第三十二章

Concurrency

Common Concurrency Problems

Liu yufeng

Fx_yfliu@163.com

Hunan University

Introduction

- Researchers have spent a great deal of time and effort looking into concurrency bugs over many years.
- Much of the early work focused on **deadlock**.
- More recent work focuses on studying other types of common concurrency bugs (i.e., **non-deadlock bugs**).

What Types Of Bugs Exist?

- What types of **concurrency bugs** manifest in complex, concurrent programs?
- This question is difficult to answer in general, but fortunately, some others have done the work for us.

[L+08] "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" by Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. ASPLOS '08, March 2008, Seattle, Washington. *The first in-depth study of concurrency bugs in real software, and the basis for this chapter. Look at Y.Y. Zhou's or Shan Lu's web pages for many more interesting papers on bugs.*

Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**

Findings on Bug Patterns (Section 3)
(1) Almost all (97%) of the examined non-deadlock bugs belong to one of the *two simple bug patterns*: atomicity-violation or order-violation*.
(2) About one third (32%) of the examined non-deadlock bugs are *order-violation bugs*, which are *not* well addressed in previous work.

Findings on Manifestation (Section 4)
(3) Almost all (96%) of the examined concurrency bugs are guaranteed to manifest if certain partial order between *2 threads* is enforced.
(4) Some (22%) of the examined deadlock bugs are caused by *one thread* acquiring resource held by itself.
(5) Many (66%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *only one variable*.
(6) One third (34%) of the examined non-deadlock concurrency bugs' manifestation involves concurrent accesses to *multiple variables*.
(7) Almost all (97%) of the examined deadlock bugs involve two threads circularly waiting for at most *two resources*.
(8) Almost all (92%) of the examined concurrency bugs are guaranteed to manifest if certain partial order among *no more than 4 memory accesses* is enforced.

Findings on Bug Fix Strategies (Section 5)
(9) Three quarters (73%) of the examined non-deadlock bugs are fixed by techniques *other than* adding/changing locks. Programmers need to consider correctness, performance and other issues to decide the most appropriate fix strategy.
(10) Many (61%) of the examined deadlock bugs are fixed by preventing one thread from acquiring one resource (e.g. lock). Such fix can introduce non-deadlock concurrency bugs.

Findings on Bug Avoidance (Section 5.3)
(11) Transactional memory (TM) can help avoid about one third (39%) of the examined concurrency bugs.
(12) TM *could* help avoid over one third (42%) of the examined concurrency bugs, if some *concerns* are addressed.
(13) Some (19%) of the examined concurrency bugs *cannot* benefit from basic TM designs, because of their bug patterns.

Non-Deadlock Bugs

- 非死锁问题占了并发问题的大多数。它们是怎么发生的？我们如何修 复?
- 主要讨论其中两种：违反原子性（atomicity violation）缺陷和错误顺序（order violation）缺陷。.

Atomicity-Violation Bugs

- Here is a simple example, **found in MySQL**. Before reading the explanation, try figuring out what the bug is.

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

两个线程都要访问 thd 结构中的成员 proc_info。

```
Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

Thread 2::
thd->proc_info = NULL;
```

- 第一个线程检查 proc_info 非空，然后打印出值；第二个线程设置其为空。显然，当第一个线程检查之后，在 fputs() 调用之前被中断，第二个线程把指针置为空；当第一个线程恢复执行时，由于引用空指针， 导致程序奔溃

- 违反原子性的定义是：“违反了多次内存访问中预期的可串行 性（即代码段本意是原子的，但在执行中并没有强制实现原子性）”。在我们的例子中， proc_info 的非空检查和 fputs()调用打印 proc_info 是假设原子的，当假设不成立时，代码 就出问题了。

- Finding a fix for this type of problem is often (but not always) straightforward. Can you think of how to fix the code above?

```
pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;

Thread 1::
pthread_mutex_lock(&proc_info_lock);
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}
pthread_mutex_unlock(&proc_info_lock);

Thread 2::
pthread_mutex_lock(&proc_info_lock);
thd->proc_info = NULL;
pthread_mutex_unlock(&proc_info_lock);
```

Order-Violation Bugs

- Here is another simple example; once again, see if you can figure out why the code below has a bug in it.

```
Thread 1::
void init() {
   mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
    mState = mThread->State;
}
```

```
Thread 1::
void init() {
   mThread = PR_CreateThread(mMain, ...);
}

Thread 2::
void mMain(...) {
    mState = mThread->State;
}
```

- 线程 2 的代码中似乎假定变量 mThread 已经被初始化了（不为空）。然而，如果线程 1 并没有首先执行，线程 2 就可能因为引用空指针奔溃（假设 mThread 初始值为空；否则，可能会产生更加奇怪的问题，因为线程 2 中会读到任意的内存位置 并引用）

- 违反顺序更正式的定义是：“两个内存访问的预期顺序被打破了（即 A 应该在 B 之前执行，但是实际运行中却不是这个顺序）”

- The fix to this type of bug is generally to **enforce ordering**. Using condition variables is an easy and robust way to do this.

```
pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
int mtInit             = 0;

Thread 1::
void init() {
   ...
   mThread = PR_CreateThread(mMain, ...);

   // signal that the thread has been created...
   pthread_mutex_lock(&mtLock);
   mtInit = 1;
   pthread_cond_signal(&mtCond);
   pthread_mutex_unlock(&mtLock);
   ...
}

Thread 2::
void mMain(...) {
    ...
    // wait for the thread to be initialized...
    pthread_mutex_lock(&mtLock);
    while (mtInit == 0)
        pthread_cond_wait(&mtCond, &mtLock);
    pthread_mutex_unlock(&mtLock);

    mState = mThread->State;
    ...
}
```

增加了一个锁（mtLock）、一个条件变量（mtCond）以及状态的变量（mtInit）。初始化代码运行时，会将 mtInit 设置为 1，并发出信号表明它已做了这件事。如果线程 2 先运行，就会一直等待信号和对应的状态变化；如果后运行，线程 2 会检查是否初始化（即 mtInit 被设置为 1），然后正常运行。

Non-Deadlock Bugs: Summary

- A large fraction (**97%**) of non-deadlock bugs studied by Lu et al. are either atomicity or order violations.
- Unfortunately, not all bugs are as easily fixable as the examples we looked at above.
- Some **require a deeper understanding** of what the program is doing, or a larger amount of code or data structure reorganization to fix.

Deadlock Bugs

- Beyond the above concurrency bugs, a classic problem in many concurrent systems is known as **deadlock**.

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

- If Thread 1 grabs lock L1 and then a context switch occurs to Thread 2. Then, Thread 2 grabs L2, and tries to acquire L1, deadlock occurs.

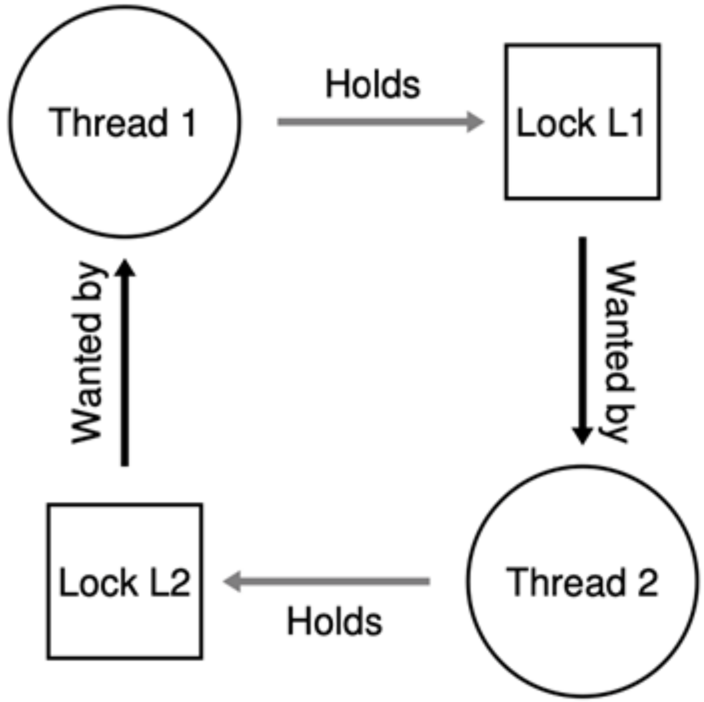


Figure 32.7: **The Deadlock Dependency Graph**

- The presence of a **cycle** in the graph is indicative of the deadlock.
- How should programmers write code so as to handle deadlock in some way?

CRUX: HOW TO DEAL WITH DEADLOCK

How should we build systems to prevent, avoid, or at least detect and recover from deadlock? Is this a real problem in systems today?

Conditions for Deadlock

- **Four conditions** need to hold for a deadlock to occur：
  - **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
  - **Hold-and-wait:** Threads hold resources allocated to them (e.g., locks that they have already acquired) while waiting for additional resources (e.g., locks that they wish to acquire).
  - **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
  - **Circular wait:** There exists a circular chain of threads such that each thread holds one more resources (e.g., locks) that are being requested by the next thread in the chain.
- If **any of** these four conditions are **not met**, deadlock **cannot occur**.

Prevention (Circular Wait)

- 获取锁时提供一个全序（total ordering）。假如系统共有两个锁（L1 和 L2），那么我们每次都先申请 L1 然后申请 L2，就可以避免死锁。这样严格的顺序避免了循环等待，也就不会产生死锁.
- 复杂的系统中不会只有两个锁，锁的全序可能很难做到.
- It requires a deep understanding of the code base; just one mistake could result in the wrong ordering of lock acquisition, and hence deadlock.

当一个函数要抢多个锁时，我们需要注意死锁。比如有一个函数：do_something(mutex t *m1, mutex t *m2)，如果函数总是先抢 m1，然后 m2，那么当一个线程调用 do_something(L1, L2)，而另一个线程调 用 do_something(L2, L1)时，就可能会产生死锁。

为了避免这种特殊问题，聪明的程序员根据锁的地址作为获取锁的顺序。按照地址从高到低，或者 从低到高的顺序加锁，do_something()函数就可以保证不论传入参数是什么顺序，函数都会用固定的顺 序加锁。具体的代码如下：

```
if (m1 > m2) { // grab locks in high-to-low address
order pthread_mutex_lock(m1);
pthread_mutex_lock(m2);
}
else {
pthread_mutex_lock(m2);
pthread_mutex_lock(m1);
} // Code assumes that m1 != m2 (it is not the same
lock)
```

Prevention (Hold-and-wait)

- 死锁的持有并等待条件，可以通过原子地抢锁来避免。实践中，可以通过如下代码来实现:

```
pthread_mutex_lock(prevention);   // begin acquisition
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

- 先抢到 prevention 这个锁之后，代码保证了在抢锁的过程中，不会有不合时宜的线程切 换，从而避免了死锁。

Prevention (No Preemption)

- 在调用 unlock 之前，都认为锁是被占有的，多个抢锁操作通常会带来麻烦，因为我们 等待一个锁时，同时持有另一个锁。很多线程库提供更为灵活的接口来避免这种情况。具 体来说，trylock()函数会尝试获得锁，或者返回−1，表示锁已经被占有。你可以稍后重试 一下

```
top:
  lock(L1);
  if (trylock(L2) == -1) {
    unlock(L1);
    goto top;
}
```

- It could be used as follows to build a deadlock-free, ordering-robust lock acquisition protocol:

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

- 另一个线程可以使用相同的加锁方式，但是不同的加锁顺序（L2 然后 L1），程序仍然不会产生死锁

- 新的问题：活锁（livelock）。两个线程有可能一直重复这一序列，又同时都抢锁失败。这种情况下，系统一直在运行这段代码（因此不是死锁），但是又不会有进展，因此名为活锁。

- 解决方法：可以在循环结束的时候，先随机等待一个时间，然后再重复整个动作，这样可以降低线程之间的重复互相干扰。

Prevention (Mutual Exclusion)

- The final prevention technique would be to **avoid the need for mutual exclusion** at all. What can we do?
- 通过强大的硬件指令，构造出不需要锁的数据结构.

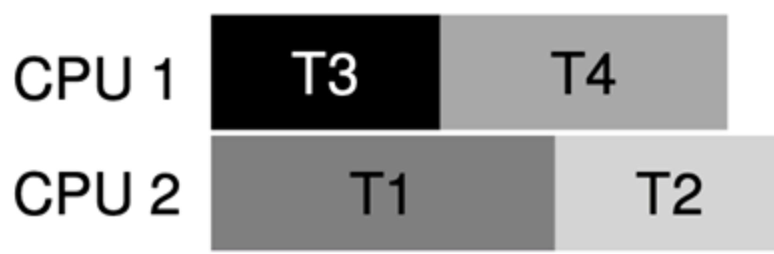
Deadlock Avoidance via Scheduling

- 除了死锁预防，某些场景更适合死锁避免（avoidance）。我们需要了解全局的信息，包括不同线程在运行中对锁的需求情况，从而使得后续的调度能够避免产生死锁。

假设我们需要在两个处理器上调度 4 个线程。假设我们知道线程 1（T1）需要用锁 L1 和 L2，T2 也需要抢 L1 和 L2，T3 只需要 L2，T4 不需要锁。

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

- 一种比较聪明的调度方式是，只要 T1 和 T2 不同时运行，就不会产生死锁



- T3 和 T1 重叠，或者和 T2 重叠都是可以的。虽然 T3 会抢占锁 L2，但是由于它只用到一把锁，和其他线程并发执行都不会产生死锁。

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

- 线程 T1、T2 和 T3 执行过程中，都需要持有锁 L1 和 L2。下面是一种不会产 生死锁的可行方案：

- T1、T2 和 T3 运行在同一个处理器上，这种保守的静态方案会明显增加 完成任务的总时间。尽管有可能并发运行这些任务，但为了避免死锁，我们没有这样做， 付出了性能的代价。

通过调度来避免死锁不是广泛使用的通用方案

Safe State

- 当进程申请一个有效的资源的时候，系统必须确定分配后是安全的。
- 如果存在一个安全序列，系统处于安全态。
- 对于进程序列< P1, P2, ···, Pn >，如果每个Pi 对资源的请求都能够由当前有效地资源加上Pj（j < i）释放的资源来满足，那么这个序列是安全的。在这种情形下，如果Pi 的资源需求不能够立即得到满足，那么它可以等到Pj 结束。当它们结束时，Pi 可以获得所需的全部资源，完成指定的工作，返回获取的资源并结束运行。

银行家算法

最有代表性的避免死锁算法，由Dijkstra提出。

1、银行家算法中的数据结构

资源向量Resource =(10, 5, 7)

- 可利用资源向量Available=(3, 3, 2)。它是一个含有m个元素的数组，其中每个元素代表一类可利用资源的数目。
- 如：

	A	B	C
Resource	10	5	7
Available	3	3	2

银行家算法

- 最大需求矩阵Max n*m矩阵，表示n个进程的每一个对m类资源的最大需求。

MAX	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

银行家算法

- 分配矩阵Allocation　n*m矩阵，表示每个进程已分配的资源数。

Allocation	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	1	3	2
P4	0	0	2

银行家算法

- 需求矩阵Need　n*m矩阵，表示每个进程还需要各类资源数。Need=Max-Allocation

Need	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

安全性检查算法

为进行安全性检查，定义数据结构：

```
Work:ARRAY[0..m-1] of integer;
Finish:ARRAY[0..n-1] of Boolean;
```

m代表资源的数量，n代表进程的数量

(1) Work:=Available;
    Finish:=false;

(2) 寻找满足下列条件的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

向量比较，Need[i]为行向量，Need[i]<=Work表示Need[i]的每一个元素都小于等于Work的对应元素

尝试着把进程执行完毕后的资源情况计算出来。即会有多少资源可用。

Need[i]≤Work

$$Max = \begin{Bmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{Bmatrix}$$

Available = (3, 3, 2)

资源\进程	Work(当前可用)			Need			Allocation			Work:=Work+Allocation[i]			Possible
	A	B	C	A	B	C	A	B	C	A	B	C	
P_0	7	4	3	7	4	3	0	1	0	7	5	3	T
P_1	3	3	2	1	2	2	2	0	0	5	3	2	T
P_2	7	5	3	6	0	0	3	0	2	10	5	5	T
P_3	5	3	2	0	1	1	2	1	1	7	4	3	T
P_4	10	5	5	4	3	1	0	0	2	10	5	7	T

Resource = (10, 5, 7)

银行家算法（资源分配拒绝法）

（1）如果Request[i]≤Need[i]，跳到第2步，否则出错；

（2）如果Request[i]≤Available，跳到第3步,否则挂起进程等待；

（3）根据进程的资源请求,先把资源试探性分配给它。

（4）执行安全性检查算法，如果安全状态则承认试分配，否则抛弃试分配，进程Pi等待。

（5）假定系统当前满足申请条件：则按以下方式修改状态

Available= Available- Request[i];

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

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

银行家算法实例

Resource（10,5,7）已用（7,2,5）

	Allocation			Max			Need			Available		
	已分配矩阵			最大需求矩阵			尚需资源矩阵			系统还剩可用资源数		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

银行家算法实例

T0时刻的安全序列$\{P_1, P_3, P_4, P_2, P_0\}$。

	当前可用资源			自身尚需求量			已分配			当前可用＋已分配			
	A	B	C	A	B	C	A	B	C	A	B	C	TRUE
P1	3	3	2	1	2	2	2	0	0	5	3	2	TRUE
P3	5	3	2	0	1	1	2	1	1	7	4	3	TRUE
P4	7	4	3	4	3	1	0	0	2	7	4	5	TRUE
P2	7	4	5	6	0	0	3	0	2	10	4	7	TRUE
P0	10	4	7	7	4	3	0	1	0	10	5	7	TRUE

（2) P1请求资源

进程P1申请资源request1=(1,0,2)

检查request1 (1,0,2) ≤ Need[i]

request1 (1,0,2) ≤ Available(3,3,2)

比较结果满足条件,试分配,得到新状态:

检查申请量是否超过需要量

检查申请量是否超过可分配量

process	Allocation A	Allocation B	Allocation C	Need A	Need B	Need C	Available A	Available B	Available 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			

安全性检查

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{P1,P3,P4,P0,P2}能满足安全性条件,可正式把资源分配给进程P1;

	当前可用资源			自身尚需求量			已分配			当前可用＋已分配			
	A	B	C	A	B	C	A	B	C	A	B	C	TRUE
P1	2	3	0	0	2	0	3	0	2	5	3	2	TRUE
P3	5	3	2	0	1	1	2	1	1	7	4	3	TRUE
P4	7	4	3	4	3	1	0	0	2	7	4	5	TRUE
P0	7	4	5	7	4	3	0	1	0	7	5	5	TRUE
P2	7	5	5	6	0	0	3	0	2	10	5	7	TRUE

（3）进程P4请求资源

进程P4请求资源(3,3,0),
Request (3,3,0)≤Need[4]
Request (3,3,0)>Available (2,3,0)
由于可用资源不足,申请被系统拒绝。

process	Allocation	Need	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	

（4）进程P0请求资源

进程**P0**发出资源申请，系统按银行家算法进行检查：

Request（0,2,0)≤ need（0）（7,3,1)

Request（0,2,0) ≤ Available（2,3,0)

process	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 1 0	7 3 1	2 3 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	

- 系统先为P0作尝试性分配，修改对应数据，得到一下中间结果：

process	Allocation	Need	Available
	A B C	A B C	A B C
P0	0 3 0	7 1 1	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	

- 利用安全性算法检查发现：系统能满足进程P0的资源请求(0,2,0)，但可以看出剩余资源已不能满足任何进程需求，故系统已处于不安全状态，故不能为进程P0分配资源。

银行家算法的缺点

- 进程很难在运行前知道其所需资源的最大值。
- 系统中各进程之间必须是无关的，即没有同步要求，无法处理有同步关系的进程。
- 进程的数量和资源的数目是固定不变的，无法处理进程数量和资源数目动态变化的情况。

Detect and Recover

- 允许死锁偶尔发生，检查到死锁时再采取行动。举个例子， 如果一个操作系统一年死机一次，你会重启系统，然后愉快地（或者生气地）继续工作。 如果死锁很少见，这种不是办法的办法也是很实用的.
- 很多数据库系统使用了死锁检测和恢复技术。死锁检测器会定期运行，通过构建资源 图来检查循环。当循环（死锁）发生时，系统需要重启。如果还需要更复杂的数据结构相 关的修复，那么需要人工参与.

作业

1、设系统中有三种类型的资源（A，B，C）和五个进程（P1，P2，P3，P4，P5），A 资源的数量 17，B 资源的数量为 5，C 资源的数量为 20。在 T0 时刻系统状态如下表所示。系统采用银行家算法来避免死锁。请回答下列问题：

（1）T0 时刻是否为安全状态？若是，请给出安全序列。

（2）若进程 P4 请求资源（2，0，1），能否实现资源分配？为什么？

（3）在（2）的基础上，若进程 P1 请求资源（0，2，0），能否实现资源分配？为什么？

T0 时刻系统状态

进程	最大资源需求量			已分配资源量			系统剩余资源数量		
	A	B	C	A	B	C	A	B	C
P1	5	5	9	2	1	2	2	3	3
P2	5	3	6	4	0	2			
P3	4	0	11	4	0	5			
P4	4	2	5	2	0	4			
P5	4	2	4	3	1	4			

End