3.5.1 资源问题

1. 可重用性资源和消耗性资源

1) 可重用性资源　　性质如下：

(1) 每一个可重用性资源中的单元只能分配给一个进程使用；

(2) 遵循可重用资源的使用顺序：请求、使用、释放；

(3) 系统中每一类可重用性资源中的单元数目是相对固定的，系统中的大多数资源都是可重用的。

3.5.1 资源问题

1. 可重用性资源和消耗性资源

1) 可重用性资源

2) 可消耗性资源，又称为临时性资源

是进程运行期间，由进程动态创建和消耗的，其性质如下：

(1) 每一类可消耗性资源中的单元数目在进程运行期间是可以不断变化的；

(2) 进程运行期间，可不断地创造可消耗性资源；

(3) 进程在运行期间可以请求若干个可消耗性资源单元。

3.5.1 资源问题

1. 可重用性资源和消耗性资源

2. 可抢占性资源和不可抢占性资源

1) 可抢占性资源

可抢占资源如CPU、主存，这类资源不会引起死锁。

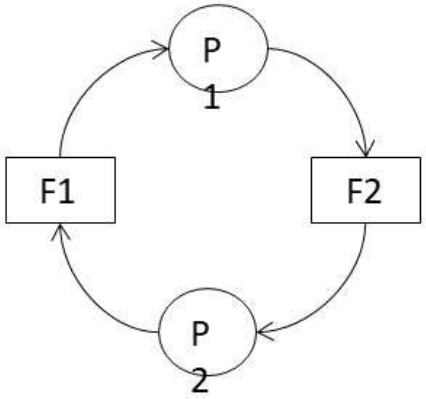
2) 不可抢占性资源

不可抢占资源只能在使用后释放，如刻录机、磁带机、打印机等，这类资源因竞争会引发死锁。

1. 竞争不可抢占性资源引起死锁

如进程P1、P2共享两个写文件F1、F2，
写文件属于可重用和不可抢占性资源。

P1	P2
.....	.....
Open(f1,w)	Open(f2,w)
Open(f2,w)	Open(f1,w)
.....	.....



用资源分配图描述如图所示

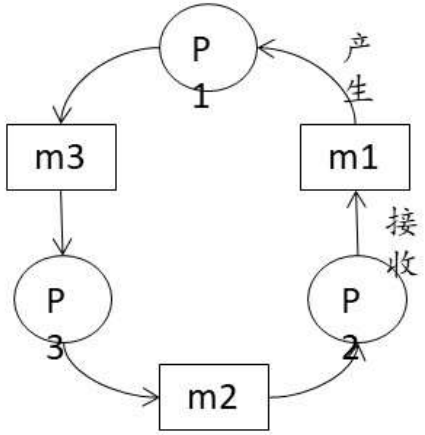
3.5.2 计算机系统中的死锁

1. 竞争不可抢占性资源引起死锁

2. 竞争可消耗资源引起死锁

如进程P1、P2、P3用消息通信机制进行通信，如图

```
P1:  … send(p2,m1) ; receive(p3,m3);…
P2:  … send(p3,m2) ; receive(p1,m1);…
P3:  … send(p1,m3) ; receive(p2,m2);…
```



先发送信件不会死锁，若改成先接收信件就会导致死锁的发生。

```
P1:  … receive(p3,m3); send(p2,m1) ;…
P2:  … receive(p1,m1); send(p3,m2) ;…
P3:  … receive(p2,m2); send(p1,m3) ;…
```

思考题

- 一个OS有20个进程，竞争使用65个同类资源，申请方式是逐个进行的，一旦某个进程获得它所需要的全部资源，则立即归还所有资源。每个进程最多使用4个资源。若仅考虑这类资源，该系统有无可能产生死锁，为什么?
- 答：不会产生死锁。因为死锁产生的原因有两点：系统资源不足或推进顺序不当。
- 本题中，在最坏的情况下，每一进程获得了3个资源，系统还剩余5个资源，至少能满足5个进程需求，从而释放资源，因此该系统不会产生死锁。

3.5.3 死锁的定义、必要条件和处理方法

1. 死锁的定义

如果一组进程中的每一个进程都在等待仅由该组进程中的其它进程才能引发的事件，那么该组进程是死锁的。

❖产生死锁的四个必要条件中，互斥条件由设备的固有特性所决定的，不仅不能改变，相反还应加以保证，因此实际上只有三个条件予以破坏。

2. 产生死锁的必要条件

(1) 互斥条件　　(2) 请求和保持条件
(3) 不剥夺条件　　(4) 环路等待条件

3. 处理死锁的方法

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

3.6.1 破坏“请求和保持”条件

前提是在进程请求资源时，它没有不可抢占的资源。通过如下2个不同协议实现：

1. 第一种协议

进程在运行前，必须一次性地申请其在整个运行过程中所需的全部资源。如能满足则运行，这就破坏了“请求和保持”条件 ，从而可以预防死锁。

优点：简单、易行且安全

缺点：资源浪费严重；
进程会产生饥饿现象。

3.6.3 破坏“循环等待”条件

- 系统中的所有资源按类都被赋予一个唯一的编号，每个进程只能按编号的升序申请资源。即对同一个进程而言，它一旦申请了一个编号为i的资源，就不允许再申请编号比i小的资源了，因此，破坏了循环等待条件。
- 优点：安全且资源利用率比静态资源分配法有所提高，因为它实际是一种半动态的资源分配法。
- 缺点：实现较困难，因为难给出合适的资源编号，不便于系统增添新设备，不便于用户编程，且仍有一定的资源浪费现象。

资源编号磁带机1，硬盘驱动器5，打印机12等等。
该方法称为按序分配。

3.6.3 破坏“循环等待”条件

【利用反证法证明按序分配不会产生死锁】

- 假设按序分配出现了死锁，则由死锁的定义可知系统中存在一组进程（设有n个进程）构成了一个循环等待链：设进程P1等待的资源R_{k1}被进程P2所占有，进程P2等待的资源R_{k2}被进程P3所占有，进程Pn等待的资源R_{kn}被进程P1所占有如图，依按序分配的原则有：k1<k2<k3<.....ki<k(i+1)<.....<k(n-1)<kn<k1，
- 即得k1<k1，这是错误的结论，因此假设前提不成立，即按序分配不会出现死锁。证明完毕。

图 按序分配示意图

3.7 避免死锁

死锁的避免与死锁防止策略不同，它不对进程申请资源加任何限制，而是对进程提出的每一次资源请求进行动态检查，并根据检查结果决定是否分配资源以满足进程的请求。由于采用了动态的资源分配策略，所以资源利用率比死锁的防止办法高。

因此将系统状态划分为安全状态、与非安全状态。

3.7.1 系统安全状态

1. 安全状态与不安全状态

所谓安全状态，是指系统能按某种进程顺序(P_1, P_2, …, P_n)(称〈P_1, P_2, …, P_n〉序列为安全序列)，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。**

安全序列的实质

序列中的每一个进程Pi（i=1, 2, …, n）到以后运行完成所需要的资源量不超过系统当前剩余的资源量与所有在序列中排在它前面的进程当前所占有的资源量之和。

2. 安全状态之例

安全序列：P_2 P_1 P_3

因此T_0时刻是安全的

假定系统中有三个进程P_1、P_2和P_3，共有12台磁带机。进程P_1总共要求10台磁带机，P_2和P_3分别要求4台和9台。假设在T_0时刻，进程P_1、P_2和P_3已分别获得5台、2台和2台磁带机，尚有3台空闲未分配，如下表所示：

进程	最大需求	已分配	可用	还需要
P_1	10	5	5	5
P_2	4	2		2
P_3	9	2	10	7
			12	

P_2

P_1

P_3

3. 由安全状态向不安全状态的转换

进程	最大需求	已分配	可用	还需要
P_1	10	5	4	5
P_2	4	2		2
P_3	9	3		6

P_2

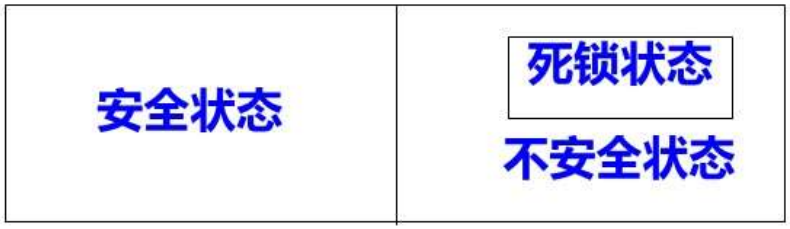
例如，在T_0时刻以后，P_3又请求1台磁带机，假设完成此次请求。

此时找不到一个安全序列，则系统进入不安全状态。

3.7.1 系统安全状态

注意:

- ✓ （1）系统在某一时刻的安全序列可能不唯一，但这不影响对系统安全性的判断。
- ✓ （2）安全状态是非死锁状态，而不安全状态并不一定是死锁状态。即系统处于安全状态一定可以避免死锁，而系统处于不安全状态则仅仅可能进入死锁状态。



3.7.2 利用银行家算法避免死锁

银行家算法执行的假设前提条件：

设系统有M类资源（$R_1, R_2, \cdots, R_M$），每类资源有若干个，系统拥有的各类资源数量是固定的；

系统有N个进程（$P_1, P_2, \cdots, P_N$），每一进程运行前要先提出自己的最大资源请求，此后进程对资源的需求是随着进程的推进而分步提出的。

- 银行家算法就是对每一个客户的请求进行检查，如果满足客户请求会使系统进入不安全状态，则不满足该请求，否则就满足请求。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(1) 可利用资源向量Available, 定义为 int av[M];

其初始值是系统中所配置的全部可用资源的数目，其数值记录系统中各类资源的当前可利用数目，如果av[j]=k，则表示系统中现有R_j类资源k个。

(2) 最大需求矩阵Max。定义为 int max[N][M];

它定义了系统中N个进程中的每一个进程对M类资源的最大需求。如果max[i][j]=k，则表示进程P_i需要R_j类资源的最大数目为k。

3.7.2 利用银行家算法避免死锁

1. 银行家算法中的数据结构

(3) 分配矩阵Allocation，定义为 alt[N][M];

它定义了系统已为每个进程分配的数量或者说每个进程对各类资源当前的占有量。如果alt[i][j]=k，则表示进程P_i当前已分得R_j类资源的数目为k。

(4) 需求矩阵Need。定义为 need[N][M];

用以表示每一个进程尚需的各类资源数。如果need[i][j]=k，则表示进程P_i还需要R_j类资源k个，方能完成其任务。

显然：need[i][j]=max[i][j]-alt[i][j]

3.7.2 利用银行家算法避免死锁

2. 银行家算法

请求向量Request：当某个进程P_i运行时，它可对各类资源提出使用请求，称之为请求向量。定义为 $req_i[M]$；

如果$req_i[j]=k$，表示进程P_i此次需要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$re_i[j] \leqslant need[i][j]$，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$req_i[j] \leqslant av[j]$，便转向步骤(3)；否则表示尚无足够资源，P_i须等待。

2. 银行家算法

(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：

$av[j]=av[j]-req_i[j];$

$alt[i][j]=alt[i][j]+req_i[j];$

$need[i][j]=need[i][j]-req_i[j];$

(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。

3.7.2 利用银行家算法避免死锁

3. 安全性算法

(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（ int work[M] ），其初值为av；

② 标志向量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（int finish[N] ），其初值为0；当有足够资源分配给进程时，再令finish[i]=1。

3.7.2 利用银行家算法避免死锁

3. 安全性算法

(2) 从进程集合中找到一个能满足下述条件的进程：

① finish[i]=0；② need[i][j]≤work[j]； 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：

```
work[j]=work[i]+alt[i][j];
finish[i]=1;
go to step 2;
```

(4) 如果所有进程的finish[i]= =1，则表示系统处于安全状态；否则，系统处于不安全状态。

4. 银行家算法举例

假定系统中有五个进程{P_0、P_1、P_2、P_3、P_4}和三种类型的资源{A，B，C}，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：

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	

T_0时刻是否安全？

3.7.2 利用银行家算法避免死锁

4. 银行家算法举例—T0时刻安全验证

T_0时刻是否安全？

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			

work		
A	B	C
3	3	2
5	3	2
7	4	3
7	4	5
7	5	5
10	5	7

即安全序列为P_1、P_3、P_4、P_0、P_2；T_0时刻是安全的

4. 银行家算法举例-- P1请求req_1（1,0,2）

系统进行检查：

①$request_1$（1,0,2）≤ $need_1$（1,2,2）；

②$request_1$（1,0,2）≤ $available_1$（3,3,2）；

③系统先假定可为P_1分配资源，并修改如下：

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	2 3 0
P_1	3 2 2	3 0 2	0 2 0	
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	

T_1时刻是否安全？

4. 银行家算法举例-- P1请求req_1（1,0,2）

T_1时刻是否安全？

资源情况 / 进程	max	allocation	need	available	work
	A B C	A B C	A B C	A B C	A B C
P_0	7 5 3	0 1 0	7 4 3	2 3 0	2 3 0
P_1	3 2 2	3 0 2	0 2 0		5 3 2
P_2	9 0 2	3 0 2	6 0 0		7 4 3
P_3	2 2 2	2 1 1	0 1 1		
P_4	4 3 3	0 0 2	4 3 1		

即安全序列为P_1、P_3、P_4、P_0、P_2；T_1时刻是安全的

4. 银行家算法举例 - P4请求req_4（3, 3, 0）

系统进行检查：

①$request_4(3,3,0) \le need_4(4,3,1)$；

②$request_4(3,3,0) \le available(2,3,0)$；

条件②不满足，拒绝P4请求，令其阻塞。

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	2 3 0
P_1	3 2 2	3 0 2	0 2 0	
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	

4. 银行家算法举例 - P0请求req_0（0, 2, 0）

T_2时刻是否安全？

系统进行检查：

①$request_0(0,2,0) \le need_0(7,4,3)$；

②$request_0(0,2,0) \le available(2,3,0)$；

③系统先假定可为P_0分配资源，并修改如下：

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	3	0	7	2	3	2	1	0
P_1	3	2	2	3	0	2	0	2	0			
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			

4. 银行家算法举例 - P0请求req_0（0, 2, 0）

T_2时刻是否安全？

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	3	0	7	2	3	2	1	0
P_1	3	2	2	3	0	2	0	2	0			
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			

work		
A	B	C
2	1	0

进程P_1、P_2、P_3、P_4、P_0都不能满足，故T_2时刻是不安全的

撤销为P_0分配的资源（0 2 0）

3.7.2 利用银行家算法避免死锁

银行家算法小结：

- 银行家算法从避免死锁的角度上说是非常有效的。
- 从某种意义上说，它缺乏实用价值，因为很少有进程能够在运行前就知道其所需资源的最大值，而且进程数也不是固定的，往往在不断地变化（如新用户登录或退出），况且原本可用的资源也可能突然间变成不可用（如磁带机可能坏掉）。
- 因此，在实际中，如果有，也只有极少的系统使用银行家算法来避免死锁。

3.7.2 利用银行家算法避免死锁

例题：某系统有同类互斥资源m个，供n个进程共享使用。如果每个进程最多申请x个资源（1≤x≤m），

试证明：当n*(x-1)+1≤m时，系统不会发生死锁。

- 证明：
- 因为每个进程最多申请x个资源，所以最坏情况是每个进程都得到了（x-1）个资源，并且现在均需申请最后一个资源。
- 此时，系统剩余资源数为m-n*(x-1)，于是只要系统中至少还有一个资源可供使用，就可以使这n个进程中某个进程得到其所需要的全部资源，并能够继续执行到完成，归还资源可供其他进程继续使用。因而不会发生死锁。
- 即只要m-n*(x-1)≥1时，系统就一定不会发生死锁。亦即当n*(x-1)+1≤m时，系统不会发生死锁。

死锁概率小，可对资源分配不予控制，系统可提供2个算法：死锁检测以及死锁解除。

3.8.1 死锁的检测

1. 资源分配图(Resource Allocation Graph)

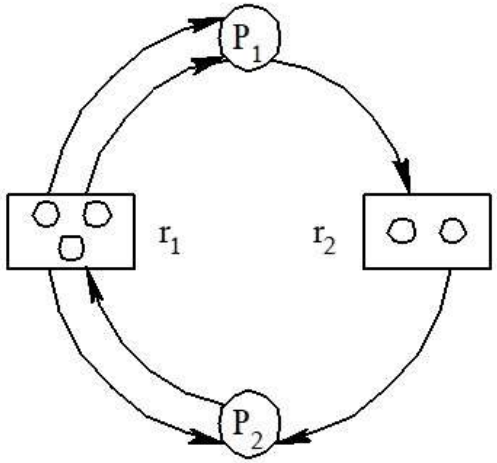


图 3-19 每类资源有多个时的情况

凡属于E中的一个边e∈E，都连接着P中的一个结点和R中的一个结点，$e=\{p_i, r_j\}$是资源请求边，由进程p_i指向资源r_j，它表示进程p_i请求一个单位的r_j资源。$e=\{r_j, p_i\}$是资源分配边，由资源r_j指向进程p_i，它表示把一个单位的资源r_j分配给进程p_i。

2. 死锁定理

S为死锁状态的充分条件是：当且仅当S状态的资源分配图是不可完全简化的。

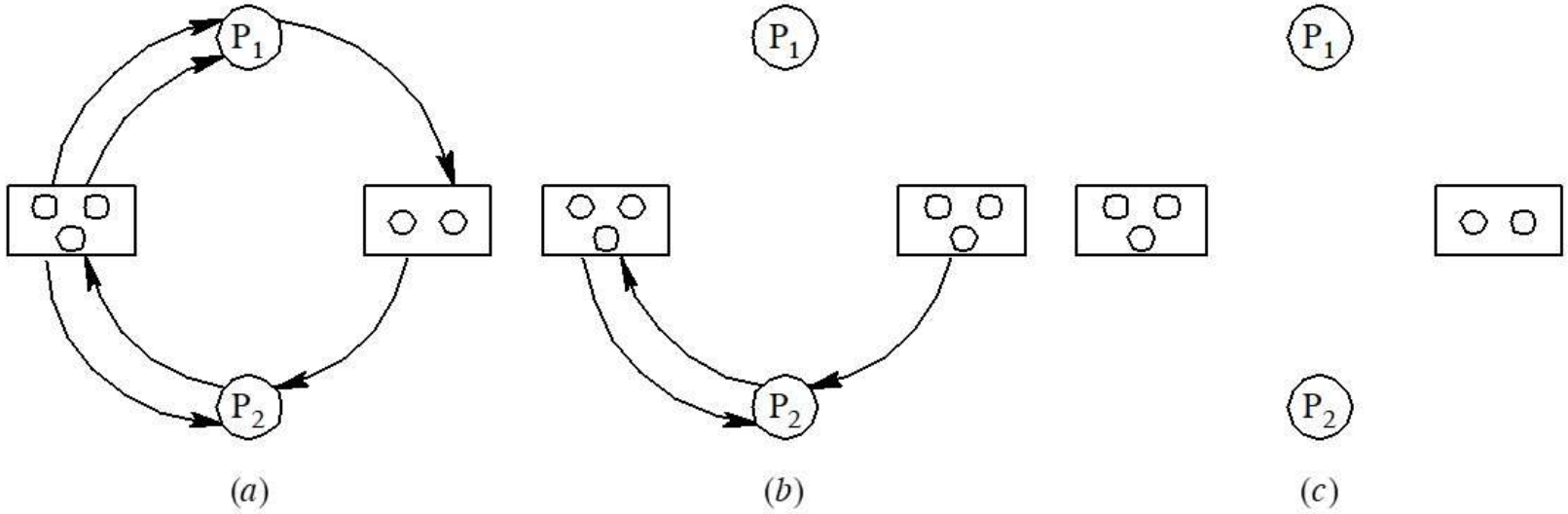


图 3-20 资源分配图的简化

3. 死锁检测中的数据结构

(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。

(2) 把不占用资源的进程(向量Allocation：=0)记入L表中， 即$L_i \cup L$。

(3) 从进程集合中找到一个$Request_i \leqslant Work$的进程，做如下处理：

① 将其资源分配图简化，释放出资源，增加工作向量Work：=$Work+Allocation_i$。

② 将它记入L表中。

3. 死锁检测中的数据结构

(4) 若不能把所有进程都记入L表中，便表明系统状态S的资源分配图是不可完全简化的。因此，该系统状态将发生死锁。

Work =Available;

L= {L_i | $Allocation_i$=0∩$Request_i$=0}

while (all L_i ∉ L)

{

while (all $Request_i$≤Work)

{

Work=Work+$Allocation_i$;

L= L_i ∪ L;

}

}

deadlock = ～ (L=={p_1, p_2, …, p_n});

3.8.2 死锁的解除

解除死锁的方法：

(1)抢占资源；

(2) 撤消（或终止）进程。

1. 终止进程的方法

(1)终止所有死锁进程，代价较高；

(2)逐个终止进程，直至有足够的资源，打破循环等待以解除死锁。

选择被终止的进程依据：

- 优先级大小；
- 已占用CPU时间，还需要的时间；
- 已使用系统资源，还需要的系统资源；
- 进程的性质是交互式还是批处理式。

改进的死锁解除方式

- 假定在死锁状态S时，有死锁进程P_1、 P_2、… P_k。
  分别撤销进程P_1， P_2， P_k，由状态S–> U_1、 U_2、 U_k，
  选出代价最小的为C_{ui}，撤销P_i。
  付出的代价是 $R(S)_{min}=min\{C_{ui}\}+min\{C_{uj}\}+min\{C_{uk}\}+...$