

2.4 经典的进程同步问题

- 生产者-消费者问题 (Producer-Consumer Problem)
- 哲学家就餐问题 (Dining-Philosophers Problem)
- 读者 - 写者问题 (Reader-Writer Problem)

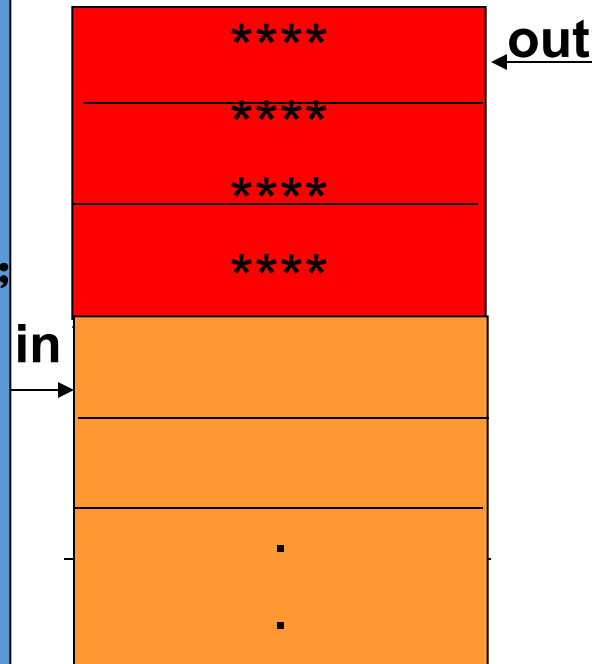
2.4.1 生产者—消费者问题

producer_i

consumer_j

producer: repeat

```
produce an item in nextp;  
while counter=n do no-op;  
buffer [in] :=nextp;  
in:=(in+1) mod n;  
counter:=counter+1;  
until false;
```



buffer

consumer: repeat

```
while counter=0 do no-op;  
nextc:=buffer [out] ;  
out:=(out+1) mod n;  
counter:=counter-1;  
consume the item in nextc;  
until false;
```

2.4.1 生产者—消费者问题

1. 利用记录型信号量解决生产者—消费者问题

□ 生产者和消费者之间应满足下列二个同步条件：

- 只有在缓冲区中至少有一个缓冲单元已存入消息后，消费者才能从中提取消息，否则消费者必须等待。
- 只有缓冲区中至少有一个缓冲单元是空时，生产者才能把消息放入缓冲区，否则生产者必须等待。

□ 生产者和消费者问题的互斥情况：

- 全互斥（一次只允许一个进程对缓冲区进行操作）
- 生产者/消费者内部对栈指针（in/out）互斥

2.4.1 生产者—消费者问题-全互斥

- Producer i ($i=1,2,\dots,k$):

repeat

produce an item;

wait(empty);

wait(mutex);

(in)=item;

in=(in+1)mod n;

signal(mutex);

signal(full);

until *false*

consumer j ($j=1,2,\dots,m$):

repeat

wait(full);

wait(mutex);

item=(out);

out=(out+1)mod n;

signal(mutex);

signal(empty);

consume the item;

until false

设互斥信号量
mutex=1,对缓冲区实现互斥;同步信号量
empty=n,代表缓冲区中的空白单元的数目;同步信号量full=0,代表缓冲区中的待取走数据单元的数目,
empty+full=n.

2.4.1 生产者—消费者问题

Var mutex, empty, full: semaphore:=1, n, 0;

buffer:array[0, ..., n-1] of item;

in, out: integer:=0, 0;

begin

parbegin

producer:

begin

repeat

...

produce an item nextp;

...

wait(empty);

wait(mutex);

buffer(in):=nextp;

in:=(in+1) mod n;

signal(mutex);

signal(full);

until false;

end

2.4.1 生产者—消费者问题

consumer:

begin

repeat

wait(full);

wait(mutex);

nextc:=buffer(out);

out:=(out+1) mod n;

signal(mutex);

signal(empty);

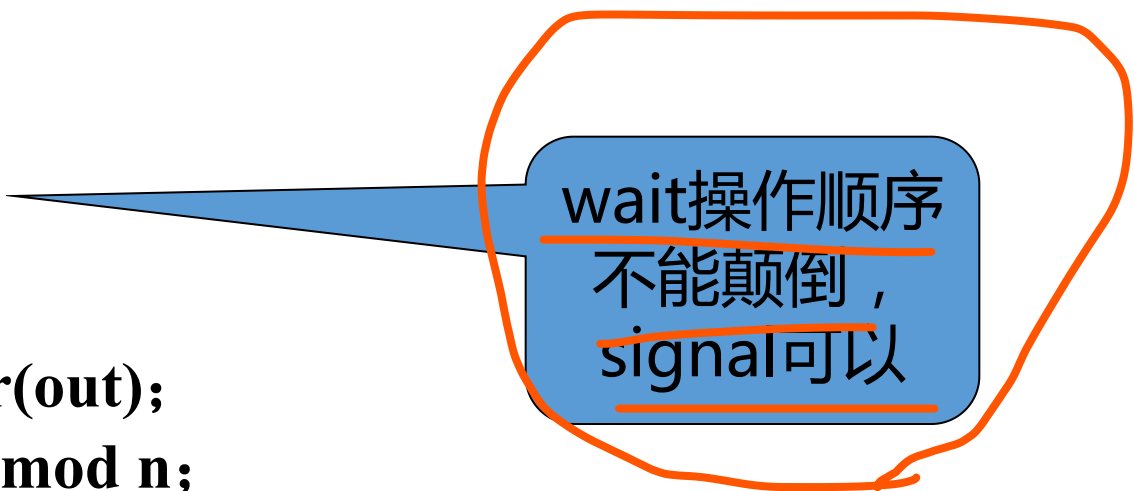
consume the item in nextc;

until false;

end

parend

end



wait操作顺序
不能颠倒,
signal可以

2.4.1 生产者—消费者问题-wait操作顺序颠倒

• Producer i ($i=1,2,\dots,k$):

repeat

produce an item;

wait(mutex);

wait(empty);

(in)=item;

in=(in+1)mod n;

signal(mutex);

signal(full);

until *false*

consumer j ($j=1,2,\dots,m$):

repeat

wait(mutex);

wait(full);

item=(out);

out=(out+1)mod n;

signal(mutex);

signal(empty);

consume the item;

until *false*

Buffer (full=n)

生产者之间以及消费者之间互斥，但生产者与消费者之间不互斥的解决方案

• Producer i ($i=1,2,\dots,k$):

repeat

produce an item;

wait(empty);

wait(mutex1);

(in)=item;

in=(in+1)mod n;

signal(mutex1);

signal(full);

until *false*

consumer j ($j=1,2,\dots,m$):

repeat

wait(full);

wait(mutex2);

item=(out);

out=(out+1)mod n;

signal(mutex2);

signal(empty);

consume the item;

until *false*

设两个互斥信号量mutex1, mutex2, 分别对in和out指针互斥

当缓冲区无限大时的解决方案

- Producer i ($i=1,2,\dots,k$):

repeat

.....

produce an item;



wait(mutex);

$(in)=item;$

$in=(in+1)\bmod n;$

signal(mutex);

signal(full);

until *false*;

consumer process:

repeat

.....

wait(full);

wait(mutex);

item=(out);

$out=(out+1)\bmod n;$

signal(mutex);



consume the item;

until false;

2.4.1 生产者—消费者问题

2 . 利用AND信号量解决生产者—消费者问题

- ① Swait(empty , mutex)来代替wait(empty)和wait(mutex) ;
- ② Ssignal(mutex , full)来代替signal(mutex)和signal(full) ;
- ③ Swait(full , mutex)来代替wait(full)和wait(mutex) ,
- ④ Ssignal(mutex , empty)代替Signal(mutex)和Signal(empty)

利用AND信号量来解决生产者—消费者问题的算法描述如下：

2.4.1 生产者—消费者问题

```
Var mutex, empty, full: semaphore:=1, n, 0;  
  buffer:array[0, ..., n-1] of item;  
  in,out: integer:=0, 0;  
begin  
  parbegin  
  
    producer: begin  
      repeat  
        produce an item in nextp;  
        Swait(empty, mutex);  
        buffer(in):=nextp;  
        in:=(in+1)mod n;  
        Ssignal(mutex, full);  
      until false;  
    end
```

2.4.1 生产者—消费者问题

```
consumer:begin
    repeat
        Swait(full, mutex);
        Nextc:=buffer(out);
        out:=(out+1) mod n;
        Ssignal(mutex, empty);
        consume the item in nextc;
    until false;
end
parend
end
```

2.4.1 生产者—消费者问题

3. 利用管程解决生产者—消费者问题

在利用管程方法来解决生产者—消费者问题时，首先是为它们建立一个管程，并命名为`ProducerConsumer`，或简称为PC。其中包括两个过程：

(1) `put(item)`过程。生产者利用该过程将自己生产的产品投放到缓冲池中，并用整型变量`count`来表示在缓冲池中已有的产品数目，当 $\text{count} \geq n$ 时，表示缓冲池已满，生产者须等待。

(2) `get(item)`过程。消费者利用该过程从缓冲池中取出一个产品，当 $\text{count} \leq 0$ 时，表示缓冲池中已无可取用的产品，消费者应等待。

2.4.1 生产者—消费者问题

PC管程可描述如下：

type PC=monitor

Var in, out, count: integer;

buffer: array[0, ..., n-1] of item;

notfull, notempty: condition;

begin in:=out:=0; count:=0; end

procedure entry put(item)

begin

if count \geq n then notfull.wait;

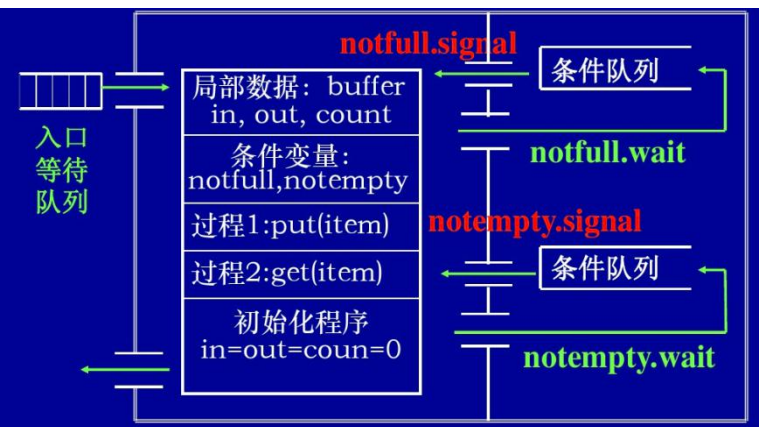
buffer(in):=nextp;

in:=(in+1) mod n;

count:=count+1;

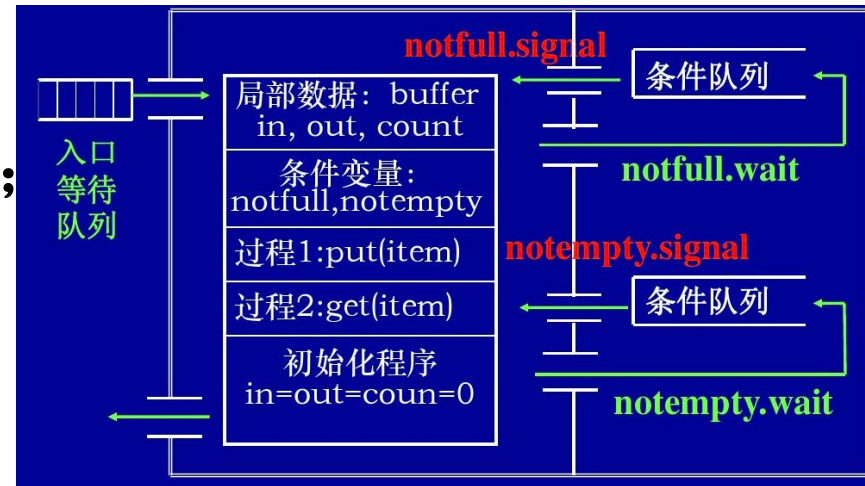
if notempty.queue then notempty.signal;

end



2.4.1 生产者—消费者问题

```
procedure entry get (item)
begin
    if count <= 0 then notempty.wait;
    nextc := buffer(out);
    out := (out + 1) mod n;
    count := count - 1;
    if notfull.queue then notfull.signal;
end
```



2.4.1 生产者—消费者问题

在利用管程解决生产者—消费者问题时，其中的生产者和消费者可描述为：

producer:

begin

repeat

produce an item in nextp;

PC.put(item);

until false;

end

consumer:

begin

repeat

PC.get(item);

consume the item in nextc;

until false;

end

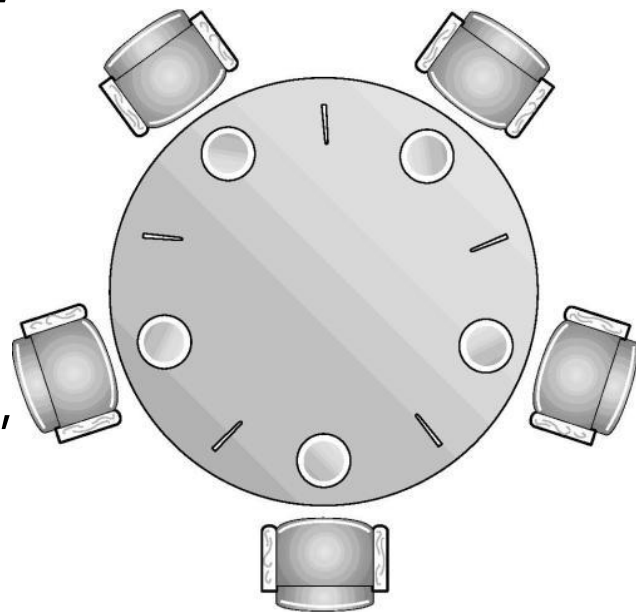
2.4.2 哲学家进餐问题

有五个哲学家，他们的生活方式是交替地进行思考和进餐。他们共用一张圆桌，分别坐在周围的五张椅子上，在圆桌上有五个碗和五支筷子，平时哲学家进行思考，饥饿时便试图取其左、右最靠近他的筷子，只有在他拿到两支筷子时才能进餐，该哲学家进餐完毕后，放下左右两只筷子又继续思考。

1. 利用记录型信号量解决哲学家进餐问题

经分析可知，放在桌子上的筷子是临界资源，在一段时间内只允许一位哲学家使用。为了实现对筷子的互斥使用，可以用一个信号量表示一只筷子，由这五个信号量构成信号量数组。其描述如下：

Var chopstick: array[0, ..., 4] of semaphore;



2.4.2 哲学家进餐问题

所有信号量均被初始化为1，第 i 位哲学家的活动可描述为：

```
repeat
    wait(chopstick[i]);
    wait(chopstick[(i+1)mod 5]);
    eating;
    ⋮
    signal(chopstick[i]);
    signal(chopstick[(i+1)mod 5]);
    thinking;
    ⋮
until false;
```

2.4.2 哲学家进餐问题

在以上描述中，当哲学家饥饿时，总是先去拿他左边的筷子，即执行`wait(chopstick[i])`；成功后，再去拿他右边的筷子，即执行`wait(chopstick[(i+1)mod 5])`；又成功后便可进餐。进餐完毕，又先放下他左边的筷子，然后再放右边的筷子。

虽然，上述解法可保证不会有两个相邻的哲学家同时进餐，但有可能引起死锁。假如五位哲学家同时饥饿而各自拿起左边的筷子时，就会使五个信号量`chopstick`均为0；当他们再试图去拿右边的筷子时，都将因无筷子可拿而无限期地等待。

对于这样的死锁问题，可采取以下几种解决方法：

2.4.2 哲学家进餐问题

(1) 至多只允许有四位哲学家同时去拿左边的筷子，最终能保证至少有一位哲学家能够进餐，并在用毕时能释放出他用过的两只筷子，从而使更多的哲学家能够进餐。

(2) 仅当哲学家的左、右两只筷子均可用时，才允许他拿起筷子进餐。

(3) 规定奇数号哲学家先拿他左边的筷子，然后再去拿右边的筷子，而偶数号哲学家则相反。按此规定，将是1、2号哲学家竞争1号筷子；3、4号哲学家竞争3号筷子。即五位哲学家都先竞争奇数号筷子，获得后，再去竞争偶数号筷子，最后总会有一位哲学家能获得两只筷子而进餐。

2.4.2 哲学家进餐问题

2 . 利用AND信号量机制解决哲学家进餐问题

在哲学家进餐问题中，要求每个哲学家先获得两个临界资源(筷子)后方能进餐，这在本质上就是前面所介绍的AND同步问题，故用AND信号量机制可获得最简洁的解法。描述如下：

```
Var chopstick array of semaphore:=(1,1,1,1,1);  
processi  
repeat  
    think;  
    Sswait(chopstick[(i+1)mod 5], chopstick[i]);  
    eating;  
    Ssignal(chopstick[(i+1)mod 5], chopstick[i]);  
until false;
```

2.4.3 读者—写者问题

条件要求：

- 写者不能同时写;
- 读者可以同时读;
- 读者读时写者不能写, 写者写时读者不能读.

2.4.3 读者—写者问题

1. 利用记录型信号量解决读者—写者问题

为实现Reader与Writer进程间在读或写时的互斥而设置了一个互斥信号量Wmutex。另外，再设置一个整型变量rcount表示正在读的进程数目。

由于只要有一个Reader进程在读，便不允许Writer进程去写。因此，仅当rcount=0，表示尚无Reader进程在读时，Reader进程才需要执行Wait(Wmutex)操作。若Wait(Wmutex)操作成功，Reader进程便可去读，相应地，做rcount+1操作。

同理，仅当Reader进程在执行了rcount减1操作后其值为0时，才须执行signal(Wmutex)操作，以便让Writer进程写。又因为rcount是一个可被多个Reader进程访问的临界资源，因此，也应该为它设置一个互斥信号量rmutex。

2.4.3 读者—写者问题

读者—写者问题可描述如下:

```
Var rmutex, wmutex: semaphore:=1,1;
```

```
    rcount: integer:=0;
```

```
    begin
```

```
        parbegin
```

```
            writer: ... ..
```

```
            reader ... ..
```

```
        parend
```

```
    end
```


Reader:

```
begin  
  repeat  
    wait(rmutex);  
    if rcount=0 then wait(wmutex);  
    rcount:=rcount+1;  
    signal(rmutex);  
  
    perform read operation;  
  
    wait(rmutex);  
    rcount:=rcount-1;  
    if rcount=0 then signal(wmutex);  
    signal(rmutex);  
  until false;  
end
```

writer:

```
begin  
  repeat  
    wait(wmutex);  
    perform write operation;  
    signal(wmutex);  
  until false;  
end
```

2.4.3 读者—写者问题

2. 利用信号量集机制解决读者—写者问题

这里的读者—写者问题与前面的略有不同，它增加了一个限制，即最多只允许RN个读者同时读。为此，又引入了一个信号量L，并赋予其初值为RN，通过执行wait(L, 1, 1)操作，来控制读者的数目。

每当有一个读者进入时，就要先执行wait(L, 1, 1)操作，使L的值减1。当有RN个读者进入读后，L便减为0，第RN+1个读者要进入读时，必然会因wait(L, 1, 1)操作失败而阻塞。对利用信号量集来解决读者—写者问题的描述如下：

2.4.3 读者—写者问题

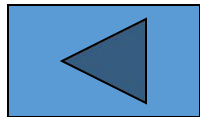
```
Var RN integer;  
    L, mx: semaphore:=RN,1;  
begin  
    parbegin  
        reader: begin  
            repeat  
                Swait(L,1,1);  
                Swait(mx,1,0);  
                :  
                perform read operation;  
                :  
                Ssignal(L,1);  
            until false;  
        end
```

2.4.3 读者—写者问题

```
writer: begin  
    repeat  
        Swait(mx,1,1; L,RN,0);  
        perform write operation;  
        Ssignal(mx,1);  
    until false;  
end  
  
parend  
  
end
```

2.4.3 读者—写者问题

其中， $\text{Swait}(mx, 1, 0)$ 语句起着开关的作用。只要无writer进程进入写， $mx=1$ ，reader进程就都可以进入读。但只要一旦有writer进程进入写时，其 $mx=0$ ，则任何reader进程就都无法进入读。 $\text{Swait}(mx, 1, 1; L, RN, 0)$ 语句表示仅当既无writer进程在写($mx=1$)，又无reader进程在读($L=RN$)时，writer进程才能进入临界区写。



2.5 进程通信

□ 进程之间有两种信息要交换：

➤ 低级通信(传递控制信息)：信号量 (semaphore)、管程 (monitor)、锁 (lock) ...

➤ 高级通信 (传递数据)：消息传递 (Message)、信箱方式 (mail box)、共享存储区 (share memory)、管道通信 (pipe line)、剪贴板 (clipboard)、套接字 (socket)...

2.5.1 进程通信的类型

1. 共享存储器系统

(1) 基于共享数据结构的通信方式：要求进程公用某些数据结构，借以实现诸进程间的信息交换。如在生产者—消费者问题中，就是用有界缓冲区这种数据结构来实现通信的。

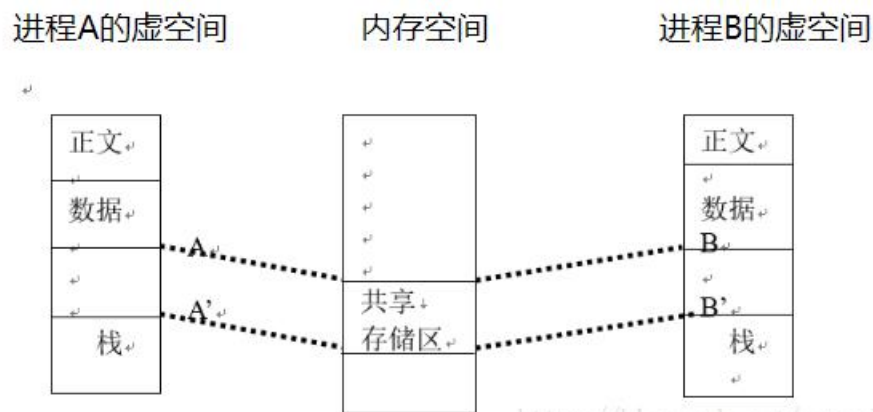
公用数据结构的设置及对进程间同步的处理，都是程序员的职责。这无疑增加了程序员的负担，而操作系统却只须提供共享存储器。因此这种通信方式是低效的，只适于传递相对少量的数据。

2.5.1 进程通信的类型

(2) 基于共享存储区的通信方式。为了传输大量数据，在存储器中划出了一块共享存储区，进程可通过对共享存储区中数据的读写来实现通信。

进程在通信前，先向系统申请获得共享存储区中的一个分区，并指定该分区的关键字；若系统已经给其他进程分配了这样的分区，则将该分区的描述符返回给申请者，继之，由申请者把获得的共享存储分区连接到本进程上；此后，便可像读、写普通存储器一样地读、写该公用存储分区。

共享存储区机制只为进程提供了用于实现通信的共享存储区和对共享存储区进行操作的手段，然而并未提供对该区进行互斥访问及进程同步的措施。因而，需要程序员自己设置同步和互斥措施才能保证实现正确的通信。



2.5.1 进程通信的类型

UNIX的共享存储区系统调用

- 创建或打开共享存储区(shmget)：依据用户给出的共享存储区的名字，创建新区或打开现有区，返回一个共享存储区ID。
- 连接共享存储区(shmat)：连接共享存储区到本进程的地址空间，可以指定虚拟地址或由系统分配，返回共享存储区首地址。父进程已连接的共享存储区可被fork创建的子进程继承。
- 拆除共享存储区连接(shmdt)：拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl)：对共享存储区进行控制。如：共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)；

2.5.1 进程通信的类型

2 . 消息传递系统

消息传递系统(Message passing system)是当前应用最为广泛的一种进程间的通信机制。在该机制中，进程间的数据交换是以格式化的消息(message)为单位的；在计算机网络中，又把message称为报文。

程序员直接利用操作系统提供的一组通信命令(原语)，不仅能实现大量数据的传递，而且还隐藏了通信的实现细节，使通信过程对用户是透明的，从而大大减化了通信程序编制的复杂性，因而获得了广泛的应用。

2.5.1 进程通信的类型

在当今流行的微内核操作系统中，微内核与服务器之间的通信，无一例外地都采用了消息传递机制。又由于它能很好地支持多处理机系统、分布式系统和计算机网络，因此它也成为这些领域最主要的通信工具。消息传递系统的通信方式属于高级通信方式。又因其实现方式的不同而进一步分成：

- 直接通信方式
- 间接通信方式

2.5.2 消息传递通信的实现方法

1. 直接通信方式

这是指发送进程利用OS所提供的发送命令，直接把消息发送给目标进程。此时，要求发送进程和接收进程都以显式方式提供对方的标识符。通常，系统提供下述两条通信命令(原语)：

`Send(Receiver, message)`；发送一个消息给接收进程；

`Receive(Sender, message)`；接收Sender发来的消息；

例如，原语`Send(P_2 , m_1)`表示将消息 m_1 发送给接收进程 P_2 ；而原语`Receive(P_1 , m_1)`则表示接收由 P_1 发来的消息 m_1 。

2.5.2 消息传递通信的实现方法

```
producer:  
  begin  
    repeat  
      produce an item in nextp;  
      ...  
      send(consumer, nextp);  
    until false;  
  end
```

```
consumer:  
  begin  
    repeat  
      receive(producer, nextc);  
      ...  
      consume the item in nextc;  
    until false;  
  end
```

2.5.2 消息传递通信的实现方法

2 . 间接通信方式

间接通信方式指进程之间的通信需要通过作为共享数据结构的实体。该实体用来暂存发送进程发送给目标进程的消息；接收进程则从该实体中取出对方发送给自己的消息。通常把这种中间实体称为信箱。消息在信箱中可以安全地保存，只允许核准的目标用户随时读取。因此，利用信箱通信方式，既可实现实时通信，又可实现非实时通信。

系统为信箱通信提供了若干条原语，分别用于信箱的创建、撤消和消息的发送、接收等。

2.5.2 消息传递通信的实现方法

(1) 信箱的创建和撤消。进程可利用信箱创建原语来建立一个新信箱。创建者进程应给出信箱名字、信箱属性(公用、私用或共享)；对于共享信箱，还应给出共享者的名字。当进程不再需要读信箱时，可用信箱撤消原语将之撤消。

2.5.2 消息传递通信的实现方法

(2) 消息的发送和接收。当进程之间要利用信箱进行通信时，必须使用共享信箱，并利用系统提供的下述通信原语进行通信：

Send(mailbox, message)；将一个消息发送到指定信箱；

Receive(mailbox, message)；从指定信箱接收一个消息；

信箱可由操作系统创建，也可由用户进程创建，创建者是信箱的拥有者。据此，可把信箱分为以下三类。

2.5.2 消息传递通信的实现方法

1) 私用信箱

用户进程建立。信箱的拥有者有权从信箱中读取消息，其他用户则只能将自己构成的消息发送到该信箱中。这种私用信箱可采用单向通信链路的信箱来实现。当拥有该信箱的进程结束时，信箱也随之消失。

2) 公用信箱

操作系统创建，并提供给系统中的所有核准进程使用。核准进程既可把消息发送到该信箱中，也可从信箱中读取发送给自己的消息。显然，公用信箱应采用双向通信链路的信箱来实现。通常，公用信箱在系统运行期间始终存在。

3) 共享信箱

用户进程创建并指明是可共享的信箱，还要给出共享此信箱的进程名。共享进程有权从信箱中读取消息。

2.5.2 消息传递通信的实现方法

□ 发送者与接收者存在以下四种关系：

- 一对一关系：一个发送进程与一个接收进程用专线进行交互。
- 多对一关系：多个客户进程与一个服务进程与交互。
- 一对多关系：允许一个发送进程与多个接收进程进行交互，使发送进程可用广播方式向接收者(多个)发送消息。
- 多对多关系：允许建立一个公用信箱，让多个进程都能向信箱中发送消息；也可从信箱中取走属于自己的消息。

2.5.3 消息传递系统实现中的若干问题

1. 通信链路

为使在发送进程和接收进程之间能进行通信，必须在两者之间建立一条通信链路(communication link)。有两种方式建立通信链路。

- 第一种方式是由发送进程在通信之前用显式的“建立连接”命令(原语)请求系统为之建立一条通信链路；在链路使用完后，也用显式方式拆除链路。这种方式主要用于计算机网络中。
- 第二种方式是发送进程无须明确提出建立链路的请求，只须利用系统提供的发送命令(原语)，系统会自动地为之建立一条链路。这种方式主要用于单机系统中。

2.5.3 消息传递系统实现中的若干问题

根据通信链路的连接方法，又可把通信链路分为两类：

- (1) 点一点连接通信链路，这时的一条链路只连接两个结点(进程)；
- (2) 多点连接链路，指用一条链路连接多个($n > 2$)结点(进程)。

2.5.3 消息传递系统实现中的若干问题

而根据通信方式的不同，则又可把链路分成两种：

- 单向通信链路：只允许发送进程向接收进程发送消息，或者相反；
- 双向通信链路：发送进程与接收进程互相发送/接受消息。

还可根据通信链路容量的不同而把链路分成两类：

- 无容量通信链路，在这种通信链路上没有缓冲区，因而不能暂存任何消息；
- 有容量通信链路，指在通信链路中设置了缓冲区，因而能暂存消息。缓冲区数目愈多，通信链路的容量愈大。

2.5.3 消息传递系统实现中的若干问题

2 . 消息的格式

在消息传递系统中所传递的消息，必须具有一定的消息格式。通常，可把一个消息分成消息头和消息正文两部分。消息头包括消息在传输时所需的控制信息，如源进程名、目标进程名、消息长度、消息类型、消息编号及发送的日期和时间；而消息正文则是发送进程实际上所发送的数据。

在某些OS中，消息采用比较短的定长消息格式，这便减少了对消息的处理和存储开销。在有的OS中，采用变长的消息格式，即进程所发送消息的长度是可变的。系统无论在处理还是在存储变长消息时，都可能会付出更多的开销，但这方便了用户。这两种消息格式各有其优缺点，故在很多系统(包括计算机网络)中，是同时都用的。

2.5.3 消息传递系统实现中的若干问题

3. 进程同步方式

在进程之间进行通信时，同样需要有进程同步机制，以使诸进程间能协调通信。不论是发送进程，还是接收进程，在完成消息的发送或接收后，都存在两种可能性，即进程或者继续发送(接收)，或者阻塞。由此，我们可得到以下三种情况：

(1) 发送进程阻塞，接收进程阻塞。这种情况主要用于进程之间紧密同步(tight synchronization)，发送进程和接收进程之间无缓冲时。这两个进程平时都处于阻塞状态，直到有消息传递时。这种同步方式称为汇合(rendezvous)。

2.5.3 消息传递系统实现中的若干问题

(2) 发送进程不阻塞，接收进程阻塞。这是一种应用最广的进程同步方式。平时，发送进程不阻塞，因而它可以尽快地把一个或多个消息发送给多个目标；而接收进程平时则处于阻塞状态，直到发送进程发来消息时才被唤醒。

例如，在服务器上通常都设置了多个服务进程，它们分别用于提供不同的服务，如打印服务。平时，这些服务进程都处于阻塞状态，一旦有请求服务的消息到达时，系统便唤醒相应的服务进程，去完成用户所要求的服务。处理完后，若无新的服务请求，服务进程又阻塞。

2.5.3 消息传递系统实现中的若干问题

(3) 发送进程和接收进程均不阻塞。这也是一种较常见的进程同步形式。平时，发送进程和接收进程都在忙于自己的事情，仅当发生某事件使它无法继续运行时，才把自己阻塞起来等待。

例如，在发送进程和接收进程之间联系着一个消息队列时，该消息队列最多能接纳 n 个消息，这样，发送进程便可以连续地向消息队列中发送消息而不必等待；接收进程也可以连续地从消息队列中取得消息，也不必等待。

只有当消息队列中的消息数已达到 n 个时，即消息队列已满，发送进程无法向消息队列中发送消息时才会阻塞；类似地，只有当消息队列中的消息数为0，接收进程已无法从消息队列中取得消息时才会阻塞。

2.5.4 消息缓冲队列通信机制

1) 消息缓冲区

在消息缓冲队列通信方式中，主要利用的数据结构是消息缓冲区。它可描述如下：

```
type message buffer=record
```

```
    sender ; 发送者进程标识符
```

```
    size   ; 消息长度
```

```
    text   ; 消息正文
```

```
    next   ; 指向下一个消息缓冲区的指针
```

```
end
```

2.5.4 消息缓冲队列通信机制

2) PCB中有关通信的数据项

在操作系统中采用了消息缓冲队列通信机制时，除了需要为进程设置消息缓冲队列外，还应在进程的PCB中增加消息队列队首指针，用于对消息队列进行操作，以及用于实现同步的互斥信号量mutex和资源信号量sm。在PCB中应增加的数据项可描述如下：

2.5.4 消息缓冲队列通信机制

type processcontrol block=record

⋮

mq ; 消息队列队首指针

mutex ; 消息队列互斥信号量

sm ; 消息队列资源信号量

⋮

end

2.5.4 消息缓冲队列通信机制

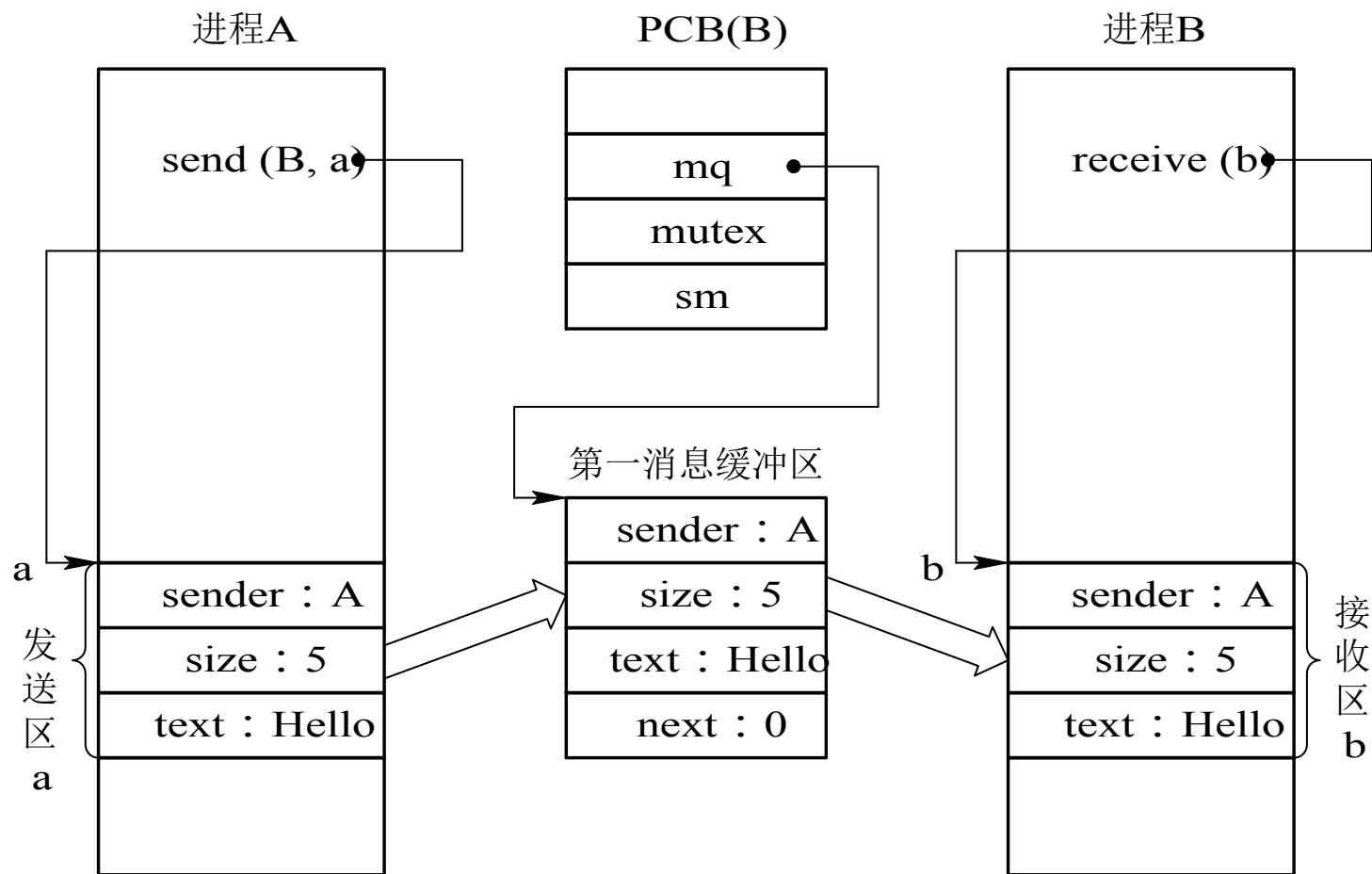


图 2-14 消息缓冲通信

2.5.4 消息缓冲队列通信机制

2. 发送原语

发送进程在利用发送原语发送消息之前，应先在自已的内存空间设置一发送区a，见图2-14所示。把待发送的消息正文、发送进程标识符、消息长度等信息填入其中，然后调用发送原语，把消息发送给目标(接收)进程。

发送原语首先根据发送区a中所设置的消息长度a size来申请一缓冲区i，接着把发送区a中的信息复制到缓冲区i中。为了能将i挂在接收进程的消息队列mq上，应先获得接收进程的內部标识符j，然后将i挂在j.mq上。由于该队列属于临界资源，故在执行insert操作的前后，都要执行wait和signal操作。

2.5.4 消息缓冲队列通信机制

发送原语可描述如下：

```
procedure send(receiver, a)
```

```
begin
```

```
    getbuf(a.size,i);      根据a.size申请缓冲区；
```

```
    i.sender:= a.sender;   将发送区a中的信息复制到消息缓冲区i中；
```

```
    i.size:=a.size;
```

```
    i.text:=a.text;
```

```
    i.next:=0;
```

```
    getid(PCB set, receiver.j); 获得接收进程内部标识符；
```

```
    wait(j.mutex);
```

```
    insert(j.mq, i);          将消息缓冲区插入消息队列；
```

```
    signal(j.mutex);
```

```
    signal(j.sm);
```

```
end
```

2.5.4 消息缓冲队列通信机制

3 . 接收原语

接收进程调用接收原语receive(b)，从自己的消息缓冲队列mq中摘下第一个消息缓冲区i，并将其中的数据复制到以b为首址的指定消息接收区内。接收原语描述如下：

2.5.4 消息缓冲队列通信机制

procedure receive(b)

begin

j:= internal name; j为接收进程内部的标识符；

wait(j.sm);

wait(j.mutex);

remove(j.mq, i); 将消息队列中第一个消息移出；

signal(j.mutex);

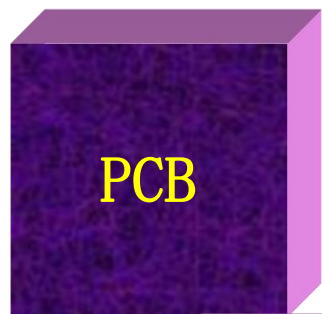
b.sender:=i.sender; 将消息缓冲区i中的信息复制到接收区b；

b.size:=i.size;

b.text:=i.text;

end

发送进程 S



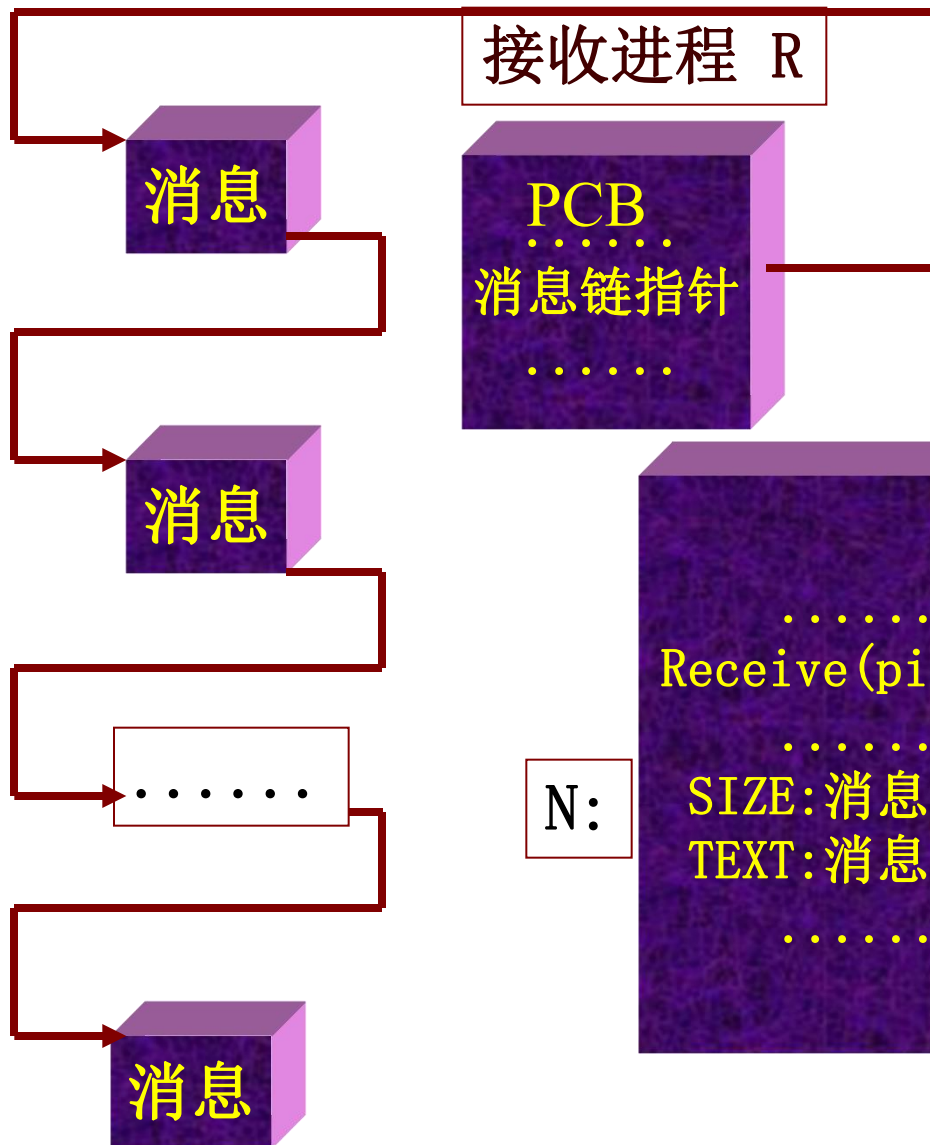
M:



接收进程 R



N:



2.5.4 消息缓冲队列通信机制

3 . 管道通信



字符流方式按先进先出顺序写入读出

所谓“管道”，是指用于连接一个读进程和一个写进程以实现它们之间通信的一个共享文件，又名pipe文件。写进程以字符流形式将大量的数据送入管道；而读进程则从管道中接收(读)数据。由于发送进程和接收进程是利用管道进行通信的，故又称为管道通信。

2.5.4 消息缓冲队列通信机制

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：

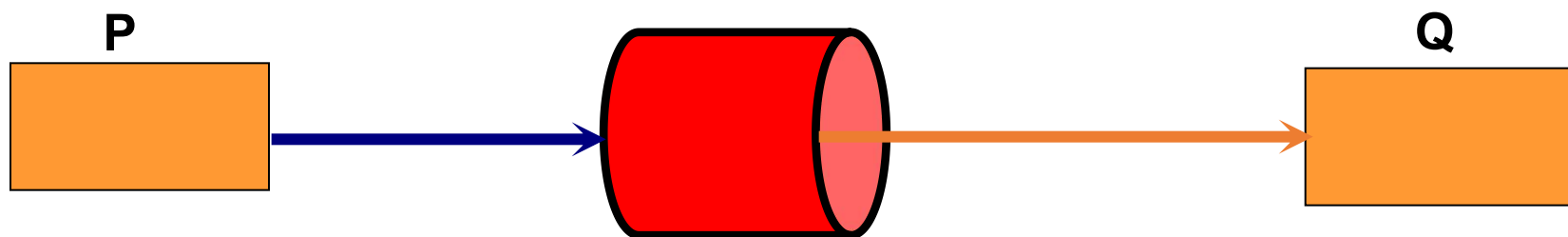
- (1) 互斥，即当一个进程正在对pipe执行读/写操作时，其它(另一)进程必须等待。
- (2) 同步，指当写(输入)进程把一定数量(如4 KB)的数据写入pipe，便去睡眠等待，直到读(输出)进程取走数据后，再把它唤醒。当读进程读一空pipe时，也应睡眠等待，直至写进程将数据写入管道后，才将之唤醒。
- (3) 确定对方是否存在，只有确定了对方已存在时，才能进行通信。



1.消息机制。适应少量信息传递，系统负责解决同步问题



2.共享区。中等信息传递，用户负责解决同步问题



3.管道。大量信息传递，系统负责解决同步问题

2.6 线程

2.6.1 线程的基本概念

1. 线程的引入

如果说，在操作系统中引入进程的目的，是为了使多个程序能并发执行，以提高资源利用率和系统吞吐量，那么，在操作系统中再引入线程，则是为了减少程序在并发执行时所付出的时空开销，使OS具有更好的并发性。

进程的两个基本属性：

- ① 进程是一个可拥有资源的独立单位；
- ② 进程同时又是一个可独立调度和分派的基本单位。

2.6.1 线程的基本概念

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。

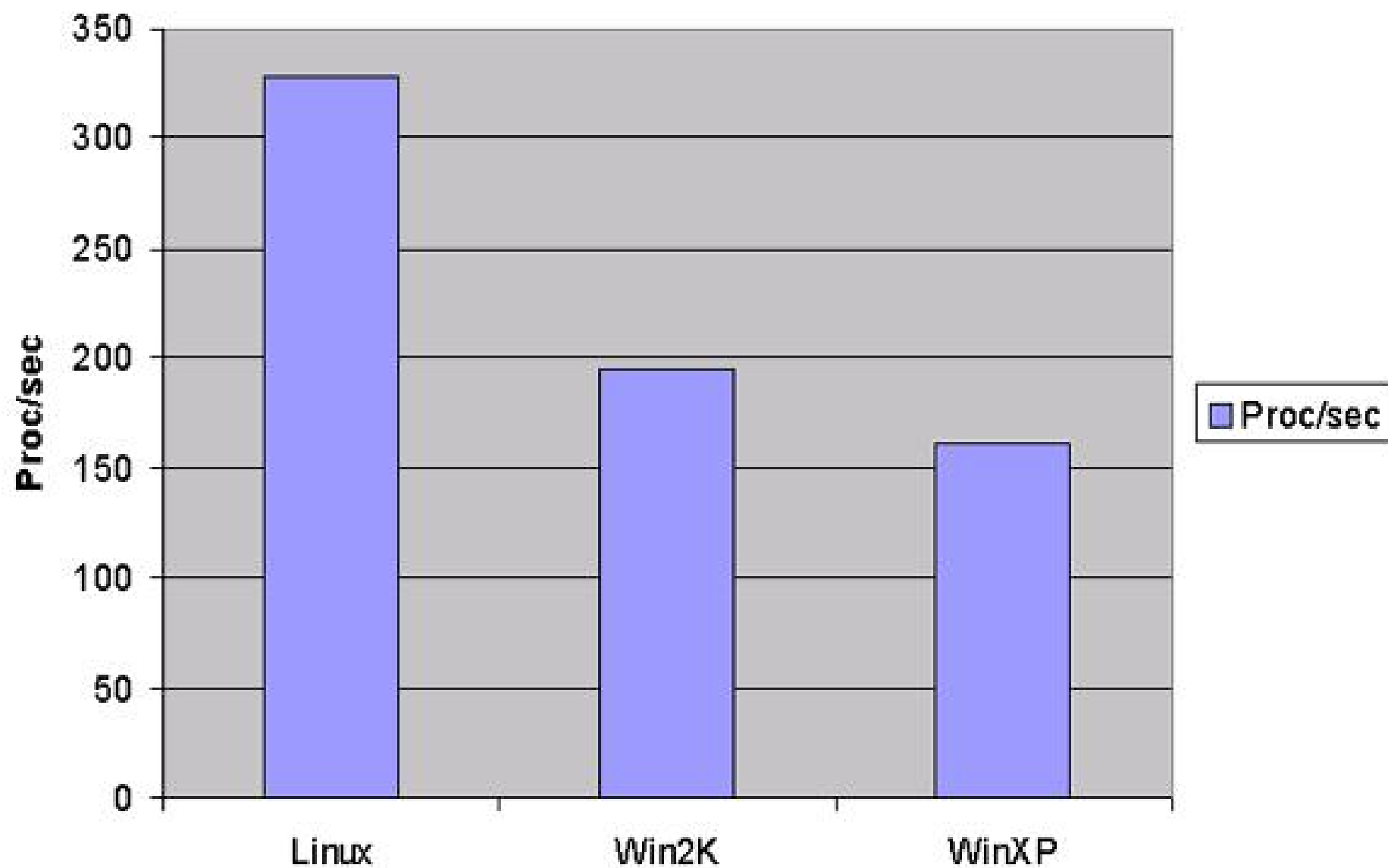
2.6.1 线程的基本概念

3) 进程切换

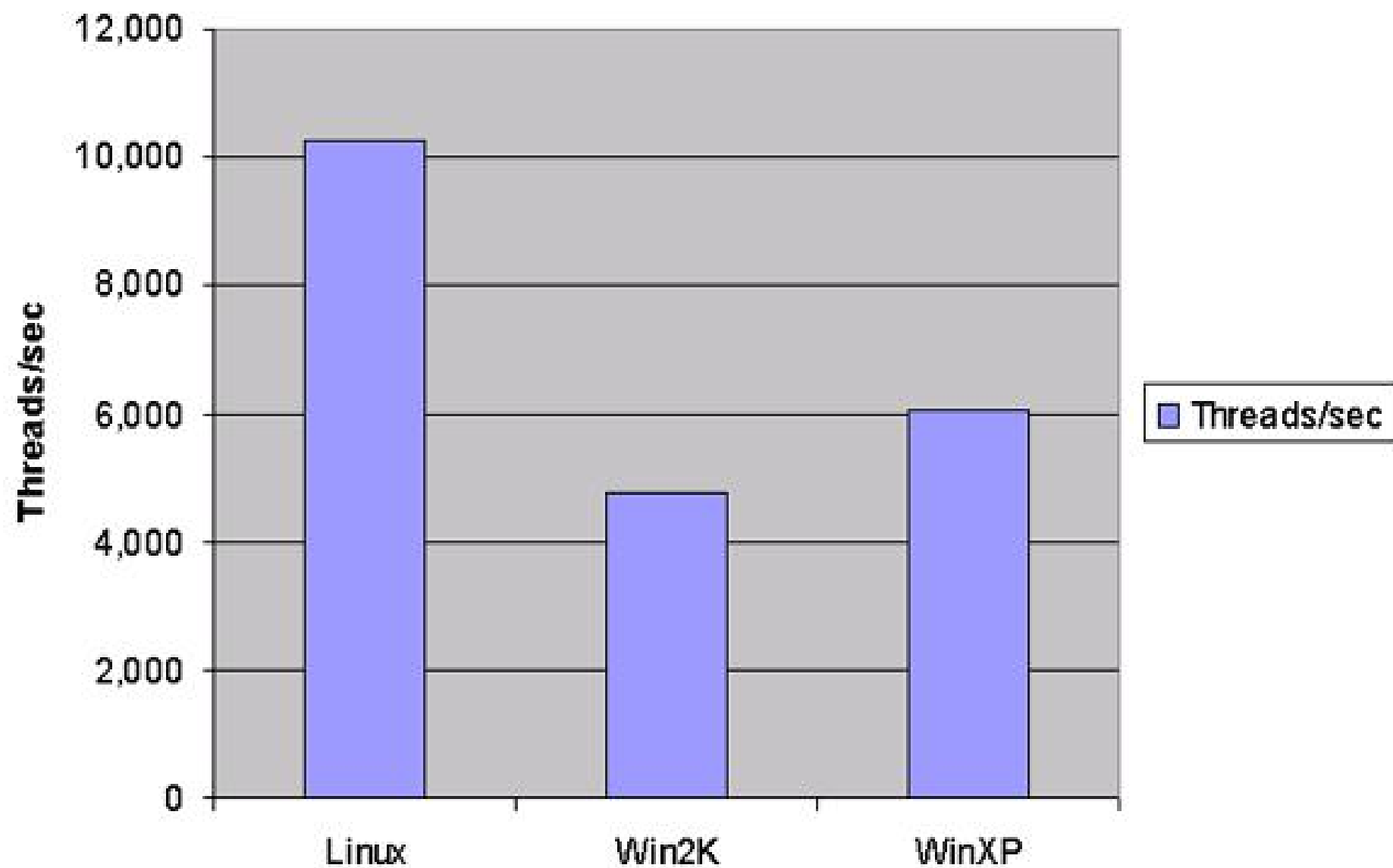
对进程进行切换时，由于要保留当前进程的CPU环境和设置新选中进程的CPU环境，因而须花费不少的处理机时间。

换言之，由于进程是一个资源的拥有者，因而在创建、撤消和切换中，系统必须为之付出较大的时空开销。正因如此，在系统中所设置的进程，其数目不宜过多，进程切换的频率也不宜过高，这也就限制了并发程度的进一步提高。

Process Create Speed (big is good)



Thread Creation Speed (big is good)

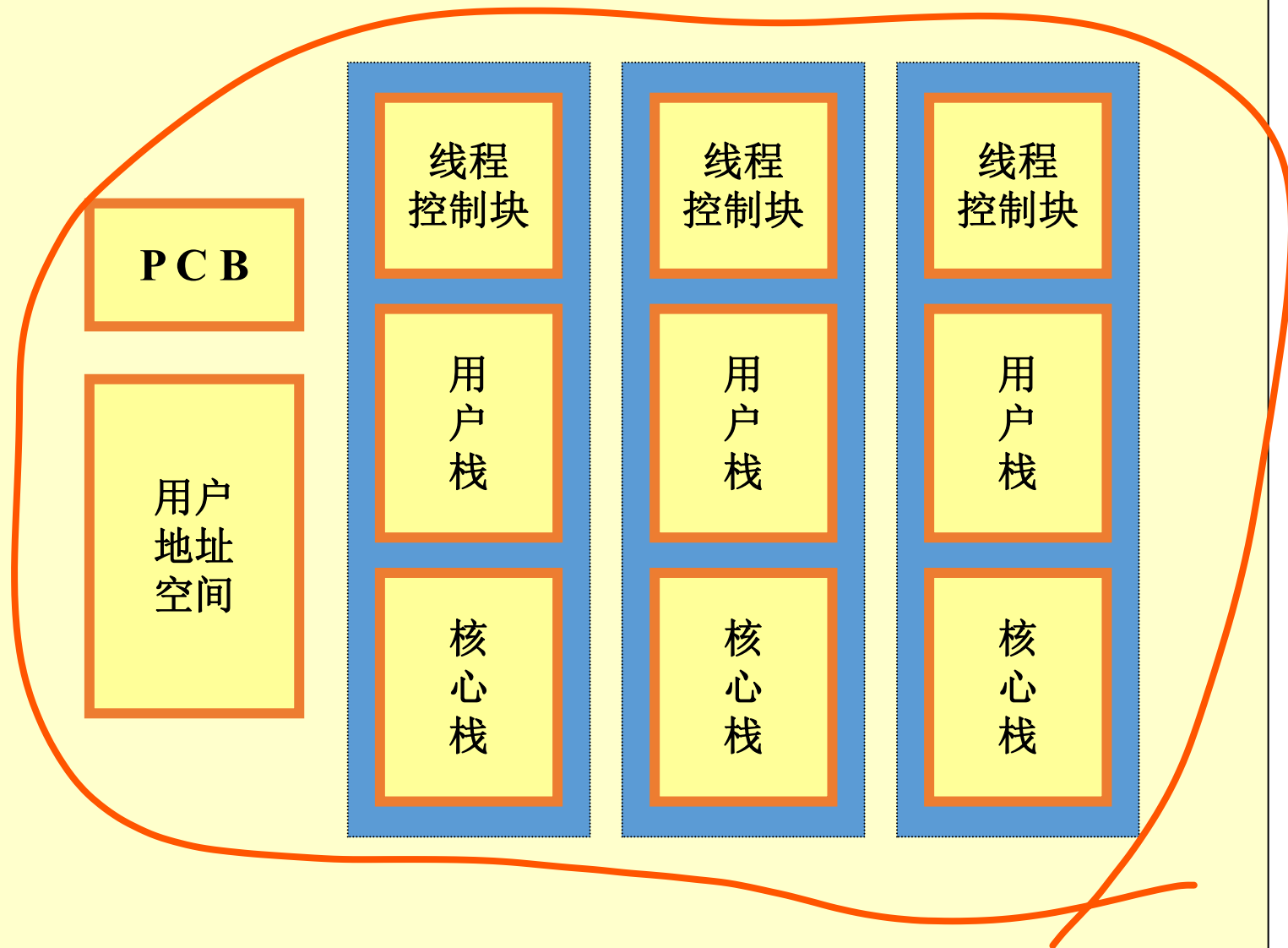


2.6.1 线程的基本概念

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，是设计操作系统时所追求的重要目标。

有不少研究操作系统的学者们想到，若能将进程的上述两个属性分开，由操作系统分开处理，亦即对于作为调度和分派的基本单位，不同时作为拥有资源的单位，以做到“轻装上阵”；而对于拥有资源的基本单位，又不对之进行频繁的切换。正是在这种思想的指导下，形成了线程的概念。

多线程进程模型



2.6.1 线程的基本概念

2. 线程与进程的比较

1) 调度

在传统的操作系统中，作为拥有资源的基本单位和独立调度、分派的基本单位都是进程。

而在引入线程的操作系统中，则把线程作为调度和分派的基本单位，而进程作为资源拥有的基本单位，把传统进程的两个属性分开，使线程基本上不拥有资源，这样线程便能轻装前进，从而可显著地提高系统的并发程度。

在同一进程中，线程的切换不会引起进程的切换，但从一个进程中的线程切换到另一个进程中的线程时，将会引起进程的切换。

2.6.1 线程的基本概念

2) 并行性

在引入线程的操作系统中，不仅进程之间可以并发执行，而且在一个进程中的多个线程之间亦可并发执行，使得操作系统具有更好的并行性，从而能更加有效地提高系统资源的利用率和系统的吞吐量。

例如，在一个未引入线程的单CPU操作系统中，若仅设置一个文件服务进程，当该进程由于某种原因而被阻塞时，便没有其它的文件服务进程来提供服务。在引入线程的操作系统中，则可以在一个文件服务进程中设置多个服务线程。

2.6.1 线程的基本概念

3) 拥有资源

不论是传统的操作系统，还是引入了线程的操作系统，进程都可以拥有资源，是系统中拥有资源的一个基本单位。

一般而言，线程自己不拥有系统资源(仅有一点必不可少的资源)，但它可以访问其隶属进程的资源，即一个进程的代码段、数据段及所拥有的系统资源，如已打开的文件、I/O设备等，可以供该进程中的所有线程所共享。

2.6.1 线程的基本概念

4) 系统开销

在创建或撤消进程时，系统都要为之创建和回收进程控制块，分配或回收资源，如内存空间和I/O设备等，操作系统所付出的开销明显大于线程创建或撤消时的开销。类似地，在进程切换时，涉及到当前进程CPU环境的保存及新被调度运行进程的CPU环境的设置，而线程的切换则仅需保存和设置少量寄存器内容，不涉及存储器管理方面的操作，所以就切换代价而言，进程也是远高于线程的。

此外，由于一个进程中的多个线程具有相同的地址空间，在同步和通信的实现方面线程也比进程容易。

2.6.1 线程的基本概念

3 . 线程的属性

在多线程OS中，通常是在一个进程中包括多个线程，每个线程都是作为利用CPU的基本单位，是花费最小开销的实体。线程具有下述属性。

(1) 轻型实体。线程中的实体基本上不拥有系统资源，只有一点必不可少的、能保证其独立运行的资源，比如，在每个线程中都应具有一个用于控制线程运行的线程控制块TCB，用于指示被执行指令序列的程序计数器，保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。

2.6.1 线程的基本概念

(2) 独立调度和分派的基本单位。在多线程OS中，线程是能独立运行的基本单位，因而也是独立调度和分派的基本单位。由于线程很“轻”，故线程的切换非常迅速且开销小。

(3) 可并发执行。在一个进程中的多个线程之间可以并发执行，甚至允许在一个进程中的所有线程都能并发执行；同样，不同进程中的线程也能并发执行。

(4) 共享进程资源。在同一进程中的各个线程都可以共享该进程所拥有的资源，这首先表现在所有线程都具有相同的地址空间(进程的地址空间)。这意味着线程可以访问该地址空间中的每一个虚地址；此外，还可以访问进程所拥有的已打开文件、定时器、信号量机构等。

2.6.1 线程的基本概念

4 . 线程的状态

(1) **状态参数**。在OS中的每一个线程都可以利用线程标识符和一组状态参数进行描述。状态参数通常有这样几项：

- ① 寄存器状态，它包括程序计数器PC和堆栈指针中的内容；
- ② 堆栈，在堆栈中通常保存有局部变量和返回地址；
- ③ 线程运行状态，用于描述线程正处于何种运行状态；
- ④ 优先级，描述线程执行的优先程度；
- ⑤ 线程专有存储器，用于保存线程自己的局部变量拷贝；
- ⑥ 信号屏蔽，即对某些信号加以屏蔽。

2.6.1 线程的基本概念

(2) **线程运行状态**。如同传统的进程一样，在各线程之间也存在着共享资源和相互合作的制约关系，致使线程在运行时也具有间断性。相应地，线程在运行时也具有下述三种基本状态：

- ① 执行状态，表示线程正获得处理机而运行；
- ② 就绪状态，指线程已具备了各种执行条件，一旦获得CPU便可执行的状态；
- ③ 阻塞状态，指线程在执行中因某事件而受阻，处于暂停执行时的状态。

2.6.1 线程的基本概念

5. 线程的创建和终止

在多线程OS环境下，应用程序在启动时，通常仅有一个线程在执行，该线程被人们称为“初始化线程”。它可根据需要再去创建若干个线程。

在创建新线程时，需要利用一个线程创建函数(或系统调用)，并提供相应的参数，如指向线程主程序的入口指针、堆栈的大小，以及用于调度的优先级等。在线程创建函数执行完后，将返回一个线程标识符供以后使用。

2.6.1 线程的基本概念

6 . 多线程OS中的进程

在多线程OS中，进程是作为拥有系统资源的基本单位，通常的进程都包含多个线程并为它们提供资源，但此时的进程就不再作为一个执行的实体。多线程OS中的进程有以下属性：

(1) **作为系统资源分配的单位**。在多线程OS中，仍是将进程作为系统资源分配的基本单位，在任一进程中所拥有的资源包括受到分别保护的用户地址空间、用于实现进程间和线程间同步和通信的机制、已打开的文件和已申请到的I/O设备。

2.6.1 线程的基本概念

(2) 可包括多个线程。通常，一个进程都含有多个相对独立的线程，其数目可多可少，但至少也要有一个线程，由进程为这些线程提供资源及运行环境，使这些线程可并发执行。在OS中的所有线程都只能属于某一个特定进程。

(3) 进程不是一个可执行的实体。在多线程OS中，是把线程作为独立运行的基本单位，所以此时的进程已不再是一个可执行的实体。虽然如此，进程仍具有与执行相关的状态。例如，所谓进程处于“执行”状态，实际上是指该进程中的某线程正在执行。此外，对进程所施加的与进程状态有关的操作，也对其线程起作用。例如，在把某个进程挂起时，该进程中的所有线程也都将被挂起；又如，在把某进程激活时，属于该进程的所有线程也都将被激活。

2.6.3 线程的实现方式

1. 内核支持线程

对于通常的进程，无论是系统进程还是用户进程，进程的创建、撤消，以及要求由系统设备完成的I/O操作，都是利用系统调用而进入内核，再由内核中的相应处理程序予以完成的。进程的切换同样是在内核的支持下实现的。

2.6.3 线程的实现方式

这里所谓的内核支持线程KST(Kernel Supported Threads)，也都同样是在内核的支持下运行的，即无论是用户进程中的线程，还是系统进程中的线程，它们的创建、撤消和切换等也是依靠内核，在内核空间实现的。此外，在内核空间还为每一个内核支持线程设置了一个线程控制块，内核是根据该控制块而感知某线程的存在，并对其加以控制。

这种线程实现方式主要有如下四个优点：

(1) 在多处理器系统中，内核能够同时调度同一进程中多个线程并行执行；

2.6.3 线程的实现方式

(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) 内核本身也可采用多线程技术，可以提高系统的执行速度和效率。

内核支持线程的主要缺点是：对于用户的线程切换而言，其模式切换的开销较大，在同一个进程中，从一个线程切换到另一个线程时，需要从用户态转到内核态进行，这是因为用户进程的线程在用户态运行，而线程调度和管理是在内核实现的，系统开销较大。

2.6.3 线程的实现方式

2. 用户级线程

用户级线程ULT(User Level Threads)仅存在于用户空间中。对于这种线程的创建、撤消、线程之间的同步与通信等功能，都无须利用系统调用来实现。对于用户级线程的切换，通常发生在一个应用进程的诸多线程之间，这时，也同样无须内核的支持。由于切换的规则远比进程调度和切换的规则简单，因而使线程的切换速度特别快。

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须内核的帮助，因而内核完全不知道用户级线程的存在。

2.6.3 线程的实现方式

值得说明的是，对于设置了用户级线程的系统，其调度仍是以进程为单位进行的。在采用轮转调度算法时，各个进程轮流执行一个时间片，这对诸进程而言似乎是公平的。但假如在进程A中包含了一个用户级线程，而在另一个进程B中含有100个用户级线程，这样，进程A中线程的运行时间将是进程B中各线程运行时间的100倍；相应地，其速度要快上100倍。

假如系统中设置的是内核支持线程，则调度便是以线程为单位进行的。在采用轮转法调度时，是各个线程轮流执行一个时间片。同样假定进程A中只有一个内核支持线程，而在进程B中有100个内核支持线程。此时进程B可以获得的CPU时间是进程A的100倍，且进程B可使100个系统调用并发工作。

2.6.3 线程的实现方式

使用用户级线程方式有许多优点，主要表现在如下三个方面：

(1) 线程切换不需要转换到内核空间，对于一个进程而言，其所有线程的管理数据结构均在该进程的用户空间中，管理线程切换的线程库也在用户地址空间运行。因此，进程不必切换到内核方式来做线程管理，从而节省了模式切换的开销，也节省了内核的宝贵资源。

(2) 调度算法可以是进程专用的。在不干扰操作系统调度的情况下，不同的进程可以根据自身需要，选择不同的调度算法对自己的线程进行管理和调度，而与操作系统的低级调度算法是无关的。

2.6.3 线程的实现方式

(3) 用户级线程的实现与操作系统平台无关，因为对于线程管理的代码是在用户程序内的，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。因此，用户级线程甚至可以在不支持线程机制的操作系统平台上实现。

2.6.3 线程的实现方式

用户级线程实现方式的主要缺点在于如下两个方面：

(1) **系统调用的阻塞问题**。在基于进程机制的操作系统中，大多数系统调用将阻塞进程，因此，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。而在内核支持线程方式中，则进程中的其它线程仍然可以运行。

(2) **在单纯的用户级线程实现方式中，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，在该线程放弃CPU之前，其它线程只能等待。

2.6.3 线程的实现方式

3 . 组合方式

有些操作系统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。

1. 在组合方式线程系统中，内核支持多KST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。
2. 一些内核支持线程对应多个用户级线程，程序员可按应用需要和机器配置对内核支持线程数目进行调整，以达到较好的效果。
3. 组合方式线程中，同一个进程内的多个线程可以同时多处理器上并行执行，而且在阻塞一个线程时，并不需要将整个进程阻塞。
4. 所以，组合方式多线程机制能够结合KST和ULT两者的优点，并克服了其各自的不足。

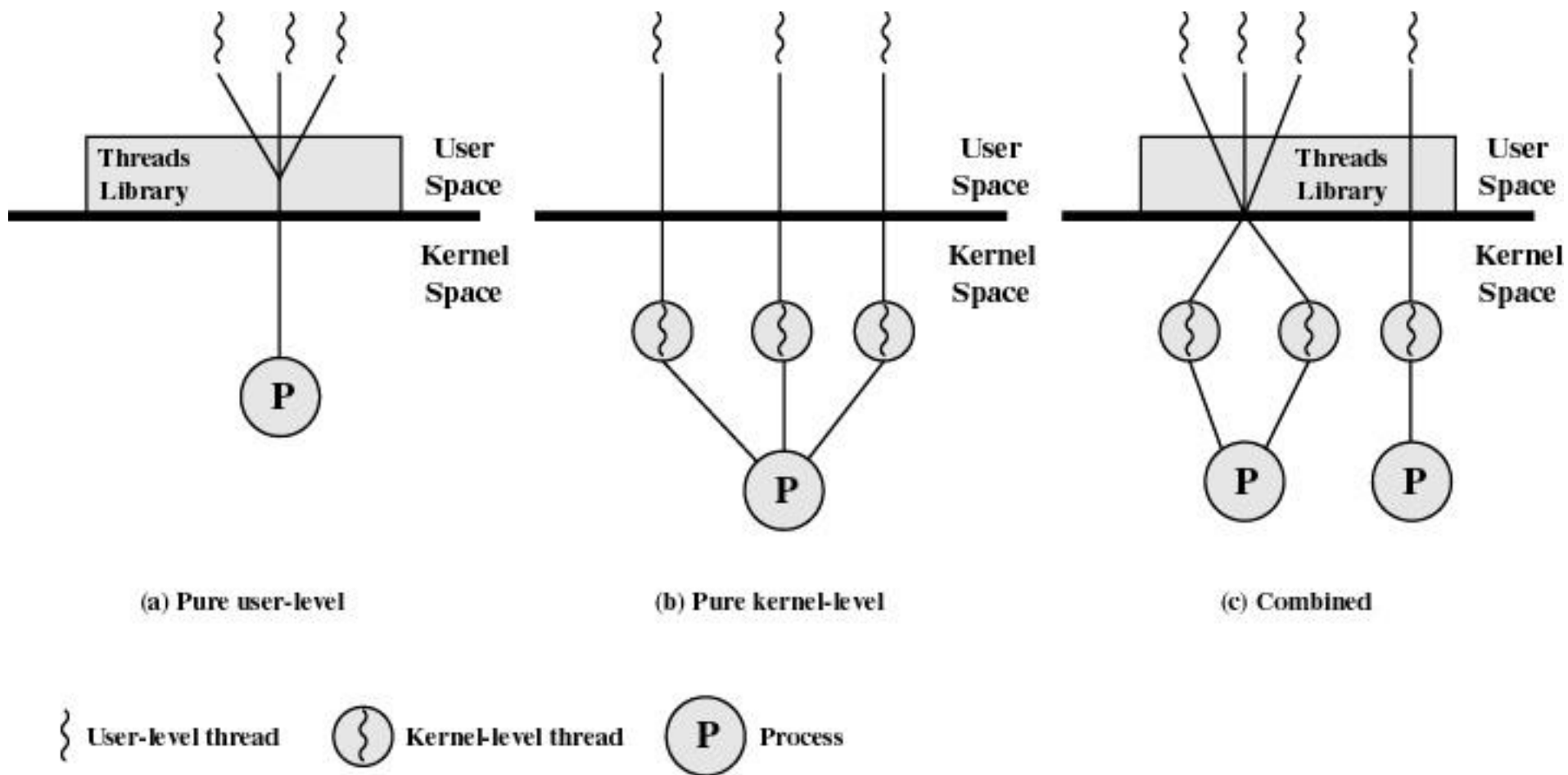


Figure 4.6 User-Level and Kernel-Level Threads

2.6.3 线程的实现方式

3 . 用户级线程与内核控制线程的连接

1) 一对一模型

该模型是为每一个用户线程都设置一个内核控制线程与之连接，当一个线程阻塞时，允许调度另一个线程运行。在多处理机系统中，则有多个线程并行执行。

该模型并行能力较强，但每创建一个用户线程相应地就需要创建一个内核线程，开销较大，因此需要限制整个系统的线程数。Windows 2000、Windows NT、OS/2等系统上都实现了该模型。

2.6.3 线程的实现方式

2) 多对一模型

该模型是将多个用户线程映射到一个内核控制线程，为了管理方便，这些用户线程一般属于一个进程，运行在该进程的用户空间，对这些线程的调度和管理也是在该进程的用户空间中完成。当用户线程需要访问内核时，才将其映射到一个内核控制线程上，但每次只允许一个线程进行映射。

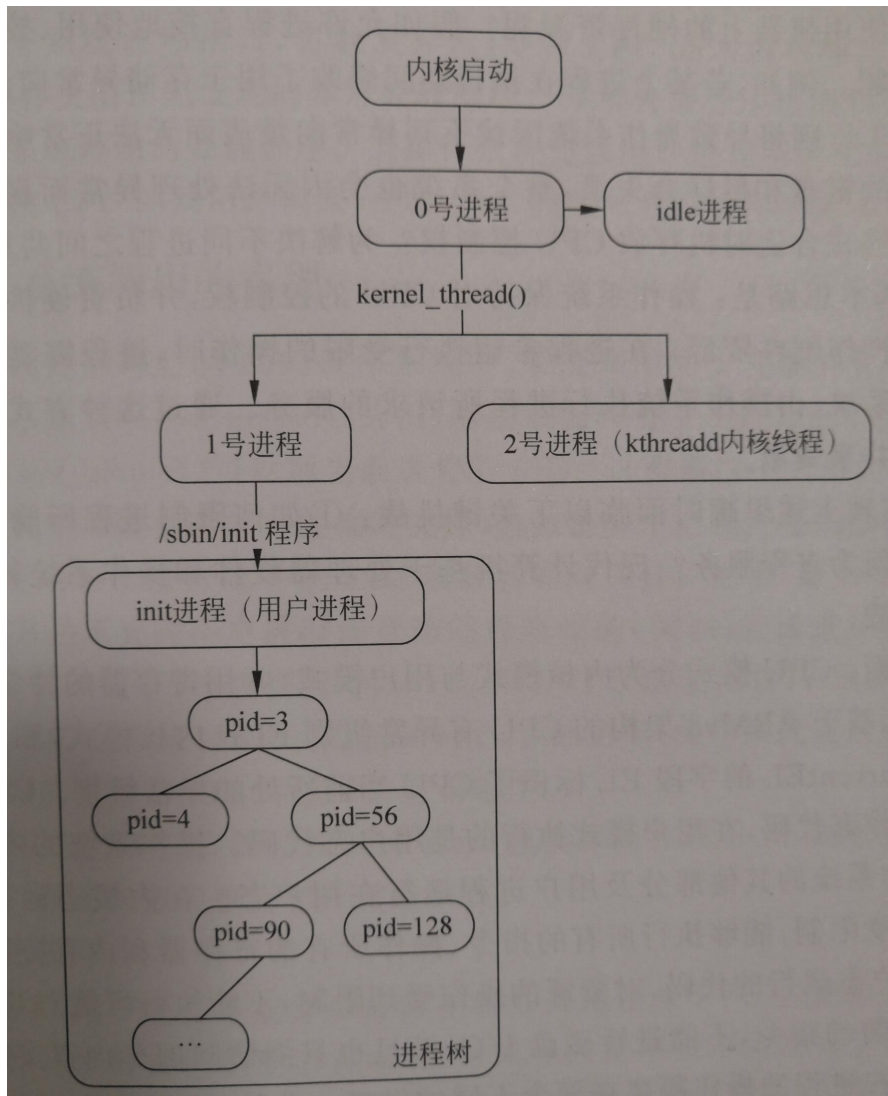
该模型的主要优点是线程管理的开销小，效率高，但当—一个线程在访问内核时发生阻塞，则整个进程都会被阻塞，而且在多处理机系统中，一个进程的多个线程无法实现并行。

2.6.3 线程的实现方式

3) 多对多模型

该模型结合上述两种模型的优点，将多个用户线程映射到多个内核控制线程，内核控制线程的数目可以根据应用进程和系统的不同而变化，可以比用户线程少，也可以与之相同。

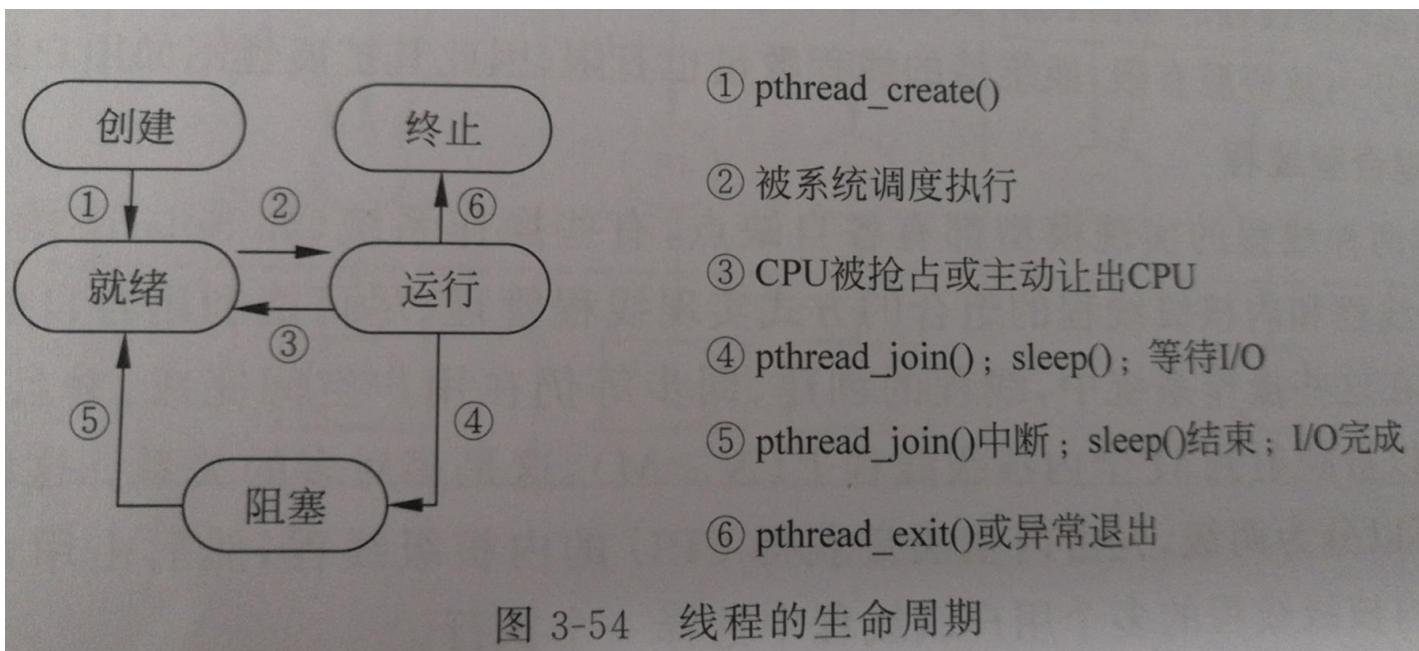
2.7.1 openEuler中的进程树



- 0号进程完成系统初始化：初始化页表、中断处理表、系统时间等，之后调用 `kernel_thread` 创建1号和2号进程，0号进程变成idle进程。
- 1号进程完成剩余的初始化，执行 `/sbin/init` 程序，初始化用户空间，成为init进程，运行在用户态。
- init进程是所有用户进程的共同祖先。
- 2号进程又称为kthreadd内核线程，运行在内核空间，对内核线程进行管理和调度。

2.7.2 openEuler中线程的实现

openEuler使用的是内核级线程，面向用户的线程库是NPTL。

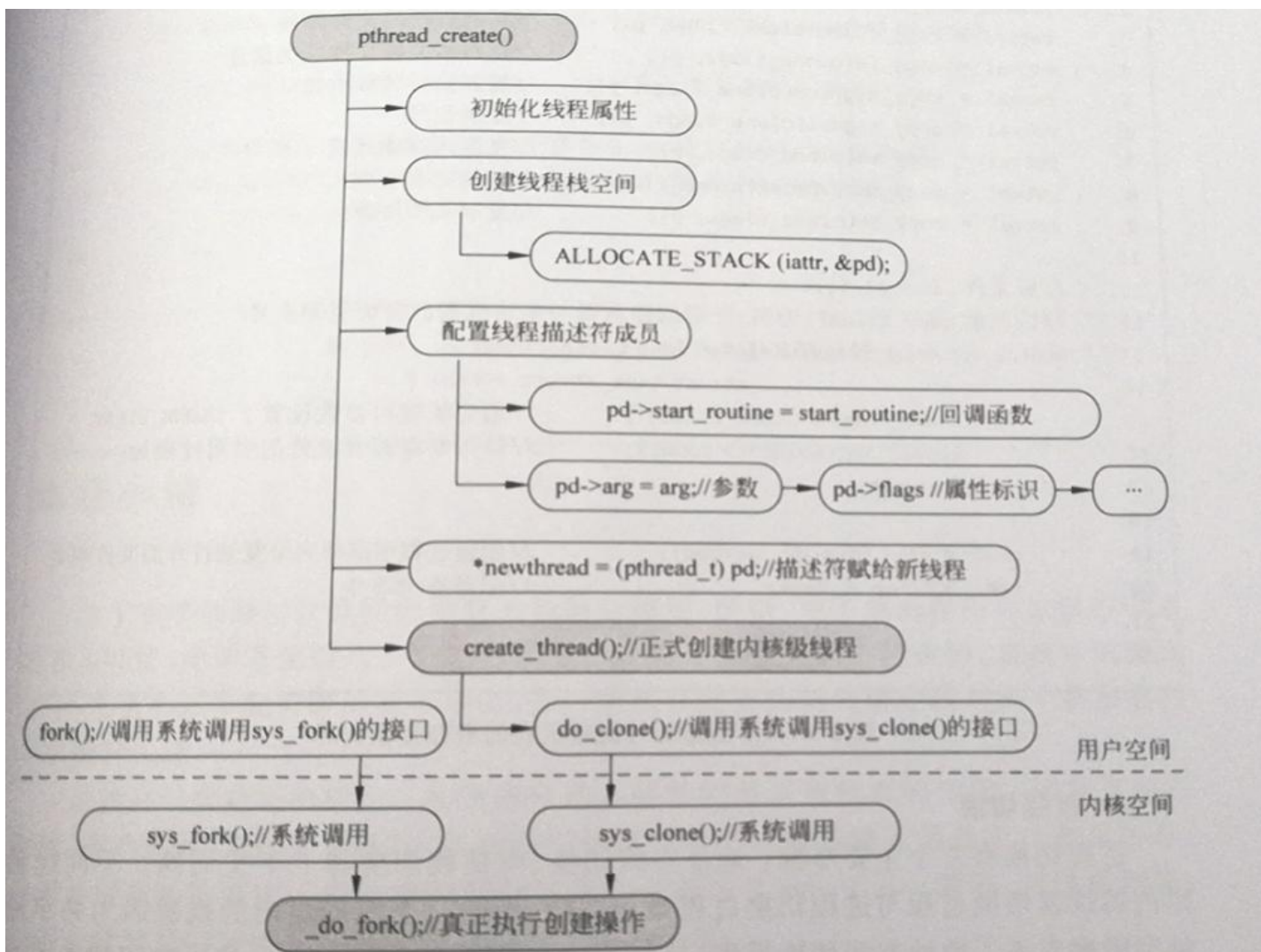


2.7.2 openEuler中线程的实现

表 3-2 线程控制接口与进程原语的对应关系

基 本 控 制	线程库 API 函数	进 程 原 语
创建	pthread_create()	fork()/clone()
终止	pthread_exit()	exit()
等待回收	pthread_join()	wait()/waitpid()
获取 ID	pthread_self()	getpid()

2.7.2 函数pthread_create()的创建流程



2.7.2 Linux中fork和clone区别与联系

Unix标准的复制进程的系统调用是fork（即分叉），但是Linux，BSD等操作系统并不止实现这一个，确切的说linux实现了三个，fork,vfork,clone（确切说vfork创造出来的是轻量级进程，也叫线程，是共享资源的进程）

系统调用	描述
fork	fork创造的子进程是父进程的完整副本，复制了父亲进程的资源，包括内存的内容task_struct内容
vfork	vfork创建的子进程与父进程共享数据段,而且由vfork()创建的子进程将先于父进程运行
clone	Linux上创建线程一般使用的是pthread库 实际上linux也给我们提供了创建线程的系统调用，就是clone

clone函数功能强大，带了众多参数，因此由它创建的进程要比前面2种方法要复杂。clone可以让你有选择性的继承父进程的资源，你可以选择想vfork一样和父进程共享一个虚存空间，从而使创造的是线程，你也可以不和父进程共享，你甚至可以选择创造出来的进程和父进程不再是父子关系，而是兄弟关系。

<https://blog.csdn.net/gatieme/article/details/51417488>

2.7.2 创建进程和线程的资源复制差异

```
1. //源文件: kernel/fork.c
2. //函数 copy_process()
3. retval = copy_files(clone_flags, p); //复制打开的文件列表
4. retval = copy_fs(clone_flags, p); //复制相关联文件系统信息
5. retval = copy_sighand(clone_flags, p); //复制信号处理函数
6. retval = copy_signal(clone_flags, p); //复制信号
7. retval = copy_mm(clone_flags, p); //复制内存描述符
8. retval = copy_namespaces(clone_flags, p); //复制命名空间
9. retval = copy_io(clone_flags, p); //复制 I/O 资源
10. ...
11. //源文件: kernel/fork.c
12. //以函数 copy_files()为例,介绍创建进程与创建线程的资源复制差异
13. static int copy_files(unsigned long clone_flags,
14.                        struct task_struct * tsk) {
15.     if (clone_flags & CLONE_FILES) { //创建线程时参数设置了 CLONE_FILES
16.         atomic_inc(&oldf->count); //只需要将打开文件的引用计数加一
17.         goto out;
18.     }
19.     newf = dup_fd(oldf, &error); //创建进程则需要完全复制打开的文件列表
20.     tsk->files = newf; //记录在 PCB 中
21.     ...
22. }
```

小结

- 概念：进程、线程、进程控制、原语、PCB、同步、互斥、临界区、信号量
- 程序的顺序执行与并发执行的特点。
- 进程状态及转换原因
- 进程PCB包含内容及组成形式
- 进程控制原语的主要功能
- 并发进程之间的基本关系
- 临界区调度三准则
- 信号量解决进程同步与互斥
- 高级通信的几种方式
- 线程的优点、缺点、实现方式及与进程的比较
- openEuler中的进程树以及线程的实现

作业

- 判断对错

1. 进程在申请处理机得不到满足时,处于等待状态。
2. 任何时刻,处于执行状态的进程至少有一个。
3. 用户进程必须通过进程调度才能获得处理机。
4. 进程的状态有就绪、执行、等待等,每个进程在它的生命期中都可能处于这几种状态之一若干次。
5. 不同的进程执行的代码也不同。
6. 用户进程可通过读取PCB的信息,了解本身当前的情况。
7. 进程不能由阻塞状态直接转变为就绪状态。
8. 单用户系统中,任何时刻,只能有一个用户进程。
9. 处于临界区中的进程是不可中断的。
10. 在基于线程的系统中,系统中所有资源的分配都是以线程为基本单位的。