

# EECE7376: 操作系统 接口与实现

并发性



## 回顾:进程及其状态

### ❖ 操作系统加载器为操作系统创建一个进程

程序从磁盘加载到内存地址空间并将其标记为“就绪”。

当操作系统调度程序将进程移至“运行”状态时,处理器程序计数器 (PC)

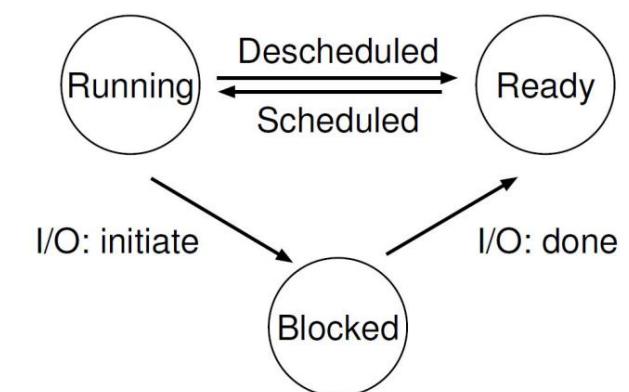
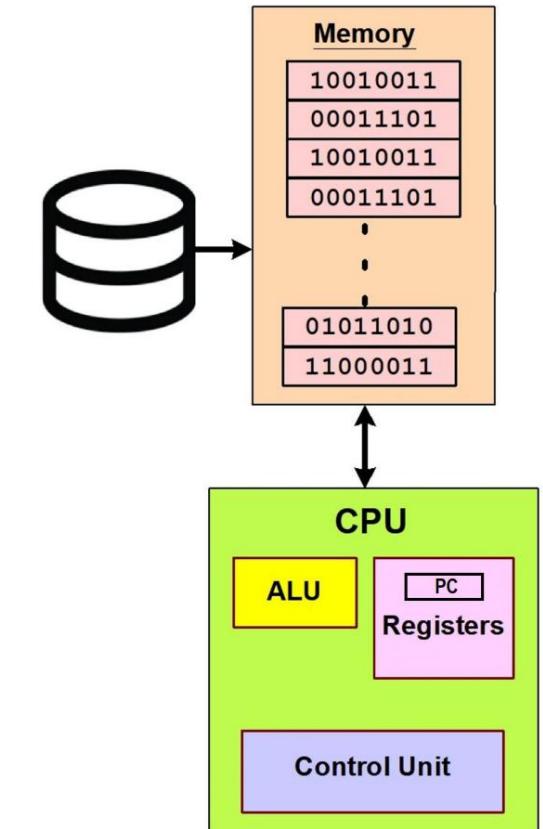
将更新为指向进程第一条指令的地址。

### ❖ 进程可以处于以下三种状态之一:

Ø **运行**:处理器正在执行进程  
指示。

Ø **Ready** (xv6的Runnable) :在就绪状态下,一个  
进程已准备好运行,但由于某种原因,操作系统选择在此时不运行它。

Ø **阻塞** (xv6的Sleep) :在阻塞状态下,一个  
进程执行了某种操作 (例如系统调用),导致其尚未准备好运行。





## 多线程程序

而不是我们经典的单点执行观点

在一个程序（即单个程序计数器 PC, 从中获取并执行指令）内，多线程程序具有多个执行点，每个执行点都从中获取和执行。

另一种思考方式是，每个线程非常像一个单独的进程，但有以下例外：

❖ 线程共享相同的地址空间，因此可以访问相同的数据。

一个线程崩溃将会导致整个进程崩溃。



## 为什么使用线程?

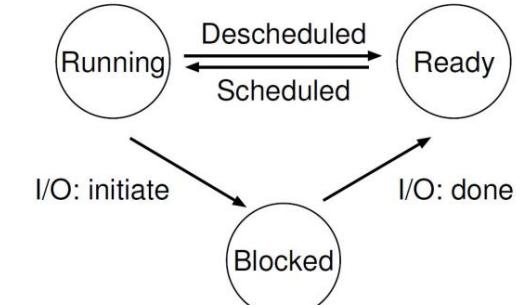
- ❖ 并行性:这是转变你的标准的任务

将单线程程序转换为通过多线程执行此类工作的程序,以使程序在现代硬件(例如多核处理器)上运行得更快。

- ❖ 重叠:使用线程是避免卡住的自然方法;当程序中的一个线程等待(即被阻塞等待I/O)时,CPU调度程序可以切换到其他线程,这些线程已准备好运行并执行一些有用的操作。

线程允许I/O与单个程序中的其他活动重叠。

- ❖ 使用多进程来实现上述功能,缺乏线程共享地址空间从而方便共享数据的优点。





## 简单的线程创建代码

❖ 从以下位置下载t0.c代码：

<https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-intro>

<https://github.com/remzi-arpacidusseau/ostep-code/tree/master/include>

❖ 使用 `gcc -pthread t0.c -o t0` 编译并运行。

观察： 代码创建了两个

线程 p1 和 p2，每个线程运行函数 `mythread()` 来独立打印 “A” 和 “B”。 ❖ 创建两个线程 p1 和 p2 后，主线程调用

`Pthread_join()`，等待特定线程完成。

总的来说，这次运行期间使用了三个线程：主线程、

p1 线程和 p2 线程。

我们甚至可以看到 “B” 打印在 “A” 之前！

❖ 结论：

❖ 线程不像调度程序那样常规函数调用

决定先运行 p2，即使 p1 是较早创建的；没有理由假设首先创建的线程将首先运行。



t0.c

```

void *mythread(void *arg)
{
    printf("%s\n", (char *) arg); 返回空值;
}

int main(int argc, char *argv[]) { if (argc != 1)
    { fprintf(stderr, 用法:
        main\n); 退出 (1) ;
    }
}

```

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

```

```

pthread_t p1, p2;
printf( 主要:开始\n);
pthread_create(&p1, NULL, mythread, A );
pthread_create(&p2, NULL, mythread, B ); // join 等待线程
完成

```

```

Pthread_join(p1, NULL);
Pthread_join(p2, NULL); printf( 主
要:结束\n); 返回0;
}

```



## 线程上下文切换

### ❖ 当从运行一个线程 T1 切换到运行另一个线程 T2 时

对于其他线程 T2, 必须进行上下文切换。在运行 T2 之前, 必须保存 T1 的寄存器状态并恢复 T2 的寄存器状态。

线程之间的上下文切换与进程之间的上下文切换非常相似, 但有以下区别:

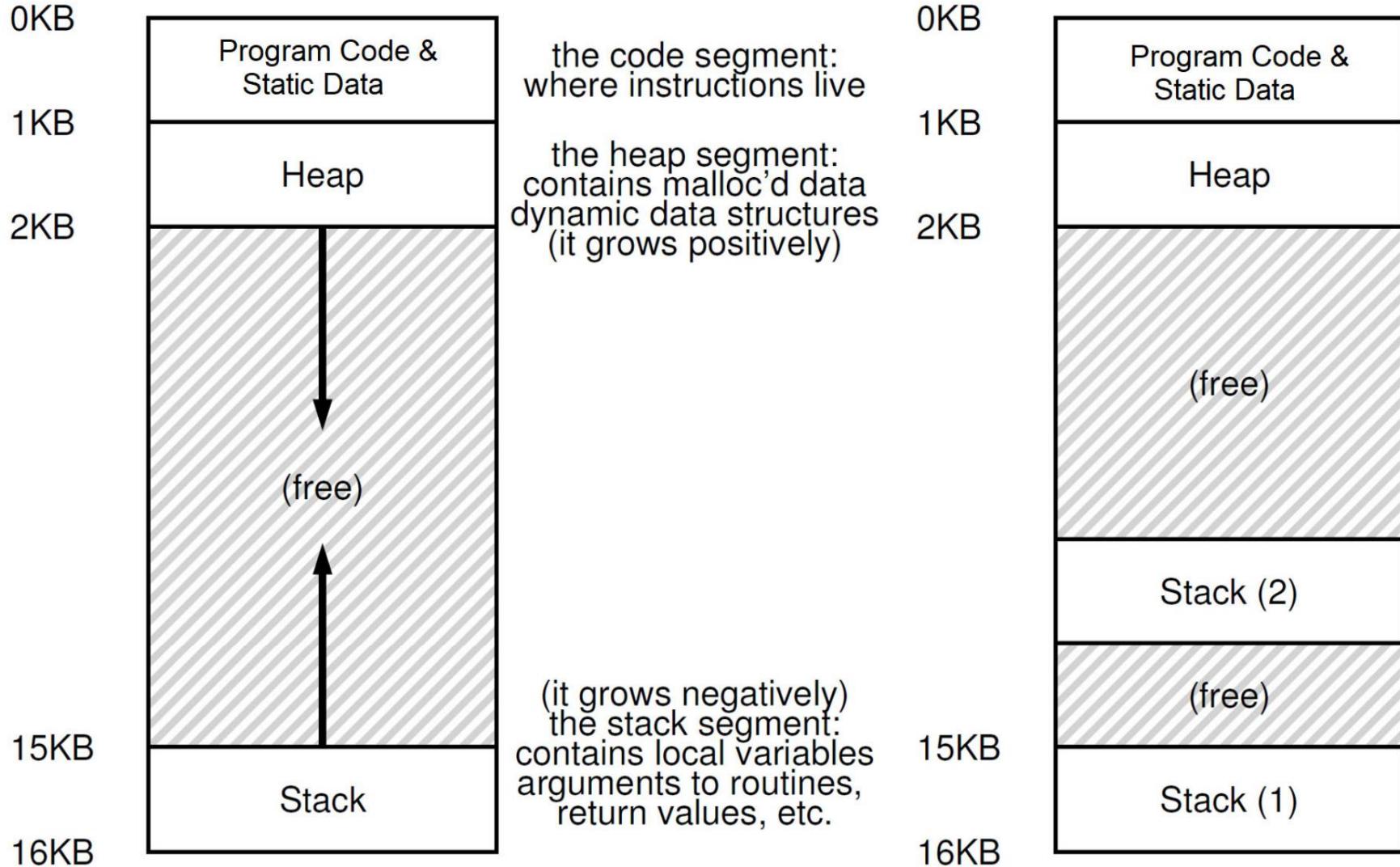
对于进程, 其当前状态 (例如寄存器值) 保存在称为进程控制块 (PCB) 的结构中; 现在, 我们需要一个或多个线程控制块 (TCB)。

### ❖ 地址空间保持不变 (即不需要切换我们正在使用的页表)。

地址空间中不再有单个堆栈, 而是每个线程都有一个堆栈, 因为每个线程可以独立调用各种例程。



## 多线程地址空间



## Single-Threaded And Multi-Threaded Address Spaces



## 具有共享数据的线程

下载t1.c代码,编译并运行。

<https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-intro>

❖ 观察：

两个线程试图将共享变量计数器加1

最大迭代次数的循环。

而不是总是收到期望的计数器最终结果

$2 \times \text{max}$ ,每次运行程序都会得到不同的结果!

❖ 结论：

计数器加1并不是处理器执行的一条指令,而是RISC-V中的一系列指令,如下所示:

```
lw x9, 4(x6) //从内存中的地址加载计数器
add x9, x9, 1 sw      //加1
x9, 4(x6) //存储回更新后的值
```

❖ 如果线程在“存储回”更新后的内容之前进行上下文切换

计数器,另一个线程将获得更新前计数器的相同值。这将导致即使在执行两个添加指令之后两个线程也存储回相同的结果



## t1.c (2 中的 1)

---

```
#include <stdio.h>
#include <stdlib.h>

#include <pthread.h>
#include "common.h"
#include "common_threads.h"
```

整数最大值；

易失性 int 计数器 = 0; // 共享全局变量

```
void *mythread(void *arg) { char *letter =
    arg; 整数我; // 堆栈 (每个线
程私有) printf( %s: begin [addr of
    i: %p]\n , letter, &i); for (i = 0; i < max; i++) 计数器 = 计数器 + 1; // 共享:只有一个
    printf( %s: done\n , letter); 返回空值;
}
```



## t1.c (2 / 2)

```
int main(int argc, char *argv[]) { if (argc != 2)
{ fprintf(stderr, 用法:
main-first <loopcount>\n );退出 (1) ; }
```

最大值= atoi(argv[1]); pthread\_t  
 p1, p2; printf( main:开始  
 [counter = %d] [%x]\n , counter, (unsigned int) &counter);

pthread\_create(&p1, NULL, mythread, A );
 pthread\_create(&p2, NULL, mythread, B ); // join 等待线程完成

Pthread\_join(p1, NULL);
 Pthread\_join(p2, NULL); printf( 主  
 要:完成\n [计数器: %d]\n [应该: %d]\n ,  
 计数器,最大\*2);返回0;

}



## 竞争条件和临界区

- ❖ 我们在前面的示例中演示的称为竞争条件（或数据竞争）,其中结果取决于代码的执行时间。
- ❖ 我们称这个结果为不确定的,而不是我们习惯于从计算机中进行的良好的确定性计算。

因为多个线程执行计数器递增代码

可能会导致竞争条件,我们将此代码称为关键部分。

临界区是访问共享变量（或更一般地说,共享资源)的一段代码,并且不能由多个线程同时执行。

如果多个线程大致同时进入临界区,就会出现竞争条件;两者都尝试更新共享数据结构,导致令人惊讶的（也许是不受欢迎的结果)结果。



## 检测竞争条件的工具

❖ Valgrind是一个用于调试和分析 Linux 程序的系统。

借助其工具套件,您可以自动检测许多内存管理和线程错误。资源:

❖ <https://valgrind.org/docs/manual/quick-start.html>

❖ <https://valgrind.org/docs/manual/drd-manual.html>

通过在终端中输入以下命令来安装 valgrind:

```
$sudo apt 更新
```

```
$sudo apt install valgrind
```

确保使用-g选项编译您的程序以包含

调试信息,允许将源代码行号包含在 valgrind 报告中。

```
$gcc -g -pthread t1.c -o t1
```

使用可执行文件t1及其参数,运行以下命令让工具检查竞争条件:

```
$valgrind --tool=helgrind ./t1 100000
```



## 互斥和原子性

- ❖ 为了解决竞争条件,我们需要关键部分  
我们所说的互斥。

互斥保证如果一个线程在临界区内执行,其他线程将被阻止执行。

在此上下文中,原子性意味着“作为一个单元”,即  
有时我们认为“要么全有,要么全无”。

- ❖ 我们想要的是执行三指令序列

原子地:

长x9, 4(x6)  
添加 x9, x9, 1  
sw x9, 4(x6)

- ❖ 在存在中断的情况下我们如何做到这一点?



## 锁:基本思想

锁可用于在关键代码上提供互斥

部分如下例所示：

```
pthread_mutex_t 互斥体 = PTHREAD_MUTEX_INITIALIZER;  
...  
pthread_mutex_lock(&mutex);  
计数器 = 计数器 + 1;  
pthread_mutex_unlock(&mutex);
```

锁只是一个变量,比如上面的互斥体。 程序员应该为每个关键部分分配一个专用的锁变量 (这将有助于增加线程之间的并发性)。

该锁保持任意时刻的锁状态。 ♦♦它要么是可用的 (解锁、自由) ,要么是获得的 (锁定、持有)。

♦♦在任何时候,只有一个线程持有锁并且大概线程处于临界区。



## lock() 和unlock() 例程

调用例程[lock\(\)](#)尝试获取锁;如果没有其他线程持有锁（即空闲）,则该线程将获取锁并进入临界区;该线程有时被称为锁的所有者。

❖如果另一个线程随后在同一个锁上调用[lock\(\)](#)

变量（[在我们的示例中为互斥体](#)）,当锁被另一个线程持有时它不会返回;这样,当第一个持有锁的线程位于临界区时,其他线程就被阻止进入临界区。

一旦锁的所有者调用[unlock\(\)](#)并且有等待线程（卡在[lock\(\)](#)中）,其中一个线程将（最终）注意到（或被告知）锁状态的这一变化,获取锁,并进入临界区。



## 建锁的标准

- ❖ 锁的有效实现涉及所有三个级别

计算系统的：

1. 硬件
2. 操作系统
3. 用户级库软件

- ❖ 评价一把锁是否好用的基本标准：

互斥: 锁是否阻止多个线程进入临界区？

公平性: 当任何线程争用锁时是否会挨饿  
这样做，从而永远无法获得它？

性能: 使用  
锁？



## 通过控制中断来锁定

建立锁的一种方法是禁用关键中断

部分（即，在锁定时禁用中断并在解锁时重新启用它们）。

通过在进入临界区之前关闭中断（使用特殊的硬件指令），我们可以确保临界区内的代码不会被中断，从而像原子一样执行。

❖ 这种方法的优点是它的简单性。 ❖ 缺点是：

❖ 信任用户程序执行特权操作。恶意的

程序可以调用`lock()`并进入无限循环。除非我们重新启动操作系统，否则它永远不会重新获得对系统的控制。

它不适用于多处理器，因为线程可以在多处理器上运行其他处理器。

❖ 长时间关闭中断可能会导致丢失

导致严重系统问题的重要中断（例如，CPU 错过了磁盘设备已完成读取请求的事实）。



## 通过测试和设置锁定:硬件

- ◆ 这里硬件提供了允许原子操作的指令  
交换 (读/写)内存操作。

在 RISC-V 中,提供了以下两条指令:

负载保留: `LR.W rd,(rs1)`

存储条件: `SC.W rd,(rs1),rs2`

两条指令中的内存地址必须相同 (通常  
它是互斥锁地址)。

`LR.W`从`rs1`中的内存地址加载一个字到`rd`,并在该内存地址上进行“保留”(一种方法是将  
该地址存储在特殊寄存器中)。

- ◆ `SC.W`尝试将`rs2`内容存储到 `rs1` 中的地址。

Ø 如果内存地址上仍然存在有效的保留,则成功,使该保留无效,将`rs2`内容存储在内存地址中,  
并在`rd`中返回0。

Ø 如果位置有无效的预留并在`rd`中返回非零值,则失败



## RISC-V 锁定和解锁代码

❖ Lock:假设互斥量的地址在寄存器x20中传递，并且它是  
初始化为零表示解锁状态。

注意:在 RISC-V 寄存器中,x0 的值始终为零

addi x12,x0,1 Again:

// x12 的锁定值是 1

lr.w x10,(x20) bne x10,x0,again

// 读取互斥锁并对其进行“保留”

sc.w x11,(x20),x12 // 尝试将

//继续尝试,直到为0 (即解锁)

1 存储到互斥锁以锁定它。

// 如果互斥锁仍然有有效的保留 成功并在 x11 中存储零

bne x11,x0,again // 如果

//如果失败 (即 x11 不为零) ,请重试。

另一个线程击败了我们,就会发生这种情况!

注意: sc.w 不能用常规的 sw 指令替换,为什么?

解锁:仅由持有锁的进程使用 sw x0,0(x20) // 释放锁

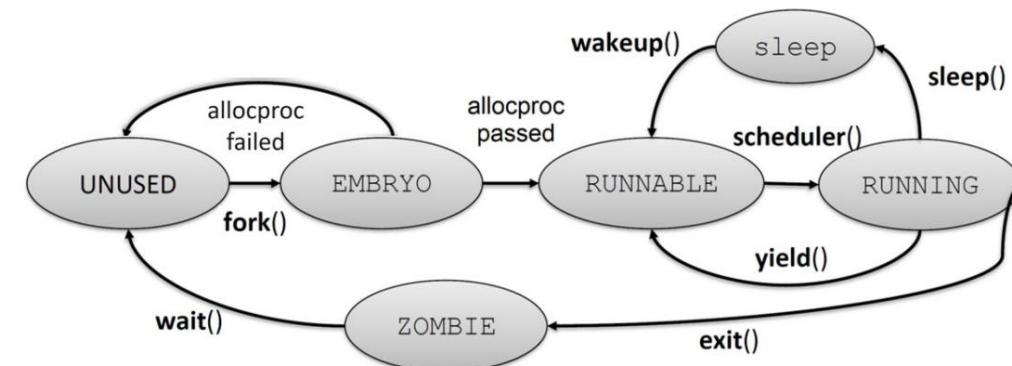


## 自旋锁

**所**提出的测试和设置锁通常称为自旋锁,因为它只是使用 CPU 周期进行自旋,直到锁可用。

- ❖为了在单处理器上正常工作,需要一个抢占式调度程序 (即通过计时器中断线程,以便不时运行不同的线程)。

**如果**没有抢占,自旋锁在单个 CPU 上没有多大意义,因为在 CPU 上旋转的线程永远不会放弃它。



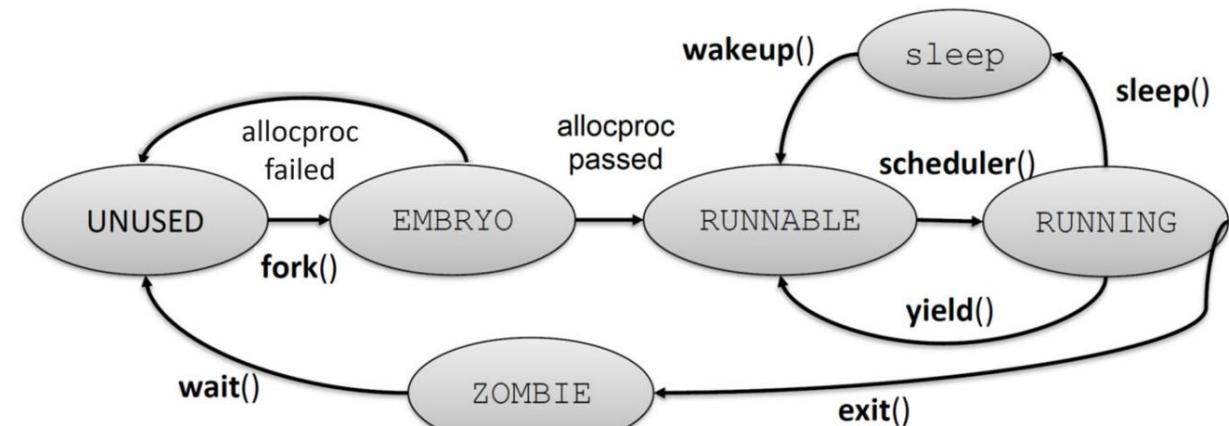


## 自旋锁公平性和性能

到目前为止所讨论的简单自旋锁是不公平的,并且可能导致饥饿。

❖ 在单CPU情况下,性能开销可以是  
相当痛苦。

想象一下持有锁的线程是  
在关键部分内被抢占。然后调度程序可能会运行所有其他等待线程。这  
些线程在放弃 CPU 之前会旋转一个时间片的持续时间,这会浪费 CPU 周  
期。





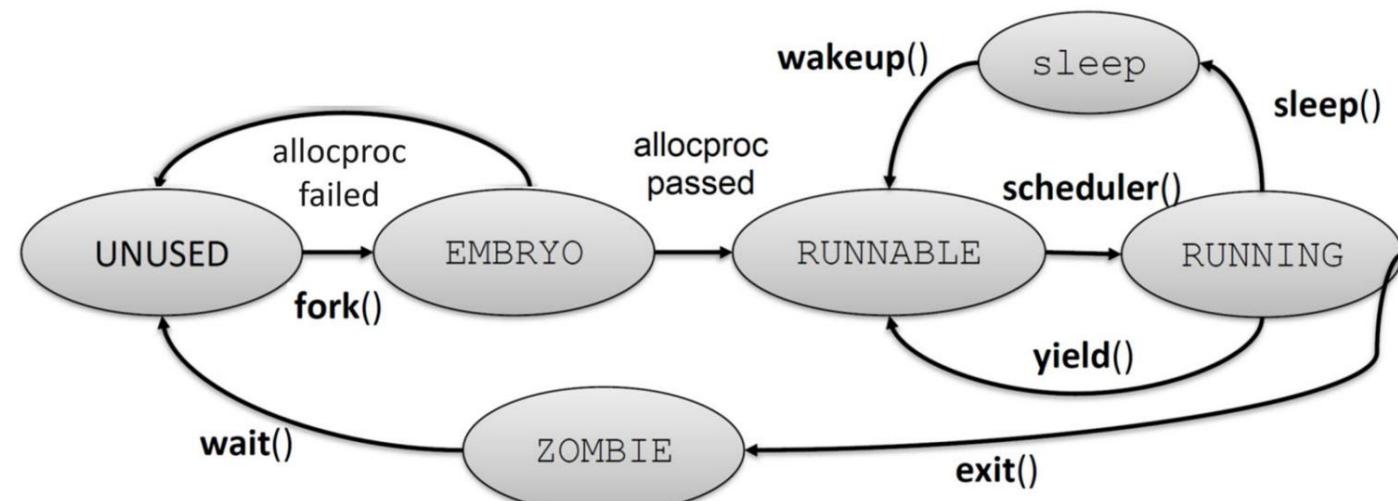
## 操作系统原始产量() (1/2)

硬件通过工作锁来支持我们。我们还有一个

问题:当临界区发生上下文切换,线程开始无休止地旋转,等待被中断(持有锁)的线程再次运行时,该怎么办?

一种解决方案是操作系统提供一个原始的yield()

当一个线程发现另一个线程持有锁时,它允许线程放弃CPU并返回到Runnable状态。





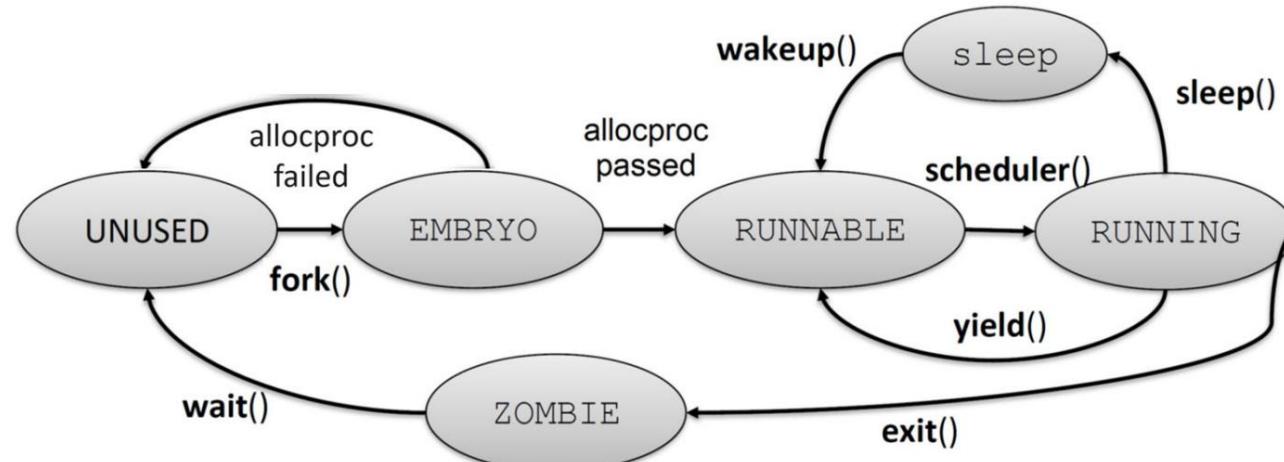
## 操作系统原始产量() (2/2)

❖ Yield 只是一个系统调用,它将调用者从运行状态立即变为就绪 (xv6 的可运行)状态 (而不是在一个时间片的持续时间内旋转) ,从而促进另一个线程运行。因此,产生的

线程本质上是自行取消调度。

尽管比旋转方法更好,但这种方法仍然成本较高;上下文切换的成本可能很大。

❖而且,它并没有解决饥饿问题。





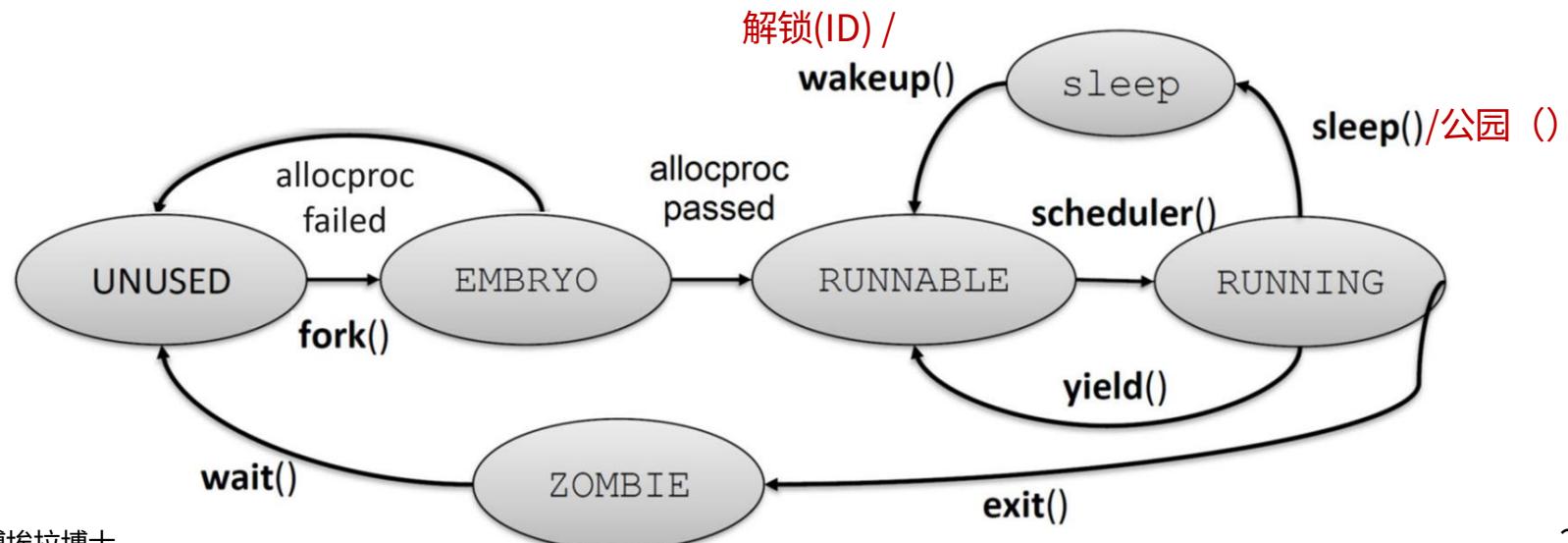
## 排队睡觉 (1/2)

在我们到目前为止研究的方法中,调度程序决定下一个运行哪个线程,它可能会做出以下错误的选择:

运行一个必须自旋等待锁 (我们的第一种方法)或立即让出CPU (我们的第二种方法)的线程。

❖ Solaris操作系统提供了两个原语:

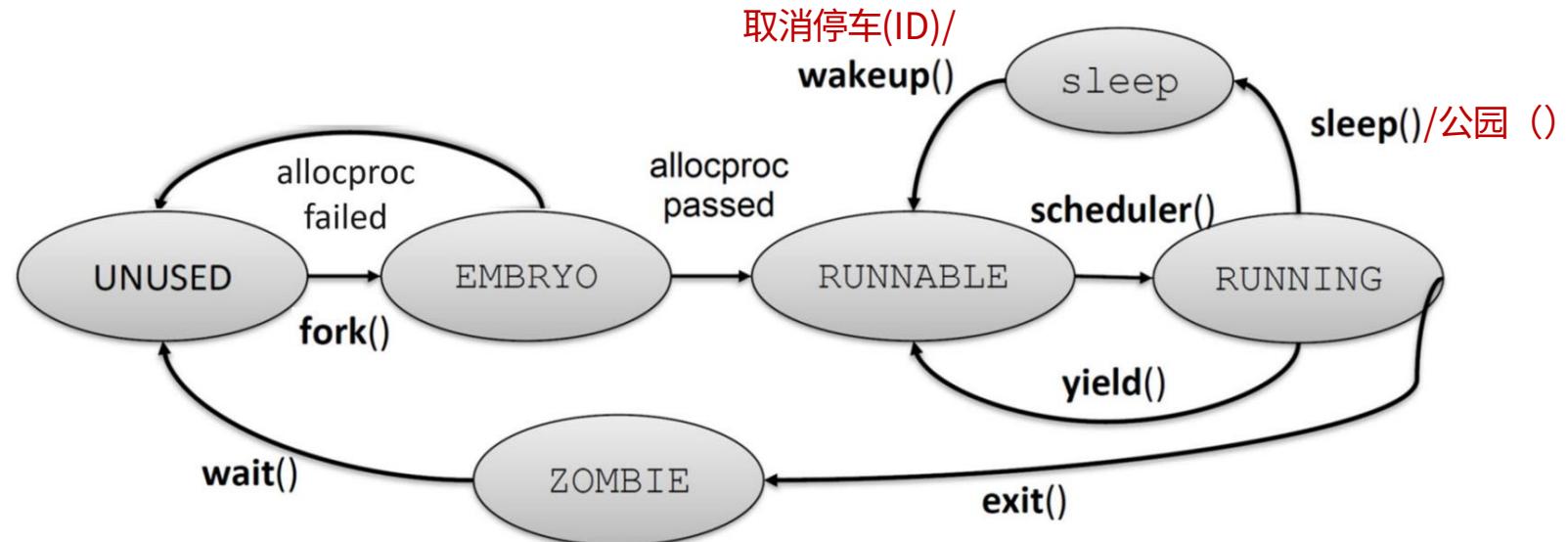
`park()`将调用线程置于睡眠状态 (而不是像`yield`那样准备好) , `unpark` (`threadID`)唤醒特定线程。





## 排队睡觉 (2/2)

- ❖ 通过 park 和 unpark 原语, 可以利用锁等待者队列 (已“停放”的线程) 来创建更高效的锁。队列有助于控制谁接下来获得锁 (“解锁”), 从而避免饥饿。
  
  
  
  
  
- ❖ 当一个线程调用 lock 并且锁不可用时,  
添加到队列并停放。  
当一个线程调用 unlock 时, 一个已停放的线程将被取消停放至 “测试和设置” 操作  
以获取锁。





## 线程不安全与线程安全计数器

不安全

```
typedef struct __counter_t {int值;}  
counter_t;  
  
void init(counter_t * c) {c->value  
= 0;}  
  
无效增量(counter_t * c) {c->value++;}  
  
void decrement(counter_t * c) {c-  
>value--;}  
  
int get(counter_t * c) {return  
c->value;}
```

typedef struct \_\_counter\_t { int值;

pthread\_mutex\_t 锁; }  
counter\_t;

void init(counter\_t \* c){//在创建任何线程之前调用c->value = 0 ;

pthread\_mutex\_init(& c->锁 ,无效的) ;}无

效增量(counter\_t \* c) {

pthread\_mutex\_lock(& c->lock); c->值++;

pthread\_mutex\_unlock(& c->lock); }

无效减量 (counter\_t \* c){

pthread\_mutex\_lock(& c->lock); c->值--;

pthread\_mutex\_unlock(& c->lock); }

int get(counter\_t \* c) {

pthread\_mutex\_lock(& c->lock); int rc = c-  
>值;

pthread\_mutex\_unlock(& c->lock);返回rc; }

安全的



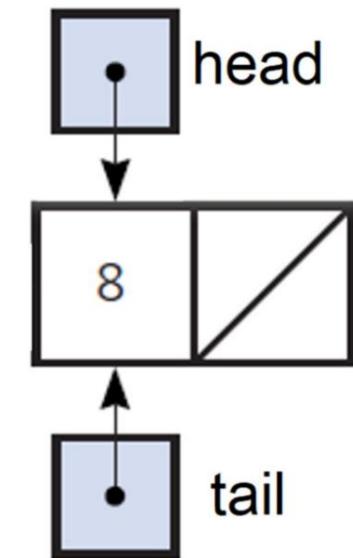
## 单锁与多锁

对所有关键部分使用单个锁具有性能问题。

如果你的数据结构太慢,你要做的不仅仅是添加一个锁。

❖例如,对于队列数据结构,我们可以使用两个锁来实现入队和出队操作的并发性。

然而,使单节点队列出队或入队到空队列将需要同时更新 tail 和 head !



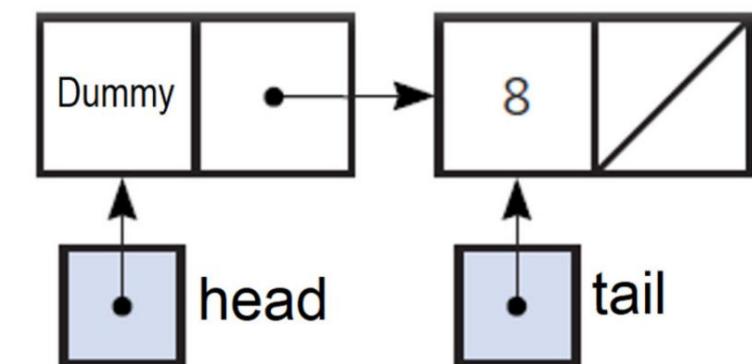
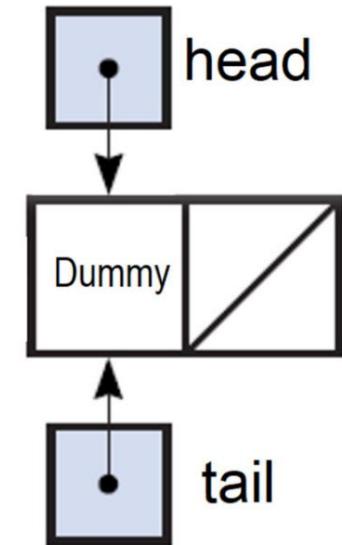


## Michael & Scott 队列算法

Michael 和 Scott 使用的一个技巧是添加一个虚拟节点（在队列初始化代码中分配）；其中 head 始终指向该虚拟节点。

虚拟节点使得头操作和尾操作分离，因为使单节点队列出队只需要更新头，而入队到空队列只需要更新尾。

以下几页中的代码使用两个锁实现了一个线程安全的队列数据结构，一个用于队列头，一个用于队列尾。

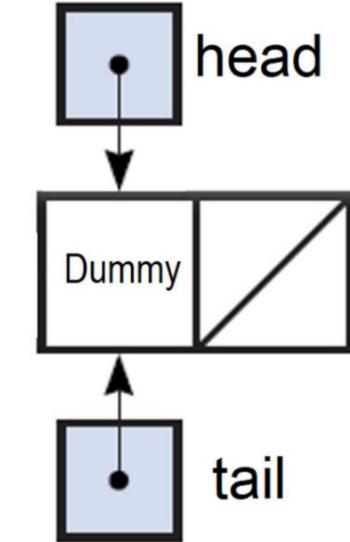




## Michael & Scott 并发队列(1/3)

```
typedef struct __node_t { int值;结构
    __node_t
    *next ; }节点_t;
```

```
typedef struct __queue_t { node_t
    *head ;节点_t *尾;
    pthread_mutex_t
    head_lock , tail_lock ; }队列_t;
```



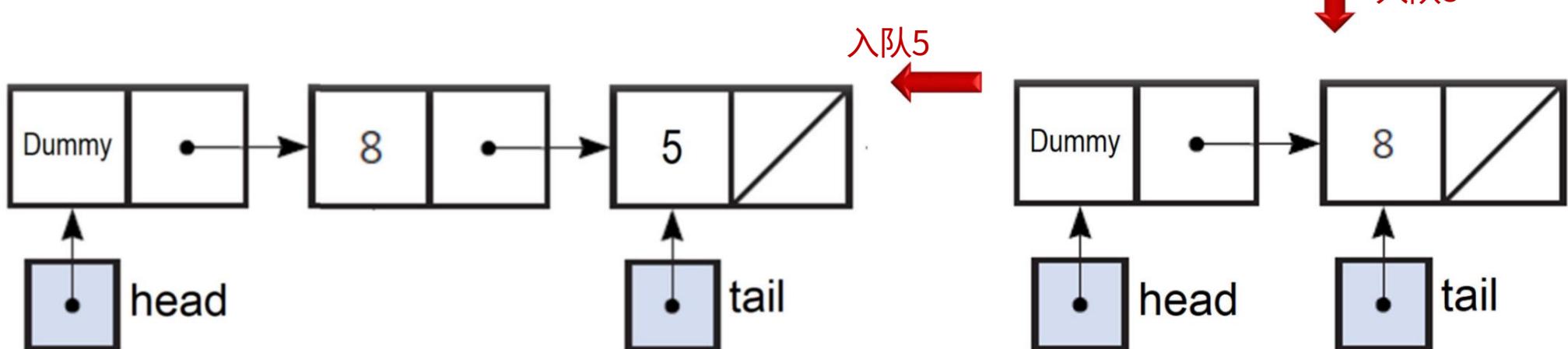
void Queue\_Init (queue\_t \* q) { //创建虚拟

节点,以便将“尾部”更新与“头部”更新分开。 node\_t \*tmp = malloc (sizeof (node\_t)); tmp->下一个= NULL; //虚拟节点的next=NULL,不需要设置其值属性q->head = q->tail = tmp ; pthread\_mutex\_init(& q->head\_lock , NULL); pthread\_mutex\_init(& q->tail\_lock , NULL); }



## Michael & Scott 并发队列(2/3)

```
// 在尾部添加一个新节点,内容为“value” void Queue_Enqueue
(queue_t * int value) { q , node_t *tmp = malloc (sizeof (node_t)); 断言 (tmp !=
NULL) ; tmp->值=值; tmp->下一个=NULL; pthread_mutex_lock(&
q->tail_lock); q->尾->下一个=tmp; q-
>tail = tmp ;// tail 始终指向最后一个入队节点或 // 虚拟节点 (如果队列为
空) pthread_mutex_unlock(& q->tail_lock); }
```





## Michael & Scott 并发队列(3/3)

// 删除队列中的第一个节点 (如果存在)并在 “value”参数中返回其值。

// 该函数成功时返回 0,失败时返回 -1。 int Queue\_Dequeue (queue\_t

```
* q , pthread_mutex_lock(& q->head_lock); node_t 整数*值) {
```

```
*tmp = q->head ; node_t *new_head = tmp->next ; if (new_head
```

```
== NULL) { //表示空队列,只有虚拟节点
```

pthread\_mutex\_unlock(& q->head\_lock); \*值= - 1; 返回-

1; \*值= new\_head-

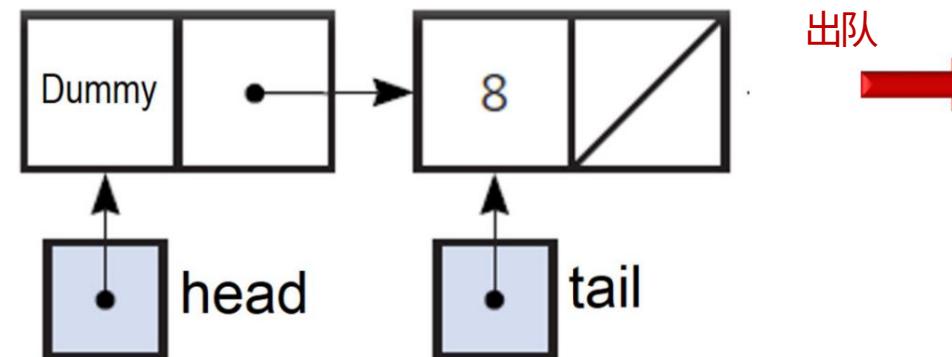
>值; q->head = }

new\_head; //head总是指向一个虚拟节

点, //它要么是原始虚拟节点,要么是最近出列的节点pthread\_mutex\_unlock(&

q->head\_lock);免费 (tmp) ; // 旧的虚拟节点被删除return 0; }

为什么没有显式删除  
值为 8 的节点?





## 有条件的并发

锁并不是构建过程中唯一需要的原语

并发程序。

在很多情况下,线程希望在继续执行之前检查条件是否为真。

例如,父线程可能希望在继续之前检查子线程是否已完成 (这通常通过调用[join\(\)](#)来完成) ; ♦这样的等待应该如何实现?

我们可以尝试使用共享变量,如示例所示:

而 (完成== 0) ; // 旋转

♦这个解决方案通常是有效的,但是作为父级,它的效率非常低  
旋转并浪费 CPU 时间。

相反,我们需要一种方法让父进程进入睡眠状态,直到它等待的条件 (例如,子进程完成执行)实现为止。



## 条件变量

条件变量是一个显式队列,线程在等待条件时可以将自己放入其中。

一些其他线程,当它改变所述条件时,可以唤醒那些等待线程中的一个 (或多个) 。

❖要声明这样一个条件变量,只需编写

像这样的东西: `pthread_cond_t c`将`c`声明为条件变量

❖条件变量有两个与之相关的操作: `wait()`和`signal()`。 o当线程希望让自己进入睡眠状态时,

将执行`wait ()`调用; o当线程更改

了条件中的某些内容并因此想要唤醒等待此条件 (队列)的睡眠线程时,将执行`signal()`调用。



## 条件变量实现 (1/2)

```
int 完成 = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER; pthread_cond_t
c = PTHREAD_COND_INITIALIZER;
```

```
无效 thr_exit () {
    pthread_mutex_lock(&m); 完成 = 1;
    pthread_cond_signal(&c);
    pthread_mutex_unlock(&m); }

void *child(void *arg) { printf
( " child\n" ); } thr_exit(); 返回空值; }
```

```
int main (int argc, char *argv[]) { printf (" 父级:开始
\n" ); pthread_t p;

pthread_create(&p,          无效的 ,          , 无效的 ); 孩子
thr_join(); printf
( " 父级:结束\n" ); return 0; }
```

无效 thr\_join() {  
 pthread\_mutex\_lock(&m); while  
 (done == 0) //while, 不是if, 所以当wait返回时, 再用一次while迭代来确认done为0  
 pthread\_cond\_wait(&c & m); //wait() 解锁m  
 pthread\_mutex\_unlock(&m); }

为什么需要锁?



## 条件变量实现 (2/2)

前面代码的一个运行场景是：

父线程创建子线程,但继续运行,因此立即调用[thr\\_join\(\)](#)等待子线程完成。

在这种情况下,它将获取锁,检查子进程是否完成 (没有完成) ,并通过调用[wait\(\)](#)让自己进入睡眠状态 (从而释放锁) 。

[子进程最终会运行,打印消息“child” ,并调用](#)  
[thr\\_exit\(\)](#)唤醒父线程;这段代码只是获取锁,设置状态变量done,并向父进程发出信号,从而唤醒它。

最后,父进程将运行 (从[wait\(\)](#)返回并持有锁) ,解锁锁,并打印最终消息“parent: end”。

❖ 使用锁对于避免儿童使用锁时发生争用情况非常重要

在父级检查[完成后](#)、调用[wait\(\)](#) 之前立即中断父级。然后子进程调用[thr\\_exit\(\)](#)并在没有线程等待时发出信号,因此没有线程被唤醒。稍后,当父进程运行[wait\(\)](#) 时,它将永远休眠。

如果子进程在父进程调用 lock 之前中断了父进程怎么办?



## 生产者/消费者问题 (2 中的 1)

❖ 生产者/消费者问题是另一个同步问题

涉及一个或多个生产者线程和一个或多个消费者线程的问题。

❖ 生产者生成数据项并将其放入缓冲区;消费者从缓冲区中抓取所述物品并以某种方式消费它们。

**示例:**在多线程 Web 服务器中,生产者将 HTTP 请求放入工作队列 (有界缓冲区) ,而消费者从该队列中取出请求并处理它们。



## 生产者/消费者问题 (2 of 2)

由于有界缓冲区是共享资源,所以我们必须  
需要同步访问它。

❖ 我们将使用以下方法解决生产者/消费者问题  
简单的整数队列缓冲区。

生产者将整数放入缓冲区中,除非它已经被填满。  
消费者从缓冲区获取整数,除非缓冲区为空。

由于速度在此类应用中至关重要,因此使用数组来实现有界缓冲区队列。

回想一下,在 Michael & Scott 并发队列实现中,我们使用在堆中动态创建的链表,因此不需要检查队列是否已满。

❖ 如何使用固定大小数组实现的队列执行入队和出队操作?

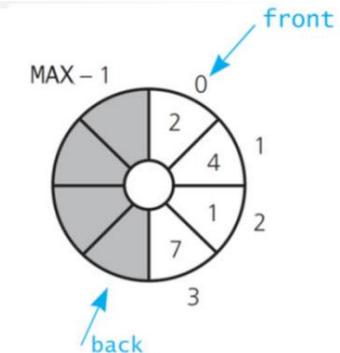


## Put 和 Get 例程

```
int 缓冲区[最大值]; //实现为循环队列 int back = 0; int 前面= 0; 整数  
计数= 0;
```

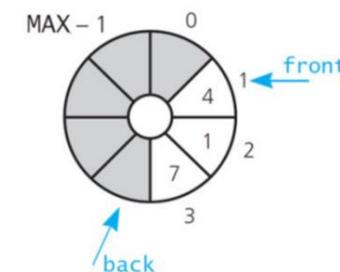
```
//在调用 put 之前检查缓冲区是否未满 void put(int  
value) { buffer[back] = value; 后退  
= (后退+ 1) % MAX; 计数++; }
```

得到 ()

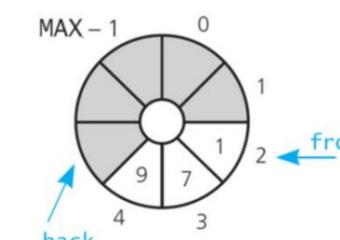
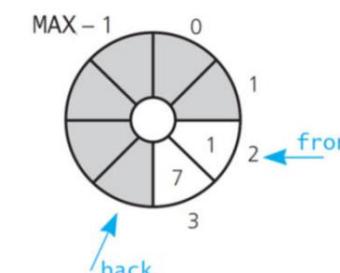


```
//在调用 get 之前检查缓冲区是否不为空 int get() { int  
tmp = buffer[front];  
前面= (前面+ 1) % MAX; 数数 - ; 返回  
tmp; }
```

得到 ()



放(9)





## 生产者/消费者同步 (1/2)

```
cond_t 不满,不空; mutex_t 互斥
体; void *生产者(int 循环) { int i; for (i = 0; i < 循环; i++) {
    {
        pthread_mutex_lock(&mutex); while
        (count == MAX) //缓冲区已满
            Pthread_cond_wait(&notfull, &mutex); 把(一世);
            Pthread_cond_signal(&notempty);
        pthread_mutex_unlock(&mutex); }
    }

void *consumer (int 循环) { int i, tmp;
    for (i = 0; i < 循
    环; i++) {
        pthread_mutex_lock(&mutex); while
        (count == 0) //缓冲区为空
            Pthread_cond_wait(&notempty, &mutex);
        tmp = get();           pthread_cond_signal(&notfull);
        pthread_mutex_unlock(&mutex); printf
        ( "%d\n" , tmp); }
    }
```



## 生产者/消费者同步 (2/2)

代码使用两个条件变量,以便在缓冲区状态发生变化时正确指示应唤醒哪种类型的线程。

- ❖ 生产者线程等待条件未满,并发出信号不为空。 ❖ 消费者线程等待notempty并发  
出notfull 信号。

该代码允许在睡眠前生成多个值,并且类似地在睡眠前消耗多个值。这减少了上下文切换。

它还利用互斥体 (锁)来允许并发生产或消费发生。



## 信号量

到目前为止,我们利用锁和条件变量来解决各种相关且有趣的并发问题。

Dijkstra和同事发明了信号量作为锁和条件变量的单一原语。

信号量是一个具有整数值的对象,我们可以  
通过一些例程进行操作。

信号量对象被定义为一个全局对象,它执行以下操作:

不属于任何特定线程。

- ❖ 信号量的初始值决定了它的行为。
- ❖ 因此,在调用任何例程与  
信号量,我们必须首先将其初始化为适合所需行为的值。



## POSIX 信号量例程

- ❖ 可移植操作系统接口( POSIX) 引入了以下信号量例程：

```
#include <semaphore.h> sem_t  
s;  
sem_init(& s      ,      , 1); // 将信号量s初始化为值1。第二个0  
                                // 参数设置为 0 表示信号量在同一进程中的线程之间共  
                                // 享。 sem_wait(& s); // 将信号量的值减一,如果  
信号量的值为负,则阻塞 (在//队列中等待) ,否则立即返回。  
  
sem_post(& s); // 将信号量的值增加1,并唤醒// 等待线程之一 (如果有) 。它立即返回。  
  
// 注意:信号量的绝对值,当为负时,等于等待线程的数量。信号量的用户通常看不到该值
```



## 信号量实现 (1/2)

以下例程必须以原子方式执行,以便两个进程不能同时在同一信号量上执行其中任何一个例程。

这可以使用与我们研究的用汇编代码实现互斥锁类似的方法来实现。

```
sem_wait(&s){ //必须以原子方式执行 (临界区)
```

```
    s->值--;
```

```
    if (s->值< 0)
```

```
        堵塞 () ;//调用进程被放入等待队列
```

```
}
```

```
sem_post(&s){ //必须以原子方式执行 (关键部分)
```

```
    s->值++;
```

```
    if (s->value >= 0) 唤醒(P); //
```

```
        从等待队列中删除一个进程
```

```
}
```



## 信号量实现 (2/2)

- ❖ 每个信号量都有一个关联的等待队列。**块-**  
放置调用 `sem_wait` 的进程  
适当的等待队列上的例程。
- ❖ 唤醒    删除正在等待的进程之一  
队列并将其放入就绪队列中。



## 信号量作为锁

要使用信号量作为锁,只需用`sem_wait()` / `sem_post()`对包围感兴趣的关键部分。♦通过下面的代码,临界区中只能运行一个线程。其他线程将等待,直到正在运行的线程调用`sem_post()`:

```
sem_t m; sem_init(&m, 0, 1); // 初始化为1
```

```
.....
```

```
sem_wait(&m); // 这  
里是临界区
```

```
sem_post(&m);
```

由于锁只有两种状态（持有和未持有）,所以我们有时将用作锁的信号量称为二元信号量。



## 信号量作为条件变量

❖这是使用信号量作为条件变量的示例代码,其中Parent等待其Child完成执行:

```
sem_t s; void
*child (void *arg) { printf( child\n );}
sem_post(& s); // 这里发出信号:子进
程完成return NULL ; } int main (int argc , sem_init(& s 0 printf
( parent: begin\n );
```

```
pthread_t c; Pthread_create(& c      字符*argv[]){
```

 ~~child , NULL); sem\_wait(& , 0); // 初始化为 0,以便父进程等待 sem\_wait()~~
 ~~s); // 在这里等待子进程。如果子进程在父进程到达之前完~~
~~成? printf( parent:~~
~~end\n ); return 0 ; }~~
 ~~, 无效的 ,~~



## 生产者/消费者的信号量

```
sem_t 互斥体; //信号量实现的锁sem_t notfull;
sem_t 非空;

void *生产者(int循环) { int i; for (i = 0; i < 循环;
    i++)
{ sem_wait(&notfull); sem_wait(&互斥体); 把(一
    世); sem_post(&互斥体);
    sem_post(&notempty); }
```

}

代码中需要互斥锁以避免多个生产者可能尝试更新缓冲区中相同位置的竞争条件（存在旧数据被覆盖的问题）。

```
void *consumer (int循环) { int i; for (i = 0; i <
    循环; i++)
{ sem_wait(&notempty); sem_wait(&互斥体); int
    tmp = get(); sem_post(&互斥体);
    sem_post(&notfull); printf
        ( "%d\n" , tmp); }
```

}

```
int main (int argc, { //假设缓冲区有    字符*argv[])
```

```
最大容量sem_init(&mutex,,1); 0 sem_init(&notfull,,MAX); 0 sem_init(&notempty,,0); 0 // ... }
```



## 用于线程限制的信号量 (1/2)

线程限制是限制同时执行一段代码的线程数量的方法。

示例：

假设数百个线程需要处理某些任务

并行问题。然而，在代码的某个部分，每个线程都需要获取大量的内存。

◆ 如果所有线程都进入内存密集区域

同时，所有内存分配请求的总和将超过机器上的物理内存量。

结果，机器将开始颠簸（即与磁盘交换页面），整个计算速度将变得缓慢。



## 用于线程限制的信号量 (2/2)

一个简单的信号量可以解决这个问题,通过将信号量的值初始化为一次允许进入内存密集区域的最大线程数,然后在该区域周围放置一个sem\_wait()和sem\_post() ,一个信号量可以自然地限制代码危险区域中并发的线程数量。

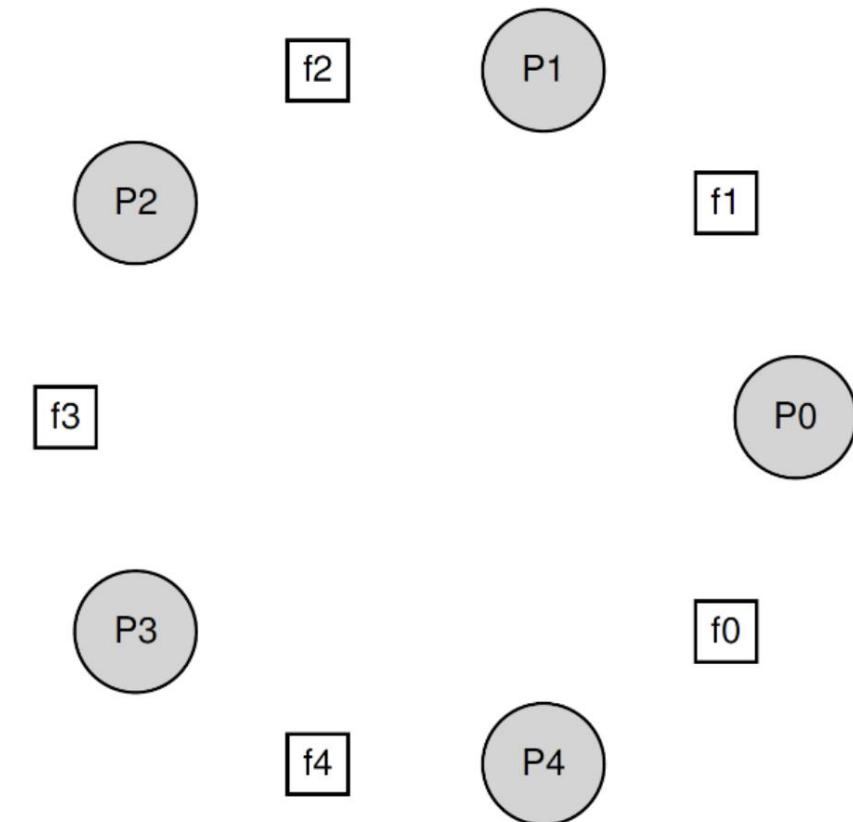


## 用餐哲学家

- ❖ Dijkstra 提出并解决的最著名的并发问题之一是哲学家就餐问题。

假设有五位“哲学家”围坐在一张桌子旁。每对哲学家之间都有一个叉子。❖哲学家们都有不需要叉子的思考时间，也有吃饭的时间。

为了吃饭，哲学家需要两把叉子，一把在左边，一把在右边。对这些分叉的争用以及随之而来的同步问题，使得这成为我们在并发编程中研究的问题。





## 哲学家就餐僵局

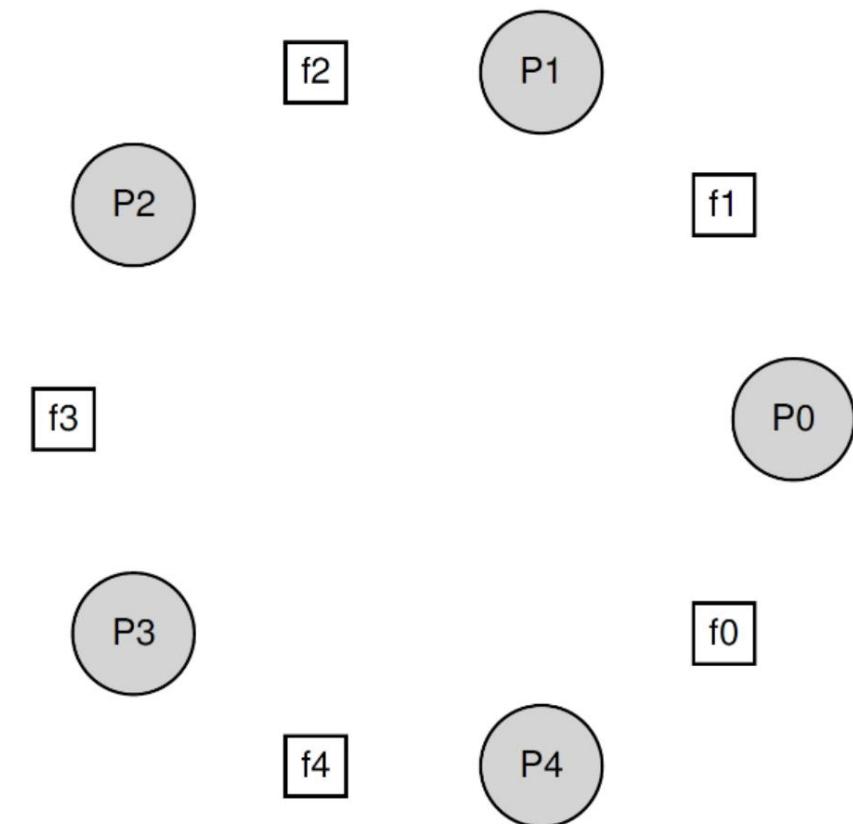
❖ 如果每个哲学家之前都碰巧抓住了他们左边的叉子

任何哲学家都可以抓住他们右边的叉子,每个人都会被困住,拿着一个叉子并等待另一个叉子,永远(死锁)。

❖ Dijkstra 解决了这个僵局

通过改变至少一位哲学家获得叉子的方式来解决这个问题。❖ 假设P4 尝试以与其他人不同的顺序获取叉子 (即,尝试在左边之前先获取右边)。

这将使P3能够抓住所需的两个叉子来完成吃/释放叉子,从而打破等待循环。





## 死锁错误

如示例所示,当线程 1 被执行时,就会发生死锁

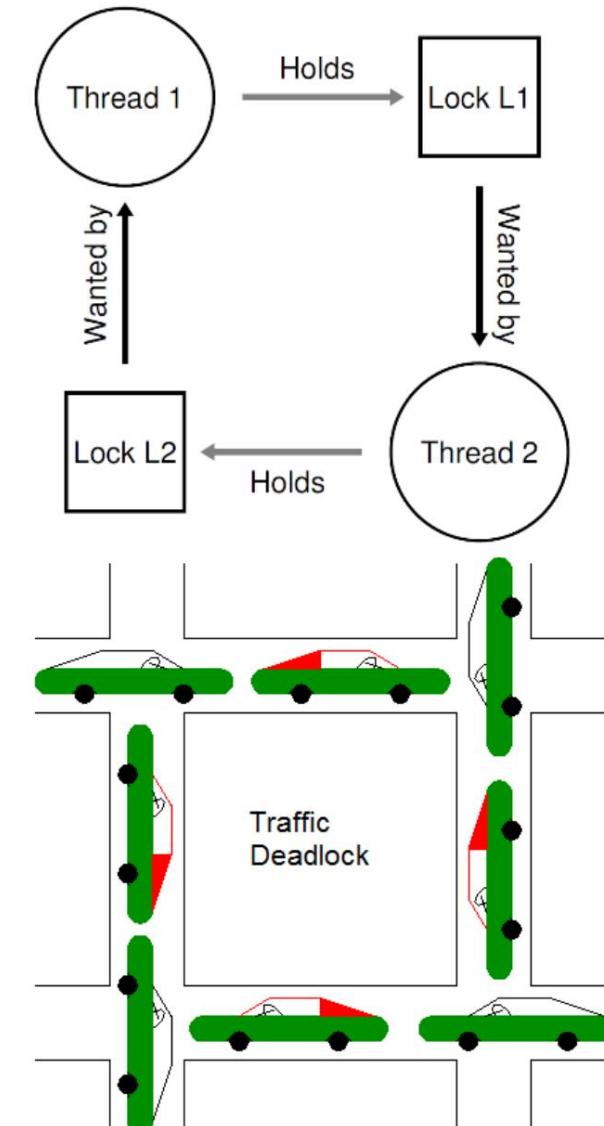
持有锁 L1 并等待锁 L2;持有锁 L2 的线程 2 正在等待 L1 被释放。

❖ 图中出现循环就表明出现了死锁。

❖ 简单的死锁,如本例所示,似乎很容易避免。例如,如果线程 1 和 2 都确保以相同的顺序获取锁,则永远不会出现死锁。

❖ 然而,锁定的设计

大型系统中的策略必须谨慎执行,以避免在代码中自然出现的循环依赖的情况下出现死锁





## 存在潜在死锁的代码

```
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
```

```
void * worker (void * arg ){ if ((long )
    arg == 0 ){
        pthread_mutex_lock(&m1);
        pthread_mutex_lock(&m2); }别的{
            pthread_mutex_lock(&m2);
            pthread_mutex_lock(&m1);
        }
        pthread_mutex_unlock(&m1);
        pthread_mutex_unlock(&m2);返回空值;
    }
```

```
int main (int argc , pthread_t 字符*argv[]){

    p1 , p2 ;
    pthread_create(&p1 , 无效的 , 工人 ,
                  (void *) (长) 0);
    pthread_create(&p2 , 无效的 , 工人 ,
                  (void *) (长) 1);
    pthread_join (p1 , NULL);
    pthread_join (p2 , NULL);返回0 ;
}
```

注意： Valgrind工具还可用于检测可执行程序中潜在的死锁。



## 死锁的条件

❖发生死锁需要满足四个条件：

1)循环等待:存在一个循环的线程链,使得每个线程

线程持有链中下一个线程正在请求的一个或多个资源（例如,锁）。

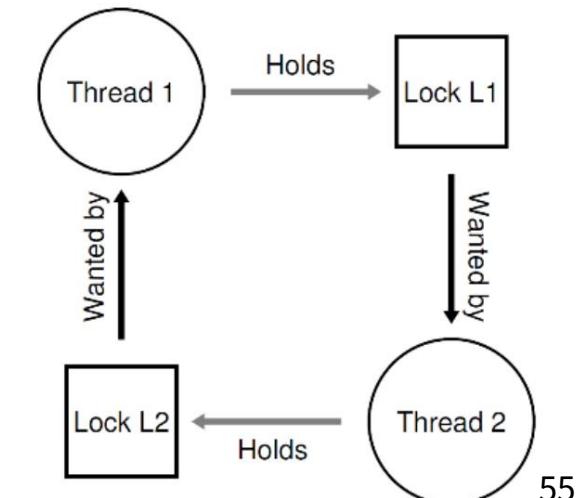
2)无抢占:资源（例如锁)不能从持有它们的线程中强行移除。

3)保留并等待:线程保留分配给它们的资源（例如,它们已经获取的锁）,同时等待额外的资源（例如,它们希望获取的锁）。

4)互斥:线程声明对其资源的独占控制权

require（例如,一次只有一个线程可以获取特定的锁）。

❖如果不满足这四个条件中的任何一个，  
不可能发生死锁。



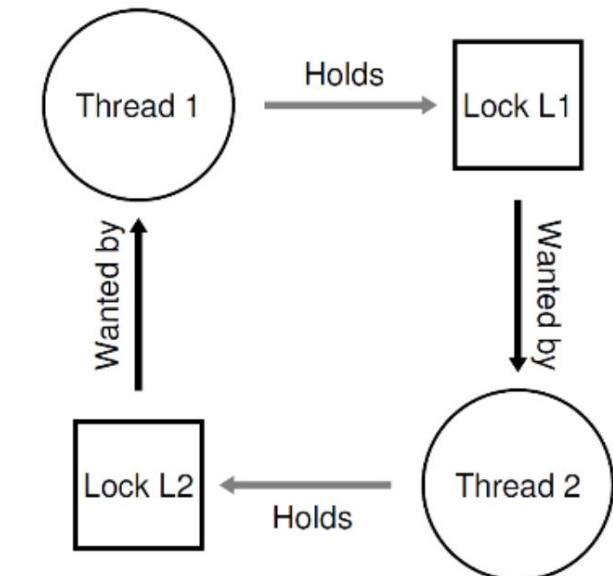


## 防止循环等待

为了防止循环等待,请在锁定获取时使用全序编写锁定代码。例如,如果系统中只有两个锁 (L1 和 L2) ,则可以通过始终在获取 L2 之前获取 L1 来防止死锁。如此严格的排序确保不会出现循环等待;因此,没有僵局

- ❖ 总的锁排序可能很难实现,因此偏序是构造锁获取以避免死锁的有用方法。

- ❖ 锁排序需要深入了解代码库以及如何调用各种例程;只要一个错误就可能导致僵局。





## 防止抢占

许多线程库提供例程 `pthread_mutex_trylock()` 要么获取锁（如果可用）并返回成功，要么返回指示锁已被持有的错误代码。

❖ 这个例程可以用来避免无抢占问题：

顶

```
部: pthread_mutex_lock (L1) ;  
if (pthread_mutex_trylock (L2) != 0 )  
{ pthread_mutex_unlock (L1);  
    转到顶  
部; };
```

这种方法并没有真正增加抢占（从拥有它的线程中夺走锁的强制操作），而是使用“尝试锁”方法来允许开发人员退出锁所有权（即抢占他们的锁）。自己的所有权以一种优雅的方式。



## 防止等待

❖ 死锁的保持等待要求可以通过以下方式避免

一次获取所有锁,原子方式如下:

```
pthread_mutex_lock(预防); // 开始获取  
pthread_mutex_lock(L1);  
pthread_mutex_lock(L2);  
...  
pthread_mutex_unlock(预防); // 结尾
```

❖ 这里要求任何时候任何线程获取锁时,它首先

获取全局预防锁。这段代码保证了在获取锁的过程中不会发生不合时宜的线程切换。

这种技术可能会降低并发性,因为所有的锁必须尽早 (立即)获得,而不是在真正需要时获得。



## 防止相互排斥

- ❖ 避免互斥需要消除关键的代码中的部分。

一种方法是使用强大的硬件指令,其中数据结构可以以一种不需要的方式构建

需要显式锁定。

- ❖ 作为一个例子,我们之前讨论过原子交换指令序列。例如,它可用于以原子方式将值增加一定量。



## 通过调度避免死锁

在某些情况下,避免死锁比预防死锁更可取。

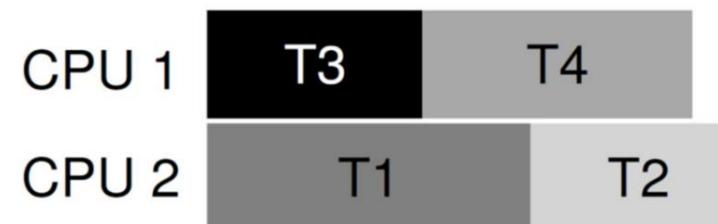
- ❖ 回避要求系统有一些额外的先验

各个线程在执行期间可能获取哪些锁的信息,并随后以保证不会发生死锁的方式调度所述线程。  
这一般是不可行的!

**示例:**两个处理器运行四个线程 (T1、T2、T3、T4)。每个线程抓取  
(yes)或不抢 (no) 锁 L1 和 L2 如下图:

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

因此,智能调度程序可以计算出,只要 T1 和 T2 不同时运行,就不会出现死锁。这是一个这样的时间表:





## 死锁检测和恢复

一种策略是允许死锁偶尔发生,然后在检测到此类死锁时采取一些操作。

- ❖ 死锁检测器定期运行,构建进程

来自资源分配图 (RAG) 的“等待”图并检查其周期。

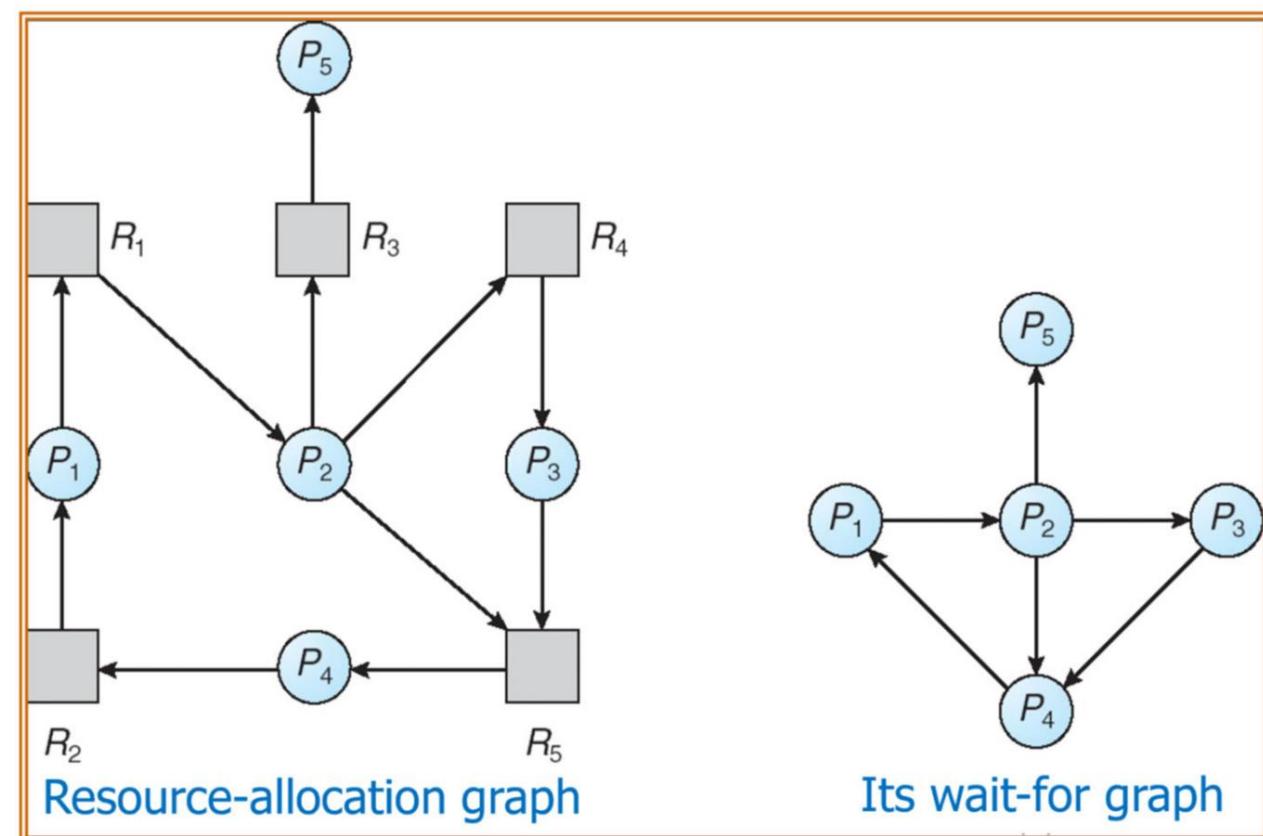
- ❖ 要从死锁中恢复,要么中止

所有死锁进程,要么一次中止一个  
进程,直到消除死锁循环。

- ❖ 注意:如果我们没有循环 (例如通过  
删除 P)

4 等待

P 1) 那么就不会出现死锁。为什么?





## 资源分配图 (RAG)

一组顶点V和一组边E。  $V$  分为两种类型：

❖  $P = \{P_1, P_2, \dots, P_n\}$ , 系统中所有进程组成的集合。 ❖  $R = \{R$

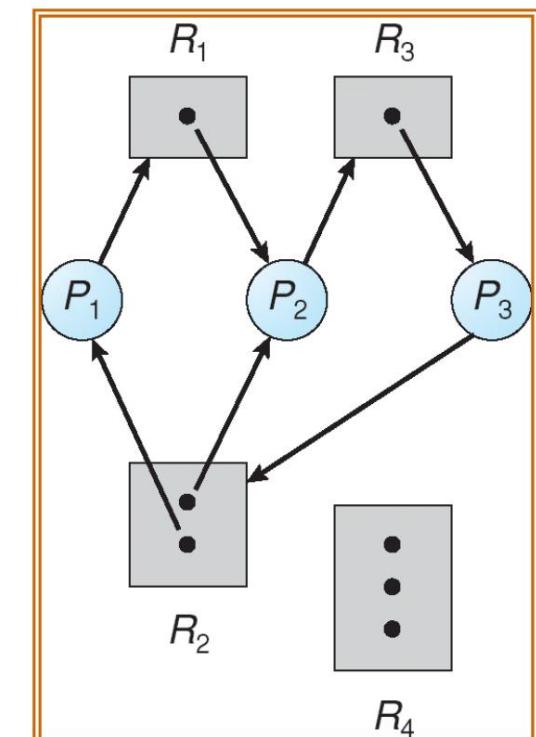
$1, R_2, \dots, R_m\}$ , 所有资源组成的集合

系统中的类型。 请求边 -

有向边  $P_i \rightarrow R_j$  分配边 - 有向边  $R_j \rightarrow P_i$  资源节点内的点

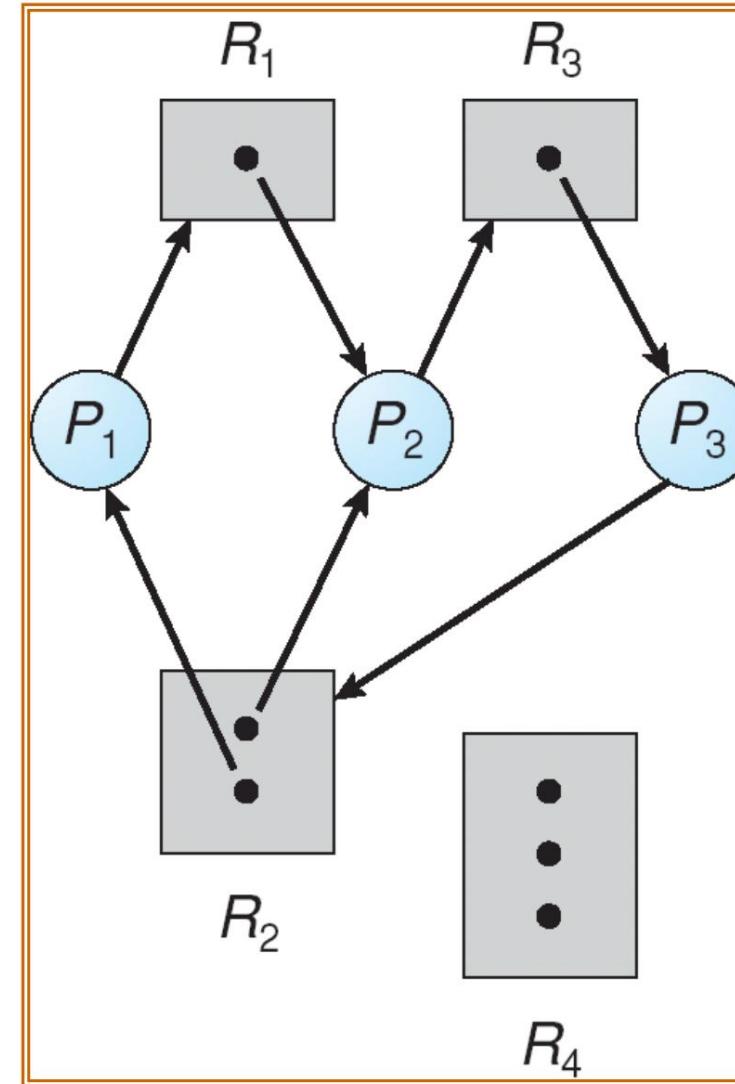
表示资源实例的数量 (例如,  $R$

4 有三个实例)





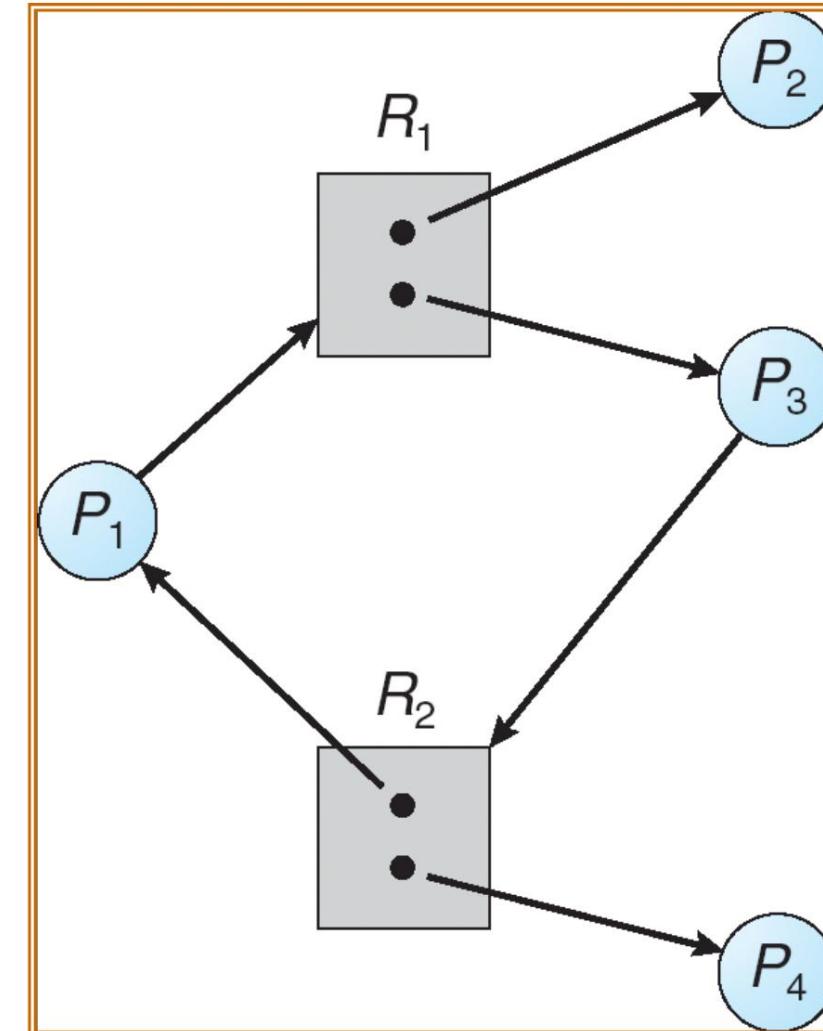
## 存在死锁的资源分配图





## 无死锁的资源分配图

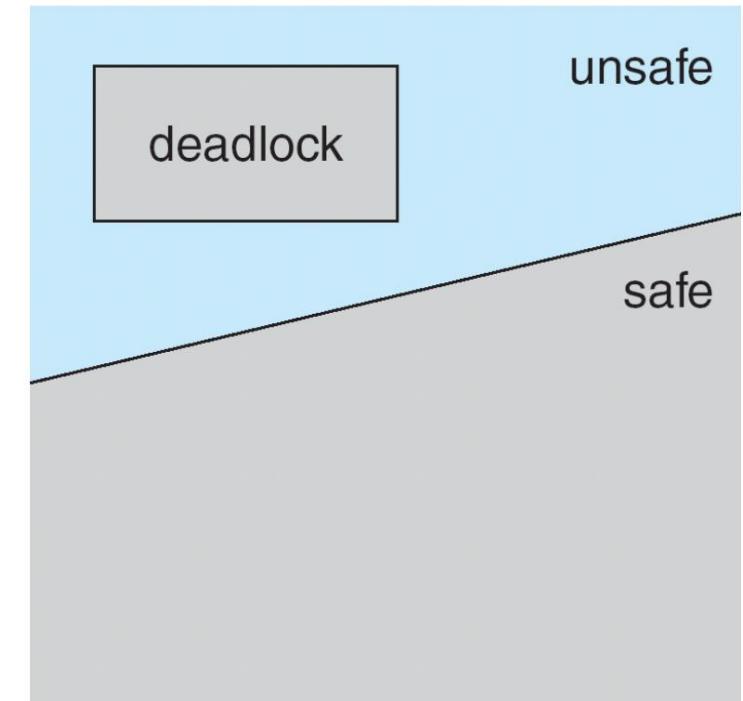
即使有循环,但有些进程并不处于hold and wait状态 (P2和P4)





## 安全状态

- ❖ 当进程请求可用资源时, 系统必须决定立即分配是否使系统处于安全状态。
- ❖ 如果存在所有的安全序列, 则系统处于安全状态流程来完成。
- ❖ 如果系统处于安全状态 没有死锁
- ❖ 如果系统处于不安全状态 可能会出现死锁
- ❖ 避免死锁❖ 确保系统永远不会进入不安全状态。





## 安全和不安全状态示例

### 示例 1: 安全 (为什么?)

系统共有 12 个资源 (当前可用 =  $12 - 5 - 2 - 2 = 3$ )

	<u>最多需要完成</u>	<u>10 4 9</u>	<u>目前举行</u>
P0:			5
P1:			2
P2:			2

### 示例 2: 除非进程释放其当前持有的内容,否则不安全 (为什么?)

系统共有 12 个资源 (当前可用 =  $12 - 5 - 2 - 3 = 2$ )

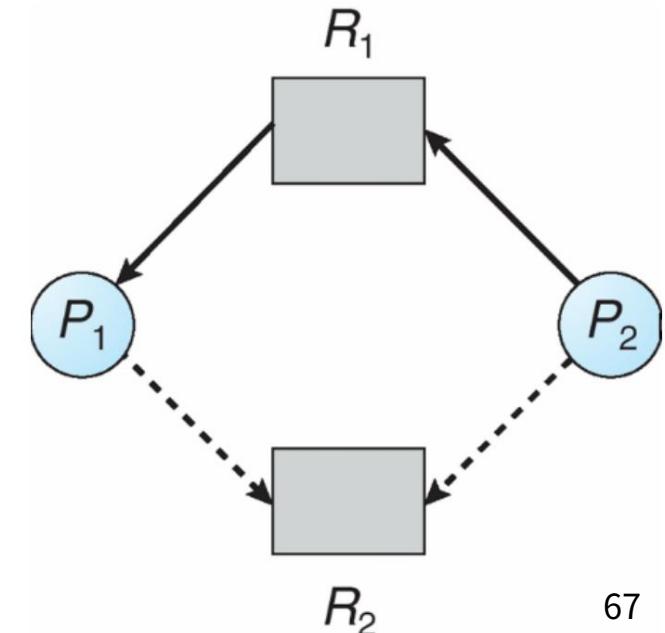
	<u>最多需要完成</u>	<u>10 4 9</u>	<u>目前举行</u>
P0:			5
P1:			2
P2:			3 (这是唯一的区别)



## 具有索赔边缘的 RAG (1/2)

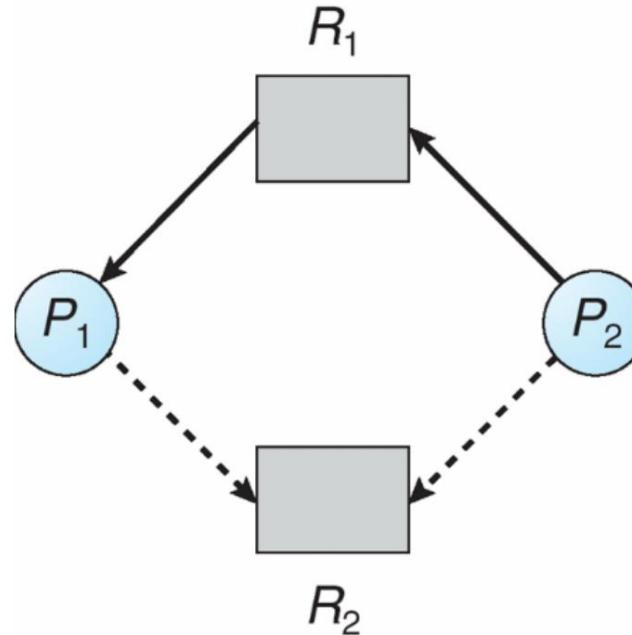
- ◆ 声明边  $P_i \rightarrow R_j$  表示进程  $P_i$  将来可能会请求资源  $R_j$  ;由虚线表示。
- ◆ 当进程发生时,声明边转换为请求边  
请求资源并等待它。当资源被分配给进程时,  
请求边转换为分配边。 ◆ 资源必须在系统中预先声明。

$P_2$  可以请求  $R_2$   
(  $R_2$  仍然免费)

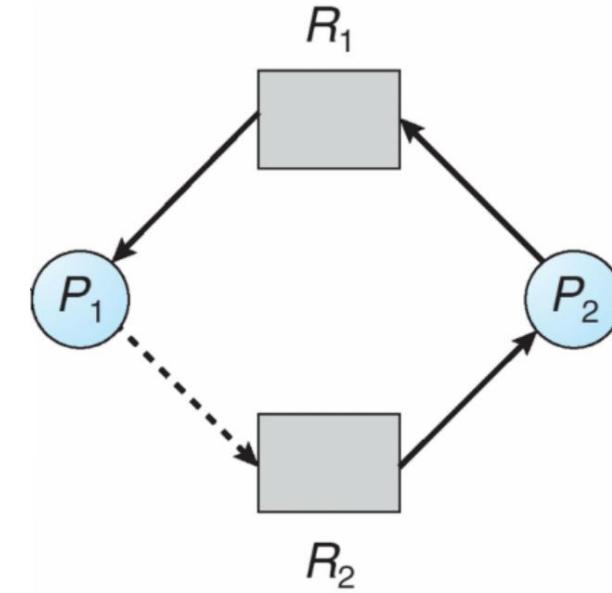




## 具有索赔边缘的 RAG (2/2)



$P_2$ 可以请求 $R_2$   
( $R_2$ 仍然免费)



不安全状态

算法: 只有当将请求转换为赋值不会导致不安全状态时, 请求才能被授予。



## 银行家算法

- ❖ 当一个资源包含多个实例时

资源分配图方法不起作用,因为可能存在循环但没有死锁。

银行家算法 (BA)因它是一种算法而得名。

银行家可以使用这种方法来确保当他们放出现金时,他们仍然能够满足所有客户的  
要求。

- ❖ 当一个线程启动时,它必须预先声明

它可能请求的最大资源分配,直至系统上可用的数量。

- ❖ 当发出请求时,调度程序确定是否

批准请求将使系统处于安全状态。如果不是,则线程必须等待,直到可以安全地授予请  
求。



## 数据结构文学学士

令  $p$  = 进程数，  $r$  = 资源类型数。

可用：长度为  $r$  的向量。如果  $\text{available}[j] = k$ ，则有  $k$  个资源类型  $R_j$  的实例可用。

最大值：  $p \times r$  矩阵。如果  $\text{Max}[i,j] = k$ ，则进程  $P_i$  最多可以请求资源类型  $R_j$  的  $k$  个实例。

分配：  $p \times r$  矩阵。如果  $\text{Allocation}[i,j] = k$ ，则  $P_i$  当前分配有  $R_j$  的  $k$  个实例。

需要：  $p \times r$  矩阵。如果  $\text{Need}[i,j] = k$ ，则  $P_i$  可能还需要  $k$  个  $R_j$  实例才能完成其任务。

$$\text{需要}[i,j] = \text{最大}[i,j] - \text{分配}[i,j]。$$



## BA 数据结构示例

5 个进程 P0 到 P4 (p = 5)

3 种资源类型 A (10 个实例)、B (5 个实例) 和 C (7 个实例) → (r = 3)

◆ T0 时刻数据结构的状态:

	<u>最大可用分配</u>			<u>需要</u>
	ABCABCABC			ABC
P <sub>0</sub>	0 1 0	7 5 3	3 3 2	7 4 3
P <sub>1</sub>	2 0 0	3 2 2		1 2 2
P <sub>2</sub>	3 0 2	9 0 2		6 0 0
P <sub>3</sub>	2 1 1	2 2 2		0 1 1
P <sub>4</sub>	0 0 2	4 3 3		4 3 1



## BA安全算法

❖以下安全算法用于确定特定状态是否安全。

1.令Work和Finish分别为长度为r和p的向量。

- 工作是可用资源的工作副本,将在分析过程中进行修改。
- Finish是一个布尔向量,指示特定进程是否可以完成。 (或者到目前为止分析已经完成。)
- 对于所有进程,将“Work”初始化为“Available”,并将“Finish”初始化为“False”。

2.找到一个i,使得Finish[i]为False并且Need[i,j] < Work[j] (对于j = 1 到r)。这表明进程i尚未完成,但可以使用给定的可用工作集完成。如果不存在这样的i,则转至步骤4。

3.设置Work[j] = Work[j] + Allocation[i, j] (对于j = 1 到r)并将Finish[i]设置为True。这对应于进程i完成并将其资源释放回工作池。然后循环回到步骤 2。

4.如果Finish[i]对于所有都为True,则该状态是安全状态,因为已经找到安全序列,否则该状态是不安全的。



## BA 安全算法示例

❖ P0 到 P4 共 5 个进程； 3 种资源

类型A (10 个实例) 、 B (5 个实例) 和 C (7 个实例) 。

❖ T0时刻数据结构的状态：

	<u>最大可用分配</u>			<u>需要</u>
	ABCABCABC			ABC
磷 <sub>0</sub>	0 1 0	7 5 3	3 3 2	7 4 3
磷 <sub>1</sub>	2 0 0	3 2 2		1 2 2
磷 <sub>2</sub>	3 0 2	9 0 2		6 0 0
磷 <sub>3</sub>	2 1 1	2 2 2		0 1 1
磷 <sub>4</sub>	0 0 2	4 3 3		4 3 1

❖ 通过应用安全算法, 系统自启动以来就处于安全状态

序列 P<sub>1, 3, 0</sub>, 磷<sub>4, 2, 0</sub>, 磷<sub>0</sub> 满足安全标准



## BA 资源请求算法 (1/2)

- ❖ 该算法确定新请求是否安全，并且仅在安全时才授予该请求。
- ❖ 当提出请求时（不超过当前可用的资源），假装它已被授予，然后查看结果状态是否是安全的。如果是，则同意该请求，如果否，则拒绝该请求。



## BA 资源请求算法 (2/2)

1. 设 Request:  $p \times r$  矩阵, 其中如果  $\text{Request}[i, j] = k$ , 则表示进程  $P_i$  请求  $k$  个资源类型  $R_j$  的实例。

1. 如果  $k > \text{Need}[i, j]$  (对于任何  $j = 1$  到  $r$ ) , 则引发错误条件并停止。
2. 如果  $k > \text{available}[j]$  (对于任何  $j = 1$  到  $r$ ) , 则该进程必须等待资源变得可用并停止。

2. 通过假装请求已被授予, 然后查看结果状态是否安全 (使用安全算法) , 检查是否可以安全地授予请求。如果是, 则授予该请求, 如果不是, 则该进程必须等待, 直到其请求可以安全地授予。批准请求 (或假装出于测试目的) 的程序是:

- ❖ 可用 = 可用 - 请求
- ❖ 分配 = 分配 + 请求
- ❖ 需求 = 需要 - 请求



## BA 资源请求示例

对于前面的例子,假设来自P<sub>1</sub>的请求

P<sub>1</sub>的(1,0,2)

◆回忆P<sub>1</sub>分配、可用和P<sub>1</sub>需要的是200,332和122分别。

◆由于 Request ≤ Available,即 (1,0,2) ≤ (3,3,2),则授予请求并更新状态如下所示:

	<u>最大可用分配</u>			<u>需要</u>
	ABC	ABC	ABC	ABC
磷 <sub>0</sub>	0 1 0	7 5 3	2 3 0	7 4 3
磷 <sub>1</sub>	3 0 2	3 2 2		0 2 0
磷 <sub>2</sub>	3 0 2	9 0 2		6 0 0
磷 <sub>3</sub>	2 1 1	2 2 2		0 1 1
磷 <sub>4</sub>	0 0 2	4 3 3		4 3 1

◆通过应用安全算法,系统自启动以来就处于安全状态

序列P<sub>1</sub>、磷<sub>3</sub>、磷<sub>4</sub>、磷<sub>0</sub>、磷<sub>2</sub>满足安全标准。



## 阅读材料和资源列表

来自 Arpaci-Dusseau 教科书：  
第 27 章：插  
曲：Thread API