西安电子科技大学 XIDIAN UNIVERSITY | 计算机学院 School of Computer Science and Technology

操作系统原理

第三章 进程管理

讲者：黄伯虎

第二章 小结

基本内容	一、用户接口 1.作业控制级接口 2.程序级接口 二、作业管理 1.作业的定义 2.作业的分类 3.作业的处理过程：输入、后备、执行、完成
重难点	• 系统功能调用及其过程 • SPOOLing系统及其工作原理 • 作业调度算法及其性能分析

引言

主要内容

- 本章主要阐述进程的基本概念及围绕进程这一概念的一些重要问题，如调度问题、相互作用问题，通信问题等。
- 本章是操作系统课程最为重要的部分，重难点多。

一、进程概念的提出

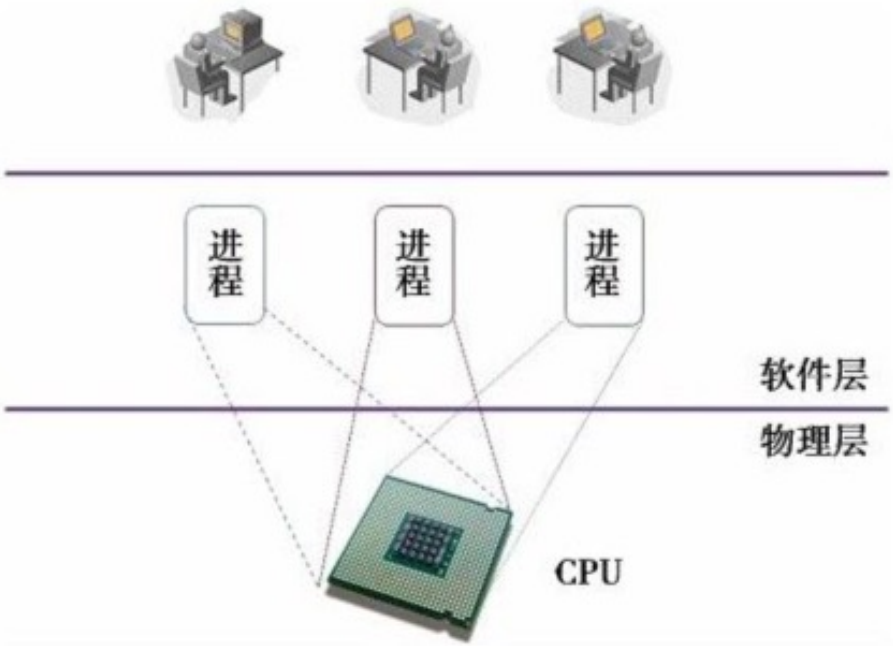
- 早期：
  - 单道环境——程序顺序执行——资源独占，程序执行不间断，结果可再现——程序的执行过程简单，不需要额外的复杂描述。
- 现在：
  - 多道环境——程序并发执行——资源共享，程序执行可间断，相互制约性，结果的不可再现——程序的执行成为一个复杂的动态性很强的过程——需要引入新的概念来清晰的描述这种过程——于是便出现了“进程”这一概念。

“进程”的概念是伴随着多道程序设计技术，即并发的出现而出现的！

一、进程概念的提出

概念的提出

- 最早出现在IBM的CTSS/360系统中，那时称为“工作(Job)”。
- “进程(Process)”则最早出现在MIT的MULTICS系统中。



并发进程将一个物理的CPU虚拟为多个逻辑CPU！

二、进程的基本概念

进程(Process)的定义

直观的理解：

- 进程就是进展中的程序 / 执行中的程序

  进程 = 程序 + 执行

教材中定义：

- 进程是程序的一次执行，该进程可与其它进程并发执行；它是一个动态的实体，是资源的基本分配单元。(在早期的操作系统中，进程也是基本的执行单元)。

二、进程的基本概念

进程与程序的关系

❖ 区别：

① 程序是静态的，是有序代码的集合；

进程是动态的，是程序的一次执行。

② 程序的永久的，没有生命周期，可长久保存；

进程是暂时的，有生命周期，是一个动态不断变化的过程。

③ 进程是操作系统资源分配和保护的基本单位；

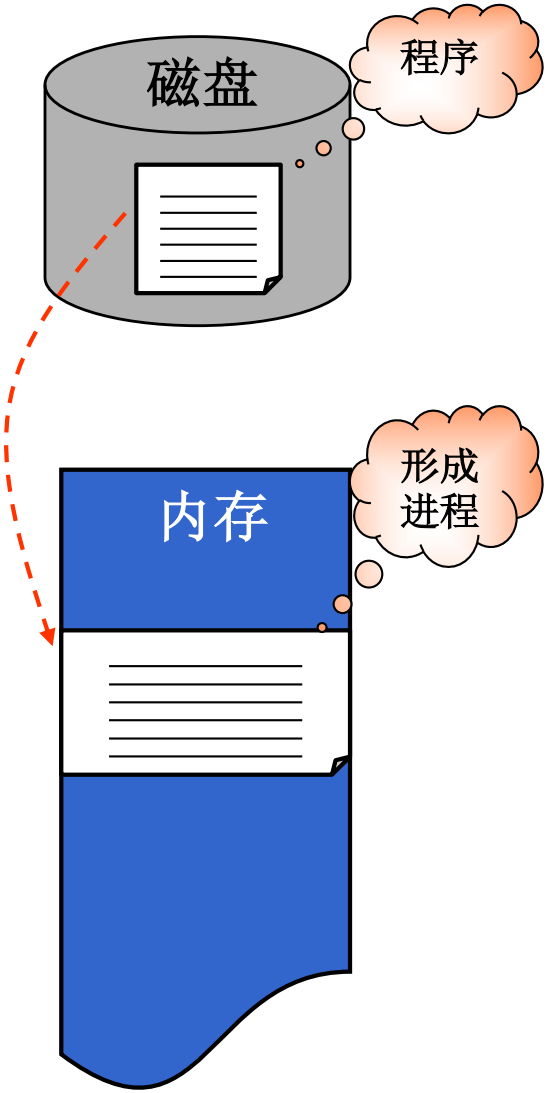
程序没有此功能。

④ 进程与程序的结构不同。

❖ 联系：

① 通过多次执行，一个程序可对应多个进程；

② 通过调用关系，一个进程可包括多个程序。



二、进程的基本概念

进程与程序的关系——一个生活中的比喻

做蛋糕:

如果要你做个蛋糕，你会怎么做？

(1)需要一个食谱;

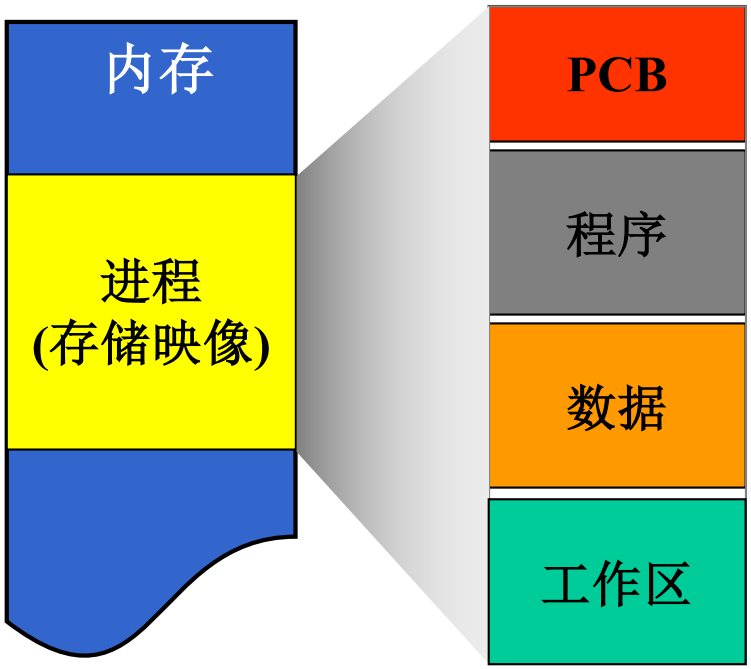
(2)需要原料：面粉，鸡蛋，糖，香精，奶油…;

(3)按照食谱加工。

程序：	食谱（即用适当形式描述的方法）
硬盘：	食谱书（存储载体）
CPU：	你（执行单位）
内存：	你的大脑（记忆）
数据：	各种原料（输入数据）
进程：	所有动作的总和（阅读食谱，取来原料，烘制蛋糕）

二、进程的基本概念

进程的组成(内存映像)



“Process Control Block”，包含着进程的描述和控制信息，进程存在的唯一标志。 } 灵魂

“纯代码”部分，也称为“文本段”，描述了进程要完成的功能，是进程执行时不可修改的部分。

进程执行时用到的数据(全局变量，静态变量，常量)。

} 实体

一片可供进程使用的动态区域(堆栈区)，可用于：内存的动态分配，保存局部变量，传递参数、函数调用地址等。

二、进程的基本概念

进程控制块(PCB)

定义：

- 是操作系统用来记录进程详细状态和相关信息的基本数据结构，它和进程是一一对应的，是进程存在的唯一标识。

作用：

- 提供进程的各种信息，以便操作系统查询、控制和管理。
- 进程的档案，描述进程的特征，记载进程的历史，决定进程的命运。

二、进程的基本概念

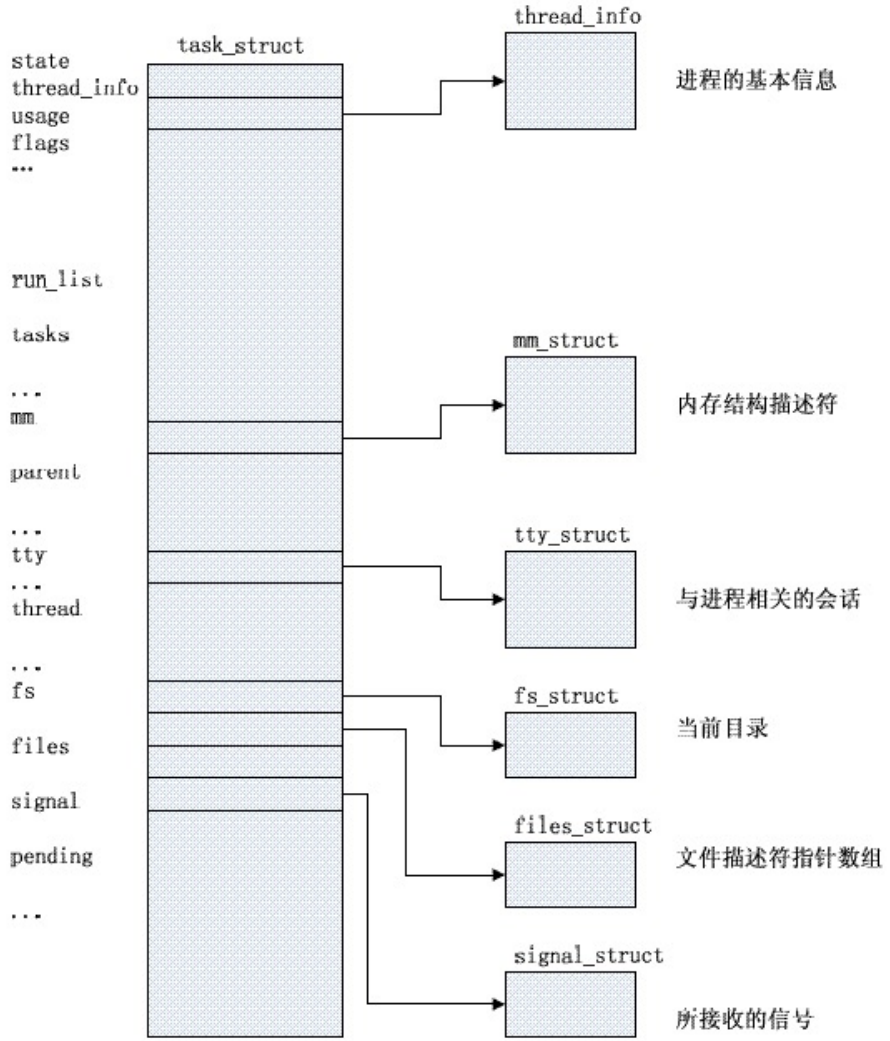
PCB结构

类型	内容	作用
标识信息	• 进程标识符（ID）； • 父进程标识符（ID）； • 用户标识（User ID）。	标识一个进程
状态（现场）信息	• 控制和状态寄存器：程序计数器（PC），CPU状态寄存器； • 用户可见寄存器； • 栈指针。	记录处理机上下文信息，以备中断恢复之用
控制信息	• 调度和状态信息：进程状态，优先级，调度相关信息； • 数据结构：进程的组织方式（队列，环...） • 进程间通信：进程通信的信息，如消息队列，互斥、同步； • 进程特权； • 存储管理；虚存段表/页表指针； • 资源使用权和使用情况：进程控制资源，如打开文件等。	用于进程的调度管理

二、进程的基本概念

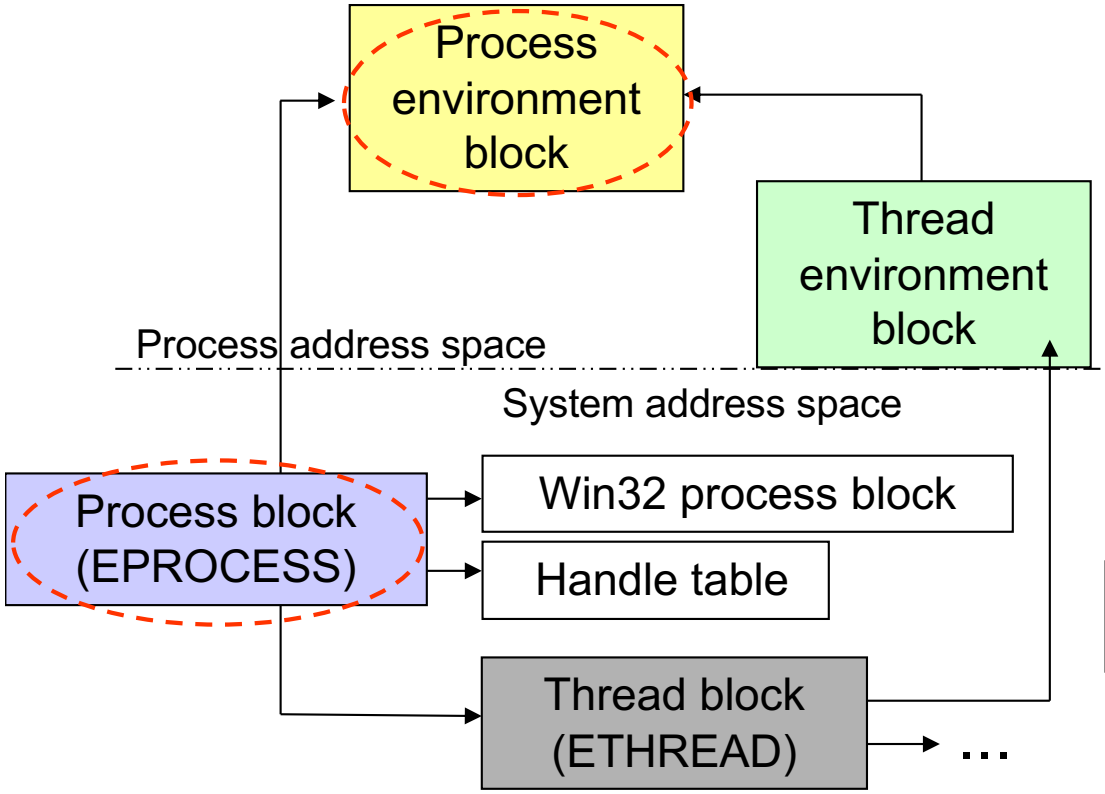
实例：Linux PCB——task_struct

```
include/linux/sched.h
384 struct task_struct {
385 volatile long state;
386 struct thread_info *thread_info;
387 atomic_t usage;
388 unsigned long flags;
389 unsigned long ptrace;
390
391 int lock_depth;
392
393 int prio, static_prio;
394 struct list_head run_list;
395 prio_array_t *array;
396
397 unsigned long sleep_avg;
398 long interactive_credit;
399 unsigned long long timestamp;
400 int activated;
401
302 unsigned long policy;
403 cpumask_t cpus_allowed;
404 unsigned int time_slice, first_time_slice;
405
406 struct list_head tasks;
407 struct list_head ptrace_children;
408 struct list_head ptrace_list;
409
410 struct mm_struct *mm, *active_mm;
...
413 struct linux_binfmt *binfmt;
414 int exit_code, exit_signal;
415 int pdeath_signal;
...
```

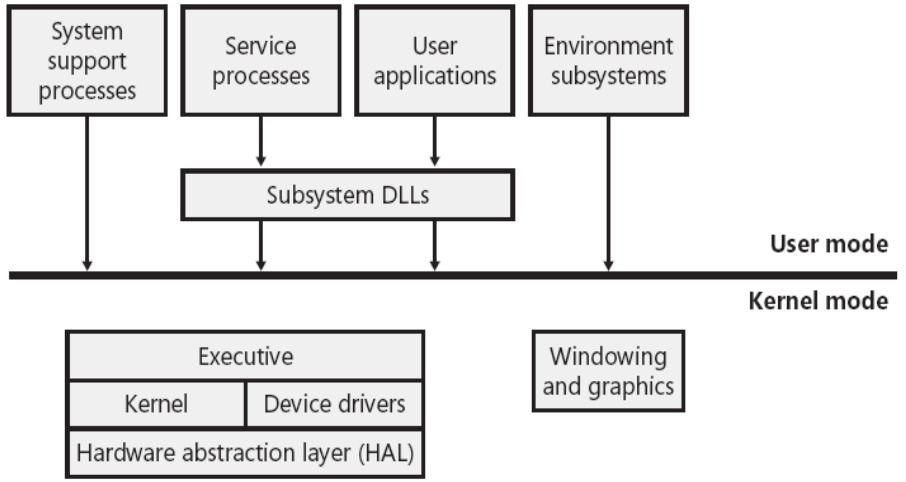


二、进程的基本概念

例：Windows PCB




```
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x078 ProcessLock      : _EX_PUSH_LOCK
   +0x080 CreateTime       : _LARGE_INTEGER
0x1c9a12b`a6109f10
   +0x088 ExitTime         : _LARGE_INTEGER 0x0
   +0x090 RundownProtect   : _EX_RUNDOWN_REF
   +0x094 UniqueProcessId  : 0x000000c0
   +0x098 ActiveProcessLinks : _LIST_ENTRY
[ 0x808a6f40 - 0x811be338 ]
   +0x0a0 QuotaUsage       : [3] 0x320
   +0x0ac QuotaPeak        : [3] 0x320
   +0x0b8 CommitCharge     : 0x31
   +0x0bc PeakVirtualSize  : 0x6cd000
   +0x0c0 VirtualSize      : 0x6cd000
   +0x0c4 SessionProcessLinks : _LIST_ENTRY
[ 0xfa12c010 - 0x811be364 ]
   +0x0cc DebugPort        : (null)
   +0x0d0 ExceptionPort    : 0xe1292bc0
   +0x0d4 ObjectTable      : 0xe1138f40
_HANDLE_TABLE
……
```



简化的windows结构图

三、进程的产生与消失

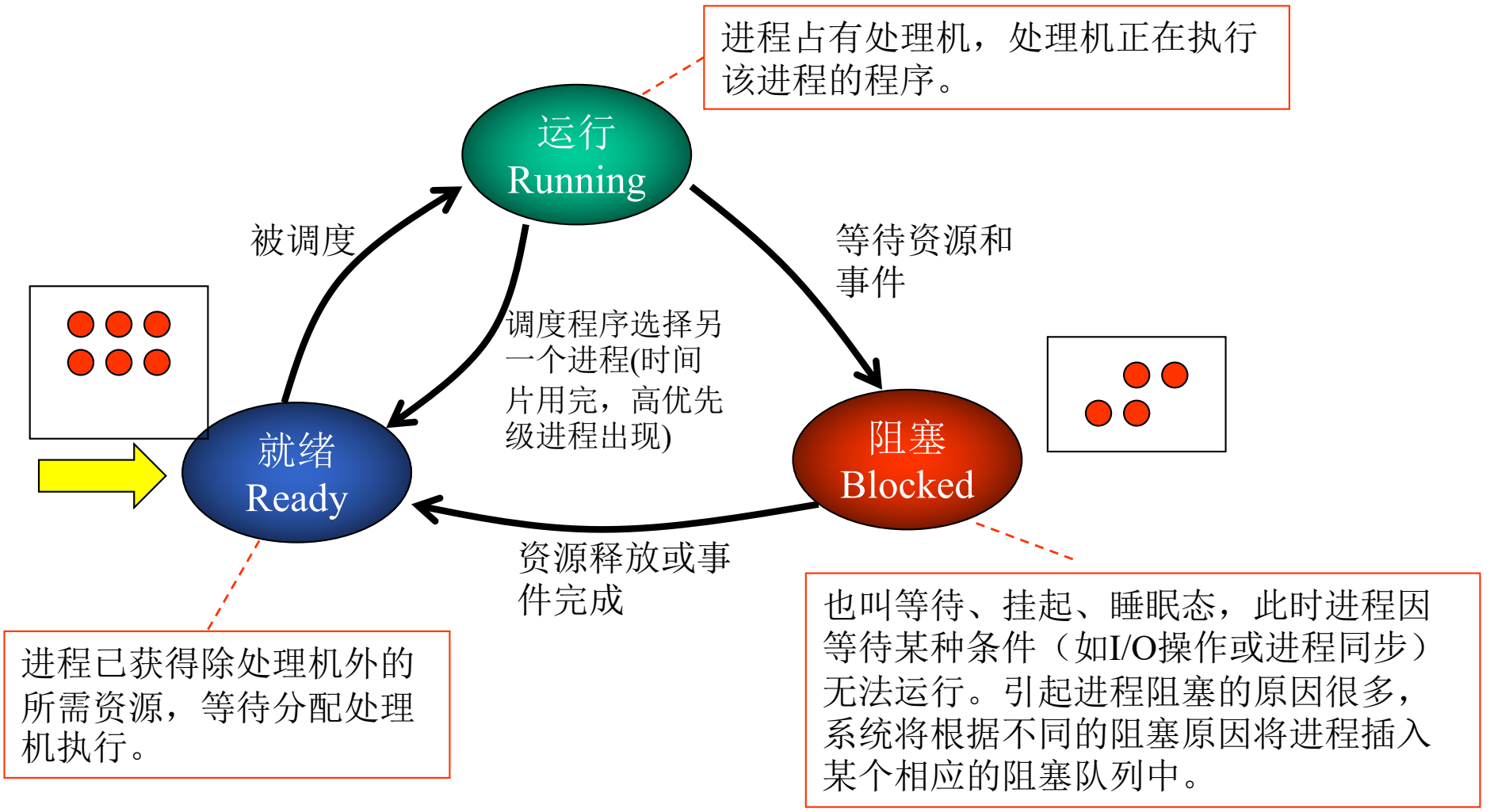
进程的产生

- 系统初始化
- 用户执行程序（命令，双击）
- 程序启动程序（子进程）
- 批处理系统：作业初始化

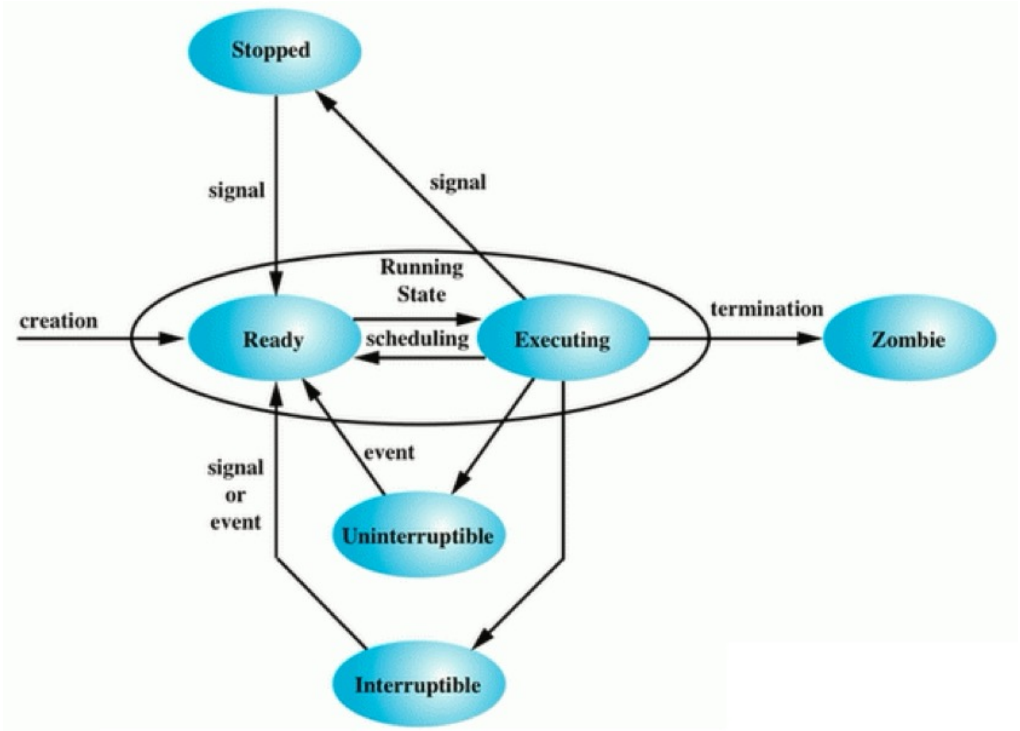
进程的消失

- 寿终：运行结束而退出
- 自杀：因错误而自行终止
- 他杀：被其他进程/用户强行终止
- 处决：因异常而被系统强行终结

进程的基本状态及其转换

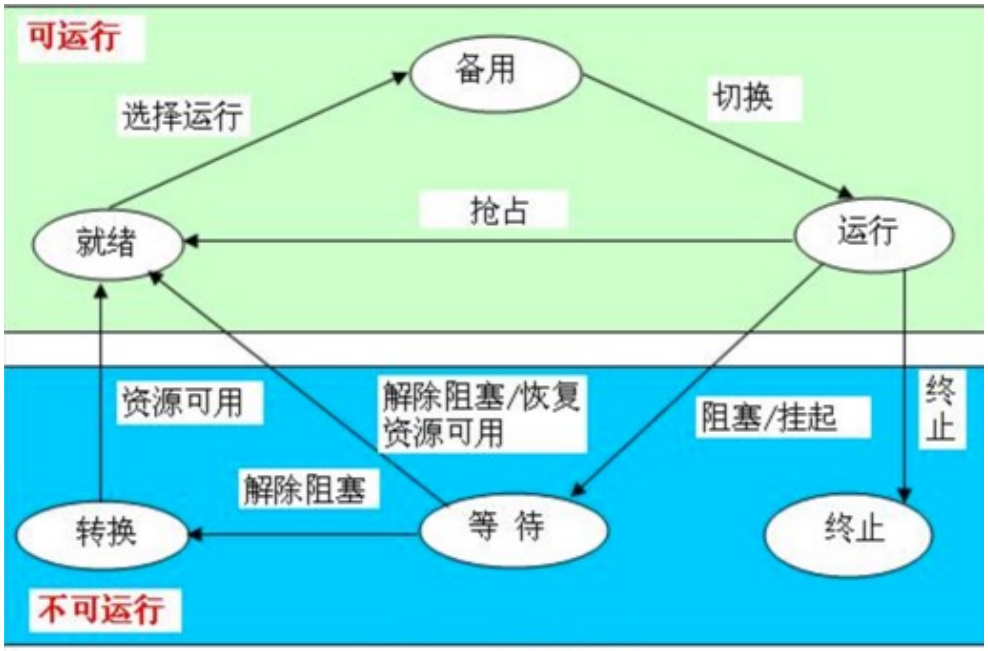


Linux进程状态及转换



1. Ready：准备运行
2. Executing：运行过程
3. Stopped：暂停状态(收到SIGSTOP等信号就会进入暂停，调试状态，可通过SIGCONT信号转换到Ready状态)
4. Uninterruptible：不可中断睡眠(深度睡眠，不响应信号，如信号量阻塞)
5. Interruptible：可中断睡眠——浅度睡眠，可以响应信号(如Sleep状态)
6. Zombie：僵尸状态(进程已退出或者结束，但是父进程不知道，未彻底回收)

Windows进程状态及转换



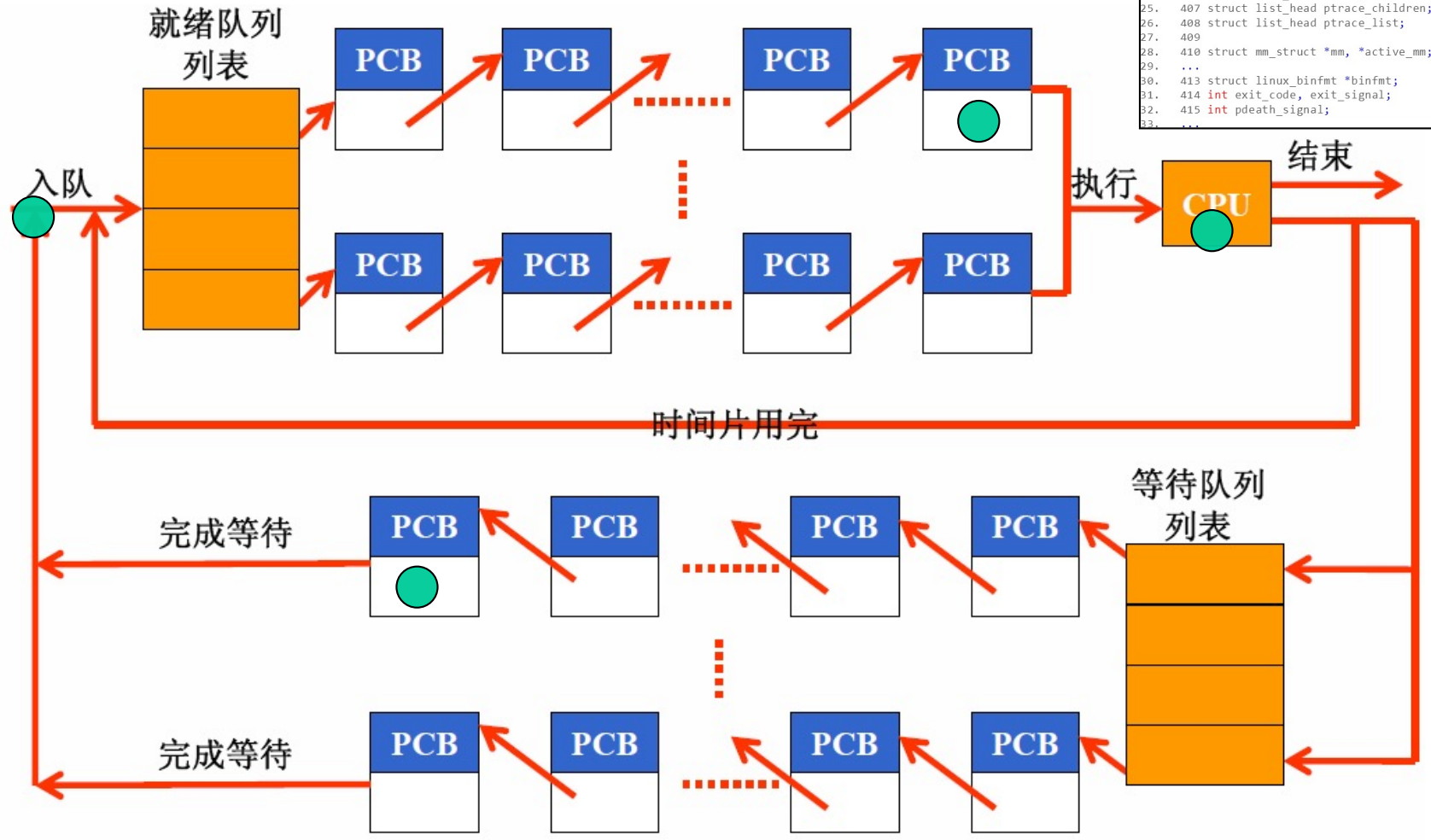
- 就绪态：可以被调度执行。
- 备用态：备用线程已经被选择下一次在一个特定的处理器上运行。该线程在这个状态等待，直到那个处理器可用，如果备用线程的优先级足够高，正在那个处理器上运行的线程可能被这个备用线程抢占。否则，该备用线程要等到正在运行的线程被阻塞或结束其时间片。
- 运行态：备用线程将进入运行状态并开始执行。

- 等待态：进程进入等待状态。当等待的条件满足时，如果它的所有资源都可用，则线程转到就绪态。
- 转换态：一个线程在等待后，如果准备好运行但资源不可用时，进入该状态。
- 终止态：一个线程可以被自己或者被另一个线程终止，或者当它的父进程终止时终止。

四、进程的执行与控制

进程的组织管理——链式队列

```
include/linux/sched.h
384 struct task_struct {
385 volatile long state;
386 struct thread_info *thread_info;
387 atomic_t usage;
388 unsigned long flags;
389 unsigned long ptrace;
390
391 int lock_depth;
392
393 int prio, static_prio;
394 struct list_head run_list;
395 prio_array_t *array;
396
397 unsigned long sleep_avg;
398 long interactive_credit;
399 unsigned long long timestamp;
400 int activated;
401
302 unsigned long policy;
403 cpumask_t cpus_allowed;
404 unsigned int time_slice, first_time_slice;
405
406 struct list_head tasks;
407 struct list_head ptrace_children;
408 struct list_head ptrace_list;
409
410 struct mm_struct *mm, *active_mm;
...
413 struct linux_binfmt *binfmt;
414 int exit_code, exit_signal;
415 int pdeath_signal;
...
```



四、进程的执行与控制

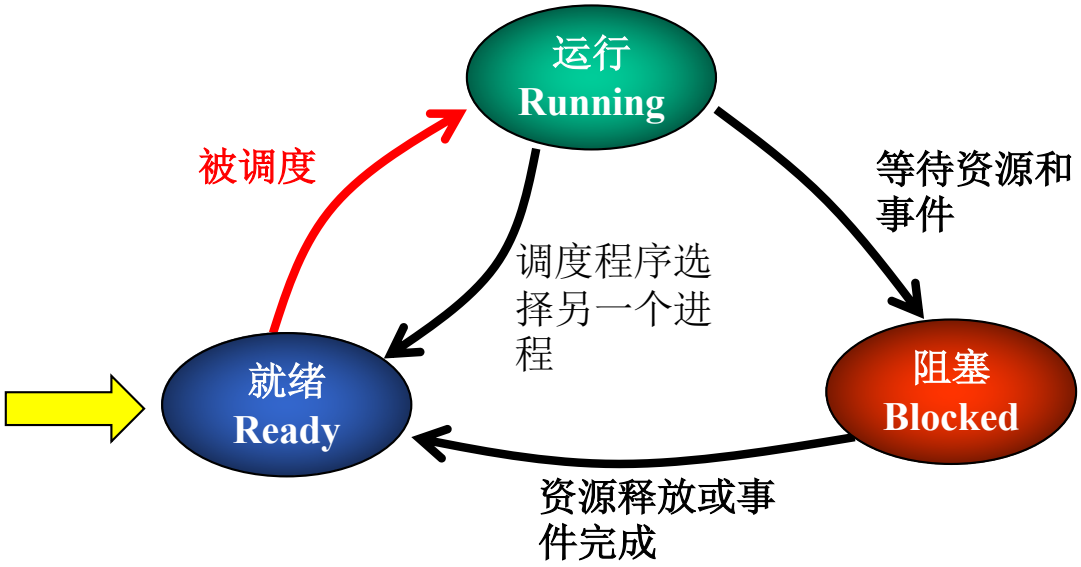
进程控制

- 系统对进程的控制和管理是通过操作系统内核中的原语实现的。
- 原语
  - 由若干条指令构成的可完成特定功能的程序段，必须在管态下运行，它是一个“原子操作(atomic operation)”过程，执行过程不能被中断。
  - 原语的原子性主要是通过屏蔽中断或原语固化保证的。
- 原语分类
  - 进程创建、撤销、阻塞、唤醒、挂起、激活原语。

五、进程调度

概念

- 就是按照一定的算法，从就绪队列中选择某个进程占用CPU的方法。



调度时机：

1. 新进程创建时；
2. 进程退出时；
3. 进程阻塞时；
4. 时间片用完时；
5. 进程被唤醒时。

设计原则

设计者

系统

- 公平性
- 吞吐量

用户 / 进程

- 及时性
- 较短的周转时间

不同环境下的调度算法目标

所有系统
- 公平——给每个进程公平的CPU份额
- 策略强制执行——看到所宣布的策略执行
- 平衡——保持系统的所有部分都忙碌

批处理系统
- 吞吐量——每小时最大作业数
- 周转时间——从提交到终止间的最小时间
- CPU利用率——保持CPU始终忙碌

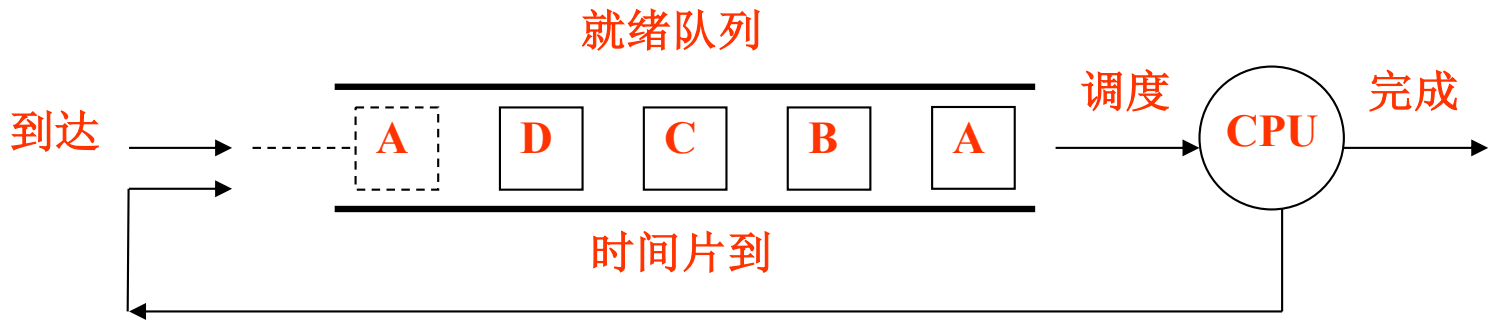
交互式系统
- 响应时间——快速响应请求
- 均衡性——满足用户的期望

实时系统
- 满足截止时间——避免丢失数据
- 可预测性——在多媒体系统中避免品质降低

交互式系统(分时系统)下的调度策略

时间片轮转法(RR, Round Robin)

- 思想：系统规定一个时间长度(时间片)作为允许一个进程运行的时间，如果在这段时间该进程没有执行完，则必须让出CPU给下一个进程使用，自己则排到就绪队列末尾，等待下一次调度；如果时间片结束之前被阻塞或结束，则CPU立即切换。



RR算法特点

- 响应速度快，交互性强
- 处理器虚拟化(*n*个进程似乎感觉有*n*个处理器)——1个快处理器变为*n*个慢处理器。

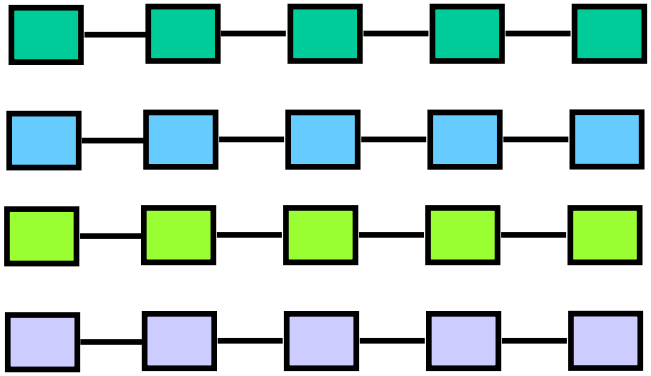
RR算法设计难点

- 时间片大小
  - ① 过长：退化为FCFS算法，进程在一个时间片内都执行完，响应时间长。
  - ② 过短：用户的一次请求需要多个时间片才能处理完，上下文切换次数增加，开销时间显著增大，响应时间也会变长。

问题：采用RR算法后程序的执行时间会缩短吗？

优先级调度算法

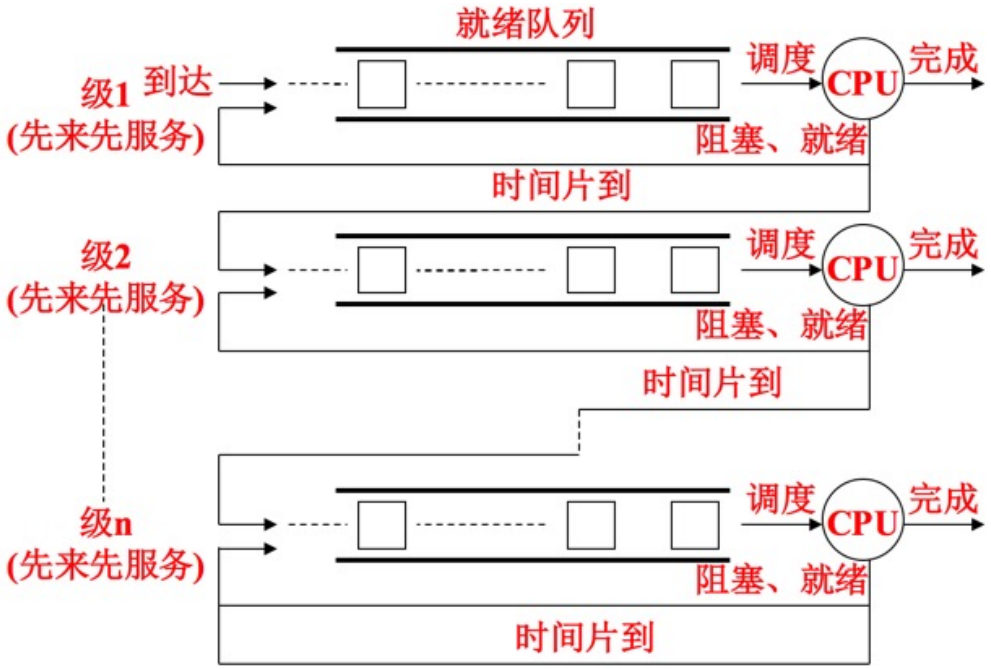
- 基本思想：为系统中的每个进程规定一个优先数，就绪队列中具有最高优先数的进程有优先获得处理机的权利；如果几个进程的优先数相同，可则对它们实行RR调度策略。



设置优先数的方法

1. 静态优先数法：
   - 在进程创建时指定优先数，在进程运行时优先数不变；
   - 存在的问题：可能会产生“饥饿”现象。
2. 动态优先数法：
   - 在进程创建时创立一个优先数，但在其生命周期内优先数可以动态变化。

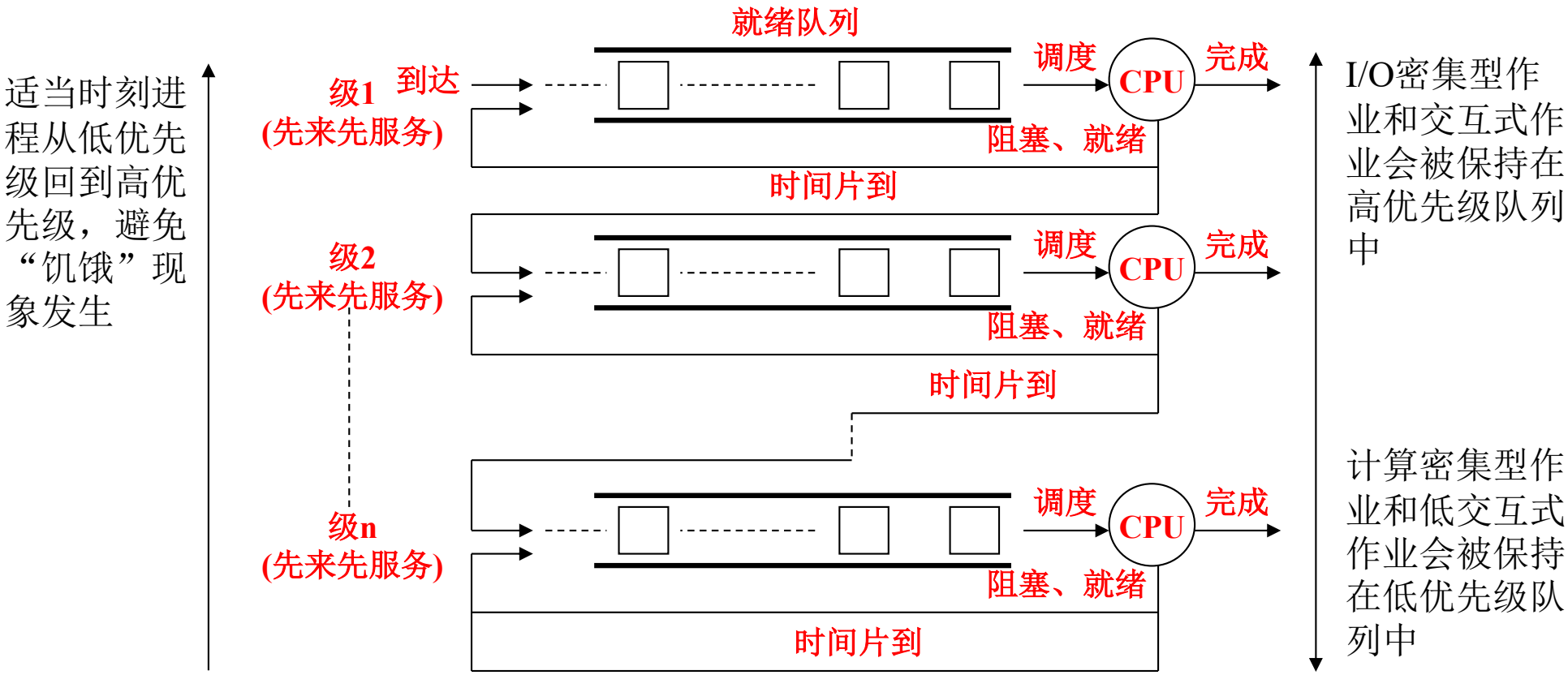
多级反馈队列调度算法(Multilevel Feedback Queue Scheduling)



基本思想：

- 系统中维持多个不同优先级的就绪队列。
- 每个就绪队列具有不同长度的时间片。优先级高的就绪队列里的进程，获得的时间片短；优先级低的就绪队列里的进程，获得的时间片长。
- 新进程进入时加入优先级最高的就绪队列的末尾。

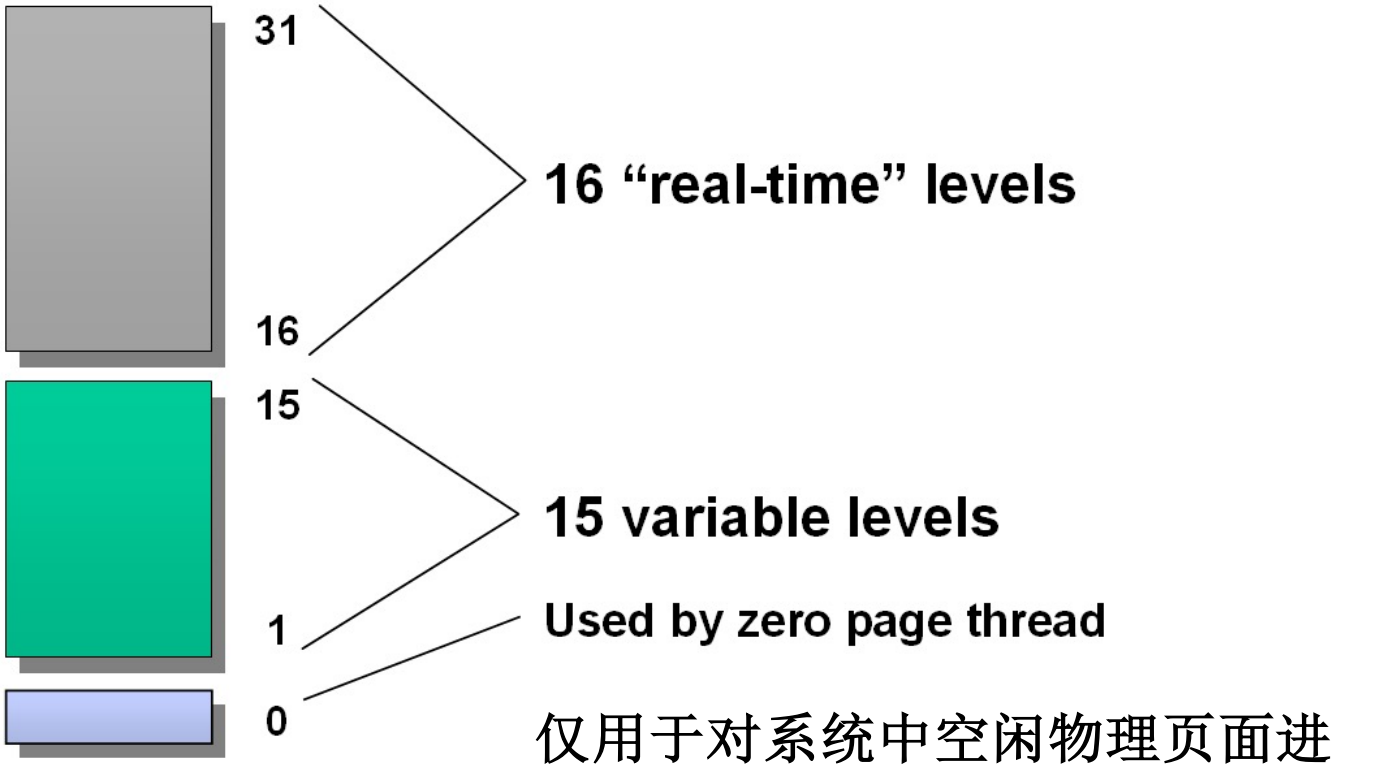
五、进程调度



多级反馈队列调度算法是目前最为通用的CPU调度策略，经过适当的配置和改进即可适应不同的系统！

实例：Windows进程(线程)调度

- 调度算法(NT系列)
  - 采用“抢占式动态优先级多队列调度算法”
- 优先级

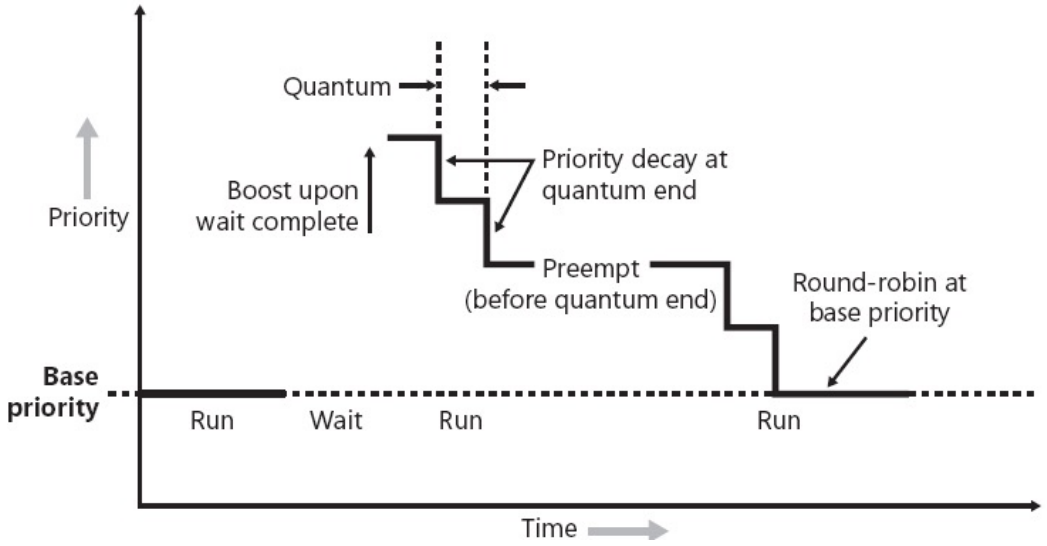


实例：Windows进程(线程)调度

Windows线程的基本优先级

		Win32 Process Classes					
		Realtime	High	Above Normal	Normal	Below Normal	Idle
Win32 Thread Priorities	Time-critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above-normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below-normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

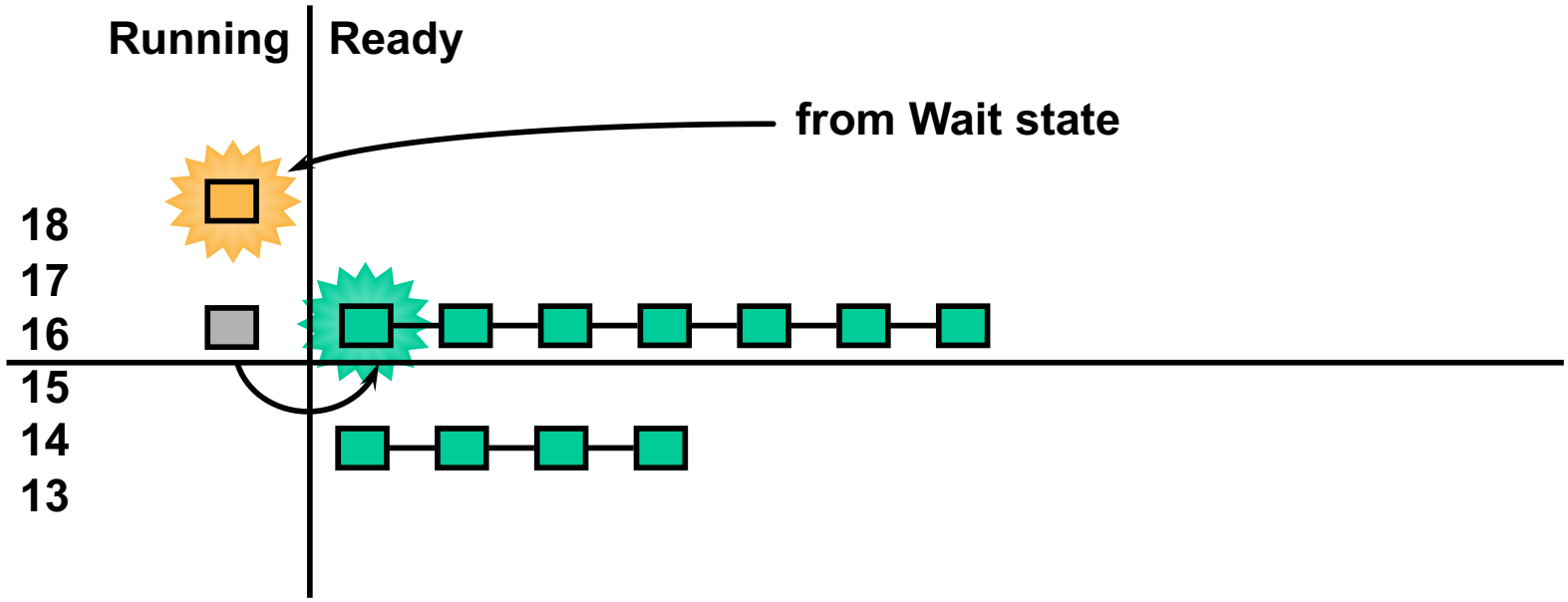
动态优先级：
每个线程的“动态优先级”以线程的基本优先级为初始值，随着进程所做工作类型的不同而上下浮动



实例：Windows进程(线程)调度

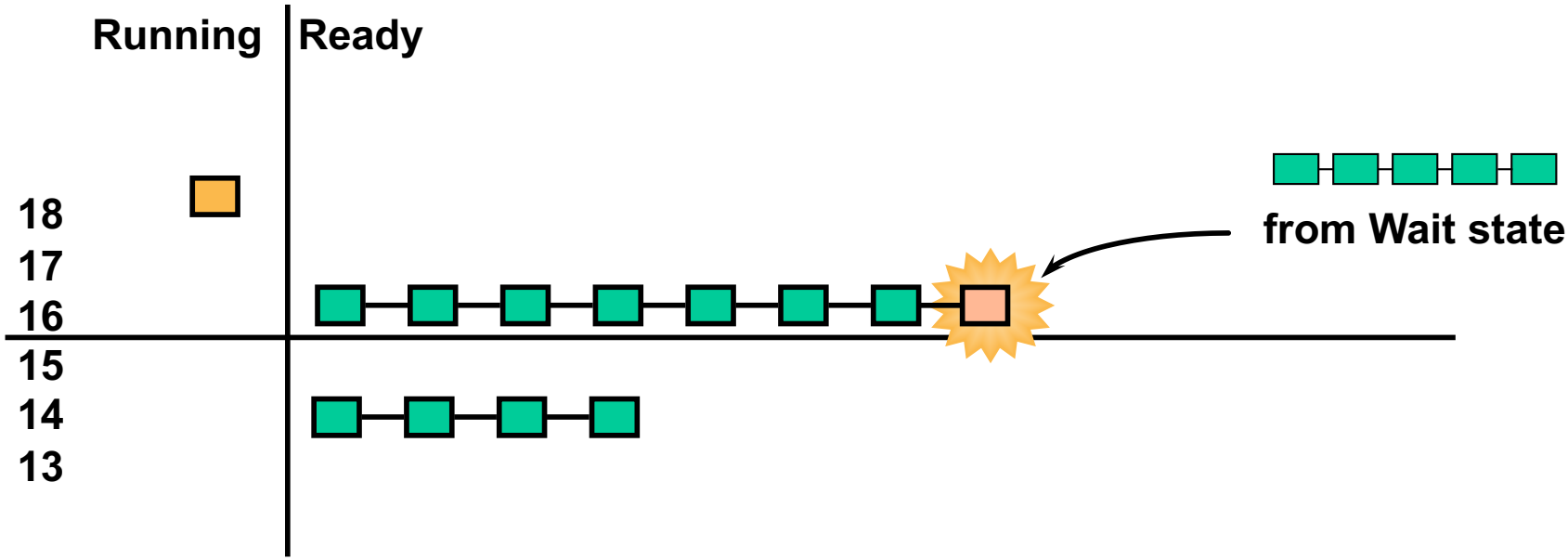
调度情形

Preemption(抢占)



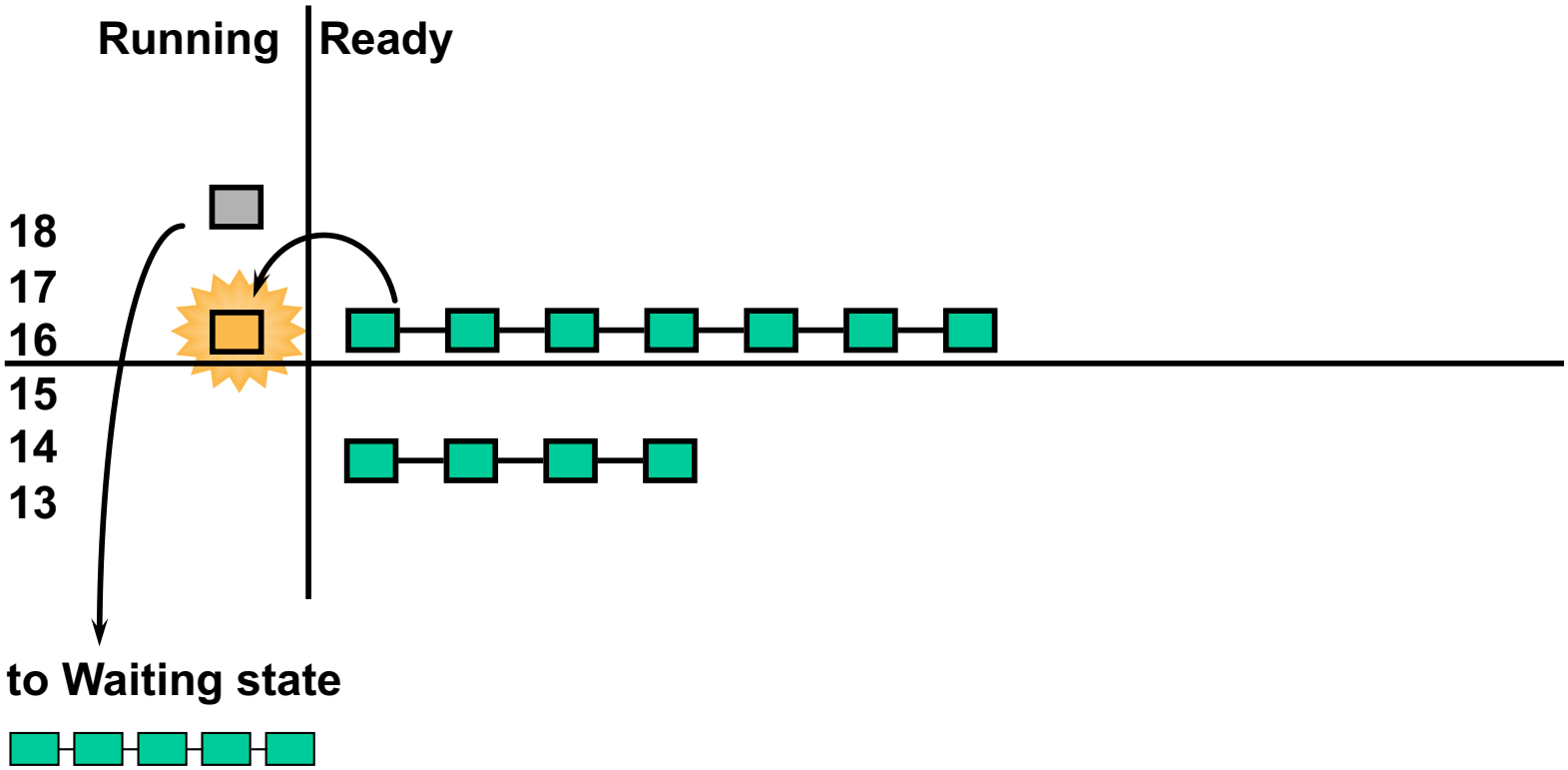
实例：Windows进程(线程)调度

- Ready after Wait Resolution(等待结束进入就绪状态)



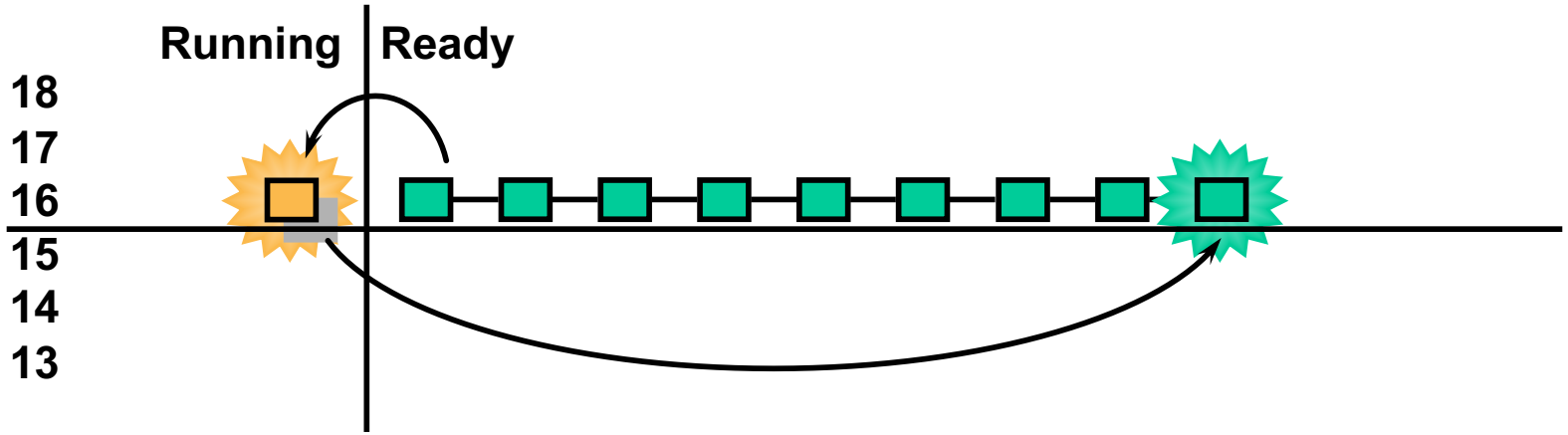
实例：Windows进程(线程)调度

- Voluntary Switch(自愿切换)

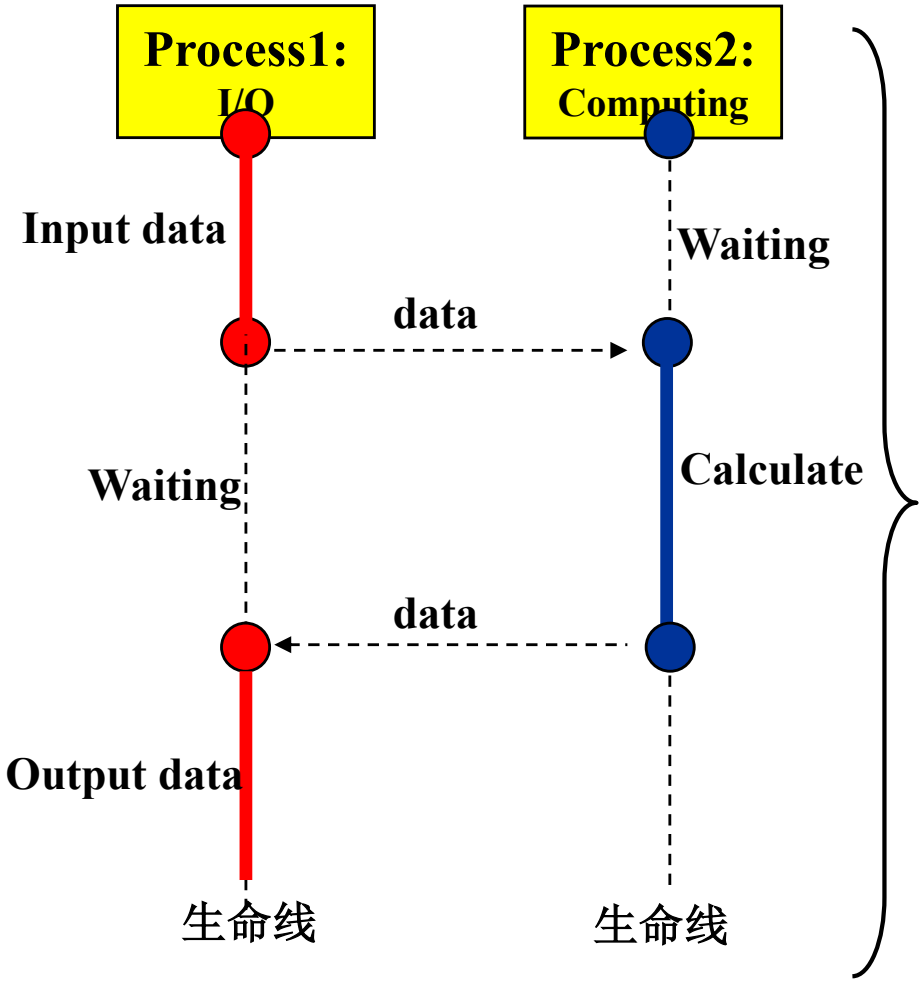


实例：Windows进程(线程)调度

- Quantum End (时间片用完)



六、进程间的相互作用

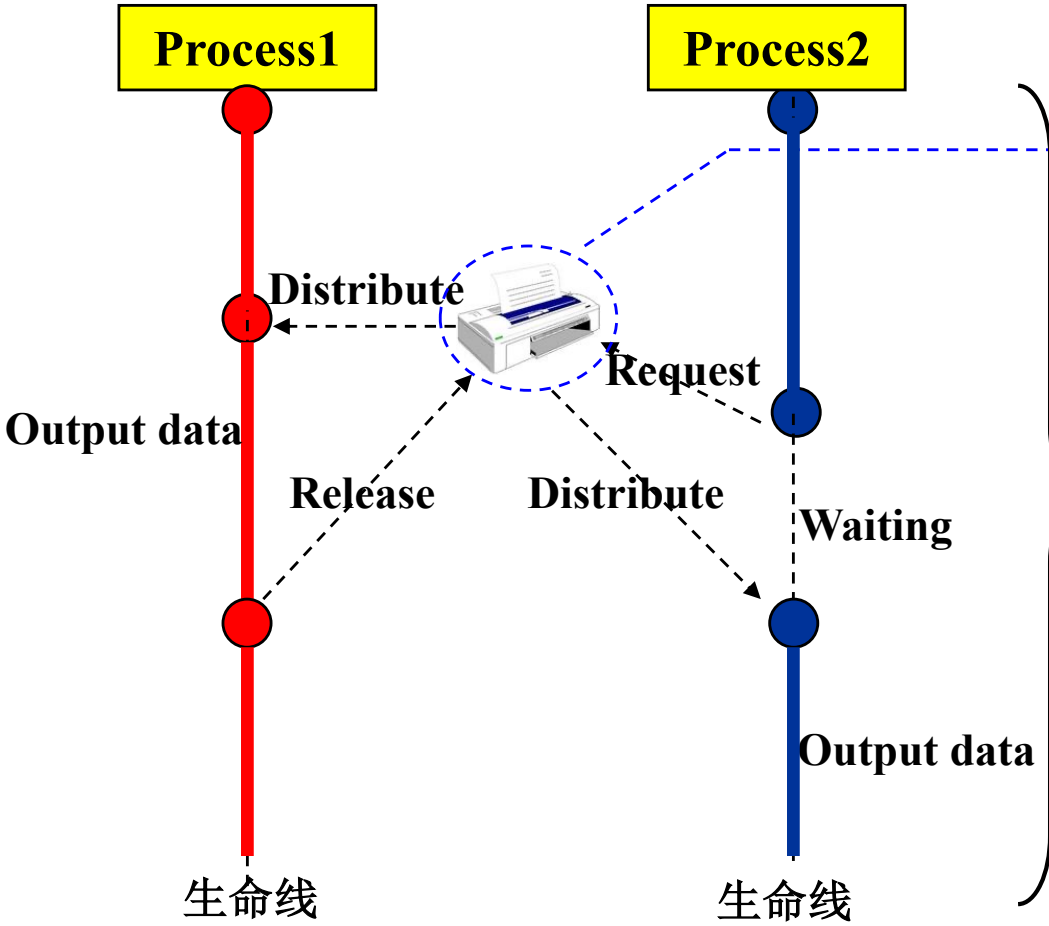


进程之间这种相互合作、协同工作的关系称为进程的同步。

简单说来就是：多个相关进程在执行次序上的协调。

制约关系：直接制约。

六、进程间的相互作用



临界资源：也称独占资源，是指在一段时间内只允许一个进程访问的资源。例如打印机，磁带机，也可以是进程共享的数据、变量等。

多个进程因为争夺临界资源而相互排斥执行的过程称为进程的互斥。

制约关系：间接制约。

六、进程间的相互作用

并发进程的问题

❖ 例1：

余票数n=10

```
…
x=n;
x=x-1;
n=x;
…
```

```
…
x=n;
x=x-1;
n=x;
…
```

设各窗口分别买了1张票，请问最后的余票数n=?

六、进程间的相互作用

并发进程的问题

例2：

```
Process A
i = 0;
while( i < 10)
    i++;
printf("A finished");
```

```
Process B
i = 0;
while( i > -10)
    i--;
printf("B finished");
```

请问那个进程会先执行完?

问题：如何约束进程的行为，使其不管如何动态执行，其结果都能保持正确？

六、进程间的相互作用

加锁法——自旋锁(spinlock)

- 做法：设置一个共享变量W (锁)，初值为0。当一个进程想进入其临界区(使用临界资源的程序段)时，它首先测试这把锁：如果锁的值为0，则进程将其置为1并进入临界区。若锁已经为1，则进程等待直到其变成0。
- 实现：
  - 加锁原语LOCK(W)：L: if W=1 then goto L else W=1;
  - 解锁原语UNLOCK(W)：W=0;

```
P1:
.....
LOCK(W);
临界区;
UNLOCK(W);
.....
```

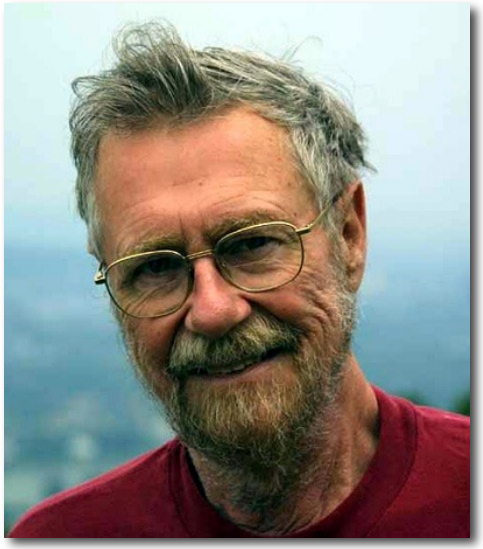
```
P2:
.....
LOCK(W);
临界区;
UNLOCK(W);
.....
```

现象：进程会一直处于忙等待状态(Busy waiting)。

六、进程间的相互作用

信号量(Semaphore)和P(down)、V(up)操作

- 1965年由荷兰学者Dijkstra提出，目前被广泛应用于单处理机和多处理机系统以及计算机网络中。



艾兹格·W·迪科斯彻(Edsger Wybe Dijkstra)
1 提出“goto有害论”；
2 提出信号量和PV原语;
3 解决了“哲学家聚餐”问题;
4 最短路径算法(SPF)和银行家算法的创造者;
5 第一个Algol 60编译器的设计者和实现者;
6 THE操作系统的设计者和开发者;
1972年图灵奖得主。

六、进程间的相互作用

❖ 信号量

特点：

- 表示资源的实体——是一个与队列有关的整型变量。
- 其值只能通过初始化操作和P、V操作来访问。

信号量的类型：

•公用信号量：用于进程间的互斥，初值通常为1；

•私有信号量：用于进程间的同步，初值通常为0或n。

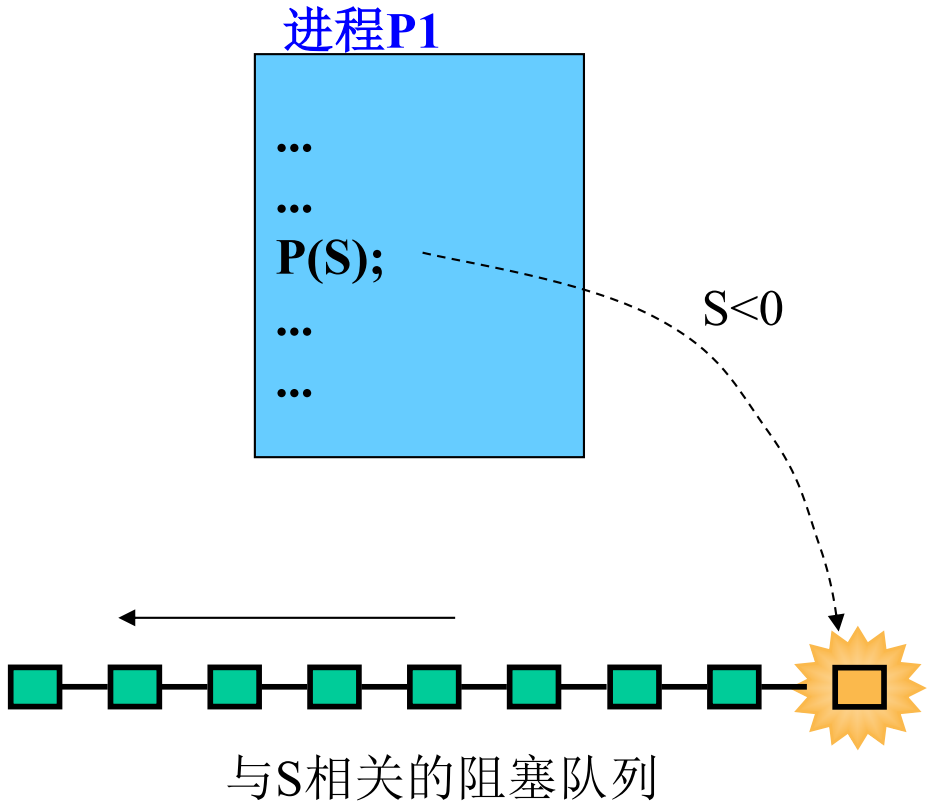
六、进程间的相互作用

❖ P操作(或down操作)

- 荷兰语“proberen”——“尝试”之意。意味着请求分配一个单位资源 或 进行测试。
- 原语操作，表示如下：

P(S)： //S为信号量

```
{
    S = S - 1;
    if (S < 0)
    {
       调用进程被阻塞，
       进入S的等待队列；
    }
}
```

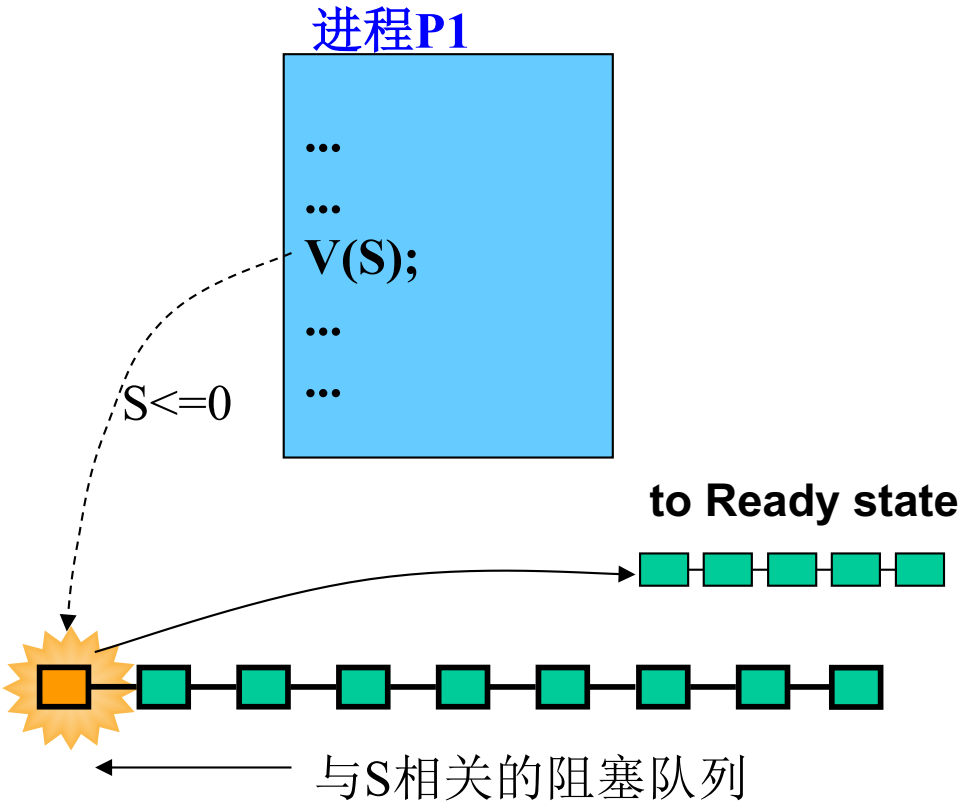


六、进程间的相互作用

- V操作(或up操作)
  - V操作：荷兰语“verhogen”——“增量/升高”之意，意味着释放/增加一个单位资源。
  - 原语操作，表示如下：

V(S)： //S为信号量

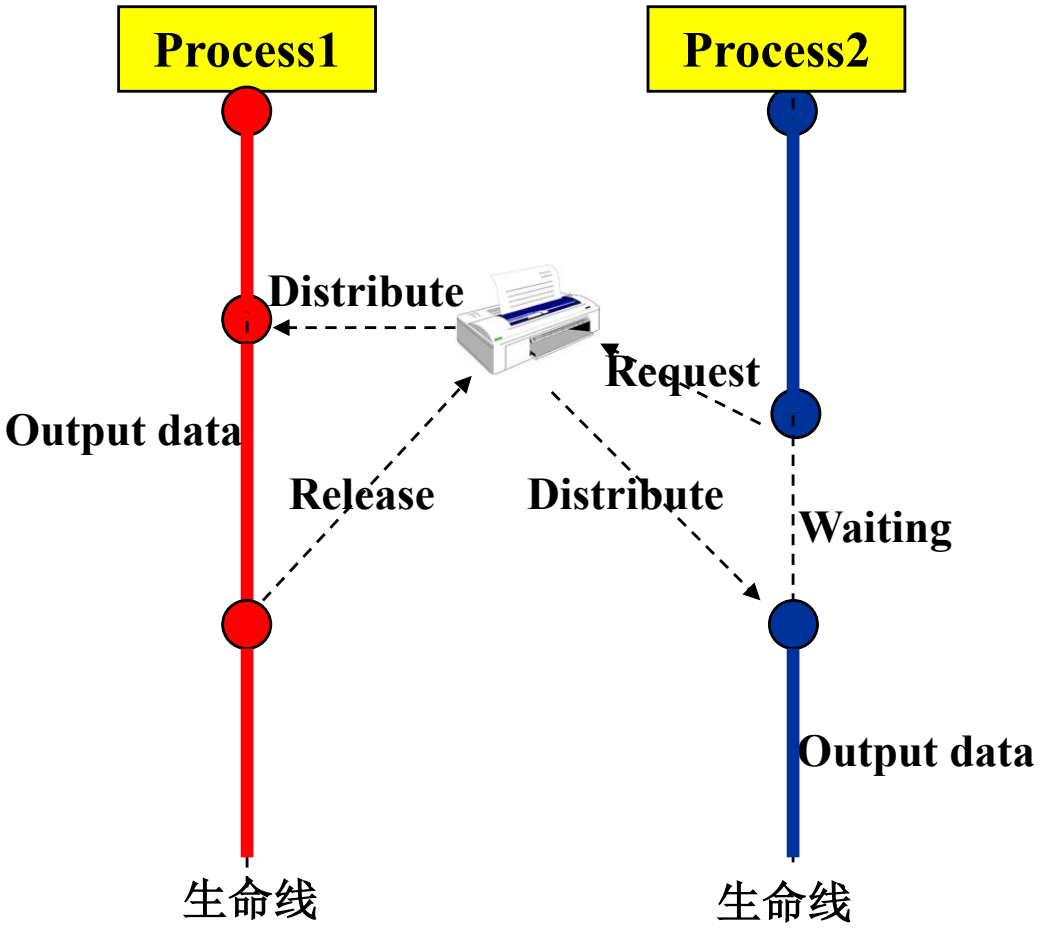
```
{
    S = S + 1;
    if (S <= 0)
    {
        从S的等待队列中唤醒一个进程
        使其进入就绪状态；
    }
}
```



问题：执行V操作的进程在“唤醒”其他进程时，自己是否会阻塞？

六、进程间的相互作用

使用P、V操作实现进程互斥



设置信号量：

Semaphore: S; //公用信号量

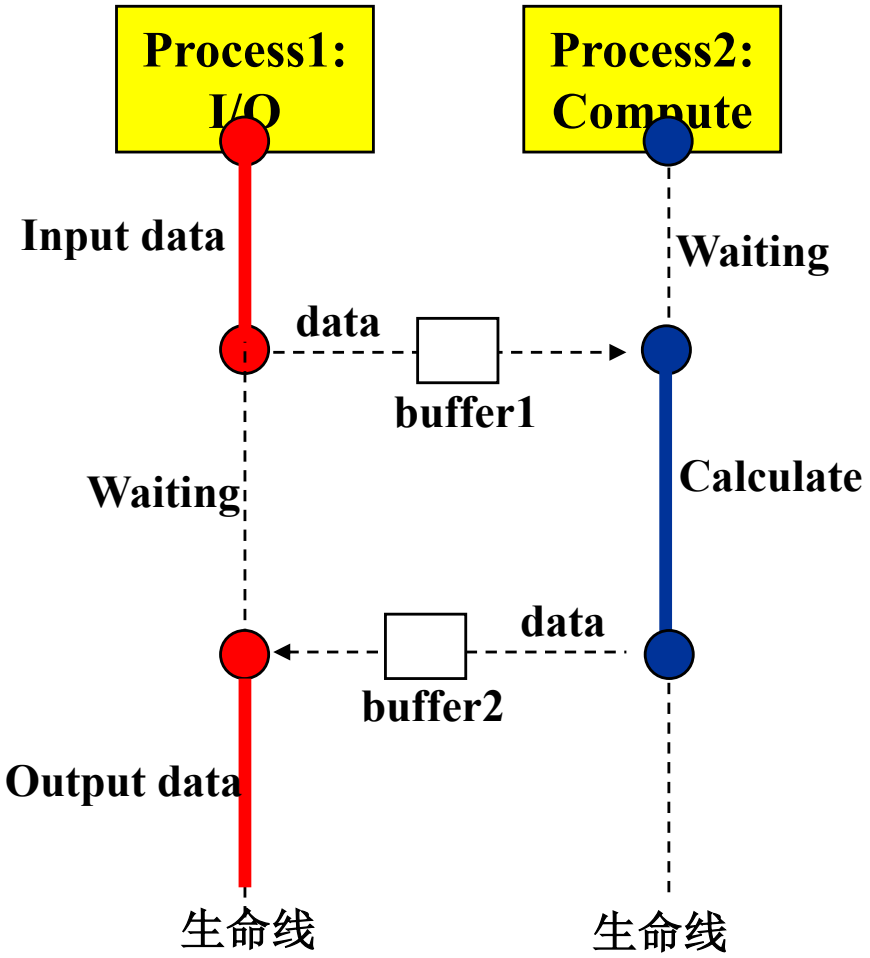
S=1;

```
Process1
{
    ....
    P(S);
    output data;
    V(S);
    ....
}
```

```
Process2
{
    ....
    P(S);
    output data;
    V(S);
    ....
}
```

六、进程间的相互作用

使用P、V操作实现进程同步



设置信号量：

Semaphore: full1, full2; //私有信号量

full1=0; //表示buffer1是否有数据, 初值0没数据

full2=0; //表示buffer2是否有数据, 初值0没数据

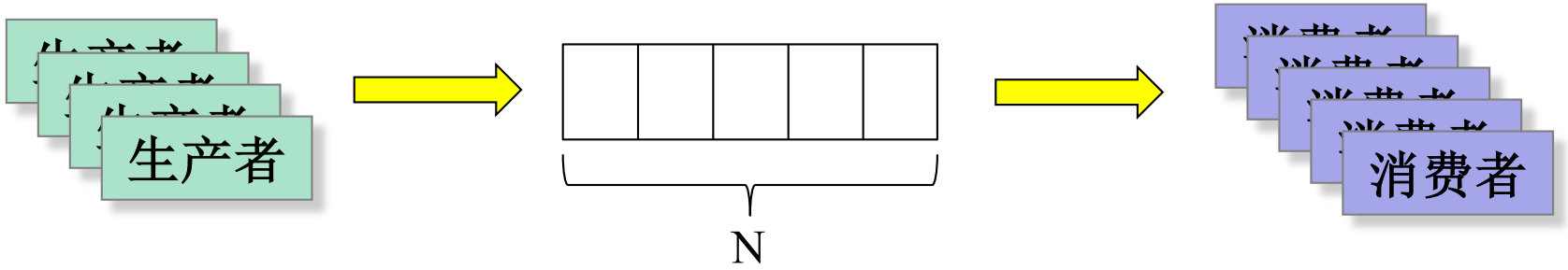
```
Process1
{
    Input data;
    put data to buffer1;
    V(full1);
    P(full2);
    get data from buffer2;
    output data;
}
```

```
Process2
{
    P(full1);
    get data from buffer1;
    calculate;
    put data to buffer2;
    V(full2);
}
```

六、进程间的相互作用

经典问题——生产者消费者问题(有限缓冲问题)

- 描述：生产者和消费者共享*n*个缓冲区，生产者生产产品放入缓冲区，消费者从缓冲区中取产品消费。请写出能够正确反映它们逻辑关系的代码。

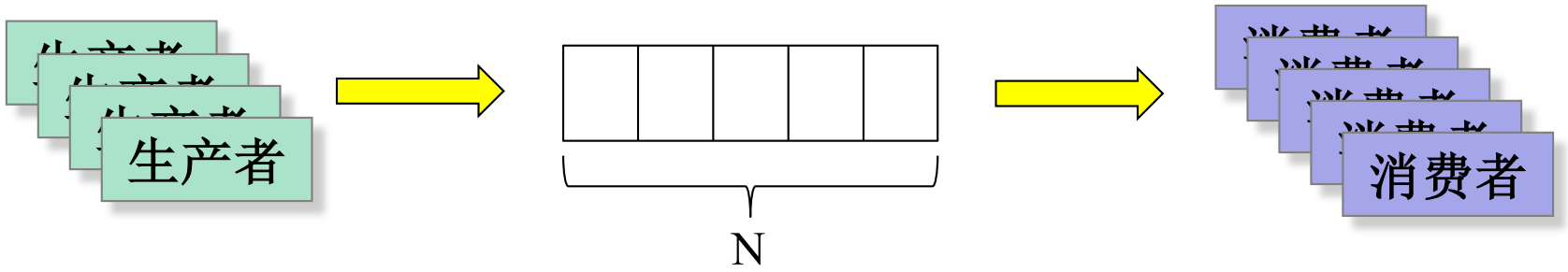


隐含条件：

1.消费者和生产者数量不固定;

2.消费者和生产者不能同时使用缓冲区。

六、进程间的相互作用



行为分析：

生产者：生产产品，放置产品（有空缓冲）

消费者：取出产品（有产品），消费产品

行为关系：

生产者之间：互斥（放置产品）

消费者之间：互斥（取出产品）

生产者消费者之间：互斥（放/取产品）

同步（放置-取出）

信号量设置：

semaphore mutex=1　//互斥

semaphore empty=N　//缓冲区空闲数, 同步

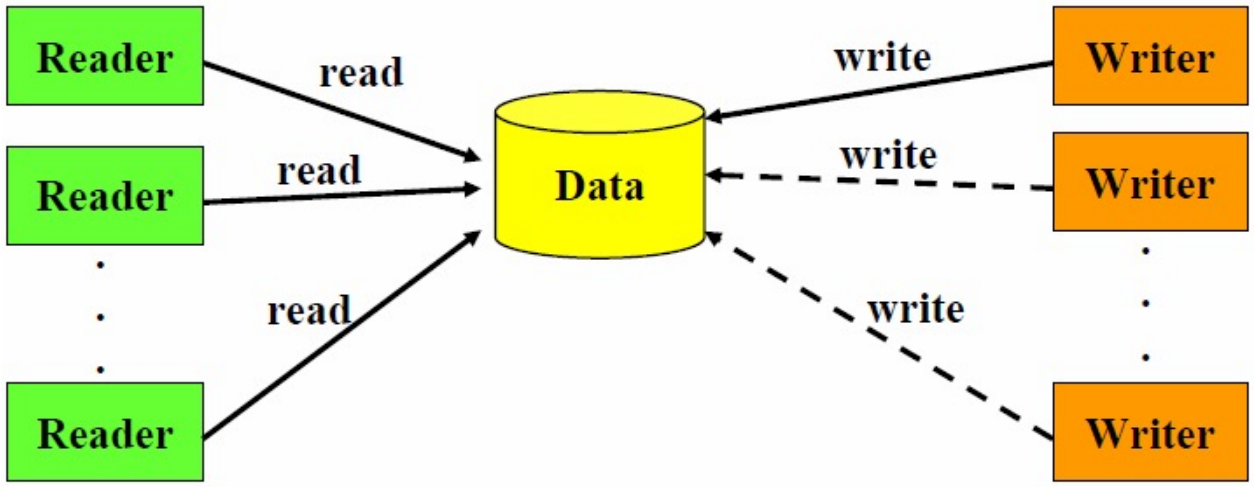
semaphore full=0　//产品数量，同步

```
生产者：
while(1) {
    produce;
    P(empty);
    P(mutex);
    insert to buffer;
    V(mutex);
    V(full);
}
```

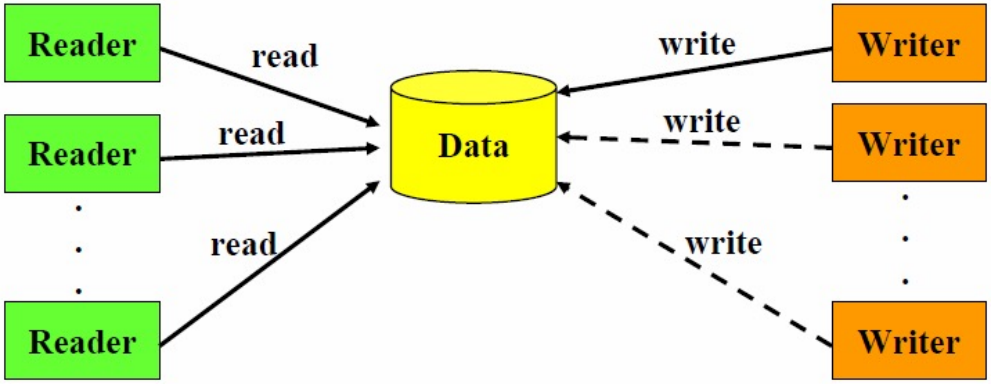
```
消费者：
while(1) {
    P(full);
    P(mutex);
    get from buffer;
    V(mutex);
    V(empty);
    consume;
}
```

经典问题——读者写者问题

- 问题描述：一个数据对象（文件、记录）可以为多个并发进程共享。其中有的进程只需要读其中的内容，我们称为“读者”；有的进程负责更新(读写)其中内容，我们称为“写者”。规定：“读者”可以同时读取共享数据对象；“写者”不能和其它任何进程同时访问共享数据对象。



六、进程间的相互作用



- 分析：
  - 进程行为：
    - 读进程的行为：
      - 访问共享数据
    - 行为特征
      - 多个读进程可同时访问共享数据。
    - 行为区别
      - 可以分为三类：第一个进入的读进程(占有资源)、最后一个离开的读进程(释放资源)和其它读进程。
    - 如何区分？
      - 设置一个计数器readnum来记录读进程的数目。
    - 写进程的行为：
      - 排他性的使用资源。

六、进程间的相互作用

❖ 分析：

➢ 确定同步和互斥关系：

① “读者-读者”：

互斥访问readnum

① “读者-写者”

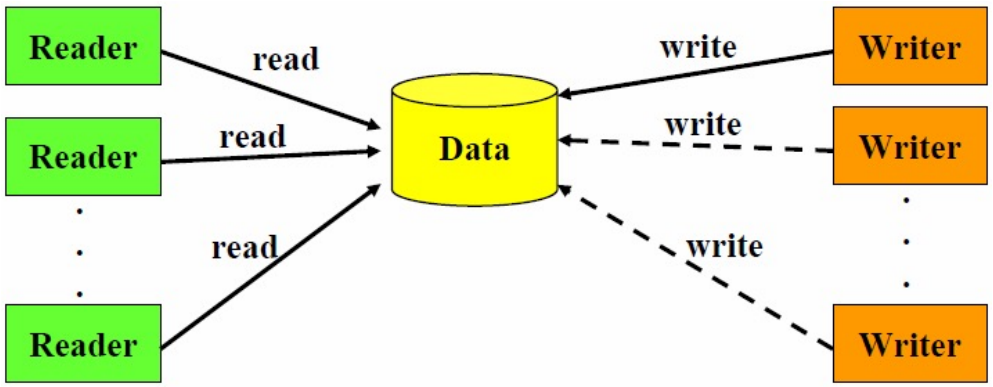
互斥访问Data

② “写者-写者”：

互斥访问Data

➢ 确定临界资源：

Data; readnum



六、进程间的相互作用

设置信号量：

int readnum=0; //计数，用于记录读者的数目

semaphore mutex=1; //公用信号量，用于readnum互斥

semaphore write=1; //公用信号量，用于Data访问的互斥

读者进程代码：

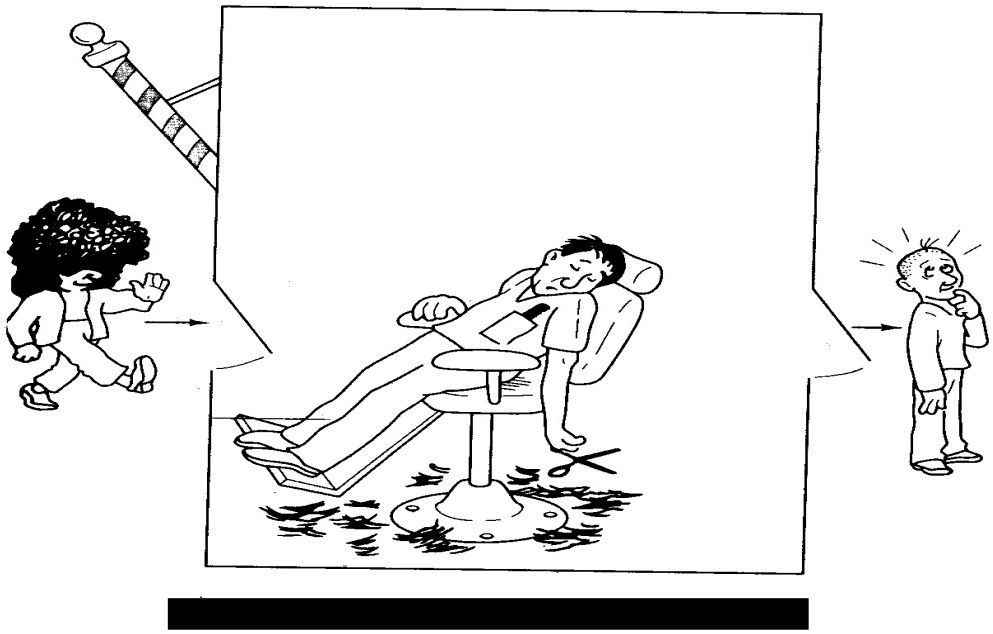
```
P(mutex); //对readnum互斥
readnum ++; //读者数目加1
if (readnum==1)   //第一个读进程
    P(write); //申请使用data资源
V(mutex); //释放readnum
reading;
P(mutex); //对readnum互斥
readnum --;
if (readnum==0)  //最后一个读进程
    V(write); //释放data资源
V(mutex); //释放readnum
```

写者进程代码：

```
P(write); //申请使用data资源
writing;
V(write); //释放data资源
```

六、进程间的相互作用

经典问题——理发师问题



问题描述（简化的）：理发店有一位理发师和一把理发椅。如果没有顾客，则理发师在理发椅上睡觉；当有顾客到达时，如理发师在睡觉则唤醒他理发，如果理发师正忙着理发，则顾客等待。编写程序实现理发师和顾客行为的正确描述。

六、进程间的相互作用

- 分析
  - 理发师行为：

    睡觉或理发。没有顾客睡觉，有顾客理发。
  - 顾客行为：

    理发或等待。到达后如果理发师睡觉，则唤醒理发师理发；如果理发师在理发，则等待。
- 相互作用
  - 理发师和顾客之间：同步
  - 顾客和顾客之间：无

六、进程间的相互作用

设置信号量：

Semaphore customers=0;　　//customers表示等候理发的顾客数量

Semaphore barbers=0;　　//barbars表示可用理发师数量

理发师进程代码

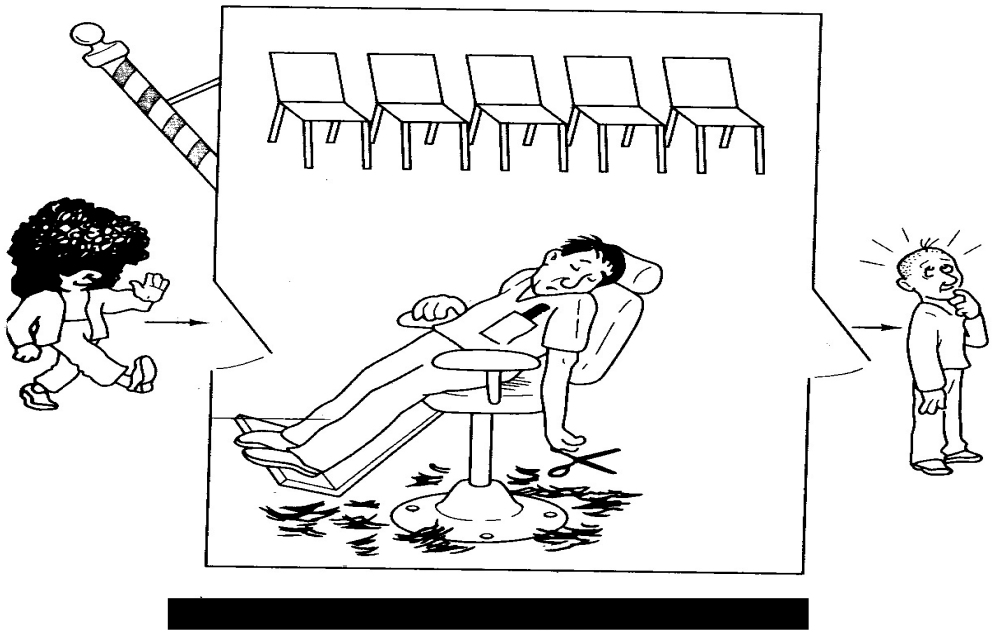
```
{
  While(1)
  {
     P(customers);//检查是否有顾客
     V(barbers);
     Cut hair();
  }
}
```

顾客进程代码

```
{
     V(customers)
     P(barbers); //检测是否有理发师
     Get_haircut();
}
```

六、进程间的相互作用

经典问题——理发师问题



修改条件：理发店有*n*把椅子，顾客到达时如果理发师空闲则理发，如果理发师忙，则看椅子上是否还有空位置，有空位置等待，没有空位置就离开。请写出相应的进程。

六、进程间的相互作用

设置信号量：

```
Semaphore customers=0;        //customers表示等候理发的顾客数量
Semaphore barbers=0;          //barbars表示理发师数量
int waiting=0;                //等待人数
Semaphore mutex=1;            //用于waiting的互斥
```

理发师进程

```
{
  While(1)
  {
     P(customers);//检查是否有顾客
      P(mutex);
        waiting=waiting-1;
      V(mutex);
     V(barbers);
     Cut hair();
  }
}
```

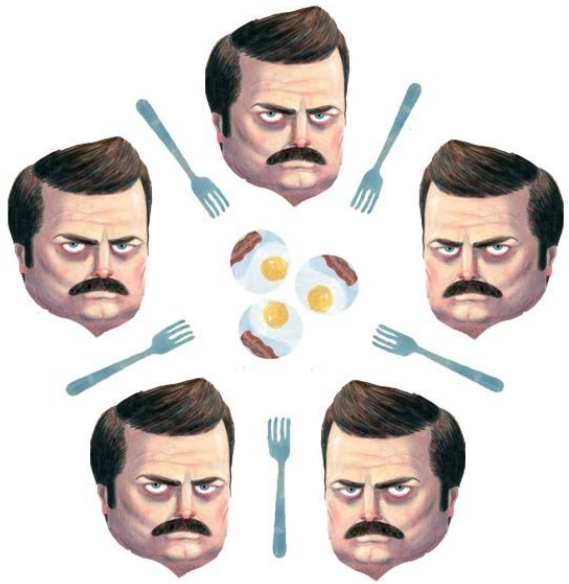
顾客进程

```
{
  P(mutex);
  if (waiting<n) then
  {
     waiting=waiting+1;
     V(mutex);
     V(customers);
     P(barbers); //检测是否有理发师
     Get_haircut();
  }
  else
  {
      V(mutex);
  }
}
```

六、进程间的相互作用

经典问题——哲学家问题

- 五个哲学家围坐在一圆桌旁，桌中央有一盘通心粉，每人面前有一只空盘子，每两人之间放一只叉子。每个哲学家的行为是思考，感到饥饿，然后吃通心粉。为了吃通心粉，每个哲学家必须拿到两只叉子，并且每个人只能直接从自己的左边或右边去取叉子。请编程实现哲学家的上述行为。



课后自行思考和完成！

六、进程间的相互作用

使用P、V操作时的注意事项

- P、V操作（对同一信号量）总是成对出现的；互斥操作时他们处于同一进程中；同步操作时他们处于不同进程中。
- 信号量初始值的设置和P、V操作的位置及次序是关键，要十分小心的设置，一定要保持正确的逻辑关系和较高的执行效率。

信号量设置：

semaphore mutex=1　//互斥

semaphore empty=n　//缓冲区空闲数

semaphore full=0　//产品数量

```
生产者：
while(1) {
    produce;
    P(empty);
    P(mutex);
    add to buffer;
    V(mutex);
    V(full);
}
```

```
消费者：
while(1) {
    P(full);
    P(mutex);
    get from buffer;
    V(mutex);
    V(empty);
    consume;
}
```

六、进程间的相互作用

信号量机制的缺点

- 对信号量的操作（P、V操作）被分散于各个进程当中，会使得程序：

易读性差

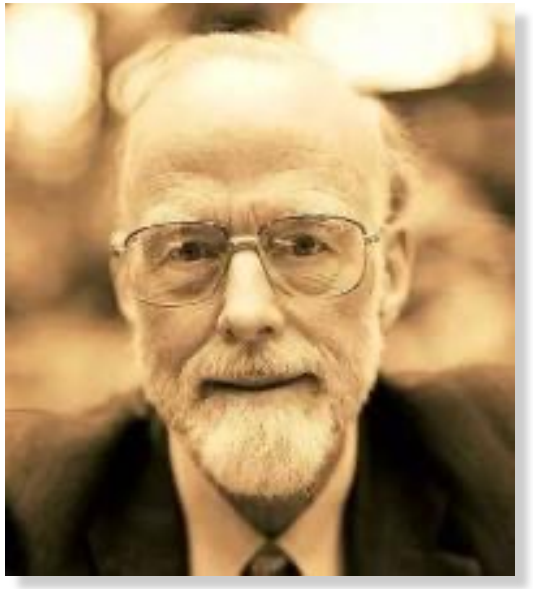
不利于修改和维护

正确性难以保证

- 要了解对于一组信号量的操作是否正确，必须通读整个系统或并发程序；
- 因为程序的局部性差，所以任一组变量或一段代码的修改都可能影响全局；
- 并发程序通常具有一定的规模，要保证一个这样的系统没有逻辑错误是很难的。

六、进程间的相互作用

- Hore和Hansen在20世纪70年代提出一个概念——管程
  - 将对信号量的组织管理工作交给系统去做——提高易读性、可维护性、和正确性——减轻程序员的负担。



C. A. R. Hoare
别名： Tony Hoare
发明： 1. Quicksort算法 2. Hoare Logic 3. Communicating Sequential Processes //并发进程理论
荣誉： • 1980年图灵奖得主 • 在2000年，由于其在计算机科学和教育方面的杰出贡献被英国皇家授予爵士爵位。
Monitors: An Operating System Structuring Concept. Commun. ACM, 17 (10): 549-557 (1974)

六、进程间的相互作用

- 管程的定义：
  - 关于共享资源的一组数据结构和在这组数据结构上的一组相关操作。
- 管程的思想——集中式同步机制
  - 将共享变量以及对共享变量能够进行的所有操作集中在一个模块中，以过程调用的形式提供给进程使用。

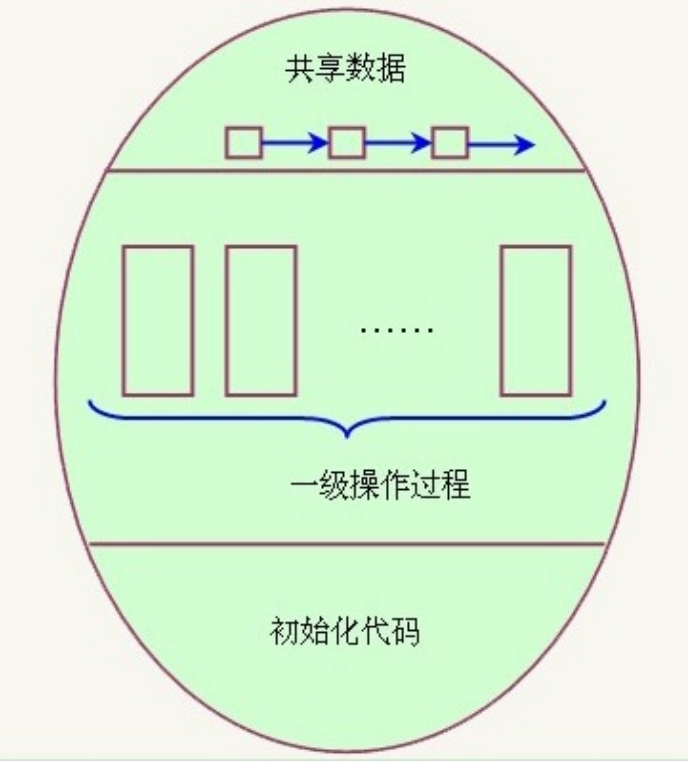
六、进程间的相互作用

管程的特征

- 模块化：管程是一个基本的软件模块，可以单独进行编译。
- 抽象数据类型：管程中封装了数据及对于数据的操作。
- 信息隐藏：管程外的进程或其他软件模块只能通过管程对外的接口来访问管程提供的操作，管程内部的实现细节对外界是透明的。
- 使用的互斥性：任何一个时刻，管程只能有一个活跃进程。进入管程时的互斥由编译器负责完成。

管程的结构

```
TYPE monitor_name=MONITOR
{
  局部变量说明;
  条件变量说明;
  初始化语句;
  define 管程内定义的在管程外可调用的过程或函数名表;
  use 管程外定义的在管程内将调用的过程或函数名表;
  PROCEDURE 过程名(形参表) {
        <过程/函数体>;
  }
        …
  PROCEDURE 过程名(形参表) {
        <过程/函数体>;
  }
}
```

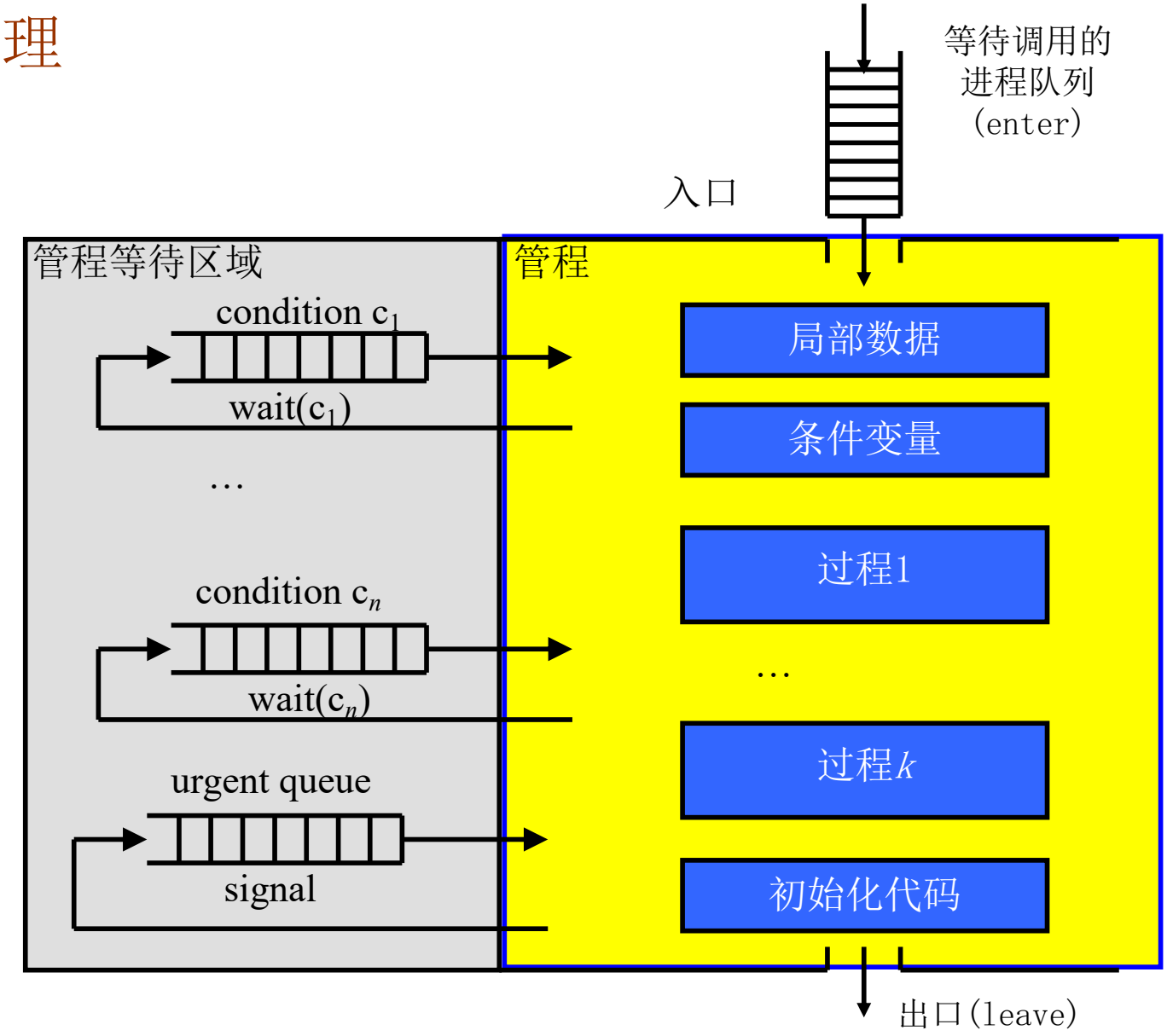


管程的工作原理

条件变量(c)：是管程内的一种数据结构，只在管程中才能被访问，通过两个原语wait,signal操作来控制它。

wait(c)：在c上阻塞调用进程，直到另一个进程在该条件变量上执行signal()。

signal(c)：在c上执行signal操作，如果存在其他进程由于c而被阻塞，唤醒之。



六、进程间的相互作用

使用管程解决生产者消费者问题

```
type ProducerConsumer = monitor  {
    condition full, empty;  //full缓冲区
                            满, empty缓冲区空
    integer count;    //count记录共有几件物品
    define insert, remove;
    use wait, signal;
    procedure insert( ) {
          if(count=N)  wait(full);
          insert to buffer;
          count=count+1;
          signal(empty);
    }
    procedure remove( ) {
          if(count=0) wait(empty);
          get from buffer;
          count= count-1;
          signal(full);
    }
    count=0;
}
```

```
Producer(生产者进程)  {
    while (true)  {
          produce;
          ProducerConsumer.insert();
    }
}
```

```
Consumer(消费者进程)  {
        while (true)  {
        ProducerConsumer.remove();
        consume;
        }
}
```

六、进程间的相互作用

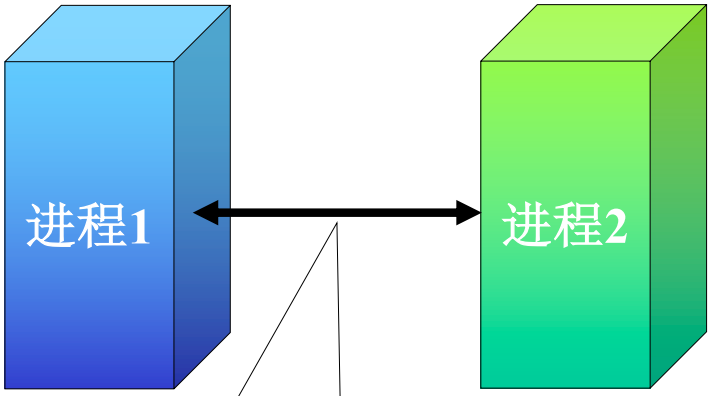
评价

- 管程可以使得进程对临界资源的互斥由（编译）系统保证，同时集中存放的方法便于管理，从而减轻了程序员的负担，在一定程度上避免错误的发生。
- 但是，管程不能保证程序执行其他逻辑的正确性，也就是说管程内函数的逻辑正确性仍然需要程序员保证，系统对此无能为力。
- 故而大量进程/线程之间的逻辑正确性保证时至今日仍是一个困扰程序人员的难题。

七、进程通信

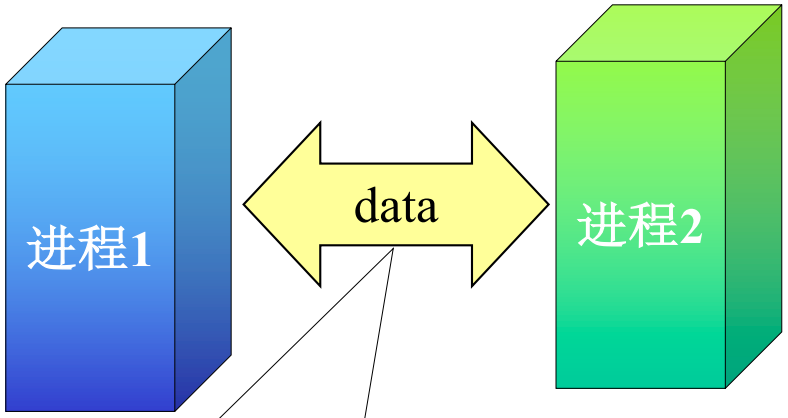
- 进程通信(IPC)：简单来说就是在进程间传输数据/交换信息。
- 根据交换信息量的多少和效率的高低，分为：

低级通信



只能传递状态和整数值（控制信息），一次传输传送信息量小。

高级通信



能够传递大量数据，通信效率高。

系统提供的高级通信方式：

（1）共享内存模式；

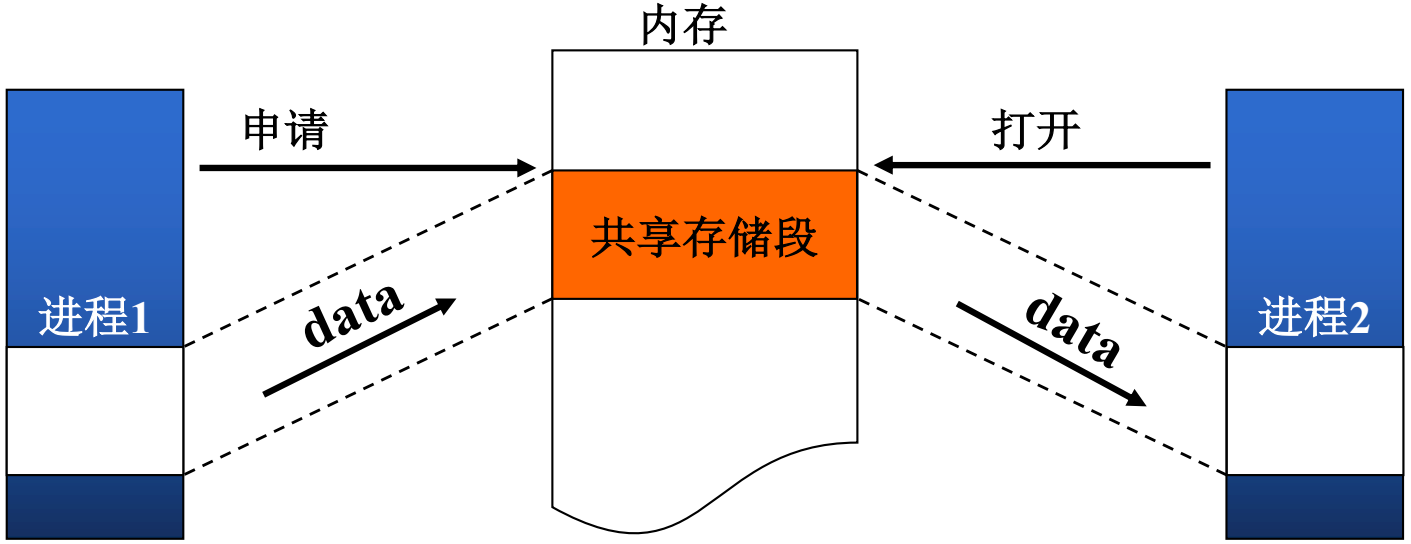
（2）消息传递模式；

（3）共享文件模式。

七、进程通信

共享内存(share memory)

- 使用一段共享的内存区域交换数据。
- 最为快捷有效的方式之一。
- 原理：



优点：

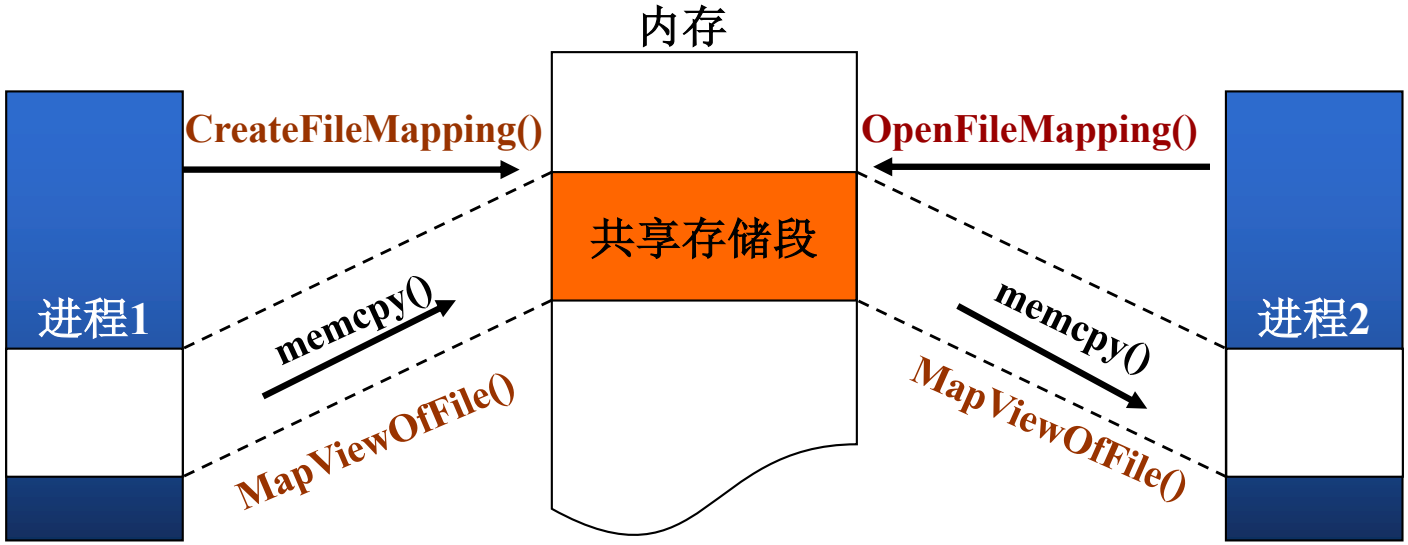
- 能够传输大量复杂数据

局限：

- 系统本身不提供对内存共享存储区的互斥控制，因此互斥要通过其它机制实现；
- 数据的发送方不关心数据由谁接收，数据的接收方也不关心数据是由谁发送的，存在安全隐患。容易造成病毒的传播。

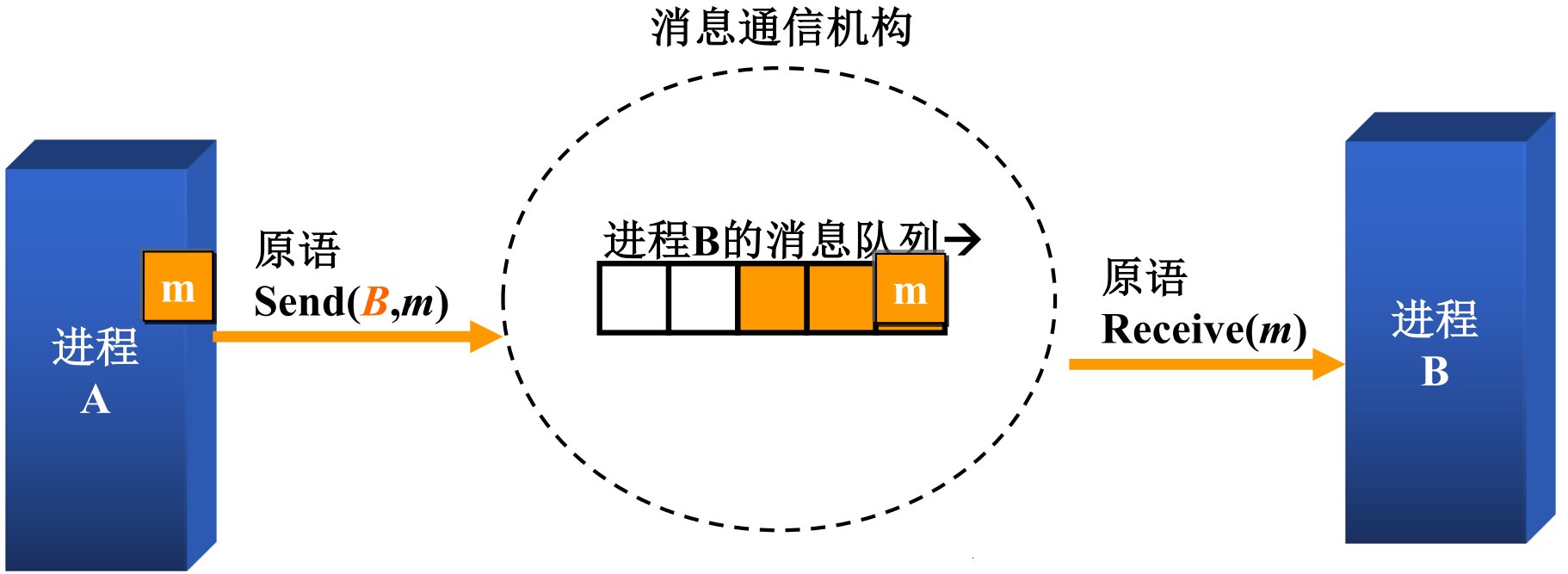
Windows中共享内存模式的实现方式(API函数)

- CreateFileMapping()：函数创建一个内存映射文件对象
- MapViewOfFile()：将文件映射对象的视图映射进地址空间
- memcpy()：数据复制到共享内存
- OpenFileMapping()：打开一个命名的共享文件对象



消息传递(message passing)

- 消息的概念：由发送方形成，通过一定的机制传递给接收方的一组信息，它的长度可以固定，也可以变化。
- 原理：

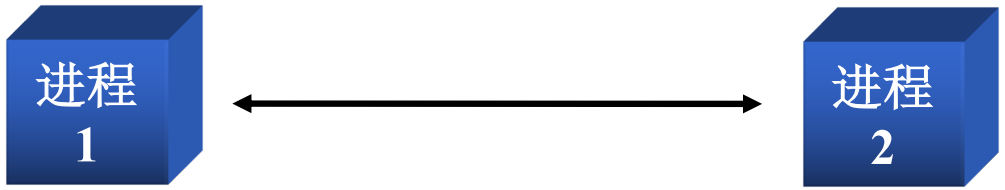


❖ 消息传递的方式

- 直接通信方式：点到点的发送。

  Send (*DestProcessName*, *Message*);

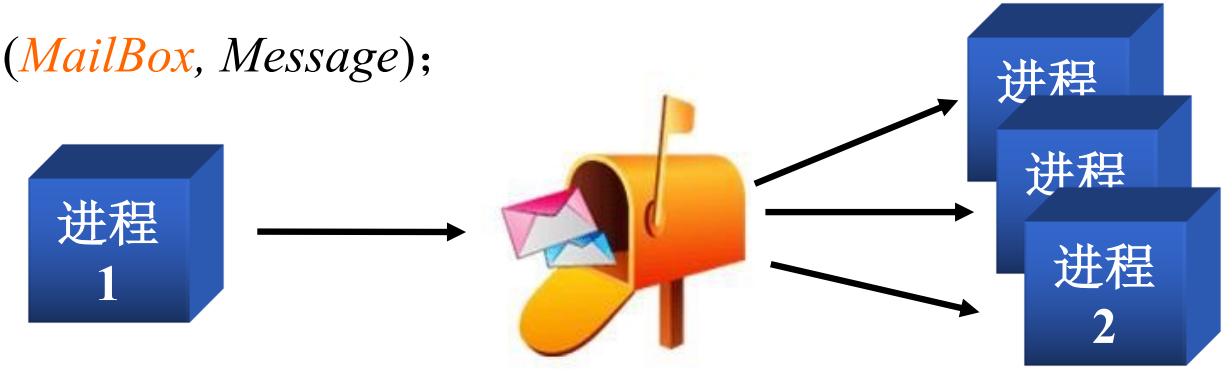
  Receive (*SourceProcessName*, *Message*);



- 间接通信方式：以信箱为媒介进行传递，可以广播。

  Send (*MailBox; Message*);

  Receive (*MailBox, Message*);

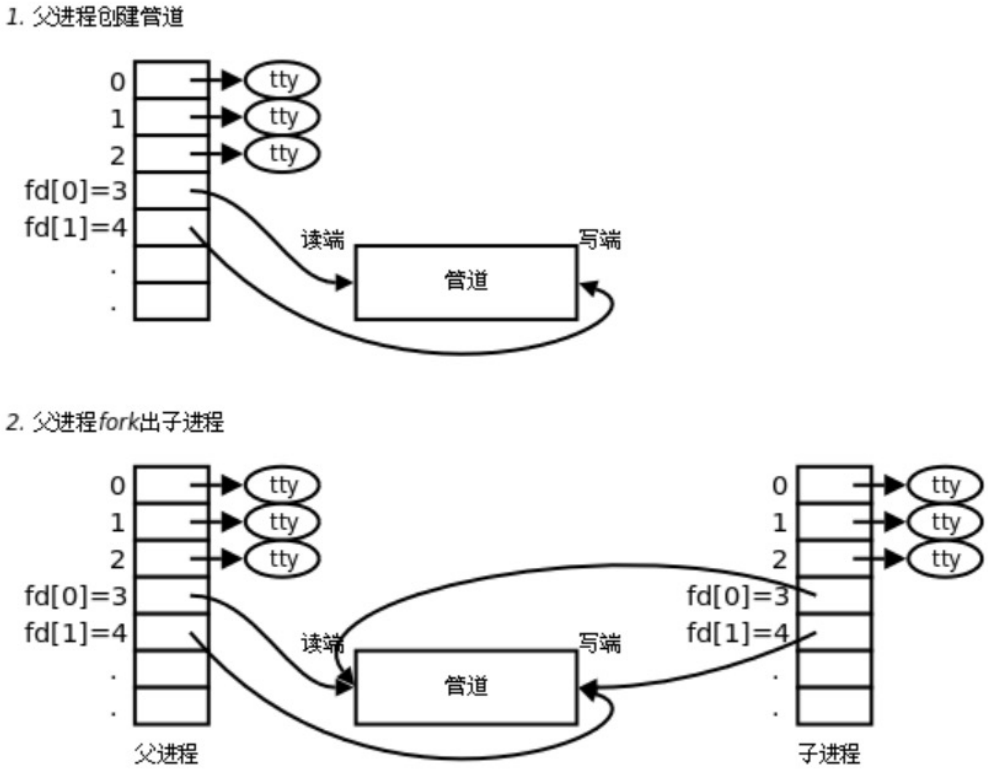


七、进程通信

共享文件模式：管道(pipe)

- 管道是一种信息流缓冲机构(线性字节数组)，使用文件读写方式进行访问。但管道并不是文件。
- 以先进先出(FIFO) 方式的单向传送数据。
- 匿名管道(Anonymous Pipes)
  - 通常用来在父进程和子进程之间传输数据。匿名管道总是本地的，不能在网络之间传递数据。
- 命名管道(Named Pipes)
  - 命名管道是一种有名称的，可在同一台计算机的不同进程之间或跨越一个网络的计算机的进程之间传输数据。

❖ 原理及实现(匿名管道)



UNIX管道的实现：

•创建管道：

int pipe(int fd[2]);

//fd[0]为管道里的读端

//fd[1]则为管道的写端

•写管道：

int write (int handle,char *buf,unsigned len)

•读管道

int read (int handle,void *buf,unsigned len)