南京大学
NANJING UNIVERSITY

计算机操作系统
第六章 并发程序设计
6.6 死锁

南京大学软件学院

南京大学
NANJING UNIVERSITY

6.6 死锁

6.6.1 死锁产生

6.6.2 死锁防止

6.6.3 死锁避免

6.6.4 死锁检测和解除

6.6.1 死锁产生

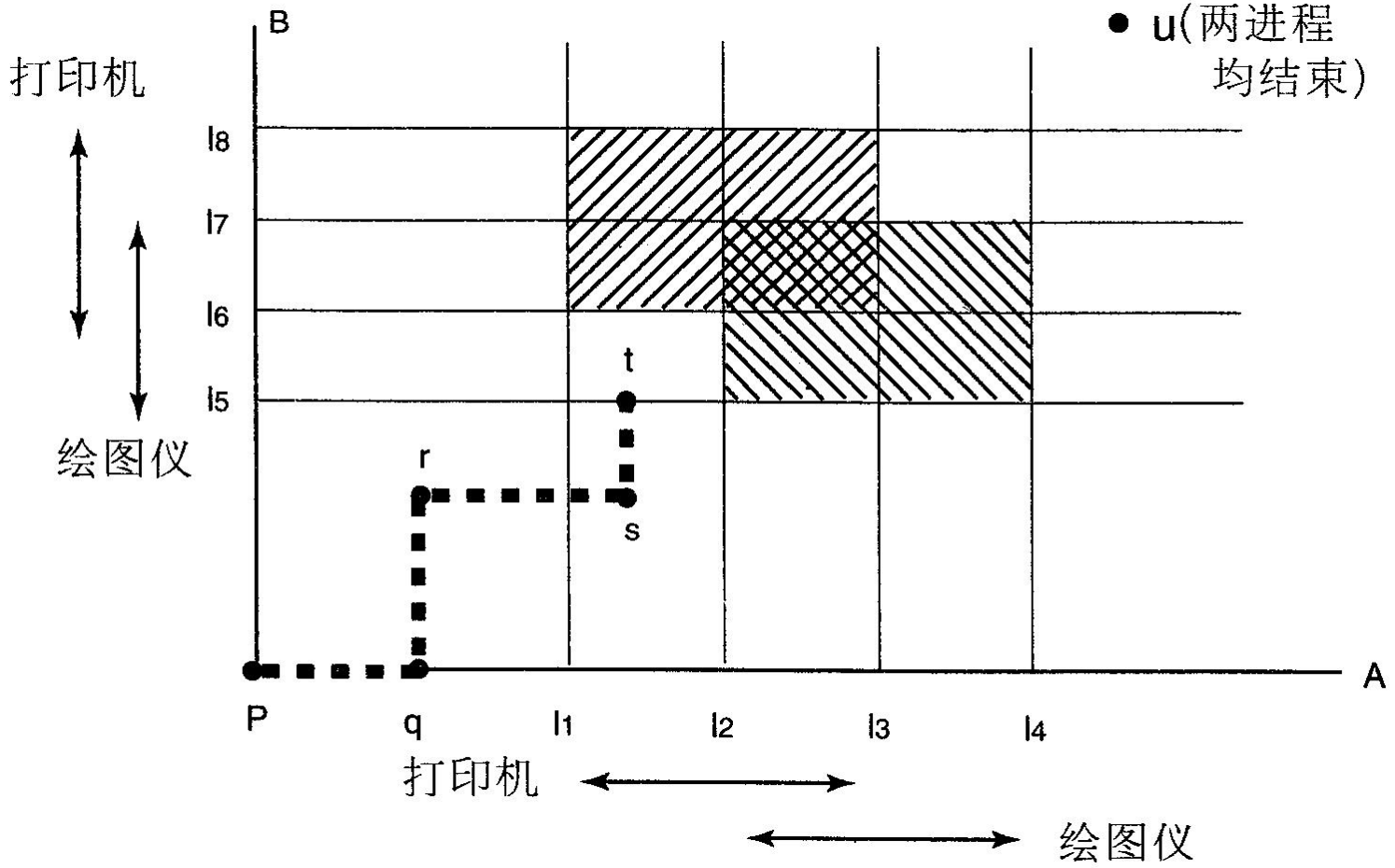
若干死锁的例子(1)

例1 进程推进顺序不当产生死锁

设系统有打印机、绘图仪各一台，被进程Q1和Q2共享。两个进程并发执行，按下列次序请求和释放资源：

进程Q1	进程Q2
请求绘图仪	请求打印机
请求打印机	请求绘图仪
使用绘图仪和打印机	使用绘图仪和打印机
释放绘图仪	释放绘图仪
释放打印机	释放打印机

进程资源轨迹图



若干死锁的例子(2)

例2 PV操作使用不当产生死锁

进程P1	进程P2
……	……
P(s1);	P(s2);
P(s2);	P(s1);
使用r1和r2;	使用r1和r2
V(s1);	V(s2);
V(s2);	V(s1);

说明：一个系统中有r1和r2两种资源各1个，分别用s1和s2信号量表示互斥，信号量处置设置为1

南京大学 NANJING UNIVERSITY

若干死锁的例子(3)

例3 资源分配不当引起死锁

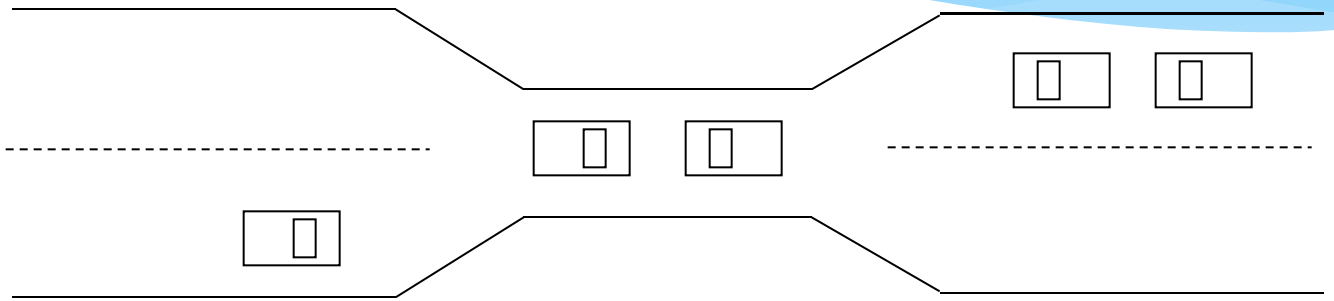
若系统中有m个资源被n个进程共享，每个进程都要求K个资源，而m < n·K时，即资源数小于进程所要求的总数时，如果分配不得当就可能引起死锁

若干死锁的例子(4)

例4 对临时性资源使用不加限制引起死锁

* 进程通信使用的信件是一种临时性资源，如果对信件的发送和接收不加限制，可能引起死锁。
* 进程P1等待进程P3的信件S3来到后再向进程P2发送信件S1；P2又要等待P1的信件S1来到后再向P3发送信件S2；而P3也要等待P2的信件S2来到后才能发出信件S3。这种情况下形成了循环等待，产生死锁。

独木桥(例)



* 只能在一个方向行车
* 可能发生死锁
* 也可能发生饥饿

南京大学
NANJING UNIVERSITY

死锁定义

* 操作系统中的死锁指：如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生死锁。
* 例如，n个进程P1，P2，...，Pn，Pi因为申请不到资源Rj而处于等待状态，而Rj又被Pi+1占有，Pn欲申请的资源被P1占有，此时这n个进程的等待状态永远不能结束，则说这n个进程处于死锁状态。

产生死锁因素

* 不仅与系统拥有的资源数量有关，而且与资源分配策略，进程对资源的使用要求以及并发进程的推进顺序有关

6.6.2 死锁防止(1)

系统形成死锁的四个必要条件

* 互斥条件(mutual exclusion)：系统中存在临界资源，进程应互斥地使用这些资源
* 占有和等待条件(hold and wait)：进程请求资源得不到满足而等待时，不释放已占有的资源
* 不剥夺条件(no preemption)：已被占用的资源只能由属主释放，不允许被其它进程剥夺
* 循环等待条件(circular wait)：存在循环等待链，其中，每个进程都在链中等待下一个进程所持有的资源，造成这组进程永远等待

死锁防止(2)

破坏第一个条件

* 使资源可同时访问而不是互斥使用，
* 可再入程序、只读数据文件、时钟、磁盘等软硬件资源均可用这种办法管理，但有许多资源如可写文件、磁带机等由于特殊性质决定只能互斥占有，而不能被同时访问，所以这种做法许多场合行不通。//有些资源具有天生的互斥性

破坏第二个条件

* 静态分配，
* 进程在执行中不再申请资源，就不会出现占有某些资源再等待另一些资源的情况。
* 实现简单，被许多操作系统采用，但会严重地降低资源利用率，因为在每个进程占有的资源中，有些资源在运行后期使用，甚至有些资源在例外情况下才被使用，可能造成进程占有一些几乎不用的资源，而使其他想用这些资源的进程产生等待

死锁防止(2)

破坏第一个条件

* 使资源可同时访问而不是互斥使用，

破坏第二个条件

* 静态分配，

破坏第三个条件

* 采用剥夺式调度方法，
* 当进程在申请资源未获准许的情况下,如主动释放资源(一种剥夺式),然后才去等待。

破坏第四个条件

* 上述死锁防止办法造成资源利用率和吞吐率低。介绍两种比较实用的死锁防止方法。

死锁的防止(3)

采用层次分配策略(破坏条件2和4)

* 资源被分成多个层次
* 当进程得到某一层的一个资源后，它只能再申请较高层次的资源
* 当进程要释放某层的一个资源时，必须先释放占有的较高层次的资源
* 当进程得到某一层的一个资源后，它想申请该层的另一个资源时，必须先释放该层中的已占资源

死锁防止(4)

层次策略的变种按序分配策略

* 把系统的所有资源排一个顺序，例如，系统若共有n个进程，共有m个资源，用ri表示第i个资源，于是这m个资源是：

  r1,r2……,rm

* 规定如果进程不得在占用资源ri(1≤i≤m)后再申请rj(j<i)。不难证明，按这种策略分配资源时系统不会发生死锁。

南京大学
NANJING UNIVERSITY

6.6.3 死锁避免

* 银行家算法
  * 银行家拥有一笔周转资金
  * 客户要求分期贷款，如果客户能够得到各期贷款，就一定能够归还贷款，否则就一定不能归还贷款
  * 银行家应谨慎的贷款，防止出现坏帐
* 用银行家算法避免死锁
  * 操作系统(银行家)
  * 操作系统管理的资源(周转资金)
  * 进程(要求贷款的客户)

[Dijkstra 1965a] E. W. Dijkstra. "Cooperating Sequential Processes". Technical Report, Technological University, Eindhoven, the Netherlands (1965).

南京大学
NANJING UNIVERSITY

银行家算法的数据结构(1)

一个系统有n个进程和m种不同类型的资源，定义包含以下向量和矩阵的数据结构：

- 系统每类资源总数--该m个元素的向量为系统中每类资源的数量

  $Resource=(R_1,R_2,\ldots,R_m)$

- 每类资源未分配数量--该m个元素的向量为系统中每类资源尚可供分配的数量

  $Available=(V_1,V_2,\ldots,V_m)$

南京大学
NANJING UNIVERSITY

银行家算法的数据结构(2)

* 最大需求矩阵--每个进程对每类资源的最大需求量,C_{ij}表示进程P_i需R_j类资源最大数

$$Claim = \begin{pmatrix} C11 & C12 & \cdots & C1m \\ C21 & C22 & \cdots & C2m \\ & & & \\ Cn1 & Cn1 & \cdots & Cnm \end{pmatrix}$$

银行家算法的数据结构(3)

分配矩阵——表示进程当前已分得的资源数，A_{ij}表示进程P_i已分到R_j类资源的个数

Allocation=

A11	A12	…	A1m
A21	A21	…	A21
An1	An1	…	Anm

银行家算法中下列关系式确保成立

- $R_i = V_i + \sum A_{ki}$ 对i=1,..,m,k=1,..,n;

  表示所有资源要么已被分配、要么尚可分配

- $C_{ki} \leqslant R_i$ 对i=1,..,m, k=1,..,n;

  表示进程申请资源数不能超过系统拥有的资源总数

- $A_{ki} \leqslant C_{ki}$ 对i=1,..,m,k=1,..,n;

  表示进程申请任何类资源数不能超过声明的最大资源需求数

一种死锁避免策略

系统中若要启动一个新进程工作,其对资源Ri的需求仅当满足下列不等式:

$$R_i \geqslant C_{(n+1)i} + \sum C_{ki} \quad \text{对} i=1,..,m,k=1,..,n;$$

即应满足当前系统中所有进程对资源R_i的最大资源需求数加上启动的新进程的最大资源需求数不超过系统拥有的最大数。

南京大学
NANJING UNIVERSITY

系统安全性定义

系统安全性定义：在时刻T0系统是安全的，仅当存在一个进程序列$P_1,..,P_n$，对进程P_k满足公式：

$$Cki-Aki \leqslant Vi+ \sum Aji$$

$$k=1,\ldots,n;\ i=1,\ldots,m;$$

实例说明系统所处的安全或不安全状态(1)

* 如果系统中共有五个进程和A、B、C三类资源
* A类资源共有10个,B类资源共有5个,C类资源共有7个
* 在时刻T_0,系统目前资源分配情况如下：

实例说明系统所处的安全或不安全状态(2)

process	Allocation			Claim			Available		
	A	B	C	A	B	C	A	B	C
* P_0	0	1	0	7	5	3	3	3	2
* P_1	2	0	0	3	2	2			
* P_2	3	0	2	9	0	2			
* P_3	2	1	1	2	2	2			
* P_4	0	0	2	4	3	3			

资源总数 [10 5 7]

实例说明系统所处的安全或不安全状态(3)

每个进程目前还需资源为Cki-Aki

process	C_{ki}-A_{ki}		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Available		
A	B	C
3	3	2

实例说明系统所处的安全或不安全状态(4)

可以断言目前系统处于安全状态，因为，序列{P_1,P_3,P_4,P_2,P_0}能满足安全性条件

实例说明系统所处的安全或不安全状态(5)

资源 / 进程	currentavil			$C_{ki}-A_{ki}$			allocation			currentavil+allocation			possible
	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

此可行序列不唯一

思考问题：

银行家算法在探测安全序列是否可以只计算第一步，省略后续步骤？

举例：假设初始资源是A/B两种资源，初始资源向量是(90, 90)

进程	Claim		Allocation		Cki-Aki		Available	
	A	B	A	B	A	B	A	B
P1	2	2	1	1	1	1	29	29
P2	80	80	30	30	50	50		
P3	80	80	30	30	50	50		

演算：剩余资源先满足P1，当时当做完P1并回收资源后是(31,31)，后续无法推进

实例说明系统所处的安全或不安全状态(6)

此前剩余向量为(3，3，2)，
如果满足P1进程请求request1=(1,0,2)，
剩余资源向量为(2, 3, 0)

进程P1申请资源request1=(1,0,2)，检查request1≤Available、比较(1,0,2) ≤(3,3,2)，结果满足条件，试分配，得到新状态：

process	allocation			$C_{ki}-A_{ki}$			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			

实例说明系统所处的安全或不安全状态(7)

* 判定新状态是否安全?可执行安全性测试算法，找到一个进程序列{P_1,P_3,P_4,P_0,P_2}能满足安全性条件,可正式把资源分配给进程P1;

资源 / 进程	currentavil			$C_{ki}-A_{ki}$			allocation			currentavil+allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_1	2	3	0	0	2	0	3	0	2	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_0	7	4	5	7	4	3	0	1	0	7	5	5	TRUE
P_2	7	5	5	6	0	0	3	0	2	10	5	7	TRUE

实例说明系统所处的安全或不安全状态(8)

此前剩余向量为(2，3，0)，
如果满足P0进程请求(0，2，0)，
剩余资源向量为(2, 1, 0)

* 系统若处在下面状态中，进程P4请求资源(3,3,0)，由于可用资源不足，申请被系统拒绝；此时，系统能满足进程P0的资源请求(0,2,0)；但可看出系统已处于不安全状态了。

资源 / 进程	allocation	Cki-Aki	available
	A B C	A B C	A B C
P0	0 3 0	7 2 3	2 1 0
P1	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	0 0 2	4 3 1	

银行家算法的基本思想(1)

* 系统中的所有进程进入进程集合，
* 在安全状态下系统收到进程的资源请求后，先把资源试探性分配给它。
* 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较，在进程集合中找到剩余资源能满足最大需求量的进程，从而，保证这个进程运行完毕并归还全部资源。
* 把这个进程从集合中去掉，系统的剩余资源更多了，反复执行上述步骤。//进程退出系统，资源回收
* 最后，检查进程集合，若为空表明本次申请可行，系统处于安全状态，可实施本次分配；否则，有进程执行不完，系统处于不安全状态，本次资源分配暂不实施，让申请进程等待。

银行家算法的程序及简短说明(1)

```
typedef struct state {//全局数据结构
        int resource[m];
        int available[m];
        int claim[n][m];
        int allocation[n][m];
};
```

南京大学
NANJING UNIVERSITY

银行家算法的程序及简短说明(2)

```
void resource_allocation( ) { //资源分配算法
        if(allocation[i,*]+request[*]>claim[i,*])
          {error};          //申请量超过最大需求值
        else {
          if(request[*]>available[*])
            {suspend process.};
          else {            //尝试分配，define newstate by:
          allocation[i,*]=allocation[i,*]+request[*];
          available[*]=available[*]-request[*];
        }
      }
if(safe(newstate))
        {carry out allocation};
else {
        {restore original state};
        {suspend process};
                  }
    }
```

银行家算法的程序及简短说明(3)

```
bool safe(state s) {   //安全性测试算法
        int currentavail[m];
        set <process> rest;
        currentavail[*]=available[*];
        rest={all process};
        possible=true;
        while(possible){  //rest中找一个Pk，满足以下条件
                claim[k,*]-allocation[k,*]<=currentavail[*];
                if(found){
        currentavail[*]=currentavail[*]+allocation[k,*];
                  rest=rest–{Pk};
                }
                else
                  possible=false;
         }
        return(rest=null);
}
```

6.6.4 死锁检测和解除

资源分配图和死锁定理

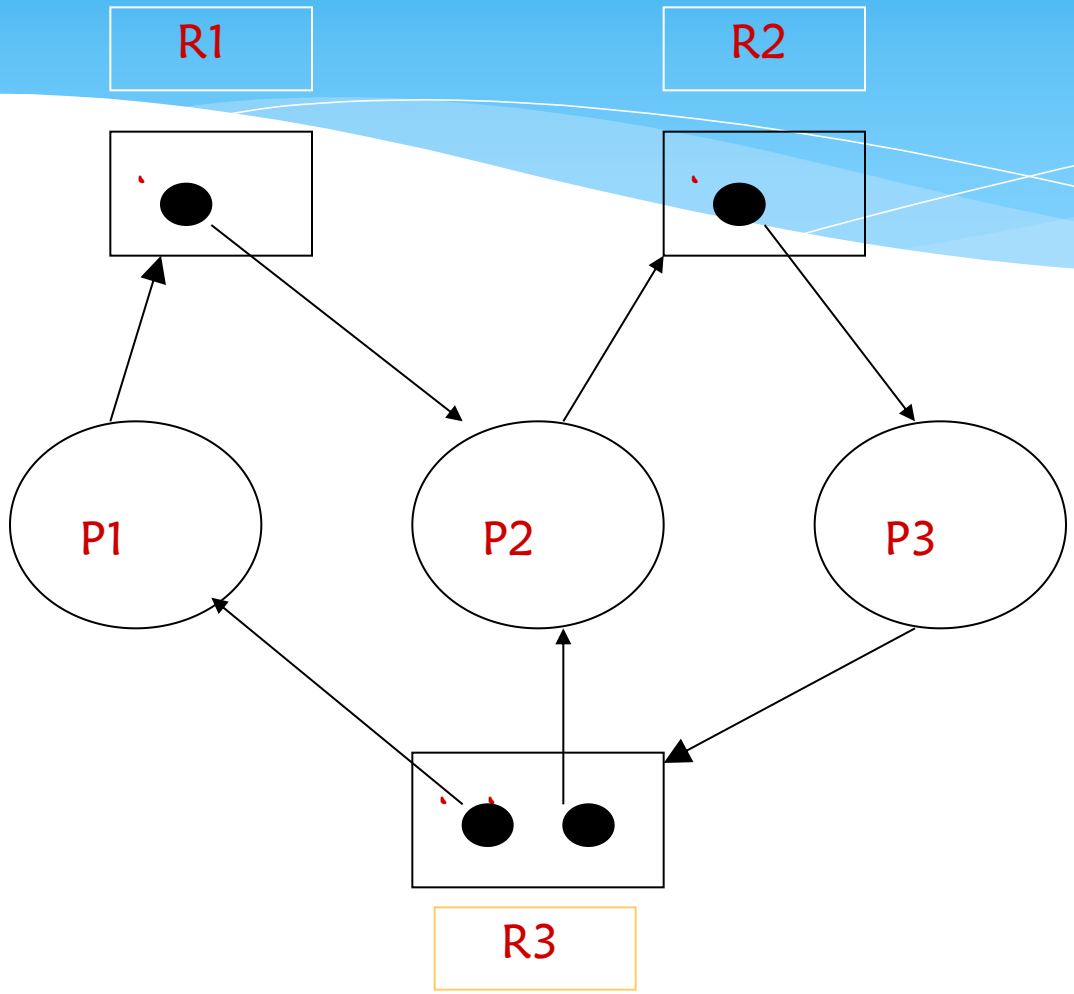
* 解决死锁问题的一条途径是死锁检测和解除，这种方法对资源的分配不加任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检测”程序，判断系统内是否已出现死锁，如果检测到系统已发生了死锁，再采取措施解除它。

南京大学 NANJING UNIVERSITY

进程-资源分配图

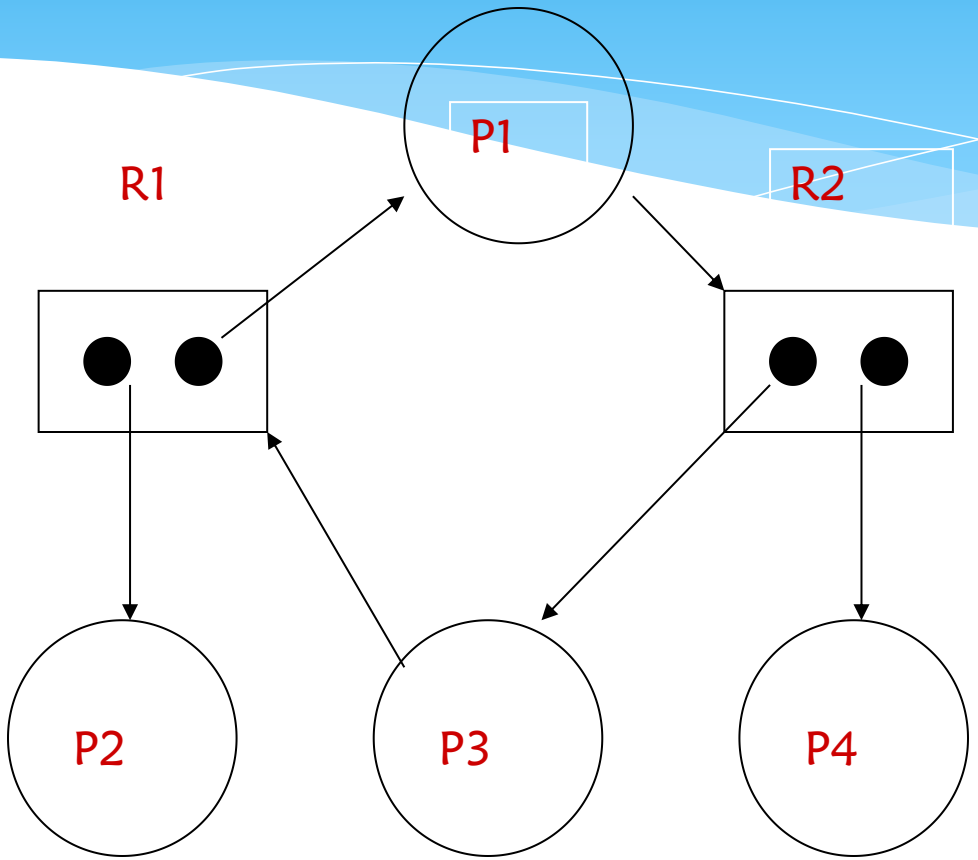
* 约定Pi→Rj为请求边，表示进程Pi申请资源类Rj中的一个资源得不到满足而处于等待Rj类资源的状态，该有向边从进程开始指到方框的边缘，表示进程Pi申请Rj类中的一个资源。
* Rj→Pi为分配边，表示Rj类中的一个资源已被进程Pi占用，由于已把一个具体的资源分给了进程Pi，故该有向边从方框内的某个黑圆点出发指向进程。

资源分配图的一个例子



死锁的例子

资源分配图的另一个例子



一个有环路而无死锁的例子

简化进程-资源分配图检测系统是否处于死锁状态(1)

(1)如果进程-资源分配图中无环路，则此时系统没有发生死锁

(2)如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程

(3)如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件

简化进程-资源分配图检测系统是否处于死锁状态(2)

* 如果能在进程-资源分配图中消去此进程的所有请求边和分配边，成为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的
* 系统为死锁状态的充分条件是：当且仅当该状态的进程-资源分配图是不可完全简化的。该充分条件称为死锁定理

孤立结点含义?

死锁的检测和解除方法(1)

(1) 借助于死锁的安全性测试算法来实现。死锁检测算法与死锁避免算法是类似的，不同在于前者考虑了检查每个进程还需要的所有资源能否满足要求；而后者则仅要根据进程的当前申请资源量来判断系统是否进入了不安全状态

死锁的检测和解除方法(2)

一种具体的死锁检测方法，检测算法步骤如下：

* 1)currentavail=available；
* 2)如果allocation[k,*]!=0，令finish[k]=false;否则finish[k]=true；
* 3)寻找一个k，它应满足条件：
  (finish[k]==false)&&(request[k,*]<=currentavail[*]);若找不到这样的k，则转向5)；
* 4)修改currentavail[*]=Currentavail[*]+allocation[k,*];
  finish[k]=true;然后转向3)；
* 5)如果存在k(1≤k≤n),finish[k]=false,则系统处于死锁状态，并且finish[k]=false的Pk为处于死锁的进程。

死锁的解除(1)

* 结束所有进程的执行，重新启动操作系统。方法简单，但以前工作全部作废，损失很大。
* 撤销陷于死锁的所有进程，解除死锁继续运行。
* 逐个撤销陷于死锁的进程，回收其资源重新分派，直至死锁解除。

死锁的解除(2)

* 剥夺陷于死锁的进程占用的资源，但并不撤销它，直至死锁解除。可仿照撤销陷于死锁进程的条件来选择剥夺资源的进程
* 根据系统保存的检查点，让所有进程回退，直到足以解除死锁，这种措施要求系统建立保存检查点、回退及重启机制。
* 当检测到死锁时，如果存在某些未卷入死锁的进程，而随着这些进程执行到结束，有可能释放足够的资源来解除死锁。

补充内容

Petri Nets for Automata Communication

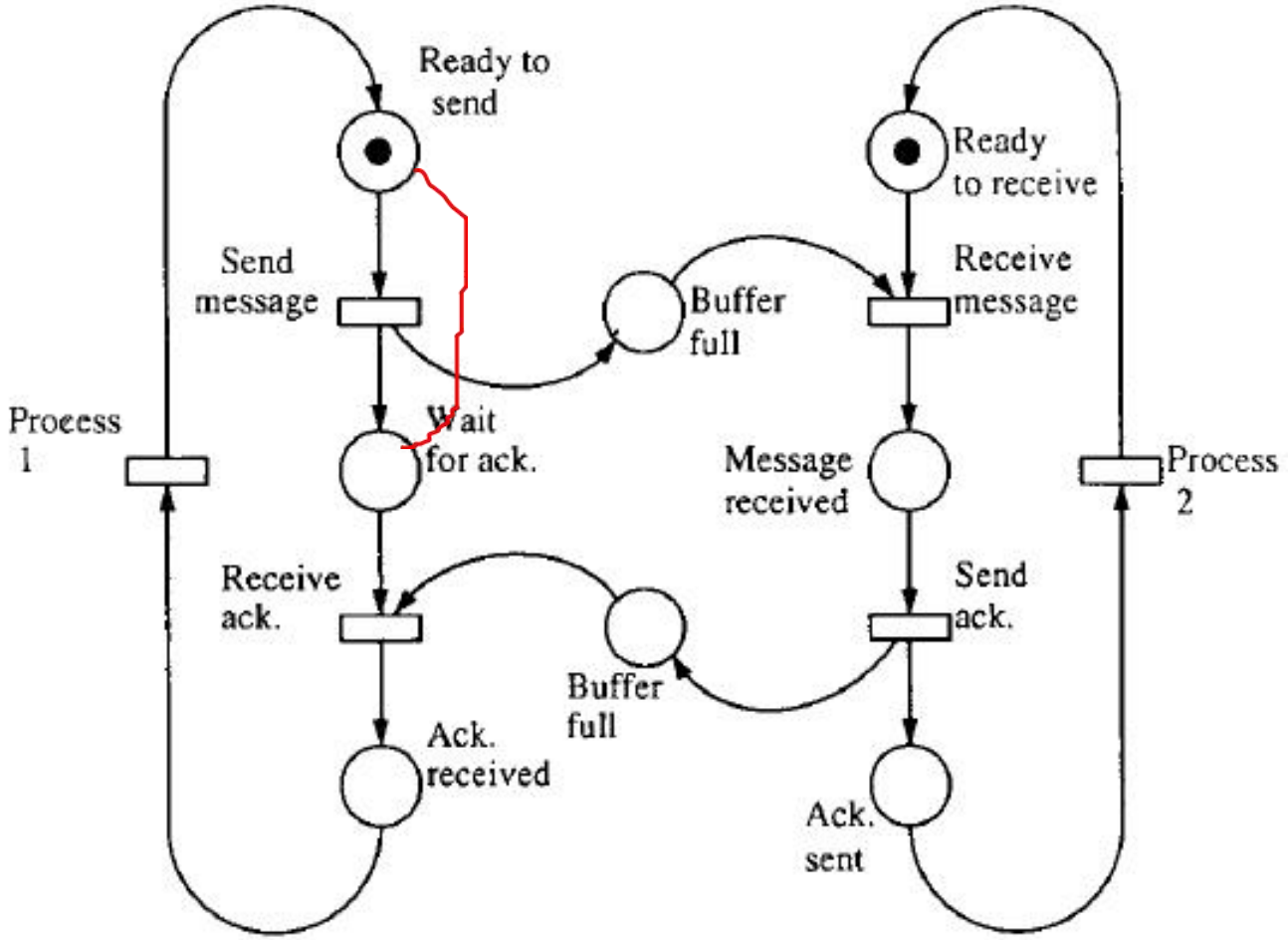


Fig. 9. A simplified model of a communication protocol.

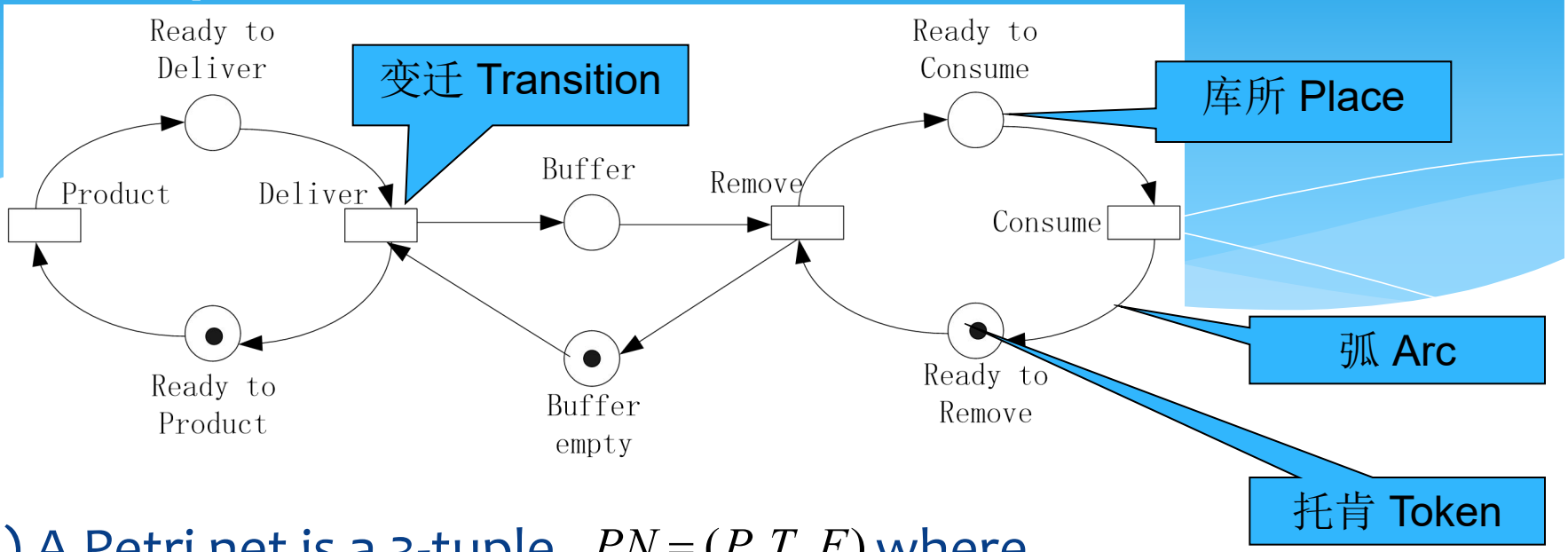
Carl Adam Petri
(1926-2010)

Petri网, 并发理论
2008年获IEEE计算机先驱奖

Brief History of Petri nets

* C. A. Petri: Kommunikation mit Automaten, 博士论文, Bonn Univ. 1962
* Use – A formal modeling tool for concurrent and distributed system
* Earlier research – GMD Funding at MIT
  * Commoner Theorem for Free Choice Petri Nets, 1971
  * Hack -- Decidability questions for Petri nets, 1976
* Academic annals
  * International Conference on Application and Theory of Petri Nets, since 1980 , published by Springer-LNCS

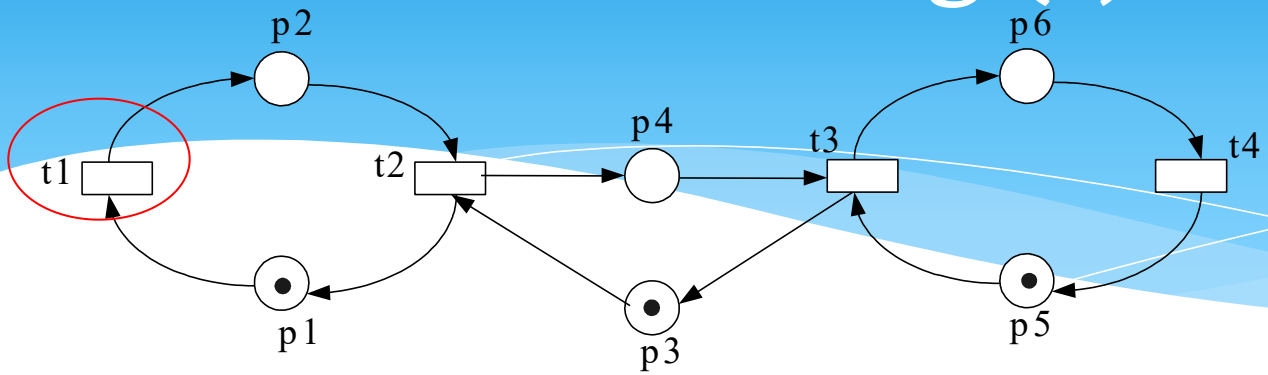
Example – Producer/Consumer System



* (1) A Petri net is a 3-tuple $PN = (P, T, F)$ where
  * P is a finite set of places,
  * T is a finite set of transitions, $P \cap T = \phi$
  * $F \subseteq P \times T \cup T \times P$ is a set of arcs (flow relation)
* (2)Preset $\bullet x = \{y \mid y \in P \cup T, (y, x) \in F\}$
* Postset $x\bullet = \{y \mid y \in P \cup T, (x, y) \in F\}$

Token Game

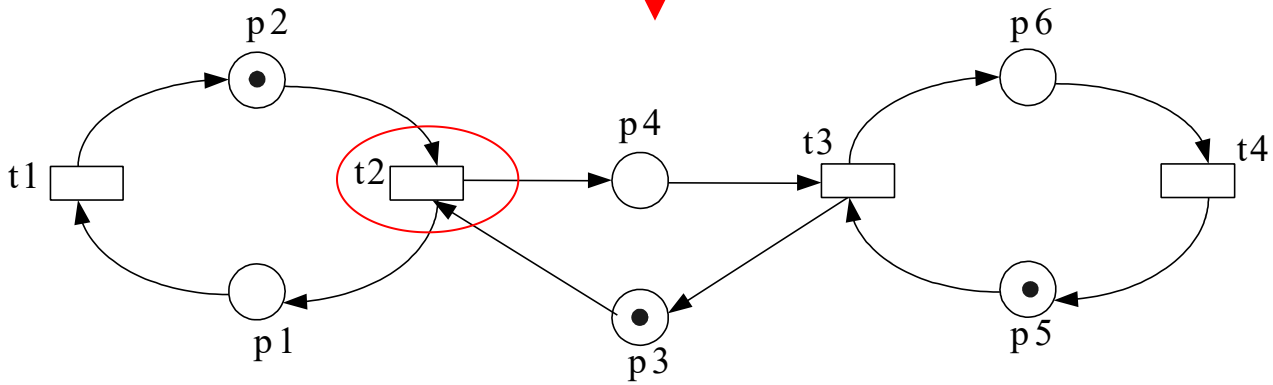
Running (1)



M_1=p1+p3+p5

[1 0 1 0 1 0]

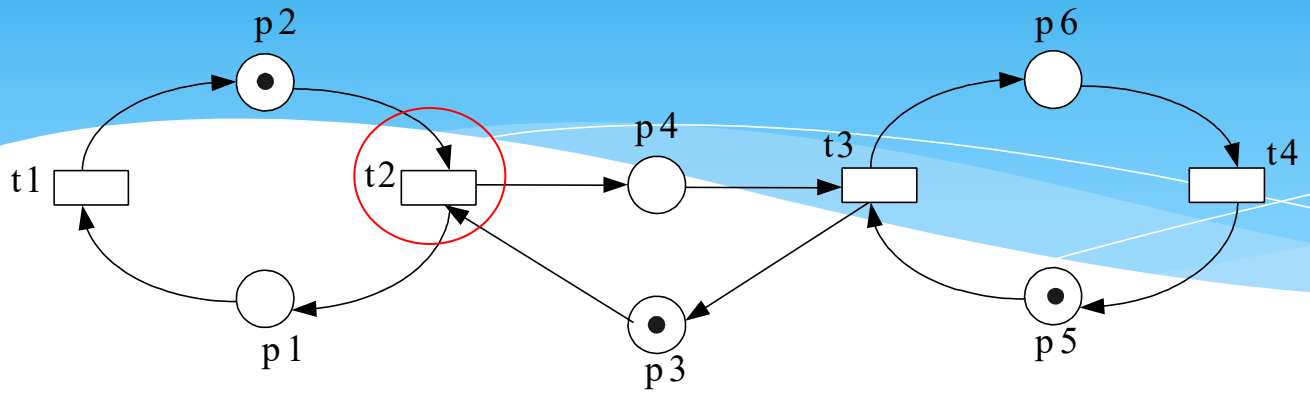
t1



M_2=p2+p3+p5

[0 1 1 0 1 0]

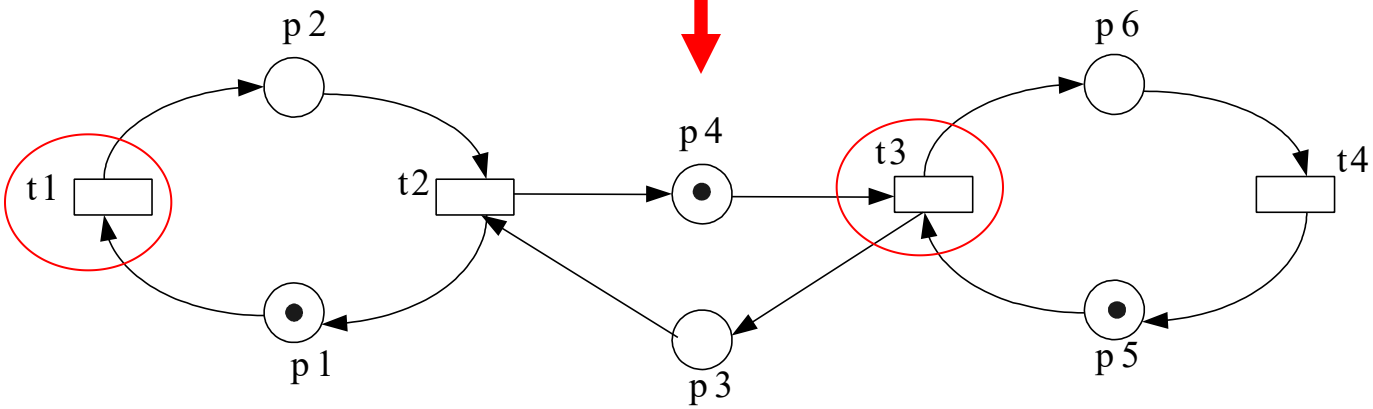
Running (2)



M_2=p2+p3+p5

[0 1 1 0 1 0]

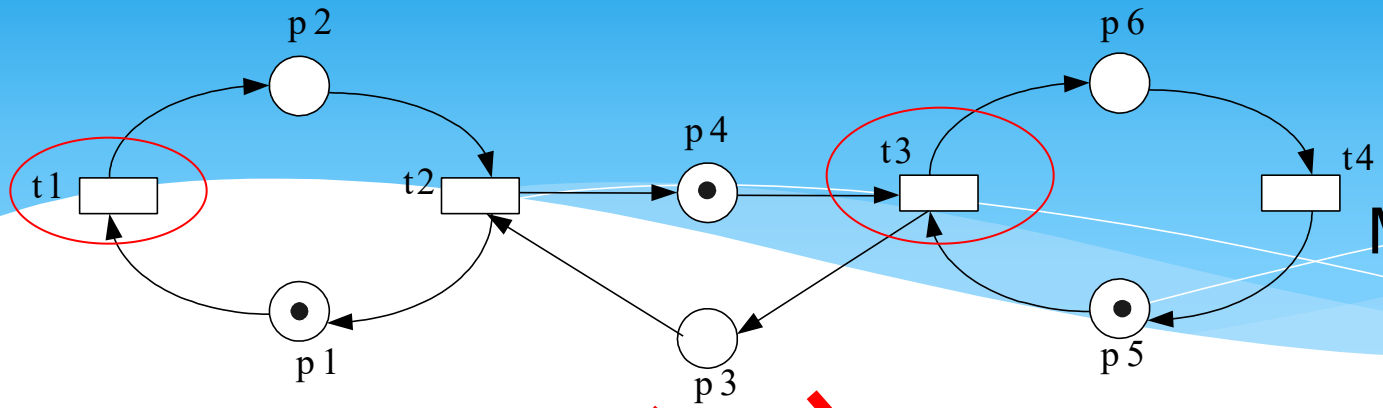
t2



M_3=p1+p4+p5

[1 0 0 1 1 0]

Running (3)

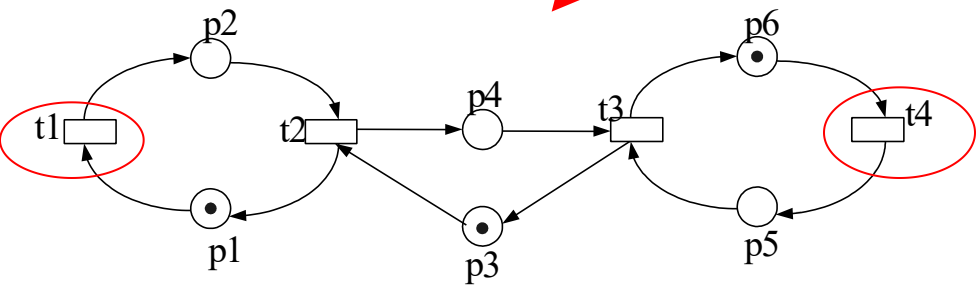


$M_3=p1+p4+p5$

[1 0 0 1 1 0]

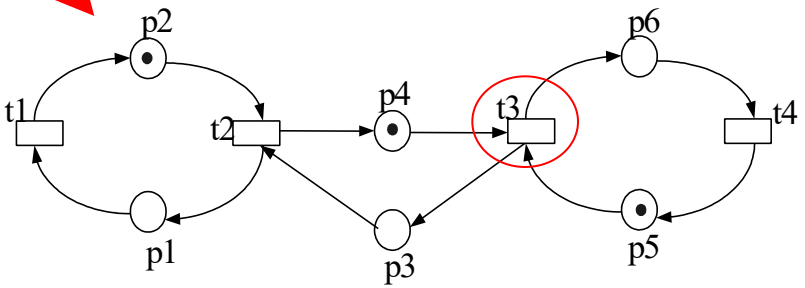
t3

t1



$M_4=p1+p3+p6$

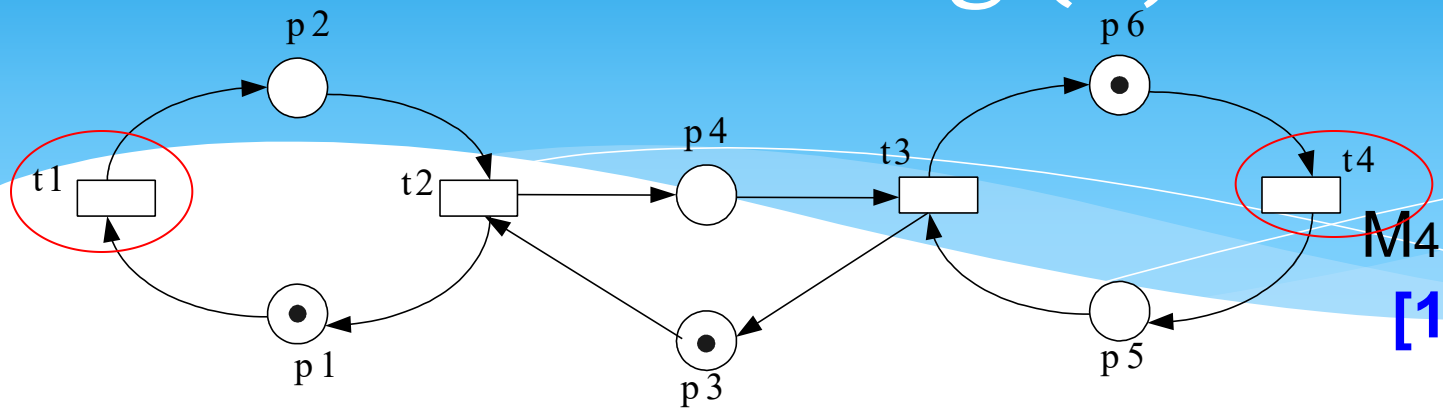
[1 0 1 0 0 1]



$M_5=p2+p4+p5$

[0 1 0 1 1 0]

Running (4)

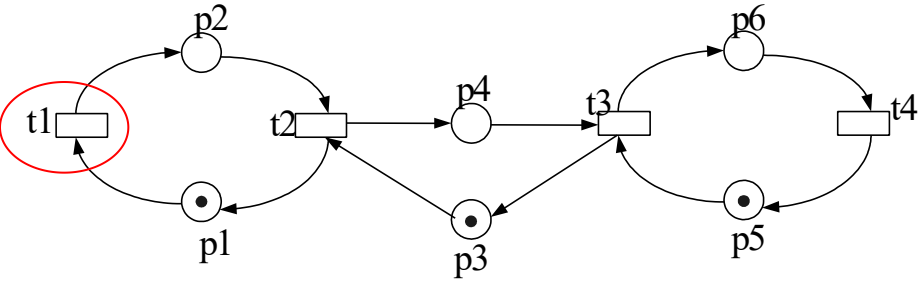


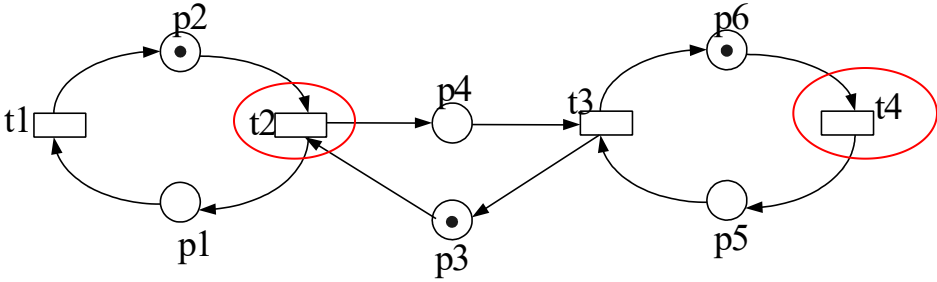
M_4=p1+p3+p6

[1 0 1 0 0 1]

t4

t1





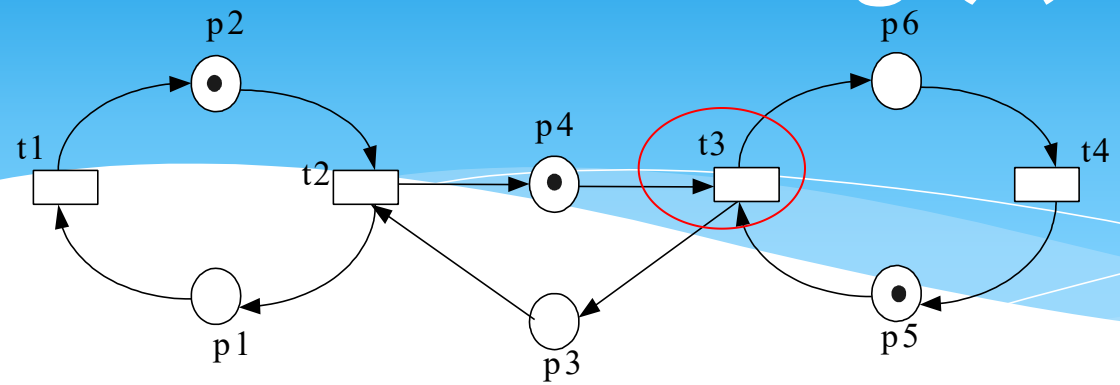
M_1=p1+p3+p5 (back to initial)

[1 0 1 0 1 0]

M_6=p2+p3+p6

[0 1 1 0 0 1]

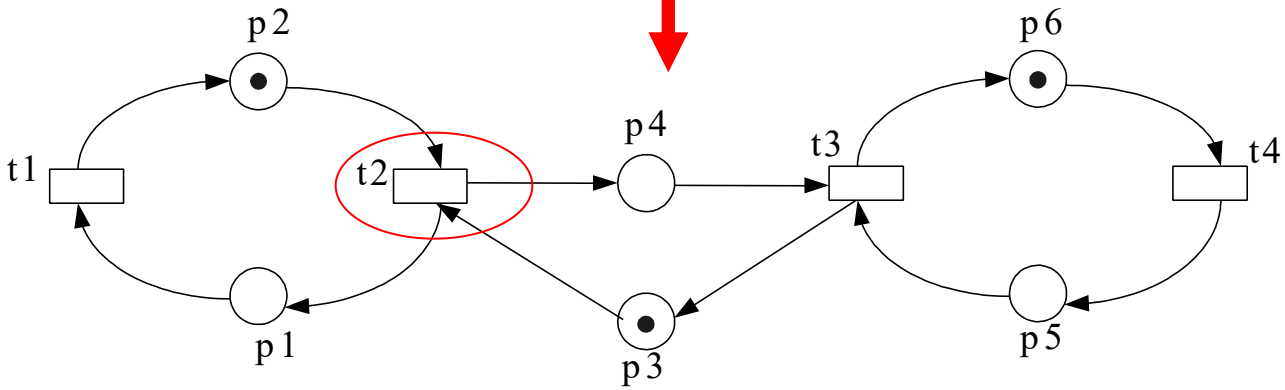
Running (5)



$M_5 = p2 + p4 + p5$

[1 0 1 0 1 0]

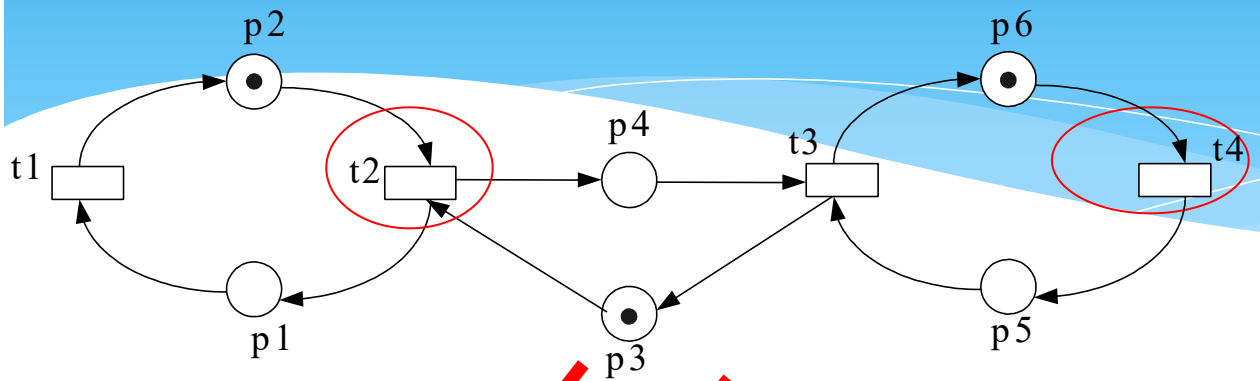
t3



$M_6 = p2 + p3 + p6$

[0 1 1 0 0 1]

Running (6)

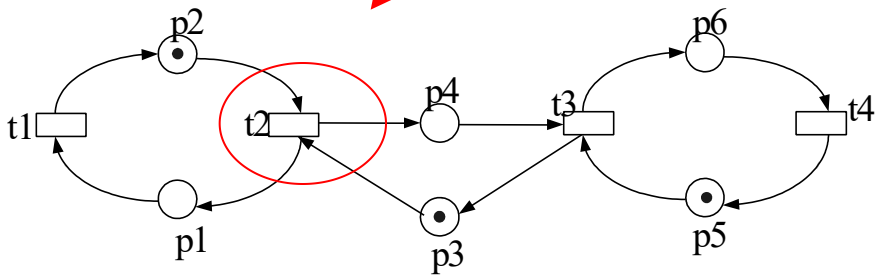


$M_6=p2+p3+p6$

[0 1 1 0 0 1]

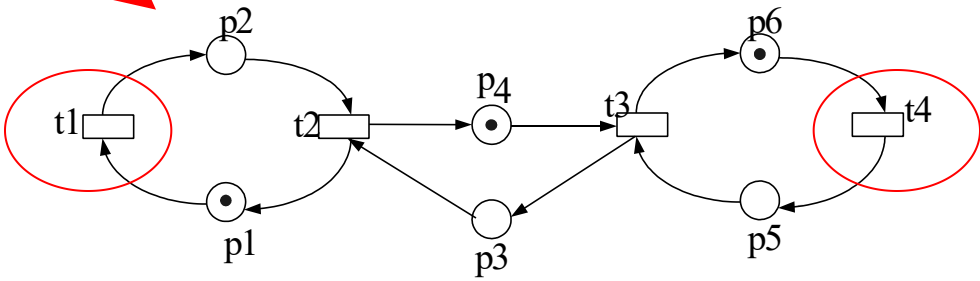
t4

t2



$M_2=p2+p3+p5$

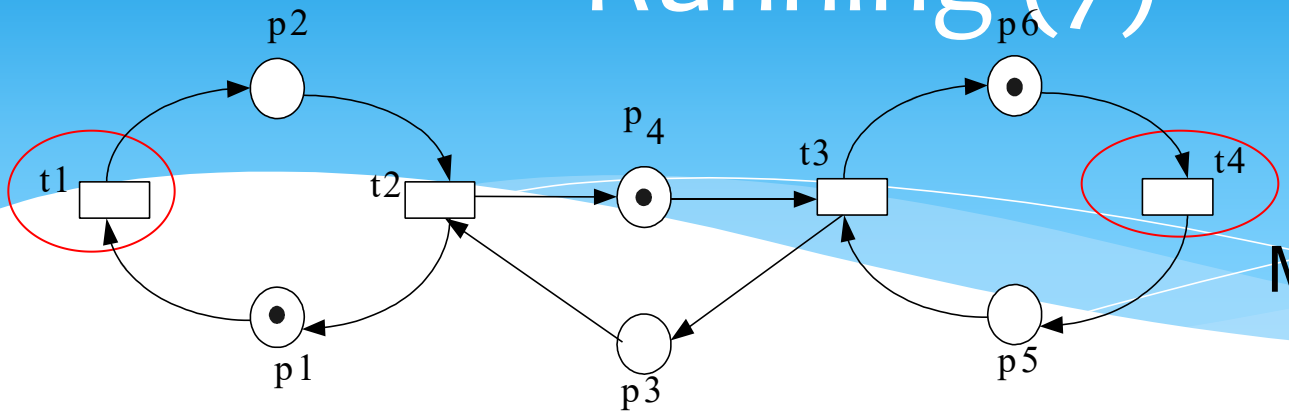
[0 1 1 0 1 0]



$M_7=p1+p4+p6$

[1 0 0 1 0 1]

Running (7)

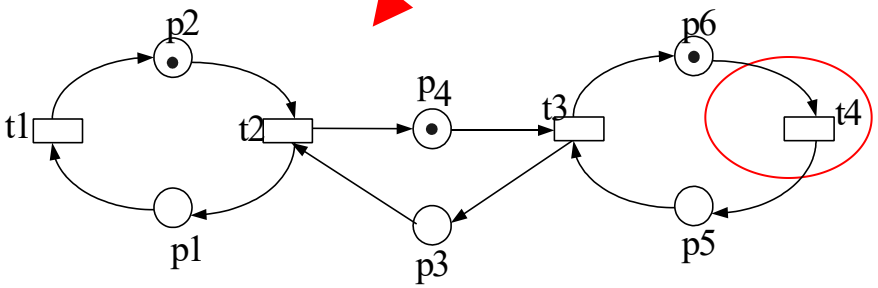


M_7=p1+p4+p6

[1 0 0 1 0 1]

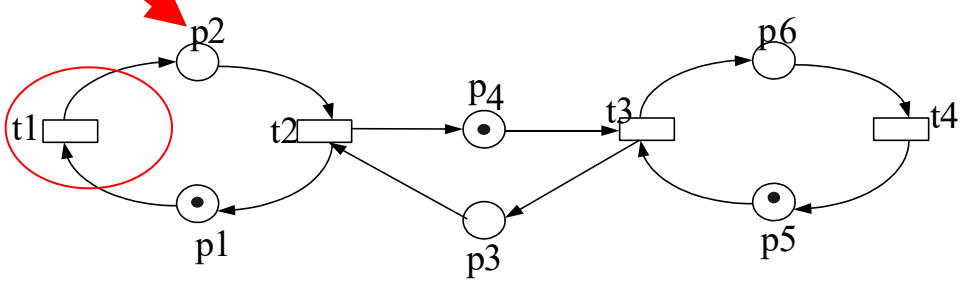
t1

t4



M_8=p2+p4+p6

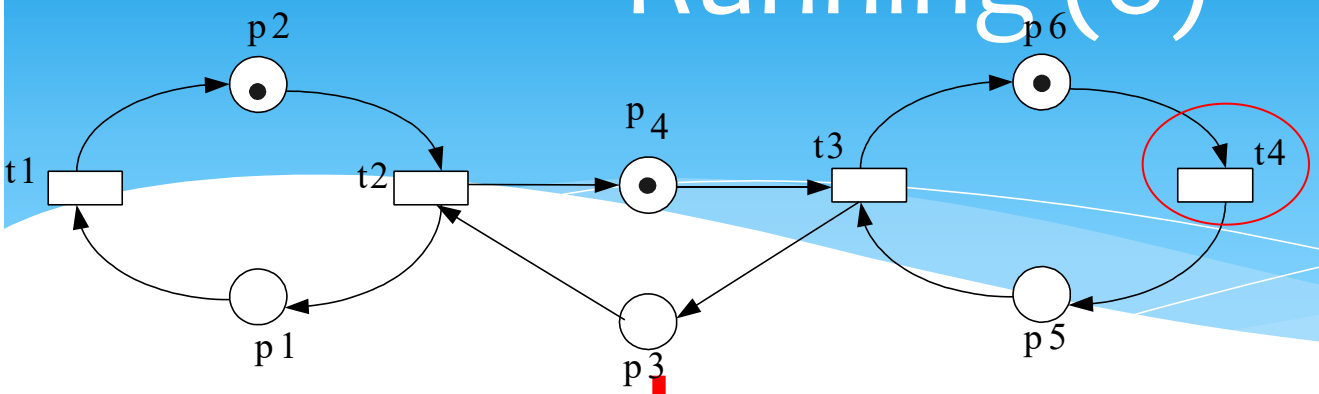
[0 1 0 1 0 1]



M_3=p1+p4+p5

[1 0 0 1 1 0]

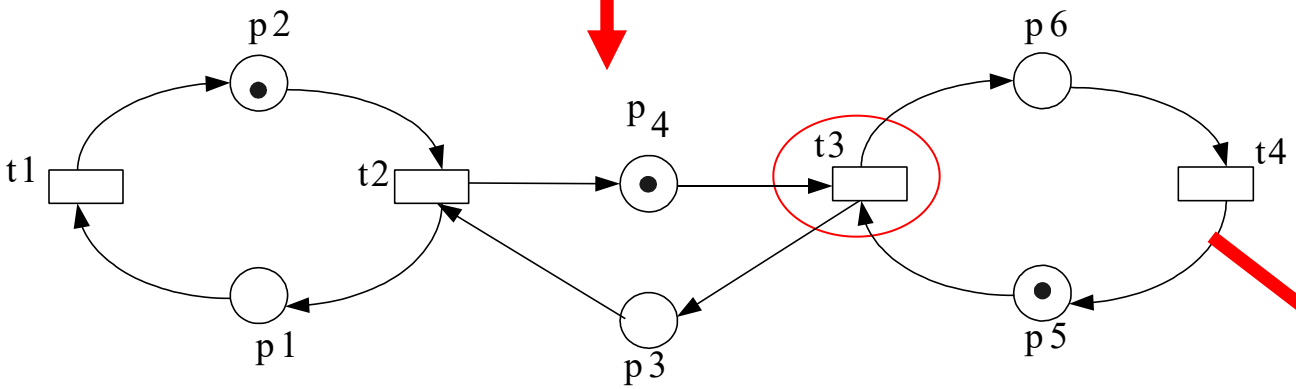
Running (8)



M_8=p2+p4+p6

[0 1 0 1 0 1]

t4



t3

M_5=p2+p4+p5

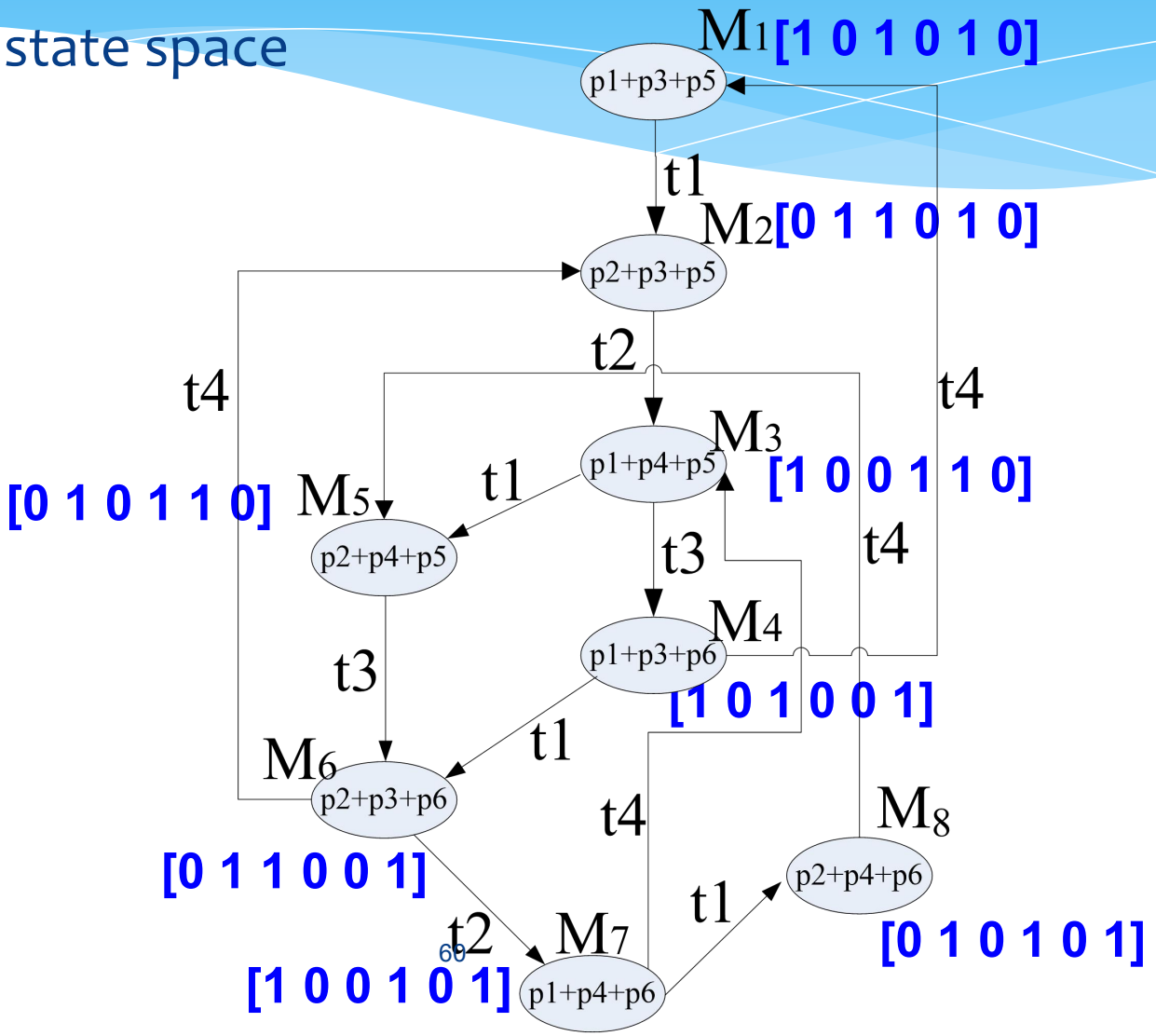
[1 0 1 0 1 0]

M_6=p2+p3+p6

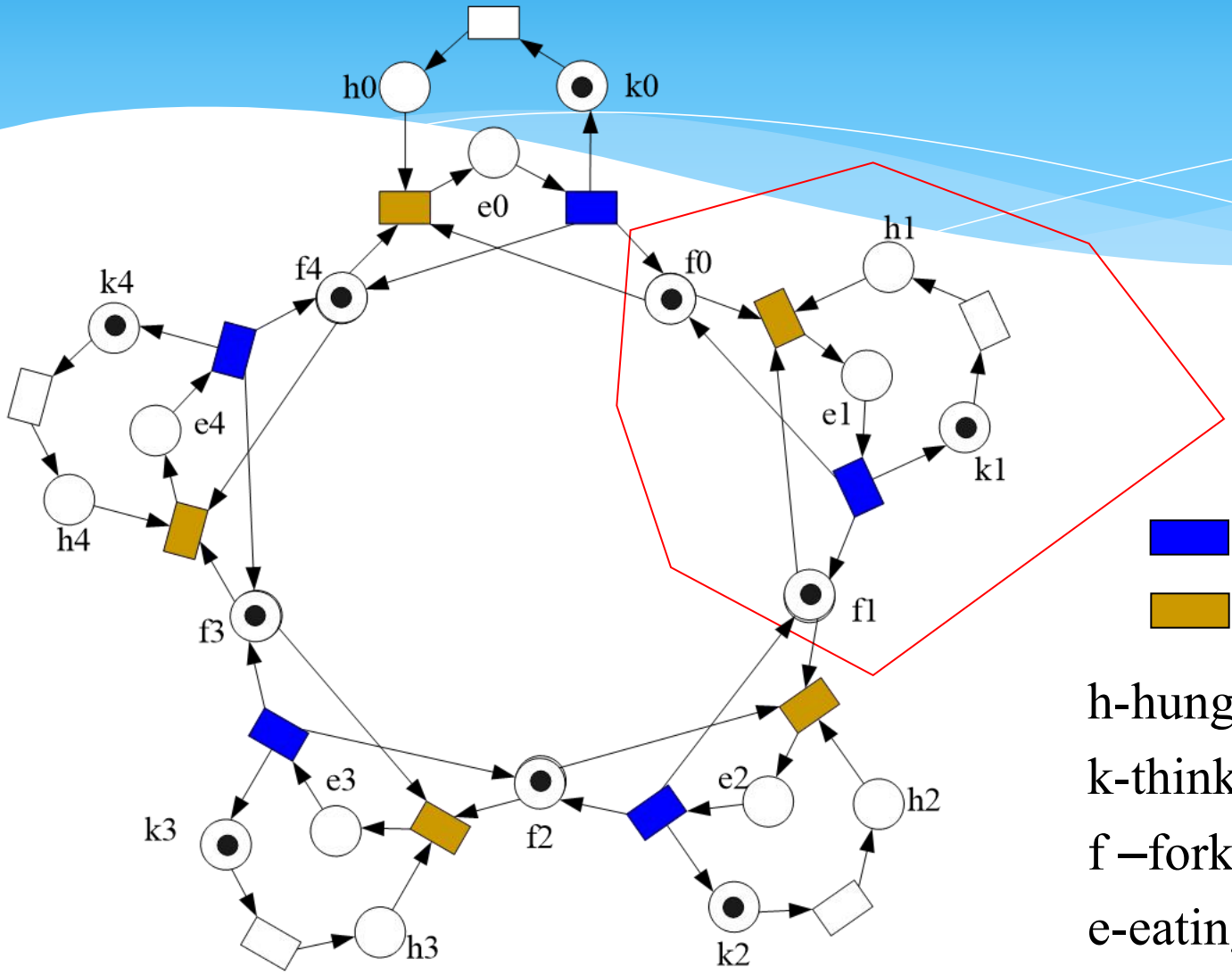
[0 1 1 0 0 1]

State space -- reachability graph

* Connect the arc relations between the markings
  * -- get the state space



例:哲学家问题的P/T系统描述



释放筷子

获取筷子

h-hunger

k-thinking

f –fork/chopstick

e-eating

State space method

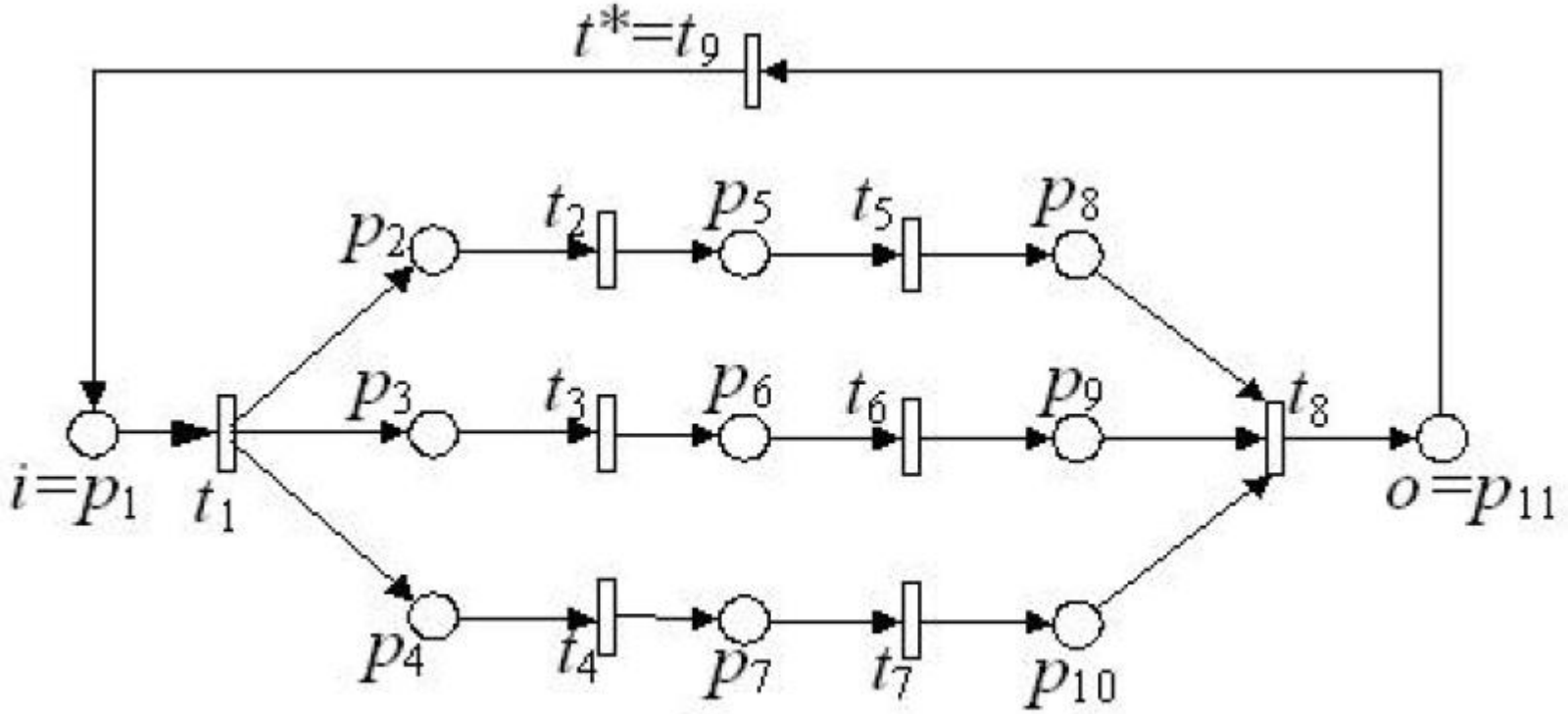


Figure 3.7 A workflow net with three concurrent branches

State space method

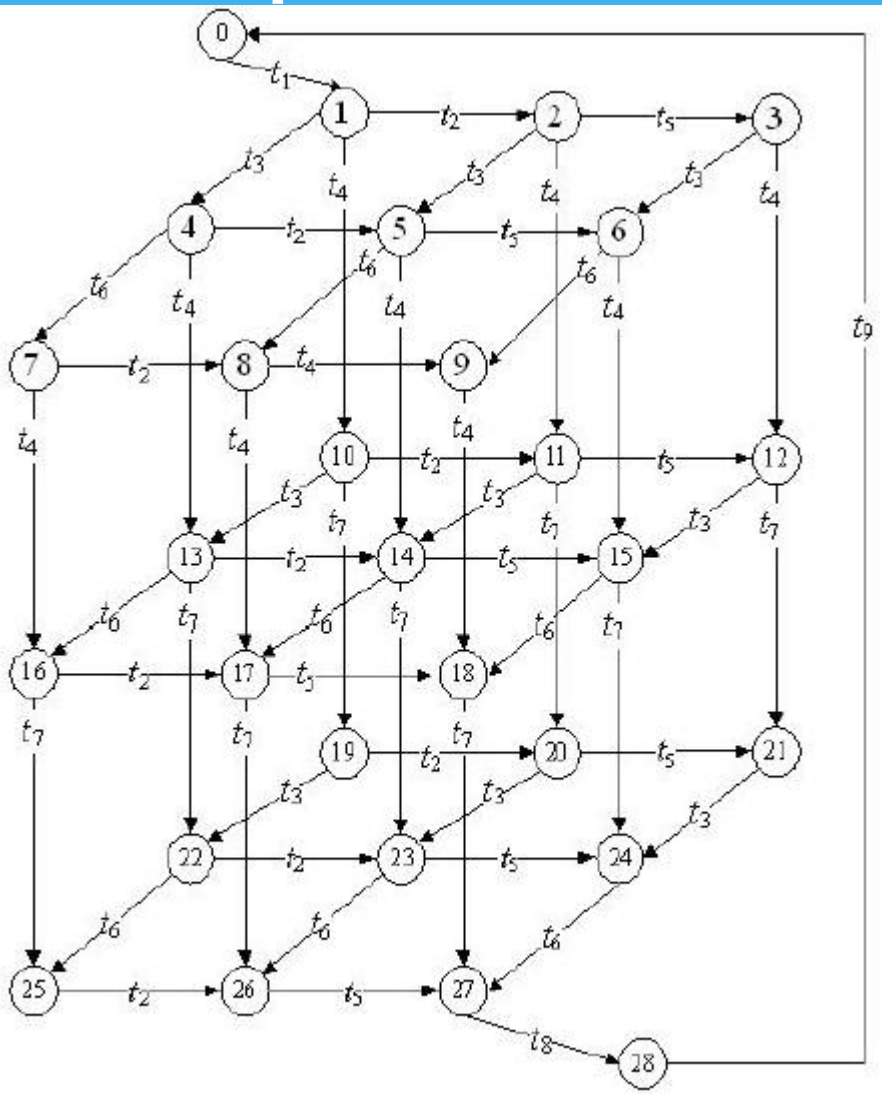


Figure 3.8 The Reachability Graph of Figure 3.7

State space method and Model Checking

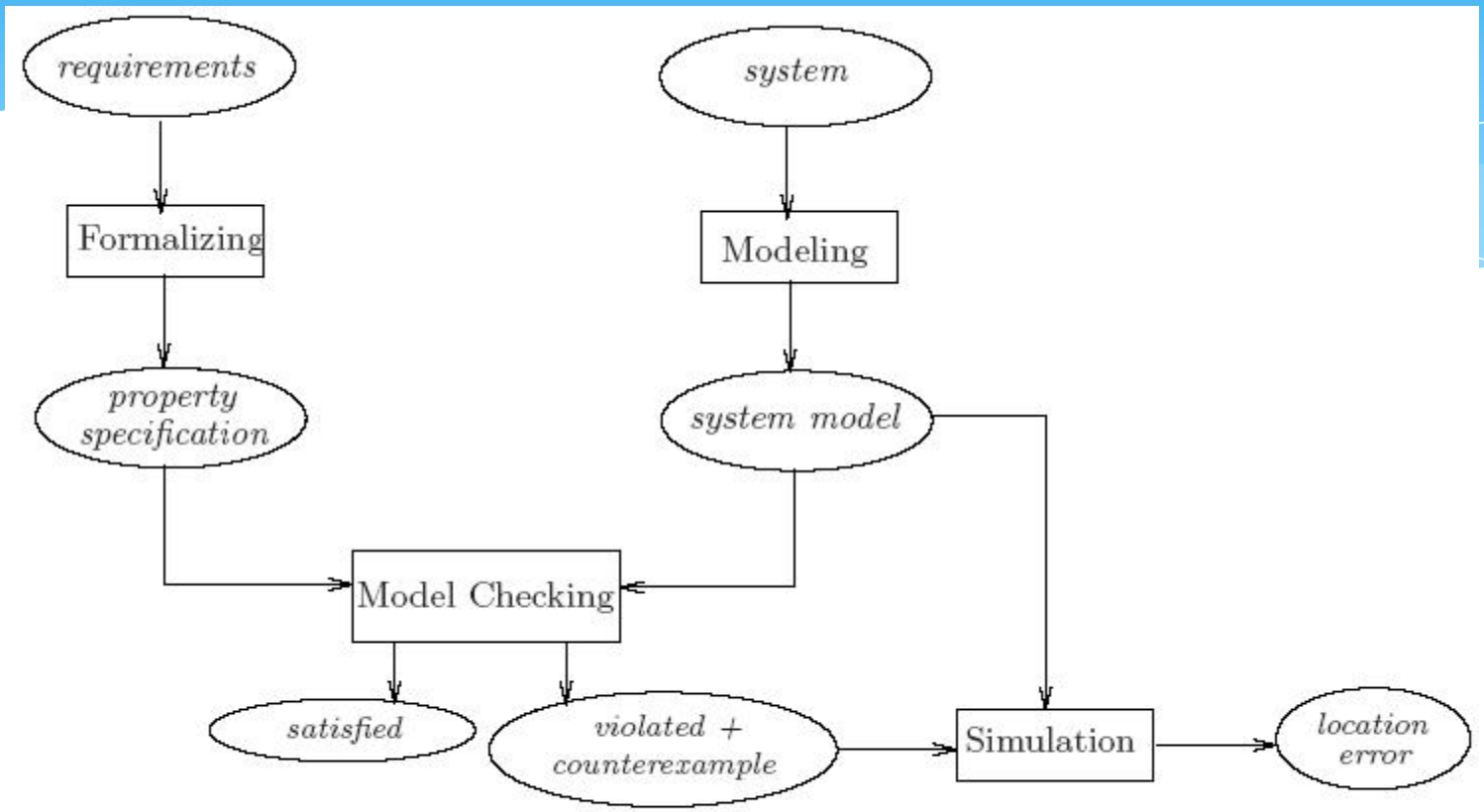
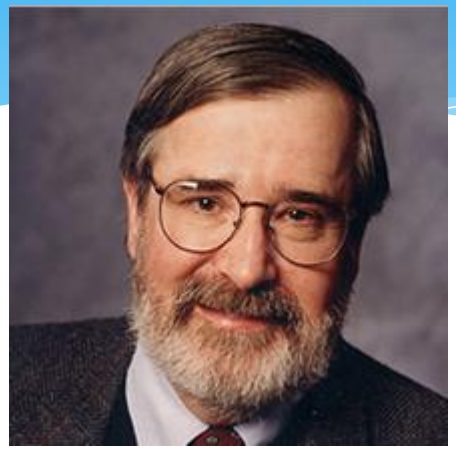


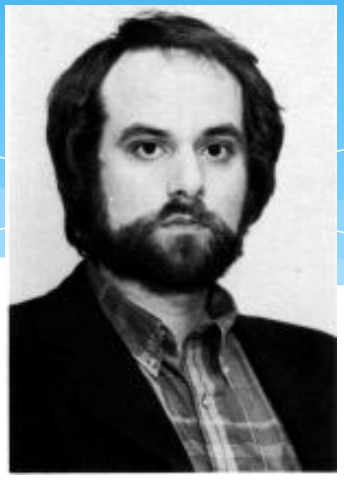
Figure 1.4: Schematic view of the model-checking approach

Model checking is an effective technique to expose potential design errors.

与并发理论有关的图灵奖得主



爱德蒙·克拉克 (Edmund Clarke)
(1945~)
CMU教授, 2007年图灵奖
http://www.cs.cmu.edu/~emc/



艾伦·爱默生 (Ernest Allen Emerson)
(1954~)
2007年图灵奖
http://www.cs.utexas.edu/~emerson/

约瑟夫·斯发基斯 (Joseph Sifakis)
(1945~)
2007年图灵奖
http://www-verimag.imag.fr/~sifakis/

与并发理论有关的图灵奖得主

阿米尔·伯努利(Amir Pnueli)

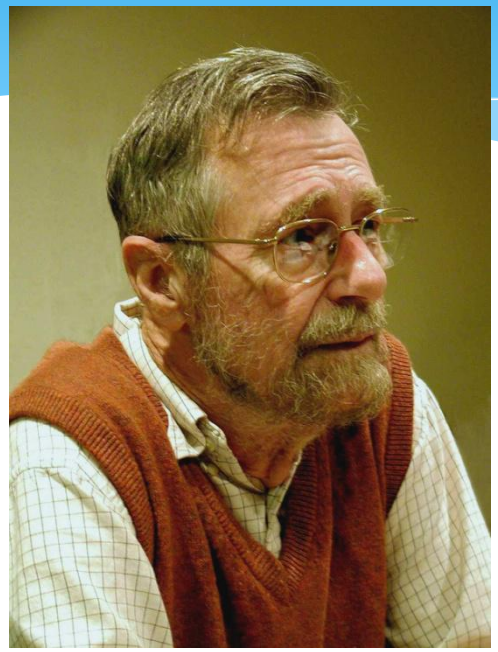
(1941~ 2009)

New York University

1996年图灵奖

http://www.cs.nyu.edu/cs/faculty/pnueli/

与并发理论有关的图灵奖得主



E. W. Dijkstra
(1930-2002)
UT Austin大学, 1972年图灵奖

PV操作, 程序设计理论,
并发理论, 算法, etc.
http://www.cs.utexas.edu/users/EWD/

Tony Hoare
C. A. R. Hoare
(1934-
牛津大学, 1980年图灵奖

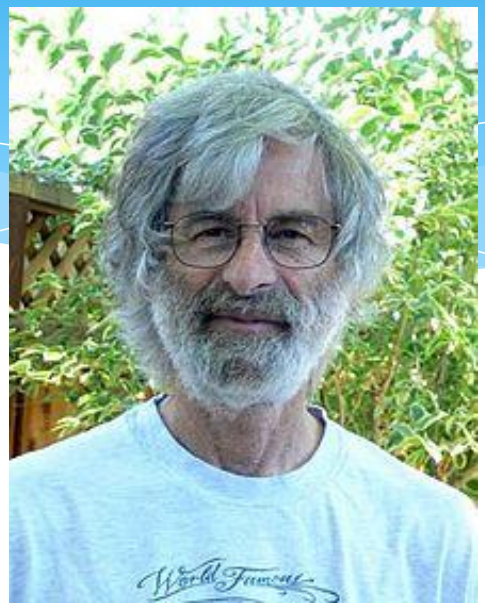
Communicating Sequential Processes,
Hoare Logic, etc.

与并发理论有关的图灵奖得主

Robin Milner
(1934-2010)
剑桥大学, **1991**年图灵奖

http://www.cl.cam.ac.uk/archive/rm135/

CCS – Calculus of Communication Systems
Pi – calculus, ML-Meta Language
LCF: Logic of Computable Functions, etc.
1991年图灵奖



Leslie Lamport
(1941-)
2013年图灵奖

http://www.lamport.org/
http://research.microsoft.com/users/lamport/
LaTex, Lamport's Tex
分布式算法
分布式系统的正确性验证, 时序逻辑,
TLA -- The Temporal Logic of Actions
ACM ToPAL 16, 3 (May 1994), 872-923

Symbolic Model Checking (符号模型检验)
The Cadence SMV Model Checker

R.E. Bryant曾经担任CS.CMU系主任
R.E. Bryant, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Transactions on Computers, Vol. C - 35, No. 8, August, 1986, pp. 677-691. 累计被引用10495(谷歌学术20170613)

Kenneth L. McMillan http://www.kenmcmil.com/
Edmund Clarke的学生
代表性学术贡献: Symbolic Model Checking (符号模型检验,1992年)
支持10^20状态, 博士论文为ACM优秀博士论文
使用OBDD数据结构 (BDD数据结构源于CMU的Randal Bryant教授)
https://en.wikipedia.org/wiki/Binary_decision_diagram
The Cadence SMV Model Checker
http://www.cs.cmu.edu/~modelcheck/smv.html

Spin Model Checker

Gerard Holzmann
http://spinroot.com/gerard/
nasa/jpl lab for reliable software
ACM System Software Awards (2001)

Tony Hoare对中国的影响

周巢尘院士
中科学院软件研究所

何积丰院士
华东师范大学软件学院