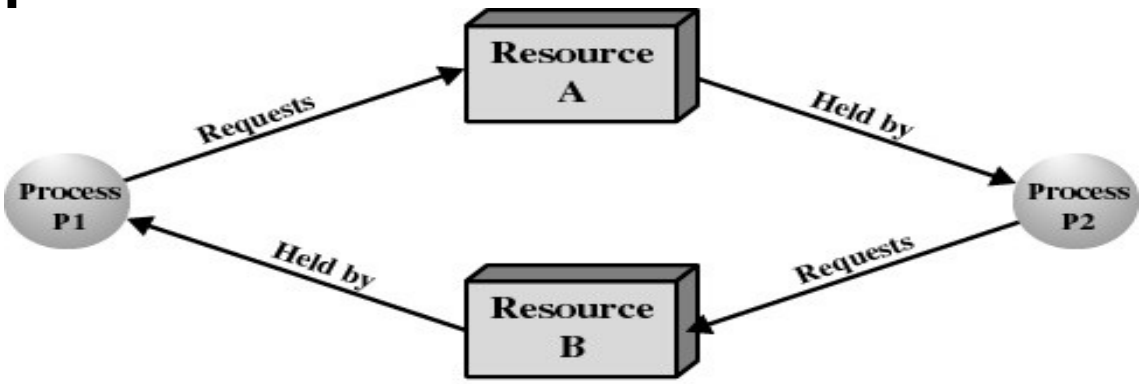


Figure 6.5 Circular Wait

Conditions for Deadlock

- 条件 Mutual exclusion 、 Hold-and-wait 、 No preemption 是死锁产生的必要条件，而非充分条件。

- 条件 Circular wait 是前 3 个条件产生的结果。

Deadlock Prevention（预防死锁）

间接方法，禁止前 3 个条件之一的发生：

1. *互斥* ： 是某些系统资源固有的属性，不能禁止
2. 禁止“*保持并等待*”条件：要求进程一次性地申请其所需的全部资源。若系统中没有足够的资源可分配给它，则进程阻塞。
3. 禁止“*不剥夺*”条件：①若一个进程占用了某些系统资源，又申请新的资源，则不能立即分配给它。必须让它首先释放出已占用资源，然后再重新申请；②若一个进程申请的资源被另一个进程占有， OS 可以剥夺低优先权进程的资源分配给高优先权的进程（要求此类可剥夺资源的状态易于保存和恢复，否则不能剥夺）

Deadlock Prevention（预防死锁）

- ***直接方法***，禁止条件 4 （环路等待）的发生

  即禁止“***环路等待***”条件：可以将系统的所有资源按类型不同进行线性排队，并赋予不同的序号。进程对某类资源的申请只能按照序号递增的方式进行。

显然，此方法是低效的，它将影响进程执行的速度，甚至阻碍资源的正常分配。

Deadlock Avoidance（避免死锁）

- ***预防死锁***通过实施较强的限制条件实现，降低了系统性能。
- ***避免死锁***的关键在于为进程分配资源之前，首先通过计算，判断此次分配是否会导致死锁，只有不会导致死锁的分配才可实行。
- A decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock.
- Requires knowledge of future process request.

Approaches to Deadlock Avoidance

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

Resource Allocation Denial

- Referred to as the banker's algorithm.
- State of the system is the current allocation of resources to process.
- **Safe state** is where there is at least one sequence that does not result in deadlock.
- **Unsafe state** is a state that is not safe.

Safe State

- 指系统能按某种顺序如 <P1， P2，…， Pn>（称 <P1， P2，…， Pn> 为安全序列），来为每个进程分配其所需资源，直至最大需求，使每个进程都可顺序完成，则称系统处于 *safe state.*

- 若系统不存在这样一个安全序列，则称系统处于 *unsafe state.*

Determination of a Safe State

Initial State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

(a) Initial state

Determination of a Safe State

P2 Runs to Completion

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

(b) P2 runs to completion

Determination of a Safe State

P1 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

(c) P1 runs to completion

Determination of a Safe State

P3 Runs to Completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	4

Available Vector

(d) P3 runs to completion

Determination of an *Unsafe State*

Initial State

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
1	1	2

Available Vector

(a) Initial state

Determination of an *Unsafe State*

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
0	1	1

Available Vector

(b) P1 requests one unit each of R1 and R3

Safe State *vs.* Unsafe State

- 并非所有不安全状态都是死锁状态
- 当系统进入不安全状态后，便可能进入死锁状态
- 只要系统处于安全状态，则可避免进入死锁状态。

Safe State *to* Unsafe State

例，假设系统中有 3 个进程 P1 、 P2 、 P3 ，共有 12 台磁带机。进程 P1 共需要 10 台， P2 、 P3 分别需要 4 台和 9 台。设 T0 时刻，进程 P1 、 P2 、 P3 已分别获得 5 台、 2 台和 2 台，尚有 3 台未分配，即图示：

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

Safe State *to* Unsafe State

- *T0 时刻系统是安全的*，因为存在一个*安全序列 <P2， P1， P3>*，即只要系统按此进程序列分配资源，每个进程都可顺利完成。

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

- 但是，如果不按照安全序列分配资源，则系统

Safe State *to* Unsafe State

例如， T0 时刻以后， P3 又申请 1 台磁带机。若系统将剩余 3 台中的 1 台分配给 P3，则系统进入不安全状态。

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

银行家算法

- 该算法可用于银行发放一笔贷款前，预测该笔贷款是否会引起银行资金周转问题。
- 这里，银行的资金就类似于计算机系统的资源，贷款业务类似于计算机的资源分配。银行家算法能预测一笔贷款业务对银行是否是安全的，该算法也能预测一次资源分配对计算机系统是否是安全的。
- 为实现银行家算法，系统中必须设置若干数据结构。

数据结构

1. 可利用资源向量 Available: 是一个具有 m 个元素的数组，其中的每一个元素代表一类可利用资源的数目，其初始值为系统中该类资源的最大可用数目。其值将随着该类资源的分配与回收而动态改变。 Available[j] = k, 表示系统中现有 R_j 类资源 k 个。

2. 最大需求矩阵 Max: 是一个 n * m 的矩阵，定义了系统中 n 个进程中的每一个进程对 m 类资源的最大需求。 Max(i,j) = k，表示进程 i 对 R_j 类资源的最大需求数目为 k 个。

数据结构

- 分配矩阵 Allocation: 是一个 n * m 的矩阵，定义了系统中每一类资源的数量。例如， Allocation(i,j) = k, 表示进程 i 当前已分得 R_j 类资源的数目为 k 个。

- 需求矩阵 Need: 是一个 n * m 的矩阵，用以表示每一个进程尚需的各类资源数。例如， Need[i,j] = k, 表示进程 i 还需要 R_j 类资源 k 个，方能完成其任务。

数据结构

上述三类矩阵存在下述关系：

Need [i,j] = Max[i,j] - Allocation [i,j]

设 $Request_i$ 是进程 P_i 的请求向量。 $Request_i[j]$ = k, 表示进程 P_i 需要 k 个 R_j 类资源。当进程 P_i 发出资源请求后，系统按下述步骤进行检查：

1. 如果， $Request_i \leq Need_i$, 则转向步骤 2；否则，出错

2. 如果， $Request_i \leq Available$, 则转向步骤 3；否则，表示尚无足够资源可供分配，进程 P_i 必须阻塞等待。

3. 系统试探性地将 P_i 申请的资源分配给它，并修改下列数据结构中的值：

Available := Available - $Request_i$;

Allocation := Allocation + $Request_i$;

$Need_i$:= $Need_i$ - $Request_i$;

4 系统利用安全性算法，检查此次资源分配以后，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，完成本次资源分配；否则，试探分配失效，让进程 P_i 阻塞等待。

安全性算法

（1）设置两个工作向量：

① 设置一个数组 Finish[n] 。当 Finish[i] ＝ True （ 0 ≤ i ≤ n， n 为系统中的进程数）时，表示进程 P_i 可以获得其所需的全部资源，而顺利执行完成。

② 设置一个临时向量 Work，表示系统可提供给进程继续运行的资源的集合。安全性算法刚开始执行时， Work := Available

安全性算法

（2）从进程集合中找到一个满足下列条件的进程：

Finish[i] = false； 并且 Need ≤ Work；

若找到满足该条件的进程，则执行步骤（3），否则执行步骤（4）；

（3） 当进程 P_i 获得资源后，将顺利执行直至完成，并释放其所拥有的全部资源，故应执行以下操作：

Work := Work + $Allocation_i$； 以及

Finish[i] := True；转向步骤（2）；

安全性算法

(4) 如果所有进程的 Finish[i] = True，则表示系统处于安全状态，否则系统处于不安全状态。

举例

T_0 时刻的资源分配情况

假定系统中有四个进程 P1, P2, P3, P4 和三种类型的资源 R1, R2, R3，每一种资源的数量分别为 9、3、6，T_0 时刻的资源分配情况如下表所示：

资源 / 进程	Max			Allocation			Need			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	2	2	2	0	1	1
P2	6	1	3	6	1	2	0	0	1			
P3	3	1	4	2	1	1	1	0	3			
P4	4	2	2	0	0	2	4	2	0			

T_0 时刻的安全性

从 T_0 时刻的安全性分析可知， T_0 时刻存在着一个安全序列 <P2， P1， P4， P3>，故， T_0 时刻系统是安全的。

- 假设 T_0 时刻，进程 P1 申请资源，其请求向量为 $Request_1(0,0,1)$, 系统按银行家算法进行检查：

  $Request_1\ (0,0,1) \leq Need_1(2,2,2)$，且

  $Request_1\ (0,0,1) \leq Available\ (0,1,1)$

- 故，系统试探性地为 P1 分配资源，并修改

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

利用安全性算法检查此时系统是否安全：

- 此时，系统的可用资源向量为 Available （ 0,1,0 ），比较各进程的需求向量 Need，系统不能满足任何进程的资源请求，系统进入不安全状态。
- 所以， P1 请求的资源不能分配，只能让进程 P1 阻塞等待。

- 假设 T_0 时刻，进程 P4 申请资源，其请求向量为 $Request_4(1,2,0)$, 系统按银行家算法进行检查：

  $Request_4\ (1,2,0) \leq Need_4(4,2,0)$，且

  $Request_4\ (1,2,0) > Available\ (0,1,1)$

- P4 的请求向量超过系统的可用资源向量，故 P4 的请求不能满足。进程 P4 阻塞等待

- 如果 T_0 时刻，进程 P4 申请资源，其请求向量为 $Request_4(0,1,0)$, 系统是否能将资源分配给它？

Deadlock Avoidance

- Maximum resource requirement must be stated in advance(预先必须申明每个进程需要的资源总量)
- Processes under consideration must be independent; no synchronization requirements (进程之间相互独立，其执行顺序取决于系统安全，而非进程间的同步要求)
- There must be a fixed number of resources to allocate(系统必须提供固定数量的资源供分配)
- No process may exit while holding resources(若进程占有资源，则不能让其退出系统)

Deadlock Detection(检测死锁)

- 检测死锁不同于预防死锁，不限制资源访问方式和资源申请。
- OS 周期性地执行死锁检测例程，检测系统中是否出现 *“环路等待”*。

Strategies once Deadlock Detected

- Abort all deadlocked processes.
- Back up each deadlocked process to some previously defined checkpoint, and restart all process.
  - original deadlock may occur.
- Successively abort deadlocked processes until deadlock no longer exists.
- Successively preempt resources until deadlock no longer exists.

Selection Criteria Deadlocked Processes

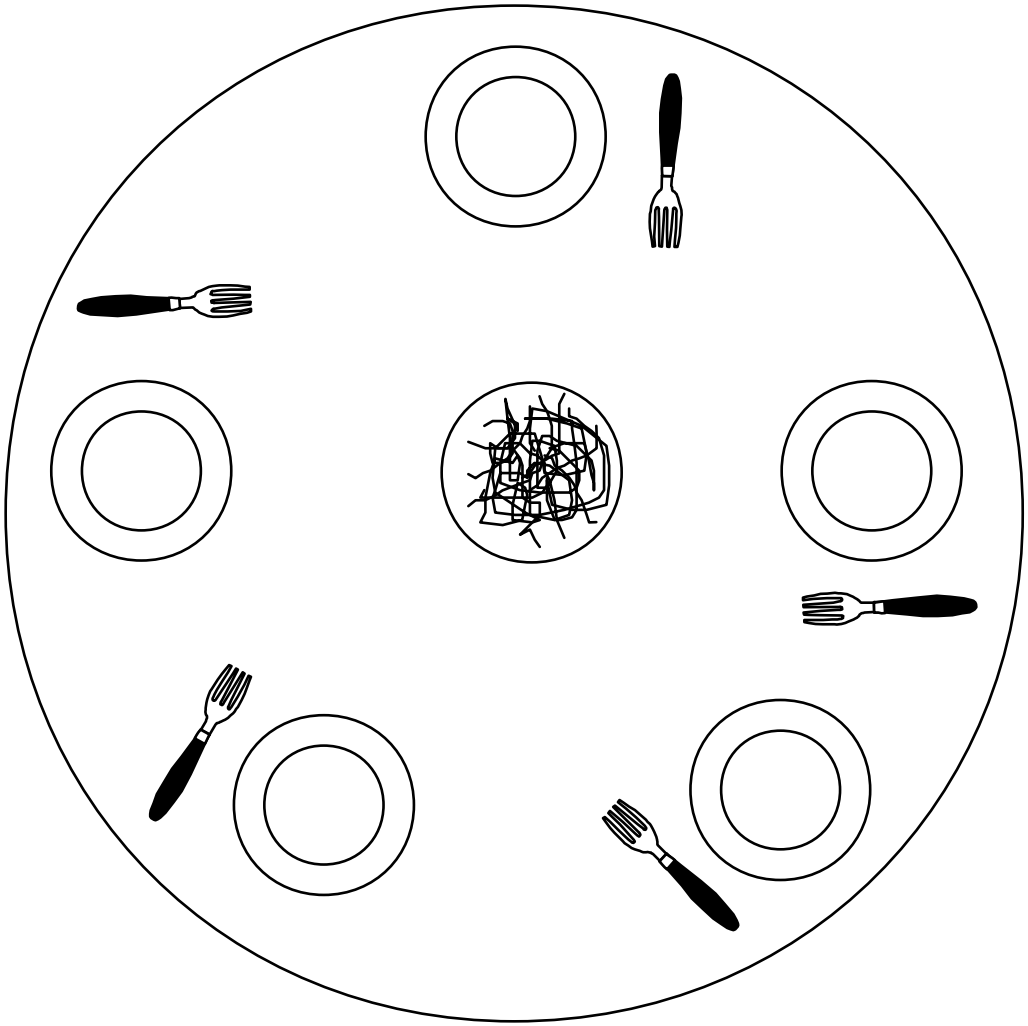
- Least amount of processor time consumed so far.
- Least number of lines of output produced so far.
- Most estimated time remaining.
- Least total resources allocated so far.
- Lowest priority.

Dining Philosophers Problem

- ***描述***：有 5 个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们共用一张圆桌，分别坐在周围的五张椅子上。圆桌中间放有一大碗面条，每个哲学家分别有 1 个盘子和 1 支叉子。如果哲学家想吃面条，则必须拿到靠其最近的左右两支叉子。进餐完毕，放下叉子继续思考。

- ***要求***：设计一个合理的算法，使全部哲学家都能进餐（非同时）。算法必须避免死锁和饥饿，哲学家互斥共享叉子。

Dining Philosophers Problem



Dining Philosophers Problem - by Semaphores

```
Program diningphilosophers;
Var  fork:array [0..4] of semaphore (:= 1);
     i : integer;
procedure philosopher (i : integer);
begin
  repeat
    think;                                  /*  哲学家正在思考     */
      wait(fork[i]);                        /*  取其左边的筷子     */
      wait(fork[(i + 1) mod 5]);  /*  取其右边的筷子     */
      eat;                                  /*  吃面条     */
      signal(fork[(i + 1) mod 5]);                    /*  放回右边的筷子     */
      signal(fork[i]);    /*  放回左边的筷子     */
forever
end;
begin
  parbegin
    philosopher(0); philosopher(1); philosopher(2); philosopher(3); philosopher(4);
  parend
```

Dining Philosophers Problem

- 可能产生死锁！
- 可行的解决方案：只允许 4 个哲学家同时进餐厅用餐，则至少有一个哲学家可以拿到两支叉子进餐，完毕，放下叉子，其他哲学家就可进餐。不会出现死锁和饥饿

```
Program diningphilosophers;
Var  fork:array [0..4] of semaphore (:= 1);
     room : semaphore (:= 4);
i : integer;
procedure philosopher (i : integer);
begin
  repeat
    think;                                  /*  哲学家正在思考    */
    wait(room);    /*  第 5 位哲学家将被阻塞在 room 信号量队列  */
    wait(fork[i]);                          /*  取其左边的筷子    */
    wait(fork[(i + 1) mod 5]);   /*  取其右边的筷子    */
     eat;                                   /*  吃面条    */
    signal(fork[(i + 1) mod 5]);                    /*  放回右边的筷子    */
    signal(fork[i]);   /*  放回左边的筷子    */
    signal(room); /*  唤醒阻塞在 room 信号量队列中的哲学家  */
   forever
end;
begin
  parbegin
    philosopher(0); philosopher(1); philosopher(2); philosopher(3); philosopher(4);
  parend
end.
```

Thank You !

UESTC

电子科技大学
University of Electronic Science and Technology of China