



第二章 进程的描述与控制

2.1 进程的描述

2.2 进程控制

2.3 进程同步

2.4 经典进程的同步问题

2.5 进程通信

2.6 线程(Threads)的基本概念

2.7 线程的实现

韩明峰 QQ 2022446359



2.1 进程的描述

2.1.1 进程的定义和特征

1. 进程的定义

进程是具有独立功能的程序在一个数据集合上运行的过程，它是系统进行[资源分配](#)和[调度](#)的一个独立单位。



2. 进程的特征

进程和程序是两个截然不同的概念，进程除了具有程序所没有的**PCB**结构外，还具有下面一些特征：

- (1) 动态性。一个程序可以对应多个进程。
- (2) 并发性。
- (3) 独立性。进程是资源分配和调度的基本单位。
- (4) 异步性。尽管进程是以不可预知的速度向前推进，但如果配置完善的进程同步机制，则多次运行也可获得相同的结果。



2.1.2 进程的基本状态及转换

1. 进程的三种基本状态

进程在其生命周期内可能具有多种状态。每一个进程至少应处于以下三种基本状态之一：

(1) 就绪 (Ready) 状态：万事俱备，只欠调度。

(2) 执行 (Running) 状态。

(3) 阻塞 (Block) 状态：因等待资源或等待事件发生而阻塞。

2. 三种基本状态的转换

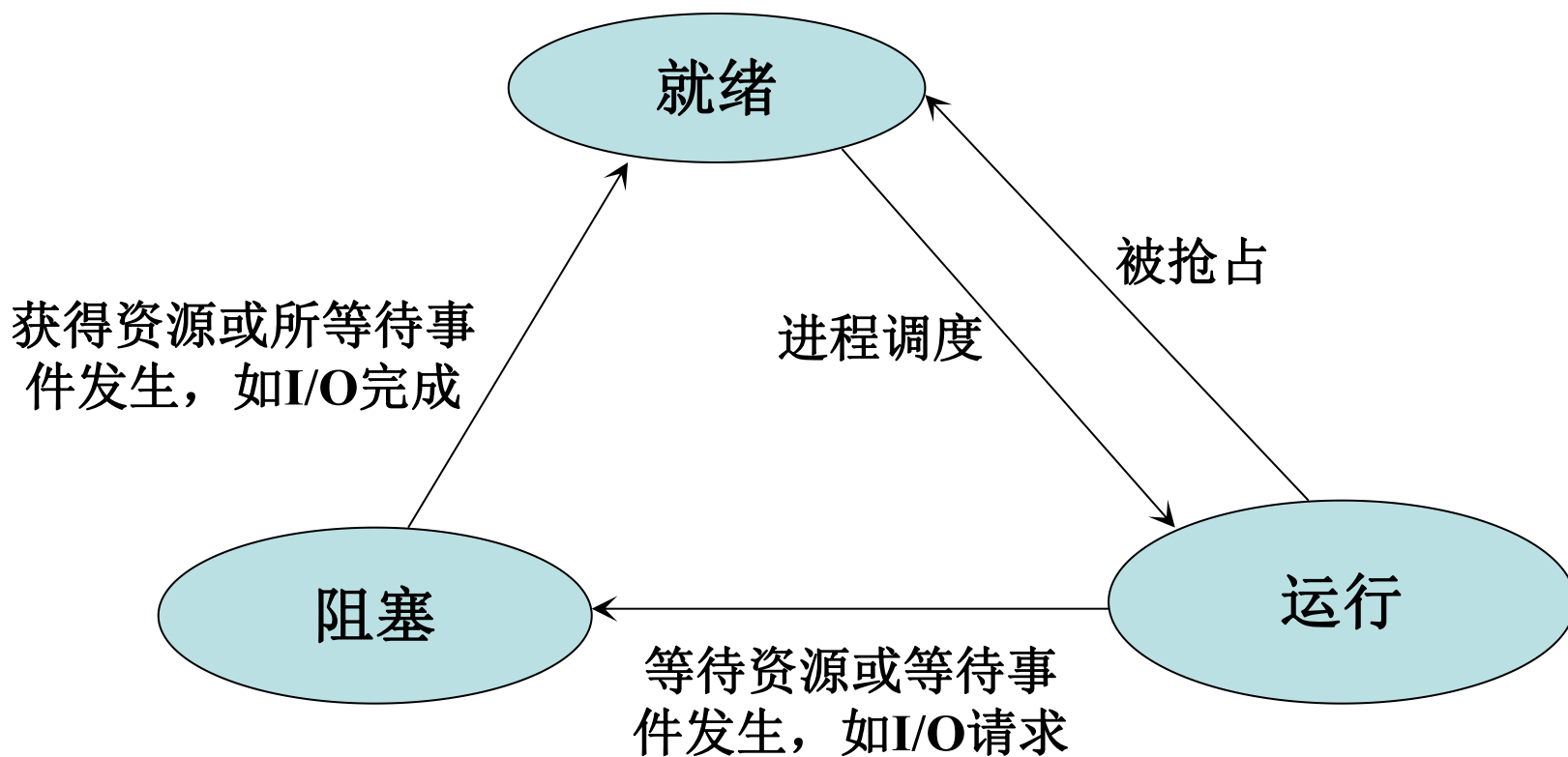


图2-5 进程的三种基本状态及其转换



2.1.3 挂起操作和进程状态的转换

1. 挂起操作的引入

(1) 交换。OS需要释放足够的内存空间，以调入并执行处于就绪态的进程。

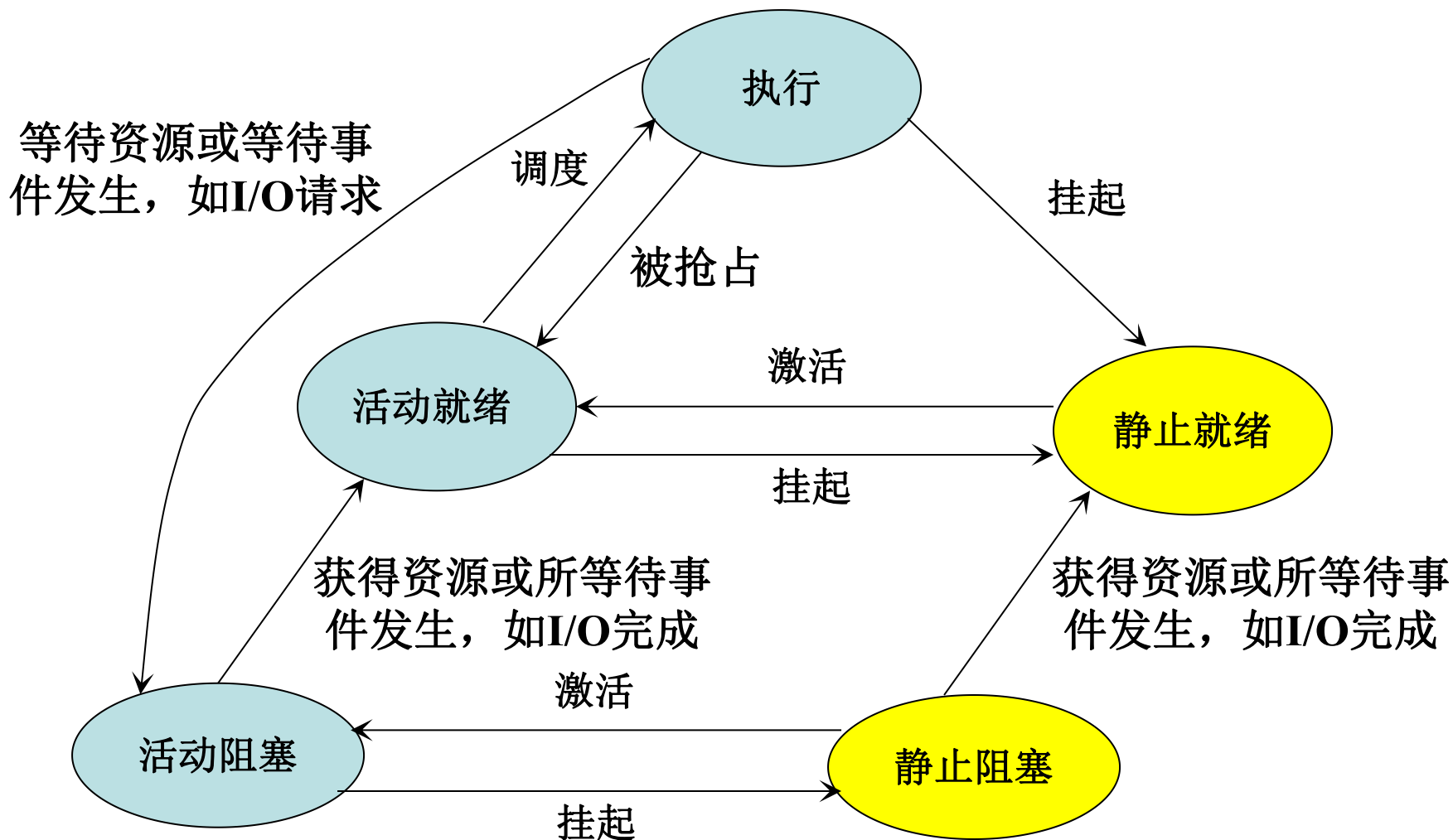
(2) 分时因素也会影响交换。

(3) 其它OS的原因。如OS挂起可能会导致问题的进程。

(4) 交互式用户可能因调试程序或其它原因而挂起程序的运行。

(5) 父进程请求。如进程A创建进程B来执行文件读操作；进程B在读文件中遇到错误，并报告给进程A；进程A挂起进程B，调查错误原因。

2. 引入挂起操作后进程状态的转换





2.1.4 进程管理中的数据结构

1. 进程控制块中的信息

(1) 标识符

- 1) 进程标识符;
- 2) 父进程的标识符;
- 3) 用户标识符。



2.1.4 进程管理中的数据结构

1. 进程控制块中的信息

(2) 处理机状态信息

1) 通用寄存器;

2) 程序计数器PC;

3) 程序状态字PSW, 如中断允许/禁用标志、执行模式等;

4) 栈指针。

现场信息一般保存在栈中, PCB中有栈指针。



(3) 进程调度信息

1) 进程状态。指明进程的当前状态，它是作为进程调度的依据。

2) 进程优先级。是用于描述进程使用处理机的优先级别的一个整数，优先级高的进程应优先获得处理机。

3) 进程调度所需的其它信息。它们与所采用的进程调度算法有关，比如，进程已等待CPU的时间总和、进程已执行的时间总和等。

4) 事件。是指进程由执行状态转变为阻塞状态所等待发生的事件，即阻塞原因。如等待资源释放，等待I/O结束等。



(4) 进程控制信息

1) 进程同步和通信信息。如消息队列、信号量等，它们可能全部或部分地放在PCB中。

2) 存储管理信息。如指向本进程页表的指针等。

3) 所用资源列表。说明由进程打开和使用的系统资源。如打开的文件、I/O设备等的描述、状态信息。

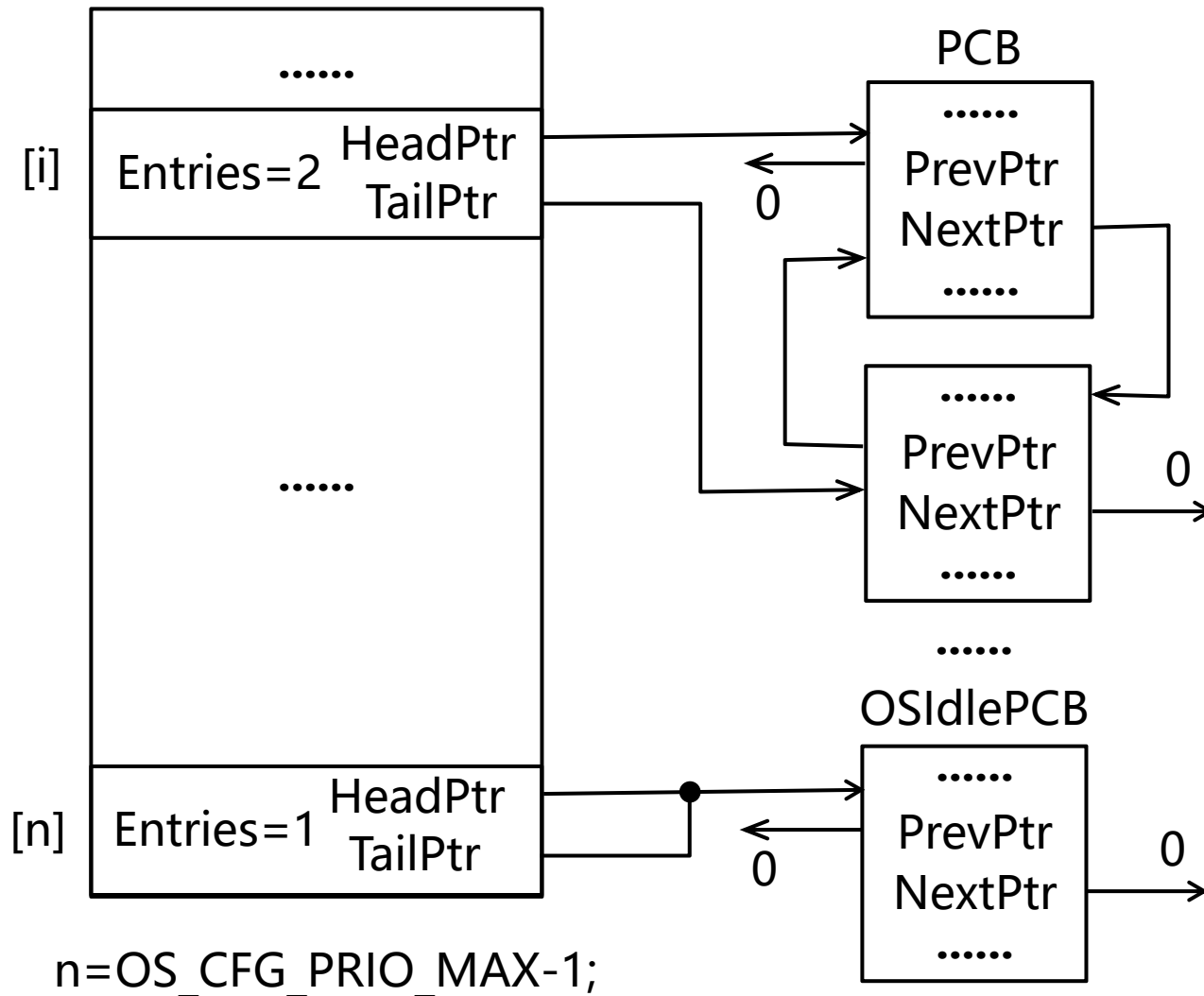
4) 链接指针。它给出了本进程(PCB)所在队列中的下一个进程的PCB的首地址。



2. 进程控制块的组织方式

(1) 进程控制块通常用链表来进行管理，如PCB中的NextPtr和PrevPtr在进程就绪表中双向链接就绪进程的PCB。

OS_RDY_LIST OSRdyList[OS_CFG_PRIO_MAX]



就绪进程链表

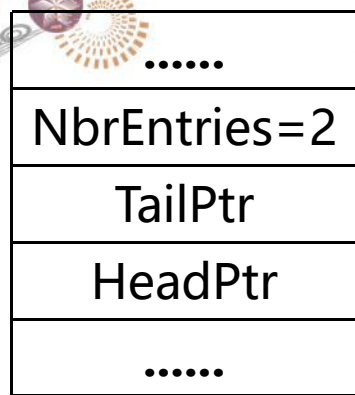


2. 进程控制块的组织方式

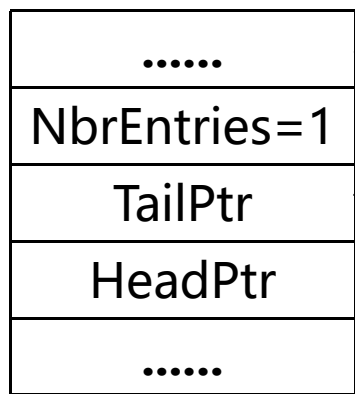
(2) 等待进程链表通常使用专门的等待控制块，在等待控制块中有指针指向等待进程的PCB。

一个进程还可以同时等待多个内核对象。

等待进程链表的排队通常基于优先级或FIFO原则。

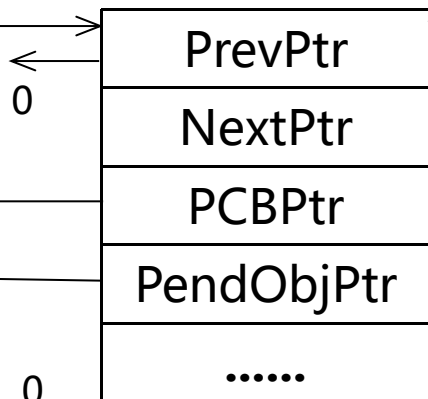


信号量

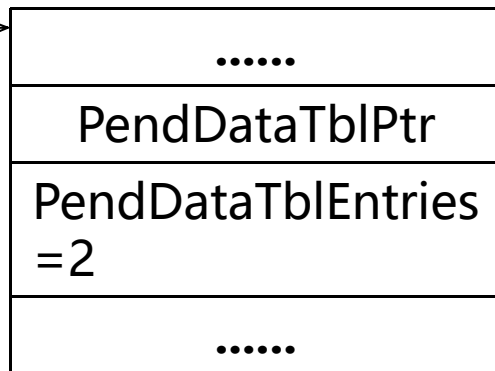


消息队列

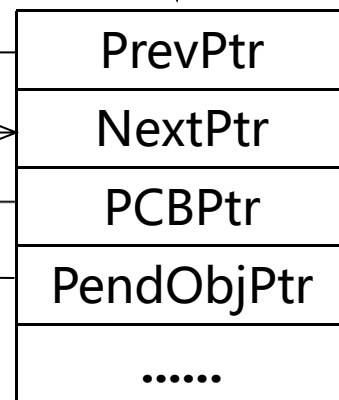
多个进程等待同一个内核对象以及一个进程等待多个内核对象的示意图



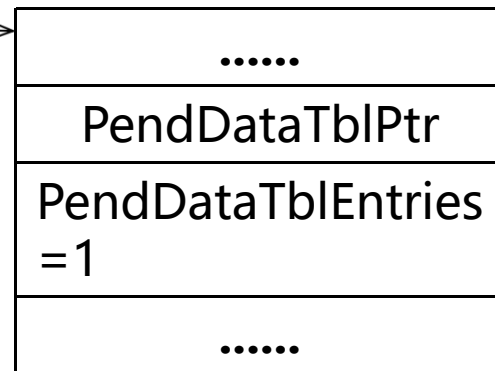
等待控制块数组



高优先级进程PCB



等待控制块



低优先级进程PCB

2.2 进程控制



2.2.1 原语

所谓“原语”一般是指由若干条指令所组成，用来实现某个特定功能，在执行过程中不可被中断的程序段。

原语和系统调用是两个不同的概念：

(1) 原语主要强调操作的不可分割性，可以认为是一个不可中断的子程序调用；

(2) 操作系统中有些系统调用是以原语的形式出现的，但并不是所有系统调用都是原语。因为如果那样的话，整个系统的并发性就不可能得到充分的发挥。



2.2.2 进程的创建

1. 进程的层次结构

通常把创建进程的进程称为父进程，而把被创建的进程称为子进程。子进程可继续创建更多的孙进程，由此便形成了一个进程的层次结构。

如在UNIX中，进程与其子孙进程共同组成一个进程家族；（作用？）

但Windows中不存在任何进程层次结构的概念，进程间只是控制与被控制的简单关系。

2. 进程图

所谓进程图就是用于描述进程间关系的一棵有向树。

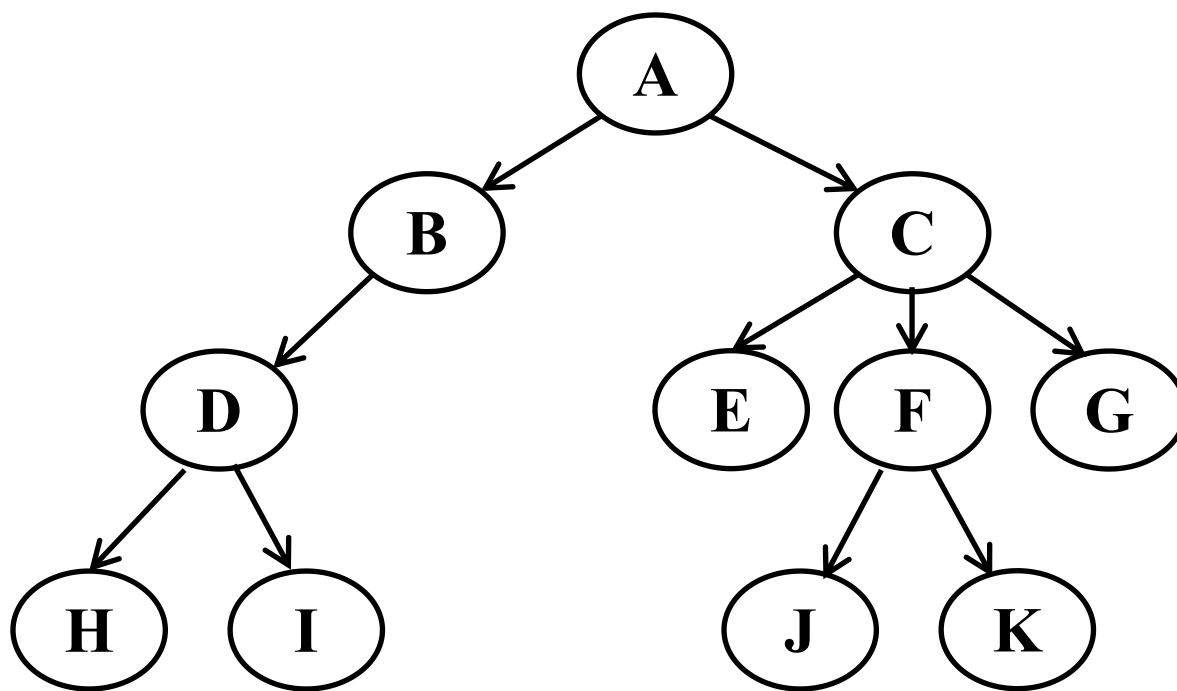


图2-13 进程树



3. 引起创建进程的事件

(1) 用户登录。

(2) 作业调度。

(3) 用户程序提出某种请求，系统内核为用户创建一个新进程来提供服务。

如用户进程要求打印文件，系统将为它创建一个打印进程。

(4) 用户进程自己创建新进程。

如某用户进程需要不断地从键盘读入数据，再对读入的数据进行处理，最后将处理结果以表格的形式显示在屏幕上。

可以分别创建键盘输入进程、数据处理进程和表格输出进程。





4. 进程的创建 (Creation of Process)

(1) 申请空白**PCB**，为新进程申请获得唯一的数字标识符。

(2) 为新进程分配其运行所需的资源，包括各种物理和逻辑资源，如内存、文件、**I/O**设备和**CPU**时间等。

(3) 初始化进程控制块 (PCB)。

(4) 如果进程就绪队列能够接纳新进程，便将新进程插入就绪队列。



2.2.3 进程的终止

1. 引起进程终止(Termination of Process)的事件

(1) 正常结束

(2) 异常结束

越界错；保护错；非法指令；特权指令错；运行超时；
等待超时；算术运算错；I/O故障

(3) 外界干预



2. 进程的终止过程

(1) 根据被终止进程的标识符，从PCB集合中检索出该进程的PCB，从中读出该进程的状态；

(2) 若被终止进程正处于执行状态，应立即终止该进程的执行，并置调度标志为真；

(3) 若该进程还有子孙进程，还应将其所有子孙进程也都予以终止，以防它们成为不可控的进程；

(4) 将被终止进程所拥有的全部资源或者归还给其父进程，或者归还给系统；

(5) 将被终止进程(PCB)从所在队列(或链表)中移出。



2.2.4 进程的阻塞与唤醒

1. 引起进程阻塞的事件

- (1) 向系统请求共享资源失败。
- (2) 等待某种操作(I/O操作)的完成。
- (3) 新数据(另一进程提供)尚未到达。
- (4) 等待新任务的到达。一些系统进程，完成任务后就把自己阻塞起来，等待新的任务，如网络守护进程。



2. 进程阻塞过程

(1) 进入block过程后，由于该进程还处于执行状态，所以应先立即停止执行，把进程控制块中的现行状态由“执行”改为阻塞，并将PCB插入阻塞队列。

(2) 如果系统中设置了因不同事件而阻塞的多个阻塞队列，则应将本进程插入到具有相同事件的阻塞队列。

(3) 最后，转调度程序进行重新**调度**，将处理机分配给另一就绪进程，并进行**切换**，亦即，保留被阻塞进程的处理机状态，按新进程的PCB中的处理机状态设置CPU的环境。



3. 进程唤醒过程

- (1) 首先把被阻塞的进程从等待该事件的阻塞队列中移出，将其PCB中的现行状态由阻塞改为就绪；
- (2) 然后再将该PCB插入到就绪队列中；
- (3) 最后，如果基于抢占调度策略，则转调度程序进行重新调度，可能发生进程切换。



2.2.5 进程的挂起与激活

1. 进程挂起引起的状态改变

- (1) 活动就绪改为静止就绪;
- (2) 活动阻塞改为静止阻塞;
- (3) 运行态改为静止就绪，此时需要运行调度程序。



2. 进程的激活过程

(1) 将进程从外存调入内存;

(2) 静止就绪改为活动就绪。如果基于抢占调度策略，则需要运行**调度程序**，以判断是否需要运行被激活进程。

(3) 静止阻塞改为活动阻塞。

2.3 进 程 同 步

2.3.1 进程同步的基本概念

1. 资源同步

指进程间相互竞争使用独占型资源，一旦一个进程占用独占型资源，其它的进程必须等待；占用独占型资源的进程释放资源后，等待进程之一才能使用独占型资源。这种现象也称为互斥。

举例：当一个进程正在使用打印机时，另一个进程必须等待。

2. 活动(行为)同步



一个(一些)进程以另一个(另一些)进程的**执行结果**作为本进程的**执行条件**，为完成同一个任务而相互合作。

举例1：计算进程必须等待输入进程向缓冲区输入数据后，才能进行计算，否则必须等待；

而输入进程必须等待计算进程将缓冲区的数据读走后，才能进行下一次输入，否则必须等待。

此例是双向同步。





举例2：打印进程必须等待计算进程向缓冲区写入计算结果后，才能进行打印输出；

而计算进程必须等待打印进程将缓冲区中的计算结果打印输出后，才能向缓冲区写入下一次的计算结果，否则就会造成混乱。

此例是双向同步。





举例3：高铁服务员们关门后司机才能启动高铁运行；
高铁停止后服务员们才能开门。

此例是双向同步。

举例4：生活中的约会。





3. 临界资源(Critical Resouce)

我们把某段时间内只能允许一个进程使用的共享资源称为**临界资源**。

4. 临界区(critical section)

每个进程中访问临界资源的那段代码称为**临界区**。

对临界资源必须互斥访问，即一个进程在临界区，其它要访问临界资源的进程必须在临界区外等待；临界区里的进程退出临界区后，等待进程之一才能进入临界区。

多个进程访问共享资源，如果不互斥，执行结果取决于每个进程的精确时序，叫做**竞争条件**。

【竞争条件举例】假脱机打印程序。
两个进程想同时访问假脱机目录：

(1) 进程A:

NextFreeSlot=in;

进程切换;

(2) 进程B:

NextFreeSlot=in;

存入ddd. txt;

in=++NextFreeSlot;

进程切换;

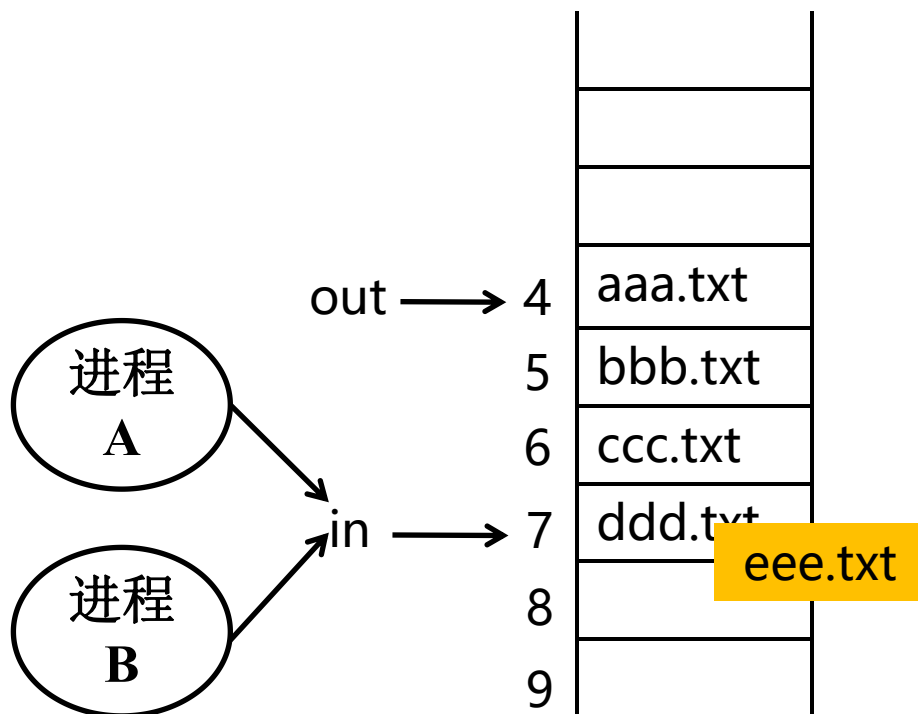
(3) 进程A:

存入eee. txt;

in=++NextFreeSlot;

此时ddd. txt被覆盖丢失。

假脱机目录





5. 互斥机制应遵循的规则

(1) 空闲让进。

(2) 忙则等待。

(3) 有限等待。

(4) 让权等待：当进程不能进入临界区时，应立即释放处理机，以避免“忙等”。



2.3.2 硬件互斥机制

1. 关中断

(1) 关中断实现互斥的原理：


- ① 关中断避免了进程（线程）与ISR间的竞争条件。
- ② 关中断避免了因中断引起的进程切换。

(2) 关中断实现临界区的方式：

关中断

临界区

开中断



(3) 关中断的方法存在一些缺点：

① 滥用关中断可能导致严重后果，如忘开中断；

② 关中断时间过长，会影响系统的实时性；

③ 关中断方法也不适用于多CPU 系统，因为在一个处理器上关中断并不能防止进程在其它处理器上执行相同的临界段代码。



(4) 关中断对操作系统本身是一项很有用的技术。

(5) 通用操作系统一般不会为用户提供关中断的调用，而一般实时嵌入式操作系统都为用户提供关中断的调用。

2. 利用Test-and-Set指令实现互斥

这是一种借助一条硬件指令——“测试并置位”指令 TS(Test-and-Set)以实现互斥的方法。在**多CPU**下该指令的执行将**锁住内存总线**。该指令的示意代码如下：

```
Boolean TS(boolean *lock)
{
    Boolean old;
    old=*lock;
    *lock=TRUE;
    return old;//返回原先的锁状态
}
```



Test-and-Set指令的使用方式:

do {

...

while TS(&lock) //锁状态为TRUE, 表示资源被占用

; //空转

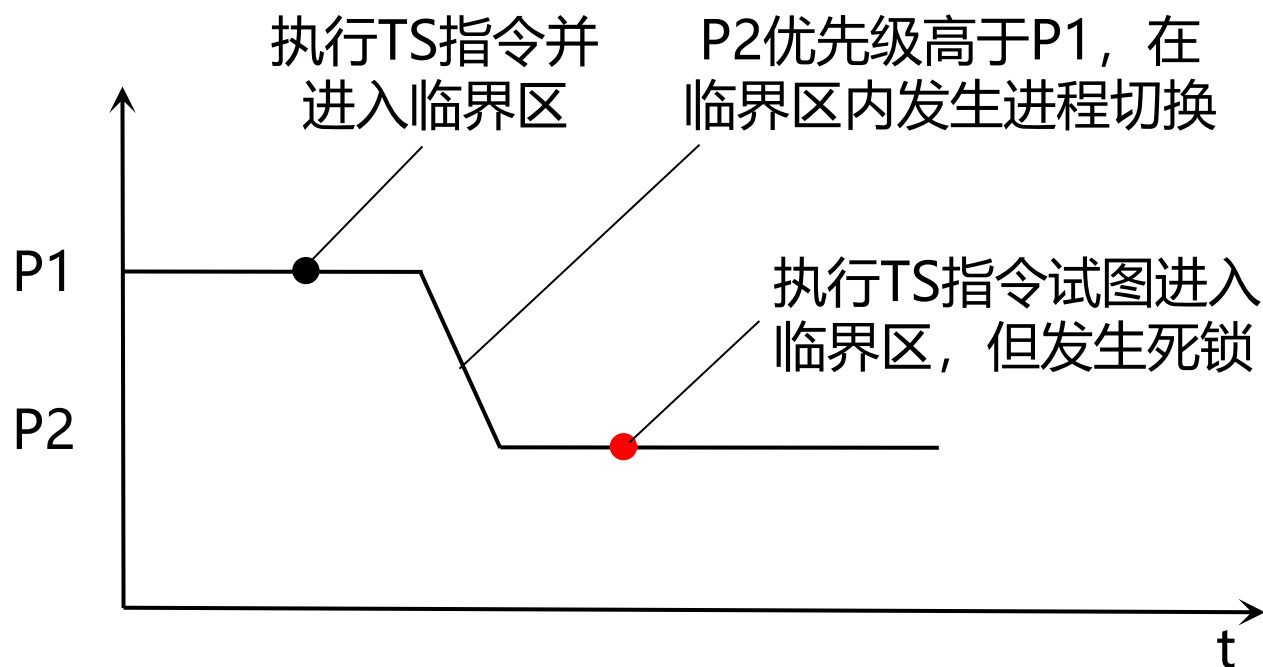
critical section;

lock=FALSE;

non critical section;

}while(TRUE);

注意：Test-and-Set指令使用“忙等待”（自旋）的方式，在单处理器下有可能导致死锁。



自旋锁在单处理器下的死锁示意图



2.3.3 信号量机制

1. 信号量

(1) 信号量用一个整型变量来累计唤醒次数。

具体使用时信号量既可以表示资源数目，也可以表示事件的发生或事件发生的次数。



2.3.3 信号量机制

1. 信号量

(2) 申请信号量的操作:

```
P (semaphore *S)
```

```
{
```



```
    S->value- -;
```

```
    if (S->value<0 )
```

```
        block (S->list); //排队策略包括FIFO、优先级
```

```
}
```





(3) 释放信号量的操作:

```
V (semaphore *S)
{
    S->value++;
    if (S->value <= 0)
        wakeup (S->list);
}
```





(4) 信号量的注意事项:

①信号量本身就是**临界资源**。在单CPU上利用**关中断**，在多CPU上利用**自旋锁**来实现信号量。

②信号量操作的时间很短，因此关中断或自旋锁带来的副作用不大。

③信号量的值可以是负数，此时它的绝对值就是等待它的进程数；另一些系统的实现中信号量的值总是 ≥ 0 。此时信号量操作变为：



```
P (semaphore *S) //申请信号量的操作。
{
    if (S->value>0)
        S->value- -;
    else
        block (S->list); //排队策略包括FIFO、优先级
}
V (semaphore *S) //释放信号量的操作。
{
    if (S->list)
        wakeup (S->list);
    else
        S->value++;
}
```

2. 信号量集

信号量集一次可操作多个信号量，并且为每个信号量指定了可分配的**下限值**和一次的**操作值**。

$\text{Swait}(S_1, t_1, d_1, \dots, S_n, t_n, d_n);$

$\text{Ssignal}(S_1, d_1, \dots, S_n, d_n);$

3. 事件组

在实时嵌入式操作系统中一般都实现了二进制的信号量集合（事件组）：

（1）允许对指定的信号量子集按“与”或者“或”的逻辑关系进行获取；

（2）对信号量（事件）值的操作指定了“消耗”或者“非消耗”的方式。



2.3.4 信号量的应用

1. 利用信号量实现资源同步（含互斥）

【例2.1】某博物馆最多可容纳 500 人同时参观，有一个出入口，该出入口一次仅允许一个人通过。参观者的活动按照进入、参观和离开的顺序进行。请使用必要的信号量和 P、V操作来描述参观博物馆的活动。



解:

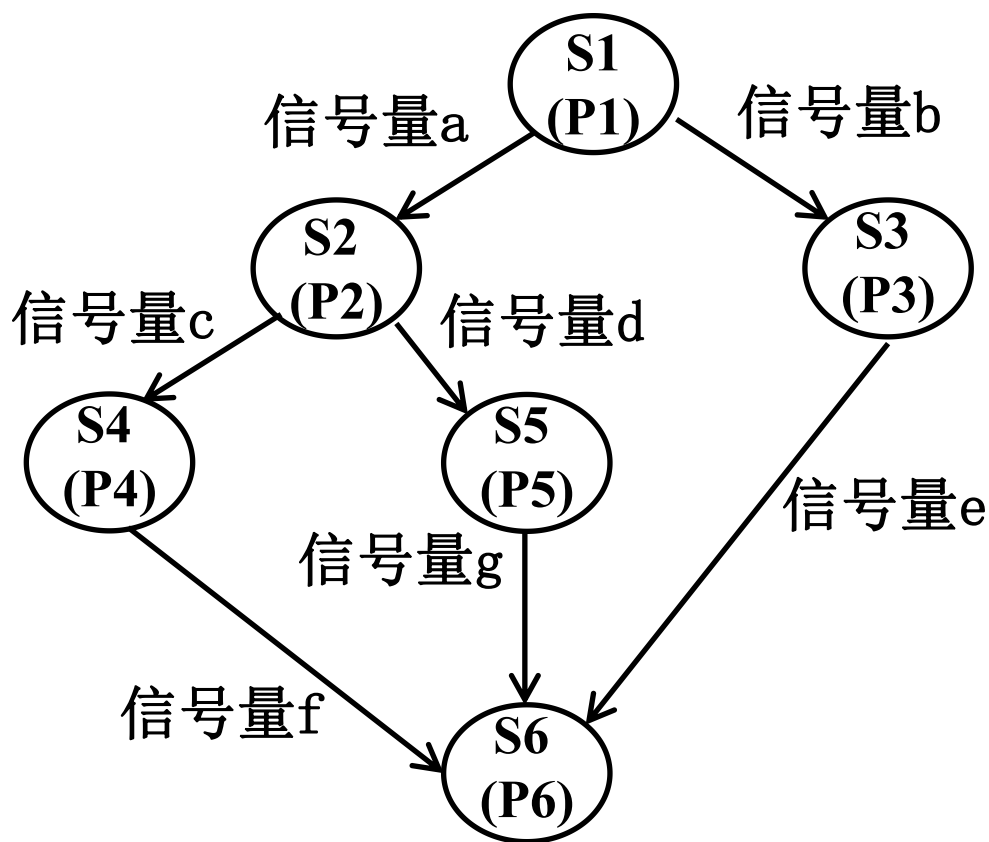
```
semaphore count-available= 500;//博物馆的空余容量  
semaphore mutex = 1; //一个出入口。
```



```
//参观者进程:
```

```
{ ...  
    P(count-available);  
    P(mutex);  
    进门;  
    V(mutex);  
    参观;  
    P(mutex);  
    出门;  
    V(mutex);  
    V(count-available);  
    ... }  
}
```

2. 利用信号量实现活动同步

【例2.2】如图所示，语句S1-S6分别位于进程P1-P6中，进程间的前驱图如下，利用信号量写出6个进程的程序。





解:

//初始值为事件没有发生。

semaphore a=0, b=0, c=0, d=0, e=0, f=0, g=0;

P1 {S1;V(a);V(b);}



P2 {P(a);S2;V(c); V(d);}

P3 {P(b);S3;V(e); }

P4 {P(c);S4;V(f);}

P5 {P(d);S5;V(g);}

P6 {P(e);P(f); P(g);S6;}





【例2.1】和【例2.2】的注意事项：

- (1) 如何设置【例2.1】和【例2.2】中信号量的初始值？
- (2) 【例2.1】和【例2.2】中资源同步和活动同步对信号量的使用方式有什么不同？





2.3.5 条件变量

1. 条件变量的使用场合

有时进程(线程)需要判断临界资源的某种条件满足后才能继续运行, 否则必须等待。

条件变量一般与互斥信号量一起使用。

2. 等待条件成立的操作

当某线程发现条件不能满足时, 将其排在条件变量的等待队列上, 并释放互斥信号量, 这两者作为一个原子操作。如pthread的pthread_cond_wait()。



等待条件成立的用法:

```
pthread_mutex_lock(&mutex);
```

```
/*关于共享数据任意复杂表达式的判断；唤醒后重新获取互斥信号量时条件可能已被其他线程改过，所以要用while。
```

```
*/
```

```
while(condition is false)
```

```
/*等待和释放互斥信号量作为一个原子操作，否则可能会造成唤醒丢失；唤醒后重新获取互斥信号量。*/
```

```
    pthread_cond_wait(&mutex, &condition) ;
```

```
do something;
```


```
pthread_mutex_unlock(&mutex);
```

3. 唤醒等待线程的操作

当另一线程改变该资源之后，唤醒条件变量等待队列中的队首线程或以广播方式唤醒所有等待线程。如pthread的pthread_cond_signal()和pthread_cond_broadcast()。

唤醒等待线程的用法：



```
pthread_mutex_lock(&mutex);  
do something;  
/*唤醒可能存在的等待线程。*/  
pthread_cond_signal(&condition);  
pthread_mutex_unlock(&mutex);
```



【例2.3】 采用两种方式访问共享资源池：

`res_t res_pool[RES_MAX]`。

- (1) 只使用互斥信号量。
- (2) 使用互斥信号量和条件变量。





解：(1) 只使用互斥信号量。

```
res_t res_pool[RES_MAX];  
int res_count = RES_MAX;  
//申请资源  
res_t res_allocate(void)  
{  
    res_t res;  
    pthread_mutex_lock(&mutex);  
    if( res_count == 0 )  
        res = RES_NONE;  
    else{  
        res_count--;  
        res = res_pool[res_count];  
        pthread_mutex_unlock(&mutex);  
        return res;  
    }  
}
```





//释放资源

```
void res_free(res_t res)
{
    pthread_mutex_lock(&mutex);
    res_pool[res_count] = res;
    res_count++;
    pthread_mutex_unlock(&mutex);
}
```

(2) 使用互斥信号量和条件变量。

```
res_t res_pool[RES_MAX];  
int res_count = RES_MAX;  
//申请资源  
res_t res_allocate(void)  
{  
    res_t res;  
    pthread_mutex_lock(&mutex);  
    while( res_count == 0 )  
        pthread_cond_wait(&mutex, &condition);  
    res_count--;  
    res = res_pool[res_count];  
    pthread_mutex_unlock(&mutex);  
    return res;  
}
```





//释放资源

```
void res_free(res_t res)
{
    pthread_mutex_lock(&mutex);
    res_pool[res_count] = res;
    res_count++;

    pthread_cond_signal(&condition);
    pthread_mutex_unlock(&mutex);
}
```

思考：两种访问共享资源池的方式哪种效率高？





2.3.6 管程机制

每个访问共享资源的进程都需要自备互斥操作，既麻烦又会因操作不当产生死锁。因此，可在编程语言的层面由编译器来实现互斥。

管程的组成（类似于面向对象中的**类**）：

- (1) 管程的名称；
- (2) 局部于管程的共享数据结构说明；
- (3) 对该数据结构进行操作的一组过程；
- (4) 对局部于管程的共享数据设置初始值的语句。



管程的作用：保证一次只有一个进程能在管程内活动，
从而提供互斥机制，保证管程数据的一致性。

管程中可以使用条件变量。




2.4 经典进程的同步问题

2.4.1 生产者-消费者问题

1. 利用信号量解决生产者-消费者问题

问题描述：假定在生产者和消费者之间的公用缓冲池中具有 n 个缓冲区。只要缓冲池未滿，生产者便可将消息送入缓冲池；只要缓冲池未空，消费者便可从缓冲池中取走一个消息。





信号量的设置分析:



(1) 利用互斥信号量**mutex**实现诸进程对缓冲池的互斥使用;



(2) 利用同步信号量**empty**和**full**分别表示缓冲池中空闲缓冲区和满缓冲区的数量。

(3) 信号量**empty**和**full**用来保证某种事件的顺序发生或不发生。在本例中, 它们保证当缓冲区满 ($\text{empty}=0$) 的时候生产者停止运行, 以及当缓冲区空 ($\text{full}=0$) 的时候消费者停止运行。这种用法与互斥是不同的。







```
解: semaphore  mutex=1, empty=n, full=0;
      int  in=0, out=0;  item  buffer[n];
void  producer( )      /*生产者进程*/
{
    while(1)
    {
        /*生产产品本身不需要在临界区*/
        produce next product;
        P(empty);    /*同步*/
        P(mutex);    /*互斥*/
        buffer[in]=product;
        in=(in+1) % n;
        V(mutex);
        V(full);     /*同步*/
    }
}
```







```
void consumer( ) /*消费者进程*/
{
    while(1)
    {
        P(full);    /*同步*/
        P(mutex);   /*互斥*/
        product =buffer[out];
        out=(out+1) % n;
        V(mutex);
        V(empty);   /*同步*/
        /*消费产品本身不需要在临界区*/
        consume the product;
    }
}
```





2. 生产者-消费者问题的注意事项



- (1) 互斥信号量的P和V操作在**一个进程**中必须成对出现。
- (2) 同步信号量的P和V操作在**不同进程**中必须成对出现。
- (3) 多个P操作的顺序不能颠倒，应先执行对同步信号量的P操作，再执行对互斥信号量的P操作，否则可能引起进程死锁。
- (4) 注意信号量初始值的设置。





2.4.2 读者-写者问题



读者-写者问题的要求：

- (1) 读写之间是互斥的；
- (2) 写者之间是互斥的；
- (3) 读者之间不是互斥的。



```
解: int readcount=0;
    semaphore  rmutex=1, wrmutex=1;
void reader( ) /*读者进程*/
{
    while (1) {
        P(rmutex);
        readcount++;
        if (readcount== 1)
            P(wrmutex);  /*仅仅第一个读者执行*/
        V(rmutex);
        READ;
        P(rmutex);
        readcount- -;
        if (readcount == 0)
            V(wrmutex); /*仅仅最后一个读者执行*/
        V (rmutex);
    }
}
```





```
void writer( ) /*写者进程*/
{
    while (1)
    {
        P(wrmutex);
        WRITE;
        V(wrmutex);
    }
}
```

2.5 进 程 通 信

信号量是一种基于共享内存的低级通信方式：

- (1) 进程间交换的信息量少，主要用于控制；
- (2) 不能实现分布式系统中不同计算机之间的通信。

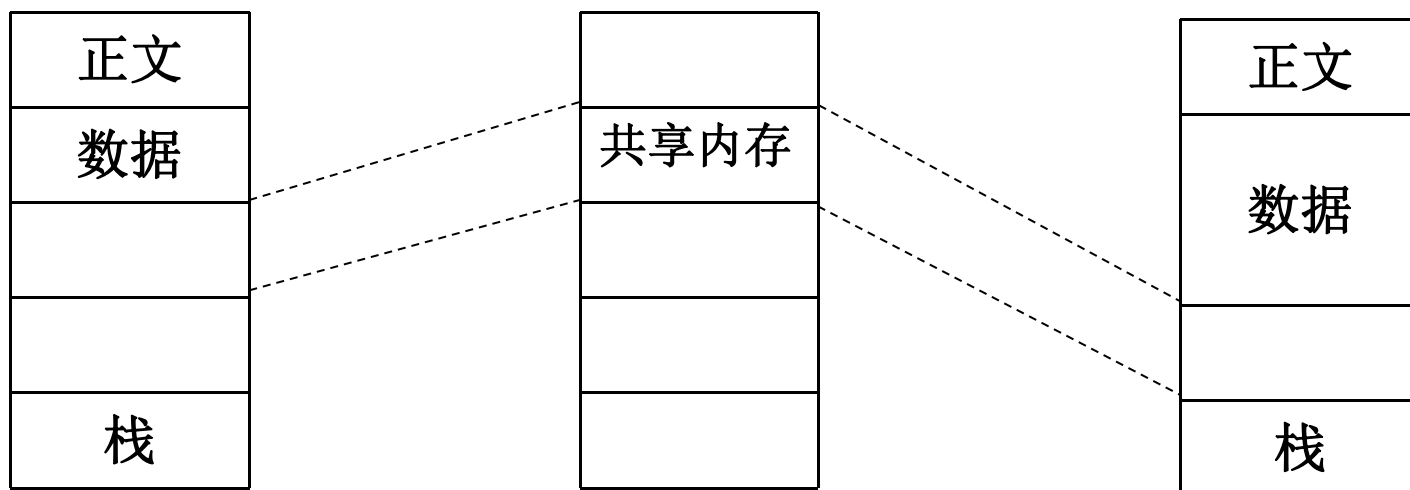
因此，我们还需要OS提供高级通信工具，其特点是：

- (1) 使用方便。通信过程对用户是透明的，这样就大大减少了通信程序编制上的复杂性；
- (2) 高效地传送大量数据。

2.5.1 进程通信的类型

1. 共享存储器系统 (Shared-Memory System)

进程A的虚空间 物理内存空间 进程B的虚空间



- (1) 共享存储区在不同进程中的虚拟地址未必一样;
- (2) 由应用进程来保证共享存储区的同步与互斥问题。



2. 管道(pipe)通信系统

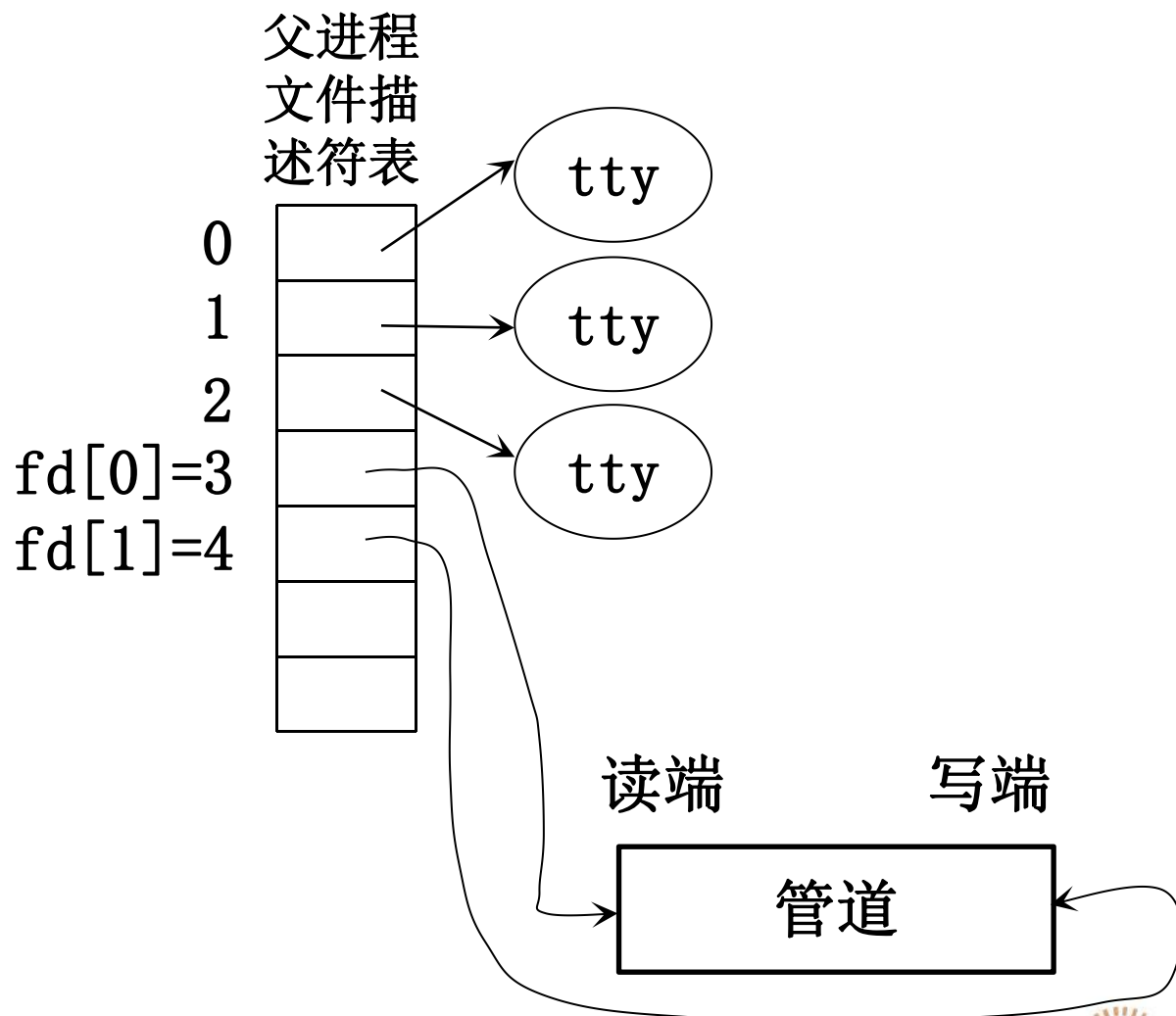
管道是连接读写进程的一个特殊文件，允许进程按先进先出方式传送数据，也能使进程同步执行操作。

(1) 无名管道：只能实现具有亲缘关系的进程间通信。

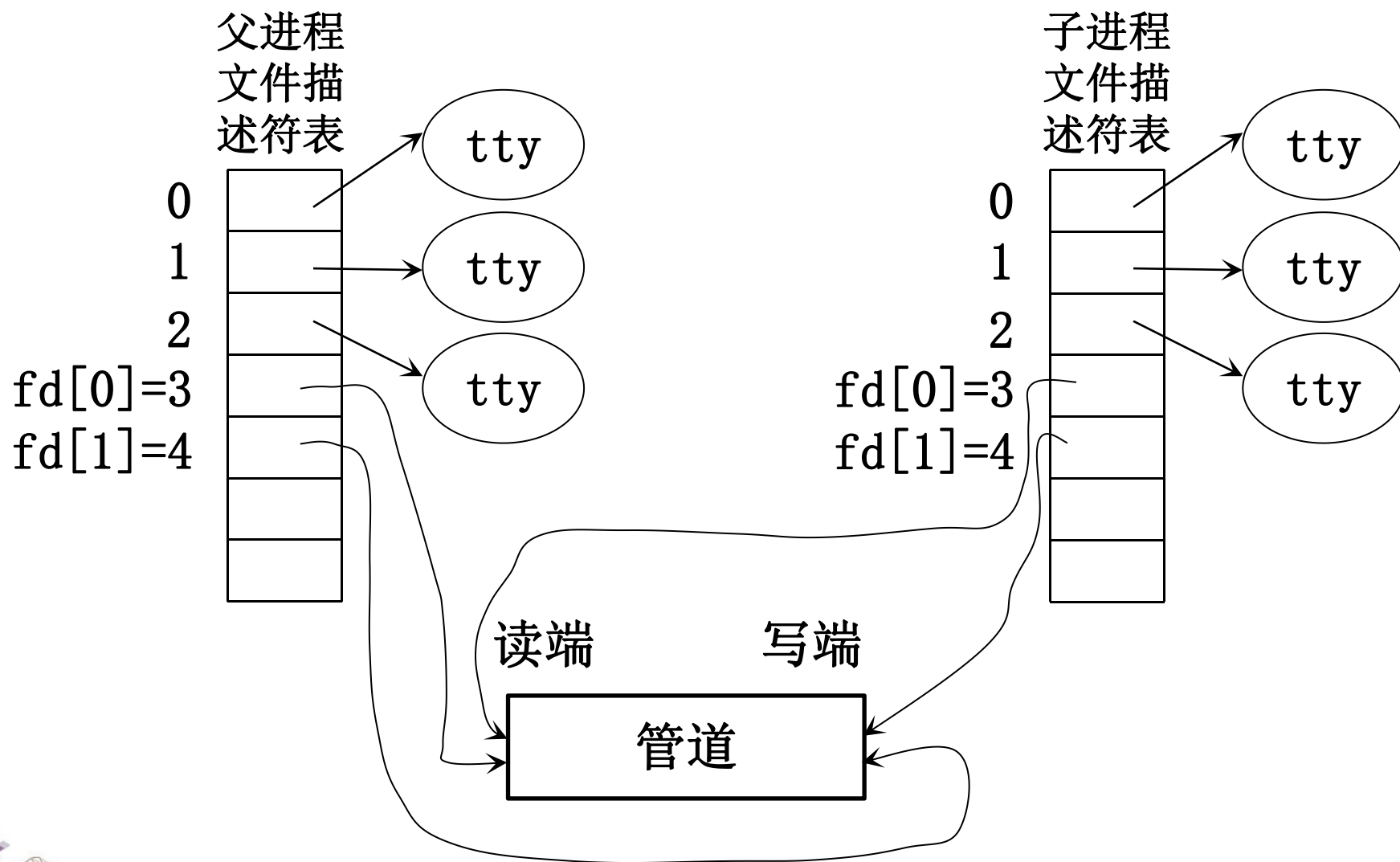
UNIX系统无名管道应用举例:

1) 父进程创建一个无名管道。

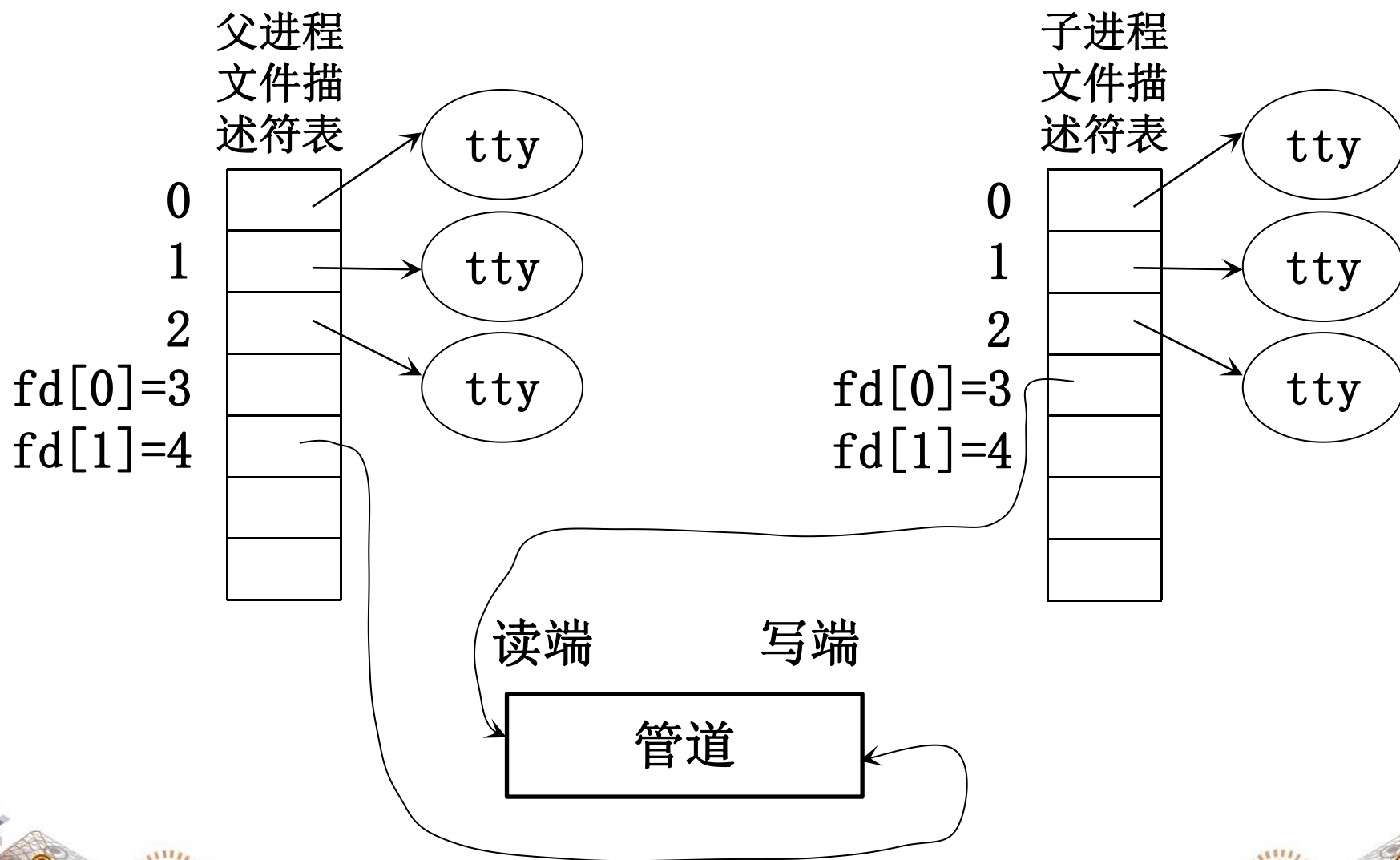
```
int fd[2];  
pipe(fd);
```





2) 父进程利用fork()创建子进程。



3) 父进程关闭fd[0]，子进程关闭fd[1]，实现了父子进程间的单工通信。





(2) 有名管道:以磁盘文件的形式存在, 可实现本机任意进程间通信。



(3) 为了协调双方的通信，管道机制必须提供以下三方面的协调能力：

1) 互斥，即当一个进程正在对**pipe**执行读/写操作时，其它进程必须等待。

2) 同步，指当写(输入)进程把一定数量的数据写入**pipe**，便去睡眠等待，直到读(输出)进程取走数据后再把它唤醒；当读进程读一空**pipe**时，也应睡眠等待，直至写进程将数据写入管道后才将之唤醒。

3) 确定对方是否存在，只有确定了对方已存在时才能进行通信。



3. 消息传递系统(Message passing system)

在单处理机系统、多处理机系统、多计算机系统和分布式系统的进程间进行消息传递，完成进程间的数据交换。

可进一步分成两类：

(1) 直接通信方式（目标进程）

(2) 间接通信方式（中间实体）

通过网络通信一般采用上述方式（2）。



4. 客户机-服务器系统 (Client-Server system)

(1) 套接字 (Socket)

用IP地址、传输层协议和端口号标识的通信端点。

(2) 远程过程调用和远程方法调用

该协议允许运行于一台主机(本地)系统上的进程调用另一台主机(远程)系统上的过程, 而对程序员表现为常规的过程调用。

如果涉及的软件采用面向对象编程, 那么远程过程调用亦可称做远程方法调用。

远程过程调用和远程方法调用对用户隐藏了显式的消息传递。

2.5.2 消息传递通信的同步方式

	同步（阻塞）	异步（无阻塞）
发送	直到消息已被接收且收到确认，控制权才返回	只是将消息复制到内核后，就返回控制。
接收	直到消息到达，控制权才返回	只是通知内核接收缓冲区的位置，就马上返回控制。

消息传递中进程的同步方式比较表

组合方式：

- (1) 发送进程阻塞，接收进程阻塞（紧耦合）；
- (2) 发送进程不阻塞，接收进程阻塞（最常用）；
- (3) 发送进程不阻塞，接收进程不阻塞。

2.5.3 直接消息传递系统实例

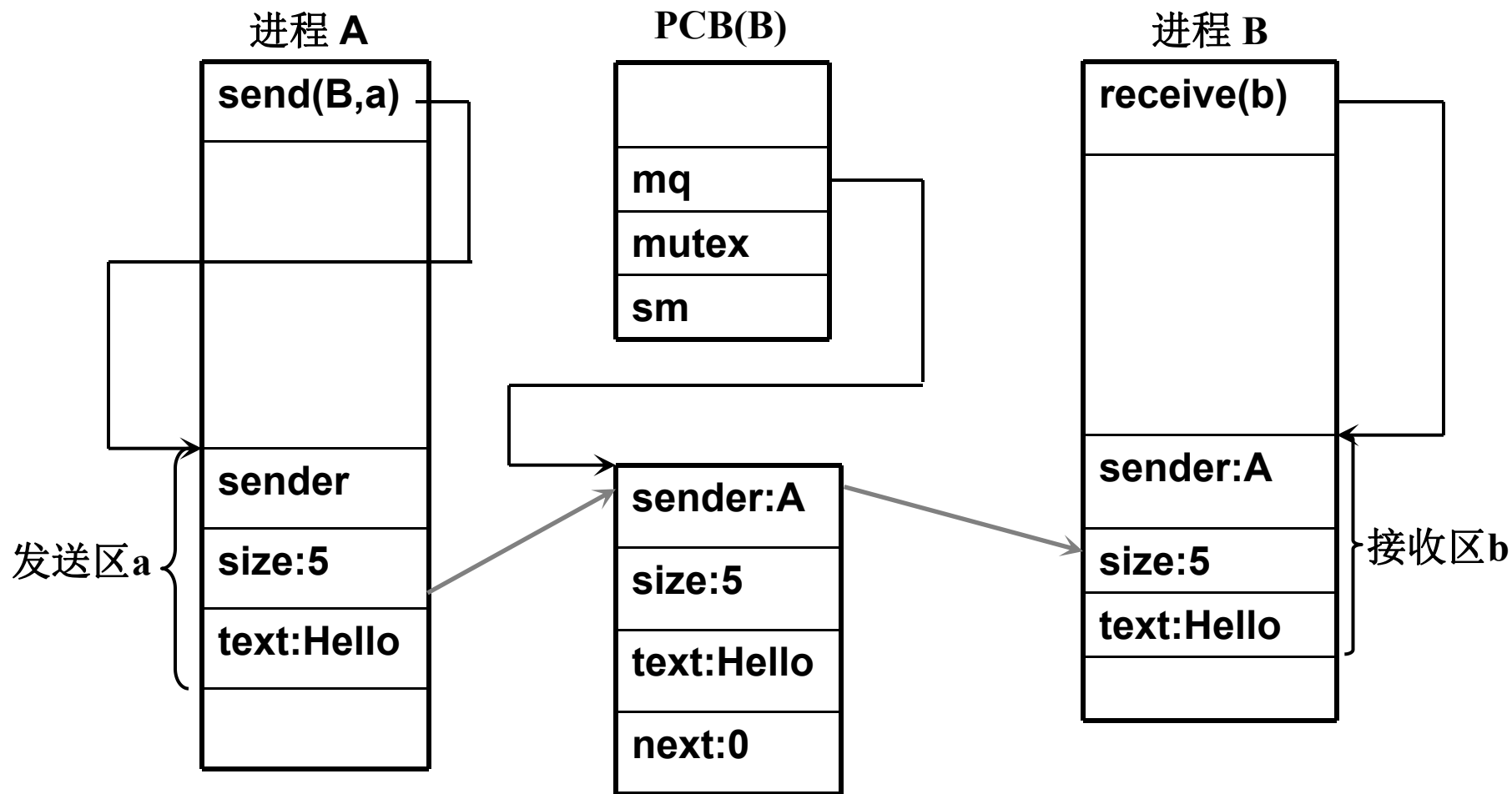


图 2-17消息缓冲通信

(1) 发送消息的实现过程如下：

```
void send(B, a) //a为用户消息buf
{
    c=getbuf( ); //获取内核消息buf
    c.sender=a.sender;
    c.size=a.size;
    copy(c.text, a.text) ;
    c.next=0;
    pcb=getpcb(B); //获取进程B的进程控制块
    P(pcb.mutex);
    insert(pcb.mq, c);
    V(pcb.mutex);
    V(pcb.sm); //通知接收进程
}
```



(2) 读取消息的实现过程如下：

```
void  receive(b) //b为用户消息buf
{
    P(pcb.sm); //是否能获取到消息
    P(pcb.mutex);
    c=remove(pcb.mq);
    V(pcb.mutex);
    b.sender=c.sender;
    b.size=c.size;
    copy(b.text, c.text);
    putbuf(c); //释放内核消息buf
}
```



2.6 线程(Threads)的基本概念

2.6.1 线程的引入

1. 进程的两个基本属性

- (1) 进程是拥有资源的基本单位。
- (2) 进程同时又是调度的基本单位。



2. 程序并发执行所需付出的时空开销

(1) 创建进程。系统在创建一个进程时，必须为它分配其所必需的、除处理机以外的所有资源，如内存空间、I/O设备，以及建立相应的PCB。

(2) 撤消进程。系统在撤消进程时，又必须先对其所占有的资源执行回收操作，然后再撤消PCB。

(3) 进程切换。对进程进行上下文切换时，需要保留当前进程的CPU环境，设置新选中进程的CPU环境，因而需花费不少的处理机时间。

(4) 进程有自己独立的地址空间，进程间交换数据需要系统调用支持。



3. 线程

如何能使多个程序更好地并发执行，同时又尽量减少系统的开销？

(1) 进程是拥有资源的基本单位。

(2) 进程内创建的线程（轻量级进程）共享进程的资源，是调度的基本单位。



2.6.2 线程的状态和线程控制块

1. 线程运行的三个状态

线程在运行时也具有下述三种基本状态：

- (1) 执行状态，表示线程已获得处理机而正在运行；
- (2) 就绪状态，指线程已具备了各种执行条件，只须再获得CPU便可立即执行；
- (3) 阻塞状态，指线程在执行中因某事件受阻而处于暂停状态。例如，当一个线程执行从键盘读入数据的系统调用时，该线程就被阻塞。



2. 线程控制块TCB

所有用于控制和管理线程的信息记录在线程控制块中。

- (1) 线程标识符;
- (2) 一组寄存器;
- (3) 线程运行状态;
- (4) 优先级;
- (5) 线程专有存储区;
- (6) 信号屏蔽;
- (7) 堆栈指针, 包括用户栈的指针和内核栈的指针。

2.7 线程的实现

(1) 在有的系统中，特别是一些数据库管理系统，如 **infomix** 所实现的是用户级线程；

(2) 而另一些系统（如Macintosh和OS/2操作系统）所实现的是内核支持线程；



(3) 还有一些系统如Solaris操作系统，则同时实现了这两种类型的线程。



1. 内核支持线程KST (Kernel Supported Threads)

内核支持线程是在内核的支持下运行的，它们的创建、阻塞、撤消和切换等，也都是在内核空间实现的。

为了对内核线程进行控制和管理，在内核空间也为每一个内核线程设置了一个线程控制块，内核根据该控制块而感知某线程的存在，并对其加以控制。



内核支持线程的实现方式主要有四个优点：

(1) 在多处理器系统中，内核能够同时调度同一进程中的多个线程并行执行；

(2) 如果进程中的一个线程被阻塞了，内核可以调度该进程中的其它线程占有处理器运行，也可以运行其它进程中的线程；

(3) 内核支持线程具有很小的数据结构和堆栈，线程的切换比较快，切换开销小；

(4) 内核本身也可以采用多线程技术，可以提高系统的执行速度和效率。

缺点：线程切换是在内核进行的，**模式切换**开销大。



2. 用户级线程ULT (User Level Threads)

用户级线程是在用户空间中实现的。对线程的创建、撤消、同步与通信等功能，都无需内核的支持，即用户级线程是与内核无关的。

由于这些线程的任务控制块都是设置在用户空间，而线程所执行的操作也无需内核的帮助，因而内核完全不知道用户级线程的存在。



用户级线程方式有许多优点：

(1) 线程切换不需要转换到内核空间。

(2) 调度算法可以是进程专用的。

(3) 用户级线程的实现与OS平台无关，因为线程管理的代码（线程库）是属于用户程序的一部分，所有的应用程序都可以对之进行共享。



用户级线程方式的主要缺点：

(1) 系统调用的阻塞问题。当线程执行一个系统调用时，不仅该线程被阻塞，而且进程内的所有线程都会被阻塞。

(2) 多线程应用不能利用多处理机进行多重处理的优点，内核每次分配给一个进程的仅有一个CPU，因此，进程中仅有一个线程能执行。

3. 组合方式

有些OS把用户级线程和内核支持线程两种方式进行组合，提供了组合方式ULT/KST 线程。在组合方式线程系统中，内核支持多个内核支持线程的建立、调度和管理，同时，也允许用户应用程序建立、调度和管理用户级线程。

本质上是一种多路复用技术。

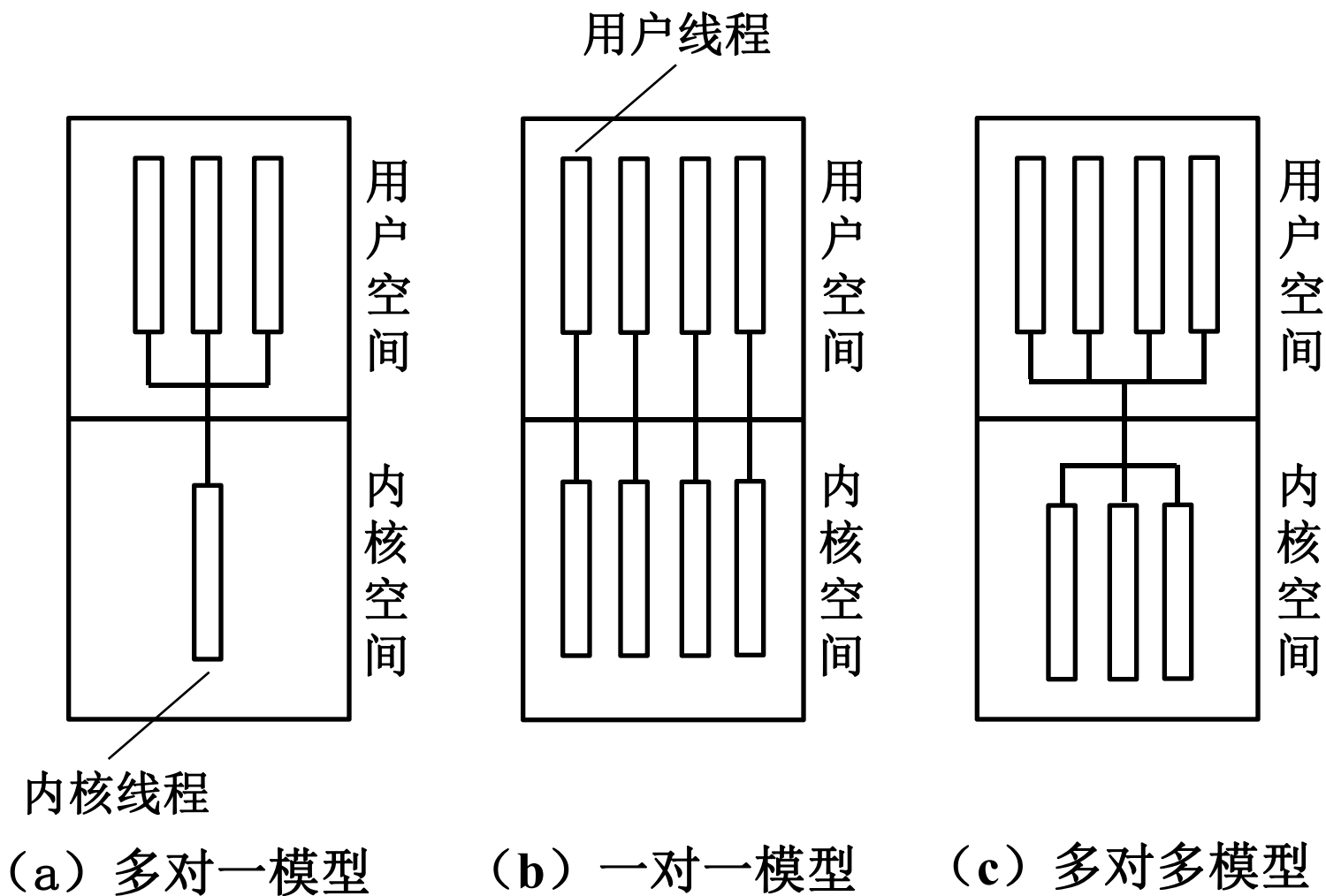
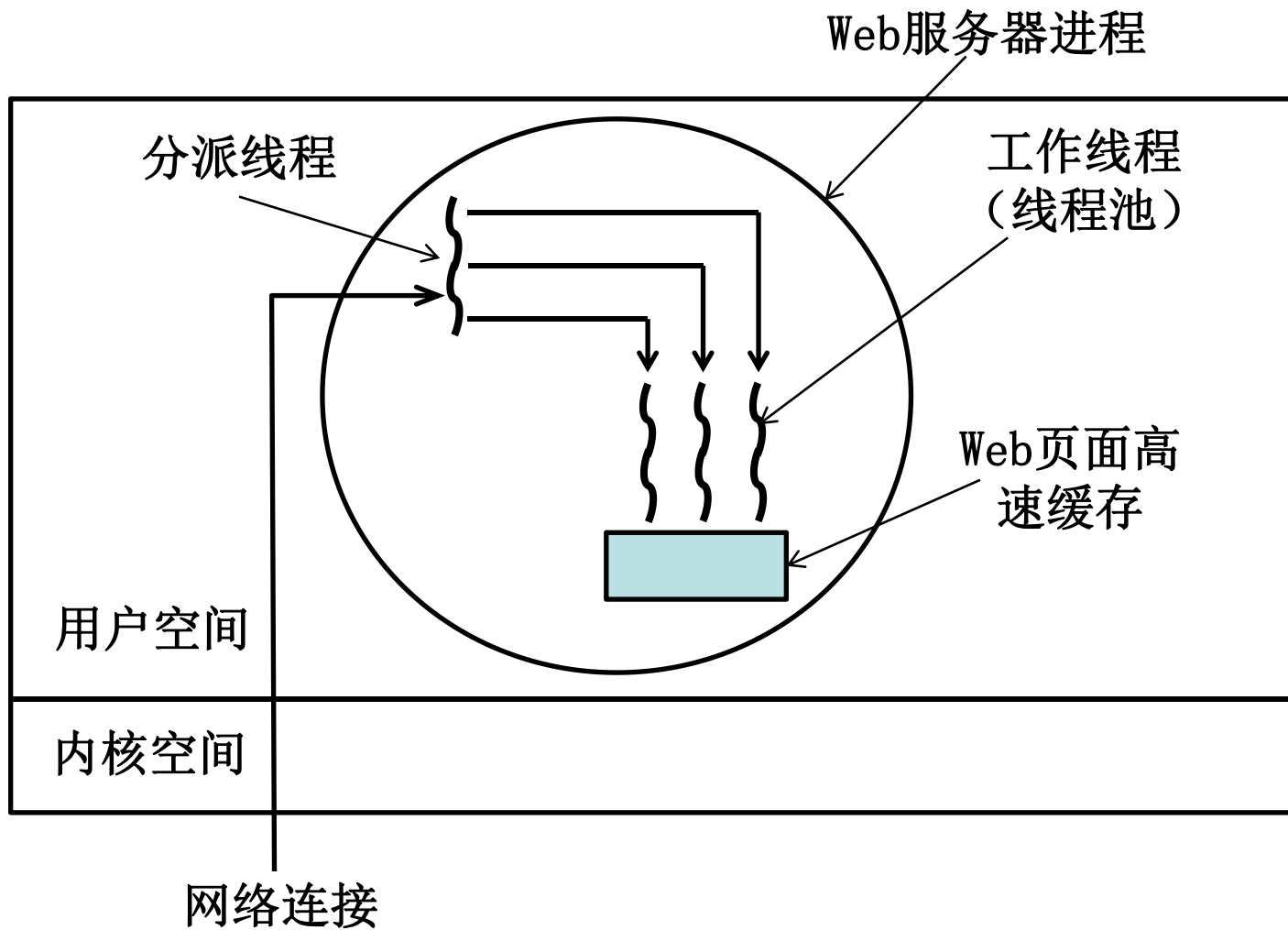


图2-18 多线程模型

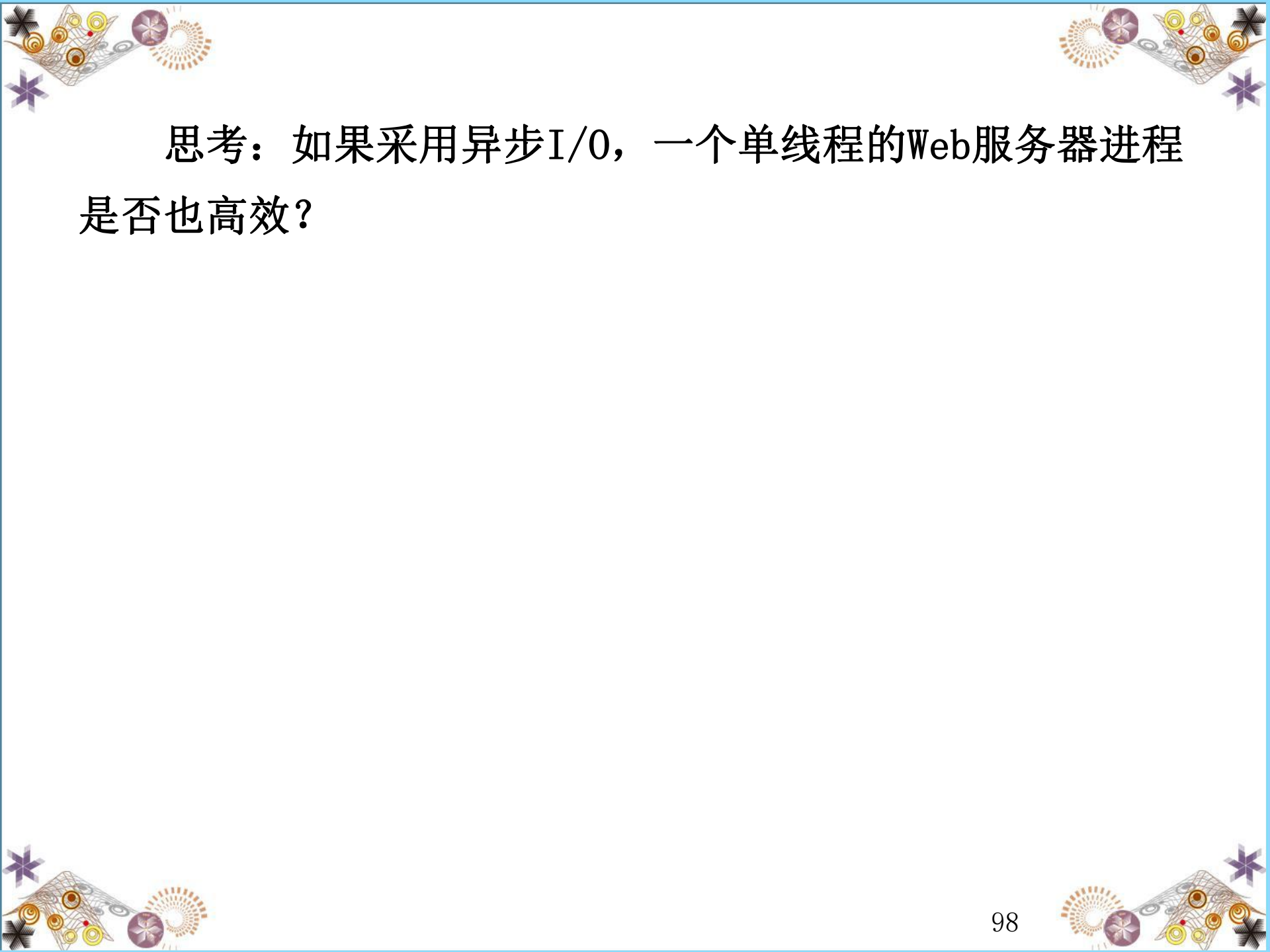


4. 多线程应用举例

采用同步I/O时，一个多线程的Web服务器进程相比于单线程的Web服务器进程更加高效。



一个多线程的Web服务器



思考：如果采用异步I/O，一个单线程的Web服务器进程是否也高效？